# 第 4 章　数据字典

Oracle 通过数据字典来管理和展现数据库信息，数据字典通常存储数据库的元数据，是数据库的“数据库”，其中存储的信息至关重要。正确理解这部分内容会有助于提高大家对 Oracle 数据库的认知，加强自学习能力。本章对 Oracle 的数据字典进行探讨。

## 4.1 数据字典概述

数据字典（Data Dictionary）是 Oracle 数据库的一个重要组成部分，是元数据（Metadata）的存储地点。Oracle RDBMS 使用数据字典记录和管理对象信息和安全信息等，用户和数据库系统管理员可以通过数据字典来获取数据库相关信息。

数据字典包括以下内容：

- 所有数据库 Schema 对象的定义(表、视图、索引、聚簇、同义词、序列、过程、函数、包、触发器等等)
- 数据库的空间分配和使用情况
- 字段的缺省值
- 完整性约束信息
- Oracle 用户名称、角色、权限等信息
- 审计信息
- 其他数据库信息

总之，数据字典是数据库核心，通过数据字典，Oracle 数据库基本上可以实现自解释。

一般来说，数据字典是只读的，通常不建议对任何数据字典表中的任何信息进行手工更新或改动，**对于数据字典的修改很容易就会导致数据库紊乱，造成无法恢复的后果**，而且 Oracle 公司不对此类操作带来的后果负责。

通常所说的数据字典由四部分组成：内部 RDBMS（X $）表、数据字典表、动态性能(V$)视图和数据字典视图。作为数据字典的辅助管理，还可以为对象创建同义词。

## 4.2 内部 RDBMS（X$）表

X$表是 Oracle 数据库的核心部分，这些表用于跟踪内部数据库信息，维持数据库的正常运行。X$表是加密命名的，而且 Oracle 不作文档说明，这部分知识是 Oracle 公司的技术机密，Oracle 通过这些 X$建立起其他大量视图提供用户查询管理数据库之用。但是由于 X$表记录了大量有用的信息，所以也不停的被全球 DBA 不懈的探索着，最为人所熟知的有:X$BH，X$KSMSP 等。

X$表的本质上是一系列的 C 结构体，是 Oracle 数据库的运行基础，在数据库启动时由 Oracle 应用程序动态创建。这部分表对数据库来说至关重要，所以 Oracle 不允许 SYSDBA 之外的用户直接访问，显示授权不被允许。

如果显式授权会收到如下错误:

```
SQL> grant select on x$ksppi to eygle;
grant select on x$ksppi to eygle
                *
ERROR at line 1:
ORA-02030: can only select from fixed tables/views
```

Oracle 的解释是:

```
ORA-02030 can only select from fixed tables/views
Cause: An attempt is being made to perform an operation other than a retrieval from a fixed table/view.
Action: You may only select rows from fixed tables/views.
```

一句话，这些对象你最好只是查询。

发现、观察、研究 X$表的一个好办法是借用 Oracle 的 AUTOTRACE 功能，当我们查询一些常用视图的时候，我们可以通过 AUTOTRACE 功能发现这些 View 的底层表。

以下是 Oracle10gR2 中的一个示例：

```
SQL> set autotrace trace explain
SQL> select * from v$parameter;
Execution Plan
----------------------------------------------------------
Plan hash value: 1128103955
------------------------------------------------------------------------------
| Id  | Operation          | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |          |     1 |   926 |     1 (100)| 00:00:01 |
|*  1 |  HASH JOIN         |          |     1 |   926 |     1 (100)| 00:00:01 |
|*  2 |   FIXED TABLE FULL| X$KSPPI   |     1 |   249 |     0   (0)| 00:00:01 |
|   3 |   FIXED TABLE FULL| X$KSPPCV |   100 | 67700 |     0   (0)| 00:00:01 |
------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("X"."INDX"="Y"."INDX")
       filter(TRANSLATE("KSPPINM",'_','#') NOT LIKE '#%' OR
              "KSPPSTDF"='FALSE' OR BITAND("KSPPSTVF",5)>0)
   2 - filter("X"."INST_ID"=USERENV('INSTANCE') AND
              TRANSLATE("KSPPINM",'_','#') NOT LIKE '##%')
```

这些研究和探索是极有趣味的，如果你能就此深入下去，一定能够时常发现意外的收获。

顺便介绍一个有意思的 X$表，也许你曾经关注过：X$KVIT 。

其名称含义为：

X$KVIT-[**K**]ernel Layer Performance Layer [**V**] [**I**]nformation tables

[**T**]ransitory Instance parameters

这个视图记录的是和实例相关的一些内部参数设置，你可以看到一些很有意思的内容：

```
SQL> select * from v$version where rownum <2;
BANNER
----------------------------------------------------------------
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Prod
SQL> select kvittag,kvitval,kvitdsc from x$kvit;
KVITTAG            KVITVAL KVITDSC
---------------- ---------- -------------------------------------------------------
ksbcpu                   4 number of logical CPUs in the system used by Oracle
ksbcpucore               0 number of physical CPU cores in the system used by Oracle
ksbcpusocket             2 number of physical CPU sockets in the system used by Oracle
ksbcpu_hwm               4 high water mark of number of CPUs used by Oracle
ksbcpucore_hwm           0 high water mark of number of CPU cores on system
ksbcpusocket_hwm         2 high water mark of number of CPU sockets on system
ksbcpu_actual            4 number of available CPUs in the system
ksbcpu_dr                1 CPU dynamic reconfiguration supported
kcbnbh              238518 number of buffers
kcbldq                  25 large dirty queue if kcbclw reaches this
kcbfsp                  40 Max percentage of LRU list foreground can scan for free
kcbcln                   2 Initial percentage of LRU list to keep clean
kcbnbf                 750 number buffer objects
kcbwst                   0 Flag that indicates recovery or db suspension
kcteln                   0 Error Log Number for thread open
kcvgcw                   0 SGA: opcode for checkpoint cross-instance call
kcvgcw                   0 SGA:opcode for pq checkpoint cross-instance call

17 rows selected
```

不知道大家是否还记得，触发后台进程 DBWR 写动作的条件包含这样两个：

1．脏缓冲（Dirty Buffers）阈值（threshold）达到

-那么这个 threshold 是多少呢？从以上视图中可以知道，这个值是 25%,即：

```
kcbldq                  25 large dirty queue if kcbclw reaches this
```

这个限制同时受到一个隐含参数的控制（来自 Oracle9iR2 数据库）：

```
SQL> @GetHidPar
Enter value for par: db_writer_scan_depth_pct
```

```
old   6:    AND x.ksppinm LIKE '%&par%'
new   6:    AND x.ksppinm LIKE '%db_writer_scan_depth_pct%'

NAME                         VALUE DESCRIB
------------------------ ----- --------------------------------------------------------
_db_writer_scan_depth_pct 25    Percentage of LRU buffers for dbwr to scan when looking for
                                dirt
```

2．No Free Buffer-也就是当进程扫描 LRU 一定数量的 Block 之后，如果还找不到足够的 free 空间，则触发 DBWR 执行写出。

-那么这个扫描数量是多少呢？从以上视图中，我们可以知道，这个比例为 40%，即：

```
kcbfsp                     40 Max percentage of LRU list foreground can scan for free
```

这个限制也受到另外一个隐含参数的限制，这个参数是_db_block_max_scan_pct:

```
SQL> @GetParDescrb.sql
Enter value for par: db_block_max_scan
old   6:    AND x.ksppinm LIKE '%&par%'
new   6:    AND x.ksppinm LIKE '%db_block_max_scan%'
NAME                      VALUE  DESCRIB
--------------------- ------ ------------------------------------------------------
_db_block_max_scan_pct 40     Percentage of buffers to inspect when looking for free
```

通过这些内容，我们可以把很多数据库抽象的概念具体化，有兴趣的，大家可以继续探索。我们将在第五章对这些内容作一点进一步的说明。

# 4.3数据字典表

数据字典表（Data dictionary table）用以存储表、索引、约束以及其他数据库结构的信息。这些对象通常以“$”结尾(例如 TAB$、OBJ$、TS$等)，在创建数据库的时候通过运行 sql.bsq 脚本来创建。

sql.bsq 是非常重要的一个文件，其中包含了数据字典表的定义及注释说明，每个试图深入学习 Oracle 数据库的人都应该仔细阅读一下该文件。该文件位于$ORACLE_HOME/rdbms/admin 目录下.

自 Oracle Database 11g 开始, sql.bsq 中的内容依据功能的不同被拆分为多个脚本, SQL 被分别归类到不同的 bsq 文件，sql.bsq 文件起到一个入口的作用，在创建数据库时分别顺序调用其他文件创建数据库，以下是 11g 中 sql.bsq 文件的一点摘录:

```
rem !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! IMPORTANT !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
rem Whenever new column is created to store internal, user or kernel column
rem number, be sure to update the structure adtDT in atb.c so that those
rem columns will be updated properly during drop column.
rem !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
rem
dcore.bsq
dsqlddl.bsq
dmanage.bsq
dplsql.bsq
dtxnspc.bsq
dfmap.bsq
denv.bsq
drac.bsq
dsec.bsq
doptim.bsq
dobj.bsq
djava.bsq
dpart.bsq
drep.bsq
daw.bsq
dsummgt.bsq
dtools.bsq
dexttab.bsq
ddm.bsq
dlmnr.bsq
ddst.bsq
```

拆分方便了管理和维护,也方便了我们的研究和学习,下面通过一些 BSQ 对象引用了解一下其内容.

以下是我们曾经提到过的 bootstrap$表的定义：

```
create table bootstrap$
( line#         number not null,                       /* statement order id */
  obj#          number not null,                           /* object number */
  sql_text      varchar2("M_VCSZ") not null)                   /* statement */
  storage (initial 50K)            /* to avoid space management during IOR I */
//                                              /* "//" required for bootstrap */
```

OBJ$字典表的部分结构：

```
create table obj$                                           /* object table */
( obj#          number not null,                           /* object number */
  /* DO NOT CREATE INDEX ON DATAOBJ#  AS IT WILL BE UPDATED IN A SPACE
   * TRANSACTION DURING TRUNCATE */
  dataobj#      number,                         /* data layer object number */
  owner#        number not null,                       /* owner user number */
```

```
  name          varchar2("M_IDEN") not null,                  /* object name */
  namespace     number not null,         /* namespace of object (see KQD.H): */
 /* 1 = TABLE/PROCEDURE/TYPE, 2 = BODY, 3 = TRIGGER, 4 = INDEX, 5 = CLUSTER, */
                                                  /* 8 = LOB, 9 = DIRECTORY, */
  /* 10 = QUEUE, 11 = REPLICATION OBJECT GROUP, 12 = REPLICATION PROPAGATOR, */
                                     /* 13 = JAVA SOURCE, 14 = JAVA RESOURCE */
                                                 /* 58 = (Data Mining) MODEL */
  subname       varchar2("M_IDEN"),               /* subordinate to the name */
  type#         number not null,                 /* object type (see KQD.H): */
  /* 1 = INDEX, 2 = TABLE, 3 = CLUSTER, 4 = VIEW, 5 = SYNONYM, 6 = SEQUENCE, */
             /* 7 = PROCEDURE, 8 = FUNCTION, 9 = PACKAGE, 10 = NON-EXISTENT, */
              /* 11 = PACKAGE BODY, 12 = TRIGGER, 13 = TYPE, 14 = TYPE BODY, */
      /* 19 = TABLE PARTITION, 20 = INDEX PARTITION, 21 = LOB, 22 = LIBRARY, */
                                             /* 23 = DIRECTORY , 24 = QUEUE, */
    /* 25 = IOT, 26 = REPLICATION OBJECT GROUP, 27 = REPLICATION PROPAGATOR, */
    /* 28 = JAVA SOURCE, 29 = JAVA CLASS, 30 = JAVA RESOURCE, 31 = JAVA JAR, */
                 /* 32 = INDEXTYPE, 33 = OPERATOR , 34 = TABLE SUBPARTITION, */
                                                  /* 35 = INDEX SUBPARTITION */
                                                 /* 82 = (Data Mining) MODEL */
。。。。。。。
)
  storage (initial 10k next 100k maxextents unlimited pctincrease 0)
/
```

注意通常大家习惯查询的 DBA_OBJECTS 字典视图就是基于 OBJ$ 数据字典表创建的，DBA_OBJECTS 中有两个字段经常使人误解：OBJECT_ID 和 DATA_OBJECT_ID。这两个字段分别来自 OBJ$中的 OBJ#和 DATAOBJ#，其中 OBJ#（也即 OBJECT_ID）可以被看作是对象的逻辑号（类似序列号一样分配），该序号一旦分配之后就不会发生改变；而 DATAOBJ#（也即 DATA_OBJECT_ID）则是和物理存储关联的编号，通常被认为是对象的物理号，这个编号会随着对象物理存储结构的改变而发生改变。注意下面的注释中已经清晰的指出了这一点，Oracle 提示“不要在 DATAOBJ#上创建索引，因为在空间事务如 TRUNCATE 中，DATAOBJ#会发生改变：

```
  obj#          number not null,                            /* object number */
  /* DO NOT CREATE INDEX ON DATAOBJ#  AS IT WILL BE UPDATED IN A SPACE
   * TRANSACTION DURING TRUNCATE */
  dataobj#      number,                          /* data layer object number */
```

通过如下测试就可以发现 TRUNCATE 的这一特点：

```
SQL> create table test as select * from dba_users;
Table created.
SQL> select object_id,data_object_id from dba_objects
```

```
  2  where owner='EYGLE' and object_name='TEST';
 OBJECT_ID DATA_OBJECT_ID
---------- --------------
     15036          15036
SQL> truncate table test;
Table truncated.
SQL> select object_id,data_object_id from dba_objects
  2  where owner='EYGLE' and object_name='TEST';
 OBJECT_ID DATA_OBJECT_ID
---------- --------------
     15036          15037
```

实际上这也暗示了 TRUNCATE 作为 DDL 可以快速回收空间的本质，在执行 TRUNCATE 操作时，数据库只是简单的回收空间，将空间标记为可用（并不会去数据块上真正去删除数据），同时将对象的数据对象重新定位，完成空间回收。那么实际上，虽然 Oracle 并未提供直接的办法，在原对象存储位置被重新写入数据之前，TRUNCATE 数据仍然是有办法恢复的（就如同在 Windows 上误删除的文件，在覆盖之前，是可以通过软件进行恢复的）。

不再过多引用，只要打开这些文件，可能你会发现，很多困扰许久的问题，在这里可以轻易的找到注解及答案。

由于数据字典表对于数据库的稳定运行生死攸关，所以通常 Oracle 不允许直接对数据字典进行操作，当用户执行 DDL 操作或者某些 DML 操作时，在后台 Oracle 将这些操作解析为对于数据字典的操作自动执行。

例如当用户创建一张数据表时，Oracle 将会在后台执行一系列的内部操作，比如向 obj$表中插入数据、向 tab$表中记录表数据、向 col$表中记录字段信息、向 con$记录约束信息、向 seg$中记录数据段信息等，以下测试来自 Oracle Database 11g，通过跟踪我们可以获得这些内部操作的具体步骤：

```
SQL> select * from v$version where rownum <2;
BANNER
--------------------------------------------------------------------------------
Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
SQL> alter session set events '10046 trace name context forever,level 12';
Session altered.
SQL> create table eygle as select * from dba_users;
Table created.
SQL> SELECT VALUE FROM V$DIAG_INFO WHERE NAME = 'Default Trace File';
VALUE
--------------------------------------------------------------------------------
/opt/oracle/diag/rdbms/phsdb/phsdb/trace/phsdb_ora_2854.trc
```

注意，以上查询跟踪文件的视图 V$DIAG_INFO 是 11g 中引入的，其他版本中并不存在。

摘录一点跟踪文件中的数据，从中可以清晰的看到前台的一个 DDL 语句在后台是怎样被

转化一系列的 DML 语句进行执行的，首先记录的是创建语句：

```
.....................
sqlid='dghqcjggp7t96'
create table eygle as select * from dba_users
END OF STMT
```

后台的分解操作，向 obj$中增加记录的 DML 语句：

```
sqlid='4bjwv5sp99589'
insert into
obj$(owner#,name,namespace,obj#,type#,ctime,mtime,stime,status,remoteowner,linkname,subname,dataobj#,flags,oid$,spare1,spare2,spare3)
values(:1,:2,:3,:4,:5,:6,:7,:8,:9,:10,:11,:12,:13,:14,:15,:16,:17,:18)
END OF STMT
```

更新 con$表数据：

```
sqlid='bajr90ryjd2w8'
update con$ set con#=:3,spare1=:4 where owner#=:1 and name=:2
```

增加段信息，向 seg$表插入数据的语句：

```
sqlid='g7mt7ptq286u7'
insert into seg$
(file#,block#,type#,ts#,blocks,extents,minexts,maxexts,extsize,extpct,user#,iniexts,lists,groups,cachehint,hwmincr, spare1, scanhint, bitmapranges) values
(:1,:2,:3,:4,:5,:6,:7,:8,:9,:10,:11,:12,:13,:14,:15,:16,DECODE(:17,0,NULL,:17),:18,:19)
```

向 tab$表增加表记录信息：

```
insert into
tab$(obj#,ts#,file#,block#,bobj#,tab#,intcols,kernelcols,clucols,audit$,flags,pctfree$,pctused$,initrans,maxtrans,rowcnt,blkcnt,empcnt,avgspc,chncnt,avgrln,analyzetime,samplesize,cols,property,degree,instances,dataobj#,avgspc_flb,flbcnt,trigflag,spare1,spare6)values(:1,:2,:3,:4,decode(:5,0,null,:5),decode(:6,0,null,:6),:7,:8,decode(:9,0,null,:9),:10,:11,:12,:13,:14,:15,:16,:17,:18,:19,:20,:21,:22,:23,:24,:25,decode(:26,1,null,:26),decode(:27,1,null,:27),:28,:29,:30,:31,:32,:33)
```

向 col$表增加字段信息：

```
sqlid='60uw2vh6q9vn2'
insert into
col$(obj#,name,intcol#,segcol#,type#,length,precision#,scale,null$,offset,fixedstorage,segcollength,deflength,default$,col#,property,charsetid,charsetform,spare1,spare2,spare3)values(:1,:2,:3,:4,:5,:6,decode(:5,182/*DTYIYM*/,:7,183/*DTYIDS*/,:7,decode(:7,0,null,:7)),decode(:5,2,decode(:8,-127/*MAXSB1MINAL*/,null,:8),178,:8,179,:8,180,:8,181,:8,182,:8,183,:8,231,:8,null),:9,0,:10,:11,decode(:12,0,null,:12),:13,:14,:15,:16,:17,:18,:19,:20)
END OF STMT
```

Oracle 通过将 DDL 解析为 DML 操作，并且将这些操作记录在数据字典表中，通过将这

些信息反向解析，可以得到原始的创建语句，从 Oracle 9i 开始，一个工具包 DBMS_METADATA 被引入到数据库中来完成这项工作：

```
SQL> set pagesize 99
SQL> set long 12000
SQL>  select dbms_metadata.get_ddl('TABLE','EYGLE') from dual;
DBMS_METADATA.GET_DDL('TABLE','EYGLE')
--------------------------------------------------------------------------------
  CREATE TABLE "SYS"."EYGLE"
   (    "USERNAME" VARCHAR2(30) NOT NULL ENABLE,
        "USER_ID" NUMBER NOT NULL ENABLE,
        "PASSWORD" VARCHAR2(30),
        "ACCOUNT_STATUS" VARCHAR2(32) NOT NULL ENABLE,
        "LOCK_DATE" DATE,
        "EXPIRY_DATE" DATE,
        "DEFAULT_TABLESPACE" VARCHAR2(30) NOT NULL ENABLE,
        "TEMPORARY_TABLESPACE" VARCHAR2(30) NOT NULL ENABLE,
        "CREATED" DATE NOT NULL ENABLE,
        "PROFILE" VARCHAR2(30) NOT NULL ENABLE,
        "INITIAL_RSRC_CONSUMER_GROUP" VARCHAR2(30),
        "EXTERNAL_NAME" VARCHAR2(4000),
        "PASSWORD_VERSIONS" VARCHAR2(8),
        "EDITIONS_ENABLED" VARCHAR2(1)
   ) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
  TABLESPACE "SYSTEM"
```

# 4.4 静态数据字典视图

由于 X$表和数据字典表通常不能直接访问，Oracle 创建了静态数据字典视图来提供用户对于数据字典信息的访问，由于这些信息通常相对稳定、不能直接修改，所以又被称为静态数据字典视图。数据字典视图在创建数据库时由 catalog.sql 脚本（该脚本位于$ORACLE_HOME/rdbms/admin/目录下）创建。

## 4.4.1 静态数据字典视图的分类

静态数据字典视图按照前缀的不同通常被分为三类：

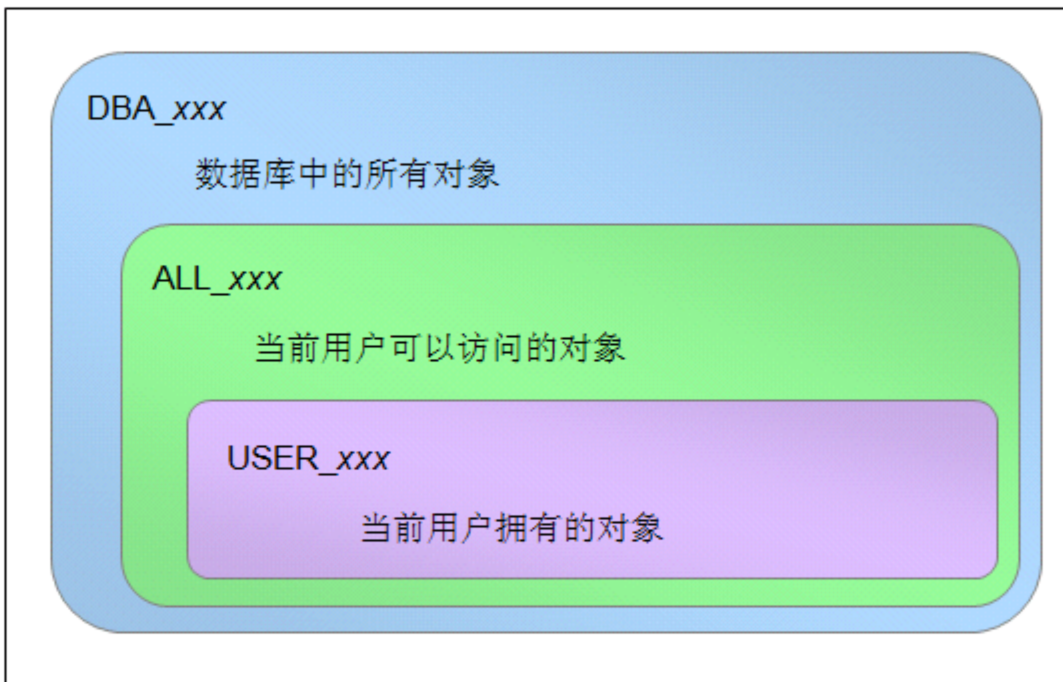
1．USER_ 类视图包含了用户所拥有的相关对象信息，用户可以通过这个视图查询自己

拥有的对象信息。

2．ALL_ 类视图包含了用户有权限访问的所有对象的信息。

3．DBA_ 类视图包含了数据库所有相关对象的信息，用户需要 SELECT ANY TABLE 权限才能访问。

这三者的关系大致可以用如下示意图描述：




通过三类视图在本质上是为了实现权限的控制。在 Oracle 数据库中，每个用户与方案（Schema）是对应的，Schema 是用户所拥有的对象的集合。数据库通过 Schema 将不同用户的对象隔离开来，用户可以自由访问自己 Schema 的对象，但是对于其他 Schema 对象的访问则需要获得授权。

以下通过常用的字典视图做一个简单的示范说明。

USER_TABLES 视图是 USER 类视图的一个典型代表。这个视图中记录了当前用户所拥有的所有表的信息：

```
SQL> connect eygle/eygle
Connected.
SQL> select table_name,tablespace_name from user_tables;
TABLE_NAME                     TABLESPACE_NAME
------------------------------ ------------------------------
CUSTOM                         USERS
```

而对于 ALL_TABLES 视图，不仅包含用户所拥有的表，还包括用户有权限能够访问的表，这些表可能来自其他用户的授权：

```
SQL> connect gqgai/eygle
Connected.
```

```
SQL> grant select on sales to eygle;
Grant succeeded.
```

获得授权之后，用户就能够通过 ALL_TABLES 视图获得这些表的信息：

```
SQL> select table_name,owner from all_tables where owner in ('EYGLE','GQGAI');
TABLE_NAME                     OWNER
------------------------------ ------------------------------
CUSTOM                         EYGLE
SALES                          GQGAI
```

而 DBA_TABLES 则是一个超级集合，包含了数据库所有的表对象，查询这个视图需要 DBA 的权限或者 SELECT ANY TABLE 的系统权限。通过以下查询可以简要统计不同用户拥有的表对象数量：

```
SQL>  select owner,count(*) from dba_tables group by owner order by 2;
OWNER                            COUNT(*)
------------------------------ ----------
EYGLE                                   1
GQGAI                                   1
……………………
SYSTEM                                150
SYS                                   837
23 rows selected.
```

## 4.4.2 静态数据字典视图的内部实现

既然以上提到的这三类字典都是视图（VIEW），那这些视图是怎样建立起来的？又是如何实现的权限控制呢？我们一起来探索一下 Oracle 的实现。

通常 USER_类视图不包含 Owner 字段，查询潜在的返回当前用户的对象信息，我们以 USER_TABLES 视图为例（篇幅原因，省略了部分内容）看一下其创建及结构：

```
create or replace view USER_TABLES
    (TABLE_NAME, TABLESPACE_NAME, CLUSTER_NAME, IOT_NAME,
     PCT_FREE, PCT_USED,
    …….
     DEGREE, INSTANCES, CACHE, TABLE_LOCK,
     SAMPLE_SIZE, LAST_ANALYZED, PARTITIONED,
     IOT_TYPE, TEMPORARY, SECONDARY, NESTED,
     BUFFER_POOL, ROW_MOVEMENT,
     GLOBAL_STATS, USER_STATS, DURATION, SKIP_CORRUPT, MONITORING,
     CLUSTER_OWNER, DEPENDENCIES, COMPRESSION)
as
select o.name, decode(bitand(t.property, 4194400), 0, ts.name, null),
```

```
      decode(bitand(t.property, 1024), 0, null, co.name),
      ……
from sys.ts$ ts, sys.seg$ s, sys.obj$ co, sys.tab$ t, sys.obj$ o,
    sys.obj$ cx, sys.user$ cu
where o.owner# = userenv('SCHEMAID')
  and o.obj# = t.obj#
  …..
  and t.dataobj# = cx.obj# (+)
  and cx.owner# = cu.user# (+)
/
```

我们注意到 Where 条件中有这样一个限制：

```
where o.owner# = userenv('SCHEMAID')
```

这就限制了当前查询只返回当前用户的 SCHEMA 对象信息。

而对于 ALL_TABLES 视图，在 Where 子句中，关于用户部分，增加了这样一个条件：

```
  and (o.owner# = userenv('SCHEMAID')
      or o.obj# in
           (select oa.obj#
            from sys.objauth$ oa
            where grantee# in ( select kzsrorol
                                from x$kzsro
                              )
           )
      or /* user has system privileges */
        exists (select null from v$enabledprivs
                where priv_number in (-45 /* LOCK ANY TABLE */,
                                      -47 /* SELECT ANY TABLE */,
                                      -48 /* INSERT ANY TABLE */,
                                      -49 /* UPDATE ANY TABLE */,
                                      -50 /* DELETE ANY TABLE */)
                )
     )
```

这个条件扩展了关于用户有权限访问的对象信息，所以实际上 USER_TABLES 的结果是 ALL_TABLES 结果的一个子集。

DBA_TABLES 视图的 Where 条件中，则没有关于 Owner 的限制，所以查询返回了数据库中所有表的信息：

```
where o.owner# = u.user#
  and o.obj# = t.obj#
  and bitand(t.property, 1) = 0
  and t.bobj# = co.obj# (+)
```

```
  and t.ts# = ts.ts#
  and t.file# = s.file# (+)
  and t.block# = s.block# (+)
  and t.ts# = s.ts# (+)
  and t.dataobj# = cx.obj# (+)
  and cx.owner# = cu.user# (+)
/
```

这就是这几类数据字典视图的区别所在，也就是 Oracle 实现权限隔离的方式。

## 4.4.3 同义词

由于同义词在随后会被反复提及，我们有必要对同义词做一些简要的介绍。

同义词（synonym）也是数据库中的常见对象，可以看作是为表、视图、物化视图、序列、过程、函数、包、类型（type）、Java 类对象（Java class schema object）、用户定义对象类型（user-defined object type）或是另一个同义词所创建的别名。

同义词在数据库中只是一个简单的别名而已，所以只在数据字典中存储其定义，无需额外存储空间。

**使用同义词是出于方便或安全上的考虑**。例如，可以使用同义词进行以下工作：

- 隐藏一个数据库对象的名字和拥有者（owner）
- 隐藏分布式环境（distributed database）中远程对象（remote object）的位置
- 简化数据库用户的 SQL 语句
- 和视图类似能够限制访问，用于实现更精细的访问控制（fine-grained access control）

用户可以创建公共（public）或私有（private）的同义词。公共同义词由特殊的用户组 PUBLIC 所拥有，数据库中的每个用户都能够访问。而私有同义词属于某个用户，此用户能够控制哪些用户可以使用这些私有同义词。

同义词在分布式和非分布式环境中都有很大用处，因为同义词隐藏了相关对象的具体信息，包括对象在分布式环境中的位置信息。这样做的好处是，如果相关对象必须重命名或移动位置的话，只需重新定义同义词，而使用同义词的应用程序不会受这类数据库修改的影响。

同义词还可以简化分布式数据库环境中用户的 SQL 语句。以下例子显示了数据库管理员为什么以及如何使用公共同义词（public synonym）来隐藏数据库对象的信息，从而减少 SQL 语句的复杂性。假使情况如下：

1) SALES_DATA 表位于 EYGLE 用户的方案中。
2) 授予 PUBLIC 用户组对 SALES_DATA 表的 SELECT 权限。

此时，用户可以使用以下 SQL 语句查询 SALES_DATA 表：

```
    SELECT * FROM eygle.sales_data;
```

注意用户必须在语句中指定表名，及此表所属的方案（schema）。

如果数据库管理员使用如下 SQL 语句创建了公共同义词：

```
    CREATE PUBLIC SYNONYM sales FOR eygle.sales_data;
```

创建了公共同义词后，用户可以使用更简单的 SQL 语句来查询 SALES_DATA 表：

```
SELECT * FROM sales;
```

注意公共同义词 SALES 隐藏了 SALES_DATA 表的名称，及其所属的方案 EYGLE。

如果要删除不再使用的同义词，可使用如下 SQL 语句：

```
DROP PUBLIC SYNONYM sales;
```

如果要查看数据库中有哪些同义词，可用过以下视图查询：

- ALL_SYNONYMS:描述当前用户可访问的全部同义词
- DBA_SYNONYMS:描述数据库中的全部同义词
- USER_SYNONYMS:描述当前用户拥有的全部同义词

### 4.4.4 常用数据字典视图举例

在 DBA 进行数据库管理和维护过程中，经常需要查询一些数据字典视图获得相关的数据库信息，为了方便管理和学习，有一些视图或同义词需要熟悉，以下简要列举一些常用的数据字典视图。

**1. DICT / DICTIONARY**

为了方便检索，Oracle 提供一个名为字典（DICTIONARY）的视图，基于这个视图，Oracle 又创建了两个名为 DICT 和 DICTIONARY 的同义词：

```
SQL> select owner,object_name,object_type
  2  from dba_objects where object_name in ('DICT','DICTIONARY');
OWNER           OBJECT_NAME                    OBJECT_TYPE
--------------- ------------------------------ -------------------
SYS             DICTIONARY                     VIEW
PUBLIC          DICT                           SYNONYM
PUBLIC          DICTIONARY                     SYNONYM
```

DICT / DICTIONARY 对象的内容很简单，仅仅包含两个字段：

```
SQL> desc dictionary
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 TABLE_NAME                                         VARCHAR2(30)
 COMMENTS                                           VARCHAR2(4000)
```

不要被 TABLE_NAME 这个描述迷惑，这里应该用 OBJECT_NAME 更符合实际，DICTIONARY 视图包含的是当前用户可以访问的所有数据字典视图。通过 DICTIONARY 的创建语句可以更详细的了解数据的来源（为了说明，在创建语句中增加了部分注释）：

```
CREATE OR REPLACE FORCE VIEW SYS.DICTIONARY (table_name, comments) AS
   SELECT o.NAME, c.comment$ FROM SYS.obj$ o, SYS.com$ c
    WHERE o.obj# = c.obj#(+) AND c.col# IS NULL AND o.owner# = 0 AND o.type# = 4
    /* 这里 owner#=0 为 SYS 用户，o.type#=4 为 VIEW 类型，可以参考 user$、obj$表 */
      AND (   o.NAME LIKE 'USER%' OR o.NAME LIKE 'ALL%'
```

```
              OR ( o.NAME LIKE 'DBA%'
               AND EXISTS (SELECT NULL FROM SYS.v$enabledprivs
                            WHERE priv_number = -47 /* SELECT ANY TABLE */) ) )
    /*  所有用户都能够访问 USER 和 ALL 类别的视图，但是访问 DBA 类视图则需要判断是否具有
SELECT ANY TABLE 的权限 */
   UNION ALL
   SELECT o.NAME, c.comment$ FROM SYS.obj$ o, SYS.com$ c
    WHERE o.obj# = c.obj#(+) AND o.owner# = 0 AND o.NAME IN
       ('AUDIT_ACTIONS', 'COLUMN_PRIVILEGES', 'DICTIONARY','DICT_COLUMNS',
        'DUAL', 'GLOBAL_NAME', 'INDEX_HISTOGRAM','INDEX_STATS',
        'RESOURCE_COST', 'ROLE_ROLE_PRIVS','ROLE_SYS_PRIVS', 'ROLE_TAB_PRIVS',
        'SESSION_PRIVS','SESSION_ROLES', 'TABLE_PRIVILEGES', 'NLS_SESSION_PARAMETERS',
         'NLS_INSTANCE_PARAMETERS', 'NLS_DATABASE_PARAMETERS','DATABASE_COMPATIBLE_LEVEL',
              'DBMS_ALERT_INFO','DBMS_LOCK_ALLOCATED')  AND c.col# IS NULL
    /* 这里一个 UNION ALL 引入一系列所有用户都可以访问的视图 */
   UNION ALL
   SELECT so.NAME, 'Synonym for ' || sy.NAME FROM SYS.obj$ ro, SYS.syn$ sy, SYS.obj$ so
    WHERE so.type# = 5 AND ro.linkname IS NULL AND so.owner# = 1 AND so.obj# = sy.obj#
      AND so.NAME <> sy.NAME AND sy.owner = 'SYS' AND sy.NAME = ro.NAME
      AND ro.owner# = 0 AND ro.type# = 4
    /* 这里 type# =5 为 SYNONYM，owner#=1 为 PUBLIC，这个查询是找出为视图建立的共用同义词 */
      AND (   ro.owner# = USERENV ('SCHEMAID')
    /* 根据用户 SCHEMA 进行权限隔离 */
           OR ro.obj# IN (SELECT oa.obj# FROM SYS.objauth$ oa
                           WHERE grantee# IN (SELECT kzsrorol FROM x$kzsro))
           OR EXISTS (SELECT NULL FROM v$enabledprivs WHERE priv_number IN
                           (-45 /* LOCK ANY TABLE */,
                            -47 /* SELECT ANY TABLE */,
                            -48 /* INSERT ANY TABLE */,
                            -49 /* UPDATE ANY TABLE */,
                            -50 /* DELETE ANY TABLE */ )));
```

在完成这是 VIEW 的创建之后，以下两个语句创建了 DICT 和 DICTIONARY 同义词：

```
CREATE PUBLIC SYNONYM DICTIONARY FOR SYS.DICTIONARY;
CREATE PUBLIC SYNONYM DICT FOR SYS.DICTIONARY;
```

有了这个字典表之后，我们可以通过它查询我们感兴趣的内容，比如以下查询返回和 TABLES 相关的视图：

```
select table_name from dict where table_name like '%TABLES%';
```

**2. DICT_COLUMNS**

同 DICT 类似，DICT_COLUMNS 视图记录了字典视图列（COLUMN）及其相关说明：

```
SQL> select column_name,comments from dict_columns
  2  where table_name ='DICT';
COLUMN_NAME                    COMMENTS
------------------------------ ------------------------------
TABLE_NAME                     Name of the object
COMMENTS                       Text comment on the object
```

通过这个视图我们可以快速获得很多有用的信息，比如找到具有较多字段的 Top 10 字典视图：

```
SQL> SELECT *
  2    FROM (SELECT   table_name, COUNT (*)
  3               FROM dict_columns
  4           GROUP BY table_name
  5           ORDER BY 2 DESC)
  6   WHERE ROWNUM < 6;
TABLE_NAME                       COUNT(*)
------------------------------ ----------
GV$SESSION                             92
V$SESSION                              91
GV$SQL                                 77
V$SQL                                  76
GV$SQLAREA                             73
```

类似的，和 COLUMN 相关的视图还有很多，通过 DICT 视图可以很快速的找到这些视图，以下是一些常用的视图输出：

```
SQL> select table_name from dict where table_name like 'DBA%COLUMNS';
TABLE_NAME
------------------------------
DBA_CONS_COLUMNS
DBA_IND_COLUMNS
DBA_TAB_COLUMNS
。。。。。。。。
DBA_COMPARISON_COLUMNS
DBA_STREAMS_COLUMNS
25 rows selected.
```

DBA_TAB_COLUMNS 视图记录了所有表、视图和 Cluster 的字段信息

DBA_IND_COLUMNS 视图记录了所有表和 Cluster 上的索引字段信息。

通过这些字典表，很容易可以实现类似之前的命题：获得数据库中字段最多的表和索引等。

**3.　OBJ$/DBA_OBJECTS/OBJ**

OBJ$是一个底层的字典表，其中记录了数据库中所有对象的信息，DBA_OBJECTS 视图基于 OBJ$建立，一脉相承的，ALL_OBJECTS 和 USER_OBJECTS 视图也随之建立。

OBJ 是一个对于 USER_OBJECTS 建立的同义词，其创建语句为：

```
CREATE PUBLIC SYNONYM OBJ FOR SYS.USER_OBJECTS;
```

用户可以通过这个同义词来获得当前用户所拥有的对象信息:

```
SQL> connect eygle/eygle
Connected.
SQL> select object_name,object_type from obj;
OBJECT_NAME                    OBJECT_TYPE
------------------------------ -------------------
CUSTOM                         TABLE
```

类似的字典表还有 IND$/DBA_INDEXES/IND 等，掌握了这一规律之后，对于很多类的数据字典视图我们就能够举一反三了。

**4.　*_SOURCE 视图**

DBA_SOURCE/USER_SOUCE/ALL_SOURCE 用于保存存储对象（Stored Object）的源码。这类视图存储的对象类型包括 FUNCTION, JAVA SOURCE, PACKAGE, PACKAGE BODY, PROCEDURE, TRIGGER, TYPE, TYPE BODY 等：

```
SQL> desc dba_source
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 OWNER                                              VARCHAR2(30)
 NAME                                               VARCHAR2(30)
 TYPE                                               VARCHAR2(12)
 LINE                                               NUMBER
 TEXT                                               VARCHAR2(4000)
```

通过 TEXT 字段能够获取相关对象的创建文本：

```
SQL> select text from dba_source where name='CALLING';
TEXT
--------------------------------------------------------------------------------
procedure calling
 is
begin
           pining;
           dbms_lock.sleep(10);
end;

6 rows selected.
```

# 4.5动态性能视图

动态性能（V $）视图（Dynamic Performance View）记录了数据库运行时信息和统计数据，大部分动态性能视图被实时更新以反映数据库当前状态。

Oracle 通过动态性能视图将 Oracle 数据库的状态展示出来，提供给用户和数据库管理员，Oracle 对 V$视图给出了详细的文档说明供开发管理人员参考，是我们研究和管理数据库的主要依据。

## 4.5.1 GV$和 V$视图

在数据库启动时，Oracle 动态创建 X$表，在此基础之上，Oracle 创建了 GV$和 V$视图。从 Oracle8 开始，GV$视图开始被引入，其含义为 Global V$。

除了一些特例以外，每个 V$视图都有一个对应的 GV$视图存在。

GV$视图的产生是为了满足 OPS/RAC 环境的需要，在 OPS/RAC 环境中，查询 GV$视图返回所有实例信息，而每个 V$视图是基于 GV$视图，增加了 INST_ID 列的 WHERE 条件限制建立，只包含当前连接实例信息。

注意，每个 V$视图都包含以下类似语句，用于限制返回当前实例信息：

```
where inst_id = USERENV('Instance')
```

下面以 Oracle10gR2 RAC 环境为例，看一下 GV$和 V$视图的输出异同。以下是 GV$的输出，包含了 2 个实例的信息：

```
SQL> select inst_id,instance_name,status,version
  2  from gv$instance;
   INST_ID INSTANCE_NAME    STATUS       VERSION
---------- ---------------- ------------ -----------------
         1 RACDB1           OPEN         10.2.0.1.0
         2 RACDB2           OPEN         10.2.0.1.0
```

而对于 V$视图的查询，则只有单实例信息输出：

```
SQL> select instance_number,instance_name,status
  2  from v$instance;
INSTANCE_NUMBER INSTANCE_NAME    STATUS
--------------- ---------------- ------------
              1 RACDB1           OPEN
```

Oracle 提供了一些特殊视图用以记录其他视图的创建方式。v$fixed_view_definition 就是其中之一。

从 GV$FIXED_TABLE 和 V$FIXED_TABLE 开始，我们来看一下 GV$视图和 V$视图的结构及创建方式：

```
SQL> select view_definition from v$fixed_view_definition where view_name='V$FIXED_TABLE';
VIEW_DEFINITION
```

```
--------------------------------------------------------------------------------
select NAME , OBJECT_ID , TYPE , TABLE_NUM from GV$FIXED_TABLE
where inst_id = USERENV('Instance')
```

这里我们看到 V$FIXED_TABLE 基于 GV$FIXED_TABLE 创建。

```
SQL> select view_definition from v$fixed_view_definition where view_name='GV$FIXED_TABLE';
VIEW_DEFINITION
--------------------------------------------------------------------------------
select inst_id,kqftanam, kqftaobj, 'TABLE', indx from x$kqfta
union all
select inst_id,kqfvinam, kqfviobj, 'VIEW', 65537 from x$kqfvi
union all
select inst_id,kqfdtnam, kqfdtobj, 'TABLE', 65537 from x$kqfdt
```

这样我们找到了 GV$FIXED_TABLE 视图的创建语句，该视图基于 X$表创建，然后 V$视图基于 GV$视图创建。

总结一下，Oracle 的 GV$视图和 V$视图是在数据库创建过程中建立起来的，内置于数据库中，Oracle 通过 v$fixed_view_definition 视图为我们展现这些定义。

### 4.5.2 GV_$,V_$视图和 V$,GV$同义词

在 GV$和 V$之后，Oracle 建立了 GV_$和 V_$视图，随后为这些视图建立了公用同义词。这些工作都是通过 catalog.sql 脚本（该脚本位于$ORACLE_HOME/rdbms/admin/目录下）实现的。

从 catalog.sql 脚本中摘录一段:

```
create or replace view v_$fixed_table as select * from v$fixed_table;
create or replace public synonym v$fixed_table for v_$fixed_table;

create or replace view gv_$fixed_table as select * from gv$fixed_table;
create or replace public synonym gv$fixed_table for gv_$fixed_table;
```

从以上脚本中可以注意到，首先 V_$和 GV_$视图基于 V$和 GV$视图被创建，然后基于 V_$和 GV_$视图的同义词被创建。

通过 V_$视图，Oracle 把 V$视图和普通用户隔离，V_$视图的权限可以授予其他用户，而 Oracle 不允许任何对于 V$视图的直接授权，我们看以下例子：

```
SQL> select * from v$version where rownum <2;
BANNER
----------------------------------------------------------------
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Prod
SQL> grant select on v$sga to eygle;
grant select on v$sga to eygle
                *
```

```
ERROR at line 1:
ORA-02030: can only select from fixed tables/views
SQL> grant select on v_$sga to eygle;
Grant succeeded.
```

对于内部 X$表及 V$视图的限制，Oracle 是通过软件机制实现的，而并非通过数据库权限控制。所以，实际上**通常大部分用户访问的 V$对象，并不是视图，而是指向 V_$视图的同义词；而 V_$视图是基于真正的 V$视图(这个视图是基于 X$表建立的)创建的。**

通过以下测试来加深一下理解，首先创建一个测试用户，仅授予该用户 CREATE SESSION 的权限：

```
SQL> create user eyglee identified by eygle;
User created.
SQL> grant create session to eyglee;
Grant succeeded.
```

如果此时使用该用户连接，查询 V$SESSION 视图，数据库会给出错误信息，因为该用户无权访问该视图，但是注意这里提示的视图是 V_$SESSION 视图的访问问题：

```
SQL> connect eyglee/eygle
Connected.
SQL> desc v$session
ERROR:
ORA-04043: object "SYS"."V_$SESSION" does not exist
```

这样的提示正是因为用户对于 V$SESSION 同义词的访问最终会被解析为对于底层视图 V_$SESSION 的访问，而真正的 V$SESSION 视图是不允许 SYS 之外用户直接访问的。

接下来通过 SYS 用户将 SELECT_CATALOG_ROLE 的角色授予用户，这一角色可以使用户获得查询数据字典的权限，然后此时用户就获得了访问 V$SESSION 的权限：

```
SQL> connect / as sysdba
Connected.
SQL> grant SELECT_CATALOG_ROLE to eyglee;
Grant succeeded.
SQL> connect eyglee/eygle
Connected.
SQL> select count(*) from v$session;
  COUNT(*)
----------
        25
```

接下来授予用户 CREATE PROCEDURE 的权限，进行进一步的测试：

```
SQL> connect / as sysdba
Connected.
SQL> grant create procedure to eyglee;
Grant succeeded.
```

在用户模式下创建一个用于显示连接用户列表的存储过程：

```
SQL> connect eyglee/eygle
Connected.
SQL> CREATE OR REPLACE PROCEDURE userlist
  2  AS
  3     TYPE mytable IS TABLE OF v$session%ROWTYPE;
  4
  5     l_data   mytable;
  6     l_refc   sys_refcursor;
  7  BEGIN
  8     OPEN l_refc
  9      FOR
 10         SELECT * FROM v$session WHERE username IS NOT NULL;
 11     FETCH l_refc BULK COLLECT INTO l_data;
 12     CLOSE l_refc;
 13
 14     FOR i IN 1 .. l_data.COUNT
 15     LOOP
 16        DBMS_OUTPUT.put_line (   l_data (i).username
 17                              || '(sid='
 18                              || l_data (i).SID
 19                              || ',serial#='
 20                              || l_data (i).serial#
 21                              || ')'
 22                              || ' is connected from machine '
 23                              || l_data (i).machine
 24                              || ' with program:'
 25                              || l_data (i).program
 26                             );
 27     END LOOP;
 28  END;
 29  /
Warning: Procedure created with compilation errors.
```

注意创建出现了错误，显示一下具体错误信息：

```
SQL> show error
Errors for PROCEDURE USERLIST:
LINE/COL ERROR
-------- -----------------------------------------------------------------
3/4      PL/SQL: Item ignored
```

```
3/29     PLS-00201: identifier 'V$SESSION' must be declared
10/8     PL/SQL: SQL Statement ignored
10/22    PL/SQL: ORA-00942: table or view does not exist
11/4     PL/SQL: SQL Statement ignored
11/35    PLS-00597: expression 'L_DATA' in the INTO list is of wrong type
16/7     PL/SQL: Statement ignored
16/43    PLS-00487: Invalid reference to variable 'V$SESSION%ROWTYPE'
```

注意以上的错误信息主要是 ORA-00942，也就是存储过程发现不了 V$SESSION 视图，这说明用户不具备访问 V$SESSION 的权限。

这是在存储过程或触发器中访问字典对象可能遇到的典型错误，虽然当我们使用用户身份登录时可以访问 V$SESSION 视图，但是该访问权限是通过 SELECT_CATALOG_ROLE 角色获得的，角色权限需要登录才能激活，在过程和触发器等对象中不能生效。

所以为了获得对 V$SESSION 视图的访问权限，需要对用户进行显示授权，当然这个授权要基于 V_$SESSION 进行：

```
SQL> connect / as sysdba
Connected.
SQL> grant select on v_$session to eyglee;
Grant succeeded.
```

现在具备了足够的权限，存储过程可以顺利编译通过：

```
SQL> alter procedure userlist compile;
Procedure altered.
```

现在来执行一下这个存储过程可以看到成功执行的输出：

```
SQL> set serveroutput on
SQL> exec userlist
EYGLEE(sid=128,serial#=5600) is connected from machine localhost.localdomain
with program:sqlplus@localhost.localdomain (TNS V1-V3)
EYGLE(sid=133,serial#=12772) is connected from machine localhost.localdomain
with program:sqlplus@localhost.localdomain (TNS V1-V3)

PL/SQL procedure successfully completed.
```

了解是视图结构与关系之后，再来看一下访问顺序。需要了解的是，在进行数据访问时，**Oracle 访问 VIEW 优先，然后是同义词**，通过以下实验来验证一下这个结论。

首先参考 Oracle 处理机制，创建 X$EYGLE,V$EYGLE,V_$EYGLE 和公用同义词 V$EYGLE：

```
SQL> connect eygle/eygle
Connected.
SQL> select * from v$version where rownum <2;
BANNER
----------------------------------------------------------------
```

```
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
SQL> create table x$eygle as select username from dba_users;
Table created.
SQL> create view v$eygle as select * from x$eygle;
View created.
SQL> create view v_$eygle as select * from v$eygle;
View created.
SQL> create public synonym v$eygle for v_$eygle;
Synonym created.
```

然后在 sys 用户下创建 V$EYGLE 视图：

```
SQL> connect / as sysdba
Connected.
SQL> create view v$eygle as select username,user_id from dba_users;
View created.
```

此时查询，得到的 SYS 的 V$EYGLE 视图信息：

```
SQL> desc v$eygle;
 Name                                Null?    Type
 -----------------------------------------------------------------
 USERNAME                            NOT NULL VARCHAR2(30)
 USER_ID                             NOT NULL NUMBER
```

当删除这个视图以后，再次访问时，Oracle 选择访问了 V$EYGLE 同义词：

```
SQL> drop view v$eygle ;
View dropped.
SQL> desc v$eygle
 Name                                Null?    Type
 -----------------------------------------------------------------
 USERNAME                            NOT NULL VARCHAR2(30)
```

v$fixed_view_definition 视图是我们研究 Oracle 对象关系的一个入口，仔细理解 Oracle 的数据字典机制，有助于深入了解和学习 Oracle 数据库知识。

### 4.5.3 进一步的说明

Oracle 的 X$表信息可以从 v$fixed_table 中查到：

```
SQL> select count(*) from v$fixed_table where name like 'X$%';
  COUNT(*)
----------
       394
```

对于 Oracle9iR2，共有 394 个 X$对象被记录。

X$表建立以后，基于 X$表的 GV$和 V$视图得以创建。这部分视图我们也可以通过查询

V$FIXED_TABLE 得到。

在以下查询中，GV$视图和 V$视图各有 259 个：

```
SQL> select count(*) from v$fixed_table where name like 'GV$%';
  COUNT(*)
----------
       259
SQL> select count(*) from v$fixed_table where name like 'V$%';
  COUNT(*)
----------
       259
```

v$fixed_table 共记录了: 394 + 259 + 259 共 912 个对象。

```
SQL> select count(*) from v$fixed_table;


  COUNT(*)
----------
       912
```

以上是 Oracle9iR2 单机环境中的数据：

```
SQL> select * from v$version where rownum <2;
BANNER
----------------------------------------------------------------
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
```

而在 Oracle10gR2 中，v$fixed_table 中的对象数量增加到 1383 个（不同安装可能不同）. 在 Oracle11gR1 中，这一数字又增加到 1741 个左右。从 Oracle10g 开始，一个同以前不同新的命名规则被引入：

```
SQL> select name from v$fixed_table
   2 where name not like 'GV%' and name not like 'V$%' and name not like 'X$%';
NAME
------------------------------
GO$SQL_BIND_CAPTURE
O$SQL_BIND_CAPTURE
```

大家可以通过这个视图对 Oracle 数据库不同版本的变化进行对比和研究，本文不再赘述。

## 4.5.4 动态性能视图与数据库启动

由于动态性能视图是在数据库启动过程中自动创建的，所以在数据库启动的不同阶段，我们能够访问的视图也各不相同。

1． 在 Nomount 阶段

当数据库启动到 nomount 状态时，实际上仅仅启动了数据库实例，此时的实例信息主

要来自参数文件，因此和参数文件记录的相关信息可以查询，以下是这一阶段可以获取信息的主要视图：

V$PARAMETER、V$SPPARAMETER

V$SGA、V$SGASTAT、V$BH、V$INSTANCE

V$OPTION、V$VERSION

V$PROCESS、V$SESSION

2． 在 Mount 阶段

当数据库启动到 Mount 状态时，控制文件被读取，和控制文件相关的视图此时可以进行查询，以下是这一阶段可以获取信息的主要视图：

V$THREAD、V$CONTROLFILE、V$DATABASE、

V$DATAFILE、V$LOGFILE、V$DATAFILE_HEADER

3． 在 Open 阶段

当数据库 Open 之后，所有的动态性能视图和数据字典都可以被查询。

## 4.6最后的验证

最后让我们通过 V$PARAMETER 视图来追踪一下数据库的架构：

### 4.6.1 V$PARAMETER 的结构

```
SQL> select view_definition from v$fixed_view_definition a where a.VIEW_NAME='V$PARAMETER';
VIEW_DEFINITION
--------------------------------------------------------------------------------
select  NUM , NAME , TYPE , VALUE , ISDEFAULT , ISSES_MODIFIABLE , ISSYS_MODIFIA
BLE , ISMODIFIED , ISADJUSTED , DESCRIPTION, UPDATE_COMMENT from GV$PARAMETER wh
ere inst_id = USERENV('Instance')
```

这里清晰的看到 V$PARAMETER 是由 GV$PARAMETER 创建的，GV$PARAMETER 则是由 X$创建的：

```
SQL> select view_definition from v$fixed_view_definition a where a.VIEW_NAME='GV$PARAMETER';
VIEW_DEFINITION
--------------------------------------------------------------------------------
SELECT x.inst_id, x.indx + 1, ksppinm, ksppity, ksppstvl, ksppstdf,
       DECODE (BITAND (ksppiflg / 256, 1), 1, 'TRUE', 'FALSE'),
       DECODE (BITAND (ksppiflg / 65536, 3),
               1, 'IMMEDIATE',
               2, 'DEFERRED',
               3, 'IMMEDIATE',
               'FALSE'
```

```
              ),
       DECODE (BITAND (ksppstvf, 7), 1, 'MODIFIED', 4, 'SYSTEM_MOD', 'FALSE'),
       DECODE (BITAND (ksppstvf, 2), 2, 'TRUE', 'FALSE'), ksppdesc,
       ksppstcmnt
  FROM x$ksppi x, x$ksppcv y
 WHERE (x.indx = y.indx)
   AND ((TRANSLATE (ksppinm, '_', '#') NOT LIKE '#%') OR (ksppstdf = 'FALSE')
       )
```

说明：在这里我们看到 GV$PARAMETER 来源于 x$ksppi,x$ksppcv 两个 X$表。x$ksppi,x$ksppcv 基本上包含所有数据库参数，v$parameter 展现的是不包含"_"开头的参数。以"_"开头的参数我们通常称为隐含参数，一般不建议修改，但很多因为功能强大经常使用而广为人知。

## 4.6.2 视图还是同义词

在非 SYS 用户下查询，很多朋友曾经提出过疑问，那就是当访问 V$PARAMETER 对象时，访问的是视图还是同义词?

如果你还记得我们前面讲过的内容，那么你会知道，毫无疑问，这里访问的是同义词，因为除了 SYS 用户以外，其他用户不能查询 V$视图，V$视图也不能被授权给其他用户。

```
SQL> connect / as sysdba
Connected.
SQL> grant select on v$parameter to eygle;
grant select on v$parameter to eygle
                *
ERROR at line 1:
ORA-02030: can only select from fixed tables/views

SQL> connect eygle/eygle
Connected.
SQL> desc sys.v$parameter
ERROR:
ORA-04043: object sys.v$parameter does not exist
SQL> desc v$parameter
 Name                           Null?    Type
 -----------------------------------------------------------------
 NUM                                     NUMBER
 NAME                                    VARCHAR2(64)
 TYPE                                    NUMBER
 VALUE                                   VARCHAR2(512)
```

```
ISDEFAULT                                    VARCHAR2(9)
ISSES_MODIFIABLE                             VARCHAR2(5)
ISSYS_MODIFIABLE                             VARCHAR2(9)
ISMODIFIED                                   VARCHAR2(10)
ISADJUSTED                                   VARCHAR2(5)
DESCRIPTION                                  VARCHAR2(64)
UPDATE_COMMENT                               VARCHAR2(255)
```

### 4.6.3 Oracle 如何通过同义词定位对象

如果愿意的话，可以进一步来进行追溯，使用 10046 事件跟踪，我们可以看到更多的东西。

通过 10046 事件跟踪查询：

```
SQL> connect eygle/eygle
Connected.
SQL> alter session set events '10046 trace name context forever,level 12';
Session altered.
SQL> select count(*) from v$parameter;
  COUNT(*)
----------
       262
```

在这里不要使用 tkprof 格式化，因为 tkprof 可能会隐去重要信息（本文仅摘取几段重要跟踪信息，你完全可以通过实验获得相同的输出）：

第一段重要代码是：

```
PARSING IN CURSOR #2 len=198 dep=1 uid=0 oct=3 lid=0 tim=1092440257023120 hv=2703824309
ad='567681f0'
select obj#,type#,ctime,mtime,stime,status,dataobj#,flags,oid$, spare1, spare2 from
obj$ where owner#=:1 and name=:2 and namespace=:3 and remoteowner is null and linkname is null
and subname is null
END OF STMT
PARSE #2:c=0,e=1601,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=0,tim=1092440257023088
BINDS #2:
 bind 0: dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=08 oacfl2=1 size=24 offset=0
   bfp=b701cf24 bln=22 avl=02 flg=05
   value=25
 bind 1: dty=1 mxl=32(11) mal=00 scl=00 pre=00 oacflg=18 oacfl2=1 size=32 offset=0
   bfp=b701c7b4 bln=32 avl=11 flg=05
```

```
  value="V$PARAMETER"
 bind 2: dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=08 oacfl2=1 size=24 offset=0
  bfp=b701c790 bln=24 avl=02 flg=05
  value=1
```

Oracle 根据三个传入参数 **owner#=25,name=V$PARAMETER,namespace=1**，来判断对象类型，按照表、视图优先规则来定位判断，对于本例用户 EYGLE 下是不存在这个表或视图的。

接下来 Oracle 继续判断，此时需要验证同义词了：

```
PARSING IN CURSOR #4 len=46 dep=1 uid=0 oct=3 lid=0 tim=1092440257028409 hv=3378994511
ad='576eb040'
select node,owner,name from syn$ where obj#=:1
END OF STMT
PARSE #4:c=0,e=1278,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=0,tim=1092440257028379
BINDS #4:
 bind 0: dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=08 oacfl2=1 size=24 offset=0
  bfp=b701b3cc bln=22 avl=03 flg=05
  value=841
```

传入绑定变量值是 841，我们看看 841 是什么：

```
SQL> select object_name,object_id,object_type from dba_objects where object_id=841;
OBJECT_NAME                          OBJECT_ID OBJECT_TYPE
----------------------------------------   ------------------
V$PARAMETER                                841 SYNONYM
```

841 正是这个同义词，我们再继续看这个递归 SQL 的作用：

```
SQL> select node,owner,name from syn$ where obj#=841;
NODE     OWNER                          NAME
-------- ------------------------------ ------------------------------
         SYS                            V_$PARAMETER
```

原来这个 SQL 获得的是同义词的底层对象，这里得到了 V_$PARAMETER。继续向下查看这个跟踪文件：

```
PARSING IN CURSOR #8 len=37 dep=1 uid=0 oct=3 lid=0 tim=1092440257074273 hv=3468666020
ad='576db210'
select text from view$ where rowid=:1
END OF STMT
PARSE #8:c=0,e=1214,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=0,tim=1092440257074242
BINDS #8:
 bind 0: dty=11 mxl=16(16) mal=00 scl=00 pre=00 oacflg=18 oacfl2=1 size=16 offset=0
  bfp=b7018770 bln=16 avl=16 flg=05
  value=000001CD.0013.0001
EXEC #8:c=0,e=972,p=0,cr=0,cu=0,mis=0,r=0,dep=1,og=4,tim=1092440257075602
```

注意这里，Oracle 执行查询访问 view$视图，获得视图定义文本，看一下这里访问的是什

么对象，绑定变量传入的 rowid 值为 000001CD.0013.0001，注意这是个受限 rowid,查询时需要转换一下处理：

```
SQL> select obj# from view$
   2 where dbms_rowid.rowid_to_restricted(rowid,0) = '000001CD.0013.0001';
      OBJ#
----------
       840
SQL> select object_name,object_type from dba_objects where object_id=840;
OBJECT_NAME                    OBJECT_TYPE
------------------------------ ------------------
V_$PARAMETER                   VIEW
```

这里 Oracle 访问的正是 V_$PARAMETER 视图的定义方式。执行查询可以得到：

```
select text from view$ where obj#=840;
TEXT
--------------------------------------------------------------------------------
select
"NUM","NAME","TYPE","VALUE","ISDEFAULT","ISSES_MODIFIABLE","ISSYS_MODIFIABLE","ISMODIFIED
","ISADJUSTED","DESCRIPTION","UPDATE_COMMENT" from v$parameter
```

至此就完成了查询中的回溯及定位，当然，实际过程中 Oracle 后台的递归操作比这还要复杂的多，感兴趣的朋友可以按照文中的方法测试研究一下。

最后总结一下 SQL 语句中 Oracle 对于对象名的解析顺序：

1）Oracle 首先查看在发出命令的用户模式中是否存在表或视图。
2）如果表或视图不存在，Oracle 检查私有同义词是否存在。
3）如果私有同义词存在，将使用这个同义词所引用的对象。
4）如果私有同义词不存在，检查同名的公共同义词是否存在。
5）如果公共同义词存在，将使用这个同义词所引用的对象。
6）如果公共同义词不存在，Oracle 返回消息“**ORA-00942 table or view does not exist**”。

用伪代码来大致描述一下这个过程就是:

```
Parse for Object T
if (TABLE t or VIEW t)
     return
elseif (SYNONYM t)
     return
elseif (public SYNONYM t)
     return
else
     signal ORA-00942
```

```
end
```

## 4.7同义词优化案例一则

在分析一个客户数据库中的 statspack 报告时，发现有 1 条 SQL 引起了大量的逻辑读，执行次数也比较高，在 2 小时内执行了 74,131 次，每次的逻辑读为 517.9，主要内容如下所示：

```
    ------------------------------------------------------------
 SQL ordered by Gets for DB: EREADYE  Instance: ereadye  Snaps: 195 -197
-> End Buffer Gets Threshold:    10000
-> Note that resources reported for PL/SQL includes the resources used by
   all SQL statements called within the PL/SQL code.  As individual SQL
   statements are also reported, it is possible and valid for the summed
   total % to exceed 100

                                                     CPU      Elapsd
  Buffer Gets    Executions  Gets per Exec  %Total Time (s)  Time (s) Hash Value
--------------- ------------ -------------- ------ -------- --------- ----------
     38,391,045       74,131          517.9    0.6   833.81    820.26 3806450524
Module: JDBC Thin Client
SELECT NULL AS table_cat,        t.owner AS table_schem,
t.table_name AS table_name,        t.column_name AS column_name,
        DECODE (t.data_type, 'CHAR', 1, 'VARCHAR2', 12, 'NUMBER'
, 3,                 'LONG', -1, 'DATE', 93, 'RAW', -3, 'LONG RAW
', -4, 1111)              AS data_type,        t.data_type AS t
ype_name,        DECODE (t.data_precision, null, t.data_length,
t.data_precision)              AS column_size,        0 AS buff
er_length,        t.data_scale AS decimal_digits,        10 AS n
um_prec_radix,        DECODE (t.nullable, 'N', 0, 1) AS nullable
,         NULL AS remarks,        t.data_default AS column_def,
      0 AS sql_data_type,        0 AS sql_datetime_sub,        t
.data_length AS char_octet_length,        t.column_id AS ordinal
_position,        DECODE (t.nullable, 'N', 'NO', 'YES') AS is_nu
llable FROM all_tab_columns t WHERE t.owner LIKE :1 ESCAPE '/'
 AND t.table_name LIKE :2 ESCAPE '/'   AND t.column_name LIKE :3
 ESCAPE '/'  ORDER BY table_schem, table_name, ordinal_position
```

通过和开发部门的沟通，了解到**应用基础架构，就是从数据字典中查询相关信息（如表结构、字段及约束），然后动态拼装 SQL，继而执行并返回结果**，这种做法在系统负荷低时可能会无关紧要，但是，当业务系统并发量大、解析频繁时，这样的查询就可能带来严重的性能问题。

该条 SQL 的全文整理如下：

```
SELECT NULL AS table_cat,
       t.owner AS table_schem,
       t.table_name AS table_name,
       t.column_name AS column_name,
       DECODE (t.data_type, 'CHAR', 1, 'VARCHAR2', 12, 'NUMBER', 3,'LONG', -1, 'DATE', 93,
'RAW', -3, 'LONG RAW', -4, 1111) AS data_type,
       t.data_type AS type_name,
       DECODE (t.data_precision, null, t.data_length,t.data_precision) AS column_size,
       0 AS buffer_length,
       t.data_scale AS decimal_digits,
       10 AS num_prec_radix,
```

```
       DECODE (t.nullable, 'N', 0, 1) AS nullable,
       NULL AS remarks,
       t.data_default AS column_def,
       0 AS sql_data_type,
       0 AS sql_datetime_sub,
       t.data_length AS char_octet_length,
       t.column_id AS ordinal_position,
       DECODE (t.nullable, 'N', 'NO', 'YES') AS is_nullable
FROM all_tab_columns t
WHERE t.owner LIKE :1 ESCAPE '/'
  AND t.table_name LIKE :2 ESCAPE '/'
  AND t.column_name LIKE :3
  ESCAPE '/'
 ORDER BY table_schem, table_name, ordinal_position;
```

我们可以先来观察以下该 SQL 的执行计划：

```
SQL> explain plan for ....;
 Explained
SQL> select * from table(dbms_xplan.display());
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------
| Id  | Operation                              | Name               |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT                       |                    |
|   1 |  TABLE ACCESS BY INDEX ROWID           | OBJ$               |
|   2 |   INDEX RANGE SCAN                     | I_OBJ1             |
|   3 |  TABLE ACCESS BY INDEX ROWID           | OBJ$               |
|   4 |   INDEX RANGE SCAN                     | I_OBJ1             |
|   5 |  TABLE ACCESS BY INDEX ROWID           | OBJ$               |
|   6 |   INDEX RANGE SCAN                     | I_OBJ1             |
|   7 |  TABLE ACCESS BY INDEX ROWID           | OBJ$               |
|   8 |   INDEX RANGE SCAN                     | I_OBJ1             |
|   9 |  TABLE ACCESS BY INDEX ROWID           | OBJ$               |
|  10 |   INDEX RANGE SCAN                     | I_OBJ1             |
|  11 |  SORT ORDER BY                         |                    |
|  12 |   FILTER                               |                    |
|  13 |    NESTED LOOPS OUTER                  |                    |
|  14 |     NESTED LOOPS OUTER                 |                    |
|  15 |      NESTED LOOPS OUTER                |                    |
|  16 |       NESTED LOOPS OUTER               |                    |
```

```
|  17 |        NESTED LOOPS                   |                   |
|  18 |         NESTED LOOPS                  |                   |
|  19 |          NESTED LOOPS                 |                   |
|  20 |           TABLE ACCESS BY INDEX ROWID| USER$              |
|  21 |            INDEX UNIQUE SCAN          | I_USER1            |
|  22 |           TABLE ACCESS BY INDEX ROWID| OBJ$               |
|  23 |            INDEX RANGE SCAN           | I_OBJ5             |
|  24 |          TABLE ACCESS CLUSTER         | USER$              |
|  25 |           INDEX UNIQUE SCAN           | I_USER#            |
|  26 |         TABLE ACCESS CLUSTER          | COL$               |
|  27 |          INDEX UNIQUE SCAN            | I_OBJ#             |
|  28 |        TABLE ACCESS CLUSTER           | COLTYPE$           |
|  29 |       INDEX RANGE SCAN                | I_HH_OBJ#_INTCOL# |
|  30 |      TABLE ACCESS BY INDEX ROWID      | OBJ$               |
|  31 |       INDEX RANGE SCAN                | I_OBJ3             |
|  32 |     TABLE ACCESS CLUSTER              | USER$              |
|  33 |      INDEX UNIQUE SCAN                | I_USER#            |
|  34 |    TABLE ACCESS CLUSTER               | TAB$               |
|  35 |     INDEX UNIQUE SCAN                 | I_OBJ#             |
|  36 |    NESTED LOOPS                       |                    |
|  37 |     FIXED TABLE FULL                  | X$KZSRO            |
|  38 |     INDEX RANGE SCAN                  | I_OBJAUTH2         |
|  39 |    FIXED TABLE FULL                   | X$KZSPR            |
|  40 |    NESTED LOOPS                       |                    |
|  41 |     INDEX RANGE SCAN                  | I_OBJ4             |
|  42 |     TABLE ACCESS CLUSTER              | USER$              |
|  43 |      INDEX UNIQUE SCAN                | I_USER#            |
------------------------------------------------------------------
```

由于 all_tab_columns 是 sys 用户下的一个视图，从执行计划中可以看出，下面嵌套多层基础数据字典表，因此，执行计划较为复杂。由此我们应当获得一个基本常识：**在业务系统的底层，应该尽量避免频繁使用复杂的数据字典视图**。

对于这个案例，用户很难去修改底层的框架，所以只能尝试通过其他手段进行优化。由于这个查询是用于获得数据表的字段信息等，**对于已经长时间运行的业务系统，除了特殊的业务变更，基本上不可能再去修改表结构及增减字段**，因此，为了改善执行计划，我们想到了**用一张预先创建好的表来取代 all_tab_columns，精确的说应该是取代对 sys 下 all_tab_columns 的访问**。

我们计划将该表创建在应用用户下，取名为 all_tab_columns_temp，并使用 all_tab_columns 来作为它的同义词，使应用对 all_tab_columns 的访问需求从 all_tab_columns_temp 表中直接获

取。

由于 sys 下的 all_tab_columns 中存在 LONG 数据类型，因此，对 all_tab_columns_temp 表的创建采用 sqlplus 提供的 copy 命令来实现：

```
SQL> copy from oti/oti@fwx to oti/oti@fwx create all_tab_columns_temp using select * from all_tab_columns t where t.owner = 'OTI';
```

创建同义词：

```
SQL> create synonym all_tab_columns for oti.all_tab_columns_temp;
Synonym created
```

根据查询条件创建组合索引：

```
SQL> create index oti.all_tab_columns_temp_inx1 on
oti.all_tab_columns_temp(owner,table_name,column_name);
Index created
```

调整后的执行计划如下：

```
SQL> explain plan for ....;
 Explained
SQL> select * from table(dbms_xplan.display());
PLAN_TABLE_OUTPUT
-----------------------------------------------------------------
| Id  | Operation                     | Name                       |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT              |                            |
|   1 |  SORT ORDER BY                |                            |
|   2 |   TABLE ACCESS BY INDEX ROWID| ALL_TAB_COLUMNS_TEMP       |
|   3 |    INDEX RANGE SCAN           | ALL_TAB_COLUMNS_TEMP_INX1 |
-----------------------------------------------------------------
```

调整后，该 SQL 查询的逻辑读降低为 4，系统的性能获得了大幅度的提升。 在深入了解了 Oracle 的表、视图、同义词的逻辑之后，就可以据此做出很多有趣的尝试。

参考文献：

1.使用 SQL_TRACE 进行数据库诊断
http://www.eygle.com/case/Use.sql_trace.to.Diagnose.database.htm

2.Oracle 数据库创建脚本 sql.bsq 文件

3.Oracle Database Reference 10g Release 2 (10.2) B14237-02

4.关于数据库 open 的深入探究
http://www.itpub.net/199099.html

5. Oracle® Database Security Guide 10*g* Release 2 (10.2) B14266-02

6. 《Oracle DBA 手记 3》