API 业务权限需求分析

根据 API 面向的请求主体性质不同，API 业务权限可以分成 3 类：

1. 面向终端用户的 API 业务权限 —— 简称 User Permissions 用户权限

请求主体是终端用户，一般是 **user** 人，也可能是 **agent** 机器人，或 **dev** 设备。资源整体归属于租户，但可能基于各种层级的容器进行权限管控。用户能访问的资源往往会跨越租户，用户在一个租户中访问某一类资源往往还会受到更多关系约束。这就是一般业务应用的权限需求。

以协作场景下的项目和任务为例，用户并不能访问租户下的任意项目、任务。用户能访问的项目可能是：他参与的项目，租户下公开的项目，或者因为某些授权链路关系继承了权限的项目，如部门的 PM 可以访问部门名下所有的项目，即便他不是相关项目成员。用户能访问的任务可能是：分配、执行或订阅的任务，参与的项目下的任务，因为某些授权链路关系继承了权限的项目下的任务。
用户对于文件夹、文件的访问权限同样也会有复杂的关系约束，在文件夹上定义的权限还涉及继承或中断，如果再允许文件夹归属多个管理容器（如空间、项目），并在容器上能定义用户权限，那么权限计算的复杂度就非常恐怖了。

试想一个典型场景是用户希望跨租户搜索任务、文件，即带权限过滤的资源搜索，该如何高效的实现？

2. 面向后端应用服务、资源归属于租户的 API 业务权限 —— 简称 Application Permissions 应用权限

请求主体是 **app** 业务应用或者 **svc** 服务，资源归属于租户。如果 app 或 svc 以用户身份在后端调用其它服务的 API，那么涉及的权限逻辑与 1 中的描述一致，Microsoft Graph 中称之为 **Delegated Permissions 委派权限**，我们可以把这类请求主体归类为 1 中描述的 **agent**。
我们再看 app 或 svc 以它自身的身份调用其它服务的 API，Microsoft Graph 中称之为 **Application Permissions 应用权限**。app 或 svc 通过 API 操作资源不但要取得 API 提供方的授权，还需要取得租户的授权。

业务开放平台中面向应用开放的 API 通常是这样一个场景，开发者开发应用时要申请必要的 API 资源权限，租户管理员安装应用时要授权应用访问本租户的对应资源。应用虽然获得了 API 资源权限，但它不能访问未经授权的租户的资源。

应用权限比用户权限逻辑简单了很多。

3. 面向后端应用服务、资源归属单一的 API 业务权限 —— 简称 Service Permissions 服务权限

请求主体是 **app** 业务应用或者 **svc** 服务，资源要么属于 API 提供方且共享，要么属于 app 或 svc 自身且只能访问自己的资源。API 可能会通过权限进一步限制只能访问一部分数据，或者数据的部分字段。

以企业通讯录服务为例，通讯录资源属于企业，企业名下开发的 app 或 svc 都能访问，但可能会限制某些 app 只能访问指定部门的通讯录，或者限制 app 只能访问通讯录的部分字段。
以阿里云中间件服务为例，开发者申请了一个 OSS 资源实例（租户），OSS 资源属于开发者，开发者的 app 对该实例有全部访问权限，或者配置局部权限，开发者的 app 不能访问别人的 OSS 资源。

服务权限比应用权限逻辑简单了很多。

我们在设计 API 时，一定要明确面向的请求主体和服务的资源的归属，采用合适的权限方案，其中用户权限方案最为复杂，有些需求可能会复杂到暂时无法实现。
我们可以使用什么样的权限方案呢：
RBAC(Role-Based Access Control)?
ABAC(Attribute-Based Access Control)?
LBAC(Label-Based Access Control)?
以上三个都可能满足简单的权限需求，但无法满足复杂的用户权限需求。我推荐一种新的方案：
GBAC(Graph-Based Access Control)，Wikipedia 上已经有相关 条目描述，但不完善（不要被它误导），本文最后会给出一个可行的 GBAC 设计框架。

鉴权执行点

鉴权执行点通常分布在 3 个位置：

1. 应用客户端鉴权

应用客户端应该根据当前登录用户的权限情况来渲染 UI 界面，从而带来最好的用户体验。比如任务详情页要不要显示编辑按钮或编辑模式，要不要显示评论框，任务菜单要不要显示归档选项，项目配置页面应该显示哪些配置项等。

用户权限变动也会相对比较频繁，比如部门变动、角色调整、文件夹移动等都引发权限链路的变化，

因为权限继承等原因一个小小的调整可能会影响一波庞大的用户群体。更复杂的是这个调整可能并不会影响某些用户，因为他们从另外的权限链路也获得了权限。

一个用户能不能修改一个任务？能不能在任务表格视图中批量修改一组任务？最好的用户体验肯定是任务应用应该读取用户对各条任务的最新权限，来渲染这个任务表格视图。这会要求应用后端提供足够细粒度的业务权限查询 API。

2. API 网关鉴权

有一些带业务属性的 API 网关，可以验证请求主体身份，并鉴定请求主体对相关请求的权限。
对于上述的服务权限场景，API 网关鉴权能够满足需求；
对于上述的应用权限场景，API 网关鉴权也能满足需求， 一方面要鉴定请求主体对相应 API 有访问权限，另一方面要确保目标资源所归属的租户已对请求主体授权，一般做法是租户授权信息体现在访问令牌中，这就要求请求主体使用不同的访问令牌来访问对应租户的资源。
对于上述的用户鉴权场景，除非权限逻辑特别简单，否则不适合让 API 网关来完成鉴权。

3. API 服务鉴权

API 服务应该对请求进行完整鉴权，除非是上述的服务权限场景确认 API 网关能完成鉴权。
鉴权通常是在完成请求主体身份验证、请求数据验证之后。

权限命名

权限是资源提供方与资源消费方的一种约定，好的权限命名规则，应该是简洁易懂，又极具扩展性的。我们参考了 Microsoft Graph，权限名称遵循模式：

Resource.Operation.Constraint

比如：

User.Get 读取一条正常用户数据
User.Get.BasicInfo 读取一条精简的用户数据，可能只包含 id, name 等必要信息，不包括 email, phone_number 等敏感字段
User.Get.All 读取一条完整用户数据，可能包含密码等敏感信息
User.List 读取一组用户数据
User.List.BasicInfo 读取一组精简的用户数据，可能只包含 id, name 等必要信息，不包括 email, phone_number 等敏感字段

Mail.List.Me 读取“我”的邮件列表
Todo.List.Me 读取“我”的待办列表
Contacts.List.Me 读取“我”的联系人列表
Contacts.List.MeBasicInfo 读取“我”的精简的联系人列表，只包含 id, name 等必要信息，可用于渲染选人组件

Task.Update.Title 更新任务标题
Task.Update.Executor 更新任务执行者

Folder.Create 创建文件夹
Folder.Add.File 文件夹下添加文件

Resource，资源名称，如 User, Organization, Group, Project, Device, Application, Folder, File, Task, Event, Permission 等，单数，首字母大写，复合资源名称则遵循驼峰规则。

Operation，权限操作，如 Create, Add, Update, Delete, Get, List, Share, Send, Quit, Subscribe, Follow 等。

Constraint，权限约束，如 All, BasicInfo, Executor, Level0, Level1, Level2 等，可以不指定，完全由业务方约定，用于进一步约束（收缩）权限作用的资源范围。与 Microsoft Graph 的有所不同，本处 Constraint 用于约束资源名下的子资源或者资源字段等，而 Microsoft Graph 中权限约束 **All** 用于描述租户下的所有该类型资源，扩大了权限作用的范围。

权限名由业务服务自行定义，应该命名简洁，权限边界清晰，保持原子性，避免定义 **ReadWrite** 这种复合操作权限，原则是一项命名权限能够承载一个 API 操作的鉴权，不需要 API 服务组合多项权限来鉴权。比如若同时定义了 **User.Read** 和 **User.ReadWrite**，则读取用户这个 API 需要查询两项权限来鉴定请求主体有没有权限，虽然可以实现，但增加了复杂度。

GBAC 权限模型设计

概述

Graph 是描述各种实体真实关系的最简洁有效的方式，GBAC(Graph-Based Access Control) 正是借助了 Graph 这个特性，可以用于实现几乎任意复杂的权限需求。GBAC 模型框架本身是非常简单的，基于它能够实现标准的 RBAC、ABAC、LBAC 等权限模型，也能覆盖上述的服务权限、应用权限、用户权限等场景，大家不要看到 Graph 就认为它会很复杂。

实现 GBAC 权限模型的关键是关系链路查询能力，它决定了权限查询性能，所以一个高效的 Graph 数据库才能支撑起复杂的 GBAC 实现。但如果权限逻辑比较简单（链路关系很浅），用传统的 SQL 或 NoSQL 数据库也能实现。总之，取决于业务对权限需求的复杂度。

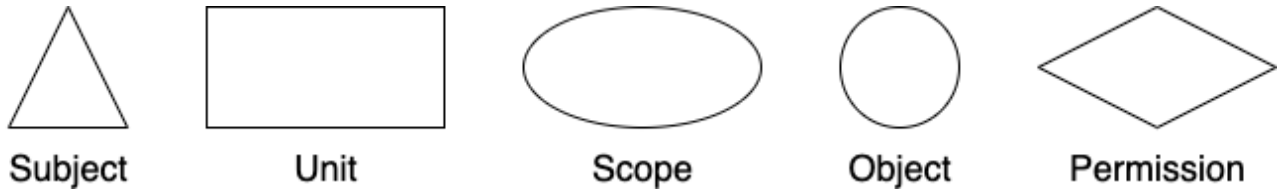
GBAC 定义了一组基本的节点类型和有向边的关系约束，确保它的 Graph 有规则可循是可计算的，而不是任意复杂的。

本 GBAC 设计受到了国际信息技术标准委员会 INCITS 565-202x - Next Generation Access Control 的启发，感兴趣的同学可以阅读 NGAC 标准的原文。

基本术语 Basic Terms

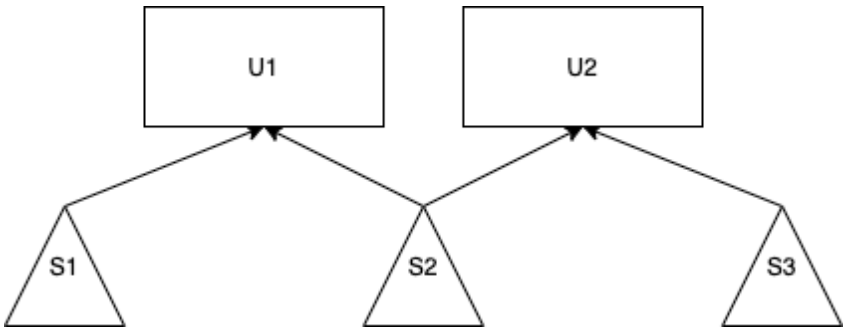
Subject —— 请求主体，可以是 user 用户、dev 设备、agent 机器人代理、app 应用、svc 服务等。它是权限申请方、资源消费方；
Unit —— Administrative Unit，管理单元，可以是角色、项目、空间、部门、群组、企业等。它是一种容器，一方面圈定请求主体，另一方面被设置权限；
Scope —— Scope constraint，范围约束，一般是租户。它是最顶层的约束，并且它不是必须的，只有当权限模型中涉及到租户时才有必要存在，比如以租户为单位安装、授权应用，又比如群组在租户内公开可见；
Object —— 资源对象，可以是任何资源对象，即被消费的资源数据，它也可能是一种容器资源对象，容器中拥有各种类型的子资源对象；
Permission —— 权限，由资源提供方（即 API 服务）定义，是一组常量。它可以被设置在 Unit 上，用于描述管理单元具有的权限；也可以被设置在 Object 上，基于白名单的机制来描述树形资源结构下权限继承的中断，详见后面的场景推演示例。

在后续的 Graph 示意图中，将分别用三角形、长方形、椭圆形、圆形、菱形来描述这几种节点：



关系约定 Relationship Conventions

GBAC 中的 Graph 都是 Directed Acyclic Graph(**DAG**)，各个基本类型节点可以建立的有向边如下：
1. Subject 只能指向 Unit，一个 Subject 可以指向多个 Unit，表示 Subject 被这些 Unit 管理，获得了相关权限关系。

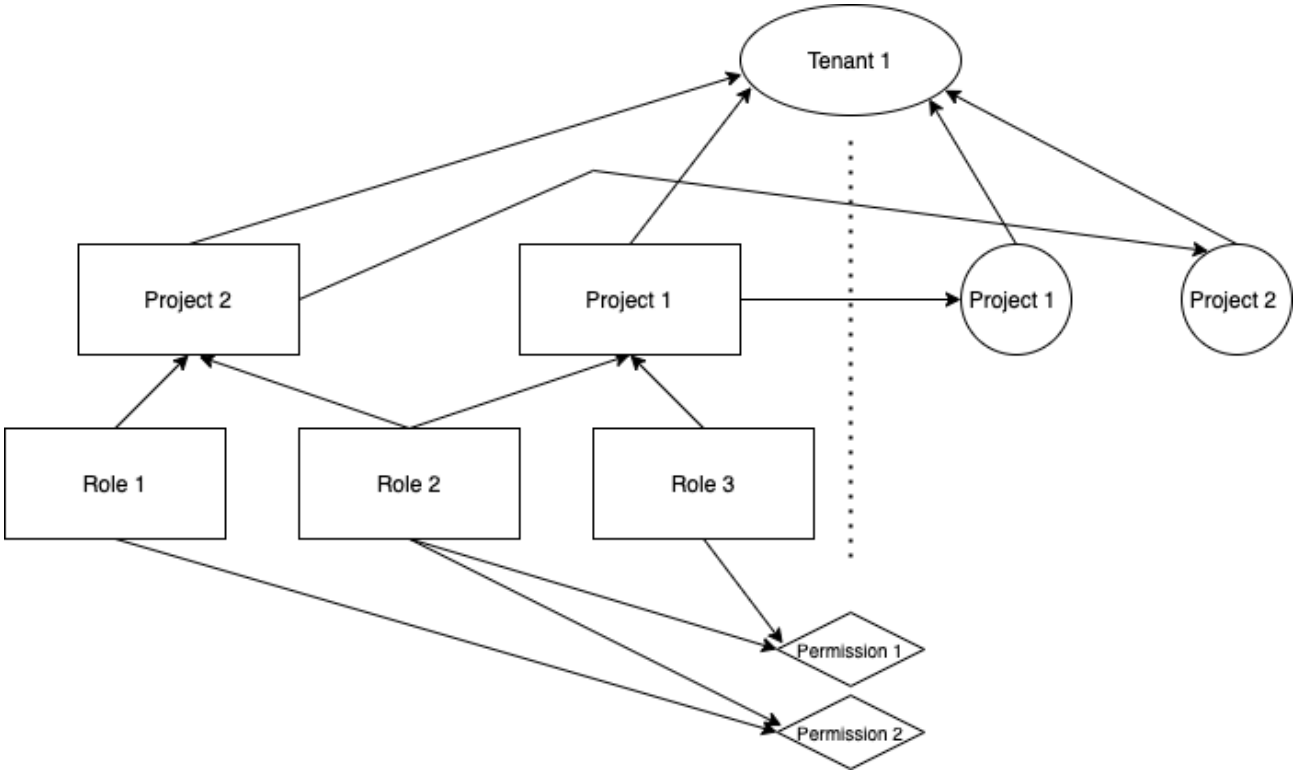


2. Unit 可以指向 Unit，一个 Unit 可以指向多个 Unit。一方面，父 Unit 的权限往子 Unit 继承，即子孙 Unit 下的所有 Subject 都将获得相关权限关系；另一方面，在 DAG 中进行权限计算，请求主体通过子孙 Unit 获得的权限往父 Unit 汇聚，**所以当两个 Unit 之间没有实质的权限管理关系时，不应该建立父子关系。**

3. Unit 可以指向 Permission，一个 Unit 可以指向多个 Permission，表示该 Unit 被分配了这些 Permission，Permission 是 Ending Vertice。

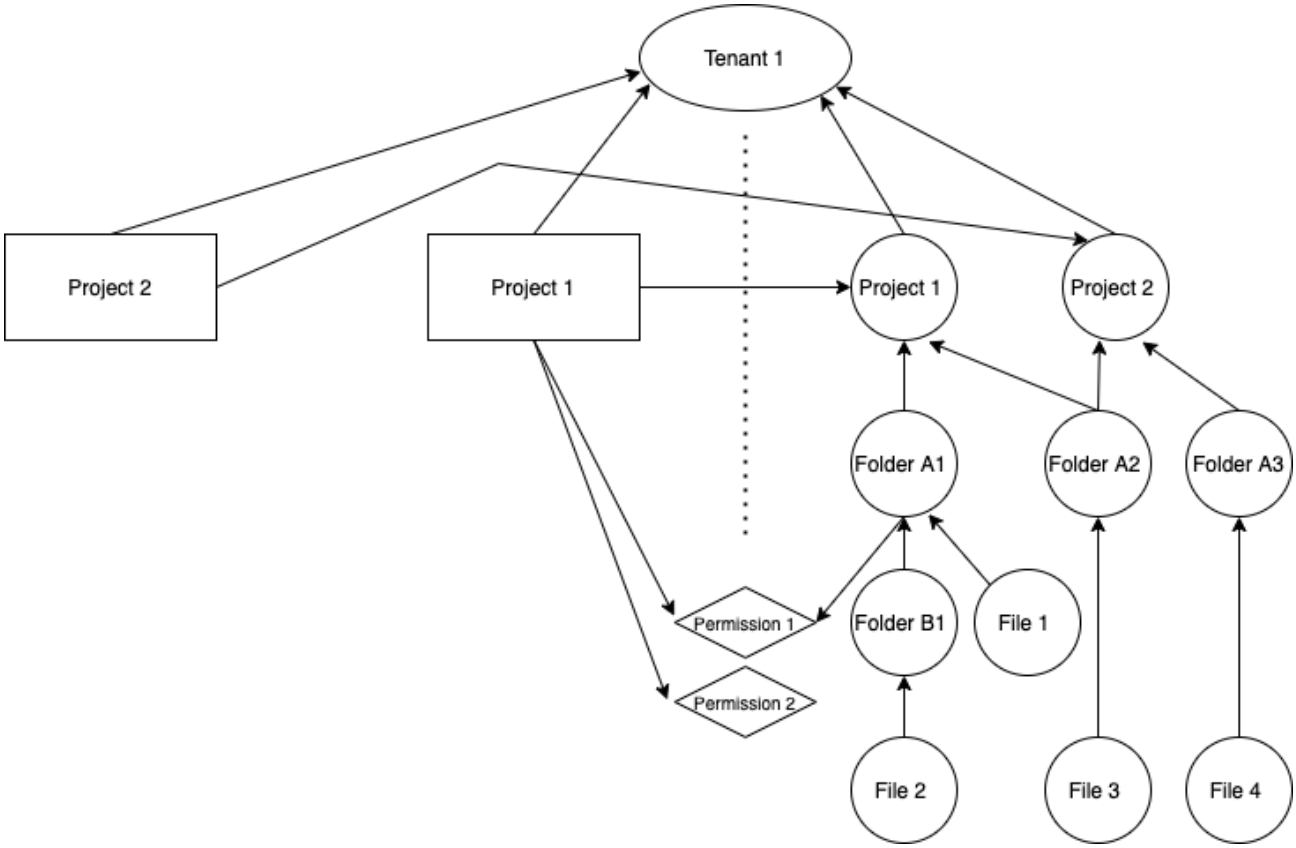
4. Unit 可以指向 Object，一个 Unit 可以指向多个 Object，表示请求主体在 Unit 节点聚集的权限可以作用于 Object 及其子孙。

5. Unit 可以指向 Scope，一个 Unit 可以指向多个 Scope，表示该 Unit 的权限关系从属于 Scope，Scope 是 Ending Vertice。



6. Object 可以指向 Object，一个 Object 可以指向多个 Object，表示了 Object 之间的父子关系或归属关系，父子关系或归属关系通常也象征权限透传关系，默认场景下，能访问父对象，就能访问子

对象，除非有权限透传的中断。
7. Object 可以指向 Permission，一个 Object 可以指向多个 Permission，表示该 Object 仅允许将这些 Permission 往子级透传，这里是基于白名单的机制来描述树形资源结构下权限透传的中断，当 Object 没有指向任何 Permission 时，表示 Subject 到达该 Object 时的所有 Permission 都往子级透传；当 Object 指向了 1 到 n 个 Permission 时，表示 Subject 到达该 Object 时的所有 Permission 中只有这 n 个能往子透传。
8. Object 可以指向 Scope，一个 Object 可以指向多个 Scope，表示该 Object 归属于 Scope，并在此处终止。



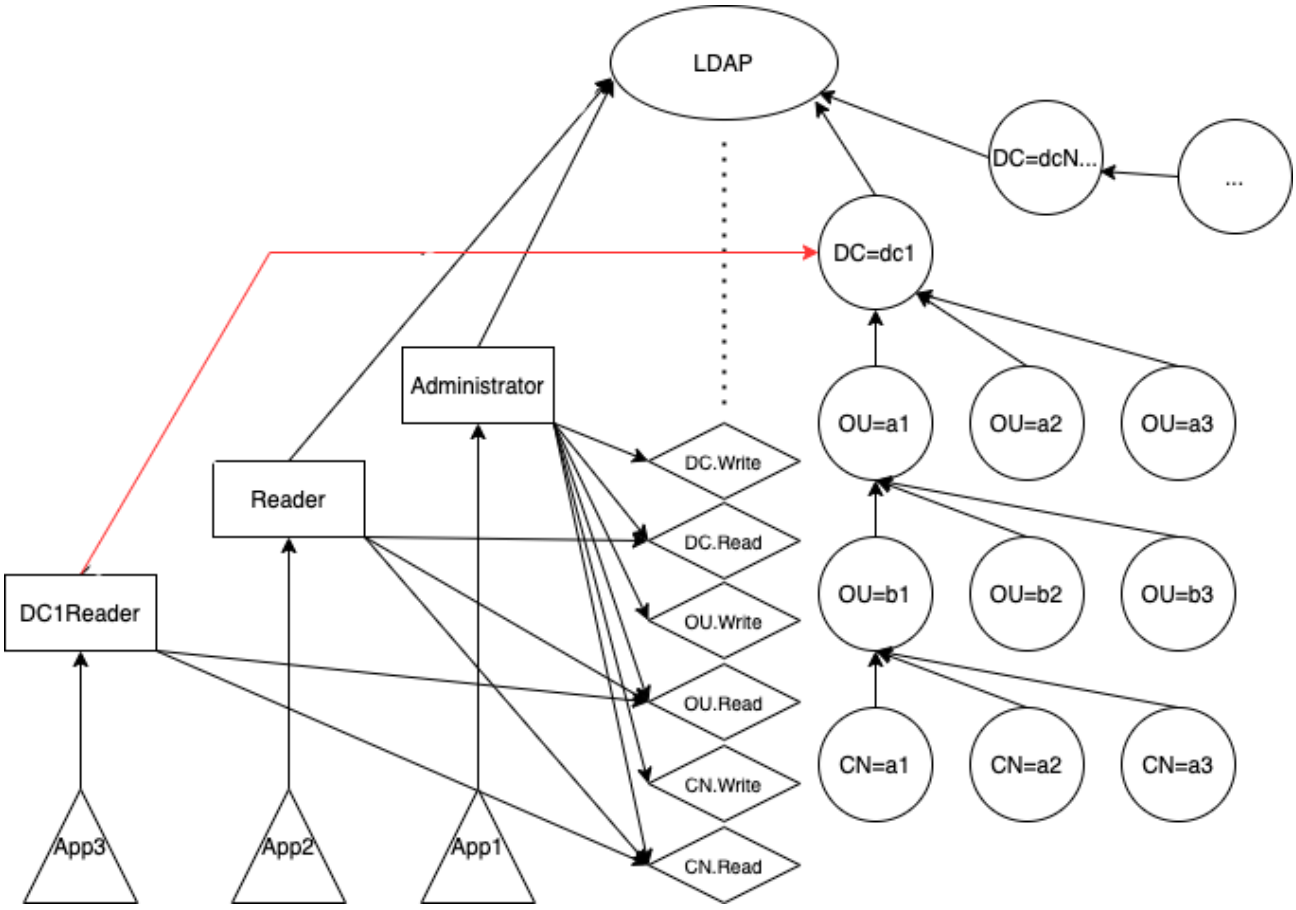
基于上面的关系约束，我们可以看到在 DAG 中：
1. Subject 只可能是 Starting Vertice，它总会指向 Unit；
2. Permission 和 Scope 只可能是 Ending Vertice；
3. Unit 既不可能是 Starting Vertice，也不可能是 Ending Vertice，它可能终止在 Scope 或 Unit；
4. Object 可能是 Starting Vertice，也可能是 Ending Vertice，当 Graph 中没有 Scope 时，Object 就可能是 Ending Vertice；
5. 当 Subject 与 Object 通过某些关系建立了连接时，则认为有了最基础的读权限，即 Subject 可以感知到 Object 的存在，至于 Subject 能不能读到 Object 的更多字段信息，则由业务自行定义。比如 Subject 通过连接关系能搜索到或者列出一组 Object，但进入或打开某个 Object 可能需要具体的读权限。

如果数据中存在违反上述断言的数据，表明权限关系还没有构建完整（正在构建中）。
但是在取得一份 DAG 数据后进行权限计算的过程中，我们可能 Reverse 该 DAG 方便计算。

权限场景推演示例

服务权限场景

我们以某公司的 LDAP 通讯录服务为例，公司有些应用需要调用通讯录数据，这些应用和通讯录数据都归属公司，一个简化的 GBAC 图如下：

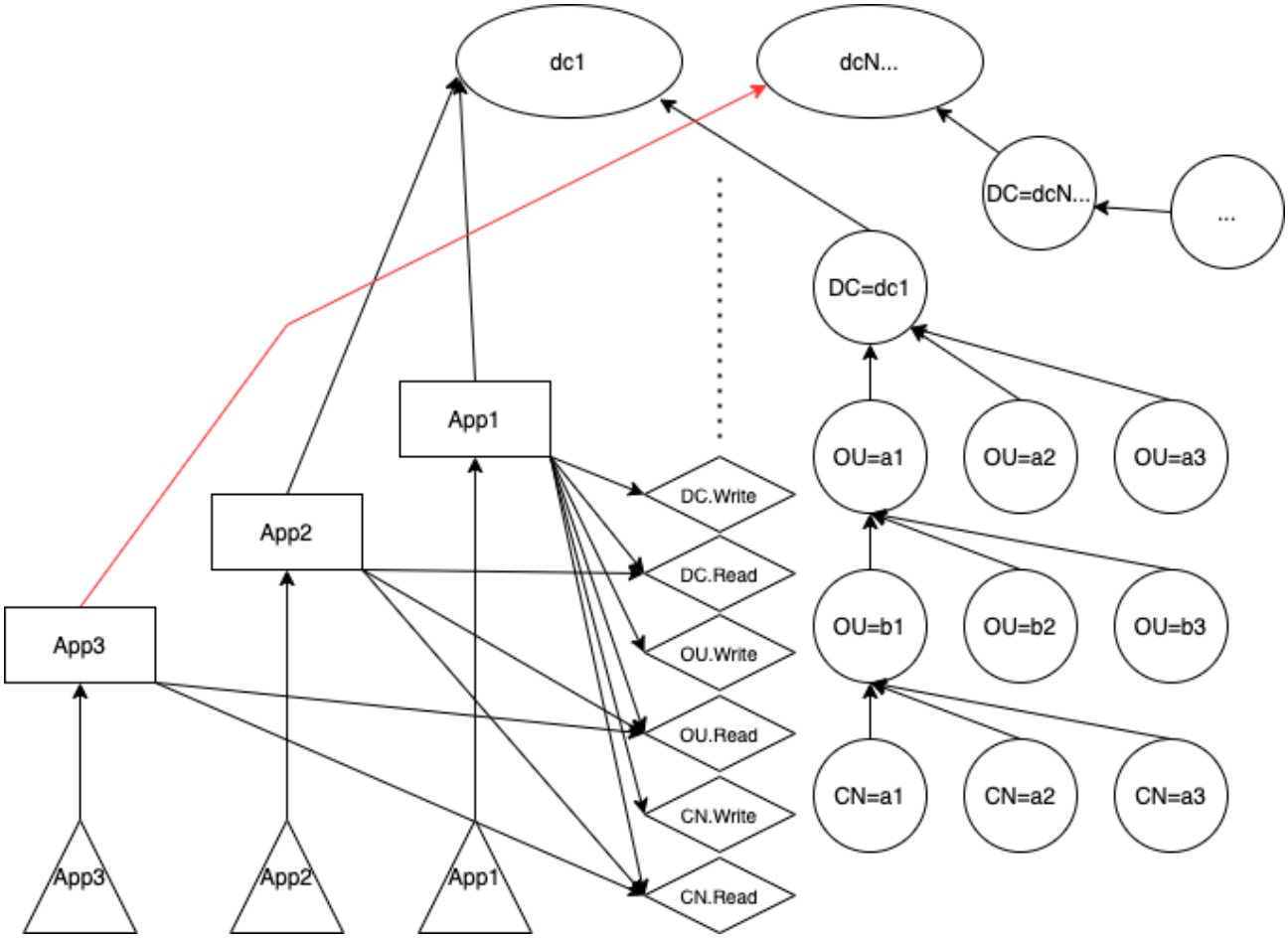


从图可知：
App1 被授予 Administrator 角色，通过 Scope:LDAP 对整个通讯录数据拥有所有读写权限。
App2 被授予 Reader 角色，通过 Scope:LDAP 对整个通讯录数据拥有所有读权限。
App3 被授予 DC1Reader 角色，只对 DC=dc1 及其所有子孙资源拥有读权限。

应用权限场景

我们还是以某公司的 LDAP 通讯录服务为例，这时候引入了多租户的概念，DC 作为租户。这时候

App 不但要申请相关权限，还需要获得租户授权才能访问对应数据。



从图可知：

App1 申请了通讯录服务的所有权限，并被 dc1 租户授权，所以它能对 dc1 租户下的所有资源进行读写操作。

App2 申请了通讯录服务的所有读权限，并被 dc1 租户授权，所以它能对 dc1 租户下的所有资源进行读操作。

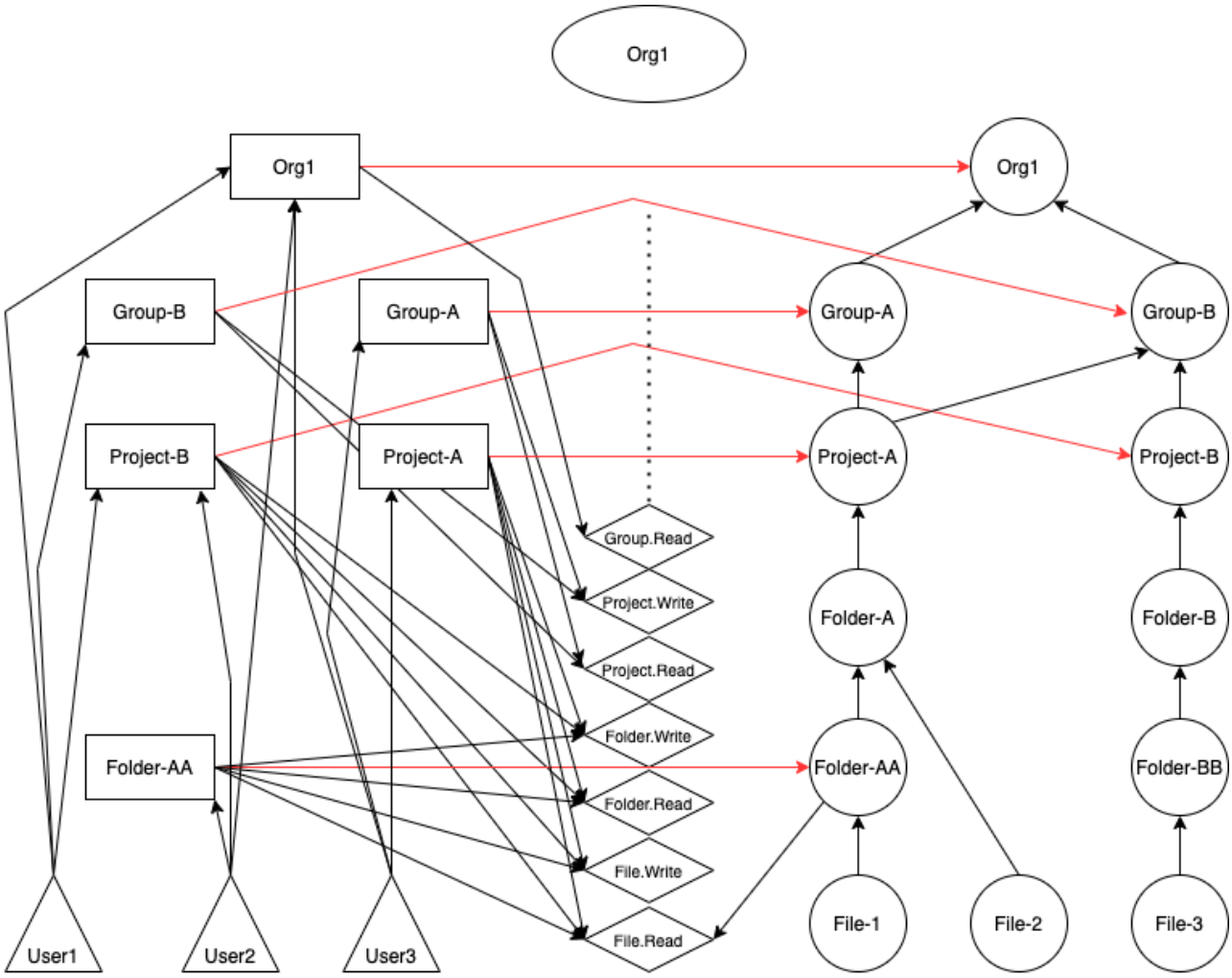
App3 只申请了通讯录服务的 OU 和 CN 类型资源的读权限，并被某个 dcN 租户授权，所以它能对那个 dcN 租户下的 OU 和 CN 资源进行读操作。如果移除红色的有向边，那么 App3 处于只申请了开发权限，还没有被任何租户授权的状态，它不能操作任何数据。

用户权限场景

我们构建了一个复杂但又简化了的用户权限场景，它的特征如下：

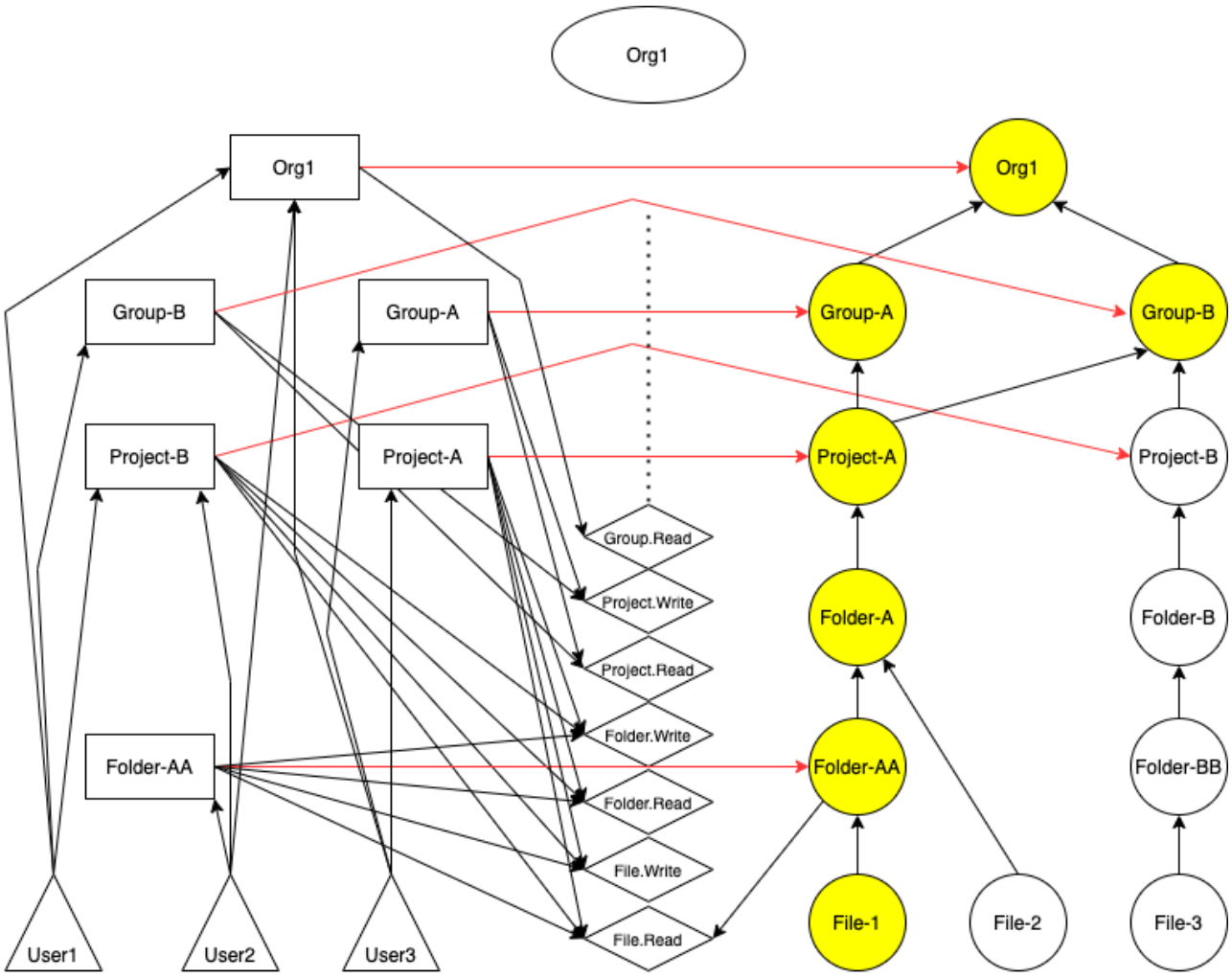
1. 本系统包含了 Org 组织、Group 群组、Project 项目容器、Folder 文件容器、File 文件及 User 用户（省略了角色等）；
2. 允许项目多归属，如图中 Project-A 同时归属于 Group-A 和 Group-B（我们也可以定义文件夹多归属，对于 DAG 而言，计算的复杂度是一样的）；

3. 我们在文件夹 Folder-AA 上设置了权限透传中断，即只有 File.Read 权限能往子级对象透传。但 Folder-A、Folder-B、Folder-BB 并没有任何设置，它们默认允许所有权限往子级透传；
4. 在这个场景下，Scope 不是必须的，所以图中没有画出 Scope:Org1 相关有向边。

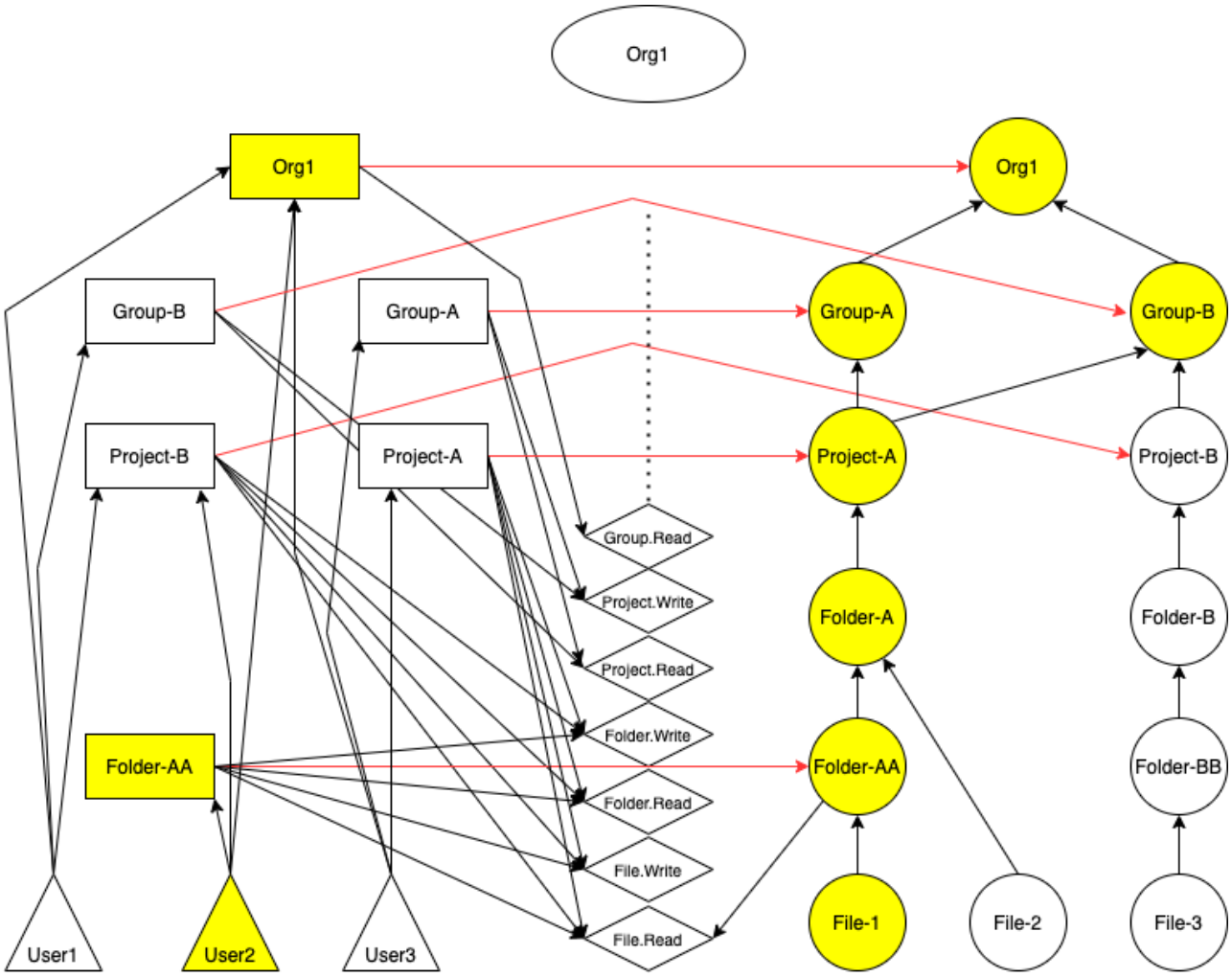


从图可知：
Unit:Org1 有 User1、User2、User3 三个成员，他们都能读 Group-A、Group-B。
Unit:Group-A 有 User3 一个成员，他能读写 Project-A。
Unit:Group-B 有 User1 一个成员，他能读写 Project-A 和 Project-B。
Unit:Project-A 有 User3 一个成员，他能读写 Project-A 下的 Folders 和 Files，除了 File-1，因为 Folder-AA 定义了权限透传中断，只有读文件的权限能往下透传。
Unit:Project-B 有 User1、User2 两个成员，他们能读写 Project-B 下的 Folders 和 Files。
Unit:Folder-AA 有 User2 一个成员，他能读写 Folder-AA 的所有文件和文件夹。

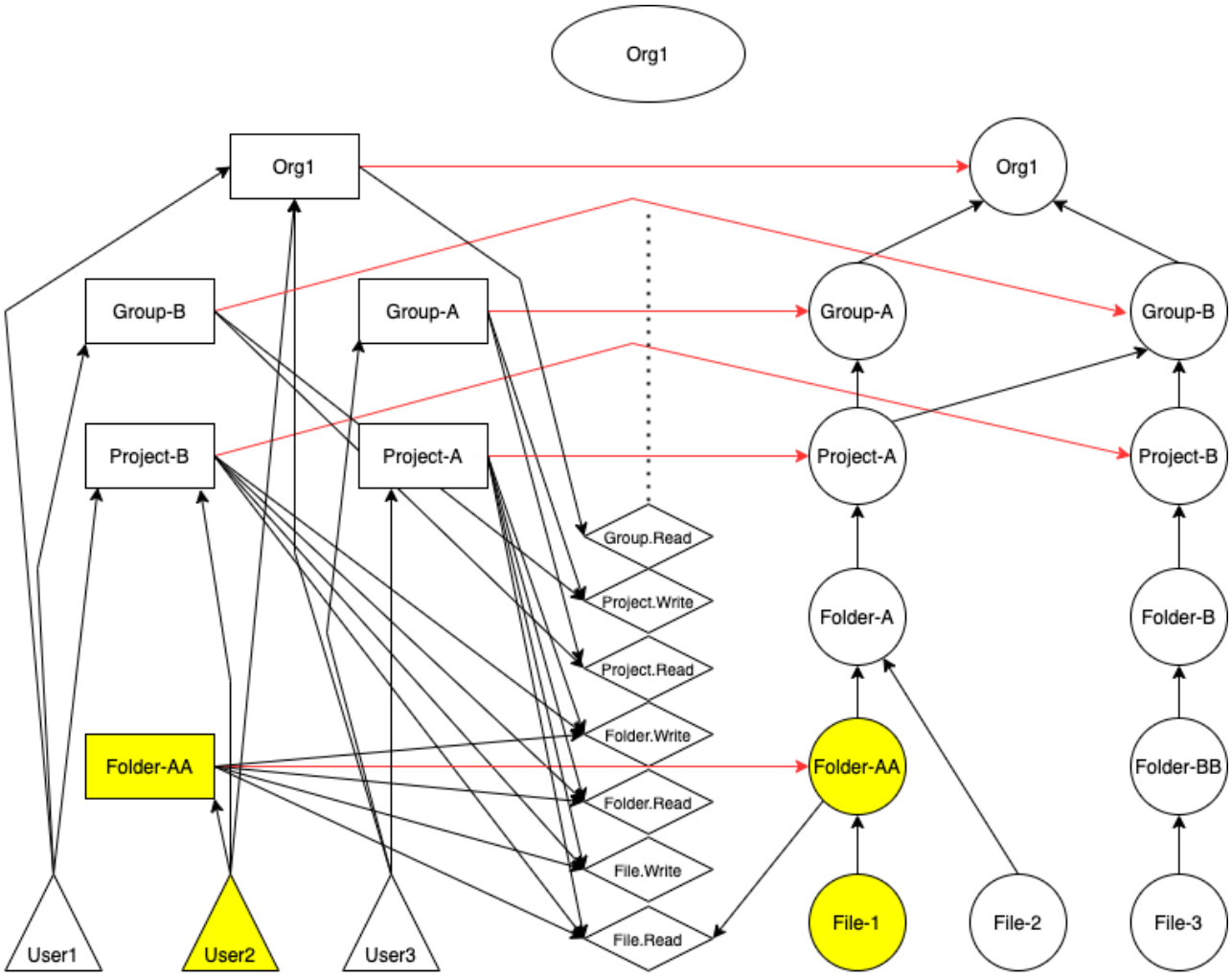
一个请求主体 Sub1 对一个资源 Obj1 有没有操作权限 P1，其计算步骤一般为：
1. 在数据库中读取该 Obj1 的 transitive closure DAG，其 Starting Vertice 为 Obj1，Ending Vertices 为绑定了 Unit 的 Object 节点，记为 DAG1（见示例图黄色节点）；



2. 在数据库中读取该 Sub1 的 transitive closure DAG，其 Starting Vertice 为 Sub1，Ending Vertices 为 DAG1 中绑定到 Object 的 Unit 节点，记为 DAG2（见示例图左侧黄色节点）；



3. DAG1 和 DAG2 中的 Permission 都转换为对应节点的 Attribute 属性；
4. DAG2 从 Sub1 出发，通过 Iterate 迭代计算所有路径，从各节点 Attribute 属性中聚集权限项集合 PS，针对每个 Ending Vertices 聚集各自的 PS，Sub1 可能通过多种 Unit 关系获得目标资源操作权限；
5. 如果所有 Ending Vertices 的 PS 中不存在 P1 权限，可以终止计算，Sub1 无权限；
6. 如果任意 Ending Vertices 的 PS 中存在 P1 权限，且资源没有定义权限中断，可以终止计算，Sub1 有权限；



7. 对 DAG1 进行 Reverse 反转，进一步计算权限透传中断。到达 Obj1 的有些路径可能定义了权限中断，而有些路径可能允许权限透传；
8. 仅考虑 DAG2 中含有 P1 权限的 Ending Vertices（示例图中的 Unit:Folder-AA），对应的的 Object 节点此时变成了 Starting Vertice。用 PS 权限集从每一个 Starting Vertice 出发，通过 Iterate 迭代计算所有路径，如果某个路径中定义了权限中断带没有 P1，则从 PS 移出 P1。当某一个迭代计算得到的最终权限集包含 P1 时，则 Sub1 有权限。

相关 DAG 计算库可参考 https://github.com/open-trust/dag-go

核心 API 设计

说明

1. 所有写操作都应该幂等，部分 API 可以通过 Header "**Prefer: respond-conflict**" 来声明对于写入资源冲突时响应 409 错误
2. 建议业务方创建 Permission("**Default**") 这一特殊类型的权限，可用于 Object 的权限中断场景或

代表其它没有资源实体的默认操作权限
3. 本 API 列表尤其是 Search 相关的 API，并不要求 GBAC 系统全部实现，按照业务需求实现即可
4. List 类的 API 都支持基于游标的分页

类型

```
type Target {
  type: String
  id: String
}

type Permission `Resource.Operation.Constraint`
```

访问控制查询 API

```
// 检查请求主体到指定管理单元有没有指定权限，如果未指定管理单元，则会检查请求主体能触达的所有管理单元
CheckUnit(subject: String!, unit: Target = null, permission: Permission!)

// 检查请求主体到指定范围约束有没有指定权限
CheckScope(subject: String!, scope: Target!, permission: Permission!)

// 检查请求主体通过 Scope 或 Unit-Object 的连接关系到指定资源对象有没有指定权限，如果 byUnitObject 为 true，则要求必须有 Unit-Object 的连接关系
CheckObject(subject: String!, object: Target!, permission: Permission!, byUnitObject: Boolean = false)

// 列出请求主体到指定管理单元的符合 resource 的权限，如果未指定管理单元，则会查询请求主体能触达的所有管理单元，如果 resources 为空，则会列出所有触达的有效权限
ListPermissionsByUnit(subject: String!, unit: Target = null, resources: [String])

// 列出请求主体到指定范围约束的符合 resource 的权限，如果 resources 为空，则会列出所有触达的有效权限
ListPermissionsByScope(subject: String!, scope: Target!, resources: [String])

// 列出请求主体到指定资源对象的符合 resource 的权限，如果 resources 为空，则会列出所有触达的有效权限
ListPermissionsByObject(subject: String!, object: Target!, resources: [String]!, byUnitObject
```

```
Boolean = false)

// 列出请求主体在指定资源对象中能触达的所有指定类型的子孙资源对象
// depth 定义对 targetType 类型资源对象的递归查询深度，而不是指定 object 到 targetType 类型资源对象的深度，默认对 targetType 类型资源对象查到底
ListObject(subject: String!, object: Target!, permission: Permission!, targetType: String!, byUnitObject: Boolean = false, depth: Int = MaxInt)

// 根据关键词，在指定资源对象的子孙资源对象中，对请求主体能触达的所有指定类型的资源对象进行搜索
SearchObject(subject: String!, object: Target!, permission: Permission!, targetType: String!, term: String!, byUnitObject: Boolean = false)
```

Scope 范围约束 API

```
// 创建范围约束
Add(scope: Target!)

// 删除范围约束
Delete(scope: Target!)

// 删除范围约束及范围内的所有 Unit 和 Object
DeleteAll(scope: Target!)

// 更新范围约束的状态，-1 表示停用
UpdateStatus(scope: Target!, status: Int!)

// 列出该系统当前所有指定目标类型的范围约束
List(targetType: String!)

// 列出范围约束下指定目标类型的直属的管理单元
ListUnits(scope: Target!, targetType: String!)

// 列出范围约束下指定目标类型的直属的资源对象
ListObjects(scope: Target!, targetType: String!)
```

Subject 请求主体 API

// 批量添加请求主体
BatchAdd(subjects: [String]!)

// 更新请求主体，-1 表示停用
UpdateStatus(subject: String!, status: Int!)

Permission 权限 API

// 批量添加权限
BatchAdd(permissions: [Permission]!)

// 删除权限
Delete(permission: Permission!)

// 列出该系统当前指定资源类型的权限，当 resource 为空时列出所有权限
List(resource: String = "")

Unit 管理单元 API

// 批量添加管理单元，当检测到将形成环时会返回 400 错误
BatchAdd(units: [Target]!, parent: Target = null, scope: Target = null)

// 建立管理单元与父级管理单元的关系，当检测到将形成环时会返回 400 错误
AssignParent(unit: Target!, parent: Target!)

// 建立管理单元与范围约束的关系
AssignScope(unit: Target!, scope: Target!)

// 建立管理单元与资源对象的关系
AssignObject(unit: Target!, object: Target!)

// 清除管理单元与父级对象的关系
ClearParent(unit: Target!, parent: Target!)

// 清除管理单元与范围约束的关系
ClearScope(unit: Target!, scope: Target!)

// 清除管理单元与资源对象的关系

```
ClearObject(unit: Target!, object: Target!)

// 删除管理单元及其所有子孙管理单元和链接关系
Delete(unit: Target!)

// 更新管理单元的状态，-1 表示停用
UpdateStatus(unit: Target!, status: Int!)

// 管理单元批量添加请求主体，当请求主体不存在时会自动创建
AddSubjects(unit: Target!, subjects: [String]!)

// 管理单元批量移除请求主体
ClearSubjects(unit: Target!, subjects: [String]!)

// 给管理单元添加权限，权限必须预先存在
UpdatePermissions(unit: Target!, permissions: [Permission])

// 覆盖管理单元的权限，权限必须预先存在，当 permissions 为空时会清空权限
OverridePermissions(unit: Target!, permissions: [Permission])

// 移除管理单元的权限
ClearPermissions(unit: Target!, permissions: [Permission])

// 列出管理单元的指定目标类型的子级管理单元，不包含 status 为 -1 的节点
ListChildren(unit: Target!, targetType: String!)

// 列出管理单元的指定目标类型的所有子孙管理单元，不包含 status 为 -1 的管理单元
// depth 定义对 targetType 类型管理单元的递归查询深度，而不是指定 unit 到 targetType 类型管理单元的深度，默认对 targetType 类型管理单元查到底
ListDescendant(unit: Target!, targetType: String!, depth: Int = MaxInt)

// 列出管理单元的直属权限
ListPermissions(unit: Target!)

// 列出管理单元的直属请求主体，不包含 status 为 -1 的请求主体
ListSubjects(unit: Target!)

// 列出管理单元及子孙管理单元下所有的请求主体，不包含 status 为 -1 的请求主体
```

ListDescendantSubjects(unit: Target!)

// 根据 start 和 ends 找出一个 DAG，其中 start 可以为 Subject 或 Unit，ends 为 0 到多个 Unit，不包含 status 为 -1 的节点
GetDAG(start: Target!, ends: [Target]!)

Object 资源对象 API

// 批量添加资源对象，当检测到将形成环时会返回 400 错误
BatchAdd(objects: [Target]!, parent: Target = null, scope: Target = null)

// 添加资源对象，并同时添加对应的管理单元和建立连接关系，当检测到将形成环时会返回 400 错误
AddWithUnit(object: Target!, parent: Target = null, scope: Target = null)

// 建立资源对象与父级对象的关系，当检测到将会形成环时会返回 400 错误
AssignParent(object: Target!, parent: Target!)

// 建立资源对象与范围约束的关系
AssignScope(object: Target!, scope: Target!)

// 清除资源对象与父级对象的关系
ClearParent(object: Target!, parent: Target!)

// 清除资源对象与范围约束的关系
ClearScope(object: Target!, scope: Target!)

// 删除资源对象及其所有子孙资源对象和链接关系
Delete(object: Target!)

// 更新资源对象的搜索关键词
UpdateTerms(object: Target!, terms: [String]!)

// 给资源对象添加可透传的权限，权限必须预先存在
UpdatePermissions(object: Target!, permissions: [Permission])

// 覆盖资源对象可透传的权限，权限必须预先存在，当 permissions 为空时会清空权限
OverridePermissions(object: Target!, permissions: [Permission])

// 移除资源对象可透传的权限
ClearPermissions(object: Target!, permissions: [Permission])

// 列出资源对象的指定目标类型的子级资源对象
ListChildren(object: Target!, targetType: String!)

// 列出资源对象的所有指定目标类型的子孙资源对象
// depth 定义对 targetType 类型资源对象的递归查询深度，而不是指定 object 到 targetType 类型资源对象的深度，默认对 targetType 类型资源对象查到底
ListDescendant(object: Target!, targetType: String!, depth: Int = MaxInt)

// 列出资源对象可透传的权限
ListPermissions(object: Target!)

// 根据 start 和 ends 找出一个 DAG，其中 start 为 Object，ends 为 0 到多个 Object
GetDAG(start: Target!, ends: [Target]!)

// 根据关键词在资源对象的所有指定类型的子孙资源对象中进行搜索，term 为空不匹配任何资源对象
Search(object: Target!, targetType: String!, term: String!)