

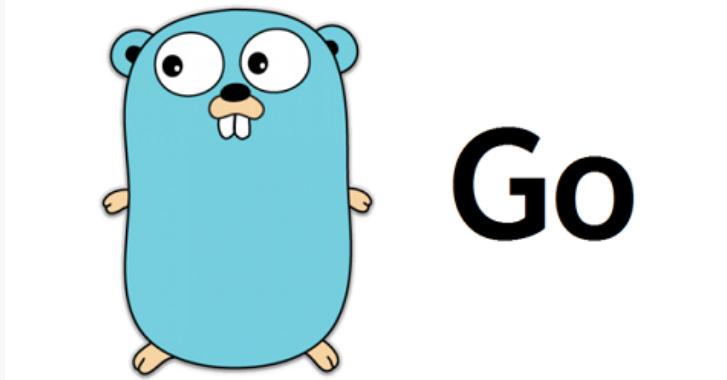
概述

Casbin 是一个强大和高效的开放源码访问控制库，它支持各种 [访问控制模型](#) 以强制全面执行授权。

执行一套规则与列出下述问题一样简单。对象和所需的 在 [策略](#) 文件中允许的动作(或根据您的需要提供任何其他格式)。这是Casbin使用的所有流的同义词。开发者/管理员有 完全控制布局，通过 [模型](#) 文件设置的执行和授权条件。Casbin提供了一个[执行者](#) 根据提供给执行者的策略和模型文件验证传入的请求。

Casbin 支持以下编程语言：

Casbin 为各种编程语言提供支持，随时准备将 纳入任何项目和工作流：

	 Go
Casbin	jCasbin
可用于生产环境	可用于生产环境



PyCasbin	Casbin.NET
可用于生产环境	可用于生产环境

在不同语言中支持的特性

我们一直致力于让 Casbin 在不同的编程语言中拥有相同的特性。但是现实总是不完美的。

特性	Go	Java	Node.js	PHP	Python	C#	Delphi	Rust	C++	Lua	Dart	Elixir
具体实施	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
RBAC	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ABAC	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Scaling ABAC (eval())	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
适配器	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
管理接口	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
RBAC API	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Batch API	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✗

特性	Go	Java	Node.js	PHP	Python	C#	Delphi	Rust	C++	Lua	Dart	Elixir
Filtered Adapter	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✗
Watcher	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
Role Manager	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗
Multi-Threading	✓	✓	✓	✗	✓	✗	✗	✓	✗	✗	✗	✗
matcher 中的'in'语法	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✓

注-✓适用于观察者或角色管理器 只意味着在核心库中拥有接口。这并不表明 是否有可用的监督员或角色管理器实现.

Casbin 是什么?

Casbin 是一个授权库, 在我们希望特定用户访问特定的 对象 或实体的流程中可以使用 主题 访问类型, 例如 动作 可以是 读取, 写入, 删除 或开发者设置的任何其他动作。这是Casbin最广泛的使用, 它叫做"标准" 或经典 `{ subject, object, action }` 流程。

Casbin能够处理除标准流量以外的许多复杂的许可使用者。可以添加 角色 (RBAC), 属性 (ABAC) 等。

Casbin 可以:

1. 以经典的 `{ subject, object, action }` 形式或您定义的自定义形式实施策略。支持允许和拒绝授权。
2. 具有访问控制模型model和策略policy两个核心概念。
3. 支持RBAC中的多层角色继承, 不止主体可以有角色, 资源也可以具有角色。
4. 支持内置超级用户, 如 `root` 或 `administrator`。超级用户可以在没有明确权限的情况下做任何事情。
5. 支持规则匹配的多个内置运营商。例如, `keyMatch` 可以映射 资源密钥 `/fo/bar` to the pattern `/foo*`。

Casbin不可以做什么

1. 身份认证 authentication (即验证用户的用户名和密码), Casbin 只负责访问控制。应该有其他专门的组件负责身份认证, 然后由 Casbin 进行访问控制, 二者是相互配合的关系。
2. 管理用户列表或角色列表。

该项目更容易管理他们的用户、角色或密码列表。 用户通常有他们的密码，但是 Casbin 的设计思想并不是把它作为一个存储密码的容器。 而是存储RBAC方案中用户和角色之间的映射关系。

开始使用

安装

[Go](#) [Java](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [C++](#) [Rust](#)
[Delphi](#) [Lua](#)

```
go get github.com/casbin/casbin/v2
```

对于Maven:

```
<!-- https://mvnrepository.com/artifact/org.casbin/jcasbin -->
```

```
<dependency>
    <groupId>org.casbin</groupId>
    <artifactId>jcasbin</artifactId>
    <version>1.x.y</version>
</dependency>
```

```
# NPM
```

```
npm install casbin --save
```

```
# Yarn
```

```
yarn add casbin
```

在您项目的 `comper.json` 中 require 这个包。 下载软件包:

```
composer require casbin/casbin
```

```
pip install casbin

dotnet add package Casbin.NET

# 下载源
git clone https://github.com/casbin/cabin-cop.git

# 生成项目文件
cd casbin-cpp && mkdir build && cd build && cmake ...
-DCMAKE_BUILD_TYPE=Release

# 构建并安装 casbin
cmake --build . --config Release --target casbin install -j 10

cargo install cargo-edit
cargo add casbin

// 如果你使用 async-std 作为异步执行器
cargo add async-std

// 如果你使用 tokio 作为异步执行器
cargo add tokio // 确保你启用了 "macros" 特性
```

Casbins4D 以包的形式提供（目前为 Delphi 10.3 Rio），您可以在 IDE 中安装它。然而，没有可视化组件，意味着你可以在包外独立使用这些 unit。只需在你的项目中导入 unit 即可（如果你不介意它们的数量）。

```
luarocks install casbin
```

如果报出错误：您的用户没有写入/usr/local/lib/luarocks/rocks 的权限，您可能需要以 root 用户身份运行或使用本地树，加上 --local 参数。您可以将 --local 参数添加到您的命令后面，就像这样修改：

```
luarocks install casbin --local
```

新建一个Casbin enforcer

Casbin使用配置文件来设置访问控制模型

它有两个配置文件, `model.conf` 和 `policy.csv`。其中, `model.conf` 存储了我们的访问模型, 而 `policy.csv` 存储的是我们具体的用户权限配置。Casbin的使用非常精炼。基本上, 我们只需要一种主要的结构: `enforcer` 当构造这个结构的时候, `model.conf` 和 `policy.csv` 将会被加载。

用另一种说法就是, 当新建Casbin enforcer的时候 你必须提供一个 [Model](#) 和一个 [Adapter](#)。

Casbin拥有一个 a [FileAdapter](#), 想知道更多请查阅 “更多Adapter” 中的[Adapter](#)

- 使用Model文件和默认 [FileAdapter](#):

Go Java Node.js PHP Python .NET C++ Delphi

Rust Lua

```
import "github.com/casbin/casbin/v2"

e, err := casbin.NewEnforcer("path/to/model.conf", "path/to/
policy.csv")

import org.casbin.jcasbin.main.Enforcer;
```

```
import { newEnforcer } from 'casbin';

const e = await newEnforcer('path/to/model.conf', 'path/to/
policy.csv');

require_once './vendor/autoload.php';

use Casbin\Enforcer;

$e = new Enforcer("path/to/model.conf", "path/to/policy.csv");

import casbin

e = casbin.Enforcer("path/to/model.conf", "path/to/policy.csv")

using NetCasbin;

var e = new Enforcer("path/to/model.conf", "path/to/
policy.csv");

#include <iostream>
#include <casbin/casbin.h>

int main() {
    // 创建一个执行者
    casbin::Enforcer e("path/to/model.conf", "path/to/
policy.csv");

    // 你的代码 ...
}

var
casbin: ICasbin;
```

```

use casbin::prelude::*;

// 如果你使用 async_std 作为异步执行器
#[cfg(feature = "runtime-async-std")]
#[async_std::main]
async fn main() -> Result<()> {
    let mut e = Enforcer::new("path/to/model.conf", "path/to/
policy.csv").await?;
    Ok(())
}

// 如果你使用 tokio 作为异步执行器
#[cfg(feature = "runtime-tokio")]
#[tokio::main]
async fn main() -> Result<()> {
    let mut e = Enforcer::new("path/to/model.conf", "path/to/
policy.csv").await?;
    Ok(())
}

local Enforcer = require("casbin")
local e = Enforcer:new("path/to/model.conf", "path/to/
policy.csv") -- The Casbin Enforcer

```

- 与其他Adapter一起使用Model text

[Go](#) [Python](#)

```

import (
    "log"

    "github.com/casbin/casbin/v2"

```

```
import casbin
import casbin_sqlalchemy_adapter

# 将SQLAlchemy Casbin适配器与SQLite DB一起使用
adapter = casbin_sqlalchemy_adapter.Adapter('sqlite:///test.db')

# 创建配置模型策略
with open("rbac_example_model.conf", "w") as f:
    f.write("""
[request_definition]
r = sub, obj, act

[policy_definition]
p = sub, obj, act

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = r.sub == p.sub && r.obj == p.obj && r.act == p.act
""")

# 从适配器和配置策略创建执行器
e = casbin.Enforcer('rbac_example_model.conf', adapter)
```

检查权限

在访问发生之前，在代码中添加强制挂钩：

Go Java Node.js PHP Python .NET C++ Delphi

Rust Lua

```
sub := "alice" // 想要访问资源的用户。
obj := "data1" // 将被访问的资源。
act := "read" // 用户对资源执行的操作。

ok, err := e.Enforce(sub, obj, act)

if err != nil {
    // 处理err
}

if ok == true {
    // 允许alice读取data1
} else {
    // 拒绝请求，抛出异常
}

// 您可以使用BatchEnforce()来批量执行一些请求
// 这个方法返回布尔切片，此切片的索引对应于二维数组的行索引。
// 例如results[0] 是{"alice", "data1", "read"}的结果
results, err := e.BatchEnforce([][]interface{}{{"alice",
    "data1", "read"}, {"bob", "data2", "write"}, {"jack", "data3",
    "read"}})
```

```
String sub = "alice"; // 想要访问资源的用户
String obj = "data1"; // 将要被访问的资源
String act = "read"; // 用户对资源进行的操作

if (e.enforce(sub, obj, act) == true) {
    // 允许alice读取data1
} else {
```

```
const sub = 'alice'; // 想要访问资源的用户
const obj = 'data1'; // 将要被访问的资源
const act = 'read'; // 用户对资源进行的操作

if ((await e.enforce(sub, obj, act)) === true) {
    // 允许alice读取data1
} else {
    // 拒绝请求，抛出异常
}

$sub = "alice"; // 想要访问资源的用户
$obj = "data1"; // 将要被访问的资源
$act = "read"; // 用户对资源进行的操作

if ($e->enforce($sub, $obj, $act) === true) {
    // 允许alice读取data1
} else {
    // 拒绝请求，抛出异常
}

sub = "alice" # 想要访问资源的用户
obj = "data1" # 将要被访问的资源
act = "read" # 用户对资源进行的操作

if e.enforce(sub, obj, act):
    # 允许alice读取data1
    pass
else:
    # 拒绝请求，抛出异常
    pass

var sub = "alice"; # 想要访问资源的用户
var obj = "data1"; # 将要被访问的资源
var act = "read"; # 用户对资源进行的操作
```

```
casbin::Enforcer e("../assets/model.conf", "../assets/
policy.csv");

if (e.Enforce({"alice", "/alice_data/Hello", "GET})) {
    std::cout << "Enforce OK" << std::endl;
} else {
    std::cout << "Enforce NOT Good" << std::endl;
}

if (e.Enforce({"alice", "/alice_data/Hello", "POST})) {
    std::cout << "Enforce OK" << std::endl;
} else {
    std::cout << "Enforce NOT Good" << std::endl;
}

if casbin.enforce(['alice,data1,read']) then
    // Alice很高兴它能够读取data1了
else
    // Alice很伤心

let sub = "alice"; // 想要访问资源的用户
let obj = "data1"; // 将会被访问的资源
let act = "read"; // 用户对资源的操作

if e.enforce((sub, obj, act)).await? {
    // 允许alice读取data1
} else {
    // 发生错误
}

if e:enforce("alice", "data1", "read") then
    -- 允许alice读取data1
else
    -- 拒绝请求，抛出异常
```

Casbin还提供了在运行时进行权限管理的API。例如，你可以获得分配给一个用户的所有角色，如下所示：

Go Java Node.js PHP Python .NET Delphi Rust

Lua

```
roles, err := e.GetRolesForUser("alice")

Roles roles = e.getRolesForUser("alice");

const roles = await e.getRolesForUser('alice');

$roles = $e->getRolesForUser("alice");

roles = e.get_roles_for_user("alice")

var roles = e.GetRolesForUser("alice");

roles = e.rolesForEntity("alice")

let roles = e.get_roles_for_user("alice");

local roles = e:GetRolesForUser("alice")
```

更多使用方法见 [Management API](#)和 [RBAC API](#)。

请查看测试用例以获取更多使用方式。

工作原理

在 Casbin 中, 访问控制模型被抽象为基于 **PERM (Policy, Effect, Request, Matcher)** 的一个配置文件。因此, 切换或升级项目的授权机制与修改配置一样简单。您可以通过组合可用的模型来定制您自己的访问控制模型。例如, 您可以在一个model中结合RBAC角色和ABAC属性, 并共享一组策略规则。

PERM模式由四个基础 (策略、效果、请求、匹配) 组成, 描述了资源与用户之间的关系。

请求

定义请求参数。基本请求是一个元组对象, 至少需要主题(访问实体)、目标(访问资源)和动作(访问方式)

例如, 一个请求可能长这样: `r={sub, obj, act}`

它实际上定义了我们应该提供访问控制匹配功能的参数名称和顺序。

策略

定义访问策略的模式。事实上, 它在策略规则文件中界定了字段的名称和顺序。

例如: `p={sub, obj, act}` 或 `p={sub, obj, act, eft}`

注: 如果未定义`eft` (policy result), 则策略文件中的结果字段将不会被读取, 和匹配的策略结果将默认被允许。

匹配器

匹配请求和政策的规则。

例如: `m = r.sub == p.sub && r.act == p.act && r.obj == p.obj` 这个简单和常见的匹配规则意味着如果请求的参数(访问实体, 访问资源和访问方式)匹配, 如果可以在策略中找到资源和方法, 那么策略结果 (`p.eft`) 便会返回。策略的结果将保存在 `p.eft` 中。

效果

它可以被理解为一种模型, 在这种模型中, 对匹配结果再次作出逻辑组合判断。

例如: `e = some (where (p.eft == allow))`

这句话意味着, 如果匹配的策略结果有一些是允许的, 那么最终结果为真。

让我们看看另一个示例: `e = some (where (p.eft == allow)) && !some(where (p.eft == deny))` 此示例组合的逻辑含义是: 如果有符合允许结果的策略且没有符合拒绝结果的策略, 结果是为真。换言之, 当匹配策略均为允许 (没有任何否认) 是为真 (更简单的是, 既允许又同时否认, 拒绝就具有优先地位)。

Casbin最基本和最简单的模式是ACL。 ACL的模式CONF是:

```
# Request definition
[request_definition]
r = sub, obj, act

# Policy definition
[policy_definition]
p = sub, obj, act
```

ACL模型的一个示例策略类似:

```
p, alice, data1, read  
p, bob, data2, write
```

它意味着:

- alice可以读取data1
- bob可以编写data2

我们还支持多行模式，通过在结尾处追加“\”:

```
# 匹配器  
[matchers]  
m = r.sub == p.sub && r.obj == p.obj \  
&& r.act == p.act
```

此外，对于 ABAC，您可以在 Casbin golang 版本中尝试下面的 [in](#) (jCasbin 和 Node-Casbin 尚不支持) 操作:

```
# Matchers  
[matchers]  
m = r.obj == p.obj && r.act == p.act || r.obj in ('data2',  
'data3')
```

但是你应确保数组的长度大于 1，否则的话将会导致 panic。

对于更多操作，你可以查看[govaluate](#)。

使用指南

阅读前, 请注意某些教程适用于Casbin的模型, 适用于所有Casbin的不同语言的实现。其他的一些教程是关于特定语言的。

我们的论文

- PML: An Interpreter-Based Access Control Policy Language for Web Services
(PML: 基于解释器的Web服务访问控制策略语言)

这篇论文深入剖析了Casbin的设计细节。如果您在论文中使用Casbin/PML作为参考, 请引用以下BibTex:

```
@article{luo2019pml,  
    title={PML: An Interpreter-Based Access Control Policy  
Language for Web Services},  
    author={Luo, Yang and Shen, Qingni and Wu, Zhonghai},  
    journal={arXiv preprint arXiv:1903.09756},  
    year={2019}  
}
```

- 一种基于元模型的访问控制策略描述语言

这里是另外一种更长版本的论文, 发表在《软件学报》上 不同格式的引文 (Refworks, EndNote 等) 可在以下网址找到: [\(另一个版本\) 基于元模型的访问控制政策规格语言\(中文\)](#)

视频

- 一个安全保险库 - 实现与 Casbin 的中间件的授权 - JuniorDevSG
- 基于Casbin的微型服务架构分享用户权限(俄文)
- Nest.js - Casbin RESTful RBAC授权中间件
- Gin 教程 第10章：30分钟内学习 Casbin 基础模型
- Gin 教程第11章：编码, API 和Casbin中的自定义功能
- Gin+Casbin权限实战速学(中文)
- jCasbin 基础：一个简单的RBAC示例(中文)
- 基于Casbin的Golang RBAC模型(中文)
- 学习Gin + Casbin(1)：通路& 概述(中文)
- ThinkPHP 5.1 + Casbin：导言(中文)
- ThinkPHP 5.1 + Casbin：RBAC授权 (中文)
- ThinkPHP 5.1 + Casbin: RESTfull & 中间件(中文)
- PHP-Casbin 快速上手(中文)
- ThinkPHP 5.1 + Casbin:如何使用自定义匹配函数(中文)
- Webman实战教程：如何使用casbin权限控制 (中文)

PERM元模型(策略、效果、请求、匹配器)

- 使用不同的访问控制模型配置来了解Casbin
- 利用Casbin的PERM模型进行访问控制
- 使用 Casbin 设计一个灵活的权限系统
- 授权访问控制列表
- 使用PERM和Casbin的访问控制(波斯语)
- RBAC? ABAC? .. PERM! New Way of Authorization for Cloud-Based Web Services and Apps (in Russian) (基于云的Web服务和应用程序授权的新方式 (俄语))

- 练习 & 使用 Casbin & PERM 的灵活授权实例 (俄语)
- Casbin权限管理 (中文)
- Casbin分析 (中文)
- 系统权限设计 (中文)
- Casbin: 一个权限引擎(中文)
- 使用 Casbin 实现ABAC (中文)
- Casbin 源代码分析(中文)
- Casbin 的权限评估(中文)
- Casbin: Go今日的库(中文)

Go Java Node.js PHP .NET Rust Lua

HTTP & RESTful

- 在Go中使用 Casbin 实现基础的基于角色的 HTTP 授权 (或 中文翻译)

监视器

- RBAC 通过Casbin Watcher分发同步(中文)

Beego

- 使用 Casbin 与 Beego: 1. 开始测试(中文)
- 使用 Casbin 与 Beego: 2. 策略储存(中文)
- 使用 Casbin 与 Beego: 3. 策略查询 (中文)
- 使用 Casbin 与 Beego: 4. 更新策略(中文)
- 使用 Casbin 与 Beego: 5. 更新策略(续)(中文)

Gin

- 使用 Casbin 的 Golang 项目授权
- 教程: 将 Gin 与 Casbin 集成

- 带Pipeline的K8s上的策略执行
- 使用JWT和Casbin在Gin应用程序中进行身份验证和授权
- 后端API与Go: 1. 基于JWT的身份验证(中文)
- 后端API与Go: 2. 基于Casbin的授权(中文)
- 在Gin和GORM使用Go的授权库Casbin(日语)

Echo

- Casbin网络授权

Iris

- Iris + Casbin权限控制实战
- 基于Casbin的HTTP基于角色的访问控制(中文)
- 从头学习iris + Casbin

VMware Harbor

- Casbin: Golang访问控制框架(中文)
- Harbor访问控制(中文)

Argo CD

- Argo CD中使用Casbin建立RBAC权限体系

GShark

- GShark: 轻松有效地扫描Github中的敏感信息

SpringBoot

- jCasbin: 更轻量级的权限管理解决方案(中文)
- JCasbin与JFinal的集成(中文)

Express

- 如何将基于角色访问控制添加到您的AWS上的服务器

Koa

- [Casbin and Koa授权 Part 1](#)
- [Casbin and Koa授权 Part 2](#)

Nest

- 如何使用 Casbin 和 Nest.js 创建基于角色的认证中间件
- nest.js: Casbin RESTful RBAC授权插件（视频）
- 基于 Casbin 的 Node.js 基于属性的访问控制演示应用程序
- 多租户SaaS 启动工具包，带有cqrs graphql microservice 架构

Fastify

- 使用 Fastify 和 Casbin 在 Node.js 中的访问控制
- Casbin, 您的项目中强大和高效的 ACL

Laravel

- Laravel授权：支持ACL、RBAC、ABAC和其他模型的授权库
- 在 .Net 中使用 Casbin 授权
- 在 Rust 中用 Casbin 实现基础的基于角色的 HTTP 授权
- 如何在您的rust web应用程序中使用casbin 授权 [Part - 1]
- 如何在您的rust web应用程序中使用casbin 授权 [Part - 2]

APISIX

- 使用 Casbin 在 APISX 中授权



>

访问控制模型

访问控制模型

支持的模型

支持的 Casbin 模型

Model 的语法

Model 的语法

Effector

Effector是用于Casbin effector的API。

函数

你可以使用内置函数，或者指定你自己的函数。

RBAC

Casbin RBAC的使用

RBAC with Pattern

RBAC with Pattern

域内基于角色的访问控制

域内基于角色的访问控制

Casbin RBAC和RBAC96

Casbin RBAC和RBAC96之间的差异

ABAC

基于 Casbin 的 ABAC

优先级模型

优先级模型

超级管理员

超级管理员是整个系统的管理员。我们可以在RBAC, ABAC以及带域的RBAC等模型中使用它

支持的模型

1. **ACL (Access Control List, 访问控制列表)**
2. 具有 **超级用户** 的 ACL
3. 没有用户的 ACL: 对于没有身份验证或用户登录的系统尤其有用。
4. 没有资源的 ACL: 某些场景可能只针对资源的类型, 而不是单个资源, 诸如 `write-article`, `read-log` 等权限。它不控制对特定文章或日志的访问。
5. **RBAC (基于角色的访问控制)**
6. 支持资源角色的 RBAC: 用户和资源可以同时具有角色 (或组)。
7. 支持域/租户的 RBAC: 用户可以为不同的域/租户设置不同的角色集。
8. **ABAC (基于属性的访问控制):** 支持利用 `resource.Owner` 这种语法糖获取元素的属性。
9. **RESTful:** 支持路径, 如 `/res/*`, `/res/: id` 和 HTTP 方法, 如 `GET`, `POST`, `PUT`, `DELETE`。
10. **拒绝优先:** 支持允许和拒绝授权, 拒绝优先于允许。
11. **优先级:** 策略规则按照先后次序确定优先级, 类似于防火墙规则。

例子

访问控制模型	Model 文件	Policy 文件
ACL	<code>basic_model.conf</code>	<code>basic_policy.csv</code>
具有超级用户的 ACL	<code>basic_with_root_model.conf</code>	<code>basic_policy.csv</code>
没有用户的	<code>basic_without_users_model.conf</code>	<code>basic_without_users_policy.csv</code>

访问控制模型	Model 文件	Policy 文件
ACL		
没有资源的 ACL	basic_without_resources_model.conf	basic_without_resources_policy.csv
RBAC	rbac_model.conf	rbac_policy.csv
具有资源角色的RBAC	rbac_with_resource_roles_model.conf	rbac_with_resource_roles_policy.csv
带有域/租户的 RBAC	rbac_with_domains_model.conf	rbac_with_domains_policy.csv
ABAC	abac_model.conf	无
RESTful	keymatch_model.conf	keymatch_policy.csv
拒绝改写	rbac_with_not_deny_model.conf	rbac_with_deny_policy.csv
同意与拒绝	rbac_with_deny_model.conf	rbac_with_deny_policy.csv
优先级	priority_model.conf	priority_policy.csv
明确优先级	priority_model_explicit	priority_policy_explicit.csv
主体优先	subject_priority_model.conf	subject_priority_policyl.csv

访问控制模型	Model 文件	Policy 文件
先级		

Model 的语法

- Model CONF 至少应包含四个部分: [request_definition], [policy_definition], [policy_effect], [matchers]。
- 如果 model 使用 RBAC, 还需要添加 [role_definition] 部分。
- 模型CONF可以包含注释。 注释开头是 #, # 将注释该行的其余部分。

Request定义

[request_definition] 是访问请求的定义。 它定义了 e.Enforce(...) 函数中的参数。

```
[request_definition]
r = sub, obj, act
```

sub, obj, act 表示经典三元组: 访问实体 (Subject), 访问资源 (Object) 和访问方法 (Action)。但是, 你可以自定义你自己的请求表单, 如果不需要指定特定资源, 则可以这样定义 sub、act , 或者如果有两个访问实体, 则为 sub、sub2、obj、act。

Policy定义

[policy_definition] 是策略的定义。 它界定了该策略的含义。 例如, 我们有以下模式:

```
[policy_definition]
p = sub, obj, act
p2 = sub, act
```

这些是我们对policy规则的具体描述

```
p, alice, data1, read
p2, bob, write-all-objects
```

policy部分的每一行称之为一个策略规则，每条策略规则通常以形如p, p2的policy type开头。如果存在多个policy定义，那么我们会根据前文提到的policy type与具体的某条定义匹配。上面的policy的绑定关系将会在matcher中使用，罗列如下：

```
(alice, data1, read) -> (p.sub, p.obj, p.act)
(bob, write-all-objects) -> (p2.sub, p2.act)
```

💡 提示

策略规则中的元素总被视为字符串。如果您对此有任何疑问，请在<https://github.com/casbin/casbin/issues/113>上进行讨论

Policy effect定义

[policy_effect] 部分是对policy生效范围的定义，原语定义了当多个policy rule同时匹配访问请求request时，该如何对多个决策结果进行集成以实现统一决策。以下示例展示了一个只有一条规则生效，其余都被拒绝的情况：

```
[policy_effect]
```

该Effect原语表示如果存在任意一个决策结果为 `allow` 的匹配规则，则最终决策结果为 `allow`，即 `allow-override`。其中 `p.eft` 表示策略规则的决策结果，可以为 `allow` 或者 `deny`，当不指定规则的决策结果时，取默认值 `allow`。通常情况下，`policy` 的 `p.eft` 默认为 `allow`，因此前面例子中都使用了这个默认值。

这是另一个 `policy effect` 的例子：

```
[policy_effect]  
e = !some(where (p.eft == deny))
```

该Effect原语表示不存在任何决策结果为 `deny` 的匹配规则，则最终决策结果为 `allow`，即 `deny-override`。`some` 量词判断是否存在一条策略规则满足匹配器。`any` 量词则判断是否所有的策略规则都满足匹配器（此处未使用）。`policy effect` 还可以利用逻辑运算符进行连接：

```
[policy_effect]  
e = some(where (p.eft == allow)) && !some(where (p.eft == deny))
```

该Effect原语表示当至少存在一个决策结果为 `allow` 的匹配规则，且不存在决策结果为 `deny` 的匹配规则时，则最终决策结果为 `allow`。这时 `allow` 授权和 `deny` 授权同时存在，但是 `deny` 优先。

① 备注

虽然我们设计了上述政策效果的语法，但目前的执行只是使用硬编码的政策效果。我们认为这种灵活性没有多大必要。目前为止你必须使用内置的 `policy effects`，不能自定义。

支持的 `policy effects` 如下：

Policy effect 定义	意义	示例
some(where (p.eft == allow))	allow-override	ACL, RBAC, etc.
!some(where (p.eft == deny))	deny-override	拒绝改写
some(where (p.eft == allow)) && !some(where (p.eft == deny))	allow-and-deny	同意与拒绝
priority(p.eft) deny	priority	优先级
subjectPriority(p.eft)	基于角色的优先级	主题优先级

匹配器

[matchers] 是策略匹配器的定义。匹配器是表达式。它确定了如何根据请求评估策略规则。

[matchers]

```
m = r.sub == p.sub && r.obj == p.obj && r.act == p.act
```

上面的这个匹配器是最简单的，它表示请求的三元组：主题、对象、行为都应该匹配策略规则中的表达式。

在匹配器中，你可以使用算术运算符如 +, -, *, /，也可以使用逻辑运算符如：&&, ||, !。

Orders of expressions in matchers

The order of expressions can greatly affect performance.

Look at the following example for details:

```
const rbac_models = `

[request_definition]
r = sub, obj, act


[policy_definition]
p = sub, obj, act


[role_definition]
g = _, _


[policy_effect]
e = some(where (p.eft == allow))`


[matchers]
m = g(r.sub, p.sub) && r.obj == p.obj && r.act == p.act`


func TestManyRoles(t *testing.T) {

    m, _ := model.NewModelFromString(rbac_models)
    e, _ := NewEnforcer(m, false)

    roles := []string{"admin", "manager", "developer", "tester"}

    // 2500 projects
    for nbPrj := 1; nbPrj < 2500; nbPrj++ {
        // 4 objects and 1 role per object (so 4 roles)
        for _, role := range roles {
            roleDB := fmt.Sprintf("%s_project:%d", role, nbPrj)
        }
    }
}
```

The enforce time may be very very long, up to 6 seconds

```
go test -run ^TestManyRoles$ github.com/casbin/casbin/v2 -v

==== RUN TestManyRoles
    rbac_api_test.go:598: RESPONSE abu
/projects/1      GET : true IN: 438.379µs
    rbac_api_test.go:598: RESPONSE abu      /projects/
2499      GET : true IN: 39.005173ms
    rbac_api_test.go:598: RESPONSE jasmine
/projects/1      GET : true IN: 1.774319ms
    rbac_api_test.go:598: RESPONSE jasmine      /projects/
2499      GET : true IN: 6.164071648s
    rbac_api_test.go:600: More than 100 milliseconds for
jasmine /projects/2499 GET : 6.164071648s
    rbac_api_test.go:598: RESPONSE jasmine      /projects/
2499      GET : true IN: 12.164122ms
--- FAIL: TestManyRoles (6.24s)
FAIL
FAIL      github.com/casbin/casbin/v2      6.244s
FAIL
```

However, if we can adjust the order of the expressions in matchers, and put more time-consuming expressions like functions behind, the execution time will be very short.

Changing the order of expressions in matchers in the above example to

```
[matchers]
m = r.obj == p.obj && g(r.sub, p.sub) && r.act == p.act
```

```
go test -run ^TestManyRoles$ github.com/casbin/casbin/v2 -v
==== RUN TestManyRoles
    rbac_api_test.go:599: RESPONSE abu
```

多节类型

如果您需要多个策略定义或多个匹配器，您可以使用 `p2`, `m2`。事实上，以上四节都可以使用多个类型，语法是 `r+number`。例如 `r2`, `e2`。默认情况下，这四个部分应当对应一个。如您的 `r2` 只能使用匹配器 `m2` 匹配策略 `p2`。

您可以通过 `EnforceContext` 作为 `的第一个参数执行` 方法来指定类型，`EnforceContext` 就像这个

[Go](#) [Node.js](#) [Java](#)

```
EnforceContext{"r2", "p2", "e2", "m2"}  
type EnforceContext struct {  
    RType string  
    PType string  
    EType string  
    MType string  
}  
  
const enforceContext = new EnforceContext('r2', 'p2', 'e2',  
'm2');  
class EnforceContext {  
    constructor(rType, pType, eType, mType) {  
        this.pType = pType;  
        this.eType = eType;  
        this.mType = mType;  
        this.rType = rType;  
    }  
}
```

```
EnforceContext enforceContext = new EnforceContext("2");
public class EnforceContext {
    private String pType;
    private String eType;
    private String mType;
    private String rType;
    public EnforceContext(String suffix) {
        this.pType = "p" + suffix;
        this.eType = "e" + suffix;
        this.mType = "m" + suffix;
        this.rType = "r" + suffix;
    }
}
```

示例用法, 请参阅 [model](#) 和 [policy](#), 请求如下所示:

[Go](#) [Node.js](#) [Java](#)

```
/ 在后缀将参数传入NewEnforceContext, 例如2或3, 它将创建 r2,p2,等。
enforceContext := NewEnforceContext("2")
// 您还可以单独指定特定类型
enforceContext.EType = "e"
// 不要在EnforceContext中传递, 默认值为r,p,e,m
e.Enforce("alice", "data2", "read")      // true
// 在EnforceContext中传递
e.Enforce(enforceContext, struct{ Age int }{Age: 70}, "/data1",
"read")      //false
e.Enforce(enforceContext, struct{ Age int }{Age: 30}, "/data1",
"read")      //true
```

/ 在后缀将参数传入NewEnforceContext, 例如2或3, 它将创建 r2,p2,等。

```
/ 在后缀将参数传入NewEnforceContext, 例如2或3, 它将创建 r2, p2, 等。
EnforceContext enforceContext = new EnforceContext("2");
// 您还可以单独指定特定类型
enforceContext.setType("e");
// 不要在EnforceContext中传递, 默认值为r、p、e、m
e.enforce("alice", "data2", "read"); // true
// 在EnforceContext中传递
// TestEvalRule位于https://github.com/casbin/jcasbin/blob/master/
// src/test/java/org/casbin/jcasbin/main/AbacAPIUnitTest.java#L56
e.enforce(enforceContext, new
AbacAPIUnitTest.TestEvalRule("alice", 70), "/data1", "read");
// false
e.enforce(enforceContext, new
AbacAPIUnitTest.TestEvalRule("alice", 30), "/data1", "read");
// true
```

特殊语法

您也可以使用 `in`, 这是唯一一个有文本名称的操作符。此操作符检查右侧数组以查看它是否包含等于左侧数值的值。相等是由使用 `==` 运算符决定的，这个库不检查值之间的类型。任何两个值在投射到 `interface{}` 时仍然可以使用 `==` 检查相等，它们是否按预期的那样起作用。请注意，你可以使用数组的参数，但它必须是 `[]interface{}`。

请参考 `rbac_model_matcher_using_in_op`, `keyget2_model` 以及 `keyget_model`

示例:

```
[request_definition]
r = sub, obj
...
[matchers]
m = r.sub.Name in (r.obj.Admins)
```

```
e.Enforce(Sub{Name: "alice"}, Obj{Name: "a book", Admins: []interface{}{"alice", "bob"}})
```

表达式评估器

Casbin的匹配器运算是由不同语言的表达式运算器实现的。 Casbin整合了他们的能力以提供统一的PERM语言。 除了这里提供的所有模型语法外，那些表达式运算器可能提供额外的功能，这些功能可能不会被另一种语言或实现所支持。 请自己承担使用的后果。

不同语言的Casbin实现使用的表达式运算器有：

实现	语言	Expression evaluator
Casbin	Golang	https://github.com/Knetic/govaluate
jCasbin	Java	https://github.com/killme2008/aviator
Node-Casbin	Node.js	https://github.com/donmccurdy/expression-eval
PHP-Casbin	PHP	https://github.com/symfony/expression-language
PyCasbin	Python	https://github.com/danthedeckie/simpleeval
Casbin.NET	C#	https://github.com/davideicardi/DynamicExpresso
Casbin4D	Delphi	https://github.com/casbin4d/Casbin4D/tree/master/SourceCode/Conce/Third%20Party/

实现	语言	Expression evaluator
		TExpressionParser
casbin-rs	Rust	https://github.com/jonathandturner/rhai
casbin-cpp	C++	https://github.com/ArashPartow/exprtk

① 备注

如果您遇到关于Casbin的性能问题，这可能是表达式评估器效率低下造成的。您可以直接发送issue到Casbin或表达式运算器的开发团队以获得提高效率的建议。详情请参阅 [基准](#) 部分。

Effector

Effect是一个policy rule的结果 [Effector](#) 是用于Casbin effector的API

MergeEffects()

MergeEffects将 enforcer 收集的所有匹配结果合并为一项决定。

例如:

[Go](#)

```
Effect, explainIndex, err = e.MergeEffects(expr, effects,  
matches, policyIndex, policyLength)
```

在本示例中:

- `Effect` 是此函数合并的最后决定(初始参数为 `indeterminate`)。
- `explainIndex` 是 `eft` 的索引, `eft` 的值可为 `Allow` 或者 `Deny`.(初始值是 `-1`)
- `err` 用于检查 `effect` 是否受到支持。
- `expr` 是被存储为 `string` 的 `policy_effects`
- `effects` 是 Effect 的数组, 其中值可以为 `Allow`, `Indeterminate` 或者 `Deny`
- `matches` 是显示结果是否符合策略的数组。
- `policyIndex` 是模型中的策略索引。
- `policyLength` 是策略的长度。

上面的代码说明了我们如何将参数传递到 `MergeEffects` 函数，并且该函数将根据 `expr` 处理效果和匹配。

要部署一个Effector，我们可以这样做：

Go

```
var e Effector
Effect, explainIndex, err = e.MergeEffects(expr, effects,
matches, policyIndex, policyLength)
```

`MergeEffects` 表明如果 `expr` 可以匹配结果，也就是说 `p_eft` 是 `allow`，我们就可以合并所有效果。如果没有符合拒绝的规则，我们就允许这样做。

ⓘ 备注

如果 `expr` 不能匹配 `"priority(p_eft) || deny"` 以及 `policyIndex` 短于 `policyLength-1`，它将 短路 中间的一些effect。

函数

Matchers中的函数

你甚至可以在Matcher中指定函数，使它更强大。你可以使用内置函数，或者指定你自己的函数。所有内置的函数均采用以下格式：

```
bool function_name(string url, string pattern)
```

它返回一个布尔值表示 `url` 是否匹配 `模式`。

支持的内置函数如下：

函数	url	模式	示例
keyMatch	一个URL 路径，例如 <code>/alice_data/resource1</code>	一个URL 路径或 <code>*</code> 模式下，例如 <code>/alice_data/*</code>	keymatch_model.conf/keymatch_policy.csv
keyMatch2	一个URL 路径，例如 <code>/alice_data/resource1</code>	一个URL 路径或 <code>:</code> 模式下，例如 <code>/alice_data/:resource</code>	keymatch2_model.conf/keymatch2_policy.csv
keyMatch3	一个URL 路径，例如 <code>/alice_data/resource1</code>	一个URL 路径或 <code>{}</code> 模式下，例如 <code>/alice_data/{resource}</code>	https://github.com/casbin/casbin/blob/277c1a2b85698272f764d71a94d2595a8d425915/util/builtin_operators_test.go#L171-L196
keyMatch4	一个URL 路径，例如 <code>/alice_data/resource1</code>	一个URL 路径或 <code>{}</code> 模式下，例如 <code>/alice_data//{id}/book/{id}</code>	https://github.com/casbin/casbin/blob/277c1a2b85698272f764d71a94d2595a8d425915/util/builtin_operators_test.go#L208-L222
regexMatch	任意字符串	正则表达式模式	keymatch_model.conf/keymatch_policy.csv
ipMatch	一个 IP 地址，例如 <code>192.168.2.123</code>	一个 IP 地址或一个 CIDR，例如 <code>192.168.2.0/24</code>	ipmatch_model.conf/ipmatch_policy.csv
globMatch	类似路径的 <code>/alice_data/resource1</code>	一个全局模式，例如 <code>/alice_data/*</code>	https://github.com/casbin/casbin/blob/277c1a2b85698272f764d71a94d2595a8d425915/util/builtin_operators_test.go#L426-L466

获取密钥函数通常需要三个参数(除了 `keyget`):

```
bool function_name(string url, string pattern, string key_name)
```

如果密钥 `key_name` 与模式匹配, 他们将返回它的值。如果没有匹配, 则返回 `""`。

例如, `KeyGet2("/resource1/action", "/:res/action", "res")` 将返回 `"resource1"`, `KeyGet3("/resource1_admin/action", "/{res}_admin/*", "res")` 将返回 `"resource1"` 至于 `KeyGet`, 需要两个参数, `KeyGet("/resource1/action", "/")` 将返回 `"resource1/action"`

函数	url	模式	密钥名称	示例
keyGet	一个URL 路径, 例如 <code>/alice_data/resource1</code>	一个URL 路径或 <code>*</code> 模式下, 例如 <code>/alice_data/*</code>	\	<code>keyget_model.conf/keymatch_policy.csv</code>
keyGet2	一个URL 路径, 例如 <code>/proj/resource1/</code>	一个URL 路径或 <code>:</code> 模式下, 例如 <code>/alice_data/:resource</code>	模式中指定的密钥名称	<code>keyget_model.conf/keymatch_policy.csv</code>
keyGet3	一个URL 路径, 例如 <code>/proj/res3_admin/</code>	一个URL 路径或 <code>{}</code> 模式下, 例如 <code>/proj/{resource}_admin/</code>	模式中指定的密钥名称	https://github.com/casbin/casbin/blob/7bd496f94f5a2739a392d333a9aaa10ae397673/util/builtin_operators_test.go#L209-L247

详情见: https://github.com/casbin/casbin/blob/master/util/builtin_operators_test.go

怎样增加自定义函数

首先准备您的函数。它接受一些参数, 然后返回一个布尔类型:

```
func KeyMatch(key1 string, key2 string) bool {
    i := strings.Index(key2, "*")
    if i == -1 {
        return key1 == key2
    }

    if len(key1) > i {
        return key1[:i] == key2[:i]
    }
    return key1 == key2[:i]
}
```

然后用 `interface{}` 类型的接口包装它:

```
func KeyMatchFunc(args ...interface{}) (interface{}, error) {
    name1 := args[0].(string)
    name2 := args[1].(string)

    return (bool)(KeyMatch(name1, name2)), nil
}
```

最后，在Casbin的执行者(enforcer)中注册这个函数:

```
e.AddFunction("my_func", KeyMatchFunc)
```

现在，您可以在您的模型CONF中像这样使用这个函数:

```
[matchers]
m = r.sub == p.sub && my_func(r.obj, p.obj) && r.act == p.act
```

RBAC

Role定义

[role_definition] 原语定义了RBAC中的角色继承关系。Casbin支持RBAC系统的多个实例，例如，用户可以有角色和继承关系，而资源也可以有角色和继承关系。这两个RBAC系统不会干扰。

此部分是可选的。如果在模型中不使用RBAC角色，则省略此部分。

```
[role_definition]
g = _, _
g2 = _, _
```

上述角色定义表明，`g` 是一个 RBAC 系统，`g2` 是另一个 RBAC 系统。`_, _` 表示角色继承关系的前项和后项，即前项继承后项角色的权限。一般来讲，如果您需要进行角色和用户的绑定，直接使用`g` 即可。当您需要表示角色（或者组）与用户和资源的绑定关系时，可以使用`g` 和 `g2` 这样的表现形式。请参见 [rbac_model](#) 和 [rbac_model_with_resource_roles](#) 的示例。

在Casbin里，我们以policy表示中实际的用户角色映射关系（或是资源-角色映射关系），例如：

```
p, data2_admin, data2, read
g, alice, data2_admin
```

这意味着 `alice` 是角色 `data2_admin` 的一个成员。`alice` 在这里可以是用户、资源或角色。Casbin 只是将其识别为一个字符串。

接下来在matcher中，应该像下面的例子一样检查角色信息：

```
[matchers]
m = g(r.sub, p.sub) && r.obj == p.obj && r.act == p.act
```

这意味着在请求中应该在policy中包含sub角色。

① 备注

1. Casbin 只存储用户角色的映射关系。
2. Cabin 没有验证用户是否是有效的用户，或者角色是一个有效的角色。这应该通过认证来解决。
3. RBAC 系统中的用户名和角色名称不应相同。因为Casbin将用户名和角色识别为字符串，所以当前语境下Casbin无法得出这个字面量到底指代用户alice 还是角色 alice。这时，使用明确的 role_alice，问题便可迎刃而解。
4. 假设 A 具有角色 B，B 具有角色 C，并且 A 有角色 C。这种传递性在当前版本会造成死循环。

① TOKEN名称协议

通常主题令牌名称在策略定义中是 sub 并置于开头。现在Golang Casbin 支持定制令牌名称 & 位置。如果主题令牌名称是 sub，主题令牌可以放置在任意位置，不需要额外的操作。如果域令牌名称不是 sub，应在启动执行者后调用 e.SetFieldIndex() 用于 constant.SubjectIndex 不论其地位如何。

```
# `subject` 这里指 sub
[policy_definition]
p = obj, act, subject
```

```
.SetFieldIndex("p", constant.SubjectIndex, 2) // 索引从0开始  
ok, err := e.DeleteUser("alice") // 没有 SetFieldIndex, 它会  
引起一个错误
```

角色层次

Casbin 的 RBAC 支持 RBAC1 的角色层次结构功能，如果 `alice` 具有 `role1`, `role1` 具有 `role2`，则 `alice` 也将拥有 `role2` 并继承其权限。

下面是一个称为层次结构级别的概念。因此，此示例的层次结构级别为2。对于Casbin中的内置角色管理器，可以指定最大层次结构级别。默认值为10。这意味着终端用户 `alice` 只能继承10个级别的角色。

```
// NewRoleManager is the constructor for creating an instance  
of the  
// default RoleManager implementation.  
func NewRoleManager(maxHierarchyLevel int) rbac.RoleManager {  
    rm := RoleManager{}  
    rm.allRoles = &sync.Map{}  
    rm.maxHierarchyLevel = maxHierarchyLevel  
    rm.hasPattern = false  
  
    return &rm  
}
```

如何区分用户和角色？

在RBAC中，Casbin不对用户和角色进行区分。它们都被视为字符串。如果你只使用单层的RBAC模型（角色不会成为另一个角色的成员）。可以使用 `e.GetAllSubjects()`

获取所有用户，`e.GetAllRoles()` 获取所有角色。它们会为规则 `g, u, r` 分别列出所有的 `u` 和 `r`。

但如果你在使用多层RBAC（带有角色继承），并且你的应用没有记录下一个名字（字符串）对应的是用户还是角色，或者你将用户和角色用相同的名字命名。那么你可以给角色加上像 `role::admin` 的前缀再传递到Casbin中。由此可以通过查看前缀来区分用户和角色。

如何查询隐性角色或权限？

当用户通过RBAC层次结构继承角色或权限，而不是直接在策略规则中分配它们时，我们将这种类型的分配称为 `implicit`。要查询这种隐式关系，需要使用以下两个api：
`GetImplicitRolesForUser()` 以及 `GetImplicitPermissionsForUser()` 替代
`GetRolesForUser()` 以及 `GetPermissionsForUser()`. 有关详情，请参阅 [this GitHub issue](#)。

在 RBAC 中使用模式匹配

详情请参阅 [RBAC with Pattern](#)

角色管理器

详情见 [角色管理器](#) 部分。

RBAC with Pattern

快速入门

- 在 `g(_, _)` 中使用模式

```
e, _ := NewEnforcer("./example.conf", "./example.csv")
e.AddNamedMatchingFunc("g", "KeyMatch2", util.KeyMatch2)
```

- 使用域名的模式

```
e.AddNamedDomainMatchingFunc("g", "KeyMatch2", util.KeyMatch2)
```

- 使用所有模式

仅使用两个APIs

As shown above, after you create the `enforcer` instance, you need to activate pattern matching via `AddNamedMatchingFunc` and `AddNamedDomainMatchingFunc` API, which determine how the pattern matches.

 备注

如果您使用在线编辑器，它会在左下角指定模式匹配函数。

```
g10smatche
12      *
  matchingDomainForGFunction:
  'keyMatch'
13      */
14      matchingForGFunction:
  'keyMatch2',
15
  matchingDomainForGFunction:
  'keyMatch2'
16  };
17 })();
```

Request

```
1 /book/1
2 /book/1
3
```

在 RBAC 中使用模式匹配

有时，您希望一些具有特定模式的subjects, object 或者 domains/tenants能够被自动授予角色。 RBAC中的模式匹配函数可以帮助做到这一点。 模式匹配函数与前一个函数共享相同的参数和返回值： [matcher function](#)。

模式匹配函数支持g的每一个参数

我们知道，在matcher里面RBAC通常被表示为 `g(r.sub, p.sub)` 接下来我们将使用如下策略：

```
p, alice, book_group, read
g, /book/1, book_group
g, /book/2, book_group
```

因此 `alice` 可以阅读所有书籍，包括 `book 1` 和 `book 2`。但是当有数千本书时，如果我们仅仅使用 `g` 策略规则将每本书一个一个地添加到书籍角色（或组），那将会是非常繁琐的。

不过，凭借着模式匹配函数，你可以把整个策略只用一行写下！

```
g, /book/:id, book_group
```

Casbin会自动将`/book/1`和`/book/2`匹配为模式`/book/:id`。您需要做的仅仅是向enforcer注册该方法，例如像这样：

[Go](#) [Node.js](#)

```
e.AddNamedMatchingFunc("g", "KeyMatch2", util.KeyMatch2)
```

```
await e.addNamedMatchingFunc('g', Util.keyMatch2Func);
```

当在domains/tenants里面使用模式匹配函数的时候，你需要把这个函数向enfoecer以及model进行注册

[Go](#) [Node.js](#)

```
e.AddNamedDomainMatchingFunc("g", "KeyMatch2", util.KeyMatch2)
```

```
await e.addNamedDomainMatchingFunc('g', Util.keyMatch2Func);
```

如果您不理解`g(r.sub, p.sub, r.dom)`意味着什么，请阅读[rbac-with-domains](#)。简而言之，`g(r.sub, p.sub, r.dom)`将检查用户`r.sub`在域内`r.dom`是否具有角色`p.sub`因此，这正是匹配器的工作方式。您可以在这里查看完整的示例。

除了上面的模式匹配语法外，我们还可以使用纯域模式。

例如，如果我们想要 `sub` 在不同的域中访问，`domain1` 和 `domain2`，我们可以使用纯域模式：

```
p, admin, domain1, data1, read
p, admin, domain1, data1, write
p, admin, domain2, data2, read
p, admin, domain2, data2, write

g, alice, admin, *
g, bob, admin, domain2
```

在这个示例中，我们希望 `alice` 阅读并将 `数据` 写入 `domain1` 和 `domain2`，模式匹配 `*` 在 `g` 中，让 `alic` 可以访问两个域。

通过使用模式匹配，尤其是在更加复杂和我们需要考虑的大量域或对象的情况下，我们可以执行 `policy_definition` 更加美雅和有效。

域内基于角色的访问控制

域租户的角色定义

在Casbin中的RBAC角色可以是全局或是基于特定于域的。特定域的角色意味着当用户处于不同的域/租户群体时，用户所表现的角色也不尽相同。这对于像云服务这样的大型系统非常有用，因为用户通常分属于不同的租户群体。

域/租户的角色定义应该类似于：

```
[role_definition]  
g = -, -, -
```

第三个 `-` 表示域/租户的名称，此部分不应更改。然后，政策可以是：

```
p, admin, tenant1, data1, read  
p, admin, tenant2, data2, read  
  
g, alice, admin, tenant1  
g, alice, user, tenant2
```

该实例表示 `tenant1` 的域内角色 `admin` 可以读取 `data1`，`alice` 在 `tenant1` 域中具有 `admin` 角色，但在 `tenant2` 域中具有 `user` 角色，所以 `alice` 可以有读取 `data1` 的权限。同理，因为 `alice` 不是 `tenant2` 的 `admin`，所以她访问不了 `data2`。

接下来在matcher中，应该像下面的例子一样检查角色信息：

```
[matchers]
m = g(r.sub, p.sub, r.dom) && r.dom == p.dom && r.obj == p.obj
&& r.act == p.act
```

更多示例参见: [rbac_with_domains_model.conf](#)。

① TOKEN名称协议

注意: 政策定义中的常规域令牌名称是 `dom` 并被放置为第二个令牌(`sub`、`dom`、`obj`、`act`)。现在Golang Casbin 支持定制令牌名称 & 位置。如果域令牌名称是 `dom`, 域令牌可以放置在任意位置, 不需要额外的操作。如果域令牌名称不是 `dom`, 应在启动执行者后调用 `eSetFieldIndex()` 用于 `constant.DomainIndex` 不论执行者的地位如何。

```
# "domain" 在这里为 "dom"
[policy_definition]
p = sub, obj, act, domain
```

```
e.SetFieldIndex("p", constant.DomainIndex, 3) // 索引从 0 开始
users := e.GetAllUsersByDomain("domain1") // 没有
SetFieldIndex, 它会引起一个错误
```

Casbin RBAC和RBAC96

Casbin RBAC和RBAC96

在这个文档中，我们会比较Casbin RBAC与 RBAC96的区别。

Casbin RBAC支持RBAC96的几乎所有特点，并在此基础上增加了新的特点。

RBAC 版本	支持级别	说明
RBAC0	完全支持	RBAC0是RBAC96的基本版本。它澄清了使用者、角色和权限之间的关系。
RBAC1	完全支持	Casbin 的 RBAC 支持 RBAC1 的角色层次结构功能，如果 <code>alice</code> 具有 <code>role1</code> , <code>role1</code> 具有 <code>role2</code> , 则 <code>alice</code> 也将拥有 <code>role2</code> 并继承其权限。
RBAC2	支持相互专用 处理(示例), 但量化限制不 是	RBAC2在RBAC0的基础上添加了约束 因此，RBAC2可以处理政策中相互排斥的问题。
RBAC3	支持相互专用 处理(示例), 但量化限制不 是	RBAC3是RBAC1和RBAC2的组合。RBAC3支持RBAC1和RBAC2中的角色等级和制约因素。

Casbin RBAC和RBAC96之间的差异

1. 在Casbin, 用户和角色之间的区分不明确。

在Casbin中, 用户和角色都被视为字符串。如果您写了类似于以下的策略文件:

```
p, admin, book, read  
p, alice, book, read  
g, amber, admin
```

并调用方法 `GetAllSubjects()` 就像这样(`e` 是一个Casbin Enforcer的实例):

```
e.GetAllSubjects()
```

然后您将得到下面的返回值:

```
[admin alice bob]
```

因为在Casbin, 主体包括用户和角色。

然而, 如果你调用方法 `GetAllRoles()` 就像这样:

```
e.GetAllRoles()
```

然后您将得到下面的返回值:

```
[admin]
```

现在你知道卡斯宾的用户和角色之间有区别，但并不像RBAC96那样尖锐。当然，您可以为您的政策添加一些前缀，如 `user::alice`, `role::admin` 以澄清他们之间的关系。

2. Casbin RBAC比RBAC96提供了更多的权限

RBAC96中只定义了7个权限：读、写、添加、执行、信贷、借记、查询。

然而，在Casbin中，我们将权限视为字符串。这种方式，您可以创建一些更适合您的权限。

3. Casbin RBAC支持域

在 Casbin 中，您可以通过域名进行授权。此功能使您的访问控制模型更加灵活。

ABAC

什么是ABAC模式？

ABAC是 基于属性的访问控制，可以使用主体、客体或动作的属性，而不是字符串本身来控制访问。您之前可能就已经听过 XACML，是一个复杂的 ABAC 访问控制语言。与 XACML相比，Casbin的ABAC非常简单：在ABAC中，可以使用struct(或基于编程语言的类实例) 而不是字符串来表示模型元素。

例如，ABAC的官方实例如下：

```
[request_definition]
r = sub, obj, act

[policy_definition]
p = sub, obj, act

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = r.sub == r.obj.Owner
```

我们在 matcher 中使用 r.obj.Owner 代替 r.obj。在 Enforce() 函数中传递的 r.obj 函数是结构或类实例，而不是字符串。Casbin将使用映像来检索 obj 结构或类中的成员变量。

这里是 r.obj construction 或 class 的定义：

```
type testResource struct {
    Name string
    Owner string
}
```

如何使用ABAC?

简单地说，要使用ABAC，您需要做两件事：

1. 在模型匹配器中指定属性。
2. 将元素的结构或类实例作为Casbin的Enforce() 的参数传入。



目前，仅有形如 `r.sub`, `r.obj`, `r.act` 等请求元素支持ABAC。您不能在 `policy` 元素上使用它，比如 `p.sub`，因为在Casbin的 `policy` 中没有定义结构或者类。



您可以在匹配器中使用多个ABAC属性，例如：`m = r.sub.Domain == r.obj.Domain`。



逗号与csv的分隔符相冲突，如果你需要在策略中使用逗号，需要避免冲突。

Casbin 通过 `csv` 库解析策略文件，你可以用引号包裹语句。例如，

`"keyMatch("bob", r.sub.Role)"` 将不会被视为csv文件分割。

适配复杂而庞大的ABAC规则

上述ABAC实例的核心非常简单，但授权系统通常需要许多非常复杂的ABAC规则。为了满足这一需要，上述实现将在很大程度上使得模型冗长复杂。因此，我们可以选择在策略中添加规则代替在模型中添加规则。这是通过引入一个 `eval()` 功能结构完成的。下面是此类ABAC模型的示例。

这是用于定义ABAC模型的 `CONF` 文件。

```
[request_definition]
r = sub, obj, act

[policy_definition]
p = sub_rule, obj, act

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = eval(p.sub_rule) && r.obj == p.obj && r.act == p.act
```

在这里，`p.sub_rule` 是由策略中使用的必要属性组成的结构类型或类类型(用户定义类型)。

这是针对 `Enforcement` 模型使用的策略 现在您就可以使用作为参数传递到 `eval()` 函数的对象实例来定义某些ABAC约束条件。

```
p, r.sub.Age > 18, /data1, read
p, r.sub.Age < 60, /data2, write
```

优先级模型

Casbin支持参考优先级加载策略。

通过隐式优先级加载策略

这非常简单，顺序决定了策略的优先级，策略出现的越早优先级就越高。

model.conf:

```
[policy_effect]
e = priority(p.eft) || deny
```

通过显式优先级加载策略

另见: [casbin#550](#)

较小的优先值将具有较高的优先级。如果在优先级中有一个非数字字符，它将是最后的，而不是出现错误。

⚠ TOKEN名称协议

策略定义中的优先级令牌名称通常是“优先级”。一个定制的需要调用 `e.SetFieldIndex()` 和重新加载策略(详情见 [TestCustomedFieldIndex](#))。

model.conf:

```
[policy_definition]
p = customized_priority, sub, obj, act, eft
```

Golang代码示例：

```
e, _ := NewEnforcer("./example/
priority_model_explicit_customized.conf",
                    "./example/
priority_policy_explicit_customized.csv")
//由于自定义优先级令牌，执行者未能处理优先级。
ok, err := e.Enforce("bob", "data2", "read") // 结果将是
"ture, nil"
// 设置 PriorityIndex 并重新加载
e.SetFieldIndex("p", constant.PriorityIndex, 0)
err := e.LoadPolicy()
if err != nil {
    log.Fatalf("LoadPolicy: %v", err)
}
ok, err := e.Enforce("bob", "data2", "read") // 结果将是
`false, nil`
```

现在，明确的优先级仅支持 `AddPolicy` & `AddPolicies`，如果 `UpdatePolicy` 被调用，那么您不应该改变优先级属性。

model.conf:

```
[request_definition]
r = sub, obj, act

[policy_definition]
p = priority, sub, obj, act, eft
```

policy.csv

```
p, 10, data1_deny_group, data1, read, deny
p, 10, data1_deny_group, data1, write, deny
p, 10, data2_allow_group, data2, read, allow
p, 10, data2_allow_group, data2, write, allow

p, 1, alice, data1, write, allow
p, 1, alice, data1, read, allow
p, 1, bob, data2, read, deny

g, bob, data2_allow_group
g, alice, data1_deny_group
```

请求:

```
alice, data1, write --> true // `p, 1, alice, data1, write,
allow` 拥有最高级的优先权
bob, data2, read --> false
bob, data2, write --> true // 对于bob是 `data2_allow_group` 的角色
可以写入data2, 而且没有具有更高优先级的否认策略
```

基于角色和用户层次结构以优先级加载策略

角色和用户的继承结构只能是多棵树，而不是图。如果一个用户有多个角色，您必须确保用户在不同树上有相同的等级。如果两种角色具有相同的等级，那么出现早的策略（相应的角色）就显得更加优先。更多详情请看 [casbin#833](#)、[casbin#831](#)

model.conf:

```
[request_definition]
r = sub, obj, act

[policy_definition]
p = sub, obj, act, eft

[role_definition]
g = _, _

[policy_effect]
e = subjectPriority(p.eft) || deny

[matchers]
m = g(r.sub, p.sub) && r.obj == p.obj && r.act == p.act
```

policy.csv

```
p, root, data1, read, deny
p, admin, data1, read, deny

p, editor, data1, read, deny
p, subscriber, data1, read, deny

p, jane, data1, read, allow
p, alice, data1, read, allow

g, admin, root

g, editor, admin
g, subscriber, admin

g, jane, editor
g, alice, subscriber
```

请求:

```
jane, data1, read --> true //jane在最底部,所以优先级高于editor,  
admin 和 root  
alice, data1, read --> true
```

像这样的角色层次结构:

```
角色: 根  
  └ 角色: 管理员  
    └ 角色编辑器  
      └ 用户: 简  
    └ 角色: 订阅者  
      └ 用户: 爱丽丝
```

优先级类似于:

```
role: root # 自动优先级: 30  
--role: admin# 自动优先级: 20  
--role: editor # 自动优先级: 10  
--role: subscriber # 自动优先级: 10
```

超级管理员

超级管理员是整个系统的管理员。我们可以在RBAC, ABAC以及带域的RBAC等模型中使用它 具体例子如下:

```
[request_definition]
r = sub, obj, act

[policy_definition]
p = sub, obj, act

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = r.sub == p.sub && r.obj == p.obj && r.act == p.act || r.sub
== "root"
```

它说明了对于定义的 `request_definition`, `policy_definition`, `policy_effect` 和 `matcher`, Casbin 判断如果请求可以匹配的上策略, 或者更重要的, 如果 `sub` 是超级用户的话。一旦判决正确, 就允许授权, 而且用户有权做一切。
就像Linux系统的root一样, 用户被授权为 root, 我们就有访问所有文件和设置的权限。因此, 如果我们想要 `sub` 能够完全访问整个系统。我们可以让它成为超级管理员, 然后 `sub` 才有权做一切。



>

存储

存储



Model 的存储

Model 的存储



Policy的存储

Policy的存储



策略子集加载

加载过滤策略

Model 的存储

与 policy 不同，model 只能加载，不能保存。因为我们认为 model 不是动态组件，不应该在运行时进行修改，所以我们没有实现一个 API 来将 model 保存到存储中。

但是，好消息是，我们提供了三种等效的方法来静态或动态地加载模型：

从 .CONF 文件中加载 model

这是使用Casbin的最常见方式。当您请求Casbin团队帮助时，对初学者很容易理解，并且方便分享。

.CONF 文件的内容 examples/rbac_model.conf:

```
[request_definition]
r = sub, obj, act

[policy_definition]
p = sub, obj, act

[role_definition]
g = _, _

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = g(r.sub, p.sub) && r.obj == p.obj && r.act == p.act
```

接着你可以加载模型文件如下：

```
e := casbin.NewEnforcer("examples/rbac_model.conf", "examples/rbac_policy.csv")
```

从代码加载 model

模型可以从代码中动态初始化，而不是使用 `.CONF` 文件。以下是RBAC模式的一个示例：

```
import (
    "github.com/casbin/casbin/v2"
    "github.com/casbin/casbin/v2/model"
    "github.com/casbin/casbin/v2/persist/file-adapter"
)

// 从Go代码初始化模型
m := model.NewModel()
m.AddDef("r", "r", "sub, obj, act")
m.AddDef("p", "p", "sub, obj, act")
m.AddDef("g", "g", "_, _")
m.AddDef("e", "e", "some(where (p.eft == allow))")
m.AddDef("m", "m", "g(r.sub, p.sub) && r.obj == p.obj && r.act
== p.act")

// 从CSV文件adapter加载策略规则
// 使用自己的 adapter 替换。
a := fileadapter.NewAdapter("examples/rbac_policy.csv")

// 创建enforcer
e := casbin.NewEnforcer(m, a)
```

从字符串加载的 model

或者您可以从多行字符串中加载整个模型文本。 这种方式的好点是你不需要维护模型文件。

```
import (
    "github.com/casbin/casbin/v2"
    "github.com/casbin/casbin/v2/model"
)

// 从字符串初始化模型
text :=
`  

[request_definition]
r = sub, obj, act

[policy_definition]
p = sub, obj, act

[role_definition]
g = _, _

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = g(r.sub, p.sub) && r.obj == p.obj && r.act == p.act
`  

m, _ := model.NewModelFromString(text)

// 从CSV文件adapter加载策略规则
// 使用自己的 adapter 替换
a := fileadapter.NewAdapter("examples/rbac_policy.csv")
```


Policy的存储

在Casbin, 策略存储作为 [适配器](#) 来实现。

从 CSV 文件载入策略

这是使用Casbin的最常见方式。 当您请求Casbin团队帮助时, 对初学者很容易理解, 并且方便分享。

[CSV](#) 文件示例 [rbac_policy.csv](#)

```
p, alice, data1, read
p, bob, data2, write
p, data2_admin, data2, read
p, data2_admin, data2, write
g, alice, data2_admin
```

备注

如果你的文件包含逗号  , 你应该用双引号把它包裹, 例如:

```
p, alice, "data1,data2", read --correcy
p, alice, data1,data2, read --insur ("data1,data2" 应该是一个整体)
```

如果您的文件包含逗号  和双引号  , 你应该用双引号将字段放在一起, 并将任何嵌入的双引号加倍。

```
p, alice, data, "r.act in (''get'', ''post'')" --correct  
p, alice, data, "r.act in ('get', 'post')" --insur --  
unction (should use "" to fescape "")
```

相关问题: [issue#886](#)

适配器 API

接口名	类型	描述
LoadPolicy()	基本设置	从持久层中加载policy规则
SavePolicy()	基本设置	将policy规则保存至持久层
AddPolicy()	可选	添加单条policy规则至持久层
RemovePolicy()	可选	从持久层删除单条policy规则
RemoveFilteredPolicy()	可选	从持久层删除符合筛选条件的policy规则

数据库存储格式

您的策略文件

```
p, data2_admin, data2, read  
p, data2_admin, data2, write
```

相应的数据库结构(比如 MySQL)

<code>id</code>	<code>ptype</code>	<code>v0</code>	<code>v1</code>	<code>v2</code>	<code>v3</code>	<code>v4</code>	<code>v5</code>
1	p	data2_admin	data2	可读			
2	p	data2_admin	data2	可写			
3	g	alice	admin				

每一列的含义

- `id`: 仅存在于数据库中作为主键。 不是 级联策略 的一部分。 它生成的方式取决于特定的适配器
- `ptype`: 它对应 `p`, `g`, `g2`, 等等。
- `v0-v5`: 列名称没有特定的意义, 并对应 `policy csv` 从左到右的值。 列数取决于您自己定义的数量。 理论上, 可以有无限的列数。 但通常在适配器中只有 6 列。 如果您觉得还不够, 请向相应的适配器仓库提交问题。

适配器详情

更多关于适配器api和数据库表结构设计的详细信息, 请转到: </docs/adapters>

策略子集加载

一些adapter支持过滤策略管理。这意味着Casbin加载的策略是基于给定过滤器的存储策略的子集。当解析整个策略成为性能瓶颈时，这将会允许在大型多租户环境中有效地执行策略。

要使用支持的adapter处理过滤后的策略，只需调用 `LoadFilteredPolicy` 方法。过滤器参数的有效格式取决于所用的适配器。为了防止意外数据丢失，当策略已经加载，`SavePolicy` 方法会被禁用。

例如，下面的代码片段使用内置的过滤文件adapter和带有域的RBAC模型。在本例中，过滤器将策略限制为单个域。除 `"domain1"` 以外的任何域策略行被忽略：

```
import (
    "github.com/casbin/casbin/v2"
    fileadapter "github.com/casbin/casbin/v2/persist/file-
adapter"
)

enforcer, _ := casbin.NewEnforcer()

adapter := fileadapter.NewFilteredAdapter("examples/
rbac_with_domains_policy.csv")
enforcer.InitWithAdapter("examples/
rbac_with_domains_model.conf", adapter)

filter := &fileadapter.Filter{
    P: []string{"", "domain1"},
    G: []string{"", "", "domain1"},
}
enforcer.LoadFilteredPolicy(filter)
```

还有另一种方法支持子集加载功能：`LoadIncrementalFilteredPolicy`。
负载增量
过滤策略 类似于`LoadFilteredPolicy`, 但它没有清除之前加载的策略, 只是附加到了策略里



>

扩展功能

扩展功能

执行器

执行器是Casbin的主要结构。它作为用户在规则和模式上开展业务的一个接口

适配器

支持的适配器和用法

监视器

保持多个Casbin执行实例之间的一致性

调度器

调度器提供了同步递增策略变化的方法。

角色管理器

角色管理器用于Casbin的RBAC角色层次结构

中间件

Casbin 中间件

Graphql-中间件

graphQL端点授权

云端原生中间件

云端原生中间件

执行器

`Enforcer` 是 Casbin 的主要结构。它是用户就规则和模式开展业务的一个接口。

支持的执行器

Casbin 的适配器完整列表如下。我们欢迎来自第三方的适配器，请通知我们，以将您的适配器加入列表：)

[Go](#)

执行器	作者	说明
执行器	Casbin	<code>Enforcer</code> 是用户与 Casbin 策略和模型交互的基本结构。您可以在这里 找到与 Enforcer 相关 API 的更多详细信息 。
CachedEnforcer	Casbin	<code>CachedEnforcer</code> 基于 <code>Enforcer</code> 。他支持将请求的执行结果缓存在内存中(通过使用 map)，并且在预定的过期时间后清除缓存。此外，它通过 Read-Write 锁来保证线程安全。您可以使用 <code>EnableCache</code> 来启用执行结果缓存(默认启用)。 <code>CachedEnforcer</code> 的其他 API 与 <code>Enforcer</code> 相同。
分布式执行	Casbin	<code>DistributedEnforcer</code> 支持分布式集群中的多个实例。它为调度器包装了 <code>SyncedEnforcer</code> 。您可以在 这里找到更多关于调度器的详细信息 。

执行器	作者	说明
SyncedEnforcer	Casbin	SyncedEnforcer 基于 Enforcer 并提供同步访问。它是线程安全的。

适配器

在Casbin中，策略存储作为adapter(Casbin的中间件)实现。Casbin用户可以使用adapter从存储中加载策略规则(aka `LoadPolicy()`)或者将策略规则保存到其中(aka `SavePolicy()`)。为了保持代码轻量级，我们没有把adapter代码放在主库中。

目前支持的适配器列表

Casbin的适配器完整列表如下。我们欢迎任何第三方对adapter进行新的贡献，如果有请通知我们，我们将把它放在这个列表中:)

[Go](#) [Java](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Ruby](#) [Swift](#) [Lua](#)

适配器	类型	作者	自动保存	描述
File Adapter (内置)	File	Casbin	✗	For .CSV (Comma-Separated Values) files
Filtered File Adapter (内置)	File	File	✗	对于 CSV (逗号分隔的值) 个带策略子集加载支持的文件
SQL Adapter	SQL	@Blank-Xu	✓	MySQL, PostgreSQL, SQL Server和SQLite3在 <code>master</code> 分支中受到 <code>database/sql</code> 的支持，Oacle在 <code>oracle</code> 分支中也受到 <code>database/sql</code> 的支持
Xorm Adapter	ORM	Casbin	✓	通过 Xorm 实现，支持 MySQL, PostgreSQL, Sqlite3, SQL Server 等多种存储引擎的 adapter

适配器	类型	作者	自动保存	描述
GORM Adapter	ORM	Casbin	✓	MySQL, PostgreSQL, Sqlite3, SQL Server are supported by GORM
GORM Adapter Ex	ORM	Casbin	✓	MySQL, PostgreSQL, Sqlite3, SQL Server are supported by GORM
Ent Adapter	ORM	Casbin	✓	MySQL, MariaDB, PostgreSQL, SQLite, 基于 Gremlin的图数据库由 [ent ORM](https://entgo.io/) 支持。
Beego ORM Adapter	ORM	Casbin	✓	MySQL, PostgreSQL, Sqlite3 由 Beego ORM 支持
SQLX Adapter	ORM	@memwey	✓	MySQL, PostgreSQL, SQLite, Oracle 由 SQLX 支持
Sqlx Adapter	SQL	@Blank-Xu	✓	MySQL, PostgreSQL, SQL Server, SQLite3 由 master 分支支持, Oracle由 oracle 分支受到 sqlx 的支持
GF ORM Adapter	ORM	@vance-liu	✓	MySQL, SQLite, PostgreSQL, Oracle, SQL Server 受到 GoFrame ORM 的支持
GoFrame ORM Adapter	ORM	@kotlin2018	✓	MySQL, SQLite, PostgreSQL, Oracle, SQL Server 受到 GoFrame ORM 的支持
Filtered PostgreSQL Adapter	SQL	Casbin	✓	用于 PostgreSQL
Filtered pgx Adapter	SQL	@pckhoi	✓	使用 pgx 支持PostgreSQL

适配器	类型	作者	自动保存	描述
PostgreSQL Adapter	SQL	@cychiuae	✓	用于 PostgreSQL
RQLite Adapter	SQL	EDOMO Systems	✓	用于 RQLite
MongoDB Adapter	NoSQL	Casbin	✓	基于 MongoDB Go Driver 用于 MongoDB
RethinkDB Adapter	NoSQL	@adityapandey9	✓	用于 RethinkDB
Cassandra Adapter	NoSQL	Casbin	✗	用于 Apache Cassandra DB
DynamoDB Adapter	NoSQL	HOOQ	✗	用于 Amazon DynamoDB
Dynacasbin	NoSQL	NewbMiao	✓	用于 Amazon DynamoDB
ArangoDB Adapter	NoSQL	@adamwasila	✓	用于 ArangoDB
Amazon S3 Adapter	云	Soluto	✗	用于 Minio 和 Amazon S3
Azure Cosmos DB Adapter	云	@spacycoder	✓	用于 Microsoft Azure Cosmos DB
GCP Firestore Adapter	云	@reedom	✗	用于 Google Cloud Platform Firestore

适配器	类型	作者	自动保存	描述
GCP Cloud Storage Adapter	云	qurami	✗	用于 Google Cloud Platform Cloud Storage
GCP Cloud Storage Adapter	云	@flowerinthenight	✓	用于 Google Cloud Platform Cloud Spanner
Consul Adapter	KV store	@ankitm123	✗	用于 HashiCorp Consul
Redis 适配器 (Redigo)	KV store	Casbin	✓	用于 Redis
Redis Adapter (go-redis)	KV store	@milsen	✓	用于 Redis
Etcd 适配器	KV store	@sebastianliu	✗	用于 etcd
BoltDB Adapter	KV store	@speza	✓	用于 Bolt
Bolt Adapter	KV store	@wirepair	✗	用于 Bolt
BadgerDB Adapter	KV store	@inits	✓	用于 BadgerDB
Protobuf Adapter	Stream	Casbin	✗	用于 Google Protocol Buffers

适配器	类型	作者	自动保存	描述
JSON Adapter	String	Casbin	✗	用于 JSON
String Adapter	String	@qiangmzsx	✗	用于 String
HTTP File Adapter	HTTP	@h4ckedneko	✗	用于 http.FileSystem
FileSystem Adapter	File	@naucon	✗	用于 fs.FS 和 embed.FS
适配器	类型	作者	自动保存	描述
File Adapter (内置)	File	Casbin	✗	For .CSV (Comma-Separated Values) files
JDBC Adapter	JDBC	Casbin	✓	MySQL, Oracle, PostgreSQL, DB2, Sybase, SQL 服务器由 JDBC 支持
Hibernate Adapter	ORM	Casbin	✓	Oracle, DB2, SQL Server, Sybase, MySQL, PostgreSQL 由 Hibernate 支持
MyBatis Adapter	ORM	Casbin	✓	MySQL, Oracle, PostgreSQL, DB2, Sybase, SQL Server (与 JDBC 相同) 由 MyBatis 3 支持
Hutool Adapter	ORM	@mapleafgo	✓	MySQL, Oracle, PostgreSQL, SQLite 由 Hutool 支持

适配器	类型	作者	自动保存	描述
MongoDB Adapter	NoSQL	Casbin	✓	MongoDB 由 Mongodb-driver-sync 支持
DynamoDB Adapter	NoSQL	Casbin	✗	用于 Amazon DynamoDB
Redis Adapter	KV store	Casbin	✓	用于 Redis
适配器	实现要素	作者	自动保存	描述
File Adapter (内置)	File	Casbin	✗	For .CSV (Comma-Separated Values) files
Filtered File Adapter (内置)	File	Casbin	✗	对于 CSV (逗号分隔的值) 个带策略子集加载支持的文件
String Adapter (内置)	String	@calebfaruki	✗	用于字符串
Basic Adapter	Native ORM	Casbin	✓	pg, mysql, mysql2, sqlite3, oracedb, mssql 是适配器本身支持的
Sequelize Adapter	ORM	Casbin	✓	MySQL、PostgreSQL、SQLite、Microsoft SQL Server 由 Sequelize 支持

适配器	实现要素	作者	自动保存	描述
TypeORM Adapter	ORM	Casbin	✓	MySQL, PostgreSQL, MariaDB, SQLite, MS SQL Server, Oracle, WebSQL, MongoDB 由 TypeORM 支持
Prisma Adapter	ORM	Casbin	✓	MySQL, PostgreSQL, MariaDB, SQLite, MS SQL Server, AWS Aurora, Azure SQL 由 Prisma 支持
Knex Adapter	ORM	@sarneeh and knex	✓	MSSQL, MySQL, PostgreSQL, SQLite3, Oracle 由 Knex.js 支持
Objection.js Adapter	ORM	@willsoto	✓	MSSQL, MySQL, PostgreSQL, SQLite3, Oracle 由 Objection.js 支持
Node PostgreSQL Native Adapter	SQL	@touchifyapp	✓	PostgreSQL 适配器，拥有高级策略子集加载支持以及由 [node-postgres](https://node-postgres.com/) 构建的更好的性能。
MongoDB Adapter	NoSQL	elastic.io 和 Casbin	✓	MongoDB 由 Mongoose 支持
Mongoose 适配器 (无交易)	NoSQL	minhducck	✓	MongoDB 由 Mongoose 支持
Node MongoDB Native Adapter	NoSQL	@juicycleff	✓	用于 Node MongoDB Native
DynamoDB Adapter	NoSQL	@fospitia	✓	用于 Amazon DynamoDB

适配器	实现要素	作者	自动保存	描述
Couchbase Adapter	NoSQL	@MarkMYoung	✓	用于 Couchbase
Redis Adapter	KV store	Casbin	✗	用于 Redis
Redis Adapter	KV store	@NandaKishorJeripothula	✗	用于 Redis
适配器	类型	作者	自动保存	描述
File Adapter (内置)	File	Casbin	✗	For .CSV (Comma-Separated Values) files
数据库适配器	ORM	Casbin	✓	MySQL, PostgreSQL, SQLite, Microsoft SQL Server 由 techone/database 支持
Zend Db 适配器	ORM	Casbin	✓	MySQL, PostgreSQL, SQLite, Oracle, IBM DB2, Microsoft SQL Server, 其他 PDO Driver 由 zend-db 支持
Doctrine DBAL 适配器(建议)	ORM	Casbin	✓	强大的 PHP 数据库抽象层(DBAL), 具有数据库架构内省和管理的许多功能。
Medoo 适配器	ORM	Casbin	✓	Medoo 是一个用来加速开发的轻量PHP 数据库框架。 支持所有 SQL 数据库, 包括 MySQL, MSSQL, SQLite, MariaDB, PostgreSQL, Sybase, Oracle 以及更多。

适配器	类型	作者	自动保存	描述
Laminas-db 适配器	ORM	Casbin	✓	MySQL, PostgreSQL, Oracle, IBM DB2, Microsoft Sql Server, PDO等都由 laminas-db 支持。
Zend-db 适配器	ORM	Casbin	✓	MySQL, PostgreSQL, Oracle, IBM DB2, Microsoft Sql Server, PDO等都由 zend db 支持。
Redis Adapter	KV store	@nsnake	✗	用于 Redis
适配器	类型	作者	自动保存	描述
File Adapter (内置)	File	Casbin	✗	For .CSV (Comma-Separated Values) files
Django ORM Adapter	ORM	Casbin	✓	PostgreSQL、MariaDB、MySQL、Oracle、SQLite、IBM DB2、Microsoft SQL Server、Firebird、ODBC 都由 Django ORM 支持
SQLObject Adapter	ORM	Casbin	✓	PostgreSQL、MySQLite、Microsoft SQL

适配器	类型	作者	自动保存	描述
				Server、 Firebird、 Sybase、MAX DB、 pyfirebirdsql 都 由 SQLObject 支持。
SQLAlchemy 适配器	ORM	Casbin	✓	PostgreSQL、 MySQLite、 Oracle、 Microsoft SQL Server、 Firebird、 Sybase 由 SQLAlchemy 支 持
异步数据库适 配器	ORM	sampingantech	✓	PostgreSQL、 MySQLite、 Oracle、 Microsoft SQL Server、 Firebird、 Sybase 由 Databases 支 持。
Peewee 适配 器	ORM	@shblhy	✓	PostgreSQL、 MySQL、 SQLite 由 Peewee 支持
MongoEngine 适配器	ORM	@zhangbailong945	✗	MongoDB 由 MongoEngine 支持

适配器	类型	作者	自动保存	描述
Pony ORM Adapter	ORM	@drorvinkler	✓	MySQL, PostgreSQL, SQLite, Oracle, CockroachDB 由Pony ORM支持
Tortoise ORM Adapter	ORM	@thearchitector	✓	PostgreSQL (>=9.4)、 MySQL、 MariaDB 和 SQLite 由 Tortoise ORM 支持
Async Ormar Adapter	ORM	@shepilov-vladislav	✓	PostgreSQL、 MySQL、 SQLite 由 Ormar 支持
SQLModel Adapter	ORM	@shepilov-vladislav	✓	PostgreSQL、 MySQL、 SQLite 由 SQLModel
Couchbase Adapter	NoSQL	ScienceLogic	✓ (没有 <code>remove_filtered_policy()</code>)	用于 Couchbase
DynamoDB Adapter	NoSQL	@abqadeer	✓	用于 DynamoDB
Pymongo 适配器	NoSQL	Casbin	✗	MongoDB 由 [Pymongo 支持](https://pypi.org/project/pymongo/)

适配器	类型	作者		自动保存	描述
适配器	类型	作者	自动保存	描述	
GCP Firebase Adapter	Cloud	@devrushi41	✓	用于Google Cloud Platform Firebase	
File Adapter (内置)	File	Casbin	✗	For .CSV (Comma-Separated Values) files	
EF 适配器	ORM	Casbin	✗	MySQL、PostgreSQL、SQLite、Microsoft SQL Server、Oracle, DB2 等都由 Entity Framework 6 支持。	
EFCORE 适配器	ORM	Casbin	✓	MySQL, PostgreSQL、SQLite, Microsoft SQL Server, Oracle、DB2, 等都由 Entity Framework Core 支持	
EFCORE 适配器 (.NET Core 5)	ORM	@g4dvali	✓	MySQL, PostgreSQL、SQLite, Microsoft SQL Server, Oracle、DB2, 等都由 Entity Framework Core 支持	
Linq2DB Adapter	ORM	@Tirael	✓	MySQL, PostgreSQL, SQLite, Microsoft SQL Server, Oracle, Access, Firebird, Sybase, etc. are supported by linq2db	
适配器	类型	作者		自动保存	描述
File Adapter (内置)	File	Casbin		✗	For .CSV (Comma-Separated Values) files
Diesel 适配器	ORM	Casbin		✓	SQLite, PostgreSQL, MySQL 由 Diesel 支持

适配器	类型	作者	自动保存	描述
Sqlx Adapter	ORM	Casbin	✓	PostgreSQL, MySQL 由 Sqlx 支持, 可实现完全异步操作
SeaORM Adapter	ORM	@lingdu1234	✓	PostgreSQL, MySQL 由 Sqlx 支持, 可实现完全异步操作
Rbatis Adapter	ORM	rbatis	✓	MySQL, PostgreSQL, Sqlite, SQL Server, MariaDB, TiDB, CockroachDB, Oracle are supported by Rbatis
DynamodDB Adapter	NoSQL	@fospitia	✓	用于 Amazon DynamoDB
JSON Adapter	String	Casbin	✓	用于 JSON
YAML 适配器	String	Casbin	✓	用于 [YAML](https://yaml.org/)
适配器	类型	作者	自动保存	描述
File Adapter (内置)	File	Casbin	✗	For .CSV (Comma-Separated Values) files
Sequelize Adapter	ORM	CasbinRuby	✓	ADO, Amalgalite, IBM_DB, JDBC, MySQL, Mysql2, ODBC, Oracle, PostgreSQL, SQLAnywhere SQLite3, 和 TinyTDS 都由 Sequel 支持

适配器	类型	作者	自动保存	描述
File Adapter (内置)	File	Casbin	✗	For .CSV (Comma-Separated Values) files
Memory Adapter (内置)	内存	Casbin	✗	用于内存
Fluent Adapter	ORM	Casbin	✓	PostgreSQL、SQLite、MySQL、MongoDB 由 Fluent 支持
适配器	类型	作者	自动保存	描述
File Adapter (内置)	File	Casbin	✗	For .CSV (Comma-Separated Values) files
Filtered File Adapter (内置)	File	Casbin	✗	对于 CSV (逗号分隔的值) 个带策略子集加载支持的文件
LuaSQL Adapter	ORM	Casbin	✓	MySQL, PostgreSQL, SQLite3 由 LuaSQL 支持
4DaysORM Adapter	ORM	Casbin	✓	MySQL, SQLite3 由 4DaysORM 支持

① 备注

1. 如果使用显式或隐式adapter调用 `casbin.NewEnforcer()`, 策略将自动加载。
2. 可以调用 `e.LoadPolicy()` 来从存储中重新加载策略规则。
3. 如果adapter不支持 `Auto-Save` 特性, 则在添加或删除策略时不能将策略规则自动保存回存储器。
你必须手动调用 `SavePolicy()` 来保存所有的策略规则

例子

我们为您提供以下示例作为参考:

文件适配器 (内置)

下面的代码演示了如何从File adapter初始化enforcer:

[Go](#) [PHP](#) [Rust](#)

```
import "github.com/casbin/casbin"

e := casbin.NewEnforcer("examples/basic_model.conf", "examples/basic_policy.csv")

use Casbin\Enforcer;

$e = new Enforcer('examples/basic_model.conf', 'examples/basic_policy.csv');

use casbin::prelude::*;

let mut e = Enforcer::new("examples/basic_model.conf", "examples/
basic_policy.csv").await?;
```

它等同于如下代码：

[Go](#) [PHP](#) [Rust](#)

```
import (
    "github.com/casbin/casbin"
    "github.com/casbin/casbin/file-adapter"
)

a := fileadapter.NewAdapter("examples/basic_policy.csv")
e := casbin.NewEnforcer("examples/basic_model.conf", a)

use Casbin\Enforcer;
use Casbin\Persist\Adapters\FileAdapter;

$a = new FileAdapter('examples/basic_policy.csv');
$e = new Enforcer('examples/basic_model.conf', $a);

use casbin::prelude::*;

let a = FileAdapter::new("examples/basic_policy.csv");
let e = Enforcer::new("examples/basic_model.conf", a).await?;
```

MySQL 适配器

下面展示了如何从MySQL数据库初始化一个enforcer。此处样例中的MySQL数据库运行在127.0.0.1:3306上，用户为root，密码为空。

[Go](#) [Rust](#) [PHP](#)

```
import (
    "github.com/casbin/casbin"
    "github.com/casbin/mysql-adapter"
)

a := mysqladapter.NewAdapter("mysql", "root:@tcp(127.0.0.1:3306)/")
e := casbin.NewEnforcer("examples/basic_model.conf", a)

// https://github.com/casbin-rs/diesel-adapter
// 请确保您激活了 `mysql` 特性

use casbin::prelude::*;
use diesel_adapter::{ConnOptions, DieselAdapter};

let mut conn_opts = ConnOptions::default();
conn_opts
    .set_hostname("127.0.0.1")
    .set_port(3306)
    .set_host("127.0.0.1:3306") // overwrite hostname, port config
    .set_database("casbin")
    .set_auth("casbin_rs", "casbin_rs");

let a = DieselAdapter::new(conn_opts)?;
let mut e = Enforcer::new("examples/basic_model.conf", a).await?;

// https://github.com/php-casbin/dbal-adapter

use Casbin\Enforcer;
use CasbinAdapter\DBAL\Adapter as DatabaseAdapter;

$config = [
    // Either 'driver' with one of the following values:
    // pdo_mysql,pdo_sqlite,pdo_pgsql,pdo_oci (unstable),pdo_sqlsrv,pdo_sqllsr,
    // mysqli,sqlanywhere,sqllsr,ibm_db2 (unstable),drizzle_pdo_mysql
```

使用自建的adapter

您可以参考如下代码来使用自建的 adapter

```
import (
    "github.com/casbin/casbin"
    "github.com/your-username/your-repo"
)

a := yourpackage.NewAdapter(params)
e := casbin.NewEnforcer("examples/basic_model.conf", a)
```

在不同适配器之间转移/转换

如果您想将适配器从 A 转换为 B, 您可以这样做:

1. 从 A 加载策略到内存

```
e, _ := NewEnforcer(m, A)
```

或者

```
e.SetAdapter(A)
e.LoadPolicy()
```

2. 将您的适配器从 A 转换为 B

```
e.SetAdapter(B)
```

3. 将策略从内存保存到 B

```
e.LoadPolicy()
```

在运行时进行加载或保存配置信息

如果您在初始化后仍想重载model和policy的配置（或是保存policy的配置信息），那么可以参照如下方法：

```
// 从CONF配置文件中加载model  
e.LoadModel()  
// 从文件或数据库中加载policy  
e.LoadModel()  
// 保存当前的policy (通常在调用Casbin API改变了配置信息后) 至文件或数据库  
e.SavePolicy()
```

自动保存

自动保存机制（Auto-Save）是adapter的特性之一。支持自动保存机制的adapter可以自动向存储回写内存中单个policy规则的变更（删除/更新）。与自动回写机制不同，调用SavePolicy()会直接删除所有存储中的policy规则并将当前Casbin enforcer存储在内存中的policy规则悉数持久化到存储中。因此，当内存中的policy规则过多时，直接调用SavePolicy()会引起一些性能问题。

当适配器支持自动保存机制时，您可以通过Enforcer.EnableAutoSave()函数来开启或关闭该机制。默认情况下启用该选项（如果适配器支持自动保存的话）。

① 备注

1. Auto-Save特性是可选的。Adapter可以选择是否实现它。
2. Auto-Save只在Casbin enforcer使用的adapter支持它时才有效。
3. 查看上述adapter列表中的AutoSave列，查看adapter是否支持Auto-Save。

以下示例演示了Auto-Save的使用方法：

```
import (  
    "github.com/casbin/casbin"  
    "github.com/casbin/xorm-adapter"  
    _ "github.com/go-sql-driver/mysql"  
)  
  
// enforcer会默认开启AutoSave机制。  
a := xormadapter.NewAdapter("mysql",
```

更多示例参见: https://github.com/casbin/xorm-adapter/blob/master/adapter_test.go

如何编写 Adapter

Adapter应实现Adapter中定义的接口，其中必须实现的为LoadPolicy(model model.Model) error和SavePolicy(model model.Model) error。

其他三个函数是可选的。如果adapter支持Auto-Save特性，则应该实现它们。

接口名	类型	描述
LoadPolicy()	必须	从持久层中加载policy规则
SavePolicy()	必须	将policy规则保存至持久层
AddPolicy()	可选	添加单条policy规则至持久层
RemovePolicy()	optional	从持久层删除单条policy规则
RemoveFilteredPolicy()	可选	从持久层删除符合筛选条件的policy规则

① 备注

如果适配器不支持自动保存，它应该为以下三个可选函数提供一个空实现。下面是Golang的一个例子：

```
// AddPolicy 添加一个policy规则至持久层
func (a *Adapter) AddPolicy(sec string, ptype string, rule []string) error {
    return errors.New("not implemented")
}

// RemovePolicy 从持久层删除单条policy规则
func (a *Adapter) RemovePolicy(sec string, ptype string, rule []string) error {
    return errors.New("not implemented")
}

// RemoveFilteredPolicy 从持久层删除符合筛选条件的policy规则
func (a *Adapter) RemoveFilteredPolicy(sec string, ptype string, fieldIndex int,
    fieldValues ...string) error {
    return errors.New("not implemented")
}
```

Casbin enforcer在调用这三个可选实现的接口时，会忽略返回的 `not implemented` 异常。

关于如何写入适配器的详细信息。

- 数据结构 适配器应 **最少** 支持六列。
- 数据库名称: 默认数据库名称应该是 `casbin`。
- 表格名称 默认表名应该是 `casbin_rule`。
- Ptype 栏。此列的名称应该是 `ptype` 而不是 `p_type` 或 `Ptype`。
- 表定义应该是 `(id int priorkey, ptype varchar, v0 varchar, v1 varchar, v2 varchar, v3 varchar, v4 varchar, v5 varchar)`
- 唯一的密钥索引应该建立在列 `ptype, v0, v1, v2, v3, v4, v5` 上。
- `LoadFilteredPolicy` 需要一个 `filter` 作为参数。Filter应该像这样。

```
{  
    "p": [ [ "alice" ], [ "bob" ] ],  
    "g": [ [ "", "book_group" ], [ "", "pen_group" ] ],  
    "g2": [ [ "alice" ] ]  
}
```

谁负责创建数据库？

我们通常约定，如果相应的数据库结构尚未建立，那么使用adapter作为policy的持久化工具时，它应具有自动创建一个名为 `casbin` 的数据库的能力。请使用Xorm adapter作为参考实现:<https://github.com/casbin/xorm-adapter>

监视器

我们支持使用分布式消息系统，例如 [etcd](#) 来保持多个Casbin执行器实例之间的一致性。因此，我们的用户可以同时使用多个Casbin 执行器来处理大量的权限检查请求。

与策略存储适配器类似，我们没有把watcher的代码放在主库中。任何对新消息系统的支持都应该作为watcher程序来实现。完整的Casbin `watchers`列表如下所示。欢迎任何第三方对 watcher做出新的贡献，如果有请告知我们，我将把它放在这个列表中:)

[Go](#) [Java](#) [Node.js](#) [Python](#) [.NET](#) [Ruby](#) [PHP](#)

Watcher	类型	作者	描述
PostgreSQL WatcherEx	Database	@IguteChung	WatcherEx for PostgreSQL
Redis WatcherEx	KV store	Casbin	WatcherEx for Redis
Redis Watcher	KV store	@billcobbler	Redis的监视器
Etcd Watcher	KV store	Casbin	etcd的监视器
TiKV 监视器	KV store	Casbin	TiKV 的监视器
Kafka 监视器	Messaging system	@wgarunap	Apache Kafka的监视器

Watcher	类型	作者	描述
NATS Watcher	Messaging system	Soluto	NATS的监视器
ZooKeeper Watcher	Messaging system	Grepstr	Apache ZooKeeper的监视器
NATS, RabbitMQ, GCP Pub/Sub, AWS SNS & SQS, Kafka, InMemory	Messaging System	@rusenask	监视器基于与领先的云供应商和自托管基础设施运作的Go Cloud Dev Kit
RocketMQ 观察器	Messaging system	@fmyxyz	Apache RocketMQ的监视器

Watcher	类型	作者	描述
Etcd Adapter	KV store	@mapleafgo	etcd的监视器
Redis Watcher	KV store	Casbin	Redis的监视器
Kafka 监视器	Messaging system	Casbin	Apache Kafka的监视器

Watcher	实现要素	作者	描述
Etcd Watcher	KV store	Casbin	etcd的监视器
Redis Watcher	KV store	Casbin	Redis的监视器

Watcher	实现要素		作者	描述
Pub/Sub Watcher	Messaging system		Casbin	Google Cloud Pub/Sub的监视器
Postgres Watcher	数据库		Matteo Collina	PostgreSQL的监视器
Watcher	类型		作者	描述
Etcd Watcher	KV store		Casbin	etcd的监视器
Redis Watcher	KV store		Casbin	Redis的监视器
Redis Watcher	KV store		ScienceLogic	Redis的监视器
PostgreSQL Watcher	数据库		Casbin	PostgreSQL的监视器
Watcher	类型	作者	描述	
Redis Watcher	KV store	@Sbou	Redis的监视器	
Watcher	类型		作者	描述
Redis Watcher	KV store		CasbinRuby	Redis的监视器
RabbitMQ 监视器	Messaging system		CasbinRuby	RabbitMQ 的监视器

Watcher	类型	作者	描述
Redis Watcher	KV store	@Tinywan	Redis的监视器

WatcherEx

为了支持多个实例之间的增量同步，我们提供 `WatcherEx` 接口。我们希望它能够在策略改变时通知其他实例，但目前尚未实现 `watcherEx`。我们推荐您使用调度器来实现它。

与 `个监视器` 接口相比，`watcherEx` 可以区分收到的更新动作类型。`添加策略` 和 `删除策略`。

WatcherEx 的调用接口：

应用程序 接口	描述
	当数据库中的策略已被其他实例更改时 <code>SetUpdateCallback(func(string)) error</code>
	<code>Update() error</code>
	<code>Close()</code>
	<code>UpdateForAddPolicy(sec, ptype string, params ...string) error</code>
	<code>UpdateForRemovePolicy(sec, ptype string, params ...string) error</code>
	<code>UpdateForRemoveFilteredPolicy(sec, ptype string, field Index int,</code>

应用程序 接口	描述
	fieldValues ...string) error
	UpdateForSavePolicy(model model.Model) error
	UpdateForAddPolicies(sec string, ptype string, rules ...[]string) error
	UpdateForRemovePolicies(sec string, ptype string, rules ...[]string) error

调度器

调度器提供了同步递增策略变化的方法。它应以手工艺等一致性算法为基础，以确保所有执行者的一致性和一致性。通过调度器用户们可以轻松地建立分布式集群。

调度器的方法分为两部分。第一种是与Casbin相结合的方法。这些方法应该在Casbin内部调用。用户们可以使用由Casbin本身提供的更完整的api。

另一个部分是调度器本身定义的方法，包括调度器初始化方法，和不同算法提供的不同函数，如动态资格、配置变更等。

① 备注

我们希望调度器在运行时确保Casbin执行的一致性。因此，如果初始化时策略不一致，调度器将无法正常工作。用户在使用调度器之前需要确保所有实例的状态一致。

完整的Casbin调度器列表如下所示。我们欢迎来自任何第三方的调度器，请通知我们，以将您的调度器加入列表中:)

Go

调度器	类型	作者	描述
Hashicorp Raft Dispatcher	raft	Casbin	基于 hashicorp/raft 的调度器

分布式执行

DistributedEnforcer 为调度器包装 SyncedEnforcer.

Go

```
e, _ := casbin.NewDistributedEnforcer("examples/  
basic_model.conf", "examples/basic_policy.csv")
```

角色管理器

角色管理器用于管理Casbin中的RBAC角色层次结构（用户角色映射）。角色管理员可以从 Casbin 策略规则或外部来源，例如LDAP、Okata、Auth0、Azure AD等检索角色数据。我们支持不同的角色管理器实现。为了保持轻量，我们不会将角色管理器代码放在主库中（默认角色管理器除外）。Casbin角色管理器的完整列表如下所示。我们欢迎任何第三方的角色管理器实现，如果您通知我们，我们很乐意将它加入以下列表：）

[Go](#) [Java](#) [Node.js](#) [PHP](#) [Python](#)

角色管理器	作者	描述
Default Role Manager (built-in)	Casbin	支持存储在Casbin策略中的角色层次结构
Session Role Manager	EDOMO Systems	支持存储在Casbin策略中的角色层次结构，以及基于时间范围的会话
Okta Role Manager	Casbin	支持存储在Okta中的角色层次结构
Auth0 Role Manager	Casbin	支持存储在 Auth0's Authorization Extension 授权扩展名中的角色层次结构

对于开发人员：所有角色管理器都必须实现 [RoleManager](#) 接口。会话角色管理器可以用作参考实现。

角色管理器	作者	描述
Default Role Manager (built-in)	Casbin	支持存储在Casbin策略中的角色层次结构

对于开发人员：所有角色管理器都必须实现 [RoleManager](#) 接口。默认角色管理器 可以用作参考实现。

角色管理器	作者	描述
Default Role Manager (built-in)	Casbin	支持存储在Casbin策略中的角色层次结构
Session Role Manager	Casbin	支持存储在Casbin策略中的角色层次结构，以及基于时间范围的会话

对于开发人员：所有角色管理器都必须实现 [RoleManager](#) 接口。默认角色管理器 可以用作参考实现。

角色管理器	作者	描述
Default Role Manager (built-in)	Casbin	支持存储在Casbin策略中的角色层次结构

对于开发人员：所有角色管理器都必须实现 [RoleManager](#) 接口。默认角色管理器 可以用作参考实现。

角色管理器	作者	描述
Default Role Manager (built-in)	Casbin	支持存储在Casbin策略中的角色层次结构

对于开发人员：所有角色管理器都必须实现 [RoleManager](#) 接口。 [默认角色管理器](#) 可以用作参考实现。

API

详情请参阅 [API](#) 部分。

中间件

Web框架

[Go](#) [Java](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Lua](#) [Swift](#)

名称	描述
Gin	一个有着更好性能的 HTTP 网络框架，支持类似于 Martini 的 API，通过以下插件实现： authz 或 gin-casbin
Beego	一个 Go 语言的开源、高性能网络框架，通过以下插件实现： plugins/authz
Caddy	快速、跨平台的有自动HTTPS的HTTP/2 web服务器，通过插件： caddy-authz 实现。
Traefik	云端本地应用程序代理，通过插件： Traefik-auth插件 实现
Kratos	Your ultimate Go microservices framework for the cloud-native era, via plugin: kratos-casbin
Go kit	一个用于微服务的工具包，通过内置插件： plugins/authz 实现。
Fiber	一个受到Express启发的用Go写成的Web框架，通过中间件: fiber-casbin 或者 fiber-casbinrest 或者 fiber-boilerplate 实现。

名称	描述
Revel	一个用Go语言编制的高效、全栈的web框架，通过插件： auth/casbin 实现。
Echo	高性能、简约的web框架，通过插件： echo-authz (感谢 @xqbumu) or casbinrest 实现
Iris	(这个) 地球上用Go语言编写的最快的web框架。HTTP/2 Ready-To-GO，通过插件： casbin (感写 @hiveminded) 或者 iris-middleware-casbin 实现。
GoFrame	模块化的，强力的，高性能的和企业级的Golang的应用开发框架，通过插件 gf-casbin 实现.
Negroni	Golang的惯用HTTP中间件，通过插件： negroni-authz 实现
Chi	一个用于构建 HTTP 服务的轻量级的、常用的和可组合的路由器, 通过插件: chi-authz 实现
Buffalo	基于Go的网络开发生态，致力于让你的生活更简单，通过插件： buffalo-mw-rbac 实现
Macaron	一个使用Go语言实现的高产能、模块化的网络框架，通过插件： authz 实现
DotWeb	简易的Go网络微框架，通过插件： authz 实现
Tango	微型 & 插拔式的Go网络框架，通过插件 authz 实现

名称	描述
Baa	一个带有路由，中间件，依赖注入和http context 的express Go网络框架，通过插件 authz 实现
Tyk	一个开源企业API网关，通过插件支持REST、GraphQL、TCP和gRPC 协议： tyk-authz
名称	描述
Spring Boot	让创建Spring程序和服务更加简单，通过插件： casbin-spring-boot-starter 或 jcasbin-springboot-plugin 或 使用jCasbin的SpringBoot安全示例 实现
Apache Shiro	一个强大且易于使用的 Java 安全框架，通过插件进行身份验证、授权、加密和会话管理，通过插件： shirro-casbin 或 shiro-jcasbin-spring-boot-starter 实现
Vert.x	一个用于在JVM上创建灵活应用的工具箱，通过插件： vertx-auth-jcasbin 实现
JFinal	一个简单、轻量、迅速、独立、可扩展的Java WEB + ORM 框架，通过插件： jcasbin-jfinal-plugin 实现
Nutz	适合所有所有Java开发者的WEB框架 (MVC/IOC/AOP/DAO/JSON)， 通过插件： jcasbin-nutz-plugin 实现
mangoo I/O	一个直观，轻量、高性能、全栈Java Web框架通过内置插件 AuthorizationService.java

名称	描述
Shield	一个建于 casbin 顶部的 authZ 服务器和 authZ 认知反向代理。
Express	用于node的快速，无意的，最小化的网络框架，通过插件 express-authz 实现
Koa	用于Node.JS的高表达性的中间件使用ES2017异步函数，通过 koa-authz 或者 koajs-starter 或 koa-casbin 插件实现
LoopBack 4	一个高扩展性的 Node.js 和 TypeScript 框架，通过插件构建API和微型服务： loopback4-authorization
Nest	使用TypeScript和JavaScript构建高效和可伸缩的服务器端应用程序的先进Node.js框架 通过插件： nest-authz or nest-casbin 或 NestJS Casbin 模块 或 nestjs-casbin 或 shanbe-api 或 acl-nest 或 nestjs-casbin-typeorm 实现
Fastify	用于Node.js的快速和低开销网页框架： fastify-casbin or fastify-casbin-rest
Egg	用于使用Node.Js & Koa更好地来构建企业框架和应用，通过 egg-authz 或者 egg-zrole 插件实现
hapi	简单、安全、值的开发者信赖的框架。 通过插件： hapi-authz 实现
Casbin JWT Express	使用无状态JWT token来使Casbin ACL(访问控制列表) 有效的授权中间件

名称	描述
Laravel	为网络工程师设计的PHP框架，通过插件 laravel-casbin 实现
Yii PHP Framework	快速、安全和高效的PHP框架，通过插件： yii-permission or yii-casbin
CakePHP	创建快速的稳定的PHP框架，通过 cake-casbin 插件实现
CodeIgniter	在 CodeIgniter4 网页框架中通过插件将具有角色和权限的用户联系起来，通过插件： CodeIgniter Permission 实现
ThinkPHP 5.1	ThinkPHP 5.1框架，通过插件 think-casbin 实现
ThinkPHP 6.0	ThinkPHP 6.0框架，通过 think-authz 插件实现
Symfony	Symfony PHP框架，通过插件： symfony-permission 或者 symfony-casbin
Hyperf	一个聚焦于快速和灵活的协程框架，通过插件： hyperf-permission 或者 hyperf-casbin 实现
EasySwoole	基于 Swoole 扩展实现的一个分布式的，持续存储的 PHP 框架，用过插件： easyswoole-permission 或者 easyswoole-hyperfOrm-permission 实现
Slim	一个 PHP 微型框架，通过插件帮助您快速实现简单但强大的 Web 应用程序和API，通过插件： casbin-with-slim 实现

名称	描述
Phalcon	以C-扩展形式发送的全堆栈PHP框架，通过插件: phalcon-permission 实现
Webman	High performance HTTP Service Framework for PHP based on Workerman, via plugin: webman-permission
名称	描述
Django	高级 Python Web 框架，通过插件: django-casbin 或 django-authority 即可实现
Flask	基于Werkzeug和Jinja的 Python 微型框架，通过插件 flask-authz , Flask-Casbin (第三方实现，或许更好) 或者 rbac-flask 。
FastAPI	使用 Python 3.6+ API的现代、快速(高性能)、网页框架，基于标准 Python 类型提示，通过插件: fastapi-authz 或者 Fastapi-app 实现
OpenStack	部署最广泛的开放源码云软件，通过插件: openstack-patron 实现
名称	描述
ASP.NET Core	一个开放源码和跨平台框架，用于建立以云为基础的现代互联网连接应用。例如Web应用、IoT应用和移动后端，通过插件: Casbin.AspNetCore
ASP.NET Core	通过插件在ASP.NET核心框架中使用Casbin的简单演示: CasbinACL-aspNetCore

名称	描述
Actix	Rust 框架, 通过插件 actix-casbin 实现
Actix web	一个小型、务实和快速的rust网络框架, 通过插件: actix-casbin-auth 实现
Rocket	为 Rust 设计的网页框架, 它使得在不牺牲灵活性、可用性或类型安全的情况下实现快速、安全的 web 应用程序, 通过插件: rocket-authz 或 rocket-casbin-auth 或 rocket-casbin-demo
Axum 网络	一个符合人体工程学和模块化的rust网络框架, 通过插件: actix-casbin-auth 实现
Poem 网页	通过插件 poem-casbin 便可使用Rust 编程语言的 web 框架, 其功能齐全、易于操作。
名称	描述
OpenResty	基于 NGINX 和 LuaJIT 的动态网络平台, 通过插件: lua-resty-casbin 和 casbin-openresty-example 实现
Kong	一个云原生、可伸展的 API 网关, 有着高性能和扩展性能, 通过插件: kong-authz 实现
APISIX	一个动态的、实时的高性能API网关, 通过插件: apisix-authz 实现
名称	描述
Vapor	服务器端的Swift web 框架, 通过插件: vapor-authz 实现

云提供商

Node.js

名称	描述
Okta	一个可信的平台通过插件保护身份: casbin-spring-boot-demo
Auth0	一个实现简单、可适应的认证和授权平台，通过插件: casbin-auth0-rbac 实现

Graphql-中间件

Casbin采用官方建议的方式为GraphQL终点提供授权，方法是采用单一的授权来源：
<https://graphql.org/learn/authorization/> 换句话说，Casbin应该放置在 GraphQL 层和您的业务逻辑之间。

```
// Casbin授权逻辑位于postRepository内
var postRepository = require('postRepository');

var postType = new GraphQLObjectType({
  name: 'Post',
  fields: {
    body: {
      type: GraphQLString,
      resolve: (post, args, context, { rootValue }) => {
        return postRepository.getBody(context.user, post);
      }
    }
  }
});
```

支持的 GraphQL 中间件

下面提供了Casbin GraphQL 中间件的完整列表。 我们欢迎来自第三方的中间件，请通知我们以将您的中间件加入列表中:)

[Go](#) [Node.js](#) [Python](#)

中间件	GraphQL 实现	作者	描述
graphql-authz	graphql	Casbin	用于graphql-go 的授权中间件
graphql-casbin	graphql	@esmaeilpour	使用 GraphQL 和 Casbin 共同实现
gqlgen_casbin_RBAC_example	gqlgen	@WenyXu	(empty)
中间件	GraphQL 实现	作者	描述
graphql-authz	GraphQL.js	Casbin	一个用于 GraphQL.js 的授权中间件
中间件	GraphQL 实现	作者	描述
graphql-authz	GraphQL-core 3	@Checho3388	一个用于 GraphQL.js 的授权中间件

云端原生中间件

云原生项目

[Go](#) [Node.js](#)

Project	作者	描述
k8s-authz	Casbin	授权 Kubernetes 的中间件
envoy-authz	Casbin	授权 Istio 和 Envoy 的中间件
kubesphere-authz	Casbin	授权 Kubernetes 的中间件

Project	作者	描述
ODPF Shield	Open Data Platform	ODPF Shield 是 Cloud Native 基于角色授权的逆向代理服务。



>

API

API

API概述

Casbin API 使用

管理 API

原始的 API 为 Casbin 策略管理提供了充分支持

RBAC API

用于RBAC的更友好的 API 此 API 是管理 API 中的一个子集。 RBAC 用户可以使用此 API 来简化代码。

域内基于角色的访问控制 API

一个更友好的域内基于角色的访问控制的API。 此 API 是管理 API 中的一个子集。 RBAC 用户可以使用此 API 来简化代码。

角色管理器API

RoleManager提供接口来定义管理角色的操作。 将匹配函数添加到角色名称和域名中使用通配符

数据权限

数据权限解决方案

API概述

这个概述只告诉你如何使用Casbin的API，并没有解释Casbin是如何安装和如何工作的。你可以在这里找到这些教程：[Casbin的安装](#)和[Casbin如何工作](#)。因此，当你开始阅读本教程时，我们假设你已经完全安装并将Casbin导入你的代码中。

用于强制执行的API(Enforce API)

让我们从Casbin的Enforce API开始。我们将从`model.conf`加载一个RBAC模型，并从`policy.csv`加载策略。你可以在这里[学习模型的语法](#)，在本教程中我们将不谈这个问题。我们假设你能理解下面给出的配置文件。

`model.conf`

```
[request_definition]
r = sub, obj, act

[policy_definition]
p = sub, obj, act

[role_definition]
g = _, _

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = g(r.sub, p.sub) && r.obj == p.obj && r.act == p.act
```

policy.csv

```
p, admin, data1, read
p, admin, data1, write
p, admin, data2, read
p, admin, data2, write
p, alice, data1, read
p, bob, data2, write
g, amber, admin
g, abc, admin
```

阅读完配置文件后，请阅读以下代码：

```
// 从文件中加载信息
enforcer, err := casbin.NewEnforcer("./example/model.conf",
"./example/policy.csv")
if err != nil {
    log.Fatalf("error, detail: %s", err)
}
ok, err := enforcer.Enforce("alice", "data1", "read")
```

这段代码从本地文件加载访问控制模型和策略。函数 `casbin.NewEnforcer()` 将返回一个执行者。它将识别其两个参数为文件路径，并从那里加载文件。过程中发生的错误被存储在 `err` 中。这段代码使用默认的适配器来加载模型和策略。当然，你也可以通过使用第三方适配器得到同样的结果。

代码 `ok, err := enforcer.Enforce("alice", "data1", "read")` 是为了确认访问权限。如果 `alice` 可以通过 `read` 操作访问 `data1`，返回值 `ok` 将是 `true`，否则将是 `false`。在这个例子中，`ok` 的值是 `true`。

EnforceEx API

有时你可能想知道哪个政策允许这个请求，所以我们准备了函数 `EnforceEx()`。你可以像这样使用它：

```
ok, reason, err := enforcer.EnforceEx("amber", "data1", "read")
fmt.Println(ok, reason) // true [admin data1 read]
```

函数 `EnforceEx()` 将在返回值 `reason` 中返回准确的策略字符串。在这个例子中，`amber` 是 `admin` 的一个角色，所以策略 `p, admin, data1, read` 使得这个请求 `true`。这段代码的输出在注释中。

Casbin准备了很多这样的API。这些API在基本的功能上增加了一些额外的功能。它们是：

- `ok, err := enforcer.EnforceWithMatcher(matcher, request)`

与一个匹配器。

- `ok, reason, err := enforcer.EnforceExWithMatcher(matcher, request)`

`EnforceWithMatcher()` 和 `EnforceEx()` 的组合。

- `boolArray, err := enforcer.BatchEnforce(requests)`

做一个列表工作(list job)，然后返回一个数组。

这是一个简单的Casbin使用案例。你可以使用Casbin通过这些API来启动一个授权服务器。我们将在接下来的段落中向你展示一些其他类型的API。

管理 API

Get API

这些API用于获取策略中的确切对象。这一次，我们像最后一个例子一样加载了一个enforcer 并从它中获得一些东西。

请阅读以下代码：

```
enforcer, err := casbin.NewEnforcer("./example/model.conf",
"./example/policy.csv")
if err != nil {
    fmt.Printf("Error, details: %s\n", err)
}
allSubjects := enforcer.GetAllSubjects()
fmt.Println(allSubjects)
```

就行上一个例子一样，前四行代码从本地文档中载入了一些必要的信息。我们将不再在这里谈这一点。

代码 `allSubjects := enforcer.GetAllSubjects()` 把策略文档里的所有 subjects 提取出来，并且将它们作为一个数组返回。然后我们打印这个数组

通常情况下，这段代码的输出会是这样：

```
[admin alice bob]
```

您也可以更改函数 `GetAllSubjects()` 到 `GetAllNamedSubjects()`，以获取当前命名策略中显示的 subjects 列表。

同样地，我们为 `Objects`, `Actions`, `Roles` 准备了 `GetAll` 函数。你唯一需要做的是，如果你想访问这些函数，将函数名称中的 `Subject` 一词改为你想要的。

同时，我们还有更多用于策略获取的 API。它们的调用方法和返回的值，都和上面提到的非常相似。

- `policy = e.GetPolicy()` 用于获取策略中的授权规则。
- `filteredPolicy := e.GetFilteredPolicy(0, "alice")` 用于获取策略中的授权的规则，可以自定义“域过滤器”(field filters)。
- `namedPolicy := e.GetNamedPolicy("p")` 用于获取被命名的策略中所有被授权的规则
- `filteredNamedPolicy = e.GetFilteredNamedPolicy("p", 0, "bob")` 用于获取被命名的策略中所有被授权的规则，可以自定义“域过滤器”(field filters)。
- `groupingPolicy := e.GetGroupingPolicy()` 用于获取策略中所有任务继承规则。
- `filteredGroupingPolicy := e.GetFilteredGroupingPolicy(0, "alice")` 用于获取策略中所有任务继承规则，可以自定义“域过滤器”(field filters)。
- `namedGroupingPolicy := e.GetNamedGroupingPolicy("g")` 用于获取策略中所有任务继承规则。
- `namedGroupingPolicy := e.GetFilteredNamedGroupingPolicy("g", 0, "alice")` 用于获取策略中所有任务继承规则。

添加, 删除, 更新 API

Casbin 为策略准备了很多 API。这些 API 允许您在运行时动态地添加、删除或编辑策略。

此代码向您展示了如何添加、删除和更新您的政策，并告诉您如何确认政策存在：

```
// 从文件中加载信息
```

使用这四种API可以编辑策略。像这些一样，我们为 `FilteredPolicy`, `NamedPolicy`, `FilteredNamedPolicy`, `GroupingPolicy`, `NamedGroupingPolicy`, `FilteredGroupingPolicy`, `FilteredNamedGroupingPolicy` 准备了同样种类的API。要使用它们，您只需要把函数名中的单词 `Policy` 替换为上面提到的单词。

此外，如果你将参数更改为数组，你可以批量编辑你的策略。

例如，对于这样的函数：

```
enforcer.UpdatePolicy([]string{"eve", "data3", "read"},  
[]string{"eve", "data3", "write"})
```

如果我们把 `Policy` 改为 `Policies`，并编辑参数为：

```
enforcer.UpdatePolicies([][]string{{"eve", "data3", "read"},  
{"jack", "data3", "read"}}, [][]string{{"eve", "data3",  
"write"}, {"jack", "data3", "write"}})
```

然后我们可以批量编辑这些策略。

同样的操作对 `GroupingPolicy`、`NamedGroupingPolicy` 也有用。

基于角色的访问控制接口

Casbin为你提供了来API来修改RBAC模型和策略。如果你熟悉RBAC，你可以很容易地使用这些API：

这里我们只告诉你如何使用Casbin的RBAC APIs，而不会谈论RBAC本身。您可以从[这里](#)获取详情。

我们使用这段代码来加载模型和政策，就像以前一样。

```
enforcer, err := casbin.NewEnforcer("./example/model.conf",
"./example/policy.csv")
if err != nil {
    fmt.Printf("Error, details: %s\n", err)
}
```

然后，使用Enforcer `enforcer` 来访问这些API。

```
roles, err := enforcer.GetRolesForUser("amber")
fmt.Println(roles) // [admin]
users, err := enforcer.GetUsersForRole("admin")
fmt.Println(users) // [amber abc]
```

`GetRolesForUser()` 返回一个数组，其中包含所有的角色。在这个例子中，`amber`只有一个管理员角色，所以数组 `roles` 是 `[admin]`。并且相似地，你可以用 `GetUsersForRole()`，来获得所有归属于这个角色的用户。此函数的返回值也是一个数组。

```
enforcer.HasRoleForUser("amber", "admin") // true
```

您可以使用 `HasRoleForUser()` 来确认用户是否属于该角色。在这个例子中，`amber`是管理员的成员，所以这个函数的返回值是 `true`。

```
fmt.Println(enforcer.Enforce("bob", "data2", "write")) // true
enforcer.DeletePermission("data2", "write")
fmt.Println(enforcer.Enforce("bob", "data2", "write")) // false
```

您可以使用 `DeletePermission()` 来删除权限。

```
fmt.Println(enforcer.Enforce("alice", "data1", "read")) // true  
enforcer.DeletePermissionForUser("alice", "data1", "read")  
fmt.Println(enforcer.Enforce("alice", "data1", "read")) // false
```

然后使用 `DeletePermissionForUser()` 来删除用户的权限。

Casbin有很多这样的API。它们的调用方式和返回值，都与上文提到的API有同样的样式。你可以在[随后的文档](#)中找到这些API。

管理 API

提供对Casbin策略管理完全支持的基本API。

筛选的 API

几乎所有的带有过滤器的api有着相同的参数 `(fieldIndex int, fieldValues ...string)`. `field index` 是匹配起始点的索引。 `field value` 表示结果应该有的值。请注意字段值中的空字符串可以是任意单词。

示例:

```
p, alice, book, read
p, bob, book, read
p, bob, book, write
p, alice, pen, get
p, bob, pen ,get
```

```
e.GetFilteredPolicy(1, "book") // 将返回: [Alice book read] [bob book
read] [bob book write]]]

e.etFilteredPolicy(1, "book", "read") // 将返回: [[Alice book read]
[bob book read]]]

e.etFilteredPolicy(0, "Alice", "", "read") // 将返回: [[Alice book
read]]]

e.GetFilteredPolicy(0, "Alice") // 将返回: [Alice book read] [Alice
penge]]]
```

参考

全局变量 `e` 是 Enforcer 实例。

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
e, err := NewEnforcer("examples/rbac_model.conf", "examples/
rbac_policy.csv")
```

```
const e = await newEnforcer('examples/rbac_model.conf', 'examples/
rbac_policy.csv')
```

```
$e = new Enforcer('examples/rbac_model.conf', 'examples/
rbac_policy.csv');
```

```
e = casbin.Enforcer("examples/rbac_model.conf", "examples/
rbac_policy.csv")
```

```
var e = new Enforcer("path/to/model.conf", "path/to/policy.csv");
```

```
let mut e = Enforce::new("examples/rbac_model.conf", "examples/
rbac_policy.csv").await?;
```

```
Enforcer e = new Enforcer("examples/rbac_model.conf", "examples/
rbac_policy.csv");
```

Enforce()

Enforce决定“主体”是否能够用“动作”操作访问“对象”，输入参数通常是: (sub, obj, act)。

例如:

```
ok, err := e.Enforce(request)

const ok = await e.enforce(request);

$ok = $e->enforcer($request);

ok = e.enforcer(request)

boolean ok = e.enforce(request);
```

EnforceWithMatcher()

EnforceWithMatcher使用自定义匹配器来决定“subject”是否可以通过“action”操作访问“object”，输入参数通常为：(matcher, sub, obj, act)，在匹配器为""时默认使用模型匹配器。

例如：

```
ok, err := e.EnforceWithMatcher(matcher, request)

$ok = $e->enforceWithMatcher($matcher, $request);

ok = e.enforce_with_matcher(matcher, request)

boolean ok = e.enforceWithMatcher(matcher, request);
```

EnforceEx()

EnforceEx 通过通知匹配的规则来解释执行

例如:

[Go](#) [Node.js](#) [PHP](#) [Python](#)

```
ok, reason, err := e.EnforceEx(request)

const ok = await e.enforceEx(request);

list($ok, $reason) = $e->enforceEx($request);

ok, reason = e.enforce_ex(request)
```

EnforceExWithMatcher()

EnforceExWEMatcher 使用自定义匹配器并通过通知匹配的规则来解释强制执行。

例如:

[Go](#)

```
ok, reason, err := e.EnforceExWithMatcher(matcher, request)
```

BatchEnforce()

BatchEnforce 强制执行每个请求并返回一个布尔数组的结果

例如:

Go Node.js Java

```
boolArray, err := e.BatchEnforce(requests)

const boolArray = await e.batchEnforce(requests);

List<Boolean> boolArray = e.batchEnforce(requests);
```

GetAllSubjects()

GetAllSubjects 获取当前策略中显示的主题列表。

例如:

Go Node.js PHP Python .NET Rust Java

```
allSubjects := e.GetAllSubjects()

const allSubjects = await e.getAllSubjects()

$allSubjects = $e->getAllSubjects();

all_subjects = e.get_all_subjects()

var allSubjects = e.GetAllSubjects();

let all_subjects = e.get_all_subjects();

List<String> allSubjects = e.getAllSubjects();
```

GetAllNamedSubjects()

GetAllNamedSubjects 获取当前命名策略中显示的主题列表。

例如:

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
allNamedSubjects := e.GetAllNamedSubjects("p")  
  
const allNamedSubjects = await e.getAllNamedSubjects('p')  
  
$allNamedSubjects = $e->getAllNamedSubjects("p");  
  
all_named_subjects = e.get_all_named_subjects("p")  
  
var allNamedSubjects = e.GetAllNamedSubjects("p");  
  
let all_named_subjects = e.get_all_named_subjects("p");  
  
List<String> allNamedSubjects = e.getAllNamedSubjects("p");
```

GetAllObjects()

GetAllObjects 获取当前策略中显示的对象列表。

例如:

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
allObjects := e.GetAllObjects()

const allObjects = await e.getAllObjects()

$allObjects = $e->getAllObjects();

all_objects = e.get_all_objects()

var allObjects = e.GetAllObjects();

let all_objects = e.get_all_objects();

List<String> allObjects = e.getAllObjects();
```

GetAllNamedObjects()

GetAllNamedObjects 获取当前命名策略中显示的对象列表。

例如:

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
allNamedObjects := e.GetAllNamedObjects("p")

const allNamedObjects = await e.getAllNamedObjects('p')

$allNamedObjects = $e->getAllNamedObjects("p");

all_named_objects = e.get_all_named_objects("p")

var allNamedObjects = e.GetAllNamedObjects("p");
```

```
let all_named_objects = e.get_all_named_objects("p");

List<String> allNamedObjects = e.getAllNamedObjects("p");
```

GetAllActions()

GetAllActions 获取当前策略中显示的操作列表。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
allActions := e.GetAllActions()

const allActions = await e.getAllActions()

$allActions = $e->getAllActions();

all_actions = e.get_all_actions()

var allActions = e.GetAllActions();

let all_actions = e.get_all_actions();

List<String> allActions = e.getAllActions();
```

GetAllNamedActions()

GetAllNamedActions 获取当前命名策略中显示的操作列表。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
allNamedActions := e.GetAllNamedActions("p")

const allNamedActions = await e.getAllNamedActions('p')

$allNamedActions = $e->getAllNamedActions("p");

all_named_actions = e.get_all_named_actions("p")

var allNamedActions = e.GetAllNamedActions("p");

let all_named_actions = e.get_all_named_actions("p");

List<String> allNamedActions = e.getAllNamedActions("p");
```

GetAllRoles()

GetAllRoles 获取当前策略中显示的角色列表。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
allRoles = e.GetAllRoles()

const allRoles = await e.getAllRoles()

$allRoles = $e->getAllRoles();
```

```
all_roles = e.get_all_roles()

var allRoles = e.GetAllRoles();

let all_roles = e.get_all_roles();

List<String> allRoles = e.getAllRoles();
```

GetAllNamedRoles()

GetAllNamedRoles 获取当前命名策略中显示的角色列表。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
allNamedRoles := e.GetAllNamedRoles("g")

const allNamedRoles = await e.getAllNamedRoles('g')

$allNamedRoles = $e->getAllNamedRoles('g');

all_named_roles = e.get_all_named_roles("g")

var allNamedRoles = e.GetAllNamedRoles("g");

let all_named_roles = e.get_all_named_roles("g");

List<String> allNamedRoles = e.getAllNamedRoles("g");
```

GetPolicy()

GetPolicy 获取策略中的所有授权规则。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
policy = e.GetPolicy()

const policy = await e.getPolicy()

$policy = $e->getPolicy();

policy = e.get_policy()

var policy = e.GetPolicy();

let policy = e.get_policy();

List<List<String>> policy = e.getPolicy();
```

GetFilteredPolicy()

GetFilteredPolicy 获取策略中的所有授权规则，可以指定字段筛选器。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
filteredPolicy := e.GetFilteredPolicy(0, "alice")

const filteredPolicy = await e.getFilteredPolicy(0, 'alice')

$filteredPolicy = $e->getFilteredPolicy(0, "alice");

filtered_policy = e.get_filtered_policy(0, "alice")

var filteredPolicy = e.GetFilteredPolicy(0, "alice");

let filtered_policy = e.get_filtered_policy(0,
    vec!["alice".to_owned()]);

List<List<String>> filteredPolicy = e.getFilteredPolicy(0, "alice");
```

GetNamedPolicy()

GetNamedPolicy 获取命名策略中的所有授权规则。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
namedPolicy := e.GetNamedPolicy("p")

const namedPolicy = await e.getNamedPolicy('p')

$namedPolicy = $e->getNamedPolicy("p");

named_policy = e.get_named_policy("p")
```

```
var namedPolicy = e.GetNamedPolicy("p");

let named_policy = e.get_named_policy("p");

List<List<String>> namedPolicy = e.getNamedPolicy("p");
```

GetFilteredNamedPolicy()

GetFilteredNamedPolicy 获取命名策略中的所有授权规则，可以指定字段过滤器。

例如：

Go Node.js PHP Python .NET Rust Java

```
filteredNamedPolicy = e.GetFilteredNamedPolicy("p", 0, "bob")

const filteredNamedPolicy = await e.getFilteredNamedPolicy('p', 0,
  'bob')

$filteredNamedPolicy = $e->getFilteredNamedPolicy("p", 0, "bob");

filtered_named_policy = e.get_filtered_named_policy("p", 0, "alice")

var filteredNamedPolicy = e.GetFilteredNamedPolicy("p", 0, "alice");

let filtered_named_policy = e.get_filtered_named_policy("p", 0,
  vec!["bob".to_owned()]);

List<List<String>> filteredNamedPolicy = e.getFilteredNamedPolicy("p",
  0, "bob");
```

GetGroupingPolicy()

GetGroupingPolicy 获取策略中的所有角色继承规则。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
groupingPolicy := e.GetGroupingPolicy()

const groupingPolicy = await e.getGroupingPolicy()

$groupingPolicy = $e->getGroupingPolicy();

grouping_policy = e.get_grouping_policy()

var groupingPolicy = e.GetGroupingPolicy();

let grouping_policy = e.get_grouping_policy();

List<List<String>> groupingPolicy = e.getGroupingPolicy();
```

GetFilteredGroupingPolicy()

GetFilteredGroupingPolicy 获取策略中的所有角色继承规则，可以指定字段筛选器。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
filteredGroupingPolicy := e.GetFilteredGroupingPolicy(0, "alice")

const filteredGroupingPolicy = await e.getFilteredGroupingPolicy(0,
  'alice')

$filteredGroupingPolicy = $e->getFilteredGroupingPolicy(0, "alice");

filtered_grouping_policy = e.get_filtered_grouping_policy(0, "alice")

var filteredGroupingPolicy = e.GetFilteredGroupingPolicy(0, "alice");

let filtered_grouping_policy = e.get_filtered_grouping_policy(0,
  vec!["alice".to_owned()]);
}

List<List<String>> filteredGroupingPolicy =
e.getFilteredGroupingPolicy(0, "alice");
```

GetNamedGroupingPolicy()

GetNamedGroupingPolicy 获取策略中的所有角色继承规则。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
namedGroupingPolicy := e.GetNamedGroupingPolicy("g")

const namedGroupingPolicy = await e.getNamedGroupingPolicy('g')

$namedGroupingPolicy = $e->getNamedGroupingPolicy("g");
```

```
named_grouping_policy = e.get_named_grouping_policy("g")

var namedGroupingPolicy = e.GetNamedGroupingPolicy("g");

let named_grouping_policy = e.get_named_grouping_policy("g");

List<List<String>> namedGroupingPolicy = e.getNamedGroupingPolicy("g");
```

GetFilteredNamedGroupingPolicy()

GetFilteredNamedGroupingPolicy 获取策略中的所有角色继承规则。

例如：

Go Node.js PHP Python .NET Rust Java

```
namedGroupingPolicy := e.GetFilteredNamedGroupingPolicy("g", 0,
"alice")

const namedGroupingPolicy = await
e.getFilteredNamedGroupingPolicy('g', 0, 'alice')

$namedGroupingPolicy = $e->getFilteredNamedGroupingPolicy("g", 0,
"alice");

named_grouping_policy = e.get_filtered_named_grouping_policy("g", 0,
"alice")

var namedGroupingPolicy = e.GetFilteredNamedGroupingPolicy("g", 0,
"alice");

let named_grouping_policy = e.get_filtered_named_groupingPolicy("g",
```

```
List<List<String>> filteredNamedGroupingPolicy =  
e.getFilteredNamedGroupingPolicy("g", 0, "alice");
```

HasPolicy()

HasPolicy 确定是否存在授权规则。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
hasPolicy := e.HasPolicy("data2_admin", "data2", "read")  
  
const hasPolicy = await e.hasPolicy('data2_admin', 'data2', 'read')  
  
$hasPolicy = $e->hasPolicy('data2_admin', 'data2', 'read');  
  
has_policy = e.has_policy("data2_admin", "data2", "read")  
  
var hasPolicy = e.HasPolicy("data2_admin", "data2", "read");  
  
let has_policy = e.has_policy(vec!["data2_admin".to_owned(),  
"data2".to_owned(), "read".to_owned()]);  
  
boolean hasPolicy = e.hasPolicy("data2_admin", "data2", "read");
```

HasNamedPolicy()

HasNamedPolicy 确定是否存在命名授权规则。

例如：

```
hasNamedPolicy := e.HasNamedPolicy("p", "data2_admin", "data2", "read")

const hasNamedPolicy = await e.hasNamedPolicy('p', 'data2_admin',
    'data2', 'read')

$hasNamedPolicy = $e->hasNamedPolicy("p", "data2_admin", "data2",
    "read");

has_named_policy = e.has_named_policy("p", "data2_admin", "data2",
    "read");

var hasNamedPolicy = e.HasNamedPolicy("p", "data2_admin", "data2",
    "read");

let has_named_policy = e.has_named_policy("p",
    vec!["data2_admin".to_owned(), "data2".to_owned(), "read".to_owned()]);

boolean hasNamedPolicy = e.hasNamedPolicy("p", "data2_admin", "data2",
    "read");
```

AddPolicy()

AddPolicy 向当前策略添加授权规则。如果规则已经存在，函数返回false，并且不会添加规则。否则，函数通过添加新规则并返回true。

例如：

```
added := e.AddPolicy('eve', 'data3', 'read')

const p = ['eve', 'data3', 'read']
const added = await e.addPolicy(...p)

$added = $e->addPolicy('eve', 'data3', 'read');

added = e.add_policy("eve", "data3", "read")

var added = e.AddPolicy("eve", "data3", "read");
or
var added = await e.AddPolicyAsync("eve", "data3", "read");

let added = e.add_policy(vec!["eve".to_owned(), "data3".to_owned(),
"read".to_owned()]);

boolean added = e.addPolicy("eve", "data3", "read");
```

AddPolicies()

AddPolicy 向当前策略添加授权许多规则。该操作本质上是原子的。因此，如果授权规则由不符合现行政策的规则组成，函数返回false，当前政策中没有添加任何政策规则。如果所有授权规则都符合政策规则，则函数返回true，每项政策规则都被添加到目前的政策中。

例如：

Go Node.js Python Rust Java

```
rules := [][] string {
    []string {"jack", "data4", "read"},
    []string {"katy", "data4", "write"},
    []string {"leyo", "data4", "read"},
```

```

const rules = [
    ['jack', 'data4', 'read'],
    ['katy', 'data4', 'write'],
    ['leyo', 'data4', 'read'],
    ['ham', 'data4', 'write']
];

const areRulesAdded = await e.addPolicies(rules);

rules = [
    ["jack", "data4", "read"],
    ["katy", "data4", "write"],
    ["leyo", "data4", "read"],
    ["ham", "data4", "write"]
]
are_rules_added = e.add_policies(rules)

let rules = vec![
    vec!["jack".to_owned(), "data4".to_owned(), "read".to_owned()],
    vec!["katy".to_owned(), "data4".to_owned(), "write".to_owned()],
    vec!["leyo".to_owned(), "data4".to_owned(), "read".to_owned()],
    vec!["ham".to_owned(), "data4".to_owned(), "write".to_owned()],
];
let are_rules_added = e.add_policies(rules).await?

String[][] rules = {
    {"jack", "data4", "read"},
    {"katy", "data4", "write"},
    {"leyo", "data4", "read"},
    {"ham", "data4", "write"},
};

boolean areRulesAdded = e.addPolicies(rules);

```

AddNamedPolicy()

AddNamedPolicy 向当前命名策略添加授权规则。如果规则已经存在，函数返回false，并且不会添加规则。否则，函数通过添加新规则并返回true。

例如：

Go Node.js PHP Python .NET Rust Java

```
added := e.AddNamedPolicy("p", "eve", "data3", "read")

const p = ['eve', 'data3', 'read']
const added = await e.addNamedPolicy('p', ...p)

$added = $e->addNamedPolicy("p", "eve", "data3", "read");

added = e.add_named_policy("p", "eve", "data3", "read")

var added = e.AddNamedPolicy("p", "eve", "data3", "read");
or
var added = await e.AddNamedPolicyAsync("p", "eve", "data3", "read");

let added = e.add_named_policy("p", vec!["eve".to_owned(),
"data3".to_owned(), "read".to_owned()]).await?;

boolean added = e.addNamedPolicy("p", "eve", "data3", "read");
```

AddNamedPolicies()

AddNamedPolicies 向当前命名策略中添加授权规则。该操作本质上是原子的。因此，如果授权规则由不符合现行政策的规则组成，函数返回false，当前政策中没有添加任何政策规则。如果所有授权规则都符合政策规则，则函数返回true，每项政策规则都被添加到目前的政策中。

例如：

Go Node.js Python Rust Java

```
rules := [][] string {
    []string {"jack", "data4", "read"},
    []string {"katy", "data4", "write"},
    []string {"leyo", "data4", "read"},
    []string {"ham", "data4", "write"},
}

areRulesAdded := e.AddNamedPolicies("p", rules)

const rules = [
    ['jack', 'data4', 'read'],
    ['katy', 'data4', 'write'],
    ['leyo', 'data4', 'read'],
    ['ham', 'data4', 'write']
];

const areRulesAdded = await e.addNamedPolicies('p', rules);

rules = [
    ["jack", "data4", "read"],
    ["katy", "data4", "write"],
    ["leyo", "data4", "read"],
    ["ham", "data4", "write"]
]
are_rules_added = e.add_named_policies("p", rules)

let rules = vec![
    vec!["jack".to_owned(), "data4".to_owned(), "read".to_owned()],
    vec!["katy".to_owned(), "data4".to_owned(), "write".to_owned()],
    vec!["leyo".to_owned(), "data4".to_owned(), "read".to_owned()],
    vec!["ham".to_owned(), "data4".to_owned(), "write".to_owned()],
];
```

```
List<List<String>> rules = Arrays.asList(
    Arrays.asList("jack", "data4", "read"),
    Arrays.asList("katy", "data4", "write"),
    Arrays.asList("leyo", "data4", "read"),
    Arrays.asList("ham", "data4", "write")
);
boolean areRulesAdded = e.addNamedPolicies("p", rules);
```

RemovePolicy()

RemovePolicy 从当前策略中删除授权规则。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
removed := e.RemovePolicy("alice", "data1", "read")

const p = ['alice', 'data1', 'read']
const removed = await e.removePolicy(...p)

$removed = $e->removePolicy("alice", "data1", "read");

removed = e.remove_policy("alice", "data1", "read")

var removed = e.RemovePolicy("alice", "data1", "read");
or
var removed = await e.RemovePolicyAsync("alice", "data1", "read");

let removed = e.remove_policy(vec!["alice".to_owned(),
"data1".to_owned(), "read".to_owned()]).await?;
```

```
boolean removed = e.removePolicy("alice", "data1", "read");
```

RemovePolicies()

RemovePolicies 从当前策略中删除授权规则。该操作本质上是原子的。因此，如果授权规则由不符合现行政策的规则组成，函数返回 `false`，当前政策中没有任何政策规则被删除。如果所有授权规则都符合政策规则，则函数返回`true`，每项政策规则都从现行政策中删除。

例如：

[Go](#) [Node.js](#) [Python](#) [Rust](#) [Java](#)

```
rules := [][] string {
    []string {"jack", "data4", "read"},
    []string {"katy", "data4", "write"},
    []string {"leyo", "data4", "read"},
    []string {"ham", "data4", "write"},
}

areRulesRemoved := e.RemovePolicies(rules)

const rules = [
    ['jack', 'data4', 'read'],
    ['katy', 'data4', 'write'],
    ['leyo', 'data4', 'read'],
    ['ham', 'data4', 'write']
];

const areRulesRemoved = await e.removePolicies(rules);

rules = [
    ["jack", "data4", "read"],
    ["katy", "data4", "write"],
    ["leyo", "data4", "read"],
```

```

let rules = vec![
    vec!["jack".to_owned(), "data4".to_owned(), "read".to_owned()],
    vec!["katy".to_owned(), "data4".to_owned(), "write".to_owned()],
    vec!["leyo".to_owned(), "data4".to_owned(), "read".to_owned()],
    vec!["ham".to_owned(), "data4".to_owned(), "write".to_owned()],
];
let are_rules_removed = e.remove_policies(rules).await?;

String[][] rules = {
    {"jack", "data4", "read"},
    {"katy", "data4", "write"},
    {"leyo", "data4", "read"},
    {"ham", "data4", "write"},
};
boolean areRulesRemoved = e.removePolicies(rules);

```

RemoveFilteredPolicy()

RemoveFilteredPolicy 移除当前策略中的授权规则，可以指定字段筛选器。 RemovePolicy 从当前策略中删除授权规则。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
removed := e.RemoveFilteredPolicy(0, "alice", "data1", "read")
```

```
const p = ['alice', 'data1', 'read']
const removed = await e.removeFilteredPolicy(0, ...p)
```

```
$removed = $e->removeFilteredPolicy(0, "alice", "data1", "read");
```

```
removed = e.remove_filtered_policy(0, "alice", "data1", "read")

var removed = e.RemoveFilteredPolicy("alice", "data1", "read");
or
var removed = await e.RemoveFilteredPolicyAsync("alice", "data1",
"read");

let removed = e.remove_filtered_policy(0, vec!["alice".to_owned(),
"data1".to_owned(), "read".to_owned()]).await?;

boolean removed = e.removeFilteredPolicy(0, "alice", "data1", "read");
```

RemoveNamedPolicy()

RemoveNamedPolicy 从当前命名策略中删除授权规则。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
removed := e.RemoveNamedPolicy("p", "alice", "data1", "read")

const p = ['alice', 'data1', 'read']
const removed = await e.removeNamedPolicy('p', ...p)

$removed = $e->removeNamedPolicy("p", "alice", "data1", "read");

removed = e.remove_named_policy("p", "alice", "data1", "read")

var removed = e.RemoveNamedPolicy("p", "alice", "data1", "read");
or
var removed = await e.RemoveNamedPolicyAsync("p", "alice", "data1",
```

```
let removed = e.remove_named_policy("p", vec!["alice".to_owned(),
"data1".to_owned(), "read".to_owned()]).await?;

boolean removed = e.removeNamedPolicy("p", "alice", "data1", "read");
```

RemoveNamedPolicies()

RemoveNamedPolicies 从当前命名策略中删除授权规则。该操作本质上是原子的。因此，如果授权规则由不符合现行政策的规则组成，函数返回 `false`，当前政策中没有任何政策规则被删除。如果所有授权规则都符合政策规则，则函数返回`true`，每项政策规则都从现行政策中删除。

例如：

[Go](#) [Node.js](#) [Python](#) [Rust](#) [Java](#)

```
rules := [][] string {
    []string {"jack", "data4", "read"},
    []string {"katy", "data4", "write"},
    []string {"leyo", "data4", "read"},
    []string {"ham", "data4", "write"},
}

areRulesRemoved := e.RemoveNamedPolicies("p", rules)

const rules = [
    ['jack', 'data4', 'read'],
    ['katy', 'data4', 'write'],
    ['leyo', 'data4', 'read'],
    ['ham', 'data4', 'write']
];

const areRulesRemoved = await e.removeNamedPolicies('p', rules);
```

```

rules = [
    ["jack", "data4", "read"],
    ["katy", "data4", "write"],
    ["leyo", "data4", "read"],
    ["ham", "data4", "write"]
]
are_rules_removed = e.remove_named_policies("p", rules)

let rules = vec![
    vec!["jack".to_owned(), "data4".to_owned(), "read".to_owned()],
    vec!["katy".to_owned(), "data4".to_owned(), "write".to_owned()],
    vec!["leyo".to_owned(), "data4".to_owned(), "read".to_owned()],
    vec!["ham".to_owned(), "data4".to_owned(), "write".to_owned()],
];
let areRulesRemoved = e.remove_named_policies("p", rules).await?;

List<List<String>> rules = Arrays.asList(
    Arrays.asList("jack", "data4", "read"),
    Arrays.asList("katy", "data4", "write"),
    Arrays.asList("leyo", "data4", "read"),
    Arrays.asList("ham", "data4", "write")
);
boolean areRulesRemoved = e.removeNamedPolicies("p", rules);

```

RemoveFilteredNamedPolicy()

RemoveFilteredNamedPolicy 从当前命名策略中移除授权规则，可以指定字段筛选器。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
removed := e.RemoveFilteredNamedPolicy("p", 0, "alice", "data1",
```

```
const p = ['alice', 'data1', 'read']
const removed = await e.removeFilteredNamedPolicy('p', 0, ...p)

$removed = $e->removeFilteredNamedPolicy("p", 0, "alice", "data1",
"read");

removed = e.remove_filtered_named_policy("p", 0, "alice", "data1",
"read")

var removed = e.RemoveFilteredNamedPolicy("p", 0, "alice", "data1",
"read");
or
var removed = e.RemoveFilteredNamedPolicyAsync("p", 0, "alice",
"data1", "read");

let removed = e.remove_filtered_named_policy("p", 0,
vec!["alice".to_owned(), "data1".to_owned(),
"read".to_owned()]).await?;

boolean removed = e.removeFilteredNamedPolicy("p", 0, "alice",
"data1", "read");
```

HasGroupingPolicy()

HasGroupingPolicy 确定是否存在角色继承规则。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
has := e.HasGroupingPolicy("alice", "data2_admin")
```

```
const has = await e.hasGroupingPolicy('alice', 'data2_admin')

$has = $e->hasGroupingPolicy("alice", "data2_admin");

has = e.has_grouping_policy("alice", "data2_admin")

var has = e.HasGroupingPolicy("alice", "data2_admin");

let has = e.has_grouping_policy(vec!["alice".to_owned(),
"data2_admin".to_owned()]);

boolean has = e.hasGroupingPolicy("alice", "data2_admin");
```

HasNamedGroupingPolicy()

HasNamedGroupingPolicy 确定是否存在命名角色继承规则。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
has := e.HasNamedGroupingPolicy("g", "alice", "data2_admin")

const has = await e.hasNamedGroupingPolicy('g', 'alice', 'data2_admin')

$has = $e->hasNamedGroupingPolicy("g", "alice", "data2_admin");

has = e.has_named_grouping_policy("g", "alice", "data2_admin")

var has = e.HasNamedGroupingPolicy("g", "alice", "data2_admin");
```

```
let has = e.has_named_grouping_policy("g", vec!["alice".to_owned(),
"data2_admin".to_owned()]);
```

```
boolean has = e.hasNamedGroupingPolicy("g", "alice", "data2_admin");
```

AddGroupingPolicy()

AddGroupingPolicy 向当前策略添加角色继承规则。如果规则已经存在，函数返回false，并且不会添加规则。否则，函数通过添加新规则并返回true。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
added := e.AddGroupingPolicy("group1", "data2_admin")

const added = await e.addGroupingPolicy('group1', 'data2_admin')

$added = $e->addGroupingPolicy("group1", "data2_admin");

added = e.add_grouping_policy("group1", "data2_admin")

var added = e.AddGroupingPolicy("group1", "data2_admin");
or
var added = await e.AddGroupingPolicyAsync("group1", "data2_admin");

let added = e.add_grouping_policy(vec!["group1".to_owned(),
"data2_admin".to_owned()]).await?;

boolean added = e.addGroupingPolicy("group1", "data2_admin");
```

AddGroupingPolicies()

AddGroupingPolicy 向当前策略添加角色继承规则。该操作本质上是原子的。因此，如果授权规则由不符合现行政策的规则组成，函数返回false，当前政策中没有添加任何政策规则。如果所有授权规则都符合政策规则，则函数返回true，每项政策规则都被添加到目前的政策中。

例如：

Go Node.js Python Rust Java

```
rules := [][] string {
    []string {"ham", "data4_admin"},
    []string {"jack", "data5_admin"},
}

areRulesAdded := e.AddGroupingPolicies(rules)

const groupingRules = [
    ['ham', 'data4_admin'],
    ['jack', 'data5_admin']
];

const areRulesAdded = await e.addGroupingPolicies(groupingRules);

rules = [
    ["ham", "data4_admin"],
    ["jack", "data5_admin"]
]

are_rules_added = e.add_grouping_policies(rules)

let rules = vec![
    vec!["ham".to_owned(), "data4_admin".to_owned()],
    vec!["jack".to_owned(), "data5_admin".to_owned()],
];
```

```
String[][] groupingRules = {
    {"ham", "data4_admin"},
    {"jack", "data5_admin"}
};
boolean areRulesAdded = e.addGroupingPolicies(groupingRules);
```

AddNamedGroupingPolicy()

AddNamedGroupingPolicy 将命名角色继承规则添加到当前策略。如果规则已经存在，函数返回false，并且不会添加规则。否则，函数通过添加新规则并返回true。

例如：

Go Node.js PHP Python .NET Rust Java

```
added := e.AddNamedGroupingPolicy("g", "group1", "data2_admin")

const added = await e.addNamedGroupingPolicy('g', 'group1',
    'data2_admin')

$added = $e->addNamedGroupingPolicy("g", "group1", "data2_admin");

added = e.add_named_grouping_policy("g", "group1", "data2_admin")

var added = e.AddNamedGroupingPolicy("g", "group1", "data2_admin");
or
var added = await e.AddNamedGroupingPolicyAsync("g", "group1",
    "data2_admin");

let added = e.add_named_grouping_policy("g", vec!["group1".to_owned(),
    "data2_admin".to_owned()]).await?;
```

```
boolean added = e.addNamedGroupingPolicy("g", "group1", "data2_admin");
```

AddNamedGroupingPolicies()

AddNamedGroupingPolicies 将命名角色继承规则添加到当前策略。该操作本质上是原子的。因此，如果授权规则由不符合现行政策的规则组成，函数返回false，当前政策中没有添加任何政策规则。如果所有授权规则都符合政策规则，则函数返回true，每项政策规则都被添加到目前的政策中。

例如：

[Go](#) [Node.js](#) [Python](#) [Rust](#) [Java](#)

```
rules := [][] string {
    []string {"ham", "data4_admin"},
    []string {"jack", "data5_admin"},
}

areRulesAdded := e.AddNamedGroupingPolicies("g", rules)

const groupingRules = [
    ['ham', 'data4_admin'],
    ['jack', 'data5_admin']
];

const areRulesAdded = await e.addNamedGroupingPolicies('g',
groupingRules);

rules = [
    ["ham", "data4_admin"],
    ["jack", "data5_admin"]
]

are_rules_added = e.add_named_grouping_policies("g", rules)
```

```
let rules = vec![
    vec!["ham".to_owned(), "data4_admin".to_owned()],
    vec!["jack".to_owned(), "data5_admin".to_owned()],
];

let are_rules_added = e.add_named_grouping_policies("g", rules).await?;

String[][] groupingRules = {
    {"ham", "data4_admin"},
    {"jack", "data5_admin"}
};
boolean areRulesAdded = e.addNamedGroupingPolicies("g", groupingRules);
```

RemoveGroupingPolicy()

RemoveGroupingPolicy 从当前策略中删除角色继承规则。

例如：

Go Node.js PHP Python .NET Rust Java

```
removed := e.RemoveGroupingPolicy("alice", "data2_admin")

const removed = await e.removeGroupingPolicy('alice', 'data2_admin')

$removed = $e->removeGroupingPolicy("alice", "data2_admin");

removed = e.remove_grouping_policy("alice", "data2_admin")

var removed = e.RemoveGroupingPolicy("alice", "data2_admin");
or
var removed = await e.RemoveGroupingPolicyAsync("alice",
"data2_admin");
```

```
let removed = e.remove_grouping_policy(vec!["alice".to_owned(),
"data2_admin".to_owned()]).await?;

boolean removed = e.removeGroupingPolicy("alice", "data2_admin");
```

RemoveGroupingPolicies()

RemoveGroupingPolicy 从当前策略中删除角色继承规则。该操作本质上是原子的。因此，如果授权规则由不符合现行政策的规则组成，函数返回 `false`，当前政策中没有任何政策规则被删除。如果所有授权规则都符合政策规则，则函数返回`true`，每项政策规则都从现行政策中删除。

例如：

[Go](#) [Node.js](#) [Rust](#) [Python](#) [Java](#)

```
rules := [][] string {
    []string {"ham", "data4_admin"},
    []string {"jack", "data5_admin"},
}

areRulesRemoved := e.RemoveGroupingPolicies(rules)

const groupingRules = [
    ['ham', 'data4_admin'],
    ['jack', 'data5_admin']
];

const areRulesRemoved = await e.removeGroupingPolicies(groupingRules);

let rules = vec![
    vec!["ham".to_owned(), "data4_admin".to_owned()],
    vec!["jack".to_owned(), "data5_admin".to_owned()],
];
```

```
rules = [
    ["ham", "data4_admin"],
    ["jack", "data5_admin"]
]

are_rules_removed = e.remove_grouping_policies(rules)

String[][] groupingRules = {
    {"ham", "data4_admin"},
    {"jack", "data5_admin"}
};
boolean areRulesRemoved = e.removeGroupingPolicies(groupingRules);
```

RemoveFilteredGroupingPolicy()

RemoveFilteredGroupingPolicy 从当前策略中移除角色继承规则，可以指定字段筛选器。

例如：

Go Node.js PHP Python .NET Rust Java

```
removed := e.RemoveFilteredGroupingPolicy(0, "alice")

const removed = await e.removeFilteredGroupingPolicy(0, 'alice')

$removed = $e->removeFilteredGroupingPolicy(0, "alice");

removed = e.remove_filtered_grouping_policy(0, "alice")

var removed = e.RemoveFilteredGroupingPolicy(0, "alice");
or
var removed = await e.RemoveFilteredGroupingPolicyAsync(0, "alice");
```

```
let removed = e.remove_filtered_grouping_policy(0,
vec!["alice".to_owned()]).await?;

boolean removed = e.removeFilteredGroupingPolicy(0, "alice");
```

RemoveNamedGroupingPolicy()

RemoveNamedGroupingPolicy 从当前命名策略中移除角色继承规则。

例如：

Go Node.js PHP Python .NET Rust Java

```
removed := e.RemoveNamedGroupingPolicy("g", "alice")

const removed = await e.removeNamedGroupingPolicy('g', 'alice')

$removed = $e->removeNamedGroupingPolicy("g", "alice");

removed = e.remove_named_grouping_policy("g", "alice", "data2_admin")

var removed = e.RemoveNamedGroupingPolicy("g", "alice");
or
var removed = await e.RemoveNamedGroupingPolicyAsync("g", "alice");

let removed = e.remove_named_grouping_policy("g",
vec!["alice".to_owned()]).await?;

boolean removed = e.removeNamedGroupingPolicy("g", "alice");
```

RemoveNamedGroupingPolicies()

RemoveNamedGroupingPolicies 从当前策略中删除角色继承规则。该操作本质上是原子的。因此，如果授权规则由不符合现行政策的规则组成，函数返回 `false`，当前政策中没有任何政策规则被删除。如果所有授权规则都符合政策规则，则函数返回`true`，每项政策规则都从现行政策中删除。

例如：

[Go](#) [Node.js](#) [Python](#) [Rust](#) [Java](#)

```
rules := [][] string {
    []string {"ham", "data4_admin"},
    []string {"jack", "data5_admin"},
}

areRulesRemoved := e.RemoveNamedGroupingPolicies("g", rules)

const groupingRules = [
    ['ham', 'data4_admin'],
    ['jack', 'data5_admin']
];

const areRulesRemoved = await e.removeNamedGroupingPolicies('g',
groupingRules);

rules = [
    ["ham", "data4_admin"],
    ["jack", "data5_admin"]
]
are_rules_removed = e.remove_named_grouping_policies("g", rules)

let rules = vec![
    vec!["ham".to_owned(), "data4_admin".to_owned()],

```

```
String[][] groupingRules = {
    {"ham", "data4_admin"},
    {"jack", "data5_admin"}
};
boolean areRulesRemoved = e.removeNamedGroupingPolicies("g",
groupingRules);
```

RemoveFilteredNamedGroupingPolicy()

RemoveFilteredNamedGroupingPolicy 从当前命名策略中移除角色继承规则，可以指定字段筛选器。

例如：

Go Node.js PHP Python .NET Rust Java

```
removed := e.RemoveFilteredNamedGroupingPolicy("g", 0, "alice")

const removed = await e.removeFilteredNamedGroupingPolicy('g', 0,
'alice')

$removed = $e->removeFilteredNamedGroupingPolicy("g", 0, "alice");

removed = e.remove_filtered_named_grouping_policy("g", 0, "alice")

var removed = e.RemoveFilteredNamedGroupingPolicy("g", 0, "alice");
or
var removed = await e.RemoveFilteredNamedGroupingPolicyAsync("g", 0,
"alice");

let removed = e.remove_filtered_named_groupingPolicy("g", 0,
vec!["alice"].to_owned()).await?;
```

```
boolean removed = e.removeFilteredNamedGroupingPolicy("g", 0, "alice");
```

UpdatePolicy()

UpdatePolicy 把旧的政策更新到新的政策

例如：

[Go](#) [Node.js](#) [Python](#) [Java](#)

```
updated, err := e.UpdatePolicy([]string{"eve", "data3", "read"},  
[]string{"eve", "data3", "write"})  
  
const update = await e.updatePolicy(["eve", "data3", "read"], ["eve",  
"data3", "write"]);  
  
updated = e.update_policy(["eve", "data3", "read"], ["eve", "data3",  
"write"])  
  
boolean updated = e.updatePolicy(Arrays.asList("eve", "data3",  
"read"), Arrays.asList("eve", "data3", "write"));
```

UpdatePolicies()

UpdatePolicies 将所有的旧政策更新到新政策

例如：

[Go](#) [Python](#)

```
updated, err := e.UpdatePolicies([][]string{{"eve", "data3", "read"},  
{"jack", "data3", "read"}}, [][]string{{"eve", "data3", "write"},  
{"jack", "data3", "write"}})  
  
old_rules = [["eve", "data3", "read"], ["jack", "data3", "read"]]  
new_rules = [["eve", "data3", "write"], ["jack", "data3", "write"]]  
  
updated = e.update_policies(old_rules, new_rules)
```

AddFunction()

AddFunction 添加自定义函数。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [Rust](#) [Java](#)

```
func CustomFunction(key1 string, key2 string) bool {  
    if key1 == "/alice_data2/myid/using/res_id" && key2 ==  
"/alice_data/:resource" {  
        return true  
    } else if key1 == "/alice_data2/myid/using/res_id" && key2 ==  
"/alice_data2/:id/using/:resId" {  
        return true  
    } else {  
        return false  
    }  
}  
  
func CustomFunctionWrapper(args ...interface{}) (interface{}, error) {  
    key1 := args[0].(string)  
    key2 := args[1].(string)  
  
    return bool(CustomFunction(key1, key2)), nil  
}
```

```

function customFunction(key1, key2){
    if(key1 == "/alice_data2/myid/using/res_id" && key2 ==
"/alice_data/:resource") {
        return true
    } else if(key1 == "/alice_data2/myid/using/res_id" && key2 ==
"/alice_data2/:id/using/:resId") {
        return true
    } else {
        return false
    }
}

e.addFunction("keyMatchCustom", customFunction);

func customFunction($key1, $key2) {
    if ($key1 == "/alice_data2/myid/using/res_id" && $key2 ==
"/alice_data/:resource") {
        return true;
    } elseif ($key1 == "/alice_data2/myid/using/res_id" && $key2 ==
"/alice_data2/:id/using/:resId") {
        return true;
    } else {
        return false;
    }
}

func customFunctionWrapper(...$args){
    $key1 := $args[0];
    $key2 := $args[1];

    return customFunction($key1, $key2);
}

$e->addFunction("keyMatchCustom", customFunctionWrapper);

def custom_function(key1, key2):
    return ((key1 == "/alice_data2/myid/using/res_id" and key2 ==
"/alice_data/:resource") or (key1 == "/alice_data2/myid/using/res_id"

```

```

fn custom_function(key1: SString, key2: String) {
    key1 == "/alice_data2/myid/using/res_id" && key2 ==
"/alice_data/:resource" || key1 == "/alice_data2/myid/using/res_id" &&
key2 == "/alice_data2/:id/using/:resId"
}

e.add_function("keyMatchCustom", custom_function);

public static class CustomFunc extends CustomFunction {
    @Override
    public AviatorObject call(Map<String, Object> env, AviatorObject
arg1, AviatorObject arg2) {
        String key1 = FunctionUtils.getStringValue(arg1, env);
        String key2 = FunctionUtils.getStringValue(arg2, env);
        if (key1.equals("/alice_data2/myid/using/res_id") &&
key2.equals("/alice_data/:resource")) {
            return AviatorBoolean.valueOf(true);
        } else if (key1.equals("/alice_data2/myid/using/res_id") &&
key2.equals("/alice_data2/:id/using/:resId")) {
            return AviatorBoolean.valueOf(true);
        } else {
            return AviatorBoolean.valueOf(false);
        }
    }

    @Override
    public String getName() {
        return "keyMatchCustom";
    }
}
}

FunctionTest.CustomFunc customFunc = new FunctionTest.CustomFunc();
e.addFunction(customFunc.getName(), customFunc);

```

LoadFilteredPolicy()

LoadFilteredPolicy从文件/数据库加载过滤完成的政策

例如：

[Go](#) [Node.js](#) [Python](#) [Java](#)

```
err := e.LoadFilteredPolicy()

const ok = await e.loadFilteredPolicy();

class Filter:
    P = []
    G = []

adapter =
casbin.persist.adapters.FilteredAdapter("rbac_with_domains_policy.csv")
e = casbin.Enforcer("rbac_with_domains_model.conf", adapter)
filter = Filter()
filter.P = ["", "domain1"]
filter.G = ["", "", "domain1"]
e.load_filtered_policy(filter)

e.loadFilteredPolicy(new String[] { "", "domain1" });
```

LoadIncrementalFilteredPolicy()

LoadFilteredPolicy从文件/数据库添加过滤完成的政策

例如：

[Go](#) [Node.js](#) [Python](#)

```
err := e.LoadIncrementalFilteredPolicy()

const ok = await e.loadIncrementalFilteredPolicy();

adapter =
casbin.persist.adapters.FilteredAdapter("rbac_with_domains_policy.csv")
```

UpdateGroupingPolicy()

UpdateGroupingPolicy 在 g 段更新oldRule到newRule

例如：

[Go](#) [Java](#)

```
succeed, err := e.UpdateGroupingPolicy([]string{"data3_admin",  
"data4_admin"}, []string{"admin", "data4_admin"})  
  
boolean succeed = e.updateGroupingPolicy(Arrays.asList("data3_admin",  
"data4_admin"), Arrays.asList("admin", "data4_admin"));
```

UpdateNamedGroupingPolicy()

UpdateNamedGroupingPolicy 在 g 部分将名为 ptype 的旧策略更新为新策略

例如：

[Go](#) [Java](#)

```
succeed, err := e.UpdateGroupingPolicy("g1", []string{"data3_admin",  
"data4_admin"}, []string{"admin", "data4_admin"})  
  
boolean succeed = e.updateNamedGroupingPolicy("g1",  
Arrays.asList("data3_admin", "data4_admin"), Arrays.asList("admin",  
"data4_admin"));
```

SetFieldIndex()

设置FieldIndex 子端口自定义常规名称和位置 子类, obj, 域 和 优先级.

```
[policy_definition]
p = customized_priority, obj, act, eft, subject
```

例如:

Go

```
e.SetFieldIndex("p", constant.PriorityIndex, 0)
e.SetFieldIndex("p", constant.SubjectIndex, 4)
```

RBAC API

一个更友好的RBAC API。这个API是Management API的子集。RBAC用户可以使用这个API来简化代码。

参考

全局变量 `e` 是 Enforcer 实例。

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
e, err := NewEnforcer("examples/rbac_model.conf", "examples/rbac_policy.csv")
```

```
const e = await newEnforcer('examples/rbac_model.conf', 'examples/rbac_policy.csv')
```

```
$e = new Enforcer('examples/rbac_model.conf', 'examples/rbac_policy.csv');
```

```
e = casbin.Enforcer("examples/rbac_model.conf", "examples/rbac_policy.csv")
```

```
var e = new Enforcer("path/to/model.conf", "path/to/policy.csv");
```

```
let mut e = Enforcer::new("examples/rbac_model.conf", "examples/
```

```
Enforcer e = new Enforcer("examples/rbac_model.conf", "examples/rbac_policy.csv");
```

GetRolesForUser()

GetRolesForUser 获取用户具有的角色。

例如:

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

[Go](#)

```
res := e.GetRolesForUser("alice")
```

[Node.js](#)

```
const res = await e.getRolesForUser('alice')
```

[PHP](#)

```
$res = $e->getRolesForUser("alice");
```

[Python](#)

```
roles = e.get_roles_for_user("alice")
```

[.NET](#)

```
var res = e.GetRolesForUser("alice");
```

[Rust](#)

```
let roles = e.get_roles_for_user("alice", None); // 无域
```

[Java](#)

```
List<String> res = e.getRolesForUser("alice");
```

GetUsersForRole()

GetUsersForRole 获取具有角色的用户。

例如:

Go Node.js PHP Python .NET Rust Java

```
res := e.GetUsersForRole("data1_admin")  
  
const res = await e.getUsersForRole('data1_admin')  
  
$res = $e->getUsersForRole("data1_admin");  
  
users = e.get_users_for_role("data1_admin")  
  
var res = e.GetUsersForRole("data1_admin");  
  
let users = e.get_users_for_role("data1_admin", None); // No  
domain  
  
List<String> res = e.getUsersForRole("data1_admin");
```

HasRoleForUser()

HasRoleForUser 确定用户是否具有角色。

例如:

Go Node.js PHP Python .NET Rust Java

```
res := e.HasRoleForUser("alice", "data1_admin")
```

```
const res = await e.hasRoleForUser('alice', 'data1_admin')

$res = $e->hasRoleForUser("alice", "data1_admin");

has = e.has_role_for_user("alice", "data1_admin")

var res = e.HasRoleForUser("alice", "data1_admin");

let has = e.has_role_for_user("alice", "data1_admin", None); //  
No domain

boolean res = e.hasRoleForUser("alice", "data1_admin");
```

AddRoleForUser()

AddRoleForUser 为用户添加角色。如果用户已经拥有该角色，则返回false。

例如：

Go Node.js PHP Python .NET Rust Java

```
e.AddRoleForUser("alice", "data2_admin")

await e.addRoleForUser('alice', 'data2_admin')

$e->addRoleForUser("alice", "data2_admin");

e.add_role_for_user("alice", "data2_admin")
```

```
var added = e.AddRoleForUser("alice", "data2_admin");
or
var added = await e.AddRoleForUserAsync("alice", "data2_admin");

let added = e.add_role_for_user("alice", "data2_admin",
None).await?; // No domain

boolean added = e.addRoleForUser("alice", "data2_admin");
```

AddRolesForUser()

AddRolesForUser 为用户添加多个角色。如果用户已经拥有该角色，则返回false。(不会受影响)

例如:

[Go](#) [Node.js](#) [Rust](#)

```
var roles = []string{"data2_admin", "data1_admin"}
e.AddRolesForUser("alice", roles)

const roles = ["data1_admin", "data2_admin"];
roles.map((role) => e.addRoleForUser("alice", role));

let roles = vec!["data1_admin".to_owned(),
"data2_admin".to_owned()];
let all_added = e.add_roles_for_user("alice", roles,
None).await?; // 无域
```

DeleteRoleForUser()

DeleteRoleForUser 删除用户的角色。如果用户没有该角色，则返回false。

例如：

Go

Node.js

PHP

Python

.NET

Rust

Java

```
e.DeleteRoleForUser("alice", "data1_admin")  
  
await e.deleteRoleForUser('alice', 'data1_admin')  
  
$e->deleteRoleForUser("alice", "data1_admin");  
  
e.delete_role_for_user("alice", "data1_admin")  
  
var deleted = e.DeleteRoleForUser("alice", "data1_admin");  
or  
var deleted = await e.DeleteRoleForUser("alice", "data1_admin");  
  
let deleted = e.delete_role_for_user("alice", "data1_admin",  
None).await?; // No domain  
  
boolean deleted = e.deleteRoleForUser("alice", "data1_admin");
```

DeleteRolesForUser()

DeleteRolesForUser 删除用户的所有角色。如果用户没有任何角色，则返回false。

例如：

Go Node.js PHP Python .NET Rust Java

```
e.DeleteRolesForUser("alice")

await e.deleteRolesForUser('alice')

$e->deleteRolesForUser("alice");

e.delete_roles_for_user("alice")

var deletedAtLeastOne = e.DeleteRolesForUser("alice");
or
var deletedAtLeastOne = await
e.DeleteRolesForUserAsync("alice");

let deleted_at_least_one = e.delete_roles_for_user("alice",
None).await?; // 无域

boolean deletedAtLeastOne = e.deleteRolesForUser("alice");
```

DeleteUser()

DeleteUser 删除一个用户。如果用户不存在，则返回false（也就是说不受影响）。

例如：

Go Node.js PHP Python .NET Rust Java

```
e.DeleteUser("alice")

await e.deleteUser('alice')

$e->deleteUser("alice");

e.delete_user("alice")

var deleted = e.DeleteUser("alice");
or
var deleted = await e.DeleteUserAsync("alice");

let deleted = e.delete_user("alice").await?;

boolean deleted = e.deleteUser("alice");
```

DeleteRole()

DeleteRole 删除一个角色。

例如:

Go Node.js PHP Python .NET Rust Java

```
e.DeleteRole("data2_admin")

await e.deleteRole("data2_admin")
```

```
$e->deleteRole("data2_admin");

e.delete_role("data2_admin")

var deleted = e.DeleteRole("data2_admin");
or
var deleted = await e.DeleteRoleAsync("data2_admin");

let deleted = e.delete_role("data2_admin").await?;

e.deleteRole("data2_admin");
```

DeletePermission()

DeletePermission 删除权限。 如果权限不存在，则返回false（aka不受影响）。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
e.DeletePermission("read")

await e.deletePermission('read')

$e->deletePermission("read");

e.delete_permission("read")
```

```
var deleted = e.DeletePermission("read");
or
var deleted = await e.DeletePermissionAsync("read");

let deleted =
e.delete_permission(vec!["read".to_owned()]).await?;

boolean deleted = e.deletePermission("read");
```

AddPermissionForUser()

AddPermissionForUser 为用户或角色添加权限。如果用户或角色已经拥有该权限(aka不受影响)，则返回false。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
e.AddPermissionForUser("bob", "read")

await e.addPermissionForUser('bob', 'read')

$e->addPermissionForUser("bob", "read");

e.add_permission_for_user("bob", "read")

var added = e.AddPermissionForUser("bob", "read");
or
var added = await e.AddPermissionForUserAsync("bob", "read");
```

```
let added = e.add_permission_for_user("bob",
    vec!["read".to_owned()]).await?;

boolean added = e.addPermissionForUser("bob", "read");
```

AddPermissionsForUser()

AddPermissionForUser 为用户或角色添加多个权限。如果用户或角色已经有一个权限，则返回 false (不会受影响)。

例如：

[Go](#) [Node.js](#) [Rust](#)

```
var permissions = [][]string{{"data1",
    "read"}, {"data2", "write"}}
for i := 0; i < len(permissions); i++ {
    e.AddPermissionsForUser("alice", permissions[i])
}

const permissions = [
    ["data1", "read"],
    ["data2", "write"],
];
permissions.map((permission) => e.addPermissionForUser("bob",
    ...permission));

let permissions = vec![
    vec!["data1".to_owned(), "read".to_owned()],
```

DeletePermissionForUser()

DeletePermissionForUser 删除用户或角色的权限。如果用户或角色没有权限（aka不受影响），则返回false。

例如：

Go Node.js PHP Python .NET Rust Java

```
e.DeletePermissionForUser("bob", "read")

await e.deletePermissionForUser("bob", "read")

$e->deletePermissionForUser("bob", "read");

e.delete_permission_for_user("bob", "read")

var deleted = e.DeletePermissionForUser("bob", "read");
or
var deleted = await e.DeletePermissionForUserAsync("bob",
"read");

let deleted = e.delete_permission_for_user("bob",
vec!["read".to_owned()]).await?;

boolean deleted = e.deletePermissionForUser("bob", "read");
```

DeletePermissionsForUser()

DeletePermissionsForUser 删除用户或角色的权限。如果用户或角色没有任何权限(aka不受影响) , 则返回false。

例如:

Go Node.js PHP Python .NET Rust Java

```
e.DeletePermissionsForUser("bob")  
  
await e.deletePermissionsForUser('bob')  
  
$e->deletePermissionsForUser("bob");  
  
e.delete_permissions_for_user("bob")  
  
var deletedAtLeastOne = e.DeletePermissionsForUser("bob");  
or  
var deletedAtLeastOne = await  
e.DeletePermissionsForUserAsync("bob");  
  
let deleted_at_least_one =  
e.delete_permissions_for_user("bob").await?;  
  
boolean deletedAtLeastOne = e.deletePermissionForUser("bob");
```

GetPermissionsForUser()

GetPermissionsForUser 获取用户或角色的权限。

例如：

Go Node.js PHP Python .NET Java

```
e.GetPermissionsForUser("bob")
```

```
await e.getPermissionsForUser('bob')
```

```
$e->getPermissionsForUser("bob");
```

```
e.get_permissions_for_user("bob")
```

```
var permissions = e.GetPermissionsForUser("bob");
```

```
List<List<String>> permissions = e.getPermissionsForUser("bob");
```

HasPermissionForUser()

HasPermissionForUser 确定用户是否具有权限。

例如：

Go Node.js PHP Python .NET Rust Java

```
e.HasPermissionForUser("alice", []string{"read"})  
  
await e.hasPermissionForUser('alice', 'read')  
  
$e->hasPermissionForUser("alice", []string{"read"});  
  
has = e.has_permission_for_user("alice", "read")  
  
var has = e.HasPermissionForUser("bob", "read");  
  
let has = e.has_permission_for_user("alice",  
    vec!["data1".to_owned(), "read".to_owned()]);  
  
boolean has = e.hasPermissionForUser("alice", "read");
```

GetImplicitRolesForUser()

GetImplicitRolesForUser 获取用户具有的隐式角色。与GetRolesForUser()相比，该函数除了直接角色外还检索间接角色。

例如：

g, Alice, role:admin

g, role:admin, role:user

GetRolesForUser("Alice") 只能获取：["role:admin"]。

但 GetImplicitRolesForUser("alice") 将获取：["role:admin", "role:user"]。

例如：

```
e.GetImplicitRolesForUser("alice")  
  
await e.getImplicitRolesForUser("alice")  
  
$e->getImplicitRolesForUser("alice");  
  
e.get_implicit_roles_for_user("alice")  
  
var implicitRoles = e.GetImplicitRolesForUser("alice");  
  
e.get_implicit_roles_for_user("alice", None); // No domain  
  
List<String> implicitRoles = e.getImplicitRolesForUser("alice");
```

GetImplicitUsersForRole()

GetImplicitUsersForRole 获取所有继承该角色的用户 与GetUsersForRole() 相比，这个函数检索间接用户。

例如：

g, Alice, role:admin
g, role:admin, role:user

GetRolesForUser("Alice") 只能获取： ["role:admin"]。

但 GetImplicitUsersForRole("role:alice") 将获取： ["role:admin", "alice"]。

例如：

[Go](#) [Node.js](#) [Java](#)

```
users := e.GetImplicitUsersForRole("role:user")  
  
const users = e.getImplicitUsersForRole("role:user");  
  
List<String> users = e.getImplicitUsersForRole("role:user");
```

GetImplicitPermissionsForUser()

GetImplicitPermissionsForUser 获得用户或角色的隐含权限。

与GetPermissionsForUser() 相比，此函数获取继承角色的权限。

例如：

```
p, admin, data1, read  
p, alice, data2, read  
g, alice, admin
```

GetPermissionsForUser("alice") 只能获取：[["alice", "data 2", "read"]]。

但GetImplicitPermissionsForUser("alice") 将获取：[["admin", "data1", "read"], ["alice", "data2", "read"]]。

例如：

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
e.GetImplicitPermissionsForUser("alice")
```

```
await e.getImplicitPermissionsForUser("alice")  
  
$e->getImplicitPermissionsForUser("alice");  
  
e.get_implicit_permissions_for_user("alice")  
  
var implicitPermissions =  
e.GetImplicitPermissionsForUser("alice");  
  
e.get_implicit_permissions_for_user("alice", None); // 无域  
  
List<List<String>> implicitPermissions =  
e.getImplicitPermissionsForUser("alice");
```

GetNamedImplicitPermissionsForUser()

GetNamedImplicitPermissionsForUser 通过命名策略获得用户或角色的隐含权限 与 GetImplicitPermissionPermissionsForUser() 相比，此函数允许您指定策略名称。

例如： p, admin, data1, read p2, admin, creaty g, alice, admin

GetImplicitPermissionsForUser("Alice") 只能获取到默认为“p”的权限: [[{"admin", "data1", "read"}]]

但是你可以指定策略为 "p2" 来获取: [[{"admin", "create"}]] 由
GetNamedImplicitPermissionsForUser("p2","Alice")

例如：

[Go](#) [Python](#)

```
e.GetNamedImplicitPermissionsForUser("p2", "alice")  
e.get_named_implicit_permissions_for_user("p2", "alice")
```

GetDomainsForUser()

GetDomainsForUser 获取用户拥有的所有域名。

例如： p, admin, domain1, data1, read p, admin, domain2, data2, read p, admin, domain2, data2, write g, alice, admin, domain1 g, alice, admin, domain2

GetDomainsForUser("Alice") 可以获取 ["domain1", "domain2"]

例如：

Go

```
result, err := e.GetDomainsForUser("alice")
```

GetImplicitResourcesForUser()

GetImplicitResourcesForUser 返回为true的策略给用户。

例如：

```
p, alice, data1, read  
p, bob, data2, write  
p, data2_admin, data2, read  
p, data2_admin, data2, write  
  
g, alice, data2_admin
```

`GetImplicitResourcesForUser("alice")` 将会返回 `[[alice data1 read] [alice data2 read] [alice data2 write]]`

Go

```
resources, err := e.GetImplicitResourcesForUser("alice")
```

域内基于角色的访问控制 API

一个更友好的域内基于角色的访问控制的API。这个API是Management API的子集。RBAC用户可以使用这个API来简化代码。

参考

全局变量 `e` 是 Enforcer 实例。

[Go](#) [Node.js](#) [PHP](#) [Python](#) [.NET](#) [Rust](#) [Java](#)

```
e, err := NewEnforcer("examples/rbac_with_domains_model.conf",
"examples/rbac_with_domains_policy.csv")
```

```
const e = await newEnforcer('examples/
rbac_with_domains_model.conf', 'examples/
rbac_with_domains_policy.csv')
```

```
$e = new Enforcer('examples/rbac_with_domains_model.conf',
'examples/rbac_with_domains_policy.csv');
```

```
e = casbin.Enforcer("examples/rbac_with_domains_model.conf",
"examples/rbac_with_domains_policy.csv")
```

```
var e = new Enforcer("examples/rbac_with_domains_model.conf",
"examples/rbac_with_domains_policy.csv");
```

```
let mut e = Enforcer::new("examples/
rbac_with_domains_model.conf", "examples/
rbac_with_domains_policy.csv").await?;
```

```
Enforcer e = new Enforcer("examples/
rbac_with_domains_model.conf", "examples/
rbac_with_domains_policy.csv");
```

GetUsersForRoleInDomain()

GetUsersForRoleInDomain 获取具有域内角色的用户。

例如:

[Go](#) [Node.js](#) [Python](#)

```
res := e.GetUsersForRoleInDomain("admin", "domain1")
```

```
const res = e.getUsersForRoleInDomain("admin", "domain1")
```

```
res = e.get_users_for_role_in_domain("admin", "domain1")
```

GetRolesForUserInDomain()

GetRolesForUserInDomain 获取域内用户的角色

例如:

Go Node.js Python Java

```
res := e.GetRolesForUserInDomain("admin", "domain1")  
  
const res = e.getRolesForUserInDomain("alice", "domain1")  
  
res = e.get_roles_for_user_in_domain("alice", "domain1")  
  
List<String> res = e.getRolesForUserInDomain("admin",  
"domain1");
```

GetPermissionsForUserInDomain()

GetPermissionsForUserInDomain 获取域内用户或角色的权限。

例如:

Go Java

```
res := e.GetPermissionsForUserInDomain("alice", "domain1")  
  
List<List<String>> res =  
e.getPermissionsForUserInDomain("alice", "domain1");
```

AddRoleForUserInDomain()

AddRoleForUserInDomain 在域内为用户添加角色 如果用户已经拥有该角色，则返回 false。

例如:

[Go](#) [Python](#) [Java](#)

```
ok, err := e.AddRoleForUserInDomain("alice", "admin", "domain1")  
  
ok = e.add_role_for_user_in_domain("alice", "admin", "domain1")  
  
boolean ok = e.addRoleForUserInDomain("alice", "admin",  
"domain1");
```

DeleteRoleForUserInDomain()

DeleteRoleForUserInDomain 在域内删除用户的角色 如果用户没有该角色, 则返回 false。

例如:

[Go](#) [Java](#)

```
ok, err := e.DeleteRoleForUserInDomain("alice", "admin",  
"domain1")  
  
boolean ok = e.deleteRoleForUserInDomain("alice", "admin",  
"domain1");
```

DeleteRolesForUserInDomain()

DeleteRolesForUserInDomain 删除域内用户的所有角色 如果用户没有任何角色，则返回false。

例如:

Go

```
ok, err := e.DeleteRolesForUserInDomain("alice", "domain1")
```

GetAllUsersByDomain()

GetAllUsersByDomain 将获得所有与该域相关联的用户。 如果模型中没有定义域名，将返回空字符串。

例如:

Go

```
res := e.GetAllUsersByDomain("domain1")
```

DeleteAllUsersByDomain()

DeleteAllUsersByDomain 将删除与该域相关的所有用户 如果模型中没有定义域名，则返回 false。

例如:

[Go](#)

```
ok, err := e.DeleteAllUsersByDomain("domain1")
```

DeleteDomains()

DeleteDomains 将删除所有相关的用户和角色。如果没有提供参数，它会删除所有域。

例如:

[Go](#)

```
ok, err := e.DeleteDomains("domain1", "domain2")
```

GetAllDomains()

GetAllDomains 将获得所有域。

例如:

[Go](#)

```
res, _ := e.GetAllDomains()
```

① 备注

如果您正在处理类似 `name::domain` 的域，这可能会导致意外的行为。在 Casbin 里，`::` 是一个倒置的关键词，就像编程语言中的关键字 `for`, `if` 一样，我们不应该在一个域中放置 `::`

角色管理器API

角色管理器

RoleManager提供接口来定义管理角色的操作。添加匹配函数到角色管理器允许在角色名称和域中使用通配符。

AddNamedMatchingFunc()

AddNamedMatchingFunc 通过 ptype 角色管理器添加MatchingFunc MatchingFunc 将在操作角色匹配时工作。

[Go](#) [Node.js](#)

```
e.AddNamedMatchingFunc("g", "", util.KeyMatch)
_, _ = e.AddGroupingPolicies([][]string{{"*", "admin",
"domain1"}})
_, _ = e.GetRoleManager().HasLink("bob", "admin",
"domain1") // -> true, nil

await e.addNamedMatchingFunc('g', Util.keyMatchFunc);
await e.addGroupingPolicies([["*", 'admin', 'domain1']]);
await e.getRoleManager().hasLink('bob', 'admin', 'domain1');
```

例如:

[Go](#) [Node.js](#)

```
e, _ := casbin.NewEnforcer("path/to/model", "path/to/
policy")
e.AddNamedMatchingFunc("g", "", util.MatchKey)

const e = await newEnforcer('path/to/model', 'path/to/
policy');
await e.addNamedMatchingFunc('g', Util.keyMatchFunc);
```

AddNamedDomainMatchingFunc()

AddNamedDomainMatchingFunc 通过 ptype 把MatchingFunc 添加到 RoleManager 中。`DomainMatchingFunc` 类似于上面列出的 `MatchingFunc`

For example:

[Go](#) [Node.js](#)

```
e, _ := casbin.NewEnforcer("path/to/model", "path/to/
policy")
e.AddNamedDomainMatchingFunc("g", "", util.MatchKey)

const e = await newEnforcer('path/to/model', 'path/to/
policy');
await e.addNamedDomainMatchingFunc('g', Util.keyMatchFunc);
```

GetRoleManager()

GetRoleManager 获取现存的 `g` 的role manager。

例如：

Go Node.js Python

```
rm := e.GetRoleManager()  
  
const rm = await e.getRoleManager();  
  
rm = e.get_role_manager()
```

GetNamedRoleManager()

GetNamedRoleManager通过命名的ptype 获取角色管理器。

例如：

Go Node.js Python

```
rm := e.GetNamedRoleManager("g2")  
  
const rm = await e.getNamedRoleManager("g2");  
  
rm = e.get_named_role_manager("g2")
```

SetRoleManager()

GetRoleManager 获取现存的 g 的role manager。

例如：

Go Node.js Python

```
e.SetRoleManager(rm)  
e.setRoleManager(rm);  
rm = e.set_role_manager(rm)
```

SetNamedRoleManager()

SetNamedRoleManager 将角色管理员设置为命名的ptype

For example:

Go Python

```
rm := e.SetNamedRoleManager("g2", rm)  
rm = e.set_role_manager("g2", rm)
```

Clear()

Clear清除所有存储的数据并将角色管理器重置到初始状态。

例如：

[Go](#) [Node.js](#) [Python](#)

```
rm.Clear()  
  
await rm.clear();  
  
rm.clear()
```

AddLink()

AddLink添加了两个角色之间的继承链接。 角色: 名称1 和 角色: 名称2 域是角色的前缀(可以用于其他目的)。

例如:

[Go](#) [Node.js](#) [Python](#)

```
rm.AddLink("u1", "g1", "domain1")  
  
await rm.addLink('u1', 'g1', 'domain1');  
  
rm.add_link("u1", "g1", "domain1")
```

DeleteLink()

DeleteLink 删除两个角色之间的继承链接。 角色: 名称1 和 角色: 名称2 域是角色的前缀(可以用于其他目的)。

例如：

Go Node.js Python

```
rm.DeleteLink("u1", "g1", "domain1")  
  
await rm.deleteLink('u1', 'g1', 'domain1');  
  
rm.delete_link("u1", "g1", "domain1")
```

HasLink()

HasLink 决定两种角色之间是否存在联系。 role: name1 继承自 role: name2. 域是角色的前缀(可以用于其他目的)。

例如：

Go Node.js Python

```
rm.HasLink("u1", "g1", "domain1")  
  
await rm.hasLink('u1', 'g1', 'domain1');  
  
rm.has_link("u1", "g1", "domain1")
```

GetRoles()

GetRoles 获取一个用户所继承的角色 域是角色的前缀(可以用于其他目的)。

例如：

Go Node.js Python

```
rm.GetRoles("u1", "domain1")  
  
await rm.getRoles('u1', 'domain1');  
  
rm.get_roles("u1", "domain")
```

GetUsers()

GetUsers 获取继承自一个角色的用户 域是用户的前缀(可以用于其他目的)。

例如：

Go Node.js Python

```
rm.GetUsers("g1")  
  
await rm.getUsers('g1');  
  
rm.get_users("g1")
```

PrintRoles()

PrintRoles 打印所有的角色到日志。

例如:

Go Node.js Python

```
rm.PrintRoles()  
  
await rm.printRoles();  
  
rm.print_roles()
```

SetLogger()

SetLogger设置角色管理器的日志。

例如:

Go

```
logger := log.DefaultLogger{}  
logger.EnableLog(true)  
rm.SetLogger(&logger)  
_ = rm.PrintRoles()
```

GetDomains()

GetDomains 获取用户拥有的域

例如:

[Go](#)

```
result, err := rm.GetDomains(name)
```

数据权限

我们有两个解决方案用于数据权限(过滤)。 使用隐式作业 API。 或者只使用 BatchEnforce() API.

1. 查询隐含角色或权限

当用户通过RBAC层次结构继承角色或权限，而不是直接在策略规则中分配它们时，我们将这种类型的分配称为 `implicit`。要查询这种隐式关系，需要使用以下两个api:
`GetImplicitRolesForUser()` 以及 `GetImplicitPermissionsForUser` 替代
`GetRolesForUser()` 以及 `GetPermissionsForUser`。有关详情，请参阅 [this GitHub issue](#)。

2. 使用 `BatchEnforce()`

`BatchEnforce` 强制执行每个请求并返回一个布尔数组的结果

例如:

[Go](#) [Node.js](#) [Java](#)

```
boolArray, err := e.BatchEnforce(requests)
```

```
const boolArray = await e.batchEnforce(requests);
```

```
List<Boolean> boolArray = e.batchEnforce(requests);
```



> 高级用法

高级用法

多线程

以多线程方式使用 Casbin

性能测试

Casbin政策执行的开销

性能优化

性能优化

Kubernetes的授权

Kubernetes (k8s) RBAC & ABAC授权基于Casbin 的中间件

Admission Webhook For K8s

Kubernetes (k8s) RBAC & 基于Casbin的ABAC授权中间件

使用 Envoy 实现 Service Mesh 权限管理

使用 Envoy 实现 Service Mesh 权限管理

多线程

如果您想以多线程方式使用 Casbin，您可以使用 Casbin enforcer 提供的同步包装器：
https://github.com/casbin/casbin/blob/master/enforcer_synced.go (GoLang) 和
https://github.com/casbin/casbin-cpp/blob/master/casbin/enforcer_synced.cpp (C++)。

它还支持 `AutoLoad` 特性，这意味着如果最新的策略规则发生了更改，Casbin enforcer 将自动从DB加载这些规则。调用 `StartAutoLoadPolicy()` 启动定期自动加载策略，调用 `StopAutoLoadPolicy()` 停止策略。

性能测试

Go C++ Lua (JIT)

策略执行的负载在[model_b_test.go](#)中进行基准测试。 测试是：

英特尔 酷睿 i7-6700HQ CPU @ 2.60GHz, 2601 Mhz, 4 核, 8 处理器

`go test -bench= -benchmem` 的测试结果如下 (op = 一次 `Enforce()` 调用, ms = 毫秒, KB = 千字节)：

测试用例	规则大小	时间开销 (ms/op)	内存开销 (KB)
ACL	2 规则 (2用户)	0.015493	5.649
RBAC	5条规则 (2用户, 1个角色)	0.021738	7.522
RBAC (小型)	1100条规则 (1000用户, 100个角色)	0.164309	80.620
RBAC (中型)	11000条规则 (10000用户, 1000个角色)	2.258262	765.152
RBAC (大型)	110000条规则 (100000用户, 10000个角色)	23.916776	7606

测试用例	规则大小	时间开销 (ms/op)	内存开销 (KB)
具有资源角色的 RBAC	6条规则 (2用户, 2个角色)	0.021146	7.906
带有域/租户的 RBAC	6 条规则 (2个用户, 1个角色, 2 个域)	0.032696	10.755
ABAC	0 规则 (0用户)	0.007510	2.328
RESTful	5 规则 (3用户)	0.045398	91.774
拒绝改写	6条规则 (2用户, 1个角色)	0.023281	8.370
优先级	9条规则 (2用户, 2个角色)	0.016389	5.313

Casbin CPP 的策略执行测试是在 [tests/benchmarks](#) 中, 使用 [Google's benchmarking tool](#) 完成的 这些基准的测试结果是:

英特尔 酷睿 i5-6300HQ CPU @ 2.30GHz, 4 核, 4 线程

以下是执行以 `Release` 配置编译的 `casbin_benchmark` 基准结果(`op = 一次 enforce()` 调用, `ms = 毫秒`):

测试用例	规则大小	时间开销 (ms/op)
ACL	2 规则 (2用户)	0.0195
RBAC	5条规则 (2用户, 1个角色)	0.0288

测试用例	规则大小	时间开销 (ms/op)
RBAC (小型)	1100条规则 (1000用户, 100个角色)	0.300
RBAC (中型)	11000条规则 (10000用户, 1000个角色)	2.113
RBAC (大型)	110000条规则 (100000用户, 10000个角色)	21.450
具有资源角色的 RBAC	6条规则 (2用户, 2个角色)	0.03
带有域/租户的 RBAC	6 条规则 (2个用户, 1个角色, 2个域)	0.041
ABAC	0 规则 (0用户)	NA
RESTful	5 规则 (3用户)	NA
拒绝改写	6条规则 (2用户, 1个角色)	0.0246
优先级	9条规则 (2用户, 2个角色)	0.035

Lua Casbin的 策略执行的测试是在 [ben.lua](#) 中进行的。 测试台是 Ubuntu VM , CPU型号为

AMD Ryzen(TM) 5 4600H CPU @ 3.0GHz, 6 核, 12 Threads

`luajit bey.lua` 的基准结果如下(op = 一个 `enforce()` 调用, ms = 毫秒):

测试用例	规则大小	时间开销 (ms/op)
ACL	2 规则 (2用户)	0.0533
RBAC	5条规则 (2用户, 1个角色)	0.0972
RBAC (小型)	1100条规则 (1000用户, 100个角色)	0.8598
RBAC (中型)	11000条规则 (10000用户, 1000个角色)	8.6848
RBAC (大型)	110000条规则 (100000用户, 10000个角色)	90.3217
具有资源角色的 RBAC	6条规则 (2用户, 2个角色)	0.1124
带有域/租户的 RBAC	6 条规则 (2个用户, 1个角色, 2个域)	0.1978
ABAC	0 规则 (0用户)	0.0305
RESTful	5 规则 (3用户)	0.1085
拒绝改写	6条规则 (2用户, 1个角色)	0.1934
优先级	9条规则 (2用户, 2个角色)	0.1437

性能优化

当应用于数以百万计的用户或权限的生产环境时，您可能会在Casbin 的强制执行中遇到性能降级，通常有两个原因：

高访问量

每秒到来的请求数量非常庞大，例如：单个Casbin实例每秒就能收到10000条请求。在这种情况下，仅靠一个Casbin实例通常难以处理完所有请求。现在有两种解决方案：

1. 运用多线程来运行多个Casbin实例，这样以来您就可以充分利用机器中的所有内核。 详情请参阅：[多线程](#)
2. 将Casbin实例部署到机器集群(多台机器)。 使用[Watcher](#)来确保所有Casbin实例运行一致。 详情请参阅：[Watcher](#)。

备注

您可以同时使用上述方法，例如将Casbin部署到10台机群。 每台机器同时具有5个线程来满足Casbin强制执行请求。

大量的策略规则

在云或多租户环境中，可能需要数百万条策略规则。 每次执行请求甚至是在最初期加载策略规则的速度非常缓慢。 这类事件通常可以通过以下几种方式缓解：

1. 您的Casbin模型或规则设计得不够好。 精致的模型和策略将抽象每个用户/租户的重复逻辑，并将规则的数量减少到一个非常小的级别(< 100)： 例如：您可以在所有租户之间分享一些默认规则，让用户稍后自定义他们的规则。 自定义规则可以覆盖默认规则。 如果您仍然有疑问，请将GitHub issue发送到Casbin报告。

2. 通过共享让Casbin 的执行者只需要加载一套小套策略规则，例如：enforcer_0 只为 tenant_0 到tenant_99提供服务, enforcer_1 只为tenant_100到tenant_199提供服务。 只需要加载所有策略规则的一部分，详情请参阅：[策略子集加载](#)。
3. 以授予RBAC角色权限，取代直接授予用户权限。 Casbin的RBAC是通过角色继承树来实现的(作为缓存)。 因此授予类似Alice这样的用户权限， Casbin只使用O(1) 时间查询RBAC树来获取角色用户关系并执行操作。 如果您的 g 规则不会经常改变， 那么RBAC 树将不需要进行更新。 详情请参阅这条讨论: <https://github.com/casbin/casbin/issues/681#issuecomment-763801583>

① 备注

您可以同时尝试以上的方法。

Kubernetes的授权

[K8s-authz](#) 是基于Casbin实现的 Kubernetes (k8s) RBAC & ABAC授权中间件 这个中间件使用 K8s 验证访问 webhook 来检查 casbin 定义的策略，了解每个 K8s 资源的请求。这些自定义接入控制器在由 api 服务器转发并基于逻辑的请求对象上执行某种验证，向包含允许或拒绝请求信息的 api 服务器发送回复。这些控制器使用 [ValidatingAdmissionWebhook](#) 在 Kubernetes 注册。

K8s API服务器需要知道何时将传入请求发送到我们的接入控制器。这方面，我们已经定义了一个验证 webhook，它将代理任何类型的 K8s 资源/子资源的请求，并对其进行策略核查。只有在casbin enforcer 授权的情况下，才允许用户利用这些资源进行操作。[enforcer](#) 检查策略中定义的用户的角色。这种中间件将部署在K8s集群中。

需求

在继续之前，请确保以下各项：

- 正在运行的 k8s 集群。您可以通过 Docker 桌面上启用它来运行集群，也可以在本地或在您的服务器上设置完整的 K8s 生态系统。您可以按照这个详细的 [指南](#) 来设置本地在 Windows 上的 k8s 集群，如果想要为 Linux 设置，请点击这个 [指南](#)。
- Kubectl CLI 这是在Windows上设置它的 [指南](#)。还有这个 [Linux的指南](#)。
- OpenSSL

使用方法

- 通过使用 openssl 并运行 make certs 或以下脚本为每个用户生成证书和密钥： -

```
./gen_cert.sh
```

- 通过 Dockerfile 手动运行下面的命令构建Dockerfile， 然后根据build在 部署文件上更改构建版本。

```
docker build -t casbin/k8s_authz:0.1
```

- 在 model.conf 和 policy.csv 中定义 casbin 策略。 您可以参考 docs 来了解更多关于这些策略工作的信息。
- 在部署之前， 您可以根据您的用法更改 main.go 的端口， 同时也可以更改验证 webhook 配置文件 中的端口。
- 通过运行以下程序来在 k8s 集群上部署验证控制器和 webhook : -

```
kubectl apply -f deployment.yaml
```

- 对于生产服务器， 我们需要创建一个 k8s secret 来放置证书以保证安全性。

```
kubectl create secret generic casbin -n default \
--from-file=key.pem=certs/casbin-key.pem \
--from-file=cert.pem=certs/casbin-crt.pem
```

- 只要完成了这一部分我们需要更改 证书目录 ， 然后在 manifest中用 secret 显示

现在服务器应该运行并准备验证在 k8s 资源上操作的请求。

如果有任何疑问， 您可以在我们的 Gitter频道 上询问。

Kubernetes + Casbin 插件：K8s-Gatekeeper

1. 概述 Casbin K8s-Gatekeeper

Casbin K8s-GateKeeper 是 Kubernetes 的一个接入 webhook，该接口将 Casbin 作为访问控制工具。通过使用 Casbin K8s-gatekeeper，您可以建立灵活的规则来授权或拦截 K8s 资源上的任何操作，而无需编写任何代码，只需编写几行关于 Casbin 模型和策略的声明性配置(它们是 Casbin ACL(访问控制列表)语言的一部分)。

Casbin K8s-GateKeeper 由 Casbin 社区开发、维护。该项目仓库如下：<https://github.com/casbin/k8s-gatekeeper>

0.1 简单的例子

例如，你无需写任何代码，只需使用以下配置即可实现此功能：“禁止在任何部署中使用带有特定标签的图像”：

模型：

```
[request_definition]
r = obj

[policy_definition]
p = obj, eft

[policy_effect]
e = !some(where (p.eft == deny))

[matchers]
m = r.obj.Request.Namespace == "default" && r.obj.Request.Resource.Resource
== "deployments" && \
contain(split(accessWithWildcard(${OBJECT}.Spec.Template.Spec.Containers , "*",
"Image"), ":" , 1) , p.obj)
```

策略：

```
p, "1.14.1", 否认
```

这些都是用常见的 Casbin ACL 语言编写的。如若你已阅读过相关章节，那就很容易理解。

Casbin K8s-Gatekeeper 具有以下优势：

- 它简单易用。写几行 ACL 比写大量代码要好得多。

- 它允许快速更新配置。您无需关闭整个插件来修改配置。
- 它是新兴工具。您可以在任何k8s资源上制定任意规则，它都可以探索到。
- 它对k8s接入webhook进行了复杂检查。你无需了解“K8s admission webhook”是什么，也无需知道如何为它编写代码。您需要做的是了解您想要设置约束的资源，然后写入Casbin ACL。大家都知道K8s十分复杂，但使用Casbin K8s-Gatekeeper可以节省您的时间。
- 它由Casbin 社区管理。如果此插件有任何内容使您感到困惑，或者您在尝试此插件时遇到任何问题，请随时联系我们。

1.1 Casbin K8s-gatekeeper是如何运作的？

K8s-gatekeeper 是 k8s 的许可 webhook，使用 [Casbin](#) 可应用任意用户定义的访问控制规则，来帮助管理员阻止 k8 上的任何不当操作。

Casbin是一个强大、高效的开放源码访问控制库。它支持根据各种出入控制模式执行授权。关于Casbin的更多详情，请参阅 [概述](#)。

K8 中的许可 webhooks 是 HTTP 回调，负责接收“许可请求”并执行相关程序。K8s-gatekeeper 是一种特殊类型的 webhook: '验证准入Webhook'，它可以决定是否接受准入请求。准入请求，是指特定的 K8 资源操作的 HTTP 请求(例如，创建/删除一个部署)。更多关于接入 webhooks 的信息，请参阅 [K8s官方的联系方式](#)

1.2 说明其运作方式的一个例子。

例如，如果想要创建一个包含正在运行nginx 的pod 的部署(使用 kubectl 或 k8 s 客户端)，K8s将生成一个许可请求，这个请求(如果翻译成yaml格式)可能是这样的。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.1
          ports:
            - containerPort: 80
```

此请求将经过图片中显示的所有中间件审核，包括我们的K8s-gatekeeper。K8s-gatekeeper可以检测到 K8s中存储的

Casbin 执行器，这些执行器由用户创建和维护(通过我们提供的 kubectl 或 go-client)。 每个执行器都有Casbin模型和 Casbin政策。 许可请求将由执行器一一处理，只有通过所有执行器， k8 -gatekeeper才会接受该请求。

(如果您不理解Casbin 执行器、模型或政策， 见本文档: [开始](#))

例如，由于某些原因， 管理员想禁止图像“nginx:1.14.1”， 同时允许“nginx:1.3”。那么可以创建一个执行器， 其中包括以下规则和政策：（我们将解释如何创建一个执行器， 这些模式和政策是什么以及如何将其写入以下章节。）

模型:

```
[request_definition]
r = obj

[policy_definition]
p = obj,eft

[policy_effect]
e = !some(where (p.eft == deny))

[matchers]
m = r.obj.Request.Namespace == "default" && r.obj.Request.Resource.Resource
== "deployments" && \
access(r.obj.Request.Object.Object.Spec.Template.Spec.Containers , 0, "Image") == p.obj
```

策略:

```
p, "nginx:1.13.1",allow
p, "nginx:1.14.1",deny
```

通过创建一个包含上述模型和策略的执行器， 此执行程序将拒绝先前的许可申请， 这意味着K8s不会创建此部署。

2 安装 K8s-gatekeeper

有三种方法安装 K8s-gatekeeper: 外部 webhook、内部 webhook 和 helm。

① 备注

注意: 这些方法只适用于用户体验K8s-gatekeeper， 并不安全。 如果您想在生产环境中使用，请务必阅读 [第五章。](#) [高级设置](#) 并在安装前根据需要做相应修改。

2.1 内部 webhook

2.1.1 第1步：构建图像

Internal webhook 意味着 webhook 本身将会作为 k8s 内的服务运行。 创建服务和部署需要 K8s-gatekeeper 的图像。 您可以建立自己的图像。

运行

```
docker build --target webhook -t k8s-gatekeeper .
```

然后将会有个本地图像叫做“k8s-gatekeeper:latest”。

① 备注

注意：如果您正在使用 minikube，请在运行 docker 构建之前执行 `val $(minikube -p minikube docker-env)`

2.1.2 第2步：为K8s-gatekeeper设置服务和部署

执行以下命令：

```
kubectl apply -f config/rbac.yaml  
kubectl apply -f config/webhook_deployment.yaml  
kubectl apply -f config/webhook_internal.yaml
```

即将运行 K8s-gatekeeper，您可以使用 `kubectl 获取` 来确认。

2.1.3 第3步：为K8s-gatekeeper 安装Crd 资源

执行以下命令：

```
kubectl apply -f config/auth.casbin.org_casbinmodels.yaml  
kubectl apply -f config/auth.casbin.org_casbinpolicies.yaml
```

2.2 内部 webhook

外部Webhook 意味着 K8s-gatekeeper 将在 K8s 之外运行，K8s 将访问 K8s-gatekeeper，像访问普通网站一样。K8s 要求 webhook 必须是 HTTPS。为了让您体验 K8s-gatekeeper，我们已经为您提供了一套证书以及私钥(尽管它安全系数不高)。如果您想用自己的证书，请参阅 [第五章。高级设置](#) 以调整证书和私钥。

我们提供的证书是为 'webhook.domain.local' 颁发的，所以请修改主机 (例如 /etc/hosts)，point webhook。K8sgatekeeper 正在运行的 omain.local 到 IP 地址。

然后执行

```
go mod tidy
go mod vendor
go run cmd/webhook/main.go
kubectl apply -f config/auth.casbin.org_casbinmodels.yaml
kubectl apply -f config/auth.casbin.org_casbinpolicies.yaml
kubectl apply -f config/webhook_external.yaml
```

2.3 通过 helm 安装 K8s-gatekeeper

2.3.1 第1步：构建图像

请参阅 [第 2.1.1 章](#)

2.3.2 helm 安装

运行 `helm install k8sgatekeepeer ./k8sgatekeepeer`

3. 试用K8s-gatekeeper

3.1 创建Casbin 模型和策略

您有两种方法来创建模型和策略：通过我们提供的kubectl 或go-client。

3.1.1 通过 kubectl 创建/更新Casbin 模型和策略

在K8s-gatekeeper， Casbin模型储存在“CasbinModel”的CRD资源中。其定义可参见 `config/auth.casbin.org_casbinmodels.yaml`

`example/allowed_repo/model.yaml` 中有示例。您应该注意以下几点：

- `metadata.name`: 模型名称 该名称与此模型相关的CasbinPolicy objectives名称相同，这样K8s-gatekeeper 就可以将它们进行匹配并创建一个执行器。
- 启用：如果此字段设置为“false”，将忽略此模型(以及与此模型相关的 CasbinPolicy 对象)。
- 样式文本：一个包含contains模型文本的字符串。

Casbin 策略存储在另一种'CasbinPolicy' 的CRD 资源中，其定义可参见 `config/auth.casbin.org_casbinpolicies.yaml`。

`example/allowed_repo/policy.yaml` 中有示例。您应该注意以下几点：

- `metadata.name`: 策略名称 该名称与此模型相关的CassbinModel object r名称相同，这样K8s-gatekeeper 就可以将它们进行匹配并创建一个执行器

- spec.policyitem: 一个包含casbin模型策略文本的字符串。

创建您自己的 Casbin 模型和 政策文件后，使用

```
kubectl apply -f <filename>
```

使其生效。

Casbin模型和策略一旦创建，最多5秒钟内K8s-gatekeeper便能检测到。

3.1.2 通过我们提供的 go-client 创建/更新 Casbin 模型和策略

我们考虑到可能有这样的情况，用 shell直接在 K8s群组的节点上执行命令并不方便。例如，当您正在为公司建立自动云平台时。因此，我们已经开发了go-client来创建Casbin模型和策略。

go-client 库可参见pkg/client。

在 client.go 中，我们提供了创建客户端的功能。

```
func NewK8sGateKeeperClient(externalClient bool) (*K8sGateKeeperClient, error)
```

外部客户端参数决定K8s-gatekeeper 是否在 K8s 集群中运行。

在 model.go 中，我们提供了各种功能以创建/删除/修改Casbin模型。您可以在model_test.go中找到接口的使用方式。

在 policy.go 中，我们提供了各种功能以创建/删除/修改Casbin模型。您可以在policy_test.go中找到这些接口的使用方式。

3.1.2 试用K8s-gatekeeper是否运作

假定您已经在example/allowed_repo中创建了完整的模型和策略，便可输入：

```
kubectl apply -f example/allowed_repo/testcase/reject_1.yaml
```

你会发现k8s会拒绝这个请求，并提到webhook是拒绝此请求的原因。然而，当您尝试输入example/allowed_repo/testcase/approve_2.yaml时，k8s将会接受该请求。

4. 如何写K8s-gatekeeper模型和策略

首先，您应该知道Casbin模型和策略的基本语法。如果您还未曾了解，请先阅读 [开始](#)。在本章中，我们将假定您已知Casbin模型和策略。

4.1 模型的请求定义

当K8s-gatekeeper 授权请求时，总是输入：the go object of the Admission Request。这意味着执行器总是这样运作。

```
ok, err := enforcer.Enforce(admission)
```

admission是一个由K8s官方go api“K8s .io/api/admission/v1”定义的AdmissionReview对象。您可以在这个资源库看到这个结构的定义：<https://github.com/kubernetes/api/blob/master/admission/v1/types.go>。或查看<https://kubernetes.io/docs/reference/access-authz/extensible-admission-controllers/#webhook-request-and-response> 获取更多信息

因此，对于K8s-gatekeeper使用的任何模型，请求的定义应该始终是这样的：

```
[request_definition]
r = obj
```

名称“obj”不是强制性的，只要名称与 [matchers] 中使用的名称一致即可。

4.2 模型匹配器

您应该使用 Casbin 的 ABAC 特性写下您的规则。然而，Casbin集成的表达式评估工具既不支持在地图或阵列中进行索引，也不支持扩大阵列。因此，K8s-gatekeeper 提供了各种的 “Casbin 功能”作为这些特性的延伸元素。如果您发现这些扩展仍无法满足您的需求，欢迎您提出您的问题，或者直接创建pr。

如果您不知道什么是casbin功能，请参阅 [Function](#) 以获取更多信息。

以下是扩展功能。

4.2.1 外部功能

4.2.1.1 访问

访问主要是用来解决Casbin不支持在地图或数组中索引的问题。[example/allowed_repo/model.yaml](#) 是该功能的例子。

```
[matchers]
m = r.obj.Request.Namespace == "default" && r.obj.Request.Resource.Resource
== "deployments" && \
access(r.obj.Request.Object.Object.Spec.Template.Spec.Containers , 0, "Image") == p.obj
```

在这个匹配器中，`access(r.obj.Request.Object.Object.Spec.Template.Spec.Containers , 0, "Image")` 相当于 `r.obj.Request.Object.Object.Template.Spec.Containers[0].Image`，其中

```
r.obj.Request.Object.Object.Spec.Template.Containers
```

显然是一种slice。

访问还可以调用没有参数和单个返回值的简单功能。可参见例子：`example/container_resource_limit/model.yaml`

```
[matchers]
  m = r.obj.Request.Namespace == "default" && r.obj.Request.Resource.Resource
  == "deployments" && \
    parseFloat(access(r.obj.Request.Object.Object.Spec.Template.Spec.Containers , 0,
  "Resources", "Limits", "cpu", "Value")) >= parseFloat(p.cpu) && \
    parseFloat(access(r.obj.Request.Object.Object.Spec.Template.Spec.Containers , 0,
  "Resources", "Limits", "memory", "Value")) >= parseFloat(p.memory)
```

在这个匹配器中，`access(r.obj.Request.Object.Object.Spec.Template.Spec.Containers , 0,
 "Resources", "Limits", "cpu", "value")` 相当于
`r.obj.Request.Object.Object.Spec.Template.Spec.Containers[0].Resources.Limits["cpu"].value()`，
其中 `r.obj.Request.Object.Object.Spec.Template.Spec.Containers[0].Resources.Limits` 是一个地图，
`value()` 是一个没有参数和返回值的简单功能。

4.2.1.2 访问Withildcard

有时我们经常有像这样的需求：数组中的所有元素都必须有前缀“aaa”。然而，Casbin 不支持。但使用 `accessWithwildcard` 和“map/slice extension” 功能，这样的需求可以很容易地实现。

例如，假定 `a.b.` 是一个数组 `[aaa, bbbb, cccddd, eeee]`，然后 `accessWithwildcard(a, "b", "c", "*")` 将成为 slice `[aa, bbb, ccc, dd, ee]` 我们可以看到，使用通配符 `*` 这个切片便扩大了。

同样，通配符可以多次使用。例如，`用Withildcard输入(a, "b", "c", "*", "*")` 将会得到这样的结果 `[a.b.[0][0], a.b.c[0][1]... a.b.c[1][0], a.b.c[1][1]...]`

4.2.1.3 支持可变长度参数的功能

在Casbin的表达式评估器中，当参数是一个数组时，它将自动扩展为变量长度参数。利用此功能支持数组/切片/地图扩展，我们还整合了服务器功能，接受数组/切片作为参数。

- `包含()`，接受多个参数，并返回除了最后一个参数之外是还有参数等于最后一个参数。
- `分割(a,b,c...,sep,index)` 它返回一个切片包含 `[分割(a,sep)[index], 分割(b,sep)[index], 分割(c,sep)[index]...]`
- `len()` 返回变量长度参数的长度
- `matchRegex(a,b,c...regex)` 返回是否匹配既定的regex

下面是示例 `example/disallowed_tag/model.yaml`

```
[matchers]
```

假定 `访问Wildcard(r.obj.Request.Object.Object.Spec.Template.Spec.containers , "*", "Image")` 返回 `["a:b", "c:d", "e:f", "g:h"]` 因为分割支持可变长度参数，并拆分操作将应用于每个元素，最终索引为1的元素将被选中并返回，所以分隔

`(accessWithWildcard(r.obj.Request.Object.Object.Spec.Template.Spec.containers , "*", "Image"), ":" , 1)` 得到的是 `["b", "d", "f", "h"]`。并且 `contain(split(accessWithWildcard(r.obj.Request.Object.Object.Spec.Template.Spec.containers , "*", "Image"), ":" , 1), p.obj)` 返回 `p.obj` 是否包含在 `"b","d","f","h"` 中。

4.2.1.2 类型转换功能

- `ParseFloat()`: 解析一个整数为浮点数。(因为在比较时的任何数字都必须转换成浮点数)。
- `ToString()`: 将对象转换为字符串。此对象必须有基本类型的字符串。例如，当语句类型为XXX字符串时，使用XXX型对象。
- `IsNil()`: 返回参数为nil

5. 高级设置

5.1 关于证书

在 k8s 中， webhook 必须使用 HTTPS。有两种办法：

- 使用自签名证书(本版本中的示例使用此方法)
- 使用普通证书

5.1.1 自签名证书

使用自签名证书意味着签发证书的CA不是众所周知的CA，因此，您必须让k8s知道此 CA。

当前这个repo 中的示例使用自制CA，其私钥和证书储存在 `config/ca` 中。`rt` 和 `config/certificate/ca.key`。webhook 的证书是 `config/certificate/server.crt`，由自制CA发出。此证书的域名是 "webhook.domain.local"(用于外部 webhook) 和 "casbin-webhook-svc.default.svc"(内部Webhook)

有关CA的信息通过 webhook 配置文件传递到 k8s。`config/webhook_external.yaml` 和 `config/webhook_internal.yaml` 都有名为"CABundle"的字段，包含 CA 证书的 base64 编码字符串。

如果您需要更改证书/域(例如也许您想要在使用内部Webhook时将此webhook放入另一个k8s的命名空间；或者，您可能想要在使用外部Webhook时更改域名)，应该采取以下程序：

1. 生成新密钥

为假 CA 生成私钥

```
openssl genrsa -des3 -out ca.key 2048
```

移除私钥的密码保护。

```
openssl rsa -in ca.key -out ca.key
```

2. 为 webhook 服务器生成私钥

```
openssl genrsa -des3 -out server.key 2048  
openssl rsa -in server.key -out server.key
```

3. 使用自制的 CA 来签署 webhook 证书

复制您系统的 openssl 配置文件临时使用。您可以使用 `openssl` 版本 `-a` 来查找配置文件的位置，您可以调用 `openssl.cnf`

找到 [req] 段并添加以下行: `req_extension = v3_req`

找到 [req] 段并添加以下行: `subjectAltName = @alt_names`

在文件中附加以下行:

```
[alt_names]  
DNS.2=<The domain you want>
```

“casbin-webhook-svc.default.svc”应该替换为您自己服务的实际服务名称(如果您决定修改服务名称)

使用修改后的配置文件生成证书请求文件

```
openssl req -new -nodes -keyout server.key -out server.csr -config openssl.cnf
```

使用自制CA响应请求并签署证书

```
openssl x509 -req -days 3650 -in server.csr -out server.crt -CA ca.crt -CAkey ca.key  
-CAcreateserial -extensions v3_req -extensions SAN -extfile openssl.cnf
```

3. 替换“CABundle”字段

`config/webhook_external.yaml` 和 `config/webhook_internal.yaml` 都有名为“CABundle”的字段，包含 CA 证书的 base64 编码字符串。

4. 如果您正在使用 helm，需要对 helm 图进行类似的更改。

5.1.2 法律证书

如果您使用合法证书，您就不需要这些程序。删除 `config/webhook_external.yaml` 和 `config/webhook_internal.yaml` 中的“CABundle”字段，并将这些文件中的域更改为您的域。

使用 Envoy 实现 Service Mesh 权限管理

[Envoy-authz](#) 是供 Envoy 通过 Casbin 执行外部 RBAC 和 ABAC 权限管理的中间件。这个中间件通过 gRPC 服务器使用 [Envoy 的外部权限 API](#)。该代理可部署在 Istio 等基于 Envoy 的 Service Mesh 上。

需求

- Envoy 1.17 以上版本
- Istio 或 任意 Service Mesh 类型
- gRPC 依赖项

依赖项通过 `go.mod` 进行管理。

中间件工作原理

- 客户端发起 http 请求
- Envoy 代理发送该请求至 grpc 服务器
- gRPC 服务器基于 Casbin 策略为请求进行授权
- 通过授权后，就会发送请求，否则会被禁止

gRPC 服务器基于 Envoy 中 [external_auth.proto](#) 的通信数据协议（Protocol Buffer，简称为 Protobuf）。

```
// 对传入网络服务的请求
// 执行授权检查的通用接口
service Authorization {
    // 基于传入请求的属性执行授权检查
    // 并返回 `OK` 或非 `OK` 的状态。
    rpc Check(v2.CheckRequest) returns (v2.CheckResponse);
}
```

上面的协议文件中，必须使用权限服务器中的 `Check()` 服务。

用法

- 遵循此[指南](#)，在配置文件中定义 Casbin 策略。

You can verify/test your policies on online [casbin-editor](#).

- 使用以下命令启动授权服务器：

```
$ go build .
$ ./authz
```

- Load the envoy configuration:-

```
$ envoy -c authz.yaml -l info
```

Envoy 启动后，就会开始拦截请求以进行授权处理。

集成至 Istio

为了让此中间件顺利运行，需要发送 JWT 的 Token 中包含用户名信息的自定义请求

头。更多修改请求头的内容，请参阅 [Istio 官方文档](#)。



>

管理

管理



Admin Portal

Casbin 管理门户网站



Casbin服务

使用 Casbin 作为服务



日志 & 错误处理

Casbin 日志 & 错误处理

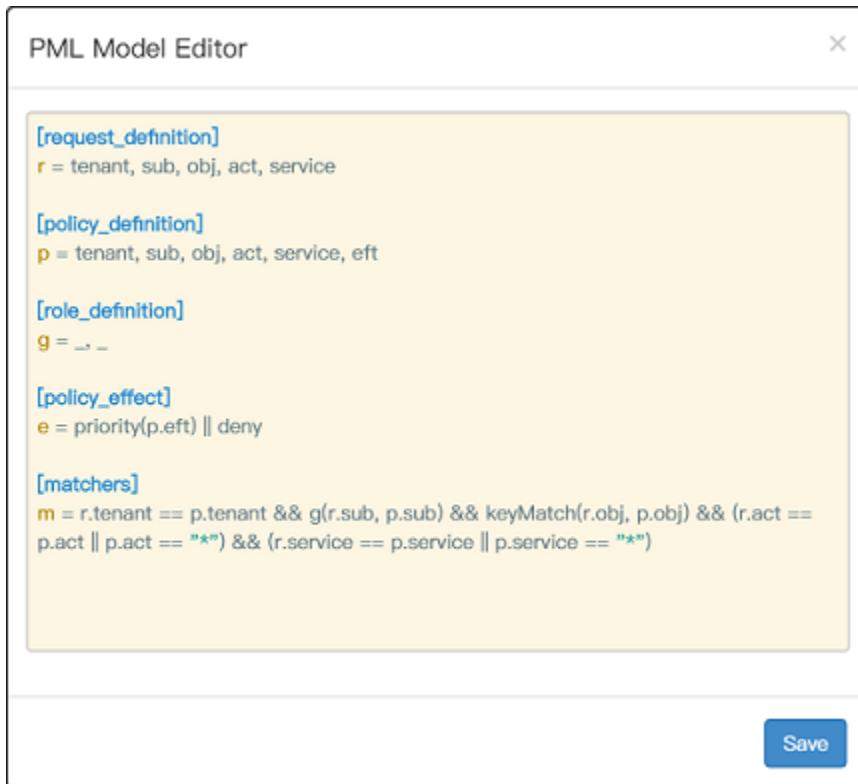


前端使用

Casbin.js是一个能够帮助你在前端应用中管理访问控制权限的Casbin前端版本。

Admin Portal

我们提供了一个用于模型管理和策略管理的门户网站 [Casdoor](#)



PML Policy Editor

Tenant List / Policy Tree / PML Policy Editor

Welcome, Company A

Policy List

Rule Type	Tenant	User	Resource Path	Action	Service	Auth Effect	Option
p	tenant1	admin1	/*	*	*	allow	
p	tenant1	user12	/*	*	nova	allow	
p	tenant1	user13	/*	*	glance	allow	
g	user11	admin1					

Save

还有第三方门户管理项目，使用Casbin作为授权引擎。您可以基于这些项目建立您自己的 Cabin 服务。

[Go](#) [Java](#) [Node.js](#) [Python](#) [PHP](#)

Project	作者	前端	后端	描述
Casdoor	Casbin	React + Ant Design	Beego	基于 Beego + XORM + React
go-admin-team/go-admin	@go-admin-team	Vue + Element UI	Gin	go-admin 基于 Gin + Casbin + GORM
gin-vue-admin	@piexlmax	Vue + Element	Gin	基于 Gin + GORM + Vue

Project	作者	前端	后端	描述
		UI		
gin-admin	@LyricleTian	React + Ant Design	Gin	RBAC脚手架基于Gin + GORM + Casbin + Ant设计反应
go-admin	@hequan2017	无	Gin	基于Gin + GORM + JWT + RBAC (Casbin) 的 RESTful API网关
zeus-admin	bullteam	Vue + Element UI	Gin	基于JWT + Casbin的统一权限管理平台
IrisAdminApi	@snowlyg	Vue + Element UI	Iris	基于 Iris + Casbin 的 后端 API
Gfast	@tiger1103	Vue + Element UI	Go Frame	基于 GF (Go Frame) 的管理门户网站
echo-admin (前端, 后端)	@RealLiuSha	Vue 2.x + Element UI	Echo	基于 Echo + Gorm + Casbin + Uber-FX 的管理门户网站
Spec-Center	@atul-wankhade	无	Mux	基于 Casbin + MongoDB 的 Golang RESTful 平台

Project	作者	前端	后端	描述
spring-boot-web	@BazookaW	无	SpringBoot	基于 SpringBoot 2.0 + MyBatisPlus + Casbin 的管理门户网站
Project	作者	前端	后端	描述
node-mysql-rest-api	@JoemaNequinto	无	Express	一个使用 Express, Sequelize, JWT 和 Casbin 在Node.js 建造RESTful APIs 微型服务的应用。
Casbin-Role-Mgt-Dashboard-RBAC	@alikhan866	React + Material UI	Express	带有Enforcer可以检查过程中执行结果且对新手友好的RBAC 管理
Project	作者	前端	后端	描述
fastapi-mysql-generator	@CoderCharm	无	FastAPI	FastAPI + MySQL + JWT + Casbin
FastAPI-MySQL-Tortoise-Casbin	@xingxingzaixian	无	FastAPI	FastAPI + MySQL + Tortoise + Casbin

Project	作者	前端	后端	描述
openstack-policy-editor	Casbin	Bootstrap	Django	Casbin 的 Web 界面
Project	作者	前端	后端	描述
Tadmin	@techoner	AmazeUI	ThinkPHP	基于 ThinkPHP 5.1+ 的非侵入式后端框架
video.tinywan.com	@Tinywanner	LayUI	ThinkPHP	基于 ThinkPHP5 + ORM + JWT + RBAC (Casbin) 的 RESTful API 网关
laravel-casbin-admin	@pl1998	Vue + Element UI	Laravel	基于vue-element-admin和 Laravel 的 RBAC 许可管理系统
larke-admin (前端, 后端)	@deatil	Vue 2 + Element UI	Laravel 8	基于 Laravel 8, JWT 和 RBAC 的门户网站
hyperf-vuetify-admin	@TragicMale	Vue + Vuetify	Hyperf	基于 Hyperf、Vuetify 和

Project	作者	前端	后端	描述
		2.x		Casbin的管理 门户网站

Casbin服务

如何使用Casbin作为服务？

名称	描述
Casbin服务	基于gRPC的官方 Casbin as a Service，提供了Management API 和 RBAC API。
PaySuper Casbin Server	PaySuper的上述官方 Casbin-Server 分支, 但是维护更加积极。它为 Casbin 授权提供 go-micro 接口。
middleware-acl	基于 Casbin 的 RESTful 访问控制插件。
Buttress	基于Casbin访问控制的服务解决方案。
auth-server	验证服务器校对服务。

日志 & 错误处理

日志

Cabin默认使用内置的 `log` 来将日志输出到控制台，如：

```
2017/07/15 19:43:56 [Request: alice, data1, read ---> true]
```

日志记录不是默认启用的。您可以通过调用 `Enforcer.EnableLog()` 或 `NewEnforcer()` 函数中的最后一个参数来切换它。

备注

我们已经支持日志模型、强制请求、角色、Golang策略。您可以定义您自己的日志来记录Casbin。如果您正在使用 Python, pycasbin 会影响默认的 Python 日志机制。Pycasbin 软件包调用`logging.getLogger()`来设置日志。除了初始化父应用程序中的日志记录器外，不需要特殊配置的日志。如果父应用程序内没有输入日志，您将不会看到来自pycasbin的日志消息。

对不同的执行器使用不同的记录器

每个执行器都可以有自己的记录器来记录信息，并且可以在运行时进行更改。

而且你可以通过 `NewEnterer()` 的最后一个参数使用一个适当的日志。如果您使用这种方式来初始化您的执行器，由于日志中启用的字段的优先级更高，您不需要使用启用参数。

```

// 设置默认记录器作为执行器e1的记录器。
// 此操作也可以被视为在运行时更改e1的记录器。
e1.SetLogger(&Log.DefaultLogger{})

// 设置另一个记录器作为执行器e2的日志记录器。
e2.SetLogger(&YouOwnLogger)

// 初始化执行器e3时设置您的记录器。
e3, _ := casbin.NewEnforcer("examples/rbac_model.conf", a,
logger)

```

支持的记录器

我们提供了一些记录器来帮助您记录信息。

[Go](#) [PHP](#)

记录器	作者	描述
Defatule logger (内置)	Casbin	默认使用golang日志。
Zap logger	Casbin	使用 zap , 提供json 编码日志, 您可以使用自己的 zap-logger 自定义更多信息。

记录器	作者	描述
psr3-bridge 记录器	Casbin	提供一个 PSR-3 兼容桥。

如何编写一个记录器

您的记录器应该实现 [Logger](#) 接口。

接口名	实现要素	描述
EnableLog()	必须	控制是否打印消息。
IsEnabled()	必须	显示当前日志启用的状态。
LogModel()	必须	与模型相关的日志信息。
LogEnforce()	必须	与执行器相关的日志信息。
LogRole()	必须	与角色相关的日志信息。
LogPolicy()	必须	与策略相关的日志信息。

您可以将您的自定义 [记录器](#) 传给 [Enforcer.SetLogger\(\)](#) 函数。

这是一个关于如何自定义Golang日志的示例：

```
import (
    "fmt"
    "log"
    "strings"
)

// 默认日志是使用golang日志的日志实现的。
type DefaultLogger struct {
    enabled bool
```

错误处理

当您使用Casbin时可能会由于以下原因发生错误或恐慌：

1. Model文件 (.conf) 中的语法有误。
2. 策略文件(.csv)中语法有误。
3. 来adapter的自定义错误信息（譬如连接MySQL失败）。
4. Casbin的bug。

您可能需要五个主要功能来处理错误或恐慌：

Function	异常时表现
NewEnforcer()	返回error
LoadModel()	返回error
LoadPolicy()	返回error
SavePolicy()	返回error
Enforce()	返回error

① 备注

NewEnforcer() 通过内部调用 LoadModel() 和 LoadPolicy()。所以当您使用 NewEnforcer() 函数时不需要再去调用这两个函数。

启用 & 禁用

可以通过 `Enforcer.EnableEnforce()` 函数禁用执行器。当它被禁用时, `Enforcer.Enforce()` 将总是返回 `true`。诸如添加或删除政策之类的其他操作将不受影响。下面是一个示例:

```
e := casbin.NewEnforcer("examples/basic_model.conf", "examples/basic_policy.csv")

// 将会返回false
// 默认情况下enforcer是启用的
e.Enforce("non-authorized-user", "data1", "read")

// 在运行时禁用enforcer
e.EnableEnforce(false)

// 对任何请求都返回true
e.Enforce("non-authorized-user", "data1", "read")

// 打开enforcer
e.EnableEnforce(true)

// 将会返回false
e.Enforce("non-authorized-user", "data1", "read")
```


前端使用

[Casbin.js](#)是一个能够帮助你在前端应用中管理访问控制权限的Casbin前端版本。

安装

```
npm install casbin.js  
npm install casbin
```

或者

```
yarn add casbin.js
```

前端中间件

中间件	类型	作者	描述
react-authz	React	Casbin	Casbin.js 的 React 包装器
rbac-react	React	@daobeng	React 基于角色的访问控制, 使用 HOCs, CASL 和 Casbin.js
vue-	Vue	Casbin	Casbin.js 的 Vue 包装器

中间件	类型	作者	描述
authz			
angular-authz	Angular	Casbin	Casbin.js的Angular包装器

快速入门

您可以在您的前端应用程序中使用 `manual` 模式，并随时设置权限。

```
const casbinjs = require("casbin.js");
// 设置用户权限
// 他/她可以可以读取data1和data2并且可以写入data1
const permission = {
  "read": ["data1", "data2"],
  "write": ["data1"]
}

// 在manual模式使用Casbin.js需要您手动设置权限
const authorizer = new casbinjs.Authorizer("manual");
```

现在我们有了一个授权者 `authorizer`。我们可以通过使用API `authorizer.can()` 和 `authorizer.cannot()` 获得得许可规则。这2个API的返回值是JavaScript Promise ([详细信息](#)) 所以我们应该使用 `then()` 返回值的方法，例如：

```
result = authorizer.can("write", "data1");
result.then((success, failed) => {
  if (success) {
    console.log("you can write data1");
```

和 `cannnot()` 以同样方式使用：

```
result = authorizer.cannot("read", "data2");
result.then((success, failed) => {
    if (success) {
        console.log("you cannot read data2");
    } else {
        console.log("you can read data2");
    }
});
// 输出：您可以读取data2
```

在上面的代码中，变量 `success` 意味着请求获得结果而不产生错误，而不意味着权限规则是 `true` `failed` 也和权限规则无关 只有在请求过程中出现错误时才有意义。

您可以参考我们的[React示例](#)来查看Casbin.js的实际用法。

高级用法

Casbin.js提供了一个完美的解决方案来将您的前端访问控制管理和后端Casbin服务一体化。

在初始化 Casbin.js `Authorizer` 时使用 `auto` 模式并指定你的后端地址，它会自动同步权限并调整前端状态。

```
const casbinjs = require('casbin.js');

// 设置您的后端Casbin服务url
const authorizer = new casbinjs.Authorizer(
    'auto', // 模式
    {endpoint: 'http://your_endpoint/api/casbin'}
);
```

因此，您需要开放一个接口(例如一个 RestAPI)来创建权限对象并将其返回前端。在你的 API 控制器中，调用 `CasbinJsGetUserPermission` 以创建权限对象。下面是一个 Beego 框架的示例：

① 备注

注意您的端点服务器应该返回类似的内容

```
{  
    "other": "other",  
    "data": "What you get from  
`CasbinJsGetPermissionForUser`"  
}
```

```
// 路由器  
beego.Router("api/casbin", &controllers.APIController{},  
"GET:GetFrontendPermission")  
  
// 控制器  
func (c *APIController) GetFrontendPermission() {  
    // 在 GET 请求的参数中获取访客。 (其中的键是"casbin_subject")  
    visitor := c.Input().Get("casbin_subject")  
    // `e` 是一个初始化的 Casbin Enforcer 实例  
    c.Data["perm"] = casbin.CasbinJsGetPermissionForUser(e,  
    visitor)  
    // 将数据传到前端  
    c.ServeJSON()  
}
```

① 备注

目前，`CasbinJsGetPermissionForUser` api 仅在 Go Casbin 和 Node-Casbin 中被支持。如果您希望这个api支持其它语言，请在此[提交issue](#) 或者留

下评论。

API 列表

setPermission(permission: string)

设置权限对象。 始终在 `manual` 模式中使用。

setUser(user: string)

设置访客身份并更新权限。 始终在 `auto` 模式中使用。

can(action: string, object: string)

检查用户是否能对 `object` 执行 `action`。

cannot(action: string, object: string)

检查用户是否不能对 `object` 执行 `action`。

canAll(action: string, objects: Array<object>)

检查用户是否能在 `objects` 对所有对象执行 `action`。

canAny(action: string, objects: Array<object>)

检查用户是否能对 `objects` 中的任意一个执行 `action`。

为什么选择 Casbin.js

人们可能会想知道Node-Casbin和Casbin.js之间的区别。总的来说，Node-Casbin 是

一个用 NodeJS 环境实现的 Casbin 核心，它通常在服务端用作一个访问控制工具包。Casbin.js 是一个能帮助您在客户端使用 Casbin 为您的网页里的用户授权的前端库。

通常，由于以下问题，直接构建一个 Casbin 服务并在网页前端执行授权/执行是不妥当的：

1. 当有人启动客户端时，执行器会被初始化，随后在后端持久层中拉取所有策略。高并发会为数据库带来巨大的压力并带来极高的网络成本。
2. 将所有策略悉数加载进客户端会带来安全风险。
3. 区分客户端和服务器以及灵活的开发有困难。

我们希望有一种能够简化 Casbin 前端开发的工具。实际上，Casbin.js 的核心是在客户端操纵当前用户的权限。正如你提到的，Casbin.js 从一个指定的后端获取数据。这个程序会与 Casbin 后端服务同步权限。取得权限数据之后，开发者可以使用 Casbin.js 提供的接口来在前端管理用户行为。

Casbin.js 避免了以上提及的两个问题：Casbin 服务将不再被反复请求数据，并且减少了客户端与服务端之间传递数据量。我们也避免了将所有的策略都储存在前端的问题。用户仅能操作与之相关的权限，对其他诸如访问控制模型、其他用户的权限的内容将一无所知。此外，Casbin.js 还能有效地在授权管理方面分离客户端和服务端。



>

编辑器

编辑器

**Online Editor**

在 web 浏览器中写入 Casbin 模型和策略

**IDE 插件**

Casbin 的 IDE 插件

Online Editor

You can also use the [online editor](#) to write your Casbin model and policy in your web browser. 它提供了一些比如 [语法高亮](#) 以及 [代码补全](#) 这样的功能，就像编程语言的 IDE一样。

使用模式

如果您使用 [RBAC与模式](#) 或 [RBAC与所有模式](#)，它指定了左下角的匹配函数。

The screenshot shows the Casbin code editor with some syntax highlighting and a red arrow pointing to the 'keyMatch' section. To the right, there is a 'Request' panel showing three items: 1 /book/, 2 /book/, and 3.

```
12      *
  matchingDomainForGFunction:
  'keyMatch'
13      */
14      matchingForGFunction:
  'keyMatch2',
15
  matchingDomainForGFunction:
  'keyMatch2'
16  };
17 })();
```

Request

```
1 /book/
2 /book/
3
```

如果您想编写等效代码，您需要通过相关api指定模式匹配函数。 使用模式查看 [RBAC](#)

ⓘ 备注

编辑器基于 [node-casbin](#)。由于casbin不同语言之间的同步延迟，[编辑器](#)的身份验证结果可能与您正在使用的casbin的身份验证结果不同。如果是，请将问题提交给您正在使用的casbin仓库。

IDE 插件

我们有这些IDE插件：

JetBrains

- 下载: <https://plugins.jetbrains.com/plugin/14809-casbin>
- 源代码: <https://github.com/will7200/casbin-idea-plugin>

VSCode (正在制作)

- 源代码: <https://github.com/casbin/casbin-vscode-plugin>



>

更多

更多

使用者

Casbin的采用者

参与贡献

为 Casdoor 做贡献

隐私政策

Casbin 网站隐私政策

服务条款

Casbin 服务条款

使用者

直接集成

[Go](#) [Java](#) [Node.js](#) [Python](#)

名称	描述	模型	策略
VMware Harbor	VMware的开源可信云本地注册表项目，用于存储、签名和扫描内容。	Code	Beego ORM
Intel RMD	英特尔的资源管理守护进程。	.conf	.csv
VMware Dispatch	用于部署和管理无服务器风格应用程序的框架。	Code	Code
Skydive	一个开源的实时网络拓扑和协议分析器。	Code	.csv
Zenpress	用Golang编写的CMS系统。	.conf	Gorm
Argo CD	为Kubernetes持续提供的GitOps。	.conf	.csv
Muxi Cloud	一种更容易管理Kubernetes集群的方法。	.conf	Code
EngineerCMS	CMS管理工程师的知识。	.conf	SQLite
Cyber Auth	一个Golang身份验证API项目。	.conf	.csv

名称	描述	模型	策略
API			
IRIS Community	IRIS社区活动网页	.conf	.csv
Metadata DB	BB归档元数据数据库。	.conf	.csv
Qilin API	游戏内容下的ProtocolONE许可证管理工具。	Code	.csv
Devtron Labs	Kubernetes软件发送Workflow。	.conf	Xorm

名称	描述	模型	策略
lighty.io	OpenDaylight的SDN控制器解决方案。	README	无

名称	描述	模型	策略
Notadd	基于Nest.js的微服务开发架构。	.conf	DB adapter

名称	描述	模型	策略
dtrace	EduScaled的跟踪系统。	Commit	无

通过插件集成

名称	描述	插件	模型	策略
Docker	全球领先的软件容器平台	casbin-authz-plugin (Docker 推荐)	.conf	.csv
Gobis	用go编写的Orange的轻量级API网关	casbin	Code	请求

参与贡献

Casbin 是一个支持访问控制模型的，强大的授权库，支持使用多种语言，只要你擅长一种语言，你就可以参与Casbin的开发。始终欢迎您为项目作出贡献。

目前，主要有两类项目。

- <偏重算法——第一类项目是不同语言实现的与算法相关的项目，包括大部分主流编程语言，从Golang、Java、C++到Elixir、Dart和Rust及其外围产品。

Casbin	jCasbin
可用于生产环境	可用于生产环境



PyCasbin	Casbin.NET
可用于生产环境	可用于生产环境

- 面向应用程序——第二类项目是与应用程序相关的项目。

Project	Demo	详情	技能栈...
Casdoor	Casdoor	Casdoor 是基于 OAuth 2.0 / OIDC 的 UI 首次集中身份验证/单点登录(SSO) 平台	JavaScript + React 和 Golang + Beego + SQL
Casnnode	Casbin 论坛	Casnnode 是下一代的论坛软件	JavaScript + React 和 Golang + Beego + SQL
Casbin-OA	OA system	Casbin-OA是Casbin技术作者的官方手稿处理、评价和显示系统。	JavaScript + React 和 Golang + Beego

Project	Demo	详情	技能栈...
			+ MySQL
Casbin Editor	Casbin Editor	Casbin Editor是一个在线的 Casbin 模型和策略编辑器	TypeScript + React

加入我们

为Casbin作出贡献的方式多种多样，可以从以下几方面入手：

- **使用Casbin 并报告问题！** 当使用Casbin时，报告问题以促进Casbin的发展，不论是缺陷还是建议。在 GitHub 上提交问题之前，最好先在 [Gitter](#), [Casbin Forum](#), [Google group](#) 或 [QQ group: 546057381](#)上讨论。

注意：在报告问题时，请使用英文描述您问题的细节。
- **帮助优化文档！** 从文档开始贡献是参与贡献的良好开端。
- **帮助解决问题！** 我们准备了一个适合初学者参与的简单任务的表格，有不同级别，不同标签的挑战，[在此查看表格](#)。

合并请求

Casbin使用GitHub 作为其开发平台。因此，合并请求是贡献的主要来源。

在您打开拉取请求之前，您需要知道一些基本准则：

- 解释您为什么要发送此 PR 以及此 PR 将会在仓库中做些什么。

- 请确保每个PR 只做一件事， 否则请分开提交。
- 如果有新添加的文件， 请将Casbin许可证添加到新文件的顶部。

```
// Copyright 2021 The casbin Authors. All Rights Reserved.  
//  
// Licensed under the Apache License, Version 2.0 (the  
"License");  
// you may not use this file except in compliance with the  
License.  
// You may obtain a copy of the License at  
//  
//     http://www.apache.org/licenses/LICENSE-2.0  
//  
// Unless required by applicable law or agreed to in  
writing, software  
// distributed under the License is distributed on an "AS  
IS" BASIS,  
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either  
express or implied.  
// See the License for the specific language governing  
permissions and  
// limitations under the License.
```

- 在向Casdoor, Casnodel 和 Casbin OA提交时，您可能需要搭建演示以向维护者证明您的PR能帮助到项目。
- 当您打开PR 并提交您的贡献时，最好是使用Conventional Commits， 格式：
`<type>(<scope>): <subject>`, `<scope>` 是可选的。欲了解更详细的用法，请参阅 Conventional Commits

许可协议

为Casbin作出贡献，即代表您同意您的贡献将遵循 Apache 许可协议。

隐私政策

您的隐私对我们很重要。 It is Casbin's policy to respect your privacy regarding any information we may collect from you across our [docs website](#), and other sites we own and operate.

我们只是在我们真正需要你的信息向你提供服务时才要求你提供您的个人信息。 我们在你知情和同意的情况下，以公正和合法的方式加以收集。 我们还告诉你我们为什么要收集以及如何使用它。

我们只在必要的时候保留您的信息来为您提供您请求的服务 无论我们保留了什么信息，我们将会在商业上可以接受的手段保护它以防止丢失和被盗取，以及未经授权的访问、披露、复制使用或修改

我们不公开或与第三方分享任何个人信息，除非法律有此要求。

我们的网站可能会链接到不属于我们管理的外部网站。 请注意，我们对这些网站的内容和做法没有控制权。 而且不能接受其各自的隐私政策的责任或赔偿责任。

您可以自由拒绝我们的个人信息请求。 我的理解是，我们可能无法向你提供你所需要的一些服务。

你对我们的网站的继续使用将被视为接受我们有关隐私和个人信息的做法。 如果您有任何关于我们如何处理用户数据和个人信息的问题，请随时联系我们。

这项政策自2020年6月29日起生效。

服务条款

1. 条款

By accessing the website at <https://casbin.org>, you are agreeing to be bound by these terms of service, all applicable laws and regulations, and agree that you are responsible for compliance with any applicable local laws. 如果您不同意其中的任何一条，那您将没有权限访问或使用我们的网站。此网站的所有资源均受到相应版权和商标法的保护。

2. 使用许可证

1) 当您从 Casbin 官网下载资料副本（信息或者软件）时，我们将授予您一份临时许可，仅供您个人的非商业阅览。我们仅授权您使用，但并不将产权转让给您。基于此条款，您不可以：

- i. 修改或复制此资料
- ii. 将此资料商用，或者用于公开展示（无论是否商用）；
- iii. 尝试反编译，或反工程化 Casbin 网站上的任何软件。
- iv. 从资料中移除任何版权或所有权标记
- v. 将此资料转移给其他人，或在其他服务器上为此资料创建镜像

b. 如果您违反了其中的任何限制，本条款将自动终止。同时，Casbin 随时可以终止本条款。当您阅览完这些资料，或者条款终止后，您必须删除并销毁包括电子版和印刷版在内的所有 Casbin 资料。

3. 免责声明

a. Casbin 网站上的资料均按其现状提供。Casbin 不做任何明示或暗示的保证，并且在此否定其他所有保证（包括但不限于默示保证或适销性条件、特定用途的适用性，不侵犯

知识产权或其他侵犯权利的情况）。

b. 此外，Casbin 不保证本网站、指向本站的其它网站或其他提及此资源的网站的正确性和可靠性，也不对可能导致的结果负责，并且不对此进行任何声明。

4. 限制

在任何情况下，Casbin 及其提供者不对任何使用或不能使用 Casbin 网站资料造成的损害（包括但不限于数据或收益损失、业务中断）负责，不论您是否已经以口头或书面形式通知了 Casbin 或 Casbin 的授权代表。由于某些法域不允许对默示担保加以限制（或限制后果性或附带损害的责任），那么这些限制可能不适用于你。

5. 资料的准确性

Casbin 官网的资料可能包含技术、排版或摄影错误。Casbin 不保证官网资料的准确性、完整性或时效性。Casbin 随时可以在不予通知的情况下修改网站内的资料。并且 Casbin 也没有作出任何更新资料的承诺。

6. 链接

Casbin 没有对任何指向 Casbin 的网站进行审查，并且对这些网站的内容不负任何责任。引用任何链接均不代表 Casbin 认同该网站。使用任何上述链接时，您需要自行承担风险。

7. 修改

Casbin 随时可以在不予通知的情况下修改这些条款。使用这个网站意味着您认同当前的服务条款。

8. 管辖法

这些条款受加利福尼亚州旧金山法律管辖并根据其解释，您不可撤销地服从于该国/该地所在法院的专属管辖权。