任何收存和保管本论文各种版本的单位和个人，未经本论文作者授权，不得对本论文进行复制、修改、发行、出租、改编等有碍作者著作权益之行为，否则，将可能承担法律责任。

硕士学位论文

基于角色访问控制的理论与应用研究

俞诗鹏

北京大学数学学院信息科学系

2003 年 5 月

摘　要

基于角色的访问控制模型 RBAC 是目前主流的访问控制模型，它比传统的自主访问控制和强制访问控制更优越，同时也提供了更高的灵活性和扩展性。目前的 RBAC 模型在理论上和应用中仍存在着诸多问题，本文针对如下的三个问题提出了自己的解决方案。

首先，访问控制模型中角色的语义可能并不单一，当访问控制应用系统规模逐渐扩大以后，角色的数目可能成百上千，很难实行高效的管理。针对这种实际情况，传统的 RBAC 模型没有相应的对策，处理起来十分庞杂。本文从角色定义的语义出发，在角色集中引入维数的概念，提出一种全新的多维 RBAC 模型框架。我们给出了多维 RBAC 模型的形式化定义，并且详细讨论了基于多维 RBAC 模型的分布式角色管理模型。同时我们提出了一种角色命名机制来充分利用多维 RBAC 模型的特性简化系统角色管理。多维 RBAC 模型在一个基于角色的访问控制系统 WebDaemon 系统中得以实施。

然后，我们针对访问控制系统中查询操作远多于修改操作这个特性，在 RBAC 模型中引入缓存的概念，将原先通过递归运算得到的各种对应关系记录下来，从而提高后续查找的效率。我们讨论了一种 RBAC 模型外缓存和四种不同的 RBAC 模型内缓存，并针对不同的缓存策略给出合理高效的缓存更新算法，使得缓存系统得到合理充分的利用。我们随后分析比较了各种缓存算法的优劣，并在数据库和 LDAP 上进行了试验，取得了不错的统计结果。

最后，我们针对访问控制模块化的特点，提出了 RBAC 中间件的概念，并且给出具体设计方案和实现。该中间件对下端的存取模型给出原子操作接口，对上端的应用程序给出应用接口，并设计了灵活简便的方式实行整个模型的管理，方便访问控制模型的二次开发。多维 RBAC 模型和 RBAC 模型内缓存在中间件中也可以很方便的加以实现。本文在数据库和 LDAP 上进行了中间件的存取试验，并在一个大型访问控制系统中进行了配置和二次开发，基本实现了中间件的功能。

关键词：基于角色的访问控制，访问控制，中间件，网络安全

Research on Theory and Application of Role-Based Access Control

by

Shipeng YU
School of Mathematical Sciences
Peking University

Supervised by Professor Zuoquan LIN

May, 2003

Abstract

Compared with traditional DAC and MAC models, Role-Based Access Control (RBAC) Model can provide better flexibility and scalability, and is nowdays the best and most popular access control model. However, current RBAC models still have many problems in theoretical and application level. The following three problems are focused on this thesis by providing solutions in detail.

First of all, one role may contain several concepts in RBAC model, and edges in the role hierarchies can show different conceptual inheritance relations. The security officer can only manage roles and cannot deal with concepts, which makes RBAC administration difficult and tedious. We introduce dimensionality into role set and present a novel Multi-Dimentional RBAC Model (MDRBAC). A formal definition of MDRBAC model is described and an administrative model based on MDRBAC model is also discussed. We also introduce a role-naming strategy in the implementation level to simplify role administration. MDRBAC model is successfully deployed in a large RBAC system called WebDaemon.

Second, we observe that in access control system the number of query operations are far more than that of modification operations (add/delete). Therefore, we introduce cache strategy into RBAC model to improve the overall efficiency of access control system. One out-model cache and four in-model cache strategies are discussed, and a refresh algorithm is given to each cache strategy. We compare in detail all the five cache strategies, and perform experiments on database and LDAP, both showing that cache is very effective to improve efficiency.

Last but not least, we introduce the concept of RBAC middleware, and explain in detail its designation and implementation. The middleware reserves various primitive operations with respect to different storage models, and provides high-level interfaces to the up-level applications. The MDRBAC model and the in-model cache can be easily added to the middleware. The middleware is experienced in both database and LDAP, and is used for configuration and further development in a large system.

Keywords: Role-Based Access Control, Access Control, Middleware, Network Security

图表目录

表格目录

目　录

第1章 概述

随着计算机技术、通信技术和互联网的飞速发展，计算机安全性已经越来越引起人们的重视。访问控制作为一种重要的安全技术，已经渗透到操作系统、数据库、网络的各个方面。本章首先讲述访问控制的定义、目标并介绍一些已有的访问控制模型，然后在此基础上提出本文所考虑的问题和解决方案。

1.1 访问控制

访问控制（access control）是实施允许被授权的主体对某些客体的访问，同时拒绝向非授权的主体提供服务的策略。这里主体（subject）可以是人，也可以是任何主动发出访问请求的智能体，包括程序、进程、服务等；客体（object）包括所有受访问控制保护的资源，在不同应用背景下可以有相当广泛的定义，比如在操作系统中可以是一段内存空间，在数据库里可以是一个表中的记录，在Web 上可以是一个页面。访问的方式取决于客体的类型，一般是对客体的一种操作，比如请求内存空间，修改表中记录，浏览页面等。

通过对主体的授权，计算机系统可以在一个合法的范围内被使用，从而保证了客体被正确合理的访问，同时也维护了被授权主体的利益。这是访问控制的目的，同时也是一个安全系统所必须具备的特性。

访问控制在操作系统、数据库和网络上都有十分重要的应用。现代操作系统如 Windows 系列，Solaris 系列，UNIX 系列都实现了不同程度的访问控制；许多大型数据库产品如 Oracle，Sybase，Informix 等都支持访问控制功能；Web 安全产品如 getAccess，TrustedWeb，Tivoli 都把访问控制模块作为其核心模块之一。ISO 7498-2[8]更把访问控制作为设计安全的信息系统的基础架构中必须包含的五种安全服务之一。

形式化的说，给定一个主体集 S、客体集 O 以及对应于该客体集的访问操作集 R，访问控制的授权操作实际上是通过确定如下的函数 f 完成的：

$$f : S \times O \to 2^R$$

当授权结束时，每个主体对应于每个客体都将有一组操作许可，这样访问控制实

际上可以通过检查请求的访问操作是否在这组操作许可里来实现。

访问控制的实现机制包括访问控制表（access control list）、访问能力表（capability list）以及授权关系表（authorization list），它们实际上都是函数 f 定义的访问控制矩阵的一种实现方式。

根据应用背景的不同，访问控制可以有不同的实现策略。二十世纪七十年代以来，如下三种访问控制策略被依次提出，在现有的访问控制产品中应用较广。它们分别是：自主访问控制，强制访问控制，基于角色的访问控制。

1.2 自主访问控制（DAC）

自主访问控制（Discretionary Access Control）是根据访问者和（或）它所属组的身份来控制对客体目标的授权访问[35, 51]。一个对客体具有自主性访问权限的主体能够把该客体信息共享给其他的主体。UNIX 安全模型类似于 DAC 模型，文件的所有者通过设置文件许可权决定谁有权以何种方式对其进行访问。

围绕 DAC 模型中的权限转交、所有权变更、权限级联吊销等问题，DAC 模型有很多种形式。由于主体访问者对访问的控制有一定权利，信息在移动过程中其访问权限关系会被改变，如用户 A 可以将其对客体目标 O 的访问权限传递给用户 B，从而使不具备对 O 访问权限的 B 也可以访问 O。

自主访问控制模型的优点是具有相当的灵活性。客体的创建者具有对该客体的所有访问许可并且可以授权其他主体，这比较符合人们对权限的基本认识。但是访问许可的转移很容易产生安全漏洞，使得客体的所有者最终都不能控制对该客体的所有访问许可。所以综合说来，自主访问控制的安全级别较低。

1.3 强制访问控制（MAC）

强制访问控制（Mandatory Access Control）又称为基于格的访问控制 LBAC（Lattice-Based Access Control），是基于主体和客体的安全标记来实现的一种访问控制策略[15, 20, 24, 45]。系统中的每个主体被分配到不同的具有安全调查权限的域中，而系统中的每一个客体属于一个安全级别，这样的等级形成一个具有支配关系的框架，比如可以分为绝密级、机密极、秘密级、无密级等。强制性访问控制策略是体现在客体安全级别的敏感性上，而不是主体的行为属性。主体可

以访问一个文件，但是因为文件具有它自己的安全级别，所以在具体的应用环境中，我们可以实现满足多种安全级别需要的 MAC 安全模型。

在 Bell-Lapadula 模型[15]中，仅当主体对某一客体具有支配权时，它对该客体享有读许可权；仅当客体被某一主体所支配时，它对该客体享有写许可权。这被称为下读/上写。Biba 模型[20]考虑的是完全相反的情况（上读/下写）。在 MAC 模型中，只有当两个实体处在同一级别上时，才可以进行双向通信或通过被信任中间体进行通信，这样就可以利用上读/下写来保证数据完整性，利用下读/上写来保证数据的保密性，并且通过这种梯度安全标签实现信息的单向流通。

强制访问控制的优点是管理集中，根据事先定义好的安全级别实现严格的权限管理，因此适宜于对安全性要求较高的应用环境。美国军方就一直使用这种访问控制模型。但这种强制访问控制太严格，实现工作量太大，管理不便，不适用于主体或客体经常更新的应用环境。

1.4 基于角色的访问控制（RBAC）

前面两种访问控制模型都存在的不足是将主体和客体直接绑定在一起，授权时需要对每对（主体，客体）指定访问许可。这样存在的问题是当主体和客体达到较高的数量级之后，授权工作将非常困难。二十世纪九十年代以来，随着对在线的多用户、多系统的研究不断深入，角色的概念逐渐形成，并逐步产生了以角色为中心的访问控制模型（Role-Based Access Control）[28, 29, 40, 49]。

在 RBAC 模型中，角色是实现访问控制策略的基本语义实体。系统管理员可以根据职能或机构的需求策略来创建角色、给角色分配权限和给用户分配角色等。基于角色访问控制的核心思想是将权限同角色关联起来，而用户的授权则通过赋予相应的角色来完成，用户所能访问的权限就由该用户所拥有的所有角色的权限集合的并集决定。角色之间可以有继承、限制等逻辑关系，并通过这些关系影响用户和权限的实际对应。在实际应用中，根据事业机构中不同工作的职能可以创建不同的角色，每个角色代表一个独立的访问权限实体。然后在建立了这些角色的基础上根据用户的职能分配相应的角色，这样用户的访问权限就通过被授予角色的权限来体现。在用户机构或权限发生变动时，可以很灵活的将该用户从一个角色移到另一个角色来实现权限的协调转换，降低了管理的复杂度，而且这

些操作对用户完全透明。另外在组织机构发生职能性改变时，应用系统只需要对角色进行重新授权或取消某些权限，就可以使系统重新适应需要。这些都使得基于角色访问控制策略的管理和访问方式具有无可比拟的灵活性和易操作性。

基于角色的访问控制模型是一个策略中立的模型，完全独立于其他安全手段。由于引入角色的概念，最小权限原则得以实现，系统高度灵活并且低冗余。根据不同的应用需要可以实现不同级别的 RBAC 模型，并且可以实现多管理员协同管理。这些特点都进一步推动了 RBAC 模型的理论和应用研究。目前基于 RBAC 的实际系统已经逐步出现，并以飞快的速度递增。二十一世纪初，一个统一的 RBAC 模型框架被提出并即将成为标准[29]，基于角色的访问控制模型正逐步被理论界和工业界所接受。

1.5 本文的工作

RBAC 模型虽然已逐渐成熟，但是仍有许多亟待研究解决的问题。在理论上，由于引入角色作为主体和客体的中介，角色的语义十分模糊，并没有在模型中给出解释；在应用上，如何将角色理论模型应用于实际系统中并实现高效的访问控制，一直都是很难也是业界很关心的问题。本文针对 RBAC 模型的如下三点不足提出了自己的解决方案，并在一个实际的访问控制系统 WebDaemon[2]中付诸实施，取得了较好的效果。

在一个实际系统中运用 RBAC 理论模型时，角色本身的管理最为复杂，也直接影响到主体与客体之间的授权关系。当应用系统规模逐渐扩大以后，角色可能成百上千，并且和具体的业务逻辑密切相关。此时如果不对角色模型做一定的改进，是很难管理好整个访问控制平台的。针对这种实际情况，传统的 RBAC 模型没有相应的对策，处理起来十分庞杂，效率十分低下。本文从角色定义的语义出发，在角色集中引入维数的概念，并提出一种全新的多维 RBAC 模型框架。这种模型成功的扩展了 RBAC 的基本模型，简化了角色配置和管理工作，并且利用角色命名机制极大的提高了权限检索的效率。该模型在一个实际的访问控制系统中实现并取得了预期的应用效果。本文第三章讲述多维 RBAC 模型框架及应用。

由于角色之间存在继承关系，而继承关系又会导致权限的传递，在实际系统

中检查一个角色是否具有某种权限的时候实际上是一个递归操作。当递归深度比较高的时候，无论采用何种存取手段都会得到很低的效率。为了充分利用访问控制系统中查询操作远多于修改操作这个特性，本文在 RBAC 模型中引入缓存的概念，将原先通过递归运算得到的各种对应关系记录下来，从而提高后续查找的效率。这种缓存机制不改变原有的 RBAC 模型，只作用在应用层面上，但可以全面提高访问控制的效率。针对不同的缓存策略，我们同时给出一套合理高效的缓存更新算法，使得缓存系统在不产生错误的基础上得到合理充分的利用。本文提出的缓存机制在数据库和 LDAP 上都作了试验，取得了不错的统计结果。本文第四章介绍 RBAC 模型的缓存机制及其应用。

RBAC 访问控制模型是一个比较深刻的理论概念。虽然模型有很多种应用模式，但是在角色配置和角色管理上都大同小异，可以封装为一个模块。而且在实际系统配置中要求所有的管理人员都掌握这个模型并不十分容易，业界迫切需要一个功能优越，配置简便，扩展性强的 RBAC 模型处理元件。本文综合 RBAC 模型的理论框架和应用背景，提出了访问控制中间件的概念，封装了一个具备所有特性但尽可能基本的 RBAC 模型。该中间件对下端的存取模型给出原子操作接口，对上端的应用程序给出应用接口，并设计了灵活简便的方式实现整个模型的管理，方便访问控制模型的二次开发。本文在数据库和 LDAP 上进行了存取试验，在一个大型访问控制系统中进行了配置和二次开发，基本实现了中间件的功能。本文第五章讲述 RBAC 中间件的设计与实现。

除以上三部分内容之外，本文第二章全面介绍了 RBAC 模型的基本组成以及相关的理论和应用研究，涵盖了该领域的主要成果，并为本文后面的工作做铺垫。第六章总结本文工作，并指出进一步的研究方向。

访问控制是一个很大的研究领域，涉及诸多应用背景。本文为了清晰简明的介绍对访问控制模型的改进，在后续的论述中采用如下的 Web 访问控制作为实例，但所有的工作对其他应用背景同样适用。

假定该访问控制系统中的主体是人或智能程序，客体是 Web 服务器端的静态和动态页面，主体对客体的访问局限于 HTTP 协议中定义的对页面的 get，post，head 等八种方法[9]。访问控制的目的就是限制主体对客体的访问，简单的说就是不同的人能看到不同的页面。这个应用背景简单但十分重要，便于我们展开后

续的论述。

本文所讨论的对 RBAC 模型的改进都是基于对实际应用系统的研究，所进行的改进也都在基于前面这个应用背景的一个访问控制系统 WebDaemon 中得到真正应用。WebDaemon 系统实现了基于角色的 Web 访问控制，目前在国内某大型企业运行良好[2]。

第2章 RBAC 模型回顾

在基于角色的访问控制模型 RBAC 出现之前，自主访问控制 DAC 和强制访问控制 MAC 已经提出了二十年并且在诸多应用领域取得了巨大的成功。访问控制已经成为计算机安全产品中必不可少的组成部分之一。但是在二十世纪八十年代末九十年代初，随着计算机网络的快速发展和应用系统规模的不断扩大，这两种传统的访问控制模型已经无法适应新的应用环境。MAC 模型太强，DAC 模型太弱，它们都无法提供一种策略中立的、具有强扩展性的访问控制框架。

RBAC 模型就是在这种背景下被提出来的。它实际上是一种强制的访问控制模型，即用户自己不能进行自主授权和权限转移，但是它没有如 MAC 中那样限制信息的流向，而是引入了一种抽象的中介元素——角色——来传递授权信息，从而提供了足够的灵活性和扩展性。

本章回顾 RBAC 模型的发展历程并介绍最著名的 RBAC 模型 RBAC96 和 ARBAC97，然后讲述 RBAC 模型在理论上和应用中已取得的一些成果，最后讨论 RBAC 模型的优点。

2.1 RBAC 模型的提出

1992 年，美国国家标准与技术研究所（NIST）的 David Ferraiolo 和 Rick Kuhn 在综合了大量的实际研究之后，率先提出基于角色的访问控制模型框架，并给出了 RBAC 模型的一种形式化定义[28]。该模型第一次引入了角色的概念并给出其基本语义，指出 RBAC 模型实现了最小权限原则（least privilege）和职责分离原则（separation of duty）。该模型中给出了一种集中式管理的 RBAC 管理方案。1995 年他们以一种更直观的方式对该模型进行了描述[27]。

Matunda Nyanchama 和 Sylvia Osborn 在 1994 年仔细研究了 RBAC 模型中角色继承关系和角色权限指派，形式化的给出了角色管理的一系列算法[40]。他们指出，他们提出的角色组织结构足够基本，能够模拟其他形式的权限模型比如树状层次结构（hierarchies）[54]和权限图（privilege graphs）[12]。

Ravi Sandhu 和他领导的位于 George Mason 大学的信息安全技术实验室

（LIST）于 1996 年提出了著名的 RBAC96 模型[49]，将传统的 RBAC 模型根据不同需要拆分成四种嵌套的模型并给出形式化定义，极大的提高了系统灵活性和可用性。1997 年他们更进一步，提出了一种分布式 RBAC 管理模型 ARBAC97，实现了在 RBAC 模型基础上的分布式管理[48]。这两个模型清晰的表征了 RBAC 概念并且蕴涵了他人的工作，成为 RBAC 的经典模型。绝大多数基于角色的访问控制研究都以这两个模型作为出发点。

2001 年，RBAC 领域的领军人物 David Ferraiolo，Ravi Sandhu 等人联合拟定了一个 RBAC 模型的美国国家标准草案，力图统一不同模型中的术语，并对所有 RBAC 的基本操作给出伪码定义[29]。该模型类似于 RBAC96 模型，只是对权限部分做了一定的细化，分成操作（operation）和对象（object）。今年内 NIST 要给出该标准的 1.1 版（http://csrc.nist.gov/rbac/），相信不久以后就即将成为美国国家标准。

2.2 RBAC 基本模型

本节首先给出 RBAC 模型中各种术语的基本定义，然后介绍目前比较成熟的 RBAC96 和 ARBAC97 模型，作为后续研究的出发点。

2.2.1 术语定义

RBAC 模型中的常用术语如下：

用户（User）：是一个访问计算机系统中的数据或者用数据表示的其它资源的主体。我们用 U 表示全体用户的集合。用户一般情况下指人，也可为 Agent 等智能程序。

权限（Permission）：是对计算机系统中的数据或者用数据表示的其它资源进行访问的许可。我们用 P 表示全体权限的集合。权限一般是一种抽象概念，表示对于某种客体资源的某种操作许可。因此有的模型中将权限细化为二元组（操作，对象）[28]，其中对象是访问控制系统中的真正客体，操作是作用在该对象上的一种访问方式。由于该二元组中操作一般是与具体的对象相关的，我们在今后的模型中认为权限是一个语义统一体。

角色（Role）：是指一个组织或任务中的工作或位置，代表了一种资格、权

利和责任。我们用 R 表示全体角色的集合。角色是一种语义综合体，可以是一种抽象概念，也可以对应于具体应用领域内的职位和权利。

管理员角色（Administrator Role）：是一种特定的角色，用来对角色的访问权限进行设置和管理。在集中式管理控制模型中，管理员角色由一个系统安全管理员来完成；而在分布式管理控制模型中，可以采用制定区域管理员来对系统进行分布式管理，每个管理员可以管理该区域内的角色权限的配置情况。当然区域管理员的创建和权限授予则统一由顶级的系统安全管理员完成。

用户指派（User Assignment）：是用户集 U 到角色集 R 的一种多对多的关系，即有 $UA \subseteq U \times R$，也称为角色授权（Role Authorization）。$(u,r) \in UA$ 表示用户 u 拥有角色 r，从语义上来说就表示 u 拥有 r 所具有的权限。

权限指派（Permission Assignment）：是权限集 P 到角色集 R 的一种多对多的关系，即有 $PA \subseteq P \times R$。$(p,r) \in PA$ 表示权限 p 被赋予角色 r，从语义上来说就表示拥有 r 的用户拥有 p。

角色激活（Role Activation）：是指用户从被授权的角色中选择一组角色的过程。用户访问的时候实际具有的角色只包含激活后的角色，未激活的角色在访问中不起作用。相对于静态的角色授权来说，角色激活是一种动态的过程，提供了相当的灵活性。

会话（Session）：对应于一个用户和一组激活的角色，表征用户进行角色激活的过程。一个用户可以进行几次会话，在每次会话中激活不同的角色，这样用户也将具有不同的访问权限。用户必须通过会话才能激活角色。

角色继承关系（Role Inheritance）：是角色集 R 中元素间的一种偏序关系 $\geq$，满足

1．自反性：$\forall r \in R$，$r \geq r$；

2．反对称性：$\forall r_1, r_2 \in R$，$r_1 \geq r_2 \cap r_2 \geq r_1 \Rightarrow r_1 = r_2$；

3．传递性：$\forall r_1, r_2, r_3 \in R$，$r_1 \geq r_2 \cap r_2 \geq r_3 \Rightarrow r_1 \geq r_3$。

从语义上来说，两个角色 $r_1 \geq r_2$ 是指前者比后者级别更高，具有更大的权利。形式化的说，$r_1 \geq r_2$ 蕴涵 r_2 对应的权限指派 r_1 也拥有，同时 r_1 对应的用户指派 r_2 也拥

有，即有

$$r_1 \geq r_2 \Rightarrow Permission(r_2) \subseteq Permission(r_1) \cap User(r_1) \subseteq User(r_2)$$

其中 *Permission(r)*表示 *r* 对应的权限集，*User(r)*表示 *r* 对应的角色集。角色继承关系允许存在各种形式，包括多重继承。在这个偏序的意义下，角色集中并不一定存在最大角色和最小角色。

角色层次图（Role Hierarchies）：是给定了角色继承关系之后整个角色集形成的一个层次图：如果 $r_1 \geq r_2$，那么在图上就存在 r_1 到 r_2 的一条有向边。根据不同的角色偏序定义，角色层次图可以是树，倒装树，格，甚至极为复杂的图。一般为了简化角色层次图，有向边的箭头被省略，继承关系默认为自上而下。图 2.1 就是一个角色层次图的例子[49]。

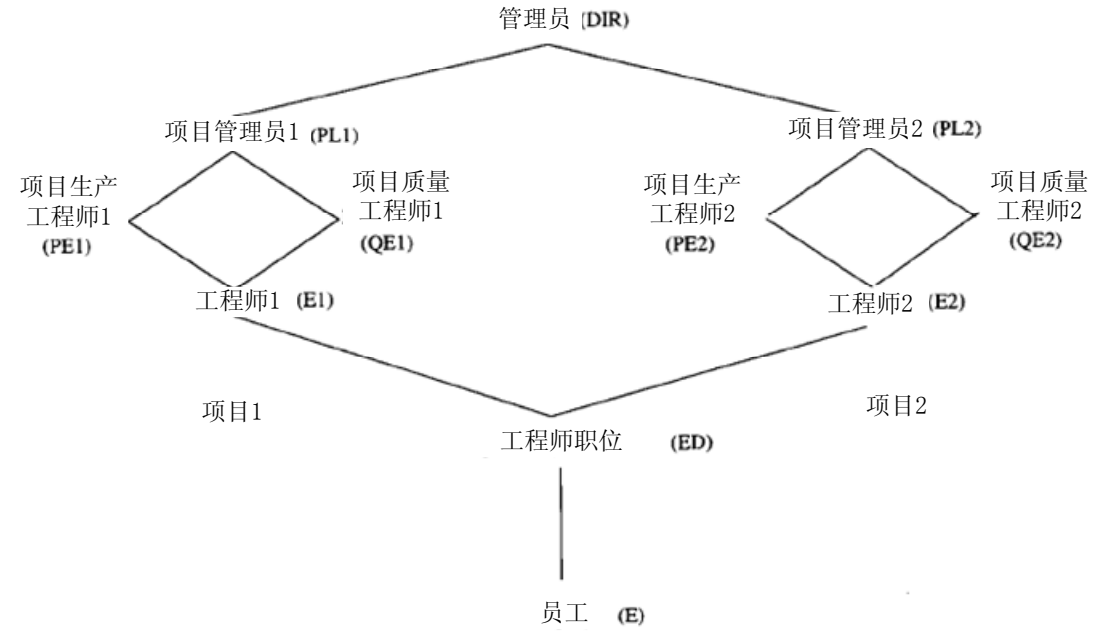


图 2.1. 角色层次图示例

限制（Constraints）：是在整个模型上的一系列约束条件，用来控制指派操作，指定职责分离（SD，Separation of Duty）以及避免冲突等。这是一个非常抽象的概念，RBAC 模型中并没有给出限制的类型和表述方式。任何独立于前面诸多术语的约束条件都是限制的一种形式。典型的限制包括指定角色互斥关系，角色基数限制等。根据职责分离的不同阶段，限制一般可分为静态职责分离和动态职责分离。

角色互斥关系（Mutually Exclusive Roles）：限制的一种，用于指定两个角

色具有不同的职责，不能让一个用户同时拥有。银行的出纳和会计便是角色互斥的简单例子。角色互斥关系的目的是为了在 RBAC 模型中引入业务逻辑的规则，避免冲突的发生。根据互斥的程度和影响的范围不同，角色互斥有很多种形式。文献[34]对此做过深入研究。

角色基数限制（Role Cardinality Constraints）：限制的一种，用于指定一个角色可被同时授权或激活的数目。比如总经理只能由一个用户担任，那么总经理角色的角色基数就为 1。根据下面定义的静态和动态的职责分离，角色基数限制有静态角色基数限制和动态角色基数限制两种。

静态职责分离（Static SD）：是指限制定义在用户指派（角色授权）阶段，与会话及角色激活无关。以角色互斥为例，如果定义两个角色为静态的角色互斥，那么任何一个用户都不能同时被指派到这两个角色。静态职责分离实现简单，语义清晰，便于管理，但是不够灵活，有些实际情况无法处理。

动态职责分离（Dynamic SD）：是指限制定义在角色激活阶段，作用域在会话内部。仍以角色互斥为例，如果定义两个角色为动态的角色互斥，那么一个用户可以同时被指派这两个角色，但是在任何一个会话中都不能同时激活它们。由此可见动态职责分离更灵活，基本上能处理各种实际情况，但实现略复杂。

2.2.2 基本模型 RBAC96

RBAC96 模型为 Ravi Sandhu 等人于 1996 年提出来的[49]。模型分四个层次，并具有如图 2.2 所示的包含关系。

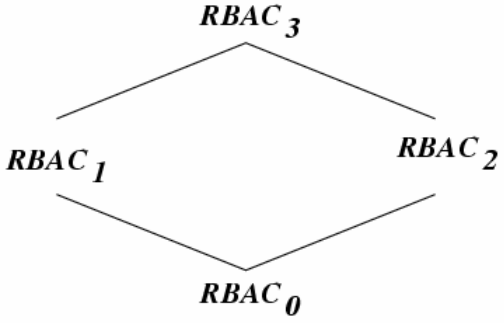


图 2.2. RBAC96 模型间的关系

2.2.2.1 $RBAC_0$

$RBAC_0$ 包含 RBAC 模型的核心部分（Core RBAC），是最基本的模型。$RBAC_0$ 可形式化的定义如下。

定义 2.1 $RBAC_0$ 模型包含如下元素：

1) 若干实体集 U（用户集），R（角色集），P（权限集），S（会话集）；
2) $UA \subseteq U \times R$，为多对多的用户角色指派关系；
3) $PA \subseteq P \times R$，为多对多的权限角色指派关系；
4) $user: S \to U$，映射每个会话到一个用户；
5) $roles: S \to 2^R$，映射每个会话到一组角色 $roles(s) \subseteq \{r \mid (user(s), r) \in UA\}$，并且会话 s 拥有权限 $\cup_{r \in roles(s)} \{p \mid (p, r) \in PA\}$。 ∎

从定义 2.1 中可以看出，$RBAC_0$ 只包含最基本的 RBAC 元素：用户，角色，权限，会话。所有的角色都是平级的，没有指定角色层次关系；所有的对象都没有附加约束，没有指定限制。

2.2.2.2 $RBAC_1$

$RBAC_1$ 模型包含 $RBAC_0$，然后定义了角色继承关系。$RBAC_1$ 的形式化定义如下。

定义 2.2 $RBAC_1$ 模型包含如下元素：

1) $U, R, P, S, UA, PA, user$ 与 $RBAC_0$ 一致；
2) $RH \subseteq R \times R$ 是 R 上的偏序关系，记为 $\geq$，称作角色继承；
3) $roles: S \to 2^R$ 修改为 $roles(s) \subseteq \{r \mid (\exists r' \geq r)[(user(s), r') \in UA]\}$，同时会话 s 拥有权限 $\cup_{r \in roles(s)} \{p \mid (\exists r'' \leq r)[(p, r'') \in PA]\}$。 ∎

这里 $RBAC_1$ 体现了 RBAC 模型中角色继承关系的语义。一个会话拥有的角色包含 UA 关系里面指定的角色以及它们的父角色，会话拥有的权限包含其拥有的所有角色在 PA 关系里面的权限以及它们的子角色对应的权限。

2.2.2.3 $RBAC_2$

$RBAC_2$ 模型同样包含 $RBAC_0$，但是只定义了限制。$RBAC_2$ 有一个并非形式化的定义如下。

定义 2.3 $RBAC_2$ 模型包含如下元素：

1) $RBAC_0$ 中的所有元素；

2) 一组限制条件，用于刻画 $RBAC_0$ 中各元素的组合合法性。 ▮

$RBAC_2$ 模型并没有指定限制条件的表现形式，只是从语义上给出了一个简短说明。这给了 RBAC 模型诸多扩展形式。Sandhu 的文章[49]中介绍了两种主要的限制：角色互斥和角色基数限制，这也是实际系统中最通常考虑的两种限制形式。

2.2.2.4 $RBAC_3$

$RBAC_3$ 包含 $RBAC_1$ 和 $RBAC_2$，自然也包含 $RBAC_0$。这是一个完整的 RBAC 模型，包含一切模型元素，也是最复杂的一种模型。角色层次和限制同时存在，限制也可以作用在角色层次之上。图 2.3 给出了 RBAC96 模型的基本元素关系以及不同层次 RBAC 模型。

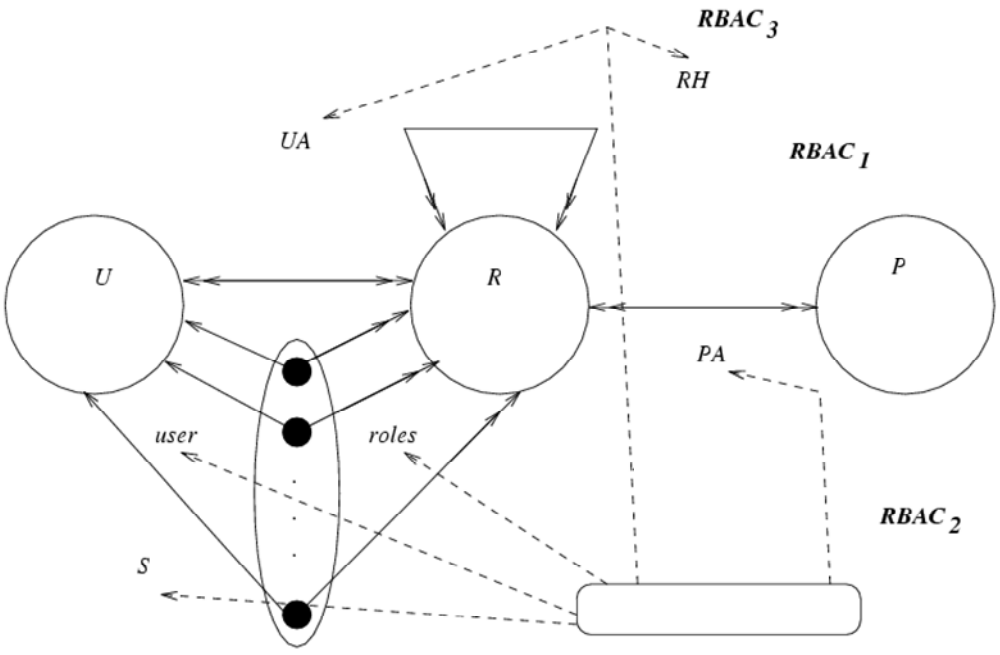


图 2.3. RBAC96 模型

2.2.3 角色管理模型 ARBAC97

Ravi Sandhu 等在 RBAC96 模型中就曾提出了角色的分布式管理的问题，但是没有详细谈到具体如何进行管理。之后他们很快就提出了著名的分布式角色管理模型 ARBAC97（Administrative RBAC）[48]，从理论上给出了 RBAC 模型中角色管理的办法。

ARBAC97 模型的基本思想是利用 RBAC 模型本身来进行 RBAC 模型的管理，包括用户角色管理，权限角色管理，角色层次关系管理，限制管理等几个部分。模型的管理员本身也具有角色，称作管理员角色，并且也有角色继承关系。管理员用户通过拥有管理员角色得到对角色继承关系的管理权。相对于非管理员的角色继承关系，管理员角色继承关系可以是一个单独的继承关系，并且该继承关系上的每个管理员角色将对应非管理员角色继承关系上的一部分管理区域，实现一种分工明确的分布式角色管理。图 2.4 给出了 ARBAC97 模型的基本框架。

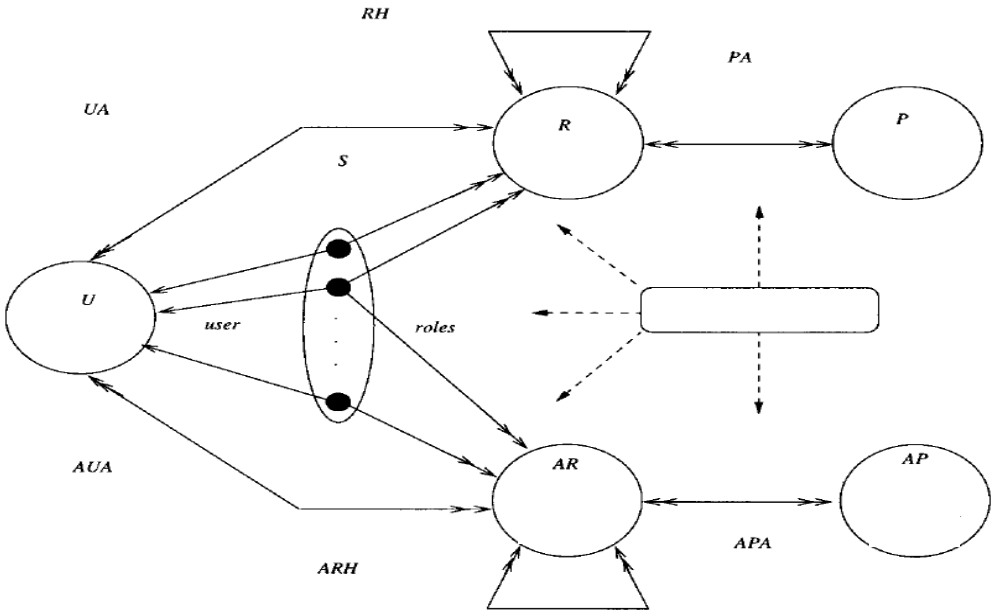


图 2.4. ARBAC97 模型

ARBAC97 模型分为三个部分：用户角色指派管理（URA97），权限角色指派管理（PRA97），以及角色继承关系管理（RRA97）。我们下面分别进行简要介绍。

2.2.3.1 URA97 模型

URA97 模型管理用户角色指派。从管理员的职责来看，URA97 模型分为两个部分：指派模型（Grant Model）和吊销模型（Revoke Model）。

在指派模型中，为了刻画不同层次的管理员能够管理的用户角色指派的范围，模型定义了一个 can_assign 关系，确定每个管理员对应于每个角色是否能够进行用户指派。考虑到管理员之间也存在一个层次关系，实际中的指派模型给每个管理员指定了一个管理范围，可以用一个区间来表示。高级的管理员角色的管理区间包含下级角色的管理区间，从而形成了一个有层次的、职责分明的管理层次。

对应于吊销模型，每个管理员也有一个吊销的角色区间，他可以在该区间中吊销任何角色的对应用户。根据管理员层次，我们同样有一个吊销的继承关系保证管理的不越级操作。由于一个角色对应的用户可以是通过角色继承关系得到的，因此在吊销模型中又可以分为强吊销（Strong Revoke）和弱吊销（Weak Revoke）。如果一个吊销操作是弱吊销，那么如果该用户是通过继承关系成为该角色的对应用户，吊销操作将不起作用；如果是强吊销，那么将强行剥夺该用户属于上层角色的权利。一般来说强吊销可能产生一些不可预知的后果，所以处理起来应该比较慎重。

文献[47]中还给出了一个 URA97 模型在 Oracle 上的实现，可供实际参考。

2.2.3.2 PRA97 模型

PRA97 模型管理权限角色指派。由于在 RBAC96 模型中权限和用户的地位是对称的，因此 PRA97 模型实际上是 URA97 模型的一个对偶模型。PRA97 模型中同样可以定义指派模型和吊销模型，也同样可以定义管理员的管理区间。在吊销模型中同样存在强吊销和弱吊销之分。文献[46]给出了一个 PRA97 模型在 Oracle 上的实现。

2.2.3.3 RRA97 模型

角色本身的管理是整个 RBAC 模型中最复杂的部分。由于角色的继承关系会影响到用户角色指派和权限角色指派，因此管理员可管理的角色区间会更加严格。类似于 URA97 模型，每个管理员针对每一种改变角色的操作都可以定义一个 can_modify 方法，刻画是否可以添加角色、删除角色以及改变角色间的继承关系。RRA97 模型中定义了多种角色区间的概念，并且给出了一些形式化的证明，保证了这些区间能够安全的实现角色模型的分布式管理。更多细节请参看文献[48, 50]。

2.3 RBAC 模型的理论研究

随着 RBAC 模型各种扩展的提出以及各种 RBAC 模型的相互比较，RBAC 模型在理论上取得了很多研究成果。这里我们简单介绍一些相关的工作。

RBAC 模型最初实际上是为了解决 MAC 和 DAC 模型的问题而提出来的。那么 RBAC 模型究竟能够在多大程度上解决 MAC 和 DAC 的问题呢？2000 年 Osborn 等人研究了这个问题，证明了 RBAC 是一种更一般的访问控制模型[42]。他们利用 RBAC 模型成功的模拟了 MAC 和 DAC，并且给出了如图 2.5 所示的包含关系。通过定义合适的用户、角色和权限，RBAC 模型可以衍生出更多种类的访问控制模型，从而可以真正取代已有的两种模型。

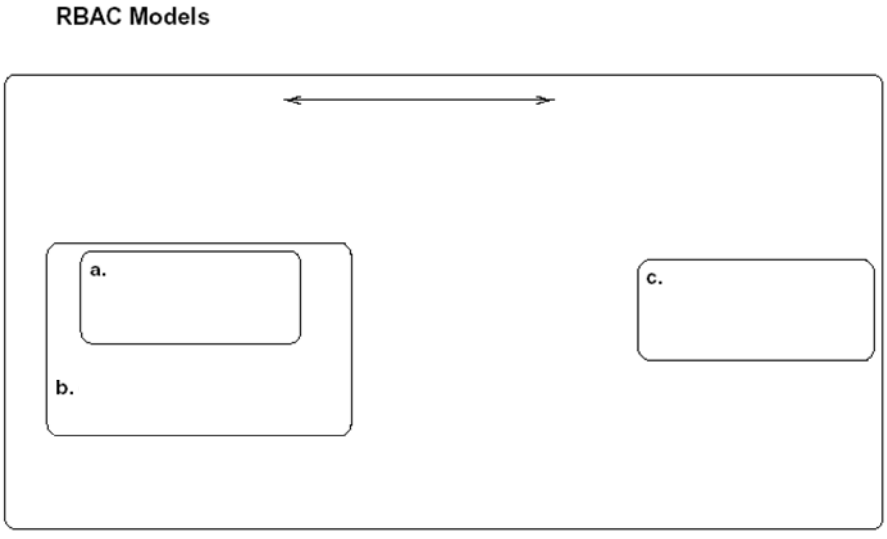


图 2.5. 几种权限管理模型的包含关系

角色管理是 RBAC 模型中最重要的部分。虽然 ARBAC97 模型给出了一种角色模型的分布式管理模式，但是许多定义都只在理论层面，无法付诸实施。有许多研究都是针对如何处理角色间的关系以及如何优化角色模型的管理，相关的文献包括[25, 32, 37, 38, 41]。

限制是 RBAC96 模型引入的一个角色间关系的概念。由于限制是一个十分抽象的概念，如何定义限制，如何管理限制都是很重要的问题。Kuhn 在 1997 年详细探讨了限制的分类，并给出了限制的诸多性质[34]。2000 年 Ahn 进一步给出了一种基于角色的限制语言 RCL 2000，用一种逻辑的方式刻画了限制的语义及其推理机制[11]。Ahn 进一步给出了 RCL 2000 的可靠性与完全性定理，使得 RBAC 模型可以在该框架下实现自动推理。

角色代理模型（Role-Based Delegation Model）是最近提出来的一个研究方向。它的基本思想是实现上级角色在不通过管理员的情况下将自己的权限代理给下级角色，有点类似于 DAC 模型中的授权模式。由于角色代理可以有十分复杂的类型，RBDM 的研究也进展的十分缓慢。Barka 首先提出了 $RBDM_0$ 模型，在 $RBAC_0$ 模型上引入角色代理[13, 14]。但是一旦角色间存在角色层次关系，角色代理就会变得异常复杂。目前还没有一个合适的模型来处理这种复杂的情况。其它有关角色代理的文章包括[39, 55, 56]。

角色模型的目的是为了确定给定的用户对给定的资源是否具有访问权限。由于 RBAC 模型提供了足够的灵活性，使得可以从代数的角度来研究角色间的关系以及角色对用户和权限的影响。Bonatti 等人在 2002 年给出了一种构建访问控制策略的代数，初步实现了这一想法[21]。他们进一步在 2003 年给出了一种基于访问控制模型的推理机制[17]。

针对不同类型的应用，RBAC 模型在理论上有很多种扩展。近年来有如下一些主要工作：时序 RBAC 模型（Temporal RBAC Model）[16]，分布式 RBAC 模型（Distributed RBAC Model）[30]，扩展层次的 RBAC 模型（Extended Hierarchy RBAC Model）[1]，基于联合的访问控制模型（Coalition-Based Access Control）[22]等等。这些工作充分利用了 RBAC 模型的灵活性来实现具体应用领域的访问控制管理。本文后面要介绍的多维 RBAC 模型也是 RBAC 基本模型的一种扩展。

2.4 RBAC 模型的应用研究

一个模型无论理论上提的多么好，最终的实用性才是检验一个模型好坏的标准。RBAC 模型从提出至今就一直有着各种应用。我们在这里简要叙述如下。

David Ferraiolo 在提出第一个 RBAC 模型之后，就实现了一个简单的原型系统[27]。Ravi Sandhu 也实现了自己的基于 RBAC96 和 ARBAC97 模型的原型系统[48, 49]。虽然他们发表的文章中给出了系统实现的框架和界面描述，但是系统仍然太简单，无法实用化，也无法处理各种复杂的角色关系问题。

由于在 RBAC 模型中有角色继承关系，很多研究者从其他包含继承关系的角度来研究 RBAC 模型的实现。这里有 OO 的方法[23]，Java 的方法[31]，CORBA 的方法[19]，Graph 的方法[33]等等多种角度。目前并没有见到具体的采用何种方法来实现的实际 RBAC 系统，但是不可否认的是一个真正的 RBAC 系统必须综合采用这几种技术才能实现准确、高效的角色管理。

对于在一个实际系统，尤其是在 Web 访问控制中如何实现角色模型，Park 有一篇很经典的文章刻画这个问题[43]。他提出了两种利用角色模型实现访问控制的机制 user_pull 和 server_pull，并且很好的分析了两种方法的优劣，如图 2.6 和图 2.7 所示。他同时在文章中给出了三种实现角色模型的方式：安全 cookies，智能认证，LDAP 目录服务。这也是目前实际系统中采用的主要实现手段。文章针对每种实现策略都给出了系统框架，但是没有涉及到角色模型本身的管理。其它介绍 RBAC 实现的文章包括 RBAC 在企业内联网的实现[26]，RBAC 在工作流上的实现[18]，RBAC 在卫生保健上的应用[36]等等。

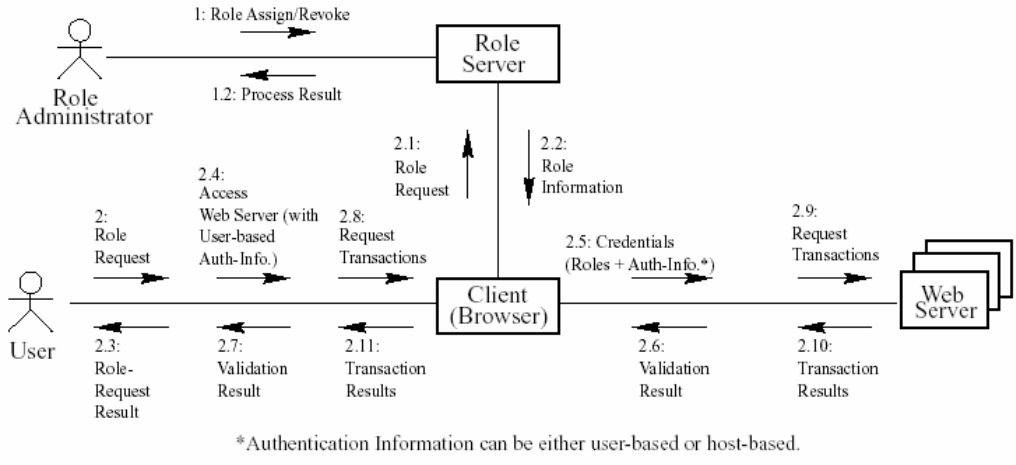


图 2.6. user-pull 的实现结构

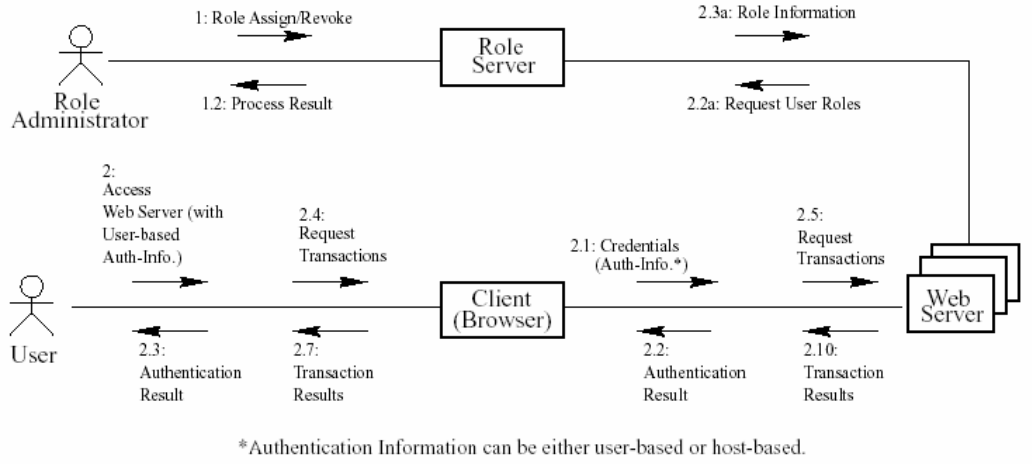


图 2.7. server-pull 的实现结构

目前已有相当的 RBAC 实用产品。在操作系统方面，Windows 系列产品[53]、Solaris[10]都已经蕴涵了角色的概念；数据库产品诸如 Oracle 8.0，Sybase 11.5，Informix 7.2 等都实现了不同程度的 RBAC 模型[44]；Web 安全产品中 GetAccess[5]，TrustedWeb[4]，Tivoli[6]等都实现了 RBAC 的一部分功能。但是目前的 RBAC 应用仍然不是很广，许多理论问题都没有对应的实际解决方案，因此 RBAC 的应用层面仍然需要进行大量研究。

2.5 RBAC 模型相关讨论

RBAC 模型作为一个较新的模型，具有如下的一些优点：

1. 通过角色配置用户和权限，增加了灵活性

由于增加了角色作为中介，用户和权限不再直接相关，权限配置时就非常灵活。而且角色一般会对应于实际系统中的一些具体的语义概念，管理员配置的时候也十分方便、直观。

2. 支持多管理员的分布式管理

一个大型访问控制系统不可能由一个管理员负责全面管理，因此 RBAC 模型提供的分布式管理模式就非常重要了。虽然配置分布式管理仍不是很容易，但是从模型的角度就支持这种管理模式不失为一大进步。

3. 支持由简到繁的层次模型，适合各种安全需要

RBAC96 模型提供了不同层次的模型可供实际选用，并且在每一层模型上都给出了形式化的定义。虽然 $RBAC_0$ 十分简单，但是实际中确实存在很多应用只

需要应用这么一种简单的模型即可。文献[52]中谈到的例子就只需要这种简单的模型。其它的模型也都有实际系统与之对应。这一点也是其它访问控制策略所没有的特性。

4. 完全独立于其它安全手段，是策略中立的。

对于一个安全的访问控制系统来说，策略中立是十分重要的。访问控制本身必须不依赖于原有系统的任何其它安全措施，这样才能使得该访问控制模型在不同的应用背景下都能得到应用。RBAC 模型很好的做到了这一点，使得它可以无缝结合到一般的安全产品中去。这也是我们第五章设计 RBAC 中间件的理论基础。

另外，RBAC 模型中角色的概念有点类似传统的用户组（group）的概念，但是它们是有区别的。用户组是一部分具有类似属性的用户集合，只能代表一组用户的属性。在权限配置过程中，还是需要针对不同的用户组赋予相应的权限。而角色是一个更抽象的概念，它可以代表一组用户（文献[48]中称为 Groups），也可以代表一组权限（文献[48]中称为 Abilities），还可以代表纯粹的没有具体用户组或者权限组对应的角色（文献[48]中称为 UP-Roles）。因此在 RBAC 模型中，角色的概念比传统的用户组的概念丰富的多，因此也能够更灵活的控制和管理整个访问控制模型。

第3章 多维 RBAC 模型及其应用

本章介绍 RBAC 模型的一种扩展：多维 RBAC 模型（MDRBAC，Muiti-Dimentional RBAC Model），并给出基于多维 RBAC 模型的一种分布式角色管理模型 MDARBAC（Muiti-Dimentional Administrative RBAC Model）。首先我们介绍传统的 RBAC 模型的缺陷，然后给出多维 RBAC 模型的形式化描述与多维 RBAC 角色管理模型，最后给出多维 RBAC 模型的实现及其应用。

3.1 多维 RBAC 模型的提出

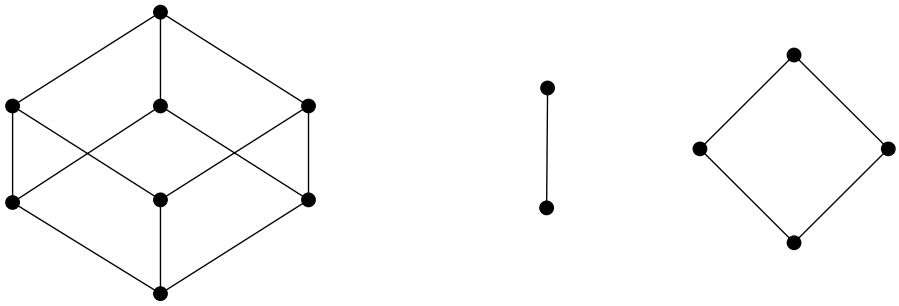
在传统的 RBAC 模型中，角色模型本身并没有任何结构，角色的语义也不清楚。在一个实际的应用系统中，如何合理的定义和配置角色一直都是一个很头疼的问题。文献[25]利用“使用情况语义”（concept of use cases）来决定角色的权限，文献[32]给出了一个更完整的角色模型并讨论了角色的各种属性，但是有关角色的语义仍不清楚，我们缺乏一个理论框架来系统地描述角色语义并指出如何利用角色语义简化角色层次关系。

以为企业管理设计的访问控制系统为例。企业内部存在诸多机构层次，每个机构下面一般都有一些固定的职务。在权限管理里面必须对所有的这些机构和职务进行角色定义和管理。虽然不同机构具有类似的属性，每个机构下面的同一个职务也有类似的设置，管理员也只有对每个机构的每个职务进行单独管理，无法直接管理一定规模的角色。我们提出的 MDRBAC 模型和 MDARBAC 模型就是为了一定程度的解决这个问题而提出的[3]。

为了更清楚地说明 MDRBAC 模型的直观意义，这里我们给出一个实例进行说明。

假设某公司由一个总公司和一个分公司两个行政部门组成，每个行政部门都有四个人员级别：经理，开发人员，销售人员和普通职员，我们可以得到八个角色。如果权限定义为对公司 Web 页面的访问，直观上，在每个行政部门内部经理继承开发人员和销售人员的权限，这二者继承普通职员的权限。然后总公司的相应级别继承分公司的相应级别的权限。因此我们可以得到如图 3.1(a)所示的角

色层次关系图，图中默认有向边的方向为自上而下。



(a) 角色层次关系图　　(b) 行政部门语义　　(c) 人员级别语义

图 3.1. 角色层次关系图实例

这里每个角色实际上是由两个语义维组成的：一个是行政部门，一个是行政部门内部的人员级别。图 3.1(a)的层次关系混杂了两种语义，不易理解。如果将每个角色的相应单元取出，我们可以得到两个图(b)和(c)，分别刻画了行政部门的继承关系和人员级别的继承关系。实际上，图 3.1(a)是(b)和(c)的直积。通过这样一个直积分解，每一个语义维的层次关系可以单独刻画和管理，并通过直积直接反映到全局的层次关系上。我们通过管理图 3.1(b)和(c)中的六个角色单元，就可以实施对图 3.1(a)中八个角色的管理。

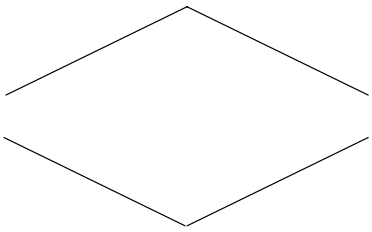
总公司开发

我们在下一节给出多维 RBAC 模型的形式化定义，并仍以图 3.1 为例讲述多维 RBAC 模型的管理模式。

3.2 多维 RBAC 模型 MDRBAC

分公司开发

类似于 RBAC96 模型，我们定义四个层面的 MDRBAC 模型，分别称为 $MDRBAC_0$, $MDRBAC_1$, $MDRBAC_2$ 和 $MDRBAC_3$，它们具有如图 3.2 所示的包含



分公司

图 3.2. MDRBAC 模型的包含关系

关系，与 RBAC96 模型中的图 2.2 类似。

3.2.1 $MDRBAC_0$

基础模型 $MDRBAC_0$ 定义如下。

定义 3.1 $MDRBAC_0$ 模型包含如下元素：

1) 若干实体集 U（用户集），P（权限集），S（会话集）。
2) d 个互不相交的实体集 $VR_1, VR_2, \cdots, VR_d$，称为虚拟角色维（virtual role set），其中的元素称为虚拟角色（virtual role），d 称为角色维数（role dimensionality）。
3) 角色集 R 为各个虚拟角色维的直积，即 $R = \bigotimes_{i=1}^{d} VR_i$。
4) $UA \subseteq U \times R$，为多对多的用户角色指派关系。
5) $PA \subseteq P \times R$，为多对多的权限角色指派关系。
6) $user: S \to U$，映射每个会话到一个用户。
7) $roles: S \to 2^R$，映射每个会话到一组角色，有 $roles(s) \subseteq \{r \mid (user(s), r) \in UA\}$。 ∎

对比 RBAC96 模型，我们没有直接定义角色集，而是首先定义虚拟角色维，根据定义，$\forall r \in R$，$\exists r_i \in VR_i \ (i = 1, 2, \cdots, d)$，使得 $r = (r_1, r_2, \cdots, r_d)$，我们称这种表示法为该角色的维数表示，每个分量对应一个虚拟角色。从语义上来说，每个虚拟角色维代表了一定的语义信息，每个角色在每个语义集中都有一个分量。不同的虚拟角色维不允许相交，保证了这种语义信息的独立性。在图 3.1 中，(b)和(c)分别对应一个虚拟角色维，并且通过直积得到角色集(a)。

模型中如果 $d = 1$，则角色集由唯一的虚拟角色维组成，角色与虚拟角色等价。因此 $MDRBAC_0$ 模型退化到 $RBAC_0$ 模型。对于下面将要定义的其它模型也有类似结果。因此可以认为 MDRBAC 模型是 RBAC96 模型的一种扩展。

3.2.2 $MDRBAC_1$

在 $MDRBAC_0$ 基础上通过增加角色继承关系定义 $MDRBAC_1$。

定义 3.2 $MDRBAC_1$ 模型包含如下元素：

1) $U, P, S, VR_i, R, UA, PA, user$ 同定义 3.1。

2) $VRH_i \in VR_i \times VR_i$ 是集合 VR_i 上的一个偏序关系，记为 $\geq_i$ $(i = 1,2,\cdots,d)$。

3) 隐式角色层次 $IRH \in R \times R$ 是 R 上的一个关系，记为 $\geq_{IRH}$。设 $u = (u_1, u_2, \cdots, u_d)$，$v = (v_1, v_2, \cdots, v_d) \in R$，定义 $u \geq_{IRH} v \Leftrightarrow \bigcap_{i=1}^{d} u_i \geq_i v_i$。

4) 显式角色层次 $ERH \in R \times R$ 是 R 上的偏序关系，记为 $\geq_{ERH}$。

5) $RH \in R \times R$ 是 R 上的一个关系，记为 $\geq$。$\forall u, v \in R$，$u \geq v$ 当且仅当

   a) $u \geq_{IRH} v$，或

   b) $u \geq_{ERH} v$，或

   c) 存在 $w \in R$，有 $u \geq w$ 且 $w \geq v$。

6) 显式角色层次限制条件 $ERHC: \forall u, v \in R[u \geq v \wedge v \geq u \Rightarrow u = v]$，规定 $\geq_{ERH}$ 必须满足 $ERHC$ 条件。

7) $roles: S \to 2^R$ 作如下改动：$roles(s) \subseteq \{r \mid (\exists r' \geq r)[(user(s), r') \in UA]\}$。

∎

$MDRBAC_1$ 的角色层次比 $RBAC_1$ 复杂得多。我们首先在每个虚拟角色维上定义偏序关系 VRH，由于角色集 R 是虚拟角色维的直积，我们可以直接得到 R 上的一种关系 IRH，称之为隐式角色层次。

我们有如下定理保证 IRH 是一个偏序。

定理 3.1 IRH 是一个偏序。

证明 根据定义 3.2，$u \geq_{IRH} v \Leftrightarrow \bigcap_{i=1}^{d} u_i \geq_i v_i$，其中每个 $\geq_i$ 均为偏序。设 $u = (u_1, u_2, \cdots, u_d)$，$v = (v_1, v_2, \cdots, v_d)$，$w = (w_1, w_2, \cdots, w_d) \in R$，则

1) 由于$\geq_i$为偏序，根据自反性我们有$u_i \geq_i u_i \ (i=1,2,\cdots,d)$，因此$\bigcap_{i=1}^{d} u_i \geq_i u_i$为真，$u \geq_{IRH} u$。

2) 设$u \geq_{IRH} v$，且$v \geq_{IRH} u$，根据定义有$u_i \geq_i v_i$，且$v_i \geq_i u_i \ (i=1,2,\cdots,d)$。由$\geq_i$为偏序，根据反对称性我们有$u_i = v_i \ (i=1,2,\cdots,d)$，因此有$u = v$。

3) 设$u \geq_{IRH} v$，$v \geq_{IRH} w$，根据定义有$u_i \geq_i v_i$，且$v_i \geq_i w_i \ (i=1,2,\cdots,d)$。由$\geq_i$为偏序，根据传递性我们有$u_i \geq_i w_i \ (i=1,2,\cdots,d)$，因此有$u \geq_{IRH} w$。

因此得 *IRH* 是一个偏序。 ∎

与隐式角色层次相反，显式角色层次 *ERH* 是直接定义在 *R* 上的偏序。*R* 上完整的角色层次 *RH* 是这二者的一种递归组合

如下定理保证 *RH* 是一个偏序。

定理 3.2 如果 *ERH* 满足 *ERHC* 条件，$\geq$是一个偏序。

证明 设$u,v,w \in R$，则

1) 由于$\geq_{IRH}$是一个偏序，由自反性我们有$u \geq_{IRH} u$。根据定义 3.2，$u \geq u$。

2) 设$u \geq v$，且$v \geq u$。由$\geq_{ERH}$满足 *ERHC* 条件，我们有$u = v$。反对称性成立。

3) 设$u \geq v$，$v \geq w$，则由$\geq$的定义知$u \geq w$。

因此得$\geq$是一个偏序。 ∎

ERHC 是一个很强的限制条件，它同时保证了 *ERH* 和 *IRH* 之间并不存在矛盾，即$u \geq_{ERH} v \Rightarrow \neg(v \geq_{IRH} u)$。

在角色管理中，某个VR_i中的一个偏序发生变化将引起 *IRH* 中多条偏序变化，因此隐式角色层次管理对应于粗粒度的角色管理；而 *ERH* 是直接定义在角色集 *R* 上的偏序，因此显式角色层次管理对应于细粒度的角色管理。

对应于图 3.1 的实例，如果我们将总公司和分公司分别用 1 和 0 表示，将经理，开发人员，销售人员和普通职员分别用 ABC，AC，BC，C 表示，则我们可以得到如图 3.3 所示的角色直积图。图中左边是两个虚拟角色维 VR_1 和 VR_2，其中 1，0，ABC，AC，BC，C 都是虚拟角色，在两个虚拟角色维上定义的偏序关

系分别为 VRH_1 和 VRH_2。右边是它们直积得到的角色偏序集 R，其中实线是 IRH 偏序，虚线是 ERH 偏序。这里 ERH 与图 3.1(a)不符，仅用作示例。

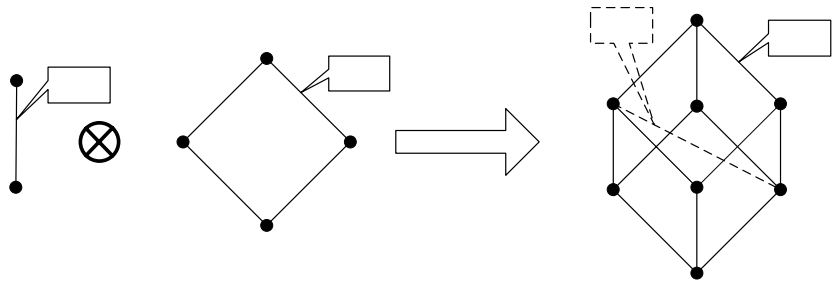


图 3.3. MDRBAC 模型角色直积以及偏序关系定义实例

3.2.3 $MDRBAC_2$

在 $MDRBAC_0$ 基础上通过增加限制关系定义 $MDRBAC_2$。

定义 3.3 $MDRBAC_2$ 模型包含如下元素：

1) $MDRBAC_0$ 中的所有元素。

2) 一组限制条件，用于刻画 $MDRBAC_0$ 中各元素的组合合法性。 ∎

和 $RBAC_2$ 一样，我们这里没有明确定义限制的具体形式。但是需要注意的是，这里限制的作用域除了 $RBAC_0$ 中的元素之外，还包括各个虚拟角色维。因此，我们能够更灵活的控制限制的发生条件和作用范围。

根据发生的阶段不同，限制可以分为静态限制和动态限制，前者定义在角色授权阶段，后者定义在角色激活阶段。角色基数限制和角色互斥是最重要的两种限制。我们将在下一节给出 $MDRBAC_2$ 模型中相关的定义策略。

3.2.4 $MDRBAC_3$

现在我们给出完整模型 $MDRBAC_3$，它包含 $MDRBAC_1$ 和 $MDRBAC_2$ 的所有元素。图 3.4 诠释了整个 $MDRBAC_3$ 模型框架。

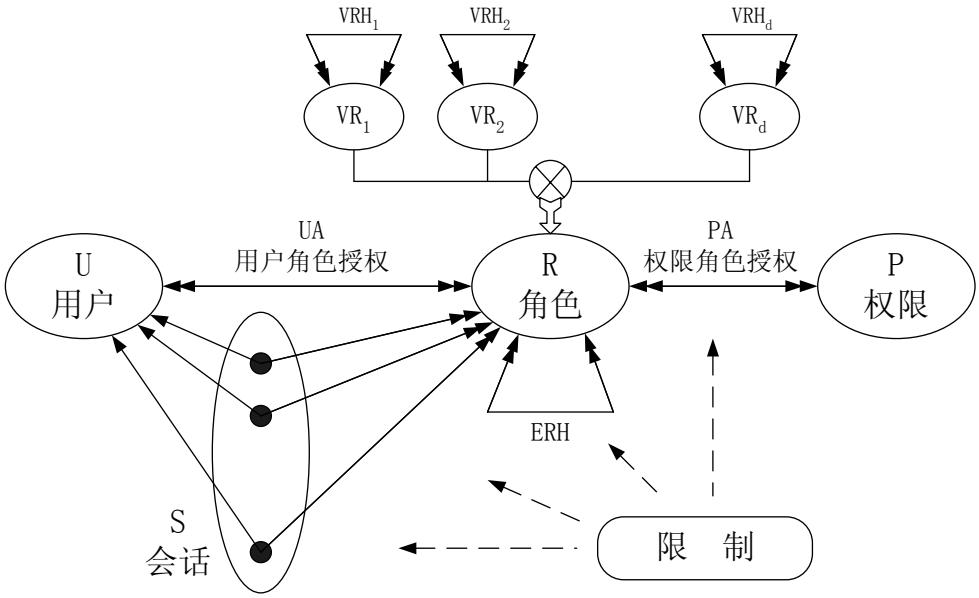
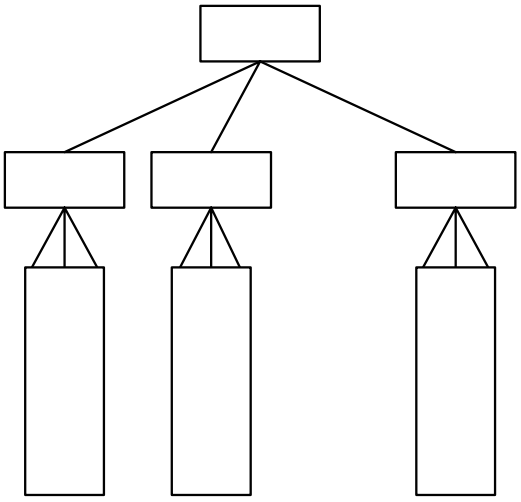


图 3.4. MDRBAC$_3$ 模型框架

3.3 多维 RBAC 角色管理模型 MDARBAC

在多维 RBAC 的模型框架下，RBAC 管理模型会引进一些新的特性。我们称之为多维 RBAC 角色管理模型 MDARBAC（Multi-Dimensional Administrative RBAC Model）。和 ARBAC97 模型类似，我们首先给出 MDARBAC 模型的整体框架，然后分为 MDURA，MDPRA，MDRRA 三部分来讲述。

3.3.1 多维 RBAC 分布式角色管理模型



user

图 3.5. 多维分布式角色管理模型 MDARBAC

在 MDRBAC 模型中，每一个虚拟角色维包含若干虚拟角色，也存在一定的层次关系和限制条件。如果把每个虚拟角色维看作是一个独立的角色偏序集，则可以给每个虚拟角色维指定一个最高管理员（称为 DSO，Dimension Security Officer），负责管理相应的虚拟角色维。这样就形成了一个分工明确的分布式管理模型，称之为 MDARBAC 模型，如图 3.5 所示。

MDARBAC 模型中，每个 DSO 实际上是他（她）对应的虚拟角色维上的最高管理员，因此在该虚拟角色维上可以运用 ARBAC97 模型框架将管理员角色进行进一步的细分。由于语义关系明确，虚拟角色相对较少，一个虚拟角色维内部的分布式管理框架较容易确定。针对某一个虚拟角色维，可以有一个单独的管理员角色层次图来实现对该虚拟角色维的管理。该虚拟角色维上的管理员角色层次图的创建和角色管理的全部权限都交由该维上的 DSO 负责。系统最高管理员 SSO（Senior Security Officer）继承所有 DSO 的权限，负责协调各 DSO 进行协作，可进行全局角色管理及角色细化管理。

当 $d=1$ 时，MDARBAC 模型退化到 ARBAC97 模型，因此 ARBAC97 模型实际上是 MDARBAC 模型的一个特例。在利用虚拟角色维作了顶层的分布式管理划分之后，每一个虚拟角色维可以使用 ARBAC97 模型实施角色管理定制。

3.3.2 基于角色维数的用户角色指派 MDURA

ARBAC97 模型已经对用户角色授权管理作了理论上的分析，并提出了 URA97 模型。在 URA97 模型中，定义了关系 can_assign 和 can_revoke，分别用来刻画授权（Grant）模型和撤销（Revoke）模型。但是对于一个实际的系统来说，究竟如何高效的完成用户角色的授权仍然是一个十分复杂的问题。

利用角色维数，我们可以考虑一种分阶段的用户角色授权模型：

1. 系统最高管理员 SSO（或者与各 DSO 协商）根据各个角色维数的重要性程度（或者参考其他指数）将各角色维排序，设排序结果为 L_1，L_2，…，L_d。
2. 管理员依次授权该用户在各个角色维中的角色分量。如果系统仅存在一个管理员 SSO，则 SSO 负责授权该用户的每个角色分量；如果存在多个域管理员 DSO，则每个 DSO 负责在他所管辖的维数上给该用户指定一

个角色分量。针对每一个角色维管理员可以考虑该用户的不同属性。在每一个角色维上还可以采用 URA97 模型细化用户角色指派。

3. 假设对应于角色维 L_i 的角色分量是 r_i，则经过步骤 2，该用户被授权角色 $r=(r_1,r_2,\cdots,r_n)$。

重复步骤 2 和 3，管理员可以给用户授权任意的角色。

通过角色维数，管理员可以方便的针对用户的属性在每个角色维给用户指定虚拟角色，从而有效地完成赋权操作。

同样，我们利用角色维数可以考虑一种有层次的用户角色撤销模型。管理员可以方便的在一个角色维内部改变用户的虚拟角色而不影响到该用户的其他虚拟角色。当然，如果需要彻底撤销一个用户对应的一个完整角色，管理员可以直接撤销该用户对应的一个完整角色的所有角色分量。

3.3.3 基于角色维数的权限角色指派 MDPRA

对于系统管理员来说，权限角色指派是一件十分复杂的工作，需要耗费大量的时间。我们可以利用角色维数简化这里的工作。

在实际工作中我们发现，如果两个角色在某一个角色维数上的角色分量是相同的，那么在进行权限指派的时候，它们具有相当的相似性。在图 3.1 所示的例子中，如果总公司下设两个分公司 M 和 N，那么分公司 M 的开发人员和分公司 N 的开发人员就会具有十分类似的访问权限，比如说他们能够访问资源的区别只是分公司的名字不同。这种情况下，我们只需要详细的定义分公司 M 的开发人员的访问权限，就可以很容易的给出分公司 N 的开发人员的权限。

这种有共同的角色分量的角色在权限角色指派时候的相似性会随着角色维的选取不同而不同，而且和所考虑的应用领域也有关。很多时候，这种相似性还和应用系统中资源的表述形式有关。究竟如何利用角色维数进行简化需要管理员针对特定的应用领域进行权衡。比如上例中我们考虑同一个分公司的开发人员和销售人员，虽然这两个角色在行政级别这个角色维上的角色分量是相同的，但是在进行权限角色指派的时候并不具有很大的相似性。

3.3.4 基于角色维数的角色管理 MDRRA

RBAC 系统中的角色管理主要包括角色的添加和删除，角色继承关系的添加和删除，以及角色间的限制关系。MDRBAC 模型引入角色维数的概念，既可通过虚拟角色维实行角色的粗粒度管理，又可直接进行角色的细粒度管理。对于实际系统，我们推荐以粗粒度管理为主，细粒度管理为辅的角色管理策略。

3.3.4.1 角色的添加和删除

在 MDRBAC 中，角色通过其维数表示得到。我们考虑两种情形：虚拟角色的添加和删除以及角色的添加和删除。

每个虚拟角色对应一个虚拟角色维，添加虚拟角色只需要考虑该虚拟角色维上的限制关系。通常的限制条件包括不能重复添加，不能产生继承循环等等。添加了一个虚拟角色之后，直积后的系统角色集将增添一系列的角色。添加虚拟角色的例子包括在 3.1 节的实例中增设一个分公司，以及在每个行政部门内增设一个测试人员等等。

与添加类似，虚拟角色的删除将在一个虚拟角色维内部进行。删除一个虚拟角色同时必须删除所有具有该分量的系统角色。

MDRBAC 模型中角色的添加实际上已经弱化为一条无效指令，这是因为所有的角色都在虚拟角色建立的同时建立好了。如果待添加角色的每个分量都已存在于相应的虚拟角色维中，那么该角色已经存在；如果某个分量还不存在，那么相当于在相应的虚拟角色维添加该分量对应的虚拟角色。

角色删除利用角色基数限制实现。将该角色的静态基数设为零即可保证没有用户和权限能与之产生授权关系。

3.3.4.2 角色继承关系的添加和删除

角色继承关系的添加和删除也有两个层面：虚拟角色继承关系的添加和删除以及角色继承关系的添加和删除。

虚拟角色继承关系的添加和删除只在一个虚拟角色维上考虑有效性，但继承关系一旦修改，将影响到所有具有该分量的系统角色。这种影响通过 *IRH* 直接

表现出来，并通过 *RH* 间接反映。为保证系统角色偏序不出现环路情形，添加虚拟角色时应考虑一定的有效性检验。这种检验一方面在所添加的虚拟角色维内部，另一方面在整个系统角色集上。

角色继承关系的添加和删除实际上就是修改 *ERH* 关系。由于 *ERH* 关系将影响 *RH* 关系，因此在添加角色继承关系时同样需要做相关的有效性检验。同时，在添加 *ERH* 记录时一定要考虑 *ERHC* 限制条件。

3.3.4.3 角色基数限制

一个角色允许被授权的最大用户数称为该角色的静态基数，允许同时被激活的最大用户数称为该角色的动态基数。显然，角色的动态基数应小于或等于它的静态基数。在 MDRBAC 模型中，我们有如下的角色基数定义策略。

定义 3.4 (虚拟角色基数). 设集合 N 包含零和所有自然数。$Card_i^s : VR_i \to N$ 定义了 VR_i 中虚拟角色的静态基数；$Card_i^d : VR_i \to N$ 定义了 VR_i 中虚拟角色的动态基数。 ∎

定义 3.5 (角色基数). $Card^s : R \to N$ 定义了 R 中角色的静态基数；$Card^d : R \to N$ 定义了 R 中角色的动态基数；设角色 $r = (r_1, r_2, \cdots, r_d) \in R$，定义

$$Card^s(r) = \min(Card_1^s(r_1), Card_2^s(r_2), \cdots, Card_d^s(r_d))，$$

$$Card^d(r) = \min(Card_1^d(r_1), Card_2^d(r_2), \cdots, Card_d^d(r_d))。$$ ∎

在这种策略下，首先根据语义关系在每个虚拟角色维内部定义虚拟角色基数，然后取每个角色分量的虚拟基数的极小作为整个角色的基数。这是一种粗粒度的角色基数定义策略。对于实际中不可避免的对某个系统角色进行细粒度的基数维护，系统管理员可强行定义一个新的基数赋予该角色，比如前面角色删除的情形。

3.3.4.4 角色互斥

角色互斥控制角色的职能冲突，用于职责分离定义。两个角色静态互斥是指

一个用户不能同时被授权这两个角色，两个角色动态互斥是指一个用户可以同时被授权这两个角色，但不能在一个会话中同时激活这两个角色。

角色互斥可有多种考虑角度[34]，除了静态互斥/动态互斥之外，还有部分互斥/完全互斥（考虑互斥角色对应的权限集是否有交叉）、两两互斥/集合互斥（考虑两个角色互斥还是两组角色互斥）等等。不失一般性，我们这里给出角色间动态、完全、两两互斥的定义策略。

定义 3.6 (虚拟角色互斥). $VRME_i \in VR_i \times VR_i$ 定义了 VR_i 上的角色互斥关系，记作 $\#_i$。 ∎

定义 3.7 (角色互斥). $RME \in R \times R$ 定义了 R 上的角色互斥关系，统一记作 $\#$；设 $u=(u_1,u_2,\cdots,u_d)$， $v=(v_1,v_2,\cdots,v_d) \in R$，定义如下三种角色互斥策略：

严格互斥： $u \#_s v \Leftrightarrow \exists i\left[(u_i \#_i v_i) \wedge \forall j(j \neq i) \Rightarrow (u_j = v_j)\right]$

累进互斥： $u \#_p v \Leftrightarrow \exists i\left[(u_i \#_i v_i) \wedge \forall j(j \neq i) \Rightarrow (u_j = v_j) \vee (u_j \#_j v_j)\right]$

松散互斥： $u \#_l v \Leftrightarrow \exists i(u_i \#_i v_i)$ ∎

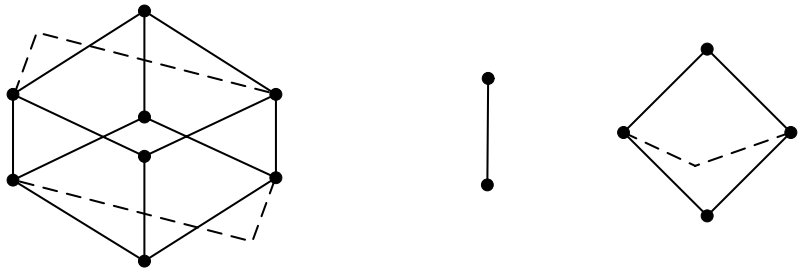
根据定义 3.7，严格互斥策略定义分量中仅有一对互斥其余相等的两个角色为互斥；累进互斥策略定义分量中至少有一对互斥其余的要么相等要么互斥的两个角色为互斥；而松散互斥策略定义存在一对互斥分量的两个角色为互斥。如下定理刻画了这几种互斥策略之间的关系。

定理 3.3 $\forall u,v \in R$， $u \#_s v \Rightarrow u \#_p v$， $u \#_p v \Rightarrow u \#_l v$。

证明 由定义及演绎推理可直接推得结果。 ∎

根据这种角色互斥策略，管理员先对一个虚拟角色维内的虚拟角色定义互斥关系，然后选取不同的互斥策略，系统即可自动生成角色的互斥关系定义。定义虚拟角色互斥必须考虑相关的限制条件，比如这两个虚拟角色不能存在继承关系等。

在图 3.1 中，如果开发人员和销售人员被定义为严格互斥，那么在完整的角色图中，所有行政部门的开发人员和销售人员都自动被定义为互斥。我们只需给出一条互斥的语句定义(如图 3.6(c)所示)即可管理一系列的角色互斥关系(如图 3.6(a)所示)。



(a) 定义了互斥关系的角色层次图　　(b) VR_1　　(c) VR_2 包括互斥关系

图 3.6. 定义了互斥关系的角色层次图与两个虚拟角色维

3.4 多维 RBAC 模型的实现

我们重点介绍 MDRBAC 模型中有关角色管理的实现机制。在粗粒度的角色管理中，我们只需要记录虚拟角色集内部虚拟角色的层次关系和限制条件；对于细粒度的角色管理，我们需要考虑完整角色之间的关系。这里模型实现的关键问题就是角色在系统中的表示以及角色关系的刻画。利用虚拟角色集和下面介绍的角色命名机制，我们可以极大的简化角色维护工作。

在下面的介绍中，我们会使用一个表 RRA 来记录角色与角色之间的关系。注意这里的角色包括系统原有的角色以及虚拟角色。这个表具有如下所示的格式：

[角色名称 1，角色名称 2，角色关系]

其中角色名称我们将在下面介绍，角色关系取自集合{>, #}，其中>指继承关系[1]，#指互斥关系。我们给定分隔符集 $Sep=\{/_1,/_2,\cdots,/_d\}$，其中 $/_i$ 对应虚拟角色维 $VR_i\ (i=1,2,\cdots,d)$。

3.4.1 虚拟角色名称定义

在每个虚拟角色维内部，我们首先给每个虚拟角色一个语义名，然后自顶向下定义虚拟角色名称。如果该虚拟角色维不含最大元，我们指定一个虚拟的最大元 r_max（语义名为 r_max_name）。我们假定当前虚拟角色维的分隔符为 /。递

[1] $\forall u,v\in R$，$u>v \Leftrightarrow u\geq v \wedge u\neq v$。

归策略如下：

1．最大元的虚拟角色名称即为它的语义名或 r_max_name。

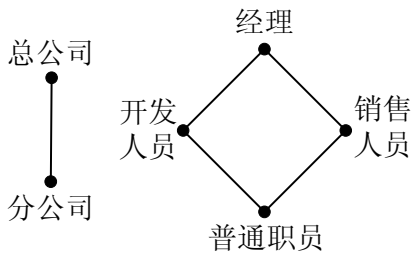
2．对于下级虚拟角色 r，设它的语义名为 r_name，它的直接父虚拟角色集为 UP(r)。

a) 如果 UP(r)只含一个虚拟角色，设为 rf，语义名为 rf_name，那么定义 r 的虚拟角色名称为 rf_name/r_name。

b) 如果 UP(r)不止一个角色，则我们在其中选取一个角色 rf，语义名为 rf_name，定义 r 的虚拟角色名称为 rf_name/r_name。然后对于 UP(r)中剩下的每一个虚拟角色 rfe，设其语义名为 rfe_name，在 RRA 表中添加记录 [rfe_name, rf_name/r_name, >]。

采用这种定义方法，虚拟角色名称本身就已经包含了一定的层次关系，这样可以大大减少 RRA 表中的数据量。对于自顶向下呈树状结构的虚拟角色层次图，我们可以完全舍弃 RRA 表，直接利用虚拟角色名称确定虚拟角色之间的层次关系。

对应于图 3.7(a)的虚拟角色层次图，图 3.7(b)给出了每个虚拟角色的语义名及其对应的虚拟角色名称。图 3.7(c)给出了添加到 RRA 表中的记录。



(a)

虚拟角色	语义名称	虚拟角色名称
总公司	company	company
分公司	filiale	company/$_1$filiale
经理	manager	manager
开发人员	developer	manager/$_2$developer
销售人员	salesman	manager/$_2$salesman
普通职员	employee	manager/$_2$developer/$_2$employee

(b)

角色名称 1	角色名称 2	角色关系
manager/$_2$salesman	manager/$_2$developer/$_2$employee	>

(c)

图 3.7. 虚拟角色名称定义示例

(a) 两个角色维上的虚拟角色层次关系; (b) 每个虚拟角色的语义名称和虚拟角色名称;

(c) 定义虚拟角色名称之后的 RRA 表

采用了这种表示法之后，我们就有一个十分直观高效的算法 IsVirtualInherit 确定两个虚拟角色是否存在继承关系。算法递归的利用虚拟角色名称和 RRA 表里的记录来判断虚拟角色之间的继承关系。算法的伪代码描述见图 3.8。

```
Algorithm: IsVirtualInherit
Input: VirtualRole vr1, vr2
Output: Bool
Description: Return true if virtual role vr1 inherits vr2 and false else
Begin
    IF vr1.name in vr2.name THEN       //use string manipulation
        RETURN true            //just the role name can judge
    ENDIF
    recordset = records in RRA with the father role to be vr1
    FOR EACH record IN recordset DO
        IF vr2 = record.sonrole THEN
            IF record.relation = '>' THEN RETURN true   //RRA table tells
            ELSE RETURN false         //they are mutual exclusive
            ENDIF
            IF IsVirtualInherit(record.sonrole, vr2) THEN        //recursively judge
                RETURN true
            ENDIF
        ENDIF
    ENDFOR
    RETURN false
End
```

图 3.8. 算法 IsVirtualInherit

3.4.2 角色名称定义

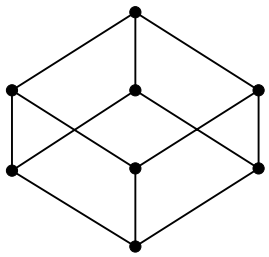
通过写成维数表示的形式，我们可以利用虚拟角色名称的向量形式表示一个角色。比如前例中总公司销售人员的角色可以写成(company, manager/$_2$salesman)。

由于角色可以写成维数表示的形式，我们采用如下方式合并各个虚拟角色分量，得到一个完整的角色名称。设角色 r 的维数表示为$(r_1, r_2, \cdots, r_d)$，并设$\overline{r_i}$ 为 r_i 的虚拟角色名称，则我们定义 r 的角色名称为$\overline{r_1}/_1\overline{r_2}/_2\cdots\overline{r_{d-1}}/_{d-1}\overline{r_d}$，其中$/_i$为虚拟角色维$VR_i$上的分隔符$(i=1,2,\cdots,d)$。

由于除了虚拟角色之间的分隔符之外，只有$\overline{r_i}$ 中包含分隔符$/_i$，因此给定一个角色名称，最后一个$/_1$之前的部分为$\overline{r_1}$，最后一个$/_{i-1}$和最后一个$/_i$之间的部分

为 $\overline{r_i}\,(i=2,\cdots,d-1)$，最后一个 $/_{d-1}$ 之后的部分为 $\overline{r_d}$。这种角色名称定义完全标识出了一个角色在各个虚拟角色维上的角色分量，并与该角色的维数表示一一对应。

前例中我们可以得到如图 3.9(b)所示的角色名称。



(a)

角色	角色名称
总公司经理	company/$_1$manager
分公司经理	company/$_1$filiale/$_1$manager
总公司开发	company/$_1$manager/$_2$developer
分公司开发	company/$_1$filiale/$_1$manager/$_2$developer
总公司销售	company/$_1$manager/$_2$salesman
分公司销售	company/$_1$filiale/$_1$manager/$_2$salesman
总公司职员	company/$_1$manager/$_2$developer/$_2$employee
分公司职员	company/$_1$filiale/$_1$manager/$_2$developer/$_2$employee

(b)

图 3.9. 角色的完整名称定义

(a) 角色继承关系图; (b) 各个角色及其对应的完整的角色名称

根据这里介绍的虚拟角色和角色的命名方法，虚拟角色名称本身就已经包含了一定的层次关系，大大减少了 RRA 表中的数据量。只要给定两个角色 u,v 及其角色名称，我们就可得到它们的虚拟角色分量并判断是否有 $u \geq_{IRH} v$，极大的提高了效率。要判断是否 $u \geq v$，我们还应该考虑直接添加到 RRA 表中的 *ERH* 偏序。

设从角色名称到其维数表示的过程为 GetRoleVector，则我们可以定义判断两个角色是否存在继承关系的算法 IsInherit。该算法主要是判断两个角色的每对虚拟角色是否存在继承关系。算法的伪代码见图 3.10。

总公司经理

```
Algorithm: IsInherit
Input: Role r1, r2; Integer N
Output: Bool
Description: Return true if role r1 inherits r2 and false else. N is the role dimension
Begin
    RoleVector rv1 = GetRoleVector(r1)
    RoleVector rv2 = GetRoleVector(r2)
    FOR i = 1 TO N DO
        IF (!IsVirtualInherit (rv1[i], rv2[i]) ) THEN
```

总公司开发

总公

分公司经理

总公司职员

```
            BREAK
        ENDIF
        RETURN true
    ENDFOR
    recordset = records in RRA with the father role to be r1
    FOR EACH record IN recordset DO
        IF r2 = record.sonrole THEN
            IF record.relation = '>' THEN RETURN true   //RRA table tells
            ELSE RETURN false         //they are mutual exclusive
            ENDIF
            IF IsInherit(record.sonrole, r2) THEN          //recursively judge
                RETURN true
            ENDIF
        ENDIF
    ENDFOR
    RETURN false
End
```

图 3.10. 算法 IsInherit

3.4.3 角色管理算法的形式化描述

这一节我们将给出前面讲述的角色管理算法的形式化描述，包括添加和删除角色，添加和删除角色继承关系，添加和删除限制等。

3.4.3.1 添加角色

这里我们考虑两种角色添加算法：添加角色算法 AddRole 以及添加虚拟角色算法 AddVirtualRole。由于我们这里的角色实际上是由虚拟角色构成的，因此前者实际上是在每个角色维调用后者。后者算法相对简单标准。

```
Algorithm: AddRole
Input: Role r
Output: Bool
Description: Return true if successfully add a role

Begin
    RoleVector rv = GetRoleVector(r)
    FOR EACH vr IN rv DO
        IF !AddVirtualRole(vr) THEN
            RETURN false
```

```
Algorithm: AddVirtualRole
Input: VirtualRole vr
Output: Bool
Description: Return true if successfully add a virtual role

Begin
    IF Exist(vr) THEN
        RETURN true
    Add vr into this dimension considering its virtual role name
```

```
        ENDIF
    ENDFOR
    RETURN true
End
```

```
    RETURN true
End
```

图 3.11. 添加角色算法

3.4.3.2 删除角色

与添加角色类似，我们考虑两种角色删除算法：删除角色算法 DeleteRole 以及删除虚拟角色算法 DeleteVirtualRole。前者首先删除相关的授权信息，然后将该角色的基数设置为零。后者循环调用前者，并做实际删除操作。

```
Algorithm: DeleteRole
Input: Role r
Output: Bool
Description: Return true if successfully delete a role

Begin
    Delete URA and PRA records related to role r
    r.cardinality = 0
    RETURN true
End
```

```
Algorithm: DeleteVirtualRole
Input: VirtualRole vr
Output: Bool
Description: Return true if successfully delete a virtual role

Begin
    FOR EACH r that contains vr DO
        DeleteRole(r)
    ENDFOR
    Delete vr in its virtual role set and preserve the implied relations of other virtual roles
    RETURN true
End
```

图 3.12. 删除角色算法

3.4.3.3 添加角色继承关系

由于角色继承关系由隐式角色层次和显式角色层次构成，添加角色继承关系也分为添加角色层次 AddInheritance 和添加虚拟角色层次 AddVirtualInheritance 两部分构成。前者实际上是添加显式角色层次，后者实现在一个角色维上添加一个虚拟角色继承关系。二者都需要考虑一定的约束条件，包括不能出现环路，不能已经定义为互斥等。

```
Algorithm: AddInheritance
Input: Role r1, r2
Output: Bool
```

```
Algorithm: AddVirtualInheritance
Input: VirtualRole vr1, vr2
Output: Bool
```

```
Description: Return true if successfully add an explicit role inheritance

Begin
    IF IsInherit(r2, r1) THEN
        RETURN false
    ENDIF
    IF IsMutualExclusive(r1, r2) THEN
        RETURN false
    ENDIF
    Add (r1, r2) into ERH
    RETURN true
End
```

```
Description: Return true if successfully add a virtual inheritance

Begin
    IF IsVirtualInherit(vr2, vr1) THEN
        RETURN false
    ENDIF
    IF IsVirtualMutualExclusive(vr1, vr2)
    THEN
        RETURN false
    ENDIF
    Add (vr1, vr2) into corresponding VRH
    RETURN true
End
```

图 3.13. 添加角色继承关系算法

3.4.3.4 删除角色继承关系

删除角色继承关系也包括删除角色继承关系 DeleteInheritance 和删除虚拟角色继承关系 DeleteVirtualInheritance 两种。前者考虑删除显式角色层次，后者考虑删除一个角色维上的隐式角色层次。它们二者都需要考虑由于删除继承关系导致的权限继承的改变。由于这里可以考虑的情形较多，我们在算法里只做了大致说明。有关删除角色继承关系的详细说明请参看本文第五章有关中间件的叙述。

```
Algorithm: DeleteInheritance
Input: Role r1, r2
Output: Bool
Description: Return true if successfully delete an explicit role inheritance

Begin
    IF Exist(r1, r2) in ERH THEN
        Delete (r1, r2) from ERH
        Preserve RRA if needed
        Preserve URA if needed
        Preserve PRA if needed
    ENDIF
    RETURN true
End
```

```
Algorithm: DeleteVirtualInheritance
Input: VirtualRole vr1, vr2
Output: Bool
Description: Return true if successfully delete a virtual inheritance

Begin
    IF Exist(vr1, vr2) in corresponding VRH
    THEN
        Delete (vr1, vr2) from this VRH
        Preserve RRA if needed
        Preserve URA if needed
        Preserve PRA if needed
    ENDIF
    RETURN true
End
```

图 3.14. 删除角色继承关系算法

3.4.3.5 处理角色互斥

MDRBAC 模型中定义的角色互斥是由虚拟角色维上定义的虚拟角色互斥通过直积得到的。因此我们给出添加虚拟角色互斥 AddVirtualMutualExclusion 和删除虚拟角色互斥 DeleteVirtualMutualExclusion 两个算法的伪码表示。算法简单的添加和删除给定角色维上的角色互斥，然后管理员可以通过指定不同的角色互斥策略（定义 3.7）来生成系统角色集上的角色互斥定义。

```
Algorithm: AddVirtualMutualExclusion
Input: VirtualRole vr1, vr2
Output: Bool
Description: Return true if successfully add a virtual mutual exclusion

Begin
    IF IsVirtualInherit(vr1, vr2) OR
            IsVirtualInherit(vr2, vr1)
    THEN
        RETURN false
    ENDIF
    IF !ExistMutualExclusive(vr1, vr2)
    THEN
        Add (vr1, vr2) into current VRME
    ENDIF
    RETURN true
End
```

```
Algorithm: DeleteVirtualMutualExclusion
Input: VirtualRole vr1, vr2
Output: Bool
Description: Return true if successfully delete a virtual mutual exclusion

Begin
    IF ExistMutualExclusive(vr1, vr2)
    THEN
        Delete (vr1, vr2) from current
        VRME
    ENDIF
    RETURN true
End
```

图 3.15. 处理角色互斥算法

3.5 多维 RBAC 模型在 WebDaemon 系统中的应用

WebDaemon 是一个网络资源访问控制系统[2]。将它架构在用户和源服务器之间，可以有效保护源服务器的资源，实现用户对资源的细粒度访问控制。

WebDaemon 系统各组件的逻辑关系图如图 3.16 所示。WebDaemon 服务器的功能包括对来访的用户进行身份验证以及提供访问控制服务器，来保证资源的合法访问；源服务器就是向外提供服务的 WWW 服务器；源服务器资源管理控制台是源服务器管理员进行服务器的资源管理的工具，包括对用户授权、对资源权限分配等。WebDaemon 管理控制台是用来管理 WebDaemon 的系统数据库，

包括创建角色、删除角色、创建用户、删除用户、定义角色关系和给用户授权角色等等。系统检测和日志分析是对系统的访问进行审计。缓存和过滤模块是专门提供对系统性能优化的。

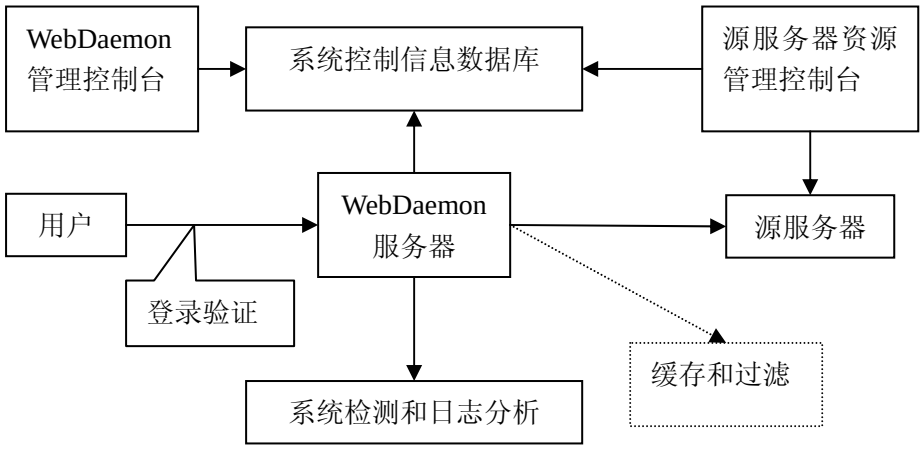
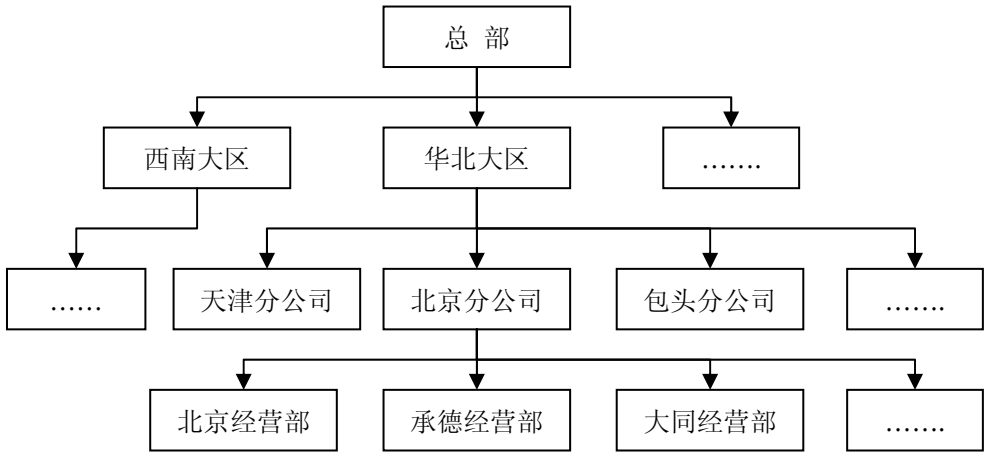


图 3.16. WebDaemon 系统各组件逻辑关系图

WebDaemon 系统目前在某大型公司实际运行，并取得安全可靠和稳定有效的结果[2]。

角色定义和角色管理是整个 WebDaemon 系统的一项核心任务。下面我们结合该公司的角色组织介绍一下角色维数的实现。

公司内联网访问控制的角色集是一个立体结构，可以分成三个独立的角色维数：机构体系，部门管理以及数据查询。图 3.17 描述了这三个角色维数内部的层次关系。



(a)

[2] TCL 集团（http://info.tcl.com.cn）。

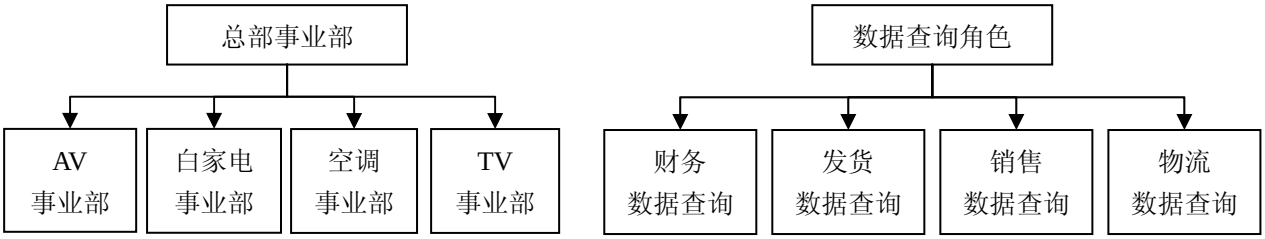


图 3.17. 公司角色的立体结构

(a) 机构体系角色维数; (b) 部门管理角色维数; (c) 数据查询角色维数

在图 3.17 中，为了简洁，我们只列出了各个虚拟角色，并没有给出完整的系统角色名称。实际上，由于各个角色维基本上都是树状结构，我们依靠角色名称就可以管理几乎所有的角色继承关系。最后该公司的角色继承关系表 RRA 只有小于 10 条的记录，使得查找角色继承关系变得异常迅速。整个系统的角色集中一共有 8300 多个角色，通过定义角色维，我们将角色的维护工作简化到了 260 多个，大大简化了角色管理任务。

通过引入角色命名机制，我们极大的提高了用户权限对应关系检索的效率。由于 90%以上的角色继承关系可以直接通过角色命名得到，对数据库的大量递归检索都不需要了，因此整体性能会有很大的提高。在 WebDaemon 系统的具体测试中，我们发现角色命名机制可以带来 57%的性能提高，大大缩短了访问控制所需的时间。

3.6 有关多维 RBAC 模型的讨论

多维 RBAC 模型的主要思想是在角色集中引入维数的概念，将一个复杂的角色集分解成诸多虚拟角色集，并通过对它们的管理简化对整个系统角色集的管理。这一小节我们主要剖析一下引入角色维数的优缺点。

3.6.1 角色维数的优点

在整个角色集上引入角色维数有如下优点：

1．将整个复杂的角色集分割成相对较简单的角色集

考虑角色层次关系图的角色维数为 d，对应于角色维 L_i 的虚拟角色的个数为

m_i，则整个系统一共有角色 $m_1 \times m_2 \times \cdots \times m_d$ 个。如果直接对所有的角色进行管理，我们需要管理非常多的角色。如果只对各个虚拟角色进行管理，我们只需要管理 $m_1 + m_2 + \cdots + m_d$ 个角色即可。这将大大简化角色的管理工作。

2．方便管理员进行角色的相关管理工作

一般来说，每个角色维都带有一定的语义信息，每个角色在一个角色维上的角色分量都代表了该角色的一部分语义信息。我们将各个角色维分开管理，可以避免不同的语义信息产生的复杂的角色继承关系。管理员管理每一个角色维的时候可以专注于一项语义信息，这样既方便管理，又可以减少出错的概率。具体的角色管理工作我们前一小节都作了详细描述，这里不再赘述。

3．易于实现，能够很方便的刻画角色之间的关系

角色命名机制为定义了角色维数的 RBAC 模型提供了简便、高效的实现方式，对于目前国内大多数企业的基于树状结构的角色定义策略，能够很好的提供模型上的支持。对于一般的角色层次也能够简化角色管理。

4．可以实现一定程度的分布式管理

与 ARBAC97 模型不同，MDARBAC 从维数的角度对整个角色集做了一个纵向的切分，并且允许在每个纵向上继续使用 ARBAC97 模型，因此提供了很好的扩展性。各个管理员职责分明，任务明确，实现了真正意义上的分布式管理。

3.6.2 角色维数的局限及解决方案

在提供了巨大的管理易用性的同时，角色维数也给整个系统的管理带来了一定的局限性。我们在这一节剖析一下这个问题，并给出一些解决办法。

1．角色概念定义过死

根据角色维数的定义，如果一个角色层次图的角色维数为 d，那么每一个角色都可以表示成一个 d 维向量。这在许多实际系统中无法严格的实现。同时，限制每一个角色的维数也使得整个系统难于扩展。我们需要一些辅助机制来更好的利用角色维数机制。

针对特定的应用领域，我们需要特定的处理方法。如果一个角色在某个角色维上没有定义，我们可以赋一个该角色维上最低的虚拟角色给这个角色分量，这

样一般都能够解决问题。一般来说，在定义好了角色维数之后，整个系统不会超出这些角色维能够描述的范围。所以系统扩展并不是一个太大的问题。

2．不易实现角色的细量级管理

在上面的讨论中我们可以看出，利用角色维数可以实现粗量级的管理，但是作为一个实际系统来说，角色的细量级管理是必不可少的。因此还需要一定的细量级管理相配合。

这里需要管理员按照由粗到细的管理策略进行管理。首先进行粗量级的设置和维护，然后针对某些特殊的角色进行细量级的管理。这种粗细程度的划分要针对特定的系统考虑。

3．角色维数的实际确定有一定难度

对于一个实际系统，我们应该从系统的实际应用领域来得到它的角色维数。一般来说，角色维数的获取工作并不十分困难。由于一个角色维实际上代表了角色的一个方面的特性，因此我们根据实际需要是可以比较容易的定义出角色维的。比如对于公司内联网管理，角色维数不外乎机构层次，部门关系，人员关系等等几个层面。

第4章 RBAC 模型实现的缓存机制

缓存机制目前已经应用到包括操作系统，数据库，网络的各个层面，引入缓存机制的初衷就是为了充分利用历史访问记录来尽可能的提高今后访问的效率。在 RBAC 模型中，由于需要频繁的检索给定用户对于给定权限的访问许可，引入缓存是实际系统中必须考虑的一种提高效率的方法。本章针对这个问题提出从模型角度定义的缓存机制，并给出具体实现及应用分析。

4.1 引入缓存机制的必要性

在一个实际的访问控制系统中，一般来说用户和权限的数量比较大，角色的数量不大但是却有着十分复杂的继承关系。一个中等规模的访问控制系统中的角色模型可以有深达 8 层的继承关系。在这种情况下，给定了一个用户和一种权限，判断该用户是否具有该权限可能是一个很深的递归查找过程，这是整个权限管理中最消耗资源的部分。在这种反复需要查询的情况下，引入缓存机制可以极大的提高后续查找的效率，从而提升系统的整体性能。

一个这样的实例如图 4.1 所示。图中我们只列出了一部分用户和权限，而给出了一个相对完整的角色集以及角色层次关系。我们假定角色间没有复杂的限制关系，并且只有用户 U1, U2, U3 以及权限 P1, P2, P3 和角色集有指派关系。

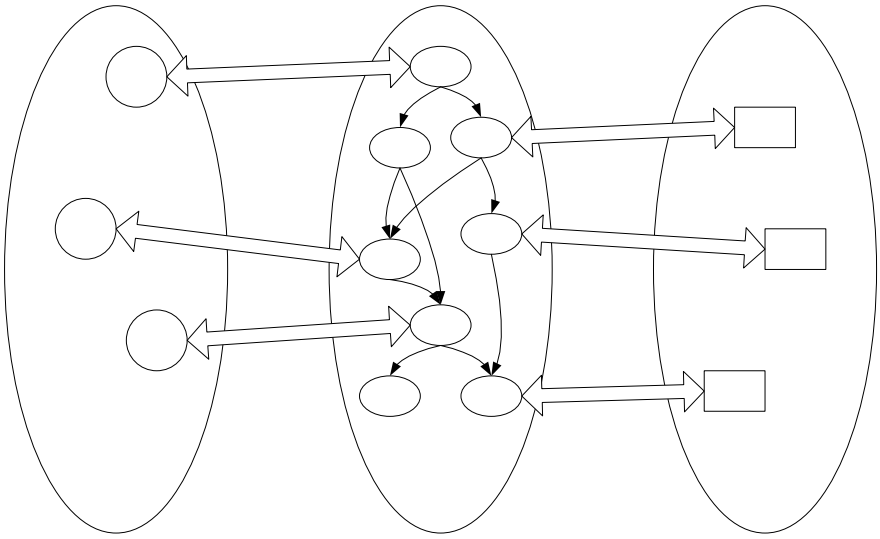


图 4.1. RBAC 模型示例

我们首先来判断用户 U2 是否具有权限 P1。从用户角色指派关系中可以看出，目前用户 U2 只与角色 R4 对应，但是 R4 不直接具备权限 P1。根据角色层次关系，我们需要递归查找 R4 的所有子角色，只要有一个子角色具有这个权限，那么 R4 就具有该权限。很容易看出，R4 的子角色 R8 具有权限 P1，因此 R4 也具有权限 P1，从而 U1 具有权限 P1。

如果系统经常需要查找 U2 对于权限 P1 的拥有情况，那么将这条查询结果预先记录下来则可以在后续查询的时候直接返回给查询终端。这是角色缓存的一个直接应用。

如果今后需要判断用户 U1 是否具有权限 P1，我们需要一个类似的过程。首先查找用户 U1 对应的角色集（R1），然后递归查找 R1 的子角色看其是否具有权限 P1。如果我们不做任何缓存操作，即没有前面查询出来的结果可作参考，那么当我们查到 R1 的一个子角色 R4 的时候，我们还是需要继续查找 R4 的子角色，直到查到 R8 具有权限 P1。如果我们将前面一次查询得到的 R4 拥有权限 P1 记录下来，那么当我们查到 R1 有 R4 这样一个子角色的时候，我们立刻就可以知道 R1 拥有权限 P1，从而 U1 拥有权限 P1。这是角色缓存的一个间接应用。

从这个简单例子可以看出，角色缓存作为 RBAC 模型的一个辅助手段，确实是一个简便且高效的方法。

注意本章考虑的问题不是模型本身的管理，而是在一个已经明确定义的 RBAC 模型上具体实施访问控制。如果将整个访问控制作为一个黑箱，那么输入是一个用户标识 U 和一个权限标识 P，输出是一个布尔值：允许（Y）或拒绝（N）。

4.2 RBAC 模型的缓存机制

缓存机制可以在系统的不同层面定义。我们首先介绍一种不依赖于模型的缓存——RBAC 模型外缓存——作为对比，然后介绍 RBAC 模型内缓存。我们将给出四种不同类型的 RBAC 模型内缓存，并给出如何进行缓存更新。

4.2.1 RBAC 模型外缓存

RBAC 模型外缓存是定义在整个访问控制模型之外的缓存策略，如图 4.2 所示。

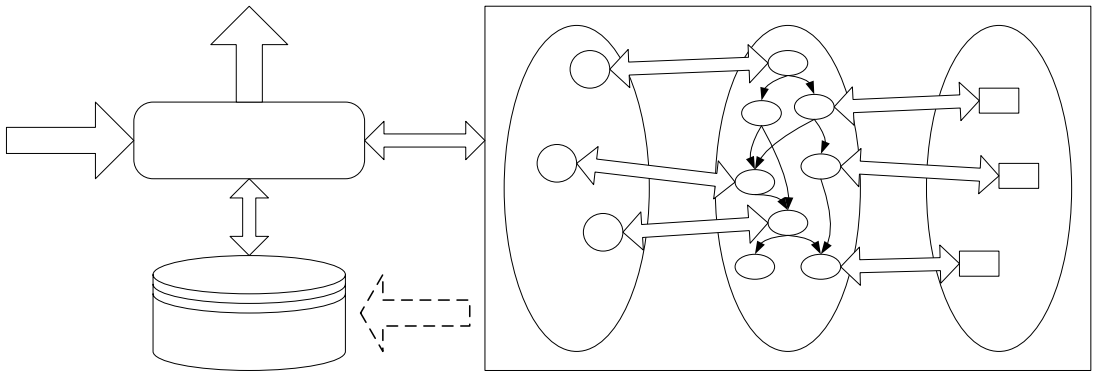


图 4.2. RBAC 模型外缓存结构图

在这种缓存定义策略中，给定用户 U 和权限标识 P，我们首先在外部缓存中寻找是否有缓存数据，如果有就直接返回。否则我们将它们输入到访问控制的黑箱中，得到访问控制结果。如果返回结果是允许（Y），则将用户权限对（U，P）存入外部缓存中。

缓存更新是缓存策略里面一个很重要的方面，目的是为了保持一致性。RBAC 模型外缓存由于独立于访问控制模型本身，因此当模型发生变化时必须手动更新所有的模型外缓存。由于模型本身并不知道这些缓存数据的格式，因此更新的时候只有将所有缓存数据全部清除。

RBAC 模型外缓存的优点是实现简单，与模型无关，可以由应用程序自己来定义缓存格式。但是由于无法由访问控制模型本身来控制，缓存的更新十分困难，难以维持一致性。而且这种模式只能实现一种粗粒度的缓存，无法缓存相关的其它信息，并且在更新时不能只更新模型改变的部分。

4.2.2 RBAC 模型内缓存

由于 RBAC 模型外缓存存在以上诸多问题，我们考虑在 RBAC 模型内部实现一定的缓存机制，灵活定义和处理缓存。在 RBAC 模型内部我们可以缓存很多中间结果，从而实现一种细粒度的缓存机制。

在 RBAC 模型中，我们假定用户、角色、权限的基本信息都存储在相关介质中。用户角色指派、权限角色指派、角色层次关系以及限制等其他关系都以某种形式被存储起来。由于角色层次关系的存在，用户通过角色继承可以间接拥有某些角色，权限通过角色继承也可以间接对应某些角色，角色还可以通过角色继承间接继承其它角色。比如图 4.1 中 U1 间接拥有角色 R2，P3 间接对应角色 R1，

R2 间接继承角色 R7。

由于用户和权限相对较多，角色层次关系也相对较复杂，一般系统中只会存取直接的用户角色赋值、权限角色赋值以及角色层次关系，不会将间接生成的各种赋值关系和层次关系直接存储起来。这样就需要在实际的访问控制中递归的查找这些间接关系。这时如果我们将每次查找得到的中间结果（包括间接的用户赋值，权限赋值，角色层次）以缓存的方式储存起来，就可以提高后续查找的效率，同时不至于缓存全部的间接赋值与继承关系，提高系统的可靠性和可用性。在模型发生变化时，只需要针对改变的部分更新相关的缓存，尽可能的保存历史资源。RBAC 模型内缓存的基本框架如图 4.3 所示。

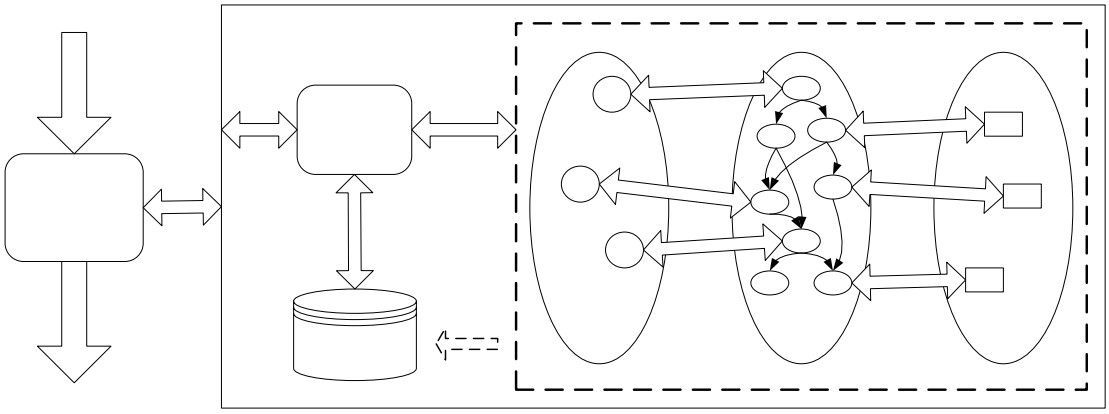


图 4.3. RBAC 模型内缓存结构图

在 RBAC 模型中考虑缓存策略，可以缓存的信息包括以下四点：用户权限直接对应关系，间接用户角色指派，间接权限角色指派，间接角色继承关系。我们可以在系统中实现一种缓存策略，也可以综合考虑多种缓存策略。

4.2.2.1 缓存用户权限对应关系

这种缓存策略缓存的内容和 RBAC 模型外缓存相同，唯一的不同点就是在模型内部控制缓存的内容以及进行更新。模型内缓存控制器负责将新查询到的用户权限对应关系添加到 RBAC 模型内缓存中，并在模型发生变化的时候自动更新缓存的内容。模型内缓存控制器可以细化缓存更新的粒度，尽可能的保留有用的信息。

对 RBAC 模型的如下修改将导致缓存更新：

1．删除用户 DeleteUser(u)

更新策略：将缓存中包含用户 u 的记录删除。

2．删除权限 DeletePermission(p)

更新策略：将缓存中包含权限 p 的记录删除。

3．删除用户角色指派 DeleteRelationUserRole(u, r)

更新策略：由于用户 u 与角色 r 不再关联，缓存中包含 u 的记录可能失效（利用了 u 和 r 的对应关系），也可能不失效（没有利用该关系）。因此如果要进行细粒度的维护，必须实时的检测哪些缓存记录用到了这个信息。当然，另一种简单的策略就是删除所有与该用户关联的缓存纪录。

4．删除角色权限指派 DeleteRelationRolePermission(r, p)

更新策略：和前面删除用户角色指派类似。有两种选择：检测并删除与该指派相关的缓存纪录；或者删除与权限 p 相关的所有缓存纪录。

5．删除角色 DeleteRole(r)

更新策略：由于用户和权限一定是通过角色产生关联的，删除一个角色很可能会破坏大量的用户权限对应关系。虽然可以实时检测每一条缓存纪录看是否与该角色有关，直接彻底删除所有的缓存记录可能是一种更简单有效的方法。

6．删除角色层次关系 DeleteInheritance(r1, r2)

更新策略：删除角色层次关系和删除角色一样，会大量影响到用户权限对应关系。因此直接彻底删除所有缓存纪录是一种简单有效的方法。

4.2.2.2 缓存用户角色指派关系

这种缓存完全是在模型内部的一种细化的缓存策略，添加缓存的时机是在查询用户权限对应关系的时候。下面我们用伪码的方式描述如何查询用户权限对应关系。

```
Algorithm: QueryRelationUserPermission
Input: User u, Permission p
Output: Bool
Description: Return true if user u has permission p and false else
Begin
    RoleVector rv = user-role relation in UA including cached relations          (2)
    FOR EACH r IN rv DO
        IF (r, p) in PA THEN
```

```
            RETURN true
        ENDIF
    ENDFOR
    FOR EACH r IN rv DO
        RoleVector childrv = all child roles of r
        FOR EACH childrole IN childrv DO
            Add (u, r) into cache                    //缓存用户角色指派关系          (1)
            IF (childrole, p) in PA THEN
                RETURN true
            ENDIF
        ENDFOR
    ENDFOR
    RETURN false
End
```

图 4.4. 查询时缓存用户角色对应关系

算法直截了当，其中在(1)处增加了用户角色指派关系的缓存，并且后续的查找在(2)处利用到了这些缓存。可以发现，查询一条用户权限对应关系可能添加多条缓存记录，因此缓存起的作用非常大。

设想在图 4.1 中首先查询了用户 U1 和权限 P3 的关系，通过角色层次的递归，发现用户 U1 除了是角色 R1 的用户，还是角色 R3 和 R5 的用户。那么下次如果需要查询用户 U1 和权限 P2 的关系，在上面算法中第一个 FOR 循环就可以发现缓存的角色 R3 就拥有权限 P2，从而不需要进入第二个循环就可以迅速判断结果。

如果系统随机的查询用户权限对应关系，那么经过长时间的缓存，每个用户对应的全部角色都会缓存下来，系统效率达到顶点。如果事先限制了缓存记录条数，那么还需要一种动态的方法更新缓存数据。

对 RBAC 模型的如下修改将导致缓存更新：

1．删除用户 DeleteUser(u)

更新策略：将缓存中包含用户 u 的记录删除。

2．删除用户角色指派 DeleteRelationUserRole(u, r)

更新策略：由于用户 u 与角色 r 不再关联，缓存中包含 u 的记录可能失效（利用了 u 和 r 的对应关系），也可能不失效（没有利用该关系）。因此如果要进行细粒度的维护，必须实时的检测哪些缓存记录用到了这个信息。当然，另一种简单的策略就是删除所有与该用户关联的缓存纪录。我们后面会讲到这

种缓存策略的一种扩展提高这种更新的性能。

3．删除角色 DeleteRole(r)

更新策略：由于用户角色关系的缓存是通过中间角色产生的，删除一个角色很可能会破坏大量的用户角色关系。虽然可以实时检测每一条缓存纪录看是否与该角色有关，直接彻底删除所有的缓存记录可能是一种更简单有效的方法。

4．删除角色层次关系 DeleteInheritance(r1, r2)

更新策略：删除角色层次关系和删除角色一样，会大量影响到用户角色关系。因此直接彻底删除所有缓存纪录是一种简单有效的方法。

4.2.2.3 缓存权限角色指派关系

这种缓存也是模型内部的一种细化的缓存策略，添加缓存的时机是在查询用户权限对应关系的时候。我们同样用伪码的方式描述如何查询用户权限对应关系。

```
Algorithm: QueryRelationUserPermission
Input: User u, Permission p
Output: Bool
Description: Return true if user u has permission p and false else
Begin
    RoleVector rv = user-role relation in UA
    FOR EACH r IN rv DO
        IF (r, p) in PA including cached relations THEN                    (2)
            RETURN true
        ENDIF
    ENDFOR
    FOR EACH r IN rv DO
        RoleVector childrv = all child roles of r
        FOR EACH childrole IN childrv DO
            IF (childrole, p) in PA including cached relations THEN        (3)
                RoleVector rolepath = Get role inheritance path from r to childrole
                FOR EACH inpathrole IN rolepath DO
                    ADD (inpathrole, p) into cache     //缓存权限角色指派关系 (1)
                ENDFOR
                RETURN true
            ENDIF
        ENDFOR
    ENDFOR
```

```
    RETURN false
End
```

图 4.5. 查询时缓存权限角色对应关系

算法和前面的类似，其中在(1)处增加了权限角色指派关系的缓存，并且后续的查找在(2)和(3)处利用到了这些缓存。为了尽可能多的利用这一次查找信息增加缓存的量，我们记录下了从用户直接指派的角色到最后具体拥有该权限的角色的一条继承路径，并将路径上的所有角色都添加对于该权限的缓存记录。

设想在图 4.1 中首先查询了用户 U1 和权限 P1 的关系，通过角色层次的递归，我们添加了四条缓存记录：（R1, P1），（R2, P1），（R4, P1），（R6, P1）。这样当下次查询 U2 对于权限 P1 的访问许可时，就可以直接从 U2 的对应角色 R4 上发现它对于权限 P1 的访问许可，减少一次 FOR 循环的开销。

与缓存用户角色对应关系类似，如果系统随机的查询用户权限对应关系，那么经过长时间的缓存，每个权限对应的全部角色都会缓存下来，系统效率达到顶点。

对 RBAC 模型的如下修改将导致缓存更新：

1．删除权限 DeletePermission(p)

更新策略：将缓存中包含权限 p 的记录删除。

2．删除角色权限指派 DeleteRelationRolePermission(r, p)

更新策略：由于权限 p 与角色 r 不再关联，缓存中包含 p 的记录可能失效（利用了 p 和 r 的对应关系），也可能不失效（没有利用该关系）。因此如果要进行细粒度的维护，必须实时的检测哪些缓存记录用到了这个信息。当然，另一种简单的策略就是删除所有与该权限关联的缓存纪录。

3．删除角色 DeleteRole(r)

更新策略：由于权限角色关系的缓存是通过中间角色产生的，删除一个角色很可能会破坏大量的权限角色关系。虽然可以实时检测每一条缓存纪录看是否与该角色有关，直接彻底删除所有的缓存记录可能是一种更简单有效的方法。

4．删除角色层次关系 DeleteInheritance(r1, r2)

更新策略：删除角色层次关系和删除角色一样，会大量影响到权限角色关系。因此直接彻底删除所有缓存纪录是一种简单有效的方法。

4.2.2.4 缓存角色继承关系

这种缓存实际上是把角色层次所蕴涵的间接角色继承关系储存起来，那么下次在对角色关系表进行查找的时候就可以直接得到当前角色的大部分子女。

实际上，在前面两种缓存策略中，已经蕴涵的缓存了角色继承关系，但是以一种与用户或者权限挂钩的方式实现的。直接缓存角色继承关系，只能在查询用户权限关系的算法中第二个 FOR 循环里面得到当前角色的所有子女的地方起作用。比起前面两种缓存策略，缓存角色继承关系的作用相对较小。

对 RBAC 模型的如下修改将导致缓存更新：

1．删除角色 DeleteRole(r)

更新策略：缓存的角色继承关系非常依赖于角色以及角色层次关系的存在，因此删除一个角色很可能会破坏大量缓存的角色继承关系。虽然可以实时检测每一条缓存纪录看是否与该角色有关，直接彻底删除所有的缓存记录可能是一种更简单有效的方法。

2．删除角色层次关系 DeleteInheritance(r1, r2)

更新策略：删除角色层次关系和删除角色一样，会大量影响到角色继承关系。因此直接彻底删除所有缓存纪录是一种简单有效的方法。

4.2.3 RBAC 模型缓存机制的相关讨论

衡量一种缓存机制的优劣需要考虑诸多方面的因素，包括缓存必要性，提升效率程度，实现难度，缓存操作本身的成本，缓存更新成本，缓存数据需要的容量等方面。综合考虑前面五种 RBAC 模型的缓存策略，我们给出表 4.1 的比较。

	必要性	效率提升度	实现难度	操作成本	更新成本	缓存容量
模型外缓存	中	低	低	低	低，手动	高
缓存用户权限关系	中	中	低	低	高	高
缓存用户角色指派	高	高	中	中	高	中
缓存权限角色指派	高	高	中	高	高	中
缓存角色继承关系	中	中	中	中	中	低

表 4.1. 缓存策略比较表

模型外缓存的必要性一般，效率提升度低，但是实现难度低，缓存操作本身成本比较低，容易实现。由于这种缓存策略独立于 RBAC 模型本身，虽然更新成本低，但是需要手动更新，管理起来不方便。由于只能缓存用户权限对应关系，需要比较高的缓存容量才能提供充分的历史数据。

在 RBAC 模型内部实现用户权限关系缓存，与模型外缓存基本具有一致的属性。但是由于可以细化缓存的更新，更多的缓存数据得到了利用，效率提升度升高，但同时更新成本也比较高。

用户角色指派缓存和权限角色指派缓存基本具有一致的属性。其缓存的必要性比较高，也能极大的提升效率。相对前面两种缓存策略，它们需要在查询中嵌入缓存操作，因此实现难度略高。由于细化了缓存的类型，更新成本较高，但缓存的容量一般不大。它们唯一的区别就是缓存操作本身的成本。权限角色缓存需要找寻一条角色继承的路径，因此成本比用户角色缓存高。

缓存角色继承关系的所有指标基本上都位于中游，由于仅缓存角色本身的信息，需要的缓存容量相对较低。

综上所述，缓存用户角色指派或者权限角色指派是一种比较合适的方法，而且二者并不矛盾。实际系统中可以同时缓存二者，这样虽然进一步提高了缓存操作成本和更新成本，但是系统的效率会得到进一步提高，符合实际访问控制系统的需要。直接缓存用户权限关系是前面两种缓存的一种补充，实际中也可以考虑，可根据实际需要考虑在模型内部实现或者由外部应用程序来实现。

4.3 RBAC 模型内缓存的实现

上一节中我们已经给出了几种缓存算法的伪代码以及更新操作的描述，这一节我们简要叙述一下从底层的存储介质上考虑的 RBAC 模型内缓存的实现。由于我们要在 RBAC 模型内部考虑缓存机制的实现，最直接的方法就是将缓存数据直接写入底层的存储介质上去。考虑到 RBAC 模型的主要应用是在查询方面，我们除了提供一种数据库的实现之外，还讨论一种基于 LDAP（轻量级目录访问协议）的实现策略。

4.3.1 在数据库上实现 RBAC 模型内缓存

对于四种 RBAC 模型内缓存，我们分别从数据库底层定义缓存的底层结构如下。

由于底层数据库没有一个表单专门用来存储用户权限对应关系，要缓存这些关系，必须定义一个新的数据表结构，如表 4.2 所示。其中第四个字段为缓存建立的时间，可有可无。

字段名称	字段描述	字段类型
f_userid	用户 ID	varchar
f_permissionid	权限 ID	varchar
f_ispermitted	是否允许（Y/N）	char
f_createtime	缓存建立的时间	datetime

表 4.2. 存储用户权限对应关系

要缓存用户角色指派，只需要在用户角色指派表中添加一个字段，表示是否是缓存的数据，便于更新。

字段名称	字段描述	字段类型
f_userid	用户 ID	varchar
f_roleid	角色 ID	varchar
…	…	…
f_iscached	是否是缓存数据（Y/N）	char

表 4.3. 存储用户角色指派

为了更细致的控制用户角色指派关系的缓存，我们还可以另建一个新表，在其中增加一个字段表示具体和用户发生指派关系的角色，这样可以在用户角色指派关系发生变化的时候只更新受影响的缓存数据。不过在利用缓存数据的时候就不是很方便了，因为需要到两个表里面读取数据。

字段名称	字段描述	字段类型
f_userid	用户 ID	varchar
f_roleid	角色 ID	varchar

f_directroleid	实际与 f_userid 发生指派的角色	varchar

表 4.4. 存储用户角色指派

对于缓存权限角色指派关系以及缓存角色继承关系的 RBAC 模型内缓存机制，我们有类似的数据表结构，这里就不再赘述了。

在 WebDaemon 系统的实际测试中，我们应用了一个小时的访问日志来测试各种缓存策略的性能。缓存用户角色指派和缓存权限角色指派得到了类似的性能提升，在 25%~30%之间；缓存角色继承关系得到了大概 20%的提升，而缓存用户权限对应关系大概只有 10%的提升。这些测试数据与我们前面的比较相符。

用数据库来存储 RBAC 模型的后台数据是一种常用的方法，但是由于访问控制中大部分是查询操作，数据库的效率并不是很高。用下面提到的 LDAP 一般会得到比较好的性能。

4.3.2 在 LDAP 上实现 RBAC 模型内缓存

LDAP 提供了一种树状的存储结构，并且专门对查询做了大量的优化，因此比较适用于 RBAC 模型这种应用[7]。RBAC 模型内缓存的实现依赖于 RBAC 模型本身的 LDAP 实现。我们假定 RBAC 模型本身的存储结构如图 4.6 所示。

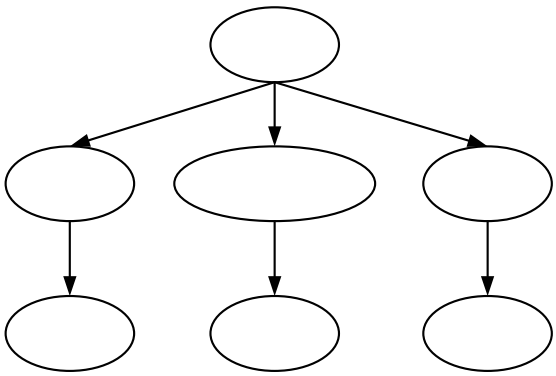


图 4.6. RBAC 模型的 LDAP 存储结构

图中 Root 是根节点，也就是整个 LDAP 的树根。User, Role, Permission 是一些定义好的属性节点，每个节点包含自己相关的属性，类似于面向对象里面的类的概念。在 Root 下的第一层次上，User 节点代表所有的用户，每添加一个用户就会在 User 的子树下面添加一个新的节点；Permission 节点代表所有的权限，每添加一个权限就会在 Permission 的子树下面添加一个新的节点；Role 节点代表所

有的角色，每添加一个角色就会在 Role 的子树下面添加一个新的节点。然后在树的第二层，User 下面的 Role 节点代表每个 user 的用户角色指派，添加一个指派只需要在相应的用户节点下面添加一个角色节点。同样，Permission 下面的 Role 节点代表每个 permission 的权限角色指派。Role 下面的 Role 节点代表每个 role 的直接子角色，也就是角色层次中的直接继承关系。这样整个 RBAC 模型就可以完全用这棵树来表示了。

作为一个示例，图 4.1 中的 RBAC 模型可以用如下的 LDAP 树来表示。

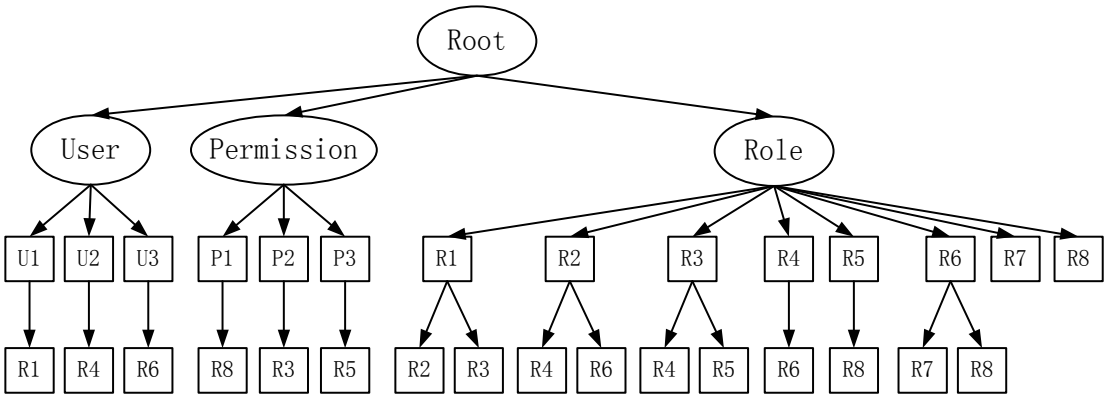


图 4.7. LDAP 存储示例

对于直接缓存用户权限对应关系的缓存模型，需要另建一个新的第一层节点用于缓存，或者直接将缓存的 Permission 放在原来模型的 User 节点下面，如图 4.8 所示。这样方便在下一次检索的时候查询。

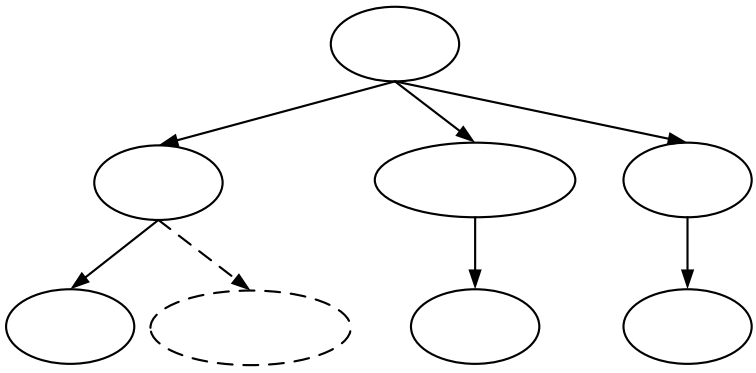


图 4.8. 缓存用户权限对应关系

对于缓存用户角色指派的缓存模型，可以直接将缓存的用户角色指派也放在 User 节点下面，同时添加一个属性标明该指派是否是缓存结果，如图 4.9 所示。

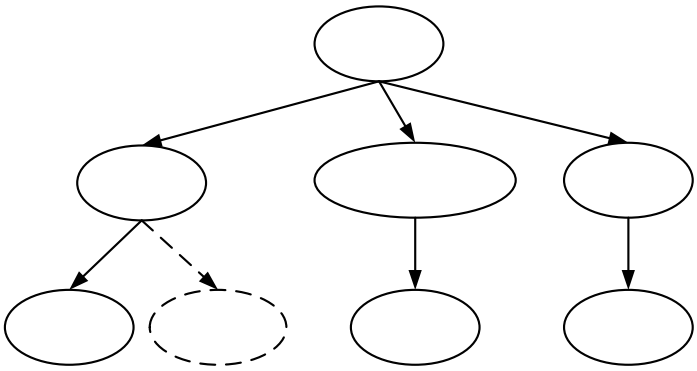


图 4.9. 缓存用户角色指派

对于缓存权限角色指派的缓存模型，可以直接将缓存的权限角色指派也放在 Permission 节点下面，同时添加一个属性标明该指派是否是缓存结果，如图 4.10 所示。

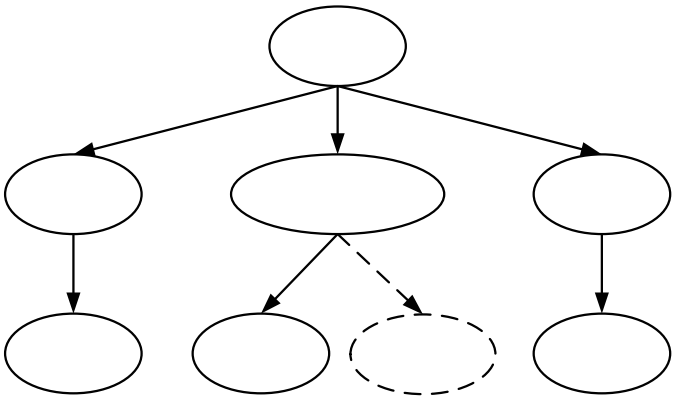


图 4.10. 缓存权限角色指派

对于缓存角色继承关系的缓存模型，可以直接将缓存的继承关系放在 Role 节点下面，同时添加一个属性标明该指派是否是缓存结果，如图 4.11 所示。

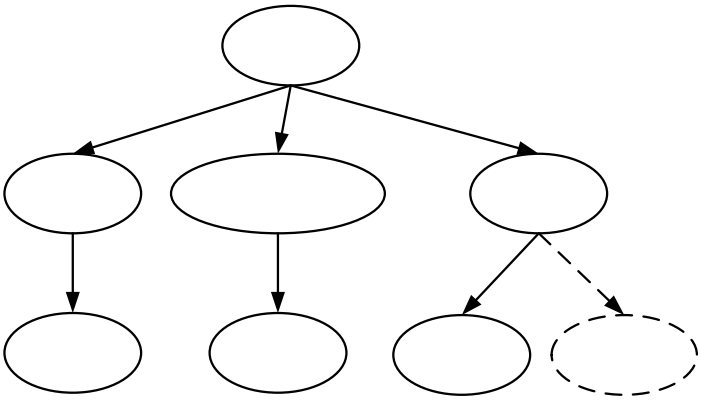


图 4.11. 缓存角色继承关系

将缓存的记录直接放在相关节点下面，可以有效提高缓存的利用率，并且在

缓存的利用上提供了一种一致的方法。由于 LDAP 更符合面向对象的一种存储组织形式，查询和管理起来更方便，缓存也实现得很直观。

采用与数据库同样的测试数据，我们在 LDAP 上得到了类似的结果，但是总体提升的比例没有数据库那么大。缓存用户角色指派和缓存权限角色指派得到了 20%左右的性能提升，缓存角色继承关系得到了大概 15%的提升，而缓存用户权限对应关系大概只有 5%~10%的提升。这个测试结果是由于利用 LDAP 本身已经能够比较高的提高性能了，增加缓存对整体性能提高并不是很大。

第5章 RBAC 模型处理中间件

传统的访问控制技术目前还占据着绝大部分的市场，虽然 RBAC 访问控制模型从各方面来说都优于传统模型，但是由于没有一种比较方便的配置工具，使得更改模型的成本较高，实施困难。这一章提出了 RBAC 访问控制中间件（RBAC Middleware）的概念，将整个 RBAC 访问控制模型封装在一个模块中，并尽可能的提供对外接口，提高中间件的可重用性和扩展性。

5.1 RBAC 中间件的目标

中间件（Middleware）现在已经不是一个新名词了，它已经被广泛应用到了计算机的各个领域。顾名思义，中间件就是架构在底层数据与应用平台之间的模块部分，封装特定功能并对外提供接口，提供足够的灵活性和扩展性。外部应用程序可以根据中间件提供的接口方便的进行二次开发，实现模块的重用。

RBAC 模型对于一个访问控制系统来说是一个相对独立的模块，并且具有自己的一套管理方法。将 RBAC 模型部分作为一个中间件实现有利于访问系统的模块化，并且可以尽量减少管理的成本。

作为实现 RBAC 模型的中间件，必须将与 RBAC 模型有关的部分完全封装起来，对外提供与一般访问控制类似的接口，管理员经过一定的配置就可以应用 RBAC 的高效功能。具体来说，中间件必须能够处理两种形式的访问：访问控制查询端以及 RBAC 模型管理控制端。RBAC 模型中间件的示意图如图 5.1 所示。

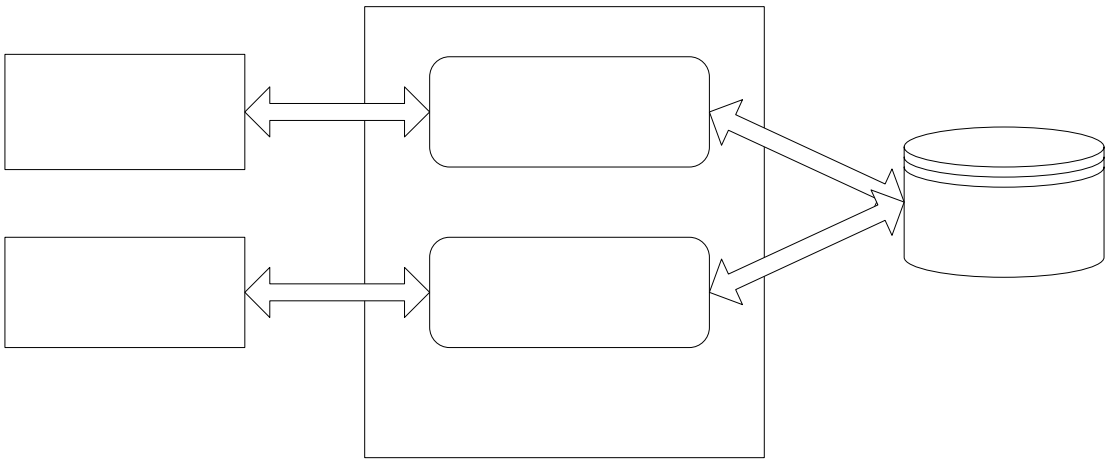


图 5.1. RBAC 模型处理中间件示意图

首先中间件提供一个 RBAC 模型的管理接口，外部的模型管理控制应用程序可以通过该接口进行 RBAC 模型的模型定义以及数据存取，构建整个 RBAC 应用模型。然后查询访问控制应用程序通过访问控制查询接口检索后端存储介质，返回访问控制结果。整个模型处理中间件必须能够提供足够灵活的 API，并且能够处理不同的后端存储介质。系统 RBAC 应用模型的建立、管理都在管理接口上实现，不应该通过其他的方式实现。

5.2 RBAC 中间件的设计

RBAC 模型中包含几个基本元素：用户，角色，权限，角色层次，限制。我们在设计中间件的时候都必须考虑。为了通用性，我们没有对该角色模型做任何约束，只针对最一般的 RBAC96 模型来实现，这样以后可以灵活添加其他组件。由于限制是一个比较复杂的概念，我们目前在中间件里仅考虑了角色基数的限制，没有考虑角色互斥。如果实际系统有比较清晰的角色互斥定义，我们可以很方便的添加进该中间件中。

RBAC 模型中间件必须对底层数据存储给出统一接口，也必须对上层应用程序提供接口，我们首先介绍中间件的类设计，然后分两部分来讲述中间件的接口设计。

5.2.1 中间件类设计

中间件中包含如下三个类：用户类 User，角色类 Role，权限类 Permission。对每个类，我们只定义最基本的关键字字段，给具体应用程序提供了最大的扩充空间。应用程序可以扩充这些基本类，比如可以将 Permission 类扩充为两个类 Object（对象）和 Operation（操作）。

整个中间件的实现只关心 RBAC 模型的部分，因此将用户、角色和权限都看作一个抽象的实体。具体定义这些实体的任务留给了应用程序。

5.2.2 中间件底层接口设计

由于 RBAC 模型的底层数据模型可以是数据库，也可以是 LDAP 或者其他

专门的数据存储结构，因此我们必须提供一套统一的底层接口适用于不同的存储模型。

RBAC 模型对底层数据的操作比较简单，因此我们可以定义一套统一的原子操作来进行数据的实际添加、删除与查询。对于不同的底层存储模型，我们可以分别实现不同的原子操作，但是保持每个原子操作的基本语义；上层的操作视图则可以调用统一的原子操作来进行实际操作，因此在这些原子操作之上，底层模型是统一的。

根据原子操作的不同功能，我们有如下五种类型的原子操作。我们定义的原子操作函数名都以“P_”开头，并且大部分的原子操作返回布尔值，表示当前操作是否成功。

5.2.2.1 基本添加删除操作

这些操作提供了最基本的添加和删除的操作，操作的对象包括用户，角色，权限，各种赋权关系，角色层次关系等。主要的原子操作定义如下：

- 添加用户 P_AddUser(user)
- 删除用户 P_DeleteUser(user)
- 添加权限 P_AddPermission(perm)
- 删除权限 P_DeletePermission(perm)
- 添加角色 P_AddRole(role)
- 删除角色 P_DeleteRole(role)
- 添加角色继承关系 P_AddInheritance(role1, role2)
- 删除角色继承关系 P_DeleteInheritance(role1, role2)
- 添加用户角色对应关系 P_AddRelationUserRole(user, role)
- 删除用户角色对应关系 P_DeleteRelationUserRole(user, role)
- 添加角色权限对应关系 P_AddRelationRolePermission(role, perm)
- 删除角色权限对应关系 P_DeleteRelationRolePermission(role, perm)

对于数据库存储，每条原子操作实际上就是一条 SQL 语句的执行，对于 LDAP 存储，原子操作就对应着 LDAP 树中的一个节点的添加或者删除。

5.2.2.2 批量删除操作

这些操作实现批量删除，虽然也可以通过前面定义的删除语句一条一条的删除，但是定义批量删除操作可能会显著的提高性能。

- 指定用户删除用户角色指派 P_BatchDeleteRelationUserRoleByUser(user)
- 指定角色删除用户角色指派 P_BatchDeleteRelationUserRoleByRole(role)
- 指定权限删除角色权限指派 P_BatchDeleteRelationRolePermissionByPermission(perm)
- 指定角色删除角色权限指派 P_BatchDeleteRelationRolePermissionByRole(role)
- 批量删除指定角色的直接子角色 P_BatchDeleteInheritanceByParent(role)
- 批量删除指定角色的直接父角色 P_BatchDeleteInheritanceByChild(role)

在数据库中实现这些操作可能也就是一条 SQL 语句，在 LDAP 中可能会删除整棵子树。

5.2.2.3 检验存在性

存在性检验原子操作判断给定的信息是否在当前的 RBAC 模型中存在，返回布尔值。

- 检验用户是否存在 P_ExistUser(user)
- 检验角色是否存在 P_ExistRole(role)
- 检验权限是否存在 P_ExistPermission(perm)
- 检验角色继承关系是否存在 P_ExistInheritance(prole, crole)
- 检验用户角色关系是否存在 P_ExistRelationUserRole(user, role)
- 检验角色权限关系是否存在 P_ExistRelationRolePermission(role, perm)
- 指定用户检验用户角色关系是否存在 P_HasRoleByUser(user)
- 指定角色检验用户角色关系是否存在 P_HasUserByRole(role)
- 指定权限检验角色权限关系是否存在 P_HasRoleByPermission(perm)
- 指定角色检验角色权限关系是否存在 P_HasPermissionByRole(role)
- 检验指定角色是否有直接子角色 P_HasChildRole(role)

- 检验指定角色是否有直接父角色 P_HasParentRole(role)

这些原子操作在数据库上和 LDAP 上都比较容易实现。

5.2.2.4 查询操作

查询操作返回查询结果（一个向量），仅查找在模型中明确定义的指派和角色层次关系。

- 指定用户查询对应角色 P_QueryRoleByUser(user)
- 指定角色查询对应用户 P_QueryUserByRole(role)
- 指定权限查询对应角色 P_QueryRoleByPermission(perm)
- 指定角色查询对应权限 P_QueryPermissionByRole(role)
- 查询指定角色的子角色 P_QueryChildRole(role)
- 查询指定角色的父角色 P_QueryParentRole(role)

查询操作在数据库中通过 SELECT 子句实现，在 LDAP 中通过检索子树实现。

5.2.2.5 事务有关操作

这些操作定义事务的开始，提交和回滚，目的是保持操作的完整性。

- 事务开始 P_TransactionBegin()
- 事务回滚 P_TransactionRollback()
- 事务提交 P_TransactionCommit()

如果需要实现其他扩展的 RBAC 特性，比如实现多维 RBAC 模型或者添加模型内缓存，还需要添加一些原子操作。我们后面的 XX 节会讨论这个问题。

5.2.3 中间件应用层接口设计

在这些原子操作之上，可以定义各种应用层接口，其中包括很多类似原子操作但是功能更完全的用户视图。比如添加用户的应用层操作（Java 实现）如图 5.2 所示。

```
public boolean AddUser(User user) throws RBACException
{
```

```
        P_TransactionBegin();
        if (P_ExistUser(user))
            throw new RBACException("This user currently exists!");
        if (P_AddUser(user))
        {
            P_TransactionCommit();
            return true;
        }
        else
        {
            P_TransactionRollback();
            return false;
        }
    }
```

图 5.2. AddUser 接口实现

我们下面列出主要的应用层接口，外部应用程序可以利用这些接口实现对 RBAC 模型的完全管理。我们将省略伪码实现。

5.2.3.1 添加操作

大部分的添加操作与图 5.2 中的算法类似，除了添加角色继承关系。由于角色继承关系不能出现环路情形，添加一个角色继承关系必须检查是否构成环路，这是通过调用 CheckInheritance 函数实现的，我们下面会介绍。为了方便在管理员知道不会构成环路的时候不做这种检查，我们在第一个添加角色继承关系函数中给了一个布尔变量作为输入。第二个函数调用第一个函数，并将参数 bNeedCheck 设置为 true。

- 添加用户 AddUser(user)
- 添加权限 AddPermission(perm)
- 添加角色 AddRole(role)
- 添加角色继承关系 1 AddInheritance(role1, role2, bNeedCheck)
- 添加角色继承关系 2 AddInheritance(role1, role2)
- 添加用户角色对应关系 AddRelationUserRole(user, role)
- 添加角色权限对应关系 AddRelationRolePermission(role, perm)

5.2.3.2 删除操作

删除操作比较复杂一些，因为被删除的对象可能有其他相关的纪录，并且可能需要考虑删除后恢复原有的继承关系和角色继承关系。我们具体介绍如下：

- 删除用户 DeleteUser(user)

首先调用原子操作删除所有的用户指派，然后删除用户本身。

- 删除权限 DeletePermission(perm)

首先调用原子操作删除所有的权限指派，然后删除权限本身。

- 删除角色 DeleteRole(role)

由于删除角色将导致许多指派关系发生变化，并且会影响到角色继承关系，我们这里提出如下几种考虑。这些都需要管理员根据实际需要定义具体的删除操作。

如果当前有用户与该角色有指派关系，那么在删除该角色以后，需要考虑是否需要添加这个用户到这个角色的子角色的指派。

如果当前有权限与该角色有指派关系，那么在删除该角色以后，需要考虑是否需要添加这个权限到这个角色的子角色的指派。

如果存在包含该角色的角色继承关系，那么在删除角色以后，需要考虑是否添加这个角色的父角色到其子角色的角色继承关系。

- 删除角色继承关系 DeleteInheritance(role1, role2)

删除角色继承关系和删除角色类似，也会影响到大量的指派和角色继承关系。我们说明如下。

如果当前有用户与 role1 有指派关系，那么在删除该角色继承关系以后，需要考虑是否需要添加这个用户到 role2 的指派。

如果当前有权限与 role2 有指派关系，那么在删除该角色继承关系以后，需要考虑是否需要添加这个权限到 role1 的指派。

如果 role1 有父角色，那么在删除该角色继承关系以后，需要考虑是否需要添加这个角色到 role2 的角色继承关系。

如果 role2 有子角色，那么在删除该角色继承关系以后，需要考虑是否需要添加 role1 到这个角色的角色继承关系。

- 删除用户角色对应关系 DeleteRelationUserRole(user, role)

- 删除角色权限对应关系 DeleteRelationRolePermission(role, perm)

这两条删除关系直接删除相关记录即可。

5.2.3.3 查询操作

应用层面的查询操作不仅包括查询直接的赋权和角色继承关系，还包括批量查询，就是查询所有的赋权以及继承关系，这里面需要考虑角色继承关系对赋权和角色继承的影响。

- 查询用户对应的直接角色 QueryDirectRoleByUser(user)
- 查询用户对应的所有角色 QueryAllRoleByUser(user)

首先查询用户对应的直接角色，然后返回这些角色以及他们的所有子角色。

- 查询权限对应的直接角色 QueryDirectRoleByPermission(perm)
- 查询权限对应的所有角色 QueryAllRoleByPermission(perm)

首先查询权限对应的直接角色，然后返回这些角色以及他们的所有子角色。

- 查询一个角色的直接子角色 QueryDirectChildRole(role)
- 查询一个角色的所有子角色 QueryAllChildRole(role)

递归调用上面的函数和该函数自身。

- 查询一个角色的直接父角色 QueryDirectParentRole(role)
- 查询一个角色的所有父角色 QueryAllParentRole(role)

递归调用上面的函数和该函数自身。

- 查询角色对应的直接用户 QueryDirectUserByRole(role)
- 查询角色对应的所有用户 QueryAllUserByRole(role)

首先查询角色对应的直接用户，然后返回这些用户以及该角色所有父角色的用户。

- 查询角色对应的直接权限 QueryDirectPermissionByRole(role)
- 查询角色对应的所有权限 QueryAllPermissionByRole(role)

首先查询角色对应的直接权限，然后返回这些权限以及该角色所有子角色的权限。

5.2.3.4 检查操作

这些操作是 RBAC 应用的核心，包括检查一个用户是否拥有某项权限。我们说明如下。

- 检查某用户是否有某角色 CheckRelationUserRole(user, role)

得到该用户所对应的所有角色，然后检查 role 是否在这个集合中。

- 检查某角色是否有某权限 CheckRelationRolePermission(role, perm)

得到该权限所对应的所有角色，然后检查 role 是否在这个集合中。

- 检查某角色是否是另一角色的父角色 CheckInheritance(prole, crole)

首先检查两个角色是否具有直接的继承关系，如果没有，取得 prole 的直接子角色，判断 crole 是否在其中。重复这个过程直到到达角色层次树的底部。前面的添加角色继承关系的函数需要调用这个函数判断角色层次图中是否有环路。

- 检查某用户是否具有某权限 CheckUserPermission(user, perm)

这是整个访问控制系统最关键的一个函数了。外部查询访问控制的应用程序就是调用这个函数完成判断任务的。有很多种实现方法，比如可以得到用户的所有对应角色，然后检查每个角色与 perm 的对应关系。也可以反过来，检查 user 与每个与 perm 关联的角色的关系。当然也可以同时得到两个关联角色集，然后判断与 user 关联的角色集是否包含与 perm 关联的角色集。具体如何实现可以根据实际需要选择。

5.3 RBAC 中间件的扩展

在前面给出的 RBAC 中间件的基础上，可以考虑多种扩展。针对我们本文前两章讨论的多维 RBAC 模型和 RBAC 模型内缓存机制，我们在中间件的层面上给出它们的具体实现。

5.3.1 在中间件中实现多维 RBAC 模型

多维 RBAC 模型从本质上来说是扩展了角色的语义，利用虚拟角色的直和得到一个完整的角色。因此，在中间件的实现上我们可以扩展角色类 Role，定义虚拟角色类 VirtualRole，并且规定一个角色 role 是多个虚拟角色 virtualrole 的

组合。对整个角色集的添加、删除和查询的算法也需要做相应的修改，我们在第三章已经做了详细说明，这里不再赘述。

多维 RBAC 模型中的角色命名机制可以直接定义到中间件中，这样查询两个角色是否具有继承关系就需要判断所有的显式角色层次和隐式角色层次。中间件的底层原子操作都不需要做修改，只需要在上层应用视图中添加有关角色命名部分的代码即可。

角色互斥可以添加到类 VirtualRole 中，并且通过一定的方式生成角色类 Role 的互斥关系。添加了互斥关系的中间件在进行角色层次关系的维护上还需要做一系列的修改，保证层次关系和互斥关系不出现矛盾。

5.3.2 在中间件中实现 RBAC 模型内缓存

在中间件中添加模型内缓存十分方便。针对不同的模型内缓存，首先需要添加一部分原子操作，然后在相关的应用视图上添加缓存更新的操作即可。缓存更新影响到的函数我们上一章都做了说明。

在这里以缓存权限角色指派关系为例来说明如何在中间件中添加对缓存的支持。首先需要添加如下的一些原子操作。

- 添加缓存权限角色指派 P_AddCacheRelationRolePermission(role, perm)
- 删除缓存权限角色指派 P_DeleteCacheRelationRolePermission(role, perm)
- 批量删除缓存权限角色指派 P_BatchDeleteCacheRelationRolePermissionByPermission (perm)
- 批量删除缓存权限角色指派 P_BatchDeleteCacheRelationRolePermissionByCache(cacherole, perm)
- 检验缓存的权限角色指派是否存在 P_ExistCacheRelationRolePermission(role, perm)
- 指定角色检验缓存权限角色指派是否存在 P_HasCachePermissionByRole(role)

这些原子操作和前面介绍的其它原子操作一样，都是很简单的一句 SQL 语句或者是对 LDAP 树的一个更新操作。

然后，需要重写在缓存更新时会产生问题的一些函数，这里包括

DeletePermission，DeleteRole，DeleteInheritance，DeleteRelationRolePermission，以及检查函数 CheckRelationRolePermission。缓存更新策略上一章已经介绍了一些，管理员也可以根据需要重新定义。

这样，整个中间件对上层的接口没有发生任何变化，模型内缓存的功能就已经实现。

5.4 RBAC 中间件的实现及相关讨论

由于我们定义的 RBAC 中间件用原子操作封装了底层的数据存储接口，我们不关心底层用的是数据库、LDAP 或者其它的存储介质，也不关心数据库的库结构、LDAP 的 Schema 如何设计。底层开发者只需要在定义好存储结构之后实现我们前面介绍的所有原子操作，中间件即可工作。

由于我们提供了多种上层用户视图，上层应用程序可以灵活调用这些接口实现对整个 RBAC 应用模型的管理。管理员也可以开发专门的针对查询访问控制系统的应用程序的接口，充分提高效率。

由于考虑到充分的灵活性和扩展性，我们实现的 RBAC 模型中间件并没有完整考虑到所有的 RBAC 特性。角色互斥、会话机制都没有包含在目前的中间件结构中。不过由于开放性的接口设计，这些新的功能可以很方便的添加到中间件中，从而增强中间件对 RBAC 模型的控制和管理。

第6章 结论与下一步工作

基于角色的访问控制模型 RBAC 是目前主流的访问控制模型，它比传统的自主访问控制和强制访问控制更优越，同时也提供了更高的灵活性和扩展性。目前的 RBAC 模型在理论上和应用中仍存在着诸多问题，本文针对如下的三个问题提出了自己的解决方案。

首先，当访问控制应用系统规模逐渐扩大以后，角色模型中角色的数目可能成百上千，并且和具体的业务逻辑密切相关。针对这种实际情况，传统的 RBAC 模型没有相应的对策，处理起来十分庞杂。本文从角色定义的语义出发，在角色集中引入维数的概念，提出一种全新的多维 RBAC 模型框架。我们给出了多维 RBAC 模型的形式化定义，并且详细讨论了基于多维 RBAC 模型的分布式角色管理模型。同时我们提出了一种角色命名机制来充分利用多维 RBAC 模型的特性简化系统角色管理。多维 RBAC 模型在 WebDaemon 系统中得以实施，并取得了成功。

然后，我们针对访问控制系统中查询操作远多于修改操作这个特性，在 RBAC 模型中引入缓存的概念，将原先通过递归运算得到的各种对应关系记录下来，从而提高后续查找的效率。我们讨论了一种 RBAC 模型外缓存和四种不同的 RBAC 模型内缓存，并针对不同的缓存策略给出合理高效的缓存更新算法，使得缓存系统得到合理充分的利用。我们随后分析比较了各种缓存算法的优劣，并在数据库和 LDAP 上进行了试验，取得了不错的统计结果。

最后，我们针对访问控制模块化的特点，提出了 RBAC 中间件的概念，并且给出具体设计方案和实现。该中间件对下端的存取模型给出原子操作接口，对上端的应用程序给出应用接口，并设计了灵活简便的方式实行整个模型的管理，方便访问控制模型的二次开发。多维 RBAC 模型和 RBAC 模型内缓存在中间件中也可以很方便的加以实现。本文在数据库和 LDAP 上进行了中间件的存取试验，并在一个大型访问控制系统中进行了配置和二次开发，基本实现了中间件的功能。

我们的上述工作都是基于访问控制系统的实际问题提出的解决方案，并且在

实际的访问控制系统中加以实施，充分评测各种方法的优劣。相信这些工作对今后该领域的应用研究有着指导性作用。

在下一步的工作中，多维 RBAC 模型还需要进一步扩展和完善，需要对限制进行更深入的研究，并详细探讨 MDARBAC 管理模型的管理策略；RBAC 模型内缓存有必要进一步仔细定义并添加到 RBAC 模型的规范中去；RBAC 中间件的功能需要进一步扩充，并且提供友好的用户界面方便管理员进行管理；一个包含所有这些 RBAC 特性的理论框架和中间件设计还需要进一步的研究。

参考文献

[1] 钟华, 冯玉琳, 姜洪安. 扩充角色层次关系模型及其应用. 软件学报, 2000, 11(6), pp 779-784.

[2] 桂艳峰, 林作铨. 一个基于角色的 WEB 安全访问控制系统. 计算机研究与发展(录用). 2002.

[3] 俞诗鹏, 桂艳峰, 赵琛, 林作铨. 多维 RBAC 模型及其应用. 北大金科网络实验室技术报告（已投稿软件学报）. 2002.

[4] Baltimore TrustedWeb. http://www.baltimore.com/solutions/TrustedBusinessSuite/TrustedNetworks/TrustedWeb/index.asp .

[5] Entrust GetAccess. http://www.entrust.com/getaccess/.

[6] IBM Tivoli. http://www.tivoli.com/.

[7] OpenLDAP. http://www.openldap.org/.

[8] ISO/IEC7498-2, Information Processing Systems-Open Systems Interconnection Reference Model-Part 2: Security Architecture. 1989.

[9] Hypertext Transfer Protocol -- HTTP/1.1. RFC 2616. 1999. http://www.w3.org/Protocols/rfc2616/rfc2616.html

[10] "Sun Software White Papers: RBAC in the Solaris Operating Environment," 2001.

[11] Ahn, G.-J. and Sandhu, R. S., Role-Based Authorization Constraints Specification, ACM Transactions on Information and System Security, Vol. 3, No. 4, 2000, pp. 207-226.

[12] Baldwin, R. W., Naming & Grouping Privileges to Simplify Security Managementin Large Databases, In IEEE Symposium on Research in Security and Privacy, IEEE Computer Society Press, 1990, pp. 116-132.

[13] Barka, E. and Sandhu, R. S., A Role-Based Delegation Model and Some Extensions, In National Information Systems Security Conference 2000, 2000.

[14] Barka, E. and Sandhu, R. S., Framework for Role-Based Delegation Models, In Annual Computer Security Applications Conference 2000, 2000.

[15] Bell, D. E. and Lapadula, L. J., "Secure computer systems: Mathematical foundations,"

MITRE, Technical Report MTR-2547, 1973.

[16] Bertino, E., Bonatti, P., and Ferrari, E., TRBAC: A Temporal Role-Based Access Control Model, ACM Transactions on Information and System Security, Vol. 4, No. 3, 2001, pp. 191-223.

[17] Bertino, E., Catania, B., Ferrari, E., and Perlasca, P., A Logical Framework for Reasoning about Access Control Models, ACM Transactions on Information and System Security, Vol. 6, No. 1, 2003, pp. 71-127.

[18] Bertino, E., Ferrari, E., and Atluri, V., The Specification and Enforcement of Authorization Constraints in Workflow Management Systems, ACM Transactions on Information and System Security, Vol. 2, No. 1, 1999, pp. 65-104.

[19] Beznosov, K. and Deng, Y., A Framework for Implementing Role-based Access Control Using CORBA Security Service, In Fourth ACM Workshop on Role-Based Access Control (RBAC'99), 1999.

[20] Biba, K. J., "Integrity considerations for secure computer systems," MITRE, Technical Report TR-3153, 1977.

[21] Bonatti, P., Vimercati, S., and Samarati, P., An Algebra for Composing Access Control Policies, ACM Transactions on Information and System Security, Vol. 5, No. 1, 2002, pp. 1-35.

[22] Cohen, E., Thomas, R. K., Winsborough, W., and Shands, D., Models for Coalitionbased Access Control (CBAC), In SACMAT'02, 2002.

[23] Demurjian, S. A. and Ting, T. C., Role-Based Access Control for Object-Oriented/C++ Systems, In First ACM Workshop on Role-Based Access Control (RBAC'96), 1996.

[24] Denning, D. E., A Lattice Model of Secure Information Flow, Communications of the ACM, Vol. 19, No. 5, 1976, pp. 236-243.

[25] Fernandez, E. B. and Hawkins, J. C., Determining Role Rights from Use Cases, In Second ACM Workshop on Role-Based Access Control (RBAC'97), 1997.

[26] Ferraiolo, D., Barkley, J. F., and Kuhn, R., A Role-Based Access Control Model and Reference Implementation Within a Corporate Intranet, ACM Transactions on Information and System Security, Vol. 2, No. 1, 1999, pp. 34-64.

[27] Ferraiolo, D., Cugini, J., and Kuhn, R., Role Based Access Control: Features and

Motivations, In Computer Security Applications Conference, 1995.

[28] Ferraiolo, D. and Kuhn, R., Role Based Access Controls, In 15th NIST-NCSC National Computer Security Conference, 1992, pp. 554-563.

[29] Ferraiolo, D., Sandhu, R. S., Gavrila, S., Kuhn, R., and Chandramouli, R., Proposed NIST Standard for Role-Based Access Control, ACM Transactions on Information and System Security, Vol. 4, No. 3, 2001, pp. 224-274.

[30] Freudenthal, E., Pesin, T., Port, L., Keenan, E., and Karamcheti, V., "dRBAC: Distributed Role-based Access Control for Dynamic Coalition Environments," TR2001-819, 2001.

[31] Giuri, L., Role-Based Access Control in Java, In Third ACM Workshop on Role-Based Access Control (RBAC'98), 1998.

[32] Goh, C. and Baldwin, A., Towards a more Complete Model of Role, In Third ACM Workshop on Role-Based Access Control (RBAC'98), 1998.

[33] Koch, M., Mancini, L. V., and Parisi-Presicce, F., A Graph-Based Formalism for RBAC, ACM Transactions on Information and System Security, Vol. 5, No. 3, 2002, pp. 332-365.

[34] Kuhn, R., Mutual Exclusion of Roles as a Means of Implementing Separation of Duty in Role-Based Access Control Systems, In Second ACM Workshop on Role-Based Access Control (RBAC'97), 1997.

[35] Lampson, B. W., Protection, In 5th Princeton Conference on Information Sciences and Systems, Princeton, 1971.

[36] Longstaff, J. J., Lockyer, M. A., Capper, G., and Thick, M. G., A Model of Accountability, Confidentiality and Override for Healthcare and other Applications, In Fifth ACM Workshop on Role-Based Access Control (RBAC'00), 2000.

[37] Moffett, J. D., Control Principles and Role Hierarchies, In Third ACM Workshop on Role-Based Access Control (RBAC'98), 1998.

[38] Moffett, J. D. and Lupu, E. C., The Uses of Role Hierarchies in Access Control, In Fourth ACM Workshop on Role-Based Access Control (RBAC'99), 1999.

[39] Na, S. and Cheon, S., Role Delegation in Role-Based Access Control, In Fifth ACM Workshop on Role-Based Access Control (RBAC'00), 2000.

[40] Nyanchama, M. and Osborn, S., Access Rights Administration in Role-Based Security Systems, In IFIP WG 11.3 Database Security, 1994, pp. 37-56.

[41] Oh, S. and Sandhu, R. S., A Model for Role Administration Using Organization Structure, In SACMAT'02, 2002.

[42] Osborn, S., Sandhu, R. S., and Munawer, Q., Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies, ACM Transactions on Information and System Security, Vol. 3, No. 2, 2000, pp. 85-106.

[43] Park, J. S. and Sandhu, R. S., Role-Based Access Control on the Web, ACM Transactions on Information and System Security, Vol. 4, No. 1, 2001, pp. 37-71.

[44] Ramaswamy, C. and Sandhu, R. S., Role-based access control features in commercial database management systems, In Proceedings of the 21st National Information Systems Security Conference, 1998.

[45] Sandhu, R. S., Lattice-Based Access Control Models, IEEE Computer, Vol. 26, No. 11, 1993, pp. 9-19.

[46] Sandhu, R. S. and Bhamidipati, V., An Oracle Implementation of the PRA97 Model for Permission-Role Assignment, In Proceedings of the 3rd ACM Workshop on Role-Based Access Control (RBAC'98), 1998.

[47] Sandhu, R. S. and Bhamidipati, V., Role-Based Administration of User-Role Assignment: The URA97 Model and its Oracle Implementation, Journal of Computer Security, Vol. 7, 1999.

[48] Sandhu, R. S., Bhamidipati, V., and Munawer, Q., The ARBAC97 Model for Role-Based Administration of Roles, ACM Transactions on Information and System Security, Vol. 2, No. 1, 1999, pp. 105-135.

[49] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E., Role-Based Access Control Models, IEEE Computer, Vol. 29, No. 2, 1996, pp. 38-47.

[50] Sandhu, R. S. and Munawer, Q., The RRA97 Model for Role-Based Administration of Role Hierarchies, In Annual Computer Security Applications Conference, 1998.

[51] Sandhu, R. S. and Samarati, P., Access Control: Principles and Pratice , IEEE Communications, Vol. 32, No. 9, 1994, pp. 40-48.

[52] Schaad, A., Moffett, J. D., and Jacob, J., The Role-Based Access Control System of a European Bank: A Case Study and Discussion, In SACMAT '01, 2001.

[53] Swift, M. M., Hopkins, A., Brundrett, P., Van Dyke, C., Garg, P., Chan, S., Goertzel, M.,

and Jensenworth, G., Improving the Granularity of Access Control for Windows 2000, ACM Transactions on Information and System Security, Vol. 5, No. 4, 2002, pp. 398-437.

[54] Ting, T. C., Demurjian, S. A., and Hu, M. Y., Requirements Capabilities and Functionalities of User-Role Based Security for an Object-Oriented Design Model, Database Security V: Status & Propects, 1992, pp. 275-296.

[55] Zhang, L., Ahn, G.-J., and Chu, B.-T., A Rule-Based Framework for Role-Based Delegation, In SACMAT'01, 2001.

[56] Zhang, L., Ahn, G.-J., and Chu, B.-T., A Role-Based Delegation Framework for Healthcare Information Systems, In SACMAT'02, 2002.

致谢

首先我要感谢我的导师林作铨教授。林老师在我读研究生前就开始指导我进入实验室做一些工作，并且从应用和理论两个层面教导我如何进行研究。我先后参加了实验室的多个小组，学到了很多东西，为我能做出这篇论文打下了良好的基础。在我发现对 RBAC 比较感兴趣的时候，林老师鼓励我积极从理论层面多探索 RBAC 的相关领域，并且具体实现到一个应用系统中，从而使研究真正有效。林老师先后给我两次机会到实际的公司里面考察，使我了解公司的具体需求，做出真正有实际效用的研究。

另外，我要感谢 WebDaemon 小组的所有成员，包括已经毕业的桂艳峰、石浩，以及仍然在读的赵琛、窦燕、张冲、赵新宇、努尔买买提。正是大家的通力合作使得我们能够做出来一个真正实用的访问控制系统，并且部署到企业中。我也正是从系统本身以及和大家的讨论中找到有意义的研究方向。

最后，我要感谢我的家人，王正秦老师以及实验室的所有其它同学。他们的鼓励和支持是我做好研究的力量源泉。