技术备忘录：PL/SQL与MongoDB服务端逻辑的综合语法与操作分析

执行摘要

本报告旨在为数据库架构师、开发人员及系统分析师提供一份详尽的技术参考文档，深入剖析两种截然不同的数据库技术——Oracle PL/SQL（过程化SQL）与MongoDB（文档型NoSQL）在服务端应用逻辑层面的语法结构、操作范式及功能特性。基于Lecture 5至Lecture 10的教学与研究材料，本报告不仅是对语法规则的简单罗列，更是对数据操纵逻辑在关系型与非关系型环境下的深度解析。

报告重点针对“列举型”语法问题（Enumeration-type questions）进行了针对性的结构化梳理，通过详尽的表格形式展示关键词、功能定义及代码示例，以满足技术人员在应对复杂业务逻辑实现及技术考核时的需求。全文分为两个主要部分：第一部分探讨在关系型数据库管理系统（RDBMS）中，如何利用PL/SQL的过程化扩展能力实现复杂的服务器端逻辑；第二部分则转向MongoDB的灵活架构，剖析其CRUD操作、高级查询运算符及强大的聚合框架（Aggregation Framework）。

第一部分：PL/SQL服务器端应用逻辑与过程化控制

1. 服务器端逻辑架构概述

服务器端应用逻辑（Server-Side Application Logic）是指那些直接在数据库服务器上而非客户端或应用服务器上执行的程序模块。这种架构设计的核心驱动力在于最小化网络间的数据传输。通过在数据驻留的本地执行代码，应用程序能够充分利用数据库服务器的内存管理与并行处理能力，从而实现巨大的性能提升 [1]。

在Oracle环境中，PL/SQL（Procedural Language/Structured Query Language）是实现这一逻辑

的核心工具。它将过程化编程语言的控制结构（如条件判断、循环）与SQL的数据操作能力无缝集成，使得开发者能够构建复杂的事务处理逻辑、数据验证规则及批处理作业[1]。

2. PL/SQL块结构与匿名块

PL/SQL的基本单元被称为“块”（Block）。深入理解块的解剖结构是掌握PL/SQL语法的基础。块可以分为匿名块（Anonymous Blocks）和命名块（如存储过程、函数、触发器）。

2.1 PL/SQL块的解剖结构

一个标准的PL/SQL块由三个主要部分组成：声明部分（Declarative）、执行部分（Executable）和异常处理部分（Exception Handling）。其中，只有执行部分是强制要求的。

表 1：**PL/SQL** 块结构组件详表

组件部分	关键字标识	状态	功能描述与逻辑意义
声明部分 **(Declarative)**	DECLARE	可选	用于定义变量、常量、游标（Cursor）及用户定义类型。在匿名块中以 DECLARE 开头；在命名程序中位于 IS 或 AS 之后。此部分为后续逻辑分配内存空间。
执行部分 **(Executable)**	BEGIN	强制	包含实现业务逻辑的过程化语句和SQL语句。这是程序的核心控制流区域。必须以 END; 结束。
异常处理 **(Exception)**	EXCEPTION	可选	用于捕获和处理运行时错误（Exception

			)。此部分代码仅在执行部分发生错误时触发，保证程序的健壮性。

代码示例：匿名块的构建

匿名块通常用于测试或一次性脚本执行，它们在运行时被编译并立即执行，但不会作为对象存储在数据库中 [1]。

SQL

```
SET SERVEROUTPUT ON; -- 启用控制台输出功能

DECLARE
   -- 声明部分：定义变量及其初始值
   v_course_code VARCHAR2(10) := 'DBS311';
   v_student_count NUMBER := 0;
BEGIN
   -- 执行部分：业务逻辑
   DBMS_OUTPUT.PUT_LINE('Course Code: ' |

| v_course_code);

   -- 模拟逻辑处理
   v_student_count := v_student_count + 1;
   DBMS_OUTPUT.PUT_LINE('Current Student Count: ' |

| v_student_count);
EXCEPTION
   -- 异常处理部分
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('An unexpected error occurred.');
END;
/
```

1

2.2 变量声明与类型锚定

在PL/SQL中，强类型系统要求所有变量必须先声明后使用。其基本语法为 variable_name datatype [:= initial_value];。为了增强代码的可维护性，应对底层表结构的变化，PL/SQL提供了锚定类型（Anchored Types）。

- **%TYPE** 属性：声明一个变量，使其数据类型与数据库表中的列或另一个变量完全一致。例如 v_lastname employees.lastname%TYPE。如果表结构发生变化（如列长度增加），变量定义会自动更新，无需修改代码 [1]。
- **%ROWTYPE** 属性：声明一个记录类型变量，其结构与表的一整行或游标返回的一行相匹配。例如 v_emp_rec employees%ROWTYPE。这允许一次性处理整行数据，极大地简化了代码 [1]。

3. 控制流语句：条件与迭代

PL/SQL 提供了强大的控制结构，使得在数据库层处理复杂业务规则成为可能。这主要包括条件分支和循环结构。

3.1 条件控制语句

条件语句允许程序根据数据的状态选择不同的执行路径。

表 2：条件控制语句分类

语句类型	语法结构特征	适用场景
IF-THEN	IF condition THEN... END IF;	单一条件判断，满足则执行。
IF-THEN-ELSE	IF... THEN... ELSE... END IF;	二元分支，处理满足与不满足两种情况。
IF-THEN-ELSIF	IF... THEN... ELSIF... ELSE...	多重分支，按顺序评估多个条件，执行第一个为真的分

	END IF;	支。
CASE (Selector)	CASE selector WHEN val1 THEN... ELSE... END CASE;	针对单一变量与多个离散值的等值比较，语法更简洁。
CASE (Searched)	CASE WHEN cond1 THEN... ELSE... END CASE;	针对复杂的布尔表达式（如范围判断），灵活性更高。

代码示例：**CASE** 语句的应用

CASE 语句在处理状态机或枚举类型逻辑时比 IF 更具可读性 [1]。

SQL

```
DECLARE
   v_grade CHAR(1) := 'B';
BEGIN
   CASE v_grade
      WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
      WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
      WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
      ELSE DBMS_OUTPUT.PUT_LINE('Needs Improvement');
   END CASE;
END;
/
```

1

3.2 迭代控制语句（循环）

迭代语句用于重复执行代码块，直到满足特定终止条件。PL/SQL 支持三种主要的循环结构。

表 **3**：迭代语句及其控制关键词

循环类型	语法结构	终止机制与特点
Basic LOOP	LOOP... END LOOP;	默认为无限循环。必须配合 EXIT 或 EXIT WHEN condition 语句来强制终止循环。
WHILE LOOP	WHILE condition LOOP... END LOOP;	在每次迭代开始前评估条件。如果初始条件为假，循环体一次都不执行。
FOR LOOP	FOR i IN lower..upper LOOP... END LOOP;	确定性循环。循环变量 i 隐式声明，自动在指定范围内递增。可使用 REVERSE 关键字递减。

循环控制关键词：

- **EXIT**: 立即终止当前循环，控制权跳转到循环体后的第一条语句 [1]。
- **EXIT WHEN**: 当指定条件为真时终止循环，是 Basic Loop 的标准退出方式 [1]。
- **CONTINUE**: 跳过当前迭代的剩余部分，直接开始下一次迭代 [1]。
- **CONTINUE WHEN**: 当条件为真时跳过当前迭代 [1]。

4. 游标（Cursors）：多行数据处理

SQL 是一种面向集合的语言，通常一次性操作一组行。然而，过程化逻辑往往需要逐行处理数据。游标（Cursor）正是这种集合导向与行导向之间的桥梁。游标是指向上下文区域（Context Area）的指针，该区域包含了 SQL 语句执行后的结果集 [1]。

4.1 隐式游标与显式游标

- 隐式游标（**Implicit Cursors**）：由 Oracle 自动为所有 DML 语句（INSERT, UPDATE, DELETE）和单行 SELECT INTO 语句创建。开发者无法直接控制其打开和关闭，但可以通过属性检查其状态 [1]。
- 显式游标（**Explicit Cursors**）：由开发者在声明部分显式定义，用于处理返回多行数据的

SELECT 语句。开发者拥有完全的控制权[1]。

4.2 显式游标的生命周期

使用显式游标必须遵循严格的四个步骤。

表 4：显式游标操作步骤

步骤	关键字/语法	功能说明
1. 声明 (Declare)	CURSOR name IS select_stmt;	在声明部分定义游标及其对应的查询语句。此时不执行查询。
2. 打开 (Open)	OPEN name;	分配内存，执行查询，识别结果集（Active Set），指针指向第一行之前。
3. 提取 (Fetch)	FETCH name INTO variables;	将当前行的数据加载到变量中，并将指针下移。通常在循环中执行。
4. 关闭 (Close)	CLOSE name;	释放游标占用的资源和上下文区域。

代码示例：带参数的显式游标

游标可以接受参数，使其在不同条件下复用[1]。

SQL

```
DECLARE
  -- 声明带参数的游标
  CURSOR c_products (p_min_price NUMBER) IS
    SELECT product_name, list_price
```

```
        FROM products
        WHERE list_price > p_min_price;

    v_pname products.product_name%TYPE;
    v_price products.list_price%TYPE;
BEGIN
    -- 打开游标，传入参数值 100
    OPEN c_products(100);
    LOOP
        FETCH c_products INTO v_pname, v_price;
        -- 检查游标是否已耗尽数据
        EXIT WHEN c_products%NOTFOUND;

        DBMS_OUTPUT.PUT_LINE(v_pname |

| ': ' |
| v_price);
    END LOOP;
    CLOSE c_products;
END;
/
```

[1]

4.3 游标属性

游标属性用于在运行时获取游标的状态信息，对于控制循环逻辑至关重要。

表 5：游标属性详解

属性	返回类型	描述
%FOUND	Boolean	如果最近一次 FETCH 成功返回一行，则为 TRUE；否则为 FALSE [1]。
%NOTFOUND	Boolean	如果最近一次 FETCH 未能返回行（已到末尾），则为

		TRUE。常用于循环退出条件[1]。
%ROWCOUNT	Number	返回迄今为止已提取的总行数。在打开前或关闭后访问会报错或返回无效值[1]。
%ISOPEN	Boolean	如果游标处于打开状态，则为 TRUE。用于避免重复打开已打开的游标[1]。

4.4 SELECT INTO 与异常处理

SELECT INTO 是一种隐式游标操作，用于将单行查询结果直接赋值给变量。它必须严格返回一行数据，否则会触发异常。

- **NO_DATA_FOUND**: 当查询未返回任何行时触发[1]。
- **TOO_MANY_ROWS**: 当查询返回多于一行时触发。这是与游标的主要区别，游标可以处理多行，而 SELECT INTO 不能[1]。

5. 存储过程与用户定义函数

存储过程（Stored Procedures）和函数（User Defined Functions, UDFs）是命名的 PL/SQL 块，存储在数据库中，可被反复调用。

5.1 过程与函数的区别

表 6：存储过程与函数的对比

特性	存储过程 **(Procedure)**	用户定义函数 **(Function)**
返回值	不强制返回值。可以通过	必须通过 RETURN 子句返回

	OUT 参数返回多个值。	且仅返回一个值[1]。
调用方式	作为独立的语句执行（如 EXECUTE 或 CALL）。	作为表达式的一部分在 SQL 语句（SELECT, WHERE 等）或 PL/SQL 赋值中使用。
用途	主要用于执行操作（如数据更新、复杂的业务流程）。	主要用于计算并返回结果（如数值计算、字符串处理）。

5.2 参数模式

参数决定了数据如何传入和传出子程序。

- **IN**: 默认模式。参数值由调用者传入，在子程序内部为只读，不能修改[1]。
- **OUT**: 参数值由子程序返回给调用者。在子程序内部初始化为 NULL，必须赋值[1]。
- **IN OUT**: 参数既传入初始值，又可被子程序修改并返回新值[1]。

代码示例：创建存储过程

SQL

```
CREATE OR REPLACE PROCEDURE UPDATE_STAFF_COMM (
    p_yearsin IN NUMBER,
    p_new_comm IN NUMBER
) AS
BEGIN
    UPDATE staff
    SET comm = p_new_comm
    WHERE years = p_yearsin;

    IF SQL%ROWCOUNT = 0 THEN
        DBMS_OUTPUT.PUT_LINE('No staff updated.');
    ELSE
        DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT |

| ' staff updated.');
```

```
    END IF;
END UPDATE_STAFF_COMM;
/
```

1

代码示例：创建函数

函数可以直接在 SQL 查询中使用，例如计算价格差异。

SQL

```
CREATE OR REPLACE FUNCTION CALCULATE_MARKUP (
    p_msrp IN NUMBER,
    p_buyprice IN NUMBER
) RETURN NUMBER AS
    v_markup NUMBER;
BEGIN
    v_markup := p_msrp - p_buyprice;
    RETURN v_markup;
END CALCULATE_MARKUP;
/
```

6. 触发器（Triggers）

触发器是一种特殊的存储过程，它不能被显式调用，而是由特定的数据库事件（如 INSERT, UPDATE, DELETE）自动触发执行。它们常用于审计、强制复杂的完整性约束或自动生成派生数据[1]。

6.1 触发器语法要素

创建触发器需要定义时机（Timing）、事件（Event）和作用域（Scope）。

表 7：触发器关键语法组件

组件	选项	说明
时机 **(Timing)**	BEFORE	在数据库执行触发事件之前运行逻辑（常用于验证或修改数据）。
	AFTER	在触发事件执行之后运行逻辑（常用于审计或级联操作）。
事件 **(Event)**	INSERT, UPDATE, DELETE	激活触发器的 DML 操作。
作用域 **(Level)**	FOR EACH ROW	行级触发器。对受影响的每一行执行一次。
	（省略）	语句级触发器。无论影响多少行，每个语句只执行一次。
引用变量	:OLD	引用更新或删除前的旧值（仅行级有效）。
	:NEW	引用插入或更新后的新值（仅行级有效）[1]。

代码示例：审计触发器

此触发器在更新或删除客户信息后，将旧数据记录到审计表中 [1]。

SQL

```
CREATE OR REPLACE TRIGGER audit_customer_changes
  AFTER UPDATE OR DELETE ON customers
  FOR EACH ROW
DECLARE
```

```
  v_action VARCHAR2(10);
BEGIN
  IF UPDATING THEN
    v_action := 'UPDATE';
  ELSIF DELETING THEN
    v_action := 'DELETE';
  END IF;

  INSERT INTO audit_log (table_name, action, user_name, trans_date, old_id)
  VALUES ('CUSTOMERS', v_action, USER, SYSDATE, :OLD.customer_id);
END;
/
```

1

第二部分：MongoDB 语法与非关系型数据操作

7. NoSQL 转型与 MongoDB 基础

与 Oracle 的强模式（Schema-heavy）关系型结构不同，MongoDB 是一种面向文档（Document-Oriented）的 NoSQL 数据库。它放弃了固定的表结构，转而使用灵活的 BSON（Binary JSON）文档。这种设计支持动态模式，允许在不中断服务的情况下修改数据结构，非常适合快速迭代开发 [1]。

表 8：关系型数据库与 MongoDB 术语映射

关系型数据库 (RDBMS)	MongoDB (NoSQL)	概念差异解析
Database	**Database**	顶层容器，概念基本一致。
Table	**Collection** (集合)	集合不强制统一结构，文档可以有不同的字段。
Row	**Document** (文档)	数据以键值对（Key-Value）形式存储，支持嵌套结构（数

		组、子文档）。
Column	**Field** (字段)	字段是动态的，无需预先定义类型。
Join	**Embedded Document**	通过嵌入子文档来减少关联查询，提升读取性能；也支持 $lookup 进行类似 Join 操作。

8. CRUD 基础操作

MongoDB 的增删改查（CRUD）操作通过 JavaScript 方法在 Shell 或驱动程序中执行。

8.1 创建（Create）

插入操作用于将文档添加到集合中。如果集合不存在，MongoDB 会自动创建。

- insertOne(): 插入单个文档。
- insertMany(): 插入文档数组。

代码示例：插入操作

JavaScript

```
// 插入单个文档，包含嵌套对象
db.MyCustomers.insertOne({
   "_id": 103,
   "name": "Atelier graphique",
   "city": "Nantes",
   "contact": { "first": "Carine", "last": "Schmitt" }, // 嵌入式文档
   "creditlimit": 21000
});
```

```
// 批量插入
db.MyCustomers.insertMany( }
]);
```

1

8.2 读取（Read）

find() 方法是查询的核心。它接受两个可选参数：查询过滤器（Filter）和投影（Projection）。

- 语法: db.collection.find({query}, {projection})

代码示例：查询与投影

JavaScript

```
// 查询所有居住在 USA 的客户
db.MyCustomers.find({ "country": "USA" });

// 投影：仅返回 name 和 city 字段，显式排除 _id（默认包含）
db.MyCustomers.find(
  { "country": "USA" },
  { "name": 1, "city": 1, "_id": 0 }
);

// 游标修饰符：限制返回结果数量
db.MyCustomers.find({ "country": "USA" }).limit(5);
```

1

8.3 删除（Delete）

推荐使用 deleteOne() 和 deleteMany()，旧的 remove() 方法已被弃用。

JavaScript

```
// 删除 _id 为 124 的特定文档
db.MyCustomers.deleteOne({ "_id": 124 });

// 删除所有 country 为 France 的文档
db.MyCustomers.deleteMany({ "country": "France" });
```

1

9. 高级查询运算符

MongoDB 的强大之处在于其丰富的查询运算符，均以 $ 符号开头。这对于应对列举型问题至关重要。

表 9：比较与逻辑运算符

运算符	描述	代码示例
$eq	等于（Equal to）	{"age": {$eq: 25}} (通常简写为 {"age": 25})
$gt / $gte	大于 / 大于等于	{"credit": {$gte: 5000}}
$lt / $lte	小于 / 小于等于	{"date": {$lt: new Date("2023-01-01")}}
$ne	不等于（Not Equal）	{"status": {$ne: "inactive"}}
$in	包含于数组中（In Array）	{"_id": {$in: }}
$nin	不包含于数组中	{"type": {$nin: ["food", "drink"]}}

$or	逻辑或（OR），匹配任一条件	{$or: [{"qty": {$lt: 20}}, {"price": 10}]}
$and	逻辑与（AND），匹配所有条件	{$and: [{"qty": {$ne: 20}}, {"price": 10}]}
$not	逻辑非，反转条件	{"price": {$not: {$gt: 100}}}
$exists	检查字段是否存在	{"phone2": {$exists: true}}

1

表 10：数组查询运算符

针对数组字段的查询是 MongoDB 的特色功能。

运算符	描述	代码示例
$all	匹配包含所有指定元素的数组（顺序无关）。	{"tags": {$all: ["red", "blue"]}}
$elemMatch	匹配数组中至少有一个元素同时满足多个条件。这对对象数组至关重要。	{"scores": {$elemMatch: {$gt: 80, $lt: 90}}}
$size	匹配特定长度的数组。	{"tags": {$size: 3}}

1

10. 更新操作与原子修改器

MongoDB 的更新操作不仅可以修改值，还可以重构文档结构。

10.1 更新方法

- **updateOne() / updateMany()**: 仅更新匹配到的文档中的特定字段，保留其他字段不变 [1]。
- **replaceOne()**: 用新文档完全替换旧文档（_id 除外），旧文档中未在新文档中出现的字段将丢失 [1]。

10.2 字段更新运算符

在 update 语句的第二个参数（修改器文档）中使用。

表 11：字段修改运算符

运算符	功能	代码示例
$set	设置字段的值。如果字段不存在，则创建该字段。	{$set: {"status": "active", "score": 10}}
$inc	对数值字段进行增量更新（可增可减）。	{$inc: {"views": 1, "balance": -50}}
$unset	删除字段。	{$unset: {"temporary_flag": 1}}

[1]

10.3 数组修改运算符

用于高效地操作数组字段，而无需读取整个数组。

表 12：数组修改运算符

运算符	功能	复杂组合示例
$push	向数组末尾添加一个元素。	{$push: {"comments": "Great!"}}

$each	配合 $push 使用，一次添加多个元素。	{$push: {"scores": {$each: }}}
$sort	配合 $push 使用，在添加元素后对数组排序。	{$push: {"scores": {$each: , $sort: 1}}}
$slice	配合 $push 使用，限制数组的最大长度（保留最后 N 个）。	{$push: {"scores": {$each:, $slice: -5}}}
$addToSet	仅当元素不存在时才添加到数组（去重）。	{$addToSet: {"tags": "unique"}}

[1]

11. 聚合框架（Aggregation Framework）

聚合框架是 MongoDB 进行高级数据分析和转换的核心工具，类似于 SQL 中的 GROUP BY 和复杂报表查询。它基于管道（Pipeline）模型，文档流经多个阶段（Stage），每个阶段对数据进行处理并传递给下一阶段 [1]。

语法: db.collection.aggregate([{stage1}, {stage2},...])

11.1 核心管道阶段

表 13：聚合管道阶段详解

阶段	描述	SQL 等价概念
$match	过滤文档流，仅允许符合条件的文档通过。应尽量置于管道首部以减少后续数据量。	WHERE

$project	字段投影与重塑。可选择字段、重命名字段或通过表达式计算新字段。	SELECT
$group	将文档按指定标识符分组，并应用累加器计算聚合值。	GROUP BY
$sort	对文档流进行排序。	ORDER BY
$limit	限制传递给下一阶段的文档数量。	LIMIT
$skip	跳过前 N 个文档。	OFFSET

1

11.2 累加器与表达式

在 $group 和 $project 阶段中，使用特定的操作符进行计算。

表 **14**：聚合累加器（用于 **$group**）

累加器	功能	示例代码
$sum	计算数值总和。$sum: 1 可用于计数。	"totalSales": {$sum: "$amount"}
$avg	计算平均值。	"avgPrice": {$avg: "$price"}
$min / $max	找出最小值 / 最大值。	"maxScore": {$max: "$score"}
$first / $last	返回组内第一个 / 最后一个文档的值（需配合 $sort 使用才有意义）。	"latestDate": {$first: "$date"}

[1]

代码示例：综合聚合查询

此示例筛选类型为 "Biography" 的有声书，按流派分组，计算最短和最长时长，以及总时长，最后按总时长降序排列 [1]。

JavaScript

```
db.audioBooks.aggregate();
```

代码示例：字符串投影与生成

在 $project 阶段使用字符串操作符生成新字段 [1]。

JavaScript

```
db.audioBooks.aggregate([
  {
    "$project": {
      "title": 1,
      "generated_email": {
        "$concat": [
          { "$substr": ["$author.first", 0, 1] }, // 取名字首字母
          ".",
          "$author.last",
          "@audiobooks.com"
        ]
      }
    }
  }
]);
```

12. 结论

本报告系统地梳理了 PL/SQL 与 MongoDB 在服务端逻辑层面的语法体系。PL/SQL 以其严谨的块结构、显式的异常处理和精细的游标控制，为构建强一致性、高事务性的业务逻辑提供了坚实基础。其过程化特性使得开发者能够精确控制数据处理的每一步。

相比之下，MongoDB 的语法反映了其作为文档型数据库的灵活性与扩展性。从 $elemMatch 对嵌套数组的精准查询，到 $push 配合 $slice 的复杂数组维护，再到聚合管道对数据流的转化，MongoDB 提供了一套完全不同但同样强大的语义，特别适用于处理层级数据和快速迭代的数据模型。

掌握这两套截然不同的语法体系，不仅能够应对技术考核中的列举型问题，更能使技术人员在面对不同的业务场景（强事务 vs 高并发/灵活模式）时，做出最恰当的技术选型与实现方案。

引用的著作

1. DBS311 Lecture 5 F25 - Tagged.pdf