摘要

安全性是任何企业应用系统的基本组成部分，企业应用需要约束访问应用系统的用户和控制应用系统用户所能完成的操作类型。而以往的企业应用系统往往把安全内嵌入应用系统中，使得对安全管理复杂化，本论文从把安全与企业应用系统分离的思想并由此提高安全功能的可重用性为出发点，简化应用系统的安全功能的复杂度，使得应用系统只需关注其业务逻辑的具体实现，而把安全留给底层系统去实现，把安全看作是系统服务，通过系统提供安全服务、应用系统配置安全需求来实现企业应用系统的安全。

J2EE 应用服务器的安全服务正是基于上述思想，提供了基于声明性的安全模型和基于程序性安全模型。J2EE 规范为 EJB 和 Web 组件定义了简单的、基于角色的安全性模型。声明性安全模型体现在，组件使用标准的 XML 安全描述符描述安全性角色和许可，而不是将安全性嵌入在业务组件中。因此，它将安全性从业务级代码中隔离开，因为安全性只是为部署组件添加的新功能，而不是业务逻辑层面的内容。开发者借助于标准的 J2EE 部署描述符能够为 J2EE 应用提供基于应用安全性需求规范的保护。程序性安全模型体现在，当 J2EE 应用服务器的声明性安全服务无法满足企业应用的需求时，也可以由应用系统开发者和部署者自定义相关安全性要求，J2EE 应用服务器提供了相关的编程接口。

关键词 J2EE 应用服务器，安全服务，认证，访问控制，审计

Abstract

Security is a basic requirement of an enterprise software application system. Enterprise usually allow the legal users(or authenticated users) access to their application system and the legal users do some operations according to their permission. But software application system in the pass was designed embeded the code which exceute the security check and authentication into their application system, then it made the business logic disorder and the security function can't change following the future requirements(such as:change in the mapping from users to roles and the requirement of more fine granulrity of the permission checking), it also made the management of security more complex. This athesis is based on the ideal of detachment of the security funciton and the software application system and the reusement of securtiy function. It's target is simplifying the complexity of the security excutement and management, and make the application system doing implement their business logic regardless of the security. Leaving the security implement to the platform system. Regard the security as a platform's service and not their responsibility. They use the security service by declaring their securtiy requirements saving in an xml file(ejb-jar.xml) in a form of security deployment descriptions.

J2EE application server's security service is just based on the above ideal, and provides two types of security services: declarative security and programmatic security. J2EE specification define an simply security module based on role for EJB/web components. Declarative security module is realized by components can declare their security requirement in an xml file not embeded the code into their implement business logic function modules. So, it detached the security function and enterprise appliction system itself, and in the help of the security deployment description enterprise application system is still in the shield of required security. When the security deployment description is disable to express the requirement of security(some application system need more spatial security requirement), it also provides an interface through which the developers of enterprise application system can write codes in order to achieve their security requirements.

key words: J2EE Application server, authentication, permission check

华南理工大学

学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写的成果作品。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律后果由本人承担。

作者签名：吴国祥　　　　日期：2005年 6月 3 日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华南理工大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保密□，在____年解密后适用本授权书。

本学位论文属于

不保密□。

（请在以上相应方框内打“√”）

作者签名：吴国祥　　　　日期：2005 年 6 月 3 日

导师签名：　　　　日期：2005 年 6 月 3 日

第1章 绪论

1.1 研究背景

1.1.1 J2EE应用服务器

随着企业应用环境逐渐向 Internet/Intranet/Extranet 环境转移，以 Internet 为基础的企业分布应用不仅要求在分布式环境下实现信息的采集、管理、发布、交换和处理，而且要求能够快速开发和构建企业应用，并使所开发的应用具有易扩展性、互操作性、高可靠性、可伸缩性以及高安全性等，这样，传统的两层客户/服务器计算模式已不能适应这种需求。

为了满足这种企业应用，引入了三层计算模式，它由处理业务表示逻辑的表示层(Presentation Tier)、执行应用业务逻辑的业务逻辑层(Business Logic Tier)和描述应用所需数据的数据层(Data Tier)构成。而三层/多层计算模式需要以网络和分布式计算的底层技术为基础，构建一个整体应用框架，其中关键在于位于中间层的软件。在这种情况下，研究人员提出了 Web 应用服务器的概念[1]，即在面向 Internet 的 Web 计算环境下，为开发、部署、运行、集成、维护和管理中间层应用服务提供一个通用运行环境，用户只需关心中间层应用服务的业务逻辑，而中间层应用服务的名字解析、路由选择、负载平衡、事务控制、状态迁移、升级扩展等功能则都由 Web 应用服务器提供。而 Web 应用服务器中，有遵从 J2EE 系列规范的服务器，我们称它为 J2EE 应用服务器。

J2EE 的体系结构如图 1．2 所示，它是由 Sun 公司面向企业计算提出的一种规范。J2EE 规范为事务性 Web 应用的开发、部署、运行和管理提供一系列的规范和标准，主要包括 Java Servlets、JSP、EJB、JTA、JTS、JMS、JAXP、JMX、RMI－IIOP、JNDI、JCA、JAAS 和 JAF 规范等。这些 J2EE 规范为 Web 应用服务器的实现提供了一个完整的底层框架和一套标准的规范，为不同厂商的 Web 应用服务器产品的标准化提供了一条可行途径，使得在不同的 J2EE 应用服务器之上的应用组件也可以很好地进行互操作，从而降低移植的风险性和代价，提高应用的灵活性，保护用户的已有投资。

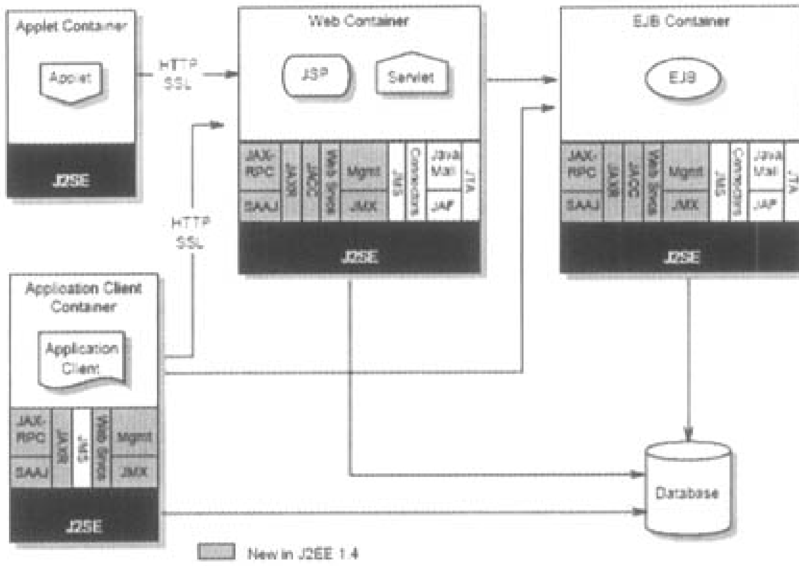

图 1.1 J2EE 的体系结构

1.1.2 J2EE 应用服务器的安全性

任何企业应用都有安全需求，譬如多用户环境中对敏感资源的受限访问以及在未受保护的公共网(如 Internet)传输数据等，这些都需要特定的安全机制和体系结构来满足。J2EE 规范[2]声明了安全需求，并指出具体的安全实现可能具有的特性:

(1) 认证(Authentication)指通讯实体间互相证明它们代表特定的身份，以便进行授权访问;

(2) 资源访问控制(Access Control for resources):指对资源的访问仅可被授权的用户或程序访问，从而达到完整性、机密性和可用性的限制;

(3) 数据完整性(Data Integrity):指信息发出后未被第三方修改过;

(4) 数据机密性(Confidentiality 或 Data Privacy):指信息仅对被授权的用户有效;

(5) 不可抵赖性(Non-repudiation):指一个用户不能否认其执行过的操作;

(6) 审计(Auditing):通过捕捉安全相关的事件，从而能够估计安全策略和机制的有效性。

1.1.3 Apollo 应用服务器简介

Apollo 应用服务器是由广州中间件研究中心推出的，遵循 J2EE 规范的企业级应用服务器。它已经被列为国家“985 工程”二期重点建设项目及广州市重点科技攻关项目。Apollo 基于现有的工业规范及各种安全方面的协议和标准，采用构件化思想，实现了 J2EE 规定的标准容器和服务，易于设计实现便于移植、互操作性好的可重用构件，为构建多层、分布的企业应用提供了一个稳定高效，安全可靠的平台。

Apollo 实现了 J2EE 规范定义的框架，其体系结构如图所示。Apollo 2.2 应用服务器基于 J2EE 体系结构,支持 J2EE1.3, 并支持 Web Service 和 JMX 的相关规范。Apollo 在整体设计中，采用了业界领先的 JMX 架构，主要功能模块均按 JMX 规范的要求创建，具有良好的灵活性和可扩展性，可以方便地进行管理和维护。Apollo 的主要功能模块均为“框架＋插件”的结构，可以在不修改已有源码的条件下替换原有功能模块的实现、增加新的功能模块或者删除已有的功能模块，从而使系统具有高度的灵活性。图 1.2 是该应用服务器的关键机制与核心功能。

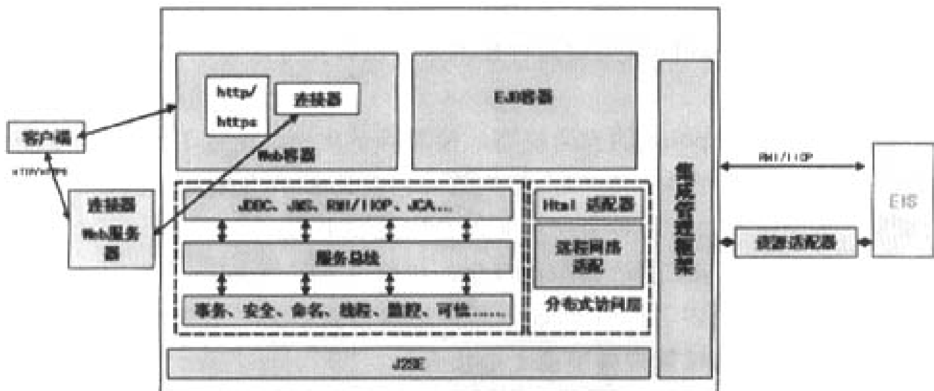

图 1.2 Apollo 应用服务器的关键机制与核心功能

1.2 论文工作

本论文主要研究 Apollo 应用服务器的安全服务，旨在为 Apollo 应用服务器提供一个具有高度灵活性和扩展性的安全框架，并通过各种可配置的安全策略和机制解决企业应用中的认证、授权、访问控制等安全问题。

论文的工作主要在如下几个方面：

(1) 定义安全参考模型，根据 J2EE 应用服务器的安全需求及开放式的软件架

构的要求，提出 Apollo 安全服务的参考模型，具体分为：认证鉴别模型、访问控制模型、审计模型及策略、安全域管理模型、代理模型等子模型；

(2) 安全服务的设计与实现，依据安全参考模型的指导，分层设计与逐步实现 Apollo 安全服务；

(3) 安全管理的设计与实现，基于 Apollo 应用服务器的管理框架，设计并实现对安全服务的管理。

1.3 论文组织

论文首先介绍了相关原理及技术，然后基于此原理及技术详细论述了为达到安全系统与应用系统分离这个主要目标所进行的系统设计与实现。论文的后续章节按以下方式组织：

第二章，首先概述了整个安全体系及 J2EE 应用服务器在整个安全体系中的位置及它主要关注的安全问题是什么，然后论述 J2EE 标准中的安全要求：基于容器的安全、资源验证需求、访问控制及授权需求等等；

第三章，介绍 JAVA 的安全体系的概念及相关术语，接着详细介绍了 JAAS 技术和 JACC 技术，Apollo 采用这两种技术来实现其安全服务功能，但并不依赖于现有的技术；

第四章，提出了 Apollo 的安全模型，接着基于此模型论述了 Apollo 安全服务的设计与实现，给出了 Apollo 安全架构的总体设计图,并分别详细描述了：应用层，表现层，实现层和数据层各个层是如何设计与实现的；

第五章，阐述了 JMX 管理模型的概念，接着展示了 Apollo 应用服务器系统内核管理模型，然后论述了作为 Apollo 应用服务器内核的一部分 Apollo 安全服务是如何实现与系统内核管理模型的无缝接合；

最后一部分，总结了以上的工作，及为继续完善本课题，提出下一步要做的工作。

第2章 J2EE 安全需求

本章概述了 J2EE 标准中的安全要求：基于容器的安全、资源验证需求、访问控制及授权需求等等。

2.1 概述

几乎所有的企业都有具体的机制和架构满足他们的安全需求。安全敏感资源能被许多用户访问，或者要经过不受保护的开放网络（如 Internet）传输，这就需要有安全机制保证如下几点[3]：

- 身份认证：在访问资源时，通讯双方必须相互证明其代表了一个具体的授权用户；
- 资源访问控制：基于保证资源的完整性、机密性、可用性等方面要求，资源总是控制被一部分人或程序访问；
- 数据完整性：采取措施保证数据信息不被第三方修改。例如，一个接收者接收公网数据的时，必须能够检测并丢弃那些在发送之后被修改的数据；
- 机密性或数据保密性：保证信息只被授权用户访问；
- 不可抵赖性：采取措施证明用户确实做了某些操作致使用户无法抵赖；
- 审计：捕获安全相关的事件，并写入日志。

2.2 安全需求

J2EE 标准规定，一个符合 J2EE 规范的应用服务器的安全体系必须满足以下的功能目标：

1) 可移植性：J2EE 安全体系结构必须支持应用程序的 Write Once，Run Anywhere 特性；
2) 透明性：应用组件开发商在写组件时不对安全感知；
3) 孤立性：J2EE 平台必须按照相应的部署描述符执行认证和访问控制；
4) 可扩展性：平台的服务的使用不能应为应用程序对安全的感知，而对应用程序的可移植性进行妥协；
5) 灵活性：可方便地对安全技术和机制进行集成和配置；
6) 抽象性：应用组件地安全需求通过部署描述符指定即可；

7) 独立性：可以使用多种安全技术实现安全行为和部署契约；

8) 兼容性测试：符合 J2EE 规范的安全需求；

9) 安全互操作：在一个 J2EE 产品中执行的组件能够调用来自另外一个开发商提供的服务，无论它们是否实施了相同的安全策略。

2.2.1 基于容器的安全

在 J2EE 环境中，组件的安全由容器提供，容器为达到该目标，容器必须提供两种类型的安全服务：声明性安全和程序性安全。

声明性安全

声明性安全是指把应用系统的安全架构以一种与系统分离的方式表述，安全架构包括：安全角色、访问控制、身份认证要求等。在 J2EE 平台中，部署描述符是声明性安全的主要载体。

部署描述符是一个契约，它规定了应用系统提供者和部署者或装配者双方应该遵循的原则。它可以被应用系统的编程者用来表达应用系统在部署阶段的组件与环境及组件与组件之间的安全相关要求。

容器在运行时刻，对某个组件采取的安全策略正是来源于部署描述符和部署授权。

2.2.1.1 程序性安全

程序性安全指对于安全有特别要求（用声明性安全无法表达的安全要求）的应用系统，安全裁决由应用系统自己而不是容器做出。本标准要求程序性安全提供的编程接口（API）由 EJBContext 接口的两个函数和 HttpServletRequest 接口的两个函数组成：

- isCallerInRole（EJBContext）
- getCallerPrincipal（EJBContext）
- isUserInRole（HttpServletRequest）
- getUserPrincipal（HttpServletRequest）

这些方法使得组件能够根据调用者或的远程用户的安全角色在组件内作安全逻辑运算并作出安全裁决。例如：isCallerInRole 可能会有如下的使用方式（来自 EJB 规范）：

```
public class payrollBean ... {
    EntityContext ejbContext;
```

```
    Public void updateEmployeeInfo(EmplInfo info) {
        oldInfo = ... // read from database;
        // 当调用者拥有特定的角色，只能从事特定的操作
        if ( ejbContext.isCallerInRole("payroll1") ) {
        ...
        // 角色 payroll1 的特定操作

        } else if ( ejbContext.isCallerInRole("payroll2") ) {
        // 角色 payroll2 的特定操作
        ...
        }else {
            throw new Exception("调用者的角色没有进行任何操作的权限！");
        }
    }
}
```

2.2.1.2 授权模型

J2EE 授权模型是基于安全角色的概念，安全角色是应用系统的供应商或部署者定义的一个逻辑上的用户组，在操作环境中，部署者把角色映射成安全标识符（如：主体，和组）。声明性安全和程序性安全都使用安全角色。

声明性授权能够用于企业 Bean 方法的访问控制上，通过在该 Bean 的部署描述符上附加一个 method-permission 元素，用 method-permission 元素包含可以访问该 Bean 的方法的那些安全角色。所以如果一个调用者的主体所对应的角色属于被允许访问该方法之列，则调用者被允许执行该方法。访问 web 资源也可以用类似方法实现。

安全角色在接口 EJBContext 中的方法 isCallerInRole 和接口 HttpServletRequest 中的方法 isUserInRole 中使用，两者都返回 true 如果调用者的主体相应的角色属于允许访问之列。

◆ 角色映射

无论采用声明性安全还是采用程序性安全，对 web 资源或企业 Bean 资源的安全限制的实施最终依靠于一个判断，判断调用者的主体的角色是否属于给定的允许角色之列。资源容器基于调用者主体的安全属性把调用者的主体映射到某个安全角色上，做出上述判断。例如：

- 在一个操作环境中，部署者已经把安全角色映射到一个用户组。在这样的情况下，资源容器通过检索调用者主体的安全属性值获得其所属的用户组，根据判断调用者所属的用户组是否与安全角色映射到的那个用户组相匹配，来判断该调用是否处于这个安全角色之中。
- 在同一个安全域中，部署者已经把安全角色映射到一个主体的名称上。在该情况下，资源容器通过检索调用者主体的安全属性值获得其主体名称，根据判断调用者主体名称是否与安全角色映射到的那个主体名称相匹配，来判断该调用者是否处于这个安全角色之中。

主体安全属性值的来源，随着各个 J2EE 平台产品具体实现的不同而不同。安全属性值可以存放于主体的机密区域或者一个安全上下文中，再有甚者，也可以从第三方检索得到，例如，从可信的目录服务或安全服务中检索。

◆ 用户验证

用户验证是一个用户向系统证明身份的过程。之后，使用这个被验证了的身份裁决是否具有访问某个资源的权限。终端用户采用以下两种之一的客户端验证类型：

- web 客户端
- 应用程序客户端

2.2.1.3 web 客户端验证

要求 web 客户端能够让一个用户被 web 服务器采用以下几种机制验证（由应用系统的部署者或系统管理员决定对一个应用采用哪种验证机制）：

- HTTP 基本验证

  HTTP 基本验证是唯一被 HTTP 协议所支持的一种基于用户名和密码验证方式。Web 服务器要求 web 客户端（浏览器）验证终端用户，作为验证请求的一部分，web 服务器会把验证域传给终端用户。Web 客户端收集到用户输入的用户名和密码之后，把它传输给 web 服务器，web 服务器就会对该终端用户进行验证。

基于 HTTP 的验证并非是一个安全验证，因为 web 客户端只是把密码进行 64 位简单的编码之后就传输出去了。外加的保护措施可以弥补这种机制的弱点。可以通过在传输层施加安全措施（例如 HTTPS）或者在网络层施加安全措施（例如 IPSEC 或者 VPN）来保护密码的传送。由于该机制的广泛使用，所以本标准要求应用服务器必须支持。

- HTTPS 客户端验证

  终端客户使用 HTTPS（即通过 SSL 来传输 HTTP 协议）是一个强

健的验证机制。该机制要求用户拥有一个公钥证书（Public Key Certification）。虽然，PKC 很少被终端用户使用，但是它在基于电子交易的应用系统中或在支持在浏览器中单一登陆的策略中，非常有用。J2EE 平台也必须支持该种验证机制。

■ 基于表单页的验证

使用浏览器内嵌的验证机制时，登陆界面的外观不能被应用系统定义，所以 J2EE 标准引入基于 HTML 或 JSP/SERVLET 的可自定义的表单登陆界面的表单页验证机制。

2.2.1.4 web 单一登陆

由于 HTTP 协议是无状态的，但是很多 web 应用需要会话支持以维持与客户端之间的状态，因此需要：

1) 把登陆机制和策略作为 web 应用所部署在的环境的一个属性；
2) 当访问其他所有的 web 应用时，使用同样的登陆会话所代表的用户；
3) 只有当用户访问资源要跨越安全策略边界时，才被要求再次验证用户身份。

通过 web 登陆验证所要求的机密数据被附加到会话中，资源容器使用这些机密信息为这个会话建立一个安全上下文。容器同时使用这个安全上下文作为用户访问资源权限审查的凭据，和建立一个与其他组件交互的安全关联对象。

2.2.1.5 登陆会话

在 J2EE 平台中，web 容器必须支持登陆会话。当一个终端用户成功地进行了登陆验证之后，资源容器必须为该用户建立一个登陆会话上下文，该会话包含了登陆用户的机密信息。

◆ 应用程序客户端验证

应用程序客户端是指运行在客户端上的程序，程序不通过 web 浏览器和 web 服务器的帮助直接访问企业 Bean 组件或 web 资源。应用程序客户端跟其他的 J2EE 应用组件一样也应该运行在某个容器提供的一个受管理的环境中。当用户访问受保护资源时，应用程序客户端应充当 J2EE 平台对用户进行验证的手段。

◆ 滞后验证

由于身份验证过程要花费一定的代价，所以，滞后验证的需求应运而生，它要求身份验证操作只在被需要的时候执行，所以当使用滞后验证时，仅当用户访问受保护资源时才需要验证。

2.2.1.6 用户身份认证需求

关于用户验证，J2EE 产品提供者必须满足以下要求：

所有的 J2EE web 服务器必须为每一个 web 用户维持一个用户登陆会话。一个登陆会话可能会跨越多个 web 应用，并允许用户只需登陆一次就能访问多个 web 应用的资源。企业应用系统能够独立于登陆信息维护和建立安全环境的实现细节（这留给资源容器去完成）。

Web 服务器必须支持滞后验证，当必须进行验证时，以下三个之一验证机制应被使用：

- HTTP 基本验证（同时包括在 SSL 之上的 HTTP 基本验证）
- SSL 双向验证
- 基于表单的验证

2.2.1.7 未验证用户

Web 容器应该支持那些未经验证的用户访问 web 资源，这是在因特网上访问资源的一个通用模式，web 资源容器通过 HttpServletRequest 接口的方法 getUser 主体调用返回空值（null），就表示未经验证的用户，但是这不同于 EJB 容器情况，在 EJB 标准中要求接口 EJBContext 的方法 getCaller 主体总是返回一个有效的主体对象，不能返回一个空值（null）。所以，当一个未经验证的用户访问了 web 组件，而这个组件访问 EJB 组件时，为了能够顺利访问 EJB 组件，web 容器必须提供一个主体，具体怎么提供，标准没有任何限制，但是建议使用运行时代理（Run As）。

2.2.1.8 应用程序客户端验证

为了满足服务端 EJB 容器和 web 容器对资源访问的身份验证和访问控制的要求，应用程序客户端所在的容器必须能够对终端用户提供相应的验证机制。具体采取的技术，可以因客户端容器的实现不同而不同，可以集成 J2EE 产品的验证系统、提供单一登陆能力或者在一启动应用程序时就进行验证，容器应能支持滞后验证，容器应该提供一个恰当的用户交互界面以便于收集用户输入的登陆信息。此外，客户端应用程序可以提供一个实现了 javax.security.auth.callback.CallbackHandler 接口的类并在部署描述符中指明该类

的类名来实现自定义回调句柄，从而自定义验证界面，当然，部署者可以重载该回调句柄，强制使用系统默认值。如果一个自定义回调句柄被使用了，由客户端容器的负责实例化该对象，并在身份认证时使用它与用户交互。

2.2.2 资源验证需求

企业资源部署在多个安全策略域中，可能不同于组件所在的安全策略域，由于采用了多种验证机制，所以要求支持以下两种方式：

1) 预配置身份认证：资源容器必须能够用预先部署上的主体和认证信息来验证身份；
2) 程序性认证：资源容器必须支持组件可通过编程接口提供主体和认证信息。

此外，规范还推荐了但不是必备的三种技术：主题映射、调用者模仿和证书映射。

2.2.3 访问控制及授权需求

为支持 3.2.3 的授权模型，以下要求必须满足：

- 代码授权：为保证系统的正常运作和安全，J2EE 产品必须严格使用 J2SE 的类和方法。给予 J2EE 产品所需求的最小权限许可集（在 J2EE 标准 6.2 节中给出），所有的 J2EE 产品都应该能在这个许可集中正常部署及工作；
- 调用者授权：J2EE 产品必须执行在部署阶段配置的访问控制规则，这些规则的设置在 EJB 和 servlet 规范中详细描述。
- 调用者身份传播（Propagate Caller Identities）：J2EE 产品应支持把它配置成调用者身份传播方式，在该方式下对资源的访问控制验证都使用最初登陆的那个用户身份，所以在 EJB 中调用接口 EJBContext 的方法 getCaller 主体所返回的主体对象应与在该调用链上第一个 EJB 调用得到的结果相同，在 jsp 或 servlet 上的情况应该与此相同。
- 以指定身份运行（Run As Identities）：J2EE 产品具体以指定身份运行的能力，允许部署者指定 EJB 组件或 web 组件运行时的身份，在这种情况下，应把这个指定的身份作为以后调用的传播身份而不是初始调用者的身份

2.3 本章小节

本章主要从 J2EE 规范出发，描述了具体的安全要求，说明 J2EE 应用服务器应具备怎样的安全特性，后续章节中的设计与实现的目标是本章提出的要求。

第3章 JAVA 安全体系及技术

本章首先介绍 JAVA 的安全体系的概念及相关术语，接着详细介绍了 JAAS 技术和 JACC 技术，Apollo 采用这两种技术来实现其安全服务功能，但并不依赖于现有的技术(在第 4 章中将会详细论述 Apollo 的安全框架)。

3.1 JAVA 安全体系

3.1.1 概述

Java 安全体系主要在三个方面上保证了 Java 程序的安全性[4]：Java 语言[5]本身的安全性、Java 虚拟机[6]、Java 核心类库，基于此我们能够方便的开发安全的应用系统。

为了便于论述，首先介绍几个在本论文以下的章节中都会用到的概念术语：

- 主体（principal)：主体是一个用来被企业部署的安全服务通过某种验证协议验证的实体，用实体名称和它所附带的验证数据来标识实体，实体名称和验证数据的格式因验证它的验证协议的不同而不同。
- 主题（subject)：表示认证请求的发起方，如用户或服务；一个主题可以包含多个主体，比如某个人构成一个主题，它有多个主体：名字主体（John Doe)，社会保障号（123-45-6789)，和用户名主体（johnd)，所有的这些主体使得该主题与其他主题区分开来；
- 权限（Permission)：指允许代码执行的具体操作，如可以读取但不能写某个文件，权限包含三个要素：权限类型、权限名称及允许操作；
- 策略（Policy)：策略定义了执行代码被授予的操作权限；
- 域（Domain)：由一组资源对象组成，对该域内的所有对象统一实施某个策略。域是用来分类或隔离各个保护单元。沙箱就是一个具有固定边界的域的实例；

Java 的安全性在语言层次上表现在：

第一，Java 语言被设计为是类型安全的，这便于使用并减轻了程序员的负担，减少出现象使用其它编程语言时（如 C 或 C++）所出现的编程错误。

第二，自动内存管理、内存垃圾回收及字符串和数组的越界检查等，都有助于程序员编写安全的代码。

第三，Java 语言不允许指针运算，这保证了程序不会出现内存越界访问问题，而该问题在其他语言中常常导致严重问题，此外 Java 语言规范明确定义了未被初始化变量的行为，堆内的变量将被自动初始化(如果程序没有初始化它的话)，Java 语言编译器把未初始化就使用该变量这种情况视为语言错误，在编译阶段将会给出编译错误信息；

第四，Java 的垃圾回收机制，Java 的垃圾回收器在运行时刻自动、动态回收程序不再使用的内存块，这使得程序员不必考虑什么时候是安全释放内存块之时的头痛问题，从而避免了因在不适当时机释放了内存块而导致内存泄漏问题。

第五，Java 的强类型特征使得一块内存区不能被转换成任意类型的数据结构，此外由于 Java 语言是纯面向对象的语言，它的访问控制修饰符：public、protect、private、final 有助于构建强健的程序。

Java 安全在 Java 虚拟机（JVM）上表现在：

首先，JVM 依靠软件机制使得所有的 Java 程序都运行在一个特定的沙箱内，沙箱首先要核查在其内运行的 Java 程序（或类）是否是合法的、有没有被签名等等安全性检测，再次在运行时不断地检测程序是否具有进行该项操作的权限并形成一道围栏以隔离和控制对沙箱外部资源的访问。沙箱模型的实质是，本地代码是可信的，因而可全部访问关键系统资源（如文件系统）[7]；而下载的远程代码（一个 Applet）是不可信的，因而只能访问由沙箱内部所提供的有限资源。沙箱模型从 JDK1.0、JDK1.1 到 JDK1.2 的逐步增强其安全性和演化，最终在 JDK1.4 中的沙箱模型如下图 2.1 所示：

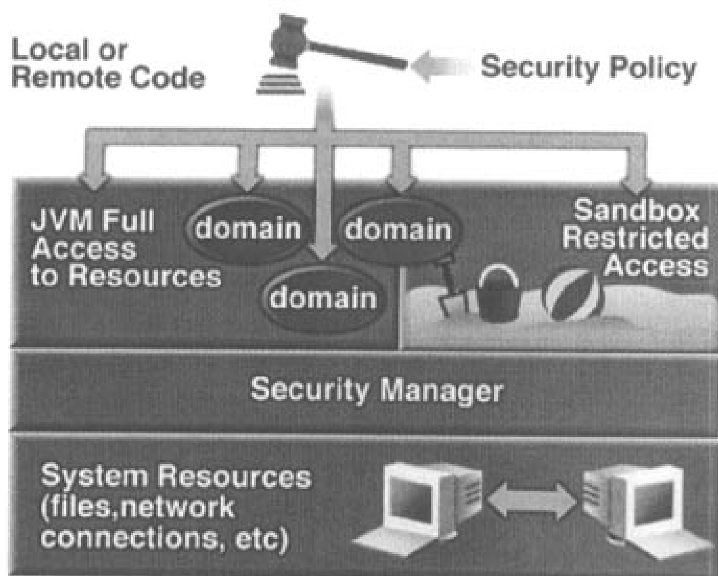

图 2.1（来自于 SUN）

沙箱模型在最新的 JDK 版本中，已经逐渐泛化成一个保护域的概念，不同的保护域有不同的访问资源的权限，如图 2.1 中，如果处在左边顶端的箭头所指的域中时，则它具有所有的访问权限，如果处在最右端箭头所指的域，正好跟前者相反，该域具有及其严格的访问控制，这两者中间的域，在资源的访问控制程度上也处于它们中间。

其次，编译器和字节码校验器可保证只有合法的 Java 字节码才能被执行。字节码校验器与 Java 虚拟机一起保证了在运行时的语言安全。

再次，类装载器可定义一个本地名空间，这可以保证一个不可信 Applet 不会干扰其它程序的运行。

最后，对重要系统资源的访问是通过 Java 虚拟机来传递的，并由 SecurityManager 类进行预先检查，这就将不可信代码的活动限制到了最小的程度。

Java 安全性在核心类库上的表现在：Java 的核心类库组成了整个 Java2 平台的安全体系结构，核心类库包含在以下包结构中：java.security、java.security.acl、java.security.interfaces、java.security.cert 及 java.securtiy.spec 等构成整个 Java 加密体系（JCA），此外在应用级上对 Java 安全进行了扩展：Java 加密扩展（JCE）、Java 认证与授权服务（JAAS）、Java 安全套接字扩展（JSSE）等，将在下一节中详细论述。

3.1.2 Java 2 安全体系总体结构

Java 的安全体系构由不同的模块组成，如图所示。

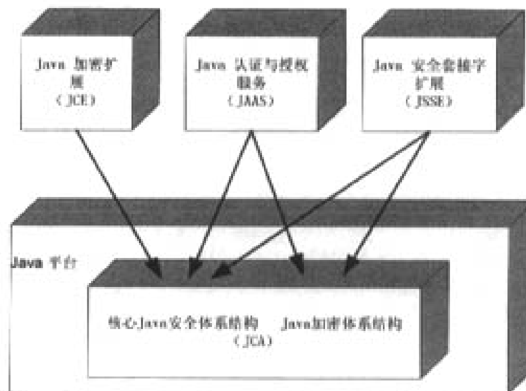

图 2.2

其中，在图的下半部分，核心的 Java 安全体系结构与 Java 加密技术体系结构（JCA）是 Java 2 平台的一部分，共同构成了 Java 2 平台安全。在图的上半部分，Java 加密扩展（JCE）、java 安全套结字扩展（JSSE）、Java 认证与授权服务（JAAS）是三个标准的 Java 安全扩展。

3.1.3 核心 Java 安全体系结构

核心 Java 安全的组成部分包括：类加载器、字节码校验器、安全管理、访问控制器、许可、策略、保护域，如图所示。

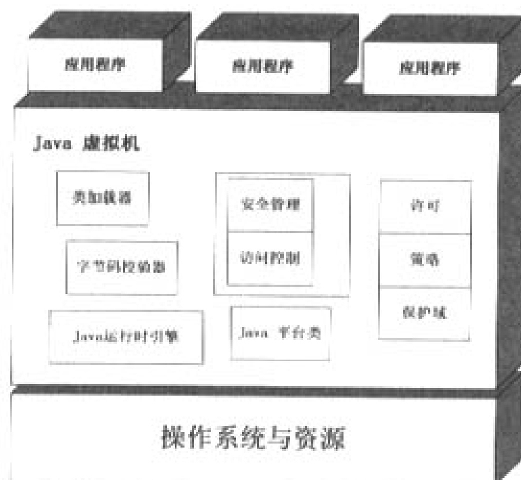

图 2.3

1) 类加载器：类加载器是 Java 平台的核心部分之一[8]。Java 平台一般包括三个类加载器：基本类加载器、安全类加载器、URL 类加载器。其中，基本类加载器负责加载核心 Java 平台类；安全类加载器扩展了基本类加载器，提供了将类、权限域代码进行关联的能力；URL 类加载器扩展了安全类加载器，提供了从 URL 列表中加载类的能力。用户可以根据自己的需要定制自己的类加载器。

当类加载器从输入流中加载类时，它将首先检查其自身缓存中的类（如果它不是基本类加载器，则首先检查基本类加载器的缓存），决定该类是否已经被加载了。如果类已经被加载了，则类加载器将直接返回这个已经加载的类；如果类没有被加载，类加载器将从与其关联的输入流中读取类代码，并交给字节码校验器进行处理。如果校验成功，则创建该类，并加入到自己的缓存中。类对其他类的引用过程也同样适合。

所以，从上述过程我们可以看出，当加载或查询一个类时，一个类加载器将只查询基本类加载器的缓存和自身的缓存，以及与其关联的输入流。同一平台

（JVM）上不同的类加载器在加载、引用类时，相互之间不查询，他们之间是相互隔离的，这将导致了类名字空间的隔离，这种类加载机制可以有效的防止“特洛依木马”型的入侵方式，即攻击者无法通过向平台装入一个恶意的类而非法的访问平台资源。

2) 字节码校验器：字节码校验器对类加载器所加载的 Java 类进行字节码流分析，并校验它是否符合 Java 语言定义的规则。这是在两个阶段中完成的：第一阶段分析 Java 类字节流自身，第二阶段校验这个类对其他类的引用，例如对其他类的引用是否合法等[9]。

3) 安全管理其余访问控制器：安全管理器负责对系统重要资源的访问控制。一个 Java 平台只能运行一个安全管理器，在启动 Java 平台时从命令行中指定。安全管理器中主要是一些检查方法，在 Java 2 中，安全管理器的主要功能最终都交由访问控制器实现。访问控制器提供了细粒度、可配置的访问控功能，基本上是对安全管理器的一种替换，尽管安全管理器仍然被支持。所谓细粒度，是指对权限许可的划分细致，所谓可配置是指系统的安全策略由专门的策略文件负责配置。

4) 许可：许可对象包括文件、套接字、显示界面、属性、运行时等，许可的操作包括：读、写、打开、关闭等。

5)策略与保护域：安全策略为开发人员提供了一种方便的安全策略配置机制，用于指定将哪些权限授予哪些保护域[10]。保护域是通过 URL 定义的，用于指示从哪里来的代码应受到一组特定安全策略的限制。一个保护域也可以通过一个或多个定义了代码从哪里来的身份定义。

安全策略是以文本形式存储在安全策略文件中的。管理人员可以手工、或者在工具的支持下方便地操纵该文件。安全策略文件的一般格式（其中的关键字是大小写无关的）如代码所示

```
[Keystore “keystore_URL”,”keystore_type”]
Grant[Signedby “signers”]
[,CodeBase “URL”]
{
   Permission Permission_Class [“Target_Name”]
        [, “Permission_Actions”]
        [, signedBy “SignerName”];
};
```

其中：

- Keystore 项为已签名的类指定向哪一个URL和什么类型的存贮机制进行查询；
- Signedby 出现了两次：第一次出现指示了谁为Java代码进行签名，第二次指示了与授权关联的、正在JVM中执行的代码必须由一个已命名的签名者签名。
- CodeBase 用于将一个授权与一个资源位置（代码从该处加载）关联在一起。如果 CodeBase URL 以/*为结尾，它指示位于对应目录中的所有 Java 类文件与 JAR 文件。如果 CodeBase URL 以/- 结尾，它指示位于对应目录及其所有子目录中的所有 Java 类文件和 JAR 文件。

一个具体的例子如代码所示。其含义为来自于地址 http://www.wplus.com.cn 的所有由 wplus 签署的 Java 代码，对于资源"c:/user/admin" 拥有读、写权限。

```
Grant CodeBase "http://www.wplus.com.cn/-", Signedby "wplus", {
    Permission java.io.FilePermission "c:/user/admin", "read, write";
};
```

6) Java 加密体系结构（JCA）：JCA 为基于 Java 的加密技术提供基本的加密框架，主要为数据的完整性提供支持。特别地，通过消息摘要对数据的传输进行支持。JCA 较多地依赖于非对称密钥体系。

在实现上，JCA 主要由一下四个部分组成[11]：

- java.security：一组核心的类与接口，提供了 JCA 服务提供商框架和加密技术操作 API；
- java.security.cert：一组证书管理类和接口；
- java.security.interfaces：一组接口，用于封装和管理 DSA 和 RSA 公钥和私钥；
- java.security.spec：一组类与接口，用于介绍公钥算法、私钥算法和参数规范。

7) Java 加密技术扩展（JCE）：JCE 是在 JCA 的基础上做了扩展，包括加密算法、密钥交换、密钥产生和消息鉴别服务等接口。JCE 主要用于数据的保密性，较多地依赖于对称密钥体系[12]。

与 JCA 类似，JCE 并不执行具体的加密算法，主要是连接应用和实际算法实现程序的一组接口。软件开发商根据 JCE 接口，将各种算法实现后打包成一个 Provider，可以动态地加入到 Java 运行环境中。

在实现上，JCE 主要由如下三个部分组成：

- java.crypto：一组核心的类与接口，提供了 JCE 即查即用 SPI 和加密技术操作 API;
- java.crypto.interface：一组用于封装和管理 Diffie-Hellman 密钥的接口；
- java.crypto.spec：一组用于密钥算法和参数规范的类。

8) Java 安全套接字扩展（JSSE）：JSSE 是支持安全数据传输技术的一组接口（含 API 和 SPI）。JSSE 不仅支持 SSL，还支持传输层安全等协议。

在实现上，JSSE 主要由如下三个部分组成[13]：

- javax.net.ssl：一组与 JSSE API 相关的核心类和接口。
- javax.net：一组支持基本套接字与服务器套接字工厂的接口。
- javax.security.cert：一组支持基本证书管理的接口。

3.2 J2EE 认证授权技术

JAAS[14]即 Java 认证与授权服务，它提供了一种基于客户身份的控制机制。核心 Java 安全体系结构中的安全策略主要针对 Java 代码的特性：代码的来源，代码是否被签名及签名者，这是一种以代码为中心的访问控制。但是在现实世界的企业应用系统中，以用户为中心的访问控制更为常见。JAAS 正是构建在以用户为中心的访问控制的基础之上，提供了一套认证、授权的标准服务。利用 JAAS 和 JDK1.2，一个应用程序可以提供代码中心的访问控制、用户中心的访问控制或者是两者的结合。

JAAS 是为如下需要而设计[15]：

- 可延展性：Java 程序界面要求鉴别和许可能够很容易地加以扩展。
- 可插入性：不同的系统可以很容易地把它们新的或者现存的鉴别机制包含到 JAAS 框架中去。
- 兼容性：JDK1.2 中基于代码的访问控制结构和 JAAS 中新的基于用户的访问控制机制可以独立地共存,而且可以完美地结合起来以实现复杂的安全策略。

3.2.1 JAAS 授权

传统的（JDK）授权方式是以代码为中心的，而 JAAS 是建立在传统的授权方式之上的，并且是以用户为中心的授权方式[16]。在 JAAS 下，相关的问题不再

是（像在 Java 2 平台安全体系结构中那样）“哪些是这段代码可以做的”，而变为“这个认证用户的访问权限是什么”，本节以下部分将着重论述 JAAS 授权的整个过程。

主题类对象代表了在给定系统中经认证后的用户。在 JAAS 架构内部，主题包含一组主体对象（及其他相关用户信息及安全属性信息），其中每个主体对象表示同一个用户的不同“身份”。

在以代码为中心的访问控制中保护域针对的主要对象是源代码（SourceCode），而在 JAAS 中保护域是针对的主要对象是主体（principal），当然，由于 JAAS 上建立在传统的访问控制之上，所以 JAAS 自然也支持以源代码为中心的访问控制。JAAS 通过用某一组主体来描述一个安全域。当系统策略设置了这样的安全域和授予它的权限之间的映射后，如果要用安全域的权限检查是否应当授予用户某个请求的权限，那么在主题中包含的、与运行这段代码的认证用户相对应的主体对象必须与这个安全域中所包含的主体对象相匹配[17]。

JAAS 主题类提供了两个静态方法，称为 doAs 和 doAsPrivileged。这些方法期待的输入是认证的用户的主题实例和 PrivilegedAction 的一个实例（它的 run() 方法应当包含需要访问受保护的资源的业务逻辑）。基本思路是应用程序应当首先认证用户，对认证的用户建立了主题后，这个用户可能希望执行的每一个操作都包装为 PrivilegedAction 、并由程序代表该主题执行。

3.2.2 JAAS 认证

JAAS 建立在一种称为可插入的认证模块（Pluggable Authentication Module，PAM）的安全体系结构之上。PAM 的体系结构是模块化，这意味着它设计为可以通过交换模块，支持从一个安全协议组件无缝地转换到另一个协议组件。这个框架中定义良好的接口使得无需改变或者干扰任何现有的登录服务就可以加入多种认证技术和授权机制。PAM 体系结构可以集成范围广泛的认证技术，包括 RSA、DCE、Kerberos 以及 S/Key，因而 JAAS 也可以集成这些技术。此外，这个框架与基于智能卡的认证系统和 LDAP 认证兼容。

就像许多 Java 2 平台技术一样，JAAS API 定义了应用程序代码与将要执行业务逻辑的物理实现之间的干净抽象。这个抽象层不用重新编译现有的应用程序代码就可以作为登录模块的运行时替代。特别是，应用程序写到 LoginContext API，而认证技术提供程序则写到 LoginModule 接口。在运行时，LoginContext 将读取配置文件以确定应使用哪一个（一些）登录模块对访问特定应用程序的用户进行认证。

JAAS 所使用的认证方案以两种非常重要的实体为基础：主体和主题。为了

惟一标识一个主题（这是认证的关键部分），一个或者多个主体必须与这个主题相关联。最后，一个主题可能拥有安全相关的属性，称为凭证（credential）。凭证可以是从简单的密码到复杂的加密密钥的任何东西。

应用程序通过实例化一个 LoginContext 对象开始认证过程。LoginContext 查询一个配置文件以确定进行认证所使用的一种（或者多种）认证技术以及相应的一个（或者多个）LoginModule。一个非常简单的 LoginModule 可能会提示输入用户名和密码并对它们进行验证。高级的可能会使用现有的操作系统登录身份进行身份验证。理论上，甚至可以将一个 JAAS 的 LoginModule 构建成与指纹识别器或者虹膜扫描仪交互[18]。

3.3 本章小节

本章主要介绍了java的安全体系，及后续章节用到的安全技术。Java 安全体系主要在三个方面上保证了 Java 程序的安全性：Java 语言本身的安全性、Java 虚拟机、Java 核心类库。后续章节将使用的java安全技术包括：JAAS认证与授权技术、java的加密、解密、消息摘要技术等等。

第4章 安全服务的设计与实现

本章首先提出了 Apollo 的安全模型，接着基于此模型论述了 Apollo 安全服务的设计与实现，给出了 Apollo 安全架构的总体设计图,并分别详细描述了：应用层，表现层，实现层和数据层各个层是如何设计与实现的。

4.1 Apollo 安全参考模型

根据 J2EE 的安全需求规范，本论文提出的 Apollo 安全参考模型，从各个方面定义了不同类型的元模型结构，对设计与实现符合 J2EE 规范的 Apollo 安全服务起了总体指导作用。

4.1.1 概述

J2EE 应用服务器介于操作系统与企业应用系统之间，其安全主要包括三个问题：验证、授权及访问控制、审计（即 AAA 问题，Authentication，Authoration，Auditing），和与此相关的管理问题，本论文提出的 Apollo 安全参考模型，总体图如图 4.1 所示：

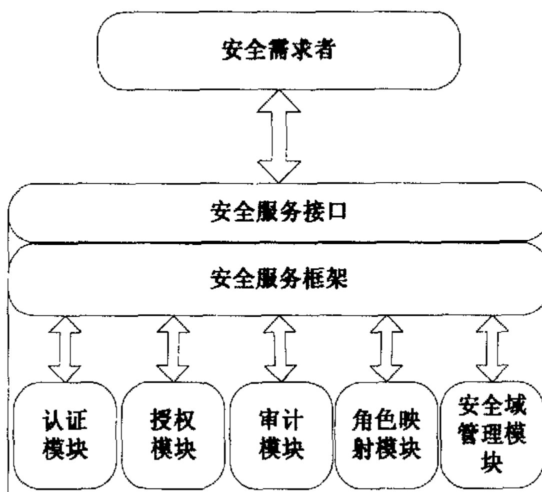

图 4.1 Apollo 安全参考模型

Apollo 安全参考模型分成：认证鉴别模型、访问控制模型、审计模型及策略、

安全域管理模型、代理模型等等，以下分别对各个模型进行论述。

4.1.2 鉴别认证模型

认证（Authentication）是鉴别申请者（用户或系统实体）是否具有其所声称的身份和/或特权(privilege)的过程[19]。其中特权是一种其拥有者不一定唯一、可能由多个用户共享的安全属性，如用户的角色。该模型是了为了满足 J2EE 应用服务器对安全敏感资源的访问进行控制的要求，只能让许可用户访问到。因 J2EE 规范要求应用服务器支持多种认证方式（详见 3.2.1），所以要求认证模型必须具有：灵活性、可扩展性、可插拔性，能够利用已有的认证技术或开发新的认证机制并将它们集成到认证控制模型中。本文提出的模型如下图 4.1 所示：

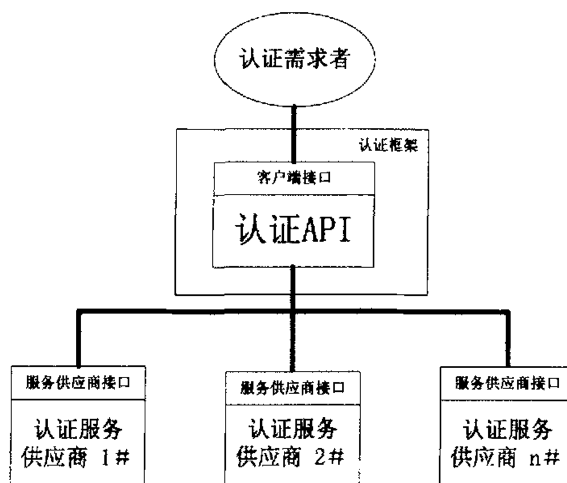

图 4.1 鉴别认证模型

图中，认证 API 是模型的核心组件，认证需求者首先与认证 API 交互，然后由认证框架通过读取配置文件中配置的系统当前使用的认证服务模块调用相应的认证服务供应商模块对用户进行认证，认证需求者不直接跟认证服务提供商交互，这使得在该模型下，认证需求者无需改变代码就可以方便的替换认证服务供应商模块，从而实现了可以灵活采用不同认证机制认证用户的特点。

4.1.3 访问控制模型

依据第 3 章中，规范关于访问控制的要求（基于角色的访问控制），本章提出了基于角色的访问控制模型（Role Based Access Control RBAC），因访问控制与授

权是两个相互联系的过程，授权过程在先，只有通过授权用户才具有相应的权限之后，访问控制才能起到保护资源的作用，所以在提出该模型之前，本文先提出了与之对应的同样的基于角色的授权模型，引入角色的概念是在用户和对象之间加入一个中间层，RBAC 主要的动机和特征是定义和执行与组织机构有自然对应关系的安全策略[20]，该模型如图 4.2 所示：

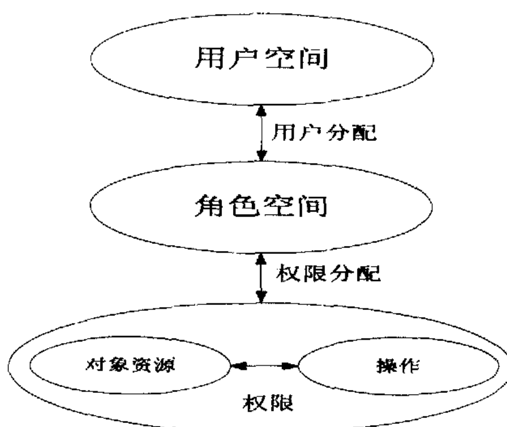

图 4.2 授权模型

图中，模型的关注点在于：(1) 用户空间与角色空间的映射,这是一个多对多的映射,实现了给一个特定的用户或用户组分配相应的角色的任务；(2) 角色空间与权限空间的映射，实现了给特定的角色分配一定的权限的任务，同样，该映射也是多对多的映射；(3) 系统资源与在其之上的操作之间的映射，该映射也是多对多的。如果把用户空间 U 定义成：{U1, U2, U3, ,,Un}, 角色空间 R 定义成：{R1,R2,R3,,,Rn}，权限空 P 间定义成：{P1,P2,P3,,,Pn}，则授权可以描述成一个三元组：（Ui，Ri，Pi），表示为某个用户被授予某种角色，且授予对某种资源的某种操作的权限。

有了基于角色的授权模型之后，本文参照了 CORBA 规范[21]提出了一个通用的访问控制模型，当然它适合于基于角色的访问控制，该模型如图 4.3 所示：

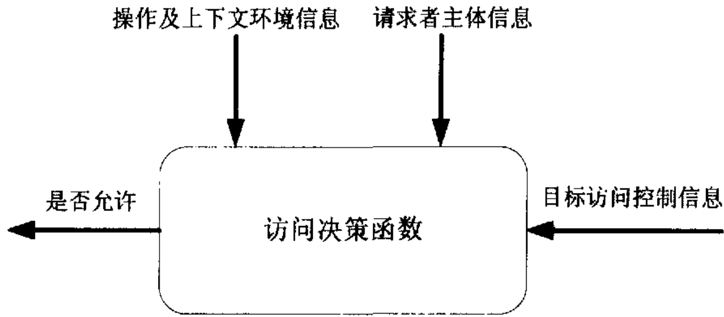

图 4.3 访问控制模型

模型中，访问决策函数需要三个输入条件：目标访问控制信息、操作及上下文环境信息、请求者主体信息，首先根据主体信息验证请求者是否是一个已被认证了身份的主体，如果没有，则根据上下文环境信息调用相应的系统认证服务提供者模块对请求者进行身份认证，之后，进行决策判断，输出是否允许该请求者访问目标资源。

4.1.4 审计模型及策略

在目前 J2EE1.4 规范中对应用服务器没有审计要求[22]，但是规范中提到这将是以后的发展方向，所以，为了使得 Apollo 能够适应新规范，本文也对审计模型做了研究与设计工作。

安全审计(security auditing)就是通过纪录系统中安全相关事件的细节，协助完成对实际或试图进行的安全入侵的检测[23]。一般地，审计将系统产生的安全相关的事件通过审计策略筛选之后，把策略认为重要的事件信息保存到文件或数据库。由审计框架提供审计编程接口函数（API）,系统通过调用审计接口函数，把相应的审计事件发送到审计框架中，同时审计框架管理者根据当前配置的审计策略对审计事件进行处理，并采取相应的动作，这些动作包括写日志、图像或声音报警、严重的可以立即关闭相关服务并通知系统管理员等等操作。审计模型图如下所示：

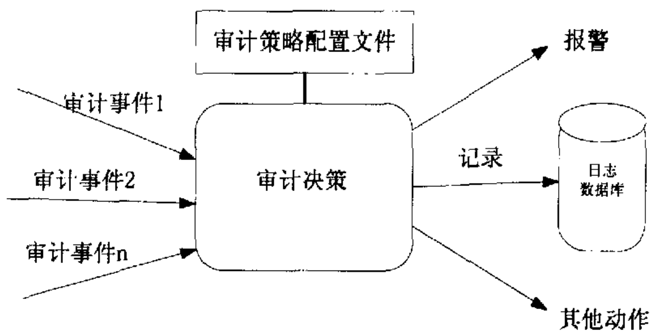

图 4.4 审计模型

系统产生的审计事件的内容应包含以下各项：

(1) 事件类型（如访问资源的读操作事件、写操作事件、对象方法调用事件等等）；

(2) 事件发生的行为主体；

(3) 事件发生的行为客体；

(4) 行为主体的操作成功与否；

(5) 事件发生的时间；

4.1.5 安全域及管理

安全域是安全参考模型的基础，安全域由一组对象组成，同一个安全域的对象具有某种共性、遵循共同的规则，也就是说对域内对象的安全相关行为施以相同的安全策略。安全策略的配置和管理是安全域的关键，目前所支持的策略包括访问控制、鉴别、角色映射、代理和审计等策略。

在一个安全域中要对安全策略进行配置、管理、实施，所以自然就引出了安全域的管理者这样一个角色，由安全管理者负责处理域中安全策略、域间映射等事务。安全域之间的关系有两种:层次关系(Hierarchies)，指域之间的关系为包含与被包含的关系，如图 4.5;和联邦关系(Federated)，指域之间的安全策略及相关策略信息存在某种形式的映射，以便实现域之间的调用，如图 4.6 所示。

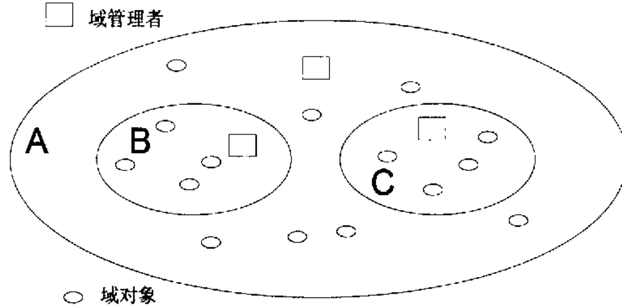

图 4.4 域层次关系

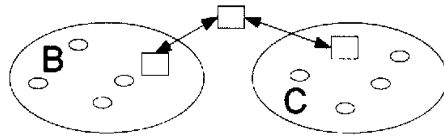

图 4.5 域的联邦关系

根据层次关系，安全域可分为子域和父域，没有父域的安全域称为根域，而其它安全域称为非根域，每个非根域有且仅有一个父域，而且子域内对象的安全行为不仅被自己的安全策略控制，还被父域所控制;而对于联邦关系，只有不继承于同一个根域的两个安全域之间的联邦关系才有意义，而且这两个域之间应存在某种形式的映射;这样，继承于同一个根域的所有安全域包括此根域形成一颗树，于是对象系统配置的所有根域对应的树便形成了一个森林。安全域间的这种关系为域之间的对象调用提供了可能。

安全管理就是对安全策略和安全策略域中对象进行管理。包括管理安全策略域本身(产生和删除)、管理域成员(设置对象所在的策略域)、管理安全策略(设置安全策略细节)。

4.1.6 代理模型

代理即用户/主体将其标识或特权授权给其他用户/主体完全或受限地使用的行为[25]。在一个分布式环境中，客户端对一个服务器端组件的调用要求其完成业务逻辑功能，一般地，在这个过程中该组件往往会调用服务端的其他组件以完成请求的功能，这样，客户端的一个调用导致了其他组件的一连串调用，形成一个调用链（Calling Chain），如图 4.6 所示。

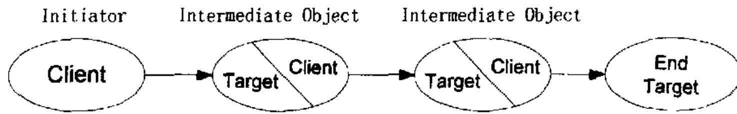

图 4.6 调用链

从某种意义上来说，代理模型的本质是一种授权模型，是对 4.1.1.2 节中讲到的授权模型的扩展，与之的区别在于前者授权动作的发起者是系统管理员，在程序运行之前就完成的，有配置文件保存结果，而后者授权动作的发起者是客户端程序，在程序动态运行过程中完成的，结果不保存。为了便于表述，首先介绍几个概念：

(1) 初始者(Initiator)：调用发起的初始者，如图 4.6 中的 Client;

(2) 中间对象（Intermediate Object)：调用链中既不是初始者又不是目的端对象的对象；

(3) 目的端对象（End Target)：调用链中最后一个对象；

(4) 直接调用者：调用链中当前对象的直接前驱，初始者没有直接调用者。

由于访问控制的存在，当一个调用链需要访问受保护资源时，就存在身份认证、调用许可检查等问题，在 Apollo 的代理模型中存在两种方式以满足 J2EE 规范的要求（详见 3.2.2 节)：一种是初始者传播（Propagation） 如图 4.7 所示，

图 4.7 Propagation 模式

也就是说，初始者主体的访问控制信息（Initiator Principal）可以授权给链上的其它对象，使其在特定环境下可以代表自己进行某些操作；

另一种是以指定身份运行（Run As Identities） 如图 4.8 所示。

图 4.8 Run As 模式

也就是说，调用链上的中间对象可以以它们自己的身份（Run As Principal),

使用自己的权限属性在其它对象上进行操作。

4.1.7 域的映射策略

域的映射用于在跨域访问时，内的映射包括有：主体映射、角色映射和证书映射，引入映射策略的主要目的在于处理安全域之间的联邦关系。假定对象 01 位于安全域 A 中，而对象 02 位于安全域 B 中，而 A 与 B 具有联邦关系，这样当 01 调用 02 时，由于 A 与 B 所拥有的安全策略信息不同，所以需要将 A 上主体的安全信息映射到 B 的主体安全信息上，根据映射的主体安全信息内容的不同，映射种类可分为:

- 主体身份映射(Principal Mapping):将主体的身份标志进行转换;
- 证书映射(Credential Mapping)将主体的安全属性进行转换。

此外，通过映射策略，为 J2EE JCA (Java Connector Architecture)规范[26]中的几种可选的映射行为的实现提供了基础。

4.2 Apollo 安全服务概述

根据 J2EE1.4 标准中关于安全的要求（详见第 3 章），Apollo 安全架构不仅实现了标准的所有安全要求，而且对标准中没有提及的其他安全要求进行了扩展，比如在客户端对 EBJ 调用的互操作过程中对调用函数的参数进行安全性检查等等，将在下述章节中详细论述。

4.2.1 Apollo 安全服务特点

Apollo 安全架构实现了一个配置性高，扩展性好，灵活的软件架构，使得各种现有的安全技术能构方便的集成到 Apollo 的安全架构中，有如下特点：

- 提供了基于 JAAS 的认证机制；
- 提供除了规范要求的基于角色的访问控制外的用户自定义访问控制机制；
- 支持第三方插件；
- 基于 JAAS 的扩展认证机制；
- 提供了灵活的模块插拔机制；
- 提供了易扩展的附加功能；
- 提供安全服务供应商接口；
- 额外支持 JACC1.0 标准。

4.2.2 Apollo 安全服务功能

安全服务的使用者通常是各种资源容器，EJB 容器[27]和 WEB 容器[28]等，和构建在应用服务器之上的企业应用系统。这些使用者的共性是要求应用服务器提供如下四个基本功能：

1) 对用户进行身份认证；
2) 对资源进行访问控制，保证只有授权用户才能访问；
3) 保证数据机密性和完整性；
4) 对用户的访问操作进行审计。

4.3 Apollo 安全服务架构设计

Apollo 安全服务架构的设计是依据上节中提出的 Apollo 安全参考模型，总体上采用了层次模型，分别从应用层、表现层、实现层、数据层上定义了整个框架，各个层分工、协作在完成各自的功能同时，为上层提供服务，最终在应用层和表现层上体现了 Apollo 的安全服务。

Apollo 安全服务的架构图如图 4.9 所示：

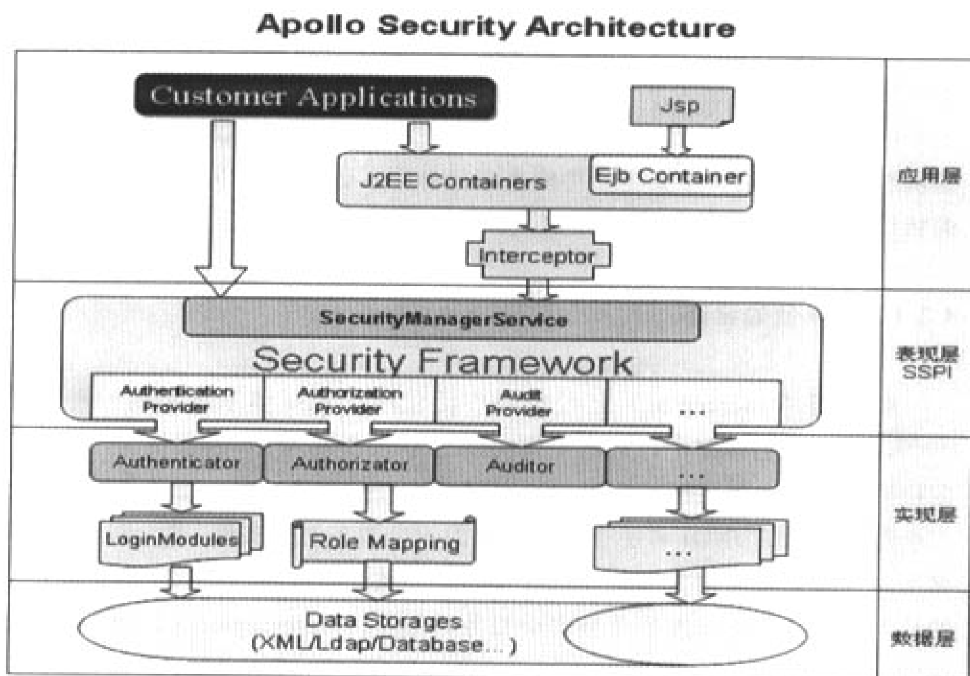

图 4.9 Apollo 安全服务的架构图

Apollo 应用服务器在安全方面，主要体现为四个层次：应用层、表现层、实现层和数据层。

- 应用层 － 主要描述在 Apollo 中，从用户角度来看，客户应用程序和 J2EE 组件（如 EJB、JSP 等）是如何访问安全服务的。
- 表现层 － 主要描述安全服务所对外公布的接口及本身安全服务框架的设计与实现。
- 实现层 － 主要描述 Apollo 的各个安全服务组件的设计及实现机制。
- 数据层 － 主要描述如何管理用户数据，如：User、Group、Role 等。

以下分别对各层进行详细论述：

4.3.1 应用层设计与实现

应用层是 Apollo 安全架构的最高层，本节中主要描述在应用层的程序（主要是指 EJB 或 WEB 容器）如何使用安全服务。

根据第三章 J2EE 规范需求，Apollo 在应用层上提供了基于声明性的安全服务和基于程序性的安全服务。

应用层主要解决用户程序和 J2EE 容器（EJB 容器、WEB 容器）如何与安全服务交互问题。在 Apollo 中，一个客户应用程序可以通过两种形式来调用安全服务。一种方式是通过 Apollo 提供的容器与安全服务的交互机制，让容器代理实现安全控制（即基于声明性的安全服务），在 Apollo 中，采取截获器（Interceptor）的机制来实现（论文在安全参考模型中已介绍了该技术）。另一种是提供安全控制的接口，用户通过显式调用该接口实现安全控制。

4.3.1.1 截获器的设计

截获器（Interceptor）已经成为广泛使用于系统体系结构的一种设计模式[29,30]，在本安全框架中，它的主要功能是为 EJB 或 WEB 容器提供用户验证、权限检测等功能。为了使整个架构具有较好的可配置性，截获器通过配置的 Apollo 安全服务对象的 JNDI 名字查找系统中的安全服务对象实例，即：截获器－>JNDI 名字 －>安全服务对象实例，这样使得在不改变截获器代码（即客户程序代码）的情况下系统可以动态的替换安全服务对象实例。Interceptor 接口及相关类图 4.10 所示：

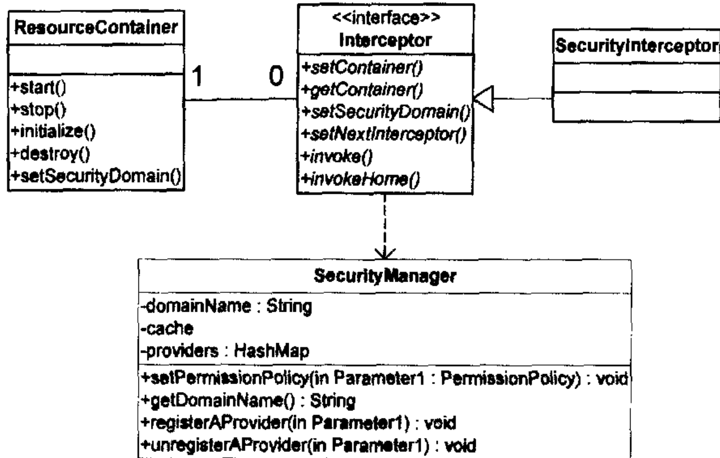

图 4.10 Interceptor 接口图

图 4.10 中 Interceptor 是广泛应用于软件体系系统结构的一种设计模式资源容器类 Container 通过 接口 Interceptor 的实现类 SecurityInterceptor 获得 SecurityProvider 接口的服务，Container 与 SecurityInterceptor 类之间通过配置文件关联（interceptor.xml）：

```
<container-configurations>
      <container-configuration>
         <container-name>Standard CMP 2.x EntityBean</container-name>
         <call-logging>false</call-logging>
         <invoker-proxy-binding-name>entity-rmi-invoker</invoker-proxy-binding-name>
         <sync-on-commit-only>false</sync-on-commit-only>
         <insert-after-ejb-post-create>false</insert-after-ejb-post-create>
         <container-interceptors>
            ....
            <interceptor>com.wplus.appserver.j2ee.ejb.plugins.SecurityInterceptor</interceptor>
            ....
         </container-interceptors>
      </container-configuration>
</container-configurations>
```

图 4.11 interceptor 配置文件

图 4.11 中是 interceptor.xml 文件，用于配置容器截获器，其中的<interceptor>标签标明该容器使用的截获器的类名，这样当容器启动时，读取该配置文件获得当前配置的截获器的类名，并把它初始化，创建实例，它的 invoke 方法用于安全检测：public Object invoke(Invocation mi) throws Exception，参数 mi 为调用组件的包装器，内部含有被调用的组件的类名、方法名、参数、调用者的身份信息等等信息，截获器通过分析传入的 mi 参数，访问 SecurtiyProvider 类，首先判定该调用者是否是一个合法用户，然后判定它是否具有相应的权限。

SecurityInterceptor 类与 SecurtiyProvider 类之间通过 JNDI 名关联：

SecurityProvider 类的实例表示一个安全域的管理者（本文在 4.1.1.4 节中描述了安全域及其管理者的概念）。在 J2EE 规范中没有描述一个企业应用系统在部署时应如何与一个操作环境中的实际的安全域相关联，本文通过在部署时增加一个扩展文件来达到此目的，如果要在一个 web 应用中增加安全域，则增加文件（在 WEB-INF 文件夹下）apollo-web.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE apollo-web PUBLIC "-//Wplus//DTD Web Application 1.0V2//EN"
          "http://www.wplus.com.cn/j2ee/dtd/apollo-web_1_0.dtd">
<apollo-web>
    <security-domain>java:/jaas/web-console</security-domain>
    <jsp-precompile>true</jsp-precompile>
    <use-session-cookies>false</use-session-cookies>
</apollo-web>
```

图 4.12 apollo-web.xml 配置文件

图中的配置信息在部署以后，系统将它们存在与该 web 应用相关的一个操作环境对象中，其中标签<security-domain>用于指定一个安全域的名称（例如图 4.12 中：web-console），该名称同时代表了该安全域的安全管理者实例的 JNDI 名称(例如图 4.12 中：java：/jaas/web-console)，这样 SecurityInterceptor 类可以用该 JNDI 名称去查找 JNDI Server（Java 名字目录服务器，Apollo 应用服务器内置的一项服务），以获得 SecurityProvider 实例的一个 Handler（引用）。例如在一个截获器内可以编写这样的代码以获得安全域管理者实例：

```
……
// 从环境中获得 SecurityManager 的 JNDI 名称
String securityDomain = getContext().getSecurityDomainJNDIName();
// 从环境中获得请求者主体信息
Principal p  =  …;

// 创建一个连接到名字目录服务的上下文对象
IntialContext iniCtx = new IntialContext();
try {
        // 从该上下文对象中查找 SecurityManager 引用
        SecurityManager sm =
                        (SecurityManager) iniCtx.lookup(securityDomain);
        ……
        // 认证用户
```

```
        sm.authentication(p);
        ……
      } catch (NamingException e) {
        throw e;
  }
```

声明性安全服务的配置

在表现层，安全服务的使用者具有两种方式，一种是上节论述的截获器通过代码调用 Apollo 的安全服务相关接口实现，另一种是本节要论述的声明性安全服务。声明性安全服务通过在部署描述文件中描述安全要求，对 EJB 组件来说是 ejb-jar.xml 或对 web 组件来说是 web.xml ，使用上节论述的技术实现。描述的格式在 J2EE 规范中有明确定义，这里以 ejb-jar.xml 文件为例说明，如图：

```
<method-permission>
    <role-name>payroll</role-name>
    <method>
        <description>Only the 'payroll' role can access the business
            interface 'updateBalance(Long count)'.</description>
        <ejb-name>Account</ejb-name>
        <method-intf>Local</method-intf>
        <method-name>updateBalance</method-name>
        <method-params>
            <method-param>java.lang.Long</method-param>
        </method-params>
    </method>
</method-permission>
```

图 4.12 声明性安全描述文件 ejb-jar.xml

在图中，角色 payroll （被标签<role-name>所包含的内容）被赋予对 EJB 组件 Account 的方法：updateBalance（java.lang.Long）调用的权限。在调用 EJB 组件时，平台如何实现该安全性限制条件，本文将在下面章节中设计。

部署描述文件的解析

Apollo 遵循 Java Authorization Contract for Container(JACC)标准来解析部署描述文件(ejb-jar.xml)。Ejb Container 初始化时，装载 ejb-jar.xml，获取 ejb 对象的安全控制信息，Apollo 通过实现 JACC 标准，把授权信息保存在 PermissionPolicy 对象中。同时，Ejb Context 提供了 isCallerInRole(String role)接口，来实现程序性安全。

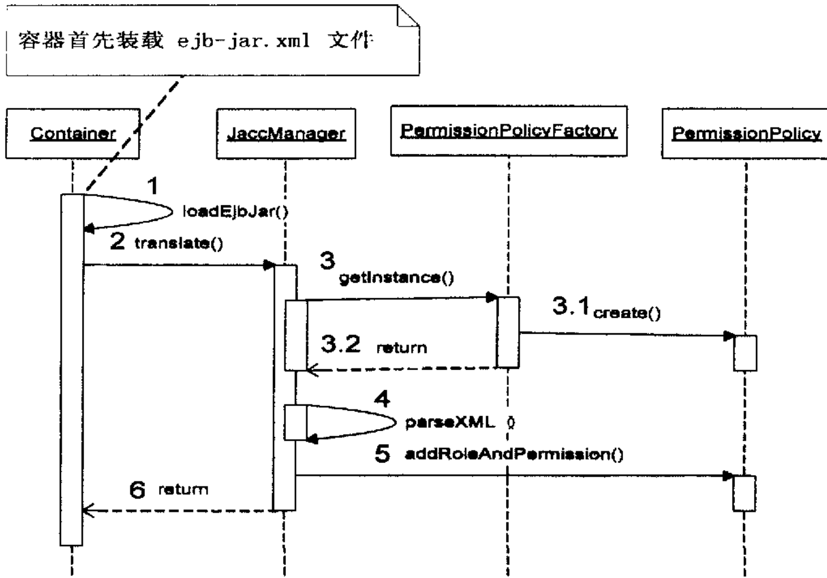

图 4.13 ejb-jar.xml 解析过程

上图中的类 PermissionPolicy 是一个存储企业应用系统部署描述中关于安全策略信息的类，当一个企业应用系统部署在 Apollo 平台之时，相应的部署器首先读取 web.xml 或 ejb-jar.xml 或 apollo-web.xml 或 apollo.xml 文件（如果它们存在的话）解析其部署时声明的安全需求描述信息，创建一个 PermissionPolicy 实例，并将解析之后的结果存放于一个 PermissionPolicy 实例中，最后将这个 PermissionPolicy 实例的 Handler 赋给当前资源容器，即通过调用资源容器方法：setPermissionPolicy（PermissionPolicy pd），由资源容器把该 Handler 传给 SecurityManager。

Ejb-jar.xml 的解析过程如下：

(1) 容器载入 ejb-jar.xml 并解析；

(2) 容器调用 JaccManager 的 translate()对 ejb-jar.xml 中的安全相关的节点进行解析；

(3) JaccManager 调用 PermissionPolicyFactory 类工厂对象的 getInstance()，获得存放 EJB 组件的安全策略信息；

(4) JaccManager 对象把有关的安全策略信息翻译成角色和权限的对应关系；

(5) JaccManager 调用 PermissionPolicy 对象把角色和权限的对应关系存放到 PermissionPolicy 中保存起来；

(6) ejb-jar.xml 文件解析完毕，程序控制流返回到容器。

4.3.1.2 声明性安全服务的实现

声明性安全服务的实现是在资源容器内部，当容器装载完组件对应的 ejb-jar.xml 文件时，一个客户端请求到达资源容器之时也就是容器对请求者的身份、权限进行安全检测之刻。其时序图如图 4.13 所示：

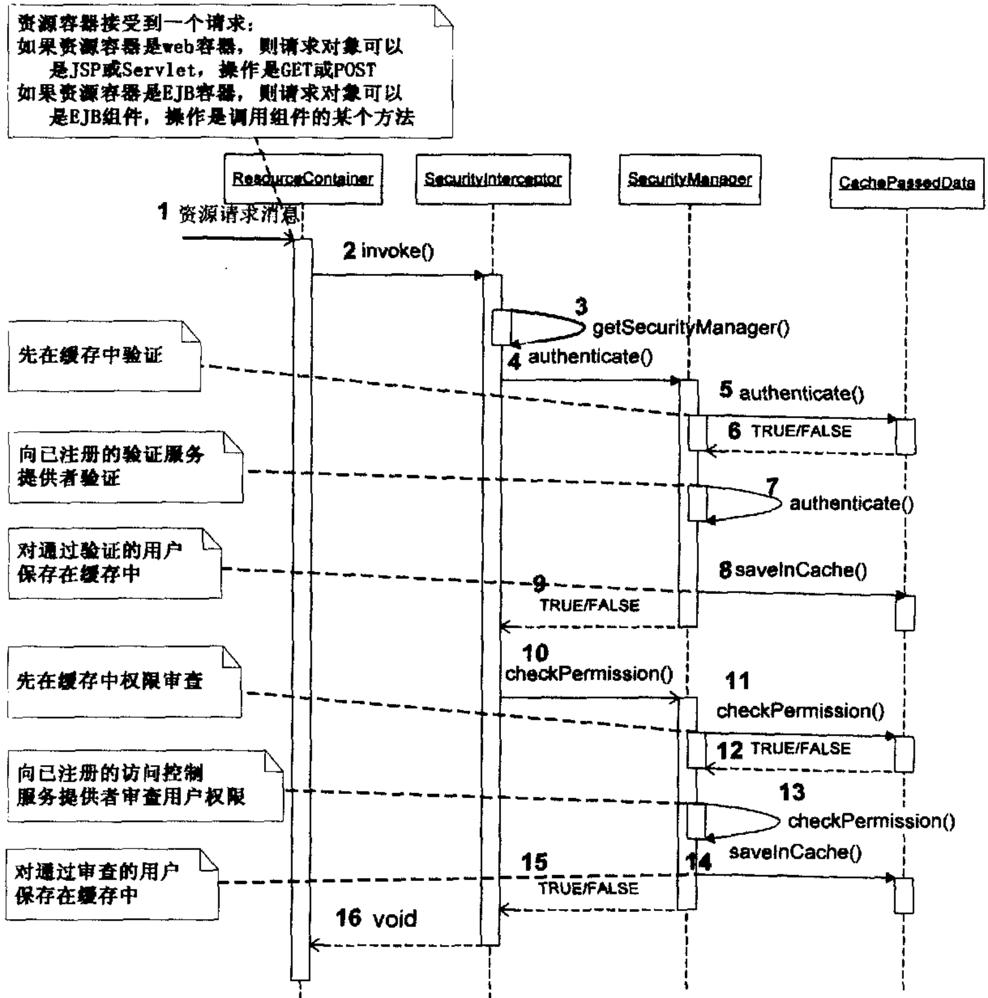

图 4.13 声明性安全服务时序图

本图说明了应用层是如何与表现层交互实现声明性安全服务：在图中资源容器类：ResourceContainer 对象和安全截获器类：SecurityInterceptor 对象是属于应用层对象，安全框架管理者类：SecurityManager 对象和缓存类：CachePassedData

对象属于表现层对象，声明性安全服务在应用层和表现层上的实现过程（其他层上的实现分别在各自的章节中论述）为：

(1) 资源容器接受到客户端的一个请求，如果资源容器是 web 容器，则请求对象可以是 JSP 或 Servlet，操作是 GET 或 POST 如果资源容器是 EJB 容器，则请求对象可以是 EJB 组件，操作是调用组件的某个方法；

(2) 安全截获器截获到该请求，方法 invoke()被触发；

(3) 安全截获器首先根据配置的安全域名称，并以此名称为参数从 JNDI 服务器中查找安全管理者对象的实例；

(4) 找得安全管理者实例之后，首先进行用户身份的合法性验证动作，安全管理者的 authenticate()方法被调用；

(5) 安全管理首先查看缓存对象 CachePassedData，该用户是否已经被验证过，如果在缓存中找到该用户，则在缓存中进行验证，并进行(6)，如果在缓存中找不到该用户，则进行(7)；

(6) 将验证结果返回给安全截获器，本次验证结束；

(7) 安全管理者查询其内部注册信息，向已注册的验证服务提供者提出验证请求，并获得其验证结果；

(8) 将通过验证的用户信息保存在缓存对象 CachePassedData 中；

(9) 将(7)中的验证结果返回给安全截获器，本次验证结束；

(10) 如果通过身份验证，则对该用户进行权限审查，进行(11)，如果没有通过身份验证，则本次客户端的资源请求由于安全问题而无法完成，安全截获者会抛出异常，向资源容器提示没有正常返回，用户最终得到表示用户非法请求出错信息；

(11) 安全管理者首先查看缓存对象 CachePassedData，该用户是否已经被审查过具有访问该资源的权限，如果在缓存中找到该用户，则进行权限审查，并进行(12)，如果在缓存中找不到该用户，则进行(13)；

(12) 将审查结果返回给安全截获器，本次权限审查结束；

(13) 安全管理者查询其内部注册信息，向已注册的访问控制服务提供者提出权限审查请求，并获得其结果；

(14) 将通过审查的用户信息保存在缓存对象 CachePassedData 中；

(15) 将(13)中的审查结果返回给安全截获器，本次权限审查结束；

(16) 如果通过权限审查，则安全截获器正常返回（返回类型为 void），没有通过权限审查，则安全截获者会抛出异常，向资源容器提示没有正常返回，用户最终得到表示用户没有权限的出错信息。

上图中的 CachePassedData 用于缓存已被验证过的或通过权限审查的用户的信息，这样下次同样的用户再次访问资源时就不用调用底层组件来进行验证或

审查动作了，直接从缓存中验证或审查，提高了安全框架的反应速度。

4.3.1.3 程序性安全的实现

程序性安全提供的编程接口（API）由 EJBContext 接口的两个函数和 HttpServletRequest 接口的两个函数组成：

- boolean isCallerInRole(String roleName)
- Principal getCallerPrincipal()
- boolean isUserInRole(String roleName)
- Principal getUserPrincipal()

这四个接口函数中分为两类：一类是获得当前用户的主体信息，另一类是获得当前用户所对应的角色集合与传入的角色元素之间的所属关系。这两类信息都是由用户程序发起请求，EJBContext 或 HttpServletRequest 分别向各自的资源容器请求获得，第一类信息在资源容器经过图 4.13 的中步骤的(1)至(9)之后（即用户认证之后），其主体信息包含在用户对象 Subject 中，并且用户程序在调用这一类函数之前（也就是在进入用户程序之前，图 4.13 中的所有步骤由资源容器自动执行）；对于第二类信息，资源容器必须询问安全框架管理者获得，而后者从已向其注册的用户、角色映射服务提供者处获得（具体调用的接口方法详见 4.3.2.2）。

4.3.2 表现层设计与实现

表现层主要的任务是安全服务框架的设计与实现，与安全域相关的一系列接口设计（为表现层提供服务）、各个安全服务提供者的一系列接口设计（用于承接实现层），并且还包括了这两部分是如何进行沟通与交互的。

本层的一个主要的组件是域管理器类：SecurityManager，它继承了接口 AuthenticationManager、PermissionCheck 和 RealmMapping，它们之间的关系如下图 4.14 所示：

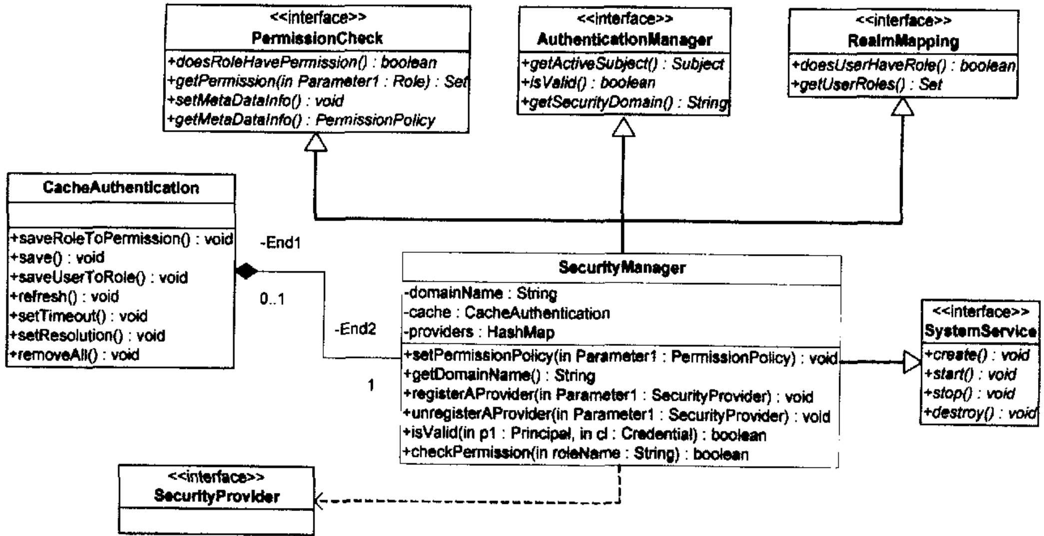

图 4.14 安全框架类及接口图

图中的接口说明如下：

(1) PermissionCheck 接口处理检测某个角色是否具有某个操作的权限；

(2) AuthenticationManager 接口处理对资源请求主体的身份认证功能；

(3) RealmMapping 接口处理主体与用户之间的映射关系和用户与角色之间的映射关系；

(4) SystemService 是当启动安全服务时的系统调用接口用于系统管理，系统通过此接口生成、启动、停止或销毁安全服务；

(5) SecurityManager 类实现了上述 4 个接口，使得 SecurityManager 具有安全处理的所有功能。以上 4 个接口对客户程序（资源容器）是可见的；

(6) CacheAuthentication 类用于缓存经认证后的主体、用户、角色信息，加快了安全检测速度，值得注意的是，缓存的信息只在一段时间内有效，为此在初始化该类的实例时，要调用其方法 setTimeout（）设置超时时间，相关的操作还有：调用方法 setResolution（）设置对缓存信息进行有效性检查的时间间隔，调用方法 removeAll（）可以强行清空缓存的所有内容（在底层数据被修改之后，为使得缓存数据和底层数据保持一致，此操作很有必要）。

4.3.2.1 SecurityManager 的实现

为了适应多个安全服务提供者及多种类型的服务提供者都能够集成到本文提出的安全框架内，实现一个灵活的架构，因此 SecurityManager 内部不包括任何具体的身份认证工作或权限检查工作，而是通过调用其依赖接口 SecurityProvider 来

具体实现，这样只要各个安全服务提供者实现了接口 SecurityProvider 就可以顺利的集成到本系统来。

值得注意的是，SecurityManager 类的方法：registerAProvider(SecurityProvider sp)，unregiesterAProvider(SecurityProvider sp) ，和属性 HashMap providers；这两个方法分别是向 SecurityManager 注册一个安全服务提供者（SecurityProvider），和注销一个已有的安全服务提供者，属性：providers 用于存放已注册了的安全服务提供者对象，其实现代码类似如下所示：

```
// 初始化 providers
HashMap providers = new HashMap();
...
public void registerAProvider(SecurityProvider sp ) {
   if(sp != null){
       providers.add(sp);
   }
   return;
}
public void unregisterAProvider(SecurityProvider sp ) {
   if(sp != null){
       providers.remove(sp);
   }
   return;
}
```

注册一个安全服务提供者可以通过两个方式实现，第一是静态地通过在安全服务的配置文件(security.xml)中预先配置实现，第二是动态地通过安全服务管理接口实现（详见第5章）。安全服务的配置文件格式如图4.15所示：

```
<application-policy
    name="web-console"  <!-- 指定使用配置策略的安全域名称 -->
    >
    <!-- 配置认证服务提供者 -->
    <authentication>
        <!-- 配置第一个认证提供者 -->
        <login-module flag="required"
                    code="com.wplus.appserver.core.infrastructure.security.auth.spi.UsersRolesLoginModule1">
            <!-- 初始化参数设置 -->
            <module-option name="rolesProperties">roles1.properties</module-option>
            <module-option name="usersProperties">users1.properties</module-option>
        </login-module>

        <!-- 配置第二个认证提供者 -->
        <login-module flag="required"
                    code="com.wplus.appserver.core.infrastructure.security.auth.spi.UsersRolesLoginModule2">
            <module-option name="rolesProperties">roles2.properties</module-option>
            <module-option name="usersProperties">users2.properties</module-option>
        </login-module>
    </authentication>


    <!-- 配置权限审查服务提供者 -->
    <permissionCheckProvider code="com.wplus.appserver.core.infrastructure.security.permission.spi.PermissionProvider1">
        <!-- 初始化参数设置 -->
        <module-option name="configFile">c:/wplus/conf/permissionConfig.xml</module-option>
    </permissionCheckProvider>


    <!-- 配置用户、角色映射服务提供者 -->
    <realmMapingProvider code="com.wplus.appserver.core.infrastructure.security.mapping.spi.mysqlRealmMapping">
        <!-- 初始化参数设置 -->
        <module-option name="jndiName">jdbc/realm/mappingDB</module-option>
    </realmMapingProvider>
</application-policy>
```

图 4.15 安全服务提供者配置文件

以下以认证接口 AuthenticationManager 的方法 isValid（Principal p1）为例说明 SecurityManager 如何实现该功能，其他接口方法的实现与此类似：

类似代码如下：

```
public boolean isValid(Principal p1, Credential cl)   throws Exception{
    // 获得已注册的安全服务提供者的列表
    Iterator pders = providers.getInterator();

    // 查找列表是否有合适的提供者可用
    While(pders.hasNext()) {
        SecurityProvider current = (SecurityProvider) pders.next();
        if( current isInstanceof   AuthenticationProvider.class) {
            // 找到对应的服务提供者
            AuthenticationProvider ap = (AuthenticationProvider)current;
            // 调用该服务提供者相应的方法
            boolean result = sp.isValid(p1, cl);
            // 返回结果
            return result;
```

```
        }
    }
    // 没有发现相应的服务提供者，则抛出异常
    throw new Exception("安全服务框架没有发现相应的服务提供者！");
}
```

SecurityManager 是曝露给客户程序（资源容器）使用的，通过对 SecurityManager 类型转换后，使用 SecurityManager 的不同接口，典型代码如下：

```
// 首先获得 SecurityManager 实例 （通过 JNDI 查询，或其他方式）
SecurityManager sm＝ …；
// 获得主体信息
Principal p1＝ …；
Credential c1＝ …；

//把 SecurityManager 转换成 AuthenticationManager 接口，用于处理身份认证
AuthenticationManager am ＝ （AuthenticationManager） sm；
If(sm.isValid(p1, cl)) {
    // 通过认证
    …….
;
}else {
    // 未通过认证
    ……..
}
```

4.3.2.2 安全服务提供者接口设计

安全服务提供者接口的经过泛化后形成了不同类型的安全服务提供者接口，它们的实现类并经过注册到安全框架后被调用以完成相应的安全功能，安全服务提供者接口是安全框架和安全提供者之间的一个契约。这样，通过安全服务提供者接口，使用现有的安全技术或将来开发出来的安全技术都可以集成的该安全框架中。安全服务提供者接口的类图如下所示：

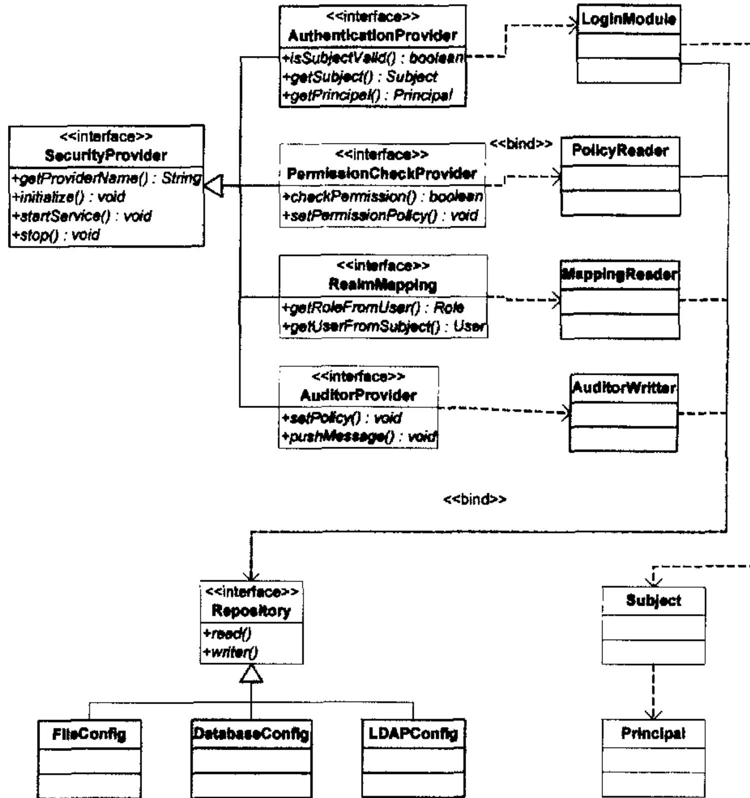

图 4.16 安全服务提供者接口

图中 SecurityProvider 接口为安全服务总接口，安全服务框架类 SecurityManager 通过这个接口调用各个安全服务的实现类对象以完成安全功能，根据提供的安全功能不同，可分为：

(1) 验证服务提供者和登陆模块，为系统提供用户身份验证功能，如果验证对象未登陆系统，则登陆模块提供某种方式的与用户交互的界面，收集用户登陆信息然后再验证用户身份，如果成功验证，将返回一个 Subject 对象，用户表示在本安全域内都认可的一个用户身份；

(2) 权限审查服务提供者，根据以设定的安全访问控制策略(来自于部署描述信息)，对安全敏感资源的访问提供权限审查；

(3) 用户、角色映射器，在 Subject 验证通过后，当它试图访问某些资源时，由角色映射器决定哪些角色作用于此 Subject，并将它们存储于该 Subject 对象中。

(4) 审计服务提供者，它收集安全框架在进行安全性活动时发来的各种消息，审计服务提供者根据配置策略（从策略存储库读取），对消息进行分析并依据当前

策略做出相应的动作。

4.3.3 实现层设计与实现

实现层的主要任务是各个安全服务提供者接口的实现类的设计与实现

4.3.3.1 验证服务提供者和登陆模块

验证服务提供者是为系统提供资源请求者的身份验证，它实现安全服务提供者接口 AuthenticationProvider 接口，以下论述系统的默认实现。

核心类说明如下：

(1) Subject：代表资源请求者实体；

(2) Principal: 代表请求者实体的某一个身份；

(3) Credential: 代表用于证明身份的凭证，如密码、证书等；

(4) Callback：登陆验证类在获取验证信息时使用的参数类型；

(5) CallbackHandler:登陆验证类使用的回调类；

(6) Configuration：JAAS 配置信息库；

(7) LoginContext: 登陆验证环境类；

(8) LonginModule：登陆验证类接口；

(9) SecurityAssociation:存放安全信息（Subject 对象、Principal 对象、Credential 对象）的类，该类是个静态类，并且类内的属性都是 java.lang.ThreadLocal 类型的，所以在同一个线程范围内，该类所含有的信息都有效；

(10) UsersRolesLoginModule：实现了 LonginModule 接口；

(11) DefaultAuthentication：验证服务提供者接口实现类，它实现了 4.3.2 节所设计的安全服务提供者类接口：AuthenticationProvider，类内部的实现采用了 JAAS 技术。

DefaultAuthentication 的直接调用者是某个安全域的 SecurityManager 实例，框架外部或上层接口都是要通过 SecurityManager 才能获取服务提供者的服务，所以 SecurityManager 是 DefaultAuthentication 的客户程序，本节论述中的客户就是指 SecurityManager，整个认证过程如下，（图 4.13 中第 7 步:SecurtiyInterceptor 调用 SecurityManager 的 login()方法是下述过程的起点）：

(1) SecurityManager 查询注册信息，获取验证服务提供者（在本例中是 DefaultAuthentication ），并调用其接口 AuthenticationProvider 的方法 isSubjectValid()；

(2) 程序控制流进入 isSubjectValid()方法内：DefaultAuthentication 调用本地

线程实例 SecurityAssociation 方法：getPrincipal 和 getCredential 方法，获取资源请求者的身份信息：principal 对象及 Credential 对象，并将之存储于新创建的 CallbackHandler 对象中；

(3) 以域名及(2)中的新建的 CallbackHandler 对象为参数，创建 LoginContext JAAS 框架类，LoginContext 查找配置文件（通过 java 运行时环境变量：javax.security.auth.login.config 所指定的文件）并从文件中逐个载入配置在本域名中的各个登陆模块，（本例登陆模块是：UsersRolesLoginModule 类），分别初始化它们（以(2)中新建的 CallbackHandler 对象为参数）；

(4) DefaultAuthentication 调用 CallbackHandler 对象的 login()方法，程序控制流进入 JAAS 框架类内：CallbackHandler 逐个调用(3)步载入的登陆模块，认证过程正真开始，UsersRolesLoginModule 登陆模块从传入的 CallbackHandler 对象参数中获取 principal 对象和 credential 对象，并根据其本身配置的用户名及其机密信息与之比较，进行验证，返回 true 或 false；

(5) 当所有的登陆模块的验证过程完成之后，如果成功验证了，则 JAAS 框架在返回 TURE 之前对所有的登陆模块调用其 commit()方法，该方法将赋予被验证者相应的身份（该身份将在角色映射时使用，使其具有相应的角色），如果验证失败，则框架返回 FALSE；

(6) 程序控制流返回到安全服务框架,获得(5)的返回结果；

(7) 安全服务框架将验证结果返回给 SecurityInterceptor。

整个验证过程的程序控制流走向如果 4.17 所示：

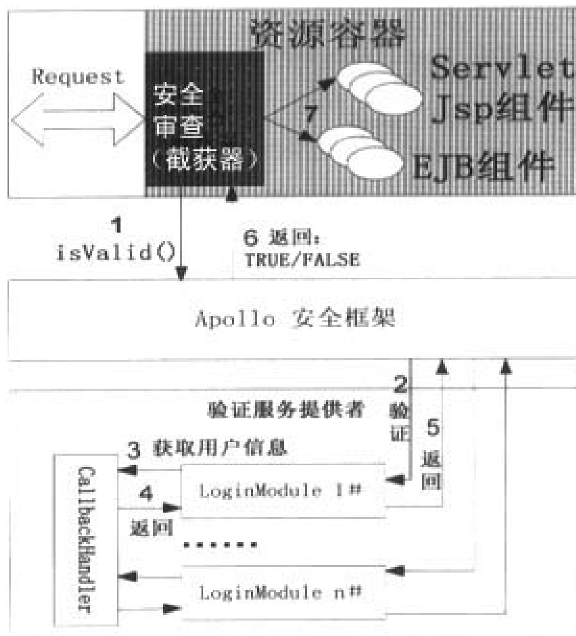

图 4.17 验证过程控制流走向

4.3.3.2 权限审查服务提供者

权限审查服务提供者提供了对资源访问的安全审查，访问控制的作用，主要处理的问题是：在资源请求者经过了身份验证之后，对请求资源执行相关操作之前审查其合法性。以下论述系统的默认实现。

本服务的请求者是资源容器的截获器，直接客户程序是安全服务框架程序（SecurityManager），权限审查的起点是 SecurityManager 调用权限审查服务提供者接口 PermissionCheckProvider 的 checkPermission(Invocation mi)方法,其实现过程的时序图如下图 4.18 所示：

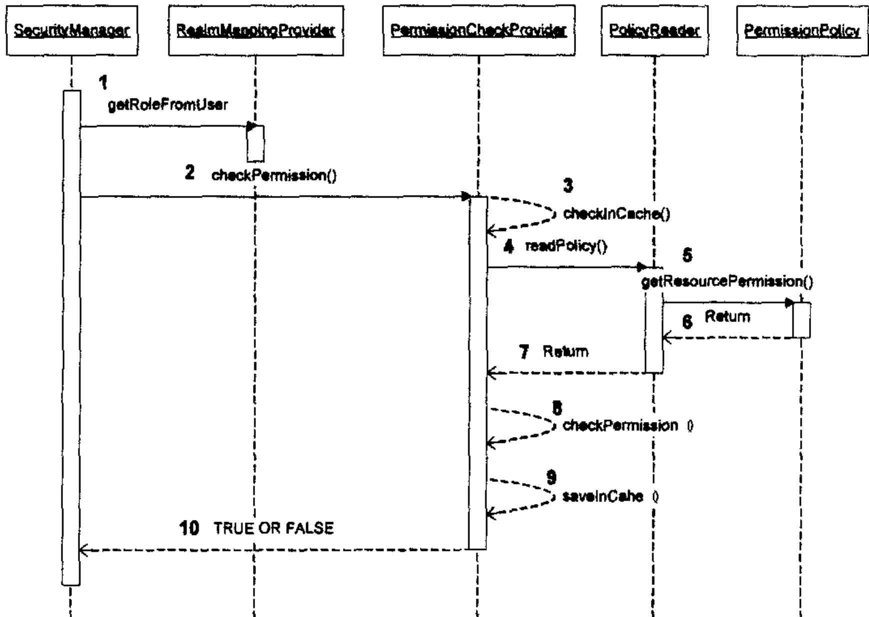

图 4.18 权限审查时序图

当资源请求者经过了身份验证之后，具有了在本安全域内认可的身份标识，安全框架将基于此身份标识进行此后的一系列的访问控制，根据上图权限审查的时序，其过程具体描述如下：

(1) SecurityManager 根据资源请求者的身份标识，调用用户、角色映射服务提供者（RealmMappingProvider）的方法 getRoleFromUser(), 获取请求者对应的

角色；

(2) SecurityManager 根据请求者的所具有的角色及要访问的资源，调用权限审查服务提供者（PermissionCheckProvider）的方法 checkPermission()，以审查请求者的权限；

(3) PermissionCheckProvider 根据 SecurityManager 传入的请求者角色及请求的资源名称，首先在缓存中审核，查看该请求者对该资源的请求及操作是否已被审核过；

(4) 如果在缓存审查失败，PermissionCheckProvider 则根据请求资源的名称调用访问控制策略读取器（PolicyReader）的方法 readPolicy()，以获得访问控制策略；

(5) PolicyReader 进而调用该安全域内的访问控制策略信息包装器（PermissionPolicy）（其策略信息来自于部署描述文件，详见 4.3.1.3 节）；

(6) PermissionPolicy 返回所请求资源的访问控制策略；

(7) 进而，PolicyReader 所请求资源的访问控制策略返回给 PermissionCheckProvider；

(8) PermissionCheckProvider 根据 PolicyReader 返回的访问控制策略对请求者进行审查；并做出 TRUE / FALSE 型决断；

(9) 同时，PermissionCheckProvider 将 PolicyReader 返回的结果及审查结果保存在缓存中，以备下次审查同样资源时可以从缓存中读取访问控制策略，加快审查速度；

(10) PermissionCheckProvider 将(8) 中的决断结果返回给 SecurityManager。

值得注意的是，在权限审核过程中，再次用到了缓存技术，因为安全审查过程本身比较耗费资源，致使服务器的负载加重，延长了对客户端的反映时间，所以缓存的使用对提高服务器的性能具有重要意义。

4.3.3.3 用户、角色映射器

经验证后的用户，通过其用户标识，获得系统分配给予的角色。而角色有两种类型，即整体(global)角色和范围(scoped)角色（或称应用系统逻辑角色）。范围角色只赋给特定的应用程序组件，如 EJB 或 WEB 应用程序。J2EE 部署描述符角色属于范围角色，当 Apollo 安全框架对在同一个角色范围内的对象做出访问控制决定时，才会使用这些范围角色。这一点与整体角色不同，后者在服务器的任何地方都可以用于控制访问决定的参考，不论是应用程序，还是资源。所以，角色可以只在局部范围内有效，但是一个用户标识必须在全局范围内有效。

应用程序部署者或系统管理员有责任，定义范围或整体角色，并把用户身份

或用户组映射到一定的角色上，一个用户可以有多个身份，所以可以有多个与之对应的角色，用户与角色的映射关系是多对多（n:n)。本论文的默认实现如下：

(1) 映射关系的定义：将范围角色和整体角色与用户之间的映射关系存放在数据库的不同的表中，存放范围角色的表以该角色所在的安全域的域名为表名，存放整体角色的表名是 global，表的字段为：用户标识、角色名。

(2) 映射关系的读取：通过用户、角色映射服务提供者接口的实现类：mysqlRealmMapping 从 mysql 数据库读取，具体过程如下：

a) 安全框架根据安全服务提供者配置文件（如图 4.15 所示），首先创建 mysqlRealmMapping 的一个实例，然后调用其初始化函数 intialize 并传入数据库连接对象的 JNDI 名称及本安全域的域名；

b) mysqlRealmMapping 根据传入的参数连接到数据库，并分组查询（group by 角色）相应的范围角色与用户的映射表的所有记录，将属于同一个角色的用户标识保存于同一个 HashSet 变量中，并将该角色与该 HashSet 的 key 到 value 型的映射关系保存在一个名为 scopedRoleUser 的 HashMap 型成员变量中；

c) 查询整体角色与用户映射关系表的所有记录，将属于同一个角色的用户标识保存于同一个 HashSet 变量中，并将该角色与该 HashSet 的 key 到 value 型的映射关系保存在一个名为 globalRoleUser 的 HashMap 型成员变量中；

(3) 服务的使用：映射查询服务首先在安全框架内注册，所以其使用者通过查询安全框架就可以获得，即通过调用安全框架（SecurityManager）的方法 RealmMappingProvider getRealmMappingProvider();

(4) 服务的配置方式：如图 4.15 安全服务提供者配置文件所示，在 security.xml 文件中加入相应的配置项。

4.3.3.4 审计服务提供者

审计服务提供者主要是对审计事件进行分析处理，并形成相应的记录，保留对系统的操作痕迹，操作者包括合法用户和非法用户，对于资源的访问控制相比验证服务提供者和权限审查提供者来说，一次审计成功或失败，并不会影响程序的主要流程的进一步执行，审计只是对系统安全起辅助性的作用，所以由于审计服务本身的特点，在设计审计服务提供者时，应依据以下原则：

(1) 审计成功与否不影响安全框架的逻辑进程；

(2) 框架发送审计事件时，即进行 pushMessage 调用时，该函数应该是非阻塞型；

(3) 审计事件本身应该包括 4.1.1.3 安全模型中审计模型提出的审计事件的所有内容；

(4) 对审计事件分析处理应在一个独立的线程或进程中执行，不应占用系统太多的资源。

根据上述 4 点原则，本论文提出的审计服务提供者默认实现描述如下：

核心类说明如下：

(1) AuditEvent：审计事件类，该类是可序列化的（Serializable），当审计处理程序来不及处理审计事件时，可先把审计事件序列化后存储到文件等介质上，该类包含的属性有：

a) Time：审计事件发生的时间；

b) Principal：引起审计事件发生的主体标识（用户名）；

c) Target：引起审计事件发生的客体（资源名称：如果是某个页面，则是该页面的文件名，如果是 Servlet 或 EJB 组件，则是组件名称加上调用的方法名）；

d) Action：主体对客体的操作类型（GET/PUT/INVOKE）；

e) Result：操作结果（SUCCESS/FAILURE）。

(2) DefaultAuditProvider：审计服务提供者实例类，该类继承了 java.lang.Thread 类，所以具有单独线程启动能力。

(3) AuditWriter：保存审计事件的执行类。

审计过程的时序图如下所示：

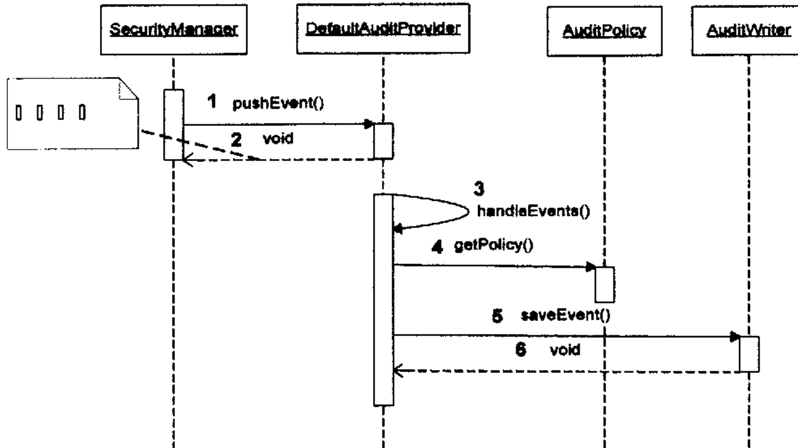

图 4.19 审计过程时序图

上图审计过程说明如下：

(1) 安全框架程序发送审计事件；

(2) 审计服务提供者立即返回；

到此安全框架程序发送审计事件完成，可以继续余下业务逻辑。

(3) 审计服务提供者独立线程处理发送过来的审计事件；

(4) 调用审计策略类，以获得如何处理审计事件的指示；

(5) 保存审计事件；

(6) AuditWriter 保存完数据之后，程序控制流返回到审计服务提供者。

4.3.4 数据层设计与实现

数据层关心的问题是数据的机密性和完整性，以及采用何种技术解决。本论文讨论的数据包括两种类型的数据：一种是存贮在文件或数据库等存贮介质上的静态数据，与此相对，另一种是在分布式环境中的客户端与 Apollo 应用服务器之间实时交互的动态数据。

4.3.4.1 静态数据的保护

因 Apollo 应用服务器涉及的配置文件相当多，如果都对其进行加密保护，则加密解密的操作将非常频繁，必将严重影响系统的性能，所以只对个别机密信息（如，密码等）采取加密保护措施，本论文采用 java2 平台的 JCA 技术，对用户表的密码字段进行加密：

(1) 为保护其机密性，采用对密码加密编码后的秘文方式，而不是明码方式存放在数据库中；

(2) 为保护其完整性，采用消息摘要方式，即在用户表中增加一个密码摘要的字段，用于存放对密码进行编码之后的消息摘要。

在(1) 中采用对称加密算法，加密过程如下（解密过程与此类似）：

(1) 调用 JCA 类 java.security.KeyPairGenerator 的 getInstance 方法获得某种对称加密算法（这里采用 AES）的密钥产生器；

(2) 用该密钥产生器产生一个 128 位的密钥；

(3) 用 JCA 类 javax.crypto. Cipher 的 getInstance 方法获得一个加密器并设定其加密方式和填充方式；

(4) 设定加密器的模式为加密模式；

(5) 用(3) 产生的加密器 和(2) 中产生的密钥对密码信息进行加密。

在(2) 中采用消息摘要方式保护其完整性，形成消息摘要过程如下：

(1) 调用 JCA 类 java.security.MessageDigest 的 getInstance 方法获得某种散列算法（这里采用 SHA-1 的 160 位算法）的消息摘要器；

(2) 调用 messageDigest.update 并输入要摘要的信息，开始产生摘要

(3) 调用 messageDigest.digest 获得摘要信息；

4.3.4.2 动态数据的保护

动态数据主要保护其在网络传输过程中的安全性，本论文通过在传输层使用 SSL（安全套接层）来保护数据在网络中传输时防止被窃取、修改。SSL（Secure Socket Layer）协议使用不对称加密技术实现会话双方之间信息的安全传递。可以实现信息传递的保密性、完整性，并且会话双方能鉴别对方身份。不同于常用的 http 协议，浏览器在与 web 服务器建立 SSL 安全连接时使用 https 协议（即采用 https://ip:port/的方式来访问）。当浏览器与 web 服务器建立 https 连接时，浏览器与 web 服务器之间经过一个握手的过程来完成身份鉴定与密钥交换，从而建立安全连接。具体过程如下：

1) 用户浏览器将其 SSL 版本号、加密设置参数、与 session 有关的数据以及其它一些必要信息发送到服务器。

2) 服务器将其 SSL 版本号、加密设置参数、与 session 有关的数据以及其它一些必要信息发送给浏览器，同时发给浏览器的还有服务器的证书。如果配置服务器的 SSL 需要验证用户身份，还要发出请求要求浏览器提供用户证书。

3) 客户端检查服务器证书，如果检查失败，提示不能建立 SSL 连接。如果成功，那么继续。客户端浏览器为本次会话生成 pre-master secret，并将其用服务器公钥加密后发送给服务器。如果服务器要求鉴别客户身份，客户端还要再对另外一些数据签名后并将其与客户端证书一起发送给服务器。

4) 如果服务器要求鉴别客户身份，则检查签署客户证书的 CA 是否可信。如果不在信任列表中，结束本次会话。如果检查通过，服务器用自己的私钥解密收到的 pre-master secret，并用它通过某些算法生成本次会话的 master secret。

5) 客户端与服务器均使用此 master secret 生成本次会话的会话密钥(对称密钥)。在双方 SSL 握手结束后传递任何消息均使用此会话密钥。这样做的主要原因是对称加密比非对称加密的运算量低一个数量级以上，能够显著提高双方会话时的运算速度。

6) 客户端通知服务器此后发送的消息都使用这个会话密钥进行加密。并通知服务器客户端已经完成本次 SSL 握手。

7) 服务器通知客户端此后发送的消息都使用这个会话密钥进行加密。并通知客户端服务器已经完成本次SSL握手。

8) 本次握手过程结束，会话已经建立。双方使用同一个会话密钥分别对发送以及接受的信息进行加、解密。

4.4 本章小节

本章根据提出的安全参考模型，对安全服务进行详细设计，在提出整个安全框架的前提下，采用分层设计的思想和方法，逐步深入，分别从应用层、表现层、实现层、数据层上进行设计与实现。

第5章 安全服务的管理

本章首先介绍了 JMX 管理模型，接着介绍了 Apollo 应用服务器系统内核管理模型，然后论述了作为 Apollo 应用服务器内核的一部分 Apollo 安全服务是如何实现与系统内核管理模型的无缝接合。

5.1 JMX 概述

JMX(Java Management Extensions) 是J2EE规范中的一部分，是一套标准的代理和服务，是一个可为应用程序植入管理功能的框架，其目的是解决分布式系统的管理问题[31]。它定义了一个管理体系结构、API和在管理规范下的各种管理服务，可以跨越一系列不同的异构操作系统平台、系统体系结构和网络传输协议，开发无缝集成的系统、网络和服务管理应用，在分布式网络环境中充当中间件的作用，并能够平滑地将管理方案应用到已存在的管理体系中。现在逐渐成为管理复杂软件系统的解决方案。

JMX是由Sun和其他几家在网络管理方面处于世界领先地位的公司联合发布的应用Java技术来管理有关Java平台应用程序和其他各种各样的网络资源新型的网络管理规范。JMX继承了Java语言的特性，具有兼容性、灵活性和快速升级能力，将管理技术提到一个新的高度。

5.1.1 JMX 体系结构

JMX 是一个为应用程序植入管理功能的框架。作为一套标准的代理和服务，用户可以在任何 Java 应用程序中使用这些代理和服务实现管理。JMX 规范明确定义了它的管理体系结构和一系列的应用程序编程接口(API)。在 JMX 体系结构中，采用三级分而治之的体系结构化方法来降低可伸缩网络管理的复杂性：装配层(Instrumentation Level)；代理层(Agent Leve)；分布式服务层(Distributed Services Level)。这三个部分之间通过远程方法调用(RMI， Remote Method Invocation)进行通信。 JMX 规范还提供了一套 Java API 用于访问已有的标准管理协议，这套 API 通常称为附加的管理协议 API(Additional Management Protocol APIs)集合。这些协议集合虽然不是 JMX 结构的一部分，但是他们在帮助 JMX 管理网络和与现有的网管系统集成方面起着十分重要的作用。图 5.1 是它的体系结构。

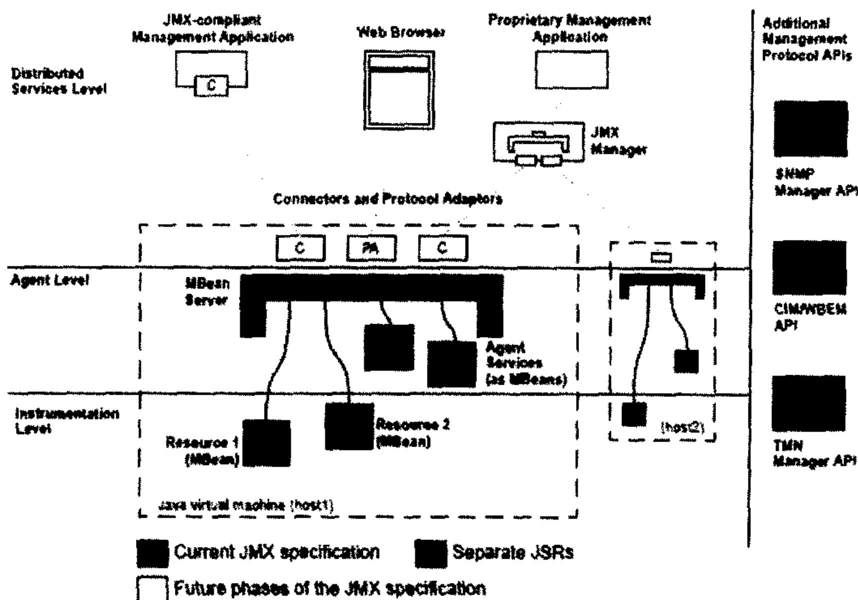

图 5.1 JMX 体系结构(来自于 SUN)

5.1.1.1 装配层

在这一层中，JMX 规范定义了如何用 Java 语言开发网络资源，使这些资源可以接受网络管理，即定义了资源开发的详细规范。

JMX 可管资源可以是一个应用程序，一个服务的实现，一个设备，一个用户等，它是用 Java 语言开发的，或者至少有一个 Java 外壳，并且已经被开发，因此可以被 JMX 兼容的应用程序所管理。

一个给定的装配由一个或多个 Managed Bean(也称作 MBean)提供。MBean 既可以是标准的，也可以是动态的。标准 MBean 是符合特定设计模式的 Java 对象，该设计模式源于 JavaBean 部件模型。动态 MBean 符合特定的接口，该接口在运行时提供了更大的灵活性。资源的装配使它可以通过代理层被管理。MBean 不需要知道 JMX 代理如何操作。MBean 被设计得灵活、简单和易于实现。应用、服务或设备的开发者可以通过一个标准的方式使他们的产品具备可管理性，而不必了解复杂的管理系统。已存在的对象也可容易地调整为标准 MBean 或包装为动态 MBean，这样就可用最小的努力使已存在的资源可管理。另外，装配层还确定了一种通知机制，这样可使 MBean 产生和传播通知事件到其他层的部件。

JMX 可管理资源可被 JMX 兼容的代理自动管理。他们也可被非 JMX 兼容，但应是支持 MBean 设计模式和接口的系统管理。

5.1.1.2 代理层

代理层是定义 MBean 从而对应用程序施加管理的层。该层包括 MBean 服务器和代理服务的定义，还至少包括一个协议适配器或连接器。代理层充当管理决策开发者，提供了实现代理的规范。代理直接控制被管理资源或者使它们能被远程的管理应用来访问。代理和它所控制的资源经常位于同一台 JVM(Java 虚拟机)。这一层利用装置层来管理一个 JMX 可管理资源。一个 JMX 代理包含至少一个管理服务器 MBean Server 和一系列要管理的 MBean，外加至少一个通信连接器或者适配器（connector adapter)。如果实现一个 JMX 规范，那么 MBean Server 和这些服务是规范中要求必须完成的部分。实际上，在 JMX 代理内可通过能够动态地装入和卸装 MBean 的 MBean 服务器(MBean Server)来管理 MBean。访问 MBean 服务器的接口由 JMX 指定。下图是 Agent 组件的结构。

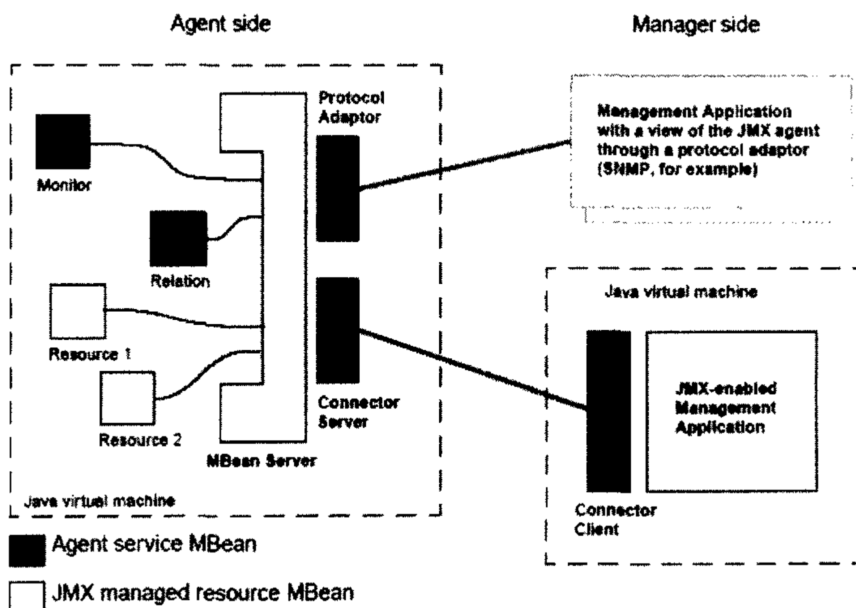

图 5.2 JMX 代理层结构

5.1.1.3 分布式服务层

这一层的目标是为 JMX Manager 组件提供接口。JMX Manager 可以访问代理或代理组来管理由代理公开的 JMX 可管理资源。

分布式服务的细节超出了现阶段的 JMX 规范。这里只是为了提供一个完整的

JMX体系结构视图。

分布式服务层定义了操作代理的管理接口和部件。这些部件可完成以下功能：

- 向管理程序提供接口，以便管理端和代理及其MBeans通过一个连接器透明的通信。
- 将代理及其MBean的视图用一种数据结构更加丰富的协议（如HTML，SNMP）映射出来。
- 将高端管理平台的管理信息向其下众多的JMX代理发布。

收集众多的JMX代理端的管理信息并且根据管理终端用户的需要筛选用户感兴趣的信息并形成逻辑视图送给相应的终端用户。提供安全保证。

5.2 Apollo 内核管理模型

Apollo的内核管理模型基于JMX技术，体现了的面向服务（SOD）的设计思想[32]， Apollo 应用服务器模块化管理各个服务子系统：事务处理服务子系统、日志服务系统、EJB容器子系统、WEB容器子系统、安全服务子系统等等，内核管理模型具有高度可插拔性，即可以在不修改已有源码的条件下替换原有服务模块的实现、增加新的服务或者删除已有的服务，从而使系统具有灵活配置能力，管理模型如下，图5.3所示：

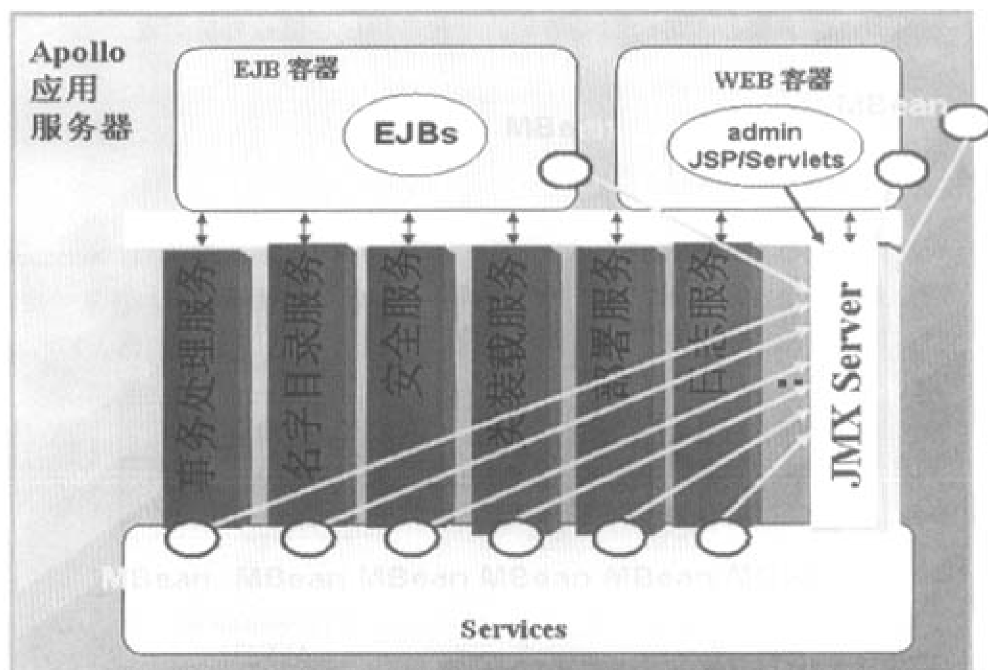

图 5.3 Apollo 管理模型

图中的各个子系统，事务处理子系统、日志子系统、EJB 容器子系统、WEB 容器子系统、安全子系统等通过它们各自对应的 MBean 对象，被 JMX Server 管理，管理的内容包括，通过各自的 MBean 曝露的通用性管理操作：初始化服务、注册服务、启动服务、停止服务、销毁服务等等，和因各个服务都有各自的特性由此曝露在对应的 MBean 接口中的不同管理、配置项。

由于 Apollo 是基于可插拔的内核管理模型，所以其服务子系统都是通过一个配置文件被配置到系统中，内核启动过程简要概述如下：

(1) 在任何服务启动之前，Apollo 引导进程首先创建 MBeanServer 实例；

(2) 接着，MBeanServer 创建管理控制对象 MBeanController 并注册它，前者用来创建服务管理对象（MBean）、后者用来处理（创建、启动、停止、销毁）与前者创建的服务管理对象对应的服务对象，这样两者分工合作启动所有核心服务；

(3) MBeanController 创建服务主部署者对象 MainDeployer 同时创建了相应的服务环境对象，使得 MainDeployer 对象能通过该环境对象与外界信息交流，MainDeployer 用来装载、部署系统的所有核心服务；

(4) MainDeployer 开始部署系统核心服务：读取并解析系统核心服务配置文件；

(5) MBeanController 部署由(4)中读取的各服务配置项，具体的操作包括：创建服务，注册服务、启动服务等等，至此系统核心服务启动完成。

5.2.1 资源管理接口

要把一个资源纳入到 Apollo 的管理模型中，必须要实现接口：ServiceMBean 同时必须把要管理的操作方法暴露给外界，并公开其调用接口，并且一定其接口的名称是：XXXMBean，其中 XXX 是实现该接口的类名称，如图 5.5 所示：

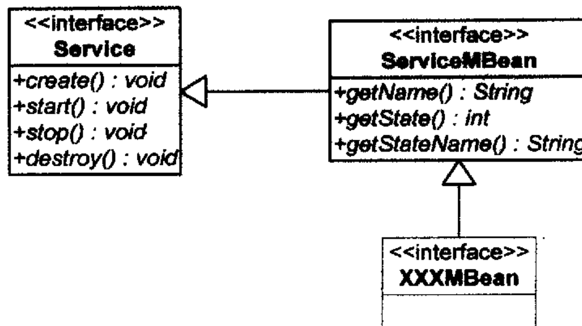

图 5.5 MBean 管理接口

受管理的资源，通过 JMX 代理层对象（如，HtmlAdaptorServlet ，通过网页方式管理资源），可以被外界动态管理该资源。

5.3 安全服务管理的实现

Apollo 安全服务架构最重要的组件是安全域管理者对象（处于表现层详见 4.3.2），安全服务的需求者通过安全域管理者对象获取所需的服务，而安全服务提供者通过向安全域管理者对象登记、注册之后，才能被外界使用，安全域内的各种配置都是通过安全域管理者对象来实现的，所以说，只要能对安全域管理者对象纳入到 Apollo 的管理框架中，也就能够对整个安全域管理了。

根据 5.2.1 节的资源管理接口的要求，本论文设计的安全域的管理接口：SecurityManagerMBean 其类图（图 5.5）如下：

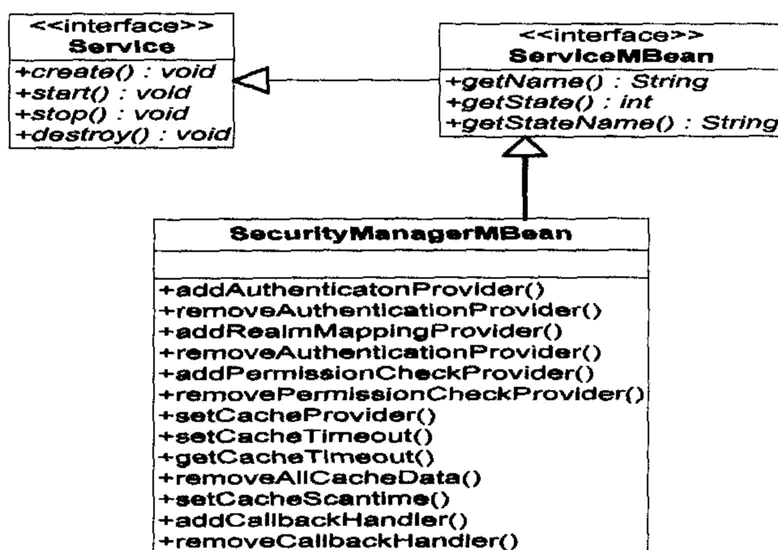

图 5.5 安全域管理接口

通过图中的接口，可被管理、配置的内容有：

(1) 增加或删除验证服务提供者；

(2) 增加或删除用户、角色映射服务提供者；

(3) 增加或删除权限审核服务提供者；

(4) 增加或删除缓存策略对象；

(5) 修改缓存数据的存活时间；

(6) 清除所有缓存时间；

(7) 修改缓存扫描时间。

通过实现上述接口，安全服务就具有被管理的条件，根据 JMX 管理模型本文提出了一个能够对安全域进行本地或远程管理的管理系统，整个系统分成两部分，一部分是客户端 UI 界面，采用 B/S 方式，另一部分是服务器端的代理（ServletProxy）组件，采用了 Servlet 技术，该 ServletProxy 组件在 JMX 模型中处于代理层，它通过 HTTP 协议接受客户端指令数据，对其分析后，通过 MBeanServer 实现对安全服务对象(SecurityManagerMBean)的操作，以达到管理的目的。示意图如下（图 5.6）：

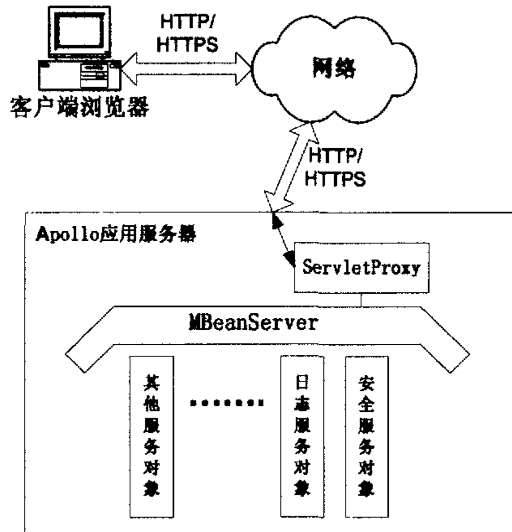

图 5.6 安全管理系统

图中的 ServletProxy 就是管理系统的服务器端的代理组件，它把客户端的管理指令代理给 Apollo 内核管理服务器对象（MBeanServer）完成，并结果返回到浏览器，在客户端与内核管理服务器端充当中间人的角色。

管理系统的核心组件：ServletProxy 的类图如下：

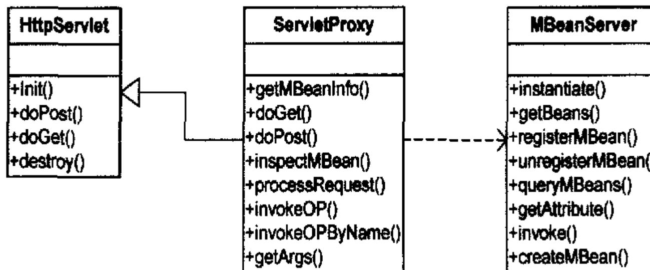

图 5.7 ServletProxy 类图

ServletProxy 类继承了 javax. servlet. http.HttpServlet 类，使它具有了处理浏览器端请求的能力，依据请求的类型（GET 或 POST）doGet()或 doPost() 是它处理网络请求的主入口点函数。

当浏览器发出，查询安全服务的可管理资源请求，或者对安全服务的可管理资源发出操作请求之后（两者仅在传送的参数上有区别），服务器端组件的处理过程如下：

(1) Apollo web 服务器首先收到请求，并把 httpRequest 对象请求转给 ServletProxy 处理；

(2) ServletProxy 调用其 doGet()方法（假设请求使用 GET 类型， POST 类型处理过程与此类似）；

(3) ServletProxy 对客户端传过来的参数进行分析，如果是查询安全服务的可管理资源请求，则执行(4)，如果是对安全服务的可管理资源发出操作请求，则跳至(6)；

(4) ServletProxy 调用其查询方法 inspectMBean()，进而调用 MBeanServer 的 invoke，后者根据传入的参数触发查询操作，获取安全服务管理对象的可管理资源信息并返回；

(5) ServletProxy 将(3)中的查询结果返回给客户端浏览器，处理过程完成；

(6) ServletProxy 根据 httpRequest 请求对象传入的参数，调用 processRequest() 对其请求操作的参数分析后，调用 MBeanServer 的 invoke 方法，后者进而调用安全服务管理对象对其进行操作，并返回操作结果；

(7) ServletProxy 将操作是否成功的结果返回给浏览器。

5.4 本章小节

本章主要围绕对安全框架的管理展开的，在介绍了 Apollo 的管理模型之后，提出了安全服务的管理如何纳入到 Apollo 的管理框架中的解决方案，并论述了具体的设计与实现。

结论

本次毕业设计的课题是关于 J2EE 应用服务器产品中的安全架构的设计与实现，在本论文中，我们完成了以下工作：

(1) 以 J2EE 应用系统设计与实现与其安全功能的设计与实现相分离的思想为出发点，在对 CORBA 安全模型的研究之后，并根据 J2EE 应用服务器规范中的安全需求的基础上，提出了符合我们自己的 J2EE 应用服务器产品安全需求的安全参考模型，以后的设计与实现以此参考模型为指导，并在安全参考模型中，本论文分节详细描述了其子模型的结构及实现要求。

(2) 在对 JAVA 的安全体系及安全技术的研究之后，采用了面向对象的设计思想和设计方法，首先提出了安全架构的并以层次逐步递进和深入的总设计方法对整个安全架构进行层次划分，然后分节论述了安全架构中的各个层的设计与实现。

(3) 安全服务作为 J2EE 应用服务器的一部分，必然要对其进行统一管理，在对 Apollo 的管理架构研究之后，本论文设计实现了安全架构的管理 Bean，使得安全架构的管理能够与整个 Apollo 应用服务器的管理无缝结合。

通过以上的工作，我们初步完成了 J2EE 应用服务器的安全架构的实施，作为一个整体产品的一部分与其他部分成功整合，并试运行，运行结果良好。

目前，虽然安全服务的功能已基本完成，但是在很多方面，仍然做得不够，这也是我们以后的工作，这些方面是：

(1) 提供 GUI 工具支持配置或管理

在本论文的设计中定义了很多 xml 的配置文件，从用户使用的便利性的角度出发，缺少一个统一的 GUI 配置工具及有效的帮助文档或向导帮助用户配置，这些配置包括：用户管理、用户与角色的映射配置、安全框架的初始配置、审计中的审计策略配置等等。

(2) 对安全参考模型的完善与扩展

完善当前安全参考模型中现有的策略，扩展其他策略，如安全框架的缓存策略、不可抵赖服务策略等等。

参考文献

[1] .C.Mohan. "Application servers:Born-Again TP Monitors for the Web", ACM SIGMOD'2002, May 21-24, 2002

[2] Sun Microsystems Inc.,Java. 2 Platform, Enterprise Edition (J2EE.) Specification November 24, 2003 page 23

[3] Sun Microsystems, Inc.Java. 2 Platform, Enterprise Edition (J2EE.) Specification November 24, 2003 page 27

[4] MageLang Institute, Fundamentals of Java Security,http://java.sun.com/developer/onlineTraining/Security/Fundamentals/Security.html

[5] GOSLING, J., JOY, B., AND STEELE, G. The Java Language Specification. Addison-Wesley, Reading, Massachusetts, 1996.

[6] LINDHOLM, T., AND YELLIN, F. The Java Virtual Machine Specification. Addison-Wesley, Reading, Massachusetts, 1996.

[7] http://java.sun.com/docs/books/tutorial/security1.2/overview/index.html

[8] Sun Microsystems, Inc. ,Secure Computing with Java: Now and the Future, http://java.sun.com/security/javaone97-whitepaper.html

[9] Sun Microsystems, Inc., Security layer two: verifying the bytecodes,http://java.sun.com/sfaq/may95/security.html

[10] Sun Microsystems, Inc.,HotJava(tm): The Security Story,http://java.sun.com/sfaq/may95/security.html

[11] Sun Microsystems Inc., JCA http://java.sun.com/products/jca

[12] Sun Microsystems Inc., JCE http:// java.sun.com/products/jce

[13] Sun Microsystems Inc., JSSE http:// java.sun.com/products/jsse

[14] Sun Microsystems Inc., JAAS http:// java.sun.com/products/jaas

[15] 宫力 java 2 平台安全技术－结构、API 设计和实现. 机械工业出版社，2000

[16] Sun Microsystems Inc., http://java.sun.com/developer/technicalArticles/Security/jaasv2/index.html

[17] Abhijit Belapurkar,Java authorization internals,http://www-128.ibm.com/developerworks/library/j-javaauth/index.html,04 May 2004

[18] Kyle Gabhart,Java security with JAAS and JSSE,http://www-128.ibm.com/developerworks/library/j-pj2ee9.html ,25 Nov 2003

[19] Andrew S. Tanenbaum, Distributed operating systems. hentice Hall, Inc.

Englewood Cliffs, 1995 , pp554-563

[20] D.R. Kuhn. Mutual Exclusion of Roles as a Means of Implementing Separation of Duty in Role-Based Access Control Systems. Second ACM Workshop on Role-Based Access Control, 1997.

[21] OMG,Common Object Request Broker Architecture: Core Specification,March 2004 Version 3.0.3

[22] Sun Microsystem Inc.,Java. 2 Platform Enterprise Edition Specification, 11/24/03 v1.4 pp24

[23] Object Management Group. Security Service Specification. Version 1.8. March 2002

[24] Zao, J.; Sanchez, L.; Condell, M.; Lynn, C.; Fredette, M.; Helinek, P.; Krishnan, P.; Jackson, A.; Mankins, D.; Shepard, M.; Kent, S.;DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings Volume 1, 25-27 Jan. 2000 Page(s):41 - 53 vol.1.

[25] 腾猛.分布对象中间件安全关键技术研究，工学博士学位论文，国防科学技术大学.2003 年 10 月.pp19.

[26] Sun Microsystems Inc. J2EE Connector Architecture Specification, Version 1.5, Nov 24, 2003.

[27] Sun Microsystem Inc.,Enterprise JavaBeansTM Specification Version 2.0,August 14, 2001

[28] Sun Microsystem Inc.,Java. Servlet Specification Version 2.3,August 13 2001

[29] 冯玉琳，黄涛，金蓓弘.网络分布计算和软件工程，科学出版社，2003 年 5 月.

[30] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Pattern-Oriented Software Architecture: Patterns for Concurrent and networked Objects. Wiley, 2000.

[31] Sun Microsystems, Inc.,JavaTM 2 Platform, Enterprise Edition Management Specification,June 18, 2002

[32] R.M. Dijkman, Service-Oriented Design, http://wwwhome.cs.utwente.nl/~dijkman/ArCo%20Deliverable%201.1.2.pdf

攻读学位期间发表的学术论文

序号	作者（全体作者，按顺序排列）	题　目	发表或投稿刊物名称、级别	发表的卷期、年月、页码	相当于学位论文的哪一部分（章、节）	被索引收录情况
1	吴国祥，陈立行	J2EE 应用服务器集群中反向代理负载均衡技术的研究与设计	第四届华南理工大学计算机学科学术研讨会论文集	2004.12.		
2	陈立行，吴国祥	动态负载均衡技术在 J2EE 应用服务器集群中的设计与应用	计算机工程			已投稿

致谢

本论文的指导老师是陈立行副教授，在此我衷心的感谢陈老师。在三年研究生学习期间，我在学习和生活中得到了陈老师的严谨细致的指导和无微不至的关心。陈老师深厚的学术造诣、丰富的实践经验、严谨的治学精神和温厚的待人态度给我留下了深刻的印象，让我收益无穷。在本论文的定稿过程中，陈老师细心地审阅，耐心地指导，是我顺利地完成论文的指南和保证。藉此机会向陈老师表示深深的谢意和感激之情。

衷心感谢计算机学院的郭荷清老师、黎绍发老师、姚耀文老师、彭宏老师、溪建清老师、祁明老师、贾德良老师等。在他们的辛勤教育下我得以顺利完成我的学业。

衷心感谢广州中间件研究中心的许洪波教授，他治学作风严谨求实、思维敏捷，在我项目开发期间和论文撰写期间许教授给了我悉心的指导和热情的帮助。

衷心感谢广州中间件研究中心所有的老师、同学和同事们：吴晓超、朱宇东、李振鹏、荀晓林、温丽芳、刘丽娟、刘佳、孙学钧、洪君生、戴天、李程雄、梁英宏、徐浩等在我实习期间和项目开发期间给予的帮助，与你们起一去食堂、共进午餐及在来回路上的欢声笑语将是一段美好的回忆。

衷心感谢生活在我身边的同学和朋友：胡耀民、杨春、刘伯豪、张佳、张传坤、王亮明、黄奕、罗凌、贾西平、余明举等在生活上、学习上给予的帮助。

衷心感谢我的家人，感谢他们多年来给予我的支持和关怀，感谢他们对我的殷切希望和给予我的无限动力。