# 视图

## 什么是视图

视图是一种数据库对象，是从一个或者多个数据表或视图中导出的虚表，视 图所对应的数据并不真正地存储在视图中，而是存储在所引用的数据表中，视图的结构和数据是对数据表进行查询的结果。

根据创建视图时给定的条件，视图可以是一个数据表的一部分，也可以是多个基表的联合，它存储了要执行检索的查询语句的定义，以便在引用该视图时使用。

使用视图的优点：

1.简化数据操作：视图可以简化用户处理数据的方式。

2.着重于特定数据：不必要的数据或敏感数据可以不出现在视图中。

3.视图提供了一个简单而有效的安全机制，可以定制不同用户对数据的访问权限。

4.提供向后兼容性：视图使用户能够在表的架构更改时为表创建向后兼容接口。

## 创建或修改视图语法

|  |
| --- |
| CREATE [OR REPLACE] [FORCE] VIEW view\_name  AS subquery  [WITH CHECK OPTION ]  [WITH READ ONLY] |

选项解释：

|  |  |
| --- | --- |
| OR REPLACE | 若所创建的试图已经存在，ORACLE 自动重建该视图 |
| FORCE | 不管基表是否存在 ORACLE 都会自动创建该视图 |
| subquery | 一条完整的 SELECT 语句，可以在该语句中定义别名； |
| WITH CHECK OPTION | 插入或修改的数据行必须满足视图定义的约束； |
| WITH READ ONLY | 该视图上不能进行任何 DML 操作 |

## 删除视图语法

|  |
| --- |
| DROP VIEW view\_name |

## 案例

### 简单视图的创建与使用

什么是简单视图？如果视图中的语句只是单表查询，并且没有聚合函数，我们就称之为简单视图。

需求：创建视图：业主类型为 1 的业主信息

语句：

|  |
| --- |
| create or replace view view\_owners1 as  select \* from t\_owners where ownertypeid=1 |

利用该视图进行查询

|  |
| --- |
| select \* from view\_owners1 where addressid=1; |

就像使用表一样去使用视图就可以了。

对于简单视图，我们不仅可以用查询，还可以增删改记录。

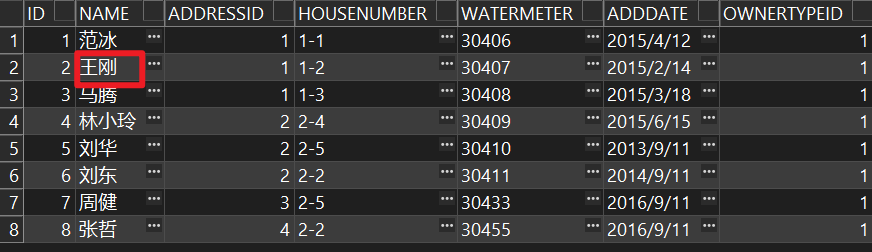
我们下面写一条更新的语句，试一下：

|  |
| --- |
| update view\_owners1 set name='王刚' where id=2; |

再次查询：

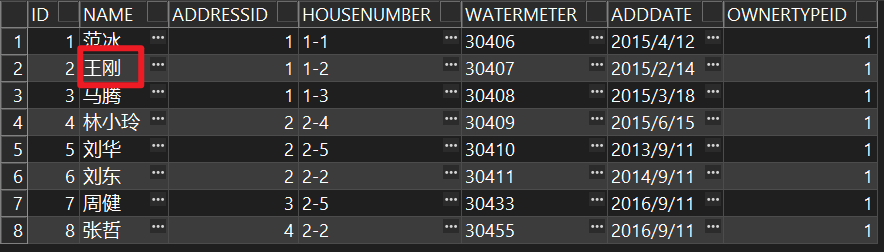
|  |
| --- |
| select \* from view\_owners1; |

查询结果如下：



结果已经更改成功。

我们再次查询表数据



发现表的数据也跟着更改了。由此我们得出结论：视图其实是一个虚拟的表，它的数据其实来自于表。如果更改了视图的数据，表的数据也自然会变化，更改了表的数据，视图也自然会变化。一个视图所存储的并不是数据，而是一条 SQL 语句。

### 带检查约束的视图

需求：根据地址表（ T\_ADDRESS）创建视图 VIEW\_ADDRESS2 , 内容为区域 ID 为 2 的记录。

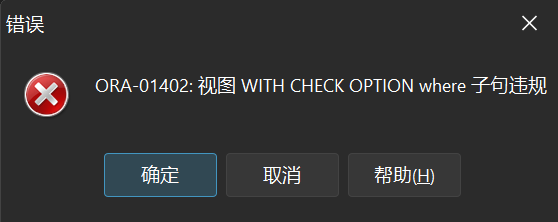
语句：

|  |
| --- |
| create or replace view view\_address2 as  select \* from T\_ADDRESS where areaid=2  with check option |

执行下列更新语句：

|  |
| --- |
| update view\_address2 set areaid=1 where id=4 |

系统提示如下错误信息：



### 只读视图的创建与使用

如果我们创建一个视图，并不希望用户能对视图进行修改，那我们就需要创建视

图时指定 WITH READ ONLY 选项，这样创建的视图就是一个只读视图。

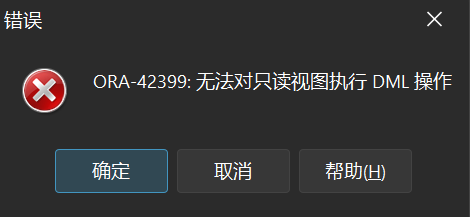
需求：将上边的视图修改为只读视图

语句：

|  |
| --- |
| create or replace view view\_owners1 as  select \* from T\_OWNERS where ownertypeid=1  with read only |

修改后，再次执行 update 语句，会出现如下错误提示

|  |
| --- |
| update view\_owners1 set name='和珅' where id=2; |

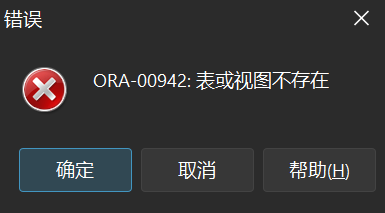


### 创建带错误的视图

我们创建一个视图，如果视图的 SQL 语句所设计的表并不存在，如下

|  |
| --- |
| create or replace view view\_temp as  select \* from t\_temp |

T\_TEMP 表并不存在，此时系统会给出错误提示



有的时候，我们创建视图时的表可能并不存在，但是以后可能会存在，我们如果

此时需要创建这样的视图，需要添加 FORCE 选项，SQL 语句如下：

|  |
| --- |
| create or replace **force** view view\_temp as  select \* from t\_temp |

此时视图创建成功。

### 复杂视图的创建与使用

所谓复杂视图，就是视图的 SQL 语句中，有聚合函数或多表关联查询。

我们看下面的例子：

#### 多表关联查询的例子

需求：创建视图，查询显示业主编号，业主名称，业主类型名称

语句：

|  |
| --- |
| create or replace view view\_owners as  select o.id 业主编号, o.name 业主名称, ot.name 业主类型  from t\_owners o, t\_ownertype ot  where o.ownertypeid=ot.id |

使用该视图进行查询

|  |
| --- |
| select \* from view\_owners; |

那这个视图能不能去修改数据呢？

我们试一下下面的语句：

|  |
| --- |
| update view\_owners set 业主名称='范小冰' where 业主编号=1 |

可以修改成功。

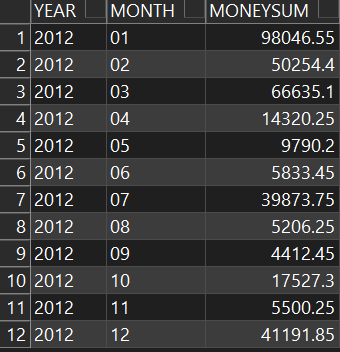
我们再试一下下面的语句：

|  |
| --- |
| update view\_owners set 业主类型='普通居民' where 业主编号=1; |

可以修改成功。

#### 分组聚合统计查询的例子

需求：创建视图，按年月统计水费金额，效果如下



语句：

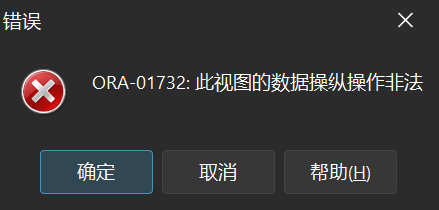
|  |
| --- |
| create view view\_accountsum as  select year,month,sum(money) moneysum  from t\_account  group by year, month  order by year, month |

那这个视图能不能去修改数据呢？

|  |
| --- |
| update view\_accountsum set year='2023' where month='01' |

发现修改失败。

此例用到聚合函数，没有键保留表，所以无法执行 update 。



# 物化视图

## 什么是物化视图

视图是一个虚拟表（也可以认为是一条语句），基于它创建时指定的查询语句返回的结果集。每次访问它都会导致这个查询语句被执行一次。为了避免每次访问都执行这个查询，可以将这个查询结果集存储到一个物化视图（也叫实体化视图）。

物化视图与普通的视图相比的区别是物化视图是建立的副本，它类似于一张表，需要占用存储空间。

而对一个物化视图查询的执行效率与查询一个表是一样的。

## 创建物化视图语法

|  |
| --- |
| create materialized view view\_name  [build immediate | build deferred ]  refresh [fast|complete|force]  [  on [commit | demand ] | start with (start\_time) next  (next\_time)  ]  as  subquery |

* BUILD IMMEDIATE 是在创建物化视图的时候就生成数据
* BUILD DEFERRED 则在创建时不生成数据，以后根据需要再生成数据。
  + 默认为 BUILD IMMEDIATE。
* 刷新（ REFRESH ）：指当基表发生了 DML 操作后，物化视图何时采用哪种方式和基表进行同步。
* REFRESH 后跟着指定的刷新方法有三种：FAST、COMPLETE、FORCE。
  + FAST 刷新采用增量刷新，只刷新自上次刷新以后进行的修改。
  + COMPLETE 刷新对整 个物化视图进行完全的刷新。
  + 如果选择 FORCE 方式，则 Oracle 在刷新时会去判断是否可以进行快速刷新，
    - 如果可以则采用 FAST 方式，否则采用 COMPLETE的方式
    - FORCE 是默认的方式。
  + 刷新的模式有两种：ON DEMAND 和 ON COMMIT 。
    - ON DEMAND 指需要 手动刷新物化视图（默认）。
    - ON COMMIT 指在基表发生 COMMIT 操作时自动刷新。

## 案例

### 创建手动刷新的物化视图

需求：查询地址 ID,地址名称和所属区域名称, 结果如下：



语句：

|  |
| --- |
| create materialized view mv\_address  as  select ad.id, ad.name adname, ar.name ar\_name  from t\_address ad,t\_area ar  where ad.areaid=ar.id |

执行上边的语句后查询

|  |
| --- |
| select \* from mv\_address |

查询结果如下：



这时，我们向地址表（ T\_ADDRESS）中插入一条新记录，

|  |
| --- |
| insert into t\_address values(8,'宏福苑小区',1,1); |

再次执行上边的语句进行查询，会发现新插入的语句并没有出现在物化视图中。

我们需要通过下面的语句（PL/SQL），手动刷新物化视图：

|  |
| --- |
| begin  dbms\_mview.refresh('mv\_address','C');  end; |

### 创建自动刷新的物化视图，和上例一样的结果集

需求: 当 T\_ADDRESS 表发生变化时，物化视图 自动跟着改变。

语句如下：

|  |
| --- |
| create materialized view mv\_address2  refresh  on commit  as  select ad.id,ad.name adname,ar.name ar\_name  from t\_address ad, t\_area ar  where ad.areaid=ar.id |

创建此物化视图后，当 T\_ADDRESS 表发生变化时，MV\_ADDRESS2 自动跟着改变。

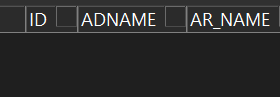
### 创建时不生成数据的物化视图

|  |
| --- |
| create materialized view mv\_address3  build deferred  refresh  on commit  as  select ad.id,ad.name adname,ar.name ar\_name  from t\_address ad,t\_area ar  where ad.areaid=ar.id; |

创建后执行下列语句查询物化视图

|  |
| --- |
| select \* from mv\_address3 |

查询结果：



执行下列语句生成数据

|  |
| --- |
| begin  dbms\_mview.refresh('MV\_ADDRESS3','C');  end; |

再次查询，得到结果：



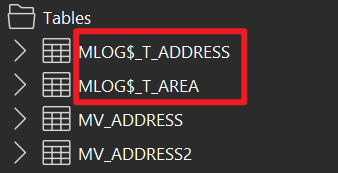
由于我们创建时指定的 on commit ,所以在修改数据后能立刻看到最新数据，无须再次执行 refresh

### 创建增量刷新的物化视图

如果创建增量刷新的物化视图，必须首先创建物化视图日志

|  |
| --- |
| create materialized view log on t\_address with rowid；  create materialized view log on t\_area with rowid; |

创建的物化视图日志名称为 MLOG$\_表名称



创建物化视图

|  |
| --- |
| create materialized view mv\_address4  refresh fast  as  select ad.rowid adrowid, ar.rowid arrowid, ad.id,ad.name adname,ar.name ar\_name  from t\_address ad, t\_area ar  where ad.areaid=ar.id; |

注意：创建增量刷新的物化视图，必须：

1. 创建物化视图中涉及表的物化视图日志。

2. 在查询语句中，必须包含所有表的 rowid ( 以 rowid 方式建立物化视图日志 )

当我们向地址表插入数据后，物化视图日志的内容：

|  |
| --- |
| insert into t\_address values(11, '天通苑北4区', 4, 5) |



SNAPTIME$$：用于表示刷新时间。

DMLTYPE$$：用于表示 DML 操作类型，I 表示 INSERT ，D 表示 DELETE ，U 表示 UPDATE。

OLD\_NEW$$：用于表示这个值是新值还是旧值。 N（EW）表示新值，O（ LD ） 表示旧值，U 表示 UPDATE 操作。

CHANGE\_VECTOR$$：表示修改矢量，用来表示被修改的是哪个或哪几个字段。

此列是 RAW 类型，其实 Oracle 采用的方式就是用每个 BIT 位去映射一个列。

插入操作显示为：FE, 删除显示为：OO 更新操作则根据更新字段的位置而显示不同的值。

当我们手动刷新物化视图后，物化视图日志被清空，物化视图更新。

|  |
| --- |
| begin  DBMS\_MVIEW.refresh('MV\_ADDRESS4','C');  end; |

# 序列

## 什么是序列

序列是 ORACLE 提供的用于产生一系列唯一[数字的数据库对](http://lib.csdn.net/base/mysql)象。

## 创建与使用简单序列

创建序列语法：

|  |
| --- |
| create sequence 序列名称 |

通过序列的伪列来访问序列的值

**NEXTVAL** 返回序列的下一个值

**CURRVAL** 返回序列的当前值

注意：我们在刚建立序列后，无法提取当前值，只有先提取下一个值时才能再次提取当前值。

提取下一个值

|  |
| --- |
| select 序列名称.nextval from dual |

提取当前值

|  |
| --- |
| select 序列名称.currval from dual |

## 创建复杂序列

语法：

|  |
| --- |
| CREATE SEQUENCE sequence //创建序列名称  [INCREMENT BY n] //递增的序列值是 n 如果 n 是正数就递增,如果是负数就递减 默认是 1  [START WITH n] //开始的值,递增默认是 minvalue 递减是 maxvalue  [{MAXVALUE n | NOMAXVALUE}] //最大值  [{MINVALUE n | NOMINVALUE}] //最小值  [{CYCLE | NOCYCLE}] //循环/不循环  [{CACHE n | NOCACHE}];//分配并存入到内存中 |

## 案例

### 有最大值的非循环序列

创建序列的语句：

|  |
| --- |
| create sequence seq\_test1  increment by 10  start with 10  maxvalue 300  minvalue 20 |



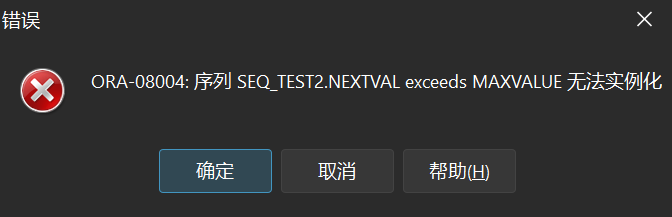
以上的错误，是由于我们的开始值小于最小值 。开始值不能小于最小值，修改

以上语句：

|  |
| --- |
| create sequence seq\_test1  increment by 10  start with 10  maxvalue 300  minvalue 5 |

我们执行下列语句提取序列值，当序列值为 300（最大值）的时候再次提取值，

系统会报异常信息。



### 有最大值的循环序列

|  |
| --- |
| create sequence seq\_test2  increment by 10  start with 10  maxvalue 300  minvalue 5  cycle |

当序列当前值为 300（最大值），再次提取序列的值

|  |
| --- |
| select seq\_test2.nextval from dual;  select seq\_test2.currval from dual; |

提取的值为：

由此我们得出结论，循环的序列，第一次循环是从开始值开始循环，而第二次循环是从最小值开始循环。

思考问题：

下列语句是否会报错？为什么？

|  |
| --- |
| create sequence seq\_test3  increment by 10  start with 10  minvalue 5  cycle |

答：此为错误的语句。因为你创建的是一个循环的序列，所以必须指定最大值，否则会报错。

### 带缓存的序列

我们执行下列语句：

|  |
| --- |
| create sequence seq\_test3  increment by 10  start with 10  maxvalue 300  minvalue 5  cycle  cache 50 |

我们执行上边语句的意思是每次取出 50 个缓存值，但是执行会提示错误



上边错误提示的意思是：缓存设置的数必须小于每次循环的数。

我们缓存设定的值是 50，而最大值是 300，那么为什么还会提示这样的信息呢？

其实我们的 cache 虽然是 50，但是我们每次增长值是 10。这样 50 次缓存提取出的数是 500 （50\*10）

我们更改为下列的语句：

|  |
| --- |
| create sequence seq\_test4  increment by 10  start with 10  maxvalue 500  minvalue 10  cycle  cache 50; |

下列语句依然会提示上边的错误，这是因为还存在一个 minvalue，minvalue 和 maxvalue 之间是 49 个数，也就是一次循环可以提取 490，但是我们的缓存是 500。

我们再次修改语句：

|  |
| --- |
| create sequence seq\_test5  increment by 10  start with 10  maxvalue 500  minvalue 9  cycle  cache 50 |

把最小值减 1，或把最大值加 1 ，都可以通过。

## 修改和删除序列

修改序列：使用 ALTER SEQUENCE 语句修改序列，不能更改序列的 START

WITH 参数

|  |
| --- |
| alter sequence 序列名称 maxvalue 5000 cycle; |

删除序列：

|  |
| --- |
| DROP SEQUENCE 序列名称; |

# 同义词

## 什么是同义词

同义词实质上是指定方案对象的一个别名。通过屏蔽对象的名称和所有者以 及对分布式数据库的远程对象提供位置透明性，同义词可以提供一定程度的安全

性。同时，同义词的易用性较好，降低了数据库用户的 SQL 语句复杂度。

同义词允许基对象重命名或者移动，这时，只需对同义词进行重定义，基于同义词的应用程序可以继续运行而无需修改。

你可以创建公共同义词和私有同义词。其中，公共同义词属于 PUBLIC 特殊 用户组，数据库的所有用户都能访问；而私有同义词包含在特定用户的方案中，

只允许特定用户或者有基对象访问权限的用户进行访问。

同义词本身不涉及安全，当你赋予一个同义词对象权限时，你实质上是在给同义词的基对象赋予权限，同义词只是基对象的一个别名。

## 创建与使用同义词

创建同义词的具体语法是：

|  |
| --- |
| create [public] SYNONYM synooym for object; |

其中 synonym 表示要创建的同义词的名称，object 表示表，[视图](http://www.chinabyte.com/keyword/%E8%A7%86%E5%9B%BE/)，序列等我们要创建同义词的对象的名称。

## 案例

### 私有同义词

需求：为表 T\_OWNERS 创建( 私有 )同义词 名称为 OWNERS

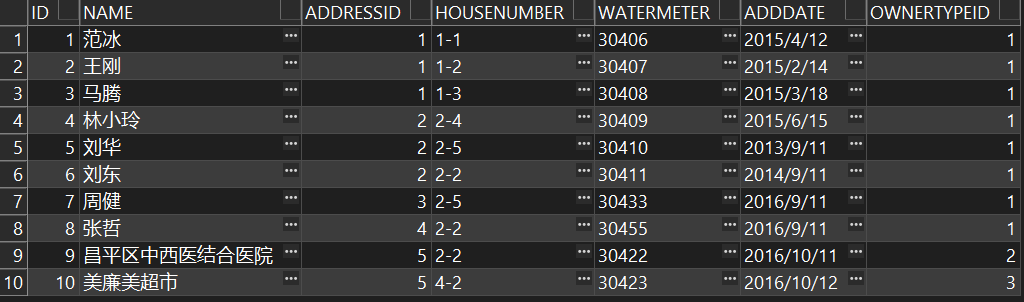
语句：

|  |
| --- |
| create synonym owners for t\_owners; |

使用同义词：

|  |
| --- |
| select \* from owners; |

查询结果如下：



### 公有同义词

需求：为表 T\_OWNERS 创建( 公有 )同义词 名称为 OWNERS2：

|  |
| --- |
| create public synonym owners2 for t\_owners; |

以另外的用户登陆，也可以使用公有同义词：

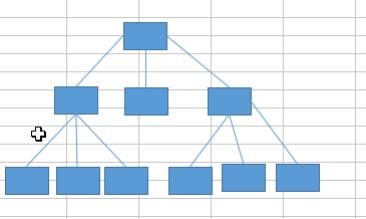
|  |
| --- |
| select \* from owners; |

# 索引

## 什么是索引

索引是用于加速数据存取的数据对象。合理的使用索引可以大大降低 i/o 次数,从而提高数据访问性能。

索引是需要占据存储空间的，也可以理解为是一种特殊的数据。形式类似于 下图的一棵“树”，而树的节点存储的就是每条记录的物理地址，也就是我们提到的伪列（ROWID）



## 普通索引

语法：

|  |
| --- |
| create index 索引名称 on 表名(列名); |

需求：我们经常要根据业主名称搜索业主信息，所以我们基于业主表的 name 字

段来建立索引。语句如下：

|  |
| --- |
| create index index\_owners\_name on t\_owners(name) |

索引性能测试：

创建一个两个字段的表

|  |
| --- |
| create table t\_indextest (  id number,  name varchar2(30)  ); |

编写 PL/SQL 插入 100 万条记录（关于 PL/SQL 我们在第四章会学到）

|  |
| --- |
| begin  for i in 1 .. 1000000 loop  insert into t\_indextest values (i, 'AA' || i);  end loop;  commit;  end; |

创建完数据后，根据 name 列创建索引

|  |
| --- |
| create index index\_testindex on t\_indextest(name); |

执行下面两句 SQL 执行

|  |
| --- |
| SELECT \* from T\_INDEXTEST where ID=765432;  SELECT \* from T\_INDEXTEST where NAME= 'AA765432'; |

我们会发现根据 name 查询所用的时间会比根据 id 查询所用的时间要短

## 唯一索引

如果我们需要在某个表某个列创建索引，而这列的值是不会重复的。这是我们可以创建唯一索引。

语法：

|  |
| --- |
| create unique index 索引名称 on 表名(列名); |

需求：在业主表的水表编号一列创建唯一索引

语句：

|  |
| --- |
| create unique index index\_owners\_watermeter on t\_owners(watermeter); |

## 复合索引

我们经常要对某几列进行查询，比如，我们经常要根据学历和性别对学员进行搜 索，如果我们对这两列建立两个索引，因为要查两棵树，查询性能不一定高。那 如何建立索引呢？我们可以建立复合索引，也就是基于两个以上的列建立一个索引 。

语法：

|  |
| --- |
| create index 索引名称 on 表名(列名,列名.....); |

根据地址和门牌号对学员表创建索引，语句如下：

|  |
| --- |
| create index owners\_index\_ah on t\_owners(addressid, housenumber); |