



[12] 发明专利说明书

专利号 ZL 03826941.4

[45] 授权公告日 2009 年 10 月 14 日

[11] 授权公告号 CN 100550010C

[22] 申请日 2003.8.21 [21] 申请号 03826941.4

[86] 国际申请 PCT/US2003/026150 2003.8.21

[87] 国际公布 WO2005/029363 英 2005.3.31

[85] 进入国家阶段日期 2006.2.20

[73] 专利权人 微软公司

地址 美国华盛顿州

[72] 发明人 W·C·吴 M·E·迪姆

E·G·谢帕德 方黎江 J·李

M·B·泰勒

[56] 参考文献

US6477564B1 2002.11.5

US6047291A 2000.4.4

US2002/0143521A1 2002.10.3

CN1255215A 2000.5.31

审查员 丛 珊

[74] 专利代理机构 上海专利商标事务所有限公司
代理人 张政权

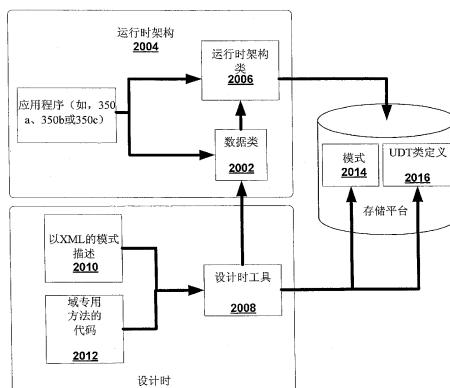
权利要求书 2 页 说明书 153 页 附图 27 页

[54] 发明名称

用于将应用程序与基于项的存储平台接口的
系统和方法

[57] 摘要

本发明的各实施例针对一种存储平台(图 20)，包括：数据存储，其中储存的数据是按照项目、元素和关系(图 20, 2014)来定义的，其中，项目是可储存在数据存储中的数据单元且包括一个或多个元素，元素是包括一个或多个字段的类型(图 20, 2016)的实例，而关系是至少两个项目之间的链接；定义不同类型项目、元素和关系(图 20, 2016)的模式组(图 20, 2014)；以及应用程序编程接口(图 20, 350a、250b 或 350c)，它对模式组中定义的不同项目、元素和关系的每一个包括一个类(图 20, 2008)。数据也可以用对现有项目类型的扩展的形式储存在数据存储中，且其中，应用程序编程接口对每一不同的项目扩展包括一个类(图 20, 2006)。



1. 一种存储平台系统，包括：

硬件/软件接口模块，所述硬件/软件接口模块是计算机系统的底层硬件组件和在计算机系统上执行的应用程序之间的接口；

数据存储模块，其中所储存的数据是按照项目、元素和关系来定义的，其中，项目是可储存在所述数据存储模块中的信息单元且可由所述硬件/软件接口模块访问，并且所述项目是具有一组由硬件/软件接口模块展现给最终用户的所有对象共同支持的基本属性的对象，其中所述项目还包括一个或多个元素和关系，元素是类型的实例，包括一个或多个所述基本属性，而关系是至少两个项目之间的链接；

模式开发工具模块，用于提供定义不同类型的项目、元素和关系的模式组；以及

应用程序编程接口模块，它对所述模式组中定义的不同项目、元素和关系的每一个生成一个类。

2. 如权利要求 1 所述的存储平台系统，其特征在于，数据也可以用对现有项目类型的扩展的形式被储存在所述数据存储模块中，且其中，所述应用程序编程接口模块对每一不同的项目扩展定义一个类。

3. 如权利要求 1 所述的存储平台系统，其特征在于，对每一类型的项目、元素和关系的类是基于定义每一类型的项目、元素和关系的所述模式组自动生成的。

4. 如权利要求 1 所述的存储平台系统，其特征在于，对每一类型的项目、元素和关系的类定义了一组数据类，且其中，所述应用程序编程接口模块还包括为所述数据类定义一组公共行为的第二组类。

5. 如权利要求 4 所述的存储平台系统，其特征在于，所述第二组类包括表示存储平台系统范围且为所述数据存储模块上的查询提供上下文的第一类，以及表示所述数据存储模块上的查询的结果的第二类。

6. 如权利要求 1 所述的存储平台系统，其特征在于，还包括其上实现所述数据存储模块的数据库引擎，且其中，所述数据存储模块中不同类型的项目、元素和关系在所述数据库引擎中被实现为用户定义类型 UDT。

7. 如权利要求 6 所述的存储平台系统，其特征在于，所述应用程序编程接口模块提供了一查询模型，所述查询模型使得应用程序员能够以将所述应用程序员与

所述数据库引擎的查询语言的细节隔离的方式，基于所述数据存储模块中的项目的各种属性来形成查询。

8. 一种用于提供应用程序和用于储存、组织、共享和搜索数据的存储平台系统之间的应用程序编程接口模块的方法，其中所述存储平台系统包括数据存储模块，其中所储存的数据是按照项目、元素和关系来定义的，其中项目是可储存在所述数据存储模块中的信息单元并可由硬件/软件接口模块访问，所述硬件/软件接口模块是计算机系统的底层硬件组件和在计算机系统上执行的应用程序之间的接口，并且所述项目是具有一组由硬件/软件接口模块展现给最终用户的所有对象共同支持的基本属性的对象，其中所述项目还包括一个或多个元素和关系，元素是类型的实例，包括一个或多个所述基本属性，而关系是至少两个项目之间的链接，所述方法包括以下步骤：

提供定义不同类型的项目、元素和关系的模式组；以及

为所述模式组中定义的不同的项目、元素和关系的每一个生成一个类，作为所述应用程序编程接口模块的一部分。

9. 如权利要求 8 所述的方法，其特征在于，数据也可以用对现有项目类型的扩展的形式被储存在所述数据存储模块中，且其中，所述方法还包括为每一不同的项目扩展生成一个类。

10. 如权利要求 9 所述的方法，其特征在于，为每一类型的项目、元素和关系生成的类定义了一组数据类，且其中，所述方法还包括提供为所述数据类定义一组公共行为的第二组类，作为所述应用程序编程接口模块的一个额外部分的步骤。

11. 如权利要求 10 所述的方法，其特征在于，所述第二组类包括表示存储平台系统范围且为所述数据存储模块上的查询提供上下文的第一类，以及表示所述数据存储模块上的查询的结果的第二类。

12. 如权利要求 8 所述的方法，其特征在于，所述存储平台系统的数据存储模块是在数据库引擎上实现的，且其中，所述方法还包括在将所述数据存储模块中不同类型的项目、元素和关系实现为所述数据引擎中的用户定义类型 UDT 的步骤。

用于将应用程序与基于项的存储平台接口的系统和方法

交叉引用

本申请的主题涉及以下共同转让的申请中公开的发明：与本申请同时提交的、标题为“SYSTEMS AND METHODS FOR REPRESENTING UNITS OF INFORMATION MANAGEABLE BY A HARDWARE/SOFTWARE INTERFACE SYSTEM BUT INDEPENDENT OF PHYSICAL REPRESENTATION（用于表示可以由硬件/软件接口系统管理但独立于物理表示的信息单元的系统和方法）”的美国专利申请（申请号未分配）（代理卷号 MSFT-1748）；与本申请同时提交的、标题为“SYSTEMS AND METHODS FOR SEPARATING UNITS OF INFORMATION MANAGEABLE BY A HARDWARE/SOFTWARE INTERFACE SYSTEM FROM THEIR PHYSICAL ORGANIZATION（用于将可由硬件/软件接口系统管理的信息单元与其物理组织分离的系统和方法）”的美国专利申请（申请号未分配）（代理卷号 MSFT-1749）；与本申请同时提交的、标题为“SYSTEMS AND METHODS FOR THE IMPLEMENTATION OF A BASE SCHEMA FOR ORGANIZING UNITS OF INFORMATION MANAGEABLE BY A HARDWARE/SOFTWARE INTERFACE SYSTEM（用于实现用于组织可由硬件/软件接口系统管理的信息单元的基本模式的系统和方法）”的美国专利申请（申请号未分配）（代理卷号 MSFT-1750）；与本申请同时提交的、标题为“SYSTEMS AND METHODS FOR THE IMPLEMENTATION OF A CORE SCHEMA FOR PROVIDING A TOP-LEVEL STRUCTURE FOR ORGANIZING UNITS OF INFORMATION MANAGEABLE BY A HARDWARE/SOFTWARE INTERFACE SYSTEM（用于实现提供用于组织可由硬件/软件接口系统管理的信息单元的顶层结构的核心模式的系统和方法）”的美国专利申请（申请号未分配）（代理卷号 MSFT-1751）；与本申请同时提交的、标题为“SYSTEMS AND METHOD FOR REPRESENTING RELATIONSHIPS BETWEEN UNITS OF INFORMATION MANAGEABLE BY A HARDWARE/SOFTWARE INTERFACE SYSTEM（用于表示可由硬件/软件接口系统管理的信息单元之间的关系的系统和方法）”的美国专利申请（申请号未分配）

(代理卷号 MSFT-1752)；与本申请同时提交的、标题为“STORAGE PLATFORM FOR ORGANIZING, SEARCHING AND SHARING DATA (用于组织、搜索和共享数据的存储平台)”的美国专利申请（申请号未分配）（代理卷号 MSFT-2734）；以及与本申请同时提交的、标题为“SYSTEMS AND METHODS FOR DATA MODELING IN AN ITEM-BASED STORAGE PLATFORM (用于基于项目的存储平台中的数据建模的系统和方法)”的美国专利申请（申请号未分配）（代理卷号 MSFT-2735）。

技术领域

本发明一般涉及信息存储和检索领域，尤其涉及用于在计算机化系统中组织、搜索和共享不同类型的数据的活动存储平台。

发明背景

在最近十年中，单个盘的容量每年增长约百分之 70 (70%)。摩尔 (Moore) 定律精确地预测了在过去数年中中央处理单元 (CPU) 能力的惊人增长。有线和无线技术已经提供了数量上巨大的连接和带宽。假设当前趋势持续，在数年内一般的膝上计算机将具有约万亿字节 (TB) 的存储并包含数百万个文件，而 5 千亿字节 (500GB) 的驱动器成为常见的。

消费者使用他们的计算机主要用于通信和组织个人信息，不论它们是传统的个人信息管理器 (PIM) 风格的数据还是如数字音乐或照片那样的媒体。数字内容的量和存储原始字节的能力已经大量地增长；然而消费者可用于组织和统一此数据的方法却跟不上步伐。知识工人花费大量时间来管理和共享信息，某些研究估计，知识工人花费 15—25% 的时间在与无效信息有关的活动上。另外研究估计，典型的知识工人每天约花费 2.5 小时搜索信息。

开发者与信息技术 (IT) 部门投资大量的时间与金钱来构建他们自己的用于公用存储抽象的数据存储，以表示如人、地方、时间和事件等事项。这不仅导致重复的工作，还形成公用数据的孤岛，没有共同搜索或共享那些数据的机制。仅考虑当今在运行 Microsoft Windows 操作系统的计算机上有多少地址簿。如电子邮件客户端和个人财务程序那样的许多应用程序保留各自的地址簿，且在每个那样的程序分别维护的地址簿数据应用程序之间只有很少共享。因而，财务程序（如 Microsoft Money）不与在电子邮件联系人文件夹（如 Microsoft Outlook 中的联系人文件夹）

中维护的地址共享支付人的地址。确实，许多用户具有多个设备，在这些设备之间和包括到如 MSN 和 AOL 等商业服务的手机电话号码的各种附加来源之间应该在逻辑上同步它们的个人数据；然而共享文档的协作大部分是通过将文档附到电子邮件消息来完成的—这是手动的且低效的。

缺乏协作的一个原因是组织计算机系统中的信息的传统方法集中在使用基于文件—文件夹—目录的系统（“文件系统”），来基于用于存储文件的存储介质的物理组织的抽象将多个文件组织到文件夹的目录分层结构中。在 1960 年代开发的 Multics 操作系统被认为是在操作系统级上使用文件、文件夹和目录来管理可存储数据单元的先驱。具体而言，Multics 在文件的分层结构中使用符号地址（从而引入文件路径的概念），其中文件的物理地址对用户（应用程序和最终用户）是不透明的。此文件系统完全不考虑任何单个文件的文件格式，且在文件中及文件之间的关系在操作系统级上被认为是无关的（即，与分层结构中文件的位置不同）。由于 Multics 的出现，在操作系统级上可存储的数据被组织成文件、文件夹和目录。这些文件一般包括放在由文件系统维护的一特定文件中的文件分层结构本身（“目录”）。此目录进而维护对应于该目录中所有其它文件和那些文件在分层结构（这里指文件夹）中的节点位置的条目的列表。这是本领域中近 40 年的状态。

然而，虽然提供了驻留在计算机的物理存储系统中的信息的合理表示，但是文件系统是物理存储系统的抽象，因而文件的利用需要在用户处理什么（具有上下文、特征以及与其它单元的关系的单元）和操作系统提供什么（文件、文件夹和目录）之间的间接（解释）层。结果，用户（应用程序和/或最终用户）只好强制把信息单元放入文件系统结构，即使这样做是低效的、不一致的、或不希望的。此外，现有的文件系统关于在各个文件中存储的数据的结构知之甚少，因此，大多数信息在文件中保持封闭，只能被写那些数据的应用程序访问（和可理解）。因此，缺乏信息的模式描述和管理信息的机制，导致形成数据的先进先出存储缓冲区（silo），只有很少数据能在各先进先出存储缓冲区之间共享。例如，许多个人计算机（PC）用户具有 5 个以上各异的存储，它们包含有关他们在某一层上交互的人的信息—如 Outlook 联系人、在线账户地址、Windows 地址簿、Quicken 受款人和即时消息（IM）伙伴列表—因为组织文件向这些 PC 用户提出重要的挑战。由于大多数现有的文件系统利用嵌套的文件夹隐喻来组织文件和文件夹，因此当文件数量增加时，为维持灵活且有效的组织模式所必需的努力变得十分惊人。在这些情况下，具有单个文件的多重分类是非常有用的；然而使用现有文件系统中的硬和软链接是麻烦且难以维

护的。

过去已作了若干不成功的尝试来克服文件系统的缺点。这些以前尝试中的某些已经涉及使用内容可定址的存储器来提供可以通过内容而不是通过物理地址来访问数据的机制。然而，这些努力被证明是不成功的，因而虽然内容可定址的存储器对由如高速缓存和存储器管理单元等设备的小规模使用被证明是有用的，但对如物理存储介质等设备的大规模使用由于各种原因尚不可能，因此那样的解决方案简直不存在。已作出使用面向对象的数据库（OODB）系统的其它尝试，但是这些尝试虽然带有强烈的数据库的特征，且良好的非文件表示，但在处理文件表示方面并不有效，并不能重现在硬件/软件接口系统级上基于分层结构的文件及文件夹的速度、效率和简单性。诸如试图使用 SmallTalk（和其它派生方法）的其它尝试在处理文件和非文件表示方面被证明相当有效，但缺乏有效地组织和利用在各种数据文件之间存在的关系所必需的数据库特征，因此那种系统的整体有效性是不可接受的。使用 BeOS（和其它那样操作系统研究）的又一种尝试尽管能够胜任适当地表示文件的同时又提供某些必要的数据库特征，在处理非文件的表示上被证明是不够的，这是传统文件系统同样的核心缺点。

数据库技术是存在类似挑战的另一专业领域。例如，虽然关系型数据库模型已取得很大的商业上的成功，实际上独立软件分销商（ISV）一般运用了关系型数据库软件产品（如 Microsoft SQL Server）中可得到的功能一小部分。相反，应用程序与那样产品的大多数交互是以简单的“gets”和“puts”的形式。对此虽然有若干容易明白的原因（如作为平台或数据库的不可知），一个常被忽略的关键的原因是数据库没有必要提供主要商业应用程序分销商确实需要的精确抽象。例如，虽然真实世界具有如“客户”或“订单”等“项目”的概念（以及订单的嵌入的“行式项目”作为其中的项目和项目本身），而关系型数据库只在表和行的方面来谈论。结果，虽然应用程序可能希望具有在项目级上的一致性、锁定、安全和/或触发器的方面（只列出一些），通常数据库只在表/行级上提供这些特征。尽管若每个项目映射到数据库某个表的单个行也能工作得很好，但在带多个行式项目的订单的情况下，存在一个项目实际上要映射到多个表的原因，且在此情况下，单个关系型数据库系统不能确切地提供正确的抽象。因此，应用程序必须在数据库的顶层构建逻辑以提供这些基本抽象。换言之，基本关系模型不提供在其上容易开发高级应用程序的存储数据的足够平台，因为基本关系模型需要在应用程序和存储系统之间的间接层，其中只在某些情况的应用程序中可以看到数据的语义结构。尽管某些数据库

分销商正将高级功能构建到他们的产品中（如提供对象关系能力，新的组织模型等），至今尚没有哪个提供需要的全面解决方案，其中真正的全面解决方案是为有用的域抽象（如“个人”、“位置”、“事件”等）提供有用的数据模型抽象（如“项目”、“扩展”、“关系”等）的解决方案。

考虑到现有数据存储和数据库技术中的上述缺点，需要一种新的存储平台，它提供了一种改进的能力以便组织、搜索和共享计算机系统中的所有类型的数据。一种存储平台，它在现有的文件系统和数据库系统之外扩展和扩大了数据平台，并且被设计为存储所有类型的数据。本发明满足这一需求。

发明概述

以下概述提供了对本发明的各方面的综述。该综述并非旨在提供对本发明的所有重要方面的详尽描述，也不旨在定义本发明的范围。相反，本概述旨在用作对以下的详细描述和附图的介绍。

本发明针对一种用于组织、搜索和共享数据的存储平台。本发明的存储平台在现有文件系统和数据库系统之外扩展和扩大了数据存储的概念，并被设计成储存所有类型的数据，包括结构化的、非结构化的或半结构化的数据。

依照本发明的一方面，本发明的存储平台包括在数据库引擎上实现的数据存储。在本发明的各实施例中，数据库引擎包括具有对象关系扩展的关系型数据库引擎。该数据存储实现支持数据的组织、搜索、共享、同步和安全性的数据模型。具体的数据类型在模式中描述，且该平台提供了一种扩展模式集以定义新数据类型（本质上是由模式提供的基本类型的子类型）的机制。一种同步能力有助于在用户或系统间共享数据。提供了类似文件系统的能力，它允许该数据存储与现有文件系统的互操作性，而不存在这种传统文件系统的限制。一种改变跟踪机制提供了跟踪数据存储的改变的能力。该存储平台还包括一组应用程序接口，它们使得应用程序能够访问该存储平台的上述的所有能力，并且能够访问在模式中描述的数据。

依照本发明的另一方面，由该数据存储实现的数据模型按照项目、元素和关系定义了数据存储的单元。项目是可在数据存储中存储的数据单元，并且可以包括一个或多个元素和关系。元素是类型的实例，包括一个或多个字段（此处也被称为属性）。关系是两个项目之间的链接。（如此处所使用的，这些以及其它特定的术语可以被大写，以便将它们从类似所使用的其它术语分离，然而，并不是旨在区别对待被大写的术语例如“Item”和不被大写时的同一个术语，例如“item”，并且

不应假设或暗示这种区别。)

依照本发明的另一方面，一种计算机系统包括多个项目，其中每个项构成可由硬件/软件接口系统操纵的离散的可存储信息单元；多个项目文件夹，它们构成了所述项目的组织结构；以及用于操纵多个项目的硬件/软件接口系统，并且其中每个项目属于至少一个项目文件夹，并且可以属于一个以上项目文件夹。

依照本发明的另一方面，一种计算机系统包括多个项目，其中，每一项目构成可由硬件/软件接口系统操纵的离散信息单元，且与从持久存储中导出相反，项目或某些项目属性值可以被动态地计算。换言之，该硬件/软件接口系统不要求项目被存储，并且支持某些操作，诸如枚举当前项目集的能力，或是给出项目在存储平台上的标识符（在描述应用程序编程接口或 API 的一节中更完整地描述）而检索项目的能力—例如，项目可以是蜂窝电话的当前位置，或从温度传感器读到的温度。

依照本发明的另一方面，一种用于计算机系统的硬件/软件接口系统还包括由该硬件/软件接口系统管理的多个关系互连的项目，其中所述硬件/软件接口系统操纵多个项目。依照本发明的另一方面，一种用于计算机系统的硬件/软件接口系统，其中所述硬件/软件接口系统操纵具有可由所述硬件/软件接口系统理解的属性的多个离散信息单元。依照本发明的另一方面，一种用于计算机系统的硬件/软件接口系统包括核心模式，以便定义所述硬件/软件接口系统可以理解，并且能够以一种预定的和可预测的方式直接进行处理的一组核心项目。依照本发明的另一方面，公开了一种用于操纵用于计算机系统的硬件/软件接口系统中的多个离散信息单元（“项目”）的方法，所述方法包括将所述项目与多个关系互连，以及在硬件/软件接口系统级上管理所述关系。

依照本发明的另一特征，该存储平台的 API 为存储平台模式集中定义的每个项目、项目扩展和关系提供了数据类。此外，该应用程序编程接口提供了一组框架类，它们为所述数据类定义了一组公共行为，并且与数据类一起为存储平台 API 提供了基本的编程模型。依照本发明的另一特征，该存储平台 API 提供了简化的查询模型，它以将应用程序员从底层数据库引擎的查询语言的细节隔离开的方式，使得应用程序员能够形成基于数据存储中的项目的各种属性的查询。依照本发明的存储平台 API 的又一方面，该 API 收集由应用程序对项目作出的改变，并且然后将它们组织到实现该数据存储的数据库引擎（或任何种类的存储引擎）所需的正确更新中。这使得应用程序员能够对在存储器中项目进行改变，而将数据存储更新的

复杂性留给 API。

通过其公共存储基础和被模式化的数据，本发明的存储平台能够为消费者、知识工作者和企业作出更有效的应用程序开发。它提供了丰富且可扩展的应用程序编程接口，该接口不仅使得其数据模型中所固有的能力可用，而且还包含并扩展了现有文件系统和数据库的访问方法。

通过阅读以下本发明的详细描述和附图，可以清楚本发明的其它特征和优点。

附图简述

当结合所附的附图进行阅读时，可以更好地理解决面的概述以及下面对本发明的详细描述。出于解释本发明的目的，在附图中示出了本发明的各个方面的示例性实施例；然而，本发明不限于所公开的具体方法和手段。在附图中：

图 1 是表示其中可结合本发明的各方面的计算机系统的框图；

图 2 是示出了被分为 3 个组件组的计算机系统的框图：硬件组件、硬件/软件系统接口组件和应用程序组件；

图 2A 示出了用于被分组到基于文件的操作系统中的目录内的文件夹中的文件的传统的基于树的分层结构；

图 3 是示出依照本发明的存储平台的框图；

图 4 示出了本发明的各实施例中项目、项目文件夹和类别之间的结构关系；

图 5A 示出了项目的结构的框图；

图 5B 是示出图 5A 的项目的复杂属性类型的框图；

图 5C 是示出“Location（位置）”项目的框图，其中进一步描述（明确地列出）其复杂类型；

图 6A 示出了作为基础模式中找到的项目的子类型的项目；

图 6B 是示出了图 6A 的子类型项目的框图，其中明确地列出了其继承的类型（除了其直接属性之外）；

图 7 是示出了基本模式的框图，该基本模式包括其两个顶层类类型，即 Item（项目）和 PropertyBase（属性基），以及从其中导出的附加基础模式类型；

图 8A 是示出核心模式中的项目的框图；

图 8B 是示出核心模式中的属性类型的框图；

图 9 是示出项目文件夹、其成员项目以及项目文件夹和其成员项目之间的互连关系的框图；

图 10 是示出了类别（它本身也是项目）、其成员项目以及类别及其成员项目之间的互连关系的框图；

图 11 是示出依照本发明的存储平台的数据模型的引用类型层次的图；

图 12 是示出依照本发明的一个实施例关系是如何被分类的图；

图 13 是示出依照本发明的一个实施例的通知机制的图；

图 14 是示出其中两个事务都向同一个 B 树插入新记录的示例的图；

图 15 示出了依照本发明的一个实施例的数据改变检测过程；

图 16 示出了示例性的目录树；

图 17 示出了依照本发明的一方面其中现有的基于目录的文件系统的文件夹被移动到该存储平台数据存储中的示例；

图 18 示出了依照本发明的一方面包含文件夹的概念；

图 19 示出了存储平台 API 的基本体系结构；

图 20 示意性地表示了存储平台 API 栈的各个组件；

图 21A 和 21B 是示例性联系人模式（项目和元素）的图形表示；

图 22 示出了依照本发明的一方面的存储平台 API 的运行时框架；

图 23 示出了依照本发明的一个实施例的“FindAll”操作的执行；

图 24 示出了依照本发明的一方面用于从存储平台模式生成存储平台 API 类的过程；

图 25 示出了依照本发明的另一方面 File（文件）API 所基于的模式；

图 26 是示出依照本发明的一个实施例用于数据安全目的的访问掩码格式的示意图；

图 27(a)、(b)、(c)给出了依照本发明的一方面的一个实施例从现有的安全区域内划分出一个新的同样地保护的安全区域；

图 28 是示出依照本发明的一方面的一个实施例的项目搜索视图的概念的示意图；

图 29 是示出依照本发明的一个实施例的示例性项目层次的示意图；

发明详细描述

目 录

A. 示例性计算环境.....	16
B. 传统的基于文件的存储	19
II. 用于组织、搜索和共享数据的新存储平台	20
A. 词汇表.....	20
B. 存储平台综述.....	21
C. 数据模型	22
1. 项目	23
2. 项目标识.....	26
a) 项目引用.....	26
(1) ItemIDReference.....	26
(2) ItemPathReference.....	26
b) 引用类型分层结构	27
3. 项目文件夹和类别	27
4. 模式	28
a) 基础模式.....	28
b) 核心模式.....	29
5. 关系	30
a) 关系声明.....	31
b) 持有关系.....	32
c) 嵌入关系.....	33
d) 引用关系.....	34
e) 规则和约束.....	34
f) 关系的排序.....	34
6. 可扩展性	38
a) 项目扩展.....	39
b) 扩展 NestedElement 类型	42
D. 数据库引擎	43
1. 使用 UDT 的数据存储实现	44
2. 项目映射	46
3. 扩展映射	47
4. 嵌套元素映射	48
5. 对象身份	49
6. SQL 对象命名	49
7. 列命名.....	50

8. 搜索视图	50
a) 项目	51
(1) 主项目搜索视图	51
(2) 类型化的项目搜索视图	51
b) 项目扩展	52
(1) 主扩展搜索视图	52
(2) 类型化的扩展搜索视图	52
c) 嵌套的元素	53
d) 关系	53
(1) 主关系搜索视图	53
(2) 关系实例搜索视图	53
9. 更新	54
10. 改变跟踪及墓碑	55
a) 改变跟踪	55
(1) “主”搜索视图中的改变跟踪	55
(2) “类型化的”搜索视图中的改变跟踪	56
b) 墓碑	57
(1) 项目墓碑	57
(2) 扩展墓碑	57
(3) 关系墓碑	58
(4) 墓碑清除	58
11. 助手 API 和函数	58
a) 函数[System.Storage].GetItem	59
b) 函数[System.Storage].GetExtension	59
c) 函数[System.Storage].GetRelationship	59
12. 元数据	59
a) 模式元数据	59
b) 实例元数据	59
E. 安全性	59
1. 综述	59
2. 安全模型的详细描述	64
a) 安全描述符结构	64
(1) 访问掩码格式	65
(2) 类属访问权限	65
(3) 标准访问权限	66

b) 项目专用权限	66
(1) 文件和目录对象专用权限	66
(2) WinFSItemRead	67
(3) WinFSItemReadAttributes	68
(4) WinFSItemWriteAttributes	68
(5) WinFSItemWrite	68
(6) WinFSItemAddLink	69
(7) WinFSItemDeleteLink	69
(8) 删除项目的权限	69
(9) 复制项目的权限	70
(10) 移动项目的权限	70
(11) 查看项目上的安全策略的权限	70
(12) 改变项目上的安全策略的权限	70
(13) 没有直接等效物的权限	71
3. 实现	71
a) 在容器中创建新项目	71
b) 向项目添加显式 ACL	71
c) 向项目添加持有关系	72
d) 从项目删除持有关系	72
e) 从项目中删除显式 ACL	72
f) 修改与项目相关联的 ACL	73
F. 通知和改变跟踪	73
1. 存储改变事件	73
a) 事件	73
b) 监视程序	74
2. 改变跟踪和通知生成机制	75
a) 改变跟踪	76
b) 时间标记管理	77
c) 数据改变检测—事件检测	77
G. 同步	78
1. 存储平台到存储平台的同步	78
a) 同步 (Sync) 控制应用程序	79
b) 模式注释	79
c) 同步配置	80
(1) 共同体文件夹—映射	81

(2) 概况	82
(3) 时间表	82
d) 冲突处理	82
(1) 冲突检测	83
(a) 基于知识的冲突	83
(b) 基于约束的冲突	83
(2) 冲突处理	84
(a) 自动冲突分解	84
(b) 冲突日志记录	85
(c) 冲突检查和分解	85
(d) 复制品的收敛和冲突分解的传播	85
2. 对非存储平台数据存储的同步	86
a) 同步服务	86
(1) 改变枚举	86
(2) 改变应用	87
(3) 冲突分解	87
b) 适配器实现	88
3. 安全性	88
4. 可管理性	88
G. 传统文件互操作性	88
1. 互操作性模型	89
2. 数据存储特征	90
a) 非卷	90
b) 存储结构	90
c) 不移植所有文件	91
d) 对存储平台文件的 NTFS 名字空间访问	91
e) 期望的名字空间/驱动器字母	91
I. 存储平台 API	91
1. 综述	92
2. 命名和范围	92
3. 存储平台 API 组件	94
4. 数据类	94
5. 运行时架构	101
a) 运行时架构类	101
(1) ItemContext	101

(2) ItemSearcher.....	102
(a) 目标类型	102
(b) 过滤器	102
(c) 准备搜索	103
(d) 查找选项	103
(3) 项目结果流 (“FindResult”)	104
b) 操作中的运行时架构.....	105
c) 公共编程模式	106
(1) 打开和关闭 ItemContext 对象.....	106
(2) 搜索对象	107
(a) 搜索选项	108
(b) FindOne 和 FindOnly.....	108
(c) 搜索 ItemContext 上的快捷方式	109
(d) 按照 ID 或路径查找	109
(e) GetSearcher 模式	110
(3) 更新存储	110
6. 安全性.....	112
7. 对关系的支持	112
a) 基础关系类型	113
(1) Relationship 类	113
(2) ItemReference 类	114
(3) ItemIdReference 类	114
(4) ItemPathReference 类	115
(5) RelationshipId 结构	115
(6) VirtualRelationshipCollection 类	116
b) 生成的关系类型	117
(1) 生成的关系类型.....	118
(2) RelationshipPrototype 类	118
(3) RelationshipPrototpyeCollection 类	119
c) Item 类中的关系支持	119
(1) Item 类	119
(2) RelationshipCollection 类	119
d) 搜索表达式中的关系支持	120
(1) 从项目遍历到关系.....	120
(2) 从关系遍历到项目	120

(3) 组合关系遍历	121
e) 关系支持的示例使用.....	121
(1) 搜索关系	121
(2) 从关系导航到源和目标项目.....	122
(3) 从源项目导航到关系	123
(4) 创建关系（以及项目）	124
(5) 删除关系（以及项目）	125
8. “扩展”存储平台 API.....	125
a) 域行为	126
b) 增值行为	126
c) 作为服务提供者的增值行为	127
9. 设计时架构	128
10. 查询形式	128
a) 过滤器基础.....	129
b) 类型强制转换	130
c) 过滤器句法	130
11. 遥控.....	131
a) API 中的本地/远程透明性.....	131
b) 遥控的存储平台实现.....	132
c) 访问非存储平台存储.....	132
d) 与 DFS 的关系	132
e) 与 GXA/Indigo 的关系.....	132
12. 约束.....	133
13. 共享.....	134
a) 表示共享	135
b) 管理共享	135
c) 访问共享	135
d) 可发现性	136
14. Find 的语义	136
15. 存储平台 Contacts API.....	136
a) System.Storage.Contact 的综述	137
b) 域行为	137
16. 存储平台 File API	138
a) 介绍	138
(1) 在存储平台中反映 NTFS 卷	139

(2) 在存储平台名字空间中创建文件和目录.....	139
b) 文件模式.....	140
c) System.Storage.Files 的综述	140
d) 代码示例.....	140
(1) 打开文件并向其写入	140
(2) 使用查询	141
e) 域行为	141
J. 总结.....	141

I. 引言

本发明的主题用细节来描述，以满足法定的要求。然而，该描述本身不试图限制本专利的范围。相反，本发明者设想要求保护的主题也能以其它方式实施，以结合其它当前和未来的技术来包括类似于本文档所描述的不同的步骤或步骤的组合。此外，虽然术语“步骤”在这里可用于意味着所采用的方法的不同元素，然而该术语不能被解释为隐含这里所揭示的各步骤之间的特定次序，除非明确地描述了各个步骤的次序。

A. 示例性计算环境

本发明的许多实施例可在计算机上执行。图 1 和下面讨论旨在提供其中实现本发明的合适计算环境的简要描述。虽然不是必需，但本发明的诸方面能以诸如由如客户工作站或服务器的计算机上执行的程序模块的计算机可执行指令的一般上下文中描述。一般而言，程序模块包括例程、程序、对象、组件、数据结构等，它们执行特定任务或实现特定抽象数据类型。此外，本发明可用其它计算机系统配置实现，包括手持设备、多处理器系统、基于微处理器的系统或可编程消费者电子设备、网络 PC、小型机、大型机等。本发明还能在分布式计算环境中实现，其中任务由通过通信网络链接的远程处理设备完成。在分布式计算环境中，程序模块能位于本地或远程存储器存储设备中。

如图 1 所示，示例性通用计算系统包括传统的个人计算机 20 等，它包括处理单元 21、系统存储器 22 和将包括系统存储器的各种系统组件耦合到处理单元 21 的系统总线 23。系统总线 23 可以是若干种总线结构的任一种，包括存储总线或存储控制器、外围总线、以及使用各种总线体系结构的任一种的局部总线。系统存储器包括只读存储器 (ROM) 24 和随机存取存储器 (RAM) 25。基本输入/输出系统 26 (BIOS) 包含如在启动时帮助在个人计算机 20 的诸元件之间传输信息的基本例程，它存储在 ROM 24 中。个人计算机 20 还可包括读写硬盘 (未示出) 的硬盘驱动器 27、读写可移动磁盘 29 的磁盘驱动器 28、读写如 CDROM 或其它光介质的可移动光盘 31 的光盘驱动器 30。硬盘驱动器 27、磁盘驱动器 28 和光盘驱动器 30 分别通过硬盘驱动器接口 32、磁盘驱动器接口 33 和光盘驱动器接口 34 连接到系统总线 23。驱动器及其相关联的计算机可读介质为个人计算机 20 提供计算机可读指令、数据结构、程序模块和其它数据的非易失性存储。虽然这里描述的示例性

环境采用硬盘、可移动磁盘 29 和可移动光盘 31，但本领域的技术人员可以理解，在示例性操作环境中也能使用可存储能由计算机访问的数据的其它类型计算机可读介质，如盒式磁带、闪存卡、数字视频盘、Bernoulli 盒式磁带、随机存取存储器（RAM）、只读存储器（ROM）等。类似地，示例环境还可包括许多类型的监视设备，如热敏和安全或火警系统，及其它信息源。

若干程序模块能存储在硬盘、磁盘 29、光盘 31、ROM 24 或 RAM 25 中，包括操作系统 35、一个或多个应用程序 36、其它程序模块 37 和程序数据 38。用户能通过如键盘 40 和定点设备 42 等输入设备将命令和信息输入到个人计算机 20。其它输入设备（未示出）可包括麦克风、操纵杆、游戏垫、圆盘式卫星天线、扫描仪等。这些和其它输入设备常通过耦合到系统总线的串行接口 46 连接到处理单元 21，但也可通过其它接口连接，如并行口、游戏口或通用串行总线（USB）。监视器 47 或其它类型的显示设备也通过如视频适配器 48 的接口连接到系统总线 23。除监视器 47 以外，个人计算机通常包括如扬声器和打印机等其它外围输出设备（未示出）。图 1 的示例系统还包括主机适配器 55、小型计算机系统接口（SCSI）总线 56 和连接到 SCSI 总线 56 的外部存储设备 62。

个人计算机 20 可使用到如远程计算机 49 的一个或多个远程计算机的逻辑连接在网络环境中操作。远程计算机 49 可以是另一台个人计算机、服务器、路由器、网络 PC、对等设备或其它常见的网络节点，并通常包括以上对个人计算机 20 描述的许多或所有元件，虽然在图 1 中只示出存储器存储设备 50。图 1 中画出的逻辑连接包括局域网（LAN）51 和广域网（WAN）52。那样的网络环境常见于办公室、企业范围计算机网络、内联网和因特网。

在 LAN 网络环境中使用时，个人计算机 20 通过网络接口或适配器 53 连接到 LAN 51。在 WAN 网络环境中使用时，个人计算机 20 通常包括调制解调器 54 或用于通过如因特网等广域网 52 建立通信的其它装置。内置或外接的调制解调器 54 通过串行端口接口 46 连接到系统总线 23。在网络环境中，相对个人计算机 20 描述的程序模块或其部分可存储在远程存储器存储设备中。可以理解，示出的网络连接是示例性的，可使用在计算机之间建立通信链路的其它手段。

如图 2 的框图所示，计算机系统 200 能被粗略地分成三个组件组：硬件组件 202、硬件/软件接口系统组件 204、以及应用程序组件 206（在这里某些上下文中也称为“用户组件”或“软件组件”）。

回到图 1，在计算机系统 200 的各实施例中，硬件组件 202 可包括中央处理单

元（CPU）21、存储器（ROM 24 和 RAM 25）、基本输入/输出系统（BIOS）26、以及各种输入/输出（I/O）设备，如键盘 40、鼠标 42、监视器 47、和/或打印机（未示出）等。硬件组件 202 包括计算机系统 200 的基本物理基础结构。

应用程序组件 206 包括各种软件程序，包括但不限于编译器、数据库系统、文字处理程序、商业程序、视频游戏等。应用程序提供计算机资源用于为各种用户（机器、其它计算机系统和/或最终用户）解决问题、提供解决方案和处理数据的手段。

硬件/软件接口系统组件 204 包括（在某些实施例中可以仅包括）操作系统，在大多数情况下后者本身包括外壳和内核。“操作系统”（OS）是担当在应用程序和计算机硬件之间的中介的特殊程序。硬件/软件接口系统组件 204 还可包括虚拟机管理器（VMM）、公用语言运行库（CLR）或其功能等效物、Java 虚拟机（JVM）或其功能等效物、或在计算机系统中代替操作系统或除操作系统外的其它软件组件。硬件/软件接口系统的目的是提供用户能在其中执行应用程序的环境。任何硬件/软件接口系统的目标是使计算机系统便于使用，以及以有效的方式利用计算机硬件。

硬件/软件接口系统一般在启动时被加载到计算机系统，并随后管理在计算机系统中所有应用程序。应用程序通过经由应用程序接口（API）请求服务来与硬件/软件接口系统交互。某些应用程序使最终用户能通过如命令语言或图形用户界面（GUI）等用户接口与硬件/软件接口系统交互。

硬件/软件接口系统传统上执行应用程序的各种服务。在多个程序同时运行的多任务硬件/软件接口系统中，硬件/软件接口系统确定哪些应用程序应以什么次序运行，以及在切换到另一应用程序以供轮流之前对每个应用程允许多少时间。硬件/软件接口系统还管理在多个应用程序之间内部存储器的共享，并处理来往于如硬盘、打印机和拨号端口等附加的硬件设备的输入和输出。硬件/软件接口系统还将有关操作状态和可能发生的任何错误的消息发送到每个应用程序（在某些情况下到最终用户）。硬件/软件接口系统也能下传（offload）批处理作业（如打印）的管理，使得启动的应用程序能摆脱此工作并重新开始其它处理和/或操作。在能提供并行处理的计算机上，硬件/软件接口系统还管理划分程序，使得它同时在多个处理器上运行。

硬件/软件接口系统外壳（这里简称“外壳”）是到硬件/软件接口系统的交互式最终用户界面。（外壳也称为“命令解释器”，或在操作系统中称为“操作系统

“外壳”）。外壳是可直接由应用程序和/或最终用户访问的硬件/软件接口系统的外层。与外壳相反，内核是直接与硬件组件交互的硬件/软件接口系统的最内层。

虽然可构想本发明的许多实施例尤其适用于计算机化的系统，然而在本文档哪个中不意味着将本发明限于那些实施例。相反，这里使用的术语“计算机系统”旨在包括能存储和处理信息和/或能使用存储的信息控制设备本身的行为或执行的任何和所有设备，而不管那些设备本质上是否为电子的、机械的、逻辑的、或虚拟的。

B. 传统的基于文件的存储

在当今大多数计算机系统中，“文件”是可存储信息的单元，它可包括硬件/软件接口系统和应用程序、数据集等。在所有现代硬件/软件接口系统中（Windows, Unix, Linux, MacOS, 虚拟机系统等），文件是能由硬件/软件接口系统处理的基本的分立（可存储和可检索）信息单元。文件组通常被组织成“文件夹”。在 Microsoft Windows、Macintosh OS 和其它硬件/软件接口系统中，文件夹是能作为单个信息单元被检索、移动和处理的文件的集合。这些文件夹进而被组织成称为“目录”（在后面详细讨论）的基于树的分层排列。在如 Dos、z/OS 和大多数基于 Unix 的操作系统的其它硬件/软件接口系统中，术语“目录”和/或“文件夹”是可互换使用的，早期的 Apple 计算机系统（如 Apple IIe）使用术语“类别”来代替目录；然而在这里使用时，所有这些术语被看成是同义语并可互换使用，并旨在还包括对分层信息存储结构及其文件夹和文件组件的所有其它等价术语的引用。

传统上，目录（又名文件夹的目录）是基于树的分层结构，其中文件被组合成文件夹，文件夹进而按构成目录树的相对节点位置排列。例如，如图 2A 所示，基于 DOS 的文件系统的基本文件夹（或“根目录”）212 可包括多个文件夹 214，其每一个可以还包括另外的文件夹（如特定文件夹的“子文件夹”）216，而这些的每一个又包括另外的文件夹 218，直到无限。这些文件夹的每一个可具有一个或多个文件 220，虽然在硬件/软件接口系统级上，文件夹中的各个文件除了它们在树形分层结构中的位置外没有什么共同点。不奇怪，将文件组织到文件分层结构的方法间接地反映了用于存储这些文件的典型存储介质（如硬盘、软盘、CD-ROM 等）的物理组织。

除上述以外，每个文件夹是对其子文件夹和其文件的容器一即，每个文件夹拥有其子文件夹和文件。例如，当文件夹被硬件/软件接口系统删除时，该文件夹的子文件夹和文件也被删除(在每个子文件夹的情况下还递归地包括它自己的子文件夹和文件)。同样，每个文件一般只由一个文件夹拥有，并且虽然文件能被复制且副本位于不同的文件夹，文件的副本本身是不同且独立单元，它与原始文件无直接连接(如对原始文件的改变在硬件/软件接口系统级上不反映到副本文件)。因此在这方面，文件和文件夹在本质上是“物理的”，因为文件夹类似于物理容器来处理，而文件作为这些容器中不同且独立的物理元素来处理。

II. 用于组织、搜索和共享数据的新存储平台

本发明针对用于组织、搜索和共享数据的存储平台。本发明的存储平台在上文讨论的文件系统及数据库系统的种类之外扩展和拓宽了数据平台，并被设计成存储所有类型的数据，包括称为项目的新形式的数据。

A. 词汇表

在这里及在权利要求书中使用的术语有下列意义：

“项目”是能存储硬件/软件接口系统可访问的信息的单元，不象简单文件，它是具有由硬件/软件接口系统外壳展现给最终用户的所有对象共同支持的基本属性集的对象。项目还具对所有项目类型共同支持的属性和关系，包括允许引入新属性和关系的特征(在下面详细讨论)。

“操作系统”(OS)是担当应用程序和计算机硬件之间的中介的特殊程序。在大多数情况下，操作系统包括外壳和内核。

“硬件/软件接口系统”是软件、或硬件及软件的组合，它起着计算机系统的底层硬件组件和在计算机系统上执行的应用程序之间的接口的作用。硬件/软件接口系统通常包括(在某些实施例中只包括)操作系统。硬件/软件接口系统还能包括虚拟机管理器(VMM)、公用语言运行库(CLR)或其功能等效物，Java 虚拟机(JVM)或其功能等效物、或在计算机系统中代替操作系统或除操作系统外的其它软件组件。硬件/软件接口系统的目的是提供用户能执行应用程序的环境。任何硬件/软件接口系统的目地是使计算机系统便于使用，并以有效方式利用计算机硬

件。

B. 存储平台综述

参考图 3, 依照本发明存储平台 300 包括在数据库引擎 314 上实现的数据存储 302。在一个实施例中, 数据库引擎包括带有对象关系扩展的关系型数据库引擎。在一个实施例中, 关系型数据库引擎 314 包括 Microsoft SQL Server 关系型数据库引擎。

数据存储 302 实现支持数据的组织、搜索、共享、同步和安全的数据模型 304。在如模式 340 等模式中描述特定的数据类型, 并且存储平台 300 提供用于采用这些模式并用于扩展这些模式的工具 346, 这在后面详述。

在数据存储 302 中实现的改变跟踪机制 306 提供跟踪数据存储的改变的能力。数据存储 302 还提供安全能力 308 和升级/降级能力 310, 这些均在后文详述。数据存储 302 还提供一组应用程序编程接口 312, 以向利用该存储平台的其它存储平台组件和应用程序 (如应用程序 350a, 350b 和 350c) 展现数据存储 302 的能力。

本发明的存储平台还包括应用程序编程接口 (API) 322, 使如应用程序 350a, 350b, 和 350c 等应用程序能访问存储平台的所有上述功能并能访问在模式中描述的数据。应用程序能结合如 OLE DB API 324 和 Microsoft Windows Win 32 API 326 等其它 API 来使用存储平台 API 322。

本发明的存储平台 300 能向应用程序提供各种服务, 包括便于在用户或系统之间共享数据的同步服务 330。例如, 同步服务 330 允许与具有与数据存储 302 相同格式的其它数据存储 340 的互操作, 并访问具有其它格式的数据存储 342。存储平台 300 还提供允许数据存储 302 与如 Windows NTFS 文件系统 318 等现有文件系统的互操作的文件系统能力。

在至少某些实施例中, 存储平台 320 还能向应用程序提供另外的能力, 以允许对数据起作用并允许与其它系统的交互。这些能力可具体化为如 Info Agent 服务 334 和通知服务 332 等附加服务 328 的形式, 以及其它实用程序 336 的形式。

在至少某些实施例中, 存储平台以计算机系统的硬件/软件接口系统来实施, 或形成其完整的一部分。例如而非限制, 本发明的存储平台能用操作系统、虚拟机管理器 (VMM)、公用语言运行库 (CLR) 或其功能等效物、或 Java 虚拟机 (JVM) 或其功能等效物来实施, 或形成其完整的一部分。

通过其公用的存储基础和模式化的数据，本发明的存储平台使消费者、知识工人和企业能够更有效地进行应用程序的开发。它提供了丰富和可扩展的编程表面区域，不仅可得到其数据模型中固有的能力，还能包括和扩展现有文件系统和数据库访问方法。

在下述描述中及在各附图中，本发明的存储平台 300 可称作“WinFS”。然而使用此名字指存储平台仅是为了描述方便，并不试图作出任何限制。

C. 数据模型

本发明的存储平台 300 的数据存储 302 实现一种数据模型，它支持对驻留在存储中的数据的组织、搜索、共享、同步和安全。在本发明的数据模型中，“项目”是存储信息的基本单元。该数据模型提供一种机制，用于声明项目和项目的扩展、用于建立项目之间的关系、以及用于将项目组织到项目文件夹和类别中，这些将在下文中更充分描述。

该数据模型依赖于两个原语机制：类型和关系。类型是提供支配类型的实例的形式的格式的结构。格式被表达成属性的有序组。属性是给定类型的值或一组值的名字。例如，USPostalAddress（美国邮政地址）类型具有属性 Street（街道）、City（城市）、Zip（邮编）、State（州），其中 Street、City 和 State 是 String 类型，而 Zip 是 Int32 类型。Street 可以是多值（即一组值），允许地址对 Street 属性具有一个以上值。系统定义能在其它类型构造中使用的某些原语类型，包括 String（串）、Binary（二进制）、Boolean（布尔）、Int16（16 位整数）、Int32（32 位整数）、Int64（64 位整数）、Single（单精度）、Double（双精度）、Byte（字节）、DateTime（日期时间）、Decimal（十进制）和 GUID。可使用任何原语类型（带有下面注释的某些限制）或任何构造的类型来定义类型的属性。例如，Location（位置）类型可被定义为具有属性 Coordinate（座标）和 Address（地址），其中 Address 属性是上述类型 USPostalAddress。属性也可以是必需的或可任选的。

关系可被声明并表示两个类型的实例集之间的映射。例如，可以在 Person（个人）类型和 Location 类型之间声明的关系，称为 LivesAt（生活在），它确定什么人生活在什么位置。关系具有名称和两个端点，即源端点和目标端点。关系也可具有属性的有序集。源端点及目标端点均具有名称和类型。例如，LivesAt 关系

具有称为类型 Person 的 Occupant (居民) 的源和称为类型 Location 的 Dwelling (住房) 的目标，且此外具有属性 StartDate (起始日期) 和 EndDate (终止日期)，表示该居民生活在该住房的时间段。注意，随时间推移，个人能生活在多个住房，且住房可有多个居民，所以放置 StartDate 和 EndDate 信息的最可能的地方是在关系本身处。

关系定义了在由作为端点类型给出的类型约束的实例之间的映射。例如 LivesAt 关系不能是其中 Automobile (汽车) 是 Occupant (居民) 的关系，因为 Automobile 不是 Person。

数据模型允许定义类型间的子类型—超类型关系。也称为基本类型 (BaseType) 关系的子类型—超类型关系以如下方式定义，若类型 A 是类型 B 的基本类型，则情况必须是 B 的每个实例也是 A 的实例。另一种表达的方法是符合 B 的每个实例也必须符合 A。例如，若 A 具有 String 类型的属性 Name (名字)，而 B 具有 Int16 类型的属性 Age (年龄)，则得出，B 的任何实例必须兼有 Name 和 Age。类型的分层结构可被设想成在根上带有单个超类型的树。根的分枝提供第一级子类型，该级分枝提供第二级子类型，依此类推，直到本身不再具有任何子类型的叶端 (leaf-most) 子类型。树不限于统一深度，但不能包含任何回路。给定的类型可具有零个或多个子类型和零个或一个超类型。给定实例可最多符合一个类型以及该类型的超类型。换言之，对树中任一级处给定的实例，该实例最多可符合该级上的一个子类型。

如果类型的实例必须也是该类型的子类型的实例，则该类型可被认为是抽象。

1. 项目

项目是可存储信息的单元，不象简单的文件，它是具有由存储平台向最终用户或应用程序展现的所有对象共同支持的基本属性集的对象。项目也具有所有项目类型共同支持的属性和关系，包括如下所述允许引入新的属性和关系的特征。

项目是公用操作的对象，如拷贝、删除、移动、打开、打印、备份、恢复、复制等。项目是能被存储和检索的单元，且由存储平台处理的可存储信息的所有形式作为项目、项目的属性、或项目之间的关系存在，其每一个在下面更详细讨论。

项目旨在表示现实的且容易理解的数据单元，如 Contacts (联系人)、People (人)、Services (服务)、Locations (位置)、(各种类型的) Documents (文档)

等。图 5A 是示出项目的结构的框图。该项目的不合格名是“Location”。该项目的合格名是“Core.Location”，它表明此项目结构被定义成核心（Core）模式中的特定类型的项目（核心模式在下面详细讨论）。

Location 项目具有多个属性，包括 EAddress（电子邮件地址）、MetropolitanRegion（都市地区）、Neighborhood（街坊）和 PostalAddress（邮政地址）。每个项目的特定类型属性紧跟属性名表示，并用冒号（“：“）与属性名分开。在类型名的右边，对该属性类型允许的值的数量在方括号（“[]”）之间表示，其中冒号（“：“）右边的星号（“*”）表示未规定的和/或无限制的数量（“许多”）。冒号右边的“1”表明最多一个值。冒号左边的零（“0”）表明该属性是可任选的（可以完全没有值）。冒号左边的“1”表明必须至少有一个值（该属性是必须的）。Neighborhood 和 MetropolitanRegion 均是“nvarchar”类型（或等效类型），它是预定义的数据类型或“简单类型”（这里用缺少大写来表示）。然而 EAddress 和 PostalAddress 分别是类型 EAddress 和 PostalAddress 的已定义类型或“复杂类型”（这里用大写标记）的属性。复杂类型是从一个或多个简单数据类型和/或从其它复杂类型导出的类型。项目的属性的复杂类型还构成“嵌套元素”，因为复杂类型的细节嵌套入直接项目中以定义其属性，而涉及这些复杂类型的信息用具有这些属性的项目来维持（在该项目的边界内，如后面讨论）。类型的这些概念是众知的，且容易被本领域的技术人员理解。

图 5B 是示出复杂属性类型 PostalAddress 和 EAddress 的框图。PostalAddress 属性类型定义属性类型 PostalAddress 的项目可期望有零个或一个 City（城市）值、零个或一个 CountryCode（国家代码）值、零个或一个 MailStop（信箱代码）值、和任何数量（零到许多）PostalAddress 类型等等。以此方式，定义了项目中的特定属性的数据的形状。EAddress 属性类型如所示类似地定义。虽然这里在本申请中可任选地使用，表示 Location 项目中复杂类型的另一方法是用其中列出的每个复杂类型的各个属性得出该项目。图 5C 是示出 Location 项目的框图，在其中进一步描述其复杂类型。然而应该理解，在图 5C 中 Location 项目的另一种表示恰是对图 5A 中示出的同一个项目。本发明的存储平台还允许子类型化，从而一个属性类型是另一个的子类型（其中一个属性类继承另一个父属性类型的属性）。

类似于但不同于属性及它们的属性类型，项目继承性地表示其自己的 Item（项目）类型，它也是子分类的主题。换言之，本发明的若干实施例中的存储平台允许一个项目是另一个项目的子类型（从而一个项目继承另一个父项目的属性）。此外，

对本发明的各种实施例，每个项目是“Item”项目类型的子类型，后者是在基础模式中找到的第一且基本的项目类型（基础模式也在后面详细讨论）。图 6A 示出一项目（在此实例中为 Location 项目）为在基础模式中找到的 Item 项目类型的子类型。在此图中，箭头表示 Location 项目（与所有其它项目一样）是 Item 项目类型的子类型。作为从中导出所有其它项目的基本项目的 Item 项目类型具有若干如 ItemId（项目 ID）等重要属性和各种时间标记，从而定义了操作系统中所有项目的标准属性。在本图中，Item 项目类型的这些属性被 Location 所继承，并从而成为 Location 的属性。

表示从 Item 项目类型继承的 Location 项目中属性的另一种方法是用来自其中列出的父项目的每个属性类型的各个属性得出 Location。图 6B 是示出 Location 项目的框图，其中除了其直接属性外描述其继承的类型。应注意和理解，此项目是图 5A 中示出的同一项目，虽然在本图中，Location 用所有其属性示出，包括直接属性（在本图及图 5A 中示出）和继承属性（在本图中示出但未在图 5A 中示出）（而在图 5A 中，通过用箭头示出 Location 项目是 Item 项目类型的子类型来引用这些属性）。

项目是独立的对象，因而若删除一项目，也删除项目的所有直接和继承的属性。类似地，当检索一项目时，接收到的是该项目及其所有直接和继承的属性（包括涉及其复杂属性类型的信息）。本发明的某些实施例可使人们能在检索特定项目时请求属性的子集；然而对许多那样的实施例默认的是在检索时向项目提供所有其直接和继承的属性。此外，项目的属性也能通过添加新的属性到该项目的类型的现有属性而加以扩展。这些“扩展”其后是该项目的真实属性，且该项目类型的子类型可自动地包括扩展属性。

项目的“边界”由其属性（包括复杂属性类型、扩展等）来表示。项目的边界也表示在项目上执行的操作的限制，包括复制、删除、移动、创建等。例如在本发明的若干实施例中，当复制项目时，在该项目边界之内的所有内容也被复制。对每个项目，边界包括下列：

- 项目的项目类型，且若该项目是另一项目的子类型（如在所有项目从基础模式的单个项目和项目类型导出的本发明的若干实施例的情况下），是任何适用的子类型信息（即涉及父项目类型的信息）。若要复制的原始项目是另一项目的子类型，该副本也能是该同一项目的子类型。

- 项目的复杂类型属性和扩展（如果有的话）。若原始项目具有复杂类型（原来的或扩展的）的属性，副本也能具有同一复杂类型。
- 在“所有权关系”上的项目的记录，即，本项目（“拥有项目”）拥有什么其它项目（“目录项目”）的项目拥有列表。这特别关系到下面充分讨论的项目文件夹和下面说到的所有项目必须至少属于一个项目文件夹的规则。此外，关于嵌入项目（下面更充分讨论），嵌入项目被认为是其中嵌入如复制、删除等操作的项目的一部分。

2. 项目标识

在全局项目空间中用 ItemID 唯一地标识项目。Base.Item 类型定义了存储该项目身份的类型 GUID 的字段 ItemID。一个项目必须在数据存储 302 中只有一个身份。

a) 项目引用

项目引用是包含定位和标识项目的信息的数据结构。在该数据模型中，定义名为 ItemReference（项目引用）的抽象类型，从中导出所有项目引用类型。ItemReference 类型定义了名为 Resolve（解析）的虚拟方法。Resolve 方法解析 ItemReference 并返回一项目。此方法被 ItemReference 的具体子类型所覆盖，后者实现给定一引用时检索项目的功能。调用 Resolve 方法作为存储平台 API 322 的一部分。

(1) ItemIDReference

ItemIDReference（项目 ID 引用）是 ItemReference 的子类型。它定义了 Locator（定位器）和 ItemID 字段。Locator 字段命名（即标识）项目域。它由能将 Locator 的值解析到项目域的定位器解析方法来处理。ItemID 字段是 ItemID 类型。

(2) ItemPathReference

ItemPathReference（项目路径引用）是定义 Locator 和 Path（路径）字段的 ItemReference 的特殊化。Locator 字段标识项目域。它由能将 Locator 的值解析到项目域的定位器解析方法来处理。Path 字段包含以由 Locator 提供的项目域为根的存储平台名字空间中的（相对）路径。

不能在集合运算中使用此类引用。引用一般必须通过路径解析过程来解析。
存储平台 API 322 的 Resolve 方法提供此功能。

b) 引用类型分层结构

上面讨论的引用形式通过图 11 示出的引用类型分层结构来表示。从这些类型继承的其它引用类型能在模式中定义。它们能在关系声明中用作目标字段的类型。

3. 项目文件夹和类别

如下面将更充分讨论的，项目组能被组织成称为项目文件夹（不要与文件的文件夹混淆）的特殊项目。然而不象大多数文件系统，一个项目可属于多个项目文件夹，使得当项目在一个项目文件夹中被访问和修订时，此修订的项目随后能直接从另一项目文件夹访问。实质上，虽然对一个项目的访问可从不同的项目文件夹发生，事实上真正访问的是同一个项目。然而，项目文件夹不必拥有其所有成员项目，或简单地结合其它文件夹共同拥有项目，使得一个项目文件夹的删除不必要导致项目的删除。然而在本发明的若干实施例中，一个项目必须至少属于一个项目文件夹，使得如果特定项目的唯一项目文件夹被删除，则对某些实施例，该项目被自动被删除，或在另外实施例中，该项目自动地成为默认项目文件夹的成员（例如，“Trash Can（垃圾箱）”项目文件夹在概念上类似于在各种基于文件和文件夹的系统中使用的类似名字文件夹。）

如下面更充分讨论的，项目也可属于基于共同描述的特征的类别，特征如：
(a) 项目类型（或类型），(b) 特定的直接或继承的属性（或属性），或(c) 对应于项目属性的特定值（或值）。例如，包括个人联系人信息的特定属性的项目可自动属于 Contact（联系人）类别，具有联系人信息属性的任何项目也自动属于此类别。同样，具有“New York City（纽约市）”值的位置属性的任何项目可自动属于 NewYorkCity（纽约市）类别。

类别在概念上不同于项目文件夹之处在于，项目文件夹可包括互相关联的项目（即无共同的描述的特征），而在类别中的每个项目具有对该类别描述的共同类型、属性或值（“共同性”），正是这个共同性形成对它与该类别中其它项目或那些项目之间的关系的基础。此外，虽然在特定文件夹中的项目的成员资格基于该项目的任何特定方面不是强制的，然而对某些实施例，具有在分类上与一类别相关的共同性的所有项目在硬件/软件接口系统级上可自动地成为该类别的成员。概念上，

类别也能看作虚拟项目文件夹，其成员资格基于特定查询（如在数据库的上下文中）的结果，而满足此查询的条件（由类别的共同性确定）的项目应构成该类别的成员资格。

图 4 示出在本发明各实施例中在项目、项目文件夹和类别之间的结构关系。多个项目 402、404、406、408、410、412、414、416、418 和 420 是各个项目文件夹 422、424、426、428 和 430 的成员。某些项目属于一个以上项目文件夹，如项目 402 属于项目文件夹 422 和 424。某些项目，如项目 402、404、406、408、410 和 412 也是一个或多个类别 432、434 和 436 的成员，而其它项目，如项目 44、416、418 和 420 可以不属于任何类别（虽然这大部分不象在某些实施例中，其中具有任何属性自动暗示类别中的成员资格，因此在那样实施例中为了不是任何类别的成员，项目应完全地无特征）。与文件夹的分层结构相反，类别和项目文件夹均有更像如所示的有向图的结构。在任何情况下，项目、项目文件夹和类别都是项目（尽管是不同的项目类型）。

与文件、文件夹和目录相反，本发明的项目、项目文件夹和类别的特征在本质上不是“物理的”，因为它们不具有物理容器的概念上的等价性，因而项目可存在于一个以上那样的位置。项目存在于一个以上项目文件位置以及被组织成类别的能力提供了在硬件/软件接口级上增强和丰富程度的数据处理及存储结构能力，超越了在本领域中当前可得到的能力。

4. 模式

a) 基础模式

为了提供创建和使用的通用基础，本发明的存储平台的各实施例包括建立用于创建和组织项目及属性的概念性框架的基础模式。基础模式定义了某些特定类型的项目和属性，以及从中进一步导出子类型的这些特定基本类型的特征。使用此基础模式使程序员能在概念上将项目（及其各自的类型）与属性（及其各自的类型）加以区别。此外，基础模式列出所有项目可拥有的基本属性集，因为所有项目（及其对应的项目类型）是从基础模式的此基本项目（及其对应的项目类型）导出的。

如图 7 所示，对于本发明的若干实施例，基础模式定义三个顶层类型：Item（项目）、Extension（扩展）和 PropertyBase（属性基）。如图所示，通过此基本“Item”项目类型的属性定义了项目类型。相反，顶层属性类型“PropertyBase”没有预定

义的属性，仅是一个定位点，从中导出所有其它属性类型，并且所有导出的属性类型通过它互相联系（共同从单个属性类型导出）。Extension 类型属性定义该扩展扩展了哪个项目，并定义将一个扩展与另一个项目相区别的标识，因为一个项目可具有多个扩展。

ItemFolder（项目文件夹）是 Item 项目类型的子类型，除了从 Item 继承的属性外，它表征用于建立到其成员（如果有的话）的链接的关系，尽管 Identitykey（身份键）和 Property（属性）均是 PropertyBase 的子类型。CategoryRef（目录引用）进而是 IdentityKey 的子类型。

b) 核心模式

本发明的存储平台的各种实施例还包括为顶层项目类型结构提供概念框架的核心模式。图 8A 是示出核心模式中的项目的框图，而图 8B 是示出核心模式中属性类型的框图。在带不同扩展名 (*.com、*.exe、*.bat、*.sys 等) 的文件和在基于文件和文件夹系统中其它准则之间作出的区分是类似于核心模式的功能。在基于项目的硬件/软件接口系统中，核心模式定义了一组核心项目类型，它们直接（按项目类型）或间接地（按项目子类型）将所有项目特征化成基于项目的硬件/软件接口系统理解并能以预定或可预测的方式直接处理的一个或多个核心模式项目类型。预定义的项目类型反映了在基于项目的硬件/软件接口系统中最常用的项目，且因此由理解这些构成核心模式的预定义项目类型的基于项目的硬件/软件接口系统获取有效性级别。

在某些实施例中，核心模式是不可扩展的，即，没有另外的类型可直接从基础模式中的项目类型子分类，除非作为核心模式的一部分的特定的预定导出的项目类型。通过禁止对核心模式的扩展（即，通过禁止向核心模式添加新的项目），存储平台托管核心模式项目类型的使用，因为每个后续的项目类型必须是核心模式项目类型的子类型。此结构允许在保持具有一组预定的核心项目类型的益处的同时在定义另外项目类型时有合理程度的灵活性。

参考图 8A，对本发明的各种实施例，由核心模式支持的特定项目类型可包括下列的一个或多个：

- Category（类别）：此项目类型（及从中导出的子类型）的项目表示在基于项目的硬件/软件接口系统中的有效类别。
- Commodity（物品）：作为值的可标识事物的项目。

- Device (设备)：具有支持信息处理能力的逻辑结构的项目。
- Document (文档)：具有不能由基于项目的硬件/软件接口系统解释而相反由对应于文档类型的应用程序解释的内容的项目。
- Event (事件)：记录环境中某些发生事件的项目。
- Location (位置)：代表物理位置（如地理位置）的项目。
- Message (消息)：在两个或多个主体（下面定义）之间通信的项目。
- Principal (主体)：具有除 ItemId 之外的至少一个肯定可证实身份（如，个人、组织、组、家庭、作者、服务等的标识）的项目。
- Statement (语句)：具有关于环境的特定信息的项目，包括但不限于：策略、预订、凭证等。

类似地参考图 8B，由核心模式支持的特定属性类型可包括下列的一个或多个：

- Certificate (证书)（从基础模式中的基本 PropertyBase 类型导出）
- PrincipalIdentityKey (主体身份键)（从基础模式中的 IdentityKey 类型导出）
- PostalAddress (邮政地址)（从基础模式中 Property 类型导出）
- RichText (多信息文本)（从基础模式中 Property 类型导出）
- EAddress (电子邮件地址)（从基础模式中 Property 类型导出）
- IdentitySecurityPackage (身份安全包)（从基础模式中 Relationship 类型导出）
- RoleOccupancy (居民角色)（从基础模式中 Relationship 类型导出）
- BasicPresence (基础存在)（从基础模式中 Relationship 类型导出）

这些项目和属性按在图 8A 和图 8B 中列出的各自属性进一步描述。

5. 关系

关系是二元关系，其中一个项目被指定为源，另一个被指定为目标。源项目和目标项目通过关系相联系。源项目一般控制关系的生命周期。即，当源项目被删除，项目之间的关系也被删除。

关系被分类成：包含（Containment）和引用（Reference）关系。包含关系控制目标项目的生命周期，而引用关系不提供任何生命周期管理语义。图 12 示出关系分类的方式。

包含关系类型又被分类成持有 (Holding) 和嵌入 (Embedding) 关系。当对一个项目的所有持有关系被移除，该项目被删除。持有关系通过引用计数机制控制目标的生命周期。嵌入关系能够对复合项目建模，并能被看作排他的持有关系。一个项目能是一个或多个持有关系的目标；但一个项目只能是一个嵌入关系的目标。作为嵌入关系的目标的项目不能是任一其它持有或嵌入关系的目标。

引用关系不控制目标项目的生命周期。它们可以是悬挂的一目标项目可以不存在。引用关系能用于在全局项目名字空间的任何处（即，包括远程数据存储）建模对项目的引用。

获得项目不自动取得其关系，应用程序必须明确地请求项目的关系。此外，修改关系不修改源或目标项目；类似地，添加关系不影响源/目标项目。

a) 关系声明

显式的关系类型用下列元素定义：

- 在 Name (名字) 属性中指定关系名
- 下列之一的关系类型：持有、嵌入、引用。这是在 Type (类型) 属性中指定的。
- 源和目标端点。每个端点指定所引用项目的名和类型。
- 源端点字段一般是 ItemID 类型（未声明），而必须引用在与关系实例相同的数据存储中的项目。
- 对持有和嵌入关系，目标端点字段必须是 ItemIDReference 类型，且它必须引用在与关系实例相同的存储中的项目。对引用关系，目标端点能是任何 ItemReference 类型，并能引用在其它存储平台数据存储中的项目。
- 能可任选地声明标量或 PropertyBase 类型的一个或多个字段。这些字段能包含与该关系相关联的数据。
- 关系实例被存储在全局关系表中。
- 每个关系实例唯一地由组合（源 ItemID、关系 ID）标识。对所有源自给定项目的关系，在给定的源 ItemID 中关系 ID 是唯一的，而不管它们的类型。

源项目是关系的拥有者。而被指定为拥有者的项目控制关系的生命周期，关系本身和与它们相关的项目分开。存储平台 API 322 提供用于展现与项目相关联的关

系的机制。

这里是一个关系声明的例子。

```
<Relationship Name="Employment" BaseType="Reference" >
  <Source Name="Employee" ItemType="Contact.Person"/>
  <Target Name="Employer" ItemType="Contact.Organization"
    ReferenceType="ItemIDReference" />
  <Property Name="StartDate" Type="the storage
    platformTypes.DateTime" />
  <Property Name="EndDate" Type="the storage
    platformTypes.DateTime" />
  <Property Name="Office" Type="the storage
    platformTypes.DateTime" />
</Relationship>
```

这是引用关系的例子。若由源引用所引用的个人项目不存在，则不能创建该关系。而且若该个人项目被删除，在个人和组织之间的关系实例也被删除。然而若组织项目被删除，关系不被删除，而它是悬挂的。

b) 持有关系

持有关系用于基于目标项目的生命周期管理来对引用计数建模。

项目可以是项目的零个或多个关系的源端点。不是嵌入项目的项目可以是在一个或多个持有关系中的目标。

目标端点引用类型必须是 ItemIDReference，且它必须引用在与关系实例相同的存储中的项目。

持有关系实施目标端点的生命周期管理。持有关系实例和作为目标的项目的创建是原子操作。可以创建将同一项目作为目标的另外的持有关系实例。当具有给定项目作为目标端点的最后一个持有关系实例被删除时，该项目也被删除。

在关系声明中指定的端点项目的类型一般在创建该关系的实例时强制。在关系建立之后端点项目的类型不能改变。

持有关系在形成项目的名字空间中起着关键作用。它们包含“Name”属性，它定义目标项目相对于源项目的名字。对所有源自给定项目的持有关系，相对名字是唯一的。从根项目开始到给定项目的相对名字的有序列表形成该项目的全名。

持有关系形成一有向非循环图（DAG）。在创建持有关系时，系统确保不产生回路，从而确保项目的名字空间形成 DAG。

虽然持有关系控制目标项目的生命周期，它不控制目标端点项目的操作的一致

性。目标项目在操作上独立于通过持有关系拥有它的项目。在作为持有关系的源的项目上的复制、移动、备份和其它操作不影响作为同一关系的目标的项目—例如，备份文件夹项目不自动地备份该文件夹中所有项目（FolderMember（文件夹成员）关系中的目标）。

下面是持有关系的例子：

```
<Relationship Name="FolderMembers" BaseType="Holding">
    <Source Name="Folder" ItemType="Base.Folder"/>
    <Target Name="Item" ItemType="Base.Item"
        ReferenceType="ItemIDReference" />
</Relationship>
```

FolderMember 关系使文件夹的概念成为项目的类属集合。

c) 嵌入关系

嵌入关系将目标项目的生命周期的排外控制的概念模型化。它们允许合成项目的概念。

嵌入关系实例和作为目标的项目的创建是原子操作。项目能是零个或多个嵌入关系的源。然而一个项目能是一个且仅是一个嵌入关系的目标。作为嵌入关系的目标的项目不能是持有关系的目标。

目标端点引用类型必须是 ItemIDReference，且它必须引用在与关系实例相同数据存储中的项目。

在关系声明中指定的端点项目的类型一般在创建该关系的实例时强制。在关系建立之后端点项目的类型不能改变。

嵌入关系控制该目标端点的操作一致性。例如，串行化项目的操作可包括串行化所有源自该项目及所有其目标的所有嵌入关系；复制项目也复制所有其嵌入项目。

下面是示例声明：

```
<Relationship Name="ArchiveMembers" BaseType="Embedding">
    <Source Name="Archive" ItemType="Zip.Archive"/>
    <Target Name="Member" ItemType="Base.Item"
        ReferenceType="ItemIDReference" />
    <Property Name="ZipSize" Type="the storage
platformTypes.bigint" />
    <Property Name="SizeReduction" Type="the storage
platformTypes.float" />
</Relationship>
```

d) 引用关系

引用关系不控制它引用的项目的生命周期。尤其是，引用关系不保证目标的存在，也不保证目标的类型如关系声明中指定的那样。这意味着引用关系能是悬挂的。而且引用关系能引用在其它数据存储中的项目。引用关系能看作类似于网页上的链接的概念。

下面是引用关系声明的例子：

```
<Relationship Name="DocumentAuthor" BaseType="Reference">
  <Source ItemType="Document"
    ItemType="Base.Document"/>
  <Target ItemType="Author" ItemType="Base.Author"
    ReferenceType="ItemIDReference" />
  <Property Type="Role" Type="Core.CategoryRef" />
  <Property Type="DisplayName" Type="the storage
    platformTypes.nvarchar(256)" />
</Relationship>
```

在目标的端点允许任何引用类型。参与引用关系的项目可以是任何项目类型。

引用关系用于对项目之间的大多数非生命周期管理关系建模。因为不强制目标的存在，引用关系便于对松散耦合的关系建模。引用关系能用于在其它存储，包括在其它计算机上的存储中的目标项目。

e) 规则和约束

下列附加规则和约束应用于关系：

1. 项目必须是（仅一个嵌入关系）或（一个或多个持有关系）的目标。一个例外是根项目。项目能是零个或多个引用关系的目标。
2. 作为嵌入关系目标的项目不能是持有关系的源。它能是引用关系的源。
3. 项目若是从文件升级，则不能是持有关系的源。它能是嵌入关系和引用关系的源。
4. 从文件升级的项目不能是嵌入关系的目标。

f) 关系的排序

在至少一个实施例中，本发明的存储平台支持关系的排序。通过在基本关系定义中的名为“Order（排序）”的属性来完成排序。在 Order 字段中无唯一性约束。

不保证带有同一“Order”属性值的关系的次序，然而保证它们能排序在带较低“Order”值的关系之后并在带较高“Order”字段值的关系之前。

应用程序通过在组合 (SourceItem ID, RelationshipID, Order) 上排序得到默认次序的关系。源自给定项目的所有关系实例被排序成单个集合，而不管在集合中关系的类型。然而这保证给定类型（如，FolderMembers）的所有关系是给定项目的关系集合的已排序子集。

用于操纵关系的数据存储 API 312 实现一组支持关系的排序的操作。引入下列术语帮助解释那些操作：

RelFirst 是有序集合中带次序值 *OrdFirst* 的第一个关系；

RelLast 是有序集合中带次序值 *OrdLast* 的最后一个关系；

RelX 是集合中带次序值 *OrdX* 的给定关系；

RelPrev 是集合中最接近于 *RelX* 的带小于 *OrdX* 的次序值 *OrdPrev* 的关系；以及

RelNext 是集合中最接近于 *RelX* 的带大于 *OrdX* 的次序值 *OrdNext* 的关系。

InsertBeforeFirst(SourceItemID, Relationship)

插入关系作为集合中的第一个关系。新关系的“Order”属性的值可小于 *OrdFirst*。

InsertAfterLast(SourceItemID, Relationship)

插入关系作为集合中的最后一个关系。新关系的“Order”属性的值可大于 *OrdLast*。

InsertAt(SourceItemID, ord, Relationship)

插入带有对“Order”属性指定的值的关系。

InsertBefore(SourceItemID, ord, Relationship)

在带有给定次序值的关系之前插入该关系。新的关系可被分配“Order”值，它在 *OrdPrev* 和 *ord* 之间，但不包括这两个值。

InsertAfter(SourceItemID, ord, Relationship)

在带有给定次序值的关系之后插入该关系。新的关系可被分配“Order”值，它在 *ord* 和 *OrdNext* 之间，但不包括这两个值。

MoveBefore(SourceItemID, ord, Relationship)

将带给定关系 ID 的关系移动到带指定“Order”值的关系之前。关系可被分配新的“Order”值，它在 *OrdPrev* 和 *ord* 之间，但不包括这两个值。

MoveAfter(SourceItemID, ord, Relationship)

将带给定关系 ID 的关系移动到带指定“Order”值的关系之后。该关系可被分配新的次序值，它在 ord 和 OrdNext 之间，但不包括这两个值。

如前提到，每个项目必须是项目文件夹的成员。按照关系，每个项目必须与一项目文件夹有关系。在本发明的若干实施例中，某些关系由在诸项目之间存在的关系表示。

如对本发明的各实施例的实现，关系提供有向的二元关系，它由一个项目（源）延伸到另一项目（目标）。关系由源项目（延伸它的项目）拥有，因此若源被移除则关系被移除（如在源项目被删除时关系被删除）。此外在某些情况下，关系可共享（共同拥有的）目标项目的拥有权，且那样的拥有权仅可反映在关系的 IsOwned（被拥有）属性（或其等效属性）中（如图 7 对关系属性类型所示）。在这些实施例中，创建新的 IsOwned 关系自动递增该目标项目上的引用计数，而删除那样的关系可递减该目标项目上的引用计数。对这些特定实施例，若项目具有大于 0 的引用计数，则继续存在，若项目计数达到 0 则自动删除。再一次，项目文件夹是具有（或能具有）与其它项目的一组关系的项目，这些其它项目包括项目文件夹的成员资格。关系的其它实际实现是可能的，并由本发明构想来实现这里描述的功能。

不管实际的实现如何，关系是从一个对象到另一对象的可选择的连接。一个项目属于一个以上项目文件夹以及一个或多个类别，而不论这些项目、文件夹和类别是公有的或私有的能力是由给予基于项目的结构中的存在（或缺乏）的意义所确定的。这些逻辑关系是分配给一组关系的意义，而不论其专门用来实现这里所述功能的物理实现如何。逻辑关系是在项目及其文件夹或类别之间建立的（或相反），因为本质上项目文件夹和类别的每一个都是特定类型的项目。因此，能象其它项目一样地对项目文件夹和类别起作用（复制、添加到电子邮件消息中、嵌入文档等等，而无限制），而项目文件夹和类别能象其它项目一样使用相同的机制串行化和反串行化（导入和导出）。（例如在 XML 中，所有项目可具有串行化的格式，且此格同等地应用于项目文件夹、类别和项目）。

表示项目及其项目文件夹之间的关系的上述关系在逻辑上能从项目延伸到项目文件夹、从项目文件夹延伸到项目、或两者。逻辑上从项目延伸到项目文件夹的关系表明该项目文件夹对于该项目是公有的，并与该项目共享其成员资格信息；相反，缺乏从项目到项目文件夹的逻辑关系表明该项目文件夹对该项目是私有的，且不与该项目共享其成员资格信息。类似地，逻辑上从项目文件夹延伸到项目的关系

表明该项目是公有的，并可与该项目文件夹共享，而缺乏从项目文件夹延伸到项目的逻辑关系表明该项目是私有的且不能共享。因此，当向其它系统导出项目文件夹时，它是“公有”的项目，它在新的环境中共享，且当项目搜索其项目文件夹寻找其它可共享的项目时，它是“公有”的项目文件夹，它向该项目提供关于属于它的可共享项目的信息。

图 9 是示出项目文件夹（它本身也是项目）、其成员项目、和项目文件夹及其成员项目之间互连关系的框图。项目文件夹 900 具有多个项目 902、904 和 906 作为其成员。项目文件夹 900 具有从它本身到项目 902 的关系 912，它表明项目 902 是公有的，且能与项目文件夹 900、其成员 904 和 906、和任何可访问项目文件夹 900 的任何其它项目文件夹、类别、或项目（未示出）共享。然而从项目 902 到项目文件夹项目 900 没有关系，这表明项目文件夹 900 对项目 902 是私有的，且不与项目 902 共享其成员资格信息。另一方面，项目 904 确实具有从它本身到项目文件夹 900 的关系 924，这表明项目文件夹 900 是公有的，且与项目 904 共享其成员资格信息。然而没有从项目文件夹 900 到项目 904 的关系，这表明项目 904 是私有的，且不与项目文件夹 900、其另外的成员 902、906、及可访问项目文件夹 900 的任何其它项目文件夹、类别、或项目（未示出）共享。与其到项目 902 和 904 的关系（或没有这些关系）相反，项目文件夹 900 具有从其本身到项目 906 的关系 916，且项目 906 具有回到项目文件夹 900 的关系 926，一起表明项目 906 是公有的，且能对项目文件夹 900、其成员 902 和 904、和可访问项目文件夹 900 的任何其它项目文件夹、类别、或项目（未示出）共享，且项目文件夹 900 是公有的，并与项目 906 共享其成员资格信息。

如前讨论，在项目文件夹中的项目不需要共享共同性，因为项目文件夹未被“描述”。另一方面，类别由对所有其成员项目共同的共同性描述。因此，类别的成员资格固有地限于具有所描述的共同性的项目，且在某些实施例中，满足类别的描述的所有项目自动地成为该类别的成员。因此，尽管项目文件夹允许由其成员资格来表示不重要的类型结构，类别基于定义的共同性来允许成员资格。

当然，类别描述本质上是逻辑的，因而类别可通过类型、属性和/或值的任何逻辑表示来描述。例如，对类别的逻辑表示可以是其成员资格，以包括具有两个属性之一或两者的项目。若这些对类别描述的属性是“A”和“B”，则该类别的成员资格可包括具有属性 A 而没有 B 的项目、具有属性 B 而没有 A 的项目、及兼有属性 A 和 B 的项目。通过逻辑运算符“OR（或）”来描述属性的逻辑表示，其中

由类别描述成员组是具有属性 A OR B 的项目。如本领域的技术人员所理解的，也能使用类似的逻辑运算符（包括但不限于单独的“AND（和）”“XOR（异或）”和“NOT（非）”或其组合）来描述类别。

尽管在项目文件夹（未描述）和类别（已描述）之间有区别，但在本发明的许多实施例中，原则上到项目的类别关系及到类别的项目关系以上面对项目文件夹和项目的同样方法揭示。

图 10 是示出类别（其本身也是项目）、其成员项目、类别及其成员项目之间的互连关系的框图。类别 1000 具有多个项目 1002、1004 和 1006 作为成员，所有这些都共享由类别 1000 描述的共同的属性、值和类型 1008（共同性描述 1008'）的某个组合。类别 1000 具有从其本身到项目 1002 的关系，它表明项目 1002 是公有的，且能与类别 1000、其成员 1004 和 1006、以及可访问类别 1000 的任何其它类别、项目文件夹、或项目（未示出）共享。然而，没有从项目 1002 到类别 1000 的关系，这表明类别 1000 对项目 1002 是私有的，且不与项目 1002 共享成员资格信息。另一方面，项目 1004 的确具有从其本身到类别 1000 的关系 1024，这表明类别 1000 是公有的，且与项目 1004 共享其成员资格信息。然而，不存在从类别 1000 延伸到项目 1004 的关系，这表明项目 1004 是私有的，且不能与类别 1000、它的其它成员 1002 和 1006、以及可访问类别 1000 的任何其它类别、项目文件夹、或项目（未示出）共享。与它与项目 1002 和 1004 的关系（或没有此关系）相反，类别 1000 具有从其本身到项目 1006 的关系 1016，且项目 1006 具有回到类别 1000 的关系 1026，这一起表明项目 1006 是公有的，且能与类别 1000、其项目成员 1002 和 1004、以及可访问类别 1000 的任何其它类别、项目文件夹、或项目（未示出）共享，且类别 1000 是公有的，且与项目 1006 共享其成员资格信息。

最后，由于类别和项目文件夹本身是项目，且项目可以互相关系，类别可关系到项目文件夹，反之亦然，且在某些另外实施例中，类别、项目文件夹和项目可分别关系到其它类别、项目文件夹和项目。然而在各种实施例中，项目文件夹结构和/或类别结构在硬件/软件接口系统级上禁止包含回路。当项目文件夹和类别结构类似于有向图时，禁止回路的实施例类似于有向非循环图（DAG），根据图论领域的数学定义，DAG 是其中没有路径在同一顶点开始与终止的有向图。

6. 可扩展性

如上所述，本存储平台旨在提供初始模式集 340。然而，至少在某些实施例中，该存储平台还允许包括独立软件分销商（ISV）等顾客创建新的模式 344（即新的项目和嵌套的元素类型）。本节讨论通过扩展在初始模式集 340 中定义的项目类型和嵌套的元素类型（或简称“元素”类型）着眼于创建该模式的机制。

较佳地，项目和嵌套元素类型的初始组的扩展如下约束：

允许 ISV 引入新的项目类型，即子类型 Base.Item；

允许 ISV 引入新的嵌套元素类型，即子类型 Base.NestedElement；

允许 ISV 引入新的扩展，即子类型 Base.NestedElement；但

ISV 不能子类型化由存储平台的初始模式集 340 定义的任何类型（项目、嵌入元素、或扩展类型）。

由于由存储平台的初始模式组定义的项目类型或嵌入元素类型可能完全匹配 ISV 应用程序的需要，必须允许 ISV 定制该类型。这就考虑了扩展的概念。扩展是强类型化的实例，但是（a）它们不能独立存在，以及（b）它们必须附属于项目或嵌套元素。

除了解决对模式可扩展性的需要之外，扩展还旨在解决“多类型化”问题。在某些实施例中，因为存储平台可能不支持多继承性或重叠子类型，应用程序可以使用扩展作为模型化重叠类型实例（如文档既是合法文档又是安全文档）的方法。

a) 项目扩展

为提供项目的可扩展性，数据模型还定义名为 Base.Extension 的抽象类型。这是扩展类型的分层结构的根类型。应用程序可以子类型化 Base.Extension，以创建特定的扩展类型。

在基础模式中如下定义 Base.Extension 类型：

```
<Type Name="Base.Extension" IsAbstract="True">
  <Property Name="ItemID"
    Type="the storage platformTypes.uniqueidentified"
    Nullable="false"
    MultiValued="false"/>
  <Property Name="ExtensionID"
    Type="the storage platformTypes.uniqueidentified"
    Nullable="false"
    MultiValued="false"/>
</Type>
```

ItemID 字段包含与该扩展关联的项目的 ItemID。带此 ItemID 的项目必须存在。若带给定 ItemID 的项目不存在，则不能创建扩展。当项目被删除，带同一 ItemID

的所有扩展被删除。元组 (ItemID, ExtensionID) 唯一地标识了扩展实例。

扩展类型的结构类似于项目类型的结构：

扩展类型具有字段；

字段可以是原语或嵌套元素类型；以及

扩展类型可被子分类。

下列限制应用于扩展类型

扩展不能是关系的源和目标；

扩展类型实例不能独立于项目存在；以及

扩展类型不能用作在存储平台类型定义中的字段类型

对能与给定的项目类型关联的扩展的类型没有约束。任何扩展类型允许扩展任何项目类型。当多个扩展实例被附加到一个项目，它们在结构和行为上彼此独立。

扩展实例被分别存储并从项目访问。所有扩展类型实例可从全局扩展视图访问。能组成一有效的查询，它将返回给定类型的扩展的所有实例，而不管它们与什么类型的项目相关联。存储平台 API 提供可存储、检索和修改项目扩展的编程模型。

扩展类型可以是使用存储平台的单个继承模型来子类型化的类型。从一个扩展类型导出创建新的扩展类型。一个扩展的结构或行为不能覆盖或替代项目类型分层结构的结构或行为。

类似于项目类型，扩展类型实例能通过与该扩展类型相关联的视图来直接访问。扩展的 ItemID 表明它们属于哪个项目，并能用于从全局项目视图检索对应的项目对象。

为操作一致性的目的，扩展被考虑成项目的一部分。复制/移动、备份/恢复和存储平台定义的其它常用操作可以在作为项目的一部分的扩展上操作。

考虑下述例子。在 Windows 类型集中定义 Contact 类型。

```
<Type Name="Contact" BaseType="Base.Item" >
  <Property Name="Name"
    Type="String"
    Nullable="false"
    MultiValued="false"/>
  <Property Name="Address"
    Type="Address"
    Nullable="true"
    MultiValued="false"/>
</Type>
```

CRM 应用程序开发者喜欢将 CRM 应用程序扩展附加到存储在存储平台中的联系人。应用程序开发者定义包含应用程序能处理的附加数据结构的 CRM 扩展。

```
<Type Name="CRMExtension" BaseType="Base.Extension" >
  <Property Name="CustomerID"
    Type="String"
    Nullable="false"
    MultiValued="false"/>
  ...
</Type>
```

HR 应用程序开发者希望也将附加数据附加到联系人。此数据独立于 CRM 应用程序数据。应用程序开发者还可创建扩展

```
<Type Name="HRExtension" EBaseType="Base.Extension" >
  <Property Name="EmployeeID"
    Type="String"
    Nullable="false"
    MultiValued="false"/>
  ...
</Type>
```

CRMExtension 和 HRExtension 是能附加到联系人项目的两个独立扩展。它们能彼此独立地创建和访问。

在上述例子中，CRMExtension 类型的字段和方法不能覆盖联系人分层结构的字段和方法。应注意，CRMExtension 类型的实例能被附加到不同于联系人的项目类型。

在检索联系人项目时，不自动地检索其项目扩展。给定联系人项目，可通过查询全局扩展视图以寻找带同一 ItemID 的扩展来访问其有关的项目扩展。

可通过 CRMExtension 类型视图来访问系统中所有的 CRMExtension 扩展，而不论它们属于什么项目。一个项目的所有项目扩展共享同一项目 ID。在上述例子中，Contact 项目实例和附加的 CRMExtension 和 HRExtension 实例共享同一 ItemID。

下表总结了在 Item（项目）、Extension（扩展）和 NestedElement（嵌套元素）类型之间的相似性和差别：

Item、ItemExtension 与 NestedElement

	Item	ItemExtension	NestedElement
项目 ID	具有自己的项目 id	共享项目的项目 id	不具有其自己的项目 id。嵌套元素是项目的一部分

存储	项目的分层结构存储在其自己表中	项目扩展分层结构存储在其自己表中	存储在项目中
查询/搜索	能查询项目表	能查询项目扩展表	通常只能在包含项目的上下文中查询
查询/搜索范围	能搜索一个项目类型的所有实例	能搜索一个项目扩展类型的所有实例	通常只能在单个(包含的)项目的嵌套元素类型实例中搜索
关系语义与项目的关联	能具有与项目的关系 能通过持有嵌入和软关系与其它项目相关	与项目扩展无关系 通常只能通过扩展来相关。扩展语义类	与嵌套元素无关系 通过字段来与项目相关。嵌套元素是项 目的一部分

b) 扩展 NestedElement 类型

嵌套元素类型不用与项目类型相同的机制来扩展。嵌套元素的扩展用与嵌套元素类型字段相同的机制来存储和访问。

数据模型定义了名为 Element (元素) 的嵌套元素类型的根：

```
<Type Name="Element"
      IsAbstract="True">
    <Property Name="ElementID"
              Type="the storage platformTypes.uniqueidentifier"
              Nullable="false"
              MultiValued="false"/>
</Type>
```

NestedElement 类型从此类型继承。NestedElement 元素类型另外定义一字段，它是多集元素。

```
<Type Name="NestedElement" BaseType="Base.Element"
      IsAbstract="True">
    <Property Name="Extensions"
              Type="Base.Element"
              Nullable="false"
              MultiValued="true"/>
</Type>
```

NestedElement 扩展在以下方面不同于项目扩展：

嵌套元素扩展不是扩展类型。它们不属于以 Base.Extension 类型为根的扩展类型分层结构。

嵌套元素扩展与该项目的其它字段一起存储，且不是全局可访问的—不能组

成检索给定扩展类型的所有实例的查询。

如存储其它嵌套元素（或项目）一样地存储这些扩展。如其它的嵌套集，NestedElement 扩展被存在 UDT 中。它们可通过嵌套元素类型的 Extension（扩展）字段来访问。

用于访问多值属性的集合接口也用于在类型扩展集上的访问和迭代。

下面的表总结和比较了 Item 扩展和 NestedElement 扩展。

Item 扩展与 NestedElement 扩展

	项目扩展	NestedElement 扩展
存储	项目扩展分层结构存入它自己的表	象嵌套元素那样存储
查询/搜索	能查询项目扩展表	通常只能在包含项目的上下文中查询
范围	能查询一个项目扩展类型的所有实例	通常只能在单个（包含）项目的嵌套元素类型实例中搜索
可编程性	需要特殊的扩展 API 和扩展表上的特殊查询	NestedElement 扩展类似嵌套元素的任何其它多值字段；使用正常的嵌套元素类型 API
行为	能关联行为	不允许行为（?）
关系语义	与项目扩展无关系	与 NestedElement 扩展无关系
项目 ID	共享项目的项目 id	不具有它自己的项目 id。 NestedElement 扩展是项目的一部分

D. 数据库引擎

如上提到，数据存储在数据库引擎上实现。在本实施例中，数据库引擎包括诸如 Microsoft SQL Server 引擎等实现 SQL 查询语言、带有对象关系扩展的关系型数据库引擎。本节按照本实施例描述了数据存储实现的数据模型到关系存储的映射，在逻辑 API 上提供由存储平台的客户机使用的信息。然而可以理解，当采用不同的数据库引擎时可采用不同的映射。确实，除了在关系型数据库引擎上实现存储平台概念数据模型之外，也可在其它类型数据库上实现，如面向对象和 XML 数据库。

面向对象（OO）数据库系统为编程语言对象（如 C++、Java）提供持续性和

事务。“项目”的存储平台概念可很好地映射到面向对象系统中的对象，虽然嵌入的集合必须添加给对象。类似继承性和嵌套元素类型等其它存储平台类型概念也映射到面向对象类型的系统。面向对象系统通常已经支持对象身份；因此，项目身份可被映射到对象身份。项目的行为（操作）很好地映射到对象方法。然而，面向对象的系统通常缺乏组织能力并在搜索方面很差。而且，面向对象的系统不提供对非结构化和半结构化数据的支持。为支持这里描述的完全存储平台数据模型，如关系、文件夹和扩展等概念需要添加到对象数据模型。此外，需要实现如升级、同步、通知和安全性等机制。

类似于面向对象的系统，基于 XSD（XML 模式定义）的 XML 数据库支持基于单继承类型的系统。本发明的项目类型系统可映射到 XSD 类型模型。XSD 也不提供对行为的支持。项目的 XSD 必须增添项目的行为。XML 数据库处理单个 XSD 文档并缺乏组织和拓宽搜索能力。如面向对象数据库那样，为支持这里描述的数据模型，如关系和文件夹等其它概念需要被结合到该 XML 数据库；而且需要实现如同步、通知和安全性等机制。

1. 使用 UDT 的数据存储实现

在本实施例中，在一个实施例中包括 Microsoft SQL Server 引擎的关系型数据库引擎 314 支持内建的标量类型。内建的标量类型是“原有”且“简单”的。它们是原有的意义是，用户不能定义他们自己的类型；它们是简单的意义是，用户不能封装复杂的结构。用户定义的类型（下文称为 UDT）通过使用户能通过定义复杂的结构化类型来扩展类型系统，提供了一种用于超过或超越原有的标量类型系统的类型可扩展性的机制。一旦由用户定义，UDT 能用于可以使用内建标量类型的类型系统中的任何地方。

按本发明的一个方面，存储平台模式被映射到数据库引擎存储中的 UDT 类。数据存储项目被映射到从 Base.Item 类型导出的 UDT 类。类似于项目，扩展也能映射到 UDT 类并使用继承。根扩展类型是 Base.Extension，从它导出所有扩展类型。

UDT 是 CLR 类，它具有状态（即数据字段）和行为（即例程）。使用任何受管语言（c#、VB.NET 等）定义 UDT。UDT 方法和操作符能在 T-SQL 中针对该类型的实例调用。UDT 能是：行中列的类型、T-SQL 中例程的参数的类型、或在 T-SQL 中变量的类型。

以下示例示出了 UDT 的基础。假定 MapLib.dll 具有称为 MapLib 的程序集。在该程序集中，在名字空间 BaseTypes 下，存在称为 Point 的类：

```
namespace BaseTypes
{
    public class Point
    {
        //返回离指定点的距离。
        public double Distance(Point p)
        {
            //返回点 p 和该点之间的距离
        }
        //该类中的其它内容
    }
}
```

以下 T-SQL 代码将类 Point 绑定到称为 Point 的 SQL Server UDT。第一步调用“Create Assembly”（创建程序集），它将 MapLib 程序集加载到数据库。第二步调用“Create Type”（创建类型），以创建用户定义类型“Point”，并将其绑定到受管类型 BaseTypes.Point：

```
CREATE ASSEMBLY Maplib
FROM '\\myserv\\share\\MapLib.dll'

go
CREATE TYPE Point
EXTERNAL NAME 'BaseTypes.Point'
go
```

一旦被创建，“Point”UDT 可用作表中的列，且该方法可以在 T-SQL 中如下所示地调用：

```
Create table Cities(
    Name varchar(20),
    State varchar(20),
    Location Point)

--检索城市到坐标(32, 23)的距离
Declare @p point(32, 23), @distance float

Select Location::Distance(@p)
From Cities
```

存储平台模式到 UDT 类的映射在高级别上完全是直接的。一般而言，存储平台模式被映射到 CLR 名字空间。存储平台类型被映射到 CLR 类。CLR 类的继承镜象了存储平台类型的继承，且存储平台属性被映射到 CLR 类属性。

图 29 所示的项目分层结构在本文档中用作一个示例。它示出了从其中导出所有项目类型的 Base.Item 类型，以及一组导出的项目类型（例如，Contact.Person 和

Contact.Employee) , 其中继承由箭头指示。

2. 项目映射

为了希望项目能够被全局地搜索，并在本实施例的关系型数据中支持继承以及类型可替代性，对在数据库存储中的项目存储的一种可能的实现是在带有类型 Base.Item 的列的单个表中存储所有项目。使用类型可替代性，能存储所有类型的项目，且可按使用 Yukon 的“is of (类型)”的操作符的项目类型和子类型来过滤搜索。

然而，由于在本实施例中牵涉到与这一方法相关联的额外开销，按顶级类型划分各项目，使得每个类型“家族”的项目存储到单独的表中。在此划分模式中，对每个直接从 Base.Item 继承的项目类型创建一个表。如上所述，继承下面这些的类型使用类型的可替代性存储在合适的类型家族表中。只有来自 Base.Item 的第一级继承被专门地处理。对于图 29 所示的示例项目分层结构，这可得到以下类型家族表：

```
create table Contact.[Table!Person](
    _Item Contact.Person not null
    {Change tracking information}
)

create table Doc.[Table!Document](
    _Item Doc.Document not null,
    {Change tracking information}
)
```

使用“阴影”表存储所有项目的全局可搜索属性的副本。此表可由存储平台 API 的 Update()方法维护，通过此方法作出所有数据的改变。不象类型家族表，此全局项目表只包含该项目的顶级标量属性，而不是全 UDT 项目对象。全局项目表的结构如下：

```
create table Base.[Table!Item](
    ItemID uniqueidentifier not null      constraint[PK_Clu_Item!ItemID] primary key clustered,
   TypeID uniqueidentifier not null,
    {Additional Properties of Base.Item},
    {Change tracking information}
)
```

全局项目表允许通过展现 ItemID 和 TypeID (类型 ID) 导航到存储在类型家族表中的项目对象。ItemID 通常唯一地标识数据存储中的项目。可使用这里未予描述的元数据将 TypeID 映射到类型名和包含该项目的视图。

由于通过其 ItemID 寻找项目在全局项目表的上下文及其它情况下都是常用操作，因此给定了项目的 ItemId，提供 GetItem() 函数来检索项目对象。该函数具有以下声明：

```
Base.Item Base.GetItem(uniqueidentifier ItemID)
```

为便于访问和尽可能地隐藏实现的细节，项目的所有查询可以对照在上述项目的表上构建的视图进行。具体而言，对每个项目类型针对合适类型的家族表创建视图。这些类型视图可选择相关联的类型，包括子类型的所有项目。为方便起见，除了 UDT 对象，视图能对包括继承字段的该类型的所有顶级域展现列。对图 29 所示的示例项目分层结构的视图如下：

```
create view Contact.Person as
    select _Item.ItemID, {Base.Item 的属性}, {Contact.Person 的属性}, {改变跟踪信息}, _Item
    from Contact.[Table!Person]

--注意 Contact.Employee 视图使用 “where” 判定
--来限制对 Contact.Employee 的实例的所找到的项目集
create view Contact.Employee as
    select _Item.ItemID, {Base.Item 的属性}, {Contact.Person 的属性}, {Contact.Employee 的属性},
        {改变跟踪信息}, cast(_Item as Contact.Employee)
    from Contact.[Table!Person]
    where _Item is of (Contact.Employee)

create view Doc.Document as
    select _Item.ItemID, {Base.Item 的属性}, {Doc.Document 的属性}, {改变跟踪信息}, _Item
    from Doc[Table!Document]

--注意 Doc.WordDocument 视图使用 “where” 判定
--来限制对 Doc.WordDocument 的实例的所找到的项目集
create view Doc.WordDocument as
    select _Item.ItemID, {Base.Item 的属性}, {Doc.Document 的属性}, {Doc.WordDocument 的属性},
        {改变跟踪信息}, cast(_Item as Doc.WordDocument)
    from Doc.[Table!Document]
    where _Item is of (Doc.WordDocument)
```

为完整性起见，视图也可在全局项目表上创建。该视图最初可展示与表相同的列：

```
create view Base.Item as
    select ItemID, TypeID, {Base.Item 的属性}, {改变跟踪信息}
    from Base.[Table!Item]
```

3. 扩展映射

扩展非常类似于项目，且具有某些相同要求。如支持继承性的另一根类型，扩

展受到存储中许多同样的考虑和折衷比较。为此，对扩展应用类似的类型家族映射，而不是单个表方法。当然，在其它实施例中，可使用单个表方法。

在本实施例中，扩展通过 ItemID 仅与一个项目关联，并包含在项目的上下文中唯一的 ExtensionID。扩展表具有以下定义：

```
create table Base.[Table!Extension](
    ItemID uniqueidentifier not null,
    ExtensionID uniqueidentifier not null,
   TypeID uniqueidentifier not null,
    {Base.Extension 的属性},
    {改变跟踪信息},

    constraint [PK_Clu_Extension!ItemID!ExtensionED]
        primary key clustered (ItemID asc, ExtensionID asc)
)
```

如同项目一样，给定包括 ItemID 和 ExtensionID 对的身份，可提供一函数用于检索扩展。该函数具有以下声明：

```
Base.Extension Base.GetExtension (uniqueidentifier ItemID, uniqueidentifier ExtensionID)
```

类似于项目类型视图，对每个扩展类型可创建视图。假定扩展分层结构与示例项目分层结构平行，具有以下类型：Base.Extension、Contact.PersonExtension、Contact.EmployeeExtension。可创建以下视图：

```
create view Base.Extension as
    select ItemID, ExtensionID, TypeID, {Base.Extension 的属性}, {改变跟踪信息}
    from Base.[Table!Extension]

create view Contact.[Extension!PersonExtension] as
    select _Extension.ItemID, _Extension.ExtensionID, {Base.Extension 的属性},
    {Contact.PersonExtension 的属性},
    {改变跟踪信息}, _Extension
    from Base.[Table!PersonExtension]

create view Contact.[Extension!EmployeeExtension] as
    select _Extension.ItemID, _Extension.ExtensionID, {Base.Extension 的属性},
    {Contact.PersonExtension 的属性},
    {Contact.EmployeeExtension 的属性}, {改变跟踪信息},
    cast(_Extension as Contact.EmployeeExtension)
    from Base.[Table!PersonExtension]
    where _Extension is of (Contact.EmployeeExtension)
```

4. 嵌套元素映射

嵌套元素是可嵌入到项目、扩展、关系、或其它嵌套元素以形成深嵌套结构的类型。类似于项目和扩展，嵌套元素作为 UDT 实现，但它们存储在项目和扩展中。

因此，嵌套元素没有超越它们的项目和扩展容器的存储映射。换言之，在系统中没有直接存储 NestedElement 类型的实例的表，且没有专门用于嵌套元素的视图。

5. 对象身份

在数据模型中的每一实体，即每个项目、扩展和关系具有唯一的键值。一个项目由其 ItemId 唯一地标识。扩展由合成键 (ItemId, ExtensionId) 唯一地标识。关系由合成键 (ItemId, RelationshipId) 标识。ItemId, ExtensionId 和 RelationshipId 均是 GUID 值。

6. SQL 对象命名

在数据存储中创建的所有对象可存储在从存储平台模式名字导出的 SQL 模式名字中。例如，存储平台基础模式（常称“Base”）可产生在 “[System.Storage]” SQL 模式中的类型，如 “[System.Storage].Item” 。生成的名字可用限定词加前缀以消除命名的冲突。在合适处可使用惊叹号 (!) 作为名字的每个逻辑部分的分隔符。下面的表概括了在数据存储用于对象的命名约定。与用于访问数据存储中的实例的修饰的命名约定一起列出每个模式元素（项目、扩展、关系和视图）。

对象	名字修饰	描述	例子
主项目搜索视图	Master!Item	在当前项目域中提供项目的综述	[System.Storage].[Master!Item]
类型化的项目搜索视图	项目类型	提供来自项目和任何父类型的所有属性数据	[AcmeCorp.Doc].[OfficeDoc]
主扩展搜索视图	Master!Extension	提供在当前项目域中所有扩展的综述	[System.Storage].[Master!Extension]
类型化的扩展搜索视图	Extension!扩展类型	对扩展提供所有属性数据	[AcmeCorp.Doc].[Extension!SlickyNote]
主关系视图	Master!Relationship	提供在当前项目域中所有关系的	[System.Storage].[Master!Relationship]

		综述	
关系视图	Relationship!关系名	提供所有与给定的关系相关联的数据	[AcmeCorp.Doc].[Relationship!AuthorsFromDocument]
视图	View!视图名	基于模式视图定义提供列/类型	[AcmeCorp.Doc].[View!DocumentTitles]

7. 列命名

当映射任一对象模型到存储时，由于与应用程序对象一起存储的附加信息，有可能发生命名冲突。为避免命名冲突，所有非类型的特定列（不直接映射到类型声明中的命名的属性的列）用下划线字符（_）加前缀。在本实施例中，下划线字符（_）不允许作为任何标识符属性的开始字符。此外，为统一在 CLR 和数据存储之间的命名，存储平台类型或模式元素的所有属性（关系等）应具有大写的第一字符。

8. 搜索视图

由存储平台提供视图，用于搜索存储的内容。对每个项目和扩展类型提供 SQL 视图。此外，提供视图以支持关系和视图（由数据模型定义）。所有 SQL 视图和在存储平台中的底层表是只读的。如下面将更充分描述的，使用存储平台 API 的 Update()方法可存储或改变数据。

在存储平台模式中明确定义的每个视图（由模式设计者定义，而非由存储平台自动地生成）可由命名的 SQL 视图[<模式名>].[View!<视图名>]来访问。例如，在模式“AcmePublisher.Books”中名为“BookSales”的视图可使用名字 “[AcmePublisher.Books].[View!BookSales]” 来访问。因为视图的输出格式在每一视图的基础上是自定义的（由定义视图的那一方提供的任意查询定义），列基于模式视图定义被直接映射。

存储平台数据存储中的所有 SQL 搜索视图使用列的下述排序约定：

1. 如 *ItemId*、*ElementId*、*RelationshipId* 等的视图结果的逻辑“键”列
2. 如 *TypeId* 等关于结果类型的元数据信息。
3. 如 *CreateVersion*（创建版本）、*UpdateVersion*（更新版本）等改变跟踪列

4. 类型专用的列（声明的类型的属性）

5. 类型专用的视图（家族视图）也包含返回对象的对象列

每个类型家族的成员可使用一系列项目视图来搜索，在数据存储中每个项目类型有一个视图。

a) 项目

每个项目搜索视图对特定类型或其子类型的项目的每个实例包含一行。例如，文档的视图能返回 Document（文档）、LegalDocument（合法文档）和 ReviewDocument（审阅文档）的实例。给定此例，能如图 28 所示那样概念化项目视图。

(1) 主项目搜索视图

存储平台数据存储的每个实例定义称为主项目视图（Master Item View）的特殊项目视图。此视图提供关于数据存储中每个项目的综述信息。视图对每个项目类型属性提供一列，其中一列描述项目的类型，若干列用于提供改变跟踪和同步信息。在数据存储中使用名字 “[System.Storage].[Master!Item]” 来标识主项目视图。

列	类型	描述
ItemId	ItemId	该项目的存储平台身份
_TypeId	TypeId	该项目的 _TypeId—标识该项目的确切类型并能用于使用元数据类别来检索关于类型的信息
RootItemId	ItemId	控制此项目的生命周期的第一个非嵌入先辈的ItemId
<全局改变跟踪>	...	全局改变跟踪信息
<项目属性>	无	对每个项目类型属性有一列

(2) 类型化的项目搜索视图

每个项目类型也具有搜索视图。尽管类似于根项目视图，但此视图还提供通过 “_Item” 列对项目对象的访问。在数据存储中使用名字[模式名].[项目类型名] 标识每个类型化的项目搜索视图。例如[AcmeCorp.Dod].[OfficeDoc]。

列	类型	描述
---	----	----

ItemId	ItemId	该项目的存储平台身份
<类型改变跟踪>	...	类型改变跟踪信息
<父属性>	<属性专用>	对每个父属性有一列
<项目属性>	<属性专用>	对此类型的每个排他属性有一列
Item	项目的 CLR 类型	CLR 对象—声明的项目的类型

b) 项目扩展

WinFs 存储中的所有项目扩展也可使用搜索视图来访问。

(1) 主扩展搜索视图

数据存储的每个实例定义一称为主扩展视图（Master Extension View）的特殊扩展视图。此视图提供关于数据存储中每个扩展的综述信息。该视图对每个扩展属性有一列，其中一列描述扩展的类型，而若干列用于提供改变跟踪和同步信息。使用名字 “[System.Storage].[Master!Extension]” 在数据存储中标识主扩展视图。

列	类型	描述
ItemId	ItemId	与此扩展关联的项目的存储平台身份
ExtensionId	ExtensionId (GUID)	此扩展实例的 id
_TypeId	TypeId	该扩展的 _TypeId—标识该扩展的确切类型，并能用于使用元数据类别来检索关于该扩展的信息
<全局改变跟踪>	...	全局改变跟踪信息
<扩展属性>	<属性专用>	对每个扩展类型属性有一列

(2) 类型化的扩展搜索视图

每个扩展类型还具有搜索视图。尽管类似于主扩展视图，但此视图还提供通过 _Extension 列对项目对象的访问。在数据存储中使用名字 [模式名].[Extension!扩展类型名] 标识每个类型化的扩展搜索视图。例如 [AcmeCorp.Doc].[Extension!OfficeDocExt]。

列	类型	描述

ItemId	ItemId	与此扩展关联的项目的存储平台身份
ExtensionId	ExtensionId(GUID)	此扩展实例的 Id
<类型改变跟踪>	...	类型改变跟踪信息
<父属性>	<属性专用>	对每个父属性有一列
<扩展属性>	<属性专用>	对每个此类型的排他属性有一列
Extension	扩展实例的 CLR 类型	CLR 对象—声明的扩展的类型

c) 嵌套的元素

所有嵌套的元素存储在项目、扩展或关系实例之中。因此，它们能通过查询合适的项目、扩展或关系搜索视图来访问。

d) 关系

如上讨论，关系形成在存储平台数据存储中各项目之间链接的基本单元。

(1) 主关系搜索视图

每个数据存储提供一主关系视图。此视图提供关于数据存储中所有关系实例的信息。在数据存储中使用名字 “[System.Storage].[Master!Relationship]” 来标识主关系视图。

列	类型	描述
ItemId	ItemId	源端点的身份 (ItemId)
RelationshipId	RelationshipId(GUID)	该关系实例的 id
_RelTypeId	RelationshipTypeId	该关系的 RelTypeId—使用元数据类别来标识该关系实例的类型
<全局改变跟踪>	...	全局改变跟踪信息
TargetItemReference	ItemReference	目标端点的身份
_Relationship	Relationship	对此实例的 Relationship 对象的实例

(2) 关系实例搜索视图

每个声明的关系也有返回该特定关系的所有实例的搜索视图。尽管类似于主关系视图，但此视图对该关系数据的每个属性提供命名的列。在数据存储中使用名

字 [模式名].[Relationship! 关系名] 来标识每个关系实例搜索视图。例如 [AcmeCorp.Doc].[Relationship!DocumentAuthor]。

列	类型	描述
ItemId	ItemId	源端点的身份 (ItemId)
RelationshipId	RelationshipId(GUID)	该关系实例的 id
<类型改变跟踪>	...	类型改变跟踪信息
TargetItemReference	ItemReference	目标端点的身份
<源名>	ItemId	源端点身份的命名属性 (ItemId 的别名)
<目标名>	ItemReference 或导出的类	目标端点身份的命名属性 (TargetItemReference 的别名和类型强制转换)
<关系属性>	<属性专用>	对每个关系定义的属性有一列
Relationship	关系实例的 CLR 类型	CLR 对象—声明关系的类型

9. 更新

存储平台数据存储中所有视图是只读的。为创建数据模型元素（项目、扩展或关系）的新实例，或更新现有的实例，必须使用存储平台 API 的 ProcessOperation 或 ProcessUpdategram 方法。ProcessOperation 方法是单个存储的过程，它是由消费细化要执行的动作的“操作”的数据存储定义的。ProcessUpdategram 方法是存储的过程，它采取称为“更新元素（updategram）”的一组有序的操作，它们共同细化要执行的一组动作。

操作格式是可扩展的，并提供在模式元素上的各种操作。某些常见操作包括：

1. 项目操作：

- a. CreateItem (在嵌入或持有关系的上下文中创建一新的项目)
- b. UpdateItem (更新一现有的项目)

2. 关系操作：

- a. CreateRelationship (创建引用或持有关系的实例)
- b. UpdateRelationship (更新一关系实例)

c. DeleteRelationship (移除一关系实例)

3. 扩展操作

a. CreateExtension (添加一扩展到现有的项目)

b. UpdateExtension (更新一现有的扩展)

c. DeleteExtension (删除一扩展)

10. 改变跟踪及墓碑

如下面更充分讨论的，由数据存储提供改变跟踪和墓碑服务。本节提供在数据存储中展现的改变跟踪信息的概述

a) 改变跟踪

由数据存储提供的每个搜索视图包含用于提供改变跟踪信息的列；那些列对所有项目、扩展和关系视图是公用的。由模式设计者明确地定义的存储平台模式视图不自动地提供改变跟踪信息—该信息是通过在其上构建视图本身的搜索视图来间接提供的。

对数据存储中的每个元素，可从两个地方得到改变跟踪信息：“主”元素视图和“类型化的”元素视图。例如，可从主项目视图 “[System.Storage].[Master!Item]” 和类型化的项目视图 [AcmeCorp.Document].[Document] 中得到关于 AcmeCorp.Document.Document 项目类型的改变跟踪信息。

(1) “主” 搜索视图中的改变跟踪

主搜索视图中的改变跟踪信息提供关于元素的创建和更新版本的信息、关于哪个同步伙伴创建该元素的信息、关于哪个同步伙伴最后一次更新该元素的信息、以及来自每个伙伴的用于创建和更新的版本号。用伙伴键来标识同步关系中的伙伴（下面描述）。类型 [System.Storage.Store].ChangeTrackingInfo 的名为 _ChangeTrackingInfo 的单个 UDT 对象包含所有这些信息。在 System.Storage 模式中定义类型。在项目、扩展和关系的所有全局搜索视图中可得到 _ChangeTrackingInfo。_ChangeTrackingInfo 的类型定义是：

```
<Type Name="ChangeTrackingInfo" BaseType="Base.NestedElement">
  <FieldProperty Name="CreationLocalTS"      Type="SqlTypes.SqlInt64"
    Nullable="False" />
  <FieldProperty Name="CreatingPartnerKey"    Type="SqlTypes.SqlInt32"
    Nullable="False" />
```

```

<FieldProperty Name="CreatingPartnerTS"          Type="SqlTypes.SqlInt64"
    Nullable="False" />
<FieldProperty Name="LastUpdateLocalTS"           Type="SqlTypes.SqlInt64"
    Nullable="False" />
<FieldProperty Name="LastUpdatingPartnerKey"      Type="SqlTypes.SqlInt32"
    Nullable="False" />
<FieldProperty Name="LastUpdatingPartnerTS"        Type="SqlTypes.SqlInt64"
    Nullable="False" />
</Type>

```

这些属性包含下述信息：

列	描述
CreationLocalTS	本地机器的创建时间标记
_CreatingPartnerKey	创建此实体的伙伴的 PartnerKey。若实体是本地创建的，这是本地机器的 PartnerKey
_CreatingPartnerTS	在对应于_CreatingPartnerKey 的伙伴处创建此实体的时间的时间标记
LastUpdateLocalTS	对应于本地机器的更新时间的本地时间标记
_LastUpdatePartnerKey	最后一次更新此实体的伙伴的 PartnerKey。若对该实体的最后一次更新在本地完成，则这是本地机器的 PartnerKey。
_LastUpdatePartnerTS	在对应于_LastUpdatingPartnerKey 的伙伴处更新此实体的时间的时间标记。

(2) “类型化的” 搜索视图中的改变跟踪

除了提供与全局搜索视图相同信息外，每个类型化的搜索视图提供记录在同步拓扑中每个元素的同步状态的附加信息。

列	类型	描述
<全局改变跟踪>	...	来自全局改变跟踪的信息
_ChangeUnitVersions	MultiSet<改变单元版本>	特定元素中的改变单元的版本号的描述
_ElementSyncMetadata	ElementSyncMetadata	关于只对同步运行库感兴趣的项目的附加版本无关元数据
VersionSyncMetadata	VersionSyncMetadata	关于只对同步运行库感兴趣的

		版本的附加版本专用元数据
--	--	--------------

b) 墓碑

数据存储为项目、扩展和关系提供墓碑信息。墓碑视图提供一个地方中有关活和墓碑实体两者（项目、扩展和关系）的信息。项目和扩展墓碑视图不提供对对应对象的访问，而关系墓碑视图提供对关系对象的访问（在墓碑关系的情况下关系对象为空）。

(1) 项目墓碑

通过视图[System.Storage].[Tombstone!item]从系统检索项目墓碑。

列	类型	描述
ItemId	ItemId	项目的身份
_TypeID	TypeId	项目的类型
<项目属性>	...	对所有项目定义的属性
_RootItemId	ItemId	包含此项目的第一非嵌入项目的ItemId
_ChangeTrackingInfo	ChangeTrackingInfo 类型的 CLR 实例	此项目的改变跟踪信息
_IsDeleted	BIT	这是标志，0是活项目，1是墓碑项目
_DeletionWallclock	UTCDATETIME	按删除项目的伙伴的 UTC 墙钟日期时间，若该项目是活的，它为空

(2) 扩展墓碑

使用视图[System.Storage].[Tombstone!Extension]从系统检索扩展墓碑。扩展改变跟踪信息类似于为项目提供的添加了 ExtensionId 属性的信息。

列	类型	描述
ItemId	ItemId	拥有该扩展的项目的身份
ExtensionId	ExtensionId	该扩展的 ExtensionId
_TypeID	TypeId	该扩展的类型

_ChangeTrackingInfo	ChangeTrackingInfo 类型的 CLR 实例	此扩展的改变跟踪信息
IsDeleted	BIT	这是标志, 0 是活项目, 1 是墓碑扩展
_DeletionWallclock	UTCDATETIME	按删除该扩展的伙伴的 UTC 墙钟日期 时间。若该扩展是活的, 它为空

(3) 关系墓碑

通过视图[System.Storage].[Tombstone!Relationship]从系统检索关系墓碑。关系墓碑信息类似于对扩展提供的信息。然而, 在关系实例的目标 ItemRef 上提供附加信息。此外, 还选择关系对象。

列	类型	描述
ItemId	ItemId	拥有该关系的项目的身份(关系的源端点的身份)
RelationshipId	RelationshipId	该关系的 RelationshipId
TypeID	TypeId	关系的类型
_ChangeTrackingInfo	ChangeTrackingInfo 类型的 CLR 实例	此关系的改变跟踪信息
IsDeleted	BIT	这是标志, 0 是活项目, 1 是墓碑扩展
_DeletionWallclock	UTCDATETIME	按删除该关系的伙伴的 UTC 墙钟日期 时间。若该关系是活的, 它为空
_Relationship	关系的 CLR 实例	这是活关系的关系对象, 对墓碑的关系 它为空
TargetItemReference	ItemReference	目标端点的身份

(4) 墓碑清除

为防止墓碑信息无限制地增长, 数据存储提供墓碑清除任务。此任务确定什么时候可以舍弃墓碑信息。该任务计算本地创建/更新版本的界限, 并随后通过舍弃所有更早的墓碑版本而截断墓碑信息。

11. 助手 API 和函数

基础映射还提供若干助手函数。提供这些函数以帮助在该数据模型上的公用操作。

a) 函数[System.Storage].GetItem

```
//给定 ItemId 返回一项目对象
Item GetItem (ItemId ItemId)
```

b) 函数[System.Storage].GetExtension

```
//给定 ItemId 和 ExtensionId 返回一扩展对象
Extension GetExtension (ItemId ItemId, ExtensionId ExtensionId)
```

c) 函数[System.Storage].GetRelationship

```
//给定 ItemId 和 RelationshipId 返回一关系对象
Relationship GetRelationship (ItemId ItemId, RelationshipId RelationshipId)
```

12. 元数据

有两类在存储中表示的元数据：实例元数据（项目的类型等），和类型元数据。

a) 模式元数据

模式元数据作为来自元模式的项目类型的实例存储在数据存储中。

b) 实例元数据

应用程序使用实例元数据来查询项目的类型，并寻找与项目相关联的扩展。给定项目的 ItemId，应用程序可查询全局项目视图，以返回该项目的类型，并使用此值来查询 Meta.Type 视图以返回关于该项目的声明的类型的信息。例如，

```
//对给定的项目实例返回元数据项目对象
SELECT m._Item AS metadataInfoObj
FROM [System.Storage].[Item] i INNER JOIN [Meta].[Type] m ON i._TypeId = m.ItemId
WHERE i.ItemId = @ItemId
```

E. 安全性

本节描述了依照一个实施例用于本发明的存储平台的安全模型。

1. 综述

依照本实施例，指定和强制实施存储平台的安全策略的粒度是在给定数据存储

中的项目上的各种操作级上；没有从整体中分离地保护项目的各部分的能力。该安全模型通过访问控制列表（ACL）指定了可以被授予或拒绝访问以在项目上执行这些操作的主体集。每一 ACL 是访问控制条目（ACE）的有序集合。

用于项目的安全策略可以完全由自由选择的访问控制策略和系统访问控制策略来描述。这些的每一个是一组 ACL。第一组（DACL）描述了由项目的所有者授予各种主体的自由选择的访问权限，而第二组 ACL 被称为 SACL（系统访问控制列表），它指定了当以某些方式操纵对象时如何完成系统监察。除这些之外，数据存储中的每一项目与对应于项目的所有者的 SID（所有者 SID）相关联。

用于在存储平台数据存储中组织项目的主要机制是包含分层结构的机制。包含分层结构是使用项目之间的持有关系来实现的。两个项目 A 和 B 之间被表达为“A 包含 B”的持有关系使得项目 A 能够影响项目 B 的生命周期。一般而言，数据存储中的项目在具有从另一项目到该项目的持有关系之前无法存在。除控制项目的生命周期之外，持有关系提供了传播项目的安全策略所必需的机制。

为每一项目指定的安全策略由两个部分组成—为该项目明确指定的部分以及在数据存储中从该项目的父项目继承的部分。任何项目的明确定义的安全策略由两个部分组成—支配对考虑中的项目的访问的部分，以及影响由包含分层结构中所有其子孙继承的安全策略的部分。由子孙继承的安全策略是由明确定义的策略和继承的策略来决定的。

由于安全策略是通过持有关系来传播的，且也可在任何项目处被覆盖，因此有必要指定如何确定项目的有效安全策略。在本实施例中，数据存储包含分层结构中的项目沿从存储的根到该项目的每一路径继承 ACL。

在对任何给定路径继承的 ACL 中，ACL 中的各 ACE 的排序确定了强制实时的最终安全策略。以下表示法用于描述 ACL 中 ACE 的排序。ACL 中由项目继承的 ACE 的排序是由以下两条规则来确定的—

第一条规则满足了在从包含分层结构的根到项目 I 的路径中从各项目继承的 ACE。从较接近的容器继承的 ACE 优先于从较远的容器继承的条目。直观上，这允许管理员能够覆盖从包含分层结构中较远处继承的 ACE。该规则如下：

对于项目 I 上所有继承的 ACL L

对于所有项目 I1、I2

对于 L 中所有 ACE A1 和 A2

I1 是 I2 的祖先，以及

I2 是 I3 的祖先，以及

A1 是从 I1 继承的 ACE，以及

A2 是从 I2 继承的 ACE

意味着

A2 在 L 中优先于 A1

第二条规则将拒绝对项目的访问的 ACE 排序在授予对项目的访问的 ACE 之前。

对于项目 I 上所有继承的 ACL L

对于所有项目 I1

对于 L 中所有 ACE A1 和 A2，

I1 是 I2 的祖先，以及

A1 是从 I1 继承的 ACCESS_DENIED_ACE，以及

A2 是从 I1 继承的 ACCESS_GRANTED_ACE

意味着

A1 在 L 中优先于 A2

在包含分层结构为树的情况下，从树的根到该项目只有一条路径，且该项目只有一个继承的 ACL。在这些情况下，按照其中 ACE 的相对排序，由项目继承的 ACL 与现有 Windows 安全模型中的文件（项目）继承的 ACL 相匹配。

然而，数据存储中的包含分层结构是有向非循环图（DAG），因为对项目允许多个持有关系。在这些情况下，从包含分层结构的根节点到项目有多条路径。由于项目沿每一条路径继承 ACL，因此每一项目与 ACL 的集合而非单个 ACL 相关联。注意，这与传统的文件系统模型不同，在该模型中，只有一个 ACL 与文件或文件夹相关联。

当包含分层结构是 DAG 而非树时，通常有两个方面要详细阐述。需要描述关于当项目从其父节点继承一个以上 ACL 时如何为其计算有效安全策略，并且如何组织和表示项目对于存储平台数据存储的安全模型的管理有直接的意义。

以下算法评估一给定主体对给定项目的访问权限。在整篇文档中，以下表示法用于描述与项目相关联的 ACL。

InheritedACLs(ItemId) — 由其项目身份为 ItemId 的项目从存储中其父项目继承的一组 ACL。

Explicit_ACL(ItemId) — 为其身份为 ItemID 的项目明确地定义的 ACL。

```

NTSTATUS
ACLAccesCheck(
    PSID          pOwnerSid,
    PDACL         pDacl,
    DWORD         DesiredAccess,
    HANDLE        ClientToken,
    PPRIVILEGE_SET pPrivilegeSet,
    DWORD         *pGrantedAccess)

```

如果期望的访问权限未被明确地拒绝，则上述例程返回 STATUS_SUCCESS，且 pGrantedAccess 确定用户期望的权限中的哪些由该指定的 ACL 授予。如果期望访问权限被明确地拒绝，则该例程返回 STATUS_ACCESS_DENIED。

```

NTSTATUS
ItemAccessCheck(
    OS_ITEMID      ItemId,
    DWORD          DesiredAccess,
    HANDLE         ClientToken,
    PPRIVILEGE_SET pPrivilegeSet)
{
    STATUS        Status;
    PDACL         pExplicitACL = NULL;
    PDACL         pInheritedACLs = NULL;
    DWORD         NumberOfInheritedACLs = 0;

    pExplicitACL = GetExplicitACLForItem(ItemId);
    GetInheritedACLsForItem(ItemId,&pInheritedACLs,&NumberOfInheritedACLs)

    Status = ACLAccessCheck(
        pOwnerSid,
        pExplicitACL,
        DesiredAccess,
        ClientToken,
        pPrivilegeSet,
        &GrantedAccess);

    if (Status != STATUS_SUCCESS)
        return Status;
    if (DesiredAccess == GrantedAccess)
        return STATUS_SUCCESS;
    for (
        i = 0;
        (i < NumberOfInheritedACLs && Status == STATUS_SUCCESS) ;
        i++ ) {
        GrantedAccessForACL = 0;

```

```

Status = ACLAccessCheck(
    pOwnerSid,
    pExplicitACL,
    DesiredAccess,
    ClientToken,
    pPrivilegeSet,
    &GrantedAccessForACL);
if (Status == STATUS_SUCCESS) {
    GrantedAccess |= GrantedAccessForACL;
}
}
If ((Status == STATUS_SUCCESS) &&
(GrantedAccess != DesiredAccess)) {
    Status = STATUS_ACCESS_DENIED;
}
return Status;
}

```

在任一项目处定义的安全策略的影响范围覆盖在数据存储上定义的包含分层结构中该项目的子孙。对于定义了显式策略的所有项目，则效果类似于定义由包含分层结构中其所有子孙所继承的策略。由所有子孙继承的有效 ACL 可通过取由该项目继承的 ACL，并将显式 ACL 中可继承的 ACE 添加到该 ACL 的开头来获得。这被称作与该项目相关联的可继承 ACL 集。

当根节点在文件夹项目处的包含分层结构中缺少安全性的显式指定时，文件夹的安全性指定应用于包含分层结构中该项目的所有子孙。因而，对其提供了显式安全策略指定的每一项目定义了类似地受保护的项目的区域，且对于该区域内的所有项目的有效 ACL 是该项目的可继承 ACL 集。在包含分层结构是树的情况下，这将完整地定义区域。如果每个区域都要与一号码相关联，那么仅将项目所属的区域随该项目一起包括在内就足够了。

然而，对于是 DAG 的包含分层结构，包含分层结构中有效安全策略改变所处的点一般由两类项目确定。第一类是已对其指定显式 ACL 的项目。通常，这些是包含分层结构中管理员显式地指定 ACL 的点。第二类是具有一个以上父节点，且各个父节点具有与其相关联的不同安全策略的项目。通常，这些是为卷指定的安全策略的汇合点的项目，并指示了一新安全策略的开始。

采用该定义，数据存储中的各项目落入两个类别之一—是类似地受保护的安全区域的根节点的那些项目，以及不是该根节点的那些项目。未定义安全区域的项目一般仅属于一个安全区域。如在树的情况下，项目的有效安全性可通过指定项目所

属的区域以及该项目来指定。这导致一种用于基于存储中各个类似地受保护的区域来管理存储平台数据存储的安全性的直截了当的模型。

2. 安全模型的详细描述

本节通过描述安全描述符及其包含的 ACL 内的各个权限如何影响各种操作提供了关于项目如何被保护的细节。

a) 安全描述符结构

在描述安全模型的细节之前，关于安全描述符的基本讨论是有帮助的。安全描述符包含与可保护对象相关联的安全信息。安全描述符由 SECURITY_DESCRIPTOR(安全描述符)结构及其相关联的安全信息构成。安全描述符可包括下列安全信息：

1. 对象的所有者及主要群组的 SID。
2. 指定对特定用户或用户组允许或拒绝的访问权限的 DACL。
3. 指定为对象生成监察记录的访问尝试的类型的 SACL。
4. 限定安全描述符或其个别成员含义的一组控制位。

较佳地，应用程序不能直接操纵安全描述符的内容。存在用于设置和检索对象的安全描述符中的安全信息的函数。另外，存在用于为新对象创建及初始化安全描述符的函数。

自由选择的访问控制列表 (DACL) 标识了被允许或拒绝访问可保护对象的受托者。当进程企图访问可保护对象时，系统核查该对象的 DACL 中的 ACE，来确定是否对其授予访问权限。如果该对象没有 DACL，则系统可对每个人授予完全访问权限。如果该对象的 DACL 没有 ACE，则系统拒绝访问该对象的企图，因为该 DACL 不允许任何访问权限。系统依次核查 ACE，直到找到允许所请求的所有访问权限的一个或多个 ACE，或直到任一所请求的访问权限被拒绝。

系统访问控制列表 (SACL) 使管理员能将访问受保护对象的企图记入日志。每个 ACE 指定了一指定受托者的访问企图的类型，该访问企图使得系统在安全事件日志中生成一个记录。当访问企图失败、当它成功或两者，SACL 中的 ACE 可生成监察记录。当未经授权的用户企图获得对于对象的访问权限时，SACL 提出警报。

所有类型的 ACE 包含下面的访问控制信息：

1. 标识向其应用 ACE 的受托者的安全标识符 (SID)。
2. 指定由该 ACE 控制的访问权限的访问掩码。
3. 指示该 ACE 的类型的标志。
4. 确定子容器或对象能否从 ACL 所附的主要对象继承 ACE 的一组位标志。

下面的表格列出由所有可保护对象所支持的三种 ACE 类型。

类型	描述
访问拒绝 ACE	在 DACL 中使用，用以对受托者拒绝访问权限。
访问允许 ACE	在 DACL 中使用，用以向受托者允许访问权限。
系统监察 ACE	在 SACL 中使用，以当受托者企图行使指定的访问权限时，用于生成监察记录。

(1) 访问掩码格式

所有可保护对象可使用在图 26 中所示的访问掩码格式来安排其访问权限。在此格式中，较低的 16 位用于对象专用访问权限，接下来的 7 位用于应用于大多数类型对象的标准访问权限，最高的 4 位用于指定每一对象类型可映射到一组标准和对象专用权限的类属访问权。ACCESS_SYSTEM_SECURITY (访问系统安全) 位对应于访问该对象的 SACL 的权限。

(2) 类属访问权限

类属权限在掩码内的 4 个最高位中指定。每一类可护对象将这些位映射到其一组标准和对象专用访问权限。例如，一文件对象可将 GENERIC_READ 位映射到 READ_CONTROL 和 SYNCHRONIZE 标准访问权限，及映射到 FILE_READ_DATA、FILE_READ_EA 和 FILE_READ_ATTRIBUTES 对象专用访问权限。其它类型的对象将 GENERIC_READ 位映射到适用于该类对象的访问权限组。

当打开对象的句柄时，类属访问权限可用于指定所需的访问类型。这通常比指定所有对应的标准和专用权限要简单。下面的表格示出了为类属访问权限定义的常量。

常量	类属含义
GENERIC_ALL	读、写及执行访问
GENERIC_EXECUTE	执行访问
GENERIC_READ	读访问
GENERIC_WRITE	写访问

(3) 标准访问权限

每一类可护对象有一组对应于对该类对象专用的操作的访问权限。除了这些对象专用访问权限之外，还有一组对应于对大多数类型的可保护对象公用的操作的标准访问权限。下面的表格示出了为标准访问权限定义的常量。

常量	含义
DELETE	删除对象的权利限。
READ_CONTROL	读对象安全描述符中的信息的权限，不包括 SACL 中的信息。
SYNCHRONIZE	使用该对象来同步的权限。这使线程能等待直到该对象处于被通知的状态。某些对象类型不支持此访问权限。
WRITE_DAC	修改对象的安全描述符中的 DACL 的权限。
WRITE_OWNER	改变对象的安全描述符中的所有者的权限。

b) 项目专用权限

在图 26 的访问掩码结构中，项目专用权限被放置在对象专用权限段（较低的 16 位）中。由于在本实施例中，存储平台展现了两组 API 来管理安全性—Win32 和存储平台 API，因此必须考虑文件系统对象专用权限以激发对存储平台对象专用权限的设计。

(1) 文件和目录对象专用权限

考虑下表：

目录	目录描述	文件	文件描述	值
FILE_LIST_DIRECTORY	列出目录的内容的权限	FILE_READ_DATA	读对应的文件数据的权限	0x0001
FILE_ADD_FILE	在目录中创建文件的权限	FILE_WRITE_DATA	将数据写入文件的权限	0x0002

FILE_ADD_SUBDIRECTORY	创建子目录的权限	FILE_APPEND_DATA	向文件追加数据的权限	0x0004
FILE_READ_EA	读扩展文件属性的权限	FILE_READ_EA	读扩展文件属性的权限	0x0008
FILE_WRITE_EA	写扩展文件属性的权限	FILE_WRITE_EA	写扩展文件属性的权限	0x0010
FILE_TRAVERSE	遍历目录的权限	FILE_EXECUTE	对于本机代码文件, 执行该文件的权限	0x0020
FILE_DELETE_CHILD	删除目录及其包含的所有文件的权限	无	无	0x0040
FILE_READ_ATTRIBUTES	读目录属性的权限	FILE_READ_ATTRIBUTES	读文件属性的权限	0x0080
FILE_WRITE_ATTRIBUTES	写目录属性的权限	FILE_WRITE_ATTRIBUTES	写文件属性的权限	0x0100

参考上表, 注意文件系统在文件和目录之间做出了基本的差别, 这就是文件和目录权限为何能在相同的位上重叠的原因。文件系统定义了非常粒度化的权限, 从而允许应用程序控制在这些对象上的行为。例如, 它们允许应用程序在与文件相关的属性 (FILE_READ/WRITE_ATTRIBUTES)、扩展属性和数据流之间进行区分。

本发明的存储平台的安全模型的目标是简化权限分配模型, 因此例如在数据存储项目 (联系人、电子邮件等) 上操作的应用程序一般无需在属性、扩展属性和数据流之间区分。然而, 对于文件和文件夹, 粒度 Win32 权限被保存, 且定义了通过存储平台的访问语义, 使得可提供与 Win32 应用程序的兼容性。这一映射对以下指定的每一项目权限讨论。

以下项目权限是用其相关联的可允许操作来指定的。也提供了这些项目权限的每一个背后的等效的 Win32 权限。

(2) WinFSItemRead

该权限允许对项目的所有元素的读访问，包括通过嵌入关系链接到项目的项目。它也允许枚举通过持有关系链接到该项目的项目（也称为目录清单）。这包括通过引用关系链接的项目的名称。这一权限映射到：

文件：

(FILE_READ_DATA|SYNCHRONIZE)

文件夹：

(FILE_LIST_DIRECTORY|SYNCHRONIZE)

语义是安全应用程序可设置 WinFSItemReadData (WinFS 项目读数据)，并指定权限掩码为以上指定的文件权限的组合。

(3) WinFSItemReadAttributes

该权限允许对项目的基本属性的读访问，这非常象基本文件属性和数据流之间的文件系统区分。较佳地，这些基本属性是驻留在从中导出所有项目的基本项目中的那些属性。该权限映射到：

文件：

(FILE_READ_ATTRIBUTES)

文件夹：

(FILE_READ_ATTRIBUTES)

(4) WinFSItemWriteAttributes

该权限允许对项目的基本属性的写访问，这非常象基本文件属性和数据流之间的文件系统区分。较佳地，这些基本属性驻留在从中导出所有项目的基本项目中的那些属性。该权限映射到：

文件：

(FILE_WRITE_ATTRIBUTES)

文件夹：

(FILE_WRITE_ATTRIBUTES)

(5) WinFSItemWrite

该权限允许对项目的所有元素进行写的能力，包括通过嵌入关系链接的项目。该权限也允许添加或删除到其它项目的嵌入关系的能力。该权限映射到：

文件：

(FILE_WRITE_DATA)

文件夹:

(FILE_ADD_FILE)

在存储平台数据存储中，在项目和文件夹之间没有区别，因为项目也可以具有到数据存储中的其它项目的持有关系。因此，如果具有 FILE_ADD_SUBDIRECTORY (或 FILE_APPEND_DATA) 权限，则可使项目为到其它项目的关系的源。

(6) WinFSItemAddLink

该权限允许添加到存储中的项目的持有关系的能力。应当注意，由于用于多个持有关系的安全模型改变项目上的安全性，且如果是来自分层结构中的较高点，则改变可绕过 WRITE_DAC，因此在目的地项目上要求 WRITE_DAC 以能够创建到其的关系。该权限映射到：

文件:

(FILE_APPEND_DATA)

文件夹:

(FILE_ADD_SUBSIRECTORY)

(7) WinFSItemDeleteLink

该权限即使在未向主体授予删除项目的权限时也允许删除到该项目的持有关系的能力。这与文件系统模型相一致，且有助于清除。该权限映射到：

文件:

(FILE_DELETE_CHILD) — 注意，文件系统没有等效于该权限的文件，但是有了具有到其它项目的持有关系的概念，并因此也可对非文件夹实现该权限。

文件夹:

(FILE_DELETE_CHILD)

(8) 删除项目的权限

如果到项目的最后一个持有关系消失，则该项目被删除。没有对删除项目的显式表示法。存在删除到项目的所有持有关系的清除操作，但是它是较高级工具且不是系统原语。

如果满足以下两个条件之一，则使用路径指定的任何项目可被取消链接：(1)

沿该路径的父项目向该项目授予写权限，或者（2）该项目本身上的标准权限授权 DELETE。当移除最后一个关系时，该项目从系统中消失。如果项目本身上的标准权限授权 DELETE，则使用 ItemID 指定的任何项目可被取消链接。

（9） 复制项目的权限

如果主体被授权项目上的 WinFSItemRead 和目的地文件夹上的 WinFSItemWrite，则项目可从源复制到目的地文件夹。

（10） 移动项目的权限

在文件系统中移动文件仅要求源文件上的 DELETE 权限和目的地目录上的 FILE_ADD_FILE，因为它保存了目的地上的 ACL。然而，可在 MoveFileEx 调用（MOVEFILE_COPY_ALLOWED）中指定让应用程序指定在跨卷移动的情况下可容忍 CopyFile 语义的标志。对于在移动之后对安全描述符发生了什么有四个可能的选择：

1. 与文件一起携带整个 ACL—默认的卷内移动语义。
2. 与文件一起携带整个 ACL 并将 ACL 标记为受保护的。
3. 仅携带显式 ACE 跨越，并在目的地上重新继承。
4. 不携带任何东西，并在目的地上重新继承—默认的卷间移动语义—与复制文件一样。

在本安全模型中，如果应用程序指定了 MOVEFILE_COPY_ALLOWED 标志，则对卷内和卷间两种情况都执行第四个选项。如果该标志未指定，则执行第二个选项，除非目的地也在同一安全区域中（即，同一继承语义）。存储平台级移动也实现第四个选项，且要求源上的 READ_DATA，与复制所要求的一样。

（11） 查看项目上的安全策略的权限

如果项目向主体授予标准权限 READ_CONTROL，则可查看项目的安全性。

（12） 改变项目上的安全策略的权限

如果项目向主体授予标准权限 WRITE_DAC，则可改变项目的安全性。然而，由于数据存储提供隐式继承，因此这具有可以如何在分层结构上改变安全性的含意。规则是如果分层结构的根授权 WRITE_DAC，则在整个分层结构上改变安全策略，而不论分层结构（或 DAG）中的特定项目是否不向主体授权 WRITE_DAC。

(13) 没有直接等效物的权限

在本实施例中, FILE_EXECUTE (用于目录的 FILE_TRAVERSE) 在存储平台中没有直接等效物。该模型为 Win32 兼容性保存这些权限, 但是没有基于这些权限为项目做出任何访问决策。如同 FILE_READ/WRITE_EA 一样, 由于数据存储项目没有扩展属性的表示法, 因此未提供该位的语义。然而, 为 Win32 兼容性保留该位。

3. 实现

定义同样受保护的区域的所有项目在安全表中具有与其相关联的条目。安全表定义如下:

项目身份	项目有序路径	显式项目 ACL	路径 ACL	区域 ACL
------	--------	----------	--------	--------

项目身份条目是同样受保护的安全区域的根的项目身份。项目有序路径是与同样受保护的安全区域的根相关联的有序路径 (ordpath)。显式项目 ACL 条目是为同样受保护的安全区域的根定义的显式 ACL。在某些情况下, 这可以为空, 例如, 当由于项目具有属于不同区域的多个父项目而定义新安全区域时。路径 ACL 是由项目继承的 ACL 集, 区域 ACL 条目是为与项目相关联的同样受保护的安全区域定义的 ACL 集。

对给定存储中的任何项目的有效安全性的计算充分利用了该表。为确定与项目相关联的安全策略, 获取与项目相关联的安全区域, 并检索与该区域相关联的 ACL。

当通过直接添加显式 ACL 或通过间接添加导致形成新安全区域的持有关系来改变与项目相关联的安全策略时, 使该安全表保持最新, 以确保上述用于确定项目的有效安全性的算法是有效的。

对存储的各种改变以及维护安全表的随附算法如下:

a) 在容器中创建新项目

当在容器中新创建项目时, 它继承与该容器相关联的所有 ACL。由于新创建的项目只有一个父项目, 因此它与其父项目属于同一区域。由此, 无需在安全表中创建新条目。

b) 向项目添加显式 ACL

当 ACL 被添加到项目时，它为包含分层结构中属于与给定项目本身相同的安全区域的所有其子孙定义新安全区域。对于包含分层结构中属于其它安全区域但是为给定项目的子孙的所有项目，安全区域保持不变，但是与该区域相关联的有效 ACL 被改变，以反映新 ACL 的添加。

对该新安全区域的引入可触发对具有与跨越旧安全区域和新定义的安全区域的祖先的多个持有关系的所有项目进一步定义区域。对于所有这样的项目，需要定义新的安全区域，并重复该过程。

图 27(a)、(b)和(c)描绘了通过引入新的显式 ACL 从现有安全区域中划分出来的新的同样受保护的安全区域。这是由标记为 2 的节点指示的。然而，对该新区域的引入导致创建另外的区域 3，因为项目具有多个持有关系。

以下对安全表的更新序列反映了对同样受保护的安全区域的分解。

c) 向项目添加持有关系

当向项目添加持有关系时，它引发三种可能性之一。如果持有关系的目标，即考虑中的项目是安全区域的根，则与该区域相关联的有效 ACL 被改变，且无需对该安全表的进一步修改。如果新持有关系的源的安全区域与项目的现有父项目的安全区域相同，则无需任何改变。然而，如果项目现在具有属于不同安全区域的父项目，则形成新的安全区域，以给定项目作为该安全区域的根。这一改变通过修改与该项目相关联的安全区域被传播到包含分层结构中的所有项目。属于与考虑中的项目相同的安全区域的所有项目及其在包含分层结构中的子孙都需要改变。一旦做出了改变，需要检查具有多个持有关系的所有项目以确定是否需要进一步的改变。如果这些项目中的任一个具有不同安全区域的父项目，则需要进一步的改变。

d) 从项目删除持有关系

当从项目删除持有关系时，如果满足某些条件，可能用其父区域来折叠安全区域。更精确地，这可以在以下条件下完成：（1）如果持有关系的移除导致有一个父项目的项目且没有为该项目指定显式 ACL；（2）如果持有关系的移除导致其父项目都在同一安全区域内的项目，且没有为该项目定义显式 ACL。在这些情况下，这些安全区域可被标记为与父项目相同。这一标记需要应用于其安全区域对应于所折叠的区域的所有项目。

e) 从项目中删除显式 ACL

当从项目中删除显式 ACL 时，可能用其父项目的安全区域来折叠根为该项目的安全区域。更精确地，如果显式 ACL 的移除导致其在包含分层结构中的父项目属于同一安全区域的项目，则可实现这一过程。在这些情况下，可将安全区域标记为与父项目相同，且改变被应用于其安全区域对应于所折叠的区域的所有项目。

f) 修改与项目相关联的 ACL

在这一情形中，不需要对安全表进行任何新的添加。与该区域相关联的有效 ACL 被更新，且将该新的 ACL 改变传播到被它所影响的安全区域。

F. 通知和改变跟踪

按本发明的另一方面，存储平台提供允许应用程序跟踪数据改变的通知能力。此特征主要供保持易失状态或执行数据改变事件上的业务逻辑的应用程序使用。应用程序注册在项目、项目扩展及项目关系上的通知。在提交了数据改变之后通知被异步地传送。应用程序可按项目、扩展和关系类型以及操作类型来过滤通知。

按一个实施例，存储平台 API 322 为通知提供两类接口。第一，应用程序注册由对项目、项目扩展和项目关系的改变触发的简单数据改变事件。第二，应用程序创建“监视程序”对象来监视项目、项目扩展和项目之间关系的组。在系统失败或系统离线超过延长的时间段之后，可保存和重新创建监视程序对象的状态。单个通知可反映多个更新。

1. 存储改变事件

本节提供了如何使用由存储平台 API 322 提供的通知接口的若干示例。

a) 事件

项目、项目扩展和项目关系展现了由应用程序用于注册数据改变通知的数据改变事件。以下代码示例示出了 Item 基类上 ItemModified（项目修改）和 ItemRemoved（项目移除）事件处理程序的定义。

```
//事件  
public event ItemModifiedEventHandler Item_ItemModified;  
public event ItemRemovedEventHandler Item_ItemRemoved;
```

所有通知都携带足够的数据以从数据存储中检索改变的项目。以下代码示例

示出了如何注册项目、项目扩展或项目关系上的事件：

```
myItem.ItemModified += new ItemModifiedEventHandler(this.onItemUpdate);
myItem.ItemRemoved += new ItemRemovedEventHandler(this.onItemDelete);
```

在本实施例中，存储平台确保如果自从最后一次传送通知以来相应的项目被修改或删除，或在自从最后一次从数据存储中取出以来的新注册的情况下将通知应用程序。

b) 监视程序

在本实施例中，存储平台定义了用于监视与（1）文件夹或文件夹分层结构，（2）项目上下文，或（3）特定项目相关联的对象的监视程序类。对于这三个类别中的每一个，存储平台提供了监视相关联的项目、项目扩展或项目关系的特定监视程序类，例如，存储平台提供相应的 FolderItemWatcher（文件夹项目监视程序）、FolderRelationshipWatcher（文件夹关系监视程序）以及 FolderExtensionWatcher（文件夹扩展监视程序）类。

当创建监视程序时，应用程序可对预先存在的项目，即项目、扩展或关系请求通知。该选项大多数用于维护专用项目高速缓存的应用程序。如果未请求，则应用程序接收对发生在创建了监视程序对象之后的所有更新的通知。

连同传送通知一起，存储平台提供“WatcherState（监视程序状态）”对象。WatcherState 可被串行化并保存在磁盘上。监视程序状态随后可用于在失败或当在离线之后重新连接时重新创建相应的监视程序。新的重新创建的监视程序将重新生成未确认的通知。应用程序通过在相应的监视程序状态上调用提供对通知的引用的“Exclude（排除）”方法来指示通知的送达。

存储平台向每一事件处理程序传送监视程序状态的单独副本。在同一事件处理程序的随后的调用上接收到的监视程序状态假定所有先前接收的通知的送达。

作为示例，以下代码示例示出了 FolderItemWatcher 的定义。

```
public class FolderItemWatcher : Watcher
{
    //构造函数
    public FolderItemWatcher_Constructor(Folder folder);
    public FolderItemWatcher_Constructor1(Folder folder, Type itemType);
    public FolderItemWatcher_Constructor2(HandlerContext context, ItemId folderId);
    public FolderItemWatcher_Constructor3(Folder folder, Type itemType,
        FolderItemWatcherOptions options);
    public FolderItemWatcher_Constructor4(HandlerContext context, ItemId folderId, Type itemType);
    public FolderItemWatcher_Constructor5(HandlerContext context, ItemId, folderId, Type itemType,
        FolderItemWatcherOptions options);
```

```

//属性
public ItemId FolderItemWatcher_FolderId{get;}
public Type FolderItemWatcher_ItemType{get;}
public FolderItemWatcherOptions FolderItemWatcher_Options{get; }

//事件
public event ItemChangedEventHandler FolderItemWatcher_ItemChanged;
}

```

以下代码示例示出了如何创建用于监视文件夹的内容的文件夹监视程序对象。监视程序在添加新音乐项目或更新或删除现有音乐项目时生成通知，即事件。文件夹监视程序监视文件夹分层结构中的特定文件夹或所有文件夹。

```

myFolderItemWatcher = new FolderItemWatcher(myFolder, typeof(Music));
myFolderItemWatcher.ItemChanged += new ItemChangedEventHandler(this.onItemChanged);

```

2. 改变跟踪和通知生成机制

该存储平台提供了一种简单而有效的机制来跟踪数据改变和生成通知。客户机在用于检索数据的同一连接上检索通知。这极大程度上简化了安全检查、移除了可能网络配置上的延时和约束。通知是通过发出选择语句来检索的。为防止轮询，客户机可使用由数据库引擎 314 提供的“等待（waitfor）”特征。图 13 示出了基本存储平台通知概念。等待查询可同步地执行，在该情况下，调用线程被阻断，直到结果可用；或者可异步地执行，在该情况下，线程不被阻断，且当结果可用时在单独的线程上返回结果。

“等待”和“选择”的组合对于监视适合特定数据范围的数据改变是有吸引力的，因为改变可通过在相应的数据范围上设置通知锁来监视。这适用于许多常见的存储平台情形。对于个别项目的改变可以通过在相应的数据范围上设置通知锁来监视。对文件夹和文件夹树的改变可通过在路径范围上设置通知锁来监视。对类型及其子类型的改变可通过在类型范围上设置通知锁来监视。

一般而言，有三个与处理通知相关联的不同阶段：（1）数据改变或甚至检测，（2）订阅匹配，以及（3）通知送达。排除同步通知送达，即作为执行数据改变的事务的一部分的通知送达，存储平台可实现以下两种形式的通知送达：

- 1) **直接事件检测：**执行事件检测和订阅匹配作为更新事务的一部分。通知被插入到由订户监视的表中；以及
- 2) **推迟的事件检测：**在提交了更新事务之后执行事件检测和订阅匹配。随

后实际的订户或中介检测事件并生成通知。

直接事件检测要求执行额外的代码作为更新操作的一部分。这允许捕捉所有感兴趣的事件，包括指示相对状态改变的事件。

推迟的事件检测移除向更新操作添加额外代码的要求。事件检测是由最终订户来完成的。推迟的事件检测通常对事件检测和事件送达进行批处理，且很好地适用于数据库引擎 314（例如，SQL Server）的查询执行基础结构。

推迟的事件检测依赖于更新操作留下的日志或跟踪。存储平台为删除的数据项目维护一组逻辑时间标记以及墓碑。当扫描数据存储中的改变时，客户机提供定义用于检测改变的低水印的时间标记以及防止重复通知的一组时间标记。应用程序可能接收到对在由低水印指示的时间之后发生的所有改变的通知。

能够访问核心视图的复杂应用程序还可通过创建专用参数和重复过滤器表，来优化和减少监视一组可能很大的项目所需的 SQL 语句的数量。具有特殊需求的应用程序，诸如必须支持丰富视图的应用程序可使用可用的改变跟踪框架来监视数据改变并刷新其专用快照。

因此，较佳地，在一个实施例中，存储平台实现推迟的时间检测方法，如下文更完整地描述的。

a) 改变跟踪

所有项目、扩展和项目关系定义携带唯一的标识符。改变跟踪为所有数据项目维护一组逻辑时间标记来记录创建、更新和删除时间。墓碑条目用于表示删除的数据项目。

应用程序使用该信息来有效地监视自从应用程序最后一次访问数据存储以来是否新添加、更新或删除了特定的项目、项目扩展或项目关系。以下示例示出了该机制。

```
create table [item-extension-relationship-table-template] (
    identifier uniqueidentifier not null default newid()
    created bigint,           not null, --@@@当创建时存入
    updated bigint,           not null, --@@@当最后一次更新时存入
    .....
)
```

所有删除的项目、项目扩展和关系被记录在对应的墓碑表中。一个模板示出如下。

```
create table [item-extension-relationship-tombstone table-template] (
```

```

identifier uniqueidentifier not null,
deleted bigint,          not null, --@@当删除时存入,
created bigint,           not null, --@@当创建时存入,
updated bigint,           not null, --@@当最后一次更新时存入
.....
)

```

为效率原因，存储平台为项目、项目扩展、关系和路径名维护一组全局表。这些全局查找表可以由应用程序用于有效地监视数据范围和检索相关联的时间标记和类型信息。

b) 时间标记管理

逻辑时间标记对于数据库存储，即存储平台卷是“本地的”。时间标记是单调递增的 64 位值。保留单个时间标记通常足以检测在最后一次连接到存储平台卷之后是否发生了数据改变。然而，在最真实的情形中，需要保存稍多一些的时间标记来检查重复。原因解释如下。

关系型数据库表是构建在一组物理数据结构上的逻辑抽象，即 B 树、堆等。向新创建或更新的记录分配时间标记不是原子动作。将该记录插入到底层数据结构可能在不同的时刻发生，由此应用程序可能会看到记录是无序的。

图 14 示出了将新记录插入到同一 B 树中的两个事务。由于事务 T3 在调度事务 T2 的插入之前插入其记录，因此扫描 B 树的应用程序可能会看到由事务 T3 插入的记录在由 T2 插入的事务之前。由此，读者可能不正确地假定他看到了在时刻“10”之前创建的所有记录。为解决这一问题，数据库引擎 314 提供了一种返回低水印的功能，在该低水印之前，提交了所有的更新且所有更新都被插入到各自的底层数据结构中。在上述示例中，返回的低水印是“5”，假定读者在提交事务 T2 之前启动。由数据库引擎 314 提供的低水印允许应用程序在扫描数据库或数据范围内的数据改变时有效地确定要忽略哪些项目。一般而言，ACID 事务被假定为持续非常短的时间，由此，预期低水印非常接近于最近分发的时间标记。在存在长持续时间的事务的情况下，应用程序可能必须保存个别的时间标记来检测和丢弃重复。

c) 数据改变检测—事件检测

当查询数据存储时，应用程序获得低水印。随后，应用程序使用该水印来扫描数据存储中其创建、更新或删除时间标记大于返回的低水印的条目。图 15 示出了该过程。

为防止重复通知，应用程序记住大于返回的低水印的时间标记，并使用这些

时间标记来滤除重复。应用程序创建会话本地临时表，以有效低处理一大组重复时间标记。在发出选择语句之前，应用程序插入所有先前返回的重复时间标记，并删除比返回的最后一个低水印更老的时间标记，如下所示。

```
delete from $duplicates where ts < @oldLowWaterMark
insert into $duplicates(ts) values(...), ..., (...);

waitfor(select *, getLowWaterMark() as newLowWaterMark
       from [global!items]
       where updated >= @oldLowWaterMark
       and updated not in (select * from $duplicates))
```

G. 同步

依照本发明的另一方面，该存储平台提供同步服务 330，它 (I) 允许存储平台的多个实例（每个有自己的数据存储 302）按一组灵活的规则来同步它们的内容的各部分，以及 (ii) 为第三方提供基础结构以将本发明的存储平台的数据存储与实现专有协议的其它数据源同步。

存储平台到存储平台的同步在一组参与的复制品之间发生。例如，参考图 3，希望在多半是在不同的计算机系统上运行的存储平台的另一实例的控制下提供在存储平台 300 的数据存储 302 和另一远程数据存储 338 之间的同步。该组的总成员资格不必在任何给定时间被任何给定复制品知道。

不同的复制可以独立地（即并发地）作出改变。将同步过程定义成使每个复制品知道由其它复制品作出的改变。此同步能力本质上是多主的（multi-master）。

本发明的同步能力允许各复制品：

- 确定另一复制品知道什么改变；
- 请求关于此复制品不知道的改变的信息；
- 传达关于其它复制品不知道的改变的信息；
- 确定两个改变何时互相冲突；
- 本地应用改变；
- 传达冲突分解到其它复制品以确保收敛性；以及
- 基于对冲突分解指定的策略分解冲突。

1. 存储平台到存储平台的同步

本发明的存储平台的同步服务 300 的主要应用是同步存储平台（每个带有它

自己的数据存储)的多个实例。同步服务在存储平台模式级上操作(而不是在数据库引擎 314 的底层表中)。因此,例如“范围(Scope)”用于定义下面讨论的同步组。

同步服务按“纯改变(net change)”的原则操作。不是记录和发送各个操作(如事务复制那样),同步服务而是发送这些操作的最终结果,因此常将多个操作的结果合并成单个最终结果。

同步服务通常不考虑事务边界。换言之,若在单个事务中对存储平台数据存储作出两个改变,不保证这些改变原子地应用到所有其它复制品上—可以示出一个改变而不示出其它改变。此原则的例外是,若在同一事务中对同一项目作出两个改变,则这些改变保证被原子地发送和应用到其它复制品。因此,项目是同步服务的一致性单元。

a) 同步(Sync)控制应用程序

任一应用程序可连接到同步服务并启动同步操作。那样的应用程序提供执行同步(见下面同步概况)所需的所有参数。那样的应用程序在这里被称为同步控制应用程序(SCA)。

在同步两个存储平台实例时,在一侧由 SCA 启动同步。该 SCA 通知本地同步服务与远程伙伴同步。在另一侧,同步服务通过由来自发起机器的同步服务发出的消息唤醒。它基于在目标机器上存在的持久配置信息(见下文的映射)作出响应。同步服务能按时间表或响应于事件运行。在这些情况下,实现时间表的同步服务成为 SCA。

为启用同步,需要采取两个步骤。首先,模式设计者必须用合适的同步语义注释存储平台模式(如下文所述的指定改变单元)。其次,同步必须在具有参与同步的存储平台的实例的所有机器上正确地配置(如下所述)。

b) 模式注释

同步服务的基本概念是改变单元(Change Unit)的概念。改变单元是由存储平台个别地跟踪的最小的模式片段。对每个改变单元,同步服务能确定自从最后一次同步以来它是被改变还是未被改变。

指定模式中的改变单元达到若干目的。首先,它确定了在线的同步服务如何罗嗦。当在改变单元内作出改变时,整个改变单元被发送到其它复制品,因为同步

服务不知道改变单元的哪部分被改变。其次，它确定了冲突检测的粒度。当对同一改变单元作出两个并发的改变（这些术语在后继章节中详细定义），同步服务引起冲突；另一方面，若对不同改变单元作出并发改变，则无冲突发生，且改变被自动地合并。第三，它严重地影响了由系统保持的元数据的量。对每个改变单元保持许多同步服务元数据；因此，使改变单元更小会增加同步的额外开销。

定义改变单元需要找出正确的折衷。为此，同步服务允许模式设计者参与此过程。

在一个实施例中，同步服务不支持大于一个元素的改变单元。然而，它支持让模式设计者指定比一个元素更小的改变单元的能力—即，将一个元素的多个属性组合到单独的改变单元中。在该实施例中，这是使用下述句法实现的：

```
<Type Name="Appointment" MajorVersion="1" MinorVersion="0" ExtendsType="Base.Item"
      ExtendsVersion="1">

  <Field Name="MeetingStatus" Type="the storage platformTypes.uniqueidentifier Nullable=False"/>
  <Field Name="OrganizerName" Type="the storage platformTypes.nvarchar(512)" Nullable=False"/>
  <Field Name="OrganizerEmail" Type="the storage platformTypes.nvarchar(512)"
        TypeMajorVersion="1"           MultiValued=True"/>
  ...
  <ChangeUnit Name="CU_Status">
    <Field Name="MeetingStatus" />
  </ChangeUnit>

  <ChangeUnit Name="CU_Organizer" />
    <Field Name="OrganizerName" />
    <Field Name="OrganizerEmail" />
  </ChangeUnit>
  ...
</Type>
```

c) 同步配置

希望保持它们数据的某些部分同步的一组存储平台伙伴被称为同步共同体。虽然共同体的成员希望保持同步，它们不需要以完全相同的方式表示数据；换言之，同步伙伴可转换他们正在同步的数据。

在对等情况下，让对等方对所有它们的伙伴维持转换映射是不现实的。相反，同步服务采取定义“共同体文件夹”的方法。共同体文件夹是代表所有共同体成员正在与之同步的假设的“共享文件夹”的抽象。

此概念最好用一例子说明。若 Joe 希望保持他的若干计算机的 My Documents（我的文档）文件夹同步，Joe 定义一共同体文件夹，如称为 JoeDocuments。随后在每台计算机上，Joe 在假设的 JoeDocuments 文件夹和本地 My Documents 文件夹

之间配置一映射。从这点出发，当 Joe 的计算机彼此同步时，它们借助 JoeDocuments 中的文档而不是它们的本地项目来交谈。以此方法，所有 Joe 的计算机互相理解，而不必知道其它人是谁一共同体文件夹成为同步共同体的通用语。

配置同步服务包括三个步骤：（1）定义在本地文件夹和共同体文件夹之间的映射；（2）定义确定哪个得到同步的同步概况（如与谁同步，以及哪个子集应当被发送、哪个被接收）；以及（3）定义不同的同步概况应当运行的时间表，或手动运行它们。

（1） 共同体文件夹一映射

共同体文件夹映射作为 XML 配置文件被存储在个别机器上。每个映射具有以下模式：

/mappings/communityFolder

此元素命名映射的共同体文件夹。名字遵循文件夹的句法规则。

/mappings/localFolder

此元素命名映射所转换到的本地文件夹。此名字遵循文件夹的句法规则。为了使映射有效，文件夹必须已存在。此文件夹中的项目被看作对每一此映射的同步。

/mappings/transformations

此元素定义如何将项目从共同体文件夹转换到本地文件夹以及如何反向转换。若缺少或为空，不执行转换。具体说来，这意味着无 ID 被映射。此配置主要用于创建文件夹的高速缓存。

/mappings/transformations/mapIDs

此元素请求新生成的本地 ID 被赋予所有从共同体文件夹映射的项目，而不是重新使用共同体 ID。同步运行库维护 ID 映射，以来回转换项目。

/mappings/transformations/localRoot

此元素请求共同体文件夹中的所有根项目作为指定根的子项目。

/mappings/runAs

此元素控制在谁的授权下处理针对此映射的请求。若不存在，则假设发送者。

/mappings/runAs/sender

存在此元素表明对此映射的消息发送者必须是人格化的，且在他的凭证下处理请求。

(2) 概况

同步概况是分离同步所需的总的参数集。由 SCA 将其提供给同步运行库以启动同步。存储平台到存储平台的同步的同步概况包含以下信息：

- 本地文件夹，用作改变的源和目标；
- 与之同步的远程文件夹名—此文件夹必须通过如上定义的映射从远程伙伴发布；
- 方向—同步服务支持只发送、只接收以及发送—接收同步；
- 本地过滤器—选择发送什么本地信息到远程伙伴。表示成本地文件夹上的存储平台查询；
- 远程过滤器—选择从远程伙伴检索什么远程信息—表示成共同体文件夹上的存储平台查询；
- 转换—定义如何在项目和本地格式间转换；
- 本地安全性—指定是在远程端点（人格化）的许可下应用从远程端点检索的改变，还是用户在本地启动同步；以及
- 冲突分解策略—指定冲突是应被拒绝、记入日志还是自动分解—在后一种情况下，指定使用哪个冲突分解器以及它的配置参数。

同步服务提供允许简单构建同步概况的运行库 CLR 类。概况可被串行化成 XML 文件或从 XML 文件串行化，以便容易存储（常与时间表一起）。然而，在存储平台中没有存储所有概况的标准地方；欢迎 SCA 在不必永久保持的点上构建概况。注意，不需要具有本地映射来启动同步。能在概况中指定所有同步信息。然而为响应于由远程方启动的同步请求，需要映射。

(3) 时间表

在一个实施例中，同步服务不提供它自己的调度基础结构。相反，它依赖于另一组件来完成此任务—在 Microsoft Windows 操作系统中可得到的 Windows Scheduler。同步服务包括命令行实用程序，它担当 SCA 并基于保存在 XML 文件中的同步概况触发同步。该实用程序使得按时间表或者响应于如用户登录或登出等事件来配置 Windows Scheduler 运行同步变得非常容易。

d) 冲突处理

同步服务中的冲突处理被划分成三个阶段：(1)发生在改变应用时的冲突检

测一此步骤判断是否可安全地应用改变；（2）自动冲突分解并记入日志—在此步骤（发生在紧接着冲突检测之后）资咨询自动冲突分解器以查看冲突是否能被分解—若不能，可选地将冲突记入日志；以及（3）冲突检查与分解—若某些冲突已被记入日志，且发生在同步会话的环境之外，则采取此步骤—此时，被记入日志的冲突能被分解并从日志中移除。

（1）冲突检测

在本实施例中，同步服务检测两种类型的冲突：基于知识的冲突和基于约束的冲突。

（a）基于知识的冲突

当两个复制品对同一改变单元进行独立的改变时，会造成基于知识的冲突。如果两个改变是在彼此没有知识的情况下进行的，则这两个改变被称作为是独立的—换言之，第一个的版本没有被第二个的知识覆盖，反之亦然。基于如以上所述的复制品的知识，所述同步服务自动地检测所有的这种冲突，并且处理这些冲突，在此如下面所述。

有时，将冲突认为是改变单元的版本历史中的分叉是有帮助的。如果在改变单元的生命中没有冲突发生，则其版本历史就是一个简单的链—每个改变都在之前的一个后面发生。在基于知识的冲突的情况下，两个改变并行发生，使得所述链分裂并且变成版本树。

（b）基于约束的冲突

存在一些情况，在这些情况下，当一起被应用时，独立的改变违反了完整性约束。例如，在相同目录中创建具有相同名称的文件的两个复制品会使得这样的冲突发生。

基于约束的冲突涉及两个独立的改变（正如用于基于知识的冲突一样）；然而，它们不影响相同改变单元。相反，它们影响不同改变单元，但在它们之间存在约束。

同步服务在改变应用时检测约束违反，并自动引发基于约束的冲突。分解基于约束的冲突通常要求以不违反约束的方式来修改改变的自定义代码；同步服务不提供用于完成此过程的通用机制。

(2) 冲突处理

当检测到冲突时，同步服务可以采取 3 个动作中的一个（由同步概况中的同步启动器选择）：（1）拒绝改变，将其返回给发送者；（2）将冲突记录到冲突日志中；或者（3）自动分解冲突。

如果拒绝改变，那么如果改变没有到达所述复制品，则同步服务起作用。否定确认被发送回启动器。这种分解策略主要在无头复制品（例如文件服务器）上有用，其中冲突日志记录是不可行的。相反，这种复制品强迫其它复制品通过拒绝改变来处理冲突。

同步启动器配置其同步概况中的冲突分解。同步服务支持在单个概况中通过以下方法来组合多个冲突分解器—首先，指定冲突处理器列表一个接一个地进行尝试，直到其中一个成功为止；其次，将冲突处理器与冲突类型相关联，例如，将更新一更新基于知识的冲突指向一个冲突处理器，而将所有其它冲突指向日志。

(a) 自动冲突分解

同步服务提供多种默认的冲突分解器。该列表包括：

- 本地优胜：如果与本地储存的数据冲突，则丢弃传入改变；
- 远程优胜：如果与传入改变冲突，则丢弃本地数据；
- 最后写入者优胜：基于改变的时间标记挑选本地优胜者或者远程优胜者的任一个（注意，通常同步服务不依赖于时钟值；这种冲突分解器对于那个规则是惟一例外）；
- 确定性的：以保证在所有复制品上相同的方式挑选优胜者，但不是另外有意义的一同步服务的一个实施例可能使用伙伴 ID 的字典式比较来实现这个特征。

另外，ISV 可以实现并且安装其自己的冲突处理器。自定义冲突处理器可以接受配置参数；这种参数必须由同步概况的冲突分解部分中的 SCA 来指定。

当冲突分解器处理冲突时，它将需要执行的操作的列表（代替冲突改变）返回给运行时间。然后，同步服务应用这些操作，适当地调整远程知识，以便包括冲突处理器已经考虑过的信息。

可能在应用分解的同时检测到了另一个冲突。在这种情况下，在重新进行原始处理之前，新的冲突必须被分解。

当将冲突看作项目的版本历史中的分支时，冲突分解可以被看作接点一组合两个分支以便形成单独的点。因此，冲突分解将版本历史转为有向非循环图（DAG）。

(b) 冲突日志记录

一种非常特定类型的冲突处理器是冲突日志记录器。同步服务将冲突记录在日志中作为类型 ConflictRecord 的项目。这些记录反过来与冲突的项目有关（除非项目本身已经被删除）。每个冲突记录包含：引发冲突的传入改变；冲突的类型：更新一更新、更新一删除、删除一更新、插入一插入，或者约束；以及传入改变的版本和发送它的复制品的知识。记入日志的冲突可用于如下所述的检查和分解。

(c) 冲突检查和分解

同步服务提供用于应用程序检查冲突日志并且建议对其中的冲突的分解方法的 API。API 允许应用程序枚举所有冲突或者与给定的项目有关的冲突。它还允许这些应用程序以 3 种方式中的一种来分解被记入日志的冲突：（1）远程优胜一接受被记入日志的改变并且覆盖相冲突的本地改变；（2）本地优胜一忽略被记入日志的改变的冲突部分；以及（3）建议新的改变一其中应用程序建议一种合并，该合并在其看法中分解所述冲突。一旦由应用程序分解了冲突，同步服务就将它们从日志中删除。

(d) 复制品的收敛和冲突分解的传播

在复杂的同步情形中，在多个复制品上可以检测到相同的冲突。如果发生了这种情形，则许多事情都可能发生：（1）在一个复制品上分解冲突，而分解被发送给另一个；（2）在两个复制品上自动分解冲突；或者（3）在两个复制品上手动分解冲突（通过冲突检查 API）。

为了确保收敛，同步服务将冲突分解传递给其它复制品。当分解冲突的改变到达复制品时，同步服务自动地找到日志中通过该更新分解的任意冲突记录，并且删除它们。在这种情况下，一个复制品上的冲突分解将绑定在所有其它复制品上。

如果对于相同的冲突，由不同的复制品选择不同的优胜者，则同步服务应用绑定冲突分解的原理，并且自动地挑选两种分解中优于另一种的一种分解。以确定的方式挑选出优胜者，该确定方式保证在所有的時候都产生相同的结果（一个实施例使用复制品 ID 字典式比较）。

如果对相同的冲突，不同的复制品建议不同的“新改变”，则同步服务将这种新的冲突看作特殊的冲突，并且使用冲突日志记录器来防止它传播到其它复制品。这种情况通常会引起手动冲突分解。

2. 对非存储平台数据存储的同步

按本发明的存储平台的另一方面，存储平台提供 ISV 用于实现同步适配器的体系结构，同步适配器使存储平台能与如 Microsoft Exchange、AD、Hotmail 等传统系统同步。同步适配器得益于由下述同步服务提供的许多同步服务。

不管其名称如何，同步适配器不需要作为某个存储平台体系结构的插件来实现。在需要时，“同步适配器”能简单地是利用同步服务运行库接口来获得如改变枚举和应用等服务的任何应用程序。

为了使其他人能更容易地配置和运行到给定后端的同步，鼓励同步适配器的编写者展现标准同步适配器接口，它在给定上述的同步概况时运行同步。概况提供配置信息给适配器，适配器将某些信息传送到同步运行库以控制运行库服务（如，要同步的文件夹）。

a) 同步服务

同步服务向适配器编写者提供若干同步服务。在本节余下部分，方便地将存储平台在其上完成同步的机器称为“客户机”，而适配器正与其对话的非存储平台后端称为“服务器”。

(1) 改变枚举

基于由同步服务维持的改变跟踪数据，改变枚举允许同步适配器容易地枚举自从最后一次与该伙伴试图作出同步以来对数据存储文件夹发生的改变。

基于“定位点”的概念来枚举改变—这是表示有关最后一次同步的信息的不透明的结构。如以前章节所述，定位点采取存储平台知识的形式。利用改变枚举服务的同步适配器落入两大类别：使用“存储的定位点”的适配器和使用“提供的定位点”的适配器。

区别基于关于最后一次同步的信息存储在哪里—在客户机上还是在服务器上。适配器常常容易地存储此信息在客户机上—后端往往不能容易地存储此信息。另一方面，若多个客户机与同一后端同步，则将此信息存储在客户机上是低效且在

某些情况下是不正确的一这使一个客户机不知道其它客户机已推到服务器的改变。若适配器希望使用服务器存储的定位点，则适配器需要在改变枚举时将其送回到存储平台。

为了让存储平台维护定位点（用于本地或远程存储），存储平台需要知道成功地应用在服务器上的改变。这些且只有这些改变能包括在定位点中。在改变枚举期间，同步适配器使用确认（Acknowledgement）接口，以报告哪个改变已被成功地应用。在同步结束时，使用提供的定位点的适配器必须读出新定位点（它结合了所有成功应用的改变）并将其发送到它们的后端。

各适配器常常需要存储适配器专用数据以及插入到存储平台数据存储中的各项目。该数据存储的常见例子是远程 ID 和远程版本（时间标记）。同步服务提供用于存储此数据的机制，而改变枚举提供接收此额外数据以及要返回的改变的机制。在大多数情况下，这消除了适配器重新查询数据库的需求。

(2) 改变应用

改变应用允许同步适配器将从它们的后端接收的改变应用到本地存储平台。期望适配器将改变转换到存储平台模式。

改变应用的主要功能是自动检测冲突。如在存储平台到存储平台同步的情况下，冲突被定义成在互相不知道时作出的两个重叠的改变。当适配器使用改变应用时，它们必须指定对其执行冲突检测的定位点。若检测到未被适配器的知识覆盖的重叠的本地改变，则改变应用引起冲突。类似于改变枚举，适配器可使用存储的或提供的定位点。改变应用程序支持适配器专用元数据的有效存储。那样的数据可由适配器将其附加到要应用的改变上，且可被同步服务存储。数据可在下次改变枚举时返回。

(3) 冲突分解

上述冲突分解机制（包括日志记录和自动分解选项）也对同步适配器可用。在应用改变时，同步适配器能指定用于冲突分解的策略。若指定，则冲突可被传递到指定的冲突处理程序并予以分解（若可能）。冲突也能被记入日志。当试图将本地改变应用到后端时，适配器可检测冲突。在那样情况下，适配器仍可以将冲突传递到同步运行库，以按策略分解。此外，同步适配器可请求任何由同步服务检测的冲突发回给它们以便处理。在后端能存储或分解冲突的情况下这特别方便。

b) 适配器实现

虽然某些“适配器”简单地是利用运行库接口的应用程序，然而鼓励适配器实现标准的适配器接口。这些接口允许同步控制应用程序：请求适配器按给定的同步概况执行同步；取消正进行的同步；以及接收关于正进行同步的进展报告（完成百分比）。

3. 安全性

同步服务努力将尽可能少的同步引入到由存储平台实现的安全模式中。不是对同步定义新的权限，而是使用现有的权限。具体地，

- 能读数据存储项目的任何人可枚举对该项目的改变；
- 能写到数据存储项目的任何人可向该项目应用改变；以及
- 能扩展数据存储项目的任何人可将同步元数据与该项目关联。

同步服务不维护安全授权信息。当在复制品 A 由用户 U 作出改变，且将其转发到复制品 B 时，该改变最初在 A 处（由 U）作出的事实丢失了。若 B 将此改变转发到复制品 C，则这是在 B 的授权而不是在 A 的授权下完成的。这就导致下述限制：若不信任一个复制品对一个项目作出它自己的改变，它不能转发由其它复制品作出的改变。

在启动同步服务时，由同步控制应用程序完成。同步服务人格化 SCA 的身份，并在该身份下完成所有操作（本地的和远程的）。作为说明，观察到用户 U 不能使本地同步服务从远程存储平台检索对用户 U 不具有读访问的项目的改变。

4. 可管理性

监视复制品的分布式共同体是复杂的问题。同步服务可使用“扫描”算法来收集和分发关于该复制品的状态的信息。扫描算法的属性确保关于所有所配置的复制品的信息最终被收集，且检测到该失败（无响应）的复制品。

在每个复制品上可得到共同体范围的监视信息。可在任意选取的复制品上运行监视工具，以检查此监视信息并作出管理决策。在受影响的复制品上必须直接作出配置改变。

G. 传统文件互操作性

如上提到，至少在某些实施例中，本发明的存储平台旨在被实施为计算机系统的硬件/软件接口系统的整体部分。例如，本发明的存储平台可被实施为如 Microsoft Windows 家族操作系统的操作系统整体部分。在这方面，存储平台 API 成为操作系统 API 的一部分，应用程序通过它与操作系统交互。因此，存储平台成为装置，应用程序通过它将信息存到操作系统上，且从而基于项目的存储平台的数据模型替代了这一操作系统的传统文件系统。例如，当在 Microsoft Windows 家族操作系统中实施时，存储平台可替代在该操作系统中实现的 NTFS 文件系统。当前，应用程序通过由 Windows 家族操作系统展现的 Win32 API 来访问 NTFS 文件系统的服务。

然而，应认识到，完全用本发明的存储平台替代 NTFS 文件系统需要重新编码现有的基于 Win32 的应用程序，且那样的重新编码可能是不合需要的，因此本发明的存储平台提供与如 NTFS 等现有文件系统的某种互操作性是有益的。从而，在本发明的一个实施例中，存储平台使依赖于 Win32 编程模型的应用程序能同时访问存储平台的数据存储以及传统的 NTFS 文件系统的内容。为此，存储平台使用作为 Win32 命名约定的超集的命名约定以便于容易的互操作性。此外，存储平台支持通过 Win32 API 访问存储在存储平台卷中的文件和目录。

1. 互操作性模型

依照本发明的这一方面，且依照上述示例性实施例，存储平台实现了一种名字空间，其中可组织非文件和文件项目。采用该模型，可获得以下优点：

1. 数据存储中的文件夹可包含文件和非文件项目，由此为文件和模式化的数据呈现了单个名字空间。此外，它也为所有用户数据提供了同一的安全、共享和管理模型。
2. 由于文件和非文件项目都可使用存储平台 API 来访问，且该方法中没有为文件施加特殊规则，因此它为应用程序开发者提出了一种更清楚的编程模型来工作。
3. 所有名字空间操作都通过存储平台，且因此被同步地处理。重要的是注意，深层的属性升级（文件内容的驱散）仍异步地发生，但是同步操作为用户和应用程序提供了更可预测的环境。

作为该模型的结果，在本实施例中，可能不在被合并到存储平台数据存储的

数据源上提供搜索能力。这包括可移动媒体、远程服务器和本地磁盘上的文件。提供了为驻留在外部文件系统中的项目显示存储平台中的代理服务器项目(快捷方式+升级的元数据)的同步适配器。代理服务器项目不试图按照数据源的名字空间层次或安全性来模仿文件。

在文件和非文件内容之间的名字空间和编程模型上实现的对称性提供了应用程序用于随时间的推移将来自文件系统的内容移植到存储平台数据存储中的更结构化的项目的更好路径。通过提供存储平台数据存储中的本机文件项目类型，应用程序可将文件数据转移到存储平台，同时仍能够通过 Win32 来操纵该数据。最终，应用程序可能完全移植到存储平台 API，且按照存储平台项目而非文件来结构化其数据。

2. 数据存储特征

为提供期望的互操作性等级，在一个实施例中，实现存储平台数据存储的以下特征。

a) 非卷

存储平台数据存储不被展现为单独的文件系统卷。存储平台充分利用了直接主存在 NTFS 上的 FILESTREAM。由此，没有对盘上格式的改变，由此消除了在卷级将存储平台展现为新文件系统的需求。

相反，数据存储（名字空间）是对应于 NTFS 卷来构造的。后退名字空间的该部分的数据库和 FILESTREAM 位于与存储平台数据存储相关联的 NTFS 卷上。也提供了对应于系统卷的数据存储。

b) 存储结构

存储的结构最好用一个示例来说明。作为一个示例，考虑名为 HomeMachine 的机器的系统卷上的目录树，如图 16 所示。依照本发明的文件系统互操作性特征，对应于 c:\drive，存在通过例如称为“WinFSOnC”的 UNC 共享展现给 Win32 API 的存储平台数据存储。这使得相关联的数据存储可通过以下 UNC 名来访问：

\HomeMachine\WinFSOnC。

在该实施例中，文件和/或文件夹需要从 NTFS 显式地移植到存储平台。因此，如果用户希望将 My Documents 文件夹移至存储平台数据存储以利用由存储平台提

供的所有额外的搜索/分类特征，则该分层结构将如图 17 所示。重要的是注意，这些文件夹在本示例中实际移动。另一点要注意的是，名字空间移至存储平台中，实际的流被重命名为 FILESTREAM，在存储平台内挂钩适当的指针。

c) 不移植所有文件

对应于用户数据或需要存储平台提供的搜索/分类的文件是移植到存储平台数据存储的候选者。较佳地，为限制与存储平台的应用程序兼容性问题，在 Microsoft Windows 操作系统的上下文中，移植到本发明的存储平台的该组文件被限于 MyDocuments 文件夹中的文件、Internet Explorer(IE) Favorites、IE History 和 Documents and Settings 目录中的 Desktop.ini 文件。较佳地，移植 Windows 系统文件是不允许的。

d) 对存储平台文件的 NTFS 名字空间访问

在此处所描述的实施例中，期望移植到存储平台的文件不能通过 NTFS 名字空间来访问，即使实际文件流是以 NTFS 储存的。以此方式，避免了由多线程实现引起的复杂锁定和安全性问题。

e) 期望的名字空间/驱动器字母

对存储平台中的文件和文件夹的访问是通过\\<机器名>\<WinFS 共享名>形式的 UNC 名来提供的。对于需要驱动器字母用于操作的应用程序类，可将驱动器字母映射到该 UNC 名。

I. 存储平台 API

如上所述，存储平台包括 API，它使应用程序能访问上面讨论的存储平台的特征和能力，并访问存储在数据存储中的项目。本节描述本发明的存储平台的存储平台 API 的一个实施例。

图 19 示出按本实施例的存储平台 API 的基本体系结构。存储平台 API 使用 SQL 客户机 1900 与本地数据存储 302 对话，并还使用 SQL 客户机 1900 与远程数据存储（如数据存储 340）对话。本地存储还可使用 DQP（分布式查询处理器）或通过上述的存储平台同步服务（“Sync”）与远程数据存储 340 对话。存储平台 API 322 还担当数据存储通知的桥接器 API，将应用程序的下标传送到通知引擎 332，

并如上所述将通知路由到应用程序（如应用程序 350a、350b 或 350c）。在一个实施例中，存储平台 API 322 还定义受限制的“提供者”体系结构，使得它能访问 Microsoft Exchange 和 AD 中的数据。

1. 综述

本发明的存储平台 API 的本实施例的数据访问机制解决了四个领域：查询、导航、动作、事件。

查询

在一个实施例中，存储平台数据存储是在关系型数据库引擎 314 上实现的；结果，SQL 语言的完全表达性能力在存储平台中是固有的。较高级查询对象提供了用于查询存储的简化模型，但是可能不封装存储的完全表达性能力。

导航

存储平台数据模型在底层数据库抽象上构建了一种丰富的、可扩展的类型系统。对于开发者而言，存储平台数据是项目的网。存储平台 API 允许通过过滤、关系、文件夹等在项目之间导航。这是比基本 SQL 查询更高级的抽象；同时，它允许对熟悉的 CLR 编码模式使用丰富的过滤和导航能力。

动作

存储平台 API 展现所有项目上公用的动作—Create（创建）、Delete（删除）、Update（更新）；这些被展现为对象上的方法。另外，诸如 SendMail（发送邮件）、CheckFreeBusy（检查空闲和忙碌）等域专用动作也作为方法可用。API 框架使用 ISV 可用于通过定义附加动作来添加值的良好定义的模式。

事件

存储平台中的数据是动态的。为使应用程序在存储中的数据改变时做出反应，API 向开发者展现了丰富的事件、订阅和通知能力。

2. 命名和范围

在名字空间和命名之间进行区分是有用的。如常用的，术语名字空间指的是某一系统中可用的所有名字的集合。系统可以是 XML 模式、程序、web、所有 ftp 站点（及其内容）的集合等等。命名是用于向名字空间内所有感兴趣的条目分配唯一名字的过程或算法。由此，命名是感兴趣的，因为期望无歧义地涉及名字空间内

的给定单元。由此，如此处所使用的，术语“名字空间”指的是全域中所有存储平台示例中可用的所有名字的集合。项目是存储平台名字空间中的命名实体。UNC 命名约定用于确保项目名字的唯一性。全域中每一存储平台存储中的每一项目可通过 UNC 名字来寻址。

存储平台名字空间中的最高组织级别是服务—它仅仅是存储平台的一个实例。组织的下一级是卷。卷是项目的最大匿名容器。每一存储平台实例包含一个或多个卷。在卷内的是项目。项目是存储平台中的数据原子。

真实世界中的数据几乎总是依照在给定领域中有意义的某一系统来组织。在所有这样的数据组织模式下的是将数据的全域划分成命名组的概念。如上所述，这一概念在存储平台中由文件夹的概念来建模。文件夹是一种特殊类型的项目；有两种类型的文件夹：包含文件夹和虚拟文件夹。

参考图 18，包含文件夹是一个项目，它包含与其它项目的持有关系，且与通常概念的文件系统文件夹等价。每个项目“包含”在至少一个包含文件夹中。

虚拟文件夹是组织项目集合的更动态方法；它仅仅是给定一组项目的名字—该组项目是显式地枚举或通过查询指定的。虚拟文件夹本身是项目，且可以被认为是表示与一组项目的一组（非持有）关系。

有时候，需要对包含的更紧密概念进行建模；例如，嵌入在电子邮件消息中的 Word 文档在某种意义上比例如包含在文件夹中的文件更紧密地绑定到其容器。这一概念是由嵌入项目概念来表达的。嵌入项目具有一种特殊的关系，它引用另一项目；引用的项目可仅绑定到包含项目或仅在包含项目的上下文中操纵。

最后，存储平台提供了类别的概念作为对项目和元素的分类的方式。存储平台中的每一项目或元素可具有与其相关联的一个或多个类别。类别本质上只是被标记到项目/元素上的名字。该名字可以在搜索中使用。存储平台数据模型允许定义类别的分层结构，从而允许数据的树状分类。

项目的一种无歧义的名字是三元组：(<服务名>, <卷 ID>, <项目 ID>)。某些项目（尤其是文件夹和虚拟文件夹）是其它项目的集合。这引起一种标识项目的替换方式：(<服务名>, <卷 ID>, <项目路径>)。

存储平台名字包括服务上下文的概念：服务上下文是映射到(<卷名>, <路径>)对的名字。它标识了一个项目或一组项目—例如，文件夹、虚拟文件夹等。对于服务上下文，用于存储平台名字空间中的任何项目的 UNC 名成为：

\<服务名>\<服务上下文>\<项目路径>

用户可创建和删除服务上下文。同样，每一卷中的根目录具有预定义的上下文：卷名\$。

项目上下文通过限制返回给存活在指定路径中的项目的结果来定查询（例如，查找操作）的范围。

3. 存储平台 API 组件

图 20 示意性地表示依照本发明的本实施例的存储平台 API 的各种组件。存储平台 API 包括下列组件：(1) 数据类 2002，它代表存储平台元素和项目类型；(2) 运行库架构 2004，它管理对象的持久性并提供支持类 2006；以及(3) 工具 2008，它用于从存储平台模式生成 CLR 类。

依照本发明的一个方面，在设计时，模式作者向一组存储平台 API 设计时向工具 2008 提交模式文档 2010 以及用于域方法的代码 2012。这些工具生成客户机方数据类 2002 和存储模式 2014 以及该模式的存储类定义 2016。“域”指的是特定的模式；例如，谈论联系人模式中的类的域方法等等。这些数据类 2002 在运行时由应用程序开发者与存储平台 API 运行库架构类 2006 协作使用来操纵存储平台数据。

为说明本发明的存储平台 API 的各方面，基于示例性联系人模式呈现了若干示例。该示例性模式的图示表示在图 21A 和 21B 中示出。

4. 数据类

依照本发明的一方面，存储平台数据存储中的每一项目、项目扩展和元素类型以及每一关系在存储平台 API 中具有对应的类。大致上，类型的字段映射到类的字段。存储平台中的每一项目、项目扩展和元素作为存储平台 API 中的对应类的对象可用。开发者可查询、创建、修改或删除这些对象。

存储平台包括一组初始模式。每一模式定义了一组项目和元素类型，以及一组关系。以下是用于从这些模式实体生成数据类的算法的一个实施例：

对于每一模式 S：

对于 S 中的每一项目 I，生成名为 System.Storage.S.I 的类。该类具有以下成员：

- 重载的构造函数，包括允许指定新项目的初始文件夹和名字的构造函数。
- I 中每一字段的属性。如果字段是多值的，则属性将是对应的元素类型的

集合。

- 找出匹配过滤器的多个项目的重载的静态方法（例如，名为“FindAll” 的方法）。
- 找出匹配过滤器的单个项目的重载的静态方法（例如，名为“FindOne” 的方法）。
- 给定其 ID 找出项目的静态方法（例如，名为“FindByID” 的方法）。
- 给定其相对于项目上下文的名字找出项目的静态方法（例如，名为 “FindByName” 的方法）。
- 保存对项目的改变的方法（例如，名为“Update” 的方法）。
- 创建项目的新实例的重载的静态 Create 方法。这些方法允许以各种方式指定项目的初始文件夹。

对于 S 中的每一元素 E，生成名为 System.Storage.S.E 的类。该类具有以下成员：

- E 中的每一字段的属性。如果字段是多值的，则属性将是对应的元素类型的集合。

对于 S 中的每一元素 E，生成名为 System.Storage.S.ECollection 的类。该类遵循用于强类型化的集合类的通用.NET 构架方针。对于基于关系的元素类型，该类还将包括以下成员：

- 找出匹配隐式地包括其中集合在源角色中出现的项目的过滤器的项目的多个对象的重载的方法。重载包括允许基于项目子类型的过滤的某些方法（例如，名为“FindAllTargetItems” 的方法）。
- 找出匹配隐式地包括其中集合在源角色中出现的项目的过滤器的单个项目对象的重载的方法。重载包括允许基于项目子类型来过滤的某些方法（例如，名为“FindOneTargetItem” 的方法）。
- 找出匹配隐式地包括其中集合在源角色中出现的项目的过滤器的嵌套元素类型的对象的重载的方法（例如，名为“FindAllRelationships” 的方法）。
- 找出匹配隐式地包括其中集合在源角色中出现的项目的过滤器的嵌套元素类型的对象的重载的方法（例如，名为“FindAllRelationshipsForTarget” 方法）。
- 找出匹配隐式地包括其中集合在源角色中出现的项目的过滤器的嵌套元素类型的单个对象的重载的方法（例如，名为“FindOneRelationship” 的

方法)。

- 找出匹配隐式地包括其中集合在源角色中出现的项目的过滤器的嵌套元素类型的单个对象的重载的方法(例如,名为“FindOneRelationshipForTarget”的方法)。

对于 S 中的关系 R, 生成名为 System.Storage.S.R 的类。该类具有一个或两个子类, 取决于是一个还是两个关系角色指定了端点字段。

也可以此方式为所创建的每一项目扩展创建类。

数据类存在于 System.Storage.<模式名>名字空间中, 其中<模式名>是对应的模式的名字—诸如 Contacts(联系人)、Files(文件)等。例如, 对应于联系人模式的所有类在 System.Storage.Contacts 名字空间中。

作为示例, 参考图 21A 和 21B, 联系人模式导致包含在 System.Storage.Contact 名字空间中的以下类:

- 项目: Item、Folder、WellKnownFolder、LocalMachineDataFolder、UserDataFolder、Principal、Service、GroupService、PersonService、PresenceService、ContactService、ADService、Person、User、Group、Organization、HouseHold
- 元素: NestedElementBase、NestedElement、IdentityKey、SecurityID、EAddress、ContactEAddress、TelephoneNumber、SMTPEAddress、InstantMessagingAddress、Template、TemplateRelationship、LocationRelationship、FamilyEventLocationRelationship、HouseHoldLocationRelationship、RoleOccupancy、EmployeeData、GroupMemberShip、OrganizationLocationRelationship、HouseHoldMemberData、FamilyData、SpouseData、ChildData

作为另一示例, 如在联系人模式中定义的 Person 类型的详细结构以 XML 示出如下:

```
<Type Name="Person" MajorVersion="1" MinorVersion="0"
ExtendsType="Core.Principal" ExtendsVersion="1">

<Field Name="Birthdate" Type="the storage platformTypes.datetime"
Nullable="true" TypeMajorVersion="1"/>

<Field Name="Gender" Type="Base.CategoryRef"
Nullable="true" MultiValued="false"
TypeMajorVersion="1"/>

<Field Name="PersonalNames" Type="Contact.FullName"
Nullable="true" MultiValued="true"
TypeMajorVersion="1"/>

<Field Name="PersonalEAddresses" Type="Core.EAddress"
Nullable="true" MultiValued="true"
TypeMajorVersion="1"/>

<Field Name="PersonalPostalAddresses"
Type="Core.PostalAddress" Nullable="true"
MultiValued="true" TypeMajorVersion="1"/>

<Field Name="PersonalPicture" Type="the storage platformTypes.image"
Nullable="true" TypeMajorVersion="1"/>

<Field Name="Notes" Type="Core.RichText" Nullable="true"
MultiValued="true" TypeMajorVersion="1"/>

<Field Name="Profession" Type="Base.CategoryRef"
Nullable="true" MultiValued="true"
TypeMajorVersion="1"/>

<Field Name="DataSource" Type="Base.IdentityKey"
Nullable="true" MultiValued="true"
TypeMajorVersion="1"/>

<Field Name="ExpirationDate" Type="the storage platformTypes.datetime"
Nullable="true" TypeMajorVersion="1"/>

<Field Name="HasAllAddressBookData" Type="the storage platformTypes.bit"
Nullable="true" TypeMajorVersion="1"/>

<Field Name="EmployeeOf" Type="Contact.EmployeeData"
Nullable="true" MultiValued="true"
TypeMajorVersion="1"/>

</Type>
```

该类型导致以下类（仅示出了公有成员）：

```
partial public class Person :  
    System.Storage.Core.Principal,  
    System.Windows.Data.IDataUnit  
{  
  
    public System.Data.SqlTypes.SqlDateTime  
        Birthdate { get; set; }  
  
    public System.Storage.Base.CategoryRef  
        Gender { get; set; }  
  
    public System.Storage.Contact.FullNameCollection  
        PersonalNames { get; }  
  
    public System.Storage.Core.EAddressCollection  
        PersonalEAddresses { get; }  
  
    public System.Storage.Core.PostalAddressCollection  
        PersonalPostalCodes { get; }  
  
    public System.Data.SqlTypes.SqlBinary  
        PersonalPicture { get; set; }  
  
    public System.Storage.Core.RichTextCollection  
        Notes { get; }  
  
    public System.Storage.Base.CategoryRefCollection  
        Profession { get; }  
  
    public System.Storage.Base.IdentityKeyCollection  
        DataSource { get; }  
  
    public System.Data.SqlTypes.SqlDateTime  
        ExpirationDate { get; set; }  
  
    public System.Data.SqlTypes.SqlBoolean  
        HasAllAddressBookData { get; set; }  
  
    public System.Storage.Contact.EmployeeDataCollection  
        EmployeeOf { get; }  
  
    public Person();  
  
    public Person( System.Storage.Base.Folder folder, string name );  
  
    public static new System.Storage.FindResult  
        FindAll( System.Storage.ItemStore store );  
  
    public static new System.Storage.FindResult  
        FindAll(  
            System.Storage.ItemStore store,  
            string filter );  
  
    public static new Person  
        FindOne(  
            System.Storage.ItemStore store,
```

```
        string filter );  
  
    public new event  
        System.Windows.Data.PropertyChangedEventHandler  
        PropertyChangedHandler;  
  
    public static new Person  
        FindByID(  
            System.Storage.ItemStore store,  
            long item_key );  
}
```

作为又一示例，联系人模式中定义的 TelephoneNumber 类型的详细结构以 XML 示出如下：

```
<Type Name="TelephoneNumber" ExtendsType="Core.EAddress"  
MajorVersion="1" MinorVersion="0" ExtendsVersion="1">  
  
<Field Name="CountryCode" Type="the storage platformTypes.nvarchar(50)"  
Nullable="true" MultiValued="false"  
TypeMajorVersion="1"/>  
  
<Field Name="AreaCode" Type="the storage platformTypes.nvarchar(256)"  
Nullable="true" TypeMajorVersion="1"/>  
  
<Field Name="Number" Type="the storage platformTypes.nvarchar(256)"  
Nullable="true" TypeMajorVersion="1"/>  
  
<Field Name="Extension" Type="the storage platformTypes.nvarchar(256)"  
Nullable="true" TypeMajorVersion="1"/>  
  
<Field Name="PIN" Type="the storage platformTypes.nvarchar(50)"  
Nullable="true" TypeMajorVersion="1"/>  
  
</Type>
```

该类型导致以下类（仅示出了公有成员）：

```
partial public class TelephoneNumber :  
    System.Storage.Core.EAddress,  
    System.Windows.Data.IDataUnit  
{  
  
    public System.Data.SqlTypes.SqlString CountryCode  
    { get; set; }  
  
    public System.Data.SqlTypes.SqlString AreaCode  
    { get; set; }  
  
    public System.Data.SqlTypes.SqlString Number  
    { get; set; }  
  
    public System.Data.SqlTypes.SqlString Extension  
    { get; set; }  
  
    public System.Data.SqlTypes.SqlString PIN  
    { get; set; }  
}
```

从给定模式得到的类的分层结构直接反映了该模式中的分层结构。作为一个示例，考虑联系人模式中定义的 Item 类型（见图 21A 和 21B）。在存储平台 API 中对应于该类型的类分层结构如下：

```
Object  
  DataClass  
    ElementBase  
      RootItemBase  
        Item  
          Principal  
            Group  
            Household  
            Organization  
            Person  
              User  
                Service  
                  PresenceService  
                  ContactService  
                  ADSERVICE  
                    RootNestedBase  
                      ...(元素类)
```

又一模式，即允许表示系统中的所有音频/视频媒体（断开的音频文件、音频 CD、DVD、家庭视频等）的模式使用户/应用程序能够存储、组织、搜索和操纵不同种类的音频/视频媒体。基本媒体文档模式足够一般以表示任何媒体，且对该基本模式的扩展被设计成为音频和视频媒体分开处理域专用属性。该模式以及许多其它模式被构想为直接或间接在核心模式下操作。

5. 运行时架构

基本存储平台 API 编程模型是对象持久性的。应用程序在存储上执行搜索，并在存储中检索表示数据的对象。应用程序修改检索的对象或创建新对象，然后使其改变被传播到存储。该过程是由 ItemContext（项目上下文）对象来管理的。搜索使用 ItemSearcher（项目搜索器）对象来执行，且搜索结果可通过 FindResult（寻找结果）对象来访问。

a) 运行时架构类

依照本发明的另一发明性方面，运行时架构实现多个类来支持数据类的操作。这些架构类为数据类定义了一组公共的行为，且连同数据类一起为存储平台 API 提供了基本编程模型。运行时架构中的类属于 System.Storage 名字空间。在本实施例中，架构类包括以下主要类：ItemContext、ItemSearcher 和 FindResult。也可提供其它次要类、枚举值以及代表。

(1) ItemContext

ItemContext 对象 (i) 表示应用程序希望搜索的一组项目域，(ii) 为每一对象维护表示从存储平台中检索的数据的状态的状态信息，以及 (iii) 管理当与存储平台交互时使用的事务以及可与存储平台互操作的任何文件系统。

作为对象持久性引擎，ItemContext 提供了以下服务：

1. 将从存储中读取的数据反串行化成对象。
2. 维护对象身份（同一对象用于表示给定的项目，而无论该项目被包括在查询结果中多少次）。
3. 跟踪对象状态。

ItemContext 也执行对存储平台唯一的多个服务：

1. 生成和执行持久保存改变所需的存储平台更新元素操作。

2. 创建允许引用关系的无缝导航并允许从多域搜索中检索要修改和保存的对象所需的到多个数据存储的连接。
3. 确保当对象的改变表示要保存该项目时支持文件的项目被正确地更新。
4. 管理跨多个存储平台连接的事务，并且在更新储存在支持文件的项目中的数据以及文件流属性时管理以事务方式处理的文件系统。
5. 执行将存储平台关系语义、支持文件的项目以及流类型化的属性考虑在内的创建、复制、移动和删除操作。

附录 A 提供了依照其一个实施例列出 ItemContext 类的源代码。

(2) ItemSearcher

ItemSearcher 类支持简单的搜索，该搜索返回整个 Item 对象、Item 对象的流、或从 Item 对象投影的值的流。ItemSearcher 封装了对以下所有公共的核心功能：应目标类型的概念以及应用于该目标类型的参数化的过滤器。ItemSearcher 也允许预编译或准备搜索器，作为在以多种类型执行同一操作时的优化。附录 B 提供了依照其一个实施例列出 ItemSearcher 类和若干紧密相关的类的源代码。

(a) 目标类型

搜索目标类型是在构造 ItemSearcher 时设置的。目标类型是映射到可由数据存储查询的范围的 CLR 类型。具体地，它是映射到项目、关系和关系扩展类型以及模式化视图的 CLR 类型。

当使用 ItemContext.GetSearcher 方法检索搜索器时，搜索器的目标类型被指定为参数。当在项目、关系或项目扩展类型上调用静态 GetSearcher 方法（例如，Person.GetSearcher）时，目标类型是项目、关系或项目扩展类型。

ItemSearcher 中提供的搜索表达式（例如，搜索过滤器和通过查询操作，或投影定义）总是相对于搜索目标类型。这些表达式可指定目标类型的属性（包括嵌套元素的属性），并可指定到如别处所描述的关系和项目扩展的联结。

搜索目标类型可通过只读属性（例如，Item.Searcher.Type 属性）获得。

(b) 过滤器

ItemSearcher 包含指定过滤器的属性（例如，名为“Filters”的属性，作为 SearchExpression 对象的集合），该过滤器定义了搜索中使用的过滤器。集合中的所有过滤器在执行搜索时使用逻辑和运算符来组合。过滤器可包含参数引用。参数

值是通过 Parameters 属性来指定的。

(c) 准备搜索

在可能仅用一个参数改变来重复执行同一搜索的情况下，可通过预编译或准备搜索来提高某些性能。这是用 ItemSearcher 上的一组准备方法（例如，准备返回一个或多个项目的 Find（查找）的方法，可能名为“PrepareFind”；以及准备返回投影的 Find 的方法，可能名为“PrepareProject”）来实现的。例如：

```
ItemSearcher searcher = ...;
PreparedFind pf = searcher.PrepareFind();
...
result = pf.FindAll();
...
result = pf.FindAll();
```

(d) 查找选项

存在可应用于简单搜索的多个选项。这些可在例如 FindOptions（查找选项）对象中指定并被传递到 Find 方法。例如：

```
ItemSearcher searcher = Person.GetSearcher( context );
FindOptions options = new FindOptions();
options.MaxResults = 10;
options.SortOptions.Add( "PersonalNames.Surname", SortOrder.Ascending );
FindResult result = searcher.FindAll( options );
```

作为一种便利，也可将排序选项直接传递给 Find 方法：

```
ItemSearcher searcher = Person.GetSearcher( context );
FindResult result = searcher.FindAll(
    new SortOption( "PersonalNames.Surname", SortOrder.Ascending ) );
```

DelayLoad（延迟负载）选项确定当检索搜索结果时是否加载大二进制属性的值，或者是否将加载延迟到它们被引用之后。MaxResults（最大结果）选项确定了返回的结果的最大数量。这等效于在 SQL 查询中指定 TOP。它通常结合排序来使用。

SortOption（排序选项）对象的序列可被指定（例如，使用 FindOptions.SortOptions 属性）。搜索结果将如由第一个 SortOption 对象所指定的那样排序，然后如由第二个 SortOption 对象所指定的排序，依此类推。SortOption

指定了指示将用于排序的属性的搜索表达式。表达式指定了以下之一：

1. 搜索目标类型中的标量属性；
2. 可通过遍历单值属性从搜索目标类型到达的嵌套元素中的标量属性；或者
3. 具有有效自变量的集合函数的结果（例如，Max 应用于可通过遍历多值属性或关系从搜索目标类型到达的嵌套元素中的标量属性）。

例如，假定搜索目标类型是 System.Storage.Contact.Person：

1. “Birthdate” 一有效，Birthdate 是 Person 类型的标量属性；
2. “PersonalNames.Surname” 一无效，PersonalNames 是多值属性，且未使用任何集合函数；
3. “Count(PersonalNames)” 一有效，PersonalNames 的计数；
4. “Case(Contact.MemberOfHousehold).Household.HouseholdEAddresses.StartDate” 一无效，使用了关系和多值属性，而没有集合函数。
5. “Max(Cast(Contact.MemberOfHousehold).Household.HouseholdEAddresses.StartDate)” 一有效，最近的家庭电子地址起始日期。

(3) 项目结果流 (“FindResult”)

ItemSearcher（例如，通过 FindAll 方法）返回可用于访问由搜索返回的对象的对象（例如，“FindResult”对象）。附录 C 提供了依照其一个实施例列出 FindResult 类和若干紧密相关的类的源代码。

存在用于从 FindResult 对象获得结果的两种不同的方法：使用由 IObjectReader（对象读取器接口）（以及 IAsyncObjectReader（异步对象读取器接口））定义的读取器模式，以及使用 IEnumerable（可枚举接口）和 IEnumerator（枚举器接口）定义的枚举器模式。枚举器模式是 CLR 中的标准，且支持如 C# 的 foreach 等语言构造。例如：

```
ItemSearcher searcher = Person.GetSearcher( context );
searcher.Filters.Add( "PersonalNames.Surname = 'Smith' " );
FindResult result = searcher.FindAll();
foreach( Person person in result ) ...;
```

支持读取器模式，因为它允许通过在某些情况下消除数据副本更有效地处理结果。例如：

```

ItemSearcher searcher = Person.GetSearcher( context );
searcher.Filters.Add( "PersonalNames.SurName = 'Smith'" );
FindResult result = searcher.FindAll();
while( result.Read() )
{
    Person person = (Person)result.Current;
    ...
}

```

另外，读取器模式支持异步操作：

```

ItemSearcher searcher = Person.GetSearcher( context );
searcher.Filters.Add( "PersonalNames.SurName = 'Smith'" );
FindResult result = searcher.FindAll();
IAyncResult asyncResult = result.BeginRead( new AsyncCallback( MyCallback ) );

void MyCallback( IAyncResult asyncResult )
{
    if( result.EndRead( asyncResult ) )
    {
        Person person = (Person)result.Current;
        ...
    }
}

```

在本实施例中，FindResult 在它不再需要时应当被关闭。这可以通过调用 Close（关闭）方法或使用诸如 C# 所使用的语句等语言构造来完成。例如：

```

ItemSearcher searcher = Person.GetSearcher( context );
searcher.Filters.Add( "PersonalNames.SurName = 'Smith'" );
using( FindResult result = searcher.FindAll() )
{
    while( result.Read() )
    {
        Person person = (Person)result.Current;
        ...
    }
}

```

b) 操作中的运行时架构

图 22 示出了操作中的运行时架构。该运行时架构如下操作：

1. 应用程序 350a、350b 或 350c 绑定到存储平台中的项目。
2. 架构 2004 创建对应于绑定的项目的 ItemContext(项目上下文)对象 2202，并将其返回给应用程序。
3. 应用程序在该 ItemContext 上提交 Find 以获得项目集合；所返回的集合

在概念上是对象图 2204 (由于关系)。

4. 应用程序改变、删除和插入数据。
5. 应用程序通过调用 Update()方法保存改变。

c) 公共编程模式

本节提供了存储平台 API 架构类如何能够用于操纵数据存储中的项目的各种示例。

(1) 打开和关闭 ItemContext 对象

应用程序例如通过调用静态的 ItemContext.Open 方法并提供标识将与 ItemContext 相关联的项目域的一个或多个路径来获得它将用于与数据存储交互的 ItemContext 对象。项目域定使用 ItemContext 所执行的搜索的范围，使得仅域项目和包含在该项目中的项目将遭受搜索。示例如下：

打开在本地计算机上具有 DefaultStore 存储平台共享的 ItemContext

```
ItemContext ic = ItemContext.Open()
```

打开具有给定存储平台共享的 ItemContext

```
ItemContext ic = ItemContext.Open(@"\\myserver1\\DefaultStore");
```

打开具有在存储平台共享下的项目的 ItemContext

```
ItemContext ic = ItemContext.Open(@"\\myserver1\\WinFSSpecs\\api\\m6");
```

打开具有多个项目域的 ItemContext

```
ItemContext ic = ItemContext.Open(@"\\myserver1\\My Documents",
                                @"\\jane1\\My Documents",
                                @"\\jane2\\My Documents");
```

当不再需要 ItemContext 时，它必须被关闭。

显式地关闭 ItemContext

```
ItemContext ic = ItemContext.Open();
```

```
...
ic.Close();
```

使用具有 ItemContext 的语句来关闭

```
using(ItemContext ic = ItemContext.Open())
{
    ...
}
```

(2) 搜索对象

依照本发明的另一方面，存储平台 API 提供了使得应用程序员能够基于数据存储中的项目的各种属性，以将应用程序员与底层数据库引擎的查询语言的细节隔离开的方式来形成查询的简化的查询模型。

应用程序可跨在使用由 ItemContext.GetSearcher 方法返回的 ItemSearcher 对象打开 ItemContext 时指定的域执行搜索。搜索结果使用 FindResult 对象来访问。假定对以下示例的以下声明：

```
ItemContext ic = ...;
ItemSearcher searcher = null;
FindResult result = null;
Item item = null;
Relationship relationship = null;
ItemExtension itemExtension = null;
```

基本搜索模式涉及使用通过调用 GetSearcher(获得搜索器)方法从 ItemContext 检索到的 ItemSearcher 对象。

搜索给定类型的所有项目

```
searcher = ic.GetSearcher(typeof(Person));
result = searcher.FindAll();
foreach(结果中的 Person p)...;
```

搜索满足过滤器的给定类型的项目

```
searcher = ic.GetSearcher(typeof(Person));
searcher.Filters.Add("PersonalNames.Surname = 'Smith'");
result = searcher.FindAll();
foreach(结果中的 Person p)...;
```

使用过滤器串中的参数

```
searcher = ic.GetSearcher(typeof(Person));
searcher.Filters.Add("Birthdate < @Date");
searcher.Parameters["Date"] = someDate;
result = searcher.FindAll();
foreach(结果中的 Person p)...;
```

搜索给定类型且满足过滤器的关系

```
searcher = ic.GetSearcher(typeof(EmployeeEmployer));
searcher.Filters.Add("StartDate <= @Date AND (EndDate >= @Date OR isnull(EndDate))");
searcher.Parameters["Date"] = someDate;
result = searcher.FindAll();
Foreach(结果中的 EmployeeEmployer ee)...;
```

搜索具有给定类型且满足过滤器的关系的项目

```
searcher = ic.GetSearcher(typeof(Folder));
searcher.Filters.Add("MemberRelationships.Name like 'A%'"); //见[ApiRel]
result = searcher.FindAll();
foreach(结果中的 Folder f)...;
```

搜索给定类型且满足过滤器的项目扩展

```
searcher = ic.GetSearcher(typeof(ShellExtension));
searcher.Filters.Add("Keywords.Value = 'Foo'");
result = searcher.FindAll();
foreach(结果中的 ShellExtension se)...;
```

搜索具有给定类型且满足过滤器的项目扩展的项目

```
searcher = ic.GetSearcher(typeof(Person));
searcher.Filters.Add("Extensions.Cast(@Type).Keywords.Value = 'Foo'");//见[ApiExt]
searcher.Parameters["Type"] = typeof(ShellExtension);
result = searcher.FindAll();
foreach(结果中的 Person p)...;
```

(a) 搜索选项

可在执行搜索时指定各种选项，包括排序、延迟加载以及限制结果数。

排序搜索结果

```
searcher = ic.GetSearcher(typeof(Person));
searcher.Filters.Add("PersonalNames.Surname = 'Smith'");
SearchOptions options = new SearchOptions();
options.SortOptions.Add(new SortOption("Birthdate", SortOrder.Ascending));
result = searcher.FindAll(options);
foreach(结果中的 Person p)...;
```

//有快捷方式可用

```
searcher = ic.GetSearcher(typeof(Person));
searcher.Filters.Add("PersonalNames.Surname = 'Smith'");
result = searcher.FindAll(new SortOption("Birthdate", SortOrder.Ascending));
foreach(结果中的 Person p)...;
```

限制结果计数

```
searcher = ic.GetSearcher(typeof(Person));
searcher.Filters.Add("PersonalNames.Surname = 'Smith'");
SearchOptions options = new SearchOptions();
options.MaxResults = 10;
result = searcher.FindAll(options);
foreach(结果中的 Person p)...;
```

(b) **FindOne** 和 **FindOnly**

有时仅检索第一个结果是有用的，尤其是当指定排序准则的时候。另外，期望某些搜索仅返回一个对象，且不期望它不返回任何对象。

搜索一个对象

```
searcher = ic.GetSearcher( typeof( Person ) );
searcher.Filters.Add( "PersonalNames.Surname = 'Smith' " );
Person p = searcher.FindOne( new SortOption( "Birthdate" SortOrder.Ascending ) ) as Person;
if( p != null ) ...;
```

搜索期望总是存在的单个对象

```
searcher = ic.GetSearcher( typeof( Person ) );
searcher.Filters.Add( "PersonalNames[Surname = 'Smith' AND Givenname 'John']" );
try
{
    Person p = searcher.FindOnly();
    ...
}
catch( Exception e )
{
    ...
}
```

(c) 搜索 ItemContext 上的快捷方式

ItemCount 上也有使得简单的搜索尽可能容易的若干快捷方式。

使用 ItemContext.FindAll 快捷方式搜索

```
result = ic.FindAll(typeof(Person), "PersonalNames.Surname = 'Smith'");
foreach(结果中的 Person p)...;
```

使用 Item.Context.FindOne 快捷方式搜索

```
Person p = ic.FindOne(typeof(Person), "PersonalNames.Surname = 'Smith'" ) as Person;
```

(d) 按照 ID 或路径查找

另外，可通过提供其 id 来检索项目、关系以及项目扩展。项目也可按照路径来检索。

给定 id 获得项目、关系以及项目扩展

```
item = ic.FindItemById( iid );
relationship = ic.FindRelationshipById( iid, rid );
itemExtension = ic.FindItemExtensionById( iid, eid );
```

给定路径获得项目

```
//仅单个域
item = ic.FindItemByPath(@"temp\foo.txt");
```

```
//单个或多个域
result = ic.FindAllItemsByPath(@"temp\foo.txt");
foreach(结果中的 Item I)...;
```

(e) GetSearcher 模式

在存储平台 API 中有许多期望提供在另一对象的上下文中或用特定的参数来执行搜索的助手方法的许多地方。GetSearcher 模式允许这些情形。在 API 中有许多 GetSearcher 方法。其每一个都返回预先被配置成执行给定搜索的 ItemSearcher。例如：

```
searcher = itemContext.GetSearcher();
searcher = Person.GetSearcher();
searcher = EmployeeEmployer.GetSearcherGivenEmployer( organization );
searcher = person.GetSearcherForReports();
```

可在执行搜索之前添加其它过滤器：

```
searcher = person.GetSearcherForReports();
searcher.Filters.Add( "PersonalNames.Surname='Smith'" );
```

可选择如何希望结果：

```
FindResult findResult = searcher.FindAll();
Person person = searcher.FindOne();
```

(3) 更新存储

一旦通过搜索检索了对象，它可按应用程序所需修改。也可创建新对象并与现有对象相关联。一旦应用程序做出了形成逻辑组的所有改变，应用程序调用 ItemContext.Update 来将这些改变持久保存到存储中。依照本发明的存储平台 API 的另一方面，API 收集由应用程序对项目做出的改变，然后将它们组织成其上实现数据存储的数据库引擎（或任何种类的存储引擎）所需的正确更新。这使得应用程序员能够对存储器中的项目做出改变，而将数据存储更新的复杂性留给 API。

保存对单个项目的改变

```
Person p = ic.FindItemById( pid ) as Person;
p.DisplayName = "foo";
p.TelephoneNumbers.Add( new TelephoneNumber( "425-555-1234" ) );
ic.Update();
```

保存对多个项目的改变

```
Household h1 = ic.FindItemById( hid1 ) as Household;
Household h2 = ic.FindItemById( hid2 ) as Household;
Person p = ic.FindItemById( pid ) as Person;
h1.MemberRelationships.Remove( p );
h2.MemberRelationships.Add( p );
ic.Update();
```

创建新项目

```
Folder f = ic.FindItemById( fid ) as Folder;
Person p = new Person();
p.DisplayName = "foo";
f.Relationships.Add( new FolderMember( p, "foo" ) );
ic.Update();
```

//或使用快捷方式...

```
Folder f = ic.FindItemById( fid ) as Folder;
Person p = new Person();
p.DisplayName = "foo";
f.MemberRelationships.Add( p, "foo" );
ic.Update();
```

删除关系（以及可能的目标项目）

```
searcher = ic.GetSearcher( typeof( FolderMember ) );
searcher.Filters.Add( "SourceItemId=@fid" );
searcher.Filters.Add( "TargetItemId=@pid" );
searcher.Parameters.Add( "fid", fid );
searcher.Parameters.Add( "pid", pid );
foreach(searcher.FindAll()中的 FolderMember fm) fm.MarkForDelete();
ic.Update();
```

//或使用快捷方式...

```
Folder f = ic.FindItemById( fid ) as Folder;
f.MemberRelationships.Remove( pid );
ic.Update();
```

添加项目扩展

```
Item item = ic.FindItemById( iid );
MyExtension me = new MyExtension();
me.Foo = "bar";
item.Extensions.Add( me );
ic.Update();
```

删除项目扩展

```
searcher = ic.GetSearcher( typeof( MyExtension ) );
searcher.Filters.Add( "ItemId=@iid" );
searcher.Parameters.Add( "iid", iid );
```

```
foreach(searcher.FindAll()中的 MyExtension me) me.MarkForDelete();
ic.Update();

//或使用快捷方式...

Item i = ic.FindItemById( iid );
i.Extensions.Remove( typeof( MyExtension ) );
ic.Update();
```

6. 安全性

参考上文的第 II.E 节（安全性），在存储平台 API 的本实施例中，在项目上下文上有五种方法可用于检索和修改与存储中的项目相关联的安全策略。这些方法是：

1. GetItemSecurity;
2. SetItemSecurity;
3. GetPathSecurity;
4. SetPathSecurity；以及
5. GetEffectiveItemSecurity。

GetItemSecurity 和 SetItemSecurity 提供了检索和修改与项目相关联的显式 ACL 的机制。该 ACL 独立于所存在的到项目的路径，且在进行中独立于具有该项目作为目标的持有关系。这使得管理员能够如所期望的独立于所存在的到项目的路径推断出项目的安全性。

GetPathSecurity 和 SetPathSecurity 提供了用于检索和修改由于来自另一文件夹的持有关系而存在于项目上的 ACL 的机制。该 ACL 由考虑中的路径从各个祖先的 ACL 到该项目以及对该路径的显式 ACL（如果提供的话）组成。该 ACL 和前一情况下的 ACL 之间的差别在于该 ACL 仅当对应的持有关系存在时才在进行中保留，而显式项目 ACL 独立于对该项目的任何持有关系。

可以在项目上用 SetItemSecurity 和 SetPathSecurity 设置的 ACL 被限于可继承的和对象专用的 ACE。它们不能包含被标记为继承的任何 ACE。

GetEffectiveItemSecurity 基于 ACL 以及项目上的显式 ACL 来检索各种路径。这反映了给定项目上有效的授权策略。

7. 对关系的支持

如上所述，存储平台的数据模型定义了允许项目彼此相关的“关系”。当生

成模式的数据类时，为每一关系类型产生以下类：

1. 表示关系本身的类。该类从 Relationship 类导出，且包含对关系类型专用的成员。
2. 强类型化的“虚拟”集合类。该类从 VirtualRelationshipCollection（虚拟关系集合）导出，且允许创建和删除关系实例。

本节描述了存储平台 API 中对关系的支持。

a) 基础关系类型

存储平台 API 提供了 System.Storage 名字空间中形成关系 API 的基础的多种类型。这些类型是：

1. Relationship（关系）—所有关系类的基础类型
2. VirtualRelationshipCollection（虚拟关系集合）—所有关系集合的基础类型
3. ItemReference（项目引用）、ItemIdReference（项目 ID 引用）、ItemPathReference（项目路径引用）—表示项目引用类型；这些类型之间的关系在图 11 中示出。

(1) Relationship 类

```
public abstract class Relationship: StoreObject
{
    //用默认值创建
    protected Relationship(ItemIDReference targetItemReference);

    //通知关系它被添加到关系集合。对象将询问集合来确定源项目、项目上下文等。
    internal AddedToCollection(virtualRelationshipCollection collection);

    //关系的 ID
    public RelationshipId RelationshipId{get;}

    //源项目的 ID
    public ItemId SourceItemId{get;}

    //获得源项目
    public Item SourceItem{get;}

    //对目标项目的引用
    public ItemIdReference TargetItemReference{get;}

    //获得目标项目（调用 TargetItemReference.GetItem()）
    public Item TargetItem{get;}

    //确定 ItemContext 是否已经具有到目标项目的域的连接
    //（调用 TargetItemReference.IsDomainConnected）
}
```

```

public bool IsTargetDomainConnected{get;}

//目标项目在名字空间中的名字。该名字必须在所有源项目的持有关系上是唯一的
public OptionalValue<string> Name{get; set;}

//确定这是持有还是引用关系
public OptionalValue<bool> IsOwned{get; set;}
}

```

(2) ItemReference 类

以下是项目引用类型的基类。

```

public abstract class ItemReference : NestedElement
{
    //用默认值创建
    protected ItemReference();

    //返回引用的项目
    public virtual Item GetItem();

    //确定是否建立了到引用的项目的域的连接
    public virtual bool IsDomainConnected();
}

```

ItemReference 对象可标识存在于除项目引用本身所驻留的存储之外的存储中的项目。每一导出的类型指定了如何构造和使用对远程存储的引用。导出类中 GetItem 和 IsDomainConnected 的实现使用 ItemContext 的多域支持来从所需的域中加载项目，并确定是否已建立了到该域的连接。

(3) ItemIdReference 类

以下是ItemIdReference 类—使用项目 ID 来标识目标项目的项目引用。

```

public class ItemIdReference : ItemReference
{
    //用默认值构造新的 ItemIdReference
    public ItemIdReference();

    //构造对指定项目的新 ItemIdReference。与该项目相关联的域用作定位符
    public ItemIdReference(Item item);

    //用空定位符和给定的目标项目 ID 构造新的 ItemIdReference
    public ItemIdReference(ItemId itemId);

    //用给定定位符和项目 ID 值构造新的 ItemIdReference
    public ItemIdReference(string locator, ItemId itemId);

    //目标项目的 ID
    public ItemId ItemId{get; set;}

    //标识在其域中包含目标项目的 WinFS 项目的路径。如果为空，则包含该项目的域是未知的。
    public OptionalValue<string> Locator {get; set;}
}

```

```

//确定是否建立了到所引用的项目的域的连接
public override bool IsDomainConnected();

//检索所引用的项目
public override Item GetItem();
}

```

GetItem 和 IsDomainConnected 使用 ItemContext 的多域支持来从所需的域中加载项目，并确定是否已经建立了到该域的连接。该特征尚未实现。

(4) ItemPathReference 类

ItemPathReference 类是使用路径来标识目标项目的项目引用。用于该类的代码如下：

```

public class ItemPathReference : ItemReference
{
    //用默认值构造项目路径引用
    public ItemPathReference();

    //不用任何定位符但用给定路径构造项目路径引用
    public ItemPathReference(string path);

    //用给定定位符和路径构造项目路径引用
    public ItemPathReference(string locator, string path);

    //标识在其域中包含目标项目的 WinFS 项目的路径
    public OptionalValue<string> Locator{get; set;}

    //相对于由定位符指定的项目域的目标项目路径
    public string Path{get; set;}

    //确定是否已经建立了到所引用的项目的域的连接
    public override bool IsDomainConnected();

    //检索所引用的项目
    public override Item GetItem();
}

```

GetItem 和 IsDomainConnected 使用 ItemContext 的多域支持来从所需的域中加载项目，并确定是否已经建立了到该域的连接。

(5) RelationshipId 结构

RelationshipId 结构封装了关系 ID GUID。

```

public class RelationshipId
{
    //生成新的关系 ID GUID
    public static RelationshipId NewRelationshipId();

    //用新的关系 ID GUID 初始化
    public RelationshipId();
}

```

```

//用指定的 GUID 初始化
public RelationshipId(Guid id);

//用 GUID 的串标识初始化
public RelationshipId(string id);

//返回关系 ID GUID 的串标识
public override string ToString();

//将 System.Guid 实例转换成 RelationshipId 实例
public static implicit operator RelationshipId(Guid guid);

//将 RelationshipId 实例转换成 System.Guid 实例
public static implicit operator Guid(RelationshipId relationshipId);
}

```

该值类型包装了 GUID，使得参数和属性可被强类型化为关系 ID。OptionalValue<RelationshipId> 应当在关系 ID 可为空时使用。诸如由 System.Guid.Empty 提供的空值不被展现。RelationshipId 不能用空值来构造。当使用默认的构造函数来创建 RelationshipId 时，创建新的 GUID。

(6) VirtualRelationshipCollection 类

VirtualRelationshipCollection 类实现了包括来自数据存储的对象加上被添加到集合的新对象，但不包括从存储中移除的对象的关系对象的集合。给定源项目 ID 的指定关系类型的对象被包括在该集合中。

这是为每一关系类型生成的关系集合类的基类。该类可用作源项目类型中的属性的类型，以提供对给定项目的关系的访问和简易操纵。

枚举 VirtualRelationshipCollection 的内容可能要求从存储中加载大量的关系对象。应用程序应当在枚举该集合的内容之前使用 Count（计数）属性来确定可加载多少关系。向集合添加对象/从集合移除对象不需要从存储中加载关系。

为效率起见，较佳的是应用程序搜索满足特定准则的关系，而非使用 VirtualRelationshipCollection 对象枚举所有的项目关系。向集合添加关系对象使得在调用 ItemContext.Update 时在存储中创建所表示的关系。从集合中移除关系对象使得当调用 ItemContext.Update 时在存储中删除所表示的关系。虚拟集合包含正确的对象集，而不管是否通过 Item.Relationships 集合或该项目上的任何其它关系集合添加/移除了关系对象。

以下代码定义了 VirtualRelationshipCollection 类：

```

public abstract class VirtualRelationshipCollection : ICollection
{

```

```

//该集合包含由项目 itemId 标识的项目所拥有的类型的关系
protected VirtualRelationshipCollection(ItemCount itemContext,
                                         ItemId itemId,
                                         Type relationshipType);

//枚举器将返回从存储中检索到的所有对象减去除具有状态 Inserted (已插入) 的对象
//之外的具有状态 Deleted (已删除) 的任何对象
public IEnumerator GetEnumerator();

//返回由枚举器返回的关系对象的数目的计数。该计数无需从存储中检索所有对象即可计算。
public int Count{get;}

//总是返回假
public bool ICollection.IsSynchronized(){get;}

//总是返回该对象
public object ICollection.SyncRoot{get;}

//搜索存储中的所需对象
public void Refresh();

//向集合添加指定的关系。对象必须具有状态 Constructed (已构造) 或 Removed (已移除)。
//如果状态为 Constructed, 它被改为 Added (已添加)。如果状态是 Removed, 则它适当地
//改为 Retrieved (已检索) 或 Modified (已修改)。关系的源项目 ID 必须与当构造集合时
//提供的源项目 ID 相同。
protected void Add(Relationship relationship);

//从集合中移除指定的关系。对象的状态必须是 Added、Retrieved 或 Modified。
//如果对象状态是 Added, 它将被设为 Constructed。如果对象状态是 Retrieved 或 Modified,
//它将被设为 Removed。关系的源项目 ID 必须与当构造集合时提供的源项目 ID 相同。
protected void Remove(Relationship relationship);

//从集合中移除的对象
public ICollection RemovedRelationships{get;}

//被添加到集合的对象
public ICollection AddedRelationships{get;}

//从存储中检索的对象。该集合将为空, 直到枚举了 VirtualRelationshipCollection
//或调用了 Refresh 之后 (获得该属性的值不导致填满该集合)。
public ICollection StoredRelationships{get;}

//匿名方法

public IAsyncResult BeginGetCount(IAsyncResult callback, object state);
public int EndGetCount(IAsyncResult asyncResult);

public IAsyncResult BeginRefresh(IAsyncResult callback, object state);
public void EndRefresh(IAsyncResult asyncResult);
}

```

b) 生成的关系类型

当为存储平台模式生成类时, 为每一关系声明生成一个类。除表示关系本身的类之外, 也为每一关系生成关系集合类。这些类用作关系的源或目标项目类中的属性类型。

本节描述了使用若干“原型”类生成的类。即，给定指定的关系声明，描述了生成的类。重要的是注意，原型类中使用的类、类型和端点名是关系模式中指定的名字的占位符，且不应当在文字上解释。

(1) 生成的关系类型

本节描述了为每一关系类型生成的类。例如：

```
<Relationship Name="RelationshipPrototype" BaseType="Holding">
  <Source Name="Head" ItemType="Foo"/>
  <Target Name="Tail" ItemType="Bar" ReferenceType="ItemIdReference" />
  <Property Name="SomeProperty" Type="WinFSTypes.String" />
</Relationship>
```

给定该关系定义 *RelationshipPrototype*（关系原型），将生成 *RelationshipPrototypeCollection*（关系原型集合）类。*RelationshipPrototype* 类表示关系本身。*RelationshipPrototypeCollection* 类提供了对具有指定的项目作为源端点的 *RelationshipPrototype* 实例的访问。

(2) RelationshipPrototype 类

这是用于名为“*HoldingRelationshipPrototype*（持有关系原型）”的持有关系的原型关系类，其中源端点名为“*Head*”，且指定了“*Foo*”项目类型，而目标端点名为“*Tail*”，且指定了“*Bar*”项目类型。它定义如下：

```
public class RelationshipPrototype : Relationship
{
    public RelationshipPrototype(Bar tailItem);
    public RelationshipPrototype(Bar tailItem, string name);
    public RelationshipPrototype(Bar tailItem, string name, bool IsOwned);
    public RelationshipPrototype(Bar tailItem, bool IsOwned);
    public RelationshipPrototype(ItemIdReference tailItemReference);

    //获得 Head 项目（调用 base.SourceItem）
    public Foo HeadItem{get;}

    //获得 Tail 项目（调用 base.TargetItem）
    public Bar TailItem{get;}

    //表示关系模式中声明的其它属性。这些是正如项目或嵌套元素类型中的属性那样生成的。
    public string SomeProperty{get; set;}

    public static ItemSearcher GetSearcher(ItemContext itemContext);
    public static ItemSearcher GetSearcher(Foo headItem);

    public static FindResult FindAll(string filter);
    public static RelationshipPrototype FindOne(string filter);
    public static RelationshipPrototype FindOnly(string filter);
}
```

(3) RelationshipPrototypeCollection 类

这是用 *RelationshipPrototype* 类生成的原型类，它维护由指定的项目拥有的 *RelationshipPrototype* 关系实例的集合。它定义如下：

```
public class RelationshipPrototypeCollection : VirtualRelationshipCollection
{
    public RelationshipPrototypeCollection( ItemContext itemContext,
                                             ItemId headItemId );

    public void Add( RelationshipPrototype relationship );
    public RelationshipPrototype Add( Bar bar );
    public RelationshipPrototype Add( Bar bar, string name );
    public RelationshipPrototype Add( Bar bar, string name, bool IsOwned );
    public RelationshipPrototype Add( Bar bar, bool IsOwned );

    public void Remove( RelationshipPrototype relationship );
    public void Remove( Bar bar );
    public void Remove( ItemId barItemId );
    public void Remove( RelationshipId relationshipId );
    public void Remove( string name );
}
```

c) Item 类中的关系支持

Item 类包含提供对其中该项目是关系的源的关系的访问的 Relationships 属性。 Relationships 属性具有类型 RelationshipCollection。

(1) Item 类

以下代码示出了 *Item* 类的关系上下文属性：

```
public abstract class Item : StoreObject
{
    ...
    //其中该项目是源的关系的集合
    public RelationshipCollection Relationships{get;}
    ...
}
```

(2) RelationshipCollection 类

该类提供了对其中给定项目是关系的源的关系实例的访问。它定义如下：

```

public class RelationshipCollection : VirtualRelationshipCollection
{
    public RelationshipCollection( ItemContext itemContext,
        ItemId headItemId );

    public void Add( Relationship relationship );
    public Relationship Add( Bar bar );
    public Relationship Add( Bar bar, string name );
    public Relationship Add( Bar bar, string name, bool IsOwned );
    public Relationship Add( Bar bar, bool IsOwned );

    public void Remove( Relationship relationship );
    public void Remove( Bar bar );
    public void Remove( ItemId barItemId );
    public void Remove( RelationshipId relationshipId );
    public void Remove( string name );
}

```

d) 搜索表达式中的关系支持

可能在搜索表达式中指定关系和相关项目之间的联结的遍历。

(1) 从项目遍历到关系

当搜索表达式的当前上下文是一组项目时，项目和其中该项目是源的关系实例之间的联结可使用 Item.Relationships 属性来完成。联结到特定类型的关系可使用搜索表达式的 Cast（类型强制转换）运算符来指定。

强类型化的关系集合（例如，Folder.MemberRelationships）也可在搜索表达式中使用。到关系类型的类型强制转换是隐式的。

一旦建立了该组关系，该关系的属性可用于判定或作为投影的目标。当用于指定投影的目标时，将返回该组关系。例如，以下语句将找出涉及其中关系的 StartDate 属性具有大于或等于“1/1/2000”的值的组织的所有人。

```
FindResult result = Person.FindAll( context,
    "Relationships.Cast(Contact.EmployeeOfOrganization).StartDate > '1/1/2000'" );
```

如果 Person 类型具有类型为 EmployeeSideEmployerEmployeeRelationships 的属性 EmployerContext（对 EmployeeEmployer 关系类型生成的），则这将被写为：

```
FindResult result = Person.FindAll( context,
    "EmployerRelationships.StartDate > '1/1/2000'" );
```

(2) 从关系遍历到项目

当搜索表达式的当前上下文是一组关系时，可通过指定端点名来遍历从关系到关系的任一端点的联结。一旦建立了该组相关项目，那些项目的属性可用于判定

或作为投影的目标。当用于指定投影的目标时，将返回该组项目。例如，以下语句将找出其中雇员的姓是名字“Smith”的所有 EmployeeOfOrganization(组织的雇员)关系（而不论组织是什么）：

```
FindResult result = EmployeeOfOrganization.FindAll( context,
    "Employee.PersonalNames[SurName='Smith']");
```

搜索表达式 Cast 运算符可用于过滤端点项目的类型。例如，为找出其中成员是具有姓“Smith”的 Person 项目的所有 MemberOfFolder(文件夹成员)关系实例：

```
FindResult result = MemberOfFolder.FindAll( context,
    "Member.Cast(Contact.Person).PersonalNames[Surname='Smith']");
```

(3) 组合关系遍历

前两种模式，即从项目遍历到关系以及从关系遍历到项目可被组合以实现任意复杂的遍历。例如，为找出具有姓为“Smith”的雇员的所有组织：

```
FindResult result = Organization.FindAll( context,
    "EmployeeRelationships." +
    "Employee." +
    "PersonalNames[SurName = 'Smith']");
```

以下示例将找出表示生活在“New York”区域的家庭的人的所有 Person 项目（计划项目：这不再支持……它是一种替换）。

```
FindResult result = Person.FindAll( context,
    "Relationships.Cast(Contact.MemberOfHousehold)." +
    "Household." +
    "Relationships.Cast(Contact.LocationOfHousehold)." +
    "MetropolitonRegion = 'New York'");
```

e) 关系支持的示例使用

以下是如何使用存储平台 API 中的关系支持来操纵关系的示例。对于以下示例，假定以下声明：

```
ItemContext ic = ...;
ItemId fid = ...; //文件夹项目的 ID
Folder folder = Folder.FindById(ic, fid);
ItemId sid = ...; //源项目的 ID
Item source = Item.FindById(ic, sid);
ItemId tid = ...; //目标项目的 ID
Item target = Item.FindById(ic, tid);
ItemSearcher searcher = null;
```

(1) 搜索关系

可能搜索源或目标关系。可使用过滤器来选择指定类型且具有给定属性值的关系。也可使用过滤器来选择基于关系的相关项目类型或属性值。例如，可执行以下搜索：

其中给定项目是源的所有关系

```
searcher = Relationship.GetSearcher(folder);
foreach(searcher.FindAll()中的 Relationship relationship)...;
```

其中给定项目是具有匹配“A%”的名字的源的所有关系

```
searcher = Relationship.GetSearcher(folder);
searcher.Filters.Add("Name like 'A%'");
foreach(searcher.FindAll()中的 Relationship relationship)...;
```

其中给定项目是源的所有 FolderMember 关系

```
searcher = FolderMember.GetSearcher(folder);
foreach(searcher.FindAll()中的 FolderMember folderMember)...;
```

其中给定项目是源且名如“A%”的所有 FolderMember 关系

```
searcher = FolderMember.GetSearcher(folder);
searcher.Filters.Add("Name like 'A%'");
foreach(searcher.FindAll()中的 FolderMember folderMember)...;
```

其中目标项目是 Person 的所有 FolderMember 关系

```
searcher = FolderMember.GetSearcher(folder);
searcher.Filters.Add("MemberItem.Cast(Person)");
foreach(searcher.FindAll()中的 FolderMember folderMember)...;
```

其中目标项目是姓为“Smith”的 Person 的所有 FolderMember 关系

```
searcher = FolderMember.GetSearcher(folder);
searcher.Filters.Add("MemberItem.Cast(Person).PersonalNames.Surname='Smith'");
foreach(searcher.FindAll()中的 FolderMember folderMember)...;
```

除以上示出的 GerSearcher API 之外，每一关系类支持静态 FindAll、FindOne 以及 FindOnly API。另外，关系类型可以在调用 ItemContext.GetSearcher、ItemContext.FindAll、ItemContext.FindOne 或 ItemContext.FindOnly 时指定。

(2) 从关系导航到源和目标项目

一旦通过搜索检索了关系对象，可能“导航”到目标或源项目。基本关系类提供了返回 Item 对象的 SourceItem（源项目）和 TargetItem（目标项目）属性。所生成的关系类提供了等效强类型化和命名的属性（例如，FolderMember.FolderItem

和 FolderMember.MemberItem)。例如：

导航到名为“Foo”的关系的源和目标项目

```
searcher = Relationship.GetSearcher();
searcher.Filters.Add("Name='Foo'");
foreach(searcher.FindAll()中的 Relationship relationship)
{
    Item source = relationship.SourceItem;
    Item target = relationship.TargetItem;
}
```

导航到目标项目

```
searcher = FolderMember.GetSearcher(folder);
searcher.Filters.Add("Name like 'A%'");
foreach(searcher.FindAll()中的 FolderMember folderMember)
{
    Item member = folderMember.TargetItem;
    ...
}
```

导航到目标项目即使在目标项目不在其中找到关系的域中也可运作。在这一情况下，存储平台 API 打开到所需的目标域的连接。应用程序可在检索目标项目之前确定是否需要连接。

检查未连接的域中的目标项目

```
searcher = Relationship.GetSearcher(source);
foreach(searcher.FindAll()中的 Relationship relationship)
{
    if(relationship.IsTargetDomainConnected)
    {
        Item member = relationship.TargetItem;
        ...
    }
}
```

(3) 从源项目导航到关系

给定项目对象，可能导航到该项目为其源的关系，而无需执行显式的搜索。这可使用 Item.Relationships 集合属性或诸如 Folder.MemberRelationships 等强类型化的集合属性来完成。从关系中，可能导航到目标项目。这一导航即使在目标项目不在与源项目的 ItemContext 相关联的项目域中时也能运作，包括当目标项目不在与源项目相同的存储中的时候。例如：

从源项目导航到关系再到目标项目

```
Console.WriteLine("Item{0} is the source of the following relationships:", source.ItemId);
foreach(source.Relationships 中的 Relationship relationship)
{
```

```

    Item target = relationship.TargetItem;
    Console.WriteLine("{0} ==> {1}", relationship.RelationshipId, target.ItemId);
}

```

从文件夹项目导航到文件夹成员关系再到目标项目

```

console.WriteLine("Item{0} is the source of the following relationships:", folder.ItemId);
foreach(folder.MemberRelationships 中的 FolderMember folderMember)
{
    Item target = folderMember.GetMemberItem();
    Console.WriteLine("{0} ==> {1}", folderMember.RelationshipId, target.ItemId);
}

```

项目可具有许多关系，因此应用程序在枚举关系集合时应谨慎使用。一般而言，应使用搜索来标识感兴趣的特定关系而非枚举整个集合。然而，对关系具有基于集合的编程模型以及足够有价值，且具有许多关系的项目少得足以证明开发者滥用的风险。应用程序可检查集合中的关系的数目，并在需要时使用不同的编程模型。例如：

检查关系集合的大小

```

if( folder.MemberRelationships.Count > 1000 )
{
    Console.WriteLine( "Too many relationships!" );
}
else
{
    ...
}

```

上述关系集合在它们实际上不用表示每一关系的对象来填充的意义上是“虚拟的”，除非应用程序试图枚举该集合。如果该集合被枚举，则结果反映了存储中有什么，加上应用程序添加但没有保存了什么，但不反映由应用程序移除但没有保存的任何关系。

(4) 创建关系（以及项目）

新关系通过创建关系对象、将其添加到源项目中的关系集合、并更新项目上下文来创建。为创建新项目，必须创建持有或嵌入关系。例如：

向现有文件夹添加新项目

```

Bar bar = new Bar();
folder.Relationships.Add( new FolderMember( bar, "name" ) );
ic.Update();

```

//或者

```
Bar bar = new Bar();
folder.MemberRelationships.Add( new FolderMember( bar, "name" ) );
ic.Update();
```

//或者

```
Bar bar = new Bar();
folder.MemberRelationships.Add( bar, name );
ic.Update();
```

向现有文件夹添加现有项目

```
folder.MemberRelationships.Add( target, "name" );
ic.Update();
```

向新文件夹添加现有项目

```
Folder existingFolder = ic.FindItemById( fid ) as Folder;
Folder newFolder = new Folder();
existingFolder.MemberRelationships.Add( newFolder, "a name" );
newFolder.MemberRelationships.Add( target, "a name" );
ic.Update();
```

向新文件夹添加新项目

```
Folder existingFolder = ic.FindItemById( fid ) as Folder;
Folder newFolder = new Folder();
existingFolder.MemberRelationships.Add( newFolder, "a name" );
Bar bar = new Bar();
newFolder.MemberRelationships.Add( bar, "a name" );
ic.Update();
```

(5) 删除关系（以及项目）

删除持有关系

```
//如果源项目和关系 ID 已知...
RelationshipId rid = ...;
Relationship r = ic.FindRelationshipById( fid, rid );
r.MarkForDelete();
ic.Update();

//否则...
folder.MemberRelationships.Remove( target );
ic.Update();
```

8. “扩展” 存储平台 API

如上所述，每一存储平台模式都导致一组类。这些类具有诸如 Find* 等标准方法，且还具有用于设置和获得字段值的属性。这些类和相关联的方法形成了存储平台 API 的基础。

a) 域行为

除这些标准方法之外，每一模式具有一组用于该模式的域专用方法。将这些方法成为域行为。例如，联系人模式中某些域行为是：

- 电子邮件地址是否有效？
- 给定文件夹，获得该文件夹的所有成员的集合。
- 给定项目 ID，获得表示该项目的对象。
- 给定人，获得其在线状态。
- 助手函数创建新联系人或临时联系人。
- 等等。

重要的是注意，尽管在“标准”行为（Find*等）和域行为之间做出了区别，但是它们对程序员仅仅表现为方法。这些方法之间的区别在于标准行为是由存储平台 API 设计时工具从模式文件自动生成的，而域行为是硬编码的。

就其最本性而言，域行为应当是手工制作的。这导致一个实际问题：C#的初始版本要求类的整个实现在单个文件内。由此，这迫使自动生成的类文件必须被编辑以添加域行为。这本身就是一个问题。

在 C# 中为诸如此类的问题引入一种称为局部类的特征。最基本地，局部类允许类实现跨越多个文件。局部类与常规的类相同，不同之处在于其声明前有关键字 partial：

```
partial public class Person : DerivedItemBase
{
    //实现
}
```

现在，Person 的域行为可以被放置在不同的文件中，如：

```
partial public class Person
{
    public EmailAddress PrimaryEmailAddress
    {
        get{/*实现*/}
    }
}
```

b) 增值行为

具有域行为的数据类形成了应用程序开发者在其上进行构建的基础。然而，数据类展现与该数据有关的每一可想到的行为是不可能也是不合需要的。存储平台允许开发者在由存储平台 API 提供的基础功能上进行构建。此处的基础模式是编

写其方法采用存储平台数据类的一个或多个作为参数的类。例如，用于使用 Microsoft Outlook 或使用 Microsoft Windows Messenger 来发送电子邮件的增值类可以如下：

```
MailMessage m = MailMessage.FindOne(...);  
OutlookEMailServices.SendMessage(m);  
  
Person p = Person.FindOne(...);  
WindowsMessengerServices m = new WindowsMessengerServices(p);  
m.MessageReceived += new MessageReceivedHandler(f);  
m.SendMessage("Hello");
```

这些增值类可以向存储平台注册。注册数据与存储平台为每一安装的存储平台类型维护的模式元数据相关联。该元数据作为存储平台项目储存，且可被查询。

增值类的注册是一种强大的特征；例如，它允许以下情形：Shell 资源管理器中的 Person 对象上右键点击，且可从对 Person 注册的增值类中导出允许的该组动作。

c) 作为服务提供者的增值行为

在本实施例中，存储平台 API 提供了一种机制，其中增值类可被注册为给定类型的“服务”。这使得应用程序能够设置和获得给定类型的服务提供者（=增值类）。希望利用这一机制的增值类应当实现一种公知的接口；例如：

```
interface IChatServices  
{  
    void SendMessage(string msg);  
    event MessageReceivedHandler MessageReceived;  
}  
  
class WindowsMessengerServices : IChatServices  
{  
    ...  
}  
  
class YahooMessengerServices : IChatServices  
{  
    ...  
}
```

所有存储平台 API 数据类实现 ICachedServiceProvider（高速缓存的服务提供者）接口。该接口扩展了 System.IServiceProvider 接口，如下：

```
interface ICachedServiceProvider : System.IServiceProvider
{
    void SetService(System.Type type, Object provider);
    void RemoteService(System.Type type);
}
```

使用该接口，应用程序可设置服务提供者示例以及请求特定类型的服务提供者。

为支持该接口，存储平台数据类维护以类型作为键的服务提供者的散列表。

当请求服务提供者时，实现首先在散列表中查看是否设置了指定类型的服务提供者。如果否，则使用注册的服务提供者基础结构来标识指定类型的服务提供者。然后创建该提供者的实例，将其添加到散列表，并返回。注意，也可能由数据类上的共享方法请求服务提供者，并将操作转发到该提供者。例如，这可用于提供使用由用户指定的电子邮件系统的邮件消息类上的 Send 方法。

9. 设计时架构

本节描述了依照本发明的本实施例，存储平台模式如何转换成客户机上的存储平台 API 以及服务器上的 UDT 类。图 24 的图示出了所涉及的组件。

参考图 24，模式中的类型包含在 XML 文件（框 1）中。该文件也包含与模式相关联的字段级和项目级约束。存储平台类生成器（xfs2cs.exe—框 2）采用该文件，并生成存储 UDT 的局部类（框 5）以及客户机类的局部类（框 3）。对于每一模式域，存在其它方法—称为域行为。存在在存储（框 7）、客户机（框 6）和两者中（框 4）有意义的域行为。框 4、6 和 7 中的代码是手写的（不是自动生成的）。框 3、4 和 6 中的局部类共同形成了存储平台 API 域类的完整类实现。框 3、4 和 6 被编译（框 8）以形成存储平台 API 类—框 11（实际上，存储平台 API 是编译从所有初始模式域所得的框 3、4 和 6 的结果）。除域类之外，也存在实现增值行为的其它类。这些类利用一个或多个模式域中的一个或多个类。这是由框 10 表示的。框 4、5 和 7 中的局部类共同形成了服务器 UDT 类的完整类实现。框 4、5 和 7 被编译（框 9）以形成服务器方的 UDT 程序集—框 12（实际上，服务器方的 UDT 程序级是编译器加编译从所有初始模式域所得的框 4、5 和 7 的结果）。DDL 命令生成器模块（框 13）采用该 UDT 程序集（框 12）以及模式文件（框 1），并将它们安装在数据存储上。该过程涉及表的生成以及每一模式中的类型的查看等等。

10. 查询形式

当被精简到基础时，在使用存储平台 API 时应用程序的模式为：打开 ItemContext；以过滤器准则使用 Find 来检索期望的对象；在对象上操作；以及将改变发送回存储。本节涉及对于什么进入过滤器串的句法。

当查找存储平台数据对象时提供的过滤器串描述了对象的属性必须满足以被返回的约束。由存储平台 API 使用的句法支持类型强制转换和关系遍历。

a) 过滤器基础

过滤器串要么为空，指示要返回指定类型的所有对象；要么是每一返回的对象必须满足的布尔表达式。表达式引用对象的属性。存储平台 API 运行库知道这些属性名如何映射到存储平台类型字段名，并最终映射到由存储平台存储维护的 SQL 视图。

考虑以下示例：

```
//找出所有人
FindResult res1 = Person.FindAll(ctx)

//找出其 Gender (性别) 属性值等于 “Male” (男) 的所有人
FindResult res2 = Person.FindAll(ctx, "Gender='Male'")

//找出其 Gender 属性值等于 “Male” 且在上一世纪出生的人
FindResult res3 = Person.FindAll(
    ctx,
    "Gender = 'Male' And Birthdate < '1/1/2001'''")
```

嵌套对象的属性也可在过滤器中使用。例如：

```
//找出在上 24 小时中修改的所有人
FindResult res1 = Person.FindAll(
    ctx,
    String.Format("Item.Modified > '{0}'", DateTime.NowSubtract(new TimeSpan(24, 0, 0))));
```

对于集合，可能使用方括号中的条件来过滤成员。例如：

```
//找出名为 “John” 且姓为 “Smith”的所有人
FindResult res1 = Person.FindAll(
    ctx,
    "PersonalNames[GivenName = 'John' And Surname = 'Smith']")

//找出实时地址来自提供者 “x” 且在线状态类别为 “y”的所有人
FindResult res2 = Person.FindAll(
    ctx,
    "PersonalRealtimeAddress[ProviderURI = 'x'].BasicPresence"+
    "OnlineStatus.Category='y'")
```

以下示例列出了自从 12/31/1999 以来出生的所有人：

```

ItemContext ctx = ItemContext.Open("Work Contacts");

FindResult results =
    Person.FindAll(ctx, "Birthdate > '12/31/1999'");

foreach(results 中的 Person person)
    Console.WriteLine(person.DisplayName);

ctx.Close();

```

第 1 行创建了新的 ItemContext 对象来访问本地计算机上存储平台共享上的“Work Contacts（工作联系人）”。第 3 行和第 4 行获得其中 Birthdate（生日）属性满足晚于 12/31/1999 的日期的 Person 对象的集合，如由表达式“Birthdate > '12/31/1999'”所指定的。该 FindAll 操作的执行在图 23 中示出。

b) 类型强制转换

情况通常是储存在属性中的值的类型是从属性声明的类型导出的。例如，Person 中的 PersonalEAddress（个人电子地址）属性包含从诸如 EMailAddress（电子邮件地址）和 TelephoneNumber（电话号码）等 EAddress（电子地址）中导出的类型的集合。为基于电话区号过滤，必须从 EAddress 类型强制转换到 TelephoneNumber 类型：

```

//找出电话号码 425 区号中的所有人
FindResult res1 = Person.FindAll(
    ctx,
    "PersonalEAddresses." +
    "Cast(System.Storage.Contact.TelephoneNumber))." +
    "AreaCode='425'");

//或者，可如下传递类型名：
FindResult res1 = Person.FindAll(
    ctx,
    String.Format("PersonalEAddresses.Cast({0}).AreaCode='425'",
        typeof(TelephoneNumber).FullName))

```

c) 过滤器句法

以下是依照一个实施例由存储平台 API 支持的过滤器句法的描述。

过滤器 ::= 空过滤器|条件

空过滤器 ::=

条件 ::= 简单条件|复合条件|加括号的条件

简单条件 ::= 存在检查|比较

存在检查 ::= 属性引用

比较 ::= 属性引用 比较运算符 常量

复合条件 ::= 简单条件 布尔运算符 条件

加括号的条件 ::= '(' 条件 ')'

比较运算符 ::= '!= | '==' | '=' | '<' | '>' | '>=' | '<='

布尔运算符 ::= 'And' | '&&' | 'Or' | '||'

常量 ::= 串常量|数字常量

串常量 ::= "" (任何 Unicode 字符)* ""

注意：嵌入的'字符通过复制来转义

数字常量 ::= 0-9*

属性引用 ::= 简单属性名|复合属性名

简单属性名 ::= (除'.'和空格之外的所有 Unicode 字符)* 过滤器?

过滤器 ::= '(' 条件 ')'

复合属性名 ::= (类型强制转换|关系遍历|简单属性名) ! 属性引用

类型强制转换 ::= 'Cast (' 类型名 ')

关系遍历 ::= 遍历到源|遍历到目标

遍历到源 ::= 'Source (' 完整关系名 ')

遍历到目标 ::= 'Target (' 完整关系名 ')

类型名 ::= 完全合格的 CLR 类型名

完整关系名 ::= 模式名 ! 关系名

模式名 ::= 存储平台名

关系名 ::= 存储平台名

存储平台名 ::= 如[模式定义]中所定义的

11. 遥控

a) API 中的本地/远程透明性

存储平台中的数据访问的目标为本地存储平台实例。如果查询（或其部分）涉及远程数据，则本地实例用作路由器。由此，API 层提供了本地/远程透明性：在本地和远程数据访问之间没有结构上的 API 差别。它纯粹是由所请求的范围来决定的。

存储平台数据存储也实现了分布式查询；由此，可能连接到本地存储平台实

例，并执行包括来自不同卷的项目的查询，其中某一些项目在本地存储上，而其它项目在远程存储上。存储对结果执行并操作，并将其呈现给应用程序。从存储平台 API 的观点（因此是应用程序开发者的观点）来看，任何远程访问完全是无缝和透明的。

存储平台 API 允许应用程序使用 `IsRemote` 属性确定给定的 `ItemContext` 对象（如由 `ItemContext.Open` 方法返回的）是表示本地还是远程连接，该属性是 `ItemContext` 对象上的属性。尤其是，应用程序可能希望提供视频反馈来帮助为性能、可靠性等设置用户期望。

b) 遥控的存储平台实现

存储平台数据存储使用特殊的 OLEDB 提供者来彼此交谈，该提供者在 HTTP 上运行（默认的 OLEDB 提供者使用 TDS）。在一个实施例中，分布式查询通过关系型数据库引擎的默认 OPENROWSET 功能。提供一种特殊的用户定义函数（UDF）：`DoRemoteQuery(server, queryText)`，以完成实际的遥控。

c) 访问非存储平台存储

在本发明的存储平台的一个实施例中，不存在允许任何存储参与存储平台数据访问的非一般提供者体系结构。然而，为 Microsoft Exchange 和 Microsoft Active Directory (AD) 的特定情况提供了有限提供者体系结构。这意味着开发者可如同在存储平台中那样使用存储平台 API 并访问 AD 和 Exchange 中的数据，但是他们能够访问的数据限于存储平台模式化的类型。由此，在 AD 中支持地址簿 (=存储平台 Person 类型的集合)，且对 Exchange 支持邮件、日历和联系人。

d) 与 DFS 的关系

存储平台属性升级器不升级过去的固定点。即使名字空间足够丰富以通过固定点来访问，查询仍不能通过它们。存储平台卷可以作为 DFS 树中的叶节点出现。

e) 与 GXA/Indigo 的关系

开发者可使用存储平台 API 以在数据存储的上方展示“GXA 头”。概念上，创建任何其它 web 服务没有任何区别。存储平台 API 不使用 GXA 与存储平台数据存储交谈。如上所述，API 使用 TDS 与本地存储交谈；任何遥控是由本地存储使用同步服务来处理的。

12. 约束

存储平台数据模型允许类型上的值约束。这些约束在存储上自动评估，且该过程对于用户是透明的。注意，在服务器处检查约束。注意到这一点之后，有时候期望给予开发者验证输入数据满足约束的灵活性而不会导致对服务器的往返额外开销。这在其中最终用户输入用于填充对象的数据的交互式应用程序中尤其有用。存储平台 API 提供该工具。

可以回想在 XML 文件中指定了存储平台模式，它由存储平台用于生成表示该模式的适当的数据库对象。它也由存储平台 API 的设计时架构用于自动生成类。

以下是用于生成 Contacts 模式的 XML 文件的部分清单：

```
<Schema Name="Contacts" MajorVersion="1" MinorVersion="8">
  <ReferencedSchema Name="Base" MajorVersion="1" />
  <Type Name="Person" MajorVersion="1" MinorVersion="0"
    ExtendsType="Principal" ExtendsVersion="1">
    <Field Name="Birthdate" Type="the storage platformTypes.datetime"
      Nullable="true" MultiValued="false" />
    <Field Name="Gender" Type="the storage platformTypes.nvarchar(16)"
      Nullable="true" MultiValued="false" />
    <Field Name="PersonalNames" Type="FullName" TypeMajorVersion="1"
      Nullable="true" MultiValued="true" />
    <Field Name="PersonalEmailAddresses" Type="EAddress"
      TypeMajorVersion="1" Nullable="true" MultiValued="true" />
    <Field Name="PersonalPostalCodes" Type="PostalAddress"
      TypeMajorVersion="1" Nullable="true" MultiValued="true" />
    <Check>expression</Check>
  </Type>
  ...
  ...
</Schema>
```

以上 XML 中的 Check 标签指定了 Person 类型上的约束。可以有一个以上 Check 标签。以上约束一般是在存储中检查的。为指定约束也可由应用程序显式地检查，以上 XML 被修改为：

```
<Schema Name="Contacts" MajorVersion="1" MinorVersion="8">
  <ReferencedSchema Name="Base" MajorVersion="1" />
  <Type Name="Person" ...>
    <Field Name="Birthdate" Type="the storage platformTypes.datetime"
      Nullable="true" MultiValued="false" />
    ...
    <Check InApplication="true">expression</Check>
  </Type>
  ...
  ...
</Schema>
```

注意，<Check>元素上的新的“InApplication”属性被设为真。这导致存储平台 API 通过 Person 类上称为 Validate()的实例方法在 API 中将约束置于表面。应用程序可在对象上调用该方法，以确保数据是有效的，并防止对服务器的可能的无用往返。这返回指示确认的结果的布尔值。注意，值约束仍在服务器处应用，而不论客户机是否调用了<object>.Validate()方法。以下是如何使用 Validate 的一个示例：

```
ItemContext ctx = ItemContext.Open();

//在用户的 My Contacts 文件夹中创建联系人。
Folder f = UserDataFolder.FindMyPersonalContactsFolder(ctx);
Person p = new Person(f);

//设置人的生日
p.Birthdate = new DateTime(1959, 6, 9);

//添加被归类为人名的名字
FullName name = new FullName(FullName.Category.PrimaryName);
name.GivenName = "Joe";
name.Surname = "Smith";
p.PersonalNames.Add(name);

//确认 Person 对象
if(p.Validate() == false)
{
    //数据不表示有效的人
}

//保存改变
p.Update();

ctx.Close();
```

存在到存储平台存储的多条访问路径—存储平台 API、ADO.NET、ODBC、OLEDB 和 ADO。这引发了权威约束检查的问题—即，如何确保从例如 ODBC 写出的数据与从存储平台 API 写出的数据通过相同的数据完整性约束。由于所有的约束是在存储处检查的，因此约束现在是权威的。不论用于到达存储的 API 路径如何，对存储的所有写都通过存储处的约束检查来过滤。

13. 共享

存储平台中的共享是以下形式：

\<DNS 名>\<上下文服务>

其中<DNS 名>是机器的 DNS 名，<上下文服务>是该机器上的卷中的包含文件夹、虚拟文件夹或项目。例如，假定机器“Johns/Desktop”具有名为 Johns_Information

的卷，且在该卷中存在名为 Contacts_Categories 的文件夹；该文件夹包含名为 Work 的文件夹，后者具有 John 的工作联系人：

\Johns/Desktop\Johns_Information\$\Contacts_Categories\Work
 这可以被共享为“WorkContacts”。采用该共享的定义，\\Johns/Desktop\WorkContacts\JaneSmith 是有效的存储平台名，且标识了 Person 项目 JaneSmith。

a) 表示共享

共享项目类型具有以下属性：共享名以及共享目标（这可以是非持有链接）。例如，上述共享的名称是 WorkContacts，且目标是卷 Johns_Information 上的 Contacts_Categories\Work。以下是用于 Share 类型的模式片段：

```
<Schema
  xmlns="http://schemas.microsoft.com/winf/2002/11/18/schema"
  Name="Share" MajorVersion="1" MinorVersion="0">

  <ReferencedSchema Name="Base" MajorVersion="1"/>
  <ReferencedSchema Name="the storage platformTypes" MajorVersion="1"/>

  <Type Name="Share" MajorVersion="1" MinorVersion="0"
    ExtendsType="Base.Item" ExtendsVersion="1">
    <Field Name="Name" Type="the storage platformTypes.nvarchar(512)"
      TypeMajorVersion="1"/>
    <Field Name="Target" Type="Base.RelationshipData" TypeMajorVersion="1"/>
  </Type>

</Schema>
```

b) 管理共享

由于共享是一个项目，因此共享可如同其它项目那样管理。共享可被创建、删除和修改。共享也以与其它存储平台项目一样的方式来保护。

c) 访问共享

应用程序通过在 ItemContext.Open() 方法调用中将共享名（例如，\\Johns/Desktop\WorkContacts）传递到存储平台 API 来访问远程存储平台共享。ItemContext.Open 返回 ItemContext 对象实例。存储平台 API 然后与本地存储平台服务交谈（可以回想访问远程存储平台是通过本地存储平台完成的）。本地存储平台服务进而与具有给定共享名（例如，WorkContacts）的远程存储平台服务（例如，在机器 Johns/Desktop）交谈。远程存储平台服务然后将 WorkContacts 转换成

Contacts_Categories\Work 并打开它。在那之后，与其它范围一样执行查询和其它操作。

d) 可发现性

在一个实施例中，应用程序可以用以下方式发现给定的<DNS 名>上可用的共享。依照第一种方式，存储平台 API 接受 DNS 名（例如，Johns_Desktop），作为 ItemContext.Open()方法中的范围参数。存储平台 API 然后用该 DNS 名作为连接串的一部分连接到存储平台存储。采用该连接，应用程序唯一能做的事情是调用 ItemContext.FindAll(typeof(Share))。存储平台服务然后对所有附加的卷上的所有共享执行并操作，并返回共享集合。依照第二种方式，在本地机器上，管理员可通过 FindAll(typeof(Share)) 发现特定卷上的共享，或通过 FindAll(typeof(Share), "Target(ShareDestination).Id = folderId")来发现特定的文件夹。

14. Find 的语义

Find*方法（无论它们是在 ItemContext 对象还是在个别的项目上调用）一般应用于给定上下文中的项目（包括嵌入的项目）。嵌套元素没有 Find—它们不能独立于其包含项目来搜索。这与存储平台数据模型所期望的语义相一致，其中嵌套元素从包含项目导出其“身份”。为使这一概念更清楚，以下是有效和无效查找操作的示例：

a) 示出系统中具有区号 206 的所有电话号码？

无效，因为该查找是在电话号码（元素）上完成的，而没有引用项目。

b) 示出所有 Person 内具有区号 206 的所有电话号码？

无效，即使引用了 Person (=项目)，搜索准则也不涉及该项目。

c) 示出 Murali (=单个人) 的具有区号 206 的所有电话号码？

有效，因为存在项目（名为“Murali”的 Person）上的搜索准则。

该规则的例外是对于直接或间接从 Base.Relationship 类型导出的嵌套元素。这些类型可通过关系类个别地查询。这一查询可被支持，因为存储平台实现采用了“主链表”来储存 Relationship 元素而非将它们嵌入在项目 UDT 中。

15. 存储平台 Contacts API

本节给出了存储平台 Contacts API 的综述。Contacts API 背后的模式在图 21A 和 21B 中示出。

a) System.Storage.Contact 的综述

存储平台 API 包括用于处理联系人模式中的项目和元素的名字空间。该名字空间被称为 System.Storage.Contact。

该模式具有例如以下类：

- 项目：UserDataFolder、User、Person、ADService、Service、Group、Organization、Principal、Location
- 元素：Profile、PostalAddress、EmailAddress、TelephoneNumber、RealTimeAddress、EmailAddress、FullName、BasicPresence、GroupMembership、RoleOccupancy

b) 域行为

以下是联系人模式的域行为的列表。当从足够高的级别来查看时，域行为落入良好组织的类别中：

- 静态助手，例如创建新个人联系人的 Person.CreatePersonalContact();
- 实例助手，例如将用户（User 类的实例）登录到被标记为自动登录的所有概况的 user.AutoLoginToAllProfiles();
- 类别 GUID，例如 Category.Home、Category.Work 等；
- 导出属性，例如 emailAddress.Address()—返回组合给定 emailAddress（EmailAddress 类的实例）的用户名和域字段的串；以及
- 导出集合，例如 person.PersonalEmailAddress—给定 Person 类的实例，获得其个人电子邮件地址。

下表对 Contacts 中具有域行为的每一类给出了这些方法和它们所属的类别的列表。

BasicPresence	类别 URI	UnknownCategoryURI 、 OfflineCategoryURI 、 BusyCategoryURI 、 AwayCategoryURI 、 OnlineCategoryURI
---------------	--------	-------------------------------------------------------------------------------------------------------

	静态助手	ConvertPresenceStateToString—作为本地化串的格式存在（需要添加实际的本地化需求；当前仅添加了友好的英语串）。
Category	类别 GUID	Home、Work、Primary、Secondary、Cell、Fax、Pager
EmailAddress	导出属性	Address—组合用户名和域
	静态助手	IsValidEmailAddress
Folder	导出属性	GetChildItemCollection—使得项目集合基于 FolderMembership 的目标
	静态助手	GetKnownFolder—获得公知文件夹的专用查询
		AddToPersonalContacts—将项目添加到公知的个人联系人文件夹
Items	静态助手	GetItemFromID—进行基于 ID 的查询
Relationship	实例助手	BindToTarget—返回目标的项目
Person	导出集合	PersonalRealtimeAddress、PersonalEmailAddresses、PersonalTelephoneNumber
	导出属性	OnlineStatus、OnlineStatusIconSource、PrimaryEmailAddress、PrimarySecurityID
	静态助手	CreatePersonalContact、CreateTemporaryContact—在公知文件夹中创建新的个人
		GetCurrentUser—获得当前登录的用户的 Person
SecurityID	导出属性	UserName、DomainName、DomainUserName
TelephoneNumber	实例助手	SetFromUserInputString—将电话号码串剖析成各部分
	静态助手	ParseNumber—将电话号码串剖析成各部分
User	实例助手	AutoLoginToAllProfiles—登录到被标记为自动登录的所有概况

16. 存储平台 File API

本节依照本发明的一个实施例给出了存储平台 File API 的综述。

a) 介绍

(1) 在存储平台中反映 NTFS 卷

存储平台提供了在现有 NTFS 卷中的内容上进行索引的方法。这是通过从 NTFS 中的每一文件流或目录中提取（“升级”）属性，并将这些属性作为项目储存在存储平台中来实现的。

存储平台的文件模式定义了两种项目类型—File（文件）和 Directory（目录），来储存升级的文件系统实体。Directory 类型是 Folder 类型的子类型；它是包含其它 Directory 项目或 File 项目的包含文件夹。

Directory 项目可包含 Directory 和 File 项目；它不能包含任何其它类型的项目。至于所考虑的存储平台，Directory 和 File 项目从任何数据访问 API 都是只读的。文件系统升级管理器（FSPM）服务异步地将改变的属性升级到存储平台中。File 和 Directory 项目的属性可由 Win32 API 来改变。存储平台 API 可用于读取这些项目的任何属性，包括与 File 项目相关联的流。

(2) 在存储平台名字空间中创建文件和目录

当 NTFS 卷被升级到存储平台卷时，其中的所有文件和目录都在该卷的特定部分中。该区域从存储平台的观点来看是只读的；FSPM 可创建新目录和文件，和 / 或改变现有项目的属性。

该卷的名字空间的剩余部分可包含存储平台项目类型的普通全范围—Principal（主体）、Organization（组织）、Document（文档）、Folder（文件夹）等。存储平台也允许在存储平台名字空间的任何部分中创建文件和目录。这些“本机”文件和目录在 NTFS 文件系统中没有相应物；它们完全储存在存储平台中。此外，对属性的改变是立即可见的。

然而，编程模型保持相同；就考虑到存储平台数据访问 API 而言，它们仍是只读的。“本机”文件和目录必须使用 Win32 API 来更新。这简化了开发者的思维模型，该思维模型为：

1. 任何存储平台项目类型可以在名字空间中的任何地方处创建（当然，除非被许可所阻止）；
2. 任何存储平台项目类型可以使用存储平台 API 来读取；
3. 所有存储平台项目类型可使用具有 File 和 Directory 的例外的存储平台 API 来写入
4. 为写到 File 和 Directory 项目而不管它们在名字空间中的何处，使用 Win32

API; 以及

5. 在“升级的”名字空间中对 File/Directory 项目的改变可能不直接出现在存储平台中；在“非升级”名字空间中，改变直接在存储平台中反映。

b) 文件模式

图 25 示出了 File API 所基于的模式。

c) System.Storage.Files 的综述

存储平台 API 包括用于处理文件对象的名字空间。该名字空间称为 System.Storage.Files。System.Storage.Files 中的类的数据成员直接反映了储存在存储平台存储中的信息；该信息从文件系统对象“升级”，或可以使用 Win32 API 本机创建。System.Storage.Files 名字空间具有两个类：FileItem（文件项目）和 DirectoryItem（目录项目）。这些类的成员及其方法可通过查看图 25 中的模式图来容易地猜想。FileItem 和 DirectoryItem 从存储平台 API 是只读的。为修改它们，必须使用 Win32 API 或 System.IO 中的类。

d) 代码示例

在本节中，提供了三个代码示例，示出了 System.Storage.Files 中的类的使用。

(1) 打开文件并向其写入

本示例示出了如何完成“传统”文件操纵。

```
ItemContext ctx = ItemContext.Open();
FileItem f = FileItem.FindByPath(ctx, @"\My Documents\billg.ppt");
//处理文件属性的示例—确保文件不是只读的
if(!f.IsReadOnly)
{
    FileStream fs = f.OpenWrite();
    //读、写、关闭文件流 fs
}
ctx.Close();
```

第 3 行使用了 FindByPath（按照路径查找）方法来打开文件。第 7 行示出了对升级的属性 IsReadOnly 的使用，来检查文件是否可写。如果是，则在第 9 行使用 FileItem 对象上的 OpenWrite() 方法来获得文件流。

(2) 使用查询

由于存储平台存储持有从文件系统升级的属性，因此可能容易地在文件上完成丰富的查询。在本示例中，列出了在最后三天修改的所有文件：

```
//列出在最后三天中修改的所有文件
```

```
FindResult result = FileItem.FindAll(
    ctx,
    "Modified >= '{0}'",
    DateTime.Now.AddDays(-3));

foreach(result 中的 FileItem file)
{
    ...
}
```

以下是使用查询的另一示例—该示例找出某一类型(=extension)的所有可写文件：

```
//找出特定目录中所有可写.cs 文件
//等效于：dir c:\win\src\api\*.cs /a-r-d
```

```
DirectoryItem dir =
    DirectoryItem.FindByPath(ctx, @"c:\win\src\api");
FindResult = dir.GetFiles(
    "Extension='cs' and IsReadOnly = false");

foreach(result 中的 File file)
{
    ...
}
```

e) 域行为

在一个实施例中，除标准属性和方法之外，文件类也具有域行为（手工编码的属性和方法）。这些行为一般基于对应的 System.IO 类中的方法。

J. 总结

如上所述，本发明针对一种用于组织、搜索和共享数据的存储平台。本发明的存储平台在现有文件系统和数据库系统之外扩展并拓宽了数据存储的概念，并被设计成储存所有类型的数据，包括结构化的、非结构化的或半结构化的数据，诸如关系型（表式）数据、XML 以及称为项目的新形式的数据。通过其公共存储基础和模式化的数据，本发明的存储平台对消费者、知识工作者和企业允许更有效的应用程序开发。它提供了一种丰富且可扩展的应用程序编程接口，该接口不仅使得其

数据模型中的固有能力可用，还包含并扩展了现有文件系统和数据库访问方法。可以理解，可以对上述实施例做出改变而不脱离其宽泛的发明性概念。因此，本发明不限于所揭示的具体实施例，而是旨在覆盖落入由所附权利要求书定义的本发明的精神和范围之内的所有修改。

如从上文中可以清楚的，本发明的各种系统、方法和方面的全部或部分可以用程序代码（即，指令）的形式来实施。该程序代码可被储存在计算机可读介质上，诸如磁、电、光存储介质，包括但不限于，软盘、CD-ROM、CD-RW、DVD-ROM、DVD-RAM、磁带、闪存、硬盘驱动器、或任何其它机器可读存储介质，其中，当程序代码被加载到诸如计算机或服务器等机器并由其执行时，该机器成为用于实现本发明的装置。本发明也可用通过某一传输介质，诸如通过电线或电缆、通过光纤、通过包括因特网或内联网的网络、或通过任何其它形式的传输来传输的程序代码的形式来实施，其中，当程序代码由诸如计算机等机器接受并加载到其中由其执行时，该机器成为用于实现本发明的装置。当在通用处理器上实现时，程序代码与处理器相结合以提供类似于特定逻辑电路操作的独特装置。

附录 A

```

namespace System.Storage
{
    abstract class ItemContext : IDisposable, IServiceProvider
    {

        ItemContext 创建和管理成员

        //应用程序无法直接创建 ItemContext 对象，也无法从 ItemContext 中导出类。
        internal itemContext();

        //创建可用于搜索指定路径，或者如果未指定路径，
        //则创建可用于搜索本地计算机上的默认存储的 ItemContext。
        public static ItemContext Open();
        public static ItemContext Open(string path);
        public static ItemContext Open(params string[] paths);

        //返回创建 ItemContext 时指定的路径。
        public string[] GetOpenPaths();

        //创建该 ItemContext 的副本。该副本将具有独立的事务、高速缓存和更新状态。高速缓存
        //初始为空。期望使用克隆的 ItemContext 将比使用同一项目域打开新的 ItemContext 更有效。
        public ItemContext Clone();

        //关闭 ItemContext。在关闭之后使用 ItemContext 的任何尝试将导致 ObjectDisposedException。
        public void Close();
        void IDisposable.Dispose();

        //如果当打开 ItemContext 指定的任何域解析到远程计算机，则为真。
        public bool IsRemote{get;}

        //返回可提供所请求的服务类型的对象。如果不能提供所请求的服务，则返回空。
        //对 IServiceProvider 模式的使用允许非正常使用且可能导致开发者混淆的 API 被排除在
        //ItemContext 类之外。ItemContext 可提供以下种类的服务：ItemSerialization、IStoreObjectCache
        public object GetService(Type serviceType);

        更新相关成员

        //保存由所有修改的对象以及传递到 MarkForCreate 或 MarkForDelete 的所有对象表示的改变。
        //如果检测到更新冲突，则可抛出 UpdateCollisionException。
        public void Update();

        //保存由所指定的对象表示的改变。对象必须已被修改或已被传递到 MarkForCreate 或
        //MarkForDelete，否则抛出 ArgumentException。如果检测到更新冲突，
        //可抛出 UpdateCollisionException。
        public void Update(object objectToUpdate);
        public void Update(IEnumerable objectsToUpdate);

        //从存储中刷新所指定对象的内容。如果对象已被修改，则覆盖改变，且对象不再被认为是已
        //改变的。如果指定了除项目、项目扩展或关系对象之外的任何内容，则抛出 ArgumentException。
        public void Refresh(object objectToRefresh);
        public void Refresh(IEnumerable objectsToRefresh);

        //当更新检测到检索修改的对象和试图保存它之间在存储中改变了数据时引发。如果没有注册任
        //何事件处理程序，则更新抛出异常。如果注册了事件处理程序，则它可抛出异常来中止更新。
        //包装修改的对象以覆盖存储中的数据，或合并在存储中和对象中做出的改变。
        public event ChangeCollisionEventHandler UpdateCollision;
    }
}

```

```

//在更新处理的各点处引发，以提供更新进展信息。
public event UpdateProgressEventHandler UpdateProgress;

//Update 的异步版本
public IAsyncResult BeginUpdate(I AsyncCallback callback, object state);
public IAsyncResult BeginUpdate(object objectToUpdate,
    I AsyncCallback callback,
    object state);
public IAsyncResult BeginUpdate(IEnumerable objectsToUpdate,
    I AsyncCallback callback,
    object state);
public void EndUpdate(IAsyncResult result);

//Refresh 的异步版本
public IAsyncResult BeginRefresh(object objectToRefresh,
    I AsyncCallback callback,
    object state);
public IAsyncResult BeginRefresh(IEnumerable objectsToRefresh,
    I AsyncCallback callback,
    object state);
public void EndRefresh(IAsyncResult result);

```

事务相关成员

//以指定的隔离级别启动事务。默认的隔离级别是 ReadCommitted。在所有情况下，
 //启动分布式事务，因为它可能必须包含改变流类型化项目属性。

```

public Transaction BeginTransaction();
public Transaction BeginTransaction(System.Data.IsolationLevel isolationLevel);

```

搜索相关成员

//创建在该项目上下文中搜索指定类型对象的 ItemSearcher。如果指定了除项目、关系或
 //项目扩展之外的类型，则抛出 ArgumentException。

```
public ItemSearcher GetSearcher(Type type);
```

//给定其 ID 找出项目。

```
public Item FindItemById(ItemId itemId);
```

//给定其路径找出项目。路径可以是绝对或相对的。如果是相对的，则如果当打开 ItemContext
 //时指定了多个项目域，则抛出 NotSupportedException。如果不存在这样的项目，则返回空。
 //创建到域的\\machine\\share 部分的连接以检索项目。该项目将与该域相关联。

```
public Item FindItemByPath(string path);
```

//给定其路径找出项目。路径相对于指定的项目域。创建到指定域的连接以检索项目。该项目将
 //与该域相关联。如果不存在这样的项目，则返回空。

```
public Item FindItemByPath(string domain, string path);
```

//给定其路径找出一组项目。该路径相对于当打开 ItemContext 时指定的项目域。如果不存在
 //这样的项目，则返回空结果。

```
public FindResult FindAllItemsByPath(string path);
```

//给定其 ID 找出关系。

```
public Relationship FindRelationshipById(ItemId itemID,
    RelationshipId relationshipId);
```

//给定其 ID 找出项目扩展。

```
public ItemExtension FindItemExtensionById(ItemId itemId,
    ItemExtensionId itemExtensionId);
```

```

//找出可任意地满足给定过滤器的指定类型的所有项目、关系或项目扩展。如果指定了
//这些类型之一以外的类型，则抛出 ArgumentException。
public FindResult FindAll(Type type);
public FindResult FindAll(Type type, string filter);

//找出满足给定过滤器的指定类型的任何项目、关系或项目扩展。如果指定了这些类型之一
//以外的类型，则抛出 ArgumentException。如果没有找到这样的对象，则返回空。
public object FindOne(Type type, string filter);

//找出满足给定过滤器的指定类型的项目、关系或项目扩展。如果指定了这些类型之一以外的
//类型，则抛出 ArgumentException。如果没有找到这样的对象，则抛出
//ObjectNotFoundException。如果找到一个以上对象，则抛出 MultipleObjectsFoundException。
public object FindOnly(Type type, string filter);

//如果存在满足给定过滤器的指定类型的项目、关系或项目扩展，则返回真。如果指定了这些
//类型之一以外的类型，则抛出 ArgumentException。
public bool Exists(Type type, string filter);

//指定由涉及对象的搜索返回的对象如何标识由 ItemContext 维护的映射。
public SearchCollisionMode SearchCollisionMode{get; set;}

//当为 ResultMapping 指定了 PreserveModifiedObject 时引发。该事件允许应用程序选择性地用
//以该搜索检索到的数据来更新修改的对象。
public event ChangeCollisionEventHandler SearchCollision;

//将来自另一 ItemContext 的对象结合到该项目上下文中。如果表示同一项目、关系或项目扩展的
//对象并未在该 ItemContext 的身份映射中存在，则创建该对象的克隆并将其添加到该映射。如果
//对象的确存在，则用指定对象的状态和内容以与 SearchCollisionMode 一致的方式更新该对象。
public Item IncorporateItem(Item item);
public Relationship IncorporateRelationship(Relationship relationship);
public ItemExtension IncorporateItemExtension(ItemExtension itemExtension);

}

//用于 ItemContext.UpdateCollision 和 ItemSearcher.SearchCollision 事件的处理程序。
public delegate void ChangeCollisionEventHandler(object source,
                                                    ChangeCollisionEventArgs args);

//用于 ChangeCollisionEventHandler 委托的自变量
public class ChangeCollisionEventArgs : EventArgs
{
    //修改的项目、项目扩展或关系对象。
    public object ModifiedObject{get;}

    //来自存储的属性
    public IDictionary StoredProperties{get;}
}

//用于 ItemContext.UpdateProgress 的处理程序
public delegate void UpdateProgressEventHandler(ItemContext itemContext,
                                                UpdateProgressEventArgs args);

//用于 UpdateProgressEventHandler 委托的自变量
public class ChangeCollisionEventArgs : EventArgs
{
    //当前更新操作
    public UpdateOperation CurrentOperation{get;}

    //当前更新的对象
    public object CurrentObject{get;}
}

```

```
}

//指定由涉及对象的搜索返回的对象如何标识由 ItemContext 维护的映射。
public enum SearchCollisionMode
{
    //指示应当创建和返回新对象。忽略身份映射中表示同一项目、项目扩展或关系的对象。如果
    //指定了该选项，则不引发 SearchCollision 事件。
    DoNotMapSearchResults,

    //指示应返回来自身份映射的对象。如果应用程序修改了对象的内容，则保存修改的对象的内容。
    //如果未修改对象，则用由搜索返回的数据更新其内容。应用程序可提供用于 SearchCollision
    //事件的处理程序，并在需要时选择性地更新对象。
    PreserveModifiedObjects,

    //指示应返回来自身份映射的对象。使用由搜索返回的数据更新对象的内容，即使对象已被应用
    //程序修改。如果指定了该选项，则不引发 SearchCollision 事件。
    OverwriteModifiedObjects
}

//当前更新操作
public enum UpdateOperation
{
    //当首次调用 Update 时提供。CurrentObject 将为空。
    OverallUpdateStarting,

    //仅在成功更新后的 Update 返回之前提供。CurrentObject 将为空。
    OverallUpdateCompletedSuccessfully,

    //仅在 Update 抛出异常之前提供。CurrentObject 将是异常对象。
    OverallUpdateCompletedUnsuccessfully,

    //当启动对象的更新时提供。CurrentObject 将引用用于更新的对象。
    ObjectUpdateStarting,

    //当需要新连接时提供。CurrentObject 将是包含标识传递给 ItemContext.Open 或从关系的
    //Location 字段检索的项目域的路径的串。
    OpeningConnection
}
```

附录 B

namespace System.Storage

{

//在项目上下文中执行跨特定类型的搜索。

public class ItemSearcher
{

构造函数

```
public ItemSearcher();  
public ItemSearcher(Type targetType, ItemContext context);  
public ItemSearcher(Type targetType, ItemContext context,  
    params SearchExpression[] filters);
```

属性

//用于标识匹配对象的过滤器。

public SearchExpressionCollection Filters{get;}

//指定将被搜索的域的 ItemContext。

public ItemContext ItemContext{get; set;}

//搜索参数集合。

public ParameterCollection Parameters{get;}

//搜索器将用于操作的类型。对于简单的搜索，这是将返回的对象的类型。

public Type TargetType{get; set;}

搜索方法

//找出满足由 Filters 指定的条件的 TargetType 的对象。如果不存在这样的对象，则返回空的
//FindResult。

```
public FindResult FindAll();  
public FindResult FindAll(FindOptions findOptions);  
public FindResult FindAll(params SortOption[] sortOptions);
```

//找出满足由 Filters 指定的条件的 TargetType 的任何一个对象。如果不存在这样的对象则返回空。

```
public object FindOne();  
public object FindOne(FindOptions findOptions);  
public object FindOne(params SortOption[] sortOptions);
```

//找出满足由 Filters 指定的条件的 TargetType。如果未找到这样的对象则抛出

//ObjectNotFoundException。如果找到一个以上对象，则抛出 MultipleObjectsFoundException。

```
public object FindOnly();  
public object FindOnly(FindOptions findOptions);
```

//确定满足由 Filters 指定的条件的 TargetType 的对象是否存在。

public bool Exists();

//创建可用于更有效地重复执行同一搜索的对象。

```
public PreparedFind PrepareFind();  
public PreparedFind PrepareFind(FindOptions findOptions);  
public PreparedFind PrepareFind(params SortOption[] sortOptions);
```

//检索将由 FindAll() 返回的记录的数目。

public int GetCount();

```
//各种方法的异步版本
public IAsyncResult BeginFindAll(AsyncCallback callback,
                                object state);

public IAsyncResult BeginFindAll(FindOptions findOptions,
                                AsyncCallback callback,
                                object state);

public IAsyncResult BeginFindAll(SortOption[] sortOptions,
                                AsyncCallback callback,
                                object state);

public FindResult EndFindAll(IAsyncResult ar);

public IAsyncResult BeginFindOne(AsyncCallback callback,
                                object state);

public IAsyncResult BeginFindOne(FindOptions findOptions,
                                AsyncCallback callback,
                                object state);

public IAsyncResult BeginFindOne(SortOption[] sortOptions,
                                AsyncCallback callback,
                                object state);

public object EndFindOne(IAsyncResult asyncResult);

public IAsyncResult BeginFindOnly(AsyncCallback callback,
                                object state);

public IAsyncResult BeginFindOnly(FindOptions findOptions,
                                AsyncCallback callback,
                                object state);

public IAsyncResult BeginFindOnly(SortOption[] sortOptions,
                                AsyncCallback callback,
                                object state);

public object EndFindOnly(IAsyncResult asyncResult);

public IAsyncResult BeginGetCount(AsyncCallback callback,
                                object state);

public int EndGetCount(IAsyncResult asyncResult);

public IAsyncResult BeginExists(AsyncCallback callback,
                               object state);

public bool EndExists(IAsyncResult asyncResult);

}

//当执行搜索时使用的选项
public class FindOptions
{
    public FindOptions();

    public FindOptions(params SortOption[] sortOptions);

    //指定是否应当延迟加载可延迟加载的字段。
}
```

```
public bool DelayLoad{get; set;}  
    //返回的匹配的数目。  
    public int MaxResults{get; set;}  
    //排序选项的集合。  
    public SortOptionCollection SortOptions{get; }  
}  
  
//表示参数名和值  
public class Parameter  
{  
    //使用名称和值初始化 Parameter 对象  
    public Parameter(string name, object value);  
  
    //参数名  
    public string Name{get; }  
  
    //参数值  
    public object Value{get; set; }  
}  
  
//参数名/值对的集合  
public class ParameterCollection : ICollection  
{  
    public ParameterCollection();  
  
    public int Count{get; }  
  
    public object this[string name] {get; set; }  
  
    public object SyncRoot{get; }  
  
    public void Add(Parameter parameter);  
    public Parameter Add(string name, object value);  
  
    public bool Contains(Parameter parameter);  
    public bool Contains(string name);  
  
    public void CopyTo(Parameter[] array, int index);  
    void ICollection.CopyTo(Array array, int index);  
  
    IEnumerator IEnumerable.GetEnumerator();  
  
    public void Remove(Parameter parameter);  
    public void Remove(string name);  
}  
  
//表示已为重复执行进行优化的搜索。  
public class PreparedFind  
{  
    public ItemContext ItemContext{get; }  
  
    public ParameterCollection Parameters{get; }  
  
    public FindResult FindAll();
```

```
public object FindOne();

public object FindOnly();

public bool Exists();

}

//指定搜索中使用的排序选项。
public class SortOption
{

    //使用默认值初始化对象。
    public SortOption();

    //使用 SearchExpression、order 初始化 SortOptions 对象。
    public sortOption(SearchExpression searchExpression, SortOrder order);

    //标识将排序的属性的搜索 SearchExpression。
    public SearchExpression Expression {get; set;}

    //指定升序或降序的排序顺序
    public SortOrder Order{get; set; }

}

//排序选项对象的集合
public class SortOptionCollection : IList
{
    public SortOptionCollection();

    public SortOption this[int index] {get; set; }

    public int Add(SortOption value);
    public int Add(SearchExpression expression, SortOrder order);
    int IList.Add(object value);

    public void Clear();

    public bool Contains(SortOption value);
    bool IList.Contains(object value);

    public void CopyTo(SortOption[] array, int index);
    void ICollection.CopyTo(Array array, int index);

    public int Count {get; }

    IEnumerator IEnumerable.GetEnumerator();

    public void Insert(int index, SortOption value);
    void IList.Insert(int index, object value);

    public int indexOf(SortOption value);
    int IList.IndexOf(object value);

    public void Remove(SortOption value);
    void IList.Remove(object value);
    public void RemoveAt(int index);
```

```
public object SyncRoot{get;}  
}  
  
//使用 SortOption 对象指定排序顺序  
public enum SortOrder  
{  
    Ascending,  
    Descending  
}
```

附录 C

```

namespace System.Storage
{
    public abstract class FindResult : IAsyncObjectReader
    {
        public FindResult();

        //将 FindResult 移至结果中的下一位置。
        public bool Read();
        public IAsyncResult BeginRead(AsyncCallback callback, object state);
        public bool EndRead(IAsyncResult asyncResult);

        //当前对象。
        public object Current{get;}

        //返回 FindResult 是否包含任何对象。
        public bool HasResults{get;}

        //返回 FindResult 是否被关闭。
        public bool IsClosed{get;}

        //返回该 FindResult 中的项目的类型。
        public Type ObjectType{get;}

        //关闭 FindResult
        public void Close();
        void IDisposable.Dispose();

        //返回 FindResult 上的枚举器，从当前位置开始。在 FindResult 上前进任何枚举器将前进所有的
        //枚举器以及 FindResult 本身。
        IEnumerator IEnumerable.GetEnumerator();
        public FindResultEnumerator GetEnumerator();

    }

    public abstract class FindResultEnumerator : IEnumerator, IDisposable
    {
        public abstract object Current{get;}
        public abstract bool MoveNext();
        public abstract void Reset();
        public abstract void Close();

        void IDisposable.Dispose();

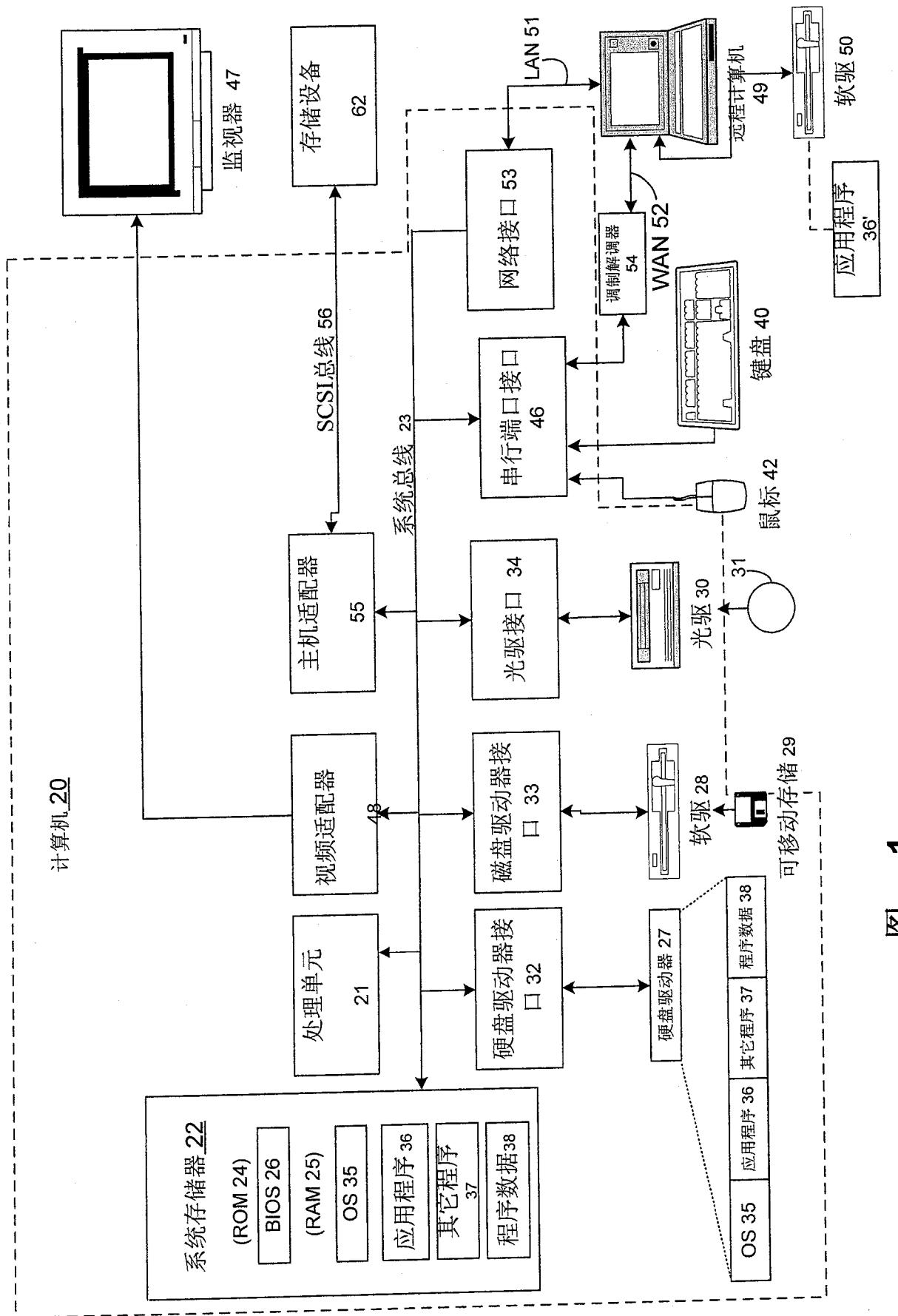
    }

}

namespace System
{
    //用于在对象上迭代的公共接口
    public interface iObjectReader : IEnumerable, IDisposable
    {
        object Current{get;}
}

```

```
bool IsClosed{get;}  
bool HasResults{get;}  
Type objectType{get;}  
  
bool Read();  
void Close();  
  
}  
  
//向 IObjectReader 添加异步方法  
public interface IAsyncObjectReader : IObjectReader  
{  
  
    IAsyncResult BeginRead(AsyncCallback callback, object state);  
    bool EndRead(IAsyncResult result);  
  
}  
}
```



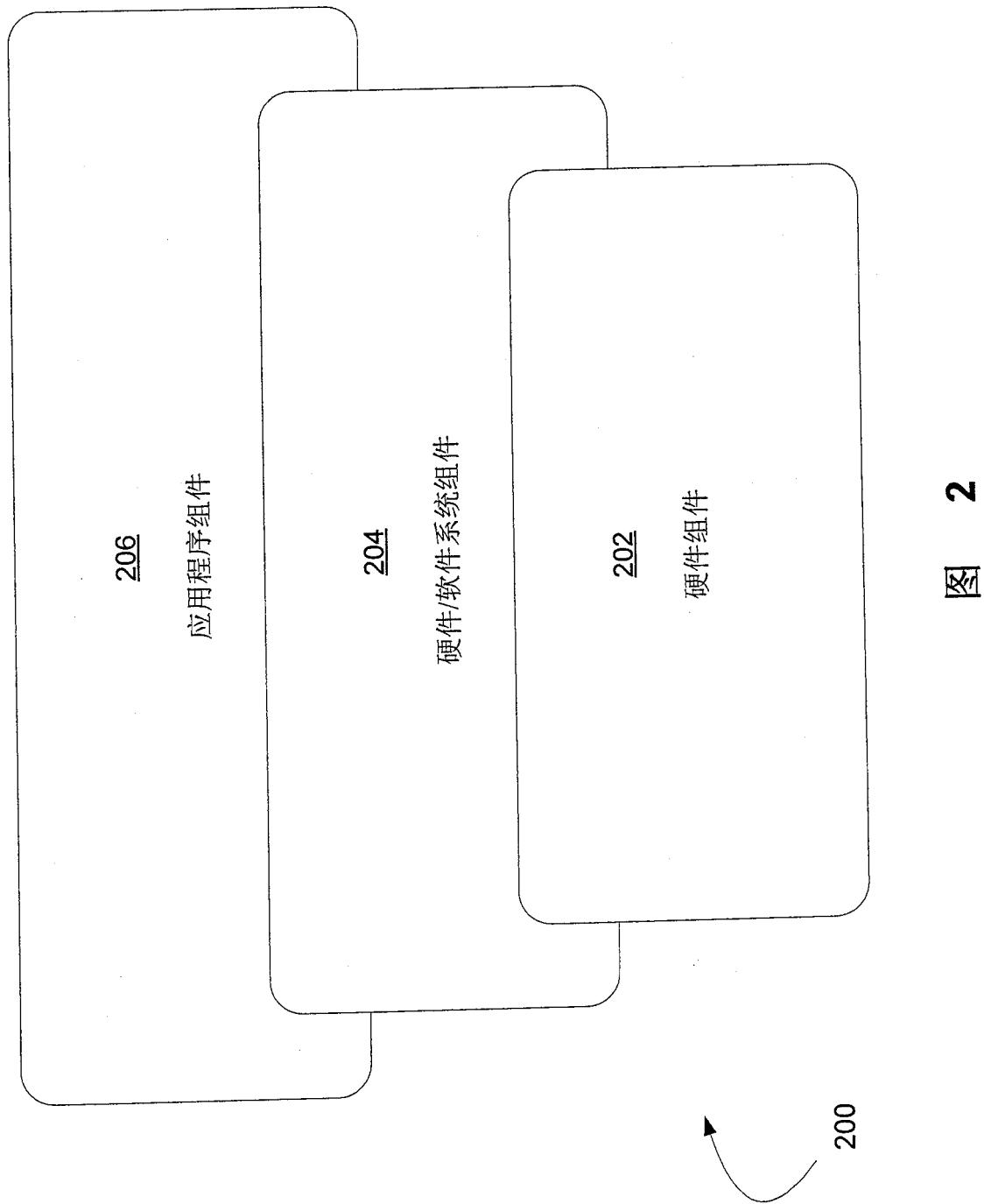


图 2

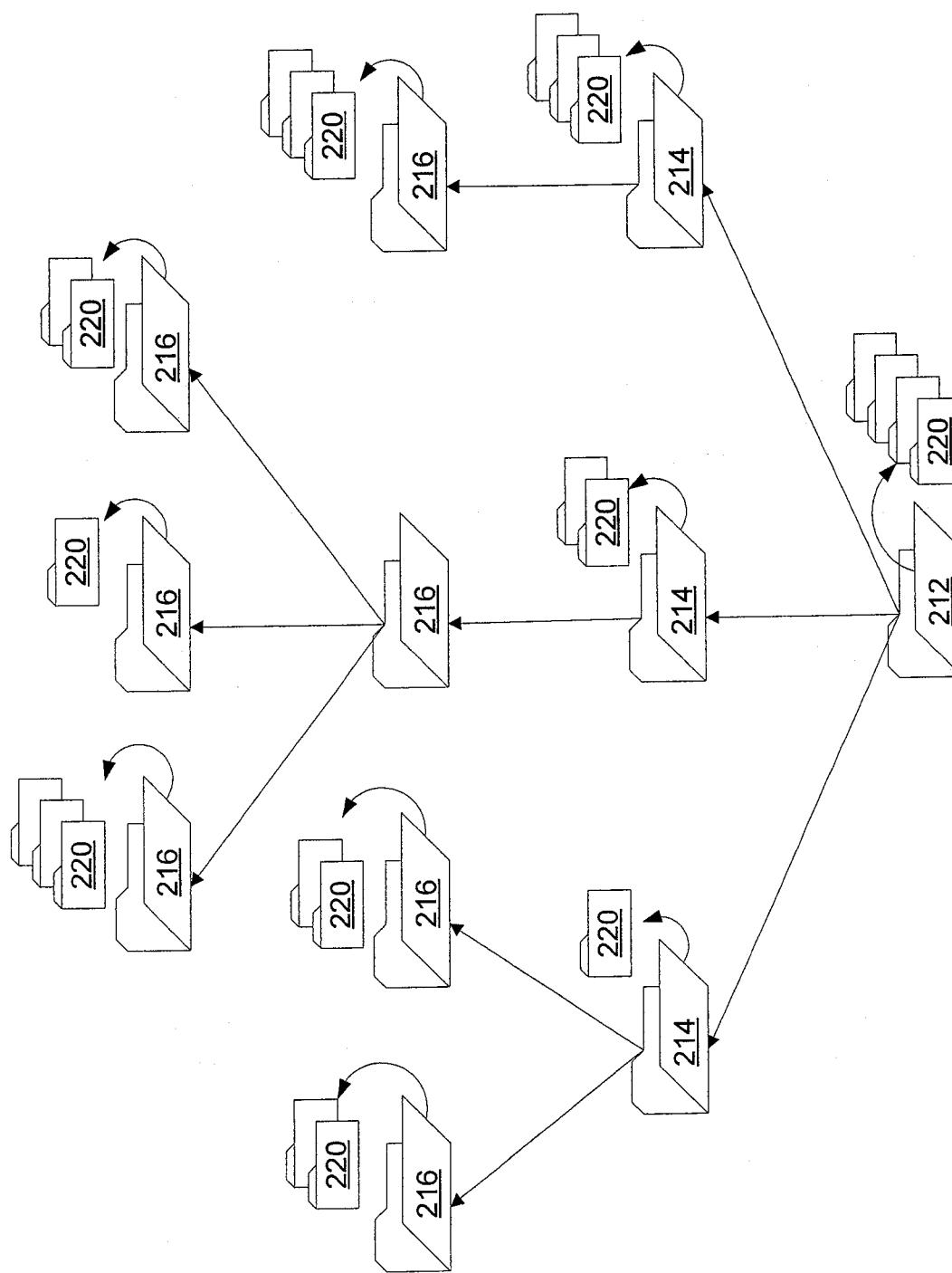


图 2A

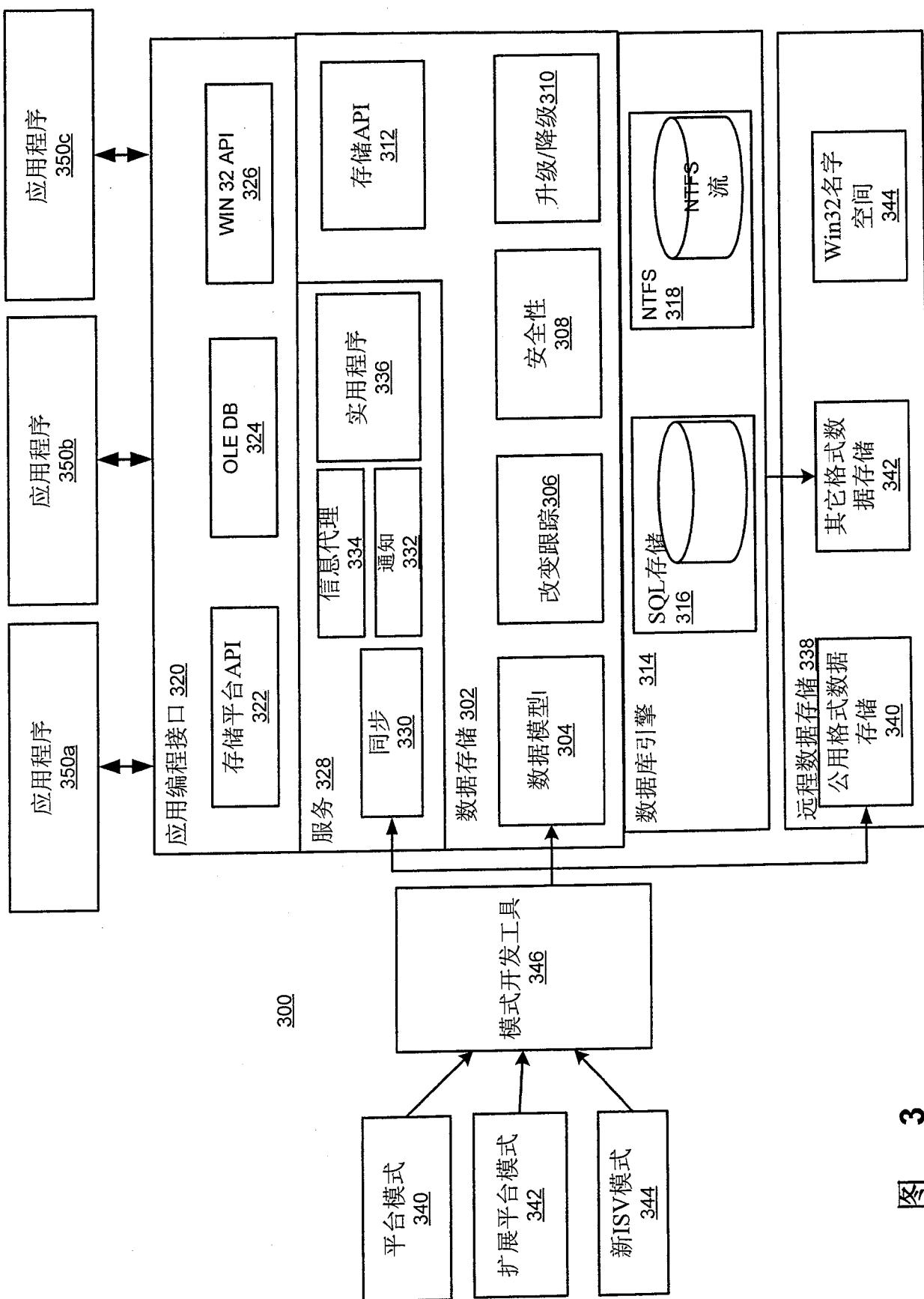


图 3

图

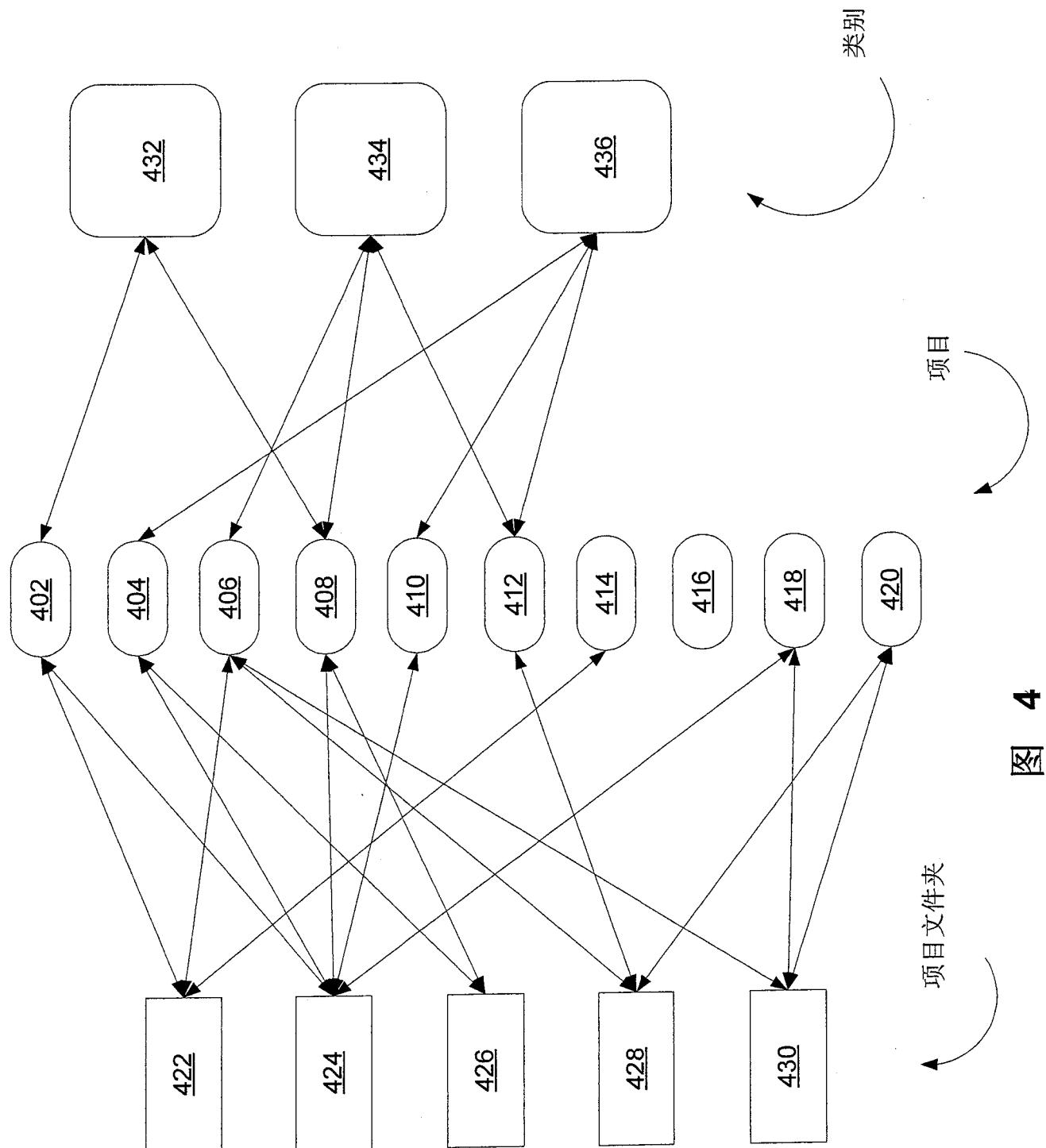
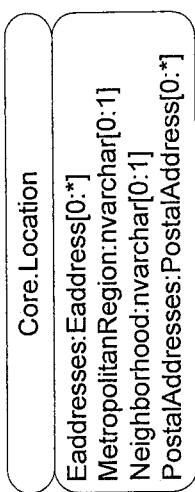
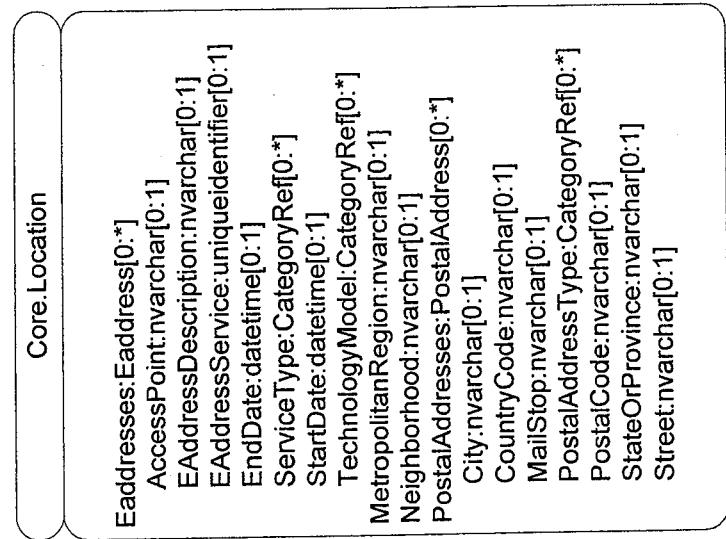
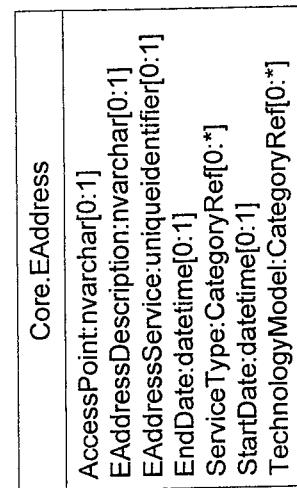


图 4

**图 5A****图 5B****图 5C**

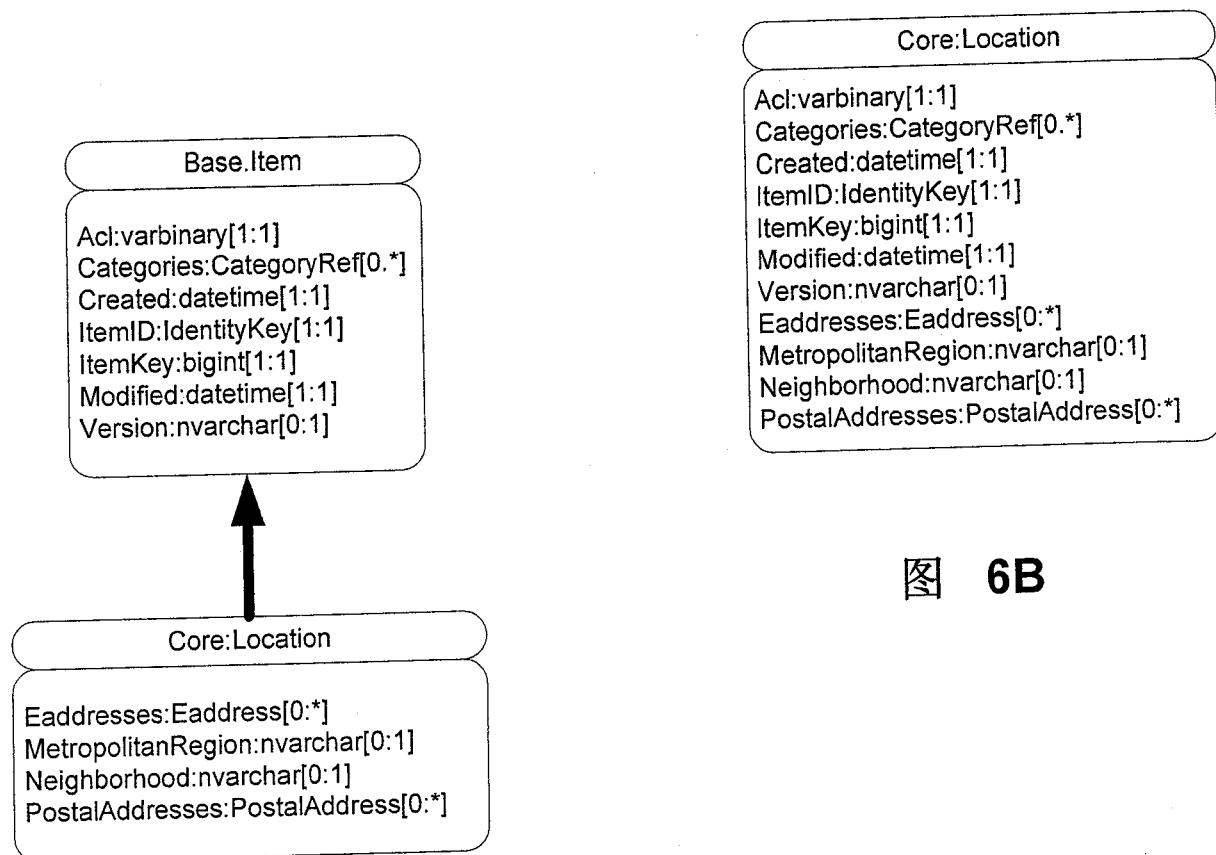


图 6B

图 6A

基础模式项目

基础模式扩展

基础模式属性

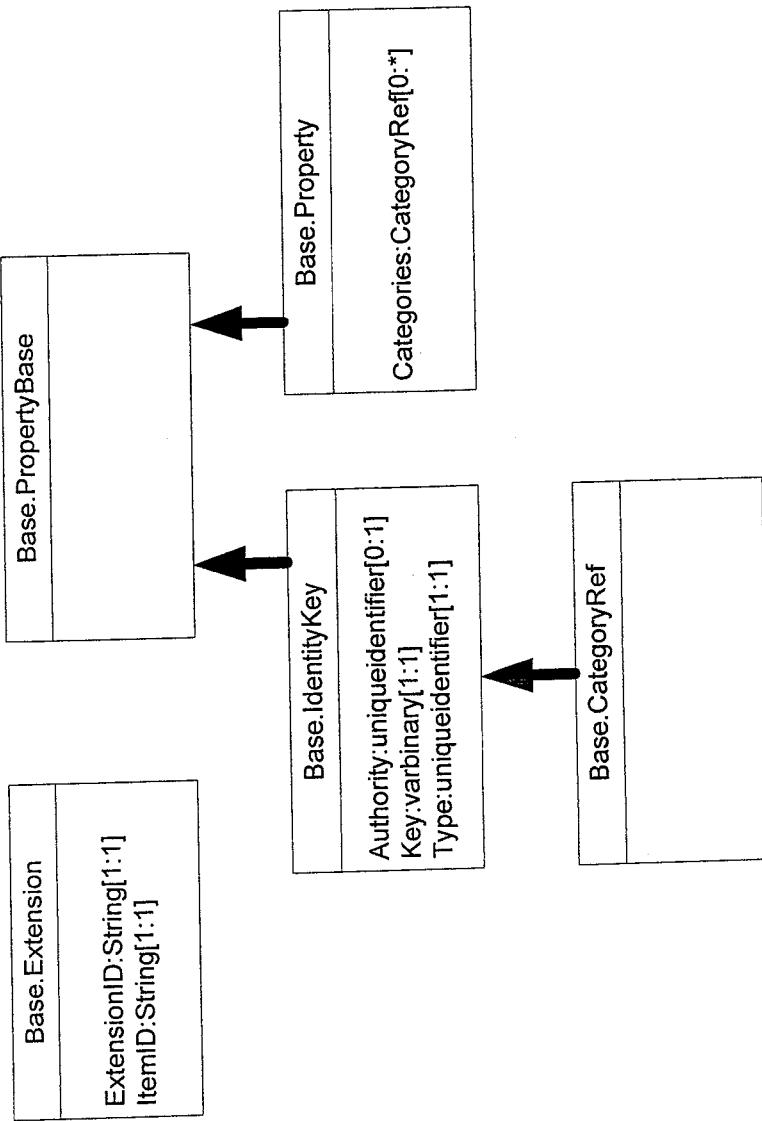
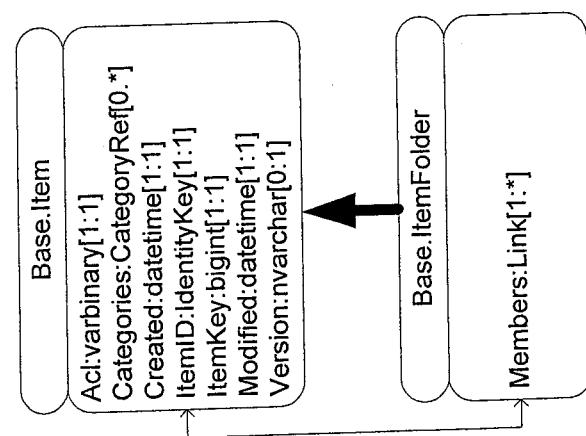
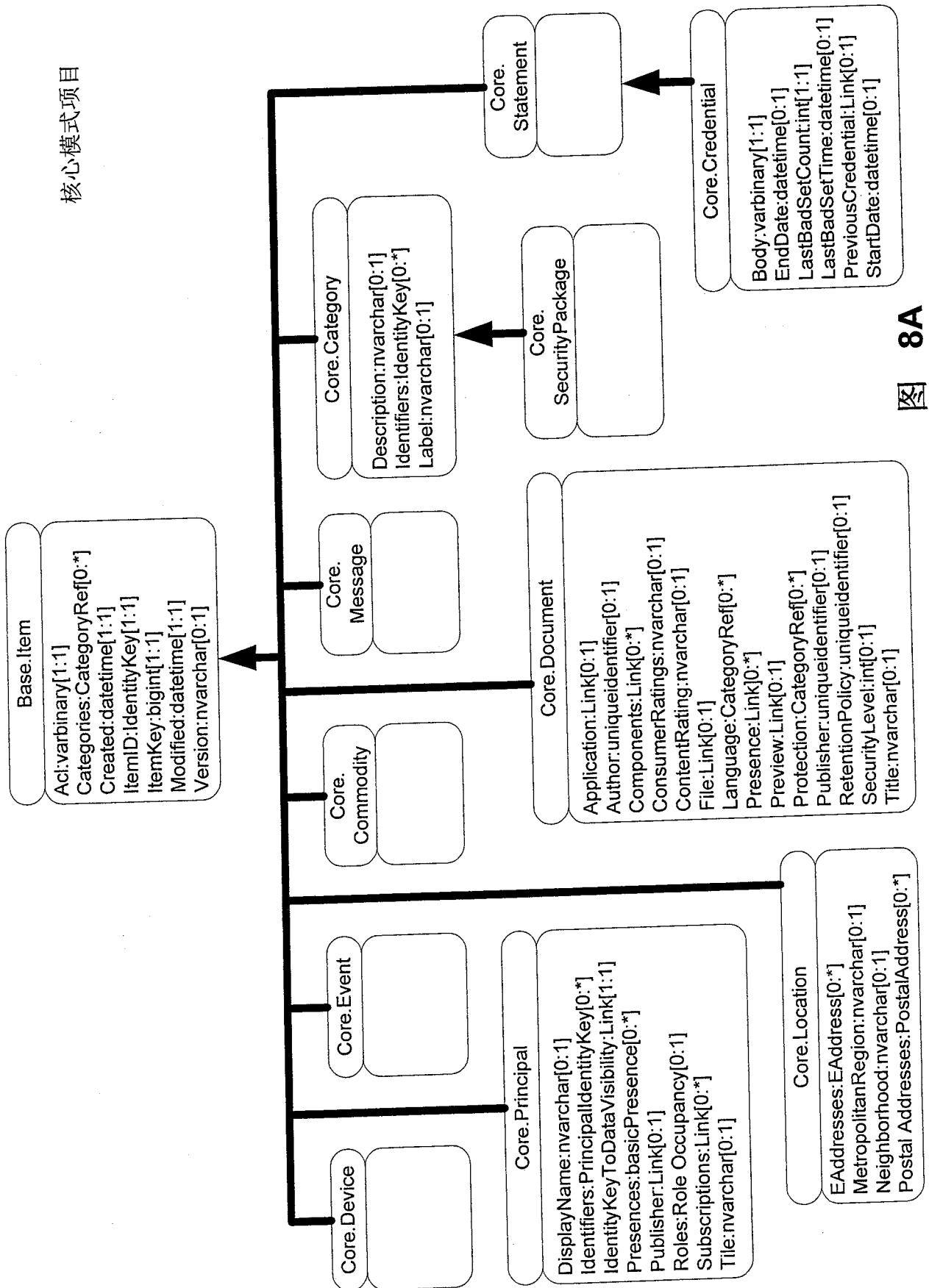


图 7



8A



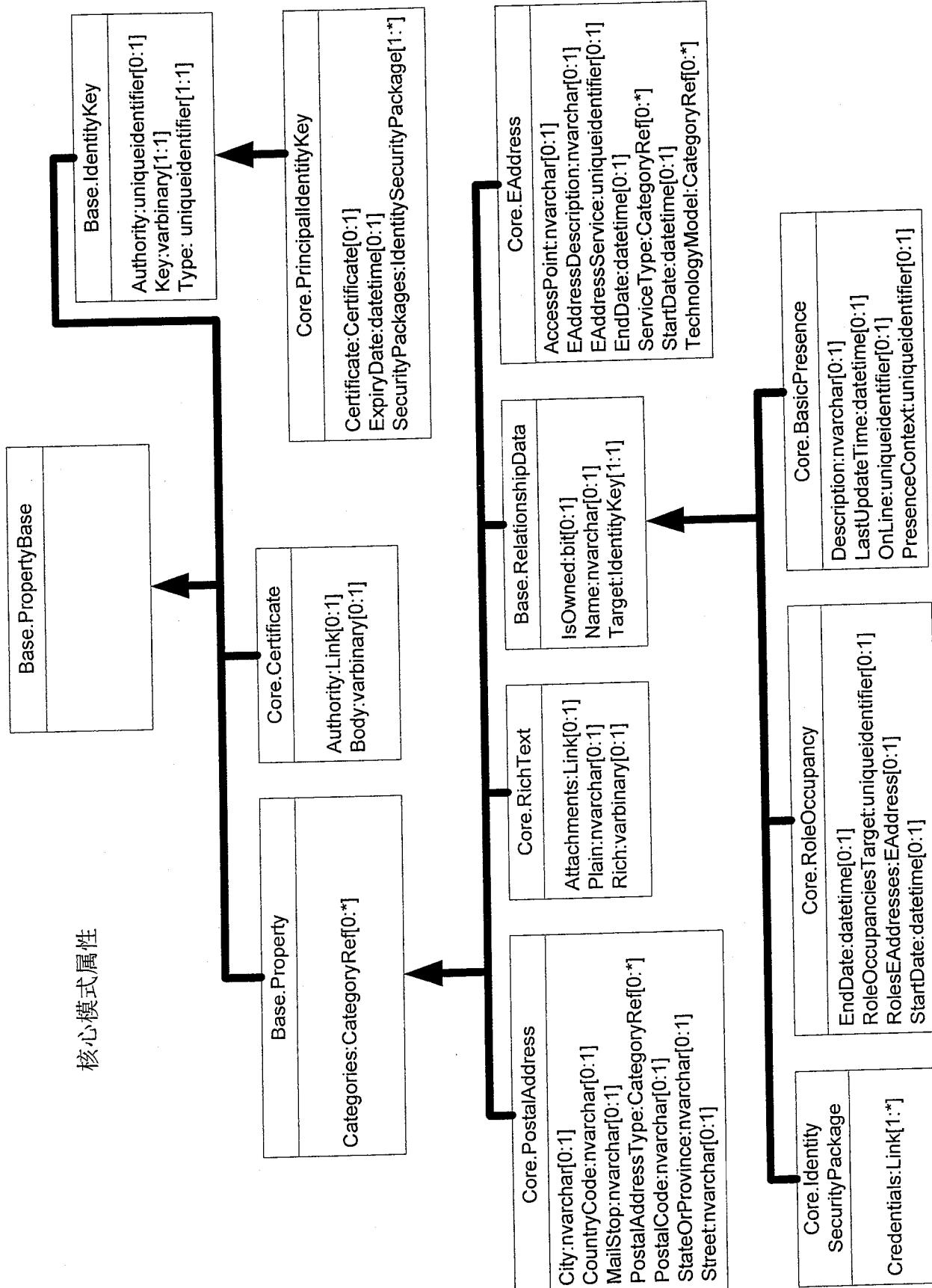


图 8B

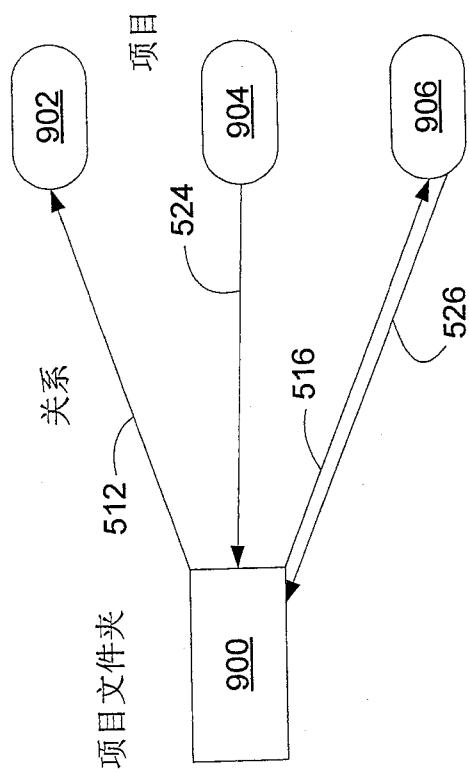


图 9

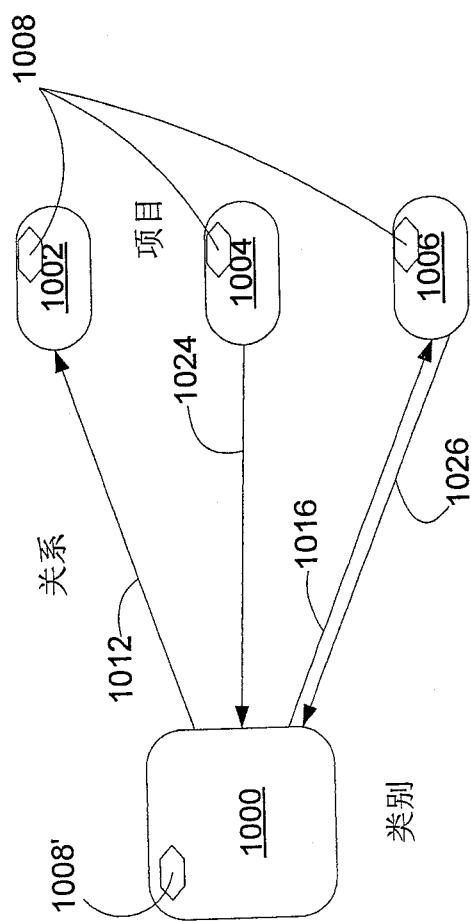


图 10

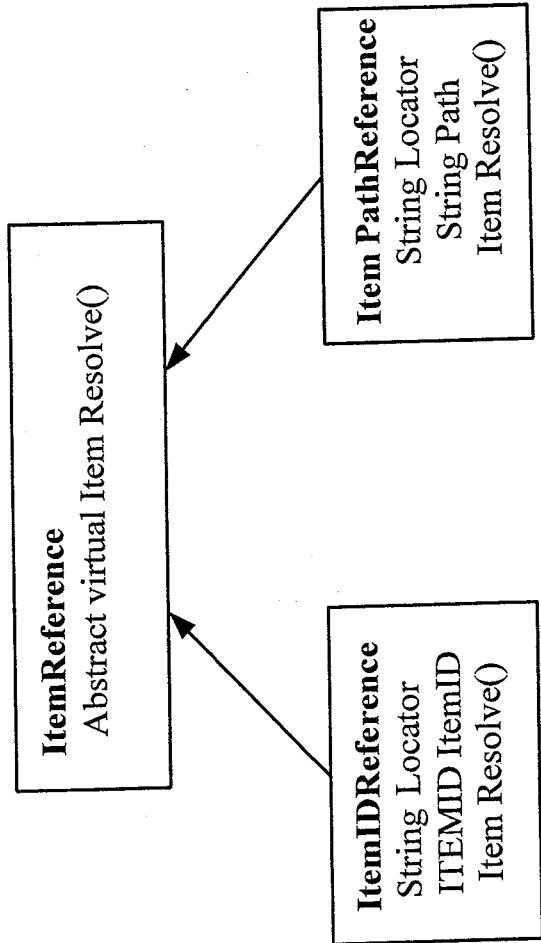


图 11

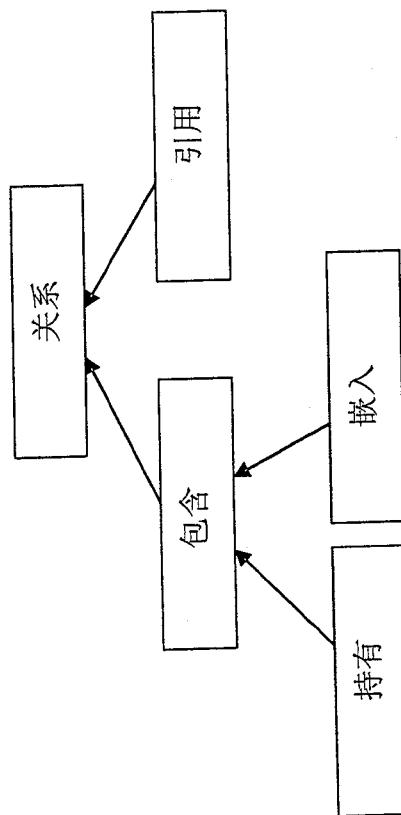


图 12

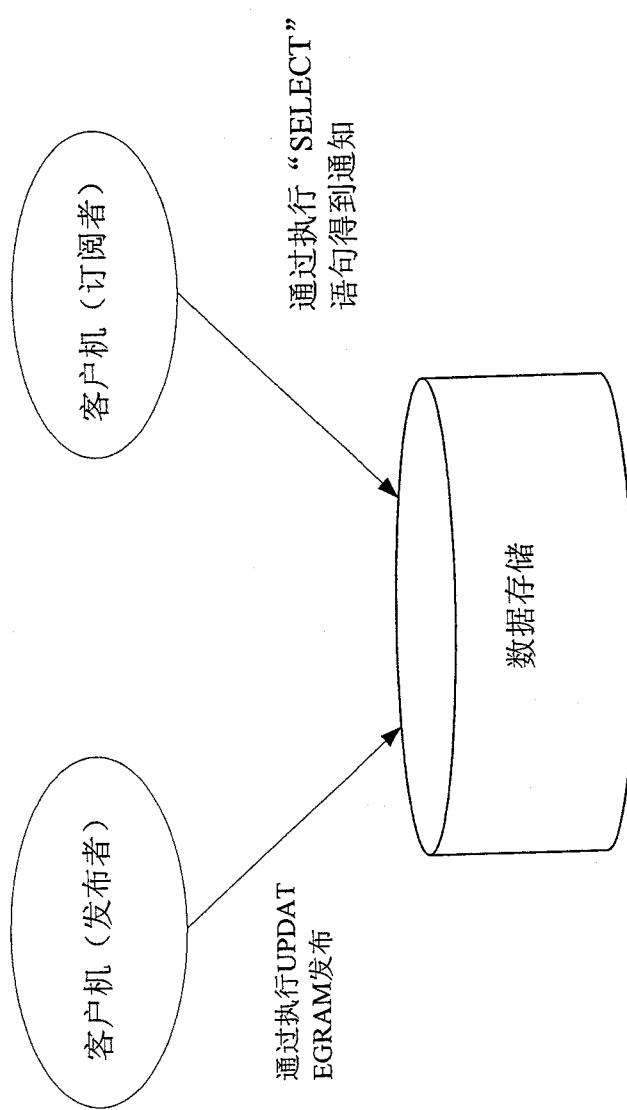


图 13

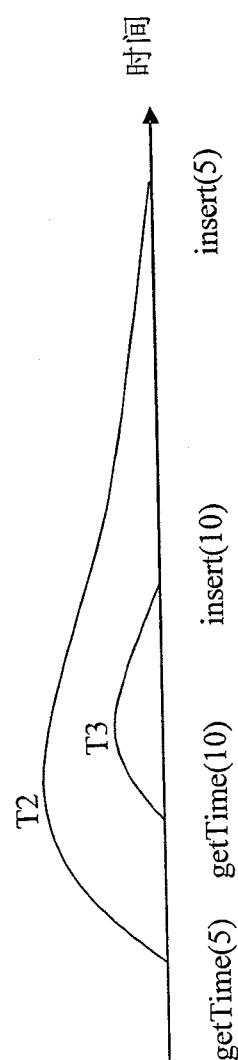


图 14

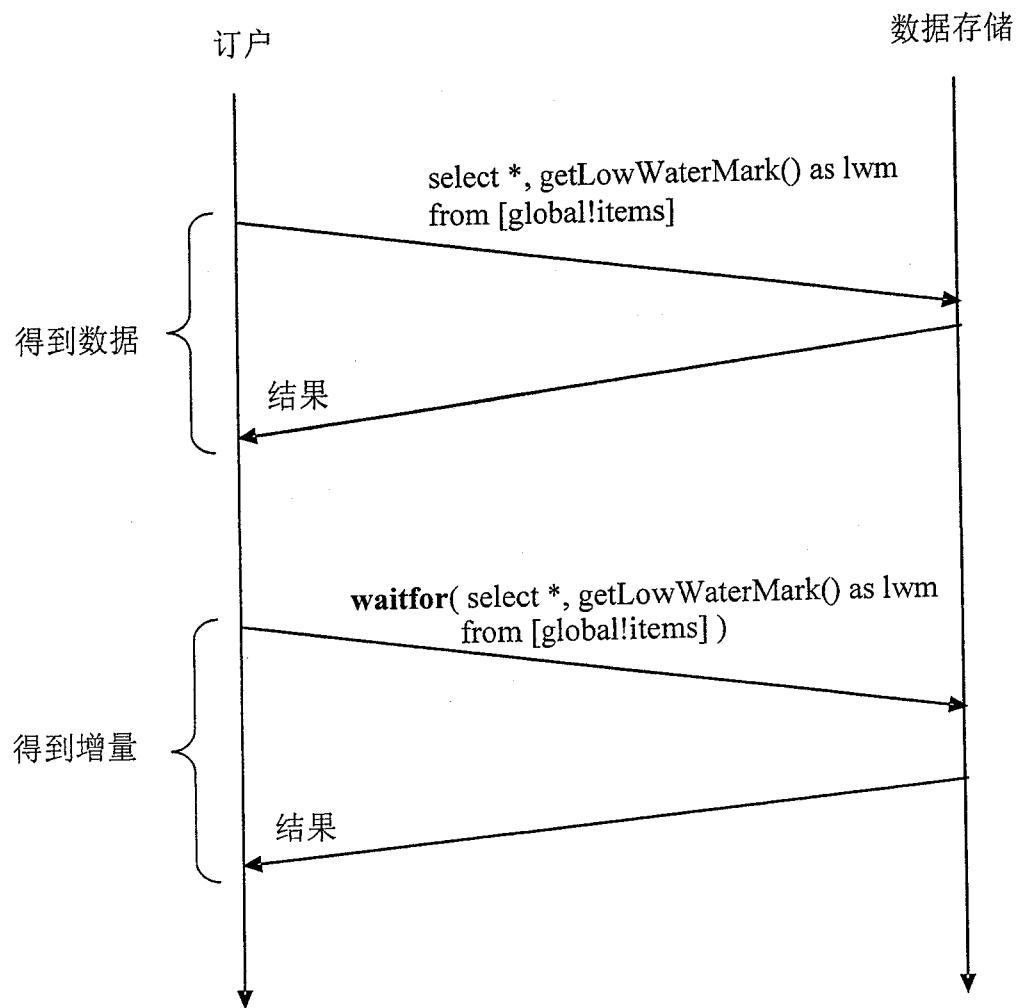


图 15

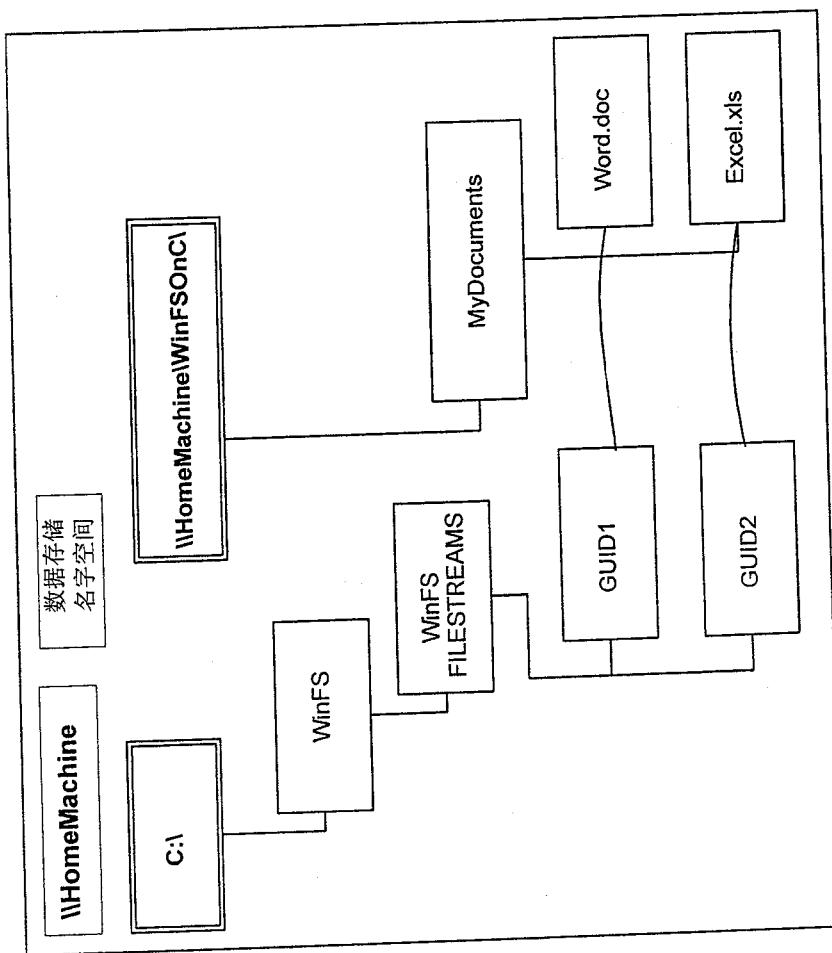


图 17

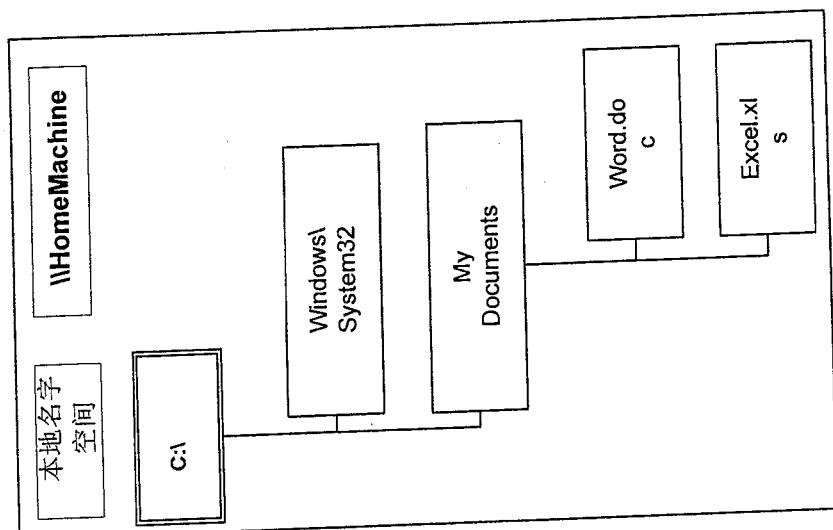


图 16

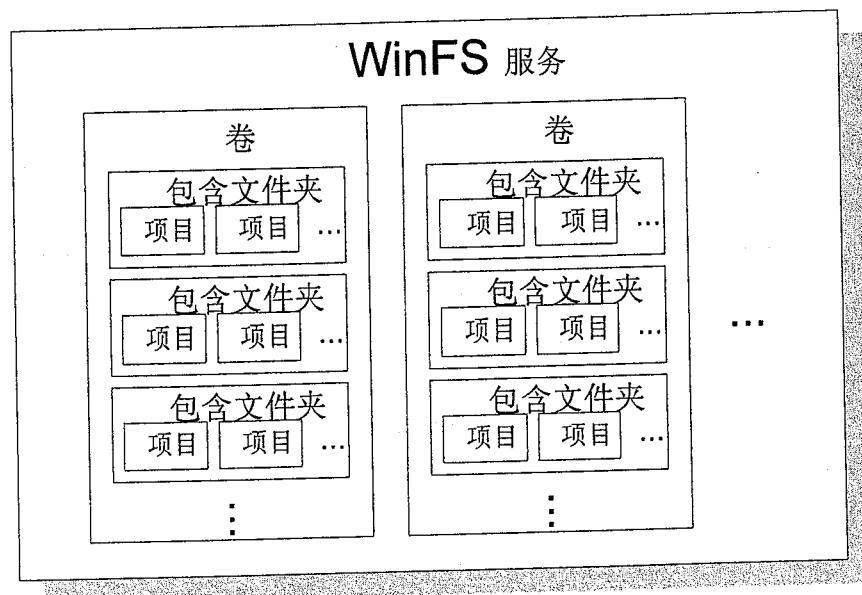


图 18

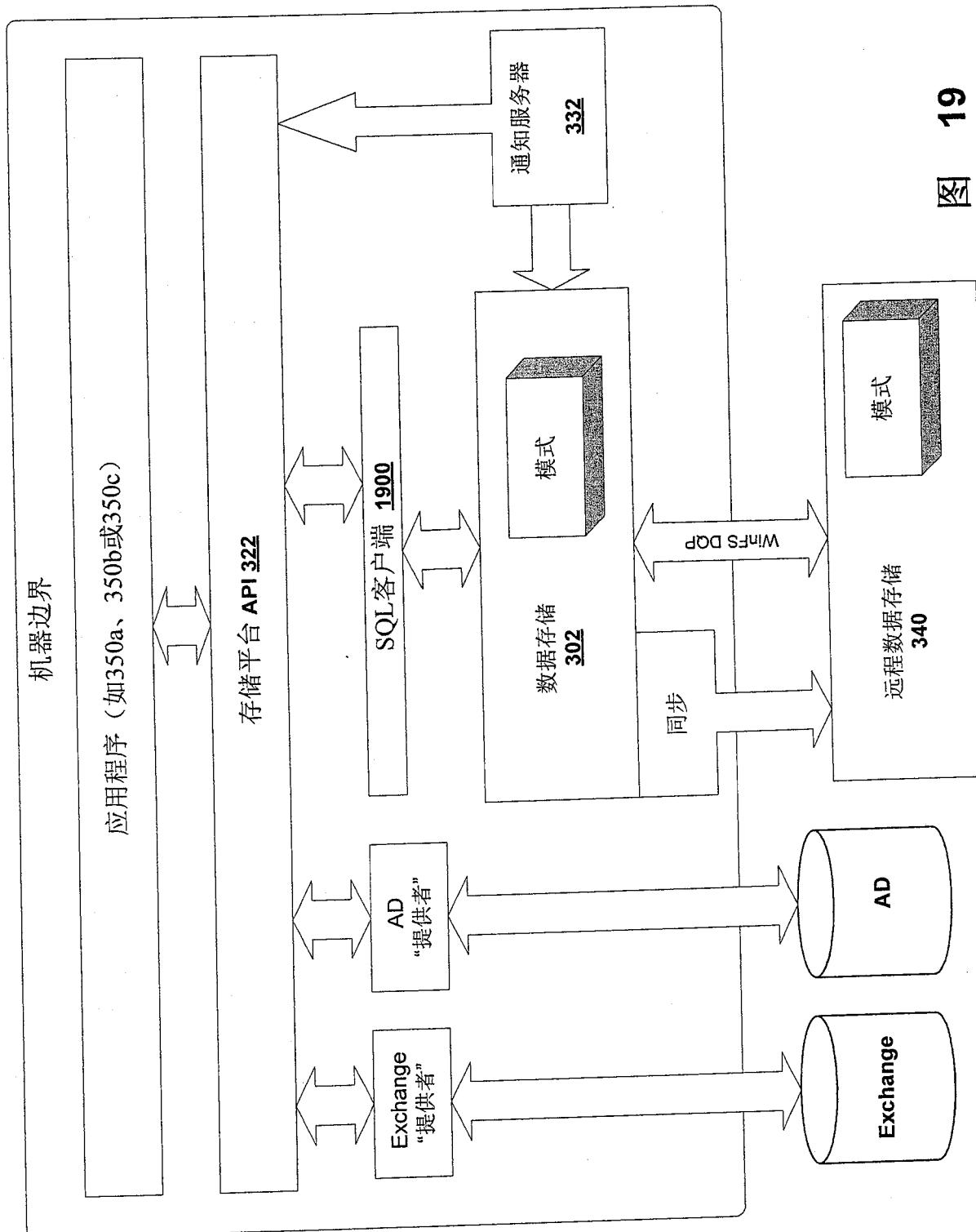


图 19

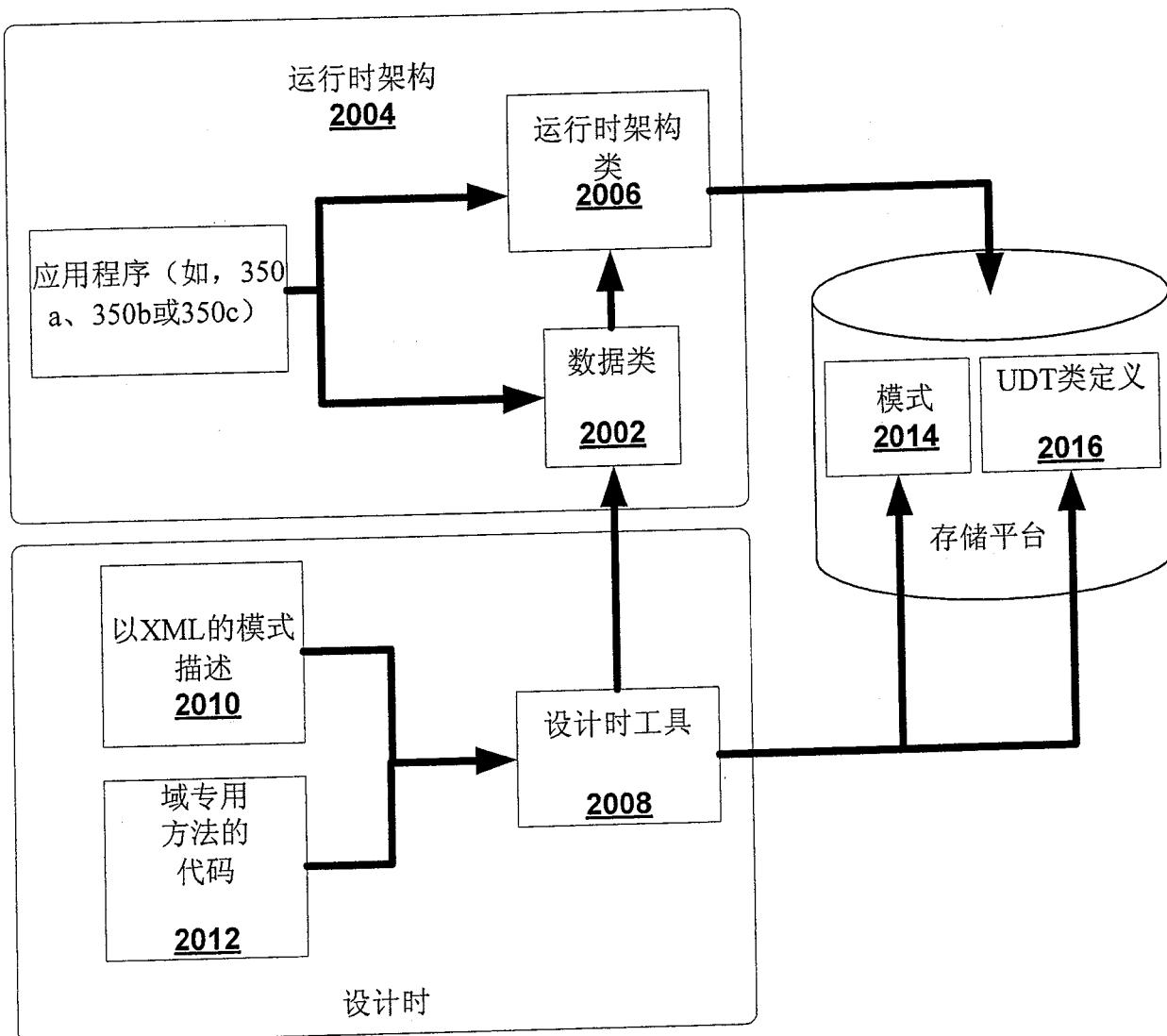


图 20

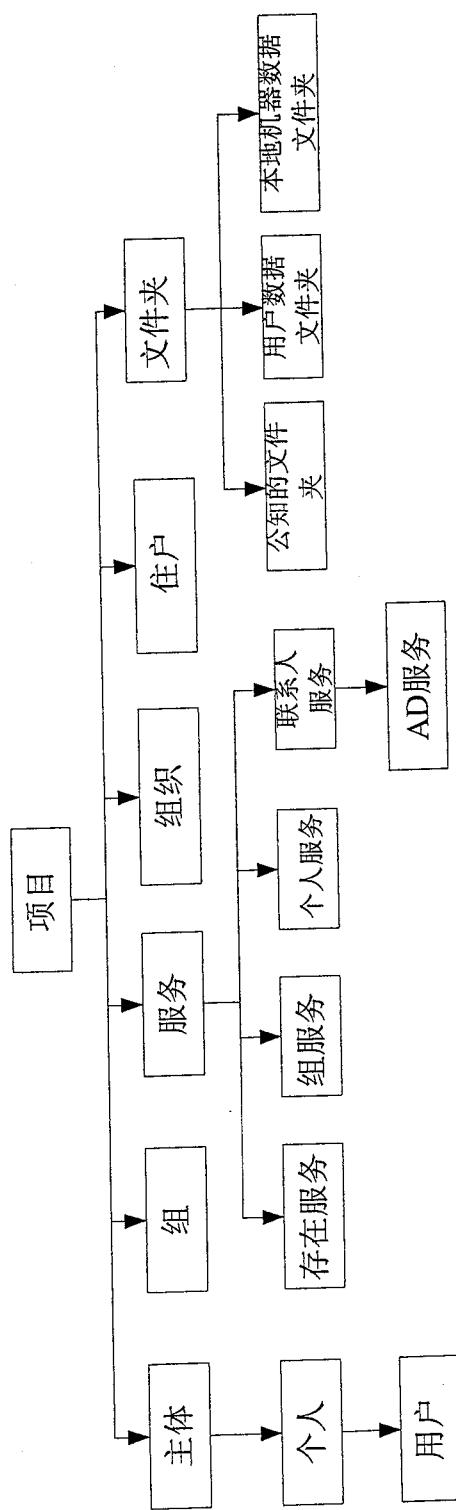


图 21A

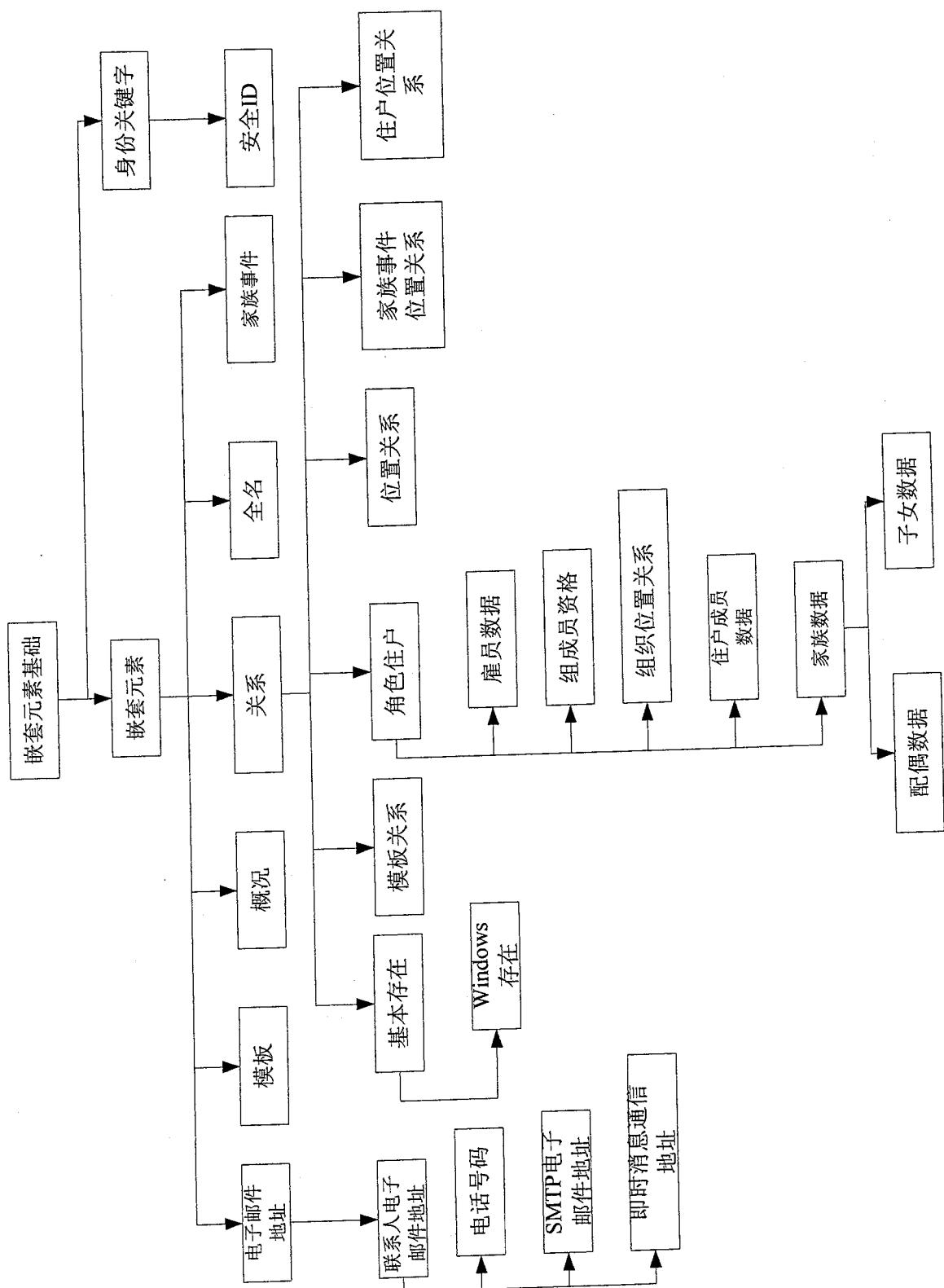


图 21B

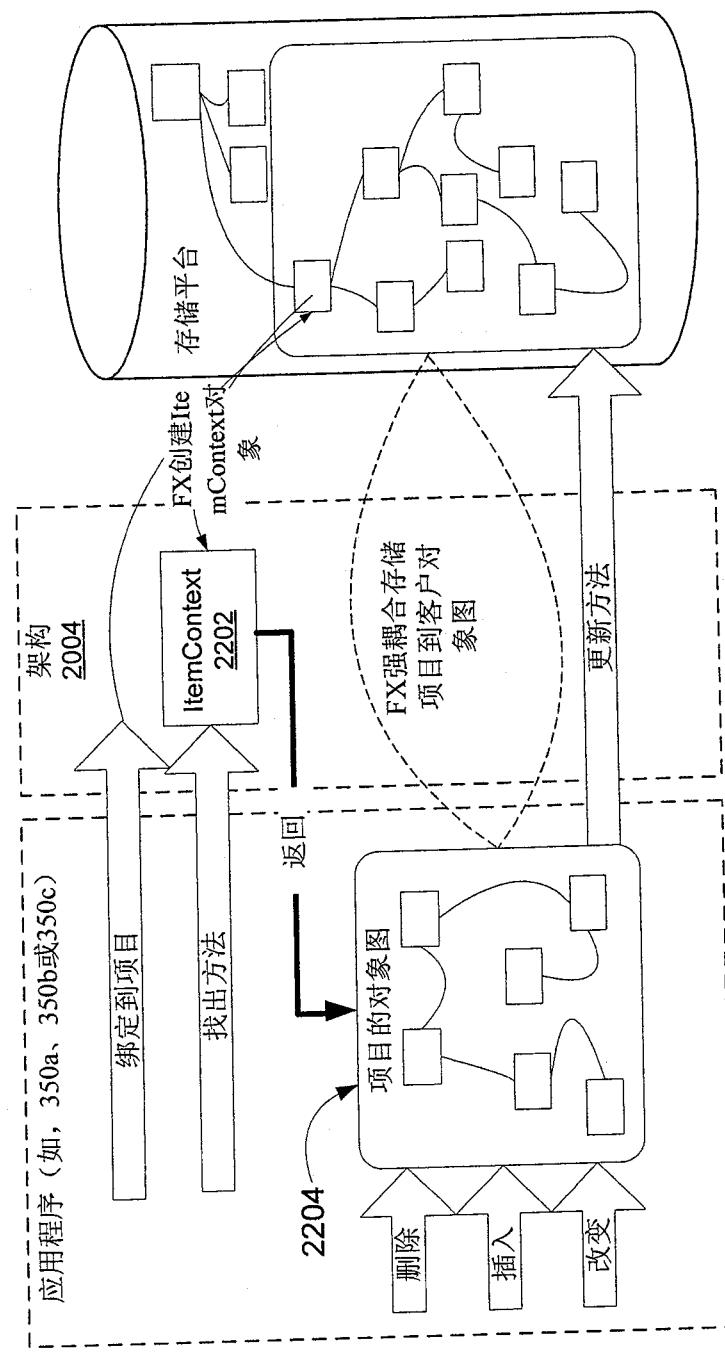


图 22

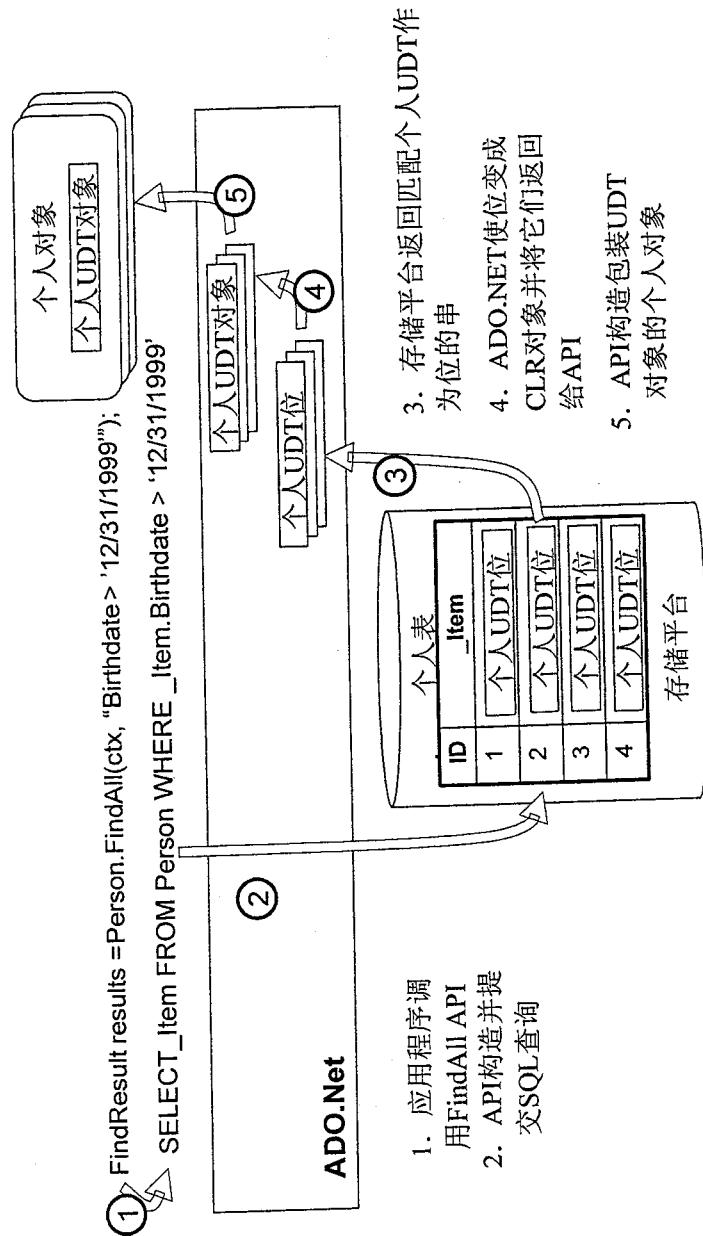


图 23

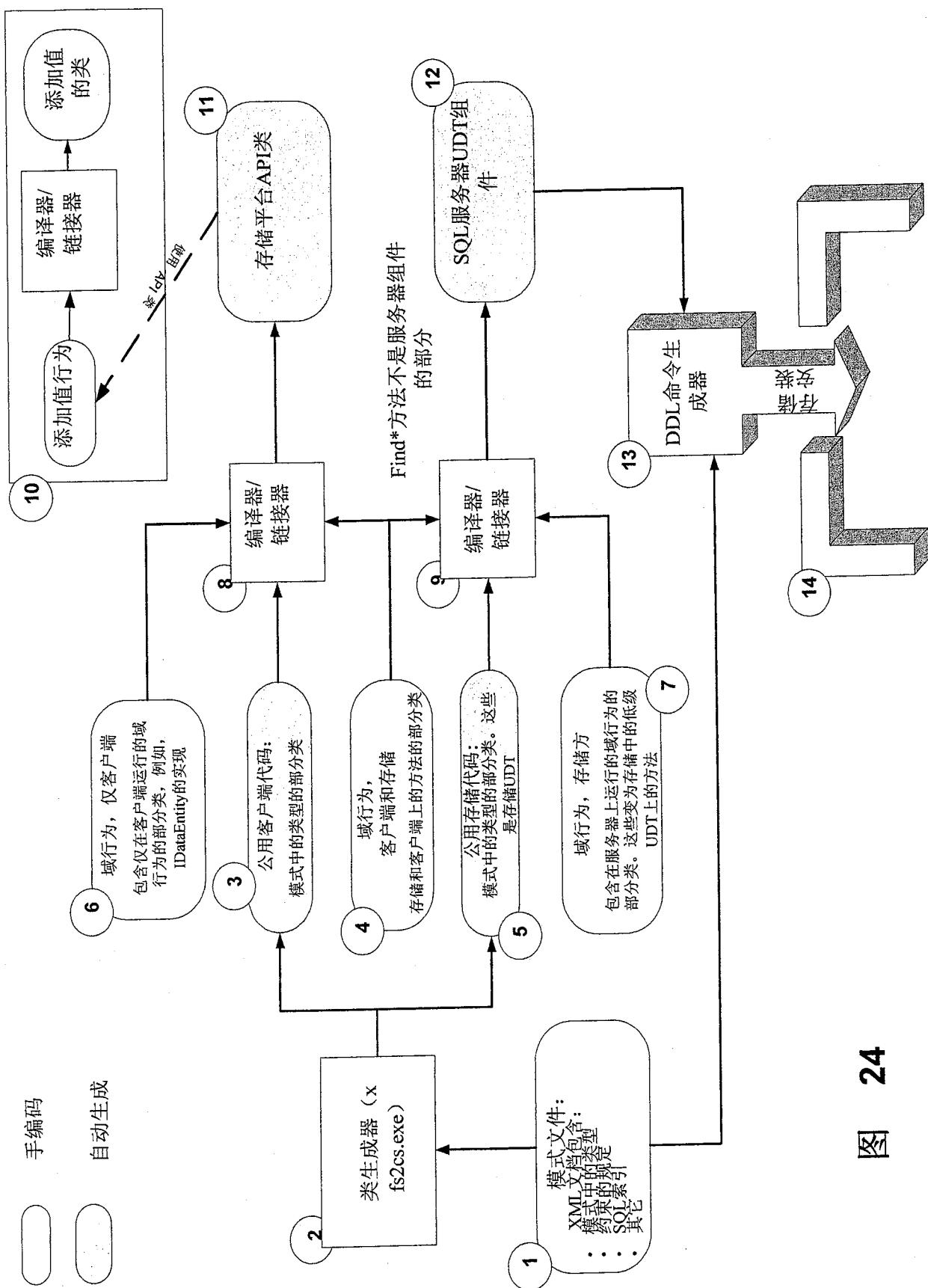
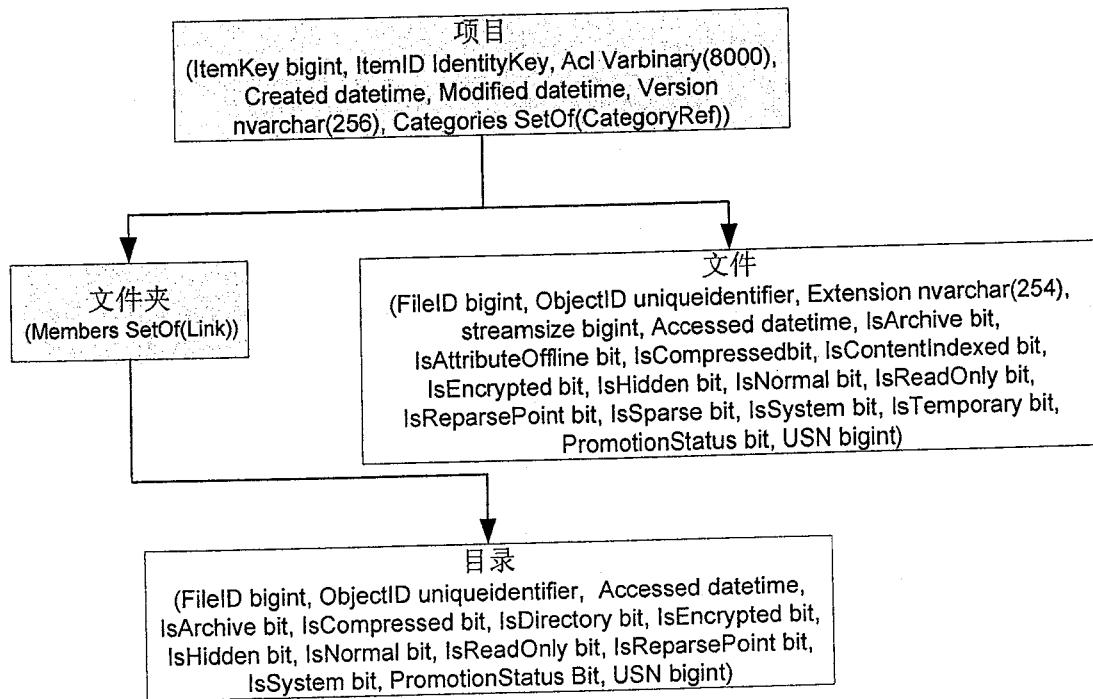


图 24



注意：灰框是在基础模式中定义的类
型，但为完整起见在这里示出

在文件模式中定义的项目

图 25

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
G R	G W	G E	G A	保留	A S	标准访问权限	对象专用访问权限																								

GR --> Generic_Read
GW --> Generic_Write
GE --> Generic_Execute
GA --> Generic_ALL
AS --> 访问SACL的权限

图 26

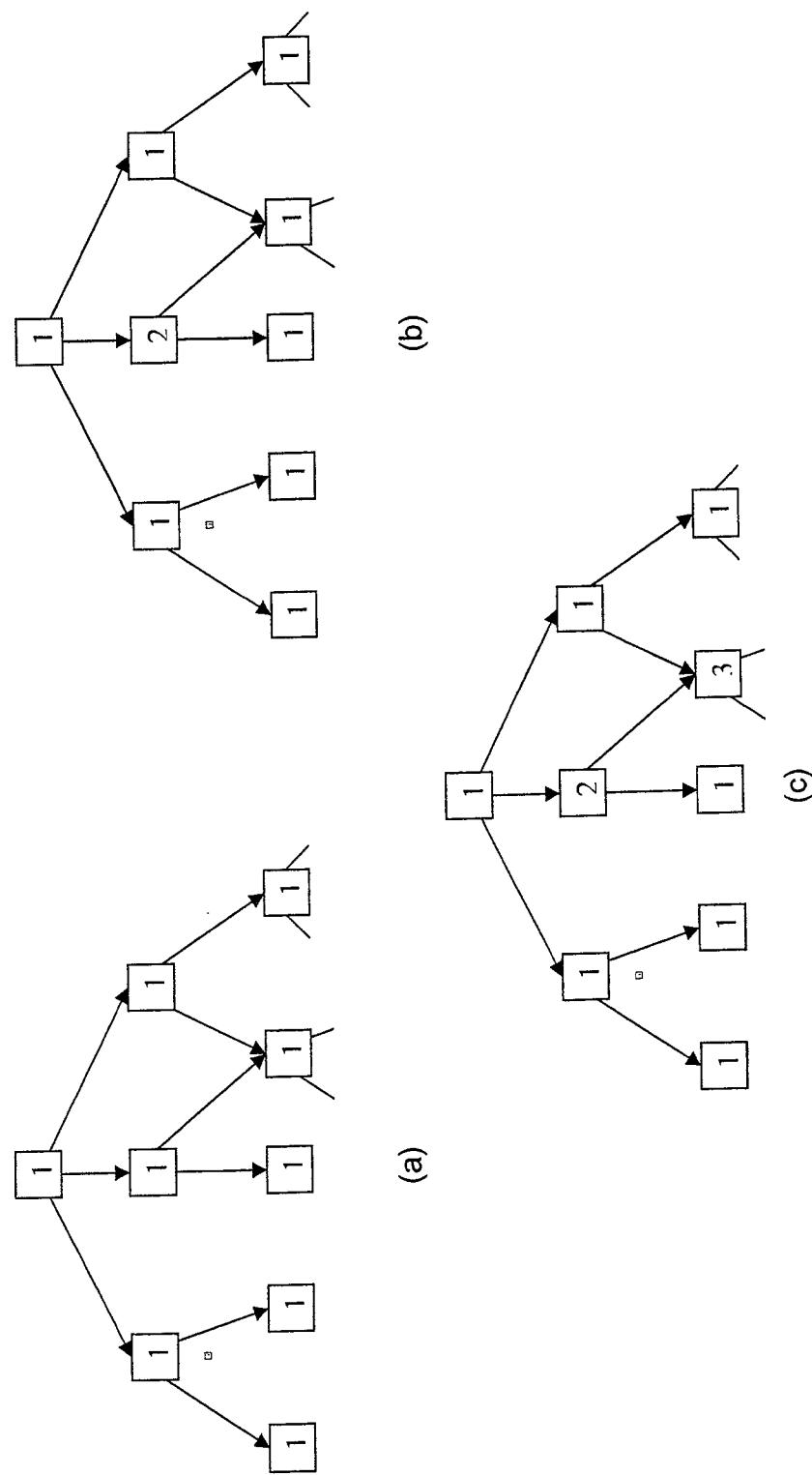


图 27

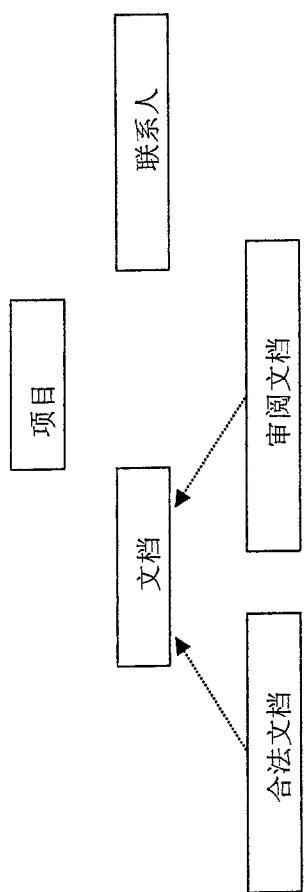


图 28

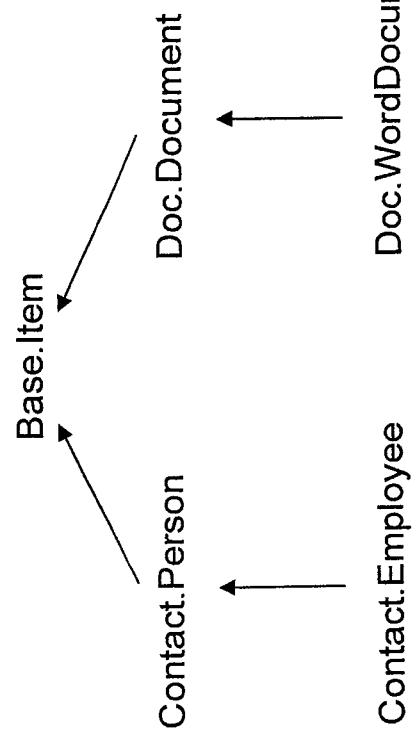


图 29