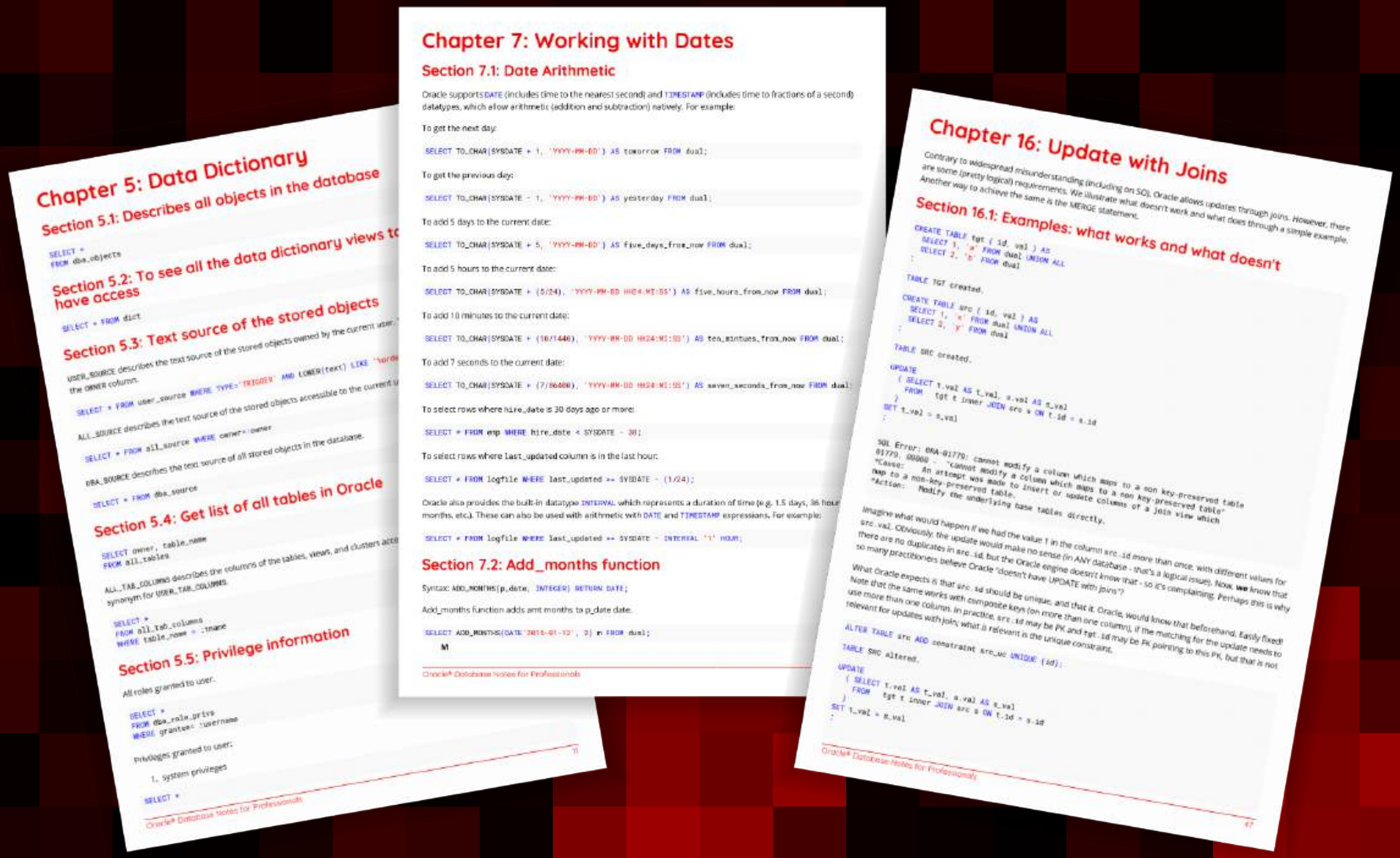
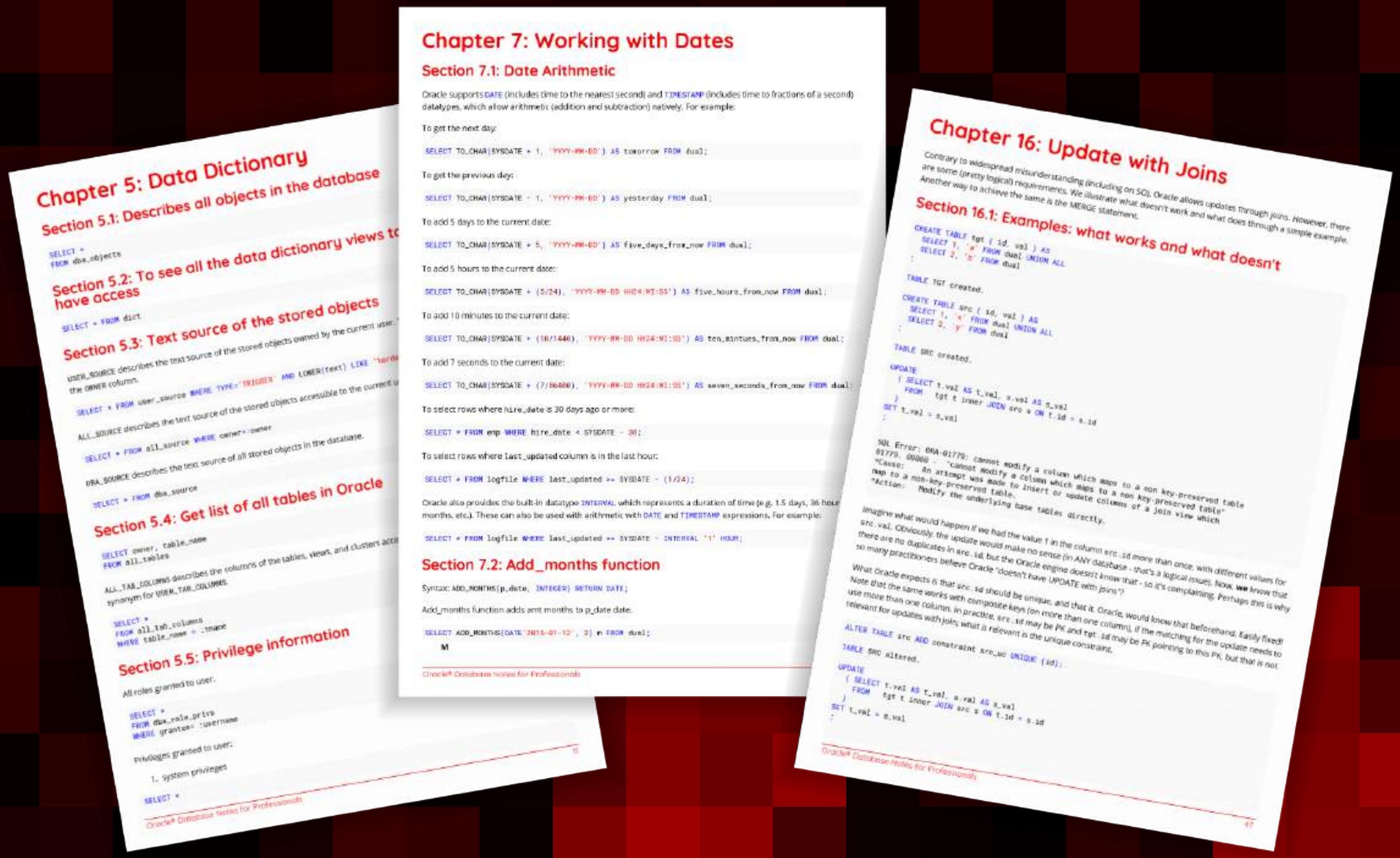
甲骨文数据库
专业人士笔记

甲骨文®
数据库
专业人士笔记


Chapter 5: Data Dictionary
Section 5.1: Describes all objects in the database
Section 5.2: To see all the data dictionary views to which you have access
Section 5.3: Text source of the stored objects
Section 5.4: Get list of all tables in Oracle
Section 5.5: Privilege information
Chapter 7: Working with Dates
Section 7.1: Date Arithmetic
Section 7.2: Add_months function
Chapter 16: Update with Joins
Section 16.1: Examples: what works and what doesn't

100+ 页
专业提示和技巧

Oracle®
Database
Notes for Professionals


Chapter 5: Data Dictionary
Section 5.1: Describes all objects in the database
Section 5.2: To see all the data dictionary views to which you have access
Section 5.3: Text source of the stored objects
Section 5.4: Get list of all tables in Oracle
Section 5.5: Privilege information
Chapter 7: Working with Dates
Section 7.1: Date Arithmetic
Section 7.2: Add_months function
Chapter 16: Update with Joins
Section 16.1: Examples: what works and what doesn't

100+ pages
of professional hints and tricks

目录

Contents

关于

About

第1章：Oracle数据库入门

版本	发布日期
版本1（未发布）	1978-01-01
Oracle V2	1979-01-01
Oracle版本3	1983-01-01
Oracle 版本 4	1984-01-01
Oracle 版本 5	1985-01-01
Oracle 版本 6	1988-01-01
Oracle7	1992-01-01
Oracle8	1997-07-01
Oracle8i	1999-02-01
Oracle9i	2001-06-01
Oracle 10g	2003-01-01
Oracle 11g	2007-01-01
Oracle 12c	2013-01-01

第1.1节：你好，世界

```
SELECT 'Hello world!' FROM dual;
```

在 Oracle 版本的 SQL 中，“dual 只是一个方便的表”。它最初是为了通过 JOIN 来复制行，但现在包含一行，DUMMY 值为 'X'。

第 1.2 节：SQL 查询

列出本世纪出生且收入超过 50000 美元的员工。列出他们的姓名、出生日期和薪水，按姓名字母顺序排序。

```
SELECT employee_name, date_of_birth, salary
FROM   employees
WHERE  salary > 50000
   AND date_of_birth >= DATE '2000-01-01'
ORDER BY employee_name;
```

显示每个部门中至少有 5 名员工的员工数量。按员工数量从多到少列出最大的部门。

```
SELECT department_id, COUNT(*)
FROM   employees
GROUP BY department_id
HAVING COUNT(*) >= 5
ORDER BY COUNT(*) DESC;
```

第 1.3 节：来自表的 Hello world！

创建一个简单的表格

```
创建表 MY_table (
   what VARCHAR2(10),
   who VARCHAR2(10),
```

Chapter 1: Getting started with Oracle Database

Version	Release Date
Version 1 (unreleased)	1978-01-01
Oracle V2	1979-01-01
Oracle Version 3	1983-01-01
Oracle Version 4	1984-01-01
Oracle Version 5	1985-01-01
Oracle Version 6	1988-01-01
Oracle7	1992-01-01
Oracle8	1997-07-01
Oracle8i	1999-02-01
Oracle9i	2001-06-01
Oracle 10g	2003-01-01
Oracle 11g	2007-01-01
Oracle 12c	2013-01-01

Section 1.1: Hello World

```
SELECT 'Hello world!' FROM dual;
```

In Oracle's flavor of SQL, "dual is just a convienence table". It was originally intended to double rows via a JOIN, but now contains one row with a DUMMY value of 'X'.

Section 1.2: SQL Query

List employees earning more than $50000 born this century. List their name, date of birth and salary, sorted alphabetically by name.

```
SELECT employee_name, date_of_birth, salary
FROM   employees
WHERE  salary > 50000
   AND date_of_birth >= DATE '2000-01-01'
ORDER BY employee_name;
```

Show the number of employees in each department with at least 5 employees. List the largest departments first.

```
SELECT department_id, COUNT(*)
FROM   employees
GROUP BY department_id
HAVING COUNT(*) >= 5
ORDER BY COUNT(*) DESC;
```

Section 1.3: Hello world! from table

Create a simple table

```
CREATE TABLE MY_table (
   what VARCHAR2(10),
   who VARCHAR2(10),
mark VARCHAR2(10)
);
```

插入值（如果为所有列提供值，可以省略目标列）

```
INSERT INTO my_table (what, who, mark) VALUES ('Hello', 'world', '!' );
INSERT INTO my_table VALUES ('Bye bye', 'ponies', '?' );
INSERT INTO my_table (what) VALUES('Hey');
```

记得提交，因为Oracle使用事务

```
COMMIT ;
```

查询你的数据：

```
SELECT what, who, mark FROM my_table WHERE what='Hello';
```

第1.4节：来自PL/SQL的Hello World

```
/* PL/SQL 是 Oracle 数据库的核心技术，允许您构建简洁、安全、优化的 SQL 和业务逻辑接口。 */


SET serveroutput ON

BEGIN
   DBMS_OUTPUT.PUT_LINE ('Hello World!');
END;
```

```
   mark VARCHAR2(10)
);
```

Insert values (you can omit target columns if you provide values for all columns)

```
INSERT INTO my_table (what, who, mark) VALUES ('Hello', 'world', '!' );
INSERT INTO my_table VALUES ('Bye bye', 'ponies', '?' );
INSERT INTO my_table (what) VALUES('Hey');
```

Remember to commit, because Oracle uses *transactions*

```
COMMIT;
```

Select your data:

```
SELECT what, who, mark FROM my_table WHERE what='Hello';
```

Section 1.4: Hello World from PL/SQL

```
/* PL/SQL is a core Oracle Database technology, allowing you to build clean, secure,
   optimized APIs to SQL and business logic. */

SET serveroutput ON

BEGIN
   DBMS_OUTPUT.PUT_LINE ('Hello World!');
END;
```

第2章：PL/SQL入门

第2.1节：Hello World

```
SET serveroutput ON

DECLARE
message CONSTANT VARCHAR2(32767):= 'Hello, World!';
BEGIN
   DBMS_OUTPUT.put_line(message);
END;
/
```

命令 SET serveroutput ON 在 SQL*Plus 和 SQL Developer 客户端中是必需的，用于启用 DBMS_OUTPUT 的输出。没有该命令将不会显示任何内容。

END; 行表示匿名 PL/SQL 块的结束。要从 SQL 命令行运行代码，您可能需要在代码最后一行之后的第一个空行开头输入/。当上述代码在 SQL 提示符下执行时，会产生以下结果：

```
你好，世界！

PL/SQL 过程成功完成。
```

第2.2节：PL/SQL的定义

PL/SQL（过程语言/结构化查询语言）是甲骨文公司为SQL和甲骨文关系数据库提供的过程扩展。PL/SQL可用于甲骨文数据库（自版本7起）、TimesTen内存数据库（自版本11.2.1起）和IBM DB2（自版本9.7起）。

PL/SQL中的基本单元称为块，由三部分组成：声明部分、可执行部分和异常处理部分。

```
声明
   <声明部分>
开始
   <可执行命令(s)>
异常
   <异常处理>
结束;
```

声明- 本部分以关键字DECLARE开始。它是可选部分，定义程序中使用的所有变量、游标、子程序及其他元素。

可执行命令 - 本部分包含在关键字 BEGIN 和 END 之间，是一个必需的部分。它由程序的可执行 PL/SQL 语句组成。它应至少包含一行可执行代码，该代码可以只是一个 NULL 命令，表示不执行任何操作。

异常处理 - 本部分以关键字 EXCEPTION 开始。该部分是可选的，包含处理程序中错误的异常。

每条 PL/SQL 语句以分号 (;) 结尾。PL/SQL 块可以嵌套在其他 PL/SQL 块中，使用 BEGIN 和 END。

Chapter 2: Getting started with PL/SQL

Section 2.1: Hello World

```
SET serveroutput ON

DECLARE
   message CONSTANT VARCHAR2(32767):= 'Hello, World!';
BEGIN
   DBMS_OUTPUT.put_line(message);
END;
/
```

Command `SET serveroutput ON` is required in SQL*Plus and SQL Developer clients to enable the output of `DBMS_OUTPUT`. Without the command nothing is displayed.

The `END;` line signals the end of the anonymous PL/SQL block. To run the code from SQL command line, you may need to type / at the beginning of the first blank line after the last line of the code. When the above code is executed at SQL prompt, it produces the following result:

```
Hello, World!

PL/SQL procedure successfully completed.
```

Section 2.2: Definition of PL/SQL

PL/SQL (Procedural Language/Structured Query Language) is Oracle Corporation's procedural extension for SQL and the Oracle relational database. PL/SQL is available in Oracle Database (since version 7), TimesTen in-memory database (since version 11.2.1), and IBM DB2 (since version 9.7).

The basic unit in PL/SQL is called a block, which is made up of three parts: a declarative part, an executable part, and an exception-building part.

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
   <EXCEPTION handling>
END;
```

Declarations - This section starts with the keyword DECLARE. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.

Executable Commands - This section is enclosed between the keywords BEGIN and END and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a NULL command to indicate that nothing should be executed.

Exception Handling - This section starts with the keyword EXCEPTION. This section is again optional and contains exception(s) that handle errors in the program.

Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using BEGIN and END.

在匿名块中，仅需要块的可执行部分，其他部分不是必需的。以下是一个简单匿名代码的示例，该代码不执行任何操作，但不会报错。

```
BEGIN
    NULL;
END;
/
```

缺少可执行指令会导致错误，因为 PL/SQL 不支持空块。例如，执行以下代码会导致错误：

```
BEGIN
END;
/
```

应用程序将引发错误：

```
END;
*
错误 在 第 2 行:
ORA-06550: 第2行，第1列:
PLS-00103: 遇到符号 "END" WHEN 期待以下之一:
( BEGIN CASE DECLARE EXIT FOR GOTO IF LOOP MOD NULL PRAGMA
RAISE RETURN SELECT UPDATE WHILE WITH <一个标识符>
<一个双引号限定的标识符> <一个绑定变量> <<
continue CLOSE CURRENT DELETE FETCH LOCK INSERT OPEN ROLLBACK
SAVEPOINT SET SQL EXECUTE COMMIT FORALL MERGE pipe purge
```

关键字 "END;" 下方的符号 " * " 表示以该块结束的代码块为空或结构错误。每个执行块都需要有执行指令，即使它什么也不做，就像我们的示例一样。

第2.3节：%TYPE 和 %ROWTYPE 的区别

%TYPE: 用于声明与指定表列类型相同的字段类型。

```
声明
vEmployeeName    Employee.Name%TYPE;
BEGIN
        SELECT Name
        INTO    vEmployeeName
        FROM    Employee
        WHERE  ROWNUM = 1;

        DBMS_OUTPUT.PUT_LINE(vEmployeeName);
END;
/
```

%ROWTYPE：用于声明一个记录，其类型与指定的表、视图或游标中的类型相同（= 多列）。

```
声明
rEmployee       Employee%ROWTYPE;
BEGIN
rEmployee.Name := '马特';
       rEmployee.Age := 31;

        DBMS_OUTPUT.PUT_LINE(rEmployee.Name);
```

In anonymous block, only executable part of block is required, other parts are not nessesary. Below is example of simple anonymous code, which does not do anything but perform without error reporting.

```
BEGIN
    NULL;
END;
/
```

Missing excecutable instruction leads to an error, becouse PL/SQL does not support empty blocks. For example, excecution of code below leads to an error:

```
BEGIN
END;
/
```

Application will raise error:

```
END;
*
ERROR AT line 2:
ORA-06550: line 2, column 1:
PLS-00103: Encountered the symbol "END" WHEN expecting one OF the following:
( BEGIN CASE DECLARE EXIT FOR GOTO IF LOOP MOD NULL PRAGMA
RAISE RETURN SELECT UPDATE WHILE WITH <an identifier>
<a double-quoted delimited-identifier> <a bind variable> <<
continue CLOSE CURRENT DELETE FETCH LOCK INSERT OPEN ROLLBACK
SAVEPOINT SET SQL EXECUTE COMMIT FORALL MERGE pipe purge
```

Symbol " * " in line below keyword "END;" means, that the block which ends with this block is empty or bad constructed. Every execution block needs instructions to do, even if it does nothing, like in our example.

Section 2.3: Difference between %TYPE and %ROWTYPE

%TYPE: Used to declare a field with the same type as that of a specified table's column.

```
DECLARE
        vEmployeeName    Employee.Name%TYPE;
BEGIN
        SELECT Name
        INTO    vEmployeeName
        FROM    Employee
        WHERE  ROWNUM = 1;

        DBMS_OUTPUT.PUT_LINE(vEmployeeName);
END;
/
```

%ROWTYPE: Used to declare a record with the same types as found in the specified table, view or cursor (= multiple columns).

```
DECLARE
        rEmployee      Employee%ROWTYPE;
BEGIN
        rEmployee.Name := 'Matt';
        rEmployee.Age := 31;

        DBMS_OUTPUT.PUT_LINE(rEmployee.Name);
        DBMS_OUTPUT.PUT_LINE(rEmployee.Age);
END;
/
```

第2.4节：创建或替换视图

在本例中，我们将创建一个视图。
视图主要用作从多个表中简单获取数据的方式。

示例1：
基于单个表的选择创建的视图。

```
创建或替换视图 LessonView 为
       选择     L.*
       FROM       Lesson L;
```

示例 2：
对多个表进行选择的视图。

```
CREATE OR REPLACE VIEW ClassRoomLessonView AS
       SELECT     C.Id,
C.Name,
                  L.Subject,
                  L.Teacher
       FROM       ClassRoom C,
               Lesson L
       WHERE      C.Id = L.ClassRoomId;
```

要在查询中调用这些视图，可以使用 select 语句。

```
SELECT * FROM LessonView;
SELECT * FROM ClassRoomLessonView;
```

第 2.5 节：创建表

下面我们将创建一个包含 3 列的表。
列 Id 必须填写，因此我们定义为 NOT NULL。
在“Contract”列上，我们还添加了一个检查，确保唯一允许的值是“Y”或“N”。如果执行插入操作且在插入时未指定此列，则默认插入“N”。

```
CREATE TABLE Employee (
Id           NUMBER NOT NULL,
        Name          VARCHAR2(60),
        Contract      CHAR DEFAULT 'N' NOT NULL,
        ---
CONSTRAINT p_Id PRIMARY KEY(Id),
        CONSTRAINT c_Contract CHECK (Contract IN('Y','N'))
);
```

第2.6节：关于PL/SQL

PL/SQL代表SQL的过程语言扩展。PL/SQL仅作为其他软件产品中的“启用技术”存在；它并不存在作为独立语言。您可以在Oracle关系数据库、Oracle服务器以及客户端应用开发工具（如Oracle Forms）中使用PL/SQL。

以下是您可能使用PL/SQL的一些方式：

```
        DBMS_OUTPUT.PUT_LINE(rEmployee.Age);
END;
/
```

Section 2.4: Create or replace a view

In this example we are going to create a view.
A view is mostly used as a simple way of fetching data from multiple tables.

Example 1:
View with a select on one table.

```
CREATE OR REPLACE VIEW LessonView AS
       SELECT     L.*
       FROM       Lesson L;
```

Example 2:
View with a select on multiple tables.

```
CREATE OR REPLACE VIEW ClassRoomLessonView AS
       SELECT     C.Id,
                  C.Name,
                  L.Subject,
                  L.Teacher
       FROM       ClassRoom C,
                  Lesson L
       WHERE      C.Id = L.ClassRoomId;
```

To call this views in a query you can use a select statement.

```
SELECT * FROM LessonView;
SELECT * FROM ClassRoomLessonView;
```

Section 2.5: Create a table

Below we are going to create a table with 3 columns.
The column `Id` must be filled is, so we define it `NOT NULL`.
On the column `Contract` we also add a check so that the only value allowed is 'Y' or 'N'. If an insert in done and this column is not specified during the insert then default a 'N' is inserted.

```
CREATE TABLE Employee (
        Id            NUMBER NOT NULL,
        Name          VARCHAR2(60),
        Contract      CHAR DEFAULT 'N' NOT NULL,
        ---
        CONSTRAINT p_Id PRIMARY KEY(Id),
        CONSTRAINT c_Contract CHECK (Contract IN('Y','N'))
);
```

Section 2.6: About PL/SQL

PL/SQL stands for Procedural Language extensions to SQL. PL/SQL is available only as an "enabling technology" within other software products; it does not exist as a standalone language. You can use PL/SQL in the Oracle relational database, in the Oracle Server, and in client-side application development tools, such as Oracle Forms. Here are some of the ways you might use PL/SQL:

1. 构建存储过程。
2. 创建数据库触发器。
3. 在您的Oracle Forms应用程序中实现客户端逻辑。
4. 将万维网页主页链接到Oracle数据库。

1. To build stored procedures. .
2. To create database triggers.
3. To implement client-side logic in your Oracle Forms application.
4. To link a World Wide Web home page to an Oracle database.

第3章：匿名PL/SQL块

第3.1节：匿名块示例

```
声明
    -- 声明一个变量
message VARCHAR2(20);
BEGIN
  -- 给变量赋值
message := 'HELLO WORLD';

  -- 将消息打印到屏幕
  DBMS_OUTPUT.PUT_LINE(message);
END;
/
```

Chapter 3: Anonymous PL/SQL Block

Section 3.1: An example of an anonymous block

```
DECLARE
    -- declare a variable
    message VARCHAR2(20);
BEGIN
  -- assign value to variable
  message := 'HELLO WORLD';

  -- print message to screen
  DBMS_OUTPUT.PUT_LINE(message);
END;
/
```

第4章：PL/SQL过程

PL/SQL过程是一组存储在服务器上的SQL语句，用于重复使用。它提高了性能，因为SQL语句不必每次执行时都重新编译。

存储过程在多个应用程序需要相同代码时非常有用。使用存储过程可以消除冗余，使代码更简洁。当客户端和服务器之间需要数据传输时，过程在某些情况下可以降低通信成本。

第4.1节：语法

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] 类型 [, ...])]
{IS | AS}
  < 声明 >
BEGIN
  < 过程体 >
EXCEPTION                        -- 异常处理部分开始
  <EXCEPTION 处理代码写在这里 >
   WHEN exception1 THEN
       exception1-处理-语句
END procedure_name;
```

- procedure-name 指定过程的名称。
- [OR REPLACE] 选项允许修改现有的存储过程。
- 可选参数列表包含参数的名称、模式和类型。IN 表示该值将从外部传入，OUT 表示该参数将用于向过程外部返回值。如果未指定模式，则参数默认为 IN 模式。

- 在声明部分，我们可以声明将在主体部分使用的变量。
- 过程主体包含可执行部分。
- 创建独立过程时，使用 AS 关键字代替 IS 关键字。
- 异常部分将处理过程中的异常。此部分是可选的。

第4.2节：Hello World

下面的简单过程在支持DBMS_OUTPUT的客户端中显示文本“Hello World”。

```
CREATE OR REPLACE PROCEDURE helloworld
AS
BEGIN
   DBMS_OUTPUT.put_line('Hello World!');
END;
/
```

你需要在SQL提示符下执行此操作以在数据库中创建该过程，或者你可以运行下面的查询以获得相同的结果：

```
SELECT 'Hello World!' FROM dual;
```

第4.3节：输入/输出参数

PL/SQL使用IN、OUT、IN OUT关键字来定义传入参数的处理方式。

IN指定该参数为只读，过程不能更改其值。

Chapter 4: PL/SQL procedure

PL/SQL procedure is a group of SQL statements stored on the server for reuse. It increases the performance because the SQL statements do not have to be recompiled every time it is executed.

Stored procedures are useful when same code is required by multiple applications. Having stored procedures eliminates redundancy, and introduces simplicity to the code. When data transfer is required between the client and server, procedures can reduce communication cost in certain situations.

Section 4.1: Syntax

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] TYPE [, ...])]
{IS | AS}
  < declarations >
BEGIN
  < procedure_body >
EXCEPTION                       -- Exception-handling part begins
  <EXCEPTION handling goes here >
   WHEN exception1 THEN
       exception1-handling-statements
END procedure_name;
```

- procedure-name specifies the name of the procedure.
- [OR REPLACE] option allows modifying an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure. If no mode is specified, parameter is assumed to be of IN mode.
- In the declaration section we can declare variables which will be used in the body part.
- procedure-body contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.
- exception section will handle the exceptions from the procedure. This section is optional.

Section 4.2: Hello World

The following simple procedure displays the text "Hello World" in a client that supports DBMS_OUTPUT.

```
CREATE OR REPLACE PROCEDURE helloworld
AS
BEGIN
   DBMS_OUTPUT.put_line('Hello World!');
END;
/
```

You need to execute this at the SQL prompt to create the procedure in the database, or you can run the query below to get the same result:

```
SELECT 'Hello World!' FROM dual;
```

Section 4.3: In/Out Parameters

PL/SQL uses IN, OUT, IN OUT keywords to define what can happen to a passed parameter.

IN specifies that the parameter is read only and the value cannot be changed by the procedure.

OUT指定该参数为只写，过程可以为其赋值，但不能引用其值。

IN OUT指定该参数既可引用也可修改。

```
PROCEDURE procedureName(x IN INT, strVar IN VARCHAR2, ans OUT VARCHAR2)
…
...
END procedureName;


procedureName(firstvar, secondvar, thirdvar);
```

上述示例中传入的变量需要按照过程参数部分定义的类型进行声明。

OUT specifies the parameter is write only and a procedure can assign a value to it, but not reference the value.

IN OUT specifies the parameter is available for reference and modification.

```
PROCEDURE procedureName(x IN INT, strVar IN VARCHAR2, ans OUT VARCHAR2)
...
...
END procedureName;


procedureName(firstvar, secondvar, thirdvar);
```

The variables passed in the above example need to be typed as they are defined in the procedure parameter section.

第5章：数据字典

第5.1节：描述数据库中的所有对象

```
SELECT *
FROM dba_objects
```

第5.2节：查看您有权限访问的所有数据字典视图

```
SELECT * FROM dict
```

第5.3节：存储对象的文本源代码

USER_SOURCE 描述当前用户拥有的存储对象的文本源代码。该视图不显示OWNER列。

```
SELECT * FROM user_source WHERE TYPE='TRIGGER' AND LOWER(text) LIKE '%order%'
```

ALL_SOURCE 描述当前用户可访问的存储对象的文本源。

```
SELECT * FROM all_source WHERE owner=:owner
```

DBA_SOURCE 描述数据库中所有存储对象的文本源。

```
SELECT * FROM dba_source
```

第5.4节：获取Oracle中所有表的列表

```
SELECT owner, table_name
FROM all_tables
```

ALL_TAB_COLUMNS 描述当前用户可访问的表、视图和聚簇的列。COLS 是USER_TAB_COLUMNS的同义词。

```
SELECT *
FROM all_tab_columns
WHERE table_name = :tname
```

第5.5节：权限信息

授予用户的所有角色。

```
SELECT *
FROM dba_role_privs
WHERE grantee= :username
```

授予用户的权限：

1.系统权限

```
SELECT *
```

Chapter 5: Data Dictionary

Section 5.1: Describes all objects in the database

```
SELECT *
FROM dba_objects
```

Section 5.2: To see all the data dictionary views to which you have access

```
SELECT * FROM dict
```

Section 5.3: Text source of the stored objects

`USER_SOURCE` describes the text source of the stored objects owned by the current user. This view does not display the `OWNER` column.

```
SELECT * FROM user_source WHERE TYPE='TRIGGER' AND LOWER(text) LIKE '%order%'
```

`ALL_SOURCE` describes the text source of the stored objects accessible to the current user.

```
SELECT * FROM all_source WHERE owner=:owner
```

`DBA_SOURCE` describes the text source of all stored objects in the database.

```
SELECT * FROM dba_source
```

Section 5.4: Get list of all tables in Oracle

```
SELECT owner, table_name
FROM all_tables
```

`ALL_TAB_COLUMNS` describes the columns of the tables, views, and clusters accessible to the current user. `COLS` is a synonym for `USER_TAB_COLUMNS`.

```
SELECT *
FROM all_tab_columns
WHERE table_name = :tname
```

Section 5.5: Privilege information

All roles granted to user.

```
SELECT *
FROM dba_role_privs
WHERE grantee= :username
```

Privileges granted to user:

1. system privileges

```
SELECT *
```

```
FROM dba_sys_privs
WHERE grantee = :username
```

2.对象权限

```
SELECT *
FROM dba_tab_privs
WHERE grantee = :username
```

授予角色的权限。

授予其他角色的角色。

```
SELECT *
FROM role_role_privs
WHERE role IN (SELECT granted_role FROM dba_role_privs WHERE grantee= :username)
```

1.系统权限

```
SELECT *
FROM role_sys_privs
WHERE role IN (SELECT granted_role FROM dba_role_privs WHERE grantee= :username)
```

2.对象权限

```
SELECT *
FROM role_tab_privs
WHERE role IN (SELECT granted_role FROM dba_role_privs WHERE grantee= :username)
```

第5.6节：Oracle版本

```
SELECT *
FROM v$version
```

```
FROM dba_sys_privs
WHERE grantee = :username
```

2. object grants

```
SELECT *
FROM dba_tab_privs
WHERE grantee = :username
```

Permissions granted to roles.

Roles granted to other roles.

```
SELECT *
FROM role_role_privs
WHERE role IN (SELECT granted_role FROM dba_role_privs WHERE grantee= :username)
```

1. system privileges

```
SELECT *
FROM role_sys_privs
WHERE role IN (SELECT granted_role FROM dba_role_privs WHERE grantee= :username)
```

2. object grants

```
SELECT *
FROM role_tab_privs
WHERE role IN (SELECT granted_role FROM dba_role_privs WHERE grantee= :username)
```

Section 5.6: Oracle version

```
SELECT *
FROM v$version
```

第6章：日期

第6.1节：日期运算 - 日期之间的差异（以天、小时、分钟和/或秒计）

在Oracle中，两个DATE之间的差值（以天及其分数表示）可以通过减法计算得到：

```
SELECT DATE '2016-03-23' - DATE '2015-12-25' AS difference FROM DUAL;
```

输出两个日期之间的天数：

```
DIFFERENCE
----------
89
```

并且：

```
SELECT TO_DATE( '2016-01-02 01:01:12', 'YYYY-MM-DD HH24:MI:SS' )
         - TO_DATE( '2016-01-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS' )
         AS difference
FROM   DUAL
```

输出两个日期之间的天数小数部分：

```
DIFFERENCE
----------
1.0425
```

小时、分钟或秒数的差异可以通过分别将该数字乘以24、24*60或24*60*60来计算。

前面的例子可以修改为使用以下方法获取两个日期之间的天数、小时、分钟和秒数：

```
SELECT TRUNC( difference                      ) AS days,
       TRUNC( MOD( difference * 24,       24 ) ) AS hours,
       TRUNC( MOD( difference * 24*60,    60 ) ) AS minutes,
       TRUNC( MOD( difference * 24*60*60, 60 ) ) AS seconds
FROM   (
  SELECT TO_DATE( '2016-01-02 01:01:12', 'YYYY-MM-DD HH24:MI:SS' )
         - TO_DATE( '2016-01-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS' )
         AS difference
  FROM   DUAL
```

);

（注：使用TRUNC()而非FLOOR()以正确处理负数差异。）

输出结果：

```
天  小时 分钟  秒
---- ----- ------- -------
1    1     1       12
```

Chapter 6: Dates

Section 6.1: Date Arithmetic - Difference between Dates in Days, Hours, Minutes and/or Seconds

In oracle, the difference (in days and/or fractions thereof) between two DATEs can be found using subtraction:

```
SELECT DATE '2016-03-23' - DATE '2015-12-25' AS difference FROM DUAL;
```

Outputs the number of days between the two dates:

```
DIFFERENCE
----------
89
```

And:

```
SELECT TO_DATE( '2016-01-02 01:01:12', 'YYYY-MM-DD HH24:MI:SS' )
         - TO_DATE( '2016-01-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS' )
         AS difference
FROM   DUAL
```

Outputs the fraction of days between two dates:

```
DIFFERENCE
----------
1.0425
```

The difference in hours, minutes or seconds can be found by multiplying this number by 24, 24*60 or 24*60*60 respectively.

The previous example can be changed to get the days, hours, minutes and seconds between two dates using:

```
SELECT TRUNC( difference                      ) AS days,
       TRUNC( MOD( difference * 24,       24 ) ) AS hours,
       TRUNC( MOD( difference * 24*60,    60 ) ) AS minutes,
       TRUNC( MOD( difference * 24*60*60, 60 ) ) AS seconds
FROM   (
  SELECT TO_DATE( '2016-01-02 01:01:12', 'YYYY-MM-DD HH24:MI:SS' )
         - TO_DATE( '2016-01-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS' )
         AS difference
  FROM   DUAL
```

);

(Note: TRUNC() is used rather than FLOOR() to correctly handle negative differences.)

Outputs:

```
DAYS HOURS MINUTES SECONDS
---- ----- ------- -------
1    1     1       12
```

前面的例子也可以通过将数值差转换为间隔来解决
NUMTODSINTERVAL()：

```
SELECT EXTRACT( DAY    FROM difference ) AS 天数,
       EXTRACT( HOUR   FROM difference ) AS 小时,
       EXTRACT( MINUTE FROM difference ) AS 分钟,
       EXTRACT( SECOND FROM difference ) AS 秒数
FROM   (
  SELECT NUMTODSINTERVAL(
           TO_DATE( '2016-01-02 01:01:12', 'YYYY-MM-DD HH24:MI:SS' )
             - TO_DATE( '2016-01-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS' ),
           '天'
         ) AS difference
  FROM   DUAL
);
```

第6.2节：设置默认日期格式模型

当 Oracle 隐式地将 DATE 类型转换为字符串或反之（或者当显式调用 TO_CHAR() 或 TO_DATE() 且未指定格式模型时），将使用 NLS_DATE_FORMAT 会话参数作为转换的格式模型。如果字面值与格式模型不匹配，则会引发异常。

您可以使用以下命令查看此参数：

```
SELECT VALUE FROM NLS_SESSION_PARAMETERS WHERE PARAMETER = 'NLS_DATE_FORMAT';
```

您可以在当前会话中使用以下命令设置此值：

```
ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH24:MI:SS';
```

（注意：这不会更改其他用户的值。）

如果您依赖NLS_DATE_FORMAT来为TO_DATE()或TO_CHAR()提供格式掩码，那么当此值被更改时，查询出错也不足为奇。

第6.3节：日期运算——日期之间以月或年为单位的差异

两个日期之间的月份差异可以使用MONTHS_BETWEEN(date1, date2)函数计算：

```
SELECT MONTHS_BETWEEN( DATE '2016-03-10', DATE '2015-03-10' ) AS difference FROM DUAL;
```

输出结果：

```
DIFFERENCE
----------
12
```

如果差值包含部分月份，则会根据每个月有31天来返回该月的分数部分：

```
SELECT MONTHS_BETWEEN( DATE '2015-02-15', DATE '2015-01-01' ) AS difference FROM DUAL;
```

输出结果：

The previous example can also be solved by converting the numeric difference to an interval using NUMTODSINTERVAL():

```
SELECT EXTRACT( DAY    FROM difference ) AS days,
       EXTRACT( HOUR   FROM difference ) AS hours,
       EXTRACT( MINUTE FROM difference ) AS minutes,
       EXTRACT( SECOND FROM difference ) AS seconds
FROM   (
  SELECT NUMTODSINTERVAL(
           TO_DATE( '2016-01-02 01:01:12', 'YYYY-MM-DD HH24:MI:SS' )
             - TO_DATE( '2016-01-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS' ),
           'DAY'
         ) AS difference
  FROM   DUAL
);
```

Section 6.2: Setting the Default Date Format Model

When Oracle implicitly converts from a DATE to a string or vice-versa (or when TO_CHAR() or TO_DATE() are explicitly called without a format model) the NLS_DATE_FORMAT session parameter will be used as the format model in the conversion. If the literal does not match the format model then an exception will be raised.

You can review this parameter using:

```
SELECT VALUE FROM NLS_SESSION_PARAMETERS WHERE PARAMETER = 'NLS_DATE_FORMAT';
```

You can set this value within your current session using:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH24:MI:SS';
```

(Note: this does not change the value for any other users.)

If you rely on the NLS_DATE_FORMAT to provide the format mask in TO_DATE() or TO_CHAR() then you should not be surprised when your queries break if this value is ever changed.

Section 6.3: Date Arithmetic - Difference between Dates in Months or Years

The difference in months between two dates can be found using the MONTHS_BETWEEN(date1, date2):

```
SELECT MONTHS_BETWEEN( DATE '2016-03-10', DATE '2015-03-10' ) AS difference FROM DUAL;
```

Outputs:

```
DIFFERENCE
----------
12
```

If the difference includes part months then it will return the fraction of the month based on there being **31** days in each month:

```
SELECT MONTHS_BETWEEN( DATE '2015-02-15', DATE '2015-01-01' ) AS difference FROM DUAL;
```

Outputs:

```
DIFFERENCE
----------
1.4516129
```

由于MONTHS_BETWEEN假设每个月有31天，而实际上每个月的天数可能更少，因此跨越月份边界的差值可能会产生不同的结果。

示例：

```
SELECT MONTHS_BETWEEN( DATE'2016-02-01', DATE'2016-02-01' - INTERVAL '1' DAY ) AS "JAN-FEB",
       MONTHS_BETWEEN( DATE'2016-03-01', DATE'2016-03-01' - INTERVAL '1' DAY ) AS "FEB-MAR",
       MONTHS_BETWEEN( DATE'2016-04-01', DATE'2016-04-01' - INTERVAL '1' DAY ) AS "MAR-APR",
       MONTHS_BETWEEN( DATE'2016-05-01', DATE'2016-05-01' - INTERVAL '1' DAY ) AS "APR-MAY"
FROM   DUAL;
```

输出：

```
JAN-FEB FEB-MAR MAR-APR APR-MAY
------- ------- ------- -------
0.03226 0.09677 0.03226 0.06452
```

年份差值可以通过将月份差值除以12来计算。

第6.4节：提取日期的年、月、日、时、分或秒组成部分

可以使用EXTRACT([YEAR | MONTH | DAY

] FROM datevalue)来获取DATE数据类型的年、月或日部分

```
SELECT EXTRACT (YEAR  FROM DATE '2016-07-25') AS YEAR,
       EXTRACT (MONTH FROM DATE '2016-07-25') AS MONTH,
       EXTRACT (DAY   FROM DATE '2016-07-25') AS DAY
FROM DUAL;
```

输出结果：

```
YEAR MONTH DAY
---- ----- ---
2016     7  25
```

时间（小时、分钟或秒）部分可以通过以下方式获取：

- 使用CAST(datevalue AS TIMESTAMP)将DATE转换为TIMESTAMP，然后使用EXTRACT([HOUR | MINUTE | SECOND] FROM timestampvalue)；或者
- 使用TO_CHAR(datevalue, format_model)将值作为字符串获取。

例如：

```
SELECT EXTRACT( HOUR   FROM CAST( datetime AS TIMESTAMP ) ) AS Hours,
       EXTRACT( MINUTE FROM CAST( datetime AS TIMESTAMP ) ) AS Minutes,
       EXTRACT( SECOND FROM CAST( datetime AS TIMESTAMP ) ) AS Seconds
FROM   (
  SELECT TO_DATE( '2016-01-01 09:42:01', 'YYYY-MM-DD HH24:MI:SS' ) AS datetime FROM DUAL
```

```
DIFFERENCE
----------
1.4516129
```

Due to MONTHS_BETWEEN assuming 31 days per month when there can be fewer days per month then this can result in different values for differences spanning the boundaries between months.

Example:

```
SELECT MONTHS_BETWEEN( DATE'2016-02-01', DATE'2016-02-01' - INTERVAL '1' DAY ) AS "JAN-FEB",
       MONTHS_BETWEEN( DATE'2016-03-01', DATE'2016-03-01' - INTERVAL '1' DAY ) AS "FEB-MAR",
       MONTHS_BETWEEN( DATE'2016-04-01', DATE'2016-04-01' - INTERVAL '1' DAY ) AS "MAR-APR",
       MONTHS_BETWEEN( DATE'2016-05-01', DATE'2016-05-01' - INTERVAL '1' DAY ) AS "APR-MAY"
FROM   DUAL;
```

Output:

```
JAN-FEB FEB-MAR MAR-APR APR-MAY
------- ------- ------- -------
0.03226 0.09677 0.03226 0.06452
```

The difference in years can be found by dividing the month difference by 12.

Section 6.4: Extract the Year, Month, Day, Hour, Minute or Second Components of a Date

The year, month or day components of a DATE data type can be found using the EXTRACT([YEAR | MONTH | DAY] FROM datevalue)

```
SELECT EXTRACT (YEAR  FROM DATE '2016-07-25') AS YEAR,
       EXTRACT (MONTH FROM DATE '2016-07-25') AS MONTH,
       EXTRACT (DAY   FROM DATE '2016-07-25') AS DAY
FROM DUAL;
```

Outputs:

```
YEAR MONTH DAY
---- ----- ---
2016     7  25
```

The time (hour, minute or second) components can be found by either:

- Using CAST(datevalue AS TIMESTAMP) to convert the DATE to a TIMESTAMP and then using EXTRACT([HOUR | MINUTE | SECOND] FROM timestampvalue); or
- Using TO_CHAR(datevalue, format_model) to get the value as a string.

For example:

```
SELECT EXTRACT( HOUR   FROM CAST( datetime AS TIMESTAMP ) ) AS Hours,
       EXTRACT( MINUTE FROM CAST( datetime AS TIMESTAMP ) ) AS Minutes,
       EXTRACT( SECOND FROM CAST( datetime AS TIMESTAMP ) ) AS Seconds
FROM   (
  SELECT TO_DATE( '2016-01-01 09:42:01', 'YYYY-MM-DD HH24:MI:SS' ) AS datetime FROM DUAL
);
```

输出结果：

```
小时 分钟 秒
----- ------- -------
9      42       1
```

第6.5节：生成无时间部分的日期

所有DATE都有时间部分；然而，通常将不需要包含时间信息的日期存储为小时/分钟/秒均为零（即午夜）。

使用ANSIDATE字面量（使用ISO 8601日期格式）：

```
SELECT DATE '2000-01-01' FROM DUAL;
```

使用TO_DATE()将其从字符串字面量转换：

```
SELECT TO_DATE( '2001-01-01', 'YYYY-MM-DD' ) FROM DUAL;
```

（有关日期格式模型的更多信息，请参见Oracle文档。）

或者：

```
SELECT TO_DATE(
         '2000年1月1日 00:00 上午',
         '月 日, 年份, 12小时制:分钟 上午',
         'NLS_DATE_LANGUAGE = American'
       )
FROM   DUAL;
```

（如果您正在转换特定语言的术语，例如月份名称，建议在TO_DATE()函数中包含第三个nlsparam参数，并指定预期的语言。）

第6.6节：生成带时间组件的日期

使用TO_DATE()将其从字符串字面量转换：

```
SELECT TO_DATE( '2000-01-01 12:00:00', 'YYYY-MM-DD HH24:MI:SS' ) FROM DUAL;
```

或者使用TIMESTAMP字面量：

```
CREATE TABLE date_table(
  date_value DATE
);

INSERT INTO date_table ( date_value ) VALUES ( TIMESTAMP '2000-01-01 12:00:00' );
```

Oracle在将TIMESTAMP存储到表的DATE列时会隐式转换为DATE；但是您可以
显式地使用CAST()将值转换为DATE：

```
SELECT CAST( TIMESTAMP '2000-01-01 12:00:00' AS DATE ) FROM DUAL;
```

```
);
```

Outputs:

```
HOURS MINUTES SECONDS
----- ------- -------
9      42       1
```

Section 6.5: Generating Dates with No Time Component

All DATEs have a time component; however, it is customary to store dates which do not need to include time information with the hours/minutes/seconds set to zero (i.e. midnight).

Use an ANSI DATE literal (using ISO 8601 Date format):

```
SELECT DATE '2000-01-01' FROM DUAL;
```

Convert it from a string literal using TO_DATE():

```
SELECT TO_DATE( '2001-01-01', 'YYYY-MM-DD' ) FROM DUAL;
```

(More information on the date format models can be found in the Oracle documentation.)

or:

```
SELECT TO_DATE(
         'January 1, 2000, 00:00 A.M.',
         'Month dd, YYYY, HH12:MI A.M.',
         'NLS_DATE_LANGUAGE = American'
       )
FROM   DUAL;
```

(If you are converting language specific terms such as month names then it is good practice to include the 3rd nlsparam parameter to the TO_DATE() function and specify the language to be expected.)

Section 6.6: Generating Dates with a Time Component

Convert it from a string literal using TO_DATE():

```
SELECT TO_DATE( '2000-01-01 12:00:00', 'YYYY-MM-DD HH24:MI:SS' ) FROM DUAL;
```

Or use a TIMESTAMP literal:

```
CREATE TABLE date_table(
  date_value DATE
);

INSERT INTO date_table ( date_value ) VALUES ( TIMESTAMP '2000-01-01 12:00:00' );
```

Oracle will implicitly cast a TIMESTAMP to a DATE when storing it in a DATE column of a table; however you can explicitly CAST() the value to a DATE:

```
SELECT CAST( TIMESTAMP '2000-01-01 12:00:00' AS DATE ) FROM DUAL;
```

第6.7节：日期的格式

在Oracle中，DATE数据类型没有格式；当Oracle将DATE发送给客户端程序（SQL/Plus、SQL/Developer、Toad、Java、Python等）时，它会发送表示日期的7或8字节。

未存储在表中的DATE（即由SYSDATE生成且使用DUMP()命令时显示“类型13”）有8字节，结构如下（右侧数字为内部表示）
2012-11-26 16:41:09）：

```
字节值                              示例
---- ------------------------------- --------------------------------------
1    年份模256                         220
2    年份乘以256                        7   (7 * 256 + 220 = 2012)
3    月份                             11
4    日期                             26
5    小时                             16
6    分钟                             41
7    秒数                             9
8    未使用                           0
```

存储在表中的DATE（使用DUMP()命令时显示“类型12”）有7字节，结构如下（右侧数字为2012-11-26 16:41:09的内部表示）：

```
字节值                              示例
---- ------------------------------- --------------------------------------
1     （年份除以100的商）+ 100       120
2     （年份模100）+ 100             112 ((120-100)*100 + (112-100) = 2012)
3    月份                             11
4    天                                26
5    小时 + 1                          17
6    分钟 + 1                         42
7    秒 + 1                          10
```

如果您希望日期具有特定格式，则需要将其转换为具有格式的内容（即字符串）。SQL客户端可能会隐式执行此操作，或者您可以使用TO_CHAR(DATE,format_model, nls_params)显式地将值转换为字符串。

第6.8节：将日期转换为字符串

使用TO_CHAR(DATE [, format_model [, nls_params]])：

（注意：如果未提供format model，则将使用NLS_DATE_FORMAT会话参数作为默认格式模型；该参数每个会话可能不同，因此不应依赖。建议始终指定格式模型。）

```
CREATE TABLE table_name (
  date_value DATE
);

INSERT INTO table_name ( date_value ) VALUES ( DATE '2000-01-01' );
INSERT INTO table_name ( date_value ) VALUES ( TIMESTAMP '2016-07-21 08:00:00' );
INSERT INTO table_name ( date_value ) VALUES ( SYSDATE );
```

然后：

Section 6.7: The Format of a Date

In Oracle a DATE data type does not have a format; when Oracle sends a DATE to the client program (SQL/Plus, SQL/Developer, Toad, Java, Python, etc) it will send 7- or 8- bytes which represent the date.

A DATE which is not stored in a table (i.e. generated by SYSDATE and having "type 13" when using the DUMP() command) has 8-bytes and has the structure (the numbers on the right are the internal representation of 2012-11-26 16:41:09):

```
BYTE VALUE                           EXAMPLE
---- ------------------------------- --------------------------------------
1    Year modulo 256                 220
2    Year multiples of 256           7   (7 * 256 + 220 = 2012)
3    Month                           11
4    Day                             26
5    Hours                           16
6    Minutes                         41
7    Seconds                         9
8    Unused                          0
```

A DATE which is stored in a table ("type 12" when using the DUMP() command) has 7-bytes and has the structure (the numbers on the right are the internal representation of 2012-11-26 16:41:09):

```
BYTE VALUE                           EXAMPLE
---- ------------------------------- --------------------------------------
1    ( Year multiples of 100 ) + 100 120
2    ( Year modulo 100 ) + 100       112 ((120-100)*100 + (112-100) = 2012)
3    Month                           11
4    Day                             26
5    Hours + 1                       17
6    Minutes + 1                     42
7    Seconds + 1                     10
```

If you want the date to have a specific format then you will need to convert it to something that has a format (i.e. a string). The SQL client may implicitly do this or you can explicitly convert the value to a string using TO_CHAR(DATE, format_model, nls_params).

Section 6.8: Converting Dates to a String

Use TO_CHAR(DATE [, format_model [, nls_params]]):

(Note: if a format model is not provided then the NLS_DATE_FORMAT session parameter will be used as the default format model; this can be different for every session so should not be relied on. It is good practice to always specify the format model.)

```
CREATE TABLE table_name (
  date_value DATE
);

INSERT INTO table_name ( date_value ) VALUES ( DATE '2000-01-01' );
INSERT INTO table_name ( date_value ) VALUES ( TIMESTAMP '2016-07-21 08:00:00' );
INSERT INTO table_name ( date_value ) VALUES ( SYSDATE );
```

Then:

```
SELECT TO_CHAR( date_value, 'YYYY-MM-DD' ) AS formatted_date FROM table_name;
```

输出结果：

```
FORMATTED_DATE
--------------
2000-01-01
2016-07-21
2016-07-21
```

并且：

```
SELECT TO_CHAR(
date_value,
         'FMMonth d yyyy, hh12:mi:ss AM',
         'NLS_DATE_LANGUAGE = French'
       ) AS formatted_date
FROM   table_name;
```

输出结果：

```
FORMATTED_DATE
-----------------------------
一月    01 2000, 12:00:00 AM
七月    21 2016, 08:00:00 AM
七月    21 2016, 19:08:31 PM
```

第6.9节：更改SQL/Plus或SQL Developer显示日期的方式

当SQL/Plus或SQL Developer显示日期时，它们会使用默认的日期格式模型对日期进行隐式转换为字符串（参见设置默认日期格式模型示例）。

您可以通过更改NLS_DATE_FORMAT参数来改变日期的显示方式。

第6.10节：时区和夏令时

DATE数据类型不处理时区或夏令时的变化。

任一：

- 使用TIMESTAMP WITH TIME ZONE数据类型；或者
- 在您的应用程序逻辑中处理这些变化。

一个DATE可以存储为协调世界时（UTC），并像这样转换为当前会话时区：

```
SELECT FROM_TZ(
         CAST(
           TO_DATE( '2016-01-01 12:00:00', 'YYYY-MM-DD HH24:MI:SS' )
           作为时间戳
         ),
         'UTC'
       )
       在LOCAL作为时间
```

```
SELECT TO_CHAR( date_value, 'YYYY-MM-DD' ) AS formatted_date FROM table_name;
```

Outputs:

```
FORMATTED_DATE
--------------
2000-01-01
2016-07-21
2016-07-21
```

And:

```
SELECT TO_CHAR(
         date_value,
         'FMMonth d yyyy, hh12:mi:ss AM',
         'NLS_DATE_LANGUAGE = French'
       ) AS formatted_date
FROM   table_name;
```

Outputs:

```
FORMATTED_DATE
-----------------------------
Janvier   01 2000, 12:00:00 AM
Juillet   21 2016, 08:00:00 AM
Juillet   21 2016, 19:08:31 PM
```

Section 6.9: Changing How SQL/Plus or SQL Developer Display Dates

When SQL/Plus or SQL Developer display dates they will perform an implicit conversion to a string using the default date format model (see the Setting the Default Date Format Model example).

You can change how a date is displayed by changing the `NLS_DATE_FORMAT` parameter.

Section 6.10: Time Zones and Daylight Savings Time

The `DATE` data type does not handle time zones or changes in daylight savings time.

Either:

- use the `TIMESTAMP WITH TIME ZONE` data type; or
- handle the changes in your application logic.

A `DATE` can be stored as Coordinated Universal Time (UTC) and converted to the current session time zone like this:

```
SELECT FROM_TZ(
         CAST(
           TO_DATE( '2016-01-01 12:00:00', 'YYYY-MM-DD HH24:MI:SS' )
           AS TIMESTAMP
         ),
         'UTC'
       )
       AT LOCAL AS TIME
```

```
FROM    DUAL;
```

如果您执行ALTER SESSION SET TIME_ZONE = '+01:00'；那么输出是：

```
时间
------------------------------------
2016-01-01 13:00:00.000000000 +01:00
```

并且 ALTER SESSION SET TIME_ZONE = 'PST'; 则输出为：

```
时间
------------------------------------
2016-01-01 04:00:00.000000000 PST
```

第6.11节：闰秒

Oracle 不处理闰秒。详情请参见 My Oracle Support 说明 2019397.2 和 730795.1。

第6.12节：获取星期几

您可以使用 TO_CHAR(date_value, 'D') 来获取星期几。

但是，这取决于 NLS_TERRITORY 会话参数：

```
ALTER SESSION SET NLS_TERRITORY = 'AMERICA';         -- 一周的第一天是星期日
SELECT TO_CHAR( DATE '1970-01-01', 'D' ) FROM DUAL;
```

输出 5

```
ALTER SESSION SET NLS_TERRITORY = 'UNITED KINGDOM'; -- 一周的第一天是星期一
SELECT TO_CHAR( DATE '1970-01-01', 'D' ) FROM DUAL;
```

输出 4

为了独立于NLS设置执行此操作，您可以将日期截断到当天午夜（以去除任何
天数的小数部分），然后减去截断到当前ISO周开始的日期（ISO周总是从
星期一）开始：

```
SELECT TRUNC( date_value ) - TRUNC( date_value, 'IW' ) + 1 FROM DUAL
```

```
FROM    DUAL;
```

If you run ALTER SESSION SET TIME_ZONE = '+01:00'; then the output is:

```
TIME
------------------------------------
2016-01-01 13:00:00.000000000 +01:00
```

and ALTER SESSION SET TIME_ZONE = 'PST'; then the output is:

```
TIME
------------------------------------
2016-01-01 04:00:00.000000000 PST
```

Section 6.11: Leap Seconds

Oracle does not handle leap seconds. See My Oracle Support note 2019397.2 and 730795.1 for more details.

Section 6.12: Getting the Day of the Week

You can use TO_CHAR(date_value, 'D') to get the day-of-week.

However, this is dependent on the NLS_TERRITORY session parameter:

```
ALTER SESSION SET NLS_TERRITORY = 'AMERICA';        -- First day of week is Sunday
SELECT TO_CHAR( DATE '1970-01-01', 'D' ) FROM DUAL;
```

Outputs 5

```
ALTER SESSION SET NLS_TERRITORY = 'UNITED KINGDOM'; -- First day of week is Monday
SELECT TO_CHAR( DATE '1970-01-01', 'D' ) FROM DUAL;
```

Outputs 4

To do this independent of the NLS settings, you can truncate the date to midnight of the current day (to remove any fractions of days) and subtract the date truncated to the start of the current iso-week (which always starts on Monday):

```
SELECT TRUNC( date_value ) - TRUNC( date_value, 'IW' ) + 1 FROM DUAL
```

第7章：日期处理

第7.1节：日期运算

Oracle支持DATE（包含精确到秒的时间）和TIMESTAMP（包含精确到秒的小数部分时间）数据类型，允许原生的加减运算。例如：

获取下一天：

```
SELECT TO_CHAR(SYSDATE + 1, 'YYYY-MM-DD') AS tomorrow FROM dual;
```

获取前一天：

```
SELECT TO_CHAR(SYSDATE - 1, 'YYYY-MM-DD') AS yesterday FROM dual;
```

将当前日期加5天：

```
SELECT TO_CHAR(SYSDATE + 5, 'YYYY-MM-DD') AS five_days_from_now FROM dual;
```

将当前日期加5小时：

```
SELECT TO_CHAR(SYSDATE + (5/24), 'YYYY-MM-DD HH24:MI:SS') AS five_hours_from_now FROM dual;
```

将当前日期加10分钟：

```
SELECT TO_CHAR(SYSDATE + (10/1440), 'YYYY-MM-DD HH24:MI:SS') AS ten_mintues_from_now FROM dual;
```

将当前日期加7秒：

```
    SELECT TO_CHAR(SYSDATE + (7/86400), 'YYYY-MM-DD HH24:MI:SS') AS seven_seconds_from_now FROM dual;
```

选择 hire_date 在30天前或更早的行：

```
SELECT * FROM emp WHERE hire_date < SYSDATE - 30;
```

选择last_updated列在最近一小时内的行：

```
SELECT * FROM logfile WHERE last_updated >= SYSDATE - (1/24);
```

Oracle 还提供了内置数据类型`INTERVAL`，表示一段时间（例如1.5天、36小时、2个月等）。这些也可以与`DATE`和`TIMESTAMP`表达式进行算术运算。例如：

```
SELECT * FROM logfile WHERE last_updated >= SYSDATE - INTERVAL '1' HOUR;
```

第7.2节：Add_months函数

语法：`ADD_MONTHS(p_date, INTEGER) RETURN DATE;`

Add_months函数将amt个月添加到p_date日期上。

```
SELECT ADD_MONTHS(DATE'2015-01-12', 2) m FROM dual;
```

M

Chapter 7: Working with Dates

Section 7.1: Date Arithmetic

Oracle supports `DATE` (includes time to the nearest second) and `TIMESTAMP` (includes time to fractions of a second) datatypes, which allow arithmetic (addition and subtraction) natively. For example:

To get the next day:

```
SELECT TO_CHAR(SYSDATE + 1, 'YYYY-MM-DD') AS tomorrow FROM dual;
```

To get the previous day:

```
SELECT TO_CHAR(SYSDATE - 1, 'YYYY-MM-DD') AS yesterday FROM dual;
```

To add 5 days to the current date:

```
SELECT TO_CHAR(SYSDATE + 5, 'YYYY-MM-DD') AS five_days_from_now FROM dual;
```

To add 5 hours to the current date:

```
SELECT TO_CHAR(SYSDATE + (5/24), 'YYYY-MM-DD HH24:MI:SS') AS five_hours_from_now FROM dual;
```

To add 10 minutes to the current date:

```
SELECT TO_CHAR(SYSDATE + (10/1440), 'YYYY-MM-DD HH24:MI:SS') AS ten_mintues_from_now FROM dual;
```

To add 7 seconds to the current date:

```
SELECT TO_CHAR(SYSDATE + (7/86400), 'YYYY-MM-DD HH24:MI:SS') AS seven_seconds_from_now FROM dual;
```

To select rows where `hire_date` is 30 days ago or more:

```
SELECT * FROM emp WHERE hire_date < SYSDATE - 30;
```

To select rows where `last_updated` column is in the last hour:

```
SELECT * FROM logfile WHERE last_updated >= SYSDATE - (1/24);
```

Oracle also provides the built-in datatype `INTERVAL` which represents a duration of time (e.g. 1.5 days, 36 hours, 2 months, etc.). These can also be used with arithmetic with `DATE` and `TIMESTAMP` expressions. For example:

```
SELECT * FROM logfile WHERE last_updated >= SYSDATE - INTERVAL '1' HOUR;
```

Section 7.2: Add_months function

Syntax: `ADD_MONTHS(p_date, INTEGER) RETURN DATE;`

Add_months function adds amt months to p_date date.

```
SELECT ADD_MONTHS(DATE'2015-01-12', 2) m FROM dual;
```

M

2015-03-12

您也可以使用负数amt来减去月份

```
SELECT ADD_MONTHS(DATE'2015-01-12', -2) m FROM dual;
```

M
2014-11-12

当计算出的月份天数少于给定日期时，将返回计算月份的最后一天。

```
SELECT TO_CHAR( ADD_MONTHS(DATE'2015-01-31', 1),'YYYY-MM-DD') m FROM dual;
```

M
2015-02-28

2015-03-12

You can also subtract months using a negative `amt`

```
SELECT ADD_MONTHS(DATE'2015-01-12', -2) m FROM dual;
```

M
2014-11-12

When the calculated month has fewer days as the given date, the last day of the calculated month will be returned.

```
SELECT TO_CHAR( ADD_MONTHS(DATE'2015-01-31', 1),'YYYY-MM-DD') m FROM dual;
```

M
2015-02-28

第8章：DUAL表

第8.1节：以下示例返回当前操作系统的日期和时间

```
SELECT SYSDATE FROM dual
```

第8.2节：以下示例生成start_value和end_value之间的数字

```
SELECT :start_value + LEVEL -1 n
FROM dual
CONNECT BY LEVEL <= :end_value  - :start_value + 1
```

Chapter 8: DUAL table

Section 8.1: The following example returns the current operating system date and time

```
SELECT SYSDATE FROM dual
```

Section 8.2: The following example generates numbers between start_value and end_value

```
SELECT :start_value + LEVEL -1 n
FROM dual
CONNECT BY LEVEL <= :end_value  - :start_value + 1
```

第9章：连接（JOINS）

第9.1节：交叉连接（CROSS JOIN）

交叉连接（CROSS JOIN）在两个表之间执行连接，不使用显式的连接子句，结果是两个表的笛卡尔积。笛卡尔积意味着一个表的每一行都会与另一个表的每一行组合。例如，如果TABLEA有20行，TABLEB有20行，结果将是20*20=

400输出行数。

示例：

```
SELECT *
FROM TABLEA CROSS JOIN TABLEB;
```

这也可以写成：

```
SELECT *
FROM TABLEA, TABLEB;
```

以下是两个表之间使用交叉连接的SQL示例：

示例表格： TABLEA

```
+-------+---------+
| VALUE |   NAME  |
+-------+---------+
|   1   |   ONE   |
|   2   |   TWO   |
+-------+---------+
```

示例表格： TABLEB

```
+-------+--------+
| VALUE |  NAME  |
+-------+--------+
|   3   |  THREE |
|   4   |  FOUR  |
+-------+--------+
```

现在，如果你执行查询：

```
SELECT *
FROM TABLEA CROSS JOIN TABLEB;
```

输出：

```
+-------+--------+-------+--------+
| VALUE |  NAME  | VALUE |  NAME  |
+-------+--------+-------+--------+
|   1   |   ONE  |   3   |  THREE |
|   1   |   ONE  |   4   |  FOUR  |
|   2   |   TWO  |   3   |  THREE |
|   2   |   TWO  |   4   |  FOUR  |
```

Chapter 9: JOINS

Section 9.1: CROSS JOIN

A CROSS JOIN performs a join between two tables that does not use an explicit join clause and results in the Cartesian product of two tables. A Cartesian product means each row of one table is combined with each row of the second table in the join. For example, if TABLEA has 20 rows and TABLEB has 20 rows, the result would be 20*20 = 400 output rows.

Example:

```
SELECT *
FROM TABLEA CROSS JOIN TABLEB;
```

This can also be written as:

```
SELECT *
FROM TABLEA, TABLEB;
```

Here's an example of cross join in SQL between two tables:

Sample Table: TABLEA

```
+-------+---------+
| VALUE |   NAME  |
+-------+---------+
|   1   |   ONE   |
|   2   |   TWO   |
+-------+---------+
```

Sample Table: TABLEB

```
+-------+--------+
| VALUE |  NAME  |
+-------+--------+
|   3   |  THREE |
|   4   |  FOUR  |
+-------+--------+
```

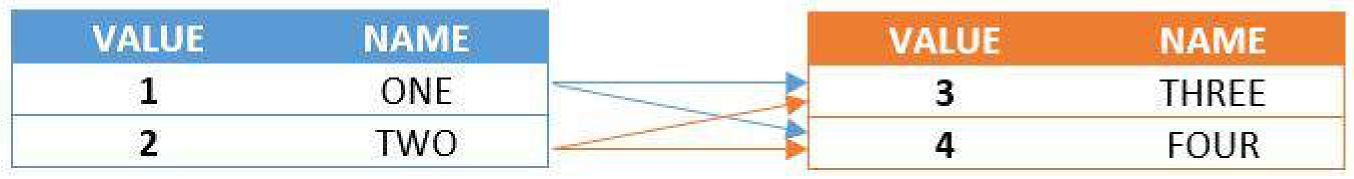
Now, If you execute the query:

```
SELECT *
FROM TABLEA CROSS JOIN TABLEB;
```

Output:

```
+-------+--------+-------+--------+
| VALUE |  NAME  | VALUE |  NAME  |
+-------+--------+-------+--------+
|   1   |   ONE  |   3   |  THREE |
|   1   |   ONE  |   4   |  FOUR  |
|   2   |   TWO  |   3   |  THREE |
|   2   |   TWO  |   4   |  FOUR  |
+--------+--------+-------+--------+
```

这就是两个表之间进行交叉连接的方式：



关于交叉连接的更多信息：Oracle 文档

第9.2节：左外连接（LEFT OUTER JOIN）

A左外连接执行两个表之间的连接，要求显式的连接条件，但不会排除第一个表中未匹配的行。

示例：

```
SELECT
       ENAME,
       DNAME,
       EMP.DEPTNO,
       DEPT.DEPTNO
FROM
       SCOTT.EMP LEFT OUTER JOIN SCOTT.DEPT
       ON EMP.DEPTNO = DEPT.DEPTNO;
```

尽管ANSI语法是推荐的方式，但很可能经常遇到遗留语法。在条件中使用`(+)`可以确定方程的哪一侧被视为*外连接*。

```
SELECT
       ENAME,
       DNAME,
       EMP.DEPTNO,
       DEPT.DEPTNO
FROM
       SCOTT.EMP,
SCOTT.DEPT
WHERE
EMP.DEPTNO = DEPT.DEPTNO(+);
```

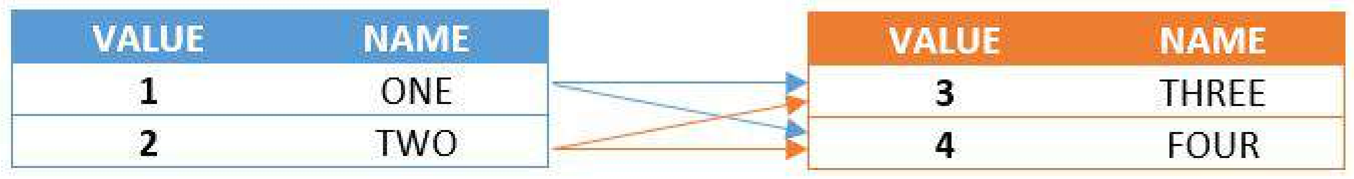
以下是两个表之间左外连接的示例：

示例表： EMPLOYEE（员工）

```
+-----------+---------+
|    NAME   |  DEPTNO |
+-----------+---------+
|     A     |    2    |
|     B     |    1    |
|     C     |    3    |
|     D     |    2    |
|     E     |    1    |
|     F     |    1    |
|     G     |    4    |
|     H     |    4    |
```

```
+--------+--------+-------+--------+
```

This is how cross joining happens between two tables:



More about Cross Join: Oracle documentation

Section 9.2: LEFT OUTER JOIN

A `LEFT OUTER JOIN` performs a join between two tables that requires an explicit join clause but does not exclude unmatched rows from the first table.

Example:

```
SELECT
       ENAME,
       DNAME,
       EMP.DEPTNO,
       DEPT.DEPTNO
FROM
       SCOTT.EMP LEFT OUTER JOIN SCOTT.DEPT
       ON EMP.DEPTNO = DEPT.DEPTNO;
```

Even though ANSI syntax is the recommended way, it is likely to encounter legacy syntax very often. Using `(+)` within a condition determines which side of the equation to be considered as *outer*.

```
SELECT
       ENAME,
       DNAME,
       EMP.DEPTNO,
       DEPT.DEPTNO
FROM
       SCOTT.EMP,
       SCOTT.DEPT
WHERE
       EMP.DEPTNO = DEPT.DEPTNO(+);
```

Here's an example of Left Outer Join between two tables:

Sample Table: EMPLOYEE

```
+-----------+---------+
|    NAME   |  DEPTNO |
+-----------+---------+
|     A     |    2    |
|     B     |    1    |
|     C     |    3    |
|     D     |    2    |
|     E     |    1    |
|     F     |    1    |
|     G     |    4    |
|     H     |    4    |
+-----------+---------+
```

示例表： DEPT（部门）

```
+---------+--------------+
|  部门编号 |   部门名称    |
+---------+--------------+
|    1    |  会计部       |
|    2    |  财务部       |
|    5    |  市场部       |
|    6    |  人力资源部   |
+---------+--------------+
```

现在，如果你执行查询：

```
SELECT
      *
FROM
EMPLOYEE LEFT OUTER JOIN DEPT
      ON EMPLOYEE.DEPTNO = DEPT.DEPTNO;
```

输出：

```
+-----------+---------+---------+--------------+
|    姓名    | 部门编号 | 部门编号 |   部门名称    |
+-----------+---------+---------+--------------+
|     F     |    1    |    1    |  会计部       |
|     E     |    1    |    1    |  会计部       |
|     B     |    1    |    1    |  会计部       |
|     D     |    2    |    2    |  财务部       |
|     A     |    2    |    2    |  财务部       |
|     C     |    3    |         |              |
|     H     |    4    |         |              |
|     G     |    4    |         |              |
+-----------+---------+---------+--------------+
```

第9.3节：右外连接

右外连接（RIGHT OUTER JOIN）在两个表之间执行连接，要求显式的连接条件，但不会排除第二个表中未匹配的行。

示例：

```
SELECT
      ENAME,
      DNAME,
      EMP.DEPTNO,
      DEPT.DEPTNO
FROM
      SCOTT.EMP RIGHT OUTER JOIN SCOTT.DEPT
      ON EMP.DEPTNO = DEPT.DEPTNO;
```

由于包含了 SCOTT.DEPT 中未匹配的行，但不包含 SCOTT.EMP 中未匹配的行，上述语句等价于使用 LEFT OUTER JOIN 的以下语句。

```
+-----------+---------+
```

Sample Table: DEPT

```
+---------+--------------+
|  DEPTNO |   DEPTNAME   |
+---------+--------------+
|    1    |  ACCOUNTING  |
|    2    |    FINANCE   |
|    5    |   MARKETING  |
|    6    |      HR      |
+---------+--------------+
```

Now, If you execute the query:

```
SELECT
      *
FROM
      EMPLOYEE LEFT OUTER JOIN DEPT
      ON EMPLOYEE.DEPTNO = DEPT.DEPTNO;
```

Output:

```
+-----------+---------+---------+--------------+
|    NAME   | DEPTNO  | DEPTNO  |   DEPTNAME   |
+-----------+---------+---------+--------------+
|     F     |    1    |    1    |  ACCOUNTING  |
|     E     |    1    |    1    |  ACCOUNTING  |
|     B     |    1    |    1    |  ACCOUNTING  |
|     D     |    2    |    2    |    FINANCE   |
|     A     |    2    |    2    |    FINANCE   |
|     C     |    3    |         |              |
|     H     |    4    |         |              |
|     G     |    4    |         |              |
+-----------+---------+---------+--------------+
```

Section 9.3: RIGHT OUTER JOIN

A `RIGHT OUTER JOIN` performs a join between two tables that requires an explicit join clause but does not exclude unmatched rows from the second table.

Example:

```
SELECT
      ENAME,
      DNAME,
      EMP.DEPTNO,
      DEPT.DEPTNO
FROM
      SCOTT.EMP RIGHT OUTER JOIN SCOTT.DEPT
      ON EMP.DEPTNO = DEPT.DEPTNO;
```

As the unmatched rows of `SCOTT.DEPT` are included, but unmatched rows of `SCOTT.EMP` are not, the above is equivalent to the following statement using `LEFT OUTER JOIN`.

```
SELECT
      ENAME,
      DNAME,
      EMP.DEPTNO,
      DEPT.DEPTNO
FROM
      SCOTT.DEPT RIGHT OUTER JOIN SCOTT.EMP
      ON DEPT.DEPTNO = EMP.DEPTNO;
```

下面是两个表之间右外连接的示例：

示例表： EMPLOYEE（员工）

```
+-----------+---------+
|    姓名    |  部门号 |
+-----------+---------+
|     A     |    2    |
|     B     |    1    |
|     C     |    3    |
|     D     |    2    |
|     E     |    1    |
|     F     |    1    |
|     G     |    4    |
|     H     |    4    |
+-----------+---------+
```

示例表： DEPT（部门）

```
+---------+--------------+
| 部门编号 |   部门名称    |
+---------+--------------+
|    1    |  会计部       |
|    2    |  财务部       |
|    5    |  市场部       |
|    6    |  人力资源部   |
+---------+--------------+
```

现在，如果你执行查询：

```
SELECT
      *
FROM
员工 右外连接 部门
      ON 员工.部门号 = 部门.部门号;
```

输出：

```
+-----------+---------+---------+--------------+
|    姓名    |  部门号 |  部门号  |   部门名称    |
+-----------+---------+---------+--------------+
|     A     |    2    |    2    |    财务部    |
|     B     |    1    |    1    |  会计部   |
|     D     |    2    |    2    |    财务部    |
|     E     |    1    |    1    |  会计部   |
|     F     |    1    |    1    |  会计部   |
|           |         |    5    |   市场部   |
|           |         |    6    |      人力资源部       |
```

```
SELECT
      ENAME,
      DNAME,
      EMP.DEPTNO,
      DEPT.DEPTNO
FROM
      SCOTT.DEPT RIGHT OUTER JOIN SCOTT.EMP
      ON DEPT.DEPTNO = EMP.DEPTNO;
```

Here's an example of Right Outer Join between two tables:

Sample Table: EMPLOYEE

```
+-----------+---------+
|    NAME   |  DEPTNO |
+-----------+---------+
|     A     |    2    |
|     B     |    1    |
|     C     |    3    |
|     D     |    2    |
|     E     |    1    |
|     F     |    1    |
|     G     |    4    |
|     H     |    4    |
+-----------+---------+
```

Sample Table: DEPT

```
+---------+--------------+
|  DEPTNO |   DEPTNAME   |
+---------+--------------+
|    1    |  ACCOUNTING  |
|    2    |    FINANCE   |
|    5    |   MARKETING  |
|    6    |      HR      |
+---------+--------------+
```

Now, If you execute the query:

```
SELECT
      *
FROM
      EMPLOYEE RIGHT OUTER JOIN DEPT
      ON EMPLOYEE.DEPTNO = DEPT.DEPTNO;
```

Output:

```
+-----------+---------+---------+-------------+
|    NAME   |  DEPTNO |  DEPTNO |   DEPTNAME  |
+-----------+---------+---------+-------------+
|     A     |    2    |    2    |    FINANCE  |
|     B     |    1    |    1    |  ACCOUNTING |
|     D     |    2    |    2    |    FINANCE  |
|     E     |    1    |    1    |  ACCOUNTING |
|     F     |    1    |    1    |  ACCOUNTING |
|           |         |    5    |   MARKETING |
|           |         |    6    |      HR     |
+-----------+---------+---------+--------------+
```

该查询的 Oracle (+) 语法等价为：

```
SELECT *
FROM 员工, 部门
WHERE 员工.部门号(+) = 部门.部门号;
```

第9.4节：全外连接

全外连接 在两个表之间执行连接，需要显式的连接条件，但不排除任一表中不匹配的行。换句话说，它返回每个表中的所有行。

示例：

```
SELECT
        *
FROM
        EMPLOYEE FULL OUTER JOIN DEPT
        ON EMPLOYEE.DEPTNO = DEPT.DEPTNO;
```

以下是两个表之间的全外连接示例：

示例表： EMPLOYEE（员工）

```
+-----------+---------+
|    姓名    |  部门号 |
+-----------+---------+
|     A     |    2    |
|     B     |    1    |
|     C     |    3    |
|     D     |    2    |
|     E     |    1    |
|     F     |    1    |
|     G     |    4    |
|     H     |    4    |
+-----------+---------+
```

示例表： DEPT（部门）

```
+---------+--------------+
| 部门编号 |    部门名称    |
+---------+--------------+
|    1    |  会计部       |
|    2    |  财务部       |
|    5    |  市场部       |
|    6    |  人力资源部    |
+---------+--------------+
```

现在，如果你执行查询：

```
SELECT
        *
FROM
EMPLOYEE FULL OUTER JOIN DEPT
```

```
+-----------+---------+---------+--------------+
```

Oracle (+) syntax equivalent for the query is:

```
SELECT *
FROM EMPLOYEE, DEPT
WHERE EMPLOYEE.DEPTNO(+) = DEPT.DEPTNO;
```

Section 9.4: FULL OUTER JOIN

A `FULL OUTER JOIN` performs a join between two tables that requires an explicit join clause but does not exclude unmatched rows in either table. In other words, it returns all the rows in each table.

Example:

```
SELECT
        *
FROM
        EMPLOYEE FULL OUTER JOIN DEPT
        ON EMPLOYEE.DEPTNO = DEPT.DEPTNO;
```

Here's an example of Full Outer Join between two tables:

Sample Table: EMPLOYEE

```
+-----------+---------+
|    NAME   |  DEPTNO |
+-----------+---------+
|     A     |    2    |
|     B     |    1    |
|     C     |    3    |
|     D     |    2    |
|     E     |    1    |
|     F     |    1    |
|     G     |    4    |
|     H     |    4    |
+-----------+---------+
```

Sample Table: DEPT

```
+---------+--------------+
|  DEPTNO |   DEPTNAME   |
+---------+--------------+
|    1    |  ACCOUNTING  |
|    2    |    FINANCE   |
|    5    |   MARKETING  |
|    6    |      HR      |
+---------+--------------+
```

Now, If you execute the query:

```
SELECT
        *
FROM
        EMPLOYEE FULL OUTER JOIN DEPT
```

```
    ON EMPLOYEE.DEPTNO = DEPT.DEPTNO;
```

输出

```
+-----------+---------+---------+--------------+
|    NAME   |  DEPTNO |  DEPTNO |   DEPTNAME   |
+-----------+---------+---------+--------------+
|     A     |    2    |    2    |    财务部    |
|     B     |    1    |    1    |  会计部      |
|     C     |    3    |         |              |
|     D     |    2    |    2    |    财务部    |
|     E     |    1    |    1    |  会计部      |
|     F     |    1    |    1    |  会计部      |
|     G     |    4    |         |              |
|     H     |    4    |         |              |
|           |         |    6    |      人力资源部     |
|           |         |    5    |  市场部      |
+-----------+---------+---------+--------------+
```

这里不匹配的列被保留为NULL。

第9.5节：反连接（ANTIJOIN）

反连接返回谓词左侧中在谓词右侧没有对应行的行。它返回那些未能匹配（NOT IN）右侧子查询的行。

```
SELECT * FROM employees
   WHERE department_id NOT IN
   (SELECT department_id FROM departments
       WHERE location_id = 1700)
   ORDER BY last_name;
```

以下是两个表之间反连接的示例：

示例表： EMPLOYEE（员工）

```
+-----------+---------+
|    姓名   |  部门号 |
+-----------+---------+
|     A     |    2    |
|     B     |    1    |
|     C     |    3    |
|     D     |    2    |
|     E     |    1    |
|     F     |    1    |
|     G     |    4    |
|     H     |    4    |
+-----------+---------+
```

示例表： DEPT（部门）

```
|  DEPTNO |   DEPTNAME    |
+---------+---------------+
|    1    |  会计部       |
|    2    |  财务部       |
```

```
    ON EMPLOYEE.DEPTNO = DEPT.DEPTNO;
```

Output

```
+-----------+---------+---------+--------------+
|    NAME   |  DEPTNO |  DEPTNO |   DEPTNAME   |
+-----------+---------+---------+--------------+
|     A     |    2    |    2    |    FINANCE   |
|     B     |    1    |    1    |  ACCOUNTING  |
|     C     |    3    |         |              |
|     D     |    2    |    2    |    FINANCE   |
|     E     |    1    |    1    |  ACCOUNTING  |
|     F     |    1    |    1    |  ACCOUNTING  |
|     G     |    4    |         |              |
|     H     |    4    |         |              |
|           |         |    6    |      HR      |
|           |         |    5    |   MARKETING  |
+-----------+---------+---------+--------------+
```

Here the columns that do not match has been kept NULL.

Section 9.5: ANTIJOIN

An antijoin returns rows from the left side of the predicate for which there are no corresponding rows on the right side of the predicate. It returns rows that fail to match (NOT IN) the subquery on the right side.

```
SELECT * FROM employees
   WHERE department_id NOT IN
   (SELECT department_id FROM departments
       WHERE location_id = 1700)
   ORDER BY last_name;
```

Here's an example of Anti Join between two tables:

Sample Table: EMPLOYEE

```
+-----------+---------+
|    NAME   |  DEPTNO |
+-----------+---------+
|     A     |    2    |
|     B     |    1    |
|     C     |    3    |
|     D     |    2    |
|     E     |    1    |
|     F     |    1    |
|     G     |    4    |
|     H     |    4    |
+-----------+---------+
```

Sample Table: DEPT

```
+---------+--------------+
|  DEPTNO |   DEPTNAME   |
+---------+--------------+
|    1    |  ACCOUNTING  |
|    2    |    FINANCE   |
|   5   |  市场部     |
|   6   |  人力资源部  |
+---------+--------------+
```

现在，如果你执行查询：

```
SELECT
      *
FROM
EMPLOYEE WHERE DEPTNO NOT IN
      (SELECT DEPTNO FROM DEPT);
```

输出：

```
+-----------+---------+
|    姓名   |  部门编号 |
+-----------+---------+
|     C     |    3    |
|     H     |    4    |
|     G     |    4    |
+-----------+---------+
```

输出显示了EMPLOYEE表中部门编号不在DEPT表中的行。

第9.6节：内连接（INNER JOIN）

内连接是一种允许你指定显式连接条件的连接操作。

语法

TableExpression [INNER] JOIN TableExpression { ON booleanExpression | USING 子句 }

您可以通过指定带有布尔表达式的 ON 来指定连接子句。

ON 子句中表达式的作用域包括当前表以及当前 SELECT 的外部查询块中的任何表。在以下示例中，ON 子句引用的是当前表：

```
-- 连接 EMP_ACT 和 EMPLOYEE 表
-- 选择 EMP_ACT 表中的所有列，并
-- 将 EMPLOYEE 表中的员工姓氏（LASTNAME）
-- 添加到结果的每一行
SELECT SAMP.EMP_ACT.*, LASTNAME
 FROM SAMP.EMP_ACT JOIN SAMP.EMPLOYEE
 ON EMP_ACT.EMPNO = EMPLOYEE.EMPNO
-- 连接 EMPLOYEE 和 DEPARTMENT 表，
-- 选择员工编号（EMPNO），
-- 员工姓氏（LASTNAME），
-- 部门编号（EMPLOYEE 表中的 WORKDEPT 和
-- DEPARTMENT 表）
-- 以及部门名称（DEPTNAME）
-- 所有出生日期（BIRTHDATE）早于1930年的员工。
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
 FROM SAMP.EMPLOYEE JOIN SAMP.DEPARTMENT
 ON WORKDEPT = DEPTNO
 AND YEAR(BIRTHDATE) < 1930

-- 另一个"生成"新数据值的示例，
```

```
|    5    |   MARKETING  |
|    6    |      HR      |
+---------+--------------+
```

Now, If you execute the query:

```
SELECT
      *
FROM
      EMPLOYEE WHERE DEPTNO NOT IN
      (SELECT DEPTNO FROM DEPT);
```

Output:

```
+-----------+---------+
|    NAME   |  DEPTNO |
+-----------+---------+
|     C     |    3    |
|     H     |    4    |
|     G     |    4    |
+-----------+---------+
```

The output shows that only the rows of EMPLOYEE table, of which DEPTNO were not present in DEPT table.

Section 9.6: INNER JOIN

An INNER JOIN is a JOIN operation that allows you to specify an explicit join clause.

Syntax

TableExpression [INNER] JOIN TableExpression { ON booleanExpression | USING clause }

You can specify the join clause by specifying ON with a boolean expression.

The scope of expressions in the ON clause includes the current tables and any tables in outer query blocks to the current SELECT. In the following example, the ON clause refers to the current tables:

```
-- Join the EMP_ACT and EMPLOYEE tables
-- select all the columns from the EMP_ACT table and
-- add the employee's surname (LASTNAME) from the EMPLOYEE table
-- to each row of the result
SELECT SAMP.EMP_ACT.*, LASTNAME
 FROM SAMP.EMP_ACT JOIN SAMP.EMPLOYEE
 ON EMP_ACT.EMPNO = EMPLOYEE.EMPNO
-- Join the EMPLOYEE and DEPARTMENT tables,
-- select the employee number (EMPNO),
-- employee surname (LASTNAME),
-- department number (WORKDEPT in the EMPLOYEE table and DEPTNO in the
-- DEPARTMENT table)
-- and department name (DEPTNAME)
-- of all employees who were born (BIRTHDATE) earlier than 1930.
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
 FROM SAMP.EMPLOYEE JOIN SAMP.DEPARTMENT
 ON WORKDEPT = DEPTNO
 AND YEAR(BIRTHDATE) < 1930

-- Another example of "generating" new data values,
-- 使用从 VALUES 子句中选择的查询（这是一种
-- 完整选择的替代形式）。
-- 此查询展示了如何派生一个名为“X”的表
-- 该表有2列“R1”和“R2”，以及1行数据
SELECT *
FROM (VALUES (3, 4), (1, 5), (2, 6))
AS VALUESTABLE1(C1, C2)
JOIN (VALUES (3, 2), (1, 2),
(0, 3)) AS VALUESTABLE2(c1, c2)
ON VALUESTABLE1.c1 = VALUESTABLE2.c1
-- 结果是：
-- C1         |C2         |C1         |2
-- ----------------------------------------------
-- 3          |4          |3          |2
-- 1          |5          |1          |2


-- 列出每个部门及其员工人数和
-- 经理的姓氏

SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME
FROM DEPARTMENT INNER JOIN EMPLOYEE
ON MGRNO = EMPNO

-- 列出每个员工编号和姓氏
-- 包含其经理的员工编号和姓氏
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
FROM EMPLOYEE E INNER JOIN
DEPARTMENT INNER JOIN EMPLOYEE M
    ON MGRNO = M.EMPNO
    ON E.WORKDEPT = DEPTNO
```

第9.7节：JOIN（连接）

JOIN 操作在两个表之间执行连接，排除第一个表中任何不匹配的行。从Oracle 9i开始，JOIN 的功能等同于 INNER JOIN。此操作需要显式的连接子句，与 CROSS JOIN 和 NATURAL JOIN 操作符不同。

示例：

```
SELECT t1.*,
t2.DeptId
  FROM table_1 t1
  JOIN table_2 t2 ON t2.DeptNo = t1.DeptNo
```

Oracle 文档：

- 10g
- 11g
- 12g

第9.8节：半连接（SEMIJOIN）

半连接查询可以用于例如查找所有至少有一名员工薪水超过2500的部门。

```
SELECT * FROM departments
   WHERE EXISTS
```

```
-- using a query which selects from a VALUES clause (which is an
-- alternate form of a fullselect).
-- This query shows how a table can be derived called "X"
-- having 2 columns "R1" and "R2" and 1 row of data
SELECT *
FROM (VALUES (3, 4), (1, 5), (2, 6))
AS VALUESTABLE1(C1, C2)
JOIN (VALUES (3, 2), (1, 2),
(0, 3)) AS VALUESTABLE2(c1, c2)
ON VALUESTABLE1.c1 = VALUESTABLE2.c1
-- This results in:
-- C1         |C2         |C1         |2
-- ----------------------------------------------
-- 3          |4          |3          |2
-- 1          |5          |1          |2


-- List every department with the employee number and
-- last name of the manager

SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME
FROM DEPARTMENT INNER JOIN EMPLOYEE
ON MGRNO = EMPNO

-- List every employee number and last name
-- with the employee number and last name of their manager
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
FROM EMPLOYEE E INNER JOIN
DEPARTMENT INNER JOIN EMPLOYEE M
    ON MGRNO = M.EMPNO
    ON E.WORKDEPT = DEPTNO
```

Section 9.7: JOIN

The JOIN operation performs a join between two tables, excluding any unmatched rows from the first table. From Oracle 9i forward, the JOIN is equivalent in function to the INNER JOIN. This operation requires an explicit join clause, as opposed to the CROSS JOIN and NATURAL JOIN operators.

Example:

```
SELECT t1.*,
       t2.DeptId
  FROM table_1 t1
  JOIN table_2 t2 ON t2.DeptNo = t1.DeptNo
```

Oracle documentation:

- 10g
- 11g
- 12g

Section 9.8: SEMIJOIN

A semijoin query can be used, for example, to find all departments with at least one employee whose salary exceeds 2500.

```
SELECT * FROM departments
   WHERE EXISTS
   (SELECT 1 FROM employees
       WHERE departments.department_id = employees.department_id
       AND employees.salary > 2500)
   ORDER BY department_name;
```

这比完全连接的替代方案更高效，因为先对员工进行内连接，然后使用一个where子句限定工资必须大于2500，可能会多次返回同一个部门。比如，如果消防部门有 n 名员工，且工资均为3000，SELECT * FROM departments, employees 并在id上进行必要的连接以及我们的where子句将返回消防部门 n 次。

第9.9节：自然连接

自然连接不需要显式的连接条件；它基于连接表中所有同名字段自动构建连接条件。

```
CREATE TABLE tab1(id NUMBER,  descr VARCHAR2(100));
CREATE TABLE tab2(id NUMBER,  descr VARCHAR2(100));
INSERT INTO tab1 VALUES(1, 'one');
INSERT INTO tab1 VALUES(2, 'two');
INSERT INTO tab1 VALUES(3, 'three');
INSERT INTO tab2 VALUES(1, 'ONE');
INSERT INTO tab2 VALUES(3, 'three');
```

连接将基于两个表共有的字段ID和DESCR进行：

```
SQL> SELECT *
  2  FROM tab1
  3        NATURAL JOIN
  4       tab2;

ID DESCR
---------- ----------
3 three
```

具有不同名称的列将不会用于JOIN条件：

```
SQL> SELECT *
  2  FROM (SELECT id AS id, descr AS descr1 FROM tab1)
  3       NATURAL JOIN
  4       (SELECT id AS id, descr AS descr2 FROM tab2);

ID DESCR1     DESCR2
---------- ---------- ----------
1 one        ONE
3 three      three
```

如果连接的表没有公共列，则会执行无条件的JOIN：

```
SQL> SELECT *
  2  FROM (SELECT id AS id1, descr AS descr1 FROM tab1)
  3       NATURAL JOIN
  4       (SELECT id AS id2, descr AS descr2 FROM tab2);

ID1 DESCR1           ID2 DESCR2
---------- ---------- ---------- ----------
```

```
   (SELECT 1 FROM employees
       WHERE departments.department_id = employees.department_id
       AND employees.salary > 2500)
   ORDER BY department_name;
```

This is more efficient than the full join alternatives, as inner joining on employees then giving a where clause detailing that the salary has to be greater than 2500 could return the same department numerous times. Say if the Fire department has n employees all with salary 3000, `SELECT * FROM departments, employees` with the necessary join on ids and our where clause would return the Fire department n times.

Section 9.9: NATURAL JOIN

NATURAL JOIN requires no explitic join condition; it builds one based on all the fields with the same name in the joined tables.

```
CREATE TABLE tab1(id NUMBER,  descr VARCHAR2(100));
CREATE TABLE tab2(id NUMBER,  descr VARCHAR2(100));
INSERT INTO tab1 VALUES(1, 'one');
INSERT INTO tab1 VALUES(2, 'two');
INSERT INTO tab1 VALUES(3, 'three');
INSERT INTO tab2 VALUES(1, 'ONE');
INSERT INTO tab2 VALUES(3, 'three');
```

The join will be done on the fields ID and DESCR, common to both the tables:

```
SQL> SELECT *
  2  FROM tab1
  3        NATURAL JOIN
  4       tab2;

ID DESCR
---------- ----------
3 three
```

Columns with different names will not be used in the JOIN condition:

```
SQL> SELECT *
  2  FROM (SELECT id AS id, descr AS descr1 FROM tab1)
  3       NATURAL JOIN
  4       (SELECT id AS id, descr AS descr2 FROM tab2);

ID DESCR1     DESCR2
---------- ---------- ----------
1 one        ONE
3 three      three
```

If the joined tables have no common columns, a JOIN with no conditions will be done:

```
SQL> SELECT *
  2  FROM (SELECT id AS id1, descr AS descr1 FROM tab1)
  3       NATURAL JOIN
  4       (SELECT id AS id2, descr AS descr2 FROM tab2);

ID1 DESCR1           ID2 DESCR2
---------- ---------- ---------- ----------
```

```
1  one                   1  ONE
2  two                   1  ONE
3  three                1  ONE
1  one                   3  three
2  two                   3  three
3  three                3  three
```

```
1 one                        1 ONE
2 two                        1 ONE
3 three                      1 ONE
1 one                        3 three
2 two                        3 three
3 three                      3 three
```

第10章：处理NULL值

当某列没有值时，无论该列的数据类型如何，该列即为NULL。绝不应使用此语法 a = NULL来比较某列是否为NULL，因为结果将是UNKNOWN。应改用 a IS NULL或 a IS NOT NULL条件。NULL不等于NULL。要比较可能为null的两个表达式，请使用下面描述的函数之一。除连接运算符外，所有运算符如果其中一个操作数为NULL，则返回NULL。例如，3 * NULL + 5的结果是null。

第10.1节：包含NULL的运算结果为NULL，连接运算除外

```
SELECT 3 * NULL + 5, 'Hello ' || NULL || 'world'   FROM DUAL;
```

```
3*NULL+5 'HELLO'||NULL||'WORLD'
(null)        Hello world
```

第10.2节：NVL2函数用于根据值是否为null返回不同结果

如果第一个参数不为NULL，NVL2将返回第二个参数。否则，它将返回第三个参数。

```
SELECT NVL2(NULL, 'Foo', 'Bar'), NVL2(5, 'Foo', 'Bar') FROM DUAL;
```

```
NVL2(NULL,'FOO','BAR')  NVL2(5,'FOO','BAR')
Bar                     Foo
```

第10.3节：使用COALESCE返回第一个非NULL值

```
SELECT COALESCE(a, b, c, d, 5) FROM
    (SELECT NULL A, NULL b, NULL c, 4 d FROM DUAL);
```

```
COALESCE(A,B,C,D,5)
4
```

在某些情况下，当第二个参数不是常量时，使用两个参数的COALESCE比使用NVL更快。NVL总是会计算两个参数。COALESCE会在遇到第一个非NULL值时停止。这意味着如果第一个值非NULL，COALESCE会更快。

第10.4节：任何数据类型的列都可以包含NULL值

```
SELECT 1 NUM_COLUMN, 'foo' VARCHAR2_COLUMN FROM DUAL
UNION ALL
SELECT NULL, NULL FROM DUAL;
```

```
NUM_COLUMN VARCHAR2_COLUMN
1          foo
(null)     (null)
```

第10.5节：空字符串即为NULL

```
SELECT 1 a, '' b FROM DUAL;
```

```
A  B
1 (null)
```

Chapter 10: Handling NULL values

A column is NULL when it has no value, regardless of the data type of that column. A column should never be compared to NULL using this syntax `a = NULL` as the result would be UNKNOWN. Instead use `a IS NULL` or `a IS NOT NULL` conditions. NULL is not equal to NULL. To compare two expressions where null can happen, use one of the functions described below. All operators except concatenation return NULL if one of their operand is NULL. For instance the result of `3 * NULL + 5` is null.

Section 10.1: Operations containing NULL are NULL, except concatenation

```
SELECT 3 * NULL + 5, 'Hello ' || NULL || 'world'   FROM DUAL;
```

```
3*NULL+5 'HELLO'||NULL||'WORLD'
(null)        Hello world
```

Section 10.2: NVL2 to get a different result if a value is null or not

If the first parameter is NOT NULL, NVL2 will return the second parameter. Otherwise it will return the third one.

```
SELECT NVL2(NULL, 'Foo', 'Bar'), NVL2(5, 'Foo', 'Bar') FROM DUAL;
```

```
NVL2(NULL,'FOO','BAR') NVL2(5,'FOO','BAR')
Bar                    Foo
```

Section 10.3: COALESCE to return the first non-NULL value

```
SELECT COALESCE(a, b, c, d, 5) FROM
    (SELECT NULL A, NULL b, NULL c, 4 d FROM DUAL);
```

```
COALESCE(A,B,C,D,5)
4
```

In some case, using COALESCE with two parameters can be faster than using NVL when the second parameter is not a constant. NVL will always evaluate both parameters. COALESCE will stop at the first non-NULL value it encounters. It means that if the first value is non-NULL, COALESCE will be faster.

Section 10.4: Columns of any data type can contain NULLs

```
SELECT 1 NUM_COLUMN, 'foo' VARCHAR2_COLUMN FROM DUAL
UNION ALL
SELECT NULL, NULL FROM DUAL;
```

```
NUM_COLUMN VARCHAR2_COLUMN
1          foo
(null)     (null)
```

Section 10.5: Empty strings are NULL

```
SELECT 1 a, '' b FROM DUAL;
```

```
A  B
1 (null)
```

第10.6节：使用NVL替换空值

```
SELECT a column_with_null, NVL(a, 'N/A') column_without_null FROM
  (SELECT NULL a FROM DUAL);
```

```
COLUMN_WITH_NULL  COLUMN_WITHOUT_NULL
(null)            N/A
```

NVL 对比含有 NULL 的两个值时非常有用：

```
SELECT
    CASE WHEN a = b THEN 1 WHEN a <> b THEN 0 ELSE -1 END comparison_without_nvl,
    CASE WHEN NVL(a, -1) = NVL(b, -1) THEN 1 WHEN NVL(a, -1) <> NVL(b, -1) THEN 0 ELSE -1 END
comparison_with_nvl
  FROM
    (SELECT NULL a, 3 b FROM DUAL
     UNION ALL
     SELECT NULL, NULL FROM DUAL);
```

```
COMPARISON_WITHOUT_NVL  COMPARISON_WITH_NVL
-1                      0
-1                      1
```

Section 10.6: NVL to replace null value

```
SELECT a column_with_null, NVL(a, 'N/A') column_without_null FROM
  (SELECT NULL a FROM DUAL);
```

```
COLUMN_WITH_NULL COLUMN_WITHOUT_NULL
(null)           N/A
```

NVL is useful to compare two values which can contain NULLs :

```
SELECT
    CASE WHEN a = b THEN 1 WHEN a <> b THEN 0 ELSE -1 END comparison_without_nvl,
    CASE WHEN NVL(a, -1) = NVL(b, -1) THEN 1 WHEN NVL(a, -1) <> NVL(b, -1) THEN 0 ELSE -1 END
comparison_with_nvl
  FROM
    (SELECT NULL a, 3 b FROM DUAL
     UNION ALL
     SELECT NULL, NULL FROM DUAL);
```

```
COMPARISON_WITHOUT_NVL COMPARISON_WITH_NVL
-1                     0
-1                     1
```

第11章：字符串操作

第11.1节：INITCAP

INITCAP函数将字符串的大小写转换为每个单词首字母大写，其余字母小写。

```
SELECT INITCAP('HELLO mr macdonald!') AS NEW FROM dual;
```

输出

```
NEW
-------------------
Hello Mr Macdonald!
```

第11.2节：正则表达式

假设我们只想替换两位数：正则表达式将用 (\d\d) 找到它们

```
SELECT REGEXP_REPLACE ('2, 5, and 10 are numbers in this example', '(\d\d)', '#')
FROM dual;
```

结果为：

```
'2, 5, and # are numbers in this example'
```

如果我想交换文本的部分内容，我使用 \1, \2, \3 来调用匹配的字符串：

```
 SELECT REGEXP_REPLACE ('swap around 10 in that one ', '(.*)(\d\d )(.*)', '\3\2\1\3')
 FROM dual;
```

第11.3节：SUBSTR

SUBSTR通过指定起始位置和要提取的字符数来检索字符串的一部分

```
SELECT SUBSTR('abcdefg',2,3) FROM DUAL;
```

返回：

```
bcd
```

要从字符串末尾开始计数，SUBSTR接受第二个参数为负数，例如

```
SELECT SUBSTR('abcdefg',-4,2) FROM DUAL;
```

返回：

```
de
```

获取字符串中的最后一个字符：SUBSTR(mystring,-1,1)

Chapter 11: String Manipulation

Section 11.1: INITCAP

The INITCAP function converts the case of a string so that each word starts with a capital letter and all subsequent letters are in lowercase.

```
SELECT INITCAP('HELLO mr macdonald!') AS NEW FROM dual;
```

Output

```
NEW
-------------------
Hello Mr Macdonald!
```

Section 11.2: Regular expression

Let's say we want to replace only numbers with 2 digits: regular expression will find them with (\d\d)

```
SELECT REGEXP_REPLACE ('2, 5, and 10 are numbers in this example', '(\d\d)', '#')
FROM dual;
```

Results in:

```
'2, 5, and # are numbers in this example'
```

If I want to swap parts of the text, I use \1, \2, \3 to call for the matched strings:

```
 SELECT REGEXP_REPLACE ('swap around 10 in that one ', '(.*)(\d\d )(.*)', '\3\2\1\3')
 FROM dual;
```

Section 11.3: SUBSTR

SUBSTR retrieves part of a string by indicating the starting position and the number of characters to extract

```
SELECT SUBSTR('abcdefg',2,3) FROM DUAL;
```

returns:

```
bcd
```

To count from the end of the string, SUBSTR accepts a negative number as the second parameter, e.g.

```
SELECT SUBSTR('abcdefg',-4,2) FROM DUAL;
```

returns:

```
de
```

To get the last character in a string: SUBSTR(mystring,-1,1)

第11.4节：连接：操作符 || 或 concat() 函数

Oracle SQL 和 PL/SQL 中的 || 运算符允许你将两个或多个字符串连接在一起。

示例：

假设有以下customers表：

```
id  firstname    lastname
--- -----------  ----------
1  Thomas        Woody
```

查询：

```
SELECT firstname || ' ' || lastname || ' 在我的数据库中。' AS "我的句子"
  FROM customers;
```

输出：

```
我的句子
---------------------------------
Thomas Woody 在我的数据库中。
```

Oracle 也支持标准 SQL CONCAT(str1, str2) 函数：

示例：

查询：

```
SELECT CONCAT(firstname, ' 在我的数据库中。') FROM customers;
```

输出：

```
Expr1
---------------------------------
Thomas 在我的数据库中。
```

第11.5节：UPPER

UPPER函数允许您将字符串中的所有小写字母转换为大写字母。

```
SELECT UPPER('My text 123!') AS result FROM dual;
```

输出：

```
结果
------------
MY TEXT 123!
```

Section 11.4: Concatenation: Operator || or concat() function

The Oracle SQL and PL/SQL || operator allows you to concatenate 2 or more strings together.

Example:

Assuming the following `customers` table:

```
 id  firstname    lastname
---  -----------  ----------
1  Thomas       Woody
```

Query:

```
SELECT firstname || ' ' || lastname || ' is in my database.' AS "My Sentence"
  FROM customers;
```

Output:

```
My Sentence
---------------------------------
Thomas Woody is in my database.
```

Oracle also supports the standard SQL `CONCAT(str1, str2)` function:

Example:

Query:

```
SELECT CONCAT(firstname, ' is in my database.') FROM customers;
```

Output:

```
Expr1
---------------------------------
Thomas is in my database.
```

Section 11.5: UPPER

The UPPER function allows you to convert all lowercase letters in a string to uppercase.

```
SELECT UPPER('My text 123!') AS result FROM dual;
```

Output:

```
RESULT
------------
MY TEXT 123!
```

第11.6节：LOWER

LOWER将字符串中的所有大写字母转换为小写字母。

```
SELECT  LOWER('HELLO  World123!')  text  FROM  dual;
```

输出结果：

```
     text
hello world123!
```

第11.7节：LTRIM ／ RTRIM

LTRIM 和 RTRIM 分别从字符串的开头或结尾移除字符。可以提供一个或多个字符集合（默认是空格）进行移除。

例如，

```
SELECT LTRIM('<===>HELLO<===>', '=<>')
      ,RTRIM('<===>HELLO<===>', '=<>')
FROM dual;
```

返回：

```
HELLO<===>
<===>HELLO
```

Section 11.6: LOWER

LOWER converts all uppercase letters in a string to lowercase.

```
SELECT LOWER('HELLO World123!') text FROM dual;
```

Outputs:

```
     text
hello world123!
```

Section 11.7: LTRIM / RTRIM

LTRIM and RTRIM remove characters from the beginning or the end (respectively) of a string. A set of one or more characters may be supplied (default is a space) to remove.

For example,

```
SELECT LTRIM('<===>HELLO<===>', '=<>')
      ,RTRIM('<===>HELLO<===>', '=<>')
FROM dual;
```

Returns:

```
HELLO<===>
<===>HELLO
```

第12章：IF-THEN-ELSE语句

第12.1节：IF-THEN

```
声明
v_num1 NUMBER(10);
v_num2 NUMBER(10);

BEGIN
v_num1 := 2;
  v_num2 := 1;

  IF v_num1 > v_num2 THEN
     DBMS_OUTPUT.put_line('v_num1 大于 v_num2');
  END IF;
END;
```

第12.2节：IF-THEN-ELSE

```
声明
v_num1 NUMBER(10);
v_num2 NUMBER(10);

BEGIN
v_num1 := 2;
  v_num2 := 10;

  IF v_num1 > v_num2 THEN
     DBMS_OUTPUT.put_line('v_num1 大于 v_num2');
  ELSE
    DBMS_OUTPUT.put_line('v_num1 不大于 v_num2');
  END IF;
END;
```

第12.3节：IF-THEN-ELSIF-ELSE

```
声明
v_num1 NUMBER(10);
v_num2 NUMBER(10);

BEGIN
v_num1 := 2;
  v_num2 := 2;

  IF v_num1 > v_num2 THEN
     DBMS_OUTPUT.put_line('v_num1 大于 v_num2');
  ELSIF v_num1 < v_num2 THEN
    DBMS_OUTPUT.put_line('v_num1 不大于 v_num2');
  ELSE
    DBMS_OUTPUT.put_line('v_num1 等于 v_num2');
  END IF;
END;
```

Chapter 12: IF-THEN-ELSE Statement

Section 12.1: IF-THEN

```
DECLARE
v_num1 NUMBER(10);
v_num2 NUMBER(10);

BEGIN
  v_num1 := 2;
  v_num2 := 1;

  IF v_num1 > v_num2 THEN
     DBMS_OUTPUT.put_line('v_num1 is bigger than v_num2');
  END IF;
END;
```

Section 12.2: IF-THEN-ELSE

```
DECLARE
v_num1 NUMBER(10);
v_num2 NUMBER(10);

BEGIN
  v_num1 := 2;
  v_num2 := 10;

  IF v_num1 > v_num2 THEN
     DBMS_OUTPUT.put_line('v_num1 is bigger than v_num2');
  ELSE
    DBMS_OUTPUT.put_line('v_num1 is NOT bigger than v_num2');
  END IF;
END;
```

Section 12.3: IF-THEN-ELSIF-ELSE

```
DECLARE
v_num1 NUMBER(10);
v_num2 NUMBER(10);

BEGIN
  v_num1 := 2;
  v_num2 := 2;

  IF v_num1 > v_num2 THEN
     DBMS_OUTPUT.put_line('v_num1 is bigger than v_num2');
  ELSIF v_num1 < v_num2 THEN
    DBMS_OUTPUT.put_line('v_num1 is NOT bigger than v_num2');
  ELSE
    DBMS_OUTPUT.put_line('v_num1 is EQUAL to v_num2');
  END IF;
END;
```

第13章：限制查询返回的行数（分页）

第13.1节：使用行限制子句获取前N行

Oracle 12c R1 引入了 FETCH 子句：

```
SELECT   val
FROM     mytable
ORDER BY val DESC
FETCH FIRST 5 ROWS ONLY;
```

一个不使用 FETCH 的示例，也适用于早期版本：

```
SELECT * FROM (
   SELECT   val
   FROM     mytable
   ORDER BY val DESC
) WHERE ROWNUM <=5;
```

第 13.2 节：从多行中获取第 N 行到第 M 行（Oracle 12c 之前）

使用分析函数 row_number()：

```
WITH t AS (
  SELECT col1
  , col2
  , ROW_NUMBER() over (ORDER BY col1, col2) rn
  FROM TABLE
)
SELECT col1
, col2
FROM t
WHERE rn BETWEEN N AND M; -- N 和 M 都包含在内
```

Oracle 12c 使用 OFFSET 和 FETCH 更容易处理这个问题。

第13.3节：从表中获取N条记录

我们可以使用 rownum 子句限制结果中的行数

```
SELECT * FROM
(
  SELECT val FROM  mytable
) WHERE rownum<=5
```

如果我们想要第一条或最后一条记录，则需要在内层查询中使用 order by 子句，这样才能基于排序返回结果。

最近五条记录：

```
SELECT * FROM
(
    SELECT val FROM  mytable ORDER BY val DESC
```

Chapter 13: Limiting the rows returned by a query (Pagination)

Section 13.1: Get first N rows with row limiting clause

The FETCH clause was introduced in Oracle 12c R1:

```
SELECT   val
FROM     mytable
ORDER BY val DESC
FETCH FIRST 5 ROWS ONLY;
```

An example without FETCH that works also in earlier versions:

```
SELECT * FROM (
   SELECT   val
   FROM     mytable
   ORDER BY val DESC
) WHERE ROWNUM <= 5;
```

Section 13.2: Get row N through M from many rows (before Oracle 12c)

Use the analytical function row_number():

```
WITH t AS (
  SELECT col1
  , col2
  , ROW_NUMBER() over (ORDER BY col1, col2) rn
  FROM TABLE
)
SELECT col1
, col2
FROM t
WHERE rn BETWEEN N AND M; -- N and M are both inclusive
```

Oracle 12c handles this more easily with OFFSET and FETCH.

Section 13.3: Get N numbers of Records from table

We can limit no of rows from result using rownum clause

```
SELECT * FROM
(
  SELECT val FROM  mytable
) WHERE rownum<=5
```

If we want first or last record then we want order by clause in inner query that will give result based on order.

Last Five Record :

```
SELECT * FROM
(
    SELECT val FROM  mytable ORDER BY val DESC
) WHERE rownum<=5
```

前五条记录

```
SELECT * FROM
(
    SELECT val FROM  mytable ORDER BY val
) WHERE rownum<=5
```

第13.4节：跳过一些行然后取一些行

在Oracle 12g及以上版本

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 ROWS FETCH NEXT 20 ROWS ONLY;
```

在早期版本中

```
SELECT Id,
   Col1
 FROM (SELECT Id,
Col1,
           ROW_NUMBER() over (ORDER BY Id) RowNumber
      FROM TableName)
WHERE RowNumber BETWEEN 21 AND 40
```

第13.5节：跳过结果中的某些行

在Oracle 12g及以上版本

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 5 ROWS;
```

在早期版本中

```
SELECT Id,
   Col1
 FROM (SELECT Id,
Col1,
           ROW_NUMBER() over (ORDER BY Id) RowNumber
      FROM TableName)
WHERE RowNumber > 20
```

第13.6节：SQL中的分页

```
SELECT val
FROM   (SELECT val, ROWNUM AS rnum
        FROM   (SELECT val
                FROM   rownum_order_test
                ORDER BY val)
        WHERE ROWNUM <= :upper_limit)
WHERE  rnum >= :lower_limit ;
```

```
) WHERE rownum<=5
```

First Five Record

```
SELECT * FROM
(
    SELECT val FROM  mytable ORDER BY val
) WHERE rownum<=5
```

Section 13.4: Skipping some rows then taking some

In Oracle 12g+

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 ROWS FETCH NEXT 20 ROWS ONLY;
```

In earlier Versions

```
SELECT Id,
   Col1
 FROM (SELECT Id,
              Col1,
              ROW_NUMBER() over (ORDER BY Id) RowNumber
      FROM TableName)
WHERE RowNumber BETWEEN 21 AND 40
```

Section 13.5: Skipping some rows from result

In Oracle 12g+

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 5 ROWS;
```

In earlier Versions

```
SELECT Id,
   Col1
 FROM (SELECT Id,
              Col1,
              ROW_NUMBER() over (ORDER BY Id) RowNumber
      FROM TableName)
WHERE RowNumber > 20
```

Section 13.6: Pagination in SQL

```
SELECT val
FROM   (SELECT val, ROWNUM AS rnum
        FROM   (SELECT val
                FROM   rownum_order_test
                ORDER BY val)
        WHERE ROWNUM <= :upper_limit)
WHERE  rnum >= :lower_limit ;
```

通过这种方式，我们可以对表数据进行分页，就像网页搜索页面一样

this way we can paginate the table data , just like web serch page

第14章：使用WITH子句的递归子查询分解（又名公共表表达式）

第14.1节：拆分分隔字符串

示例数据：

```
CREATE TABLE table_name ( VALUE VARCHAR2(50) );

INSERT INTO table_name ( VALUE ) VALUES ( 'A,B,C,D,E' );
```

查询:

```
WITH items ( list, item, lvl ) AS (
  SELECT VALUE,
REGEXP_SUBSTR( VALUE, '[^,]+', 1, 1 ),
         1
  FROM   table_name
UNION ALL
  SELECT VALUE,
REGEXP_SUBSTR( VALUE, '[^,]+', 1, lvl + 1 ),
         lvl + 1
  FROM   items
  WHERE  lvl < REGEXP_COUNT( VALUE, '[^,]+' )
)
SELECT * FROM items;
```

输出:

```
列表      项目级别
--------- ---- ---
A,B,C,D,E    A   1
A,B,C,D,E    B   2
A,B,C,D,E    C   3
A,B,C,D,E    D   4
A,B,C,D,E    E   5
```

第14.2节：一个简单的整数生成器

查询:

```
WITH generator ( VALUE ) AS (
  SELECT 1 FROM DUAL
UNION ALL
  SELECT VALUE + 1
  FROM   generator
  WHERE  VALUE < 10
)
SELECT VALUE
FROM   generator;
```

输出:

Chapter 14: Recursive Sub-Query Factoring using the WITH Clause (A.K.A. Common Table Expressions)

Section 14.1: Splitting a Delimited String

Sample Data:

```
CREATE TABLE table_name ( VALUE VARCHAR2(50) );

INSERT INTO table_name ( VALUE ) VALUES ( 'A,B,C,D,E' );
```

Query:

```
WITH items ( list, item, lvl ) AS (
  SELECT VALUE,
         REGEXP_SUBSTR( VALUE, '[^,]+', 1, 1 ),
         1
  FROM   table_name
UNION ALL
  SELECT VALUE,
         REGEXP_SUBSTR( VALUE, '[^,]+', 1, lvl + 1 ),
         lvl + 1
  FROM   items
  WHERE  lvl < REGEXP_COUNT( VALUE, '[^,]+' )
)
SELECT * FROM items;
```

Output:

```
LIST      ITEM LVL
--------- ---- ---
A,B,C,D,E    A   1
A,B,C,D,E    B   2
A,B,C,D,E    C   3
A,B,C,D,E    D   4
A,B,C,D,E    E   5
```

Section 14.2: A Simple Integer Generator

Query:

```
WITH generator ( VALUE ) AS (
  SELECT 1 FROM DUAL
UNION ALL
  SELECT VALUE + 1
  FROM   generator
  WHERE  VALUE < 10
)
SELECT VALUE
FROM   generator;
```

Output:

```
VALUE
-----
1
10
```

```
VALUE
-----
1
2
3
4
5
6
7
8
9
10
```

第15章：更新记录的不同方法

第15.1节：使用合并进行更新

使用合并

```
合并到
      TESTTABLE
使用
      (选择
T1.ROWID 作为 RID,
          T2.TESTTABLE_ID

TESTTABLE T1
          内连接 JOIN
                MASTERTABLE T2
            ON TESTTABLE.TESTTABLE_ID = MASTERTABLE.TESTTABLE_ID
       WHERE ID_NUMBER=11)
ON
      ( ROWID = RID )
当匹配时
然后
    更新 设置 TEST_COLUMN= 'Testvalue';
```

第15.2节：带示例的更新语法

普通更新

```
更新
TESTTABLE
设置
TEST_COLUMN= 'Testvalue',TEST_COLUMN2= 123
条件
      存在
          (选择 MASTERTABLE.TESTTABLE_ID
           来自 MASTERTABLE
           条件 ID_NUMBER=11);
```

第15.3节：使用内联视图进行更新

使用内联视图（如果Oracle认为可更新）

注意：如果遇到非键保持行错误，请添加索引以解决该问题，使其可更新

```
更新
      (选择
TESTTABLE.TEST_COLUMN 作为 OLD,
            'Testvalue' 作为 NEW
       FROM
                TESTTABLE
内连接JOIN
                MASTERTABLE
            ON TESTTABLE.TESTTABLE_ID = MASTERTABLE.TESTTABLE_ID
       WHERE ID_NUMBER=11) T
设置
```

Chapter 15: Different ways to update records

Section 15.1: Update using Merge

Using Merge

```
MERGE INTO
      TESTTABLE
USING
      (SELECT
            T1.ROWID AS RID,
            T2.TESTTABLE_ID
       FROM
                TESTTABLE T1
            INNER JOIN
                MASTERTABLE T2
            ON TESTTABLE.TESTTABLE_ID = MASTERTABLE.TESTTABLE_ID
       WHERE ID_NUMBER=11)
ON
      ( ROWID = RID )
WHEN MATCHED
THEN
    UPDATE SET TEST_COLUMN= 'Testvalue';
```

Section 15.2: Update Syntax with example

Normal Update

```
UPDATE
      TESTTABLE
SET
      TEST_COLUMN= 'Testvalue',TEST_COLUMN2= 123
WHERE
      EXISTS
          (SELECT MASTERTABLE.TESTTABLE_ID
           FROM MASTERTABLE
           WHERE ID_NUMBER=11);
```

Section 15.3: Update Using Inline View

Using Inline View (If it is considered updateable by Oracle)

Note: If you face a non key preserved row error add an index to resolve the same to make it update-able

```
UPDATE
      (SELECT
            TESTTABLE.TEST_COLUMN AS OLD,
            'Testvalue' AS NEW
       FROM
                TESTTABLE
            INNER JOIN
                MASTERTABLE
            ON TESTTABLE.TESTTABLE_ID = MASTERTABLE.TESTTABLE_ID
       WHERE ID_NUMBER=11) T
SET
```

```
T.OLD         = T.NEW;
```

第15.4节：使用示例数据合并

```
DROP TABLE table01;
DROP TABLE table02;

CREATE TABLE table01 (
       code int,
name VARCHAR(50),
      old int
);

CREATE TABLE table02 (
       code int,
name VARCHAR(50),
      old int
);

清空表TABLE table01;
插入数据到 table01 VALUES (1, 'A', 10);
插入数据到 table01 VALUES (9, 'B', 12);
插入数据到 table01 VALUES (3, 'C', 14);
插入数据到 table01 VALUES (4, 'D', 16);
插入数据到 table01 VALUES (5, 'E', 18);

清空表TABLE table02;
插入数据到 table02 VALUES (1, 'AA', NULL);
插入数据到 table02 VALUES (2, 'BB', 123);
插入数据到 table02 VALUES (3, 'CC', NULL);
插入数据到 table02 VALUES (4, 'DD', NULL);
插入数据到 table02 VALUES (5, 'EE', NULL);

从 table01 a 选择所有按2排序;
从 table02 a 选择所有按2排序;

--

合并到 table02 a 使用 (
      选择 b.code, b.old 从 table01 b
) c 基于 (
a.code = c.code
)
当匹配时 更新设置 a.old = c.old
;

--

选择 a.*, b.* 从 table01 a
内连接 table02 b 基于 a.code = b.codetable01;

选择所有从 table01 a
WHERE
       存在
       (
         SELECT 'x' FROM table02 b WHERE a.code = b.codetable01
       );

SELECT * FROM table01 a WHERE a.code IN (SELECT b.codetable01 FROM table02 b);

--
```

```
       T.OLD        = T.NEW;
```

Section 15.4: Merge with sample data

```
DROP TABLE table01;
DROP TABLE table02;

CREATE TABLE table01 (
       code int,
       name VARCHAR(50),
       old int
);

CREATE TABLE table02 (
       code int,
       name VARCHAR(50),
       old int
);

truncate TABLE table01;
INSERT INTO table01 VALUES (1, 'A', 10);
INSERT INTO table01 VALUES (9, 'B', 12);
INSERT INTO table01 VALUES (3, 'C', 14);
INSERT INTO table01 VALUES (4, 'D', 16);
INSERT INTO table01 VALUES (5, 'E', 18);

truncate TABLE table02;
INSERT INTO table02 VALUES (1, 'AA', NULL);
INSERT INTO table02 VALUES (2, 'BB', 123);
INSERT INTO table02 VALUES (3, 'CC', NULL);
INSERT INTO table02 VALUES (4, 'DD', NULL);
INSERT INTO table02 VALUES (5, 'EE', NULL);

SELECT * FROM table01 a ORDER BY 2;
SELECT * FROM table02 a ORDER BY 2;

--

MERGE INTO table02 a USING (
      SELECT b.code, b.old FROM table01 b
) c ON (
  a.code = c.code
)
WHEN matched THEN UPDATE SET a.old = c.old
;

--

SELECT a.*, b.* FROM table01 a
inner JOIN table02 b ON a.code = b.codetable01;

SELECT * FROM table01 a
WHERE
       EXISTS
       (
         SELECT 'x' FROM table02 b WHERE a.code = b.codetable01
       );

SELECT * FROM table01 a WHERE a.code IN (SELECT b.codetable01 FROM table02 b);

--
```

```
SELECT * FROM table01 a
WHERE
       NOT EXISTS
       (
         SELECT 'x' FROM table02 b WHERE a.code = b.codetable01
       );

SELECT * FROM table01 a WHERE a.code NOT IN (SELECT b.codetable01 FROM table02 b);
```

```
SELECT * FROM table01 a
WHERE
       NOT EXISTS
       (
         SELECT 'x' FROM table02 b WHERE a.code = b.codetable01
       );

SELECT * FROM table01 a WHERE a.code NOT IN (SELECT b.codetable01 FROM table02 b);
```

第16章：使用连接的更新

与普遍误解（包括在Stack Overflow上）相反，Oracle允许通过连接进行更新。然而，有一些（相当合理的）要求。我们通过一个简单的例子说明什么方法不可行，什么方法可行。
实现相同目的的另一种方法是MERGE语句。

第16.1节：示例：有效的方法与无效的方法

```
CREATE TABLE tgt ( id, val ) AS
  SELECT 1, 'a' FROM dual UNION ALL
  SELECT 2, 'b' FROM dual
;

表 TGT 已创建。

CREATE TABLE src ( id, val ) AS
  SELECT 1, 'x' FROM dual UNION ALL
  SELECT 2, 'y' FROM dual
;

表 SRC 已创建。

更新
  ( SELECT t.val AS t_val, s.val AS s_val
    FROM   tgt t inner JOIN src s ON t.id = s.id
  )
SET t_val = s_val
;
```

```
SQL错误：ORA-01779：无法修改映射到非键保持表的列01779. 00000 -  "无法修改映射到非键保持表的列"*原
因：尝试插入或更新连接视图中映射到非键保持表的列。


*操作：直接修改底层基础表。
```

想象一下，如果我们在列 src.id 中有值 1 出现多次，并且对应的 src.val 有不同的值，会发生什么情况。显然，更新操作是没有意义的（在任何数据库中都是如此——这是一个逻辑问题）。现在，我们知道 src.id 中没有重复值，但 Oracle 引擎并不知道这一点——所以它会报错。也许这就是为什么许多从业者认为 Oracle“不支持带连接的 UPDATE”的原因？

Oracle 期望 src.id 是唯一的，并且它（Oracle）事先知道这一点。这个问题很容易解决！
请注意，如果更新匹配需要使用多列，复合键（多个列）同样适用。实际上，src.id 可能是主键（PK），而 tgt.id 可能是指向该主键的外键（FK），但这对带连接的更新操作并不重要；重要的是唯一约束。

```
ALTER TABLE src ADD constraint src_uc UNIQUE (id);

TABLE SRC altered.

更新
  ( SELECT t.val AS t_val, s.val AS s_val
    FROM   tgt t inner JOIN src s ON t.id = s.id
  )
SET t_val = s_val
;
```

Chapter 16: Update with Joins

Contrary to widespread misunderstanding (including on SO), Oracle allows updates through joins. However, there are some (pretty logical) requirements. We illustrate what doesn't work and what does through a simple example. Another way to achieve the same is the MERGE statement.

Section 16.1: Examples: what works and what doesn't

```
CREATE TABLE tgt ( id, val ) AS
  SELECT 1, 'a' FROM dual UNION ALL
  SELECT 2, 'b' FROM dual
;

TABLE TGT created.

CREATE TABLE src ( id, val ) AS
  SELECT 1, 'x' FROM dual UNION ALL
  SELECT 2, 'y' FROM dual
;

TABLE SRC created.

UPDATE
  ( SELECT t.val AS t_val, s.val AS s_val
    FROM   tgt t inner JOIN src s ON t.id = s.id
  )
SET t_val = s_val
;
```

```
SQL Error: ORA-01779: cannot modify a column which maps to a non key-preserved table
01779. 00000 -  "cannot modify a column which maps to a non key-preserved table"
*Cause:    An attempt was made to insert or update columns of a join view which
map to a non-key-preserved table.
*Action:   Modify the underlying base tables directly.
```

Imagine what would happen if we had the value 1 in the column `src.id` more than once, with different values for `src.val`. Obviously, the update would make no sense (in ANY database - that's a logical issue). Now, **we** know that there are no duplicates in `src.id`, but the Oracle engine doesn't know that - so it's complaining. Perhaps this is why so many practitioners believe Oracle "doesn't have UPDATE with joins"?

What Oracle expects is that `src.id` should be unique, and that it, Oracle, would know that beforehand. Easily fixed! Note that the same works with composite keys (on more than one column), if the matching for the update needs to use more than one column. In practice, `src.id` may be PK and `tgt.id` may be FK pointing to this PK, but that is not relevant for updates with join; what *is* relevant is the unique constraint.

```
ALTER TABLE src ADD constraint src_uc UNIQUE (id);

TABLE SRC altered.

UPDATE
  ( SELECT t.val AS t_val, s.val AS s_val
    FROM   tgt t inner JOIN src s ON t.id = s.id
  )
SET t_val = s_val
;
```

```
2行已更新.

SELECT * FROM tgt;

ID  VAL
--  ---
 1   x
 2   y
```

相同的结果也可以通过 MERGE 语句实现（MERGE 语句值得有一篇专门的文档文章），我个人在这些情况下更喜欢使用 MERGE，但原因并不是“Oracle 不支持带连接的更新”。正如这个例子所示，Oracle 确实 支持带连接的更新。

```
2 rows updated.

SELECT * FROM tgt;

ID  VAL
--  ---
 1   x
 2   y
```

The same result could be achieved with a MERGE statement (which deserves its own Documentation article), and I personally prefer MERGE in these cases, but the reason is not that "Oracle doesn't do updates with joins." As this example shows, Oracle *does* do updates with joins.

第17章：函数

第17.1节：调用函数

使用函数有几种方式。

使用赋值语句调用函数

```
声明
x NUMBER := functionName(); --函数可以在声明部分调用
BEGIN
x := functionName();
END;
```

在IF语句中调用函数

```
IF functionName() = 100 THEN
    NULL;
END IF;
```

在SELECT语句中调用函数

```
SELECT functionName() FROM DUAL;
```

Chapter 17: Functions

Section 17.1: Calling Functions

There are a few ways to use functions.

Calling a function with an assignment statement

```
DECLARE
    x NUMBER := functionName(); --functions can be called in declaration section
BEGIN
    x := functionName();
END;
```

Calling a function in IF statement

```
IF functionName() = 100 THEN
    NULL;
END IF;
```

Calling a function in a SELECT statement

```
SELECT functionName() FROM DUAL;
```

第18章：统计函数

第18.1节：计算一组值的中位数

自Oracle 10g起，MEDIAN函数 是一个易于使用的聚合函数：

```
SELECT MEDIAN(SAL)
FROM EMP
```

它返回值的中位数

也适用于DATETIME类型的值。

> MEDIAN的结果通过先对行进行排序计算。设组内行数为N，Oracle使用公式RN = (1 + (0.5*(N-1)))计算感兴趣的行号（RN）。聚合函数的最终结果通过对行号为CRN = CEILING(RN)和FRN = FLOOR(RN)的两行值进行线性插值计算得出。

自Oracle 9i起，可以使用PERCENTILE_CONT函数，其工作方式与MEDIAN函数相同，百分位值默认是0.5

```
SELECT PERCENTILE_CONT(.5) WITHIN GROUP(ORDER BY SAL)
FROM EMP
```

Chapter 18: Statistical functions

Section 18.1: Calculating the median of a set of values

The MEDIAN function since Oracle 10g is an easy to use aggregation function:

```
SELECT MEDIAN(SAL)
FROM EMP
```

It returns the median of the values

Works on DATETIME values too.

> The result of MEDIAN is computed by first ordering the rows. Using N as the number of rows in the group, Oracle calculates the row number (RN) of interest with the formula RN = (1 + (0.5*(N-1)). The final result of the aggregate function is computed by linear interpolation between the values from rows at row numbers CRN = CEILING(RN) and FRN = FLOOR(RN).

Since Oracle 9i you can use PERCENTILE_CONT which works the same as MEDIAN function with percentile value defaults to 0.5

```
SELECT PERCENTILE_CONT(.5) WITHIN GROUP(ORDER BY SAL)
FROM EMP
```

第19章：窗口函数

第19.1节：Ratio_To_Report

提供当前行的值与窗口内所有值的比率。

```
--数据
CREATE TABLE 员工 (姓名 VARCHAR2(30), 薪水 NUMBER(10));
INSERT INTO 员工 VALUES ('Bob',2500);
INSERT INTO 员工 VALUES ('Alice',3500);
INSERT INTO 员工 VALUES ('Tom',2700);
INSERT INTO 员工 VALUES ('Sue',2000);
--查询
SELECT 姓名, 薪水, RATIO_TO_REPORT(薪水) OVER () AS 比率
FROM 员工
ORDER BY 薪水, 姓名, 比率;
--输出
```

```
姓名                              薪水      比率
------------------------------ ---------- ----------
Sue                                  2000 .186915888
Bob                                  2500  .23364486
Tom                                  2700 .252336449
Alice                                3500 .327102804
```

Chapter 19: Window Functions

Section 19.1: Ratio_To_Report

Provides the ratio of the current rows value to all the values within the window.

```
--Data
CREATE TABLE Employees (Name VARCHAR2(30), Salary NUMBER(10));
INSERT INTO Employees VALUES ('Bob',2500);
INSERT INTO Employees VALUES ('Alice',3500);
INSERT INTO Employees VALUES ('Tom',2700);
INSERT INTO Employees VALUES ('Sue',2000);
--Query
SELECT Name, Salary, RATIO_TO_REPORT(Salary) OVER () AS Ratio
FROM Employees
ORDER BY Salary, Name, Ratio;
--Output
```

```
NAME                               SALARY      RATIO
------------------------------ ---------- ----------
Sue                                  2000 .186915888
Bob                                  2500  .23364486
Tom                                  2700 .252336449
Alice                                3500 .327102804
```

第20章：创建上下文

参数	详细信息
或替换	重新定义现有的上下文命名空间
命名空间	上下文名称——这是调用SYS_CONTEXT时的命名空间
模式	包的所有者
包	设置或重置上下文属性的数据库包。注意：创建上下文时，数据库包不必存在。
已初始化	指定除 Oracle 数据库之外可以设置上下文的实体。
EXTERNALLY	允许 OCI 接口初始化上下文。
GLOBALLY	允许在建立会话时由 LDAP 目录初始化上下文。
全局访问	允许上下文在整个实例中可访问——多个会话只要具有相同的客户端 ID 就可以共享属性值。

第 20.1 节：创建上下文

```
CREATE CONTEXT my_ctx USING my_pkg;
```

这将创建一个只能由数据库包 my_pkg 中的例程设置的上下文，例如：

```
CREATE PACKAGE my_pkg AS
  PROCEDURE set_ctx;
END my_pkg;

CREATE PACKAGE BODY my_pkg AS
  PROCEDURE set_ctx IS
  BEGIN
    DBMS_SESSION.set_context('MY_CTX','THE KEY','Value');
    DBMS_SESSION.set_context('MY_CTX','ANOTHER','Bla');
  END set_ctx;
END my_pkg;
```

现在，如果一个会话执行以下操作：

```
my_pkg.set_ctx;
```

它现在可以通过以下方式检索该键的值：

```
SELECT SYS_CONTEXT('MY_CTX','THE KEY') FROM dual;

VALUE
```

Chapter 20: Creating a Context

Parameter	Details
OR REPLACE	Redefine an existing context namespace
namespace	Name of the context - this is the namespace for calls to SYS_CONTEXT
schema	Owner of the package
package	Database package that sets or resets the context attributes. Note: the database package doesn't have to exist in order to create the context.
INITIALIZED	Specify an entity other than Oracle Database that can set the context.
EXTERNALLY	Allow the OCI interface to initialize the context.
GLOBALLY	Allow the LDAP directory to initialize the context when establishing the session.
ACCESSED GLOBALLY	Allow the context to be accessible throughout the entire instance - multiple sessions can share the attribute values as long as they have the same Client ID.

Section 20.1: Create a Context

```
CREATE CONTEXT my_ctx USING my_pkg;
```

This creates a context that can only be set by routines in the database package my_pkg, e.g.:

```
CREATE PACKAGE my_pkg AS
  PROCEDURE set_ctx;
END my_pkg;

CREATE PACKAGE BODY my_pkg AS
  PROCEDURE set_ctx IS
  BEGIN
    DBMS_SESSION.set_context('MY_CTX','THE KEY','Value');
    DBMS_SESSION.set_context('MY_CTX','ANOTHER','Bla');
  END set_ctx;
END my_pkg;
```

Now, if a session does this:

```
my_pkg.set_ctx;
```

It can now retrieve the value for the key thus:

```
SELECT SYS_CONTEXT('MY_CTX','THE KEY') FROM dual;

VALUE
```

第21章：拆分分隔字符串

第21.1节：使用层次查询拆分字符串

示例数据：

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- 列表中的多个项目
SELECT 2, 'e'       FROM DUAL UNION ALL -- 列表中的单个项目
SELECT 3, NULL      FROM DUAL UNION ALL -- 空列表
SELECT 4, 'f,,g'    FROM DUAL;          -- 列表中的空项
```

查询:

```
SELECT t.id,
REGEXP_SUBSTR( list, '([^,]*)(,|$)', 1, LEVEL, NULL, 1 ) AS VALUE,
       LEVEL AS lvl
FROM   table_name t
CONNECT BY
id = PRIOR id
AND    PRIOR SYS_GUID() IS NOT NULL
AND    LEVEL < REGEXP_COUNT( list, '([^,]*)(,|$)' )
```

输出:

```
ID 项          LVL
---------- ------- ----------
1 a                  1
1 b              2
1 c              3
1 d              4
2 e              1
3 (NULL)         1
4 f             1
4 (NULL)         2
4 g              3
```

第21.2节：使用PL/SQL函数拆分字符串

PL/SQL函数:

```
创建或替换函数 split_String(
  i_str     输入  VARCHAR2,
i_delim  输入  VARCHAR2 默认值 ','
) 返回 SYS.ODCIVARCHAR2LIST 确定性
作为
p_result      SYS.ODCIVARCHAR2LIST := SYS.ODCIVARCHAR2LIST();
  p_start         数字(5) := 1;
p_end          数字(5);
  c_len 常量 数字(5) := 长度( i_str );
  c_ld  常量 数字(5) := 长度( i_delim );
开始
  如果 c_len > 0 则
p_end := 查找字符串位置( i_str, i_delim, p_start );
    当 p_end > 0 循环
p_result.扩展;
```

Chapter 21: Splitting Delimited Strings

Section 21.1: Splitting Strings using a Hierarchical Query

Sample Data:

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- Multiple items in the list
SELECT 2, 'e'       FROM DUAL UNION ALL -- Single item in the list
SELECT 3, NULL      FROM DUAL UNION ALL -- NULL list
SELECT 4, 'f,,g'    FROM DUAL;          -- NULL item in the list
```

Query:

```
SELECT t.id,
       REGEXP_SUBSTR( list, '([^,]*)(,|$)', 1, LEVEL, NULL, 1 ) AS VALUE,
       LEVEL AS lvl
FROM   table_name t
CONNECT BY
       id = PRIOR id
AND    PRIOR SYS_GUID() IS NOT NULL
AND    LEVEL < REGEXP_COUNT( list, '([^,]*)(,|$)' )
```

Output:

```
        ID ITEM           LVL
---------- ------- ----------
1 a                1
1 b                2
1 c                3
1 d                4
2 e                1
3 (NULL)           1
4 f                1
4 (NULL)           2
4 g                3
```

Section 21.2: Splitting Strings using a PL/SQL Function

PL/SQL Function:

```
CREATE OR REPLACE FUNCTION split_String(
  i_str    IN  VARCHAR2,
  i_delim  IN  VARCHAR2 DEFAULT ','
) RETURN SYS.ODCIVARCHAR2LIST DETERMINISTIC
AS
  p_result       SYS.ODCIVARCHAR2LIST := SYS.ODCIVARCHAR2LIST();
  p_start        NUMBER(5) := 1;
  p_end          NUMBER(5);
  c_len CONSTANT NUMBER(5) := LENGTH( i_str );
  c_ld  CONSTANT NUMBER(5) := LENGTH( i_delim );
BEGIN
  IF c_len > 0 THEN
    p_end := INSTR( i_str, i_delim, p_start );
    WHILE p_end > 0 LOOP
      p_result.EXTEND;
```

```
p_result( p_result.计数 ) := 子串( i_str, p_start, p_end - p_start );
      p_start := p_end + c_ld;
p_end := 查找字符串位置( i_str, i_delim, p_start );
     结束循环;
     如果 p_start <= c_len + 1 则
p_result.扩展;
p_result( p_result.COUNT ) := SUBSTR( i_str, p_start, c_len - p_start + 1 );
     END IF;
  END IF;
  RETURN p_result;
END;
/
```

示例数据：

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- 列表中的多个项目
SELECT 2, 'e'       FROM DUAL UNION ALL -- 列表中的单个项目
SELECT 3, NULL      FROM DUAL UNION ALL -- 空列表
SELECT 4, 'f,,g'    FROM DUAL;          -- 列表中的空项
```

查询:

```
SELECT t.id,
v.column_value 作为 VALUE,
       ROW_NUMBER() OVER ( 按 id 分区 ORDER BY ROWNUM ) 作为 lvl
FROM   table_name t,
       TABLE( split_String( t.list ) ) (+) v
```

输出:

```
ID 项          LVL
---------- ------- ----------
1 a                  1
1 b              2
1 c              3
1 d              4
2 e              1
3 (NULL)         1
4 f              1
4 (NULL)         2
4 g              3
```

第21.3节：使用递归子查询分割字符串Factoring 子句

示例数据：

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- 列表中的多个项目
SELECT 2, 'e'       FROM DUAL UNION ALL -- 列表中的单个项目
SELECT 3, NULL      FROM DUAL UNION ALL -- 空列表
SELECT 4, 'f,,g'    FROM DUAL;          -- 列表中的空项
```

查询:

```
WITH bounds ( id, list, start_pos, end_pos, lvl ) AS (
```

```
      p_result( p_result.COUNT ) := SUBSTR( i_str, p_start, p_end - p_start );
      p_start := p_end + c_ld;
      p_end := INSTR( i_str, i_delim, p_start );
    END LOOP;
    IF p_start <= c_len + 1 THEN
      p_result.EXTEND;
      p_result( p_result.COUNT ) := SUBSTR( i_str, p_start, c_len - p_start + 1 );
    END IF;
  END IF;
  RETURN p_result;
END;
/
```

Sample Data:

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- Multiple items in the list
SELECT 2, 'e'       FROM DUAL UNION ALL -- Single item in the list
SELECT 3, NULL      FROM DUAL UNION ALL -- NULL list
SELECT 4, 'f,,g'    FROM DUAL;          -- NULL item in the list
```

Query:

```
SELECT t.id,
       v.column_value AS VALUE,
       ROW_NUMBER() OVER ( PARTITION BY id ORDER BY ROWNUM ) AS lvl
FROM   table_name t,
       TABLE( split_String( t.list ) ) (+) v
```

Output:

```
        ID ITEM           LVL
---------- ------- ----------
1 a                1
1 b                2
1 c                3
1 d                4
2 e                1
3 (NULL)           1
4 f                1
4 (NULL)           2
4 g                3
```

Section 21.3: Splitting Strings using a Recursive Sub-query Factoring Clause

Sample Data:

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- Multiple items in the list
SELECT 2, 'e'       FROM DUAL UNION ALL -- Single item in the list
SELECT 3, NULL      FROM DUAL UNION ALL -- NULL list
SELECT 4, 'f,,g'    FROM DUAL;          -- NULL item in the list
```

Query:

```
WITH bounds ( id, list, start_pos, end_pos, lvl ) AS (
  SELECT id, list, 1, INSTR( list, ',' ), 1 FROM table_name
UNION ALL
  SELECT id,
list,
end_pos + 1,
         INSTR( list, ',', end_pos + 1 ),
         lvl + 1
  FROM   bounds
  WHERE  end_pos > 0
)
SELECT id,
       SUBSTR(
list,
start_pos,
         CASE end_pos
           WHEN 0
           THEN LENGTH( list ) + 1
           ELSE end_pos
         END - start_pos
       ) AS item,
lvl
FROM   bounds
ORDER BY id, lvl;
```

输出:

```
ID 项          LVL
---------- ------- ----------
1 a                   1
1 b                2
1 c                3
1 d                4
2 e                1
3 (NULL)           1
4 f               1
4 (NULL)           2
4 g                3
```

第21.4节：使用相关表表达式拆分字符串

示例数据：

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- 列表中的多个项目
SELECT 2, 'e'       FROM DUAL UNION ALL -- 列表中的单个项目
SELECT 3, NULL      FROM DUAL UNION ALL -- 空列表
SELECT 4, 'f,,g'    FROM DUAL;          -- 列表中的空项
```

查询:

```
SELECT t.id,
v.COLUMN_VALUE AS VALUE,
       ROW_NUMBER() OVER ( 按 id 分区 ORDER BY ROWNUM ) 作为 lvl
FROM   table_name t,
       表(
         CAST(
MULTISET(
```

```
  SELECT id, list, 1, INSTR( list, ',' ), 1 FROM table_name
UNION ALL
  SELECT id,
         list,
         end_pos + 1,
         INSTR( list, ',', end_pos + 1 ),
         lvl + 1
  FROM   bounds
  WHERE  end_pos > 0
)
SELECT id,
       SUBSTR(
         list,
         start_pos,
         CASE end_pos
           WHEN 0
           THEN LENGTH( list ) + 1
           ELSE end_pos
         END - start_pos
       ) AS item,
      lvl
FROM   bounds
ORDER BY id, lvl;
```

Output:

```
        ID ITEM            LVL
---------- ------- ----------
1 a                  1
1 b                  2
1 c                  3
1 d                  4
2 e                  1
3 (NULL)             1
4 f                  1
4 (NULL)             2
4 g                  3
```

Section 21.4: Splitting Strings using a Correlated Table Expression

Sample Data:

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- Multiple items in the list
SELECT 2, 'e'       FROM DUAL UNION ALL -- Single item in the list
SELECT 3, NULL      FROM DUAL UNION ALL -- NULL list
SELECT 4, 'f,,g'    FROM DUAL;          -- NULL item in the list
```

Query:

```
SELECT t.id,
       v.COLUMN_VALUE AS VALUE,
       ROW_NUMBER() OVER ( PARTITION BY id ORDER BY ROWNUM ) AS lvl
FROM   table_name t,
       TABLE(
         CAST(
           MULTISET(
            SELECT REGEXP_SUBSTR( t.list, '([^,]*)(,|$)', 1, LEVEL, NULL, 1 )
            FROM   DUAL
            CONNECT BY LEVEL < REGEXP_COUNT( t.list, '[^,]*(,|$)' )
          )
          AS SYS.ODCIVARCHAR2LIST
        )
      ) v;
```

输出:

```
ID 项          LVL
---------- ------- ----------
1 a                1
1 b             2
1 c             3
1 d             4
2 e             1
3 (NULL)        1
4 f             1
4 (NULL)        2
4 g             3
```

第21.5节：使用CROSS APPLY拆分字符串（Oracle 12c）

示例数据：

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- 列表中的多个项目
SELECT 2, 'e'       FROM DUAL UNION ALL -- 列表中的单个项目
SELECT 3, NULL      FROM DUAL UNION ALL -- 空列表
SELECT 4, 'f,,g'    FROM DUAL;          -- 列表中的空项
```

查询:

```
SELECT t.id,
REGEXP_SUBSTR( t.list, '([^,]*)($|,)', 1, l.lvl, NULL, 1 ) AS item,
       l.lvl
FROM   table_name t
      CROSS APPLY
       (
         SELECT LEVEL AS lvl
         FROM   DUAL
         CONNECT BY LEVEL <= REGEXP_COUNT( t.list, ',' ) + 1
       ) l;
```

输出:

```
ID 项          LVL
---------- ------- ----------
1 a                1
1 b             2
1 c             3
1 d             4
2 e             1
3 (NULL)        1
4 f             1
4 (NULL)        2
4 g             3
```

```
            SELECT REGEXP_SUBSTR( t.list, '([^,]*)(,|$)', 1, LEVEL, NULL, 1 )
            FROM   DUAL
            CONNECT BY LEVEL < REGEXP_COUNT( t.list, '[^,]*(,|$)' )
          )
          AS SYS.ODCIVARCHAR2LIST
        )
      ) v;
```

Output:

```
        ID ITEM           LVL
---------- ------- ----------
1 a                1
1 b                2
1 c                3
1 d                4
2 e                1
3 (NULL)           1
4 f                1
4 (NULL)           2
4 g                3
```

Section 21.5: Splitting Strings using CROSS APPLY (Oracle 12c)

Sample Data:

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- Multiple items in the list
SELECT 2, 'e'       FROM DUAL UNION ALL -- Single item in the list
SELECT 3, NULL      FROM DUAL UNION ALL -- NULL list
SELECT 4, 'f,,g'    FROM DUAL;          -- NULL item in the list
```

Query:

```
SELECT t.id,
       REGEXP_SUBSTR( t.list, '([^,]*)($|,)', 1, l.lvl, NULL, 1 ) AS item,
       l.lvl
FROM   table_name t
       CROSS APPLY
       (
         SELECT LEVEL AS lvl
         FROM   DUAL
         CONNECT BY LEVEL <= REGEXP_COUNT( t.list, ',' ) + 1
       ) l;
```

Output:

```
        ID ITEM           LVL
---------- ------- ----------
1 a                1
1 b                2
1 c                3
1 d                4
2 e                1
3 (NULL)           1
4 f                1
4 (NULL)           2
4 g                3
```

第21.6节：使用XMLTable和FLWOR表达式拆分字符串

此解决方案使用了Oracle 11及以上版本提供的ora:tokenize XQuery函数。

示例数据：

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- 列表中的多个项目
SELECT 2, 'e'       FROM DUAL UNION ALL -- 列表中的单个项目
SELECT 3, NULL      FROM DUAL UNION ALL -- 空列表
SELECT 4, 'f,,g'    FROM DUAL;          -- 列表中的空项
```

查询:

```
SELECT t.id,
x.item,
x.lvl
来自    table_name t,
XMLTABLE(
           'let $list := ora:tokenize(.,","),
               $cnt := count($list)
          for $val at $r in $list
          where $r < $cnt
return $val'
        传入 list||','
        列
item VARCHAR2(100) 路径 '.',
           lvl 作为 序号
        ) (+) x;
```

输出:

```
ID 项          LVL
---------- ------- ----------
1 a                  1
1 b              2
1 c              3
1 d              4
2 e              1
3 (NULL)      (NULL)
4 f              1
4 (NULL)          2
4 g              3
```

第21.7节：使用XMLTable拆分定界字符串

示例数据：

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- 列表中的多个项目
SELECT 2, 'e'       FROM DUAL UNION ALL -- 列表中的单个项目
SELECT 3, NULL      FROM DUAL UNION ALL -- 空列表
SELECT 4, 'f,,g'    FROM DUAL;          -- 列表中的空项
```

Section 21.6: Splitting Strings using XMLTable and FLWOR expressions

This solution uses the ora:tokenize XQuery function that is available from Oracle 11.

Sample Data:

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- Multiple items in the list
SELECT 2, 'e'       FROM DUAL UNION ALL -- Single item in the list
SELECT 3, NULL      FROM DUAL UNION ALL -- NULL list
SELECT 4, 'f,,g'    FROM DUAL;          -- NULL item in the list
```

Query:

```
SELECT t.id,
       x.item,
       x.lvl
FROM   table_name t,
       XMLTABLE(
         'let $list := ora:tokenize(.,","),
              $cnt := count($list)
          for $val at $r in $list
          where $r < $cnt
          return $val'
       PASSING list||','
       COLUMNS
         item VARCHAR2(100) PATH '.',
         lvl FOR ORDINALITY
       ) (+) x;
```

Output:

```
        ID ITEM           LVL
---------- ------- ----------
1 a                 1
1 b                 2
1 c                 3
1 d                 4
2 e                 1
3 (NULL)       (NULL)
4 f                 1
4 (NULL)            2
4 g                 3
```

Section 21.7: Splitting Delimited Strings using XMLTable

Sample Data:

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- Multiple items in the list
SELECT 2, 'e'       FROM DUAL UNION ALL -- Single item in the list
SELECT 3, NULL      FROM DUAL UNION ALL -- NULL list
SELECT 4, 'f,,g'    FROM DUAL;          -- NULL item in the list
```

查询:

```
SELECT t.id,
        SUBSTR( x.item.getStringVal(), 2 ) 作为 item,
       x.lvl
来自   table_name t
CROSS JOIN
XMLTABLE(
          ( '"#' || REPLACE( t.list, ',', '","#' ) || '"' )
          COLUMNS item XMLTYPE PATH '.',
                  lvl  FOR ORDINALITY
        ) x;
```

（注：添加#字符是为了方便提取NULL值；之后使用SUBSTR(item, 2)将其移除。如果不需要NULL值，则可以简化查询并省略此步骤。）

输出:

```
ID 项          LVL
---------- ------- ----------
1 a                 1
1 b              2
1 c              3
1 d              4
2 e              1
3 (NULL)         1
4 f              1
4 (NULL)         2
4 g              3
```

Query:

```
SELECT t.id,
       SUBSTR( x.item.getStringVal(), 2 ) AS item,
       x.lvl
FROM   table_name t
       CROSS JOIN
       XMLTABLE(
         ( '"#' || REPLACE( t.list, ',', '","#' ) || '"' )
         COLUMNS item XMLTYPE PATH '.',
                 lvl  FOR ORDINALITY
       ) x;
```

(Note: the # character is appended to facilitate extracting `NULL` *values; it is later removed using* `SUBSTR( item, 2 )`*. If* `NULL` *values are not required then you can simplify the query and omit this.)*

Output:

```
        ID ITEM           LVL
---------- ------- ----------
1 a                 1
1 b                 2
1 c                 3
1 d                 4
2 e                 1
3 (NULL)            1
4 f                 1
4 (NULL)            2
4 g                 3
```

第22章：集合与记录

第22.1节：将集合用作拆分函数的返回类型

需要声明类型；这里是 t_my_list；集合是某种TABLE OF

```
CREATE OR REPLACE TYPE t_my_list AS TABLE OF VARCHAR2(100);
```

这是函数。注意()用作一种构造器，以及COUNT和EXTEND关键字，它们帮助你创建和扩展集合；

```
CREATE OR REPLACE
FUNCTION cto_table(p_sep IN VARCHAR2, p_list IN VARCHAR2)
  RETURN t_my_list
AS
--- 该函数接收一个字符串列表，元素由p_sep分隔
--                                              作为分隔符
l_string VARCHAR2(4000) := p_list || p_sep;
  l_sep_index PLS_INTEGER;
l_index PLS_INTEGER := 1;
  l_tab t_my_list     := t_my_list();
BEGIN
  LOOP
l_sep_index := INSTR(l_string, p_sep, l_index);
    EXIT
  WHEN l_sep_index = 0;
    l_tab.EXTEND;
l_tab(l_tab.COUNT) := TRIM(SUBSTR(l_string,l_index,l_sep_index - l_index));
    l_index            := l_sep_index + 1;
  END LOOP;
  RETURN l_tab;
END cto_table;
/
```

然后你可以使用 SQL 中的 TABLE() 函数查看集合的内容；它可以作为 SQL IN (..) 语句中的列表使用：

```
SELECT * FROM A_TABLE
 WHERE A_COLUMN IN ( TABLE(cto_table('|','a|b|c|d')) )
--- 返回 A_COLUMN 在 ('a', 'b', 'c', 'd') 中的记录 --
```

Chapter 22: Collections and Records

Section 22.1: Use a collection as a return type for a split function

It's necessary to declare the type; here `t_my_list`; a collection is a `TABLE OF` *something*

```
CREATE OR REPLACE TYPE t_my_list AS TABLE OF VARCHAR2(100);
```

Here's the function. Notice the `()` used as a kind of constructor, and the `COUNT` and `EXTEND` keywords that help you create and grow your collection;

```
CREATE OR REPLACE
FUNCTION cto_table(p_sep IN VARCHAR2, p_list IN VARCHAR2)
  RETURN t_my_list
AS
--- this function takes a string list, element being separated by p_sep
--                                                         as separator
  l_string VARCHAR2(4000) := p_list || p_sep;
  l_sep_index PLS_INTEGER;
  l_index PLS_INTEGER := 1;
  l_tab t_my_list     := t_my_list();
BEGIN
  LOOP
    l_sep_index := INSTR(l_string, p_sep, l_index);
    EXIT
  WHEN l_sep_index = 0;
    l_tab.EXTEND;
    l_tab(l_tab.COUNT) := TRIM(SUBSTR(l_string,l_index,l_sep_index - l_index));
    l_index            := l_sep_index + 1;
  END LOOP;
  RETURN l_tab;
END cto_table;
/
```

Then you can see the content of the collection with the `TABLE()` function from SQL; it can be used as a list inside a SQL `IN ( ..)` statement:

```
SELECT * FROM A_TABLE
 WHERE A_COLUMN IN ( TABLE(cto_table('|','a|b|c|d')) )
--- gives the records where A_COLUMN in ('a', 'b', 'c', 'd') --
```

第23章：对象类型

第23.1节：访问存储的对象

```
CREATE SEQUENCE test_seq START WITH 1001;

CREATE TABLE test_tab
(
test_id  INTEGER,
   test_obj base_type,
   PRIMARY KEY (test_id)
);

INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, base_type(1,'BASE_TYPE'));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, base_type(2,'BASE_TYPE'));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, mid_type(3, 'MID_TYPE',SYSDATE - 1));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, mid_type(4, 'MID_TYPE',SYSDATE + 1));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, leaf_type(5, 'LEAF_TYPE',SYSDATE - 20,'枫树'));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, leaf_type(6, 'LEAF_TYPE',SYSDATE + 20,'橡树'));
```

返回对象引用：

```
SELECT test_id
      ,test_obj
  FROM test_tab;
```

返回对象引用，全部推送到子类型

```
选择 test_id
      ,TREAT(test_obj AS mid_type) AS obj
  FROM test_tab;
```

返回每个对象的字符串描述符，按类型区分

```
选择 test_id
      ,TREAT(test_obj AS base_type).to_string() AS to_string -- 函数名后需要括号，否则Oracle会查找同名属性。

  FROM test_tab;
```

第23.2节：BASE_TYPE

类型声明：

```
CREATE OR REPLACE TYPE base_type AS OBJECT
(
base_id       INTEGER,
base_attr     VARCHAR2(400),
null_attr     INTEGER, -- 仅用于演示非默认构造函数
   CONSTRUCTOR FUNCTION base_type
   (
i_base_id 整数,
```

Chapter 23: Object Types

Section 23.1: Accessing stored objects

```
CREATE SEQUENCE test_seq START WITH 1001;

CREATE TABLE test_tab
(
   test_id  INTEGER,
   test_obj base_type,
   PRIMARY KEY (test_id)
);

INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, base_type(1,'BASE_TYPE'));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, base_type(2,'BASE_TYPE'));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, mid_type(3, 'MID_TYPE',SYSDATE - 1));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, mid_type(4, 'MID_TYPE',SYSDATE + 1));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, leaf_type(5, 'LEAF_TYPE',SYSDATE - 20,'Maple'));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, leaf_type(6, 'LEAF_TYPE',SYSDATE + 20,'Oak'));
```

Returns object reference:

```
SELECT test_id
      ,test_obj
  FROM test_tab;
```

Returns object reference, pushing all to subtype

```
SELECT test_id
      ,TREAT(test_obj AS mid_type) AS obj
  FROM test_tab;
```

Returns a string descriptor of each object, by type

```
SELECT test_id
      ,TREAT(test_obj AS base_type).to_string() AS to_string -- Parenthesis are needed after the
function name, or Oracle will look for an attribute of this name.
  FROM test_tab;
```

Section 23.2: BASE_TYPE

Type declaration:

```
CREATE OR REPLACE TYPE base_type AS OBJECT
(
   base_id     INTEGER,
   base_attr   VARCHAR2(400),
   null_attr   INTEGER, -- Present only to demonstrate non-default constructors
   CONSTRUCTOR FUNCTION base_type
   (
      i_base_id INTEGER,
i_base_attr VARCHAR2
   ) 返回 SELF 作为 结果,
   成员函数 get_base_id 返回 整数,
   成员函数 get_base_attr 返回 VARCHAR2,
   成员过程 set_base_id(i_base_id 整数),
   成员过程 set_base_attr(i_base_attr VARCHAR2),
   成员函数 to_string 返回 VARCHAR2
) 可实例化 非 最终
```

类型体：

```
创建或替换 类型体 base_type 作为
   构造函数 函数 base_type
   (
i_base_id 整数,
      i_base_attr VARCHAR2
   ) 返回 SELF 作为 结果
   是
   BEGIN
self.base_id := i_base_id;
      self.base_attr := i_base_attr;
      返回;
   结束 base_type;

   成员函数 get_base_id 返回整数类型 IS
   开始
      RETURN self.base_id;
   END get_base_id;

   MEMBER FUNCTION get_base_attr RETURN VARCHAR2 IS
   BEGIN
      RETURN self.base_attr;
   END get_base_attr;

   MEMBER PROCEDURE set_base_id(i_base_id INTEGER) IS
   BEGIN
self.base_id := i_base_id;
   END set_base_id;

   MEMBER PROCEDURE set_base_attr(i_base_attr VARCHAR2) IS
   BEGIN
self.base_attr := i_base_attr;
   END set_base_attr;

   MEMBER FUNCTION to_string RETURN VARCHAR2 IS
   BEGIN
      RETURN 'BASE_ID ['||self.base_id||']; BASE_ATTR ['||self.base_attr||']';
   END to_string;
END;
```

第23.3节：MID_TYPE

类型声明：

```
CREATE OR REPLACE TYPE mid_type UNDER base_type
(
mid_attr DATE,
   CONSTRUCTOR FUNCTION mid_type
   (
i_base_id    INTEGER,
```

```
      i_base_attr VARCHAR2
   ) RETURN SELF AS RESULT,
   MEMBER FUNCTION get_base_id RETURN INTEGER,
   MEMBER FUNCTION get_base_attr RETURN VARCHAR2,
   MEMBER PROCEDURE set_base_id(i_base_id INTEGER),
   MEMBER PROCEDURE set_base_attr(i_base_attr VARCHAR2),
   MEMBER FUNCTION to_string RETURN VARCHAR2
) INSTANTIABLE NOT FINAL
```

Type body:

```
CREATE OR REPLACE TYPE BODY base_type AS
   CONSTRUCTOR FUNCTION base_type
   (
      i_base_id INTEGER,
      i_base_attr VARCHAR2
   ) RETURN SELF AS RESULT
   IS
   BEGIN
      self.base_id := i_base_id;
      self.base_attr := i_base_attr;
      RETURN;
   END base_type;

   MEMBER FUNCTION get_base_id RETURN INTEGER IS
   BEGIN
      RETURN self.base_id;
   END get_base_id;

   MEMBER FUNCTION get_base_attr RETURN VARCHAR2 IS
   BEGIN
      RETURN self.base_attr;
   END get_base_attr;

   MEMBER PROCEDURE set_base_id(i_base_id INTEGER) IS
   BEGIN
      self.base_id := i_base_id;
   END set_base_id;

   MEMBER PROCEDURE set_base_attr(i_base_attr VARCHAR2) IS
   BEGIN
      self.base_attr := i_base_attr;
   END set_base_attr;

   MEMBER FUNCTION to_string RETURN VARCHAR2 IS
   BEGIN
      RETURN 'BASE_ID ['||self.base_id||']; BASE_ATTR ['||self.base_attr||']';
   END to_string;
END;
```

Section 23.3: MID_TYPE

Type declaration:

```
CREATE OR REPLACE TYPE mid_type UNDER base_type
(
   mid_attr DATE,
   CONSTRUCTOR FUNCTION mid_type
   (
      i_base_id    INTEGER,
i_base_attr VARCHAR2,
        i_mid_attr   DATE
    ) 返回 SELF 作为 结果,
    MEMBER FUNCTION get_mid_attr RETURN DATE,
    MEMBER PROCEDURE set_mid_attr(i_mid_attr DATE),
    OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2
) 可实例化 非 最终
```

类型体：

```
CREATE OR REPLACE TYPE BODY mid_type AS
    CONSTRUCTOR FUNCTION mid_type
    (
i_base_id   INTEGER,
        i_base_attr VARCHAR2,
        i_mid_attr   DATE
    ) RETURN SELF AS RESULT
    IS
    BEGIN
self.base_id := i_base_id;
        self.base_attr := i_base_attr;
        self.mid_attr := i_mid_attr;
        RETURN;
    END mid_type;

    成员函数 get_mid_attr 返回日期是
    开始
        RETURN self.mid_attr;
    END get_mid_attr;

    MEMBER PROCEDURE set_mid_attr(i_mid_attr DATE) IS
    BEGIN
self.mid_attr := i_mid_attr;
    END set_mid_attr;

    OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2
    IS
    BEGIN
        RETURN (SELF AS base_type).to_string || '; MID_ATTR [' || self.mid_attr || ']';
    END to_string;
END;
```

第23.4节：叶子类型（LEAF_TYPE）

类型声明：

```
创建或替换类型 leaf_type 继承自 mid_type
(
leaf_attr VARCHAR2(1000),
   构造函数 函数 leaf_type
    (
i_base_id    INTEGER,
i_base_attr VARCHAR2,
i_mid_attr   DATE,
i_leaf_attr VARCHAR2
    ) 返回 SELF 作为 结果,
    成员函数 get_leaf_attr 返回 VARCHAR2,
    成员过程 set_leaf_attr(i_leaf_attr VARCHAR2),
    重写成员函数 to_string 返回 VARCHAR2
) 可实例化 终结
```

```
        i_base_attr VARCHAR2,
        i_mid_attr   DATE
    ) RETURN SELF AS RESULT,
    MEMBER FUNCTION get_mid_attr RETURN DATE,
    MEMBER PROCEDURE set_mid_attr(i_mid_attr DATE),
    OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2
) INSTANTIABLE NOT FINAL
```

Type body:

```
CREATE OR REPLACE TYPE BODY mid_type AS
    CONSTRUCTOR FUNCTION mid_type
    (
        i_base_id   INTEGER,
        i_base_attr VARCHAR2,
        i_mid_attr   DATE
    ) RETURN SELF AS RESULT
    IS
    BEGIN
        self.base_id := i_base_id;
        self.base_attr := i_base_attr;
        self.mid_attr := i_mid_attr;
        RETURN;
    END mid_type;

    MEMBER FUNCTION get_mid_attr RETURN DATE IS
    BEGIN
        RETURN self.mid_attr;
    END get_mid_attr;

    MEMBER PROCEDURE set_mid_attr(i_mid_attr DATE) IS
    BEGIN
        self.mid_attr := i_mid_attr;
    END set_mid_attr;

    OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2
    IS
    BEGIN
        RETURN (SELF AS base_type).to_string || '; MID_ATTR [' || self.mid_attr || ']';
    END to_string;
END;
```

Section 23.4: LEAF_TYPE

Type declaration:

```
CREATE OR REPLACE TYPE leaf_type UNDER mid_type
(
    leaf_attr VARCHAR2(1000),
    CONSTRUCTOR FUNCTION leaf_type
    (
        i_base_id   INTEGER,
        i_base_attr VARCHAR2,
        i_mid_attr   DATE,
        i_leaf_attr VARCHAR2
    ) RETURN SELF AS RESULT,
    MEMBER FUNCTION get_leaf_attr RETURN VARCHAR2,
    MEMBER PROCEDURE set_leaf_attr(i_leaf_attr VARCHAR2),
    OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2
) INSTANTIABLE FINAL
```

类型体：

```
创建或替换类型体 leaf_type 为
   构造函数 函数 leaf_type
   (
i_base_id    整数,
       i_base_attr VARCHAR2,
       i_mid_attr   日期,
       i_leaf_attr VARCHAR2
   ) 返回 自身 为 结果
   是
   BEGIN
self.base_id := i_base_id;
       self.base_attr := i_base_attr;
       self.mid_attr := i_mid_attr;
       self.leaf_attr := i_leaf_attr;
        返回;
   结束 leaf_type;

   成员函数 get_leaf_attr 返回 VARCHAR2 是
   开始
      返回 self.leaf_attr;
   结束 get_leaf_attr;

   成员过程 set_leaf_attr(i_leaf_attr VARCHAR2) 是
   开始
self.leaf_attr := i_leaf_attr;
   END set_leaf_attr;

   重写成员函数 to_string 返回 VARCHAR2 类型 IS
   BEGIN
      RETURN (SELF 作为 mid_type).to_string || '; LEAF_ATTR [' || self.leaf_attr || ']';
   END to_string;
END;
```

Type Body:

```
CREATE OR REPLACE TYPE BODY leaf_type AS
   CONSTRUCTOR FUNCTION leaf_type
   (
      i_base_id   INTEGER,
      i_base_attr VARCHAR2,
      i_mid_attr  DATE,
      i_leaf_attr VARCHAR2
   ) RETURN SELF AS RESULT
   IS
   BEGIN
      self.base_id := i_base_id;
      self.base_attr := i_base_attr;
      self.mid_attr := i_mid_attr;
      self.leaf_attr := i_leaf_attr;
      RETURN;
   END leaf_type;

   MEMBER FUNCTION get_leaf_attr RETURN VARCHAR2 IS
   BEGIN
      RETURN self.leaf_attr;
   END get_leaf_attr;

   MEMBER PROCEDURE set_leaf_attr(i_leaf_attr VARCHAR2) IS
   BEGIN
      self.leaf_attr := i_leaf_attr;
   END set_leaf_attr;

   OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2 IS
   BEGIN
      RETURN (SELF AS mid_type).to_string || '; LEAF_ATTR [' || self.leaf_attr || ']';
   END to_string;
END;
```

第24章：循环

第24.1节：简单循环

```
声明
v_counter 数字(2);

BEGIN
v_counter := 0;
  循环
v_counter := v_counter + 1;
    DBMS_OUTPUT.put_line('行号' || v_counter);

    当 v_counter = 10时退出;
  结束循环;
END;
```

第24.2节：WHILE循环

WHILE 循环会一直执行，直到满足结束条件。简单示例：

```
声明
v_counter NUMBER(2); --计数器变量声明

BEGIN
v_counter := 0; --起始点，我们迭代的第一个值

  WHILE v_counter < 10 LOOP --退出条件

    DBMS_OUTPUT.put_line('当前循环迭代次数为 ' || v_counter); --在 dbms 脚本输出中显示当前迭代次数

v_counter := v_counter + 1; --计数器值递增，非常重要的一步

  END LOOP; --循环声明结束
END;
```

该循环会一直执行，直到变量 v_counter 的当前值小于十。

结果如下：

```
CURRENT iteration OF LOOP IS 0
CURRENT iteration OF LOOP IS 1
CURRENT iteration OF LOOP IS 2
CURRENT iteration OF LOOP IS 3
CURRENT iteration OF LOOP IS 4
CURRENT iteration OF LOOP IS 5
CURRENT iteration OF LOOP IS 6
CURRENT iteration OF LOOP IS 7
CURRENT iteration OF LOOP IS 8
CURRENT iteration OF LOOP IS 9
```

最重要的是，我们的循环从“0”开始，因此结果的第一行是“当前循环迭代为
0”。

第24.3节：FOR循环

FOR循环的工作原理与其他循环类似。FOR循环执行的次数是确定的，这个次数

Chapter 24: Loop

Section 24.1: Simple Loop

```
DECLARE
v_counter NUMBER(2);

BEGIN
  v_counter := 0;
  LOOP
    v_counter := v_counter + 1;
    DBMS_OUTPUT.put_line('Line number' || v_counter);

    EXIT WHEN v_counter = 10;
  END LOOP;
END;
```

Section 24.2: WHILE Loop

The WHILE loop is executed untill the condition of end is fulfilled. Simple example:

```
DECLARE
v_counter NUMBER(2); --declaration of counter variable

BEGIN
  v_counter := 0; --point of start, first value of our iteration

  WHILE v_counter < 10 LOOP --exit condition

    DBMS_OUTPUT.put_line('Current iteration of loop is ' || v_counter); --show current iteration
number in dbms script output
    v_counter := v_counter + 1; --incrementation of counter value, very important step

  END LOOP; --end of loop declaration
END;
```

This loop will be executed untill current value of variable v_counter will be less than ten.

The result:

```
CURRENT iteration OF LOOP IS 0
CURRENT iteration OF LOOP IS 1
CURRENT iteration OF LOOP IS 2
CURRENT iteration OF LOOP IS 3
CURRENT iteration OF LOOP IS 4
CURRENT iteration OF LOOP IS 5
CURRENT iteration OF LOOP IS 6
CURRENT iteration OF LOOP IS 7
CURRENT iteration OF LOOP IS 8
CURRENT iteration OF LOOP IS 9
```

The most important thing is, that our loop starts with '0' value, so first line of results is 'Current iteration of loop is 0'.

Section 24.3: FOR Loop

Loop FOR works on similar rules as other loops. FOR loop is executed exact number of times and this number is

在开始时已知——上下限直接在代码中设置。在本例中，每一步循环的增量为1。

简单示例：

```
声明
v_counter NUMBER(2); --计数器变量声明

BEGIN
v_counter := 0; --起点，我们迭代的第一个值，变量的执行

  FOR v_counter IN 1..10 LOOP --声明循环语句的上下限的位置
- 在本例中，循环将执行10次，起始值为1

    DBMS_OUTPUT.put_line('当前循环迭代次数为 ' || v_counter); --在 dbms 脚本输出中显示当前迭代次数


  END LOOP; --循环声明结束
END;
```

结果是：

```
CURRENT iteration OF LOOP IS 1
CURRENT iteration OF LOOP IS 2
CURRENT iteration OF LOOP IS 3
CURRENT iteration OF LOOP IS 4
CURRENT iteration OF LOOP IS 5
CURRENT iteration OF LOOP IS 6
CURRENT iteration OF LOOP IS 7
CURRENT iteration OF LOOP IS 8
CURRENT iteration OF LOOP IS 9
CURRENT iteration OF LOOP IS 10
```

FOR循环还有一个附加特性，即可以反向执行。在声明循环的上下限时使用额外的关键字"REVERSE"即可实现。每次循环执行时，v_counter的值减1。

示例：

```
声明
v_counter NUMBER(2); --计数器变量声明

BEGIN
v_counter := 0; --起始点

  FOR v_counter IN REVERSE 1..10 LOOP

    DBMS_OUTPUT.put_line('当前循环迭代次数为 ' || v_counter); --在 dbms 脚本输出中显示当前迭代次数


  END LOOP; --循环声明结束
END;
```

结果如下：

```
CURRENT iteration OF LOOP IS 10
CURRENT iteration OF LOOP IS 9
CURRENT iteration OF LOOP IS 8
CURRENT iteration OF LOOP IS 7
CURRENT iteration OF LOOP IS 6
```

known at the beginning - lower and upper limits are directly set in code. In every step in this example, loop is increment by 1.

Simple example:

```
DECLARE
v_counter NUMBER(2); --declaration of counter variable

BEGIN
  v_counter := 0; --point of start, first value of our iteration, execute of variable

  FOR v_counter IN 1..10 LOOP --The point, where lower and upper point of loop statement is declared
- in this example, loop will be executed 10 times, start with value of 1

    DBMS_OUTPUT.put_line('Current iteration of loop is ' || v_counter); --show current iteration
number in dbms script output

  END LOOP; --end of loop declaration
END;
```

And the result is:

```
CURRENT iteration OF LOOP IS 1
CURRENT iteration OF LOOP IS 2
CURRENT iteration OF LOOP IS 3
CURRENT iteration OF LOOP IS 4
CURRENT iteration OF LOOP IS 5
CURRENT iteration OF LOOP IS 6
CURRENT iteration OF LOOP IS 7
CURRENT iteration OF LOOP IS 8
CURRENT iteration OF LOOP IS 9
CURRENT iteration OF LOOP IS 10
```

Loop FOR has additional property, which is working in reverse. Using additional word 'REVERSE' in declaration of lower and upper limit of loop allow to do that. Every execution of loop decrement value of v_counter by 1.

Example:

```
DECLARE
v_counter NUMBER(2); --declaration of counter variable

BEGIN
  v_counter := 0; --point of start

  FOR v_counter IN REVERSE 1..10 LOOP

    DBMS_OUTPUT.put_line('Current iteration of loop is ' || v_counter); --show current iteration
number in dbms script output

  END LOOP; --end of loop declaration
END;
```

And the result:

```
CURRENT iteration OF LOOP IS 10
CURRENT iteration OF LOOP IS 9
CURRENT iteration OF LOOP IS 8
CURRENT iteration OF LOOP IS 7
CURRENT iteration OF LOOP IS 6
CURRENT iteration OF LOOP IS 5
CURRENT iteration OF LOOP IS 4
CURRENT iteration OF LOOP IS 3
CURRENT iteration OF LOOP IS 2
CURRENT iteration OF LOOP IS 1
```

```
CURRENT iteration OF LOOP IS 5
CURRENT iteration OF LOOP IS 4
CURRENT iteration OF LOOP IS 3
CURRENT iteration OF LOOP IS 2
CURRENT iteration OF LOOP IS 1
```

第25章：游标

第25.1节：参数化的“FOR循环”游标

```
声明
  CURSOR c_emp_to_be_raised(p_sal emp.sal%TYPE) IS
    SELECT * FROM emp WHERE  sal < p_sal;
BEGIN
  FOR cRowEmp IN c_emp_to_be_raised(1000) LOOP
    DBMS_OUTPUT.Put_Line(cRowEmp .eName ||' ' ||cRowEmp.sal||'... 应该加薪 ;)');
  END LOOP;
END;
/
```

第25.2节：隐式“FOR循环”游标

```
BEGIN
  FOR x IN (SELECT * FROM emp WHERE sal < 100) LOOP
    DBMS_OUTPUT.Put_Line(x.eName ||' '||x.sal||'... 应该真的被抛出 :D');
  END LOOP;
END;
/
```

- 第一个优点是无需繁琐的声明（想想你在以前版本中使用的那个可怕的“游标”东西）
- 第二个优点是你先构建你的查询语句，然后当你得到想要的结果时，可以立即在PL/SQL循环中访问查询的字段（x.<myfield>）
- 循环打开游标，每次循环获取一条记录。循环结束时游标关闭。
- 隐式游标更快，因为解释器的工作量随着代码长度增加而增加。代码越少，解释器的工作量越小。

第25.3节：处理游标

- 声明游标以扫描记录列表
- 打开游标
- 将当前记录取出到变量中（这会增加位置）
- 使用%notfound检测列表结束
- 别忘了关闭游标，以限制当前上下文中的资源消耗

--

```
声明
  CURSOR curCols 是 -- 从给定表中选择列名和类型
         SELECT column_name, data_type FROM all_tab_columns WHERE table_name='MY_TABLE';
  v_tab_column all_tab_columns.column_name%TYPE;
  v_data_type all_tab_columns.data_type%TYPE;
  v_ INTEGER := 1;
BEGIN
  打开 curCols;
  循环
     获取 curCols 进入 v_tab_column, v_data_type;
     如果 curCols%未找到 或 v_ > 2000 则
       退出;
     结束 如果;
```

Chapter 25: Cursors

Section 25.1: Parameterized "FOR loop" Cursor

```
DECLARE
  CURSOR c_emp_to_be_raised(p_sal emp.sal%TYPE) IS
    SELECT * FROM emp WHERE  sal < p_sal;
BEGIN
  FOR cRowEmp IN c_emp_to_be_raised(1000) LOOP
    DBMS_OUTPUT.Put_Line(cRowEmp .eName ||' ' ||cRowEmp.sal||'... should be raised ;)');
  END LOOP;
END;
/
```

Section 25.2: Implicit "FOR loop" cursor

```
BEGIN
  FOR x IN (SELECT * FROM emp WHERE sal < 100) LOOP
    DBMS_OUTPUT.Put_Line(x.eName ||' '||x.sal||'... should REALLY be raised :D');
  END LOOP;
END;
/
```

- First advantage is there is no tedious declaration to do (think of this horrible "CURSOR" thing you had in previous versions)
- second advantage is you first build your select query, then when you have what you want, you immediately can access the fields of your query (`x.<myfield>`) in your PL/SQL loop
- The loop opens the cursor and fetches one record at a time for every loop. At the end of the loop the cursor is closed.
- Implicit cursors are faster because the interpreter's work grows as the code gets longer. The less code the less work the interpreter has to do.

Section 25.3: Handling a CURSOR

- Declare the cursor to scan a list of records
- Open it
- Fetch current record into variables (this increments position)
- Use `%notfound` to detect end of list
- Don't forget to close the cursor to limit resources consumption in current context

--

```
DECLARE
  CURSOR curCols IS -- select column name and type from a given table
         SELECT column_name, data_type FROM all_tab_columns WHERE table_name='MY_TABLE';
  v_tab_column all_tab_columns.column_name%TYPE;
  v_data_type all_tab_columns.data_type%TYPE;
  v_ INTEGER := 1;
BEGIN
  OPEN curCols;
  LOOP
     FETCH curCols INTO v_tab_column, v_data_type;
     IF curCols%notfound OR v_ > 2000 THEN
       EXIT;
     END IF;
```

```
    DBMS_OUTPUT.put_line(v_||':列 '||v_tab_column||' 的类型是 '|| v_data_type||'。');
   v_:= v_ + 1;
  结束 循环;

  -- 无论如何关闭
  如果 curCols%是打开状态 则
    关闭 curCols;
  结束 如果;
结束;
/
```

第25.4节：使用 SYS_REFCURSOR

当您需要轻松处理不是来自表，而是更具体地来自函数返回的列表时，SYS_REFCURSOR 可以用作返回类型：

返回游标的函数

```
创建或替换函数 list_of (required_type_in IN VARCHAR2)
   返回 SYS_REFCURSOR
是
v_ SYS_REFCURSOR;
开始
   根据 required_type_in
      当 'CATS'
      然后
         打开 v_ 用于
           选择 昵称 从 (
                选择 'minou' 昵称 从 dual
      联合所有选择 'minâ'           从 dual
      联合所有选择 'minon'          从 dual
           );
      当 'DOGS'
      则
         打开 v_ 用于
              选择 dog_call 从 (
                SELECT 'bill'   dog_call FROM dual
      UNION ALL SELECT 'nestor'          FROM dual
      UNION ALL SELECT 'raoul'           FROM dual
           );
   END CASE;
   -- 使用此方法时，您不能关闭游标。
   RETURN v_;
END list_of;
/
```

以及如何使用它：

```
声明
v_names   SYS_REFCURSOR;
   v_        VARCHAR2 (32767);
BEGIN
v_names := list_of('CATS');
   LOOP
      FETCH v_names INTO v_;
      EXIT  WHEN v_names%NOTFOUND;
      DBMS_OUTPUT.put_line(v_);
   END LOOP;
   -- 这里关闭它
   CLOSE v_names;
END;
/
```

```
    DBMS_OUTPUT.put_line(v_||':Column '||v_tab_column||' is of '|| v_data_type||' Type.');
    v_:= v_ + 1;
  END LOOP;

  -- Close in any case
  IF curCols%ISOPEN THEN
    CLOSE curCols;
  END IF;
END;
/
```

Section 25.4: Working with SYS_REFCURSOR

SYS_REFCURSOR can be used as a return type when you need to easily handle a list returned not from a table, but more specifically from a function:

function returning a cursor

```
CREATE OR REPLACE FUNCTION list_of (required_type_in IN VARCHAR2)
   RETURN SYS_REFCURSOR
IS
   v_ SYS_REFCURSOR;
BEGIN
   CASE required_type_in
      WHEN 'CATS'
      THEN
         OPEN v_ FOR
           SELECT nickname FROM (
                SELECT 'minou' nickname FROM dual
      UNION ALL SELECT 'minâ'           FROM dual
      UNION ALL SELECT 'minon'          FROM dual
           );
      WHEN 'DOGS'
      THEN
         OPEN v_ FOR
              SELECT dog_call FROM (
                SELECT 'bill'   dog_call FROM dual
      UNION ALL SELECT 'nestor'          FROM dual
      UNION ALL SELECT 'raoul'           FROM dual
           );
   END CASE;
   -- Whit this use, you must not close the cursor.
   RETURN v_;
END list_of;
/
```

and how to use it:

```
DECLARE
   v_names   SYS_REFCURSOR;
   v_        VARCHAR2 (32767);
BEGIN
   v_names := list_of('CATS');
   LOOP
      FETCH v_names INTO v_;
      EXIT WHEN v_names%NOTFOUND;
      DBMS_OUTPUT.put_line(v_);
   END LOOP;
   -- here you close it
   CLOSE v_names;
END;
/
```

第26章：序列

参数	详细信息
模式	模式名称
按数字间隔递增	
起始于	所需的第一个数字
最大值	序列的最大值
无最大值时默认最大值	
最小值	序列的最小值
无最小值时默认最小值	
循环	达到此值后重置到起始值
nocycle	默认
缓存	预分配限制
无缓存	默认
顺序	保证数字顺序
无顺序	默认

第26.1节：创建序列：示例

目的

使用 CREATE SEQUENCE 语句创建序列，序列是一种数据库对象，多个用户可以从中生成唯一的整数。您可以使用序列自动生成主键值。

当生成序列号时，序列会递增，这与事务的提交或回滚无关。如果两个用户同时递增同一个序列，则每个用户获得的序列号可能会有间隔，因为序列号是由另一个用户生成的。一个用户永远无法获得另一个用户生成的序列号。在一个用户生成序列值后，无论序列是否被其他用户递增，该用户都可以继续访问该值。

序列号的生成独立于表，因此同一个序列可以用于一个或多个表。个别序列号可能会被跳过，因为它们是在最终回滚的事务中生成并使用的。此外，单个用户可能不会意识到其他用户也在使用同一个序列。

创建序列后，您可以在 SQL 语句中使用 CURRVAL 伪列访问其值，该伪列返回序列的当前值；或者使用 NEXTVAL 伪列，该伪列递增序列并返回新值。

前提条件

要在自己的模式中创建序列，必须拥有 CREATE SEQUENCE 系统权限。

要在其他用户的模式中创建序列，必须拥有 CREATE ANY SEQUENCE 系统权限。

创建序列示例：以下语句在示例模式 oe 中创建序列 customers_seq。该序列可用于在向 customers 表添加行时提供客户 ID 号。

```
CREATE SEQUENCE customers_seq
START WITH     1000
INCREMENT BY   1
NOCACHE
```

Chapter 26: Sequences

Parameter	Details
schema	schema name
increment by	interval between the numbers
start with	first number needed
maxvalue	Maximum value for the sequence
nomaxvalue	Maximum value is defaulted
minvalue	minimum value for the sequence
nominvalue	minimum value is defaulted
cycle	Reset to the start after reaching this value
nocycle	Default
cache	Preallocation limit
nocache	Default
order	Guarantee the order of numbers
noorder	default

Section 26.1: Creating a Sequence: Example

Purpose

Use the CREATE SEQUENCE statement to create a sequence, which is a database object from which multiple users may generate unique integers. You can use sequences to automatically generate primary key values.

When a sequence number is generated, the sequence is incremented, independent of the transaction committing or rolling back. If two users concurrently increment the same sequence, then the sequence numbers each user acquires may have gaps, because sequence numbers are being generated by the other user. One user can never acquire the sequence number generated by another user. After a sequence value is generated by one user, that user can continue to access that value regardless of whether the sequence is incremented by another user.

Sequence numbers are generated independently of tables, so the same sequence can be used for one or for multiple tables. It is possible that individual sequence numbers will appear to be skipped, because they were generated and used in a transaction that ultimately rolled back. Additionally, a single user may not realize that other users are drawing from the same sequence.

After a sequence is created, you can access its values in SQL statements with the CURRVAL pseudocolumn, which returns the current value of the sequence, or the NEXTVAL pseudocolumn, which increments the sequence and returns the new value.

Prerequisites

To create a sequence in your own schema, you must have the CREATE SEQUENCE system privilege.

To create a sequence in another user's schema, you must have the CREATE ANY SEQUENCE system privilege.

Creating a Sequence: Example The following statement creates the sequence customers_seq in the sample schema oe. This sequence could be used to provide customer ID numbers when rows are added to the customers table.

```
CREATE SEQUENCE customers_seq
START WITH     1000
INCREMENT BY   1
NOCACHE
```

```
NOCYCLE;
```

对 customers_seq.nextval 的第一次引用返回 1000。第二次返回 1001。每次后续引用将返回比前一次引用大 1 的值。

```
NOCYCLE;
```

The first reference to customers_seq.nextval returns 1000. The second returns 1001. Each subsequent reference will return a value 1 greater than the previous reference.

第27章：索引

这里我将通过示例解释不同的索引，索引如何提升查询性能，索引如何降低 DML 性能等

第27.1节：B 树索引

```
CREATE INDEX ord_customer_ix ON orders (customer_id);
```

默认情况下，如果我们不做任何说明，Oracle 会创建一个 B 树索引。但我们应该知道何时使用它。B 树索引以二叉树格式存储数据。正如我们所知，索引是一个模式对象，它为被索引列的每个值存储某种条目。因此，每当在这些列上进行搜索时，它会在索引中检查该记录的确切位置以快速访问。关于索引的几点说明：

- 在索引中搜索条目时，使用某种二分查找算法。
- **当数据基数较高时，B 树索引是完美的选择。**
- 索引会使数据操作语言（DML）变慢，因为对于每条记录，索引列中都应该有一个索引条目。
- 所以，如果没有必要，我们应该避免创建索引。

第27.2节：位图索引

```
CREATE BITMAP INDEX
emp_bitmap_idx
ON index_demo (gender);
```

- 当数据基数较低时，使用位图索引。
- 这里，性别的值基数较低。可能的值有男性、女性及其他。
- 因此，如果我们为这3个值创建二叉树，搜索时会有不必要的遍历。
- 在位图结构中，会创建一个二维数组，表中每一行对应一列。每列代表位图索引中的一个不同值。这个二维数组表示索引中的每个值乘以表中的行数。
- 在行检索时，Oracle会将位图解压到RAM数据缓冲区，以便快速扫描匹配的值。这些匹配的值以行ID列表的形式传递给Oracle，这些行ID值可以直接访问所需的信息。

第27.3节：基于函数的索引

```
CREATE INDEX first_name_idx ON user_data (UPPER(first_name));


SELECT *
FROM   user_data
WHERE  UPPER(first_name) = 'JOHN2';
```

- 基于函数的索引是指基于函数创建索引。
- 如果在搜索（where子句）中经常使用某个函数，最好基于该函数创建索引。
- 这里，在示例中，搜索时使用了Upper()函数。因此，最好使用upper函数创建索引。

Chapter 27: Indexes

Here I will explain different index using example, how index increase query performance, how index decrease DML performance etc

Section 27.1: b-tree index

```
CREATE INDEX ord_customer_ix ON orders (customer_id);
```

By default, if we do not mention anything, oracle creates an index as a b-tree index. But we should know when to use it. B-tree index stores data as binary tree format. As we know that, index is a schema object which stores some sort of entry for each value for the indexed column. So, whenever any search happens on those columns, it checks in the index for the exact location of that record to access fast. Few points about indexing:

- To search for entry in the index, some sort of binary search algorithm used.
- When **data cardinality is high, b-tree** index is perfect to use.
- Index makes DML slow, as for each record, there should be one entry in the index for indexed column.
- So, if not necessary, we should avoid creating index.

Section 27.2: Bitmap Index

```
CREATE BITMAP INDEX
emp_bitmap_idx
ON index_demo (gender);
```

- Bitmap index is used when **data cardinality is low.**
- Here, **Gender** has value with low cardinality. Values are may be Male, Female & others.
- So, if we create a binary tree for this 3 values while searching it will have unnecessary traverse.
- In bitmap structures, a two-dimensional array is created with one column for every row in the table being indexed. Each column represents a distinct value within the bitmapped index. This two-dimensional array represents each value within the index multiplied by the number of rows in the table.
- At row retrieval time, Oracle decompresses the bitmap into the RAM data buffers so it can be rapidly scanned for matching values. These matching values are delivered to Oracle in the form of a Row-ID list, and these Row-ID values may directly access the required information.

Section 27.3: Function Based Index

```
CREATE INDEX first_name_idx ON user_data (UPPER(first_name));


SELECT *
FROM   user_data
WHERE  UPPER(first_name) = 'JOHN2';
```

- Function based index means, creating index based on a function.
- If in search (where clause), frequently any function is used, it's better to create index based on that function.
- Here, in the example, for search, **Upper()** function is being used. So, it's better to create index using upper function.

第28章：提示

参数	详细信息
并行度（DOP）	是指您希望查询打开的并行连接/进程数量。通常是2、4、8、16等。
表名	将应用并行提示的表名。

第28.1节：USE_NL

使用嵌套循环。

用法：use_nl(A B)

该提示将要求引擎使用嵌套循环方法连接表A和表B。即逐行比较。
该提示不强制连接顺序，只是要求使用嵌套循环。

```
SELECT /*+use_nl(e d)*/ *
FROM 员工 E
JOIN 部门 D ON E.DepartmentID = D.ID
```

第28.2节：APPEND提示

“使用直接路径方法插入新行”。

APPEND提示指示引擎使用直接路径加载。这意味着引擎不会使用传统的插入方式（使用内存结构和标准锁），而是直接将数据写入表空间。总是创建新的块并追加到表的段中。这将更快，但有一些限制：

- 在同一会话中，除非提交或回滚事务，否则无法读取你追加的表中的数据。
- 如果表上定义了触发器，Oracle将不会使用直接路径（对于sqlldr加载则是另一回事）。
- 其他

示例。

```
INSERT /*+append*/ INTO 员工
SELECT *
FROM 员工;
```

第28.3节：并行提示

语句级别的并行提示是最简单的：

```
SELECT /*+ PARALLEL(8) */ 名字, 姓氏 FROM 员工 emp;
```

对象级别的并行提示提供了更多控制，但更容易出错；开发者经常忘记使用别名
而不是对象名，或者忘记包含某些对象。

```
SELECT /*+ PARALLEL(emp,8) */ first_name, last_name FROM employee emp;
```

```
SELECT /*+ PARALLEL(table_alias,并行度) */ FROM table_name table_alias;
```

假设一个查询在不使用并行提示的情况下执行需要100秒。如果我们将同一查询的并行度（DOP）改为2，

Chapter 28: Hints

Parameters	Details
Degree of Parallelism (DOP)	It is the number of parallel connection/processes which you want your query to open up. It is usually 2, 4, 8, 16 so on.
Table Name	The name of the table on which parallel hint will be applied.

Section 28.1: USE_NL

Use Nested Loops.

Usage : `use_nl(A B)`

This hint will ask the engine to use nested loop method to join the tables A and B. That is row by row comparison.
The hint does not force the order of the join, just asks for NL.

```
SELECT /*+use_nl(e d)*/ *
FROM Employees E
JOIN Departments D ON E.DepartmentID = D.ID
```

Section 28.2: APPEND HINT

"Use DIRECT PATH method for inserting new rows".

The `APPEND` hint instructs the engine to use direct path load. This means that the engine will not use a conventional insert using memory structures and standard locks, but will write directly to the tablespace the data. Always creates new blocks which are appended to the table's segment. This will be faster, but have some limitations:

- You cannot read from the table you appended in the same session until you commmit or rollback the transaction.
- If there are triggers defined on the table Oracle will not use direct path(it's a different story for sqlldr loads).
- others

Example.

```
INSERT /*+append*/ INTO Employees
SELECT *
FROM Employees;
```

Section 28.3: Parallel Hint

Statement-level parallel hints are the easiest:

```
SELECT /*+ PARALLEL(8) */ first_name, last_name FROM employee emp;
```

Object-level parallel hints give more control but are more prone to errors; developers often forget to use the alias instead of the object name, or they forget to include some objects.

```
SELECT /*+ PARALLEL(emp,8) */ first_name, last_name FROM employee emp;
```

```
SELECT /*+ PARALLEL(table_alias,Degree of Parallelism) */ FROM table_name table_alias;
```

Let's say a query takes 100 seconds to execute without using parallel hint. If we change DOP to 2 for same query,

那么理想情况下带有并行提示的同一查询将花费50秒。同理，使用并行度为4将花费25秒。

实际上，并行执行取决于许多其他因素，且不会线性扩展。对于
较短的运行时间尤其如此，因为并行开销可能大于在多个并行服务器上运行所带来的收益。

第28.4节：USE_HASH

指示引擎使用哈希方法连接参数中的表。

用法：use_hash(TableA [TableB] ... [TableN])

正如在许多地方所解释的，“在哈希连接中，Oracle访问一张表（通常是连接结果中较小的表），
并在内存中基于连接键构建哈希表。然后扫描连接中的另一张表（通常是较大的表），
并在哈希表中查找匹配项。”

当表较大、没有索引等情况下，优先使用该方法而非嵌套循环方法。

注意：该提示不强制连接顺序，只是请求使用哈希连接方法。

使用示例：

```
SELECT /*+use_hash(e d)*/ *
FROM Employees E
JOIN 部门 D ON E.DepartmentID = D.ID
```

第28.5节：FULL

FULL提示告诉Oracle对指定表执行全表扫描，无论是否可以使用索引。

```
CREATE TABLE fullTable(id) AS SELECT LEVEL FROM dual CONNECT BY LEVEL < 100000;
CREATE INDEX idx ON fullTable(id);
```

没有提示时，使用索引：

```
SELECT COUNT(1) FROM fullTable f WHERE id BETWEEN 10 AND 100;
```

```
-------------------------------------------------------------------------
| Id  | 操作              | 名称 | 行数  | 字节数 | 代价 (%CPU)| 时间     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |     13 |      3   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE   |      |     1 |     13 |            |          |
|*  2 |   INDEX RANGE SCAN| IDX  |     2 |     26 |      3   (0)| 00:00:01 |
-------------------------------------------------------------------------
```

FULL 提示强制进行全表扫描：

```
SELECT /*+ full(f) */ COUNT(1) FROM fullTable f WHERE id BETWEEN 10 AND 100;
```

```
-------------------------------------------------------------------------------
| Id  | 操作               | 名称       | 行数  | 字节数 | 代价 (%CPU)| 时间     |
-------------------------------------------------------------------------------
|   0 | SELECT 语句        |           |     1 |     13 |     47   (3)| 00:00:01 |
|   1 |  排序聚合          |           |     1 |     13 |            |          |
|*  2 |   全表扫描         | FULLTABLE |     2 |     26 |     47   (3)| 00:00:01 |
-------------------------------------------------------------------------------
```

then *ideally* the same query with parallel hint will take 50 second. Similarly using DOP as 4 will take 25 seconds.

In practice, parallel execution depends on many other factors and does not scale linearly. This is especially true for small run times where the parallel overhead may be larger than the gains from running in multiple parallel servers.

Section 28.4: USE_HASH

Instructs the engine to use hash method to join tables in the argument.

Usage : `use_hash(TableA [TableB] ... [TableN])`

As explained in many places, "in a HASH join, Oracle accesses one table (usually the smaller of the joined results) and builds a hash table on the join key in memory. It then scans the other table in the join (usually the larger one) and probes the hash table for matches to it."

It is preferred against Nested Loops method when the tables are big, no indexes are at hand, etc.

Note: The hint does not force the order of the join, just asks for HASH JOIN method.

Example of usage:

```
SELECT /*+use_hash(e d)*/ *
FROM Employees E
JOIN Departments D ON E.DepartmentID = D.ID
```

Section 28.5: FULL

The FULL hint tells Oracle to perform a full table scan on a specified table, no matter if an index can be used.

```
CREATE TABLE fullTable(id) AS SELECT LEVEL FROM dual CONNECT BY LEVEL < 100000;
CREATE INDEX idx ON fullTable(id);
```

With no hints, the index is used:

```
SELECT COUNT(1) FROM fullTable f WHERE id BETWEEN 10 AND 100;
```

```
--------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    13 |     3   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE   |      |     1 |    13 |            |          |
|*  2 |   INDEX RANGE SCAN| IDX  |     2 |    26 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------
```

FULL hint forces a full scan:

```
SELECT /*+ full(f) */ COUNT(1) FROM fullTable f WHERE id BETWEEN 10 AND 100;
```

```
--------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |     1 |    13 |    47   (3)| 00:00:01 |
|   1 |  SORT AGGREGATE    |           |     1 |    13 |            |          |
|*  2 |   TABLE ACCESS FULL| FULLTABLE |     2 |    26 |    47   (3)| 00:00:01 |
--------------------------------------------------------------------------------
```

第28.6节：结果缓存

Oracle（11g及以上版本）允许将SQL查询缓存在SGA中并重用以提高性能。它从缓存中查询数据，而不是数据库。相同查询的后续执行更快，因为数据现在是从缓存中获取的。

```
SELECT /*+ result_cache */ NUMBER FROM main_table;
```

输出 -

```
数字
I------
```

2345678910耗时：00:00:02.20如果我现在再次运行相同的查询，执行时间将减少，因为数据现在是从首次执行时设置的缓存中获取的。输出 - 数字------12345678910耗时：00:00:00.10注意耗时从2.20秒减少到0.10秒。结果缓存会一直保存缓存，直到数据库中的数据被更新/修改/删除。任何更改都会释放缓存。

Section 28.6: Result Cache

Oracle (*11g and above*) allows the SQL queries to be cached in the SGA and reused to improve performance. It queries the data from cache rather than database. Subsequent execution of same query is faster because now the data is being pulled from cache.

```
SELECT /*+ result_cache */ NUMBER FROM main_table;
```

Output -

```
Number
------
1
2
3
4
5
6
7
8
9
10

Elapsed: 00:00:02.20
```

If I run the same query again now, the time to execute will reduce since the data is now fetched from cache which was set during the first execution.

Output -

```
Number
------
1
2
3
4
5
6
7
8
9
10

Elapsed: 00:00:00.10
```

Notice how the elapsed time reduced from **2.20 seconds** to **0.10 seconds**.

> Result Cache holds the cache until the data in database is updated/altered/deleted. Any change will release the cache.

第29章：包

第29.1节：使用函数定义包头和包体

在此示例中，我们定义了一个包头和一个带有函数的包体。
之后，我们调用包中的函数，该函数返回一个返回值。

包头：

```
CREATE OR REPLACE PACKAGE SkyPkg AS

       FUNCTION GetSkyColour(vPlanet IN VARCHAR2)
       RETURN VARCHAR2;

END;
/
```

包体：

```
CREATE OR REPLACE PACKAGE BODY SkyPkg AS

        FUNCTION GetSkyColour(vPlanet IN VARCHAR2)
        RETURN VARCHAR2
        AS
vColour VARCHAR2(100) := NULL;
        BEGIN
               IF vPlanet = '地球' THEN
                       vColour := '蓝色';
               ELSIF vPlanet = '火星' THEN
                       vColour := '红色';
               END IF;

               RETURN vColour;
        END;

END;
/
```

从包体调用函数:

```
声明
vColour VARCHAR2(100);
BEGIN
vColour := SkyPkg.GetSkyColour(vPlanet => '地球');
        DBMS_OUTPUT.PUT_LINE(vColour);
END;
/
```

第29.2节：重载

包中的函数和过程可以重载。以下包**TEST**有两个名为
print_number的过程，它们根据调用时传入的参数表现不同。

```
CREATE OR REPLACE PACKAGE TEST IS
  PROCEDURE print_number(p_number IN INTEGER);
```

Chapter 29: Packages

Section 29.1: Define a Package header and body with a function

In this example we define a package header and a package body wit a function.
After that we are calling a function from the package that return a return value.

Package header:

```
CREATE OR REPLACE PACKAGE SkyPkg AS

       FUNCTION GetSkyColour(vPlanet IN VARCHAR2)
       RETURN VARCHAR2;

END;
/
```

Package body:

```
CREATE OR REPLACE PACKAGE BODY SkyPkg AS

        FUNCTION GetSkyColour(vPlanet IN VARCHAR2)
        RETURN VARCHAR2
        AS
               vColour VARCHAR2(100) := NULL;
        BEGIN
               IF vPlanet = 'Earth' THEN
                       vColour := 'Blue';
               ELSIF vPlanet = 'Mars' THEN
                       vColour := 'Red';
               END IF;

               RETURN vColour;
        END;

END;
/
```

Calling the function from the package body:

```
DECLARE
        vColour VARCHAR2(100);
BEGIN
        vColour := SkyPkg.GetSkyColour(vPlanet => 'Earth');
        DBMS_OUTPUT.PUT_LINE(vColour);
END;
/
```

Section 29.2: Overloading

Functions and procedures in packages can be overloaded. The following package **TEST** has two procedures called **print_number**, which behave differently depending on parameters they are called with.

```
CREATE OR REPLACE PACKAGE TEST IS
  PROCEDURE print_number(p_number IN INTEGER);
  PROCEDURE print_number(p_number IN VARCHAR2);
END TEST;
/
CREATE OR REPLACE PACKAGE BODY TEST IS

  过程 print_number(p_number 输入 整数) 是
  开始
    DBMS_OUTPUT.put_line('数字: ' || p_number);
  结束;

  过程 print_number(p_number 输入 VARCHAR2) 是
  开始
    DBMS_OUTPUT.put_line('字符串: ' || p_number);
  结束;

结束 TEST;
/
```

我们调用这两个过程。第一个使用整数参数，第二个使用varchar2参数。

```
设置 serveroutput 开启;
-- 调用第一个过程
执行 test.print_number(3);
-- 调用第二个过程
exec test.print_number('three');
```

上述脚本的输出是：

```
SQL>
数字：3
PL/SQL过程成功完成
字符串：three
PL/SQL过程成功完成
```

重载的限制

只有本地或打包的子程序，或类型方法，可以被重载。因此，不能重载独立子程序。此外，如果两个子程序的形式参数仅在名称或参数模式上不同，也不能重载它们

第29.3节：包的使用

PL/SQL中的包是一组过程、函数、变量、异常、常量和数据结构的集合。
通常包中的资源彼此相关，并完成类似的任务。

为什么使用包

- 模块化
- 更好的性能/功能

包的组成部分

规范 - 有时称为包头。包含变量和类型声明，以及包中函数和过程的签名，这些函数和过程是公共的，可以从包外部调用。

包体 - 包含代码和私有声明。

```
  PROCEDURE print_number(p_number IN VARCHAR2);
END TEST;
/
CREATE OR REPLACE PACKAGE BODY TEST IS

  PROCEDURE print_number(p_number IN INTEGER) IS
  BEGIN
    DBMS_OUTPUT.put_line('Digit: ' || p_number);
  END;

  PROCEDURE print_number(p_number IN VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.put_line('String: ' || p_number);
  END;

END TEST;
/
```

We call both procedures. The first with integer parameter, the second with varchar2.

```
SET serveroutput ON;
-- call the first procedure
exec test.print_number(3);
-- call the second procedure
exec test.print_number('three');
```

The output of the above script is:

```
SQL>
Digit: 3
PL/SQL procedure successfully completed
String: three
PL/SQL procedure successfully completed
```

Restrictions on Overloading

Only local or packaged subprograms, or type methods, can be overloaded. Therefore, you cannot overload standalone subprograms. Also, you cannot overload two subprograms if their formal parameters differ only in name or parameter mode

Section 29.3: Package Usage

Packages in PL/SQL are a collection of procedures, functions, variables, exceptions, constants, and data structures. Generally the resources in a package are related to each other and accomplish similar tasks.

Why Use Packages

- Modularity
- Better Performance/ Funtionality

Parts of a Package

Specification - Sometimes called a package header. Contains variable and type declarations and the signatures of the functions and procedures that are in the package which are **public** to be called from outside the package.

Package Body - Contains the code and **private** declarations.

包规范必须先于包体编译，否则包体编译时会报错。

The package specification must be compiled before the package body, otherwise the package body compilation will report an error.

第30章：异常处理

Oracle产生各种异常。你可能会惊讶于代码因某些不明确的信息而停止是多么乏味。为了提高PL/SQL代码的易修复性，有必要在最低层处理异常。除非你只是为了自己保留这段代码，不让别人维护，否则绝不要把异常“掩盖”起来。

预定义错误。

第30.1节：语法

异常部分的一般语法：

```
声明
声明部分
BEGIN
一些语句

EXCEPTION
    WHEN exception_one THEN
        DO 执行某操作
    WHEN exception_two THEN
        DO 执行某操作
    WHEN exception_three THEN
        DO 执行某操作
    WHEN  OTHERS  THEN
        DO 执行某操作
END;
```

异常部分必须位于PL/SQL块的末尾。PL/SQL允许我们嵌套块，因此每个块都可以有自己的异常部分，例如：

```
CREATE OR REPLACE PROCEDURE nested_blocks
IS
BEGIN
一些语句
    BEGIN
一些语句

    异常
        WHEN exception_one THEN
            DO 执行某操作
    END;
异常
    WHEN exception_two THEN
        DO 执行某操作
END;
```

如果嵌套块中会引发异常，则应在内部异常部分处理该异常，但如果内部
异常部分未处理该异常，则该异常将传递到外部

块的异常部分。

第30.2节：用户自定义异常

顾名思义，用户自定义异常是由用户创建的。如果你想创建自己的异常，
你必须：

Chapter 30: Exception Handling

Oracle produces a variety of exceptions. You may be surprised how tedious it can be to have your code stop with some unclear message. To improve your PL/SQL code's ability to get fixed easily it is necessary to handle exceptions at the lowest level. Never hide an exception "under the carpet", unless you're here to keep your piece of code for you only and for no one else to maintain.

The predefined errors.

Section 30.1: Syntax

The general syntax for exception section:

```
DECLARE
    declaration Section
BEGIN
    some statements

EXCEPTION
    WHEN exception_one THEN
        DO something
    WHEN exception_two THEN
        DO something
    WHEN exception_three THEN
        DO something
    WHEN OTHERS THEN
        DO something
END;
```

An exception section has to be on the end of the PL/SQL block. PL/SQL gives us the opportunity to nest blocks, then each block may have its own exception section for example:

```
CREATE OR REPLACE PROCEDURE nested_blocks
IS
BEGIN
    some statements
    BEGIN
        some statements

    EXCEPTION
        WHEN exception_one THEN
            DO something
    END;
EXCEPTION
    WHEN exception_two THEN
        DO something
END;
```

If exception will be raised in the nested block it should be handled in the inner exception section, but if inner exception section does not handle this exception then this exception will go to exception section of the external block.

Section 30.2: User defined exceptions

As the name suggest user defined exceptions are created by users. If you want to create your own exception you have to:

1.声明异常
2.在程序中引发异常
3.创建合适的异常处理程序来捕获它。

示例

我想更新所有工人的工资。但如果没有工人，则引发异常。

```
CREATE OR REPLACE PROCEDURE update_salary
IS
no_workers 异常;
    v_counter 数字 := 0;
开始
    选择 计数(*) 进入 v_counter 从 员工;
    如果 v_counter = 0 那么
        引发 no_workers;
    否则
        更新 员工 设置 薪水 = 3000;
    END IF;

    异常
        当 no_workers 时
raise_application_error(-20991,'我们没有员工!');
结束;
/
```

RAISE是什么意思？

当需要时，数据库服务器会自动引发异常，但如果你愿意，可以使用RAISE显式引发任何异常。

过程 raise_application_error(错误编号,错误信息);

- error_number 必须在 -20000 到 -20999 之间
- error_message 是发生错误时显示的消息。

第 30.3 节：内部定义的异常

内部定义的异常没有名称，但它有自己的代码。

何时使用？

如果你知道你的数据库操作可能会引发特定的异常，这些异常没有名称，那么你可以给它们命名，以便你可以专门为它们编写异常处理程序。否则，你只能将它们与 OTHERS 异常处理程序一起使用。

语法

```
声明
my_name_exc EXCEPTION;
    PRAGMA exception_init(my_name_exc,-37);
BEGIN
    ...
异常
    WHEN my_name_exc THEN
        执行 某些操作
END;
```

1. Declare the exception
2. Raise it from your program
3. Create suitable exception handler to catch him.

Example

I want to update all salaries of workers. But if there are no workers, raise an exception.

```
CREATE OR REPLACE PROCEDURE update_salary
IS
    no_workers EXCEPTION;
    v_counter NUMBER := 0;
BEGIN
    SELECT COUNT(*) INTO v_counter FROM emp;
    IF v_counter = 0 THEN
        RAISE no_workers;
    ELSE
        UPDATE emp SET salary = 3000;
    END IF;

    EXCEPTION
        WHEN no_workers THEN
            raise_application_error(-20991,'We don''t have workers!');
END;
/
```

What does it mean RAISE?

Exceptions are raised by database server automatically when there is a need, but if you want, you can raise explicitly any exception using RAISE.

Procedure raise_application_error(error_number,error_message);

- error_number must be between -20000 and -20999
- error_message message to display when error occurs.

Section 30.3: Internally defined exceptions

An internally defined exception doesn't have a name, but it has its own code.

When to use it?

If you know that your database operation might raise specific exceptions those which don't have names, then you can give them names so that you can write exception handlers specifically for them. Otherwise, you can use them only with OTHERS exception handlers.

Syntax

```
DECLARE
    my_name_exc EXCEPTION;
    PRAGMA exception_init(my_name_exc,-37);
BEGIN
    ...
EXCEPTION
    WHEN my_name_exc THEN
        DO something
END;
```

my_name_exc EXCEPTION; 这是异常名称的声明。

PRAGMA exception_init(my_name_exc,-37); 将名称分配给内部定义异常的错误代码。

示例

我们有一个 emp_id，它是 emp 表的主键，也是 dept 表的外键。如果我们尝试删除有子记录的 emp_id，将抛出代码为 -2292 的异常。

```
CREATE OR REPLACE PROCEDURE remove_employee
IS
emp_exception 异常;
    PRAGMA exception_init(emp_exception,-2292);
开始
    从 emp 删除 WHERE emp_id = 3;
异常
    当 emp_exception 时
        DBMS_OUTPUT.put_line('你不能那样做！');
结束;
/
```

> Oracle 文档中说："带有用户声明名称的内部定义异常仍然是内部定义异常，而不是用户定义异常。"

第30.4节：预定义异常

预定义异常是内部定义的异常，但它们有名称。Oracle数据库会自动引发这类异常。

示例

```
创建或替换过程 insert_emp
是
BEGIN
    插入到 emp (emp_id, ename) 值 ('1','Jon');

异常
    当 DUP_VAL_ON_INDEX 时
        DBMS_OUTPUT.put_line('索引上有重复值！');
END;
/
```

以下是异常名称及其代码示例：

异常名称	错误代码
NO_DATA_FOUND	-1403
ACCESS_INTO_NULL	-6530
CASE_NOT_FOUND	-6592
ROWTYPE_MISMATCH	-6504
TOO_MANY_ROWS	-1422
ZERO_DIVIDE	-1476

Oracle 网站上异常名称及其代码的完整列表。

my_name_exc EXCEPTION; that is the exception name declaration.

PRAGMA exception_init(my_name_exc,-37); assign name to the error code of internally defined exception.

Example

We have an emp_id which is a primary key in emp table and a foreign key in dept table. If we try to remove emp_id when it has child records, it will be thrown an exception with code -2292.

```
CREATE OR REPLACE PROCEDURE remove_employee
IS
    emp_exception EXCEPTION;
    PRAGMA exception_init(emp_exception,-2292);
BEGIN
    DELETE FROM emp WHERE emp_id = 3;
EXCEPTION
    WHEN emp_exception THEN
        DBMS_OUTPUT.put_line('You can not do that!');
END;
/
```

> Oracle documentation says: "An internally defined exception with a user-declared name is still an internally defined exception, not a user-defined exception."

Section 30.4: Predefined exceptions

Predefined exceptions are internally defined exceptions but they have names. Oracle database raise this type of exceptions automatically.

Example

```
CREATE OR REPLACE PROCEDURE insert_emp
IS
BEGIN
    INSERT INTO emp (emp_id, ename) VALUES ('1','Jon');

EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        DBMS_OUTPUT.put_line('Duplicate value on index!');
END;
/
```

Below are examples exceptions name with theirs codes:

Exception Name	Error Code
NO_DATA_FOUND	-1403
ACCESS_INTO_NULL	-6530
CASE_NOT_FOUND	-6592
ROWTYPE_MISMATCH	-6504
TOO_MANY_ROWS	-1422
ZERO_DIVIDE	-1476

Full list of exception names and their codes on Oracle web-site.

第30.5节：定义自定义异常，触发它并查看其来源

为说明这一点，下面是一个具有三种不同“错误”行为的函数

- 参数完全错误：我们使用用户定义的表达式
- 参数有拼写错误：我们使用Oracle标准的NO_DATA_FOUND错误
- 另一个未处理的情况

欢迎根据您的标准进行调整：

```
声明
this_is_not_acceptable EXCEPTION;
  PRAGMA EXCEPTION_INIT(this_is_not_acceptable, -20077);
  g_err VARCHAR2 (200) := 'to-be-defined';
w_schema all_tables.OWNER%TYPE;

  过程 get_schema( p_table 输入 VARCHAR2, p_schema 输出 VARCHAR2)
  是
w_err VARCHAR2 (200) := '待定义';
  开始
w_err := 'get_schema-步骤-1:';
    如果 (p_table = 'Delivery-Manager-Is-Silly') 则
      引发 this_is_not_acceptable;
    END IF;
w_err := 'get_schema-步骤-2:';
    选择 所有者 进入 p_schema
      从 all_tables
     哪里 table_name 类似 (p_table||'%') ;
  异常
  当 NO_DATA_FOUND 时
    -- 处理 Oracle 定义的异常
     DBMS_OUTPUT.put_line('[警告]'||w_err||'这可能发生。请检查您输入的表名。') ;
  当 this_is_not_acceptable 时
    -- 处理您的自定义错误
     DBMS_OUTPUT.put_line('[警告]'||w_err||'请不要取笑交付经理。') ;
  当其他情况时
     DBMS_OUTPUT.put_line('[ERR]'||w_err||'未处理的异常:'||SQLERRM);
     RAISE;
  END 获取模式;

BEGIN
g_err := '全局；第一次调用：';
  获取模式('交付经理很傻', w_schema);
  g_err := '全局；第二次调用：';
获取模式('AAA', w_schema);
  g_err := '全局；第三次调用：';
  获取模式('', w_schema);
g_err := '全局；第四次调用：';
获取模式('由于之前的错误，无法达到此点。', w_schema);

异常
  当其他情况时
     DBMS_OUTPUT.put_line('[错误]'||g_err||'未处理的异常：'||SQLERRM);
  -- 如果错误日志不足，您可以再次向调用者抛出此异常。
--  raise;
END;
/
```

在常规数据库上执行：

Section 30.5: Define custom exception, raise it and see where it comes from

To illustrate this, here is a function that has 3 different "wrong" behaviors

- the parameter is completely stupid: we use a user-defined expression
- the parameter has a typo: we use Oracle standard NO_DATA_FOUND error
- another, but not handled case

Feel free to adapt it to your standards:

```
DECLARE
  this_is_not_acceptable EXCEPTION;
  PRAGMA EXCEPTION_INIT(this_is_not_acceptable, -20077);
  g_err VARCHAR2 (200) := 'to-be-defined';
  w_schema all_tables.OWNER%TYPE;

  PROCEDURE get_schema( p_table IN VARCHAR2, p_schema OUT VARCHAR2)
  IS
    w_err VARCHAR2 (200) := 'to-be-defined';
  BEGIN
    w_err := 'get_schema-step-1:';
    IF (p_table = 'Delivery-Manager-Is-Silly') THEN
      RAISE this_is_not_acceptable;
    END IF;
    w_err := 'get_schema-step-2:';
    SELECT owner INTO p_schema
      FROM all_tables
     WHERE table_name LIKE(p_table||'%');
  EXCEPTION
  WHEN NO_DATA_FOUND THEN
    -- handle Oracle-defined exception
     DBMS_OUTPUT.put_line('[WARN]'||w_err||'This can happen. Check the table name you entered.');
  WHEN this_is_not_acceptable THEN
    -- handle your custom error
     DBMS_OUTPUT.put_line('[WARN]'||w_err||'Please don''t make fun of the delivery manager.');
  WHEN OTHERS THEN
     DBMS_OUTPUT.put_line('[ERR]'||w_err||'unhandled exception:'||SQLERRM);
     RAISE;
  END Get_schema;

BEGIN
  g_err := 'Global; first call:';
  get_schema('Delivery-Manager-Is-Silly', w_schema);
  g_err := 'Global; second call:';
  get_schema('AAA', w_schema);
  g_err := 'Global; third call:';
  get_schema('', w_schema);
  g_err := 'Global; 4th call:';
  get_schema('Can''t reach this point due to previous error.', w_schema);

EXCEPTION
  WHEN OTHERS THEN
     DBMS_OUTPUT.put_line('[ERR]'||g_err||'unhandled exception:'||SQLERRM);
  -- you may raise this again to the caller if error log isn't enough.
--  raise;
END;
/
```

Giving on a regular database:

```
[WARN]get_schema-step-1:请不要取笑交付经理。
[WARN]get_schema-step-2:这种情况可能发生。请检查您输入的表名。
[ERR]get_schema-step-2:未处理的异常：ORA-01422: 精确查询返回的行数超过请求的行数

[ERR]Global; third call:未处理的异常：ORA-01422: 精确查询返回的行数超过请求的行数
```

请记住，异常是用来处理*罕见*情况的。我见过有些应用程序在每次访问时都会抛出异常，只是为了请求用户密码，提示“未连接”……浪费了大量计算资源。

第30.6节：处理连接错误异常

每个标准的Oracle错误都对应一个错误编号。预见代码中可能出错的情况非常重要。这里针对连接到另一个数据库，可能出现的情况有：

- -28000 账户被锁定
- -28001 密码过期
- -28002 宽限期
- -1017 用户名或密码错误

这里有一种方法来测试数据库链接所使用的用户出了什么问题：

```
声明
v_dummy NUMBER;
BEGIN
  -- 测试数据库链接
  EXECUTE IMMEDIATE 'select COUNT(1) from dba_users@pass.world' INTO v_dummy ;
  -- 如果执行到这里，说明没有抛出异常：显示COUNT的结果
  DBMS_OUTPUT.put_line(v_dummy||' users on PASS db');

异常
  -- 异常可以通过Oracle预定义列表中的名称来引用
    WHEN LOGIN_DENIED
    THEN
        DBMS_OUTPUT.put_line('ORA-1017 / 用户名或密码无效，请重试');
    WHEN  OTHERS
    然后
  -- 或者通过它们的编号引用：自动存储在保留变量SQLCODE中
        如果 SQLCODE = '-2019'
        则
          DBMS_OUTPUT.put_line('ORA-2019 / 无效的 db_link 名称');
        否则如果 SQLCODE = '-1035'
        然后
          DBMS_OUTPUT.put_line('ORA-1035 / 数据库处于限制会话状态，请联系您的数据库管理员');

        否则如果 SQLCODE = '-28000'
        然后
          DBMS_OUTPUT.put_line('ORA-28000 / 账户被锁定，请联系您的数据库管理员');
        否则如果 SQLCODE = '-28001'
        然后
          DBMS_OUTPUT.put_line('ORA-28001 / 密码已过期，请联系您的数据库管理员更改');
        否则如果 SQLCODE  = '-28002'
        然后
          DBMS_OUTPUT.put_line('ORA-28002 / 密码已过期，请更改');
        否则
   -- 如果不是您预期的异常
          DBMS_OUTPUT.put_line('未特别处理的异常');
          DBMS_OUTPUT.put_line('Oracle 报告'||SQLCODE||':'||SQLERRM);
```

```
[WARN]get_schema-step-1:Please don't make fun of the delivery manager.
[WARN]get_schema-step-2:This can happen. Check the table name you entered.
[ERR]get_schema-step-2:unhandled exception:ORA-01422: exact fetch returns more than requested
number of rows
[ERR]Global; third call:unhandled exception:ORA-01422: exact fetch returns more than requested
number of rows
```

Remember that exception are here to handle *rare* cases. I saw applications who raised an exception at every access, just to ask for the user password, saying "not connected"... so much computation waste.

Section 30.6: Handling connexion error exceptions

Each standard Oracle error is associated with an error number. It's important to anticipate what could go wrong in your code. Here for a connection to another database, it can be:

- -28000 account is locked
- -28001 password expired
- -28002 grace period
- -1017 wrong user / password

Here is a way to test what goes wrong with the user used by the database link:

```
DECLARE
  v_dummy NUMBER;
BEGIN
  -- testing db link
  EXECUTE IMMEDIATE 'select COUNT(1) from dba_users@pass.world' INTO v_dummy ;
  -- if we get here, exception wasn't raised: display COUNT's result
  DBMS_OUTPUT.put_line(v_dummy||' users on PASS db');

EXCEPTION
  -- exception can be referred by their name in the predefined Oracle's list
    WHEN LOGIN_DENIED
    THEN
        DBMS_OUTPUT.put_line('ORA-1017 / USERNAME OR PASSWORD INVALID, TRY AGAIN');
    WHEN OTHERS
    THEN
  -- or referred by their number: stored automatically in reserved variable SQLCODE
        IF  SQLCODE = '-2019'
        THEN
          DBMS_OUTPUT.put_line('ORA-2019 / Invalid db_link name');
        ELSIF SQLCODE = '-1035'
        THEN
          DBMS_OUTPUT.put_line('ORA-1035 / DATABASE IS ON RESTRICTED SESSION, CONTACT YOUR DBA');

        ELSIF SQLCODE = '-28000'
        THEN
          DBMS_OUTPUT.put_line('ORA-28000 / ACCOUNT IS LOCKED. CONTACT YOUR DBA');
        ELSIF SQLCODE = '-28001'
        THEN
          DBMS_OUTPUT.put_line('ORA-28001 / PASSWORD EXPIRED. CONTACT YOUR DBA FOR CHANGE');
        ELSIF SQLCODE  = '-28002'
        THEN
          DBMS_OUTPUT.put_line('ORA-28002 / PASSWORD IS EXPIRED, CHANGED IT');
        ELSE
   -- and if it's not one of the exception you expected
          DBMS_OUTPUT.put_line('Exception not specifically handled');
          DBMS_OUTPUT.put_line('Oracle Said'||SQLCODE||':'||SQLERRM);
        结束 如果;
结束;
/
```

第30.7节：异常处理

1. 什么是异常？

   PL/SQL中的异常是在程序执行过程中产生的错误。

   我们有三种类型的异常：

   - 内部定义的异常
   - 预定义的异常
   - 用户定义的异常

2. 什么是异常处理？

   异常处理是一种即使出现运行时错误（例如编码错误、硬件故障）也能保持程序运行的可能性。我们避免程序突然退出。

```
        END IF;
END;
/
```

Section 30.7: Exception handling

1. What is an exception?

   Exception in PL/SQL is an error created during a program execution.

   We have three types of exceptions:

   - Internally defined exceptions
   - Predefined exceptions
   - User-defined exceptions

2. What is an exception handling?

   Exception handling is a possibility to keep our program running even if appear runtime error resulting from for example coding mistakes, hardware failures.We avoid it from exiting abruptly.

第31章：错误日志记录

第31.1节：写入数据库时的错误日志记录

为现有的EXAMPLE表创建Oracle错误日志表ERR$_EXAMPLE：

```
EXECUTE DBMS_ERRLOG.CREATE_ERROR_LOG('EXAMPLE', NULL, NULL, NULL, TRUE);
```

使用SQL进行写入操作：

```
INSERT INTO EXAMPLE (COL1) VALUES ('example')
LOG ERRORS INTO ERR$_EXAMPLE 拒绝限制无限制;
```

Chapter 31: Error logging

Section 31.1: Error logging when writing to database

Create Oracle error log table ERR$_EXAMPLE for existing EXAMPLE table:

```
EXECUTE DBMS_ERRLOG.CREATE_ERROR_LOG('EXAMPLE', NULL, NULL, NULL, TRUE);
```

Make writing operation with SQL:

```
INSERT INTO EXAMPLE (COL1) VALUES ('example')
LOG ERRORS INTO ERR$_EXAMPLE reject limit unlimited;
```

第32章：数据库链接

第32.1节：创建数据库链接

```
CREATE DATABASE LINK dblink_name
CONNECT TO remote_username
IDENTIFIED BY remote_password
USING 'tns_service_name';
```

然后可以通过以下方式访问远程数据库：

```
SELECT * FROM MY_TABLE@dblink_name;
```

要测试数据库链接连接而无需知道链接数据库中的任何对象名称，请使用
以下查询：

```
SELECT * FROM DUAL@dblink_name;
```

要为链接数据库服务显式指定域名，可将域名添加到USING语句中。
例如：

```
使用'tns_service_name.WORLD'
```

如果未明确指定域名，Oracle 将使用创建链接的数据库的域名。

Oracle 数据库链接创建文档：

- 10g: https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_5005.htm
- 11g: https://docs.oracle.com/cd/B28359_01/server.111/b28310/ds_concepts002.htm
- 12g: https://docs.oracle.com/database/121/SQLRF/statements_5006.htm#SQLRF01205

第 32.2 节：创建数据库链接

假设我们有两个数据库“ORA1”和“ORA2”。我们可以通过数据库链接从数据库
“ORA1”访问“ORA2”的对象。

前提条件：创建私有数据库链接需要`CREATE DATABASE LINK`权限。创建
公共数据库链接需要`CREATE PUBLIC DATABASE LINK`权限。

*Oracle Net必须在两个实例上都存在。

如何创建数据库链接：

从 ORA1：

```
SQL> CREATE <public> DATABASE LINK ora2 CONNECT TO user1 IDENTIFIED BY pass1 USING <tns name OF
ora2>;
```

数据库链接已创建。

现在我们已经设置了数据库链接，可以通过在 ORA1 上运行以下命令来验证：

```
SQL> SELECT name FROM V$DATABASE@ORA2; -- 应返回 ORA2
```

Chapter 32: Database Links

Section 32.1: Creating a database link

```
CREATE DATABASE LINK dblink_name
CONNECT TO remote_username
IDENTIFIED BY remote_password
USING 'tns_service_name';
```

The remote DB will then be accessible in the following way:

```
SELECT * FROM MY_TABLE@dblink_name;
```

To test a database link connection without needing to know any of the object names in the linked database, use the following query:

```
SELECT * FROM DUAL@dblink_name;
```

To explicitly specify a domain for the linked database service, the domain name is added to the `USING` statement. For example:

```
USING 'tns_service_name.WORLD'
```

If no domain name is explicitly specified, Oracle uses the domain of the database in which the link is being created.

Oracle documentation for database link creation:

- 10g: https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_5005.htm
- 11g: https://docs.oracle.com/cd/B28359_01/server.111/b28310/ds_concepts002.htm
- 12g: https://docs.oracle.com/database/121/SQLRF/statements_5006.htm#SQLRF01205

Section 32.2: Create Database Link

Let we assume we have two databases "ORA1" and "ORA2". We can access the objects of "ORA2" from database "ORA1" using a database link.

Prerequisites: For creating a private Database link you need a `CREATE DATABASE LINK` privilege. For creating a private Database link you need a `CREATE PUBLIC DATABASE LINK` privilege.

*Oracle Net must be present on both the instances.

How to create a database link:

From ORA1:

```
SQL> CREATE <public> DATABASE LINK ora2 CONNECT TO user1 IDENTIFIED BY pass1 USING <tns name OF
ora2>;
```

Database link created.

Now that we have the DB link set up, we can prove that by running the following from ORA1:

```
SQL> SELECT name FROM V$DATABASE@ORA2; -- should return ORA2
```

只要用户user1在 ORA2 上对那些对象（如下表 TABLE1）拥有SELECT权限，也可以从ORA1访问ORA2的数据库对象：

```
SELECT COUNT(*) FROM TABLE1@ORA2;
```

前提条件：

- 两个数据库必须都已启动并打开。
- 两个数据库监听器必须都已启动并运行。
- TNS 必须配置正确。
- 用户 user1 必须存在于 ORA2 数据库中，密码必须被检查和验证。
- 用户 user1 必须至少拥有 SELECT 权限，或任何其他访问 ORA2 上对象所需的权限。

You can also access the DB Objects of "ORA2" from "ORA1", given the user `user1` has the `SELECT` privilege on those objects on ORA2 (such as TABLE1 below):

```
SELECT COUNT(*) FROM TABLE1@ORA2;
```

Pre-requistes:

- Both databases must be up and running (opened).
- Both database listeners must be up and running.
- TNS must be configured correctly.
- User user1 must be present in ORA2 database, password must be checked and verified.
- User user1 must have at least the `SELECT` privilege, or any other required to access the objects on ORA2.

第33章：表分区

分区是一种将表和索引拆分成更小部分的功能。它用于提升性能并单独管理这些更小的部分。分区键是定义每行存储在哪个分区的列或列集合。官方 Oracle 文档中的分区概述

第33.1节：选择现有分区

检查模式中的现有分区

```
SELECT * FROM user_tab_partitions;
```

第33.2节：删除分区

```
ALTER TABLE table_name DROP PARTITION partition_name;
```

第33.3节：从分区中选择数据

从分区中选择数据

```
SELECT * FROM orders PARTITION(partition_name);
```

第33.4节：拆分分区

将某个分区拆分成两个分区，带有另一个新的高界限。

```
ALTER TABLE table_name 拆分 PARTITION old_partition
    在 (new_high_bound) 处 INTO (PARTITION new_partition TABLESPACE new_tablespace,
    PARTITION old_partition)
```

第33.5节：合并分区

将两个分区合并为一个分区

```
ALTER TABLE table_name
  合并 分区 first_partition, second_partition
  成单个分区  splitted_partition TABLESPACE new_tablespace
```

第33.6节：交换分区

将分区与非分区表进行交换/转换，反之亦然。这便于在数据段之间快速“移动”数据（相比于执行“insert...select”或“create table...as select”之类的操作），因为该操作是DDL（分区交换操作是数据字典的更新，不涉及实际数据的移动），而非DML（避免了大量的撤销/重做开销）。

最基本的例子：

1. 将非分区表（表 "B"）转换为分区（表 "A" 的分区）：

表 "A" 在分区 "OLD_VALUES" 中没有数据，而表 "B" 包含数据

```
ALTER TABLE "A" EXCHANGE PARTITION "OLD_VALUES" WITH TABLE "B";
```

Chapter 33: Table partitioning

Partitioning is a functionality to split tables and indexes into smaller pieces. It is used to improve performance and to manage the smaller pieces individually. The partition key is a column or a set of columns that defines in which partition each row is going to be stored. Partitioning Overview in official Oracle documentation

Section 33.1: Select existing partitions

Check existing partitions on Schema

```
SELECT * FROM user_tab_partitions;
```

Section 33.2: Drop partition

```
ALTER TABLE table_name DROP PARTITION partition_name;
```

Section 33.3: Select data from a partition

Select data from a partition

```
SELECT * FROM orders PARTITION(partition_name);
```

Section 33.4: Split Partition

Splits some partition into two partitions with another high bound.

```
ALTER TABLE table_name SPLIT PARTITION old_partition
    AT (new_high_bound) INTO (PARTITION new_partition TABLESPACE new_tablespace,
    PARTITION old_partition)
```

Section 33.5: Merge Partitions

Merge two partitions into single one

```
ALTER TABLE table_name
  MERGE PARTITIONS first_partition, second_partition
  INTO  PARTITION  splitted_partition TABLESPACE new_tablespace
```

Section 33.6: Exchange a partition

Exchange/convert a partition to a non-partitioned table and vice versa. This facilitates a fast "move" of data between the data segments (opposed to doing something like "insert...select" or "create table...as select") as the operation is DDL (the partition exchange operation is a data dictionary update without moving the actual data) and not DML (large undo/redo overhead).

Most basic examples :

1. Convert a non-partitioned table (table "B") to a partition (of table "A") :

Table "A" doesn't contain data in partition "OLD_VALUES" and table "B" contains data

```
ALTER TABLE "A" EXCHANGE PARTITION "OLD_VALUES" WITH TABLE "B";
```

结果：数据从表 "B"（操作后无数据）“移动”到分区 "OLD_VALUES"

2. 将分区转换为非分区表：

表 "A" 在分区 "OLD_VALUES" 中包含数据，而表 "B" 没有数据

```
ALTER TABLE "A" EXCHANGE PARTITION "OLD_VALUES" WITH TABLE "B";
```

结果：数据从分区 "OLD_VALUES"（操作后无数据）“移动”到表 "B"

注意：此操作有许多额外的选项、功能和限制

更多信息请参见以下链接 --->
"https://docs.oracle.com/cd/E11882_01/server.112/e25523/part_admin002.htm#i1107555"（章节 "Exchanging Partitions"）

第33.7节：哈希分区

这将创建一个按哈希分区的表，在此示例中按商店ID分区。

```
CREATE TABLE orders (
    order_nr NUMBER(15),
    user_id VARCHAR2(2),
    order_value NUMBER(15),
    store_id NUMBER(5)
)
PARTITION BY HASH(store_id) PARTITIONS 8;
```

分区的哈希数量应使用2的幂，以便获得均匀的分区大小分布。

第33.8节：范围分区

这将创建一个按范围分区的表，在此示例中按订单值分区。

```
CREATE TABLE orders (
    order_nr NUMBER(15),
    user_id VARCHAR2(2),
    order_value NUMBER(15),
    store_id NUMBER(5)
)
PARTITION BY RANGE(order_value) (
    PARTITION p1 VALUES LESS THAN(10),
    PARTITION p2 VALUES LESS THAN(40),
    PARTITION p3 VALUES LESS THAN(100),
    PARTITION p4 VALUES LESS THAN(MAXVALUE)
);
```

第33.9节：列表分区

这将创建一个按列表分区的表，在此示例中按商店ID分区。

```
CREATE TABLE orders (
    order_nr NUMBER(15),
    user_id VARCHAR2(2),
    order_value NUMBER(15),
    store_id NUMBER(5)
)
```

Result : data is "moved" from table "B" (contains no data after operation) to partition "OLD_VALUES"

2. Convert a partition to a non-partitioned table :

Table "A" contains data in partition "OLD_VALUES" and table "B" doesn't contain data

```
ALTER TABLE "A" EXCHANGE PARTITION "OLD_VALUES" WITH TABLE "B";
```

Result : data is "moved" from partition "OLD_VALUES" (contains no data after operation) to table "B"

Note : there is a quite a few additional options, features and restrictions for this operation

Further info can be found on this link --->
"https://docs.oracle.com/cd/E11882_01/server.112/e25523/part_admin002.htm#i1107555" (section "Exchanging Partitions")

Section 33.7: Hash partitioning

This creates a table partitioned by hash, in this example on store id.

```
CREATE TABLE orders (
    order_nr NUMBER(15),
    user_id VARCHAR2(2),
    order_value NUMBER(15),
    store_id NUMBER(5)
)
PARTITION BY HASH(store_id) PARTITIONS 8;
```

You should use a power of 2 for the number of hash partitions, so that you get an even distribution in partition size.

Section 33.8: Range partitioning

This creates a table partitioned by ranges, in this example on order values.

```
CREATE TABLE orders (
    order_nr NUMBER(15),
    user_id VARCHAR2(2),
    order_value NUMBER(15),
    store_id NUMBER(5)
)
PARTITION BY RANGE(order_value) (
    PARTITION p1 VALUES LESS THAN(10),
    PARTITION p2 VALUES LESS THAN(40),
    PARTITION p3 VALUES LESS THAN(100),
    PARTITION p4 VALUES LESS THAN(MAXVALUE)
);
```

Section 33.9: List partitioning

This creates a table partitioned by lists, in this example on store id.

```
CREATE TABLE orders (
    order_nr NUMBER(15),
    user_id VARCHAR2(2),
    order_value NUMBER(15),
    store_id NUMBER(5)
)
```

```
PARTITION BY LIST(store_id) (
    PARTITION p1 VALUES (1,2,3),
    PARTITION p2 VALUES(4,5,6),
    PARTITION p3 VALUES(7,8,9),
    PARTITION p4 VALUES(10,11)
);
```

第33.10节：截断分区

```
ALTER TABLE table_name TRUNCATE PARTITION partition_name;
```

第33.11节：重命名分区

```
ALTER TABLE table_name RENAME PARTITION p3 TO p6;
```

第33.12节：将分区移动到不同的表空间

```
ALTER TABLE table_name
MOVE PARTITION partition_name TABLESPACE tablespace_name;
```

第33.13节：添加新分区

```
ALTER TABLE table_name
ADD PARTITION new_partition VALUES LESS THAN(400);
```

```
PARTITION BY LIST(store_id) (
    PARTITION p1 VALUES (1,2,3),
    PARTITION p2 VALUES(4,5,6),
    PARTITION p3 VALUES(7,8,9),
    PARTITION p4 VALUES(10,11)
);
```

Section 33.10: Truncate a partition

```
ALTER TABLE table_name TRUNCATE PARTITION partition_name;
```

Section 33.11: Rename a partition

```
ALTER TABLE table_name RENAME PARTITION p3 TO p6;
```

Section 33.12: Move partition to different tablespace

```
ALTER TABLE table_name
MOVE PARTITION partition_name TABLESPACE tablespace_name;
```

Section 33.13: Add new partition

```
ALTER TABLE table_name
ADD PARTITION new_partition VALUES LESS THAN(400);
```

第34章：Oracle高级队列（AQ）

第34.1节：简单的生产者/消费者

概述

创建一个队列，我们可以向其发送消息。Oracle 会通知我们的存储过程有消息已入队并应被处理。我们还将添加一些子程序，以便在紧急情况下停止消息出队、允许再次出队，并运行一个简单的批处理作业来处理所有消息。

这些示例在 Oracle 数据库 12c 企业版版本 12.1.0.2.0 - 64 位生产环境中进行了测试。

创建队列

我们将创建一个消息类型、一个可以存放消息的队列表和一个队列。队列中的消息将首先按优先级出队，然后按入队时间排序。如果处理消息时出现任何问题且出队操作回滚，AQ 会在 3600 秒后使该消息可再次出队。它会这样做 48 次，然后将消息移至异常队列。

```
CREATE TYPE message_t AS object
   (
sender  VARCHAR2 ( 50 ),
   message VARCHAR2 ( 512 )
   );
/
-- 类型 MESSAGE_T 已编译
BEGIN DBMS_AQADM.create_queue_table(
     queue_table         => 'MESSAGE_Q_TBL',
     queue_payload_type => 'MESSAGE_T',
     sort_list           => 'PRIORITY,ENQ_TIME',
     multiple_consumers =>  FALSE,
compatible        => '10.0.0');
  END;
/
-- PL/SQL 过程成功完成。
BEGIN DBMS_AQADM.create_queue(
     queue_name           => 'MESSAGE_Q',
     queue_table          => 'MESSAGE_Q_TBL',
     queue_type           =>  0,
max_retries        =>  48,
     retry_delay          =>  3600,
     dependency_tracking =>  FALSE);
  END;
/
-- PL/SQL 过程成功完成。
```

既然我们有了放置消息的地方，现在让我们创建一个包来管理和处理队列中的消息。

```
CREATE OR REPLACE PACKAGE message_worker_pkg
IS
queue_name_c CONSTANT VARCHAR2(20) := 'MESSAGE_Q';

   -- 允许工作者处理队列中的消息
   PROCEDURE enable_dequeue;

   -- 阻止消息被处理，但仍允许创建和入队消息
```

Chapter 34: Oracle Advanced Queuing (AQ)

Section 34.1: Simple Producer/Consumer

Overview

Create a queue that we can send a message to. Oracle will notify our stored procedure that a message has been enqueued and should be worked. We'll also add some subprograms we can use in an emergency to stop messages from being deqeued, allow dequeuing again, and run a simple batch job to work through all of the messages.

These examples were tested on Oracle Database 12c Enterprise Edition Release 12.1.0.2.0 - 64bit Production.

Create Queue

We will create a message type, a queue table that can hold the messages, and a queue. Messages in the queue will be dequeued first by priority then be their enqueue time. If anything goes wrong working the message and the dequeue is rolled-back AQ will make the message available for dequeue 3600 seconds later. It will do this 48 times before moving it an exception queue.

```
CREATE TYPE message_t AS object
   (
   sender  VARCHAR2 ( 50 ),
   message VARCHAR2 ( 512 )
   );
/
-- Type MESSAGE_T compiled
BEGIN DBMS_AQADM.create_queue_table(
     queue_table        => 'MESSAGE_Q_TBL',
     queue_payload_type => 'MESSAGE_T',
     sort_list          => 'PRIORITY,ENQ_TIME',
     multiple_consumers =>  FALSE,
     compatible         => '10.0.0');
  END;
/
-- PL/SQL procedure successfully completed.
BEGIN DBMS_AQADM.create_queue(
     queue_name          => 'MESSAGE_Q',
     queue_table         => 'MESSAGE_Q_TBL',
     queue_type          =>  0,
     max_retries         =>  48,
     retry_delay         =>  3600,
     dependency_tracking =>  FALSE);
  END;
/
-- PL/SQL procedure successfully completed.
```

Now that we have a place to put the messages lets create a package to manage and work messages in the queue.

```
CREATE OR REPLACE PACKAGE message_worker_pkg
IS
   queue_name_c CONSTANT VARCHAR2(20) := 'MESSAGE_Q';

   -- allows the workers to process messages in the queue
   PROCEDURE enable_dequeue;

   -- prevents messages from being worked but will still allow them to be created and enqueued
```

```
    PROCEDURE disable_dequeue;

    -- 仅由Oracle高级队列调用。请勿在其他地方调用。
    过程 on_message_enqueued (context          输入 RAW,
                                     reginfo          输入 sys.aq$_reg_info,
                                     descr            输入 sys.aq$_descriptor,
                                     payload          输入 RAW,
                               payloadl        IN NUMBER);

    -- 允许在我们错过通知（或重试）时处理消息
    -- 正在处理中）
    过程 work_old_messages;

END;
/

创建或替换包体 message_worker_pkg
IS
    -- 当我们尝试出队但没有更多消息准备好时，Oracle 抛出的异常
    -- 此刻将被出队
   no_more_messages_ex          异常;
    PRAGMA exception_init (no_more_messages_ex,
                            -25228);

    -- 允许工作者处理队列中的消息
    过程 enable_dequeue
    作为
    BEGIN
       DBMS_AQADM.start_queue (queue_name => queue_name_c, dequeue => TRUE);
    END enable_dequeue;

    -- 阻止消息被处理，但仍允许创建和入队消息
    PROCEDURE disable_dequeue
    AS
    BEGIN
       DBMS_AQADM.stop_queue (queue_name => queue_name_c, dequeue => TRUE, enqueue => FALSE);
    END disable_dequeue;

    PROCEDURE work_message (message_in IN OUT NOCOPY message_t)
    AS
    BEGIN
       DBMS_OUTPUT.put_line ( message_in.sender || ' says ' || message_in.message );
    END work_message;

    -- 仅由Oracle高级队列调用。请勿在其他地方调用。

    过程 on_message_enqueued (context          输入 RAW,
                                     reginfo          输入 sys.aq$_reg_info,
                                     descr            输入 sys.aq$_descriptor,
                                     payload          输入 RAW,
payloadl       IN NUMBER)
    AS
       PRAGMA autonomous_transaction;
       dequeue_options_l       DBMS_AQ.dequeue_options_t;
       message_id_l            RAW (16);
message_l             message_t;
       message_properties_l    DBMS_AQ.message_properties_t;
    BEGIN
dequeue_options_l.msgid           := descr.msg_id;
       dequeue_options_l.consumer_name := descr.consumer_name;
       dequeue_options_l.wait          := DBMS_AQ.no_wait;
       DBMS_AQ.dequeue (queue_name           => descr.queue_name,
```

```
    PROCEDURE disable_dequeue;

    -- called only by Oracle Advanced Queueing.  Do not call anywhere else.
    PROCEDURE on_message_enqueued (context         IN RAW,
                                   reginfo         IN sys.aq$_reg_info,
                                   descr           IN sys.aq$_descriptor,
                                   payload         IN RAW,
                                   payloadl        IN NUMBER);

    -- allows messages to be worked if we missed the notification (or a retry
    -- is pending)
    PROCEDURE work_old_messages;

END;
/

CREATE OR REPLACE PACKAGE BODY message_worker_pkg
IS
    -- raised by Oracle when we try to dequeue but no more messages are ready to
    -- be dequeued at this moment
    no_more_messages_ex          EXCEPTION;
    PRAGMA exception_init (no_more_messages_ex,
                           -25228);

    -- allows the workers to process messages in the queue
    PROCEDURE enable_dequeue
    AS
    BEGIN
       DBMS_AQADM.start_queue (queue_name => queue_name_c, dequeue => TRUE);
    END enable_dequeue;

    -- prevents messages from being worked but will still allow them to be created and enqueued
    PROCEDURE disable_dequeue
    AS
    BEGIN
       DBMS_AQADM.stop_queue (queue_name => queue_name_c, dequeue => TRUE, enqueue => FALSE);
    END disable_dequeue;

    PROCEDURE work_message (message_in IN OUT NOCOPY message_t)
    AS
    BEGIN
       DBMS_OUTPUT.put_line ( message_in.sender || ' says ' || message_in.message );
    END work_message;

    -- called only by Oracle Advanced Queueing.  Do not call anywhere else.

    PROCEDURE on_message_enqueued (context         IN RAW,
                                   reginfo         IN sys.aq$_reg_info,
                                   descr           IN sys.aq$_descriptor,
                                   payload         IN RAW,
                                   payloadl        IN NUMBER)
    AS
       PRAGMA autonomous_transaction;
       dequeue_options_l       DBMS_AQ.dequeue_options_t;
       message_id_l            RAW (16);
       message_l               message_t;
       message_properties_l    DBMS_AQ.message_properties_t;
    BEGIN
       dequeue_options_l.msgid         := descr.msg_id;
       dequeue_options_l.consumer_name := descr.consumer_name;
       dequeue_options_l.wait          := DBMS_AQ.no_wait;
       DBMS_AQ.dequeue (queue_name           => descr.queue_name,
dequeue_options      => dequeue_options_l,
                        message_properties   => message_properties_l,
                        payload              => message_l,
msgid                => message_id_l);
        work_message (message_l);
        COMMIT ;
    异常
        WHEN no_more_messages_ex
        THEN
            -- 可能 work_old_messages 已经出列了该消息
            COMMIT;
        WHEN OTHERS
        THEN
            -- 这里不需要抛出异常。我只是想指出
            -- 由于这是由 AQ 调用的，抛出异常回去
            -- 会让它把消息放回队列并稍后重试
            RAISE;
    END on_message_enqueued;

    -- 允许在我们错过通知（或重试）时处理消息
    -- 正在处理中）
    过程 work_old_messages
    作为
        PRAGMA autonomous_transaction;
        dequeue_options_l       DBMS_AQ.dequeue_options_t;
        message_id_l            RAW (16);
message_l               message_t;
        message_properties_l    DBMS_AQ.message_properties_t;
    BEGIN
dequeue_options_l.wait        := DBMS_AQ.no_wait;
        dequeue_options_l.navigation := DBMS_AQ.first_message;

        WHILE (TRUE) LOOP -- 退出条件是 no_more_messages_ex
            DBMS_AQ.dequeue (queue_name           => queue_name_c,
                             dequeue_options      => dequeue_options_l,
                             message_properties   => message_properties_l,
                             payload              => message_l,
msgid                => message_id_l);
            work_message (message_l);
            COMMIT ;
        结束 循环;
    异常
        WHEN no_more_messages_ex
        THEN
            NULL;
    END work_old_messages;
END;
```

接下来告诉 AQ，当消息被放入 MESSAGE_Q（并提交）时，通知我们的过程它有工作要做。AQ 会在它自己的会话中启动一个作业来处理此事。

```
BEGIN
  DBMS_AQ.register (
sys.aq$_reg_info_list (
sys.aq$_reg_info (USER || '.' || message_worker_pkg.queue_name_c,
                           DBMS_AQ.namespace_aq,
                           'plsql://' || USER || '.message_worker_pkg.on_message_enqueued',
                           HEXTORAW ('FF'))),
      1);
  提交;
结束;
```

```
                        dequeue_options      => dequeue_options_l,
                        message_properties   => message_properties_l,
                        payload              => message_l,
                        msgid                => message_id_l);
      work_message (message_l);
      COMMIT;
   EXCEPTION
      WHEN no_more_messages_ex
      THEN
         -- it's possible work_old_messages already dequeued the message
         COMMIT;
      WHEN OTHERS
      THEN
         -- we don't need to have a raise here.  I just wanted to point out that
         -- since this will be called by AQ throwing the exception back to it
         -- will have it put the message back on the queue and retry later
         RAISE;
   END on_message_enqueued;

   -- allows messages to be worked if we missed the notification (or a retry
   -- is pending)
   PROCEDURE work_old_messages
   AS
      PRAGMA autonomous_transaction;
      dequeue_options_l       DBMS_AQ.dequeue_options_t;
      message_id_l            RAW (16);
      message_l               message_t;
      message_properties_l    DBMS_AQ.message_properties_t;
   BEGIN
      dequeue_options_l.wait       := DBMS_AQ.no_wait;
      dequeue_options_l.navigation := DBMS_AQ.first_message;

      WHILE (TRUE) LOOP -- way out is no_more_messages_ex
         DBMS_AQ.dequeue (queue_name           => queue_name_c,
                          dequeue_options      => dequeue_options_l,
                          message_properties   => message_properties_l,
                          payload              => message_l,
                          msgid                => message_id_l);
         work_message (message_l);
         COMMIT;
      END LOOP;
   EXCEPTION
      WHEN no_more_messages_ex
      THEN
         NULL;
   END work_old_messages;
END;
```

Next tell AQ that when a message is enqueued to MESSAGE_Q (and committed) notify our procedure it has work to do. AQ will start up a job in its own session to handle this.

```
BEGIN
   DBMS_AQ.register (
      sys.aq$_reg_info_list (
         sys.aq$_reg_info (USER || '.' || message_worker_pkg.queue_name_c,
                           DBMS_AQ.namespace_aq,
                           'plsql://' || USER || '.message_worker_pkg.on_message_enqueued',
                           HEXTORAW ('FF'))),
      1);
   COMMIT;
END;
```

启动队列并发送消息

```
声明
enqueue_options_l      DBMS_AQ.enqueue_options_t;
   message_properties_l   DBMS_AQ.message_properties_t;
   message_id_l           RAW (16);
message_l             message_t;
开始
   -- 只需执行下一行一次
   DBMS_AQADM.start_queue (queue_name => message_worker_pkg.queue_name_c, enqueue => TRUE , dequeue
=> TRUE);

message_l := 新建 message_t ( 'Jon', 'Hello, world!' );
   DBMS_AQ.enqueue (queue_name           => message_worker_pkg.queue_name_c,
                 enqueue_options       => enqueue_options_l,
message_properties   => message_properties_l,
                    payload              => message_l,
msgid               => message_id_l);
   COMMIT;
END;
```

Start Queue and Send a Message

```
DECLARE
   enqueue_options_l      DBMS_AQ.enqueue_options_t;
   message_properties_l   DBMS_AQ.message_properties_t;
   message_id_l           RAW (16);
   message_l              message_t;
BEGIN
   -- only need to do this next line ONCE
   DBMS_AQADM.start_queue (queue_name => message_worker_pkg.queue_name_c, enqueue => TRUE , dequeue
=> TRUE);

   message_l := NEW message_t ( 'Jon', 'Hello, world!' );
   DBMS_AQ.enqueue (queue_name           => message_worker_pkg.queue_name_c,
                    enqueue_options      => enqueue_options_l,
                    message_properties   => message_properties_l,
                    payload              => message_l,
                    msgid                => message_id_l);
   COMMIT;
END;
```

第35章：约束条件

第35.1节：在Oracle中使用新值更新外键

假设你有一个表，想要更改该表的主键ID。你可以使用以下脚本。
这里的主键ID是“PK_S”

```
BEGIN
  FOR i IN (SELECT a.table_name, c.column_name
              FROM user_constraints a, user_cons_columns c
             WHERE a.CONSTRAINT_TYPE = 'R'
               AND a.R_CONSTRAINT_NAME = 'PK_S'
               AND c.constraint_name = a.constraint_name) LOOP


         EXECUTE IMMEDIATE 'update ' || i.table_name || ' set ' || i.column_name ||
                      '=to_number(''1000'' || ' ||  i.column_name || ') ';




  结束 循环;

END;
```

第35.2节：在Oracle中禁用所有相关外键

假设你有表T1，它与许多表有关联，且其主键约束名为“pk_t1”
你想禁用这些外键，可以使用：

```
BEGIN
    FOR I IN (SELECT table_name, constraint_name FROM user_constraint t WHERE
r_constraint_name='pk_t1') LOOP

        EXECUTE IMMEDIATE ' alter table ' || I.table_name || ' disable constraint ' || i.constraint_name;

   END LOOP;
END;
```

Chapter 35: constraints

Section 35.1: Update foreign keys with new value in Oracle

Suppose you have a table and you want to change one of this table primary id. you can use the following scrpit.
primary ID here is "PK_S"

```
BEGIN
  FOR i IN (SELECT a.table_name, c.column_name
              FROM user_constraints a, user_cons_columns c
             WHERE a.CONSTRAINT_TYPE = 'R'
               AND a.R_CONSTRAINT_NAME = 'PK_S'
               AND c.constraint_name = a.constraint_name) LOOP


         EXECUTE IMMEDIATE 'update ' || i.table_name || ' set ' || i.column_name ||
                      '=to_number(''1000'' || ' ||  i.column_name || ') ';




  END LOOP;

END;
```

Section 35.2: Disable all related foreign keys in oracle

Suppose you have the table T1 and it has relation with many tables and its primary key constraint name is "pk_t1"
you want to disable these foreign keys you can use:

```
BEGIN
    FOR I IN (SELECT table_name, constraint_name FROM user_constraint t WHERE
r_constraint_name='pk_t1') LOOP

EXECUTE IMMEDIATE ' alter table ' || I.table_name || ' disable constraint ' || i.constraint_name;

   END LOOP;
END;
```

第36章：自主事务

第36.1节：使用自主事务记录错误

下面的过程是一个通用过程，用于将应用程序中的所有错误记录到一个公共错误日志表中。

```
CREATE OR REPLACE PROCEDURE log_errors
(
p_calling_program IN VARCHAR2,
  p_error_code IN INTEGER,
p_error_description IN VARCHAR2
)
是
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO error_log
  VALUES
  (
p_calling_program,
  p_error_code,
  p_error_description,
  SYSDATE,
  用户
  );
  COMMIT;
END log_errors;
```

下面的匿名PL/SQL块展示了如何调用log_errors过程。

```
BEGIN
   DELETE FROM dept WHERE deptno = 10;
异常处理
   当其他情况时
log_errors('删除部门',SQLCODE, SQLERRM);
      抛出异常;
END;

SELECT * FROM error_log;

调用程序     错误代码     错误描述
 错误时间           数据库用户
DELETE dept        -2292          ORA-02292: 违反完整性约束 - 子记录 存在
08/09/2016             APEX_PUBLIC_USER
```

Chapter 36: Autonomous Transactions

Section 36.1: Using autonomous transaction for logging errors

The following procedure is a generic one which will be used to log all errors in an application to a common error log table.

```
CREATE OR REPLACE PROCEDURE log_errors
(
  p_calling_program IN VARCHAR2,
  p_error_code IN INTEGER,
  p_error_description IN VARCHAR2
)
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO error_log
  VALUES
  (
  p_calling_program,
  p_error_code,
  p_error_description,
  SYSDATE,
  USER
  );
  COMMIT;
END log_errors;
```

The following anonymous PL/SQL block shows how to call the log_errors procedure.

```
BEGIN
   DELETE FROM dept WHERE deptno = 10;
EXCEPTION
   WHEN OTHERS THEN
      log_errors('Delete dept',SQLCODE, SQLERRM);
      RAISE;
END;

SELECT * FROM error_log;

CALLING_PROGRAM    ERROR_CODE    ERROR_DESCRIPTION
 ERROR_DATETIME         DB_USER
DELETE dept        -2292         ORA-02292: integrity constraint violated - child RECORD found
 08/09/2016             APEX_PUBLIC_USER
```

第37章：Oracle MAF

第37.1节：从绑定中获取值

```
ValueExpression ve = AdfmfJavaUtilities.getValueExpression(<binding>, String.class);
       String <variable_name> = (String) ve.getValue(AdfmfJavaUtilities.getELContext());
```

这里"binding"表示要从中获取值的EL表达式。

"variable_name"是用于存储从绑定中获取的值的参数名。

第37.2节：向绑定设置值

```
ValueExpression ve = AdfmfJavaUtilities.getValueExpression(<binding>, String.class);
        ve.setValue(AdfmfJavaUtilities.getELContext(), <value>);
```

这里"binding"表示要向其存储值的EL表达式。

"value"是要添加到绑定中的目标值。

第37.3节：从绑定调用方法

```
AdfELContext adfELContext = AdfmfJavaUtilities.getAdfELContext();
    MethodExpression me;
me = AdfmfJavaUtilities.getMethodExpression(<binding>, Object.class, NEW Class[] { });
    me.invoke(adfELContext, NEW Object[] { });
```

"binding"表示要调用的方法所对应的EL表达式

第37.4节：调用javaScript函数

```
AdfmfContainerUtilities.invokeContainerJavaScriptFunction(AdfmfJavaUtilities.getFeatureId(),
<function>, NEW Object[] {
                                                             });
```

"function"是要调用的目标js函数

Chapter 37: Oracle MAF

Section 37.1: To get value from Binding

```
        ValueExpression ve = AdfmfJavaUtilities.getValueExpression(<binding>, String.class);
       String <variable_name> = (String) ve.getValue(AdfmfJavaUtilities.getELContext());
```

Here "binding" indicates the EL expression from which the value is to be get.

"variable_name" the parameter to which the value from the binding to be stored

Section 37.2: To set value to binding

```
        ValueExpression ve = AdfmfJavaUtilities.getValueExpression(<binding>, String.class);
        ve.setValue(AdfmfJavaUtilities.getELContext(), <value>);
```

Here "binding" indicates the EL expression to which the value is to be stored.

"value" is the desired value to be add to the binding

Section 37.3: To invoke a method from binding

```
    AdfELContext adfELContext = AdfmfJavaUtilities.getAdfELContext();
    MethodExpression me;
    me = AdfmfJavaUtilities.getMethodExpression(<binding>, Object.class, NEW Class[] { });
    me.invoke(adfELContext, NEW Object[] { });
```

"binding" indicates the EL expression from which a method to be invoked

Section 37.4: To call a javaScript function

```
    AdfmfContainerUtilities.invokeContainerJavaScriptFunction(AdfmfJavaUtilities.getFeatureId(),
<function>, NEW Object[] {
                                                             });
```

"function" is the desired js function to be invoked

第38章：层级查询

第38.1节：生成N条记录

```
SELECT ROWNUM NO FROM DUAL CONNECT BY LEVEL <= 10
```

第38.2节：层级查询的几种用法

/* 这是一个简单的查询，可以生成一系列数字。以下示例生成从1到100的数字序列 */

```
SELECT LEVEL FROM dual CONNECT BY LEVEL <= 100;
```

/*上述查询在多种场景中非常有用，比如从给定日期生成日期序列。
以下查询生成连续的10个日期*/

```
SELECT TO_DATE('01-01-2017','mm-dd-yyyy')+level-1 AS  dates FROM dual CONNECT BY LEVEL <= 10;
```

```
2017-01-01
2017-01-02
2017-01-03
2017-01-04
2017-01-05
2017-01-06
2017-01-07
2017-01-08
2017-01-09
2017-01-10
```

Chapter 38: level query

Section 38.1: Generate N Number of records

```
SELECT ROWNUM NO FROM DUAL CONNECT BY LEVEL <= 10
```

Section 38.2: Few usages of Level Query

/* This is a simple query which can generate a sequence of numbers. The following example generates a sequence of numbers from 1..100 */

```
SELECT LEVEL FROM dual CONNECT BY LEVEL <= 100;
```

/*The above query is useful in various scenarios like generating a sequence of dates from a given date. The following query generates 10 consecutive dates */

```
SELECT TO_DATE('01-01-2017','mm-dd-yyyy')+level-1 AS  dates FROM dual CONNECT BY LEVEL <= 10;
```

```
01-JAN-17
02-JAN-17
03-JAN-17
04-JAN-17
05-JAN-17
06-JAN-17
07-JAN-17
08-JAN-17
09-JAN-17
10-JAN-17
```

第39章：使用Oracle数据库12C进行层次检索

您可以使用层次查询根据表中行之间的自然层次关系检索数据

第39.1节：使用CONNECT BY子句

```
SELECT E.EMPLOYEE_ID,E.LAST_NAME,E.MANAGER_ID FROM HR.EMPLOYEES E
CONNECT BY PRIOR E.EMPLOYEE_ID = E.MANAGER_ID;
```

使用CONNECT BY子句定义员工与经理之间的关系。

第39.2节：指定从上到下的查询方向

```
SELECT E.LAST_NAME|| ' 汇报给 ' ||
PRIOR E.LAST_NAME "Walk Top Down"
FROM HR.EMPLOYEES E
START WITH E.MANAGER_ID IS NULL
CONNECT BY PRIOR E.EMPLOYEE_ID = E.MANAGER_ID;
```

Chapter 39: Hierarchical Retrieval With Oracle Database 12C

You can use hierarchical queries to retrieve data based on a natural hierarchical relationship between rows in a table

Section 39.1: Using the CONNECT BY Caluse

```
SELECT E.EMPLOYEE_ID,E.LAST_NAME,E.MANAGER_ID FROM HR.EMPLOYEES E
CONNECT BY PRIOR E.EMPLOYEE_ID = E.MANAGER_ID;
```

The `CONNECT BY` clause to define the relationship between employees and managers.

Section 39.2: Specifying the Direction of the Query From the Top Down

```
SELECT E.LAST_NAME|| ' reports to ' ||
PRIOR E.LAST_NAME "Walk Top Down"
FROM HR.EMPLOYEES E
START WITH E.MANAGER_ID IS NULL
CONNECT BY PRIOR E.EMPLOYEE_ID = E.MANAGER_ID;
```

第40章：数据泵

以下是创建数据泵导入/导出的步骤：

第40.1节：监控数据泵作业

可以使用以下方法监控数据泵作业

1. 数据字典视图：

```
    SELECT * FROM dba_datapump_jobs;
    SELECT * FROM DBA_DATAPUMP_SESSIONS;
    SELECT username,opname,target_desc,sofar,totalwork,message FROM V$SESSION_LONGOPS WHERE username
= 'bkpadmin';
```

2. Datapump 状态：

- 从导入/导出日志或数据字典名称中记下作业名称，
- 运行 **attach** 命令：
- 在导入/导出提示符中输入 status

```
impdp <bkpadmin>/<bkp123> attach=<SYS_IMPORT_SCHEMA_01>
Import> status
```

按 CTRL+C 退出导入/导出提示符

第40.2节：步骤3/6：创建目录

```
CREATE OR REPLACE directory DATAPUMP_REMOTE_DIR AS '/oracle/scripts/expimp';
```

第40.3节：步骤7：导出命令

命令：

```
expdp <bkpadmin>/<bkp123>  parfile=<EXP.par>
```

请根据您的环境将<>中的数据替换为适当的值。您可以根据需求添加或修改参数。上述示例中，所有剩余的参数均已添加在如下所示的参数文件中：

- 导出类型：**用户导出**
- 导出整个模式
- 参数文件详情 [例如 exp.par] ：

```
schemas=<模式>
目录= DATAPUMP_REMOTE_DIR
转储文件=<数据库名>_<模式>.dmp
日志文件=exp_<数据库名>_<模式>.LOG
```

- **导出类型：大型模式的用户导出**
- 大型数据集导出整个模式：这里导出转储文件将被拆分并压缩。
  这里使用并行处理（注意：添加并行处理会增加服务器的CPU负载）

Chapter 40: Data Pump

Following are the steps to create a data pump import/export:

Section 40.1: Monitor Datapump jobs

Datapump jobs can be monitored using

1. data dictionary views:

```
    SELECT * FROM dba_datapump_jobs;
    SELECT * FROM DBA_DATAPUMP_SESSIONS;
    SELECT username,opname,target_desc,sofar,totalwork,message FROM V$SESSION_LONGOPS WHERE username
= 'bkpadmin';
```

2. Datapump status:

- Note down the job name from the import/export logs or data dictionary name and
- Run **attach** command:
- type status in Import/Export prompt

```
impdp <bkpadmin>/<bkp123> attach=<SYS_IMPORT_SCHEMA_01>
Import> status
```

Press press **CTRL+C** to come out of Import/Export prompt

Section 40.2: Step 3/6 : Create directory

```
CREATE OR REPLACE directory DATAPUMP_REMOTE_DIR AS '/oracle/scripts/expimp';
```

Section 40.3: Step 7 : Export Commands

Commands:

```
expdp <bkpadmin>/<bkp123>  parfile=<EXP.par>
```

*Please replace the data in <> with appropriate values as per your environment. You can add/modify parameters as per your requirements. In the above example all the remaining parameters are added in parameter files as stated below: *

- Export Type : **User Export**
- Export entire schema
- Parameter file details [say exp.par] :

```
schemas=<schema>
directory= DATAPUMP_REMOTE_DIR
dumpfile=<dbname>_<schema>.dmp
logfile=exp_<dbname>_<schema>.LOG
```

- Export Type : **User Export for large schema**
- Export entire schema for large datasets: Here the export dump files will be broken down and compressed. Parallelism is used here *(Note : Adding parallelism will increase the CPU load on server)*

- 参数文件详情 [例如 exp.par] ：

```
schemas=<模式>
目录= DATAPUMP_REMOTE_DIR
转储文件=<数据库名>_<模式>_%U.dmp
日志文件=exp_<数据库名>_<模式>.LOG
压缩 = 全部
并行度=5
```

- 导出类型：**表导出** [导出一组表]
- 参数文件详情 [例如 exp.par] ：

```
tables= tname1, tname2, tname3
directory= DATAPUMP_REMOTE_DIR
dumpfile=<dbname>_<schema>.dmp
logfile=exp_<dbname>_<schema>.LOG
```

第40.4节：步骤9：导入命令

前提条件：

- 在用户导入之前，最好先删除要导入的模式或表。

命令：

```
impdp <bkpadmin>/<bkp123>  parfile=<imp.par>
```

请根据您的环境将<>中的数据替换为适当的值。您可以根据需求添加或修改参数。上述示例中，所有剩余的参数均已添加在如下所示的参数文件中：

- 导入类型：**用户导入**
- 导入整个模式
- 参数文件详情[例如 imp.par]：

```
schemas=<模式>
directory= DATAPUMP_REMOTE_DIR
dumpfile=<dbname>_<schema>.dmp
logfile=imp_<dbname>_<schema>.LOG
```

- **导入类型：大型模式的用户导入**
- 导入大型数据集的整个模式：此处使用并行处理（注：添加并行处理会增加服务器的CPU负载）

- 参数文件详情[例如 imp.par]：

```
schemas=<模式>
directory= DATAPUMP_REMOTE_DIR
dumpfile=<dbname>_<schema>_%U.dmp
logfile=imp_<dbname>_<schema>.LOG
parallel=5
```

- 导入类型：表导入 [导入一组表]

- Parameter file details [say exp.par] :

```
schemas=<schema>
directory= DATAPUMP_REMOTE_DIR
dumpfile=<dbname>_<schema>_%U.dmp
logfile=exp_<dbname>_<schema>.LOG
compression = ALL
parallel=5
```

- Export Type : **Table Export** [Export set of tables]
- Parameter file details [say exp.par] :

```
tables= tname1, tname2, tname3
directory= DATAPUMP_REMOTE_DIR
dumpfile=<dbname>_<schema>.dmp
logfile=exp_<dbname>_<schema>.LOG
```

Section 40.4: Step 9 : Import Commands

Prerequisite:

- Prior to user import it is a good practice to drop the schema or table imported.

Commands:

```
impdp <bkpadmin>/<bkp123>  parfile=<imp.par>
```

*Please replace the data in <> with appropriate values as per your environment. You can add/modify parameters as per your requirements. In the above example all the remaining parameters are added in parameter files as stated below: *

- Import Type : **User Import**
- Import entire schema
- Parameter file details [say imp.par] :

```
schemas=<schema>
directory= DATAPUMP_REMOTE_DIR
dumpfile=<dbname>_<schema>.dmp
logfile=imp_<dbname>_<schema>.LOG
```

- Import Type : **User Import for large schema**
- Import entire schema for large datasets: Parallelism is used here *(Note : Adding parallelism will increase the CPU load on server)*
- Parameter file details [say imp.par] :

```
schemas=<schema>
directory= DATAPUMP_REMOTE_DIR
dumpfile=<dbname>_<schema>_%U.dmp
logfile=imp_<dbname>_<schema>.LOG
parallel=5
```

- Import Type : **Table Import** [Import set of tables]

- 参数文件详情[例如 imp.par]：

```
tables= tname1, tname2, tname3
directory= DATAPUMP_REMOTE_DIR
dumpfile=<dbname>_<schema>.dmp
logfile=exp_<dbname>_<schema>.LOG
TABLE_EXISTS_ACTION= <APPEND /SKIP /TRUNCATE /REPLACE>
```

第40.5节：Datapump步骤

源服务器 [导出数据]	**目标服务器 [导入数据]**
1. 创建一个包含导出转储文件的datapump文件夹	4. 创建一个包含导入转储文件的datapump文件夹
2. 登录将执行导出的数据库模式。	5. 登录将执行导入的数据库模式。
3. 创建指向步骤1的目录。	6. 创建指向步骤4的目录。
7. 运行导出语句。	
8. 复制/SCP转储文件到目标服务器。	
	9. 运行导入语句
	10. 检查数据，编译无效对象并提供相关权限

第40.6节：在不同模式和表空间之间复制表

```
expdp <bkpadmin>/<bkp123> directory=DATAPUMP_REMOTE_DIR dumpfile=<customer.dmp>


impdp <bkpadmin>/<bkp123> directory=DATAPUMP_REMOTE_DIR dumpfile=<customer.dmp>
remap_schema=<源模式>:<目标模式> remap_tablespace=<源表空间>:<目标
表空间>
```

- Parameter file details [say imp.par] :

```
tables= tname1, tname2, tname3
directory= DATAPUMP_REMOTE_DIR
dumpfile=<dbname>_<schema>.dmp
logfile=exp_<dbname>_<schema>.LOG
TABLE_EXISTS_ACTION= <APPEND /SKIP /TRUNCATE /REPLACE>
```

Section 40.5: Datapump steps

Source Server [Export Data]	**Target Server [Import Data]**
1. Create a datapump folder that will contain the export dump files	4. Create a datapump folder that will contain the import dump files
2. Login to database schema that will perform the export.	5. Login to database schema that will perform the import.
3. Create directory pointing to step 1.	6. Create directory pointing to step 4.
7. Run Export Statements.	
8. Copy/SCP the dump files to Target Server.	
	9. Run Import statements
	10. check data ,compile invalid objects and provide related grants

Section 40.6: Copy tables between different schemas and tablespaces

```
 expdp <bkpadmin>/<bkp123> directory=DATAPUMP_REMOTE_DIR dumpfile=<customer.dmp>


 impdp <bkpadmin>/<bkp123> directory=DATAPUMP_REMOTE_DIR dumpfile=<customer.dmp>
remap_schema=<source schema>:<target schema> remap_tablespace=<source tablespace>:<target
tablespace>
```

第41章：批量收集

第41.1节：批量数据处理

在SELECT语句中不允许使用本地集合。因此，第一步是创建一个模式级集合。如果集合不是模式级且在SELECT语句中使用，则会导致“PLS-00642: 在SQL语句中不允许使用本地集合类型”错误。

```
创建或替换类型 table1_t 为 对象 (
a_1 整数,
a_2 VARCHAR2(10)
);
```

--授予集合权限，以便在数据库中公开使用

```
授予 执行权限 于 table1_t 给 公共用户;
     创建或替换类型 table1_tbl_typ 为 表 table1_t的集合;
    授予 执行权限 于 table1_tbl_typ 给 公共用户;
```

--从表中获取数据到集合，然后遍历集合并打印数据。

```
    声明
table1_tbl table1_tbl_typ;
    开始
table1_tbl := table1_tbl_typ();
      选择 table1_t(a_1,a_2)
      批量收集到 table1_tbl
      从 table1 条件 行号<10;

     对于 记录 在 (选择 a_1 从 表(table1_tbl))--table(table1_tbl) 不会报错)
     循环
       DBMS_OUTPUT.put_line('a_1'||记录.a_1);
       DBMS_OUTPUT.put_line('a_2'||记录.a_2);
     结束循环;
    END;
/
```

Chapter 41: Bulk collect

Section 41.1: Bulk data Processing

local collections are not allowed in select statements. Hence the first step is to create a schema level collection. If the collection is not schema level and being used in SELECT statements then it would cause "PLS-00642: local collection types not allowed in SQL statements"

```
CREATE OR REPLACE TYPE table1_t IS OBJECT (
a_1 INTEGER,
a_2 VARCHAR2(10)
);
```

--Grant permissions on collection so that it could be used publically in database

```
     GRANT EXECUTE ON table1_t TO PUBLIC;
     CREATE OR REPLACE TYPE table1_tbl_typ IS TABLE OF table1_t;
     GRANT EXECUTE ON table1_tbl_typ TO PUBLIC;
```

--fetching data from table into collection and then loop through the collection and print the data.

```
    DECLARE
     table1_tbl table1_tbl_typ;
    BEGIN
     table1_tbl := table1_tbl_typ();
      SELECT table1_t(a_1,a_2)
      BULK COLLECT INTO table1_tbl
      FROM table1 WHERE ROWNUM<10;

     FOR rec IN (SELECT a_1 FROM TABLE(table1_tbl))--table(table1_tbl) won't give error)
     LOOP
       DBMS_OUTPUT.put_line('a_1'||rec.a_1);
       DBMS_OUTPUT.put_line('a_2'||rec.a_2);
     END LOOP;
    END;
/
```

第42章：真实应用安全

Oracle 实时应用安全性在 Oracle 12c 中引入。它总结了许多安全主题，如用户-角色模型、访问控制、应用程序与数据库、终端用户安全以及行级和列级安全性

第42.1节：应用程序

要将应用程序与数据库中的某些内容关联，有三个主要部分：

应用权限：应用权限描述诸如SELECT、INSERT、UPDATE、DELETE等权限
应用权限可以汇总为一个聚合权限。

```
XS$PRIVILEGE(
name=>'privilege_name'
    [, implied_priv_list=>XS$NAME_LIST('"SELECT"', '"INSERT"', '"UPDATE"', '"DELETE"')]
)

XS$PRIVILEGE_LIST(
XS$PRIVILEGE(...),
XS$PRIVILEGE(...),
    ...
);
```

应用用户：

简单应用用户：

```
BEGIN
SYS.XS_PRINCIPAL.CREATE_USER('user_name');
END;
```

直接登录应用用户：

```
BEGIN
SYS.XS_PRINCIPAL.CREATE_USER(name => 'user_name', schema => 'schema_name');
END;

BEGIN
SYS.XS_PRINCIPAL.SET_PASSWORD('user_name', 'password');
END;
创建 配置文件 prof 限制
    PASSWORD_REUSE_TIME 1/4440
    PASSWORD_REUSE_MAX 3
PASSWORD_VERIFY_FUNCTION Verify_Pass;

BEGIN
SYS.XS_PRINCIPAL.SET_PROFILE('user_name', 'prof');
END;

BEGIN
SYS.XS_PRINCIPAL.GRANT_ROLES('user_name', 'XSONNCENT');
END;
```

（可选：）

```
BEGIN
SYS.XS_PRINCIPAL.SET_VERIFIER('user_name', '6DFF060084ECE67F', XS_PRINCIPAL.XS_SHA512");
```

Chapter 42: Real Application Security

Oracle Real Application Security was introduced in Oracle 12c. It summarize many Security Topics like User-Role-Model, Access Control, Application vs. Database, End-User-Security or Row- and Column Level Security

Section 42.1: Application

To associate an Application with something in the Database there are three main parts:

Application Privilege: An Application Privilege describes Privileges like SELECT, INSERT, UPDATE, DELETE, ...
Application Privileges can be summarized as an Aggregate Privilege.

```
XS$PRIVILEGE(
    name=>'privilege_name'
    [, implied_priv_list=>XS$NAME_LIST('"SELECT"', '"INSERT"', '"UPDATE"', '"DELETE"')]
)

XS$PRIVILEGE_LIST(
    XS$PRIVILEGE(...),
    XS$PRIVILEGE(...),
    ...
);
```

Application User:

Simple Application User:

```
BEGIN
    SYS.XS_PRINCIPAL.CREATE_USER('user_name');
END;
```

Direct Login Application User:

```
BEGIN
    SYS.XS_PRINCIPAL.CREATE_USER(name => 'user_name', schema => 'schema_name');
END;

BEGIN
    SYS.XS_PRINCIPAL.SET_PASSWORD('user_name', 'password');
END;
CREATE PROFILE prof LIMIT
    PASSWORD_REUSE_TIME 1/4440
    PASSWORD_REUSE_MAX 3
    PASSWORD_VERIFY_FUNCTION Verify_Pass;

BEGIN
    SYS.XS_PRINCIPAL.SET_PROFILE('user_name', 'prof');
END;

BEGIN
    SYS.XS_PRINCIPAL.GRANT_ROLES('user_name', 'XSONNCENT');
END;
```

(optional:)

```
BEGIN
    SYS.XS_PRINCIPAL.SET_VERIFIER('user_name', '6DFF060084ECE67F', XS_PRINCIPAL.XS_SHA512“);
```

```
END;
```

应用角色：

常规应用角色：

```
声明
st_date 带时区时间戳；
    ed_date 带时区时间戳；
开始
st_date := 系统时间戳;
ed_date := TO_TIMESTAMP_TZ('2013-06-18 11:00:00 -5:00','YYYY-MM-DD HH:MI:SS');
    SYS.XS_PRINCIPAL.创建角色
        (名称 => 'app_regular_role',
       启用 => TRUE,
       开始日期 => st_date,
       结束日期 => ed_date);
结束;
```

动态应用角色：（根据认证状态动态启用）

```
BEGIN
SYS.XS_PRINCIPAL.创建动态角色
        (名称 => 'app_dynamic_role',
       持续时间 => 40,
       范围 => SYS_PRINCIPAL.SESSION_SCOPE);
结束;
```

预定义的应用角色：

常规：

- XSPUBLIC
- XSBYPASS
- XSSESSIONADMIN
- XSNAMESPACEADMIN
- XSPROVISIONER
- XSCACHEADMIN
- XSDISPATCHER

动态：（取决于应用用户的认证状态）

- DBMS_AUTH：（直接登录或其他数据库认证方法）
- EXTERNAL_DBMS_AUTH：（直接登录或其他数据库认证方法且用户为外部用户）
- DBMS_PASSWD：（使用密码直接登录）
- MIDTIER_AUTH：（通过中间层应用认证）
- XSAUTHENTICATED：（直接或中间层应用）
- XSSWITCH：（用户从代理用户切换到应用用户）

```
END;
```

Application Role:

Regular Application Role:

```
DECLARE
    st_date TIMESTAMP WITH TIME ZONE;
    ed_date TIMESTAMP WITH TIME ZONE;
BEGIN
    st_date := SYSTIMESTAMP;
    ed_date := TO_TIMESTAMP_TZ('2013-06-18 11:00:00 -5:00','YYYY-MM-DD HH:MI:SS');
    SYS.XS_PRINCIPAL.CREATE_ROLE
        (name => 'app_regular_role',
        enabled => TRUE,
        start_date => st_date,
        end_date => ed_date);
END;
```

Dynamic Application Role: (gets enabled dynamical based on the authenatication state)

```
BEGIN
    SYS.XS_PRINCIPAL.CREATE_DYNAMIC_ROLE
        (name => 'app_dynamic_role',
        duration => 40,
        scope => XS_PRINCIPAL.SESSION_SCOPE);
END;
```

Predefined Application Roles:

Regular:

- XSPUBLIC
- XSBYPASS
- XSSESSIONADMIN
- XSNAMESPACEADMIN
- XSPROVISIONER
- XSCACHEADMIN
- XSDISPATCHER

Dynamic: (depended on the authentication state of application user)

- DBMS_AUTH: (direct-logon or other database authentication method)
- EXTERNAL_DBMS_AUTH: (direct-logon or other database authentication method and user is external)
- DBMS_PASSWD: (direct-logon with password)
- MIDTIER_AUTH: (authentication through middle tier application)
- XSAUTHENTICATED: (direct or middle tier application)
- XSSWITCH: (user switched from proxy user to application user)

第43章：赋值模型和语言

第43.1节：PL/SQL中的赋值模型

所有编程语言都允许我们给变量赋值。通常，值赋给变量，变量位于左侧。任何现代编程语言中赋值操作的原型如下所示：

```
左操作数 赋值操作符 右操作数 停止指令
```

这将把右操作数赋值给左操作数。在PL/SQL中，这个操作看起来是这样的：

```
左操作数 := 右操作数;
```

左操作数必须始终是变量。右操作数可以是值、变量或函数：

```
SET serveroutput ON
DECLARE
v_hello1 VARCHAR2(32767);
v_hello2 VARCHAR2(32767);
v_hello3 VARCHAR2(32767);
  FUNCTION hello RETURN VARCHAR2 IS BEGIN RETURN 'Hello from a function!'; END;
BEGIN
   -- 来自一个值（字符串字面量）
v_hello1  := '来自变量的问候！';
   -- 来自变量
v_hello2 := v_hello1;
  -- 来自函数
v_hello3 := hello;

  DBMS_OUTPUT.put_line(v_hello1);
  DBMS_OUTPUT.put_line(v_hello2);
  DBMS_OUTPUT.put_line(v_hello3);
END;
/
```

当代码块在 SQL*Plus 中执行时，控制台会打印以下输出：

```
来自值的问候！
来自值的问候！
来自函数的问候！
```

PL/SQL 中有一个特性允许我们进行“从右到左”的赋值。这可以在 SELECT INTO
语句中实现。该语句的原型如下：

```
SELECT [ 字面量 | 列值 ]

INTO 局部变量

FROM [ 表名 | 表别名 ]

WHERE 比较条件;
```

Chapter 43: Assignments model and language

Section 43.1: Assignments model in PL/SQL

All programming languages allow us to assign values to variables. Usually, a value is assigned to variable, standing on left side. The prototype of the overall assignment operations in any contemporary programming language looks like this:

```
left_operand assignment_operand right_operand instructions_of_stop
```

This will assign right operand to the left operand. In PL/SQL this operation looks like this:

```
left_operand := right_operand;
```

Left operand **must be always a variable**. Right operand can be value, variable or function:

```
SET serveroutput ON
DECLARE
  v_hello1 VARCHAR2(32767);
  v_hello2 VARCHAR2(32767);
  v_hello3 VARCHAR2(32767);
  FUNCTION hello RETURN VARCHAR2 IS BEGIN RETURN 'Hello from a function!'; END;
BEGIN
   -- from a value (string literal)
  v_hello1 := 'Hello from a value!';
   -- from variable
  v_hello2 := v_hello1;
  -- from function
  v_hello3 := hello;

  DBMS_OUTPUT.put_line(v_hello1);
  DBMS_OUTPUT.put_line(v_hello2);
  DBMS_OUTPUT.put_line(v_hello3);
END;
/
```

When the code block is executed in SQL*Plus the following output is printed in console:

```
Hello from a value!
Hello from a value!
Hello from a function!
```

There is a feature in PL/SQL that allow us to assign "from right to the left". It's possible to do in SELECT INTO statement. Prototype of this instrunction you will find below:

```
SELECT [ literal | column_value ]

INTO local_variable

FROM [ table_name | aliastable_name ]

WHERE comparison_instructions;
```

该代码将字符字面量赋值给局部变量：

```
SET serveroutput ON
DECLARE
v_hello VARCHAR2(32767);
BEGIN
  SELECT 'Hello world!'
  INTO v_hello
  FROM dual;

  DBMS_OUTPUT.put_line(v_hello);
END;
/
```

当代码块在 SQL*Plus 中执行时，控制台会打印以下输出：

```
Hello world!
```

“从右向左”赋值不是标准语法，但对程序员和用户来说是一个有价值的特性。通常
当程序员在PL/SQL中使用游标时会用到这种技术——当我们想要从SQL游标的一行中返回单个
标量值或一组列时，会使用这种方法。

进一步阅读：

- 给变量赋值

This code will assign character literal to a local variable:

```
SET serveroutput ON
DECLARE
  v_hello VARCHAR2(32767);
BEGIN
  SELECT 'Hello world!'
  INTO v_hello
  FROM dual;

  DBMS_OUTPUT.put_line(v_hello);
END;
/
```

When the code block is executed in SQL*Plus the following output is printed in console:

```
Hello world!
```

Asignment "from right to the left" **is not a standard**, but it's valuable feature for programmers and users. Generally it's used when programmer is using cursors in PL/SQL - this technique is used, when we want to return a single scalar value or set of columns in the one line of cursor from SQL cursor.

Further Reading:

- Assigning Values to Variables

第44章：触发器

引言：

触发器是PL/SQL中的一个有用概念。触发器是一种特殊类型的存储过程，用户无需显式调用。它是一组指令，当对特定表或关系进行特定数据修改操作，或满足某些指定条件时，会自动触发执行。触发器有助于维护数据的完整性和安全性。它们通过自动执行所需操作，使工作更加方便。

第44.1节：插入或更新前触发器

```
CREATE OR REPLACE TRIGGER CORE_MANUAL_BIUR
  BEFORE INSERT OR UPDATE ON CORE_MANUAL
  FOR EACH ROW
BEGIN
  IF inserting THEN
    -- 仅当未指定时设置当前日期
    IF :NEW.created IS NULL THEN
      :NEW.created := SYSDATE;
    END IF;
  END IF;

  -- 始终将修改日期设置为当前时间
  IF inserting OR updating THEN
    :NEW.modified := SYSDATE;
  END IF;
END;
/
```

Chapter 44: Triggers

Introduction:

Triggers are a useful concept in PL/SQL. A trigger is a special type of stored procedure which does not require to be explicitly called by the user. It is a group of instructions, which is automatically fired in response to a specific data modification action on a specific table or relation, or when certain specified conditions are satisfied. Triggers help maintain the integrity, and security of data. They make the job convenient by taking the required action automatically.

Section 44.1: Before INSERT or UPDATE trigger

```
CREATE OR REPLACE TRIGGER CORE_MANUAL_BIUR
  BEFORE INSERT OR UPDATE ON CORE_MANUAL
  FOR EACH ROW
BEGIN
  IF inserting THEN
    -- only set the current date if it is not specified
    IF :NEW.created IS NULL THEN
      :NEW.created := SYSDATE;
    END IF;
  END IF;

  -- always set the modified date to now
  IF inserting OR updating THEN
    :NEW.modified := SYSDATE;
  END IF;
END;
/
```

第45章：动态SQL

动态SQL允许你在运行时组装SQL查询代码。这种技术有一些缺点
必须非常小心地使用。同时，它允许你实现更复杂的逻辑。PL/SQL
要求代码中使用的所有对象在编译时必须存在且有效。这就是为什么你不能
直接在PL/SQL中执行DDL语句，但动态SQL允许你这样做。

第45.1节：使用动态SQL选择值

假设用户想从不同的表中选择数据。表由用户指定。

```
FUNCTION get_value(p_table_name VARCHAR2, p_id NUMBER) RETURN VARCHAR2 IS
   VALUE VARCHAR2(100);
  BEGIN
    EXECUTE IMMEDIATE 'select column_value from ' || p_table_name ||
                      ' where id = :P' INTO VALUE USING p_id;
    RETURN VALUE;
  END;
```

像平常一样调用此函数：

```
声明
table_name VARCHAR2(30) := 'my_table';
  id NUMBER := 1;
BEGIN
  DBMS_OUTPUT.put_line(get_value(table_name, id));
END;
```

测试用表：

```
CREATE TABLE my_table (id NUMBER, column_value VARCHAR2(100));
INSERT INTO my_table VALUES (1, 'Hello, world!');
```

第45.2节：在动态SQL中插入值

下面的示例将值插入到前面示例中的表中：

```
声明
query_text VARCHAR2(1000) := 'insert into my_table(id, column_value) values (:P_ID, :P_VAL)';
  id NUMBER := 2;
  VALUE VARCHAR2(100) := 'Bonjour!';
BEGIN
  EXECUTE IMMEDIATE query_text USING id, VALUE;
END;
/
```

第45.3节：在动态SQL中更新值

让我们更新第一个示例中的表：

```
声明
query_text VARCHAR2(1000) := 'update my_table set column_value = :P_VAL where id = :P_ID';
  id NUMBER := 2;
  VALUE VARCHAR2(100) := 'Bonjour le monde!';
BEGIN
```

Chapter 45: Dynamic SQL

Dynamic SQL allows you to assemble an SQL query code in the runtime. This technique has some disadvantages and have to be used very carefully. At the same time, it allows you to implement more complex logic. PL/SQL requires that all objects, used in the code, have to exist and to be valid at compilation time. That's why you can't execute DDL statements in PL/SQL directly, but dynamic SQL allows you to do that.

Section 45.1: Select value with dynamic SQL

Let's say a user wants to select data from different tables. A table is specified by the user.

```
FUNCTION get_value(p_table_name VARCHAR2, p_id NUMBER) RETURN VARCHAR2 IS
   VALUE VARCHAR2(100);
  BEGIN
    EXECUTE IMMEDIATE 'select column_value from ' || p_table_name ||
                      ' where id = :P' INTO VALUE USING p_id;
    RETURN VALUE;
  END;
```

Call this function as usual:

```
DECLARE
  table_name VARCHAR2(30) := 'my_table';
  id NUMBER := 1;
BEGIN
  DBMS_OUTPUT.put_line(get_value(table_name, id));
END;
```

Table to test:

```
CREATE TABLE my_table (id NUMBER, column_value VARCHAR2(100));
INSERT INTO my_table VALUES (1, 'Hello, world!');
```

Section 45.2: Insert values in dynamic SQL

Example below inserts value into the table from the previous example:

```
DECLARE
  query_text VARCHAR2(1000) := 'insert into my_table(id, column_value) values (:P_ID, :P_VAL)';
  id NUMBER := 2;
  VALUE VARCHAR2(100) := 'Bonjour!';
BEGIN
  EXECUTE IMMEDIATE query_text USING id, VALUE;
END;
/
```

Section 45.3: Update values in dynamic SQL

Let's update table from the first example:

```
DECLARE
  query_text VARCHAR2(1000) := 'update my_table set column_value = :P_VAL where id = :P_ID';
  id NUMBER := 2;
  VALUE VARCHAR2(100) := 'Bonjour le monde!';
BEGIN
```

```
  EXECUTE IMMEDIATE query_text USING VALUE, id;
END;
/
```

第45.4节：执行DDL语句

此代码创建表：

```
BEGIN
  EXECUTE IMMEDIATE 'create table my_table (id number, column_value varchar2(100))';
END;
/
```

第45.5节：执行匿名块

您可以执行匿名块。此示例还展示了如何从动态SQL返回值：

```
声明
query_text VARCHAR2(1000) := 'begin :P_OUT := cos(:P_IN); end;';
  in_value NUMBER := 0;
out_value NUMBER;
BEGIN
  EXECUTE IMMEDIATE query_text USING OUT out_value, IN in_value;
  DBMS_OUTPUT.put_line('匿名块的结果: ' || TO_CHAR(out_value));
END;
/
```

```
  EXECUTE IMMEDIATE query_text USING VALUE, id;
END;
/
```

Section 45.4: Execute DDL statement

This code creates the table:

```
BEGIN
  EXECUTE IMMEDIATE 'create table my_table (id number, column_value varchar2(100))';
END;
/
```

Section 45.5: Execute anonymous block

You can execute anonymous block. This example shows also how to return value from dynamic SQL:

```
DECLARE
  query_text VARCHAR2(1000) := 'begin :P_OUT := cos(:P_IN); end;';
  in_value NUMBER := 0;
  out_value NUMBER;
BEGIN
  EXECUTE IMMEDIATE query_text USING OUT out_value, IN in_value;
  DBMS_OUTPUT.put_line('Result of anonymous block: ' || TO_CHAR(out_value));
END;
/
```

鸣谢

非常感谢所有来自Stack Overflow Documentation的人员帮助提供此内容，
更多更改可发送至web@petercv.com以发布或更新新内容

艾哈迈德·穆罕默德	第13章
阿列克谢	第9章和第28章
阿南德·拉杰	第37章
阿尔卡迪乌斯·卢卡谢维奇	第21章
B·萨梅迪	第9章
巴赫蒂亚尔·哈桑	第9章
本·H	第42章
鲍勃·C	第33章
卡洛斯·B	第6、11、32和33章
克里斯·赫普	第23章
达莱克斯	第10章
丹尼尔·朗格曼	第9章和第32章
迪尼杜·赫瓦格	第2章和第4章
dipdapdop	第25章
德米特里	第45章
多鲁克	第4章
埃里克·B.	第11章
埃尔坎·哈斯普拉特	第9章
叶夫根尼·K.	第18章
弗洛林·吉塔	第7、11和28章
弗朗切斯科·塞拉	第11章
g00dy	第32章
hflzh	第11章
冰	第30章
ivanzg	第33章
J. Chomel	第1、11、22、25和30章
J.Hudler	第11章
JDro04	第4、17和29章
杰弗里·肯普	第1、7、11和20章
JeromeFr	第10章
jiri.hofman	第29章和第44章
乔恩·埃里克森	第1章
乔恩·赫勒	第28章
乔恩·特里奥	第34章
朱坎	第25章
卡米尔·伊斯拉莫夫	第33章
kasi	第32章
凯卡尔	第4章
利·里弗尔	第19章
路易斯·G.	第11章
m.misiorny	第2、24和43章
马克·斯图尔特	第1、5、7、11和28章
马丁·沙彭东克	第13章
massko	第12章和第24章
马塔斯·瓦伊特凯维休斯	第13章和第18章
数学家	第16章
MT0	第6章、第14章和第21章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

Ahmed Mohamed	Chapter 13
Aleksej	Chapters 9 and 28
Anand Raj	Chapter 37
Arkadiusz Łukasiewicz	Chapter 21
B Samedi	Chapter 9
Bakhtiar Hasan	Chapter 9
Ben H	Chapter 42
BobC	Chapter 33
carlosb	Chapters 6, 11, 32 and 33
Chris Hep	Chapter 23
Dalex	Chapter 10
Daniel Langemann	Chapters 9 and 32
Dinidu Hewage	Chapters 2 and 4
dipdapdop	Chapter 25
Dmitry	Chapter 45
Doruk	Chapter 4
Eric B.	Chapter 11
Erkan Haspulat	Chapter 9
Evgeniy K.	Chapter 18
Florin Ghita	Chapters 7, 11 and 28
Francesco Serra	Chapter 11
g00dy	Chapter 32
hflzh	Chapter 11
Ice	Chapter 30
ivanzg	Chapter 33
J. Chomel	Chapters 1, 11, 22, 25 and 30
J.Hudler	Chapter 11
JDro04	Chapters 4, 17 and 29
Jeffrey Kemp	Chapters 1, 7, 11 and 20
JeromeFr	Chapter 10
jiri.hofman	Chapters 29 and 44
Jon Ericson	Chapter 1
Jon Heller	Chapter 28
Jon Theriault	Chapter 34
Jucan	Chapter 25
Kamil Islamov	Chapter 33
kasi	Chapter 32
Kekar	Chapter 4
Leigh Riffel	Chapter 19
Luis G.	Chapter 11
m.misiorny	Chapters 2, 24 and 43
Mark Stewart	Chapters 1, 5, 7, 11 and 28
Martin Schapendonk	Chapter 13
massko	Chapters 12 and 24
Matas Vaitkevicius	Chapters 13 and 18
mathguy	Chapter 16
MT0	Chapters 6, 14 and 21

诺盖拉·小	第15章
潘乔	第5章
phonetic_man	第36章
海盗X	第28章
ppeterka	第18章
普拉纳夫·沙阿	第9、18和26章
普拉尚特·米什拉	第2和41章
罗宾·詹姆斯	第9章
罗曼	第6章
桑杰·拉达迪亚	第13章和第38章
天网	第3章
斯拉瓦·巴宾	第5章和第8章
smshafiqulislam	第27章
斯里尼夫	第9章、第11章、第15章和第26章
SSD	第35章
史蒂文·福尔斯坦	第1章
StewS2	第29章
技术爱好者	第38章
丹增	第2章和第29章
雷霆	第11章
tonirush	第6、7、13和33章
托尼·安德鲁斯	第30章
trincot	第13章
user272735	第2和43章
user3216906	第17章
瓦希德	第39章
维迪亚·托坦加雷	第40章
威廉·罗伯逊	第4章
佐哈尔·埃尔卡亚姆	第3章
zygimantus	第31章

Nogueira Jr	Chapter 15
Pancho	Chapter 5
phonetic_man	Chapter 36
Pirate X	Chapter 28
ppeterka	Chapter 18
Pranav Shah	Chapters 9, 18 and 26
Prashant Mishra	Chapters 2 and 41
Robin James	Chapter 9
Roman	Chapter 6
Sanjay Radadiya	Chapters 13 and 38
Skynet	Chapter 3
Slava Babin	Chapters 5 and 8
smshafiqulislam	Chapter 27
SriniV	Chapters 9, 11, 15 and 26
SSD	Chapter 35
Steven Feuerstein	Chapter 1
StewS2	Chapter 29
TechEnthusiast	Chapter 38
Tenzin	Chapters 2 and 29
Thunder	Chapter 11
tonirush	Chapters 6, 7, 13 and 33
Tony Andrews	Chapter 30
trincot	Chapter 13
user272735	Chapters 2 and 43
user3216906	Chapter 17
Vahid	Chapter 39
Vidya Thotangare	Chapter 40
William Robertson	Chapter 4
Zohar Elkayam	Chapter 3
zygimantus	Chapter 31