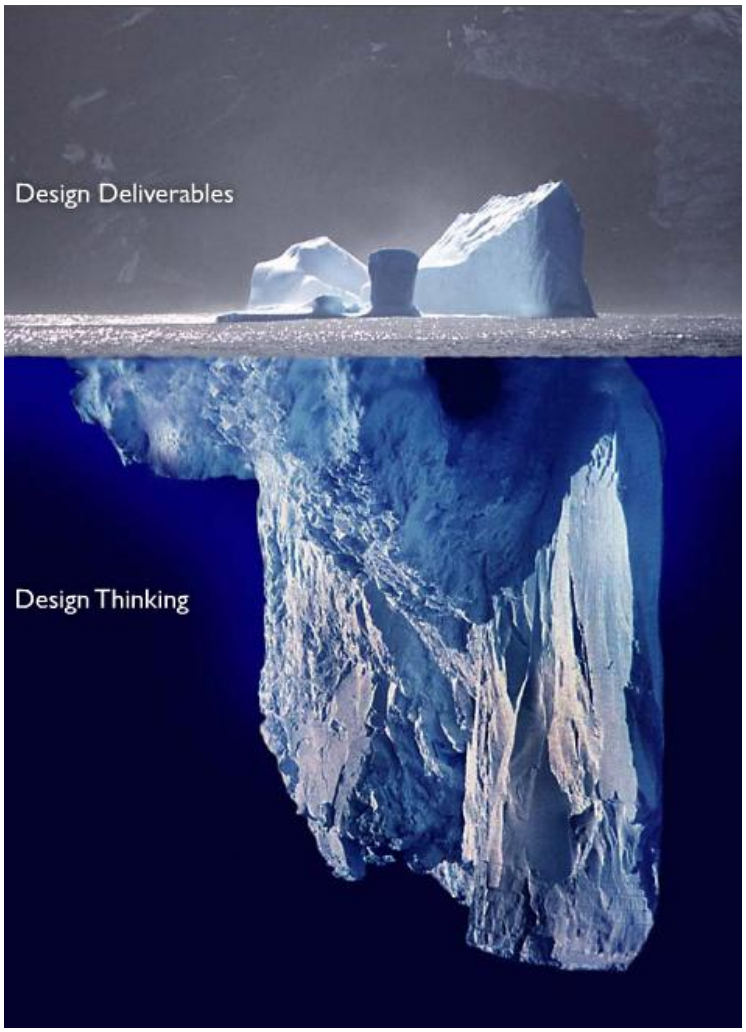
数据库设计 Step by Step

——关系数据库逻辑设计起步



作者：DBFocus
博客：www.cnblogs.com/DBFocus
邮箱：shadow_wu82@163.com

目录

数据库设计 Step by Step (1)——扬帆启航 3

数据库设计 Step by Step (2)——数据库生命周期 8

数据库设计 Step by Step (3)——基本 ER 模型构件 15

数据库设计 Step by Step (4)——高级 ER 模型构件 21

数据库设计 Step by Step (5)——理解用户需求 27

数据库设计 Step by Step (6) —— 提取业务规则 34

数据库设计 Step by Step (7)——概念数据建模 38

数据库设计 Step by Step (8)——视图集成 46

数据库设计 Step by Step (9)——ER-to-SQL 转化 53

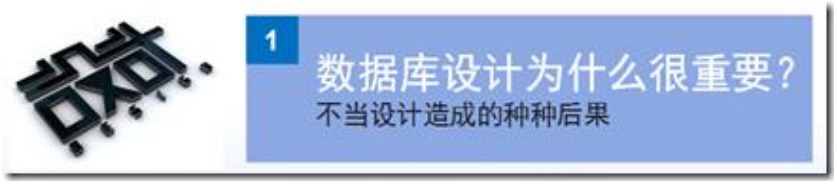
数据库设计 Step by Step (10)——范式化 70

数据库设计 Step by Step (11)——通用设计模式 80

数据库设计 Step by Step (1)——扬帆启航

引言：一直在从事数据库开发和设计工作，也看了一些书籍，算是略有心 得。很久之前就想针对关系数据库设计进行整理、总结，但因为种种原因迟迟没有动手，主要还是惰性使然。今天也算是痛下决心开始这项卓绝又令我兴奋的工作。 这将是一个系列的文章，我将以讲座式的口吻展开讨论（个人偷懒，这里的总结直接拿去公司培训新人用）。

系列的第一讲我们先来回答下面几个问题



数据库是大楼的根基

大多数程序员都很急切，在了解基本需求之后希望很快的进入到编码阶段（可能只有产出代码才能反映工作量），对于数据库设计思考得比较少。

这给系统留下了许多隐患。许多软件系统的问题，如：输出错误的数据，性能差或后期维护繁杂等，都与前期数据库设计有着密切的关系。到了这个时候再想修改数据库设计或进行优化等同于推翻重来。

我经常把软件开发比作汽车制造。汽车制造会经过图纸设计，模型制作，样车制造，小批量试生产，最后是批量生产等步骤。整个过程环环相扣，后一过程是 建立在前一过程正确的前提基础之上的。如果在图纸设计阶段发现了一个纰漏，我们可以重新进行图纸设计，如果到了样车制造阶段发现这个错误，那么我们就要把 从图纸设计到样车制造的阶段重来，越到后面发现设计上的问题，所付出的代价越大，修改的难度也越大。

数据库是整个应用的根基，没有坚实的根基，整个应用也就岌岌可危了。

强大的数据库面对不良设计也无能为力

现代数据库管理系统（DBMS）提供了方便的图形化界面工具，通过这些工具可以很方便的创建表、定义列，但我们设计出的结构好吗？

关系数据库有许多非常好的特性，但设计不当会使这些特性部分或完全的丧失。

我们来看看以下几个数据库不良设计造成的场景：

1. 数据一致性的丧失

一个订单管理系统，维护着客户和客户下的订单信息。使用该系统的用户在接到客户修改收货地址的电话后，在系统的客户信息页面把该客户的收货地址进行了修改，但原先该客户的订单还是送错了地址。

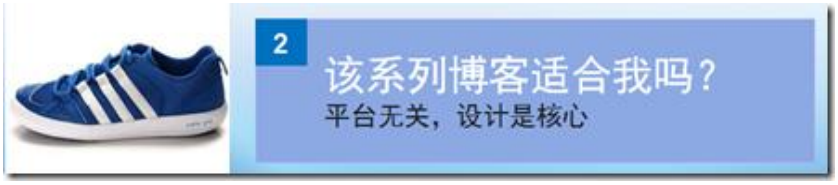
2. 数据完整性的丧失

公司战略转移，准备撤出某地区。系统操作人员顺手把该地区的配置信息在系统中进行删除，系统提示删除成功。随后问题就来了，客服人员发现该地区的历史订单页面一打开就出错。

3. 性能的丧失

一个库存管理系统，仓库管理员使用该系统记录每一笔进出货情况，并能查看当前各货物的库存情况。在系统运行几个月后，仓库管理员发现打开当前库存页面变得非常慢，而且整个趋势是越来越慢。

上面这些场景都是由于数据库设计不当造成的，根源包括：设计时引入了冗余字段，没有设计合理的约束，对性能没有进行充足设计等，上面的例子也只是沧海一粟。



数据库平台无关性

我在这个系列博客里讨论的数据库设计不针对任何一个关系数据库产品。无论你使用的是 Oracle，SQL Server，Sybase，亦或是开源数据库如：MySQL，SQLite 等，都可以用来实践我们这里讨论的设计方法和设计理念，设计是这个系列博文的核心和灵魂。

注：在文中我会选用一个数据库产品来进行演示，大家可以选用自己熟悉的数据库产品来实验。本文最后会给出一些免费数据库产品的链接，大家可以下载学习。

一起学习共同进步

无论你是数据库设计师，应用架构师，软件工程师，数据库管理员（DBA），软件项目经理，软件测试工程师等项目组成员，都能从该系列博文中有所收获。大家一起讨论，共同进步。

内容涉及领域

我对这一系列博文现在的设想是涉及数据库设计的整个过程。从需求分析开始，到数据库建模（概念数据建模），进行范式化，直至转化为 SQL 语句。

在我们一头扎进数据库设计之前，我们先了解一下除了关系型数据库之外的数据存储方式。

平面文件（Flat File）

包括以.txt 和.ini 结尾的文件。

eg: 一个.ini 文件的内容：

[WebSites]

MyBlog=http://www.cnblogs.com/DBFocus

[Directorys]

Image=E:\DBFocus Project\Img

Text=E:\DBFocus Project\Documents

Data=E:\DBFocus Project\DB

优点：

文件的存储形式非常简单，普通的编辑器都能对其进行打开、修改

缺点：

无法支持复杂的查询

没有任何验证功能

对平面文件中间的内容进行插入、删除操作其实是重新生成了一个新文件

适用场景：

存放小量，修改不频繁的数据，如应用配置信息

Windows 注册表

错误的修改 Windows 注册表会引起系统的紊乱，故不建议把很多数据存放在注册表中。

Windows 注册表为树形结构，存放着一些系统配置信息和应用配置信息。

通过把不同的配置存放在注册表的不同分支上，使得应用程序公共配置信息与用户个人配置信息分离。

eg：某文档版本管理系统，能通过配置与本主机上安装的文件比较器建立关联进行文档比较。这是一个公共配置信息，文件比较器路径可以存放在注册表的 HKEY_LOCAL_MACHINE\SOFTWARE 分支下。

同时该文档版本管理系统能记录用户最近打开的 10 个文档路径。这是用户个人配置信息，对于不同的 Windows 用户最近打开的 10 个文档可以不同，这些配置信息可存放在注册表的 HKEY_CURRENT_USER\Software 分支下。

Excel 表单（Spreadsheets）

优点：

Excel 非常普及，用户对于 Spreadsheet 的表现形式非常熟悉

可以进行简单统计，方便出各种图表

缺点：

不适用于许多 Spreadsheet 之间关系复杂的情况

无法应对复杂查询

数据验证功能弱

适用场景：

数据量不是非常大的办公自动化环境

XML

XML 是一种半结构化的数据。相比于超文本标记语言（HTML），其标签是可以自行定义的，即可扩展的。

eg：一个 XML 文件内容

```
----------
<?xml version="1.0" encoding="UTF-8" ?>
<ClassSchedule>
    <Class Name="Psychology" Room="Field 3">
        <Instructor>Richard Storm</Instructor>
        <Students>
            <Student>
                <FirstName>Ben</FirstName>
                <LastName>Breaker</LastName>
            </Student>
            <Student>
                <FirstName>Carol</FirstName>
                <LastName>Enflame</LastName>
                <NickName>Candy</NickName>
            </Student>
        </Students>
    </Class>
</ClassSchedule>
----------
```

XML 文件有几个特点。

首先，XML 标签要求严格对应，且不能出现交错的现象。

其次，XML 文件必须有一个根节点，该节点包含所有其他元素。

第三，同级别的不同节点内不必包含相同的元素，如上例中第二个学生 Carol 有一个特别的节点 NickName。这个特性使得在某些场景中 XML 比关系数据库更能应对变化。

优点：

自然的层次型结构

文本内容通过标签是自解释的

通过 XSD（XML Schema 语言）可以验证 XML 的结构

有许多辅助型技术如：XPath, XQuery, XSL, XSLT 等

一些商业数据库（如 Oracle，SQL Server）已支持 XML 数据的存储与操作

缺点：

数据的冗余信息较多

无法支持复杂的查询

验证功能有限

对 XML 中间的内容进行插入、删除操作其实是重新生成一个新文件

适用场景：

适合存放数据量不大，具有层次型结构的数据，如树形配置信息

NoSQL 数据库

非关系型数据库我接触的不是很多，除了给出一些产品名称之外不做很多展开。园子里已有一些文章，本文最后也给出了链接供大家学习、研究。

1. Key-Value 数据库

   Redis, Tokyo Cabinet, Flare

2. 面向文档的数据库

   MongoDB, CouchDB

3. 面向分布式计算的数据库

   Cassandra, Voldemort

这几年 NoSQL 非常热。我认为 NoSQL 并不是“银弹”，在某些 SNS 应用场景中 NoSQL 显示了其优越性，但在如金融行业等对数据的一致性、完整性、可用性、事务性高要求的场景下，现在的 NoSQL 就未必适用。我们应充分分析应用的需求，非常谨慎地选择技术和产品。

主要内容回顾

1.数据库设计对于软件项目成功的关键作用

2.本课程与数据库产品无关，核心是设计的理念和方法

3.各种数据存储所适用的场景

参考资料

1. *Oracle* Database 10g *Express* Edition

2. *SQL Server* 2008 R2 *Express* – Overview

3. *SQLite* Home Page

4. *NoSQL* 数据库笔谈

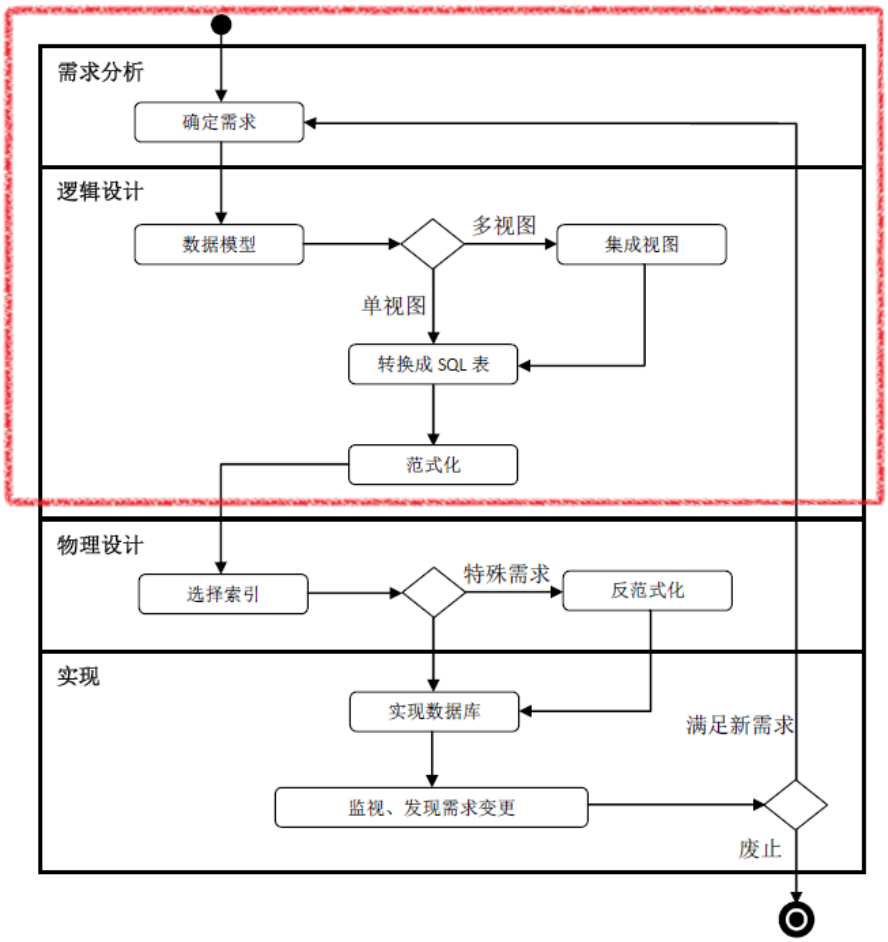
数据库设计 Step by Step (2)——数据库生命周期

引言：数据库设计 Step by Step (1)得到这么多朋友的关注着实出乎了我的意外。这也坚定了我把这一系列的博文写好的决心。近来工作上的事务比较繁重，加之我期望这个系列的文章能尽可能的系统、完整，需要花很多时间整理、思考数据库设计的各种资料，所以文章的更新速度可能会慢一些，也希望大家能够谅解。

系列的第二讲我们将站在高处俯瞰一下数据库的生命周期，了解数据库设计的整体流程

数据库生命周期

大家对软件生命周期较为熟悉，数据库也有其生命周期，如下图所示。



（图 1　数据库生命周期）

数据库的生命周期主要分为四个阶段：需求分析、逻辑设计、物理设计、实现维护。

这个系列的博文将主要关注数据库生命周期中的前两个阶段（需求分析、逻辑设计）。如图中红色框出的部分。

数据库的物理设计，包括索引的选择与优化、数据分区等内容。这些内容也非常丰富，而且可以自成体系，园子里也有很多好文章，故在本系列中不作主要关注。本文最后将给出一些链接供大家参考。

数据库生命周期的四个阶段又能细分为多个小步骤，我们配合图（1）来看看每一小步包含的内容。

阶段 1 需求分析

数据库设计与软件设计一样首先需要进行需求分析。

我们需要与数据的创造者和使用者进行访谈。对访谈获得的信息进行整理、分析，并撰写正式的需求文档。

需求文档中需包含：需要处理的数据；数据的自然关系；数据库实现的硬件环境、软件平台等；

（图 2　阶段 1 需求分析）

阶段 2 逻辑设计

使用 ER 或 UML 建模技术，创建概念数据模型图，展示所有数据以及数据间关系。最终概念数据模型必须被转化为范式化的表。

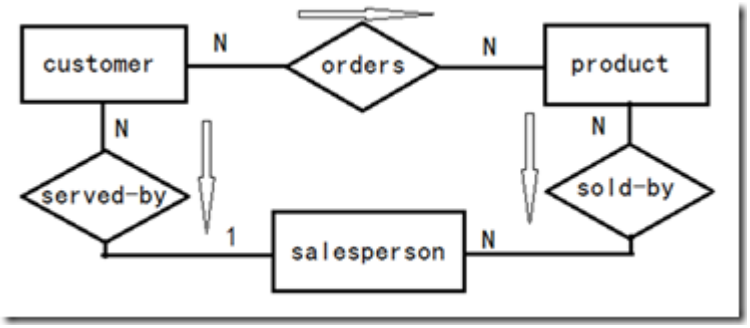
数据库逻辑设计主要步骤包括：

a) 概念数据建模

在需求分析完成后，使用 ER 图或 UML 图对数据进行建模。使用 ER 图或 UML 图描述需求中的语义，即得到了数据概念模型（Conceptual Data Model），例如：三元关系（ternary relationships）、超类（supertypes）、子类（subtypes）等。

<u>eg: 零售商视角，产品/客户数据库的 ER 模型简图</u>

注：ER 图的含义，以及详细标记方法将在该系列的下一篇博文中进行讨论



（图 3　阶段 2(a) 概念数据建模）

b) 多视图集成

当在大型项目设计或多人参与设计的情况下，会产生数据和关系的多个视图。这些视图必须进行化简与集成，消除模型中的冗余与不一致，最终形成一个全局 的模型。多视图集成可以使用 ER 建模语义中的同义词(synonyms)、聚合(aggregation)、泛化(generalization)等方 法。多视图集成在整合多个应用的场景中也非常重要。

eg: 集成零售商 ER 图与客户 ER 图

零售商 ER 图如图（3）所示。客户视角，产品/客户数据库的 ER 模型简图如下：

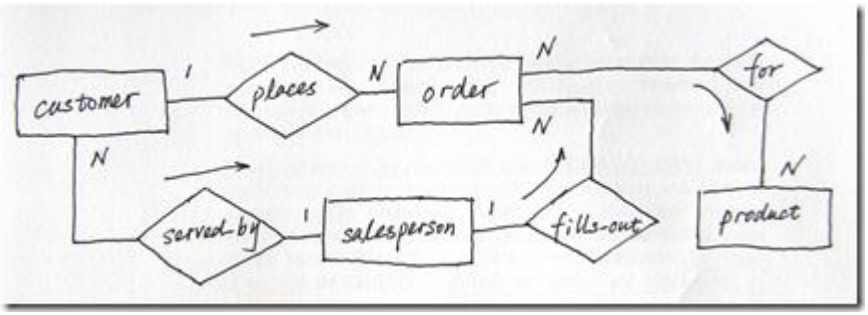
（图 4　以客户为关注点绘制的 ER 图）

注：现在市面上有许多辅助建模工具可以绘制 ER 图。使用 Sybase 的 PowerDesigner 绘制与图（4）相同语义的 ER 图如下：

其标记法与图（4）中略有不同，这将在今后的博文中加以说明。

这里需要指出的是辅助软件的使用不是设计的核心，大家不要被这些工具迷惑。所以后文中我们将主要使用手绘。只要掌握了 ER 图的语义，使用这些软件都不会是件难事。

集成零售商 ER 图与客户 ER 图

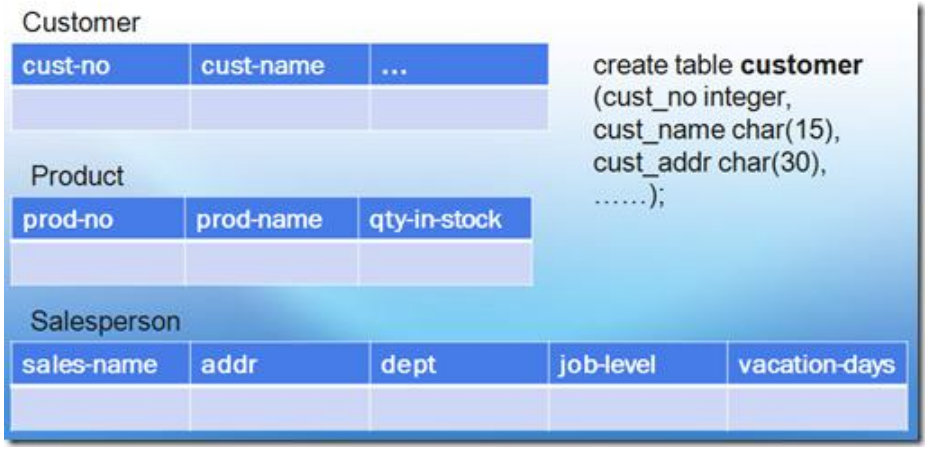


（图 5　阶段 2(b) 多视图集成）

c) 转化概念数据模型为 SQL 表

根据映射规则，把 ER 图中的实体与关系转化为 SQL 表结构。在这一过程中我们将识别冗余的表，并去除这些表。

eg: 把图（5）中的 customer, product, salesperson 实体转化为 SQL 表



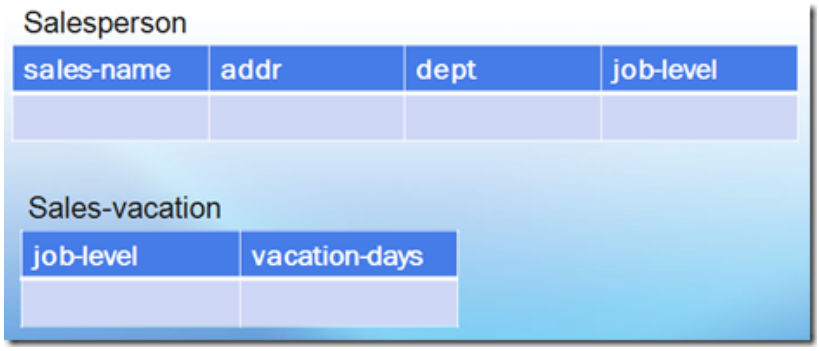
（图 6　阶段 2(c)转化概念数据模型为 SQL 表）

d) 范式化

范式化是数据库逻辑设计中的重要一步。范式化的目标是尽可能去除模型中的冗余信息，从而消除关系模型更新、插入、删除异常（anomalies）。

讲到范式化就会引出函数依赖（Functional Dependency）这一概念。函数依赖(FDs)源自于概念数据模型图，反映了需求分析中的数据关系语义。不同实体之间的函数依赖表示各个实体唯一键 之间的依赖。实体内部也有函数依赖，反映了实体中键属性与非键属性之间的依赖。在保证数据完整性约束的前提下，基于函数依赖对候选表进行范式化（分解、降 低数据冗余）。

eg: 对图（6）中的 Salesperson 表进行范式化，消除更新异常（update anomalies）



（图 7　阶段 2(d)范式化）

阶段 3 物理设计

数据库物理设计包括选择索引，数据分区与分组等。

逻辑设计方法学通过减少需要分析的数据依赖，简化了大型关系数据库的设计，这也减轻了

数据库物理设计阶段的压力。

1. 概念数据建模和多视图集成准确地反映了现实需求场景
2. 范式化在模型转化为 SQL 表的过程中保留了数据完整性

数据库物理设计的目标是尽可能优化性能。

物理设计阶段，全局表结构可能需要进行重构来满足性能上的需求，这被称为反范式化。

反范式化的步骤包括：

1. 辨别关键性流程，如频繁运行、大容量、高优先级的处理操作
2. 通过增加冗余来提高关键性流程的性能
3. 评估所造成的代价（对查询、修改、存储的影响）和可能损失的数据一致性

阶段 4 数据库的实现维护

当设计完成之后，使用数据库管理系统（DBMS）中的数据定义语言（DDL）来创建数据结构。

数据库创建完成后，应用程序或用户可以使用数据操作语言（DML）来使用（查询、修改等）该数据库。

一旦数据库开始运行，就需要对其性能进行监视。当数据库性能无法满足要求或用户提出新的功能需求时，就需要对该数据库进行再设计与修改。这形成了一个循环：监视 –> 再设计 –> 修改 –> 监视…。

在进行数据库设计之前，我们先回顾一下关系数据库的相关基本概念。

这里只做一个提纲挈领的简介，大家可以根据相应的线索进行扩展。

表、行、列

关系数据库可以想象成表的集合，每个表包含行与列。（可以想象成一个 Excel workbook，包含多个 worksheet）。

表在关系代数中被称为关系，这也是关系数据库名称的起源（不要与表之间的外键关系混淆）。

列在关系代数中被称为属性（attribute）。列中允许存放的值的集合称为列的域（域与数据类型密切相关，但并不完全相同）。

行在关系代数中的学名是元组（tuple）。

关系数据库的理论基础来自于“关系代数”。但在关系代数中，一个集合的各个元组没有次序的概念，在关系数据库中为了方便使用，定义了行的次序。

键、索引

键是一种约束，目的是保证数据完整性

1. 复合键（Compound key）：由多个数据列组成的键
2. 超键（Superkey）：列的集合，其中任何两行都不会完全相同
3. 候选键（Candidate key）：首先是一个超键，同时这个超键中的任何列的缺失都会破坏行的唯一性
4. 主键（Primary key）：指定的某个候选键

索引是数据的物理组织形式，目的是提高查询的性能

约束

基本约束

not null constraint, domain constraint

检查约束（Check Constraints）

eg: Salary > 0

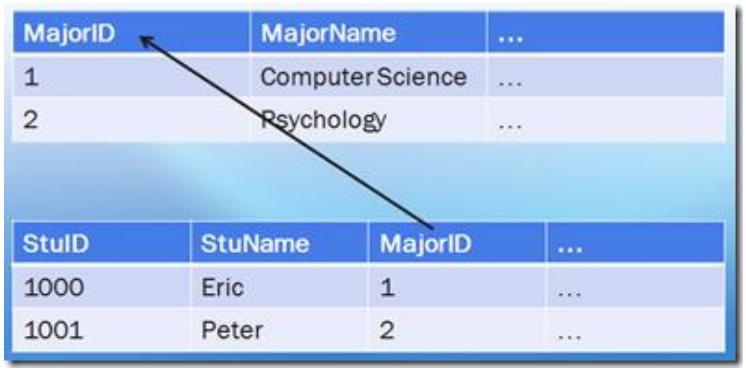
主键约束（Primary Key Constraints）

实体完整性（entity integrity），没有两条记录是完全相同的，组成主键的字段不能为 null

唯一性约束（Unique Constraints）

外键约束（Foreign Key Constraints）

也被称为引用完整性约束，eg:



关系数据库操作

1.选择（Selection）
2.映射（Projection）
3.联合（Union）
4.交集（Intersection）
5.差集（Difference）
6.笛卡尔积（Cartesian Product）
7.连接（Join）

上述 7 种是最基本的关系数据库操作，对应于集合论中的关系运算。

有些书籍中还会加入改名（Rename），除（Divide）等关系操作。

主要内容回顾

1. 数据库生命周期的四个阶段：需求分析、逻辑设计、物理设计、实现维护。
2. 关系数据库的理论基础是关系代数。

数据库物理设计参考资料

第一个链接是我针对查询优化作的读书笔记，后三个链接是 SQLServerCentral 中几篇关于索引的文章（需要简单注册后才能看到全文）

1. 查询优化系列（查询优化（1），查询优化（2），查询优化（3），查询优化（4），查询优化（5）——总结）
2. Part 1 - The basics of indexes
3. Part 2 - The Clustered Index
4. Part 3 - The Non-clustered index

数据库设计 Step by Step (3)——基本 ER 模型构件

引言：数据库设计 Step by Step (2)在 园子里发表之后，收到了一些邮件，还有朋友直接电话我询问为什么不包含数据库物理设计方面的内容。我在这里解释一下，数据库物理设计与数据库产品是密切相 关的，本系列的专注点是较为通用的数据库设计理念与方法，这也是国内软件项目中容易被忽视的一块。今天我们将学习实体关系（ER）模型构件及其语义，这是 数据库逻辑设计的基础。内容可能有些枯燥，但却非常重要和有用。

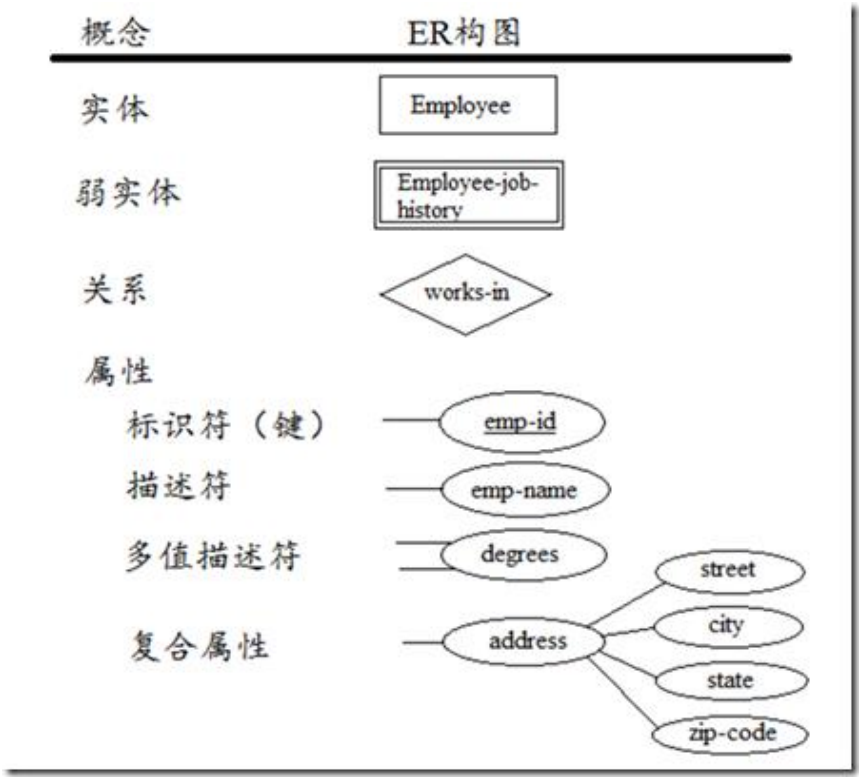
由于内容比较多，我们将分两讲来学习实体关系模型构件。

今天我们先来学习基本实体关系模型。

实体关系（ER）模型的目标是捕获现实世界的数据需求，并以简单、易理解的方式表现出来。ER 模型可用于项目组内部交流或用于与用户讨论系统数据需求。

ER 模型中的基本元素

基本的 ER 模型包含三类元素：实体、关系、属性



（图 1　实体、关系、属性的 ER 构图）

实体（Entities）：实体是首要的数据对象，常用于表示一个人、地方、某样事物或某个事件。

一个特定的实体被称为实体实例（entity instance 或 entity occurrence）。实体用长方形框表示，实体的名称标识在框内。一般名称单词的首字母大写。

关系（Relationships）：关系表示一个或多个实体之间的联系。关系依赖于实体，一般没有物理概念上的存在。关系最常用来表示实体之间， 一对一，一对多，多对多的对应。关系的构图是一个菱形，关系的名称一般为动词。关系的端点联系着角色（role）。一般情况下角色名可以省略，因为实体名 和关系名已经能清楚的反应角色的概念，但有些情况下我们需标出角色名来避免歧义。

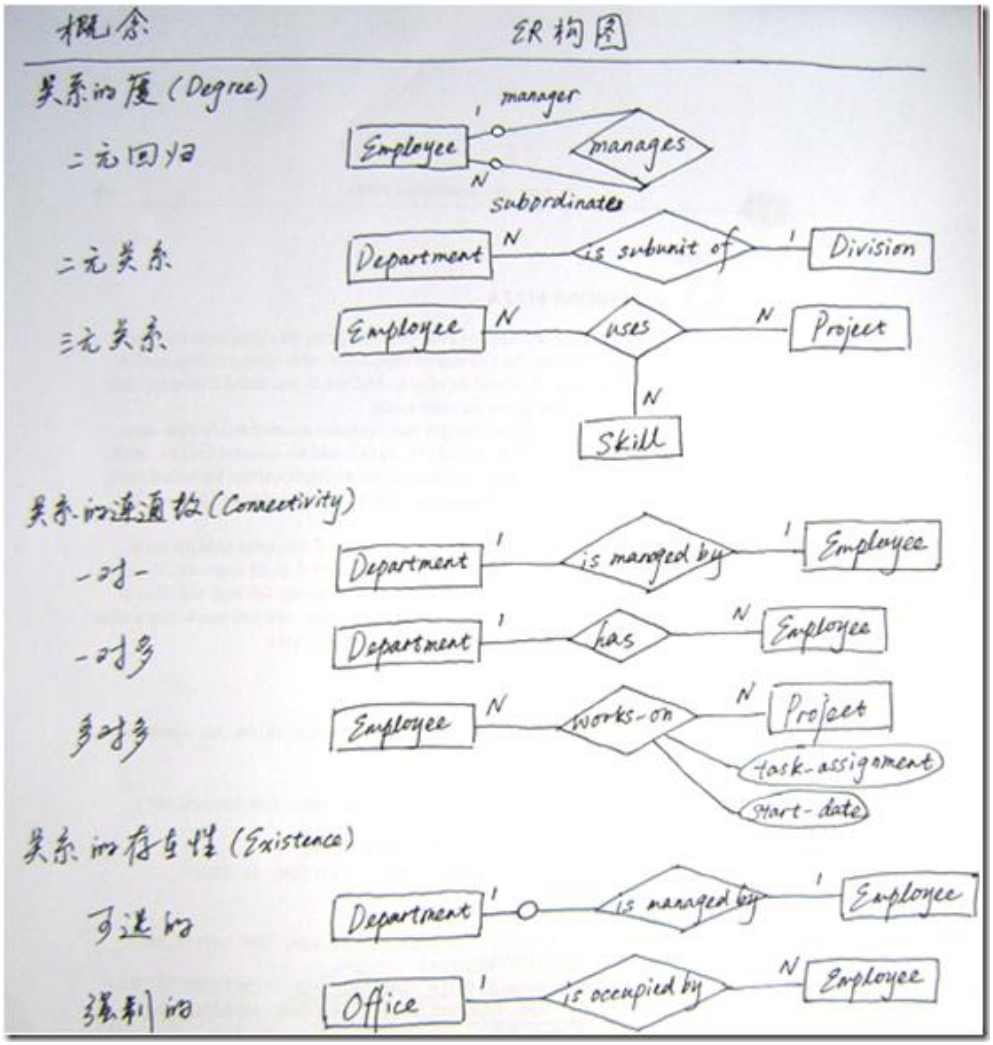
属性（Attributes）：属性为实体提供详细的描述信息。一个特定实体的某个属性被称为属性值。Employee 实体的属性可能有：emp- id, emp-name, emp-address, phone-no……。属性一般以椭圆形表示，并与描述的实体连接。属性可被分为两类：标识符（identifiers），描述符（descriptors）。Identifiers 可以唯一标识实体的一个实例（key），可以由多个属性组成。ER 图中通过在属性名下加上下划线来标 识。多值属性（multivalued attributes）用两条线与实体连接，eg：hobbies 属性（一个人可能有多个 hobby，如 reading，movies…）。复合属性（Complex attributes）本身还有其它属性。

辨别强实体与弱实体：强实体内部有唯一的标识符。弱实体（weak entities）的标识符来自于一个或多个其它强实体。弱实体用双线长方形框表示，依赖于强实体而存在。

深入理解关系

关系在 ER 模型中扮演了非常重要的角色。通过 ER 图可以描述实体间关系的度、连通数、存在性信息。

我们一一来解释这些概念。首先我们来看一下关系在 ER 图中的各种语义。



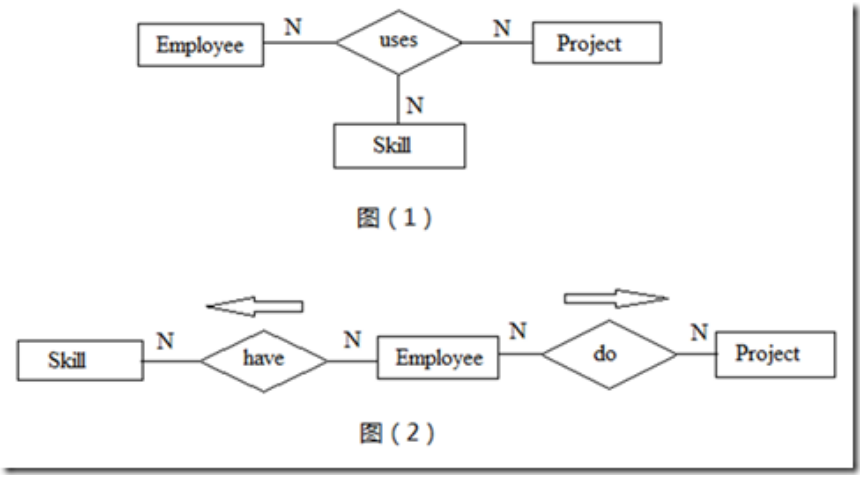
（图 2　关系的度、连通数、存在性）

关系的度（Degree of a Relationship）

表示关系所关联的实体数量。二元关系与三元关系的度分别为 2 和 3，以此可以类推至 n 元。二元关系是最常见的关系。

一个 Employee 与另一个 Employee 之间的领导关系称为二元回归关系。如图 2 中所示，Employee 实体通过关系 manages 与自身连接。由于 Employee 在这一关系中扮演两个角色，故标出了角色名（manager 和 subordinate）。

三元关系联系三个实体。当二元关系无法准确描述关联的语义时，就需要使用三元关系。我们来看下面这个例子，下图（1）能反映出一个 Employee 在某个 Project 中使用了什么 Skill。下图（2）只能看出 Employee 有什么 Skill，参与了哪些 Project，但无法知道在某个 Project 中使用的特定 Skill。



（图 3　三元关系蕴含的语义）

需要注意的是有些情况下会错误的定义三元关系。这些三元关系可分解为 2 个或 3 个二元关系，来达到化简与语义的纯净。以后的博文中会进一步详细讨论三元关系。

一个实体可以参与到任意多个关系中。每个关系可以联系任意多个元（实体），而且两个实体之间也能有任意多个二元关系。

关系的连通数（Connectivity of a Relationship）

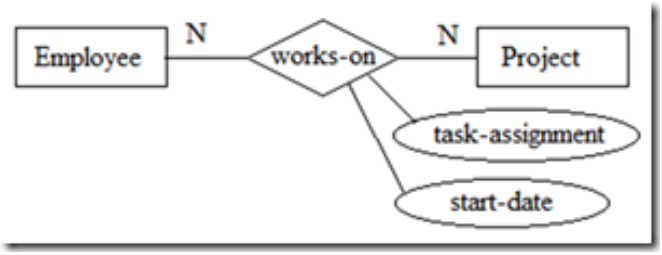
表示关系所关联的实例数量的约束。

连通数的值可以是“一”或“多”。“一”这一端，在 ER 图中通过在实体与关系间标记“1”表示。“多”一端标记“N”表示。如图 2 中关系连通数部 分，“一”对“一”：Department is managed by Employee；“一”对“多”：Department has Employees；“多”对“多”：Employee may work on many Projects and each Project may have many Employees。

有些情况下最大连通数是确定的，可以用数值代替 N。如：田径队队员有 12 人。

关系的属性

关系也能有属性。如下图 4 所示，某员工参与某项目的起始日期，某员工在某项目中被分配的任务只有放在关系 works-on 上才有意义。



（图 4　关系的属性）

需要注意的是关系的属性一般出现在“多”对“多”的二元关系或三元关系上。一般“一”对“一”或“一”对“多”关系上不会放属性（会引起歧义）。而 且这些属性可以移至一端的实体中。如下图 5 所示，如果部门与员工（经理）之间是“一”对“一”关系，在建模中可能把 start-date 作为关系 is managed by 的属性（表示被接管的时间），这个属性可以移至 Department 或 Employee 实体中。

（图 5　部门与经理之间的一对一管理关系）

大家可以思考一下如果部门和经理之间是“多”对“多”关系，即交叉管理，那又会怎样？

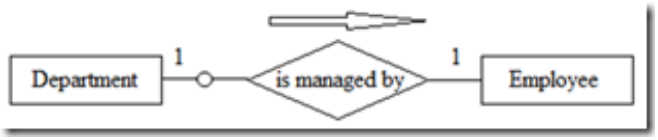
关系中实体的存在性（Existence of an Entity in a Relationship）

关系中实体的存在性可以是强制的或可选的。当关系中的某一边实体（无论是“一”或“多”端）必须总是存在，则该实体为强制的。反之，该实体为可选的。

在实体与关系之间的连接线上标识“0”来表示可选存在性。含义是最小连通数为 0。

强制存在性表示最小连通数为 1。在存在性不确定或不可知的情况下，默认最小连通数为 1。

在 ER 图中最大连通数显式地标识在实体旁边。如图 6 所示，其蕴含的语义为一个 Department 有且只有一个 Employee 来当经理，一个 Employee 可能是一个 Department 的经理，也可能不是。



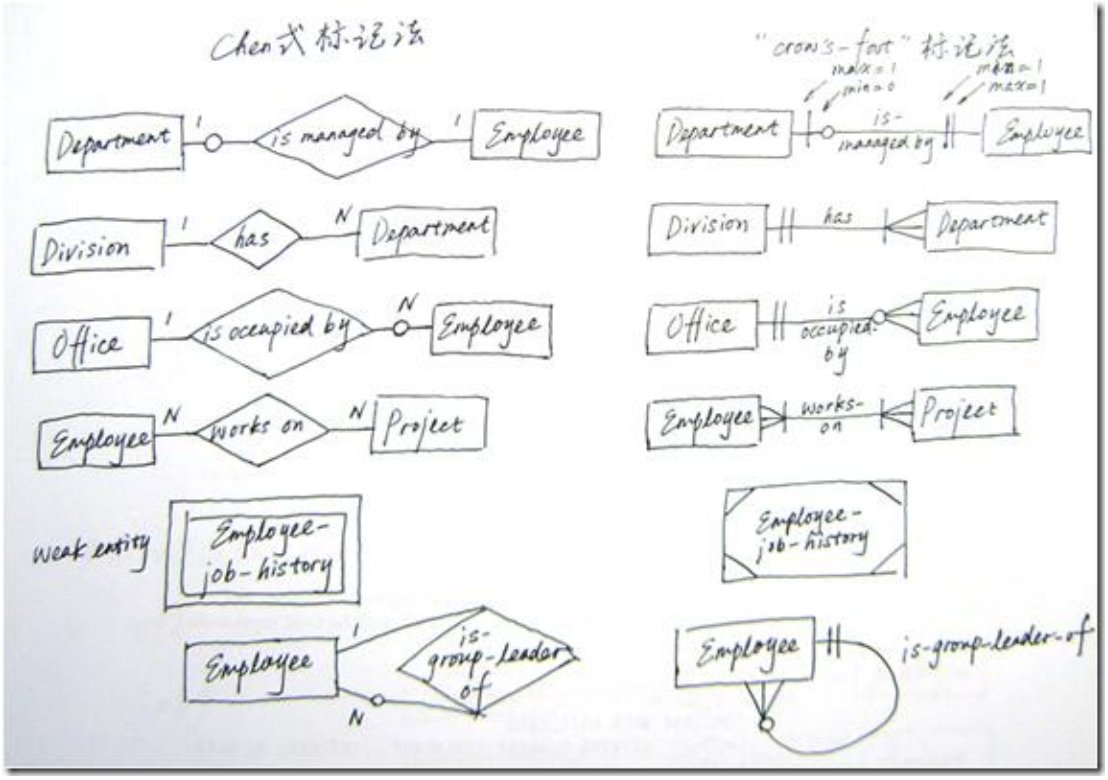
（图 6　关系中实体的存在性）

其他概念数据模型标记法

前文中使用的 ER 构图方法是 Peter Chen 1976 年提出的。在现代数据库设计领域，还有其他多种 ER 模型标记法。

我们来看一下另一种使用较多的标记法，“crow’s-foot”（鱼尾纹）标记法，并与前面介绍的标记法进行一个简单对比。

学习每一种标记法没有意义。在你的组织中推广应用一种标记法，使其成为大家共通的“语言”。



（图 7　Chen 式标记法与 crow’s-foot 标记法对照）

主要内容回顾

1. 组成 ER 模型的基本元素包括：实体、关系、属性
2. 深入理解关系中包含的语义：关系的度、关系的连通数、关系的存在性
3. 了解 ER 模型的不同标记法，掌握其中一种标记法，并在你的项目中推广使用

实体关系模型参考

1. Entity-relationship model（http://en.wikipedia.org/wiki/Entity-relationship_model）
2. Entity-relationship modeling

（http://www.inf.unibz.it/~franconi/teaching/2000/ct481/er-modelling/）

数据库设计 Step by Step (4)——高级 ER 模型构件

引言：数据库设计 Step by Step (3)中 我们讨论了基本实体关系模型构件及其语义。这些概念非常重要，是今天这一讲的基础，在开始本文内容之前建议大家可以再回顾一下上一篇的内容。今天我们将讨 论高级实体关系模型构件，与上一篇一起涵盖了 ER 模型构图的大部分内容。三元关系是今天这一讲的难点，大家可以重点关注。

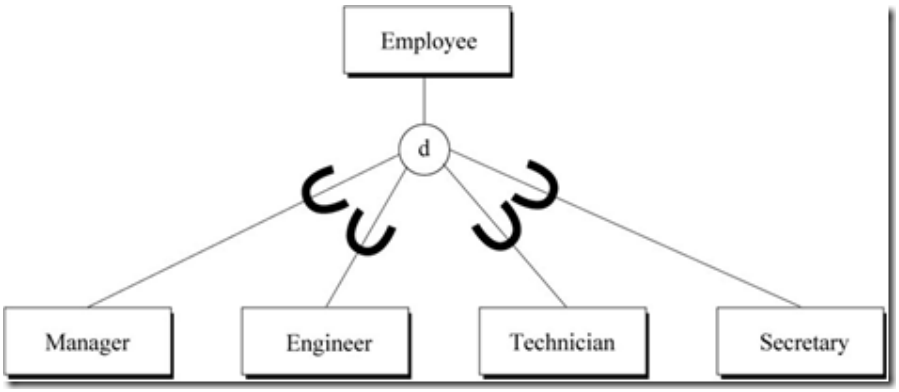
泛化（Generalization）：超类型与子类型

原始的 ER 模型已经能描述基本的数据和关系，但泛化（Generalization）概念的引入能方便多个概念数据模型的集成。

泛化关系是指抽取多个实体的共同属性作为超类实体。泛化层次关系中的低层次实体——子类型，对超类实体中的属性进行继承与添加，子类型特殊化了超类型。

ER 模型中的泛化与面向对象编程中的继承概念相似，但其标记法（构图方式）有些差异。

下图表示员工与经理、工程师、技术员、秘书之间的泛化关系。Employee 为超类实体，并包含共同属性，Manager、Engineer、Technician、Secretary 都是 Employee 的子类实体，它们能包含自身特有的属性。

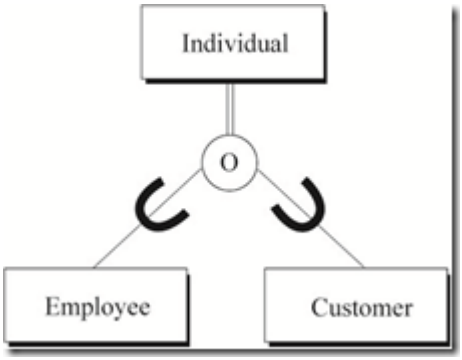


（图 1 Employee 与 Manager、Engineer、Technician、Secretary 之间的泛化关系）

泛化可以表达子类型的两种重要约束，**重叠性约束（disjointness）**与**完备性约束（completeness）**。

重叠性约束表示各个子类型之间是否是排他的。若为排他的则用字母“d”标识，否则用“o”标识（o -> overlap）。图 1 中各子类实体概念上是排他的。

对员工、客户实体进行泛化，抽象出超类实体个人，得到如下关系图。由于部分 Employee 也可能是 Customer，故子类实体 Employee 与 Customer 之间概念是重叠的。



（图 2　Individual 与 Employee、Customer 之间的泛化关系）

完备性约束表示所有子类型在当前系统中是否能完全覆盖超类型。若能完全覆盖则在超类型与圆圈之间用双线标识（可以把双线理解为等号）。在图 2 中子类实体 Employee 与 Customer 能完全覆盖超类 Individual 实体。

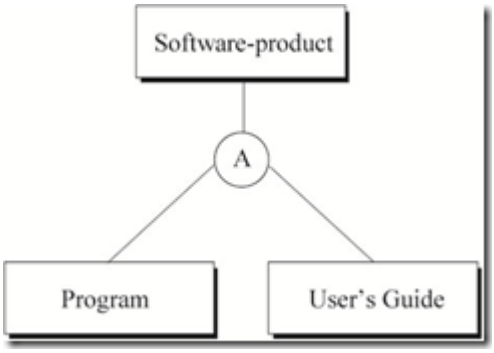
聚合（Aggregation）

聚合是与泛化抽象不同的另一种超类型与子类型间的抽象。

泛化表示“is-a”语义，聚合表示“part-of”语义。聚合中子类型与超类型间没有继承关系。

聚合关系的标记法是在圆圈中标识字母“A”来表示。

下图表示软件产品由程序与用户手册组成。



（图 3　Software-product 与 Program、User's Guide 之间的聚合关系）

三元关系（Ternary Relationships）

当通过二元关系无法准确描述三个实体间的联系时，我们需要使用三元关系。

三元关系中“连通数”的确定方法：

a) 以三元关系中的一个实体作为中心，假设另两个实体都只有一个实例

b) 若中心实体只有一个实例能与另两个实体的一个实例进行关联，则中心实体的连通数为“一”

c) 若中心实体有多于一个实例能与另两个实体实例进行关联，则中心实体的连通数为“多”

注：什么时候需要使用三元关系的实例请参看：数据库设计 Step by Step (3)中的“关系的度（Degree of a Relationship）”小节。关系的“连通数”概念请参看：数据库设计 Step by Step (3)中的“关系的连通数（Connectivity of a Relationship）”小节。

我们来看几个三元关系的实例，注意各个图中**关系的度**，并理解其中的语义。

（图 4　技术员在项目中使用手册的关系）

图 4 中蕴含的语义为：

a) 一名技术员对于每一个项目使用一本手册

b) 每一本手册对于每一个项目属于一名技术员

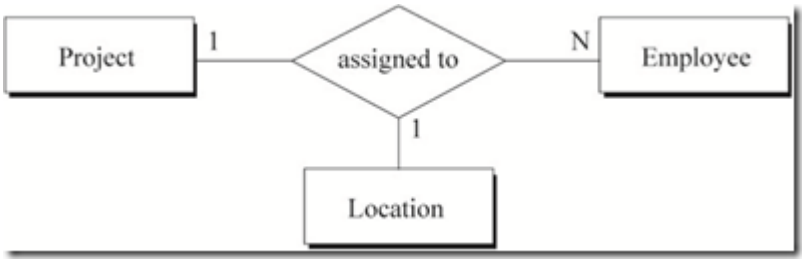
c) 一名技术员可能在做多个项目，对于不同的项目维护不同的手册

用数学中的函数依赖表示图 4 的关系：

a) emp-id, project-name -> notebook-no

b) emp-id, notebook-no -> project-name

c) project-name, notebook-no -> emp-id



（图 5　员工被分配不同地点的项目之间的关系）

图 5 中蕴含的语义为：

a) 每一个员工在一个地点只能被分配一个项目，但可以在不同地点做不同的项目

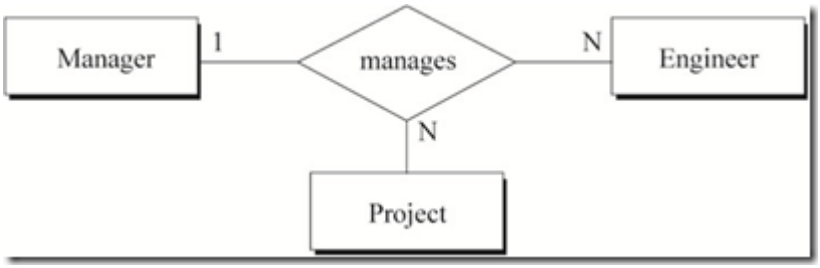
b) 在一个特定的地点，一个员工只能做一个项目

c) 在一个特定的地点，一个项目可以由多个员工来做

用数学中的函数依赖表示图 5 的关系：

a) emp-id, loc-name -> project-name

b) emp-id, project-name -> loc-name



（图 6　经理管理项目与工程师的关系）

图 6 中蕴含的语义为：

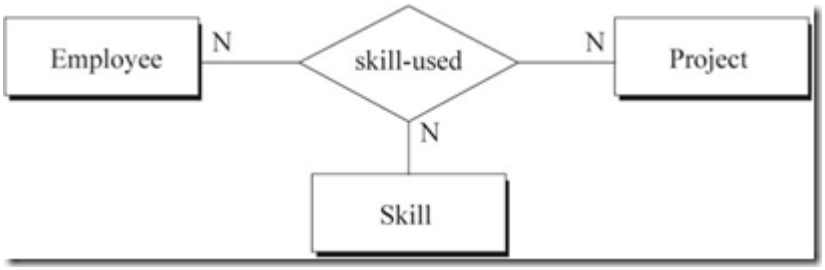
a) 一名经理手下的一名工程师可能参与多个项目

b) 一名经理管理的一个项目可能会有多名工程师

c) 做某一个项目的一名工程师只会有一名经理

用数学中的函数依赖表示图 6 的关系：

a) project-name, emp-id -> mgr-id



（图 7　员工在项目中使用技能的关系）

图 7 中蕴含的语义为：

a) 一名员工在一个项目中可以使用多种技能

b) 一名员工的一种技能可以在多个项目中使用

c) 一种技能在一个项目中可以被多名员工使用

图 7 各实体之间没有函数依赖

上述 4 种形式的三元关系，连通数为“一”的实体数量与该三元关系反映的函数依赖语义的数目一致。

三元关系也能有属性。属性值由三个实体的键的组合唯一确定。

n 元关系（General n-ary Relationships）

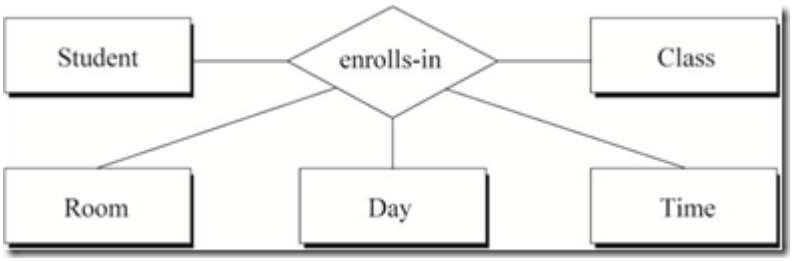
三元关系可以扩展到 n 元关系，描述 n 个实体之间的关系。

一般而言，n 元关系中每一个连通数为“一”的实体的键都会出现在一个函数依赖表达式的右侧。

对于 n 元关系，使用语言来表达其中的约束相对较为困难。建议使用数学形式即函数依赖（FD）来表现。

n 元关系的函数依赖条目数量与关系图中“一”端实体的数量相同（0～n 条）。

n 元关系的函数依赖表达式包含 n 个元素，n-1 个元素出现在表达式左侧，1 个元素出现在右侧。



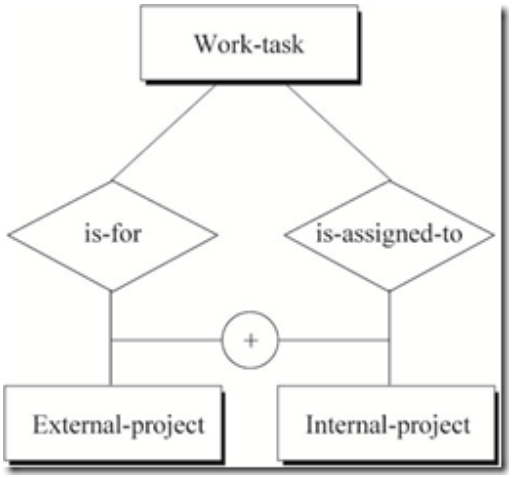
（图 8　n 元关系图例）

排他性约束（Exclusion Constraint）

一般（默认）情况下，多种关系之间是兼容的“或”关系，即允许任意或所有实体参与这些关系。

在某些情况下，多种关系之间是非兼容性“或”关系，即参与关系的实体只能选择其中一种关系，不能同时选择多种关系。

下图表示的语义为：一项工作任务要么被归为外部项目中，要么被归为内部项目中，不可能同时属于外部项目和内部项目。



（图 9　排他性约束关系图例）

我们对上一篇数据库设计 Step by Step (3)与本篇的重点内容做一个总的回顾

1. 我们讨论了 ER 模型及构图的基本概念
2. 一个实体可以是一个人，地方，东西或事件
3. 属性是实体的描述信息
4. 属性可以是唯一标识或非唯一的描述
5. 关系描述了实体之间“一对一”，“一对多”，“多对多”的联系
6. 关系的度反映了参与关系的实体数量，如二元关系，三元关系，n 元关系

7. 角色（名）定义了一个实体在一个关系中所具有的功能
8. 关系的存在概念表示一个实体在关系中是强制存在还是可选的
9. 泛化允许把实体抽象成超类与子类
10. 三元关系可使用函数依赖来定义

数据库设计 Step by Step (5)——理解用户需求

引言：数据库设计 Step by Step (4)中我们讨论了泛化关系、聚合关系、三元关系等高级实体关系模型构件及其语义。从本次讲座开始我将引领大家开始数据库设计之旅，我们将从需求分析开始，途中将经过概念数据建模、多视图集成、ER 模型转化为 SQL、范式化等过程，最终得到完整、可用的 SQL 表。

需求分析在数据库生命周期中至关重要，通常也是涉及人员最多的步骤。数据库设计师在这个阶段必须走访最终用户，与他们进行访谈，从而确定用户想在系 统中存储什么数据以及想怎样使用这些数据。我们将需求分析分为两个步骤：1.理解用户需求；2.提取业务规则。这次我们先讨论“理解用户需求”。

设计定制化产品——无论是一个数据库、一幅平面广告或一个玩具，都是一个“翻译”的过程。我们需要把浮现在客户脑海中的模糊想法、愿望挖掘出来，并“翻译”成满足他们需求的现实产品。

这个“翻译”过程的第一步就是理解用户的需求。设计最好的订单处理系统对于需要一个电路设计工具的客户来说毫无意义。对客户需求理解的不完全会造成错误或无用的设计与开发，这浪费了你、你的团队还有客户的时间与金钱。（牢记数据库是整个应用开发的根基）

制定一个计划

我们首先制定了一个计划，其中包含挖掘客户需求的一系列步骤。遵循这些步骤能更好地理解客户需求，但在一些项目中我们不需要遵循所有的步骤。举例来 说，如果客户是单个人且需求很明确时，我们就不需要进行“搞清谁是谁”与“头脑风暴”了。当客户的数据需要保密时，我们就不能“尝试客户的工作”了。在另 一些项目中，调整这些步骤的顺序会更为合适。例如我们可能在去拜访客户和观察他们工作之前先进行“头脑风暴”。

以下按照最普遍的顺序列出了各个步骤。大家根据不同项目的情况可进行灵活调整，目标只有一个就是更好地理解用户需求。

- 列出问题清单
- 拜访客户
- 搞清谁是谁
- 挖掘客户大脑
- 尝试客户的工作
- 学习现有操作
- 头脑风暴
- 展望未来

- 理解客户的质疑
- 弄清客户的真正需求
- 优先级
- 确认你的理解
- 撰写需求文档

下面我们将一一解释每一个步骤。

列出问题清单

我们需要思考，向客户问些什么问题可以帮助我们了解项目的目标和范畴（scope）。以下几个方面的问题可以作为起始点。

功能：

以下问题主要涉及系统应完成的功能与目标。

- 系统应该做些什么？
- 为什么你想建这个系统？
- 系统看上去应该是怎样的？
- 需要些什么报表？
- 用户需要自己定义新报表吗？
- 系统的操作者会是谁？

数据需求：

这些问题是为了弄清项目的数据需求。了解需要些什么数据能帮助我们定义数据库表。

- 系统界面上需要展现哪些数据？
- 这些数据应该由谁来提供？
- 这些数据是如何关联的？
- 这些工作现在是如何处理的？数据来自哪里？

数据完整性：

这些问题能帮助我们在构建数据库时定义完整性约束。

- 哪些数据是必须填写的？(eg: 一条客户记录必须有电话信息吗？)
- 数据的有效域是什么？(eg: 电话号码是否有格式规定？地址数据应有多长？)
- 系统是否需要根据邮编来检验城市的有效性？
- 系统中是否必须在定义了客户之后才能下订单？
- 系统要求多高的可用性等级？(系统需要 7×24 的可用性吗？数据的备份频率要多高？)

安全性：

这些问题能帮助我们了解客户对权限控制与审计方面的需求。

- 是否每个用户都需要一个不同的密码？
- 是否需要控制不同的用户所能访问的数据？(eg: 销售代表有权限看到客户的信

用卡账号，但订单录入专员却不能)

- 存储在数据库中的数据是否需要加密？
- 谁做了什么操作是否需要记录以便于审计？(eg: 记录销售代表提高客户级别的操作，在需要时可以追溯操作的原因)
- 系统中的客户分成几个级别？每个级别的客户有多少？
- 是否已有文档记录了用户的工作与权责？

环境：

这些问题能帮助我们了解当前项目将代替其他什么系统或流程，以及项目将与其他哪些系统进行交互。

- 当前项目是要代替或升级现有的某系统吗？

  是否有描述现有系统的文档？

  现有系统的哪些功能是需要的？哪些是不需要的？

  现有系统处理些什么数据？这些数据是如何存储的？数据之间是如何关联的？

  是否有关于现有系统数据的文档？
- 当前项目必须与其他哪些系统交互？

  项目与其他系统之间如何交互？

  新项目是否需要向现有系统提供数据？如何提供？

  新项目是否需要接收现有系统的数据？如何接收？

  是否有关于其他系统的文档？
- 客户的整个业务流程是怎样的？(了解在整个业务流程中当前项目的作用)

拜访客户

了解我们要设计和搭建的系统的最好方式是询问客户。拿着我们在上一步中准备的问题清单安排与客户进行会面。这不会像闲聊那么轻松，向客户了解需求是一个冗长且折磨人的过程。

有时我们的穷追猛问会使客户筋疲力竭感到不快。在这些时候我们必须更为耐心，可以分几次多次会议来了解需求，每次针对几个问题或流程。我们的目标是对我们要解决的问题有一个完全且彻底的理解。

即使我们的项目只是去解决整个业务中的一小部分问题，我们也要试图去了解客户的整体业务流程，这可能会给我们带来意想不到的收获。

搞清谁是谁

意识到不同的客户可能对项目有不同的愿景。我们需要分辨出谁是领导，谁是积极支持者，谁是旁观者，谁是唱反调者。

以下列出了一些常见的客户角色：

- 项目发起人——一般是管理层的某位领导，他是项目的最高推动者。他会为项目协调资源，解决项目遇到的一些障碍，但他不会参与到项目每天的事务中。
- 项目执行负责人——他对于客户的需求和整个业务最为了解。他是了解用户需求阶段最重要的人，他必须有足够的时间来帮助我们定义项目目标以及回答我们的问题。当别人对某业务环节迟疑不决时，我们需要向他请教。

- 客户代表——客户代表是回答我们问题的人，他们也可能成为系统的最终用户。他们可能是某一部分业务的专家，我们需要与多个客户代表进行访谈来了解业务全貌。
- 利益相关者——这是项目将影响到的人，其中某些人可能同时也是客户代表。这些人可能对项目也有兴趣，但未必对系统都有发言权。我们在进行系统设计时也需要考虑对这些人的影响(特别是附带损害)。
- 唱反调者——这是我们需要关注的一些人。如果唱反调者只是让其他人理性或现实地来看待项目，而并不是彻底反对这个项目的话，他将是我们非常好的资 源，他将帮助我们说服其他对项目抱有不切实幻想的客户。而如果唱反调者对整个项目抱有抵触时，我们就必须非常小心，有时需要项目执行负责人出面来协调这些人。

挖掘客户大脑

一旦搞清楚谁是谁之后，我们就要与项目执行负责人讨论客户需要什么。客户希望的解决方案是怎样的，需要包含什么数据，怎样呈现，以及不同数据之间如何关联。

与尽可能多的利益相关者进行交流，我们需要考虑每个人的意见，但心中要牢记项目执行负责人最为理解客户的需求并具有最终决定权。

根据项目的规模，这一过程短则几个小时，长则需要几周才能完成。

尝试客户的工作

观察客户每日的工作能帮助我们更好的理解业务。如果我们能做一会儿客户的工作来了解其中包括的内容那就最好了。

即使我们不能实际尝试客户的工作，一般我们还是可以坐在他们身边近距离观察。告诉客户我们将稍稍降低他们的工作效率并问一些愚蠢且恼人的问题，之后 我们就可以开问了。在这个过程中要进行记录，学习尽可能多的东西。有些时候外行者的一些看法可能转化为客户怎么也不会想到的好主意。

学习现有操作

在尝试客户的工作之后，我们还可以看一下是否有其他途径能了解现有流程。通常公司有描述客户角色和职责的操作手册或文档。

寻找客户现在使用的数据存储方式，可能是关系型数据库系统或是电子表格或是纸质的单据等等。了解这些数据是怎样使用的，之间是如何关联的。一般物理数据库之间是通过包含冗余信息来相互关联的，如：客户 ID。

头脑风暴

此刻我们已经对客户的业务和需求较为了解了。为了确认没有什么遗漏，我们需要安排头脑风暴。召集项目执行负责人和尽可能多的客户代表与利益相关者，向他们描述前期了解到的需求情况，之后让他们畅所欲言谈谈其中有什么问题或还缺什么。

在这个过程中我们不急于答应或排除任何客户的要求，我们先把客户说到的东西记录下来，并确定这些方面我们已经考虑到了。在正式开发前，我们会与项目执行负责人一起根据项目的规模与交付期限确定需求的优先级。

展望未来

在头脑风暴过程中思考一下将来的需求。问问客户他们的业务在将来是否会变化或他们希望系统将来能包含什么功能。

我们可以把他们的一些想法放入当前的项目中，即使不能也可以使我们知道将来可能会有些

什么扩展，在设计数据库时我们能预先留有余地。

理解客户的质疑

一些热心且懂些技术的用户会跑来建议我们如何设计系统，应该创建怎样结构的数据表。我们可能觉得这些建议毫无意义甚至可笑。但在忽视这些建议之前我 们应谨慎思考用户提出这些建议或质疑的深层原因是什么。客户比我们更了解业务，他们的建议或质疑中可能蕴含着我们还未了解到的业务变化点或某些特殊业务情 况。

弄清客户的真正需求

有时客户并不了解自己的真正需求。他们能看到问题的表象，但未必清楚其根源。我们需要帮助客户寻找到问题的根源并针对问题的源头提出解决方案。

有时客户认为数据库或新系统能神奇般的提高销售，减少成本。事实上一个设计精良的数据库能减少输入差错，提高操作效率，提供数据报表，帮助客户管理数据等等。我们在与客户沟通的过程中需要告诉他们新系统能做些什么，不能做些什么，让客户建立起正确的预期。

优先级

经过先前的步骤，我们已列出一张长长的期望功能列表。其中的某些功能可能不切实际或超出了当前项目的范畴。为了使项目规模可控，我们要与客户一起定义功能的优先级。

一般我们可以把功能分为三个等级。第一优先级是在本期开发中必须包含的功能，没有完成这些功能意味着项目的失败。第二优先级是可以放到下一期开发的 功能，当第一优先级的功能完成后，我们可以把第二优先级的部分功能提到当期开发。第三优先级是那些相对不重要或超出项目范畴的功能，我们可以忽略这些功 能。

有些情况下优先级是可能转化的。当第一优先级的某功能非常难实现时，我们可以与客户进行沟通，确认该功能是否如此重要，是否能移到第二优先级中以避 免影响项目进度。当第二优先级中的某些功能很容易实现，我们可以把该功能调整到第一优先级列表中。但做这些调整之前必须与客户沟通，得到客户的认可。

验证你的理解

梳理我们对业务和需求的理解，并一一与客户进行确认。当客户说“但是”、“除了”、“有时”等词时，我们要特别当心，确认客户只是强调了我们已经知道的东西，而没有出现新的情况。在这个阶段客户可能会想到他们之前没有考虑到的例外情况。

例外情况是数据库设计的大害。在需求分析阶段把例外情况挖掘出来，我们才能在数据库设计时有所准备。例如，我们向客户确认退货流程说：“到这里收货 员会输入 RMA 号并点击完成按钮是吗？”客户可能会说：“嗯…这是大多数情况，但有时没有 RMA 号，收货员会填入 None。”这就是一个客户之前没有告诉 我们的重要例外情况，我们必须立刻记录下来。再有一个例子，假设客户使用的纸质订单有配送地址与账单地址两个栏目。我们向客户确认时说：“订单需要有一个 配送地址和一个账单地址。”客户打断说：“有时我们需要两个配送地址，因为订单不同部分可能要送到不同的地方。”，并找出一张订单，第二个配送地址被标注 在订单的边沿处。这是一个重大例外，在纸上可以很容易的进行标注，但在数据库的一个表单元中增加一个地址是不可能的。只有知道这一例外，我们才能用设计的 方法解决这一需求。

撰写需求文档

需求文档描述了我们要构建的系统，该文档也被称为需求规格说明。需求文档要讲清楚我们将构建怎样的系统，该系统会完成什么工作，包含哪些功能点，并描述客户如何使用该系统

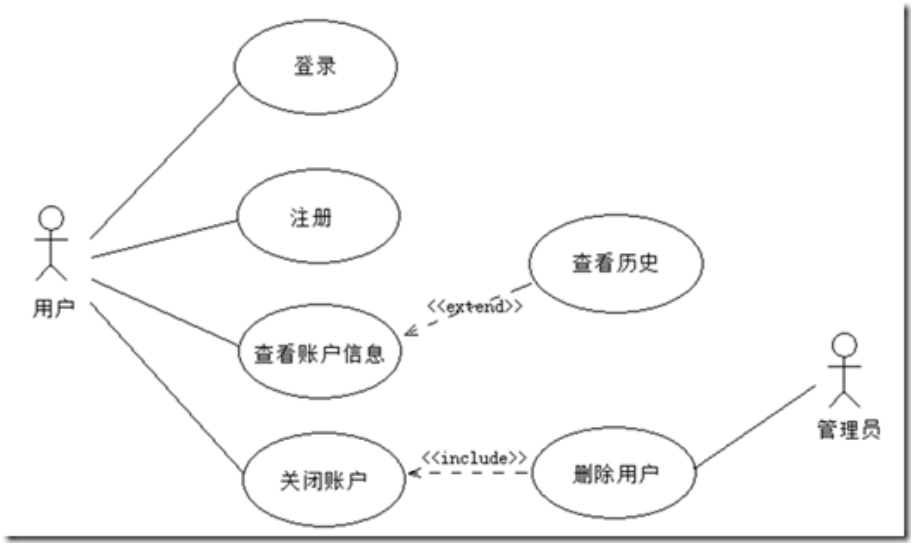
来解决他们的问题。需求文档明确了项目将完成的功能，这也避免了系统交付时出现争执的情况。

需求文档中应定义可交付成果，即里程碑。里程碑是可直观展现并能验证的中间成果。客户通过里程碑能衡量项目的进度。在需求文档中还需定义最终交付成果，这也是确定项目是否完成的标准。

用例图是一种非常好的需求分析工具，可以作为需求文档的一部分。用例图的最主要功能就是用来表达系统的功能性需求或行为。用例图从业务角度上体现谁 来使用系统、用户希望系统提供什么样的服务，以及用户需要为系统提供的服务，也便于软件开发人员最终实现这些功能。在官方文档中用例图包含六个元素，分别 是：参与者(Actor)、用例(Use Case)、关联关系(Association)、包含关系(Include)、扩展关系(Extend)以及泛化关系 (Generalization)。但是有些 UML 的绘图工具多提供了一种直接关联关系(Directed Association)。

- 参与者：是指用户在系统中扮演的角色
- 用例：是指外部可见的系统功能，对系统提供的服务进行描述
- 关联关系：连接参与者和用例，表示该参与者代表的外部系统实体与该用例描述的系统需求有关
- 包含关系：是来自于用例的抽象，即从数个不同的 Use Case 中，分离出公共的部分，而成为可以复用的用例
- 扩展关系：表示某一个用例的对话流程中，可能会根据条件临时插入另外一个用例，而前者称为基础用例后者称为扩展用例
- 泛化关系：一个用例可以被特别列举为一个或多个用例，这被称为用例泛化

eg：用户管理的用例图如下所示，图中人形图标表示参与者，椭圆表示用例（图的出处请参见“总结与参考”）



主要内容回顾

1. 搞清哪个客户扮演哪个角色
2. 从客户的脑海中挖掘信息
3. 寻找关于用户角色、职责、现有流程和现有数据的文档
4. 观察客户的工作，学习他们的业务操作
5. 进行头脑风暴，把收集到的功能需求点按优先级分成第一、第二和第三级
6. 确认对客户需求的理解
7. 撰写需求文档，包含可验证的里程碑和用例

用例图参考

1. 初学 UML 之-------用例图（http://blog.csdn.net/dl88250/archive/2007/10/16/1826713.aspx）
2. UML 用例图（http://www.alisdn.com/wordpress/?p=1161）

数据库设计 Step by Step (6) —— 提取业务规则

引言：数据库设计 Step by Step (5)中我们通过多种方法来理解客户的需求并撰写了需求文档。本文我们将回答三个问题。1. 为什么业务规则非常重要。2. 怎样识别业务规则。3. 如何修改关系模型并隔离出业务规则。

什么是业务规则

业务规则描述了业务过程中重要的且值得记录的对象、关系和活动。其中包括业务操作中的流程、规范与策略。业务规则保证了业务能满足其目标和义务。

生活中的一些业务规则可能是：

- 当顾客进入店内，最近的员工须向顾客打招呼说："欢迎来到×××"。
- 当客户兑换超过 200 元的奖券时，柜员须要求查看客户的身份证并复印。当兑换的奖券金额小于 25 元时，无需客户签字。
- 早上第一个进办公室的人需要把饮水机加热按钮打开。

本系列我们关注数据库相关的业务规则，一些例子如下：

- 只有当客户产生第一个订单时才创建该客户的记录。
- 若一名学生没有选任何一门课程，把他的状态字段设为 Inactive。
- 若销售员在一个月中卖出 10 套沙发，奖励 500 元。
- 一个联系人必须至少有 1 个电话号码和 1 个 email 邮箱。
- 若一个订单的除税总额超过 1000 元则能有 5%的折扣。
- 若一个订单的除税总额超过 500 元则免运费。
- 员工购买本公司商品能有 5%的折扣。
- 若仓库中某货品的存量低于上月卖出的总量时，则需要进货。

从数据库的视角来看，业务规则是一种约束。简单的约束如：

所有订单必须有一个联系电话。

上述这类简单的规则可以很容易的映射到关系数据库定义中，为字段确定数据类型或设定某字段为必填（不能为 NULL）。某些业务规则表达的约束会复杂些，如：

学生每天的上课时间加上项目时间必须在 1 至 14 小时之间。

我们可以通过 check 约束或外键约束来实现这类业务规则。对于一些非常复杂的业务规则，如：

一名教员每周不能少于 30 小时工作量，其中分为办公时间、实验时间和上课时间。每

1 小时的课需要 0.5 小时办公时间进行备课。每 1 小时实验需 1 小时办公准备。每周指导学生论文时间不少于 2 小时。

类似上述的业务规则需要从多个表中收集数据，故在程序代码中实现最为合适。

识别关键业务规则

记录所有的业务规则并对这些规则进行分类能帮助我们更好的在系统中实现业务逻辑。

如何实现业务规则不仅与当前的业务逻辑有关，而且与该业务逻辑将来如何变化有关。当一个规则在将来很可能变化时，我们需要使用更复杂但更灵活的方式构建该规则。

举例来说，假设公司只能向当地设有仓库的城市发货，这些城市包括：南京、长沙、西安、广州。业务规则要求订单中的发货城市字段必须为 NJ、CS、XA、GZ 之一。

我们可以把该规则简单的实现为 check 约束。但将来公司若在上海有了一个新仓库，就必须从后台数据库端修改该 check 约束。若公司随后设立更多新仓库或业务规则变化为可以向没有仓库的城市发货，每次我们都需要修改该约束。

考 虑另一种实现该业务规则的方法——使用外键。我们创建一张 ShippingCities 表，其中存放值：NJ、CS、XA、GZ，并让订单表中的发货城市 字段外键引用 ShippingCities 表中的主键。这样订单的发货城市列只能接受 ShippingCities 中存在的城市。当支持的发货城市增加或 减少时，只需要在 ShippingCities 中插入或删除记录。

两种方式的实现难度差异不大，但前一种方式每次都需要修改数据库结构，后一种只需要修改数据。修改数据不仅更省力而且技术要求也更低。

上述业务规则实现为 check 约束可能如下：

ShippingCity = ‘NJ’ or ShippingCity = ‘CS’ or ShippingCity = ‘XA’ or ShippingCity = ‘GZ’

上述代码并不复杂，但只有熟悉数据库的程序员从后台才能修改。ShippingCitis 表中的数据相对更易于理解，我们可以提供一个界面来让用户自己维护其中的城市。

要识别关键业务规则，我们可以问自己两个问题。

第一、修改规则会有多困难。越是复杂的规则，修改起来越困难且更容易出错。

第二、规则变化的可能性有多大。变化频繁的规则需要额外的设计来更好的应对将来的变化。

需要特别注意的规则（关键业务规则）：

- 枚举值。例如：有效的发货城市，订单状态（Pending, Approved, Shipped）等。
- 计算参数。例如：对 500 元以上的订单免运费。这一数值可能在将来会调整为 300 元或 600 元。
- 有效参数。例如：项目组可由 2 至 5 人组成。某些项目是否可能由 1 个人完成或有更多人参与。
- 交叉记录和交叉表检查。例如：订单中可订购的货品数量不能超过该货品的当前库存数。
- 可 概括性约束。如果可预见到将来需应用一些类似的约束，我们可以考虑把这些约束抽象出来进行管理。例如：某保险公司最近主推保险产品 A。对每月能卖出 20 份 A 产品的销售人员给予 1000 元奖金。对于不同的保险产品在不同的时间段可能有不同的推广奖励规则。我们可以把产品名称、编号、销售量、奖金数额、促销时 间段

提取出来放到一张独立的表中作为计算奖金的参数。

- 非常复杂的检查。有些检查规则非常复杂，把这些规则放到程序代码中实现更 为容易和清晰。例如：学生选择理学院的谓词演算课程的前提是已通过理学院的命题演算课程或已通过社科院的逻辑 I 和 II 课程或者需要导师的允许。该规则在某 些数据库产品中可以通过表级的 check 约束实现，但放到程序中更易于维护和理解。

一些直接可以在数据库中实现的业务规则：

- 固定枚举值。例如：性别（男、女），用手习惯（左撇子、右撇子）。
- 数据类型要求。每个字段具有确定的数据类型是关系型数据库的重要特性之一。滥用通用的数据类型（如 string）对性能和数据防错都会带来损害。
- 必填值。例如：会员必须有手机联系方式。
- 合理性检查。合理性检查设定的范围基本不会变化。例如：商品的价格大于等于 0。

作 为软件从业人员不要拒绝或回避变化。世界上唯一不变的就是变化。在收集业务规则时多去了解该规则的业务背景与历史变化历程，而不是逼迫客户保证规则不会变 化。尽可能发现所有的业务规则并记录下来。对这些业务规则按变化的可能性和修改难度进行分类，精心设计那些将来可能变化且修改困难的规则。

提取关键业务规则

识别并分类业务规则之后，我们需要在数据库中或数据库外来实现关键业务规则。我们可以参考如下方法：

1. 若规则为检验一组有效值时，把该规则转化为外键约束。先前举例中的有效发货城市就是一个很好的例子。创建 ShippingCities 表，填入允许的发货城市。然后把 Orders 表的 ShippingCity 列设为外键，引用 ShippingCities 表的主键。

2. 若规则为参数可能变化的计算式时，把这些参数提取到一张表中。例如：一个月内卖出总价超过 100 万元汽车的销售员能获得 500 元奖金。把参数 100 万元和 500 元提取到一张表中，如果需要甚至可以把一个月的时间段也作为参数提取出来。

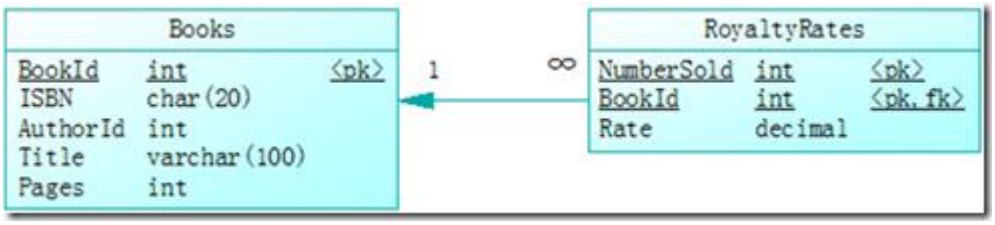
我还见过一些软件系统在数据库中有一张通用的参数表。该通用参数表中存放系统需要的各种参数，一些是用于计算、一些是作为检验、另一些决定系统的行为。每一 条记录有两个字段：Name 和 Value。例如需要确定一名销售员能获得多少奖金，我们先要查找 Name 字段为 BonusSales 的记录，检查该销售员 的销售额是否达到了 Value 字段的金额，若答案是肯定的再查找 Name 字段为 BonusAward 的记录来确定奖金数额。这种设计另有一好处，在程序启 动时可以把通用参数表读入内存的某集合中，此后使用参数值时就无需再次连接数据库。

3. 若逻辑或计算规则很复杂时，则提取到代码中进行实现。这里说的代码可以是应用程序端代码，还可以是数据库端存储过程。把规则放到代码中实现的意义在于业务 规则与数据库表结构分离了，规则的变化不会影响到数据库表结构。通过结构化编程或面向对象编程来实现复杂的规则更易于维护。

举一个综合性的例子：

一本关于数据库设计的书籍卖出前 5000 本的版税为 5%，5000 本至 10000 本之间的版税为 7%，超过 10000 本后的版税为 10%，不同类型书籍的版税可能不同。

上 述规则比较复杂且包含多个可能变化的参数，故使用第 1、2 条方法。我们可以通过存储过程来实现该规则，并把参数隔离到一张参数表中进行维护。创建的参数表 为 RoyaltyRates，并通过 BookId 与 Books 关联（如图 1 所示）。这样为不同书籍创建新的版税规则就非常容易了。



（图 1 参数表 RoyaltyRates 与 Books 表的关系）

多 层应用的概念大家都不会陌生。三层应用是最常见的分层方法。对于复杂的业务逻辑一般会在中间层（即业务层）中实现。对于一些基本的验证，如必填信息、数字 有效区间等，需要在最上层用户界面以及最底层数据库端进行双重检验。数据库端的约束是阻隔脏数据进入系统的最后一道防线，而用户界面处的检验可以避免错误 数据传输到系统后端才被拒绝，节省了系统资源。

注：关于多层应用的更多资料请参见最后的“总结与参考”部分。

主要内容回顾

1. 业务规则决定了业务如何运行，其涵盖从简单明了的入门打卡到复杂的奖金计算公式。

2. 对于数据库而言，业务规则将影响到数据模型。业务规则确定了每个字段的域（值的类型和范围），是否是必须的，以及该字段要满足的其他条件。

3. 理解业务规则并识别那些需要特别处理的关键规则至关重要。

4. 有些规则简单且基本不变，它们可以很容易的用数据库特性来实现。其他的一些规则可能复杂或时常变化，我们可以把它们从数据库中逻辑的或物理的隔离出来（隔离到参数表、存储过程或业务层中），使它们易于修改。

多层应用参考

1. 谈谈对于企业级系统架构的理解

（http://www.cnblogs.com/liping13599168/archive/2011/05/11/2043127.html）

2. Multitier architecture（http://en.wikipedia.org/wiki/Multitier_architecture）

3. Software Architecture, Architects and Architecting（http://www.bredemeyer.com/）

数据库设计 Step by Step (7)——概念数据建模

引言：在前两篇博文（数据库设计 Step by Step (5)和数据库设计 Step by Step (6) —— 提取业务规则）中，我们进行了数据库需求分析，着重讨论了两个主题：1.理解用户需求；2.提取业务规则。当需求分析完成后，我们就要进入到概念数据建模环节。本篇文章将使用之前介绍过的“基本实体关系模型构件”和“高级实体关系模型构件”作为建模的基本元素，大家可以回顾数据库设计 Step by Step (3)和数据库设计 Step by Step (4)中的模型构件及语义。

逻辑数据库设计有多种实现方式，包括：自顶至底，自底至顶以及混合方式。传统数据库设计是一个自底至顶的过程，从分析需求中的单个数据元素开始，把相关多个数据元素组合在一起转化为数据库中的表。这种方式较难应对复杂的大型数据库设计，这就需要结合自顶至底的设计方式。

使 用 ER 模型进行概念数据建模方便了项目团队内部及与最终用户之间的交流与沟通。ER 建模的高效性还体现在它是一种自顶至底的设计方法。一个数据库中的实体 数量比数据元素少很多，因为大部分数据元素表示的是属性。辨别实体并关注实体之间的关系能大大减少需要分析的对象数量。

概念数据建模连接了两端，一端是需求分析，其能辅助捕获需求中的实体及之间的关系，便于人们的交流。另一端是关系型数据库，模型可以很容易的转化为范式化或接近范式化的 SQL 表。

概念数据建模步骤

让我们进一步仔细观察应在需求分析和概念设计阶段定义的基本数据元素和关系。一般需求分析与概念设计是同步完成的。

使用 ER 模型进行概念设计的步骤包括：

- 辨识实体与属性
- 识别泛化层次结构
- 定义关系

下面我们对这三个步骤一一进行讨论。

辨识实体与属性

实体和属性的概念及 ER 构图都很简单，但要在需求中区分实体和属性不是一件易事。例如：需求描述中有句话，“项目地址位于某个城市”。这句话中的城市是一个实体还是一个属性呢？又如：每一名员工有一份简历。这里的简历是一个实体还是一个属性呢？

辨别实体与属性可参考如下准则：

- 实体应包含描述性信息

- 多值属性应作为实体来处理
- 属性应附着在其直接描述的实体上

这些准则能引导开发人员得到符合范式的关系数据库设计。

如何理解上述的三条准则呢？

实体内容：实体应包含描述信息。如果一个数据元素有描述型信息，该数据元素应被识别为实体。如果一个数据元素只有一个标识名，则其应被识别为属性。以前面的“城市”为例，如果对于“城市”有一些如所属国家、人口等描述信息，则“城市”应被识别为一个实体。如果需求中的“城市”只表示一个城市名，则把“城市”作为属性附属与其他实体，如附属 Project 实体。这一准则的例外是当值的标识是可枚举的有限集时，应作为实体来处理。例如把系统中有效的国家集合定义 为实体。在现实世界中作为实体看待的数据元素有：Employee，Task，Project，Department，Customer 等。

多值属性：把多值属性作为实体。如果一个实例的某个描述符包含多个对应值，则即使该描述符没有自己的描述信息也应作为实体进行建模。例如：一个人会有许多爱好，如：看电影、打游戏、大篮球等。爱好对于一个人来说就是多值属性，则爱好应作为实体来看待。

属性依附：把属性附加在其最直接描述的实体上。例如：“office-building-name”作为“Department”属性比作为“Employee”的属性合适。识别实体与属性，并把属性附加到实体中是一个循环迭代的过程。

识别泛化层次

如 果实体之间有泛化层次关系，则把标识符和公共的描述符（属性）放在超类实体中，把相同的标识符和特有的描述符放在子类实体中。举例来说，在 ER 模型中有 5 个实体，分别是 Employee、Manager、Engineer、Technician、Secretary。其中 Employee 可以作为 Manager、Engineer、Technician、Secretary 的超类实体。我们可以把标识符 empno，公共描述符 empname、address、date-of-birth 放在超类实体中。子类实体 Manager 中放 empno，特有描述符 jobtitle。Engineer 实体 中放 empno，特有描述符 jobtitle，highest-degree 等。

定义关系

在识别实体和属性之后我们可以处理代表实体之间联系的数据元素即关系。关系在需求描述中一般是一些动词如：works-in、works-for、purchases、drives，这些动词联系了不同的实体。

对于任何关系，需要明确以下几个方面。

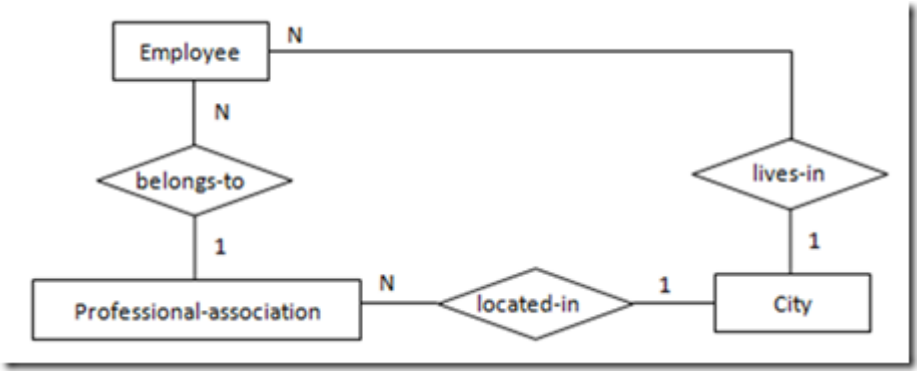
- 关系的度（二元、三元等）；
- 关系的连通数（一对一、一对多等）；
- 关系是强制的还是可选的；
- 关系本身有些什么属性。

注：关系的这些概念可参看数据库设计 Step by Step (3)，这里不再赘述。

冗余关系

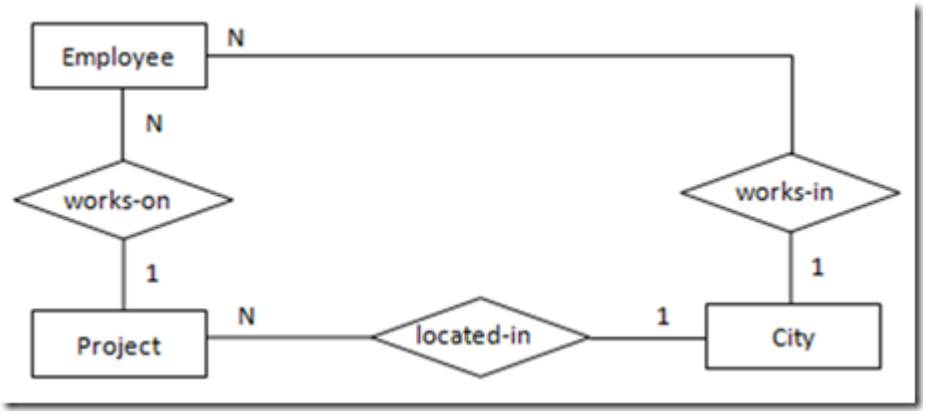
仔细分析冗余的关系。描述同一概念的两个或多个关系被认为是冗余的。当把 ER 模型转化为关系数据库中的表时，冗余的关系可能造成非范式化的表。需要注意的是两个实体间允许两个或更多关系的存在，只要这些关系具有不同的含义。在这种情况下这些关系不是冗余的。

举例来说，如下图 1 中 Employee 生活的 City 与该 Employee 所属的 Professional-association 的所在 City 可以不同（两种含义），故关系 lives-in 非冗余。



（图 1　非冗余关系）

如下图 2 中的 Employee 工作的 City 与该 Employee 参与的 Project 的所在 City 在任何情况下都一致（同种含义），故关系 works-in 冗余。



（图 2　传递性冗余关系）

三元关系

非常小心的定义三元关系，只有当使用多个二元关系也无法充分描述多个实体间的语义时，我们才会定义三元关系。以 Technician、Project、Notebook 为例。

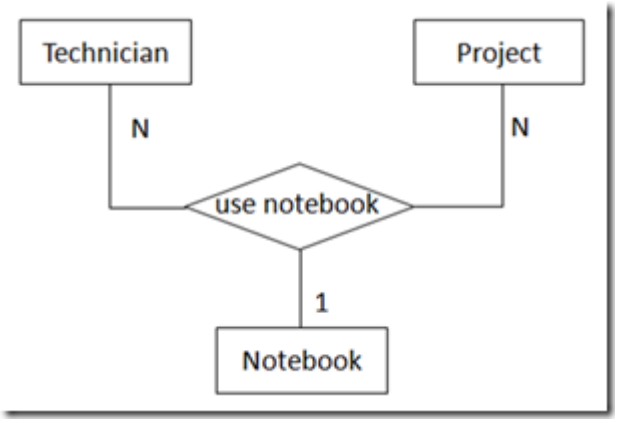
例 1：如果 一个 Technician 只做一个 Project，一个 Project 只有一个 Technician，每个 Project 会被独立记录在一本 Notebook 中。

（图 3 例 1 二元关系图）

例 2：如果一个 Technician 能同时做多个 Project，一个 Project 可以有多个 Technician 同时参与，每个 Project 有一本 Notebook（多个做同一个 Project 的 Technician 共用一本 Notebook）。

（图 4 例 2 二元关系图）

例 3：如果一个 Technician 能同时做多个 Project，一个 Project 可以有多个 Technician 同时参与，一个 Technician 在一个 Project 中使用独立的一本 Notebook。



（图 5 例 3 三元关系图）

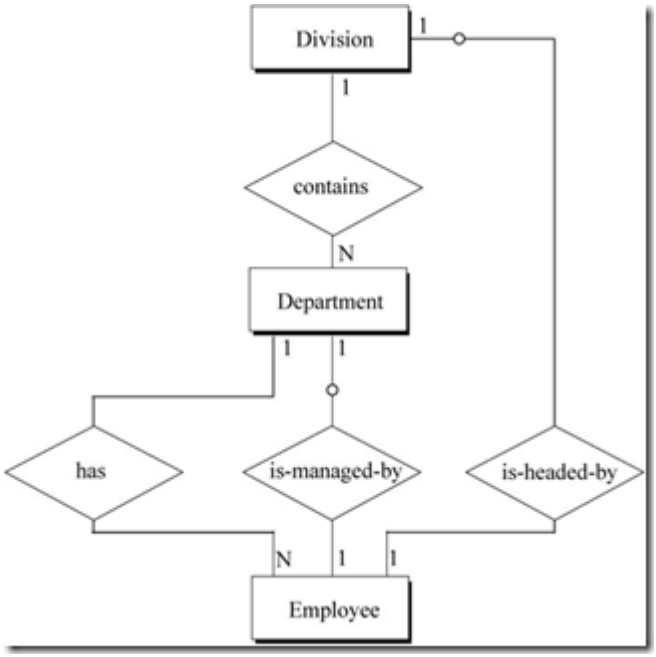
注：三元关系的语义分析可参看数据库设计 Step by Step (4)，这里不再赘述。

我们假设要为一家工程项目公司设计一个数据库来跟踪所有的全职员工，包括员工被分配的项目，所拥有的技能，所在的部门和事业部，所属于的专业协会，被分配的电脑。

单个视图的 ER 建模

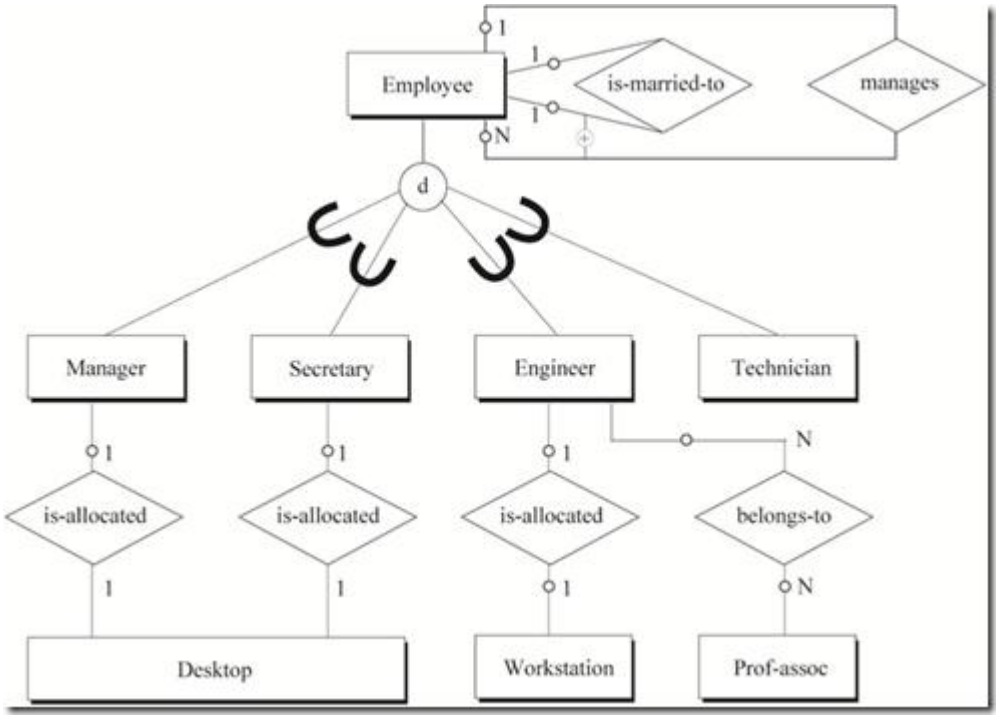
通过需求收集与分析过程，我们获得了数据库的 3 个视图。

第一个视图是人力资源管理视图。每一个员工属于一个部门。事业部是公司的基本单元，每个事业部包含多个部门。每一个部门和事业部都有一个经理，我们需要跟踪每一个经理。这一视图的 ER 模型如图 6 所示。



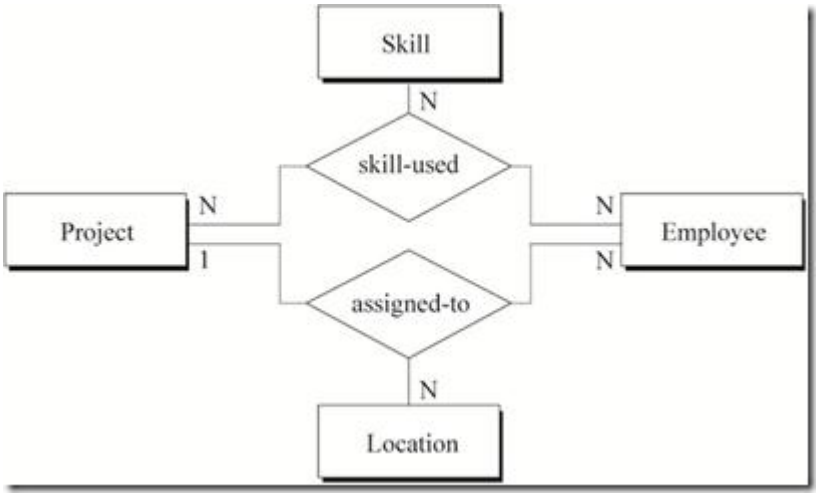
（图 6　人力资源关系视图）

第 二个视图定义了每个员工的头衔，如工程师、技术员、秘书、经理等。工程师一般属于某个专业协会，并可能被分配一台工作站。秘书和经理会被分配台式电脑。公 司会储备一些台式电脑和工作站，以分配给新员工或当员工的电脑送修时进行出借。员工之间可能有夫妻关系，这也需要在系统中进行跟踪，以防止夫妻员工之间有 直接领导关系。这一视图的 ER 模型如图 7 所示。



（图 7　员工头衔及电脑分配视图）

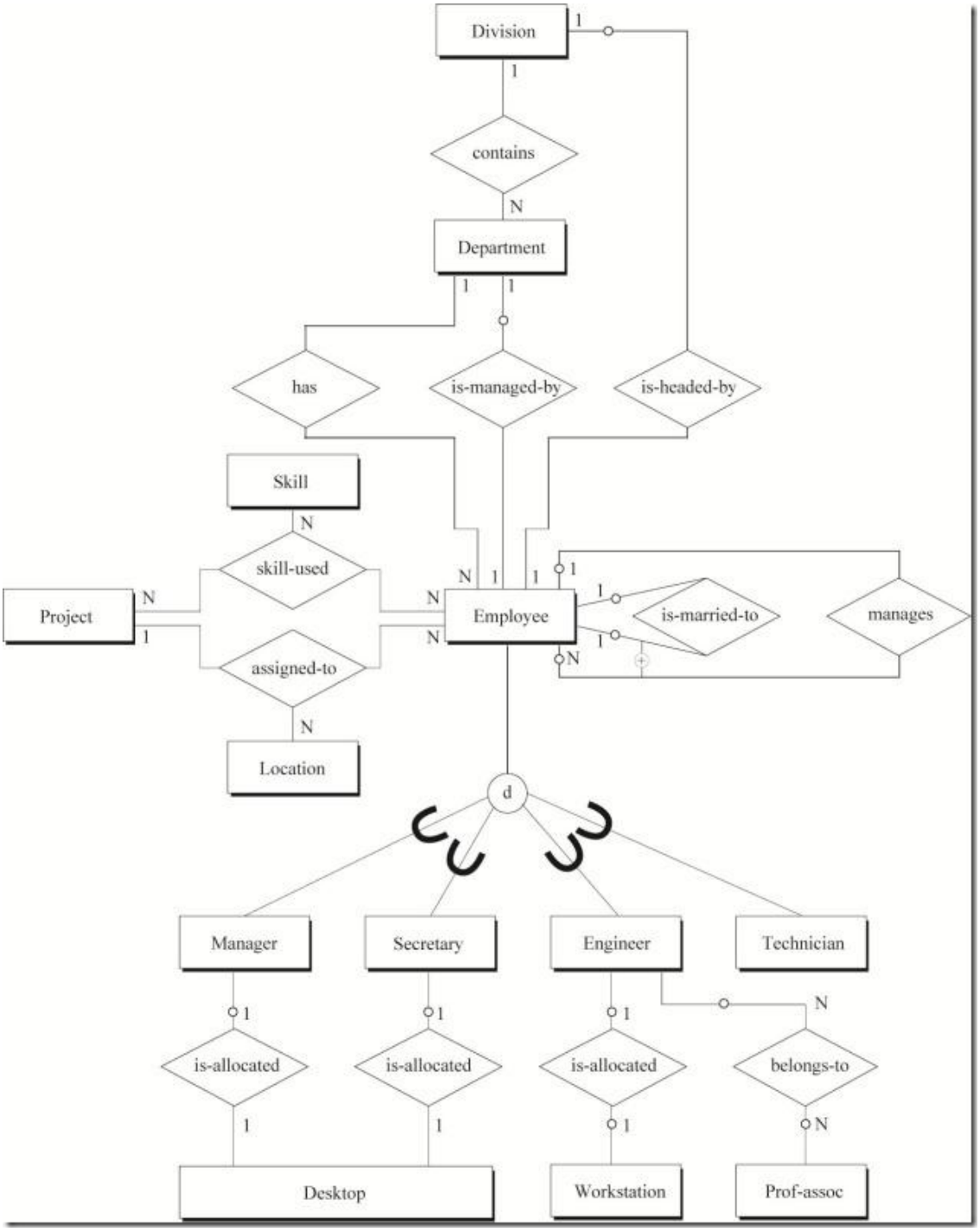
第三个视图如图 8 所示，包含员工（工程师、技术员）分配项目的信息。员工可以同时参与多个项目，每一个项目可以在不同的地方（城市）设有总部。但一个员工在指定的地点只能做当地的一个项目。员工在不同的项目中可以选用不同的技能。



（图 8　项目分配及技能使用视图）

全局 ER 图

对三个视图的简单集成可得到全局 ER 图，如图 9 所示，它是构造范式化表的基础。全局 ER 图中的每一个关系都是基于企业中实际数据的一个可验证断言。对这些断言进行分析导出了从 ER 图到关系数据库表的转化。



（图 9　全局 ER 图）

从全局 ER 图中可以看到二元、三元和二元回归关系；可选和强制存在性关系；泛化的分解约束。图 9 中三元关系“skill-used”和“assigned-to”是必须的，因为使用二元关系无法描述相同的语义。

可 选存在性的使用，Employee 与 Division 或与 Department 之间是基于常识：大多数 Employee 不会是 Division 或 Department 的经理。另一个可选存在性的例子是 desktop 或 workstation 的分配，每一台 desktop 或 workstation 未 必都会分配给一个人。总而言之，在把 ER 模型转化为 SQL 表之前，所有的关系、可选约束、泛化层次都需要与系统的最终用户进行确认。

总结来说，在关系数据库设计中应用 ER 模型会带来如下好处

1. 使用ER模型可帮助项目成员专注在讨论实体之间的重要关系上，而不受其他细节的干扰。

2. ER 模型把大量复杂的语言描述转化为精简的、易理解的图形化描述。

3. 对原始 ER 模型的扩展，如可选和强制存在性关系，泛化关系等加强了 ER 模型对现实语义的描述能力。

4. 从 ER 模型转化为 SQL 表有完整的规则，且易于使用。

实体关系（ER）模型参考资料

1. 基本实体关系模型构件——实体、关系、属性、关系的度、关系的连通数、关系的属性、关系中实体的存在性（http://www.cnblogs.com/DBFocus/archive/2011/04/24/2026142.html）

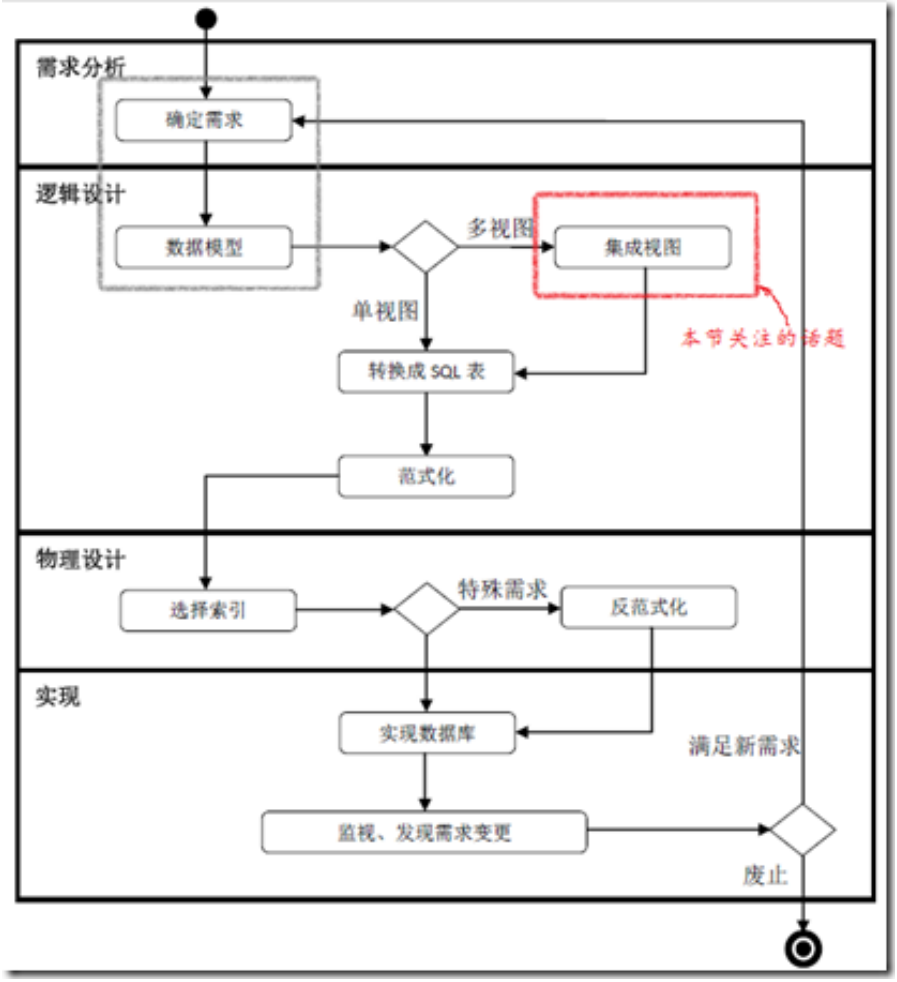
2. 高级实体关系模型构件——泛化、聚合、三元关系

（http://www.cnblogs.com/DBFocus/archive/2011/05/07/2039674.html）

数据库设计 Step by Step (8)——视图集成

引言：在前文（数据库设计 Step by Step (7)——概念数据建模）最后的案例中，我们通过集成多个局部的实体关系（ER）模型最终得到了全局 ER 图。在现实项目中视图集成可能并不会那么容易。

俯瞰整个数据库生命周期（如下图所示）。在前面的内容中，我们已完成了“确定需求”和“数据模型”（图中以灰色标出），本小节我们将详细讨论“视图集成”（图中以红色标出）

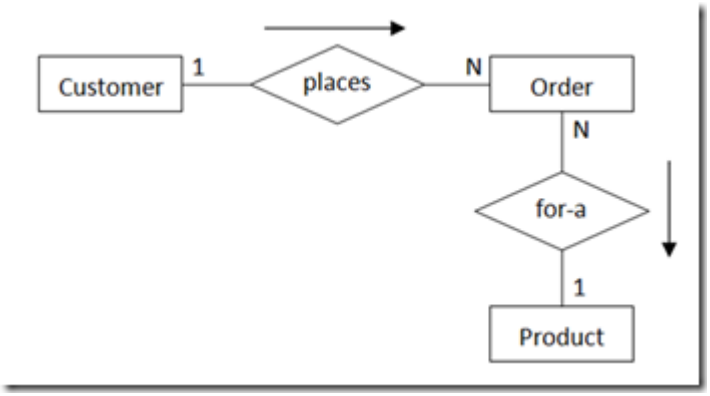


把基于不同用户视角的局部 ER 图集成为一个统一的、没有冗余的全局 ER 图在数据库设计流程中非常重要。单个局部 ER 图是通过分析用户需求进行概念数据建模得到的；全局 ER 图是通过对各个局部 ER 图进行分析，解决其中存在的视角和术语差异，最终进行组合得到的。

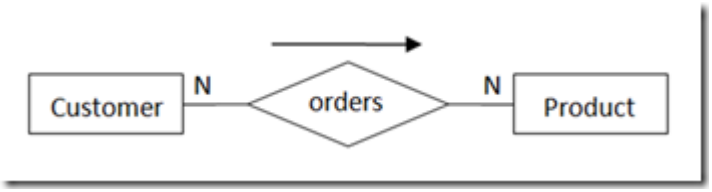
为什么会产生不一致的局部 ER 图

当 不同的用户或用户组从各自的视角来看业务时就会产生各异的 ER 图。举例来说市场部趋向于把整个产品作为销售的基本单元，但工程部可能更关注组成产品的单个 零件。另一个例子，一个用户可能关注项目的目标和产生的价值，而另一个用户则关心项目需要占用的资源和所涉及的人员。上述的这些差异造成了各个 ER 图之间 不一致的关系和术语。ER 图的不一致性会表现为：不同的泛化程度；不同的关系连通数（一对多、多对多等）；不同用户视角定义的实体、属性或关系（相同的概 念，不同的人使用了不同的名称与建模形式）。

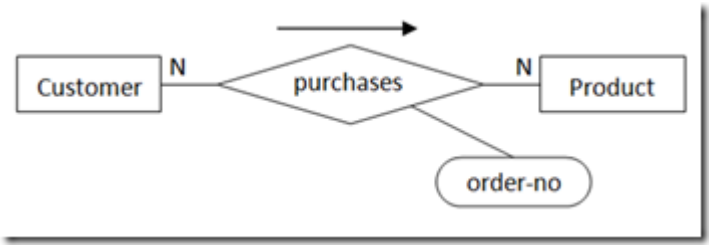
举例来说，同一个现实场景（客户下订单，订购产品），从三个不同视角建模得到的 ER 图如下。



（图 1　把 order 看作实体）



（图 2　把 order 看作关系）



（图 3　把 order 看作属性）

图 1 中，Customer、Order、Product 描述为实体，把“places”和“for-a”描述为关系。

图 2 中，“orders”定义为 Customer 和 Product 之间的关系。

图 3 中，“orders”关系被另一个关系“purchases”代替。“order-no”被作为关系“purchases”的一个属性。

同是订单（order），从不同视角出发在 ER 图中被表示为实体、关系、属性。

视图集成的步骤

局部 ER 图（概念数据模型）的集成一般有如下四个步骤。

- 集成策略选择
- 比较实体关系图

- 统一实体关系元素
- 合并、重构实体关系图

我们一一对这四个步骤进行讨论。

集成策略选择

通常的集成策略有：

1.每次集成 2 个局部 ER 图。

2.每次集成 n 个局部 ER 图（n 大于 2 且小于等于总 ER 图数）。

相对来说第一种集成策略每次所涉及的实体、关系数量较少，也更容易掌控。

比较实体关系图

设计者需要仔细观察不同 ER 图中的对应实体，发现其中因视角不同而存在的冲突。

命名上的冲突包括“同物异名”和“异物同名”。“同物异名”是指同一个概念使用了不同的名称，可以通过检视数据字典（命名及其描述对应表）来发现。“异物同名”是指对不同的概念使用了相同的名称，需要通过检视不同 ER 图中相同的名称来发现。

结 构性冲突的表现形式更多。类型冲突包括使用不同的构造方式建模同一概念。以图 1、2、3 为例，order 这一概念可以建模为一个实体，一个关系或一个属 性。依赖冲突是指类似或相同的关系在不同的局部 ER 图中被建模成不同的连通数。解决这种冲突的一种方法是使用最一般的连通数约束，如多对多。若这样做会造 成语义上的错误，则说明两种关系概念不同不能合并，应进行改名并让每个关系保持各自的连通数。键冲突是指在不同的局部 ER 图中，同一概念的实体被分配了不 同的键。举例来说，当一名员工的全名、员工号、员工身份证号在不同的局部 ER 图中被作为员工的键时，就出现了键冲突。

统一实体关系元素

基本目标是解决各局部 ER 图中的冲突，使这些元素一致化，为最终的 ER 图集成做准备。要解决各局部 ER 图之间的冲突通常需要设计开发人员与用户进行积极的沟通，了解、分析、理解冲突元素的真实语义。

我们可能需要对某些 ER 图中的实体及键属性进行改名。各局部 ER 图中被建模为实体、关系或属性的同一概念需要统一转化为三种形式之一。

集 成具有相同的度、角色和连通数属性的关系相对较为容易，但集成上述特征不同的关系就较为困难。若同一关系在不同局部 ER 图中表现形式不一致，则必须进行统 一。如：某一关系在一局部 ER 图中为泛化层次关系，在另一局部 ER 图中为排他性或（exclusive OR）关系，这种情况必须统一。

合并、重构实体关系图

合成和重构局部 ER 图，最终得到完整、最简约和可理解的全局 ER 图。

完整是要求在全局 ER 图中所有组件的语义完整。

最简约是要求去除全局 ER 图中的冗余。冗余的概念包括：重叠的实体、多余的语义关系等。例如“社会车辆”和“私家车”可能是重叠的两个实体；教授与学生之间的“指导”和“建议”关系可能代表了同一种活动，需要进一步确定是否存在冗余。

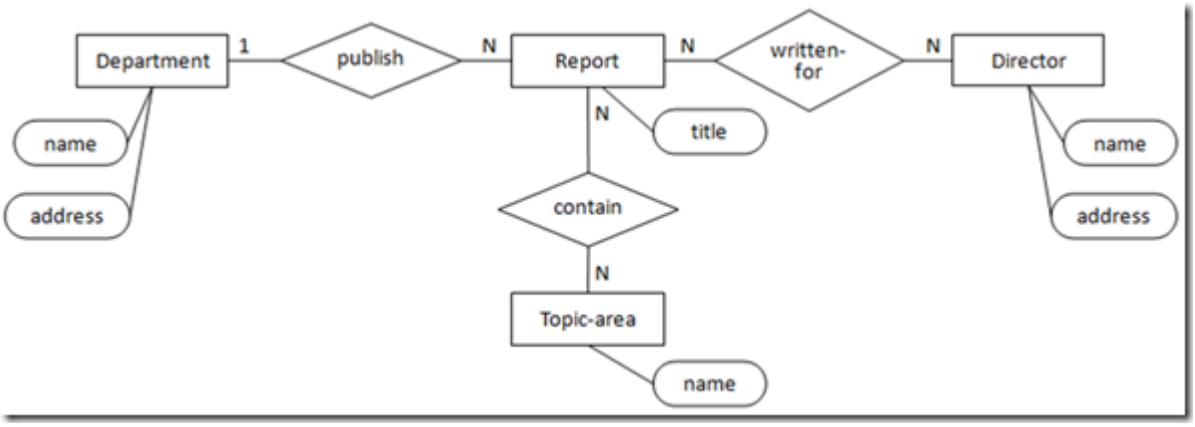
可理解要求全局 ER 图能被整个项目组成员和最终用户理解。

在进行 ER 图集成过程中，我们可以首先将相同概念的组件进行集成，之后对获得的初步全局 ER 图进行重构以使其满足上述三方面的要求。举例来说，集成后的 ER 图包含超类实体与子类实体的层次组合，若超类实体中的属性已涵盖子类实体中的某些属性，则子类实体的这些属性可以去除。

了解目标

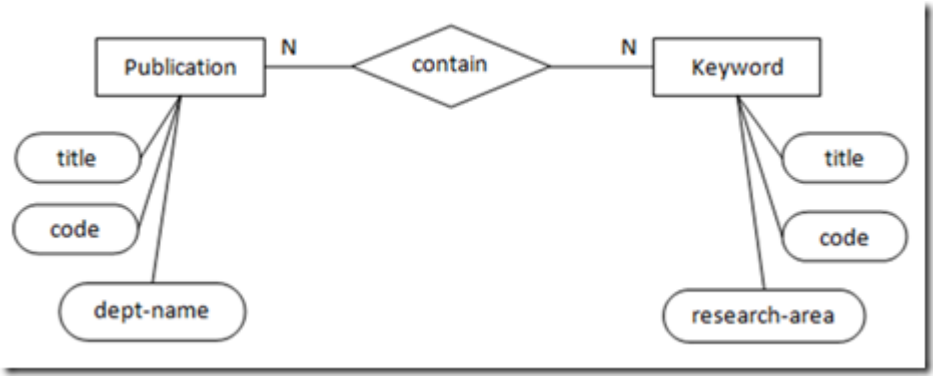
让我们看一下两张具有重叠数据的局部 ER 图。这两张 ER 图是对两组不同用户访谈后画出的。

图 4 是一张以报表为关注点的 ER 图，其中包含发布报表的部门、报表中的主题和报表提交的对象。



（图 4　关注报表）

图 5 的 ER 图以发布作为关注中心，把发布内容中的关键词建模为另一个实体。

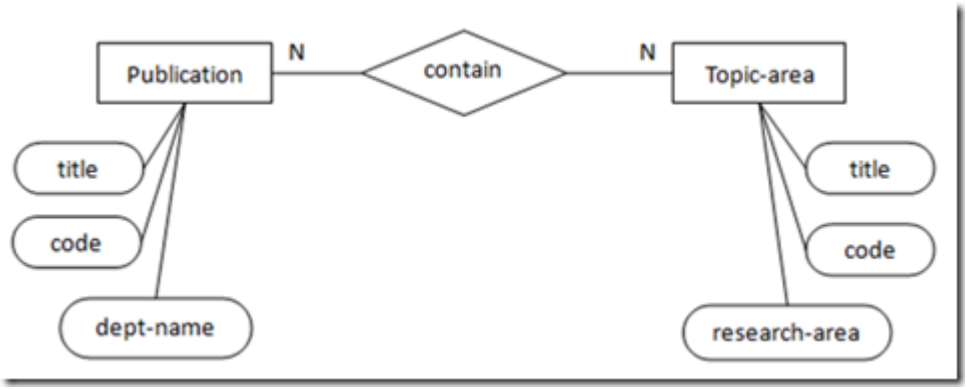


（图 5　关注发布）

我们的目标是整合这两张 ER 图，并保证合成后的 ER 图语义完整、形式最简约且易理解。

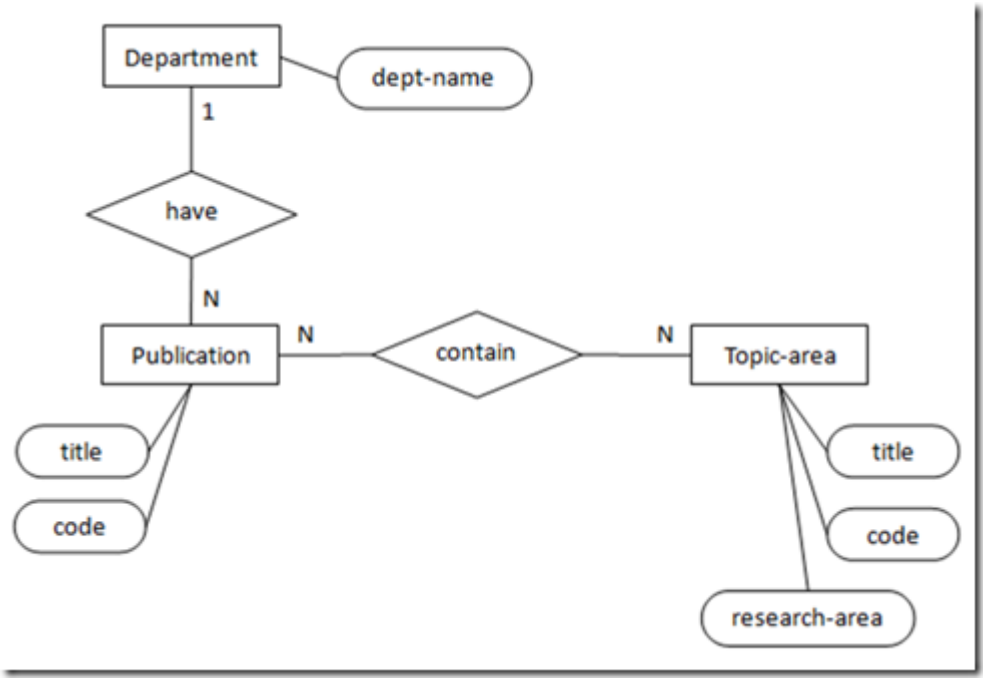
集成步骤

首先，在两张局部 ER 图中寻找是否存在“同物异名”与“异物同名”现象。图 4 中的实体 Topic-area 与图 5 中的实体 Keyword 为“同物异名”，虽然两个实体的属性不完全相同，但两者属性是兼容的，可以进行统一化。对图 5 进行修改，可得到图 6。



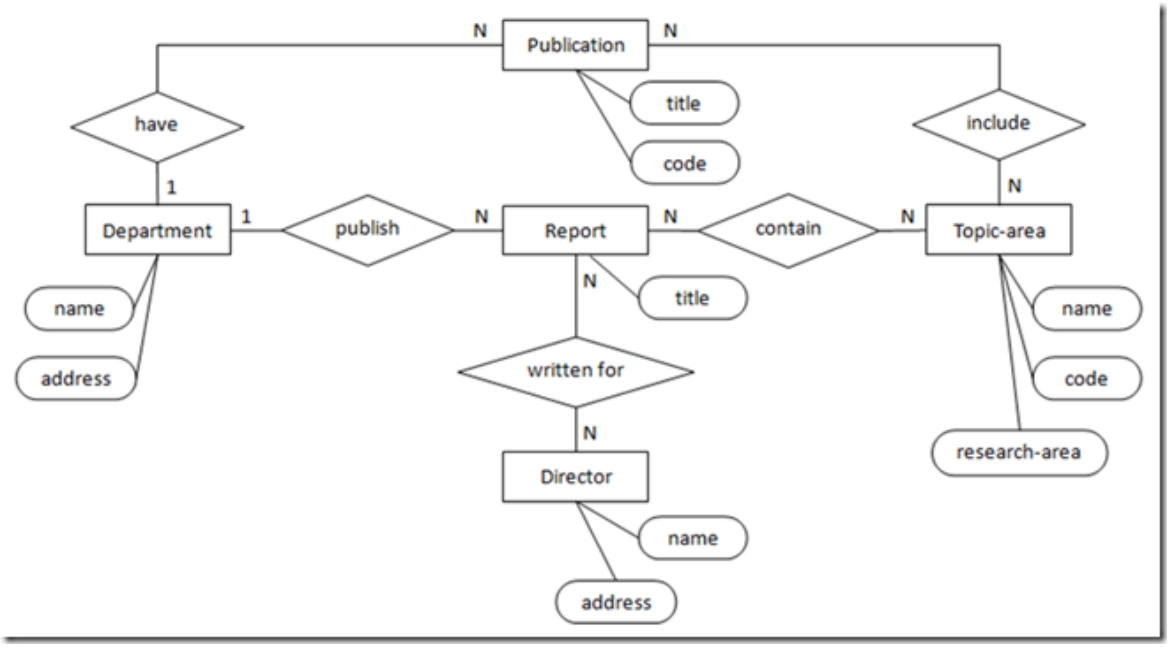
（图 6　Keyword 换为 Topic_area）

其 次，再来看两张 ER 图之间的结构性冲突。图 4 中的实体 Department 与图 5 中的属性 dept-name 为类型冲突。解决该冲突的方法是保留强类型 （实体 Department），把属性 dept-name 移至实体 Department 中。解决该冲突，把 ER 图 6 转化为 ER 图 7。



（图 7　属性 dept-name 转化为一个实体和一个属性）

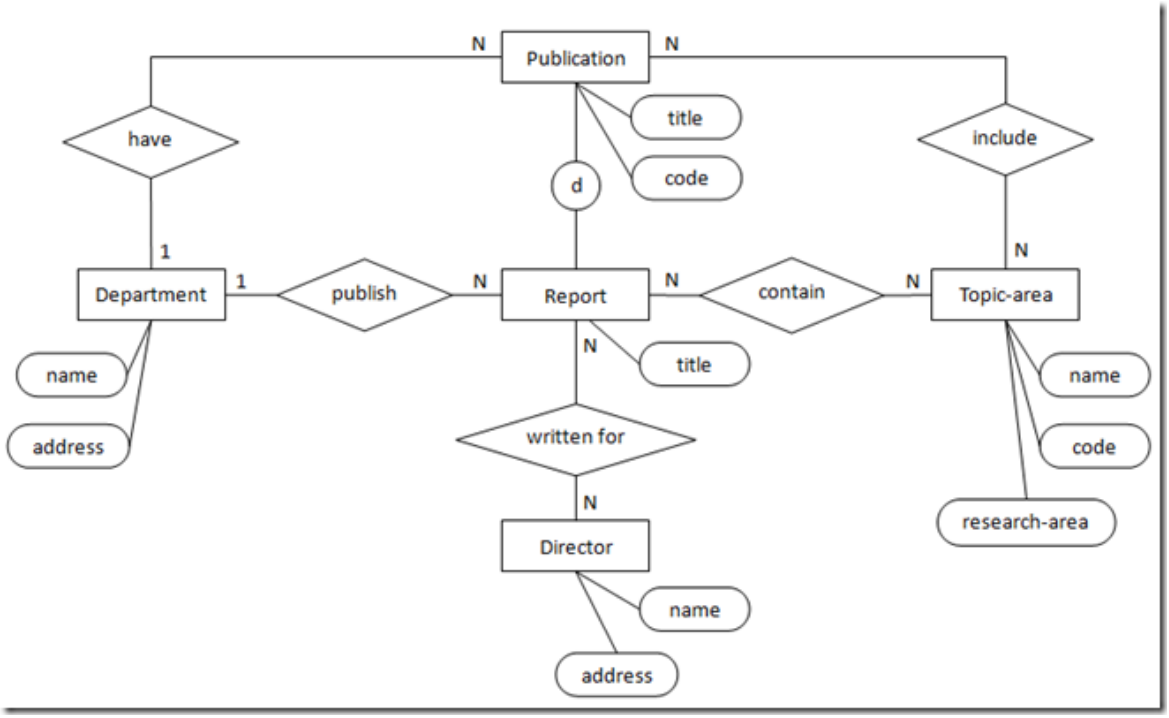
比较变化后的各局部 ER 图，寻找之间的“共同之处”进行合并。 在真正合并之前必须确认这些“共同之处”的语义概念完全等同，这也保证了合并后语义的完整性。在 ER 图 4 与 ER 图 7 中有两个共同实 体：Department 和 Topic-area，且语义一致。初步合并后的全局 ER 图如图 8 所示。



（图 8　初步合并图 4 和图 7 后的全局 ER 图）

图 8 中实体 Publication 和 Report 与实体 Department 和 Topic-area 之间的关系存在冗余。通过与用户的再次确认，了解到 Publication 是 Report 的泛化（报表只是发布材料中的一种），故不能简单的去除实体 Publication 及关系 have 和 include 来消除冗余，而可以引入泛化关系并去除冗余关系 publish 和 contain。

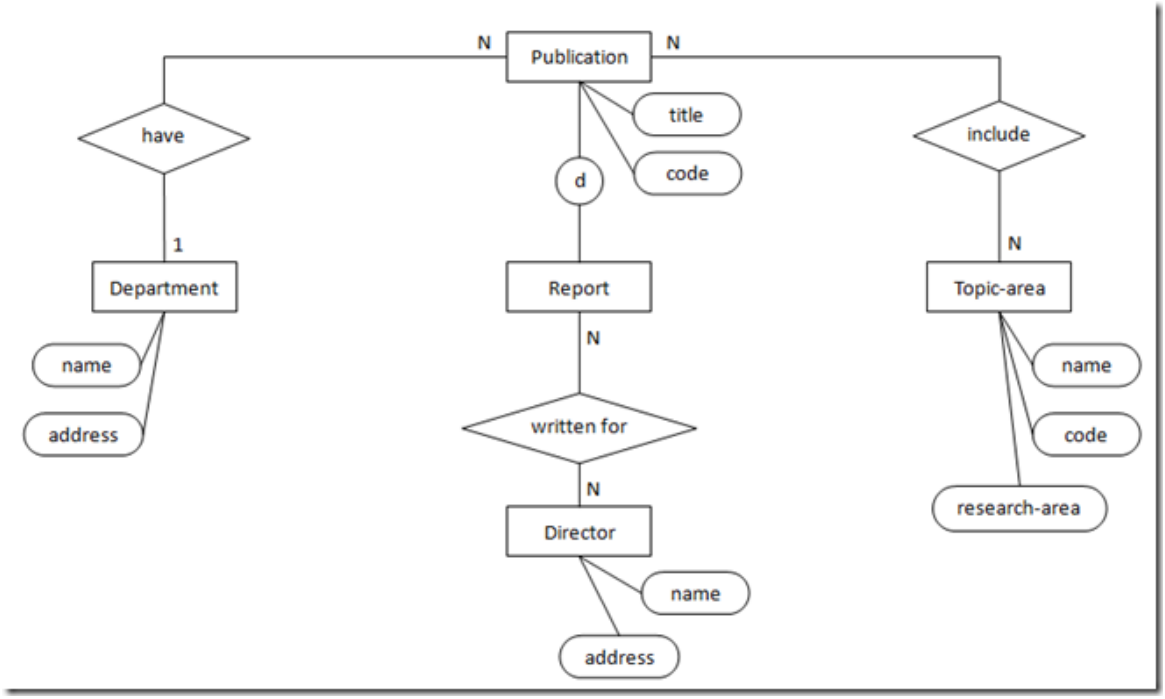
图 9 展示了增加泛化关系后的 ER 图（Publication 为超类型，Report 为子类型）。



（图 9　加入泛化关系）

图 10 中实体 Report 与实体 Department 和 Topic-area 之间的冗余关系 publish 和 contain 被去除了。Report 中的属性 title 也被去除了，因为该属性已经出现在其超类型实体 Publication

中了。



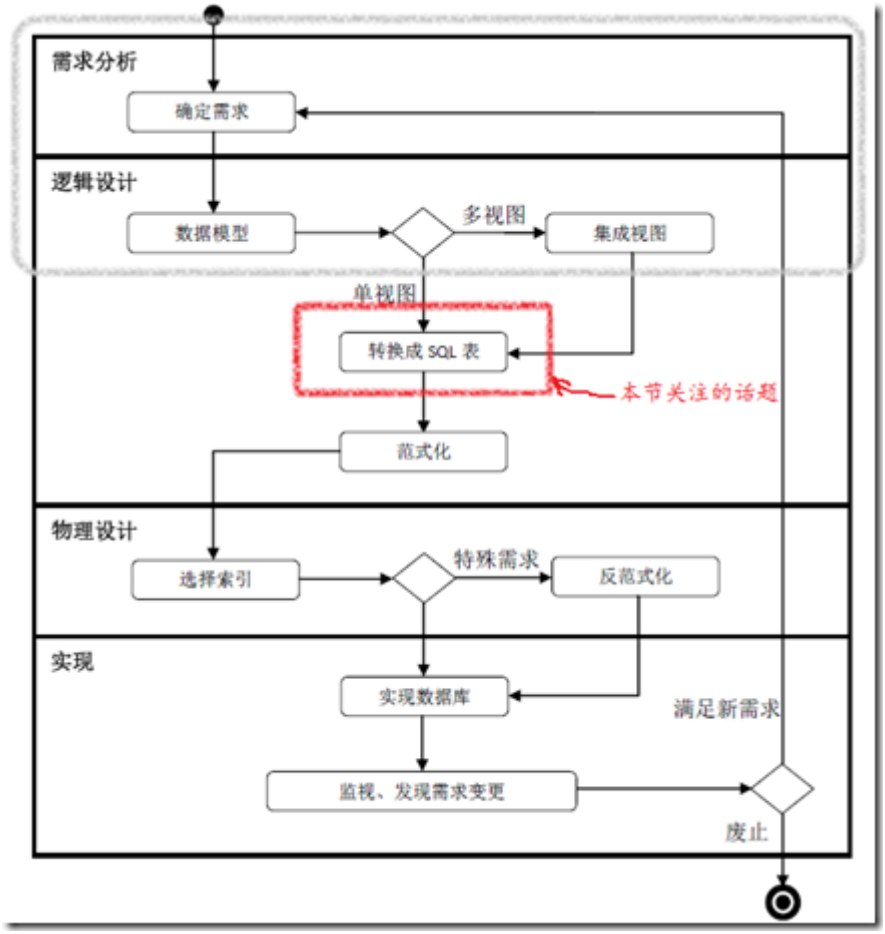
（图 10　去除冗余关系）

最终得到的 ER 图 10 达到了语义完整、最简约、易理解的目标。ER 图集成是一个持续优化和评估的过程。需要注意的是“最简约”未必会最高效。如 ER 图 10 中 去除的“publish”和“contain”关系，保留它们可能对性能有帮助。在后期的分析或测试过程中可根据需要重构 ER 图。

1. 不同的用户或用户组视角将产生不同的局部 ER 图
2. 局部 ER 图之间的冲突包括：命名冲突、类型冲突、依赖冲突、键冲突
3. 视图集成的目标是得到语义完整、形式简约且易于理解的全局 ER 图
4. 视图集成能进一步加强项目组对系统整体需求的理解与把握

数据库设计 Step by Step (9)——ER-to-SQL 转化

引言：前文（数据库设计 Step by Step (8)——视图集成）讨论了如何把局部 ER 图集成为全局 ER 图。有了全局 ER 图后，我们就可以把 ER 图转化为关系数据库中的 SQL 表了。俯瞰整个数据库生命周期（如下图所示），找到我们的“坐标”。



把 ER 图转化为关系数据库中的表结构是一个非常自然的过程。许多 ER 建模工具除了辅助绘制 ER 图外，还能自动地把 ER 图转化为 SQL 表。

从 ER 模型到 SQL 表

从 ER 图转化得到关系数据库中的 SQL 表，一般可分为 3 类。

1. 转化得到的 SQL 表与原始实体包含相同信息内容。该类转化一般适用于：

   二元“多对多”关系中，任何一端的实体

二元“一对多”关系中，“一”一端的实体

二元“一对一”关系中，某一端的实体

二元“多对多”回归关系中，任何一端的实体（注：关系两端都指向同一个实体）

三元或 n 元关系中，任何一端的实体

层次泛化关系中，超类实体

2. 转化得到的 SQL 表除了包含原始实体的信息内容之外，还包含原始实体父实体的外键。该类转化一般适用于：

二元“一对多”关系中，“多”一端的实体

二元“一对一”关系中，某一端的实体

二元“一对一”或“一对多”回归关系中，任何一端的实体

该转化是处理关系的常用方法之一，即在子表中增加指向父表中主键的外键信息。

3. 由“关系”转化得到的 SQL 表，该表包含“关系”所涉及的所有实体的外键，以及该“关系”自身的属性信息。该类转化一般适用于：

二元“多对多”关系

二元“多对多”回归关系

三元或 n 元关系

该转化是另一种常用的关系处理方法。对于“多对多”关系需要定义为一张包含两个相关实体主键的独立表，该表还能包含关系的属性信息。

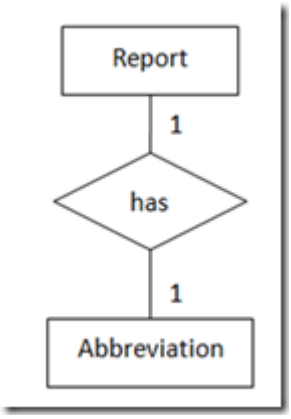
转化过程中对于 NULL 值的处理规则

1. 当实体之间的关系是可选的，SQL 表中的外键列允许为 NULL。
2. 当实体之间的关系是强制的，SQL 表中的外键列不允许为 NULL。
3. 由“多对多”关系转化得到的 SQL 表，其中的任意外键列都不允许为 NULL。

一般二元关系的转化

1. “一对一”，两实体都为强制存在

当两个实体都是强制存在的（如图 1 所示），每一个实体都对应转化为一张 SQL 表，并选择两个实体中任意一个作为主表，把它的主键放入另一个实体对应的 SQL 表中作为外键，该表称为从表。



（图 1 “一对一”，两实体都为强制存在）

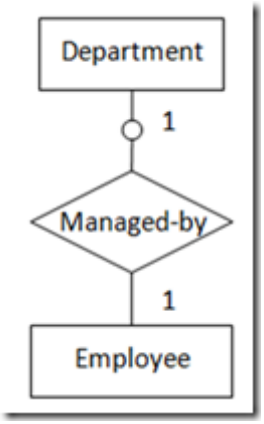
图 1 表示的语义为：每一张报表都有一个缩写，每一缩写只代表一张报表。转化得到的 SQL 表定义如下：

```
create table report
(
    report_no integer,
    report_name varchar(256),
    primary key(report_no)
);
create table abbreviation
(
    abbr_no char(6),
    report_no integer not null unique,
    primary key(abbr_no),
    foreign key(report_no) references report
        on delete cascade on update cascade
);
```

注：本节中所有 SQL 代码在 SQL Server 2008 环境中测试通过。

2. “一对一”，一实体可选存在，另一实体强制存在

当两个实体中有一个为“可选的”，则“可选的”实体对应的 SQL 表一般作为从表，包含指向另一实体的外键（如图 2 所示）。



（图 2 “一对一”，一实体可选存在，另一实体强制存在）

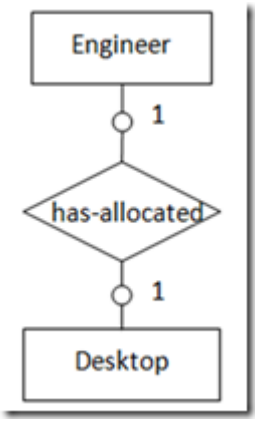
图 2 表示的语义为：每一个部门必须有一位经理，大部分员工不是经理，一名员工最多只能是一个部门的经理。转化得到的 SQL 表定义如下：

```
create table employee
(
    emp_id char(10),
    emp_name char(20),
    primary key(emp_id)
);
create table department
(
    dept_no integer,
    dept_name char(20),
    mgr_id char(10) not null unique,
    primary key(dept_no),
    foreign key(mgr_id) references employee
        on update cascade
);
```

另一种转化方式是把“可选的”实体作为主表，让“强制存在的”实体作为从表，包含外键指向“可选的”实体，这种方式外键列允许为 NULL。以图 2 为 例，可把实体 Employee 转化为从表，包含外键列 dept_no 指向实体 Department，该外键列将允许为 NULL。因为 Employee 的数 量远大于 Department 的数量，故会占用更多的存储空间。

3. “一对一”，两实体都为可选存在

当两个实体都是可选的（如图 3 所示），可选任意一个实体包含外键指向另一实体，外键列允许为 NULL 值。



（图 3 “一对一”，两实体都为可选存在）

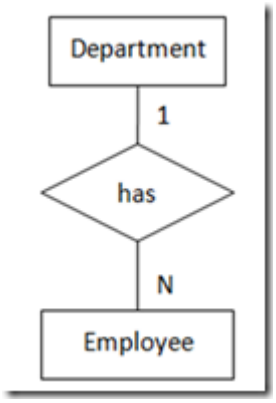
图 3 表示的语义为：部分台式电脑被分配给部分工程师，一台电脑只能分配给一名工程师，一名工程师最多只能分配到一台电脑。转化得到的 SQL 表定义如下：

```
create table engineer
(
    emp_id char(10),
    emp_name char(20),
    primary key(emp_id)
```

```
);
create table desktop
(
    desktop_no integer,
    emp_id char(10),
    primary key(desktop_no),
    foreign key(emp_id) references engineer
        on delete set null on update cascade
);
```

4. “一对多”，两实体都为强制存在

在“一对多”关系中，无论“多”端是强制存在的还是可选存在的都不会影响其转化形式，外键必须出现在“多”端，即“多”端转化为从表。当“一”端实体是可选存在时，“多”端实体表中的外键列允许为 NULL。

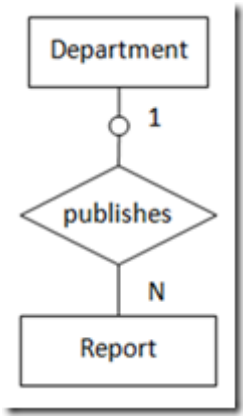


（图 4 “一对多”，两实体都为强制存在）

图 4 表示的语义为：每名员工都属于一个部门，每个部门至少有一名员工。转化得到的 SQL 表定义如下：

```
create table department
(
    dept_no integer,
    dept_name char(20),
    primary key(dept_no)
);
create table employee
(
    emp_id char(10),
    emp_name char(20),
    dept_no integer not null,
    primary key(emp_id),
    foreign key(dept_no) references department
        on update cascade
);
```

5. “一对多”，一实体可选存在，另一实体强制存在



（图 5 “一对多”，一实体可选存在，另一实体强制存在）

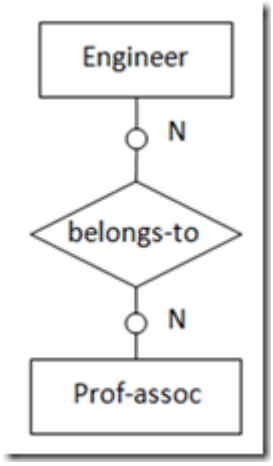
图 5 表示的语义为：每个部门至少发布一张报表，一张报表不一定由某个部门来发布。转化得到的 SQL 表定义如下：

```
create table department
(
   dept_no integer,
   dept_name char(20),
   primary key(dept_no)
);
create table report
(
   report_no integer,
   dept_no integer,
   primary key(report_no),
   foreign key(dept_no) references department
      on delete set null on update cascade
);
```

注：解释一下 report 表创建脚本的最后一行“on delete set null on update cascade”的用处。当没有这一行时，更新 department 表中 dept_no 字段会失败，删除 department 中记录也会失败，报出与外键 约束冲突的提示。如果有了最后一行，更新 department 表中 dept_no 字段，report 表中对应记录的 dept_no 也会同步更改，删除 department 中记录，会使 report 表中对应记录的 dept_no 值变为 NULL。

6. “多对多”，两实体都为可选存在

在“多对多”关系中，需要一张新关系表包含两个实体的主键。无论两边实体是否为可选存在的，其转化形式一致，关系表中的外键列不能为 NULL。实体可选存在，在关系表中表现为是否存在对应记录，而与外键是否允许 NULL 值无关。



（图 6 “多对多”，两实体都为可选存在）

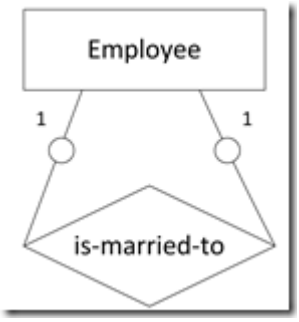
图 6 表示的语义为：一名工程师可能是专业协会的会员且可参加多个专业协会。每一个专业协会可能有多位工程师参加。转化得到的 SQL 表定义如下：

```
create table engineer
(
    emp_id char(10),
    primary key(emp_id)
);
create table prof_assoc
(
    assoc_name varchar(256),
    primary key(assoc_name)
);
create table belongs_to
(
    emp_id char(10),
    assoc_name varchar(256),
    primary key(emp_id, assoc_name),
    foreign key(emp_id) references engineer
        on delete cascade on update cascade,
    foreign key(assoc_name) references prof_assoc
        on delete cascade on update cascade
);
```

二元回归关系的转化

对于“一对一”或“一对多”回归关系的转化都是在 SQL 表中增加一列与主键列类型、长度相同的外键列指向实体本身。外键列的命名需与主键列不同，表明其用意。外键列的约束根据语义进行确定。

7. “一对一”，两实体都为可选存在

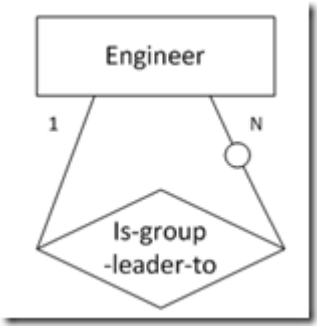


（图 7 “一对一”，两实体都为可选存在）

图 7 表示的语义为：公司员工之间可能存在夫妻关系。转化得到的 SQL 表定义如下：

```
create table employee
(
   emp_id char(10),
   emp_name char(20),
   spouse_id char(10),
   primary key(emp_id),
   foreign key(spouse_id) references employee
);
```

8. “一对多”，“一”端为强制存在，“多”端为可选存在



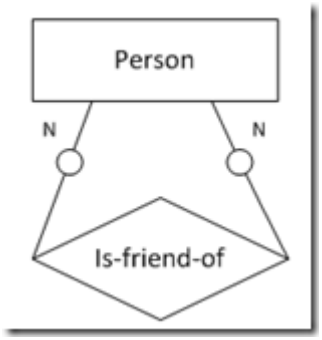
（图 8 “一对多”，“一”端为强制存在，“多”端为可选存在）

图 8 表示的语义为：工程师被分为多个组，每个组有一名组长。转化得到的 SQL 表定义如下：

```
create table engineer
(
   emp_id char(10),
   leader_id char(10) not null,
   primary key(emp_id),
   foreign key(leader_id) references engineer
);
```

“多对多”回归关系无论是可选存在的还是强制存在的都需新增一张关系表，表中的外键列须为 NOT NULL。

9. “多对多”，两端都为可选存在



（图 9 “多对多”，两端都为可选存在）

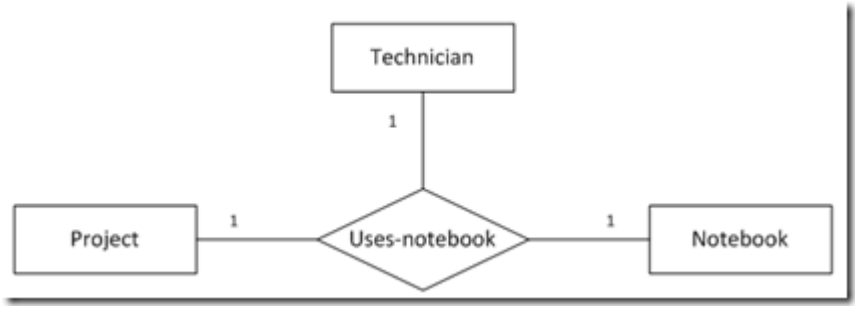
图 9 表示的语义为：社交网站中人之间的朋友关系，每个人都可能有很多朋友。转化得到的 SQL 表定义如下：

```
create table person
(
   person_id char(10),
   person_name char(20),
   primary key(person_id)
);
create table friend
(
   person_id char(10),
   friend_id char(10),
   primary key(person_id, friend_id),
   foreign key(person_id) references person,
   foreign key(friend_id) references person,
   check(person_id < friend_id)
);
```

三元和 n 元关系的转化

无论哪种形式的三元关系在转化时都会创建一张关系表包含所有实体的主键。三元关系中，“一”端实体的个数决定了函数依赖的数量。因此，“一对一对 一”关系有三个函数依赖式，“一对一对多”关系有两个函数依赖式，“一对多对多”关系有一个函数依赖式。“多对多对多”关系的主键为所有外键的联合。

10. “一对一对一”三元关系



（图 10 “一对一对一”三元关系）

图 10 表示的语义为：

1 名技术员在 1 个项目中使用特定的 1 本记事簿

1 本记事簿在 1 个项目中只属于 1 名技术员

1 名技术员的 1 本记事簿只用于记录 1 个项目

注：1 名技术员仍可以做多个项目，对于不同的项目维护不同的记事簿。

转化得到的 SQL 表定义如下：

```
create table technician
(
   emp_id char(10),
   primary key(emp_id)
);
create table project
(
   project_name char(20),
   primary key(project_name)
);
create table notebook
(
   notebook_no integer,
   primary key(notebook_no)
);
create table uses_notebook
(
   emp_id char(10),
   project_name char(20),
   notebook_no integer not null,
   primary key(emp_id, project_name),
   foreign key(emp_id) references technician
      on delete cascade on update cascade,
   foreign key(project_name) references project
      on delete cascade on update cascade,
   foreign key(notebook_no) references notebook
      on delete cascade on update cascade,
   unique(emp_id, notebook_no),
   unique(project_name, notebook_no)
);
```

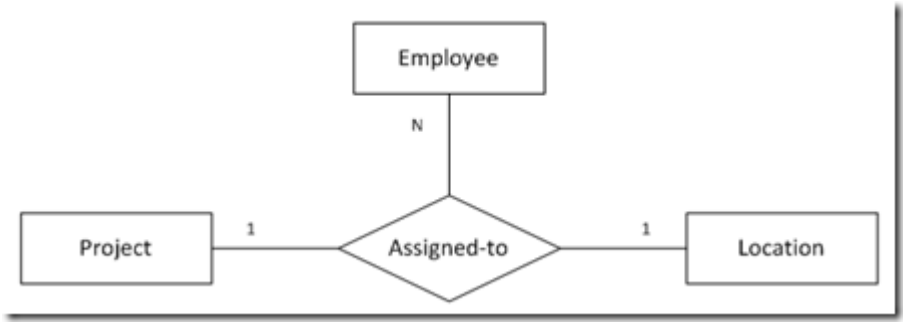
函数依赖

emp_id, project_name -> notebook_no

emp_id, notebook_no -> project_name

project_name, notebook_no -> emp_id

11. “一对一对多”三元关系



（图 11 “一对一对多”三元关系）

图 11 表示的语义为：

参与 1 个项目的 1 名员工只会在 1 个地点做该项目

1 名员工在 1 个地点只能做 1 个项目

1 个地点的 1 个项目可能有多名员工参与

注：1 名员工可以在不同的地点做不同的项目

转化得到的 SQL 表定义如下：

```
create table employee
(
   emp_id char(10),
   emp_name char(20),
   primary key(emp_id)
);
create table project
(
   project_name char(20),
   primary key(project_name)
);
create table location
(
   loc_name char(15),
   primary key(loc_name)
);
create table assigned_to
(
   emp_id char(10),
   project_name char(20),
   loc_name char(15) not null,
   primary key(emp_id, project_name),
   foreign key(emp_id) references employee
      on delete cascade on update cascade,
   foreign key(project_name) references project
      on delete cascade on update cascade,
```

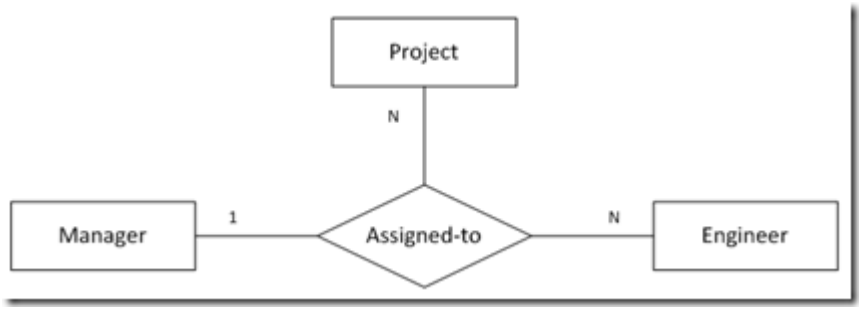
```
    foreign key(loc_name) references location
       on delete cascade on update cascade,
    unique(emp_id, loc_name)
);
```

函数依赖：

emp_id, loc_name -> project_name

emp_id, project_name -> loc_name

12. “一对多对多”三元关系



（图 12 “一对多对多”三元关系）

图 12 表示的语义为：

1 个项目中的 1 名工程师只会有 1 名经理

1 个项目中的 1 名经理会带领多名工程师做该项目

1 名经理和他手下的 1 名工程师可能参与多个项目

转化得到的 SQL 表定义如下：

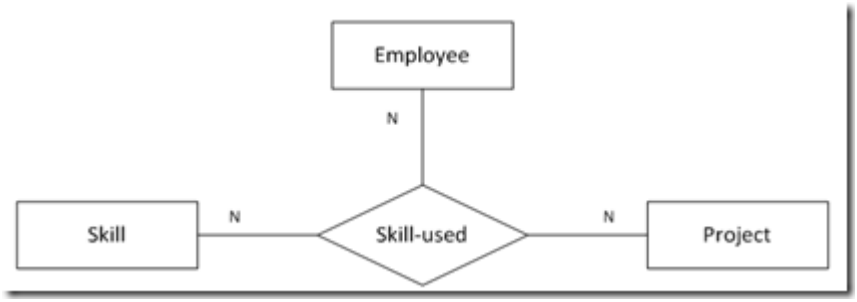
```
create table project
(
    project_name char(20),
    primary key(project_name)
);
create table manager
(
    mgr_id char(10),
    primary key(mgr_id)
);
create table engineer
(
    emp_id char(10),
    primary key(emp_id)
);
create table manages
(
    project_name char(20),
```

```
    mgr_id char(10) not null,
    emp_id char(10),
    primary key(project_name, emp_id),
    foreign key(project_name) references project
        on delete cascade on update cascade,
    foreign key(mgr_id) references manager
        on delete cascade on update cascade,
    foreign key(emp_id) references engineer
        on delete cascade on update cascade
);
```

函数依赖：

project_name, emp_id -> mgr_id

13. “多对多对多”三元关系



（图 13 “多对多对多”三元关系）

图 13 表示的语义为：

1 名员工在 1 个项目中可以运用多种技能

1 名员工的 1 项技能可以在多个项目中运用

1 个项目中的 1 项技能可以被参与该项目的多名员工运用

转化得到的 SQL 表定义如下：

```
create table employee
(
    emp_id char(10),
    emp_name char(20),
    primary key(emp_id)
);
create table skill
(
    skill_type char(15),
    primary key(skill_type)
);
create table project
(
```

```
    project_name char(20),
    primary key(project_name)
);
create table skill_used
(
    emp_id char(10),
    skill_type char(15),
    project_name char(20),
    primary key(emp_id, skill_type, project_name),
    foreign key(emp_id) references employee
        on delete cascade on update cascade,
    foreign key(skill_type) references skill
        on delete cascade on update cascade,
    foreign key(project_name) references project
        on delete cascade on update cascade
);
```

函数依赖：

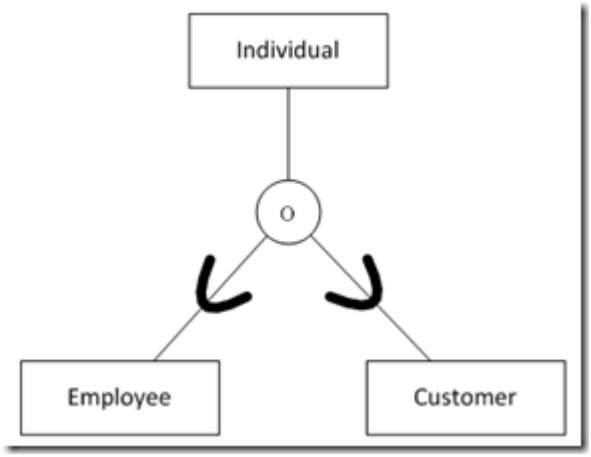
无

泛化与聚合

泛化抽象结构中的超类实体和各子类实体分别转化为对应的 SQL 表。超类实体转化得到的表包含超类实体的键和所有公共属性。子类实体转化得到的表包含超类实体的键和子类实体特有的属性。

要保证泛化层次中数据的完整性就必须保证某些操作在超类表和子类表的之间的同步。若超类表的主键需做更新，则子类表中对应记录的外键必须一起更新。 若需删除超类表中的记录，子类表中对应记录也需一起删除。我们可以在定义子类表时加入外键级联约束。这一规则对于覆盖与非覆盖的子类泛化都适用。

14. 泛化层次关系



（图 14 泛化层次关系）

图 14 表示的语义为：

个人可能是一名员工，或是一位顾客，或同时是员工与顾客，或两者都不是

转化得到的 SQL 表定义如下：

```
create table individual
(
   indiv_id char(10),
   indiv_name char(20),
   indiv_addr char(20),
   primary key(indiv_id)
);
create table employee
(
   emp_id char(10),
   job_title char(15),
   primary key(emp_id),
   foreign key(emp_id) references individual
      on delete cascade on update cascade
);
create table customer
(
   cust_no char(10),
   cust_credit char(12),
   primary key(cust_no),
   foreign key(cust_no) references individual
      on delete cascade on update cascade
);
```

有些数据库开发者还会在超类表中增加一个鉴别属性。鉴别属性对于每一种子类有不同的值，表示从哪一个子类中能获得进一步的信息。

聚合抽象的转化方式也是为超类实体和每一个子类实体生成 SQL 表，但聚合中的超类与子类没有公共属性和完整性约束。聚合的主要功能是提供一种抽象来辅助视图集成的过程。

转化步骤

以下总结了从 ER 图到 SQL 表的基本转化步骤

1. 把每一个实体转化为一张表，其中包含键和非键属性。
2. 把每一个“多对多”二元或二元回归关系转化为一张表，其中包含实体的键和关系的属性。
3. 把三元及更高元（n 元）关系转化为一张表。

让我们一一对这三个步骤进行讨论。

实体转化

若两个实体之间是“一对多”关系，把“一”端实体的主键加入到“多”端实体表中作为外键。若两实体间是“一对一”关系，把某个“一”端实体的主键放 入另一实体表中作为外键，加入外键的实体理论上可以任选，但一般会遵循如下原则：按照实体间最为自然的父子关系，把父实体的键放入子实体中；另一种策略是 基于效率，把外键加入到具有较少行的表中。

把泛化层次中的每一个实体转化为一张表。每张表都会包含超类实体的键。事实上子类实体的主键同时也是外键。超类表中还包含所有相关实体的公共非键属性，其他表包含每一子类实体特有的非键属性。

转化得到的 SQL 表可能会包含 not null, unique, foreign key 等约束。每一张表必须有一个主键（primary key），主键隐含着 not null 和 unique 约束。

“多对多”二元关系转化

每一个“多对多”二元关系能转化为一张表，包含两个实体的键和关系的属性。

这一转化得到的 SQL 表可能包含 not null 约束。在这里没有使用 unique 约束的原因是关系表的主键是由各实体的外键复合组成的，unique 约束已隐含。

三元关系转化

每一个三元（或 n 元）关系转化为一张表，包含相关实体的 n 个主键以及该关系的属性。

这一转化得到的表必须包含 not null 约束。关系表的主键由各实体的外键复合组成。n 元关系表具有 n 个外键。除主键约束外，其他候选键(candidate key)也应加上 unique 约束。

ER-to-SQL 转化步骤示例

把数据库设计 Step by Step (7)——概念数据建模中最后得到的公司人事和项目数据库的全局 ER 图（图 9）转化为 SQL 表。

1. 直接由实体生成的 SQL 表有：

Division　Department　Employee　Manager　Secretary　Engineer

Technician　Skill　Project　Location　Prof_assoc　Desktop

Workstation

2. 由“多对多”二元关系及“多对多”二元回归关系生成的 SQL 表有：

belongs_to

3. 由三元关系生成的 SQL 表有：

skill_used　assigned_to

总结与回顾

1. 通过一些简单的规则就能把 ER 模型中的实体、属性和关系转化为 SQL 表。

2. 实体在转化为表的过程中，其中的属性一一被映射为表的属性。

3. “一对一”或“一对多”关系中的“子”端实体转化成的 SQL 表必须包含另一端实体的主键，作为外键。

4. “多对多”关系转化为一张表，包含相关实体的主键，复合组成其自身的主键。同时这些键在 SQL 中定义为外键分别指向各自的实体。

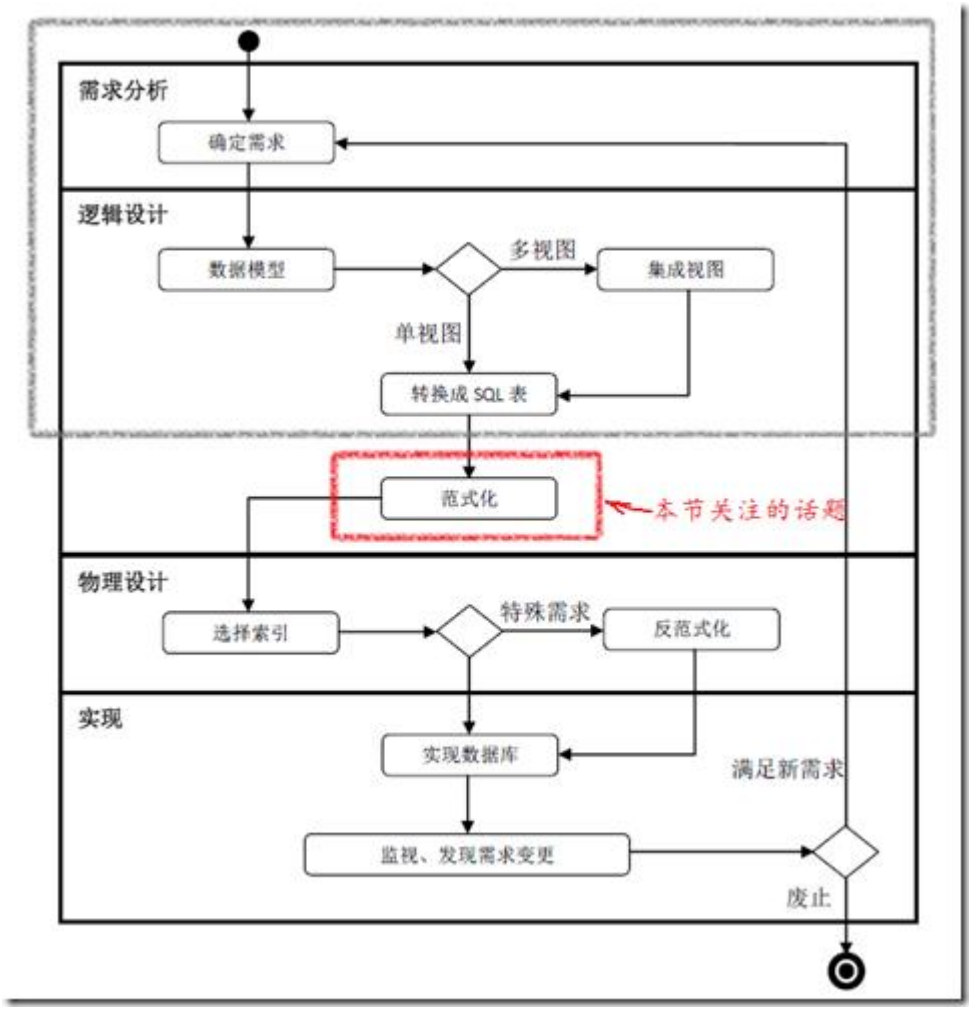
5. 三元或 n 元关系被转化为一张表，包含相关实体的主键。这些键在 SQL 中定义为外键。这些键中的子集定义为主键，其基于该关系的函数依赖。

6. 泛化层次的转化规则要求子类实体从超类实体继承主键。

7. ER 图中的可选约束在转化为 SQL 时，表现为关系的某一端实体允许为 null。在 ER 图中没有明确标识可选约束时，创建表时默认 not null 约束。

数据库设计 Step by Step (10)——范式化

引言：前文（数据库设计 Step by Step (9)——ER-to-SQL 转化）讨论了如何把 ER 图转化为关系表结构。本文将介绍数据库范式并讨论如何范式化候选表。我们先来看一下此刻处在数据库生命周期中的位置（如下图所示）。



前几篇博文中我们详细的讨论了 ER 建模的方法。精心设计的 ER 模型将帮助我们直接得到范式化的表或只需稍许修改即为范式化的表，设计、绘制 ER 图的重要性也体现在这里。概念数据建模（ER 建模）从一开始就潜移默化的引导着我们走向范式化的数据库表结构。

本文的讨论将始于第一范式，止于 BCNF 范式。在现实数据库设计中，一般需达到的范式化目标是第三或 BCNF 范式，更高级别的范式更多的是理论价值，本文也不将涉及。

范式基础

关 系数据库中的表有时会面对性能、一致性和可维护性方面的问题。举例来说，把整个数据库中的数据都定义在一张表中将导致大量冗余的数据，低效的查询和更新性 能，对某些数据的删除将造成有用数据的丢失等。如图 1 所示，products, salespersons, customers, orders 都存储在一张名为 Sales 的表中。

product_name	order_no	cust_name	cust_addr	credit	date	sales_name
vacuum cleaner	1435	Dave	Austin	6	2010-03-01	Carl
computer	2730	Qiang	Plymouth	10	2011-04-15	Ted
refrigerator	2460	Mike	Ann Arbor	8	2010-09-12	Dick
DVD player	519	Peter	Detroit	3	2010-12-05	Fred
radio	1986	Charles	Chicago	7	2011-05-10	Richard
CD player	1817	Eric	Mumbai	8	2010-08-03	Paul
vacuum cleaner	1865	Charles	Chicago	7	2010-10-01	Carl
vacuum cleaner	1885	Betsy	Detroit	8	2009-04-19	Carl
refrigerator	1943	Dave	Austin	6	2011-01-04	Dick
television	2315	Sakti	East Lansing	6	2011-03-15	Fred

（图 1 Sales 表）

在 这张表中，某些产品和客户信息是冗余的，浪费了存储空间。某些查询如“上个月哪些客户订购了吸尘器”需要搜索整张表。当要修改客户 Dave 的地址需要更新 该表的多条记录。最后删除客户 Qiang 的订单（2730）将造成该客户姓名、地址、信用级别信息的丢失，因为该客户只有唯一这个订单。

如果我们通过一些方法把该大表拆分成多个小表，从而消除上述这些问题使数据库更为高效和可靠，范式化就是为了达到这一目标。范式化是指通过分析表中各属性之间的互相依赖，并把大表映射为多个小表的过程。

第一范式（1NF）

定义：当且仅当一张表的所有列只包含原子值时，即表中每行中的每一个列只有一个值，则该表符合第一范式。

图 1 中的 Sales 表的每一行、每一列中只有原子值，故 Sales 表满足第一范式。

为了更好的理解第一范式，我们讨论一下域、属性、列之间的差异。

> 域是某属性所有可能值的集合，但同一个域可能被用于多个属性上。举例来说，人名的域包含所有可能的姓名集合，在图 1 的表中可用于 cust_name 或 sales_name 属性。每一列代表一个属性，有些情况下代表不同属性的多个列具有相同的域，这并不会违反第一范式，因为表中的值仍是原子的。

仅符合第一范式的表常会遇到数据重复、更新性能以及更新一致性等问题。为了更好的理解这些问题，我们必须定义键的概念。

超键是一个或多个属性的集合，其能帮助我们唯一确定一条记录。若组成超键属性列的子集仍为一个超键，但该子集少了任何一个属性都将使其不再是一个超键，则该属性列子集称为候选键。主键是从一张表的候选键集合中任意挑选出的，作为该表的一个索引。

作为一个例子，图 2 中表的所有属性组成一个超键。

report_no	editor	dept_no	dept_name	dept_addr	author_id	author_name	author_addr
4216	woolf	15	design	argus1	53	mantei	cs-tor
4216	woolf	15	design	argus1	44	bolton	mathrev
4216	woolf	15	design	argus1	71	koenig	mathrev
5789	koenig	27	analysis	argus2	26	fry	folkstone
5789	koenig	27	analysis	argus2	38	umar	prise
5789	koenig	27	analysis	argus2	71	koenig	mathrev

（图 2 Report 表）

在关系模型中不允许有重复的行，因此一个明显的超键是表的所有列（属性）的组合。假设表中每一个部门的地址（dept_addr）都相同，则除 dept_addr 之外的属性仍然是一个超键。对其他属性作类似的假设，逐步缩小属性的组合。我们发现属性组合 report_no, author_id 能唯一确定表中的其他属性，即是一个超键。同时 report_no 或 author_id 中的任意一个都无法唯一确定表中的一行，故属性 组合 report_no, author_id 是一个候选键。由于它们是该表的唯一候选键，它们也是该表的主键。

一张表能有多个候选 键。举例来说，在图 2 中，若有附加列 author_ssn（SSN：社会保险号），属性组合 report_no, author_ssn 也能唯一确定表中的其他属性。因此属性组合（report_no, author_id）和（report_no, author_ssn）都是候选键，可以任选其一作为主键。

第二范式（2NF）

为 了解释第二以及更高级别范式。我们需引入函数依赖的概念。一个或多个属性值能唯一确定一个或多个其他属性值称为函数依赖。给定某表（R），一组属性（B） 函数依赖于另一组属性（A），即在任意时刻每个 A 值只与唯一的 B 值相关联。这一函数依赖用 A –> B 表示。以图 2 中的表为例，表 report 的函数依赖如下：

report: report_no –> editor, dept_no

dept_no –> dept_name, dept_addr

author_id –> author_name, author_addr

定义：一张表满足第二范式（2NF）的条件是当且仅当该表满足第一范式且每个非键属性完全依赖于主键。当一个属性出现在函数依赖式的右端，且函数依赖式的左端为表的主键或可由主键传递派生出的属性组，则该属性完全依赖于主键。

report 表中一个传递函数依赖的例子：

report_no –> dept_no

dept_no –> dept_name

因为我们能派生出函数依赖（report_no –> dept_name），即 dept_name 传递依赖于 report_no。

继 续我们的例子，图 2 中表的复合键（report_no, author_id）是唯一的候选键，即为表的主键。该表存在一个 FD（dept_no –> dept_name, dept_addr），其左端没有主键的任何组成部分。该表的另两个 FD（report_no –> editor, dept_no 和 author_id –> author_name, author_addr）的左端包含主键的一部分但不是全部。故 report 表的任何一条 FD 都不满足第二范式的条件。

思考一下仅满足 第一范式的 report 表的缺陷。report_no, editor 和 dept_no 对该 Report 的每一位 author 都需要重复，故当 Report 的 editor 需要变更时，多条记录必须同步修改。这就 是所谓的更新异常（update anomaly），冗余的更新会降低性能。当没有把所有符合条件的记录同步更新时，还会造成数据的不一致。若要在表中加入一位新的 author，只有在该 author 参与了某 Report 的撰写才能插入该 author 的记录，这就是所谓的插入异常（insert anomaly）。最后，若某一张 Report 无效了，所有与该 Report 相关联的记录必须一起删除。这可能造成 author 信息的丢失（与该 Report 相关联的 author_id, author_name, author_addr 也被删除了）。这一副作用被称为删除异常（delete anomaly），使数据丧失了完整性。

上述这些缺陷可通过把仅满足第一范式的表转化为多张满足第二范式的表来克服。在保留原先函数依赖和语义关系的前提下，把 Report 表映射为三张小表（report1, report2, report3），其中包含的数据如图 3 所示。

Report 1

report_no	editor	dept_no	dept_name	dept_addr
4216	woolf	15	design	argus 1
5789	koenig	27	analysis	argus 2

Report 2

author_id	author_name	author_addr
53	mantei	cs-tor
44	bolton	mathrev
71	koenig	mathrev
26	fry	folkstone
38	umar	prise
71	koenig	mathrev

Report 3

report_no	author_id
4216	53

4216	44
4216	71
5789	26
5789	38
5789	71

（图 3　2NF 表）

这些满足第二范式表的函数依赖为：

report1: report_no –> editor, dept_no

dept_no –> dept_name, dept_addr

report2: author_id –> author_name, author_addr

report3: report_no, author_id 为候选键，无函数依赖

现在我们已得到了三张满足第二范式的表，消除了第一范式表存在的最糟糕的问题。第一、editor, dept_no, dept_name, dept_addr 不再需要为每一位 author 重复。第二、更改一位 editor 只需要更新 report1 的一条记录。第三、删除 report 不再会造 成 author 信息丢失的副作用。

我们可以注意到这三张满足第二范式的表可以直接从 ER 图转化得到。ER 图中的 Author、Report 实体以及之间的“多对多”关系可根据上一篇博文（数据库设计 Step by Step (9)——ER-to-SQL 转化）的规则很自然的转化为三张表。

第三范式（3NF）

第 二范式相对于第一范式已经有了巨大的进步，但由于存在传递依赖（transitive dependency），满足第二范式的表仍会存在数据操作异常（anomaly）。当一张表中存在传递依赖，其意味着该表中描述了两个单独的事实。每个 事实对应于一条函数依赖，函数依赖的左侧各不相同。举例来说，删除一个 report，其包含删除 report1 和 report3 表中的相应记录(如图 3 所 示)，该删除动作的副作用是 dept_no, dept_name, dept_addr 信息也被删除了。如果把表 report1 映射为包含列 report_no, editor, dept_no 的表 report11 和包含列 dept_no, dept_name, dept_addr 的表 report12（如图 4 所示），我们就能消除上述问题。

Report11

report_no	**editor**	**dept_no**
4216	woolf	15
5789	koenig	27

Report12

dept_no	**dept_name**	**dept_addr**
15	design	argus 1
27	analysis	argus 2

Report 2

author_id	author_name	author_addr
53	mantei	cs-tor
44	bolton	mathrev
71	koenig	mathrev
26	fry	folkstone
38	umar	prise
71	koenig	mathrev

Report 3

report_no	author_id
4216	53
4216	44
4216	71
5789	26
5789	38
5789	71

（图 4　3NF 表）

定　义：一张表满足第三范式（3NF）当且仅当其每个非平凡函数依赖 X –> A，其中 X 和 A 可为简单或复合属性，必须满足以下两个条件之一。1. X 为超键　或　2. A 为某候选键的成员。若 A 为某候选键的成员，则 A 被称为主属性。注：平凡函数依赖的形式为 YZ –> Z。

在上述例子中通过把 report1 映射为 report11 和 report12，消除了传递依赖 report_no –> dept_no –> dept_name, dept_addr，我们得到了如图 4 所示的第三范式表及函数依赖：

report11: repot_no –> editor, dept_no

report12: dept_no –> dept_name, dept_addr

report2:　author_id –> author_name, author_addr

report3:　report_no, author_id 为候选键（无函数依赖）

Boyce-Codd 范式（BCNF）

第三范式消除了大部分的异常，也是商业数据库设计中达到的最普遍的标准。剩下的异常情况可通过 Boyce-Codd 范式（BCNF）或更高级别范式来消除。BCNF 范式可看作加强的第三范式。

定义：一张表 R 满足 Boyce-Codd 范式（BCNF），若其每一条非平凡函数依赖 X –> A 中 X 为

超键。

BCNF 范式是比第三范式更高级别的范式因为其去除了第三范式中的第二种条件（允许函数依赖右侧为主属性），即表的每一条函数依赖的左侧必须为超键。每一张满足 BCNF 范式的表同时满足第三范式、第二范式和第一范式。

以下的例子展示了一张满足第三范式但不满足 BCNF 范式的表。这样的表和那些仅满足较低范式的表一样存在删除异常。

断言 1：一个小组里的每一名员工只由一位领导来管理。一个小组可能有多位领导。

emp_name, team_name –> leader_name

断言 2：每一位领导只会参与一个组的管理。

leader_name –> team_name

emp_name	**team_name**	**leader_name**
Sutton	Hawks	Wei
Sutton	Condors	Bachmann
Niven	Hawks	Wei
Niven	Eagles	Makowski
Wilson	Eagles	DeSmith

（图 5　team 表）

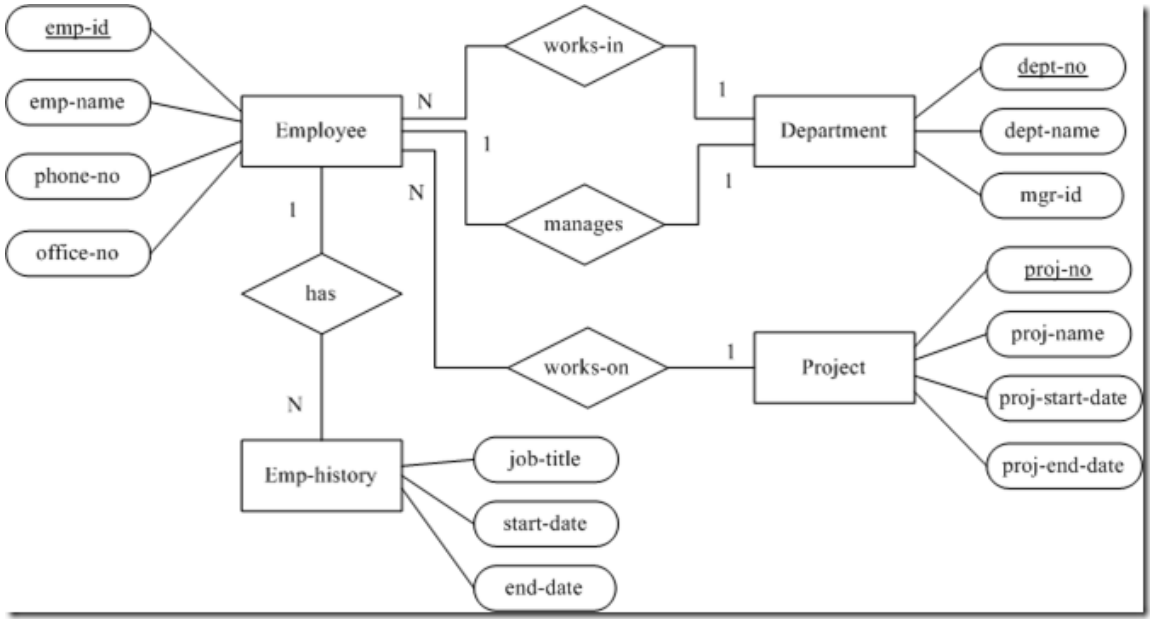
team 表满足第三范式，具有复合候选键 emp_name, team_name

team 表有如下删除异常：若 Sutton 离开了 Condors 组，Bachmann 为 Condors 组的领导这一信息将丢失。

消除这一删除异常最简单的方法是根据两条断言创建两张表，通过两张表中冗余的信息来消除删除异常。这一分解是无损的并保持了所有原先的函数依赖，但这降低了更新性能，并需要更多存储空间。为了避免删除异常，这样做是值得的。

注：无损分解是指把一张表分解为两张小表后，通过对两张小表进行 natural join 得到的表与原始表相同，不会产生任何多余行。

数据库范式化示例



（图 6　employee 数据库 ER 图）

该案例基于图 6 中的 ER 模型和以下相关函数依赖。一般而言，函数依赖可通过分析 ER 图及业务经验推得。

- emp_id, start_date –> job_title, end_date
- emp_id –> emp_name, phone_no, office_no, proj_no, proj_name, dept_no
- phone_no –> office_no
- proj_no –> proj_name, proj_start_date, proj_end_date
- dept_no –> dept_name, mgr_id
- mgr_id –> dept_no

我们的目标是设计至少能达到第三范式（3NF）的关系数据库表结构，并尽可能减少表的数量。

如 果将函数依赖 1 至 6 放入一张表，并设置复合主键：emp_id, start_date，那么我们违反了第三范式，因为函数依赖 2 至 6 的等式左侧不是超键。因此，我们需要把函数依赖 1 从其余的函数依赖中分离出来。如果将 函数依赖 2 至 6 进行合并，我们将得到很多传递依赖。故函数依赖 2、3、4、5 必须分到不同的表中。我们再来考虑函数依赖 5 和 6 是否能在不违反第三范式的前 提下进行合并。因为 mgr_id 和 dept_no 是相互依赖的，这两个属性在表中都是超键，所以可以合并。

通过合理的映射函数依赖 1 至 6，我们能得到如下表：

emp_hist:　emp_id, start_date –> job_title, end_date

employee:　emp_id –> emp_name, phone_no, proj_no, dept_no

phone:　phone_no –> office_no

project:　proj_no –> proj_name, proj_start_date, proj_end_date

department: dept_no –> dept_name, mgr_id

mgr_id –> dept_no

这一解决方案涵盖了所有函数依赖。满足第三范式和 BCNF 范式，同时该方案创建了最少数量的表。

范式化从 ER 图得到的候选表

在数据库生命周期中，对表的范式化是通过分析表的函数依赖完成的。这些函数依赖包括：从需求分析中得到的函数依赖；从 ER 图中得到的函数依赖；从直觉中得到的函数依赖。

主函数依赖代表了实体键之间的依赖。次函数依赖代表实体内数据元素间的依赖。一般来说，主函数依赖可从 ER 图中得到，次函数依赖可从需求分析中得到。表 1 展示了每种基本 ER 构件所能得到的主函数依赖。

关系的度（Degree）	关系的连通数（Connectivity）	主函数依赖
二元或二元回归	“一对一” “一对多” “多对多”	2 个：键（“一”侧） –> 键（“一”侧） 1 个：键（“多”侧） –> 键（“一”侧） 无（由两侧键组成的组合键）
三元	“一对一对一” “一对一对多” “一对多对多” “多对多对多”	3 个：键（“一”），键（“一”） –> 键（“一”） 2 个：键（“一”），键（“多”） –> 键（“一”） 1 个：键（“多”），键（“多”） –> 键（“一”） 无（有三侧键组成的组合键）
泛化	无	无

每 个候选表一般会有多个主函数依赖和次函数依赖，这决定了当前表的范式化程度。对每个表采用各种技术使其达到需求规格中要求的范式化程度，在范式化过程中要 保证数据完整性，即范式化后得到的表应包含原先候选表的所有函数依赖。精心设计的概念数据模型通常能得到基本已范式化的表，后期的范式化处理不会很困难，所以概念数据建模非常重要。

主要内容回顾

1. 不良的表结构设计将导致表数据的更新异常（update anomaly）、插入异常（insert anomaly）、删除异常（delete anomaly）

2. 范式化通过消除冗余数据，来解决数据库存在的一致性、完整性和可维护性等方面的问题。

3. 在实际数据库设计中，范式化的目标一般是达到第三范式或 BCNF 范式。

4. 精心设计的概念数据模型（ER 模型）能帮助我们得到范式化的表。

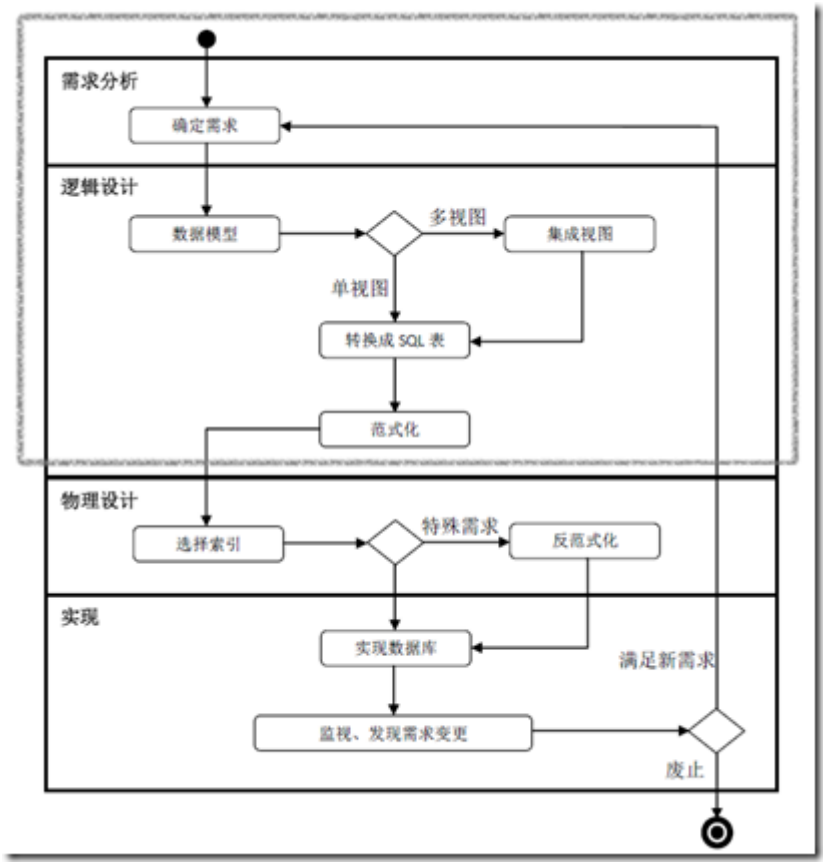
数据库范式化参考资料

1. Database Normalization（http://en.wikipedia.org/wiki/Database_normalization）

2. 3 Normal Forms Database Tutorial（http://www.phlonx.com/resources/nf3/）

数据库设计 Step by Step (11)——通用设计模式

引言：前文（数据库设计 Step by Step (10)——范式化）我们详细讨论了关系数据库范式，始于第一范式止于 BCNF 范式。至此我们完成了数据库的逻辑设计，如下图所示。



正如首篇博文数据库设计 Step by Step (1)——扬帆启航中 介绍的，本系列博文关注通用于所有关系数据库的需求分析与逻辑设计部分。无论你使用的是 Oracle，SQL Server，Sybase 等商业数据库，亦或是如 MySQL，SQLite 等开源数据库都能运用这些设计方法来优化设计。数据库的物理设计及实现部分与 数据库产品密切相关、各有差异，且内容也非常丰富，故不在本系列中讨论。

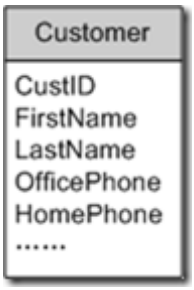
本篇博文将分为两个部分，第一部分将介绍数据库设计的一些通用模式，第二部分将对本系列的内容做一个整理并给出一些参考资料供大家参考。

这一小节我们将分析一些较为常见的业务场景，并给出对于这些场景的表结构设计方法。这些方法可以放入我们自己的数据库设计工具箱，当在面对现实需求时可灵活加以运用。

多值属性

多值属性很常见，如淘宝网中每个用户都可以设置多个送货地址，又如在 CRM 系统中客户可以有多个电话号，一个号码用于工作时间，另一个用于下班时间等。

以存储客户的联系电话为例。联系电话是客户的属性，所以首先可能想到的一种设计方案如下：

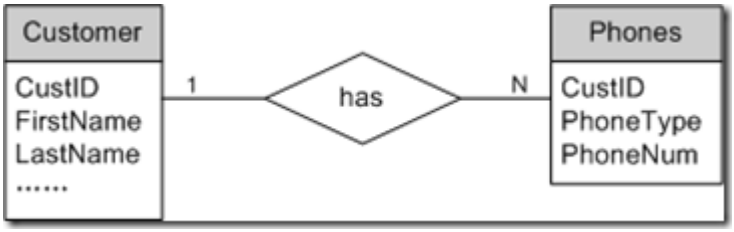


（图 1　联系电话作为客户的属性）

图 1 这一设计满足了当前的需求，但不久我们发现客户的联系电话比我们想象的多，他们还有移动电话，而且有些客户有不止一个办公电话号码，我们需要记录这些电话号码，并标识不同的办公室地点。

对于这种需求，我们可以在 Customer 表中增加列，但这样做会有两方面的问题：首先，每次增加新列都需要修改数据表结构，需要 DBA 从后台写脚 本完成，且前台显示联系电话的功能模块也需要相应进行修改。其次，每个客户具有的联系电话类型及数量各不相同，大量的联系电话单元格都是空的，浪费了许多 存储空间。

我们换一种设计方案。每个客户有多个不同类型的联系电话，可以把联系电话作为弱实体从原先 Customer 实体中分离出来，如图 2 所示：



（图 2　实体 Customer 与实体 Phones 一对多关系）

Customer 与 Phones 之间为“一对多”关系，即一个客户可以有多个不同类型的联系电话。当我们需要给某客户增加联系电话时，我们不再需要 修改表结构，只需要在 Phones 表中增加记录就可以了。这完全可以作为前台的功能让业务操作人员来完成，而且现在的 Customer 不再会存在大量空单 元格了。在关系数据库中增加行比增加列的代价要小很多。

实体 Phones 的主键是什么？

CustID 肯定是主键的一部分。主键包含的其他列根据我们想表达的不同语义，可以有所不同。

语义 1：一个客户不能有重复的联系类型。即一个客户的每个 PhoneType 不能重复，但多个不同的 PhoneType 可能对应相同的 PhoneNum（如：PhoneType 为“Office”和“Home”对应同一个号码）。符合该语义的主键为：CustID，PhoneType；

语义 2：一个客户不能有重复的联系电话。即一个客户的每个 PhoneNum 不能重复，但多个不同的 PhoneNum 可能对应相同的 PhoneType（如：PhoneType 为“Office”有多个不同号码）。符合该语义的主键为：CustID，PhoneNum；

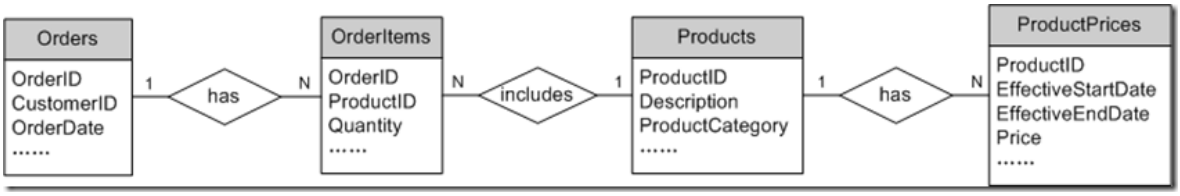
语义 3：一个客户的一个联系类型能有多个不同的联系电话，一个联系电话可能对应不同的联系类型。符合该语义的主键为：CustID，PhoneType，PhoneNum；

举一反三，该多值属性设计方法同样适用于维护客户的多个地址或 Email 等场景。

历史追溯

说到历史就会涉及时间。例如：当前物价持续上涨，同一产品的售价每个月都有可能调整，若要追溯产品价格变化的情况，仅仅记录该产品当前的一个售价是不足够的。同样对于银行中的利率变化，购入原材料的单价变化等，都需要进行历史追溯。

要跟踪一个实体随时间的变化可以在该实体中增加属性列，指明实体中每个实例的有效日期。图 3 展示了可追溯产品价格的订单结构（已经过简化）。



（图 3　简化的订单表结构）

实体 Orders 记录订单的公共信息，包括订单号（OrderID），下订单的时间（OrderDate），客户编号（CustomerID）等。 其中 OrderID 提供了到实体 OrderItems 的联接。实体 OrderItems 记录客户订购的产品条目，包括所属订单号（OrderID），产品 编号（ProductID），订购数量（Quantity）等。其中 ProductID 能联接到实体 Products。实体 Products 中包含每种产 品的描述信息。实体 ProductPrices 记录了产品的价格，包括产品编号（ProductID）对应到 Products 实体，产品价格 （Price），以及该价格的有效时间段（EffectiveStartDate，EffectiveEndDate）。

对于上述表结构，回溯历史某个订单的信息的步骤如下：

1. 根据订单号（OrderID）在 Orders 表中找到对应的记录，并记录下 OrderDate
2. 在 OrderItems 表中根据 OrderID 找到对应的所有订单明细记录。对每一条明细，记录下 Quantity 和 ProductID，之后：
   a. 通过 ProductID，在 Products 表中找到对应产品的产品描述（Description）
   b. 在 ProductPrices 表中找到对应 ProductID，且 EffectiveStartDate <= OrderDate < EffectiveEndDate 的记录。该记录中的 Price 为指定产品在历史下单时的价格。

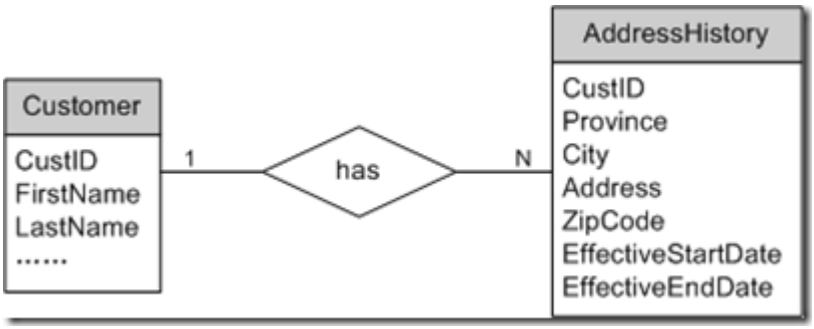
这样我们就得到了该订单的历史“快照”信息。

需要注意的几点：

1. 如果我们只需要追溯订单中产品的历史价格，可省去上述步骤中的 a。
2. 上述订单表结构在每次查看订单时都需要查询 ProductPrices 表。我们可以通过在 OrderItems 表中增加 ItemPrice 列，来避免对 ProductPrices 表的频繁查询。当创建订单明细记录时，把从 ProductPrices 中查询到的价格记录到 ItemPrice 列中，之后每次 查看订单时就不需要再查询 ProductPrices 表了。
3. ProductPrices 表的主键为 ProductID，EffectiveStartDate。同时该表还隐含着约束：同一种产品的价格有效时间段不能重叠。
4. ProductPrices 表结构中 EffectiveEndDate 列可省去，把该产品的下一个 EffectiveStartDate

作为上一个有效时间段的自然结束时间点。但这样做会增加查询的复杂度。

再举一个简单的例子，每个客户只有一个地址信息，但希望能跟踪客户地址的变更情况。我们能设计如下（图 4）表结构：

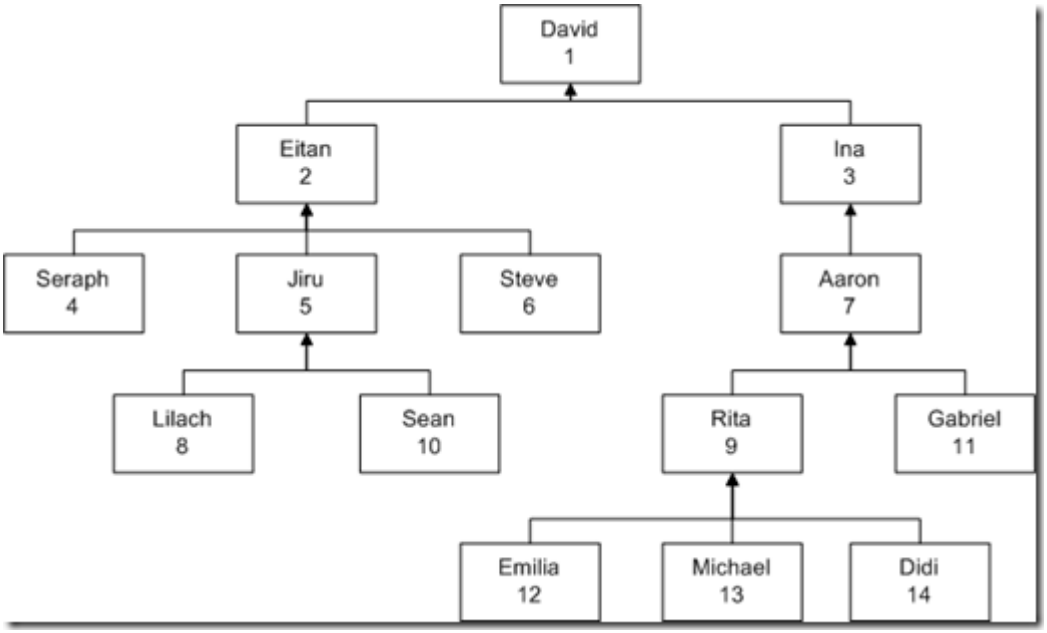


（图 4　跟踪客户地址的变更）

类似的场景包括：跟踪员工薪资的变化情况，跟踪汇率的变化情况等等。还有一种场景可使用该技术，当我们通过系统前台试图删除某信息时，系统的后台数 据库并不真正去做删除操作，而是通过 EffectiveEndDate 标识记录的无效时间。通过 EffectiveStartDate 和 EffectiveEndDate 可回溯任何历史时间点存在的记录“快照”。

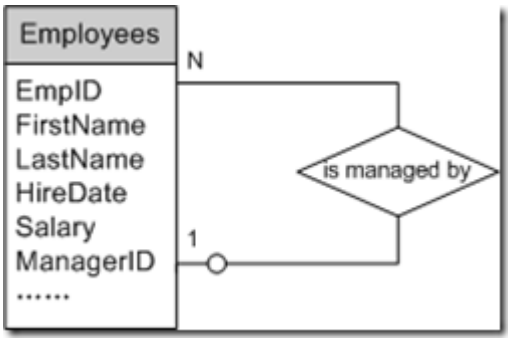
树型结构

树型结构最典型的例子是员工组织机构图，如图 5 所示。



（图 5　员工组织结构图）

树型结构中除根节点之外，每一个子节点都有一个父节点。可以把节点建模为一个实体，父子之间的联系建模为“一对多回归关系”。图 5 中的员工组织结构可建模为图 6 所示的 ER 结构。



（图 5　员工组织结构 ER 模型）

实体 Employees 中的 EmpID，FirstName，LastName，HireDate，Salary 等属性描述了员工的基本信息，树型层次关系通过 ManagerID 属性进行描述，该属性存储了该员工的经理 ID，即指向其父节点。

在节点实体中存储指向父节点的属性已足够描述树型结构的语义，但为了提高查询的效率，设计中可增加树型结构层次（Lvl）和物化路径（Path）作为辅助信息。图 5 员工组织结构样例数据如下：

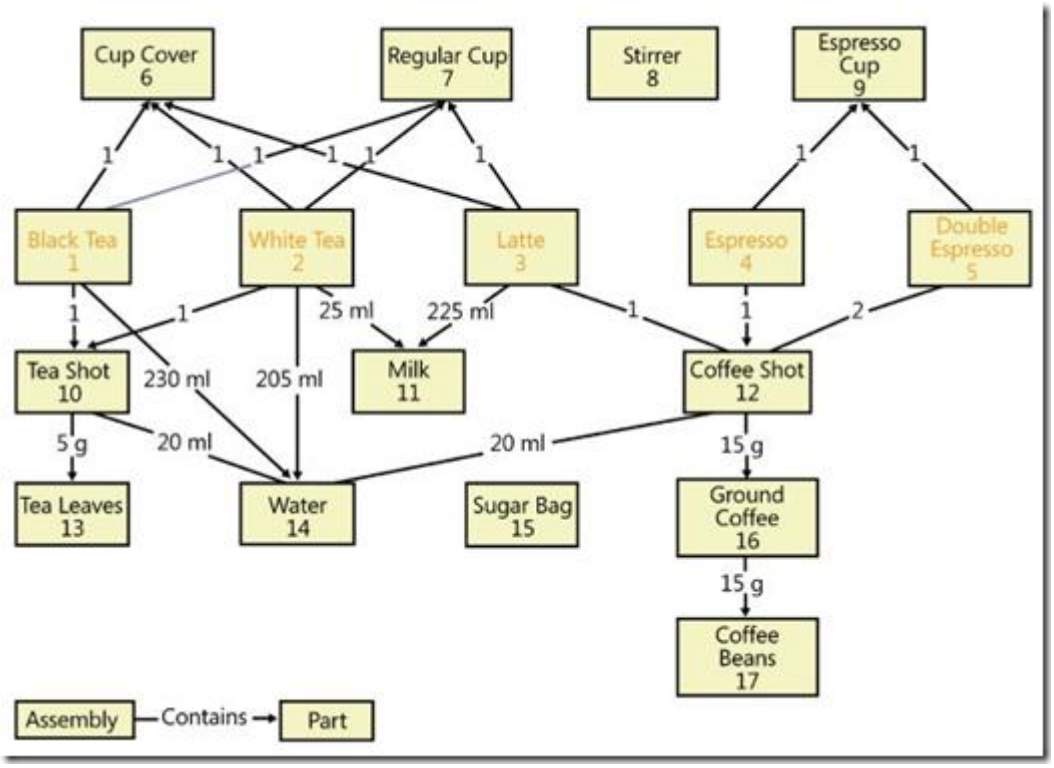
EmpID	FirstName	LastName　HireDate …	ManagerID	Lvl	Path
1	David	……	NULL	0	.1.
2	Eitan	……	1	1	.1.2.
4	Seraph	……	2	2	.1.2.4.
5	Jiru	……	2	2	.1.2.5.
10	Sean	……	5	3	.1.2.5.10.
8	Lilach	……	5	3	.1.2.5.8.
6	Steve	……	2	2	.1.2.6.
3	Ina	……	1	1	.1.3.
7	Aaron	……	3	2	.1.3.7.
11	Gabriel	……	7	3	.1.3.7.11.
9	Rita	……	7	3	.1.3.7.9.
12	Emilia	……	9	4	.1.3.7.9.12.
13	Michael	……	9	4	.1.3.7.9.13.
14	Didi	……	9	4	.1.3.7.9.14.

（表 1　员工组织结构数据，其中 Lvl 列，Path 列可选，利用该两列能提升某些查询的性能）

注：如何对树型结构数据表进行查询、遍历在这里不进行展开，可参考《Inside Microsoft SQL Server 2005 T-SQL Querying》一书。本例及以下两小节中的例子，引用自《Inside Microsoft SQL Server 2005 T-SQL Querying》，但同样适用于其他关系数据库。

有向无环图结构

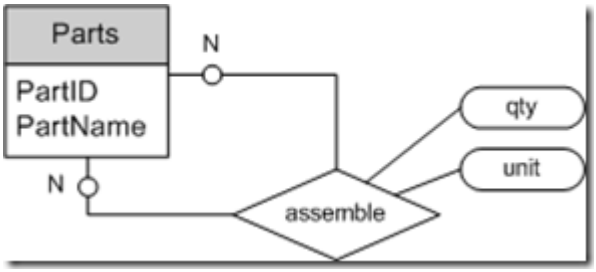
有向无环图（DAG）的典型应用场景是物料清单（BOM）。BOM 记录了产品的组装零件或配置方式，下图为某咖啡店的 BOM 图，描述了配置每种饮料的原料及剂量。



（图 6　咖啡店 BOM 图）

我们如何把这一 BOM 信息存储到数据库中呢？

BOM 场景以有向无环图为模型。有向无环图与树型层次结构的差异之处在于，有向无环图中的一个节点能有多个父节点。故 ER 模型中，有向无环图需建模 成两个实体，一个实体用于描述节点，另一个实体用于描述节点之间的边。咖啡店 BOM 场景的 ER 模型如图 7 所示，实体 Parts 表示咖啡店的原料及饮品，实 体 Assemble 表示原料配置的方向（即“有向边”），其中还包括边的权值，此例中边的权值为 qty，表示配料的剂量，unit 为配料的剂量单位（如：g，ml 等）。



（图 7　咖啡店 BOM 的 ER 模型）

把咖啡店 BOM 的 ER 图转化为 SQL：

```
create table Parts
(
    PartID int not null primary key,
```

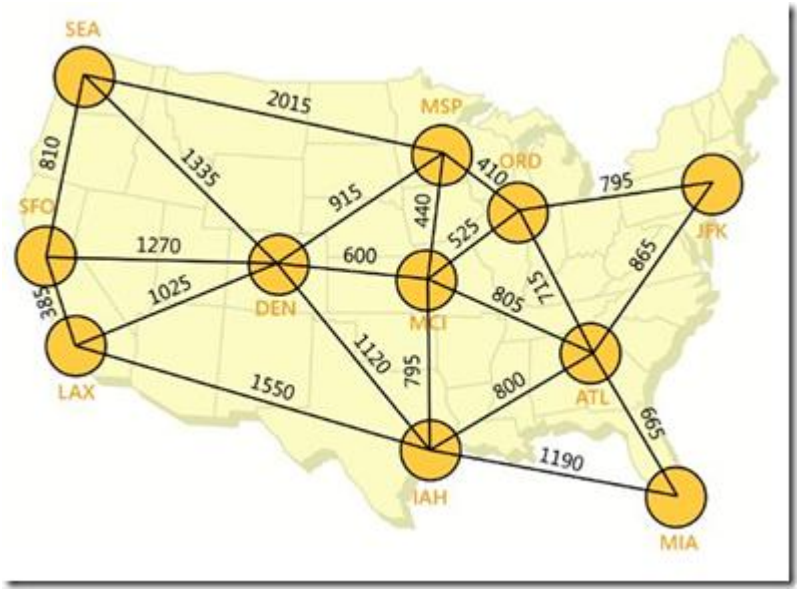
```
    PartName varchar(25) not null
);

create table Assemble
(
    PartID int not null references Parts,
    AssemblyID int not null references Parts,
    Unit varchar(3) not null,
    Qty decimal(8,2) not null,
    primary key(PartID, AssemblyID),
    check(PartID <> AssemblyID)
);
```

需要注意以下几点：

1. 上述代码在 SQL Server 2008 下测试通过。对于其他数据库产品，代码细节可能需稍作调整，但主体设计结构不变。
2. Assemble 表的主键为：PartID，AssemblyID。
3. Assemble 表的 PartID 列和 AssemblyID 列外键引用 Parts 表。
4. Assemble 表的 check 约束保证其中任何记录的 PartID 与 AssemblyID 的值不会相同。

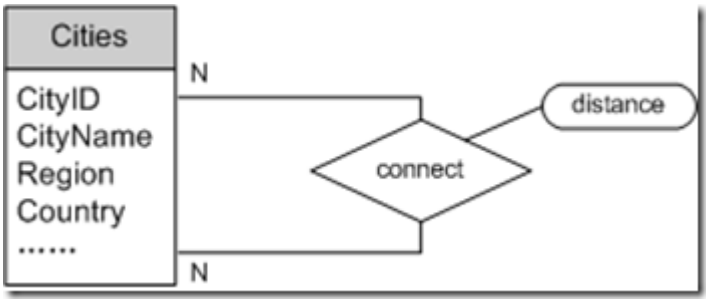
无向循环图结构

无向循环图的一个典型例子是城市道路系统。下图展示了美国主要城市之间的道路



（图 8　美国道路系统）

图 8 中每个节点表示一个城市，城市之间的连线代表城市之间的道路，连线上的数值表示距离。道路系统以无向循环图为模型，无向循环图中的节点能与任意数量的其他节点相连，且相连接的节点之间没有父子或先后关系（即“边”没有方向）。对图 8 中的道路系统进行 ER 建模得：



（图 9　道路系统 ER 模型）

把图 9 中的 ER 模型转化为 SQL：

```
create table Cities
(
   CityID char(3) not null primary key,
   CityName varchar(30) not null,
   Region varchar(30) not null,
   Country varchar(30) not null
);

create table Roads
(
   CityID char(3) not null references Cities,
   DestID char(3) not null references Cities,
   Distance int not null,
   primary key(CityID, DestID),
   check(CityID < DestID),
   check(Distance > 0)
);
```

需要注意以下几点：

1. 上述代码在 SQL Server 2008 下测试通过。对于其他数据库产品，代码细节可能需稍作调整，但主体设计结构不变。

2. 为了更易于理解，图 9 道路系统 ER 模型中的关系 connect，在转化为 SQL 表时更名为 Roads。Roads 表描述了一个无向循环赋权图。表中每一行表示一条边（道路）。Distance 属性表示权值（城市间的距离）。

3. Roads 表的 CityID 列和 DestID 列外键引用 CityID 表。

4. Roads 表的主键为 CityID，DestID。

5. Roads 表中包含 check 约束（CityID < DestID），以避免存入两个相同的边（eg：“芝加哥到纽约”和“纽约到芝加哥”）。无向循环图中节点之间是平等的，故该约束很重要，避免冗余数据。

6. 若要扩展到“有向循环图”场景（如：道路系统中的单行道），我们只要去除 check 约束（CityID < DestID），此时不同方向的数据不再是冗余。

到这里整个系列将告一段落，希望大家能觉得该系列言之有物，读了能有些许收获。最后，对本系列博文作一个回顾，同时给出一些参考资料。

本系列篇目回顾

1. 数据库设计 Step by Step (1)——扬帆启航
2. 数据库设计 Step by Step (2)——数据库生命周期
3. 数据库设计 Step by Step (3)——基本 ER 模型构件
4. 数据库设计 Step by Step (4)——高级 ER 模型构件
5. 数据库设计 Step by Step (5)——理解用户需求
6. 数据库设计 Step by Step (6) —— 提取业务规则
7. 数据库设计 Step by Step (7)——概念数据建模
8. 数据库设计 Step by Step (8)——视图集成
9. 数据库设计 Step by Step (9)——ER-to-SQL 转化
10. 数据库设计 Step by Step (10)——范式化
11. 数据库设计 Step by Step (11)——通用设计模式（系列完结篇）

参考资料

以下推荐的两本书虽然不是关于数据库设计，但对于程序开发人员会有帮助。熟练掌握数据库查询与编程能促进对数据库设计的理解与学习。

1. 《Inside Microsoft SQL Server 2005 T-SQL Querying》——这是我当初看的一本书，对于深入理解数据库查询很有帮助。现在已经出了《Inside Microsoft SQL Server 2008 T-SQL Querying》。
2. 《Inside Microsoft SQL Server 2005 T-SQL Programming》——对于学习 T-SQL 编程很有助益。