

PL/SQL 学校课程笔记

ORACLE

徐鸣飞

2023 年 12 月 27 日

目录

第一章 导言	1
1.1 常见数据库	1
1.1.1 关系型数据库	1
1.1.2 NoSQL 数据库	2
1.2 数据库发展趋势	3
1.2.1 1960s-1980s: 层次数据库	3
1.2.2 1980s-2000s: 实体关系数据库	3
1.2.3 2000s-2020s: NoSQL	4
1.2.4 2020s-未知: 图数据库	5
1.3 SQL*Plus 工具	5
1.3.1 介绍	5
1.3.2 交互过程	6
1.3.3 Oracle 数据库连接命令	7
1.3.4 常用编辑命令	7
1.3.5 SQL 文件存取命令	10
1.3.6 输出保存命令	10
第二章 PL/SQL 概述	11
2.1 介绍	11
2.2 优点	11
2.3 执行体系	12
第三章 程序块结构	14
3.1 程序块	14

3.2 程序块的嵌套	16
第四章 变量	18
4.1 变量命名规则	18
4.2 变量声明语法	18
4.3 变量数据类型	19
4.4 变量命名建议	20
4.5 代替变量和绑定变量	21
4.5.1 代替变量	21
4.5.2 绑定变量	22
4.5.3 不同	22
4.6 变量作用域	23
4.6.1 局部变量 (Local Variables)	23
4.6.2 嵌套局部变量 (Nested Local Variables)	23
4.6.3 全局变量 (Global Variables)	24
4.7 布尔变量与布尔表达式	24
4.7.1 布尔操作符 (结果为布尔类型)	25
4.8 LOB 变量	25
4.8.1 LOB 变量类型	25
4.8.2 LOB 变量使用	26
4.9 赋值: 字符串分隔符 (引号限定符)	27
4.10 类型声明: %TYPE 属性	28
第五章 词法单元	29
5.1 标识符 (Identifiers)	29
5.1.1 标识符命名惯例	30
5.2 定界符 (Delimiters)	30
5.2.1 简单定界符	30
5.2.2 组合定界符	30
5.3 文字 (Literals)	31
5.4 注释 (Comments)	31
5.5 程序设计指导原则	31

目 录	III
第六章 运算符	33
6.1 基本运算符	33
6.2 空值运算	33
第七章 预定义函数	34
7.1 单行数字函数	34
7.2 单行字符函数	35
7.3 数据类型转换函数	35
7.4 日期函数	36
7.5 CHR 函数	36
第八章 序列	38
8.1 序列概念	38
8.2 PL/SQL 中序列的操作	39
8.2.1 NEXTVAL	39
8.2.2 CURRVAL	39
第九章 SELECT	40
9.1 INTO	40
9.2 变量与列同名问题	41
第十章 DML	43
10.1 概述	43
10.2 DML 代码示例	44
10.2.1 INSERT	44
10.2.2 UPDATE	44
10.2.3 DELETE	44
10.3 MERGE	45
10.3.1 MERGE 基本语法	45
10.3.2 MERGE 示例代码	45
第十一章 条件语句	47
11.1 布尔条件	47

11.1.1 各类 AND 运算结果	47
11.1.2 各类 OR 运算结果	47
11.1.3 各类 NOT 运算结果	48
11.2 IF 语句	48
11.2.1 IF 语句语法	48
11.2.2 IF 语句示例	49
11.2.3 注意	50
11.3 CASE 表达式	50
11.3.1 CASE 表达式语法	50
11.3.2 CASE 表达式示例	51
11.4 CASE 语句	51
11.4.1 CASE 语句示例代码	51
11.4.2 使用 CASE 语句取代 DECODE	52
11.5 GOTO	53
11.5.1 用处	53
第十二章 循环语句	55
12.1 基本循环	55
12.1.1 基本循环语法	55
12.1.2 基本循环示例	56
12.2 WHILE 循环	57
12.2.1 WHILE 循环语法	57
12.2.2 WHILE 循环示例	57
12.3 FOR 循环	58
12.3.1 FOR 循环语法	58
12.3.2 FOR 循环示例	59
12.4 循环的嵌套与标号	59
12.4.1 介绍	59
12.4.2 循环的嵌套与标号示例	60
12.5 CONTINUE	60
12.5.1 介绍	60

12.5.2 CONTINUE 示例	61
第十三章 组合数据类型	62
13.1 PL/SQL 记录 (records)	62
13.1.1 介绍	62
13.1.2 创建记录语法	63
13.1.3 创建记录示例代码	64
13.1.4 %ROWTYPE	65
13.2 INDEX BY 表	66
13.2.1 INDEX BY 表创建语法	67
13.2.2 INDEX BY 表使用示例	67
13.2.3 INDEX BY 表方法	69
13.2.4 INDEX BY 记录表	70
第十四章 异常处理	71
14.1 抛出异常	72
14.2 异常处理方式	72
14.2.1 捕获异常	72
14.2.2 传播异常	74
14.3 异常类型	75
14.3.1 处理预定义的 Oracle 服务器错误	75
14.3.2 处理非预定义的 Oracle 服务器错误	76
14.3.3 处理用户定义的错误	77
14.4 捕获异常实例	78
14.5 SQLCODE 和 SQLERRM	80
14.6 处理异常建议	80
第十五章 过程	82
15.1 模块化与分层	83
15.2 子程序	84
15.3 过程	86
15.3.1 过程语法	86

15.3.2 过程参数数据类型	87
15.3.3 过程参数的参数模式	87
15.3.4 绑定变量获得过程的 OUT 输出	89
15.3.5 在过程中处理异常	89
15.3.6 在过程中抛出异常	90
15.4 实参的传递方式	90
第十六章 游标	92
16.1 游标类型	92
16.2 游标的操作过程	92
16.3 游标逐行处理数据示例	93

第一章 导言

1.1 常见数据库

422 systems in ranking, September 2023

Rank			DBMS	Database Model	Score		
Sep 2023	Aug 2023	Sep 2022			Sep 2023	Aug 2023	Sep 2022
1.	1.	1.	Oracle +	Relational, Multi-model i	1240.88	-1.22	+2.62
2.	2.	2.	MySQL +	Relational, Multi-model i	1111.49	-18.97	-100.98
3.	3.	3.	Microsoft SQL Server +	Relational, Multi-model i	902.22	-18.60	-24.08
4.	4.	4.	PostgreSQL +	Relational, Multi-model i	620.75	+0.37	+0.29
5.	5.	5.	MongoDB +	Document, Multi-model i	439.42	+4.93	-50.21
6.	6.	6.	Redis +	Key-value, Multi-model i	163.68	+0.72	-17.79
7.	7.	7.	Elasticsearch	Search engine, Multi-model i	138.98	-0.94	-12.46
8.	8.	8.	IBM Db2	Relational, Multi-model i	136.72	-2.52	-14.67
9.	↑ 10.	↑ 10.	SQLite +	Relational	129.20	-0.72	-9.62
10.	↓ 9.	↓ 9.	Microsoft Access	Relational	128.56	-1.78	-11.47
11.	11.	↑ 13.	Snowflake +	Relational	120.89	+0.27	+17.39
12.	12.	↓ 11.	Cassandra +	Wide column, Multi-model i	110.06	+2.67	-9.06
13.	13.	↓ 12.	MariaDB +	Relational, Multi-model i	100.45	+1.80	-9.70
14.	14.	14.	Splunk	Search engine	91.40	+2.42	-2.65
15.	↑ 16.	↑ 16.	Microsoft Azure SQL Database	Relational, Multi-model i	82.73	+3.22	-1.69
16.	↓ 15.	↓ 15.	Amazon DynamoDB +	Multi-model i	80.91	-2.64	-6.51

1.1.1 关系型数据库

MySQL

MySQL 是一款开源的关系型数据库管理系统（RDBMS），由瑞典 MySQL AB 公司开发，现由 Oracle 公司维护。以其轻量级、高效、快速的特性而闻名，适用于中小型应用和网站。MySQL 支持多平台运行，包括 Windows、Linux、macOS 等，提供多种存储引擎如 InnoDB、MyISAM 等，具备良好的扩展性和广泛的社区支持。作为开源软件，MySQL 允许用户免费获取、使用和修改其源代码，成为广泛应用于 Web 开发和其他应用场景的可靠数据库解决方案。

Oracle

Oracle 是一款商业性质的关系型数据库管理系统（RDBMS），由 Oracle 公司开发。以其丰富的功能集、高级事务管理、安全性和卓越的性能而著称，适用于大型企业级应用。Oracle 数据库支持复杂查询、分布式数据库、以及高并发环境下的大规模数据处理，具备出色的可伸缩性和性能优化特性。作为商业软件，Oracle 提供了专业的技术支持、认证体系和咨询服务，成为众多大型组织和企业信赖的数据库解决方案。

1.1.2 NoSQL 数据库

文档型数据库：MongoDB

MongoDB 是一种非关系型数据库管理系统（NoSQL DBMS），以其面向文档的存储模型而著称。由 MongoDB 公司开发，采用分布式架构和灵活的模式设计，适用于处理大量非结构化或半结构化的数据。MongoDB 的数据存储形式为 BSON（Binary JSON），支持动态模式，使得数据存储和查询更加灵活。其强大的横向扩展性和自动分片功能使得 MongoDB 适用于大规模数据存储和处理，尤其在 Web 应用、大数据和实时分析等场景中表现出色。由于其开源特性，MongoDB 拥有庞大的社区支持，为开发人员提供了丰富的资源和工具。

搜索引擎：Elasticsearch

Elasticsearch 是一种分布式数据库管理系统，专注于搜索和分析大规模数据。作为开源软件，Elasticsearch 构建在 Apache Lucene 之上，采用文档导向的存储模型，以 JSON 格式存储数据。其核心能力包括实时搜索、结构化查询和复杂分析，使其成为处理实时数据和日志、构建全文搜索引擎以及进行大规模数据分析的理想选择。通过支持分布式架构，Elasticsearch 实现了水平扩展，能够应对高负载和大规模数据存储需求。该系统与 Kibana 等工具的整合形成了 ELK 堆栈，为用户提供了全面的数据管理、搜索和可视化解决方案。

key-value 数据库：redis

Redis 是一款开源、高性能的键值对存储系统，属于 NoSQL 数据库的一种。作为内存数据库，Redis 将数据存储在内存中，提供了快速的读写访问速度，适用于对性能有严格要求的场景。其特色包括支持多种数据结构（如字符串、哈希表、列表、集合等），原子性操作，发布订阅机制等。虽然主要用于缓存、会话管理和实时数据分析等领域，但由于其快速响应和可持久

化存储的能力，也在一些应用中用作主数据库。Redis 的灵活性、简单性和高可用性，使其成为各种实时应用和分布式系统的理想选择。

1.2 数据库发展趋势

1.2.1 1960s-1980s: 层次数据库

20 世纪 60 年代到 80 年代的数据库技术被称为“层次结构”，也可以被称为网状结构，无论标签是什么，这个时代的想法都旨在以树状结构组织数据结构。

换言之，这个时代的数据库技术是将数据存储为相互链接的记录。在层次数据库的模型

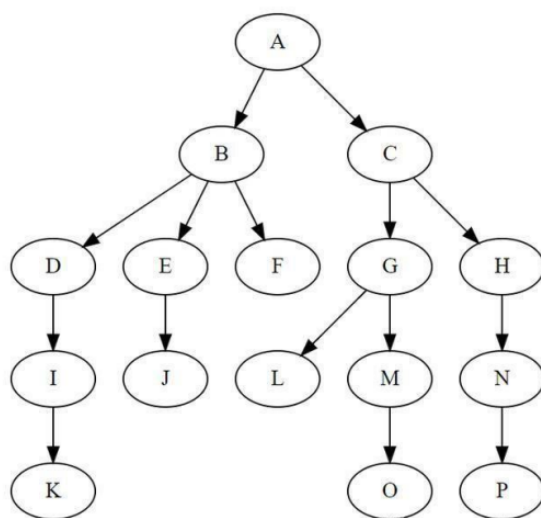


图 1.1: 层次数据库示意图

中，数据以节点的形式存在，每个节点可以包含一个或多个字段，形成父子关系。根节点是顶层节点，而叶节点是没有子节点的底层节点。层次数据库适用于需要表示层次结构信息的场景，如组织结构、文件系统等。尽管在过去曾经流行，但由于关系型数据库的普及，层次数据库在现代数据库系统中的应用相对较少。其特点包括数据重复、路径的唯一性以及专门的查询语言用于数据检索和更新。

1.2.2 1980s-2000s: 实体关系数据库

实体关系数据库（Entity-Relationship Database）是一种基于实体-关系模型的数据库系统，用于存储和管理数据。在这个模型中，数据以实体（Entity）和实体之间的关系（Relationship）为核心。每个实体都有属性，而实体之间的关系描述了这些实体之间的联系和交互。

实体关系数据库的优势在于其规范化的数据结构、ACID 特性（原子性、一致性、隔离性、持久性）以及使用 SQL（Structured Query Language）进行数据操作和查询的能力。这种数据库模型在处理复杂的关联数据和支持事务处理方面表现出色，因此在各种应用场景中广泛应用，包括企业应用、金融系统、医疗信息管理等。

关系型数据库采用了关系代数的概念，数据以表格（表）的形式组织，每个表表示一个实体，表中的行代表实体的具体实例，而列代表实体的属性。实体之间的关系则通过外键来建立。

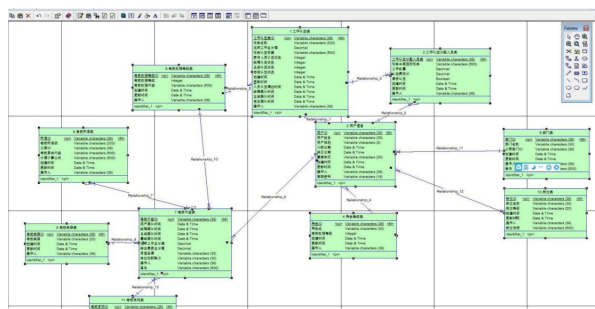


图 1.2: 实体关系数据库示意图

1.2.3 2000s-2020s:NoSQL

NoSQL 为 Not Only SQL 的缩写，是对不同于传统的关系型数据库的数据库管理系统的统称。

网络上的数据本质上不是表格的结构。

NoSQL 常用于超大规模数据的存储（例如谷歌或 Facebook 每天为他们的用户收集万亿比特的数据）。这些类型的数据存储不需要固定的模式，无需多余操作就可以横向扩展。

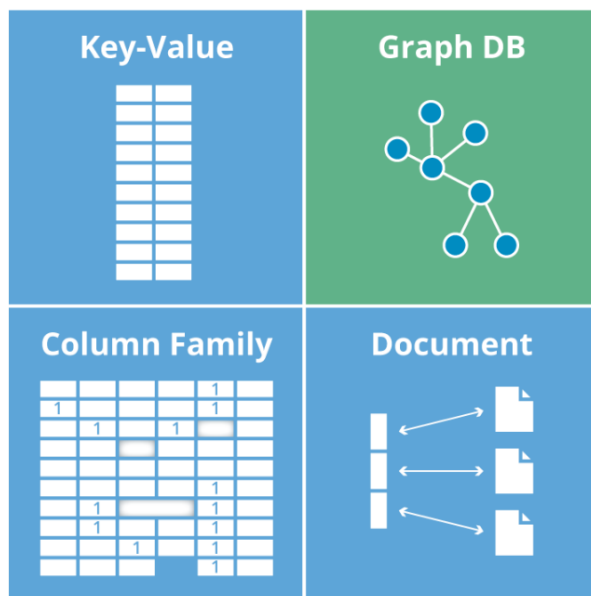


图 1.3: NoSQL 数据库示意图

1.2.4 2020s-未知：图数据库

这个创新时代正在从存储系统的效率转向从存储系统包含的数据中提取价值。

即价值从效率转移到高度连接的数据资产中衍生。

1.3 SQL*Plus 工具

1.3.1 介绍

SQLPlus 是 Oracle 数据库系统中的一种交互式查询工具和脚本处理器。它是一个**命令行工具**，允许用户连接到 Oracle 数据库并执行 SQL 查询、PL/SQL 块以及其他数据库管理任务。

1.3.2 交互过程

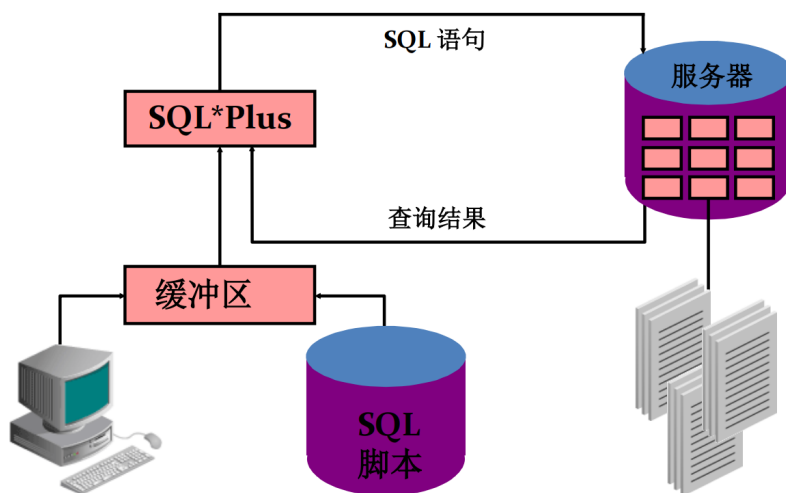


图 1.4: SQL*Plus 交互示意图

参与组件介绍：

用户界面 用户与 SQLPlus 进行交互的地方，可以是命令行界面或其他支持 SQLPlus 的用户界面。用户通过这个界面输入 SQL 查询、PL/SQL 块或 SQL*Plus 命令。

SQL*Plus 引擎 SQLPlus 引擎是一个解释器，负责解释和执行用户输入的 SQLPlus 命令，以及管理与 Oracle 数据库的交互。它解析并执行 SQL 查询、PL/SQL 块，处理输出格式，管理连接和会话等。

OCI (Oracle Call Interface) OCI 是 Oracle 数据库提供的一组 API（应用程序接口），允许应用程序（包括 SQLPlus）与 Oracle 数据库进行通信。**SQLPlus 使用 OCI** 与数据库建立连接、发送 SQL 命令，以及接收执行结果。

Oracle 数据库引擎 这是实际执行 SQL 查询、PL/SQL 块的地方。当 SQLPlus 将命令发送给数据库时，Oracle 数据库引擎负责解析和执行这些命令，然后返回结果给 SQLPlus。

数据缓冲区 SQL*Plus 通常会在本地维护一个数据缓冲区，用于存储从数据库检索到的数据。这使得用户可以在本地对结果进行分析、浏览和编辑。

命令缓冲区 用于存储执行过的 SQL 语句或 PL/SQL 块，以便可以轻松地重新执行它们或进行修改。当在 SQL*Plus 中输入一个 SQL 语句或 PL/SQL 块并按下 Enter 键时，该语句会被存储在命令缓冲区中，以便稍后再次使用。

1.3.3 Oracle 数据库连接命令

在 CMD 中运行 SQLPlus 并连接到 Oracle 服务器的命令为：

sqlplus <用户名> /<密码>@//<数据库 IP>:<Port> /<服务名>

如：

例 1.1. *sqlplus system/system@//192.168.146.132:1521/helowin*

1.3.4 常用编辑命令

DESC DESCRIBE，显示表结构，包括表的列名、数据类型和约束信息，效果见图1.5。

L 列出当前缓冲区中的 SQL 语句（未执行）的命令。

[N] number，选中第 n 行；[n] 后可追加内容 [text] 来替换第 n 行或新建第 n 行。

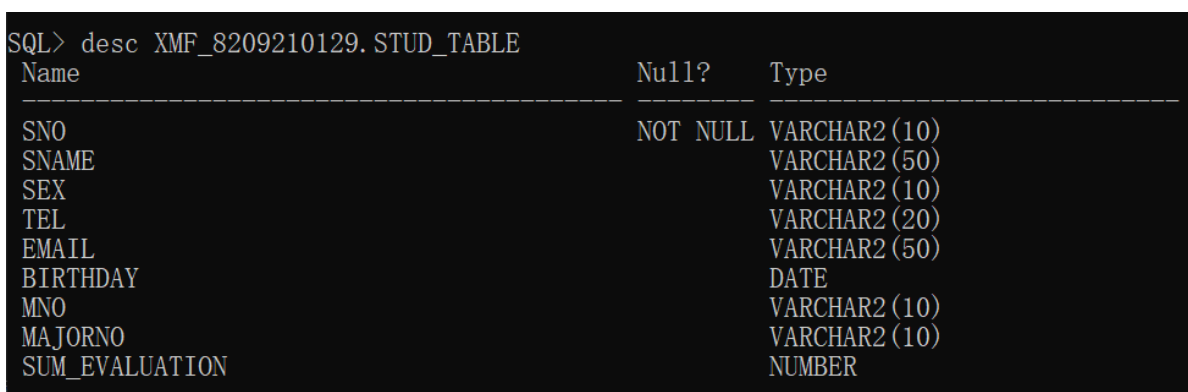
A APPEND，追加文本到缓冲区中的当前选中行。使用 “.” 单独一行表示结束追加。使用前要求缓冲区内存在文本。

C CHANGE，修改缓冲区中的当前选中行的文本，格式为 **C /<原文> /<新文>**。

DEL DELETE，删除缓冲区中的当前选中行。可追加 [n] 代表删除第 n 行；追加 [n][m] 删除第 n 至 m 行。

/ 运行在 SQL 缓冲区中的 SQL 语句。

CL CLEAR，清除 SQL*Plus 屏幕上的内容，将屏幕滚动条移至顶部。



SQL> desc XMF_8209210129.STUD_TABLE		
Name	Null?	Type
SNO	NOT NULL	VARCHAR2(10)
SNAME		VARCHAR2(50)
SEX		VARCHAR2(10)
TEL		VARCHAR2(20)
EMAIL		VARCHAR2(50)
BIRTHDAY		DATE
MNO		VARCHAR2(10)
MAJORNO		VARCHAR2(10)
SUM_EVALUATION		NUMBER

图 1.5: DESC 效果图

使用编辑缓冲区的命令前，需保证缓存区中存在内容：

```
SQL> SELECT *  
  2 FROM XMF_820921029  
  3 WHERE SNO  
  4 .
```

图 1.6: 输入内容到缓冲区，使用 `enter` 换行，使用单独的 `.` 结束

可使用 `L` 查看缓冲区：

```
SQL> L  
  1 SELECT *  
  2 FROM XMF_820921029  
  3* WHERE SNO
```

图 1.7: `L` 命令效果图

`*` 号所在行为正在编辑行。可使用数字更改选中行后，使用 `A` 命令添加内容：

```
SQL> L  
  1 SELECT *  
  2 FROM XMF_820921029  
  3* WHERE SNO  
SQL> 2  
  2* FROM XMF_820921029  
SQL> A .STUD_TABLE  
  2* FROM XMF_820921029.STUD_TABLE  
SQL> L  
  1 SELECT *  
  2 FROM XMF_820921029.STUD_TABLE  
  3* WHERE SNO
```

图 1.8: `A` 命令、`n` 命令效果图

可使用 `C` 命令更改指定行（注意原文与新文中间无空格作为间隔）：

```
SQL> L
  1  SELECT *
  2  FROM XMF_820921029.STUD_TABLE
  3* WHERE SNO
SQL> C /SNO /SEX=男
SP2-0023: String not found.
SQL> C /SNO/SEX=男
  3* WHERE SEX=男
SQL> L
  1  SELECT *
  2  FROM XMF_820921029.STUD_TABLE
  3* WHERE SEX=男
```

图 1.9: C 命令效果图

可使用 DEL 命令删除指定行:

```
SQL> L
  1  SELECT *
  2  FROM XMF_820921029.STUD_TABLE
  3* WHERE SEX=男
SQL> DEL 3
SQL> L
  1  SELECT *
  2* FROM XMF_820921029.STUD_TABLE
SQL> _
```

图 1.10: DEL 命令效果图

再次使用 n 命令新建行:

```
SQL> L
  1  SELECT *
  2* FROM XMF_82092029.STUD_TABLE
SQL> 3 WHERE SNO=982
SQL> L
  1  SELECT *
  2  FROM XMF_82092029.STUD_TABLE
  3* WHERE SNO=982
SQL> _
```

图 1.11: N 命令效果图

使用 / 运行 (使用后缓冲区不会被清除):


```
SQL> L
1 SELECT *
2 FROM XMF_8209210129.STUD_TABLE
3* WHERE SNO=982
SQL> /

SNO
-----
SNAME
-----
SEX
-----
TEL
-----
EMAIL
-----
BIRTHDAY      MNO      MAJORNO
-----
SUM_EVALUATION
-----
982
```

图 1.12: 运行命令效果图

1.3.5 SQL 文件存取命令

SAVE 把 SQL 缓冲区的内容存入指定的文件，如 *SAVE D:\SQL\SMAPLE*。

GET 将指定的脚本文件装入 SQL 缓存区，如 *GET D:\SQL\SMAPLE*。

START @ 或 **START**，把指定的脚本文件装入 SQL 缓冲区并运行，如 *@ D:\SQL\SMAPLE.sql*。

1.3.6 输出保存命令

SPOOL 命令用于将输出结果保存到文件中。该命令可以将 SQL 查询结果、PL/SQL 块的输出等保存到一个文件，这对于记录和分析数据库操作非常有用。语法为：

SPO[OL] [File_name[.ext]] [[CRE[ATE]]|REP[LACE]]|APP[END]] | OFF | OUT]。

参数说明：

File_name 指定脱机文件的名称，默认的文件扩展名为 LST。

CRE[ATE] 表示创建一个新的脱机文件，这也是 **SPOOL** 命令的默认状态。

REP[LACE] 表示替代已经存在的脱机文件。

APP[END] 表示把脱机内容附加到一个已经存在的脱机文件中。

OFF | OUT 表示关闭 **SPOOL** 输出。

第二章 PL/SQL 概述

2.1 介绍

PL/SQL (Procedural Language/Structured Query Language): 一种过程化编程语言，专门用于 **Oracle** 数据库系统。它将 SQL (Structured Query Language) 语句与结构化程序设计语言（如条件、循环等）相结合，提供了一种强大的方式来处理和操作数据库中的数据。

2.2 优点

1. **数据库集成:** PL/SQL 是为数据库设计的，能够直接嵌套在 SQL 中；许多与数据库有关的应用程序功能都已经集成在 PL/SQL 语言中。
2. **模块化开发:** PL/SQL 允许将代码模块化组织，通过存储过程和包的方式来管理和封装代码。这有助于提高代码的可维护性和重用性。
3. **性能优化:** PL/SQL 支持存储过程和函数，可以在数据库中预编译和存储，提高了执行效率。此外，PL/SQL 还支持游标，能够有效地处理大量的数据。

此外，还有其他如**安全性**（PL/SQL 能够帮助限制对数据库的直接访问，从而提高了数据库的安全性）、**灵活性**（PL/SQL 具有丰富的控制结构和数据类型，使其既可以用于简单的 SQL 查询，也可以用于复杂的业务逻辑实现）、**事务控制**（PL/SQL 提供了强大的事务控制功能，支持原子性、一致性、隔离性和持久性（ACID 属性））等优点。

2.3 执行体系

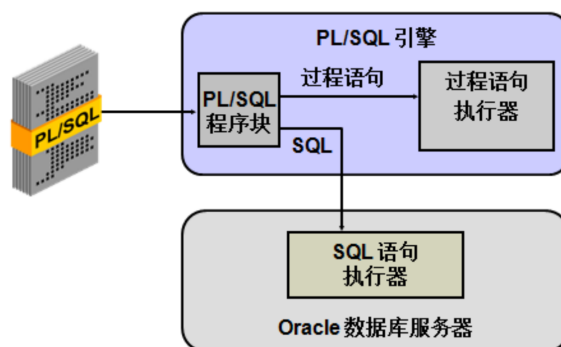


图 2.1: PLSQL 执行体系示意图

从图2.1中这些组件的角度来看 PL/SQL 的执行过程：

PL/SQL 引擎

- **PL/SQL 解释器：**编译后的 PL/SQL 代码由 PL/SQL 引擎的解释器执行。解释器负责逐行执行 PL/SQL 代码，并管理变量、控制流程、处理异常等。
- **数据交互：**在 PL/SQL 执行过程中，可能涉及到与数据库的交互，例如执行 SQL 语句、操作数据等。这些交互通过 PL/SQL 引擎协调实现。

过程语句执行器

- **编译过程性代码：**过程语句执行器会编译 PL/SQL 代码，将其转换为可执行的形式。这个编译过程包括语法检查、编译成中间代码等步骤。
- **过程语句解析：**如果 PL/SQL 程序包含了过程性语句，这些语句将被 PL/SQL 引擎的过程语句执行器处理。这可能包括存储过程、函数或触发器的调用。

SQL 执行器

- **接收 SQL 语句：**当一个包含 PL/SQL 代码的程序被触发时，其中可能包含 SQL 语句。这些 SQL 语句被传递给 SQL 执行器，它负责执行这些语句。
- **解析 SQL 语句：**SQL 执行器解析 SQL 语句，进行语法分析和语义检查。在这个阶段，执行计划被生成，这是一个描述如何执行查询的内部表示。

Oracle 数据库服务器

- 数据存储和检索：当 *PL/SQL* 程序执行 *SQL* 语句时，Oracle 数据库服务器负责实际的数据存储和检索。执行计划告诉数据库服务器如何最有效地执行查询，并从表中检索或修改数据。

当有过程调用或 *SQL* 语句执行时，流程可能涉及多次从 *PL/SQL* 引擎到 *SQL* 执行器，再到数据库服务器的交互。

需要注意的是，Oracle 数据库的内部执行细节是复杂的，并且可能受到优化、缓存、并发控制等多方面的影响。

第三章 程序块结构

3.1 程序块

PL/SQL 程序块是 PL/SQL 语言中的基本结构，它可以包含变量、常量、游标、异常处理等元素。PL/SQL 程序块有三种主要形式：**匿名块**、**存储过程**和**存储函数**。

1. **匿名块 (Anonymous blocks)**: 没有命名的程序块。匿名块是在应用程序内部需要的地方声明的，在这个应用程序每次执行时，这些匿名块都会被编译并执行。
2. **过程 (Procedures) 和函数 (Functions)**: 过程和函数也统称为**子程序 (Subprograms)**，子程序是对匿名块的补充，子程序就是被命名的 PL/SQL 程序块，而它们可以存储在数据库中。

一个简单的 PL/SQL 匿名块的结构如下：

Listing 3.1: PL/SQL 匿名块示例代码

```
1 DECLARE
2     -- 声明变量和常量
3     variable_name datatype;
4     constant_name CONSTANT datatype := value;
5
6 BEGIN
7     -- 可执行的PL/SQL代码
8     -- 可以包括各种语句，如赋值、条件语句、循环等
9     -- 例如：
10    variable_name := value;
11    IF condition THEN
12        -- do something
13    END IF;
```

```
14
15 EXCEPTION
16     -- 异常处理部分
17     -- 处理可能出现的异常
18     -- 例如:
19     WHEN others THEN
20         -- handle exception
21
22 END;
23 /
```

一个简单的 PL/SQL 函数的示例代码（获取员工薪水）：

Listing 3.2: 获取员工薪水函数示例代码

```
1 CREATE OR REPLACE FUNCTION get_employee_salary(p_employee_id NUMBER)
2     RETURN NUMBER
3     IS
4     -- 声明变量
5     v_salary NUMBER;
6 BEGIN
7     -- 查询员工薪水
8     SELECT salary INTO v_salary
9     FROM employees
10    WHERE employee_id = p_employee_id;
11
12    -- 返回薪水
13    RETURN v_salary;
14 END;
15 /
```

一个简单的 PL/SQL 过程的示例代码（更新员工薪水）：

Listing 3.3: 更新员工薪水过程示例代码

```
1 CREATE OR REPLACE PROCEDURE update_employee_salary(p_employee_id NUMBER,  
    p_new_salary NUMBER)  
2 IS  
3 BEGIN  
4     -- 更新员工薪水  
5     UPDATE employees  
6     SET salary = p_new_salary  
7     WHERE employee_id = p_employee_id;  
8  
9     -- 提交事务  
10    COMMIT;  
11 END;  
12 /
```

程序块结构说明：

DECLARE 可选，声明段，以关键字 DECLARE 开始并以执行段的开始而结束。

BEGIN 必选，执行段，以关键字 BEGIN 开始而以关键字 END 或关键字 EXCEPTION 结束。

EXCEPTION 可选，异常处理段，以关键字 EXCEPTION 开始，以关键字 END 结束。

执行规则：

1. 每一个 PL/SQL 控制语句都是以分号 (;) 结束。
2. 使用正斜杠 (/) 运行 SQL*Plus 内存缓冲区的匿名 PL/SQL 程序块。

3.2 程序块的嵌套

因为 PL/SQL 是一个过程化的程序设计语言，所以它具备语句嵌套能力。只要运行执行语句存在的地方就可以使用嵌套程序块，这种嵌套式可以许多层的。

在一个 PL/SQL 程序块中声明的变量（标识符）对于这一程序块本身来说是局部（本地）变量，而对于所有子块就是全局变量。如果一个全局变量在一个子块中又再次进行了声明（与全局变量同名），内部块中的变量将隐藏外部块中的同名变量。

Listing 3.4: 程序块嵌套示例代码

```
1 DECLARE
2   -- 全局变量
3   global_var NUMBER := 10;
4 BEGIN
5   DBMS_OUTPUT.PUT_LINE('Global Variable: ' || global_var);
6
7   -- 子块中声明同名的局部变量
8   DECLARE
9     local_var NUMBER := 20; -- 局部变量
10  BEGIN
11    DBMS_OUTPUT.PUT_LINE('Local Variable: ' || local_var);
12    -- 在子块中使用局部变量，优先使用局部变量的值
13    global_var := 30; -- 修改全局变量
14    DBMS_OUTPUT.PUT_LINE('Global Variable (Modified in Subblock): ' ||
15                          global_var);
16  END;
17
18  -- 子块结束后，回到全局作用域
19  DBMS_OUTPUT.PUT_LINE('Global Variable (Back to Global Scope): ' ||
20                        global_var);
21 END;
```

信息

DBMS 导出

```
Global Variable: 10
Local Variable: 20
Global Variable (Modified in Subblock): 30
Global Variable (Back to Global Scope): 30
```

图 3.1: 嵌套程序块示例代码运行结果图

第四章 变量

变量，就是内存中一个命名的临时存储区，而变量中所存储的信息就是这个变量的当前值。

在 PL/SQL 中，在使用一个变量之前，必须首先声明这个变量。一旦声明了这个变量，就可以在 SQL 语句和过程化（PL/SQL）语句中使用这个变量了。

变量使用步骤：

1. 在声明部分（DECLARE 块）**声明和初始化变量**
2. 在执行部分（BEGIN 块）**为变量赋新值**
3. 通过参数将值传入 PL/SQL 块
4. 通过输出变量来查看结构

4.1 变量命名规则

- 必须以**英语字母**开始
- 可以包含**英语字母或数字或特殊字符**——美元号（\$）、下划线（_）、和井号（#）
- 长度最长为 **30** 个字符
- **不区分大小写**
- 不能是保留关键字
- 不能（不宜）与数据库的表或列同名

4.2 变量声明语法

在引用 PL/SQL 程序块中的变量之前，必须在声明段声明所有的变量（标识符）。在声明变量的同时还可以为变量赋初值，但是在声明变量时赋予初始值是可选的。如果在一个变量声明

中引用了其他变量，一定要确保在之前的语句中已经声明了引用的变量。

声明语法：标识符 [CONSTANT] 数据类型 [NOT NULL] [<:= | DEFAULT> initial_value]

Listing 4.1: PL/SQL 变量声明语法示例

```
1 DECLARE
2   v_dogid NUMBER(10) NOT NULL := 38;
3   v_name VARCHAR2(25) := 'White Tiger'
4   c_color CONSTANT VARCHAR2(15) := 'White'
5   v_birthday DATE;
```

4.3 变量数据类型

PL/SQL 支持四大类数据类型（最常用的是标量数据类型，目前熟悉这个即可）：

1. 标量（Scalar）数据类型：

- NUMBER: 用于存储数值，可以是整数或小数。
 - NUMBER(p,s): 数据型数据，p 表示精度（总位数），s 表示小数部分的位数
 - BINARY_INTEGER: 基本整型数
 - PLS_INTEGER: 基本带符号整数型
 - BINARY_FLOAT: 存储单精度浮点数，以 IEEE754 格式表示浮点数，它需要 5 个字节来存储数字
 - BINARY_DOUBLE: 存储双精度浮点数，以 IEEE754 格式表示浮点数，它需要 9 个字节来存储数字
- character: 字符型数据类型，用于存储字符串。
 - VARCHAR2(size): 基本变长字符型数据
 - CHAR(size): 基本定长字符型数据
- DATE: 用于存储日期和时间。
 - DATA: 基本日期和时间数据
 - TIMESATMP(precision): 该数据类型除了日期和时间之外还包括了多达小数点后

9 位秒数

- **BOOLEAN**: 用于存储布尔值（TRUE 或 FALSE）。
 - **BOOLEAN**: 基本逻辑类型，它只能存储逻辑计算的 3 个可能值之一

2. 组合（Composite）数据类型：

- **%ROWTYPE**: 表示一个表行的结构，通常在变量声明中使用。
- **RECORD**: 类似于%ROWTYPE，但可以自定义结构。
- **TABLE**: 用于存储同一数据类型的集合。可以是索引表（INDEX BY 表，也称关联数组）或嵌套表。

3. 引用（Reference）数据类型：

- **游标（CURSOR）**: 游标是一种引用数据类型，用于对查询结果进行迭代。可以使用显式游标或隐式游标。
- **引用游标（REF CURSOR）**: 用于存储游标引用，可用于动态查询结果的处理。

4. 大对象（LOB）数据类型：

- **CLOB (Character Large Object)**: 用于存储大量字符数据，如文本。
- **BLOB (Binary Large Object)**: 用于存储二进制数据，如图像、音频等。
- **NCLOB (National Character Large Object)**: 类似于 CLOB，但用于存储国家字符集数据。
- **BFILE**: 用于存储二进制文件的地址引用，文件存储在数据库外部。

4.4 变量命名建议

1. 遵守命名的规则，变量的命名规则与 SQL 对象的命名规则完全相同。
2. 必须初始化被指定为非空（NOT NULL）和常量（Constant）的变量。
3. 一行最好只声明一个标识符以提高代码的易读性和方便代码的维护。
4. 通过使用赋值操作符（:=）或默认关键字（DEFAULT）来初始化标识符。
5. 变量最好不要与列名重名。
6. 两个 PL/SQL 变量（对象）只要在不同的程序块中是可以同名的。

7. 如果没有必要就不要在 PL/SQL 变量上强加非空 (NOT NULL) 约束。
8. 代码中使用了 PL/SQL 游标、BLOB/CLOB Locator 或其他需要手动释放的资源, 请确保在不再需要它们时及时释放这些资源, 以避免内存泄漏。

4.5 代替变量和绑定变量

4.5.1 代替变量

因为 PL/SQL 本身没有输入和输出功能, 所以必须依赖于执行 PL/SQL 程序的环境变量值传入或传出 PL/SQL 程序块。

在 SQL*Plus 环境中, 可以使用 **SQL*Plus 的代替变量** 将运行时的值传给 PL/SQL 程序块。在 PL/SQL 程序块中可以使用前导的 & 符号引用代替变量, 就像在 SQL 语句中引用 SQL*Plus 的代替变量一样。在 PL/SQL 程序执行前, 正文的值被代替进 PL/SQL 程序块中。

例如, 一个使用代替变量的查询可以如下所示:

```
1 Select *
2 FROM XMF_8209210129.STUD_TABLE
3 WHERE SNO=&ReplaceVariable1;
```

运行效果:

```
SQL> DEFINE ReplaceVariable1 = 982
SQL> Select * from XMF_8209210129.STUD_TABLE where SNO=&ReplaceVariable1;
old 1: Select * from XMF_8209210129.STUD_TABLE where SNO=&ReplaceVariable1
new 1: Select * from XMF_8209210129.STUD_TABLE where SNO=982

SNO
-----
SNAME
-----
```

图 4.1: 代替变量案例运行结果

在这里, ReplaceVariable1 是一个代替变量, SQL*Plus 会在执行查询之前提示用户输入 ReplaceVariable1 的值 (如果未定义变量), 并将其替换到 SQL 语句中。DEFINE ReplaceVariable1 = 982 用于定义 SQLPlus 变量。

注意: 如果需要 '&variable' 这样的字符串, 可以使用两个连续的 & 来避免 SQL*Plus 将其解释为代替变量, 如 '&&your_value'。

4.5.2 绑定变量

绑定变量是在**使用（或调用）PL/SQL 的环境**中创建的，而不是在 PL/SQL 程序的声明段中定义的。在一个 PL/SQL 程序块中声明的所有变量只在执行这个程序块时可以使用。而在这个程序块执行后，这些变量所使用的内存就释放了。然而，绑定变量则不同，在程序块执行后，绑定变量依然存在并允许访问。

绑定变量是在 SQL 语句中使用占位符（通常是冒号: 后跟变量名）来代替实际的数值或表达式。**在 PL/SQL 中，通过使用绑定变量，可以减少 SQL 语句的解析次数，提高性能。**绑定变量在 SQL 语句执行时被动态绑定，而不是在每次执行时重新解析整个 SQL 语句。

运行演示：

```
SQL> variable v_sno number;
SQL> exec :v_sno := 982;

PL/SQL procedure successfully completed.

SQL> select *
   2  FROM XMF_8209210129.STUD_TABLE
   3  WHERE SNO=:v_sno;

SNO
-----
```

图 4.2: 绑定变量使用案例

variable v_sno number; 定义绑定变量 v_sno

exec :v_sno := 982 给绑定变量赋值为 982

4.5.3 不同

1. 绑定变量（Bind Variables）

- 绑定变量是用于在 SQL 语句中传递数值或数据的一种机制，通过绑定变量，可以将变量的值绑定到 SQL 语句中，而不是直接在 SQL 语句中使用硬编码的值。
- 这有助于提高性能，因为数据库可以缓存已编译的 SQL 语句，并在执行时仅替换绑定变量的值，而不需要重新编译整个 SQL 语句。

2. 代替变量（Substitution Variables）

- 代替变量是一种 SQLPlus 工具的功能，它允许在 SQLPlus 中使用变量，这些变量在脚本运行之前由用户手动输入，或者可以通过 DEFINE 命令定义。
- 代替变量在 SQL*Plus 脚本中被替换为相应的值，类似于宏替换。

4.6 变量作用域

一个变量（标识符）在声明它的程序块中是可见的，并且在所有嵌套的子块中也是可见的。如果一个 PL/SQL 块没有发现本地声明的变量，该 PL/SQL 程序块将向上查找包含它的父块，而不会向下查看所包含的子块或查找同一级别（没有嵌套关系）块。

作用域适用所有的对象，包括变量、游标、用户定义的异常和约束等。

4.6.1 局部变量（Local Variables）

1. 局部变量是在一个程序块内部声明的变量，仅在该程序块内部可见和可访问。
2. 局部变量的生命周期仅限于包含它的程序块的执行期间。一旦程序块执行完毕，局部变量就会被销毁，不再可用。
3. 局部变量通常在 DECLARE 部分声明，并可以用于存储临时数据或中间计算结果。

Listing 4.2: 局部变量作用域示例

```
1 DECLARE
2     local_variable NUMBER := 10;
3 BEGIN
4     -- 在此处访问和使用局部变量
5 END;
6 -- 局部变量在此处不可见
```

4.6.2 嵌套局部变量（Nested Local Variables）

1. 嵌套程序块中声明的局部变量对外部程序块不可见，但对内部程序块可见。
2. 如果内部程序块声明了与外部程序块相同名称的局部变量，它们不会相互冲突。**内部变量会遮蔽外部变量。**

Listing 4.3: 嵌套局部变量作用域示例

```
1 DECLARE
2     outer_variable NUMBER := 10;
3     PROCEDURE inner_procedure IS
4         inner_variable NUMBER := 20;
```

```
5 BEGIN
6     -- 在内部程序块中，可以访问 inner_variable 和outer_variable
7 END inner_procedure;
8 BEGIN
9     -- 在外部程序块中，只能访问 outer_variable
10 END;
```

注意：此处定义了一个内部过程 inner_variable。

4.6.3 全局变量（Global Variables）

1. 全局变量在整个 PL/SQL 块中声明，对所有程序块可见。
2. 全局变量的生命周期与整个 PL/SQL 块的执行周期相同。
3. 全局变量通常用于存储在多个程序块之间需要共享的数据。

4.7 布尔变量与布尔表达式

布尔变量和布尔表达式在任何程序语言设计上都是非常重要和广泛使用的。在 PL/SQL 程序中，可以在 SQL 语句中也可以在过程化语句中进行变量的比较，这样的比较表达式被称为布尔表达式，它们是由单个表达式或由关系操作符所分隔的复杂表达式所组成。

特性：

- 只有值 **TURE**、**FALSE** 和 **NULL** 可以赋给一个布尔变量。
- 可以通过**逻辑操作符 AND、OR 和 NOT** 对布尔变量进行运算，运算总是产生 TURE、FALSE 或 NULL。
- 数字、字符和日期表达式可以被用来返回一个布尔值。

4.7.1 布尔操作符（结果为布尔类型）

操作符	描述
=	等于
>	大于
>=	大于等于
<	小于
<=	小于等于
<>	不等于
BETWEEN...AND...	二个值之间（包括这二个值）
IN(set)	匹配值列表中的任意值
LIKE	匹配某一字符模式
IS NULL	是空值

图 4.3: 布尔操作符

4.8 LOB 变量

大对象 LOB 是 large object 的缩写，就意味着存储大量的数据，在数据库中，表中的列定义为 LOB 类型（如 CLOB 和 BLOB）。利用 LOB 数据类型，可以在数据库中存储大量的无结构数据块（如正文、图形、声音和影像信息），其存储量可多达 128T（数据量的多少取决于数据块大小）。

4.8.1 LOB 变量类型

CLOB 数据类型（Character Large Object）

CLOB 用于在数据库中存储字节流类型的大数据对象，如演讲稿、说明书或简历等。

BLOB 数据类型（Binary Large Object）

用于在数据库中存储大的二进制对象，如照片或幻灯片等。当从数据库中提取这样的数据或向数据库中插入这样的数据时，数据库并不解释这些数据。使用这些数据的外部应用程序必须自己解释这些数据。

BFILE 数据类型 (Binary File)

用于在数据库外的操作系统文件中存储大的二进制对象，如电影胶片等。与其他的 LOB 数据类型不同，BFILE 数据类型是外部数据类型。BFILE 类型是存储在数据库之外的，它们可能是操作系统文件。

NCLOB 数据类型 (National Language Character Large Object)

用于在数据库中存储 NCHAR 类型的单字节或定长多字节的 Unicode 大数据对象。

4.8.2 LOB 变量使用

Listing 4.4: LOB 变量使用示例代码

```
1 DECLARE
2   clob_data CLOB;
3   buffer VARCHAR2(32767);
4   amount NUMBER;
5   offset NUMBER := 1;
6 BEGIN
7   -- 初始化 CLOB 数据
8   clob_data := '这是一个CLOB示例。';
9   -- 读取 CLOB 数据
10  amount := DBMS_LOB.GETLENGTH(clob_data);
11  DBMS_LOB.READ(clob_data, amount, offset, buffer);
12  DBMS_OUTPUT.PUT_LINE(buffer);
13 END;
```

运行结果：

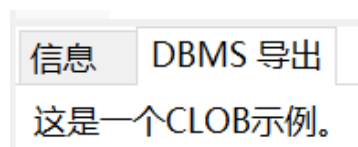


图 4.4: CLOB 示例代码运行结果

4.9 赋值：字符串分隔符（引号限定符）

介绍

`q` 和 `p` 是引号限定符，用于创建带有引号的字符串字面量。具体而言，`q` 和 `p` 用于创建带有引号的标识符或字符串，这样可以在字符串中包含特殊字符，而不必使用转义字符。`q` 用于创建带有双引号的标识符，而 `p` 用于创建带有单引号的字符串。注意：`p` 定界符可以使用大多数字符作为定界符，但不能使用双引号`'`，单引号`'`，反引号```，右括号`)`，右中括号`]`，右大括号`}` 和感叹号`!` 作为定界符。

演示

不使用字符串分隔符的情况：

Listing 4.5: 不使用字符串分隔符

```
1 DECLARE
2   v_message VARCHAR2(100);
3 BEGIN
4   -- 使用双单引号进行转义
5   v_message := 'He said, ''Hello World!''';
6
7   -- 打印字符串
8   DBMS_OUTPUT.PUT_LINE(v_message);
9 END;
```

使用字符串分隔符后：

Listing 4.6: 使用字符串分隔符

```
1 DECLARE
2   v_message VARCHAR2(100);
3 BEGIN
4   -- 使用 q'...' 创建原始字符串
5   v_message := q'{He said, 'Hello World!'}';
6
7   -- 打印字符串
```

```
8 DBMS_OUTPUT.PUT_LINE(v_message);  
9 END;
```

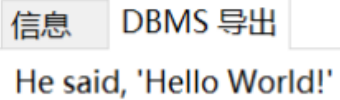


图 4.5: 字符串分隔符输出演示

4.10 类型声明：%TYPE 属性

为了避免这种变量数据类型和精度的硬编码（即数据类型和精度必须显式定义），PL/SQL 引入%TYPE 属性。程序员（开发人员）可以使用%TYPE 属性按照之前已经声明过的变量或数据库的列来声明一个变量。当存储在一个变量中值来自于数据库中的表时，使用%TYPE 属性来声明这个变量时再适合不过的了。

使用语法

1. 标识符 表名. 列名%TYPE
2. 标识符 定义好的变量%TYPE

案例

```
1 DECLARE  
2   v_data emp.data%TYPE  
3   v_Month_Value NUMBER(6,4) := 3.25  
4   v_Year_Value v_Month_Value%TYPE := 4.40
```

好处：

1. 可以不知道 emp 表中的 data 列或 v_Month_Value 的数据类型和精度就直接使用对应类型。
2. 当 emp 的 data 列或者 v_Month_Value 的数据类型和精度发生变化时，v_data 和 v_Year_Value 也会随之变化。

第五章 词法单元

与其他程序设计语言一样，构成任何 PL/SQL 程序块的最基本单元的程序部件是词法单元。一个词法单元就是一个字符序列，其中字符可以是字母、数字、特殊字符、空格、制表字符 (tabs)、回车符、符号。

词法单元（位）又可以进一步划分成标识符、定界符、文字和注释四大类：

标识符 (Identifiers) 标识符是命名的 PL/SQL 对象。

定界符 (Delimiters) 定界符是一些具有特殊含义的符号，其本身也经常被用作词法单元或语句的分隔符。

文字 (Literals) 任何赋予一个变量的值都是文字。

注释 (Comments) 是 PL/SQL 代码解释信息。

5.1 标识符 (Identifiers)

标识符是用来命名数据库对象（如表、列、视图、过程、变量等）的名称。

其命名规则与变量命名规则完全相同，因此标识符也不能与保留关键字同名。**如果使用保留关键字作为标识符名字或者标识符名字中包含了空格或要区分大小写，则必须在声明这个标识符时，将它用双引号括起来，如：“begin date” DATE。**

在之后使用这个标识符时也必须使用双引号。

5.1.1 标识符命名惯例

标识符	命名惯例	例子
游标	name_cursor	emp_cursor
异常	e_name	e_invalid_product
PL/SQL 表类型	name_table_type	ename_table_type
PL/SQL 表类型的变量	name_table	ename_table
记录类型	name_record_type	emp_record_type
记录类型的变量	name_record	emp_record

表 5.1: 标识符命名惯例表

5.2 定界符 (Delimiters)

定界符是用于界定代码块、语句、变量等元素范围的符号。在 PL/SQL 中，常见的定界符包括分号 (;)、引号 (') 和双引号 (") 等。

5.2.1 简单定界符

简单定界符通常只有一个符号。

操作符	含义
+	加法运算符
-	减法运算符
*	乘法运算符
/	除法运算符
=	相等操作符
@	远程访问符

表 5.2: 简单定界符表

5.2.2 组合定界符

组合定界符由二个符号组成。

操作符	含义
	连接运算符
:=	赋值操作符
!=	不等运算符
<>	不等运算符
/* */	开始/结束注释定界符
-	单行注释符

表 5.3: 组合定界符表

5.3 文字 (Literals)

任何赋予一个变量的值都是文字，即任何不是标识符的字符、数字、布尔或日期值都是文字。

文字可分为以下四类：

- 1. **字符文字：**字符文字也叫字符串文字，字符串文字的数据类型只能是 CHAR 或 VARCHAR2，如“社会核心价值观”、“中国梦”等。
- 2. **数字文字：**一个数字文字就是一个正整数或实数，如“123”、“123.45”。
- 3. **布尔文字：**赋予布尔变量的值是布尔文字，其值为 TRUE、FALSE 和 NULL。
- 4. **日期文字：**一个日期文字就是一个有效的日期类型数据，如 9-mar-2022。

5.4 注释 (Comments)

如果只注释一行：那么直接使用--。

如果注释的不止一行：那么使用符号 /* 和 */作为注释操作符。

5.5 程序设计指导原则

Oracle 推荐在编写 PL/SQL 程序代码时应该遵循的大小写规范。利用这些大小写规范可以很容易区分关键字和对象，以增加项目的协同性和开发难度。

种类	大小写	例子
SQL 语句关键字	大写	SELECT、FROM、UPDATE
PL/SQL 语句关键字	大写	BEGIN、END、DECLARE
数据类型	大写	BOOLEAN、NUMBER
标识符和参数	小写	v_job、g_job、emp_cursor
数据库中的表和列	小写	customers、empno、sal、dname

表 5.4: 大小写规范表

第六章 运算符

6.1 基本运算符

PL/SQL 语言中的运算符与 SQL 语言中的基本相同：

1. 算术（Arithmetic）运算符：+ - * / %
2. 比较（Comparison）运算符：= != /<> < <= > >=
3. 逻辑（Logical）运算符：AND/&& OR/|| NOT/!
4. 位（Bitwise）运算符：AND OR XOR NOT
5. 赋（Assignment）值运算符：:=
6. 串接 | 连接（Concatenation）运算符：||
7. 控制操作顺序的符号：()
8. 模式匹配运算符：LIKE NOT LIKE
9. 成员关系运算符：IN BETWEEN
10. 等

6.2 空值运算

可使用 **Variable IN NULL** 判断是否为 NULL，如果返回值为 true 说明是 NULL。

在操作空值（NULL）时，需要牢记：

- 在**比较表达式**（如 <>）中只要有空值，其结果总是空值；
- 对一个空值进行 NOT 运算，其结果还是空值；
- 在条件（分支）语句中，如果条件为 NULL，与这个条件相关的语句序列将不会执行。

第七章 预定义函数

为了方便地使用 Oracle 数据库，Oracle SQL 提供了大量的可以在 SQL 语句中使用预定义函数，实际上这些函数增强了 SQL 语言功能。这些函数的**绝大多数**在 PL/SQL 表达式中也是有效的，如单行数字函数、单行字符函数、数据类型转换函数、日期函数、Timestamp 函数、GREATEST、LEAST 函数等。

注意：**分组函数**（如 AVG、MIN、MAX、COUNT、SUM、STDDEV 和 VARIANCE 函数等）依旧只能在 SQL 语句中使用，不能用于 PL/SQL 的表达式中。

7.1 单行数字函数

以下为常用单行数字函数：

三角函数 如：SIN、ASIN、SINH、COS、ACOS、COSH、TAN、ATAN、TANH。

ABS(X) 绝对值。

BITAND(X,Y) 对两个数字进行按位与（AND）运算，得到的结果是一个新的整数。

CEIL(X) 返回大于或等于 X 的最小整数。

FLOOR(X) 返回小于或等于 X 的最大整数。

exp(X) 返回 e 的 x 次幂。

LN(X) 返回 X 的自然对数。

POWER(X,Y) 返回 X 的 Y 次幂。

SQRT(X) 返回 X 的平方根。

ROUND(X[,Y]) 返回对 x 取整的结果，其中 y 为可选，说明对第几位小数处取整。

7.2 单行字符函数

以下为常用单行字符函数：

concat(m,n) 将 m 和 n 连接起来，并返回连接后的字符串。

initcap(n) 将 n 中的第一个字母转换成大写。

instr(x,y,m,n) 在字符串 x 中查找字符串 y 出现的位置，m 是搜索开始的位置，n 是指定第 n 次字符串 y 出现的位置。

length(n) 求字符串 n 的长度。

lower(n) 求字符串 n 的长度。

lpad(x,n,y) 在字符串 x 的左边补齐字符 y，得到总长度为 n 个字符的字符串。

rpadd(x,n,y) 在字符串 x 的右边补齐字符 y，得到总长度为 n 个字符的字符串。

trim([LEADING | TRAILING | BOTH] trim_character FROM source_string)

去除字符串 source_string 两端（或左侧、右侧）的指定字符 trim_character。

ltrim(x,y) 去除字符串 x 左侧的指定字符 y。

rtrim(x,y) 去除字符串 x 右侧的指定字符 y。

7.3 数据类型转换函数

PL/SQL 支持动态数据类型转换，分为两大类：**隐含转换**和**显示转换**。隐含转换发生在混合使用多种数据类型的语句中，PL/SQL 会尝试自动转换数据类型，如字符型和数字型之间、字符型和日期型之间。显示转换则需要使用内置的转换函数，例如 TO_CHAR、TO_DATE、TO_NUMBER 和 TO_TIMESTAMP，用于将一个值从一种数据类型转换为另一种数据类型。

以下为常用的数据类型转换函数：

1. **TO_NUMBER(expr, [format_mask], [nls_params])**：其中 expr 是要转换的表达式，format_mask 是可选的格式掩码，nls_params 是可选的国家语言设置。
2. **TO_CHAR(expr, [format_mask], [nls_params])**：其中 expr 是要转换的表达式，format_mask 是可选的格式掩码，nls_params 是可选的国家语言设置。

3. **TO_DATE(char_string, [format_mask], [nls_params])**: 其中 char_string 是要转换的字符串, format_mask 是可选的格式掩码, nls_params 是可选的国家语言设置。
4. **CAST(num AS type)** 进行显式的数据类型转换, 用于将一个数据类型转换为另一个数据类型, 如 **str := CAST(num AS VARCHAR2(20));**。

7.4 日期函数

以下为常用的日期函数:

SYSDATE 返回当前系统日期和时间。

TO_DATE 将字符串转换为日期类型,

如 **converted_date := TO_DATE('2023-01-01', 'YYYY-MM-DD');**。

MONTHS_BETWEEN 计算两个日期之间的月份差,

如 **months_diff := MONTHS_BETWEEN(date2, date1);**。

ADD_MONTHS 在给定日期上增加指定的月数,

如 **result_date := ADD_MONTHS(start_date, 3);**。

LAST_DAY 本月的最后一天。

NEXT_DAY(start_date, day_of_week) 找到指定日期之后的下一个指定星期几的日期,

day_of_week 是星期的缩写, 例如 'MON' 表示星期一, 'TUE' 表示星期二, 以此类推。

ROUND(date_expression [, format_mask]) 将日期四舍五入到指定的精度。

如 **rounded_date := ROUND(current_date, 'MONTH');**。

TRUNC(date_expression [, format_mask]); 截断日期到指定的精度, 类似 ROUND 函数。

7.5 CHR 函数

CHR(integer) 函数的作用是将所给的数字换成对应的 ASCII 码字符 (因为有一些字符无法用键盘输入)。

如:

- **CHR(9)** – 制表符

- CHR(10) – 换行符
- CHR(13) – 回车符
- CHR(32) – 空格符
- CHR(34) – 双引号
- CHR(39) – 单引号

第八章 序列

8.1 序列概念

在 SQL 中，序列（Sequence）是一种数据库对象，用于生成唯一的数字值。它通常用于为表的主键列生成唯一标识符。序列是数据库中的一种生成器，提供了一种自动递增或递减的数字值生成方式。

创建序列的语法：

Listing 8.1: 创建序列语法

```
1 CREATE SEQUENCE sequence_name
2   [INCREMENT BY n]
3   [START WITH n]
4   [MAXVALUE n | NOMAXVALUE]
5   [MINVALUE n | NOMINVALUE]
6   [CYCLE | NOCYCLE]
7   [CACHE n | NOCACHE];
```

各参数的含义：

1. **INCREMENT BY:** 指定序列的增量，默认为 1。
2. **START WITH:** 指定序列的起始值，默认为 1。
3. **MAXVALUE** 和 **MINVALUE:** 分别指定序列的最大值和最小值。
4. **CYCLE | NOCYCLE:** 指定是否循环序列值，如果达到最大值后再次从最小值开始（或相反）。
5. **CACHE | NOCACHE:** 指定是否缓存序列值，以提高性能。

8.2 PL/SQL 中序列的操作

8.2.1 NEXTVAL

NEXTVAL 函数用于获取序列的下一个值。每次调用 **NEXTVAL** 都会获取序列的下一个值，并且序列的计数器会递增（更新序列）。

Listing 8.2: NEXTVAL 示例代码

```
1 DECLARE
2     next_id NUMBER;
3 BEGIN
4     next_id := your_sequence_name.NEXTVAL;
5     DBMS_OUTPUT.PUT_LINE('Next ID: ' || next_id);
6 END;
```

8.2.2 CURRVAL

CURRVAL 函数用于获取序列的当前值。

Listing 8.3: CURRVAL 示例代码

```
1 DECLARE
2     current_id NUMBER;
3 BEGIN
4     current_id := your_sequence_name.CURRVAL;
5     DBMS_OUTPUT.PUT_LINE('Current ID: ' || current_id);
6 END;
```

第九章 SELECT

9.1 INTO

在 PL/SQL 的 SELECT 语句中增加了 INTO 子句。通过 SELECT 子句提取（查找到的）数据（表中列的值），而通过 INTO 子句将提取的数据存放在 PL/SQL 变量中。

Listing 9.1: SELECT 示例代码

```
1 SELECT 查询列表
2 INTO {变量名[,变量名]... | 记录名}
3 FROM 表
4 [WHERE 条件表达式];
```

查询列表 (select_list) 为一个或多个（至少一个）列名、SQL 表达式、SQL 单行函数或 SQL 分组函数。

变量名 (Variable_name) 存储获取值的 PL/SQL 标量变量。

记录名 (record_name) 存储多个取值的 PL/SQL 记录类型的变量。

表 (Table) 数据库中的表名（从中提取数据的表）。

条件表达式 (condition) 由列名、表达式、常量和比较操作符组成的表达式，其中可以包括 PL/SQL 变量和常量。

注意事项：

1. 每一个 SQL 语句必须有分号 (;)。
2. INTO 子句是**强制性的**（必须存在），并只能放在 SELECT 子句和 FROM 子句之间。

3. 每一个从表中提取的值**必须存入一个 PL/SQL 的变量中**（通过 INTO），而每一个变量都必须在使用前声明过；此外必须为所选择的项（列或表达式）指定一个变量，顺序要对应
4. 在 INTO 子句中的变量的个数必须与查询列表中数据库列的个数完全相同，并且每一个变量数据类型和精度一定与对应位置的列兼容。
5. 使用 SELECT INTO 语句时，**确保查询返回的结果只有一行**。如果查询返回多行（此时应该使用 BULK COLLECT INTO 代替 INTO）或没有数据，PL/SQL 将引发 TOO_MANY_ROWS 或 NO_DATA_FOUND 异常。
6. 可以在 SQL 语句中使用分组函数。
7. INTO 子句中使用的变量可以 PL/SQL 变量，也可以是宿主变量（如绑定变量）。

Listing 9.2: INTO 案例代码

```
1 DECLARE
2     v_deptno dept.deptno%TYPE;
3     v_loc VARCHAR2(38);
4 BEGIN
5     SELECT deptno,loc
6     INTO v_deptno, v_loc
7     FROM dept
8     WHERE dname = 'ACCOUNTING';
9 END;
10 /
```

9.2 变量与列同名问题

Oracle 系统在执行 PL/SQL 中嵌入的 SQL 语句时是按照标识符的优先次序来处理标识符的，所以如果 PL/SQL 变量与列名同名，则可能造成程序的二义性。Oracle 的处理变量、参数、表和列的优先级如下：

1. PL/SQL 首先检查数据库表中**列**的名字
2. 数据库表中列的名字优先于**本地变量名**
3. 本地变量名和形式参数名优先于数据库的**表名**

列名 (Column Name) > 变量名 (Variable Name) > 表名 (Table Name)

第十章 DML

10.1 概述

在 PL/SQL 程序块中不能直接使用数据定义语言（DDL）的语句（如不能使用 CREATE TABLE、ALTER TABLE 或者 TRUNCATE TABLE 等），并且也不能直接使用数据控制语言（DCL）的语句（如不能使用 GRANT 或者 REVOKE 语句）。

注意：EXECUTE IMMEDIATE 是用于执行动态 SQL 的语句，这允许 PL/SQL 程序在运行时动态生成 SQL 语句并执行（动态 SQL 允许以字符串形式构建 SQL 语句）。在使用动态 SQL 的情况下，PL/SQL 通过动态 SQL 获得了一种间接执行 DDL 和 DCL 操作的方式。

DDL 和 DML 的区别：

1. DDL 代表数据定义语言，而 DML 代表数据操作语言。
2. DDL 中常用的命令有：create, drop, alter, truncate 和 rename 而 DML 中常用的命令有：insert, update, delete 和 select 等等。
3. DDL 命令会影响整个数据库或表而 DML 命令会影响表中的一个或多个记录。
4. 带有 DDL 命令的 SQL 语句无法回滚；带有 DML 命令的 SQL 语句可以回滚。

PL/SQL 中存在 4 类 DML 语言：

1. **INSERT 语句**：往一个表中添加一行新数据。
2. **UPDATE 语句**：更改一个表中现有的数据行。
3. **DELETE 语句**：从一个表中移除数据行。
4. **MERGE 语句**：从一个表中选择数据行以修改或插入到另一个表。

10.2 DML 代码示例

10.2.1 INSERT

Listing 10.1: INSERT 代码示例

```
1 BEGIN
2   INSERT INTO dept(deptno, dname, loc);
3   VALUES (deptid_sequence.NEXTVAL, '燕城苑', '天通苑');
4 END;
5 /
```

使用序列 deptid_sequence 的 NEXTVAL 产生新记录的 deptno。

10.2.2 UPDATE

Listing 10.2: UPDATE 代码示例

```
1 DECLARE
2   v_sal_increase emp.sal%TYPE := &p_sal_increase;
3 BEGIN
4   UPDATE emp
5   SET sal = sal + v_sal_increase
6   WHERE job = 'CLERK';
7 END;
8 /
```

10.2.3 DELETE

Listing 10.3: DELETE 代码示例

```
1 DECLARE
2   v_job emp.job%TYPE := '&p_job';
3   v_sal emp.sal%TYPE := &p_sal;
4 BEGIN
```

```
5 DELETE FROM emp_pl
6 WHERE job = v_job
7 AND sal > v_sal;
8 END;
9 /
```

10.3 MERGE

利用 MERGE 语句，可以根据设定的条件来决定对表的操作是修改、插入还是删除，从而避免了使用多个 DML 语句。

10.3.1 MERGE 基本语法

Listing 10.4: MERGE 语法

```
1 MERGE INTO target_table USING source_table
2 ON (condition)
3 WHEN MATCHED THEN
4 -- 更新操作
5 WHEN NOT MATCHED THEN
6 -- 插入操作
```

INTO 子句 说明正在修改或插入的目标表

USING 子句 标识要修改或插入的数据源，既可以是表，也可以是视图

ON 子句 定义 MERGE 语句是进行修改操作还是进行插入操作的条件

WHEN MATCHED THEN 定义当条件满足时所做的操作

WHEN NOT MATCHED THEN 定义当条件不满足时所做的操作

10.3.2 MERGE 示例代码

Listing 10.5: MERGE 示例代码 1

```
1 MERGE INTO employees target
2 USING employees_backup source
3 ON (target.employee_id = source.employee_id)
4 WHEN MATCHED THEN
5     UPDATE SET target.salary = source.salary
6 WHEN NOT MATCHED THEN
7     INSERT (employee_id, salary) VALUES
8     (source.employee_id, source.salary);
```

Listing 10.6: MERGE 示例代码 2

```
1 BEGIN
2     MERGE INTO copy_emp c
3     USING emp e
4     ON (c.empno = e.empno)
5     WHEN MATCHED THEN
6         UPDATE SET
7             c.empno = e.empno,
8             c.ename = e.ename,
9             c.job = e.job,
10            c.mgr = e.mgr,
11            c.hiredate = e.hiredate,
12            c.sal = e.sal,
13            c.comm = e.comm,
14            c.deptno = e.deptno
15     WHEN NOT MATCHED THEN
16         INSERT (empno, ename, job, mgr, hiredate, sal, comm, deptno)
17         VALUES (e.empno, e.ename, e.job, e.mgr, e.hiredate, e.sal, e.comm, e.
18                deptno);
19 END;
```

第十一章 条件语句

11.1 布尔条件

如果要构建一个复杂的布尔条件，需要使用逻辑运算符 AND（逻辑与/逻辑乘）、OR（逻辑加/逻辑或）或 NOT（逻辑非）将简单的布尔条件组合在一起。

AND 和 OR 用于把两个条件组合在一起最后产生一个结果，其语法是：

条件 1 逻辑运算符 条件 2;

11.1.1 各类 AND 运算结果

思路：有 F 就是 F，否则有 N 就是 N。

运算结果	T	F	NULL
T	T AND T = T	T AND F = F	T AND N = N
F	F AND T = F	F AND F = F	F AND N = F
NULL	N AND T = N	N AND F = F	N AND N = N

表 11.1: 各类 AND 运算结果表

11.1.2 各类 OR 运算结果

思路：有 T 就是 T，否则有 N 就是 N。

运算结果	T	F	NULL
T	T OR T = T	T OR F = T	T OR N = T
F	F OR T = T	F OR F = F	F OR N = N
NULL	N OR T = T	N OR F = N	N OR N = N

表 11.2: 各类 OR 运算结果表

11.1.3 各类 NOT 运算结果

1. NOT T = F
2. NOT F = T
3. NOT NULL = N

11.2 IF 语句

在 PL/SQL 中最常见的分支（条件）语句就是 IF 语句。PL/SQL 语言的 IF 语句的结构与其它结构化语言中 IF 语句的结构完全相似。

11.2.1 IF 语句语法

Listing 11.1: IF 语法

```
1 IF 条件 THEN
2     statements
3 [ELSIF 条件 THEN
4     statements;]
5 [ELSE
6     statements;]
7 END IF;
```

Listing 11.2: IF 语法示例 1

```
1 IF condition THEN
2     -- 仅在条件为真时执行的代码块
3 END IF;
4
5 IF condition1 THEN
6     -- 在条件1为真时执行的代码块
7 ELSIF condition2 THEN
8     -- 在条件2为真时执行的代码块
9 END IF;
```

```
10
11 IF condition1 THEN
12     -- 在条件1为真时执行的代码块
13 ELSE
14     -- 在条件1不为真时执行的默认代码块
15 END IF;
```

条件 (**condition**) 是一个返回 True、False 或 NULL 的布尔变量或表达式

THEN 引出一个子句

Statements 可以是一个或多个 PL/SQL 或 SQL 语句

ELSIF 引入一个布尔表达式的关键字

ELSE 与之前所有的条件都不成立时引入默认子句

END IF IF 语句结束的标志，END IF 后需要接 ; 以结束

11.2.2 IF 语句示例

```
1 DECLARE
2     v_age number := &p_age;
3     v_gender CHAR(1) := '&p_sex';
4 BEGIN
5     IF (v_age BETWEEN 18 AND 35) AND (v_gender = 'F')
6     THEN
7         DBMS_OUTPUT.PUT_LINE('符合研究生入学年龄');
8     END IF;
9 END;
10 /
```

运行结果：

输入 p_age 的值：22

输入 p_sex 的值：F

符合研究生入学年龄

11.2.3 注意

1. 尽管可以使用 IF 语句中嵌套一个或多个 IF 语句的方式构造出多路分支语句，但是这样做会使程序的逻辑变得复杂并且易读性下降。因此，在实际工作中应该尽量避免使用 IF 语句的嵌套，而使用 ELSIF 子句。
2. PL/SQL 的逻辑表达式求值（又称最小化求值）是一种逻辑运算符策略。只有当第一个运算数的值无法确定逻辑运算结果时，才对第二个运算数进行求值。

11.3 CASE 表达式

11.3.1 CASE 表达式语法

Listing 11.3: 带选择器

```
1 CASE selector
2   WHEN 表达式1 THEN 结果1
3   WHEN 表达式2 THEN 结果2
4   ...
5   WHEN 表达式N THEN 结果N
6   [ELSE 结果 N+1]
7 END;
```

PL/SQL 还提供了一种搜索 CASE 表达式。这种搜索 CASE 表达式没有选择器，而是 WHEN 子句本身包含了产生一个布尔值的搜索条件，因此 WHEN 子句表达式不能产生其他任何类型的值。

Listing 11.4: 不带选择器

```
1 CASE
2   WHEN 表达式1 THEN 结果1
3   WHEN 表达式2 THEN 结果2
4   ...
5   WHEN 表达式N THEN 结果N
6   [ELSE 结果 N+1]
7 END;
```

11.3.2 CASE 表达式示例

Listing 11.5: CASE 示例代码

```
1 DECLARE
2   v_degree CHAR(1) := UPPER('&p_degree');
3   v_description VARCHAR2(250);
4 BEGIN
5   v_description :=
6     CASE v_degree
7       WHEN 'B' THEN '此人拥有学士学位。'
8       WHEN 'M' THEN '此人拥有硕士学位。'
9       WHEN 'D' THEN '此人拥有博士学位。'
10      ELSE '此人拥有壮士学位。'
11    END;
12   DBMS_OUTPUT.PUT_LINE(v_description);
13 END;
14 /
```

运行结果：

输入 p_degree 的值： B

此人拥有学术学位。

CASE 表达式的结果（返回值）可以是任何数据类型，只要与要赋值的变量的数据类型兼容（匹配）就可以。

11.4 CASE 语句

CASE 表达式测试条件并返回一个值，而 CASE 语句测试条件并执行一个操作。一个 CASE 语句可以是一个完整的 PL/SQL 程序块。CASE 语句必须以“END CASE;”结束，而 CASE 表达式必须以“END;”结束。

11.4.1 CASE 语句示例代码

```
1 DECLARE
```

```
2  v_degree CHAR(1) := UPPER('&p_degree');
3  BEGIN
4  CASE v_degree
5      WHEN 'B' THEN
6          DBMS_OUTPUT.PUT_LINE('此人拥有学士学位。');
7      WHEN 'M' THEN
8          DBMS_OUTPUT.PUT_LINE('此人拥有硕士学位。');
9      WHEN 'D' THEN '此人拥有博士学位。'
10         DBMS_OUTPUT.PUT_LINE('此人拥有博士学位。');
11      WHEN 'X' THEN
12         DBMS_OUTPUT.PUT_LINE('此人拥有壮士学位。');
13  END CASE;
14  END;
15  /
```

运行结果:

输入 p_degree 的值: B

此人拥有学术学位。

11.4.2 使用 CASE 语句取代 DECODE

利用 CASE 表达式或 CASE 语句来取代 SQL 函数的 DECODE 可以使程序变得更加清晰。

Listing 11.6: DECODE 函数

```
1  v_grade := DECODE(v_score,
2      90, 'A',
3      80, 'B',
4      70, 'C',
5      'D'
6  );
```

Listing 11.7: CASE 表达式

```
1  CASE v_score
```

```
2  WHEN 90 THEN v_grade := 'A'
3  WHEN 80 THEN v_grade := 'B'
4  WHEN 70 THEN v_grade := 'C'
5  ELSE v_grade := 'D'
6  END CASE;
```

11.5 GOTO

虽然在 PL/SQL 程序设计语言中有 GOTO 语句，但是许多 PL/SQL 语言的教材并不介绍这一个极具有争议的语句。GOTO 语句一直是批评和争论的焦点，主要是负面影响是使用 GOTO 语句使程序的可读性变差，甚至成为不可维护的“面条代码”。

语法：GOTO 语句标号

11.5.1 用处

1. **错误处理和清理：**有时，当在代码中发生错误时，需要跳转到一个错误处理和清理代码块。这个错误处理代码块通常包括关闭文件、释放资源、记录错误日志等操作。在这种情况下，使用 GOTO 语句可以让你跳转到错误处理部分，以确保资源得到释放，并记录错误信息。
2. **特定性能优化：**在某些特殊情况下，使用 GOTO 语句可以实现更高效的代码执行，特别是在循环或嵌套结构中。这通常需要精心考虑和测试，以确保性能的提高值得引入 GOTO。

Listing 11.8: 错误处理示例代码

```
1  BEGIN
2      -- 一些代码
3      -- 发生错误时跳转到错误处理部分
4      IF error_condition THEN
5          GOTO error_handler;
6      END IF;
7      -- 正常代码继续执行
8      -- ...
9      EXIT; -- 跳过错误处理部分
10
```

```
11  <<error_handler>> -- 错误处理部分
12  -- 执行清理操作、记录错误日志等
13  -- ...
14  END;
15  /
```

Listing 11.9: 性能提升示例代码（循环）

```
1  DECLARE
2      total_processed NUMBER := 0;
3      max_iterations NUMBER := 1000000; -- 假设有一个很大的数据集
4      iteration NUMBER := 0;
5  BEGIN
6      <<main_loop>> -- 主循环
7      IF iteration >= max_iterations THEN
8          GOTO end_loop; -- 跳过剩余迭代
9      END IF;
10     -- 执行数据处理操作
11     total_processed := total_processed + 1;
12     iteration := iteration + 1;
13     GOTO main_loop; -- 继续下一次迭代
14
15     <<end_loop>> -- 结束循环
16     DBMS_OUTPUT.PUT_LINE('处理完成，总共处理了 ' || total_processed || ' 条数据。
17         ');
18 END;
```

第十二章 循环语句

所谓的循环就是多次重复一个语句或语句序列。PL/SQL 提供了若干个循环结构以控制语句的重复执行。循环主要用于重复执行一些语句直到一个条件满足为止。在一个循环中必须要有一个退出条件，否则就是一个死循环。

PL/SQL 提供一下三种类型的循环结构：

1. **基本循环**：执行无条件的重复操作。
2. **WHILE 循环**：执行基于一个条件的重复操作。
3. **FOR 循环**：执行基于一个计数器的重复操作。

建议：

- 当语句在循环中至少要执行一次时，一般使用基本循环。
- 如果在每次开始重复时必须测试条件，一般使用 WHILE 循环。
- 如果重复次数为已知，一般使用 FOR 循环。

12.1 基本循环

12.1.1 基本循环语法

最简单的循环语句就是基本循环，它是由包含在 **LOOP** 和 **END LOOP** 关键字之间的一个语句序列构成。每次执行的流程达到 **END LOOP** 语句时，程序的控制就是转到 **LOOP** 结构的最上面。基本循环运行它的语句至少执行一次，即使在进入这个循环时 **EXIT** 条件已经满足，也要执行一次。如果没有 **EXIT** 语句，基本循环是一个死循环。

语法：

Listing 12.1: 基本循环语法

```
1 LOOP -- 定界符
```

```
2 statement; -- 一个或多个语句
3 ...
4 EXIT [WHEN condition]; -- 退出语句(EXIT语句)
5 END LOOP; -- 定界符
```

Condition 一个布尔变量或者布尔表达式。

Statement 一个或多个 SQL 语句或一个或多个 PL/SQL 语句。

12.1.2 基本循环示例

Listing 12.2: 基本循环示例代码

```
1 DECLARE
2     v_empno emp_pl.empno%TYPE;
3     v_deptno emp_pl.deptno%TYPE := &p_deptno;
4     v_sal emp_pl.sal%TYPE := &p_sal;
5     v_hiredate emp_pl.hiredate%TYPE := SYSDATE;
6     v_job emp_pl.job%TYPE := '&p_job';
7     v_counter NUMBER(2) := 1;
8     v_max_num NUMBER(2) := &P_max_num;
9 BEGIN
10     SELECT MAX(empno) INTO v_empno FROM emp_pl;
11     LOOP
12         INSERT INTO emp_pl(empno, hiredate, job, sal, deptno)
13         VALUES((v_empno + v_counter), v_hiredate, v_job, v_sal, v_deptno);
14         v_counter := v_counter + 1;
15         EXIT WHEN v_counter > v_max_num;
16     END LOOP;
17 END;
```

12.2 WHILE 循环

12.2.1 WHILE 循环语法

PL/SQL 还提供了一种与基本（LOOP）循环相似的 WHILE LOOP 循环语句，但是使用 WHILE 循环条件为真（TRUE）时重复执行循环体中的语句，而当条件不再是 TRUE（F 或 N）时退出循环。

循环的条件是在每次重复开始时测试，这种循环是在条件为 FALSE 或 NULL 终止。WHILE LOOP 的循环语句语法如下：

Listing 12.3: WHILE 循环语法

```
1 WHILE 条件 LOOP
2     语句1;
3     语句2;
4     ...
5 END LOOP;
```

在 WHILE LOOP 循环语句中，循环的条件必须放在 WHILE 和 LOOP 两个关键字之间，而循环的条件是在每次重复开始时测试的。与 LOOP 循环语句相同，在 WHILE LOOP 循环语句中所包含的语句既可以是 PL/SQL 语句也可以是 SQL 语句。

12.2.2 WHILE 循环示例

Listing 12.4: WHILE 循环示例代码

```
1 DECLARE
2     v_deptno dept.deptno%TYPE;
3     v_loc dept.loc%TYPE := '&p_loc';
4     v_counter NUMBER(2) := 1;
5     v_max_num NUMBER(2) := &p_max_num;
6 BEGIN
7     WHILE v_counter <= v_max_num LOOP;
8         INSERT INTO dept_pl(deptno, loc)
9         VALUES(deptid_sequence.NEXTVAL, v_loc);
10        v_counter := v_counter + 1;
```



```
11  END LOOP;  
12  END;  
13  /
```

12.3 FOR 循环

12.3.1 FOR 循环语法

For 循环具有与基本循环一样的通用结构，不过 For 循环在 LOOP 关键字之前有一个控制语句用来设置 PL/SQL 程序执行的重复次数。For 循环具有如下特性：

1. 使用 FOR 循环简化了对重复次数的测试。
2. 不用声明计数器，它是隐含声明的。
3. 在语法上需要说明下限和上限。

语法：

Listing 12.5: FOR 循环语法

```
1  FOR counter IN [REVERSE] lower_bound..upper_bound LOOP  
2      statement1;  
3      statement2;  
4      ...  
5  END LOOP;
```

counter 是一个隐含声明的整型数，其值在循环的每次重复时自动增加或减少（默认是增加）。

REVERSE 指定计数器每次重复时是从上限到下限递减（默认是从下限到上限）。

lower_bound 下限。

upper_bound 上限。

语句序列每执行一次，计数器默认加 1。除了整数之外，循环范围的下限和上限还可以是文字、变量和表达式，但是必须可以转换成整数的数据类型。如果下限和上限不是整数，PL/SQL 按四舍五入的方法将其转为整数。因此，13/2 和 17/3 都是有效的上限或下限值。

12.3.2 FOR 循环示例

Listing 12.6: FOR 循环示例代码

```
1 DECLARE
2     v_deptno dept.deptno%TYPE;
3     v_loc dept.loc%TYPE := '&p_loc';
4     v_max_num NUMBER(2) := &p_max_num;
5 BEGIN
6     FOR i IN 1..v_max_num LOOP
7         INSERT INTO dept_pl(deptno, loc);
8         VALUES(deptid_sequence.NEXTVAL, v_loc);
9     END LOOP;
10 END;
11 /
```

使用 FOR 循环时应该遵守的规则：

1. 只在循环中引入计数器，计数器在循环外没有定义。
2. **不要为计数器赋值**，即不要将计数器变量放在赋值运算符的左边。
3. 上限和下限都不应该为空值 NULL。

12.4 循环的嵌套与标号

12.4.1 介绍

循环体本身也是一个 PL/SQL 程序块，所以 PL/SQL 的循环同样可以嵌套，而且可以进行多层嵌套、将基本循环、WHILE 循环和 FOR 循环彼此之间混合嵌套。

一个被嵌套的循环的结束并不结束包含它的循环（除非出现异常）。不过，可以使用标号以区分不同的块和循环体，并引用带有标号的 EXIT 语句直接退出外层循环。

标号的命名规则与其它标识符完全相同（在之前条件语句中，GOTO 章节也用到了标号）。标号必须放在一个语句之前，既可以是在同一行，也可以是在单独的行。为基本循环加标号时，标号要放在关键字 LOOP 之前并用标号定界符“«”和“»”括起来，如«dog_loop»，其中 dog_loop 为标号。在 FOR 循环和 WHILE 循环中，标号要放在关键字 FOR 或 WHILE 之前。

12.4.2 循环的嵌套与标号示例

程序有 2 个嵌套的 FOR 循环，用户利用代替变量输入一个正整数 n ，之后再外循环中的程序计算出 $1 \sim n$ 所有自然数之和。内循环计算所输入自然数的阶乘 ($n!$)。当 $i+j>4$ 成立时，程序就跳转到 Inner_loop 标号所再得语句。其中， i 和 j 分别为外循环和内循环的计数。

Listing 12.7: 嵌套循环示例代码

```
1  DECLARE
2  v_t NUMBER := 0;
3  v_f NUMBER := 1;
4  v_num NUMBER := 10;
5  BEGIN
6  <<Outer_Loop>>
7  FOR i IN 1..v_num LOOP
8      v_t := v_t + i;
9      DBMS_OUTPUT.PUT_LINE('1~'||i||'自然数的总和是: '||v_t);
10     <<Inner_Loop>>
11     FOR j IN 1..v_num LOOP
12         EXIT Inner_Loop WHEN i+j >4;
13         v_f := v_f * j;
14         DBMS_OUTPUT.PUT_LINE('自然数'||j||'的阶乘是: '||v_f);
15     END LOOP;
16     v_f := 1;
17 END LOOP Outer_Loop;
18 END;
19 /
```

12.5 CONTINUE

12.5.1 介绍

Oracle 11g 和 Oracle 12c 引入了 CONTINUE 语句，该语句能够使程序的控制转移到循环体中的下一次新循环或者直接离开循环。通常使用 CONTINUE 语句在主要的处理开始之前在一

个循环内部过滤掉不需要的数据（和处理操作）。

12.5.2 CONTINUE 示例

注意：此代码效果与嵌套循环示例代码效果相同。

Listing 12.8: CONTINUE 示例代码

```
1 DECLARE
2   v_t NUMBER := 0;
3   v_f NUMBER := 1;
4   v_num NUMBER := 20;
5 BEGIN
6   <<Outer_Loop>>
7   FOR i IN 1..v_num LOOP
8     v_f := 1;
9     v_t := v_t + i;
10    DBMS_OUTPUT.PUT_LINE('1~'||i||'自然数的总和是: '||v_t);
11    <<Inner_Loop>>
12    FOR j IN 1..v_num LOOP
13      CONTINUE Outer_Loop WHEN i+j >4;
14      v_f := v_f * j;
15      DBMS_OUTPUT.PUT_LINE('自然数'||j||'的阶乘是: '||v_f);
16    END LOOP;
17  END LOOP Outer_Loop;
18 END;
19 /
```

第十三章 组合数据类型

一个组合类型的变量可以存放多个变量类型的值或多个组合类型的值。

与标量数据类型不同，组合数据类型包含了内部结构（组件），一旦定义了一个数据类型，这个数据类型就可以重用——用来定义一个或多个组合类型变量。

PL/SQL 语言中的组合数据类型可以分为以下几类：

1. PL/SQL 记录（records）：将逻辑上相关的但是类型不同的数据当作一个逻辑单元来处理，**一个 PL/SQL 记录可以包含多个不同类型的数据**，如可以定义一个存储客户详细信息的记录。
2. PL/SQL 集合（collections）：将一组（集合）数据当作一个单独的单元来处理。
 - (a) INDEX BY 表
 - (b) 嵌套表（Nested Table）
 - (c) 变长数组（VARRAY）

PL/SQL 记录和 PL/SQL 集合都是组合数据类型，在实际编程中选择的原则：

- 如果要存储和操作的数据是逻辑上相关的但是具有不同的数据类型，一般使用 PL/SQL 记录。
- 如果要存储和操作的数据具有相同的数据类型，而且其数据类型本身又可以是组合类型，使用 INDEX BY 表。

13.1 PL/SQL 记录（records）

13.1.1 介绍

所谓一个 PL/SQL 记录就是一组存储在若干字段中的相关联的数据（类似对象），而记录中的每个字段都具有自己的名字和数据类型。

PL/SQL 记录具有如下特性：

1. 一定包含一个或多个被称为字段的组件，其组件是任何的标量、记录或 INDEX BY 表的数据类型。
2. 在结构上与第三代语言（3GL）的记录相似。
3. **与数据库中表的行不同。**
4. 每一个定义记录可以根据实际需要有任何多个字段。
5. 可以为记录赋初始值，也可以将记录定义为非空（NOT NULL），没有初始值的字段其初始值被初始化为空（NULL），也可以使用关键字 DEFAULT 为其定义初始值。
6. 是将字段的集合当作一个逻辑单元来处理。
7. 可以在任何程序块（如子程序或软件包）的声明部分定义记录类型并声明用户定义的记录变量。
8. 一个记录可以是另一个记录的组件。
9. 当从一个表中获取一行数据时，使用记录处理非常方便。

13.1.2 创建记录语法

与创建变量类型数据不同，创建一个 PL/SQL 记录变量需要两步：

1. **创建一个 PL/SQL 记录数据类型。**
2. 使用这个已创建的 PL/SQL 记录数据类型来**定义（声明）变量**。

创建记录数据类型的语法：

Listing 13.1: 创建记录类型的语法

```
1 TYPE 数据类型名 IS RECORD(  
2     字段声明[, 字段声明],  
3     字段声明[, 字段声明],  
4     ...  
5     字段声明[, 字段声明]  
6 );
```

字段的声明部分为：

Listing 13.2: 字段声明语法

```
1  字段名 { 字段类型 | 变量%TYPE  
2    | 表名.列名%TYPE | 表名%ROWTYPE}  
3  [[NOT NULL] {:= DEFAULT} 表达式]
```

定义这一记录类型的变量的语法：

Listing 13.3: 定义变量

```
1  标识符 数据类型名；
```

数据类型名 为记录（RECORD）类型的名字。

字段名 为记录内部一个字段的名字。

字段类型 为该字段的数据类型。

表达式 为字段数据类型的表达式或初始值。

引用内部组件格式：**记录名.字段名**

13.1.3 创建记录示例代码

Listing 13.4: 使用记录类型变量示例

```
1  DECLARE  
2  -- 步骤 1: 创建 PL/SQL 记录数据类型  
3  TYPE EmployeeRecord IS RECORD (  
4    employee_id NUMBER,  
5    first_name VARCHAR2(50),  
6    last_name VARCHAR2(50),  
7    hire_date DATE  
8  );  
9  -- 步骤 2: 声明一个记录类型的变量  
10 emp_info EmployeeRecord;  
11 BEGIN
```

```
12  -- 分配值给记录的字段
13  emp_info.employee_id := 101;
14  emp_info.first_name := 'John';
15  emp_info.last_name := 'Doe';
16  emp_info.hire_date := TO_DATE('2023-01-15', 'YYYY-MM-DD');
17  -- 访问记录中的字段
18  DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_info.employee_id);
19  DBMS_OUTPUT.PUT_LINE('Employee Name: ' || emp_info.first_name || ' ' ||
    emp_info.last_name);
20  DBMS_OUTPUT.PUT_LINE('Hire Date: ' || TO_CHAR(emp_info.hire_date, 'YYYY-MM-
    DD'));
21  END;
```

13.1.4 %ROWTYPE

使用%TYPE 属性每次只能定义一个变量，如果要访问或操作表中一行数据的每一列时，必须使用%TYPE 属性基于每一列定义一个相应的变量。PL/SQL 提供的%ROWTYPE 属性会极大地简化以上过程，可以利用%ROWTYPE 属性声明一个能够存储一个表或视图中一整行的记录（变量）。该记录中的每一个字段的名称和数据类型取自表或视图中对应的列。

%ROWTYPE 属性声明的记录变量的特性：

1. 可以按照数据库中一个表或视图中的列的集合来声明一个变量。
2. 将数据库表冠在%ROWTYPE 的前面。
3. 记录中的字段与表或视图中的列有相同的名称和相同的数据类型。

使用语法：

Identifier reference%ROWTYPE;

Identifier 记录类型名。

reference 表名、视图名、游标等。

使用示例：

Listing 13.5: %ROWTYPE 示例代码

```
1 DECLARE
2     -- 声明一个记录变量，具有与查询结果行相同的结构
3     emp_info employees%ROWTYPE;
4 BEGIN
5     -- 使用 SELECT 语句将查询结果中的一行数据赋予记录
6     SELECT * INTO emp_info FROM employees WHERE employee_id = 101;
7     -- 访问记录中的字段
8     DBMS_OUTPUT.PUT_LINE('Employee Name: ' || emp_info.first_name || ' ' ||
9                             emp_info.last_name);
9     -- 可以继续访问其他字段
10 END;
```

%ROWTYPE 的好处:

1. 不需要知道所基于的表中列的数目和数据类型。
2. 在运行期间所基于的表中列的数目和数据类型可能变化，但记录不需要修改。
3. 当使用 SELECT * 语句提取一行数据时，这个属性很有用。

实际上，当无法确定所基于的数据库表的结构时，使用%ROWTYPE 属性应该是最合适的，使用%ROWTYPE 属性最主要的好处就是简化 PL/SQL 程序代码的维护，当所基于的表发生变化时，使用%ROWTYPE 属性就可以确保利用这一属性声明的变量的数据类型动态的变化。如果一个 DDL 语句更改了表中的列，那么相应的 PL/SQL 程序单元被置为无效。而当这个程序被重新编译时，该程序将自动地反映这个表的新结构。

13.2 INDEX BY 表

INDEX BY 表（也称为索引表或关联数组）是一种特殊类型的数据结构，其允许使用索引值作为键来访问和操作元素。与传统的数据库表不同，INDEX BY 表是内存中的数据结构，通常用于存储临时数据、中间结果或在程序执行期间动态管理数据。

特性:

1. 由两个组件（两列）所组成:

(a) 数据类型为 `BINARY_INTEGER` 和 `PLS_INTEGER` 的“主键”。

(b) 标量或记录数据类型的列。

2. 没有界限，其大小可以动态地增加。

主键一般使用 `BINARY_INTEGER` 或 `PLS_INTEGER` 数据类型，因为与 `NUMBER` 类型的数据相比，`BINARY_INTEGER` 或 `PLS_INTEGER` 数据需要较小的存储空间。

13.2.1 INDEX BY 表创建语法

创建一个 `INDEX BY` 表型变量同样需要两步：

1. 声明一个 `INDEX BY` 表的数据类型。

2. 利用以上声明的 `INDEX BY` 表数据类型声明一个这一数据的变量。

Listing 13.6: 创建 `INDEX BY` 表类型语法

```
1 TYPE 数据类型名 IS TABLE OF
2   {列数据类型 | 变量%TYPE | 表名.列名%TYPE | 表名%ROWTYPE} [NOT NULL]
3   INDEX BY [PLS_INTEGER | BINARY_INTEGER | VARCHAR2(<size>)];
4
5 标识符 数据类型名;
```

数据类型名 为 `INDEX BY` 表数据类型的名字。

列数据类型 为任何标量和组合数据类型，如 `NUMBER`、`DATE`、`VARCHAR2` 或 `%TYPE` 等。

标识符 为表示整个 `INDEX BY` 表的标识符的名字。

注意：如果没有 `INDEX BY`，将声明为嵌套表，其通过索引进行访问。

13.2.2 INDEX BY 表使用示例

Listing 13.7: `INDEX BY` 表使用示例 1

```
1 DECLARE
2   -- 声明一个索引表类型，使用字符串作为主键
3   TYPE EmployeeInfo IS TABLE OF VARCHAR2(100) INDEX BY VARCHAR2(50);
```

```
4  -- 声明一个索引表变量
5  employee_data EmployeeInfo;
6  BEGIN
7  -- 向索引表中插入数据, 使用员工姓名作为主键
8  employee_data('John Doe') := 'Software Engineer';
9  employee_data('Alice Smith') := 'Project Manager';
10 employee_data('Bob Johnson') := 'Data Analyst';
11 -- 访问索引表中的数据
12 DBMS_OUTPUT.PUT_LINE('John Doe's Job: ' || employee_data('John Doe'));
13 DBMS_OUTPUT.PUT_LINE('Bob Johnson's Job: ' || employee_data('Bob Johnson'))
    ;
14 END;
```

Listing 13.8: INDEX BY 表使用示例 2

```
1  DECLARE
2  TYPE ename_table_type IS TABLE OF emp.ename%TYPE INDEX BY PLS_INTEGER;
3  TYPE hiredate_table_type IS TABLE OF DATE INDEX BY BINARY_INTEGER;
4  ename_table ename_table_type;
5  hiredate_table hiredate_table_type;
6  v_count NUMBER(6) := &p_count;
7  BEGIN
8  FOR i IN 1..v_count LOOP
9  ename_table(i) := '武大';
10 hiredate_table(i) := SYSDATE + 14;
11 DBMS_OUTPUT.PUT_LINE(ename_table(i) || ':' || hiredate_table(i));
12 END LOOP;
13 END;
14 /
```

13.2.3 INDEX BY 表方法

为了使 INDEX BY 表的使用更加容易，PL/SQL 引入了一些 INDEX BY 表的方法。一个 INDEX BY 表的方法就是在一个 INDEX BY 表上执行某种操作的一个内置函数或过程，它可以用点号 (.) 表示法调用，语法：（其中参数列表为一个或多个参数）

表名.方法名 [(参数列表)]

方法	描述
EXISTS(n)	如果第 n 个元素在 PL/SQL 表（数组）中存在，返回 TRUE
COUNT	返回一个 PL/SQL 表当前所包含的元素个数
FIRST	返回在一个 PL/SQL 表中第一个（最小的）下标数字 如果 PL/SQL 表是空的，返回 NULL
LAST	返回在一个 PL/SQL 表中第一个（最大的）下标数字 如果 PL/SQL 表是空的，返回 NULL
PRIOR(n)	返回在一个 PL/SQL 表中当前元素的前 n 个元素的下标值
NEXT(n)	返回在一个 PL/SQL 表中当前元素的后 n 个元素的下标值
DELETE	DELETE 即删一个 PL/SQL 表中的全部元素 DELETE(n) 即删一个 PL/SQL 表中的第 n 个元素 DELETE(m,n) 即删一个数组中 m n 范围内的元素

表 13.1: INDEX BY 表方法表

使用示例：

Listing 13.9: 表方法示例代码

```
1  -- 声明一个 INDEX BY 表类型
2  TYPE ScoresTable IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
3  -- 声明 INDEX BY 表变量
4  scores ScoresTable;
5  -- 使用方法向 INDEX BY 表中插入元素
6  scores(1) := 95;
7  scores(2) := 88;
8  -- 使用方法获取元素的值
9  DBMS_OUTPUT.PUT_LINE('Score at index 1: ' || scores(1));
```

```
10 -- 使用方法删除元素
11 scores.DELETE(1); -- 删除索引为1的元素
```

13.2.4 INDEX BY 记录表

INDEX BY 记录表，就是定义一个存放一个表中整行数据（每一列）的 INDEX BY 表变量。

Listing 13.10: INDEX BY 记录表使用示例

```
1 DECLARE
2     -- 声明一个 INDEX BY 记录表类型，该类型表示一个员工记录
3     TYPE EmployeeRecord IS RECORD (
4         emp_id NUMBER,
5         emp_name VARCHAR2(50),
6         emp_salary NUMBER
7     );
8     -- 声明一个 INDEX BY 记录表，每个元素存储一个员工记录
9     TYPE EmployeeTable IS TABLE OF EmployeeRecord INDEX BY PLS_INTEGER;
10    -- 声明一个 INDEX BY 记录表变量
11    employees EmployeeTable;
12 BEGIN
13     -- 向 INDEX BY 记录表中插入数据
14     employees(1) := EmployeeRecord(101, 'John Doe', 50000);
15     employees(2) := EmployeeRecord(102, 'Alice Smith', 60000);
16     -- 访问 INDEX BY 记录表中的数据
17     DBMS_OUTPUT.PUT_LINE('Employee 1: ' || employees(1).emp_name || ' - Salary:
18         ' || employees(1).emp_salary);
19     DBMS_OUTPUT.PUT_LINE('Employee 2: ' || employees(2).emp_name || ' - Salary:
20         ' || employees(2).emp_salary);
21 END;
```

第十四章 异常处理

在运行期间所发生的这样的错误被称为异常。当一个异常发生时，程序块会被终止。这样的异常可以在 PL/SQL 程序块中被处理。PL/SQL 运行期间的错误可能来自系统的设计缺陷、程序代码错误、硬件问题以及许多其他的来源。因此，无法预计所有可能的错误，但是可以编译异常处理程序代码让操作在出现错误时可以继续正常执行。

Listing 14.1: 示范代码

```
1 DECLARE
2     v_job emp_pl.job%TYPE;
3 BEGIN
4     SELECT job INTO v_job
5     FROM emp_pl
6     WHERE job = '保安';
7     DBMS_OUTPUT.PUT_LINE(v_job);
8 EXCEPTION
9     WHEN TOO_MANY_ROWS THEN
10        DBMS_OUTPUT.PUT_LINE('太多行了，请使用cursor解决问题。');
11 END;
12 /
```

预期输出为：**太多行了，请使用 cursor 解决问题。**

有了异常段，虽然程序和之前的一样，SELECT INTO 语句提取了多行数据问题依然存在，但这次这段 PL/SQL 程序代码就可以**正常结束**。与之前的 PL/SQL 程序代码非正常终止不同，这段程序代码是被成功地执行了。当一个异常被抛出时，程序的控制流程就转移到所定义的异常段并且执行该异常段中所有语句。

14.1 抛出异常

抛出异常的方法有两种：

1. **自动抛出异常**：当出现一个 Oracle 错误时，相关的预定义或非预定义异常被自动地抛出。
2. **显式抛出异常**：基于业务功能由程序员的程序实现，程序员可能必须要显式的抛出一个异常。通过在程序块中使用 **RAISE 语句** 显式地抛出一个异常，这个抛出的异常既可以是用户定义的，也可以是预定义的。

14.2 异常处理方式

PL/SQL 中有两种处理异常方法：

1. 捕获异常
2. 传播异常

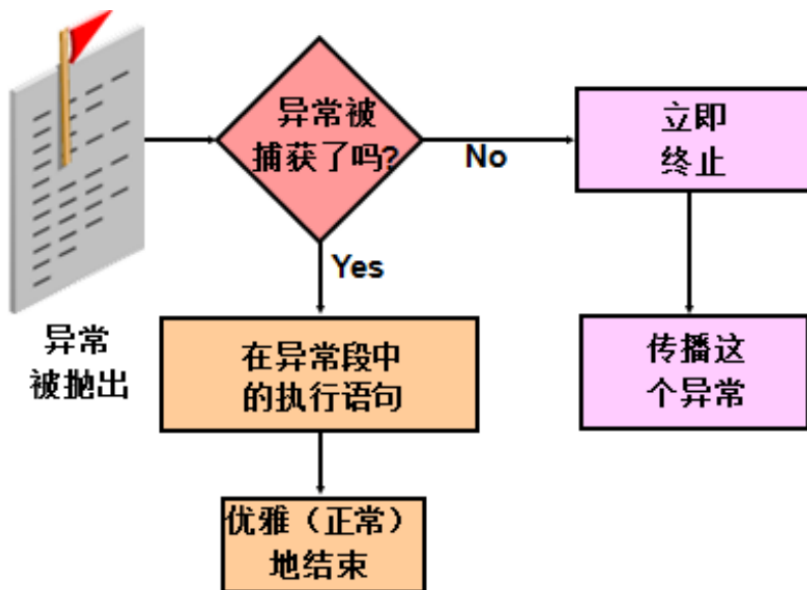


图 14.1: 异常处理方式示意图

14.2.1 捕获异常

在 PL/SQL 程序中包含了一个**异常段（EXCEPTION section）**以捕获异常。如果异常在这个程序的执行段中被抛出，那么处理就自动跳转到这个程序的异常段中的相应异常处理程序。

如果异常处理程序成功地处理了这个异常，那么这个异常就不会传播到包含它的程序段，也不会传播到调用环境，而这个 PL/SQL 程序块成功地结束。

语法：

Listing 14.2: 捕获异常语法

```
1 EXCEPTION
2  -- 处理指定异常的代码
3  WHEN 异常1 [或 异常2 ...] THEN
4      语句1;
5      语句2;
6      ...
7  [WHEN 异常1 [或 异常2 ...] THEN
8      语句1;
9      语句2;
10     ...]
11 [WHEN 异常1 [或 异常2 ...] THEN
12     语句1;
13     语句2;
14     ...]
15 -- 处理其他异常的代码
16 [WHEN OTHERS THEN
17     语句1;
18     语句2;
19     ...]
```

异常 (exception) 一个预定义异常的标准名（如 TOO_MANY_ROWS）或在声明段中用户定义的异常名。

语句 (statement) 一个或多个 PL/SQL 或 SQL 语句。

OTHERS 一个可选的异常处理子句，该子句捕获任何没有显式处理的异常（就是在之前的所有 WHEN 子句都没有捕获的异常）

可以在一个异常段中包含任意多个异常处理程序（实际上就是任意多个 WHEN 子句）以处理说明的异常。然而，对应一个单一的异常不能有多多个处理程序；异常处理段只捕获那些声明了的异常，而任何其他的异常都不捕获，除非使用了 OTHERS 异常处理程序。

Listing 14.3: 捕获异常示例代码

```
1 BEGIN
2     -- 主要执行部分
3 EXCEPTION
4     WHEN ZERO_DIVIDE THEN
5         -- 处理除零异常的代码
6     WHEN OTHERS THEN
7         -- 处理其他异常的代码
8 END;
```

14.2.2 传播异常

如果一个异常在程序的执行段被抛出并且没有对应的异常处理程序，那么**这个 PL/SQL 程序块以失败而终止**，并且这个异常被传播到包含它的程序块或调用环境，调用环境可以是任何应用程序。

使用 RAISE 传播异常

Listing 14.4: RAISE 传播异常示例代码

```
1 BEGIN
2     -- 主要执行部分
3 EXCEPTION
4     WHEN ZERO_DIVIDE THEN
5         -- 处理除零异常的代码
6         RAISE;    -- 在捕获的异常块中重新引发异常
7     WHEN OTHERS THEN
8         -- 处理其他异常的代码
9         RAISE;    -- 在捕获的异常块中重新引发异常
```

```
10 END;
```

使用 RAISE_APPLICATION_ERROR 传播异常

RAISE_APPLICATION_ERROR 是一个 Oracle 提供的过程，允许抛出一个错误号和消息，这个错误号的范围必须在-20000 到-20999 之间，以避免与 Oracle 内部错误码冲突。这允许你传播一个带有自定义消息的异常，这在封装异常信息和创建更具描述性的错误信息时非常有用。

Listing 14.5: RAISE_APPLICATION_ERROR 传播异常示例代码

```
1 EXCEPTION
2   WHEN my_exception THEN
3     RAISE_APPLICATION_ERROR(-20001, '自定义错误信息');
```

14.3 异常类型

在 PL/SQL 程序中可以使用的异常共有如下三种类型：

1. 预定义的 Oracle 服务器错误（异常）。
2. 非预定义的 Oracle 服务器错误（异常）。
3. 用户定义的错误（异常）。

14.3.1 处理预定义的 Oracle 服务器错误

Listing 14.6: 处理预定义的 Oracle 服务器错误示例

```
1 BEGIN
2   -- 主要执行部分
3 EXCEPTION
4   WHEN ZERO_DIVIDE THEN
5     -- 处理除零异常的代码
6   WHEN OTHERS THEN
7     -- 处理其他异常的代码
8 END;
```

常见预定义异常：

预定义异常名	Oracle服务器错误代码	描述
ACCESS_INTO_NULL	ORA_06530	试图为一个未初始化对象的属性赋值
CASE_NOT_FOUND	ORA_06592	在选择CASE语句的WHEN子句中没有选择条件并且没有ELSE子句
INVALID_CURSOR	ORA_01001	试图打开一个已打开的cursor
INVALID_NUMBER	ORA_01772	将字符串转换数字失败
LOGIN_DENIED	ORA_01017	以一个无效的用户名或密码登陆
NO_DATA_FOUND	ORA_01403	单行查询没有返回任何数据
NOT_LOGIN_ON	ORA_01012	在没有连服务器的情况下，PL\SQL程序发出了一个数据库调用

图 14.2: 常见预定义异常表

14.3.2 处理非预定义的 Oracle 服务器错误

非预定义的异常与预定义的异常非常相似，只是它们没有被定义为 Oracle 服务器中的 PL/SQL 异常而已，它们是标准的 Oracle 错误。需使用 **PRAGMA EXCEPTION_INIT** 将一个异常和一个 Oracle 错误代码关联起来，以便在捕获该异常时能够获取详细的错误信息。

PRAGMA(也被称为伪指令 “pesudoinstructions”) 是关键字，它表示这个语句是一个编译指令，而当 PL/SQL 程序块执行时不会被处理。PRAGMA 关键字指示 PL/SQL 编译器将这个程序块中出现的所有该异常名解释成相关的 Oracle 服务器错误代码。

Listing 14.7: 处理非预定义的 Oracle 服务器错误示例

```
1 DECLARE
2     no_permission_exception EXCEPTION;
3     PRAGMA EXCEPTION_INIT(no_permission_exception, -942);
4 BEGIN
5     FOR emp_rec IN (SELECT * FROM employees) LOOP
6         DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_rec.employee_id);
7     END LOOP;
8 EXCEPTION
9     WHEN no_permission_exception THEN -- 处理没有权限的异常
```

```
10 DBMS_OUTPUT.PUT_LINE('没有足够的权限访问 employees 表。');
11 WHEN OTHERS THEN -- 处理其他异常的代码
12 DBMS_OUTPUT.PUT_LINE('发生其他异常。');
13 END;
```

14.3.3 处理用户定义的错误

需使用 **RAISE my_exception;** 显式抛出自定义异常。

声明和捕获用户定义的异常步骤：

1. 在一个 PL/SQL 程序块的声明段中声明一个用户定义的异常。
2. 使用 RAISE 语句显式地抛出这个异常。
3. 在 EXCEPTION 段处理这个异常。

Listing 14.8: 处理用户定义的错误示例 1

```
1 DECLARE
2     my_custom_exception EXCEPTION;
3 BEGIN
4     -- 主要执行部分
5     IF some_condition THEN -- 在某个条件下显式抛出自定义异常
6         RAISE my_custom_exception;
7     END IF;
8 EXCEPTION
9     WHEN my_custom_exception THEN -- 处理自定义异常的代码
10    WHEN OTHERS THEN -- 处理其他异常的代码
11 END;
```

Listing 14.9: 处理用户定义的错误示例 2

```
1 DECLARE
2     -- 步骤 1: 在声明段中声明用户定义的异常
3     department_not_found EXCEPTION;
4     PRAGMA EXCEPTION_INIT(department_not_found, -20001);
```

```
5   v_department_id NUMBER := &p_department_id; -- 用户输入的部门号
6 BEGIN
7   -- 步骤 2: 检查部门是否存在, 如果不存在则抛出用户定义的异常
8   IF NOT EXISTS (SELECT 1 FROM departments WHERE department_id =
9       v_department_id) THEN
10      RAISE department_not_found;
11   END IF;
12   -- 在这里执行其他操作, 因为部门存在
13 EXCEPTION
14   -- 步骤 3: 在 EXCEPTION 段中捕获用户定义的异常并处理
15   WHEN department_not_found THEN
16      DBMS_OUTPUT.PUT_LINE('用户定义的异常: 部门不存在。');
```

14.4 捕获异常实例

Listing 14.10: 一个捕获异常的实例

```
1 DECLARE
2   -- 自定义异常
3   emp_not_found EXCEPTION;
4   invalid_dept EXCEPTION;
5   application_error EXCEPTION;
6   -- 假设的员工编号和部门编号
7   v_empno emp.empno%TYPE := 1001;
8   v_deptno dept.deptno%TYPE := 10;
9   -- 用于检查的变量
10  v_count NUMBER;
11 BEGIN
12   -- 检查员工是否存在
13   SELECT COUNT(*)
```

```
14 INTO v_count
15 FROM emp
16 WHERE empno = v_empno;
17 IF v_count = 0 THEN
18     RAISE emp_not_found;
19 END IF;
20 -- 检查部门是否有效
21 SELECT COUNT(*)
22 INTO v_count
23 FROM dept
24 WHERE deptno = v_deptno;
25 IF v_count = 0 THEN
26     RAISE invalid_dept;
27 END IF;
28 - 执行更新操作
29 UPDATE emp
30 SET deptno = v_deptno
31 WHERE empno = v_empno;
32 -- 提交事务
33 COMMIT;
34 EXCEPTION
35 WHEN emp_not_found THEN
36     DBMS_OUTPUT.PUT_LINE('员工编号 ' || v_empno || ' 不存在。');
37     ROLLBACK;
38 WHEN invalid_dept THEN
39     DBMS_OUTPUT.PUT_LINE('部门编号 ' || v_deptno || ' 无效。');
40     ROLLBACK;
41 WHEN OTHERS THEN -- 捕获其他所有未处理的异常
42     DBMS_OUTPUT.PUT_LINE('发生未知错误: ' || SQLERRM);
43     ROLLBACK;
44     RAISE; -- 重新抛出异常
```

```
45 END;  
46 /
```

14.5 SQLCODE 和 SQLERRM

PL/SQL 提供了两个函数，当一个异常发生时，可以通过使用这两个函数来标识相关的错误代码信息。这二个函数就是 SQLCODE 和 SQLERRM，**SQLCODE 函数为内部异常返回一个 Oracle 错误号码，而 SQLERRM 函数则返回与这个错误号码相关的信息。**

使用示例：

Listing 14.11: SQLCODE 和 SQLERRM 示例代码

```
1 DECLARE  
2     my_exception EXCEPTION;  
3 BEGIN  
4     -- 一个可能引发异常的 SQL 语句  
5     SELECT 1 / 0 INTO my_variable FROM dual;  
6 EXCEPTION  
7     WHEN OTHERS THEN  
8         -- 获取异常的错误代码  
9         DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);  
10        -- 获取异常的详细错误信息  
11        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);  
12 END;
```

14.6 处理异常建议

Oracle 推荐在开发捕获异常的 PL/SQL 程序代码时应该注意如下事项：

1. 以关键字 EXCEPTION 开始异常处理程序的程序段。
2. 在异常段中可以定义若干个异常处理程序（子句），每一个都有自己的一组操作。
3. 当一个异常发生时，PL/SQL 在离开这个异常段之前只执行一个异常处理子句。

4. 将 OTHERS 子句放在所有其他异常处理子句之后。
5. 在一个异常段中只能有一个 OTHER 子句。
6. 异常不能出现在赋值语句中，也不能出现在 SQL 语句中。

使用 SQLCODE 和 SQLERRM 的建议：

Listing 14.12: 记录异常（需先建立错误记录表）

```
1 EXCEPTION
2 WHEN OTHERS THEN
3     ROLLBACK;
4     v_error_code := SQLCODE;
5     v_error_message := SQLERRM;
6     INSERT INTO errors (user_name, error_date, error_code, error_message)
7     VALUES (USER, SYSDATE, v_error_code, v_error_message);
8     COMMIT;
```

1. **监测频繁的错误：**通过查询 errors 表并计算每个 error_code 的出现次数，可以识别出频繁出现的错误。这通常通过一个简单的 SELECT error_code, COUNT(*) FROM errors GROUP BY error_code 查询来实现。
2. **为常见错误定义异常：**对于每个频繁出现的错误代码，可以声明一个特定的异常，并使用 PRAGMA EXCEPTION_INIT 将其与错误代码关联。这样就可以在 PL/SQL 程序中直接引用这些异常。
3. **增加异常处理逻辑：**在 PL/SQL 块的异常部分，可以为每个常见的错误代码增加一个 WHEN 子句。在这个子句中，可以编写特定的处理逻辑，如记录额外的信息、发送通知或采取其他纠正措施。
4. **清理 errors 表：**在处理完一个常见错误后，可以从 errors 表中删除与该错误代码关联的所有记录。这样做可以确保 errors 表中只保留那些偶尔发生的、不常见的错误。
5. **代码维护和调试：**使用 SQLCODE 和 SQLERRM 可以在调试阶段帮助开发者理解错误的原因。一旦调试完毕，这些语句可以被注释掉或删除，以避免生产环境中不必要的性能开销。

第十五章 过程

在 PL/SQL 程序设计中，**子程序（也称为过程或函数）**是模块化程序设计的基础。要使子程序更灵活，重要的一点是可以改变所操作的数据（即可以通过计算，也可以通过使用输入参数传递给一个子程序），而子程序计算的结果可以通过输出（OUT）参数返回给子程序的调用者。**子程序的目的是通过将代码组织成可重用的单元来提高代码的可维护性和可读性。**

Listing 15.1: 子程序示例

```
1  --定义过程
2  CREATE OR REPLACE PROCEDURE CalculateSquare(
3      p_number IN NUMBER,
4      p_result OUT NUMBER
5  ) AS
6  BEGIN
7      -- 计算平方
8      p_result := p_number * p_number;
9  END;
10
11 --调用过程
12 DECLARE
13     v_input NUMBER := 5;
14     v_output NUMBER;
15 BEGIN
16     CalculateSquare(v_input, v_output);
17     DBMS_OUTPUT.PUT_LINE('Square of ' || v_input || ' is ' || v_output);
18 END;
```

15.1 模块化与分层

利用子程序进行**模块化**程序设计的基本原则是：尽可能地创建较小的、灵活的、可重用的代码段以方便程序的管理和维护。灵活性是通过使用带有参数的子程序而获得的，而正是这种灵活性又通过使用不同的输入值使得相同的程序代码能够重用。

模块化**现存**的**程序代码**的步骤：

1. 定位和标识重复的程序代码序列。
2. 将这些重复的程序代码移到一个 PL/SQL 子程序中。
3. 将原来的重复程序代码以新的 PL/SQL 子程序调用代替。

模块化的优势：

1. **语法分析重用**：优化器能够有效地重复利用经过解析的 SQL 语句，更好地利用服务器资源。
2. 可维护性：将业务逻辑分解为独立的模块，使得对系统进行更新、修复或新增功能变得更加简单。
3. 可读性：代码结构更加清晰。
4. 性能优化：通过将复杂的业务逻辑分解为独立的模块，优化器可以更精确地评估每个模块的性能，并采取相应的优化措施。
5. 减少代码冗余：将通用功能抽象为可重用的模块，可以避免在不同部分中重复编写相似的代码。

分层设计思路：

1. 数据访问层：使用 SQL 语句访问数据库子程序。
 - (a) 使用 SQL 语句访问数据库。
 - (b) 封装数据库细节。
 - (c) 提供数据访问接口。
2. 业务逻辑层：实现业务处理规则的子程序，这些子程序可以调用也可以不调用数据访问层的子程序。
 - (a) 实现业务处理规则。

(b) 调用数据访问层。

(c) 与用户界面交互。

与模块化有关的组件：

- 匿名程序块。
- 子程序（过程和函数）。
- 软件包。
- 数据库触发器。

15.2 子程序

子程序是基于标准的 PL/SQL 结构的，其实就是一个命名的 PL/SQL 程序块，它可以接收参数和在调用环境中被调用。子程序包括一个声明段、一个执行段和一个可选的异常处理段。子程序可以被编译或存储在数据库中以提高模块化、可扩展性和重用性，以及方便维护。

一个 PL/SQL 子程序就是一个命名的、可以使用一组参数调用的 PL/SQL 程序块。可以在一个 PL/SQL 程序块中也可以在另一个子程序中声明和定义子程序。

Listing 15.2: 子程序语法模版

```
1  --子程序说明部分
2  <header>
3
4  IS | AS
5  --子程序体
6      Declaration 部分
7  BEGIN
8      可执行部分
9  EXCEPTION (optional)
10     异常部分
11  END;
```

PL/SQL 有两种类型的子程序，即过程和函数。**通常，使用过程来执行一个操作，而使用一个函数计算并返回一个值。**过程和函数具有相同的结构，它们唯一的区别是函数有一个额外

的项——**RETURN** 子句或 **RETURN** 语句。

Listing 15.3: 过程示例

```
1 CREATE OR REPLACE PROCEDURE my_procedure(p_parameter IN VARCHAR2)
2   IS
3 BEGIN
4   -- Procedure code here
5   DBMS_OUTPUT.PUT_LINE('Parameter received: ' || p_parameter);
6 END my_procedure;
```

Listing 15.4: 函数示例

```
1 CREATE OR REPLACE FUNCTION calculate_area(radius NUMBER) RETURN NUMBER
2   IS
3   v_area NUMBER;
4 BEGIN
5   v_area := 3.14 * radius * radius;
6   RETURN v_area;
7 END calculate_area;
```

匿名程序块与子程序之间的差别：

1. 子程序只编译一次；匿名程序块在执行时编译。（如果想重用一個匿名塊，就必須重新運行創建這個匿名塊的腳本，這會造成重新編譯和執行）
2. 子程序可以存儲在數據庫中；匿名程序塊不可以。
3. 子程序可以被其他程序調用；匿名程序塊不可以。
4. 子程序的函數必須有返回值；匿名程序塊不能帶返回值。
5. 子程序可以有參數；匿名程序塊不能有參數。

15.3 过程

15.3.1 过程语法

过程是指可以接收参数并命名的 PL/SQL 程序块，过程是子程序的一种类型，通常使用它来执行一个动作。过程包括过程头、声明段、执行段和可选的异常处理段。在另外的 PL/SQL 程序块的执行段中通过使用过程名调用这个过程。

在 PL/SQL 中，可以使用 SQL 的 **CREATE[OR REPLACE]PROCEDURE** 语句创建存储在数据库中的独立过程。一个过程与一个微型的程序类似——执行一个特定的操作。在创建过程语句中，要说明过程名、过程的参数、过程的本地变量和包括过程代码和处理代码的 **BEGIN-END** 所包括的程序块。

Listing 15.5: 创建过程的语法

```
1 CREATE [OR REPLACE] PROCEDURE 过程名
2   [(参数1 [参数模式] 数据类型1, 参数2 [参数模式] 数据类型2, ...)]
3 IS|AS
4   [本地变量的声明; ...]
5 BEGIN
6   -- 执行的操作;
7 END [过程名];
```

参数可以被看作本地变量。在一个 PL/SQL 存储过程的定义中，参数和本地变量在合适的作用域内都是可以引用的。

Listing 15.6: 创建过程示例代码

```
1 CREATE OR REPLACE PROCEDURE ExampleProcedure (
2   p_input_param IN VARCHAR2,
3   p_output_param OUT NUMBER
4 ) IS
5   -- 这是一个本地变量，只在当前存储过程中可见
6   v_local_variable NUMBER := 0;
7 BEGIN
8   -- 使用输入参数
9   DBMS_OUTPUT.PUT_LINE('Input parameter: ' || p_input_param);
```

```
10  -- 修改本地变量的值
11  v_local_variable := 42;
12  -- 使用输出参数
13  p_output_param := v_local_variable;
14  -- 在这里可以引用输入参数、输出参数和本地变量
15  END ExampleProcedure;
```

15.3.2 过程参数数据类型

可以使用如下方法制定参数的数据类型：

1. **显式数据类型**：如 p_param1 VARCHAR2(50)。
2. **%TYPE 定义**：如 p_param2 employees.salary%TYPE。
3. **%ROWTYPE 定义**：如 p_param2 employees%ROWTYPE。

15.3.3 过程参数的参数模式

参数被用来在调用环境和过程之间进行数据的传递。参数是在子程序（过程）的头中声明的，即在过程名之后和本地变量声明段之前，调用环境和过程之间进行参数传递的操作。

1. **IN 参数**：从调用环境传递一个常数值给过程，可赋予默认值。（默认）
2. **OUT 参数**：从过程传递一个值给调用环境，实参必须为变量，不可赋予默认值。
3. **IN OUT 参数**：从调用环境传递一个值给过程，并且使用相同的参数名从过程返回给调用环境一个可能不同的值，实参必须为变量，不可赋予默认值。

可以把参数看成本地变量的一种特殊形式：当子程序被调用时，参数的输入值由调用环境初始化，并且当子程序将控制返回给调用者时，参数的输出值被返回给调用环境。

Listing 15.7: IN 参数模式实例

```
1  --为某一个员工提升工资的过程
2  CREATE OR REPLACE PROCEDURE raise_salary(p_empno IN emp_pl.empno%TYPE,p_rate
      IN NUMBER)
3  IS
4  BEGIN
```

```
5  UPDATE emp_pl
6  SET sal = sal * (1 + p_rate * 0.01)
7  WHERE empno = p_empno;
8  END raise_salary;
9  /
```

Listing 15.8: OUT 参数模式实例

```
1  --输出指定员工信息
2  CREATE OR REPLACE PROCEDURE get_employee
3  (p_empno IN emp_pl.empno%TYPE,
4   p_name OUT emp.ename%TYPE,
5   p_salary OUT emp.sal%TYPE,
6   p_job OUT emp.job%TYPE)
7  IS
8  BEGIN
9   SELECT ename, sal, job
10  INTO p_name, p_salary, p_job
11  FROM emp
12  WHERE empno = p_empno;
13  END;
14  /
```

Listing 15.9: IN OUT 参数模式实例

```
1  --将输入的电话号码转换为标准的容易阅读格式
2  CREATE OR REPLACE PROCEDURE standard_phone
3  (p_phone_no IN OUT VARCHAR2)
4  IS
5  BEGIN
6   p_phone_no := '(' || SUBSTR(p_phone_no,1,3) || ')' || SUBSTR(p_phone_no,4,3)
7   || '-' || SUBSTR(p_phone_no,7);
8  END standard_phone;
```

15.3.4 绑定变量获得过程的 OUT 输出

步骤：

1. 声明过程变量：**VARIABLE myvar NUMBER**。
2. (可选：如果是 IN OUT 参数需要为这个数字变量赋初值：**EXEC :myvar := 10**。)
3. 以绑定变量为参数执行过程：**EXEC my_procedure(:myvar)**。
4. 使用 PRINT 命令打印绑定变量：**PRINT myvar**。

15.3.5 在过程中处理异常

在过程中处理异常的方式与在一般程序块中处理异常的方法完全相同：当在一个调用过程中抛出一个异常时，程序的控制立刻跳转到这个程序块的异常段。如果这个异常段提供了一个处理所抛出异常的异常处理程序，那么这个异常被认定处理了。

PL/SQL 将执行如下代码流程：

1. 这个异常被抛出。
2. 控制转移到匹配的异常处理程序。
3. 执行异常处理程序的代码块。
4. 异常处理程序执行完成后，程序块执行结束。

如果一个 DML 操作是在异常之前在异常所在的过程中执行的，那么这个 DML 操作被回滚。

Listing 15.10: 过程异常处理示例代码

```
1 CREATE OR REPLACE PROCEDURE add_depte
2   (p_name IN dept_pl.dname%TYPE DEFAULT '服务',
3    p_loc IN dept_pl.loc%TYPE DEFAULT '狼山镇')
4 IS
5 BEGIN
6   INSERT INTO dept_pl
7   VALUES (deptid_sequence.NEXTVAL, p_name, p_loc);
8   DBMS_OUTPUT.PUT_LINE('添加部门: '||p_name);
9 EXCEPTION
```



```
10 WHEN OTHERS THEN
11     DBMS_OUTPUT.PUT_LINE('错误：添加部门：'||p_name);
12 END add_depte;
13 /
```

15.3.6 在过程中抛出异常

当被调用过程的异常段没有提供所抛出异常的处理程序时，PL/SQL 程序代码流程如下：

1. 抛出异常。
2. 程序块结束，因为不存在异常处理程序，在该过程中所执行的任何 DML 操作全部被回滚。
3. 异常传播到调用过程的异常段，即控制返回到调用块的异常段。

如果一个异常没有被处理，那么在调用过程和被调用过程中的所有 DML 语句连同对任何绑定变量的更改一起全部被回滚。但是在调用抛出异常过程之前所执行的 DML 语句并不受影响。

15.4 实参的传递方式

PL/SQL 提供了三种不同的参数传递表示法，分别按位置（position）、按名字（Named）和组合（Combination）表示法。

按位置 以所声明的形参相同的顺序列出实参。

按名字 以任意顺序列出实参和与之相关的对应形参，但要是使用关联操作符将每一个实参与对应的形参用名关联起来。

组合 部分实参按位置，部分实参按名称。

在开发一个子程序时可以将几乎所有的可能情况考虑进去，这样可能需要很多形参。但是在使用时用户并不需要了解全部参数，一个用户只需要理解他所需要的参数即可。

Listing 15.11: 按位置传递示例

```
1 CREATE OR REPLACE PROCEDURE example_proc
2   (p_name IN VARCHAR2, p_age IN NUMBER)
```

```
3 IS
4 BEGIN
5     -- procedure body
6 END example_proc;
7
8 -- 调用过程
9 EXEC example_proc('John', 30);
```

Listing 15.12: 按参数名传递示例

```
1 CREATE OR REPLACE PROCEDURE example_proc
2     (p_name IN VARCHAR2, p_age IN NUMBER)
3 IS
4 BEGIN
5     -- procedure body
6 END example_proc;
7
8 -- 调用过程，通过参数名传递
9 EXEC example_proc(p_age => 30, p_name => 'John');
```

Listing 15.13: 组合传递示例

```
1 CREATE OR REPLACE PROCEDURE example_proc
2     (p_name IN VARCHAR2, p_age IN NUMBER) IS
3 BEGIN
4     -- procedure body
5 END example_proc;
6
7 -- 调用过程，混合使用按位置和按名字传递
8 EXEC example_proc('John', p_age => 30);
```

第十六章 游标

在 PL/SQL 中，游标（CURSOR）是一种数据库查询的控制结构，它用于存储从 SQL 语句如 SELECT 查询返回的结果集。游标使得开发者能够在 PL/SQL 程序中逐行处理查询结果。

游标用途：

1. 逐行处理数据。
2. 复杂逻辑的实现。
3. 锁定行。
4. 资源优化。

16.1 游标类型

1. 显式游标：开发者需要显式声明、打开、获取数据、关闭。这给了开发者更多的控制，但也需要更多的代码。
2. 隐式游标：Oracle 为每个 SQL DML 语句（如 INSERT, UPDATE, DELETE, SELECT INTO）提供的自动游标。它们不需要显式声明和管理，但控制较少。

16.2 游标的操作过程

1. 声明：CURSOR my_cursor IS SELECT column1, column2 FROM my_table WHERE condition;
2. 打开：OPEN my_cursor;
3. 提取：FETCH my_cursor INTO variable1, variable2;
4. 关闭：CLOSE my_cursor;

16.3 游标逐行处理数据示例

Listing 16.1: 游标逐行处理数据示例 1

```
1  -- 创建一个简单的表
2  CREATE TABLE employee (
3      emp_id NUMBER PRIMARY KEY,
4      emp_name VARCHAR2(50),
5      emp_salary NUMBER
6  );
7
8  -- 插入一些示例数据
9  INSERT INTO employee VALUES (1, 'Alice', 50000);
10 INSERT INTO employee VALUES (2, 'Bob', 60000);
11 INSERT INTO employee VALUES (3, 'Charlie', 70000);
12
13 -- 创建一个存储过程，使用游标逐行处理数据
14 CREATE OR REPLACE PROCEDURE display_employee_data
15 IS
16     -- 定义游标
17     CURSOR emp_cursor IS
18         SELECT emp_id, emp_name, emp_salary FROM employee;
19
20     -- 定义变量来存储检索到的列值
21     v_emp_id employee.emp_id%TYPE;
22     v_emp_name employee.emp_name%TYPE;
23     v_emp_salary employee.emp_salary%TYPE;
24 BEGIN
25     -- 打开游标
26     OPEN emp_cursor;
27     -- 逐行处理数据
28     LOOP
```

```
29  -- 从游标中检索数据
30  FETCH emp_cursor INTO v_emp_id, v_emp_name, v_emp_salary;
31  -- 判断是否还有数据
32  EXIT WHEN emp_cursor%NOTFOUND;
33  -- 处理当前行数据，这里简单地显示在输出
34  DBMS_OUTPUT.PUT_LINE('Employee ID: ' || v_emp_id || ', Name: ' ||
    v_emp_name || ', Salary: ' || v_emp_salary);
35  END LOOP;
36
37  -- 关闭游标
38  CLOSE emp_cursor;
39  END;
40  /
41
42  -- 执行存储过程以显示数据
43  SET SERVEROUTPUT ON;
44  EXEC display_employee_data;
```

使用 FOR 循环简化：FOR 循环是一种简化的循环结构，它允许在循环声明中直接指定一个游标，并在循环体内自动处理每一行，无需显式地使用 OPEN、FETCH 和 CLOSE 语句。

Listing 16.2: 游标逐行处理数据示例 2

```
1  -- 创建一个简单的表
2  CREATE TABLE employee (
3      emp_id NUMBER PRIMARY KEY,
4      emp_name VARCHAR2(50),
5      emp_salary NUMBER
6  );
7
8  -- 插入一些示例数据
9  INSERT INTO employee VALUES (1, 'Alice', 50000);
10 INSERT INTO employee VALUES (2, 'Bob', 60000);
```

```
11 INSERT INTO employee VALUES (3, 'Charlie', 70000);
12
13 -- 创建一个存储过程，使用FOR循环处理数据
14 CREATE OR REPLACE PROCEDURE display_employee_data
15 IS
16 BEGIN
17     -- 使用FOR循环声明游标和直接处理每一行
18     FOR emp_rec IN (SELECT emp_id, emp_name, emp_salary FROM employee) LOOP
19         -- 在循环体内直接访问每一行的字段
20         DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_rec.emp_id || ', Name: ' ||
            emp_rec.emp_name || ', Salary: ' || emp_rec.emp_salary);
21     END LOOP;
22 END;
23 /
24
25 -- 执行存储过程以显示数据
26 SET SERVEROUTPUT ON;
27 EXEC display_employee_data;
```