ETL 数据结构

数据仓库的后台部分经常被称为集结区（Staging Area）。在这里的上下文中，数据的集结过程指的是写入磁盘，并且我们建议在 ETL 数据流的四个主要检查点都要有数据集结。ETL 小组需要不同的数据结构来满足不同的数据集结需求，本章的目的就是描述可能需要的所有类型的数据结构。

本章不会描述所有必须抽取的源数据的类型，我们将在第 3 章中讨论这个话题。

流程检查

规划与设计：需求/现状 -> *架构* -> *实现* -> 测试/发布

数据流：抽取 -> 清洗 -> 规格化 -> 提交

我们还将尝试从数据结构的细节回头再看建议的集结区通用设计原则，包括规划和设计标准，并介绍支持集结所需要的元数据。元数据是一个大话题，我们将在第 4 章描述一些清洗和规格化步骤中特定的元数据。另外，从第 9 章直到本书结束我们还会集中地讨论元数据问题。

是否集结数据

是将数据存储在物理集结区还是在内存中直接处理？这个问题是 ETL 架构中最根本的选择之一。开发的 ETL 处理的效率很大程度上取决于能否很好地均衡物理 I/O 与内存处理。

能够在将数据写入集结表和保持在内存两种方法间取得理想的均衡是个很大的挑战，也是优化处理过程中必需考虑的问题。最终的决定取决于下面的两个彼此矛盾的目标：

- 将数据以最快的速度从数据源获取到最终目标
- 在处理过程发生错误时，能够进行恢复而无需从头开始

根据环境和业务需求的不同，数据集结的策略会有很大的不同。如果计划在内存中处理所有的 ETL 数据处理，不要忘记任何一种数据仓库，无论其架构和运行环境如何，都包含了一个某种形式的集结区。之所以要在加载到数据仓库之前集结数据，主要是基于如下的考虑：

- 可恢复 在大多数的企业环境中，数据从源系统中抽取出来后，会进行一系列的重要的转换，假设对于某张表，其转换的工作量很大，那么根据我们的最佳实践，应该在数据一抽取完马上就进行集结。这些集结表（在数据库或者文件系统）可以作为恢复点。一旦转换过程发生错误，利用这些表，处理过程就无需再次访问源系统。同样的道理，如果加载过程失败，也无须重新进行转换。如果集结数据纯是为了恢复的目的，那么数据应该存储在文件系统中的顺序文件中，而不是数据库。以恢复为目的的数据集结对于从业务系统中抽取数据尤其重要，因为业务系统中的数据会被覆盖和修改。
- 备份 通常，巨大的数据量使得在数据库级别上进行可靠的数据仓库备份变得不可行。只要加载文件已经进行了保存、压缩和归档，那么我们就可以避免数据库故障所带来的灾难。如果集结表存储在文件系统，那么就可以被压缩成非常小的文件并存储在网络上。当需要重新向数据仓库中加载数据时，仅需要对加载文件解压缩并重新加载。
- 审计 很多时候，源系统和目标系统之间的数据沿袭在 ETL 代码中丢失，当审计

ETL 流程时，数据集结区的存在使得对 ETL 流程中的不同阶段的直接比较成为可能，因为这时候审计人员（或者程序员）可以简单的比较原始的输入文件和输出文件来检查逻辑转换规则。当源系统覆盖了历史数据时，集结数据特别有用。当一个事件发生后，你可能对于数据仓库中几天甚至几周的数据信息的完整性产生了疑问，这时候对相应时段的集结区的抽取数据进行检查将能够帮助你恢复对数据仓库的数据准确性的信心。

一旦决定要集结哪怕只是一部分数据，你都必须要为集结区选择一种合适的架构。对于任何的数据库，如果数据集结区没有认真的规划，最终将会失败。对于数据集结区的严密设计远比通常的应用设计更为重要，因为数据集结区的数据量是不断累积的（有时候会比数据仓库还要大）。下一节将讨论设计集结区时要考虑的问题以及可能的选择。

设计集结区

集结区按照自己的方式，为最终的数据仓库展示区来存储数据。有时候，保存集结区数据是为了支持那些需要历史数据才能完成的功能，而其它时候，集结区数据会在每个处理流程完成后就被删除。为维护历史信息而使用的集结区通常称为持久集结区（persistent staging area）。而临时集结区中的数据则在每次加载过程后被删除。大多数的数据集结区都使用混合模式，即同时使用临时和持久的集结表。

请确保你已经仔细地思虑了在整个数据仓库操作过程中集结区所担当的角色。集结区绝对不仅仅是为了支持下一个作业所创建的临时文件。集结区文件可以在后续的过程发生了严重的问题的时候用于恢复工作流，还可以用于审计和对处理过的数据内容进行验证。

除了考虑集结区数据的存储方式，还需要在设计和部署集结区时考虑如下的基本规则。讨论下面规则均假定你是 ETL 小组的成员，如果不是，请跳过此节。为了数据仓库项目的成功，必须建立和实践下列规则：

- 数据集结区的所有者必须是 ETL 小组。数据集结区，以及其中的所有数据不应该对 ETL 小组外的用户开放。数据集结区不是为展示来设计的。这里没有支持查询的索引和聚合表。在数据集结区没有为数据访问和一致性提供保障。这些与数据访问相关的需求应该在展示区来处理。
- 无论任何原因，用户都不许进入数据集结区。在集结区中的数据必须被视为正在处理中（a construction site area）。允许非授权的个人进入集结区将导致一系列的问题。好奇的用户通常会错误地使用数据，降低数据仓库的完整性。
- 报表不能从集结区访问数据。数据集结区是一个工作站点，ETL 小组不需要通知用户就可以对表进行增、删、改操作。注意，这并不是说，程序员可以随意地对所有的表进行增加、删除和修改。而是说，集结区是一个工作区，不是展示区。这个区域是一个受控制的区域，在生产集结区对表的修改和数据仓库展示层的生存周期是一致的。但是，同更改生产数据仓库的表不同，对数据集结区的表的修改不需要通知用户，这样报表可能会用到这些变更的表而产生错误。因此，不允许任何的报表访问集结区域，尤其当集结表是临时的。
- 只有 ETL 流程才能够读写集结区。数据仓库每天都会需要一些常规外部数据源所不能提供的数据集，例如数据质量状态类型表。即使数据仓库所需要的数据不是来自于现有的任何外部数据库系统，这些数据仍然需要同其他数据一样进入数据集结

区。要注意，数据集结区不是交易环境，绝不允许手工输入任何数据。如果必须使用手工维护的数据表，那就需要在数据集结区外面开发一个应用程序，通过应用程序将结果数据提供给 ETL 小组，然后使用 ETL 流程将数据合并到集结区。

ETL 组拥有数据集结区的完全权限。这意味着 ETL 架构师可以在这个集结区内设计表，并根据需求和相应的 ETL 流程来决定数据表应该建在数据库中，还是使用文件系统。一旦集结区创建完成，ETL 架构师必须将整个数据存储的容量估算结果提供给数据库管理员（DBA）小组和系统管理员，管理员根据这个结果为数据库、文件系统和目录结构计算空间分配和参数设定。图 2.1 描述了一个简单的集结区规模估算表，这个样例主要针对 ETL 数据流最终阶段的提交表。针对源数据拷贝和清洗及规格化步骤之后的集结区，一个完整的数据规模跟踪系统拥有类似的数据规模估算表。

表名	更新策略	频率	ETL作业	行数	行长	增长情况	每月行数	每月字节数	初始大小	6月大小
S_ACCOUNT	Truncate / Reload	Daily	SAccount	39,933	27	New accounts	9,983	269,548	1,078,191	2.57
S_ASSETS	Insert / Delete	Daily	SAssets	771,600	78	New assets	192,875	15,044,250	60,177,000	143.47
S_BUDGET	Truncate / Reload	Monthly	SBudget	39,932	104	Refreshed monthly	9,983	1,038,232	4,152,928	9.90
S_COMPONENT	Truncate / Reload	On demand	SComponent	21	31	Components added to inventory	5	163	651	0.00
S_CUSTOMER	Truncate / Reload	Daily	SCustomer	38,103	142	New customers added daily	9,526	1,352,657	5,410,626	12.90
S_CUSTOMER_HISTORY	Truncate / Reload	Daily	SCustomerHistory	2,307,707	162	Refresh with each bulk load	576,927	93,462,134	373,848,534	891.32
S_CUSTOMER_TYPE	Truncate / Reload	On demand	SCustomerType	5	21	New customer types	1	26	105	0.00
S_DEFECT	Truncate / Reload	On demand	SDefect	84	27	New defect names	21	567	2,268	0.01
S_DEFECTS	Insert Only	Daily	SDefects	8,181,132	132	Transaction defects	2,045,283	269,977,356	1,079,909,424	2,574.70
S_DEPARTMENT	Truncate / Reload	On demand	SDepartment	45	36	Departments established	11	405	1,620	0.00
S_FACILITY	Truncate / Reload	Daily	SFacility	45,260	32	New or changed facilities worldwide	11,315	362,080	1,448,320	3.45
S_HISTORY_FAIL_REASON	Insert / Delete	On demand	SHistoryFailReason	6	27	New failure codes	2	41	162	0.00
S_OFFICE	Truncate / Reload	Daily	SOffice	14	56	New offices opened	4	196	784	0.00
S_PACKAGE_MATL	Truncate / Reload	Daily	SPackageMatl	54	18	New packaged material categories	14	243	972	0.00
S_PRODUCT	Truncate / Reload	Daily	SProduct	174,641	73	New products	43,660	3,187,180	12,748,720	30.40
S_PROVIDER	Truncate / Reload	On demand	SProvider	63	45	Service providers	16	709	2,835	0.01
S_REGION	Truncate / Reload	Daily	SRegion	333	37	New or changed global regions	83	3,080	12,321	0.03
S_RESPONSES	Insert Only	Daily	SResponses	5,199,095	105	Response transaction	1,299,774	136,476,244	545,904,975	1,301.54
S_SURVEY	Truncate / Reload	Daily	SSurvey	45,891	93	Survey conducted	11,473	952,239	3,808,953	9.08

图表 2.1　集结区规模估算表

规模估算表中列出了集结区中每张表的如下信息:

- 表名称。在集结区中的表或者文件的名字。在计算表中每个集结表都占用一行。
- 更新策略。这个字段表明表的维护方式。如果是一个永久表，可能的策略为追加数据、更新或删除。如果是临时集结表，在每个过程中都会被删减和重新加载。
- 加载频率。ETL 过程对表中的数据以什么样的频率加载和更改。通常是每日一次。还可以为每周、每月或者任意的时间间隔。在一个实时环境中，在集结区中的表可能不断的被更新。
- ETL 作业。集结区表通过 ETL 作业进行操作和更新。ETL 作业指的是操作集结区表或者文件的作业或者程序。当多个作业操作单个的表的时候，在估算表的这个字段中列出所有的作业。
- 初始行数。ETL 小组必须估计在集结区初始的时候每个表中的记录数。记录数通常和源表和目的表的行数有关。
- 平均行长度。为了估算空间大小，必须将每个集结区表的平均行长度提供给 DBA。在 Oracle 环境中，我们通常在开发环境中创建表，运行统计后从 ALL_TABLES 中收集这些信息。例如，在 Oracle 中 DBMS_STATS 包可以用于生成相应的统计字段信息。
- 增长。虽然每张表都是按照调度周期进行更新的，但不会每次都增长。在估算表中的增长字段(GrowWith)是基于业务规则。你必须定义集结区中的表何时会增长，例如一个状态表只有在增加新状态的时候才会增长，尽管这些表每天都会被访问来查

看是否有变化，但是增加新的状态的情况并不常发生。

- 预计每月行数。这个估计是根据历史和业务规则。DBA 需要根据预期的增长来给表分配空间。每月行数是计算每个月增长多少字节的重要因素。
- 预计每月字节数。预计每月字节数等于平均行长度乘以预计每月行数。
- 初始表大小。初始表大小通常用字节或者兆字节来表示。取值等于平均行长度乘以初始行数。
- 6 个月表大小。6 个月表大小的估算可以帮助 DBA 小组估计集结区数据库或文件系统的增长情况，通常用兆字节表示，计算公式为 （（平均行长度 * 初始行数）+（平均行长度 * 预计月行数 * 6））/1,048,576 。

到目前为止，我们的讨论还是架设在数据库管理系统（DBMS）创建集结表。而实际上，集结区通常会同时使用 DBMS 和文件系统的平文本文件。平文本文件在使用专门的 ETL 工具的时候非常重要。大多数的工具会使用文件系统的区域存储数据以优化其流程。大多数情况，需要使用 DBMS 之外的平文件来集结数据以获得高速的连续处理性能。这种情况下，需要使用文件系统规摸估算表。通常的方式是在开发区中放置文件，然后记录所占用的空间，作为正式空间分配时候的估计值。

本章下一节的内容将帮助你为集结表选择合适的结构。

ETL 架构需要在文件系统中分配和配置数据文件作为数据集结区的一部分，来支持 ETL 处理。使用文件系统的 ETL 工具的厂商会推荐合适的空间和系统配置选项来优化性能和扩展性。对于手工编码的 ETL 流程创建的数据文件的规模估算，可以使用标准的数据量估算表。

ETL 系统中的数据结构

在本节中，我们将介绍 ETL 系统中可能使用的几种重要数据结构类型。

平面文件

在很多情况下，没有必要在 DBMS 中对数据进行集结。如果不是使用专门的 ETL 工具而是在数据库中使用 SQL 完成所有的 ETL 任务，那么就需要创建 DBMS 表来存储所有的集结数据。但是，如果决定使用 ETL 工具，或者使用那些能够操作文本文件的 Shell 脚本或脚本程序，比如 Perl，VBScript 或者 JavaScript，那么，通常情况下，可以在文件系统中使用简单文件来存储集结数据。

当集结数据像数据库表那样按照行和列存储在文件系统中的时候，我们称之为平文件（Flat File）或者顺序文件（Sequential File）。如果操作系统是 UNIX 或者 Windows 时，那么数据将按照标准化的 ASCII 编码进行存储。ETL 工具或者脚本语言可以像操作数据库表一样方便地操作 ASCII 平文件，而且在某些情况下处理速度会更快！

DBMS 需要很高的代价来维护要处理的数据的元数据，这在一些简单的数据集结区环境下并不真正需要。而且根据经验，诸如排序、合并、删除、替换以及其他的一些数据迁移功能，在 DBMS 之外进行操作要快得多。有很多专门的程序工具可以用来处理文本文件。请记住，在使用脚本语言操作平文件的时候，你有义务为所作的转换提供元数据追踪表。如果元数据追踪（比如，为了合规的目的）和转换本身同等重要，你应该考虑通过 ETL 工具

来完成这些操作，因为工具能够自动的提供元数据的上下文。

如下是有利于关系表的观点。

对数据的截断或者插入操作，通常要比写平文件快一些。而且，对于平文件来说没有真正的高效的数据更新这个概念，而这通常是反对使用平文件作为永久数据集结区的理由。操作系统工具或者厂商提供的工具通常并不支持查询或者其他随机的查找操作。最后，平文件不能为快速查找建立索引。

当对 ETL 系统中的集结区表读数据的时候，如果需要进行筛选或者需要在不同的表之间进行联接的时候，使用数据库存储是更好的选择。尽管专门的排序包可以用来提高效率，但是关系型数据库最近已经在所专注领域取得了显著的进步。考虑到能够使用 SQL 并免费获得自动的数据库并行特性，数据库方案还是非常值得关注的。

使用何种平台来存储数据集结区表取决于很多因素，包括企业标准和实践。无论如何，我们已经发现，当基本的需求是下面所列之一的时候，在 ETL 过程中使用平文件将更加实际：

- 出于保护和恢复的目的而进行的源数据的集结。当数据从数据源中抽取的时候，必须快速地进入系统，选择所需要的数据，然后退出。如果数据抽取出来后，后续的 ETL 过程发生了错误，必须能够重新执行 ETL 过程而不需要再次访问源系统。要确保能够进行恢复而不需要频繁地访问源系统的最好的方法就是，将抽取出的数据在平文件中进行保存。数据进入平文件后，后续的处理发生任何的错误，都可以很容易的重新进行，这时候仅仅需要从集结区中的平文件中获取数据。我们在第 8 章中对如何使用平文件进行 ETL 处理错误恢复进行讨论。
- 数据排序。排序实际上是每个数据集成任务的先决条件。无论是做聚合、校验还是查找，数据的预先排序都可以提高性能。在文件系统中的数据排序有可能比在数据库中利用 ORDER BY 子句来获取数据效率更高。因为 ETL 的本质是要集成不同的数据，合并数据的效率是首要考虑的问题，而这需要大量的排序操作。ETL 系统的处理器周期中会有很大的部分在完成排序。如有可能，需要认真的在 DBMS 和专门的排序软件包中模拟最大的排序任务，根据结果来选择使用哪种方式来完成排序。
- 过滤。假设需要对源数据库中尚未没有建立索引的实体进行过滤，除了强制源系统创建索引——但这通常会妨碍事务的处理——还可以考虑将整个的数据集抽取到平文件，然后 grep（一个用于对简单数据文件进行筛选的 UNIX 命令）出那些符合要求的行。如果数据来自文件而非数据库，那么使用平那个文件进行筛选将更加可行。一个常见的例子就是 Web 日志文件。在处理点击流数据的时候，我们使用带 -v 选项的 grep 命令来获得那些不包含特定内容，如.gif,.jpg 的行，这样，那些对图片的点击将被排除在数据仓库访问范围之外。与平文件的筛选相比，将数据插入数据库表，建立索引，然后使用 WHERE 子句删除或者选取数据插入到另外的一张表中，效率要低得多。
- 替换/部分替换文本串。操作系统可以用 tr 命令以令人难以置信的速度操作文件，将任意的字符串转换为另外的字符串。在数据库中进行子串搜索和数据替换需要使用嵌套的 scalar 函数和 update 语句。这种操作在操作系统一级进行处理会比数据库操作高效得多。
- 聚合。在第 6 章讨论的聚合必须通过两种方式进行支持：在将数据加载到数据库之

前的 ETL 数据流中；或者，如果聚合数据只用通过数据库筛选来获得，就在数据库中完成。如果在数据库外完成聚合，往往需要专门的排序软件包。如果是在数据库中完成，将会大量地使用数据库的排序功能，虽然偶尔也会将大量的数据导出，使用排序软件包。

- 源数据的引用。在规范化的事务处理系统中，通常会有一个唯一的参照表来支持其他的表。例如，一个通用状态表可以支持订单的状态、发货状态和付款状态。除了在源数据中反复地查询同一张参照表之外，更为高效的一种方法是一次性地将参照表抽取到数据集结区。ETL 工具将从集结区中查找所参照的数据。大多数的 ETL 工具可以将参照表加载到内存中，并使其在 ETL 整个处理过程中有效。从内存中访问参照表速度极快。另外，使用集结区中的参照表可以使源系统中的很多表联接可以被省略，这样对源系统的查询将更加的简单和高效。

XML 数据集

需要指出的是，XML 数据集在 ETL 系统中通常不用于永久存储集结区数据，它们更适合作为 ETL 系统的输入和输出的标准格式。当然 XML 数据集将来很有可能成为 ETL 系统和数据仓库查询表的永久数据集结格式，但是成为通用标准之前，XML 的层系能力必须和关系型数据库实现深度的结合。

XML 是一种数据通信语言。表面看起来， XML 利用普通文本文档的形式来存储数据以及元数据，但是不包含格式信息。XML 使用很多与 HTML 类似的符号来描述，但是与 HTML 文档架构很不相同。与 XML 相反， HTML 中包含了数据和格式化信息，但是不包含元数据。

XML 和 HTML 之间的差异对于理解 XML 如何影响数据仓库尤为重要。XML 元数据中包含了在 XML 文档中唯一标识每一个条目的标签。例如，一个发票的 XML 描述如下：

```
<Customer Name=”Bob” Address=”123 Main Street” City=”Philadelohia” />
```

这里 Customer 是 XML 元素，在发票传输之前，就已经在发送和接收方完成定义。Customer 元素被定义为包含一系列的 XML 属性，包括 Name，Addressm，City 以及其他可能的内容。

XML 在描述层系结构方面具有很强的能力，例如描述使用嵌套重复字段的表单。这些层系结构并不直接映射为标准二维关系表的行和列。当数据仓库接收到 XML 数据集以后，需要一个复杂的抽取过程将数据转换到关系型数据仓库。现在已经有一些讨论，关于如何扩展关系型数据库，提供对 XML 层系结构的本地支持，但是这需要大量的基于关系数据库的 SQL 语法和语义的扩展，目前还没有具体的实现。

如果不考虑层系结构的复杂性，XML 是目前在不兼容系统中转移数据的最有效的中间层，因为 XML（以及附加的 XML Schemas）中提供了足够的信息，可以为关系型数据库生成完全的 CREATE TABLE 语句，并按照正确的列类型提供数据。XML 定义了数据共享的通用语言，这是它的优势。对大数据量传输而言，XML 的劣势在于 XML 文档结构的冗余。如果要传送几百万条结构类似和可预知的记录，那么需要寻找比 XML 更加有效的文件结构。

DTDs, XML Schemas 和 XSLT

使用 XML 需要双方通过交换特定的类型定义文档（DTD）来识别可能的标签。对 Customer 的 DTD 定义例子如下:

<!ELEMENT Customer (Name, Address, City?, State?, Postalcode?)>

<!ELEMENT Name (#PCDATA)>

再以类似的方式来定义 Address, City, State 和 Postalcode。

在 City，State 和 Postacode 后面的问号表示这个字段是可选的。#PCDATA 声明表示 Name 是一个不带格式的字符串。

注意在 DTD 中包含的信息比 CREATE Table 中包含的信息要少，通常，不包含字段长度。

到目前为止，DTD 为双方交换 XML 建立了元数据理解的基础。一些工业标准组织已经定义了专属领域的标准 DTD。但是作为数据仓库，我们无法从 DTD 中得到建立关系表的所有的信息。为了改进这个问题，W3C 标准组织定义了继 DTD 之后的新工业标准 XML Schemas。XML Schemas 包含了大量的面向数据库的信息，包括数据类型以及 XML 元素之间的关联关系，换句话说，描述了表是如何联接的。

使用统一的 XML Schemas，在收到 XML 内容后，通过另一个规范 XSLT（可扩展的风格样式语言转换）来展示内容。事实上，XSLT 是将 XML 文档转换成另外一个 XML 文档的通用机制，但最常见的用法是用于将 XML 转换成用于屏幕输出的 HTML。

关系表

集结区数据可以存储在关系型 DBMS 中。尤其是没有使用专门的 ETL 工具时，使用数据库表就最合适。使用数据库存储集结区表有下列的优点：

- 直观的元数据。使用平文件的主要缺陷是缺乏直观的元数据。使用关系表存储数据，DBMS 自动的维护技术元数据，业务元数据可以很容易的附加在数据库表上。列名称、数据类型和长度以及基数的信息可以从数据库系统继承。表和列的业务描述通常作为元素加在 DBMS 数据目录中。
- 关系能力。实体之间的强制的数据完整性或参照完整性可以在关系数据库环境中很容易的维护。如果接收来自非关系型系统中的数据，在转换为维度模型之前，将数据集结在一个规范模型中就很有必要。
- 开放的资料库。一旦数据进入 DBMS，那么数据可以很容易的通过任何支持 SQL 的工具来访问（如果赋予了相应的权限）。进行质量保证测试和审计时，确保对数据的访问至关重要。
- DBA 支持。在很多企业环境中，DBA 小组仅对 DBMS 中的数据负责。数据库外的数据，如文件系统通常不由 DBA 维护。如果集结区数据不在 DBMS 中，那么 ETL 小组必须承担空间分配、备份和恢复、归档以及安全等任务。
- SQL 接口。很多的时候我们需要写 SQL 来操作数据，按照正确的格式获得数据。大家都知道 SQL 是标准语言，易于书写且功能强大。在 IT 领域，恐怕 SQL 是被最广泛掌握的程序语言了。大多数的数据库系统都提供了强大的 SQL 函数，帮助用户节省手工编程的时间。例如，Oracle 提供 to_char()函数可以将任意类型的数据转换为不同格式的字符串。除了强制参照完整性检查，能够使用自带的 SQL 是在

数据库环境中存储集结区数据的主要原因。

独立的 DBMS 工作表

如果决定在 DBMS 中存储集结区数据，在设计数据集结区模型时可以有不同的架构选择。在集结区设计表可能比设计事务处理系统或设计维度模型更加具有挑战性。别忘了，事务处理数据库为数据进入进行设计，维度模型为数据输出进行设计，而集结区设计则同时包含两者。因此，在集结区使用混合的数据架构是很正常的。

至于采用独立集结表的原因，可以用一句话来概括：最简单的就是最好的。之所以被称为“独立表”，是因为这些表与数据库中的其他表没有任何依赖性。在事务处理环境，这些表被称作孤表，因为它和模型中的其他表没有任何关系。因为独立集结表没有任何关联关系，因此独立表是在关系型数据库外建立存储的主要候选方式。

大多数时候，创建集结表目的是为了存储数据以便于能够使用 SQL 或脚本语言再次操作数据。很多情况下，尤其是小型数据仓库项目，独立表可以满足全部的数据集结区要求。

由于独立表不需要规范化，因此不能视作转储文件。转储文件通常任意的创建，无须考虑磁盘空间或者查询效率。独立文件或独立表的每个字段必须有主题和逻辑定义。多余的列会在任何独立表的设计中被忽略。对于数据库表，需要在所有的独立表上建立并实现一个合理的索引规划。由于访问独立表的进程只有 ETL 过程，这里没有必要建立在展示区中才用到的位图索引，位图索引是为最终用户工具和即席查询所用的。在 ETL 系统中，会更多地在单列或复合列使用 B 树索引。

三范式实体/关系模型

有一种观点认为，数据集结区或许应该是最终加载到数据仓库中的企业所有数据的中央资料库。然而，将数据集结区称为企业中央资料库是明显的用词不当，这会导致数据架构师认为这个区域必须完全的规范化。毕竟，哪一个称职的数据架构师会为企业数据留下为人诟病的冗余呢？在本章开始我们已经做了一个餐馆厨房的比喻，例如，考虑某种食物——比如说鱼——在能够上桌之前需要仔细的挑选、洗净、分割、切片并煎炒。现在假设餐馆还提供冰激凌，那么肯定不会像之前做鱼那样做冰激凌，冰激凌只需要挖出来就可以上桌。强制让所有的源系统使用相同的规范化过程——使之符合第三范式数据模型——就像要你按照作鱼的方法准备冰激凌一样得不合情理。

事实上，我们很少将数据集结区建成第三范式模型。在很多案例中，一个层系中的数据元素来自多个不同的数据源，具有不同的粒度，还包括很多来自非关系型数据源的外部数据。这种情况下，通常是在加载到维度数据模型之前，消除数据冗余和强制完整性。理想的处理方式是集中处理独立的“问题维表”，对它进行彻底检查以确保原先的脏数据已经被正确地清洗。记住规范化的主要结果是强制明确的多对一关系，除非需要人工检查模型的图形描述，否则，实体关系图的注释既不强求也无须解释。建模的时候应该具体情况具体考虑，只有在必要的时候才进行实体规范化。

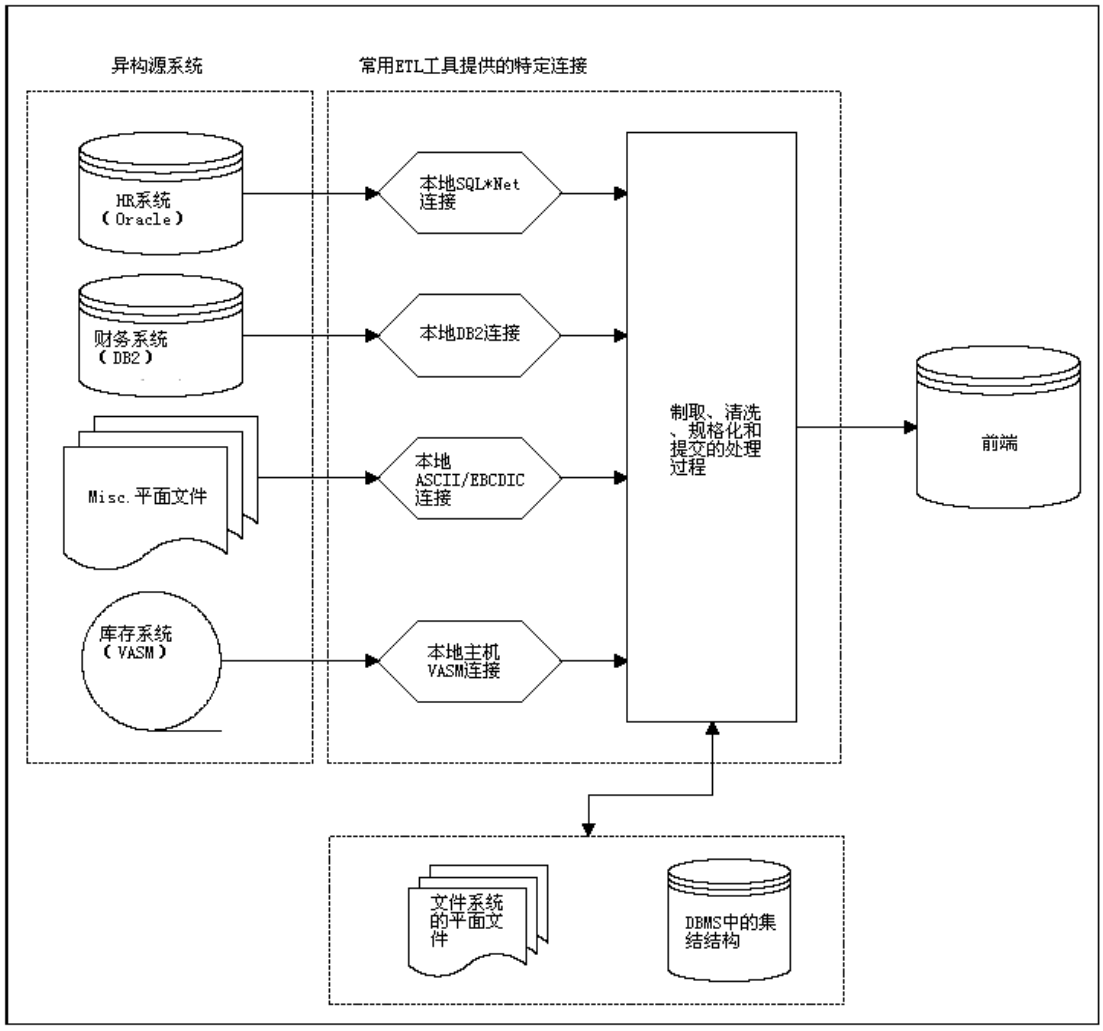
不要假定数据集结区必须作规范化。记住本章开始所讲的设计 ETL 处理的两个目标：快速和可恢复。如果 ETL 过程中所作的集结既没有物理的数据操作，也不能加快速度或支持恢复，那么它就应该被移除。

非关系数据源

通常，建立专门的集结区环境的一个重要的原因是为了集成非关系型数据。如果所有的数据都在一个 RDBMS 中，那么数据集成任务就没有什么挑战性了。集成异构数据源是 ETL 开发者经常要面对的挑战，尤其是在数据仓库扩展其数据范围，包含进越来越多的主题域的时候。

在企业数据仓库项目中，很多数据源来自非关系型数据源或者没有关联关系的多个关系型数据源。非关系型数据源可能包括 COBOL Copy Book，VSAM 文件，平文件，电子数据表等等。

通常的经验会说，将所有的异构的数据源集成到一个 DBMS 中，但是这真的必要吗？ETL 工具在处理异构数据方面的强大能力使得将所有必须的数据存储在一个数据库中的需求变得不再那么重要。图 2.2 描述了一个平台无关的 ETL 工具如何将多个异构数据源从它们的源系统进行数据集成，然后迁移到数据仓库。注意 ETL 工具连接到物理的集结区数据库和外部文件，在需要的时候可以为中间的数据处理设置临时的数据落地。



图表 2.2　ETL 工具可以随意的使用集结区来集成异构的数据源

集成非关系数据源通常需要作一些完整性检查。数据完整性的保证不是没有代价的，通常需要在数据集结区中的实际的存储，并且需要对 ETL 过程进行客户化以保证业务规则的正确，而这些业务规则在源系统的关系型数据库中是自动维护的。关系意味着表之间有联系，

通常为父子关系，这是由数据库来保证的。例如，如果一个订单表中有一个状态列，具体的值如果在一个独立的状态参照表中不存在就不能够录入。在这个场景中，状态表是订单表中对应列的父。状态表中的父不能随便地删除，除非所有的子已被删除，否则它的子就变成了孤儿记录。孤儿记录指的是任何没有父的子记录。同样，如果状态表中有外键引用到订单，那么任何的订单的主键也不能删除。孤儿记录的出现是参照完整性被破坏的标志。

非关系数据源不能保证参照完整性，非关系系统本质上是无关的表的集合。在很多遗留的事务系统中，父子关系只能通过前置应用来保证。不幸的是，经过多年运行，由于不可避免的脚本操作或者应用之外的数据操作，导致数据库系统中的任何数据完整性都已不能保证。可以肯定非关系数据源中多多少少都会有数据质量问题。

与事务系统不同，设计数据集结区的最佳实践是在 ETL 过程而不是在数据库中进行完整性检查。其中的区别在于，事务系统期望数据输入是正确的，并且人为的输入错误将导致错误抛出，提示重新输入正确的值。与此相反，ETL 过程则必须知道如何自动地处理数据异常。ETL 过程不能简单地拒绝所有的数据完整性错误，因为现在没有人会及时地重新输入正确的数据。我们需要为不同的数据质量错误场景定义不同的业务规则，并在 ETL 过程中实现这些规则。当错误的数据通过 ETL 流程时，有时你希望直接转换；有时不作修改直接加载；有时需要追加相关的代码或者追加一些对于影响及环境的描述，然后进行加载；或者如果数据不可接受，那就完全拒绝数据，并将其写入拒绝文件供进一步的调查。

不要过度使用拒绝文件！拒绝文件用于那些我们需要后续专门处理的转储数据。当记录进入拒绝文件后，除非在下一个主要加载步骤开始运行前完成对其处理，否则数据仓库和生产系统之间的同步就被破坏了。

基本的数据库参照完整性并不能完全处理上述的每一种情景。ETL 过程通常都需要手工编码逻辑，来保证成功集成非关系数据源。

维度数据模型：从后台提交到前台的成果

维度数据结构是 ETL 过程的最终目标，这些表位于前台与后台之间。多数情况下，维度模型是把表传递给最终用户环境之前的最后一步物理集结步骤。

维度数据模型是目前为止大多数最终用户查询和分析的最流行的数据结构。它们创建简单，结构异常稳定，不随数据环境的变化而变化。它们能被最终用户直观地了解，是满足通常的关系型查询的最快的数据结构。维度模型也是构建所有形式的 OLAP 立方体的基础，因为 OLAP 立方体实际上是由专门的软件创建的维度模型。

本节仅仅简单的介绍维度模型中的主要的表类型。

工具箱系列中的另外一本书将详细地讨论维度模型，给出在不同业务环境中建模的指南和目标。我们假设在这本书中，你已经完全明白维度模型的重要性（甚至已经掌握了某本维度设计的技能！）。因此在这一节中，我们就直接介绍基础的维度模型的物理结构，而不再从业务角度阐述采用这种结构的理由。在本书的第 2 部分，我们将详尽地介绍所有已知的维度模型的变种， 并讨论 ETL 系统如何为这些结构提供数据。

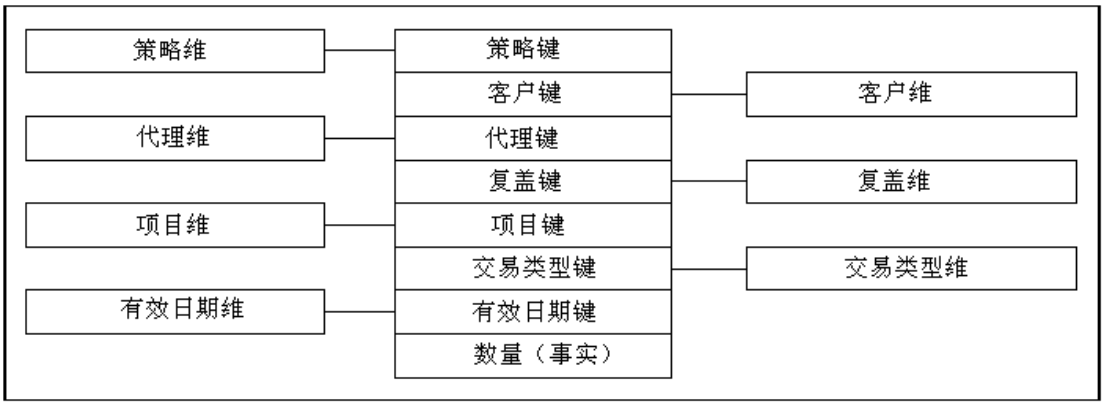
事实表

维度模型是围绕着度量过程建立的。一个度量指的是对先前未知值的观察。度量基本都

是数值型的，而且大部分度量都可以随着时间重复，这构成了时间序列。

一次度量构建一条单独的事实表记录。相对的，一条事实表的记录对应一次特定的度量事件。很明显，观察到的度量将存储在事实表记录中，但我们同时还要将度量的上下文信息存储在同一条记录中。当然，尽管我们可以将这些上下文信息直接存储在事实表记录中，但更多的时候是通过创建一系列的维表将这些上下文的属性规范化的存储在到事实表之外，也就是构建一系列的维表，这组维表可以看作是“上下文簇”。

例如，如果某个度量是某次保费登记的数额，具体的相关信息包括某个保险公司登记的某个保单、某个客户、某个代理、某个险种（比如碰撞伤害）、某个保险项目（比如汽车）、某个交易类型（比如确定保费）、在某个生效日期，那么与事实记录相关的维度就包括保单、客户、代理、险种、项目、交易类型和生效日期等等。图 2.3 描述了这个例子。



图表 2.3　保单交易的维度模型

事实表的粒度定义了构成事件表的记录的级别。在维度模型的规范中，粒度是在业务术语设计阶段而不是数据库术语设计阶段进行定义的。在上述保险的例子中，粒度应该是保险保单交易，在后续的设计过程中，当可用的维度确定之后，粒度通常以事件表的键的方式来描述。键的描述中将包含一些但通常不是全部的与事实表关联的维度的外键引用。我们假设保险事实表的键是“保单 X 交易类型 X 时间”。

维度模型的结构和内容仅仅取决于物理的度量过程本身。

维表

应该讲，维度模型的设计并不是只针对或者依赖某类特定的查询应用，事实上，它的设计目标是提供灵活的、适用于各种查询类型的、对称的框架。但是维度模型的设计还部分地依赖于设计人员的判断力。很多维度的设计可以直接参照原始的数据源结构，但是数据仓库小组还是能够添加来自其它数据源的维度，只要在度量事件发生的时候这些维度都是单值。比如，在上述保险的例子中，如果数据具备，那么就可以建立一个针对市场促销的维度。维度模型一个最大的优点是，在可以在度量事件的上下文支持某个维度的情况下灵活地增加维度。

类似的，最佳的维表设计应该包含对于维度实体的尽量详细的描述。所以在上述保险模型中，客户维度应该有很多描述性的字段，我们称这些描述字段为维度属性。幸运的是，所有的维度设计都允许在整个数据仓库的生命周期内增量的加入维度属性。数据仓库架构师的职责是识别出这些需要增加的属性，并要求 ETL 小组将它们添加到物理模型中。

维度属性通常是文本型或者离散的数值型。维表应该只有一个主键，键值使用的是由 ETL 过程生成的无实际意义的整型值，这些值称作代理键。使用代理键的优点在工具箱系

列的另外一本书中有介绍。在本书中，我们只是介绍如何创建代理键以及在很多重要情况下对代理键的管理。

每个维表中的主代理键应该与事实表中的相应的外键相匹配，当主-外键关系建立后，我们说表符合参照完整性。满足参照完整性是所有维度模型的基本要求。参照完整性管理失败，意味着一些事件表中的记录是孤儿记录，它们无法通过维度约束来访问。

原子事实表和聚合事实表

我们知道维度模型是支持用户查询的最佳的数据格式。有时我们会用到原子级别的某些元素，以实现数据在更高的级别的展示。无论如何，我们需要存储原子级数据以满足用户的某些特定约束的需求。通常情况下，业务人员不希望分析交易级的数据，因为每个维度的基数都是那样大，以致任何原子级的报表都会很多页，很难人工检查。但是，依然需要存储原子级的事实来产生用户需要的周期性快照。当用户需要原子级数据的时候，只是简单地将数据从集结区迁移到展示区就可以了。

在实践中，将集结区中的事实表进行分区是一个比较好的方法，因为原子数据的聚合通常是按照特定的时间周期进行的，比如月或者季。创建分区表减少了数据库的全表扫描，可以直接基于包含某个时间周期的数据分区创建聚合。分区同样减少了清理或归档旧数据的负担，使用表分区可以很简单的删除表中包含旧数据的部分。

集结区中的按照维度模型设计的表很多时候需要用来生成 OLAP 立方体。或者，也可以实现一种混合结构，在维度型 RDBMS 模型中存储数据量很大的原子数据层，原子层之上的日益增多的聚合结构按照 OLAP 立方体的方式进行存储。有些 OLAP 系统提供了钻透的功能，在单一应用中可以从 OLAP 立方体下钻到最低级别的原子数据。

代理键映射表

代理键映射表示用来建立各个源系统的自然键到主数据仓库代理键之间的映射，映射表是维护数据仓库代理键的一种非常有效的方法。这些表结构紧凑，专门用于为高速处理。映射表中仅仅包含那些最近期要访问的代理键值及其对应的源系统中的自然键值。由于同一个维度可以有不同的源，因此映射表中要为每个源的自然键创建单独的列。

映射表无论是存储在数据库还是文件系统中都可以有同样的高效率。如果使用数据库，可以利用数据库顺序号生成器来创建代理键，如果索引使用恰当，键值的查找会非常得高效。

由于键映射表并没有分析价值，因此，不能将其建在数据仓库的展示层，也不能暴露给最终用户。

交叉参考：在第 5 章中会详细解释如何应用映射表来为维表创建代表键，并且在第 6 章中讨论应用映射表来生成事实表。

规划和设计标准

流程检查

规划与设计：需求/现状 -> *架构* -> 实现 -> 测试/发布
数据流：抽取 -> 清洗 -> 规格化 -> 提交

同企业环境中其它的数据库相比，数据集结区需要进行更多的管理和维护。有些集结区的管理像沙盒一样，开发人员可以随心所欲地创建、删除和修改表。自然，这种缺乏管理的环境下的故障诊断和影响分析要比正常的花更多的时间，导致项目费用的增加。

数据集结区必须是一个严格控制的环境。只有数据架构师才可以在集结区中设计或修改表。所有的物理改变都应该是数据库 DBA 来完成。而且，如果有一个开发人员需要某张表，那么很大的可能是另一个开发人员也会使用它。一个忠告是：如果你创建了一张表，一定要想到，由于某种原因这张表很可能会被其他人以另外一种方式使用。人们，尤其是开发人员，在重新使用已有资源时经常会非常的有创意。

影响分析

影响分析的作用在于检查与对象（这里指的是表或者列）关联的元数据，并判断对象的变化对其内容和结构有何影响。对数据集结区对象的更改可能会破坏数据仓库的加载流程。允许任意修改数据集结区对象会对项目的成功造成危害。

一旦在数据集结区中创建了表，在做任何修改之前，都必须进行影响分析。很多 ETL 工具厂商提供影响分析功能，但是这些功能经常仅仅在 ETL 产品的 POC 阶段进行简单的评估，原因是它是个后台功能，直到数据仓库建立运行或者发生变更的时候才真正的变得重要起来。

影响分析，作为是 ETL 的一项功能，是一个非常繁重的职责，因为源系统和目标数据仓库可能不断地在改变，但是只有 ETL 过程知道这些分散的元素是怎么连接的。在数据仓库依赖的任何系统发生变化的时候都需要进行相应的影响分析，能否顺利地完成影响分析，ETL 项目经理、源系统 DBA 以及数据仓库建模小组之间的沟通至关重要。

元数据捕获

我们将在第 9 章对元数据进行深入讨论，但是事先必须了解在设计数据集结区的时候将会处理哪些类型的元数据。根据上下文的不同，元数据有不同的含义。元数据用于描述或支持其他的数据元素，涉及到组成数据仓库的每个组件。在数据仓库的范畴里，数据集结区不是元数据资料库。然而，和数据集结区相关联的很多元数据元素却对数据仓库非常有价值，必须展示给最终用户。

设计集结数据库的时候使用数据建模工具可以可视化地展示元数据，数据建模工具在它自己的资料库中存储这些可用的元数据。另外使用 ETL 工具会生成关于过程的元数据，并能够对其中的所有的转换过程进行展示。从集结区衍生出来的元数据类型包括：

- 数据谱系：所有数据仓库元数据库中最有趣的元数据可能就是数据谱系，或者称为逻辑数据映射，阐述了数据元素从原始数据源到最终数据仓库目标之间是如何转换的。
- 业务定义：数据集结区中创建的所有表都是从业务定义中衍生出来的。业务定义可以从很多地方获得，包括数据建模工具、ETL 工具、数据库自身或者电子表格和 Word 文档。无论如何，需要使用在数据仓库展示层上获取业务定义来维持其一致性。
- 技术定义：尤其对于数据集结区，技术定义要比业务定义更加的普遍。要记住，如果没有文档记录，那么就意味着技术定义不存在！如果数据集结区中表的技术定义没有详细的文档，那么这张表将可能被一次次的重建，会在数据集结区中产生大量

的数据重复，导致数据爆炸。技术定义应该描述数据元素的所有物理属性，包括结构、格式和位置。对集结区中所有表进行技术元数据文档化记录可以将不确定性降至最低，并提高重用性。

- 过程元数据：数据集结区表的加载过程的统计必须和数据仓库表加载的统计一起记录。尽管数据集结区加载过程的信息不需要展示给最终用户，但是 ETL 小组需要知道每个表中加载了多少记录，每个过程成功或失败的统计结果。而数据刷新频度方面的信息对 ETL 管理员和最终用户都是有用的。

数据集结区中所有的表和文件都应该由 ETL 架构师来设计。元数据必须很好地记录成文档。数据建模工具提供的元数据捕获功能将会减少文档的工作量。在设计集结区表的时候应该使用数据建模工具来捕获像相应的元数据，要记住从工具中获得的面向结构的元数据大约会占到 25%，另外 25%的元数据描述数据清理，而过程元数据将占 50%，请确认你选用的 ETL 工具能够提供所需要的统计信息。至少要提供每个过程的插入行数、更新行数、删除和拒绝行数等信息。另外，过程的开始时间、结束时间、持续时间应该不需要编码就可以得到。

命名规则

ETL 小组不应该替数据仓库小组开发命名规则，相反，他们应该使用数据仓库架构师所定义的命名规则，最佳实践的建议是集结区使用与数据仓库其他部分相同的命名规则。但是，因为数据集结区中有很多数据元素是不出现在展示层中的，因此很有可能没有针对它们的命名标准。这个时候，需要与数据仓库小组和 DBA 组一同工作，在命名规则中增加针对数据集结区特有的命名规则。

很多 ETL 工具和数据建模工具坚持展示很长的字母顺序的表名列表。你需要非常仔细地对你的表名进行分组，使其在字母排序时在一起。

审计数据转换步骤

在复杂的 ETL 系统中，数据转换体现了复杂的业务规则。如果需要对系统中的全部数据转换过程进行全程审计，至少要包含下列述列表中的任务：

- 用代理键代替自然键
- 对实体进行组合和剔重
- 对维度的常用实体进行规格化
- 对计算进行标准化，创建规格化的 KPI
- 在数据清洗过程中对数据进行更正和强制转换

在一个多变的环境中，源系统的数据不断的在改变，数据仓库要求要有能力证明数据的准确性。这种情况下如何处理。ETL 过程必须维护在数据清理阶段之前的数据快照。

当数据被 ETL 过程修改（清洗）过，在操作前的数据为了审计的需要必须保留。此外，所有数据清洗逻辑的元数据必须不通过代码就可以访问。原始的源数据、数据清洗元数据和最终的维度数据必须在一同发送来支持对数据清洗转换的置疑。

在数据集结区存储抽取数据的快照可以满足审计的需要。处理前后的数据快照，数据清洗逻辑的元数据共同描述了在数据仓库中数据如何的演变，提供了数据质量的可信度。

总结

在本章中，我们回顾了在 ETL 系统中需要使用的主要的数据结构。我们开始讨论了为了临时存储和永久存储而在不同位置集结数据的例子。你需要兼顾技术和投资的考虑。在这些案例中，你不仅仅需要存储数据，还需要记录数据创建的规则；如果这些信息不全，你需要证明数据没有被篡改。

一个成熟的 ETL 环境会混合文本文件、独立关系表、full-blown 规格化模型、以及其它可能的文件结构，特别是 XML 文档。但是无论何种情况，我们需要你记住 ETL 系统的核心是最终的结果：数据结构的建立目标是最终用户易于使用。当然我们建议这些表使用维度模型，使用事件表和维表。OLAP 立方体通常是通过维度模型创建的。

最后我们讨论了一些最佳实践，包括使用一致的设计标准，在你对表进行修改的时候，对你设计的表执行系统影响分析，另外，确保你在 ETL 系统的每个点上捕获了元数据。一个大的元数据目录中包含了 25%的表结构元数据，25%的数据清洗结果以及 50%的过程结果。元数据将在第 4 章和第 9 章进一步讨论。

现在我们已经组织了我们所有的工作数据的结构，在第三章中我们将深入探索组成数据仓库的源数据结构。