**关于用户角色权限的一点想法（RBAC）**

2008-03-07 15:34

|  |
| --- |
| **前言**：  权限往往是一个极其复杂的问题，但也可简单表述为这样的逻辑表达式：判断“Who对What(Which)进行How的操作”的逻辑表达式是否为真。针对不同的应用，需要根据项目的实际情况和具体架构，在维护性、灵活性、完整性等N多个方案之间比较权衡，选择符合的方案。  **目标**：  直观，因为系统最终会由最终用户来维护，权限分配的直观和容易理解，显得比较重要，系统不辞劳苦的实现了组的继承，除了功能的必须，更主要的就是因为它足够直观。  简单，包括概念数量上的简单和意义上的简单还有功能上的简单。想用一个权限系统解决所有的权限问题是不现实的。设计中将常常变化的“定制”特点比较强的部分判断为业务逻辑，而将常常相同的“通用”特点比较强的部分判断为权限逻辑就是基于这样的思路。  扩展，采用可继承在扩展上的困难。的Group概念在支持权限以组方式定义的同时有效避免了重定义时  **现状**：  对于在企业环境中的访问控制方法，一般有三种：  1.自主型访问控制方法。目前在我国的大多数的信息系统中的访问控制模块中基本是借助于自主型访问控制方法中的访问控制列表(ACLs)。  2.强制型访问控制方法。用于多层次安全级别的军事应用。  3.基于角色的访问控制方法（RBAC）。是目前公认的解决大型企业的统一资源访问控制的有效方法。其显著的两大特征是：1.减小授权管理的复杂性，降低管理开销。2.灵活地支持企业的安全策略，并对企业的变化有很大的伸缩性。  **名词**：  粗粒度：表示类别级，即仅考虑对象的类别(the type of object)，不考虑对象的某个特  定实例。比如，用户管理中，创建、删除，对所有的用户都一视同仁，并不区分操作的具体对象实例。  细粒度：表示实例级，即需要考虑具体对象的实例(the instance of object)，当然，细  粒度是在考虑粗粒度的对象类别之后才再考虑特定实例。比如，合同管理中，列表、删除，需要区分该合同实例是否为当前用户所创建。  **原则**：  权限逻辑配合业务逻辑。即权限系统以为业务逻辑提供服务为目标。相当多细粒 度的权限问题因其极其独特而不具通用意义，它们也能被理解为是“业务逻辑”的一部分。比如，要求：“合同资源只能被它的创建者删除，与创建者同组的用户可 以修改，所有的用户能够浏览”。这既可以认为是一个细粒度的权限问题，也可以认为是一个业务逻辑问题。在这里它是业务逻辑问题，在整个权限系统的架构设计 之中不予过多考虑。当然，权限系统的架构也必须要能支持这样的控制判断。或者说，系统提供足够多但不是完全的控制能力。即，设计原则归结为：“系统只提供 粗粒度的权限，细粒度的权限被认为是业务逻辑的职责”。  需要再次强调的是，这里表述的权限系统仅是一个“不完全”的权限系统，即，它不提供所有关于权限的问题的解决方法。它提供一个基础，并解决那些具有“共性”的(或者说粗粒度的)部分。在这个基础之上，根据“业务逻辑”的独特权限需求，编码实现剩余部分(或者说细粒度的)部分，才算完整。回到权限的问题公式，通用的设计仅解决了Who+What+How 的问题，其他的权限问题留给业务逻辑解决。  **概念**：  Who：权限的拥用者或主体（Principal、User、Group、Role、Actor等等）  What：权限针对的对象或资源（Resource、Class）。  How：具体的权限（Privilege, 正向授权与负向授权）。  Role：是角色，拥有一定数量的权限。  Operator：操作。表明对What的How 操作。  **说明**：  User：与 Role 相关，用户仅仅是纯粹的用户，权限是被分离出去了的。User是不能与 Privilege 直接相关的，User 要拥有对某种资源的权限，必须通过Role去关联。解决 Who 的问题。  Resource：就是系统的资源，比如部门新闻，文档等各种可以被提供给用户访问的对象。资源可以反向包含自身，即树状结构，每一个资源节点可以与若干指定权限类别相关可定义是否将其权限应用于子节点。  Privilege：是Resource Related的权限。就是指，这个权限是绑定在特定的资源实例上的。比如说部门新闻的发布权限，叫做"部门新闻发布权限"。这就表明，该Privilege是一个发布权限，而且是针对部门新闻这种资源的一种发布权限。Privilege是由Creator在做开发时就确定的。权限，包括系统定义权限和用户自定义权限用户自定义权限之间可以指定排斥和包含关系(如：读取，修改，管理三个权限，管理 权限 包含 前两种权限)。Privilege 如"删除" 是一个抽象的名词，当它不与任何具体的 Object 或 Resource 绑定在一起时是没有任何意义的。拿新闻发布来说，发布是一种权限，但是只说发布它是毫无意义的。因为不知道发布可以操作的对象是什么。只有当发布与新闻结合在一起时，才会产生真正的 Privilege。这就是 Privilege Instance。权限系统根据需求的不同可以延伸生很多不同的版本。  Role：是粗粒度和细粒度(业务逻辑)的接口，一个基于粗粒度控制的权限框架软件，对外的接口应该是Role，具体业务实现可以直接继承或拓展丰富Role的内容，Role不是如同User或Group的具体实体，它是接口概念，抽象的通称。  Group：用户组，权限分配的单位与载体。权限不考虑分配给特定的用户。组可以包括组(以实现权限的继承)。组可以包含用户，组内用户继承组的权限。Group要实现继承。即在创建时必须要指定该Group的Parent是什么Group。在粗粒度控制上，可以认为，只要某用户直接或者间接的属于某个Group那么它就具备这个Group的所有操作许可。细粒度控制上，在业务逻辑的判断中，User仅应关注其直接属于的Group，用来判断是否“同组” 。Group是可继承的，对于一个分级的权限实现，某个Group通过“继承”就已经直接获得了其父Group所拥有的所有“权限集合”，对这个Group而言，需要与权限建立直接关联的，仅是它比起其父Group需要“扩展”的那部分权限。子组继承父组的所有权限，规则来得更简单，同时意味着管理更容易。为了更进一步实现权限的继承，最直接的就是在Group上引入“父子关系”。  User与Group是多对多的关系。即一个User可以属于多个Group之中，一个Group可以包括多个User。子Group与父Group是多对一的关系。Operator某种意义上类似于Resource + Privilege概念，但这里的Resource仅包括Resource Type不表示Resource Instance。Group 可以直接映射组织结构，Role 可以直接映射组织结构中的业务角色，比  较直观，而且也足够灵活。Role对系统的贡献实质上就是提供了一个比较粗颗粒的分配单位。  Group与Operator是多对多的关系。   **解释**：  Operator的定义包括了Resource Type和Method概念。即，What和How的概念。之所以将What和How绑定在一起作为一个Operator概念而不是分开建模再建立关联，这是因为很多的How对于某What才有意义。比如，发布操作对新闻对象才有意义，对用户对象则没有意义。  How本身的意义也有所不同，具体来说，对于每一个What可以定义N种操作。比如，对于合同这类对象，可以定义创建操作、提交操作、检查冲突操作等。可以认为，How概 念对应于每一个商业方法。其中，与具体用户身份相关的操作既可以定义在操作的业务逻辑之中，也可以定义在操作级别。比如，创建者的浏览视图与普通用户的浏 览视图要求内容不同。既可以在外部定义两个操作方法，也可以在一个操作方法的内部根据具体逻辑进行处理。具体应用哪一种方式应依据实际情况进行处理。  这样的架构，应能在易于理解和管理的情况下，满足绝大部分粗粒度权限控制的功能需要。但是除了粗粒度权限，系统中必然还会包括无数对具体Instance的细粒度权限。这些问题，被留给业务逻辑来解决，这样的考虑基于以下两点：  一方面，细粒度的权限判断必须要在资源上建模权限分配的支持信息才可能得以实现。比如，如果要求创建者 和普通用户看到不同的信息内容，那么，资源本身应该有其创建者的信息。另一方面，细粒度的权限常常具有相当大的业务逻辑相关性。对不同的业务逻辑，常常意 味着完全不同的权限判定原则和策略。相比之下，粗粒度的权限更具通用性，将其实现为一个架构，更有重用价值；而将细粒度的权限判断实现为一个架构级别的东 西就显得繁琐，而且不是那么的有必要，用定制的代码来实现就更简洁，更灵活。  所以细粒度控制应该在底层解决，Resource在实例化的时候，必需指定Owner和GroupPrivilege在对Resource进行操作时也必然会确定约束类型：究竟是OwnerOK还是GroupOK还是AllOK。Group应和Role严格分离User和Group是多对多的关系，Group只用于对用户分类，不包含任何Role的意义；Role只授予User，而不是Group。如果用户需要还没有的多种Privilege的组合，必须新增Role。Privilege必须能够访问Resource，同时带User参数，这样权限控制就完备了。  **思想**：  权限系统的核心由以下三部分构成：1.创造权限，2.分配权限，3.使用权限，然后，系统各部分的主要参与者对照如下：1.创造权限 -Creator创造，2.分配权限 - Administrator 分配，3.使用权限 - User：  1. Creator 创造 Privilege， Creator 在设计和实现系统时会划分，一个子系统或称为模块，应该有哪些权限。这里完成的是 Privilege与 Resource 的对象声明，并没有真正将 Privilege 与具体Resource 实例联系在一起，形成Operator。  2. Administrator 指定 Privilege 与 Resource Instance 的关联。在这一步， 权限真正与资源实例联系到了一起， 产生了Operator（Privilege Instance）。Administrator利用Operator这个基本元素，来创造他理想中的权限模型。如，创建角色，创建用户组，给用户组分配用户，将用户组与角色关联等等...这些操作都是由 Administrator 来完成的。  3. User 使用 Administrator 分配给的权限去使用各个子系统。Administrator 是用户，在他的心目中有一个比较适合他管理和维护的权限模型。于是，程序员只要回答一个问题，就是什么权限可以访问什么资源，也就是前面说的 Operator。程序员提供 Operator 就意味着给系统穿上了盔甲。Administrator 就可以按照他的意愿来建立他所希望的权限框架可以自行增加，删除，管理Resource和Privilege之间关系。可以自行设定用户User和角色Role的对应关系。(如果将 Creator看作是 Basic 的发明者，Administrator 就是 Basic 的使用者，他可以做一些脚本式的编程) Operator是这个系统中最关键的部分，它是一个纽带，一个系在Programmer，Administrator，User之间的纽带。  用一个功能模块来举例子。  一．建立角色功能并做分配：  1．如果现在要做一个员工管理的模块(即Resources)，这个模块有三个功能，分别是：增加，修改，删除。给这三个功能各自分配一个ID，这个ID叫做功能代号：  Emp\_addEmp，Emp\_deleteEmp，Emp\_updateEmp。  2．建立一个角色(Role)，把上面的功能代码加到这个角色拥有的权限中，并保存到数据库中。角色包括系统管理员，测试人员等。  3．建立一个员工的账号，并把一种或几种角色赋给这个员工。比如说这个员工既可以是公司管理人员，也可以是测试人员等。这样他登录到系统中将会只看到他拥有权限的那些模块。  二．把身份信息加到Session中。  登录时，先到数据库中查找是否存在这个员工，如果存在，再根据员工的sn查找员工的权限信息，把员工所有的权限信息都入到一个Hashmap中，比如就把上面的Emp\_addEmp等放到这个Hashmap中。然后把Hashmap保存在一个UserInfoBean中。最后把这个UserInfoBean放到Session中，这样在整个程序的运行过程中，系统随时都可以取得这个用户的身份信息。  三．根据用户的权限做出不同的显示。  可以对比当前员工的权限和给这个菜单分配的“功能ID”判断当前用户是否有打开这个菜单的权限。例如：如果保存员工权限的Hashmap中没有这三个ID的任何一个，那这个菜单就不会显示，如果员工的Hashmap中有任何一个ID，那这个菜单都会显示。  对于一个新闻系统(Resouce)，假设它有这样的功能(Privilege)：查看，发布，删除，修改；假设对于删除，有"新闻系统管理者只能删除一月前发布的，而超级管理员可删除所有的这样的限制，这属于业务逻辑(Business logic)，而不属于用户权限范围。也就是说权限负责有没有删除的Permission，至于能删除哪些内容应该根据UserRole or UserGroup来决定(当然给UserRole or UserGroup分配权限时就应该包含上面两条业务逻辑)。  一个用户可以拥有多种角色，但同一时刻用户只能用一种角色进入系统。角色的划分方法可以根据实际情况划 分，按部门或机构进行划分的，至于角色拥有多少权限，这就看系统管理员赋给他多少的权限了。用户—角色—权限的关键是角色。用户登录时是以用户和角色两种 属性进行登录的（因为一个用户可以拥有多种角色，但同一时刻只能扮演一种角色），根据角色得到用户的权限，登录后进行初始化。这其中的技巧是同一时刻某一 用户只能用一种角色进行登录。  针对不同的“角色”动态的建立不同的组，每个项目建立一个单独的Group，对于新的项目，建立新的 Group 即可。在权限判断部分，应在商业方法上予以控制。比如：不同用户的“操作能力”是不同的(粗粒度的控制应能满足要求)，不同用户的“可视区域”是不同的(体现在对被操作的对象的权限数据，是否允许当前用户访问，这需要对业务数据建模的时候考虑权限控制需要)。  **扩展性：**  有了用户/权限管理的基本框架，Who(User/Group)的概念是不会经常需要扩展的。变化的可能是系统中引入新的 What (新的Resource类型)或者新的How(新的操作方式)。那在三个基本概念中，仅在Permission上进行扩展是不够的。这样的设计中Permission实质上解决了How 的问题，即表示了“怎样”的操作。那么这个“怎样”是在哪一个层次上的定义呢？将Permission定 义在“商业方法”级别比较合适。比如，发布、购买、取消。每一个商业方法可以意味着用户进行的一个“动作”。定义在商业逻辑的层次上，一方面保证了数据访 问代码的“纯洁性”，另一方面在功能上也是“足够”的。也就是说，对更低层次，能自由的访问数据，对更高层次，也能比较精细的控制权限。  确定了Permission定义的合适层次，更进一步，能够发现Permission实际上还隐含了What的概念。也就是说，对于What的How操作才会是一个完整的Operator。 比如，“发布”操作，隐含了“信息”的“发布”概念，而对于“商品”而言发布操作是没有意义的。同样的，“购买”操作，隐含了“商品”的“购买”概念。这 里的绑定还体现在大量通用的同名的操作上，比如，需要区分“商品的删除”与“信息的删除”这两个同名为“删除”的不同操作。  提供权限系统的扩展能力是在Operator (Resource + Permission)的概念上进行扩展。Proxy 模式是一个非常合适的实现方式。实现大致如下：在业务逻辑层(EJB Session Facade [Stateful SessionBean]中)，取得该商业方法的Methodname，再根据Classname和Methodname 检索Operator 数据，然后依据这个Operator信息和Stateful中保存的User信息判断当前用户是否具备该方法的操作权限。  应用在 EJB 模式下，可以定义一个很明确的 Business层次，而一个Business 可能意味着不同的视图，当多个视图都对应于一个业务逻辑的时候，比如，Swing Client以及 Jsp Client 访问的是同一个 EJB 实现的 Business。在 Business 层上应用权限较能提供集中的控制能力。实际上，如果权限系统提供了查询能力，那么会发现，在视图层次已经可以不去理解权限，它只需要根据查询结果控制界面就可以了。  **灵活性**：  Group和Role，只是一种辅助实现的手段，不是必需的。如果系统的Role很多，逐个授权违背了“简单，方便”的目的，那就引入Group，将权限相同的Role组成一个Group进行集中授权。Role也一样，是某一类Operator的集合，是为了简化针对多个Operator的操作。  Role把具体的用户和组从权限中解放出来。一个用户可以承担不同的角色，从而实现授权的灵活性。当然，Group也可以实现类似的功能。但实际业务中，Group划分多以行政组织结构或业务功能划分；如果为了权限管理强行将一个用户加入不同的组，会导致管理的复杂性。  Domain的应用。为了授权更灵活，可以将Where或者Scope抽象出来，称之为Domain，真正的授权是在Domain的范围内进行，具体的Resource将分属于不同的Domain。 比如：一个新闻机构有国内与国外两大分支，两大分支内又都有不同的资源（体育类、生活类、时事政治类）。假如所有国内新闻的权限规则都是一样的，所有国外 新闻的权限规则也相同。则可以建立两个域，分别授权，然后只要将各类新闻与不同的域关联，受域上的权限控制，从而使之简化。  权限系统还应该考虑将功能性的授权与资源性的授权分开。很多系统都只有对系统中的数据（资源）的维护有权限控制，但没有对系统功能的权限控制。  权限系统最好是可以分层管理而不是集中管理。大多客户希望不同的部门能且仅能管理其部门内部的事务，而不是什么都需要一个集中的Administrator或Administrators组来管理。虽然你可以将不同部门的人都加入Administrators组，但他们的权限过大，可以管理整个系统资源而不是该部门资源。  正向授权与负向授权：正向授权在开始时假定主体没有任何权限，然后根据需要授予权限，适合于权限要求严格的系统。负向授权在开始时假定主体有所有权限，然后将某些特殊权限收回。  权限计算策略：系统中User，Group，Role都可以授权，权限可以有正负向之分，在计算用户的净权限时定义一套策略。  系统中应该有一个集中管理权限的AccessService，负责权限的维护（业务管理员、安全管理模块）与使用（最终用户、各功能模块），该AccessService在实现时要同时考虑一般权限与特殊权限。虽然在具体实现上可以有很多，比如用Proxy模式，但应该使这些Proxy依赖于AccessService。各模块功能中调用AccessService来检查是否有相应的权限。所以说，权限管理不是安全管理模块自己一个人的事情，而是与系统各功能模块都有关系。每个功能模块的开发人员都应该熟悉安全管理模块，当然，也要从业务上熟悉本模块的安全规则。  **技术实现**：  1．表单式认证，这是常用的，但用户到达一个不被授权访问的资源时，Web容器就发  出一个html页面，要求输入用户名和密码。  2．一个基于Servlet Sign in/Sign out来集中处理所有的Request，缺点是必须由应用程序自己来处理。  3．用Filter防止用户访问一些未被授权的资源，Filter会截取所有Request/Response，  然后放置一个验证通过的标识在用户的Session中，然后Filter每次依靠这个标识来决定是否放行Response。  这个模式分为：  Gatekeeper ：采取Filter或统一Servlet的方式。  Authenticator： 在Web中使用JAAS自己来实现。  用户资格存储LDAP或数据库：  1.      Gatekeeper拦截检查每个到达受保护的资源。首先检查这个用户是否有已经创建  好的Login Session，如果没有，Gatekeeper 检查是否有一个全局的和Authenticator相关的session？  2.      如果没有全局的session，这个用户被导向到Authenticator的Sign-on 页面，  要求提供用户名和密码。  3.      Authenticator接受用户名和密码，通过用户的资格系统验证用户。  4.      如果验证成功，Authenticator将创建一个全局Login session，并且导向Gatekeeper  来为这个用户在他的web应用中创建一个Login Session。  5.    Authenticator和Gatekeepers联合分享Cookie，或者使用Tokens在Query字符里。 |