

[19] 中华人民共和国国家知识产权局

[51] Int. Cl⁷

G06F 15/16

G06F 9/46



[12] 发明专利申请公开说明书

[21] 申请号 200410032699.6

[43] 公开日 2005 年 3 月 30 日

[11] 公开号 CN 1601510A

[22] 申请日 2004.3.5

[74] 专利代理机构 上海专利商标事务所

[21] 申请号 200410032699.6

代理人 李家麟

[30] 优先权

[32] 2003. 3. 6 [33] US [31] 60/452, 736

[32] 2004. 2. 26 [33] US [31] 10/789,440

[71] 申请人 微软公司

地址 美国华盛顿州

[72] 发明人 T·布朗 C·D·查斯

B·塔巴拉 K·格里利施

G·C·胡恩特 A·托罗尼

A·希德里 D·诺布尔

R·V·维拉恩德 G·奥施雷德

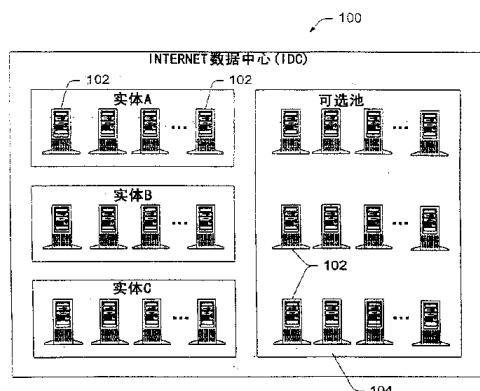
G·R·佩特森

权利要求书 3 页 说明书 184 页 附图 93 页

[54] 发明名称 分布式计算系统的架构和分布式应用程序的自动设计，部署及管理

[57] 摘要

一种设计工具，包括一服务定义模型，能够抽象的描述分布式计算系统和分布式应用程序。该设计工具还包括一模式，指示在该服务定义模型中功能性操作如何被规定。该功能性操作包括分布式应用程序设计，分布式应用程序部署，以及分布式应用程序管理。



1. 一种设计工具，包括：

—服务定义模型，能够抽象的描述分布式计算系统和分布式应用程序；以

5 及

—模式，指示在该服务定义模型中功能性操作如何被指定，其中该功能性操作包括分布式应用程序设计，分布式应用程序的部署，以及分布式应用程序管理。

2. 一种方法包括：

10 容纳一个分布式系统的实现，该分布式系统使用一服务定义模型（SDM）来架构；

公开一组API，用于操作—SDM类型；以及
跟踪用于分布式应用程序部署和管理的SDM实例。

15 3. 如权利要求2所述的方法，进一步包括跟踪用于设计时间验证的SDM实
例。

4. 如权利要求2所述的方法，进一步包括在该分布式系统上设计分布式应
用程序。

5. 如权利要求2所述的方法，进一步包括使用该服务定义模型，在该分
布式系统上设计分布式应用程序。

20 6. 一种方法，包括，通过根据一服务定义模型描述该物理数据中心，使一
物理数据中心的一个比例不变虚拟数据中心代表的设计变得容易，其中，该服
务定义模型包括与设计分布式应用程序关联的服务，与部署分布式应用程序关
联的服务，以及与管理分布式应用程序关联的服务。

25 7. 如权利要求6所述的方法，其中该虚拟数据中心由一第一数据中心和一
第二数据中心支持，以便一应用程序可以被部署到该第一数据中心或者该第二
数据中心里，而不需要求该应用程序部分和该虚拟数据中心部分之间的该逻
辑模型映射进行改变。

8. 如权利要求6所述的方法，进一步包括：

确定要为一该物理数据中心的特定逻辑组件创建的实例数量；以及

30 确定一连接线拓扑，以便实现该物理数据中心。

9. 如权利要求6所述的方法，进一步包括自动的部署物理资源和一分布式应用程序。

10. 如权利要求6所述的方法，进一步包括，使用一基于模型的管理方法来管理一在该物理数据中心上的分布式应用程序。

5 11. 如权利要求6所述的方法，其中该物理数据中心包括一操作引擎，用于配合响应一触发事件的一系列操作任务。

12. 如权利要求11所述的方法，其中该触发事件用来自该服务定义模型的应用程序上下文来注释的。

10 13. 如权利要求11所述的方法，其中该系列的操作任务包括使用一资源管

理系统来部署一个新的资源。

14. 一种方法，包括：

易于一虚拟数据中心和一分布式应用程序的设计；

逻辑放置该分布式应用程序的部分到虚拟数据中心上；以及

基于该虚拟数据中心实现一物理数据中心。

15 15. 如权利要求14所述的方法，其中当逻辑放置该分布式应用程序的部分到虚拟数据中心中时，就限制该虚拟数据中心的一操作者视图，从而只包括那些与该分布式应用程序的布局有关的部件。

16. 如权利要求14所述的方法，其中该虚拟数据中心包括大量服务器定义模型层，以便每一层被逻辑的放置在其下一层的上面。

20 17. 如权利要求14所述的方法，进一步包括分配该虚拟数据中心的资源，从而支持该分布式应用程序。

18. 一种方法，包括：

易于一虚拟数据中心和一分布式应用程序的设计；

逻辑放置该分布式应用程序的部分到虚拟数据中心上；以及

25 确定该分布式应用程序部分的布局是否是有效的。

19. 如权利要求18所述的方法，进一步包括，如果该分布式应用程序部分的布局不是有效的，就产生一警告信息。

20. 如权利要求18所述的方法，进一步包括基于该虚拟数据中心实现一物理数据中心。

30 21. 如权利要求18所述的方法，进一步包括，分配该虚拟数据中心的资源，从

而支持该分布式应用程序。

22. 一种软件架构，用于在设计，部署和管理一分布式计算系统上的分布式应用程序，该软件架构包括：

5 一第一软件层，用于将机器转换成在该分布式计算系统中所使用的服务器所使用的工具；

一第二软件层，用于分布式应用程序的网络管理和虚拟拓扑的生成；

一第三软件层，保存该分布式计算系统的一物理模型；

一第四软件层，易于该分布式应用程序所请求的逻辑资源的分配；

10 一第五软件层，用于一服务定义模型（SDM），该服务定义模型，用于描述操作处理的名字空间和上下文，并且提供一API，用于应用程序自检和应用程序资源的控制；以及

一第六软件层，定义一分布式应用程序的可重用构造块，该定义使用SDM API来上下文，命名和绑定；以及

一第七软件层，用于该分布式应用程序的操作管理。

15 23. 一种资源管理器，包括：

装置，用于在一个分布式计算系统中分配资源；

装置，用于发现可用的硬件；

装置，用于处理资源分配请求；

装置，用于跟踪资源的所有权；

20 装置，用于控制资源；以及

装置，用于在该分布式计算系统中为所有资源管理提供一通用API。

24. 如权利要求23所述的资源管理器，其中用于分配的资源的装置被配置，从而在该分布式计算系统中分配逻辑资源和物理资源。

25 25. 如权利要求23所述的资源管理器，进一步包括装置，用于管理在该分布式计算系统上的一分布式应用程序。

26. 如权利要求23所述的资源管理器，其中用于控制资源的装置被配置，从而配合响应事件一系列的操作任务。

分布式计算系统的架构和分布式 应用程序的自动设计,部署及管理

5 相关技术

本发明专利申请要求享有2003年3月6日提交的美国临时申请No.60/452,736的利益，其公开内容在这里被结合以作为参考。

本发明专利申请也涉及随后的美国专利申请（所有该美国专利申请在这里被结合以作为参考）：

10 2003年3月6日提交的美国专利申请系列NO.10/382,942，标题为“虚拟网络拓扑产生（Virtual Network Topology Generation）”；

2000年10月24日提交的美国专利申请系列NO.09/695,812，标题为“共享计算机的分布式管理的系统和方法（System and Method for Distributed Management of Shared Computers）”；

15 2000年10月24日提交的美国专利申请系列NO.09/695,813，标题为“分布式计算机系统的逻辑模拟的系统和方法（System and Method for Logical Modeling of Distributed Computer Systems）”；

20 2000年10月24日提交的美国专利申请系列NO.09/695,820，标题为“分布式计算机限制数据传输和管理软件组件的系统和方法（System and Method for Restricting Data Transfers and Managing Software Components of Distributed Computers）”；

25 2000年10月24日提交的美国专利申请系列NO.09/695,821，标题为“使用包过滤器和网络虚拟化来限制网络通信（Using Packet Filters and Network Virtualization to Restrict Network Communications）”；

2000年10月24日提交的美国专利申请系列NO.09/696,707，标题为“设计一个分布式计算机系统的逻辑模型和根据所述逻辑模型部署物理资源的系统和方法（System and Method for Designing a Logical Model of Distributed Computer System and Deploying Physical Resources According to the Logical Model）”；和

30 2000年10月24日提交的美国专利申请系列NO.09/696,752，标题为“在一个多机服务应用程序中提供自动策略执行的系统和方法（System and Method

Providing Automatic Policy Enforcement in a Multi-Computer Service Application) ”。

技术领域

本发明涉及一种分布式计算系统的架构和在所述分布式计算系统上分布式应用程序的自动设计，部署和管理。

5 背景技术

Internet应用程序在过去的几年里已经得到了迅速发展并且继续增长。人们已经变得习惯在万维网（或简称为“Web”）上提供的许多服务，例如电子邮件，在线购物，收集新闻和信息，听音乐，浏览视频剪辑，找工作，等等。为了跟上基于Internet的服务的日益增长的需求，在专用于容纳网站的计算机系统中发生了巨大的发展，为那些网站提供后端服务，并且存储与这些网站相关联的数据。
10

一分布式计算机系统的类型是Internet数据中心（IDC），该中心是一特殊设计的联合体，其中容纳了许多用于容纳基于Internet的服务的计算机。IDC，也被称为“网站群”和“服务器群”，典型的容纳了在气候控制的、物理安全建筑中的
15 成百到上千台计算机。这些计算机相互连接来运行一个或更多支持一个或更多Internet服务或网站的程序。IDC提供可靠的Internet访问，可靠的电源和一个安全的操作环境。
15

图1示出了一Internet数据中心100。它具有布置在一个特殊构造的房间里的许多服务器计算机102。这些计算机都是通用计算机，典型地被配置为服务器。
20 一个Internet数据中心可能被构造从而为单一的实体容纳单一的站点（例如，Yahoo数据中心或MSN），或者从而为多实体提供多个站点（例如，为多公司容纳站点的Exodus中心）。
20

三个共享计算机资源的实体—实体A，实体B，和实体C说明了IDC 100。这些实体表示了希望在网上出席的各个公司。所述IDC 100有一个附加计算机
25 104的池，该池可能由所述实体在通信业务繁忙时使用。例如，一个从事在线销售的实体可能在圣诞节期间经历较大的需求增长。所述附加的计算机提供IDC灵活性来满足这个需求。
25

现今，大规模的IDC是复杂的，并且经常被要求容纳多个应用程序调用。
30 例如，一些站点可能操作几千台计算机，并且容纳了很多分布式应用程序。这些分布式应用程序经常具有复杂的网络需求，这些需求需要操作者来物理连接
30

这些计算机到确定网络转换器，以及手动调整在IDC中的布线配置来支持复杂操作。结果，这种建造物理网络拓扑来确定这些应用程序需求的任务可能是倾向于人为错误的，麻烦的、耗时的过程。因此，需要用来在所述物理计算系统上设计和部署分布式应用程序的改进技术。

5 发明概述

一种在分布式计算系统上用于设计、部署和管理一种分布式应用程序的架构和方法被描述。

附图的简要说明

整个附图使用相似的引用标记来注明同样的成分和/或特征。

10 图1说明了Internet数据中心的一个例子。

图2说明了服务的一个例子。

图3-8说明了示例抽象层。

图9-10说明了一示例SDM类型空间。

图11-15说明了示例层抽象。

15 图16说明了一示例过程。

图17-19说明了在这里讨论的示例组件。

图20-21说明了一示例图形用户接口。

图22说明了一示例SDM模型。

图23说明了一示例部署。

20 图24说明了示例类型。

图25说明了示例实例请求。

图26说明了示例约束的重新生效。

图27说明了一SDM运行时间的示例逻辑架构。

图28说明了一示例服务图形表示。

25 图29说明了示例实例空间。

图30说明了打包数据到SDU的一例子。

图31说明了示例类型空间，成员空间和实例空间。

图32说明了一示例成员树。

图33说明了一示例实例树。

30 图34说明了在这里所描述的系统的示例实现。

- 图35说明了跟踪创建组件实例的例子。
- 图36-39说明了示例组件实例事件。
- 图40说明了分区运行时间的一个例子。
- 图41说明了一示例成员空间。
- 5 图42说明了一示例实例层次。
- 图43说明了分区一实例空间的例子。
- 图44说明了在各种组件之间的示例关系。
- 图45说明了一示例固定身份的信任关系。
- 图46-47说明了一示例组件的布置。
- 10 图48说明了一示例平台架构。
- 图49说明了用于应用程序部署的示例使用流程。
- 图50说明了一示例应用程序设置和宿主设置。
- 图51说明了用于部署工具的示例阶段。
- 图52说明了示例可视化数据中心描述。
- 15 图53-54说明了示例流程图。
- 图55说明了一处理SDU的例子。
- 图56-58说明了示例流程图。
- 图59说明了一示例模型架构。
- 图60说明了示例的管理层。
- 20 图61说明了一示例的系统操作。
- 图62说明了一示例连接管理。
- 图63-67说明了一示例的设备物理配置。
- 图68说明了一示例请求图。
- 图69说明了一示例答复图。
- 25 图70-86说明了一示例的本发明可能被使用的情况。
- 图87说明了一示例服务平台架构。
- 图88说明了在系统中的示例组件。
- 图89说明了一在这里所描述的系统中可能包括的产品示例。
- 图90说明了各种资源管理组件。
- 30 图91说明了一多局域网（LAN）的布局示例。

- 图92说明了一示例ADS架构。
- 图93说明了一示例ADS远程引导和图形系统。
- 图94说明了一示例拓扑布置。
- 图95说明了一SDML示例。
- 5 图96说明了在一SDU中数据的收集示例。
- 图97说明了一使用SDM运行时间API进行动态绑定的例子。
- 图98说明了一示例SDM布置。
- 图99说明了一示例部署。
- 图100说明了一示例系统架构。
- 10 图101说明了多种部署层的例子。
- 图102说明了示例操作逻辑。
- 图103-105说明由于Internet的示例改变。
- 图106说明了一示例应用程序生命周期。
- 图107说明了一新的体系结构的优点示例。
- 15 图108说明了一转换复杂系统到简单图表的例子。
- 图109说明了一示例服务。
- 图110说明了一示例SQL族。
- 图111说明了一示例SDM数据中心模型。
- 图112说明了一示例设计应用程序表面。
- 20 图113说明了一在数据中心的示例SDM服务。
- 图114说明了示例资源管理器。
- 图115说明了一资源虚拟的例子。
- 图116说明了示例程序操作逻辑。
- 图117说明了与操作逻辑交互的示例。
- 25 图118-119说明了一管理杂散的环境的例子。
- 详细描述
- 下面的公开（disclosure）描述了与架构有关的多个方面，该架构用于设计和实现一个具有大规模应用程序服务的分布式计算系统的。该公开包括一种服务定义模型（SDM）和一种SDM运行时间环境的讨论。该公开进一步包括设计方面，例如如何模拟数据中心组件，如何模拟一个分布式应用程序描述，和用

于逻辑的将一个模块化的应用程序布置到一个模块化的数据中心上，并且在设计时间里使该逻辑布局生效的技术。该公开进一步解释了部署方面，例如如何使用物理资源和在物理资源上的分布式应用程序的物理布局来示例所述模型，从而促进在物理数据中心的应用程序部署。该公开还介绍管理方面，包括使用所述SDM来提供上下文的管理反馈，跟踪，和操作反馈。该公开讨论了多种（across）在涉及物理资源的应用程序的部署中所使用，并支持所述管理方面的资源管理器。

服务定义模型（SDM）

所述服务定义模型（SDM）为应用程序架构师提供工具和上下文，从而以一种抽象方式来设计分布式计算机应用程序和数据中心。该模型定义了一组表示了所述应用程序的功能单元元素，该应用程序最终被物理计算机资源和软件实现。与所述模型元素相关联的是一模式（schema），其规定了如何由所指定的组件表示功能性操作。

SDM回顾

15 介绍

Internet时代

20 在过去的十年我们亲眼看见了Internet作为一个计算平台出现。越来越多的软件公司采用“软件作为服务”模型。这些服务典型的包括运行在多个机器上的几个组件，该机器包括服务器，网络配置和其它专用的硬件。松散耦合的，异步的程序设计模型成为标准。可缩放性，有效性和可靠性对这些分布式服务的成功是决定性的。

我们也亲眼看到了在硬件发展方向上的改变。高密度服务器和专用的网络硬件在数据中心是普遍的。转换结构正在取代系统总线并且正在系统配置里提供更大的灵活性。相对于训练和一个专门的维护操作的职员的成本，硬件成本现在在所有权的全部成本（Total Cost of Ownership, TCO）度量（metric）中扮演一个小的角色。虽然对于任何高效的服务来说，稳定的（rock-solid）操作方法（practice）是必须的，因为由人们执行人工过程产生的不可靠性，这些方法难以始终如一的重复。在这个新兴的以软件作为服务的时代，开发的焦点正在从桌面向服务器转变。随着这一焦点的改变，给软件开发者，硬件设备供应商，和IT专业人员带来了过多的新问题：

- 服务是大型的并且更复杂---服务对于开发来说是耗时的，对于维护来说是困难和成本高，并且对于扩充附加功能来说是冒险的。
- 服务趋向于单一---服务趋向于依靠定制组件和特殊配置。许多服务的多个部分不能被单独的删除，升级，或者选择性的替换而不影响所述服务的有效性。
5 ■ 服务依靠特殊的硬件配置---无论它是一个确定的网络拓扑或者在特殊网络设备上的从属，在硬件和软件之间的绑定极大的减少了在不同的数据中心环境中容纳服务的能力。
- 服务需要操作上的一致性---大多数的服务需要一个操作人员亲自来运行。
10 通用的平台的缺少降低了在服务上重用代码和制定操作最佳方法（practice）的能力。不幸的，操作人员必须在每个服务的细节处被培训，并且在每个服务发展时再培训。

贯穿本文档，术语“服务”和“应用程序”被交替使用。通常，一个应用程序可以被看作是一个分布式服务的集合。例如，Hotmail可以是一个包括多服务的应用程序，其中每个服务执行不同的功能。
15

这些问题与桌面和DOS时代的那些问题没有不同（大约19世纪80年代）。DOS为应用程序开发者定义了有价值的核心服务，例如磁盘管理，文件系统，控制台工具，等等。然而，它确实留下了很多复杂的任务给独立软件开发商们（ISV）来决定。例如，为了在它们各自的应用程序中支持打印，WordPerfect
20 和Lotus123都必须独自的写打印机驱动。同样的，打印机硬件供应商为了有一个成功的产品，必须与这些软件公司达成协议。这对独立软件开发商们和硬件供应商们来说，要进入的所述屏障格外高。这导致了在这个时代只有少数成功的软件和硬件公司。

微软公司（Microsoft）通过创造Windows平台解决这个问题，该平台显著的减少了进入的屏障。Windows为在PC平台上的多数硬件设备定义一抽象层。
25 这使ISV免于担心支持特殊的硬件设备。Windows管理在包括存储器，磁盘和网络的PC上的所有的资源。Windows也将能够带来被独立软件开发商（ISV）们利用的附加服务的财富。这个平台在工业中激发了巨大的发展。把Windows平台作为目标的独立软件开发商（ISV）们是多产的。因为具有一个通用的平台Windows
30 的商品化影响，许多新硬件供应商开始提供更便宜的硬件。

服务定义模型 (SDM)

SDM基本原理

所述SDM:

- 定义抽象体 (abstraction) 来使设计分布式应用程序/服务更容易。
- 5 ■ 允许使用复用的框架和操作方法 (practice) 的自动化。
- 简化分布式应用程序和服务的部署和操作。

可以更容易理解SDM是什么，通过认为设想它捕捉 (capture) 的是现在经常被看作一个在一服务操作者附近的墙上的复杂图。在这些图中，一个框典型的表示服务的一个运行元素，连接这些框的线表示所述服务元素之间的通信路径。例如，一个连接某些IIS前端机器的负载平衡器被依次连接到一个或更多个中间或后端服务上。
10

另一种方式来考虑所述SDM是它既是一个分布式应用程序/服务的行为的元模型又是一个在它的计算环境中正在运行的应用程序/服务的“活的”蓝图。所述SDM以一种说明性的和不变的比例的方式捕获在它的计算环境中的应用程序的结构，包括它的容许的软件操作。说明性的描述一个服务的拓扑的能力是很有效的，该拓扑包括在硬件和网络资源之间的绑定，和它的软件组件的有效操作。
15

以此类推，让我们看微软的组件对象模型 (COM)。COM标准化了组件是如何打包，注册，激活，发现，等等。COM要求与生命周期，存储器管理和接口实现相关的严格的规则。这些原语对互操作性是必需的因为它们允许组件被像黑盒子一样对待。COM是更复杂的服务诸如事件 (eventing)，自动化和对象链接和嵌入 (OLE) 的基础。
20

同样的，COM需要定义一些基本的要素，在这些要素上来建立更复杂的能力。这些原语是：

- 25 ● 组件—实现，部署和管理的单元。
- 端口—命名为终点，具有一个关联类型和一系列的有效操作。
- 连接线—端口之间的允许通信路径。
- 层—资源管理器所有权和绑定的分离。
- 映射—在每一层的组件，端口和连接线之间绑定。

30 本文档的剩余部分将更加详细的描述这些原语 (primitives) 中的每一个。

组件，端口和连接线

就本文档来说，设想使用组件、端口和连接线所绘制的，名为MyService的简单服务的一个图形表示是有用的。参见图2。在该图中，框表示组件，菱形表示端口，虚线表示连接线。

5 ● MyService是一个复合的组件，因为它使用了组件MyFrontEnd和MybackEnd。

● MyService具有一个被称为web的可视端口，该端口是一个由组件MyFrontend

实现的授权端口。

10 ● MyFrontEnd具有两个端口，授权端口和一个由目录标注的端口。

● MybackEnd具有一个由数据标注的端口。

● MyFrontEnd和MybackEnd组件具有一个潜在的连接关系，该连接使用一条连接线绑定目录端口到数据端口。

组件

15 组件是实现、部署和管理的单元。组件的例子是一个运行Windows Server，一个IIS虚拟网站或一个SQL数据库的专用服务器。组件通常具有机器边界，但是不是必须像由一台单一的IIS服务器上所包含的网络服务那样被证实。

组件通过端口的公开了(expose)功能性，并通过连接线进行通信。单一的组件只能具有端口作成员。使用其它组件的组件被被认为是复合组件，该复合组件除了其它组件以外可以具有端口和连接线作为成员。

复合组件通过组合创建，不具有任何与它们相关联的设备。复合组件端口是来自内部组件的授权的端口。复合组件使配置，封装和复用成为可能，从而可以被认为是一种有机化一个应用程序/服务和它们的性能的方法。

25 只有组件的公共端口在组件外面是可视的。复合组件对外界来说看上去像单一组件，由于它们使用通过封装进行掩藏的所述复合组件的内部结构。事实上，一个单一组件可以被一个复合组件替换，反之亦然，只要(as along as)它们两者支持的端口类型和性能是完全相同的。

端口

30 名为终点的端口定义了一系列的性能。端口具有关联类型或任务，典型的关联一系列允许的操作。端口的例子是一个HTTP服务器端口，一个具有一系列

允许操作的SOAP端口，等等。端口可以被授权（delegate），这意味着一个外部的组件可以像自己的端口一样公开（expose）一个内部组件的端口。

端口形成到一个组件的公共接口（性能）。端口是一个可以被公开（可视的）组件的唯一的成员。

5 连接线

连接线是在多个端口之间的允许绑定并且表示多个端口（和多个组件）之间的拓扑关系。连接线不指定任何实例互联拓扑，但是代替的一个实例互联拓扑表述为“潜在”（potentiality）。

10 连接线是必需的总线，能够包括一个或多个端口成员。连接线不应该被误认为是一个点到点关系。一个给出的端口不能在同一条连接线中出现超过两次。

模式

15 为了描述一个应用程序/服务，由于必需为所述SDM提供一个标准模式。所述SDM模式应该使用XSD和XML语法表述。尽管详细的描述SDM的模式超出了本文档的范围，必须为在本文档后面描述的主题内容提供某些简短的描述作为上下文。下面是所述SDM模式的简单视图（view）。

```
<sdm>
<identityReference/>
<portClasses/>
<wireClasses/>
20 <componentClasses/>
<hostRelations/>
<PortTypes/>
<WireTypes/>
<componentTypes/>
25 </sdm>
```

请在<http://big/>阅读SDM模式祥述并且回顾范例XSD文件，来得到关于SDM设计的更多详细的信息。

SDM 类

30 在一个应用程序/服务中的每个组件，端口和连接线是通过类的使用创造的类型。新类型能够从现有的类和类型中被创建。一个SDM类本质上是一个通用

特征的抽象。例如，Web Service可以被模拟为一个可以是SQL数据库的类。在所述MyService应用程序中，MyFrontEnd可以是一个从Web Service类中派生出来的新类型，MyBackEnd可以是一个从SQL数据库中派生出来的一个新类型。

下面是一个端口，连接线和组件的类模式的例子。

```
<portClass name="ServerDataAccess" layer="Application">
<settingSchema>
<xs: element name="databaseName" type="xs: string"/>
</settingSchema>
</portClass>
```

5

```
<wireClass name="DataConnection" layer="Application">
<settingSchema>
<xs: element name="useSSL" type="xs: boolean"/>
</settingSchema>
<portClassesAllowed>
<portClassRef name="ServerDataAccess" maxOccurs="1"/>
<portClassRef name="ClientDataAccess"/>
</portClassesAllowed>
</wireClass>
```

```
<componentClass name="Database" layer="Application">
<deploymentSchema>
<xs: element name="sqlScriptFilePath" type="xs: string"
maxOccurs="unbounded"/>
</deploymentSchema>
<settingSchema>
```

```
<xs: element name="databaseName" type="xs: string"/>
</settingSchema>
<portClassesAllowed closed="true">
<portClassRef name="ServerDataAccess"/>
</portClassesAllowed>
</componentClass>
```

注意每个componentClass和wireClass模式可以包含一个设置模式，部署模式，
和允许的端口类。portClass不具有一个端口类容许段。这些模式被如下定义：

- 安装模式（Setting Schema）是XSD用于在组件，端口和连接线上的配置参数，这些组件，端口和连接线可以是设计时间有效。
- 部署模式（Deployment Schema）是XSD，表述为了依次使组件，端口和连接线被安装，哪些安装参数需要被设置。这些清单可以是Fusion或某些其它安装技术模式。
- 所允许端口类（Port Classes Allowed）是组件和连接线通过参考所说明的端口类说明允许的端口。

请参考在<http://big/>上的SDM 设计详细信息（Schema Design Specification），以
得到所述类模式的更多详细的信息。

类关系

可以容纳其他组件的组件，端口和连接线使用一个宿主关系（hostRelation）
模式被说明，该宿主关系模式标识了它可以容纳的安装和组件类。可以把宿主
关系元素看为一个在类之间的直接链接，其中组件，端口或连接线之一可以作
为其他的一个宿主。

容纳一个组件意味着为一个组件代码提供执行环境。例如，如在下面的例子所示，SQL可以是Database类的组件的一个宿主。

```
<hostRelations>
<installer name="DatabaseInstaller" codeType="InstallerPlugIn"/>
<hostRelation classRef=" database"
componentHostClassRef="host:SQL" installerRef="DatabaseInstaller"
/>
</hostRelations>
```

SDM类型

这里是SDM模拟的3个截然不同的空间：资源，应用程序和实例。实例空间在本文本档后面被描述。资源空间是类存在和构件，在该构件中应用程序被构建。所述应用程序空间是类型驻留的空间。下面是一个用于端口，连接线和组件类型的XML的一个例子。

```
<portType name="UserDataServer" class=" ServerDataAccess">
<deployment/>
<settings/>
</portType>
```

```
<wireType name="UserData" class="DataConnection">
<deployment/>
<settings>
<useSSL>false</useSSL>
</settings>
<portTypeRefs>
<portTypeRef name="UserDataServer"/>
<portTypeRef name="UserDataClient" />
</portTypeRefs>
</wireType>
```

```
<componentType name="SQLBackEnd" class="Database">
<deployment>
<sqlScriptFilePath>%install%\mydatabaseDfn.sql</sqlScriptFilePath>
</deployment>
<settings>
<databaseName>UserData</databaseName>
</settings>
<ports>
<port name="userData" type="UserDataServer"/>
</ports>
</componentType>
```

注意在所述SDM模式中的每个端口类型（portType），连接线类型（wireType）和组件类型（component Type）包括安装和部署值。

- 设置值是XML用于设置的模式，它为组件，端口和连接线提供配置值，
而且可以是设计时间有效的。
- 部署是XML用于部署清单（manifest），它表述了为了使所述组件，端口或连接线被正确安装，需要被设置的配置参数的值。

请参照在<http://big>上的SDM模式（schema）设计详细描述（SDM Schema Design Specification）得到类型的更多详细信息。

复合组件

复合组件可以被用来定义一个应用程序和它到其他组件，端口和连接的拓
5 扑关系。复合组件不具有一个关联实现，代替的使用端口和宿主关系的授权来公开（expose）成员组件和端口的性能。下面的XML示出了复合组件MyService 使用SDM可能如何被描述。

```
<compoundComponentType name="MyService">
<components>
<component name="MyFrontEnd" type="IISFrontEnd"/>
<component name="MyBackEnd" type="SQLBackEnd"/>
</components>
<wires>
<wire name="data" type="UserData">
<members>
<member componentName="MyFrontEnd"
portName="serverData"/>
<member componentName="MyBackEnd"
portName="userData"/>
</members>
</wire>
</wires>
</compoundComponentType>
```

实例

尽管组件，端口和连接线定义了一个应用程序/服务的结构和性能，它们并
10 没有定义运行实例。每个组件，端口和连接线类型说明可以有一个或多个实例。实例是部署一个应用程序/服务的结果，以便物理资源（服务器，网络转换端口和磁盘）被分配，软件资源（操作系统，运行时间宿主，应用程序代码）被安装

和配置。从创建时到它们被删除的时跟踪所有的实例是SDM运行时间的工作。

SDM运行时间 (SDM Runtime)

SDM运行时间并不自己创建组件，端口和连接线的实例，相反的，它提供一系列的应用程序编程接口 (API)，用来协同进行SDM实例的创建和管理。

- 5 一个实例的实际的创建，例如一个运行用IIS作为一个对网络服务组件的宿主的
Windows Server的服务器，将典型的包括多实体并且可能使用很多小时或好几天
来完成。

所述SDM运行时间知道什么时候“创建SDM实例 (create SDM instance) ”
过程开始和什么时候它伴随成功或者失败终止。所述SDM运行时间也知道一个
10 SDM实例在它的生命周期期间发生了什么改变。一种考虑SDM运行时间的方式
是它是一个计算装置 (accountant)，记录了所有的与一个给出的应用程序/服务
SDM有关的处理，以便它可以被查询所述特定SDM有关的实例信息。

创建一个SDM实例的第一步是注册一个具有SDM运行时间的应用程序/服务SDM。
一旦所述SDM运行时间知道 (know about) 一个给出的SDM，使用多个工厂 (factory)
15 和多个资源管理器 (Resource Manager)，所述实例创建进程可以 (在后面解释) 被调用。

请阅读在<http://big/>上的SDM运行时间架构 (SDM Runtime Architecture) 详
述来获得所述API和运行时间设计的更多细节信息。

宿主和工厂

20 可以“容纳”其它组件的组件被称为宿主，它为它们支持的类担当工厂。一
个组件可以使用先前所描述的SDM模式hostRelation元素被说明为一个或多个组
件类的宿主。

尽管宿主为一个组件的代码提供执行环境，工厂是实际的服务，它创建一
个给出类型的SDM实例，并且通过SDM运行时间API与所述SDM运行时间相互
25 作用。工厂可以支持一个或多个组件类，并且必须通过所述SDM运行时间注册，该
运行时间规定它们所支持的组件类。

对于一个给出的工厂来说，支持具有不同配置的同种类型的多个宿主是可
能的，就象对于单独的工厂来说存在宿主配置的每个类型已经是可能的一样。

例如，一个IIS Factory可以支持复合类，例如Web Service和Web Application。同
30 样地，所述SQL Factory可以支持不同数据库类型，例如Database，Partitioned

Database和Highly Available Database。

工厂并不自己管理物理资源，例如存储器，网络和服务器。工厂通过多个资源管理器（Resource Manager）与多个物理资源（和它们的逻辑等价物）相互作用。

5 资源管理器

资源管理器管理物理和逻辑资源，它们是（1）作为一个引导指令进程的一部分被发现或创建或（2）通过某些物理环境的公布的基于XML的描述来指定。资源管理器拥有所有的存储器，网络和服务器资源，并公开一个通用资源管理器API来执行资源分配请求并跟踪这些资源的所有者。

10 资源管理器的例子是NRM（网络资源管理器），SRM（存储资源管理器），和PRM（PC资源管理器）。这些资源管理器的每一个负责物理端口或磁盘或服务器，和它们所公开的逻辑资源，例如虚拟局域网（VLAN），逻辑磁盘册（volume），共享文件，web服务器等等的分配。资源管理器也负责编程这些物理设备来实现分配或存储单元分配。

15 为了编程这些物理硬件，资源管理器通过资源提供者与所述硬件相互作用，该资源提供者隐藏了所述硬件设备的实现细节，从而，例如，来自多个供应方的网络转换器可以被用来交换（由现存的制造者的设备的提供者给出）。像在Windows中的硬件抽象层（HAL）和设备驱动模型，一个数据中心环境的等价的抽象层，所述环境跨越服务器，网络和存储设备。

20 层和映射

尽管当它们结合宿主，工厂，资源管理器和所述SDM运行时间时，组件，端口和连接线是有效的抽象，它们并不足够来部署和管理一个分布式应用程序/服务。为了创建和管理这些逻辑抽象的物理实例，一些附加结构是需要的。这些附加结构是层和映射。

25 层

对层的需要是由执行一个应用程序/服务的部署需求的来实现设计时间有效的期望所引起的。图3示出了由所述SDM定义的抽象层。

● 应用程序层描述了所述分布式组件，它们的部署需求和约束，和它们在一个应用程序/服务上下文中的通信关系。

30 ● 宿主层描述了为例如IIS，CLR和SQL，其它中的一个宿主进行的配置和

策略设置和约束。

- 虚拟数据中心（VDC）层描述了自操作系统，经由网络拓扑到服务器，网络和存储装置的数据中心环境设置和约束。
- 硬件层描述了物理数据中心环境，并使用例如XML以说明方式被发现或指定。这一层不是比例不变的并且因此在SDM中不被模拟，但是为了完整被包括了。

映射

因为所述SDM是分层的，需要一种在各个层之间进行绑定的方法。映射即，在一层的一组件或端口到在其下面邻近一层上的一组件或端口的绑定是必需的。一个映射可以被如下描述：

$$MT = [T_n \rightarrow T_{n-1}] + [T_{n-1} \rightarrow T_{n-2}] + [T_{n-2} \rightarrow T_{n-3}] + \dots$$

这里M表示一个映射，T表示一个组件，端口或者连接线，n表示层。箭头符号表示映射的方向，所述映射总是从一个较高层到一个较低层。

例如，在图4中，在应用程序层的名为MyFrontEnd的组件被映射到一个在宿主层的被称为IIS的组件。同样名为MyBackEnd的组件被映射到一个在宿主层的SQL组件。

设计-时间生效 (Design-time Validation)

一个组件和在下层的宿主组件之间的绑定能够在所述应用程序/服务在活数据中心被实际部署前显露问题给开发者。这些问题可以是因为不相容的类型，配置冲突，不匹配操作，丢失拓扑关系，等等。图5描述了一个设置和约束，其检测在一个组件和它的宿主之间验证错误。

在图6中，在下面的图表中尝试映射可能会引起一个错误，因为在部署层的IIS和SQL组件之间没有可能的通信关系。

尽管从MyBackEnd组件到SQL宿主组件的映射基于所述组件和宿主类型兼容性和配置冲突的缺少已经是一个有效的绑定，但因为在MyFrontEnd到MyBackEnd之间的拓扑关系所定义的所述MyService SDM了一个不存在于一特殊部署层，它是无效的。

设置和约束检测

从应用程序层到部署层（等等）的映射能力是很有效的，因为它使一组件的设计时间相对于一个宿主约束的生效变的可能，并且也允许一个宿主

的设置相对于一个组件的约束生效。

图7示出了一个在不同层的组件和宿主之间的关系的更详细的图。注意在一层的一个组件和在下面相邻一层的一个宿主组件自始到终到所述VDC层有一个绑定。

5 在图7中，MyFrontEnd是一个由IIS容纳的网络服务（Web Service），该IIS依次是由Windows服务器容纳的Windows应用程序（Windows Application）。有一个IIS工厂，该工厂支持网络服务和一个网络应用程序组件实例的创建和删除，就像有一个Windows应用程序工厂相应的负责创建和删除IIS和SQL的实例。

10 图8示出了设计时间如何使用过去描述的SDM设置和约束语义在不同层的组件之间工作。

注意使在上面的层的组件的约束相对于一个在下面层的宿主组件的设置是有效的。也注意使在宿主组件的约束相对于被拥有的组件的设置是有效的。

这一双向设置和约束检测允许一个开发者自始至终在使用SDM语义所描述的操作环境的上下文中可靠地开发他/她的应用程序/服务。为了描述一个数据15 中心，以便它的描述可以在开发过程被依靠，必需创建一个被称为VDC的数据中心的抽象。

虚拟数据中心（VDC）

一个VDC是一个物理数据中心环境的逻辑表示，它简化了开发者的数据中心的视图（view）。理想地一个IT工程师或架构师应该能够以同样的不变的20 比例的方式来描述该数据中心，该相同的方式就是一个开发者可以描述一个分布式应用程序/服务。这种考虑VDC的方式是它是在数据中心的服务器，网络和存储资源以及它们的拓扑关系的抽象。一个典型的数据中心框图是相当复杂的，具有多相互连接的多个服务器，网络设备，多个IP地址，多个虚拟局域网（VLAN），多个操作系统，存储器，等等，全部被描述在一个使用Visio或一个简单的工具25 绘制的简单的图表中。除了这种框图之外，经常有长的文档精确地描述所述数据中心是如何分区，配置和管理的。

这种复杂事物的一个例子是微软系统架构（MSA，Microsoft System Architecture）企业数据中心（EDC，Enterprise Data Center）。应该是显而易见的，如果不是不可能的任务的话，随着时间的推移，由于所应用的更新和升级，将30 手工绘制的图表和文档保持在数据中心的当前状态，变得浪费。同样地，使环

境相对于文档指示生效的能力是困难的并且倾向于人为错误。

对开发者和对IT工程师两者来说，以一种不变比例的方式表示一个诸如所述MSA EDC复杂的数据中心将会非常有效。使用组件，端口和连接线来描述一个数据中心的能力提供一种有效的框架，从而在该框架中模拟和生效部署需求，该
5 模拟和生效在现在的设计和部署过程中是缺少的。

SDM原理

所述SDM：

- 定义抽象体（abstraction）来使设计分布式应用程序/服务更容易。
- 允许使用复用的框架和操作方法（practice）的自动化。
- 简化分布式应用程序和服务的部署和操作。

可以更容易理解SDM是什么，通过认为设想它捕捉（capture）的是现在经常被看作一个在一服务操作者附近的墙上的复杂图。在这些图中，一个框典型的表示服务的一个运行元素，连接这些框的线表示所述服务元素之间的通信路径。例如，一个连接某些IIS前端机器的负载平衡器被依次连接到一个或更多个
10 中间或后端服务上。
15

另一种方式来考虑所述SDM是它既是一个分布式应用程序/服务的行为的元模型又是一个在它的计算环境中正在运行的应用程序/服务的“活的”蓝图。所述SDM以一种说明性的和不变的比例的方式捕获在它的计算环境中的应用程序的结构，包括它的容许的软件操作。说明性的描述一个服务的拓扑的能力是很有效的，该拓扑包括在硬件和网络资源之间的绑定，和它的软件组件的有效操作。
20

以此类推，让我们看微软的组件对象模型（COM）。COM标准化了组件是如何打包，注册，激活，发现，等等。COM要求与生命周期，存储器管理和接口实现相关的严格的规则。这些原语对互操作性是必需的因为它们允许组件被像黑盒子一样对待。COM是更复杂的服务诸如事件（eventing），自动化和对象链接和嵌入（OLE）的基础。
25

同样的，COM需要定义一些基本的要素，在这些要素上来建立更复杂的能力。这些原语是：

- 组件—实现，部署和管理的单元。
- 端口—命名为终点，具有一个关联类型和一系列的有效操作。

- 连接线—端口之间的允许通信路径。
- 层—资源管理器所有权和绑定的分离。
- 映射—在每一层的组件，端口和连接线之间绑定。

本文档的剩余部分将更加详细的描述这些原语（primitives）中的每一个。

5 组件，端口和连接线

就本文档来说，设想使用组件、端口和连接线所绘制的，名为MyService的简单服务的一个图形表示是有用的。在图2中，框表示组件，菱形表示端口，虚线表示连接线。

● MyService是一个复合的组件，因为它使用了组件MyFrontEnd和
10 MybackEnd。

● MyService具有一个被称为web的可视端口，该端口是一个由组件
MyFrontend

实现的授权端口。

● MyFrontEnd具有两个端口，授权端口和一个由目录标注的端口。

15 ● MybackEnd具有一个由数据标注的端口。

● MyFrontEnd和MybackEnd组件具有一个潜在的连接关系，该连接使用
一条连接线绑定目录端口到数据端口。

组件

20 组件是实现、部署和管理的单元。组件的例子是一个运行Windows Server，
一个IIS虚拟网站或一个SQL数据库的专用服务器。组件通常具有机器边界，但
是不是必须像由一台单一的IIS服务器上所包含的网络服务那样被证实。

组件通过端口的公开了（expose）功能性，并通过连接线进行通信。单一
的组件只能具有端口作成员。使用其它组件的组件被被认为是复合组件，该复
合组件除了其它组件以外可以具有端口和连接线作为成员。

25 复合组件通过组合创建，不具有任何与它们相关联的设备。复合组件端口
是来自内部组件的被授权的端口。复合组件使配置，封装和复用成为可能，从
而可以被认为是一种有机化一个应用程序/服务和它们的性能的方法。

只有组件的公共端口在组件外面是可视的。复合组件对外界来说看上去像
单一组件，由于它们使用通过封装进行掩藏的所述复合组件的内部结构。事实
30 上，一个单一组件可以被一个复合组件替换，反之亦然，只要（as along as）它

们两者支持的端口类型和性能是完全相同的。

端口

名为终点的端口定义了一系列的性能。端口具有关联类型或任务，典型的关联一系列允许的操作。端口的例子是一个HTTP服务器端口，一个具有一系列允许操作的SOAP端口，等等。端口可以被授权（delegate），这意味着一个外部的组件可以像自己的端口一样公开（expose）一个内部组件的端口。

端口形成到一个组件的公共接口（性能）。端口是一个可以被公开（可视的）组件的唯一的成员。

连接线

连接线是在多个端口之间的允许绑定并且表示多个端口（和多个组件）之间的拓扑关系。连接线不指定任何实例互联拓扑，但是代替的一个实例互联拓扑表述为“潜在”（potentially）。

连接线是必需的总线，能够包括一个或多个端口成员。连接线不应该被误认为是一个点到点关系。一个给出的端口不能在同一条连接线中出现超过两次。

模式

为了描述一个应用程序/服务，由于必需为所述SDM提供一个标准模式。所述SDM模式应该使用XSD和XML语法表述。尽管详细的描述SDM的模式超出了本文档的范围，必须为在本文档后面描述的主题内容提供某些简短的描述作为上下文。下面是所述SDM模式的简单视图（view）。

```
20   <sdm>
    <import/>
    <identityReference/>
    <information/>
    <portImplementation Type/>
    <wireImplementation Type/>
    <componentImplementation Type/>
    <hostRelations/>
    <PortTypes/>
    <WireTypes/>
30   <componentTypes/>
```

</sdm>

请在<http://big/>阅读SDM模式祥述并且回顾范例XSD文件，来得到关于SDM设计的更多详细的信息。

类型

5 在一个应用程序/服务中使用的每个组件，端口和连接线是一个类型。类型是在面向对象的语言中如C++和C#中与类实质是等价的，与类相似，新类型可以被从现有类型中被创建。所述不变比例空间在所述SDM设计中用端口类型，
10 连接类型和组件类型表示。不变比例意味着一个组件，端口或连接线在一个应用程序/服务SDM中可以被表示一次，即使在实际的数据中心中每一个可能会有多个实例。

15 一个类型被最终被从一个实现类型中派生，该实现类型实质上是普通技术特征的抽象。例如，Web Service可以被模拟像SQL数据库可以被模拟的一样的实现类型。在所述MyService应用程序中，MyFrontEnd可能是一个从实现类型Web Service中派生的新类型，MyBackEnd可能是一个从实现类型SQL Database中派生的新类型。

每个组件实现类型（componentImplementationType）和连接线实现类型（wireImplementationType）SDM模式元素可以包括一设置模式，部署清单和端口实现参考。所述端口执行类型元素不具有一个端口实现参考。图9说明了所述SDM实现类型空间的看上去像什么。

20 ● 设置模式（Settings Schema）是XSD用于在设计时间有效的组件，端口和连接线上配置参数。

● 配置清单（Deployment Manifest）是XSD表述为了组件，端口或连接线被安装所需要设置的安装参数。该清单可能是为Fusion或一些其它安装技术的模式。

25 ● 端口实现参考（Port Implementation Reference）是组件和连接线通过参考所说明的端口实现类型说明的允许端口。

另外，一个可以容纳其它组件的组件被使用宿主关系（hostRelation）SDM模式元素说明，该模式元素标识它可以容纳的安装程序和组件实现类型。一种可以认为宿主关系（hostRelation）元素是在组件实现类型之间的直接链接，其中这些组件中的一个作为其它组件（们）的宿主。容纳一个组件意味着为组件
30

的代码提供执行环境。例如，IIS是一个执行类型Web Service和Web Application组件的宿主。在本文档的后面将会有更加详细的解释宿主。

每个在SDM模式中的端口类型（portType），连接线类型（wireType）和组件类型（componentType）元素包含应用程序约束值，部署值和宿主约束值。

5 另外，所述连接线类型元素包括一个端口类型元素，该端口类型元素定义了在所指定连接线类型上的允许的端口类型，组件类型元素包括一个容纳类型列表元素，该元素识别那些可以在所指定组件类型上被容纳的实现类型。图10示出了这种SDM类型空间。

● 10 设置值（Setting Value）是设置模式的XML为组件，端口和连接线提供的配置值，并且它可以是相对于宿主的约束值设计时间有效的。

● 部署值（Deployment Value）是部署清单的XML，表述了为了使所述组件，端口或连接线正确安装所必须设置的配置参数的值。

● 约束值（Constraints Value）是设置模式的XML，提供宿主的组件，端口或连接线所必须设置的配置参数值。约束值可以是相对于下面的宿主的设置值设计时间有效的。

● 15 端口类型（Port Type）是XML列出允许的端口类型，它可以是所指定连接线的一个成员。

● 容纳类型列表（Hosted Type List）是XML，其中组件说明了它所能够容纳的组件实现类型的列表。

20 实例

尽管组件，端口和连接线定义了一个应用程序/服务的结构和性能，它们并没有定义运行实例。每个组件，端口和连接线类型说明可以有一个或更多实例。实例是部署一个应用程序/服务的结果，以便物理资源（服务器，网络转换器端口和磁盘）被分配，和软件资源（操作系统，运行时间宿主，应用程序代码）被安装和配置。

25 从实例创建时到它们被删除时跟踪所有的实例是SDM运行时间（SDM Runtime）的工作。

SDM运行时间

SDM运行时间并不自己创建组件，端口和连接线的实例，相反的，它提供30 一系列的应用程序编程接口（API），用来协同进行SDM实例的创建和管理。

一个实例的实际的创建，例如一个运行用IIS作为一个对网络服务组件的宿主的Windows Server的服务器，将典型的包括多实体并且可能使用很多小时或好几天来完成。

所述SDM运行时间知道什么时候“创建SDM实例（create SDM instance）”
5 过程开始和什么时候它伴随成功或者失败终止。所述SDM运行时间也知道一个SDM实例在它的生命周期期间发生了什么改变。一种考虑SDM运行时间的方式是它是一个计算装置（accountant），记录了所有的与一个给出的应用程序/服务SDM有关的处理，以便它可以被查询所述特定SDM有关的实例信息。

10 创建一个SDM实例的第一步是注册一个具有SDM运行时间的应用程序/服务SDM。一旦所述SDM运行时间知道（know about）一个给出的SDM，使用多个工厂（factory）和多个资源管理器（Resource Manager），所述实例创建进程可以（在后面解释）被调用。

请阅读在<http://big/>上的SDM运行时间架构（SDM Runtime Architecture）详述来获得所述API和运行时间设计的更多细节信息。

15 宿主和工厂

可以“容纳”其它组件的组件被称为宿主，它为它们支持的类担当工厂。一个组件可以使用先前所描述的SDM模式hostRelation元素被说明为一个或多个组件类的宿主。

20 尽管宿主为一个组件的代码提供执行环境，工厂是实际的服务，它创建一个给出类型的SDM实例，并且通过SDM运行时间API与所述SDM运行时间相互作用。工厂可以支持一个或多个组件类，并且必须通过所述SDM运行时间注册，该运行时间规定它们所支持的组件类。

对于一个给出的工厂来说，支持具有不同配置的同种类型的多个宿主是可能的，就象对于单独的工厂来说存在宿主配置的每个类型已经是可能的一样。
25 例如，一个IIS Factory可以支持复合类，例如Web Service和Web Application。同样地，所述SQL Factory可以支持不同数据库类型，例如Database，Partitioned Database和Highly Available Database。

30 工厂并不自己管理物理资源，例如存储器，网络和服务器。工厂通过多个资源管理器（Resource Manager）与多个物理资源（和它们的逻辑等价物）相互作用。

资源管理器

资源管理器管理物理和逻辑资源，它们是（1）作为一个引导指令进程的一部分被发现或创建或（2）通过某些物理环境的公布的基于XML的描述来指定。资源管理器拥有所有的存储器，网络和服务器资源，并公开一个通用资源管理器API来执行资源分配请求并跟踪这些资源的所有者。

资源管理器的例子是NRM（网络资源管理器），SRM（存储资源管理器），和PRM（PC资源管理器）。这些资源管理器的每一个负责物理端口或磁盘或服务器，和它们所公开的逻辑资源，例如虚拟局域网（VLAN），逻辑磁盘册（volume），共享文件，web服务器等等的分配。资源管理器也负责编程这些物理设备来实现分配或存储单元分配。

为了编程这些物理硬件，资源管理器通过资源提供者与所述硬件相互作用，该资源提供者隐藏了所述硬件设备的实现细节，从而，例如，来自多个供应方的网络转换器可以被用来交换（由现存的制造者的设备的提供者给出）。像在Windows中的硬件抽象层（HAL）和设备驱动模型，一个数据中心环境的等价的抽象层，所述环境跨越服务器，网络和存储设备。

层和映射

尽管当它们结合宿主，工厂，资源管理器和所述SDM运行时间时，组件，端口和连接线是有效的抽象，它们并不足够来部署和管理一个分布式应用程序/服务。为了创建和管理这些逻辑抽象的物理实例，一些附加结构是需要的。这些附加结构是层和映射。

层

对层的需要是由执行一个应用程序/服务的部署需求的来实现设计时间有效的期望所引起的。图11示出了由所述SDM定义的抽象层。

- 应用程序层描述了所述分布式组件，它们的部署需求和约束，和它们在一个应用程序/服务上下文中的通信关系。
- 宿主层描述了为例如IIS，CLR和SQL，其它中的一个宿主进行的配置和策略设置和约束。
- 虚拟数据中心（VDC）层描述了自操作系统，经由网络拓扑到服务器，网络和存储装置的数据中心环境设置和约束。
- 硬件层描述了物理数据中心环境，并使用例如XML以说明方式被发现或

指定。这一层不是比例不变的并且因此在SDM中不被模拟，但是为了完整被包括了。

映射

因为所述SDM是分层的，需要一种在各个层之间进行绑定的方式。映射即，在一层的一组件或端口到在其下面邻近一层上的一组件或端口的绑定是必需的。一个映射可以被如下描述：

$$MT = [T_n \rightarrow T_{n-1}] + [T_{n-1} \rightarrow T_{n-2}] + [T_{n-2} \rightarrow T_{n-3}] \dots$$

这里M表示一个映射，T表示一个组件，端口或者连接线，n表示层。箭头符号表示映射的方向，所述映射总是从一个较高层到一个较低层。

例如，在图12中，在应用程序层的名为MyFrontEnd的组件被映射到一个在部署层的被称为IIS的组件。同样名为MyBackEnd的组件被映射到一个在部署层的SQL组件。

设计--时间有效

一个组件和在下层的宿主组件之间的绑定能够在所述应用程序/服务在活
15 数据中心被实际部署前显露问题给开发者。这些问题可以是因为不相容的类型，
配置冲突，不匹配操作，丢失拓扑关系，等等。例如，图13中描述的试图映射
将会导致一个错误，因为在所述IIS和在部署层的SQL组件之间没有可能的通信
关系。

尽管从MyBackEnd组件到SQL宿主组件的映射基于所述组件和宿主类型兼容性和配置冲突的缺少已经是一个有效的绑定，但因为在MyFrontEnd到
20 MyBackEnd之间的拓扑关系所定义的所述MyService SDM了一个不存在于一特殊部署层，它是无效的。

设置和约束检测

从应用程序层到配置层（等等）的映射能力是很有效的，因为它使一个组
25 件设置的设计时间相对一个宿主的约束有效，并且也允许一个宿主的设置相对于组件的约束有效。

图14示出了一个在不同层的组件和宿主之间的关系的更详细的图。注意在一
层的一个组件和在下面相邻一层的一个宿主组件自始到终到所述VDC层有一
个绑定。

30 在图14中，MyFrontEnd是一个由IIS容纳的网络服务（Web Service），该IIS

依次是由Windows服务器（Windows Server）容纳的Windows应用程序（Windows Application）。有一个IIS工厂，该工厂支持网络服务和一个网络应用程序组件实例的创建和删除，就像有一个Windows应用程序工厂相应的负责创建和删除IIS和SQL的实例。

5 图15示出了设计时间生效如何使用以前描述的SDM设置和约束语义在不同层的组件之间工作。

注意使在上面的层的组件的约束相对于一个在下面层的宿主组件的设置是有效的。也注意使在宿主组件的约束相对于被拥有的组件的设置是有效的。

10 这一双向设置和约束检测允许一个开发者自始至终在使用SDM语义所描述的操作环境的上下文中可靠地开发他/她的应用程序/服务。为了描述一个数据
中心，以便它的描述可以在开发过程被依靠，必需创建一个被称为VDC的数据
中心的抽象。

虚拟数据中心（VDC）

15 一个VDC是一个物理数据中心环境的逻辑表示，它简单化了开发者的数据
中心的视图（view）。理想地一个IT工程师或架构师应该能够以同样的不变的
比例的方式来描述该数据中心，该相同的方式就是一个开发者可以描述一个分
布式应用程序/服务。从而一个开发者可以描述一个分布式应用程序/服务。这种
考虑VDC的方式是它是在数据中心的服务器，网络和存储资源以及它们的拓
扑关系的抽象。一个典型的数据中心框图是相当复杂的，具有多相互连接的多
20 个服务器，网络设备，多个IP地址，多个虚拟局域网（VLAN），多个操作系统，
存储器，等等，全部被描述在一个使用Visio或一个简单的工具绘制的简单的图
表中。除了这种框图之外，经常有长的文档精确地描述所述数据中心是如何分
区，配置和管理的。

25 这种复杂事物的一个例子是微软系统架构（MSA，Microsoft System
Architecture）企业数据中心（EDC，Enterprise Data Center）。应该是显而易见的，
如果不是不可能的任务的话，随着时间的推移，由于所应用的更新和升级，
将手工绘制的图表和文档保持在数据中心的当前状态，变得浪费。同样地，使
环境相对于文档指示生效的能力是困难的并且倾向于人为错误。

30 对开发者和对IT工程师两者来说，以一种不变比例的方式表示一个诸如所
述MSA EDC复杂的数据中心将会非常有效。使用组件，端口和连接线来描述一

个数据中心的能力提供一种有效的框架，从而在该框架中模拟和生效部署需求，该模拟和生效在现在的设计和部署过程中是缺少的。

议程：综述，SDM建造模块，应用程序程序例子，宿主例子，逻辑布局，部署，状态。

5 所述SDM是一个元模型，便利的用于捕获分布式应用程序的元素片断和它们的部署环境。所述SDM是可靠的：应用程序和环境是从它们的SDM构造的，对所述应用程序和环境的改变将通过所述SDM进行。给管理进程提供一个名字空间。

10 所述服务定义模型（Service Definition Model）涉及到一个相互关联的模式的集合：

类，类关系和安装模式

组件，端口和连接线类型模式

逻辑布局模式

物理布局模式

15 示例请求模式

实例模式

SDM类对分布式应用程序和部署环境是基础的构件。应用程序类：ASP.Net 网络服务（Web Service），ASP.Net网络站点（Web Site），BizTalk Orchestration Schedule，服务组件（Services Component）（COM+）等等。服务类：IIS服务器，SQL服务器，BizTalk服务器。操作系统，网络和存储器类：Windows虚拟局域网（Windows VLAN），过滤器，磁盘，等等。硬件类：服务器，转换器，防火墙，负载平衡器，存储区域网络（SAN），等等。类由系统级开发者授权并且不会频繁改变。类在所述SDM中在每个组件，端口和连接线后面。每个类包括一个为它的公共设置（简称为设置）和私有设置（称为部署）的模式。在类之间的关系被捕捉：组件类到端口类，连接线类到端口类，和从组件类到组件类。

ASP.Net 网站（Web Site）类

```
<componentClass name="WebSite" layer="Application">http://big/
<settingSchema><xs: schema><xs: complexType><xs: sequence>
<xs: element name="webSiteName" type="xs: string"/>
```

```
<xs: element name="authentication" type="xs: string"/>
<xs: element name="sessionState" type="xs: boolean"/>
</xs: sequence></xs: complexType></xs: schema></settingSchema>
<deploymentSchema><xs: schema><xs: complexType><xs: sequence>
5   <xs: element name="fusionManifest" type="xs: string"/>
    </xs: sequence></xs: complexType></xs: schema></deploymentSchema>
<portClassesAllowed closed="true">
<portClassRef name="ClientDataAccess"/>
<portClassRef name="WebServer" maxOccurs="1"/>
10  <portClassRef name=" SoapClientInterface"/>
<portClassRef name=" RemotingClientInterface"/>
</portClassesAllowed>
</componentClass>
SOAP 客户端口类
15  <portClass name=" SoapClientInterface" layer="Application">http://big
<settingSchema><xs: schema><xs: complexType><xs: sequence>
<xs: element name="formatter" type="xs: string"/>
<xs: element name="transport" type="xs: string"/>
</xs: sequence></xs: complexType></xs: schema></settingSchema>
20  <deploymentSchema><xs: schema><xs: complexType><xs: sequence>
<xs: element name="wsdlFile" type="xs: string"/>
</xs: sequence></xs: complexType></xs: schema></deploymentSchema>
</portClass>
SOAP 连接线类
25  <wireClass name="SoapConnnection" layer="Application">
<settingSchema/>
<deploymentSchema/>
<portClassesAllowed>
<portClassRef name="SoapServerInterface"/>
30  <portClassRef name=" SoapClientInterface"/>
```

```
</portClassesAllowed>
</wireClass>
IIS 组件类
<componentClass name="IIS" layer="Service">
  5 <settingSchema><xs: schema><xs: complexType><xs: sequence>
    <xs: element name="certificateAuth" type="xs: boolean"/>
    <xs: element name="ntlmAuth" type="xs: boolean"/>
    <xs: element name="sessionStateType" type="xs: string"/>
    </xs: sequence></xs: complexType></xs: schema></settingSchema>
  10 <deploymentSchema><xs: schema><xs: complexType><xs: sequence>
    <xs: element name="fusionManifest" type="xs: string"/>
    </xs: sequence></xs: complexType></xs: schema></deploymentSchema>
-<portClassesAllowed>
<portClassRef name="HTTPServer"/>
  15 <portClassRef name="HTTPClient"/>
<portClassRef name="TDSClient"/>
</portClassesAllowed>
</componentClass>
类关系和安装程序
  20 <hostRelation classRef "WebSite" hostClassRef="IIS"
installerRef="WebSiteInstaller"/>
<installer name="WebSiteInstaller" code="WebSiteInstaller, IISInstaller"
codeType "assembly"/>
<宿主关系>捕捉类之间的宿主关系: IIS可以容纳Web Site
  25 安装程序是SDM运行时的“插件程序”，它负责创建组件、端口和/或连接线
类的新的实例。安装程序也负责配置类的实例。不同的安装程序可能使用同样的
基础的部署/配置技术，例如Fusion或WMI.Config。
  分布式应用程序
  分布式应用程序是用组件，端口和连接线类构造出来的。开发者用类中创
  30 建组件，端口和连接类型。类型是类的“用途”，它提供设置值和部署模式。类
```

型是一个复用的单元。类型在Visual Studio中映射一个单一的工程（project）。

SDM通过复合组件类型支持类型的复合。复合允许较大的分布式应用程序被用较小的来构建。复合组件类型在Visual Studio中映射到一个新的工程类型—Whitehorse。

5 FMStocks.Web 组件类型

```
<componentType name="FMStocks.Web" class="WebSite">
<ports>
<port name="web" type="webServer"/>
<port name="stock" type=" StockClient"/>
10 <port name="accounts" type="AccountClient"/>
<port name="trades" type="TradeClient"/>
</ports>
<settings>
<webSiteName>FMStocks.Web</webSiteName>
15 <authentication>Certificate</authentication>
<sessionState>true</sessionState>
</settings>
<deployment>
<fusionManifest>fmstocks.web.manifest</fusionManifest>
20 </deployment>
</componentType>
```

FMStocks7 复合组件类型

```
<compoundComponentType name="FMStocks">
<components>
25 <component name="web" type="FMStocks.Web"/>
<component name="svc" type="FMStocks.WebService"/>
<component name="biz" type="FMStocks.BusinessService"/>
<component name="custdb" type="FMStocks.CustomerDatabase"/>
</components>
30 <wires/>
```

```
<delegatePorts>
<port name="web" componentname="web" portname="web"/>
<port name="svc" componentname="svc" portname="svc"/>
</delegateports>
5 </componentType>
```

SDU和部署环境

一个分布式应用程序的组件，端口和连接线类型与在一个服务部署单元（Service Deployment Unit, SDU）中的任何二进制文件一起被打包。二进制文件包括所有的.DLL文件，.EXE文件，.config文件，静态内容，等等。SDU表示一个可移植的，独立安装的，分布式的应用程序。类似于用于桌面应用程序的Windows安装程序MSI文件。但是，与主要面向单一环境（Windows）的桌面应用程序不同，分布式应用程序能够被容纳在不同的改变较大的部署环境中。必须能够在部署环境中表述它们的需求。必须服从它们的部署环境中的所有策略。

因此，我们需要一个模型来表述应用程序和部署环境两者的需求和约束。
15 My WebSite组件类型需要IIS服务器，该服务器已经通过存储在SQL数据库中的会话状态所配置。所述网页区域将仅仅容纳那些使用授权验证的webSite组件。

IIS 组件类型

```
<componentType name="WebTierIIS" class="IIS">
<ports/>
20 <settings>
<certificateAuth>true</certificateAuth>
<ntlmAuth>false</ntlmAuth>
<sessionStateType>true</sessionStateType>
</settings>
25 <deployment/>
<hostedClasses>
<hostedClass class="WebSite">
<! --使用Xpath所表示的约束语言-->
<constraint>/[authentication="certificate"</constraint>
30 </hostedClass>
```

```
</hostedClasses>
</componentType>
FMStocks.Web组件类型(再访问)
<componentType name="FMStocks.Web" class="WebSite">
5   <ports/>
   <settings>
     <webSiteName>FMStocks.Web</webSiteName>
     <authentication>Certificate</authentication>
     <sessionState>true</sessionState>
10  </settings>
   <deployment>
     <fusionManifest>fmstocks.web.manifest</fusionManifest>
   </deployment>
   <hostConstraints>
15   <hostConstraint hostClass="IIS">
     <constraints>/[sessionStateType="SQL"]</constraints>
   </hostConstraint>
   </hostConstraints>
   </componentType>
20  逻辑布局
```

在一个SDU能够被配置前，我们必须首先对在目标部署环境上的类型做一个逻辑布局。逻辑布局可以在设计时间被完成。需求和约束被检查，并且开发者被警告任何错误或警告。用所述SDU在一个分离文件捕捉逻辑布局的结果。一个SDU可以相对于不同的部署环境（开发，测试，生产，等等）有不同的逻辑布局。

约束检测是使用在每个组件，端口和连接线类上所指定的XPath和XSD来实现的。

建造部署环境
部署环境被使用所述SDM模型来建造。参见图22。本质上，它们是在不同30 层的SDM应用程序。组件，端口和连接线类型以一种相同的方式被使用来组成

服务宿主，网络架构和硬件。在Whidbey timeframe中，我们将只支持部署应用程序层。在ADS V2.0中，我们将能够部署服务宿主（Service Host），网络（Network）和硬件层（Hardware Layer）。Visual Studio构造设计者，用于构造部署环境。Visual Studio将这些看作逻辑基础结构模型。图23说明了一个部署的例子。

5 **实例请求文档（Instance Request Document）**

SDM类型是不变比例的并可以被创建为任何比例。所述实例请求文档是所述需要被创建的实例的一个说明性的定义。包括布线拓扑。图24说明了类型例子，而图25说明了实例请求的例子。

物理布局（Physical Placement）

10 物理布局是采集特殊宿主实例的行为，其中宿主实例是部署的目标。物理布局被逻辑布局约束。约束在物理布局期间重新生效。见图26。

部署

15 SDU，逻辑布局文件，实例请求，和物理布局文件被提供给SDM运行时间。所述SDM运行时间然后将会基于类和宿主关系调用相应的安装程序。所述安装程序负责在宿主上创建一个新的实例，并配置它，来匹配在类型上的设定值。SDM运行时间将包括一个所有实例，它们的最终设置值和布局的数据库。运行时间API将支持实例空间的查询。

SDM模式详细说明（SDM Schema Design Specification）

20 存在3个SDM模式的核心元素：端口，连接线和组件。端口表示了通信端点，组件表示了一个分布式应用程序的各部分，并且连接线表示了在应用程序部分之间的通信连接。这些在三个分离的空间里以不同的形式表现：资源空间，应用程序空间，和实例空间。

25 在所述资源空间中，资源类将被定义，其中该资源类将建造应用程序空间的应用程序。这些类提供了一个应用程序部分的通用分类，允许工具支持一个大范围的应用程序，并提供在设计时间类型检测的基础。我们期待这些核心类向服务设计提供一组综合特征，并且我们期待它们将随着时间改变缓慢改变。

30 在所述应用程序空间，应用程序类型被建造。我们得到一个资源类，并且填充了细节，例如提供向内容的链接，为参数提供设定值。然后我们通过联合端口类型和组件类型，使用包括在复合组件类型中的组件类型，以及通过使用连接线类型描述在复合组件类型的成员中的通信关系，用这些类型建造分布式

应用程序。

所述实例空间包括在部署和运行一个应用程序的过程中所创建的实例。我们公开我们在应用程序空间直到SDM运行时间所定义的通信关系，这样允许实例来找到其它实例。

5 资源类

为了在设计时间检测配置然后在运行时间部署，我们使用资源类来定义我们需要知道的应用程序的元素。这些元素是：

a) 谁与一个应用程序通信。为了相对于一个网络拓扑校验一个分布式应用程序，我们需要知道应用程序的部分能够用来互相通信的协议。端口类可以被用来描述协议终点。连接线类被用来描述可以在这些终点之间被构造的关系。

b) 什么设置适用于一个应用程序，及它是如何被部署的。组件类定义可以被用来构造一个应用程序的构件。组件类定义可以被用来控制到所述组件的特殊行为的设置，并且为所述文件和脚本定义一个模式，该模式可以被提供来部署所述组件。

c) 一个应用程序为了正确运行依赖于什么。为了正确工作，一个组件可能依赖特定功能性，该功能性必须已经存在于目标环境中。一个例子是一个依赖于IIS的网络服务（web service）。我们表述这些需求为在资源之间的宿主关系。使用这些关系我们可以在一系列资源类型上建造一棵依赖树，该依赖树允许我们提前检测是否一个特殊应用程序将会运行在一个特定环境中。

应用程序类型

我们使用在所述资源空间中定义的资源类建造应用程序类型。通过这些类，我们派生端口类型和连接线类型来模拟应用程序特殊通信链接，并且我们建造组件类型来模拟所述应用程序的离散部分。

25 端口类型是描述一个应用程序的特定行为的通信终点。我们得到一个端口资源，并且在应用程序中提供它的使用的特殊信息。一个例子可能是一个端口类型，采用了一个简单对象访问协议（SOAP）资源并且提供一个WSDL文件来定义所述应用程序公开的功能。

30 连接线类型定义应用程序特殊通信路径。一个连接线类型限制一个特定连接线资源来连接两个兼容的应用程序终点。例如，我们可能采用一个简单对象

访问协议（SOAP）连接线资源并且限制它连结我们上面定义的SOAP端口类型。

组件类型被用来模拟一个应用程序的部分，该应用程序可以被独立的部署，也可以跨越机器界限成为分布式。例如，一个具有网络前端和一个数据库后端的应用程序可能由多个组件类型组成。在这一情况中，我们可能采用一个网络服务资源并使用它来创建一个网络前端组件类型，并且采用一个数据库资源来创建所述数据库后端组件类型。然后为了模拟应用程序接口，我们将添加适当的端口类型到所述组件类型。我们称这些为端口成员。
5

所使用的复合组件类型是组组件类型，一起来形成一个新的组件类型。一个在复合组件中的组件类型的使用被称为一个组件成员。我们使用我们前面定义的连接线类型连接组件成员向其它成员公开的接口。这些变成这个复合组件
10 的连接线成员。

为了使复合组件看上去像一个组件，它们就需要像一个组件那样公开接口，性能和需求。我们通过从该复合组件的组件成员中授权所述端口成员的一个子集来做到这一点。

15 为了满足一个组件的需求，我们必须捆绑那个组件到另一个具有匹配能力的组件。我们称这个为过程绑定。

典型实施例

在本段我们描述我们用来定义SDM模型的元素的XML模式。设置既被应用程序又被资源使用，因此我们首先描述它们，然后我们描述资源类，再是应用
20 程序类型，最后是实例空间。

命名

名字空间被使用来定义命名范围，在该名字范围内，类和类型可以被定义。在一个名字空间中，所有的类和类型是唯一的。一个名字空间是通过名字，版本，和被用来验证所述名字空间的内容的密码值来定义。

```
<xs:attributeGroup name=" identity">
<xs:attribute name="name" type="xs:string" use="required"/>
<xs:attribute name="version" type="fourPartVersionType"
use="required"/>
<xs:attribute name="publicKeyToken" type="publicKeyTokenType"
```

```
use="optional"/>
</xs:attributeGroup>
```

一个文件版本通过一个格式为N.N.N.N的4部分的数所定义。其中0<N<65535。

```
<xs:simpleType name=" fourPartVersionType">
<xs:annotation>
<xs:documentation>Four part version numbers where the segments
are in the range 0-65535 </xs:documentation>
</xs:annotation>
<xs:restriction base="xs:string">
<xs:pattern value=" (0|[1-5][0-9] {0,4} |[7-9][0-9] {0, 3} |6[0-4][0-
9] {0,3} |6[6-9][0-9] {0,2} |65|65[0-4][0-9] {0,2} |65[6-9][0-9]? |655|655[0-
2][0-9]? |655[4-9]|6553[0-5]? ). (0|[1-5][0-9] {0,4} |[7-9][0-9] {0,3} |6[0-4][0-
9] {0,3} |6[6-9][0-9] {0,2} |65|65[0-4][0-9] {0,2} |65[6-9][0-9]? |655|655[0-
2][0-9]? |655[4-9]|6553[0-5]? ).(0|[1-5] [0-9] {0, 4} | [7-9] [0-9] {0, 3}
} |6 [0-4] [0-
9] {0,3} |6[6-9][0-9] {0,2} |65|65[0-4][0-9] {0,2} |65[6-9][0-9]? |655|655[0-
2] [0-9]? |655[4-9]...6553[0-5]? . )(0| [1-5][0-9] {0,4} |[7-9][0-9] {0,3}
|6[0-4][0-
9] {0,3} |6[6-9][0-9] {0,2} |65|65[0-4][0-9] {0,2} |65[6-9][0-9]? |655|655[0-
2][0-9]? |655[4-9]|6553[0-5]? ) ")/>
</xs: restriction>
</xs: simpleType>
```

一个公钥令牌是一个16字符16进制的字符串，它标识一个公/私钥对里的公共部分。所述文档将使用私钥来签名，允许这个文档的使用者通过公钥来校验
5 它的内容。

```
<xs:simpleType name="publicKeyTokenType">
<xs:annotation>
<xs:documentation>Public Key Token:16 hex digits in
size</xs:documentation>
</xs:annotation>
<xs:restriction base="xs:string">
<xs:pattern value=" ([0-9A-F][a-f...[A-F]] {16} )"/>
</xs:restriction>
</xs:simpleType>
```

然后，在名字空间里的简单名字被使用字符串构建。我们允许名字空间通过导入它们到当前名字空间，接着将该名字空间与一个别名相关联来引用其它的名字空间。

```
<xs:complexType name="import">
<xs:attribute name="alias" type="xs:string" use="required"/>
<xs:attributeGroup ref=" identity"/>
</xs:complexType>
```

然后，类和类型的引用既可以是涉及定义在当前名字空间中的对象的简单名字，又可以是一个复合名字，该复合名字使用一个别名和简单名字两者来标识在其它名字空间中所定义的一个对象的。

设置

资源类和应用程序类型都可以公开一个设置模式。当一个新端口，连接线或组件类型用一个类创建时，当一个端口类型被加到一个组件类型中时，或当一个连接线类型或组件类型被用到一个复合组件类型中时，这个模式被用来描述可以被提供的值。

设置模式

我们使用XSD来描述所述设置模式。对于最初的版本，我们使用一个XSD

的子集，该子集对简单类型和元素类型列表是有限的。

```
<xs:complexType name="settingSchema">
<xs:sequence>
<xs:any namespace="http://www.w3.org/2001/XMLSchema"
processContents="skip" minOccurs="0"
maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
```

设置值

当一个基于类的类型被创建时，或者当一个类型被用在一个组件或复合组件中时，设置值被提供。所述设置值是XML块，该设置值符合正确的设置模式。

```
<xs:complexType name="settingValues">
<xs:sequence>
<xs:any namespace="## other" processContents="lax"/>
</xs:sequence>
</xs:complexType>
```

设置流

我们使用设置流来允许设置值从组件类型传送到组件类型的成员。设置流是使用在设置值段中的XPATH来执行的，该设置值段从由类型定义的设置模式中选择值。

我们通过使用一个定义在SDM名字空间中的特殊属性来标识我们想要输入内容的值。如果这种属性存在于一个元素，那么我们期望这个属性值对该类型来说是这个设置模式中的XPath。

设置约束

设置约束被用来确认和约束设置值。例如一个IIS服务器可能需要它容纳的所有web service将某些它们的设置值约束到一个特定值或值的范围。我们使用XPATH来有效化设置值（或XQUERY一旦它被全面支持）。我们支持如下格式

的查询:

- 路径必须存在。
- 路径必须不存在。
- 如果路径存在, 那么[(路径必须存在|路径必须不存在) *]

5 使用第一形式我们可能需要设置被设置为一个特定值或一系列值, 使用第二种形式我们可能需要设置不被设置为一个值或一系列值, 而使用第三种形式我们可以构造要被一起设置的需要设置集合的设置之间的关系。

```
<xs: complexType name="settingConstraints">
  <xs: sequence>
    <xs: element name="mustExist" type="simpleTest"
      minOccurs="0" maxOccurs="unbounded"/>
    <xs: element name="mustNotExist" type="simpleTest"
      minOccurs="0" maxOccurs="unbounded"/>
    <xs: element name="ifExists" type="nestedTest"
      minOccurs="0" maxOccurs="unbounded"/>
    <xs: element name=" ifNotExists" type="nestedTest"
      minOccurs="0" maxOccurs="unbounded"/>
  </xs: sequence>
</xs: complexType>

<xs: attributeGroup name="testAttributes">
  <xs: attribute name="path" type="xs: string"/>
  <xs: attribute name="ifNullPath" type="ifNullPath"/>
  <xs: attribute name="error" type="xs: int"/>
  <xs: attribute name=" errorDesc" type="xs: string"/>
</xs: attributeGroup>

<xs: complexType name="simpleTest">
  <xs: attributeGroup ref="testAttributes"/>
```

```

</xs: complexType>

<xs: complexType name="nestedTest">
<xs: sequence>
<xs: element name="mustExist" type="simpleTest"
minOccurs="0" maxOccurs="unbounded"/>
<xs: element name="mustNotExist" type="simpleTest"
minOccurs="0" maxOccurs="unbounded"/>
<xs: element name="ifExists" type="nestedTest"
minOccurs="0" maxOccurs="unbounded"/>
<xs: element name=" ifNotExists" type="nestedTest"
minOccurs="0" maxOccurs="unbounded"/>
</xs: sequence>
<xs: attributeGroup ref="testAttributes"/>
</xs: complexType>

```

当路径不存在时，我们需要公开选项来解决这种情况。下面允许设计者来选择生成一个错误，插入一个值或忽略该测试。

```

<xs: simpleType name=" ifNullPath">
<xs: restriction base="xs: string">
<xs: enumeration value="skip"/>
<xs: enumeration value=" override"/>
<xs: enumeration value="returnError"/>
</xs:restriction>
</xs: simpleType>

```

例子

下面的是一个计算机类可能公开的简单模式值。所述模式具有一个单独的
5 顶级节点，该节点标识设置组和隶属于结点的三个属性。

```

<settingSchema>
  <x: schema>
    <x: element name="processorSettings">
      <x: complexType>
        <x: sequence>
          <x: element name="numberOfCpus" type="x: int"/>
          <x: element name="memory" type="x: int"/>
          <x: element name="dualHomed" type="x: Boolean"/>
        </x: sequence>
      </x: complexType>
    </x: element>
  </x: schema>
</settingSchema>

```

我们可以向所述模式提供如下类型中的值。

```

<settings>
  <processorSettings>
    <numberOfCpus>4</numberOfCpus>
    <memory>8000</memory>
    <dualHomed>false</dualHomed>
  </processorSettings>
</settings>

```

如果我们想要在类型被使用时提供设置值，那么我们将使用下面的设置。

约束可能相对于这些值被编写。在例子中，第一是一个简单的mustExist 约束。第二约束使用一个测试来确定是否评价嵌套约束。

```

<constraints>
  <mustExist path="ProcessorSettings/memory=1000">
    errorDesc="Host machine does not have enough memory"/>

```

```

<ifExists path="ProcessorSettings/cpu>=2">
  errorDesc="Host machine has two processors but not enough
  resources">
<mustExist path= "ProcessorSettings/memory>=2000">
  errorDesc="Host machine does not have enough memory"/>
</ifExists>
</constraints>

```

资源

基本类

所有的资源类模式从类中派生。它们共享一个设置模式，部署模式，以及名字和层属性。这些设置模式描述了基于这个类适用于类型的设置，它的可能取得值和每一个的描述。所述部署模式描述部署基于这个资源的类型所需要的信息，该层属性将资源与设计空间一层关联起来。名字属性被用来给出该类在名字空间中的唯一的一个名字。

```

<xs: complexType name="class">
<xs: sequence>
<xs: element name="deploymentSchema" type=" deploymentSchema"
minOccurs="0"maxOccurs="1"/>
<xs: element name=" settingSchema" type="settingsSchema"
minOccurs="0"maxOccurs="1"/>
</xs: sequence>
<xs: attribute name="name" type="xs: string" use="required"/>
<xs: attribute name="layer" type="layer" use="required"/>
</xs: complexType>

```

为了部署模式，名字空间被留下未定义。在模式上的约束对类完全是安装程序的责任。

```

<xs: complexType name="deploymentSchema">
<xs: sequence>
<xs: any namespace="#other" processContents="lax"/>
</xs: sequence>
</xs: complexType>

```

该值提供作为必须匹配这些关联的部署模式的部署段的一部分。

```

<xs: complexType name="deploymentValues">
<xs: sequence>
<xs: any namespace="#other" processContents="lax"/>
</xs: sequence>
</xs: complexType>

```

该层属性是4层类型的一个枚举。所述应用程序层包括高级应用程序组件，例如数据库和网络服务。服务层包括中间件服务，例如IIS和SQL。网络层包括操作系统，存储器和网络定义。硬件层包括一个数据中心的硬件组件的定义。

```

<xs: simpleType name="layer">
<xs:restriction base="xs: string">
<xs: enumeration value="Application"/>
<xs: enumeration value="Service"/>
<xs: enumeration value="Network"/>
<xs: enumeration value="Hardware"/>
</xs: restriction>
</xs: simpleType>

```

5 端口类

端口类不包含任何上面的信息，它在资源基本类型中定义。

```

<xs: complexType name="portClass">
<xs: complexContent>

```

```
<xs: extension base="class">
</xs: extension>
</xs: complexContent>
</xs: complexType>
```

组件类

一个组件类通过添加一个允许端口列表扩充基本类。

```
<xs: complexType name="componentClass">
<xs: complexContent>
<xs: extension base="class">
<xs: sequence>
<xs: element name="portClassesAllowed"
type= "portClassesAllowed"
minOccurs="0" maxOccurs="1"/>
</xs: sequence>
</xs: extension>
</xs: complexContent>
</xs: complexType>
```

5 端口类的列表可以被打开或关闭，如果它被关闭，那么只有那些端口类型基于出现在列表中的类的端口可以被使用在该关联组件类型上。minOccurs和maxOccurs属性定义了这些端口类型可以被使用的次数。

```
<xs: complexType name= "portClassesAllowed">
<xs: sequence>
<xs: element name="portClassRef" minOccurs="0"
maxOccurs="unbounded">
<xs: complexType>
<xs: attribute name="name" type="xs: string" use= "required"/>
<xs: attribute name="minOccurs" type="xs: int" use= "optional"/>
<xs: attribute name= "maxOccurs" type="xs: string" use="optional"/>
</xs: complexType>
```

```

</xs: element>
</xs: sequence>
<xs: attribute name="closed" type="xs: boolean"
default="true" use="optional"/>
</xs: complexType>

```

连接线类

连接线类也通过添加一个允许端口类列表扩充所述基本模式。在这一情况下，列表定义了端口类型的类，该端口类型可以与连接线类型相关联。

```

<xs: complexType name="wireClass">
<xs:complexContent>
<xs:extension base="class">
<xs:sequence>
<xs:element name="portClassesAllowed"
type="portClassesAllowed" minOccurs="0" maxOccurs="1" />
</xs:sequence>
</xs:extension>
<xs:complexContent>
</xs:complexType>

```

宿主关系

宿主关系定义是一个三重的标识，一个资源类，一个目标类，和一个安装程序。关系的实例指出一个基于源类的类型实例可以使用一个基于目标类的类型的实例和与这个关系相关联的安装程序来创建。目标类必须是组件类。

例如，一个webservice类可能是在与IIS类和webservice安装程序一起的宿主关系。在这个情况中，关系指出使用安装程序在类型MyIIS上创建一个MyWebservice类型的实例是可能的。我们不了解它是不是能够创建这个关系直到我们已经评价存在于应用程序空间和实例空间两者中的约束。

```

<xs: complexType name="hostRelation">
  <xs: attribute name="classRef type="xs: string" use="required"/>
  <xs: attribute name="componentHostClassRef type="Ns: string"
    use="required"/>
  <xs: attribute name="installerRef type="xs: string" use="required"/>
</xs: complexType>

```

安装程序是通过名称，代码类型和到实现所述安装程序的二进制文件中的链接来标识的。

```

<xs: complexType name=" installer">
  <xs: sequence>
    <xs: element name="binary" type="xs: string" minOccurs="1"
      maxOccurs="1" />
  </xs: sequence>
  <xs: attribute name="codeType" type="xs: string" use="required" />
  <xs: attribute name="name" type="xs: string" use="required" />
</xs: complexType>

```

例子

这些例子是从扩展的4层例子中选录的。参见完整的例子文件来得到细节。

5 首先我们创建某些端口类来模拟到一个数据库的访问。在这一情况中，我们有一个服务器端口和一个客户端口。

```

<portClass name=" ServerDataAccess" layer="Application">
  <settingSchema>
    <xs: schema>
      <xs: complexType>
        <xs: sequence>
          <xs: element name="databaseName" type="xs: string" />
          <! 其它连接字符串性质--other connection string properties-->
        </xs: sequence>
      </xs: complexType>
    </xs: schema>
  </settingSchema>
</portClass>

```

```
</xs: complexType>
</xs: schema>
</settingSchema>
</portClass>
<portClass name="ClientDataAccess" layer="Application"/>
```

然后我们创建一个连接线类，模拟在两个端口类之间的通信连接。连接线类具有上面定义的某一些所述两个端口类的设置和引用。在这一情况下，这里连接线约束将是在所述连接上只有一服务器，模拟所述客户端口不了解如何负载平衡连接到多服务器的事实。一个更复杂的连接线实现可能允许多服务器和实现某一些管理的形式来解决连接。

```
5 <wireClass name="DataConnection" layer="Application">
  <settingSchema>
    <xs: schema>
      <xs: complexType>
        <xs: sequence>
          <xs: element name="useSSL" type="xs: boolean"/>
        </xs: sequence>
      </xs: complexType>
    </xs: schema>
  </settingSchema>
  <portClassesAllowed>
    <portClassRef name="ServerDataAccess" maxOccurs="1"/>
    <portClassRef name="ClientDataAccess" />
  </portClassesAllowed>
</wireClass>
```

最后我们创建一个模拟一个数据库的组件类。这个类可以具有一个设置和部署模式两者，并基于这个类标识存在于一个组件类型上的端口。

```
<componentClass name="Database" layer= "Application">
<deploymentSchema>
<xs: schema>
<xs: complexType>
<xs: sequence>
<xs: element name="sqlScriptFilePath" type="xs: string"
maxOccurs="unbounded"/>
</xs: sequence>
</xs: complexType>
</xs: schema>
</deploymentSchema>
<settingSchema>
<xs: schema>
<xs: complexType>
<xs: sequence>
<xs: element name="databaseName" type="xs: string"/>
</xs: sequence>
</xs: complexType>
</xs: schema>
</settingSchema>
<portClassesAllowed closed="true">
<portClassRef name="ServerDataAccess"/>
</portClassesAllowed>
</componentClass>
```

所有的这些组件需要映射到兼容的宿主类型。在这一情况中，SQL服务器为服务器端口和数据库担当宿主，IIS担当所述SQL客户端口的宿主。这些类被定义在一个分隔的名字空间里重新命名为中间件。

```

<hostRelations>
  <installer name="DatabaseInstaller" codeType="InstallerPlugin"/>
  <hostRelation classRef="database"
    componentHostClassRef="middleware: SQL"
    installerRef="DatabaseInstaller"/>
  <hostRelation classRef="ServerDataAccess"
    componentHostClassRef="middleware: SQL"
    installerRef="DatabaseInstaller"/>
  <hostRelation classRef="ClientDataAccess"
    componentHostClassRef="middleware: IIS"
    installerRef="WebServiceInstaller"/>
</hostRelations>

```

应用程序

应用程序开发者创建在应用程序空间中的组件，端口和连接线类型来模拟他的应用程序。这些被选择类创建的类型与层匹配，在该层上所述开发者进行工作，并且然后给这些类提供值。

5 应用程序基本类

所有应用程序类型模式都是基于下面的应用程序基本模式的。所述基本模式标识类型所基于的类和类型名称。在模式主体中，我们标识允许该类型被部署的部署值，以及在相关类上的设置模式的设置。该类型也可能定义一种新的设置模式，该模式标识当这种类型被用在其它类型时可以被提供的值。最后该基本类型包括一个宿主约束的段。这一段基于宿主关系标识类型在可能宿主上的约束，该关系存在于与此类型相关的类的资源空间中。

```

<xs: complexType name="baseType">
  <xs: sequence>
    <xs: element name=" deployment" type="deploymentValues"
      minOccurs="0" maxOccurs="1"/>
    <xs: element name="settings" type="settingsValues"
      minOccurs="0" maxOccurs="1"/>
    <xs: element name="settingSchema" type="settingSchema">

```

```

minOccurs="0"maxOccurs="1"/>
<xs: element name="hostConstraints" type="hostConstraints"
minOccurs="0"maxOccurs="1"/>
</xs: sequence>
<xs: attribute name="class" type="xs: string" use="required"/>
<xs: attribute name="name" type="xs: string" use="required"/>
</xs: complexType>

```

宿主约束（hostConstraint）段包括每个类的一系列约束，即可以容纳与该类型相关的类。这些类由在所述资源空间里的宿主关系所标识。与每个类相关的约束是按照类的设置模式。约束的形式被上面定义。

```

<xs: complexType name="hostConstraints">
<xs: sequence>
<xs: element name="hostConstraint" minOccurs="1" maxOccurs="1">
<xs: complexType>
<xs: sequence>
<xs: element name="constraint" type="settingConstraint"/>
</xs: sequence>
<xs: attribute name="host" type="xs: string" use="required"/>
</xs: complexType>
</xs: element>
</xs: sequence>
</xs: complexType>

```

端口类型

5 端口类型仅仅使用基本类型。没有与一个端口类型关联的进一步的信息。

```

<xs: complexType name="portType">
<xs: complexContent>
<xs: extension base="baseType">
</xs: extension>

```

```
</xs: complexContent>  
</xs: complexType>
```

连接线类型

连接线类型扩充基本类型来增加一个所允许端口类型列表。然后，这些端口类型的使用可能与在复合组件中的一个连接线类型的使用相关联起来。通过以这种方式定义连接线类型，，通过仅仅为兼容的端口类型创建连接线类型，
5 应用程序设计者可以约束一系列的在他的应用程序的各个部分之间所允许的连接。

```
<xs: complexType name="wireType">  
<xs: complexContent>  
<xs: extension base="baseType">  
<xs: sequence>  
<xs: element name= "portTypeRefs" minOccurs="0">  
<xs: complexType>  
<xs: sequence>  
<xs: element name="portTypeRef"  
minOccurs="0" maxOccurs="unbounded">  
<xs: complexType>  
<xs: attribute name="name" type="xs: string"  
use="required"/>  
<xs: complexType>  
</xs: element>  
</xs: sequence>  
</xs: complexType>  
</xs: element>  
</xs: sequence>  
</xs: extension>  
</xs: complexContent>  
</xs: complexType>
```

组件类型

一个组件类型扩充基本类型来增加一端口成员列表和一容纳类列表。
每个端口成员是一个现有类型的使用。容纳类的列表标识该组件可以容纳的类。这些类是在资源空间里由宿主关系标识的类的子集，其中该类型的类被5 标识为一个可能的宿主。

```
<xs: complexType name="componentType">
  <xs: complexContent>
    <xs: extension base= "baseType">
      <xs: sequence>
        <xs: element name="ports" type= "portsList"
          minOccurs="0" maxOccurs=" 1"/>
        <xs: element name="hostedClasses" type="hostedClassesList"
          minOccurs="0" maxOccurs=" 1">
      </xs: sequence>
    </xs: extension>
  </xs: complexContent>
</xs: complexType>
```

每个在端口列表中的端口成员是由名字和类型来标识的。端口名字在组件中必须是唯一的。端口类型必须具有一个关联端口类，该类被允许在关联该组件类型的组件类上。我们可以为每个端口成员提供一设置列表，该设置匹配由端口类型定义的模式。

```
<xs: complexType name= "portsList">
  <xs: sequence>
    <xs: element name="port"  minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs: sequence>
          <xs: element name="settings" type="settingValues" minOccurs="0"
            maxOccurs="1"/>
```

```

</xs: sequence>
<xs: attribute name="name" type="xs: string" use="required"/>
<xs: attribute name="type" type="xs: string"/>
</xs: complexType>
</xs: element>
</xs: sequence>
</xs: complexType>

```

对于在容纳类列表中的每个类，我们可以关联一约束列表。这些约束被相对于所述容纳类的设置模式写入。

```

<xs: complexType name="hostedClassesList">
<xs: sequence>
<xs: element name="hostedClass" minOccurs="1" maxOccurs="1" >
<xs: complexType>
<xs: sequence>
<xs: element name="constraints" type="settingConstraints"
minOccurs="1" maxOccurs="1" />
</xs: sequence>
<xs: attribute name="class" type="xs: string" use="required" />
</xs: complexType>
</xs: element>
</xs: sequence>
</xs: complexType>

```

复合组件类型

复合组件类型（此后被称为复合组件）定义一个新的组件类型。当定义该复合组件时，有一个选项来规定该类型的成员应该是协同定位的。如果成员是协同定位的，那么当该类型被部署时，该类型的所有成员必须被部署在一个单一的宿主上。所述复合组件也包括一个组件成员的列表，一个连接线成员的列表，一个定义所述组件授权端口的段，和标识该组件可以容纳的类的列表。

```

<xs: complexType name="compoundComponentType">
  <xs: sequence>
    <xs: element name="components" type="components"
      minOccurs="0" maxOccurs="1"/>
    <xs: element name="wires" type="wires"
      minOccurs="0" maxOccurs="1"/>
    <xs: element name="delegatePorts" type="delegatePorts"
      minOccurs="0" maxOccurs="1"/>
    <xs: element name="delegateHostedClasses"
      type="delegateHostedClasses"
      minOccurs="0" maxOccurs="1"/>
  </xs: sequence>
  <xs: attribute name="name" type="xs: string" use="required"/>
  <xs: attribute name="colocate" type="xs:boolean"
    use="optional" default="false"/>
</xs: complexType>

```

所述组件列表标识已经被定义的组件类型的使用—我们称之为复合组件的组件成员。每个成员在所述复合组件中具有一个唯一的名字，一个定义该成员的所述类型的引用，以及一个描述它是不是唯一的标识。

如果一个组件成员被标记为唯一的，那么在包括复合组件的实例仅仅可能有这一组件成员的以前的实例。如果没有被标记为唯一的，那么成员的实例可以根据外部因素例如负载改变被创建和删除。这意味着任何连接到一个不唯一的成员的组件成员在运行时间可以看见一个或多个该成员的实例。

每个组件成员也可以为定义在关联组件类型中的设置模式提供设置值。

```

<xs: complexType name="components">
  <xs: sequence>
    <xs: element name="component" minOccurs="0"
      maxOccurs="unbounded">

```

```

<xs: complexType>
<xs: sequence>
<xs: element name="settings" type="settingValues"
minOccurs="0" maxOccurs="1"/>
</xs: sequence>
<xs: attribute name="name" type="xs: string" use="required"/>
<xs: attribute name="type". type="xs: string" use="required"/>
<xs: attribute name="singleton" type="xs: boolean"
use="optional" default="false"/>
</xs: complexType>
</xs: element>
</xs: sequence>
</xs: complexType>

```

在一个复合组件里连接线类型的使用被称为一个连接线成员。每个连接线成员具有一个名字，该名字对于复合组件是唯一的并且标识一个关联连接线类型。连接线成员也可以为在连接线类型中定义的设置模式提供设置值。

- 一个连接线成员的关键任务是标识在复合组件中的组件成员之间的连接。
 5 完成该任务的方式是给连接线成员添加端口引用。每个端口引用标识在一个复合组件中的组件成员上的端口。所述引用端口的端口类型必须匹配与连接线类型相关联的端口类型。

```

<xs: complexType name="wires">
<xs: sequence>
<xs: element name="wire" minOccurs="0" maxOccurs="unbounded">
<xs: complexType>
<xs: sequence>
<xs: element name="settings" type="settingValues" minOccurs="0"
maxOccurs="1"/>
<xs: element name="members" minOccurs="1" maxOccurs="1">
</xs: complexType>

```

```

<xs: sequence>
  <xs: element name="member" type="componentPortRef"
    minOccurs="0" maxOccurs="unbounded"/>
</xs: sequence>
<xs: complexType>
  </xs: element>
</xs: sequence>
  <xs: attribute name="name" type="xs: string" use="required"/>
  <xs: attribute name="type" type="xs: string"/>
</xs: complexType>
</xs: element>
</xs: sequence>
</xs: complexType>

```

一个端口引用标识一个在同样包括在复合组件中的组件成员。所述端口名称是在与组件成员关联的组件类型上的一个端口成员的名称。

```

<xs: complexType name="componentPortRef">
  <xs: attribute name="componentName" type="xs: string"/>
  <xs: attribute name="portName" type="xs: string" use="required"/>
</xs: complexType>

```

一个复合组件不能直接使用端口类型，因为没有代码与该端口成员可以绑定的复合组件相关。相反的，我们从所述复合组件的组件成员中授权端口成员。

- 5 这意味着当它们被用作一个组件类型时，这些端口出现就像它们属于该复合组件一样。

当一个端口被授权时，它就通过首先标识该组件成员，然后标识该组件中的端口成员来标识。作为这一过程的一部分，倘若其中有相同名字的端口被从不同组件成员中授权了，该端口为了避免名字冲突，可以被重命名。

```

<xs: complexType name="delegatePorts">
  <xs: sequence>
    <xs: element name="delegatePort" minOccurs="0">

```

```

maxOccurs="unbounded">

<xs: complexType>
<xs: attribute name="name" type="xs: string"/>
<xs: attribute name="componentName" type="xs: string"/>
<xs: attribute name="portName" type="xs: string" use="optional"/>
</xs: complexType>
</xs: element>
</xs: sequence>
</xs: complexType>

```

为了构造能够为一定范围内的不同类提供服务的宿主，我们允许一复合组件公开来自它的组件成员的宿主类说明。当所述复合组件被用作一个组件类型时，那么它表示该复合组件能够为所有被说明的类担当宿主。

我们以与我们授权端口成员相似的方式使用授权来公开这些类的说明。我们标识所述组件成员，该组件成员包括该宿主类，接着我们标识所述组件要求能够容纳的类。

```

<xs: complexType name="delegateHostedClasses">
<xs: sequence>
<xs: element name="hostedClassRef"
minOccurs="1" maxOccurs="unbounded">
<xs: complexType>
<xs: attribute name=" componentName" type="xs: string"/>
<xs: attribute name="hostedClass" type="xs: string "
use="required"/>
</xs: complexType>
</xs: element>
</xs: sequence>
</xs: complexType>

```

绑定

5 绑定是我们为一个特定复合组件的成员标识宿主的处理。我们执行这一处
理是为了检测在应用程序和该应用程序将会被容纳环境之间的兼容性，而且是
为了部署该应用程序。所述应用程序和宿主环境两者都使用复合组件来模拟，
因而绑定的过程支持从成员之间的连接拓扑的两个组件中找到匹配成员。

10 我们通过查看在资源空间的类之间的关系开始为一个成员标识兼容的宿主。
我们查看连接线合组件成员的类型，然后标识与该成员相关的类。然后我们在
具有与它们的组件类型关联的类兼容的宿主组件中寻找组件成员。接着我们查
看在与它们的组件类型相关联的类型上的宿主约束，并且看它们是否与宿主成
员的类型上的设置匹配。接着我们执行反向过程，检测在相对于我们想容纳的
成员的类型上的设置，在宿主成员类型上的宿主类约束。

如果我们试图匹配一个组件成员，那么我们需要检测，该组件成员的类型
的所有端口成员也可以被容纳在该组件成员的任一可能的宿主上。

15 如果我们试图匹配一个连接线成员，那么我们必须匹配任何存在于在宿主
之间的组件成员，我们在我们试图容纳的复合组件中为组件成员选择该宿主。

基于在先前的例子中我们描述的端口类，我们创建了两个端口类型。

```
<portType name="UserDataServer" class="ServerDataAccess">  
<deployment/>  
<settings/>  
</portType>  
<portType name="UserDataClient" class="ServerDataAccess">  
<deployment/>  
<settings/>  
</portType>
```

这些类型被一个连接线类型遵守。

```
<wireType name="UserData" class="DataConnection">  
<deployment/>
```

```

<settings>
<useSSL>false</useSSL>
</settings>
<portTypeRefs>
<portTypeRef name="UserDataServer"/>
<portTypeRef name="UserDataClient"/>
</portTypeRefs>
</wireType>

```

现在我们基于数据库类创建一个组件类型。该数据库类公开了一个服务器数据端口。

```

<componentType name="UserData" class= "Database ">
<deployment>
<sqlScriptFilePath>%install%\mydatabaseDfn.sql</sqlScriptFilePath>
</deployment>
<settings>
<databaseName>UserData</databaseName>
</settings>
<ports>
<port name="userData" type="UserDataServer"/>
</ports>
</componentType>

```

我们可以创建一个复合组件类型，该复合组件类型使用这些类型中的某一些。下面的复合组件使用3个组件类型。第1类型UserPages表示一个具有两个访问点的网络服务，第二类型QueryMangement是一个中间层逻辑组件，最后类型是我们的数据库类型。我们使用两个连接线类型将这些组件连接到一起：UserData和QueryManager。该数据连接线连接中间层到数据库，该查询连接线连接前端到中间层。接着我们公开两个端口：signup和enquiry，使用授权从前端。

```
<compoundComponentType name="UserManagementApplication">
<components>
<component name="userPages" type="UserPages"/>
<component name=" queryLogic" type="QueryManagement"/>
<component name="userData" type="UserData" singleton="true"/>
</components>
<wires>
<wire name="data" type="UserData">
<members>
<member componentName="queryLogic" portName="userData"/>
<member componentName="userData" portName="userData"/>
</members>
</wire>
<wire name="query" type=" QueryManager">
<members>
<member componentName="userPages"
portName="queryManager l"/>
<member componentName="userPages"
portName="queryManager2"/>
<member componentName="queryLogic"
portName="queryManager"/>
</members>
</wire>
</wires>
<delegatePorts>
<delegatePort name="signup" componentName="userPages"
portName="signup"/>
<delegatePort name=" enquiry" componentName="userPages"
portName="enquiry"/>
```

```
</delegatePorts>
</compoundComponentType>
```

SDM文档结构

一个SDM文档具有一个很强的同一性，该统一性定义了文档的名字空间。它导入一个其它名字空间的引用列表。该文档也包括一个信息段，该段标识文档特殊属性，例如文档的所有者，公司名称和修订日期。它还包括端口，连接线和组件类的列表，接着是宿主关系列表，再依次是端口，连接线和组件类型列表。

```
<xs: element name=" sdm' >
<xs: annotation>
<xs: documentation>SDM 根元素。它是SDM类型的根元素。
</xs: documentation>
</xs: annotation>
<xs: complexType>
<xs: sequence>
<xs: element name="import" type="import" minOccurs="0"
maxOccurs="unbounded"/>
<xs: element name=' information" type=" information"
minOccurs=" 0" maxOccurs=" 1" / >
<xs: element name="portClasses" minOccurs="0" maxOccurs=' 1" >
<xs: complexType>
<xs: sequence>
<xs: element name= "portClass" type= "portClass"
minOccurs=" 1" maxOccurs="unbounded" / >
</xs: sequence>
</xs: complexType>
</xs: element>
<xs: element name="wireClasses" minOccurs="0" maxOccurs=" 1" >
<xs: complexType>
```

```
<xs: sequence>
<xs: element name="wireClass" type="wireClass"
minOccurs="1" maxOccurs="unbounded"/>
</xs: sequence>
</xs: complexType>
</xs: element>
<xs: element name="componentClasses" minOccurs="0"
maxOccurs="1" >
<xs: complexType>
<xs: sequence>
<xs: element name="componentClass". type="componentClass"
minOccurs="1" maxOccurs="unbounded" / >
</xs: sequence>
</xs: complexType>
</xs: element>
<xs: element name="hostRelations" minOccurs="0"
maxOccurs="1" >

<xs: complexType>
<xs: sequence>
<xs: element name="installer" type="installer" minOccurs="1"
maxOccurs="unbounded"/>
<xs: element name="hostRelation" type="hostRelation"
minOccurs="1" maxOccurs="unbounded" / >
</xs: sequence>
</xs: complexType>
</xs: element>
<xs: element name="portTypes" minOccurs="0" maxOccurs="1" >
<xs: complexType>
<xs: sequence>
```

```
<xs: element name="portType" type="portType"
minOccurs="0" maxOccurs="unbounded" />
</xs: sequence>
<xs: complexType>
</xs: element>
<xs: element name="wireTypes" minOccurs='0' maxOccurs="1" >
<xs: complexType>
</xs: sequence>
<xs: element name="wireType" type="wireType"
minOccurs="0" maxOccurs="unbounded" />
</xs: sequence>
<xs: complexType>
</xs: element>
<xs: element name="componentTypes" type="componentType"
minOccurs='0' maxOccurs="unbounded" >
<xs: element name="compoundComponentTypes"
type="compoundComponentType" minOccurs='0'
maxOccurs="unbounded" >
</xs:sequence>
</xs: complexType>
</xs: element>
</xs: sequence>
<xs: attributeGroup ref=" identity"/>
</xs: complexType>
</xs: element>
```

关联 XSD

以下是一个改变请求的示例结构。

```
<?xml version="1.0" encoding="utf-8" ?>
<xs: schema targetNamespace="urn: schemas-microsoft-
```

```
com: sdmChangeRequest" xmlns="urn: schemas-microsoft-
com: sdmChangeRequest" xmlns: settings="urn: schemas-microsoft-
com: sdmSettings" xmlns: mstns="http://tempuri.org/XMLSchema.xsd"
xmlns: xs="http://www.w3. org/2001 /XML S chema"
5      elementFormDefault="qualified" version=" 0.7" id="sdmChangeRequest">
<xs: import namespace="urn:schemas-microsoft-com:sdmSettings"
schemaLocation=" SDM7Settings.xsd"/>
<xs: import namespace="urn: schemas-microsoft-com: sdmNames"
schemaLocation=" SDM7Names.xsd"/>
10     <xs: complexType name=" ChangeRequestType " >
<xs: sequence>
<xs: element name="group", type=" groupType "
minOccurs="0" max0ccurs="unbounded"/>
</xs: sequence>
15     </xs: complexType>
<xs: complexType name="groupType">
<xs: sequence>
<xs: element name=" group" type=" groupType "
minOccurs=" 0 " maxOccurs="unbounded"/>
<xs: element name=" addInstance "
20           type=" addlnstanceType " minOccurs="0" maxOccurs="unbounded"/>
<xs: element name="updateInstance "
type="updateInstancetypE" minOccurs="0" maxOccurs="unbounded"/>
<xs: element name=" deleteInstancE "
type="deleteInstancetypE" minOccurs="0" maxOccurs="unbounded"/>
25     <xs: element name=" addConnection "
type=" addConnectionType " minOccurs=" 0 " maxOccurs="unbounded"/>
<xs: element name="deleteConnection "
type=" deleteConnectionType" minOccurs="0" maxOccurs="unbounded"/>
<xs: element name=" deleteConnectionType "
type=" deleteConnectionType" minOccurs="0" maxOccurs="unbounded"/>
30     </xs: sequence>
```

```
<xs: attribute name=" canLeConcurrentlyExecuted"
    type="xs:boolean"/>
</xs: complexType>
<xs: complexType name=" addInstanceType" >
    5   <xs: sequence>
        <xs: element name=" classSettings"
            type="settings: settingValues" minOccurs="0"/>
        <xs: element name="typeSettings"
            type="settings: settingValues" minOccurs=" 0" />
    10  <! --为类设置值-->
        <! --为类型设置值-->
    </xs:sequence>
    <xs:attribute name="parent" type="reference" use="optional"
    />
    15  <xs: attribute name="host" type="reference" use="optional"
    />
    <xs: attribute name="member" type="xs: string"
        use="optional"/>
    <xs: attribute name="type" type="xs: string" use="optional" />
    20  <xs: attribute name="name" type="xs: string" use="optional"
    />
        <! --这一实例的父母-->
        <! --这一实例的宿主-->
        <! --在父类型上的成员名字-->
    25  <! --这是一个实例的全部合格类型-->
        <! --当该实例被创建时，可以被填写的标示符的别名。
            这个名字对相同成员的所有实例来说必须是唯一的-->
    </xs: complexType>
    <!--我们可以怎样改变一个实例？ -->
    30  <xs: complexType name= "updateInstanceType" ><xs: sequence>
```

```
<xs: element name=" classSettings"
    type=" settings: settingValues" minOccurs=" 0" />
<xs: element name= "typeSettings"
    type=" settings: settingValues" minOccurs="0" />
5   <! --为类设置值-->
    <! --为类型设置值-->
</xs: sequence>
< xs: attribute name="id" type="reference" use="required"
/>
10  <xs: attribute name="parent", type= "reference" use="optional"
/>
    <xs: attribute name="host" type="reference" use="optional"
/>
    <xs: attribute name="member" type="xs: string"
15  use="optional"/>
    <xs: attribute name="type" type="xs: string" use="optional"/>
    <xs: attribute name="name" type="xs: string" use="optional"
/>
<!--限定SDM运行时间的唯一标示符。这是由t_u101SDM ? 运行时间产生的,
20 并且是不可改变的-->
<!--这一实例的父母-->
    <! --这一实例的宿主-->
        <! --在父类型上的成员的名字-->
        <!--这个实例的全部合格的类型-->
25    <! --当该实例被创建时， 标示符可以被加载的别名。
        该名称必须是同一成员的所有实例中唯一的。-->
</xs: complexType>
<xs: complexType name=" deleteInstanceType">
    <xs : attribute name="id" type="reference" use="required"/><xs : attribute
30  name="Option" type="deleteOptionType"
```

```
use="required" />
<!--作用域到SDM运行时间的唯一标示符。这是由SDM运行时间产生的,
是不可改变的_cf2>
</xs: complexType>
5 <xs: complexType name=" addConnectionType" >
<xs: attribute name="port" type="reference" use="required"
/>
<xs: attribute name="wire" type="reference" use="required"
/>
10 </xs: complexType>
<xs: complexType name="deleteConnectionType">
<xs: attribute name="port" type="reference" use="required"
/>
<xs: attribute name="wire" type="reference" use="required"
/>
15 </xs: complexType>
<! --引用可以是guid或路径-->
<xs:simpleType name="reference" >
<xs:union></xs:union>
20 </xs:simpleType>
<! --删除设置是: ? ? ? -->
<xs:simpleType name="deleteOptionType">
<xs:union></xs:union>
</xs:simpleType>
25 </xs:schema>
下面是类结构的一个例子。
<?xml version=" 1.0"encoding= "utf-8"? >
<xs:schema targetNamespace="urn: schemas-microsoft-com: sdmClasses"
xmlns="um : schemas-microsoft-com : sdmClasses " xmlns:names="urn :
30 schemas-
```

```

microsoft-com: sdmNames" xmlns: settings="urn: schemas-microsoft-
com: sdmSettings" xmlns:xs="http://www.w3.org/2001 /XML Schema"
elementFormDefault="qualified" version=" 0.7" id="sdmClasses">
<xs: import namespace="http://www.w3.org/2001/XMLSchema"/>
5 <xs: import namespace="urn:schemas-microsoft-com:sdmSettings"
schemaLocation="SDM7Settings.xsd"/>
<xs: import namespace="urn:schemas-microsoft-com: sdmNames"
schemaLocation="SDM7Names.xsd"/>
<!-- TODO [BassamT]: 规格化端口类引用, 端口类型引用, 在连接线类,
10 连接线类型和连接线成员上的端口成员-->
<! --TODO [BassamT]: 为验证添加键和键引用-->
<! --TODO [BassamT]: 为内嵌类型添加支持-->
<! --TODO[BassamT]: 取消 minOccurs和 maxOccurs-->
<! --TODD[BassamT]: “类”的新名字, 可能是“部署” -->
15 <! -- TODO [BassamT]: “宿主”的新名字, 可能是“提供者” -->
<! -- REVIEW [BassamT]: 我们可以合并在这一XSD中的端口, 组件, 连接
线类的定义。将用更多语义分析为代价减少冗长-->
<!-- CONSIDER [BassamT]: 用于象唯一, 协同定位, 在线情况的通常属性
机制-->
20 <! -- TODO [BassamT]: 绑定: 成员到组件成员-->
<! --TODD [geoffo]: 端口--它们是单独的吗? -->
<! --TODO [geoffo]: 授权--我们如何组合端口? -->
<! --TODD [geoffo] 在合适的地方加回〈任何〉 -->
<! --
25
=====
```

=====-->

```

<! --SDM 根元素-->
<! --
```

30 =====

===== >

```
<xs: element name=" sdmClasses">
<xs: complexType>
5   <xs: sequence>
    <xs: element name="import"
      type="names:import" minOccurs="0" maxOccurs="unbounded"/>
    <xs: element name=" information"
      type="information"minOccurs="0" />
10  <xs: element name="portClasses"
      minOccurs=" 0" >
    <xs: complexType>
    <xs: sequence>
    <xs:element
      name="portClass" type="portClass" maxOccurs="unbounded"/>
15  </xs: sequence>
    </xs: complexType>
    </xs: element>
    <xs: element name=" componentClasses"
      minOccurs=" 0" >
20  <xs: complexType>
    <xs: sequence>
    <xs:element
      name="componentClass" type=" componentClass" maxOccurs="unbounded"/>
25  </xs: sequence>
    </xs: complexType>
    </xs: element>
    <xs: element name="protocols"
      minOccurs="0" >
30  <xs: complexType>
```

```
<xs: sequence>
<xs: element
  name="protocol" type="protocol" maxOccurs="unbounded"/>
</xs: sequence>
5 </xs: complexType>
</xs: element>
<xs: element name="hostRelations"
  minOccurs="0">
<xs: complexType>
10 <xs: sequence>
<xs:element
  name=" installer" type="installer" maxOccurs="unbounded"/>
<xs: element
  name="hostRelation" type="hostRelation" maxOccurs="unbounded"/>
15 </xs: sequence>
</xs: complexType>
</xs: element>
</xs: sequence>
<xs:attributeGroup ref="names:namespaceIdentity"/>
20 </xs: complexType>
</xs: element>
<! --SDM 类型库信息-->
<xs: complexType name=" information">
<xs: annotation>
25   <xs : documentation>与 SDM 类型库有关的人类可读信息。 </xs :
documentation>
</xs: annotation>
<xs: sequence>
<xs: element name=" friendlyName" type="xs: string"
30   minOccurs="0" />
```

```
<xs: element name="companyName" type="xs: string"
minOccurs="0" />
<xs: element name="copyright" type="xs: string"
minOccurs="0" />
5 <xs: element name="trademark" type="xs: string"
minOccurs="0" />
<xs: element name="description" type="xs: string"
minOccurs="0" />
<xs: element name="comments" type="xs: string"
10 minOccurs="0" />
</xs: sequence>
</xs: complexType>
<! --  

15 =====>  

=====>  

<! --类-->  

<! --  

20 =====>  

=====>  

<xs: complexType name="baseClass" >
<xs: sequence>
<xs: element name="deploymentSchema"
25 type="settings: deploymentSchema" minOccurs="0" />
<xs: element name="settingSchema"
type="settings: settingSchema" minOccurs="0"/>
<!—类如何被部署的XSD模式-->
<! --设置模式-->
30 </xs: sequence>
```

```

<xs: attribute name="name" type="xs: string" use="required"
/>
<xs: attribute name=" layer" type="xs: string" use="required"
/>
5   <! --回顾[BassamT]这些层仅仅是为了得益于工具，或者它们是严格的以
SDM模式实施？存在情况，其中来自不同层的混合组件是有意义的。例如，过
滤器组件可以是一个组件元类型，该类型被位于层3的ISA服务器所容纳。然而，
我们想要在层2中使用过滤器元类型（Filter meta-type），-->
</xs: complexType>
10  <! --端口类-->
<xs: complexType name= "portClass" >
<xs: complexContent>
<xs: extension base= "baseClass" />
</xs: complexContent>
15  </xs: complexType>
<! --组件类-->
<xs: complexType name=" componentClass" >
<xs: complexContent>
<xs: extension base="baseClass" >
20  <xs: sequence>
<xs: element
name= "portClassesAllowed" minOccurs=" 0" >
<xs: complexType>
<xs: sequence>
25  <xs: element
name="portClassRef" minOccurs="0" maxOccurs="unbounded"/>
</xs: sequence>
<xs: attribute
name=" closed" type="xs: boolean" use="optional" default="true" />
30  <! --允许的端口是否是封闭式列表

```

-->

<! --如果这个值是“真 (true)”，那么该端口列表是不可扩展的。如果这一值是“假 (false)”，那么该端口列表是无底限的，所列出的端口将被认为是强制性的。-->

5 </xs: complexType>

 </xs: element>

<! --这一部分将在该组可允许端口上指定一组约束，其可以显现在这一元类型的组件类型上。-->

 </xs: sequence>

10 </xs: extension>

 </xs: complexContent>

 </xs: complexType>

 <xs: complexType name= "portClassRef">

 <xs: attribute name="name" type="xs: string" use="required"

15 />

 <xs: attribute name= "required" type="xs: boolean"

 use="required"/>

 <xs: attribute name=" singleton" type="xs: boolean"

 use=" required' />

20 <! --唯一意思是在父母范围内仅仅可能存在这一端口的一个实例-->

 </xs: complexType>

 <! --

25 =====

 ==>

 <! --关系-->

 <! --

30 =====

=====

=-->

<xs: complexType name= "relation" >

<xs: attribute name="name" type="xs: string" use="required"

5 >

<xs: attribute name=" installer" type="xs: string"

use="optional" />

</xs: complexType>

<! -协议是一个或多个端口类之间的关系-->

10 <xs: complexType name="protocol">

<xs: complexContent>

<xs: extension base="relation">

<xs: sequence>

<xs: element name="portClassRef

15 type="portC lassRef' maxOccurs="unbounded"/>

</xs: sequence>

</xs: extension>

</xs: complexContent>

20 </xs: complexType>

<! -定义两个类之间的宿主关系-->

<xs: complexType name="hostRelation" >

<xs: complexContent>

<xs: extension base="relation">

25 <xs: attribute name="classRef type="xs: string"

use="required"/>

<xs: attribute name="hostClassRef"

type="xs: string" use= "required"/>

</xs: extension>

30 </xs: complexContent>

```

</xs: complexType>
<! --安装程序类型标识代码负责示例化关系-->
<xs: complexType name=" installer">
<xs: sequence>
5   <xs: element name=" binary" type="xs: string"/>
</xs: sequence>
<xs: attribute name=" codeType" type="xs: string"
use="required' />
<xs: attribute name="name" type="xs: string" use="required"
10   />
</xs: complexType>
</xs: schema>

```

以下是一部署单元的示例结构。

```

15 <?xml version="1 . 0" encoding="UTF-8" ? >
<xs: schema targetNamespace="urn: schemas-microsoft-com: sdmSDU"
xmlns:xs="http://www.w3 . org/2001/XMLSchema" xmlns:names="urn :
schemas-
microsoft-com: sdmNames" xmlns="urn: schemas-microsoft-com: sdmSDU"
20   elementFormDefault="qualified" version="0. 7" id="sdmSDU">
<xs: import namespace="http://www.w3. org/2001/XMLSchema' />
<xs: import namespace="urn:schemas-microsoft-com: sdmNames"
schemaLocation="SDM7Names.xsd"/>
<! --sdm部署单元输入一个或多个sdm类型文件，该sdm类型文件包括从
25   已输入文件到类型子组的映射-->
<xs: element name=" sdmDeploymentUnit">
<xs: annotation>
<xs: documentation>
该 sdu 约束一SDM类型到它们的实现的映射。
30 </xs: documentation>

```

```

</xs: annotation>
<xs: complexType>
<xs: sequence>
<xs: element name=' import'
5      type="names: import" minOccurs=' 0" maxOccurs="unbounded"/>
<xs: element name=' implementation'
type=" implementationMap" minOccurs="0" maxOccurs="unbounded"/>
</xs: sequence>
</xs: complexType>
10 </xs: element>
<! --这一部署单元的描述-->
<xs: complexType name=" deploymentDescription">
<xs: attribute name="name" type="xs: string' />
<xs: attribute name=" dateCreated" type="xs: string' />
15 <xs: attribute name=" creator" type="xs: string"/>
</xs: complexType>
<! --从一类型到该类型实现的映射-->
<xs: complexType name=" implementationMap" >
<xs: sequence>
20 <xs: element name="version' type="xs: string"
minOccurs=" 0" maxOccurs="unbounded"/>
</xs: sequence>
<xs: attribute name="type' type="xs: string' />
<xs: attribute name="path' type="xs: string' />
25 </xs: complexType>
</xs: schema>

```

以下是实例的一示例结构。

```

<?xml version=" 1 . 0" encoding= "utf-8" ? >
30 <xs: schema targetNamespace="urn: schemas-microsoft-com: sdmInstances"

```

```
    xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns: settings="urn:  
schemas-  
microsoft-com : sdmSettings " xmlns="urn : schemas-microsoft-com :  
sdmInstances"  
5      elementFormDefault="qualified" version="0.7" id=" sdmInstances" >  
        <xs: import namespace="http://www.w3.org/2001/XMLSchema"/>  
        <xs: import namespace="urn: schemas-microsoft-com: sdmSettings"  
            schemaLocation=' SDM7Settings.xsd'/>  
        <xs: element name=" sdmInstances" >  
10       <xs: complexType>  
            <xs: sequence>  
                <xs: element name="import" type=" import"  
                    minOccurs="0" maxOccurs="unbounded"/>  
                <xs: element name="portInstances"  
15                    minOccurs="0" >  
                    <xs: complexType>  
                        <xs: sequence>  
                            <xs: element  
                                name="portInstance" type="portInstance" minOccurs="0"  
                                maxOccurs="unbounded"/>  
                            </xs: sequence>  
                        </xs: complexType>  
                    </xs: element  
                    <xs: element name="wireInstances"  
20                    minOccurs="0" >  
                    <xs: complexType>  
                        <xs: sequence>  
                            <xs: element  
                                name="wireInstance" type="wireInstance" minOccurs="0"  
                                maxOccurs="unbounded"/>  
25
```

```
</xs: sequence>
</xs: complexType>
</xs: element>
<xs: element name="componentInstances"
5      minOccurs="0" >
<xs: complexType>
<xs: sequence>
<xs: element
name="componentInstance" type="componentInstance" minOccurs="0"
10     maxOccurs="unbounded" />

<xs: element
name="compoundComponentInstance" type="compoundComponentInstance"
minOccurs="0" maxOccurs="unbounded"/>
15
</xs: sequence>
</xs: complexType>
</xs: element>
</xs: sequence>
</xs: complexType>
20
</xs: element>
<xs: complexType name="import">
<xs: attribute name="alias" type="xs: string" use="required"/>
<xs: attributeGroup ref="identity"/>
</xs: complexType>
25
<!————实例模式————>
<xs: complexType name="instanceBase" >
<xs: sequence>
<xs: element name="classSettings"
type="settings: settingValues" minOccurs="0" />
30
<xs: element name="typeSettings"
```

```
type="settings: settingValues" minOccurs="0" />
<! --为类设置值-->
<! --为类型设置值-->
</xs: sequence>
5 <xs: attribute name=" id" type="guid" use="required"/>
<xs: attribute name="parent" type=" guid" use="optional" />
<xs: attribute name="host" type="guid" use="optional" />
<xs: attribute name="member" type="xs: string"
use="optional" />
10 <xs: attribute name="type" type="xs: string" use="required"/>
<xs: attribute name="name" type="xs: string" use="optional"
/>
<! --到SDM运行时间的SDM运行时间的唯一标示符。这是由SDM 运行时
间产生的， 并且是不可改变的-->
15 <! --这一实例的父母-->
<! --这一实例的宿主-->
<! --在父类型上的成员的名字-->
<! --这是一个实例的全部合格的类型-->
<! --当该实例被创建时， 标示符可以被加载的别名。
该名称必须是同一成员的所有实例中唯一的。 -->
20 </xs: complexType>
<xs: complexType name=" componentInstance">
<xs: complexContent>
<xs: extension base=" instanceBase" >
25 <xs: sequence>
<xs: element name="portInstances" >
<xs: complexType>
<xs: sequence>
<xs: element
30 name= "portInstance" type=" instanceRef"/>
```

```
<! --我所拥有的端口实例-->
</xs: sequence>
</xs: complexType>
</xs: element>
5   </xs: sequence>
</xs: extension>
</xs: complexContent>
</xs: complexType>
<xs: complexType name=" compoundComponentInstance" >
10  <xs: complexContent>
<xs: extension base=" instanceBase" >
<xs: sequence>
<xs: element name= "portInstances" >
<xs: complexType>
15  <xs: sequence>
<xs: element
name= "portInstance" type=" instanceRef"/>
<! --我授权的端口实例-->
</xs: sequence>
20  </xs: complexType>
</xs: element>
<xs: element
name=' componentInstances" >
<xs: complexType>
25  <xs: sequence>
<xs: element
name=" componentInstance" type=" instanceRef"/>
</xs: sequence>
</xs: complexType>
30  </xs: element>
```

```
<xs: element name="wireInstances" >  
<xs: complexType>  
  
    <xs: sequence>  
        5      <xs: element  
                name="wireInstance" type=" instanceRef"/>  
            </xs: sequence>  
        </xs: complexType>  
    </xs: element>  
  
    10     </xs: sequence>  
    </xs: extension>  
    </xs: complexContent>  
    </xs: complexType>  
  
    <xs: complexType name="portInstance" >  
    15     <xs: complexContent>  
    <xs: extension base=' instanceBase'>  
    <xs: sequence/>  
    </xs: extension>  
    </xs: complexContent>  
  
    20     </xs: complexType>  
    <xs: complexType name=" wireInstance" >  
    <xs: complexContent>  
    <xs: extension base=" instanceBase" >  
    <xs: sequence>  
    <xs: element name="portInstances" >  
    25     <xs: complexType>  
    <xs: sequence>  
    <xs: element  
            name="portInstance", type=" instanceRef"/>  
    30     <! --我已经连接的端口-->
```

```

</xs: sequence>
</xs: complexType>
</xs: element>
</xs: sequence>
5 </xs: extension>
</xs: complexContent>
</xs: complexType>
<xs: complexType name="instanceRef">
<xs: attribute name="uniqueId" type="xs: string"/>
10 </xs: complexType>
<!—————类型示例—————
-->
<xs: simpleType name="fourPartVersionType" >
<xs: annotation>

```

15

<xs: documentation>四部分版本号，其中该端位于0-65535范围内 </xs:
documentation>

```

</xs: annotation>
20 <xs: restriction base="xs: string">
<xs: pattern value=" (0| [1-5] [0-9] {0,4}) [7-9][0-
9] {0, 3} |6[0-4][0-9] {0, 3}) 6 [6-9] [0-9] {0,2} } 65 } 65 [0-4] [0-9]
{0, 2} } 65[6-9] [0-
9]? } 655 } 655[0-2][0-9]? 1655[4-9]} 6553[0-5] ? ). (0) [1-5] [0-9] {0,4
25 } } [7-9][0-
9]{0,3} } 6[0-4][0-9]{0,3} } 6[6-9][0-9]{0,2} 165165[0-4][0-9]{0,2} } 65[6-9][0-
9]? } 6551655[0-2][0-9]? 1655[4-9]} 6553[0-5] ? ). (0) [1-5] [0-9] {0,4
} } [7-9][0-
9] {0, 3} } 6[0-4][0-9] {0, 3} } 6[6-9][0-9] {0, 2} |65 } 65[0-4][0-9] {0,2
30 } ! 65 [6-9] [0-
```

```
9]? 1655|655[0-2][0-9]? |655[4-9]} 6553[0-5]? ). (0 {[1--5][0-9] {0,4} |[7-9
] [0-
9]{0,3}} 6[0-4][0-9]{0,3}} 6[6-9][0-9]{0,2}|65165[0-4][0-9]{0,2} 165[6-9][0-
9]? |655|655 [0-2] [0-9]? |655[4-9]} 6553[0-5]? )” >
5 </xs:restriction>
</xs: simpleType>
<xs: simpleType name="publicKeyTokenType" >
<xs: annotation>
<xs: documentation> (公钥标记) Public Key Token: 大小为16位16进制数
10 </xs: documentation>
</xs: annotation>
<xs: restriction base="xs: string">
<xs: pattern value=' ([0-9]| [a-f] [A-F]) {16}' />
</xs:restriction>
15 </xs: simpleType>
<xs: attributeGroup name=" identity" >
<xs: attribute name="name" type="xs: string" use="required"
/>
<xs: attribute name="version" type= "fourPartVersionType"
20 use="required"/>
<xs: attribute name="publicKeyToken"
type= "pub licKeyTokenType" use=" optional" />
</xs: attributeGroup>
<xs: simpleType name="guid">
25 <xs: restriction base="xs: string" >
<xs:pattern value=" [0-9a-fA-F] {8} -[0-9a-fA-F] {4} -[0-
9a-fA-F] {4} - [0-9a-fA-F] {4} - [0-9a-fA-F] {12}" />
</xs: restriction>
</xs: simpleType>
30 </xs: schema>
```

以下是映射的一示例结构。

```
<?xml version="1 . 0" encoding=" utf-8" ? >
<xs: schema targetNamespace="urn:schemas-microsoft-com: sdmMapping"
  xmlns: xs="http://www.w3 . org/2001/XMLSchema" xmlns: names="urn:
5   schemas-
    microsoft-com : sdmNames "  xmlns : schemas-microsoft-com:
sdmMapping"
    elementFormDefault="qualified" version=" 0. 7" id="sdmMapping">
    <!-- REVIEW [BassamT]: 我们是否允许到在相同组合组件中的组件的映射
10  ? -->
    <xs: import namespace="urn: schemas-microsoft-com: sdmNames"
      schemaLocation="SDM7Names.xsd"/>
    <xs: element name=" logicalPlacement">
      <xs: annotation>
15      <xs: documentation>
      这一文件包括SDM 成员之间的映射信息。
      映射以首先绑定外部组合组件，然后它的成员，等等的方式（fashion）被
        构造在外部。
      </xs: documentation>
20      <xs: annotation>
      <xs: complexType>
      <xs: sequence>
        <xs: element name="import"
          type="names: import" minOccurs=' 0" maxOccurs="unbounded"/>
25      <xs: element name="placement" minOccurs=' 0"
          maxOccurs="unbounded">
        <xs: complexType>
        <xs: sequence>
          <xs: element
30            name="memberBinding" type="memberBinding" maxOccurs="unbounded"/>
```

```
<xs: element  
name="wireBinding" type="wireBinding" minOccurs="0"  
maxOccurs="unbounded"/>  
</xs: sequence>  
5 <xs: attribute  
name="sourceComponentType" type="xs: string" />  
<xs: attribute  
name="targetComponentType" type="xs: string" />  
<xs: attribute name="name"  
10 type="xs: string"/>  
</xs: complexType>  
</xs: element>  
</xs: sequence>  
</xs: complexType>  
15 </xs: element>  
<! --成员绑定可以是：  
I . 组合组件成员—在组合组件成员情况下我们绑定所有组合组件成员和  
连接线  
20 2 . 一单一组件成员—在单一组件成员情况下我们绑定组件和它们的端口  
3 . 一端口成员—在端口成员情况下我们将端口成员绑定到端口，而且不存  
在进一步的绑定  
-->  
25 <xs: complexType name="memberBinding">  
<xs: sequence>  
<xs: element name="memberBinding"  
type="memberBinding" minOccurs="0" maxOccurs="unbounded"/>  
<xs: element name="wireBinding" type="wireBinding"  
minOccurs="0" maxOccurs="unbounded" />  
30 </xs: sequence>
```

```
<xs: attribute name=" sourceMember" type="xs: string"
use= "required"/>
<! --如果一目标成员没有被提供，那么该组件必须是组合组件，而且它的
成员将被绑定到它们的父母所绑定的组合组件的成员上
5    如果一目标成员被提供，而且我们绑定一组合组件，那么在原组合组件上
的端口必须能够被绑定到目标组合组件的端口上-->
<xs: attribute name="targetMember" type="xs: string"
use="optional" />
</xs: complexType>
10   <! --连接线被绑定到目标组合组件中的一路径上。
这一路径由端口，连接线和组件实例组成-->
<xs: complexType name="wireBinding">
<xs: sequence>
<xs: element name="path">
15   <xs: complexType>
<xs: sequence>
<xs: element name=" element"
maxOccurs="unbounded">
<xs: complexType>
20   <xs: attribute
name="name" type="xs: string' />
</xs: complexType>
</xs: element>

25   </xs: sequence>
</xs: complexType>
</xs: element>
</xs: sequence>
<xs: attribute name=' sourceWire" type="xs: string"/>
30   </xs: complexType>
```

```
<! --入口-->
</xs: schema>
```

以下是名字的示例结构。

```
5   <?xml version=" 1 . 0 " encoding="UTF-8" ? >
    <xs: schema targetNamespace="urn: schemas-microsoft-com: sdmNames"
      xmlns:xs="http://www.w3. org/2001/XMLSchema" xmlns="urn: schemas-
      microsoft-com: sdmNames" elementFormDefault="qualified" version=" 0. 7 "
      id=" sdmNames" >
10    <xs: import namespace="http: //www.w3. org/2001/XMLSchema"/>
    <! --入口给其他SDM文件创建一别名-->
    <xs: complexType name=" import">
      <xs: attribute name=" alias" type="xs:NCName"
        use="required" />
15    <xs: attribute name=" location" type="xs:NCName"
        use=' optional' />
      <xs: attributeGroup ref "Identity"/>
    </xs: complexType>
    <!--类和类型文件由名字，版本和公钥来标识-->
20    <xs: attributeGroup name=" Identity">
      <xs: attribute name="name" type="xs: string" use=" required"
        />
      <xs: attribute name= "version" type=" fourPartVersionType"
        use="required" />
25    <xs: attribute name="publicKeyToken"
        type="publicKeyTokenType" use="optional" />
    </xs: attributeGroup>
    <xs: attributeGroup name="namespaceldentity">
      <xs: attributeGroup ref "Identity"/>
30    <xs: attribute name=" signature" type="xs: string"
```

```

use=" optional" />
<xs: attribute name="publicKey" type="xs: string"
use="optional" />
</xs: attributeGroup>
5 <!--单一版本号-->
<xs: simpleType name=' fourPartVersionType">
<xs: annotation>
<xs: documentation>四部分版本号，其中该段位于0-65535范围内 </xs:
documentation>
10 </xs: annotation>
<xs: restriction base="xs: string">
<xs: pattern value="(0| [1-5] [0-9] {0,4} |[7-9][0-
9]{0,3})6[0-4][0-9]{0,3}16[6-9][0-9]{0,2} {65|65[0-4][0-9]{0,2} {65[6-9][0-
9]? } 655 } 655[0-2] [0-9]? |655 [4-9]} 6553 [0-5] ? ). (0 {[1-5][0-9]
15 {0, 4} { [7-9] [0-
9] {0, 3}} 6[0-4] [0-9] {0, 3}] 6 [6-9] [0-9] {0,2}] 65] 65[0-4] [0-9
] {0, 2}] 65[6-9] [0-
9]? |655|655[0-2][0-9]? 1655[4-9]16553[0-5]? ). (0 )[1-5][0-9]{0,4})}[7-9][0-
9] {0, 3} {6 [0-4] [0-9] {0, 3} |6[6-9] [0-9] {0,2}} 65 {65[0-4][0-9] {
20 0,2}) 65 [6-9] [0-
9]? {655 {655[0-2] [0-9]? } 655[4-9]16553[0-5] ? ). (0 ) [1-5] [0-9] {
0,4}) [7-9][0-
9]{0,3}16[0-4][0-9]{0,3}16[6-9][0-9]{0,2}165165[0-4][0-9]{0,2})65[6-9][0-
9]? } 655|655 [0-2] [0-9] ? |655 [4-9]} 6553[0-5] ? )” />
25 </xs:restriction>
<xs: simpleType>
<! --公钥用于校验有标记的文档段-->
<xs: simpleType name="publicKeyTokenType" >
30 <xs: annotation>

```

```
<xs: documentation>Public Key Token (公钥标记): 大小为16位16进制数  
</xs: documentation>  
<xs: annotation>  
<xs: restriction base="xs: string" >  
5   <xs:pattern value=' ([0-9]|[a-f]|[A-F]) {16}' />  
</xs:restriction>  
</xs: simpleType>  
</xs: schema>
```

10 以下是用于设置的示例结构。

```
<?xml version=" 1 . 0" encoding="utf-8" ? >  
<xs: schema targetNamespace="urn: schemas-microsoft-com: sdmSettings"  
xmlns: xs="http://www.w3. org/2001/XMLSchema" xmlns="urn: schemas-  
microsoft-com: sdmSettings" elementFormDefault=" qualified" version="0. 7"  
15 id="sdmSettings" >  
<xs: import namespace="http://www.w3. org/2001/XMLSchema"/>  
<! --设置模式, 值和约束-->  
<xs: complexType name=" openSchema">  
<xs: sequence>  
20 <xs: any namespace="##other" processContents="lax"  
/>  
</xs: sequence>  
<xs: complexType>  
<xs: complexType name=" settingSchema">  
25 <xs: sequence>  
<xs: any  
namespace="http://www.w3. org/2001/XMLSchema" processContents="skip"  
minOccurs="0" maxOccurs="unbounded" />  
</xs: sequence>  
30 </xs: complexType>
```

```
<xs: complexType name=" settingValues" >
<xs: sequence>
<xs: any namespace="#other" processContents=" lax"
/>
5   </xs: sequence>
</xs: complexType>
<! --约束-->
<xs: attributeGroup name="testAttributes" >
<xs: attribute name="path" type="xs: string"/>
10  <xs: attribute name=" ifNullPath" type=" ifNullPath"/>
<xs: attribute name="error" type="xs: int"/>
<xs: attribute name=" errorDesc" type="xs: string"/>
</xs: attributeGroup>
<xs: complexType name=" simpleTest">
15  <xs: attributeGroup ref--"testAttributes" />
</xs: complexType>
<xs: complexType name=" settingConstraints" >
<xs: sequence>
<xs: element name= "mustExist" type=" simpleTest"
20  minOccurs=" 0" max0ccurs="unbounded"/>
<xs: element name= "mustNotExist" type=" simpleTest"
minOccurs=" 0" max0ccurs="unbounded"/>

25  <xs: element name="ifExists" type="nestedTest"
minOccurs="0" max0ccurs="unbounded"/>
<xs: element name=" ifNotExists" type="nestedTest"
minOccurs="0" max0ccurs="unbounded"/>
</xs: sequence>
</xs: complexType>
30  <xs: complexType name="nestedTest' >
```

```
<xs: sequence>
<xs: element name="mustExist" type="simpleTest"
minOccurs="0" maxOccurs="unbounded"/>
<xs: element name="mustNotExist" type="simpleTest"
5 minOccurs="0" maxOccurs="unbounded"/>
<xs: element name="ifExists" type="nestedTest"
minOccurs="0" maxOccurs="unbounded"/>
<xs: element name="ifNotExists" type="nestedTest"
minOccurs="0" maxOccurs="unbounded"/>
</xs: sequence>
10 <xs: attributeGroup ref="testAttributes" />
</xs: complexType>
<xs: complexType name="deploymentSchema' ">
<xs: sequence>
<xs: any namespace="# other" processContents="lax"
15 />
</xs: sequence>
</xs: complexType>
<xs: complexType name=' deploymentValues" >
<xs: sequence>
20 <xs: any namespace="#other" processContents="lax"
/>
</xs: sequence>
</xs: complexType>
<! —————单一类型
—————>
25 <xs: simpleType name=" ifNullPath" >
<xs:restriction base="xs: string" >
<xs: enumeration value="skip" />
<xs: enumeration value=" override"/>
```

```

<xs: enumeration value="returnError"/>
</xs: restriction>
</xs: simpleType>
</xs: schema>

```

5

以下是类型的一示例结构。

```

<?xml version="1 . 0" encoding="utf-8" ? >
<xs: schema targetNamespace="urn: schemas-microsoft-com: sdmTypes"
10      xmlns:xs="http://www.w3. org/2001/XMLSchema" xmlns="urn: schemas-
      microsoft-com: sdmTypes" xmlns: names="urn: schemas-microsoft-
      com: sdmNames" xmlns: settings="urn: schemas-microsoft-com: sdmSettings"
      elementFormDefault='qualified' version="0. 7" id="sdmTypes" >
<xs: import namespace="http: //www.w3. org/2001/XMLSchema"/>
<xs: import namespace="urn:schemas-microsoft-com: sdmSettings"
15      schemaLocation=" SDM7Settings.xsd"/>
<xs: import namespace="urn:schemas-microsoft-com: sdmNames"
      schemaLocation="SDM7Names. xsd"/>

```

20 <!-- TODO [BassamT]: 规格化端口类引用，端口类型引用，在连接线类，
连接线类型和连接线成员上的端口成员-->

```

<! --TODO [BassamT]: 为验证添加键和键引用-->
<! --TODO [BassamT]: 为内嵌类型添加支持-->
<! --TODO[BassamT]: 取消 minOccurs和 maxOccurs-->
<! --TODD[BassamT]: “类”的新名字，可能是“部署”-->
25 <! -- TODO [BassamT]: “宿主”的新名字，可能是“提供者”-->
<! -- REVIEW [BassamT]: 我们可以合并在这一XSD中的端口，组件，连接
      线类的定义。将用更多语义分析为代价减少冗长-->
<! -- CONSIDER [BassamT]: 用于象唯一，协同定位，在线情况的通常属性
      机制-->
30 <! -- TODO [BassamT]: 绑定：成员到组件成员-->

```

```
>
<! --TODD [geoffo]: 端口--它们是单独的吗? -->
<! --TODO [geoffo]: 授权--我们如何组合端口? -->
<! --TODD [geoffo] 在合适的地方加回〈任何〉 -->
5   <! --
=====

=====
=====

-->
10  <! --SDM 根元素-->
<! --
=====

=====
=====

15  -->
<xs: element name=" sdmTypes" >
<xs: annotation>

<xs: documentation>SDM 根元素。它是一SDM类型的容器。</xs:
20  documentation>
</xs: annotation>
<xs: complexType>
<xs: sequence>
<xs: element name="import"
25  type="names: import" minOccurs=' 0' maxOccurs="unbounded"/>
<xs: element name=" information"
type=" information" minOccurs="0" />
<xs: element name="portTypes"
minOccurs="0" >
30  <xs: complexType>
```

```
<xs: sequence>
<xs: element
  name="portType" type="portType" minOccurs="0" maxOccurs="unbounded" />
</xs: sequence>
5 </xs: complexType>
</xs: element>
<xs: element name="componentTypes"
  minOccurs="0" >
<xs: complexType>
<xs: sequence>
10 <xs: element
  name="componentType" type="componentType" minOccurs="0"
  maxOccurs="unbounded" />
<xs: element
  name="compoundComponentType" type="compoundComponentType"
  minOccurs="0" maxOccurs="unbounded"/>
15 </xs: sequence>
</xs: complexType>
</xs: element>
</xs: sequence>
20 <xs: attributeGroup ref="names:namespaceIdentity" />
</xs: complexType>
</xs: element>
<! --SDM type library information-->
25 <xs: complexType name="information" >
<xs: annotation>
<xs: documentation>人们关于SDM类型库的可读信息。
</xs: documentation>
</xs: annotation>
30 </xs: sequence>
```

```
<xs: element name=" friendlyName" type="xs: string"
minOccurs=" 0" />
<xs: element name=" companyName" type="xs: string"
5      minOccurs=" 0" />
<xs: element name=" copyright" type="xs: string"
minOccurs="0" />
<xs: element name="trademark" type="xs: string"
minOccurs=" 0" />
10    <xs: element name=" description" type="xs: string"
minOccurs=" 0" />
<xs: element name="comments" type="xs: string"
minOccurs="0" />
</xs: sequence>
15    </xs: complexType>
<! --
```

20 <! --组件，端口和连接线类型的基本 complexType-->

<! --

25 <xs: complexType name="baseType" >
<xs: annotation>
<xs: documentation>组件类型和端口类型的基本类型。
</xs: documentation>
</xs: annotation>
30 <xs: sequence>

?

```

<xs: element name=" deployment"
  type=" settings: deploymentValues" minOccurs="0" />
<xs: element name="settings"
  type="settings: settingValues" minOccurs=' 0' />
5 <xs: element name=' settingSchema'
  type="settings: settingSchema" minOccurs=" 0" />
<xs: element name="hostConstraints"
  type="hostConstraints" minOccurs="0" />
<xs: element name="hostedClasses"
  type="hostedClassesList" minOccurs=' 0' />
10 <! --部署段包括类型的部署指示。在‘单元’上这一部署的模式被指定,
-->
```

<! -- 这一组件的设置。这些是在单元上所指定的模式为根据的。-->

15 <! --设置模式。新的设置模式可以由类型公开。
这一模式的值被设置在这一类型的成员上。还要注意，我们将支持以下在
单元上的设置的设置值。-->

<! --这一部分包括容纳这一单元的宿主设置上的任意约束。注意，每一单
元可以存在多个宿主，而且我们可以支持在每一个上的约束。-->

20 <! --这一部分包括容纳但愿的列表和任意类设置上的约束。
该约束是根据类模式来指定的。注意，可以存在这里所容纳的多个类。-->

```

</xs: sequence>
<xs: attribute name=' class" type="xs: string" use="required"
/>
<xs: attribute name="name" type="xs: string" use="required"
/>
</xs: complexType>
<! --
```

————>

<! --约束-->

<! --

5 —————

————>

<xs: complexType name="hostConstraints" >

<xs: annotation>

<xs: documentation>宿主约束包括相对于类的约束，该类可以容纳这一组
10 件或端口类型。

</xs: documentation>

</xs: annotation>

<xs: sequence>

<xs: element name="hostConstraint">

15 <xs: complexType> .

<xs: sequence>

<xs: element name="constraints" type="settings: settingConstraints" />

20 </xs: sequence>

<xs: attribute name="host" type="xs: string" use= "required"/>

<! --组件单元的名字是宿主。可以包括一别名，例如 "otherSDM:
myPortType"-->

25 </xs: complexType>

</xs: element>

</xs: sequence>

</xs: complexType>

<xs: complexType name= "hostedClassesList">

30 <xs: annotation>

<xs: documentation>这些是相对于这一类型可以容纳的类的约束</xs:
documentation>

</xs: annotation>

<xs: sequence>

5 <xs: element name="hostedClass"
maxOccurs="unbounded">

<xs: complexType>

<xs: sequence>

<xs: element name="constraints"
10 type="settings: settingConstraints" />

</xs: sequence>

<xs: attribute name="class"
type="xs: string" use="required"/>

<! --我们可以容纳的组件类的名字。

15 可以包括一别名，例如 "otherSDM: myPortType"-->

</xs: complexType>

</xs: element>

</xs: sequence>

</xs: complexType>

20 <! --

=====>

<! --端口-->

25

<! --

30 =====>

```
<xs: complexType name="portType">
<xs: complexContent>
<xs: extension base="baseType">
<xs: sequence>
5   <xs: element name="portConstraints"
type="portConstraints" minOccurs="0" maxOccurs="unbounded"/>
</xs: sequence>
</xs: extension>
</xs: complexContent>
</xs: complexType>
<xs: complexType name="portConstraints" >
<xs: annotation>
<xs: documentation>These are constraint against the
classes that this type can host</xs: documentation>
15  </xs: annotation>
<xs: sequence>
<xs: element name="portConstraint">
<xs: complexType>
<xs: sequence>
<xs: element name="constraints"
20  type="settings: settingConstraints" />
</xs: sequence>
<xs: attribute name="portClass"
type="xs: string" use="required"/>
<xs: attribute name="portType"
25  type="xs: string" use="optional" />
<xs: attribute name="minOccurs"
type="xs: int" use="optional" />
<xs: attribute name="maxOccurs"
type="xs: int" use="optional" />
30
```

```
<xs: attribute name="visible"
    type="xs: boolean" use="optional" />
    <! -- 在这里端口类型允许你绑定一没有被设置公开的属性的端口类型。在
        这一情况下，设置将不足以允许适当的端口被标识-->
5      <! -- 可见的标识了是否该应用程序想要看见与这一约束匹配的端口。一
        应用程序可以仅仅选择来约束它连接的端口，而不想要它们的终点的连接值-->
    </xs: complexType>
    </xs: element>
    <! --在这里我们很可能需要一机制来允许端口标识操作上的端口终点的设
10     置。即，我们可以绑定 X,Y 或 Z，但是我必须绑定至少一个-->
    </xs: sequence>
    </xs: complexType>
    <! --
```

```
15   ======>
=====
```

```
<! --组件-->
<! --
```

```
20   ======>
```

```
25   ======>
<xs: complexType name="componentType" >
    <xs: complexContent>
        <xs: extension base="baseType" >
            <xs: sequence>
                <xs: element name="ports"
                    type="portsList" minOccurs="0" />
            </xs: sequence>
            <! --排它的特性指示是否在一在下一层的该类型的实例可以容纳其它类型的
```

实例

-浅 (shallow) 意思是仅仅由实例所容纳的实例是这一类型的实例

-深 (deep) 意思是宿主还必须被标注为排它的-->

```
<xs: attribute name="exclusive" type="depth"
```

5 use="optional" default="notSet"/>

```
</xs: extension>
```

```
</xs: complexContent>
```

```
</xs: complexType>
```

```
<xs: complexType name="portsList">
```

10 <xs: sequence>

```
<xs: element name="port" minOccurs="0"
```

```
maxOccurs="unbounded">
```

```
<xs: complexType>
```

```
<xs: sequence>
```

15

```
<xs: element name="settings"
```

```
type="settings: settingValues" minOccurs="0" />
```

<! --成员设置值。这些是以在类型上所指定的设置模式为根据的-->

```
</xs: sequence>
```

20 <xs: attribute name="name"

```
type="xs: string" use="required"/>
```

```
<xs: attribute name="type"
```

```
type="xs: string"/>
```

```
</xs: complexType>
```

25 . </xs: element>

```
</xs: sequence>
```

```
</xs: complexType>
```

```
<! --
```

30 ==

```
=====>
<! --组合组件类型-->
<! -->

=====
5 =====
=====>

<xs: complexType name=" compoundComponentType">
<xs: sequence>
<xs: element name="components" type=" components"
minOccurs="0" />
10 <xs: element name="wires" type="wires"
minOccurs="0" />
<xs: element name=" delegatePorts"
type=" delegatePorts" minOccurs="0" />
<xs: element name=" delegateHostedClasses"
type=" delegateHostedC lasses" minOccurs="0" />
15 <! --授权端口-->
<! --授权宿主。这些允许一组合组件仅仅扮演一单一组件。-->
</xs: sequence>
<xs: attribute name="name" type="xs: string" use="required"
/>
20 <! --浅 (shallow) 协同定位意思是这一组合组件的成员的实例将被放置在
下一层的相同的宿主实例上。
--深 (deep) 协同定位意思是那个宿主还要被协同定位-->
25 <xs: attribute name="colocate" type="depth" use="optional"
default="notSet" />
<! -- 开放特性指示是否绑定可以看见组合组件的内部结构，或者它就像
一单一组件一样单一的绑定到该组合组件-->
<xs: attribute name="open" type="xs: boolean" use=" optional"
30 default=" false"/>
```

```
<! --排它特征指示是否在下一层上的类型可以容纳其它类型的实例  
浅 (shallow) 意思是仅仅由实例所容纳的实例是这一类型的实例  
-深 (deep) 意思是宿主还必须被标注为排它的-->  
<xs: attribute name=" exclusive" type="depth" use="optional"  
5      default="notS et' />  
</xs: complexType>  
<xs: complexType name=' components' >  
<xs: sequence>  
<xs: element name=" component" minOccurs="0"  
10     max0ccurs="unbounded">  
<xs: complexType>  
<xs: sequence>  
<xs: element name=" settings"  
tvne="settings: settingValues" minOccurs="0" />  
15     <! -- 成员设置值。这些是以在类型上所指定的设置模式为根据的-->  
</xs: sequence>  
<xs: attribute name="name"  
type="xs: string" use=" required"/>  
<xs: attribute name="type"  
20     tune="xs: string" use= "required"/>  
<xs: attribute name=" singleton"  
type="xs: boolean" use=" optional" default=" false"/>  
</xs: complexType>  
</xs: element>  
25     </xs: sequence>  
</xs: complexType>  
<xs: complexType name=" wires" >  
<xs: sequence>  
<xs: element name="wire" minOccurs="0"  
30     max0ccurs="unbounded">
```

```
<xs: complexType>
<xs: sequence>
<xs: element name="members" >
<xs: complexType>
5   <xs: sequence>
<xs: element
      name="member" type="" componentPortRef' minOccurs="0"
      maxOccurs="unbounded" />
</xs: sequence>
</xs: complexType>
10  </xs: element>
<! -- 成员设置值。这些是以在类型上所指定的设置模式为根据的-->
</xs: sequence>
<xs: attribute name="name"
15    type="xs: string' use="required" />
<xs: attribute name="protocol"
      type="xs: string" />
</xs: complexType>
</xs: element>
20  </xs: sequence>
</xs: complexType>
<xs: complexType name=" delegatePorts " >
<xs: sequence>
<xs: element name=" delegatePort" minOccurs="0"
25   maxOccurs="unbounded">
<xs: complexType>
<xs: attribute name="name"
      type="xs: string"/>
<xs: attribute name= " componentName "
30   type="xs: string"/>
```

```
<xs: attribute name="portName"
    type="xs: string" use=" optional" />
</xs: complexType>
</xs: element>
5   </xs: sequence>
</xs: complexType>
<xs: complexType name=" componentPortRef">
<xs: attribute name=' componentName' type="xs: string"/>

10  <xs: attribute name="portName" type="xs: string"
use="required"/>
</xs: complexType>
<xs: complexType name= "delegateHostedClasses" >
<xs: sequence>
15  <xs: element name="hostedClassRef"
maxOccurs="unbounded">
<xs: complexType>
<xs: attribute name=" componentName"
type="xs: string' />
<xs: attribute name= "hostedClass"
type="xs: string" use=" required"/>
</xs: complexType>
</xs: element>
</xs: sequence>
20  </xs: complexType>
<! --  

=====>  

30  <! --单一类型-->
```

```
<! --  
=====>  
5    <! --深度标识是否设置仅仅适用于下一层（浅的），或者必须适用于相邻  
层，并且被设置在相邻层（深）上-->  
    <xs: simpleType name="depth" >  
        <xs: restriction base="xs: string" >  
            <xs: enumeration value="notSet"/>  
10       <xs: enumeration value="shallow' />  
        <xs: enumeration value="deep" />  
        </xs: restriction>  
    </xs: simpleType>  
    </xs: schema>  
  
15    SDM运行时间  
    SDM运行时间（或就称为运行时间）包含了SDM的一个实现。它是一个高  
效的分布式服务，该服务公开出一系列API用来操控SDM类型、成员和实例空间。运  
行时间负责以一致的方式跟踪所有的SDM实例。它提供部署、版本控制、安全和  
恢复的机制。图27表述了SDM运行时间的逻辑架构。  
    SDM运行时间由以下部分组成：  
    ● SDM运行时间-这是SDM运行时间的实现。它是一个分布式的实现，该  
实现将运行于一个或多个物理机器上。运行时间通过SDM API来公开出他的功  
能性，该API是操作SDM和实例的一组调用。  
25    ● SDM存储-这是SDM模拟和实例的永久存储。这个存储是高效的并且它  
的一致性很关键。这个存储要经受得住灾难性的事件。  
    ● 服务部署单元-这是一个用于SDU的只读存储。同SDM存储一样，它也  
是高效的，并经受得住灾难性的事件。  
    ● 组件实现宿主-这是一个用来容纳SDM组件中引用到的CLR代码的框架。
```

SDM运行时间典型的被用于下列客户类:

● 组件实例-这些是使用SDM运行时间库 (RTL) 来与运行时间通信的组件实例。我们区分两种组件实例-运行时间宿入组件实例和非运行时间宿入组件实例。

- 5 ● 开发和部署工具-这些包括SDM编译器, SDU安装工具和其他开发工具。
● 管理工具-这些是用来管理和控制运行时间本身的特权工具。

客户和运行时间通过SDM运行时间库 (RTL) 通信。典型的他们执行的操作包括:

- 10 ● 安装/卸载SDU: 这是向一个SDM运行时间的正在运行的实例中添加或删除新SDU的过程。
● 添加, 消除和修改SDM类型和实例: 客户可以创建新组件, 端口和连接类型。
● 创建和删除实例: 客户可以创建新组件, 端口和连接实例。
● 提供 (source) 和接受事件: 当对类型和/或实例空间的改变发生时, 运行时间将会发送事件给受影响的客户。事件还可以当进行如设置一个端口绑定信息这样的特殊操作时被触发。
● 查询类型和实例空间: 客户可以思考类型和实例空间。

服务定义模型运行时间架构

介绍

20 这篇文档讨论了服务定义模型 (SDM) 和SDM运行时间。关于运行时间架构、核心特性和实现的技术讨论被提供。预定向导 (audience) 是BIG, 打算编写服务和组件的开发者, 或具有在系统细节兴趣的其它开发者的技术求值程序。

服务时代

25 在过去的十年我们亲眼看见了Internet作为一个计算平台出现。越来越多的软件公司采用“软件作为服务”模型。这些服务典型的包括运行在多个机器上的几个组件, 该多个机器包括服务器, 网络配置和其它专用的硬件。松散耦合的, 异步的程序设计模型成为标准。可缩放性, 有效性和可靠性对这些分布式服务的成功是决定性的。

30 我们也亲眼看到了在硬件发展方向上的改变。高密度服务器和专用的网络硬件在数据中心是普遍的。转换结构正在取代系统总线, 在系统配置里提供更

大的灵活性。硬件成本在所有权的全部成本 (Total Cost of Ownership, TCO) 量度 (metric) 中扮演一个很小的角色。这已经被维护一个专门操作的工作人员的成本所取代。稳定性 (rock-solid) 和操作方法 (practice) 对任何服务来说是稀少的但是绝对重要。这些实践，在极大程度上，使由人执行的。

5 开发的焦点是有效的从单一PC到PC网络的转换。然而随着所有这些改变，给软件开发者，硬件供应商，和IT专业人员带来了很多的新问题：

■ 服务是大型的并且复杂的—服务对于开发来说是耗时的，对于维护来说是困难和成本高，并且对于扩充附加功能来说是冒险的。

10 ■ 服务是单一—服务趋向于依靠定制组件和配置。服务的多个部分不能被单独的删除，升级，或者选择性的替换。

■ 服务趋向于特殊的硬件配置—无论它是一个特定的网络拓扑或者依赖于特殊网络应用机器，这极大的降低了在不同的环境中容纳服务的能力。

■ 服务是在silo中被开发的—由于一个通用的平台的缺少，共享代码或即使最好的操作上的方法是一个令人畏缩的任务。

15 ■ 操作盲目 (nightmare) —大多数的服务需要一个操作人员来人工运行。操作人员必须在每个服务的细节上被培训，并且在每个服务发展时再培训。

20 这些问题中的某一些与桌面和DOS时代的那些问题没有不同（大约19世纪80年代）。DOS为应用程序开发者定义了有价值的核心服务，例如磁盘管理，文件系统，控制台工具，等等。然而，它确实给留下了很多复杂的任务由独立软件开发商 (ISV) 来决定。例如，WordPerfect和Lotus123都必须分别编写打印机驱动，为了在它们各自的应用程序中支持打印。同样的，打印机硬件供应商为了一个成功的产品，必须与软件公司达成协议。这些对编写一个DOS应用程序和硬件供应商们的进入屏障格外高。这导致了在这个时代只有少数成功的软件和硬件公司。

25 Windows通过定义一个平台解决这个问题，该平台显著的减少了进入的屏障。Windows为在PC平台上的多数硬件定义一个抽象层。这样消除了开发者必须担心支持特定硬件设备。Windows管理包括存储器，磁盘和网络的个人计算机上的所有的资源。它也带来了应用程序开发者可以利用的服务的财富。这个平台在这个工业中激发了巨大的发展。把Windows平台作为目标的独立软件开发商们 (ISVs) 是多产的。因为具有一个共同的平台Windows的商品化影响，许多新

硬件硬件供应商开始提供更便宜的硬件。

该服务时代还经历这样的增长—在桌面机器上发生的革命需要与服务一起产生。

BIG 服务平台

5 BIG为高效和可缩放服务创建一个平台。该平台能够:

- 使用Visual Studio和如SQL, IIS等等的可复用构件来进行分布式, 可缩放和高有效服务的开发。
- 部署遍及一系列抽象硬件和软件资源, 该资源被自动分配 使用 (purpose) 和配置。
- 通过操作上最佳方法 (practice) 的自动化降低所有权成本。
- 标准数据中心硬件的获得调节商品经济。

该BIG平台是一个Windows平台的扩展, 它建造于现有技术之上, 如.NET, SQL服务器和其它的Microsoft资产。所述BIG服务平台是由多个块组成的, 包括:

- 硬件引用平台, 该平台聚集硬件物品来构造一个简单的大型计算机, 我们称之为BIG计算机。它包括许多互连的服务器, 网络设备和存储器。
- 硬件抽象层, 它虚拟资源。使动态硬件能够绑定和重新部署和自动网络配置。
- 服务定义模型 (SDM) 为开发者描述一个完整的服务。使开发者能够使用高效SQL, IIS和其它可重用构件组件快速建造新服务。
- 高有效运行时间, 它支持所述SDM。能够在所述BIG计算机内容纳多个可缩放服务。
- 操作逻辑框架用于自动操作的最佳方法。允许策略表述和实施。

本文档将单独聚焦在SDM和SDM运行时间上。

服务定义模式

25 本节将讨论服务定义模式 (SDM)。请参考“服务定义语言”文档来得到一个SDM和所述SDML语言的完整的技术描述。

所述SDM是基础, 在SDM上的所有的服务被建造。所述SDM:

- 能够用较小单元组合成服务。这些单元形成硬件和软件的抽象的基础。
- 担当一个服务的活的蓝图—所述SDM以一种比例不变的方式捕获一个服务的全部结构。

- 提供一个框架，用于自动操作方法和促进它们的复用。
- 为部署，复用，发现，版本，和服务的恢复定义标准。

服务的组件模型

实质上，所述SDM是一个服务的组件模型。像传统的组件模型，SDM定义原语，在该原语上更多复杂功能可以被建造。让我们考虑一个类推，Microsoft的组件对象模型（COM）为授权组件定义了一个编程模型。它标准化了组件是如何封装，注册，激活，发现等等。COM要求与生命周期，存储器管理和接口实现相关的严格的规则。这些原语对互操作性是必需的--它允许组件被像黑盒子一样对待。COM对更复杂的服务诸如永久存储器，事件（eventing），自动化和对象链接和嵌入（OLE）的基础。

SDM为服务定义组件模型。这一模型适于松散耦合，分布式和异步服务。所述SDM为部署、版本控制、恢复和范围定义标准。该SDM是一个模型，其中更多的复杂服务被送出。该复杂服务例如是网络管理，硬件管理，存储器抽象，等等。

SDM模型与其它组件模型相比是怎么样的呢？

的确，诸如DCOM和CORBA的技术，其中基于可复用组件具有良好的定义开发应用程序的方法。然而，尽管现有组件技术是有效的，它们在Internet或松散耦合情况中没有广泛的成功。这很大程度上是由于如下：

- 现有组件技术不是被设计用于大规模应用—大多数实现是优先用于单一机器或少量机器。Internet应用程序典型的包括许多在许多机器上相互关连的组件。
- 现有组件技术委托像RPC的启动协议—它们既不调节建立好的网络协议，也不允许偏离协议。
- 现有组件技术缺少应用程序的概念—多数具有组件的开发好的定义，但是缺少一个应用程序的全面的定义，该应用程序由较小组件组成。
- 现有组件技术受限于运行在一个通常目的的计算机上的软件—单一目的网络设备可以不作为组件参加。

这说明有许多想法进入现有组件技术，这些想法还是与服务世界较大的相关。

SDM原理

SDM是一个服务结构的说明性定义。这些定义是用在组件，端口和连接线方面：

- 组件是执行，部署和操作的单元。组件是专用服务器，运行.NET服务器，在共享机器上的虚拟网络站点或例如Cisco LocalDirector的网络应用程序。组件通过端口公开功能性，和通过连接线建立通信路径。可以被嵌套在外部组件中的组件被称为复合组件。
- 被命名为终点的端口具有关联类型。端口类型经常表示一个协议，例如，HTTP服务器。端口为建立通信捕获需要的信息。
- 连接线是在端口间的允许的通信路径。它们说明在端口间的拓扑关系。

10 服务被使用一个服务定义模型语言（Service Definition Model Language,SDML）语言来创造（authored）。让我们考虑一个例子：

```
using System;
using System.Iis;
using System.Sql;
15 [sdmassembly: version(1)];
componenttype MyFrontEnd: AspApplication
{
    port SqlClient catalog;
    implementation "MyFE, MyClrAssembly";
20 }
componenttype MyBackEnd: SqlDatabase
{
    implementation "MyBE, MyClrAssembly";
}
25 componenttype MyService
{
    component MyFrontEnd fe;
    component MyBackEnd be;
    port http=fe.http;
    wire SqlTds tds
30 }
```

```
  {
    fe.catalog;
    be.sqlServer;
  }
5   implementation "MyService, MyClrAssembly";
}
```

正如所见的，SDML的语法从C#借用了许多。DSML定义了组件，端口和连接类型。如果我们过一遍这些定义：

● 使用指示引用了SDM类型的名字空间。这些包括系统名字空间，该名字空间由SDM运行时间提供并定义了例如http连接线类型等基本类型。其它名字空间定义了与IIS和SQL Server相关联的名字空间。

● 组装（assembly）名字和组装版本为SDM组装提供了一个健壮的名字。注意，这与CLR组装无关。一个SDM组装是一个SDM部署的最小单元。它命名并包含了组件，端口和连接线类型的集合。SDM组装不应该与CLR组装相混淆，它们截然不同。

● 一个被称为MyFrontEnd的组件类型被说明，它继承于组件类型AspApplication，该组件类型是在System.Iis SDM组装中定义的被引用的类型。组件是一个抽象概念；它们指的是一个类而不是实例。MyFrontEnd从标识一组件0个或多个组件实例从该组件中被创建。

● 端口SqlClient分类（catalog）：说明了一个在SqlClient类型的MyFrontEnd组件上的端口。该端口被称之为“分类”。该端口还包括端口，组件和由基本组件类型AspApplication继承来的MyFrontEnd连接线（wire）。

● 实现关键字引用一个组件类型的实现。这一实现是对CLR组装中的CLR类的引用。它可以被认为是一个组件类型的入口点或构造函数。当一个组件实例被创建时，该代码被调用。

● MyService组件类型由叫做fe和de的两个子组件来定义，他们具有MyFrontEnd和MyBackEnd类型。组件MyService的实例可以相继的使fe和be实例形成组件实例层次。

● http端口=fe.http：说明了一个在MyService组件类型上的端口，该组件类型在fe组件上被授权到http端口。

● 连接线 (wire) SqlTds tds说明了一个在SqlTds类型的MyService组件类型中的连接线 (wire)，通过名字tds。两个端口被连结到连接线上。这一说明意味着一个MyService实例可以有0个或多个连接线tds的实例并且那些连接线实例的每一个可以拥有来自fe组件的分类端口，和来自他们上面所连结的组件的sql端口。

一个服务的图形表述经常会有助于理解。参考图28。框表示组件，菱形表示端口，线表示连接线 (wire)。

组件实现

在CLR组装中每一个组件都可以引用一个CLR类形式的实现。CLR组装容纳于SDM运行时间并在组件示例时被调用。实现SDM组件的CLR类可以通过调用SDM运行时间API执行SDM操作。这将在本文档的后面更加详细的描述。下面是一上面所说的MyService SDM组件类型实现的C#代码片断。

```

using System;
using Microsoft.SDM;
15 public class MyService: SDMComponentInstance
{
    public override OnCreate(...)
    {
        SDMComponent fe1 = CreateComponentInstance("fe","");
        SDMComponent fe2 = CreateComponentInstance("fe","");
        SDMComponent be1 = CreateComponentInstance("be","");
        SDMWire tds1 = CreateWire instanceance("tds");
        tds1.Members.Add(fe1.Ports["catalog"]);
        tds1.Members.Add(fe2.Ports["catalog"]);
        tds1.Members.Add(be1.Ports ["sqlServer"]);
        }
    }

```

这一代码定义了C#类MyService，该类从SDMComponent继承。该类重载了OnCreate()方法并创建了两个fe组件实例，一个组件实例和一个连接线 (wire) 实例。它随后向连接线实例添加了3个端口。

这段CLR代码被编译到称为MyClrAssembly的组装中，该组装在MyService的SDM中被引用。当一个类型为MyService的组件被初始化时，这段代码将被调用并且OnCreate()方法也将被调用。

[BassamT] Consider showing the strongly-typed version of the C# code.

实例

5 SDML被用来定义组件，端口和连接线类型；它不定义实例。像我们在上面的C#代码里看到的一样，实例可以使用SDM运行时间API来创建。上述C#代码创建许多的实例，并且在实例空间里形成布线拓扑。这些实例将由SDM运行时间跟踪。例如所述SDM运行时间将在上面OnCreate调用完成后，存储以下的信息：

```
10 component instance ms [1]
    port instance http[1]
    component instance fe[1]
    component instance fe[2]
    component instance be[1]
15    wire instance tds[1]
        fe[1].catalog
        fe[2].catalog
        be[1].SqlServer;
```

注意：这里用到的句法不是SDML；它被用来说明由所述SDM运行时间跟踪的实例空间。

ms[1]是一个有三个子组件实例fe[1]，fe[2]，be[1]的组件实例。fe[1]和fe[2]是该fe组件的实例。be[1]是所述be组件的一个实例。tds[1]是一个包括三个成员的连接线实例。用图形来表示，所述实例空间示出在图29。

25 组件实例有实际物理表示—fe[1]和fe[2]，在这一例子中，是两个ASP.NET应用程序，它们运行在一个Windows机器上的IIS上。当调用CreatComponentInstance时，一个新的ASP.NET应用程序被创建和被配置在一个IIS框上（box）。多个中间步骤也可以被调用—例如，因为使用新的资源所述调用者的信用卡已经被收费，或新的机器由于能力的缺少被分配。在本文档后面

我们将检查在组件实例后面的机器。

服务部署单元

SDM模型根据组件，端口和连接线为MyService定义了服务的结构。这导致一个SDM 组装可以被安装在一个SDM运行时间机器上。相反的，所述SDM 组装不足以来安装该服务。除SDM组装之外，我们也必须考虑CLR组装，该组装是组件的实现。我们也必须考虑ASP.NET代码，SQL脚本和任何其所述服务需要的内容。这些片断的总和被打包上传到服务部署单元（或SDU）。见图30。

SDM运行时间

SDM运行时间（或就称为运行时间）容纳了SDM的一个实现。它是一个高效的分布式服务，该服务公开出一系列API用来操控SDM类型、成员和实例空间。运行时间负责以一致的方式跟踪所有的SDM实例。它提供部署、版本控制、安全和恢复的机制。

本段描述了所述SDM运行时间的设计和实现，像为BIG V1.0版本提议的一样。当的确有SDM运行时间的不同实施例，我们在整个本文档中将集中到一个—高效率SDM运行时间实现，它将被容纳在BIG计算机上。（参见—得到更多细节）。

运行时间架构

图27表示了所述SDM运行时间的逻辑架构。

所述SDM运行时间包括如下：

- SDM运行时间—这是所述SDM运行时间实现。它是一个分布式实现，将运行在一台或多台物理机器上。所述运行时间通过SDM API公开它的功能，该API是操作SDM和实例的调用组。
- SDM存储器—这是SDM模型和实例的永久存储。这个存储是高效的并且它的相容性很关键。这个存储要经受得住灾难性的事件。
- 服务部署单元-对于SDU这是一个只读存储。同SDM存储一样，它也是高效的并且经受得住灾难性的事件。
- 组件实现宿主—它是用于容纳所述CLR代码的框架，该代码从SDM组件中被引用。

所述SDM运行时间典型的被下面的客户类使用：

- 组件实例—这是使用SDM运行时间库（RTL）与运行时间通信的组件实

例。我们在组件实例的两个类型之间区别—运行时间宿入组件实例和非运行时间宿入组件实例。

● 开发和部署工具—与其它开发工具一样，这些包括所述SDM编译器，SDU安装工具。

5 ● 管理工具—这是用来管理和控制运行时间本身的特权工具。

客户和运行时间通过SDM运行时间库（RTL）通信。他们典型的执行的操作包括：

● 安装/卸载SDU：这是向一个SDM运行时间的正在运行的实例添加或删除新SDU的过程。

10 ● 添加，除去和修改SDM类型和实例：客户可以创建新组件，端口和连接线类型。

● 创建和删除实例：客户可以创建新组件，端口和连接线实例。

15 ● 提供和接受事件：当对类型和/或实例空间的改变发生时，运行时间将会发送事件给受影响的客户。事件可以当进行如设置一个端口绑定信息这样的特殊事件时被触发。

● 查询类型和实例空间：客户可以思考类型和实例空间。

类型，成员和实例空间

在组件类型，组件和组件实例之间的关系是与类，类成员和在普通面向对象语言中的对象类似的。SDM定义在类型，成员和实例空间之间的分隔。组件类型在类型空间里，组件在所述成员空间内，并且组件实例在实例空间内。图20 31说明了在所述三个空间的分隔。

所述“成员空间”包含类型空间的实例。所述“实例空间”包括成员空间的实例。所述SDM运行时间负责跟踪所有的三种实例空间和在它们之间的关系。这种信息存储在所述SDM存储器，并且可以通过使用运行时间API被查询。组件25 和连接线可以具有0个或多个实例。端口可以只有一个实例。

所述SDM成员和实例空间确定一个严格的体系。所有在成员和实例空间内的组件被安排在一棵树内。根组件是一个特殊的组件被称为“根”或“通用”组件。让我们看在先前的段落（图32）中的MyService例子中的成员树。框表示组件，并且线表示父子关系。MyService是根组件的一成员组件。所述实例树如图3330 所示。注意有两个myService实例，该实例有不同数量的孩子实例。

myService[1].fe[1]和myService[2].fe[1]具有同样的组件成员“fe”，并具有同样的组件类型“MyFrontEnd”，但是其它是完全截然不同的组件实例。“root[1]”是所述根组件的唯一实例。

组件示例

5 一个由SDM运行时间提供的基本操作是组件示例。这是其中有一个组件实例形成的过程。不像传统的组件模型，创建一个实例（或一个对象）典型的为所述实例包括分配和初始化一个存储程序块，SDM典型的包括由不同部分执行的许多步，并且使用几小时如果不是几天来完成。例如，当一个类型ASP.NET
10 类型应用程序组件被示例，结果是一个新的在机器上的虚拟网络站点运行IIS，跟着是一个配置动作。考虑一个情况，其中在所述IIS机器上的容积被达到，并且新的一个必须在ASP.NET应用程序被示例化以前被分配。这个过程可能花费几个小时，因为它将包括从一个池中分配一个新的机器，可能导致一个记账收费，并且安装包括IIS的操作系统。所述SDM运行时间支持两种方式来示例组件
15 1) 工厂示例组件和2) 运行时间示例组件。这些方法在下面被简短地讨论。请参考所述“组件示例”描述得到更多细节。

工厂示例组件

组件工厂（或只是工厂）是负责为一个或多个组件类型的创建实例。工厂是它们自己组件，其为示例目的公开一个或多个端口。一种方式来考虑工厂是作为资源管理器。它们管理的资源是组件类型。工厂了解如何映射一个资源到一个组件的实例。例如，假定我们有一个类型“文件存储”的组件。当这个组件被示例时，一个NTFS目录将被创建，并且适当的ACL将被供应。该组件的工厂可能管理多个Windows机器，为了分配存储器的目的。该工厂负责创建NTFS共享，设置ACL，限额等等。组件工厂在SDM运行时间里扮演一个重要的角色。因为它们典型的表示服务管理资源，所以它们被期待成为可靠的和高效的。然而被SDM运行时间支持的组件工厂的数量是未限制的，我们期望BIG V1.0将拥有很多数目的基本组件工厂。它们是：

- 硬件—它是基本等级工厂，负责分配硬件的实例并且管理它们。例如，它可以收集一个有1GB内存，或如NAS的一个存储设备的服务器机器。
- 网络—这一工厂负责VLAN，公共IP地址，DNS名称，等等。
- PC—这一工厂可以分配一个机器并且部署一个完全图形OS在上面。

- 存储器—这一工厂负责管理和分配存储器。
- 软件资源—如ASP.NET, IIS网络站点, SQL服务器数据库等等。

示例过程

工厂必须向特定的SDM运行时间注册，该运行时间指定负责创建实例的组
5 件类型。在高级别，该示例过程如下：

- 调用者要求用于组件工厂的SDM运行时间来得到一个给出的组件类型。
1. 所述SDM运行时间负责找到适当的组件工厂并把它返回给调用者。
 2. 然后所述调用者与组件工厂直接通信，并询问它以创建一个或多个实例。

10 运行工厂列表

所述SDM运行时间将包括一个组件类型的列表和它们的适当的工厂。每个组件示例具有一个运行工厂表。该运行工厂表的结构是如下所示：

(ComponentTypeID, PortType)→(PortInstance, [cookie])

组件实例可以添加/除去在它们的表中的实体以及任何它们的直接孩子的
15 列表。缺省的，当一个新孩子组件示例被创建时，父母的运行工厂列表是继承的。

运行工厂列表为每个组件实例跟踪，为了给在不同的上下文的相同组件支持不同的工厂。既然工厂典型的是资源被分配的地方，容纳环境可能为资源分配委托不同政策。例如，考虑一个情况，其中容纳入口如Digex有不同的计划给
20 它们的顾客。为Gold付款的顾客将得到一个专用的IIS框，而且为Sliver付款的顾客将得到一个共享的IIS框。顾客的服务包括一个“ASP.NET应用程序”类型的组件，并且它不知道它是被容纳在一个专用的IIS机器还是一个共享的机器里。
Digex可能如图34中所示实现这一服务。

Digex是一个具有两个组件工厂Gold Factory和Sliver Factory的组件。工厂自己就是组件。Digex也定义其它被称为“Gold”和“Sliver”的组件。这些“Gold”组件将成为所有为Gold Service支付的服务的父母。
25

当Digex被示例时，它将创建一个工厂的实例，并且也创建“Gold”和“Silver”组件的实例。Gold[1]将有自己的运行工厂表。Digex将通过调用适当的SDM运行时间API注册在这个表中的Gold Factory。当一个新的客户的服务被示例为一个
30 Gold[1]的孩子时，它将继承Gold[1]的运行工厂表。这意味着当一个“ASP.NET 应

用程序”的组件实例被创建时，Gold Factory将处理这个请求，适当地向该顾客的账户收费。

工厂跟踪

所述SDM运行时间将明了创建每个组件实例的工厂。见图35。虚线表示一个在一个组件实例和创建它的工厂之间的“被创建”关系。像上面提到的，该工厂自己就是组件，因此它们必须有工厂。为终止无限的递归，如下面所描述的一样，运行时间将成为一个“运行时间—容纳组件”。也注意根组件实例是特别的，并且它是它自己的工厂。

工厂和事务处理

工厂将支持事务处理来减轻服务开发者必须考虑复合重新运行和错误处理的逻辑。不建立在处理子系统的顶端的工厂需要支持补偿。工厂也必须支持在一个分布式事务处理中的征募(enlisting)。

工厂将典型的保留多个与示例相关的帐簿信息。为了保证正确恢复，该帐簿信息必须与SDM运行时间保持一致。为了促进这些，所述SDM运行时间将为包括工厂的组件实例提供一个处理存储装置服务。一个好写入工厂将在这个存储器中存储所有帐簿信息。

工厂端口

工厂将典型的公开一个或更多可以被用于组件管理的端口。尽管所述端口类型不由我们推荐的SDM运行时间委托管理，所有组件工厂支持SDM_Factory端口。SDM_Factory是一个SOAP基础端口，这被调用来示例新组件实例。用于这一端口的C#接口如下：

```
public interface ISDMFactory
{
    ComponentInstance Instantiate(
        ComponentInstance parent,
        Component component,
        ComponentType componentType,
        object args);
    void Alloc(ComponentInstance allocInstance);
    void Construct(ComponentInstance constructInstance);
}
```

```
}
```

ISDMFactory支持三周期 (pass) 示例处理:

示例周期 (pass): 通过SDM运行时间这种周期将创建所有组件实例递归。它将不做任何分配或建造。它仅仅只创建“skeleton”所需要的组件实例。

5 分配周期: 在这个周期期间, 所有的相关组件工厂将分配该示例需要的任何资源。

构造周期: 如果分配成功, 那么构造过程将开始。这是典型的最长运行周期。在构造周期中, 该工厂将典型的做所有的实际工作。

10 工厂的确可以为示例支持其它端口类型, 但是SDM运行时间和运行时间 API有多个帮助功能, 该功能与SDM_Factory实现一起工作。这些API将必然为主要的开发者提高开发经验。

运行时间容纳 组件实例

15 除了工厂, SDM运行时间也将为SDM组件容纳实现, 该组件使用实现 SDML关键字引用一个CLR组装。被引用的CLR组装是一个字面上的串, 是一个 CLR类的全部限制名称。例如:

```
componenttype A
{
    port pt x;
    implementation
    "MyNamespace.MyClassName,MyClrAssemblyName"
}
```

或者你可以为强有力的被命名CRL组合指定特征 (culture), 版本和关键字:

```
componenttype A
{
    port pt x;
    implementation "MyNamespace.MyClassName,
    MyClrAssemblyName, culture=neutral, version=1.0.0.1234,
    PublicKeyToken=9a33f27632997fcc"
}
```

30 SDM运行时间将为这样的组件担当工厂, 它将容纳和管理这些CLR类。这

些也结束上面提到的工厂的无限的递归，因为基础级工厂被实现作为容纳于SDM运行时间的CLR组装。

CLR组装将用Microsoft的IIS Server容纳。实现关键字引用一个类，该类必须从 MarshalByRefObject 继承并且实现 IRuntimeHostedImplementation 和 ISDMFactory 接口。为了方便，基本类 SdmComponentInstance 提供这些接口的一个缺省实现。以下是上面的组件类型A的运行时间宿入CLR的一个实现例子。

```
public class A: SdmComponentInstance
{
    protected override void OnCreate(object args)
10    {
        // do something
    }
}
```

15 类A是一个从SdmComponentInstance继承的C#类，因此可以被SDM运行时间容纳。为了该类正常工作，这个类的CLR组装也必须放置于SDU的\bin子目录中。当一个类型A的组件的实例被创建的时候，运行时间负责查找一个有效的IIS宿主机器，并示例化（instantiating）那个机器上的CLR代码。CLR代码作为一个被IIS容纳的.NET远程应用程序容纳。一个SDU中的所有CLR组装共享同一IIS 20 过程，并在此过程中具有它们自己的AppDomain。

一旦CLR组装载入，运行时间将会执行一个.NET远程调用来明确定义在IRuntimeHostedImplementation接口上的入口点。从这点看来，CLR类等同于一个组件工厂并且ISDMFactory接口像我们在前一段看到的那样被消耗。

端口和连接线

25 端口和连接线是与SDM运行时间通信的基础。端口和连接线解决了许多现今在服务部署上的公共的问题：

通信信息的硬编码-许多服务典型的会在他们的代码里硬编码他们服务器的名字或ip地址。例如，前端服务器典型的会硬编码SQL服务器的名字和连接信息例如数据库名字、登录和密码。

30 定义通信拓扑-大多数服务部署典型的使用DMZ作为定义通信边界的唯一

机制。其他的约束并不强制，例如，如果前端服务器有时需要与其他前端服务通信，这将不会在任何地方被捕捉。

恢复-现今发现从服务中添加或删除新的组件是服务所面对的一个典型的问题。

5 SDM通过端口和连接线解决这些问题。端口是在组件上所公开出来的有类型的实体。端口类似于服务访问点—它是组件公开明确定义好的功能性的的地方。例如，一个“存储器”组件将定义一个SMB.server类型的端口，该端口可以用来进行文件系统操作。连接线定义端口之间的可允许的绑定。它们形成一个可以约束通信路径的通信拓扑。

10 让我们回顾前面的MyService例子：

```
componenttype MyService
{
    component MyFrontEnd fe;
    component MyBackEnd be;
    port http=fe.http;
    wire SqlTds tds
    {
        fe . catalog;
        be. sqlServer;
    }
    implementation "MyService, MyClrAssembly";
}
```

MyService包含称作tds的单一的连接线。连接线，就像组件一样，可以拥有实例。例如，下面是具有两个不同连接线实例拓扑的两个MyService组件实例
25 ms[1]和ms[2]。

```
组件实例ms[1]
连接线实例tds[1]
fe[1].catalog
fe[2].catalog
be[1].SqlServer;
```

组件实例ms[2]

连接线实例tds[1]

fe[1].catalog

be[1].SqlServer

5 连接线实例tds[2]

fe[2].catalog

b[1].SqlServer;

ms[1]拥有包含三个端口实例的单一连接线实例tds[1]。ms[2]拥有两个连接线实例tds[1]和tds[2]，每个连接线都拥有两个端口实例。第一种情况中，fe[1]和fe[2]可以彼此看见。第二种情况中fe[1]和fe[2]彼此看不见。

10 连接线实例形成物理通信拓扑。端口实例是连接线实例的成员。他们可以：

1) 彼此查询或恢复--运行时间API支持查询和恢复同一连接线实例中其他端口实例的功能。在同一连接线实例中所有成员都是可见的。并且，连接线实例的所有者可以在任何时候查询该成员。

15 2) 接收事件--连接线的成员将接收SDM操作成员端口实例所触发的事件。参见下面“事件”得到更多细节。

3) 约束通信--连接线实例约束被允许的组件实例间的通信路径。

端口绑定信息

端口是组件所公开出来的有类型实体。端口可以确切的拥有一个实例。一个端口实例可以携带绑定信息，该信息典型的是在组件之间建立通信信道所需要的任何东西。例如，上面的端口实例“be[1].SqlServer”可以拥有以下绑定信息用来连接SQL后端。

“server=mySQLServer;uid=myLogin;pwd=myPwd;”

25 这个字符串可以被传送到ADO或OLEDB，TDS连接可以被建立到后端SQLServer。SDM运行时间不妨碍通信的双方。它仅仅担当所需要的信息的持有者来启动通信。

端口可见性和连接线实例

组件实例上的端口实例只有当它们被连接到相同的连接线实例时才对其他组件实例可见。这是一个用来为服务建造逻辑网络拓扑很有用的机制。SDM运行时间还支持用于自动创建物理虚拟网络和为了实现连接线实例约束在需要时

用于自动使用包过滤的装置。更多信息请参看“网络架构”文档。

事件

SDM运行时间产生某些固有事件作为对SDM实例空间操作的结果。例如，当一个组件实例创建一个端口实例时会产生一个事件。依赖于特殊事件，目标5既可以是复合组件实例也可以是在给出的连接线上的端口实例。

所有事件被交给运行时间端口上的组件实例。SDM运行时间库负责跟踪这些事件并将它们翻译成特殊语言调用。例如，基于CLR的SDM运行时间库将产生一个CLR事件。

组件实例事件

10 当新组件实例被创建或一个被存在的组件实例被删除时这些事件被产生。该事件的目标总是父复合组件实例。该事件仅仅被发送到直接父复合事件—它们不延实例树向上传播。从上面我们的例子中，假定组件实例“u[1].foo[2]”请求运行时间来创建一个成员组件“c”的新实例。参看图36。

15 组件实例“u[1].foo[2]”的代码当前运行在机器1上。它通过使用SDM RTL要求运行时间创建一个组件“c”的新实例。运行时间知道调用组件实例的身份并可以消除歧义并执行操作。新的组建实例被创建，事件被产生并提交回调用组件实例。当一个实例被消耗或失败时，运行时间会发送适当的事件给父组件实例和适当的组件工厂。

端口实例事件

20 当一个组件实例创建一个端口实例或删除一个被存在的端口实例时，父组件实例被通知这种改变。看图37。如果一个端口实例被连接到一个连接线实例，将像父组件一样通知所有连接线实例的成员这种变化。这在下一节会详细描述。

端口状态

每一个端口实例可以是如下状态之一：

- 25 ● 被创建—这是当第一次被创建时端口的状态。这触发一个被发送到父组件实例的事件。
- 被连接—端口被连接到连接线实例时进入这种状态。这触发一个被发送到父组件实例和连接线实例中所有成员的事件。
- 在线—端口当已经准备好操作时进入这种状态。这触发一个被发送到父30组件实例和连接线实例中所有成员的事件。

● 离线—当端口想停止正常操作时，端口进入这种状态。这触发一个被发送到父组件实例和连接线实例中所有成员的事件。

● 被分离—当端口被从连接线实例中分离时，端口进入这种状态。这触发一个事件，该事件被发送到父组件实例和该连接线实例的所有成员。

5 ● 被删除—当端口被从实例空间中移除时，端口是在这种状态。这触发一个事件，该事件被发送到父组件实例。

连接线实例事件

当一个连接线实例被创建或删除时，连接线实例事件产生。这些事件的目的经常是父组件实例，该实例拥有连接线。见图38。

10 连接线实例也可以包含到它的成员的端口的引用。这种连接线成员确定某些成员端口事件的目的地。让我们从上面继续我们的例子。假定“foo[2].c[2]”已经创建多个如下的新实例：

component instance universal[1]

component instance foo[2]

15 component instance c[2]

port instance y[1]

component instance b1[1]

port instance x[1]

component instance b2[1]

20 port instance x[1]

wire instance p[1]

b1[1].x[1]

b2[1].x[1]

注意连接线实例“p[1]”包含到两个端口实例的引用“b1[1].x[1]”和“b2[1].x[1]”。

25 让我们假定组件实例“b1[1]”和“b2[2]”每个运行在分隔的机器上。图39示出了当“b2[1]”改变它的端口状态到离线时，事件产生。

注意，“b2[1]”被容纳在机器3上并且它调用在运行时间上的“设置端口状态”操作。运行时间记录该变化并发送三个事件—一个发送给连接线实例所有者“u[1].foo[2].c[2]”，两个发送给连接线端口实例成员“b1[1].x[1]”和“b2[1].x[1]”。

30 事件发送和队列

运行时间将保证事件的按次序发送但是它不保证给定连接线实例的所有成员之间的完全虚拟同步。换句话说，SDM运行时间将允许继续向前的执行即使一个组件实例运行得很慢甚至死掉了。

对每个组件实例来说，SDM事件被排队。如果事件成功的被排队在目标队列中，触发事件的操作被认为是成功的。实际上队列是环形的，如果一个组件严重的延迟或已经死了该队列可以绕回。绕回将产生一个新“绕回”事件。这个事件被发送到组件实例自己和父组件及任何拥有的工厂。

运行时间分区

为了支持大量的客户，运行时间可以被分区。由于SDM实例空间的严格层次，这个问题被公平的处理。SDM运行时间通过特殊的部署可以被容纳在许多机器上。每个SDM运行时间实例负责跟踪实例空间的一部分。组件实例使用SDM运行时间库与适当的运行时间通信。图40示出了一已分区的运行时间和一些客户机。

机器1包括两个组件实例和一个SDM运行时间库。机器2包括一个单独的组件实例和一运行时间库。机器3容纳一专用SDM运行时间。机器4具有一个SDM运行时间和一个组件实例。还要注意，在机器3和4上的两个SDM运行时间是连通的。

分区

运行时间调节（LEVERAGE）固有在SDM中的本来的层次，来对其自身进行分区。分区的行为包括分布跨越不同的在运行的运行时间实例的SDM类型和实例空间部分。分区对于可缩放性来说是必须的。对于类型和实例，发生不同的分区。

- **类型和成员空间：**已给的运行时间可以包括多个类型定义，典型的该定义被组织在带名字空间中。每一运行时间将仅仅需要知道类型和成员，其是由该运行时间正在跟踪的实例来定义的。这些可以出现在复合的运行时间上。换句话说，允许在该类型和成员空间里的重叠。

- **实例空间：**一个给出的运行时间将仅仅追踪该实例空间的一部分。在复合组件的实例界限上该实例空间被分区。在实例空间里的重叠是不被允许的。

这最好通过一个例子来解释；考虑接下来的组件类型定义：

30 componenttype B {

```

port X x;
}

componenttype C {
    port Y y;
    5      component B b 1;
    component B b2;
    wire P p {b.Lx; b2.x;}
    componenttype A {
        port internal Z z;
        10     component C c;
        wire W w {z; c.y}
    }
    componenttype universal u {
        component A foo;
        15     component A bar;
    }
}

```

这一定义包括三种组件类型A，B和C。A是根全体组件的成员。B和C是A的成员。这将便于我们如图41中所示的那样，以图画的形式表示成员空间。我们将使用框来表示复合组件。注意，该复合（COMPOUND）组件成员被描述在组件框中，其中该成员不是其他复合组件。在这个例子中，连接线“W”是复合组件“foo”和“bar”的一个成员，而且因此被表示在一个“a”框中。

在该实例空间中，可以存在每一个组件，端口和连接线的多个实例。我们如图42中所示表示该实例的层次。在这里该框表示用于组件实例跟踪的实例状态-不是该组件实例的实现代码。

假定我们想要在三种运行时间-运行时间1，运行时间2和运行时间3中分区这个SDM模型。图43是一个分区该实例空间的例子。在这一例子中，运行时间1跟踪“universal[1]”，“foo[1]”，“foo[2]”和“bar[1]”。运行时间2跟踪“foo[1].c[1]”，“foo[1].c[2]”和“foo[2].c[1]”。运行时间3跟踪“bar[1].c[1]”。另外，该运行时间必须知道所他跟踪的实例的类型。在这个例子中，由于其父亲“bar”，运行时间3必须已知组件类型“C”，“B”和“A”。它还必须已知端口类型“Y”和

连接线“W”。

不同的运行时间也必须包括它们之间的关系。这一关系是由该SDM层次要求的。在前的例子中，为了管理该“foo[1].c[1]”，“foo[1].c[2]”和“foo[2].c[1]”的关系，运行时间1和运行时间2必须知道每一个。类似的，运行时间1和运行时间5 3必须围绕“bar[1].c[1]”协同工作。注意该运行时间2和运行时间3不知道彼此。

分区策略

该运行时间将包括足够的逻辑性以便对其本身进行自我分区。特殊的分区策略将以性能，容量和SDM定义的约束条件为基础。这一分区是动态的，并且将随着该SDM模型的增长而改变。

10 单根运行时间 (Single-root Runtime)

跟踪复合组件实例的运行时间被称为单根运行时间，该复合组件实例是一单根复合组件实例的所有实例。在上述例子中，运行时间1和运行时间3是单根运行时间。运行时间1具有一开始自“universal[1]”的根实例树，而运行时间3具有一开始自“bar[1].c[1]”的根实例树。

15 多根运行时间 (Multi-root Runtime)

跟踪组成实例的运行时间被称为多根运行时间，该组成实例是不具有一根复合组件实例。在上述例子中，运行时间2是多根运行时间，因为其跟踪所有的根“foo[1].c[1]”，“foo[1].c[2]”和“foo[2].c[1]”。

服务安装

20 在一个服务可以被在一给出的SDM运行时间上实例化前，该服务必须首先被安装。该安装过程包括以下步骤：

复制该服务部署单元到一运行时间部署共享

调用该SDM运行时间API来便开始该安装

服务部署单元

25 该SDU是一服务部署的单元。其包括：

SDM组装 (assembly) -这是新服务的类型信息。包括用于那个服务的所有组件类型，连接线类型和端口类型。该组装是编译该SDML服务的结果。

所容纳运行时间组件实例代码在SDML中由该运行时间容纳的，并由该实行关键字所参考的任意CLR代码必须被包括在该SDU中。

30 其它服务二进制文件-所有其他二进制文件比如配置文件，DLL， GIF，

HTML, SQL脚本等等, 也可以被称为部分的部署单元。

该SDU是暂时的-对该SDU的改变不被允许。一旦SDU被安装, 其就不能被改变。当然, 一个SDU可以安装该SDU的新版本, 该新版本升级并潜在的作废该老版本。

5 SDU格式

该SDU是由SDM运行时间消耗的二进制文件目录和潜在的组件工厂。该目录是相当大的自由形式, 但是期望下面的结构:

```
\sduroot  
  \<assembly name>. <version>  
10  \<assembly name>. sdmassembly  
  \bin  
    \<Runtime hosted CLR assembly_1>. dll  
    \<Runtime hosted CLR assembly_2>. dll  
    \<Runtime hosted CLR assembly_n>. dll  
15  \<other files and directories>
```

该SDU将被打包成一个CAB文件。

实现

该SDM运行时间被作为运行在IIS服务器上的.NETWebService实现。该SDM存储器是可靠的SQL服务器数据库。该SDM运行时间WebService是一无状态的WebService。换句话说, 在该SDM运行时间服务里任何状态都是暂时的。所有持久的状态将被写入在清楚的事务处理界限上的存储器。

该SDM运行时间服务可以被停止, 并可以在任意点重新应用, 即使是在不同的机器上。如果其指向相同的SDM存储器, 所有的工作就将伴随很少或没有中断重新开始。

25 SDM存储器

该SDM运行时间使用一持久的存储器来存储SDM和实例。典型的, 这一存储器被典型的设置在与SDM运行时间服务相同的机器上, 但是当然它可以被不同的部署。该SDM存储器是一SQL服务器数据库, 该数据库包括关于所有SDM模型和他们的实例的信息。

30 需要SDM存储器的这种可靠性和有效性是绝对必要的。SDM的关键设计目

的之一是在最后知道的一致的状态下重新启动该系统的能力。因此该SDM需要非常可靠，而且必须经受得住灾难性的故障情况。这被以两种方式实现：

该SDM存储器将被复制，而冗余的热备份将一直有效。使用Yukon's Redundant Database Technology来实现这一性能。

5 该SDM存储器将被定期备份，而且该信息将被存储在空闲地点。该备份将是存储在SDM存储器中的当前模型，实例和任意服务状态的自身一致的瞬相（snapshot）。

服务存储装置

该SDM运行时间将提供用于在组件实例层上存储的设备。每一个组件实例可以使用运行时间API来在该SDM存储器中存储数据。在最小方面，虽然我们考虑不完全结构的存储装置，但是这一存储器是一BLOB存储器。

10 在运行时间里所存储的服务状态被保证象SDM运行时间一样可靠而持久。还保证与其它运行时间状态的一致性。当然，我们不支持所有要被存储在SDM存储器中的服务状态，相反的期望服务存储它们状态的足够的信息（一组指针）。
15 依靠恢复，服务可以找到指向其数据的指针，并且执行必需的步骤。参见下面的恢复。

SDM运行时间安全

情况描述

存在两个基本情况，其将定义用于SDM运行时间的安全模型：开发者测试
20 运行情况和操作者产品部署情况。两者的通用需求如下：

- 功能，从SDM运行时间被执行的计算机连接到目标服务器的能力。
- 使用有效目录域账号的Windows验证。
- 信任子系统模型，用于访问目标服务器资源，以便执行安装，更新和卸载操作。
- SDM运行时间作为一Windows服务来实现，并且作为一信任服务账号来运行。
- 一数据库（MSDE），被配置，来使用Windows验证和数据库角色（role），该数据库角色跟踪SDM类，类型和实例信息。

开发者测试运行情况

30 在测试环境下，一开发者必须能够部署一分布的应用程序到一个或多个服

务器。该目标服务器是独立工作组的部分或者处于相同的有效目录域中的部分。计算机必须位于与作为目标服务器相同的工作组或域中，其中从该计算机开始该测试运行部署被初始化。

1. 使用Visual Studio，开发者生成一服务部署单元（SDU）包
- 5 2. 所产生的SDU被放置在SDM运行时间服务正在执行的
计算机上的一部部署文件夹中。
3. 开发者选择一部部署动作（安装，更新，卸载），并提示
Windows验证资格输入。
4. 开发者被验证，而且被映射到一部部署角色，该部署角色
10 确定所验证的用户是否被授权来执行所请求的部署操作。
5. 开发者选择在哪一目标服务器上哪一组件要安装，更新
或删除。
6. SDM运行服务以两种方式中的一种连接所选的目标服
务器：如果该SDM运行时间服务正作为有效目录中的一信任服务账户在运行，
15 那么它就将作为在目标服务器上的账号来连接。否则，该SDM运行服务就将作
为被验证的用户来连接，如果扮演是不可能，在该目标服务器上就可能需要一
附加验证。

操作者生产部署情况

在数据中心环境中，一操作者必须能够对一个或多个服务器部署一分布式
20 应用程序。该目标服务器必须是有效目录域或者树状集合的部分。该计算机必
须位于作为该目标服务器的相同的域或树状集合中，其中从该计算机开始测试
运行部署被初始化。

1. 该应用程序SDU被放置在该SDM运行时间服务正在执
行的计算机上的一部部署文件夹中。
- 25 2. 操作者选择一部部署动作（安装，更新，卸载），并提示
域资格输入。
3. 操作者被验证，而且被映射到一部部署任务，该部署任务
确定所验证的用户是否被授权来执行所请求的部署操作。
4. 操作者选择在哪一目标服务器上哪些组件要安装，更新
30 或删除。

5. 该SDM运行服务连接作为一信任服务账号的所选目标
服务器，并执行该操作。

特征描述

特性详细说明

5 该SDM运行时间负责跟踪所有的SDM类，类型和实例。为了部署一分布式
应用程序，该SDM运行时间将公开一组SOAP接口，来在一个SDM文档上注册
和操作。

该SDM运行时间包括以下主要组件：

- 具有相关运行时间库的WebService,
- Windows服务,
- 数据库，例如MSDE（或Yukon）。

图44示出了SDM运行时间组件，部署工具和目标服务器之间的关系。在图
44中，为了启动一部署动作，用户使用部署工具UI或一命令行接口相互作用。

15 运行时间库提供由该WebService所公开的一组SOAP接口。该WebService将
信息写入为了执行部署操作，Windows服务重新找到的数据库中。WebService
使用Windows验证来验证到该SDM运行时间数据库的用户，并且授权基于定义
在数据库的任务的部署动作。

20 在一生产环境中，该Windows服务将作为一有效目录服务账号来执行，而
且该目标服务器将为了管理目的，被配置成信任该域服务器账号。Windows服务
将使用WMI，远程的连接到目标服务器，配置该目标服务器，该目标服务器使
用服务帐号的扮演。在一用户账号的基础上这一信任服务模型必须更可缩放，而
且将最小化管理目标服务器ACL的需要。为了执行部署操作，操作者将不必是
目标服务器上的管理员。

25 在一测试运行环境中，Windows服务将作为有效目录服务账号或者作为一
不在有效目录的无特权网络服务账号来执行。后者将请求在该目标服务器上的一
验证用户账号的扮演。

UI描述

30 没有用于SDM运行时间自身的UI。该SDM运行时间将公开一组API，该API
可以通过一部署工具UI或通过一组命令行工作被调用。该部署工具UI将被指定
在单独的文件中。

安全模型

用于该SDM运行时间的安全模型是信任子系统的模型，其使用固定身份来访问将部署分布式组件的目标服务器。验证用户的安全环境不会贯穿这一模型中的该目标服务器。这一安全模型的基本设想是这样的，该目标服务器信任该 SDM运行时间服务的固定身份，从而为在该目标服务器上的单独用户消除管理管理员权限的需要。图45示出了该固定身份的信任关系。

通过信任子系统模型，当然可能借助一信任域账户运行SDM运行时间服务，以至作为一本本地无特权的网络服务账户来运行该SDM运行时间服务。理解的关键点在于对于任意组件动作的验证是由SDM运行时间，使用基于角色的验证来管理，以及在该目标服务器上，一旦用户已经被验证并被映射到一允许所请求部署操作的任务上，只有SDM运行时间服务可以执行安装，更新和卸载动作。

验证

验证是检验用户身份的过程，该过程基于只有该用户知道的凭证和基本的安全基础机构。为了分布式应用程序部署的目的，使用Windows验证，通过有效目录域账号或者通过本地账号，用户将被验证。如果本地账号被使用，在部署计算机上的本地账号的用户名和密码必须是与在目标服务器上相同。

权限

一旦用户被验证了，用于执行一部署操作，例如安装，更新和删除的验证将基于数据库角色被允许，其中已验证的用户是该数据库角色的一个成员。因为Windows用户和组账号可以是SQL服务器数据库角色的成员，该基本验证次序如下：

1. 使用Windows验证的WebService验证用户。
2. WebService作为验证用户连接数据库。
3. 基于用户或组账号成员资格用户被映射到数据库角色。
4. WebService将部署动作信息写入适当的数据库表中，该表可以被该 SDM运行时间的Windows服务组件异步的读取。

注意，不需要管理该操作系统基础结构外部的密码，也不需管理在该目标服务器上的每一用户ACL。

扮演

扮演是在与当前处理所有者不同的账号的安全上下文中执行代码的能力。

目标服务器的远程连接将被建立，通过使用扮演所允许的WMI。当活动目录存在时，扮演将以该信任服务身份为基础，而当活动目录无效时，将以验证用户的安全上下文为基础（例如，测试运行环境）。

Windows服务

5 运行时间SDM的Windows服务组件必须作为具有该目标服务器上的管理权限的一服务账号来运行。管理权限的需要是由于在该目标服务器上安装软件并创建不同的IIS, SQL设置的需要和注册。

10 当缺少活动目录域账号时，该Windows服务将扮演一用户账号，该用户账号被授权来执行该目标服务器上的管理操作。在这种情况下，该Windows服务将作为一网络服务账号来运行，该网络服务账号不需要密码，而且是一本地计算机上的非特权用户。一旦连接，该Windows服务将向远程计算机给出本地计算机资格。

IIS

SQL服务器

15 SQL服务器能够以两种验证模式进行操作：Windows验证模式和混合模式。因为Windows验证模式比混合模式更安全，用于SDM运行时间数据库的SQL服务器将只被配置为Windows验证模式。这将防止该账号被用来验证SDM运行时间数据库。为了调节该活动目录验证基础结构，用于SDM运行时间数据库的管理特权必须通过Windows组成员资格来控制。通过创建用于管理SQL服务器的活动目录组，并向该组添加特定用户，将更容易控制对SDM运行时间数据库的访问，而不必管理专用账号上的密码。

20 除了该SDM 运行时间数据库之外，运行SQL服务器的目标服务器还必须使用Windows验证模式，而且必须管理通过Windows组成员资格的管理访问。用于SDM运行时间数据库的Windows组和用于目标服务器的Windows组必须是不同的组。它是一策略为用户判定是否具有用于管理该SQL服务器机器的一个或多个Windows组。

例如：

SDM运行时间管理员组

用户A, 用户B

30 SQL服务器目标1管理员组

用户C, 用户D

SQL服务器目标2管理员组

用户C, 用户E

SDM服务器概述

5 介绍

什么是SDM服务器-该SDM服务器是根据SDM所构建（BUILD）的服务组。

目前，存在两种我们可以用在部署工具的架构上的通常方法。在这里每一个都会被概要描述。

分布式方法

10 在这一方法中，依靠运行时间OM客户端库，该SDM运行时间和部署引擎所使用的工具被构建，该客户端库使用WebService依次向该SDM运行时间引擎和用于放置SDU的（二进制文件）共享文件通信。该SDM和部署引擎共享一SDM实体和部署作业的数据库。通过使用WMI和SMB（文件共享）的部署引擎，部署任务被异步的执行，以便与该目标机器进行交互。

15 简化方法

在这一方法中，该客户端，所有SDM目标模型库，SDM引擎，部署引擎和安装插件程序都运行在相同的过程中，从而不存在同样的服务。该运行时间数据库和二进制文件库可以在不同的机器上。该WMI和SMB是直接从正在运行的客户端或UI连接到目标机器上的。

20 用户接口和其他客户端

用于该SDM服务器的用户接口将包括：

- VISUAL STUDIO向导，其将提供一简单的方法来部署，更新或删除应用程序的测试实例。
- 命令行工具来载入SDM服务器，SDU服务器和实例请求。
- 完整的UI，其显露出所有目标模型的功能性，并附加提供图形工具用于组成主机模型和实例请求。

运行时间OM库

连到SDM服务器的公共接口是经由这一库的。它是一可控制代码对象模型，而且使用该接口你可以：

- 在运行时间里管理该SDM服务器。能够载入SDM服务器到该运行时间

中。SDM服务器是强烈指定的和固定的，而且每次被载入一SDM（即，载入一非单个的类型，类或映射SDM文件）。能够从该运行时间里删除SDM服务器，而且在该运行时间里为SDM产生XML文档。当有从在该运行时间里的其他SDM服务器或实例对它的引用时，SDM服务器不能被从该运行时间里删除。

- 5 ● 管理该运行时间所知的SDM服务器。
- 找出并考虑SDM元素（ELEMENT）（来自在该运行时间里被载入的SDM）。不存在为授权一新SDM所提供的API（即，这是遍及该SDM固定元素的一只读目标模型）。这包括SDM服务器，SDU服务器，身份，版本，类，类型，绑定/映射和版权。
- 10 ● 找出并考虑组件，端口，连接线和物理布局（在实例空间中的容纳关系）的实例。在该实例空间中，每一个实例可以由一GUID，一稳定路径或一基于路径的数组来识别。该路径是字符串，而且可以是相对的。包括相对路径的这些标识符允许实例被找到，以及在文档中被引用，例如实例请求文档。
- 15 ● 操作实例包括创建，更改拓扑，升级，改变设置和删除。在实例请求范围内产生实例变更，该请求提供更新的基本单元，从而任意错误或约束违规将导致该整个的请求失败。实例请求还顾及不需要绑定到主机上的，暂时存在的实例，如在该请求被提交时必须具有一主机的实例。该实例请求还顾及到许多操作，该操作将影响一单独组件的安装或要被执行的设置，并且将该安装或设置更新延迟到提交，从而在组件上产生一单独的更新。
- 20 ● 当创建一实例请求时，在实例请求范围内创建先后顺序。先后顺序允许控制由在该组件上安装产生的安装次序，和实例请求。
- 找出并考虑实例请求，包括获得它们的状态，该状态包括所有的错误信息，以及重试由该请求所影响的组件的安装/更新。
- 载入一实例请求。实例请求是表示一组实例空间操作的XML文件。这一文档可以利用指向用于创建或删除应用实例的一可重用的‘脚本’的相对路径。
- 25 ● 在数据库中由一实例请求产生一实例请求文档。这样的文档是可移植的。
- 管理对该SDM服务的安全许可。这包括设置用于操作目标机器的资格和有关实例操作的许可，例如谁可以创建特殊主机实例上所容纳的实例。

- 同意有关上述功能的事件包括，实例请求安装完成。通过载入客户库的该处理的生命周期，限制这些事件预定的生命周期（即，这些是正规的CLR事件）。

SDM运行时间引擎

5 该SDM运行时间引擎执行在该SDM模型上的推理和由该目标模型显露的功能。

10 在该分布式方法中，库作为WebService，通过调用适当的进程，该库与运行时间引擎进行通信，例如载入SDM，创建组件实例以及获得全部SDM（为考虑SDM实体）。这减少到该服务器的往返路程。用于这一WebService的多个参数格式是与用于SDM文件相同模式的XML格式。

15 在某种意义上，仅仅是他更便于使用，该WebService伴随客户库提供SDM服务的所有功能性。在该分布式方法中，引擎执行在许可上的检测（见安全详细说明部分来得到细节）。

插件安装程序

15 插件安装程序与一类容纳关系有关。它们近似的涉及使用在VISUAL STUDIO中的插件程序，该插件程序提供用于类的设计经验，并提出在SDU中的相关二进制文件和部署值。它们提供SDM服务器下述功能：

20 ● 在它们的宿主上安装，卸载和重装组件。当一实例请求引起一个新的组件实例，组件实例的更换或要求重新安装的组件的改变时，该安装程序设置在SDU中的实例，宿主实例，与组件相关的类型，以及与那些类型相关的二进制文件，并实现该实例的安装或卸载。在SDM的应用层，对于安装来说被安装在宿主上最通用的是，仅仅需要base.msi所提供的类型（通过特定参数）和在该宿主上执行的一个第二任务，该第二任务设置了适当的设置和端口视图。

25 ● 当其设置改变时或当来自它的端口之一的视图改变时（由于拓扑改变或者可见端口设置改变），更新一组件实例。在SDM的应用层上，对这一情况来说最通用的是安装的第二部分的再运行。

30 ● 映射可视端口到端口用来设置一个安装组件上的实例。在SDM和组件实例中具有端口实例，即，作为某些连接线拓扑的结果，允许该端口实例看见其他端口实例的详细内容，通常从而组件能够绑定到端口。例如，一ASP.NET网点可以具有数据库客户端端口实例，这样它就能够联入数据库。当正确连线时

它的数据库客户端口能够看见一单独的数据库服务器端口实例和在那个服务器端口上的设置。这一信息被该ASP.NET安装程序用来为服务器放置连接线，该服务器是在web.config文件中该客户端口名下的服务器。

● 该安装程序还提供执行主机和它们的客户之间的约束检测的代码。这一检测由没有在上述分布式方法中示出的SDM引擎执行。大部分安装程序预期的使用基于XML，XPath和XQuery的通用约束语言。

- 检查设置（Audit Settings）
- 检查存在（Audit Existence）
- 检查完全（Auditfull）
- 检查所容纳实例（Audit Hosted Instances）

10 映射设置到组件

接口

15 提供一组基础机构给安装程序，例如作为在宿主上的本地系统执行命令。往后，其他进一步提供仅仅需要一个网络地址和帐号的机构。接口是可管理代码。

设计

以下部分介绍如何设计数据中心和象数据中心一样被容纳的分布式应用程序。设计者使用SDM来建模各种构建，该构建被使用在架构物理资源（例如，硬件，网络，主机服务器）。

20 数据中心描述

这一部分介绍如何不表示特殊资源来构造数据中心组件模型，而不表示特殊资源，例如许多机器。使用服务定义模型（SDM）语义，其提供一物理数据中心环境的比例不变的模型。

25 一虚拟数据中心（VDC）是物理数据中心环境的一逻辑表示，其简化了该数据中心的开发者视图。理论上说，IT专业人员或架构师应该能够以与开发者能够描述一分布式应用程序/服务的相同的比例不变方式，描述该数据中心。该VDC是该数据中心中的服务器，网络和存储资源和它们的拓扑相关性的抽象。

30 一典型的数据中心图表是非常复杂的，由于多个相互连接服务器，网络设备，IP地址，VLAN，操作系统，存储器，等等所有都被表示在一张单独的使用visio或简单工具所画出的一个单独的表中。除了该图表之外，通常存在精确规

定该数据中心如何分区，配置和管理的长文档。

这一复杂性的例子是微软系统架构（MSA）企业数据中心（EDC）。显而易见的，通过当前数据中心状态保存手工绘制图表和文档，随着时间推移，当更新和升级被应用时，该保存变的成本很高，不然的话就变成不可能的。另外，依靠文档的规定确认环境的能力是困难的，并且倾向于人为错误。

以比例不变的模式表示例如MSA EDC的复杂数据中心的能力对于开发者和IT专业人员两者都非常强大。使用组件、端口和连接线描述一个数据中心的能力提供一个有效的框架，从而在该框架中模拟和确认部署要求，该模拟和确认在现在的设计和部署过程中是缺少的。

该数据中心描述的一个方面是为集合计算环境来虚拟化硬件和配置机构性能。在传统的数据中心环境中，典型的操作者增加特殊硬件环境到一特定应用程序。例如，当部署一新的邮件系统到该数据中心中时，该操作者将购买一组服务器，为不同的网络像备份和数据区域增加网络适配器，以及增加像转换器和负载平衡器的网络硬件。用于该应用程序的硬件部署需要扩大物理工作。

不止人工所构造的这些，应用程序特殊硬件配置创建是昂贵的，而且它们不易于修改；它们固有的属性导致很少的资源使用，由于资源可以象工作负荷改变一样，很容易的被移动到新的应用程序中。

这一公开描述了一种创建数据中心虚拟化环境的方法，该方法允许操作者运行物理资源的单独池，该资源包括服务器，存储器，和网络设备。从那个单独池中，资源被分配，并被配置以符合应用程序的需要。一组资源提供者跟踪资源的所有权，并知道如何配置资源来符合应用程序的需要。

当部署一新的应用程序到该数据中心环境中时，操作者创建该应用程序所需资源的抽象描述。一请求被传送给该服务平台，要求该抽象描述被解析成真实的资源。该服务平台借助资源管理器工作，以便定位可以实现请求的资源，选择最经济的实现请求的资源，标记资源为已使用的，配置该资源以便符合请求的要求，以及将所分配资源的具体描述放置到该抽象描述中。如果该应用程序需求改变，操作者就更新资源描述，并要求服务平台解析该更新应用程序描述。个别的资源提供者可以使用硬件或配置物理资源的OS特殊软件驱动器来符合应用程序的需要。

与数据中心描述相关的概念包括（1）一图表语言，用于描述想要的资源，

资源请求，以及所允许的资源；（2）一组域特定资源提供者，具有给定类型的可用资源的知识，和配置那些资源以符合应用程序需求的能力；以及（3）一资源管理器，其处理资源请求，与资源提供者通信以便找到适当的有用资源，有选择的优化特殊资源的选择，要求资源提供者来配置所选资源，以及更新该资源请求来反映所选资源。

应用程序描述

使用SDM语义方法同样的可以定义应用程序。参照在段0开始的SDM部分，在上面这一内容以被更详细描述出来。图20示出了图形用户接口（UI），其允许架构师根据SDM语义方法描述一大型的分布式应用程序。

10 在物理系统上的应用程序的逻辑布局

一旦使用SDM语义方法，架构该应用程序和虚拟数据中心，架构师就可以逻辑的在该虚拟硬件元素上尝试程序元素不同的逻辑布局。用于不同的部署环境（部署，测试，生产，等等）可以存在不同的逻辑布局。在设计时段里，可以做出逻辑布局，检测请求和约束，以及报警开发者任意错误和警告。随着使用在每一个组件，端口和连接线类上所指定的XPath和XSD，实现约束检测，该逻辑布局的结果被记录在单独的文件中。这一内容在图21中被说明。为了直观的表示，当在物理元素上放置不同的应用程序元素时，设计者可以虚拟化一UI（用户接口）。

设计时段确认

20 以下部分介绍一种方法，在物理资源上的设计应用程序逻辑布局的时间有效。SDM组件，端口和连接线的增强添加层和层间的映射图，来完成分布式应用程序设计和部署要求的设计时段有效。

虽然当与宿主，工厂，资源管理器，以及SDM运行时间组合起来时，组件，端口和连接线是强大的抽象，它们不能充分来部署并管理一分布式应用程序/服务。为了创建并管理这些逻辑抽象的物理实例，某些附加结构被包括。那些附加结构是层和映射。

层

图11示出了由SDM定义的层抽象。

30 应用程序层描述在应用程序/服务的上下文中的，分布的组件，它们的部署需求和约束，以及它们的通信关系。

部署层描述用于宿主，例如IIS，CLR和SQL的，配置和策略设置与约束。

该虚拟数据中心（VDC）层描述从操作系统通过网络拓扑下传到服务器，网络和存储设备的，数据中心环境设置和约束。

该硬件层描述了物理数据中心环境，并且以说明的方式来公开或指定，
5 例如使用XML。这一层不是比例不变的，因此不在该SDM中被建模，而是为了完整被包括在SDM中。

映射

因为该SDM是分层的，就需要有一种方法来在多种层之间进行绑定。一个
10 映射实质上是一层中的组件或端口与下面的邻接层中的组件或端口之间的绑定。一
映射可以被描述如下：

$$MT = [T_n \rightarrow T_{n-1}] + [T_{n-1} \rightarrow T_{n-2}] + [T_{n-2} \rightarrow T_{n-3}] [\dots]$$

其中M表示一映射，而T表示一组件，端口或连接线，而n表示该层。数组
符号表示映射的方向，其总是从高层到低层。

例如在图12中，在应用层的，被命名为MyFrontEnd的组件被映射到被称为
15 IIS的部署层的组件上。同样的，名为MyBackEnd的组件被映射到部署层上的SQL
组件上。

设计时段有效

在活动的数据中心里，在应用程序/服务实际上被部署之前，一组件与其
20 下面一层上的它的宿主组件之间的绑定可以将问题呈现给开发者。这一问题可能是由于不一致的类型，配置冲突，不匹配操作，缺少拓扑相关性，等等。例如，在图13中所描述的所尝试的映射会导致一个错误，因为在部署层上的IIS和SQL组件之间，没有可能的通信关系。

虽然，从MyBackEnd组件到SQL主组件的映射可能已经是基于该组件和宿
主类型的一致性和缺乏配置冲突的有效绑定，但是由于MyService SDM所定
25 定的，MyFrontEnd和MyBackEnd之间的拓扑关系并不存在于所指定部署层上，
该映射是无效的。

分层架构

图48示出了一平台架构，用于在分布式计算系统中的分布式应用程序的自动
30 设计，部署和管理。该架构示出了一基层302之上的多个层，该基层表示该分布
式计算系统中的物理计算机资源。一自动部署服务层304提供工具，从而将机

器转换成在该分布式计算系统所使用的服务器。这样的工具允许OS（操作系统）图像的创建，编辑和部署。使用完全程序化接口完成机器的远程程序设计，例如WMI（Windows管理工具），该完全程序化接口是Microsoft's Windows操作系统中的一编程接口（API），该Microsoft's Windows操作系统允许系统和网络设备被配置和管理。

—网络管理层306位于该自动部署服务层304的上面。该网络管理层306考虑网络管理和虚拟拓扑的产生。部分的，该网络管理层支持一用于网络计算机的驱动模型，该驱动模型经由连到该网关的相关端口的一单独的物理网络接口，使单独计算机到一个或多个VLAN的通过连到网络转换器的相关端口的一单独的物理网络接口的连接变得容易。根据该驱动模型，一VLAN驱动器被安装在服务器上，并被用于在单独物理网络接口之上创建虚拟网络接口（VNIC）。该VLAN驱动器为每一个VLAN创建一个虚拟网络接口VNIC。该VNIC正好位于服务器上的IP堆栈中的网络接口（NIC）之上，以便该服务器可以处理经由多个VLAN所传送的包，即使所有的包都经过相同的物理NIC传送。

该驱动器模型支持VLAN标记，从而允许使用与数据包所属于的VLAN相同的身份来标记数据包，其中，该数据包是经由该分布式计算系统被传送的。该网络转换器实施该标记，并且仅仅接收带有标识该网络转换器所属的VLAN的标记的包。在一个实施例中，该网络转换器具有标记端口和无标记端口两者。转换器的标记端口用VLAN标识符来标记，而且在与其它转换器的标记端口连接时被使用。这允许通过转换器网络包的快速传输。转换器的标记端口用来连接到服务器或计算机。当包到达它们的目标服务器时，在这个包上行到服务器之前，VLAN标记被从该包中提取出来，以便该服务器不需要知道有关该标记的任何东西。

—物理资源管理层308位于该网络管理层306的上面。该物理资源管理层308维持一分布式计算系统的物理模型，跟踪所有权和所有物理计算资源的协调分配。该物理管理层308进一步支持成批资源分配，由此能够动态的配置并管理物理计算资源。

—逻辑资源管理层310位于该物理资源管理层308的上面。该逻辑资源管理层310使该分布式应用程序所需要的逻辑资源的分配变的容易。例如，该应用程序可以请求这样的资源，如数据库，负载平衡服务，防火墙，WebService，等等。

该逻辑资源管理层310公开了这样的逻辑资源。

接下来的层是服务定义模型和运行时间层312，其允许该分布式应用程序的描述和对它的操作的跟踪。该服务定义模型（SDM）提供一名字空间和上下文，用于描述操作处理，以及一API，用于应用程序自检和应用程序资源的控制。进一步的使操作者和开发者能够共享通用应用视图。

位于计算资源层上面的第六层是组件层314。这一层允许对一分布式应用程序的可重用构件的定义，该分布式应用程序使用用于上下文，命名和绑定的SDM API。

顶层是操作逻辑层316，其调节分布式应用程序的操作状况。该操作逻辑负责开始服务，增长和缩小服务，升级和降级，错误检测和恢复，以及状态划分。该操作逻辑允许遍及部署和应用程序的验证操作上的执行的重用。通过SDM层的使用，该操作逻辑具有上下文来更好的理解可能出现的结果。例如，当故障出现时，操作逻辑可以确定该故障出现在一邮件服务的前端，而不是正好在中间位置的某个服务器上。

15 部署

以下部分介绍该数据中心和分布式应用程序的部署。其包括逻辑模型，应用程序物理布局，以及应用程序和数据中心部署的示例。图23一般说明了该部署步骤。

示例

由于SDM类型是比例不变的，并可以被创建成任意比例，部署的一方面是定义要为已给逻辑组件和布线拓扑创建的实例的数量，以便物理的实现该硬件/应用程序。一个实例请求文档被创建来提供需要被创建的实例的说明性定义。

应用程序的物理布局

物理布局是选择特殊主机实例的动作，这是部署的目标。物理布局受逻辑布局约束，并且在物理布局过程中使该约束重新生效。该物理布局被保存在物理布局文件中。

数据中心和应用程序部署

该SDU，逻辑布局文件，实例请求和物理布局文件不断供给SDM运行时间。该SDM运行时间调用适当的安装程序（基于类和容纳关系），该安装程序负责在宿主上创建一个新实例，并配置该实例以符合类型上的设置值。SDM运行时间

将包括所有实例，它们的最终设置值和布局的一数据库，一运行时间API支持该实例空间的查询。

BIG部署工具

情况描述

5 特征概述

该BIG部署工具执行分布式应用程序部署，用于数据中心操作者，并用于开发者测试他们的应用程序。它使用服务定义模型（SDM）应用程序，该程序包括应用程序的位（SDU），映射文件和一组部署约束。用户指定该应用程序在他/她服务器上的布局，并提供部署时间设置。该工具依靠远程机器安装或卸载实例，并将状态提供给操作者。之后，该操作者能够添加新的实例，非现行实例，并重新配置该应用程序的拓扑。
10

情况

一个大型企业体具有分散的数据中心和开发者机构。该数据中心部署，维持并容纳用于终端用户的应用程序，该终端用户适合于雇员和用户两者。该数据
15 中心的拓扑经常改变，并且非常符合MSA EDC 1.5，该MSA EDC 1.5不是一个BIG计算机。

该数据中心组织提供开发者它的容纳策略的比例不变的抽象，我们称该抽象为逻辑信息模型（LIM）。该策略指定宿主配置，包括在应用程序上的约束，可允许设置和基本拓扑。

20 该开发者组织代码并热修复这些应用程序从而符合最终用户的需要，并从而处于该数据中心策略中。该开发者通过指定该应用程序的需求和所期望主机，来提供部署引导。

该应用程序操作者使用该BIG部署工具来部署该数据中心中的应用程序。
25 该部署工具使用该开发者引导和数据中心策略来确认适当的部署。之后，该应
用程序操作者使用该工具来按比例放大，重新配置该应用程序拓扑，或者卸载。

特征描述

性能的详细说明

该工具如何与Whidbey和其它产品配合的概述如下所示。注意，该SDM运行时间LIM，SDM/SDU和Whidbey在其它部分被详细描述。图49说明了一个应
30 用程序部署的示例使用流程。

在图49中通信的关键点是（从左到右）：

该开发者提供一应用程序SDU，该应用程序SDU包括SDM，二进制文件和SDU映射。（我们使用二进制文件指应用程序位和内容。）

该开发者&数据中心组织是分开的，但共享相同的LIM。

5 在运行该部署工具的机器上，与存储和API一起，有SDM运行时间。

该应用程序操作者负责该数据中心描述；部署描述；并使用LIM，SDU和SDU映射。

一代理和“小型工厂”位于目标服务器上，该目标服务器采用SDU，部署描述和数据中心描述程序作为用于部署的输入。

10 该代理使用一通用小型工厂API来与该小型工厂进行交互。

在这一例子中小型工厂是SQL和IIS，但可以扩展为其它产品。这些将完成安装，配置和卸载的工作。

设置和约束的概述

15 该BIG部署工具使用SDM应用程序。为了理解该工具将如何使用设置和约束，这一部分提供一与SDM一起的设置和约束的基本的概述。为了完整的解释设置，约束和模式，可参见相关部分。在这一论述中，我们不区别是否该设置/约束是在SDM的元类型，类型或成员上的。

20 通过该SDM模型，开发者，网络架构和应用程序操作者将具有提供设置/约束（网络架构和开发者），SDU映射（开发者）和部署时间设置（应用程序操作者）的能力。对每个宿主（即IIS，SQL和BizTalk）通过每个它自身的模式，规则和值限定这些约束和设置的范围。

每一个宿主所揭露的设置群将被分成通过应用程序设置的那些设置和由主机保存的那些设置。我们称该模型为应用程序设置，并在之后称为宿主设置。

25 另外，一宿主通过指定“宿主约束”来限制该应用程序设置，并且一应用程序通过该“应用程序约束”，给出宿主设置上的在前需求。限制可以在值，特定值，或关系式的设置域上。

以下表概要介绍了用于宿主对比应用程序的设置和约束。

表1：设置定义

设置/约束的定义	例子
应用程序设置-由开发者所作的，关于应用程序的设置	shopping 应用程序： maxWorkerThreads=8 401k应用程序: maxWorkerThreads=4
应用程序约束-依据‘宿主设置’所需要来运行该应用程序的额外的约束	Mode=WorkerProcessIsolationMode
宿主设置-设置组，用于在那个资源上所容纳的所有应用程序	Mode=WorkerProcessIsolationMode
宿主约束-依据应用程序设置的限定（正确值，值域）	High-perf host: maxWorkerThreads<25 Best-effort hosts : maxWorkerThreads<5

该逻辑信息模型 (LIM) 的目的是提供一数据中心的策略和部署块的抽象视图。该LIM说明了宿主性对于应用程序的约束/设置的区别；宿主设置；和应用程序约束。该LIM获取的策略由网络架构师创造。这一策略可以由网络架构师，开发者编码成LIM文件，或通过标准Microsoft LIM的使用可以变的容易，该标准Microsoft LIM通过Notepad (记事本) 来编辑。

然后，该LIM由开发者使用，来编写应用程序，并根据它的数据中心的表示进行测试。作为应用程序的一部分，开发者提供用于LIM允许的应用程序设置的值，用于将运行该应用程序的宿主的宿主约束，以及关于宿主上组件布局的元数据。通过一映射文件，开发者提供关于映射到宿主上的应用程序布局的引导。非特定的设置将作为部署时间设置被通过，应用程序操作者将提供该部署时间设置 (即IP地址或App_pool_ID)。

一基本例子将是一网络架构，该网络架构为客户指定不同的宿主，约束该客户在High-perf对Best-effort 的宿主上购买服务。该宿主约束可以限定IO数量或不同的WorkerThread。使用IIS_6的新的模式，在这一例子中的High-perf和Best-effort的宿主设置是相同的。该开发者编写两个具有不同预算和需求的应用程序。第一shopping应用程序想要更多的WorkerThread。该401K应用程序有较少的差别。两种应用程序都约束 (需求) 以WorkerProcessIsolationMode方式运行。

图50说明应用程序对宿主设置和约束。

部署阶段

使用该BIG部署工具，关于下面所示出的SDM应用程序部署存在四个阶段。

图51说明了一部署工具的示例阶段。

- 5 初始阶段是该LIM被提出从而以比例不变的方式来表示该数据中心，然后被用来创建一硬件分类文件（数据中心描述）。

应用程序部署阶段是开发者依靠LIM编码，并使用部署工具API来测试和调试他/她的SDM应用程序的时段。

安装阶段是该应用程序操作者在一个已配置的机器上安装应用程序的情况。

- 10 运行阶段是该应用程序操作者按比例放大，重新配置拓扑，或卸载一已经在运行的应用程序的时段。

注意，遍及这一文档，尤其是在流程图中，我们使用术语“部署”来包括所有必须的宿主设置/约束检测，标记宿主对应用程序的不匹配，记录应用程序设置，以及调用该小型工厂（mini-factory）活动。小型工厂活动是所有那些执行安装；卸载；配置的活动；并且转入Fusion，MSI或将来的Microsoft安装程序中的活动。

初始化阶段

该初始化阶段是该LIM和数据中心描述程序被创建的时段。

该数据中心的网络架构师从Microsoft.com中选择并下载最匹配的，数位有20 符号的（digitally-signed）LIM。然后，该网络架构师编辑该文件，来反映想要的数据中心策略，包括网络拓扑，被允许的应用程序设计，以及容纳约束。

另外，一LIM可以在一Visual Studio Whidbey设计层面被编写。然后，该处理流程将是一网络架构，该网络架构给出该开发者组织所有相关策略和拓扑信息，该处理流程今天被保存在Word文档和Visio图表中。然后，该开发者创建描述数据中心的适当的LIM，并与网络架构重复以便确认正确性。

一旦该LIM被创建，该数据中心组织就根据该LIM，通过创建数据中心描述文件来对它们的硬件分类。该数据中心描述依靠正在运行的硬件映射该LIM组件，其被我们称为分类活动。因此，该数据中心描述不是比例不变的，而且包括象IP地址这样的机器特定细节。下图设想了一数据中心描述符，但不建议30 UI。注意，一LIM具有“IIS gold”和“IIS silver”逻辑宿主的概念。在该数据中心

描述符中，这些逻辑宿主被映射到物理机器上，因此，我们有映射到IP地址192.168.11.2的一IIS[1] gold，在IP地址192.168.11.3上的一IIS[2] gold，等等。图52说明一数据中心描述的直观示例。

注意，作为该数据中心操作者安装/配置服务器，网络，资源，应用程序下面的一切，需要停留在LIM中的活动。（记住该数据中心操作者负责应用程序下面的一切。）给网络架构师和数据中心操作两者在部署工具外部执行它们的任务。

应用程序开发阶段

在这一阶段中，开发者依靠LIM编码，并使用BIG部署工具API来测试/调试部署。由该数据中心提供这一LIM，或者由开发者组织代表该数据中心修改这一LIM。（如上面所述）

该部署工具API允许Visual Studio Whidbey来执行他们的“F5”和“测试/调试”部署的两种情况。该“F5”和“测试/调试”部署分别对应一单独的开发者框和多个机器。在该“F5”情况中，必要位已经在目标单独的开发框上。该“测试/调试”情况需要部署工具传输位到目标机器中，象在通常部署中那样。然而，F5和测试/调试情况能够使开发这被警告开发者关于冲突设置，和改写应用程序和宿主设置。（通常，只有该应用程序设置可以由该部署工具写入。）注意，这些VS情况将不使用该SDM运行时间。图53描述这些VS情况。

用于该Visual Studio “F5”和“测试/调试”情况的重要的说明限定是：

该BIG部署工具API将通过向导从VS调入。

该VS向导将选择机器从而依靠和采用部署时间设置进行部署，（即IP_address 或 App_pool_ID=17）。

VS将实现用户接口。

在F5情况下，SDM，SDU，二进制文件，以及所有位都已经在目标单独部署框上了。从而，写设置是所需要的全部。

在测试/调试循环中，“部署”包括写入必要的设置。

两种情况标记何时设置冲突，并且允许改写目标机器的设置，该两种情况包括主机和应用程序。

在图53中没有示出开发者依靠LIM和到LIM的SDU映射的概念编码应用程序。（关于LIM的更多介绍，参见LIM/LID部分。）开发者传送SDU给应用程序操作者，该SDU包括SDM，二进制文件，以及SDU映射文件。

安装阶段

在该安装阶段，该操作者被提供给应用程序（经映射的SDU）和数据中心描述符（其提供LIM）。

对于图54描述应用程序安装，以下说明限定是重要的：

5 应用程序操作者运行该工具（GUI/CLI）。

复制并载入应用程序所有的文件和数据中心描述。

在SDM运行时间里，应用程序被注册。

应用程序操作者选择该应用程序组件的宿主/机器。（例子在下一部分给出。）

10 在该选择过程中（我们称为映射），依靠该领域的运行时间图核对约束。我们不保证在引起脱离视图的这一工具的外部，是否可以修改设置。

部署执行宿主对应用程序的约束/设置检查和安装。（注意，通过远离网络层的高速缓存器上的文件隐藏和ACL文件设置来避免网络分层，该实现可能是更加复杂的。）

15 通过UI或文档工具解释了我们没有处理的状态的数据（例如填充SQL数据库）。

上述步骤产生部署描述，由于所指定部署上述步骤可以被重复使用或修改。（例子在下一部分给出。）

一预览功能允许该应用程序操作者获得该工具将产生的改变列表。然后用户就可以使用部署描述符所产生的预览再运行该工具，。

20 一已经产生的部署描述可以被载入并运行，假设该SDM运行时间知道有该应用程序，应用程序位还是有效的，而且相同的数据中心描述也是有效的。图54说明了安装情况的例子。

指定部署的例子

25 为了阐明要指定部署所需的数据流，我们使用通过一LIM限定它们的数据中心的MSN的例子。

该LIM可以是数位有符号的，时间标记的，以及有版本的。该开发组织使用LIM来编码被容纳在MSN数据中心宿主上的分两层的应用程序（IIS和SQL服务器）。开发者指定产生一SDU映射文件的宿主，在该宿主上可以容纳一组件。我们在图55中示出了这个MSN例子。

30 关于图55和在应用程序部署中的数据流，以下是重要的：

SDU包括SDM。

开发者映射SDU组件到LIM (MSNdatacenter.LIMSDU)，创建一SDU映射文件。该映射是允许的布局。

5 数据中心描述根据LIM组件分类实际/物理服务器，并且该数据中心描述不是比例不变的。

SDU, SDU映射，数据中心描述和用户输入提供给该部署工具来创建部署描述符。

该部署描述符指定组件（来自SDU）要安装在哪个机器上（来自数据中心描述）。

10 该部署描述符获得部署时间设置，例如URL。

图55说明了产生一部署描述符文件的例子。

在上述例子中，该SDU映射文件说明开发者绑定SDM 的Component 2TierApp. MyWeb 到MSN所约束的主机组件Component MSN9. IIS_MSN，并同样用于2 TierApp •MyDB-> MSN9 . SQL_MSN。（我们指定复合组件来在多
15 Myweb事件中消除歧义）

该网络架构编辑，该MSNdatacenter.LIM描述IIS和SQL约束和设置是如何被配置的。这一LIM是比例不变的，因为它描述了IIS和SQL宿主，不是正在运行IIS或SQL的机器。然后，该数据中心派生数据中心描述符，该数据中心描述符表示如在LIM中所配置的，哪一个机器正在运行IIS和SQL。我们使用符号
20 IIS_MSN[1]和SQL_MSN[1]来标记存在两个机器正运行该IIS_MSN的组件。

该BIG部署工具作为输入SDU, SDU映射数据中心描述符，部署设置（由用户提供），并产生部署描述符。在我们的例子中，该部署描述符指定一部署。运行它意味着将使软件在目标服务器上被安装/按比例放大/重新配置/卸载。

如在该部署描述符文本中详细描述的，MyWeb[1]的一实例将被安装在服务器IIS_MSN[1]上，MyWeb[2]将被安装在服务器IIS_MSN[2]上，以及MyDB[1]将被安装在服务器SQL_MSN[1]上。部署时间设置由应用程序操作者，例如IP地址或App-Pool-ID提供。注意，倘若它所依赖的文件存在，这一部署描述可以被重复的使用。

运行阶段

30 按比例放大[缩小]情况

对于一已经正在运行的应用程序，该按比例放大[缩小]情况允许该应用程序操作者来添加[删除]一组件，端口，或连接线。这一特征用途的例子是该 Joe_Millionaire 站点经历通信量显著增加，并且想要仅在正规TV时期里按比例放大，而在以后按比例缩小（或每个晚上）。

5 在应用程序按比例放大[缩小]流程图中，以下是重点：

按比例放大[缩小]是安装的子集。

该应用程序操作者选择—运行SDM应用程序，而且可以：

添加组件，端口和连接线，并输入部署设置。

删除组件，端口和连接线。

10 由于先前所产生或所修改的部署描述符，情况能够被运行。（具有相同的数据中心描述符/LIM所提供更早的警告，对应用程序的访问，以及SDM运行时间仍然具有注册应用程序）图56说明一个按比例放大情况的例子。

拓扑重新配置情况

15 该拓扑重新配置允许应用程序操作者不需要卸载，重新安装就可以为一运行应用程序重新布线。重新布线的例子是将你的前端数据库变为现在指向的一个新的后端数据库。

在拓扑重新配置中重点是：

这一情况与按比例放大是不一样的，在该情况下允许对一已经存在的端口和连接线进行编辑，而不需要卸载和重装。

20 其潜在的允许用户“桥接”两个不同的SDM应用程序。

图57说明了一个拓扑重新配置情况的例子。

拓扑重新配置在不想重新部署整个应用程序的故障情况下是有用的。作为一个例子，护照在后端存储了我的所有信用卡密码，并通过一IIS前端变成有效。该前端故障而我不想重新部署/转移数据。相反的，我部署一个新的前端（象正常安装部分），并为该新的前端到我的护照数据库重新布线。

一个拓扑重新配置的桥接例子是如果该beta_MSN10应用程序想要共享 MSN9应用程序的数据库。该应用程序操作者正常部署该beta_MSN10。现在，该beta_MSN10前端需要与MSN9数据库通信，请求一在MSN9数据库上的重新配置（和新的连接线）。

30 卸载情况

通过该卸载情况，应用程序操作者选择应用程序，所有运行实例被删除，运行时间被更新。因为其可能经由按比例缩小情况，用户不选择具体的实例来卸载。

对于该卸载情况以下点是重要的：

5 通过一存在的部署描述符（可能的编辑）程序卸载可以被执行。

用户选择应用程序来卸载，并消除所有实例

状态内容必须在这一工具的外部通过现有的方法被破坏。

图58说明一卸载情况例子。

管理

10 以下部分介绍了在它们被部署之后，对该数据中心和分布式应用程序的管理。首先，一基于模型的管理工具被描述，随后详述自检测/跟踪机构和操作逻辑。

基于模型的管理

15 基于模型的管理（或Ops逻辑）是处理，该处理将基于对操作者和应用程序开发者的意图和策略的定义，以应用程序的基于SDM模型的方式，从物理环境接收事件触发，而且将激活并与一系列在该模型的上下文中的任务或处理配合，将促成改变，并且将提供模型和物理环境之间的相容性。

20 一触发或请求或其他阀值将是一事件，该事件针对在该SDM中的特定的实例。该组件实例将接收该触发，并且基于其它关于其自身的，在被表示在SDM中的全部应用程序和硬件环境的上下文的细节，该事件将开始顺序步骤来定位由触发所识别的发布。来自SDM的应用程序和资源的上下文，给出这一自动操作的丰富和能力，从而提供服务操作人员更容易的操作性能。

图59是模型（BIG）总体架构，该模型中的管理片断我们称为Ops Logic或者基于模型的管理。概括在总体架构中的处理的所建议流程：

25 ● 一应用程序开发者将能够定义一个新的应用程序的集合模型（SDM），或组件类型的类别，其将组成终端用户应用程序或服务。

● 通过在模型中注释组件类型和用于操作的策略和准则，该开发者或操作开发者也将能够给模型添加“操作者计划”到模型中，例如设置必须正在运行的最少的服务器。

30 ● 用于一所述应用程序的特定实现的，所实现的该SDM运行时间或实例的

单元模型将以Unit Model单元模型被保存。在实例保存所想要的状态的每个机器和一物理机器之间存在一对一的通信。

- BIG的资源管理器将通过单元模式工作，以便在服务器物理环境中实现改变。
- 5 ● 每一个服务器将部分的由BIG管理，并且部分可以由操作者在模型外部进行管理。
- 在集合模型和单元模型之间的是种基于模型处理类型，该类型通过该物理机器的模型配合改变，并实现操作者目的。
 - 另一基于模型的处理的类型将遵循另一种方法，并提供在物理空间和模
- 10 型之间的一致性。
- 在管理区域，该监视系统将收集事件，并将它们分组放入警告中。
 - 预定事件和警告的组件将被通知以重要事件。该事件信息与运行时间SDM单元或所被包含实例的有关信息将传送给预定组件，提供了到模型的映射。
 - 如果事件是一操作上的触发，该事件将触发基于模型的处理，该处理可以通过一类配合ops任务，促成在物理机器中的改变。

图60示出了管理的代理层。这是上述整体架构图表模型部分的放大，该图已经变为水平的，以便集合模型对应于SDM，而单元模型对应于SDM实例空间。整体资源管理器管理对个体资源管理器（也被称为工厂）的请求。

自检/跟踪机构

20 如果给出触发，例如一用户请求，一硬件触发或达到的硬件阀值，一合适的可操作处理将被激活。该可操作处理是一组将被执行的操作任务。该操作任务的执行需要配合处理，因为每一个任务是一个可以是长寿命的事务，而且要求在下一个任务之前初始化并完成。监视这一类活动来执行可操作处理的引擎是用于Ops Logic的配合引擎。

25 对潜在的分布式服务器或者硬件资源上的一类操作任务使用配合是唯一的方法。Ops Logic的这些特性为事务处理产生了更加完善的途径：

- 长寿命-可操作处理可以在长时间段里运行，例如整天或整月。
- 异步性-触发或事件可以开始一事务或一处理，但是直到所触发任务结束才能等待处理其它事务。
- 处理事务-在一个操作处理中的步骤是动作，该动作具有一个开始或发送

它的代理，一个接收并处理它的代理，以及一个如果该任务可能失败就停止不作改变的补偿处理。

- 持久-Ops处理需要能够持续很长时间，而不会变得故障或不稳定。
- 高效性-向可靠性一样尽可能变得有效，是作为高效性BIG计算和服务的
5 操作处理的要求。

Ops Logic将基于在SDM环境中的触发，向操作和应用程序开发者提供编码并标准化操作动作顺序的可能。一旦触发被引发，相关序列的任务就被激活。用于特定状态的步骤可以包括对机器的命令，在应用程序组件实例中的改变，或者在模型或者人为步骤中的改变。每一步是一事务，该事务具有一开始和一
10 结束，而且可以成功或失败。通过使用一配合引擎来单步执行这些任务，该处理将被管理，跟踪并向上报告。该配合引擎将启动一任务，监视它的进展，并注意它的结束或失败。依靠该操作处理如何被定义，配合还将使可选的动作能够发生在部分或完全故障的事件中。参见图61。

资源管理器

15 该资源管理器负责在该分布式计算系统中分配逻辑和物理资源。该资源管理器发现有用的硬件，处理资源分配请求，并跟踪逻辑和物理资源的所有权。通过提供到一动态资源池的接口，该资源管理器提供在服务器中的有效性和缩放性的基础。

20 该资源管理器拥有并控制所有在分布式计算系统中的硬件，该分布式计算系统包括计算机和网络设备，例如转换器。在对系统中硬件资源的访问通过资源管理器来控制。另外，该资源管理器提供用于控制逻辑资源的基础机构，例如负载平衡组。

25 该资源管理器为系统中所有的资源管理提供一通用API。服务和运行时间通过该资源管理器API转换，从而产生资源查询，分配资源，改变资源需要，和释放资源。

BIG资源管理

介绍

特征概述

30 BIG定义一分布式服务运行时间，一通用硬件参考平台，以及一资源管理器。该分布式服务运行时间提供一服务，该服务具有一服务组件，它们的相互

关系和—执行环境的架构定义，并以可操作逻辑的形式提供—用于缩放性和有效性策略的执行环境。该硬件参考平台定义—通用硬件结构，该结构使服务能够运行从一个到几千计算机的系统上。

该BIG资源管理器负责在该BIG计算机中分配逻辑和物理资源。该资源管理器发现有用的硬件，处理资源分配请求，并跟踪逻辑和物理资源的所有权。通过提供到—动态资源池的接口，该资源管理器提供在服务器中的有效性和缩放性的基础。

这一文档描述该BIG资源管理器的目的，架构和实现。段1描述目的和驱动情况。段2描述资源管理器的架构和它的相关的资源提供者。段3描述实现细节和API的实现。

BIG资源管理器负责在BIG计算机中分配管理和资源使用的管理。该BIG资源管理器拥有并控制所有在BIG计算机中的硬件，该BIG计算机包括计算机和网络设备两者，例如转换器。对BIG计算机中硬件资源的访问通过资源管理器来控制。另外，资源管理器提供用于控制逻辑资源的基础机构，例如负载平衡组。

资源管理器BIG为计算机中所有的资源管理提供—通用API。服务和BIG运行时间通过该资源管理器API转换，从而产生资源查询，分配资源，改变资源需要，和释放资源。

资源提供者

虽然该资源提供者提供—进入资源管理的通用接口，但是实际资源的知识来自一组资源提供者。一资源提供者具有用于特定类别的资源的存在和管理的特殊知识。例如，该网络资源提供者了解管理VLAN的存在和细节。在BIG中的其他资源管理提供者包括物理设备提供者，IIS VRoot提供者，SQL数据库提供者，CLR AppDomain提供者和Win32表面资源提供者。

通过特殊资源知识，资源提供者扩展了资源管理器。资源提供者管理特殊资源请求到通用查询格式的转换。通过经由提供者辅助DLL的特殊资源配置API，资源提供者扩展该资源管理器API。最后，资源提供者添加合适的状态到该资源管理器数据存储中，从而允许资源特殊信息的跟踪。高层资源提供者构建在低层资源提供者之上。例如，该IIS VRoot提供者通过物理设备提供者分配机器。资源提供者的分层最小化了冗余，而且提高了资源管理的一致性。

在Windows中的I/O管理系统和BIG中的资源管理系统之间的强模拟能够被

描绘出来。象Windows的I/O管理器一样，BIG资源管理器提供一通用API，用于资源访问控制的通用逻辑，通用资源跟踪和经由各种类组的提供者传送请求的通用机构。像Windows设备驱动器一样，通过用于控制资源的清楚类的特殊知识，BIG资源管理器扩展了管理系统。该BIG资源管理器，象Windows的I/O管理器一样，提供一模型，该模型用于将不同的资源统一到一通用保护下。

自动资源管理和最优化

该BIG资源管理器将数据中心操作者从直接包含在资源上的组件的分配和布局中释放出来。例如，当一新的服务被安装到该BIG计算机中时，操作者不需要决定在哪一台计算机上放置该服务。操作者仅需要授予该服务一个资源配额；然后为了节约受限制的共享的资源，例如核心网络带宽，该资源管理器决定如何最佳的在该BIG计算机中放置该服务。

通过该资源管理器，可信资源提供者的基本租参与组件布局优化。通过向资源管理器提供布局选择和特殊提供者相对成本优先权，资源提供者参与布局优化。然后，该资源管理器通过每一个资源提供者的本地业务平衡全局业务，从而最大化效率并最小化资源使用。

最佳组件布局是一正在进行的业务。随着时间的过去，该资源需要单独的服务压缩和增长。有效物理资源改变作为新的装置被添加到该BIG计算机中，而较老的装置被取消。该资源管理器周期的复查布局决定，并估计移动组件的价值。通过向资源管理器提供移动组件的成本，资源提供者参与布局在估计。移动成本可以在用于所存储的不可移动存储器的无穷大和用于无国籍IIS组件的非常小之间。

该BIG资源管理器将操作者从涉及资源分配和组件布局中释放出来。该资源管理器还将开发者从编写复杂分配逻辑的需要中释放出来；相反的，开发者仅仅向资源管理器提供资源需求图。该资源管理器考虑本地和全局两者的资源需求，从而在该BIG计算机中最优化的放置组件。

特征描述

执行环境

BIG资源管理器作为一由高效SQL支持的CLR服务运行。期望每一个BIG机器仅有一单独的资源管理器重复经过在HA SQL群中的一对SQL服务器。

BIG资源提供者在该BIG资源管理器处理中执行。资源管理器给该资源提供

者一同步执行环境从而在该环境中进行操作，并且给该资源提供者一存储了它们的状态的共享数据库。期望所有资源提供者是使用BIG操作逻辑模型的CLR可控制编码。

所有资源提供者保存它们的状态在资源管理器数据库中。因为需要符合它们的管理需要，资源提供者可以创建它们自己的表。

在资源管理器数据库中的资源提供者状态是可信的。这样，例如，IIS元库是一个资源管理数据库中的数据的高速缓存器。如果一IIS VRoot入口在IIS元库中被找到，同时在该资源管理数据库中没有相应入口，那么在元库中的VRoot就被删除。

所有资源分配和再分配请求被合并在事务中。资源提供者，排他的使用该资源管理器数据库来在资源管理器内部排他的执行。即使被集合，交叉提供者资源请求以确定的，非分布的方式执行。这极大的简化了资源提供者的设计和实现，并且确保在故障情况下，资源绝不会在多个服务器中间丢失。

该BIG资源管理器将资源分配和资源初始化分成两个部分，独立动作。资源分配是一非分布的，确定的操作，该资源分配在资源管理器处理中排他的执行。另一方面，资源初始化是一个本质上的分布并且非确定的处理。

资源分配典型的由第一深度操作逻辑阶段开始，并且随着资源请求必然被表征，在该阶段中组件被实例化，通过连接线被连接起来。

通过从资源初始化中分离出组件实例化和资源分配，不管是资源因为它没有结束初始化而无效，还是因为它所位于的设备上恰好是不存在而无效，BIG运行时间和服务可以使用一般的错误处理机制。资源初始化将被一状态机器驱动，该状态机器象资源管理器数据库或者SDM数据库一样，在HA SQL存储器中，保存状态。

资源提供者

该BIG资源管理器拥有在BIG计算机中的所有资源。通过特殊资源资源提供者，资源管理器扩展了多种资源类别的特殊知识。该资源管理器提供存储器，集合资源操作的管理，和作为资源提供者宿主。

通过限定数量的资源提供者，该BIG资源管理器提供一小的，特殊组的资源。当数量很小，期望基本组资源提供者可以覆盖最多的需求，如果不是全部，那么覆盖目标顾客的需求。在第一产品版本中，以下资源提供者是被期待的：

- 物理资源提供者（原始设备）
- 网络资源提供者（VLAN）
- 外部资源提供者（DNS命名，外部IP地址）
- IP负载平衡组资源提供者
- 5 ■ IIS VRoot资源提供者
- SQL DB资源提供者
- CLR AppDomain资源提供者
- Win32表面资源提供者（一Win32程序）

创建模型

10 典型的，资源管理将被操作逻辑驱动，该操作逻辑被打包作为CLR可控制编码运行。操作逻辑将被写入“无实体对象”模型，在该模型中，一CLR可控制对象表示目标组件。该无实体对象负责分配组件所需的任意逻辑或物理资源，初始化那些资源，并且最后当这些组件不再被需要时解构和释放那些资源。

一个调用如同，

15 `FrontEnd f=new FrontEnd(); // 实例化物实体对象
obj ect.`

导致仅仅无实体对象的创建，一CLR类具有一组件实例记录在运行时间数据库中，而没有更多记录。操作逻辑与FrontEnd f相互作用来设置参数，象比例，需求，等等。

20 该无实体对象，在这个例子中的FrontEnd f，通过响应想要的资源图和一随后的资源设置的要求，参与任意资源分配，

`r=f.GetResourceGraph(); // 请求f来生成逻辑资源请求图（如果f是复合的，递归）`

25 `rgo=BigAllocateResources(rgi); // 请求资源管理器来执行全部分配请求
fSetResources(rgo); // 注意时机的资源分配的f（如果f是复合的，递归）`

该无实体对象指导所有对象初始化，象格式化磁盘和放下图像：

`f.BeginConstruct(); // 开始构建/实例化状态机器
machines.`

30 `f.EndConstruct(); // 当构建已经结束时取得结果
(这是.NET异步模式).`

之外，随着无实体对象指向消除，该无实体对象的寿命超过所表示对象的寿命。在先的声明不禁止对象停止。

```
f.BeginDestruct(); // 开始消除状态机器.  
f.EndDestruct(); // 当消除已经结束时取得结果.
```

5 该无实体对象还释放它的资源:

```
f.ReleaseResources();
```

在释放资源之后，该无实体对象可以被删除:

```
f=null;
```

存在少数几个值得注视的东西。因为f仅仅是一个无实体对象，而且因为资源分配与资源初始化/构造是不同的，以下行可以全部被放置在一单独的确定的事务中。它甚至可以是一个被提供给RM DB的非分布的事务，该RM DB是与SDM DB在相同的SQL中:

```
BeginTransaction();  
FrontEnd f=new FrontEnd;  
r=f.GetResourceGraph(); // 请求f来生成逻辑函数请求图  
rgo=BigAllocateResources(rgi); // 请求资源管理器来做全局分配请求  
f.SetResources(rgo); // 注意时机的资源分配的f
```

在某些点上，所有资源提供者将调用分布式操作，但不是在BigAllocateResources()调用过程中。通过自己有的SDM建模服务，给定资源提供者的实现可以调节分布式编码。

布局最优化

首先，在这一论述的上下文中，我想要定义以下表示布局最优化的术语:

I.本地最优化: 通过隐含的忽略其他组件工厂中的布局上的影响，对一个单独的组件工厂单独的最优化。

25 II.集合最优化: 最优化考虑多个组件工厂。例如，最优化认为是IIS应用程序和SQL数据库的布局。

III. 全局最优化: 最优化（包括现有的组件的移动）该实体系统，即在BIG计算机中的所有应用程序。全局最优化与集合最优化不同，因为其具有移动现有的组件的部分。

30 除非误解了人们的定位，我认为每个人都同意以下内容:

5 I. BIG VI应该提供集合分配API。把组件和连接线实例的集合以及SAM中该组件上和连接线实例上的配置参数作为集合分配API的实参。在一个单独的事务中，该集合分配API产生组件工厂来保存必要的资源。[注意，已经明确的使用术语集合替换批，以便突出分配可以包括不同的组件工厂的事实。注意，在该点上没有表述“集合最优化分配API”。]

II. 在长术语中，BIG应该提供全局布局最优化。该全局布局的目的是在BIG机器中重新排列组件实例的布局，以便使必然属性，使用BIG机器的共享资源的主属性最优化。

10 III. 集合布局最优化能够在初始化分配时发生，或者能够在所控制应用程序同意之后采取全局最优化的形式。影响布局最容易的时间是在组件实例被初始的分配时。

IV. 在初始布局之后的组件的移动可能非常昂贵，或者甚至使人望而却步的昂贵。移动一大型的SQL后端可能非常昂贵，并且可能极大的削弱应用程序的有效性。组件的移动应该认为是应用程序的要求。

15 V. 在长时间运行应用程序中，组件移动将是必然的，甚至不需要全局布局最优化。硬件可以非预期的故障。由于通常折旧和生命周期的约束，硬件将明确的退役。这意味着任意长时间运行应用程序将不可避免的需要某种用于移动组件的机构。这些机构是否由全局布局最优化来调节，对于存在实体来说是无关（orthogonal）的。

20 VI. 长时间运行应用程序将支持用于升级的某种形式的移动。例如，用于滚动升级的机构可以由全局布局最优化调节。例如，如果一应用程序的滚动升级策略带来新的前端联机，并且使老的前端（front-end）退役，那么新的前端的分配就是用于最优化它的布局的正确的时段。升级提供全局布局最优化机会的窗口。

25 基于从组中其它内容的回复，我建议以下内容用于BIG VI：

1) BIG VI提供一批分配API。该批API把组件和连接线实例的集合以及SAM中该组件上和连接线实例上的配置参数作为集合分配API的实参。在单独的事务中，该批API产生组件工厂来预定必需的资源。

30 2) BIG VI形式化组件移动。这应该包括标准组件接口，用于获得脱机组件并将其带回到另一个位置。把它看作与Iserialize相同的组件。这一形式化将被操

作逻辑用于执行滚动的升级和实体前端的复制。它还可以被用于分离SQL后端。当消去硬件等等时，它将被使用。我们应该具有可移动组件的概念，以及要移动不同类型的组件意味着什么，怎样估算成本，等等，

3) BIG VI提供一集合布局优化程序。该优化程序的复杂度变的符合开发周期的需要。象不成熟 (crude) 群组的优化那样可以是一简单的优化，或者更高级的优化。

4) 集合布局优化程序由批分配API在初始布局过程中使用。组件工厂与布局优化程序合作来辅助它的设计。

5) 整个应用程序生命周期，集合布局优化程序可以周期的被调用来移动组件实例，从而执行全局布局优化。该优化程序可以调节由一应用程序实际给出的可能的窗口。它还可以要求一应用程序在其他情况下确定组件的移动。基本上，该全局优化仅仅调节集合布局优化程序，以及用于可移动组件的在先存在的支持。

6) BIG VI IIS应用程序组件工厂实现可移动组件，受应用程序允许的控制。很可能大部分全局布局优化的益处可以通过忽略繁忙组件，例如SQL数据库和移动VRoo来识别。IIS还自然的支持例如消耗的操作，其便于VRoo的移动。实际上，IIS VRoo组件工厂变成用于组件移动和全局布局优化的V1广告字段。

物理资源建模

在下面的实体资源管理系统是一硬件资源图。该硬件资源图描述了硬件资源的全体和它们可用的到该资源管理器的连接。该硬件资源图包括服务器，网络设备和网络拓扑。另外该硬件资源图可以包括关于强信号栅极 (power grid) 和物理约束关系的信息。

该硬件资源图由三种基本组件：实体，连接器和连接组成。

一实体是软件可存取硬件的基本单元。实体的例子包括服务器，硬盘驱动器，网络设备等等。

一连接器是到一实体的物理接口。一连接器始终与一个实体精确的联系在一起。连接器的例子包括网络接口，IDE接口，AC电源连接器和物理外壳，等等。

一连接是精确 (exactly) 的两个连接器之间的物理关系。连接的例子包括网络电缆，IDE电缆和AC电缆，等等。

所有三种组件类型，实体，连接器和连接，具有相关的特性。该特性是由特征名，最大值和有效值组成的元组。

所有三种组件类型可以具有对偶。对偶是被用作失败结束的同位体（peer）。一组件和其对偶总是一起分配，以便为高有效性提供必要的冗余。对偶的典型 5 的例子包括在冗余网络中的失败结束转换器，冗余NIC，和连接冗余NIC和冗余转换器的电缆。

所有连接器具有基数，该基数指定每个连接器所允许的最大连接数量。例如，一IDE连接器具有两个基数，一个主机和一个从属设备。参见图62。

定义基本类型的原则：

- 10 ■ 什么是基本硬件协议？
- 在硬件级，设备使用什么语言？
- 基本实体具有一确切的所有者。
- 连接器和连接类别必须匹配。
- 对偶是必须被分配为一个的失败结束对。
- 实体，连接器或连接可以是对偶的。

什么是建模组件？

- 实体
- 连接器 Src=Entity
- 连接 Src=Connector, Dst=Connector

20 什么是基本类别？

- 实体类别：
 - X86 PC：描述软件/CPU/RAM相互作用。CPU和RAM是值。
 - EFI PC：描述软件/CPU/RAM相互作用。CPU和RAM是值。
 - 网络设备。使用IP+SNMP。产品标识符是一值。
- 磁盘。发送和接收扇区。
- 物理容器。
- 连接器/连接类别：
 - 以太网。带宽是值。
 - ATA。带宽和格式是值。
 - SCSI。带宽和格式是值。

- 电源。
- 物理的（容器）。
- 其他: FibreChannel, Serial, Parallel, USB, FireWire, 802, 11, Infiniband。

初始物理配置-参见图63。

5 详细例子-参见图64和65。

基于位置的设备标识符

每一个网络设备具有唯一的在网络中它的位置的标识符。

在每一层上，值=在父类转换器上的端口号。

终止层上具有终止值，“#”。

10 终止值，“#”，大于所有端口号。

例如，参见图66。

计算两个设备之间的路径

设想两个设备 (2, 0, 1) 和 (2, 1, #)

为每一个设备，计算所终止的前缀:

15 (2,0,1) → (#, #, #), (2, #, #), (2,0,#)

(2, 1, #) → (#, #, #), (2, #, #)

大部分特殊的通用终止前缀是通用父母的:

(2, #, #)

剩余终止前缀是中间转换器名字:

20 (2,0, 1) →(2,0,#)

(2, 1, #) →none.

最后路径:

(2,0,1) to (2,1, #) →(2,0,#), (2, #, #) =2个交换转发 =

3个连接线转发

25 通常还找到设备的最接近的同位体:

(2,0, 1) →(2,0,?)

(2,1, #)→(2,?, #)

参见图67。

资源请求建模

30 该BIG资源管理器模拟BIG机器为节点（资源）和边（相关性）的图。节点

和边两者可以用属性（名值对）来注释。

依靠该资源图的，最通用查询的类型之一是子图同构。客户创建一请求图，并要求资源管理器在具有相同形态和所有权的硬件资源图中找到一子图。该资源管理器找到一匹配的子图就返回一完整带注释的回复图。

5 由于部分的子图同构，该资源管理器必须不是重叠或者组合的图节点。即，如果该请求图包括两个PC节点，那么回复图就必须包括两个PC的唯一节点。

请求图可以包括搜索参数，例如找到一PC节点或找到一至少具有256MB的RAM的PC节点。回复图包括每一个匹配元素的特殊识别号（节点和边两者）。

10 在基础情况下，请求图是只读查询。然而，通用最优化考虑在资源分配形式的读写操作。当在纸上绘制时，写操作被用括号标注。

图68是一分配了一PC和一附加磁盘的请求图，该附加磁盘通过存储器通道例如IDE或SCSI连接。注意，节点被表示为圆角矩形，而边被表示为黑线带有重叠矩形，其中属性被指定。成功的分配可以导致图69的回复图。

驱动情况

15 Joe的花店产生如图70所示请求图。MSN确保Joe至少获得一个500MHz PC，因为他具有“金”SLA，并且确保他的PC是连入Switch5从而保持位置的。通过如图71所示的添加，成群的退出保证MSN总是得到在Rack17中的机器，并且还获得小磁盘，因为它具有一“第2”类存储器SLA。参见图72。

实现思想

```
20 class Graph;
class Client
{
    private IResourceMediator mediators [];
    private Object mediatorStates [];
    }
    interface IResourceMediator
    {
        public void MediateRequest(ref Graph graph, ref Object state);
        public void MediateReply(ref Graph graph, ref Object state);
    }
30 }
```

```

class ResourceManager
{
    public Graph Allocate(Graph request, Client client)
    {
        5   for (int n=0: n<client. mediators. Length; n++)
        {
            client.mediators[n].MediateRequest(ref request,
            ref client. mediatorStates[n]);
        }
        10  Graph reply=PrimitiveAllocate(request);

        for (int n=client.mediators.Length-1; n>=0; n--)
        {
            15  client.mediators[n].MediateReply(ref reply,
            ref client.mediatorStates[n]);
        }
        return reply;
    }

    20  private Graph PrimitiveAllocate(Graph request);
}

```

基本资源分配情况

这一部分列出多个情况。每种情况包括的是相应的请求图。作为查询事务的结果所分配的节点被标注为 “[Allocate]”。不会被分配的，并且对于匹配搜索必须是未分配的节点被标注为 “[Free]”。没有标注括号的节点不被分配，相反的，它们为其他请求图提供上下文。

PC

Akamai需要在Digix数据中心里分配一至少具有1GHz CPU, 512MB RAM 和100GB本地磁盘存储器的服务器。参见图73。

VLAN:

30 MSN实例信息决定实现—包括它的前端的DMZ。为了这么做，需要覆盖其

前端的2 VLAN。参见图74。

公开的IP地址或DNS名

Joe的WebService需要使其自身对外部环境可见。他需要分配一DNS入口和一可路由的IP地址。参见图75。

5 负载平衡组

Joe的WebService对于单个PC来说已增长的太大了。他需要分配一负载平衡组和另外一个PC。接着他需要将在该负载平衡组虚拟IP地址之后放置两个PC。参见图76。

路径

10 Hotmail 需要分配一80Mbps路径来从一个UStore到另一个传输邮件账号。Hotmail还可以在路径上指定等待时间和QOS需求。参见图77。

特殊存储器

Hotmail想要创建一新的UStore。它想要一Raid 1邮箱，该邮箱具有100GB，遍布至少4组以10000RPM速或更快速的旋转的非共享磁头。参见图78。

15 群（组）存储器

Hotmail想要分配一对用于覆盖故障群的带有共享硬盘的机器。它想要一Raid 1邮箱，该邮箱具有100GB，遍布至少4组以10000RPM速或更快速的旋转的非共享磁头。参见图79。

共享存储器

20 Joe的WebService需要多个机器可用的50GB通用存储器，以便返回保存服务特殊配置的图像。该存储器对0到N个机器有效。参见图80。

分布布局情况

临近机器分配

Hotmail需要分配一个新的前端。它想要在与它的其他前端相同的转换器上找到一具有足够带宽到后端群的机器。参见图81。

远距离机器分配

该Expedia用户头文件数据库需要用于SQL复制的另外一个机器。想要一机器，该机器被定位在由不同的电池备份单元覆盖的数据中心的一部分里。参见图82。或者可能如图83 的例子。

30 等待时间驱动

该Hotmail后端需要分配一机器用于群合作。该机器必须在已经在群中的机器5ms等待时间以内，但带宽很低。可选的，这可能由在一个网络中继段里需要的机器来表示。参见图84。

种子复合组件

5 Hotmail要创建一新的邮件单元。该单元必须被分配在一单独中继段群中，具有增长到至少500PC的空间，即使Hotmail可能只初始分配了一少数的几个机器。参见图85。

批分配

10 MSN搜索决定添加为所基于的小的音乐采样搜索MP3的功能。想要分配一400PC，3负载平衡，和20TB存储器的块。想要一所有或没有的分配。参见图86。

撤销 (revocation) 情况

恢复

Joe的WebService已经停止支付IDC。该IDC需要重新获得所有分配给Joe的WebService的资源，并将它们返回给有效资源池。

15 硬件生命周期撤销

Expedia前端之一是一个已达到其生命周期的结尾的PC。由IDC操作逻辑所触发，资源管理器通知Expedia在该机器被返回IDC之前它还有72小时。

被控制撤销

20 Hotmail分配20个短期机器，用于一大块的它的UStore的重新安排。依照它的SLA，现在该IDC要求一个要被返回的机器。Hotmail可能返回二十个机器中的一个，或者其他相当的机器。

BIG Vision-能够:

■ 分布式，可缩放和高效服务的开发，使用Visual Studio和可重用的构造块，象SQL，IIS，.....

25 ■ 部署覆盖一组抽象硬件和软件资源，该硬件和软件资源可以被自动分配，决定以及配置。

■ 经过自动化，通过编码可操作最好的执行过程来控制服务的有效性和增长，降低了所有权成本。

■ 获得标准化数据中心硬件，其调节了产品经济。

30 BIG服务平台架构-参见图87。

BIG计算机-硬件引用平台

减少了设计测试和操作的成本:

- 限制支持的硬件设备数量
- 约束网络拓扑
- 能够网络配置的自动化

5

消除用户考虑BIG技术部署的需求。

- PXE, DHCP, DNS, VLAN

IP通路

- 调整在外部网络和内部网络之间的IP通信量
- 网络地址转换 (NAT), 防火墙, 负载平衡

10

Internet网络

- IP地址和VLAN由BIG排他的管理
- VLAN被自动配置

硬件构造块

15

■ 批量 (commodity) 服务器, 网络转换器和硬盘的组合

参见图88。

图89说明当前产品的例子, 该产品能够位于BIG计算机内部。

资源管理特征

- 动态发现服务器, 存储器或网络硬件资源。
- 高效数据库, 包括 (物理和逻辑) 资源。
- 支持枚举, 查询和更新资源的运行时间API。
- 逻辑资源驱动模型和API, 用于绑定资源驱动器和物理硬件设备。
- 服务器资源的计划分配和再分配。
- 自动配置并管理网络资源, 例如VLAN和负载平衡组。
- 动态配置并管理块和基于文件的存储器的资源。
- 故障检测监视和通知。

25

资源管理组件

- 资源管理器负责在BIG计算机内部的硬件和软件资源的分配
- 资源管理器向BIG运行时间注册
- 资源管理器实质上是一用于给出的资源类型的工厂

30

- 硬件资源管理器
 - 基层工厂负责分配硬件实例
 - 网络资源管理器
 - 负责分配VLAN, 负载平衡组, IP地址.....
- 5 ● 存储资源管理器
- 管理存储器资源, 例如磁盘和文件
- PC资源管理器
- 分配目标服务器, 并使用iBIG服务部署OS
- 软件资源管理器
- 分配并配置IIS vroot, SQL数据库, ASP.NET,
- 10 图90说明多种资源管理组件。
- 硬件资源发现和管理
- 器件: 电源, 网络, 内部存储器, 处理器, 存储器, 定位
- 位于BIG计算机内部的硬件被自动发现。资源驱动器被绑定到硬件设备上,
- 15 并且公开逻辑资源到硬件资源管理器 (HRM)。HRM将逻辑资源分配请求转换成物理资源绑定。参见图63, 64, 65。
- 在BIG计算机中的网络资源管理
- BIG计算机为网络资源定义一抽象层。
- 网络资源管理器: 分配网络资源和程序给在BIG计算机内部的网络转换器
- 20 和负载平衡器, 以及带有网络资源驱动器的接口。
- VLAN提供在BIG计算机内部网络分离和分区。
- 网络资源实例: VLAN, 负载平衡组, 网络适配器, IP地址, DNS名字。
- BIG存储器资源管理需要
- 连接BIG计算机的存储器的全局视图, 该BIG计算机拥有文件和基于块的
- 25 存储资源。
- 虚拟化存储器互连结构。
 - 框架, 用于创建和管理高层存储器抽象, 例如LUN, 容量, 数组, 等等。
 - 一驱动器/提供者模型, 允许现有的和新的存储器设备连入BIG计算机。
 - 与SAN系统的配合操作性。
- 30 基础结构服务 (自动部署服务 (ADS)) - 特征

- 基础部署服务
 - 基础网络登陆服务（PXE）和图像建造服务
 - 预先登陆OS环境（BMonitor）
 - 为传统工具支持虚拟软交付网络
- 5 ● 图像部署和管理
 - 工具，用于创建，编辑和删除图像
 - 关于运行预先OS系统的图像部署
- 10 ● 复合设备管理（MDM）
 - 脚本，用于通用任务
 - 用于部署的并列的步骤和处理的任务序列
 - 完全程序化接口（WMI）
- 15 ● 从.NET服务器RTM传送60天
 - 支持Windows2000和.NET服务器目标

图92说明一个ADS架构的例子。
- 15 图93说明一个ADS远程登陆和图像系统的例子。

服务定义模型（SDM）

 - 整个服务的程序化描述
 - 服务的说明性定义
 - 以比例不变的方式定义服务的整个服务结构
 - 提供用于部署，管理和操作的框架
 - 20 ■ 以模块化方式，基于组件模型获得服务组件
 - SDML是说明语言，用于定义服务定义模型
 - 组件，端口和连接线
 - 类型，成员和实例空间
 - 支持合成和封装
- 25

SDM：组件，端口和连接线

 - 组件是实现，部署和操作的单元
 - 例如，专用服务器运行.NET服务器，IIS虚拟网站，SQL数据库
 - 通过端口公开功能，并通过连接线通信
 - 通过合成创建复合组件
- 30

- 端口是具有相关类型（协议）的名字（服务访问点）。
 - BIG不指示什么协议用于通信
 - 协议获得为建立通信所要求的信息
 - 连接线是端口之间可允许的绑定
 - 连接线说明端口之间的拓扑关系
- 5 参见图94。
- 图95说明SDML的例子： MyService.sdml。图28也涉及这个SDML例子。
服务部署单元（SDU）-封装所有组成服务的片断，包括：用于应用程序和
10 服务的SDM模型，用于组件实现的CLR组装，以及MSI， ASP.NET， SQL脚本，
静态内容，等等。参见图96。

- 15 SDM运行时间
- SDM运行时间负责跟踪SDM模型和实例
 - 作为由IIS容纳的网络服务来实现
 - 能够被分区用于可缩放性
 - 运行时间API公开SOAP终点
 - 通过一运行时间库进行与运行时间的通信
 - 高有效性的SDM存储（使用YuKon冗余数据库技术）
 - 两个SQL服务器和记录服务器

20 参见图27。

例子：组件示例

```
using Microsoft.SDM;
public class MyService:
    SDMComponent
{
    public OnCreate(...) {
        fe1=CreateInstance("fe","");
        be1=CreateInstance("be","");
        w1=CreateWireInstance("tds");
        w1 . Members.Add(fe1 . Ports ["catalog"]);
        w1.Members.Add(be1 . Ports ["sql"]);
    }
}
```

```
    }  
}
```

myservice. cs 是使用SDM API的C#代码。

```
5   componenttype MyService  
    {  
        component MyFrontEnd fe;  
        component MyBackEnd be;  
        port http=fe. http:  
10       wire TDS tds {  
            fe . catalog;  
            be. sql:  
        }  
        implementation "MyService, MyCLRApp"  
15    }
```

参见图35。

使用SDM运行时间API动态绑定的例子（参见图97）

1.be[1]说明sql[1]端口准备好，并使用DeclarePort()注册它的端口与SDM运行时间的连接信息

2.fe[1]初始化并向SDM运行时间要求用于 catalog[1]端口的同位体信息，并且使用GetPeerPort()接收关于sql[1]端口的信息

3.然后，fe[1]使用由SDM运行时间动态提供的端口连接信息连接be[1]

服务定义模型（SDM）工作组

● SDM工作组包括5组：

■ Indigo

■ Whitehorse

■ 合成

■ 管理

■ BIG

● 特权是用来定义用于分布式和/或离散应用程序的类级应用程序模式的。

- 使用组件，端口和连接线描述应用程序
- 包括部署，配置和管理信息
- SDM是一外部框架，该框架引入合成和管理（并可能是其他）模式
- 合成组装负责部署（在适当的布局）
- 5 ■ MBU设置和装置模式被引用并被指定用于配置和监视

SDM模式（示例的）

SDM Schema (simplified)

<sdm>

<identity> //识别定义的组

10 <porttypes> //端口的描述

<wiretypes> //拓扑的描述

<componenttypes> //在该库中被定义的组件

<componenttype>

<ports> //通信能力

15 <settings> // 为组件配置设置

<instrumentation> // 监视模式 <deployment> // 安

装程序类型，安装程序信息（例如Fusion）

<components> //

用于组合的子组件

<wires> // 定义端口之间的关系

20 </componentType>

</componenttypes>

</sdm>

25 SDM和合成-参见图98。

- 带有默认值的本地设置是在合成清单中被指定的（或其他本地安装技术）
- 在SDM中的设置由Ops逻辑和BIG运行时间处理
- 例子：“用户数”将被用来确定应用程序的初始比例放大条件

30 SDM和部署-参见图99。

以比例不变的方式描述应用程序的结构需要应用程序宿主环境的相似的比例不变的描述（即，数据中心），以便使部署请求和约束能够设计时间有效。

■ 微软和用户花费大量精力绘图精心描述它们的数据中心环境，并编写非常大型的文档来解释该图。

5 ■ 这些图和文档将许多自物理机器名，到IP地址，到VLAN，到服务器角色的信息层，合并为一个经常是很混乱的综合视图。

图100说明了一个示例系统架构。

图101说明了一个多种部署层的例子。

操作逻辑是操作的“商业逻辑”。

10 操作逻辑是CLR编码，该编码获得可重复形式编码作为可重用的最优方法

- 不指定服务或者操作环境
- 可以被开发，测试和发送
- 减少需要人来执行的人工过程的需要

Ops逻辑负责服务的整体操作

15 ■ 开始服务

- 服务增长和消减
- 升级和更新
- 故障校验和恢复
- 数据库分区

20 Ops逻辑将使用MS中间层技术来实现

- 容纳在IIS上的ASP.NET网络服务
- 用于配合转换的DTC
- 用于存储的SQL服务器
- ，用于监视和管理的WMI
- 用于发消息的MSMQ

可重复升级模式-> 操作逻辑

- 升级是可重用操作逻辑模板类型的一个例子，是我们想要用BIG输出的。
- 原地升级模式
- 移动数据的成本是高的，编码实例化的成本是低的，或者不共享资源
- 从服务中取出组件，运行更新，将该组件放回服务中

● 端对端升级模式

- 移动数据的成本是低的，编码实例化的成本是高的，具有共享资源
- 创建新组件；移出老组件；移动数据到新组件，将新组件放入到服务中

● 替换升级模式

5

- 没有数据移动

- 添加新组件，移走旧组件，协调来管理服务

● 滚动升级是一个高层操作逻辑的例子，可以重用该编码升级模式的

- 操作逻辑可以被测试，并且框架支持重新运行

- 通过让软件来执行该步骤，从执行中消除人为错误

10

操作逻辑，BIG和该微软程序模型-参见图102。

该互联网转换企业应用程序-所增加公开导致增加成本。参见图103。新的架构已经导致由HW，人们驱动的增加成本，并导致因为复杂性灵活性的降低。参见图104。摩尔律扩展到DC-显著增加磁盘密度，NW通过量和处理功率。

15

服务交付是人们强调的-人为包含冲击安全性，可靠性，适应性和成本。参见图105。

这是生命周期的问题-用户苦恼遍及开发，部署和操作阶段。参见图106。应用程序不被开发具有：以想象约束的对HW配置的缩放，想象上的可管理性，想象上的操作-在我们的数据中心里需要什么？测试-“放弃该墙”。开发者平台-> 测试配置？这如何映射到我的产品环境中。部署询问：使用哪一个服务器？什么是正确的拓扑？是否检测了服务器，存储器和网络组？需要预料多少未来的要求？操作质疑：怎么处理这些警告？那一故障网卡将如何影响我的应用程序？为什么服务性能下降？我希望复制我的邮件管理。

介绍服务交付询问-用户可行方案的核心。

在生命周期每一步的独立值

25

配置，部署，操作

合并全部生命周期的架构

提高步骤之间的协调和回馈

能够映射到改变的事物需求

映射仅仅能够进行，一旦你有灵活性

30

在最低TCO平台构造

经过按比例放大，有效调节工业标准硬件

设计summit-创新的服务交付架构。参见图106。开发服务：是可测量的，可管理的，包括配置需要，封装操作知识，以及调节标准构造块。易于部署服务：快速供应，DC资源虚拟化，自包含，一键部署，从测试到生产的一致性，
5 以及不依赖于比例。简化操作：集合管理，监视和改变管理，管理服务而不是应用程序，通过上下文精确自动化，丰富核心服务管理控制台。

映射事物需要IT系统。在工具中获得IT操作知识。

设计summit-一个综合的新的架构和广泛工业初始化。参见图107。

原理-> 架构-> 产品

10 一长时间，用户和对方驱动结果。

一主要花费开始于1999年

首先深度搜索大型MS互联网操作需要

特性

确认初始查找遍及完全的用户基本成分

15 在2000年后期，来自产品组的原型

连接开发对象的强有力设置

大型企业和服务提供者用户被包括在产品定义中

IHV和ISV对象协商帮助定义通过API所公开的功能性

初始产品用Windows Server 2003发送

20 用户转换复杂的系统为简单的图表。参见图108。

谁被包含在交付你的IT服务中？ -人是该系统完整的部分。

应用程序架构设计者-设计服务

网络架构者-配置网络

存储器设计者-维护远程存储器

25 应用程序操作者-维护服务

网络操作者-维护网络

存储器操作者-维护远程存储器

服务器操作者-维护服务器

这一模型的问题：许多人为相互作用，没有通用语言，区域知识模糊。

30

方案的详细描述:

服务定义模块

资源虚拟化

操作自动化

5 管理API和解决方案

驱动工业广泛的初始化

该服务定义模块 (SDM) -获得完全的服务。

服务的综合描述

应用程序组件和设备

10 服务拓扑

基础资源 (服务器, 存储器, 网络)

关于开发者和操作者

层和单独的责任

提供引用的相容框架

15 用于开发者以 Visual Studio 公开

在运行时间用于操作者的一活动的模型

所分配资源的逻辑一致ind.

实时跟踪资源

在服务组成上的单独授权

20 提供用于精确自动化的上下文

SDM术语

组件-服务构造块。

逻辑构造

比例不变

25 一个组件可以具有多个实例

单独或复合

单独的逻辑实体 (数据库, 网络服务, 文件分区)

组合逻辑实体 (HA数据库, 电子邮件, 等等.....)

包括清楚的对一组件指定的部署

30 DB组件包括数据库模式

- 网络服务组件包括URL目录，内容，编码
端口和连接线的互连
端口-服务访问点
连接线-端口之间的通信相关性
5 SDM提供用于抽象和封装的方法。参见图110。
能够重用
结构复杂性
将人映射到SDM-提供引用的相容框架。参见图111。
开发一个SDM应用程序-一个新的Visual Studio 设计表面。参见图112。传
10 应用程序，新的应用程序。
在数据中心的一SDM应用程序-用活动的模型跟踪资源的综合描述。参见图
113。
什么是summit计算机?
一灵活的虚拟硬件资源池
15 服务器，存储器，网络设备，管理结构。
几打到几千个服务器
由现有的HW所组装的或者由OEM作为一个SKU所安排好的
一单独的可控制实体
summit供应和管理所有HW资源w/insummit计算机
summit拥有完整的互联网络结构的配置。
20 有界限的控制域
标准化拓扑界限建造，测试和操作的复杂性。
用于该Summit计算机外部的资源所有权未改变。
用于软件革新的催化
25 问:我应该把什么样数据中心环境作为我的服务器应用程序的目标?
答:Summit计算机。
就象Win32让ISV忽略打印机和图形卡的细节一样。
用于硬件革新的催化
微软与主要硬件供应商协商定义一引用平台。
30 首先规格&革新出现在WinHEC中(2003年5月)。

Summit提供SW环境用于集合革新:

密集叶片 (dense blade), 智能支架(smart rack), 等等。

Summit能够简化硬件, 例如允许:

撤销来自服务器的KVM和来自网络设备的人为接口。

5 图114说明了示例资源管理器。

资源虚拟化-SDM和组件实例之间的桥接。负责共享, 分配和恢复。参见图
115。

服务器资源虚拟化-在Windows Server 2003中自动部署服务 (ADS)。

基础结构, 用于快速决定和重新决定Windows Server。

10 图像工具, 来获得并编辑Windows 2000和Windows Server 2003两者的图像
安全的, 远程部署框架能够零接触服务器

由非金属制造的构件

用于块服务器管理的框架

安全的, 可靠的脚本执行基础结构

15 你的Windows 数据中心的程序化模型

所有管理活动的持久日志

图形和程序化接口

用于基于GUI操作的简单的MMC UI

通过命令行工具和/或WMI层公开完全的功能性

20 ADS的关键优点

1.降低关于非金属服务器构件和基于脚本管理的TCO

能够零接触由非金属制造的服务器构件

基于象一个服务器管理一样简单的1000个服务器管理的安全脚本

2.改进稳定性, 安全性和可缩放性, Windows服务器数据中心的

25 编码可操作最优方法, 并消除人为错误

包括一所有管理活动的持久存储器

在全部Windows数据中心执行安全, 基于脚本的实体的管理

响应工作量请求的改变, 迅速改变服务器角色

3.调节现有的服务器管理投入

30 扩展并增强现有的基于脚本的自动化方法

- 可操作自动化-自动化核心原则
- 复杂框架，能够获得并重用可操作最优方法
- 操作逻辑
- 丰富上下文，在该上下文中可以自动操作
- 5 事件由SDM置入上下文中，从而能够精确自动化系统管理
 - “哪个应用程序将受在第rack 22中第5DL380上死去的NIC的影响？”
 - 能够执行（transact-able）
 - 基于模型的补偿允许恢复和取消
 - 操作逻辑-一个用于开发者和操作者自动化的框架
- 10 什么是操作逻辑？
 - 所编码操作处理是长生命周期，高有效性的和持久的
 - 调节用于上下文的SDM和summit计算机资源的控制
 - 使操作者能够改变在系统中的自动化水平
 - 适合于开发者的优点
- 15 允许开发者获得系统应该如何响应，并且
 - 决定应用程序事件和消息（比如返回代码）
 - 使微软和ISV集合体能够提供标准，
 - 所预先定义的操作上的处理，该处理是开发者可以使用，或扩展
 - 部署，升级，按比例放大和移动资源
- 20 适合于ITPro或者操作者的优点
 - 能够容易的重用所检验的用于数据中心的操作上的最佳方法
 - 操作上的自动化-编程操作逻辑。参见图116。
- SDM如何与操作逻辑的相互作用：
- 事件被注释，从而指示实例和组件信息
- 25 监视子系统产生基于时间事件的相互关联
- 警告是事件的出现
- 命令允许是：
- 由组件公开的管理命令组
- 是自描述的
- 30 能够被直接用在shell中

能够具有GUI格式说明

能够为操作者的使用提供一“人工页面”

参见图117。

操作上的自动化-能够执行 (transact-able)

5 事务处理对于支持容错操作是必要的

例子：添加一个网络服务

有力的扩展ad-hoc shell脚本

依靠事务模型的支持的所有操作逻辑功能的格式

基于补偿

10 持久的

使用配合，事务处理能够跨越多个机器

管理API和解决情况-调节SDM的丰富程度 (richness)

通过SDM可视化出现

第三组织控制台能够直接从微软管理解决方案的SDM或调节平台的专门

15 知识拖动信息

微软将为数据中心构造一基于SDM的管理控制台

用户能够创建通过SDM用户控制台

参见图118。

工业广泛的初始化-发动IHV, ISV, SI革新

20 IHV HW引用平台指定

紧密的与主OEM和转换器厂商合作

在WinHEC (五月03) 触发释放

在未来的HW商品中推行新的强制特征

保留关键第三组织ISV

25 为Visual Studio创建应用程序组件

资源管理器，用于在SDM中它们的应用程序

Visual Studio ISV创建基于管理解决方案的SDM

作为用户和合作伙伴与SI合作

用户

30 动态的降低它们的操作上的成本

合作伙伴

在这一平台上创建革新的新的服务产品

将操作经验转化为资本-）开发操作逻辑

主要用户优点：提供机会并为数据中心创建最经济，可管理的平台。

5 工业广泛的初始化-扩展SDM的丰富度到异类环境。使用Visual Studio（能够为微软开发SDM应用程序）或第三组织工具（能够为其他平台开发SDM应用程序）开发异类SDM应用程序。

总结

10 虽然本发明已经用语言对特别的结构特征和/或方法动作做了描述，但可以理解的是，在示范性附加权利要求中所定义的本发明并不限定为所描述的该特殊特征或动作。较合理的，该特殊特征或动作被公开为实现所要求发明的示范性形式。而且，依靠范围和主题这些权利要求是示范性的。在本申请之前的母申请请求时间里，许多其他在这里所描述的特征的合并和合并子集可以在之后被要求。

15 虽然上面的描述使用了对特殊结构特征和/或方法动作的语言，但可以理解的是，在附加权利要求中所定义的本发明并不限定为所描述的该特殊特征或动作。较合理的，该特殊特征或动作被公开为实现所要求发明的示范性形式。

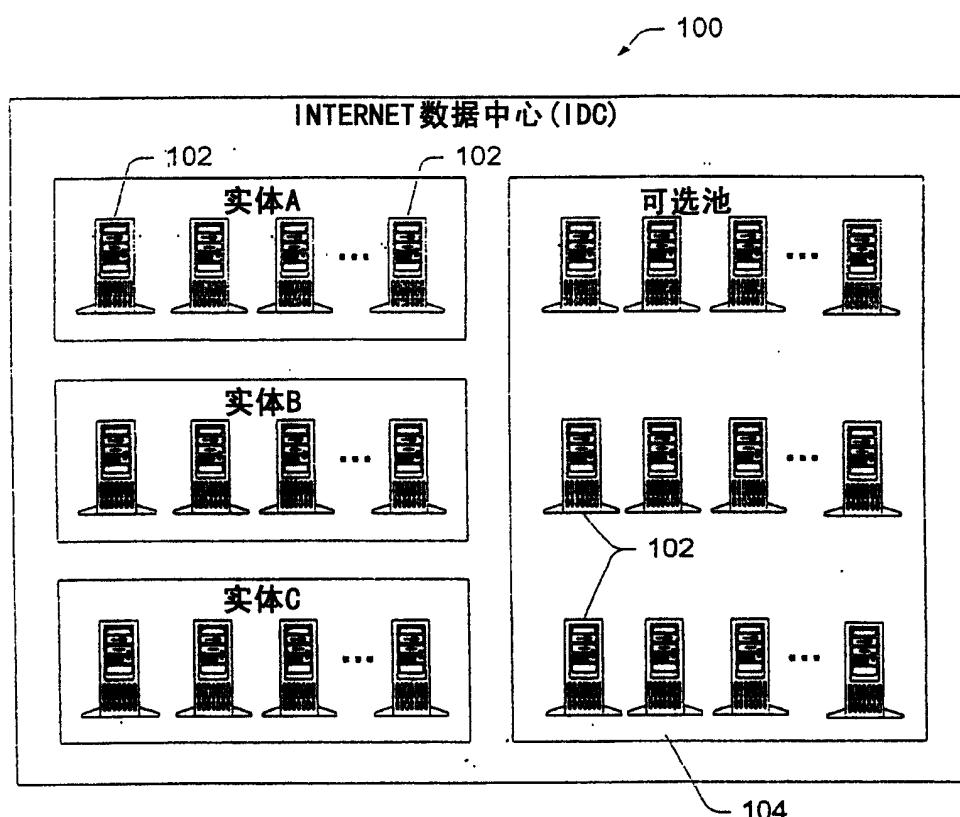


图 1

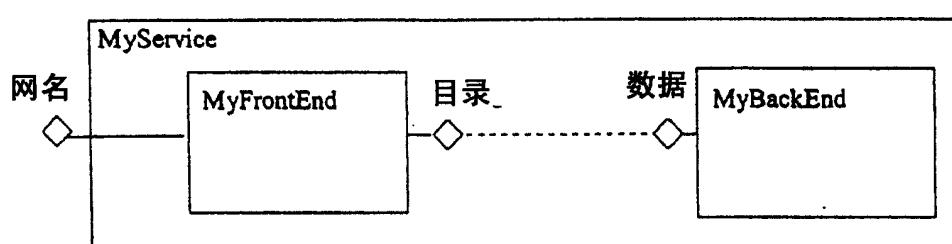


图 2

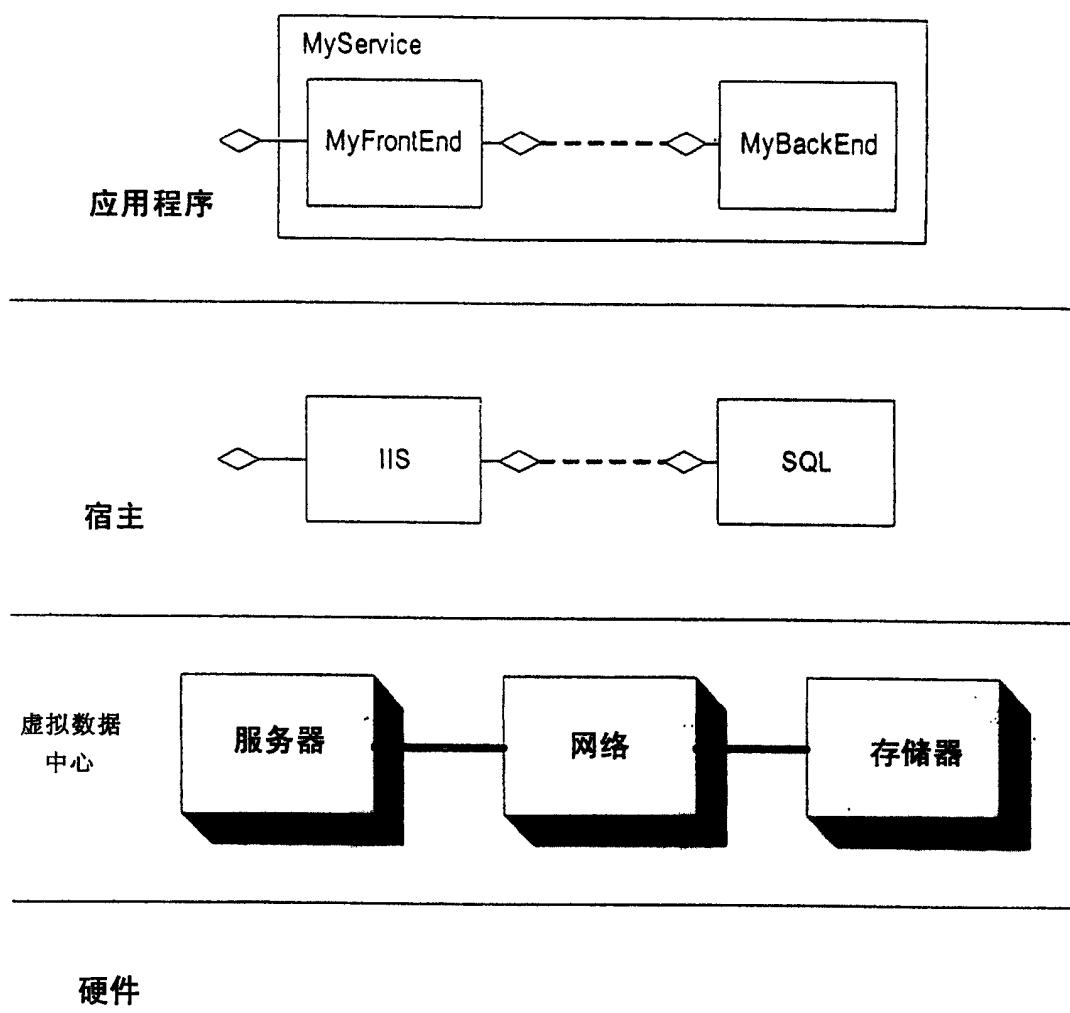


图 3

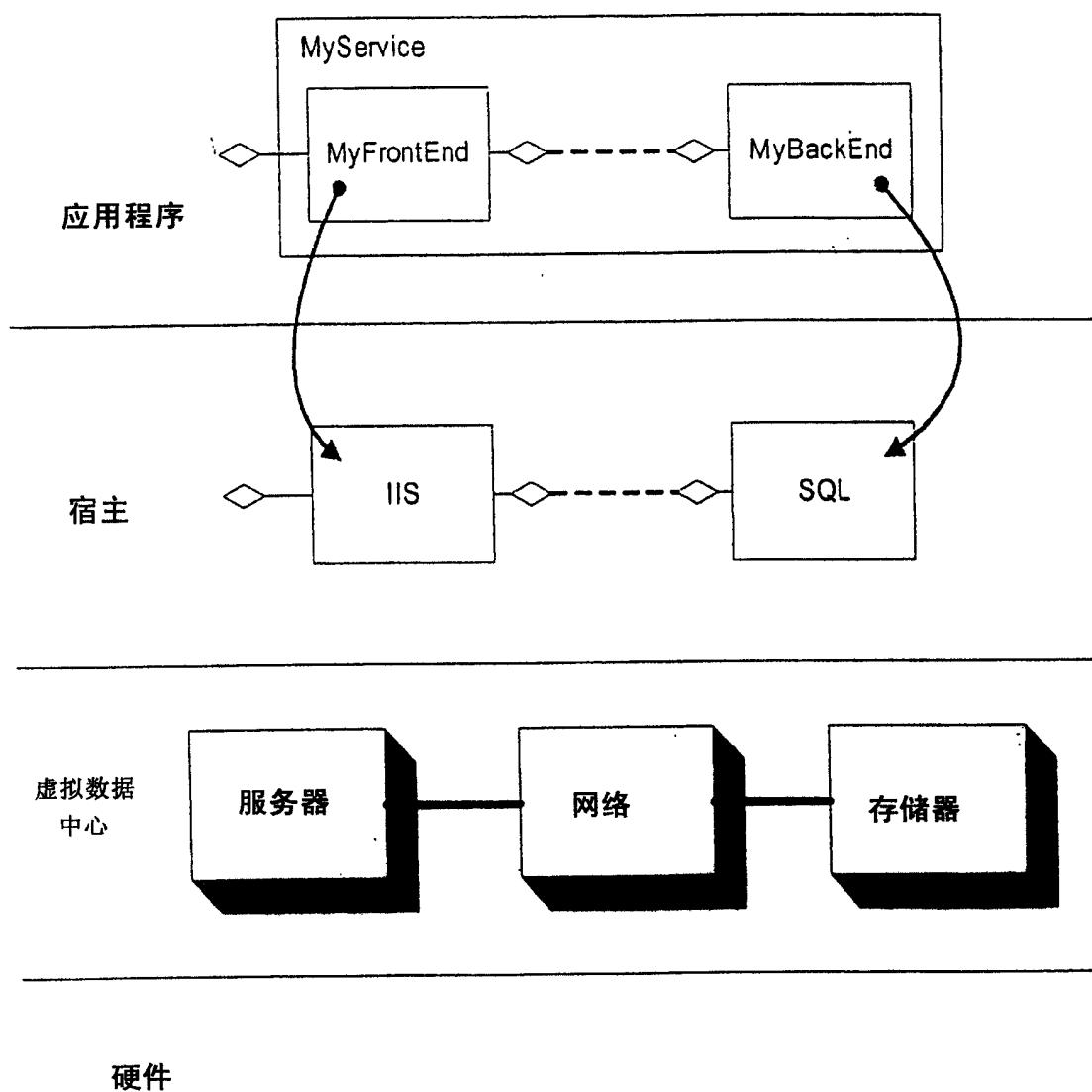


图 4

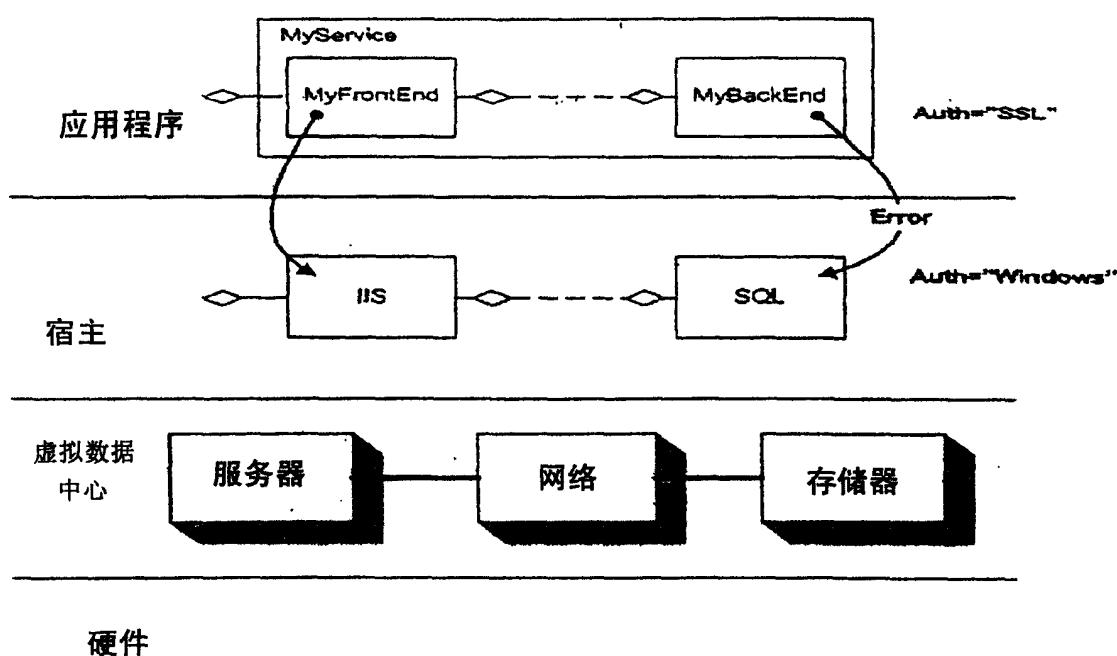


图 5

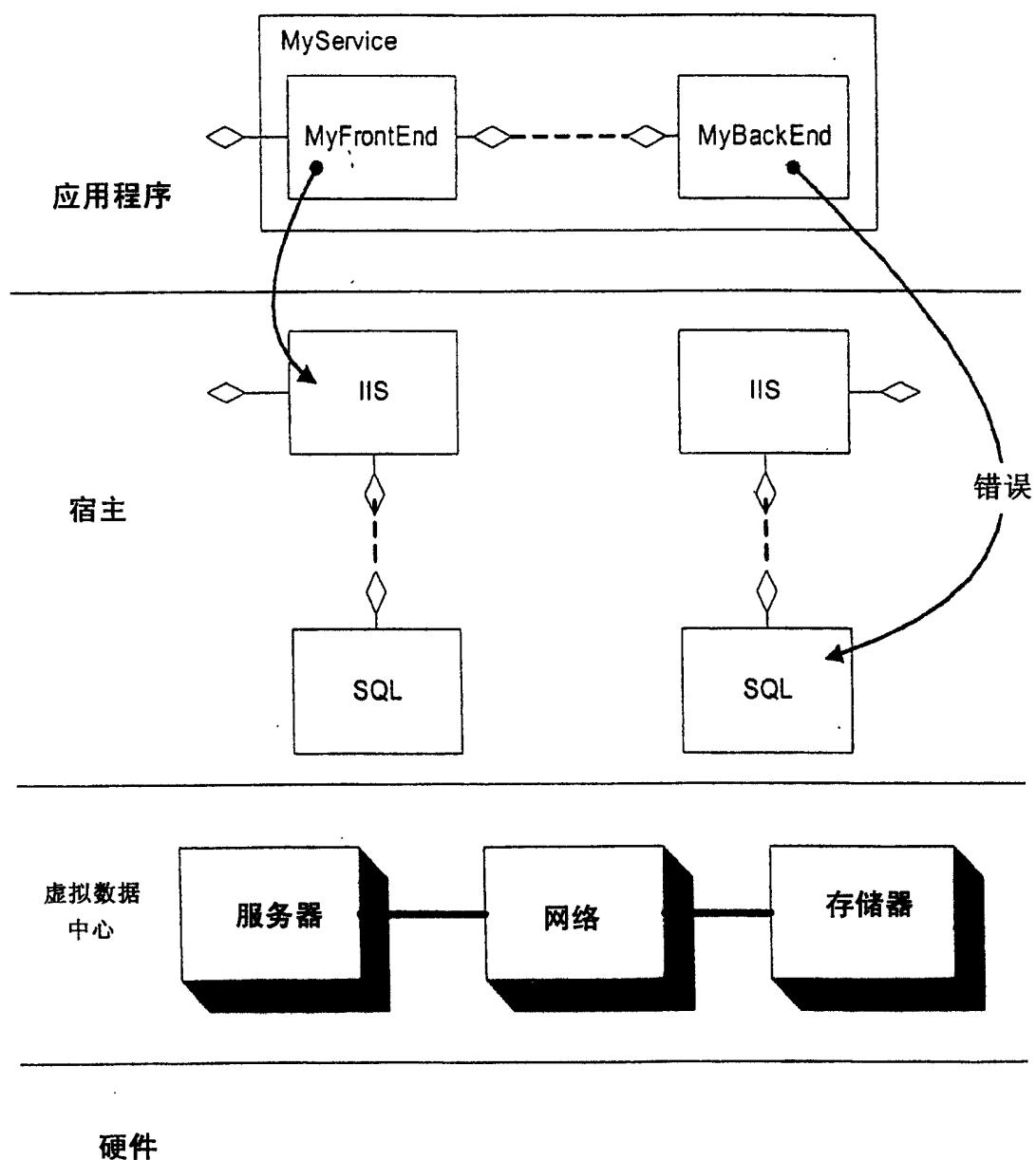


图 6

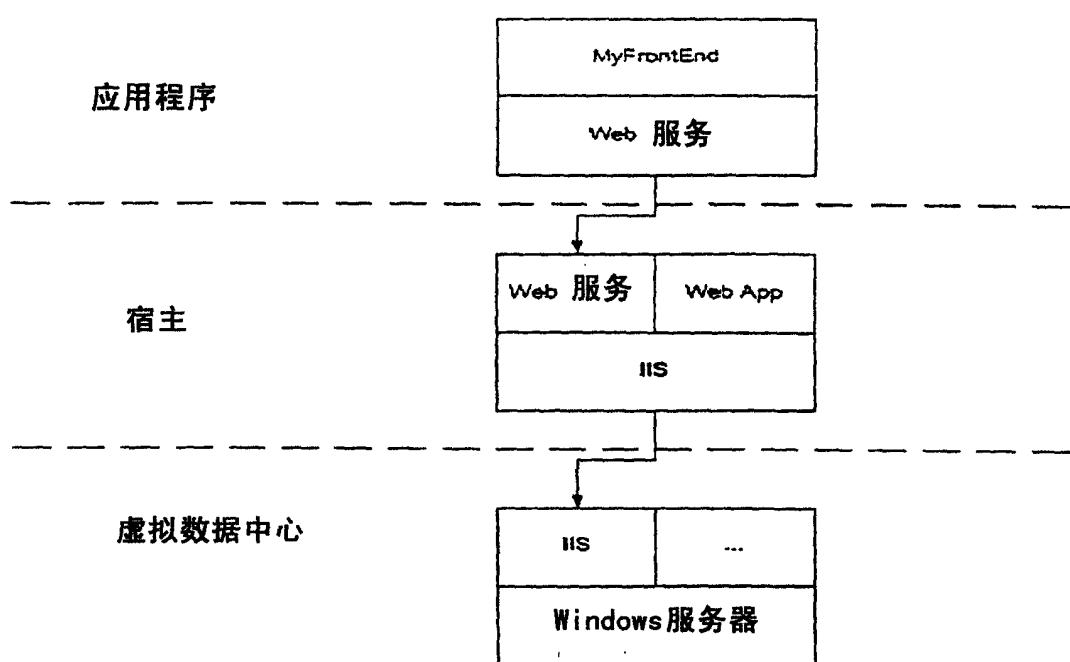


图 7

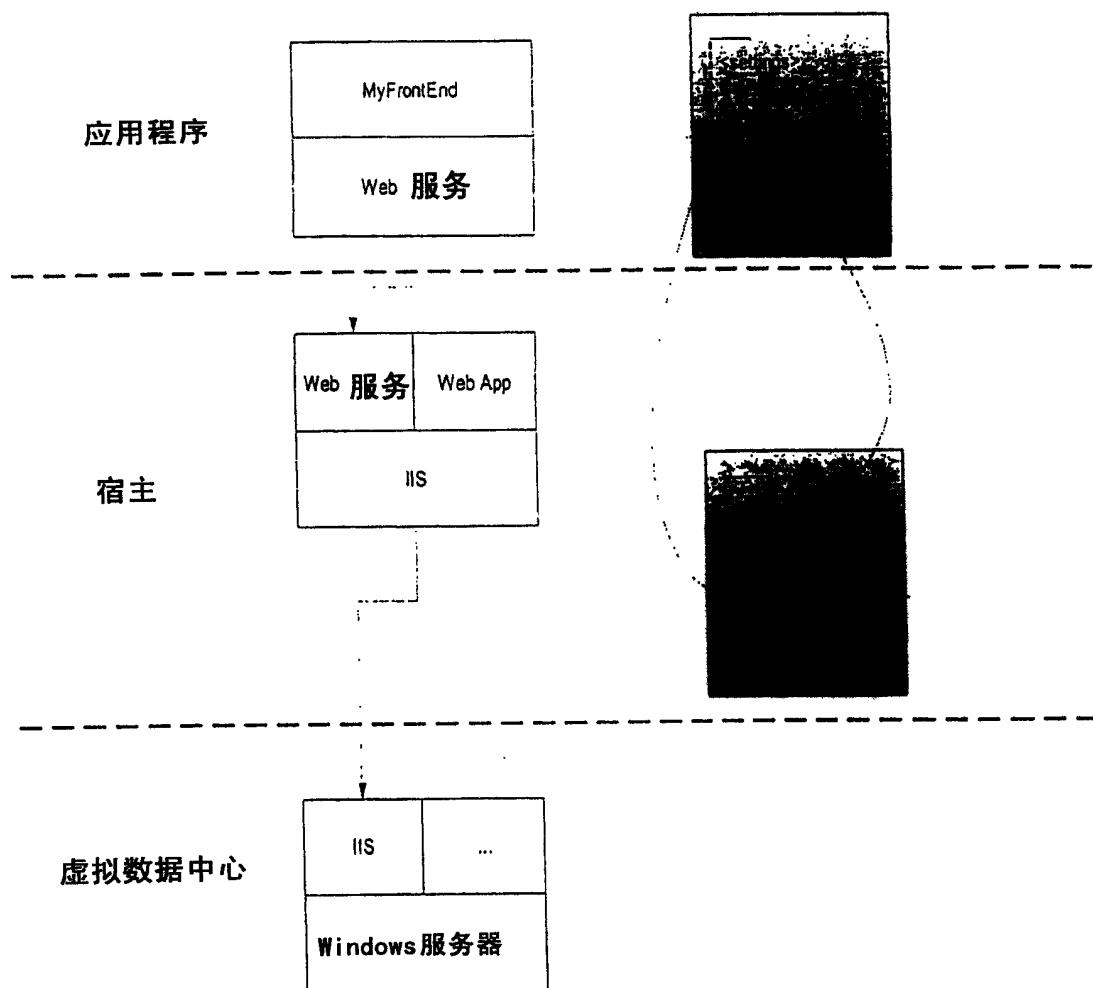


图 8

设置模式	部署清单	端口实现参考
------	------	--------

图 9

设置模式	部署值	约束值	端口类型或 容纳类型列表
------	-----	-----	-----------------

图 10

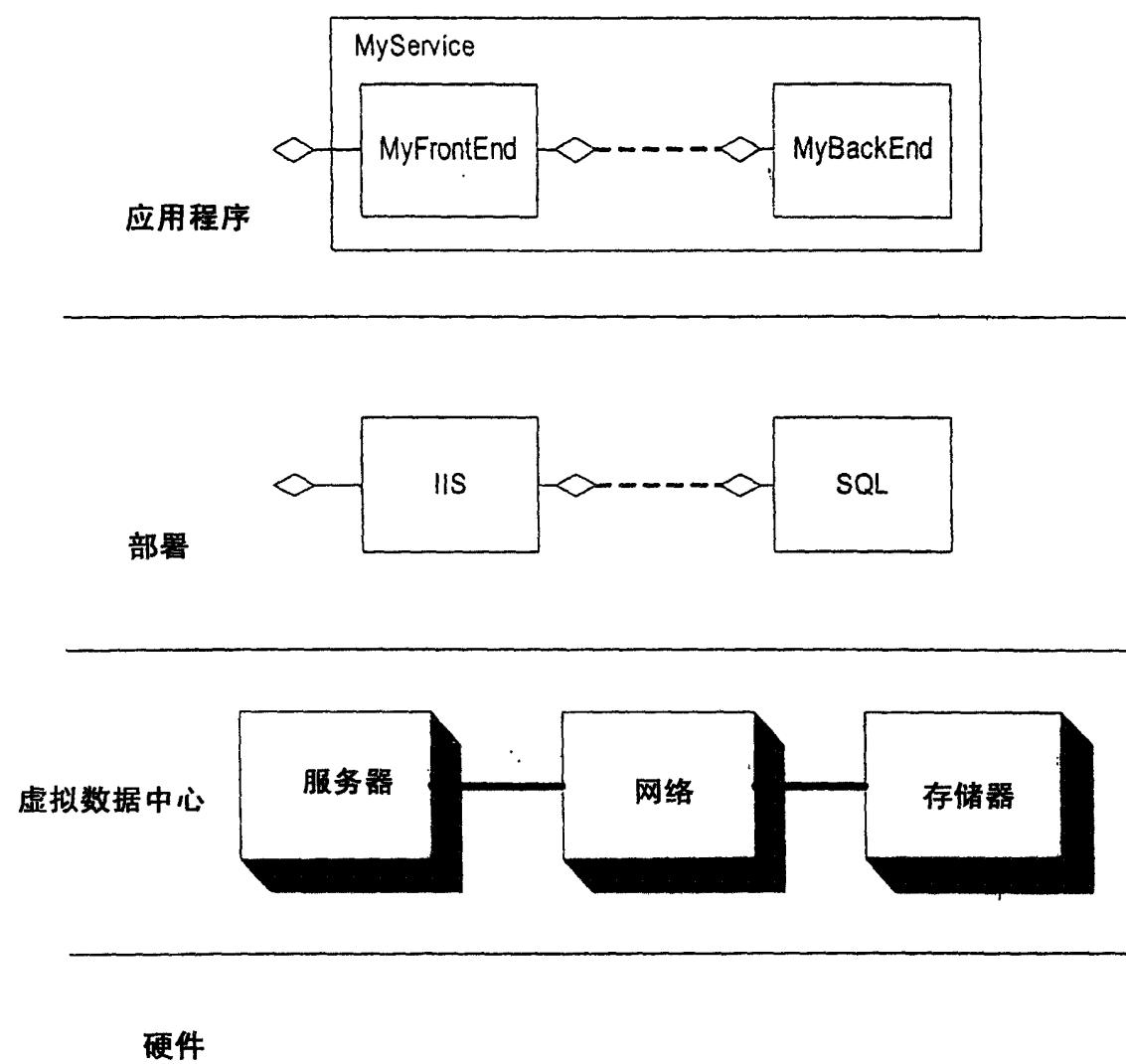


图 11

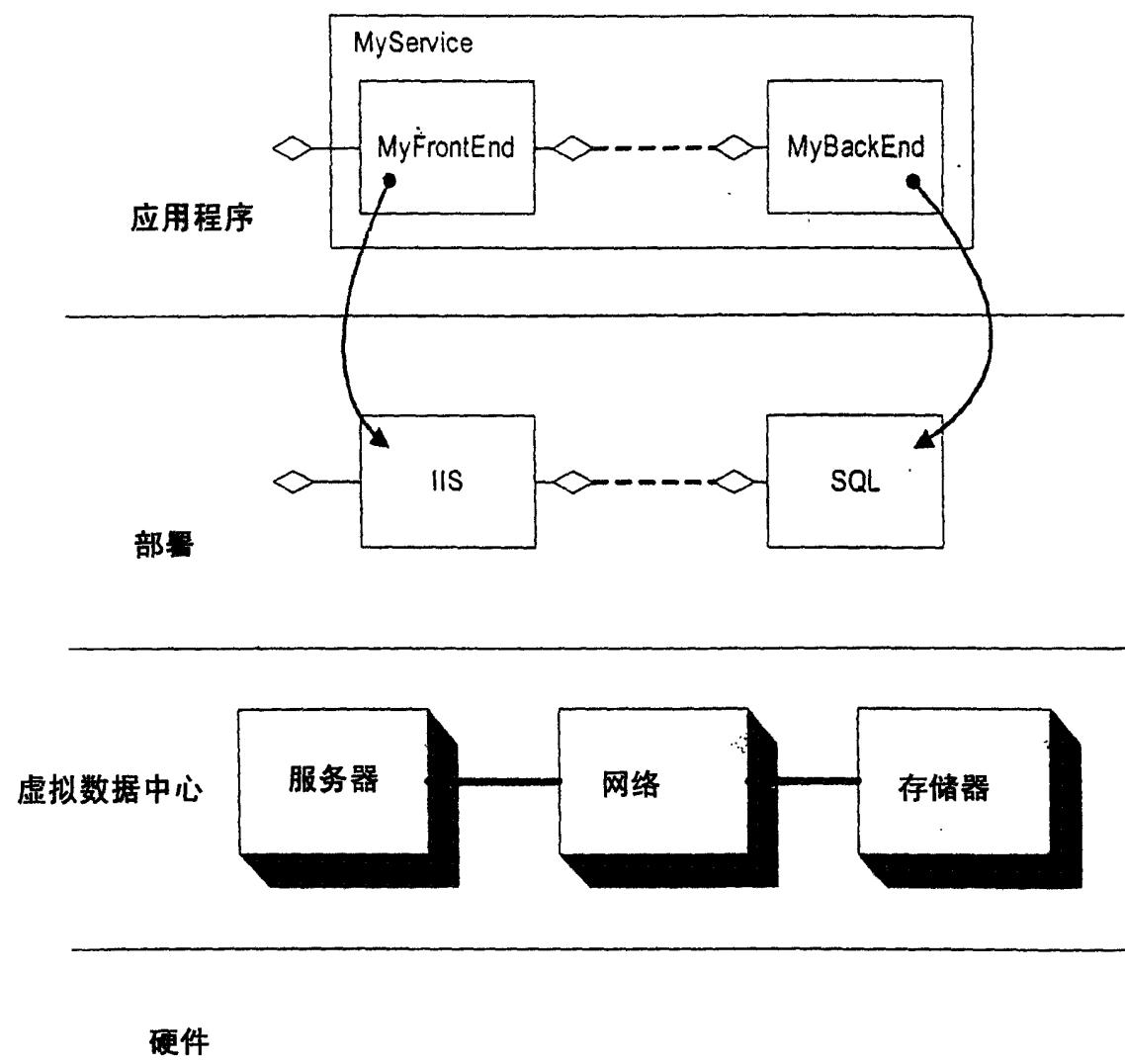


图 12

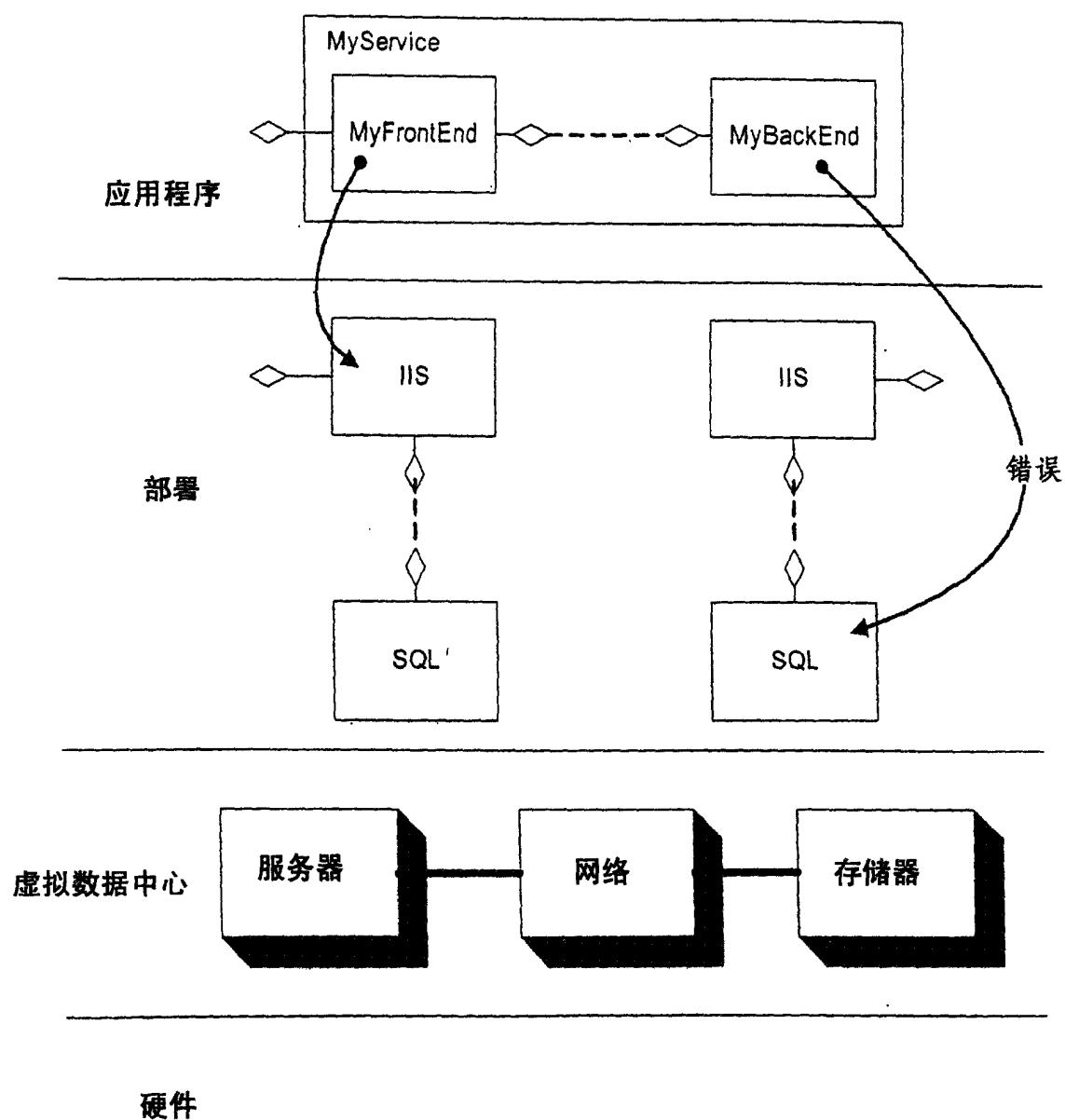


图 13

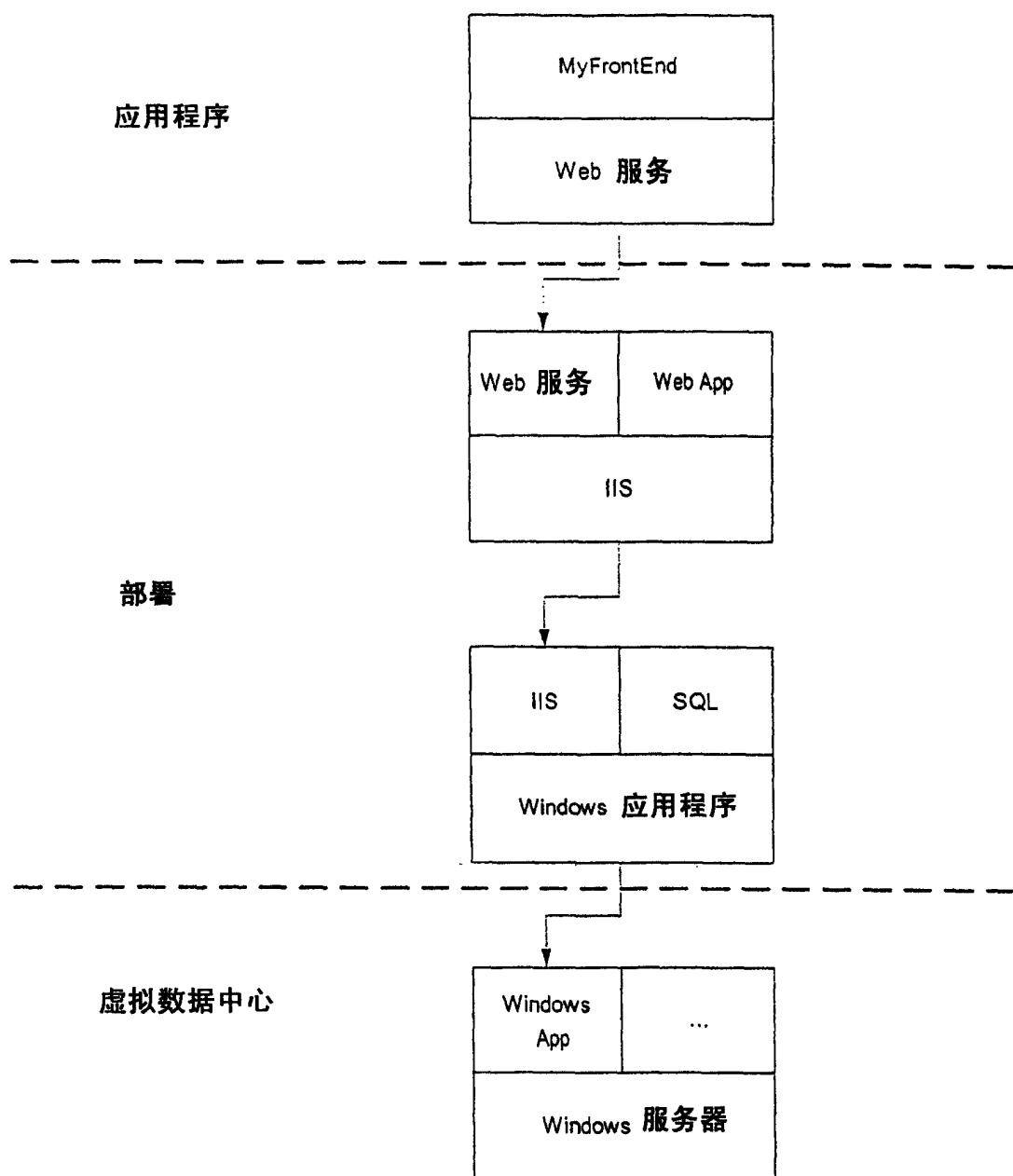


图 14

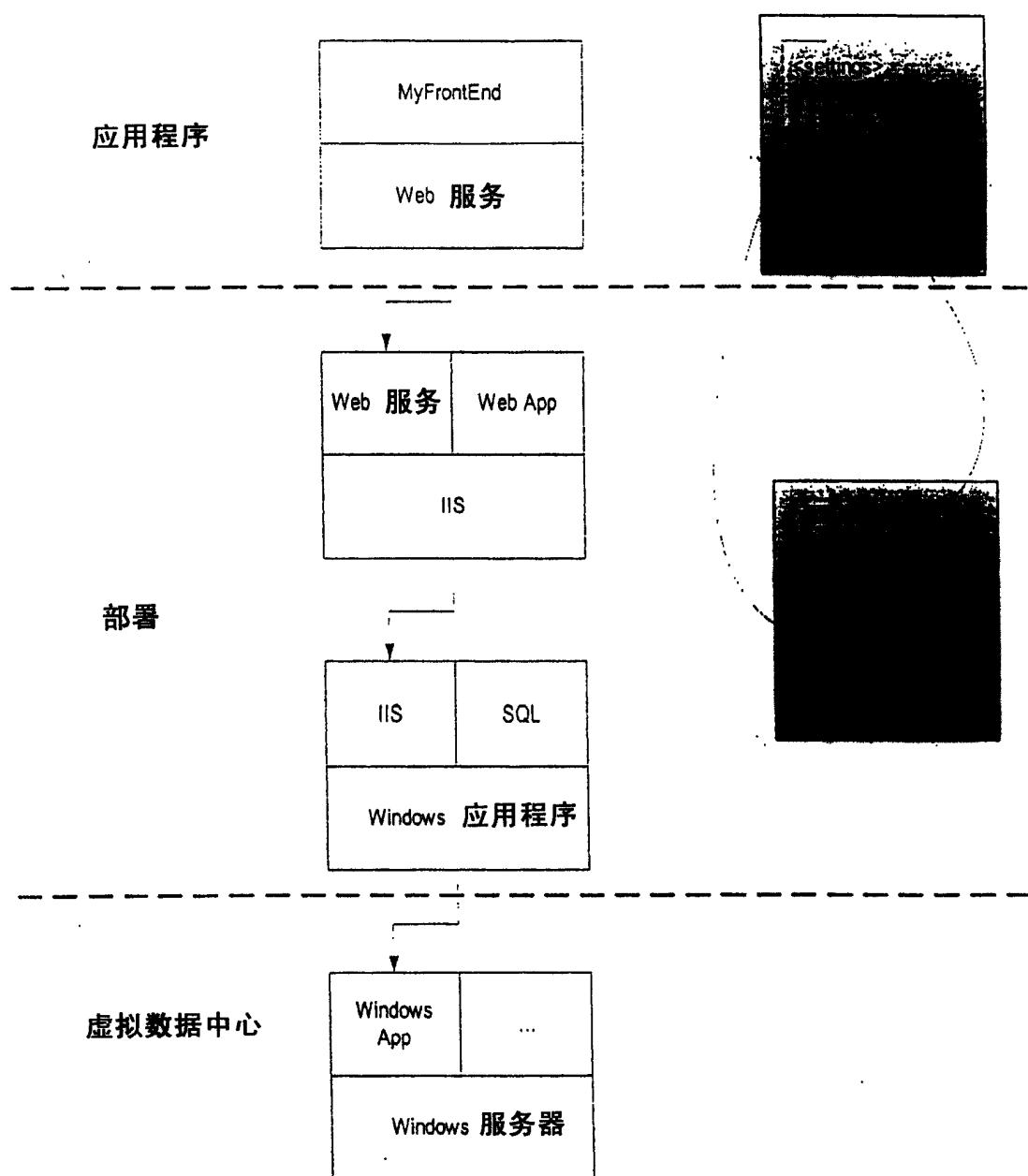


图 15

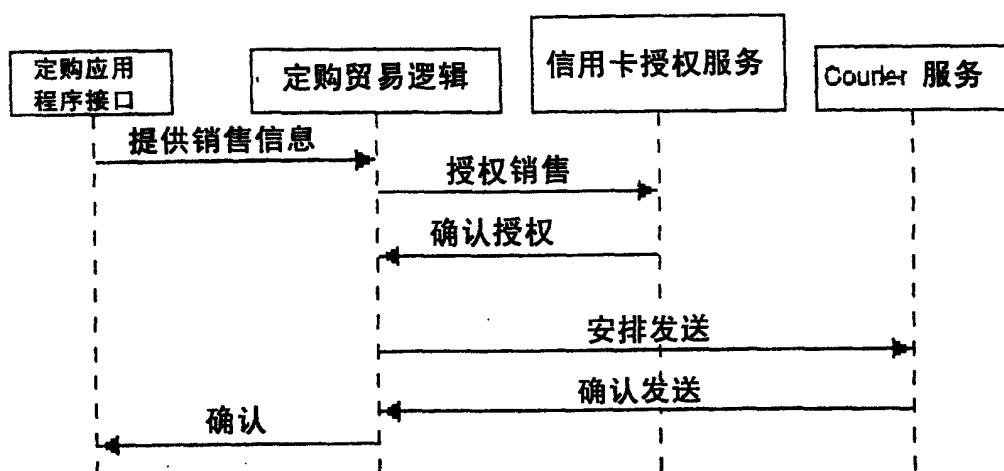


图 16

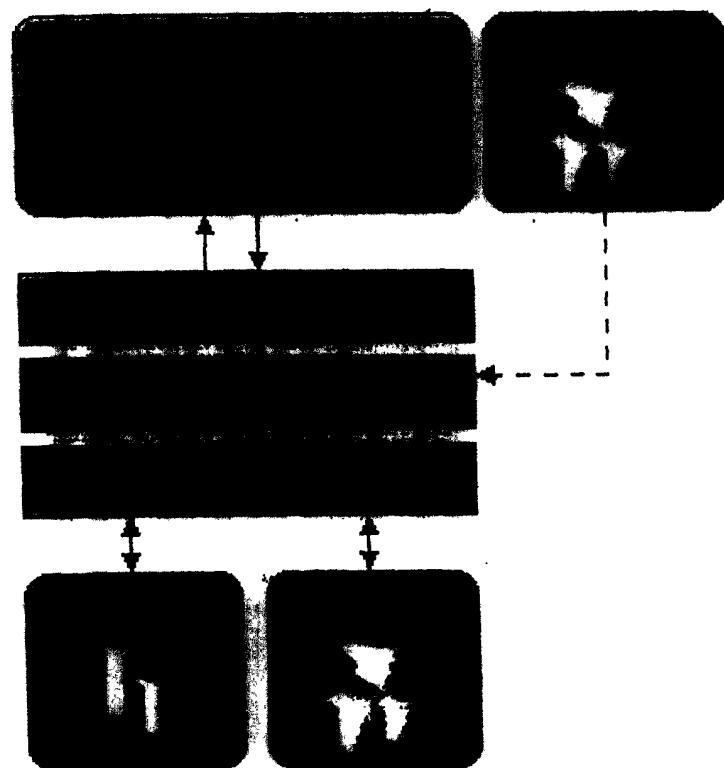


图 18

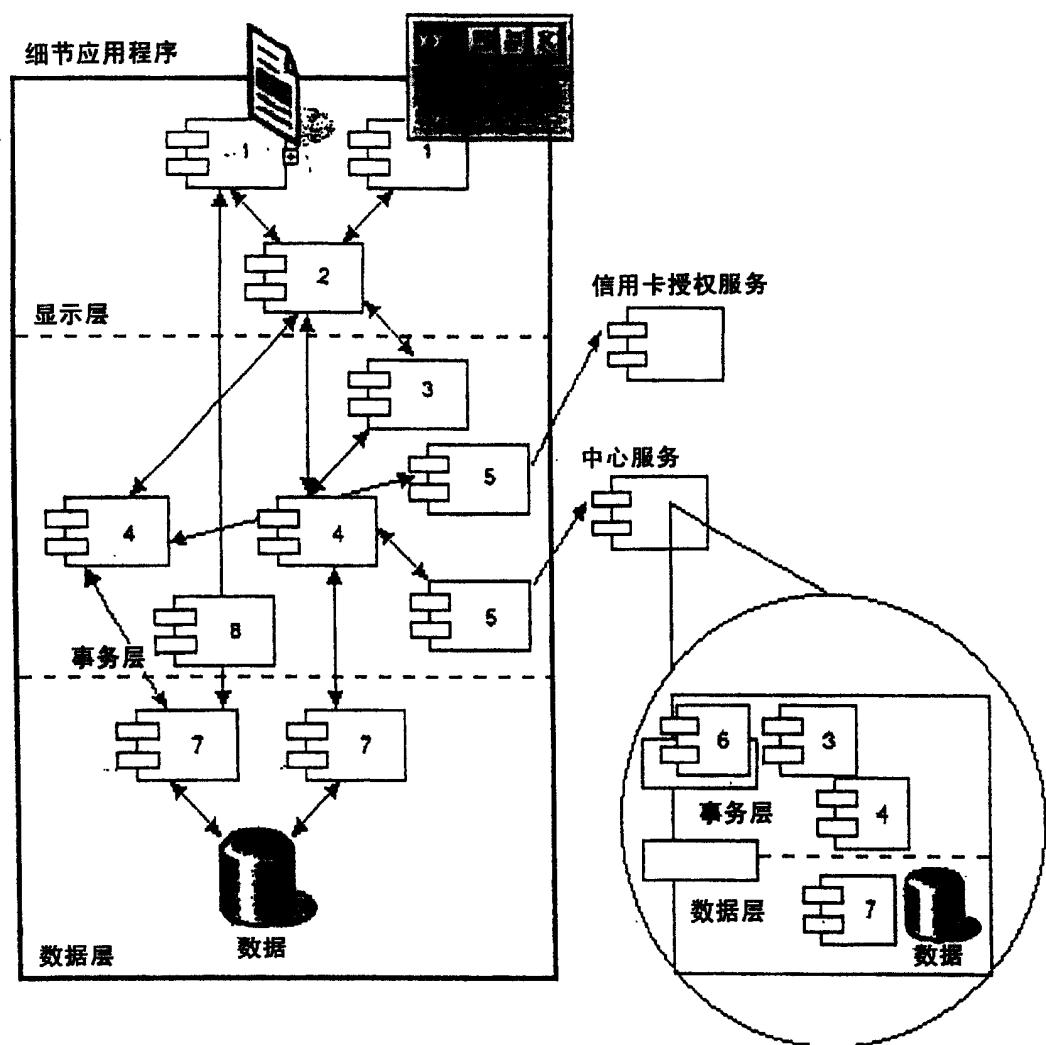


图 17

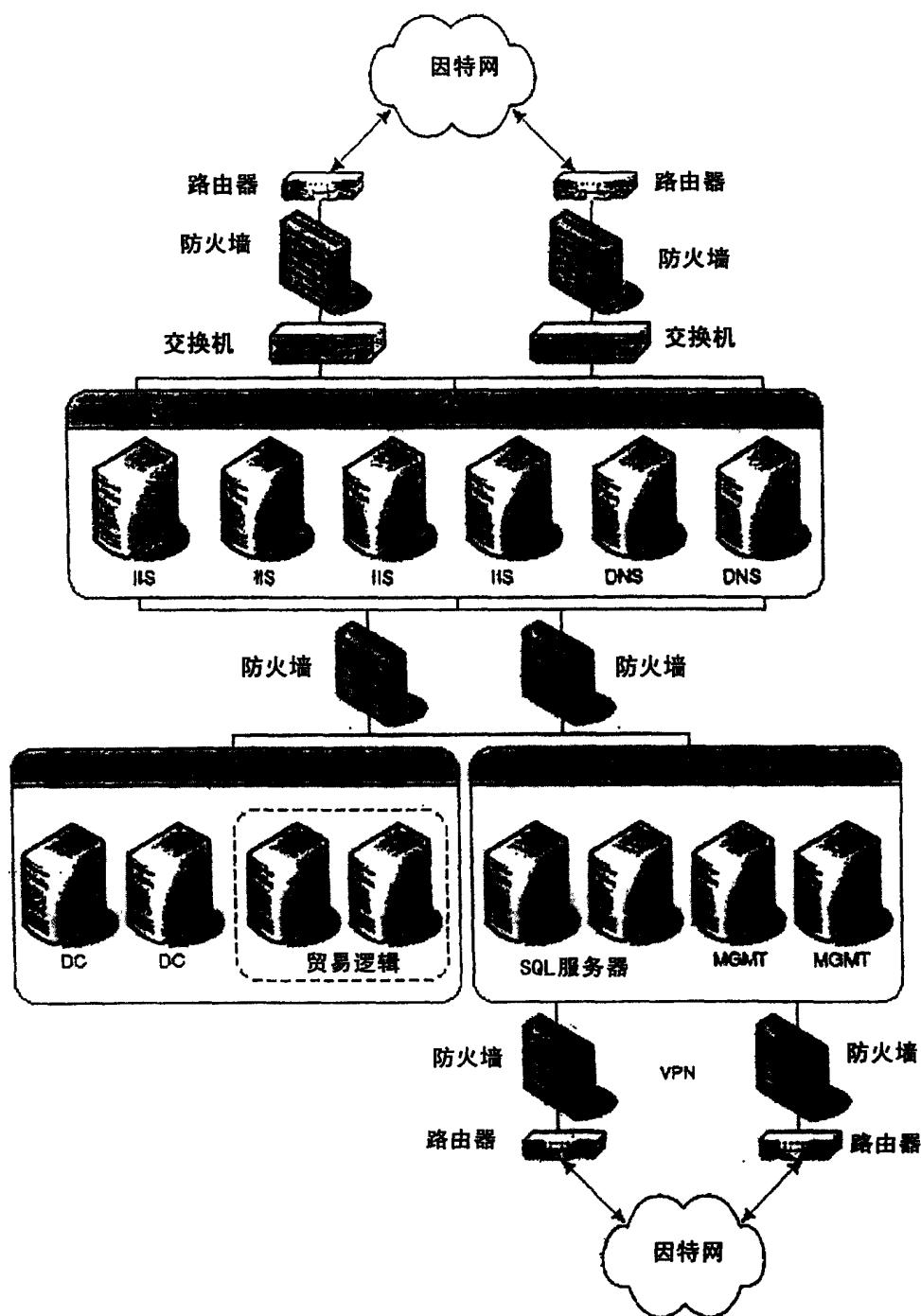


图 19

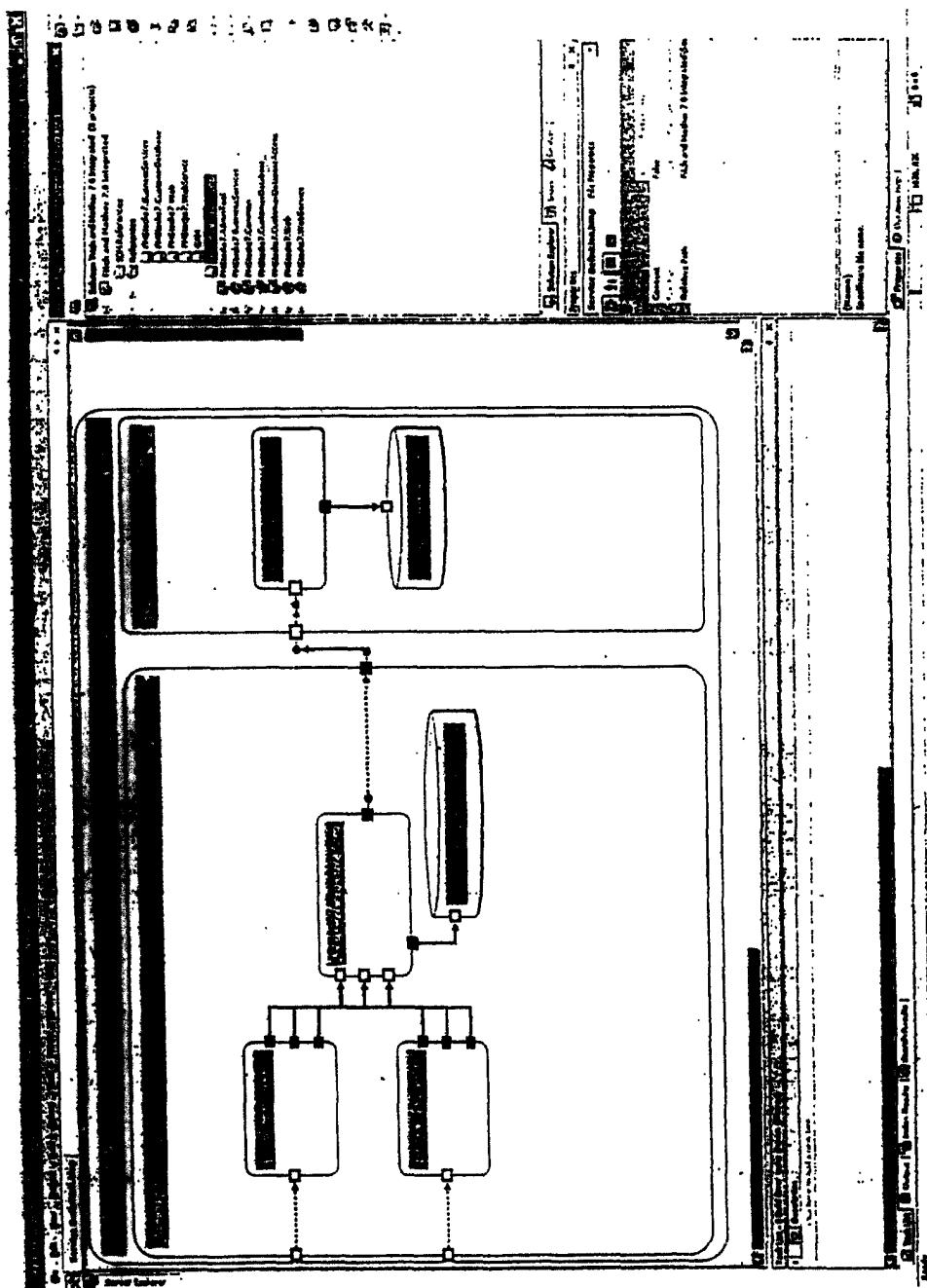
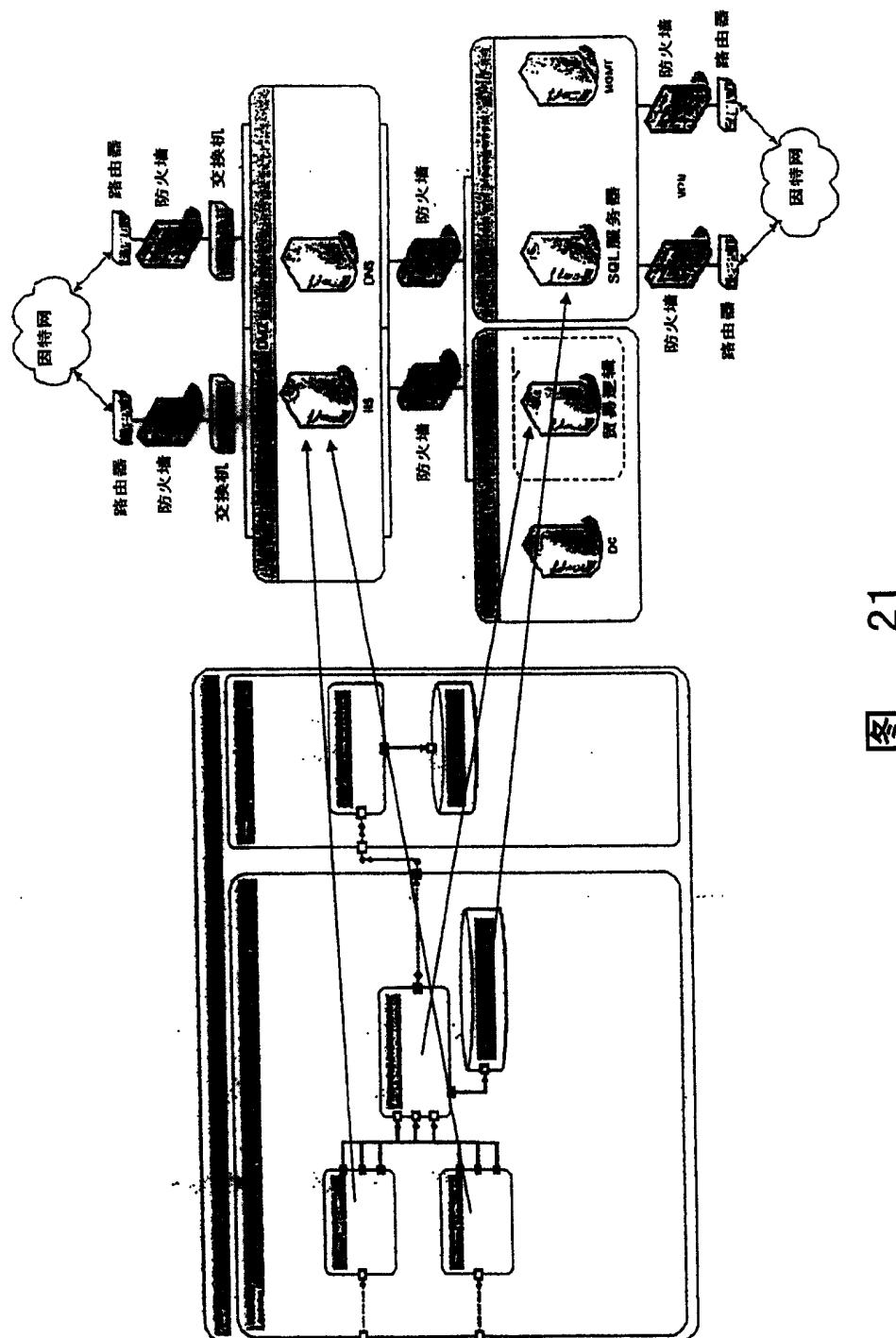


图 20



21

图

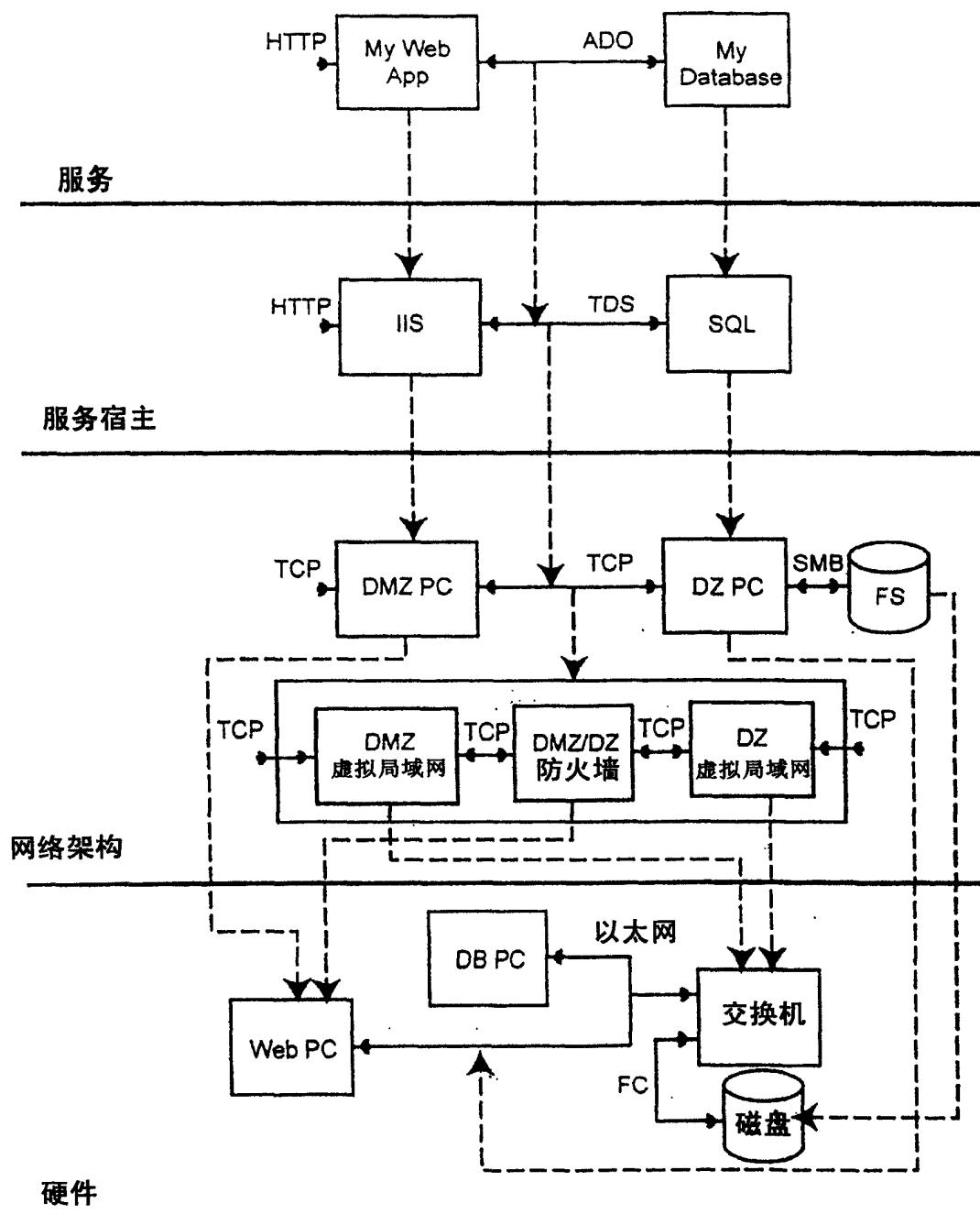


图 22

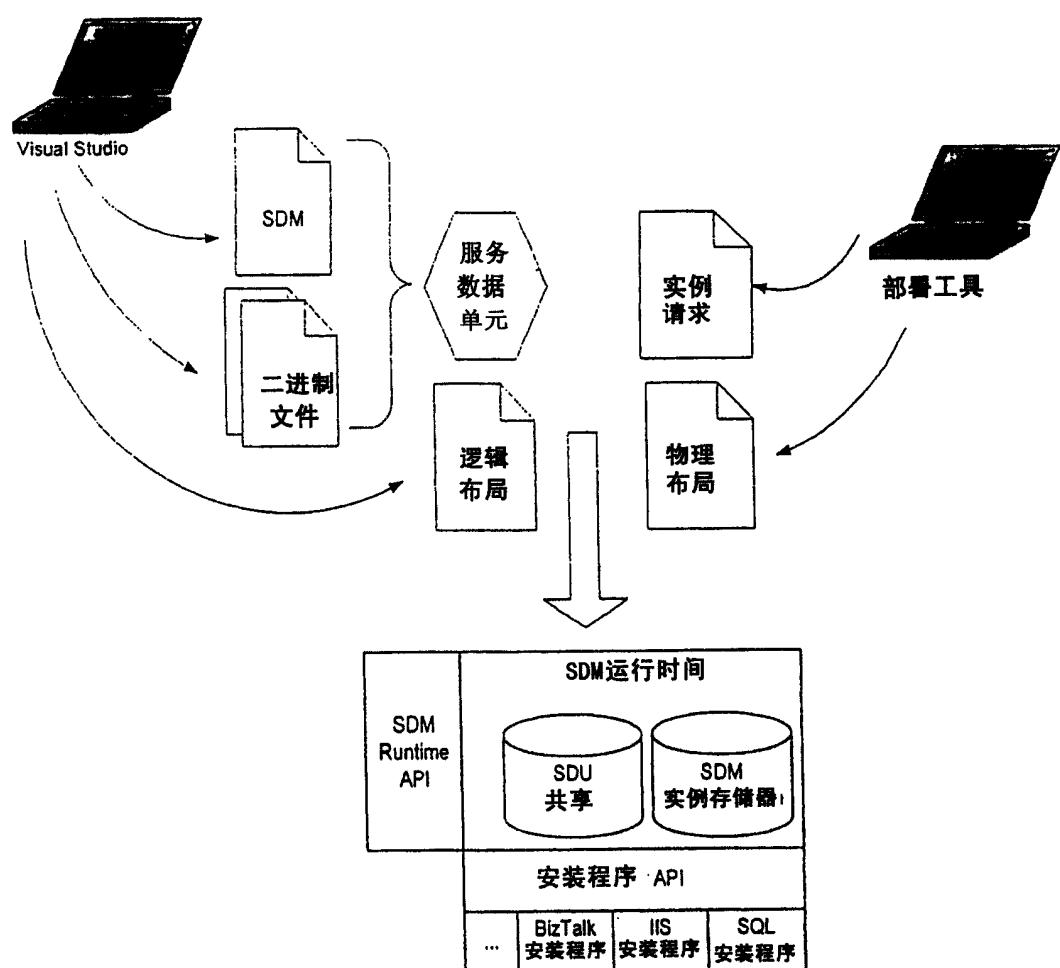


图 23

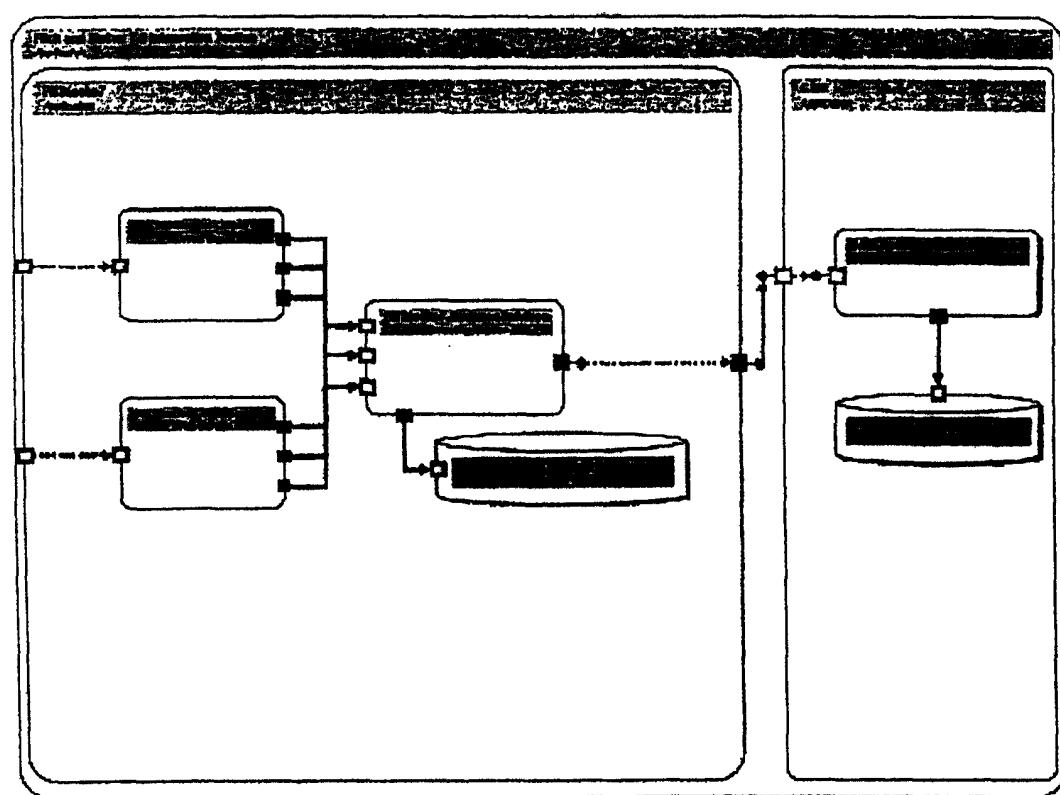


图 24

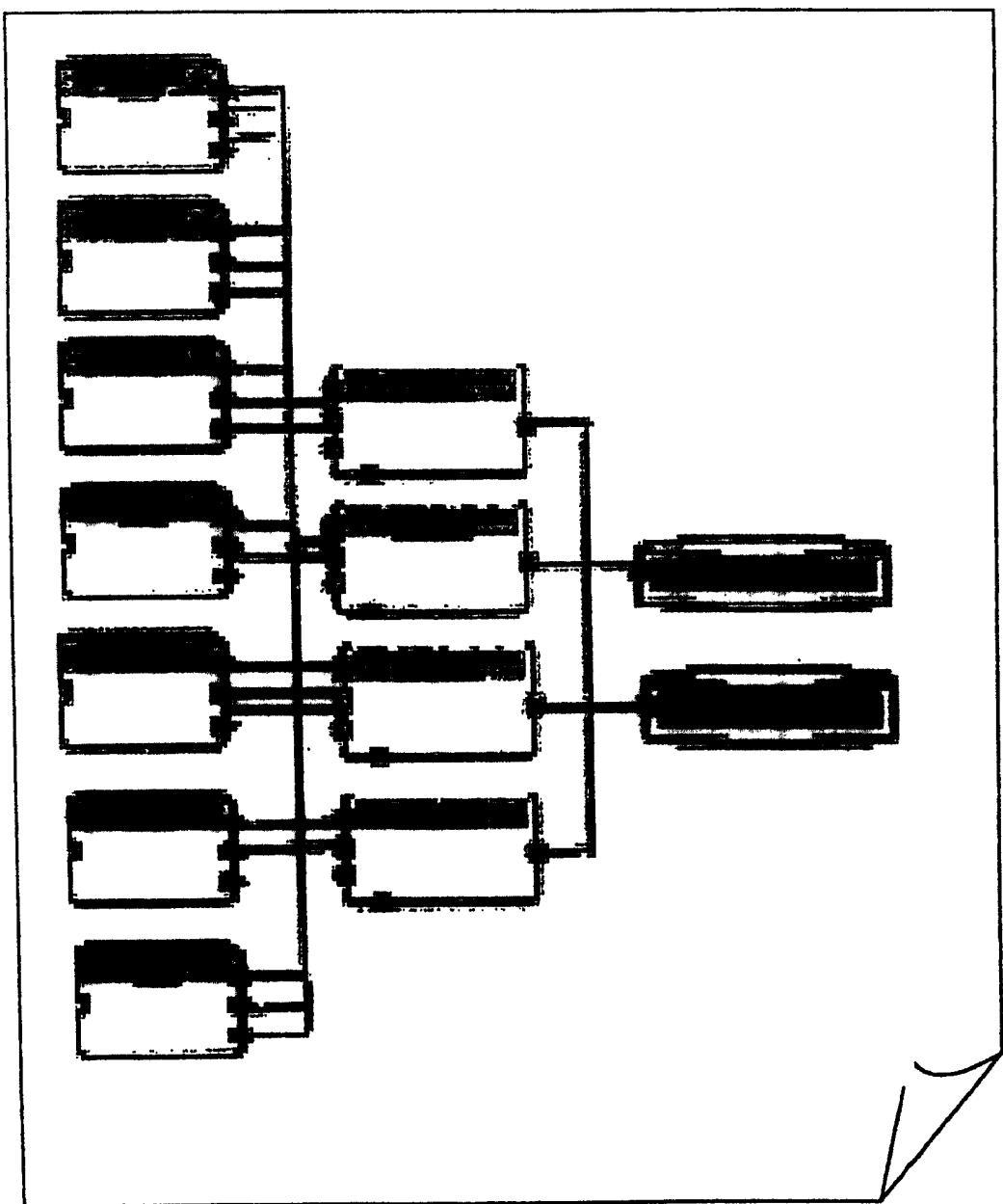


图 25

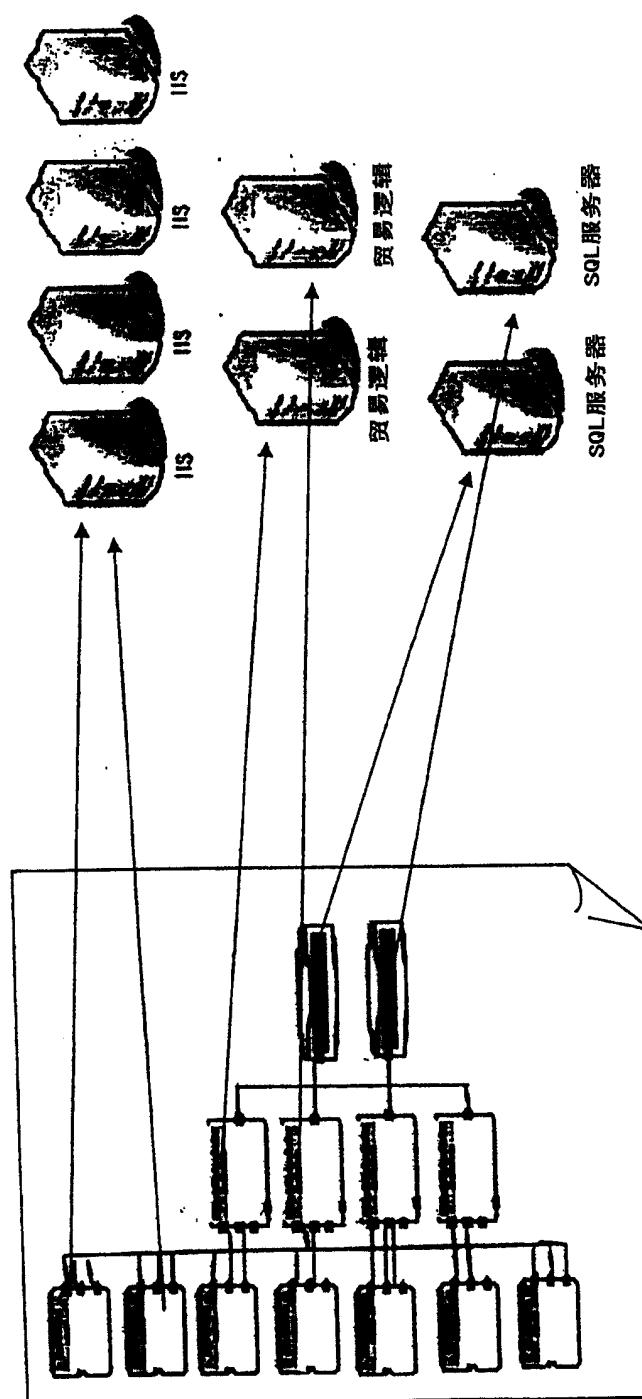


图 26

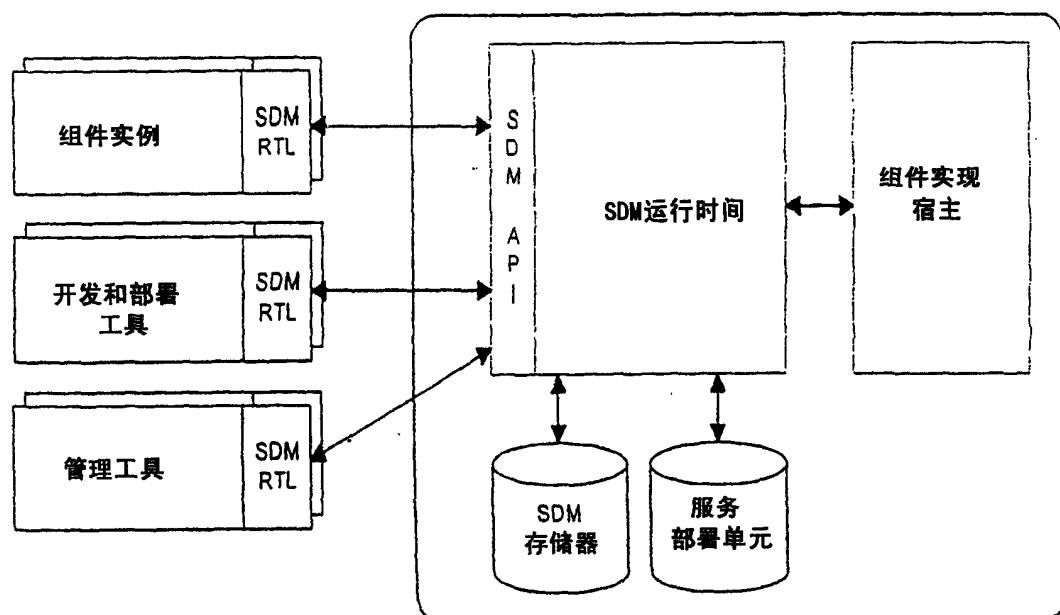


图 27

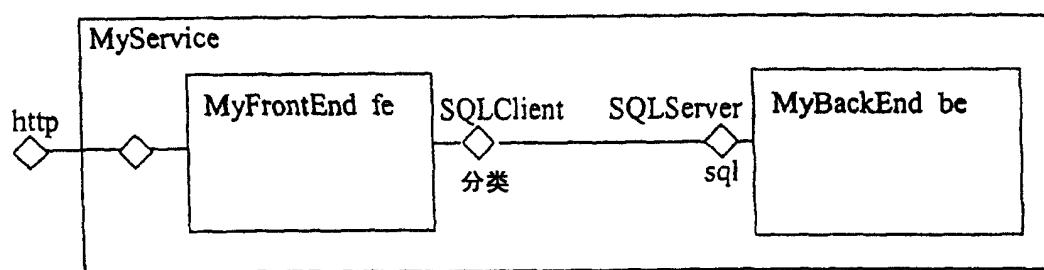


图 28

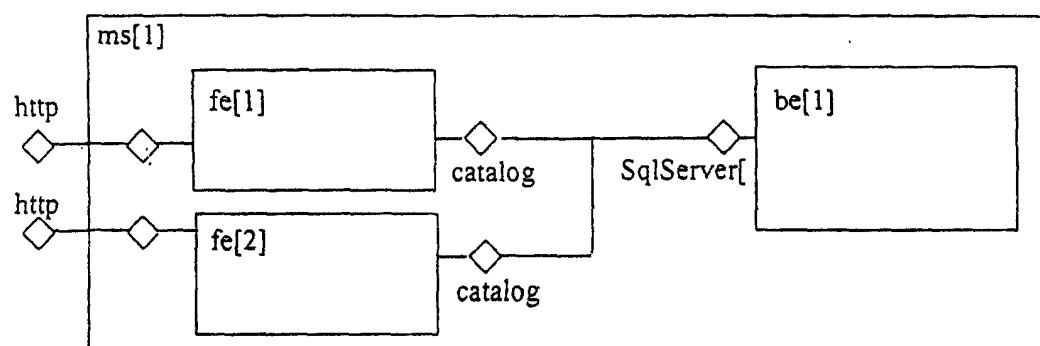


图 29

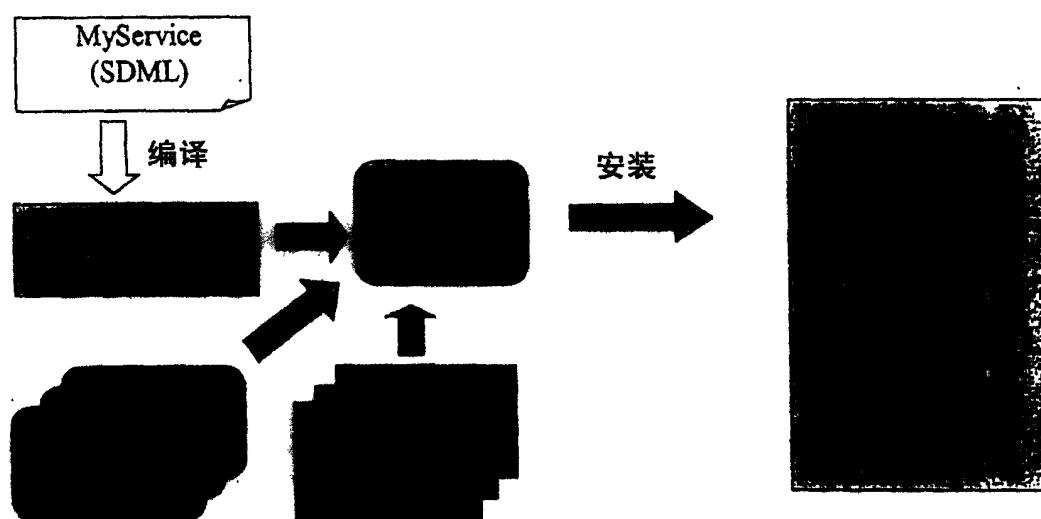


图 30

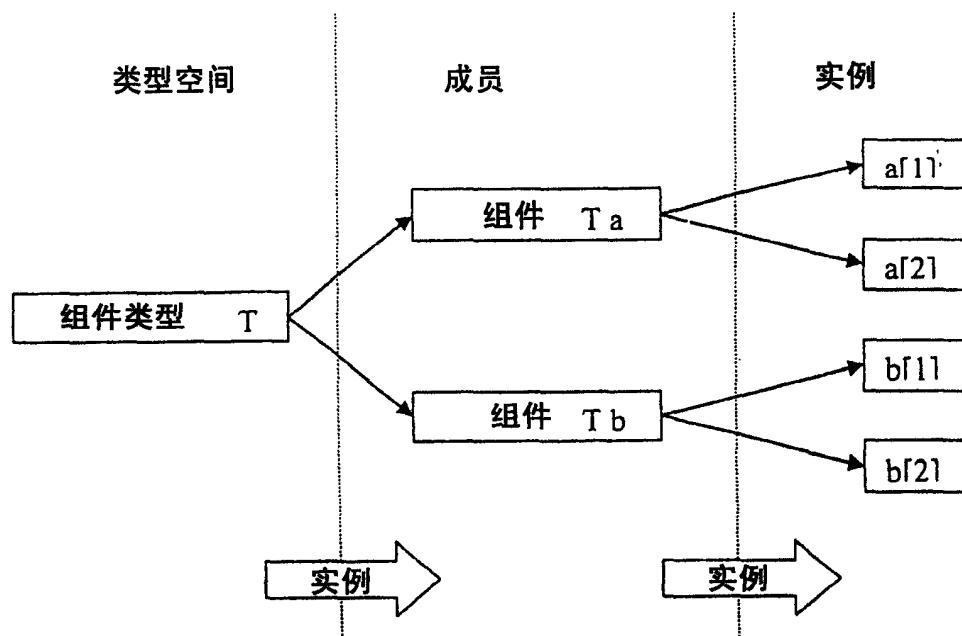


图 31

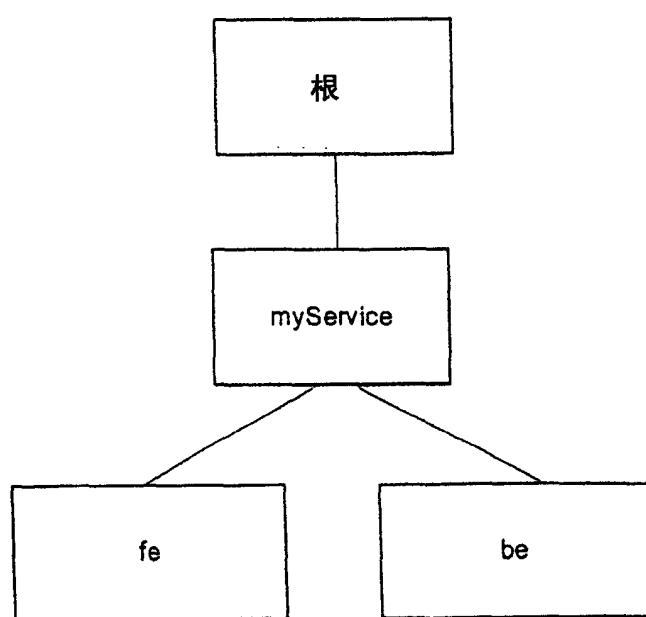


图 32

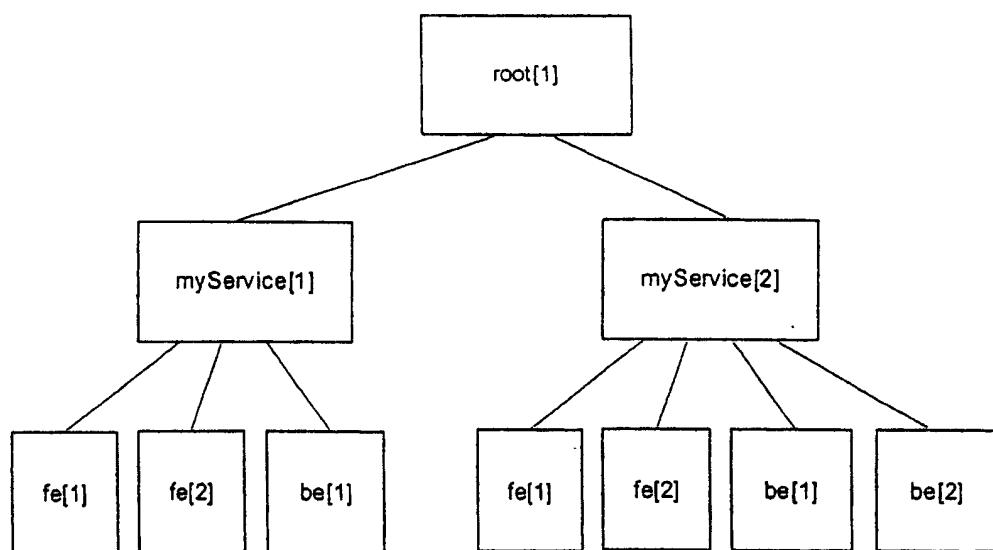


图 33

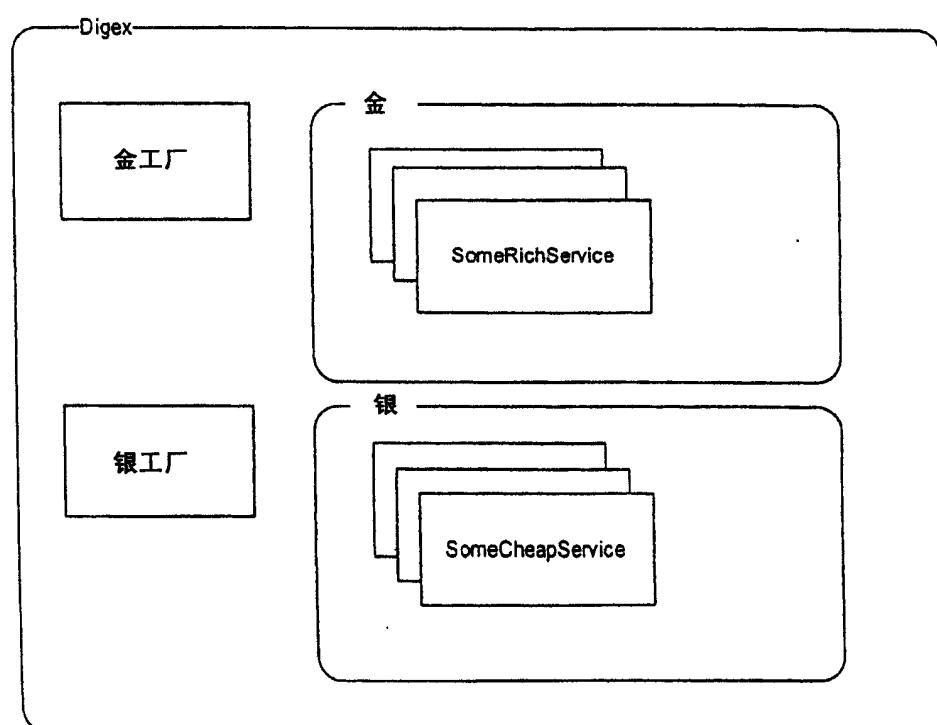


图 34

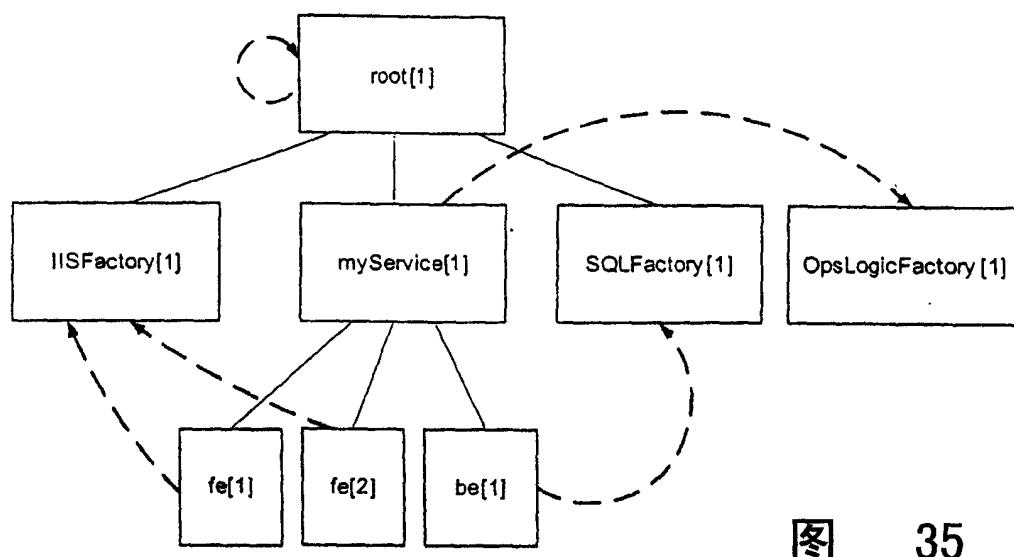


图 35

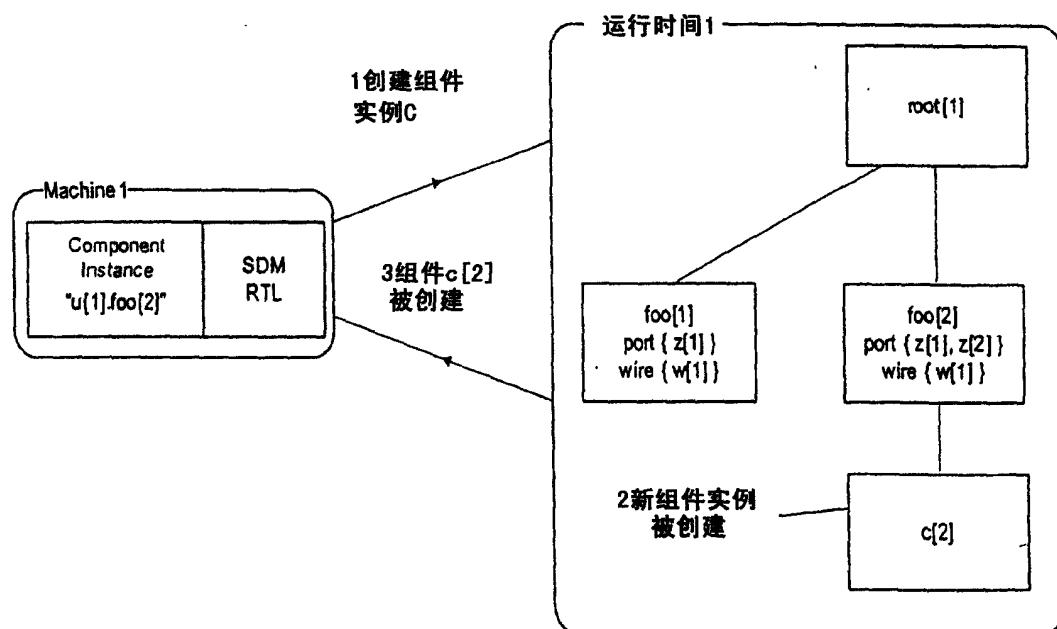


图 36

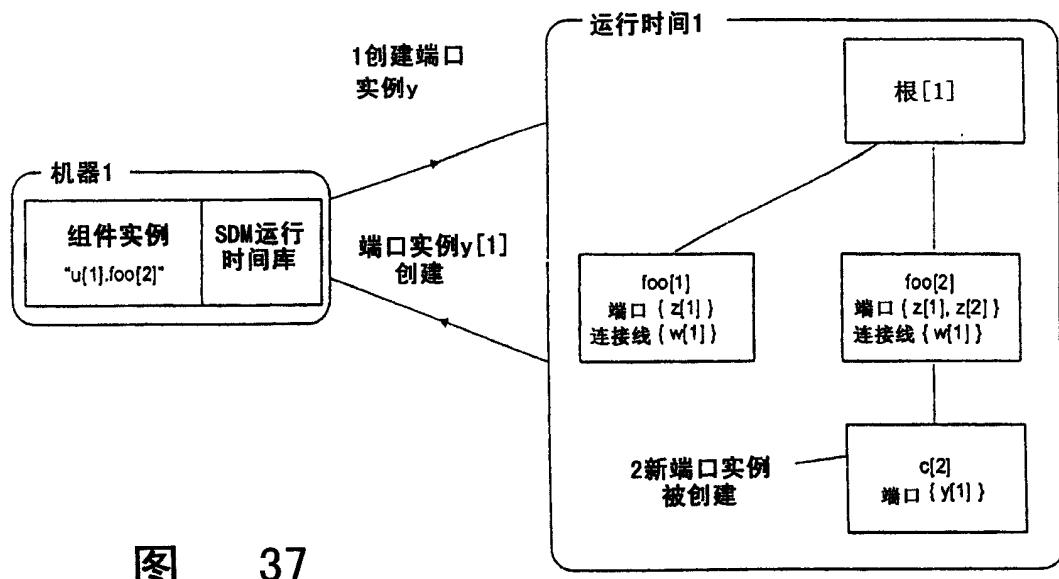


图 37

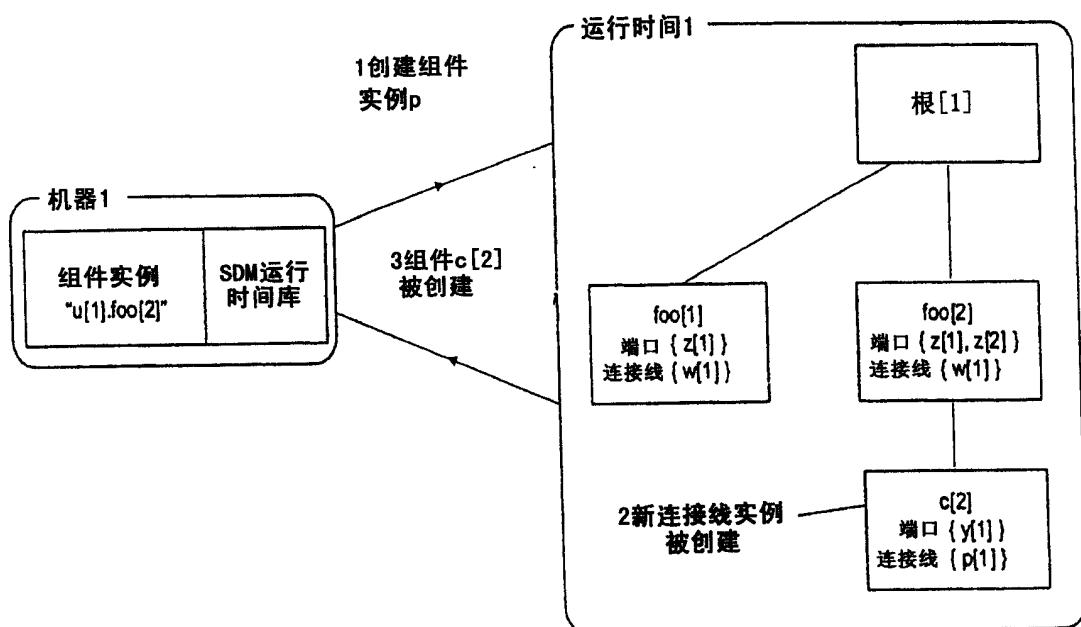


图 38

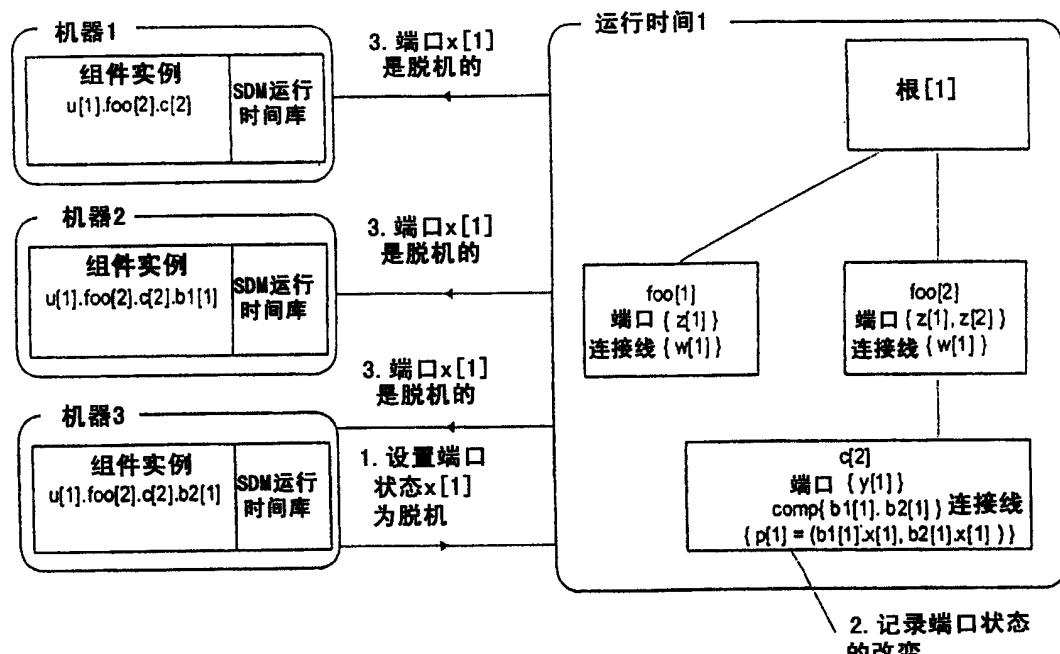


图 39

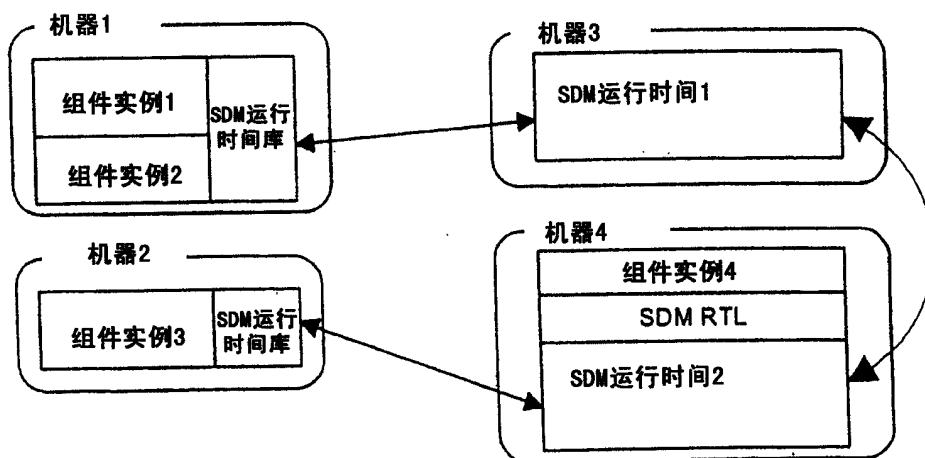


图 40

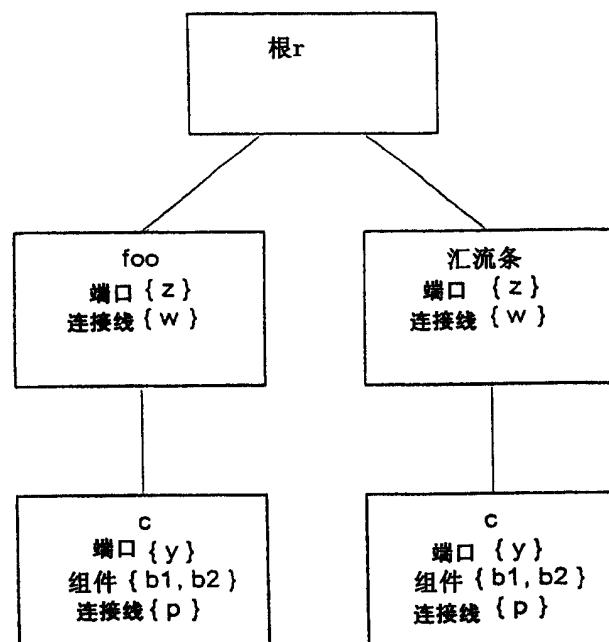


图 41

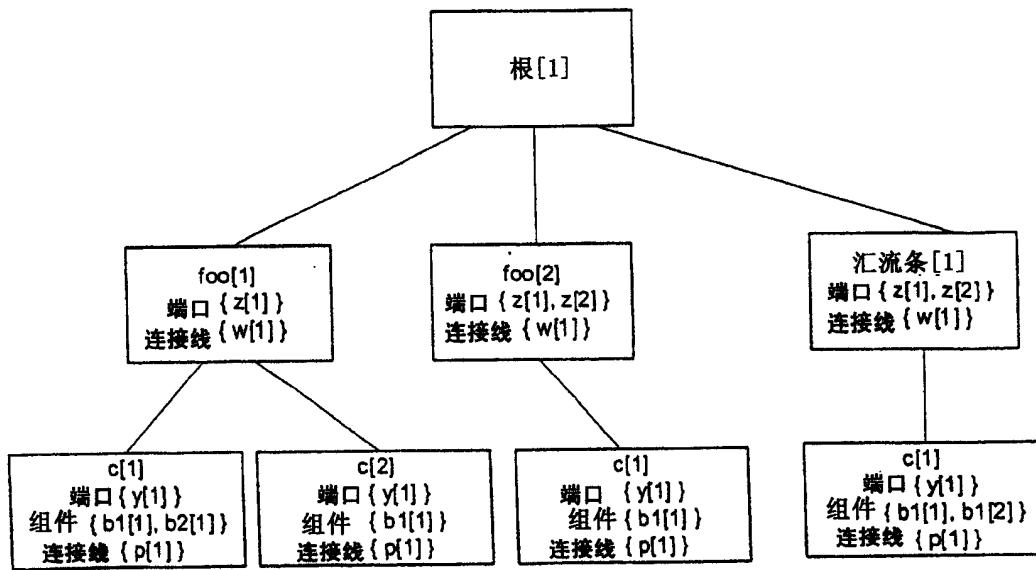


图 42

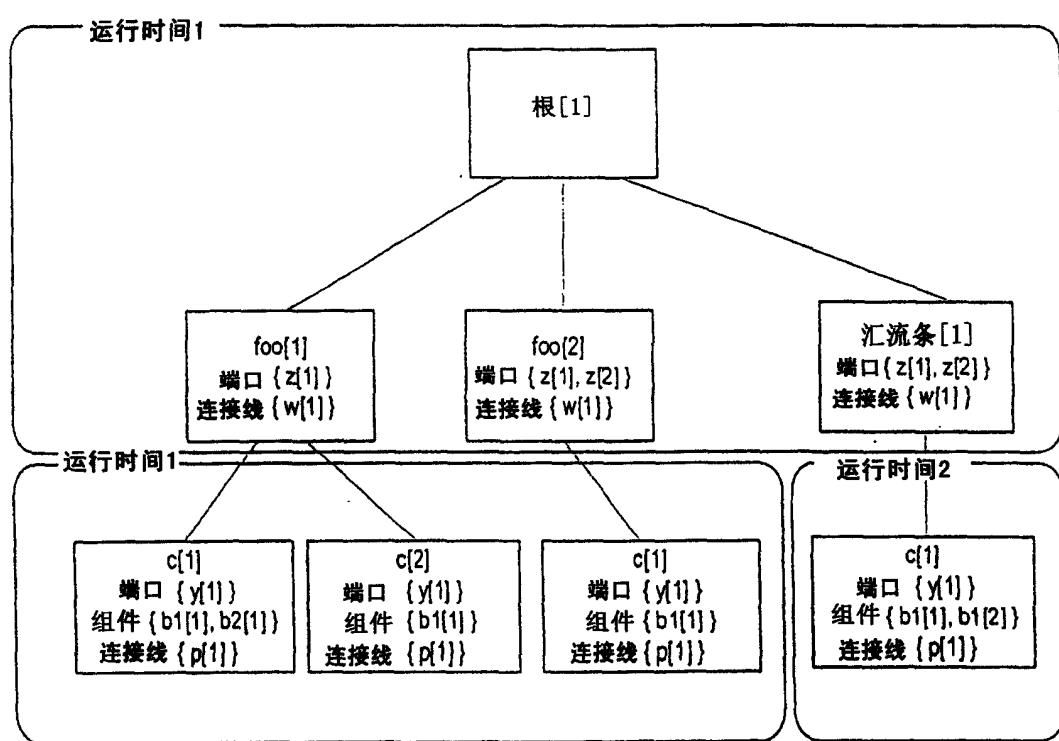


图 43

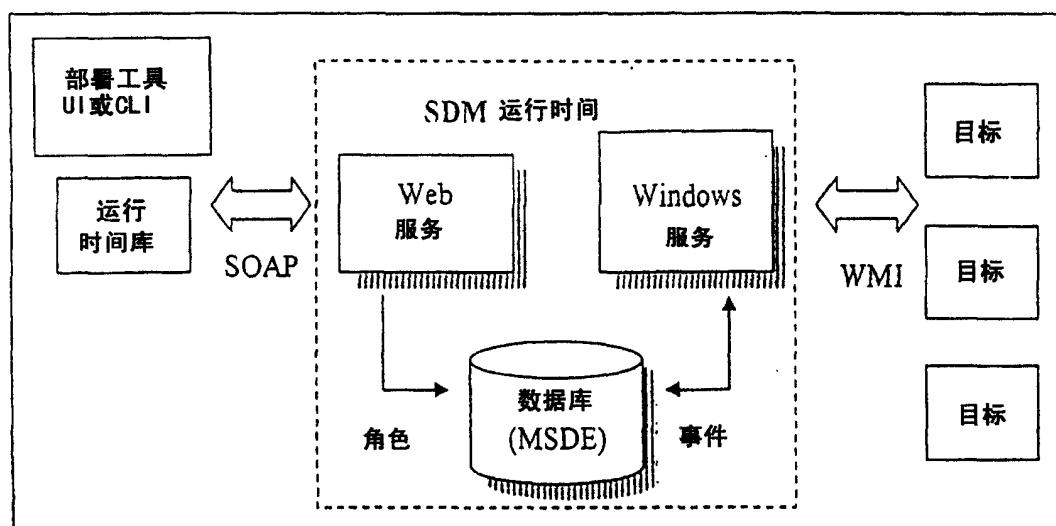


图 44

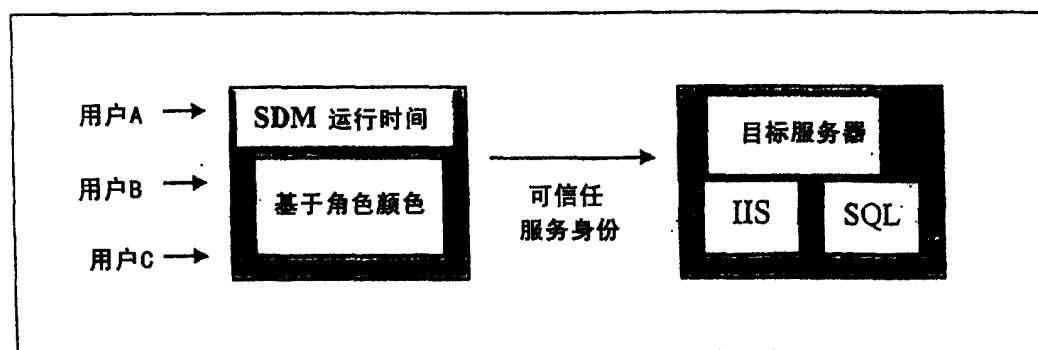


图 45

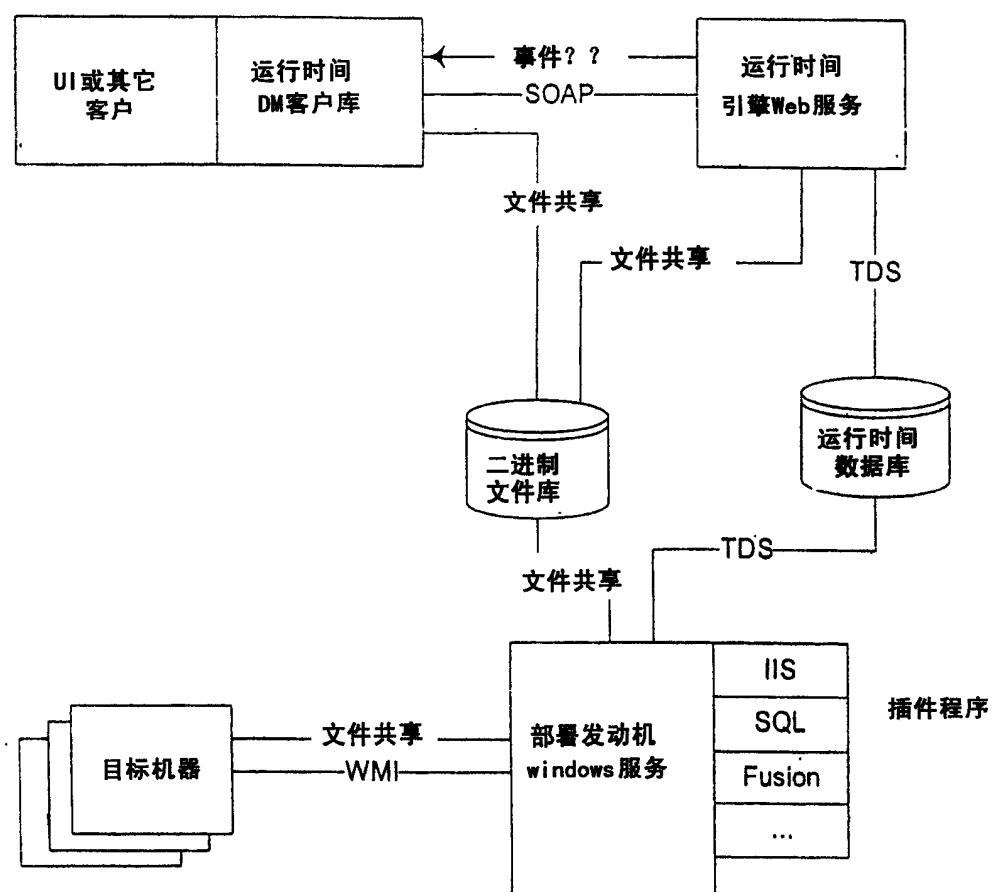


图 46

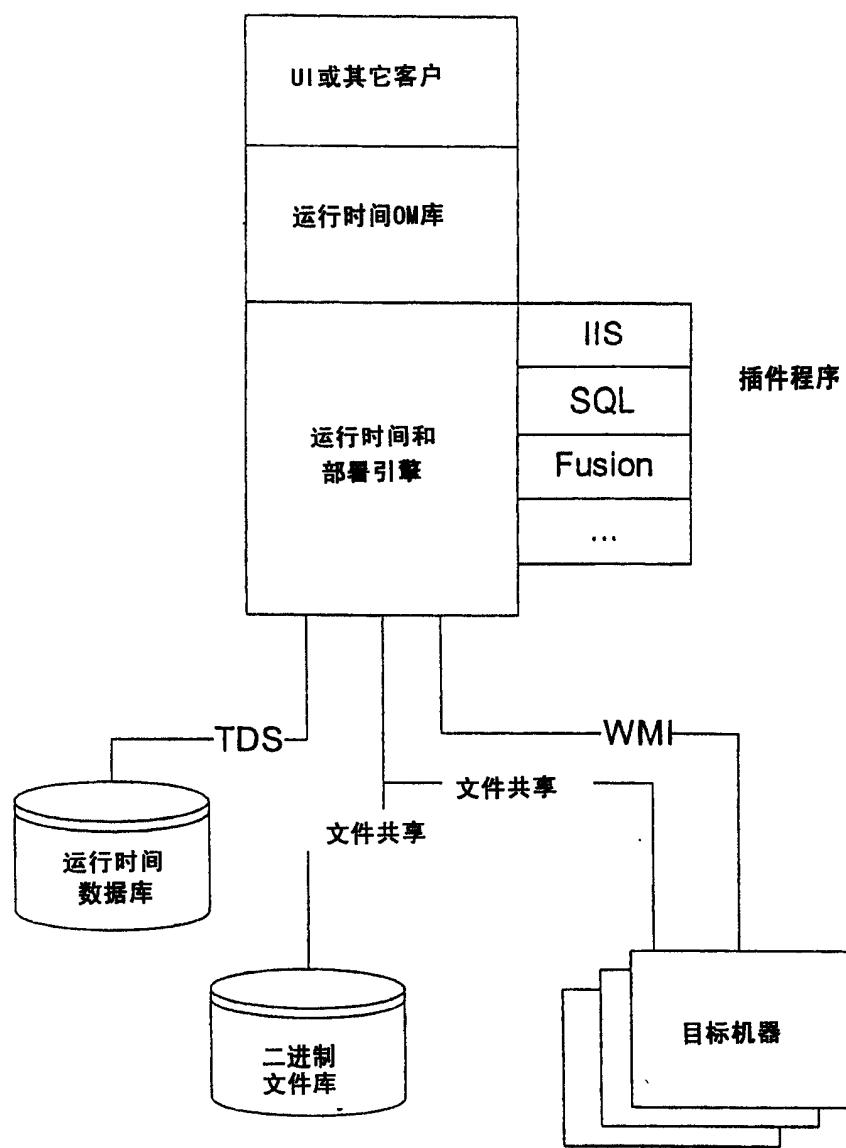


图 47

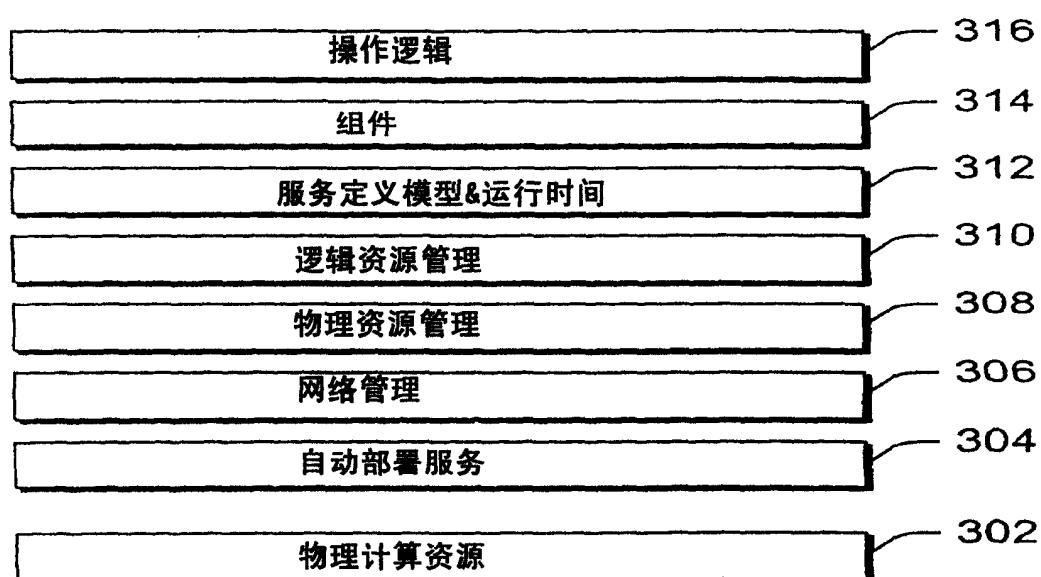


图 48

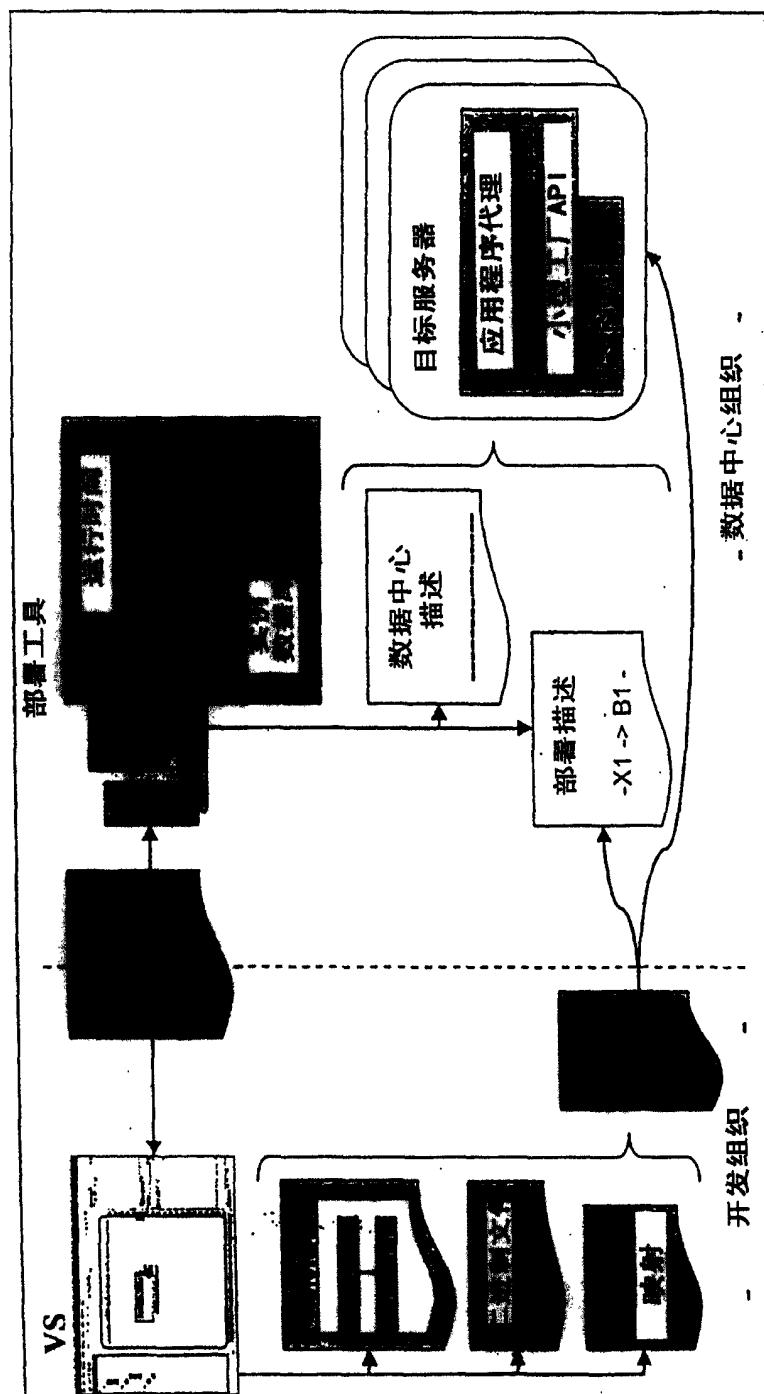


图 49

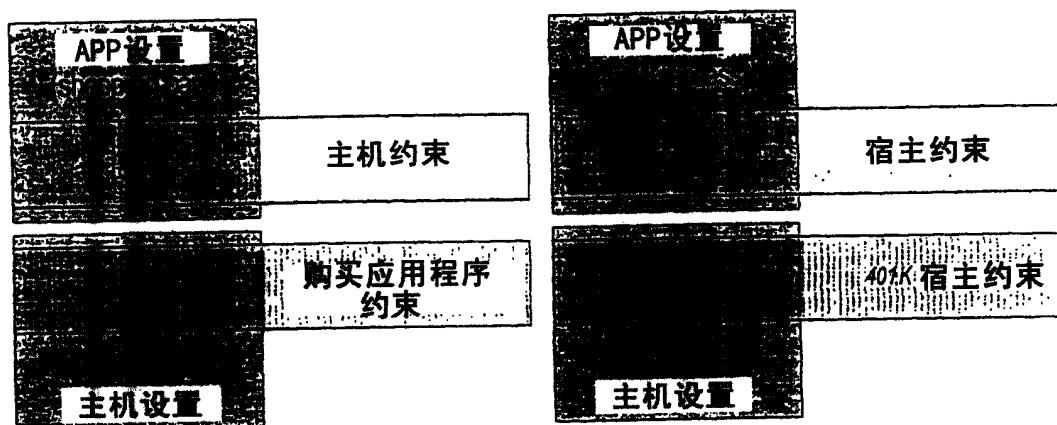


图 50

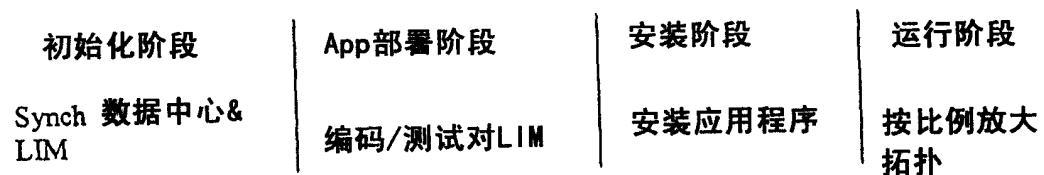


图 51

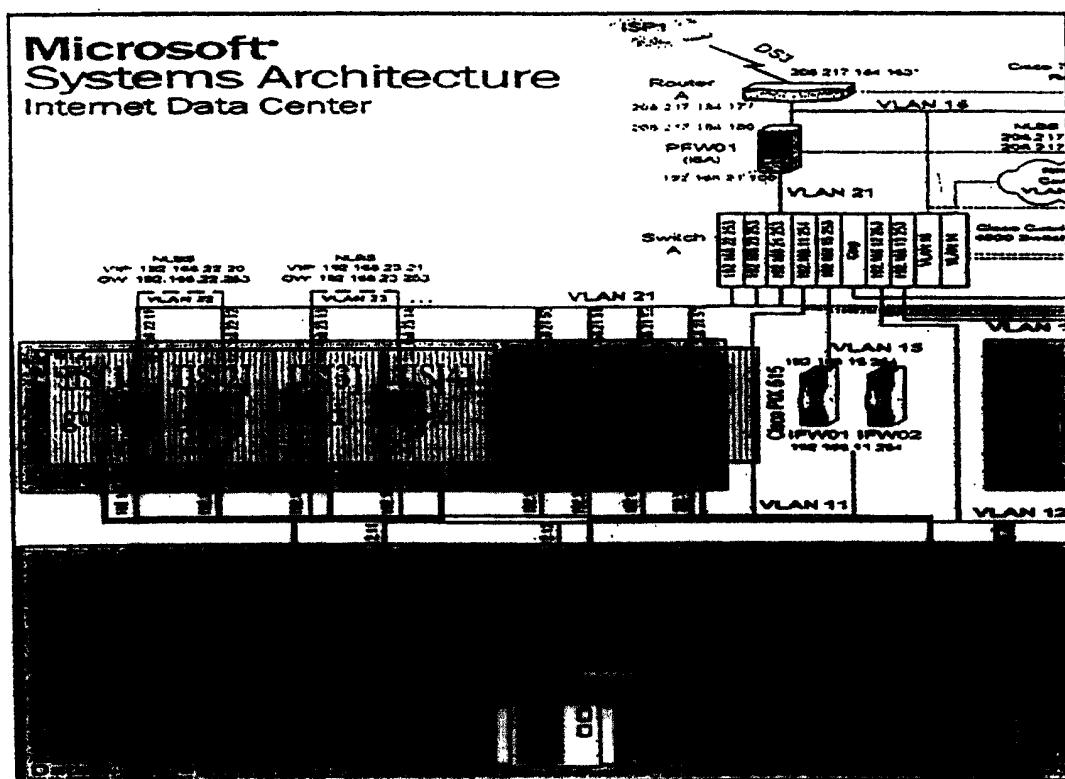


图 52

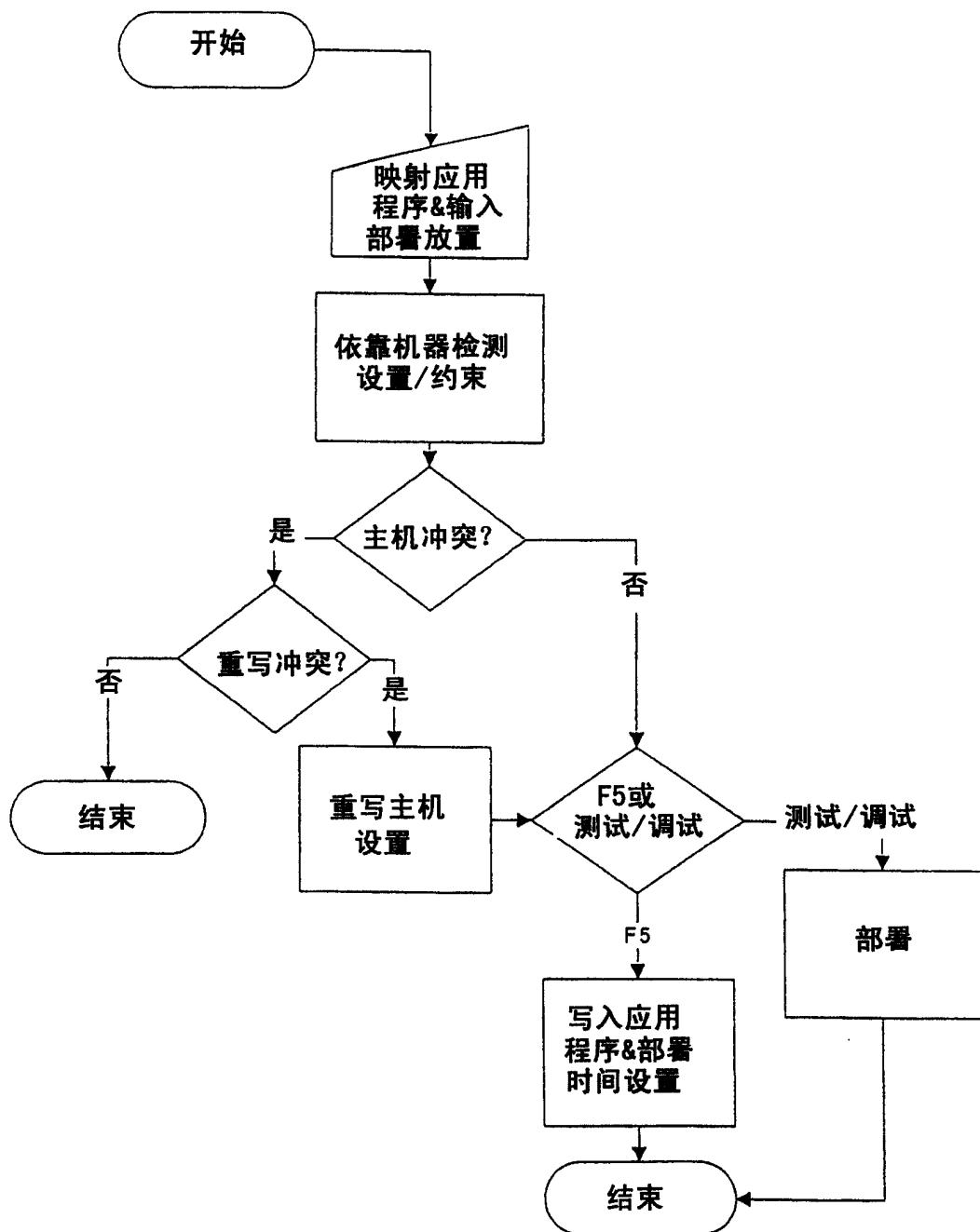


图 53

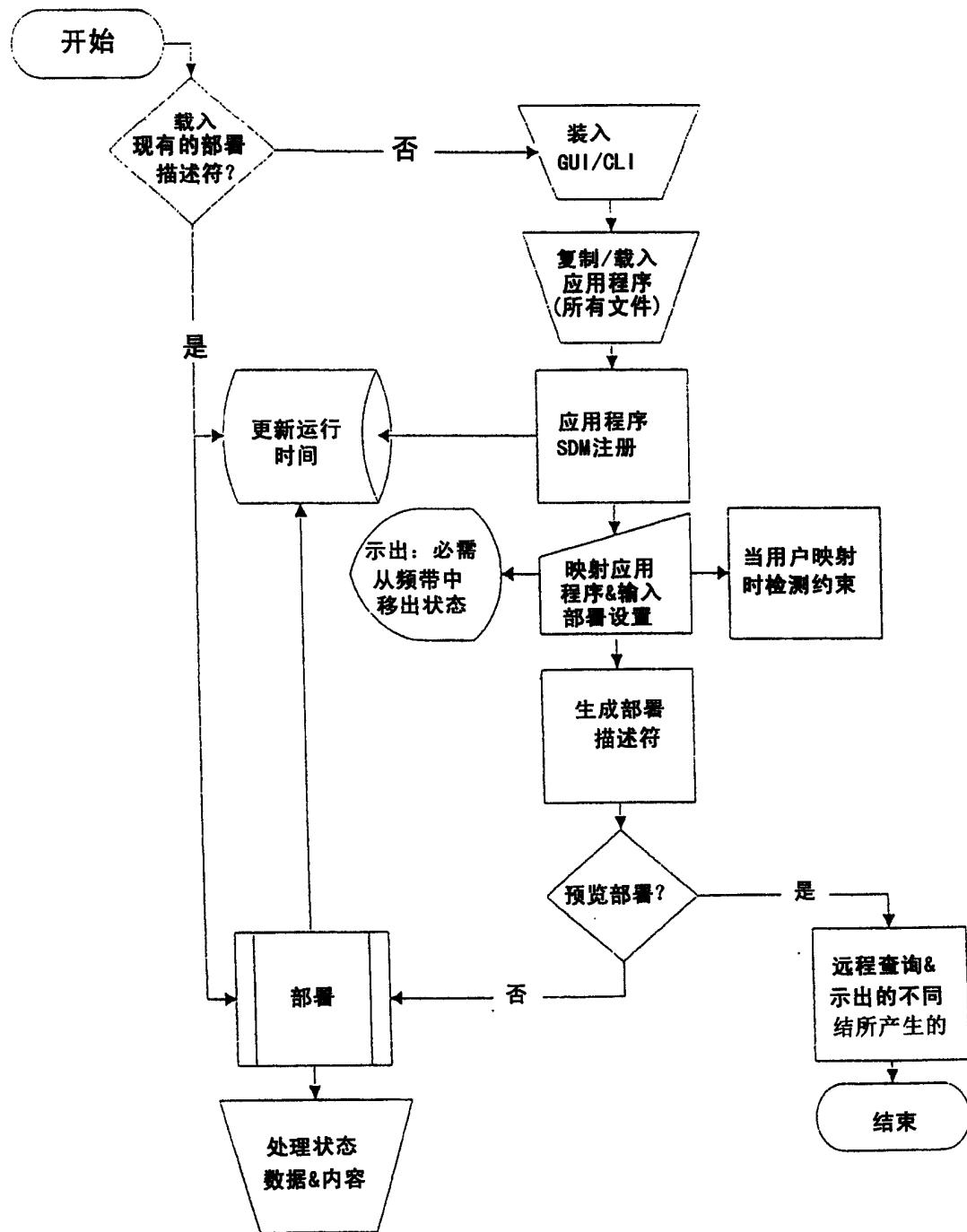


图 54

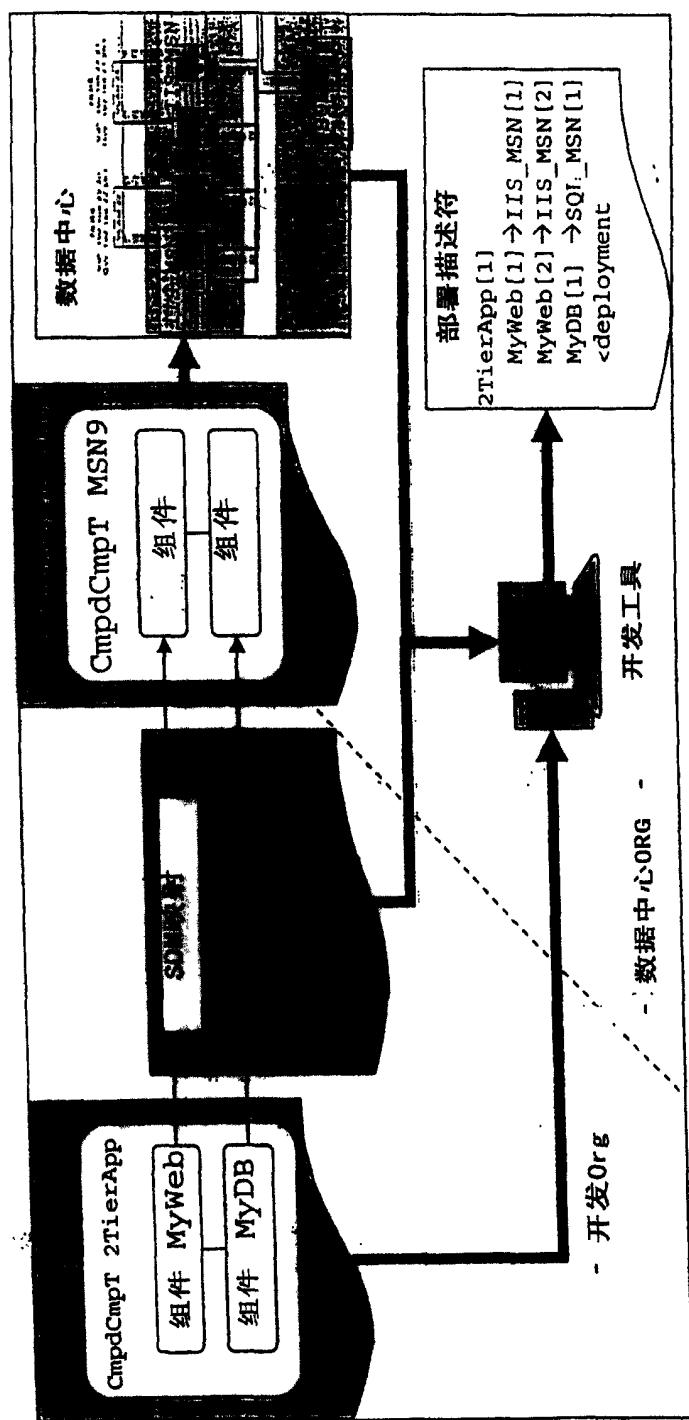


图 55

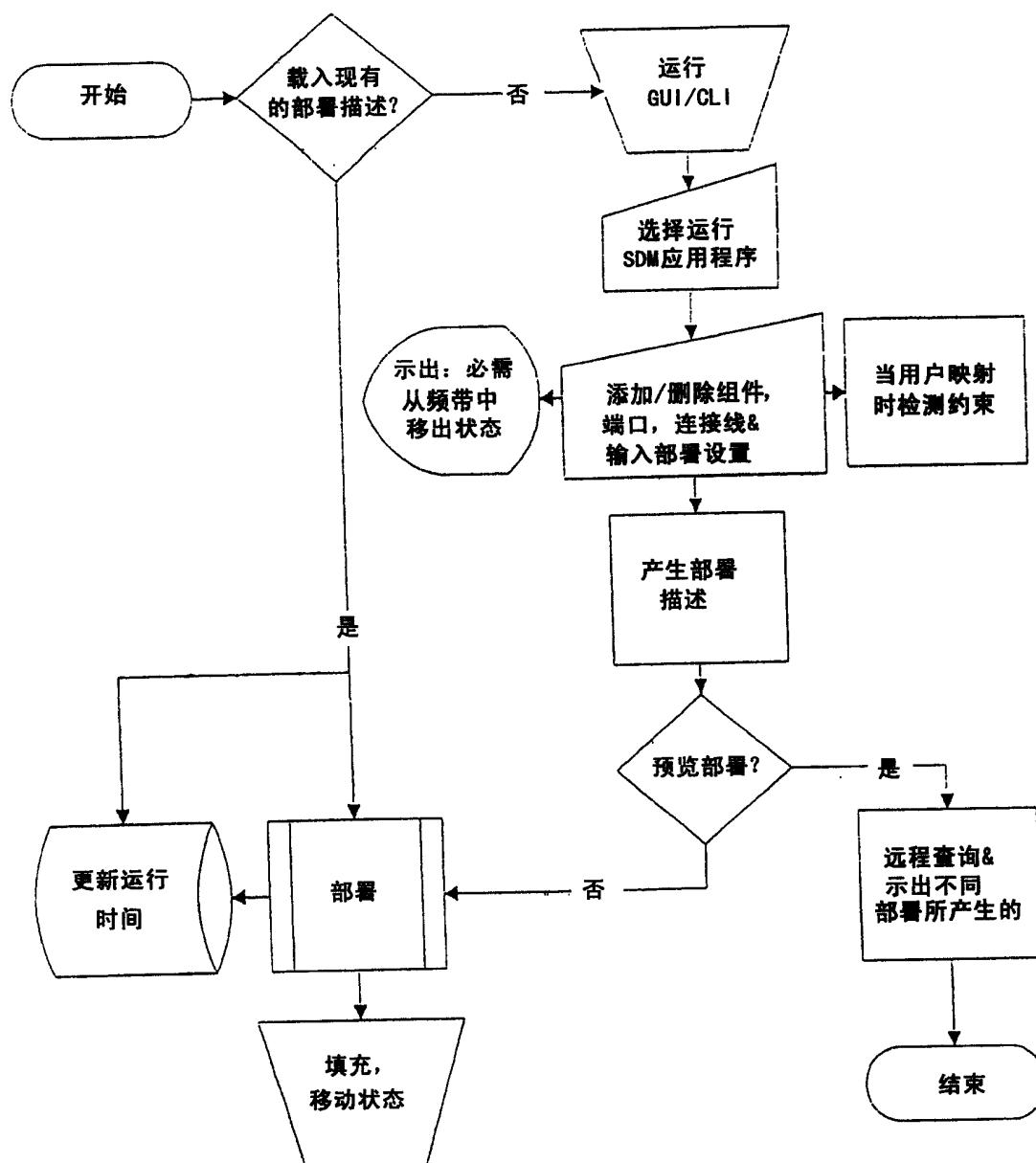


图 56

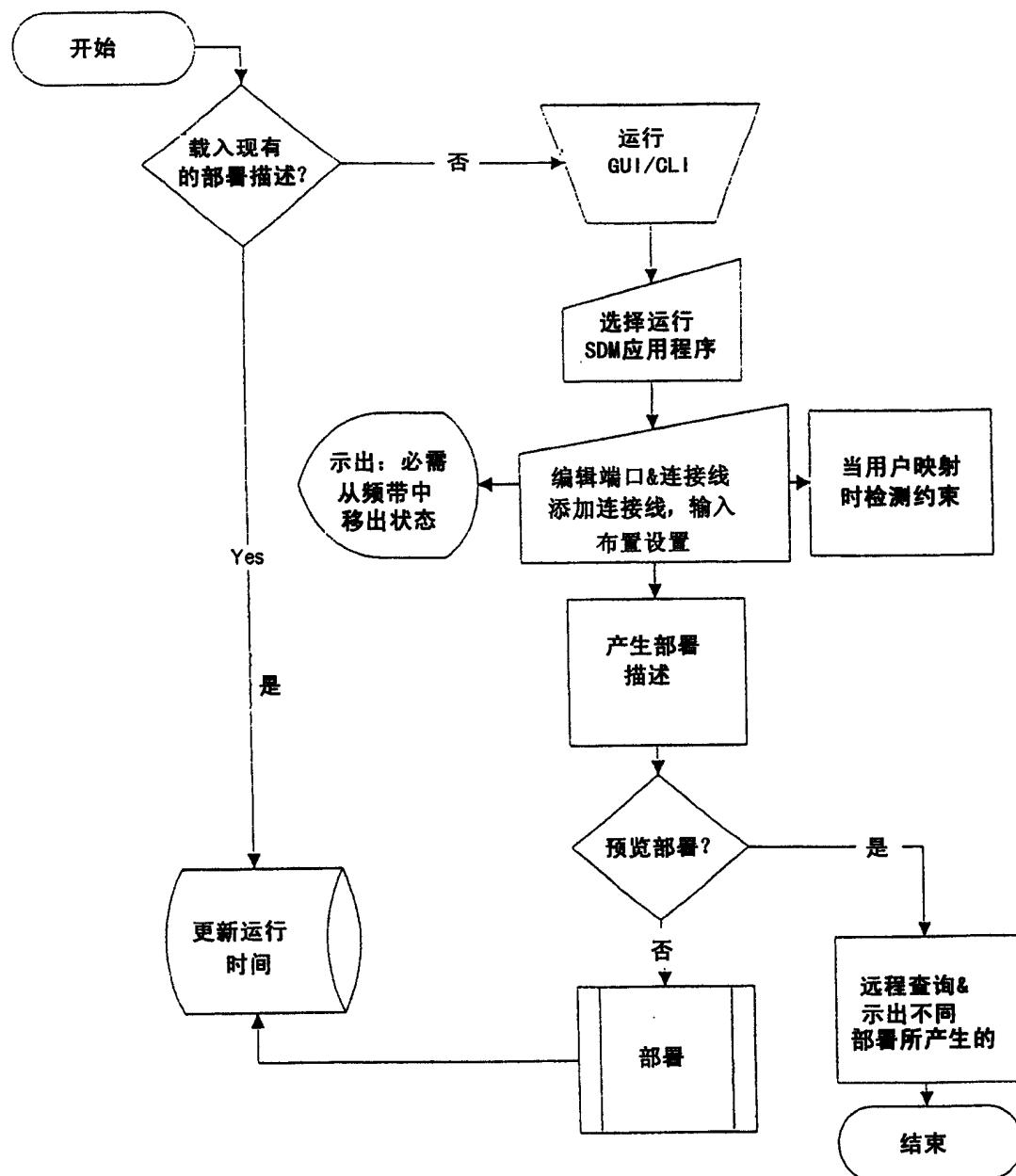


图 57

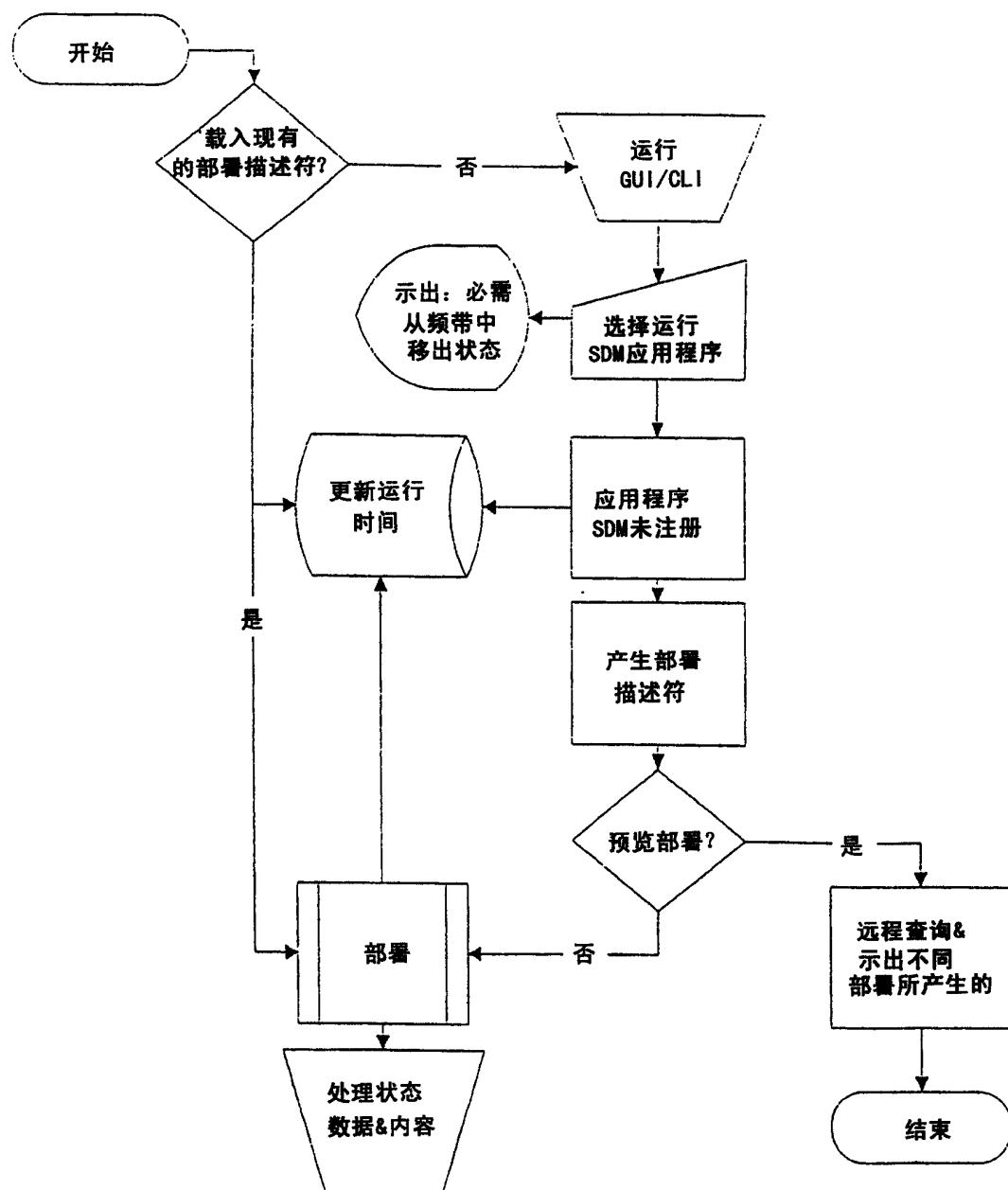
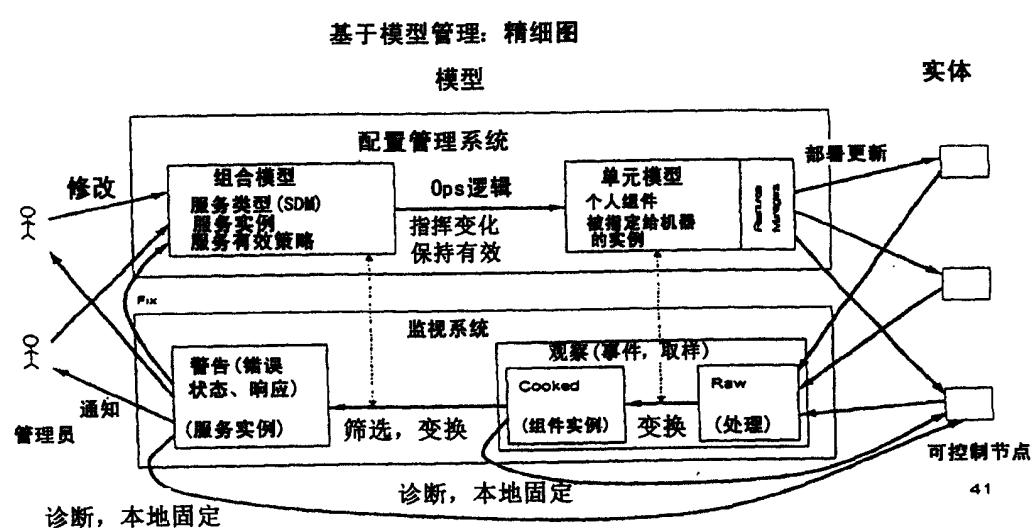


图 58

**图 59**

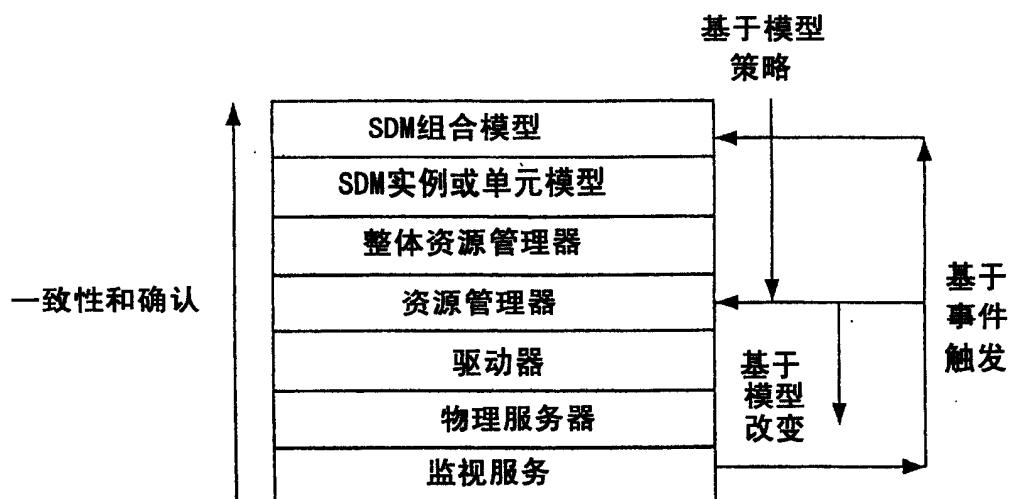


图 60

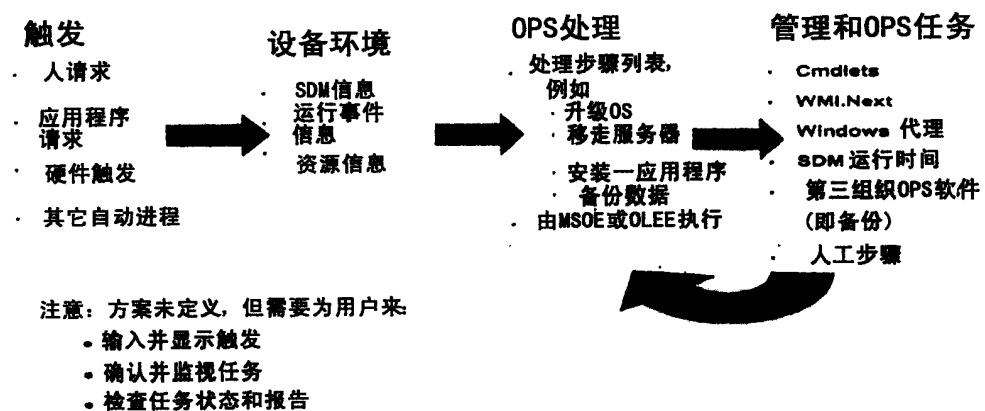


图 61

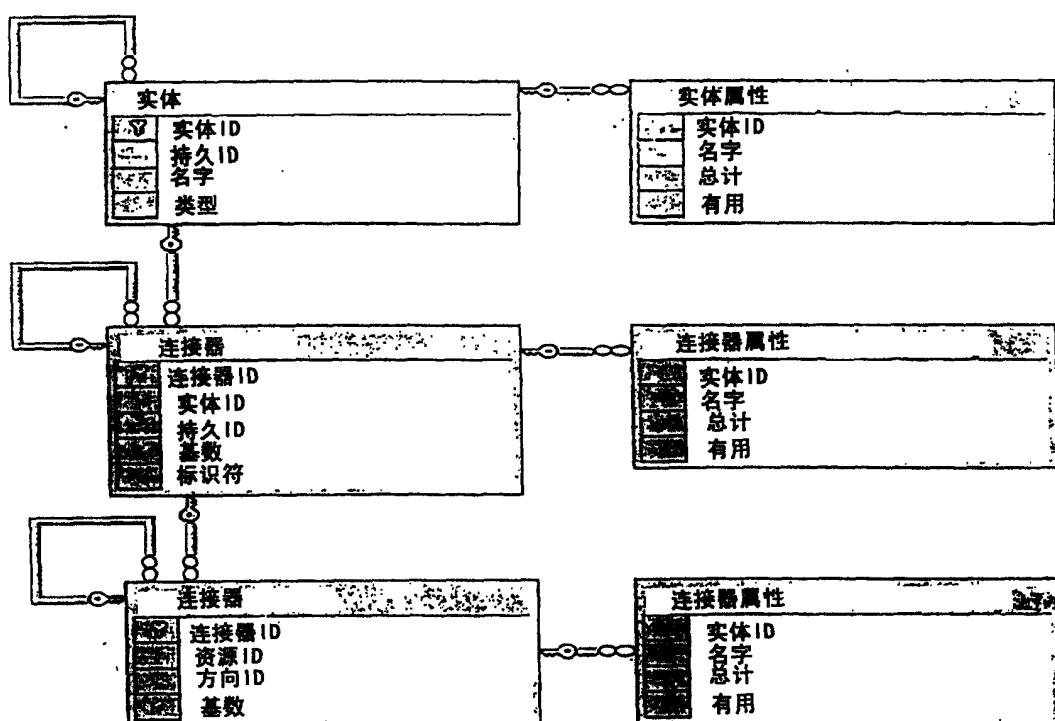


图 62

Room 42/4814,
Power Grid 4800



图 63

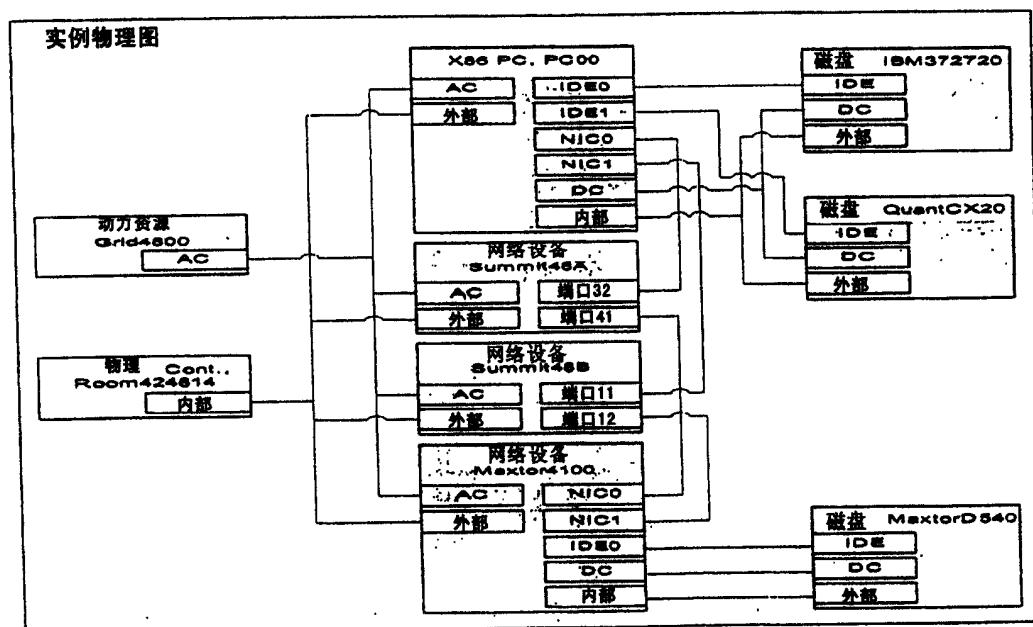


图 64

ID	Sys	Qty	Dual	类别	名称	驱动器标识符	唯一的标识符	Cardin	Notes
1				Power Source	Grid4800				
2				Physical Container	Room424814				
3				X86 PC	PC00				
4				Disk	IBM372720				
5				Disk	QuantaCX20				
6				Network Device	SummiH48A				
7				Network Device	SummiH5B				
8				Network Device	Mextor4100				
9				Disk	MextorD540X				
10	1			Power	AC Outlets				
11	2			Physical	Inside				
12	3			ATA	IDEO				
13	3			ATA	IDE1				
14	3			Ethernet	NIC0		mac:00-B0-D0-20-3F-32		
15	3			Ethernet	NIC1		mac:00-A0-C9-A0-06-06		
16	3			Power	DC Connector				
17	3			Power	AC Connector				
18	3			Physical	Outside				
19	3			Physical	Inside				
20	4			ATA	Port				
21	4			Power	DC Connector				
22	4			Physical	Outside				
23	5			ATA	Port				
24	5			Power	DC Connector				
25	5			Physical	Outside				
26	6			Ethernet	Port 32				
27	6			Ethernet	Port 41				
28	6			Power	AC Connector				
29	6			Physical	Outside				
30	7			Ethernet	Port 11				
31	7			Ethernet	Port 12				
32	7			Power	AC Connector				
33	7			Physical	Outside				
34	8			Ethernet	NIC0		mac:00-A0-29-FE-CA-20		
35	8			Ethernet	NIC1		mac:00-A0-29-FE-CA-21		
36	8			ATA	IDEO				
37	8			Power	DC Connector				
38	8			Power	AC Connector				
39	8			Physical	Outside				
40	8			Physical	Inside				
41	9			ATA	Port				
42	9			Power	DC Connector				
43	9			Physical	Outside				
44	26	14	45	Ethernet	Wire0				
45	30	15	44	Ethernet	Wire1				
46	10	17		Power	Cord0				
47	11	18		Physical	Contained				
48	12	20		ATA	Cable				
49	16	21		Power	DC Cable				
50	19	22		Physical	Internal				
51	13	23		ATA	Cable				
52	16	24		Power	DC Cable				
53	19	25		Physical	Internal				
54	27	34	55	Ethernet	Wire2				
55	31	35	54	Ethernet	Wire3				
56	10	36		Power	Cord0				
57	11	39		Physical	Contained				
58	36	41		ATA	Cable				
59	37	42		Power	Cable				
60	40	43		Physical	Internal				

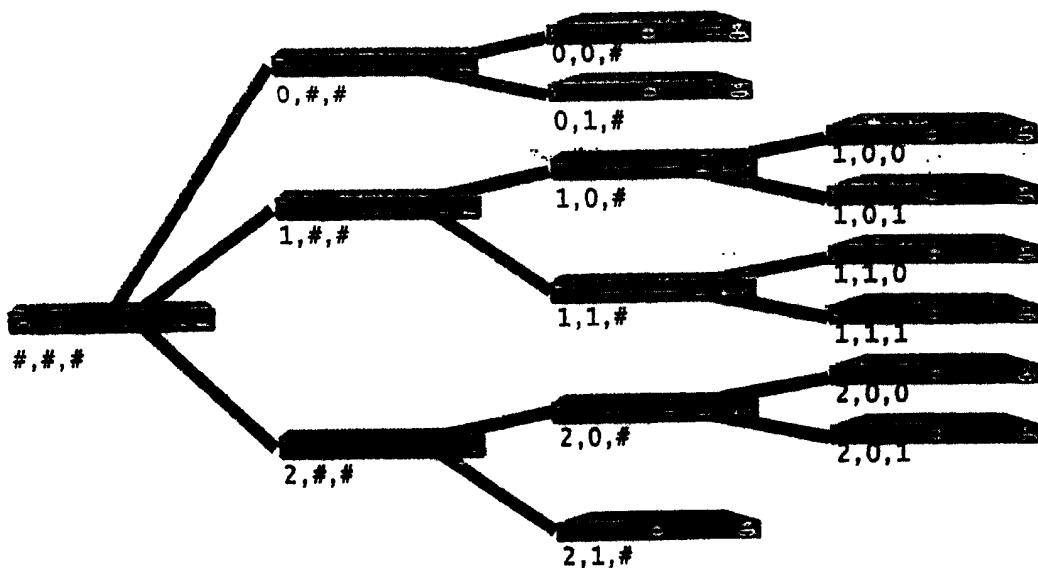


图 66

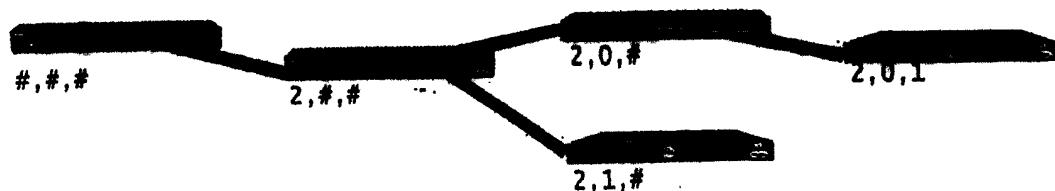


图 67

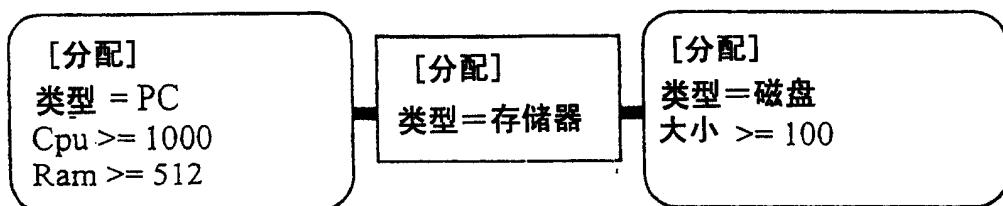


图 68

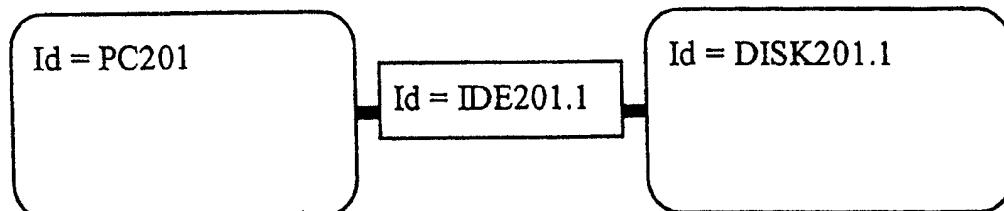


图 69

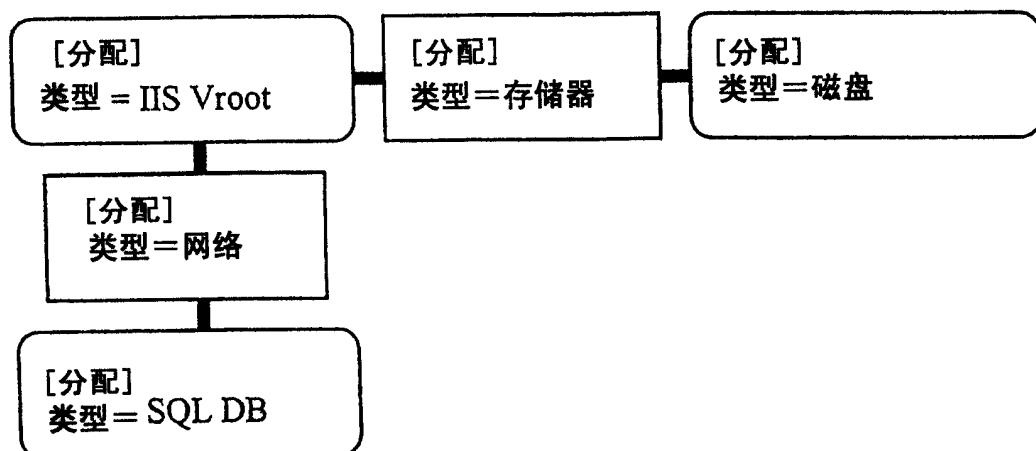


图 70

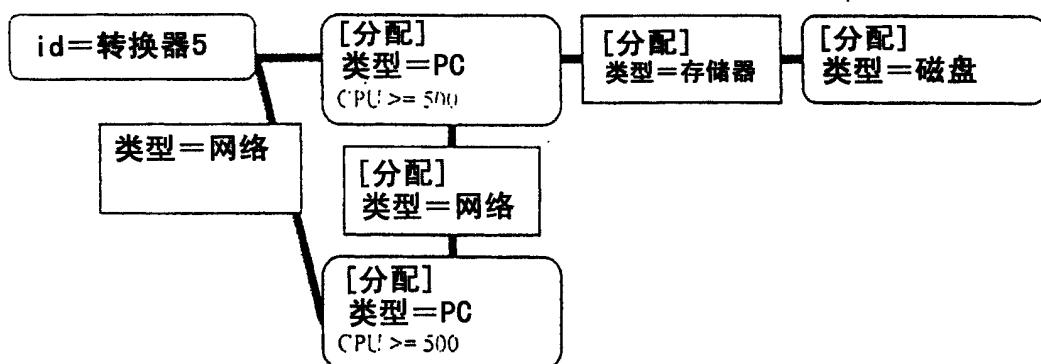


图 71

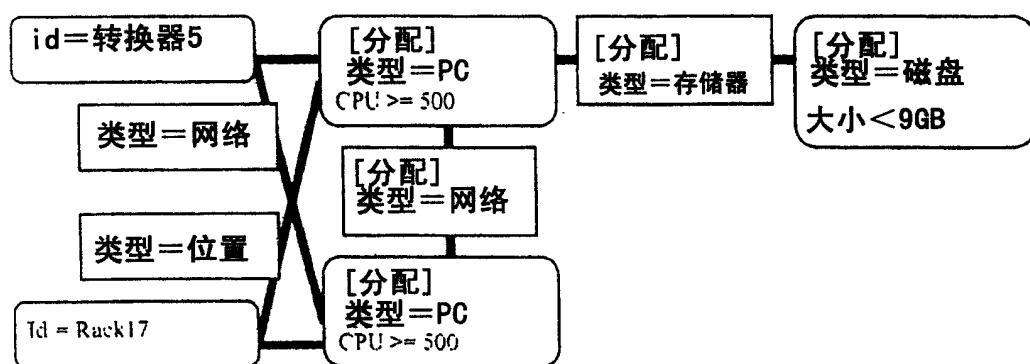
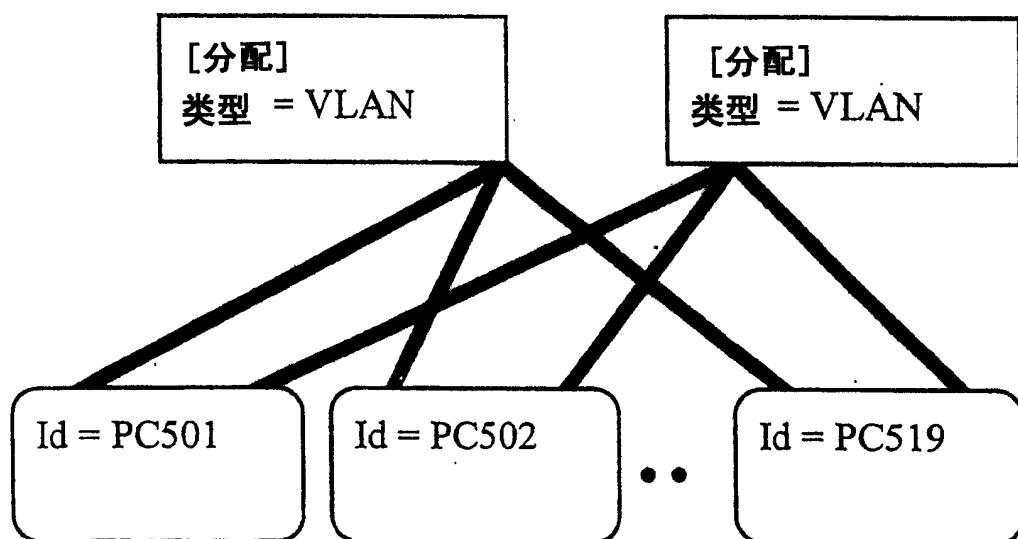
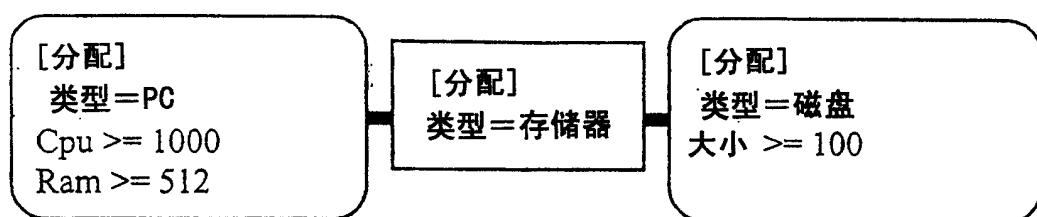


图 72



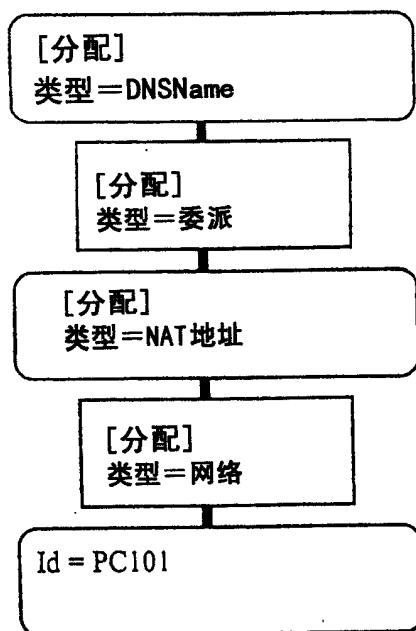


图 75

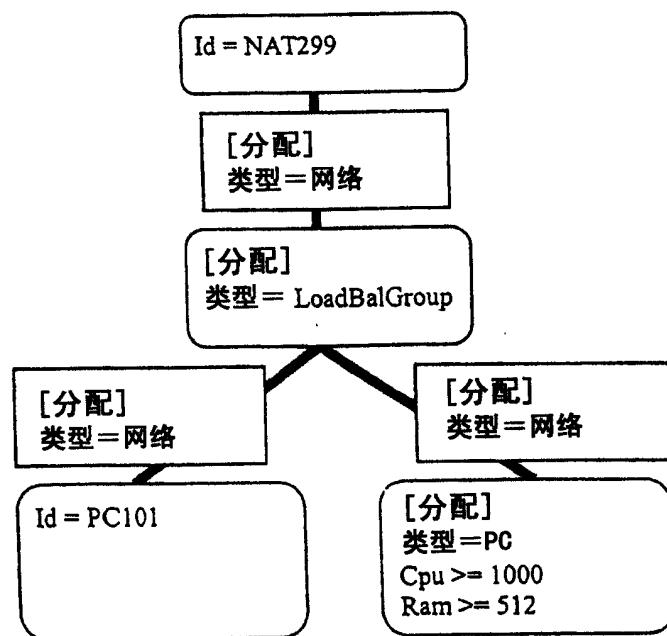


图 76

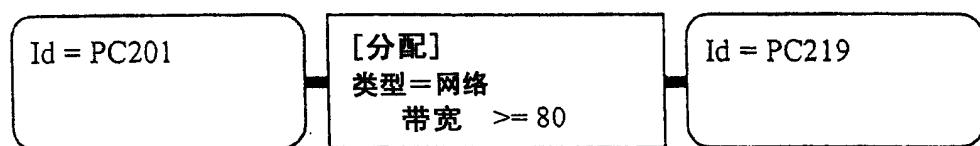


图 77

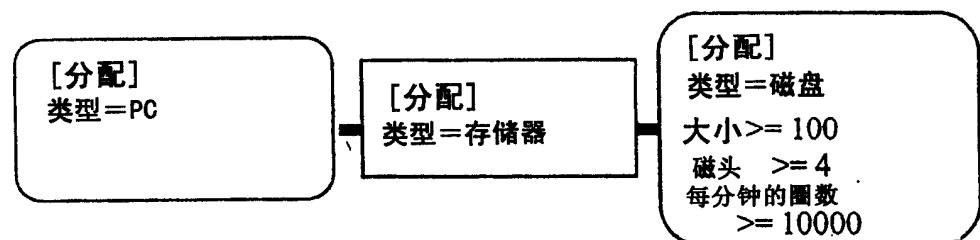


图 78

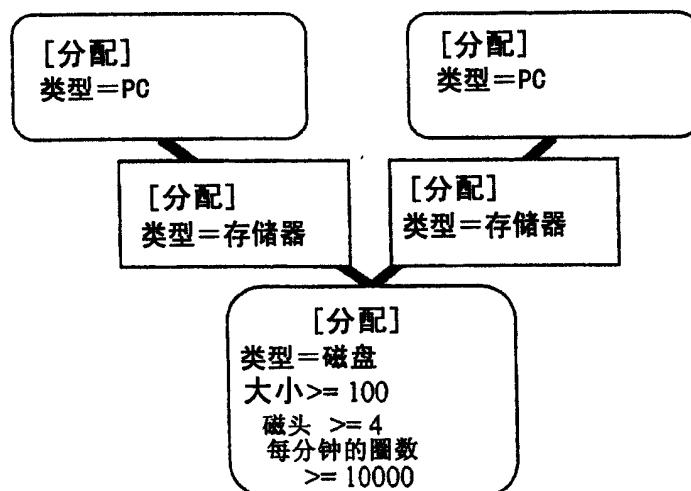


图 79

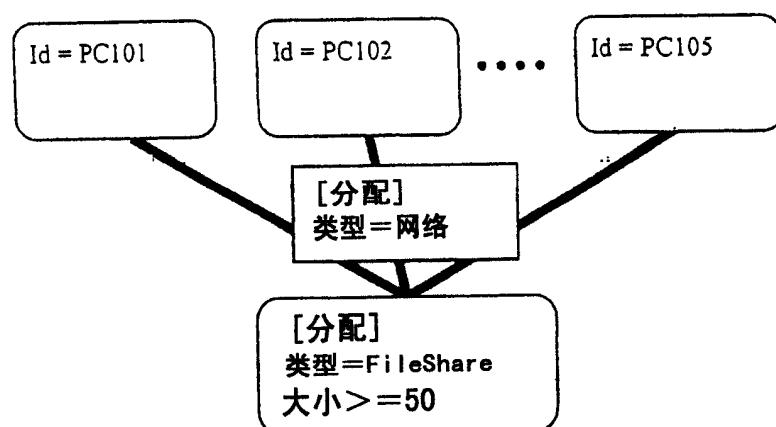


图 80

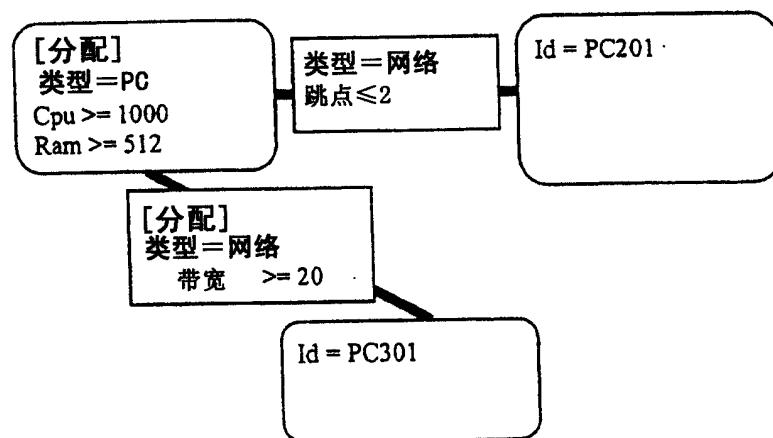


图 81

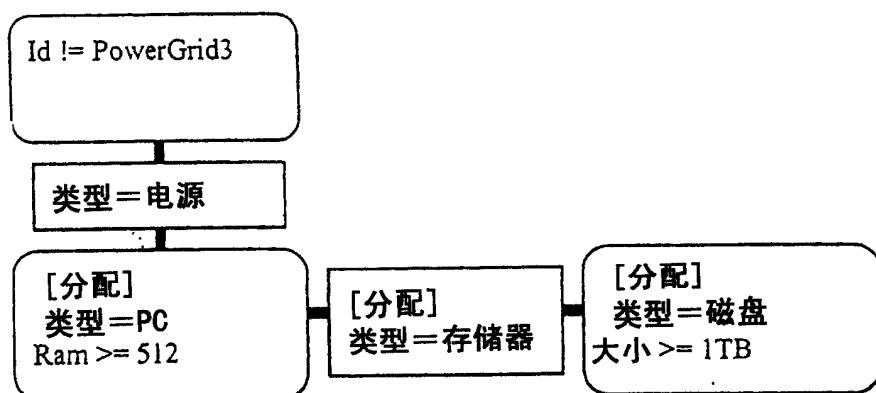


图 82

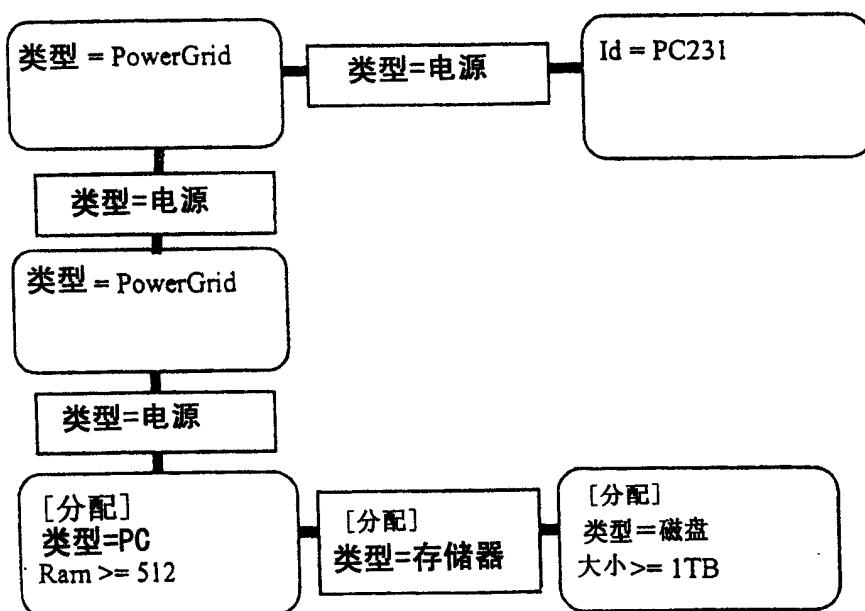


图 83

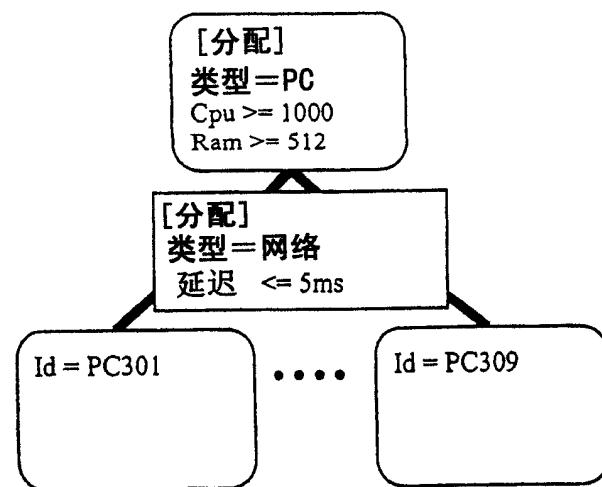


图 84

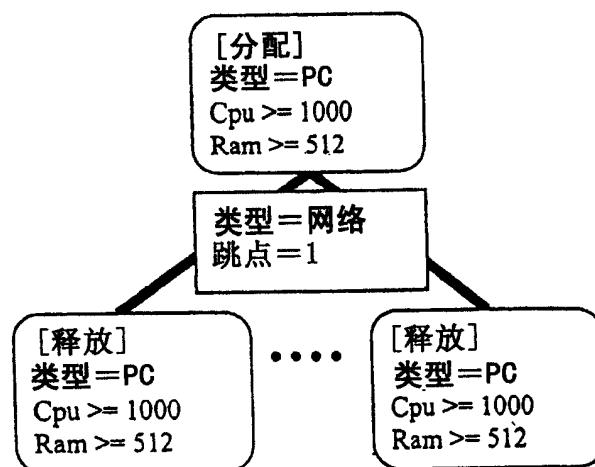


图 85

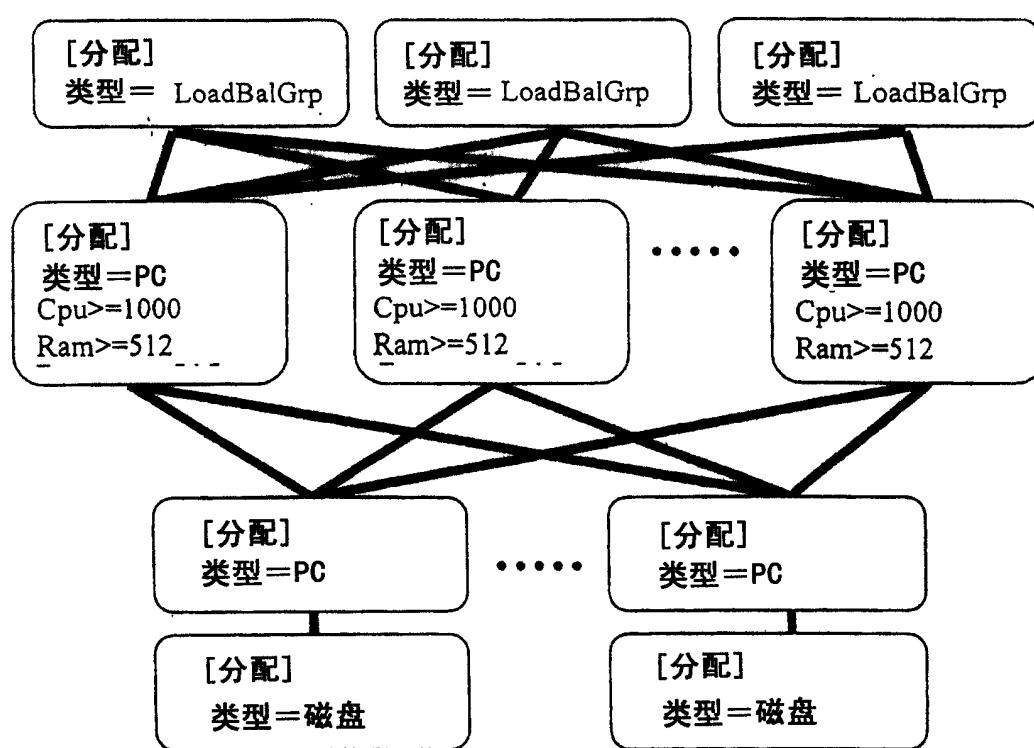
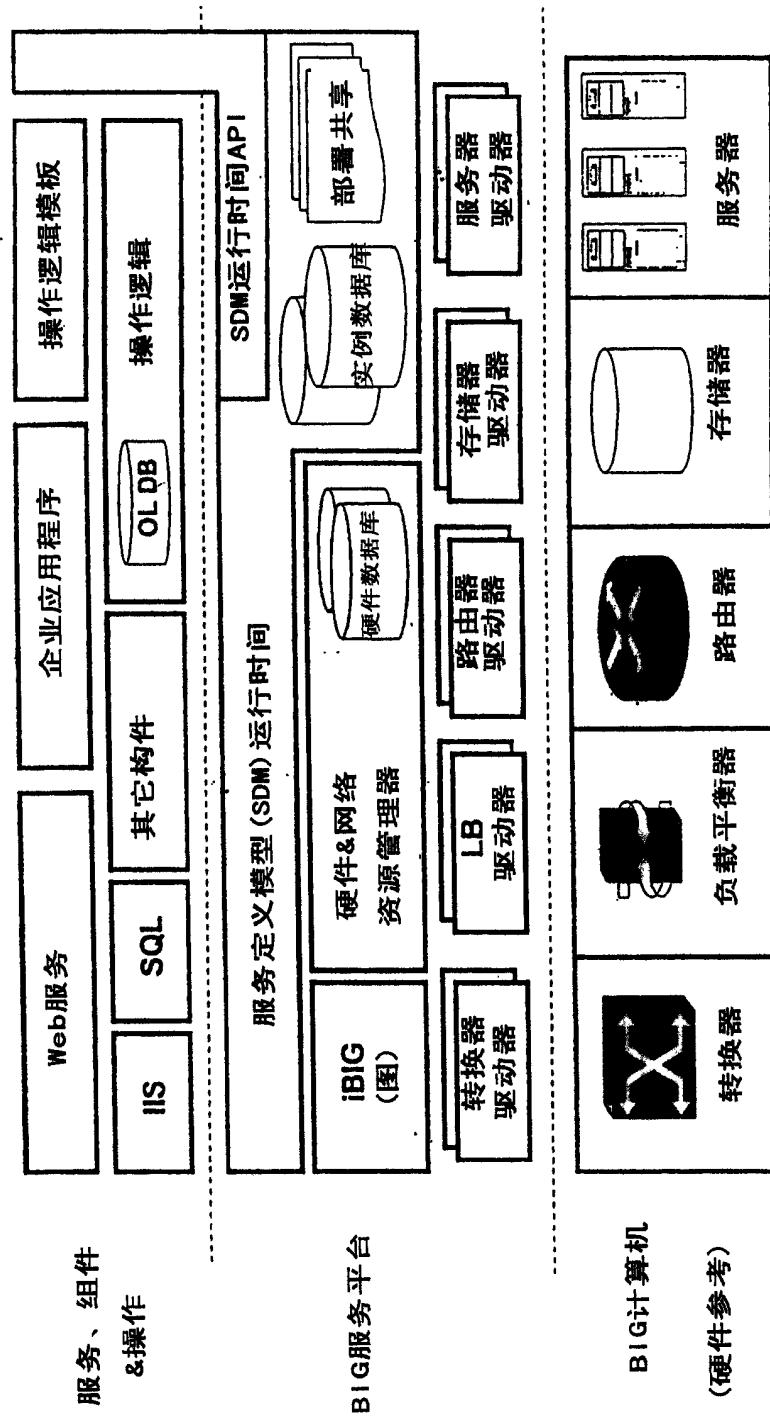


图 86



参 87

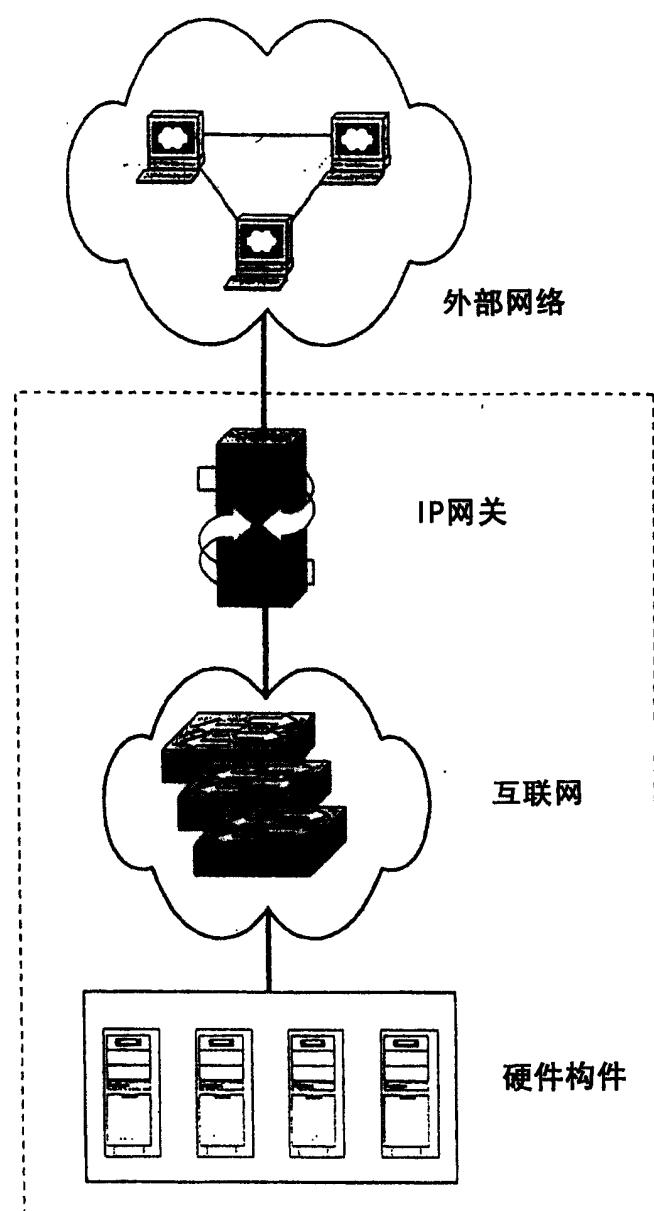
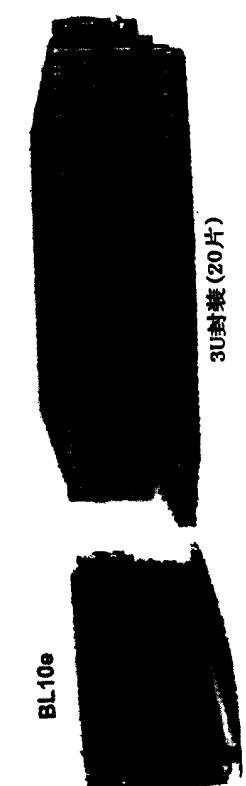
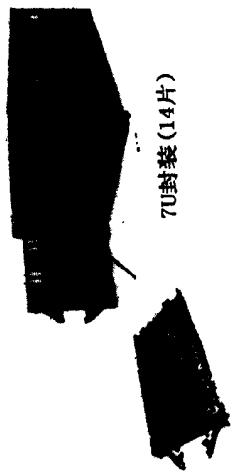


图 88

**HPQ Proliant BL e-class**

- Pentium III 700MHz
- 512MB - 1GB ECC RAM
- 30GB ATA 硬盘
- Dual 10/100 快速以太网
- Layer 2 switch, (4) 吉位上行链路
- 冗余600W电源
- 每42U架280片, 每槽25W

**IBM BladeCenter**

- 双Xeon
- 8GB ECC RAM
- iSCSI 或光纤通道存储器
- (4) 吉位以太网
- (4) 1200 W 电源
- 每42U架98片

Dell PowerEdge 1655MC

- 奔腾III 1.2GHz
- 128MB - 2GB ECC RAM
- 36-146GB SCSI 硬盘
- 倍速以太网
- (2) 层2开关, (4) 吉位上行链路
- 冗余1040W电源
- 每42U架84片



3U封装(6片)

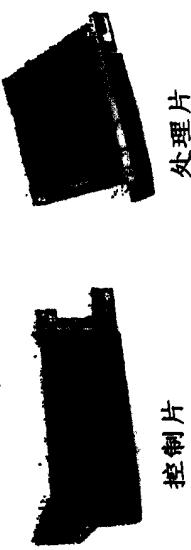
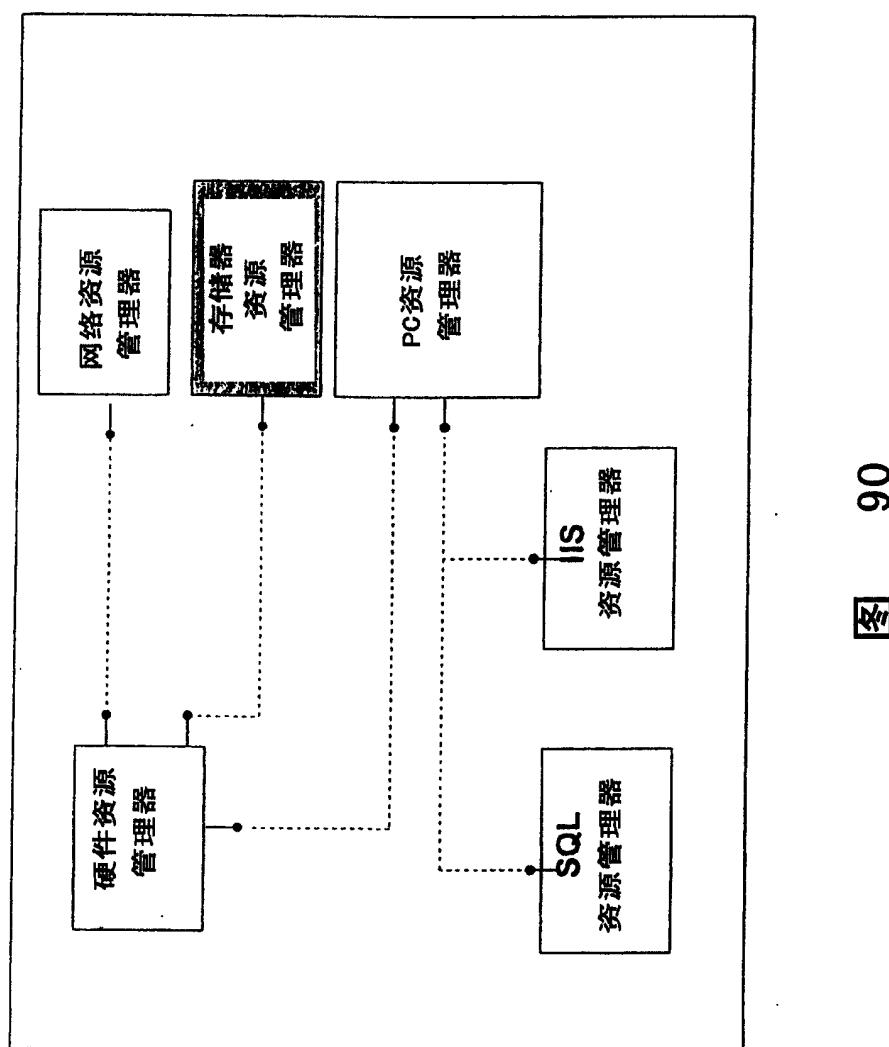


图 89



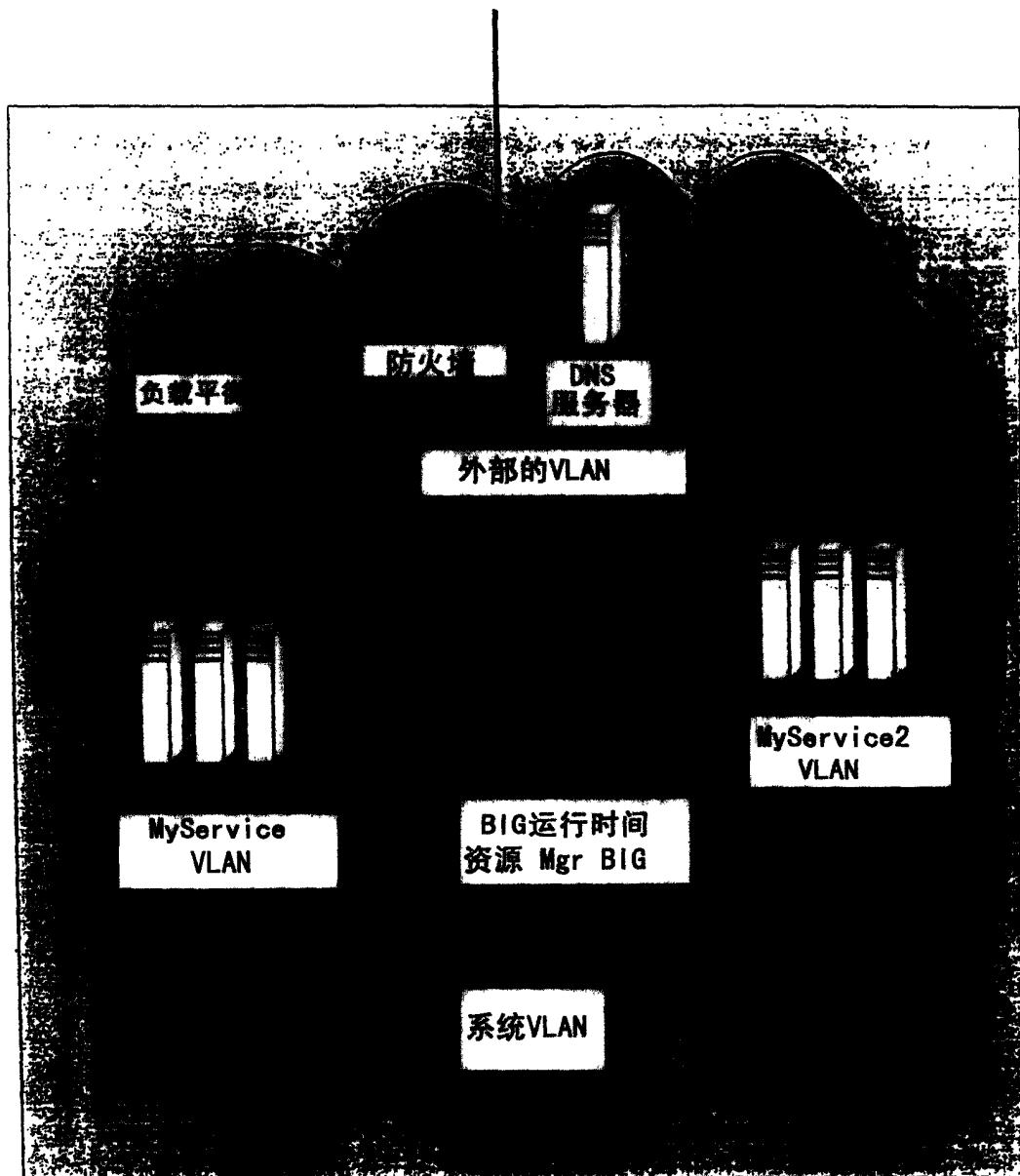


图 91

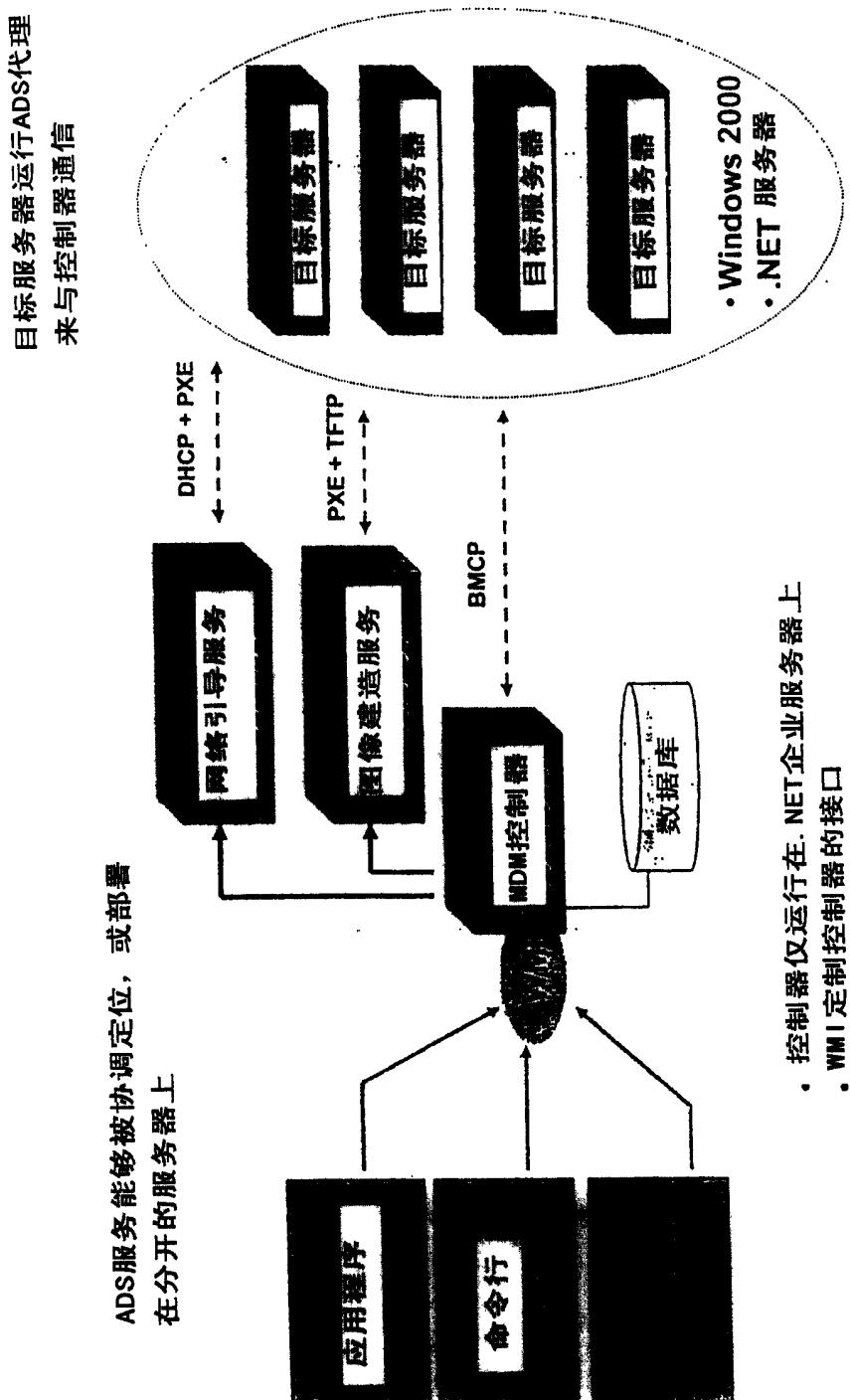


图 92

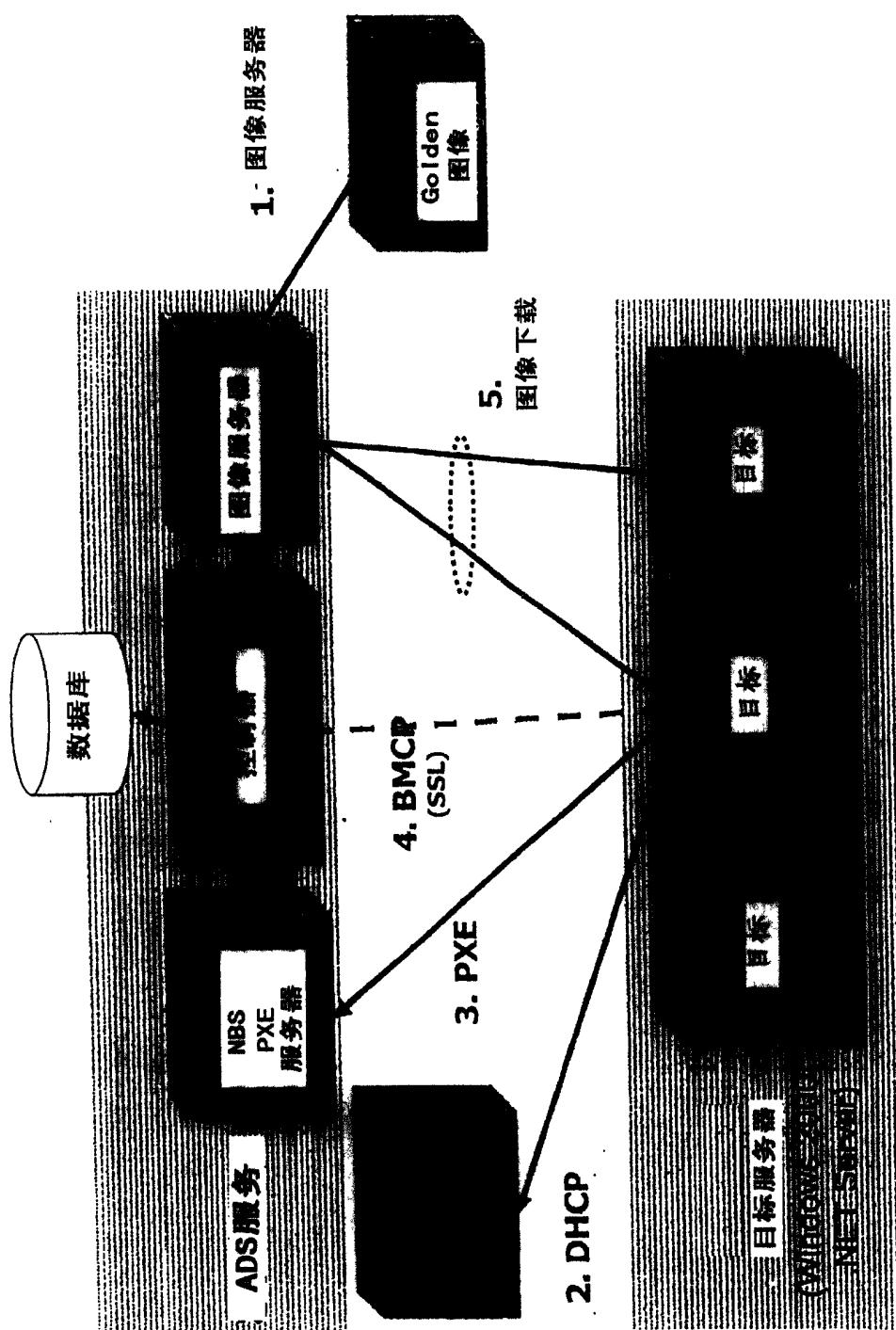


图 93

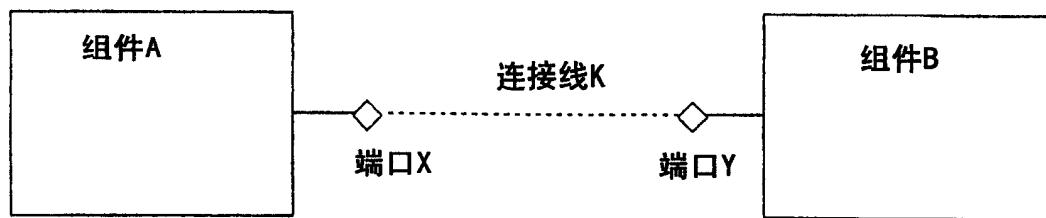
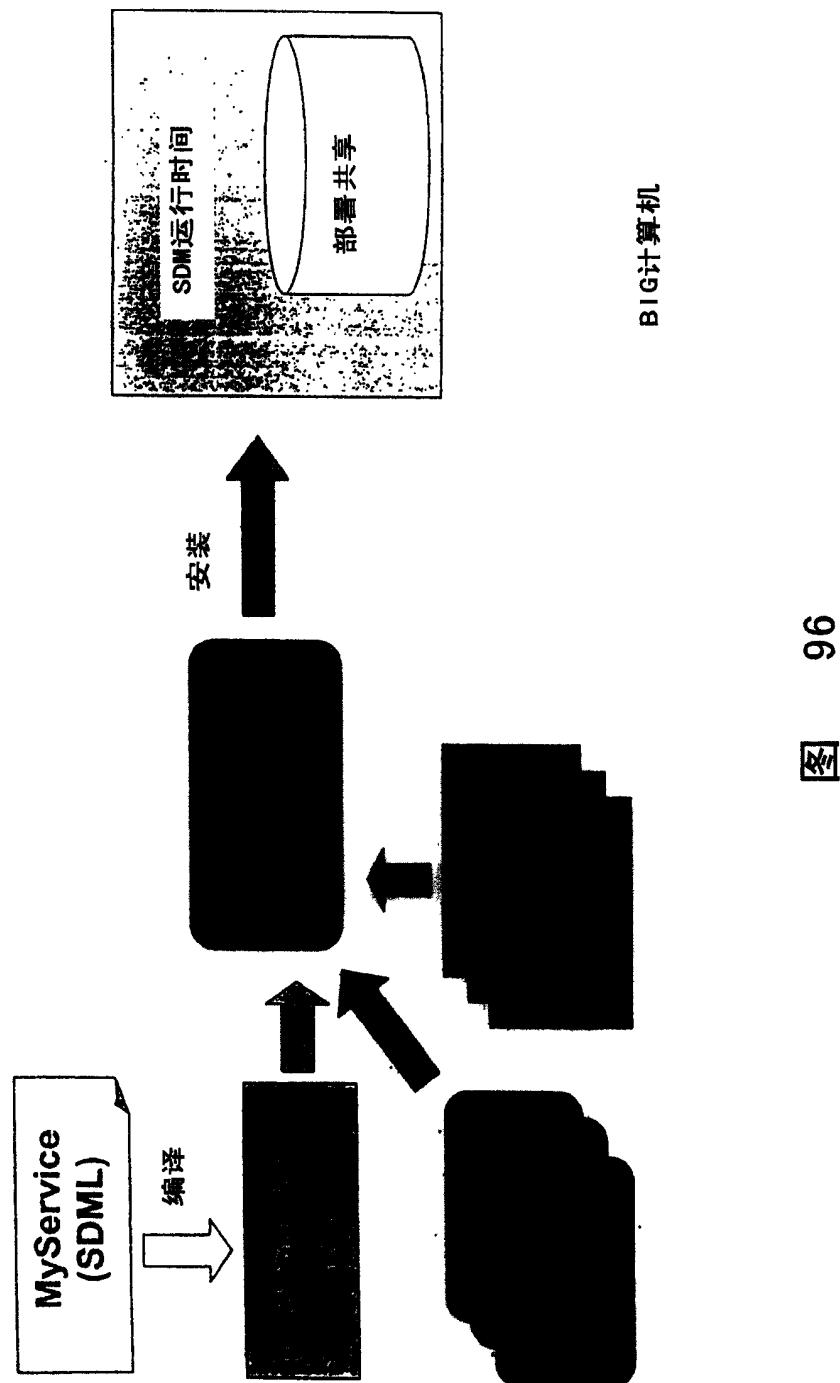


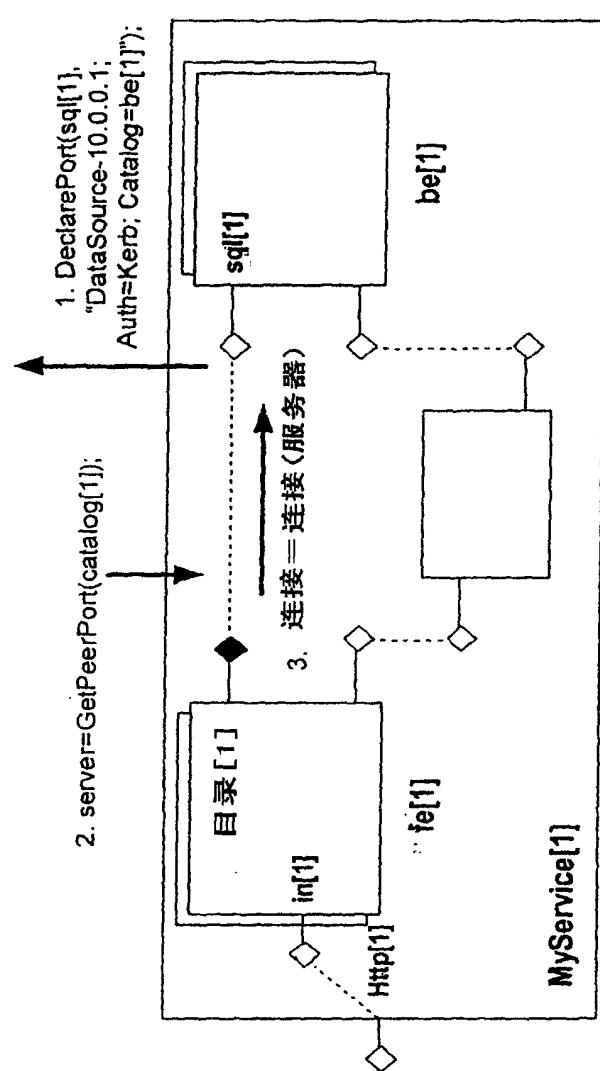
图 94

使用系统
使用系统. SQL
使用系统. IIS
组类名字MyService

```
componenttype MyService
{
    component MyFrontEnd fe;
    component MyBackEnd be;
    port http = fe.http;
    writerDBS tds {
        fe.catalog;
        be.sql;
    }
}

componenttype MyBackEnd :
SQLDatabase {
    implementation "MySQL", "MyCLRApp";
    implementation "MyService, MyCLRApp"
}
```





97

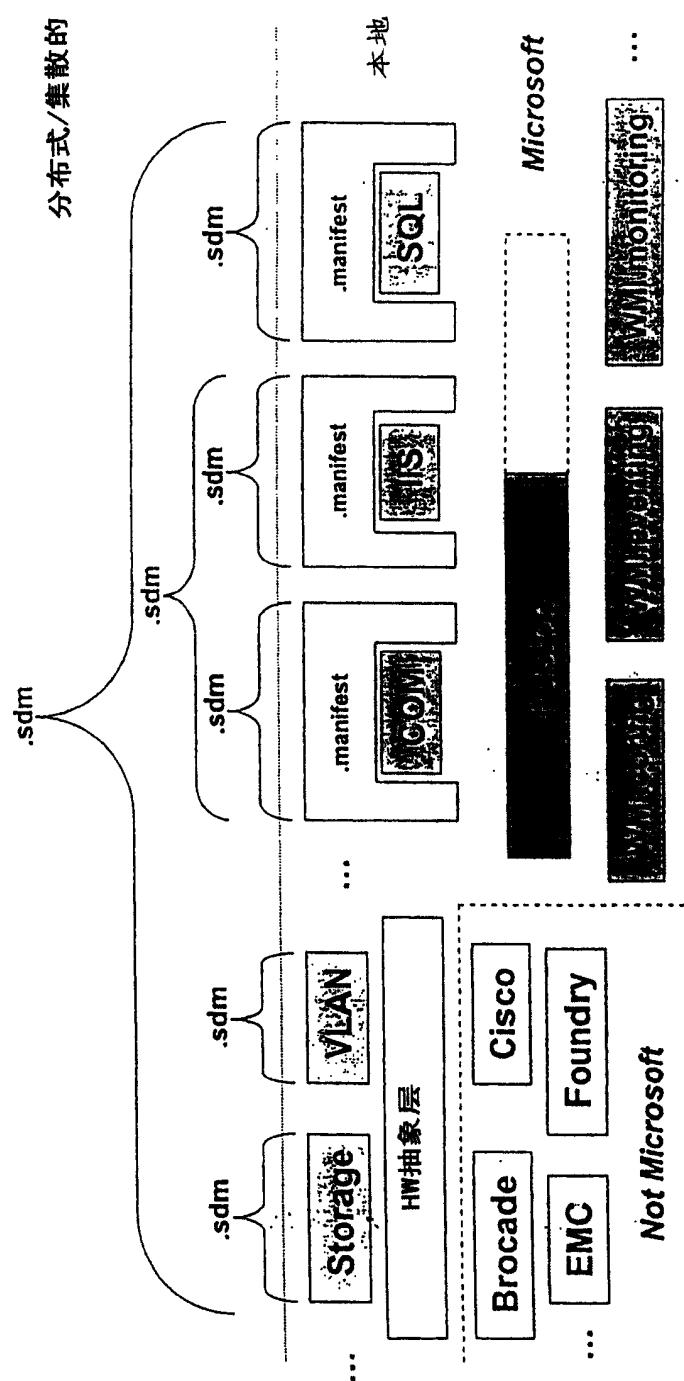


图 98

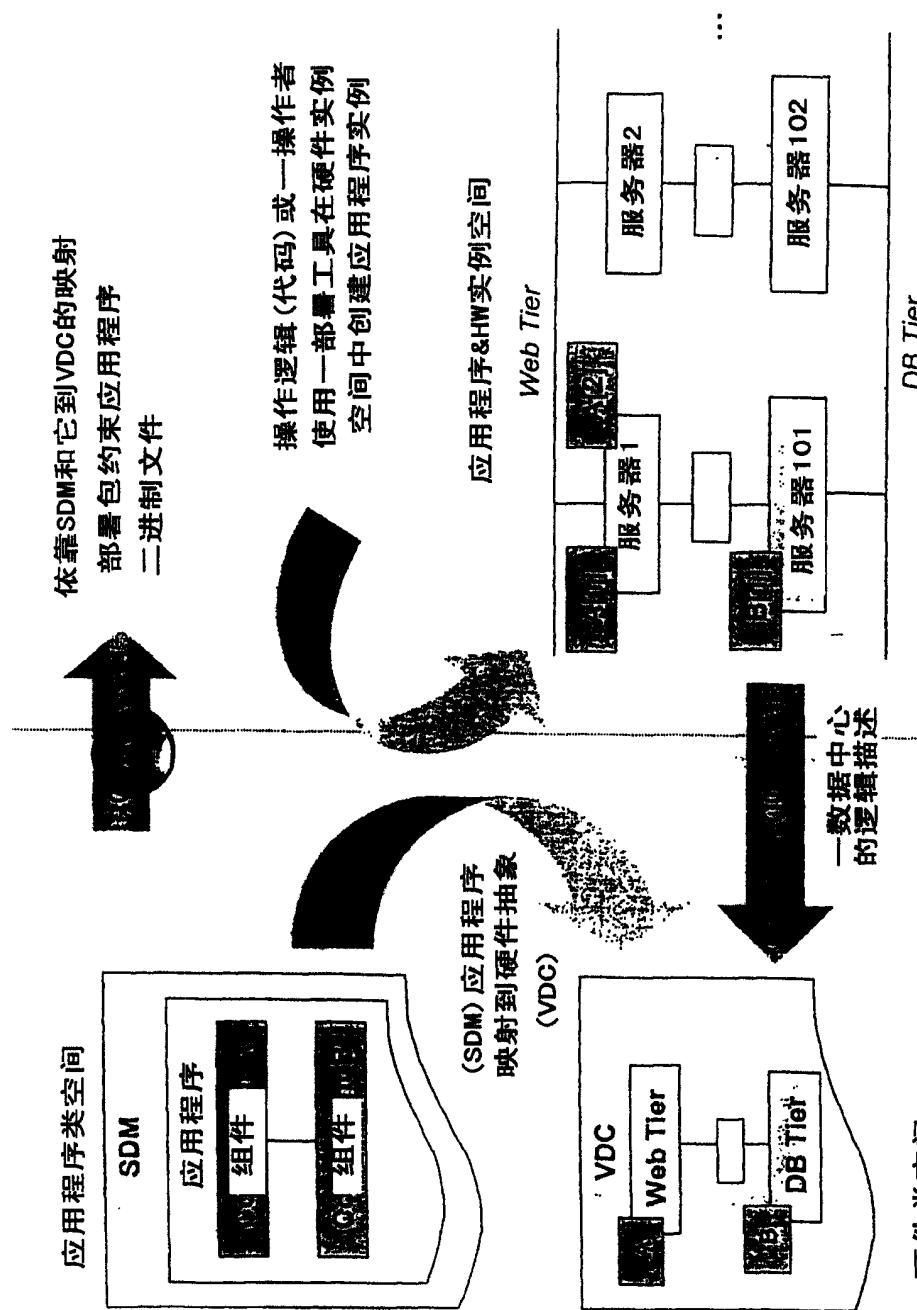


图 99

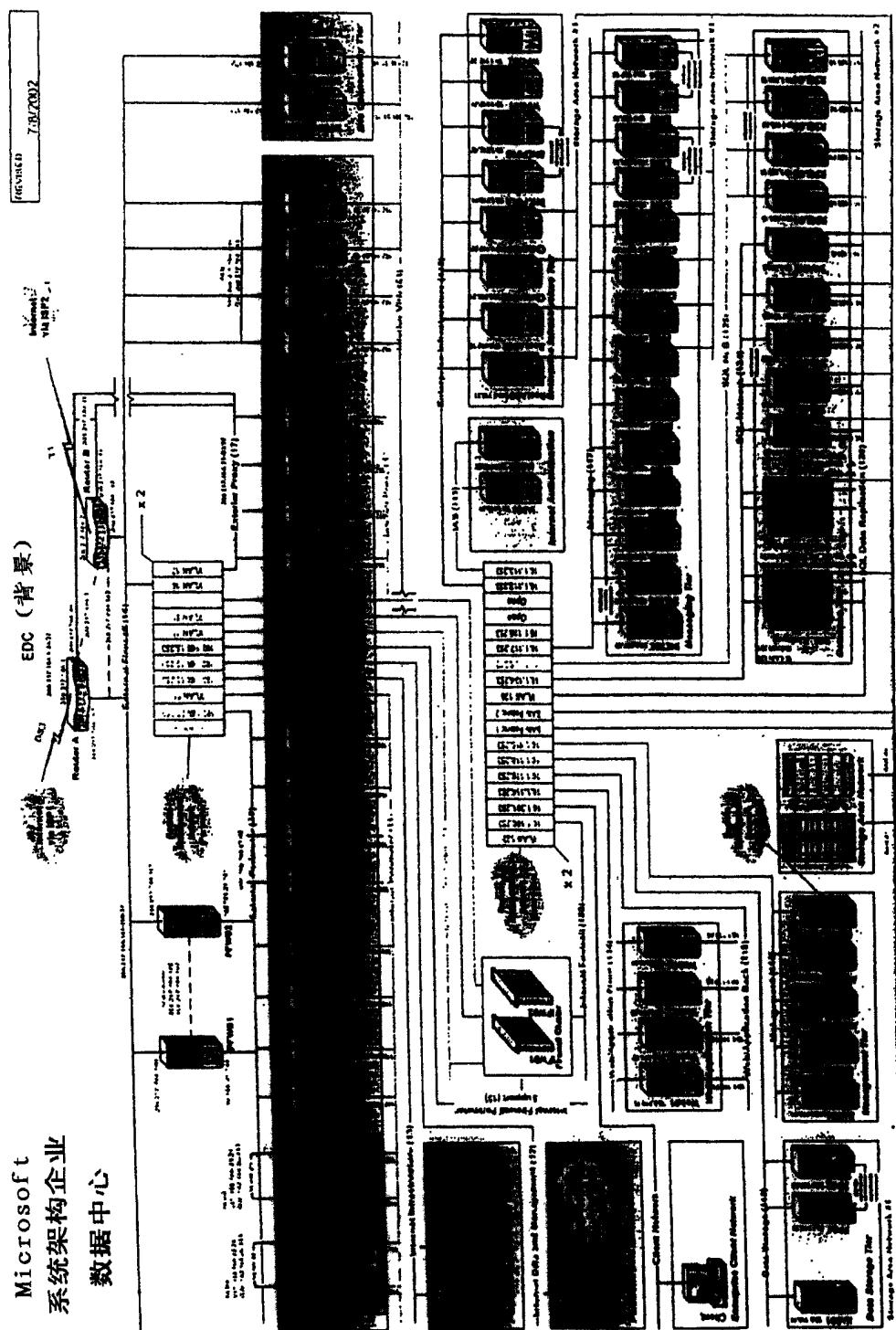
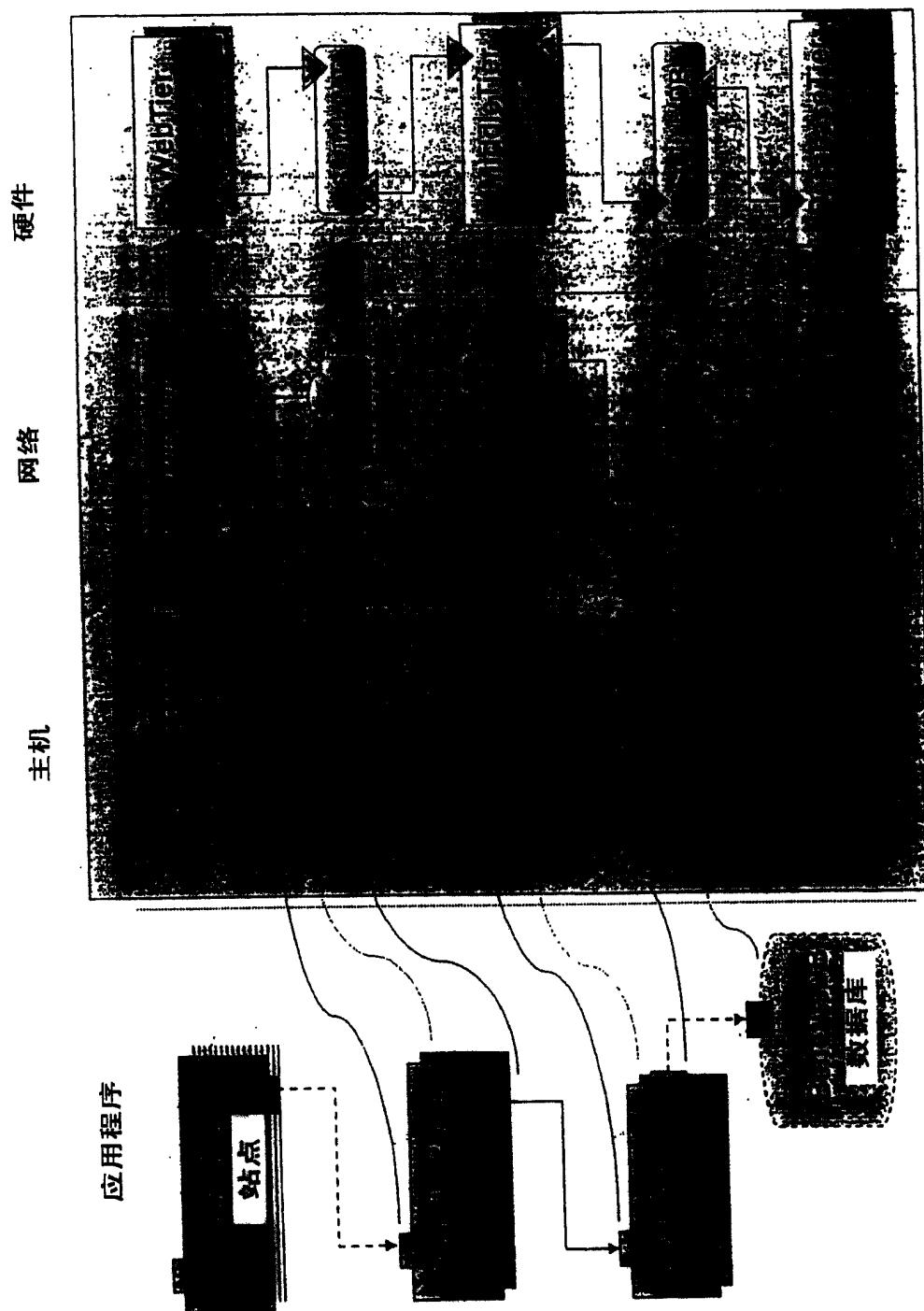
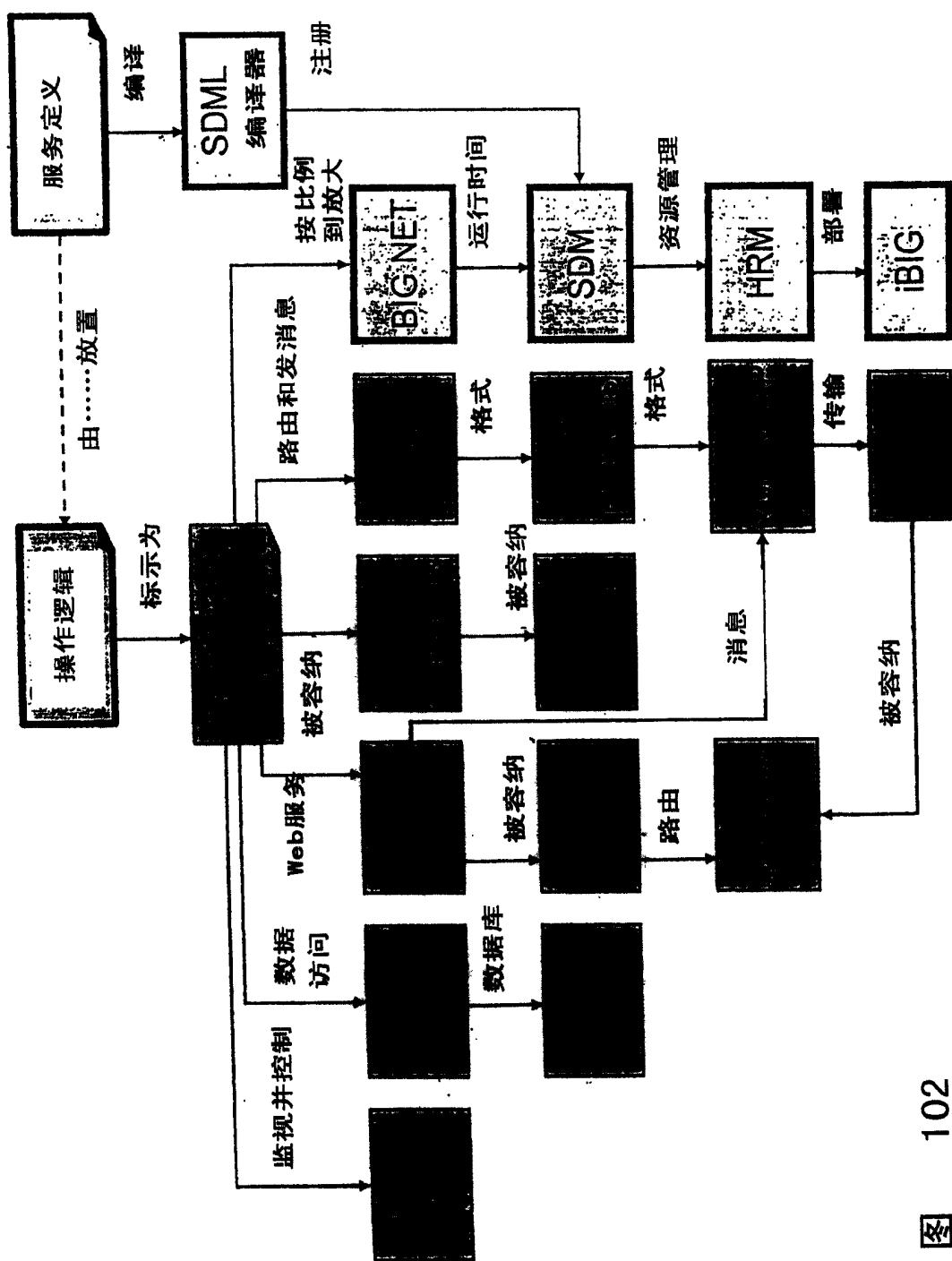


图 100



101



102
圖

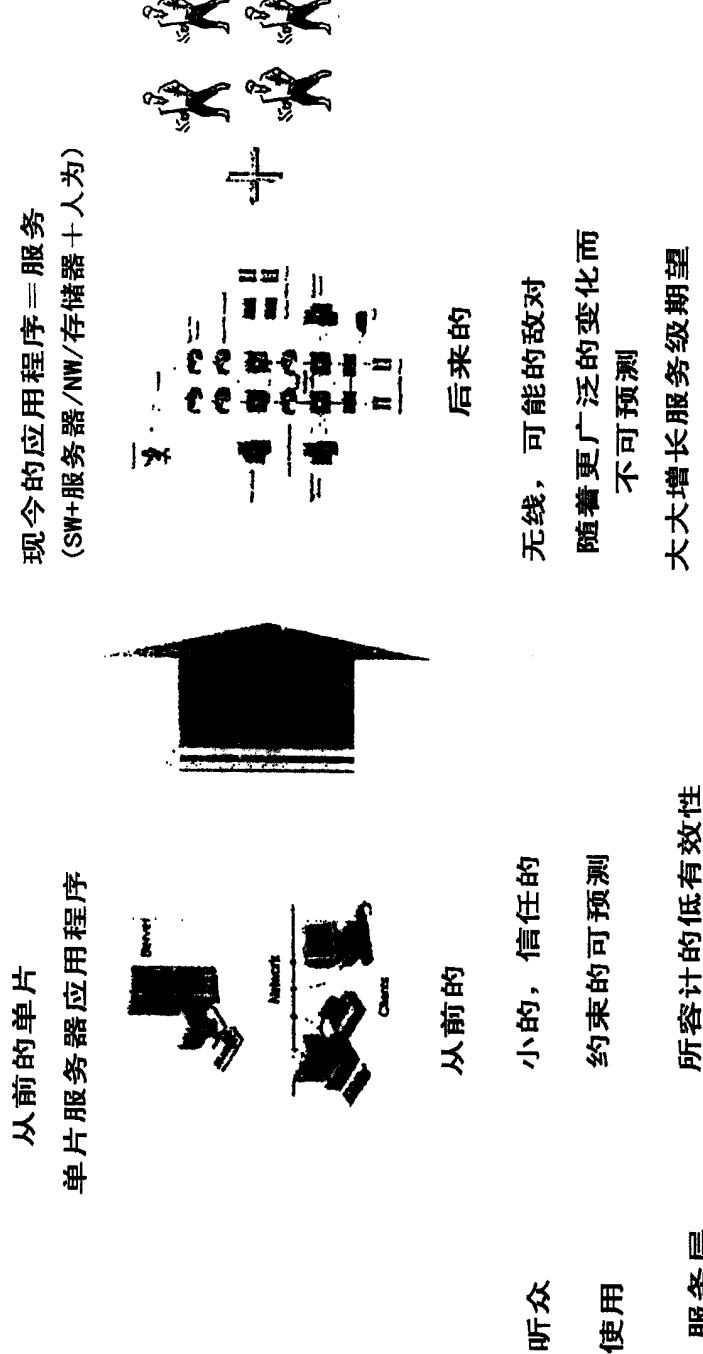
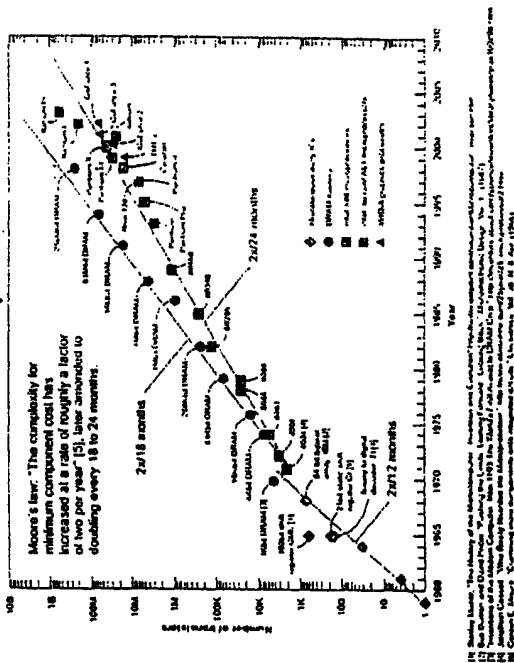
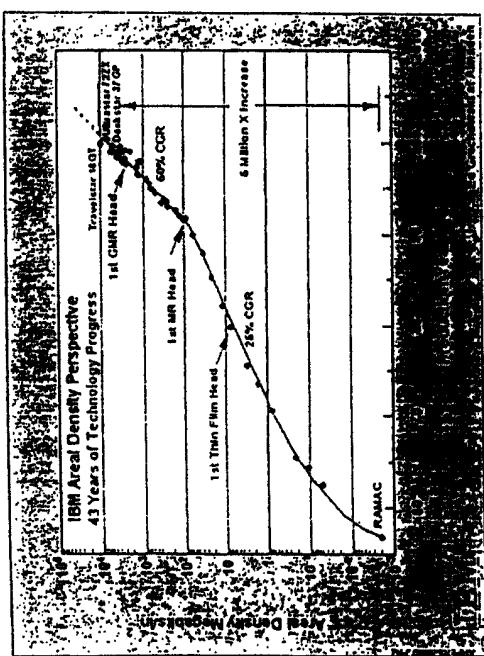


图 103

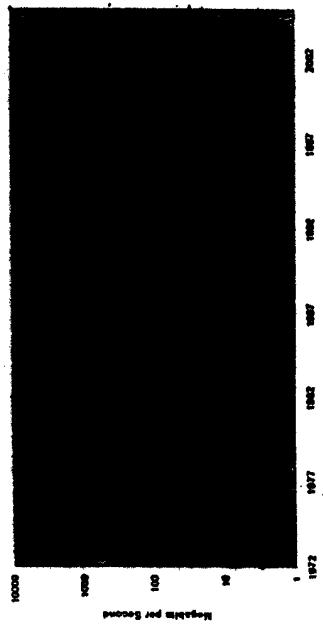


更有效CPU.....

图 104



密集硬盘.....



丰富叶片.....

成本的85%是由人驱动的，有效性和训练

TCO 服务 =人为+停机时间+训练+硬件/软件

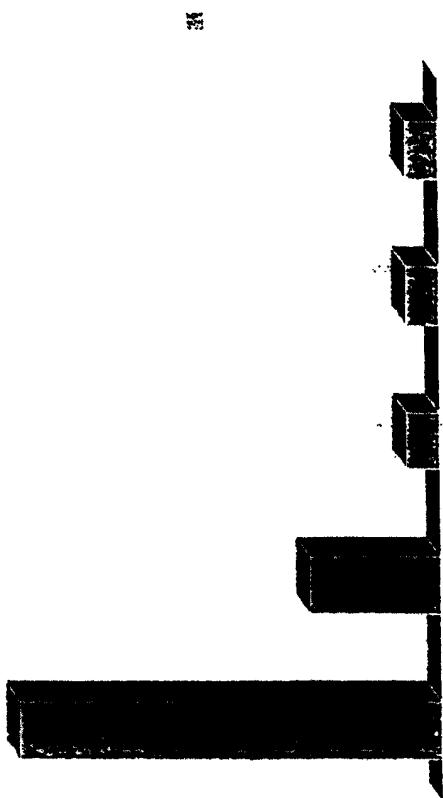


图 105

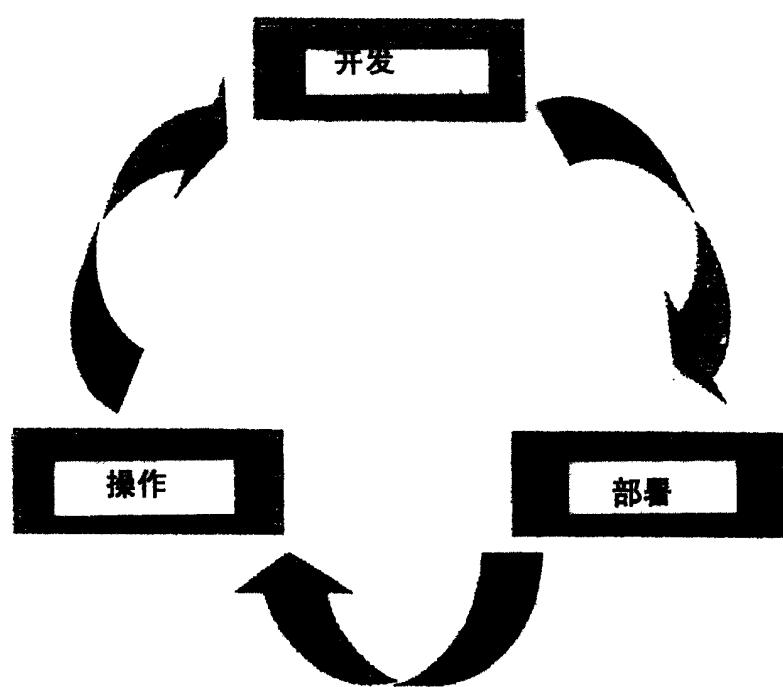


图 106

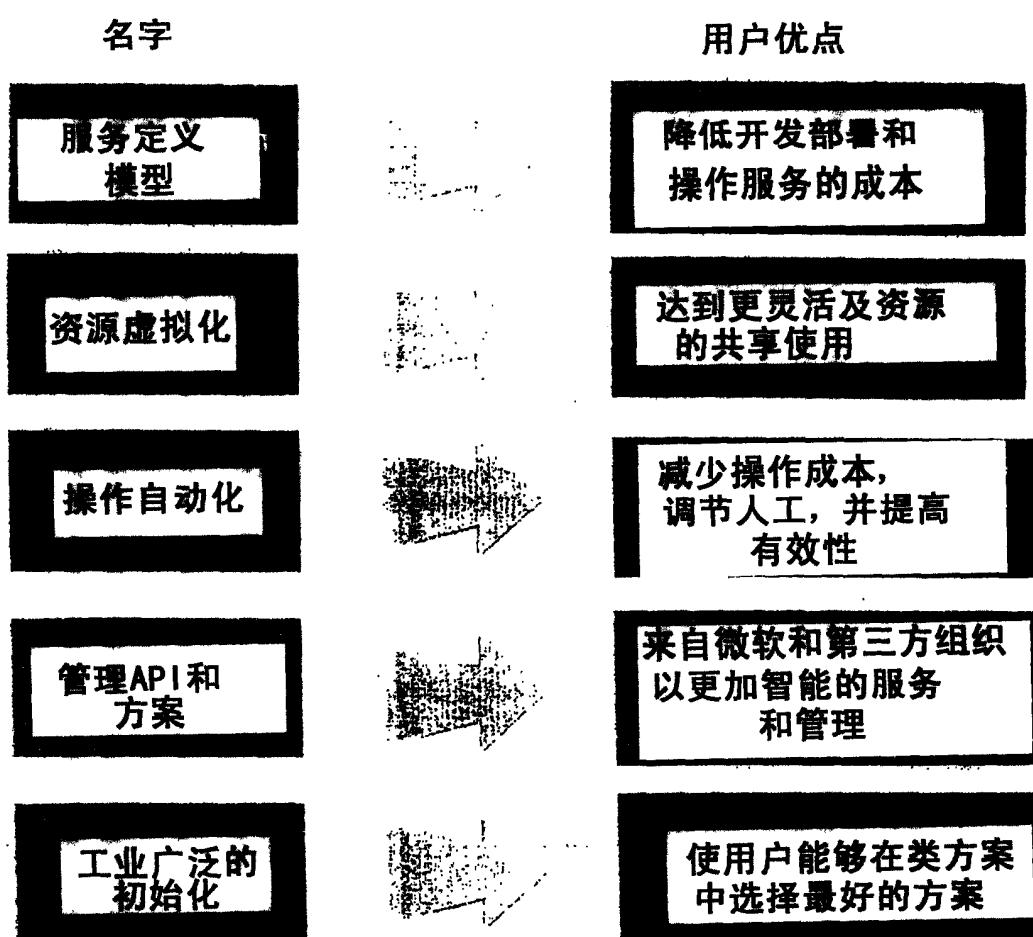


图 107

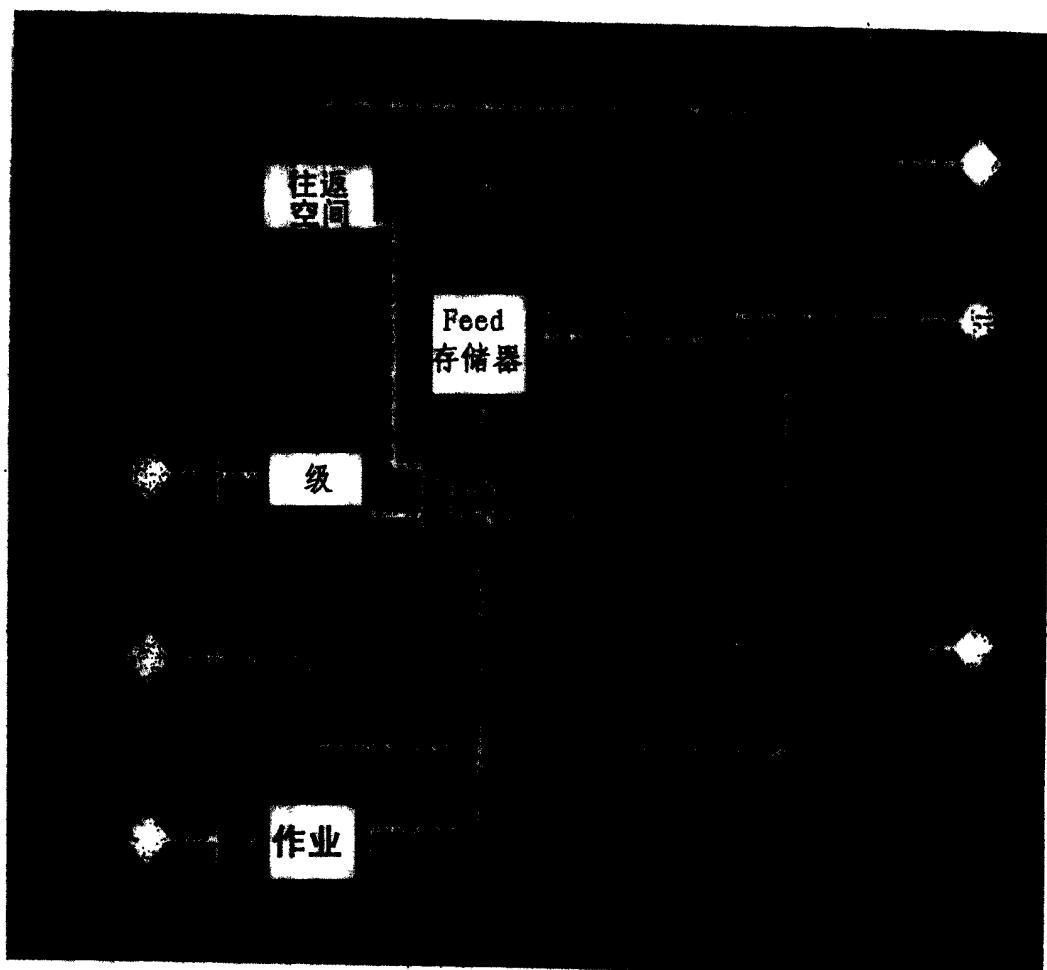


图 108

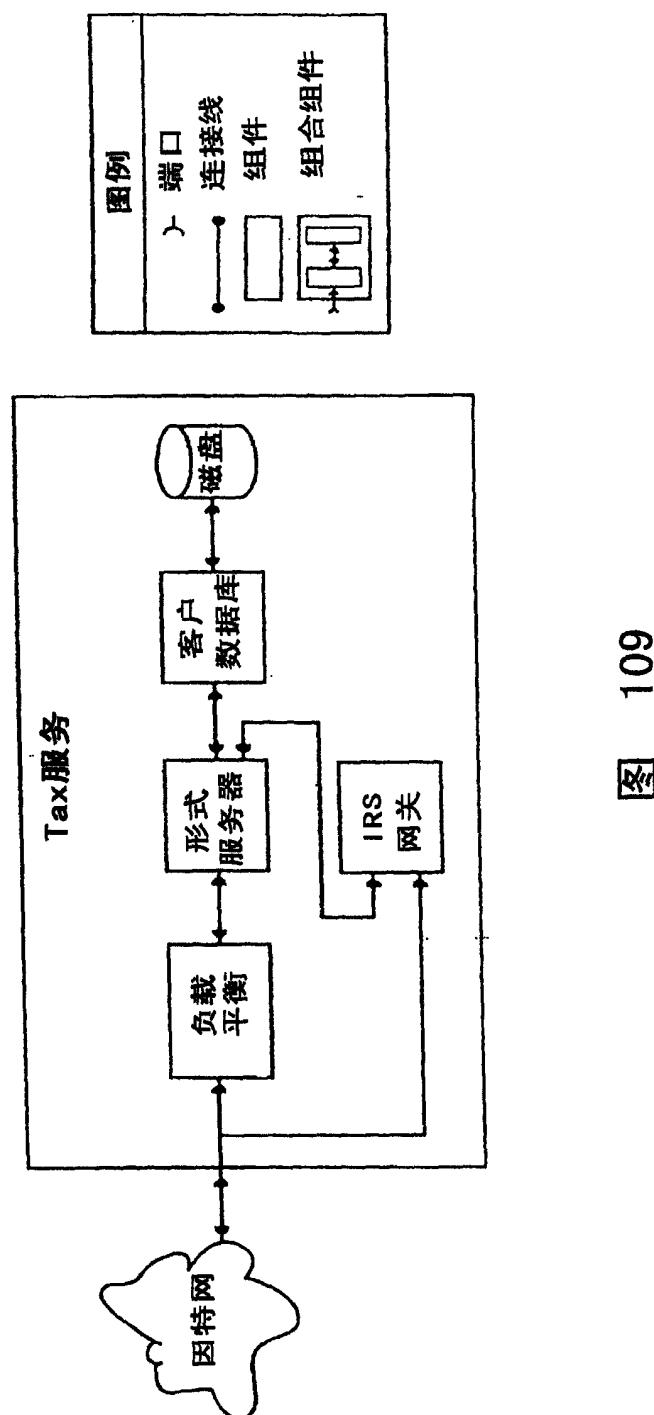


图 109

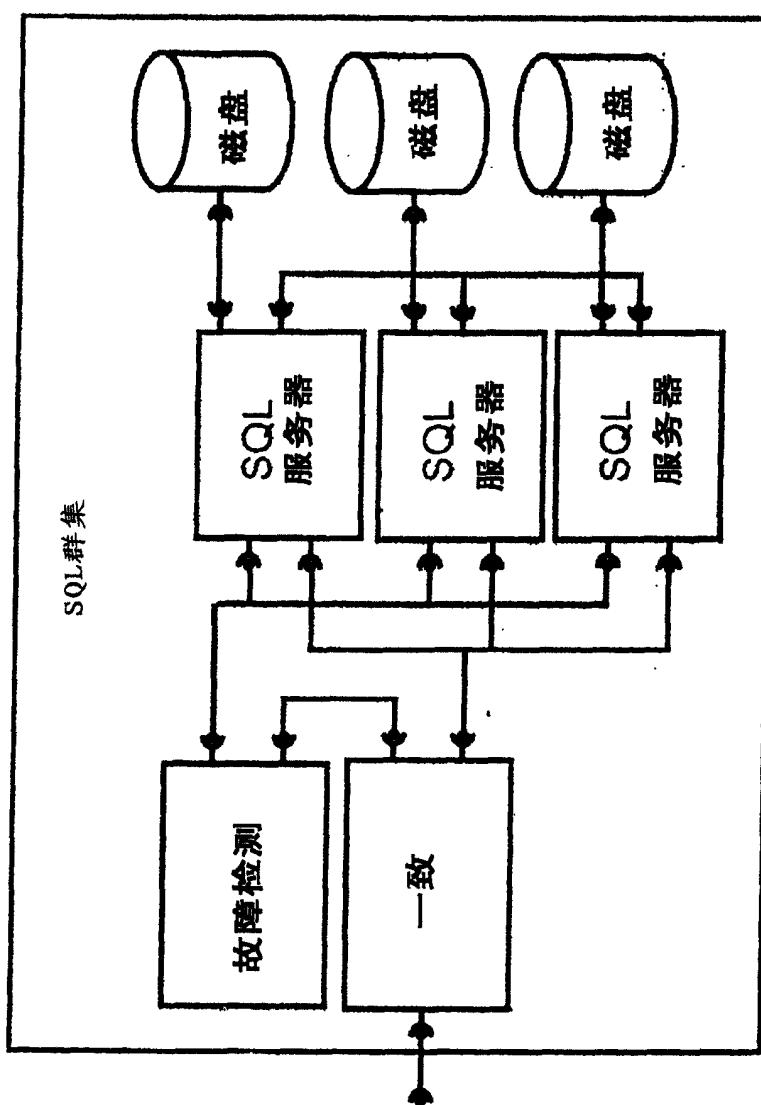
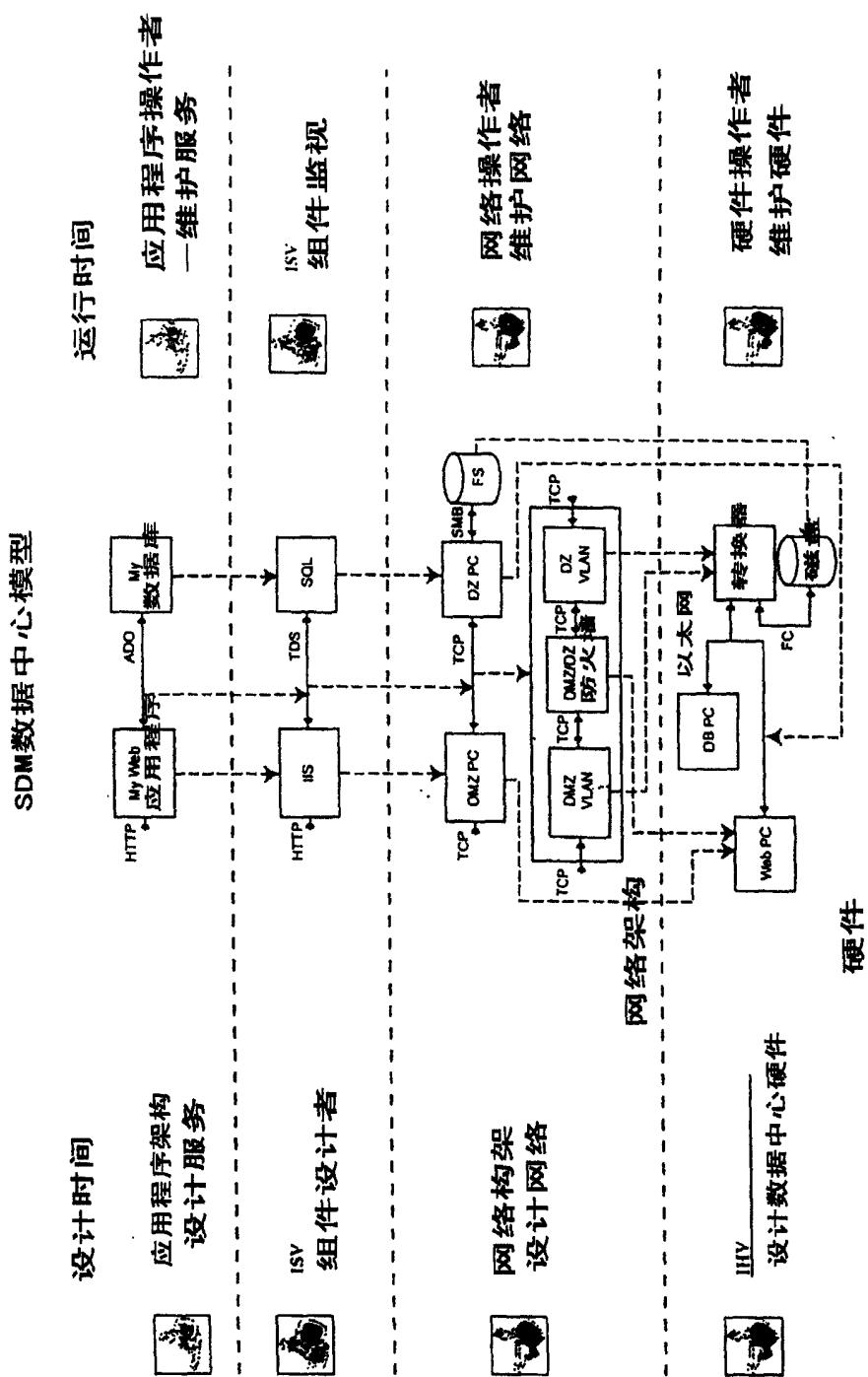
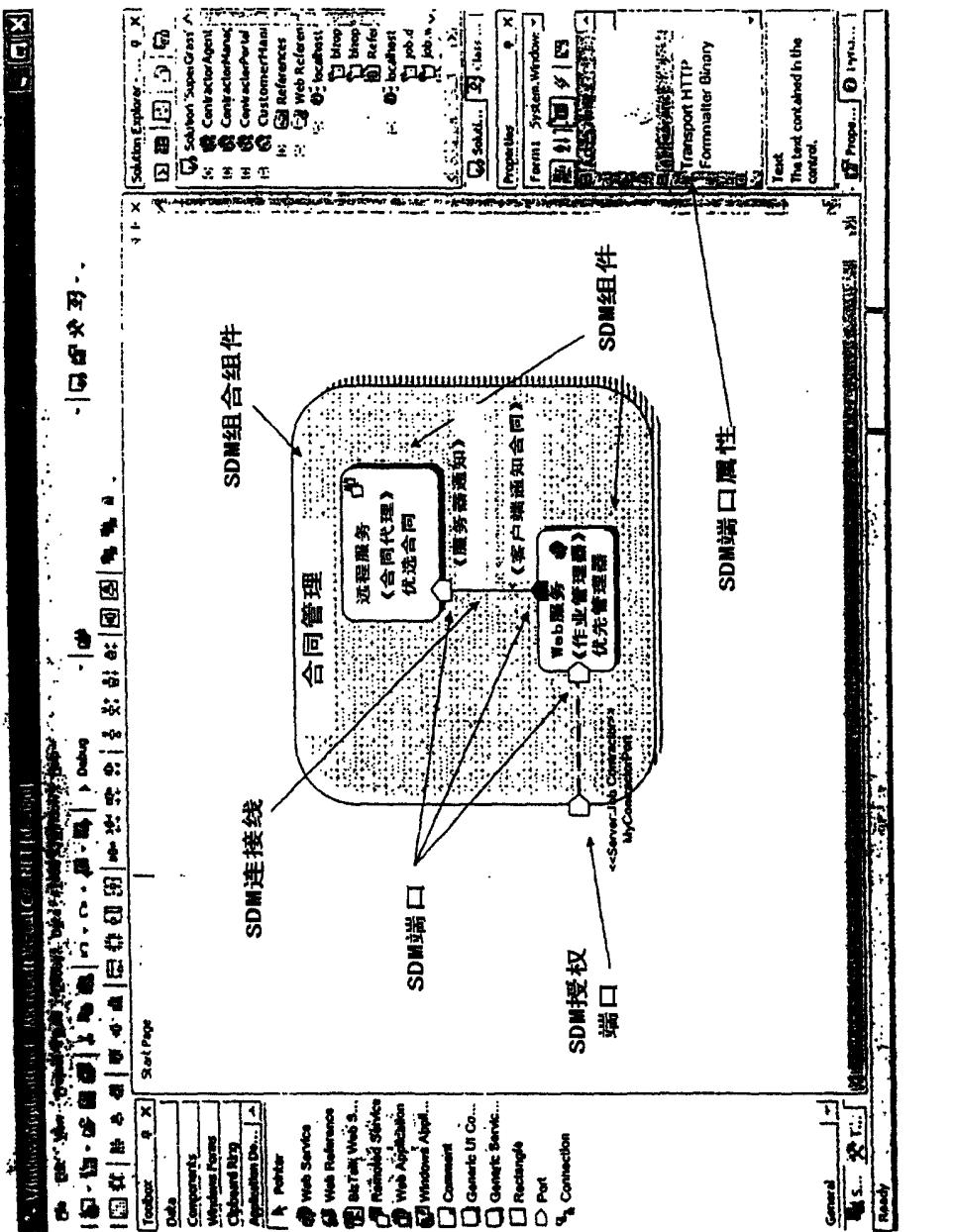


图 110

**图 111**



112
冬

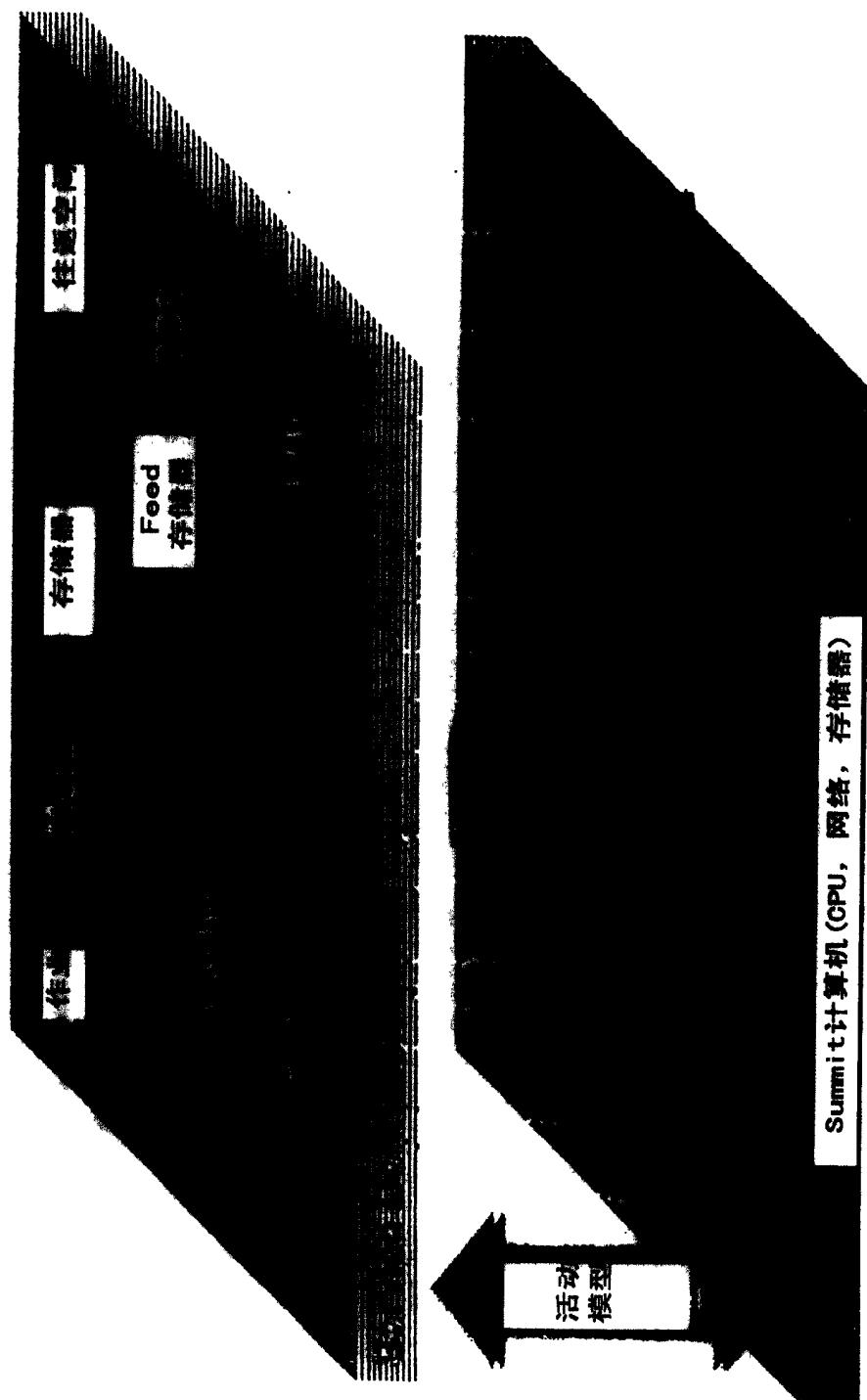


图 113

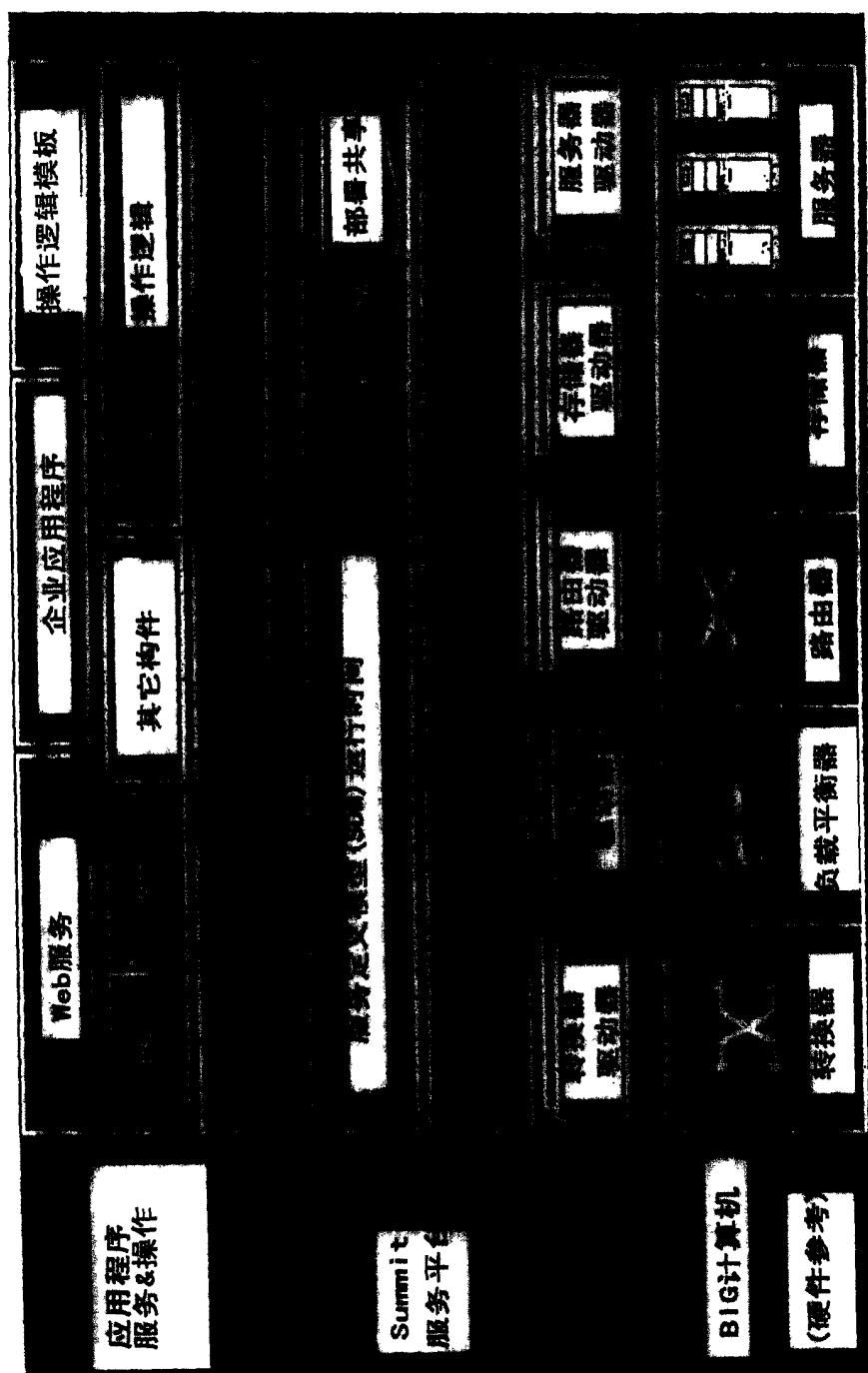


图 114

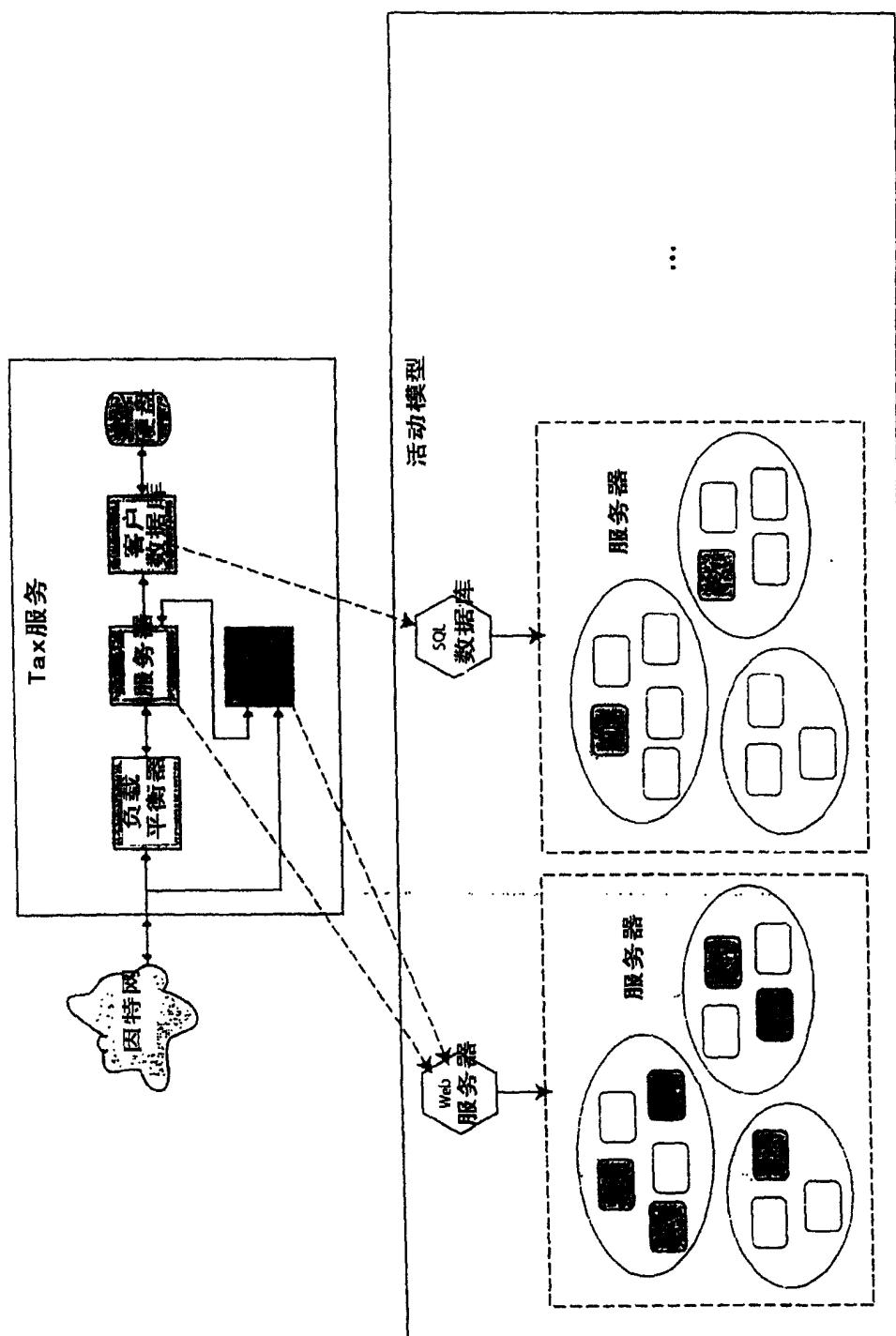


图 115

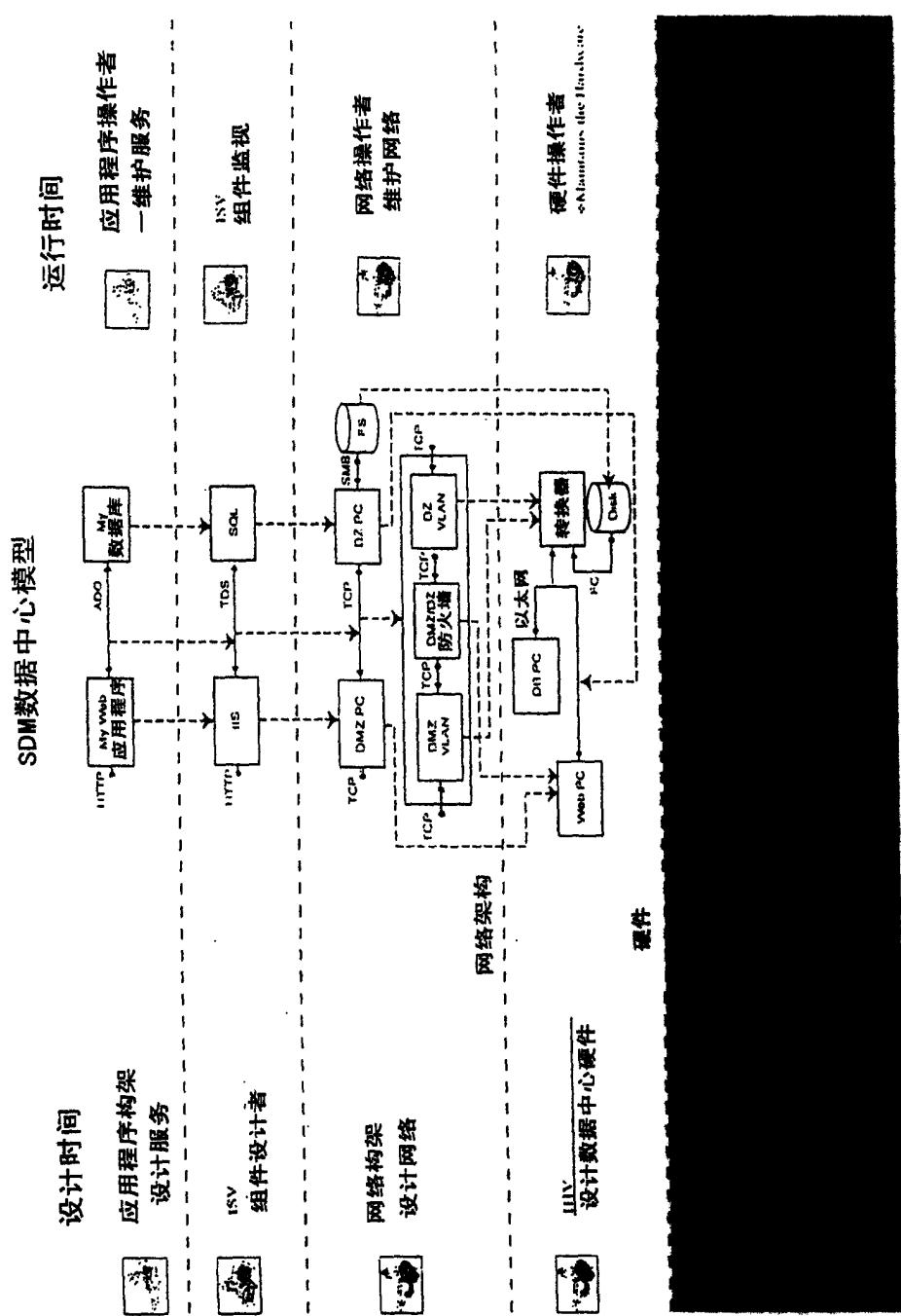


图 116

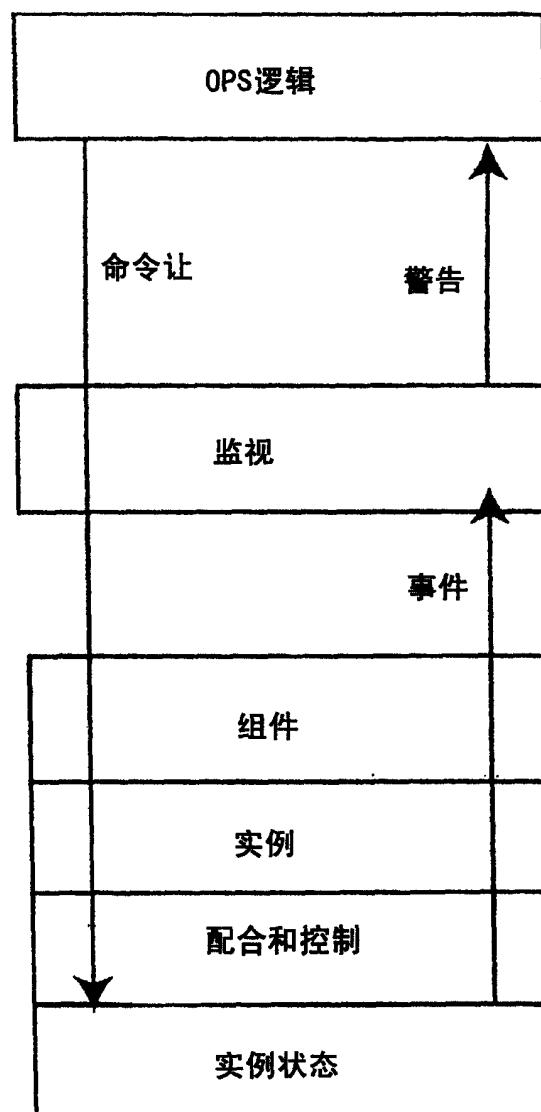


图 117

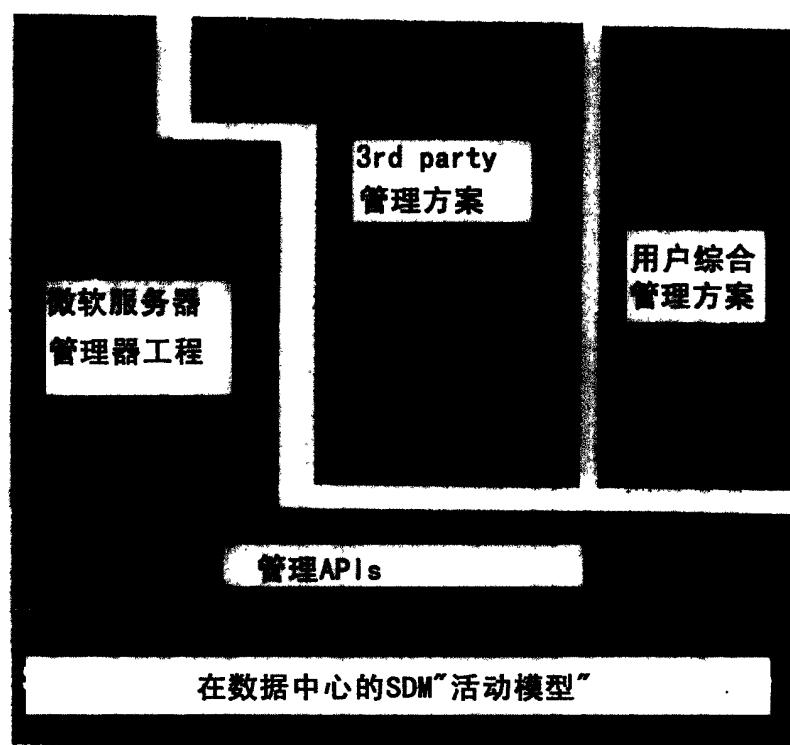


图 118

通过SDM管理杂散环境:



图 119