虚谷数据库 V12.8

SQL 语法参考指南

文档版本 01

发布日期 2025-04-15

虚谷数据库
XUGU DB

声明

未经本公司正式书面许可，任何企业和个人不得擅自摘抄、复制、使用本文档中的部分或全部内容，且不得以任何形式进行传播。否则，本公司将保留追究其法律责任的权利。

用户承诺在使用本文档时遵守所有适用的法律法规，并保证不以任何方式从事非法活动。不得利用本文档内容进行任何侵犯他人权益的行为。

商标声明

虚谷数据库 XUGU DB

为成都虚谷伟业科技有限公司的注册商标。

本文档提及的其他商标或注册商标均非本公司所有。

注意事项

您购买的产品或服务应受本公司商业合同和条款的约束，本文档中描述的部分产品或服务可能不在您的购买或使用范围之内。由于产品版本升级或其他原因，本文档内容将不定期进行更新。

除非合同另有约定，本文档仅作为使用指导，所有内容均不构成任何声明或保证。

成都虚谷伟业科技有限公司

地址：四川省成都市锦江区锦盛路 138 号佳霖科创大厦 5 楼 3-14 号

邮编：610023

网址：www.xugudb.com

前言

概述

本文档详细介绍了虚谷数据库的 SQL 语法及其用法。

读者对象

- 数据库管理员
- 软件工程师
- 技术支持工程师

符号约定

在本文中可能出现下列标志，它们所代表的含义如下。

符号	说明
⚠ 注意	用于传递设备或环境安全警示信息，若不避免，可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。
🕮 说明	对正文中重点信息的补充说明。“说明”不是安全警示信息，不涉及人身、设备及环境伤害信息。

修改记录

文档版本	发布日期	修改说明
01	2025-04-15	第一次发布

目录

1 SQL 概述

SQL 简介

SQL（Structured Query Language）是一种用于管理关系型数据库系统的标准化语言。它是一种声明式语言，通过编写简洁的语句来描述要执行的操作，而不需要指定具体的实现方式。SQL 允许用户对数据库进行各种操作，包括数据查询、插入、更新和删除等。通过使用 SQL，用户可以轻松地从数据库中检索所需的数据，进行复杂的数据操作，并定义数据库的结构和约束。

SQL 标准

虚谷数据库语法兼容主流数据库的语法，支持 SQL 89/92/99 语法标准，实现了关系型数据库常用对象管理，包括：表、视图、存储过程/函数、包、同义词、触发器等，但其它数据库的方言，需根据实际功能进行兼容性修改。

2 数据类型

2.1 概述

数据库支持的数据类型可分为基本数据类型和复合数据类型。

- 基本数据类型可分为数值类型、字符类型、时间类型、大对象类型、布尔类型、二进制类型、全局唯一标识符 (GUID) 类型。
- 复合数据类型包括用户自定义数据类型 (UDT)、记录类型和集合类型。

为兼容其他数据库的数据类型，在部署数据库时，可配置安装路径下 SETUP 文件夹中 types.ini 数据类型映射文件，将其他数据库的数据类型进行映射，后续在使用其他数据库数据类型名称时系统将自动进行数据类型映射转换。

2.2 数值数据类型

整型数据类型

TINYINT、SMALLINT、INTEGER、BIGINT

- TINYINT：存储数据范围自-128 至 127
- SMALLINT：存储数据范围自-32768 至 32767
- INTEGER：存储数据范围自-2147483648 至 2147483647
- BIGINT：存储数据范围自-9223372036854775808 至 9223372036854775807

固定精度数据类型

NUMERIC[(M[,D])]

- NUMERIC 存储定长精确数据，其中 M 为数据精度，D 为数据标度。当不指定数据精度与标度时，默认存储 NUMERIC(12,0) 的数值记录；若 D 为 0，则数值无小数部分，操作数据的小数值超过 D 值定义，系统自动进行截断存储。数值存储最大支持 38 位，D 取值范围为 [0，M]。

浮点数据类型

FLOAT、DOUBLE

- FLOAT：单精度浮点型数据类型，可存储 7 位有效数的数据

- DOUBLE：双精度浮点型数据类型，可存储 16 位有效数的数据

数据类型特性表

数据类型	长度	Java 数据类型	长度	封装器类	说明
TINYINT	1Byte	byte	1 字节 (8 位)	Byte	最大存储数据量是 255，取值范围是 [-128, 127] 的整数数据，超界报错，无法插入。
SMALLINT	2Byte	short	2 字节 (16 位)	Short	最大数据存储量是 65536，取值范围是-2 的 15 次方到 2 的 15 次方-1，即 [-32768, 32767] 的整数数据，超界报错，无法插入。
INTEGER	4Byte	int	4 字节 (32 位)	Integer	最大数据存储容量是 2 的 32 次方减 1，取值范围是-是 2 的 32 次方到是 2 的 32 次方-1，即 [-2147483648, 2147483647] 的整型数据，超界报错，无法插入。
BIGINT	8Byte	long	8 字节 (64 位)	Long	最大数据存储容量是 2 的 64 次方减 1，取值范围是-263 到 263-1，即 [-9223372036854775808, 9223372036854775807] 的整型数据，超界报错，无法插入。

接下页

数据类型	长度	Java 数据类型	长度	封装器类	说明
FLOAT[(SIZE)]	4Byte	float	4Byte	Float	保存单精度浮点数据类型，数字可以为零，也可以在-3.402E+38 到-1.175E-37 或 1.175E-37 到 3.402E+38 的范围内，SIZE 范围为 1 到 128 之间的整数值。SIZE 默认值是 7。FLOAT 的有效精度为 7，超过 7 位时数字准确性会丢失。
DOUBLE	8Byte	double	8 字节 (64 位)	Double	保存双精度浮点数据类型，数字可以为零，也可以在-1.797E+308 到-2.225E-308 以及 2.225E-308 到 1.797E+308 的范围内。有效精度默认是 15，超过 15 位数字准确性会丢失。
接下页					

数据类型	长度	Java 数据类型	长度	封装器类	说明
NUMERIC([(P)],[(S)])	2-17Byte	java.math.BigDecimal	变长	BigDecimal	固定精度和比例的数字。允许从-10 的 38 次方加 1 到 10 的 38 次方减 1 之间的数字。P 参数表示可以存储的最大位数（小数点左侧和右侧）。P 必须是 1 到 38 之间的值，默认是 12。S 必须是 0 到 P 之间的值，默认是 0。 • 当一个数的整数部分的长度 > P-S 时，报错。 • 当一个数的小数部分的长度 > S 时，舍入。

2.3 字符数据类型

定长字符类型

CHAR[(SIZE)]

变长字符类型

VARCHAR[(SIZE)]、VARCHAR2[(SIZE)]

数据库中，指定精度的字符数据类型，均按字符长度计算；若不指定精度，CHAR 类型默认允许存储一个字符长度的数据，VARCHAR 类型默认允许存储当前行记录除其他字段所占空间的剩余空间 (行记录大小为 64K)。

数据类型特性表

数据类型	长度	Java 数据类型	长度	封装器类	说明
CHAR/NCHAR	[(SIZE)]	java.lang.String	定长	String	用于保存定长的字符串数据。若数据超过类型定义时的最大长度（单位：字符）时，则：报错；或者把数据截断至最大长度后操作成功，并返回警告信息（取决于 str_trunc_warning 参数值的值）。对于中文等多字节字符，在不同字符集的库下，每个字符的长度不同。
VARCHAR/VARCHAR2	[(SIZE)]	java.lang.String	变长	String	用于保存变长的字符串数据。若数据超过类型定义时的最大长度（单位：字符）时，则：报错；或者把数据截断至最大长度后操作成功，并返回警告信息（取决于 str_trunc_warning 参数的值）。对于中文等多字节字符，在不同字符集的库下，每个字符的长度不同。

2.4 时间数据类型

数据库中时间数据类型可分为：日期、时间、日期时间、时间戳以及时间间隔数据类型。

日期类型

- DATE

存储年、月、日，支持范围为 0001-01-01 至 9999-12-31

时间类型

- TIME
- TIME WITH TIME ZONE

存储时、分、秒、毫秒 [、时区]，时间支持范围为 00:00:00 至 23:59:59，时区支持-12 至 +14

日期时间类型

- DATETIME
- DATETIME WITH TIME ZONE

存储年、月、日、时、分、秒、毫秒 [、时区]，年、月、日支持范围与日期类型一致，时、分、秒支持范围与时间类型一致，时区支持范围与时间类型一致

时间戳类型

- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

存储年、月、日、时、分、秒、毫秒 [、时区]，支持范围与日期时间类型一致，时间戳类型默认插入当前系统时间，可根据需求设置记录变更时自动变更该行数据时间戳 (AUTO UPDATE)。

时间间隔类型

INTERVAL 数据类型用来存储两个时间戳之间的时间间隔，各类型和所占空间如下表：

类型	占用字节	最大精度	最大标度
INTERVAL YEAR	4	9	
INTERVAL MONTH	4	9	
INTERVAL DAY	4	9	
INTERVAL HOUR	4	9	
INTERVAL MINUTE	4	9	
INTERVAL SECOND	8	6	
INTERVAL YEAR TO MONTH	4	8	

接下页

类型	占用字节	最大精度	最大标度
INTERVAL DAY TO HOUR	4	7	
INTERVAL DAY TO MINUTE	4	6	
INTERVAL DAY TO SECOND	8	7	6
INTERVAL HOUR TO MINUTE	4	7	
INTERVAL HOUR TO SECOND	8	7	6
INTERVAL MINUTE TO SECOND	8	7	6

时间字段可取值范围

时间字段	时间类型有效值	时间间隔类型有效值
YEAR	1 至 9999	0 至 999,999,999
MONTH	01 至 12	0 至 11
DAY	01 至 31	01 至 31
HOUR	00 至 23	0 至 23
MINUTE	00 至 59	0 至 59
SECOND	00 至 59.999（精确到小数点后三位），不适用于 DATE 类型	0 至 59.999999(SIZE)，SIZE 最大精度为 6

数据类型特性表

数据类型	长度	Java 数据类型	长度	封装器类	说明
DATE	4Byte	java.sql.Date	4 字节(32 位)	Date	格式：YYYY-MM-DD，表示范围公元前’9999-12-31 BC’ 到’0002-01-01 BC’以及公元后’0001-01-01’到’9999-12-31’。使用字符串或 TO_DATE 函数赋值。公元前、公元后日期分别以 BC 和 AD 表示，位于时间字符串末尾。Oracle 兼容模式时，DATE 类型映射为 DATETIME 类型。
DATETIME	8Byte	java.sql.Timestamp	8Byte	Timestamp	格式：YYYY-MM-DD HH24:MI:SS，支持的范围是公元前’9999-12-31 23:59:59.999 BC’ 到’0002-01-01 00:00:00.000 BC’以及公元后’0001-01-01 00:00:00.000’ 到’9999-12-31 23:59:59.999’。使用字符串或 TO_DATE 函数赋值公元前、公元后日期分别以 BC 和 AD 表示，位于时间字符串末尾。秒值存储精度为小数点后三位。

接下页

数据类型	长度	Java 数据类型	长度	封装器类	说明
TIMESTAMP[(SIZE)]	8Byte	java.sql.Timestamp	8 字节(64 位)	Timestamp	格式：YYYY-MM-DD HH24:MI:SS，支持的范围是公元前’9999-12-31 23:59:59.999 BC’到’0002-01-01 00:00:00.000 BC’以及公元后’0001-01-01 00:00:00.000’到’9999-12-31 23:59:59.999’。使用字符串或 TO_DATE 函数赋值。公元前、公元后日期分别以 BC 和 AD 表示，位于时间字符串末尾。SIZE 为 XuGu 存储秒值小数部分位数，默认为 3，可选值为 0 到 6。注：数据库只支持到秒值小数精度后三位
DATETIME WITH TIME ZONE	10Byte	java.lang.String	变长	String	DATETIME WITH TIME ZONE 值包含年、月、日、小时、分钟、秒、秒的小数部分，以及在协调通用时间（Coordinated Universal Time，简称 UTC）前后的分钟数。小数存储到 3 个小数位。其中时区的取值范围为：-12:59 + 14:59。

接下页

数据类型	长度	Java 数据类型	长度	封装器类	说明
TIMESTAMP[(SIZE)] WITH TIME ZONE	10Byte	java.lang.String	变长	String	TIMESTAMP WITH TIME ZONE 值包含年、月、日、小时、分钟、秒、秒的小数部分，以及在协调通用时间（Coordinated Universal Time，简称 UTC）前后的分钟数。小数存储到 3 个小数位。其中时区的取值范围为：-12:59 + 14:59。
TIME	4Byte	java.sql.Time	4Byte	Time	格式：HH24:MI:SS。支持的范围是'00:00:00'到'23:59:59'。秒值存储精度为小数点后三位。
TIME WITH TIME ZONE	6Byte	java.lang.String	变长	String	格式：HH24:MI:SS +UTC，支持的范围是'00:00:00.000'到'23:59:59.999'，秒值存储精度为小数点后三位。包含时区 (UTC 信息)，其中时区的取值范围为：-12:59 + 14:59。

接下页

数据类型	长度	Java 数据类型	长度	封装器类	说明
INTERVAL YEAR[(SIZE)]	4Byte	java.lang.String	变长	String	包含年的时间间隔类型。SIZE 是年字段的数字位数，最大精度为 9 位，取值范围从 1 至 9。SIZE 默认值：9。超过 SIZE 定义范围，报错，无法插入数据。取值范围从 0 到 999999999
INTERVAL MONTH[(SIZE)]	4Byte	java.lang.String	变长	String	包含月的时间间隔类型。SIZE 是月字段的数字位数，最大精度为 9 位，取值范围从 1 至 9。SIZE 默认值：9。超过 SIZE 定义范围，报错，无法插入数据。取值范围从 0 到 999999999
INTERVAL DAY[(SIZE)]	4Byte	java.lang.String	变长	String	包含日的时间间隔类型。SIZE 是日字段的数字位数，最大精度为 9 位，取值范围从 1 至 9。SIZE 默认值：9。超过 SIZE 定义范围，报错，无法插入数据。取值范围从 0 到 999999999

接下页

数据类型	长度	Java 数据类型	长度	封装器类	说明
INTERVAL HOUR[(SIZE)]	4Byte	java.lang.String	变长	String	包含小时的时间间隔类型。SIZE 是小时字段的数字位数，最大精度为 9 位，取值范围从 1 至 9。SIZE 默认值：9。超过 SIZE 定义范围，报错，无法插入数据。取值范围从 0 到 999999999
INTERVAL MINUTE[(SIZE)]	4Byte	java.lang.String	变长	String	包含分钟的时间间隔类型。SIZE 是分钟字段的数字位数，最大精度为 9 位，取值范围从 1 至 9。SIZE 默认值：9。超过 SIZE 定义范围，报错，无法插入数据。取值范围从 0 到 999999999
INTERVAL SECOND([(P)], [(S)])	8Byte	java.lang.String	变长	String	包含秒的时间间隔类型。P 是秒字段的数字位数，取值范围从 1 至 9。S 为微秒表示精度，取值范围从 1 至 6。默认值：9、6。
接下页					

数据类型	长度	Java 数据类型	长度	封装器类	说明
INTERVAL YEAR[(SIZE)] TO MONTH	4Byte	java.lang.String	变长	String	包含年、月的时间间隔类型。SIZE 是年表示数字位数，取值范围从 1 至 8，SIZE 默认值：8。YEAR 取值范围从 0 到 99999999，MONTH 取值范围从 0 到 11。
INTERVAL DAY[(SIZE)] TO HOUR	4Byte	java.lang.String	变长	String	包含日、小时的时间间隔类型。SIZE 是日表示数字位数，取值范围从 1 至 7，SIZE 默认值：7。DAY 取值范围从 0 到 999999，HOUR 取值范围从 0 到 23。
INTERVAL DAY[(SIZE)] TO MINUTE	4Byte	java.lang.String	变长	String	包含日、分钟的时间间隔类型。SIZE 是日表示数字位数，取值范围从 1 至 6，SIZE 默认值：6。DAY 取值范围从 0 到 99999，HOUR 取值范围从 0 到 23，MINUTE 取值范围从 0 到 59。最大值：’999999 23:59’
接下页					

数据类型	长度	Java 数据类型	长度	封装器类	说明
INTERVAL DAY[(P)] TO SECOND[(S)]	8Byte	java.lang.String	变长	String	包含日、秒的时间间隔类型。P 是日表示的数字位数，取值范围从 1 至 6，DAY 取值范围从 0 到 99999，HOUR 取值范围从 0 到 23，MINUTE 取值范围从 0 到 59，SECOND 取值范围从 0 到 59。S 为微秒表示精度，取值范围从 1 至 6。默认值：6、6。最大值：’999999 23:59:59.999999’
INTERVAL HOUR[(SIZE)] TO MINUTE	4Byte	java.lang.String	变长	String	包含小时、分钟的时间间隔类型。SIZE 是小时数字位数，最大精度为 7，SIZE 默认值：7。SIZE 取值范围从 0 到 9999999,MINUTE 取值范围从 0 到 59。最大值：’9999999:59’
接下页					

数据类型	长度	Java 数据类型	长度	封装器类	说明
INTERVAL HOUR[(P)] TO SECOND[(S)]	8Byte	java.lang.String	变长	String	包含时、秒的时间间隔类型。P 是小时表示的数字位数，取值范围从 1 至 7，HOUR 取值范围从 0 到 9999999，MINUTE 取值范围从 0 到 59，SECOND 取值范围从 0 到 59。S 为微秒表示精度，取值范围从 1 至 6。默认值：7、6。最大值：’9999999:59:59.999999’
INTERVAL MINUTE[(P)] TO SECOND[(S)]	8Byte	java.lang.String	变长	String	包含分、秒的时间间隔类型。P 是分钟表示的数字位数，取值范围从 1 至 7，MINUTE 取值范围从 0 到 9999999，SECOND 取值范围从 0 到 59。S 为微秒表示精度，取值范围从 1 至 6。默认值：7、6。最大值：’9999999:59.999999’

2.5 大对象数据类型

大对象数据类型

- BLOB
- CLOB

大对象数据类型最大支持 2GB 存储，单表支持多个大对象字段定义。

BLOB 类型保存二进制数据，CLOB 类型保存文本数据。未超过 512 字节存储在行内，超过 512 字节存储在行外。

数据类型特性表

数据类型	长度	Java 数据类型	长度	封装器类	说明
BLOB	2GB	Java.sql.Blob	变长	Blob	存储非结构化二进制文件。没有字符集语义的比特流，一般是图像、声音、视频等文件。默认为 NULL。
CLOB	2GB	Java.sql.Clob	变长	Clob	存储单字节或者多字节字符数据。支持固定宽度和可变宽度的字符集。默认为 NULL。存储数据与数据库字符集相关，读取字符集不一致时，可能出现乱码问题。

2.6 其他类型

布尔数据类型

BOOLEAN

在数据库系统中，布尔数据类型为 3 态数据类型，包括：TRUE、FALSE 与 UNKNOWN，分别对应数值 1、0、-1。

二进制数据类型

BINARY

BINARY 类型以行内存储的方式进行存储，最大支持 64K 长度。

全局唯一标识符 GUID

该类型用于唯一标识某一字段，可使用 SYS_GUID 函数生成数据，该值存储 32 位字符。

数据类型特性表

数据类型	长度	Java 数据类型	长度	封装器类	说明
BOOLEAN	1Byte	bool	1 字节 (8 位)	Boolean	BOOLEAN 可能是以下几种：TRUE、FALSE、UNKOWN、NULL。默认值：FALSE。
BINARY	变长，不超过 64KB	byte[]	变长	-	存储二进制数据，默认值 NULL。
GUID	16Byte	java.lang.String	变长	String	通过 SYS_GUID() 函数获取值。

2.7 JOSN 数据类型

JOSN 存储类型

数据采用大对象存储，支持最大 2GB 文本。

JOSN 数据格式

JOSN 数据支持存储值的基础类型为 STRING、BOOL、NUMBER、NULL。

> **⚠ 注意**
>
> 注意以下 JOSN 数据都是以 JOSN 串方式展示，所以数据外部应和原有 char 类型一样，由单引号' 包裹。

- json string 是由双引号包裹的字符串：

```
'"中 文"'
```

- json bool 是小写 true false：

```
'true' 'false'
```

- json number：

```
'1' '-1' '10.2'
```

- json null：

```
'null'
```

- json 数组是包含在 [ ] 字符之间以逗号分割的值列表：

```
'["abc", true, false, 1, 1.1, null]'
```

- json 对象是包含在 字符之间的多组键值对，键值必须为 string：

```
'{"key1": "value", "key2": true, "key3": false, "key4": 1, "key5
  ": 1.1, "key6": null}'
```

JSON 路径表达式（JSONPath）

虚谷支持一个路径表达式来检索 JSON 数据中特定的元素。

- 由 $ 字符打头，代表 JSON 文档本身。

```
'{"test": 1}'->'$' = {"test": 1}
```

- (.) 点字符用于寻找对象中的键值对。

```
'{"test": 1}'->'$.test' = 1
-- 或
'{"test": 1}'->'$."test"' = 1
```

- [N] 用于寻找数组中下标为 N 的元素。

```
'[1,3,5,7]'->'$[1]' = 3
```

- [M to N] 用于寻找数组中下标 M 到 N 的元素集合。

```
'[1,3,5,7]'->'$[1 to 3]' = [3,5,7]
```

- last 关键字做为数组最后一个元素或非数组元素的同义词。

```
'[1,3,5,7]'->'$[last]' = 7
-- 或 last - N 作为相对寻址
'[1,3,5,7]'->'$[last - 1 to last]' = [5,7]
-- 非数组元素 与直接使用 $ 相同
'"123"'->'$[last]' = "123"
'{"a": 1}'->'$[last]' = {"a": 1}
```

- * 通配符，代表全量元素。

```
'[1,2,3,4]'->'$[*]' = [1,2,3,4]
-- 或
'{"a": 1,"b": 2}'->'$.*' = {"a": 1,"b": 2}
```

- ** 深度查找。

```
'[1,2,[3,3,3],4]'->'$**[1]' = [2,3]
-- 或
'{"a": 1,"b": {"a": 2}}'->'$**.a' = [1,2]
```

JSON 比较与排序

JSON 类型支持 =、<> 、> 、>= 、< 、<= 比较运算符。

1. BOOL
2. ARRAY
3. OBJECT
4. STRING
5. STRING
6. NUMBER
7. NULL

相同类型按照以下排序规则：

- BOOL

  false 小于 true。

- ARRAY

  由第一个有差异的元素决定。该位置较小的数组首先排序。如果较短的数组的所有值都等于长数组中的的对应值，则较短数组首先排序。

```
[] < ["a"] < ["ab"] < ["ab", "cd", "ef"] < ["ab", "ef"]
```

- OBJECT

  由第一个有差异的键值对决定。键小优先排序，键一样，值小优先排序。如果较短的对象的所有键值对都包含在长对象中，则较短的对象优先排序。存在相同的键值，则相等。

```
{} < {"a": 1} < {"a": 2} < {"ab": 1} < {"b": 1} < {"b": 1, "c"
   : 1}

{"a": 1, "b": 2} = {"b": 2, "a": 1}
```

- STRING

  按照字典序排序。

- NUMBER

  按照大小排序。

JOSN 存储类型

JSON 类型必须是符合 JSON 格式的字符串类型。

示例：

```
-- create table
create table t_json(c_id int primary key, c_json json);

-- insert
insert into t_json values(1, '[1,2,3]')(2, '{"中文key": "中文value
   "}');

-- select
select * from t_json;
C_ID|C_JSON                |
----+--------------------+
1|[1, 2, 3]              |
2|{"中文key": "中文value"}|

-- update
update t_json set c_json='{"key": "value"}' where c_id = 1;

-- select
select * from t_json;
C_ID|C_JSON                |
----+--------------------+
1|{"key": "value"}       |
2|{"中文key": "中文value"}|

-- delete
delete from t_json where c_id = 1;

-- select
select * from t_json;
C_ID|C_JSON                |
----+--------------------+
2|{"中文key": "中文value"}|
```

数据类型特性表

数据类型	长度	说明	Java 数据类型	长度	封装器类
JSON	2GB	存储 JSON 数据格式字符串	Java.sql.String	变长	JSON

2.8 BIT 数据类型

BIT（位）数据类型用于表示精确到位的二进制数据，分为定长与变长两种。

定长位类型

语法格式：

```
BIT[(SIZE)]
```

定长位类型位数固定。

- 指定位数 SIZE 时，其取值范围为 [1,60000]。
- 未指定位数 SIZE 时，默认为 1 位。

变长位类型

语法格式：

```
BIT VARYING[(SIZE)]
 或
VARBIT[(SIZE)]
```

变长位类型位数可变。

- 指定位数 SIZE 时，其取值范围为 [1,60000]。
- 未指定位数 SIZE 时，默认为不限制，最大值为 60000。

类型转换

转换方式取决于源数据位数与目标数据所需位数大小关系。

- 源数据位数等于目标数据所需位数

来源 | 目标	BIT	VARBIT	BINARY	CHAR
BIT	直接转换	直接转换	直接转换	▪ MySQL 兼容模式：视为字符串二进制，直接转换 ▪ 其他兼容模式：按位字符串转换，直接转换
接下页				

来源 | 目标	BIT	VARBIT	BINARY	CHAR
VARBIT	直接转换	直接转换	直接转换	▪ MySQL 兼容模式：视为字符串二进制，直接转换 ▪ 其他兼容模式：按位字符串转换，直接转换
BINARY	直接转换	直接转换	-	-
CHAR	▪ MySQL 兼容模式：视为字符串二进制，直接转换 ▪ 其他兼容模式：按位字符串解析，直接转换	▪ MySQL 兼容模式：视为字符串二进制，直接转换 ▪ 其他兼容模式：按位字符串解析，直接转换	-	-

- 源数据位数小于目标数据所需位数

来源 | 目标	BIT	VARBIT	BINARY	CHAR
BIT	右侧补 0 值	直接转换	右侧补 0 值	▪ MySQL 兼容模式：视为字符串二进制，直接转换 ▪ 其他兼容模式：按位字符串转换，右侧补空格
VARBIT	右侧补 0 值	直接转换	右侧补 0 值	▪ MySQL 兼容模式：视为字符串二进制，直接转换 ▪ 其他兼容模式：按位字符串转换，右侧补空格
BINARY	左侧补 0 值	直接转换	-	-

接下页

来源 | 目标	BIT	VARBIT	BINARY	CHAR
CHAR	▪ MySQL 兼容模式：视为字符串二进制，左侧补 0 值 ▪ 其他兼容模式：按位字符串解析，右侧补 0 值	▪ MySQL 兼容模式：视为字符串二进制，直接转换 ▪ 其他兼容模式：按位字符串解析，直接转换	-	-

- 源数据位数大于目标数据所需位数

来源 | 目标	BIT	VARBIT	BINARY	CHAR
BIT	右侧截断	右侧截断	右侧截断	▪ MySQL 兼容模式：视为字符串二进制，右侧截断 ▪ 其他兼容模式：按位字符串转换，右侧截断

接下页

来源 | 目标	BIT	VARBIT	BINARY	CHAR
VARBIT	右侧截断	右侧截断	右侧截断	▪ MySQL 兼容模式：视为字符串二进制，右侧截断 ▪ 其他兼容模式：按位字符串转换，右侧截断
BINARY	报错超长	报错超长	-	-
CHAR	▪ MySQL 兼容模式：报错超长 ▪ 其他兼容模式：按位字符串解析，右侧截断	▪ MySQL 兼容模式：报错超长 ▪ 其他兼容模式：按位字符串解析，右侧截断	-	-

字面量语法

位类型字面量语法如下（以位数为 3 的位数据 101 为例）：

```
b'101'`
 或
B'101'
```

位运算符

运算表达式	说明
VARBIT & VARBIT -> VARBIT	按位与
VARBIT | VARBIT -> VARBIT	按位与
VARBIT ^ VARBIT -> VARBIT	按位异或
~ VARBIT -> VARBIT	按位取反
VARBIT << INTEGER -> VARBIT	左移位
VARBIT << INTEGER -> VARBIT	左移位
VARBIT >> INTEGER -> VARBIT	右移位

⚠ 注意

以上表达式中任意操作数为 NULL，结果为 NULL。
当位运算操作数长度不等时，将右对齐，左侧补 0 值，再进行位比较。

重要变更

- B’11’ 字面量语法此前为 BINARY 类型，内容为十六进制字符串，变更为 BIT 类型，内容为二进制字符串。
- 新增 << 与 >> 移位运算符，由于与存储过程 <<ColID>> 标签语法存在冲突，不允许连续的移位运算表达式，如“SELECT B’1’>>1>>1”，将报语法错误，但可通过括号实现：“SELECT (B’1’>>1)>>1”。

2.9 XML 数据类型

XML 简介

XML（eXtensible Markup Language）的全称是可扩展标记语言，是一种基于文本的标记语言。XML 是 W3C（World Wide Web Consortium）的推荐标准，实际上已经成为了 Web 上数据交换的标准。

XML 主要用于 Web 开发、电子商务、移动应用、配置文件，现在也被用于数据库中。

XML 数据可存储在文件、内存和数据库中。

XML 在数据库中的实现

XML 在虚谷数据库中有 XML 和 XMLTYPE 两种类型名，但在底层实现只有 XML 类型，采用 Blob 存储，支持最大 2GB 文本。XML 数据类型是数据库自定义基础类型，支持在表、视图中创建 XML 类型的列；也支持创建 XML 类型的常量和变量。

XML 相关系统函数的详细信息请参见《系统函数参考指南》的 XML 数据类型函数章节。

XML 依赖库

- xerces-c-3.2.2：对 XML 作基础处理，更多详细信息请参见Xerces-C 网站。
- XQilla-2.3.4：对 XML 数据进行操作，更多详细信息请参见XQilla 网站。

XML 结构

- XML 声明：XML 声明是一个可选的元素，用于指定 XML 的版本和字符编码。语法如下：

```
<?xml version="1.0" encoding="UTF-8"?>
```

- 标签：XML 使用标签来描述数据的结构。标签由开始标签和结束标签组成，并且可以包含内容。语法如下：

```
<tag>content</tag>
```

- 属性：XML 标签可以具有属性，用于提供有关标签的附加信息。属性由名称和值组成，并且位于开始标签中。语法如下：

```
<tag attribute="value">content</tag>
```

- 注释：XML 可以包含注释，用于提供与数据相关的额外信息。语法如下：

```
<!-- This is a comment -->
```

XML 数据格式

XML 数据以字符串格式保存，其节点值可以根据特定的格式要求转换为其他数据类型的值。

- 字符类型

```
<string>abc</string>
```

- number 类型

```
<int>11</int>
<double>11.1</double>
```

- bool 类型

```
<bool>true</bool>
<bool>false</bool>
```

- null

```
null
```

XPath

虚谷支持使用 XPath 在 XML 文档中对元素和属性进行遍历，XPath 主要用于 XML 相关的系统函数操作 XML 数据。

语法规则如下：

- 元素节点：表示 XML 或 HTML 文档中的元素。

  在 HTML 文档中，<body>、 、<p> 等都是元素节点。在 XPath 中，可以使用元素名称来选择元素节点，例如：//div 表示选择所有的 元素。

- 属性节点：表示 XML 或 HTML 文档中元素的属性。

  在 HTML 文档中，元素的 class、id、src 等属性都是属性节点。在 XPath 中，可以使用 @ 符号来选择属性节点，例如：//img/@src 表示选择所有 <img> 元素的 src 属性。

- 文本节点：表示 XML 或 HTML 文档中的文本内容。

  在 HTML 文档中，<p> 标签中的文本内容就是文本节点。在 XPath 中，可以使用 text() 函数来选择文本节点，例如：//p/text() 表示选择所有 <p> 元素中的文本内容。

- 命名空间节点：表示 XML 文档中的命名空间，命名空间是一种避免元素命名冲突的方法。

  在 XPath 中，可以使用 namespace 轴来选择命名空间节点，例如：//namespace::* 表示选择所有的命名空间节点。

- 处理指令节点：表示 XML 文档中的处理指令，处理指令是一种用来给处理器传递指令的机制。

  在 XPath 中，可以使用 processing-instruction() 函数来选择处理指令节点，例如：//processing-instruction('xml-stylesheet') 表示选择所有的 xml-stylesheet 处理指令节点。

- 注释节点：表示 XML 或 HTML 文档中的注释，注释是一种用来添加说明和备注的机制。

  在 XPath 中，可以使用 comment() 函数来选择注释节点，例如：//comment() 表示选择所有的注释节点。

- 文档节点：表示整个 XML 或 HTML 文档，文档节点也被称为根节点。

  在 XPath 中，可以使用/符号来选择文档节点，例如：/表示选择整个文档节点。

示例：

```
<bookstore>
```

```
<book category='fiction'>
<title>活着</title>
<author>余华</author>
<press>作家出版社</press>
<date>2012-8-1</date>
<page>191</page>
<price>20.00</price>
<staple>平装</staple>
<series>余华作品（2012版）</series>
<isbn>9787506365437</isbn>
</book>
<book category='non-fiction'>
<title>撒哈拉的故事</title>
<author>三毛</author>
<press>哈尔滨出版社</press>
<date>2003-8</date>
<page>217</page>
<price>15.80</price>
<staple>平装</staple>
<series>三毛全集（华文天下2003版）</series>
<isbn>9787806398791</isbn>
</book>
<book category='non-fiction'>
<title>明朝那些事儿（1-9）</title>
<author>当年明月</author>
<press>中国海关出版社</press>
<date>2009-4</date>
<page>2682</page>
<price>358.20</price>
<staple>精装16开</staple>
<series>明朝那些事儿（典藏本）</series>
<isbn>9787801656087</isbn>
</book>
</bookstore>
```

XPath 使用通配符示例：

通配符	描述	示例
*	匹配任何元素节点	//book/* 选取 <book> 元素下的任意子元素节点
@*	匹配任何属性节点	//book/@* 选取 <book> 元素上的任意属性节点，如 <book category='fiction'> 中的 category 属性
		接下页

通配符	描述	示例
node()	匹配任何类型的节点	//book/node() 选取 <book> 元素下的所有类型的子节点，包括元素节点、文本节点、注释节点等

XPath 使用谓词示例：

谓词	描述	示例
[position()=n]	选取位于指定位置的节点。n 是节点的位置（从 1 开始计数）	//book[position()=1] 选取第一个 <book> 元素
[last()=n]	选取位于指定位置的最后一个节点。n 是节点的位置（从 1 开始计数）	//book[last()=1] 选取最后一个 <book> 元素
[contains(string, substring)]	选取包含指定子字符串的节点。string 是节点的文本内容，substring 是要查找的子字符串	//book[contains(title, 'XML')] 选取标题中包含子字符串'XML' 的 <book> 元素
[starts-with(string, prefix)]	选取以指定前缀开始的节点。string 是节点的文本内容，prefix 是要匹配的前缀字符串	//book[starts-with(title, 'The')] 选取标题以'The' 开始的 <book> 元素
[text()=string]	选取文本内容完全匹配的节点。string 是要匹配的文本内容	//book[text()='Book Title'] 选取文本内容为'Book Title' 的 <book> 元素
[@category='non-fiction']	选取具有指定属性值的节点。category 是属性名称，non-fiction 是要匹配的值	//book[@category='non-fiction'] 选取具有属性 category 值为'non-fiction' 的 <book> 元素

XPath 使用路径表达式示例：

表达式	描述	示例
nodename	选取此节点的所有子节点	//bookstore/book 选取元素下所有 <book> 子元素
/	从根节点选取直接子节点	/bookstore 从根节点选取 <bookstore> 元素
//	从当前节点选取子孙节点	//book 选取所有 <book> 元素，无论它们在文档中的位置
.	选取当前节点	./title 选取当前节点的 <title> 子元素
..	选取当前节点的父节点	../price 选取当前节点的父节点的 <price> 子元素
@	选取属性	//book/@id 选取所有 <book> 元素的 id 属性

XQuery

XQuery 是 XML 数据的查询语言，类似于 SQL 是数据库的查询语言，用于查询 XML 数据。语法规则：

- XQuery 对大小写敏感。
- XQuery 的元素、属性以及变量必须是合法的 XML 名称。
- XQuery 字符串值可使用单引号或双引号。
- XQuery 变量由”$” 并跟随一个名称来进行定义，例如：\$bookstore。
- XQuery 注释被 (: 和:) 分割，例如：(: XQuery 注释:)。

更多 XQuery 的详细信息请参见XQuery 参考资料。

XML 应用示例

> **说明**
>
> XML 类型必须是符合 XML 格式的字符串类型。

```
-- create table
CREATE TABLE t_xml(c_id INT PRIMARY KEY, c_xml XML);

-- insert
INSERT INTO t_xml VALUES(1, '<num>1</num>')(2, '<str>ab</str>');

-- select
SELECT * FROM t_xml;
C_ID|C_XML                |
----+---------------------+
1|<num>1</num>             |
2|<str>ab</str>             |

-- update
UPDATE t_xml SET c_xml='<num>2</num>' WHERE c_id = 1;

-- select
SELECT * FROM t_xml;
C_ID|C_XML                |
----+---------------------+
1|<num>2</num>             |
2|<str>ab</str>             |

-- delete
DELETE FROM t_xml WHERE c_id = 1;

-- select
SELECT * FROM t_xml;
C_ID|C_XML                |
----+---------------------+
2|<str>ab</str>             |
```

2.10 ARRAY 数据类型

ARRAY 简介

ARRAY 类型允许将表的的列定义为可变长多维数组，数组允许在单个数据库字段中存储多个值，ARRAY 类型支持虚谷数据库基本数据类型。

由于 ARRAY 数组类型的特性，在利用数据库处理列表或数据集等场景中，ARRAY 类型可以发挥良好的作用，高效的实现场景下的特定需求。

ARRAY 类型在数据库中的实现

ARRAY 数据类型在虚谷数据库中有两种使用方式，分别为字符串类型的数组与 ARRAY 关键字数组，两种输入使用方式通过不同的逻辑统一转换为由 BLOB 大对象管理结构管理的 ArrayType，最大支持 2GB 输入文本。

ARRAY 数据类型是数据库内置的基础类型，支持在表、视图中创建 ARRAY 类型字段，也支持

创建 ARRAY 类型的常量与变量，两者之间的声明方式略有不同。

ARRAY 数据类型相关系统函数的详细信息请参见《系统函数参考指南》的 ARRAY 数据类型函数章节。

在非字段中声明 ARRAY

ARRAY 类型结构在非字段中有两种声明方式，分别是关键字声明和字符串声明。

- 关键字声明：

```
-- 一维数组
ARRAY[v1, v2, v3...];
-- 多维数组，以三维数组为例
ARRAY[[[v1, v2, v3...]]];
```

- 字符串声明：

```
-- 一维数组
'{v1, v2, v3}';
-- 多维数组，以三维数组为例
'{{{v1, v2, v3}}}';
```

在使用方式上，两者的区别有以下几点：

- 由 ARRAY 关键字声明的数组，维度是用成对的方括号进行标识的；字符串声明方式下的数组，是以大括号进行数组维度标识的。最大支持六维数组的声明与使用。
- 字符串类型的数组由于设计原因，一些本身存在大括号的数据类型暂时不支持以此方式使用。
- 字符串类型的数组的使用仅支持一些明确类型的场景，例如字段插入、更新、删除，并且不支持通过类型强制转换的方式转为数组类型；
- ARRAY 类型数组当成员为字符串类型时需要添加成对单引号进行包裹，声明为特定类型时，需要指明具体的数据类型，例如：

```
-- CHAR类型数组
SELECT ARRAY['aaa', 'bbb'];

-- DATE类型数组
SELECT ARRAY['2024-01-01'::DATE, '2024-12-31'::DATE];
```

在表字段中声明 ARRAY

ARRAY 类型结构在字段中可通过在数据类型后添加方括号 [] 的方式进行声明。

```
column_name type_name[]
```

- column_name：字段的名称。

- type_name：基础数据类型的名称。
- []：表示这是一个数组类型。可以根据需要添加多个方括号来声明多维数组。数组维度与类型名后的方括号对数量相同，例如 VARCHAR[][][] 表示三维数组。

ARRAY 类型支持多种基础类型作为成员数据，支持的基础类型如下表所示：

基础数据类型分类	类型名称
数值数据类型	• TINYINT • SMALLINT • INTEGER • BIGINT • NUMERIC • FLOAT • DOUBLE
字符数据类型	• CHAR • VARCHAR
时间数据类型	• DATE • TIME • TIME WITH TIME ZONE • DATETIME • DATETIME WITH TIME ZONE
接下页	

基础数据类型分类	类型名称
时间间隔类型	• INTERVAL YEAR • INTERVAL MONTH • INTERVAL DAY • INTERVAL HOUR • INTERVAL MINUTE • INTERVAL SECOND • INTERVAL YEAR TO MONTH • INTERVAL DAY TO HOUR • INTERVAL DAY TO MINUTE • INTERVAL DAY TO SECOND • INTERVAL HOUR TO MINUTE • INTERVAL HOUR TO SECOND • INTERVAL MINUTE TO SECOND
大对象数据类型	• BLOB • CLOB
	接下页

基础数据类型分类	类型名称
基础几何类型	• POINT • LINE • LSEG • BOX • PATH • POLYGON • CIRCLE
XML 数据类型	XML

为了更好的说明 ARRAY 类型在字段上的用法，以数据库中建表为例：

- 示例 1

  创建一个名为 array_types 的表，包含一个 VARCHAR 类型的 name 列，一个 INTEGER 一维数组类型的 int_arr 列，以及一个 VARCHAR 二维数组类型的 var_arr_2 列。

```
CREATE TABLE array_types (
var VARCHAR,
int_arr INTEGER[],
var_arr_2 VARCHAR[][]
);
```

- 示例 2

  在声明数组类型时，方括号中可以添加指定大小的参数，但当前实现忽略了任何指定数组大小的操作，即添加参数的声明行为与未指定大小时相同。

```
-- 同 int_arr INTEGER[][]
CREATE TABLE array_params (
int_arr INTEGER[3][3]
);
```

- 示例 3

  若声明一维数组，还提供另外关键字的声明方式。

```
-- 添 加 大 小 参 数
```

```
int_arr INTEGER ARRAY[4]

-- [] 中 不 添 加 参 数 时 不 单 独 使 用 []
int_arr INTEGER ARRAY
```

ARRAY 类型输入

ARRAY 类型结构支持两种输入方式，分别是使用关键字 ARRAY 和使用字符串。

以输入二维数组为例，更高维度的插入同样遵循此原则。

- 关键字 ARRAY 输入：

```
INSERT INTO array_types VALUES('int', '{1, 2, 3}', '
   {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}');
```

- 字符串输入：

```
INSERT INTO array_types VALUES('int', ARRAY[1, 2, 3], ARRAY
   [[1, 2, 3], [4, 5, 6], [7, 8, 9]]);
```

ARRAY 类型查询

本小节以一维数组表和三维数组表为例进行说明：

```
-- 创 建 一 维 数 组 表 并 插 入 数 据
SQL> CREATE TABLE test_arr (a int[]);
SQL> INSERT INTO test_arr VALUES ('{1, 2, 3}');

-- 创 建 三 维 数 组 表 并 插 入 2x2x3 的 三 维 数 组
SQL> CREATE TABLE test_arr_3 (a int[][][]);
SQL> INSERT INTO test_arr_3 VALUES ('
   {{{1, 2, 3}, {4, 5, 6}}, {{7, 8, 9}, {10, 11, 12}}}');
```

- 查询整组字段

  直接查询字段名即可查询整组字段信息。

  - 查询一维数组：

```
SQL> SELECT a FROM test_arr;

A |
-----------------------------------------------------------
{1,2,3}|
```

  - 查询多维数组：

```
SQL> SELECT a FROM test_arr_3;

A |
-----------------------------------------------------------
{{{1,2,3},{4,5,6}},{{7,8,9},{10,11,12}}}|
```

- 查询数组某个下标代表的元素

  通过“字段名 [数组下标]”的方式进行查询。

  - 查询一维数组的 1 号元素：

```
SQL> SELECT a[1] FROM test_arr;

EXPR1 |
------------------------------------------------------------
1 |
```

  - 查询多维数组，a[1][2][3] 访问的是第一个维度的第一个元素（即第一个 2x3 的二维数组 [[1, 2, 3], [4, 5, 6]]），第二个维度的第二个元素（即第二行 [4, 5, 6]），第三个维度的第三个元素（即 6）。

```
SQL> SELECT a[1][2][3] FROM test_arr_3;

EXPR1 |
------------------------------------------------------------
6 |
```

> **说明**
>
> 更高维的数组元素查询方式与之类似，方括号的数量需要与表中数据的维度匹配。

- 查询数组字段中的切片数据

  以切片“字段名 [起始下标: 结束下标]”的方式查询数组字段中的部分数据。

  - 查询一维数组中第 1 到第 2 个元素：

```
SQL> SELECT a[1:2] FROM test_arr;

EXPR1 |
------------------------------------------------------------
{1,2}|
```

  - 查询多维数组，a[1:2][1:2][1:2] 表示查询第一个维度的第 1 到第 2 个元素、第二个维度的第 1 到第 2 个元素以及第三个维度的第 1 到第 2 个元素。

```
SQL> SELECT a[1:2][1:2][1:2] FROM test_arr_3;

EXPR1 |
------------------------------------------------------------
{{{1,2},{4,5}},{{7,8},{10,11}}}|
```

> **说明**
>
> 多维数组的切片查询也应当与表中字段的维度匹配。当切片查询的结束下标超过当前字段中数组的个数时，数据库将以最大的元素个数修正结束下标。

ARRAY 类型更新

在更新指定下标位置的元素需要在字段名后添加想要修改元素的位置下标。

- 一维数组。

```
UPDATE test_arr SET a[1] = 4;
```

- 多维数组。

```
UPDATE test_arr_3 SET a[1][1][1] = 4;
```

- 使用切片更新。

```
UPDATE test_arr_3 SET a[1:1][1:1][1:2] = '{{{4, 5}}}';
```

通过以上方式，可以对指定位置的元素进行更新。

2.11 基础几何类型

概述

几何数据类型表示二维的空间物体，支持的几何类型如下：

类型名称	宽度（字节）	定义	数据格式
POINT	16	平面上的点	(x, y)
LINE	32	无限长的线	A, B, C
LSEG	32	有限线段	((x1, y1), (x2, y2))
BOX	32	矩形框	((x1, y1), (x2, y2))
PATH	16 + 16n	封闭路径（类似于多边形）	((x1, y1), ...)

接下页

类型名称	宽度（字节）	定义	数据格式
PATH	16 + 16n	开放路径	[(x1, y1), ...]
POLYGON	40 + 16n	多边形（类似于封闭路径）	((x1, y1), ...)
CIRCLE	24	圆	<(x, y), r>（中心点和半径）

点 (POINT)

点是几何类型的基本二维构造块。用下面的语法描述“POINT”类型的值：

```
( x , y )
x , y
```

线 (LINE)

线由线性方程“Ax + By + C = 0”表示，其中 A 和 B 都不为零。类型“LINE”的值采用以下形式输入和输出：

```
{ A, B, C }
```

另外，还可以用下列任一形式输入：

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
( x1 , y1 ) , ( x2 , y2 )
x1 , y1   ,   x2 , y2
```

其中 (x1,y1) 和 (x2,y2) 是线上不同的两点。

线段 (LSEG)

线段用一对线段的端点来表示。“LSEG”类型的值用下面的语法声明：

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
( x1 , y1 ) , ( x2 , y2 )
x1 , y1   ,   x2 , y2
```

其中 (x1,y1) 和 (x2,y2) 是线段的端点。

线段使用第一种语法输出。

方框 (BOX)

方框用其对角的点对表示。“BOX”类型的值使用下面的语法指定：

```
( ( x1 , y1 ) , ( x2 , y2 ) )
( x1 , y1 ) , ( x2 , y2 )
x1 , y1   ,   x2 , y2
```

其中 (x1,y1) 和 (x2,y2) 是方框的对角点。

方框使用第二种语法输出。

在输入时可以提供任意两个对角，但是值将根据需要被按顺序记录为右上角和左下角。

路径 (PATH)

路径由一系列连接的点组成。路径可能是开放的，也就是认为列表中第一个点和最后一个点没有被连接起来；也可能是封闭的，这时认为第一个和最后一个点被连接起来。

“PATH”类型的值用下面的语法声明：

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
( x1 , y1 ) , ... , ( xn , yn )
( x1 , y1   , ... ,   xn , yn )
x1 , y1   , ... ,   xn , yn
```

其中的点是组成路径的线段的端点。方括弧（[]）表示一个开放的路径，圆括弧（()）表示一个封闭的路径。如第三种到第五种语法所示，当最外面的圆括号被忽略时，路径将被假定为封闭。

路径的输出使用第一种或第二种语法。

多边形 (POLYGON)

多边形由一系列点代表（多边形的顶点）。多边形和封闭路径很像，但是存储方式不一样而且有自己的一套支持例程。

“POLYGON”类型的值用下列语法声明：

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
( x1 , y1 ) , ... , ( xn , yn )
( x1 , y1   , ... ,   xn , yn )
x1 , y1   , ... ,   xn , yn
```

其中的点是组成多边形边界的线段的端点。

多边形的输出使用第一种语法。

圆 (CIRCLE)

圆由一个圆心和一个半径代表。“CIRCLE”类型的值用下面的语法指定：

```
< ( x , y ) , r >
( ( x , y ) , r )
( x , y ) , r
x , y   , r
```

其中 (x,y) 是圆心，而 r 是圆的半径。

圆的输出用第一种语法。

示例

```
-- 创建表
CREATE TABLE t_geom(c_id INT PRIMARY KEY, c_pt POINT, c_le LINE,
   c_lg LSEG, c_bx BOX, c_ce CIRCLE, c_ph PATH, c_pn POLYGON);

-- 插入数据，点、线、线段、方框、圆、路径和多边形
INSERT INTO t_geom VALUES(1, '(1,1)','(1,1),(2,2)','(1,1),(3,3)','
   (1,1),(4,4)','<(1,2),3>','(1,1),(2,3),(3,1)','(1,1),(2,3),(3,1)
   ,(1,1)');

-- 查询数据
SELECT * FROM t_geom;
C_ID|C_PT      |C_LE           |C_LG                   |C_BX
                   |C_CE            |C_PH                                |
   C_PN                                    |
----+---------+---------------+-----------------------+----------
---------+---------------+-------------------------------+----
-----------------------------------+
1|(1.0,1.0)|{1.0,-1.0,0.0}|[(1.0,1.0),(3.0,3.0)]|(4.0,4.0)
   ,(1.0,1.0)|<(1.0,2.0),3.0>|((1.0,1.0),(2.0,3.0),(3.0,1.0))
   |((1.0,1.0),(2.0,3.0),(3.0,1.0),(1.0,1.0))|

-- 更新点的数据为（0,1）
UPDATE t_geom SET c_pt='(0,1)' WHERE c_id = 1;

-- 查询更新后的数据
SELECT * FROM t_geom;
C_ID|C_PT      |C_LE           |C_LG                   |C_BX
                   |C_CE            |C_PH                                |
   C_PN                                    |
----+---------+---------------+-----------------------+----------
---------+---------------+-------------------------------+----
-----------------------------------+
1|(0.0,1.0)|{1.0,-1.0,0.0}|[(1.0,1.0),(3.0,3.0)]|(4.0,4.0)
   ,(1.0,1.0)|<(1.0,2.0),3.0>|((1.0,1.0),(2.0,3.0),(3.0,1.0))
   |((1.0,1.0),(2.0,3.0),(3.0,1.0),(1.0,1.0))|

-- 删除c_id = 1的数据
DELETE FROM t_geom WHERE c_id = 1;

-- 查询删除后的数据
SELECT * FROM t_geom;
C_ID|C_PT|C_LE|C_LG|C_BX|C_CE|C_PH|C_PN|
----+----+----+----+----+----+----+----+
```

2.12 用户自定义类型（UDT）

2.12.1 概述

分类

用户自定义类型（User Defined Type，UDT）可分为：结构类型、数组类型、嵌套表类型，三者创建或声明类型时的区别如下表。

分类	成员/元素	父类	构造函数	指定容量	方法
结构类型 (OBJECT)	有	有	有	否	有
数组类型（VARRAY）	有	无	有	是	无
嵌套表类型（TABLE）	有	无	有	否	无

> ⚠ 注意
>
> UDT 不支持大对象类型作为成员或元素类型。

构造函数

在创建 UDT 时，为每个 UDT 隐式定义了一个构造函数。构造函数是系统提供的函数，用于在 SQL 语句或 PL/SQL 代码中构造类型值的实例。构造函数的名称是 UDT 的名称，参数是用户自定义类型的所有成员变量，返回值是该用户自定义类型。

> ⚠ 注意
>
> OBJECT 的构造函数支持重载，可以通过 CONSTRUCTOR 关键字重新自定义零个或多个不同参数的构造函数，且自定义的构造函数优先级高于默认构造函数。

继承子类和超类

OBJECT 类型支持继承关系，被继承的类称为超类、基类或父类。子类在继承付给的属性和方法时，可以扩展自己的属性和方法。

比较和排序

- 仅支持同一个模式下的同个 UDT 对象 ID 的 UDT 数据比较。
- 仅支持 UDT 数据等值和不等值比较。

- 支持 UDT 数据“IS NULL”或者“IS NOT MULL”比较。
- PL/SQL 支持表中数据与相同类型的数据等值和不等值比较。
- PL/SQL 支持具有相同成员或元素类型的临时类型的数据比较。
- 不支持对 UDT 数据的排序，但若成员类型为基础类型，则支持成员值的排序。

赋值

- SQL 支持初始化赋值、构造函数赋值和复制赋值。
- PL/SQL 支持初始化赋值、构造函数赋值、成员赋值和复制赋值。

成员/元素

- 构造类型的数据：使用点号. 访问对象的成员。例如有一个构造类型 OBJ，它有两个成员 ID、NAME，我们可以用“OBJ.ID”和“OBJ.NAME”来进行访问。
- 集合类型的数据：使用索引访问集合中的元素。例如用“varr(1)”访问 varr 数组的第一个元素。

依赖

对象类型、可变数组类型和嵌套表类型作为独立对象类型被使用时，需要添加该类型与使用者的依赖关系。

转换函数

不支持 UDT 和其他类型的相互转换与强制转换。

2.12.2 结构类型（OBJECT）

类型介绍

数据库使用 OBJECT 表示结构类型，允许直接表示 E-R 设计中的复合属性，其设计类似于面向对象语言（eg.Java）支持父类继承，包括结构类型规范和结构类型体两个部分，是在其他数据类型基础上建立的，可以包含多个属性（元素）和多种方法（成员函数），这些属性和方法构成了结构化的数据单元。成员函数的类型有构造函数（cosntructor）、静态函数（static）和普通成员函数（member）。

⚠ 注意

- 重载：结构类型支持方法重载，即可以有多个同名但参数不同的方法。继承：结构类型支持继承，但子类不能重写父类的成员函数（同名同参）。

模式级别创建的结构类型可以持久化存储在数据库中，记载到系统表 SYS_TYPES 和 SYS_OBJECT 中。OBEJCT 类型可以当成表的字段类型、变量类型、参数类型，返回值类型等在其他对象中调用。

语法格式

创建 OBJECT 类型：

```
CREATE [ OR REPLACE ] TYPE udt_name AS OBJECT(
    [ variable_define ],
    [ function_declare ],
    [ procedure_declare ]
);

-- 可选，当且仅当OBJECT中有声明过程或函数时，才需要该部分
CREATE [ OR REPLACE ] TYPE BODY udt_name
{ IS | AS }
    [ function_define ],
    [ procedure_define ]
END;
```

参数说明

- variable_define：变量（或称为属性）定义，其中变量可以是基础类型或 UDT 数据类型。
- function_declare：函数声明。函数可分为如下三类：
  - 构造函数: 形式为“CONSTRUCTOR udt_name(arguments)”；其中函数名和 Object 名相同，参数的个数和类型要和本 Object 中的变量相对应。
  - 静态函数：形式为“STATIC function_name([arguments]) RETURN ret_type”；其中 arguments 为函数的参数，可有一个、多个或无参数，每个参数可以用可选的关键词“（IN | OUT | IN OUT）”修饰；“ret_type”为函数的返回值。
  - 普通成员函数：形式为“MEMBER function_name([arguments]) RETURN ret_type”，其中“arguments”为函数的参数，可有一个、多个或无参数，每个参数可以用可选的关键词“（IN | OUT | IN OUT）”修饰；“ret_type”为函数的返回值。
- procedure_declare：过程声明。过程可分为如下两类：
  - 静态过程：形式为“STATIC procedure_name([arguments])”；其中“arguments”为过程的参数，可有一个、多个或无参数，每个参数可以用可选的关键词“（IN | OUT | IN OUT）”修饰。
  - 普通成员过程：形式为“MEMBER procedure_name([arguments])”；其中“arguments”为过程的参数，可有一个、多个或无参数，每个参数可以用可选的关键

词“（IN | OUT | IN OUT）”修饰。

- function_define：函数的具体定义。
- procedure_define：过程的具体定义。

示例

1. 创建用户定义类型（UDT）。创建一个对象类型 udt_obj_type，它包含四个属性：n、class、type 和 dt。

```
--创建类型：
CREATE OR REPLACE TYPE udt_obj_type AS OBJECT(n NUMERIC,class
   VARCHAR2,type VARCHAR, dt DATE);
```

2. 创建表。创建一个表 obj_tab，其中包含一个列 udt_obj，该列的类型是刚刚创建的 udt_obj_type。

```
--创建表：
CREATE TABLE obj_tab(id INT,state VARCHAR2,type VARCHAR,udt_obj
    UDT_OBJ_TYPE);
```

3. 插入数据。
   - 使用构造函数插入数据。可以直接使用构造函数在 INSERT 语句中插入数据。

```
--  字段类型
INSERT INTO obj_tab(id,state,type,udt_obj)  VALUES(1,'构造函
   数插入', 'udt_obj_type',udt_obj_type(1.0,'一层','
   udt_obj_type','2021-08-24 00:00:00'));
```

   - 使用变量插入数据。在 PL/SQL 块中声明一个变量，然后将该变量插入到表中。

```
--  变量类型
DECLARE
udt_obj UDT_OBJ_TYPE;
BEGIN
udt_obj := UDT_OBJ_TYPE(1.0,'一层','udt_obj_type','
   2021-08-24 00:00:00');
INSERT INTO obj_tab(id, state, type, udt_obj)VALUES(1, '
   plsql通过udt变量插入数据', 'udt_obj_type', udt_obj);
END;
```

4. 查询数据。
   - 可以查询表中的 udt_obj 列，查看整个 UDT 对象。

```
--查询UDT整体：
SELECT id, udt_obj FROM obj_tab;

ID | UDT_OBJ |
-----------------------------------------------------------
1 | [1,一层,udt_obj_type,2021-08-24 00:00:00]|
1 | [1,一层,udt_obj_type,2021-08-24 00:00:00]|
```

- 直接在 SQL 语句中创建 UDT 对象并查看其内容。

```
SELECT udt_obj_type(1.0,'一层','udt_obj_type','
   2021-08-24 00:00:00');

EXPR1 |
------------------------------------------------------------
[1,一层,udt_obj_type,2021-08-24 00:00:00]|
```

- 查询 UDT 的部分属性。

```
-- 查询udt_obj列中的特定属性
SELECT udt_obj.n, udt_obj.type, udt_obj.dt FROM obj_tab;

EXPR1 | EXPR2 | EXPR3 |
------------------------------------------------------------
1| udt_obj_type| 2021-08-24 AD |
1| udt_obj_type| 2021-08-24 AD |
```

2.12.3 数组类型（VARRAY）

类型介绍

数组类型是在 SQL 标准 1999 中增加的，表示有序的相同数据类型元素的集合。数据库中使用 VARRY OF 关键字可以创建或声明数组类型，数组内的每个元素均有一个索引下标，代表元素在数组中的位置，且可以根据下标直接访问数组元素。

VARRAY 可变数组类型是一个连续的相同数据类型元素的集合，成员最多包含 65535 个，成员类型可以是基础类型也可以是用户自定义类型。

语法格式

创建 VARRAY 类型：

```
CREATE OR REPLACE TYPE udt_name IS VARRAY(size) OF type_x;
```

参数说明

- type_name: 定义的类型名称。
- size: VARRAY 数组的指定容量，创建时必须指定。
- type_x: 元素的数据类型，可以是基本数据类型（如 NUMBER, VARCHAR2 等）或其他用户自定义类型。

示例

创建用户定义类型（UDT），使用这些类型创建表，并进行插入和查询操作。

1. 创建 VARRAY 类型，定义了一个对象类型 udt_obj_type 和一个可变数组类型 udt_varryofobj_type。

```
CREATE OR REPLACE TYPE udt_varryofobj_type IS VARRAY(10) OF
   udt_obj_type;
```

2. 创建表。创建了一个表 varry_tab，其中包含一个列 udt_varryofobj，其数据类型为 udt_varryofobj_type。

```
CREATE TABLE varry_tab(id INT, state VARCHAR2, type VARCHAR,
   udt_varryofobj UDT_VARRYOFOBJ_TYPE);
```

3. 插入数据。

- 使用构造函数插入数据。可以直接使用构造函数在 INSERT 语句中插入数据。

```
--字 段 类 型
INSERT INTO varry_tab (id, state, type, udt_varryofobj)
VALUES (
  2,
  '构 造 函 数 插 入',
  'udt_varryofobj_type',
  udt_varryofobj_type(
    udt_obj_type(1.0, '2层', 'udt_obj_type', TO_DATE('
       2021-08-24 00:00:00', 'YYYY-MM-DD HH24:MI:SS')),
    udt_obj_type(1.0, '2层', 'udt_obj_type', TO_DATE('
       2021-08-24 00:00:00', 'YYYY-MM-DD HH24:MI:SS'))
  )
);
```

- 使用变量插入数据。在 PL/SQL 块中声明一个变量，然后将该变量插入到表中。

```
--  变 量 类 型
DECLARE
    udt_varryofobj udt_varryofobj_type;
BEGIN
    udt_varryofobj := udt_varryofobj_type(
    udt_obj_type(1.0, '2层', 'udt_obj_type', TO_DATE('
       2021-08-24 00:00:00', 'YYYY-MM-DD HH24:MI:SS')),
    udt_obj_type(1.0, '2层', 'udt_obj_type', TO_DATE('
       2021-08-24 00:00:00', 'YYYY-MM-DD HH24:MI:SS'))
    );
  INSERT INTO varry_tab (id, state, type, udt_varryofobj)
  VALUES (2, 'plsql通 过udt变 量 插 入 数 据', '
   udt_varryofobj_type', udt_varryofobj);
END;
/
```

4. 查询数据。

- 可以查询表中的 udt_varryofobj 列，查看整个 UDT 对象。

```
SELECT id, udt_varryofobj FROM varry_tab;

ID | UDT_VARRYOFOBJ |
-----------------------------------------------------------
```

```
2 | [[1,2层,udt_obj_type,2021-08-24 00:00:00],[1,2层,
   udt_obj_type,2021-08-24 00:00:00]]|
2 | [[1,2层,udt_obj_type,2021-08-24 00:00:00],[1,2层,
   udt_obj_type,2021-08-24 00:00:00]]
```

- 直接在 SQL 语句中创建 UDT 对象并查看其内容。

```
SELECT udt_varryofobj_type(
udt_obj_type(
1.0,'2层','udt_obj_type','2021-08-24 00:00:00') ,
udt_obj_type(
1.0,'2层','udt_obj_type','2021-08-24 00:00:00') )
FROM DUAL;

EXPR1 |
------------------------------------------------------------
[[1,2层,udt_obj_type,2021-08-24 00:00:00],[1,2层,
   udt_obj_type,2021-08-24 00:00:00]]|
```

- 查询 UDT 的部分属性。

```
SELECT udt_varryofobj(1) ,udt_varryofobj(2) FROM varry_tab;

EXPR1 | EXPR2 |
------------------------------------------------------------
[1,2层,udt_obj_type,2021-08-24 00:00:00]| [1,2层,
   udt_obj_type,2021-08-24 00:00:00]|
[1,2层,udt_obj_type,2021-08-24 00:00:00]| [1,2层,
   udt_obj_type,2021-08-24 00:00:00]|
```

2.12.4 嵌套表类型（TABLE）

类型介绍

嵌套表类型（SQL 中称为多重集合类型）是在 SQL2003 中增加的，表示无序的相同类型数组元素的集合。数据库中使用 TABLE OF 关键字可以创建或声明嵌套表类型，嵌套表内的每个元素均有一个索引下标，代表元素在数组中的位置，且可以根据下标直接访问嵌套表元素。不同数组的是，嵌套表未经删减，元素数据位置是连续的，若删除中间位置的数据后，数据变成了离散的。

一般而言，E-R 模式中的多值属性可以映射到 SQL 中的以多重集合为值的属性，若顺序是重要的，则使用 SQL 数组来代替多重集合。

语法格式

创建 TABLE 类型：

```
CREATE OR REPLACE TYPE udt_name AS TABLE OF type_x;
```

查询表中 TABLE 数据：

```
-- 查询table整体
SELECT c_1, udt_col FROM obj_tab;
SELECT udt_name(...) FROM dual;

-- 查询table的部分属性
SELECT udt_name(i) FROM udt_tab;
```

参数说明

- udt_name: 定义的类型名称。
- type_x: 元素的数据类型，可以是基本数据类型（如 NUMBER, VARCHAR2 等）或其他用户自定义类型。
- c_1：普通列。
- udt_col：嵌套表列，定义的用户定义类型 udt_name 的实例。
- i：嵌套表中的某列。

示例

创建用户定义类型（UDT），使用这些类型创建表，并进行插入和查询操作。

1. 创建 VARRAY 类型，定义了一个对象类型 udt_obj_type 和一个可变数组类型 udt_ttabofobj_type。

```
CREATE OR REPLACE TYPE udt_ttabofobj_type AS TABLE OF
   udt_obj_type;
```

2. 创建表。创建了一个表 ttab_tab，其中包含一个列 udt_ttabofobj，其数据类型为 udt_ttabofobj_type。

```
CREATE TABLE ttab_tab(id INT,
state VARCHAR2,
type VARCHAR,
udt_ttabofobj udt_ttabofobj_type);
```

3. 插入数据。
   - 使用构造函数插入数据。可以直接使用构造函数在 INSERT 语句中插入数据。

```
--字段类型
INSERT INTO ttab_tab (id, state, type, udt_ttabofobj)
VALUES (
  2,
  '构造函数插入',
  'udt_ttabofobj_type',
  udt_ttabofobj_type(
    udt_obj_type(1.0, '2层', 'udt_obj_type', '
      2021-08-24 00:00:00'),
```

```
    udt_obj_type(1.0, '2层', 'udt_obj_type', '
       2021-08-24 00:00:00')
  )
);
```

- 使用变量插入数据。在 PL/SQL 块中声明一个变量，然后将该变量插入到表中。

```
-- 变量类型
DECLARE
    udt_ttabofobj udt_ttabofobj_type;
BEGIN
    udt_ttabofobj := udt_ttabofobj_type(
    udt_obj_type(1.0, '2层', 'udt_obj_type', '
       2021-08-24 00:00:00', 'YYYY-MM-DD HH24:MI:SS'),
    udt_obj_type(1.0, '2层', 'udt_obj_type', '
       2021-08-24 00:00:00', 'YYYY-MM-DD HH24:MI:SS')
  );
  INSERT INTO varry_tab (id, state, type, udt_ttabofobj)
  VALUES (2, 'plsql通过udt变量插入数据', 'udt_ttabofobj_type
   ', udt_ttabofobj);
END;
/
```

4. 查询数据。利用 table() 方法把 table 类型的数据当成表值调用，把它当成表查询包含的成员值。

- 可以查询表中的 udt_varryofobj 列，查看整个 UDT 对象。

```
SELECT id, udt_varryofobj FROM ttab_tab;

ID | UDT_TTABOFOBJ |
-------------------------------------------------------------
2 | [[1,2层,udt_obj_type,2021-08-24 00:00:00],[1,2层,
   udt_obj_type,2021-08-24 00:00:00]]|
2 | [[1,2层,udt_obj_type,2021-08-24 00:00:00],[1,2层,
   udt_obj_type,2021-08-24 00:00:00]]|
```

- 直接在 SQL 语句中创建 UDT 对象并查看其内容。

```
SELECT udt_ttabofobj_type(
udt_obj_type(
1.0,'2层','udt_obj_type','2021-08-24 00:00:00') ,
udt_obj_type(
1.0,'2层','udt_obj_type','2021-08-24 00:00:00') )
FROM DUAL;

EXPR1 |
-------------------------------------------------------------
[[1,2层,udt_obj_type,2021-08-24 00:00:00],[1,2层,
   udt_obj_type,2021-08-24 00:00:00]]|
```

- 查询 UDT 的部分属性。

```
SELECT udt_ttabofobj(1),udt_ttabofobj(2) FROM ttab_tab;

EXPR1 | EXPR2 |
-------------------------------------------------------------
[1,2层,udt_obj_type,2021-08-24 00:00:00]| [1,2层,
   udt_obj_type,2021-08-24 00:00:00]|
[1,2层,udt_obj_type,2021-08-24 00:00:00]| [1,2层,
   udt_obj_type,2021-08-24 00:00:00]|
```

2.13 记录类型和集合类型

2.13.1 记录类型

类型介绍

记录类型是由多种不同数据类型的元素组合成的数据类型，使用点号“.”访问记录的每个字段，例如用“rec.id”访问 rec 中名为 id 的成员。通常在 PL/SQL 中使用。

> **说明**
>
> 不支持为 record 数据添加新成员或删除已有成员。

语法格式

声明一个记录类型，然后声明一个该类型的变量并初始化为 EMPTY：

```
TYPE rec_name IS RECORD(c_1 type_1,
...,
c_n type_n);

var_name rec_name;
```

用构造函数初始化：

```
var_name rec_name:=rec_name(c_1_value, ..., c_n_value);
```

声明一个变量，类型与表字段组合类型关联并初始化为 EMPTY：

```
var_name table_name%ROWTYPE;
```

声明一个变量，类型与游标类型关联并初始化为 EMPTY：

```
var_name cursor_name%ROWTYPE;
```

参数说明

- rec_name：新定义的记录类型的名称。可以选择任何合法的标识符作为名称。
- c_1，...，c_n：记录中的各个字段（列）名称。

- type_1，...，type_n：各个字段的数据类型。这些数据类型可以是基础类型，也可以是用户自定义的数据类型。
- var_name: 基于 rec_name 类型的变量的名称。这个变量可以用来存储符合 rec_name 结构的数据。
- c_1_value, ..., c_n_value：记录中的各个字段（列）的值。
- table_name: 已经定义的表名称。
- cursor_name: 已经定义的游标名称。

示例

定义了一个记录类型 t_rec，并创建了一个该类型的变量 var_rec。然后初始化这个变量，并使用 DBMS_OUTPUT.PUT_LINE 输出变量的内容。

```
DECLARE
    -- 定义RECORD类型
    TYPE t_rec IS RECORD(
        id       NUMERIC,
        name     VARCHAR2,
        birthday DATE);
    -- 声明RECORD类型变量
    var_rec T_REC;
BEGIN
    -- 赋值
    var_rec := T_REC(10, 'David', '2001-02-03');
    -- 通过var_rec.[字段名]访问成员并输出
    DBMS_OUTPUT.PUT_LINE('Person Info: (' || var_rec.ID || ', ' ||
       var_rec.NAME || ', ' || var_rec.BIRTHDAY || ')');
END;
/
-- 输出
Person Info: (10, David, 2001-02-03 00:00:00)
```

2.13.2 集合类型

2.13.2.1 类型介绍

数据库提供了变长数组（VARRAY）、嵌套表（TABLE）、索引表（ITABLE，也称联合数组）三种同质元素的集合类型。

以上三者区别如下表所示。

属性	变长数组 (VARRAY)	嵌套表 (TABLE)	索引表 (ITABLE)
是否可用于 SQL	可用	可用	不可用
表字段类型	可用，数据在同一表中	可用，数据在同一表中	不可用
初始化	声明时自动完成	声明时自动完成	声明时自动完成
未初始化状态	引用时非法访问	引用时非法访问	元素未定义
数据连续性	连续	连续，删除后不连续	不连续（稀疏）
是否有界	有 size 界	可以扩展	无界
对任一元素赋值	不可以，需要先扩展	不可以，需要先扩展	可以
扩展方法	EXTEND	EXTEND,EXTEND()	给新元素赋值
比较操作	支持	支持	支持
集合操作 bulk collect into	支持	支持	支持
元素数量	指定的	未指定的	未指定的
索引类型	整数	整数	整数或字符串
密集或稀疏	总是密集的	开始密集，删除中间值变得稀疏	稀疏
定义或声明	在 PL/SQL 块、包、存储过程、存储函数或在模式级别中	在 PL/SQL 块、包、存储过程、存储函数或在模式级别中	在 PL/SQL 块、包、存储过程或存储函数中

⚠ 注意

可以在 SQL 或 PL/SQL 中使用的集合类型，若在 SQL 中使用集合类型，必须是在模式级别中用 CREATE TYPE 语句创建的类型（如：表字段类型若为集合类型，则该集合类型是模式级别创建的类型）。

在 PL/SQL 中使用集合类型主要分为三步来完成，一是声明，二是初始化，三是赋值。初始化和赋值可以在声明块中完成，也可以在执行块中完成。集合类型在声明变量时，自动完成初始化。

2.13.2.2 集合声明

语法格式

集合的声明分为集合类型声明和集合类型的变量声明。

- 变长数组：

```
-- 集合类型声明
TYPE type_name IS VARRAY(size) OF type_x;
-- 变量声明
variable_name type_name;
```

- 嵌套表：

```
-- 集合类型声明
TYPE type_name IS TABLE OF type_x;
-- 变量声明
variable_name type_name;
```

- 索引表：

```
-- 集合类型声明
TYPE type_name IS TABLE OF type_x INDEX BY PLS_INTEGER;
-- 变量声明
variable_name type_name;
```

参数说明

- type_name：定义的集合类型的名称。
- VARRAY(size)：变长数组，其中 size 是这个数组的最大长度。
- type_x：数据类型，可以是基本数据类型（如 NUMBER、VARCHAR2 等）或其他用户自定义类型。
- variable_name：变量名称。
- TABLE：表类型。
- INDEX BY PLS_INTEGER：表的索引是基于整数的。索引可以是非连续的，可以从任何整数值开始。

2.13.2.3 集合初始化和赋值

集合的初始化主要通过构造函数进行初始化。有以下 4 种方法（以 VARRAY 数组为例）：

- 在声明块中声明集合，在声明块中使用构造函数，在执行块中使用 EXTEND 方法进行赋值。

```
DECLARE
    TYPE type_name IS VARRAY(1) OF INT;
    -- 声明集合，使用构造函数初始化
    variable_name type_name:= type_name();
BEGIN
    -- 使用EXTEND赋值
    variable_name.EXTEND;
    variable_name(1) := 1;
    SEND_MSG(variable_name(1));
END;
/
-- 输出
1
```

- 在声明块中声明集合，在声明块中使用构造函数初始化并赋值。

```
DECLARE
    TYPE type_name IS VARRAY(1) OF INT;
    -- 声明集合，使用构造函数初始化并赋值
    variable_name type_name:= type_name(1);
BEGIN
    SEND_MSG(variable_name(1));
END;
/
-- 输出
1
```

- 在声明块中声明集合，在执行块使用构造函数初始化为空（EMPTY），在执行块中使用 EXTEND 方法进行赋值。

```
DECLARE
    TYPE type_name IS VARRAY(1) OF INT;
    -- 声明集合
    variable_name type_name;
BEGIN
-- 使用构造函数初始化为空
    variable_name := type_name();
    -- 使用EXTEND赋值
    variable_name.EXTEND;
    variable_name(1) := 1;
    SEND_MSG(variable_name(1));
END;
/
-- 输出
1
```

- 在声明块中声明集合，在执行块使用构造函数初始化并赋值。

```
DECLARE
    TYPE type_name IS VARRAY(1) OF INT;
    -- 声明集合
    variable_name type_name;
BEGIN
    -- 使用构造函数初始化并赋值
    variable_name := type_name(1);
    SEND_MSG(variable_name(1));
END;
/
-- 输出
1
```

⚠ 注意

初始化为空表示的是一个空集合（EMPTY，但非 NULL），而未初始化时是 NULL（UNKNOWN 值）。变成数组类型（VARRAY）和嵌套表类型（TABLE）在赋值之前必须初始化，可以用 EXTEND 扩展，用构造函数初始化，而索引表（ITABLE）无需初始化。

2.13.3 集合方法

2.13.3.1 概述

集合方法是一系列返回有关集合的信息的函数或对集合进行操作的过程，详细信息如下表所示。

集合方法	描述	使用限制
DELETE	删除集合所有元素	通用
DELETE(n)	删除集合下标为 n 的元素	对 VARRAY 非法
DELETE(n,m)	删除集合下标为 n m 的元素	对 VARRAY 非法
TRIM	从集合末端开始删除 1 个元素	对 ITABLE 非法
TRIM(n)	从集合末端开始删除 n 个元素	对 ITABLE 非法
EXTEND	为集合添加一个元素，初始值为 NULL	对 ITABLE 非法
EXTEND(n)	为集合添加 n 个元素，初始值为 NULL	对 ITABLE 非法

接下页

集合方法	描述	使用限制
EXTEND(n,m)	为集合添加 n 个元素，初始值为 m	对 ITABLE 非法
EXISTS(n)	若下标为 n 的元素存在时，返回 TRUE	通用
COUNT 或 COUNT()	该集合元素的数目	通用
LIMIT	返回 VARRAY 数组创建时指定的 SIZE	VARRAY
FIRST 或 FIRST()	返回第一个元素索引号，空返回 NULL	通用
LAST 或 LAST()	返回最后一个元素索引号，空返回 NULL	通用
PRIOR(n)	返回当前元素 n 的前一个索引号，空返回 NULL	通用
NEXT(n)	返回当前元素 n 的下一个索引号，空返回 NULL	通用

2.13.3.2 DELETE

说明

DELETE 用于删除集合中的元素。

DELETE 有三种形式：

- DELETE：从集合中删除所有元素，适用于三种类型的集合。
- DELETE(n)：删除索引为 n 的元素（如果该元素存在），否则不进行任何操作，不适用于 VARRAY。
- DELETE(n, m)：删除索引 n m 的元素，不适用于 VARRAY。

DELETE 删除元素不会删除该元素的占位符，可以直接赋值给被删除元素进行恢复。

示例

```
DECLARE
    TYPE e_type IS TABLE OF NUMBER;
    e e_type := e_type(1, 2, 3, 4);
BEGIN
    -- 删除索引2到5的元素（5不存在）
    e.DELETE(2, 5);
```

```
    FOR i IN e.FIRST .. e.LAST LOOP
        SEND_MSG(e(i));
    END LOOP;
--  重新赋值索引2的元素
    e.(2) := 22;
    FOR i IN e.FIRST .. e.LAST LOOP
        SEND_MSG(e(i));
    END LOOP;
END;
/
--  输出
1
--  第二次操作输出，被DELETE删除的元素可以直接重新赋值
1
22
```

2.13.3.3 TRIM

说明

TRIM 用于从变长数组或嵌套表的末尾删除元素并销毁位置，不支持索引表。

TRIM 有两种形式：

- TRIM：如果集合至少有一个元素，则 TRIM 从集合末尾删除 1 个元素；反之会报错“[E5210] 下标 1 超界”。
- TRIM(n)：如果集合末尾至少有 n 个元素，则从集合末尾删除 n 个元素；反之会报错“[E5210] 下标 n 超界”。

DELETE 删除一个元素但会保留占位符，而 TRIM 处理时会认为该元素存在（占据一个位置），将被删除元素的占位符销毁。因此，TRIM 可以销毁已删除的元素。

示例

```
DECLARE
    TYPE e_type IS TABLE OF NUMBER;
    e e_type := e_type(1, 2, 3, 4);
BEGIN
    --  从集合末端开始删除2个元素
    e.TRIM(2);
    FOR i IN e.FIRST .. e.LAST LOOP
        SEND_MSG(e(i));
    END LOOP;
END;
/
--  输出
1
2
```

2.13.3.4 EXTEND

说明

EXTEND 用于添加元素位置到变长数组或嵌套表的末尾，并将其设置为 NULL，不适用于索引表。

EXTEND 有三种形式：

- EXTEND：新增一个 NULL 元素到集合尾部。
- EXTEND(n)：新增 n 个 NULL 元素到集合尾部。
- EXTEND(n, m)：新增 n 个元素到集合尾部，并且给它们赋值索引为 m 的值。

示例

```
DECLARE
    TYPE e_type IS VARRAY(10) OF VARCHAR(10);
    e e_type;
BEGIN
    -- 增加1个元素
    e.EXTEND();
    -- 赋值为yi
    e(1):='yi';
    -- 增加2个元素
    e.EXTEND(2);
    -- 给索引为3的元素赋值为san
    e(3):='san';
    -- 再增加2个元素，并且为它们赋值索引为3的值，即san
    e.EXTEND(2, 3);
    FOR i IN 1..e.COUNT() LOOP
        SEND_MSG(e(i));
    END LOOP;
END;
/
-- 输出
yi
NULL
san
san
san
```

2.13.3.5 EXISTS

说明

EXISTS 用于显示指定元素是否存在。

如果集合的第 n 个元素存在，则 EXISTS(n) 返回 true，否则返回 false。如果 n 超出范围，EXISTS 将返回 false。对于已删除的元素，EXISTS(n) 返回 false。

示例

```
DECLARE
    TYPE e_type IS TABLE OF NUMBER;
    e e_type := e_type(1, 2);
BEGIN
e.DELETE(2);
    SEND_MSG(e.EXISTS(1));
    SEND_MSG(e.EXISTS(2));
    SEND_MSG(e.EXISTS(3));
END;
/
-- 输 出
true
false
false
```

2.13.3.6 COUNT

说明

COUNT 用于返回集合中可访问的元素数量。使用 EXTEND、TRIM、DELETE 时会影响 COUNT 的结果。

示例

```
DECLARE
    TYPE e_type IS TABLE OF NUMBER;
    e e_type := e_type(1, 2);
BEGIN
    SEND_MSG(e.COUNT);
    -- 为 集 合 添 加 1 个 元 素
    e.EXTEND(1);
    SEND_MSG(e.COUNT);
    -- 从 集 合 末 端 开 始 销 毁 2 个 元 素
    e.TRIM(2);
    SEND_MSG(e.COUNT);
    -- 删 除 第 1 个 元 素
    e.DELETE(1);
    SEND_MSG(e.COUNT);
END;
/
-- 输 出
2
3
1
0
```

2.13.3.7 LIMIT

说明

LIMIT 用于返回集合的最大元素数。如果集合没有最大元素数，则 LIMIT 返回 NULL。只有 VARRAY 需要设定最大元素数。

示例

```
DECLARE
    TYPE e_type IS TABLE OF INT;
    TYPE I_type IS TABLE OF INT INDEX BY PLS_INTEGER;
    TYPE v_type IS VARRAY(10) of INT;
    e e_type;
    I I_type;
    v v_type;
BEGIN
    -- 分别输出嵌套表、索引表和VARRAY数组创建时的SIZE
    SEND_MSG(e.LIMIT);
    SEND_MSG(i.LIMIT);
    SEND_MSG(v.LIMIT);
END;
/
-- 输出
NULL
NULL
10
```

2.13.3.8 FIRST/LAST

说明

FIRST 和 LAST 分别用于返回第一个和最后一个元素的索引。如果集合只有一个元素，则 FIRST 和 LAST 返回相同的索引。如果集合为空，则 FIRST 和 LAST 返回 NULL。

示例

```
DECLARE
    TYPE e_type IS TABLE OF NUMBER;
    e e_type := e_type(1, 2, 3);
BEGIN
    SEND_MSG(e.FIRST);
    SEND_MSG(e.LAST);
    -- 删除索引为1和2的元素
    e.DELETE(1,2);
    SEND_MSG(e.FIRST);
    SEND_MSG(e.LAST);
    -- 删除索引为3的元素
    e.DELETE(3);
    SEND_MSG(e.FIRST);
    SEND_MSG(e.LAST);
END;
/
-- 输出
1
3
3
```

```
3
NULL
NULL
```

2.13.3.9 PRIOR/NEXT

说明

给定一个索引，PRIOR 返回集合中的前一个现有元素的索引（如果存在），否则，PRIOR 返回 NULL。同理，NEXT 返回集合中的后一个现有元素的索引（如果存在），否则，NEXT 返回 NULL。

给定的索引不一定存在，PRIOR 和 NEXT 会根据索引类型的排序顺序查找前一个或后一个索引。

示例

```
DECLARE
    TYPE e_type IS TABLE OF NUMBER;
    e e_type := e_type(1, 2, 3, 4);
BEGIN
    SEND_MSG(e.PRIOR(1));
    SEND_MSG(e.NEXT(1));
    e.DELETE(2);
    -- 给定的索引不一定存在
    SEND_MSG(e.PRIOR(2));
    SEND_MSG(e.NEXT(2));
END;
/
-- 输出
NULL
2
1
3
```

2.14 数据类型隐式转换

数据类型隐式转换用于将一个常量或表达式从一种数据类型转换为另一种数据类型，前提是两种数据类型间能够进行转换。

下表展示虚谷中能够隐式转换的数据类型。

数据类型	INTEGER	FLOAT	DOUBLE	NUMRIC	CHAR	VARCHAR	CLOB	BOOLEAN	BINARY	BLOB	DATE	TIME	INTERVAL
INTEGER	/	√	√	√	√	√	√	√	√	×	×	×	×
FLOAT	√	/	√	√	√	√	×	√	√	×	×	×	×
DOUBLE	√	√	/	√	√	√	×	√	√	×	×	×	×
NUMERIC	√	√	√	/	√	√	×	√	√	×	×	×	×
CHAR	√	√	√	√	/	√	√	×	√	×	√	√	√
VARCHAR	√	√	√	√	√	/	√	×	√	×	√	√	√
CLOB	×	×	×	×	√	√	/	×	√	√	×	×	×
BOOLEAN	×	×	×	×	×	×	×	/	√	×	×	×	×
BINARY	×	×	×	×	√	√	√	√	/	√	×	×	×
BLOB	×	×	×	×	×	×	×	×	√	/	×	×	×
DATE	×	×	×	×	√	√	×	×	√	×	/	×	×
TIME	×	×	×	×	√	√	×	×	√	×	×	/	×
INTERVAL	×	×	×	×	√	√	×	×	√	×	×	×	/

2.14.0.1 数据类型隐式转换规则

- 在 INSERT/UPDATE 操作期间，会将值转换为受影响列的数据类型
- 在 SELECT 操作期间，会将数据从列转换为目标变量的类型
- 在操作数值时，通常会调整精度和小数位数，以实现最大容量。在这种情况下，由此类操作生成的数值数据类型可能与基础表中的数值数据类型不同
- 将字符值与数值进行比较时，会将字符数据转换为数值
- 字符值与浮点数值之间的转换可能不精确

- 进行赋值时，会将等号（=）右侧的值转换为左侧赋值目标的数据类型

2.14.0.2 数据类型隐式转换示例

示例

```
-- 将字符串转换为整数
SELECT 12+'10' FROM dual;

EXPR1 |
---------------------------------------
22 |

-- 将整数转换为浮点数，浮点数转换为整数
CREATE TABLE TEST0(A INT, B FLOAT);

INSERT INTO TEST0 VALUES(3.14,12);

SELECT * FROM TEST0;

A | B |
---------------------------------------
3 | 1.200000e+01 |

-- DDL时输入值向数据类型转换
CREATE TABLE TEST1(A INT,B VARCHAR);

INSERT INTO TEST1 VALUES(1,'AA')(2,'BB');

CREATE TABLE TEST2(A VARCHAR,B CLOB);

IMPORT TABLE TEST2 FROM SELECT * FROM TEST1;

SET SHOW_TYPE ON;

SELECT * FROM TEST2;

A(CHAR(-1)) | B(CLOB) |
---------------------------------------
1| AA|
2| BB|

-- DML时输入值向数据类型转换
INSERT INTO TEST1 VALUES('3',SYSDATE);

SELECT * FROM TEST1;

A(INTEGER) | B(CHAR(-1)) |
---------------------------------------
1 | AA|
2 | BB|
3 | 2022-10-08 11:10:04|
```

3 表达式

3.1 概述

数据库表达式是指在 SQL 查询或 PL/SQL 程序中使用的计算公式或条件语句。这些表达式可以包括常量、列名、变量、函数以及运算符，用于生成新的值或进行条件判断。数据库表达式在各种 SQL 语句中广泛使用，如 SELECT、WHERE、CASE 语句等。

虚谷数据库的表达式兼容 SQL92 标准，与常见关系数据库的表达式语法大致相同，只有少量表达式具有自身特色，其中包括：某些常量表达式中、强类型转换表达式。以下内容并不详细介绍虚谷数据库的所有表达式语法，而只重点介绍其中一些需要注意的表达式。

3.2 常量表达式

常量表达式用于表示各种类型的常数，对于 CHAR、VARCHAR、TINYINT、SMALLINT、INTEGER、BIGINT 等数据类型，其常量的表达与其它常见关系数据库的相应类型的常量表达方式一致，具有自身特色的常量表达式如下：

日期及时间常量

根据 xugu.ini 配置文件中的默认日期时间格式（def_timefmt）进行日期时间常量操作：

- DATE 类型常量表达式：如“2006-6-5”
- DATETIME 类型常量表达式：如“2006-6-5 12:32:20”
- TIME 类型常量表达式：如“12:32:20”

字符串常量

虚谷数据库兼容 MySQL 双引号的使用（需要将兼容性参数 compatible_mode 设置为 MYSQL）。

- 支持使用英文单引号和英文双引号标识字符串。

```
SET COMPATIBLE_MODE TO NONE; --无兼容模式

SELECT "hello", "'hello'", "''hello''", "hel""lo" FROM DUAL;
Error: [E10049 L1 C8] 字段变量或函数"hello"不存在

SET COMPATIBLE_MODE TO MYSQL; --将compatible_mode设置为MySQL

SELECT "hello", "'hello'", "''hello''", "hel""lo" FROM DUAL;
```

```
EXPR1 | EXPR2 | EXPR3 | EXPR4 |
------------------------------------------------------
hello| 'hello'| ''hello''| hel"lo|
```

- 查询输出时，支持字符串作为别名（如为两个连续字符串，则进行字符拼接，不作为别名）。

```
SET COMPATIBLE_MODE TO NONE; --无兼容模式

SELECT 1 "id" FROM DUAL;
id |
------------------------------------------------------
1 |

SELECT "a" "b" FROM DUAL;
Error: [E10049 L1 C8] 字段变量或函数"a"不存在

SET COMPATIBLE_MODE TO MYSQL; --将compatible_mode设置为MySQL

SELECT "a" "b" FROM DUAL;
EXPR1 |
------------------------------------------------------
ab|
```

3.3 字段值表达式

3.3.1 概述

字段值表达式可以采用多种形式，包括简单的标识符、表名、字段名以及表名加字段名组合形式、使用别名的情况。

3.3.2 标识符

在一个查询中引用一个字段时，如果这个字段的名字在整个查询中是唯一的，或者是在单表查询中，你可以直接使用字段名作为标识符。

示例“name”作为标识符。

```
SELECT name FROM users;
```

3.3.3 表名. 字段名

当需要明确指定字段来自哪个表时，尤其是在多表查询（如联接操作）中，使用表名. 字段名的方式来引用字段。

示例“users.name”和“orders.order_id”都是指定了表名和字段名的表达式。

```
SELECT users.name, orders.order_id FROM users INNER JOIN orders ON
   users.id = orders.user_id;
```

3.3.4 别名

别名可以在查询中用来简化长表名或字段名，使得查询更加易读。

示例表别名：“u”是“users”表的别名，“o”是“orders”表的别名。

```
SELECT u.name, o.order_id FROM users u INNER JOIN orders o ON u.id
    = o.user_id;
```

字段别名：“user_name”是“name”字段的别名。

```
SELECT name AS user_name FROM users;
```

3.4 强类型转换表达式

3.4.1 概述

强类型转换用于将一个常量或表达式从一种数据类型转换为另一种数据类型，前提必须是两种数据类型间能够进行转换。强类型转换有两种方式：

- 函数方式
- 操作符方式

3.4.2 函数方式

使用 CAST 函数进行转换。

示例

```
--将实数123.45转化为整数
SELECT CAST(123.45 AS INTEGER) FROM dual;

EXPR1 |
------------------------------------------------------
123 |

--将 DATETIME 时间类型转换为 DATE 类型
SELECT cast('1999-7-2 20:20:20' AS DATE) FROM dual;

EXPR1 |
------------------------------------------------------
1999-07-02 AD |
```

3.4.3 操作符方式

使用:: 操作符进行转换。

示例

操作符方式示例

```
--将实数123.45转化为整数
SELECT 1234.45::INTEGER FROM dual;

EXPR1 |
------------------------------------------------------
1234 |
```

3.5 子查询表达式

3.5.1 概述

子查询表达式在 SQL 中用于在一个查询中嵌入另一个查询。子查询可以返回单个值、一行、一列或多行多列的结果，具体取决于其用途。

3.5.2 查值型子查询表达式

此类子查询表达式在整个表达式中充当一个值表达式，如同一个变量，这类表达式输出一列。

示例

从员工信息表 empinfo 中查找出员工编号与父查询记录相同的员工地址信息，子查询“SELECT address FROM empinfo WHERE empinfo.id=emp.id”作为一个值表达式。

```
SELECT id, name, (SELECT address FROM empinfo WHERE empinfo.id =
   emp.id)
  FROM emp
WHERE emp.id = 10;
```

3.5.3 布尔型子查询表达式

布尔型子查询表达式在 SQL 中主要用于返回一个布尔值，以帮助主查询决定是否满足某个条件。

语法格式

```
EXISTS (SELECT ...)

NOT EXISTS (SELECT ...)
```

“EXISTS”在子查询结果集非空时返回真，否则返回假。同理，“NOT EXISTS”在子查询结果集为空时返回真，否则返回假。

示例

从员工表 emp 中查找地址信息（地址信息在另一表达式 empinfo 中）非空的员工信息。

```
SELECT name FROM emp WHERE EXISTS
(SELECT * FROM empinfo WHERE empinfo.address NOTNULL AND empinfo.id
    = emp.id);
```

3.5.4 比较型子查询表达式

比较型子查询表达式是 SQL 查询中用来基于子查询结果进行条件筛选的一种方式。

语法格式

```
expr op [ANY|ALL] (SELECT ...)
```

expr 是一个值表达式，op 表示比较操作符，子查询应只输出一个字段值，子查询结果可以是多行，若指定“ALL”选项，则在 expr 与所有子查询输出结果的关系比较都为真时，整个表达式返回真，否则返回假；若指定“ANY”选项，则只要 expr 与子查询结果集中至少一个结果值比较为真时，整个表达式返回真值。

示例

从员工表中找出工资在 1000 元以上的员工名字。子查询用于找出工资在 1000 元以上的员工的 ID，只要 emp 表中记录的 id 字段值与子查询结果集中任意一个相等，则该员工的名字就将被选中输出。

```
SELECT name FROM emp WHERE id = ANY (SELECT id FROM salary WHERE
   sal > 1000);
```

3.5.5 多列比较型子查询表达式

多列比较型子查询允许在子查询中同时比较多个列的值，表达式的个数应等于子查询的列数，子查询的一行记录将与多个表达式进行比较。

语法格式

```
(expr1, ..., exprn) op [ANY|ALL] (SELECT ...)
```

通常，比较操作符只有“=”与“<>”两种，由于有多列值参与比较，若使用其它类型的操作符，将难以界定各字段比较结果与整体结果的关系。子查询结果可以是多行，若指定“ALL”选项，则在 expr 与所有子查询输出的记录行的关系比较都为真时，整个表达式返回真，否则返回假；若指定“ANY”选项，则只要 expr 与子查询结果集中至少一行进行比较为真时，整个表达式返回真值。

示例

从 unit 表中选择 id 和 name 列，其中 (id, name) 组合必须与 unit1 表中 id1 > 1000 的任意 (id1, name1) 组合完全匹配。

```
SELECT id, name FROM unit WHERE (id, name) = ANY (SELECT id1, name1
    FROM unit1 WHERE id1 > 1000);
```

3.5.6 IN 型子查询表达式

IN 型子查询表达式是 SQL 中常用的一种子查询形式，用于检查某个值是否存在于子查询返回的结果集中。这种子查询通常用于基于子查询结果进行过滤。

语法格式

```
expr IN (SELECT ...)

expr NOT IN (SELECT ...)
```

expr 是一个值表达式，“expr IN (SELECT ...)”相当于“expr = ANY(SELECT ...)”；“expr NOT IN (SELECT ...)”相当于“expr<> ALL(SELECT ...)”。

示例

首先找出所有工资大于 1000 的员工 ID。使用子查询的结果作为过滤条件，从 emp 表中选择那些 ID 出现在内部查询结果中的员工名字。

```
SELECT name FROM emp WHERE id IN (SELECT id FROM salary WHERE sal
    > 1000);
```

3.6 CASE WHEN 表达式

3.6.1 概述

CASE WHEN 表达式在 SQL 中用于实现条件逻辑。它可以用于在查询中根据不同的条件返回不同的值。CASE WHEN 表达式有两种形式：简单形式和搜索形式。

3.6.2 简单形式

将表达式 expr 的值与 val_expr 进行比较，若相等，则返回与 WHEN 相对应的 THEN 后面的值，若都不匹配且语句中含 ELSE 子句，则返回 ELSEe 后的值，若都不匹配且语句中不含 ELSE 子句，则返回空值。

语法格式

```
CASE expr
    WHEN val_expr1
    THEN result_expr1
```

```
    WHEN val_expr2
    THEN result_expr2
    ...
    ELSE result_exprN
END
```

示例

```
SELECT name,
    CASE dept_no
        WHEN 1
        THEN '财 务 部'
        WHEN 2
        THEN '技 术 部'
        WHEN 3
        THEN '销 售 部'
        ELSE '其 它 部 门'
    END
FROM emp;
```

3.6.3 搜索形式

本语句 CASE 后不带表达式，各个 WHEN 后面不是值表达式而是布尔表达式，本语句将依次判断各个 WHEN 后的布尔表达式，若其值为真，则返回 THEN 后的值，都不匹配时，若语句含 ELSE 子句，则返回 ELSE 后面的值，否则返回空值。

语法格式

```
CASE
    WHEN bool_expr1
    THEN result_expr1
    WHEN bool_expr2
    THEN result_expr2
    ...
    ELSE result_exprN
END
```

示例

```
SELECT name,
    CASE
        WHEN sal < 1000
        THEN '低 收 入 水 平'
        WHEN sal > 1000
        AND sal  < 5000
        THEN '中 等 收 入'
        ELSE '高 收 入'
    END
FROM emp_salary;
```

3.7 其他类型表达式

3.7.1 IN 表达式

IN 表达式可以用来检查一个值是否存在于一个固定的值列表中。

语法格式

```
expr IN (常量1, ..., 常量n)

expr NOT IN (常量1, ..., 常量n)
```

常量 1 至常量 n 构成常量数组，若表达式与数组中任一常量相等，则表达式返回真，与所有常量皆不等，则表达式返回假。

示例

查询 unit 表中包含王二，李三，杨四，梁五的信息。

```
SELECT * FROM unit WHERE name IN ('王二', '李三', '杨四', '梁五');
```

3.7.2 行比较表达式

行比较表达式在 SQL 中用于比较两组或多组列值，允许在单个子句中同时比较多个列的值。

语法格式

```
(expr11, ..., expr1n) op (expr21, ..., expr2n)
```

op 为比较操作符，相当于“(expt11 op expr21) ... AND (expr1n op expr2n)”行比较表达式使多个值比较显得清晰，易理解。

示例

查询 unit 表中 id 和 name 分别为 1 和王二的信息。

```
SELECT * FROM unit WHERE (id, name) = (1, '王二');
```

4 PL/SQL 语言

4.1 概述

4.1.1 PL/SQL 介绍

PL/SQL（Procedural Language/Structured Query Language）是数据库中的一种扩展语言，它结合了 SQL 的强大功能和过程化语言的编程特点，在数据库中提供了一种比单纯的 SQL 更为灵活和强大的编程环境。其逻辑是把数据操作和查询语句组织在 PL/SQL 代码的过程性单元中，通过逻辑判断、循环等操作实现复杂的功能或者计算。

4.1.2 PL/SQL 的特点

数据处理及集成性

- 与 SQL 紧密集成：可以直接在 PL/SQL 代码中使用 SQL 语句进行数据的查询、插入、更新和删除等操作，无需在应用程序代码中单独编写 SQL 语句。
- 事务处理：PL/SQL 是一种高性能的基于事务处理的语言，能够处理复杂的数据库事务，确保数据的一致性和完整性。

过程化的编程特性

- 构化编程：PL/SQL 支持顺序、条件、循环等结构化编程控制流程，使得程序逻辑更加清晰、易于理解和维护。
- 模块化编程：PL/SQL 支持将代码组织为模块（存储过程、函数、触发器、包等），提高了代码的重用性和可维护性。

操作的可交互性强

- 减少网络交互：PL/SQL 程序通常存储在数据库服务器上，并在数据库内部执行，减少了客户端和服务器之间的网络交互，提高了程序的执行效率。
- 批量处理：PL/SQL 可以一次性处理多条 SQL 语句，减少了网络传输的数据量，进一步提高了程序的性能。

执行安全性有保障

- 数据库内部执行：PL/SQL 代码在数据库服务器端执行，可以在数据库内部实现安全访问控制，保护数据的安全性。

- 权限控制：虚谷数据库提供了丰富的权限控制机制，可以精确控制用户对 PL/SQL 程序的访问权限。

异常处理机制灵活

异常捕获和处理：PL/SQL 具有强大的异常处理机制，可以捕获并处理运行时错误，保证程序的稳定性和可靠性。

可重用与可移植性

- 可重用性：PL/SQL 程序单元（如存储过程和函数）可以被命名并存储在数据库中，供其他 PL/SQL 程序或 SQL 命令调用，提高了代码的重用性。
- 可移植性：PL/SQL 编写的程序可以很容易地移植到另一个同类型数据库中，保持了程序的一致性和可维护性。

4.2 PL/SQL 语法

4.2.1 PL/SQL 程序结构

PL/SQL 的程序结构由三部分组成：声明部分、可执行部分以及异常处理部分。

主要结构说明

每个 PL/SQL 语句以分号结尾。使用 BEGIN 和 END 可以将 PL/SQL 块嵌套在其他 PL/SQL 块中。以下是 PL/SQL 块的基本结构。

```
DECLARE
-- 申 明 部 分
<declarations section>
BEGIN
-- 可 执 行 部 分
<executable command(s)>
EXCEPTION
-- 异 常 处 理 部 分
<exception handling>
END;
```

⚠ 注意

[2.0]

- 在 Console 控制台执行时，匿名语句块最后需要添加“/”标识结束，在其他管理工具执行时无需添加“/”。
- PLSQL 语言有着较强的结构性限制，存在 BEGIN 时必须对应一个 END 标识结束。分支语句或嵌套语句块也需要明确的 END 标识。
- 使用较多的逻辑嵌套时，需要注意逻辑递进关系。执行区域的操作都需以分号结尾。

4.2.2 声明部分（DECLARE）

（可选部分）此部分是以关键字 DECLARE 开头，定义了程序中要使用的所有变量、类型、游标、异常、子程序和其他元素。变量用于存储数据，游标用于从数据库中提取数据。

语法格式

```
DECLARE VarDefList

VarDefList::=
VarDef ;
|    VarDefList VarDef ;

VarDef::=
name [CONSTANT] TypeName
|    CursorDef
|    ExceptionDef
|    TypeDefStmt
|    PRAGMA EXCEPTION_INIT (name,integer)

TypeDefStmt::=
{TYPE | SUBTYPE} ColId IS TypeName

TypeName::=
STypename
|    TABLE OF STypename [INDEX BY STypename]
|    VARRAY (ICONST) OF STypename
|    VARYING ARRAY (ICONST) OF STypename

STypename::=
ConstTypename
|    name_space ['%' ROWTYPE|'%' ROW TYPE|'%' TYPE]]
|    {ROWTYPE | ROW TYPE | TYPE} OF name_space
|    REF CURSOR
|    RECORD (col_defs)
```

参数说明

- name [CONSTANT] TypeName：普通变量，定义了一个变量或常量。其中 name 是变量名，[CONSTANT] 是一个可选关键字，用来指定该变量是否为常量，TypeName 指定了变量的数据类型。
- CursorDef：游标定义，游标是一种数据库对象，允许逐行处理查询结果集。
- ExceptionDef：异常定义，用于定义用户自定义的异常。
- TypeDefStmt：类型定义，用于定义新的数据类型。
- PRAGMA EXCEPTION_INIT (name, integer)：异常初始化，它将用户定义的异常与特定的错误代码关联起来。这里的 name 是异常名称，integer 是错误代码。

- STypename：定义了基本类型，包括常量类型、命名空间引用、记录类型、REF CURSOR 类型等。
- ConstTypename：定义了具体的常量类型，包括数值类型、字符类型、日期时间类型、RAW 类型和 BINARY 类型。

> **说明**
>
> [2.0] 各定义的详细信息请参见《SQL 语法参考指南》的各对应章节。

4.2.3 可执行部分以及异常处理部分（BEGIN...END）

- 可执行命令部分：此部分包含在关键字 BEGIN 和 END 之间，这是一个强制性部分。它由程序的可执行 PL/SQL 语句组成。它应该有至少一个可执行代码行，它可以只是一个 NULL 命令，表示不执行任何操作。
- 异常处理部分：（可选部分）此部分以关键字 EXCEPTION 开头。它包含处理程序中错误的异常。可以捕获并处理预定义的异常，也可以处理用户定义的异常。

语法格式

```
BEGIN
[<<ColId>> | ColId :] pl_stmt ...
[OptExceptionStmt]
END

pl_stmt::=
VarAssign
|    IfElseStmt
|    LoopStmt
|    ForStmt
|    ForallStmt
|    ThrowStmt
|    CaseStmt
|    GotoStmt
|    BreakStmt
|    ContinueStmt
|    ReturnStmt
|    NULL ;
|    ProcDef ;
|    sql_stmt ;
|    StmtBlock [;]
```

参数说明

- «ColId» | ColId :：可选标签，标记一段代码。
- pl_stmt：可以是一个或多个的过程化语句。

- OptExceptionStmt：异常处理部分。
- VarAssign：变量赋值语句。
- IfElseStmt：IF 条件语句。
- LoopStmt：LOOP 循环语句。
- ForStmt：FOR 循环语句。
- ForallStmt：批量操作语句。
- ThrowStmt：抛出异常的语句。
- CaseStmt：CASE 选择语句。
- GotoStmt：无条件跳转语句。
- BreakStmt：跳出循环。
- ContinueStmt：继续下一次循环。
- ReturnStmt：返回函数结果。
- NULL ;：空语句。
- ProcDef ;：过程定义语句，定义一个新的过程或函数。
- sql_stmt ;：SQL 语句，可以直接在 PL/SQL 中执行 SQL 操作。
- StmtBlock [;]：嵌套的可执行命令部分，可以在当前语句块中定义另一个语句块。

> **说明**
>
> [2.0] 各定义的详细信息请参见《SQL 语法参考指南》的各对应章节。

4.2.4 示例

定义变量

块语句中的变量，在声明部分（DECLARE 和 BEGIN 之间）定义。

```
DECLARE
-- 简单数据类型变量
v_int INT;
v_char VARCHAR;
-- 自定义数据类型变量
--   (1) 定义“自定义类型”
TYPE TYPE_VARRAY_INT IS VARRAY(5) OF PLS_INTEGER;
```

```
--    (2) 定义“自定义类型”的变量
v_varray_int TYPE_VARRAY_INT;
-- 游标变量
CURSOR v_cur IS SELECT name, age FROM tab_students WHERE student_id
    < 10;
BEGIN
...
END;
```

变量赋值

块语句中的变量，既可以在定义的同时赋值，也可以在“执行体”（BEGIN 和 END 之间）赋值。在执行体中有两种语法对变量进行赋值：一种是用赋值符号:= ，另一种是用 select into 语法。

```
SQL> DECLARE
v_int INT;
v_char VARCHAR;
-- 在变量定义的同时进行赋值
v_float FLOAT := 12.3;
BEGIN
-- 在执行体中给变量赋值
-- 1. 用赋值符号给变量赋值
v_int := 10;
-- 2. 用 select into 语法给变量赋值
SELECT 'string' INTO v_char;
-- 把各变量的值打印至屏幕
SEND_MSG('Value of v_int is: ' || v_int);
SEND_MSG('Value of v_char is: ' || v_char);
SEND_MSG('Value of v_float is: ' || v_float);
END;
/
-- 输出
Value of v_int is: 10
Value of v_char is: string
Value of v_float is: 12.3
```

4.3 IF 条件控制语句

IF 语句用于构造基本的条件语句，条件控制语句赋予了代码能够根据条件改变执行流程的能力。

语法格式

```
IF bool_expr THEN
    pl_stmt_list
[ELSIF bool_expr THEN
    pl_stmt_list
[ELSIF bool_expr THEN
    pl_stmt_list
...]]
```

```
[ELSE
    pl_stmt_list]
{END IF | ENDIF | END IF name_space} ;
```

参数说明

- bool_expr：布尔表达式，如果该表达式的值为真，则执行 THEN 后面的 pl_stmt_list。若有 ELSE IF 则依次进行条件判断，若均未匹配成功则执行 ELSE 分支 pl_stmt_list 或退出该条件构造逻辑。
- pl_stmt_list：一个或多个过程性 SQL 语句的列表，可以为变量赋值语句、IF 条件控制语句、循环控制语句、嵌套定义、SQL 执行语句等语句。
- ELSIF：ELSIF 关键字用于添加额外的条件判断。可以有多个 ELSIF 子句。
- ELSE：ELSE 关键字用于指定当所有 IF 和 ELSIF 条件都不满足时执行的语句。
- namespace：带命名空间的结束 IF 语句，用于标识具体的 IF 语句，通常用于嵌套的 IF 语句中。

示例

- 块语句中使用 IF

  块语句中分别定义 3 个常数变量 a、b、c，并对其赋值，若判断 a 既大于 b 又大于 c，则输出消息 a is bigger than b and c，否则输出 b 值。

```
DECLARE
    a INTEGER;
    b INTEGER;
    c INTEGER;
BEGIN
    a := 40;
    b := 20;
    c := 30;
    IF a > b AND a > c THEN
        SEND_MSG('a is bigger than b and c');
    ELSE
        SEND_MSG(b);
    END IF;
END;
/
-- 输出
a is bigger than b and c
```

[2.0] **说明**

在控制台工具中使用块语句需要在 END; 后加/表示结束，在 DBeaver 中则无需加/。

- 存储过程中使用 IF

  在存储过程中声明一个整数类型变量 id_1，并在逻辑判断（IF 语句）中使用这个变量与字符类型的值'1' 和'' 进行比较。

```
-- 创建一个表test_pro
CREATE TABLE test_pro(id INT, title VARCHAR, remark VARCHAR);

-- 创建一个块语句进行判断并执行插入操作
DECLARE
    id_1 int;
BEGIN
    id_1 := 1;
    IF id_1 = '1' OR id_1 = '' THEN
        INSERT INTO test_pro VALUES(-id_1, '-1', 'test');
    END IF;
END;
/

-- 查询结果确认结果是否正确插入
SELECT * FROM test_pro;

ID | TITLE | REMARK |
-------------------------------------------------------------
-1 | -1| test|
```

4.4 循环语句

LOOP 语句用于构造简单的循环语句。

4.4.1 LOOP 语句

语法格式

```
LOOP
    pl_stmt_list
{END LOOP | ENDLOOP};
```

LOOP 语句不含循环退出条件，因而在语句体中应插入循环体终止语句，终止循环体的关键字为 EXIT，用户可根据条件判断终止循环，语法如下：

```
LOOP
    pl_stmt_list
    IF bool_expr THEN
        EXIT;
    END IF;
END LOOP
```

或

```
LOOP
    pl_stmt_list
    EXIT WHEN bool_expr;
END LOOP
```

参数说明

- pl_stmt_list：循环体内的一组语句，这些语句在每次循环中都会被执行。
- bool_expr：布尔表达式，当该表达式为 TRUE 时，退出循环。

示例

在块语句中定义变量参数 x 并赋值为 100，循环对其进行扩增 100 处理，并且在每次增大数据后进行条件判断，当 x 大于 1000 时退出循环体并输出 x 值。

```
DECLARE
    x INTEGER;
BEGIN
    x := 100;
    LOOP
        x := x + 100;
        IF x > 1000 THEN
            EXIT;
        END IF;
    END LOOP;
    SEND_MSG(x);
END;
/
-- 输出
1100
```

4.4.2 WHILE 语句

WHILE 语句用于构造基本的条件语句。

语法格式

```
WHILE bool_expr LOOP
    pl_stmt_list
{END LOOP | ENDLOOP};
```

WHILE 为带条件的循环语句，在每次执行循环体前，先测试条件表达式，若条件成立，则进入循环体执行，否则退出循环。

参数说明

- bool_expr：布尔表达式，当该表达式为 TRUE 时，则执行循环体内的语句；否则，退出循环。
- pl_stmt_list：循环体内的一组语句，这些语句在每次循环中都会被执行。

示例

该示例实现效果等同于 LOOP 循环示例，区别在于该示例条件判断在循环体外，即是否进入循环体由 WHILE 后的表达式决定，当 x<=1000 时才进入循环体。

```
DECLARE
    x INTEGER;
BEGIN
    x:=100;
WHILE x<=1000 LOOP
    x:=x+100;
END LOOP;
    send_msg(x);
END;

-- 输出
1100
```

4.4.3 FOR 语句

FOR 循环是一种常用的控制结构，用于遍历一系列值或从游标中获取数据。FOR 循环大致可分为三种类型：

- 指定上下边界的 FOR 循环
- 使用游标的 FOR 循环
- 使用匿名游标的 FOR 循环

4.4.3.1 指定上下边界的 FOR 循环

指定上下边界的 FOR 循环由两个数值下界 lower_bound 和上界 higher_bound 构成循环变量的取值范围。

- 若无关键字 REVERSE，则循环变量的取值将从小到大，步长为 1，直到超过最大值后，程序退出循环。
- 若有关键字 REVERSE，则循环变量从大到小，步长为-1，直到低于最小值，程序退出循环。

[2.0] **说明**
当使用 REVERSE 关键字后，后续循环定义参数必须从大到小，否则将不进入循环体。

语法格式

```
FOR ColId IN lower_bound..higher_bound LOOP
    pl_stmt_list
{END FOR | ENDFOR | ENDLOOP};
```

参数说明

- ColId：循环变量，用于存储当前的迭代值。
- lower_bound：循环的起始值。
- higher_bound：循环的结束值。
- pl_stmt_list：循环体内的一组 PL/SQL 语句。

示例定义 x 与 y 参数变量，并对 x 赋予初值 100，通过 FOR 循环进行 10 次数据叠加操作，增加数值从 10 递减到 1，最后将 x 值赋予 y 并输出 y 值。

```
DECLARE
    x INT;
    y INT;
BEGIN
    x := 100;
    FOR v_counter IN REVERSE 10 .. 1 LOOP
        x := x + v_counter;
    END LOOP;
    y := x;
    SEND_MSG(y);
END;
/

-- 输出
155
```

4.4.3.2 使用游标的 LOOP 语句

ColId 取值从游标的记录集的第一条起，到游标的记录集的最后一条止，每取一条记录值时，循环体执行一次，循环体中通常可加入对记录（由 ColId 指代）字段的引用，以实现程序对查询结果的处理。

语法格式

```
FOR ColId IN cursor_name LOOP
    pl_stmt_list
{END FOR | ENDFOR | ENDLOOP};
```

参数说明

- ColId：循环变量，用于存储从游标中提取的每一行数据。
- cursor_name：显式游标的名称。

- pl_stmt_list：循环体内的一组 PL/SQL 语句。

示例通过游标循环打印出当前用户下所有表对象的表名。使用游标循环定义时，可直接使用 variable_name.filed_name 访问游标字段。

```
-- 创 建 表
CREATE TABLE t_tables (table_name VARCHAR2(100), table_type
   VARCHAR2(50), owner VARCHAR2(50));
-- 插 入 数 据
INSERT INTO t_tables (table_name, table_type, owner) VALUES ('
   TABLE1', 'TABLE', 'OWNER1')('TABLE2', 'TABLE', 'OWNER2')('TABLE3'
   , 'VIEW', 'OWNER3');

-- 主 PL/SQL 块
DECLARE
    CURSOR cur IS
        SELECT * FROM t_tables;
BEGIN
    FOR i IN cur LOOP
        SEND_MSG(i.table_name);
    END LOOP;
END;
/

-- 输 出
TABLE1
TABLE2
TABLE3
```

4.4.3.3 使用匿名游标的 LOOP 语句

匿名游标无需在声明处定义，而使用 SELECT 子句形成匿名调用，即无需通过游标名进行调用。

语法格式

```
FOR ColId IN (select_with_parens) LOOP
    pl_stmt_list
{END FOR | ENDFOR | ENDLOOP};
```

参数说明

- ColId：循环变量，用于存储从游标中提取的每一行数据。
- select_with_parens：一个包含 SELECT 语句的表达式。
- pl_stmt_list：循环体内的一组 PL/SQL 语句。

示例

```
-- 创 建 表
CREATE TABLE t_tables (table_name VARCHAR2(100), table_type
   VARCHAR2(50), owner VARCHAR2(50));
```

```
-- 插入数据
INSERT INTO t_tables (table_name, table_type, owner) VALUES ('
   TABLE1', 'TABLE', 'OWNER1')('TABLE2', 'TABLE', 'OWNER2')('TABLE3'
   , 'VIEW', 'OWNER3');

-- 主 PL/SQL 块
DECLARE
    CURSOR cur IS
        SELECT * FROM t_tables;
BEGIN
    FOR i IN (SELECT * FROM t_tables) LOOP
        SEND_MSG(i.table_name);
    END LOOP;
END;
/

-- 输出
TABLE1
TABLE2
TABLE3
```

4.4.4 EXIT 语句和 CONTINUE 语句

EXIT 和 CONTINUE 语句用于控制循环的执行。

4.4.4.1 EXIT

EXIT 语句用于立即退出当前循环，不再执行循环体内的剩余部分，并且不再进行下一次迭代。

语法格式 EXIT 语句有两种形式：

```
EXIT;
EXIT WHEN condition;
```

参数说明

condition：布尔表达式，当条件为 TRUE 时，退出循环。

示例一个 FOR 循环用于输出从 1 到 6 的数字，但在 i 等于 5 时退出循环。

```
BEGIN
    FOR i IN 1..6 LOOP
        SEND_MSG('VALUE: ' || i);
        IF i = 5 THEN
        EXIT;
    END IF;
    END LOOP;
END;
/

-- 输出
VALUE: 1
VALUE: 2
VALUE: 3
```

```
VALUE: 4
VALUE: 5
```

4.4.4.2 CONTINUE

CONTINUE 语句用于跳过当前循环的剩余部分，并立即开始下一次迭代。

语法格式 EXIT 语句有两种形式：

```
CONTINUE;
```

示例一个 FOR 循环用于输出从 1 到 6 的数字，但在 i 等于 5 时跳过循环。

```
BEGIN
    FOR i IN 1..6 LOOP
        SEND_MSG('VALUE: ' || i);
        IF i = 5 THEN
            CONTINUE;
        END IF;
    END LOOP;
END;
/

-- 输出
VALUE: 1
VALUE: 2
VALUE: 3
VALUE: 4
VALUE: 6
```

4.5 GOTO 语句

GOTO 语句是一种控制流语句，允许程序无条件跳转到代码中的另一个标签位置。PL/SQL 中对 GOTO 语句有一些限制，对于块、循环、IF 语句而言，从外层跳转到内层是非法的。

语法格式

```
GOTO ColId;
```

参数说明 ColId：标签名，在需要跳转的位置以 « 开头，以 » 结尾，即 «ColId»>。

[2.0] **说明**
标签名不可以使用虚谷关键字，否则无法执行。

示例该示例定义了循环体，每次循环均输出一条消息，当循环次数大于 5 时，跳转至标签 ENDOFLOOP。

```
DECLARE
  v_counter INTEGER;
BEGIN
  v_counter := 1;
  LOOP
    SEND_MSG('已循环:' || v_counter || '次 ');
    v_counter := v_counter + 1;
    IF v_counter > 5 THEN
      GOTO ENDOFLOOP;
    END IF;
  END LOOP;
  <<ENDOFLOOP>>
  SEND_MSG('结束LOOP循环！ 共循环： ' || v_counter || '次 ');
END;

//输出
已循环:1次
已循环:2次
已循环:3次
已循环:4次
已循环:5次
结束LOOP循环！ 共循环： 6次
```

4.6 EXECUTE 语句

EXECUTE 语句用于调用存储过程或包，存储过程或块语句中若包含 DDL 语句，需使用特定 EXECUTE 语句（EXECUTE IMMEDIATE）进行。

主要语法结构

```
ExecuteStmt::=
[schema_name.] name_space [( func_params )]
|    EXECUTE [schema_name.] name_space
|    EXECUTE [schema_name.] name_space ( func_params )
|    EXECUTE IMMEDIATE b_expr [ USING ColId [,ColId]... ] [RETURNING
     | BULK COLLECT | RETURNING BULK COLLECT] [INTO ColId [,ColId
   ]...] [LIMIT ICONST]
```

参数信息 func_params

```
func_params::=
{b_expr | param_name => b_expr}
|    func_params , {b_expr | param_name => b_expr}
|    /*EMPTY*/
```

参数说明

- schema_name：模式名，可省略，跨模式执行存储过程/函数时需获取执行权限。
- name_space：存储过程名、存储函数名、包中定义的存储过程名或存储函数名，其中如果

调用包中存储过程或存储函数则需加包名，仅存储过程或函数可直接使用过程名或函数名进行调用。

- USING：在执行动态 SQL 语句时对参数列表进行绑定，该参数后的参数列表与动态 SQL 语句中的占位符进行绑定。
- RETURNING：将插入、更新和删除受影响的行返回，通常与 INTO 子句搭配使用，INTO 子句保存和记录插入、更新和删除受影响的行的数据，若无 INTO 子句则表示仅返回受影响的行但不记录受影响行的数据，此处仅表示单个的变量或集合数据。
- BULK COLLECT：单个集合变量或多个集合变量的多记录可使用 BULK COLLECT。
- RETURNING BULK COLLECT：可以出现在插入、更新、删除或执行即时语句中。使用该子句可将其结果集存储在一个或多个集合中。
- func_params：参数值，根据指定的参数类型进行传值，支持两种方式传值，包括直接指定参数值 b_expr、参数值与形参名绑定传值 param_name=>b_expr，其中 param_name 为形参名，同一个过程或函数参数两种方式不可混用。

[2.0] **说明**

存储过程或函数参数的类型可以指定为 in、out、in out 三种模式。

示例

- 动态 SQL

  定义一个 test_exec 表与一个 execute_proc 存储过程，存储过程含 2 个输入参数 x 与 y，过程体定义一个 string 变量用于初始化记录入库语句，然后将输入参数值赋值给入库语句进行执行。通过执行存储过程，实现将参数值 1 与 TEST 插入到 test_exec 表，最后查询表记录确认存储过程执行正确性。

```
-- 创建表
CREATE TABLE test_exec(id INT,name VARCHAR(20));
-- 创建存储过程
CREATE OR REPLACE PROCEDURE execute_proc(x IN INTEGER, y IN CHAR
   (10))
AS
    string VARCHAR2(100);
BEGIN
    string := 'INSERT INTO test_exec VALUES(?, ?);';
    EXECUTE IMMEDIATE string USING x, y;
END;
/
```

```
-- 调用存储过程
EXECUTE execute_proc(1,'TEST_EXEC');
-- 查询表
SELECT * FROM test_exec;

ID | NAME |
-----------------------------------------------------------
1 | TEST_EXEC|
```

- RETURNING INTO

  将 ID 为 100 的员工薪水提高 10%，保留旧薪水记录，使用 RETURNING INTO 将更新后的 last_name 和 salary 值存入 emp_info 记录，最后输出。

```
-- 创建表
CREATE TABLE emp_ret (employee_id INT, last_name VARCHAR(30),
   salary NUMERIC(15,3));
-- 插入数据
INSERT INTO emp_ret VALUES(100,'King',31944)(101,'Kochhar',18700)
   (102,'De Haan',18700)(103,'Hunold',9900);

DECLARE
    -- 定义一个记录类型
    TYPE emp_rec IS RECORD (
    last_name emp_ret.last_name%TYPE,
    salary emp_ret.salary%TYPE
    );
    -- 声明一个记录类型的变量
    emp_info emp_rec;
    -- 声明一个变量来存储旧的薪水
    old_salary emp_ret.salary%TYPE;
BEGIN
    -- 查询员工ID为100的旧薪水
    SELECT salary INTO old_salary FROM emp_ret WHERE employee_id
        = 100;

    -- 更新员工ID为100的薪水，并返回更新后的姓名和薪水
    UPDATE emp_ret
    SET salary = salary * 1.1
    WHERE employee_id = 100
    RETURNING last_name, salary INTO emp_info;

    -- 输出更新后的信息
    DBMS_OUTPUT.PUT_LINE('Salary of ' || emp_info.last_name || '
       raised from ' || old_salary || ' to ' || emp_info.salary);
END;
/

-- 输出结果
Salary of King raised from 31944 to 35138.4
```

- BULK COLLECT INTO

  创建一个表 emp_ret1，向其中插入一些数据，并查询特定条件下的所有员工信息。将这些

信息通过 BULK COLLECT INTO 收集到一个集合类型中，并通过循环输出每个符合条件的员工的姓名和薪水。

```
-- 创建表
CREATE TABLE emp_ret1(employee_id INT, last_name VARCHAR(30),
   salary NUMERIC(15,3));
-- 插入数据
INSERT INTO emp_ret1 VALUES(100, 'King', 31944)(101, 'Kochhar'
   , 18700)(102, 'De Haan', 18700)(103, 'Hunold', 9900);

DECLARE
    -- 定义一个集合类型，元素类型为emp_ret1表的行类型
    TYPE em_tp IS TABLE OF emp_ret1%ROWTYPE;
    -- 声明一个集合变量
    emp_tb em_tp;
    -- 定义一个字符串变量，用于存储动态SQL查询语句
    str VARCHAR(100);
BEGIN
    -- 构建动态SQL查询语句
    str := 'SELECT * FROM emp_ret1 WHERE employee_id < :id';
    -- 执行动态SQL查询，并将结果批量收集到集合中
    EXECUTE IMMEDIATE str BULK COLLECT INTO emp_tb USING 103;
    -- 遍历集合并输出每个员工的姓名和薪水
    FOR i IN 1..emp_tb.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE('Name of ' || emp_tb(i).last_name
            || ' salary is ' || emp_tb(i).salary);
    END LOOP;
END;
/

-- 输出结果
Name of King salary is 31944
Name of Kochhar salary is 18700
Name of De Haan salary is 18700
```

- RETURNING BULK COLLECT INTO 创建一个表 emp_ret2，向其中插入一些数据，并删除特定条件下的员工记录。同时使用 RETURNING BULK COLLECT INTO 将删除的记录信息收集到集合中，并输出删除的记录信息。

```
-- 创建表
CREATE TABLE emp_ret2(employee_id INT, last_name VARCHAR(30),
   salary NUMERIC(15,3));
-- 插入数据
INSERT INTO emp_ret2 VALUES(100, 'King', 31944)(101, 'Kochhar'
   , 18700)(102, 'De Haan', 18700)(103, 'Hunold', 9900);

DECLARE
    -- 定义集合类型
    TYPE num_list IS TABLE OF emp_ret2.employee_id%TYPE;
    TYPE num_list1 IS TABLE OF emp_ret2.last_name%TYPE;
    -- 声明集合变量
    enums num_list;
    names num_list1;
```

```
BEGIN
    -- 删除记录并收集返回值
    DELETE FROM emp_ret2
    WHERE employee_id < 102
    RETURNING employee_id, last_name BULK COLLECT INTO enums,
       names;
    -- 输出删除记录的数量
    DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows:');
    -- 遍历集合并输出每个删除记录的信息
    FOR i IN enums.FIRST .. enums.LAST LOOP
        DBMS_OUTPUT.PUT_LINE('Employee #' || enums(i) || ': ' ||
           names(i));
    END LOOP;
END;
/

-- 输出结果:
Deleted 2 rows:
Employee #100: King
Employee #101: Kochhar
```

4.7 静态 SQL

静态 SQL 是指在编译应用程序时就已经确定并嵌入代码中的 SQL 语句。这类 SQL 语句在程序运行前就已经被预编译和优化，因此在执行时无需再进行解析和编译，直接由数据库执行。适用于简单的数据处理任务，如查询、插入、更新和删除等基本操作。

静态 SQL 和普通 SQL 具有相同的语法。但在查询语句中，静态 SQL 语句中支持 SELECT INTO 语法，而普通 SQL 中不支持该语法。

静态 SQL 有以下特点：

- 静态 SQL 在应用程序编译阶段就已经被转换成数据库可识别的格式，执行效率高。
- SQL 语句固定，减少了 SQL 注入攻击的风险，静态 SQL 的 SQL 语句清晰可见，便于维护和调试。
- 静态 SQL 在执行时通常比动态 SQL 更快，性能更好，但无法根据运行时条件动态生成 SQL 语句，限制了应用的灵活性。

示例

创建一个员工表，并插入数据，使用游标查询所有员工的信息并输出。

```
CREATE TABLE employees (id NUMBER PRIMARY KEY, name VARCHAR2(100),
   position VARCHAR2(100), salary NUMBER);

INSERT INTO employees (id, name, position, salary) VALUES (1, 'xx'
   , '软件工程师', 8000)(2, 'yy', '项目经理', 9000)(3, 'zz', '测试工
```

```
    程 师', 7500);

DECLARE
    -- 定 义 游 标
    CURSOR emp_cursor IS
    SELECT id, name, position, salary FROM employees;
    -- 定 义 变 量 来 存 储 查 询 结 果
    v_id employees.id%TYPE;
    v_name employees.name%TYPE;
    v_position employees.position%TYPE;
    v_salary employees.salary%TYPE;
BEGIN
-- 打 开 游 标
    OPEN emp_cursor;
    -- 循 环 读 取 数 据
    LOOP
        FETCH emp_cursor INTO v_id, v_name, v_position, v_salary;
        EXIT WHEN emp_cursor%NOTFOUND;
    -- 输 出 结 果
        DBMS_OUTPUT.PUT_LINE('ID: ' || v_id ||', Name: ' || v_name
            ||', Position: ' || v_position ||', Salary: ' ||
          v_salary);
    END LOOP;
    -- 关 闭 游 标
    CLOSE emp_cursor;
END;
/

-- 输 出
ID: 1, Name: xx, Position: 软 件 工 程 师, Salary: 8000
ID: 2, Name: yy, Position: 项 目 经 理, Salary: 9000
ID: 3, Name: zz, Position: 测 试 工 程 师, Salary: 7500
```

[2.0] **说明**

有关游标的详细信息请参见游标管理章节。

4.8 动态 SQL

动态 SQL 是一种在运行时生成和运行 SQL 语句的编程方法。

PL/SQL 提供了两种编写动态 SQL 的方法：原生动态 SQL 和 DBMS_SQL 包。关于后者的使用，可参阅相应文档：DBMS_SQL 包。

相比静态 SQL，动态 SQL 通过损失一定的性能，换取更大的灵活性。动态 SQL 由于包含变化的部分，因此无法以普通语句的方式执行；而是在语句组装好后，以 EXECUTE IMMEDIATE 方式执行。

原生动态 SQL 编写的代码比使用 DBMS_SQL 包写出的等效代码更易读，并且运行速度也更

快。但是用原生动态 SQL 代码必须在编译时知道输入和输出变量的数量和数据类型；如果在编译时还不能确定此类信息，则只能使用 DBMS_SQL 包。

示例

清空用户下所有表，由于用户下的表是在运行时查询出来的，因此需要动态 SQL 在运行时得到最终的执行语句。

```
-- 在用户下创建表
CREATE TABLE table1 (id NUMBER PRIMARY KEY, name VARCHAR2(100));
CREATE TABLE table2 (id NUMBER PRIMARY KEY, description VARCHAR2
   (100));

-- 插入数据
INSERT INTO table1 (id, name) VALUES (1, '记录1');
INSERT INTO table2 (id, description) VALUES (1, '描述1');

-- 查询表信息
SELECT*FROM TABLE1, TABLE2;

ID | NAME | ID | DESCRIPTION |
-------------------------------------------------------------
1| 记录1| 1| 描述1|

-- 使用动态SQL清空所有表
DECLARE
    v_sql VARCHAR2(200);
BEGIN
    -- 查询用户模式下的所有表名
    FOR t IN (SELECT table_name FROM user_tables) LOOP
        -- 构建动态SQL语句
        v_sql := 'TRUNCATE TABLE ' || t.table_name;

        -- 输出生成的SQL语句
        SEND_MSG(v_sql);

        -- 执行动态SQL语句
        EXECUTE IMMEDIATE v_sql;
    END LOOP;
END;
/

TRUNCATE TABLE TABLE1
TRUNCATE TABLE TABLE2

-- 查询表已被清空
SELECT*FROM TABLE1, TABLE2;

ID | NAME | ID | DESCRIPTION |
-------------------------------------------------------------
```

[2.0] 说明

关于动态 SQL 的其他示例，可参阅4.6EXECUTE 语句章节。

4.9 CASE 语句

CASE 语句在 PL/SQL 中用于根据不同的条件执行不同的操作。

语法格式

```
CaseStmt::=
CASE [case_value] {When_item [When_item]...} {END CASE | ENDCASE} ;
|    CASE [case_value] {When_item [When_item]...} ELSE pl_stmt_list
    {END CASE | ENDCASE} ;

When_item::=
WHEN bool_expr THEN pl_stmt_list
```

参数说明

- case_value：可选参数。case_value 表达式为 When_item 中的条件清单，该参数类型同 When_item 中条件清单类型一致，通过与 When_item 中条件清单比较，若相同则为 true 并返回第一个符合条件的值，剩下的 case 部分将会被自动忽略。
- WHEN bool_expr THEN pl_stmt_list：用于选择判断，类似于 IF..ELSE。在执行时先对条件进行判断，然后根据判断结果做出相应 SQL 操作，case 函数满足某个条件后，剩下的条件将会被自动忽略。因此，即使满足多个条件，也仅执行第一个条件下的 SQL 操作。
- CASE：实现一个复杂的条件构造。如果 bool_expr 求值为真，相应的 SQL 被执行。如果没有搜索条件匹配，在 ELSE 子句里的语句被执行，该语句与 CASE 表达式不同的是 CASE 语句不允许 ELSE NULL 子句，且结束标签为 END CASE。

示例

- 简单 CASE 表达式

  声明一个变量 id 作为 case_value，根据变量 id 的值输出不同的字符串。

```
DECLARE
id INT:=5;
BEGIN
    CASE id
        WHEN 1 THEN
            DBMS_OUTPUT.PUT_LINE('one');
        WHEN 2 THEN
            DBMS_OUTPUT.PUT_LINE('two');
        WHEN 3 THEN
```

```
            DBMS_OUTPUT.PUT_LINE('three');
        WHEN 4 THEN
            DBMS_OUTPUT.PUT_LINE('four');
        WHEN 5 THEN
            DBMS_OUTPUT.PUT_LINE('five');
        WHEN 6 THEN
            DBMS_OUTPUT.PUT_LINE('others');
    END CASE;
END;
/

-- 输出结果:
five
```

- 搜索 CASE 表达式

  定义一个存储函数 case_test_proc，存储函数接受一个输入参数，函数体通过 case 语句对输入参数进行判断处理，根据输入参数所属范围返回其对应级别。

```
-- 定义存储函数
CREATE OR REPLACE FUNCTION case_test_proc(x IN INTEGER) RETURN
   VARCHAR AS
    grade VARCHAR;
BEGIN
    CASE
        WHEN x >= 0 AND x <= 59 THEN
            grade := '不及格';
        WHEN x >= 60 AND x <= 69 THEN
            grade := '及格';
        WHEN x >= 70 AND x <= 79 THEN
            grade := '中';
        WHEN x >= 80 AND x <= 89 THEN
            grade := '良';
        WHEN x >= 90 AND x <= 100 THEN
            grade := '优';
        WHEN x > 100 OR x < 0 THEN
            grade := '输入错误!';
    END CASE;
    RETURN grade;
END;
-- 调用函数并传入参数90
SELECT case_test_proc(90);

EXPR1 |
------------------------------------------------------------
优|
```

4.10 NULL 语句

空语句，不产生任何执行动作，可以充当需要可执行语句的位置的占位符。

语法格式

```
NULL;
```

示例

在循环中，如果暂时不需要执行任何操作，可以使用 NULL 语句。

```
DECLARE
    v INT := 1;
BEGIN
    WHILE v <= 3 LOOP
        IF v = 2 THEN
            -- 占位符，表示当v等于2时不做任何操作
            NULL;
        ELSE
            DBMS_OUTPUT.PUT_LINE('V: ' || v);
        END IF;
        v := v + 1;
    END LOOP;
END;
/

-- 输出
V: 1
V: 3
```

4.11 异常处理语句

异常处理语句共包括三个部分：

- 异常定义
- 抛出异常
- 异常处理

4.11.1 异常定义 ExceptionDef

语法格式

```
ExceptionDef::=
exception_name EXCEPTION
```

参数说明

- ExceptionDef：用户自定义异常语句，使用关键字 EXCEPTION 在 DECLARE 中定义。
- exception_name：用户自定义异常名称。

4.11.2 抛出异常 ThrowStmt

语法格式

```
ThrowStmt::=
{RAISE | THROW} [EXCEPTION] exception_name;
```

参数说明

- ThrowStmt：抛出异常语句，抛出异常语句在块语句体或过程体中使用，可使用关键字 RAISE 或 THROW。
- b_expr：附带错误信息。

4.11.3 异常处理 OptExceptionStmt

语法格式

```
OptExceptionStmt::=
EXCEPTION Exception_Item [Exception_Item...]

Exception_Item::=
WHEN exception_name THEN pl_stmt_list
|    WHEN ICONST THEN pl_stmt_list
```

可以有多个 WHEN 子句，每个子句处理一种特定的异常。使用 WHEN OTHERS THEN 可以捕获所有未被捕获的异常。

参数说明

- ICONST：预定义的系统异常，如 ERR_MORE_DATA_FOUND、ERR_NO_DATA_FOUND 等。
- pl_stmt_list：异常发生时执行的 PL/SQL 语句列表。

4.11.4 示例

- 示例 1

  定义一个常数变量与两个异常定义的块语句，当常数变量 x 大于 0 时抛出异常 exce_1，否则抛出 exce_2；对应 exce_1 的异常处理输出消息“raise exception 1”，对应 exce_2 异常处理输出消息“raise exception 2”，其他异常则输出“others exception”。

```
DECLARE
    x INT;
    exce_1 EXCEPTION;
    exce_2 EXCEPTION;
BEGIN
    x:=1;
    IF x>0 THEN
```

```
        RAISE exce_1;
    ELSE
        RAISE exce_2;
    END IF;
    EXCEPTION
        WHEN exce_1 THEN
        DBMS_OUTPUT.PUT_LINE('raise exception 1');
        WHEN exce_2 THEN
        DBMS_OUTPUT.PUT_LINE('raise exception 2');
        WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('others exception');
END;
/
-- 输出:
raise exception 1
```

- 示例 2

  定义错误信息直接抛出错误不进入异常处理流程。

```
DECLARE
    exec EXCEPTION;
BEGIN
    IF 1!=2 THEN
        RAISE exec '数据错误';
    END IF;
END;
/

-- 输出:
Error: [E50001 L5 C1] 数据错误
```

> **⚠ 注意**
>
> 自 V12.7 版本起，支持预定义异常：DUP_VAL_ON_INDEX、NO_DATA_FOUND、TOO_MANY_ROWS、ZERO_DIVIDE。
> V12.7 之前的版本使用 ERR_DUP_VAL_ON_INDEX、ERR_NO_DATA_FOUND、ERR_MORE_DATA_FOUND、ERR_DIV_ZERO 替代。

- 示例 3

  存储过程直接使用系统预定义的异常。捕获异常后，执行相应的处理。

```
SQL> CREATE TABLE EMP(emp_id INT,name VARCHAR(32), age SMALLINT);

SQL> CREATE OR REPLACE PROCEDURE test_proc(id INT) AS
temp_name VARCHAR(32);
BEGIN
-- 查询并返回数据
SELECT name INTO temp_name FROM EMP WHERE emp_id = id;
DBMS_OUTPUT.PUT_LINE(temp_name);
EXCEPTION
```

```
-- 异常处理
WHEN NO_DATA_FOUND THEN -- 没有找到数据
DBMS_OUTPUT.PUT_LINE('没有找到该员工');
WHEN TOO_MANY_ROWS THEN -- 找到多条数据
DBMS_OUTPUT.PUT_LINE('找到多个员工: ' || temp_name);
WHEN OTHERS THEN -- 其他异常
DBMS_OUTPUT.PUT_LINE('发生其它错误');
END;
/

SQL> EXEC test_proc(1);
没有找到该员工
```

4.12 EXCEPTION_INIT 语句

PRAGMA EXCEPTION_INIT 将异常名称与相应错误号关联，通过关联的错误号捕获异常信息，对于未命名的内部异常则可使用该方式为其添加异常名称，并为其编写一个特定的处理程序，而不是使用 OTHERS 处理程序。

语法格式

```
PRAGMA EXCEPTION_INIT(err_name, err_code);
```

参数说明

- err_name：当前 PL/SQL 语句、子程序或包中声明的用户自定义异常名。
- err_code：任何有效的虚谷数据库错误号，该错误号与函数 SQLCODE 返回的错误号一致。

> ⚠ 注意
>
> [2.0]
> - 错误号取值范围为 [1,99999]。
> - PRAGMA 必须与 EXCEPTION_INIT 同时出现。

示例

- 示例 1 自定义异常名与数据库定义错误号 E16005 关联

```
CREATE TABLE tb_ex(id INT PRIMARY KEY);

DECLARE
    no_null EXCEPTION;
    PRAGMA EXCEPTION_INIT(no_null,16005);
BEGIN
    INSERT INTO tb_ex VALUES(NULL);
```

```
EXCEPTION
    WHEN no_null THEN
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/

-- 输 出:
字 段ID不 能 取 空 值
```

- 示例 2 自定义错误号。

```
DECLARE
    ud_errc EXCEPTION;
    PRAGMA EXCEPTION_INIT (ud_errc,99996);
BEGIN
    RAISE ud_errc;
EXCEPTION
    WHEN ud_errc THEN
        SEND_MSG(SQLCODE);
END;
/

-- 输 出:
99996
```

4.13 PREPARE 语句

预编译需要执行的 SQL 语句。

语法格式

预编译 SQL 语句 PREPARE

```
PREPARE ColId AS sql_stmt
```

释放预编译的 SQL 语句 DEALLOCATE

```
DEALLOCATE ColId
```

参数说明

- ColId：定义的预编译名称。
- sql_stmt：预编译的执行 SQL 语句。

⚠ 注意

[2.0]

- PREPARE 预编译支持 DML 语句。
- PREPARE 预编译支持 DDL 语句。支持的对象范围目前仅包括表/视图，其他对象 PREPARE 只支持 DROP 操作。
- 建议在使用预编译语句完成后释放预编译语句，减少资源浪费，可通过 DEALLOCATE 语句释放预编译语句，也可通过关闭当前会话自动释放。

示例

使用 PREPARE 语句来准备和执行 DDL 和 DML 语句，创建表并插入数据。

```
-- 预编译DDL语句。
PREPARE pre_1 AS CREATE TABLE pre_tab(col1 INT,col2 VARCHAR);
-- 执行pre_1
?pre_1
-- 预编译DML语句。
PREPARE pre_2 AS INSERT INTO pre_tab values(1,'VAL1');
-- 执行pre_2
?pre_2

SELECT * FROM pre_tab;

COL1 | COL2 |
-------------------------------------------------------------
1 | VAL1|
```

4.14 事务管理

4.14.1 概述

在 PL/SQL 中，事务控制语句用于管理事务的开始、结束以及回滚。事务是一组要么全部执行，要么全部不执行的 SQL 语句。事务控制确保了数据库的一致性和完整性。

虚谷数据库中命令执行均与事务相关，通过 COMMIT 或 ROLLBACK 子句可提交或回滚事务。另可通过连接设置或发送命令 SET AUTO_COMMIT ON/OFF 设置当前连接的事务自动提交模式。

语法格式

开始一个新的事务：

```
BEGIN [ WORK | TRAN | TRANSACTION ]
```

提交当前事务，使所有更改永久化：

```
COMMIT [ WORK | TRAN | TRANSACTION ] [AND [NO] CHAIN]
```

在当前事务中设置一个保存点。之后的部分事务可以被独立地回滚，而不影响整个事务。

```
SAVEPOINT ColId
```

回滚当前事务，撤销所有已进行的更改：

```
ABORT [ WORK | TRAN | TRANSACTION ]
```

或

```
ROLLBACK [ WORK | TRAN | TRANSACTION ] [AND [NO] CHAIN]
```

将当前事务回滚到指定的保存点。ColId 应该是一个之前通过 SAVEPOINT 命令创建的保存点名称。

```
ROLLBACK TO [ SAVEPOINT ] ColId
```

[2.0] ⚠ 注意

非自动提交模式下，用户需显式指定提交或回滚以保证数据落盘或丢弃；DDL 语句会自动提交事务，不能被回滚，如果语句结构中存在 DDL 语句，则当 DDL 命令执行后将默认将前序命令全部提交，即使发送 ROLLBACK 也不会回滚。

参数说明

- WORK | TRAN | TRANSACTION：可选关键词，这三个词在使用时是等价的，都是为了增强语句的可读性，不影响事务运行。
- AND CHAIN：当事务被提交或回滚后，立即开始一个新的事务。
- AND NO CHAIN：当事务被提交或回滚后，不自动开始新的事务。
- ColId: 保存点的名称，用于在事务中标记出可以回退到的点。

示例

```
-- 建表
CREATE TABLE t_orders (
    id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    amount DECIMAL(10, 2) NOT NULL
);
--插入数据
INSERT INTO t_orders (id, name, amount) VALUES (1, 'Alice', 1200);
INSERT INTO t_orders (id, name, amount) VALUES (2, 'Bob', 800);

-- 开始第一个事务
BEGIN;

-- 更新Alice的订单金额
UPDATE t_orders SET amount = 1500 WHERE name = 'Alice';
```

```
-- 设置保存点
SAVEPOINT update_alice;

-- 尝试更新Bob的订单金额（这里出现了一个错误，Bob金额为1000）
UPDATE t_orders SET amount = 900 WHERE name = 'Bob';

-- 假设在这里检测到了一个错误
-- 回滚到保存点
ROLLBACK TO SAVEPOINT update_alice;

-- 重新插入Bob金额
UPDATE t_orders SET amount = 1000 WHERE name = 'Bob';

-- 提交第一个事务并启动新的事务
COMMIT AND CHAIN;

-- 新的事务已经开始
-- 插入一条新的订单记录
INSERT INTO t_orders (id, name, amount) VALUES ('Charlie', 300);

-- 提交第二个事务
COMMIT;

SELECT*FROM t_orders;
ID | NAME | AMOUNT |
-----------------------------------------------------------
1 | Alice| 1500|
2 | Bob| 1000|
3 | Charlie| 300|
```

4.14.2 匿名事务

匿名事务，又称自治事务，是独立的实体，它不与主事务共享锁、资源或依赖等，可用于子程序、对象的方法等。匿名事务可以进行 SQL 操作，并提交或回滚这些操作，同时不影响主事务中的提交和数据。

例如，主事务中发生事务回滚，从匿名事务之后的某处回滚至匿名事务之前的保存点，其中主事务中的操作被回滚，但匿名事务中的操作不会被回滚。

示例主事务被回滚，主事务中的插入操作 INSERT INTO tab_person VALUES('Alice', 10); 被撤销，但匿名事务中的插入操作 INSERT INTO tab_person VALUES('Bob', 20); 不会被撤销

```
CREATE TABLE tab_person(name VARCHAR(10), age INT);

-- 执行匿名事务存储过程
CREATE PROCEDURE proc_anonymous_insert() as
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO tab_person VALUES('Bob', 20);
    COMMIT;
END;
```

```
/

-- 主事务
BEGIN
    -- 主事务中的此插入将会被回滚
    INSERT INTO tab_person VALUES('Alice', 10);
    -- 匿名事务中的语句不会被回滚
    proc_anonymous_insert();
    ROLLBACK;
END;
/

SELECT * FROM tab_person;

NAME | AGE |
-------------------------------------------------------------
Bob| 20 |
```

4.14.3 事务中的 DDL

事务的开始和结束

事务开始：一个事务在遇到第一个可执行的 SQL 语句时开始。

- DML 语句
- DDL 语句
- SET TRANSACTION 语句

事务结束：一个事务在遇到以下情况时结束。

- COMMIT
- （不指定 SAVEPOINT 的）ROLLBACK
- DDL 语句
- 用户（正常或异常）退出

如果事务中有 DDL 语句，需要格外注意，因为会隐性破坏该事务。例如在 delete 表 1 中的数据后，进行 drop 表 2 的操作，然后需要 rollback 对表 1 的 delete 操作，但实际执行结果是并未回退 delete 操作。因为在 drop 操作之前的事务已被隐性结束并提交，而后续的 rollback 操作，回退的是 drop 后的 DML 语句。

示例：

```
CREATE TABLE tab_trans_1(id INT, c2 VARCHAR);
CREATE TABLE tab_trans_2(id INT);

INSERT INTO tab_trans_1 VALUES(1, 'a');
```

```
INSERT INTO tab_trans_1 VALUES(2, 'b');
INSERT INTO tab_trans_1 VALUES(3, 'c');

-- 包含 DDL 语句的事务
BEGIN
    DELETE FROM tab_trans_1 WHERE id > 1;
    DROP TABLE tab_trans_2;
    ROLLBACK;
END;
/

-- 查询结果
SELECT * FROM tab_trans_1;

ID | C2 |
-------------------------------------------------------------
1 | a|
```

4.15 PL/SQL 子程序

4.15.1 概述

PL/SQL 子程序是一个带命名的过程或函数，可以重复调用。使用 PL/SQL 子程序可以将程序模块化管理，并且能够减少网络开销，提高性能。

包头中声明并在包体中定义子程序，该子程序为包的子程序，存储在数据库中，直到包被删除。

示例

- 包的子程序。

  在模式级别使用 CREATE FUNCTION 或者 CREATE PROCEDURE 语句创建的子程序是一个独立子程序，可以分为存储过程和存储函数。独立子程序存储在数据库中，直到该存储过程和存储函数被删除。

```
--创建包头
CREATE OR REPLACE PACKAGE pack_1
AS
    PROCEDURE pro_1(c1 INT,c2 VARCHAR2);        -- 在包头中声明子
       程序
    FUNCTION func_1(c1 INT,c2 VARCHAR2) RETURN INT;
    -- 在包头中声明子程序
END;

--创建包体
CREATE OR REPLACE PACKAGE BODY pack_1
AS
    PROCEDURE pro_1(c1 INT,c2 VARCHAR2)         -- 在包体中定义子
       程序
```

```
    AS
    id INT;
    name VARCHAR2;
    BEGIN
        name:=c2;
        SEND_MSG('c2='||name);
    END pro_1;
    FUNCTION func_1(c1 INT,c2 VARCHAR2) RETURN INT
    -- 在包体中定义子程序
    AS
    id INT;
    name VARCHAR2;
    BEGIN
        id:=c1;
        RETURN id;
    END func_1;
END pack_1;
/
```

- 存储过程。

```
CREATE OR REPLACE PROCEDURE pro_1(id INT,name INT)
AS       -- 声明区开始
    v1 INT;
BEGIN      -- 声明区开始
    v1:=1;    -- 执行语句
END pro_1;   -- 声明区结束
```

- 存储函数。

```
CREATE OR REPLACE FUNCTION func_1(id INT,name INT) RETURN INT
AS       -- 声明区开始
    v1 INT;
BEGIN      -- 声明区开始
    v1:=1;    -- 执行语句
    RETURN v1;
END func_1;   -- 声明区结束
/
```

4.15.2 嵌套子程序

在匿名块内、包内或模式级别创建的子程序是嵌套子程序，只有在包的子程序或独立子程序中的嵌套子程序是存储在数据库中，匿名块内创建的嵌套子程序仅在当前块内有效。

⚠ 注意

[2.0] 嵌套子程序结束时必须显式指示结束的程序名。

示例

- 在匿名块中嵌套子程序。

```
DECLARE     -- 匿名块的声明区开始
    v1 INT;
    v2 VARCHAR2;
BEGIN     -- 匿名块的执行区开始
    PROCEDURE subpro_1(c1 INT,c2 VARCHAR2)-- 声明并定义嵌套子过程
    IS                                  -- 嵌套子过程声明区开始
        sub_v1 INT;
        sub_v2 VARCHAR2(30) := ' ';
BEGIN                                  -- 嵌套子过程执行区开始
        sub_v1 := v1 + c1;
        sub_v2 := v2 || c2;
        v1:=sub_v1;
        v2:=sub_v2;
        SEND_MSG('v1='||sub_v1);
        SEND_MSG('v2='||sub_v1);
    END subpro_1;                       -- 嵌套子过程执行区结束
    v1 := 1;
    v2 := 'AA';
    subpro_1(v1,v2);
END;   -- 匿名块的执行区结束
/
```

- 在包的子程序中嵌套子程序。

```
--创建包头
CREATE OR REPLACE PACKAGE pack_2
AS
    v1 INT;
    v2 VARCHAR2;
    PROCEDURE pro_1(c1 INT,c2 VARCHAR2);          -- 在包头中声明子
       程序
    FUNCTION func_1(c1 INT,c2 VARCHAR2) RETURN INT;
    -- 在包头中声明子程序
END;

--创建包体
CREATE OR REPLACE PACKAGE BODY pack_2
AS
    PROCEDURE pro_1(c1 INT,c2 VARCHAR2)         -- 在包体中定义子
       程序
    AS
    id INT;
    name VARCHAR2;
    BEGIN
        PROCEDURE subpro_1(c1 INT,c2 VARCHAR2)-- 声明并定义嵌套子
           过程
        IS                                  -- 嵌套子过程声明区开始
            sub_v1 INT;
            sub_v2 VARCHAR2(30) := ' ';
        BEGIN                                 -- 嵌套子过程执行区开始
            sub_v1 := v1 + c1;
            sub_v2 := v2 || c2;
            v1:=sub_v1;
            v2:=sub_v2;
```

```
            SEND_MSG('v1='||sub_v1);
            SEND_MSG('v2='||sub_v1);
        END subpro_1;                    -- 嵌套子过程执行区结束
        v1 := 1;
        v2 := 'sub_procedure';
        subpro_1(v1,v2);
    END pro_1;
    FUNCTION func_1(c1 INT,c2 VARCHAR2) RETURN INT
    -- 在包体中定义子程序
    AS
    id INT;
    name VARCHAR2;
    BEGIN
        FUNCTION subfunc_1(c1 int,c2 VARCHAR2) RETURN INT
        IS                               -- 嵌套子过程声明区开始
            sub_v1 INT;
            sub_v2 VARCHAR2(30) := ' ';
        BEGIN                              -- 嵌套子过程执行区开始
            sub_v1 := v1 + c1;
            sub_v2 := v2 || c2;
            v1:=sub_v1;
            v2:=sub_v2;
            SEND_MSG('v1='||sub_v1);
            SEND_MSG('v2='||sub_v1);
            RETURN sub_v1;
        END subfunc_1;                   -- 嵌套子过程执行区结束
        v1 := 1;
        v2 := 'sub_function';
        v1:=subfunc_1(v1,v2);               -- 执行嵌套子过程
    END func_1;
END pack_2;
/
```

- 在独立程序中嵌套子程序。

```
CREATE OR REPLACE FUNCTION func_1(c1 INT,c2 VARCHAR2) RETURN INT
    AS
    id INT;
    name VARCHAR2;
    BEGIN
        FUNCTION subfunc_1(c1 int,c2 VARCHAR2) RETURN INT
        IS                               -- 嵌套子过程声明区开始
        sub_v1 INT;
        sub_v2 VARCHAR2(30) := ' ';
    BEGIN                                -- 嵌套子过程执行区开始
        sub_v1 := id + c1;
        sub_v2 := name || c2;
        id:=sub_v1;
        name:=sub_v2;
        SEND_MSG('v1='||sub_v1);
        SEND_MSG('v2='||sub_v1);
        RETURN sub_v1;
    END subfunc_1;                       -- 嵌套子过程执行区结束
    id := 1;
    name := 'sub_function';
```

```
    RETURN subfunc_1(id,name);          -- 执行嵌套子过程
END func_1;
/
```

4.15.3 嵌套子程序的调用

嵌套子程序的调用方式可以分为：递归调用、同级调用、父级调用、子级调用四种调用方式，在调用嵌套子程序前必须先声明和定义。

> ⚠ 注意
>
> [2.0] 虚谷数据库嵌套子程序不支持重载（优先调用第一个满足名字的同名子程序）。

示例

- 递归调用。

```
DECLARE
    TYPE t_varr IS VARRAY(20) OF INT;
    v_varr t_varr;
    ret INT;
    idex INT:=8;
    BEGIN
        FUNCTION find_key(h IN INT,e IN INT,key_val INT) RETURN
           INT
        -- 声明并定义嵌套子过程
        IS
            d_varr t_varr;
            mid INT;
    BEGIN                                   -- 嵌套子过程执行区开始
        d_varr:=v_varr;
        mid:=(h+e)/2;
        IF (h>e) THEN
            RETURN -1;
        ELSEIF (d_varr(mid)=key_val) THEN
            RETURN mid;
        ELSEIF (find_key(h,mid-1,key_val)=-1) THEN
        -- 递归方式执行嵌套子过程
            RETURN find_key(mid+1,e,key_val);
            -- 递归方式执行嵌套子过程
        END IF;
    END find_key;                           -- 嵌套子过程执行区结束

    v_varr:=t_varr(15,2,7,6,8,3,25,8);
    FOR i IN 1..v_varr.COUNT() LOOP
        SEND_MSG('varr('||i||')='||v_varr(i));
    END LOOP;
    ret:=find_key(1,v_varr.COUNT(),idex); -- 调用下级嵌套子过程
    if(ret=-1) THEN
        SEND_MSG('Not find idex '||idex);
    ELSE
```

```
        SEND_MSG('Find idex '||idex||' in '||ret);
    ENDIF;
END;
/

-- 输出
varr(1)=15
varr(2)=2
varr(3)=7
varr(4)=6
varr(5)=8
varr(6)=3
varr(7)=25
varr(8)=8
Find idex 8 in 5
```

- 调用同级子过程。

```
DECLARE
    TYPE t_varr IS VARRAY(20) OF INT;
    v_varr t_varr;
    BEGIN
        FUNCTION find_key(h IN INT,e IN INT,key_val INT) RETURN
           INT
        -- 声明并定义嵌套子过程
        IS
            d_varr t_varr;
            mid INT;
        BEGIN                                    -- 嵌套子过程执行区
           开始
            d_varr:=v_varr;
            mid:=(h+e)/2;
            IF (h>e) THEN
                RETURN -1;
            ELSEIF (d_varr(mid)=key_val) THEN
                RETURN mid;
            ELSEIF (find_key(h,mid-1,key_val)=-1) THEN
                RETURN find_key(mid+1,e,key_val);
            END IF;
        END find_key;                            -- 嵌套子过程执行区
           结束

        PROCEDURE find_res(idex IN INT)          -- 声明并定义嵌套子
           过程
        IS
        ret INT;
        BEGIN
            ret:=find_key(1,v_varr.COUNT(),idex);
            -- 同级方式执行嵌套子过程
            if(ret=-1) THEN
                SEND_MSG('Not find idex '||idex);
            ELSE
                SEND_MSG('Find idex '||idex||' in '||ret);
            ENDIF;
        END find_res;                            -- 嵌套子过程执行区
```

```
            结 束

        v_varr:=t_varr(15,2,7,6,8,3,25,8);
        find_res(2);
END;
/

-- 输 出
Find idex 2 in
```

- 调用父级过程。

```
BEGIN
    -- 声 明 和 定 义 第 一 级 子 过 程
    PROCEDURE send_father(id IN INT) IS
    BEGIN
        -- 声 明 和 定 义 第 二 级 子 过 程
        PROCEDURE send_2() IS
        BEGIN
            send_father(6);
        END send_2;
        -- 调 用 SEND_MSG 子 过 程
        SEND_MSG(id);
    END send_father;
    -- 调 用 第 一 级 子 过 程
    send_father(2);
END;
/

-- 输 出
2
```

5 数据库管理

5.1 概述

第一次启动虚谷数据库服务程序时，系统默认创建系统库 (system)，它除了用户库应有的功能外，还存储所有非系统库的控制信息以及整个系统的控制与描述信息，所有用户库的创建与删除必须在系统库中进行。

5.2 字符集

概述

GB18030（全称《信息技术中文编码字符集》）是由中国制定的中文编码标准，覆盖七万余汉字及符号，涵盖中日韩汉字、少数民族文字等。该标准完全兼容 GB2312（《信息交换用汉字编码字符集》），并向下兼容 GBK（《汉字内码扩展规范》）。

GB18030 历经三次版本迭代：

- GB18030-2000：初版标准，奠定基础编码框架。
- GB18030-2005：扩展字符集，新增部分 CJK 统一汉字及特殊符号。
- GB18030-2022：修订 Unicode 映射规则，调整 18 个字符的编码对应关系，解决早期版本与 Unicode 的兼容性问题。

Unicode 映射的兼容性挑战：

- GB18030-2005 的临时方案：部分字符因 Unicode 未分配标准码位，临时映射至 BMP 平面的 PUA（Private Use Area，私有使用区）。
- GB18030-2022 的改进：随 Unicode 5.0 引入新码位，18 个字符的映射关系被重新定义，导致新旧版本在 Unicode 兼容性上产生差异。

虚谷数据库的编码支持策略：

- 新增 GB18030_2022 字符集，严格遵循 2022 版标准，优化与 Unicode 的映射逻辑。
- 保留 GB18030_2005 字符集，确保历史系统的兼容性。

虚谷数据库专注于 GB18030-2022 编码与 Unicode 编码的映射关系，不涉及新字符字形的展示，而是运算存储这些编码。

虚谷数据库专注于 GB18030-2022 编码转换与存储，不涉及新字符字形渲染，保障数据运算与跨系统交互的准确性。

使用 GB18030_2022 字符集

虚谷数据库目前支持库级字符集，创建库时，指定 GB18030_2022 字符集即可。

```
SQL> CREATE DATABASE DB_TEST CHARACTER SET 'GB18030_2022';
SQL> SELECT DB_NAME, CHAR_SET FROM DBA_DATABASES WHERE DB_NAME = '
   DB_TEST';

DB_NAME | CHAR_SET |
-----------------------------------------------------------
DB_TEST| GB18030_2022|
```

> **说明**
>
> - GB18030_2022 为虚谷数据库定义字符集名。
> - 客户端程序（如 JDBC 连接数据库的程序）在接收数据时应当使用操作系统支持的字符集。

5.3 创建数据库

语法格式

```
CREATE DATABASE [IF NOT EXISTS] database_name
[ CHARACTER SET ColId [, Sconst] ]
[ TIME ZONE ]
[ DISABLE [, ENABLE] ENCRYPT ]
[ ENCRYPT BY Sconst ]
```

参数说明

- database_name：数据库名。
- ColId[,Sconst]：字符集名称。字符集的查询在“sys_charsets”系统表中。
- TIME ZONE：时区，其形式为’gmt+hh:mm’或’gmt-hh:mm’，表示当前时区与格林威治时间的时差。
- DISABLE[,ENABLE] ENCRYPT：是否加密。
- Sconst：指定加密机名。

示例

创建一个名为 dbtest 的数据库，其字符集是 GBK，时区是东 8 区。

```
SQL> CREATE DATABASE dbtest CHARACTER SET 'GBK' TIME ZONE 'GMT
   +08:00';
```

创建数据库时存在同名数据库，但原数据库和创建库字符类型不同。

```
SQL> CREATE DATABASE db_fea CHAR SET 'GB18030';

SQL> SELECT db_name,char_set from dba_databases WHERE db_name='
   DB_FEA';

DB_NAME | CHAR_SET |
-----------------------------------------------------------
DB_FEA| GB18030|

-- 创建一个与原数据库字符类型不同的数据库，此处会返回警告
SQL> CREATE DATABASE IF NOT EXISTS db_fea CHAR SET 'gbk_chinese_ci'
   ;
Warnning: [E2007] 数据库已存在

-- 不会对原数据库产生影响
SQL> SELECT db_name,char_set from dba_databases WHERE db_name='
   DB_FEA';

DB_NAME | CHAR_SET |
-----------------------------------------------------------
DB_FEA| GB18030|
```

5.4 删除数据库

语法格式

```
DROP DATABASE [IF EXISTS] database_name
```

参数说明

- IF EXISTS：删除数据库时若不存在则忽略此错误。
- database_name：数据库名。

示例

将删除用户库 dbtest，同时该库下所有对象与数据都将被清除。该操作为不可逆操作，需数据库管理员在系统库执行。

```
DROP DATABASE dbtest;
```

删除时数据库不存在且支持了 IF EXISTS 关键字。

```
DROP DATABASE IF EXISTS dbtest;

Warnning: [E2006] 数据库DBTEST不存在
```

5.5 重命名数据库

语法格式

```
ALTER DATABASE database_name RENAME TO database_new_name
```

参数说明

- database_name：数据库名。
- database_new_name：重命名后的数据库名。

示例

该示例将数据库 dbtest 重命名为 dbrename，该操作需要数据库管理员在系统库执行，且要求无同名数据库。

```
ALTER DATABASE dbtest RENAME TO dbrename;
```

该示例将数据库 dbtest 重命名为 dbrename，该操作需要数据库管理员在系统库执行，且要求无同名数据库。

6 表对象管理

6.1 概述

数据库中包含一个或者多个表。表是数据的集合，是用来存储数据和操作数据的逻辑结构。表是由行和列组成的，行被称为记录，是组织数据的单位；列被称为字段，字段是比记录更小的单位，字段集合组成记录，每个字段标识一个记录的属性，即数据项，在特定的表中，列名必须唯一，但相同的列名可以出现在数据库不同的表中。

6.2 创建表

数据库启动完成后，可自定义基表保存用户数据。

6.2.1 主要语法结构

语法格式

```
createtabstmt::=
CREATE [opt_temp] TABLE [opt_if_not_exists] [schema.] tab_name (
   table_elements)
    [on_commit_del]
    [opt_partitioning_clause]
    [opt_subpartitioning_clause]
    [opt_store_props]
    [opt_comment]
```

参数说明

- CREATE：关键字，表示创建操作。
- opt_temp：可选参数，用于指定创建的是临时表。有效选项包括：
  - TEMPORARY
  - TEMP
  - LOCAL TEMPORARY
  - LOCAL TEMP
  - GLOBAL TEMPORARY
  - GLOBAL TEMP

- TABLE：关键字，表示创建的对象是表。
- opt_if_not_exists：可选参数，如果表已经存在，则不执行创建操作。使用 IF NOT EXISTS 关键字。
- schema.：可选参数，用于指定表所属的模式。如果省略，则使用默认模式。
- tab_name：表的名称。
- (table_elements)：括号内定义表的列和其他元素。
- on_commit_del：对于临时表，定义事务提交后如何处理临时表的数据。
- opt_partitioning_clause：指定表的分区信息，有助于提高大型表的查询性能。
- opt_subpartitioning_clause：子分区信息，进一步细化分区规则。
- opt_store_props：存储属性，比如存储位置、文件格式等。
- opt_comment：对表或列的注释，方便后续管理和维护。

6.2.2 表元素定义 table_elements

语法格式

```
table_elements::=
col_elements[,col_elements]...
```

参数说明

- col_elements：定义表的列和其他元素。

6.2.3 列元素定义 col_elements

语法格式

```
col_elements::=
col_name type [IDENTITY(B,S)] [NOT NULL | NULL] [DEFAULT
   default_value]
[UNIQUE | PRIMARY KEY | CHECK(expression)]
[COMMENT 'string'] [reference_definition]
|    CONSTRAINT name [CHECK(expression) | UNIQUE(columnList) |
   PRIMARY KEY(columnList) | FOREIGN KEY opt_fk_name(columnList)
   reference_definition]
```

参数说明

- col_name type [IDENTITY(B,S)] [NOT NULL | NULL] [DEFAULT default_value]：定义列的基本属性。

  - col_name：列名。
  - type：列的数据类型。
  - IDENTITY(B,S)：可选参数，表示列是否为自增列及其起始值和步长。
  - NOT NULL | NULL：可选参数，指定列是否允许空值。
  - DEFAULT default_value：可选参数，指定列的默认值。
- [UNIQUE | PRIMARY KEY | CHECK(expression)]：可选参数，用于定义列级别的唯一性、主键或检查约束。
  - UNIQUE: 列的值必须唯一。
  - PRIMARY KEY: 列为主键。
  - CHECK(expression): 列值必须满足某个条件。
- COMMENT 'string'：可选参数，为列添加注释。
- reference_definition：可选参数，用于定义外键约束。
- CONSTRAINT name [CHECK(expression) | UNIQUE(columnList) | PRIMARY KEY(columnList) | FOREIGN KEY opt_fk_name(columnList) reference_definition]：对表中列或列的组合设置的一种限制条件。
- name: 约束的名称，用于标识该约束。
- CHECK(expression): 检查约束，确保列的值满足某个条件。
- UNIQUE(columnList): 唯一约束，确保指定列的组合值是唯一的。
- PRIMARY KEY(columnList): 主键约束，确保指定列的组合值是唯一的，并且不允许为空。
- FOREIGN KEY opt_fk_name(columnList) reference_definition: 外键约束，确保指定列的值引用另一个表中的主键值。
  - opt_fk_name: 可选的外键名称。
  - columnList: 要定义为外键的列列表。

6.2.4 外键定义 reference_definition

语法格式

```
reference_definition::=
```

```
REFERENCES tbl_name [(index_col_name,...)]
[ON {UPDATE | DELETE} {CASCADE | SET NULL | NO ACTION | SET DEFAULT
   }]
[ON DELETE {CASCADE | SET NULL | NO ACTION | SET DEFAULT}]
```

参数说明

- REFERENCES tbl_name [(index_col_name,...)]：指定外键引用的目标表及列。
  - tbl_name: 引用的表的名称。
  - index_col_name,...: 引用表中的列列表。通常情况下，这些列是引用表的主键或唯一键。
- [ON UPDATE CASCADE | SET NULL | NO ACTION | SET DEFAULT]：可选参数，定义更新操作时的行为。
- [ON DELETE CASCADE | SET NULL | NO ACTION | SET DEFAULT]：可选参数，定义删除操作时的行为。
  - CASCADE: 表示当父表记录变更时子表受影响记录的相应字段值随之变更，当父表记录删除时，子表中受影响记录跟着删除。
  - SET NULL: 表示父表更改或删除，子表记录的相应字段值置为空值。
  - NO ACTION: 表示当父表记录变更时，导致子表中某些记录不再满足外键约束，系统将禁止父表作变更，其效果为更改父表的事务将被回滚。
  - SET DEFAULT: 表示父表更改或删除时，子表相应记录的相应字段值置为默认值，若该字段无默认值则置空。

6.2.5 事务提交行为 on_commit_del

语法格式

```
on_commit_del::=
ON COMMIT DELETE ROWS
|   ON COMMIT PRESERVE ROWS
```

参数说明

- ON COMMIT DELETE ROWS：当事务提交时，删除临时表中的所有行。
- ON COMMIT PRESERVE ROWS：当事务提交时，保留临时表中的行。

6.2.6 表一级分区定义 opt_partitioning_clause

语法格式

```
opt_partitioning_clause::=
PARTITION BY RANGE (name_list) [INTERVAL expr {DAY|MONTH|YEAR}]
   PARTITIONS (range_parti_item[,range_parti_item]...)
|    PARTITION BY LIST (name_list) PARTITIONS (list_parti_item[,
   list_parti_item]...)
|    PARTITION BY HASH (name_list) PARTITIONS ICONST
|    PARTITION BY HASH (name_list) PARTITIONS (name_list)
```

参数说明

- PARTITION BY RANGE (name_list) [INTERVAL expr DAY|MONTH|YEAR] PARTITIONS (range_parti_item[,range_parti_item]...)：按范围分区。
- PARTITION BY LIST (name_list) PARTITIONS (list_parti_item[,list_parti_item]...)：按列表分区。
- PARTITION BY HASH (name_list) PARTITIONS ICONST：按哈希值分区，ICONST 是分区的数量。
- PARTITION BY HASH (name_list) PARTITIONS (name_list)：按哈希值分区，name_list 是分区的名称列表。

6.2.7 表二级分区定义 opt_subpartitioning_clause

语法格式

```
opt_subpartitioning_clause::=
SUBPARTITION BY HASH (name_list) SUBPARTITIONS ICONST
|    SUBPARTITION BY HASH (name_list) SUBPARTITIONS (name_list)
|    SUBPARTITION BY LIST (name_list) SUBPARTITIONS (list_parti_item
   [,list_parti_item]...)
|    SUBPARTITION BY RANGE (name_list) SUBPARTITIONS (
   range_parti_item[,range_parti_item]...)
```

参数说明

- SUBPARTITION BY HASH (name_list) SUBPARTITIONS ICONST：定义哈希子分区。
- SUBPARTITION BY HASH (name_list) SUBPARTITIONS (name_list)：定义哈希子分区，使用名称列表。
- SUBPARTITION BY LIST (name_list) SUBPARTITIONS (list_parti_item[,list_parti_item]...)：定义列表子分区。

- SUBPARTITION BY RANGE (name_list) SUBPARTITIONS (range_parti_item[,range_parti_item]...)：定义范围子分区。

6.2.8 表存储定义信息 store_prop

语法格式

```
store_prop::=
PCTFREE expr
|    PCTUSED expr
|    COPY NUMBER expr
|    {COMPRESS|NOCOMPRESS}
```

参数说明

- PCTFREE expr：定义块中预留的空间百分比。
- PCTUSED expr：定义块中已使用的空间百分比。
- COPY NUMBER expr：定义复制的数量。
- COMPRESS|NOCOMPRESS：定义是否压缩数据。

6.2.9 注释 opt_comment

语法格式

```
opt_comment::=
COMMENT SCONST
```

参数说明

- COMMENT 'string'：为表或列添加注释。

6.2.10 示例

- 示例 1

```
SQL> CREATE TABLE IF NOT EXISTS HR.employees (
-- 列定义
employee_id INT IDENTITY (1,1) NOT NULL PRIMARY KEY, -- 自增主键
first_name VARCHAR(50) NOT NULL CONSTRAINT xg_name CHECK (LENGTH(
   first_name) >= 2),  -- 列级检查约束
email VARCHAR(100) DEFAULT 'unknown@example.com',  -- 默认值
hire_date DATE NOT NULL DEFAULT SYSDATE, -- 默认值
salary NUMERIC(10,2) CHECK (salary > 0),  -- 列级检查约束
department_id NUMERIC(4,0) CONSTRAINT xg_dept REFERENCES HR.
   departments(department_id),  -- 列级外键，HR.departments表已存
   在
-- 表级约束
CONSTRAINT xg_email UNIQUE (email),  -- 命名唯一约束
CONSTRAINT xg_salary CHECK (salary <= 100000)  -- 命名检查约束
```

```
)
-- 事务控制
ON COMMIT PRESERVE ROWS  -- 保留数据至会话结束
-- 范围分区
PARTITION BY RANGE (salary)
PARTITIONS (
part1 VALUES LESS THAN (5000),
part2 VALUES LESS THAN (10000),
part3 VALUES LESS THAN (MAXVALUES)
)
-- 子哈希分区
SUBPARTITION BY HASH (employee_id) SUBPARTITIONS 2
-- 存储参数
PCTFREE 10
-- 表注释
COMMENT '员工信息表';
```

该示例表示创建一个 HR 模式下的 employees 表。

- 示例 2

```
CREATE TABLE sysdba.test_tab(
kid INT IDENTITY(1,1) COMMENT '身份ID',
name CHAR(10) NOT NULL COMMENT '姓名',
age TINYINT DEFAULT '0' COMMENT '年龄',
birth DATETIME COMMENT '生日',
proc VARCHAR(10) COMMENT '所属省份',
comments VARCHAR COMMENT '备注')
PARTITION BY LIST(proc)
PARTITIONS(('北京'),('四川'),('重庆'),(OTHERVALUES))COPY NUMBER
    2;
```

该示例表示在 sysdba 模式下创建一张名为 test_tab 的表，该表包含 6 列，该表按照“所属省份”进行列表分区，分区键值包括：北京、四川、重庆以及其他值（若无 OTHERVALUES 分区，插入所属省份值非指定列表中的值则无法插入，OTHERVALUES 范围分区中对应 MAXVALUES），并且指定该表存储版本数为 2（copy number 在多节点下有效且不能大于 3；单机默认为 1，指定大于 1 无效）。

- 示例 3

```
CREATE TABLE sysdba.test_tab2(
kid INT IDENTITY(1,1) COMMENT '身份ID',
name CHAR(10) NOT NULL COMMENT '姓名',
age TINYINT DEFAULT '0' COMMENT '年龄',
birth DATETIME COMMENT '生日',
proc VARCHAR(10) COMMENT '所属省份',
comments VARCHAR COMMENT '备注')
PARTITION BY RANGE(birth) INTERVAL 1 MONTH
PARTITIONS(part1 VALUES LESS THAN('1900-01-01 00:00:00'))
COMMENT '人员信息表';
```

该示例表示在 sysdba 模式下创建一张人员信息表 test_tab2，该表以生日字段进行范围分区，且该分区为自动扩展分区，起始分区为 1900 年 1 月 1 日，后续系统将根据插入生日时间进行自动扩展。

- 示例 4

```
SQL> CREATE TABLE test_tab(id int);
SQL> SELECT dt.table_name,dt.table_type FROM dba_tables dt WHERE
   dt.table_name='TEST_TAB';
TABLE_NAME     |TABLE_TYPE
----------------------
TEST_TAB   |0

SQL> SELECT * FROM test_tab;
ID
---------
SQL> CREATE TABLE IF NOT EXISTS test_tab(id int,name varchar(20))
   ; /*创建一个与原不表不同列属性的表此处会返回警告*/
SQL> SELECT dt.table_name,dt.table_type FROM dba_tables dt WHERE
   dt.table_name='TEST_TAB';/*不会对原表产生影响*/
TABLE_NAME     |TABLE_TYPE
----------------------
TEST_TAB   |0

SQL> SELECT * FROM test_tab;
ID
----------
```

该示例表示创建一个与原表同名但属性不同的表，此操作不会对原表产生影响。

6.2.11 行数据压缩

虚谷数据库支持行级别的数据压缩，数据行压缩可以节省磁盘存储空间和降低磁盘 I/O。另外，可以在缓冲池中缓存更多的数据，这样就可以提高缓冲池命中率。但是，使用行数据压缩与解压缩也需要占用更多的 CPU 处理周期。数据行压缩节省出来的存储空间和额外对 CPU 的性能消耗在实际应用中需要做一个权衡，对于不同的业务环境和数据，采用数据行压缩方案不一定是最佳的方案，有时可能会适得其反。默认情况下，行数据压缩功能为关闭状态，可通过在创建表对象时，尾部加上 COMPRESS 关键字进行开启表对象的行数据压缩功能。用法详解如下示例：

示例

可通过查阅系统表 sys_tables 的 COMPRESS_LEVEL 字段查看对应表对象是否开启行数据压缩功能。

```
CREATE TABLE sysdba.test_tab3(c1 INTEGER, c2 VARCHAR) COMPRESS;
```

6.3 修改表

表创建完成后，用户可根据需求对表的定义进行修改，修改的范围包括：增加列、删除列、修改列名称、数据类型、数据长度、增加约束、删除约束等。

> ⚠ 注意
>
> 修改表是一种比较危险的做法，可能造成数据的破坏。因此对表做修改时，特别是存在大量数据的表，需经过数据库管理员审核，且建议对该表进行数据备份。

6.3.1 主要语法结构

语法格式

```
altertablestmt::=
ALTER TABLE sche_name.tbl_name alter_specification
```

参数说明

- ALTER TABLE：SQL 语句的关键字，用于修改现有的表结构。
- sche_name：表所属的模式（schema）的名称。
- tbl_name：要修改的表的名称。
- alter_specification：具体的修改操作，可以包含多种不同的操作类型。
  - 添加列
  - 删除列
  - 修改列
  - 添加、修改和删除列的组合操作
  - 约束操作
  - 更改表的所有者
  - 设置表的状态
  - 启用或禁用操作权限
  - 分区操作
  - 重建堆表

- 设置慢速修改模式

> ⚠ 注意
>
> - 修改表结构为 DDL 操作，将会对对象加排他锁，此时需保证无其他用户使用操作该对象。
> - 修改表结构的操作中，删除表列、增加带有默认值非空列将重构对象与数据，后台将创建一张临时表对原表进行结构与数据复制，数据量越大消耗时间越长，故该类操作需经过慎重考虑再进行。
> - 对表添加唯一值索引后，会自动在表上创建对应的唯一值约束，此时若需删除该索引，只能通过删除对应约束的方式进行删除，不能直接删除唯一值索引。
> - 针对分区表，删除与清理分区将导致全局索引失效，需重建索引，所以在分区表上不建议使用全局索引，尽量使用局部分区索引，避免分区操作带来的重建索引开销；若需重建索引可指定 REBUILD 参数。
> - 数据库系统中非空表不允许进行非兼容类型间的数据类型转换；定长字符 CHAR 进行精度扩展将重构数据，建议创表或更改类型时使用 VARCHAR 数据类型；NUMERIC 字段类型进行精度扩展时，由于不重构数据，故针对历史记录的修改超过原类型精度则数据无法保证正确性，建议存在该操作的表记录变更采用删除再插入的方式进行。
> - CASCADE 关键字用于存在依赖关系的表对象变更，此时将忽略对象依赖关系，但是可能导致依赖对象失效或被系统删除，所以对存在依赖关系的对象进行变更操作需确认各对象依赖关系，操作完成后应检查依赖对象的有效性。

6.3.2 添加列 add_columns

语法格式

```
add_columns [CASCADE | RESTRICT] [NOWAIT | WAIT | WAIT ICONST]

add_columns::=
ADD [COLUMN] table_element[,table_element]
|   ADD [COLUMN] ( table_element[,table_element] )

table_element::=
col_name type [IDENTITY(B,S)] [col_qual_list] [COLLATE colid] [
   COMMENT SCONST]
|   [CONSTRAINT name] constraint_elem

col_qual_list::=
col_qual_list [CONSTRAINT name] col_constraint_elem

col_constraint_elem::=
NOT NULL
|   NULL
|   UNIQUE
|   PRIMARY KEY
|   CHECK ( bool_expr )
|   DEFAULT expr
|   REFERENCES name_space [(index_col_name,...)] [ON UPDATE {
   CASCADE | SET NULL | NO ACTION | SET DEFAULT}] [ON DELETE {
   CASCADE | SET NULL | NO ACTION | SET DEFAULT}]
|   FOREIGN KEY opt_fk_name REFERENCES name_space [(index_col_name
```

```
   ,...)] [ON UPDATE {CASCADE | SET NULL | NO ACTION | SET DEFAULT
   }] [ON DELETE {CASCADE | SET NULL | NO ACTION | SET DEFAULT}]

constraint_elem::=
CHECK ( bool_expr )
|    UNIQUE ( columnList )
|    PRIMARY KEY ( columnList )
|    FOREIGN KEY opt_fk_name ( columnList ) REFERENCES name_space [(
   index_col_name,...)] [ON UPDATE {CASCADE | SET NULL | NO ACTION
    | SET DEFAULT}] [ON DELETE {CASCADE | SET NULL | NO ACTION | SET
    DEFAULT}]
```

参数说明

- [CASCADE | RESTRICT]：控制是否级联修改依赖对象。
  - CASCADE：自动应用更改到所有依赖对象。
  - RESTRICT：如果有依赖对象，则阻止更改。
- [NOWAIT | WAIT | WAIT ICONST]：控制等待锁的行为。
  - NOWAIT：如果无法立即获取锁，则立即返回错误。
  - WAIT：等待直到能够获取锁。
  - WAIT ICONST：等待直到满足特定条件。
- ADD [COLUMN]：添加一个或多个新列。
- col_name: 列的名称。
- type: 列的数据类型。
- [IDENTITY(B,S)]：是否为标识列及其生成方式。
- [col_qual_list]：列的约束条件列表。
- [COLLATE colid]：列的排序规则。
- [COMMENT SCONST]：列的注释。
- [CONSTRAINT name]：约束的名称。

6.3.3 删除列 drop_columns

语法格式

```
drop_columns [CASCADE | RESTRICT] [NOWAIT | WAIT | WAIT ICONST]

drop_columns::=
DROP COLUMN name_list
```

参数说明

- DROP COLUMN：删除一个或多个列。
- name_list：要删除的列名列表。

6.3.4 修改列 alter_columns

语法格式

```
alter_columns [CASCADE | RESTRICT] [FORCE] [NOWAIT | WAIT | WAIT
   ICONST]

alter_columns::=
ALTER [COLUMN] alt_columns
|    ALTER [COLUMN] ( alt_columns )
|    MODIFY [COLUMN] alt_columns
|    MODIFY [COLUMN] ( alt_columns )

alt_columns::=
alter_column_item
|    col_name type [IDENTITY(B,S)] [col_qual_list] [COLLATE colid] [
   COMMENT SCONST]
|    alt_columns ',' alter_column_item
|    alt_columns ',' col_name type [IDENTITY(B,S)] [col_qual_list] [
   COLLATE colid] [COMMENT SCONST]

alter_column_item::=
colid SET DEFAULT [ON NULL] b_expr
|    colid DROP DEFAULT
|    colid SET NOT NULL
|    colid SET NOTNULL
|    colid SET NULL
|    colid DROP NOT NULL
|    colid DROP NOTNULL
```

参数说明

- ALTER [COLUMN]：修改一个或多个列。
- MODIFY [COLUMN]：修改一个或多个列。

6.3.5 添加、修改和删除列的组合操作

语法格式

```
add_columns drop_columns [CASCADE | RESTRICT] [NOWAIT | WAIT | WAIT
    ICONST]
| add_columns alter_columns [CASCADE | RESTRICT] [FORCE] [NOWAIT |
   WAIT | WAIT ICONST]
| add_columns alter_columns drop_columns [CASCADE | RESTRICT] [
   FORCE] [NOWAIT | WAIT | WAIT ICONST]
```

```
| alter_columns drop_columns [CASCADE | RESTRICT] [FORCE] [NOWAIT
    | WAIT | WAIT ICONST]
```

6.3.6 约束操作

语法格式

```
MODIFY CONSTRAINT colid {ENABLE | DISABLE}
| ENABLE CONSTRAINT name_list [NOWAIT | WAIT | WAIT ICONST]
| DISABLE CONSTRAINT name_list [NOWAIT | WAIT | WAIT ICONST]
| DROP CONSTRAINT name_list [KEEP INDEX | DROP INDEX] [CASCADE |
   RESTRICT] [NOWAIT | WAIT | WAIT ICONST]
```

参数说明

- MODIFY CONSTRAINT：修改约束的状态。
- ENABLE CONSTRAINT：启用一个或多个约束。
- DISABLE CONSTRAINT：禁用一个或多个约束。
- DROP CONSTRAINT：删除一个或多个约束。
- [KEEP INDEX | DROP INDEX]：控制是否保留或删除相关的索引。

6.3.7 更改表的所有者

语法格式

```
OWNER TO userid [NOWAIT | WAIT | WAIT ICONST]
```

参数说明

- OWNER TO userid：更改表的所有者。

6.3.8 设置表的状态

语法格式

```
SET ONLINE [NOWAIT | WAIT | WAIT ICONST]
| SET OFFLINE [NOWAIT | WAIT | WAIT ICONST]
```

参数说明

- SET ONLINE：将表设置为在线状态。
- SET OFFLINE：将表设置为离线状态。

6.3.9 启用或禁用操作权限

语法格式

```
ENABLE operation_commalist [NOWAIT | WAIT | WAIT ICONST]
| DISABLE operation_commalist [NOWAIT | WAIT | WAIT ICONST]

operation_commalist::=
operation[,operation]

operation::=
SELECT
|    INSERT
|    UPDATE
|    DELETE
|    EXECUTE
|    REFERENCES
|    ALTER
|    DROP
|    INDEX
|    TRIGGER
```

参数说明

- ENABLE operation_commalist：启用一个或多个操作权限。
- DISABLE operation_commalist：禁用一个或多个操作权限。
- operation_commalist：操作权限列表，如 SELECT，INSERT，UPDATE。

6.3.10 分区操作

语法格式

```
SET PARTITION colid ONLINE [NOWAIT | WAIT | WAIT ICONST]
| SET PARTITION colid OFFLINE [NOWAIT | WAIT | WAIT ICONST]
| DROP PARTITION colid [REBUILD GLOBAL INDEX] [NOWAIT | WAIT | WAIT
    ICONST]
| TRUNCATE PARTITION colid [REBUILD GLOBAL INDEX] [NOWAIT | WAIT |
   WAIT ICONST]
| ADD PARTITION colid VALUES (parti_values) [NOWAIT | WAIT | WAIT
   ICONST]
| ADD PARTITION colid VALUES LESS THAN (parti_values) [NOWAIT |
   WAIT | WAIT ICONST]
```

参数说明

- SET PARTITION ONLINE：将分区设置为在线状态。
- SET PARTITION OFFLINE：将分区设置为离线状态。
- DROP PARTITION：删除分区。
  - [REBUILD GLOBAL INDEX]：控制是否重建全局索引。

- TRUNCATE PARTITION: 截断分区。
  - [REBUILD GLOBAL INDEX]：控制是否重建全局索引。
- ADD PARTITION：添加分区。
  - VALUES (parti_values)：指定分区的值。
  - VALUES LESS THAN (parti_values)：指定分区的值范围。

6.3.11 重建堆表

语法格式

```
REBUILD HEAP [NOWAIT | WAIT | WAIT ICONST]
```

参数说明

- REBUILD HEAP：重建表的堆结构，用于优化存储或解决碎片问题。

> 说明
>
> 自 V12.7 版本开始，重建堆表时也会将表中的 BLOB、CLOB 类型等大对象存储进行重建。

6.3.12 设置慢速修改模式

语法格式

```
SET SLOW MODIFY ON [NOWAIT | WAIT | WAIT ICONST]
| SET SLOW MODIFY OFF [NOWAIT | WAIT | WAIT ICONST]
```

参数说明

- SET SLOW MODIFY ON：设置表的慢速修改模式。
- SET SLOW MODIFY OFF：取消表的慢速修改模式。

6.3.13 示例

- 示例 1

```
ALTER TABLE test_tab ADD COLUMN new_col VARCHAR(20) NOT NULL
   DEFAULT '000';
```

该示例表示对表 test_tab 增加一列带默认值 000 的非空列 new_col，此时将对该表进行重构，若原表中记录量越大，变更操作时间将越长。

- 示例 2

```
ALTER TABLE test_tab ADD CONSTRAINT test_tab_uk UNIQUE(kid,name);
```

该示例表示对表 test_tab 增加一个名为 test_tab_uk 的唯一值约束，约束键由 kid 与 name 组成。

- 示例 3

```
ALTER TABLE test_tab2 ADD PARTITION part_n VALUES LESS THAN(
'2022-05-20 10:10:10');
```

该示例表示对列表分区表 test_tab2 增加一个分区名为 part_n 的分区，分区键值为“2022-05-20 10:10:10”。

- 示例 4

```
ALTER TABLE test_tab ALTER COLUMN name VARCHAR(20);
```

该示例表示将 test_tab 表的 name 字段的数据类型变更为 VARCHAR(20)，若不修改数据类型仅将精度更改为 CHAR(20)，将重构数据，数据量越大时间开销越大。

- 示例 5

```
ALTER TABLE test_tab DROP COLUMN new_col CASCADE;
```

该示例表示强制删除 test_tab 表的 new_col 字段，此时将重构数据，数据量越大时间开销越大。

- 示例 6

```
ALTER TABLE test_tab DROP CONSTRAINT test_tab_uk;
```

该示例表示删除 test_tab 的 test_tab_uk 约束。

- 示例 7

```
ALTER TABLE test_tab DROP PARTITION part1 REBUILD GLOBAL INDEX;
```

该示例表示删除表 test_tab 的分区，分区名为 part1，并重建该表全局索引。

⚠ 注意

- 删除分区时不允许删除异常分区 (MAXVALUES/OTHERVALUES)。
- 哈希分区表的分区不允许删除。
- 删除分区将导致表的全局索引失效，此时需重建索引或在删除时指定 REBUILD 参数。
- 不允许删除分区表的所有分区，必须至少保留一个分区。

- 示例 8

```
ALTER TABLE test_tab DISABLE INSERT;
```

该示例表示将表 test_tab 置为不可插入状态，即该表不接受插入操作。

- 示例 9

```
ALTER TABLE test_tab ENABLE INSERT;
```

该示例表示将表 test_tab 置为允许插入状态，其含义同 DISABLE INSERT 相反。

- 示例 10

```
ALTER TABLE sysdba.test_tab RENAME TO test_tab_new;
```

该示例表示将 sysdba 模式下的表 test_tab 重命名为 test_tab_new。

- 示例 11

```
ALTER TABLE test_tab_new RENAME kid TO keyid;
```

该示例表示将 test_tab_new 的列名 kid 改为 keyid。

- 示例 12

```
ALTER TABLE test_tab_new OWNER TO guest;
```

该示例表示将 test_tab_new 的属主改为 guest，此时用户 guest 拥有该表的所有权，可进行所有操作，其他用户访问该表需带上 guest 模式信息。

- 示例 13

```
ALTER TABLE guest.test_tab_new ALTER COLUMN PROC SET DEFAULT '北
    京';
```

该示例表示对 guest 模式下的 test_tab_new 表中的 proc 字段设置默认值，指定默认值为北京。

- 示例 14

```
ALTER TABLE guest.test_tab_new TRUNCATE partition part2;
```

该示例表示清除 guest 模式下的 test_tab_new 表的 part2 分区数据，此时该表上的全局索引将被置为失效，若该索引为唯一值，则会导致主键/唯一值约束失效，无法进行数据插入，需重建索引。

- 示例 15

```
ALTER TABLE test_tab REBUILD HEAP;
```

该示例表示重整表，重整表功能主要用于回收无效存储空间。当表数据通过 DELETE 操作删除非连续存储数据后，会产生无法再利用的无效存储，此部分存储只有通过重整表功能回收后，才能再次使用。

> **⚠ 注意**
>
> 重整表将重新转移表的所有数据到新存储，对于数据量较大的表，此操作耗时较长，请谨慎操作。

数据库运维人员可以通过下列语句查询无效存储比例，评估是否需要执行重整表，当表存储碎片率超过 0.5（50%）时，建议执行重整表操作，以提升数据查询效率与存储利用率。

```
-- 创建查询表中存储碎片率的存储函数
CREATE OR REPLACE FUNCTION GET_TABLE_FRAGMENTATION_RATE(
arg_db_name VARCHAR,
arg_schema_name VARCHAR,
arg_table_name VARCHAR
) RETURN DOUBLE AS
TYPE ret_type IS RECORD(del_rows BIGINT, all_rows BIGINT);
ret ret_type;
BEGIN
SELECT
sum(del_num) del_rows,
sum(row_num)+del_rows all_rows
INTO ret
FROM sys_databases db
INNER JOIN sys_schemas scm ON db.db_id=scm.db_id
INNER JOIN sys_tables tb ON scm.schema_id=scm.schema_id
INNER JOIN sys_gstores gsto ON tb.gsto_no=gsto.head_no
INNER JOIN sys_stores sto ON gsto.gsto_no=sto.gsto_no
WHERE db_name=arg_db_name
AND schema_name=arg_schema_name
AND table_name=arg_table_name;
RETURN ret.del_rows/ret.all_rows;
END;

-- 创建表并插入 1000 条数据
CREATE TABLE tb_rebuild_heap(id INT IDENTITY(1,1),c1 CHAR(8192));
BEGIN
FOR i IN 1..1000 LOOP
INSERT INTO tb_rebuild_heap VALUES(DEFAULT,'test');
END LOOP;
END;

-- 删除表中 id 为偶数的 500 条数据
DELETE FROM tb_rebuild_heap WHERE MOD(id,2)=0;

-- 查询表中存储碎片率，此时为 0.5（50%）
SELECT GET_TABLE_FRAGMENTATION_RATE('SYSTEM','SYSDBA','
   tb_rebuild_heap');

-- 执行重整表
ALTER TABLE tb_rebuild_heap REBUILD HEAP;

-- 查询表中存储碎片率，此时为 0（0%）
SELECT GET_TABLE_FRAGMENTATION_RATE('SYSTEM','SYSDBA','
   tb_rebuild_heap');
```

6.3.14 其他功能

修改列定义时，字段类型一致的变更不报错（兼容 Oracle）

- 原有状态

```
CREATE TABLE test_tab(f1 VARCHAR(10));

ALTER TABLE test_tab MODIFY f1 VARCHAR(10);
Error: [E16016] 字段F1的新定义与旧定义完全一致
```

当配置兼容模式为 Oracle 时（compatible_mode=oracle），错误 16016 级别设置为警告，不以错误方式返回客户端。

- 当前状态

```
SET COMPATIBLE_MODE TO ORACLE;
CREATE TABLE test_tab(f1 VARCHAR(10));

ALTER TABLE test_tab MODIFY f1 VARCHAR(10);
Warnning: [E16016] 字段F1的新定义与旧定义完全一致
```

变更表对象定义的操作语法（兼容 Oracle）

- 修改列新增方式，使用 modify 关键字，兼容原来的的 alter column。
- 当前支持默认值的 ON NULL 功能。

示例：

```
CREATE TABLE test_tab(a int);
--使用modify修改字段
ALTER TABLE test_tab MODIFY a SMALLINT DEFAULT 1;
INSERT INTO test_tab VALUES(DEFAULT);
--使用alter column修改字段
ALTER TABLE test_tab ALTER COLUMN a INT DEFAULT ON NULL 10;
INSERT INTO test_tab VALUES(DEFAULT);

SELECT * FROM test_tab;
A |
------------------------------------------
1 |
10 |
```

6.4 删除表

语法格式

```
droptabstmt::=
DROP TABLE [IF EXISTS] sche_name.tab_name [CASCADE|RESTRICT];
```

参数说明

- DROP TABLE：关键字，用于删除表。
- schema_name：表所属的模式（schema）。
- table_name：表的名称。

RESTRICT ：可选关键字，用于指定删除表时的行为。

  - CASCADE：强制删除该表，包括依赖于该表的对象，如存储过程、触发器等。
  - RESTRICT：默认值，如果存在依赖对象，则拒绝删除表。

- IF EXISTS：可选关键字，用于检查表是否存在。如果表不存在，不会抛出错误，而是忽略该操作。如果不使用 IF EXISTS，而表不存在，则会抛出错误。

示例

- 示例 1

```
DROP TABLE test_tb;
```

- 示例 2

```
DROP TABLE IF EXISTS test_tb;
```

> **⚠ 注意**
>
> 删除基表过程中，若存在依赖于该表的对象（如存储过程、触发器等），用户如果强制删除该表（指定 CASCADE 参数），此时基于该表的所有对象都将被删除，默认为 RESTRICT。

6.5 约束管理

6.5.1 主键约束

主键为表记录的唯一标识约束，在整个表中，不允许主键字段的记录出现重复，主键字段默认非空；一个表中仅能创建一个主键。

语法格式

```
primary_constriant_stmt::=
[CONSTRAINT name] PRIMARY KEY(column_1[,...n])
```

参数说明

- [CONSTRAINT name]：可选关键字，用于指定主键约束的名称。
- name：约束的名称，用于唯一标识该约束。

- PRIMARY KEY：关键字，用于定义主键约束。
- (column_1[,...n])：括号内的列名列表，用于指定哪些列组合成主键。

示例

创建一个表 test，并在 id 和 tel 列上定义一个复合主键。复合主键确保 id 和 tel 列的组合值是唯一的。

```
-- 创建表
CREATE TABLE test(id INTEGER ,name CHAR(10), tel INTEGER,
   CONSTRAINT test_pk PRIMARY KEY (id, tel));
-- 插入值
INSERT INTO test (id, name, tel) VALUES (1, 'He', 1234567890);
-- 查询表信息
SELECT * FROM test;

ID | NAME | TEL |
-------------------------------------------------------------
1 | He| 1234567890 |

-- 插入一条违反主键约束的记录，返回错误
INSERT INTO test (id, name, tel) VALUES (1, 'Zhang', 1234567890);
Error: [E13001] 违反唯一值约束
```

若用户不指定约束名可采用以下语法进行表创建，系统默认为缺省的约束创建一个约束名称。

```
-- 创建不带约束名的表
CREATE TABLE test2(id INTEGER, name CHAR(10), tel INTEGER, PRIMARY
   KEY(id, tel));

-- 查询约束定义为id,tel的约束名称
SELECT CONS_NAME FROM sys_constraints WHERE DEFINE = '"id","tel"';

CONS_NAME |
-------------------------------------------------------------
K_S24183173287646248|
```

> **注意**
>
> 查询约束信息详细说明请参见《系统字典参考》> 模式对象 > 约束-sys_constraints 章节。

6.5.2 外键约束

外键约束，用于规定表中指定字段取值只能是依赖表的记录中的主键或唯一字段的值，不能为其它值。

语法格式

```
foreign_constriant_stmt::=
[CONSTRAINT name] FOREIGN KEY opt_fk_name REFERENCES name_space [(
   index_col_name,...)] [key_actions]
```

```
key_actions::=
ON UPDATE {CASCADE | SET NULL | NO ACTION | SET DEFAULT}
|    ON DELETE {CASCADE | SET NULL | NO ACTION | SET DEFAULT}
|    /*empty*/
```

参数说明

- [CONSTRAINT name]：可选关键字，用于指定外键约束的名称。
- FOREIGN KEY：关键字，用于定义外键约束。
- opt_fk_name：指定外键的名称。
- REFERENCES name_space [(index_col_name,...)]：指定引用的目标表和列。
- key_actions：外键约束的反向规则描述子句，当被引用的表（父表）发生记录更改或删除时，引用表（子表）中引用了父表中发生更改的记录的字段值的记录将不再符合约束条件，此时，系统将根据外键约束规则对子表记录重新设置值、置空或不进行操作而报错。
- NO ACTION：当父表记录变更时，导致子表中某些记录不再满足外键约束，系统将禁止父表作变更，其效果为更改父表的事务将被回滚。
- CASCADE：当父表记录变更时子表受影响记录的相应字段值随之变更，当父表记录删除时，子表中受影响记录跟着删除。
- SET NULL：父表更改或删除，子表记录的相应字段值置为空值。
- SET DEFAULT：父表更改或删除时，子表相应记录的相应字段值置为默认值，若该字段无默认值则置空。

示例

表 teacher 用于记载学校中所有教师的基本情况，另一张表 course 用于记载学校所有的课程安排，course 中一个字段用于表示某门课程的授课教师。若该库在逻辑上是完整一致的，则 course 中用于表示授课教师的字段的取值都应在表 teacher 中出现，否则，说明表 course 中某记录的表示授课教师的字段的取值错误，或者表 teacher 并未完全包含所有教师的信息。为防止此类错误的出现，可以在创建表 course 时设定外键约束，令表 course 中的授课教师字段引用表 teacher 中的教师姓名字段。

```
-- 创建teacher表，teacher_name作为主键
CREATE TABLE teacher (
teacher_name VARCHAR(255) PRIMARY KEY,
teacher_age INT
);
```

```
-- 插入数据
INSERT INTO teacher (teacher_name, teacher_age) VALUES ('He', 35);
INSERT INTO teacher (teacher_name, teacher_age) VALUES ('Wang', 40)
   ;
INSERT INTO teacher (teacher_name, teacher_age) VALUES ('Zhang'
   , 38);

-- 创建course表，添加外键约束引用teacher_name字段
CREATE TABLE course (
course_id INT PRIMARY KEY,
course_name VARCHAR(255) NOT NULL,
teacher_name VARCHAR(255),
CONSTRAINT fk_course_teacher FOREIGN KEY (teacher_name) REFERENCES
   teacher (teacher_name)
);

--插入数据
INSERT INTO course (course_id, course_name, teacher_name) VALUES
    (101, 'Math', 'He');
INSERT INTO course (course_id, course_name, teacher_name) VALUES
    (102, 'English', 'Wang');
INSERT INTO course (course_id, course_name, teacher_name) VALUES
    (103, 'Chinese', 'Zhang');

-- 插入一条无效的记录，即该teacher_name在teacher表中不存在，返回错
   误
INSERT INTO course (course_id, course_name, teacher_name) VALUES
    (104, 'Art', 'Zhao');
Error: [E13005] 违反外键约束
```

6.5.3 唯一值约束

唯一值约束（UNIQUE）与主键约束相似，但它只用于规定某个或者多个字段的取值唯一，一个表可以有多个唯一值约束，但却只能有一个主键约束。

语法格式

```
unique_constriant_stmt::=
[CONSTRAINT name] UNIQUE (columnList)
```

语法说明

- [CONSTRAINT name]：可选关键字，用于指定唯一性约束的名称。
- name：约束的名称，用于唯一标识该约束。
- UNIQUE：关键字，用于定义唯一性约束。
- (columnList)：括号内的列名列表，用于指定哪些列组合成唯一性约束。

示例

创建表 test_uniq 时，创建名为 test_uk 的唯一值约束，约束键为 id 与 name。

```
-- 创建表
CREATE TABLE test_uniq(id INTEGER,name CHAR(10),CONSTRAINT test_uk
   UNIQUE (id,name));

-- 插入数据
INSERT INTO test_uniq (id, name) VALUES (1, 'He');
INSERT INTO test_uniq (id, name) VALUES (2, 'Zhang');
INSERT INTO test_uniq (id, name) VALUES (1, 'Wang');

-- 查询表
SELECT * FROM test_uniq;

ID | NAME |
-------------------------------------------------------------
1 | He|
2 | Zhang|
1 | Wang|

-- 插入一条违反唯一性约束的记录，返回错误
INSERT INTO test_uniq (id, name) VALUES (1, 'He');
Error: [E13001] 违反唯一值约束
```

> **说明**
>
> 唯一值约束与主键约束相比少了字段非空约束，此时若约束键值某个字段或多个字段值为 null，则唯一值失效，因为 null 无法与 null 做比较。所以唯一值约束键需根据实际情况指定为非空。

6.5.4 默认值约束

默认值约束用于给表中指定列赋予一个常量值（默认值）。当向表插入数据时，如果用户没有明确给出该列的值，默认值约束会自动添加默认值。每一列只能有一个默认值。

语法格式

```
default_constriant_stmt::=
[CONSTRAINT name] DEFAULT expr
|    [CONSTRAINT name] DEFAULT (expr)
```

参数说明

- [CONSTRAINT name]：可选关键字，用于指定默认值约束的名称。
- name：约束的名称，用于唯一标识该约束。
- DEFAULT：关键字，用于定义默认值。
- expr: 表达式，可以是一个常量值、函数或其他表达式。

示例

创建表 test_default 并指定 name 字段默认值为 good!，当插入语句中未指定 name 值时，系统将默认对其赋值为 good!。

```
-- 创建表test_default并指定name字段默认值为good!
CREATE TABLE test_default(
id INTEGER IDENTITY(1,1),
name CHAR(10) CONSTRAINT def_cons DEFAULT 'good!'
);
-- 插入语句中未指定name值
INSERT INTO test_default DEFAULT VALUES;
-- 查询表
SELECT * FROM test_default;

ID | NAME |
-------------------------------------------------------------
1 | good!|
```

6.5.5 值检查约束

值检查约束用于限定一个字段的取值范围，也可以限定多个字段值相互间的关系，其形式为：CHECK（值约束表达式）。

语法格式

```
check_constriant_stmt::=
[CONSTRAINT name] CHECK (expression)
```

参数说明

- [CONSTRAINT name]：可选关键字，用于指定默认值约束的名称。
- name：约束的名称，用于唯一标识该约束。
- DEFAULT：关键字，用于定义默认值。
- expression：布尔表达式，指定列值必须满足的条件。

示例

创建一张名为 test_check 的表，该表插入数据值要求 id>10 或者 name 值为 chk，否则数据无法插入。

```
-- 创建表
CREATE TABLE test_check(id INTEGER, name CHAR(10), CHECK(id>10 OR
   name ='chk'));

-- 插入数据
INSERT INTO test_check (id, name) VALUES (11, '张三');
INSERT INTO test_check (id, name) VALUES (5, 'chk');

-- 查询表
SELECT * FROM test_check;
```

```
ID | NAME |
-------------------------------------------------------------
11 | abc|
5 | chk|

-- 插入无效数据，返回错误
INSERT INTO test_check (id, name) VALUES (5, 'cba');
Error: [E13008] 违反值检查约束
--
```

6.6 表分区管理

6.6.1 概述

虚谷数据库中，表分为堆表和分区表，分区表是指在创建表时，按照一定的划分条件，将表切分成若干个子表，在逻辑上，分区表与堆表具有相同的特性，插入、更改、删除、选择等操作的命令与堆表的操作命令完全一致。针对数据量过大（如单表超过 1 亿）情况，则建议采用分区表进行表结构设计。

虚谷数据库提供了两级三种分区方式，一级分区支持：列表分区、范围分区和哈希分区，二级分区支持：列表分区、范围分区和哈希分区。若一级分区为哈希分区，则无法创建二级分区。

6.6.2 一级分区

6.6.2.1 列表分区

列表分区是指在分区时将分区键的各种取值罗列出来，指定每个取值或几个取值对应一个分区，各个分区中的记录有一个共性，那就是其分区键字段的取值都是分区的条件值或是条件值之一，列表分区主要针对分区键的取值为有限的离散值的情况。

> **说明**
>
> 正常情况下，各个记录应归属于所有列表分区中的某一个分区，但由于列表分区的条件值是离散的，若某记录的取值不符合任何一个分区的分区条件，则它将不归属于任何一个分区，该记录只能丢失才能保证数据的正确性，为避免此情况的发生，引入异常分区，当记录不满足任何一个分区的条件时，将其归入 OTHER-VALUES 异常分区。

语法格式

```
list_partitioning_clause::=
PARTITION BY LIST (name_list) PARTITIONS (list_parti_item[,
   list_parti_item]...)[opt_subpartitioning_clause]
```

```
list_parti_item::=
(parti_values)
|    colid VALUES   (parti_values)
|    (OTHERVALUES)
|    colid VALUES   (OTHERVALUES)
```

参数说明

- PARTITION BY LIST：关键字，用于指定使用列表分区。
- name_list：一个或多个列的列表，用于分区的列。
- PARTITIONS：关键字，用于指定分区列表。
- list_parti_item：每个分区的定义。
- (parti_values)：一个包含具体值的列表，用于定义分区。
- colid VALUES (parti_values)：指定列 colid 的值列表。
- (OTHERVALUES)：一个特殊的分区，用于捕获所有其他未指定的值。
- colid VALUES (OTHERVALUES)：指定列 colid 的其他值。
- opt_subpartitioning_clause：描述所有分区进行子分区时使用的子分区模板，即各个分区无需分别描述其子分区，它们使用的子分区皆遵守子分区模板。当分区描述子句中含有子分区描述时，子分区模板将对该分区无效。

示例

创建一个带列表分区的分区表，分区键为 city，根据分区键值划分 4 个分区，当分区键值为“四川”、“云南”、“贵州”时数据分别存储在划定分区中，若为其他值则存储在 OTHERVALUES 分区。

```
CREATE TABLE test_part1(id INT,
name VARCHAR(20),
city CHAR(20))
PARTITION BY LIST(city)
PARTITIONS
(('四川'),('云南'),('贵州'),(OTHERVALUES));
```

> **说明**
>
> 若分区表设定了异常分区，则无法再增加分区，且无法删除异常分区。

6.6.2.2 范围分区

若分区键字段的取值是连续的，则不适合使用列表分区，此种情况的最佳分区模式是范围分区，即按条件将键的取值划分成若干个范围，每个范围对应一个分区，在向表插入记录前，先测试记录满足哪个分区的条件，再将记录插入至相应的分区中。

> **说明**
>
> 范围分区为连续分区，分区段之间为“前闭后开”的划分模式，现系统暂不支持分区融合与分裂操作；若分区表设定了异常分区，则无法再增加分区。

语法格式

```
range_partitioning_clause::=
PARTITION BY RANGE (name_list) [INTERVAL expr {DAY|MONTH|YEAR}]
   PARTITIONS (range_parti_item[,range_parti_item]...) [
   opt_subpartitioning_clause]

range_parti_item::=
(parti_values)
|    (MAXVALUES)
|    ColId VALUES LESS THAN (parti_values)
|    ColId VALUES LESS THAN (MAXVALUES)
```

参数说明

- PARTITION BY RANGE：关键字，用于指定使用范围分区。
- name_list：一个或多个列的列表，用于分区的列。
- [INTERVAL expr DAY|MONTH|YEAR]：用于指定分区的时间间隔。
- PARTITIONS (range_parti_item[,range_parti_item]...)：分区定义
- (parti_values)：一个包含具体值的列表，用于定义分区。
- (MAXVALUES)：一个特殊的分区，用于捕获所有大于最后一个分区的最大值的值。
- ColId VALUES LESS THAN (parti_values)：指定列 ColId 的值小于 parti_values 的分区。
- ColId VALUES LESS THAN (MAXVALUES)：指定列 ColId 的值小于最大值的分区。
- [opt_subpartitioning_clause]：可选子分区定义，用于进一步对分区进行子分区。

示例

创建一个以 ID 作为分区键的范围分区表，按照规则系统将 ID 小于 1 的数据划分在 part1，将大于等于 1 小于 1000 的数据划分在 part2，依次类推，最后大于等于 2000 的数据将划分在

MAXVALUES 异常分区内。

```
CREATE TABLE test_part2(
id INTEGER IDENTITY(1,2),
name VARCHAR,
city VARCHAR
)
PARTITION BY RANGE (id)
PARTITIONS (
part1 VALUES LESS THAN (1),
part2 VALUES LESS THAN (1000),
part3 VALUES LESS THAN (1500),
part4 VALUES LESS THAN (2000),
part5 VALUES LESS THAN (MAXVALUE)
);
```

6.6.2.3 哈希分区

HASH 分区主要用来确保数据在预先确定数目的分区中平均分布。在 RANGE 和 LIST 分区中，必须明确指定一个给定的列值或列值集合应该保存在哪个分区中；而在 HASH 分区中，数据库系统自动完成这些工作，用户要做的是指定被哈希的列值，以及分区数量。

> 📖 **说明**
>
> 哈希分区表无异常分区，且不可增加分区或删除分区。

语法格式

```
hash_partitioning_clause::=
PARTITION BY HASH (col_list) PARTITIONS ICONST
|   PARTITION BY HASH (col_list) PARTITIONS (name_list)
```

参数说明

- PARTITION BY HASH：关键字，用于指定使用哈希分区。
- col_list：一个或多个列的列表，用于分区的列。
- PARTITIONS：关键字，用于指定分区的数量。
- ICONST：一个确定整数，系统将根据其值进行表分区划分。
- name_list：分区名称的列表。

示例

创建 5 个哈希分区的分区表，分区名由系统默认。

```
CREATE TABLE test_part3
```

```
(t_no INTEGER,t_name VARCHAR,t_address VARCHAR)
PARTITION BY HASH(t_no)
PARTITIONS 5;
```

6.6.3 二级分区

6.6.3.1 列表分区

二级分区是对一级分区的再次分割。

> **说明**
>
> 虚谷数据库系统中定义二级分区即默认所有一级分区按照指定二级分区进行再分割，不能针对一级分区进行不同的二级分区划分。

语法格式

```
list_subpartitioning_clause::=
SUBPARTITION BY LIST (name_list) SUBPARTITIONS (list_parti_item[,
   list_parti_item]...)

sub_part_list::=
(parti_values)
|    subpart_name VALUES  (parti_values)
|    (OTHERVALUES)
|    subpart_name VALUES  (OTHERVALUES)
```

参数说明

- SUBPARTITION BY LIST：关键字，用于指定使用列表二级分区。
- name_list：一个或多个列的列表，用于二级分区的列。
- SUBPARTITIONS：关键字，用于指定二级分区列表。
- sub_part_list：每个二级分区的定义。格式与一级子分区的列表分区一致。

示例

创建一个以 city 为一级列表分区，addr 为二级列表子分区的分区表。

```
CREATE TABLE test_sub2(id INT,name VARCHAR,city VARCHAR,addr
   VARCHAR)
PARTITION BY LIST(city) PARTITIONS(
par1 VALUES('重庆'),
par2 VALUES('北京'),
par3 VALUES('上海'),
par4 VALUES(OTHERVALUES))
SUBPARTITION BY LIST(addr) SUBPARTITIONS(
subpart1 VALUES ('青羊'),
subpart2 VALUES ('武侯'),
```

```
subpart3 VALUES ('金 牛'),
subpart4 VALUES ('高 新'),
subpart5 VALUES ('锦 江'));
```

6.6.3.2 范围分区

二级分区是对一级分区的再次分割。

> **说明**
>
> 虚谷数据库系统中定义二级分区即默认所有一级分区按照指定二级分区进行再分割，不能针对一级分区进行不同的二级分区划分。

参数说明

- SUBPARTITION BY LIST：关键字，用于指定使用列表二级分区。
- name_list：一个或多个列的列表，用于二级分区的列。
- SUBPARTITIONS：关键字，用于指定二级分区列表。
- sub_part_list：每个二级分区的定义。格式与一级子分区的列表分区一致。

语法格式

```
range_subpartitioning_clause::=
SUBPARTITION BY RANGE (name_list) SUBPARTITIONS (range_parti_item[,
   range_parti_item]...)

sub_part_range::=
(parti_values)
|    (MAXVALUES)
|    subpart_name VALUES LESS THAN (parti_values)
|    subpart_name VALUES LESS THAN (MAXVALUES)
```

示例

创建一个以 city 为一级列表分区，以 id 为二级范围子分区的分区表。

```
CREATE TABLE test_part_ran(id INT,name VARCHAR,city VARCHAR,addr
   VARCHAR)
PARTITION BY LIST(city) PARTITIONS(
part1 VALUES('成 都'),
part2 VALUES('重 庆'),
part3 VALUES('北 京'),
part4 VALUES(OTHERVALUES))
SUBPARTITION BY RANGE(id) SUBPARTITIONS(
partitionsub_1 VALUES LESS THAN(1),
partitionsub_2 VALUES LESS THAN(10),
partitionsub_3 VALUES LESS THAN(23),
partitionsub_4 VALUES LESS THAN (MAXVALUES));
```

6.6.3.3 哈希分区

二级分区是对一级分区的再次分割。

> **说明**
>
> 虚谷数据库系统中定义二级分区即默认所有一级分区按照指定二级分区进行再分割，不能针对一级分区进行不同的二级分区划分。

语法格式

```
hash_subpartitioning_clause::=
SUBPARTITION BY HASH (name_list) SUBPARTITIONS ICONST
|    SUBPARTITION BY HASH (name_list) SUBPARTITIONS (name_list)
```

参数说明

- SUBPARTITION BY HASH：关键字，用于指定使用哈希分区。
- col_list：一个或多个列的列表，用于分区的列。
- SUBPARTITIONS：关键字，用于指定分区的数量。
- ICONST：一个确定整数，系统将根据其值进行表分区划分。
- name_list：分区名称的列表。

示例

创建一个以 address 为一级分区键的列表分区，并以 sex 键进行二级分区，二级分区数为 2。

```
CREATE TABLE test_sub1(
id INT IDENTITY(1,1) NOT NULL ,
name CHAR(8) NOT NULL ,
sex VARCHAR(5) NOT NULL ,
birthday DATETIME NOT NULL,
address VARCHAR NOT NULL )
PARTITION BY LIST(address) PARTITIONS
(('中国'),(OTHERVALUES))
SUBPARTITION BY HASH(sex) SUBPARTITIONS 2;
```

6.6.4 自动扩展分区

虚谷数据库范围分区提供了针对日期时间类型的自动扩展分区功能，分区间隔支持 HOUR、DAY、MONTH、YEAR。

> **说明**
>
> 对于包含固定分区的自动扩展分区表，支持删除除 0 号固定分区以外的其他分区的数据。

语法格式

```
range_auto_partitioning_clause::=
PARTITION BY RANGE (name_list) INTERVAL expr {HOUR|DAY|MONTH|YEAR}
   PARTITIONS (range_parti_item[,range_parti_item]...)
```

参数说明

- PARTITION BY RANGE：关键字，用于指定使用范围分区。
- name_list：一个或多个列的列表，用于分区的列。
- [INTERVAL expr HOUR|DAY|MONTH|YEAR]：用于指定分区的时间间隔。
- PARTITIONS (range_parti_item[,range_parti_item]...)：分区定义
- (parti_values)：一个包含具体值的列表，用于定义分区。

示例

- 示例 1

  创建一个以时间为分区键的自动扩展分区表 range_test。1970-01-01 00:00:00 这个时间点以前的数据落在一个区间内，1970-01-01 00:00:00 时间点包含该时间以后的数据以 5 天为间隔进行分区。

```
-- 创建包含固定分区的自动扩展分区表
CREATE TABLE range_test(id INT,create_time DATETIME)PARTITION BY
   RANGE(create_time)INTERVAL 5 DAY PARTITIONS(('
   1970-01-01 00:00:00'));

-- 查看表分区信息
SQL> SELECT * from SYS_PARTIS;

DB_ID | TABLE_ID | PARTI_NO | PARTI_NAME | PARTI_VAL | GSTO_NOS
    | ONLINE | RESERVED1 | RESERVED2 |
-----------------------------------------------------------
1 | 1048578 | 0 | PART1| '1970-01-01 00:00:00'| 202 | T | <NULL
   >| <NULL>|
```

- 示例 2

  分区管理，删除固定分区报错。

```
-- 创建包含固定分区的自动扩展分区表
SQL> CREATE TABLE test(c1 INTEGER, c2 DATETIME, c3 VARCHAR)
   PARTITION BY RANGE(c2) INTERVAL 1 DAY PARTITIONS(
```

```
p1 VALUES LESS THAN('2000-01-01 00:00:00'),
p2 VALUES LESS THAN('2023-09-30 00:00:00'),
p3 VALUES LESS THAN('2023-10-01 00:00:00')
);

-- 查看表分区信息
SQL> SELECT * from SYS_PARTIS;

DB_ID | TABLE_ID | PARTI_NO | PARTI_NAME | PARTI_VAL | GSTO_NOS
    | ONLINE | RESERVED1 | RESERVED2 |
-----------------------------------------------------
1 | 1048585 | 0 | P1| '2000-01-01 00:00:00'| 201 | T | <NULL>| <
   NULL>|
1 | 1048585 | 1 | P2| '2023-09-30 00:00:00'| 205 | T | <NULL>| <
   NULL>|
1 | 1048585 | 2 | P3| '2023-10-01 00:00:00'| 206 | T | <NULL>| <
   NULL>|

-- 删除自动扩展分区表固定分区返回错误
SQL> ALTER TABLE test DROP PARTITION p1;
Error: [E21093] 不允许删除自动扩展分区表的固定分区

-- 删除除固定分区以外的其他分区
SQL> ALTER TABLE test DROP PARTITION p2;
SQL> ALTER TABLE test DROP PARTITION p3;

-- 删除p1、p2分区后查看表分区信息
SQL> SELECT * FROM SYS_PARTIS;

DB_ID | TABLE_ID | PARTI_NO | PARTI_NAME | PARTI_VAL | GSTO_NOS
    | ONLINE | RESERVED1 | RESERVED2 |
-----------------------------------------------------
1 | 1048586 | 0 | P1| '2000-01-01 00:00:00'| 201 | T | <NULL>| <
   NULL>|
```

6.6.5 按分区访问数据

虚谷数据库针对分区表提供了按分区访问数据的功能，用户可以在查询语句中指定访问任意分区的数据。

语法格式

```
parti_query::=
SELECT column_list FROM table_name opt_parti_clip_clause;

opt_parti_clip_clause::=
/*EMPTY*/
|   PARTITION ( part_name_list )
|   SUBPARTITION ( subpart_name_list )
```

参数说明

- column_list：访问目标列。

- table_name：访问目标表。
- opt_parti_clip_clause：指定访问目标分区。
- part_name_list：一级分区的分区名列表，多个分区以逗号分隔。
- subpart_name_list：二级分区的分区名列表，多个分区以逗号分隔。

示例

- 示例 1

  分别指定查询分区的数据。

```
-- 创建表test_query1，并创建一级分区
CREATE TABLE test_query1(id INT,name VARCHAR,city VARCHAR)
PARTITION BY LIST(city) PARTITIONS(
part1 VALUES('成都'),
part2 VALUES('重庆'),
part3 VALUES('北京'),
part4 VALUES(OTHERVALUES));

-- 插入数据
INSERT INTO test_query1 VALUES(1,'路人1','成都');
INSERT INTO test_query1 VALUES(2,'路人2','重庆');
INSERT INTO test_query1 VALUES(3,'路人3','重庆');
INSERT INTO test_query1 VALUES(4,'路人4','成都');
INSERT INTO test_query1 VALUES(5,'路人5','成都');
INSERT INTO test_query1 VALUES(6,'路人6','北京');

-- 查询part1的数据并按id进行排列
SELECT id,name,city FROM test_query1 PARTITION(part1) ORDER BY id
   ;

ID | NAME | CITY |
-------------------------------------------------------------
1 | 路人1| 成都|
4 | 路人4| 成都|
5 | 路人5| 成都|

-- 查询part1和part3的数据并按id进行排列
SELECT id,name,city FROM test_query1 PARTITION(part1,part3) ORDER
    BY id;

ID | NAME | CITY |
-------------------------------------------------------------
1 | 路人1| 成都|
4 | 路人4| 成都|
5 | 路人5| 成都|
6 | 路人6| 北京|
```

- 示例 2

  分别指定查询二级分区的数据。

```
-- 创建表test_query2，并创建一级分区和二级分区
CREATE TABLE test_query2(id INT,name VARCHAR,city VARCHAR,addr
   VARCHAR)
PARTITION BY LIST(city) PARTITIONS(
part1 VALUES('成都'),
part2 VALUES('北京'),
part3 VALUES(OTHERVALUES))
SUBPARTITION BY LIST(addr) SUBPARTITIONS(
subpart1 VALUES ('高新区'),
subpart2 VALUES ('锦江区'),
subpart3 VALUES ('西城区'),
subpart4 VALUES ('东城区'));

-- 插入数据
INSERT INTO test_query2 VALUES(1,'路人1','成都','高新区');
INSERT INTO test_query2 VALUES(2,'路人2','成都','锦江区');
INSERT INTO test_query2 VALUES(3,'路人3','成都','锦江区');
INSERT INTO test_query2 VALUES(4,'路人4','北京','西城区');
INSERT INTO test_query2 VALUES(5,'路人5','北京','西城区');
INSERT INTO test_query2 VALUES(6,'路人6','北京','东城区');

-- 查询subpart2的数据并按id进行排列
SELECT id,name,city,addr FROM test_query2 SUBPARTITION(subpart2)
   ORDER BY id;

ID | NAME | CITY | ADDR |
-------------------------------------------------------------
2 | 路人2| 成都| 锦江区|
3 | 路人3| 成都| 锦江区|

-- 查询subpart2和subpart3的数据并按id进行排列
SELECT id,name,city,addr FROM test_query2 SUBPARTITION(subpart2,
   subpart3) ORDER BY id;

ID | NAME | CITY | ADDR |
-------------------------------------------------------------
2 | 路人2| 成都| 锦江区|
3 | 路人3| 成都| 锦江区|
4 | 路人4| 北京| 西城区|
5 | 路人5| 北京| 西城区|
```

说明

通过 partition 和 subpatition 关键字指定分区查找语法和 PG 指定表别名以及输出字段名语法冲突。设计上表现为不抛出语法错误。若要查询不同分区数据，应严格参照以上语法进行查询。

6.7 清空表

语法格式

```
truncatestmt::=
```

```
TRUNCATE [ TABLE ] sche_name.table_name [NOWAIT | WAIT | WAIT
   ICONST]
```

参数说明

- schema_name：模式名。
- table_name：表名。

> **说明**
>
> TRUNCATE 是将表中所有的记录全部清除，而 DELETE 语句带上条件后是选择性地清除记录。如果不带条件类似于 TRUNCATE，如 DELETE FROM student;。TRUNCATE 操作可收回数据存储空间用于系统复用，而 DELETE 只是在行级上标示该记录被删除，不会回收数据存储空间。

示例

MySQL 兼容模式下，TRUNCATE 表后，该表的自增字段重置为 1。

```
-- 设置兼容模式为MySQL
SET compatible_mode TO 'MYSQL';

-- 创建表，并插入数据
CREATE TABLE customers(id int IDENTITY, tel VARCHAR(11));

INSERT INTO customers(tel) values('cdscdsc');

SELECT * FROM customers c ;
ID|TEL    |
--+-------+
1|cdscdsc|

-- 清空表
TRUNCATE table customers ;

-- 再次插入数据，并查询可以看到id字段是重置为1开始
INSERT INTO customers(tel) values('cdscdsc');
INSERT INTO customers(tel) values('cdscdsc');

SELECT * FROM customers c ;
ID|TEL    |
--+-------+
1|cdscdsc|
2|cdscdsc|
```

6.8 数据操作

6.8.1 INSERT

添加一个或多个记录到表中。

6.8.1.1 主要语法结构

语法格式

```
INSERT INTO [schema_name.] table_name
[PARTITION ( part_name_list ) | SUBPARTITION ( subpart_name_list )]
insert_rest
[RETURNING target_list opt_bulk opt_into_list];
```

参数说明

- schema_name：模式名。
- table_name：表名。
- part_name_list：表的分区名。
- subpart_name_list：表的子分区名。

6.8.1.2 插入内容 insert_rest

语法格式

```
insert_rest::=
VALUES ( target_list )[( target_list ) | ,( target_list )]...
|    DEFAULT VALUES
|    selectstmt
|    ( columnList ) VALUES ( target_list )[( target_list ) | ,(
   target_list )]...
|    ( columnList ) selectstmt
|    VALUES ident
|    ( columnList ) VALUES ident
```

参数说明

- VALUES (target_list)[(target_list) | ,(target_list)]...：直接插入一组或多组值。
- DEFAULT VALUES：插入默认值，适用于所有列都有默认值的情况。
- selectstmt：一个 SELECT 语句，返回要插入的数据。从子查询中插入数据。
- (columnList) VALUES (target_list)[(target_list) | ,(target_list)]...：指定列列表并插入一组或多组值。(target_list) 插入的数据有以下形式
  - (expr1, expr2, ..., exprn)，括号中以逗号分隔的每项可以是常值、表达式等，每项与要插入数据的列对应。
  - (expr1, expr2, ..., exprn) (expr1, expr2, ..., exprn) ... 等，用以在一条语句里插入多行数据。

- (expr1, expr2, ..., exprn),(expr1, expr2, ..., exprn),... 等，使用英文逗号分隔每一行插入的数据，插入多行数据。

- (columnList) selectstmt：指定列列表并从子查询中插入数据。
- VALUES ident：插入一个标识符（通常是变量或参数）。
- (columnList) VALUES ident ：指定列列表并插入一个标识符（通常是变量或参数）。

6.8.1.3 返回结果 target_list

语法格式

```
target_list::=
target_el[,target_el]...

target_el::=
b_expr AS ColLabel
|    b_expr ColId
|    DEFAULT
|    b_expr
|    ident  .*
|    *

opt_bulk::=
/*EMPTY*/
|    BULK COLLECT

opt_into_list::=
INTO ident[,ident]...
|    /*empty*/
```

参数说明

- target_list：可选项，RETURNING 后的 target_list 用来指定返回的内容，有以下形式：
  - b_expr：是一个表达式，它可以是一个简单的列名、计算表达式等。
  - AS ColLabel 或 ColId：用于给返回的结果列重命名。
  - DEFAULT：关键字，可以用来表示默认值。
  - ident .* 和 * 用于选择所有列。
- opt_bulk：可选的关键词，指定该关键词后结果将一次性输出，而不是每次输出一行数据。
- opt_into_list：可选的关键词, 把返回的结果输出至指定的变量或数据结构中。

示例

- 示例 1 不指定插入列，插入值为常值。

```
INSERT INTO test_tab_1 VALUES(DEFAULT, NULL, '2000-01-01', 2.459E
   +23);
```

- 示例 2 指定插入列，插入值为表达式。

```
INSERT INTO test_tab_2(c3, c4) VALUES(2+3, SUBSTR(c3,LEN(c3)-3,1)
   );
```

- 示例 3 指定插入列，插入值为子查询。

```
INSERT INTO test_tab_3(c3, c4)SELECT salary,addr FROM t2 WHERE t2
   .id > 10;
```

- 示例 4 一条语句中插入多行数据。

```
INSERT INTO test_tab_4 VALUES(1, 'alpha')(DEFAULT, '')(NULL, '
   beta')(4, 'gamma');
```

- 示例 5 插入时将结果 RETURNING。

```
DECLARE
TYPE type_table_varchar IS TABLE OF VARCHAR;
var_chr TYPE_TABLE_VARCHAR;
BEGIN
FOR i IN 1..10 LOOP
INSERT INTO test_tab_5(id, c2, c3) VALUES(i, i*2.1, 'STRING_'||i)
RETURNING '('||c2||','||c3||')' BULK COLLECT INTO var_chr;
-- 将返回结果输出至屏幕
FOR j IN 1..var_chr.COUNT() LOOP
SEND_MSG(var_chr(j));
END LOOP;
END LOOP;
END;
```

- 示例 6 插入 RECORD 类型。

```
-- 创建表1
CREATE TABLE t1(a INT,name VARCHAR(100));

-- 插入t1表数据
INSERT INTO t1 VALUES(1,'123')(2,'123')(3,'345');

-- 创建表2
CREATE TABLE t2(a INT,name VARCHAR(100));

-- 插入t2数据
DECLARE
TYPE r IS TABLE OF t1%ROWTYPE;
rs r;
BEGIN
SELECT * BULK COLLECT INTO rs FROM t1;
FOR i IN rs.FIRST ..rs.LAST loop
```

```
INSERT INTO t2 VALUES rs(i);
END LOOP;
END;
```

6.8.2 INSERT IGONRE

使用 INSERT INTO 语句向表插入多行数据时，可能会在语法层、规划层或执行层遇到错误，INSERT INTO 语句会直接中止当前插入操作返回错误信息，只会向表插入当前错误行之前的行数据，剩下的行数据也会被中断。

INSERT IGNORE INTO 语句忽略插入多行数据时一些违反约束信息导致的错误行，过滤不满足约束的行数据，将正确行数据都插入表中。

> 说明
>
> 忽略执行层检查约束信息时遇到的错误，而语法层、规划层遇到的错误依旧正常报错。

6.8.2.1 主要语法结构

语法格式

```
ignore_stmt::=
INSERT IGNORE INTO [schema_name.] table_name [PARTITION (
   part_name_list ) | SUBPARTITION ( subpart_name_list )]
   insert_rest
```

参数说明

- schema_name：模式名。
- table_name：表名。
- part_name_list：表的分区名。
- subpart_name_list：表的子分区名。

6.8.2.2 插入内容 insert_rest

语法格式

```
insert_rest::=
VALUES ( target_list )[( target_list ) | ,( target_list )]...
|    DEFAULT VALUES
|    selectstmt
|    ( columnList ) VALUES ( target_list )[( target_list ) | ,(
   target_list )]...
|    ( columnList ) selectstmt
```

```
|    VALUES ident
|    ( columnList ) VALUES ident
```

参数说明

- VALUES (target_list)[(target_list) | ,(target_list)]...：直接插入一组或多组值。
- DEFAULT VALUES：插入默认值，适用于所有列都有默认值的情况。
- selectstmt：一个 SELECT 语句，返回要插入的数据。从子查询中插入数据。
- (columnList) VALUES (target_list)[(target_list) | ,(target_list)]...：指定列列表并插入一组或多组值。(target_list) 插入的数据有以下形式
  - (expr1, expr2, ..., exprn)，括号中以逗号分隔的每项可以是常值、表达式等，每项与要插入数据的列对应。
  - (expr1, expr2, ..., exprn) (expr1, expr2, ..., exprn) ... 等，用以在一条语句里插入多行数据。
  - (expr1, expr2, ..., exprn),(expr1, expr2, ..., exprn),... 等，使用英文逗号分隔每一行插入的数据，插入多行数据。
- (columnList) selectstmt：指定列列表并从子查询中插入数据。
- VALUES ident：插入一个标识符（通常是变量或参数）。
- (columnList) VALUES ident ：指定列列表并插入一个标识符（通常是变量或参数）。

6.8.2.3 示例

- 示例 1 同时执行多行插入命令，向含有唯一值约束的表插入三行数据，其中前两行为相同的数据，若不带 IGNORE 关键字，插入第二行数据时会检测约束信息限制，终止当前插入操作返回错误信息，因此只有第一行数据插入成功。若带 IGNORE 关键字则忽略当前行的数据插入操作，继续执行下一行数据插入操作，因此第一和第三行数据插入成功。

```
CREATE TABLE table_1(id INT,name VARCHAR,CONSTRAINT u_1 UNIQUE(id
   ));

--INSERT 同时插入多行数据
INSERT INTO table_1 VALUES(1,'abc');
INSERT INTO table_1 VALUES(1,'abc');
INSERT INTO table_1 VALUES(2,'abc');

--只有第一行数据插入成功
SELECT * FROM table_1;
```

```
--INSERT IGNORE 同时插入多行数据
INSERT IGNORE INTO table_1 VALUES(1,'abc');
INSERT IGNORE INTO table_1 VALUES(1,'abc');
INSERT IGNORE INTO table_1 VALUES(2,'abc');

--过滤第二行错误，成功插入第一、三行数据
SELECT * FROM table_1;
```

- 示例 2 执行一个插入命令，向含有唯一值约束的表插入三行数据，其中前两行为相同的数据，若不带 IGNORE 关键字，插入第二行数据时会检测到约束信息限制，终止当前插入操作返回错误信息，因此当前插入命令失败，没有数据插入。若带 IGNORE 关键字则忽略当前行的数据错误，继续执行下一行数据插入操作，因此第一和第三行数据插入成功。

```
--INSERT 同时插入多行数据
INSERT INTO table_1 VALUES(3,'abc')(3,'abc')(4,'abc');

--没有数据插入成功
SELECT * FROM table_1;

--INSERT IGNOR 同时插入多行数据
INSERT IGNORE INTO table_1 VALUES(3,'abc')(3,'abc')(4,'abc');

--过滤第二组错误，成功插入第一、三组数据
SELECT * FROM table_1;
```

6.8.3 INSERT REPLACE

INSERT REPLACE INTO 语句用于检查插入数据时的主键冲突或唯一索引冲突，若有冲突就会先删除之前的数据，然后再执行插入，若没有冲突则与 INSERT INTO 一样执行插入。

6.8.3.1 主要语法结构

语法格式

```
replace_stmt::=
REPLACE INTO [schema_name.] table_name [PARTITION ( part_name_list
    ) | SUBPARTITION ( subpart_name_list )] insert_rest
```

参数说明

- schema_name：模式名。
- table_name：表名。
- part_name_list：表的分区名。
- subpart_name_list：表的子分区名。

6.8.3.2 插入内容 insert_rest

语法格式

```
insert_rest::=
VALUES ( target_list )[( target_list ) | ,( target_list )]...
|   DEFAULT VALUES
|   selectstmt
|   ( columnList ) VALUES ( target_list )[( target_list ) | ,(
   target_list )]...
|   ( columnList ) selectstmt
|   VALUES ident
|   ( columnList ) VALUES ident
```

参数说明

- VALUES (target_list)[(target_list) | ,(target_list)]...：直接插入一组或多组值。
- DEFAULT VALUES：插入默认值，适用于所有列都有默认值的情况。
- selectstmt：一个 SELECT 语句，返回要插入的数据。从子查询中插入数据。
- (columnList) VALUES (target_list)[(target_list) | ,(target_list)]...：指定列列表并插入一组或多组值。(target_list) 插入的数据有以下形式
  - (expr1, expr2, ..., exprn)，括号中以逗号分隔的每项可以是常值、表达式等，每项与要插入数据的列对应。
  - (expr1, expr2, ..., exprn) (expr1, expr2, ..., exprn) ... 等，用以在一条语句里插入多行数据。
  - (expr1, expr2, ..., exprn),(expr1, expr2, ..., exprn),... 等，使用英文逗号分隔每一行插入的数据，插入多行数据。
- (columnList) selectstmt：指定列列表并从子查询中插入数据。
- VALUES ident：插入一个标识符（通常是变量或参数）。
- (columnList) VALUES ident ：指定列列表并插入一个标识符（通常是变量或参数）。

6.8.3.3 示例

- 示例 1

  同时执行多行插入命令，向含有唯一值约束的表插入三行数据，其中前两行为相同的数据，若不带 REPLACE 关键字，插入第二行数据时会检测到约束冲突，终止当前插入操作返回错误信息，因此只有第一行数据插入成功。若带 REPLACE 关键字则检查约束冲突，

若存在冲突则删除原有数据，插入最新数据，因此第一和第三行数据插入成功，第二行数据更新成最新值。

```
CREATE TABLE table_2(i INT IDENTITY(1,1),id int,name VARCHAR
   CONSTRAINT d_1 DEFAULT 'default',CONSTRAINT u_1 UNIQUE(id));

--INSERT 插入多行数据，违反自增序列的唯一值约束
INSERT INTO table_2(id,name) VALUES(1,'abc');
INSERT INTO table_2(id,name) VALUES(1,'bbb');
INSERT INTO table_2(id,name) VALUES(2,'abc');

--只有第一行数据插入成功
SELECT * FROM table_2;

--REPLACE 插入多行数据
REPLACE INTO table_2(id,name) VALUES(1,'abc');
REPLACE INTO table_2(id,name) VALUES(1,'bbb');
REPLACE INTO table_2(id,name) VALUES(2,'abc');

--第二行数据被替换，新插入第一、三行数据
SELECT * FROM table_2;

--REPLACE 字段取默认值
REPLACE INTO table_2(id) VALUES(3);

--数据被替换成默认值
SELECT * FROM table_2;
```

- 若 REPLACE 的数据段不包含全部的数据，且字段含义默认值，则未指定的数据会被替换成默认值。
- 若行存在自增字段，REPLACE 含冲突的字段，对应的行该字段也会自增 1。

- 示例 2

执行一个插入命令，向含有唯一值约束的表插入三行数据，其中前两行为相同的数据，若不带 REPLACE 关键字，插入第二行数据时会检测到约束信息限制，终止当前插入操作返回错误信息，没有数据插入成功。若带 REPLACE 关键字则检查约束冲突，若存在冲突则删除原有数据，插入最新数据，不冲突数据照常插入。

```
    --INSERT 同时插入多行数据
    INSERT INTO table_1 VALUES(3,'abc')(3,'abc')(4,'abc');

    --没有数据插入成功
    SELECT * FROM table_1;

    --INSERT IGNOR 同时插入多行数据
    INSERT IGNORE INTO table_1 VALUES(3,'abc')(3,'abc')(4,'abc');

    --过滤第二组错误，成功插入第一、三组数据
    SELECT * FROM table_1;
```

6.8.4 MULTITABLE INSERT

利用 INSERT FIRST/ALL 使得 INSERT 语句可以同时插入多张表，还可以根据判断条件来决定每条记录插入到哪张或哪几张表中。区别如下：

- INSERT FIRST：对于每一行数据，只插入到第一个 WHEN 条件成立的表，不继续检查其他条件。
- INSERT ALL：对于每一行数据，对每一个 WHEN 条件都进行检查，如果满足条件就执行插入操作。

6.8.4.1 主要语法结构

语法格式

```
multi_stmt::=
INSERT {FIRST | ALL} WHEN bool_expr THEN insert_into_list [WHEN
   bool_expr THEN insert_into_list]... ELSE insert_into_list
   select_no_parens
|    INSERT ALL insert_into_list select_no_parens
```

参数说明

- ALL：如果指定 ALL，则数据库将检查每个 WHEN 子句，如果每一个 WHEN 子句均为 true，数据库执行相应的插入操作。
- FIRST：如果指定 FIRST，则数据库将按照子句在语句中出现的顺序检查 WHEN 子句，对于检查为 true 的第一个 WHEN 子句，数据库将执行 insert_into_list 子句。
- ELSE：对于给定的行，WHEN 子句均为 false 且指定了 ELSE 子句，则数据库将执行 ELSE 子句后的 insert_into_list。
- select_no_parens：查询语句。

6.8.4.2 插入内容 insert_rest

语法格式

```
insert_into_list::=
insert_into_clause [insert_into_clause]...

insert_into_clause::=
INTO name_space [PARTITION (name_list) | SUBPARTITION (name_list)
   ] [(columnList)] VALUES (target_list)
|    INTO name_space [PARTITION (name_list) | SUBPARTITION (
   name_list)] (columnList)
```

```
|   INTO name_space [PARTITION (name_list) | SUBPARTITION (
  name_list)] %prec UMINUS
```

参数说明

- name_space：指定要插入数据的目标表或视图。
- [PARTITION (name_list) | SUBPARTITION (name_list)]：可选部分，指定插入到表的特定分区或子分区。
- VALUES (target_list)：当使用 VALUES 关键字时，直接提供要插入的具体值列表。target_list 是一系列表达式，对应于 columnList 中指定的列。
- columnList：如果没有 VALUES 关键字，从 select_no_parens 查询结果中获取的列。

6.8.4.3 示例

- 示例 1

  INSERT FIRST

```
SQL> CREATE TABLE ins_tab1(col1 INT,col2 VARCHAR(20),col3 NUMERIC
   (5,2),col4 DATE);

SQL> INSERT INTO ins_tab1 VALUES(11,'AB',111.11,TO_DATE('
   2007-04-15','YYYY-MM-DD'))(6,'CD',222.22,TO_DATE('2008-04-15',
   'YYYY-MM-DD'))(1,'EF',333.33,TO_DATE('2009-04-15','YYYY-MM-DD'
   ))(4,'GH',444.44,TO_DATE('2010-04-15','YYYY-MM-DD'))(NULL,'GH'
   ,555.55,TO_DATE('2011-04-15','YYYY-MM-DD'));

SQL> CREATE TABLE ins_tab2(col1 INT,col2 VARCHAR(20),col3 NUMERIC
   (5,2),col4 DATE);

SQL> CREATE TABLE ins_tab3(col1 INT,col2 VARCHAR(20),col3 NUMERIC
   (5,2),col4 DATE);

SQL> INSERT FIRST WHEN col1>10 THEN INTO ins_tab2 VALUES(col1,
   col2,col3,col4) WHEN col1>3 THEN INTO ins_tab3 VALUES(col1,
   col2,col3,col4) SELECT col1,col2,col3,col4 FROM ins_tab1;

SQL> SELECT * FROM ins_tab2;

COL1 | COL2 | COL | COL4 |
-------------------------------------------------------------
11 | AB| 111.11| 2007-04-15 AD |

SQL> SELECT * FROM ins_tab3;

COL1 | COL2 | COL | COL4 |
-------------------------------------------------------------
6 | CD| 222.22| 2008-04-15 AD |
4 | GH| 444.44| 2010-04-15 AD |
```

- 示例 2

  INSERT ALL

```
SQL> CREATE TABLE ins_tab1(col1 INT,col2 VARCHAR(20),col3 NUMERIC
   (5,2),col4 DATE);

SQL> INSERT INTO ins_tab1 VALUES(11,'AB',111.11,TO_DATE('
   2007-04-15','YYYY-MM-DD'))(6,'CD',222.22,TO_DATE('2008-04-15',
   'YYYY-MM-DD'))(1,'EF',333.33,TO_DATE('2009-04-15','YYYY-MM-DD'
   ))(4,'GH',444.44,TO_DATE('2010-04-15','YYYY-MM-DD'))(NULL,'GH'
   ,555.55,TO_DATE('2011-04-15','YYYY-MM-DD'));

SQL> CREATE TABLE ins_tab2(col1 INT,col2 VARCHAR(20),col3 NUMERIC
   (5,2),col4 DATE);

SQL> CREATE TABLE ins_tab3(col1 INT,col2 VARCHAR(20),col3 NUMERIC
   (5,2),col4 DATE);

SQL> INSERT ALL WHEN col1>10 THEN INTO ins_tab2 VALUES(col1,col2,
   col3,col4) WHEN col1>3 THEN INTO ins_tab3 VALUES(col1,col2,
   col3,col4) SELECT col1,col2,col3,col4 FROM ins_tab1;

SQL> SELECT * FROM ins_tab2;

COL1 | COL2 | COL | COL4 |
-------------------------------------------------------------
11 | AB| 111.11| 2007-04-15 AD |

SQL> SELECT * FROM ins_tab3;

COL1 | COL2 | COL | COL4 |
-------------------------------------------------------------
11 | AB| 111.11| 2007-04-15 AD |
6 | CD| 222.22| 2008-04-15 AD |
4 | GH| 444.44| 2010-04-15 AD |
```

6.8.5 UPDATE

修改表中现有的的数据。

6.8.5.1 主要语法结构

语法格式

更新指定表中的数据。

```
UPDATE base_table_refs SET update_target_list [ opt_from_clause
    ] [ opt_where_clause ] [ opt_returning ] [ opt_bulk ] [
   opt_into_list ]
```

更新当前游标指向的数据。

```
UPDATE base_table_refs SET update_target_list [ opt_from_clause
    ] [ opt_where_clause ] [ opt_returning ] [ opt_bulk ] [
   opt_into_list ]
```

通过子查询来设置更新的目标值。

```
UPDATE base_table_refs SET update_target_list [ opt_from_clause
    ] [ opt_where_clause ] [ opt_returning ] [ opt_bulk ] [
   opt_into_list ]
```

参数说明

- base_table_refs：更新的表或视图名，前面也可加上相应的模式名。
- update_target_list：更新的目标，可以是多个用逗号分隔的部分。
- select_with_parens：查询语句。
- opt_from_clause：可选的 from 从句，用于从另一个或多个表中查询结果作为目标表列的更新值。
- opt_where_clause：可选的 where 从句，用于对从表中查询出的数据做过滤或限制。
- WHERE CURRENT OF name：通常用于游标更新时的定位条件，其中 name 为游标变量的名字。
- opt_returning、opt_into_list：可选的关键词，用于把更新后的结果输出至指定的变量或数据结构。
- opt_bulk：可选关键词，用于修饰 opt_returning opt_into_list 关键词，指定该关键词后把结果一次性输出，而不是每次仅输出一行数据。

说明

- 多表插入前面的条件表达式和 VALUES 中的字段必须出现在后续的查询输出列中。
- 多表插入时插入对象不支持为视图。
- 如下为与 oracle 设计差异部分：
  - 情况 1：如果条件字段不在后续查询输出字段中不报错可插入成功。
  - 情况 2：如果插入 value 中的字段不在后续查询输出字段中不报错可插入成功。

6.8.5.2 目标表引用 base_table_refs

语法格式

```
base_table_refs::=
base_table_ref [,base_table_ref]...

base_table_ref::=
relation_expr
|   relation_expr alias_clause

relation_expr::=
name_space [PARTITION (name_list) | SUBPARTITION (name_list)]

alias_clause::=
AS ColId (name_list)
|   AS ColId
|   ColId (name_list)
|   ColId
```

参数说明

- name_space [PARTITION (name_list) | SUBPARTITION (name_list)]：name_space 是表名或视图名，而 [PARTITION (name_list) | SUBPARTITION (name_list)] 部分用于指定要更新的表的分区或子分区。
- AS ColId (name_list)：为表指定别名，并为该表中的列指定新的名称。
- AS ColId：仅指定表的别名。
- ColId (name_list)：为表指定别名，并为该表中的列指定新的名称，但没有 AS 关键字。
- ColId：仅指定表的别名，没有 AS 关键字。

6.8.5.3 更新目标列表 update_target_list

语法格式

```
update_target_list::=
[ table_alias ] column_name = expr
[, [ table_alias ] column_name = expr]...
```

参数说明

- table_alias：表名或表的别名，该部分为可选项。
- column_name：列名。
- expr：是目标要更新成的值的表达式，具体又可分为：固定值（NULL 或非 NULL 的常值）、关键字 DEFAULT、运算表达式、子查询（查询结果不能大于 1 行）。

6.8.5.4 示例

- 示例 1 更新结果为常值。

```
UPDATE t2 SET c2 = 'new_a_1', c3 = DEFAULT WHERE c1 = 5;
```

- 示例 2 更新结果为子查询。

```
UPDATE t2 SET (c2, c3) = (SELECT MAX(c2), MIN(c3)+10 FROM t1)
   WHERE c1 = 6;
```

- 示例 3 更新时将结果 RETURNING。

```
DECLARE
TYPE type_table_1 IS TABLE OF T1%ROWTYPE;      -- UDT类型：T1 行类
   型的嵌套表
var_t1 TYPE_TABLE_1;                           -- UDT类型变量，用来
   保存游标中查询的数据
TYPE type_table_int IS TABLE OF t2.c2%TYPE;    -- UDT类型：INT 类型
   的嵌套表
var_int TYPE_TABLE_INT;                        -- UDT类型变量，用来
   保存 RETURNING 的值
CURSOR cur IS SELECT * FROM t1;
BEGIN
OPEN cur;
FETCH cur BULK COLLECT INTO var_t1;
UPDATE t2
SET c2 = var_t1(1).c2,
c3 = c3 + 100
RETURNING c1 BULK COLLECT INTO var_int;
CLOSE cur;
-- 如果需要，可把 RETUNING 的数据输出至屏幕或表中
FOR i IN 1..var_int.COUNT() LOOP
SEND_MSG(var_int(i));
END LOOP;
END;
```

6.8.6 DELETE

删除表中数据。

语法格式

```
deletestmt::=
DELETE del_targ_tab [ opt_from_clause ] [ opt_where_clause ] [
   opt_returning ] [ opt_bulk ] [ opt_into_list ]
|    DELETE del_targ_tab [ opt_from_clause ] WHERE CURRENT OF name
    [ opt_returning ] [ opt_bulk ] [ opt_into_list ]
```

参数说明

- del_targ_tab：删除数据的表或视图名，前面也可加上相应的模式名；名称前可加或不加关键字 FROM。

- opt_from_clause：可选的 FROM 从句，用于从另一个或多个表中查询结果作为目标删除数据的定位条件。
- opt_where_clause：可选的 WHERE 从句，用于对从表中查询出的数据做过滤或限制。
- WHERE CURRENT OF 语法：通常用于游标删除时的定位条件。
- opt_returning、opt_into_list：可选的关键词，用于把删除后的数据输出至指定的变量或数据结构。
- opt_bulk：可选关键词，用于修饰 opt_returning opt_into_list 关键词，指定该关键词后把结果一次性输出，而不是每次输出一行数据。

示例

- 示例 1 更新结果为常值。

```
DELETE t1 WHERE c3 = '2005-05-05';
```

- 示例 2 更新结果为子查询。

```
DELETE FROM t2 b FROM t1 a WHERE a.c1 = b.c1 AND a.c3 > '
   2003-01-01';
```

- 示例 3 更新时将结果 RETURNING。

```
DECLARE
TYPE type_table_1 IS TABLE OF t1%ROWTYPE;   -- UDT类型：T1 行类型
   的嵌套表
var_t1 TYPE_TABLE_1;                        -- UDT类型变量，用来保
   存游标中查询的数据
TYPE type_c3 IS TABLE OF t2.c2%TYPE;        -- UDT类型：DATE 类型
   的嵌套表
var_c3 TYPE_C3;                             -- UDT类型变量，用来保
   存 RETURNING 的值
CURSOR cur IS SELECT * FROM t1;
BEGIN
OPEN cur;
FETCH cur BULK COLLECT INTO var_t1;
DELETE FROM t2 WHERE t2.c1 = (var_t1(4).c1 + 1)
RETURNING c3 BULK COLLECT INTO var_c3;
-- 如果需要，可把 RETURNING 的数据输出至屏幕或表中
FOR i IN 1..var_c3.COUNT() LOOP
SEND_MSG(var_c3(i));
END LOOP;
END;
```

6.8.7 MERGE INTO

MERGE INTO 语句可以同时实现 UPDATE 和 INSERT 的功能。

> **说明**
>
> 目前 MERGE INTO 功能暂不支持 UPDATE SQL 中带 Delete 的操作。

6.8.7.1 主要语法结构

语法格式

WHEN MATCHED THEN UPDATE：该子句指定目标表的新列值。如果 ON 子句的条件为真，则执行此更新。

```
MERGE INTO [ schema. ] { table | view } [ t_alias ]
USING { [ schema. ] { table | view } | subquery } [ t_alias ]
ON (bool_expr)
WHEN MATCHED THEN
UPDATE SET
column = { expr | DEFAULT },
...
[WHERE bool_expr]
[DELETE WHERE bool_expr]
```

WHEN NOT MATCHED THEN INSERT：如果 ON 子句的条件为 false，则该子句指定要插入到目标表列中的值。如果合并插入关键字之后省略列，则目标表中的列数必须与 values 子句中的字段数相匹配。

```
MERGE INTO [ schema. ] { table | view } [ t_alias ]
USING { [ schema. ] { table | view } | subquery } [ t_alias ]
ON (bool_expr)
WHEN NOT MATCHED THEN
INSERT insert_merge
[WHERE bool_expr]
```

同时判断：

```
MERGE INTO [ schema. ] { table | view } [ t_alias ]
USING { [ schema. ] { table | view } | subquery } [ t_alias ]
ON (bool_expr)
WHEN MATCHED THEN
UPDATE SET
column = { expr | DEFAULT },
...
[WHERE bool_expr]
[DELETE WHERE bool_expr]
WHEN NOT MATCHED THEN
INSERT insert_merge
[WHERE bool_expr]
```

参数说明

- schema：可选，表或视图所在的模式名。

- table | view：目标表或视图的名称。
- t_alias：可选，为目标表或视图指定别名。
- bool_expr：条件表达式。
- USING [schema.] table | view | subquery
- [t_alias]：指定源表、视图或子查询及其别名。

6.8.7.2 插入值 insert_merge

语法格式

```
insert_merge::=
VALUES insert_values
|    DEFAULT VALUES
|    (columnList2) VALUES insert_values

insert_values::=
(target_list)
|    insert_values (target_list)
|    insert_values,(target_list)
```

参数说明

- VALUES insert_values：插入特定值。
- insert_values：
  - (target_list): 一组值。
  - insert_values (target_list): 多组值。
  - insert_values,(target_list): 多组值，用逗号分隔。
- DEFAULT VALUES：插入所有列的默认值。
- (columnList2) VALUES insert_values：指定要插入的列及其值。
- columnList2：列名列表。

6.8.7.3 示例

- 示例 1

  只 INSERT。

```
CREATE TABLE a_merge(id NUMBER NOT NULL,name VARCHAR2(12) NOT
   NULL,year NUMBER);
```

```
CREATE TABLE b_merge(id NUMBER NOT NULL,aid NUMBER NOT NULL,name
   VARCHAR2(12) NOT NULL,year NUMBER,city VARCHAR2(12));

INSERT INTO a_merge VALUES(1,'liuwei',20)(2,'zhangbin',21)(3,'
   fuguo',20);
INSERT INTO b_merge VALUES(1,2,'zhangbin',30,'吉林')(2,4,'yihe'
   ,33,'黑龙江')(3,3,'fuguo',44,'山东');

MERGE INTO a_merge a USING (SELECT b.aid,b.name,b.year FROM
   b_merge b) c ON(a.id=c.aid)
WHEN NOT MATCHED THEN
INSERT(a.id,a.name,a.year) VALUES(c.aid,c.name,c.year);

SELECT * FROM a_merge;

ID | NAME | YEAR |

-----------------------------------------------------

1| liuwei| 20|
2| zhangbin| 21|
3| fuguo| 20|
4| yihe| 33|
```

- 示例 2

  只 UPDATE。

```
CREATE TABLE a_merge(id NUMBER NOT NULL,name VARCHAR2(12) NOT
   NULL,year NUMBER);
CREATE TABLE b_merge(id NUMBER NOT NULL,aid NUMBER NOT NULL,name
   VARCHAR2(12) NOT NULL,year NUMBER,city VARCHAR2(12));

INSERT INTO a_merge VALUES(1,'LIUWEI',20)(2,'ZHANGBIN',21)(3,'
   FUGUO',20);
INSERT INTO b_merge VALUES(1,2,'ZHANGBIN',30,'吉林')(2,4,'YIHE'
   ,33,'黑龙江')(3,3,'FUGUO',44,'山东');

MERGE INTO a_merge a USING (SELECT b.aid,b.name,b.year FROM
   b_merge b) c ON(a_merge.id=c.aid) WHEN MATCHED THEN UPDATE SET
    a_merge.year=c.year;

SELECT * FROM a_merge;

ID | NAME | YEAR |

-----------------------------------------------------

1| LIUWEI| 20|
2| ZHANGBIN| 30|
3| FUGUO| 44|
```

- 示例 3

  匹配时更新，不匹配时新增。

```
CREATE TABLE a_merge(id NUMBER NOT NULL,name VARCHAR2(12) NOT
   NULL,year NUMBER);
CREATE TABLE b_merge(id NUMBER NOT NULL,aid NUMBER NOT NULL,name
   VARCHAR2(12) NOT NULL,year NUMBER,city VARCHAR2(12));

INSERT INTO a_merge VALUES(1,'LIUWEI',80)(2,'ZHANGBIN',30)(3,'
   FUGUO',20)(4,'YIHE',33)(5,'TIANTIAN',23);
INSERT INTO b_merge VALUES(1,2,'ZHANGBIN',70,'吉林')(2,4,'YIHE++'
   ,33,'黑龙江')(3,3,'FUGUO',44,'山东')(4,1,'LIUWEI++',80,'江西')
   (5,5,'TIANTIAN',23,'河南')(6,6,'NINGQIN',23,'江西')(7,7,'BING'
   ,24,'吉安');

MERGE INTO a_merge a USING (SELECT b.aid,b.name,b.year,b.city
   FROM b_merge b) c
ON(a_merge.id=c.aid)
WHEN MATCHED THEN
UPDATE SET a.name=c.name
WHEN NOT MATCHED THEN
INSERT(a.id,a.name,a.year) VALUES(c.aid,c.name,c.year);

SELECT * FROM a_merge;

ID | NAME | YEAR |

-----------------------------------------------------

1| LIUWEI++| 80|
2| ZHANGBIN| 30|
3| FUGUO| 20|
4| YIHE++| 33|
5| TIANTIAN| 23|
6| NINGQIN| 23|
7| BING| 24|
```

6.9 数据复制

6.9.1 INSERT INTO SELECT

将一个查询的结果插入到一个表中。

语法格式

```
insert_select_stmt::=
INSERT INTO targ_table[targ_col_list] selectstmt
```

参数说明

- targ_table：目标表。
- targ_col_list：目标列，多个列之间用逗号分隔。
- selectstmt：查询语句，详情参考查询章节。

> **说明**
>
> targ_col_list 字段数必须和 selectstmt 中的字段数一致。

示例

```
SQL> CREATE TABLE copy_tab1(col1 INT,col2 VARCHAR);

SQL> INSERT INTO copy_tab1 VALUES(1,'VALUE1')(2,'VALUE2');

SQL> CREATE TABLE copy_tab2(col1 INT,col2 VARCHAR);

SQL> CREATE TABLE copy_tab3(col1 INT,col2 VARCHAR);

-- 通过INSERT INTO ..SELECT 复制表copy_tab1数据到copy_tab2;
SQL> INSERT INTO copy_tab2 SELECT * FROM copy_tab1;

-- 通过INSERT INTO ..SELECT 复制表copy_tab1数据到copy_tab3;
SQL> INSERT INTO copy_tab3(col1,col2) SELECT * FROM copy_tab1;

SQL> SELECT * FROM copy_tab2;

COL1 | COL2 |
-------------------------------------------------------------
1 | VALUE1|
2 | VALUE2|

SQL> SELECT * FROM copy_tab3;

COL1 | COL2 |
-------------------------------------------------------------
1 | VALUE1|
2 | VALUE2|
```

6.9.2 CREATE TABLE SELECT

创建一个新表，并通过 SELECT 语句的结果填充该表。

语法格式

```
createasstmt::=
CREATE [TEMPORARY | TEMP | LOCAL TEMPORARY | LOCAL TEMP | GLOBAL
   TEMPORARY | GLOBAL TEMP] TABLE [IF NOT EXISTS] name_space [(
   ColId [,ColId]... )] AS selectstmt
```

参数说明

- [TEMPORARY | TEMP | LOCAL TEMPORARY | LOCAL TEMP | GLOBAL TEMPORARY | GLOBAL TEMP]：指定创建的表是临时表还是全局临时表。
- TEMPORARY、TEMP ：创建临时表，这些表在会话结束时会被自动删除。

- LOCAL TEMPORARY、LOCAL TEMP：表的作用范围仅限于当前会话。
- GLOBAL TEMPORARY、GLOBAL TEMP：创建全局临时表。
- name_space：表名，可带模式名也可忽略模式名。
- (ColId [,ColId]...)：列名，可选值，缺省时列名为查询中的列名。selectstmt：查询语句，详情参考查询章节。

> **说明**
>
> ColId 字段数必须和 selectstmt 中的字段数保持一致。

示例

```
SQL> CREATE TABLE copy_tab4(col1 INT,col2 VARCHAR);

SQL> INSERT INTO copy_tab4 VALUES(1,'VALUE1')(2,'VALUE2');

SQL> CREATE TABLE copy_tab5 AS SELECT * FROM copy_tab4;

SQL> SELECT * FROM copy_tab5;

COL1 | COL2 |

-----------------------------------------------------

1 | VALUE1|
2 | VALUE2|
```

6.9.3 IMPORT...SELECT

将数据从一个查询结果导入到现有的表中。

语法格式

```
ImportStmt::=
IMPORT [APPEND | REPLACE] TABLE name_space [( columnList )] FROM
   selectstmt
```

参数说明

| REPLACE ：可选。

  - APPEND: 默认值，表示将 selectstmt 的结果追加到目标表中，而不影响已有数据。
  - REPLACE: 表示在导入新数据之前，先删除目标表中的所有现有数据。

- name_space：表名，可带模式名也可忽略模式名。

- columnList：列名，可选值，缺省时列名为查询中的列名。selectstmt：查询语句，详情参考查询章节。

> 说明
>
> - columnList 字段数必须和 selectstmt 中的字段数保持一致。
> - IMPORT 语法复制数据不支持目标表出现在查询子句中。

示例

- 示例 1 APPEND 模式

```
SQL> CREATE TABLE tb_ap(id INT,name VARCHAR(20));

SQL> INSERT INTO tb_ap VALUES(1,'one')(2,'two');

SQL> CREATE TABLE tb_ap1(id INT,name VARCHAR(20));

SQL> INSERT INTO tb_ap1 VALUES(66,'abc');

SQL> IMPORT APPEND TABLE tb_ap1 FROM SELECT * FROM tb_ap;

SQL> SELECT * FROM tb_ap1;

ID | NAME |

------------------------------------------------------

66 | abc|
1 | one|
2 | two|
```

- 示例 2 REPLACE 模式

```
SQL> IMPORT REPLACE TABLE tb_ap1 FROM SELECT * FROM tb_ap;

SQL> SELECT * FROM tb_ap1;

ID | NAME |
--
   ------------------------------------------------------------------

1 | one|
2 | two|
```

6.10 修改备注信息

支持 SQL 语句方式修改对象备注信息，增加对象备注信息可补充说明对应对象作用，方便后续操作。

语法格式

```
CommentStmt::=
COMMENT ON obj_type name_space IS Sconst

obj_type::=
DATABASE
|    SCHEMA
|    TABLE
|    COLUMN
|    SEQUENCE
|    INDEX
|    VIEW
|    PROCEDURE
|    PACKAGE
|    TRIGGER
|    TABLESPACE
|    DATABASE LINK
|    REPLICATION
|    SNAPSHOT
|    SYNONYM
|    USER
|    ROLE
|    JOB
|    DIR
```

参数说明

- name_space：对象名根据需要修改的对象进行设置。
- Sconst：修改的备注信息串。
- obj_type：增加或修改备注信息的对象。

示例

修改表字段备注信息。

```
SQL> CREATE TABLE tb_com(id INT COMMENT 'id信息')COMMENT '备注信息
   用例';

SQL> SELECT comments FROM user_tables WHERE table_name='TB_COM';

COMMENTS |

-----------------------------------------------------------

备注信息用例|
```

```
SQL> SELECT comments FROM user_columns WHERE table_id=(SELECT
   table_id FROM user_tables WHERE table_name='TB_COM');

COMMENTS |

-----------------------------------------------------

id信 息|

COMMENT ON COLUMN tb_com.id IS '新 修 改 的 备 注';

SQL> SELECT comments FROM user_columns WHERE table_id=(SELECT
   table_id FROM user_tables WHERE table_name='TB_COM');

COMMENTS |

-----------------------------------------------------

新 修 改 的 备 注|
```

> **说明**
>
> 部分版本的控制台屏蔽关键字 COMMENT 导致报错，使用最新版本控制台或管理器等工具即可。

6.11 重新开表

开表是一种更新内存中表的元信息的操作，当表上的索引失效等情况下，需要重新开表。

语法格式

```
ALTER TABLE table_name REOPEN;
```

参数说明

table_name：表名。

示例

对表 person_info 重新开表。

```
ALTER TABLE person_info REOPEN;
```

6.12 自增列

在数据库中，自增列用于自动生成唯一的数值，通常作为表的主键。

虚谷数据库通过“INDENTITY(B,S)”语法和“AUTO_INCREMENT”语法实现自增列。控制自增列的插入值填充模式，可通过系统参数 def_identity_mode 或会话级参数

IDENTITY_MODE 进行设置。

自增列是一组序列值，可以通过系统表 sys_columns 获取对应自增列的序列值发生器的 ID 号，即“SERIAL_ID”，再通过 SERIAL_ID 在系统表 sys_sequences 中查看对应自增列的详细信息。

6.12.1 定义自增列

语法格式

```
CREATE TABLE table_name (col_name INT {IDENTITY IDENTITY[(B,S)]|
   AUTO_INCREMENT} [col_elements]);
```

参数说明

- table_name：表名。
- col_name：列名。
- IDENTITY[(B,S)]：设置自增列，B 为初始值，S 为步长，均为 INTEGER 类型，(B,S) 省略时默认为 (1,1)。
- AUTO_INCREMENT：设置自增列，效果同 IDENTITY(1,1)。
- [col_elements]：可选其他列定义操作，如设置主键约束、值检查约束、外键约束，添加列注释等，详细信息请参见创建表章节。

6.12.2 修改自增列

语法格式

```
ALTER TABLE table_name {ALTER|MODIFY} [COLUMN] col_name INT
   IDENTITY[(B,S)] [alter_specification];
```

参数说明

- table_name：表名。
- {ALTER|MODIFY}：修改列关键字，两者效果相同。
- [COLUMN]：列关键字，可省略。
- col_name：列名。
- IDENTITY[(B,S)]：设置自增列，B 为初始值，S 为步长，均为 INTEGER 类型，(B,S) 省略时默认为 (1,1)。

- [alter_specification]：可选其他列修改操作，修改约束、修改注释等，详细信息请参见修改表章节。

6.12.3 示例

创建一个自增列初始值为 0，步长为-1 的表，并插入值。

```
SQL> CREATE TABLE t1 (id INT IDENTITY(0,-1) PRIMARY KEY, name
   VARCHAR(50));

SQL> INSERT INTO t1 (name) VALUES ('小白'),('小何');

SQL> SELECT * FROM t1;

ID | NAME |
-----------------------------------------------------------
0 | 小白|
-1 | 小何|
```

修改自增列的初始值和步长，再次插入值。

```
SQL> ALTER TABLE t1 ALTER id INT IDENTITY(1,5);

SQL> INSERT INTO t1 (name) VALUES ('小张'),('小王');

SQL> SELECT * FROM t1;

ID | NAME |
-----------------------------------------------------------
0 | 小白|
-1 | 小何|
1 | 小张|
6 | 小王|
```

去掉自增列属性，再次插入值。

```
SQL> ALTER TABLE t1 ALTER id INT;

SQL> INSERT INTO t1 (name) VALUES ('小李'),('小赵');

SQL> SELECT * FROM t1;

ID | NAME |
-----------------------------------------------------------
0 | 小白|
-1 | 小何|
1 | 小张|
6 | 小王|
<NULL>| 小李|
<NULL>| 小赵|
```

创建一个具有自增列的表，通过连接查询查看自增列的“CURR_VAL”当前值和“STEP_VAL”增长步长。

```
SQL> CREATE TABLE t2 (id INT IDENTITY(5,2));

SQL> SELECT
        CURR_VAL, STEP_VAL
    FROM
        sys_sequences seq
    INNER JOIN
        sys_columns col
    ON
        seq.SEQ_ID = col.SERIAL_ID
    INNER JOIN
        sys_tables tab
    ON
        col.TABLE_ID = tab.TABLE_ID
    WHERE
        tab.TABLE_NAME = 't2';

CURR_VAL | STEP_VAL |
-------------------------------------------------------------
5 | 2 |
```

7 回收站管理

7.1 概述

回收站是所有被删除表以及表的相关依赖对象的一个逻辑存储容器。

回收站提供了一种安全的数据恢复机制，当表对象被意外删除时，可以通过回收站的表对象进行查找，并单独的将此对象恢复到原始状态，从而无需通过恢复数据库或从备份中进行恢复等复杂操作，这会使得数据库的管理和维护更加方便简单。

7.2 查看是否开启回收站

回收站可通过此方式查看是否开启。

语法格式

```
show enable_recycle;
```

示例

```
SQL> show enable_recycle;

ENABLE_RECYCLE |
---------------------------------------------------
T |
Use time:0 ms.
```

7.3 开启回收站

回收站可以通过此方法开启（立即生效），也可以在启动数据库时将 xugu.ini 中的 enable_recycle 参数设置为 true（默认为 true）。

语法格式

```
set enable_recycle on;
```

示例

```
SQL> set enable_recycle on;

Execute successful.Use time:3 ms.

SQL> show enable_recycle;
```

```
ENABLE_RECYCLE |
-----------------------------------------------------
T |
Use time:1 ms.
```

7.4 关闭回收站

回收站可以通过此方式关闭（立即生效）或者在启动数据库时将 xugu.ini 中的 enable_recycle 参数设置为 false（默认为 true）。

语法格式

```
set enable_recycle off;
```

示例

```
SQL> set enable_recycle off;

Execute successful.Use time:3 ms.

SQL> show enable_recycle;

ENABLE_RECYCLE |
-----------------------------------------------------
F |
Use time:0 ms.
```

7.5 删表进入回收站

表会被逻辑删除，将表对象，以及表相依赖的的对象放入回收站。

语法格式

```
DROP TABLE [IF EXISTS] [schema_name.]tab_name [CASCADE CONSTRAINTS
    | CASCADE | RESTRICT]
```

示例删除表进入回收站，可以将表恢复。

```
-- 创建表
SQL> CREATE TABLE index_table1(a INT,b VARCHAR,c INT,CONSTRAINT
   pk_table1 PRIMARY KEY(a));

-- 删除表
SQL> DROP TABLE index_table1;

-- 恢复表
SQL> FLASHBACK TABLE index_table1 TO BEFORE DROP;

Execute successful.
```

7.6 删表不进入回收站

表会被实际删除，以及和表相依赖的索引，约束，触发器，同义词全部删除。

语法格式

```
DROP TABLE [IF EXISTS] [schema_name.]tab_name [CASCADE CONSTRAINTS
    | CASCADE | RESTRICT]
```

示例永久删除表后进行恢复，返回错误。

```
-- 创建表
SQL> CREATE TABLE index_table1(a INT,b VARCHAR,c INT,CONSTRAINT
   pk_table1 PRIMARY KEY(a));

-- 永久删除表
SQL> DROP TABLE index_table1 PURGE;

-- 恢复表返回错误
SQL> FLASHBACK TABLE index_table1 TO BEFORE DROP;
Error: [E5021] 表或视图INDEX_TABLE1不存在
```

7.7 查询回收站

当前可通过系统表 sys_recyclebin，系统视图 DBA_RECYCLEBIN、USER_RECYCLEBIN，以及视图 USER_RECYCLEBIN 的同义词 RECYCLEBIN 这三种方式进行回收站对象查询。

回收站系统表

当前可通过系统表回收站-sys_recyclebin 进行回收站对象查询。

系统表回收站-sys_recyclebin 的详细信息请参见《系统字典参考》的回收站-sys_recyclebin 章节。

回收站系统视图

当前可通过系统视图 DBA_RECYCLEBIN、USER_RECYCLEBIN 进行回收站对象查询。

- DBA_RECYCLEBIN 的详细信息请参见《系统视图参考》的 DBA_RECYCLEBIN 章节。
- USER_RECYCLEBIN 的详细信息请参见《系统视图参考》的 USER_RECYCLEBIN 章节。

公共同义词 RECYCLEBIN

RECYCLEBIN 是 USER_RECYCLEBIN 视图的同义词，查询当前用户的所有回收对象。

```
SQL> SELECT * FROM RECYCLEBIN;
```

```
DB_ID | USER_ID | OBJECT_ID | RELATEDOBJ_ID | OBJECT_TYPE |
   OBJECT_TYPE_NAME | SCHEMA_ID | OBJECT_NAME | RECYCLE_NAME |
   DROPED_TIME | CAN_UNDROP | CAN_PURGE | RESERVED1 |
-------------------------------------------------------------
1 | 1 | 1048723 | 1048723 | 5 | Table| 1 | INDEX_TABLE1|
   BIN$0800020014| 2024-12-02 11:03:47.439 AD | YES| YES| <NULL>|
1 | 1 | 1048724 | 1048723 | 10 | Index| 1 |
   PK_IDX_S7349173313740585| BIN$0E00010010
   | 2024-12-02 11:03:47.439 AD | NO| YES| <NULL>|
```

7.8 操作回收站表对象

通过回收站表名 RECYCLE_NAME 进行表数据查询，其余 DML/DDL 操作均不支持。

示例

```
-- 创建表并插入数据
SQL> CREATE TABLE index_table1(a INT,b VARCHAR,c INT,CONSTRAINT
   pk_table1 PRIMARY KEY(a));

SQL> INSERT INTO index_table1 (a, b, c) VALUES
(2, 'Bob', 40),
(3, 'Charlie', 50),
(4, 'David', 60);

-- 删除表
SQL> DROP TABLE index_table1;

-- 查询回收站中被删除表的RECYCLE_NAME
SQL> SELECT * FROM RECYCLEBIN;

DB_ID | USER_ID | OBJECT_ID | RELATEDOBJ_ID | OBJECT_TYPE |
   OBJECT_TYPE_NAME | SCHEMA_ID | OBJECT_NAME | RECYCLE_NAME |
   DROPED_TIME | CAN_UNDROP | CAN_PURGE | RESERVED1 |
-------------------------------------------------------------
1 | 1 | 1048723 | 1048723 | 5 | Table| 1 | INDEX_TABLE1|
   BIN$0800020014| 2024-12-02 11:03:47.439 AD | YES| YES| <NULL>|
1 | 1 | 1048724 | 1048723 | 10 | Index| 1 |
   PK_IDX_S7349173313740585| BIN$0E00010010
   | 2024-12-02 11:03:47.439 AD | NO| YES| <NULL>|

-- 查询RECYCLE_NAME
SQL> SELECT*FROM "BIN$0800020014";

A | B | C |
-------------------------------------------------------------
2 | Bob| 40 |
3 | Charlie| 50 |
4 | David| 60 |
```

7.9 恢复表

指定表名或回收站表名将其恢复到原始状态。

若指定的表名在回收站中存在同名，则恢复最后进入回收站的表对象。

为了避免恢复后重名报错，也可以在恢复时进行重名名。

语法格式

```
FLASHBACK TABLE tab_name TO BEFORE DROP [ RENAME TO new_name ] ;
```

参数说明

- tab_name：指定要恢复的表的名字。这个表必须已经存在于回收站中。
- [RENAME TO new_name]：可选，重命名恢复后可能重名的表。

示例

删除表后进行恢复。

```
-- 创 建 表
SQL> CREATE TABLE index_table1(a INT,b VARCHAR,c INT,CONSTRAINT
   pk_table1 PRIMARY KEY(a));

-- 删 除 表
SQL> DROP TABLE index_table1;

-- 恢 复 表
SQL> FLASHBACK TABLE index_table1 TO BEFORE DROP;

Execute successful.
```

7.10 清理回收站

回收站是逻辑存储容器，依赖于原有表所占空间。如果不对回收站进行清除操作，回收站中的表对象会一直存在，导致空间一直被占用。

回收站提供 PURGE 命令来清除回收站对象，以及原有的 DROP 数据库/用户/模式也会清除回收站对象，从而释放空间。

包含以下五种方式：

- 表
- 索引
- 当前用户的所有对象

- 当前库下所有用户的所有对象
- 删除库/用户/模式

7.10.1 表

清理回收站时，与此表相关的索引和触发器也同时清理。若是存在同名表，则清理第一个进入回收站的表。

语法格式

```
PURGE TABLE  old_tab_name/"recycle_tab_name"
```

示例

```
SQL> PURGE TABLE INDEX_TABLE4;

Execute successful.Use time:29 ms.
```

7.10.2 索引

若是存在同名表的同名索引，则清理第一个进入回收站的索引。

> 说明
>
> 主键索引无法删除。

语法格式

```
PURGE INDEX old_tab_name.old_idx_name/"recycle_tab_name"."
   recycle_idx_name"
```

示例

```
SQL> PURGE INDEX INDEX_TABLE4.INDEX_TABLE4;

Execute successful.Use time:26 ms.
```

7.10.3 当前用户的所有对象

语法格式

```
PURGE RECYCLEBIN;
```

示例

```
SQL> PURGE RECYCLEBIN;

Execute successful.Use time:11 ms.
```

7.10.4 当前库下所有用户的所有对象

当前用户需要 DBA 权限。

语法格式

```
PURGE DBA_RECYCLEBIN;
```

示例

```
SQL> PURGE DBA_RECYCLEBIN;

Execute successful.Use time:12 ms.
```

7.10.5 删除库/用户/模式

删除库、用户和模式时，回收站中若存在该库、用户和模式的所属对象，也会从回收站中删除。

8 索引管理

8.1 创建索引

索引是数据库中重要的对象，利用索引对数据进行各种操作可以极大地提高系统性能，在数据定位方面表现非常突出。

索引可以为数据定位带来高性能、高效率，但并非表中创建的索引越多越好，因为利用索引提高查询效率是以额外占用存储空间为代价的，而且为了维护索引的有效性，当对表中数据进行操作时，数据库需要对索引维护。

索引设计可参考以下原则：

- 在常用的数据定位列上创建索引（即 WHERE 子句中出现的列）。
- 在表的主键、外键上创建索引（默认创建的有索引）。
- 在经常用于表与表之间连接的字段上创建索引。
- 查询几乎没有涉及到的列不纳入索引列。
- 针对重复度过高的列不纳入索引列或不创建索引。
- 当该表写的性能比查询的性能要求高时，应少建或者不建索引。

8.1.1 主要语法结构

语法格式

```
IndexStmt::=
    CREATE [UNIQUE] [IF NOT EXISTS] INDEX index_name ON name_space
    ( index_elem [,index_elem]... )
[index_type_opt]  [ftidx_opt]  [opt_idx_parti]  [opt_online] [
   parallel_opt2] [opt_wait] [opt_sort]
```

参数说明

- CREATE：关键字，用于创建索引。
- UNIQUE：当指定参数 UNIQUE 时，系统将自动为表增加一个对应的唯一值约束，后续若要删除该索引，需通过删除对应约束的方式进行。
- IF NOT EXISTS：当创建索引时存在同名索引则忽略此错误，该关键字无法验证已有同名索引与当前创建索引结构是否一致。

- index_name：指定要创建的索引名称。
- ON name_space (index_elem [,index_elem]...)：指定索引所在的表或视图（name_space），以及索引的列（index_elem）。
- [index_type_opt]：索引类型，通常用于指定自定义索引类型。
- [ftidx_opt]：全文索引选项，用于创建全文索引。
- [opt_idx_parti]：分区选项，用于创建分区索引。
- [opt_online]：在线/离线模式，指定索引创建是否可以在数据库在线状态下进行。
- [parallel_opt2]：并行处理选项，指定索引创建是否并行执行。
- [opt_wait]：等待选项，指定索引创建是否等待其他操作完成。
- [opt_sort]：排序选项，指定索引列的排序顺序（升序或降序）。

8.1.2 索引类型 index_type_opt

语法格式

```
index_type_opt::=
    /*EMPTY*/
|   INDEXTYPE IS ColId
```

参数说明

- /*EMPTY*/：没有指定索引类型时，默认空。
- INDEXTYPE IS ColId：指定索引类型。

8.1.3 全文索引 ftidx_opt

语法格式

```
ftidx_opt::=
    /*EMPTY*/
|   full_index_opts

full_index_opts::=
    full_index_opt_item
|   full_index_opts full_index_opt_item

full_index_opt_item::=
    USING VOCABLE TABLE Sconst
|   FOR EVERY ICONST VOCABLE
|   USING FILTER Sconst
|   USING LEXER Sconst
```

参数说明

- /*EMPTY*/：没有指定全文索引选项时，默认空。
- full_index_opt_item：指定全文索引的选项。
- FILTER：文档过滤器。
- VOCABLE：词表。
- LEXER：语法分析器。

8.1.4 分区索引 opt_idx_parti

语法格式

```
opt_idx_parti::=
    LOCAL
|   (GLOBAL [opt_partitioning_clause] [opt_subpartitioning_clause])

opt_partitioning_clause::=
    PARTITION BY RANGE '(' name_list ')' opt_parti_interval
   PARTITIONS '(' range_parti_items ')'
|   PARTITION BY LIST '(' name_list ')' PARTITIONS '('
   list_parti_items ')'
|   PARTITION BY HASH '(' name_list ')' PARTITIONS iconst
|   PARTITION BY HASH '(' name_list ')' PARTITIONS '(' name_list ')
   '

opt_subpartitioning_clause ::=
    SUBPARTITION BY HASH '(' name_list ')' SUBPARTITIONS iconst
|   SUBPARTITION BY HASH '(' name_list ')' SUBPARTITIONS '('
   name_list')'
|   SUBPARTITION BY LIST '(' name_list ')' SUBPARTITIONS '('
   list_parti_items ')'
|   SUBPARTITION BY RANGE '('name_list')' SUBPARTITIONS '('
   range_parti_items ')'
```

参数说明

- LOCAL：用于指定索引的分区模式为局部分区，即针对各个表分区各建一个子索引，局部索引仅限于分区表。
- GLOBAL：用于指定索引的分区模式为全局索引，即无论表是否分区，索引都不分区。
- opt_partitioning_clause 为一级分区子句。
- opt_subpartitioning_clause 为二级分区子句。

8.1.5 其他选项

语法格式

```
opt_online::=
    ONLINE
|   OFFLINE

parallel_opt2::=
    NOPARALLEL
|   (PARALLEL [ICONST [","  ICONST]])

opt_wait::=
    empty
|   NOWAIT
|   WAIT
|   WAIT ICONST

opt_sort::=
    ASC
|   DESC
```

参数说明

- opt_online：索引创建是否在线进行。
- parallel_opt2：索引创建是否使用并行处理。
- opt_wait：索引创建时的等待行为。
- opt_sort：索引的排序顺序。

8.1.6 示例

- 示例 1

  创建一个堆表 index_list, 在其上创建一个全局列表分区索引，分区键 city_name，索引键值根据 city_name 进行分区。

```
CREATE TABLE index_list (city_id INT ,city_name VARCHAR,
   city_population INT);

CREATE INDEX index_l ON index_list(city_name) GLOBAL PARTITION BY
    LIST (city_name) PARTITIONS
(
partition1 VALUES('成 都'),
partition2 VALUES('重 庆'),
partition3 VALUES('上 海'),
partition4 VALUES('北 京'),
partition5 VALUES('广 州'),
partition6 VALUES('台 湾'),
partition7 VALUES('香 港')
);
```

- 示例 2

为表 index_list 创建一个全局分区索引，该索引以 city_name 作为一级列表分区的分区键，以 city_population 作为二级范围分区的分区键。

```
CREATE INDEX index_list_range ON index_list(city_name) GLOBAL
   PARTITION BY LIST (city_name) PARTITIONS
(
partition_chengdu VALUES('成都'),
partition_chongqing VALUES('重庆'),
partition_shanghai VALUES('上海'),
partition_beijing VALUES('北京'),
partition_guangzhou VALUES('广州'),
partition_taiwan VALUES('台湾'),
partition_hongkong VALUES('香港')
)
SUBPARTITION BY RANGE(city_population) SUBPARTITIONS
(
partition_3000w VALUES LESS THAN (3000),
partition_2500w VALUES LESS THAN (2500),
partition_2000w VALUES LESS THAN (2000),
partition_1500w VALUES LESS THAN (1500),
partition_1000w VALUES LESS THAN (1000),
partition_500w VALUES LESS THAN (500)
);
```

- 示例 3

  创建索引时，支持排序关键字（ASC/DESC）的使用。

```
CREATE TABLW tab(id INT);

CREATE INDEX idx1 ON tab(id ASC); #执行成功
```

该示例表示为表 tab 创建一个升序索引。

> **说明**
>
> 默认采用升序索引。

- 示例 4

  创建索引时，支持关键字（IF NOT EXISTS）的使用。创建索引已存在同名索引，但要创建的索引类型与原索引不同，此处创建不会抛出错误，也不会改变原索引。

```
SQL> CREATE TABLE index_tab(id int,name varchar(20));
SQL> CREATE INDEX test_idx on index_tab(id);
SQL> SELECT di.index_name,di.index_type FROM dba_indexes di WHERE
    di.index_name='TEST_IDX';
INDEX_NAME   |INDEX_TYPE
---------------------
TEST_IDX     |0
SQL> CREATE UNIQUE INDEX IF NOT EXISTS test_idx on index_tab(id);
```

```
SQL> SELECT di.index_name,di.index_type FROM dba_indexes di WHERE
    di.index_name='TEST_IDX';
INDEX_NAME   |INDEX_TYPE
---------------------
TEST_IDX     |0
```

8.2 重建索引

重建索引是指保留索引原定义，删除索引的物理存储，并重新生成索引数据。在重建索引的过程中，系统将会对表对象加排他锁，影响业务访问，所以不应在访问频繁时执行此操作。

语法格式

```
ReindexStmt::=
    REINDEX tab_name [("." "*")|(PARTITION name_space)] [opt_force]
   [opt_fast] [opt_online] [parallel_opt2] [opt_wait]
```

参数说明

- tab_name：待进行重建索引操作的对象表名。
- name_space：索引名或符号 *。索引名表示重建指定索引；如果是采用符号 *, 则表示重建指定表下的所有索引。

示例

- 示例 1

  ```
      REINDEX test.index_1;
  ```

  该示例指示重建 test 表下的 index_1 索引。

- 示例 2

  ```
      REINDEX student.*;
  ```

  该示例表示重建 student 表下的所有索引。

8.3 删除索引

删除已创建的索引。

语法格式

```
DropIdxStmt::=
    (DROP INDEX [IF EXISTS] tab_name.name_space)
```

参数说明

- IF EXISTS：删除索引时索引不存在则忽略错误。
- table_name：待进行删除索引操作的对象表名。
- index_name：指定操作索引名。

示例

- 示例 1 删除 test 表上的索引 idx_1。

```
DROP INDEX test.idx_1;
```

- 示例 2 删除索引时支持 IF EXISTS。在添加 IF EXISTS 的情况下删除 test 表上的索引 idx_1。

```
DROP INDEX IF EXISTS test.idx_1;
```

8.4 索引添加列表分区

语法格式

```
AlterIndexStmt::=
    ALTER INDEX name_space ADD PARTITION ColId VALUES [LESS THAN]
    "(" parti_values ")"
|   ALTER INDEX name_space  SLOW MODIFY ON|OFF opt_wait
```

参数说明

- name_space：要修改的对象名，形式为表名. 索引名。
- parti_values：索引分区的条件。

示例

```
ALTER INDEX t_base_3.idx_t3_1 ADD PARTITION p3 VALUES LESS
    THAN('2020-01-01');
```

该示例表示添加新的索引分区 p3。

8.5 重命名索引

更改现有索引的名称，而不影响索引的功能或性能。

语法格式

```
AlterIndexStmt::=
    ALTER INDEX name_space RENAME TO new_name_space
```

参数说明

- name_space：要修改的对象名，形式为表名. 索引名。
- new_name_space：修改后的对象名，形式为表名. 索引名。

示例

将索引 idx_t3_1 重命名为 idx_t3_2。

```
ALTER INDEX t_base_3.idx_t3_1 RENAME TO t_base_3.idx_t3_2;
```

9 视图管理

9.1 概述

视图也称逻辑表，是建立在查询基础上的非物理存在的表，其基表可以是一个或多个物理表，视图中的数据随基表中数据变化而变化，视图一旦被定义后，在查询中其地位与物理表相当，虚谷数据库允许视图作为其它视图的基表，同时也允许针对某些视图进行记录的插入、修改与删除。

用户利用视图对数据进行操作比用户直接对表操作有更多的优势，主要体现在以下四个方面。

简化数据操作

视图可以只将有用的数据展示给用户，以便用户处理，而且不用关心数据表中的数据结构，这样就简化了数据处理的过程。因为用户只能看到视图中定义的数据，而不是基础表中的所有数据。

自定义所需的数据

有时用户对数据库中的数据需求并不是直接的，而是需要某些计算后的结果，这时利用视图来取得这样的数据是非常方便的。例如：用户想查看学生成绩表的平均成绩，就可以这样写 SELECT avg(语文), avg(数学), avg(英语) FROM 学生成绩;。

从多个表中汇总

视图可以将来自不同的两个表或者多个表（或者其它视图）中的有用数据组合成单一的结果集，这对用户来说看到的是一个单一的数据区，可以像独立的表一样进行操作，从而简化了用户对数据的处理。

通过视图可以修改数据用户在利用视图来浏览表中的数据时，可以在视图的结果集中进行数据修改。如果是建立在多表基础上的视图，则不可以对其进行修改。通过再次浏览视图可以看到数据相应的变化。

但在对视图进行插入、更改、删除时，操作成功与否会受到基表上的约束的影响，由于视图呈现的字段可能只是基表的部分字段，进行数据插入时未出现在视图中的字段的取值将被设置成空值，若这些字段中存在不允许置为空值的字段，则插入操作将失败。

9.2 创建视图

语法格式

```
ViewStmt ::=
    CREATE [OR REPLACE] VIEW [IF NOT EXISTS] view_name
    [opt_column_list]
    AS {SelectStmt | joined_table | '(' joined_table ')'
       alias_clause}
    [view_opt]
    [opt_comment]

view_opt ::=
    | WITH READ ONLY
    | WITH CHECK OPTION
```

参数说明

- OR REPLACE：指定视图存在则替换，用于更新视图定义。
- IF NOT EXISTS：创建视图时存在同名视图则忽略错误，该关键字无法验证已有同名视图与当前创建视图结构一致。
- view_name：视图名。
- opt_column_list：指定视图中各字段名，当不指定时，字段名将从 SELECT 子句中抽取，若 SELECT 子句中的输出项目不是字段而是表达式，则视图字段名将由系统生成，视图创建者可根据实际情况选择是否指定视图字段名。
- SelectStmt：SELECT 子句，定义了视图的数据来源。
- joined_table：表连接结构。
- ’(’ joined_table ’)’ alias_clause：JOIN 操作后的结果集以及指定的临时表别名。
- view_opt：视图可执行操作选项。
  - WITH READ ONLY：创建的视图是只读视图，不能进行更新、插入、删除操作。
  - WITH CHECK OPTION：可以进行插入和更新操作，但应该满足 WHERE 子句的条件。
- opt_comment：对视图进行注释。

示例

- 示例 1

  创建名为 student_view 的视图用于展示学生的基本信息，其基表为学生信息表 student，

视图中的信息是基表信息在 st_no、st_name、st_sex、st_department 字段上的投影，投影输出字段名对应变为 id、name、sex、department。

```
CREATE VIEW student_view (id, name, sex, department) AS SELECT
   st_no, st_name, st_sex, st_department FROM student;
```

- 示例 2

创建与原视图同名的视图，不会对原视图产生影响。

```
SQL> CREATE TABLE test_view_tab1(id INT, name VARCHAR);
SQL> CREATE VIEW view1 AS SELECT * FROM test_view_tab1;
SQL> SELECT * FROM VIEW1;
ID  |NAME
-----------

SQL> CREATE VIEW view1 AS SELECT id FROM test_view_tab1; -- 创建
   一个与原始图列不同的视图,此处会返回警告
SQL> SELECT * FROM VIEW1; -- 不会对原存储过程产生影响
ID  |NAME
-----------
```

- 示例 3

创建一个视图，将多个表通过内连接合并，查询最终结果。

```
-- 表1：存储用户基本信息
SQL> CREATE TABLE table1 (
id INT PRIMARY KEY,
name VARCHAR(50),
email VARCHAR(100)
);

-- 表2：存储用户的订单信息
SQL> CREATE TABLE table2 (
order_id INT PRIMARY KEY,
user_id INT,           -- 外键关联到 table1.id
product VARCHAR(50),
FOREIGN KEY (user_id) REFERENCES table1(id)
);

-- 表3：存储订单的物流信息
SQL> CREATE TABLE table3 (
tracking_id INT PRIMARY KEY,
order_id INT,
status VARCHAR(20),
FOREIGN KEY (order_id) REFERENCES table2(order_id)    -- 外键关联
   到 table2.order_id
);

-- 向 table1 插入用户数据
SQL> INSERT INTO table1 (id, name, email) VALUES(1, 'Alice', '
   alice@example.com'), (2, 'Bob', 'bob@example.com');

-- 向 table2 插入订单数据
```

```
SQL> INSERT INTO table2 (order_id, user_id, product) VALUES
   (101, 1, 'Laptop'), (102, 1, 'Phone'), (103, 2, 'Tablet');

-- 向 table3 插入物流数据
SQL> INSERT INTO table3 (tracking_id, order_id, status) VALUES
   (1001, 101, 'Shipped'), (1002, 102, 'Pending'), (1003, 103, '
   Delivered');

SQL> CREATE VIEW alice_orders_with_tracking AS
SELECT
t1.id AS user_id,
t1.name,
t1.email,
t1.order_id AS t1_order_id,  -- 来自 table2 的 order_id
table3.order_id AS table3_order_id,  -- 来自 table3 的 order_id
t1.product,
table3.tracking_id,
table3.status
FROM (
(table1 JOIN table2 ON table1.id = table2.user_id) AS t1
JOIN table3 ON t1.order_id = table3.order_id
)
WHERE t1.name = 'Alice' WITH CHECK OPTION COMMENT 'Alice的订单信
   息';

SQL> SELECT*FROM alice_orders_with_tracking;

USER_ID | NAME | EMAIL | T1_ORDER_ID | TABLE3_ORDER_ID | PRODUCT
    | TRACKING_ID | STATUS |
-----------------------------------------------------------------
1 | Alice| alice@example.com| 101 | 101 | Laptop| 1001 | Shipped|
1 | Alice| alice@example.com| 102 | 102 | Phone| 1002 | Pending|
```

9.3 删除视图

语法格式

```
DropViewStmt::=
    DROP VIEW [IF EXISTS] name_space [CASCADE | RESTRICT]
```

参数说明

- IF EXISTS：删除视图时不存在，忽略此错误。
- view_name: 视图名。
- CASCADE：如果该视图被其他对象（如其他视图或存储过程）引用，则级联删除这些依赖对象。
- RESTRICT：如果该视图被其他对象引用，则拒绝删除操作。

示例

- 示例 1

```
DROP VIEW view_test;
```

- 示例 2

```
DROP VIEW IF EXISTS view_test;
```

10 游标管理

10.1 概述

游标提供了一种灵活的手段，可以对表中检索出的数据进行操作。就本质而言，游标实际上是一种能从包括多条数据记录的结果集中每次提取一条记录的机制。它从数据表中提取出来的数据，是以临时表的形式存放在内存中，所以在大数据规模下应慎用游标，避免需缓存数据过多导致系统内存资源不足。

游标分为两种类型，分别为显式游标与隐式游标。

10.2 显式游标

由用户以变量的形式定义的游标。

10.2.1 定义游标

在块语句、存储过程、存储函数、包的变量定义部分进行游标定义。

语法格式

```
CursorDef::=
CURSOR ColId [ func_args ] { IS | AS } SelectStmt [ parallel_opt
    ] [ opt_wait ]
```

```
CursorStmt::=
DECLARE ColId [ func_args ] CURSOR FOR SelectStmt [ parallel_opt
    ] [ opt_wait ]
```

参数说明

- ColId：定义的游标名称。
- func_args：可选参数列表，若游标定义为参数游标，则打开游标时需相应的给出实参。
- SelectStmt: 一个完整的 SELECT 语句，它定义了游标将要操作的数据集，可以为单表查询、联合查询、视图查询、分组统计、排序查询等。
- parallel_opt：并行选项，指定游标是否应该并行执行。
- opt_wait：等待选项，指明当没有立即可用的资源时，游标的行为（例如等待直到资源可用，还是立即返回错误）。

10.2.2 打开游标

在定义了游标后，访问游标数据前必须进行打开游标操作，否则无法正常访问游标数据。

语法格式

```
OPEN ColId [(paramter type[,paramter type...])];
```

10.2.3 提取数据

提取游标数据用于逐行获取游标的查询返回结果，同时可将对应结果填充至变量，需要注意的是提取数据是默认每次获取一行数据，要返回多行记录需重复执行或在循环语句中执行。

语法格式

```
FETCH ColId [ opt_bulk ] [ opt_into_list ]

opt_bulk::=
BULK COLLECT
```

参数说明

- opt_bulk：用于指定是否使用批量获取（BULK COLLECT）
- opt_into_list：指定将检索到的数据存储到变量或集合中。

游标返回结果包含多列时，可采用多变量赋值或记录变量赋值的方式进行，记录变量可通过 tab_name%ROWTYPE 或 cursor_name%ROWTYPE 方式定义。

10.2.4 关闭游标

游标使用完毕后必须显式关闭游标，释放其所占资源。

语法格式

```
CLOSE ColId;
```

10.2.5 游标属性

显式游标的属性：

- %ROWCOUNT：指示 FETCH 语句返回记录行数。
- %FOUND：指示前一 FETCH 是否返回一行数据，若提取到记录该值为 TRUE，反之为 FALSE。
- %NOTFOUND：与%FOUND 含义相反，当 FETCH 提取到数据时该值为 FALSE，未提取到数据返回 TRUE。

- %ISOPEN：指示游标是否打开，显式在提取数据前必须执行打开游标操作，若已打开则该值为 TURE，否则为 FALSE。
- 可以通过游标名% 属性的方式使用上述游标属性。

示例

使用 CURSOR IS 定义了一个带有参数 inid 的游标 cur，该游标将返回 id 大于 inid 的所有记录并返回总行数。

```
-- 创建表并插入数据
SQL> CREATE TABLE test_cur(id INT,name VARCHAR(20));

SQL> INSERT INTO test_cur VALUES(1,'TEST1')(2,'TEST2')(3,'TEST3');

-- 定义并使用游标
SQL> DECLARE
CURSOR cur(inid INT) IS
SELECT * FROM test_cur WHERE id > inid;
oid   INT;
oname VARCHAR;
BEGIN
IF NOT cur%ISOPEN THEN
OPEN cur(1);
END IF;
FETCH cur INTO oid, oname;
WHILE cur%FOUND LOOP
SEND_MSG('ID IS:' || oid || ' NAME IS:' || oname);
FETCH cur INTO oid, oname;
END LOOP;
SEND_MSG('TOTAL FETCH :' || cur%ROWCOUNT || ' ROWS');
CLOSE cur;
END;
/
-- 输出
ID IS:2 NAME IS:TEST2
ID IS:3 NAME IS:TEST3
TOTAL FETCH :2 ROWS
```

使用 DECLARE CURSOR FOR 方式定义游标。

```
-- 定义游标
SQL> DECLARE cur(inid INT) CURSOR FOR SELECT * FROM test_cur WHERE
   id > inid;
/

ID | NAME |
-------------------------------------------------------------

-- 打开游标
SQL> OPEN cur(1);

-- 获取数据
SQL> FETCH cur;
```

```
ID | NAME |
-----------------------------------------------------------
2 | TEST2|
3 | TEST3|

-- 关闭游标
SQL> CLOSE cur;
```

使用 BULK COLLECT 批量返回结果。

```
SQL> DECLARE
CURSOR cur IS
SELECT * FROM test_cur;
TYPE trow IS TABLE OF cur%ROWTYPE;
otab trow;
BEGIN
OPEN cur;
FETCH cur BULK COLLECT INTO otab;
CLOSE cur;
FOR i IN 1 .. otab.COUNT LOOP
SEND_MSG(otab(i).id || '|' || otab(i).name);
END LOOP;
END;
/
1|TEST1
2|TEST2
3|TEST3
```

10.3 隐式游标

在执行块语句或存储过程/函数的过程中，DML 语句会使用隐式游标，即：

- 插入操作：INSERT
- 更新操作：UPDATE
- 删除操作：DELETE
- 单行查询操作：SELECT ..... INTO ....

当系统使用隐式游标时，可通过游标属性获取操作状态与结果，需注意的是游标属性只针对前一个 DML 操作或单行查询语句有效，所以若想正确获得游标属性需在执行 DML 或单行查询语句后立即使用。

游标属性包括：

- SQL%FOUND：返回一个布尔值，以指示隐式游标执行是否成功，当 DML 操作或单行查询语句执行成功该值为 TRUE，失败为 FALSE。

> **⚠ 注意**
>
> 隐式游标语句执行前该属性值为 NULL，而非 FALSE。

- SQL%NOTFOUND：与 SQL%FOUND 相反，当 SQL%FOUND 为 TRUE 时，SQL%NOTFOUND 为 FALSE；当 SQL%FOUND 为 FALSE 时，SQL%NOTFOUND 为 TRUE。同理，在隐式游标执行前该值为 NULL。
- SQL%ROWCOUNT：前一 DML 语句或单行查询语句执行影响行数。

示例

更新 test_cur 表中 id 大于 1 的所有记录，将 id 增加 1 并输出受影响的行数。

SQL%ROWCOUNT 返回上一条 DML 语句影响的行数。

```
-- 创建表并插入数据
SQL> CREATE TABLE test_cur(id INT,name VARCHAR(20));

SQL> INSERT INTO test_cur VALUES(1,'TEST1')(2,'TEST2')(3,'TEST3');

SQL> DECLARE
BEGIN
UPDATE test_cur SET id = id + 1 WHERE id > 1;
SEND_MSG('TOTAL UPDATE:' || SQL%ROWCOUNT || ' ROWS');
ROLLBACK;
END;
/
-- 输出
TOTAL UPDATE:2 ROWS
```

10.4 系统引用游标

引用游标是一种对结果集的引用。

相比静态的普通游标在编译时定义，系统引用游标可以动态打开，并在运行时定义。

系统引用游标只能用于块语句、存储过程、存储函数中。

语法格式

- 系统引用游标声明。

```
ColId SYS_REFCURSOR;
```

- 系统引用游标打开。

```
OPEN ColId FOR SelectStmt;
```

- 系统引用游标关闭。

```
CLOSE ColId;
```

使用说明

- ColId：游标变量的名称。
- SelectStmt：数据库查询语句，决定了系统引用游标所引用的结果集。

示例

假设表 tab_test 已存在并有 4 条数据。

```
DECLARE
  i INT;
  v_chr VARCHAR;
  -- (1) 系统引用游标声明
  cur_sysref SYS_REFCURSOR;
BEGIN
  -- (2) 系统引用游标打开
  OPEN cur_sysref FOR SELECT c2 FROM tab_test;
  i := 1;
  LOOP
      FETCH cur_sysref INTO v_chr;
      EXIT WHEN cur_sysref%NOTFOUND;
      SEND_MSG('Row ' || i || ' is: ' || v_chr);
      i := i + 1;
  END LOOP;
  -- (3) 系统引用游标关闭
  CLOSE cur_sysref;
END;

--------------------
Row 1 is: a
Row 2 is: b
Row 3 is: c
Row 4 is: d
```

10.5 批量提取游标数据

游标提取数据的方式除了逐行提取外，还可批量提取。

语法格式

```
FETCH cursor_name BULK COLLECT INTO table_name;
```

示例

```
DECLARE
  CURSOR cur IS
    SELECT * FROM test_cur;
  TYPE trow IS TABLE OF cur%ROWTYPE;
  otab trow;
```

```
BEGIN
  OPEN cur;
  FETCH cur BULK COLLECT INTO otab;
  CLOSE cur;
  FOR i IN 1 .. otab.COUNT LOOP
    SEND_MSG(otab(i).id || '|' || otab(i).name);
  END LOOP;
END;
```

11 存储过程/函数管理

11.1 存储过程

11.1.1 概述

存储过程是为了执行一定的任务而组合在一起的 SQL 语句集，通过存储过程封装，用户与数据库间只需少量的信息通讯即可完成多语句操作，减少了用户与数据库系统间的交互，可提升一定访问性能。存储过程的实现在减少用户端工作的同时，会相应的增加数据库端的工作，在一定程度上增加数据库服务器的负荷，故是否使用存储过程，如何使用存储过程需经过数据库管理员评估。

通常存储过程包含以下三个部分：

1. 声明部分
2. 执行部分
3. 异常处理部分

执行部分为必选项，声明部分和异常处理部分可根据情况而定。

11.1.2 创建存储过程

11.1.2.1 主要语法结构

语法格式

```
CreProcedureStmt::=
    {CREATE | CREATE OR REPLACE | CREATE OR REPLACE FORCE} ProcDef

ProcDef::=
    ProcDecl [COMMENT SCONST] {IS | AS} LANGUAGE {PLSQL | C} NAME
       ColId
    |   ProcDecl [COMMENT SCONST] {IS | AS} [DECLARE] VarDefList
       StmtBlock [name_space]

ProcDecl::=
    PROCEDURE [IF NOT EXISTS] name_space func_args [AUTHID {DEFAULT
        | DEFINER | CURRENT_USER | USER}]
```

参数说明

- 存储过程创建语句：
  - CREATE OR REPLACE：替换已存在的同名存储过程（若存在），否则新建。

- CREATE OR REPLACE FORCE：强制创建存储过程，忽略非语法/编译错误（如依赖对象无效），生成警告且创建的存储过程无效。

- 过程定义：
  - COMMENT SCONST：添加注释，COMMENT ' 注释'。
  - LANGUAGE PLSQL | C NAME ColId：引用的内部接口。

DECLARE VarDefList StmtBlock [name_space]：

  - VarDefList：变量定义语句。包括变量定义、游标定义、异常定义等。详细信息请参见 PL/SQL 语言 > PL/SQL 语法章节。
  - StmtBlock：BEGIN...END 块语句执行体，若执行体包含 DDL 语句，则需用动态 SQL 方式进行，如 EXECUTE IMMEDIATE('$SQL');。详细信息请参见 PL/SQL 语言 > PL/SQL 语法章节。

- 过程声明：
  - IF NOT EXISTS：创建存储过程时存在同名存储过程则忽略此错误，该关键字无法判断已有同名存储过程与当前创建存储过程属性是否一致。
  - name_space：定义的存储过程名称。
  - func_args：形式参数信息，注意不要使用系统关键字作为形式参数名。
  - AUTHID DEFAULT：默认权限。
  - AUTHID DEFINER：使用定义者的权限执行。
  - AUTHID CURRENT_USER | AUTHID USER：使用调用者的权限执行。

[2.0] **说明**

当 def_compatible_mode 设置为 Oracle 模式时，在创建存储过程/函数会默认强制创建，忽略掉除语法、编译等错误之外的错误，相当于虚谷数据库的 CREATE OR REPLACE FORCE 方式，创建成功但是存在错误的存储过程会是一个无效的存储过程。

11.1.2.2 形式参数 func_args

语法格式

```
func_args::=
(FuncArg [,FuncArg]...)
|    ()
```

```
FuncArg::=
ColLabel [IN | OUT | IN OUT] TypeName [DEFAULT b_expr]
```

参数说明

- ColLabel：参数名称。
- IN：表示该参数为输入型参数，即调用时采用值传递，不能携带返回值给调用者。若未明确指示参数属性，则默认该参数为输入型参数。
- OUT：表示该参数可用于携带返回值给调用者。在过程执行中的主要任务是被赋值。当过程结束时，形参会被赋给实参，返回给调用者。
- IN OUT：综合 IN 和 OUT。它可以将实参传递进过程。在过程的内部，形参可以被读取也可以被写入。在过程结束时，形参的内容同时也被赋给实参。
- TypeName：对应参数的数据类型，详细信息请参见 PL/SQL 语言 > PL/SQL 语法章节。
- [DEFAULT b_expr]：可选的默认值表达式。

11.1.2.3 示例

- 示例 1

  无参数存储过程。创建无参存储过程，其封装了一个循环执行体与一条消息输出命令。循环体循环 10 次向 test_proc_tab 中插入数据，完成循环执行后输出循环次数消息。

```
CREATE TABLE test_proc_tab(id INT,dtime TIME);

CREATE OR REPLACE PROCEDURE proc_test() IS
loop_num INT;
BEGIN
loop_num := 0;
FOR i IN 1 .. 10 LOOP
INSERT INTO test_proc_tab (id, dtime) VALUES (i, current_time);
loop_num := loop_num + 1;
END LOOP;
SEND_MSG('过程执行完成' || loop_num || '次');
COMMIT;
END;

EXEC proc_test();

-- 输出
过程执行完成10次

SELECT COUNT(*) FROM test_proc_tab;

EXPR1 |
```

```
-----------------------------------------------------
10 |
```

- 示例 2

  带输入型参数的存储过程。创建一个带输入参数的存储过程 proc_test2，通过输入的参数 parameter 控制向表 test_proc_tab 中插入数据条数，执行插入操作后输出插入次数信息。

```
CREATE OR REPLACE PROCEDURE proc_test2(parameter INTEGER) AS
x int;
BEGIN
x := 0;
FOR i IN 1 .. parameter LOOP
INSERT INTO test_proc_tab VALUES (i, sysdate);
x := x + SQL%ROWCOUNT;
END LOOP;
SEND_MSG('共插入：' || x || '次');
COMMIT;
END;

EXEC proc_test2(3);

-- 输出
共插入：3次

SELECT COUNT(*) FROM test_proc_tab;

EXPR1 |
-----------------------------------------------------
13 |
```

- 示例 3

  带输出型参数的存储过程。创建一个带输出参数的存储过程 proc_test3，循环 10 次向 test_proc_tab 表中插入数据，并将循环次数赋值给输出参数；块语句执行存储过程 proc_test3，并打印输出参数信息。

```
CREATE OR REPLACE PROCEDURE proc_test3(parameter OUT INTEGER) AS
x int;
BEGIN
x := 0;
FOR i IN 1 .. 10 LOOP
INSERT INTO test_proc_tab VALUES (i, sysdate);
x := x + SQL%ROWCOUNT;
END LOOP;
parameter := x;
END;

DECLARE
OUTRET INT;
BEGIN
EXEC proc_test3(OUTRET);
SEND_MSG('共插入：' || OUTRET || '次');
```

```
END;

-- 输出
共插入：10次

SELECT COUNT(*) FROM test_proc_tab;

EXPR1 |
------------------------------------------------------
23 |
```

- 示例 4

  创建存储过程支持 IF NOT EXISTS 关键字。创建一个与已有存储过程同名但属性不同的存储过程，不会改变原存储过程。

```
CREATE PROCEDURE proc_pre1 AS
BEGIN
send_msg(123);
END;

EXEC pro_pre1;
123
SQL> CREATE PROCEDURE IF NOT EXISTS proc_pre1 AS
BEGIN
send_msg(1234566);
END; /*创建一个与原存储过程内容不同的存储过程，此处会返回警告*/

EXEC pro_pre1; /*不会对原存储过程产生影响*/
123
```

- 示例 5

  创建存储过程使用 FORCE。强制创建一个有错误的存储过程，插入时 col3 不存在，若无 FORCE 创建则会报错。此处示例使用 FORCE 强制创建，不会报错 Error 但会返回警告 Warning，并且创建的存储过程无效。

```
-- 创建表tb_var
SQL> CREATE TABLE tb_var(var1 VARCHAR(30),var2 CHAR(3));

-- 强制创建存储过程proc_pre2
SQL> CREATE OR REPLACE FORCE PROCEDURE proc_pre2(col1 VARCHAR,
   col2 VARCHAR) IS
BEGIN
INSERT INTO tb_var VALUES(col1,col3);
END;
/
Execute successful.
Use time:3 ms.

-- 返回警告col3不存在
Warning: [E19212] 代码编译错误
[E19182 L3 C33] 编译SQL失败字段变量或函数"COL3"不存在
```

```
-- 查询创建的存储过程proc_pre2
SQL> SELECT proc_name,valid FROM DBA_PROCEDURES;

PROC_NAME | VALID |
-----------------------------------------------------
PROC_PRE2| F |

-- 调用存储过程proc_pre2，返回错误存储过程无效
SQL> EXEC proc_pre2;
Error: [E8014] 存储过程或函数PROC_PRE2被标识为失效
```

11.1.3 删除存储过程

语法格式

```
DropProcedureStmt::=
DROP PROCEDURE [IF EXISTS] name_space alter_behavior

alter_behavior::=
[CASCADE | RESTRICT]
```

参数说明

- IF EXISTS：删除时存储过程或函数不存在，忽略此错误。
- alter_behavior：若该参数不提供则默认使用 RESTRICT，在删除存储过程或函数时会检测存储过程或存储函数是否被其他对象依赖，若存在依赖则删除失败，若无依赖则删除成功；CASCADE 则表示强制删除存储过程或存储函数，强制删除被依赖对象后，数据库还会对依赖对象的 valid 状态进行对应处理。

示例由于存储过程 pro2 依赖于 pro1，因此在尝试删除 pro1 时，数据库阻止了这一操作以保护依赖关系。使用 CASCADE 强制删除。

```
-- 创建存储过程pro1
SQL> CREATE OR REPLACE PROCEDURE pro1 IS
BEGIN
DBMS_OUTPUT.PUT_LINE('abc');
END;
/

-- 创建存储过程pro2
SQL> CREATE PROCEDURE pro2 IS
BEGIN
EXEC pro1;
END;
/

-- 默认方式删除pro1返回错误
SQL> DROP PROCEDURE pro1;
Error: [E9002] 存在对存储过程或函数PRO1依赖的对象
```

```
SQL> SELECT valid FROM user_procedures WHERE proc_name='PRO2';

VALID |
------------------------------------------------------------
T |

-- 强制删除pro1
SQL> DROP PROCEDURE pro1 CASCADE;

SQL> SELECT valid FROM user_procedures WHERE proc_name='PRO2';

VALID |
------------------------------------------------------------
F |
```

11.1.4 重编译失效存储过程

语法格式

```
RecompileStmt::=
ALTER PROCEDURE name_space RECOMPILE
```

示例

1. 创建存储过程 pro1 和 pro2，pro2 依赖于 pro1。

```
-- 创建存储过程pro1
SQL> CREATE OR REPLACE PROCEDURE pro1 IS
BEGIN
DBMS_OUTPUT.PUT_LINE('abc');
END;
/

-- 创建存储过程pro2
SQL> CREATE PROCEDURE pro2 IS
BEGIN
EXEC pro1;
END;
/
```

2. 强制删除 pro1，导致 pro2 失效。

```
-- 强制删除存储过程pro1
SQL> DROP PROCEDURE pro1 CASCADE;

-- 存储过程pro2已失效
SQL> SELECT valid FROM user_procedures WHERE proc_name='PRO2';

VALID |
------------------------------------------------------------
F |
```

3. 重新创建 pro1，并对 pro2 进行重编译，功能恢复。

```
-- 重新创建存储过程pro1
SQL> CREATE OR REPLACE PROCEDURE pro1 IS
BEGIN
DBMS_OUTPUT.PUT_LINE('abc');
END;
/

-- 此时pro2依然失效
SQL> SELECT valid FROM user_procedures WHERE proc_name='PRO2';

VALID |
------------------------------------------------------
F |

-- 重编译pro2
SQL> ALTER PROCEDURE pro2 RECOMPILE;

-- 存储过程pro2已恢复
SQL> SELECT valid FROM user_procedures WHERE proc_name='PRO2';

VALID |
------------------------------------------------------
T |
```

11.2 存储函数

11.2.1 概述

存储函数与存储过程相类似，均是为了提高 sql 语句的重用性而对一组 SQL 语句进行的封装，存储过程无返回值而存储函数可根据用户需求自定义返回值。

存储函数的返回值类型一般为数据库系统定义的类型，但也可以为特殊数据类型，如：游标、记录、表等。若返回类型为表时，该存储函数为表值函数，在访问时与普通存储函数不同，需采用 SELECT TABLE(function_name) 的方式。

11.2.2 创建存储函数

语法格式

```
CreProcedureStmt::=
    {CREATE | CREATE OR REPLACE | CREATE OR REPLACE FORCE} ProcDef

ProcDef::=
    ProcDecl [COMMENT SCONST] {IS | AS} LANGUAGE {PLSQL | C} NAME
      ColId
|   ProcDecl [COMMENT SCONST] {IS | AS} [DECLARE] VarDefList
   StmtBlock [name_space]

ProcDecl::=
```

```
    FUNCTION [IF NOT EXISTS] name_space func_args RETURN TypeName [
       PIPELINED] [AUTHID {DEFAULT | DEFINER | CURRENT_USER | USER}]
|   FUNCTION [IF NOT EXISTS] name_space func_args RETURN SELF AS
   RESULT [AUTHID {DEFAULT | DEFINER | CURRENT_USER | USER}]
```

参数说明

- 存储函数创建语句：
  - CREATE OR REPLACE：替换已存在的同名存储函数（若存在），否则新建。
  - CREATE OR REPLACE FORCE：强制创建存储函数，忽略非语法/编译错误（如依赖对象无效），生成警告且创建的存储函数无效。
- 函数定义：
  - COMMENT SCONST：添加注释，COMMENT ’ 注释’。
  - LANGUAGE PLSQL | C NAME ColId：引用的内部接口。

DECLARE VarDefList StmtBlock [name_space]：

- VarDefList：变量定义语句。包括变量定义、游标定义、异常定义等。详细信息请参见 PL/SQL 语言 > PL/SQL 语法章节。
- StmtBlock：BEGIN...END 块语句执行体，若执行体包含 DDL 语句，则需用动态 SQL 方式进行，如 EXECUTE IMMEDIATE(’$SQL’);。详细信息请参见 PL/SQL 语言 > PL/SQL 语法章节。

- 函数声明：
  - IF NOT EXISTS：创建存储函数时存在同名存储函数则忽略此错误，该关键字无法判断已有同名存储函数与当前创建存储函数属性是否一致。
  - name_space：自定义函数名。
  - func_args：形式参数名。
  - TypeName：参数的类型名或返回值类型。
  - SELF AS RESULT：返回对象自身实例。
  - AUTHID DEFAULT：默认权限。
  - AUTHID DEFINER：使用定义者的权限执行。
  - AUTHID CURRENT_USER | AUTHID USER：使用调用者的权限执行。

> **⚠ 注意**
>
> 使用 PIPELINED 函数以外的存储函数时需给出返回值。

使用 PIPELINED 关键字指定的函数即成为 PIPELINED 函数，此类函数遵循以下规则：

- 返回结果必须为自定义 TABLE 与 VARRAY 类型。
- 返回结果支持嵌套 RECORD 类型，但不支持嵌套包内自定义类型。
- 函数可以不包含 RETURN 子句，若包含 RETURN 子句，RETURN 子句不能返回任何表达式。
- 函数只支持函数表的用法，不支持出现语句的其他位置。
- 使用 PIPELINED 函数作为函数表的查询语句不支持字段筛选、不支持字段使用其他函数或其他表达式、不支持条件过滤等。
- PIPELINED 函数目前只支持配合 PIPE ROW 子句提供实时数据返回功能。
- 使用 PIPE ROW 语句用于快速实时返回结果，并且 PIPE ROW 语句只能出现在 PIPELINED 函数中。PIPE ROW 返回结果必须为函数返回值中的元素类型。

普通函数示例

- 示例 1

  创建存储函数获取表 test_proc_tab 中 id 字段的最大值，并返回该值；执行存储函数与执行存储过程不同点在于：执行存储过程使用关键字 EXEC，执行存储函数使用 SELECT。

```
CREATE OR REPLACE FUNCTION func_test()
RETURN INT
AS
ret INT;
BEGIN
SELECT MAX(id) INTO ret FROM test_proc_tab;
RETURN ret;
END;

SELECT func_test() FROM dual;

EXPR1 |

------------------------------------------------------

10 |
```

- 示例 2

创建一个表值函数，返回类型为集合类型，与记录类型相似可以返回多行记录。

```
CREATE OR REPLACE FUNCTION func_tab(num INT) RETURN TABLE OF
   RECORD(id INT, name VARCHAR(10)) IS
DECLARE  subtype rec IS RECORD(id INT, name CHAR(10));
tab TABLE OF rec;
BEGIN
FOR i IN 1 .. num LOOP
tab.EXTEND;
tab(tab.last) := rec(i, 'name' || i);
END FOR;
RETURN tab;
END;

SELECT * FROM TABLE(func_tab(3));

ID | NAME |

-----------------------------------------------------

1 | name1|
2 | name2|
3 | name3|
```

PIPELINED 函数示例

- 自定义函数中使用 PIPELINED 函数。

```
-- 创建table
CREATE OR REPLACE TYPE tab AS TABLE OF VARCHAR(100);

-- 创建函数，pipe row内返回的varchar为tab集合的元素类型
CREATE OR REPLACE FUNCTION fun(a INT)
RETURN tab PIPELINED
IS
j VARCHAR:='test';
BEGIN
FOR i IN 1..10 LOOP
PIPE ROW('abc'||j);
j:= j||(a+i);
END LOOP;
PIPE ROW('pipe row end');
RETURN;
END;

-- 使用函数
SELECT * FROM TABLE(fun(1));
COLUMN_VALUE     |
-----------------+
abctest          |
abctest2         |
abctest23        |
abctest234       |
abctest2345      |
abctest23456     |
abctest234567    |
```

```
abctest2345678   |
abctest23456789  |
abctest2345678910|
pipe row end     |
```

错误使用示例：

- 查询使用 PIPELINED 函数当作目标字段，返回错误 E19067。

```
SQL> SELECT fun(1);
Error: [E19067] 系 统 不 支 持 的 操 作
```

- RETURN 语句包含表达式，返回错误 E19212，E10133 L6 C5。

```
SQL> CREATE OR REPLACE FUNCTION fun()
RETURN tab PIPELINED
IS
BEGIN
PIPE ROW('PIPE ROW END');
RETURN 123;
END;
/

Error: [E19212] 代 码 编 译 错 误
[E10133 L6 C5] 管 道 函 数 中 的 RETURN 语 句 不 能 包 含 表 达 式
```

- PIPE ROW 子句在非 PIPELINED 函数中使用，返回错误 E19212，E10134 L5 C5。

```
SQL> CREATE OR REPLACE FUNCTION fun()
RETURN tab
IS
BEGIN
PIPE ROW('PIPE ROW END');
END;
/

Error: [E19212] 代 码 编 译 错 误
[E10134 L5 C5] PIPE 语 句 只 能 在 管 道 函 数 中 使 用
```

- 返回类型不为有效的集合类型，返回错误 E10135。

```
SQL> CREATE OR REPLACE FUNCTION fun()
RETURN VARCHAR PIPELINED
IS
BEGIN
PIPE ROW('PIPE ROW END');
END;
/

Error: [E10135] 管 道 函 数 必 须 具 有 支 持 的 集 合 返 回 类 型
```

- 包定义 PIPELINED 函数。

```
-- 创 建 包 头
```

```
CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS
TYPE num_tab IS TABLE OF NUMBER;
FUNCTION pkg_fun(x NUMBER) RETURN num_tab PIPELINED;
END;

-- 创 建 包 体
CREATE OR REPLACE PACKAGE BODY pkg AS
FUNCTION pkg_fun(x NUMBER) RETURN num_tab PIPELINED IS
BEGIN
PIPE ROW(x);
PIPE ROW(x + 100);
RETURN;
END;
END;

-- 使 用 包 函 数
SELECT * FROM TABLE(pkg.pkg_fun(103));
COLUMN_VALUE|
------------+
103|
203|
```

返回游标结果 RECORD 类型。

```
-- 创 建 表
CREATE TABLE t_tab(id int, c1 VARCHAR,c2 VARCHAR,c3 VARCHAR,c4
   VARCHAR);

-- 插 入 数 据
INSERT INTO t_tab VALUES(1,'1','a','26','z')(2,'2','b','25','y')
   (3,'3','c','24','x')(4,'4','d','23','w');

-- 创 建 包 头
CREATE OR REPLACE PACKAGE pkg IS
TYPE var1 IS t_tab%ROWTYPE;
TYPE var2 IS TABLE OF var1;
FUNCTION pkg_func() RETURN var2 PIPELINED;
END;

-- 创 建 包 体
CREATE OR REPLACE PACKAGE BODY pkg IS
FUNCTION pkg_func() RETURN var2 PIPELINED
IS
TYPE curtype IS REF CURSOR;
c2 curtype;
out_rec t_tab%ROWTYPE;
BEGIN
OPEN c2 FOR SELECT * FROM t_tab;
LOOP
FETCH c2 INTO out_rec;
EXIT WHEN c2%NOTFOUND;
PIPE ROW(out_rec);
END LOOP;
CLOSE c2;
RETURN;
END;
```

```
END;

-- 使用包函数
SELECT * FROM TABLE(pkg.pkg_func());
ID|C1|C2|C3|C4|
--+--+--+--+--+
1|1 |a |26|z |
2|2 |b |25|y |
3|3 |c |24|x |
4|4 |d |23|w |
```

错误使用示例：

- 返回集合类型嵌套了包内定义类型（RECORD 除外），返回错误 E10135。

```
SQL> CREATE OR REPLACE PACKAGE pkg IS
TYPE var1 IS VARRAY(10) OF VARCHAR;
TYPE var2 IS VARRAY(10) OF var1;
FUNCTION pkg_func() RETURN var2 PIPELINED;
END;
/

Error: [E10135] 管道函数必须具有支持的集合返回类型
```

- 声明为 PIPELINED 函数、定义没有指定 PIPELINED 关键字，返回错误 E8012。

```
-- 创建包头
SQL> CREATE OR REPLACE PACKAGE pkg IS
TYPE var1 IS VARRAY(10) OF VARCHAR;
FUNCTION pkg_func() RETURN var1 PIPELINED;
END;
/
-- 创建包体
SQL> CREATE OR REPLACE PACKAGE BODY pkg is
FUNCTION pkg_func() RETURN var1
IS
BEGIN
RETURN var1(1);
END;
END;
/

-- 使用函数
SQL> SELECT * FROM TABLE(pkg.pkg_func());

Error: [E8012] 包PKG成员函数PKG_FUNC不可用
```

- UDT 中使用 PIPELINED 函数。

```
-- 创建类型var
CREATE OR REPLACE TYPE var AS VARRAY(10) OF VARCHAR(10);

-- 创建类型va2
CREATE OR REPLACE TYPE var2 AS VARRAY(10) OF var;
```

```
-- 创建类型obj
CREATE OR REPLACE TYPE obj AS OBJECT(
a INT,
MEMBER FUNCTION obj_func() RETURN var2 PIPELINED
);

-- 创建类型obj body
CREATE OR REPLACE TYPE BODY obj AS
MEMBER FUNCTION obj_func() RETURN var2 PIPELINED
IS
x var2 := var2();
BEGIN
x.extend(8);
x(1):=var(1);
x(5):=var(5);
FOR i IN x.FIRST..x.LAST LOOP
PIPE ROW(x(i));
END LOOP;
RETURN;
END;
END;

-- 使用函数
SELECT * FROM TABLE(obj(1).obj_func());
COLUMN_VALUE|
------------+
[1]         |
<NULL>      |
<NULL>      |
<NULL>      |
[5]         |
<NULL>      |
<NULL>      |
<NULL>      |
```

错误使用示例：

- 声明为 PIPELINED 函数、定义没有指定 PIPELINED 关键字，返回错误 E9015 L2 C5。

```
    -- 创建类型obj函数声明使用pipelined关键字
    CREATE OR REPLACE TYPE obj AS OBJECT(
    a INT,
    MEMBER FUNCTION obj_func() RETURN var2 PIPELINED
    );

    -- 创建类型obj body 函数声明不使用使用pipelined关键字
    CREATE OR REPLACE TYPE BODY obj AS
    MEMBER FUNCTION obj_func() RETURN var2
    IS
    x var2 := var2();
    BEGIN
    x.EXTEND(8);
    x(1):=var(1);
    x(5):=var(5);
```

```
    FOR i IN x.FIRST..x.LAST LOOP
    PIPE ROW(x(i));
    END LOOP;
    RETURN;
    END;
    END;

    Error: [E9015 L2 C5] 成员函数OBJ_FUNC未声明
```

11.2.3 删除存储函数

语法格式

```
DropProcedureStmt::=
DROP FUNCTION [IF EXISTS] name_space alter_behavior

alter_behavior::=
[CASCADE | RESTRICT]
```

参数说明

- IF EXISTS：删除时存储过程或函数不存在，忽略此错误。
- alter_behavior：若该参数不提供则默认使用 RESTRICT，在删除存储过程或函数时会检测存储过程或存储函数是否被其他对象依赖，若存在依赖则删除失败，若无依赖则删除成功；CASCADE 则表示强制删除存储过程或存储函数，强制删除被依赖对象后，数据库还会对依赖对象的 valid 状态进行对应处理。

示例

```
-- 创建函数
SQL> CREATE OR REPLACE FUNCTION func_tab(num INT) RETURN TABLE OF
   RECORD(id INT, name VARCHAR(10)) IS
DECLARE  subtype rec IS RECORD(id INT, name CHAR(10));
tab TABLE OF rec;
BEGIN
FOR i IN 1 .. num LOOP
tab.EXTEND;
tab(tab.last) := rec(i, 'name' || i);
END FOR;
RETURN tab;
END;
/

-- 使用函数
SQL> SELECT * FROM TABLE(func_tab(3));

ID | NAME |
-------------------------------------------------------------
1 | name1|
2 | name2|
3 | name3|
```

```
-- 删除函数
SQL> DROP FUNCTION func_tab;

-- 已删除，再次执行返回错误
SQL> SELECT * FROM TABLE(func_tab(3));
Error: [E10049 L1 C21] 字段变量或函数"FUNC_TAB"(3)不存在
```

11.2.4 重编译失效存储函数

语法格式

```
RecompileStmt::=
ALTER FUNCTION name_space RECOMPILE
```

示例

1. 创建表并插入数据，再创建存储函数并使用。

```
SQL> CREATE TABLE test_proc_tab (id NUMBER);

SQL> INSERT INTO test_proc_tab (id) VALUES (1)(2)(3);

SQL> CREATE OR REPLACE FUNCTION  func_test()
RETURN INT
AS
ret INT;
BEGIN
SELECT MAX(id) INTO ret FROM test_proc_tab;
RETURN ret;
END;
/

SQL> SELECT func_test() FROM dual;

EXPR1 |
------------------------------------------------------------
3 |
```

2. 强制删除表，存储函数失去依赖表失效。

```
SQL> DROP TABLE test_proc_tab CASCADE;

SQL> SELECT func_test() FROM dual;
Error: [E8014 L1 C8] 存储过程或函数FUNC_TEST被标识为失效
```

3. 重新创建表并插入数据，重编译存储函数，可正常使用。

```
SQL> CREATE TABLE test_proc_tab (id NUMBER);

SQL> INSERT INTO test_proc_tab (id) VALUES (1)(2)(3);

SQL> ALTER FUNCTION func_test RECOMPILE;

SQL> SELECT func_test() FROM dual;
```

```
EXPR1 |
------------------------------------------------------------
3 |
```

12 程序包管理

12.1 概述

在大型应用系统中，一个模块可能涉及到多个存储过程、存储函数、游标等对象，为满足模块化管理，相应的可通过定义包对象管理程序模块所需的数据库对象。一个包对象结构包括：包规范 (包头) 与包体。

12.2 创建程序包

一个包对象结构包括：包规范 (包头) 与包体。

包头定义了包对象所有的公有元素信息，可包括游标、数据类型、存储过程/函数等元素。

包体是对包头定义的具体实现，在包体内定义公有存储过程或存储函数的执行体。同时包体内还可以声明包的私有元素，声明的游标、变量、用户类型等在包头内未定义，则这些对象为私有对象，用户不可通过包进行访问，但可在包体内进行调用。

12.2.1 包头定义

语法格式

```
packdecl::=
packdecl::=
CREATE [FORCE | OR REPLACE | OR REPLACE FORCE] PACKAGE name_space
[AUTHID {DEFAULT | DEFINER | CURRENT_USER | USER}]
[COMMENT SCONST]
{IS | AS} PackSpecItem [PackSpecItem]...
END [name_space]

PackSpecItem::=
VarDef ;
|   ProcDecl ;
```

参数说明

- 创建选项：
  - FORCE：强制创建包，即使存在依赖对象无效。
  - OR REPLACE：替换现有包（若存在）。
  - OR REPLACE FORCE：组合使用，允许替换并强制创建。

- 包名称（name_space）：
  - 格式：[schema_name.]package_name（可能包含模式名）。
- 权限控制（AUTHID）：
  - DEFAULT：默认权限。
  - DEFINER：使用定义者的权限执行。
  - CURRENT_USER | USER：使用调用者的权限执行。
- 注释（COMMENT）：
  - 可选，后接字符串字面量（如’This is a package’）。
- 包体声明：
  - IS 或 AS 关键字引导包规范体。
  - 包含一个或多个 PackSpecItem（变量或过程声明），每个项以分号; 结尾。
- 结束标记：
  - END 关键字后可选跟包名（需与开头的 name_space 一致）。
- 变量定义（VarDef）：
  - 示例：var_name datatype [NOT NULL] [:= default_value];
  - 需符合变量声明语法规则（如类型、约束、默认值）。
- 过程声明（ProcDecl）：
  - 示例：PROCEDURE proc_name (param_list) [RETURN datatype];
  - 包含过程名、参数列表、返回类型（可选）。

说明

在包头定义中，针对存储过程与函数的定义可通过重载的方式实现不同输入参数的同名过程/函数声明。

12.2.2 包体创建

语法格式

```
PackBody::=
    CREATE [FORCE | OR REPLACE | OR REPLACE FORCE] PACKAGE BODY
       name_space
    {IS | AS}
    [DECLARE VarDefList]
```

```
    PackMemberProcs
    [BEGIN pl_stmt_list]
    EXCEPTION Exception_Item [Exception_Item]...
    END [name_space]

VarDefList::=
    VarDef ;
|   VarDefList VarDef ;

PackMemberProcs::=
    ProcDef ;
|   PackMemberProcs ProcDef ;
```

参数说明

- 创建选项：
  - FORCE：强制创建包体，忽略依赖对象状态。
  - OR REPLACE：替换现有包体（若存在）。
  - OR REPLACE FORCE：组合使用（需按固定顺序）。
- 声明部分（DECLARE VarDefList）：
  - DECLARE 关键字可选（标准 PL/SQL 包体中通常省略，变量直接定义在 IS|AS 后）。
- 变量定义列表（VarDefList）：
  - 每个 VarDef 为变量声明（如 counter NUMBER := 0;）。
  - 分号; 分隔多个变量定义。
- 成员过程（PackMemberProcs）：
  - 包含一个或多个过程/函数定义（ProcDef），每个以分号; 结尾。
- 初始化块（BEGIN pl_stmt_list）：
  - 可选，用于执行包初始化逻辑（如赋值、调用过程）。
  - pl_stmt_list 为 PL/SQL 语句列表（如 DBMS_OUTPUT.PUT_LINE('Initialized');）。
- 异常处理（EXCEPTION）：
  - 可选，捕获包级异常。
  - Exception_Item 格式：WHEN exception_name THEN stmt_list。
- 结束标记：
  - END 后可选跟包名（需与开头的 name_space 一致）。

- 变量定义列表（VarDefList）：
  - 每个 VarDef 为变量声明（如 counter NUMBER := 0;）。
  - 分号; 分隔多个变量定义。

> **说明**
>
> 在包体内声明与定义的对象在包头内若无声明，则对象为私有元素对象，这类元素对象只能在包体内调用，用户无法通过包进行访问使用。

12.2.3 示例

创建一个封装了一个存储过程的 pack_test 包，存储过程实现了输出 USER_TABLES 系统表中表名的功能。调用该包可使用命令 exec pack_test.proc1。

```
-- 创建包头：
CREATE OR REPLACE PACKAGE pack_test IS
  PROCEDURE proc1();
END;
/

-- 创建包体：
CREATE OR REPLACE PACKAGE body pack_test IS
  PROCEDURE proc1() IS
    CURSOR mycur IS
      SELECT table_name FROM user_tables;
    con  INTEGER;
    name VARCHAR;
  BEGIN
    SELECT count(*) INTO con FROM user_tables;
    OPEN mycur;
    FOR i IN 1 .. con LOOP
      FETCH mycur
        INTO name;
      send_msg(name);
    END LOOP;
    close mycur;
  END;
END;
/
```

12.3 包的使用

语法格式

调用包规范中声明的对象可以使用点标记。

```
pack_use::=
    package_name.subprogram_name
```

包对象内的成员对象访问与普通对象访问类似，使用包仅仅起到封装作用。同时经过包封装，应用程序在调用包时即编译包对象，并将对象信息加载至缓存，后续调用无需再次编译以提高使用性能。

参数说明

- package_name：包的名称，即包含子程序的包。必须与创建包时定义的名称完全匹配。
- subprogram_name：包内子程序的名称，可以是过程（Procedure）或函数（Function）。

示例

对包的创建章节的示例进行使用，输出 USER_TABLES 系统表中表名。

```
BEGIN
    -- 调用 pack_test 包中的 proc1 过程
    pack_test.proc1;
EXCEPTION
    WHEN OTHERS THEN
    -- 处理可能发生的异常
    DBMS_OUTPUT.PUT_LINE('Error calling pack_test.proc1: ' ||
       SQLERRM);
END;
/

-- 输出
TEST_PERMISSION
TEST_COL_PRE
OBJ_TAB
SPATIAL_REF_SYS
VARRY_TAB
USERS
TEST_PROC_TAB
```

也可以直接使用 EXEC 命令，等价于 BEGIN ... END; 块的一种简化形式。

```
EXEC pack_test.proc1;
```

12.4 重编译失效包

语法格式

```
RecompileStmt::=
ALTER PACKAGE name_space RECOMPILE
```

13 触发器管理

13.1 概述

触发器是一种特殊类型的存储过程，它在插入、删除或修改特定表中数据时起作用。触发器通过维持不同表中逻辑上相关的数据的一致性，保持数据的相关完整性。它的执行不是由程序调用，也不是手工启动，而是由事件来触发，比如当对一个表进行操作（INSERT、DELETE、UPDATE）时就会激活执行它。触发器经常用于加强数据的完整性约束和业务规则等。

> **说明**
>
> 目前仅支持 DML 触发器，暂未支持 DDL 触发器、登录触发器。

13.2 创建触发器

触发器的基础对象可分为表或视图，基于表的触发器支持 AFTER 与 BEFORE 触发时机，基于视图的触发器支持 INSTEAD OF 触发时机。

> **说明**
>
> 支持同一库下触发器名称与现有对象名重复。触发器属于被动调用的对象，不需要通过名称的唯一性去区分库中的对象，所以创建触发器的时候不需要检测当前库中已存在的对象名。

13.2.1 主要语法结构

语法格式

- 表或视图触发器

```
CreateTrigStmt::=
CREATE [OR REPLACE] TRIGGER [IF NOT EXISTS] name_space
TriggerActionTime TriggerEvents ON name_space
[OptTrggParamAlias]
[FOR {EACH ROW | STATEMENT}]
[WHEN bool_expr]
[COMMENT SCONST]
[DECLARE VarDefList]
StmtBlock
[name_space]
```

- 表或视图触发器

```
CreateTrigStmt::=
CREATE [OR REPLACE] TRIGGER name_space
AFTER LOGON ON DATABASE
[COMMENT SCONST]
[DECLARE VarDefList]
StmtBlock
[name_space]
```

> ⚠ 注意
>
> 当过程体含变量定义时，变量定义应在 BEGIN 关键字前通过 DECLARE 关键词显式定义；若无变量定义，则直接从 BEGIN 开始。

参数解释

- IF NOT EXISTS：创建触发器时若同名触发器存在则忽略错误。该关键字无法验证已有同名触发器与当前创建触发器属性是否一致。
- name_space：此处为触发器名或触发器的基表名称或视图名称，使用地方不同含义不同。
- TriggerActionTime：触发时机，有 BEFORE、AFTER、INSTEAD OF 三种情况，分别为被监视的动作之前、被监视的动作之后、替代受监视的动作，其中 INSTEAD OF 用于基于视图的触发器使用。
- TriggerOneEvent：受监视的操作类型，有 INSERT、UPDATE、DELETE 几种类型，INSERT 表示触发器将在表的插入操作时被触发，其余类推，其中 UPDATE 可以精确监视在某个字段上。
- OptTrggParamAlias：重命名 NEW 或 OLD 关键字。
- FOR EACH ROW：指定触发器为行级触发器，即每操作一行记录则触发器点火一次。
- FOR STATEMENT：指定触发器为语句级触发器，即每条操作命令触发点火一次。
- WHEN: 指定触发器约束。指定触发事件时必须满足的条件。
- DECLARE VarDefList：变量定义语句。包括变量定义、游标定义、异常定义等，详细信息请参见 PL/SQL 语言 > PL/SQL 语法 > DECLARE 声明部分。
- StmtBlock：触发器的过程体，详细信息请参见 PL/SQL 语言 > PL/SQL 语法 > BEGIN...END 块部分。

13.2.2 触发时机 TriggerActionTime

语法格式

```
TriggerActionTime::=
BEFORE
|    AFTER
|    INSTEAD OF
```

参数解释

- BEFORE：触发器在指定事件发生之前执行。
- AFTER：触发器用于在事件发生后执行后续操作，如日志记录或更新其他表。
- INSTEAD OF：触发器用于替代事件的默认行为，主要用于视图上。

13.2.3 触发事件 TriggerEvents

语法格式

```
TriggerEvents::=
TriggerOneEvent [OR TriggerOneEvent]...

TriggerOneEvent::=
INSERT
|    DELETE
|    UPDATE
|    UPDATE OF name_list
|    UPDATE OF ( name_list )
```

参数解释

- OF name_list：可选项目，为表的列名，只对触发事件为 UPDATE 的触发器有效，当给定列名后，只有在修改动作变更指定列时，触发器才被点火。

13.2.4 触发器别名 OptTrggParamAlias

语法格式

```
OptTrggParamAlias::=
REFERENCING NEW AS ColId
|    REFERENCING OLD AS ColId
|    REFERENCING NEW AS ColId OLD AS ColId
|    REFERENCING OLD AS ColId NEW AS ColId
```

参数解释

- REFERENCING：为触发器中的 OLD 和 NEW 行定义别名。

13.2.5 示例

- 示例 1

  表 test_trig_tab 上创建名为 trig_test 的触发器，触发器用于监视表上的插入操作，当插入的记录的 ID 不小于 10 时，将新记录同时插入到表 test_trig_tab2 中。

```
CREATE TABLE TEST_TRIG_TAB(ID INT,NAME VARCHAR(20));
CREATE TABLE TEST_TRIG_TAB2(ID INT,NAME VARCHAR(20));

CREATE OR REPLACE TRIGGER trig_test
AFTER INSERT ON test_trig_tab
FOR EACH ROW
WHEN (ID >= 10)
BEGIN
INSERT INTO test_trig_tab2 VALUES (new.id, new.name);
END;
/

INSERT INTO TEST_TRIG_TAB VALUES(1, 'test_trig1');
INSERT INTO TEST_TRIG_TAB VALUES(10, 'test_trig10');
```

- 示例 2

  表 test_trig_tab 上创建名为 trig_test2 的触发器，该触发器用于监视表的 ID 字段上的更改操作，当被更改的记录原 ID 值大于等于 10 时，向 test_trig_tab2 中插入一条记录，记录 ID 为 test_trig_tab 修改记录的旧值，记录 NAME 为字符串'trig_update'。

```
CREATE OR REPLACE TRIGGER trig_test2
AFTER UPDATE OF ID ON test_trig_tab
FOR EACH ROW
WHEN (old.ID >= 10)
DECLARE
num INTEGER;
BEGIN
INSERT INTO test_trig_tab2 VALUES (new.id, 'trig_update');
END;
```

- 示例 3

  创建一个使用 referencing 的触发器，referencing new AS nn 表示把 new 关键字重命名为 nn。当对表 test_trig_tab 插入记录的 ID 大于 1000 或者小于 10 时，自动往表 test_trig_tab2 插入一条记载消息记录。

```
CREATE OR REPLACE TRIGGER trig_test3
BEFORE INSERT ON test_trig_tab
REFERENCING new AS nn
FOR each ROW
BEGIN
IF nn.id > 1000 THEN
INSERT INTO test_trig_tab2 VALUES (nn.id,
```

```
'插入了一个过大的数！');
ELSE
IF nn.id < 10 THEN
INSERT INTO test_trig_tab2 VALUES (nn.id,
'插入了一个过小的数！');
END IF;
END IF;
END;
```

- 示例 4

  创建触发器支持 IF NOT EXISTS。创建一个与原触发器同名但依赖表不同的触发器，不会影响原触发器。

```
SQL> CREATE TABLE test_tab_trig1(a int);
SQL> CREATE TRIGGER TRIG1
BEFORE INSERT
ON test_tab_trig1
FOR EACH ROW
BEGIN
INSERT INTO test_tab_trig1 VALUES(99);
END;
SQL> SELECT dt.table_name,dr.trig_name FROM dba_triggers dr JOIN
   dba_tables dt ON dr.db_id=dt.db_id AND dr.OBJ_id=dt.table_id
   WHERE dt.table_name='TEST_TAB_TRIG1';
TABLE_NAME |TRIG_NAME
-----------------------
TEST_TAB_TRIG1 |TRIG1

SQL> CREATE TABLE test_tab_trig2(a int);
SQL> CREATE TRIGGER IF NOT EXISTS TRIG1 /*创建一个与原触发器依赖
   表不同的触发器，此处会返回警告*/
BEFORE INSERT
ON test_tab_trig2
FOR STATEMENT
BEGIN
INSERT INTO test_tab_trig2 VALUES(99);
END;
SQL> SELECT dt.table_name,dr.trig_name FROM dba_triggers dr JOIN
   dba_tables dt ON dr.db_id=dt.db_id AND dr.OBJ_id=dt.table_id
   WHERE dt.table_name='TEST_TAB_TRIG2';/*不会对原触发器产生影响
   */
TABLE_NAME |TRIG_NAME
-----------------------
```

13.2.6 条件谓词

如果触发器的触发事件设置超过一种类型的 DML 语句，例如触发事件包括：INSERT、DELETE 和 UPDATE，那么触发体就可以使用条件谓词 INSERTING、DELETING 和 UPDATING 进行代码逻辑处理。

若触发器触发事件包括：

```
INSERT OR DELETE OR UPDATE OF column_name ON table_name
```

则在触发器处理代码中可使用以下条件进行相关代码逻辑处理：

```
IF inserting THEN  .... END IF;
IF updating THEN .... END IF;
IF deleting THEN .... END IF;
```

有如下示例：

```
CREATE OR REPLACE TRIGGER trig_test4
  AFTER INSERT OR UPDATE OF id, name ON test_trig_tab
  FOR each ROW
BEGIN
  IF updating THEN
    INSERT INTO test_trig_tab2 VALUES (new.id, 'update tab');
  END IF;
  IF inserting THEN
    INSERT INTO test_trig_tab2 VALUES (new.id, 'insert tab');
  END IF;
END;
```

⚠ 注意

条件谓词触发器使用时，IF 判断不能嵌套判断，各条件谓词间需并列处理。

13.3 修改触发器状态

系统支持启用与禁用触发器。

语法格式

```
TrigEnableStmt::=
    ALTER TRIGGER trigger_name {ENABLE | DISABLE}
```

参数解释

- trigger_name：待操作触发器名称。
- ENABLE|DISABLE：指示触发器的触发功能生效或者失效。

示例关闭触发器。

```
ALTER TRIGGER trig_test4 DISABLE ;
```

13.4 删除触发器

语法格式

```
DropTrigStmt::=
DROP TRIGGER [IF EXISTS] trigger_name
```

参数解释

IF EXISTS：删除触发器时触发器不存在，忽略此错误。

示例

删除触发器。

```
DROP TRIGGER trig_test4;
```

14 序列值管理

14.1 创建序列值

序列值主要用于产生整数序列值，其产生整数值按照用户指定规则进行递增或递减。

14.1.1 主要语法结构

语法格式

```
CreateSeqStmt::=
CREATE SEQUENCE [IF NOT EXISTS] sequence_name [OptSeqList] [COMMENT
    SCONST]
```

参数说明

- IF NOT EXISTS：创建序列值时若同名序列值存在则忽略错误。该关键字无法验证已有同名序列值与当前创建序列值属性是否一致。
- sequence_name：序列值名称。
- OptSeqList：可选的序列值表达式，用来指定序列值的最小值、最大值、起始值、增长步长等约束条件。
- COMMENT SCONST：可选的序列值注释信息。

14.1.2 序列值表达式 OptSeqList

语法格式

```
OptSeqList::=
[OptSeqElem]...

OptSeqElem::=
START WITH var_value
|     [CACHE var_value | NOCACHE]
|    [CYCLE | NOCYCLE]
|    INCREMENT BY var_value
|    [MAXVALUE var_value | NOMAXVALUE ]
|    [MINVALUE var_value | NOMINVALUE ]
```

参数说明

- START WITH var_value：var_value 为序列的初始值；如果序列递增, 未指定初始值，则初始值为最小值；如果序列递减, 未指定初始值，则初始值为最大值。

- [CACHE var_value | NOCACHE]：value 是缓存序列值个数，默认 1；若系统重启，则序列值起始值将变为 cache 后的下一个值。

  集群环境下，缓存在当前工作节点 1，若在工作节点 2 使用序列值，则序列值起始站为节点 1 cache 后的下一个值，其他工作节点使用起始值依此类推。

  举例：（起始值 1，cache 20，节点 1：1-20，节点 2：21-40，节点 3：41-60.....; 集群重启后序列值起始值从 61 开始）。

- [CYCLE | NOCYCLE]：值达到最大值后是否循环，如果不循环，达到最大值后，继续产生新值会发生错误；默认不循环。

- INCREMENT BY var_value：var_value 为序列步长（序列增加的幅度），默认为 1，如果是负则按此步长递减。

- [MAXVALUE var_value | NOMAXVALUE]：定义序列生成器能产生的最大值，如果序列递减, 未指定最大值，默认为-1；NOMAXVALUE 无最大值限制（默认为 9223372036854775807）。

- [MINVALUE var_value | NOMINVALUE]：定义序列生成器能产生的最小值，如果序列递增, 未指定最小值，默认为 1；NOMINVALUE 无最小值限制（默认为-9223372036854775808）。

14.1.3 示例

- 示例 1

  创建一个名为 seq_test 的序列值，该序列值最小值为 1，最大值为 1000，起始值为 100，步长为 1。

```
CREATE SEQUENCE seq_test MINVALUE 1  MAXVALUE 1000  START WITH
    100  INCREMENT BY 1;
```

- 示例 2

  创建序列时支持 NOMINVALUE、NOMAXVALUE 两种选项。

  - NOMINVALUE：没有最小值定义，对于递减序列，数据库能够产生的最小值是 BIGINT 的最小值；对于递增序列，最小值是 1。
  - NOMAXVALUE：没有最大值定义，对于递增序列，数据库能够产生的最大值是 BIGINT 的最大值；对于递减序列，最大值是-1。

```
--(1)NOMINVALUE、MAXVALUE xxxxx
```

```
CREATE SEQUENCE seq1 START WITH 1 NOMINVALUE MAXVALUE 1000
   NOCYCLE CACHE 500 ORDER;

--(2)NOMAXVALUE、MINVALUE xxxxx
CREATE SEQUENCE seq2 START WITH 10 MINVALUE 1 NOMAXVALUE NOCYCLE
   CACHE 500 ORDER;

--(3)NOMINVALUE、NOMAXVALUE
CREATE SEQUENCE seq3 START WITH 10  NOMINVALUE NOMAXVALUE NOCYCLE
    CACHE 500 ORDER;

--(4)MINVALUE xxxxx、MAXVALUE xxxxx
CREATE SEQUENCE seq4 START WITH 10 MINVALUE 1 MAXVALUE 1000
   NOCYCLE CACHE 500 ORDER;
```

- 示例 3

  创建一个与原序列值名称相同但最大值属性不同的序列值，不会影响原序列值。

```
SQL> CREATE SEQUENCE IF NOT EXISTS seq_1
MINVALUE 100
MAXVALUE 200
START WITH 199
INCREMENT BY -100
CYCLE;
SQL> SELECT ds.seq_name,ds.max_val FROM dba_sequences ds WHERE ds
   .seq_name='SEQ_1';
SEQ_NAME   |MAX_VAL
---------------------
SEQ_1      |200
SQL> CREATE SEQUENCE IF NOT EXISTS seq_1 /*创建一个与原序列值
   MAXVALUE不同的序列值，此处会返回警告*/
MINVALUE 100
MAXVALUE 500
START WITH 199
INCREMENT BY -100
CYCLE;
SQL> SELECT ds.seq_name,ds.max_val FROM dba_sequences ds WHERE ds
   .seq_name='SEQ_1';/*不会对原序列值产生影响*/
SEQ_NAME   |MAX_VAL
---------------------
SEQ_1      |200
```

14.2 修改序列值

序列值修改语法中的各参数含义与创建一致，用户在使用过程中可根据需求进行调整。

语法格式

```
AlterSeqStmt::=
    ALTER SEQUENCE sequence_name OptSeqList
```

示例

```
-- 创建序列值
CREATE SEQUENCE seq_test MINVALUE 1 MAXVALUE 1000 START WITH 100
   INCREMENT BY 1;

-- 修改序列值
ALTER SEQUENCE seq_test MINVALUE 1 MAXVALUE 1000 INCREMENT BY 5
   CACHE 20 CYCLE ORDER;
```

14.3 使用序列值

序列值通过标识访问使用。

```
SELECT seq_test.nextval FROM DUAL;
```

14.3.1 NEXTVAL

获取序列的下一个值。

具体实例如下：

```
-- 创建序列值
SQL> CREATE SEQUENCE seq_test2 MINVALUE 1 MAXVALUE 1000 START WITH
    100 INCREMENT BY 1;

-- 获取序列的下一个值
SQL> SELECT seq_test2.NEXTVAL FROM DUAL;

EXPR1 |
------------------------------------------------------------
100 |
```

14.3.2 CURRVAL

获取序列的当前值。

CURRVAL 返回的是当前会话中最近一次通过 NEXTVAL 获取的序列值。如果在调用 NEXTVAL 之前尝试使用 CURRVAL 则会报错，因为此时 CURRVAL 还没有可用的值。具体实例如下：

```
-- 创建序列值
SQL> CREATE SEQUENCE seq1_test MINVALUE 1 MAXVALUE 1000 START WITH
    100 INCREMENT BY 1;

-- 第一次尝试使用CURRVAL
SQL> SELECT CURRVAL('seq1_test') FROM DUAL;
Error: [E7019 L1 C8] 当前会话中序列值seq1_test.CURRVAL不可用

-- 调用NEXTVAL
SQL> SELECT seq1_test.NEXTVAL FROM DUAL;
```

```
EXPR1 |
-------------------------------------------------------------
100 |

-- 再次尝试使用CURRVAL
SQL> SELECT CURRVAL('seq1_test') FROM DUAL;

EXPR1 |
-------------------------------------------------------------
100 |
```

> ⚠ 注意
>
> 若指定序列值在当前用户会话中还未获取过序列值，则此函数将抛出错误（如示例中所示）。CURRVAL 的生命周期：仅存在于会话中，且每个会话之间独立。

14.4 删除序列值

删除数据库中的序列对象。

语法格式

```
DropSeqStmt::=
DROP SEQUENCE [IF EIXSTS] name_space [CASCADE | RESTRICT]
```

参数说明

IF EXISTS：删除序列值时序列值不存在则忽略此错误。

示例

```
-- 先创建一个序列值
CREATE SEQUENCE seq_test MINVALUE 1 MAXVALUE 1000 START WITH 100
   INCREMENT BY 1;

-- 删除
DROP SEQUENCE seq_test;
```

15 同义词管理

15.1 概述

在数据库系统中，有一种对象叫同义词，它是对数据库对象的映射，相当于一个别名，经常用于简化对象访问和提高数据库对象安全性。与视图类似，同义词不占用实际存储空间，只是在数据字典中保存同义词定义。系统中，可针对表、视图、存储过程、函数、包等对象定义同义词。

虚谷数据库系统中同义词可分为公共同义词与私有同义词，公共同义词顾名思义为数据库用户均可使用的同义词，私有同义词为对象属主或拥有权限访问的用户使用。

15.2 创建同义词

创建同义词权限分为以下三种：

- CREATE PUBLIC SYNONYM：具有创建全局同义词的权限。（仅限库级权限）
- CREATE SYNONYM：可以在当前模式创建私有同义词。（仅限库级权限）
- CREATE ANY SYNONYM：可以在所有模式下创建私有同义词。（仅限库级，模式级权限）

> ⚠ 注意
>
> 创建同义词前请确认目标对象是否有读取权，若没有读取权，则报权限不足错误。

语法格式

```
CreateSynonymStmt::=
    CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema_syn_name.]
       synonym_name FOR [schema_obj_name.]object_name ;
```

参数说明

- OR REPLACE：指定同义词存在则替换，用于更新同义词定义。
- PUBLIC：指定同义词属性，使用参数表示同义词为公共同义词。
- schema_syn_name：指定包含同义词的模式。如果省略 schema，那么将在您自己的

schema 中创建同义词。如果指定了 PUBLIC，则不能为同义词指定模式会报错。

- synonym_name：同义词名称。
- schema_obj_name：指定目标对象的模式，如果省略 schema，那么将为您自己的 schema 中的目标对象创建同义词。
- object_name：同义词映射对象名称。

示例

- 示例 1

  表 synonym_table_student 创建一个同义词 student。当用户查询 synonym_table_student 表时就可以这样使用：SELECT * FROM student;。

```
CREATE SYNONYM student FOR synonym_table_student;
```

- 示例 2

  表 sysdba.mytablemy_db_link 带模式名后过长，创建同义词 my_r_tab 后即可通过同义词访问：SELECT * FROM my_r_tab;。

```
CREATE SYNONYM my_r_tab FOR sysdba.mytablemy_db_link;
```

> **⚠ 注意**
>
> - 新创建同义词名称不能与数据库已存在的对象名称相同，否则报错。
> - 同义词存放于系统表 sys_synonyms 中。

15.3 删除同义词

删除同义词权限按照全局同义词和私有同义词分为以下两种：

- DROP PUBLIC SYNONYM：具有删除所有全局同义词的权限。（仅限库级权限）
- DROP ANY SYNONYM：具有删除所有模式的私有同义词的权限。（仅限库级，模式级权限）

> **⚠ 注意**
>
> 授予权限语法详细信息请参见权限管理章节。
> 同义词的删除，不影响创建同义词时作用的对象。

语法格式

```
DropSynonymStmt::=
    DROP [PUBLIC] SYNONYM [schema_syn_name].syn_name;
```

参数说明

- PUBLIC：删除全局同义词，
- schema_syn_name: 指定包含同义词的模式。如果省略 schema，那么将在您自己的 schema 中删除同义词。如果指定了 PUBLIC，则不能为同义词指定模式会报错。
- synonym_name：同义词名称。

示例

```
DROP SYNONYM my_r_tab;
```

16 模式管理

16.1 概述

模式是数据库系统中数据结构和数据对象的集合。模式可看作一个数据库对象容器，该容器可管理对象包括：表、视图、序列值、包、存储过程、存储函数、触发器、同义词、索引等。

16.2 创建模式

语法格式

```
CreateSchemaStmt::=
    CREATE SCHEMA schema_name [AUTHORIZATION user_name];
```

> **说明**
>
> - 新建模式不能与已存在用户同名，因为创建用户时会默认对其创建同名模式，同名模式在创建用户时占位但不可见，在其用户下创建所属对象时模式可见。
> - 新建模式不能与已存在角色同名，即使采用双引号引用或大小写区别，也无法创建成功。

参数说明

- schema_name：新建模式名。
- AUTHORIZATION：表示将新建模式属主授予给指定用户，此用户拥有该模式的一切权限(修改与删除)。

示例在当前库下创建名为 SCH_TEST 的模式，设置该模式属主为 GUEST。

```
CREATE SCHEMA SCH_TEST AUTHORIZATION GUEST;
```

16.3 修改模式

语法格式

```
AlterSchemaStmt::=
    ALTER SCHEMA schema_name alter_specification;

alter_specification::=
    RENAME TO new_name;
|   OWNER TO user_name;
```

> **⚠ 注意**
>
> 修改模式名仅限于对用户创建模式进行，不能更改用户默认模式信息。

参数说明

- schema_name：模式名。
- new_name：新模式名。
- user_name：模式新属主的名称。

示例

- 示例 1

  将 sch_test 模式重命名为 new_sch_test。

```
ALTER SCHEMA sch_test RENAME TO new_sch_test;
```

- 示例 2

  将 sch_test 模式更换属主为 user_test。

```
ALTER SCHEMA sch_test OWNER TO user_test;
```

16.4 删除模式

语法格式

```
DropSchemaStmt::=
    DROP SCHEMA schema_name [CASCADE | RESTRICT];
```

> **⚠ 注意**
>
> - 删除模式仅限于对用户创建模式进行，不能删除用户默认模式。
> - 删除用户时，同时删除创建用户时默认创建的模式。

参数说明

- schema_name：删除的模式的名称。
- CASCADE：强制删除模式。

示例

```
DROP SCHEMA SCH_TEST;
```

17 定时作业

17.1 概述

虚谷数据库提供定时作业机制，用于定时、定期、自动的进行某些操作，可通过系统包 dbms_scheduler 进行定时作业创建、调度、查看、删除等。

DBMS_SCHEDULER 系统包封装了以下过程/函数：ENABLE、DISABLE、SET_JOB_ARGUMENT_VALUE、DROP_JOB、RUN_JOB。

- 启用作业。

```
DBMS_SCHEDULER.ENABLE(job_name)
```

- 禁用作业。

```
    DBMS_SCHEDULER.DISABLE(job_name, TURE)
```

- 对 job 调用存储过程进行参数赋值，param_order 为参数序，param_val 为参数值。

```
DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE(job_name, param_order,
   param_val)
```

- 显式调用定时作业。

```
DBMS_SCHEDULER.RUN_JOB(job_name, TRUE)
```

- 删除定时作业。

```
DBMS_SCHEDULER.DROP_JOB(job_name, TRUE)
```

17.2 创建作业

语法格式

```
DBMS_SCHEDULER.CREATE_JOB (
    job_name              IN VARCHAR,
    job_type              IN VARCHAR,
    job_action            IN VARCHAR,
    number_of_arguments   IN INTEGER              DEFAULT 0,
    start_date            IN TIMESTAMP            DEFAULT NULL,
    repeat_interval       IN VARCHAR              DEFAULT NULL,
    end_date              IN TIMESTAMP            DEFAULT NULL,
    job_class             IN VARCHAR              DEFAULT '
      DEFAULT_JOB_CLASS',
```

```
    enabled                 IN BOOLEAN                  DEFAULT FALSE,
    auto_drop               IN BOOLEAN                  DEFAULT TRUE,
    comments                IN VARCHAR                  DEFAULT NULL);
```

参数说明

- job_name：作业名称。
- job_type：作业类型，可指定为 stored_procedure、plsql_block 或 plsql_command；若为 stored_procedure 则 job_action 内容为数据库存储过程名称；若为 plsql_block 则 job_action 为可执行块语句；若为 plsql_command 则 job_action 可调用包中的存储过程或存储函数。
- job_action：作业动作，与 job_type 相关。
- number_of_arguments：作作业中存储过程、存储函数或包的参数个数。
- start_date：作业开始时间。
- repeat_interval：作业重复间隔说明。
- end_date：作业结束时间。
- job_class：作业类型——该参数暂时无效，预留。
- enabled：作业是否已激活，若该参数置为 true 则表示该作业默认为启用状态，作业根据其计划自动运行，若该参数置为 false 则该作业为禁用状态，不会自动执行，需手动执行作业或将该作业启用后方可自动执行。
- auto_drop：作业执行完是否自动删除。若该参数置为 true 则在作业完成后将自动删除作业，反之不会删除该作业。
- comments：作业备注说明。

repeat_interval 说明 repeat_interval 语法结构：

```
REPEAT_INTERVAL => 'Freq=<time_period>; Interval=<integer>;
[BYHOUR=<hour_list>
| BYDAY=<day_list>
| BYMONTHDAY=<day_list>
| BYMONTH=<month_list>
| BYWEEKNO=<week_list>
| BYYEARDAY=<day_list>
| BYDATE=<date_list>
| BYMINUTE=<minute_list>
| BYSECOND=<second_list>]'
```

- Freq 关键字用于指定作业间隔的时间周期，<time_period> 可选参数包括：YEARLY（年）、MONTHLY（月）、WEEKLY（周）、DAILY（日）、HOURLY（小时）、MINUTELY（分）、SECONDLY（秒）。
- Interval 关键字用于指定作业间隔频度，该值为一个整数，默认为 1，可指定范围为 1-999。
- BYHOUR：指示定时作业在指定小时执行，可指定范围 0-23，若要指定多个时间参数使用逗号进行分割，如 BYHOUR=2,5,7。
- BYDAY：指示在每周的第几天运行，可使用数字或英文缩写，如：MON|TUE|WED|THU|FRI|SAT|SUN 等。
- BYMONTHDAY：指示在每月的第几天运行。
- BYMONTH：指示在每年的月份，可使用数字或英文缩写，如：JAN|FEB|MAR|APR|MAY|JUN 等。
- BYWEEKNO：指定在一年的哪一周运行，范围 1-53。
- BYYEARDAY：指定在一年的哪一天运行，范围 1-366。
- BYDATE：指定一年的哪一天运行，范围 1-366，类型与 BYYEARDAY 不同。
- BYMINUTE：指定在小时内的哪一分钟运行，范围 0-59。
- BYSECOND：指定在分钟内的哪一秒运行，范围 0-59。

示例

- 基于存储过程的定时作业。

  创建一个基于存储过程的定时作业，每天的 5 点与 14 点时，每隔 30 秒运行一次。

```
CREATE TABLE JOB_TEST(ID INT,DT DATETIME);

CREATE OR REPLACE PROCEDURE JOB_PROC1(INPUT INTEGER) IS
BEGIN
INSERT INTO JOB_TEST VALUES (INPUT, SYSDATE);
END;

EXEC dbms_scheduler.create_job(
'job1',
'stored_procedure',
'JOB_PROC1',
1,
sysdate,
'freq=secondly;INTERVAL=30;byhour=5,14;',
```

```
'2029-01-01  01:00:00',
'default_class',
FALSE,
TRUE,
'这是一个测试');
```

该示例表示

- 基于 PLSQL 块语句的定时作业。

  创建一个基于 PLSQL 块语句的定时作业，该定时作业在每天的 5 点与 14 点时，每隔 30 秒运行一次。

```
EXEC dbms_scheduler.create_job('job2',
'plsql_block',
'DECLARE BEGIN FOR i IN 1..10 LOOP INSERT INTO JOB_TEST VALUES(i,
   sysdate);END LOOP;END;',
0,
sysdate,
'freq=secondly;INTERVAL=30;byhour=5,14;',
'2029-01-01  01:00:00',
'default_class',
FALSE,
TRUE,
'这是一个测试');
```

17.3 查询作业

定时作业创建后，可通过以下系统视图查询定时作业详细信息：

- ALL_JOBS
- DBA_JOBS
- USER_JOBS

示例

创建立即启用的示例定时作业，1 分钟后结束，每 5 秒执行一次：

```
EXEC DBMS_SCHEDULER.CREATE_JOB('job1',
'PLSQL_BLOCK',
'DECLARE BEGIN SELECT SLEEP(3000); END;',
0,
SYSDATE,
'FREQ=SECONDLY;INTERVAL=5;',
SYSDATE+1/24/60,
'DEFAULT_CLASS',
TRUE,
FALSE,
NULL);
```

- 示例 1

  查询定时作业当前是否已启用，ENABLE 为 TRUE 即已启用。

```
SELECT JOB_NAME,ENABLE FROM ALL_JOBS WHERE JOB_NAME='job1';
```

- 示例 2

  查询定时作业当前是否正在执行，STATE 为 RUNNING 表示运行状态，IDLE 表示闲置状态。

```
SELECT STATE FROM ALL_JOBS WHERE JOB_NAME='job1';
```

- 示例 3

  查询定时作业是否已完成结束时间前最后一次执行。

  若 STATE 为 IDLE，且结束时间 END_T 与最后运行时间 LAST_RUN_T 之差的绝对值小于间隔时间，则定时作业已完成结束时间前最后一次执行；若大于间隔时间，则定时作业后续还会继续触发执行。

```
SELECT STATE,END_T,LAST_RUN_T,REPET_INTERVAL FROM ALL_JOBS WHERE
   JOB_NAME='job1';
```

17.4 设置作业参数

该过程用于设置作业的参数信息。可通过位置绑定参数也可以通过参数名称绑定参数。

语法格式

ARGUMENT_VALUE 支持 VARCHAR、TINYINT、SMALLINT、INTEGER、BIGINT、FLOAT、DOUBLE、NUMERIC、BOOLEAN、TIME、DATE、DATETIME 数据类型，此处语法仅列出 VARCHAR 类型作为参考。

```
DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE (
    job_name                IN VARCHAR,
    argument_position       IN INTEGER,
    argument_value          IN VARCHAR);

DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE (
    job_name                IN VARCHAR,
    argument_name           IN VARCHAR,
    argument_value          IN VARCHAR);
```

参数说明

- job_name：需要设置参数的作业名称。
- argument_position：需要设置参数值的参数位置。

- argument_name：需要设置参数值的参数名称。
- argument_value：设置的参数值，该处根据参数值类型自动选择重载的存储过程。

17.5 修改作业属性

该存储过程用于修改定时作业属性值。

语法格式

```
DBMS_SCHEDULER.SET_ATTRIBUTE (
    name            IN VARCHAR,
    attribute       IN VARCHAR,
    value           IN {VARCHAR | TIMESTAMP | INTEGER | BOOLEAN});
```

参数说明

- name：需要修改属性的作业名称。
- attribute：需要修改的属性名称。
- value：修改的属性对应的新值，该参数可接受 VARCHAR、TIMESTAMP、PLS_INTEGER、BOOLEAN 数据类型的值。

> **⚠ 注意**
>
> 定时作业 JOB_TYPE 在作业创建完成后不允许修改。

17.6 显式调用作业

该存储过程用于显式调用作业。

语法格式

```
DBMS_SCHEDULER.RUN_JOB (
   job_name                IN VARCHAR,
   use_current_session     IN BOOLEAN DEFAULT TRUE);
```

参数说明

- job_name：调用的作业名称。
- use_current_session：参数选用 true 时表示使用当前会话执行该作业，若作业未执行完成则当前连接会话一直处于阻塞状态直至作业完成后方可执行其他 SQL；若参数选用 false 则表示使用其他会话执行该作业，当前会话可继续执行其他 SQL。

17.7 启用/禁止作业

ENABLE 存储过程用于启用作业，将作业是否启用属性设置为 true，然后作业根据其开始时间、结束时间和间隔时间自动运行作业。

DISABLE 存储过程用于禁用作业同 ENABLE 作用相反。

语法格式

```
DBMS_SCHEDULER.ENABLE (
    name                    IN VARCHAR);

DBMS_SCHEDULER.DISABLE (
    name                    IN VARCHAR,
    force                   IN BOOLEAN DEFAULT FALSE);
```

参数说明

- name：禁用或启用的作业名称。
- force：如果设置为 true 则需先停止正在运行的作业实例，再禁用作业；如果设置为 false 则允许正在运行的作业执行完成，然后再禁用作业。

17.8 删除作业

该存储过程用于删除作业。

语法格式

```
DBMS_SCHEDULER.DROP_JOB (
   job_name                 IN VARCHAR,
   force                    IN BOOLEAN DEFAULT FALSE);
```

参数说明

- job_name：删除的作业名称。
- force：如果设置为 true 则需先停止正在运行的作业实例，再删除作业；如果设置为 false 则允许正在运行的作业执行完成，然后再删除这些作业。

18 安全性管理

18.1 权限管理

天然权限

天然权限即用户自创建以来就拥有的权限，在虚谷数据库中，默认所有用户均拥有连接数据库的天然权限，同时用户对其拥有对象 (属主为该用户的对象) 拥有一切权限，即使未对该用户赋予任何权限，只要属主为该用户他就可对属主是他的对象进行一切操作。

例如：新用户 a 未被赋予任何权限，如果用户 b 在 a 用户模式下创建了表对象，则用户 a 拥有对该表的一切操作权限，包括增删改查以及删除表。

权限分类

权限有以下 4 种类型：

- 库级权限：管理粒度为整个逻辑库，拥有库级权限的用户或角色可对该库下所有对象进行权限允许范围内的操作。
- 模式级权限：管理粒度为模式，拥有模式级权限的用户或角色可对该模式下所有对象进行权限允许范围内的操作。
- 对象级权限：管理粒度为对象，拥有对象级权限的用户或角色可对指定对象进行权限允许范围内的操作。
- 列级权限：管理粒度为表列或视图列，拥有列级权限的用户或角色可访问和操作权限允许范围内的表列或视图列。

权限继承关系

在虚谷数据库系统中，权限继承主要针对用户与角色或角色与角色，用户继承其所属角色所有权限；用户之间不存在权限继承关系。

权限的授予和回收

在数据库系统中除了用户或角色的天然权限无需进行权限授予外，其他任何权限均需通过权限授予的方式进行权限赋予。

权限回收的语法格式与权限授予一一匹配，在权限回收时仅需将权限授予的关键字 GRANT、TO 改为 REVOKE、FROM 即可。

> **说明**
>
> 权限回收在执行后立即生效。

18.1.1 库级权限

语法格式

- 授予库级权限

```
GrantStmt::=
GRANT sys_privilege [,sys_privilege] TO grantee_list [WITH GRANT
   OPTION]
```

> **说明**
>
> 授权操作类型与操作对象类型应保证匹配，例如针对 PROCEDURE 对象，可为其赋予 CREATE、ALTER、DROP、EXECUTE 等权限，但不能赋予 INSERT、UPDATE、DELETE、SELECT 等权限。

- 回收库级权限

```
RevokeStmt::=
REVOKE GRANT OPTION FOR sys_privilege [,sys_privilege] FROM
   grantee_list
|   REVOKE sys_privilege [,sys_privilege] FROM grantee_list
```

- 权限信息

```
sys_privilege::=
{DBA | SSO | AUDITOR | BACKUP DATABASE | BACKUP | RESTORE |
   RESTORE DATABASE | sys_operation obj_type}

sys_operation::=
{CREATE | CREATE ANY | ALTER ANY | DROP ANY | SELECT ANY | INSERT
    ANY | UPDATE ANY | DELETE ANY | EXECUTE ANY | REFERENCES ANY
    | VACUUM ANY | ENCRYPT ANY}

obj_type::=
{DATABASE | SCHEMA | TABLE | SEQUENCE | INDEX | VIEW | PROCEDURE
    | PACKAGE | TRIGGER | DATABASE LINK | REPLICATION | SYNONYM
    | PUBLIC SYNONYM | USER | ROLE | JOB}

grantee_list::=
{ROLE  UserId | UserId} [{,} {ROLE  UserId | UserId}]...
```

参数说明

- sys_privilege：库级权限。

- grantee_list：被授予/回收权限角色组或用户组，可同时授予/回收多个用户或角色，若角色权限授予/回收则角色下的所有用户均自动授予/回收该权限。
- WITH GRANT OPTION：权限可转授，权限的赋予和回收是关联的，如将 WITH GRANT OPTION 用于对象授权时，被授予的用户也可把此对象权限授予其他用户或角色，权限回收时转授的权限会一同被收回，WITH GRANT OPTION 只能在授予对象级和列级权限时使用。
- REVOKE GRANT OPTION FOR：回收 GRANT 语句中指定的 WITH GRANT OPTION 的转授权限，用户仍然具有该权限，但是不能将该权限转授予其他用户。
- sys_operation：操作类型，包含 DDL、DML 类型，如：CREATE、ALTER、DROP、INSERT、DELETE、UPDATE、SELECT 等，若选择 ANY 参数则表示被授权对象可跨模式进行操作，否则只能在其所有模式进行相应操作。
- obj_type：表示操作对象类型，包含 TABLE、VIEW、PROCEDURE 等。（SYNONYM 为私有同义词、PUBLIC SYNONYM 为全局同义词）

示例

授予 test_user 在当前库下所有模式的创表权限，若无 ANY 关键字则表示，test_user 只能在属于他的模式下进行创表。

```
GRANT CREATE ANY TABLE TO test_user;
```

18.1.2 模式级权限

语法格式

- 授予模式级权限

```
GrantStmt::=
GRANT sys_privilege [,sys_privilege] IN SCHEMA name TO
   grantee_list [WITH GRANT OPTION]
```

- 回收模式级权限

```
RevokeStmt::=
REVOKE GRANT OPTION FOR sys_privilege [,sys_privilege] IN SCHEMA
   name FROM grantee_list
|    REVOKE sys_privilege [,sys_privilege] IN SCHEMA name FROM
   grantee_list
```

- 权限信息

```
sys_privilege::=
{DBA | SSO | AUDITOR | BACKUP DATABASE | BACKUP | RESTORE |
   RESTORE DATABASE | sys_operation obj_type}

sys_operation::=
{CREATE | CREATE ANY | ALTER ANY | DROP ANY | SELECT ANY | INSERT
    ANY | UPDATE ANY | DELETE ANY | EXECUTE ANY | REFERENCES ANY
    | VACUUM ANY | ENCRYPT ANY}

obj_type::=
{DATABASE | SCHEMA | TABLE | SEQUENCE | INDEX | VIEW | PROCEDURE
    | PACKAGE | TRIGGER | DATABASE LINK | REPLICATION | SYNONYM
    | PUBLIC SYNONYM | USER | ROLE | JOB}

grantee_list::=
{ROLE  UserId | UserId} [{,} {ROLE  UserId | UserId}]...
```

参数说明

- sys_privilege：库级权限。
- grantee_list：被授予/回收权限角色组或用户组，可同时授予/回收多个用户或角色，若角色权限授予/回收则角色下的所有用户均自动授予/回收该权限。
- WITH GRANT OPTION：权限可转授，权限的赋予和回收是关联的，如将 WITH GRANT OPTION 用于对象授权时，被授予的用户也可把此对象权限授予其他用户或角色，权限回收时转授的权限会一同被收回，WITH GRANT OPTION 只能在授予对象级和列级权限时使用。
- REVOKE GRANT OPTION FOR：回收 GRANT 语句中指定的 WITH GRANT OPTION 的转授权限，用户仍然具有该权限，但是不能将该权限转授予其他用户。
- sys_operation：操作类型，包含 DDL、DML 类型，如：CREATE、ALTER、DROP、INSERT、DELETE、UPDATE、SELECT 等，若选择 ANY 参数则表示被授权对象可跨模式进行操作，否则只能在其所有模式进行相应操作。
- obj_type：表示操作对象类型，包含 TABLE、VIEW、PROCEDURE 等。（SYNONYM 为私有同义词、PUBLIC SYNONYM 为全局同义词）

示例

授予角色 role_1 在模式 SYSDBA 下的创表权限。

```
GRANT CREATE ANY TABLE IN SCHEMA SYSDBA TO role_1;
```

18.1.3 对象级权限

语法格式

- 授予对象级权限

```
GrantStmt::=
GRANT privileges ON [TABLE | VIEW | PROCEDURE | SEQUENCE]
   name_space TO grantee_list [WITH GRANT OPTION]
```

- 回收对象级权限

```
RevokeStmt::=
REVOKE privileges ON [TABLE | VIEW | PROCEDURE | PACKAGE |
   SEQUENCE] name_space FROM grantee_list
|    REVOKE GRANT OPTION FOR privileges ON name_space FROM
   grantee_list
```

- 权限信息

```
privileges::=
ALL PRIVILEGES
|    ALL
|    operation [,operation]...

operation::=
{SELECT | INSERT | UPDATE | DELETE | EXECUTE | REFERENCES | ALTER
    | DROP | INDEX | TRIGGER | VACUUM}

grantee_list::=
{ROLE  UserId | UserId} [{,} {ROLE  UserId | UserId}]...
```

参数说明

- privileges：授予/回收权限操作类型，包括 SELECT、UPDATE、EXECUTE 等，若要授予/回收对象的所有可操作权限，可使用 ALL 或 ALL PRIVILEGES 代替。
- name_space：此处为指定授予/回收操作权限对象名，对象名与 privileges 操作类型必须匹配，如该对象为 TABLE，则不能授予 EXECUTE 权限。
- grantee_list：被授予/回收权限的用户名或角色名。
- WITH GRANT OPTION：权限可转授，权限的赋予和回收是关联的，如将 WITH GRANT OPTION 用于对象授权时，被授予的用户也可把此对象权限授予其他用户或角色，权限回收时转授的权限会一同被收回，WITH GRANT OPTION 只能在授予对象级和列级权限时使用。
- REVOKE GRANT OPTION FOR：回收 GRANT 语句中指定的 WITH GRANT OPTION 的转授权限，用户仍然具有该权限，但是不能将该权限转授予其他用户。

示例

- 示例 1

  授予用户 U1 对于表 test_permission 的数据插入权限，则用户 U1 对表 test_permisssion 无删除、查询、更改、引用等权限。

```
GRANT CREATE ANY TABLE IN SCHEMA SYSDBA TO role_1;
```

- 示例 2

  授予用户 U1 对于表 test_premission 所有数据库所允许的操作权限，包括：INSERT、UPDATE、DELETE、SELECT、REFERENCES 等。

```
GRANT ALL ON test_permission TO u1;
```

18.1.4 列级权限

语法格式

- 授予列级权限

```
GrantStmt::=
GRANT operation [,operation]... ( name_list ) ON [TABLE]
   name_space TO grantee_list [WITH GRANT OPTION]
```

- 回收列级权限

```
RevokeStmt::=
REVOKE GRANT OPTION FOR operation_commalist ( name_list ) ON
   name_space FROM grantee_list
|    REVOKE operation_commalist ( name_list ) ON [TABLE]
   name_space FROM grantee_list
```

- 权限信息

```
operation::=
{SELECT | INSERT | UPDATE | DELETE | EXECUTE | REFERENCES | ALTER
    | DROP | INDEX | TRIGGER | VACUUM}

grantee_list::=
{ROLE  UserId | UserId} [{,} {ROLE  UserId | UserId}]...
```

> **说明**
>
> 列级权限只包括 SELECT 与 UPDATE 操作权限，且必须指定可操作的对象名，对象类型只能是表或视图。

参数说明

- operation：表示操作类型，包含 DDL、DML 类型，如 CREATE、ALTER、DROP、INSERT、DELETE、UPDATE、SELECT 等。
- name_list：授予/回收可操作的列名。
- name_space：此处指授予/回收可操作的对象名，可为表名或视图名。
- grantee_list：授予/回收权限的用户名或角色名。

示例

授予用户 TU1 针对表 TEST_COL_PRE 的列 ID2 的查询权限与列 ID 的变更权限。

```
CREATE TABLE TEST_COL_PRE(ID INT,ID2 INT,ID3 INT);
CREATE USER TU1 IDENTIFIED BY 'test_123@';
GRANT SELECT(ID2) ON TEST_COL_PRE TO TU1;
GRANT UPDATE(ID) ON TEST_COL_PRE TO TU1;
```

18.2 黑白名单管理

18.2.1 概述

数据库通过配置信任 IP 地址、信任用户和指定数据库来限制用户登录，即黑白名单。其中不允许或不可信的配置项被称为黑名单；允许或可信的项被称为白名单。

黑白名单可通过两种方式进行配置，一种是修改 trust.ini 配置文件，另一种是执行数据库命令。

⚠ 注意

若配置 trust.ini 文件进行控制限制，需在启动数据库服务前或配置后重启生效。

18.2.2 修改配置文件设置

通过修改数据库安装包根路径下 SETUP 文件夹的 trust.ini 文件配置黑白名单。

📖 说明

配置 trust.ini 文件仅对当前节点生效，且需在启动数据库服务前或重启生效，建议所有节点配置信息一致。

语法格式

```
{trust | untrust} {db_name | everydb} { user_name | everyone } from
    ip_str1 [to ip_str2]
```

参数说明

- trust：表示该条配置信息为可信策略，允许配置用户访问数据库。
- untrust：表示该条配置信息为不可信策略，不允许配置用户访问数据库。
- db_name：指定策略所影响的数据库名，若针对所有数据库有效则可配置为 everydb。
- user_name：指定策略所影响的用户名，若针对所有用户有效则可配置为 everyone。
- ip_str1：指定策略所影响的 ip 地址，若针对所有来访 IP 地址则可配置为 anywhere。
- to ip_str2：可选参数，表示从 ip_str1 到 ip_str2 的 IP 段均受策略影响。

示例

- 示例 1

  当前节点只允许由 192.168.1.100 的 sysdba 用户登录到 system 库进行访问。

```
    trust system  sysdba  from  192.168.1.100
```

- 示例 2

  当前节点允许从 192.168.1.100 至 192.168.1.109 这个 IP 段的 SYSDBA 用户登录访问 SYSTEM 库。

```
    trust system  sysdba  from  192.168.1.100 to  192.168.1.109
```

18.2.3 执行数据库命令设置

除了通过上述的修改配置文件进行可信策略配置外，还可通过命令进行可信策略配置。

⚠ 注意

- 所有的可信策略配置只能在系统库进行设置，普通用户库的管理员无权设置可信策略。
- 删除可信策略时必须谨慎，若删除了所有可信策略，则当前登录客户端必须在未断连前新增至少一个白名单，否则所有用户均无法登录数据库，只有修改配置后重启服务才行。

语法格式

```
ConnectPolicyStmt::=
ALTER CONNECT POLICY ADD {ENABLE | DISABLE} user_name LOGIN db_name
    FROM ip_addr [TO ip_addr] opt_on_node
|   ALTER CONNECT POLICY DROP {ENABLE | DISABLE} user_name LOGIN
   db_name FROM ip_addr [TO ip_addr] opt_on_node
|   ALTER CONNECT POLICY DROP ALL {ENABLE | DISABLE} opt_on_node
|   ALTER CONNECT POLICY DROP ALL {ENABLE | DISABLE} LOGIN db_name
   opt_on_node
```

```
|   ALTER CONNECT POLICY DROP ALL {ENABLE | DISABLE} user_name
   LOGIN db_name opt_on_node
|   ALTER CONNECT POLICY DROP ALL opt_on_node

opt_on_node::=
[ON ALL NODE | ON NODE ICONST]
```

参数说明

- ENABLE：表示该条可信策略为白名单策略。
- DISABLE：表示该条可信策略为黑名单策略。
- user_name：表示可信策略所影响的用户名，所有用户可用 everyone 表示。
- db_name：表示可信策略所影响的库名，所有库可用 everydb 表示。
- ip_addr：表示可信策略所影响的 IP 地址或 IP 地址段，所有地址可用 from anywhere 表示。
- ON ALL NODE：多节点，可信策略对所有节点生效，单节点可不配置。
- ON NODE ICONST：其中 ICONST 为节点号，表示黑白名单在 ICONST 节点号生效，若未使用 ON ALL NODE 和 ON NODE ICONST 则表示黑白名单在当前节点生效。

> ⚠ 注意
>
> 在 trust.ini 中默认配置 trust everydb everyone from anywhere，该项覆盖了所有白名单选项，单独设置白名单无意义，若需白名单生效，需删除该配置项，可在启动服务前修改配置生效或通过命令删除该项。

示例

- 示例 1

  增加一个白名单，允许用户 u1 使用 192.168.3.105 访问任意数据库。

```
ALTER CONNECT POLICY ADD ENABLE 'u1' LOGIN 'everydb' FROM
'192.168.3.105' ON ALL NODE;
```

- 示例 2

  增加一个白名单，允许用户 u2 使用 192.168.3.107 至 192.168.3.110 IP 地址段访问数据库 test。

```
ALTER CONNECT POLICY ADD ENABLE 'u2' LOGIN 'test' FROM
'192.168.3.107' TO '192.168.3.110' ON ALL NODE;
```

- 示例 3

增加一个黑名单，不允许用户 u3 使用任何 IP 地址访问数据库 test，即拒绝 u3 登录 test。

```
ALTER CONNECT POLICY ADD DISABLE 'u3' LOGIN 'test' FROM 'anywhere
   ' ON ALL NODE;
```

- 示例 4

  增加一个黑名单，不允许任何用户使用 192.168.3.107 至 192.168.3.120 IP 地址段登录数据库 test。

```
ALTER CONNECT POLICY ADD DISABLE 'everyone' LOGIN 'test' FROM
'192.168.3.107' TO '192.168.3.120' ON ALL NODE;
```

- 示例 5

  删除一个白名单策略，该策略为允许用户 u1 使用 192.168.3.105 登录到任意数据库。

```
ALTER CONNECT POLICY DROP ENABLE 'u1' LOGIN 'everydb' FROM
'192.168.3.105' ON ALL NODE;
```

- 示例 6

  删除一个白名单策略，该策略为允许用户 u2 使用 192.168.3.107 至 192.168.3.110 IP 地址段访问数据库 test。

```
ALTER CONNECT POLICY DROP ENABLE 'u2' LOGIN 'test' FROM
'192.168.3.107' TO '192.168.3.110' ON ALL NODE;
```

- 示例 7

  删除一个黑名单策略，该策略为不允许用户 u3 使用任何 IP 地址访问数据库 test。

```
ALTER CONNECT POLICY DROP DISABLE 'u3' LOGIN 'test' FROM '
   anywhere' ON ALL NODE;
```

- 示例 8

  删除一个黑名单策略，该策略为不允许所有用户使用 192.168.3.107 至 192.168.3.120 IP 地址段访问数据库 test。

```
ALTER CONNECT POLICY DROP DISABLE 'everyone' LOGIN 'test' FROM
'192.168.3.107' TO '192.168.3.120' ON ALL NODE;
```

- 示例 9

  删除集群中所有节点上的白名单策略。

```
ALTER CONNECT POLICY DROP ALL ENABLE ON ALL NODE;
```

- 示例 10

  删除关于数据库 test 的所有白名单策略。

```
ALTER CONNECT POLICY DROP ALL ENABLE LOGIN 'test' ON ALL NODE;
```

- 示例 11

  删除所有黑名单策略。

```
ALTER CONNECT POLICY DROP ALL DISABLE ON ALL NODE;
```

- 示例 12

  删除关于数据库 test 的所有黑名单策略。

```
ALTER CONNECT POLICY DROP ALL DISABLE LOGIN 'test' ON ALL NODE;
```

- 示例 13

  删除所有配置的可信策略，包括黑名单与白名单。

```
ALTER CONNECT POLICY DROP ALL ON ALL NODE;
```

⚠ 注意

- 所有的可信策略配置只能在系统库进行设置，普通用户库的管理员无权设置可信策略。
- 删除可信策略时必须谨慎，若删除了所有可信策略，则当前登录客户端必须在未断连前新增至少一个白名单，否则所有用户均无法登录数据库，只有修改配置后重启服务才行。

18.3 安全策略管理

18.3.1 概述

数据库通过安全策略实现强制访问控制。安全策略是一组预定义的标记，根据标记的值来确定主体（用户）对客体（表）是否拥有某种权限，用于实现细粒度的访问控制。

安全策略中，包括等级和范畴两种组件。

安全策略由数据库的安全管理员（SYSSSO）或拥有安全管理员权限的用户管理：

- 创建、更改和删除安全策略。更改安全策略，包括更改安全策略名、给安全策略添加或删除组件（等级和范畴）、更改安全策略中组件的名称或值。
- 给指定的主体或客体添加、更改或删除安全策略。

📖 说明

数据库最多支持 47 个策略。

安全等级

等级由名称和值两部分组成，其中值为从 0 到 30000 的整数。

等级用于读访问控制时，主体（用户）的等级必须大于客体（表）的等级；用于写访问控制时，主体（用户）的等级必须小于等于客体（表）的等级。如果不满足相应条件，则读或写会被拒绝。

安全范畴

范畴是集合类型，集合中每个元素都是一个名称。不同的范畴之间没有等级高低之分，但可以进行比较（采用集合间的包含关系）。

范畴用于读写访问控制时，主体（用户）的标记必须包含客体（表）的所有范畴。如果不满足相应条件，则读或写会被拒绝。

安全标记

当把安全策略应用于客体（表）时，相应的主客体便获得了相应的安全标记。一个安全标记由等级和范畴名组成。

格式为’ 等级: 范畴’，如’level_3:category_2’。

此外，添加或更改安全标记时，可指定标记是否被隐藏。

18.3.2 创建安全策略

语法格式

```
CreatePolicyStmt::=
    CREATE POLICY policy_name [ PolicyMemberClass
    [, PolicyMemberClass [, ... ] ] ]

PolicyMemberClass::=
    ADD LEVEL level_name AS number_value
|   ADD CATEGORY category_name
```

参数说明

- policy_name: 要创建的安全策略名。
- level_name: 安全等级名。
- level_number: 安全等级的值，取值为正整数。
- category_name: 安全范畴名。

示例

- 创建不带等级和范畴的安全策略。

```
SQL> CREATE POLICY policy_1;
```

- 创建带等级的安全策略。

```
SQL> CREATE POLICY policy_2 ADD LEVEL level_1 AS 1;
```

- 创建带范畴的安全策略。

```
SQL> CREATE POLICY policy_3 ADD CATEGORY category_1;
```

- 创建带等级和范畴的安全策略。

```
SQL> CREATE POLICY policy_4 ADD LEVEL level_1 AS 1,ADD LEVEL
   level_2 AS 2,ADD CATEGORY category_1,ADD CATEGORY category_2;
```

18.3.3 修改安全策略

语法格式

```
AlterPolicyStmt::=
    ALTER POLICY policy_name AlterMemberClauss [, AlterMemberClauss
    [, ... ] ]

AlterMemberClauss::=
    RENAME TO new_policy_name
|   ADD LEVEL level_name AS number_value
|   ADD CATEGORY category_name
|   ALTER LEVEL level_name RENAME TO new_level_name
|   ALTER CATEGORY category_name RENAME TO new_category_name
|   DROP LEVEL level_name
|   DROP CATEGORY category_name
```

参数说明

- policy_name: 安全策略名。
- new_policy_name: 新的安全策略名。
- level_name: 安全等级名。
- number_value: 安全等级的值，取值为正整数。
- category_name: 安全范畴名。
- new_level_name: 新的安全等级名。
- new_category_name: 新的安全范畴名。

示例

- 更改安全策略名。

```
SQL> ALTER POLICY policy_1 RENAME TO policy_001;
```

- 更改等级名。

```
SQL> ALTER POLICY policy_2 ALTER LEVEL level_1 RENAME TO level_2;
```

- 删除范畴。

```
SQL> ALTER POLICY policy_3 DROP CATEGORY category_1;
```

- 增加多个等级的同时删除一个范畴。

```
SQL> ALTER POLICY policy_4 ADD LEVEL level_3 AS 3, ADD LEVEL
   level_4 AS 4,
DROP LEVEL level_1,DROP CATEGORY category_1;
```

18.3.4 删除安全策略

语法格式

```
DropPolicyStmt::=
    DROP POLICY policy_name
```

参数说明 policy_name: 安全策略名。

示例

```
SQL> DROP POLICY policy_4;
```

18.3.5 为用户（主体）添加、更改、删除安全策略

语法格式

```
AlterUserPolicyStmt::=
    ALTER USER POLICY user_name {ADD | ALTER} policy_name LEVEL
   level_name [CATEGORY category_name]
|   ALTER USER POLICY user_name DROP policy_name
```

参数说明

- user_name: 用户名。
- policy_name: 安全策略名。
- level_name: 安全等级名。
- category_name: 安全范畴名。

示例

- 为用户添加安全策略、等级、范畴。

```
SQL> ALTER USER POLICY usr_1 ADD policy_4 LEVEL level_3 CATEGORY
   category_2;
```

- 为用户删除安全策略。

```
SQL> ALTER USER POLICY usr_1 DROP policy_4;
```

18.3.6 为表（客体）添加、更改、删除安全策略

语法格式

```
AlterTabPolicyStmt::=
    ALTER TABLE POLICY user_name.table_name ADD policy_name COLUMN
   column_name [ [NOT] HIDE ] LABEL label_content
|   ALTER TABLE POLICY user_name.table_name ALTER column_name
    [ [NOT] HIDE ]
|   ALTER TABLE POLICY user_name.table_name DROP policy_name
```

参数说明

- user_name: 表所属模式名。
- table_name: 表名。
- policy_name: 安全策略名。
- column_name: 列名，需为目标表中不存在的列名。
- label_content: 安全标记的内容，内容整体要用单引号包裹。格式如下。
  - ’ 安全等级名:’
  - ’ 安全等级名: 安全范畴名’

示例

- 为表添加安全策略、带等级和范畴的列。

```
SQL> ALTER TABLE POLICY usr_1.tab_test ADD policy_4 COLUMN col_4
   NOT HIDE LABEL 'level_3:category_2';
```

- 为表删除安全策略。

```
SQL> ALTER TABLE POLICY usr_1.tab_test DROP policy_4;
```

18.3.7 综合示例

本章节创建一个用户 usr_1，并创建两张表插入数据，对主体（用户）和客体（表）添加不同的安全等级，用户通过对表进行查询、修改等操作进行示例。

1. SYSDBA 用户登录，创建用户并创建表插入值。

```
-- 创建用户与授权
SQL> CREATE USER usr_1 IDENTIFIED BY 'QWEasd12#@';
SQL> GRANT DBA TO usr_1;
```

```
-- 创建表并插入值
SQL> CREATE TABLE usr_1.tab_test_1(c1 INT, c2 VARCHAR);
SQL> CREATE TABLE usr_1.tab_test_2(c1 INT, c2 VARCHAR);
SQL> INSERT INTO usr_1.tab_test_1 VALUES(1, 'a');
SQL> INSERT INTO usr_1.tab_test_1 VALUES(2, 'b');
SQL> INSERT INTO usr_1.tab_test_2 VALUES(1, 'alpha');
SQL> INSERT INTO usr_1.tab_test_2 VALUES(2, 'beta');
```

2. SYSSSO 用户登录，创建安全策略，包括 3 个等级，并为用户和表添加不同的等级。

```
-- 创建安全策略
SQL> CREATE POLICY policy_1 ADD LEVEL level_1 AS 1,ADD LEVEL
   level_2 AS 3,ADD LEVEL level_3 AS  5;

-- 用户等级为3
SQL> ALTER USER POLICY usr_1 ADD policy_1 LEVEL level_2;

-- 表等级分别为1、5
SQL> ALTER TABLE POLICY usr_1.tab_test_1 ADD policy_1 COLUMN c3
    NOT HIDE LABEL 'level_1:';
SQL> ALTER TABLE POLICY usr_1.tab_test_2 ADD policy_1 COLUMN c3
    NOT HIDE LABEL 'level_3:';
```

3. usr_1 用户登录，等级为 3，大于 tab_test_1 的等级 1，小于 tab_test_2 的等级 5，因此修改表 tab_test_1 出现错误，并无法读取表 tab_test_2 数据。

```
-- 修改表数据
SQL> UPDATE tab_test_1 SET c2 = 'c' WHERE c1 = 2;
[E18028] 更改操作违反强制安全控制策略

-- 读取表数据
SQL> SELECT * FROM tab_test_1;
C1 | C2 | C3 |
-----------------------------------------------------------------
1 | a| 281474976710656 |
2 | b| 281474976710656 |

SQL> SELECT * FROM tab_test_2;

C1 | C2 | C3 |
-----------------------------------------------------------------
```

19 会话变量

19.1 概述

会话变量存在于会话环境中，即连接的控制变量区中，各个连接拥有的会话变量的名称相同，但各会话都有会话变量的独立拷贝，它们可以具有与别的会话的同名变量不相同的取值，会话变量对于与该会话关联的事务的行为方式具有重要影响，修改会话变量的值，即可影响该连接下事务处理的行为方式。

在会话（即连接）刚创建时，该会话环境中的会话变量也同时创建，而且系统将为它们设置默认值。在会话创建后，客户端可以发送命令修改会话变量，使该连接下的事务按照用户需要的方式执行，会话变量中有一些为预定义的，其它的则是在会话过程中动态创建的。

会话环境中的会话变量分为全局会话变量和局部会话变量。全局指的是连接到数据库的所有会话都遵循的变量设置，局部指的是当前会话遵循当前会话变量设置。比如 SET error_level TO 3 为全局变量设置，SET auto_commit ON 为局部变量设置。

全局变量的设置一般都是由系统管理员进行，普通用户没有权限设置全局变量。

19.2 自定义会话变量

功能特性

- 访问任何未经定义的自定义会话变量都将返回一个 NULL 值。
- 自定义会话变量是特定于会话的，即其生命周期由会话决定。
- 自定义会话变量由一个客户端定义、其它客户端无法查看和使用。
- 自定义会话变量会在客户端会话结束时自动回收释放。
- 自定义会话变量不支持右值为一个查询语句。

语法格式

- 定义会话变量

```
SET @var_name = expr [, @var_name = expr] ...
```

- 使用会话变量

```
@var_name
```

> **说明**
>
> 会话变量支持在 SQL、块语句、存储过程或函数中使用。

参数说明

- var_name：变量名，可以是字母或下划线开头的，且由大小写字母、数字、下划线、$ 或 # 中任意字符组成的的标识符。
- expr：变量值。

示例

```
SQL> SET @abc = sysdate();
SQL> SELECT @abc;

EXPR1
----------------
2024-06-05 15:46:17.281 AD

SQL> SET @abc = 1000;
SQL> SET @abc = @abc * 10;
SQL> SELECT @abc;

EXPR1
----------------
10000
```

19.3 查看变量

查看数据库监听服务信息

```
SHOW server_info;
```

数据库监听服务信息, 包括数据库服务所在计算机名称、服务器 ip 地址和监听端口号。

查看数据库版本信息

```
SHOW version;
```

显示当前服务器的版本号。

当前数据库信息

```
SHOW db_info;
```

显示当前连接逻辑库信息，如数据库名称、库 id 等。

查看服务线程状态

```
SHOW thd_status;
```

此命令显示当前系统中所有线程的当前状态。

查看系统数据类型

```
SHOW data_types;
```

此命令显示当前系统中有效的数据类型，包括系统内建的和外扩的数据类型。

查看系统函数

```
SHOW methods;
```

此命令显示当前系统中所有内建函数。

查看系统操作符

```
SHOW operators;
```

此命令显示当前系统中所有内建操作符。

查看数据库字符集

```
SHOW charsets;
```

显示系统支持的字符集。

查看全局内存状态

```
SHOW mem_status;
```

此命令执行后显示当前节点内存使用状态。

查看服务器操作系统类型

```
SHOW os_type;
```

显示服务器当前操作系统的类型。

查看计算机指令集

```
SHOW machine_type;
```

x86 是一个 intel 通用计算机系列的标准编号缩写, 也标识一套通用的计算机指令集合。

查看操作记载状态

```
SHOW reg_command;
```

此命令用于显示是否记载数据库操作命令，若为 TRUE，则会在 XGLOG 目录生成对应的 COMMAND.LOG 用于操作命令记载。

查看错误记载等级

```
SHOW error_level;
```

此命令用于显示数据库错误记载等级。错误等级包括 4 个等级：

- 1：记载系统错误。
- 2：记载 SQL 语句错误。
- 3：记载前面两种错误类型。
- 4：记载 SQL 语句警告。

等级越高，错误信息记载越详细。

查看当前会话默认模式

```
SHOW current_schema;
```

显示用户当前所在模式。

```
SET current_schema TO schema_name;
```

设置用户当前所在模式为 schema_name。

查看当前会话时区

```
SHOW def_timezone;
```

显示当前会话的时区。

```
SET def_timezone TO  'gmt+08:00';
```

设置当前会话到某个时区。

查看当前会话事务提交类型

```
SHOW auto_commit;
```

显示当前的‘读已提交’是否开启，返回值为 BOOLEAN 类型，TRUE 表示当前会话为自动提交，FALSE 表示当前会话为非自动提交。

```
SET auto_commit ON[OFF];
```

设置事务是否为自动提交。

19.4 连接会话参数

19.4.1 概述

连接会话参数主要用于在客户端向虚谷数据库服务端建立连接请求后，在进行会话验证时，传输额外的信息内容从而达到修饰连接会话自身属性的目的。详细的连接会话参数信息请参见《系统配置参数参考》的连接会话参数章节。

语法格式

- 设置会话变量

```
VariableSetStmt::=
SET ColId TO var_value ;
|       SET ColId TO ColId;
|       SET ColId ON;
|       SET ColId OFF;
|       ALTER SESSION SET ColId '=' var_value;
|       ALTER SESSION SET ColId '=' ColId;

var_value::=
opt_boolean
|       SCONST
|       ICONST
|       '-' ICONST
|       FCONST
|       '-' FCONST
|       DEFAULT

opt_boolean::= _TRUE
|       _FALSE
|       ON
|       OFF
```

- 设置事务隔离级别

```
VariableSetStmt::=
SET TRANSACTION ISOLATION LEVEL opt_level;
|       SET TRANSACTION READ ONLY;
|       SET TRANSACTION READ WRITE;
|       SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION
   LEVEL opt_level;
|       SET SESSION TRANSACTION ISOLATION LEVEL opt_level;

opt_level::=
READ ONLY
|       READ COMMITTED
|       _REPEATABLE READ
|       SERIALIZABLE
```

- 设置会话授权用户

```
VariableSetStmt::=
SET SESSION AUTHORIZATION SCONST;
|       SET SESSION AUTHORIZATION UserId;
```

- 设置自动提交行为

```
VariableSetStmt::=
SET AUTO COMMIT ON
|       SET AUTO COMMIT OFF
```

- 设置当前模式

```
VariableSetStmt::=
SET CURRENT SCHEMA name
|       SET CURRENT SCHEMA DEFAULT
|       SET SCHEMA name
|       SET SCHEMA DEFAULT
|       SET SCHEMA Sconst
```

- 设置空值处理

```
VariableSetStmt::=
SET _NULL _NULL
|       SET _NULL SCONST
```

参数说明

- ColId ： 会话变量。
- var_value：数据类型返回模式。
- opt_level：事务隔离级别。

19.4.2 AUTO_COMMIT

功能特性

客户端连接会话上事务的提交模式，提交模式分为自动提交和非自动提交。

参数说明

- ON|TRUE：自动提交模式
- 其它值：非自动提交模式

示例

- 设置提交模式

```
SET AUTO_COMMIT ON;
```

- 显示参数状态

```
SHOW AUTO_COMMIT;
```

> 说明
>
> 该参数在驱动程序缺省未指定的情况下，由系统全局参数 def_auto_commit 决定，默认自动提交。

19.4.3 CHAR_SET

功能特性

客户端连接会话所使用的字符集编码，服务端会依据 char_set 参数给定的字符集对来自该连接会话上的字符内容和对经由该连接会话发往客户端的字符内容进行转码。

数据库端字符集转换：

- 读入：会话字符集-> 库字符集转换
- 写出：库字符集-> 会话字符集

> **说明**
>
> - bin 和 ci 后缀字符集区别：ci 不区分大小写，bin 区分大小写。
> - 创建库和连接会话字符集在不指定 bin 和 ci 后缀时使用 bin 类字符集（同一字符集不同后缀也涉及字符集转换问题）。
> - 当前版本 JDBC 驱动不支持指定字符集后缀。

示例

- 设置字符集为 gbk

```
SET CHAR_SET to 'gbk';
```

- 查看字符集

```
SHOW CHAR_SET;
```

> **说明**
>
> 字符集默认使用 def_charset。

19.4.4 COMPATIBLE_MODE

功能特性

用于兼容其他数据库模式。

参数说明

- ORACLE：

  兼容 Oracle：不带双引号标识符转大写

- MYSQL：

  兼容 MySQL：

  - 无论标识符是否有双引号转都不做大小写转换

- 兼容 MySQL 双引号使用
- 无别名的非字段表达式返回名称兼容

- POSTGRESQL：

  兼容 PG：不带双引号标识符转小写

- NONE：Xugu：

  不带双引号标识符转小写

示例

- 设置模式为 NONE

```
SET COMPATIBLE_MODE TO 'NONE';
```

- 显示参数状态

```
SHOW COMPATIBLE_MODE;
```

> **说明**
>
> 相关参数 def_compatible_mode，连接串未设置时，COMPATIBLE_MODE 设置为 def_compatible_mode。

19.4.5 DATABASE

功能特性

当前会话是否禁用 BINLOG 日志记载。

19.4.6 DISABLE_BINLOG

功能特性

用于兼容其他数据库模式。

参数说明

- OFF|FALSE：不禁用
- ON|TRUE：禁用

示例

- 设置不禁用 BINLOG 日志记载

```
SET DISABLE_BINLOG OFF;
SET DISABLE_BINLOG TO 'FALSE'
```

- 显示参数状态

```
SHOW DISABLE_BINLOG;
```

19.4.7 DRIVER_VERSION

功能特性

客户端连接会话所使用的通讯协议版本号，指定不同的版本号将使用不同的通讯协议。

参数说明

- 301：301 通讯协议
- 其它值：201 通讯协议

示例

查看当前数据库通信协议。

```
SHOW DRIVER_VERSION;
```

19.4.8 EMPTY_STR_AS_NULL

功能特性

数据库服务端将结果集发向客户端时，是否将空字符串当作 NULL 处理。

该参数在驱动程序缺省未指定的情况下，由系统全局参数 def_empty_str_as_null 决定；若该参数已经由驱动程序明确指定，则会覆盖系统全局参数 def_empty_str_as_null 给定的值。

参数说明

- TRUE：将空字符串当作 NULL 处理
- FALSE：不将空字符串当作 NULL 处理

示例

- 设置不将空字符串当作 NULL 处理

```
SET EMPTY_STR_AS_NULL TO 'false';
```

- 显示参数状态

```
SHOW EMPTY_STR_AS_NULL;
```

> **说明**
>
> 在数据库初始化 ini 文件中存在重名参数，会话参数仅影响 SQL 参数处理（如 prepare 批量插入）。

19.4.9 IDENTITY_MODE

功能特性

自增列模式用于控制自增列（identity）插入值填充模式，以兼容 MySQL，默认值为 def_identity。

参数说明

- NULL_AS_AUTO_INCREMENT：1，指定 NULL 值时，插入自增值，等价于连接会话参数 IDENTITY_MODE 取值
- ZERO_AS_AUTO_INCREMENT：2，ZERO_AS_AUTO_INCREMENT：指定 NULL 和 0 时，插入自增值，等价于连接会话参数 IDENTITY_MODE 取值
- DEFAULT：0，不取值或 DEFAULT：不指定列时，使用自增值，如果指定值是 NULL 和 0，按自增列约束报错

示例

- 设置 NULL_AS_AUTO_INCREMENT 模式

```
    SET IDENTITY_MODE TO NULL_AS_AUTO_INCREMENT;
```

- 显示参数状态

```
    SHOW IDENTITY_MODE;
```

> **说明**
>
> 默认模式为”DEFAULT”。

19.4.10 ISO_LEVEL

功能特性

客户端连接会话上的事务隔离级别。

参数说明

- READ ONLY：只读（0）
- READ_COMMITTED：读已提交（1）
- REPEATABLE READ：可重复读（2）
- SERIALIZABLE：序列化（3）

示例

- 设置隔离级别，支持两种设置

```
SET ISO_LEVEL TO '0';
SET TRANSACTION ISOLATION LEVEL READ ONLY;
```

- 显示隔离级别

```
SHOW ISO_LEVEL;
SHOW TRANSACTION ISOLATION LEVEL;
```

> 说明
>
> 不指定则使用 def_iso_level 配置值，默认读已提交（1）。

19.4.11 KEYWORD_FILTER

功能特性

关键字过滤，当设置的过滤关键字出现在特定操作关键字之后（如：SELECT/INSERT/UPDATE/DELETE/MERGE/CREATE TABLE），则设置的过滤关键字会被作为标识符进行处理。

> 说明
>
> 关键字过滤后，只有 DML 语句和创建表的对象可以使用被过滤的关键字，其他对象（如库、序列、过程、函数等）将不能使用这些关键字。

示例

将需要做对象名、列表、别名的参数通过逗号分隔进行设置，禁止使用空格，区分大小写。

```
--连接串设置方式
KEYWORD_FILTER=TABLE
KEYWORD_FILTER=TABLE,FUNCTION
KEYWORD_FILTER=TABLE,FUNCTION,CONSTANT

--set设置方式
SET KEYWORD_FILTER TO 'TABLE';
SET KEYWORD_FILTER TO 'TABLE,FUNCTION';
SET KEYWORD_FILTER TO 'TABLE,FUNCTION,CONSTANT';
```

可通过 sys_sessions 系统表 KEYWORD_FILTER 字段和命令 SHOW KEYWORD_FILTER 查看。

> **说明**
>
> 连接上配置关键字后，此关键字相关 DML 语法将无法使用，如需使用相关语法只能通过无关键字过滤的连接执行。如将 SELECT 语句中 FROM/GROUP/ORDER/OR/AND 设置为过滤关键字后，相关 SELECT 将无法执行。

19.4.12 LANGUAGE

功能特性

数据库服务端可支持的 SQL 语法树规则。

参数说明

- TSQL：TSQL 语法树规则（已弃用）
- 其它值：PLSQL 语法树规则

示例

查看当前数据库支持的 SQL 语法树规则。

```
SHOW LANGUAGE;
```

19.4.13 LOB_RET

功能特性

在获取大对象结果集时，是否将大对象作为描述符返回。

参数说明

- ON|TRUE：返回大对象描述符
- 其它值：以二进制的形式返回大对象

示例

- 设置以二进制的形式返回大对象

```
SET LOB_RET TO 'false';
```

- 显示参数状态

```
SHOW LOB_RET;
```

> **说明**
>
> 默认 false 直接返回数据。

19.4.14 OPTIMIZER_MODE

功能特性

优化模式。

参数说明

- ALL ROWS：0
- FIRST ROWS：1

示例

- 设置为 ALL_ROWS

```
SET OPTIMIZER_MODE TO 'ALL_ROWS';
```

- 显示参数状态

```
SHOW OPTIMIZER_MODE;
```

> **说明**
>
> 优化模式，默认使用 def_optimize_mode，默认设置为 0。

19.4.15 PASSWORD

指定参数 USER 表示的用户身份认证口令（静态口令）。

<DATABASE, USER, PASSWORD> 属性组用于登录时用户身份认证。登录成功后 DATABASE 和 USER 保存相关信息在该连接会话中，在连接会话生命周期内作为所有操作的执行上下文，如用于：

- 当前模式
- 执行时权限检测
- 创建模式对象的属主设置
- 审计记载

19.4.16 RESULT

功能特性

各数据类型返回值模式。

参数说明

- CHAR 类型：客户端以字符串形式接收处理
- 其它值：客户端以类型定义方式接受处理

示例

- 客户端以字符返回

```
    SET RESULT TO 'char';
```

- 客户端以类型定义方式返回

```
    SET RESULT TO '';
```

- 查看返回类型

```
    SHOW RESULT;
```

各类型转字符格式说明如下表所示。

编号	类型	格式
1	bool	不支持
2	tinyint	数字串
3	short	数字串
4	int	数字串
5	bigint	数字串
6	float	数字串
7	double	数字串
8	date	yyy-mm-dd
9	datetime	yyyy-mm-dd hh24:mi:ssss
10	timestamp	yyyy-mm-dd hh24:mi:ssss
11	time	%02d:%02d:%02d（无毫秒） %02d:%02d:%02d:%f（带毫秒）

接下页

编号	类型	格式
12	datetime with time zone	yyyy-mm-dd hh24:mi:sssss
13	timestamp with time zone	yyyy-mm-dd hh24:mi:sssss
14	interval year	数字串
15	interval month	数字串
16	interval day	数字串
17	interval hour	数字串
18	interval minute	数字串
19	interval second	%d.%06d
20	interval year to month	%d-%02d（-分隔 year 和 month）
21	interval day to hour	%d %02d（空格分隔 day 和 hour）
22	interval day to minute	%d %02d:%02d
23	interval day to second	%d %02d:%02d:%02d.%06d(带毫秒)
24	interval hour to minute	%d:%d
25	interval hour to second	%d:%d:%.9g
26	interval minute to second	%d:%.9g
27	udt	[%s,%s...]

说明

未列出的类型为本身就是字符串方式或不支持 CHAR 方式返回（如 binary、blob、bool、rowid、char、varchar 等）。

19.4.17 RETURN_CURSOR_ID

功能特性

使用游标获取结果集时是否返回游标名。

参数说明

- ON|TRUE：返回游标名
- 其它值：不返回游标名

示例

- 设置不返回游标名

```
SET RETURN_CURSOR_ID to 'false';
```

- 显示参数状态

```
SHOW RETURN_CURSOR_ID;
```

> 说明
>
> 默认 false 直接返回结果集。

19.4.18 RETURN_ROWID

功能特性

查询是否默认返回 ROWID 开关。

参数说明

- ON|TRUE：返回结果集的同时返回 ROWID 信息
- 其它值：返回结果集的同时不返回 ROWID 信息

示例

- 设置返回结果集的同时不返回 ROWID 信息

```
    SET RETURN_ROWIDd TO 'false';
```

- 显示参数状态

```
    SHOW RETURN_ROWID;
```

> 说明
>
> 默认 false 不返回。

19.4.19 RETURN_SCHEMA

功能特性

查询是否返回对象所属的模式名信息。

参数说明

- ON|TRUE：返回对象所属的模式名信息
- 其它值：不返回对象所属的模式名信息

示例

- 设置不返回对象所属的模式名信息

```
SET RETURN_SCHEMA TO 'false';
```

- 显示参数状态

```
SHOW RETURN_SCHEMA;
```

> **说明**
>
> 默认 false 不返回。

19.4.20 SESSION_USER

功能特性

查看和切换连接会话当前用户。

示例

- 查看当前用户

```
SHOW SESSION_USER;
```

- 切换当前用户

```
SET SESSION_USER TO username;
SET SESSION AUTHORIZATION username;
```

> **注意**
>
> - 任何用户都不可切换到安全管理员（SYSSSO）和审计管理员（SYSAUDITOR）。
> - 登录用户具备 DBA 权限或是系统管理员、安全管理员和审计管理员之一时，才能进行切换当前用户。
> - 安全管理员（SYSSSO）仅支持切换到安全员，切换后不可再切换回安全管理员（SYSSSO）。
> - 审计管理员（SYSSSO）仅支持切换到审计员，切换后不可再切换回审计管理员（SYSSSO）。
> - 切换仅改变当前用户（sys_session.curr_user_id 和 curr_user_name），登录用户不发生变化（sys_session.user_id 和 user_name）。

19.4.21 STRICT_COMMIT

功能特性

客户端连接会话上非只读事务严格提交开关。严格提交事务模型下事务提交需等候所有事务的所有 redo 日志落盘写实。

参数说明

- ON|TRUE：严格提交
- OFF|FALSE：非严格提交

示例

- 设置提交模式

```
SET STRICT_COMMIT ON;
```

- 显示参数状态

```
SHOW STRICT_COMMIT;
```

> 说明
>
> 该参数在驱动程序缺省未指定的情况下，由系统全局参数 strictly_commit 决定；若该参数已经由驱动程序明确指定，则会覆盖系统全局参数 strictly_commit 给定的值。

19.4.22 TIME_FORMAT

功能特性

客户端连接会话上的时间值格式，该参数决定发往客户端的时间值类型数据的显示格式。

- TO_CHAR，TO_DATE 函数不指定时间格式时使用会话时间格式。
- 字符常量赋值各时间类型。
- 字符常量和时间类型比较。

示例

- 设置时间格式

```
SET TIME_FORMAT TO 'YYYY-MM-DD HH24:MI:SS';
```

- 显示时间格式

```
SHOW TIME_FORMAT;
```

> 说明
>
> 默认使用 def_timefmt（”YYYY-MM-DD HH24:MI:SS”）格式。

19.4.23 TRANS_READONLY

功能特性

当前会话是否只执行只读事务。

参数说明

- TRUE：是
- FALSE：否

示例

- 设置当前会话只执行只读事务

```
SET TRANS_READONLY TO TRUE;
```

- 显示参数状态

```
SHOW TRANS_READONLY;
```

> 说明
>
> 默认为”false”。

19.4.24 USER

功能特性

指定新建连接会话的用户名，通过用户名查找其系统内唯一值身份标识（用户 ID），不存在则登录失败。

20 系统函数

为了方便用户能够快速对数据进行简单的运算，虚谷数据库向用户提供了大量的系统级函数，用户只需填充相应的参数即可获得运算后的结果。详细信息请参见《系统函数参考》。

21 关键字

21.1 概述

- 在数据库中，关键字是 SQL 中有实际意义的词，它们用于表示特定的使用功能。为防止歧义，一般不推荐使用关键字作为对象名或集合名等。
- 关键字分为 **保留字和非保留字**。其中保留字是数据库内部强制不能使用的字，例如 CREATE，在数据库的 DDL 语句中用于定义对象创建的操作，如果使用这个关键字作为表对象名，就会产生歧义，所以数据库一般会报错。非保留字虽不推荐使用，但没有强制不允许使用。
- 如果某些保留字确实需要被使用，可以通过特殊的处理（添加双引号）才能作为表名或列名等标识符。非保留关键字，不加双引号即可作为标识符。
- 对于某些保留字有兼容性需求，所以在数据库内部虽然被定义为保留字，但仍允许用户使用。
- 非保留字是作为将来数据库可能会被用到的功能预留。

> ⚠ 注意
>
> 虚谷数据库不推荐在 SQL 中使用保留关键字。

类型	说明	使用范围
保留关键字	不能用作用户定义的变量。在加双引号的情况下可以使用，但不推荐	不能使用
非保留关键字	可以作为用户定义的变量，但不推荐	用户定义变量

21.2 A

虚谷	类别	作为表名	作为列名	作为列别名
ABORT	非保留	×	×	√
ABOVE	保留	×	×	×
ABSOLUTE	非保留	√	√	√
ACCESS	非保留	√	√	√
ACCOUNT	非保留	√	√	√
ACTION	非保留	√	√	√
ADD	非保留	√	√	√
AFTER	非保留	√	√	√
AGGREGATE	非保留	√	√	√
ALL	保留	×	×	×
ALL_ROWS	非保留	√	√	√
ALTER	保留	√	√	√
ANALYSE	非保留	×	×	×
ANALYZE	保留	×	×	×
AND	保留	×	×	√
ANY	保留	×	×	√
AOVERLAPS	保留	×	×	×

接下页

虚谷	类别	作为表名	作为列名	作为列别名
APPEND	非保留	√	√	√
ARCHIVELOG	非保留	√	√	√
ARE	非保留	√	√	√
ARRAY	保留	√	√	√
AS	保留	×	×	×
ASC	保留	×	×	√
AT	非保留	√	√	√
AUDIT	保留	×	×	×
AUDITOR	非保留	√	√	√
AUTHID	非保留	×	×	×
AUTHORIZATION	非保留	√	√	√
AUTO	非保留	×	×	×

21.3 B

虚谷	类别	作为表名	作为列名	作为列别名
BACKUP	非保留	×	×	×
BACKWARD	非保留	√	√	√
BADFILE	非保留	√	√	√
BCONTAINS	保留	×	×	×
BEFORE	非保留	√	√	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
BEGIN	保留	×	×	×
BETWEEN	保留	×	×	√
BINARY	保留	×	×	√
BINTERSECTS	保留	×	×	×
BIT	保留	×	×	√
BLOCK	非保留	×	×	×
BLOCKS	非保留	√	√	√
BODY	保留	√	√	√
BOTH	保留	×	×	√
BOUND	非保留	×	×	√
BOVERLAPS	保留	×	×	×
BREAK	非保留	×	×	√
BUFFER_POOL	非保留	×	×	×
BUILD	非保留	√	√	√
BULK	保留	×	×	×
BWITHIN	保留	×	×	×
BY	保留	√	√	√

21.4 C

虚谷	类别	作为表名	作为列名	作为列别名
CACHE	非保留	×	×	×
CALL	非保留	×	×	×
CASCADE	保留	×	×	×
CASE	保留	×	×	√
CAST	非保留	×	×	√
CATCH	非保留	×	×	×
CATEGORY	非保留	√	√	√
CHAIN	非保留	√	√	√
CHAR	非保留	×	×	√
CHARACTER	非保留	×	×	√
CHARACTERISTICS	非保留	√	√	√
CHECK	保留	×	×	√
CHECKPOINT	保留	×	×	×
CHOOSE	非保留	√	√	√
CHUNK	非保留	×	×	×
CLOSE	保留	×	×	×
CLUSTER	保留	×	×	√
COALESCE	非保留	×	×	√
COLLATE	非保留	×	×	√
COLLECT	非保留	×	×	×

接下页

虚谷	类别	作为表名	作为列名	作为列别名
COLUMN	保留	×	×	√
COLUMNS	非保留	√	√	√
COMMENT	非保留	×	×	√
COMMIT	保留	×	×	×
COMMITTED	非保留	√	√	√
COMPLETE	非保留	×	×	√
COMPRESS	保留	×	×	×
COMPUTE	非保留	×	×	×
CONNECT	保留	×	×	×
CONNECT_NODES	非保留	√	√	√
CONSTANT	保留	×	×	×
CONSTRAINT	非保留	×	×	√
CONSTRAINTS	非保留	√	√	√
CONSTRUCTOR	保留	×	×	×
CONTAINS	非保留	×	×	×
CONTEXT	非保留	√	√	√
CONTINUE	保留	×	×	×
COPY	非保留	×	×	√
CORRESPONDING	非保留	×	×	√
CPU_PER_CALL	非保留	√	√	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
CPU_PER_SESSION	非保留	√	√	√
CREATE	保留	√	√	√
CREATEDB	非保留	√	√	√
CREATEUSER	非保留	√	√	√
CROSS	非保留	×	×	√
CROSSES	保留	×	×	×
CUBE	非保留	×	×	√
CURRENT	非保留	×	×	√
CURSOR	保留	×	×	×
CURSOR_QUOTA	非保留	×	×	×
CYCLE	非保留	√	√	√

21.5 D

虚谷	类别	作为表名	作为列名	作为列别名
DATABASE	非保留	√	√	√
DATAFILE	非保留	√	√	√
DATE	保留	×	√	√
DATETIME	保留	×	×	×
DAY	非保留	×	√	√
DBA	非保留	√	√	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
DEALLOCATE	非保留	×	×	×
DEC	保留	×	×	√
DECIMAL	保留	×	×	√
DECLARE	保留	×	×	×
DECODE	保留	×	×	×
DECRYPT	非保留	√	√	√
DEFAULT	保留	×	×	√
DEFERRABLE	非保留	×	×	√
DEFERRED	非保留	√	√	√
DELETE	保留	√	√	√
DELIMITED	非保留	√	√	√
DELIMITERS	非保留	√	√	√
DEMAND	非保留	√	√	√
DESC	保留	×	×	√
DESCRIBE	非保留	√	√	√
DETERMINISTIC	非保留	×	×	×
DIR	非保留	√	√	√
DISABLE	非保留	√	√	√
DISASSEMBLE	非保留	×	×	×
DISCORDFILE	非保留	√	√	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
DISJOINT	保留	×	×	×
DISTINCT	保留	×	×	√
DO	非保留	×	×	√
DOMAIN	非保留	√	√	√
DOUBLE	保留	×	×	×
DRIVEN	非保留	×	×	√
DROP	保留	×	×	×

21.6 E

虚谷	类别	作为表名	作为列名	作为列别名
EACH	非保留	√	√	√
ELEMENT	非保留	√	√	√
ELSE	保留	×	×	√
ELSEIF	保留	×	×	×
ELSIF	保留	×	×	×
ENABLE	非保留	√	√	√
ENCODING	非保留	√	√	√
ENCRYPT	非保留	√	√	√
ENCRYPTOR	非保留	√	√	√
END	非保留	×	×	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
ENDCASE	保留	×	×	×
ENDFOR	保留	×	×	×
ENDIF	保留	×	×	×
ENDLOOP	保留	×	×	×
EQUALS	保留	×	×	×
ESCAPE	非保留	√	√	√
EVERY	非保留	×	×	×
EXCEPT	非保留	×	×	√
EXCEPTION	保留	×	×	×
EXCEPTIONS	非保留	×	×	×
EXCEPTION_INIT	非保留	×	×	×
EXCLUSIVE	非保留	√	√	√
EXEC	非保留	√	√	√
EXECUTE	非保留	√	√	√
EXISTS	保留	×	×	√
EXIT	保留	×	×	×
EXPIRE	非保留	×	×	×
EXPLAIN	非保留	×	×	√
EXPORT	非保留	×	×	×
EXTEND	非保留	√	√	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
EXTERNAL	非保留	√	√	√
EXTRACT	保留	×	×	×

21.7 F

虚谷	类别	作为表名	作为列名	作为列别名
FAILED_LOGIN_ATTEMPTS	非保留	×	×	×
FALSE	保留	×	×	√
FAST	非保留	×	×	√
FETCH	保留	×	×	×
FIELD	非保留	√	√	√
FIELDS	非保留	√	√	√
FILTER	非保留	√	√	√
FINAL	非保留	×	×	×
FINALLY	非保留	×	×	√
FIRST	非保留	√	√	√
FIRST_ROWS	非保留	√	√	√
FLASHBACK	非保留	×	×	×
FLOAT	保留	×	×	√
FOLLOWING	非保留	×	×	×

接下页

虚谷	类别	作为表名	作为列名	作为列别名
FOR	保留	×	×	×
FORALL	非保留	×	×	×
FORCE	非保留	√	√	√
FOREIGN	非保留	×	×	√
FORWARD	非保留	√	√	√
FOUND	非保留	√	√	√
FREELIST	非保留	×	×	×
FREELISTS	非保留	×	×	×
FROM	保留	×	×	√
FULL	保留	×	×	×
FUNCTION	保留	×	×	√

21.8 G

虚谷	类别	作为表名	作为列名	作为列别名
G	非保留	√	√	√
GENERATED	非保留	√	√	√
GET	非保留	×	×	√
GLOBAL	非保留	×	×	√
GOTO	非保留	×	×	×
GRANT	保留	√	√	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
GREATEST	非保留	×	×	√
GROUP	保留	×	×	√
GROUPING	非保留	×	√	√
GROUPS	非保留	×	×	×

21.9 H

虚谷	类别	作为表名	作为列名	作为列别名
HANDLER	非保留	√	√	√
HASH	非保留	√	√	√
HAVING	保留	×	×	√
HEAP	非保留	√	√	√
HIDE	非保留	×	×	×
HINT	非保留	×	×	×
HOTSPOT	非保留	√	√	√
HOUR	非保留	×	√	√

21.10 I

虚谷	类别	作为表名	作为列名	作为列别名
IDENTIFIED	非保留	×	×	×
IDENTIFIER	非保留	√	√	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
IDENTITY	非保留	√	√	√
IF	保留	×	×	×
IGNORE	非保留	√	√	√
ILIKE	非保留	×	×	√
IMMEDIATE	保留	×	×	×
IMPORT	非保留	×	×	×
IN	保留	×	×	√
INCLUDE	非保留	√	√	√
INCREMENT	非保留	√	√	√
INDEX	保留	√	√	√
INDEXTYPE	非保留	√	√	√
INDEX_ASC	非保留	√	√	√
INDEX_DESC	非保留	√	√	√
INDEX_FSS	非保留	√	√	√
INDEX_JOIN	非保留	√	√	√
INDICATOR	非保留	×	×	√
INDICES	非保留	×	×	×
INHERITS	非保留	√	√	√
INIT	非保留	×	×	×
INITIAL	非保留	×	×	×

接下页

虚谷	类别	作为表名	作为列名	作为列别名
INITIALLY	保留	×	×	√
INITRANS	非保留	×	×	×
INNER	非保留	×	×	√
INOUT	非保留	×	×	√
INSENSITIVE	非保留	×	×	×
INSERT	非保留	√	√	√
INSTANTIABLE	非保留	×	×	×
INSTEAD	非保留	√	√	√
INTERSECT	保留	×	×	√
INTERSECTS	保留	×	×	×
INTERVAL	保留	×	×	×
INTO	保留	×	×	×
IO	非保留	√	√	√
IS	保留	×	×	√
ISNULL	非保留	×	×	√
ISOLATION	非保留	√	√	√
ISOPEN	非保留	√	√	√

21.11 J

虚谷	类别	作为表名	作为列名	作为列别名
JOB	非保留	√	√	√
JOIN	非保留	×	×	√

21.12 K

虚谷	类别	作为表名	作为列名	作为列别名
K	非保留	√	√	√
KEEP	非保留	√	√	√
KEY	非保留	√	√	√
KEYSET	非保留	×	×	√

21.13 L

虚谷	类别	作为表名	作为列名	作为列别名
LABEL	非保留	×	√	√
LANGUAGE	保留	×	√	√
LAST	非保留	√	√	√
LEADING	非保留	×	×	√
LEAST	非保留	×	×	√
LEAVE	保留	×	×	×
LEFT	非保留	×	×	√
LEFTOF	保留	×	×	×

接下页

虚谷	类别	作为表名	作为列名	作为列别名
LENGTH	非保留	√	√	√
LESS	非保留	√	√	√
LEVEL	非保留	√	√	√
LEVELS	非保留	√	√	√
LEXER	保留	×	×	×
LIBRARY	非保留	√	√	√
LIKE	保留	×	×	√
LIMIT	保留	×	×	√
LINK	保留	×	√	√
LIST	非保留	√	√	√
LISTEN	非保留	×	×	√
LOAD	非保留	×	×	√
LOB	非保留	×	×	×
LOCAL	非保留	×	×	√
LOCATION	非保留	√	√	√
LOCATOR	非保留	√	√	√
LOCK	保留	×	×	√
LOGFILE	非保留	√	√	√
LOGGING	非保留	×	×	×
LOGIN	非保留	√	√	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
LOGOFF	非保留	×	×	×
LOGON	非保留	×	×	×
LOGOUT	非保留	√	√	√
LOOP	保留	×	×	×
LOVERLAPS	保留	×	×	×

21.14 M

虚谷	类别	作为表名	作为列名	作为列别名
M	非保留	√	√	√
MATCH	非保留	√	√	√
MATCHED	非保留	×	×	×
MATERIALIZED	非保留	√	√	√
MAX	非保留	√	√	√
MAXEXTENTS	非保留	×	×	×
MAXSIZE	非保留	√	√	√
MAXTRANS	非保留	×	×	×
MAXVALUE	非保留	√	√	√
MAXVALUES	保留	×	×	×
MAX_CONNECT_TIME	非保留	√	√	√
MAX_IDLE_TIME	非保留	√	√	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
MAX_STORE_NUM	非保留	√	√	√
MEMBER	保留	×	×	×
MEMORY	非保留	√	√	√
MERGE	非保留	√	√	√
MINEXTENTS	非保留	×	×	×
MINUS	非保留	×	×	√
MINUTE	非保留	×	√	√
MINVALUE	非保留	√	√	√
MISSING	非保留	√	√	√
MODE	非保留	√	√	√
MODIFY	非保留	×	×	×
MONTH	非保留	×	√	√
MOVEMENT	非保留	√	√	√

21.15 N

虚谷	类别	作为表名	作为列名	作为列别名
NAME	非保留	√	√	√
NAMES	非保留	√	√	√
NATIONAL	非保留	√	√	√
NATURAL	非保留	×	×	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
NCHAR	非保留	×	×	√
NESTED	非保留	×	×	×
NEW	非保留	√	√	√
NEWLINE	保留	×	×	×
NEXT	非保留	√	√	√
NO	非保留	√	√	√
NOAPPEND	非保留	√	√	√
NOARCHIVELOG	非保留	√	√	√
NOAUDIT	非保留	×	×	×
NOCACHE	非保留	×	×	√
NOCOMPRESS	非保留	×	×	×
NOCREATEDB	非保留	√	√	√
NOCREATEUSER	非保留	√	√	√
NOCYCLE	非保留	×	×	√
NODE	非保留	×	×	×
NOFORCE	非保留	√	√	√
NOFOUND	非保留	√	√	√
NOINDEX	非保留	√	√	√
NOLOGGING	非保留	×	×	×
NONE	非保留	√	√	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
NOORDER	非保留	×	×	√
NOPARALLEL	保留	×	×	×
NOT	保留	×	×	√
NOTFOUND	非保留	√	√	√
NOTHING	非保留	√	√	√
NOTIFY	非保留	√	√	√
NOTNULL	非保留	×	×	√
NOVALIDATE	非保留	×	×	×
NOWAIT	非保留	×	×	√
NULL	保留	×	×	√
NULLIF	非保留	×	×	√
NULLS	非保留	√	√	√
NUMBER	保留	×	×	√
NUMERIC	保留	×	×	√
NVARCHAR	保留	×	×	×
NVARCHAR2	保留	×	×	×
NVL	保留	×	×	×
NVL2	保留	×	×	×

21.16 O

虚谷	类别	作为表名	作为列名	作为列别名
OBJECT	非保留	√	√	√
OF	非保留	×	×	×
OFF	非保留	×	×	√
OFFLINE	非保留	√	√	√
OFFSET	保留	√	√	√
OIDINDEX	非保留	√	√	√
OIDS	非保留	√	√	√
OLD	非保留	√	√	√
ON	保留	×	×	×
ONLINE	非保留	√	√	√
ONLY	非保留	×	×	√
OPEN	非保留	×	×	×
OPERATOR	非保留	√	√	√
OPTION	非保留	√	√	√
OR	保留	×	×	√
ORDER	保留	×	×	√
ORDERD	非保留	√	√	√
ORGANIZATION	非保留	√	√	√
OTHERVALUES	保留	×	×	×
OUT	保留	×	×	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
OUTER	非保留	×	×	√
OVER	保留	×	×	×
OVERLAPS	非保留	√	√	√
OWNER	非保留	√	√	√

21.17 P

虚谷	类别	作为表名	作为列名	作为列别名
PACKAGE	非保留	√	√	√
PARALLEL	保留	√	√	√
PARAMETERS	非保留	√	√	√
PARTIAL	非保留	×	×	×
PARTITION	保留	×	×	×
PARTITIONS	非保留	√	√	√
PASSWORD	非保留	√	√	√
PASSWORD_LIFE_PERIOD	非保留	×	×	×
PASSWORD_LOCK_TIME	非保留	×	×	×
PCTFREE	保留	√	√	√
PCTINCREASE	非保留	×	×	×
PCTUSED	非保留	√	√	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
PCTVERSION	非保留	×	×	×
PERIOD	非保留	×	√	√
POLICY	保留	×	×	×
PRAGMA	保留	×	×	×
PREBUILT	非保留	√	√	√
PRECEDING	非保留	×	×	×
PRECISION	非保留	×	√	√
PREPARE	非保留	×	×	×
PRESERVE	非保留	√	√	√
PRIMARY	保留	×	×	√
PRIOR	非保留	×	×	√
PRIORITY	非保留	√	√	√
PRIVATE_SGA	非保留	√	√	√
PRIVILEGES	非保留	√	√	√
PROCEDURAL	非保留	√	√	√
PROCEDURE	保留	×	×	×
PROFILE	非保留	√	√	√
PROTECTED	非保留	√	√	√
PUBLIC	非保留	×	×	√

21.18 Q

虚谷	类别	作为表名	作为列名	作为列别名
QUERY	非保留	√	√	√
QUOTA	非保留	√	√	√

21.19 R

虚谷	类别	作为表名	作为列名	作为列别名
RAISE	保留	×	×	×
RANGE	非保留	√	√	√
RAW	保留	×	×	×
READ	非保留	√	√	√
READS	非保留	√	√	√
READS_PER_CALL	非保留	√	√	√
READS_PER_SESSION	非保留	√	√	√
REBUILD	非保留	√	√	√
RECOMPILE	非保留	×	×	√
RECORD	非保留	×	√	√
RECORDS	非保留	√	√	√
RECYCLE	非保留	×	×	×
REDUCED	非保留	√	√	√
REF	非保留	√	√	√
REFERENCES	非保留	×	×	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
REFERENCING	非保留	√	√	√
REFRESH	非保留	√	√	√
REINDEX	非保留	√	√	√
RELATIVE	非保留	√	√	√
RENAME	保留	√	√	√
REPEATABLE	非保留	√	√	√
REPLACE	非保留	√	√	√
REPLICATION	非保留	×	×	√
RESOURCE	非保留	√	√	√
RESTART	非保留	×	×	√
RESTORE	非保留	×	×	×
RESTRICT	保留	×	×	×
RESULT	非保留	√	√	√
RESULT_CACHE	非保留	√	√	√
RETURN	保留	×	×	×
RETURNING	保留	×	×	×
REVERSE	保留	√	√	√
REVOKE	非保留	√	√	√
REWRITE	非保留	√	√	√
RIGHT	非保留	×	×	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
RIGHTOF	保留	×	×	×
ROLE	非保留	√	√	√
ROLLBACK	保留	×	×	×
ROLLUP	非保留	×	×	√
ROVERLAPS	保留	×	×	×
ROW	非保留	√	√	√
ROWCOUNT	非保留	√	√	√
ROWID	非保留	√	√	√
ROWS	非保留	√	√	√
ROWTYPE	非保留	×	×	×
RULE	非保留	√	√	√
RUN	非保留	√	√	√

21.20 S

虚谷	类别	作为表名	作为列名	作为列别名
SAVEPOINT	非保留	×	×	×
SCHEMA	非保留	√	√	√
SCROLL	非保留	×	×	×
SECOND	非保留	×	√	√
SEGMENT	非保留	×	√	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
SELECT	保留	×	×	√
SELF	非保留	×	×	√
SEQUENCE	非保留	√	√	√
SERIALIZABLE	非保留	√	√	√
SESSION	非保留	√	√	√
SESSION_PER_USER	非保留	√	√	√
SET	保留	×	×	×
SETOF	非保留	×	×	√
SETS	非保留	×	√	√
SHARE	非保留	√	√	√
SHOW	非保留	×	×	√
SHUTDOWN	非保留	×	×	√
SIBLINGS	非保留	√	√	√
SIZE	非保留	×	×	×
SLOW	非保留	√	√	√
SNAPSHOT	非保留	×	×	×
SOME	非保留	×	×	√
SPATIAL	非保留	√	√	√
SPLIT	非保留	√	√	√
SSO	非保留	√	√	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
STANDBY	非保留	√	√	√
START	保留	×	×	×
STATEMENT	非保留	√	√	√
STATIC	保留	×	×	×
STATISTICS	非保留	×	×	×
STEP	非保留	×	√	√
STOP	非保留	√	√	√
STORAGE	非保留	√	√	√
STORE	非保留	×	×	×
STORE_NODES	非保留	√	√	√
STREAM	非保留	√	√	√
SUBPARTITION	保留	×	×	×
SUBPARTITIONS	非保留	√	√	√
SUBTYPE	保留	×	√	√
SUCCESSFUL	非保留	×	×	×
SYNONYM	非保留	×	×	×
SYSTEM	非保留	√	√	√

21.21 T

虚谷	类别	作为表名	作为列名	作为列别名
TABLE	保留	×	×	×
TABLESPACE	非保留	×	×	×
TEMP	非保留	√	√	√
TEMPLATE	非保留	√	√	√
TEMPORARY	非保留	√	√	√
TEMPSPACE_QUOTA	非保留	√	√	√
TERMINATED	非保留	√	√	√
THAN	非保留	√	√	√
THEN	保留	×	×	×
THROW	非保留	×	×	×
TIME	保留	×	√	√
TIMESTAMP	保留	×	√	√
TO	保留	×	×	×
TOP	非保留	×	×	√
TOPOVERLAPS	非保留	×	×	×
TOTAL_RESOURCE_LIMIT	非保留	√	√	√
TOUCHES	保留	×	×	×
TRACE	非保留	×	×	√
TRAILING	保留	×	×	√
TRAN	非保留	×	×	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
TRANSACTION	非保留	×	×	√
TRIGGER	保留	√	√	√
TRUE	非保留	×	×	√
TRUNCATE	非保留	×	×	×
TRUSTED	非保留	√	√	√
TRY	非保留	×	×	×
TYPE	保留	×	√	√

21.22 U

虚谷	类别	作为表名	作为列名	作为列别名
UNBOUNDED	保留	×	×	×
UNDER	保留	×	×	×
UNDO	非保留	√	√	√
UNIFORM	非保留	√	√	√
UNION	保留	×	×	√
UNIQUE	保留	×	×	√
UNLIMITED	非保留	√	√	√
UNLISTEN	非保留	×	×	×
UNLOCK	非保留	×	×	×
UNPROTECTED	非保留	√	√	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
UNTIL	非保留	√	√	√
UOVERLAPS	保留	×	×	×
UPDATE	保留	×	×	×
USE	保留	×	×	×
USER	保留	×	×	×
USE_HASH	非保留	√	√	√
USING	非保留	×	×	√

21.23 V

虚谷	类别	作为表名	作为列名	作为列别名
VACUUM	非保留	×	×	√
VALID	非保留	√	√	√
VALIDATE	非保留	×	×	×
VALUE	非保留	√	√	√
VALUES	保留	√	√	√
VARCHAR	保留	×	×	√
VARCHAR2	保留	×	×	×
VARRAY	非保留	×	×	×
VARYING	保留	×	×	×
VERBOSE	保留	×	×	√

接下页

虚谷	类别	作为表名	作为列名	作为列别名
VERSION	非保留	√	√	√
VIEW	保留	√	√	√
VOCABLE	非保留	√	√	√

21.24 W

虚谷	类别	作为表名	作为列名	作为列别名
WAIT	非保留	×	×	√
WHEN	非保留	×	×	√
WHENEVER	非保留	×	×	×
WHERE	保留	×	×	×
WHILE	保留	×	×	×
WITH	保留	×	×	×
WITHIN	保留	×	×	×
WITHOUT	保留	×	×	×
WORK	非保留	×	×	√
WRITE	非保留	×	×	×

21.25 X

虚谷	类别	作为表名	作为列名	作为列别名
XML	非保留	√	√	√

21.26 Y

虚谷	类别	作为表名	作为列名	作为列别名
YEAR	非保留	×	√	√

21.27 Z

虚谷	类别	作为表名	作为列名	作为列别名
ZONE	非保留	√	√	√

22 查询

22.1 查询语法

22.1.1 顶层查询语句 selectstmt

语法格式

```
selectstmt::=
select_no_parens
|   select_with_parens
|   with_clauses select_no_parens
```

参数说明

- select_no_parens：没有外部括号的简单选择语句。
- select_with_parens：带有外部括号的选择语句，可以用于嵌套查询。
- with_clauses select_no_parens：包含 WITH 子句的选择语句。

22.1.2 select_no_parens

语法格式

```
select_no_parens ::=
simple_select
|   {simple_select | select_with_parens} sort_clause [NULLS FIRST
    | NULLS LAST] opt_for_update_clause opt_select_limit
|   {simple_select | select_with_parens} sort_clause
   opt_for_update_clause opt_select_limit
|   {simple_select | select_with_parens} for_update_clause
   opt_select_limit
|   {simple_select | select_with_parens} select_limit
|   {simple_select | select_with_parens} FOR SNAPSHOT OF ICONST
   AFTER ICONST START AT ICONST opt_select_limit
```

参数说明

- simple_select：简单的选择语句。
- sort_clause：对结果集进行排序。
- opt_for_update_clause、for_update_clause：在查询时对结果集中的行进行锁定、只读。
- opt_select_limit、select_limit：查询中用于限制返回行数。

- FOR SNAPSHOT OF ICONST AFTER ICONST START AT ICONST opt_select_limit：基于时间点的快照选择语句。

22.1.3 简单查询 simple_select

语法格式

```
simple_select ::=
base_select
|   set_operation

base_select ::=
SELECT [HINT_TEXT] [TOP ICONST] [DISTINCT | ALL] target_list
opt_bulk opt_into_list opt_from_clause opt_where_clause
[opt_connect_by] opt_group_clause opt_having_clause

set_operation ::=
{simple_select | select_with_parens} UNION [ALL | DISTINCT] {
   simple_select | select_with_parens}
|   {simple_select | select_with_parens} INTERSECT [ALL | DISTINCT
   ] {simple_select | select_with_parens}
|   {simple_select | select_with_parens} EXCEPT [ALL | DISTINCT] {
   simple_select | select_with_parens}
|   {simple_select | select_with_parens} MINUS [ALL | DISTINCT] {
   simple_select | select_with_parens}
```

参数说明

- base_select：包含所有可选组件的基础 SELECT 语句。
  - SELECT：核心关键字，用于指定要检索的数据。
  - [HINT_TEXT]：可选部分，优化访问路径。
  - [TOP ICONST]：可选部分，用来限制返回结果集中的记录条数，同 LIMIT ICONST 一致，不可与 LIMIT 同时使用。
  - [DISTINCT | ALL]：可选部分，排除重复的记录。
  - target_list：必填部分，指定要选择的列或表达式。
  - opt_bulk：可选部分，用于批量操作。
  - opt_into_list：可选部分，用于将查询结果插入到变量或表中。
  - opt_from_clause：必填部分，指定查询的数据来源。
  - opt_where_clause：可选部分，用于指定过滤条件。
  - [opt_connect_by]：可选部分，用于层次查询。

  - opt_group_clause：可选部分，用于对查询结果进行分组。
  - opt_having_clause：可选部分，用于对分组后的结果进行过滤。

- set_operation：多个 SELECT 语句之间的集合操作，如并集（UNION）、交集（INTERSECT）、差集（EXCEPT 或 MINUS）。

22.1.4 排序 sort_clause

语法格式

```
sort_clause::=
ORDER BY sortby[,sortby]...

sortby::=
b_expr [USING ROp | ASC | DESC] [NULLS FIRST | NULLS LAST]

ROp::=
<
|    >
|    =
|    <=
|    >=
|    <>
```

参数说明

- sort_clause：包含一个或多个 sortby 项的 ORDER BY 子句，可分为降序和升序，若无排序子句则默认为升序。
- sortby：表达式及其排序方向（ASC 或 DESC），以及 NULL 值的处理方式（NULLS FIRST 或 NULLS LAST）。
- ROp：比较运算符。

22.1.5 锁定 opt_for_update_clause

语法格式

```
opt_for_update_clause ::=
/* EMPTY */
|    for_update_clause

for_update_clause ::=
FOR UPDATE OF name_space [, name_space]...
|    FOR READ ONLY
|    FOR UPDATE
```

参数说明

- opt_for_update_clause：可选 FOR UPDATE 子句，允许为空。

- for_update_clause：明确指定要锁定的对象，或只读模式。

22.1.6 结果限制 opt_select_limit

语法格式

```
opt_select_limit ::=
/* EMPTY */
|   select_limit

select_limit ::=
LIMIT {ICONST | Param | :ICONST} ',' {ICONST | ALL | Param | :
   ICONST}
|   LIMIT {ICONST | ALL | Param | :ICONST} OFFSET {ICONST | Param
    | :ICONST}
|   LIMIT {ICONST | ALL | Param | :ICONST}
```

参数说明

- opt_select_limit：可选的结果限制子句，允许为空。
- select_limit：包含 LIMIT 和 OFFSET 的具体格式，数据分页限制子句，LIMIT 限制数据条数，OFFSET 限制数据开始位置。

22.1.7 目标列表 tar get_list

语法格式

```
target_list ::= target_el [, target_el]...

target_el ::=
b_expr AS ColLabel
|   b_expr ColId
|   DEFAULT
|   b_expr
|   ident . *
|   *
|   rop_bool_expr AS ColLabel
|   rop_bool_expr ColId
|   bool_expr1
|   '(' rop_bool_expr ')'

rop_bool_expr::=
b_expr ROp b_expr
|   bool_expr1 ROp b_expr
|   b_expr ROp bool_expr1
|   bool_expr1 ROp bool_expr1

bool_expr1::=
'(' bool_expr AND bool_expr ')'
|   '(' bool_expr OR bool_expr ')'
|   '(' NOT bool_expr ')'
|   '(' bool_expr1 ')'
```

参数说明

- target_list：由多个 target_el 组成的列表。
- target_el：单个目标元素，可以是表达式、别名、通配符 *、函数调用等。
- rop_bool_expr：使用比较运算符连接的两个表达式。
- b_expr：表达式的具体形式，如算术表达式、字符串表达式等。
- bool_expr1：括号内的 bool_expr 逻辑运算形式。

22.1.8 批量处理 opt_bulk

语法格式

```
opt_bulk::=
/*EMPTY*/
|    BULK COLLECT
```

参数说明

- /*EMPTY*/：BULK COLLECT 子句为空，即不使用 BULK COLLECT，查询结果将逐行处理。
- BULK COLLECT：批量处理，将查询结果一次性加载到集合中，而不是逐行处理。

22.1.9 插入 opt_into_list

语法格式

```
opt_into_list::=
INTO ident[,ident]...
|    /*empty*/
```

参数说明

- INTO ident[,ident]...：将查询结果插入到一个或多个目标标识符（ident）中。
- /*empty*/：INTO 子句为空，即不使用 INTO 子句。

22.1.10 参数和标识符 ident

语法格式

```
ident::=
: ColId
|    : ICONST
|    ColId
|    COLUMN
|    SELF
```

```
|    ISNULL
|    NOTNULL
|    ident . ColLabel
|    ident (func_params)
|    OVERLAPS (func_params)
|    ident (ALL expr_list)
|    ident (DISTINCT b_expr[,b_expr]...)
|    ident (*)
|    EXTRACT (YEAR FROM b_expr)
|    EXTRACT (MONTH FROM b_expr)
|    EXTRACT (DAY FROM b_expr)
|    EXTRACT (HOUR FROM b_expr)
|    EXTRACT (MINUTE FROM b_expr)
|    EXTRACT (SECOND FROM b_expr)
|    ident (position_list)
|    ident (b_expr FROM b_expr FOR b_expr)
|    ident (BOTH {b_expr FROM b_expr | FROM b_expr})
|    ident (LEADING {b_expr FROM b_expr | FROM b_expr})
|    ident (TRAILING {b_expr FROM b_expr | FROM b_expr})
|    ident ({b_expr FROM b_expr | FROM b_expr})

expr_list::=
b_expr
|    expr_list USING b_expr

position_list::=
b_expr IN b_expr
```

参数说明

- ident：标识符，可以是列名、常量、关键字、函数调用、表达式等。
- expr_list：表达式列表，后面跟随 USING 关键字和一个布尔表达式，用于指定额外的条件或参数。
- position_list：指定一个值列表、子查询或表，来判断左侧的表达式是否与右侧的任何一个值相匹配。

22.1.11 FROM 子句 opt_from_clause

语法格式

```
opt_from_clause::=
/*EMPTY*/
|    FROM table_ref[,table_ref]...

table_ref::=
relation_expr
|    relation_expr alias_clause
|    select_with_parens
|    select_with_parens alias_clause
|    (relation_expr)
|    (relation_expr) alias_clause
```

```
|    joined_table
|    (joined_table) alias_clause
|    TABLE (c_expr)
|    TABLE (c_expr) alias_clause
|    TABLE select_with_parens
|    TABLE select_with_parens alias_clause

relation_expr ::=
name_space [PARTITION (name_list) | SUBPARTITION (name_list)]
|    name_space '@' name

alias_clause ::=
AS ColId (name_list)
|    AS ColId
|    ColId (name_list)
|    ColId
```

参数说明

- opt_from_clause：可选的 FROM 子句，允许为空。
- table_ref：表引用，可以是表名、子查询、连接表等。
- relation_expr：表名或表的分区引用，可以带有链接名。
- alias_clause：别名子句，可以是简单的别名或带有列别名的复杂别名。

22.1.12 分组 opt_group_clause

语法格式

```
opt_group_clause::=
GROUP BY group_item [,group_item]...
|    /*EMPTY*/

group_item::=
b_expr
|    ROLLUP (group_item [,group_item]...)
|    CUBE (group_item [,group_item]...)
|    GROUPING SETS (group_item [,group_item]...)
|    group_item,group_item [,group_item]...
|    ( )
```

参数说明

- GROUP BY group_item [,group_item]...：表示 GROUP BY 子句后面跟随一个或多个 group_item，每个 group_item 之间用逗号分隔。
- /* EMPTY */：GROUP BY 子句可以被省略，不进行任何分组。
- b_expr：表示一个简单的表达式，通常是列名或基于列的表达式。这是最常见的分组项形式。

- ROLLUP ：GROUP BY 的扩展，n+1 次分组统计，其中 n 为分组字段数，按照从右至左依次递减字段生成分组统计，即对每个分组结果再进行小计和总和。如 ROLLUP(a,b) 的分组为 (a,b)、(a)、() 三种情况，第一次将整体作为条件进行分组小计，第二次对分组统计的最后一个字段删减作为条件再进行分组小计，第三次再次从右至左删减字段作为条件进行分组小计即为无条件将整个表进行统计。
- CUBE：GROUP BY 的扩展，多字段分组时按照聚合字段排列组合进行分组统计并对每一种组合结果进行汇总统计。如 CUBE(a,b) 统计的分组为 (a,b)、(a)、(b)、() 情况。
- GROUPING SETS：GROUP BY 的扩展，无需所有分组字段的排列组合仅返回每个字段的分组小计。如 GROUPING SETS(a,b) 统计的分组为 (a)、(b) 情况。
- group_item,group_item [,group_item]...：多个 group_item 的组合，类似于直接在 GROUP BY 子句中列出多个分组项。
- ( )：空分组项，对整个表进行汇总，不分组。

22.1.13 HAVING 子句 opt_having_clause

语法格式

```
opt_having_clause ::=
HAVING bool_expr
|    /* EMPTY */
```

参数说明

- HAVING bool_expr：表示 HAVING 子句后面跟随一个布尔表达式（bool_expr），该表达式用于筛选分组后的结果。
- /* EMPTY */：HAVING 子句可以被省略，不对分组结果进行任何额外的筛选。

22.1.14 层次查询 opt_connect_by

语法格式

```
opt_connect_by ::=
CONNECT BY bool_expr START WITH bool_expr [KEEP bool_expr]
|    CONNECT BY NOCYCLE bool_expr START WITH bool_expr [KEEP
   bool_expr]
|    START WITH bool_expr CONNECT BY bool_expr [KEEP bool_expr]
|    START WITH bool_expr CONNECT BY NOCYCLE bool_expr [KEEP
   bool_expr]
|    CONNECT BY bool_expr [KEEP bool_expr]
|    CONNECT BY NOCYCLE bool_expr [KEEP bool_expr]
```

参数说明

- CONNECT BY bool_expr：指定层次结构的连接条件，使用一个布尔表达式来定义父节点和子节点之间的关系。
- START WITH bool_expr：指定层次结构的根节点，使用一个布尔表达式来确定层次结构的起点。
- NOCYCLE：防止循环引用。当层次结构中可能存在循环时（即某个节点最终指向自己），使用 NOCYCLE 可以避免无限递归。
- KEEP bool_expr：用于在层次结构中保留特定的值。

22.1.15 select_with_parens

语法格式

```
select_with_parens ::=
(select_no_parens parallel_opt opt_wait)
|    (select_with_parens)
|    (with_clauses select_no_parens)
```

参数说明

select_with_parens：在 SELECT 语句外部添加括号。

22.1.16 并行选项 parallel_opt

语法格式

```
parallel_opt ::=
/* EMPTY */
|    PARALLEL ICONST
```

参数说明

- /* EMPTY */：不设置并行选项。
- PARALLEL ICONST：数据库查询弹射并发数，并发数的值和逻辑 CPU 相关，小于逻辑 CPU 数。

22.1.17 with_clauses

语法格式

```
with_clauses::=
WITH with_name AS select_with_parens
|    with_clauses  , with_name AS select_with_parens
|    WITH ProcDef ;
```

```
with_name::=
ColId [( name_list )]
```

参数说明

- WITH：关键字，用于引入一个或多个公用表表达式。
- with_name：CTE 的名称，用于在后续查询中引用这个临时结果集。它是一个标识符（ColId），可以带有可选的列名列表（name_list）。
- AS select_with_parens：定义 CTE 的内容，即一个带括号的 SELECT 语句。这个 SELECT 语句的结果将作为 CTE 的临时结果集。
- with_clauses：可以包含多个 CTE，每个 CTE 用逗号分隔。多个 CTE 可以相互引用，但不能循环引用。
- ProcDef：扩展语法，表示可以在 WITH 子句中定义过程或函数。

22.1.18 示例

使用 HINT

两种不同的查询执行计划：

- 使用 HINT：/*+INDEX(tb_hint idx_hit)*/
- 不使用 HINT。

根据 EXPLAIN 语句的结果，对目标表 tb_hint 使用 HINT，SQL 语句的执行计划为 BtIdxScan，表示数据库使用了 B 树索引，能够快速查找特定值。不使用 HINT，SQL 语句的执行计划为 SeqScan，表示数据库使用了全表扫描。全表扫描会逐行读取整个表，直到找到符合条件的记录。这种方式在小表或没有合适索引时可能是合理的，但在大表上性能较差。

```
SQL> CREATE TABLE tb_hint(id INT,name VARCHAR(20));

SQL> CREATE INDEX idx_hit ON tb_hint(id);

-- 使用HINT查看SQL的执行计划
SQL> EXPLAIN SELECT /*+INDEX(tb_hint idx_hit)*/* FROM tb_hint WHERE
    id=3;

plan_path |
------------------------------------------------------------
1   BtIdxScan[(1 1) cost=300,result_num=1](table=TB_HINT)(index=
   IDX_HIT)|

-- 默认情况下的SQL执行计划
SQL> EXPLAIN SELECT * FROM tb_hint WHERE id=3;
```

```
plan_path |
-----------------------------------------------------------
1   SeqScan[(1 1) cost=0,result_num=1](table=TB_HINT)|
```

使用 TOP

使用 TOP 2 返回排列最前面的 2 行数据。

```
SQL> CREATE TABLE tb_top(id INT,name VARCHAR(20));

SQL> INSERT INTO tb_top VALUES(1,'one')(2,'two')(3,'three')(4,'four
   ');

SQL> SELECT TOP 2 * FROM tb_top ORDER BY id DESC;

ID | NAME |
-----------------------------------------------------------
4 | four|
3 | three|
```

使用 DISTINCT

id 列中有两个 1，使用 DISTINCT 只返回一个 1，消除查询结果中的重复行。

```
SQL> CREATE TABLE tb_dis(id INT,name VARCHAR(20));

SQL> INSERT INTO tb_dis VALUES(1,'one')(1,'two')(3,'three')(4,'four
   ');

SQL> SELECT DISTINCT id FROM tb_dis;

ID |
-----------------------------------------------------------
1 |
3 |
4 |
```

BULK COLLECT 批量处理方法

使用 BULK COLLECT 将查询结果批量加载到集合中，批量输出。

```
SQL> CREATE TABLE tb_coll(id INT,name VARCHAR(20));

SQL> INSERT INTO tb_coll VALUES(1,'one')(2,'two')(3,'three');

SQL> DECLARE
TYPE v_table IS VARRAY(10) OF VARCHAR;
st v_table;
BEGIN
SELECT name BULK COLLECT INTO st FROM tb_coll ;
FOR i IN st.FIRST..st.LAST
LOOP
SEND_MSG(st(i));
END LOOP;
END;
/
```

```
-- 输出结果:
one
two
three
```

分组统计并过滤

使用 WHERE 关键字查询表中 id > 10 的数据，并对结果进行分组，对分组后的数据再使用 HAVING 关键字过滤 id > 20 的数据并输出结果。

```
SQL> CREATE TABLE tb_grp_hav(id INT,name VARCHAR(20));

SQL> INSERT INTO tb_grp_hav VALUES(1,'a')(1,'b')(20,'c')(20,'d')
   (100,'e')(100,'f')(100,'g');

SQL> SELECT id FROM tb_grp_hav WHERE id > 10 GROUP BY id HAVING id
    > 20;

ID |
-----------------------------------------------------------
100 |
```

指定分区名查询指定分区数据

使用范围分区将表 tb_part_sel 分成了三个部分 vp1、vp2、vp3，查询 vp1 分区的数据。

```
SQL> CREATE TABLE tb_part_sel(id INT,name VARCHAR(20))PARTITION BY
   RANGE(id) PARTITIONS(vp1 VALUES LESS THAN(100),vp2 VALUES LESS
   THAN(1000),vp3 VALUES LESS THAN(20000));

SQL> INSERT INTO tb_part_sel VALUES(1,'one')(100,'two')(10003,'
   three');

SQL> SELECT * FROM tb_part_sel PARTITION(vp1);

ID | NAME |
-----------------------------------------------------------
1 | one|
```

GROUP BY 的三种扩展方式

- ROLLUP(department,gender) 的分组为 (department,gender)、(department)、() 三种情况。
- CUBE(department,gender) 的分组为 (department,gender)、(department)、(gender)、() 情况。
- GROUPING SETS(department,gender) 分组为 (department)、(gender) 情况。

```
SQL> CREATE TABLE tb_grp3(id INT PRIMARY KEY,name VARCHAR(15),
   department VARCHAR(10),salary NUMBER(8, 2),gender VARCHAR(10));
```

```
SQL> INSERT INTO tb_grp3 VALUES (1001, 'John', 'IT', 35000, 'Male')
   ;
INSERT INTO tb_grp3 VALUES (1002, 'Smith', 'HR', 45000, 'Male');
INSERT INTO tb_grp3 VALUES (1003, 'James', 'Finance', 50000, 'Male'
   );
INSERT INTO tb_grp3 VALUES (1004, 'Mike', 'Finance', 50000, 'Male')
   ;
INSERT INTO tb_grp3 VALUES (1005, 'Linda', 'HR', 75000, 'Female');
INSERT INTO tb_grp3 VALUES (1006, 'Anurag', 'IT', 35000, 'Male');
INSERT INTO tb_grp3 VALUES (1007, 'Priyanla', 'HR', 45000, 'Female'
   );
INSERT INTO tb_grp3 VALUES (1008, 'Sambit', 'IT', 55000, 'Female');
INSERT INTO tb_grp3 VALUES (1009, 'Pranaya', 'IT', 57000, 'Female')
   ;
INSERT INTO tb_grp3 VALUES (1010, 'Hina', 'HR', 75000, 'Male');
INSERT INTO tb_grp3 VALUES (1011, 'Warner', 'Finance', 55000, '
   Female');

-- 仅 GROUP BY
SQL> SELECT department,gender,COUNT(*) group_cou FROM tb_grp3 GROUP
    BY(department,gender);

DEPARTMENT | GENDER | GROUP_COU |
-------------------------------------------------------------
Finance| Female| 1 |
Finance| Male| 2 |
HR| Female| 2 |
HR| Male| 2 |
IT| Female| 2 |
IT| Male| 2 |

-- ROLLUP
SQL> SELECT department,gender,COUNT(*) rollup_cou FROM tb_grp3
   GROUP BY ROLLUP(department,gender);

DEPARTMENT | GENDER | ROLLUP_COU |
-------------------------------------------------------------
Finance| Female| 1 |
Finance| <NULL>| 3 |
Finance| Male| 2 |
HR| Female| 2 |
HR| <NULL>| 4 |
HR| Male| 2 |
IT| Female| 2 |
<NULL>| <NULL>| 11 |
IT| <NULL>| 4 |
IT| Male| 2 |

-- CUBE
SQL> SELECT department,gender,COUNT(*) cube_cou FROM tb_grp3 GROUP
   BY CUBE(department,gender);

DEPARTMENT | GENDER | CUBE_COU |
-------------------------------------------------------------
Finance| Female| 1 |
```

```
Finance| <NULL>| 3 |
Finance| Male| 2 |
HR| Female| 2 |
HR| <NULL>| 4 |
HR| Male| 2 |
IT| Female| 2 |
<NULL>| <NULL>| 11 |
IT| <NULL>| 4 |
IT| Male| 2 |
<NULL>| Female| 5 |
<NULL>| Male| 6 |

-- GROUPING SETS
SQL> SELECT department,gender,COUNT(*) group_set_cou FROM tb_grp3
   GROUP BY GROUPING SETS(department,gender);

DEPARTMENT | GENDER | GROUP_SET_COU |
-------------------------------------------------------------
Finance| <NULL>| 3 |
HR| <NULL>| 4 |
IT| <NULL>| 4 |
<NULL>| Female| 5 |
<NULL>| Male| 6 |
```

SELECT 后支持比较表达式

1>1 返回结果错误。

```
SQL> SELECT 1>1 FROM dual;
EXPR1 |
-------------------------------------------------------------
F |
```

(1<=2 and 1<=3) 返回结果正确。

```
SQL> SELECT (1<=2 and 1<=3) FROM dual;

EXPR1 |
-------------------------------------------------------------
T |
```

查询使用函数表

将函数 aa(1) 返回的集合转换为表，使其可以被 SELECT 语句查询。

```
-- 创建table类型
SQL> CREATE TYPE obj AS TABLE OF VARCHAR(100);
/

-- 创建返回obj函数
SQL> CREATE FUNCTION aa(i INT) RETURN obj
IS
o obj:= obj();
BEGIN
o.EXTEND;
o(o.COUNT):='123';
```

```
o.EXTEND;
o(o.COUNT):='234';
RETURN o;
END;
/

-- 函数表为函数
SQL> SELECT * FROM TABLE(aa(1));

COLUMN_VALUE |
-------------------------------------------------------------
123|
234|

-- 函数表为子查询
SQL> SELECT * FROM TABLE(SELECT aa(1) FROM dual);

COLUMN_VALUE |
-------------------------------------------------------------
123|
234|
```

查询使用 CONNECT BY LEVEL

LEVEL 是伪层次列名，在树形结构中表示层级。

```
SQL> CREATE TABLE tb_level(id VARCHAR2(1));
SQL> INSERT INTO tb_level (id) VALUES ('a');
SQL> INSERT INTO tb_level (id) VALUES ('b');
SQL> INSERT INTO tb_level (id) VALUES ('c');

-- 树形结构：
a b c
SQL> SELECT id,level FROM tb_level CONNECT BY LEVEL < 2;
ID | EXPR1 |
-------------------------------------------------------------
a| 1 |
b| 1 |
c| 1 |

-- 树形结构：
a         b         c
a b c     a b c     a b c
SQL> SELECT id,level FROM tb_level CONNECT BY LEVEL < 3;
ID | EXPR1 |
-------------------------------------------------------------
a| 1 |
a| 2 |
b| 2 |
c| 2 |
b| 1 |
a| 2 |
b| 2 |
c| 2 |
c| 1 |
a| 2 |
```

```
b| 2 |
c| 2 |

-- 构造连续数字
SQL> SELECT LEVEL FROM dual CONNECT BY LEVEL <=10;
EXPR1 |
------------------------------------------------------------
1 |
2 |
3 |
4 |
5 |
6 |
7 |
8 |
9 |
10 |

-- 构造连续日期
SQL> SELECT TO_DATE('202403', 'yyyymm') + LEVEL AS FDATE FROM dual
   CONNECT BY LEVEL <= 10;
FDATE |
------------------------------------------------------------
2024-03-02 00:00:00.000 AD |
2024-03-03 00:00:00.000 AD |
2024-03-04 00:00:00.000 AD |
2024-03-05 00:00:00.000 AD |
2024-03-06 00:00:00.000 AD |
2024-03-07 00:00:00.000 AD |
2024-03-08 00:00:00.000 AD |
2024-03-09 00:00:00.000 AD |
2024-03-10 00:00:00.000 AD |
2024-03-11 00:00:00.000 AD |
```

查询使用 AS 别名做 WHERE 筛选

在 WHERE 子句中，虚谷数据库支持使用别名，且和字段定义名相比，别名优先。这和 Oracle、MySQL 不同，两者均不支持在 WHERE 子句中使用别名。对于 GROUP、ORDER BY、HAVING 子句，三者均支持使用别名，其中虚谷数据库对别名优先，Oracle 和 MySQL 对字段名优先。

```
SQL> CREATE TABLE tb_as(id INT NOT NULL, interfaceid VARCHAR(100));
SQL> INSERT INTO tb_as(id, interfaceid) VALUES (5,'ww');
SQL> INSERT INTO tb_as(id, interfaceid) VALUES (1,'aa');
SQL> INSERT INTO tb_as(id, interfaceid) VALUES (4,'bb');

-- interface列别名为id, id列别名为actionid
SQL> SELECT interfaceid AS id,id AS actionid FROM tb_as WHERE
   actionid = 5;
ID |ACTIONID |
------------------------------------------------------------
ww| 5 |
```

```
-- WHERE id = 5即原始的interface = 5，无结果返回
SQL> SELECT interfaceid AS id,id AS actionid FROM tb_as WHERE id
    = 5;
ID |ACTIONID |
-----------------------------------------------------------

SQL> SELECT interfaceid AS id, id AS actionid FROM tb_as GROUP BY
   id,actionid HAVING actionid < 5 ORDER BY actionid DESC;
ACTIONID |
-----------------------------------------------------------

bb| 4 |
aa| 1 |
```

使用 LIMIT

创建一个名为 tb_limit 的表，并插入 100 条记录：

- LIMIT 2：从 tb_limit 表中选择前 2 行记录。
- LIMIT 2,2：跳过前 2 行，然后返回接下来的 2 行记录。
- LIMIT 2 OFFSET 2：同 LIMIT 2,2
- LIMIT 10,1：跳过前 10 行，然后返回接下来的 1 行记录。

```
-- 创建数据表
SQL> CREATE TABLE tb_limit(id INT, name VARCHAR(20));

-- 插入数据
SQL> BEGIN
FOR i IN 1..100 LOOP
INSERT INTO tb_limit VALUES(i, 'limit');
END LOOP;
END;
/

-- 返回指定数据量
SQL> SELECT * FROM tb_limit LIMIT 2;

ID | NAME |
-----------------------------------------------------------
1 | limit|
2 | limit|

-- 偏移指定条数后返回指定数据量
SQL> SELECT * FROM tb_limit LIMIT 2,2;

ID | NAME |
-----------------------------------------------------------
3 | limit|
4 | limit|

SQL> SELECT * FROM tb_limit LIMIT 2 OFFSET 2;
```

```
ID | NAME |
-------------------------------------------------------------
3 | limit|
4 | limit|

-- 子查询使用LIMIT
SQL> SELECT * FROM tb_limit WHERE id = (SELECT id FROM tb_limit
   LIMIT 10,1);

ID | NAME |
-------------------------------------------------------------
11 | limit|
```

22.2 谓词

功能描述

SQL 语句中的谓词是一种特殊的函数，所有返回值均为逻辑值。

通过比较运算符形成的比较谓词：=、>、<、<>、!=、>=、<=

其他由关键字形成的谓词：LIKE、NOT LIKE、BETWEEN、IS NULL、IS NOT NULL、IN、NOT IN、EXISTS、NOT EXISTS、ESCAPE 等

谓词使用场景如下：

- 在查询语句的 WHERE 子句或者 HAVING 中使用（仅部分谓词可在 HAVING 中使用）。
- 在连接查询中确定连接条件中使用。
- 在更新或删除语句的 WHERE 条件中使用。
- 在复制表或复制表数据的 WHERE 条件中使用。
- 在存储过程、函数、触发器确定数据使用。

谓词说明如下：

- 比较谓词可用于可比较数据类型间的数据比较，不可用于 NULL 数据的比较，使用比较谓词同 NULL 返回值均为空集即无数据返回，若需判断数据是否为 NULL 须使用“IS NULL”和“IS NOT NULL”谓词。
- [NOT] LIKE：字符和通配符“%”配合使用匹配数据，该谓词主要用于模糊匹配。
- BETWEEN：“>=”和“<=”谓词的联合使用方式，必须指定上下边界即范围限制值。
- IS [NOT] NULL：判断字段数据是否为 NULL 值，如要获取是否含有 NULL 值仅可使用该谓词。

- [NOT] IN：字段值与 IN 后列表中的常量值进行匹配，若与其中任一个常量匹配则返回 true 否则返回 false。
- [NOT] EXISTS：使用子查询查询指定表是否存在一行或多行。
- ESCAPE：自定义转义符，通常与 LIKE 搭配使用，用于匹配包含通配符 (如% 和 _) 的文本。

示例

比较谓词：

```
SQL> CREATE TABLE pre_tb(id INT,name VARCHAR(30));

SQL> INSERT INTO pre_tb VALUES(1,'abc')(2,null)(3,'def')(4,'one');

SQL> SELECT * FROM pre_tb WHERE id>3;

ID | NAME |
-------------------------------------------------------------
4 | one|
```

LIKE 和 ESCAPE 谓词：

```
SQL> SELECT * FROM pre_tb WHERE name LIKE 'on%';

ID | NAME |
-------------------------------------------------------------
4 | one|

SQL> CREATE TABLE tt_escape(id INT,cc VARCHAR);

SQL> INSERT INTO tt_escape VALUES(1,'test1')(2,'test2')(3,'%test3')
   ;

-- 查询cc列中以%开头的数据
SQL> SELECT * FROM tt_escape WHERE cc LIKE '\%%' ESCAPE '\';

ID | CC |
-------------------------------------------------------------
3 | %test3|
```

EXISTS 谓词：

```
-- 返回结果为false情况
SQL> SELECT * FROM pre_tb WHERE EXISTS (SELECT * FROM pre_tb WHERE
   id=66);

ID | NAME |
-------------------------------------------------------------

-- 返回结果为true情况
SQL> SELECT * FROM pre_tb WHERE EXISTS (SELECT * FROM pre_tb WHERE
   id=1);
```

```
ID | NAME |
-------------------------------------------------------------
1 | abc|
2 | <NULL>|
3 | def|
4 | one|
```

22.3 连接

功能描述

连接是将两个或者两个以上的表或视图的行列进行组合的查询。连接查询可返回单个表字段信息也可返回多表字段信息，用户可自定义。如果多表连接时有相同字段返回，则需对重复字段进行别名定义防止同名返回出错。一般连接查询都含一个或多个连接条件用于筛选需要返回的行数据，连接条件 WHERE 子句可以是相同字段连接，也可以是不同字段相同属性进行连接，还可以仅引用其中一个表的字段信息限制连接查询返回的行。数据库支持的连接查询包括：内连接、外连接、交叉连接。

22.3.1 内连接

内连接根据连接条件分为等值连接、不等值连接、自然连接。使用内连接时，返回满足条件的所有行数据，如果两个表的相关字段满足连接条件且查询未指定应返回的表的字段，则从这两个表中获取满足条件的数据并组合成新的记录。自连接是一种特殊的内连接，是对表自身的连接。

- 等值连接：连接条件为等值比较即使用“=”比较运算符。
- 不等值连接：等值连接的相反情况，使用“>、<、<>、!=、>=、<=”比较运算符。
- 自然连接：特殊的等值连接，不可指定连接条件，数据库根据关系表中的相同字段进行条件连接，使用选择列表指出查询结果集合中所包括的列，并删除连接表中的重复列，若无相同字段则返回连接表笛卡尔积。

示例

- 示例 1 等值连接

```
SQL> CREATE TABLE pre_tb(id INT,name VARCHAR(30));
SQL> INSERT INTO pre_tb VALUES(1,'abc')(2,null)(3,'def');

SQL> CREATE TABLE pre_tb1(id INT,col1 VARCHAR(30));
SQL> INSERT INTO pre_tb1 VALUES(1,'abc')(2,null)(3,'def')(5,'two'
   );
```

```
SQL> SELECT * FROM pre_tb t1 INNER JOIN pre_tb1 t2 ON t1.id=t2.id
   ;

ID | NAME | ID | COL1 |
----------------------------
1 | abc| 1 | abc|
2 | <NULL>| 2 | <NULL>|
3 | def| 3 | def|
```

- 示例 2 自然连接

```
SQL> SELECT * FROM pre_tb t1 NATURAL JOIN pre_tb1 t2;

ID | NAME | COL1 |
----------------------------
1 | abc| abc|
2 | <NULL>| <NULL>|
3 | def| def|
```

- 示例 3 不等值连接

```
SQL> SELECT * FROM pre_tb t1 INNER JOIN pre_tb1 t2 ON t1.id>t2.id
   ;

ID | NAME | ID | COL1 |
----------------------------
2 | <NULL>| 1 | abc|
3 | def| 1 | abc|
3 | def| 2 | <NULL>|
```

22.3.2 外连接

外连接根据需要返回的行分为左外连接、右外连接、全外连接。外连接返回满足连接条件的所有行并同时返回一个表中的部分或全部不满足另一个表中的行。

- 左外连接：返回左表所有的行数据，对于左表中右表没有匹配的所有行，包含右表列的任何列均返回 null 反之返回右表数据。
- 右外连接：返回右表所有的行数据，对于右表中左表没有匹配的所有行，包含左表列的任何列均返回 null 反之返回左表数据。

左外连接以左表为基准，右外连接以右表为基准，全外连接则以左右连表为基准。

示例

- 示例 1 左外连接

```
SQL> CREATE TABLE out_tb(id INT,name VARCHAR(30));
SQL> INSERT INTO out_tb VALUES(1,'abc')(2,null)(3,'def')(4,'aaa')
   ;
```

```
SQL> CREATE TABLE out_tb1(id INT,col1 VARCHAR(30));
SQL> INSERT INTO out_tb1 VALUES(1,'abc')(2,null)(3,'def')(5,'two'
   );

SQL> SELECT * FROM out_tb t1 LEFT JOIN out_tb1 t2 ON t1.id=t2.id;

ID | NAME | ID | COL1 |
-------------------------------------------------
1 | abc| 1 | abc|
2 | <NULL>| 2 | <NULL>|
3 | def| 3 | def|
4 | aaa| <NULL>| <NULL>|
```

- 示例 2 右外连接

```
SQL> SELECT * FROM out_tb t1 RIGHT JOIN out_tb1 t2 ON t1.id=t2.id
   ;

ID | NAME | ID | COL1 |
-------------------------------------------------
1 | abc| 1 | abc|
2 | <NULL>| 2 | <NULL>|
3 | def| 3 | def|
<NULL>| <NULL>| 5 | two|
```

- 示例 3 全外连接

```
SQL> SELECT * FROM out_tb t1 FULL JOIN out_tb1 t2 ON t1.id=t2.id;

ID | NAME | ID | COL1 |
-------------------------------------------------
1 | abc| 1 | abc|
2 | <NULL>| 2 | <NULL>|
3 | def| 3 | def|
4 | aaa| <NULL>| <NULL>|
<NULL>| <NULL>| 5 | two|
```

22.3.3 交叉连接

交叉连接即笛卡尔积，无连接条件，数据库将返回一个表的每一行与另一个表的每一行相结合的结果行，若连接表数据量过多则应避免使用交叉连接产生过多数据。以下示例中，out_tb 表包含 4 行数据，out_tb1 表包含 4 行数据，则交叉连接后共 16 行数据。

示例

笛卡尔积

```
SQL> SELECT * FROM out_tb t1 CROSS JOIN out_tb1 t2;

ID | NAME | ID | COL1 |
----------------------------
```

```
1 | abc| 1 | abc|
1 | abc| 2 | <NULL>|
1 | abc| 3 | def|
1 | abc| 5 | two|
2 | <NULL>| 1 | abc|
2 | <NULL>| 2 | <NULL>|
2 | <NULL>| 3 | def|
2 | <NULL>| 5 | two|
3 | def| 1 | abc|
3 | def| 2 | <NULL>|
3 | def| 3 | def|
3 | def| 5 | two|
4 | aaa| 1 | abc|
4 | aaa| 2 | <NULL>|
4 | aaa| 3 | def|
4 | aaa| 5 | two|
```

22.4 集合

功能描述

数据库支持的集合操作主要包括交集、并集、差集，其中交集对应的关键字为 INTERSECT，并集对应的关键字为 UNION，差集对应的关键字为 MINUS 或 EXCEPT。

数据库集合操作要求集合关键字两边的表必须含有相同字段数，且两边表对应字段的数据类型必须相同。

22.4.1 交集

交集是取两个表查询结果的相交部分，若左右两表字段名称不一致则返回结果字段信息以左边为准。

示例

求交集。

```
CREATE TABLE mtb(id INT,col1 INT,col2 INT,name VARCHAR(10));
INSERT INTO mtb VALUES(1,10,4500,'a')(2,20,3000,'b')(3,10,4500,'c')
   ;

CREATE TABLE mtb1(id INT IDENTITY(1,1),name VARCHAR,deptno INT,sal
   NUMERIC(10,2));
INSERT INTO mtb1 VALUES(1,'王 五',10,4500)(2,'张 三',10,5000)(3,'李 四
   ',10,5500);

SQL> SELECT id,deptno,sal FROM mtb1 INTERSECT SELECT id,col1,col2
   FROM mtb;

ID | DEPTNO | SAL |
----------------------------
```

```
1 | 10 | 4500|
```

22.4.2 并集

并集是将两个查询的结果组合在一起。并集分为去重和不去重两种情况，若关键字使用 UNION 则表示返回去重后的数据行，若关键字使用 UNION ALL 则表示返回未去重的数据行，用户可依照实际情况自行选择。

示例

- 示例 1 并集去重

```
SQL> SELECT id,deptno,sal FROM mtb1 UNION SELECT id,col1,col2
   FROM mtb;

ID | DEPTNO | SAL |
-----------------------------------------------------
1 | 10 | 4500|
3 | 10 | 4500|
3 | 10 | 5500|
2 | 20 | 3000|
2 | 10 | 5000|
```

- 示例 2 并集不去重

```
SQL> SELECT id,deptno,sal FROM mtb1 UNION ALL SELECT id,col1,col2
    FROM mtb;

ID | DEPTNO | SAL |
-----------------------------------------------------
1 | 10 | 4500|
2 | 10 | 5000|
3 | 10 | 5500|
1 | 10 | 4500|
2 | 20 | 3000|
3 | 10 | 4500|
```

22.4.3 差集

差集是取两个查询结果不相同的部分，关键字可使用 MINUS 或 EXCEPT，其中 MINUS 兼容 Oracle，EXCEPT 兼容 MySQL。

示例

求两表差集。

```
SQL> SELECT id,deptno,sal FROM mtb1 MINUS SELECT id,col1,col2 FROM
   mtb;

ID | DEPTNO | SAL |
----------------------------
```

```
3 | 10 | 5500|
2 | 10 | 5000|

SQL> SELECT id,deptno,sal FROM mtb1 EXCEPT SELECT id,col1,col2 FROM
    mtb;

ID | DEPTNO | SAL |
----------------------------
3 | 10 | 5500|
2 | 10 | 5000|
```

22.5 分组

功能描述

分组子句关键字为“GROUP BY”，通过 GROUP BY 指定的规则将数据划分为若干组，再对每一组的数据进行处理，分组子句通常配合聚合函数进行统计数据。GROUP BY 指定的规则即字段信息可以包含 FROM 子句中的表、视图的任何列，若查询仅返回聚合函数结果则 GROUP BY 指定的列可以不在选择列表中，若查询返回聚合函数结果和其他字段信息则 GROUP BY 指定的列必须在返回的选择列表中。

GROUP BY 扩展功能：

- ROLLUP：n+1 次分组统计，其中 n 为分组字段数，按照从右至左依次递减字段生成分组统计，即对每个分组结果再进行小计和总和。如 ROLLUP(a,b) 的分组为 (a,b)、(a)、() 三种情况，第一次将整体作为条件进行分组小计，第二次对分组统计的最后一个字段删减作为条件再进行分组小计，第三次再次从右至左删减字段作为条件进行分组小计即为无条件将整个表进行统计。
- CUBE：多字段分组时按照聚合字段排列组合进行分组统计并对每一种组合结果进行汇总统计。如 CUBE(a,b) 统计的分组为 (a,b)、(a)、(b)、() 情况。
- GROUPING SETS：无需所有分组字段的排列组合仅返回每个字段的分组小计。如 GROUPING SETS(a,b) 统计的分组为 (a)、(b) 情况。

语法格式请参见查询语法章节。

示例

- 示例 1 无指定字段返回。

```
SQL> CREATE TABLE grp_tb(id INT,name VARCHAR(20));
SQL> INSERT INTO grp_tb VALUES(1,'abc')(2,'def')(1,'one');

SQL> SELECT COUNT(name) FROM grp_tb GROUP BY id;
```

```
EXPR1 |
-------------------------------------------------------------
2 |
1 |
\item 示例2 GROUP BY指定字段在返回列表字段中。
\begin{lstlisting}[language=xgsql]
SQL> SELECT COUNT(name),id FROM grp_tb GROUP BY id;

EXPR1 | ID |
-------------------------------------------------------------
2 | 1 |
1 | 2 |
```

- 示例 3 GROUP BY 的三种扩展方式。

```
SQL> CREATE TABLE tb_grp3(id INT PRIMARY KEY,name VARCHAR(15),
   department VARCHAR(10),salary NUMBER(8, 2),gender VARCHAR(10))
   ;

SQL> INSERT INTO tb_grp3 VALUES (1001, 'John', 'IT', 35000, 'Male
   ');
INSERT INTO tb_grp3 VALUES (1002, 'Smith', 'HR', 45000, 'Male');
INSERT INTO tb_grp3 VALUES (1003, 'James', 'Finance', 50000, '
   Male');
INSERT INTO tb_grp3 VALUES (1004, 'Mike', 'Finance', 50000, 'Male
   ');
INSERT INTO tb_grp3 VALUES (1005, 'Linda', 'HR', 75000, 'Female')
   ;
INSERT INTO tb_grp3 VALUES (1006, 'Anurag', 'IT', 35000, 'Male');
INSERT INTO tb_grp3 VALUES (1007, 'Priyanla', 'HR', 45000, '
   Female');
INSERT INTO tb_grp3 VALUES (1008, 'Sambit', 'IT', 55000, 'Female'
   );
INSERT INTO tb_grp3 VALUES (1009, 'Pranaya', 'IT', 57000, 'Female
   ');
INSERT INTO tb_grp3 VALUES (1010, 'Hina', 'HR', 75000, 'Male');
INSERT INTO tb_grp3 VALUES (1011, 'Warner', 'Finance', 55000, '
   Female');
-- 仅GROUP BY
SQL> SELECT department,gender,COUNT(*) group_cou FROM tb_grp3
   GROUP BY(department,gender);

DEPARTMENT | GENDER | GROUP_COU |
-------------------------------------------------------------
Finance| Female| 1 |
Finance| Male| 2 |
HR| Female| 2 |
HR| Male| 2 |
IT| Female| 2 |
IT| Male| 2 |

-- ROLLUP
SQL> SELECT department,gender,COUNT(*) rollup_cou FROM tb_grp3
   GROUP BY ROLLUP(department,gender);
```

```
DEPARTMENT | GENDER | ROLLUP_COU |
-------------------------------------------------------------
Finance| Female| 1 |
Finance| <NULL>| 3 |
Finance| Male| 2 |
HR| Female| 2 |
HR| <NULL>| 4 |
HR| Male| 2 |
IT| Female| 2 |
<NULL>| <NULL>| 11 |
IT| <NULL>| 4 |
IT| Male| 2 |

-- CUBE
SQL> SELECT department,gender,COUNT(*) cube_cou FROM tb_grp3
   GROUP BY CUBE(department,gender);

DEPARTMENT | GENDER | CUBE_COU |
-------------------------------------------------------------
Finance| Female| 1 |
Finance| <NULL>| 3 |
Finance| Male| 2 |
HR| Female| 2 |
HR| <NULL>| 4 |
HR| Male| 2 |
IT| Female| 2 |
<NULL>| <NULL>| 11 |
IT| <NULL>| 4 |
IT| Male| 2 |
<NULL>| Female| 5 |
<NULL>| Male| 6 |

-- GROUPING SETS
SQL> SELECT department,gender,COUNT(*) group_set_cou FROM tb_grp3
    GROUP BY GROUPING SETS(department,gender);

DEPARTMENT | GENDER | GROUP_SET_COU |
-------------------------------------------------------------
Finance| <NULL>| 3 |
HR| <NULL>| 4 |
IT| <NULL>| 4 |
<NULL>| Female| 5 |
<NULL>| Male| 6 |
```

22.6 排序

功能描述

排序子句关键字为“ORDER BY”，通过 ORDER BY 指定规则对返回的数据行进行排序，最终展现排序后的结果。排序指定字段既可在返回字段列表中也可以不在返回字段列表中，如果

没有 ORDER BY 子句，多次执行同一查询返回的结果不能保证是相同的顺序，数据库按照堆上顺序进行返回。

排序子句后可跟 ASC 和 DESC 关键字，ASC 表示升序，DESC 表示降序，如果排序子句未指定 ASC 或 DESC 则默认为 ASC（升序）。

语法格式请参见查询语法章节。

示例

- 示例 1 默认排序。

```
SQL> CREATE TABLE odr_tb(id INT,name VARCHAR(20));
SQL> INSERT INTO odr_tb VALUES(1,'abc')(2,'one')(66,'ooo');

SQL> SELECT * FROM odr_tb ORDER BY id;

ID | NAME |
-----------------------------------------------
1 | abc|
2 | one|
66 | ooo|
```

- 示例 2 指定升序。

```
SQL> SELECT * FROM odr_tb ORDER BY id ASC;

ID | NAME |
-----------------------------------------------
1 | abc|
2 | one|
66 | ooo|
```

- 示例 3 指定降序。

```
SQL> SELECT * FROM odr_tb ORDER BY id DESC;

ID | NAME |
-----------------------------------------------
66 | ooo|
2 | one|
1 | abc|
```

22.7 LIMIT

功能描述

通过指定返回结果集的行数来控制查询结果的大小。通常与 SELECT 语句一起使用，可以显著减少数据库服务器的负载，提高查询性能。

LIMIT 子句的基本语法是在 SELECT 语句的末尾添加 LIMIT 关键字，后面跟一个或多个参数。参数可以是具体的行数，也可以是一个表示偏移量（OFFSET）的值和一个表示返回行数的值。当前版本 LIMIT 最大支持到 2147483647，超过报错。

语法格式请参见查询语法章节。

示例

```
-- 创建表
SQL> CREATE TABLE limit_t1(id INT);

SQL> DECLARE
BEGIN
FOR i IN 1..3 LOOP
INSERT INTO limit_t1 VALUES(i);
END LOOP;
END;

SQL> SELECT * FROM limit_t1 LIMIT 2147483647;

ID |
-----------------------------------------------------------
1|
2|
3|

SQL> SELECT * FROM limit_t1 LIMIT 2147483648;
[E19132] 语法错误

-- 查询表limit_t1中跳过前1条数据并返回接下来的2条数据
SQL> SELECT * FROM limit_t1 LIMIT 2 OFFSET 1;
ID |
-----------------------------------------------------------
2|
3|

-- 查询表limit_t1中跳过前1条数据并返回接下来的2条数据
SQL> SELECT * FROM limit_t1 LIMIT 1,2;
ID |
-----------------------------------------------------------
2|
3|
```

22.8 WITH

功能描述

WITH 查询被称为 CTE（common table expression，公共表表达式）。WITH 查询方式可以认为是存在于查询中的临时表，生命周期随查询消亡存在，可以被看作为单个 SQL 语句（实际是单个或多个查询组合而成）。

如果一个查询语句需要重复使用则可以使用 WITH 实现 SQL 重用。

CTE 后面也可以跟其他的 CTE，但只能使用一个 WITH，多个 CTE 中间用英文逗号分隔，说明 WITH 查询可被 WITH 查询引用。

若 WITH 中未指定列且返回中带系统默认设置的列，则数据库会自动补齐列名如 EXPR1，随后依次递增。

语法格式请参见查询语法章节。

示例

- 示例 1 单个 WITH 语句。

```
SQL> WITH with1 AS (SELECT 'abc' FROM dual) SELECT * FROM with1;

EXPR1 |
------------------------------------------
abc|
```

- 示例 2 多个 WITH 语句。

```
SQL> WITH with1 AS (SELECT 'one' FROM dual),with2 AS (SELECT 'two
   ' FROM dual) SELECT * FROM with1 UNION SELECT * FROM with2;

EXPR1 |
------------------------------------------
one|
two|
```

22.9 WITH FUNCTION

功能描述

WITH FUNCTION 子句用于在 SQL 语句中临时声明并定义存储函数，这些存储函数可以在其作用域内被引用。相比模式对象中的存储函数，通过 WITH FUNCTION 定义的存储函数在对象名解析时拥有更高的优先级。WITH FUNCTION 定义的存储函数对象也不会存储到系统表中，且只在当前 SQL 语句内有效。

语法格式

```
WITH
    FUNCTION[PROCEDURE] <name_function[name_procedure]>
    BEGIN
    ...
    END;
SELECT <name_function[name_procedure]> FROM <TABLE>;
```

参数说明

name_function[name_procedure]：临时声明并定义的存储函数/过程。

示例

使用 WITH FUNCTION 和 WITH PROCEDURE 定义临时函数和临时过程：

```
SQL> CREATE TABLE T_with_tab(id INT,name VARCHAR(20));

SQL> INSERT INTO T_with_tab VALUES(1,'aa');

-- WITH FUNCTION
SQL> WITH FUNCTION with_function(p_id IN NUMBER) RETURN NUMBER IS
   BEGIN RETURN p_id;END;SELECT with_function(id) FROM T_with_tab
   WHERE rownum = 1;

EXPR1 |
------------------------------------------------------------
1|

-- WITH PROCEDURE
SQL> WITH PROCEDURE with_procedure(p_id IN NUMBER) IS BEGIN
   DBMS_OUTPUT.put_line('p_id='||p_id); END;SELECT id FROM
   T_with_tab WHERE rownum = 1;

ID |
------------------------------------------------------------
1 |
```

使用动态 SQL 调用 WITH FUNCTION：

```
SQL> CREATE TABLE test_with_function_1(id INT,name VARCHAR(20));

SQL> INSERT INTO test_with_function_1 VALUES(1,'AB');

SQL> DECLARE
l_sql VARCHAR2(32767);
l_cursor SYS_REFCURSOR;
l_value NUMBER;
BEGIN
l_sql := 'WITH
FUNCTION with_function(p_id IN NUMBER) RETURN NUMBER IS
BEGIN
RETURN p_id;
END;
SELECT with_function(id)
FROM TEST_WITH_FUNCTION_1
WHERE rownum = 1';
OPEN l_cursor FOR l_sql;
FETCH l_cursor INTO l_value;
DBMS_OUTPUT.PUT_LINE('输 出:l_value=' || l_value);
CLOSE l_cursor;
END;
/
输 出:l_value=1
```

⚠ 注意

- WITH FUNCTION 功能暂不支持多个存储过程/函数。
- 定义的函数的作用域为所在的查询表达式内。
- 定义的函数不能是外部函数。

22.10 子查询

功能描述

子查询，也为嵌套查询，允许出现在任何可以为表达式的地方。子查询可以是一个完整独立的查询语句，可包含选择列表的简单 SELECT 查询语句、包含一个甚至多个表或视图名称的 FROM 子句、可选的 WHERE 子句、可选的 GROUP BY 子句、可选的 HAVING 子句；子查询中亦可以嵌套子查询。

子查询可在查询选择列表中，如果子查询位于查询选择列表中且作为独立字段则只允许返回单行数据否则数据库抛出错误，若子查询位于 WHERE 条件后则需配合比较谓词和其他谓词进行使用。

⚠ 注意

子查询有以下限制：

- WHERE 条件中若需要子查询返回多行则需要使用 EXISTS、IN、ANY、ALL 等关键字修饰
- 比较谓词、BETWEEN、LIKE 等关键字引入的子查询必须返回单行且单个字段
- 如果多个子查询中的列与需要查询的列具有相同的名称，则必须对表的列的任何引用加上表名或别名

示例

- 示例 1 字段上的子查询。

  如果返回字段是子查询，则要求返回数据为单行数据。

```
SQL> CREATE TABLE sub_sel(id INT,name VARCHAR(30));
SQL> INSERT INTO sub_sel VALUES(1,'one')(2,'two');
SQL> CREATE TABLE sub_sel1(id INT,col1 VARCHAR(30));
SQL> INSERT INTO sub_sel1 VALUES(1,'abc')(66,'bcd');

SQL> SELECT id,name,(SELECT col1 FROM sub_sel1 WHERE id=66) FROM
   sub_sel;

ID | NAME | EXPR1 |

-----------------------------------------------------
```

```
1 | one| bcd|
2 | two| bcd|
```

- 示例 2 WHERE 子句上使用比较谓词的子查询。

  如果使用比较谓词，则子查询要求返回数据为单行数据。

```
SQL> SELECT * FROM sub_sel WHERE id >= (SELECT id FROM sub_sel1
   WHERE col1='abc');

ID | NAME |

-----------------------------------------------------

1 | one|
2 | two|
```

- 示例 3 WHERE 子句使用 IN 的子查询。

  使用 IN 子查询，则无需限制数据返回行数。

```
SQL> SELECT * FROM sub_sel WHERE id IN (SELECT id FROM sub_sel1);

ID | NAME |

-----------------------------------------------------

1 | one|
```

- 示例 4 使用 ALL 的子查询。

  ALL 可引入比较运算符，=ALL 则需指定条件等于子查询返回的所有数据，若存在一个不满足则无数据返回；若使用 ANY 则表示要使某一行满足查询中指定的条件，子查询的列中的值必须至少等于子查询返回的值列表中的一个值。

```
SQL> SELECT * FROM sub_sel WHERE id = ALL(SELECT id FROM sub_sel1
   );

ID | NAME |
```

- 示例 5 使用 EXISTS 的子查询。

  EXISTS 无需指定条件，EXISTS 后的子查询在无数据情况下返回 false，存在数据情况下返回 true，仅用作数据存在判断。

```
SQL> SELECT * FROM sub_sel WHERE EXISTS(SELECT id FROM sub_sel1);

ID | NAME |

-----------------------------------------------------

1 | one|
```

```
2 | two|
```

22.11 q’ 转义

功能描述

q’ 转义功能用来对包含特殊字符的字符串进行转义。

q’ 转义符通常后面使用!! [] () <> \\等转义符号（转义符需成对出现）。

语法格式

```
q'转义符 expr 转义符
```

参数说明 expr：待转义字符串。

示例

```
SQL> SELECT q'[it's an example1]' FROM dual;

EXPR1 |
---------------------------
it's an example1|

SQL> SELECT q'{it's an example2}' FROM dual;

EXPR1 |
---------------------------
it's an example2|

SQL> SELECT q'(it's an example3)' FROM dual;

EXPR1 |
---------------------------
it's an example3|

SQL> SELECT q'<it's an example4>' FROM dual;

EXPR1 |
---------------------------
it's an example4|

SQL> SELECT q'\it's an example5\' FROM dual;

EXPR1 |
---------------------------
it's an example5|

SQL> SELECT q'!it's an example6!' FROM dual;

EXPR1 |
---------------------------
it's an example6|
```

> **⚠ 注意**
>
> 目前只支持转义符号!! [] () <> \\且需成对出现。

22.12 HINT

功能描述

HINT 是一种以固定的格式和位置出现在 SQL 文本中的特殊注释，其作用是指定执行计划，用户可通这种方式改变优化器对执行计划的规划，从而使 SQL 按更优的执行计划执行。

语法格式

```
/*+HINT_TEXT*/
```

参数说明

HINT_TEXT 目前支持以下选项。

- INDEX(table_name index_name)：指示对目标表使用索引扫描。
- NOINDEX(table_name index_name)：指示对目标表不使用指定索引。
- FULL(table_name)：指示对目标表使用全表扫描。
- USE_HASH(table_name1,table_name2)：指示目标表 join 路径使用 hash join。

> **⚠ 注意**
>
> - 若 HINT_TEXT 中指定的表，在查询语句中指定了别名，则在 HINT_TEXT 中需使用此别名，否则 hint 将失效。
> - + 前不可存在空格，否则 hint 将失效。

示例

先创建表并创建索引。

```
-- 创建表并插入数据
SQL> CREATE TABLE t_hint (c1 INT, c2 INT);

SQL> BEGIN
FOR i IN 1..10 LOOP
INSERT INTO t_hint VALUES (i, i * 2);
END LOOP;
END;
/

-- 创建索引
```

```
SQL> CREATE INDEX i_hint ON t_hint(c1);
```

- 未使用 HINT 的查询执行计划：

```
SQL> EXPLAIN SELECT * FROM t_hint WHERE c1 = 3;

plan_path |
----------------------------------------------------------------
1   SeqScan[(1 1) cost=10,result_num=1](table=T_HINT)|
```

- 使用 HINT 的查询执行计划：

```
SQL> EXPLAIN SELECT /*+index(t_hint i_hint)*/ * FROM t_hint WHERE
    c1 = 3;

plan_path |
----------------------------------------------------------------
1   BtIdxScan[(1 1) cost=300,result_num=1](table=T_HINT)(index=
   I_HINT)|

-- 使用表别名
SQL> EXPLAIN SELECT /*+index(t i_hint)*/ * FROM t_hint t WHERE t.
   c1 = 3;

plan_path |
----------------------------------------------------------------
1   BtIdxScan[(1 1) cost=300,result_num=1](table=T_HINT)(index=
   I_HINT)|
```

22.13 PARALLEL

功能描述

查询执行过程中并行处理。

指定并行度可以指示数据库使用多个线程并行执行查询操作，从而提高执行效率，通常用于大数据量处理。

> ⚠ 注意
>
> 并不是设置 PARALLE，就会并行执行。当前数据库设计上禁用了包含 WITH 子句的所有并行。

基本语法是在 SQL 语句的末尾添加 PARALLEL 关键字，后面跟一个整数值，表示并行度。取值范围 [0,1024]，小于 2 均走串行执行。

语法格式请参见查询语法章节。

示例

```
-- 创建表
SQL> CREATE TABLE test_parallel(i1 int)

SQL> DECLARE
BEGIN
FOR i IN 1..3 LOOP
INSERT INTO test_parallel VALUES(i);
END LOOP;
END;

--PARALLE设置总是位于SELECT语句的末尾
SQL> SELECT * FROM test_parallel PARALLEL 3;
I1 |
-----------------------------------------------------------
1|
2|
3|

SQL> SELECT * FROM test_paralell ORDER BY i1 PARALLEL 3;
I1 |
-----------------------------------------------------------
1|
2|
3|

SQL> SELECT * FROM test_parallel GROUP BY i1 PARALLEL 3;
I1 |
-----------------------------------------------------------
1|
2|
3|
```

22.14 开窗

22.14.1 概述

窗口函数是一种 SQL 函数，其输入值是从 SELECT 语句结果集中的一个或多个“窗口”中获取所得。

窗口函数在一组与当前行相关的表行之间执行计算，这与使用聚合函数可以完成的计算类型类似。但与常规的聚合函数不同的是，使用窗口函数不会将行分组到单个输出行中——这些行保留它们各自的特性。

窗口函数与其他 SQL 函数的区别在于包含 OVER 子句。如果一个函数包含一个 OVER 子句，那么它就是一个窗口函数。如果它缺少 OVER 子句，则它是普通聚合或标量函数。

语法格式

```
over_clause::=
```

```
    analytic_function([ arguments ]) OVER (analytic_clause)

analytic_clause::=
      [opt_win_parti] [opt_win_order] [opt_win_range]

opt_win_parti::=
    PARTITION BY expr [, expr]...

opt_win_order::=
    ORDER BY expr [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]...

opt_win_range::=
    { ROWS | RANGE } { BETWEEN  { UNBOUNDED PRECEDING | CURRENT ROW
    | value_expr { PRECEDING | FOLLOWING } } AND { UNBOUNDED
   FOLLOWING | CURRENT ROW | value_expr { PRECEDING | FOLLOWING } }
    | { UNBOUNDED PRECEDING | CURRENT ROW  | value_expr PRECEDING
     }
}
```

参数说明

- analytic_function：窗口函数的名称。
- arguments：窗口函数的参数。
- OVER (analytic_clause)：定义窗口的范围。

opt_win_parti 规则

- 使用 PARTITION BY 子句根据一个或多个值（表字段）将查询结果集划分为多个组。如果省略此子句，则该函数将查询结果集的所有行作为单个组处理。
- 同一个查询中可指定多个分析函数，每个函数都具有相同或不同的 PARTITION BY 键。
- 如果要查询的对象具有 parallel 属性，并且如果使用 opt_win_parti 指定一个分析函数，那么函数计算也将并行化。
- opt_win_parti 参数的有效值是常量、列、非解析函数、函数表达式或涉及其中任何一个表达式的表达式。

opt_win_order 规则

使用 opt_win_order 指定分组内的数据排序方式。对于所有分析函数，可以将分组中的值按多个键排序，每个键由 opt_win_order 参数定义，每个键定义一个排序序列。

使用 opt_win_order 后多行生成相同的值时：

- UME_DIST、DENSE_RANK、NTILE、PERCENT_RANK 和 RANK 对每一行返回相同的结果。

- ROW_NUMBER 为每一行分配一个不同的值，该值基于按 opt_win_order 处理行的顺序，如果 ORDER BY 不能保证总的顺序，则结果可能是不确定的。
- 对于所有其他分析函数，结果取决于窗口规格。如果使用 RANGE 关键字指定一个逻辑窗口，那么该函数将为每一行返回相同的结果。如果使用 ROWS 关键字指定物理窗口，则结果是不确定的。

opt_win_order 使用时有如下限制：在分析函数中使用时，opt_win_order 必须接受一个表达式 (expr)，位置 (Position) 和列别名 (c_alias) 是无效的。若使用位置或列别名，则 opt_win_order 按照堆存储的排序返回结果。

使用 RANGE 关键字的分析函数可以在其中使用多个排序键 ORDER BY 子句，如果它指定了以下任何一个窗口：

- RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
- RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
- RANGE BETWEEN CURRENT ROW AND CURRENT ROW
- RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

在 ORDER 中，上述四个窗口边界之外的窗口边界只能有一个排序键，此限制不适用于 ROWS 关键字指定的窗口边界。

opt_win_range 规则

- opt_win_range 指定 ROWS：
  - value_expr 是物理偏移量，它必须是常量或值为非负数的表达式。
  - 若 value_expr 是起点的一部分，那它必须在终点之前对行求值。
- opt_win_range 指定 RANGE：
  - value_expr 是逻辑偏移量，它必须是常量或值为非负数的表达式、时间、INTERVAL 时间值或文字常量。
  - 若 value_expr 值为一个数字，那 opt_win_order 中的 expr 必须为数字或时间类型。
  - 若 value_expr 为一个间隔值，那 opt_win_order 中的 expr 必须是一个时间类型。

22.14.2 ROWS 和 RANGE

功能描述

ROWS 和 RANGE 关键字为每一行定义一个窗口（一组物理或逻辑行），用于计算函数结果，

然后将该函数应用于窗口中的所有行。窗口从上到下在查询结果集或分区中移动。

- ROWS 窗口由物理行构成
- RANGE 窗口由逻辑偏移量构成
- 若完全忽略 opt_win_range，则默认的窗口范围为 range between unbounded preceding and current row

ROWS 窗口计算规则

Window|ASC/DESC	ASC 窗口计算规则	DESC 窗口计算规则
RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING	窗口开始于分组第一行，结束于分组最后一行	同 ASC 一致
ROWS [BETWEEN] UNBOUNDED PRECEDING [AND CURRENT ROW]	窗口开始于分组第一行，结束于当前行	同 ASC 一致
ROWS BETWEEN UNBOUNDED PRECEDING AND value_expr PRECEDING	窗口开始于分组第一行，结束于当前行前 value_expr 行	同 ASC 一致
ROWS BETWEEN UNBOUNDED PRECEDING AND value_expr FOLLOWING	窗口开始于分组第一行，结束于当前行后 value_expr 行	同 ASC 一致
ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING	窗口开始于当前行，结束于分组最后一行	同 ASC 一致
ROWS [BETWEEN CURRENT ROW AND] CURRENT ROW	窗口开始于当前行，结束于当前行	同 ASC 一致
ROWS BETWEEN CURRENT ROW AND value_expr FOLLOWING	窗口开始于当前行，结束于当前行后 value_expr 行	同 ASC 一致

接下页

Window|ASC/DESC	ASC 窗口计算规则	DESC 窗口计算规则
ROWS BETWEEN value_expr PRECEDING AND UNBOUNDED FOLLOWING	窗口开始于当前行前 value_expr 行，结束于分组最后一行	同 ASC 一致
ROWS [BETWEEN value_expr] PRECEDING [AND CURRENT ROW]	窗口开始于当前行前 value_expr 行，结束于当前行	同 ASC 一致
ROWS BETWEEN value_expr1 PRECEDING AND value_expr2 PRECEDING	窗口开始于当前行前 value_expr1 行，结束于当前行前 value_expr2 行	同 ASC 一致
ROWS BETWEEN value_expr1 PRECEDING AND value_expr2 FOLLOWING	窗口开始于当前行前 value_expr1 行，结束于当前行后 value_expr2 行	同 ASC 一致
ROWS BETWEEN value_expr FOLLOWING AND UNBOUNDED FOLLOWING	窗口开始于当前行后 value_expr 行，结束于分组最后一行	同 ASC 一致
ROWS BETWEEN value_expr1 FOLLOWING AND value_expr2 FOLLOWING	窗口开始于当前行后 value_expr1 行，结束于当前行后 value_expr2 行	同 ASC 一致
ROWS UNBOUNDED PRECEDING	（与 2 等价）	同 ASC 一致
ROWS CURRENT ROW	（与 6 等价）	同 ASC 一致
ROWS value_expr PRECEDING	（与 9 等价）	同 ASC 一致

接下页

Window|ASC/DESC	ASC 窗口计算规则	DESC 窗口计算规则
ROWS BETWEEN CURRENT ROW AND value_expr PRECEDING	无效	无效
ROWS BETWEEN value_expr FOLLOWING AND CURRENT ROW	无效	无效
ROWS BETWEEN value_expr1 FOLLOWING AND value_expr2 PRECEDING	无效	无效
ROWS UNBOUNDED FOLLOWING	无效	无效
ROWS value_expr FOLLOWING	无效	无效

RANGE 窗口计算规则

Window|ASC/DESC	ASC 窗口计算规则	DESC 窗口计算规则
RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING	WHEREexprBETWEENfirst_valueANDlast_value	WHEREexprBETWEENlast_valueANDfirst_value
RANGE [BETWEEN] UNBOUNDED PRECEDING [AND CURRENT ROW]	WHEREexprBETWEENfirst_valueANDcurrent_value	WHEREexprBETWEENcurrent_valueANDfirst_value
RANGE BETWEEN UNBOUNDED PRECEDING AND value_expr PRECEDING	WHEREexprBETWEENfirst_valueANDcurrent_value-value_expr	WHEREexprBETWEENcurrent_value+value_exprANDfirst_value
RANGE BETWEEN UNBOUNDED PRECEDING AND value_expr FOLLOWING	WHEREexprBETWEENfirst_valueANDcurrent_value+value_expr	WHEREexprBETWEENcurrent_value-value_exprANDfirst_value

接下页

Window|ASC/DESC	ASC 窗口计算规则	DESC 窗口计算规则
RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING	WHEREexprBETWEENcurrent_valueANDlast_value	WHEREexprBETWEENlast_valueANDcurrent_value
RANGE [BETWEEN CURRENT ROW AND] CURRENT ROW	WHEREexpr=current_value	WHEREexpr=current_value
RANGE BETWEEN CURRENT ROW AND value_expr FOLLOWING	WHEREexprBETWEENcurrent_valueANDcurrent_value+value_expr	WHEREexprBETWEENcurrent_value-value_exprANDcurrent_value
RANGE BETWEEN value_expr PRECEDING AND UNBOUNDED FOLLOWING	WHEREexprBETWEENcurrent_value-value_exprANDlast_value	WHEREexprBETWEENlast_valueANDcurrent_value+value_expr
RANGE [BETWEEN value_expr] PRECEDING [AND CURRENT ROW]	WHEREexprBETWEENcurrent_value-value_exprANDcurrent_value	WHEREexprBETWEENcurrent_valueANDcurrent_value+value_expr
RANGE BETWEEN value_expr1 PRECEDING AND value_expr2 PRECEDING	WHEREexprBETWEENcurrent_value-value_expr1ANDcurrent_value-value_expr2	WHEREexprBETWEENcurrent_value+value_expr2ANDcurrent_value+value_expr1
RANGE BETWEEN value_expr1 PRECEDING AND value_expr2 FOLLOWING	WHEREexprBETWEENcurrent_value-value_expr1ANDcurrent_value+value_expr2	WHEREexprBETWEENcurrent_value-value_expr2ANDcurrent_value+value_expr1

接下页

Window|ASC/DESC	ASC 窗口计算规则	DESC 窗口计算规则
RANGE BETWEEN value_expr FOLLOWING AND UNBOUNDED FOLLOWING	WHEREexprBETWEENcurrent_value+value_exprANDlast_value	WHEREexprBETWEENlast_valueANDcurrent_value-value_expr
RANGE BETWEEN value_expr1 FOLLOWING AND value_expr2 FOLLOWING	WHEREexprBETWEENcurrent_value+value_expr1ANDcurrent_value+value_expr2	WHEREexprBETWEENcurrent_value-value_expr2ANDcurrent_value-value_expr1
RANGE UNBOUNDED PRECEDING （与 RANGE [BETWEEN] UNBOUNDED PRECEDING [AND CURRENT ROW] 等价）	WHEREexprBETWEENfirst_valueANDcurrent_value	WHEREexprBETWEENcurrent_valueANDfirst_value
RANGE CURRENT ROW （与 RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING 等价）	WHEREexpr=current_value	WHEREexpr=current_value
RANGE value_expr PRECEDING （与 RANGE [BETWEEN value_expr] PRECEDING [AND CURRENT ROW] 等价）	WHEREexprBETWEENcurrent_value-value_exprANDcurrent_value	WHEREexprBETWEENcurrent_valueANDcurrent_value+value_expr
RANGE BETWEEN CURRENT ROW AND value_expr PRECEDING	无效	无效
RANGE BETWEEN value_expr FOLLOWING AND CURRENT ROW	无效	无效

接下页

Window|ASC/DESC	ASC 窗口计算规则	DESC 窗口计算规则
RANGE BETWEEN value_expr1 FOLLOWING AND value_expr2 PRECEDING	无效	无效
RANGE UNBOUNDED FOLLOWING	无效	无效
RANGE value_expr FOLLOWING	无效	无效

ORDER BY 排序为单字段

相当于给 opt_win_order 中的 expr 加 WHERE 限定条件，即 WHERE expr BETWEEN a AND b 构成了一个逻辑窗口，此窗口在 expr 包含的行上滑动求值。a 和 b 的值可结合 opt_win_range 窗口求值确定。order_by_clause 子句为 asc 和 desc，a 和 b 的求值可能有所不同。

假设分组排序后第一行的值为 first_value，最后一行的值为 last_value，当前行的值为 current_value：

- UNBOUNDED PRECEDING = first_value
- UNBOUNDED FOLLOWING = last_value
- CURRENT ROW = current_value
- value_expr PRECEDING = current_value -/+ value_expr
- value_expr FOLLOWING = current_value +/- value_expr

> ⚠ 注意
>
> 排序指定 DESC，窗口为 value_expr preceding 则 current_value + value_expr，value_expr following 则为 current_value - value_expr。

ORDER BY 排序为多字段仅允许下述开窗。

- RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
- RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
- RANGE BETWEEN CURRENT ROW AND CURRENT ROW

- RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED

示例

- ROWS

```
SQL> CREATE TABLE over_test(name VARCHAR(20),age INT);

SQL> INSERT INTO over_test VALUES('a',15)('e',10)('f',21)('c',15)
   ('d',9)('b',9);

SQL> SELECT name,age,COUNT(age) OVER(PARTITION BY age ORDER BY
   age ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) FROM
   over_test ORDER BY age;

NAME | AGE | EXPR1 |
-------------------------------------------------------------
b| 9 | 2 |
d| 9 | 1 |
e| 10 | 1 |
c| 15 | 2 |
a| 15 | 1 |
f| 21 | 1 |
```

- RANGE

```
SQL> SELECT name,age,COUNT(age) OVER(PARTITION BY age ORDER BY
   age RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) FROM
   over_test ORDER BY age;

NAME | AGE | EXPR1 |
-------------------------------------------------------------
b| 9 | 2 |
d| 9 | 2 |
e| 10 | 1 |
c| 15 | 2 |
a| 15 | 2 |
f| 21 | 1 |
```

22.14.3 排名函数

功能描述

排名函数就是对开窗后查询的结果进行排名，其中 RANK、DENSE_RANK、ROW_NUMBER、NTILE 均为排名函数和开窗配合使用返回特定的排名结果。

- 当使用 RANK 函数出现相同排名时，相同的排名使用相同的排名号且后续排名号依次递增跳过，例如 a 和 b 并列排名号为 1 下一个数据 c 排名号则为 3，RANK 函数不支持只包括分区的情况。
- 当使用 DENSE_RANK 函数出现相同排名时，相同排名使用相同的排名号且后续排名号依

次递增不跳过，例如 a 和 b 并列排名号为 1 下一个数据 c 的排名号则为 2，DENSE_RANK 函数不支持只包括分区的情况。

- ROW_NUMBER 函数为每一组的行按顺序生成一个唯一的排名号，排名号从 1 开始，依次递增，ROW_NUMBER 函数不支持只包括分区的情况。
- NTILE 按照指定的数目将数据进行分组，并为每一组生成一个序号，将每组的序号分配给每一行，NTILE 函数不支持只包括分区的情况。
- RANK、DENSE_RANK、ROW_NUMBER 为无参函数，NTILE 为单参数函数，其中参数表示需要分配的组数。

示例

- 示例 1 RANK 排名。

```
CREATE TABLE sal_info(id INT IDENTITY(1,1),name VARCHAR,deptno
   INT,sal NUMERIC(10,2));
INSERT INTO sal_info VALUES(DEFAULT,'张 三',10,5000);
INSERT INTO sal_info VALUES(DEFAULT,'李 四',10,5500);
INSERT INTO sal_info VALUES(DEFAULT,'王 五',10,4500);
INSERT INTO sal_info VALUES(DEFAULT,'赵 六',10,4800);
INSERT INTO sal_info VALUES(DEFAULT,'陈 七',10,5500);
INSERT INTO sal_info VALUES(DEFAULT,'刘 八',20,3000);
INSERT INTO sal_info VALUES(DEFAULT,'李 九',20,3500);
INSERT INTO sal_info VALUES(DEFAULT,'周 十',20,3800);
INSERT INTO sal_info VALUES(DEFAULT,'张 一',20,2300);
INSERT INTO sal_info VALUES(DEFAULT,'李 二',20,3500);
INSERT INTO sal_info VALUES(DEFAULT,'吴 二',20,3800);
INSERT INTO sal_info VALUES(DEFAULT,'张 二',20,3800);
INSERT INTO sal_info VALUES(DEFAULT,'刘 二',20,4000);
INSERT INTO sal_info VALUES(DEFAULT,'周 二',20,4300);

SQL> SELECT *,RANK() OVER(PARTITION BY deptno ORDER BY sal) rk
   FROM sal_info ORDER BY deptno,rk;

ID | NAME | DEPTNO | SAL | RK |
----------------------------
3 | 王 五| 10 | 4500| 1 |
4 | 赵 六| 10 | 4800| 2 |
1 | 张 三| 10 | 5000| 3 |
5 | 陈 七| 10 | 5500| 4 |
2 | 李 四| 10 | 5500| 4 |
9 | 张 一| 20 | 2300| 1 |
6 | 刘 八| 20 | 3000| 2 |
10 | 李 二| 20 | 3500| 3 |
7 | 李 九| 20 | 3500| 3 |
11 | 吴 二| 20 | 3800| 5 |
8 | 周 十| 20 | 3800| 5 |
12 | 张 二| 20 | 3800| 5 |
13 | 刘 二| 20 | 4000| 8 |
14 | 周 二| 20 | 4300| 9 |
```

- 示例 2 DENSE_RANK 排名。

```
SQL> SELECT *,DENSE_RANK() OVER(PARTITION BY deptno ORDER BY sal)
    dr FROM sal_info ORDER BY deptno,dr;

ID | NAME | DEPTNO | SAL | DR |
---------------------------
3 | 王五| 10 | 4500| 1 |
4 | 赵六| 10 | 4800| 2 |
1 | 张三| 10 | 5000| 3 |
5 | 陈七| 10 | 5500| 4 |
2 | 李四| 10 | 5500| 4 |
9 | 张一| 20 | 2300| 1 |
6 | 刘八| 20 | 3000| 2 |
10 | 李二| 20 | 3500| 3 |
7 | 李九| 20 | 3500| 3 |
11 | 吴二| 20 | 3800| 4 |
8 | 周十| 20 | 3800| 4 |
12 | 张二| 20 | 3800| 4 |
13 | 刘二| 20 | 4000| 5 |
14 | 周二| 20 | 4300| 6 |
```

- 示例 3 ROW_NUMBER 排名。

```
SQL> SELECT *,ROW_NUMBER() OVER(PARTITION BY deptno ORDER BY sal)
    rn FROM sal_info ORDER BY deptno,rn;

ID | NAME | DEPTNO | SAL | RN |
---------------------------
3 | 王五| 10 | 4500| 1 |
4 | 赵六| 10 | 4800| 2 |
1 | 张三| 10 | 5000| 3 |
2 | 李四| 10 | 5500| 4 |
5 | 陈七| 10 | 5500| 5 |
9 | 张一| 20 | 2300| 1 |
6 | 刘八| 20 | 3000| 2 |
7 | 李九| 20 | 3500| 3 |
10 | 李二| 20 | 3500| 4 |
8 | 周十| 20 | 3800| 5 |
11 | 吴二| 20 | 3800| 6 |
12 | 张二| 20 | 3800| 7 |
13 | 刘二| 20 | 4000| 8 |
14 | 周二| 20 | 4300| 9 |
```

- 示例 4 NTILE 排名。

```
SQL> SELECT *,NTILE(5) OVER(PARTITION BY deptno ORDER BY sal) nt
   FROM sal_info ORDER BY deptno,nt;

ID | NAME | DEPTNO | SAL | NT |
---------------------------
3 | 王五| 10 | 4500| 1 |
4 | 赵六| 10 | 4800| 2 |
1 | 张三| 10 | 5000| 3 |
2 | 李四| 10 | 5500| 4 |
```

```
5 | 陈七| 10 | 5500| 5 |
9 | 张一| 20 | 2300| 1 |
6 | 刘八| 20 | 3000| 1 |
10 | 李二| 20 | 3500| 2 |
7 | 李九| 20 | 3500| 2 |
8 | 周十| 20 | 3800| 3 |
11 | 吴二| 20 | 3800| 3 |
13 | 刘二| 20 | 4000| 4 |
12 | 张二| 20 | 3800| 4 |
14 | 周二| 20 | 4300| 5 |
```

23 用户管理

23.1 概述

用户是一个数据库账号，用来登录数据库，并根据被授予的权限对数据库进行使用和管理。

在数据库系统中包括两类用户，一类是系统用户，一类是普通用户，系统用户是数据库创建时默认自带的用户，他们拥有系统自定义划分的权限；普通用户由数据库管理员创建，并且根据应用需求划分相应的权限。针对普通用户的管理，应采用最小权限的方式进行权限管理，避免用户越权操作数据库，影响数据库安全。

每个用户有所属的（逻辑）库，不同（逻辑）库中的用户彼此不同，即使其名称相同。

例如在 DB_1 库中创建 usr_1 用户，该用户只能用来登录 DB_1 库，而无法用来登录 DB_2 库，除非在 DB_2 库中也创建同名的用户。此外即使 DB_2 中也创建了 usr_1 用户，但其用户口令、用户权限等也可能和 DB_1 中的 usr_1 用户不同。

23.2 创建用户

23.2.1 主要语法结构

语法格式

```
CreateUserStmt::=
CREATE USER user_name [ LOGIN alias_name ] IDENTIFIED BY 'password'
[ DEFAULT ROLE role_name_1[, role_name_2[, ...] ] ]
[ VALID UNTIL date_time_expr ]
[ ACCOUNT { LOCK | UNLOCK } ]
[ PASSWORD EXPIRE ]
[ opt_trust_ip ]
[ opt_user_quotas ]
[ ENCRYPT BY 'encryptor_name' ]
```

参数说明

- user_name：要创建的用户名。
- alias_name：用户的别名。
- password：用户口令字符串。
- role_name_1, role_name_2, ... ：角色名。

- VALID UNTIL date_time_expr：用户有效期截止时间的字符串，格式为日期或日期时间。
- ACCOUNT LOCK | UNLOCK ：账户是否锁定。
- PASSWORD EXPIRE：密码失效。
- opt_trust_ip：IPV4 地址表达式，外部用单引号包裹。表达式可以为如下形式：
  - 单个 IP 地址：例如'192.168.2.21'。
  - 多个 IP 地址：用逗号分隔的多个地址，例如'192.168.2.21,192.168.2.22,192.168.2.105'。
  - IP 地址范围；用减号分隔的地址上下界，例如'192.168.2.20-192.168.2.29'。
  - 任意 IP 地址：关键字'ANY'。
- opt_user_quotas：设置用户在数据库中各种资源上的配额限制。
- encryptor_name：加密机（即加密用的密钥）的名称。加密机相关内容请参见《数据加密指南》。

23.2.2 资源配额限制 opt_user_quotas

语法格式

```
opt_user_quotas::=
QUOTA int_value quantity_unit ON MEMORY
|    QUOTA UNLIMITED ON MEMORY
|    QUOTA int_value quantity_unit ON TEMP TABLESPACE
|    QUOTA UNLIMITED ON TEMP TABLESPACE
|    QUOTA int_value quantity_unit ON UNDO TABLESPACE
|    QUOTA UNLIMITED ON UNDO TABLESPACE
|    QUOTA int_value ON CURSOR
|    QUOTA UNLIMITED ON CURSOR
|    QUOTA int_value ON SESSION
|    QUOTA UNLIMITED ON SESSION
|    QUOTA int_value ON IO
|    QUOTA UNLIMITED ON IO
|    QUOTA int_value ON PROCEDURE
|    QUOTA UNLIMITED ON PROCEDURE
|    QUOTA int_value quantity_unit ON TABLESPACE
|    QUOTA UNLIMITED ON TABLESPACE
```

参数说明

- int_value：设置的数值，取值为整数。
- quantity_unit：设置数值的单位。例如要设置的资源配额为临时表空间，则单位可以为 M、G 等。

- UNLIMITED：无限制。
- MEMORY：内存配额。
- TEMP TABLESPACE：临时表空间配额。
- UNDO TABLESPACE：回滚表空间配额。
- CURSOR：游标配额。
- SESSION：会话配额。
- IO：I/O 配额。
- PROCEDURE：存储过程配额。
- TABLESPACE：表空间配额。

23.2.3 示例

- 创建一个名为 usr_test 的用户，具有以下属性：
  - 登录别名为 ut。
  - 登录密码为 123QWEasd!@。
  - 默认角色为 role_1 和 role_2。
  - 用户有效期截至 2024-12-31 12:00:00。
  - 账户状态为 ACCOUNT LOCK 锁定状态。
  - 临时表空间配额 QUOTA 20 MB ON TEMP TABLESPACE，为用户设置了 20 MB 的临时表空间配额。

```
SQL> CREATE USER usr_test LOGIN ut IDENTIFIED BY '123QWEasd!@'
DEFAULT ROLE role_1, role_2
VALID UNTIL '2021-12-31 12:00:00'
ACCOUNT LOCK
QUOTA 20 M ON TEMP TABLESPACE;
```

- 使用 ACCOUNT LOCK 创建用户，支持对用户进行创建对象、授权操作，切换与登录报错。

```
-- 创建用户
SQL> CREATE USER user_test IDENTIFIED BY '123QWE$$&' ACCOUNT LOCK
   ;

-- 用户下创表
```

```
SQL> CREATE TABLE user_test.t1(id INT);

SQL> INSERT INTO user_test.t1 VALUES(1);

SQL> SELECT * FROM user_test.t1;

ID |
-----------------------------------------------------------
1|

-- 授权与回收权限
SQL> GRANT DBA TO user_test;

SQL> REVOKE DBA FROM user_test;

-- 切换用户失败
SQL> SET SESSION AUTHORIZATION user_test;
[E18063] 切换用户失败，账户已锁定

-- 登录用户失败
SQL> USE SYSTEM USER = user_test PASSWORD = '123QWE$$&';
[E18019] 登录验证失败

-- 解锁
SQL> ALTER USER user_test ACCOUNT UNLOCK;

-- 切换用户成功
SQL> SET SESSION AUTHORIZATION user_test;

-- 切换回SYSDBA
SQL> SET SESSION AUTHORIZATION SYSDBA;

-- 登录用户成功
SQL> USE SYSTEM USER = user_test PASSWORD = '123QWE$$&';

-- 登录回SYSDBA
SQL> USE SYSTEM USER = SYSDBA PASSWORD = 'SYSDBA';
```

- 使用 PASSWORD EXPIRE 创建用户，支持对用户进行创建对象、授权操作，切换与登录报错。

```
-- 创建用户
SQL> CREATE USER user_test_2 IDENTIFIED BY '123qwe###' PASSWORD
   EXPIRE;

-- 用户下创表
SQL> CREATE TABLE user_test_2.t2(id INT);

SQL> INSERT INTO user_test_2.t2 VALUES(1);

SQL> SELECT * FROM user_test_2.t2;

ID |
-----------------------------------------------------------
```

```
1|

-- 授权与回收权限
SQL> GRANT DBA TO user_test_2;

SQL> REVOKE DBA FROM user_test_2;

-- 切换用户失败
SQL> SET SESSION AUTHORIZATION user_test_2;
[E18064] 切换用户失败，密码已失效

-- 登录用户失败
SQL> USE SYSTEM USER = user_test_2 PASSWORD = '123qwe###';
[E18019] 登录验证失败

-- 修改用户密码
SQL> ALTER USER user_test_2 IDENTIFIED BY 'pass_1234';

-- 切换用户成功
SQL> SET SESSION AUTHORIZATION user_test_2;

-- 切换回SYSDBA
SQL> SET SESSION AUTHORIZATION SYSDBA;

-- 登录用户成功
SQL> USE SYSTEM USER = user_test_2 PASSWORD = 'pass_1234';

-- 登录回SYSDBA
SQL> USE SYSTEM USER = SYSDBA PASSWORD = 'SYSDBA';
```

- 使用 VALID UNTIL 创建用户，支持对用户进行创建对象、授权操作，切换与登录报错。

```
-- 创建用户
SQL> CREATE USER user_test_3 IDENTIFIED BY '123qwe###' VALID
   UNTIL '2008-08-08';

-- 用户下创表
SQL> CREATE TABLE user_test_3.t3(id INT);

SQL> INSERT INTO user_test_3.t3 VALUES(1);

SQL> SELECT * FROM user_test_3.t3;

ID |
-----------------------------------------------------------
1|

-- 授权与回收权限
SQL> GRANT DBA TO user_test_3;

SQL> REVOKE DBA FROM user_test_3;

-- 切换用户失败
SQL> SET SESSION AUTHORIZATION user_test_3;
[E18062] 切换用户失败，时间已过期
```

```
-- 登录用户失败
SQL> USE SYSTEM USER = user_test_3 PASSWORD = '123qwe###';
[E18019] 登录验证失败

-- 修改用户时间
SQL> ALTER USER user_test_3 VALID UNTIL '2099-10-1';

-- 切换用户成功
SQL> SET SESSION AUTHORIZATION user_test_3;

-- 切换回SYSDBA
SQL> SET SESSION AUTHORIZATION SYSDBA;

-- 登录用户成功
SQL> USE SYSTEM USER = user_test_3 PASSWORD = '123qwe###';

-- 登录回SYSDBA
SQL> USE SYSTEM USER = SYSDBA PASSWORD = 'SYSDBA';
```

23.3 修改用户信息

23.3.1 主要语法结构

语法格式

```
AlterUserStmt::=
ALTER USER user_name
[ LOGIN alias_name ]
[ IDENTIFIED BY 'password' ]
[ VALID UNTIL date_time_expr ]
[ ACCOUNT { LOCK | UNLOCK } ]
[ PASSWORD EXPIRE ]
[ opt_trust_ip ]
[ opt_user_quotas ]
```

参数说明

- user_name：要修改的用户名。
- alias_name：用户的别名。
- password：用户口令字符串。
- role_name_1, role_name_2, ... ：角色名。
- date_time_expr：用户有效期截止时间的字符串，格式为日期或日期时间。
- opt_trust_ip：IPV4 地址表达式，外部用单引号包裹。表达式可以为如下形式：
  - 单个 IP 地址：例如’192.168.2.21’。

  - 多个 IP 地址：用逗号分隔的多个地址，例如’192.168.2.21,192.168.2.22,192.168.2.105’。
  - IP 地址范围；用减号分隔的地址上下界，例如’192.168.2.20-192.168.2.29’。
  - 任意 IP 地址：关键字’ANY’。
- opt_user_quotas：设置用户在数据库中各种资源上的配额限制。
- quantity_unit：设置数值的单位。例如要设置的资源配额为临时表空间，则单位可以为 M、G 等。
- encryptor_name：加密机（即加密用的密钥）的名称。

23.3.2 资源配额限制 opt_user_quotas

语法格式

```
opt_user_quotas::=
QUOTA int_value quantity_unit ON MEMORY
|   QUOTA UNLIMITED ON MEMORY
|   QUOTA int_value quantity_unit ON TEMP TABLESPACE
|   QUOTA UNLIMITED ON TEMP TABLESPACE
|   QUOTA int_value quantity_unit ON UNDO TABLESPACE
|   QUOTA UNLIMITED ON UNDO TABLESPACE
|   QUOTA int_value ON CURSOR
|   QUOTA UNLIMITED ON CURSOR
|   QUOTA int_value ON SESSION
|   QUOTA UNLIMITED ON SESSION
|   QUOTA int_value ON IO
|   QUOTA UNLIMITED ON IO
|   QUOTA int_value ON PROCEDURE
|   QUOTA UNLIMITED ON PROCEDURE
|   QUOTA int_value quantity_unit ON TABLESPACE
|   QUOTA UNLIMITED ON TABLESPACE
```

参数说明

- int_value：设置的数值，取值为整数。
- quantity_unit：设置数值的单位。例如要设置的资源配额为临时表空间，则单位可以为 M、G 等。
- UNLIMITED：无限制。
- MEMORY：内存配额。
- TEMP TABLESPACE：临时表空间配额。
- UNDO TABLESPACE：回滚表空间配额。

- CURSOR：游标配额。
- SESSION：会话配额。
- IO：I/O 配额。
- PROCEDURE：存储过程配额。
- TABLESPACE：表空间配额。

23.3.3 示例

修改名为 usr_test 的用户的信息。

```
SQL> ALTER USER usr_test
IDENTIFIED BY 'abcPAS135@#'
VALID UNTIL '2099-12-31 12:00:00'
ACCOUNT UNLOCK
'192.168.2.20-192.168.2.30';
```

23.4 删除用户

语法格式

```
DROP USER user_name [alter_behavior];
```

> ⚠ 注意
>
> 删除用户后，则所有属主为它的数据库对象均会被删除，所以删除用户前需慎重考虑或对其相关数据库对象进行备份。

参数解释

- user_name：待删除的用户名。
- alter_behavior：可选关键字 RESTRICT（默认值）或 CASCADE。
  - RESTRICT：删除用户时，只有在该用户及其对象没有被其他用户或模式对象依赖，才能成功删除。如果用户拥有其他被依赖对象，数据库返回错误，提示无法删除用户。
  - CASCADE：删除用户时，无论该用户及其对象是否被其他用户或模式对象依赖，都强制删除用户。

示例

强制删除用户 usr_test。

```
SQL> DROP USER usr_test CASCADE;
```

24 角色管理

24.1 概述

角色是数据库中一组权限的集合。

角色可以被授予用户，此时该用户便有了角色所规定的权限；角色可以从被授予的用户中收回，此时该用户便失去了被收回角色所规定的权限。

数据库的每个（逻辑）库中，都存在一些缺省的角色，其权限已被预先定义好。除此外的其他角色，需要有权限的用户手动管理，如创建、授予、收回和删除角色。

24.2 创建角色

语法格式

```
CreateRoleStmt::=
CREATE ROLE role_name [ { INIT | INITIALLY } user_name_1[,
   user_name_2 [, ...] ] ]
```

参数说明

- role_name: 要创建的角色名。
- user_name_1, user_name_2, ... : 用户名。用于在创建角色的同时，指定该角色被授予的用户。

示例

创建角色 role_1 并授予用户 usr_1 和用户 usr_2。

```
SQL> CREATE ROLE role_1 INIT USER usr_1, usr_2;
```

24.3 授予角色

语法格式

```
AlterRoleStmt::=
GRANT ROLE role_name TO user_name_1[, user_name_2 [, ...] ]
```

参数说明

- role_name: 要授予的角色名。

- user_name_1, user_name_2, ... : 角色被授予的用户名。

示例

将已创建的角色 role_1 授予用户 usr_1 和用户 usr_2。

```
SQL> GRANT ROLE role_1 TO usr_1, usr_2;
```

24.4 收回角色

语法格式

```
AlterRoleStmt::=
REVOKE ROLE role_name FROM user_name_1[, user_name_2 [, ...] ]
```

参数说明

- role_name: 要收回的角色名。
- user_name_1, user_name_2, ... : 角色被收回的用户名。其中角色能被收回的前提是，这些用户已经拥有了该角色，否则会报错。

示例

收回用户 usr_1 和用户 usr_2 的角色 role_1。

```
SQL> REVOKE ROLE role_1 FROM usr_1, usr_2;
```

24.5 删除角色

语法格式

```
DropRoleStmt::=
    DROP ROLE role_name
```

参数说明

- role_name: 要删除的角色名。

示例

```
SQL> DROP ROLE role_1;
```

25 DBLink 管理

25.1 概述

DBLink 功能实现了跨数据源访问的能力，用户可以在本地数据库上访问远端数据库。

使用说明

使用 DBLink 对远端数据库进行访问，有以下使用说明和注意事项。

- 支持访问的远端数据库类型如下：
  - 虚谷数据库
  - Oracle
  - MySQL
  - PostgreSQL
  - 符合 MS ODBC 3.X 标准的其他异构库
- 支持复杂 SQL 中多个远程表的操作，如 TPCH 的 24 条语句。
- 支持虚谷数据库数据类型的查询与 DML 操作。
- 支持 Prepare 语句对 DBLink 查询进行预处理。
- 不支持对 DBLink 表进行 DDL 操作，如 DROP TABLE，ALTER TABLE 等操作。
- 不支持分区查询。
- DML 不支持 Prepare 参数化执行。
- 虚谷数据库本地没有对远程表进行缓存的功能。

> ⚠ 注意
>
> 在使用 DBLink 时，尽量使用 ODBC 标准包含的数据类型，并验证类型是否能正确映射。如果虚谷数据库与远程库之间的数据类型无法正确映射，那么将抛出用户异常，终止当前命令。

权限说明

创建 DBLink 时，指定 PUBLIC，则创建对所有用户可见的公共数据库链接。如果省略，则创建私有数据库链接。

- 公共数据库链接：具有 DBLink 创建权限的用户，创建 DBLink 实例时，指定 PUBLIC 关键字，创建的 DBLink 实例为公有数据库链接，允许其它用户访问。
- 私有数据库链接：具有 DBLink 创建权限的用户，创建 DBLink 实例时，缺省 PUBLIC 关键字，创建 DBLink 实例为私有数据库链接，只允许当前创建 DBLink 的用户才能访问。

用户权限	共有 DBLink	私有 DBLink
普通无权限用户	不支持操作	不支持操作
仅有创建私有 DBLink 权限	不支持操作	支持创建和删除
仅有创建公有 DBLink 权限	支持创建，不支持删除	不支持操作
同时有创建私有公有 DBLink 权限	支持创建，不支持删除	支持创建和删除
仅有删除公有 DBLink 权限	不支持创建，支持删除	不支持操作

⚠ 注意

- 具有创建 DBLink 权限，无删除 DBLink 权限，只能删除自己创建的私有 DBLink。
- 用户删除后，用户拥有的私有 DBLink 同步删除，公有不删除。
- 除创建用户以外的所有用户均不能删除私有 DBLink。

25.2 创建 DBLink

创建一个访问指定远端数据库的 DBLink，需要指定 DBLink 名称并且提供远端数据库的用户名、密码、IP 地址、端口号以及访问类型。

⚠ 注意

- 创建 DBLink 的用户需具备 CREATE ANY DATABASE LINK 权限。
- 不支持连接自身实例的 DBLink。

语法格式

```
CreateLinkStmt::=
CREATE [PUBLIC] DATABASE LINK name [FOR Sconst] [CONNECT TO Sconst
   ] USER username IDENTFIED BY password [USING SCONST];
```

参数解释

- PUBLIC：此链接对象是否能被创建者之外的用户使用，缺省则为私有。
- name：指定 DBLink 的名称。
- FOR Sconst：远端数据库类型。

  FOR 为关键字，Sconst 为远程连接的数据库名称。缺省，默认连接的是虚谷数据库，使用 XGCI。

  如果使用 ODBC 连接虚谷数据库，Sconst 需要给定 OXuGu 名字。异构库类型名称可通过系统表 SYS_DBLINK_SP_TYPES 查看。
- CONNECT TO Sconst：CONNECT TO 为关键字，远程连接 url，格式为 db_name@ip:port。
- username：远程连接数据库的用户名。
- password：远程数据库用户的登录密码。
- USING SCONST：USING 为关键字，SCONST 为数据源名称，即 odbc.ini 文件中配置的数据源。如果不使用 ODBC 连接虚谷数据库，则缺省。

⚠ 注意

- XGCI 方式连接：需正确配置 CONNECT TO Sconst 字段，USING SCONST 字段缺省。
- ODBC 方式连接：需正确配置 USING SCONST 字段，配置为本端环境中 ODBC 配置 odbc.ini 中的数据源名称。CONNECT TO Sconst 字段无效，可缺省。

示例

- 示例 1

  使用 ODBC 方式，创建一个连接到其他服务器上的虚谷数据库的外部链接，参数如下所示：

  - 远端数据库名为 DB_LINK_TEST。
  - IP 地址为 192.168.121.102。
  - 登录到远端的用户名为 SYSDBA。
  - 登录口令为 SYSDBA。
  - 外部字符串 xugu_102，对应 odbc.ini 中的数据源名称。

```
CREATE PUBLIC DATABASE LINK link_xg FOR OXUGU CONNECT TO '
   DB_LINK_TEST@192.168.121.102:5138' USER 'SYSDBA' IDENTIFIED
    BY 'SYSDBA' USING 'xugu_102' ;
```

- 示例 2

  创建一个连接到其他服务器上的 Oracle 数据库的外部链接，参数如下所示：

  - IP 地址为 192.168.121.128。
  - 登录到远端的用户名为 u1。
  - 登录口令为 123456。

```
CREATE PUBLIC DATABASE LINK link_orcl FOR ORACLE CONNECT TO '
   orcl@192.168.121.128:1521' USER 'u1' IDENTIFIED BY '123456'
    USING 'linux_orcl_155' ;
```

- 示例 3

  使用 XGCI 方式，创建一个连接到其他服务器上的虚谷数据库的外部链接，参数如下所示：

  - 远端数据库名为 DB_LINK_TEST。
  - IP 地址为 192.168.121.102:5138。
  - 登录到远端的用户名为 u1。
  - 登录口令为 123456。

```
CREATE PUBLIC DATABASE LINK link_xg CONNECT TO '
   DB_LINK_TEST@192.168.121.102:5138' USER 'SYSDBA' IDENTIFIED
    BY 'SYSDBA' ;
```

25.3 查看 DBLink

创建 DBLink 后，可以通过 SYS_DBLINKS、DBA_DBLINKS 视图来查看已创建的 DBLink 信息。

示例

- 示例 1

  查看已创建的 DBLink 信息。

```
SQL> SELECT * FROM SYS_DBLINKS;

DB_ID | DBLK_ID | DB_TYPE | DBLK_NAME | IP | PORT | RDB_NAME |
   USER_NAME | CONNECT_STR | VALID | IS_PUBLIC | OWNER_ID |
-------------------------------------------------------------
1|1048578|      3|LINK_ORCL  |192.168.121.128|1521|orcl        |
   u1       |linux_orcl_155|     |true     |       0|
```

  参数解释：

  - DB_ID：库 ID
  - DBLK_ID：DBLink 的 ID
  - DB_TYPE：远端数据库类型
  - DBLK_NAME：DBLink 名称
  - IP：远端数据库的主机 IP 地址
  - PORT：远端数据库的端口
  - RDB_NAME：远端库名
  - USERNAME：连接远端数据库时使用的用户名
  - CONNECT_STR：连接字符串
  - VALID：是否有效
  - IS_PUBLIC：是否是公有外部链接
  - OWNER_ID：拥有者的用户 ID

- 示例 2

  通过系统表查看远端数据库类型 DB_TYPE。

```
SQL> SELECT * FROM SYS_DBLINK_SP_TYPES;
```

```
TYPE_ID|NAME      |ALIAS_NAME|FIRST_VERSION|LAST_VERSION|
-------+----------+----------+-------------+------------+
1|XuGu      |          |             |            |
2|OXuGu     |          |             |            |
3|Oracle    |          |             |            |
4|SQLServer |MSSQL     |             |            |
5|MySQL     |          |             |            |
6|DB2       |          |             |            |
7|PostgreSQL|          |             |            |
8|SQLITE    |          |             |            |
9|MARIADB   |          |             |            |
```

25.4 使用 DBLink

25.4.1 访问远端数据库中的数据

通过 DBLink 访问远端数据库的对象，对象包括表、视图、同义词、序列、索引、约束、子查询等。

访问远端数据库的对象，使用方式是在对象名后用 @ 符号连接 DBLink 名。

示例

使用外部链接 LINK1 查询远程表 tbl。

```
SELECT * FROM tbl@LINK1;
```

25.4.2 修改远端数据库中的数据

通过 DBLink 修改远端数据库中数据，包括插入数据、更新数据和删除数据等操作，主要通过 INSERT、UPDATE 和 DELETE 三种 DML 语句实现。

示例

- 示例 1 向远程表 tbl 插入数据。

```
INSERT INTO tbl@LINK1 VALUES(1,2);
```

- 示例 2 查询本地表或其他链接的表对远程表进行操作。

```
UPDATE tbl@LINK1 SET C1 = C1*2 WHERE C2 NOT IN (SELECT ID FROM
   tbl);
DELETE FROM tbl@LINK1 WHERE C1 IN (SELECT ID FROM T2@LINK2);
```

- 示例 3 使用 INSERT INTO SELECT 语句，复制一张本地表 tbl 信息到远程表 tbl@dblk。

```
INSERT INTO tbl@dblk SELECT id+100,c2,c3 FROM tbl;
```

25.5 删除 DBLink

可根据业务需求进行删除 DBLink 操作。

⚠ 注意

- 执行该语句的用户需要具备 DROP ANY DATABASE LINK 权限。
- 删除不存在的链接会返回错误。

语法格式

```
DROP DATABASE LINK dblink_name;
```

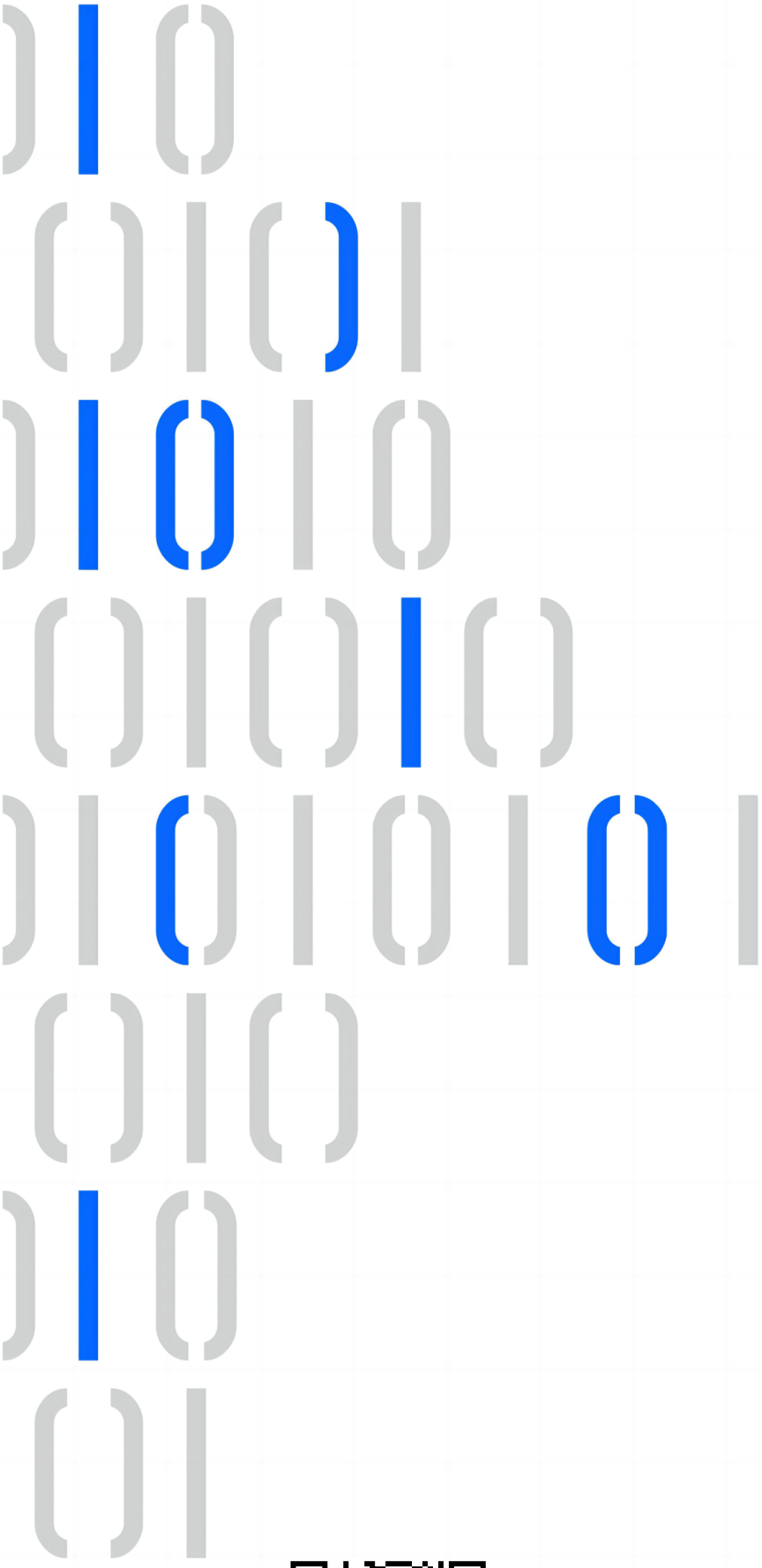
参数解释

dblink_name：待操作的数据库链接名称。

示例

删除数据库链接 DBLINK1。

```
DROP DATABASE LINK DBLINK1;
```



成都虚谷伟业科技有限公司

联系电话：400-8886236

官方网站：www.xugudb.com