# Oracle 复习

## Week 11

使用PL/SQL时，不会自动提交，需要补上 **commit** 或手动提交。手动提交可以利用 **rollback** 实现回滚。

### PL/SQL 程序结构包括以下三部分

1. 声明部分（Declarative Section）：定义变量、常量、游标等。
2. 执行部分（Executable Section）：包含执行的PL/SQL语句。
3. 异常处理部分（Exception Handling Section）：处理程序中的异常或错误。

```
DECLARE
    -- 声明部分：在这里定义变量
    v_empno emp.empno%TYPE;
    v_ename emp.ename%TYPE;
    v_sal emp.sal%TYPE;
    v_new_sal emp.sal%TYPE;
BEGIN
    -- 执行部分：包含PL/SQL执行的逻辑
    -- 从EMP表中选择一名员工的信息
    SELECT empno, ename, sal INTO v_empno, v_ename, v_sal
    FROM emp
    WHERE ename = 'SMITH';

    -- 计算新的工资，假设增加10%的工资
    v_new_sal := v_sal * 1.10;

    -- 更新员工的工资
    UPDATE emp
    SET sal = v_new_sal
    WHERE empno = v_empno;

    -- 提交事务
    COMMIT;

    -- 输出更新后的员工信息
    DBMS_OUTPUT.PUT_LINE('员工编号: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('员工姓名: ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('原工资: ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('新工资: ' || v_new_sal);
EXCEPTION
    -- 异常处理部分：处理在执行部分中发生的错误
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('员工未找到');
    WHEN OTHERS THEN
        -- 捕获所有其他异常
        DBMS_OUTPUT.PUT_LINE('发生错误: ' || SQLERRM);
END;
/
```

设置环境变量 **serveroutput** 启用服务器输出。这一行告诉SQLPlus将服务器端的输出显示在客户端屏幕上。

```
set serveroutput on
```

## 基本语法

> IF 语句

```
IF condition Then
    statements;
ELSIF condition THEN
    statements;
ELSE
    statements;
END IF;
```

> LOOP 语句

```
LOOP
    statement1;
    ...
    EXIT [WHEN condition];
END LOOP
```

> While-Loop 语句

```
WHILE condition LOOP
    statement 1;
    statement 2;
    ...
END LOOP
```

> > 例子

```
DECLARE
    v_counter NUMBER := 1;  -- 声明并初始化计数器变量
BEGIN
    -- WHILE循环
    WHILE v_counter <= 5 LOOP
        -- 输出当前计数器值
        DBMS_OUTPUT.PUT_LINE('当前计数器值: ' || v_counter);
        -- 增加计数器值
        v_counter := v_counter + 1;
    END LOOP;
END;
/
```

> For-loop 语句

```
FOR counter IN [REVERSE]
        lower_bound...upper bound LOOP
    statement1;
    statement2;
    ...
END LOOP;
```

> 例子

```
BEGIN
    -- 正向FOR循环
    FOR i IN 1..5 LOOP
        DBMS_OUTPUT.PUT_LINE('正向循环: i = ' || i);
    END LOOP;
END;
/
```

**在PL/SQL中使用SQL**

1. 可直接使用 **DML** 语句
2. select ... into ...
3. 不能使用 **DDL** 语句

以下是DML（数据操作语言）和DDL（数据定义语言）语句的列表：

## DML 语句

| 语句 | 描述 |
| --- | --- |
| `SELECT` | 从数据库表中检索数据 |
| `INSERT` | 向数据库表中插入新的数据 |
| `UPDATE` | 更新数据库表中的现有数据 |
| `DELETE` | 从数据库表中删除数据 |
| `MERGE` | 合并两张表的数据，进行插入或更新操作 |
| `CALL` | 调用存储过程或函数 |
| `LOCK TABLE` | 锁定数据库表以控制并发访问 |
| `EXPLAIN PLAN` | 显示SQL语句的执行计划 |

## DDL 语句

| 语句 | 描述 |
| --- | --- |
| `CREATE` | 创建数据库对象（如表、视图、索引、序列等） |
| `ALTER` | 修改现有的数据库对象 |
| `DROP` | 删除数据库对象 |

| 语句 | 描述 |
| --- | --- |
| `TRUNCATE` | 清空表中的所有数据，但保留表结构 |
| `COMMENT` | 添加或修改数据库对象的注释 |
| `RENAME` | 重命名数据库对象 |
| `GRANT` | 授予用户权限 |
| `REVOKE` | 撤销用户权限 |
| `ANALYZE` | 收集有关数据库对象的统计信息 |

## DML 示例

```
-- SELECT 示例
SELECT * FROM emp WHERE empno = 7369;

-- INSERT 示例
INSERT INTO emp (empno, ename, job, sal) VALUES (1234, 'JOHN', 'MANAGER', 5000);

-- UPDATE 示例
UPDATE emp SET sal = sal * 1.1 WHERE empno = 1234;

-- DELETE 示例
DELETE FROM emp WHERE empno = 1234;
```

## DDL 示例

```
-- CREATE 示例
CREATE TABLE test_table (id NUMBER, name VARCHAR2(50));

-- ALTER 示例
ALTER TABLE test_table ADD (address VARCHAR2(100));

-- DROP 示例
DROP TABLE test_table;

-- TRUNCATE 示例
TRUNCATE TABLE emp;

-- COMMENT 示例
COMMENT ON TABLE emp IS 'Employee table';

-- RENAME 示例
RENAME test_table TO new_test_table;
```

# PL/SQL 程序的几种形式

## 匿名块 与SQL语句类似，在客户端中直接做adhoc查询

- 没有名称，不存储在数据库中。
- 常用于临时操作或测试。
- 可以在SQL*Plus、SQL Developer等工具中直接执行。

```
DECLARE

BEGIN

END;
/
```

## 存储过程 作为数据库对象保存在数据库中的代码，通过参数进行输入输出数据交换

- 存储在数据库中，可以被多个用户和应用程序共享。
- 可以接受输入参数和输出参数，但不返回值。
- 常用于封装业务逻辑、数据验证和批量操作。

```
CREATE OR REPLACE PROCEDURE update_salary (
    p_empno IN emp.empno%TYPE,
    p_percentage IN NUMBER
) AS
```

调用存储过程

```
BEGIN
    update_salary(7369, 10); -- 将员工编号为7369的工资增加10%
END;
/
```

## 存储函数 作为数据库对象保存在数据库中的代码，需要有返回值作为函数的输出

- 存储在数据库中，可以被多个用户和应用程序共享。
- 需要返回一个值。
- 常用于计算和数据转换。

```
CREATE OR REPLACE FUNCTION get_employee_salary (
    p_empno IN emp.empno%TYPE
) RETURN NUMBER
AS
```

调用存储函数

```
DECLARE
    v_salary NUMBER;
BEGIN
    v_salary := get_employee_salary(7369); -- 获取员工编号为7369的工资
    DBMS_OUTPUT.PUT_LINE('员工工资: ' || v_salary);
END;
/
```

### 使用存储过程与存储函数的好处

- 集中修改，方便维护
- 速度更快（省去网络传输与编译时间）
- 可以受到数据库权限机制保护

### 怎样调用存储过程和存储函数

- 在sqlplus中使用存储过程，打入“exec <过程名>”
- 在程序中使用存储过程，直接使用<过程名>即可
- 使用存储函数与使用标准SQL函数没有区别

## 存储过程示例

这个存储过程将给定员工编号的工资增加一个指定的百分比。

```
CREATE OR REPLACE PROCEDURE increase_salary (
    p_empno IN emp.empno%TYPE,
    p_percentage IN NUMBER
) AS
BEGIN
    UPDATE emp
    SET sal = sal * (1 + p_percentage / 100)
    WHERE empno = p_empno;

    COMMIT;
END;
/
```

### 调用存储过程

#### 在SQL*Plus中使用存储过程

```
-- 启用服务器输出以便查看结果
SET SERVEROUTPUT ON;

-- 使用 EXEC 调用存储过程
EXEC increase_salary(7369, 10); -- 将员工编号为7369的工资增加10%
```

#### 在PL/SQL块中调用存储过程

```
BEGIN
    increase_salary(7369, 10); -- 将员工编号为7369的工资增加10%
END;
/
```

# 存储函数示例

## 创建存储函数

这个存储函数返回给定员工编号的工资。

```
CREATE OR REPLACE FUNCTION get_salary (
    p_empno IN emp.empno%TYPE
) RETURN NUMBER
AS
    v_sal emp.sal%TYPE;
BEGIN
    SELECT sal INTO v_sal
    FROM emp
    WHERE empno = p_empno;

    RETURN v_sal;
END;
/
```

## 调用存储函数

### 在SQL*Plus中调用存储函数

```
-- 启用服务器输出以便查看结果
SET SERVEROUTPUT ON;

DECLARE
    v_salary NUMBER;
BEGIN
    v_salary := get_salary(7369); -- 获取员工编号为7369的工资
    DBMS_OUTPUT.PUT_LINE('员工工资: ' || v_salary);
END;
/
```

### 在SQL查询中使用存储函数

```
SELECT get_salary(7369) AS salary FROM dual;
```

# wm_concat 函数

- Oracle的内测函数
- 另一种方式的列转行
- Mysql中类似的函数是group_concat
- 输出是clob类型

### 示例

```
-- 使用WM_CONCAT函数按部门将员工名字连接成一个字符串
SELECT deptno, WM_CONCAT(ename) AS employees
FROM emp
GROUP BY deptno;
```

## listagg 函数

```
-- 使用LISTAGG函数按部门将员工名字连接成一个字符串
SELECT deptno, LISTAGG(ename, ',') WITHIN GROUP (ORDER BY ename) AS employees
FROM emp
GROUP BY deptno;
```

# Week 11

## %type 变量类型

在 PL/SQL 中，%TYPE 是一种变量声明方式，允许我们基于数据库表中的列定义变量。这样声明的变量会自动采用对应列的属性，包括数据类型和长度。这种方式有助于保持数据类型的一致性和提高代码的可维护性。

### 语法

```
identifier Table.column_name%TYPE;
```

- **identifier**：变量名称。
- **Table.column_name**：数据库表和列的名称。

### 示例解释

```
DECLARE
    v_name employees.last_name%TYPE;    -- 声明一个变量，其类型与 employees 表中 last_name 列的类型相同
    v_balance NUMBER(7,2);              -- 声明一个变量，类型为 NUMBER，精度为 7，标度为 2
    v_min_balance v_balance%TYPE := 10;-- 声明一个变量，其类型与 v_balance 变量的类型相同，并初始化为 10
BEGIN

END;
/
```

### 优点

- **类型一致性**：使用 `%TYPE` 可以确保变量与表列的数据类型一致，避免数据类型不匹配的问题。
- **代码可维护性**：当表结构发生变化时（例如列的数据类型变化），不需要修改 PL/SQL 代码中变量的类型，只需修改表结构即可。

## %rowtype 变量类型

在 PL/SQL 中，`%ROWTYPE` 是一种变量声明方式，用于声明一个记录（record）类型的变量，该变量的结构与指定表或游标的行结构相同。`%ROWTYPE` 使我们能够处理整行数据，而不仅仅是单个列的数据。

### 语法

```
identifier Table%ROWTYPE;
identifier Cursor%ROWTYPE;
```

- **identifier**：变量名称。
- **Table**：数据库表的名称。
- **Cursor**：PL/SQL 游标的名称。

### 示例解释

```
DECLARE
    -- 声明一个记录变量 v_emp_row，其结构与 EMP 表的行结构相同
    v_emp_row emp%ROWTYPE;
BEGIN
    -- 查询员工信息并赋值给记录变量 v_emp_row
    SELECT * INTO v_emp_row
    FROM emp
    WHERE empno = 7369;

    -- 输出记录变量 v_emp_row 中的各列值
    DBMS_OUTPUT.PUT_LINE('员工编号: ' || v_emp_row.empno);
    DBMS_OUTPUT.PUT_LINE('员工姓名: ' || v_emp_row.ename);
    DBMS_OUTPUT.PUT_LINE('员工职位: ' || v_emp_row.job);
    DBMS_OUTPUT.PUT_LINE('员工经理: ' || v_emp_row.mgr);
    DBMS_OUTPUT.PUT_LINE('入职日期: ' || v_emp_row.hiredate);
    DBMS_OUTPUT.PUT_LINE('员工工资: ' || v_emp_row.sal);
    DBMS_OUTPUT.PUT_LINE('员工佣金: ' || v_emp_row.comm);
    DBMS_OUTPUT.PUT_LINE('部门编号: ' || v_emp_row.deptno);
END;
/
```

1. `v_emp_row emp%ROWTYPE`：
    - 声明一个记录变量 `v_emp_row`，它的结构与 `emp` 表的行结构相同。
    - 这个变量将包含 `emp` 表的所有列。
2. `SELECT * INTO v_emp_row FROM emp WHERE empno = 7369`：
    - 从 `emp` 表中选择一行（员工编号为 7369 的员工）并将其赋值给记录变量 `v_emp_row`。
    - 这样 `v_emp_row` 变量的每个字段将对应 `emp` 表的一列。
3. **输出记录变量 `v_emp_row` 中的各列值**：
    - 使用 `DBMS_OUTPUT.PUT_LINE` 输出记录变量 `v_emp_row` 中每个字段的值。

## 使用优点

- **方便处理整行数据**：使用 `%ROWTYPE` 可以简化对整行数据的操作，而不需要单独声明每一列的变量。
- **类型一致性**：确保记录变量的各个字段与表或游标的列类型一致。
- **提高代码可读性**：使代码更简洁易读，特别是在需要处理多列数据时。

## 使用场景

- **处理表或游标的一整行数据**：在需要处理表或游标的一整行数据时，使用 `%ROWTYPE` 变量类型可以避免单独声明每一列的变量。
- **批量处理和游标循环**：在批量处理和游标循环中，使用 `%ROWTYPE` 可以简化代码结构。

## 示例：在游标中使用 %ROWTYPE

```
DECLARE
    -- 声明一个游标，选择 EMP 表中的所有记录
    CURSOR emp_cursor IS
        SELECT * FROM emp;

    -- 声明一个记录变量 v_emp_row，其结构与游标 emp_cursor 的行结构相同
    v_emp_row emp_cursor%ROWTYPE;
BEGIN
    -- 打开游标
    OPEN emp_cursor;

    -- 循环遍历游标中的每一行
    LOOP
        -- 提取游标中的一行数据到记录变量 v_emp_row
        FETCH emp_cursor INTO v_emp_row;

        -- 当没有更多行时退出循环
        EXIT WHEN emp_cursor%NOTFOUND;

        -- 输出记录变量 v_emp_row 中的各列值
        DBMS_OUTPUT.PUT_LINE('员工编号: ' || v_emp_row.empno);
        DBMS_OUTPUT.PUT_LINE('员工姓名: ' || v_emp_row.ename);
    END LOOP;

    -- 关闭游标
    CLOSE emp_cursor;
END;
/
```

在这个示例中，使用 `%ROWTYPE` 简化了游标循环中的数据处理，使代码更加简洁和易于维护。

# Record 类型

## 理解 PL/SQL 中的 Record 类型

在 PL/SQL 中，`Record` 类型是一种复合数据类型，它可以包含多个字段，每个字段可以具有不同的数据类型。`Record` 类型类似于编程语言中的结构体（struct）或对象（object），用于存储相关联的数据。使用 `Record` 类型可以更方便地组织和管理复杂的数据结构。

## 定义和使用 Record 类型

使用 `TYPE` 关键字可以在 PL/SQL 块、包或数据库级别声明自定义的 Record 类型。

### 示例

```
DECLARE
    -- 声明一个 Record 类型
    TYPE employee_record IS RECORD (
        empno    emp.empno%TYPE,
        ename    emp.ename%TYPE,
        job      emp.job%TYPE,
        sal      emp.sal%TYPE
    );

    -- 声明一个变量，该变量的类型为 employee_record
    v_employee employee_record;
BEGIN
    -- 使用 SELECT 语句将数据赋值给 Record 变量
    SELECT empno, ename, job, sal
    INTO v_employee
    FROM emp
    WHERE empno = 7369;

    -- 输出 Record 变量的各个字段
    DBMS_OUTPUT.PUT_LINE('员工编号: ' || v_employee.empno);
    DBMS_OUTPUT.PUT_LINE('员工姓名: ' || v_employee.ename);
    DBMS_OUTPUT.PUT_LINE('员工职位: ' || v_employee.job);
    DBMS_OUTPUT.PUT_LINE('员工工资: ' || v_employee.sal);
END;
/
```

### 使用 Record 类型的优点

1. **方便管理相关数据**：将相关数据封装在一个记录中，方便管理和传递。
2. **提高代码可读性**：代码更简洁，易于理解和维护。
3. **类型一致性**：使用 `%ROWTYPE` 保证记录字段类型与表或游标列类型一致，减少数据类型不匹配的问题。

### Record 类型的使用场景

- **复杂数据结构**：在需要管理复杂的数据结构时，使用 Record 类型可以将相关数据封装在一起。
- **批量处理和游标循环**：在批量处理和游标循环中，使用 Record 类型可以简化数据处理。

- **存储过程和函数**：在存储过程和函数中，可以使用 Record 类型作为参数或返回值，以便传递复杂的数据结构。

## 游标详细用法

在 PL/SQL 中，游标（Cursor）是一种用于处理查询结果集的机制。游标允许我们逐行处理查询返回的多行数据。PL/SQL 支持两种类型的游标：显式游标（Explicit Cursor）和隐式游标（Implicit Cursor）。

### 游标使用的基本步骤

1. **DECLARE（声明游标）**：创建一个命名的 SQL 区域。
2. **OPEN（打开游标）**：标识活动集（active set）。
3. **FETCH（提取数据）**：将当前行加载到变量中。
4. **CLOSE（关闭游标）**：释放活动集。

### 示例：使用显式游标

1. **声明游标**：

```
CURSOR emp_cursor IS
    SELECT empno, ename, job, sal
    FROM emp;
```

   - 这段代码声明了一个名为 `emp_cursor` 的游标，该游标选择 `EMP` 表中的所有记录。

2. **声明记录变量**：

```
v_emp_row emp_cursor%ROWTYPE;
```

   - 声明一个记录变量 `v_emp_row`，其结构与游标 `emp_cursor` 的行结构相同。

3. **打开游标**：

```
OPEN emp_cursor;
```

   - 打开游标 `emp_cursor`，准备提取数据。

4. **循环提取数据**：

```
LOOP
    FETCH emp_cursor INTO v_emp_row;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('员工编号: ' || v_emp_row.empno);
    DBMS_OUTPUT.PUT_LINE('员工姓名: ' || v_emp_row.ename);
    DBMS_OUTPUT.PUT_LINE('员工职位: ' || v_emp_row.job);
    DBMS_OUTPUT.PUT_LINE('员工工资: ' || v_emp_row.sal);
END LOOP;
```

   - 在循环中，从游标 `emp_cursor` 提取一行数据到记录变量 `v_emp_row`。
   - 当提取到的行数为 0 时，退出循环。
   - 输出记录变量 `v_emp_row` 中的各列值。

5. **关闭游标**：

```
CLOSE emp_cursor;
```

   - 关闭游标 `emp_cursor`，释放相关资源。

### 小结

使用显式游标时，需要手动管理游标的生命周期，包括声明、打开、提取和关闭。通过显式游标，可以逐行处理查询结果集，适用于需要逐行处理数据的场景。

### 隐式游标

隐式游标由 Oracle 数据库自动管理，通常在执行 DML 语句（如 `INSERT`、`UPDATE`、`DELETE` 和 `SELECT INTO`）时使用。

### 示例：使用隐式游标

```
DECLARE
    v_empno emp.empno%TYPE;
    v_ename emp.ename%TYPE;
    v_job emp.job%TYPE;
    v_sal emp.sal%TYPE;
BEGIN
    -- 使用隐式游标选择一行数据
    SELECT empno, ename, job, sal INTO v_empno, v_ename, v_job, v_sal
    FROM emp
    WHERE empno = 7369;

    -- 输出选择的数据
    DBMS_OUTPUT.PUT_LINE('员工编号: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('员工姓名: ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('员工职位: ' || v_job);
    DBMS_OUTPUT.PUT_LINE('员工工资: ' || v_sal);
END;
/
```

隐式游标简单易用，适用于处理单行查询结果的情况。Oracle 会自动管理隐式游标的生命周期。

## 查看编译错误

```
show errors
```

## 例外处理

- **系统预定义异常**：Oracle预先定义的标准异常，直接使用异常名称进行处理。

系统预定义异常是Oracle预先定义的标准异常。这些异常与常见的错误条件相关联，并且有具体的名称。可以直接在异常处理部分使用这些异常名称进行处理。

### 常见的系统预定义异常

| 异常名称 | 错误代码 | 描述 |
| --- | --- | --- |
| `NO_DATA_FOUND` | ORA-01403 | 当`SELECT INTO`语句未返回任何行时触发。 |
| `TOO_MANY_ROWS` | ORA-01422 | 当`SELECT INTO`语句返回多于一行时触发。 |
| `ZERO_DIVIDE` | ORA-01476 | 当尝试除以零时触发。 |
| `INVALID_CURSOR` | ORA-01001 | 当尝试操作无效游标时触发。 |
| `DUP_VAL_ON_INDEX` | ORA-00001 | 当尝试插入重复值违反唯一性约束时触发。 |

- **非预定义的系统异常**：Oracle中存在但没有预定义名称的异常，使用错误代码和`PRAGMA EXCEPTION_INIT`进行处理。

```
DECLARE
    -- 声明自定义异常
    e_custom_exception EXCEPTION;
    -- 将错误代码与自定义异常关联
    PRAGMA EXCEPTION_INIT(e_custom_exception, -1403);
BEGIN
    -- 尝试执行一个返回无数据的查询
    DECLARE
        v_name emp.ename%TYPE;
    BEGIN
        SELECT ename INTO v_name FROM emp WHERE empno = 9999;
    EXCEPTION
        WHEN e_custom_exception THEN
            DBMS_OUTPUT.PUT_LINE('错误：没有找到数据');
    END;
END;
/
```

- **用户自定义异常**：程序员定义的异常，用于处理特定业务逻辑中的错误，通过声明、引发和处理自定义异常进行管理。
    1. **声明异常**：在声明部分定义异常。
    2. **引发异常**：在执行部分使用`RAISE`语句引发异常。
    3. **处理异常**：在异常处理部分捕获和处理异常。

```
DECLARE
    -- 声明用户自定义异常
    e_salary_too_high EXCEPTION;
    -- 定义工资上限
    v_max_salary CONSTANT NUMBER := 10000;
BEGIN
    -- 假设有一个员工的工资
    DECLARE
        v_salary emp.sal%TYPE := 15000;
    BEGIN
        -- 检查工资是否超过上限
        IF v_salary > v_max_salary THEN
```

```
            -- 引发自定义异常
            RAISE e_salary_too_high;
        END IF;
    EXCEPTION
        WHEN e_salary_too_high THEN
            DBMS_OUTPUT.PUT_LINE('错误：工资过高');
    END;
END;
/
```

## 触发器