

第二部分 数据库的组成部分

第4章 类和实体

对类和类之间关系的正确识别是数据模型的关键所在。本章将讨论如何发现、识别、以及描述类。要想将建模过程缩减为一个简单的、逐步进行的过程是不太可能的。从本质上讲，建模是一项艺术。对一个给定的复杂情况而言，不存在唯一正确的数据模型，然而却存在好的数据模型；与之相对，也存在着差的数据模型。一个企业或机构的某个数据模型可能会优于另一个数据模型，但就如何为一个特定的系统建立数据模型，却没有唯一的解决方案。

好模型的目标是将工程项目整个生存期内的花费减至最小。将目光集中在最大限度地降低开发费用上是一个错误。好的模型会考虑到随时间的推移系统将可能发生的变化，因而设计时也要很容易地能适应这些变化。

建模过程也是十分主观的，有不同的建立数据模型的风格。设计者会对同类问题提出不同的处理方法。最优数据模型不只是被模型化的情况的一个函数，它还依赖于如下因素：

开发组的技能。结构越复杂，对开发人员的技能要求越高。

设计环境。大多数CASE产品在其所能生成的结构方面都有所局限，设计者可能需要将设计调整为CASE友好的。

开发环境。开发工具（尤其是4GL）在其能轻易支持的结构种类方面存在局限，由CASE生成的应用在这方面则更为有限，设计者可能希望修改设计以适应所选择的开发环境。

系统期望的生存期。短期的项目（如支持奥林匹克运动会或工程项目的系统）用于长期的可能性会小一些。

项目的时间/费用压力。尽管灵活的模型降低了长期项目的费用，但如果具有很大压力，如需要在该系统的第一版保持短时间和低开发费用的话，某些灵活性就要牺牲掉了。

由于工具变得更为丰富以及UML概念的出现，与使用ERD相比，建模变得更像一门艺术了。现在这些新的关系类型使得对同一情况建模有了更多的方法。

尽管本章讨论类的识别，关系在另一单独章节（第8章）中讨论，但我们不想造成这样一种印象，即顺序处理是设计数据库的正确方法。事实上，不能通过简单地先识别类，然后再识别这些类之间的关系来设计，这两个过程必须同时出现。即使过去使用ERD时，首先确定所有实体，然后用关系将它们连接在一起也是不可能的。然而，许多不良数据模型却正是用这种方法建立的。

本章将给出特定的实例，同时给出问题及答案来帮助正确地识别类、属性和主码。

4.1 识别类

第3章列出了对象的几个定义。从本质上讲，类（对象组）是某一特定组织关注的、数据库要对之进行跟踪的事物。通常，在ERD风格的建模过程中，分析员会暗自发问：“需要明确

什么？”从某种程度上讲，对象建模人员也可以从同样的问题开始。然而，对象建模人员必须同时想到人们所关注的项目的抽象。例如，在零售系统中，我们设置了发票购买、购货单等与组织相关的类，所有这些类要被归纳到“商品流动”类当中。这个简单的抽象使得能够将模型的大小降低 30%。因此，在识别人们关注的项目的同时，识别与这些项目相关的抽象也是十分重要的。

下面给出一系列例子以及相应的问题及答案，以此更完整地说明识别类的方法。

例1：一个设备需要维护。在这个特定组织中，所需要的维护产生了一个工作单。这个工作单可以被分割成许多任务，其中每个任务又可继续被分割成若干动作。最后，每个动作又被分割成若干个步骤。这些任务、动作、步骤可以应用到整个设备上，或应用到这个设备的某些部件或子部件上。通常，工作单应用于整个设备，而任务、动作、步骤则应用于该设备的某个部件。

问题：这个例子中相应的类有哪些？

答案：显而易见的类有：

设备 (Equipment)。

部件 (Assembly)。

子部件 (Subassembly)。

工作单 (Work Order)。

任务 (Task)。

动作 (Action)。

步骤 (Step)。

观察整个数据模型，实际上可看到，工作单、任务、动作、步骤的每一个都将与设备、部件、子部件交互作用。这包含了类之间的大量交互作用。

另外两个较为“抽象”的类同样可以识别：

工作 (Work)。

对工作单、任务、动作、步骤的一个综合。

维护对象 (Maintenance Object)。

对设备、部件、子部件的综合。

使用这些类使得我们只关心工作和维护之间的交互作用，而不是在开头提到的 7 个类。这里的要点在于不仅要仔细地识别人们关注的项目，而且要仔细地识别对这些项目的综合。

对每个类有一个精确的定义是十分重要的。一个类总是对应于真实世界的某一事物，对这个类给出好的定义比对它命名更重要。当与客户一道工作时，常常愿意让客户给类任意取一个他们喜欢的名字。类名中的字可能在开发这个系统的组织中具有特殊的含义。对这个类的描述则精确地确定了该类所代表的事物。在建模过程中，应尽量在开发组开始考虑类的最佳名字之前就给出该类的描述。在还未确定一个类精确代表什么之前，试图给出该类的名字是没有意义的。在建立数据模型时应坚持在有关类名的任何争论发生之前就对所有的类加以描述。

显而易见，类描述必须精确地描述这个类。记住，一个类代表真实世界的一个事物，因此，类描述说明的是现实世界的一个事物，而非数据库中的一个表。也许你经常看到以 “This table contains information about... (这个表包含关于...的信息)” 开头的描述，这种描述 (尽管本质上并无错误) 通常意味着接下来的文字不是描述一个真实世界的对象。好一点的段落描述以 “Every object in this class is a ... (该类中的每一个对象是...)” 开始，这个短语更为自然地引导设计者描述一个真实世界的对象。

类描述中还应该包含使设计者能够理解这个类所必需的相关信息。如果该类是过载的（即它存放不止一种对象），那么类描述必须覆盖所有能被存放于该类的对象类型。如果某些特定的属性能修改类中对象的含义，那么这些说明也可放在类描述中。

在类描述中不要引用类名，以免类名在将来改变。

例2：一个零售商的“销售（Sale）”类的描述：“将商品卖给顾客的一次销售活动。注意，如果是现金交易，则不需说明顾客。这个类包括所有种类的销售活动（如现金、信用卡，以及赊帐），它们通过销售种类来加以区分。错误输入的销售活动不被删除，但其状态变成‘ERROR’。购买的商品存放在明细表中。”

注意，假定短语“Every object in this class represents a...（该类中每个对象代表...）”是每个类描述的一部分，在此被省略掉了。

识别组织中的真实事物并将之与类联系起来是十分关键的，因为我们正试着识别该组织关注的项目或动作。设计者应该从模型化该组织的角度考虑问题，而不是从设计一个数据库的角度考虑问题。

一个常见的错误是将文件从遗留系统直接翻译成类。遗留文件（甚至那些来自于关系系统的文件）通常并不等同于类。不加分析地使用遗留文件会导致设计者重复其所包含的所有错误和不足之处。对遗留系统必须仔细分析，它是系统需求的主要来源。然而，不应简单地认为遗留表能直接映射为类。

另一个错误是将一个事务产生的文档模型化为同事务类本身一样——例如，建立一个用于提货单的类而非发货单的类。通常，一个组织的提货单、包装单、购货单以及发票都不是合适的类（尽管它们可能一一对应于该事务恰当的相关类）。购买、发货、购买请求以及资金请求都是可能有兴趣跟踪的真实事务，因而适于被模型化为类。

当然，具有称为“发票（Invoice）”或“购货单（Purchase Order）”的类也是可能的，只要记住它们代表与“付款请求（requests for payments）”和“购买请求（requests for purchases）”相联系的事务。前面讲过，我们更关心类的描述，而不是给它起的名字，但我们不推荐建立这样的类。通常，同时跟踪事务和文档是十分重要的。如果用文档名代替事务，则很容易忘记其实并没有模型化文档，而仅仅模型化了事务这一事实。

除了跟踪事务之外，跟踪与文档相联系的活动也是十分重要的。例如，在一家贷款公司中，跟踪与某个特定用户相关的每一款发票、信件、业务往来以及这些文档所包含的信息是十分重要的。在这种情况下，建立用于跟踪实际文档的类是合适的，因为已经认识到被跟踪的是文档和事务。总之，跟踪文档是合适的，但一定要确保同时也模型化了相应的事务，且清楚地理解了二者的区别。

例3：假设客户是IRS，纳税申报单最初是由纳税人上交的。此外，人们可能会对他们的纳税申报单提出修改意见。他们向我们付款，我们付给他们偿还金。如果偿还金被纳税人丢失了，我们还要重发。

问题：相关的类有哪些？

答案：这个问题实际上比看上去复杂一些。关键在于要注意我们实际上是在讨论两件事情：纳税申报单和个人税责。文档、付款，以及偿还金都影响税责。相关的类有：

纳税人（Taxpayer）

纳税申报单（Tax Return）

付款 (Payment)。

偿还 (Refund)。

偿还请求 (Request for Refund)。

年税责 (Annual Tax Liability)。

再次需要更为抽象地考虑这些类，才能提出“年税责”类。

4.2 识别属性

注意，对于类，潜在地有两组属性。第一组是传统地将之视为关系表中属性的属性。这种属性是类的一项特性，它与类中每个对象的该项特性的特定值相联系。例如，“人 (Person)”这个类，属性可以是“年龄 (Age)”，它与类中每个人的一个特定数值 (年龄) 相联系。属性从特定域中取值，域可以是相对概指的，如年龄域可为“0~150之间的正整数”；域也可以是特别专指的，如性别域“男/女”。域将在第6章详细讨论。

第二组属性是正在开发应用程序的人 (在对象术语中称之为“客户”) 会视之为类属性的属性。由于封装的缘故，有可能完全隐藏实际的属性 (因为它们存在于对象表中) 而向客户显示一组不同的属性。例如，可以存储“生日 (DATE_OF_BIRTH)”作为“雇员 (Employee)”类的一个属性，但也许希望在使用雇员类时只是利用年龄。通过方法中的参数，可以提供对年龄的存取，而将生日信息存在实际实现的表中。

一个类可以有多个与之关联的属性，一个简单的类可以取 2~10个任意属性。复杂表可能含有100甚至更多个属性。通常，具有大量属性的类是不良设计的产物。在设计良好的系统中，比较典型的情况是类具有 50或更少的属性。确定类合适的属性比表面上看更具挑战性。识别明显与该类相关联的属性是十分重要的。如果不适当地处理了一个属性，它将使数据模型难于维护，甚至不可能维护。通常，错误地处理属性是由于理解类定义不准确造成的。

4.3 识别主码

主码是一个或多个属性值的组合，这些属性组合地或者单个地标识类中的一个对象。主码的各个成分均不允许为空。由于物理实现上的原因，主码不允许修改。在使用类进行对象建模的过程中，识别决定主码的一个或多个属性仍然是合适的。有了 Oracle 8的对象扩充后，对主码不可改变的要求可稍稍放松。如果你正在使用传统的结构实现数据库，则主码是不允许改变的。Oracle 8中新加的特性OID (对象标识) 是C++或JAVA应用程序用来在独立的对象类之间导航的逻辑指针。OID允许在外部应用程序中操纵存储在数据库中的数据。

例4：一个组织需要为其系统中的类确定合适的主码。

问题：对于一个组织，“人 (Person)”这个类的主码是什么？

答案：有好几个明显的答案，但它们的大多数都是错的：

姓名不是理想的主码，因为不同的人可能会有同样的名字。

社会保障号码也不适合作为主码。原因是多方面的：首先，有些人可能没有社会保障号码，如在其他国家中为某个跨国机构工作的雇员。第二，可能需要在不知道社会保障号码的情况下将一个雇员的资料输入系统。第三，数字可能会输错，从而需要后续的修改。第四，可能对社会保障号码有隐私上的考虑，通常这些号码对大多数用户是隐藏的。在传统的关系系统中，主码被传送到系统的许多地方，因而损害了系统的安

全性考虑。最后一点，社会保障号码并不总是唯一的。尽管极其罕见，但在美国的确有人具有重复的社会保障号码。

“人 (Person)”这个类唯一有资格作为主码的是系统生成的数字。在大多数机构中，这个数字就是雇员的标识号。

有时，确定主码会非常具有挑战性，下例就是一个良好的说明。这是一个非常有趣的例子，因为答案看上去十分容易。然而，大多数设计者开始时都选择了不正确的主码候选方案。

例5：某组织希望跟踪打电话的费用，并将电话呼叫与特定的项目联系起来。为此建立了一个称为“电话呼叫 (Telephone Call)”的类。

问题：什么是“电话呼叫 (Telephone Call)”类合适的主码？

答案：有几个备选方案可供考虑：

打电话的人。

接电话的人。

呼叫方的电话号码。

被呼叫方的电话号码。

呼叫的日期/时间。

最常见的错误是将呼叫方的电话号码和被呼叫方的电话号码组合在一起作为主码。之所以这是错误的是因为这样做意味着某个特定的电话只能拨另一个特定的电话一次。另一个常见的错误是将被呼电话号码与呼叫时间放在一起作为主码。初看起来，这似乎是正确的，但它意味着一个组织中的两个人不能同时呼叫同一电话号码（而这完全是可能的，除非“时间”作为一个连续变量来记录）。“电话呼叫 (Telephone Call)”类合适的主码应由呼叫方电话号码和该呼叫的日期及时间组合而成，这在逻辑上讲得通，但并未提供直观上吸引人的唯一标识 (UID)。在关系系统中，为了计费目的而将电话呼叫分配给一个或多个帐号需要额外的工作。通过使用对象标识 (OID) 或系统产生的标识作为主码，这些问题得以消除。

使用OID还有其他好处。如果对数据库进行会影响 UID的结构修改，而又没有使用系统产生的标识，则这些变化将会渗透到其他表和约束中。这将影响触发器、应用程序以及 UID生成之后出现的其他工作。如果使用的是生成的标识，修改可以正常进行且不会传播到系统中。对于参照表（值列表类），使用顺序生成的标识是没有道理的，它们明显不会变化，或者变化不频繁。这些类的UID应该是码属性。

确定合适的主码对理解类是至关重要的，确定主码的过程对系统中类的精确定义又做了一次检查。对于定义得不好的类，确定合适的主码是一件困难的工作。同时，如果主码没有适当地指定，在类中寻找一个特定的对象也是困难的，甚至是不可能的。在一个类中可以选出好几个可能的主码也并不罕见，关系理论中，这些可供选择的码叫作候选码。

4.4 规范化仍然很重要

既然有了OID，为什么还要担心规范化呢？原因是规范化给设计过程引入了精确级别的概念。规范化的作用就像一次测试，确保我们正确处置了属性。而且，它有助于我们确定类被正确地说明了。在进行面向对象数据库的设计时，最重要的因素之一就是属性的正确处置。属性的正确处置完全依赖于设计人员的经验和技巧，而没有一个全局的指导理论的支持。

一些关系模型设计者在与对象模型设计者相似的环境下工作一段时间后，他们对所有的

表都使用系统产生的主码。然而在审计这种系统时，常常发现其建模错误高于其他严格的关系型系统。去掉逻辑主码，不考虑规范化问题似乎助长了不良的建模技巧。因此，我们提倡设计者为所有类都定义逻辑主码，就像这些类是传统的关系实体一样，即便他们打算使用对象技术来实现该模型。

建立面向对象的模型并不意味着规范化原则不适用了。对数据库的存取仍然要使用 SQL 完成。SQL 从数据库中提取任意子集的能力依赖于数据库中类或表的规范化结构，因此，规范化原则仍然要遵循。当属性加到类中时，注意不要破坏这些原则。

4.4.1 违反第一范式

正如在第2章中讨论的那样，当你加入多个属性，而这些属性是同一性质的重复度量时，就违反了第一范式，例如，如果你有一个购货单且该购货单中只有很少项，即项 1/量1~项5/量5 (Item 1/Quantity1~Item 5/Quantity 5)，这就是对第一范式的违反。违反第一范式会使得信息的聚集变得困难，这需要五个信息源。这个问题可通过在表上建立视图得以灵活地解决。当需要在购货定单中加入第六项时，就会出现极为严重的问题。这就是我们努力避免违反第一范式的主要原因。

关于一种类存在一场长期的争论。这个类存储一个部门每周的销售额，它有七个属性——每个属性对应于一周中某一天的销售数字。这是否违反了第一范式？从理论的角度看，这绝对没有违反第一范式。一周中正好有七天，每天都各不相同。然而，用这种方法建立数据结构带来许多与非第一范式结构相同的问题。因此，从实现的角度看，存储诸如一周中若干天之类的信息作为属性不是最佳的建模方法。决定是否使用这种结构纯粹是基于物理实现上的优化，而不是违背了规范化原则。

Oracle 8的性能特征

Oracle 8提供两种新技术用于非第一范式设计的实现：嵌套表和 VARRAYS。这两者都提供将相关数据紧密地存放在一起的能力，并提供高效率的检索。这些结构与将一对多联系存放在分离但有联系的表中是类似的。

嵌套表中的对象与它们所依附的主对象是独立存放的，需要索引来提高检索性能。然而，嵌套表中并不真正存在独立于嵌套表赖以起源的主对象类的对象。因而，只能通过主对象类中的对象存取与这些对象相连的嵌套表中的对象。正如第1章中说明的那样，可以对存放在嵌套表中的行“去除嵌套”，但仍然必须在查询中指明主表。Oracle 8的当前版本只支持一级嵌套，换句话说，一个嵌套表本身不能再含有一个嵌套表。

VARRAYS，即变量数组，已作为新的数据类型实现了。VARRAYS可以支持前面记录每日销售数字的例子。为了满足这一要求，可以建立一个基于 VARRAY的对象类型。本例中，该VARRAY有7个事件，每个事件对应于一周中的一天，称之为 DAILY_RECORDING。此外，再创建一个称为 WEEKLY_TEST的表，该表中有一列是这种新的数据类型，其目的是跟踪每日记录值，如代码4-1所示。

与嵌套表相似，VARRAYS不能嵌套自身。它们包含的数据也是在主对象类的对象范围之外所无法存取的。VARRAYS不能被索引，尽管你不太会想这么做，因为它们通常用于小或中等大小的重复组。同时，SQL的当前版本不能存取存放在 VARRAYS中的数据，因此当唯一的存取路径是PL/SQL时，应该实现VARRAYS。

代码 4-1

```
CREATE OR REPLACE TYPE DAILY_RECORDING
AS VARRAY (7) of number (5,2)
/

CREATE TABLE WEEKLY_TEST (
WEEK_ID      NUMBER (10) NOT NULL,
DAILY_NBR    DAILY_RECORDING)
/
```

4.4.2 违反第二范式

第二范式是另一种不同的规则，有时可能违反了它但不会产生影响。在第 2 章曾解释过，当有一个多列主码时，第二范式问题就出现了。当有仅仅依赖于主码一部分的属性时，就违反了第二范式。如果所有属性均只依赖于主码的一部分，这就意味着主码中有额外的列。当非码属性的一个子集依赖于主码的一部分时，问题就出现了，这表明或者该属性放到了一个错误的类中，或者存在一个更为严重的问题：你没有建立一个必需的类。

4.4.3 违反第三范式

当一个属性依赖于一个非码属性时，就违背了第三范式。违背第三范式总是灾难性的，应尽可能地避免。例 6 是违反第三范式的一个实例。

例 6：这里有一个“SALE（销售）”和“SALE DETAIL（销售明细）”表，它的结构如下：

SALE

Sale Number（销售编号） **PK**

Transaction date（交易日期）

ID of Employee who sold the goods（销售人员标识）

Name of Salesperson 销售人员姓名

SALE DETAIL

Sale Number（销售编号） **PK, FK(sale)**

Sale Detail Number（销售明细编号） **PK**

Item Number（项目号）

Quantity（数量）

Item Style（项目格式）

Item Model（项目型号）

问题：这些属性中哪个违背了第三范式？

答案：一共有三个属性违犯了第三范式。项目型号和项目款式只依赖于项目号，销售人员姓名只依赖于销售人员标识。这些属性应该分别移到“库存（Inventory）”类和“人（Person）”类中。

4.5 值列表类

存储一个属性的有效值列表通常是有用的。有效值列表者若能随时间而变化将是特别重

要的。因此，建立一个类来存储性别是没有意义的。然而，仅仅因为一个经理的突发奇想而重新定义工作流，一个项目的有效状态就可能轻易地改变。在这种情况下，将合法的值存放在一个类中是合适的。为了区分这种类和主类，这种类叫做值列表（VL）类。

普遍使用的VL类包括颜色（Color）、状态（Status）和类型（Type）。它们各自存储一个属性域的所有可能值。有时为日期（Date）建立一个域类也是合适的，这需要存储与年历相关的信息，如周的结束、月的结束、财政季度结束、财政年、指定日期的计算机生产运营状态，等等。这样的VL类常常能通过数据库函数来支持。然而，诸如生产运营状态这样的属性不能通过函数来支持。不过，其他属性的确提供数值。

在数据库界，有一场关于某个事物是属性（带有值检查约束列表）还是VL类的争论。不知道为什么，许多时间花在争论某个特定项在建模时应该作为属性还是作为VL类上了。这是一个相对简单的设计问题，只要问一下：“保留域中合法值的列表是否合适？”如果答案为“是”，则你应该使用VL类；若答案为“否”，则不需要这样的类。VL类是很常见的，通常占数据模型中50%以上的比重。第7章会对它们做详细的讨论。

4.6 实体/类的物理实现

本章前面解释过，在某种程度上，实体和类是“需要跟踪的事物”。Oracle Designer保持了用于逻辑和物理数据建模的两个截然不同的建模方法，ODD代替实体关系建模技术而UML代替逻辑建模技术。ODD的服务器模型允许将关系结构和对象关系结构合并到一个模型中。关于ODD的进一步讨论可见第19章。

4.6.1 关系的实现方法

在Oracle的关系体系结构中，实体及其属性一一对应演变为表和列——子类型除外，一般倾向于将多个子类型综合成一个属类结构。

以美国的州为例，在各式各样的系统都需要记录这些数据。下面列出了要存入STATE表的一部分数据。紧随其后在代码4-2中提供了定义STATE表的语法，它与STATE实体相对应。

STATE_CD	DESCR_TX
NJ	New Jersey （新泽西）
MD	Maryland （马里兰）
PA	Pennsylvania （宾夕法尼亚）
CA	California （加利福尼亚）
FL	Florida （佛罗里达）

代码 4-2

```
CREATE TABLE STATE(  
STATE_CD VARCHAR2(5) NOT NULL PRIMARY KEY,  
DESCR_TX VARCHAR2(40) NOT NULL UNIQUE)  
/
```

州代码存放在STATE_CD列，而州名存放在DESCR_TX列。注意STATE_CD列的长度为5，比州代码值的长度要多3个字符。有人会说STATE_CD列的正确长度应该为2，然而，此系统

中“代码”列全都使用长度为5的VARCHAR2类型。任何建立于此表上的应用程序可以将该列的显示字符数限制为2，从而消除对用户在该列中输入错误数据的担心。第6章对域的物理实现进行了更全面的讨论。

注意，在STATE实体及其属性中声明的命名约定已用于对STATE表及其列的命名。在某些情况下，对实体及其属性使用全名，然后在创建表和列定义时应用一组标准的缩写会好一些。我们发现，一组清晰的缩写（每个单词缩写为5个或更少的字母）可以用于逻辑的和物理的数据结构。关于命名约定的详细讨论见第5章。

此外还要注意数据类型及长度的定义。在属性这一级，可以使用域来定义数据类型、长度及精度。这是一个有用的技巧，用于保证整个数据模型中数据类型定义的一致性。域可以在Oracle Designer中定义来一般化列结构，但这只是元数据，因而在实际的CREATE TABLE语句中是不可用的。另外，不能通过使用一个Oracle Designer域来定义一个指定的列是否为强制的，尽管这显然是该属性的一个附加特性。

可以在数据库中创建CODE类型的物理域，然后该域可以应用到每个具有VARCHAR2(5)数据类型的列的定义中。如果用户定义的域不需要在数据类型和最大长度之外做说明的话，它们可以发展成为对象类型。当前版本中，此范围之外不能够说明任何东西。为了将数据行插入STATE表，使用如下RDBMS语句，如代码4-3所示。

代码 4-3

```
INSERT INTO STATE (STATE_CD, DESCR_TX
VALUES ('NJ', 'New Jersey')
/
```

如果给出了表中每一列的值，那么无需列出想插入的列。我们提倡这样做；否则，必须指明给出了值的列。所有强制型的列都必须提供值，除非在表定义中指定隐含值。

若要从STATE表中查出“New Jersey”，使用如下SELECT语句，如代码4-4所示。

代码 4-4

```
SELECT DESCR_TX
FROM STATE
WHERE STATE_CD = 'NJ'
/
```

本例的SELECT语句使用基本的Oracle 7语法，其中的WHERE子句用来过滤该查询返回的记录个数。

4.6.2 对象-关系的实现方法

Oracle 8在定义数据结构的功能上有很大提高。在数据结构的物理定义上首先一个显著的区别就是最初可以为对象定义一个类型。这些对象类型可以通过3种方式实现：

作为用于列的域。

作为对表的扩展，该扩展加入一组公共的列。

作为一个对象类结构的完整定义。

第二个区别是UML提供将两个对象类关联起来的不同选择方案。下面将分别详细地解释

这些领域，最后将讨论使用对象类型和类的一些总体缺点。

1. 创建对象类型

对象类型的功能是定义要创建的类的基本结构，特别是与该类相关的属性的格式以及这些属性各自的数据类型。不用指定对象类型的可选择性，但可指定在这些对象定义上的可选择性。注意，不需要区分描述对象元素的逻辑的（属性）和物理的（列）实现。相反，可以简单地将它们都称为属性，而不考虑项目的阶段。Object类等价于TYPES，而不是表。每一种类型的实例可以存储在多个表或多个列中。下面的代码 4-5是类类型定义的一个例子。

代码 4-5

```
CREATE OR REPLACE TYPE STATE_TYPE
AS OBJECT
(STATE_CD  VARCHAR2(5),
DESCR_TX  VARCHAR2(40) )
/
```

STATE_TYPE类类型将作为定义STATE对象的基础。STATE定义声明我们需要两列：代码列用于唯一地识别STATE类的每一次出现；文本列用于对每个州做更详细的描述。

2. 创建类

类可以按两种不同的方式来创建：显式和隐式。显式的方法是使用关系数据库中的CREATE TABLE语句定义类、类中的列及其数据格式信息，这和 Oracle 7的语法是一样的。用显式方法创建类的例子如代码 4-6所示。

代码 4-6

```
CREATE TABLE STATE(
STATE_CD VARCHAR2(5) NOT NULL,
DESCR_TX VARCHAR2(40) NOT NULL )
/
```

显式创建表与在Oracle 7中使用的方法本质上是一样的。给表及其列命名，定义数据类型及其可选择性，所有这些均在同一语句中完成。

隐式方法是基于某个对象类型来定义一个类的。下面的代码 4-7显示了STATE表的建立过程，它基于前面第1项“创建对象类型”部分声明的STATE_TYPE。

代码 4-7

```
CREATE TABLE STATE OF STATE_TYPE
(STATE_CD  NOT NULL PRIMARY KEY,
DESCR_TX  NOT NULL)
/
```

在STATE表中有两列，STATE_CD和DESCR_TX，它们的特性是从STATE_TYPE对象类型定义中继承过来的。隐式的STATE表表面上和行为上与显式的STATE表几乎完全一致，区别在于以STATE_TYPE对象类型为基础，可以创建多个与STATE类具有相同结构的类。

不管是显式还是隐式，不包含对象参照的表可以使用代码 4-3加入数据。

我们可能已经创建了一个较为概括的类类型，如Territory或TERR_TYPE，它与STATE_TYPE相对。定义一个带有总括名字如TERR_TYPE的对象类型的原因是可以使用其结

构作为定义其他许多地理区域如县、省、国家等的基础。通过这种方法，可以将单一一个对象类型包含在许多类的定义中。

使用Oracle Designer可以在列这一级定义域。域是为了将数据插入到一列中，其所必须具有的特征的总定义。某些特征可能是一组合法值，如状态代码，其中“ O ”代表开放，“ C ”代表已关闭。值区间可以使用上限和下限来指定。此外，还有其他一些有用的数据特征，它们被定义成域的组成部分，如可选择性、格式、结构等。Oracle 8提供了有限的能力来实现域，尽管域还不能超越基本数据类型的能力。然而，一旦另一个对象类参照了某个对象类型，就不能再在该对象类型中加入一个新列或修改一个已经存在的列了。这些功能以及其他一些功能将在Oracle 8的后续版本中推出。第6章将对此做更详细的讨论。