

(12) 发明专利

(10) 授权公告号 CN 101405729 B

(45) 授权公告日 2011.04.20

(21) 申请号 200780009944.3

(22) 申请日 2007.03.22

(30) 优先权数据

60/785,672 2006.03.23 US

11/725,206 2007.03.16 US

(85) PCT申请进入国家阶段日

2008.09.22

(86) PCT申请的申请数据

PCT/US2007/007261 2007.03.22

(87) PCT申请的公布数据

W02007/112009 EN 2007.10.04

(73) 专利权人 微软公司

地址 美国华盛顿州

(72) 发明人 A·阿达雅 J·A·布莱克利

P·A·拉森 S·梅尔尼克

(74) 专利代理机构 上海专利商标事务所有限公

司 31100

代理人 陈斌

(51) Int. Cl.

G06F 17/30(2006.01)

(56) 对比文件

CN 1547137 A, 2004.11.17, 全文.

US 2005/0050068 A1, 2005.03.03, 说明书第 0048 段 - 第 0062 段、图 2, 图 4.

CN 1647080 A, 2005.07.27, 全文.

审查员 杨盈霄

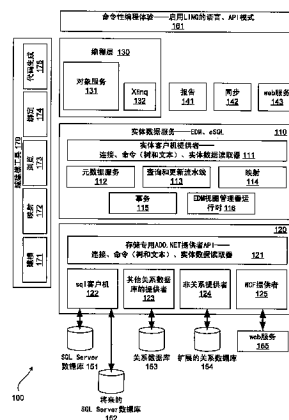
权利要求书 2 页 说明书 40 页 附图 15 页

(54) 发明名称

具有增量式视图维护的映射体系结构

(57) 摘要

提供包括用于将如可由应用程序使用的数据库映射到如持久保存在数据库中的数据的数据的映射体系结构的数据访问体系结构。映射体系结构使用两种类型的映射视图——帮助转换查询的查询视图以及帮助转换更新的更新视图。可使用增量式视图维护在应用程序与数据库之间转换数据。



1. 一种用于向应用程序提供数据服务的方法，包括：

生成按照与数据库相关联的数据库模式表达与所述应用程序相关联的应用程序模式的至少一部分的查询视图 (1302)；

生成按照所述应用程序模式表达所述数据库模式的至少一部分的更新视图 (1412)；

利用所述查询视图来代表所述应用程序查询所述数据库 (1303)；

利用所述更新视图来代表所述应用程序更新所述数据库 (1414)；

其中，利用所述更新视图来计算对所述数据库的更新包括对所述更新视图应用视图维护算法，对所述更新视图应用视图维护算法产生所述更新视图的增量 Δ 表达式，还包括使用视图展开 (1715) 将所述增量 Δ 表达式与查询视图组合。

2. 如权利要求 1 所述的方法，其特征在于，还包括从所述应用程序接收 (1411) 采用编程语言的对象，所述采用编程语言的对象包括用于更新所述数据库的数据。

3. 如权利要求 1 所述的方法，其特征在于，还包括从所述应用程序接收 (1411) 创建、插入、更新或删除指令，所述创建、插入、更新或删除指令包括用于更新所述数据库的数据。

4. 如权利要求 1 所述的方法，其特征在于，还包括从所述应用程序接收 (1411) 采用数据操纵语言 DML 的表达式，所述表达式包括用于更新所述数据库的数据。

5. 如权利要求 1 所述的方法，其特征在于，利用所述更新视图来计算对所述数据库的更新包括对所接收的用于更新所述数据库的数据应用视图维护算法。

6. 如权利要求 1 所述的方法，其特征在于，所述应用程序模式支持类、关系、继承、聚合和复杂类型。

7. 如权利要求 1 所述的方法，其特征在于，所述更新视图是使用使所述应用程序模式与所述数据库模式相关的映射来生成的。

8. 如权利要求 7 所述的方法，其特征在于，所述查询视图和所述更新视图是使用所述映射生成的。

9. 一种用于向应用程序提供数据服务的数据访问系统，包括：

生成按照与数据库相关联的数据库模式表达与所述应用程序相关联的应用程序模式的至少一部分的查询视图的组件 (110)；

生成按照所述应用程序模式表达所述数据库模式的至少一部分的更新视图的组件 (110)；

利用所述查询视图来代表所述应用程序查询所述数据库的组件 (113)；

利用所述更新视图来代表所述应用程序更新所述数据库的组件 (113)；

其中，利用所述更新视图来计算对所述数据库的更新的所述组件 (113) 对所述更新视图应用视图维护算法，对所述更新视图应用视图维护算法产生所述更新视图的增量 Δ 表达式，还包括使用视图展开将所述增量 Δ 表达式与查询视图组合的组件 (113)。

10. 如权利要求 9 所述的数据访问系统，其特征在于，还包括从所述应用程序接收采用编程语言的对象 (113)，所述采用编程语言的对象包括用于更新所述数据库的数据。

11. 如权利要求 9 所述的数据访问系统，其特征在于，还包括从所述应用程序接收创建、插入、更新或删除指令的组件 (113)，所述创建、插入、更新或删除指令包括用于更

新所述数据库的数据。

12. 如权利要求 9 所述的数据访问系统，其特征在于，还包括从所述应用程序接收采用数据操纵语言 DML 的表达式的组件 (113)，所述表达式包括用于更新所述数据库的数据。

13. 如权利要求 9 所述的数据访问系统，其特征在于，利用所述更新视图来计算对所述数据库的更新的所述组件 (113) 对所接收的用于更新所述数据库的数据应用视图维护算法。

14. 如权利要求 9 所述的数据访问系统，其特征在于，所述应用程序模式支持类、关系、继承、聚合和复杂类型。

15. 如权利要求 9 所述的数据访问系统，还包括用于生成使所述应用程序模式与所述数据库模式相关的映射的组件 (114)。

16. 如权利要求 15 所述的数据访问系统，其特征在于，所述查询视图和所述更新视图是使用所述映射生成的。

具有增量式视图维护的映射体系结构

[0001] 背景

[0002] 将应用程序与数据库连接起来一直是个问题。在 1996 年, Carey 和 DeWitt 概括了为什么众多技术, 包括面向对象数据库和持久编程语言在内, 由于查询和更新处理、事务吞吐量和缩放性而没有被普遍接受。他们推测对象关系 (O/R) 数据库在 2006 年将占据统治地位。实际上, **DB2**®和**Oracle**®数据库系统包括在常规关系引擎上方使用硬连线 O/R 映射的内置对象层。然而, 这些系统提供的 O/R 特征看上去除多媒体和空间数据类型以外, 很少用于存储企业数据。原因有数据和厂商独立、移植传统数据库的成本、当业务逻辑在数据库内而非中间层内运行时的超范围问题、以及与编程语言的集成不足。

[0003] 自 1990 年中期以来, 客户机侧数据映射层由于因特网应用程序的增长而得以普及。这样的层的核心功能在于提供展示了严格对准应用程序的数据模型、由显式映射驱动的可更新视图。众多商业产品和开放源代码项目开始提供这些能力。基本上每个企业框架提供客户机侧持久层 (例如, J2EE 中的 EJB)。大多数封装的业务应用程序, 诸如 ERP 和 CRM 应用程序, 包括专有数据访问接口 (例如, SAP R/3 中的 BAPI)。

[0004] 一种广泛使用的用于 **Java**®的开放源代码对象关系映射 (ORM) 框架是 **Hibernate**®。它支持多个继承映射场景、乐观并发控制以及综合对象服务。Hibernate 的最新版本遵循 EJB3.0 标准, 它包括 Java 持久查询语言。在商品化方面, 流行的 ORM 包括 Oracle TopLink 和 LLBLGen。后者在 .NET 平台上运行。这些和其它 ORM 与其目标编程语言的对象模型紧密耦合。

[0005] BEA 近来引入了称为 AquaLogic Data Services Platform (ALDSP, AquaLogic 数据服务平台)。它使用 XML 模式来对应用程序数据建模。使用 XQuery 从数据库和 web 服务汇编 XML 数据。ALDSP 的运行支持对多个数据源的查询, 并执行客户机侧查询优化。更新被执行为对 XQuery 视图的视图更新。如果更新不具有唯一的转换, 则开发人员需要使用命令性的代码来覆盖更新逻辑。ALDSP 的编程面基于服务数据对象 (SDO)。

[0006] 今天的客户机侧映射层提供各种不同程度的能力、健壮性和所有权的总成本。一般, 应用程序和 ORM 所使用的数据库工件 (artifact) 之间的映射具有模糊的语义, 并驱动逐个情况的推理。场景驱动实现限制了所支持的映射的范围, 且通常产生难以扩展的脆弱的运行时。很少数据访问解决方案利用由数据库社区开发的数据变换技术, 且通常依赖于自组织的解决方案来查询和更新转换。

[0007] 数据库研究贡献了可用于构建持久层的众多强大的技术。然而, 存在显著的间隙。最关键的间隙之一是支持通过映射的更新。与查询相比, 更新难以处理得多, 因为它们需要跨映射保持数据一致性, 可能会触发业务规则等。因此, 商品化数据库系统和数据访问产品对可更新视图提供非常有限的支持。近来, 研究者转向了替换方法, 诸如双向变换。

[0008] 传统上, 概念建模限于数据库和应用程序设计、反向工程和模式转换。众多设计工具使用 UML。仅最近的概念建模开始渗入产业强度 (industry-strength) 数据映射解决方案。例如, 实体和关系的概念加在 ALDSP 和 EJB3.0 两者的表面。ALDSP 将 E-R 型

联系覆盖在复杂类型的 XML 数据上方，而 EJB3.0 允许使用类注释指定对象之间的关系。

[0009] 模式映射技术用于众多数据集成产品，诸如 Microsoft 的 BizTalkServer、IBM 的 Rational Data Architect 以及 ETL 工具。这些产品允许开发人员设计数据变换或从映射中编译它们以转换电子商务消息或加载数据仓库。

[0010] 概述

[0011] 提供了用于实现和使用包括映射体系结构的数据访问体系结构的系统、方法和计算机可读介质，该映射体系结构用于将如可由应用程序使用的数据库中的数据映射到如在数据库中保存的数据。在一个实施例中，映射体系结构使用两种类型的映射视图——帮助转换查询的查询视图以及帮助转换更新的更新视图。增量式的视图维护可用于在应用程序和数据库之间转换数据。以下描述其它方面和实施例。

[0012] 附图简述

[0013] 将参考附图进一步描述根据本发明的用于具有增量式视图维护的映射体系结构的系统和方法，附图中：

[0014] 图 1 示出了如此所构想的示例性实体框架的体系结构。

[0015] 图 2 示出了示例性关系模式。

[0016] 图 3 示出了示例性实体数据模型 (EDM) 模式。

[0017] 图 4 示出了实体模式 (左) 和数据库模式 (右) 之间的映射。

[0018] 图 5 示出了按照对实体模式和关系模式的查询表示的映射。

[0019] 图 6 示出了映射编译器为图 5 中的映射生成的双向视图——查询和更新视图。

[0020] 图 7 示出了用于利用物化视图维护算法来通过双向视图传播更新的过程。

[0021] 图 8 示出了映射设计器用户界面。

[0022] 图 9 示出了编译用映射规范语言 (MSL) 指定的映射以生成查询和更新视图。

[0023] 图 10 示出了更新处理。

[0024] 图 11 示出了对象关系 (OR) 映射器的示例性逻辑部分。

[0025] 图 12 示出了当处理用 MSL 规范指定的映射时由实体数据平台 (EDP) 生成查询和更新视图。

[0026] 图 13 示出了在查询转换中使用 QM 视图。

[0027] 图 14 示出了在更新转换中使用 UM 视图。

[0028] 图 15 示出了映射视图的编译时和运行时处理。

[0029] 图 16 示出了视图编译过程中各个组件的交互。

[0030] 图 17 示出了 EDP 查询转换器 (EQT) 体系结构。EQT 利用映射元数据来将查询从对象 / EDM 空间转换到数据库空间。

[0031] 图 18 示出了构成各种 Δ 表达式以根据对象的 Δ 表达式获得表的 Δ 表达式。

[0032] 详细描述

[0033] 新颖的数据访问体系结构

[0034] 在一个实施例中，本发明可在如本章节中所述的新颖的数据访问体系结构——“实体框架”内实现，并包括其各方面。这样的实体框架的示例有由 MICROSOFT 公司开发的 ADO.NET vNEXT 数据访问体系结构。以下是对 ADO.NET vNEXT 数据访问体系结构以及众多实现专用细节的一般描述，它们不应被认为是实现本发明所必需的。

[0035] 概观

[0036] 传统的客户机—服务器应用将对其数据的查询和持久操作转交给数据库系统。数据库系统对行和表形式的数据进行操作，而应用程序按照（类、结构等）等更高级编程语言构造来对数据操作。应用程序和数据库层之间的数据操纵服务中的阻抗失配甚至在传统系统中也是成问题的。随着面向服务的体系结构（SOA）、应用程序服务器和多层应用程序的出现，对与编程环境有良好集成并且能在任何层中操作的数据访问和操纵服务的需求也大量增长。

[0037] 微软的 ADO.NET 实体框架是用于针对将抽象级从关系级提升到概念（实体）级从而显著减少了应用程序和数据中心服务的阻抗失配的数据进行编程的平台。实体框架、总体系统体系结构和底层技术的各个方面将在以下描述。

[0038] 介绍

[0039] 现代应用程序要求所有层中的数据管理服务。它们需要日益丰富的各种形式的数据，这不仅包括结构化业务数据（诸如，顾客和订单）而且还包括半结构化和非结构化内容，诸如电子邮件、日历、文件和文档。这些应用程序需要集成来自多个数据源的数据以及收集、净化、变换和存储该数据以启用更灵活的决策过程。这些应用程序的开发者需要数据访问、编程和开发工具来增加其生产率。尽管关系数据库已经成为大多数结构化数据实际上的存储，但在这样的数据库所展示的数据模型（及能力）与应用程序所需的建模能力之间往往存在失配——即公知的阻抗失配问题。

[0040] 另有两个因素在企业系统设计中扮演了重要的角色。首先，应用程序的数据表示与底层数据库的数据表示往往不同地演化。其次，众多系统由具有不同程度的能力的不同数据库后端组成。中间层的应用程序逻辑负责调和这些不同并表示更统一的数据视图的数据变换。这些数据变换很快变得复杂。尤其当底层数据需要可更新时，实现它们是一个难题，并对应用程序增加了复杂性。应用程序开发的相当部分——在某些情况下高达 40%——专用于编写定制数据访问逻辑以便围绕这些问题进行工作。

[0041] 对数据中心服务也存在相同且一样严重的问题。常规的服务，诸如查询、更新和事务在逻辑模式（关系）级实现。然而，大量较新的服务，诸如复制和分析，最好在一般与较高级的概念数据模型相关联的工件上操作。例如，SQL SERVER®复制发明了被称为“逻辑记录”的结构来表示有限形式的实体。类似地，SQL Server 报告服务在被称作语义数据模型语言（SDML）的实体型数据模型的顶部构建报告。这些服务中的每一个具有定制工具来定义概念实体并将其向下映射到关系表——顾客实体从而将需要按照一路用于复制，另一路用于报告构建还有一路用于其它分析服务等的方式被定义和映射。与应用程序一样，每一服务通常以构建该问题的定制解决方案而告终，且因此在这些服务之间存在代码重复和有限的互操作性。

[0042] 对象到关系映射（ORM）技术，诸如 HIBERNATE 和 ORACLE TOPLINK 是定制数据访问逻辑的流行的替换方案。数据库与应用程序之间的映射以定制结构或经由模式注释来表达。这些定制结构可能看起来像概念模型；然而，应用程序不能直接针对该概念模型进行编程。尽管映射在数据库与应用程序之间提供了某一程度的独立性，但这些解决方案并未良好解决处理具有相同数据稍有不同的视图的多个应用程序（例如，考虑想要查看顾客实体的不同投影的两个应用程序）或对倾向于更加动态的服务的需求（先

验类生成技术不能良好用于数据服务，因为底层数据库演化得更快）的问题。

[0043] ADO.NET 实体框架是显著减少应用程序和数据中心服务的阻抗失配的用于针对数据进行编程的平台。它至少在以下方面区别于其它系统和解决方案：

[0044] 1. 实体框架定义一丰富概念数据模型（实体数据模型即 EDM）以及在该模型的实例上操作的新数据操纵语言（实体 SQL）。与 SQL 一样，EDM 是基于值的，即 EDM 定义实体的结构方面而非行为（或方法）。

[0045] 2. 该模型由包括支持强大的查询和更新双向（EDM- 关系）映射的中间件映射引擎的运行时具体化。

[0046] 3. 应用程序和服务可直接针对基于值的概念层或针对可层叠在概念（实体）抽象上的编程语言专用对象抽象来编程，提供 ORM 型功能。我们相信，与对象相比，基于值的 EDM 概念抽象是用于在应用程序和数据中心服务之间共享数据的更灵活的基础。

[0047] 4. 最后，实体框架利用了采用查询表达式来本机扩展编程语言的微软的新语言集成查询（LINQ）技术以进一步减少，对某些情形甚至完全消除应用程序的阻抗失配。

[0048] ADO.NET 实体框架可并入更大的框架，诸如微软的 .NET 框架。

[0049] 数据访问体系结构的描述的其余部分，在 ADO.NET 实体框架实施例的上下文中，被如下组织。“动机”章节提供了实体框架的附加动机。“实体框架”章节提供了实体框架和实体数据模型。“编程模式”章节描述了实体框架的编程模式。“对象服务”章节概括了对象服务模块。“映射”章节关注实体框架的映射组件，而“查询处理”和“更新处理”章节说明了如何处理查询和更新。“元数据”和“工具”描述了实体框架的元数据子系统和工具组件。

[0050] 动机

[0051] 该章节讨论了为何较高级数据建模层对应用程序和数据中心服务变为必需的。

[0052] 数据应用程序中的信息级

[0053] 当今用于数据库设计的主要信息建模方法将信息模型分解成四个主要的级：物理、逻辑（关系）、概念和编程 / 表示。

[0054] 物理模型描述了数据在物理资源，诸如存储器、电线或盘中如何表示。在此层讨论的概念的词汇包括记录格式、文件分区和分组、堆以及索引。物理模型对应用程序一般是不可见的——对物理模型的改变不应影响应用程序逻辑，但可能影响应用程序执行。

[0055] 逻辑数据模型是目标域的完全且精确的信息模型。关系模型是大多数逻辑数据模型的选择的表示。在逻辑级讨论的概念包括表、行、主键 / 外键约束和规范化。尽管规范化帮助实现数据一致性、增加的并发性和更好的 OLTP 性能，但它也对应用程序引入显著的挑战。逻辑级的规范化数据通常太过细碎，应用程序逻辑需要将来自多个表的行汇编成更加类似于应用程序域的工件的较高层实体。

[0056] 概念模型从问题域及其关系捕捉核心信息实体。公知的概念模型有由 Peter Chen 在 1976 年引入的实体关系模型。UML 是更新近的概念模型的示例。大多数应用程序在应用程序开发生命周期的早期涉及概念设计阶段。然而，不幸的是，概念数据模型示意图保持“钉在墙上”，随时间与应用程序实现的实际情况日益脱节。实体框架的一个重要目的在于使得概念数据模型（由在下一章节中描述的实体数据模型体现）成为数据平台

的具体的可编程的抽象。

[0057] 编程 / 表示模型描述了概念模型的实体和关系需要如何基于手头的任务以不同形式显示（表示）。某些实体需要被变换成编程语言对象以实现应用程序业务逻辑；其它的实体需要被变换成 XML 流用于 web 服务调用；还有其它的实体出于用户界面数据绑定目的而需要被变换成存储器内结构，诸如列表或字典。自然地，不存在通用编程模型或表示形式，因此，应用程序需要灵活的机制将实体变换成各种表示形式。

[0058] 大多数的应用程序和数据中心服务想要按照诸如订单的高级概念而非关于在关系数据库模式中对其规范化订单的若干表来思考。订单可将其自己在表示 / 编程级上显示为封装与订单相关联的状态和逻辑的 VisualBasic 或 C# 中的类实例或用于与 web 服务通信的 XML 流。不存在一个正确的表示模型，然而提供具体的概念模型，然后能够使用该模型作为对各个表示模型的和其它较高级数据服务的灵活映射的基础是有价值的。

[0059] 应用程序和服务的演化

[0060] 基于数据的应用程序在 10-20 年前一般被结构化为数据单块；具有在逻辑模式级由与数据库系统交互的带有由动宾功能（例如，创建一订单、更新一顾客）分解的逻辑的封闭系统。若干重要的趋势造就了今天分解和部署现代的基于数据的应用程序的方式。这其中主要有面向对象分解、服务级应用程序组成以及较高层数据中心服务。概念实体是当今应用程序的重要部分。这些实体必须被映射到各种表示，并绑定至各种服务。不存在一个正确的表示或服务绑定：XML、关系和对象表示都是重要的，但没有一个能满足所有的应用程序。从而，存在对支持较高级数据建模层且也允许多个表示层插入的框架的需求——实体框架的目标在于满足这些要求。

[0061] 数据中心服务也以类似方式演化。20 年前“数据平台”提供的服务是最小的，且围绕 RDBMS 的逻辑模式。这些服务包括查询和更新、原子事务以及诸如备份和加载 / 提取的大块操作。

[0062] SQL Server 本身从传统的 RDBMS 演化成对在概念模式级实现的实体提供多个高价值数据中心服务的完全数据平台。SQL Server 产品中的若干较高级数据中心服务，例如复制、报告构建器，越来越多地在概念模式级提供其服务。当前，这些服务中的每一个具有描述概念实体并将其向下映射至底层逻辑模式级的单独工具。实体框架的目标在于提供所有这些服务可共享的通用的高级概念抽象。

[0063] 实体框架

[0064] 在本文所述的实体框架之前存在的微软的 ADO.NET 框架是允许应用程序连接至数据存储并以各种方式操纵包含在其中的数据的数据访问技术。它是微软 .NET 框架的一部分，且它与 .NET 框架类库的其余部分高度集成。之前的 ADO.NET 框架具有两个主要部分：提供者和服务。ADO.NET 提供者是了解如何与特定数据存储对话的组件。提供者由三个核心功能组成：连接管理对底层数据源的访问；命令表示要针对数据源执行的命令（查询、过程调用等）；而数据读取器表示命令执行的结果。ADO.NET 服务包括提供者中立的组件，诸如允许离线数据编程场景的 DataSet（数据集）。（DataSet 是无论数据源是什么都提供一致关系编程模型的数据驻留在存储器上的表示。）

[0065] 实体框架——概观

[0066] ADO.NET 实体框架构建在之前存在的 ADO.NET 提供者模型上，并添加了以下

功能：

- [0067] 1. 帮助构建概念模式的新概念数据模型，实体数据模型 (EDM)。
- [0068] 2. 操纵 EDM 的实例的新数据操纵语言 (DML) 实体 SQL，以及与不同的提供者通信的查询的程序表示 (规范命令树)。
- [0069] 3. 定义概念模式和逻辑模式之间的映射的能力。
- [0070] 4. 针对概念模式的 ADO.NET 提供者编程模型。
- [0071] 5. 提供 ORM 型功能的对象服务层。
- [0072] 6. 使得易于针对如来自 .NET 语言的对象的数据编程的与 LINQ 技术的集成。

[0073] 实体数据模型

[0074] 实体数据模型 (EDM) 允许开发丰富数据中心应用程序。它来自 E-R 域的概念扩展经典关系模型。在此处提供的示例性实施例中，EDM 中的组织概念包括实体和关系。实体表示具有身份的顶级项目，而关系用于使两个或多个实体相关 (或描述其间关系)。

[0075] 在一个实施例中，EDM 是如同关系模型 (和 SQL) 一样基于值的，而非 C#(CLR) 一样基于对象 / 引用的。可容易地在 EDM 上方层叠若干对象编程模型。类似地，EDM 可映射到一个或多个 DBMS 实现供持久保存。

[0076] EDM 和实体 SQL 表示更丰富的数据模型和数据平台的数据操纵语言，且旨在允许诸如 CRM 和 ERP 的应用程序，诸如报告、业务智能、复制和同步的数据密集服务以及数据密集应用程序在更接近其需求的结构和语义级别上建模和操纵数据。现在讨论关于 EDM 的各个概念。

[0077] EDM 类型

[0078] EntityType (实体类型) 描述实体的结构。实体可具有描述实体的结构的零个或多个属性 (性质、字段)。另外，实体类型必须定义键——其值在实体的集合内唯一标识该实体实例的一组属性。实体类型可从另一实体类型 (或子类型) 导出——EDM 支持单继承模型。实体的属性可以是简单或复杂类型的。SimpleType (简单类型) 表示标量 (或原子) 类型 (例如，整数、串)，而 ComplexType (复杂类型) 表示结构化属性 (例如，地址)。ComplexType 由零个或多个属性组成，它们本身可以是标量或复杂类型的属性。Relationship Type (关系类型) 描述两个 (或多个) 实体类型之间的关系。EDM 模式提供类型的分组机制——必须按照模式定义类型。与类型名组合的模式的名空间唯一标识特定类型。

[0079] EDM 实例模型

[0080] 实体实例 (或简称为实体) 在逻辑上被包含在 EntitySet (实体集) 内。EntitySet 是实体的同质集合，即 EntitySet 中的所有实体必须具有相同 (或派生的) EntityType。EntitySet 在概念上类似于数据库表，而实体类似于表的行。实体实例必须恰好属于一个实体集。以类似方式，关系实例在逻辑上被包含在 RelationshipSet (关系集) 内。RelationshipSet 的定义为关系确定范围。即，它标识了持有参与关系的实体类型的实例的 EntitySet。RelationshipSet 在概念上类似于数据库中的链接表。SimpleType 和 ComplexType 仅可被实例化为 EntityType 的属性。EntityContainer (实体容器) 是 EntitySet 和 RelationshipSet 的逻辑分组——类似于模式是 EDM 类型的分组机制。

[0081] 示例 EDM 模式

[0082] 以下示出示例 EDM 模式：

[0083] <? xml version = " 1.0" encoding = " utf-8" ? >

[0084] <Schema Namespace = " AdventureWorks" Alias = " Self" ...>

[0085] <EntityContainer Name = " AdventureWorksContainer" >

[0086] <EntitySet Name = " ESalesOrders"

[0087] EntityType = " Self.ESalesOrder" />

[0088] <EntitySet Name = " ESalesPersons"

[0089] EntityType = " Self.ESalesPerson" />

[0090] <AssociationSet Name = " ESalesPersonOrders"

[0091] Association = " Self.ESalesPersonOrder" >

[0092] <End Role = " ESalesPerson"

[0093] EntitySet = " ESalesPersons" />

[0094] <End Role = " EOrder" EntitySet = " ESalesOrders" />

[0095] </AssociationSet>

[0096] </EntityContainer>

[0097] <!--销售订单类型分层结构-->

[0098] <EntityType Name = " ESalesOrder" Key = " Id" >

[0099] <Property Name = " Id" Type = " Int32"

[0100] Nullable = " false" />

[0101] <Property Name = " AccountNum" Type = " String"

[0102] MaxLength = " 15" />

[0103] </EntityType>

[0104] <EntityType Name = " EStoreSalesOrder"

[0105] BaseType = " Self.ESalesOrder" >

[0106] <Property Name = " Tax" Type = " Decimal"

[0107] Precision = " 28" Scale = " 4" />

[0108] </EntityType>

[0109] <!--个人实体类型-->

[0110] <EntityType Name = " ESalesPerson" Key = " Id" >

[0111] <!--Propertiesfrom SSalesPersons table-->

[0112] <Property Name = " Id" Type = " Int32"

[0113] Nullable = " false" />

[0114] <Property Name = " Bonus" Type = " Decimal"

[0115] Precision = " 28" Scale = " 4" />

[0116] <!--Propertiesfrom SEmployees table-->

[0117] <Property Name = " Title" Type = " String"

[0118] MaxLength = " 50" />

[0119] <Property Name = " HireDate" Type = " DateTime" />

```
[0120] <!--Propertiesfrom the SContacts table-->
[0121] <Property Name = " Name" Type = " String"
[0122]         MaxLength = " 50" />
[0123] <Property Name = " Contact" Type = " Self.ContactInfo"
[0124]         Nullable = " false" />
[0125] </EntityType>
[0126] <ComplexType Name = " ContactInfo" >
[0127] <Property Name = " Email" Type = " String"
[0128]         MaxLength = " 50" />
[0129] <Property Name = " Phone" Type = " String"
[0130]         MaxLength = " 25" />
[0131] </ComplexType>
[0132] <Association Name = " ESalesPersonOrder" >
[0133] <End Role = " EOrder" Type = " Self.ESalesOrder"
[0134]         Multiplicity = " *" />
[0135] <End Role = " ESalesPerson" Multiplicity = " 1"
[0136]         Type = " Self.ESalesPerson" />
[0137] </Association>
[0138] </Schema>
```

[0139] 高级体系结构

[0140] 该章节概括了 ADO.NET 实体框架的体系结构。其主要功能组件在图 1 示出，并包括以下：

[0141] 数据源专用提供者。实体框架 100 在 ADO.NET 数据提供者模型上构建。存在用于诸如 SQL Server 151、152、关系源 153、非关系 154 和 Web 服务 155 源的若干数据源的专用提供者 122-125。提供者 122-125 可从存储专用 ADO.NET 提供者 API 121 调用。

[0142] EntityClient(实体客户机)提供者。EntityClient 提供者 110 表示具体的概念编程层。它是新的基于值的数据提供者，其中按照 EDM 实体和关系来访问数据，且使用基于实体的 SQL 语言(实体 SQL)来查询和更新。EntityClient 提供者 111 形成实体数据服务 110 包的一部分，该包还包括元数据服务 112、查询和更新流水线 113、事务支持 115、视图管理器运行时 116 以及支持平面关系表上的可更新 EDM 视图的视图映射子系统 114。表和实体之间的映射经由映射规范语言来声明地指定。

[0143] 对象服务和其它编程层。实体框架 100 的对象服务组件 131 提供实体上的丰富对象抽象、这些对象上的丰富的服务集，并允许应用程序使用熟悉的编程语言构造在命令性编码体验 161 中编程。该组件提供对象的状态管理服务(包括改变跟踪、身份解析)，支持用于导航和加载对象和关系的服务，使用诸如 Xlinq 132 的组件支持经由 LINQ 和实体 SQL 的查询，并允许更新和持久保存对象。

[0144] 实体框架允许类似于 130 的多个编程层插到由 EntityClient 提供者 111 所展示的基于值的数据服务层 110 上。对象服务 131 组件是覆盖在 CLR 对象表面并提供 ORM 型功能的一个这样的编程层。

[0145] 元数据服务 112 组件为实体框架 100 的设计时和运行时需求管理元数据，并对实体框架应用。经由元数据接口展示与 EDM 概念（实体、关系、EntitySet、RelationshipSet）、存储概念（表、列、约束）以及映射概念相关联的所有元数据。元数据组件 112 也用作支持模型驱动应用程序设计的域建模工具之间的链接。

[0146] 设计和元数据工具。实体框架 100 与域设计器 170 集成以启用模型驱动的应用程序开发。工具包括 EDM 设计工具、建模工具 171、映射设计工具 172、浏览设计工具 173、绑定设计工具 174、代码生成工具 175 和查询建模器。

[0147] 服务。可使用实体框架 100 构建诸如报告 141、同步 142、Web 服务 143 和业务分析的丰富数据中心服务。

[0148] 编程模式

[0149] ADO.NET 实体框架与 LINQ 一起通过显著减少应用程序代码与数据之间的阻抗失配来增加应用程序开发员的生产率。在本章节中，讨论在逻辑、概念和对象抽象层上的数据访问编程模式的演化。

[0150] 考虑以下基于示例 AdventureWorks（企业工作）数据库的关系模式片段。该数据库由 SContacts（联系人）201、SEmployees（雇员）202，SSalesPersons（销售员）203 以及 SSalesOrders（销售订单）204 表组成，它们遵循如图 2 所示的关系模式。

[0151] SContacts(ContactId(联系人 Id), Name(名字), Email(电子邮件), Phone(电话))

[0152] SEmployees(EmployeeId(雇员 Id), Title(职位), HireDate(雇佣日期))

[0153] SSalesPersons(SalesPersonId(销售员 Id), Bonus(奖金))

[0154] SSalesOrders(SalesOrderId(销售订单 Id), SalesPersonId)

[0155] 考虑获取在某一日期之前雇佣的销售员的名字和雇佣日期的应用程序代码片段（以下示出）。在该代码片段中存在与需要回答的业务问题几乎无关的四个主要的缺点。首先，即使查询可用英语非常简洁地陈述，但 SQL 语句相当冗长，且需要开发员了解规范化的关系模式以制定从 SContacts、SEmployees 和 SSalesPerson 表收集适当列所需的多表联接。另外，对底层数据库模式的任何改变将需要对以下代码片段的相应改变。其次，用户必须定义至数据源的显式连接。第三，由于所返回的结果不是强类型的，对非存在列名的任何引用将仅在执行查询之后才会被发觉。第四，SQL 语句对命令 API 是串属性，且其表述中的任何错误将仅在执行时发现。尽管该代码是使用 ADO.NET2.0 编写的，但代码模式及其缺点适用于任何其它关系数据访问 API，诸如 ODBC、JDBC 或 OLE-DB。

[0156] void EmpsByDate(Date Time date) {

[0157] using (SqlConnection con =

[0158] new SqlConnection(CONN_STRING)) {

[0159] con.Open();

[0160] SqlCommand cmd = con.CreateCommand();

[0161] cmd.CommandText = @"

[0162] SELECT SalesPersonID, FirstName, HireDate

[0163] FROM SSalesPersons sp

```
[0164]     INNER JOIN SEmployees e
[0165]     ON sp.SalesPersonID = e.EmployeeID
[0166]     INNER JOIN SContacts c
[0167]     ON e.EmployeeID = c.ContactID
[0168]     WHERE e.HireDate<@date" ;
[0169]     cmd.Parameters.AddWithValue(" @date" , date) ;
[0170]     DbDataReader r = cmd.ExecuteReader() ;
[0171]     while(r.Read()){
[0172]         Console.WriteLine(" {0:d}:t{1}" ,
[0173]             r[" HireDate" ], r[" FirstName" ]) ;
[0174]     }}}
```

[0175] 如图 3 中所示，示例关系模式可经由 EDM 模式在概念级捕捉。它定义从 SContacts201、SEmployees202 和 SSalesPersons203 表片段抽出的实体类型 ESalesPerson(销售员)302。它也捕捉 EStoreOrder(商店订单)301 与 ESalesOrder(销售订单)303 实体类型之间的继承关系。

[0176] 概念层上的等效程序编写如下：

```
[0177] void EmpsByDate(DateTime date) {
[0178]     using (EntityConnection con =
[0179]         new EntityConnection(CONN_STRING)) {
[0180]         con.Open() ;
[0181]         EntityCommand cmd = con.CreateCommand() ;
[0182]         cmd.CommandText = @"
[0183]         SELECT VALUE sp
[0184]         FROM ESalesPersons sp
[0185]         WHERE sp.HireDate<@date" ;
[0186]         cmd.Parameters.AddWithValue(" date" ,
[0187]             date) ;
[0188]         DbDataReader r = cmd.ExecuteReader(
[0189]             CommandBehavior.SequentialAccess) ;
[0190]         while(r.Read()){
[0191]             Console.WriteLine(" {0:d}:t{1}" ,
[0192]                 r[" HireDate" ]], r[" FirstName" ])
[0193]         }}}
```

[0194] SQL 语句被相当程度地简化了——用户不再必须了解精确的数据库布局。而且，应用程序逻辑可与对底层数据库模式的改变分离。然而，该片段仍是基于串的，仍不能获得编程语言类型检查的好处，且返回弱类型结果。

[0195] 通过在实体周围添加薄对象包装，并使用 C# 的语言集成查询 (LINQ) 扩展，可如下重新编写没有阻抗失配的等效函数：

```
[0196] void EmpsByDate(DateTime date) {
```

```
[0197] using (AdventureWorksDB aw =  
[0198]     new AdventureWorksDB ()) {  
[0199]     var people = from p in aw.SalesPersons  
[0200]         where p.HireDate < date  
[0201]         select p ;  
[0202]     foreach (SalesPerson p in people) {  
[0203]         Console.WriteLine (" {0:d}\t{1} " ,  
[0204]             p.HireDate, p.FirstName) ;  
[0205]     } } }
```

[0206] 该查询是简单的；应用程序（大部分）与对底层数据库模式的改变分离；且查询由 C# 编译器进行完全的类型检查。除查询以外，可与对象交互并对于对象执行常规的创建、读、更新和删除（CRUD）操作。这些的示例将在更新处理章节描述。

[0207] 对象服务

[0208] 对象服务组件是概念（实体）层上的编程 / 表示层。它容纳便于编程语言与基于值的概念层实体之间的交互的若干组件。我们期望对每个编程语言运行时（例如，.NET、Java）存在一个对象服务。如果它被设计成支持 .NET CLR，则使用任何 .NET 语言的程序可与实体框架交互。对象服务由以下主要组件组成：

[0209] `ObjectContext`（对象上下文）类容纳数据库连接、元数据工作空间、对象状态管理器以及对象物化器（materializer）。该类包括允许制定实体 SQL 或 LINQ 句法的查询的对象查询接口 `ObjectQuery<T>`，并返回如 `ObjectReader（对象读取器）<T>` 的强类型对象结果。`ObjectContext` 也展示编程语言层与概念层之间的查询和更新（即，`SaveChanges`（保存改变））对象级接口。对象状态管理器具有三个主要的功能：(a) 高速缓存查询结果，提供身份解析，并管理融合来自重叠的查询结果的对象的策略，(b) 跟踪存储器中的改变，以及 (c) 构造更新处理基础架构的改变列表输入。对象状态管理器在高速缓存中维护每一实体的状态——（从高速缓存）分离、添加、无改变、修改和删除——并跟踪其状态转移。对象物化器在查询和更新期间在来自概念层的实体值与相应的 CLR 对象之间执行变换。

[0210] 映射

[0211] 在一个实施例中，诸如 ADO.NET 实体框架的通用数据访问层的主干可以是在应用程序数据与存储在数据库中的数据之间建立关系的映射。应用程序在对象或概念级查询和更新数据，且这些操作经由映射被转换到存储。存在必须由任一映射解决方案解决的多个技术挑战。构建使用一对一映射以将关系表中的每一行展示为对象的 ORM 是相对直接的，尤其当不需要声明性数据操纵时。然而，随着更复杂的映射、基于集合的操作、性能、多 DBMS 厂商支持和其它要求的介入，自组织解决方案迅速变得难以控制。

[0212] 问题：经由映射更新

[0213] 经由映射访问数据的问题是按照“视图”建模，即客户机层中的对象 / 实体可被视为表行上的丰富视图。然而，如公知的，仅有限的一类视图是可更新的，例如商业数据库系统不允许对包含联接或并的视图中的多个表进行更新。由于视图的更新行为固有地低于规范，即使在相当简单的视图上找到唯一的更新转换也是几乎不可能的。研究

显示，从视图中挑出更新语义是困难的，且可能要求相当的用户专门技能。然而，对映射驱动的数据访问，对视图的每个更新存在良好定义的转换是有利的。

[0214] 而且，在映射驱动的情形中，可更新性要求超出单个视图。例如，操纵顾客和订单实体的业务应用程序针对两个视图有效地执行操作。有时，一致的应用程序状态仅可通过同时更新若干视图来实现。这样的更新的逐个情况的转换可产生更新逻辑的组合激增。将其实现委托给应用程序开发人员是不令人满意的，因为它要求他们手动应付数据访问的最复杂的部分之一。

[0215] ADO.NET 映射方法

[0216] ADO.NET 实体框架支持旨在解决以上挑战的革新性的映射体系结构。它采用以下想法：

[0217] 1. 指定：使用良好定义语义并使大量映射情形能为非专家用户所及的声明性语言来指定映射。

[0218] 2. 编译：映射被编译成驱动运行时引擎中的查询和更新处理的双向视图，被称为查询和更新视图。

[0219] 3. 执行：使用利用物化视图维护（一种健壮的数据库技术）的通用机制进行更新转换。查询转换使用视图展开。

[0220] 新映射体系结构允许以有原则、防过时的方式构建一堆强大的映射驱动技术。而且，它打开了即时实践相关性的有意思的研究方向。以下子章节示出映射的指定和编译。以下在查询处理和更新处理章节中考虑执行。如此处所提供的示例性映射体系结构的其它方面和实施例将在以下题为“其它方面和实施例”的章节中描述。

[0221] 映射的指定

[0222] 使用一组映射片段来指定映射。每一映射片段是 Q 实体 = Q 表形式的约束，其中 Q 实体是对实体模式的查询（应用程序侧），而 Q 表是对数据库模式的查询（存储侧）。映射片段描述实体数据的一部分如何对应于关系数据的一部分。即，映射片段是指定中可独立于其它片段理解的基本单位。

[0223] 为了示出，考虑图 4 中的实例映射情形。图 4 示出了实体模式（左）和数据库模式（右）之间的映射。映射可使用 XML 文件或图形工具定义。实体模式对应于本文中实体数据模型章节中的一个实体模式。在存储侧，存在四个表：SSalesOrders（销售订单）、SSalesPersons（销售员）、SEmployees（雇员）以及 SContacts（联系人）。在实体模式侧，存在两个实体集：ESalesOrder（销售订单）和 ESalesPersons（销售员）以及一个关联集 ESalesPersonOrders（销售员订单）。

[0224] 如图 5 所示，按照对实体模式和关系模式的查询表示映射。

[0225] 在图 5 中，片段 1 表明 ESalesOrders 中确切类型为 ESalesOrder（销售订单）的所有实体的 (Id, AccountNum(帐号)) 值的集合等于从 SSalesOrders 表中检索的 IsOnline（是否在线）为 true（真）的值的 (SalesOrderId(销售订单 Id), AccountNum) 的集合。片段 2 是类似的。片段 3 将关联集 ESalesPersonOrders 映射到 SSalesOrders 表，并表明每一关联实体条目对应于该表中每一行的主键、外键对。片段 4、5 和 6 表明 ESalesPersons 实体集中的实体跨三个表 SSalesPersons、SContacts、SEmployees 划分。

[0226] 双向视图

[0227] 映射被编译成驱动运行时的双向实体 SQL 视图。查询视图按照表来表示实体，而更新视图按照实体来表示表。

[0228] 更新视图可能有点违反直觉，因为它们按照虚拟构造指定持久数据，但如下所示，它们可用于适度地 (elegant) 支持更新。所生成的视图以良好定义的意义“关注”映射，且具有以下属性（注意，表示稍被简化——具体地，持久状态不是完全由虚拟状态定义的）：

[0229] 实体 = 查询视图（表）

[0230] 表 = 更新视图（实体）

[0231] 实体 = 查询视图（更新视图（实体））

[0232] 最后一个条件是往返准则，这确保所有实体数据可按照无损方式从数据库持久保存和重新汇编。包括在实体框架中的映射编译器保证所生成的视图满足往返准则。如果不能从输入映射中产生这样的视图，则它提出错误。

[0233] 图 6 示出了映射编译器为图 5 中的映射生成的双向视图——查询和更新视图。一般，视图可比输入映射复杂得多，因为它们显式指定所需的数据变换。例如，在 QV_1 中，ESalesOrders 实体集从 SSalesOrders 表中构造，使得 ESalesOrder 或 EStoreSalesOrder 中的任何一个根据 IsOnline 标志是否为真而实例化。为了从关系表中重新汇编 ESalesPersons 实体集，需要在 SSalesPersons、SEmployees 和 SContacts 表之间执行联接 (QV_3)。

[0234] 手写满足往返准则的查询和更新视图是棘手的，要求相当的数据库专门知识，从而实体框架的本实施例仅接受由内建映射编译器产生的视图，尽管接受由其它编译器或手动产生的视图在替换实施例中当然是有可能的。

[0235] 映射编译器

[0236] 实体框架包含从 EDM 模式、存储模式和映射（映射工件在本文中的元数据章节中讨论）生成查询和更新视图的映射编译器。这些视图由查询和更新流水线消费。可在设计时或运行时，当针对 EDM 模式执行第一个查询时调用编译器。编译器中所使用的视图生成算法基于用于精确重写的使用视图回答查询 (answering-queries-using-views) 技术。

[0237] 查询处理

[0238] 查询语言

[0239] 在一个实施例中，实体框架可被设计成用于多种查询语言。此处更详细描述实体 SQL 和 LINQ 实施例，但可以理解相同或相似的原理可扩展到其它实施例。

[0240] 实体 SQL

[0241] 实体 SQL 是设计成查询和操纵 EDM 实例的 SQL 的派生物。实体 SQL 按照以下方式扩展标准 SQL。

[0242] 1. 对 EDM 构造（实体、关系、复杂类型等）的本机支持：构造器、成员访问器、类型询问、关系导航、嵌套 / 取消嵌套等。

[0243] 2. 名字空间。实体 SQL 使用名空间作为类型和函数的分组构造（类似于 XQuery 和其它编程语言）。

[0244] 3. 可扩展函数。实体 SQL 不支持内建函数。所有函数（最小、最大、子串等）在名空间中外部定义，且通常从底层存储导入到查询。

[0245] 4. 与 SQL 相比, 对子查询和其它构造更为正交的对待。

[0246] 实体框架例如可支持实体 SQL 作为 EntityClient 提供者层以及对象服务组件中的查询语言。在本文的编程模式章节中示出示例实体 SQL 查询。

[0247] 语言集成查询 (LINQ)

[0248] 语言集成查询即 LINQ 是 .NET 编程语言中对诸如 C# 和 Visual Basic 的主流编程语言引入查询相关构造的革新。查询表达式不由外部工具或语言预处理器处理, 相反是语言本身的第一类表达式。LINQ 允许查询表达式受益于丰富元数据、编译时句法检查、静态类型和之前仅可用于命令性代码的 IntelliSense (智能感知)。LINQ 定义一组通用标准查询操作符, 它们允许遍历、过滤、联接、投影、分类和分组操作以使用任何基于 .NET 的编程语言以直接且声明性的方式表达。诸如 VisualBasic 和 C# 的 .NET 语言也支持查询综合 (querycomprehension)——利用标准查询操作符的语言句法扩展。使用 C# 的 LINQ 的示例查询在本文的编程模式章节中示出。

[0249] 规范命令树

[0250] 在一个实施例中, 规范命令树 (或简称为命令树) 可以是实体框架中所有查询的程序性 (树) 表示。经由实体 SQL 或 LINQ 表示的查询可首先被解析和转化成命令树; 可对命令树执行所有后续处理。实体框架也可允许经由命令树构造 / 编辑 API 来动态构造 (或编辑) 查询。命令树可表示查询、插入、更新、删除和过程调用。命令树由一个或多个表达式组成。表达式仅表示某种计算——实体框架可提供各种表达式, 包括常量、参数、算术运算、关系运算 (投影、过滤、联接等)、函数调用等。最后, 命令树可用作 EntityClient 提供者与底层存储专用提供者之间的查询的通信手段。

[0251] 查询流水线

[0252] 在实体框架的一个实施例中, 查询执行可委托给数据存储。实体框架的查询处理基础架构负责将实体 SQL 或 LINQ 查询分解成一个或多个可由底层存储评估的基本的仅关系的查询以及附加汇编信息, 这些信息可用于将较简单查询的平面结果改造成较丰富的 EDM 结构。

[0253] 实体框架可例如假设存储必须支持类似于 SQL Server 2000 的能力。查询可分解成适合该配置的较简单的平面关系查询。实体框架的其它实施例可允许存储承担大部分的查询处理。

[0254] 典型的查询可如下处理。

[0255] 句法和语义分析。实体 SQL 查询首先使用来自元数据服务组件的信息来解析和语义分析。LINQ 查询被解析和分析, 作为适当语言编译器的一部分。

[0256] 转化成规范命令树。现在将查询转化成命令树, 而不考虑它原始如何表示和确认。

[0257] 映射视图展开。实体框架中的查询的目标为概念 (EDM) 模式。这些查询必须被转换以引用底层数据库表和视图。该过程被称为映射视图展开, 类似于数据库系统中的视图展开机制。EDM 模式和数据库模式之间的映射被编译成查询和更新视图。然后在用户查询中展开查询视图——查询现在的目标是数据库表和视图。

[0258] 结构化类型消除。现在从查询中消除对结构化类型的所有引用, 并将其添加到重新汇编信息 (以引导结果汇编)。这包括对类型构造器、成员访问器、类型询问表达式

的引用。

[0259] 投影修剪。 查询被分析，查询中未被引用的表达式被消除。

[0260] 嵌套拔起。 使查询中的任何嵌套操作（构造嵌套集合）对于仅包含平面关系操作符的子树逼近查询树的根部。一般，嵌套操作被转换成左外联接（或外部应用），且来自随后查询的平面结果然后被重新汇编（见以下的结果汇编）成适当的结果。

[0261] 变换。 应用一组试探变换以简化查询。这些包括过滤下推、应用至联接转化、情况表达式折叠等。在该阶段消除冗余联接（自联接、主键、外键联接）。注意，此处查询处理基础架构不执行任何基于成本的优化。

[0262] 转换成提供者专用命令。 查询（即，命令树）现在被转交给提供者以产生提供者专用命令，可能用提供者的本机 SQL 方言。称此步骤为 SQLGen (SQL 生成)。

[0263] 执行。 执行提供者命令。

[0264] 结果汇编。 来自提供者的结果 (DataReaders (数据读取器)) 然后使用之前收集的汇编信息被改造成适当的形式，且将单个 DataReaders 返回给调用者。

[0265] 物化。 对经由对象服务组件发出的查询，结果然后被物化成适当的编程语言对象。

[0266] SQLGen

[0267] 如在之前章节中所述，查询执行可被委托给底层存储。在这样的实施例中，查询必须首先被转换成适于存储的形式。然而，不同的存储支持 SQL 的不同方言，且实体框架本机支持所有这些方言是不实际的。相反，查询流水线可将命令树形式的查询移交给存储提供者。存储提供者然后可将命令树转换成本机命令。这可通过将命令树转换成提供者的本机 SQL 方言来完成——因此，为该阶段使用术语 SQLGen。然后可执行得到的命令来产生相关结果。

[0268] 更新处理

[0269] 本章节描述如何在示例性 ADO.NET 实体框架中执行更新处理。在一个实施例中，更新处理存在两个阶段：编译时和运行时。在本文提供的双向视图章节中，描述了将映射指定编译成视图表达式的集合的过程。本章节描述如何在运行时利用这些视图表达式以将在对象层执行的对象修改（或 EDM 层的实体 SQLDML 更新）转换成关系层的等效 SQL 更新。

[0270] 经由视图维护的更新

[0271] 在示例性 ADO.NET 映射体系结构中采用的见识之一是物化视图维护算法可用于通过双向视图传播更新。该过程在图 7 中示出。

[0272] 数据库内的表，如图 7 右侧所示，保存持久数据。如图 7 左侧所示，EntityContainer 表示该持久数据的虚拟状态，因为一般仅 EntitySet 中一小部分实体在客户机上物化。目标在于将实体状态上的更新 Δ 实体转换成表的持久状态上的更新 Δ 表。该过程被称为增量视图维护，因为更新是基于表示实体所改变的方面的更新 Δ 实体来执行的。

[0273] 这可使用以下两个步骤来完成：

[0274] 1. 视图维护：

[0275] Δ 表 = Δ 更新视图 (实体, Δ 实体)

[0276] 2. 视图展开：

[0277] Δ 表 = Δ 更新视图 (查询视图 (表) , Δ 实体)

[0278] 在步骤 1 中, 将视图维护算法应用于更新视图。这产生一组 Δ 表达式 Δ 更新视图, 这告诉我们如何从 Δ 实体和实体的快照获取 Δ 表。由于后者未在客户机上完全物化, 在步骤 2 中, 使用视图展开来将 Δ 表达式与查询视图组合。这些步骤一起生成将初始数据库状态和对实体的更新作为输入的表达式, 并计算对数据库的更新。

[0279] 该方法产生用于一次一个对象和基于组的更新 (即, 使用数据操纵语句表达的) 清楚、统一的算法, 并利用健壮数据库技术。在实践中, 步骤 1 通常足以用于更新转换, 因为众多更新不直接取决于当前数据库状态; 在这些情况中, 令 Δ 表 = Δ 更新视图 (Δ 实体) 如果给定 Δ 实体作为对高速缓存的实体的一组一次一个对象修改, 则步骤 1 可通过对经修改的实体直接执行视图维护算法而非计算 Δ 更新视图表达式来进一步优化。

[0280] 转换对象的更新

[0281] 为了示出以上概括的方法, 考虑以下示例, 其中向在公司工作至少 5 年的合格的销售员给予奖金和提升。

```
[0282] using (AdventureWorksDB aw =
[0283] new AdventureWorksDB (...)) {
[0284] // 至少雇佣 5 年的人
[0285] Datetime d = DateTime.Today.AddYears(-5);
[0286] var people = from p in aw.SalesPeople
[0287] where p.HireDate < d
[0288] select p;
[0289] foreach (SalesPerson p in people) {
[0290] if (HRWebService.ReadyForPromotion (p)) {
[0291] p.Bonus+ = 10;
[0292] p.Title = " Senior Sales Representative" ;
[0293] }
[0294] }
[0295] aw.SaveChanges (); // 将改变推送到数据库
[0296] }
```

[0297] AdventureWorksDB (企业工作数据库) 是从被称为 ObjectContext 的普通对象服务类导出的工具生成的类, 它容纳数据库连接、元数据工作空间以及对象高速缓存数据结构并展示 SaveChanges 方法。如在对象服务章节中所述, 对象高速缓存维护实体的列表, 其中每一个实体处于以下状态之一中: (从高速缓存) 分离、添加、无改变、修改和删除。以上代码片段描述了修改 ESalesPerson 对象分别存储在 SEmployees 和 SSalesPersons 表的职位和奖金属性的更新。由对 SaveChanges 方法的调用触发的将对象更新变换成相应的表更新的过程可包括以下四个步骤:

[0298] 改变列表生成。从对象高速缓存为每个实体集创建一个改变列表。更新被表示为删除和插入元素的列表。添加的对象成为插入。删除的对象成为删除。

[0299] 值表达式传播。该步骤取得改变列表和更新视图 (保存在元数据工作空间中),

并使用增量物化视图维护表达式 Δ 更新视图，将对象改变的列表变换成针对底层受影响的表的代数基表插入和删除表达式的序列。对此示例，相关更新视图为图 6 中所示的 UV2 和 UV3。这些视图是简单的投影选择查询，因此应用视图维护规则是直截了当的。获取以下 Δ 更新视图表达式，它们对于插入 (Δ^+) 和删除 (Δ^-) 是相同的：

[0300] Δ SSalesPersons = SELECT p.Id, p.Bonus

[0301] FROM Δ ESalesPersons AS p

[0302] Δ SEmployees = SELECT p.Id, p.Title

[0303] FROM Δ ESalesPersons AS p

[0304] Δ SContacts = SELECT p.Id, p.Name, p.Contact.Email,

[0305] p.Contact.Phone FROM Δ ESalesPersons AS p

[0306] 假定以上示出的循环将实体 $E_{老} = \text{ESalesPersons}(1, 20, "", "Alice", \text{Contact}("a@sales", \text{NULL}))$ 更新为 $E_{新} = \text{ESalesPersons}(1, 30, "Senior\cdots(\text{高级}\cdots)", "Alice", \text{Contact}("a@sales", \text{NULL}))$ 。然后，初始 Δ 对插入是 $\Delta^+ \text{ESalesOrders} = \{E_{新}\}$ ，对删除是 $\Delta^- \text{ESalesOrders} = \{E_{老}\}$ 。得到 $\Delta^+ \text{SSalesPersons} = \{(1, 30)\}$ ， $\Delta^- \text{SSalesPersons} = \{(1, 20)\}$ 。计算得到的对 SSalesPersons 表的插入和删除然后被组合成令 Bonus(奖金) 值为 30 的单个更新。类似计算对 SEmployees 的 Δ 。对 SContacts，得到 $\Delta^+ \text{SContacts} = \Delta^- \text{SContacts}$ ，因此不需要更新。

[0307] 除了计算对受影响的基表的 Δ 以外，该阶段负责 (a) 对必须执行表更新的次序正确定序，考虑引用完整性约束，(b) 在向数据库提交最终更新之前检索所需的存储生成的键以及 (c) 收集用于乐观并发控制的信息。

[0308] SQL DML 或存储过程调用生成。该步骤将插入和删除的 Δ 加上与并发处理有关的附加注释变换成 SQL DML 语句或存储过程调用的序列。在该示例中，为受影响的销售员生成的更新语句是：

[0309] BEGIN TRANSACTION

[0310] UPDATE[dbo].[SSalesPersons]SET[Bonus] = 30

[0311] WHERE[SalesPersonID] = 1

[0312] UPDATE[dbo].[SEmployees]

[0313] SET[Title] = N' Senior Sales Representative'

[0314] WHERE[EmployeeID] =1

[0315] END TRANSACTION

[0316] 高速缓存同步。一旦执行了更新，高速缓存的状态就与数据库的新状态同步。因此，如有需要，执行小型查询处理步骤以将新修改的关系状态变换成其相应的实体和对象状态。

[0317] 元数据

[0318] 元数据子系统类似于数据库目录，且被设计成满足实体框架的设计时和运行时元数据需求。

[0319] 元数据工件

[0320] 元数据工件可例如包括以下：

[0321] 概念模式 (CSDL 文件)：概念模式可在 CSDL(概念模式定义语言)文件中定

义，并包含 EDM 类型（实体类型、关系）和描述应用程序对数据的概念视图的实体集。

[0322] 存储模式 (SSDL 文件)：存储模式信息（表、列、键等）可使用 CSDL 词汇术语来表达。例如，EntitySets 表示表，而属性表示列。这些可在 SSDL（存储模式定义语言）文件中定义。

[0323] C-S 映射指定 (MSL 文件)：在映射指定中，一般在 MSL 文件（映射指定语言）中捕捉概念模式与存储模式之间的映射。该指定由映射编译器用于产生查询和更新视图。

[0324] 提供者清单：提供者清单可提供每一提供者所支持的功能的描述，且可包括以下示例性信息：

[0325] 1. 提供者所支持的原语类型（varchar、int 等），以及它们所对应的 EDM 类型（string、int32 等）。

[0326] 2. 提供者的内建函数（及其签名）。

[0327] 该信息可由实体 SQL 解析器使用作为查询分析的一部分。除这些工件之外，元数据子系统也可跟踪生成的对象类，以及这些和相应的概念实体类型之间的映射。

[0328] 元数据服务体系结构

[0329] 实体框架所消费的元数据可来自不同格式的不同源。元数据子系统可建立在一组统一的低级元数据接口上，这些接口允许元数据运行时独立于不同元数据持久格式/源的细节工作。

[0330] 示例性元数据服务可包括：

[0331] 不同类型的元数据的枚举。

[0332] 按键进行的元数据搜索。

[0333] 元数据浏览 / 导航。

[0334] 瞬态元数据的创建（例如，用于查询处理）。

[0335] 会话无关的元数据高速缓存和重用。

[0336] 元数据子系统包括以下组件。元数据高速缓存，高速缓存从不同源检索到的元数据并向消费者提供检索和操纵元数据的通用 API。由于元数据可用不同形式表示，并存储在不同位置，元数据子系统可有利地支持加载器接口。元数据加载器实现加载器接口，并负责从适当的源（CSDL/SSDL 文件等）加载元数据。元数据工作空间聚集若干元数据来向应用程序提供完整的元数据集。元数据工作空间一般包含关于概念模型、存储模式、对象类以及这些构造之间的映射的信息。

[0337] 工具

[0338] 在一个实施例中，实体框架也可包括设计时工具的集合来增加开发生产率。示例性工具有：

[0339] 模型设计器：应用程序开发的早期步骤之一是定义概念模型。实体框架允许应用程序设计者和分析者按照实体和关系来描述其应用程序的主要概念。模型设计器是允许该概念建模任务交互式执行的工具。设计的工件在可将其状态持久保存在数据库中的元数据组件中直接捕捉。模型设计器也可生成和消费模型描述（经由 CSDL 指定），并可从关系元数据合成 EDM 模型。

[0340] 映射设计器：一旦设计了 EDM 模型之后，开发员可指定如何将概念模型映射到

关系数据库。该任务由映射设计器来促进，该设计器可呈现如图 8 中所示的用户界面。映射设计器帮助开发人员描述在用户界面的左侧呈现的实体模式中的实体和关系如何映射到如图 8 的用户界面的右侧呈现的数据库模式所反映的数据库中的表和列。在图 8 的中间部分呈现的图中的链接使声明性指定为实体 SQL 查询的等式的映射表达式形象化。这些表达式成为生成查询和更新视图的双向映射编译组件的输入。

[0341] 代码生成：EDM 概念模型对众多应用程序是足够的，因为它提供基于 ADO.NET 代码模式的熟悉的交互模型（命令、连接、数据读取器）。然而，众多应用程序偏好与作为强类型对象的数据交互。实体框架包括取 EDM 模型作为输入并为实体类型产生强类型 CLR 类的一组代码生成工具。代码生成工具还可生成强类型对象上下文（例如，AdventureWorksDB），它对由该模型所定义的所有实体和关系集展示强类型集合（例如，ObjectQuery（对象查询）<SalesPerson（销售员）>）。

[0342] 其它方面和实施例

[0343] 映射服务

[0344] 在一个实施例中，诸如图 1 中的 114 的映射组件管理映射的所有方面，并由实体客户机提供者 111 内部使用。映射在逻辑上指定两个潜在不同的类型空间中的构造之间的变换。例如，概念空间中的实体（如以上使用的该术语一样）可按照存储空间中的数据库表来指定，如图 8 中图形所示。

[0345] 规定映射是系统为构造自动确定适当映射的那些映射。非规定映射允许应用程序设计者控制映射的各个方面。映射可具有若干方面。映射的端点（实体、表等）、所映射的属性集、更新行为、诸如延迟加载等运行时影响、对更新的冲突解决行为等仅是这样的方面中的部分列表。

[0346] 在一个实施例中，映射组件 114 可产生映射视图。考虑存储空间与模式空间之间的映射。实体由来自一个或多个表的行组成。查询视图按照存储空间中的表将模式空间中的实体表达为查询。可通过评估查询视图来物化实体。

[0347] 当对一组实体的改变需要回过头来向相应的存储表反映时，改变可通过查询视图以逆向方式传播。这类似于数据库中的视图更新问题——更新传播过程在逻辑上对查询视图的逆执行更新。出于此目的，引入更新视图的概念——这些视图按照实体描述存储表，且可被认为是查询视图的逆。

[0348] 然而，在众多情况中，我们真正感兴趣的是增量式改变。更新 Δ 视图是按照对相应实体集合的改变描述对表的改变的视图（查询）。从而，对实体集合（或应用程序对象）的更新处理包括通过评估更新 Δ 视图然后将这些改变应用到表来计算对表的适当改变。

[0349] 以类似方式，查询 Δ 视图按照对底层表的改变描述对实体集合的改变。无效，更一般地是通知，是可能需要使用查询 Δ 视图的情形。

[0350] 与数据库中的视图一样，表达为查询的映射视图然后可与用户查询组合——导致对映射更通用的处理。类似地，表达为查询的映射 Δ 视图允许处理更新的更一般且适度的方法。

[0351] 在一个实施例中，映射视图的能力可能受到限制。在映射视图中使用的查询构造可能仅是实体框架所支持的所有查询构造的子集。这允许更简单和更高效的映射表达

式——尤其是在 Δ 表达式的情况中。

[0352] Δ 视图可使用代数改变计算方案在映射组件 114 中计算以从更新（和查询）视图产生更新（和查询） Δ 视图。稍后讨论代数改变计算方案的其它方面。

[0353] 更新 Δ 视图通过将计算应用程序所作的实体改变自动转换成数据库中的存储级更新来允许实体框架支持更新。然而，在众多情况中，对于性能和 / 或数据完整性，映射必须扩充以附加信息。

[0354] 在某些情况中，将实体上的更新直接映射到其部分或全部底层存储表可能不是合乎需要的。在这样的情况中，更新必须通过存储过程以允许数据确认以及维护信任边界。映射允许指定存储过程以对于实体处理更新和查询。

[0355] 映射也可在对象服务 131 中提供对乐观并发控制的支持。具体地，实体的属性可被标记为并发控制字段，诸如时戳或版本字段，且对这些对象的改变仅当存储处的并发控制字段的值与实体中的相同才会成功。注意，这两个乐观并发控制字段仅在应用程序对象层相关，而不在存储专用层 120 相关。

[0356] 在一个实施例中，应用程序设计器可使用映射规范语言 (MSL) 来描述映射的各个方面。典型的映射指定包含以下区段中的一个或多个。

[0357] 1. 数据区可包含对类、表和 / 或 EDM 类型的描述。这些描述可描述现有的类 / 表 / 类型，或可用于生成这样的实例。若干生成的值、约束、主键等被指定，作为该区段的一部分。

[0358] 2. 映射区段描述类型空间之间的实际映射。例如，按照来自表（或表集）的一个或多个列指定 EDM 实体的每一属性。

[0359] 3. 运行时区可指定控制执行的各个调节器，例如乐观并发控制参数和取数策略。

[0360] 映射编译器

[0361] 在一个实施例中，域建模工具映射组件 172 可包括映射编译器，该编译器将映射指定编译成查询视图、更新视图以及相应的 Δ 视图。图 9 示出编译 MSL 来生成更新和查询视图。

[0362] 编译流水线执行以下步骤：

[0363] 1. 从 API900 调用的视图生成器 902 转换对象 \leftrightarrow 实体映射信息 901（通过 MSL 指定），并在对象 \leftrightarrow 实体空间中产生查询视图、更新视图和相应的（查询和更新） Δ 表达式 904。该信息可被放置在元数据存储 908 中。

[0364] 2. 视图生成器 906 转换实体 \leftrightarrow 存储映射信息 903（通过 MSL 指定），并在实体 \leftrightarrow 存储空间中产生查询视图、更新视图和相应的（查询和更新） Δ 表达式 907。该信息可被放置在元数据存储 908 中。

[0365] 3. 依赖性分析 909 组件检查由视图生成器 906 产生的视图，并对不违反引用完整性和其它这样的约束的更新确定一致的依赖次序 910。该信息可被放置在元数据存储 908 中。

[0366] 4. 视图、 Δ 表达式和依赖性次序 908 然后被传递到元数据服务组件（图 1 中的 112）上。

[0367] 更新处理

[0368] 本章节描述更新处理流水线。在一个实施例中，实体框架可支持两种更新。单对象改变是在导航对象图的同时对个别对象作出的改变。对单对象改变，系统跟踪已经在当前事务中创建、更新和删除的对象。这仅在对象层可用。基于查询的改变是通过发出基于对象查询的更新 / 删除语句来执行的改变，如关系数据库中为更新表所做的。诸如图 1 中的 131 的对象提供者可被配置成支持单对象改变，但不支持基于查询的改变。另一方面，实体客户机提供者 111 可支持基于查询的改变但不支持单对象改变。

[0369] 图 10 提供了对一个示例性实施例中的更新处理的说明。在图 10 中，应用程序层 1000 处的应用程序的用户 1001 可保存由这样的应用程序操纵的数据改变 1002。在对象提供层 1010 中，在 1011 编译改变列表。在改变列表上执行改变分组 1012。约束处理 1013 可产生约束信息以及被保存到元数据存储 1017 的依赖性模型 1022。在 1014 执行扩展操作。在 1015 生成并发控制表达式，且并发模型 1023 被保存到元数据存储 1017。对象到实体转化器 1016 可将对象到实体 Δ 表达式 1024 保存到元数据存储 1017。

[0370] 将实体表达式树 1018 向下传给 EDM 提供者层 1030。选择性更新拆分器 1031 可选择某些更新，并按需对其拆分。EDM 存储转化器 1032 可将实体到存储 Δ 表达式 1033 保存到元数据存储 1036。查询视图展开组件 1035 可将查询映射视图 1035 保存到元数据存储 1036。执行实体到存储补偿 1037，且将存储表达式树 1038 传递给存储提供者层 1040。

[0371] 在存储提供者层 1040，简化器组件 1041 可首先操作，接着是 SQL 生成组件 1042，它生成要在数据库 1044 上执行的 SQL 更新 1043。可将任何更新结果传递给 EDM 提供者层 1030 中的组件 1039 以处理服务器生成的值。组件 1039 可将结果向上传给对象提供者层 1021 中的相似组件。最后，任何结果或更新确认 1003 被返回给应用程序层 1000。

[0372] 如上所述，作为映射编译的一部分生成更新 Δ 视图。这些视图在更新过程中使用以标识对存储处的表的改变。

[0373] 对存储处的一组相关表，实体框架可按照某一次序有利地应用更新。例如，外键约束的存在可能需要以特定顺序应用改变。依赖性分析阶段（映射编译的一部分）标识可在编译时计算的任何依赖性定序要求。

[0374] 在某些情况中，静态依赖性分析技术可能不是足够的，例如对于循环引用完整性约束（或自引用完整性约束）。实体框架采用乐观方法，并允许这样的更新通过。在运行时，如果检测到循环，则提出异常。

[0375] 如图 10 中所示，用于应用程序层 1000 处的基于实例的更新的更新处理流水线具有以下步骤：

[0376] 改变分组 1012：根据来自改变跟踪器的不同对象集合来对改变进行分组，例如将用于集合 Person（个人）的所有改变分组到该集合的插入、删除和更新集中。

[0377] 约束处理 1013：该步骤执行对在值层上不执行任何业务逻辑的事实进行补偿的任何操作——实质上，它允许对象层扩展改变集。在此执行级联删除补偿和依赖性定序（对于各个 EDM 约束）。

[0378] 扩展操作执行 1014：执行额外（例如，删除）操作，使得相应的商业逻辑可运行。

[0379] 并发控制表达式生成器 1015：为了检测修改的对象是否过时，可生成检查时戳列或如在映射元数据中指定的一组列的表达式。

[0380] 对象到EDM转化 1016：按照插入、删除和更新对象集指定的改变列表现在使用存储在元数据存储 1017 中的映射 Δ 表达式来转化，这些表达式是在参考图 9 描述的映射编译之后存储的。在该步骤之后，改变可用作仅按照 EDM 实体表达的表达式树 1018。

[0381] 来自步骤 1018 的表达式树然后被传给 EDM 提供者层 1030 中的 EDM 提供者。在 EDM 提供者中，处理表达式树，并将改变提交给存储。注意到，表达式树 1018 也可按照另一方式产生——当应用程序直接针对 EDM 提供者编程时，它可针对其执行 DML 语句。这样的 DML 语句首先由 EDM 提供者转化成 EDM 表达式树 1018。从 DML 语句或从应用程序层 1000 获取的表达式树然后按照以下方式处理：

[0382] 选择性更新拆分器 1031：在此步骤，有些更新被拆分成插入和删除。一般按原样将更新传播到较低层。然而，在某些情况中，可能不可能执行这样的更新，因为未对该情况开发 Δ 表达式规则，或者因为正确的转换实际上导致对基表的插入和 / 删除。

[0383] EDM 到存储转化 1032：使用来自适当映射的 Δ 表达式来将 EDM 级表达式树 1018 转换到存储空间中。

[0384] 查询映射视图展开 1034：表达式树 1018 可包含某些 EDM 级概念。为了消除它们，使用查询映射视图 1035 展开表达式树以获取仅按照存储级概念的树 1038。树 1038 可任选地由实体到存储补偿组件 1037 处理。

[0385] 现将按照存储空间的表达式树 1038 给予存储提供者层 1040 中的存储提供者，它们执行以下步骤：

[0386] 简化 1041：通过使用逻辑表达式转换规则来简化表达式树。

[0387] SQL 生成 1042：给定表达式树，存储提供者从表达式树 1038 生成实际的 SQL 1043。

[0388] SQL 执行 1044：在数据库上执行实际的改变。

[0389] 服务器生成的值：由服务器生成的值被返回给 EDP 层 1030。提供者 1044 将服务器生成的值传递给层 1030 中的组件 1039，后者使用映射将其转换成 EDM 概念。应用程序层 1000 拾取这些改变 1003，并将其传播到对象级概念以在该层利用的各个应用程序和对象中安装。

[0390] 在众多情况中，例如由于数据库管理员 (DBA) 策略，存储表不能直接更新。对表的更新仅可经由存储过程，因此可执行某些确认检查。在这样的情形中，映射组件可将对象改变转换成对这些存储过程的调用，而非执行“原始”的插入、删除和更新 SQL 语句。在其它情况中，“存储”过程可在 EDP 1010 或应用程序层 1000 处指定——在这样的情况中，映射组件必须将经修改的对象转换到 EDM 空间，然后调用适当的过程。

[0391] 为了允许这些情形，MSL 允许存储过程被指定为映射的一部分；另外，MSL 也支持指定如何将各个数据库列映射到存储过程的参数的机制。

[0392] EDP 层 1010 支持乐观并发控制。当 CDP 向存储发送一组改变时，所改变的行可能已经被另一事务改变。CDP 必须支持让用户能够检测这样的冲突然后解决这样的冲突的方式。

[0393] MSL 支持简单机制——时戳、版本号、所改变的列——用于冲突检测。当检测

到冲突时，提出异常，且发生冲突的对象（或 EDM 实体）可由应用程序进行冲突解决。

[0394] 示例性映射要求

[0395] 映射基础架构可有利地提供将各个操作从应用程序空间转换成关系空间（例如由开发人员编写的对象查询被转换到关系（存储）空间）的能力。这些转换应是高效的，而无需对数据进行过多的复制。映射器可提供转换用于以下示例性操作：

[0396] 1. 查询：对象查询需要被转化到后端关系域，且从数据库获取的元组需要被转换成应用程序对象。注意这些查询可能是基于组的查询（例如，CSQL 或 C# 序列）或是基于导航的（例如，仅跟随引用）。

[0397] 2. 更新：应用程序对其对象所作的改变需要被传回数据库。再一次，对于对象所作的改变可能是基于组的或对个别对象进行的。考虑的另一个方面是正在修改的对象是被完全加载到存储器中还是被部分加载（例如，移交 (handoff) 对象的集合可能目前不在存储器中）。对于对部分加载的对象的更新，其中不要求这些对象被完全加载到存储器中的设计可能是更优选的。

[0398] 3. 无效和通知：运行在中间层或客户机层的应用程序可能想要在一些对象在后端改变时得到通知。因此，OR 映射组件应将对象级的注册转换到关系空间；类似地，当客户机接收到关于所修改的元组的消息时，OR 映射器必须将这些通知转换成对象改变。注意，WinFS 经由其看守人机制支持这样的“通知”——然而，在这种情况下，映射是规定的，而实体框架应支持对非规定的映射的看守人。

[0399] 4. 也需要类似于通知的机制来从运行在中间层或客户机层中的实体框架进程中使过时对象无效——如果实体框架提供对乐观并发控制的支持以处理发生冲突的读/写，则应用程序可确保在实体框架高速缓存的数据是相当新鲜的（因此，无需因对于对象的读/写而异常中止事务）；否则，它们可对老数据作出决策和/或使其事务在稍后异常中止。因此，与通知相同，OR 映射器可能必须将来自数据库服务器的“无效”消息转换成对象无效。

[0400] 5. 备份/还原/同步：实体的备份和镜像是可并入某些实施例中的两个特征。对这些特征的要求可简单地转换成从 OR 映射器的角度对实体的专门查询；否则，可提供对这样的操作的特殊支持。类似地，同步可能需要对将对象改变、冲突等从 OR 映射引擎转换到存储及其反向转换的支持。

[0401] 6. 参与并发控制：OR 映射器可有利地支持乐观并发控制可由应用程序使用的不同方式，例如使用时戳值、某些特定的字段组等。OR 映射器应将诸如时戳属性等并发控制信息向/自对象空间和自/向关系空间转换。OR 映射器甚至可提供对悲观并发控制的支持（例如，如 Hibernate）。

[0402] 7. 运行时错误报告。在此处所示的示例性实施例中，运行时错误一般在存储级发生。这些错误可被转换到应用程序级。OR 映射器可用于便于这些错误的转换。

[0403] 映射情形

[0404] 在讨论实体框架可能支持的示例性开发情形之前，示出 OR 映射器的各个逻辑部分。在一个实施例中，如图 11 中所示，OR 映射中有五个部分。

[0405] 1. 对象/类/XML（也称为应用程序空间）1101：开发人员用选择的语言指定类和对象——最终，这些类被编译成 CLR 汇编，并可通过反射 API 访问。这些类包括持久和

非持久成员；而且，在该部分中可包括语言专用细节。

[0406] 2. 实体数据模型模式（也称为概念空间）1102：EDM 空间由开发人员用于对数据建模。如上所述，数据模型的指定按照 EDM 类型、实体之间的关系经由关联、继承等来完成。

[0407] 3. 数据库模式（也称为存储空间）1103：在该空间中，开发人员指定表如何布局；也在此处指定诸如外键和主键约束的约束。该空间中的指定可利用厂商专用特征，例如嵌套表、UDT 等。

[0408] 4. 对象-EDM 映射 1104：该映射指定各个对象和 EDM 实体如何彼此相关，例如数组可被映射到一对多关联。注意该映射不必是平凡 / 恒等的，例如多个类可映射到给定 EDM 类型反之亦然。注意，在这些映射中可能有或可能没有冗余 / 非规范化（当然，对于非规范化，有可能会碰到保持对象 / 实体一致的问题）。

[0409] 5. EDM- 存储映射 1105：该映射指定 EDM 实体和类型如何与数据库中的不同表相关，例如此处可使用不同的继承映射策略。

[0410] 开发人员可指定一个或多个空间 1101、1102 或 1103 及其之间的一个或多个映射之间的相应映射。如果遗失了任何数据空间，则开发人员可给出如何生成该空间的提示，或可期望 EDP 采用相应的规定的映射来自动生成这些空间。例如，如果开发人员指定现有的类、表及其之间的映射，则 EDP 生成内部 EDM 模式和相应的对象-EDM 和 EDM- 存储映射。当然，在最普通的情况中，开发人员可具有完全的控制，并在这三个空间以及两个映射中指定数据模型。下表示出 EDP 中所支持的不同情形。这是对其中开发人员是否可指定对象、EDM 实体、表的情况的穷举列表。

[0411]

情形	对象被指定?	EDM 被指定?	表被指定?	映射被指定
(A)	是			
(B)		是		
(C)			是 是	
(D)	是	是		OE(对象-实体)
(E)	是		是	OS(对象-存储)
(F)		是	是	ES(实体-存储)
(G)	是	是	Y	OE, ES

[0412] 取决于 EDP 想要支持的以上情形，必须提供工具来产生未指定的数据空间和映射（按照规定的方式或基于所提供的提示）。内部 OR 映射引擎假定，映射的所有 5 个部分（对象、EDM 指定、表、OE 映射、ES 映射）均是可用的。因此，映射设计应支持最普通的情况，即上表中的 (G)。

[0413] 映射规范语言

[0414] 从开发员的角度，OR 映射器的一个“可见”部分是映射规范语言即 MSL——开发员使用该语言指定映射的各个部分如何彼此连系。运行时控制（例如，延迟取、乐

观并发控制问题) 也使用 MSL 指定。

[0415] 将映射分成三个不同的概念——每一概念解决映射过程的不同方面。注意并未规定这些指定是存储在单个文件、多个文件中还是通过外部储存库(例如,用于数据指定)来指定的。

[0416] 1. 数据指定: 在该区中, 开发人员可指定类描述、表描述和 EDM 描述。这些描述可提供为对生成目的的指定, 或它们可以是已经存在的表/对象的指定。

[0417] 对象和表指定可按照我们的格式描述, 或者可使用某种导入工具从外部元数据储存库导入它们。

[0418] 注意, 服务器生成值、约束、主键等的指定在该部分中完成(例如, 在 EDM 指定中, 约束被指定为类型指定的一部分)。

[0419] 2. 映射指定: 开发人员指定各个对象、EDM 类型和表的映射。允许开发人员指定对象-EDM、EDM-存储和对象-存储映射。该部分试图使数据指定具有最小冗余。

[0420] 在所有三个映射情况(OS、ES 和 OE)中, 为每一类或者在顶级“直接”或者在另一类内“间接”指定映射。在每一映射中, 字段/属性被映射到另一字段、字段的标量函数、组件或集。为了允许更新, 这些映射需要是双向的, 即对象与存储空间的往来应不会丢失任何信息, 也允许非双向映射, 使得对象是只读的。

[0421] 对象-EDM 映射: 在一个实施例中, 按照 EDM 类型为每个对象指定映射。

[0422] EDM-存储映射: 在一个实施例中, 按照表为每个实体指定映射。

[0423] 对象-存储映射: 在一个实施例中, 按照表为每个对象指定映射。

[0424] 3. 运行时指定: 在一个实施例中, 允许开发人员指定控制执行的各个调节器, 例如乐观并发控制参数和取数策略。

[0425] 此处是 OPerson(个人)对象包含一组地址的情况的映射文件的示例。该对象被映射到 EDM 实体类型, 且该集被映射到内联集类型。该数据被存储到两个表中——一个用于个人, 另一个用于地址。如前所述, 开发人员不必指定所有对象、EDM 类型和表——仅示出上表中的情况(G)。指定未被假定描述任何具体的句法, 它们旨在示出和允许围绕此处公开的概念设计系统。

[0426] 对象指定

[0427]

<pre>ObjectSpec OPerson{ string name ; Set<Address>addrs ; }</pre>	<pre>ObjectSpec OAddress{ string state ; }</pre>
--	--

[0428] EDM 指定

[0429] 指定一个实体类型 CPerson(个人)和内联类型 CAddress(地址), 使得每一 CPerson 具有 CAddress 项目的集合。

[0430]

EDMSpec Entity CPerson{ string name ; int pid ; Set<CAddress>addrs ; Key:{pid} }	EDMSpec Inline Type CAddress{ string state ; int aid ; }
---	---

[0431] 存储指定

[0432] 指定两个表类型 SPerson(个人) 和 SAddress(地址) 及其键 (tpid 和 taid)。

[0433]

TableSpec SPerson{ int pid ; nvarchar (10) name ; Key:{pid} }	TableSpec SAddress{ int aid ; string state ; Key:{aid} }
---	--

[0434] 对象-CDM 映射

[0435] 以下对 OPerson 的映射表明对象类型 OPerson 被映射到实体 CPerson。之后的列表指定 OPerson 的每一字段如何映射——名字 (name) 被映射到名字，而地址 (addr) 集合被映射到地址集合。

[0436]

Object-CDM OPerson { EntitySpec = CPerson name ↔ name addrs ↔ addrs }	Object-CDM OAddress { InlineTypeSpec = CAddress state ↔ state }
--	--

[0437] EDM- 存储映射：

[0438] EDM 实体类型 CPerson 及其键和名字 cname 属性被映射到表类型 SPerson。内联类型 CAddress 以简单的方式映射到 SAddress。注意表 SAddress 可将外键存储到 SPerson；该约束可能已经在表的数据模型指定中指定而非在映射中指定。

[0439]

EDM-Store CPerson { TableSpec = SPerson name ↔ name pid ↔ pid }	EDM-Store CAddress { TableSpec = SAddress aid ↔ aid state ↔ state }	EDM-Store CPerson_Address { TableSpec = SAddress aid ↔ aid pid ↔ pid }
--	--	---

[0440] 运行时指定：

[0441] 开发人员可能想要指定，对 OPerson 的乐观并发控制在 pid 和 name 字段上完成。对 OAddress(地址)，他 / 她可对 state(状态) 字段指定并发控制。

[0442]

<pre> RuntimeSpec OPerson{ Concurrency fields:{pid, name} } </pre>	<pre> RuntimeSpec OAddress Concurrency fields:{state} } </pre>
--	--

[0443] 映射设计概观

[0444] 大多数 OR 映射技术，诸如 Hibernate 和 ObjectSpaces 具有一个重要的缺点——它们以相对自组织的方式处理更新。当对象改变需要被推送回服务器时，这些系统所使用的机制在逐个情况的基础上处理更新，从而限制了系统的可扩展性。当支持更多的映射情况中，更新流水线变得复杂且难以为更新组成映射。随着系统的演化，系统的这一部分变得相当难以在确保其正确的情况下进行改变。

[0445] 为了避免这样的问题，使用其中使用两种类型的“映射视图”（其一帮助转换查询，另一个帮助转换更新）来执行映射过程的新颖的方法。如图 12 中所示，当 MSL 指定 1201 由 EDP 处理时，它内部生成两个视图 1202 和 1203 用于核心映射引擎的执行。如稍后可见，通过按照这些视图对映射建模，能够利用关系数据库中对物化视图技术的现有知识——具体地，利用增量式视图维护技术用于以正确、适度且可扩展的方式对更新建模。现在讨论这两种类型的映射视图。

[0446] 使用查询映射视图即 QM 视图的概念将表数据映射到对象，使用更新映射视图即 UM 视图将对象改变映射到表更新。这些视图根据其构造的（主要）理由而命名。查询视图将对象查询转换成关系查询，并将传入的关系元组转化成对象。因此，对 EDM- 存储映射，每一查询视图示出如何从各个表构造 EDM 类型。例如，如果根据两个表 T_P 和 T_A 的联接构造 Person 实体，则按照这两个表之间的联接来指定 Person。当对 Person 集合请求查询时，Person 的 QM 视图用按照 T_P 和 T_A 的表达式代入 Person，该表达式然后生成适当的 SQL。然后在数据库执行查询，当从服务器接收到回复时，QM 视图从所返回的元组物化对象。

[0447] 为了处理对象更新，可想象通过 QM 视图推送改变，并利用为关系数据库开发的“视图更新”技术。然而，可更新视图具有多个限制，例如 SQL Sever 不允许多个基表通过视图更新来修改。因此，代替限制 EDP 中所允许的映射的类型，本发明的实施例利用物化视图技术中具有少得多的限制的另一面——视图维护。

[0448] 按照 EDM 类型指定更新映射视图即 UM 视图用于表达系统中的每一表，即在某种意义上，UM 视图是 QM 视图的逆。EDM- 存储边界上的表类型的 UM 视图提供了使用不同 EDM 类型来构造该表类型的列的方式。因此，如果指定 Person 对象类型映射到表类型 T_P 和 T_A，则不仅按照 T_P 和 T_A 为 Person 类型生成 QM 视图，也生成指定给定 Person 对象类型可如何构造 T_P 的行的 UM 视图（对 T_A 是类似的）。如果事务创建、删除或更新某些 Person 对象，则可使用更新视图将这样的改变从对象转换成对 T_P 和 T_A 的 SQL 插入、更新和删除语句——UM 视图有助于执行这些改变，因为它们告诉我们如何从对象获取关系元组（经由 CDM 类型）。图 13 和 14 在高级示出如何在查询和更新转换中使用 QM 视图和 UM 视图。

[0449] 给定用于将表建模成对象上的视图的这种方法，将对于对象的更新传播回表的过程类似于其中对象是“基本关系”而表是“视图”的视图维护问题。存在解决视图维护问题的大量数据库文献，可为我们的用途使用它们。例如，存在示出如何将对基本关系的增量式改变转换成对视图的增量式改变的大量作品。使用代数方法来确定对视图执行增量式更新所需的表达式——将这些表达式称为 Δ 表达式。与过程方法相对，使用代数方法对增量式视图维护是适当的，因为它更顺应优化和更新简化。

[0450] 一般，在 EDP 核心引擎中使用映射视图的优点包括：

[0451] 1. 视图提供用于表达对象和关系之间的映射的大量能力和灵活性。可以 OR 映射引擎的核心部分中的受限视图表达式语言来开始。当时间和资源许可时，视图的能力可用于适度地演化系统。

[0452] 2. 已知视图是由查询、更新和视图本身适度组成。尤其对于更新，可组成性对早先尝试的某些 OR 映射方法是成问题的。通过采用基于视图的技术，可不必对此费心。

[0453] 使用视图的概念允许我们利用数据库文献中的大量作品。

[0454] 更新的体系结构层叠

[0455] 在实现本发明的各方面时考虑的一个重要问题是，用其表达查询和更新映射视图的映射视图语言（即 MVL）的能力是什么。它足以捕捉对象与 EDM 之间所有的非规定映射以及 EDM 与存储之间的映射。然而，对本机支持所有非关系 CLR 和 EDM 概念的 MVL，需要为所有这样的构造设计 Δ 表达式或增量式视图更新规则。具体地，一个示例性实施例可能要求以下非关系代数运算符 / 概念的更新规则：

[0456] 复杂类型——对于对象、元组构造器、展平、复杂常量等一部分的访问。

[0457] 集合——嵌套和取消嵌套、组构造 / 展平、交叉应用等。

[0458] 数组 / 列表——元素的次序不是关系构造；显然，有序列表的代数是相当复杂的。

[0459] CLR/C# 中需要建模的其它 EDM 构造和对象构造。

[0460] 有可能为这些构造的增量式更新开发 Δ 表达式。用 MVL 本机支持大集合构造的主要问题在于，它可能会在相当程度上使核心引擎复杂化。在一个实施例中，更合乎需要的方法可以是使系统层叠，使得“核心映射引擎”处理简单 MVL 然后将非关系构造层叠到该核心的顶部。现在讨论这一设计。

[0461] 我们用于 OR 映射的方法通过“层叠”来解决以上问题——在编译时，首先将对象、EDM 和数据库空间中的每一非关系构造（WinFS 支持嵌套、UDT 等）按照规定方式转换成相应的关系构造，然后执行关系构造之间所需的非规定转换。将该方法称为层叠视图映射方法。例如，如果类 CPerson 包含地址的集合，首先按照一对多关联将该集合转换成关系构造，然后对关系空间中的表执行所需的非规定转换。

[0462] MVL 分解

[0463] MVL 被分成两层——按照关系处理实际的非规定映射的一层，以及将非关系构造转换成关系方面的规定转换。前一语言被称为 R-MVL（关系 MVL），而相应的映射被称为 R-MVL 映射；类似的，后一（更强大）语言被称为 N-MVL（非关系 MVL）而映射被称为 N-MVL 映射。

[0464] 在一个实施例中，通过构造设计使得所有的非关系构造被推送到查询和更新流

水线的结尾来提供映射。例如，对象物化可涉及构造对象、数组、指针等——这样的“操作符”被推送到查询流水线的顶部。类似地，当对于对象发生更新时，在流水线的开头转换对于非关系对象（例如，嵌套集合、数组）的改变然后通过更新流水线传播这些改变。在如 WinFS 的系统中，需要在更新流水线的结尾转换成 UDT。

[0465] 通过将非规定映射限制成 R-MVL，现在具有了需要增量式视图维护规则的一小组关系构造——已经为关系数据库开发了这样的规则。将在 R-MVL 中所允许的简化的构造 / 模式称为关系表达的模式即 RES。因此，当在对象域中需要（例如）支持某些非关系构造时，得到相应的 RES 构造以及对象与 RES 构造之间的规定转换，例如将对象集合转换为 RES 空间中的一对多关联。而且，为了传播对非关系构造 N 的更新，要提出将插入、删除和更新从 N 转换到 N 相应的 RES 构造的 Δ 表达式。注意，这些 Δ 表达式是规定的，且由我们在设计时生成，例如知道如何将对集合的改变推送到一对多关联上。使用关系数据库的增量式视图维护规则自动生成实际非规定映射的 Δ 表达式。该层叠方法不仅移除了为过多的非关系构造提出通用的增量式视图维护规则的要求，而且也简化了内部更新流水线。

[0466] 注意到，我们的层叠映射方法对于通知流水线也有类似的好处——当从服务器接收到对于元组的改变时，需要将其转换成对于对象的增量式改变。除了需要使用查询映射视图来传播这些改变，即生成 QM 视图的 Δ 表达式以外，这是与更新流水线相同的要求。

[0467] 除了简化更新和通知流水线，层叠 MVL 有一个重要的优点——它允许“上层语言”（对象、EDM、数据库）进行演化，而不会对核心映射引擎造成显著影响。例如，如果将新概念添加到 EDM，所有需要做的是提出将其转化成该构造的相应 RES 的规定方式。类似地，如果在 SQL Server 中存在非关系概念（例如，UDT、嵌套），可按照规定方式将这些构造转换成 MVL，且对 MVL 和核心引擎具有最小的影响。注意到，RES 存储和存储表之间的转换不必是恒等的转换。例如，在支持 UDT、嵌套等的后端系统（诸如 WinFS 后端）中，转换类似于规定的对象关系。

[0468] 图 15 示出了对映射视图的编译时和运行时处理。给定如 1501、1502 和 1503 示出的采用 MSL 的数据模型和映射指定，首先为非关系构造 1511、1522 和 1523 生成相应的 RES1521、1522 和 1523 以及这些构造与 RES 之间规定的转换，即 N-MVL 映射。然后为开发人员所请求的非规定映射生成查询和更新映射视图，采用 R-MVL 的对象-EDM 以及采用 R-MVL 的对象-EDM——注意，这些映射视图使用 R-MVL 语言对 RES 进行操作。此时，为查询和更新映射视图生成 Δ 表达式（视图维护表达式）——已经为关系数据库开发了这样的规则。注意，出于通知的目的，需要 QM 视图的 Δ 表达式。对 N-MVL 映射，在设计时由我们确定 Δ 表达式，因为这些映射是规定的，例如当将 Address 集合映射到一对多关联时，也设计相应的视图维护表达式。

[0469] 给定以上视图和转换（N-MVL 和 R-MVL），可将其合成以获得可按照存储 1533 中的表表达对象 1531 的查询映射视图以及可按照对象 1531 表达表 1533 的更新映射视图。如图所示，可选择保留映射视图，使得 1532 中的 EDM 实体并未从运行时的映射中完全消除——保存这些视图的一个可能的理由是启用利用 EDM 约束的某种查询优化。当然，这不意味着在运行时实际存储 EDM 实体。

[0470] 图 16 示出了不同的组件如何实现上述的视图编译过程。应用程序调用 API1600。视图生成器 1601、1603 负责三个功能：将非关系构造转换成 RES 构造、生成查询/更新视图以及生成 Δ 表达式以便传播更新和通知。它们在实现这些功能时可使用元数据 1602。OE 视图合成器 1605 取得对象和 EDM 信息，并合成它，使得我们有了按照 EDM 类型的对象代数表达式；类似地，ES 视图合成器 1606 按照表产生 EDM 类型的代数表达式。在 OS 视图合成器 1607 中进一步合成这些视图，并在元数据存储 1608 中得到单个视图集。如上所述，保存两个视图集用于可能的优化机会。最后，依赖性分析组件 1604 也可对 ES 视图生成器输出进行操作以向元数据存储 1608 提供依赖次序。

[0471] 映射编译概述

[0472] 总而言之，对类、EDM 类型或表的每一指定 M，生成相应的 RES 以及 M 与相应的 RES 之间的规定转换。因此，如图 15 所示，生成以下内容：

[0473] 1. 对应于 M 的 RES——表示为 RES-CDM(M)、RES-对象(M)或 RES-存储(M)。

[0474] 2. 按照 RES 关系表达每一指定 M 的规定的转换

[0475] 3. 按照 M 表达这样的 RES 关系的规定的转换

[0476] 4. 查询映射视图：有两个这样的视图——OE QM 视图按照 EDM 类型表达对象，ES QM 视图按照存储(表)表达 EDM 类型

[0477] 5. 更新映射视图：有两个这样的视图——OE UM 视图按照对象表达 EDM 类型，ES UM 视图按照 EDM 类型表达存储表

[0478] 6. 对更新的增量式维护，也对 UM 视图生成 Δ 表达式。

[0479] 在合成这些视图之后，以四个映射结束。这些映射被存储在元数据存储 1608 中，并被统称为经编译的映射视图：

[0480] 查询映射：按照 CDM/表来表达对象/CDM。

[0481] 更新映射：按照 CDM/对象来表达表/CDM。

[0482] 更新 Δ 表达式：按照 CDM/对象的 Δ 来表达表/CDM 的 Δ 。

[0483] 通知 Δ 表达式：按照 CDM/表的 Δ 来表达对象/CDM 的 Δ 。

[0484] 依赖次序：对不同关系执行各个插入、删除、更新操作的必须次序——该次序确保数据库约束在更新过程期间不会被违反。

[0485] 集合示例

[0486] 现在简要示出所考虑的 Person 示例的规定转换和非规定映射。呈现查询和更新映射视图两者——以下进一步讨论相应的视图维护表达式。

[0487] RES

[0488] 将 OPerson 转换成仅反映 name 和 pid 的 RES 构造 R_OPerson；类似地，将 OAddress 转换成 R_OAddress。为了转换地址的集合，使用一对多关联 R_OPerson_Address。类似地，对于 EDM 构造也一样。表 (R_SPerson, R_SAddress) 的 RES 是对 Sperson 和 Saddress 的恒等映射。这些 RES 是：

[0489]

R_OPerson(pid, name)	R_CPerson(pid, name)	R_SPerson(pid, name)
R_OAddress(aid, state)	R_CAddress(aid, state)	R_SAddress(pid, aid, state)
R_OPerson_Address(pid, aid)	R_CPerson_Address(pid, aid)	

[0490] 查询映射视图

[0491] 示出对象 - 存储映射（在对象 - EDM 和 EDM - 存储映射上组成）。

[0492] RES 空间中的非规定视图

[0493] 对象与 EDM 空间之间的映射实质上是恒等的。所有三个视图 R_Cperson、R_CAddress 以及 R_CPerson_Address 仅是对 R_SPerson 和 R_SAddress 的投影。

[0494]

CREATE VIEW R_OPerson(pid, name) AS SELECT pid, name FROM R_CPerson	CREATE VIEW R_OPerson_Address(pid, aid) AS SELECT pid, aid FROM R_CPerson_Address	CREATE VIEW R_OAddress(aid, state) AS SELECT aid, state FROM R_CAddress
--	--	--

[0495]

CREATE VIEW R_CPerson(pid, name) AS SELECT pid, name FROM R_SPerson	CREATE VIEW R_CPerson_Address(pid, aid) AS SELECT pid, aid FROM R_SAddress	CREATE VIEW R_CAddress(aid, state) AS SELECT aid, state FROM R_SAddress
--	---	--

[0496] 规定转换（按照 RES - 对象的对象）

[0497] 通过使 R_OPerson_Address 与 R_OAddress 联接并对结果嵌套来使用 R_Operson、R_OAddress 以及 R_OPerson_Address 表达 OPerson 对象。CREATE PRESCRIBED VIEW OPerson(pid, name, addr) AS

[0498] SELECT pid, name, NEST(SELECT Address(a.aid, a.state)

[0499] FROM R_OAddress a, R_OPerson_Address pa

[0500] WHERE pa.pid = p.pid AND a.aid = pa.aid)

[0501] R_OPerson p

[0502] Cperson 的合成视图

[0503] 简化之后的合成表达式可以是（记得对该示例，我们在表与其 RES 构造之间具有恒等转换）：

[0504] CREATE VIEW OPerson(pid, name, addr) AS

[0505] SELECT pid, name, NEST(SELECT Address(a.aid, a.state)

[0506] FROM SAddress a

[0507] WHERE a.pid = p.pid)

[0508] SPerson p

[0509] 最终的视图表明通过使用“直接”映射方法可能期望获得的视图。当查看更新流水线的 Δ 表达式时以及在需要查询映射视图的 Δ 表达式的通知流水线中，RES 方法的好处出现。

[0510] 更新映射视图

[0511] RES 空间中的非规定视图

[0512] R_SPerson 的 UM 视图仅是对 R_CPerson 的投影，而 R_SAddress 是通过将 R_CAddress 与一对多关联表 R_CPerson_Address 联接而构造成的。CDM 与对象空间之间的映射是恒等的。

[0513]

CREATE VIEW R_CPerson (pid, name) AS SELECT pid, name FROM R_OPerson	CREATE VIEW R_CPerson_Address (pid, aid) AS SELECT pid, aid FROM R_OPerson_Address	CREATE VIEW R_CAddress (aid, state) AS SELECT aid, state FROM R_OAddress
---	---	---

[0514]

CREATE VIEW R_SPerson (pid, name) AS SELECT pid, name, FROM R_CPerson	CREATE VIEW R_SAddress (aid, pid, state) AS SELECT aid, pid, state FROM R_C Person_Address, R_CAddress WHERE R_CPerson_Address.aid = R_CAddress.aid
---	--

[0515] 规定转换（按照对象的 RES-对象）

[0516] 需要将对象转换成 RES，使得更新可从对象空间推送到 RES 空间。R_OPerson 的规定转换是简单的投影，而 R_OAddress 和 R_OPerson_Address 的转换是通过在个人与其地址之间执行联接而实现的。这是“指针联接”或“导航联接”。

[0517]

CREATE PRESCRIBED VIEW R_OPerson (name, pid) AS SELECT name, pid FROM OPerson	CREATE PRESCRIBED VIEW R_OAddress (state, aid) AS SELECT a.state, a.aid FROM OPerson p, p.addr a	CREATE PRESCRIBED VIEW R_OPerson_Address (pid, aid) AS SELECT pid, aid FROM OPerson p, p.addr a
--	---	--

[0518] 合成的更新映射视图

[0519] 合成以上视图（以及采用某种简化）来得到以下合成的更新映射视图：

[0520]

CREATE VIEW SPerson (pid, name) AS SELECT pid, name, FROM OPerson	CREATE VIEW SAddress (aid, pid, state) AS SELECT a.aid, p.pid, a.state FROM OPerson p, p.addr a
---	---

[0521] 因此，表 SPerson 可被表达为对 OPerson 的简单投影，而 SAddress 是通过将 OPerson 与其地址联接来获得的。

[0522] 视图确认

[0523] 所生成的视图需要满足的一个重要性质是它们必须是“往返的”，即防止任何信息损失，必须确保当实体 / 对象被保存然后检索时，没有信息损失。换言之，想要对所有的实体 / 对象 D 确保：

[0524] $D = QM$ 视图 (UM 视图 (D))

[0525] 我们的视图生成算法确保该性质。如果该性质为真，则也说“查询和更新视图是往返的”或是双向的。现在为个人 - 地址示例阐述该性质。为简单起见，关注 RES 空间中的往返。

[0526] 对 R_OPerson 的确认

[0527] 在 OPerson 的查询视图中代入 SPerson，得到：

[0528] $R_OPerson(pid, name, age) =$

[0529] $SELECT pid, name, age FROM (SELECT pid, name, age FROM R_SPerson)$

[0530] 简化而得到

[0531] $R_OPerson(pid, name, age) = SELECT pid, name, age FROM R_SPerson$ 这与 $SELECT * FROM Person$ 等效。

[0532] 对 $OPerson_Address$ 的确认

[0533] 对 $R_OPerson_Address$, 这稍许复杂。我们有 :

[0534] $R_OPerson_Address(pid, aid) = SELECT pid, aid FROM R_SAddress$ 代入 $R_SAddress$, 得到 :

[0535] $R_OPerson_Address(pid, aid) =$

[0536] $SELECT pid, aid$

[0537] $FROM (SELECT aid, pid, state$

[0538] $FROM R_OPerson_Address pa, R_OAddress a$

[0539] $WHERE pa.aid = a.aid)$

[0540] 这被简化为 :

[0541] $R_OPerson_Address(pid, aid) =$

[0542] $SELECT pid, aid FROM R_OPerson_Address pa, R_OAddress a WHERE pa.aid = a.aid$

[0543] 为了示出以上的是 $SELECT * FROM R_OPerson_Address$, 需要外键依赖 $R_OPerson_Address.aid \rightarrow R_OAddress.aid$ 。如果该依赖性不成立, 则不能往返。这确实成立, 因为设置值的属性 $addr$ (地址) 的范围是 $R_OAddress$ 。该外键约束可按照两种方式表明 :

[0544] 1. $R_OPerson_Address.aid \subseteq R_OAddress.aid$

[0545] 2. $\pi_{aid, pid}(R_OPerson_Address \bowtie_{aid} R_OAddress) =$

[0546] $R_OPerson_Address$

[0547] 在以上表达式中代入该约束, 我们得到 :

[0548] $R_OPerson_Address(pid, aid) = SELECT pid, aid FROM R_OPerson_Address$

[0549] 地址确认

[0550] $R_OAddress$ 被给定为 :

[0551] $R_OAddress(aid, state) = SELECT aid, state FROM R_SAddress$

[0552] 代入 $R_SAddress$, 得到 :

[0553] $R_OAddress(aid, state) =$

[0554] $SELECT aid, state$

[0555] $FROM (SELECT aid, pid, state$

[0556] $FROM R_OPerson_Address pa, R_OAddress a$

[0557] $WHERE pa.aid = a.aid)$

[0558] 这可被重述为 :

[0559] $R_OAddress(aid, state) = SELECT aid, state FROM R_OPerson_Address pa, R_$

OAddress a

[0560] WHERE pa.aid = a.aid

[0561] 此处，如果外键依赖性 R_OAddress.aid → R_OPerson_Address.aid 成立，则与 R_OPerson_Address 的联接是冗余的。该依赖性仅当 R_OAddress 在依赖于 R_OPerson 存在时（即，`attrs` 是合成）时才成立。如果这不是真的，则我们的视图不能往返。因此，我们具有约束：

[0562] $\pi_{aid, state}(R_OAddress \bowtie_{aid} R_OPerson_Address) = R_OAddress$

[0563] 因此，得到以下表达式：

[0564] R_OAddress(aid, state) = SELECT aid, state FROM R_OAddress

[0565] 查询转换

[0566] 查询转换器

[0567] EDP 查询转换器 (EQT) 负责通过利用映射元数据将查询从对象 /EDM 空间转换到提供者空间。用户查询可用各种句法表达，例如 eSQL、C# 序列、VB SQL 等。EQT 体系结构在图 17 中示出。现在描述 EQT 的不同组件。

[0568] 解析器 1711 通过解析用若干形式中的一种表达的用户查询（包括 eSQL、语言集成查询 (LINQ)、C# 序列以及 VB sql）来执行句法分析。此时将检测并标记任何句法错误。

[0569] 对 LINQ，句法分析（以及语义分析）与语言（C#、VB 等）本身的句法分析阶段集成。对 eSQL，句法分析阶段是查询处理器的一部分。一般，每个语言有一个句法分析器。

[0570] 句法分析阶段的结果是解析树。该树被馈送到语义分析阶段 1712。

[0571] 参数绑定器和语义分析器组件 1712 管理用户查询中的参数。该模块跟踪查询中的参数的数据类型和值。

[0572] 语义分析阶段在语义上确认由句法分析阶段 1711 产生的解析树。查询中的任何参数此时必须已被绑定，即其数据类型必须已知。此处检测并标记任何语义错误；如果成功，则该解析的结果是语义树。

[0573] 注意对 LINQ，如前所述，语义分析阶段与语言的语义分析阶段本身集成。一般每个语言有一个语义分析器，因为每个语言有一个句法树。

[0574] 语义分析阶段逻辑上由以下组成：

[0575] 1. 名字解析：查询中的所有名字在此时解析。这包括对范围、类型、类型的属性、类型的方法等的引用。作为副作用，也推断这样的表达式的数据类型。该子阶段与元数据组件交互。

[0576] 2. 类型检查和推断：可类型检查查询中的表达式，并推断结果类型。

[0577] 3. 确认：其他种类的确认在这里进行。例如，在 SQL 处理器，如果查询块包含分组子句，则该阶段可用于实施选择列表仅可引用分组键或聚合函数的限制。

[0578] 语义分析阶段的结果是语义树。此时，认为查询是有效的——在查询编译期间之后将不应出现任何其他语义错误。

[0579] 代数化阶段 1713 取得语义分析阶段 1712 的结果，并将其转化成更适于代数变换的形式。该阶段的结果是逻辑扩展关系操作符树，又称为代数树。

[0580] 代数树是基于核心关系代数操作符的——选择、投影、联接、并以及将其扩充以附加的操作，如嵌套 / 取消嵌套以及透视 (pivot) / 取消透视。

[0581] 查询转换器的视图展开阶段 1714 有可能递归地代入用户查询中所引用的任何对象的 QM 视图表达式。在视图转换过程的结尾，得到按照存储描述查询的树。

[0582] 在对象层的情况中，视图展开可能已对至存储空间所有方式进行（在我们有存储在元数据储存库中的优化的 OS 映射的情况中）或查询树可能已经被变换到 EDM 层。在后一情况中，需要取得该树，并在 EDM 概念现在要被转换成存储概念的要求下将其重新馈送到视图展开组件。

[0583] 变换 / 简化组件 1715 可以是提供者 1730 专用的，或在替换实施例中，可以是可由各个提供者利用的 EDP 通用组件。对查询树执行变换有几个理由：

[0584] 1. 推送到存储的操作符：EQT 将复杂操作符（例如，联接、过滤、聚合）推送到存储。否则，这样的操作将必须在 EDP 中实现。EDP 的值物化层仅执行“非关系补偿”操作，诸如嵌套。如果无法将操作符 X 向下推送到查询树中的值物化节点以下，且值物化层不能执行操作 X，则将该查询声明为非法的。例如，如果查询具有不能推送到提供者的聚合操作，将声明该查询非法，因为值物化层不能执行任何聚合。

[0585] 改进的性能：查询复杂性降低是重要的，且要避免将巨大的查询发送到后端存储。例如，WinFS 中的某些当前查询是非常复杂的，且花费大量时间来执行（相应的手写查询要快一个数量级）。

[0586] 改进的调试能力：较简单的查询也使得开发人员易于调试系统以及理解正在进行的事情。

[0587] 变换 / 简化模块 1715 可将表示查询的某些或全部代数树变换成等效的子树。注意到，这些基于试探的变换是逻辑上的，即不是使用成本模型进行的。这种逻辑变换可包括以下示例性提供者专用服务：

[0588] 子查询展平（视图和嵌套的子查询）

[0589] 联接消除

[0590] 谓词消除和整合

[0591] 谓词下推

[0592] 共有子表达式消除

[0593] 投影修剪

[0594] 外联接→内联接变换

[0595] 消除左相关

[0596] 该 SQL 生成模块 1731 是提供者组件 1730 的一部分，因为所生成的 SQL 是提供者专用的。在简化之后，将代数树传递给在生成适当 SQL 之前可能进一步执行提供者专用变换或简化的提供者。

[0597] 当查询在服务器执行之后，结果被流传送到 EDP 客户机。提供者 1730 展示可由应用程序用来获取作为 EDM 实体的结果的 DataReader。值物化服务 1741 可取得这些读取器，并将其转化成相关 EDM 实体（作为新的 DataReader）。这些实体可由应用程序消费，或新的 DataReader 可被传递到上层对象物化服务。

[0598] EQT1700 将物化表示为查询树中的操作符。这允许常规的查询转换流水线产生

EDM 空间中的对象，这然后可被直接馈送给用户，代替要求特殊的带外操作来执行实际的物化。这也允许如部分对象取、急切加载等各种优化在用户查询上执行。

[0599] 查询示例

[0600] 考虑已经开发的个人 - 地址示例。假定用户想要运行以下查询——找到 WA (华盛顿) 中的所有人。可将该查询用伪 CSQL 写成：

```
[0601] SELECT x.name FROM OPerson x, x.addrs y WHERE y.state = "WA"
```

[0602] 如果此时使用 Person 的查询视图进行视图展开，则得到：

```
[0603] SELECT x.name
```

```
[0604] FROM (SELECT pid, name,
```

```
[0605]     NEST (SELECT OAddress (a.aid, a.state) FROM SAddress a where a.pid = p.pid)
```

```
[0606] FROM SPerson p) as x, x.addrs y
```

```
[0607] WHERE y.state = "WA"
```

[0608] 该查询可在发送给后端服务器之间进行简化：

```
[0609] SELECT p.name
```

```
[0610] FROM SPerson p, SAddress a
```

```
[0611] WHERE p.pid = a.pid
```

[0612] 元数据

[0613] EQT 在查询的编译和执行期间要求各种元数据。元数据包括

[0614] 应用程序空间元数据：关于语义分析期间确认用户查询所需的范围 / 集合、类型、类型属性、类型方法的信息。

[0615] 模式空间元数据：关于视图编译期间所需的实体集合、CDM 类型和属性的信息。用于变换的关于实体以及实体上的约束之间的关系的的信息。

[0616] 存储空间元数据：如上所述。

[0617] 应用程序 -> 模式映射：表示视图展开所需的视图定义的逻辑操作符树。

[0618] 模式 -> 存储映射：如上所述。

[0619] 错误报告流水线

[0620] 查询处理各个阶段的错误可按照用户可理解的方式报告。各种编译和执行时错误可在查询处理期间发生。句法和语义分析期间的错误大多数处于应用程序空间中，且要求很少的特殊处理。变换期间的错误大多数是资源错误（在存储器外等），且需要某种特殊处理。代码生成和随后的查询执行期间的错误可能需要适当处理。错误报告中关键的挑战是将在较低的抽象级发生的运行时错误映射到应用程序级有意义的错误。这意味着需要通过存储、概念和应用程序映射处理较低级错误。

[0621] 查询示例

[0622] 我们的示例 00 查询取得在华盛顿有地址的所有人的名字：

```
[0623] SELECT p.name
```

```
[0624] FROM OPerson p, p.addrs as a
```

```
[0625] WHERE a.state = ' WA'
```

[0626] 步骤 1：转化到关系方面

[0627] 该查询可被转化到以下按照 R_OPerson、R_OPerson_Address 和 R_OAddress 表达

的纯关系查询。实质上，如有需要，将各个导航属性（点“.”表达式）展开成联接表达式。

[0628] SELECT p.name

[0629] FROM R_OPerson p, R_OPerson_Address pa, R_OAddress a

[0630] WHERE p.pid = pa.pid AND pa.aid = a.aid AND a.state = ' WA '

[0631] 注意，查询仍在对象域，且按照对象范围。

[0632] 步骤 2：视图展开：转化到存储空间

[0633] 现在进行视图展开以将查询转化成 SQL：

[0634] SELECT p.name

[0635] FROM (SELECT pid, name, age FROM SPerson)p,

[0636] (SELECT pid, aid FROM SAddress)pa,

[0637] (SELECT aid, state FROM SAddress)a

[0638] WHERE p.pid = pa.pid AND pa.aid = a.aid AND a.state = ' WA '

[0639] 步骤 3：查询简化

[0640] 现在应用一系列逻辑变换来简化该查询。

[0641] SELECT p.name

[0642] FROM SPerson p, SAddress pa, SAddress a

[0643] WHERE p.pid = pa.pid AND pa.aid = a.aid AND a.state = ' WA '

[0644] 现在，可消除 SAddress 的主键 (aid) 上的冗余自联接，并获得：

[0645] SELECT p.name

[0646] FROM SPerson p, SAddress a

[0647] WHERE p.pid = a.pid AND a.state = ' WA '

[0648] 以上所有是相当直接的。现在有了可被发送给 SQL Server 的查询。

[0649] 更新的编译时处理

[0650] EDP 允许应用程序创建新对象、更新它们、删除它们，然后持久存储这些改变。OR 映射组件需要确保这些改变被正确地转换成后端存储改变。如前所述，使用按照对象来声明表的更新映射视图。通过使用这样的视图，基本上将更新传播问题缩减为物化视图维护问题，其中对基本关系的改变需要被传播到视图；在 UM 视图的情况下，“基本关系”是对象而“视图”是表。通过按照此方式对问题建模，可利用已经在关系数据库领域开发的视图维护技术的知识。

[0651] 更新映射视图生成

[0652] 如在查询情况中一样，大量用于更新的映射工作在编译时执行。与类、EDM 类型和表的关系表达模式一样，生成这些类型和相应的 RES 构造之间的规定转换。也生成类的 RES 构造与 EDM 类型之间以及 EDM 类型的 RES 构造与存储表之间的更新映射视图。

[0653] 在已经开发的个人 - 地址示例的帮助下，理解这些 UM 视图。记得所构造的对象 RES 构造 (R_OPerson、R_OAddress、R_OPerson_Address)。

[0654] 更新映射视图（按照对象 RES 的表的 RES）

[0655] R_OPerson 的 UM 视图仅是对 R_SPerson 的投影，而 R_SAddress 是通过将 R_

OAddress 与一对多关联表 R_OPerson_Address 联接而构造成的。

[0656]

CREATE VIEW R_SPerson(pid, name) AS SELECT pid, name, FROM R_OPerson	CREATE VIEW R_SAddress(aid, pid, state) AS SELECT aid, pid, state FROM R_OPerson_Address pa, R_OAddress a WHERE pa.aid = a.aid
--	---

[0657] 规定转换（按照对象的 RES）

[0658] 需要将对象转换成 RES，使得更新可从对象空间推送到 RES 空间。使用“o2r”功能将对象的虚拟存储器地址转换成 pid 和 aid 键——在实现中，简单地从对象的阴影状态得到键。R_OPerson 的规定转换是简单的投影，而 R_OAddress 和 R_OPerson_Address 的转换是通过在个人与其地址之间执行联接而实现的。

[0659]

CREATE PRESCRIBED VIEW R_OPerson(name, pid) AS SELECT name, pid FROM OPerson	CREATE PRESCRIBED VIEW R_OAddress(state, aid) AS SELECT a.state, a.aid FROM OPerson p, p.addr a
---	--

[0660]

CREATE PRESCRIBED VIEW R_OPerson_Address(pid, aid) AS SELECT p.pid, a.aid FROM OPerson p, p.addr a	
---	--

[0661] 合成的更新映射视图

[0662] 合成以上视图（以及采用某种简化）来得到以下合成的更新映射视图：

[0663]

CREATE VIEW SPerson(pid, name) AS SELECT pid, name FROM OPerson	CREATE VIEW SAddress(aid, pid, state) AS SELECT a.aid, p.pid, a.state FROM OPerson p, p.addr a
---	--

[0664] 因此，表 SPerson 可被表达为对 OPerson 的简单投影，而 SAddress 是通过将 OPerson 与其地址联接来获得的。

[0665] Δ 表达式生成

[0666] 当应用程序要求其对象改变被保存到后端时，实施例可将这些改变转换到后端存储，即可按照对象的 Δ 表达式（基本关系）生成表（视图）的 Δ 表达式。这是视图生成 / 编译过程分解成 RES 构造实际有帮助的地方。非规定映射的 Δ 表达式可相对简单地生成，因为这些映射是在关系空间（RES 是纯关系的），且已经做了大量关系数据库中的工作以实现该目标。例如，在数据库文献中，已经为按照关系操作符，诸如选择、投影、内或外或半联接、并、交和差表达的视图开发了 Δ 表达式。对非关系构造，所有需要做的是设计将非关系构造转换至 RES 空间 / 自 RES 空间转换的规定 Δ 表达式。

[0667] 采用我们的 Person 示例来理解 Δ 表达式。考虑其中 RES 构造（例如，R_

SAddress) 被表达为两个对象集合 (R_OAddress 和 R_OPerson_Address) 的联接的情况。这样的视图的 Δ 表达式可使用以下规则来获取 (假定联接视图为 $V = R \text{ JOIN } S$) :

[0668] $i(V) = [i(R) \text{ JOIN } S \text{ 新 }] \text{ UNION } [i(S) \text{ JOIN } R \text{ 新 }]$

[0669] $d(V) = [d(R) \text{ JOIN } S] \text{ UNION } [d(S) \text{ JOIN } R]$

[0670] 在该表达式中, $i(X)$ 和 $d(X)$ 表示关系或视图 X 所插入和删除的元组, 而 $R^{\text{新}}$ 表示基本关系 R 在应用了其所有更新之后的新值。

[0671] 因此, 为了便于运行时的更新, 一个示例性实施例可首先在编译时生成以下 Δ 表达式:

[0672] 1. 按照更新的对象集合 1081 的组的 Δ 改变表达式的 RES 关系 1811 的规定 Δ 改变表达式 1803, 例如按照 $i(OPerson)$ 的 $i(R_OPerson)$ 。

[0673] 2. 按照 RES 关系 1812 的 Δ 改变表达式的表 1802 的规定 Δ 改变表达式 1804, 例如按照 $i(R_SPerson)$ 的 $i(SPerson)$ 。

[0674] 3. 按照对象的 RES 关系的 Δ 表达式表达的表的 RES 关系的 Δ 表达式 1813, 例如, 按照 $i(R_OPerson)$ 的 $i(R_SPerson)$ 。

[0675] 可合成 (1)、(2) 和 (3) 来获得按照对象 1821 (例如, $OPerson$) 的 Δ 表达式的表 1822 (例如, $SPerson$) 的 Δ 表达式 1820。该合成在图 18 示出。因此, 与在查询的情况中一样, 在编译时, 现在具有从对象到表的直接转换。在更新的情况中, 实际利用了 RES 分解来生成 Δ 表达式 (对 QM 视图, 该优点适用于通知)。

[0676] 注意, 不需要更新的 Δ 表达式——如稍后可见, 模型更新可通过将其置入插入和删除集来建模; 处理后步骤在稍后在改变实际应用于数据库之前将它们再次转化回更新。该方法的一个理由在于, 对增量式视图维护的现有工作一般不具有用于更新的 Δ 表达式。或者, 开发这样的表达式的更复杂实施例是可行的。

[0677] 当执行视图合成之后, 表的 Δ 表达式是完全按照对象集合和对象的插入和删除集的, 即 $i(SPerson)$ 是按照 $OPerson$ 、 $i(OPerson)$ 和 $d(OPerson)$ 的。这些 Δ 表达式中的某些需要计算对象集合, 例如 $i(OPerson)$ 需要 $Eperson$ 用于其计算。然而, 整个集合可不在 EDP 客户机处高速缓存 (或可能想要在集合最一致且最新的值上运行操作)。为了解决该问题, 使用相应的查询映射视图来展开对象集合, 例如使用 $OPerson$ 的 QM 视图并按照 $SPerson$ 和所需的其他关系来表达它。因此, 在一个实施例中, 在编译过程的结尾处, $SPerson$ 的所有 Δ 表达式是按照 $i(OPerson)$ 、 $d(OPerson)$ 和关系 $SPerson$ 本身表达的——在运行时, 给定 $OPerson$ 的插入和删除集, 现在可生成可在服务器执行的相关 SQL 语句。

[0678] 总而言之, 给定表和对象的 RES 构造之间的 QM 视图和 UM 视图以及这些构造与表 / 对象之间的规定转换, 可执行以下示例性步骤:

[0679] 1. 生成以上在步骤 1、2 和 3 中提及的 Δ 表达式。

[0680] 2. 合成这些表达式, 使得具有按照对象 ($OPerson$) 的 Δ 表达式以及对象集合本身的表 ($SPerson$) 的 Δ 表达式。

[0681] 3. 使用其 QM 视图来展开对象集合以获得按照对象的 Δ 表达式以及表本身的表 ($SPerson$) 的 Δ 表达式, 即消除对象集合。可能存在允许实施例避免这种展开或者了解整个集合在客户机高速缓存的特例。

[0682] 4. 简化 / 优化表达式，使得它可减少运行时工作。

[0683] 除了此处明确描述的特定实现之外，本领域的技术人员通过考虑此处公开的说明书，其他方面和实现是显然的。说明书和示出的实现旨在仅被认为是示例，落入所附权利要求书的真正范围和精神之内。

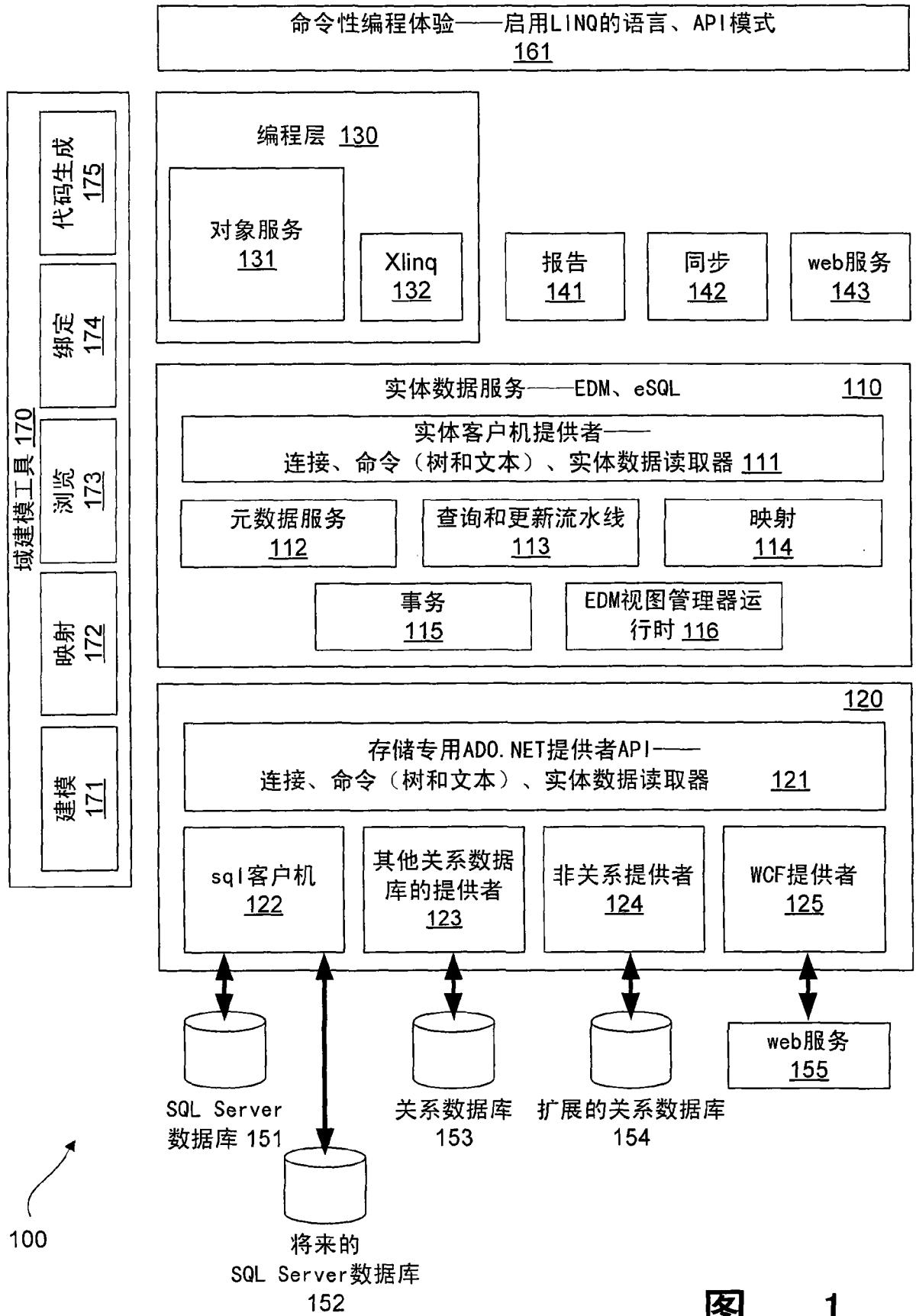


图 1

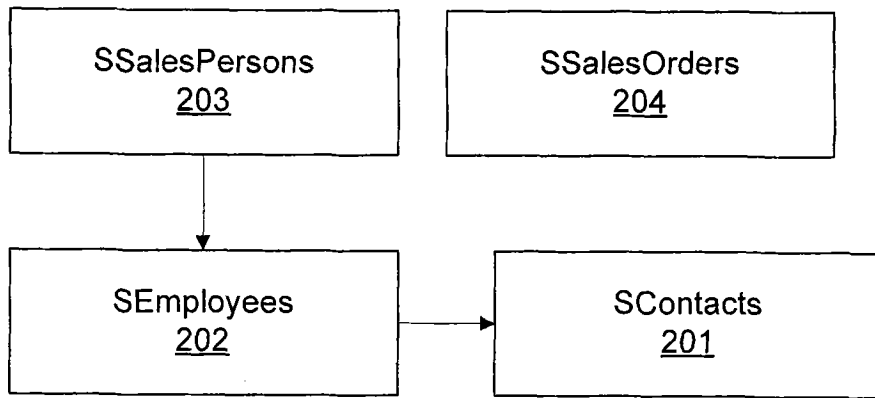


图 2

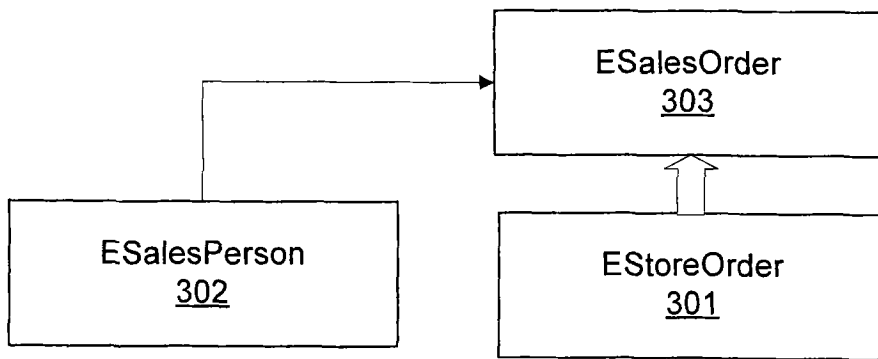


图 3

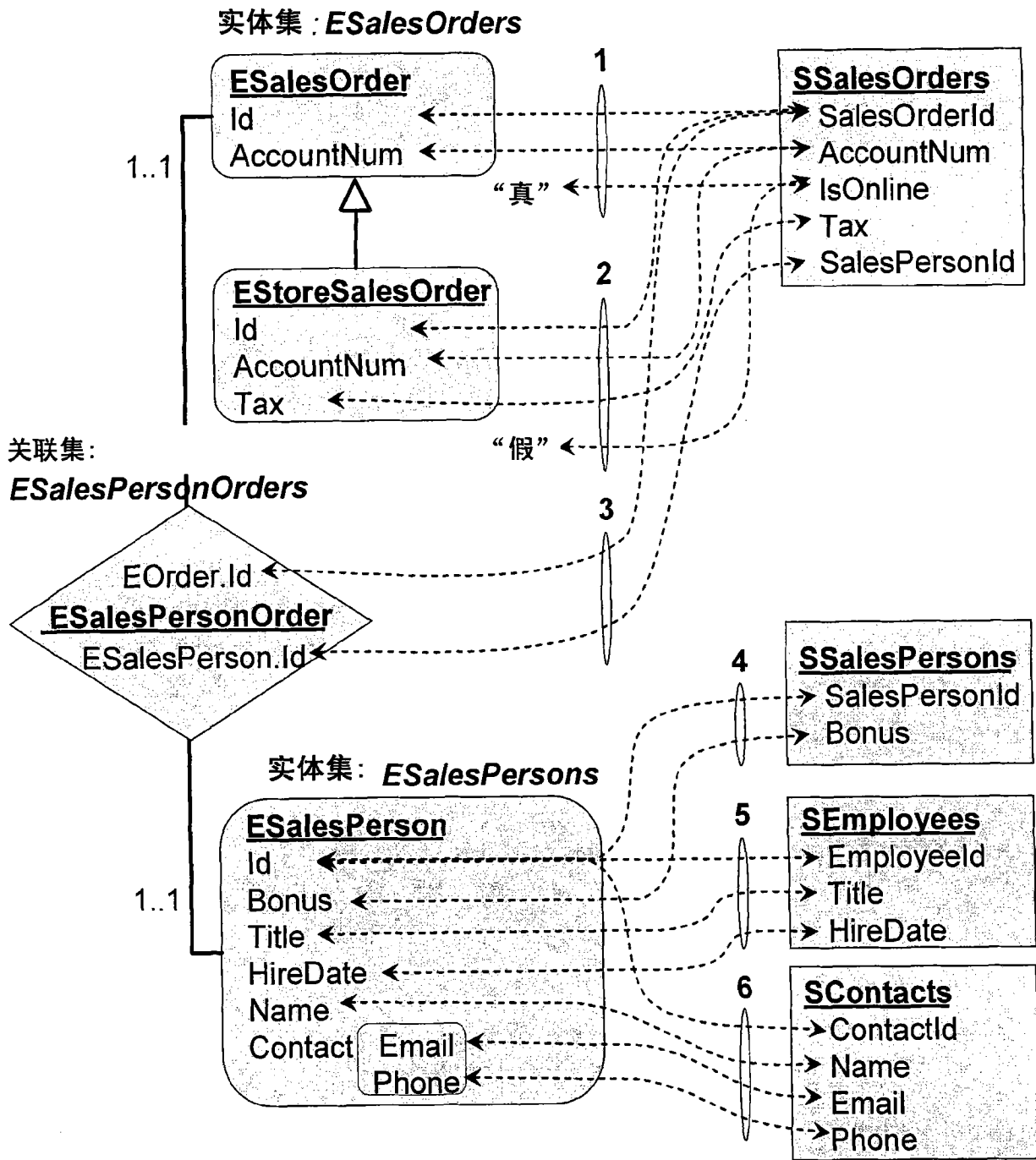


图 4

SELECT o.Id, o.AccountNum FROM ESalesOrders o WHERE o IS OF ESalesOrder	=	SELECT SalesOrderId, AccountNum FROM SSalesOrders WHERE IsOnline = "true"	1
SELECT o.Id, o.AccountNum, o.Tax FROM ESalesOrders o WHERE o IS OF EStoreSalesOrder	=	SELECT SalesOrderId, AccountNum, Tax FROM SSalesOrders WHERE IsOnline = "false"	2
SELECT o.EOrder.Id, o.ESalesPerson.Id FROM ESalesPersonOrders o	=	SELECT SalesOrderId, SalesPersonId FROM SSalesOrders	3
SELECT p.Id, p.Bonus FROM ESalesPersons p	=	SELECT SalesPersonId, Bonus FROM SSalesPersons	4
SELECT p.Id, p.Title, p.HireDate FROM ESalesPersons p	=	SELECT EmployeeId, Title, HireDate FROM SEmployees	5
SELECT p.Id, p.Name, p.Contact.Email, p.Contact.Phone p	=	SELECT ContactId, Name, Email, Phone FROM SContacts	6



5

查询
视图



```

ESalesOrders = QV1
SELECT
  CASE WHEN T.IsOnline = True
    THEN ESalesOrder(T.SalesOrderId, T.AccountNum)
    ELSE EStoreSalesOrder(T.SalesOrderId,
                          T.AccountNum, T.Tax)
  END
FROM SSalesOrders AS T

ESalesPersonOrders = QV2
SELECT ESalesPersonOrder(
  CreateRef(ESalesOrders, T.SalesOrderId),
  CreateRef(ESalesPersons, T.SalesPersonId))
FROM SSalesOrders AS T

```

更新
视图



```

SSalesOrders = UV1
SELECT o.Id, po.Id, o.AccountNum,
  TREAT(o AS EStoreSalesOrder).Tax AS Tax,
  CASE WHEN o IS OF ESalesOrder THEN TRUE ELSE FALSE END
  AS IsOnline
FROM ESalesOrders AS o
INNER JOIN ESalesPersonOrders AS po
ON o.SalesOrderId = Key(po.EOrder).Id

```

查询
视图



```

ESalesPersons = QV3
SELECT ESalesPerson(p.SalesPersonId , p.Bonus ,
  e.Title , e.HireDate ,
  c.Name , Contact(c.Email , c.Phone ))
FROM SSalesPersons AS p, SEmployees AS e, SContacts AS c
WHERE p.SalesPersonId = e.EmployeeId
AND e.EmployeeId = c.ContactId

```

更新
视图



```

SSalesPersons = UV2
SELECT p.Id, p.Bonus FROM ESalesPersons AS p

SEmployees = UV3
SELECT p.Id, p.Title , p.HireDate FROM ESalesPersons AS p

SContacts = UV4
SELECT p.Id, p.Name, p.Contact.Email, p.Contact.Phone
FROM ESalesPersons AS p

```

图 6

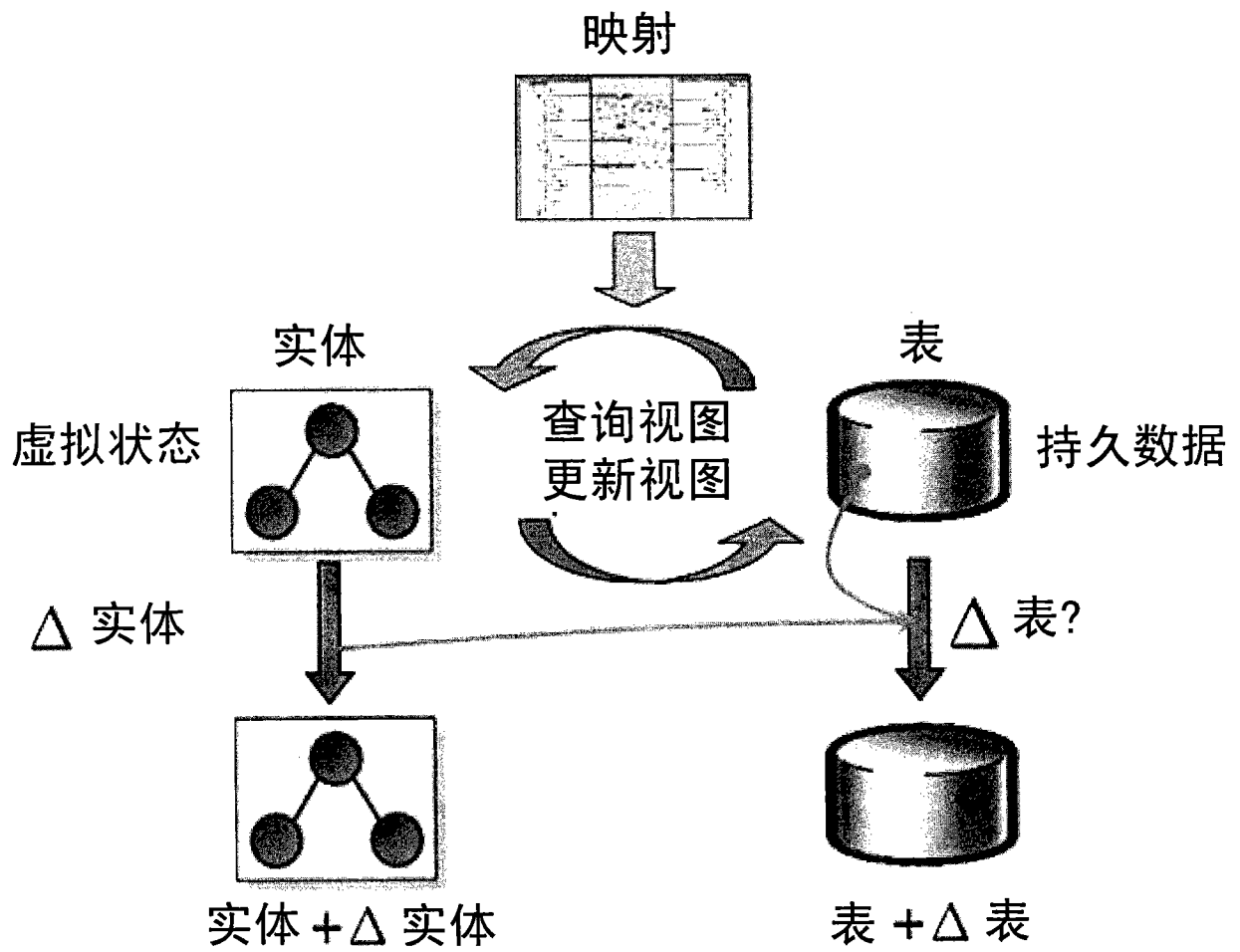


图 7

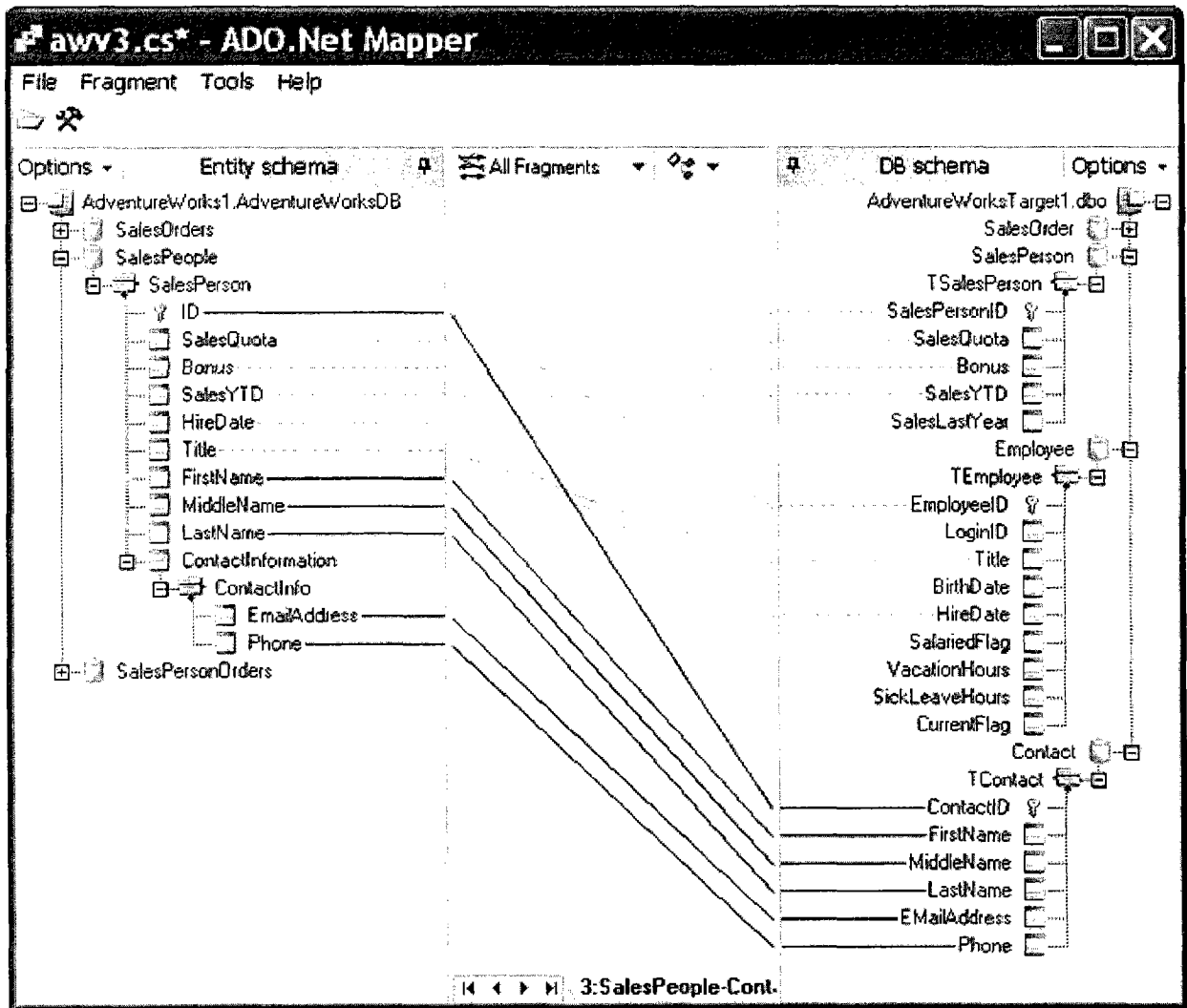


图 8

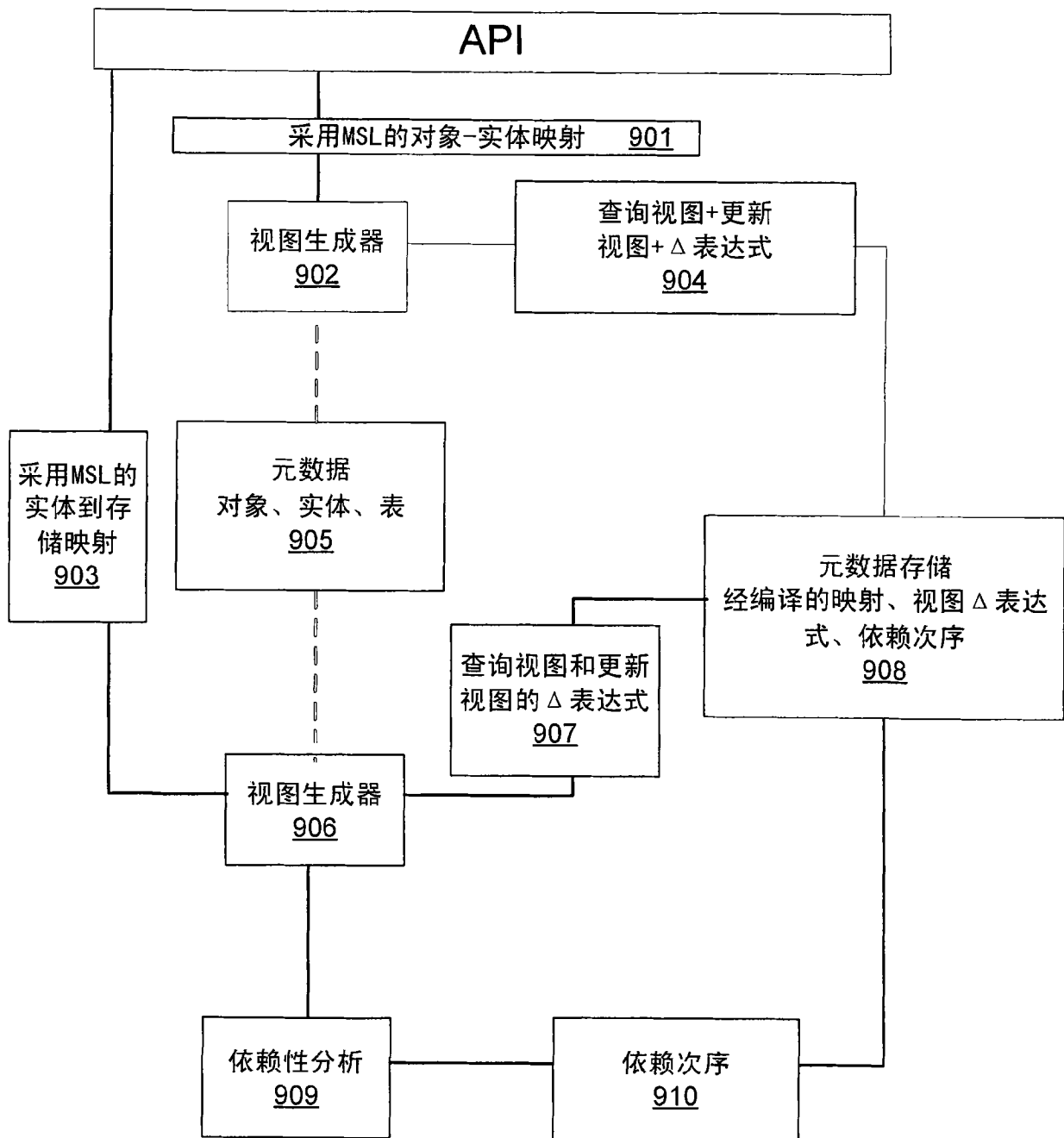


图 9

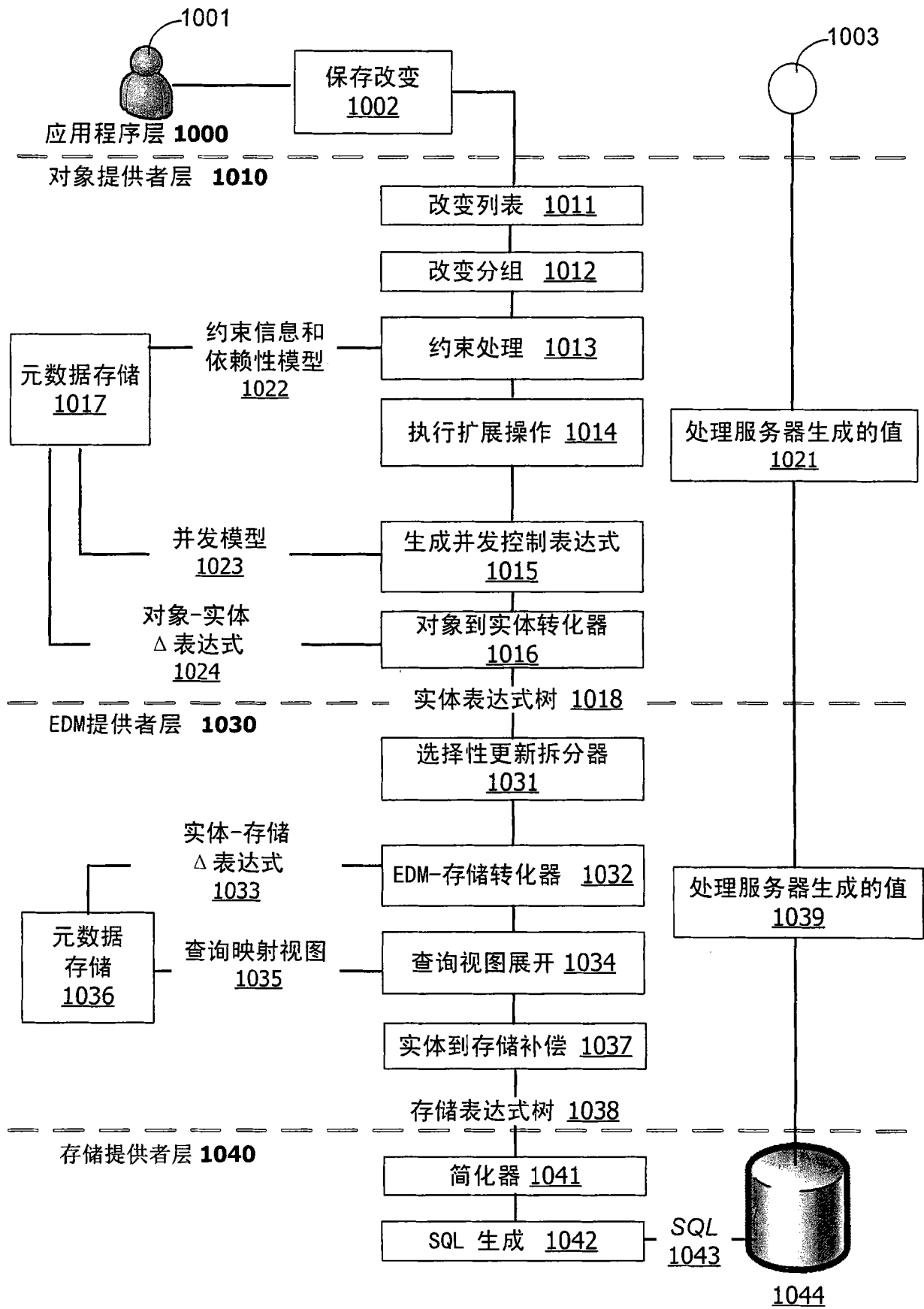


图 10

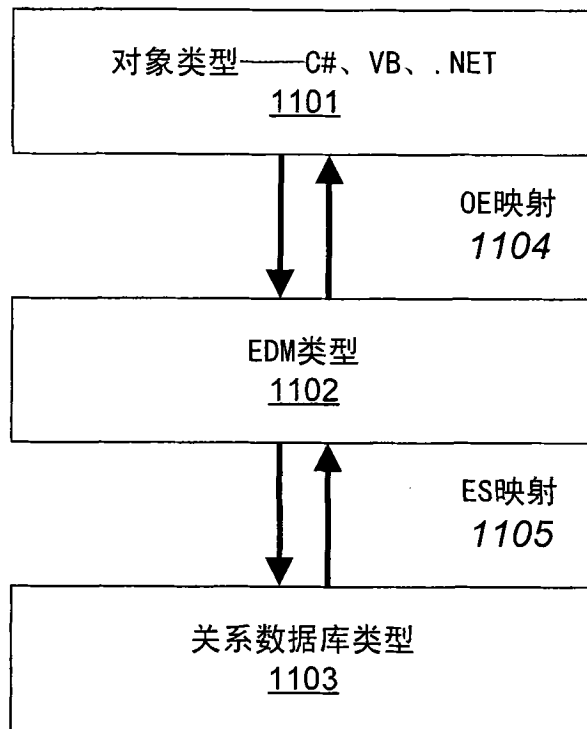


图 11

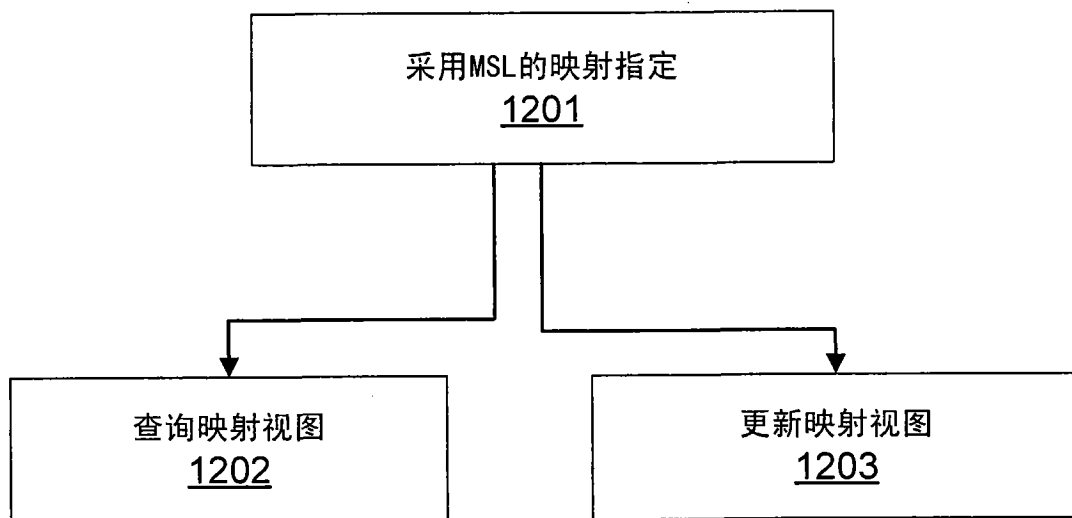


图 12

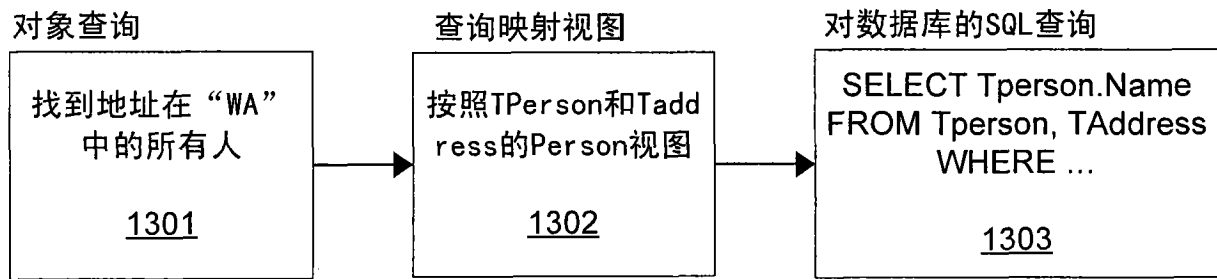


图 13

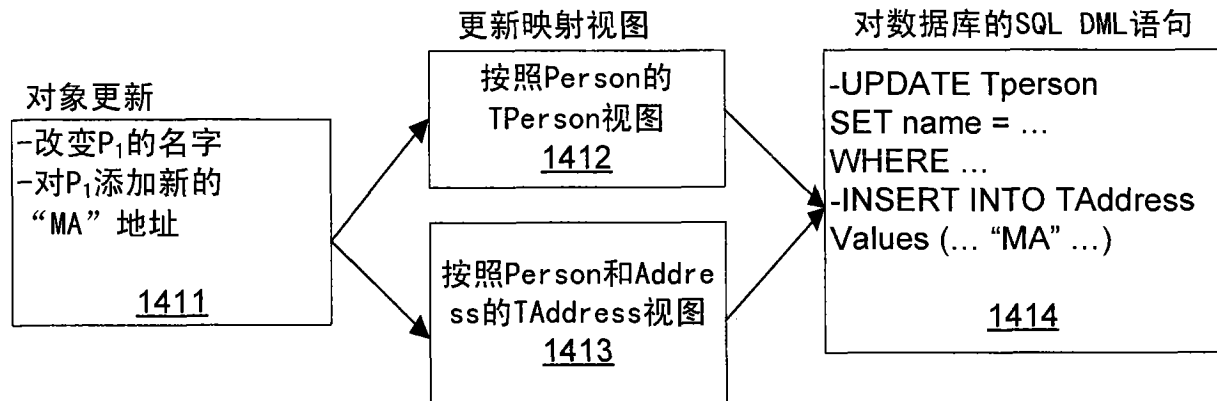


图 14

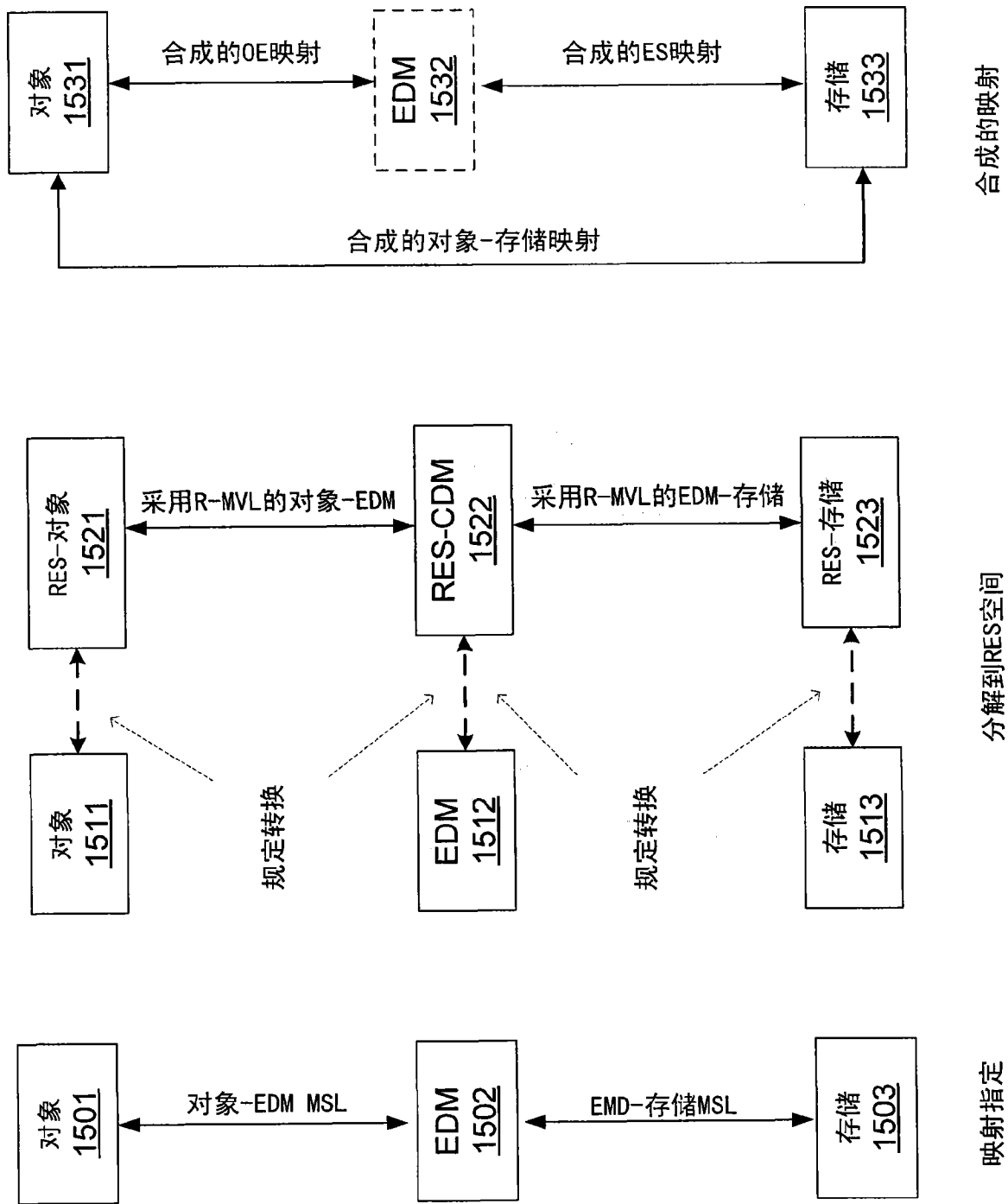


图 15

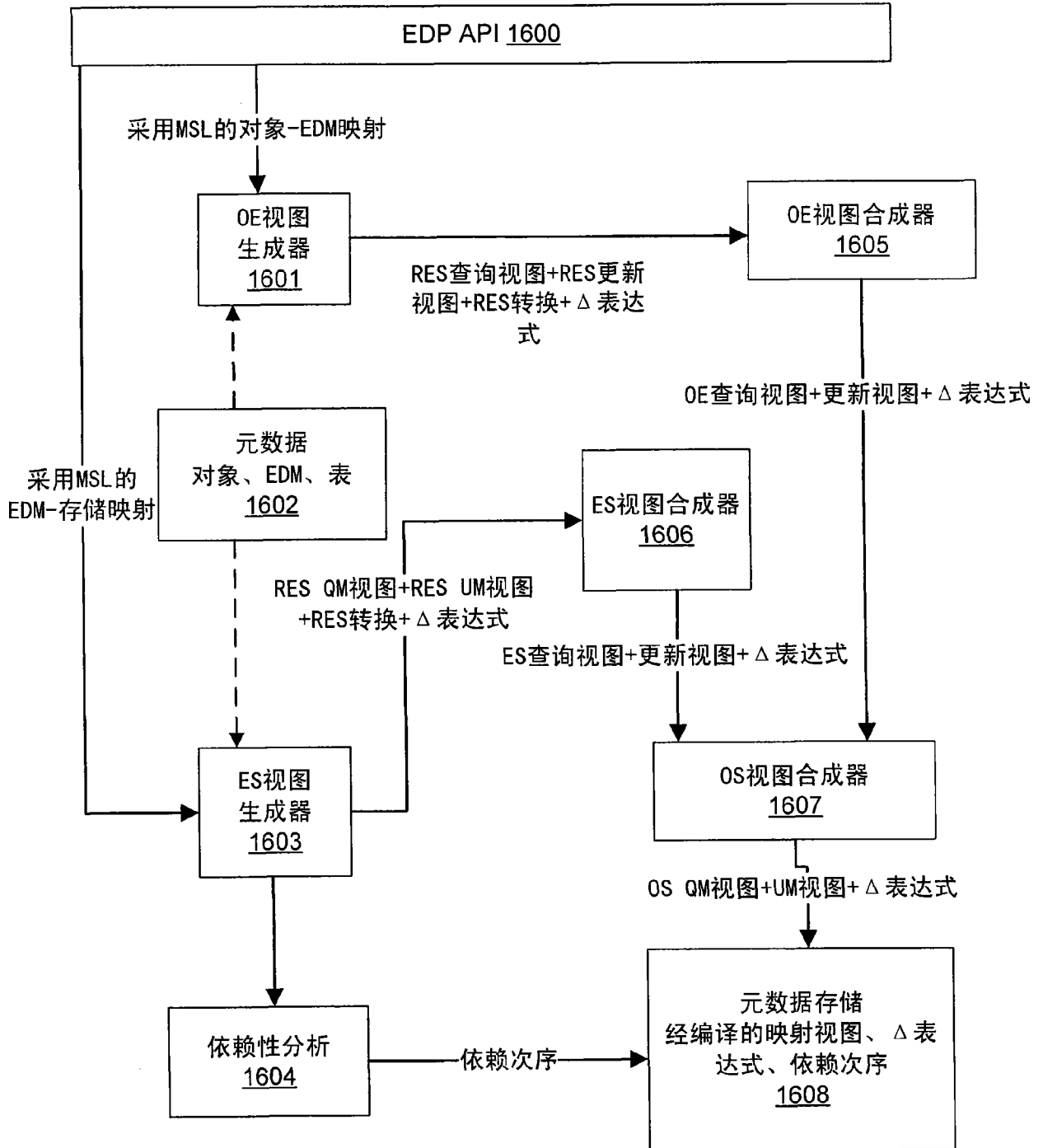


图 16

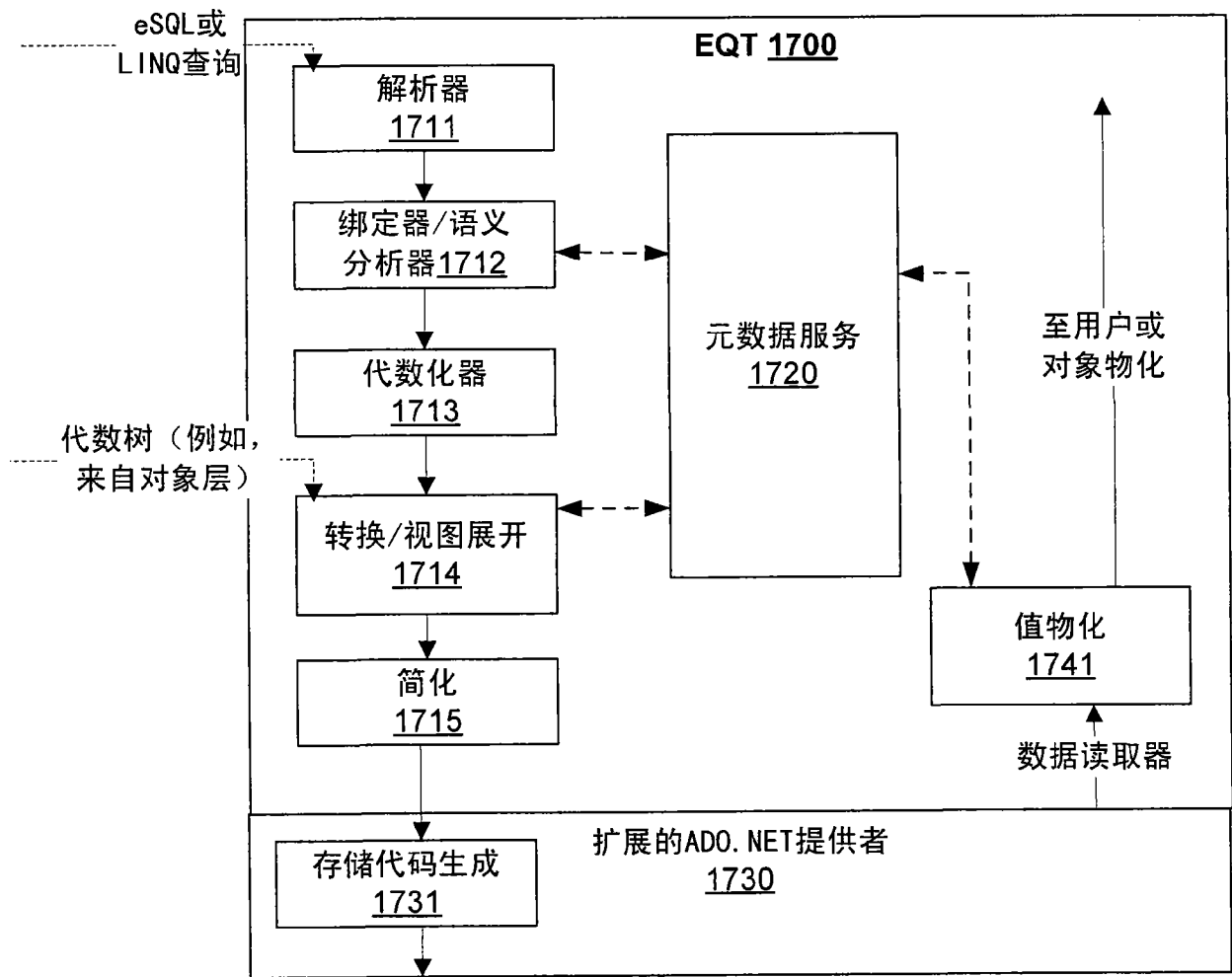


图 17

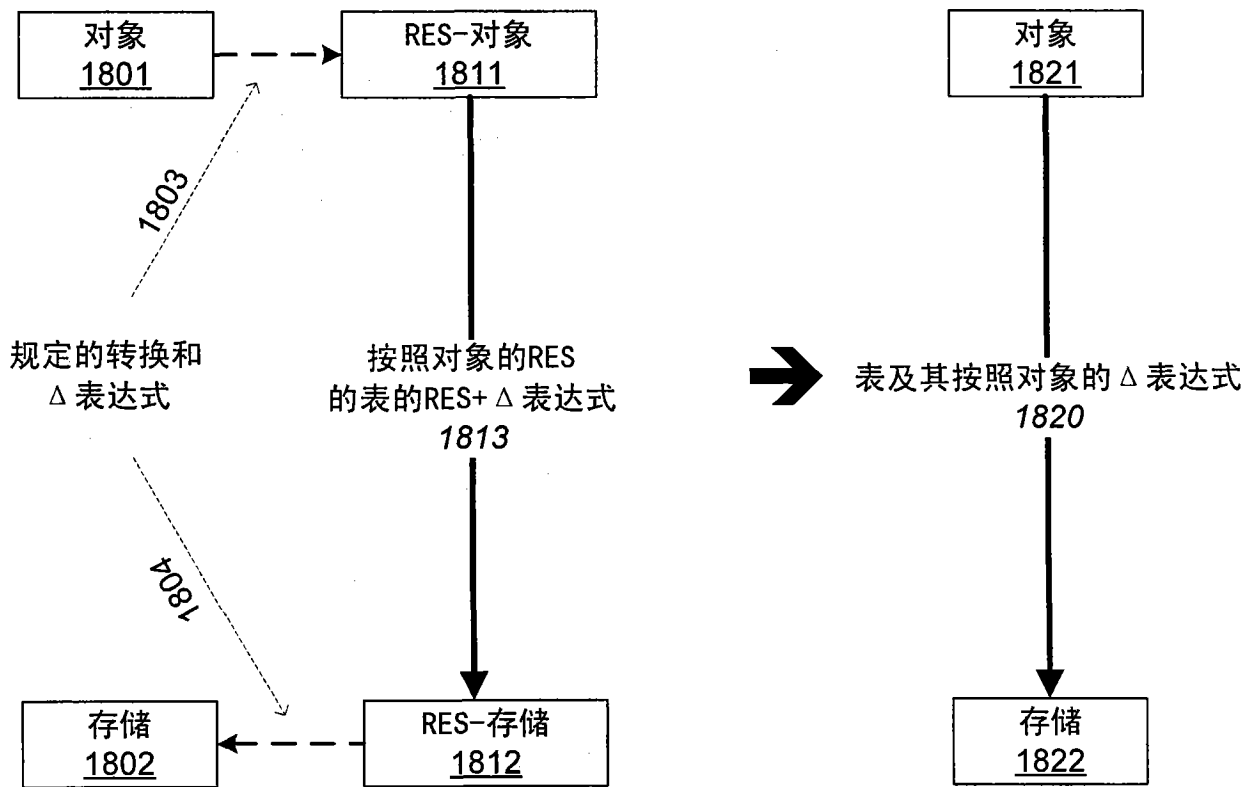


图 18