MyDUL 测试流程和结果

作者　Tomcoding

2017 年 9 月 28 日

测试内容

1. 普通表测试
2. cluster 表测试
3. row dependencies 表测试
4. column unused 测试
5. column hidden 测试
6. column dropped 测试
7. chained row 测试
8. migratted row 测试
9. 7+8 联合测试
10. long column 测试
11. basic lob 测试
12. secure lob 测试
13. table partition 测试
14. table subpartition 测试
15. nchar 和 nvarchar2 测试
16. 中文测试，char、nchar、clob、nclob
17. ASSM Extent Map 测试
18. MSSM Extent Map 测试

测试步骤

普通表测试

在本节的测试中，我们创建两个 Oracle 数据库用户，一个是 andy，另一个是 jake，我们在用户 andy 下创建 4 张表，通过系统的数据字典视图插入数据。然后通过 mydul 程序把用户 andy 中的表导出来，然后通过 myimp 程序把数据装载到用户 jake 下，然后比较两个用

户下数据的差异，测试 mydul 程序导出数据的准确性。

创建表空间

使用缺省块大小为 8K 的表空间。

```
create tablespace ts_dul datafile 'ts_dul_01.dbf' size 100M autoextend on;
```

创建用户

使用上面创建的表空间做为用户的缺省表空间。

```
create user andy identified by andy default tablespace ts_dul;
```

为用户授权，使得用户能使用资源和查询任意数据字典

```
grant connect, resource to andy;
grant select any dictionary to andy;

create user jake identified by jake default tablespace ts_dul;
grant connect, resource to jake;
```

创建测试表

连接到用户 andy，通过系统视图创建普通表。

```
conn andy/andy

  CREATE TABLE MYOBJECT AS SELECT * FROM DBA_OBJECTS;
  CREATE TABLE MYPRIV AS SELECT * FROM DBA_SYS_PRIVS;
  CREATE TABLE MYSPACE AS SELECT * FROM DBA_TABLESPACES;
  CREATE TABLE MYTABLE AS SELECT * FROM DBA_TABLES;
```

连接到 Oracle 系统用户，执行 checkpoint 把数据写入到数据库文件中

```
conn / as sysdba
```

```
alter system checkpoint;
```

创建后,表中已经有数据了,通过 mydul 导出用户 andy 下的数据存储成文件,通过 myimp 导入到另一个用户 jake 中，通过 minus 比较两个用户下的数据是否一致。

导出用户数据

编辑 mydul 的配置文件,把系统表空间数据文件和刚创建的 ts_dul 表空间数据文件加进去。

```
[road@CentOS etc]$ cat config.ini
dictdir=/home/road/mydul/dict
datadir=/home/road/mydul/data
datafiles=/home/road/mydul/etc/dbfiles.list

[road@CentOS etc]$ cat dbfiles.list
/oracle11/oradata/ora11g/system01.dbf
/oracle11/oradata/ora11g/users01.dbf
/oracle11/product/11.2.0/dbs/road_ts01.dbf
/oracle11/product/11.2.0/dbs/ts_dul_01.dbf
```

在执行目录 bin 下运行

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini

MYDUL> export dict
MYDUL> load dict
MYDUL> unload user andy
```

导入用户数据

转到数据存放目录，看到导出文件名为 ANDY.dat 的数据文件，通过 myimp 装载到用户 jake 下。

```
[road@CentOS bin]$ cd ../data
```

```
[road@CentOS data]$ ls
ANDY.dat
[road@CentOS data]$ ../bin/myimp jake/jake file=ANDY.dat touser=jake
```

比较数据一致性

连接到系统用户

```
conn / as sysdba
SQL> select * from andy.myobject minus select * from jake.myobject;
no rows selected

SQL> select * from andy.mypriv minus select * from jake.mypriv;
no rows selected

SQL> select * from andy.myspace minus select * from jake.myspace;
no rows selected

SQL> select * from andy.mytable minus select * from jake.mytable;
no rows selected
```

如果上面的两个用户中表的数据有不一致的地方，就会显示不同的数据行出来。

结论

原始数据与 mydul 导出的数据一致，测试通过。

cluster 表测试

本节的测试我们创建两张 cluster 成员表，为了方便测试，并且能有做够数据，我们创建的表还是从数据字典视图中获取数据。用户仍然使用 andy 和 jake。

创建 cluster

连接到用户 andy。

```
conn andy/andy

create cluster c_object_id (object_id number);
create index i_object_id on cluster c_object_id;
```

创建成员表

连接到用户 andy。

```
conn andy/andy

create table clu_object
(
  owner           varchar2(30),
  object_name     varchar2(30),
  object_id       number,
  data_object_id  number,
  object_type     varchar2(19),
  created         date,
  status          varchar2(7)
)
cluster c_object_id (object_id);

create table clu_table
(
  object_id        number,
  owner            varchar2(30),
  table_name       varchar2(30),
  tablespace_name  varchar2(30),
  cluster_name     varchar2(30),
  status           varchar2(8),
  pct_free         number,
  pct_used         number,
  ini_trans        number,
  max_trans        number,
  logging          varchar2(3),
  degree           varchar2(10),
  partitioned      varchar2(3)
)
cluster c_object_id (object_id);
```

在成员表中插入数据

连接到用户 andy。

```
conn andy/andy

insert into clu_object
select
  owner, object_name, object_id, data_object_id,
```

```
  object_type, created, status
from dba_objects;

commit;

insert into clu_table
select
  o.object_id, t.owner, t.table_name, t.tablespace_name,
  t.cluster_name, t.status, t.pct_free, t.pct_used,
  t.ini_trans, t.max_trans, t.logging, t.degree, t.partitioned
from dba_objects o, dba_tables t
where o.owner = t.owner and o.object_name = t.table_name;

commit;

checkpoint 把数据刷新到磁盘中
SQL> conn / as sysdba
Connected.
SQL> alter system checkpoint;
System altered.
```

在第一个成员表 clu_object 中插入数据时，操作过程及其缓慢，这是由于 Oracle 要为 cluster 的块预留空间，造成磁盘空间消耗非常快，要不断分配 extent。所以数据量大的时候用 cluster 表是不明智的。上面的操作，如果要减小数据量，可以把 SYS 用户的对象剔除掉。

导出表数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini

MYDUL> export dict
MYDUL> load dict
MYDUL> unload table andy.clu_object
MYDUL> unload table andy.clu_table
```

导入表数据

```
[road@CentOS bin]$ cd ../data
[road@CentOS data]$ ls
```

```
ANDY_CLU_OBJECT.dat  ANDY_CLU_TABLE.dat
[road@CentOS data]$ ../bin/myimp jake/jake file=ANDY_CLU_OBJECT.dat touser=jake

[road@CentOS data]$ ../bin/myimp jake/jake file=ANDY_CLU_TABLE.dat touser=jake
```

注意 cluster 表在目标端创建后，成为普通表。

比较数据一致性

连接到系统用户

```
conn / as sysdba

SQL> select
  2    object_id, owner, object_name, data_object_id,
  3    object_type, created, status
  4  from andy.clu_object
  5  minus
  6  select
  7    object_id, owner, object_name, data_object_id,
  8    object_type, created, status
  9  from jake.clu_object;
no rows selected

SQL> select * from andy.clu_table minus select * from jake.clu_table;
no rows selected
```

第一个表由于建表时的字段顺序与源表不一致,所以要把查询的字段按一致的顺序排列才能进行比较。

多键 cluster

同上面的测试步骤，只是把 cluster 键设为 owner 和 object_name。

连接到用户 andy。

```
conn andy/andy
```

创建 cluster

```
create cluster c_owner_name (owner varchar2(30), object_name varchar2(30));
create index i_owner_name on cluster c_owner_name;
```

创建成员表

```
create table c_object
(
  owner           varchar2(30),
  object_name     varchar2(30),
  object_id       number,
  data_object_id  number,
  object_type     varchar2(19),
  created         date,
  status          varchar2(7)
)
cluster c_owner_name (owner, object_name);

create table c_table
(
  owner            varchar2(30),
  table_name       varchar2(30),
  tablespace_name  varchar2(30),
  cluster_name     varchar2(30),
  status           varchar2(8),
  pct_free         number,
  pct_used         number,
  ini_trans        number,
  max_trans        number,
  logging          varchar2(3),
  degree           varchar2(10),
  partitioned      varchar2(3)
)
cluster c_owner_name (owner, table_name);
```

插入数据

```
insert into c_object
select
  owner, object_name, object_id, data_object_id,
  object_type, created, status
from dba_objects where owner <> 'SYS';
commit;

insert into c_table
select
  owner, table_name, tablespace_name,
  cluster_name, status, pct_free, pct_used,
  ini_trans, max_trans, logging, degree, partitioned
```

```
from dba_tables
where owner <> 'SYS';
commit;
```

checkpoint 把数据刷新到磁盘中

```
SQL> conn / as sysdba
Connected.
SQL> alter system checkpoint;
System altered.
```

导出成员表数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini

MYDUL> export dict
MYDUL> load dict
MYDUL> unload table andy.c_object
MYDUL> unload table andy.c_table
```

导入成员表数据

```
[road@CentOS bin]$ cd ../data
[road@CentOS data]$ ls
ANDY_C_OBJECT.dat  ANDY_C_TABLE.dat
[road@CentOS data]$ ../bin/myimp jake/jake file=ANDY_C_OBJECT.dat touser=jake

[road@CentOS data]$ ../bin/myimp jake/jake file=ANDY_C_TABLE.dat touser=jake
```

注意 cluster 表在目标端创建后，成为普通表。

比较数据一致性

连接到系统用户

```
conn / as sysdba

SQL> conn / as sysdba
Connected.
SQL> select * from andy.c_object minus select * from jake.c_object;

no rows selected

SQL> select * from andy.c_table minus select * from jake.c_table;

no rows selected
```

用户 andy 和 jake 下的表 c_object 和 c_table 数据一致。

结论

cluster 成员表的原始数据与 mydul 导出的数据一致，测试通过。

删除 cluster

有两种方法，一种是按创建的相反顺序，把对象依次删除。

```
drop table clu_object purge;
drop table clu_table purge;
drop index i_object_id;
drop cluster c_object_id;
```

第二种方法是删除 cluster 连带一起删除成员表。

```
drop cluster c_object_id including tables;
```

行依赖表的测试

本节测试还在用户 andy 下进行。仍然从视图中获取数据。

创建表

连接到用户 andy，通过系统视图创建行依赖表。

```
conn andy/andy

create table rd_object rowdependencies as select * from dba_objects where rownum<1;
```

插入数据

```
declare
  v_obj  dba_objects%ROWTYPE;
  cursor c_obj is select * from dba_objects;
begin
  open c_obj;
  loop
    fetch c_obj into v_obj;
    exit when c_obj%NOTFOUND;
    insert into rd_object values (v_obj.owner, v_obj.object_name,
```

```
    v_obj.subobject_name, v_obj.object_id, v_obj.data_object_id,
    v_obj.object_type, v_obj.created, v_obj.last_ddl_time,
    v_obj.timestamp, v_obj.status, v_obj.temporary,
    v_obj.generated, v_obj.secondary, v_obj.namespace,
    v_obj.edition_name
    );
    commit;
  end loop;
  close c_obj;
end;
/
```

行依赖表中的每条数据的 ora_rowscn 是数据提交时的 SCN，为了使每行的 SCN 不一样，我们每插入一条数据，进行一次提交。

查询出 20 行看一下，结果如下：

```
SQL> select ora_rowscn, object_id from rd_object where rownum < 21;

ORA_ROWSCN  OBJECT_ID
---------- ----------
   4093190         85
   4093192         86
   4093194         87
   4093196         88
   4093198         89
   4093200         90
   4093202         91
   4093203         92
   4093205         93
   4093207         94
   4093209         95

ORA_ROWSCN  OBJECT_ID
---------- ----------
   4093211         96
   4093213         97
   4093215         98
   4093217         99
   4093218        100
   4093219        101
   4093220        102
   4093221        103
   4093223        104
```

```
20 rows selected.

checkpoint 把数据刷新到磁盘中
SQL> conn / as sysdba
Connected.
SQL> alter system checkpoint;
System altered.
```

导出表数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini

MYDUL> export dict
MYDUL> load dict
MYDUL> unload table andy.rd_object
```

导入表数据

```
[road@CentOS bin]$ cd ../data
[road@CentOS data]$ ls
ANDY_RD_OBJECT.dat
[road@CentOS data]$ ../bin/myimp jake/jake file=ANDY_RD_OBJECT.dat touser=jake
```

比较数据一致性

```
SQL> select * from andy.rd_object minus select * from jake.rd_object;
no rows selected
```

结论

原始数据与 mydul 导出的数据一致，测试通过。

在数据导入建表的过程中，只建立了表中简单的列关系，表的其他属性都忽略了，如果需要创建表的其他属性，可以手动建表，在导入数据时忽略建表错误即可。

从下面看到，用户 jake 下的 rd_object 表的 rowdependencies 属性没有了。

```
SQL> select dependencies from dba_tables
```

```
 where owner='ANDY' and table_name='RD_OBJECT';

DEPENDEN
--------
ENABLED

SQL> select dependencies from dba_tables
 where owner='JAKE' and table_name='RD_OBJECT';

DEPENDEN
--------
DISABLED
```

unused 列测试

本节测试中仍然使用 andy 用户，从 dba_objects 视图中获取数据，数据分两部分，一部分是正常表时的数据，然后 unuse 一列，把剩余数据插入。

创建表

```
SQL> select count(*) from dba_objects;

  COUNT(*)
----------
     71998

SQL> select count(*) from dba_objects where object_id < 20000;

  COUNT(*)
----------
     19514
```

先创建表，把 object_id 小于 20000 的数据导入进去。

```
create table un_object as select * from dba_objects where object_id < 20000;
```

unuse column

```
SQL> desc un_object;
 Name                                      Null?    Type
```

OWNER		VARCHAR2(30)
OBJECT_NAME		VARCHAR2(128)
SUBOBJECT_NAME		VARCHAR2(30)
OBJECT_ID		NUMBER
DATA_OBJECT_ID		NUMBER
OBJECT_TYPE		VARCHAR2(19)
CREATED		DATE
LAST_DDL_TIME		DATE
TIMESTAMP		VARCHAR2(19)
STATUS		VARCHAR2(7)
TEMPORARY		VARCHAR2(1)
GENERATED		VARCHAR2(1)
SECONDARY		VARCHAR2(1)
NAMESPACE		NUMBER
EDITION_NAME		VARCHAR2(30)

```
SQL> alter table un_object set unused column timestamp;

Table altered.

SQL> desc un_object;
```

Name	Null?	Type
OWNER		VARCHAR2(30)
OBJECT_NAME		VARCHAR2(128)
SUBOBJECT_NAME		VARCHAR2(30)
OBJECT_ID		NUMBER
DATA_OBJECT_ID		NUMBER
OBJECT_TYPE		VARCHAR2(19)
CREATED		DATE
LAST_DDL_TIME		DATE
STATUS		VARCHAR2(7)
TEMPORARY		VARCHAR2(1)
GENERATED		VARCHAR2(1)
SECONDARY		VARCHAR2(1)
NAMESPACE		NUMBER
EDITION_NAME		VARCHAR2(30)

插入剩余数据

```
insert into un_object
 select owner, object_name, subobject_name, object_id,
```

```
 data_object_id, object_type, created, last_ddl_time,
 status, temporary, generated, secondary, namespace, edition_name
 from dba_objects where object_id >= 20000;

commit;
```

checkpoint 把数据刷新到磁盘中

```
SQL> conn / as sysdba
Connected.
SQL> alter system checkpoint;
System altered.
```

导出表数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini

MYDUL> export dict
MYDUL> load dict
MYDUL> unload table andy.un_object
```

导入表数据

```
[road@CentOS bin]$ cd ../data
[road@CentOS data]$ ls
ANDY_UN_OBJECT.dat
[road@CentOS data]$  ../bin/myimp jake/jake file=ANDY_UN_OBJECT.dat touser=jake
```

比较数据一致性

```
SQL> conn / as sysdba
Connected.

SQL> select * from andy.un_object minus select * from jake.un_object;
no rows selected
```

结论

原始数据与 mydul 导出的数据一致，测试通过。

dropped 列测试

本节的测试步骤与 unused 列测试相似，只是在准备数据时把列 drop 掉。仍然使用 andy 用户。

数据准备

```
SQL> conn andy/andy
Connected.
SQL> create table dr_object as select * from dba_objects where object_id < 20000;

Table created.

SQL> desc dr_object
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 OWNER                                              VARCHAR2(30)
 OBJECT_NAME                                        VARCHAR2(128)
 SUBOBJECT_NAME                                     VARCHAR2(30)
 OBJECT_ID                                          NUMBER
 DATA_OBJECT_ID                                     NUMBER
 OBJECT_TYPE                                        VARCHAR2(19)
 CREATED                                            DATE
 LAST_DDL_TIME                                      DATE
 TIMESTAMP                                          VARCHAR2(19)
 STATUS                                             VARCHAR2(7)
 TEMPORARY                                          VARCHAR2(1)
 GENERATED                                          VARCHAR2(1)
 SECONDARY                                          VARCHAR2(1)
 NAMESPACE                                          NUMBER
 EDITION_NAME                                       VARCHAR2(30)

SQL> alter table dr_object drop column temporary;

Table altered.

SQL> desc dr_object;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 OWNER                                              VARCHAR2(30)
```

```
 OBJECT_NAME                                       VARCHAR2(128)
 SUBOBJECT_NAME                                    VARCHAR2(30)
 OBJECT_ID                                         NUMBER
 DATA_OBJECT_ID                                    NUMBER
 OBJECT_TYPE                                       VARCHAR2(19)
 CREATED                                           DATE
 LAST_DDL_TIME                                     DATE
 TIMESTAMP                                         VARCHAR2(19)
 STATUS                                            VARCHAR2(7)
 GENERATED                                         VARCHAR2(1)
 SECONDARY                                         VARCHAR2(1)
 NAMESPACE                                         NUMBER
 EDITION_NAME                                      VARCHAR2(30)

SQL> insert into dr_object
  2   select owner, object_name, subobject_name, object_id,
  3   data_object_id, object_type, created, last_ddl_time,
  4   timestamp, status, generated, secondary, namespace, edition_name
  5   from dba_objects where object_id >= 20000;

52487 rows created.

SQL> commit;

Commit complete.

SQL> conn / as sysdba
Connected.
SQL> alter system checkpoint;

System altered.
```

导出表数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini

MYDUL> export dict
MYDUL> load dict
MYDUL> unload table andy.dr_object
```

导入表数据

```
[road@CentOS bin]$ cd ../data
[road@CentOS data]$ ls
ANDY_DR_OBJECT.dat
[road@CentOS data]$  ../bin/myimp jake/jake file=ANDY_DR_OBJECT.dat touser=jake
```

比较数据一致性

```
SQL> conn / as sysdba
Connected.

SQL> select * from andy.dr_object minus select * from jake.dr_object;
no rows selected
```

结论

原始数据与 mydul 导出的数据一致，测试通过。

hidden 列测试

隐藏列有两种形式，一种是上面看到的 unused，查询时不显示，但是已有的列值还存储在数据段中。另一种是基于函数索引（function based index）的列，这是 Oracle 系统内部产生的列，可能需要在数据段中存储计算表达式的值，也可能不需要。

准备数据

```
SQL> conn andy/andy
Connected.
SQL> create table hi_object as select * from dba_objects where object_id<10000;

Table created.

SQL> select
  2   column_id col#, segment_column_id segcol#, internal_column_id intcol#,
  3   column_name, hidden_column
```

```
  4   from dba_tab_cols where owner='ANDY' and table_name='HI_OBJECT';

      COL#    SEGCOL#    INTCOL# COLUMN_NAME                    HID
---------- ---------- ---------- ------------------------------ ---
         1          1          1 OWNER                          NO
         2          2          2 OBJECT_NAME                    NO
         3          3          3 SUBOBJECT_NAME                 NO
         4          4          4 OBJECT_ID                      NO
         5          5          5 DATA_OBJECT_ID                 NO
         6          6          6 OBJECT_TYPE                    NO
         7          7          7 CREATED                        NO
         8          8          8 LAST_DDL_TIME                  NO
         9          9          9 TIMESTAMP                      NO
        10         10         10 STATUS                         NO
        11         11         11 TEMPORARY                      NO

      COL#    SEGCOL#    INTCOL# COLUMN_NAME                    HID
---------- ---------- ---------- ------------------------------ ---
        12         12         12 GENERATED                      NO
        13         13         13 SECONDARY                      NO
        14         14         14 NAMESPACE                      NO
        15         15         15 EDITION_NAME                   NO

15 rows selected.


SQL> create index f_sub_index on hi_object (substr(object_name, 1, 8));

Index created.

SQL> select
  2   column_id col#, segment_column_id segcol#, internal_column_id intcol#,
  3   column_name, hidden_column
  4   from dba_tab_cols where owner='ANDY' and table_name='HI_OBJECT';

      COL#    SEGCOL#    INTCOL# COLUMN_NAME                    HID
---------- ---------- ---------- ------------------------------ ---
         1          1          1 OWNER                          NO
         2          2          2 OBJECT_NAME                    NO
         3          3          3 SUBOBJECT_NAME                 NO
         4          4          4 OBJECT_ID                      NO
         5          5          5 DATA_OBJECT_ID                 NO
         6          6          6 OBJECT_TYPE                    NO
         7          7          7 CREATED                        NO
```

```
         8          8          8 LAST_DDL_TIME                  NO
         9          9          9 TIMESTAMP                      NO
        10         10         10 STATUS                         NO
        11         11         11 TEMPORARY                      NO

      COL#    SEGCOL#    INTCOL# COLUMN_NAME                    HID
---------- ---------- ---------- ------------------------------ ---
        12         12         12 GENERATED                      NO
        13         13         13 SECONDARY                      NO
        14         14         14 NAMESPACE                      NO
        15         15         15 EDITION_NAME                   NO
                              16 SYS_NC00016$                   YES

16 rows selected.
```

从上面看到多出一列来，这个隐藏列就是函数索引创建的列。

导出表数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini

MYDUL> export dict
MYDUL> load dict
MYDUL> unload table andy.hi_object
```

导入表数据

```
[road@CentOS bin]$ cd ../data
[road@CentOS data]$ ls
ANDY_HI_OBJECT.dat
[road@CentOS data]$  ../bin/myimp jake/jake file=ANDY_HI_OBJECT.dat touser=jake
```

比较数据一致性

```
SQL> conn / as sysdba
Connected.

SQL> select * from andy.hi_object minus select * from jake.hi_object;
no rows selected
```

结论

原始数据与 mydul 导出的数据一致，测试通过。

行连接测试

为了方便造成行连接，我们需要把表空间的数据块变小，设为 2K 大小。

创建表空间

由于后面还要用到这个表空间做 extent map 测试，所以把表空间的 extent 设成一致大小，并设置为最小值。

先把 db_2k_cache_size 设置打开，才能创建 2K 块大小的表空间。

```
conn / as sysdba;
alter system set db_2k_cache_size=10M;

create tablespace ts_small_block
  datafile 'ts_small_01.dbf' size 100M autoextend on
  blocksize 2048
  extent management local uniform size 10K;
```

创建用户

创建一个新用户 rock，使用上面的表空间。

```
create user rock identified by rock default tablespace ts_small_block;
```

为用户授权。

```
grant connect, resource to rock;
```

创建表

在这儿创建一个能人为产生行连接的表，让多个字段长度足够大，一个数据块不能完全存储一行数据。

```
conn rock/rock;

create table t_chain (id number, f1 varchar2(2000),
  f2 varchar2(2000), f3 varchar2(2000));
```

插入数据

```
begin
  for i in 1 .. 100 loop
    insert into t_chain values (i, rpad(i, 1000, 'a'),
      rpad(i, 1000, 'b'), rpad(i, 400, 'c'));
  end loop;
  commit;
end;
/
```

执行下面的语句，查看行连接的情况。

```
SQL> analyze table t_chain compute statistics;
Table analyzed.

SQL> select chain_cnt from user_tables where table_name='T_CHAIN';

 CHAIN_CNT
----------
       100
```

我们看到，所有插进去的 100 条数据都是行连接的。

导出表数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini

MYDUL> export dict
MYDUL> load dict
MYDUL> unload table rock.t_chain
```

导入表数据

```
[road@CentOS bin]$ cd ../data
[road@CentOS data]$ ls
ROCK_T_CHAIN.dat
```

```
[road@CentOS data]$ ../bin/myimp jake/jake file=ROCK_T_CHAIN.dat touser=jake
```

比较数据一致性

```
SQL> conn / as sysdba
Connected.
SQL> select * from rock.t_chain minus select * from jake.t_chain;

no rows selected
```

结论

原始数据与 mydul 导出的数据一致，测试通过。

行迁移测试

仍然使用行连接测试中用到的 2K 数据块表空间和用户 rock 进行行迁移测试。

创建表

```
create table mig_tab (id number, f1 varchar2(2000),
  f2 varchar2(2000), f3 varchar2(2000));
```

准备数据

每行插入 800 多字节的数据，这样每块中有两行数据。

```
begin
  for i in 1 .. 100 loop
    insert into mig_tab values (i, rpad(i, 200, 'a'),
      rpad(i, 200, 'b'), rpad(i, 400, 'c');
  end loop;
  commit;
end;
/

SQL> analyze table mig_tab compute statistics;
```

```
Table analyzed.

SQL> select chain_cnt from user_tables where table_name='MIG_TAB';

 CHAIN_CNT
----------
         0
```

从上面的结果看到，没有行连接和行迁移的数据。

把一个块中的第二行数据扩大到 1200 字节，这样第二行数据就会被迁移出去。

```
begin
  for i in 1 .. 50 loop
    update mig_tab set f1=rpad(i*2, 600, 'a') where id=i*2;
  end loop;
  commit;
end;
/
SQL> analyze table mig_tab compute statistics;

Table analyzed.

SQL> select chain_cnt from user_tables where table_name='MIG_TAB';

 CHAIN_CNT
----------
        50
```

在这儿看到，有 50 行数据迁移出了原来的块。

如果要看得更精确些，可以用 analyze table mig_tab list chained rows 语句分析后，查看 chained_rows 表的结果。

先创建 chained_rows 表

```
create table CHAINED_ROWS (
  owner_name         varchar2(30),
  table_name         varchar2(30),
  cluster_name       varchar2(30),
  partition_name     varchar2(30),
  subpartition_name  varchar2(30),
  head_rowid         urowid,
  analyze_timestamp  date
);

SQL> analyze table mig_tab list chained rows;
```

```
Table analyzed.

SQL> select m.id, m.rowid from mig_tab m, chained_rows c
  2    where m.rowid = c.head_rowid
  3    order by m.id;

        ID ROWID
---------- ------------------
         2 AAASdMAAKAAAAEAAAB
         4 AAASdMAAKAAAAD/AAB
         6 AAASdMAAKAAAAEFAAB
         8 AAASdMAAKAAAAEBAAB
        10 AAASdMAAKAAAAECAAB
        12 AAASdMAAKAAAAEEAAB
        14 AAASdMAAKAAAAEDAAB
        16 AAASdMAAKAAAAEKAAB
        18 AAASdMAAKAAAAEGAAB
        20 AAASdMAAKAAAAEHAAB
        22 AAASdMAAKAAAAEJAAB

        ID ROWID
---------- ------------------
        24 AAASdMAAKAAAAEIAAB
        26 AAASdMAAKAAAAEPAAB
        28 AAASdMAAKAAAAEMAAB
        30 AAASdMAAKAAAAEOAAB
        32 AAASdMAAKAAAAENAAB
        34 AAASdMAAKAAAAEUAAB
        36 AAASdMAAKAAAAEQAAB
        38 AAASdMAAKAAAAERAAB
        40 AAASdMAAKAAAAETAAB
        42 AAASdMAAKAAAAESAAB
        44 AAASdMAAKAAAAEZAAB

        ID ROWID
---------- ------------------
        46 AAASdMAAKAAAAEVAAB
        48 AAASdMAAKAAAAEWAAB
        50 AAASdMAAKAAAAEYAAB
        52 AAASdMAAKAAAAEXAAB
        54 AAASdMAAKAAAAEeAAB
        56 AAASdMAAKAAAAEbAAB
        58 AAASdMAAKAAAAEdAAB
```

```
        60 AAASdMAAKAAAAEcAAB
        62 AAASdMAAKAAAAEjAAB
        64 AAASdMAAKAAAAEfAAB
        66 AAASdMAAKAAAAEgAAB

        ID ROWID
---------- ------------------
        68 AAASdMAAKAAAAEiAAB
        70 AAASdMAAKAAAAEhAAB
        72 AAASdMAAKAAAAEoAAB
        74 AAASdMAAKAAAAEkAAB
        76 AAASdMAAKAAAAElAAB
        78 AAASdMAAKAAAAEnAAB
        80 AAASdMAAKAAAAEmAAB
        82 AAASdMAAKAAAAEtAAB
        84 AAASdMAAKAAAAEqAAB
        86 AAASdMAAKAAAAEsAAB
        88 AAASdMAAKAAAAErAAB

        ID ROWID
---------- ------------------
        90 AAASdMAAKAAAAEyAAB
        92 AAASdMAAKAAAAEuAAB
        94 AAASdMAAKAAAAEvAAB
        96 AAASdMAAKAAAAExAAB
        98 AAASdMAAKAAAAEwAAB
       100 AAASdMAAKAAAAE3AAB

50 rows selected.
```

这样就能看到具体是哪些行发生了迁移。

别忘了 checkpoint 数据。

```
SQL> conn / as sysdba
Connected.
SQL> alter system checkpoint;

System altered.
```

导出表数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini

MYDUL> export dict
MYDUL> load dict
```

```
MYDUL> unload table rock.mig_tab
```

导入表数据

```
[road@CentOS bin]$ cd ../data
[road@CentOS data]$ ls
ROCK_MIG_TAB.dat
[road@CentOS data]$ ../bin/myimp jake/jake file=ROCK_MIG_TAB.dat touser=jake
```

比较数据一致性

```
SQL> conn / as sysdba
Connected.
SQL> select * from rock.mig_tab minus select * from jake.mig_tab;
no rows selected
```

结论

原始数据与 mydul 导出的数据一致，测试通过。

行迁移后连接的测试

本节测试是把前面行连接和行迁移综合起来进行测试，行迁移后，再次出现行连接，行头数据会有新的变化,本节测试针对这种情况。本节测试在上一节行迁移测试的基础上进行，在上一节中，偶数行已经是行迁移状态了，我们只需把它再变成行连接就可以了。

数据准备

在上一节形成的连接行，f1 是 600 字节，f2 是 200 字节，f3 是 400 字节。如果把 f2 更新成 1000 字节，那么一行数据总共就有 2000 多字节，在加上块头的和其他空间，那么在一个块中肯定是存不下了，就会出现行连接。

从上面的描述中看到，在理论上应该会出现行迁移后再连接的情况，但在实际操作中，发现 Oracle 不希望这种情况发生，总是千方百计让数据存储在最少的块中。

试试制造一个单条的行迁移后连接的数据行。

还是用 mig_tab 表，把数据都删除掉，重新执行下面的操作，插入三行数据。

```
insert into mig_tab values (1, rpad(1, 400, 'a'), rpad(1, 400, 'b'), null);
insert into mig_tab values (2, rpad(2, 400, 'a'), rpad(2, 400, 'b'), null);
insert into mig_tab values (3, rpad(3, 400, 'a'), rpad(3, 400, 'b'), null);
commit;
```

看看这三条数据都在哪个块中。

```
select
  id,
  dbms_rowid.rowid_relative_fno(rowid) rfn,
  dbms_rowid.rowid_block_number(rowid) blk#
from mig_tab;
```

```
SQL> select
  2    id,
  3    dbms_rowid.rowid_relative_fno(rowid) rfn,
  4    dbms_rowid.rowid_block_number(rowid) blk#
  5  from mig_tab;

        ID        RFN       BLK#
---------- ---------- ----------
         1         10        394
         2         10        394
         3         10        395
```

看到前两行数据在 394 块中，那么我们修改 id=2 的数据，使它成为迁移行。

```
update mig_tab set f2=rpad(2, 1000, 'b') where id=2;
commit;
```

把 394 块 dump 出来看一下，在这儿把主要的粘贴过来。

```
tab 0, row 1, @0x128
tl: 9 fb: --H----- lb: 0x2  cc: 0
nrid:  0x0280018e.0
end_of_block_dump
End dump data blocks tsn: 10 file#: 10 minblk 394 maxblk 394
```

看到原来的行迁移到了 0x0280018e.0 处，也就是 0x0280018e<<10>>10=398 块的第一行。
执行下面的语句，使这一行变成连接行。

```
update mig_tab set f3=rpad(2, 1000, 'c') where id=2;
commit;
```

把 398 块 dump 出来看一下。

```
tab 0, row 0, @0x1f0
tl: 1424 fb: ----F--- lb: 0x1  cc: 3
hrid: 0x0280018a.1
nrid:  0x0280018d.0
col  0: [ 2]  c1 03
col  1: [400]
```

终于看到行连接了，看看行头既有 hrid 又有 nrid，正是我们希望出现的情况。先拿这三条数据测试一下吧。

导出表数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini

MYDUL> export dict
MYDUL> load dict
MYDUL> unload table rock.mig_tab
```

导入表数据

```
[road@CentOS bin]$ cd ../data
[road@CentOS data]$ ls
ROCK_MIG_TAB.dat
[road@CentOS data]$ ../bin/myimp jake/jake file=ROCK_MIG_TAB.dat touser=jake
```

比较数据一致性

```
SQL> conn / as sysdba
Connected.
SQL> select * from rock.mig_tab minus select * from jake.mig_tab;
no rows selected
```

结论

原始数据与 mydul 导出的数据一致，测试通过。

LONG 字段表测试

本节测试还是使用块大小为 2K 的表空间，用户 rock 来进行测试，这样可以造成更多的连接块，是测试深入。测试分三种情况，第一，long 数据小于 200 字节，当做普通 varchar2 测试；第二，long 数据比较大，造成行连接，并且连接多个数据块；第三，long 字段后还有其他字段，long 数据比较大。

创建表

```
create table long_tab (id number, f1 varchar2(200), f2 varchar2(400),
  f3 varchar2(200), f1 long);
```

插入数据

```
begin
  for i in 1 .. 40 loop
    insert into long_tab values (i, rpad(i, 100, 'a'), rpad(i, 100, 'b'),
      rpad(i, 100, 'c'), rpad(i, 200, 'l'));
  end loop;
  commit;
end;
/
```

checkpoint 数据

导出表数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini

MYDUL> export dict
MYDUL> load dict
MYDUL> unload table rock.long_tab
```

导入表数据

```
[road@CentOS bin]$ cd ../data
```

```
[road@CentOS data]$ ls
ROCK_LONG_TAB.dat
[road@CentOS data]$ ../bin/myimp jake/jake file=ROCK_LONG_TAB.dat touser=jake
```

比较数据一致性

有 long 字段的表，不能直接用 select * 查询比较了。
先来比较一下非 long 字段的数据。

```
SQL> select id, f1, f2, f3 from rock.long_tab
  2  minus
  3  select id, f1, f2, f3 from jake.long_tab;

no rows selected
```

用下面的 PL/SQL 检查 long 字段的一致性

```
set serveroutput on size 10000;

declare
  match  boolean;
  v_id1  number;
  v_id2  number;
  v_fl1  varchar2(20000);
  v_fl2  varchar2(20000);
  cursor c1 is select id, fl from rock.long_tab order by id;
  cursor c2 is select id, fl from jake.long_tab order by id;
begin
  match := true;
  open c1;
  open c2;
  loop
    fetch c1 into v_id1, v_fl1;
    exit when c1%NOTFOUND;
    fetch c2 into v_id2, v_fl2;
    if v_fl1 <> v_fl2 then
      match := false;
      dbms_output.put_line('row id='||v_id1||' long column not matach.');
    end if;
  end loop;
  close c1;
  close c2;
  if match then
    dbms_output.put_line('all long column data matched.');
  end if;
```

```
end;
/

SQL> declare
  2    match  boolean;
  3    v_id1  number;
  4    v_id2  number;
  5    v_fl1  varchar2(20000);
.............................
 27  end;
 28  /
all long column data matched.

PL/SQL procedure successfully completed.
```

大数据量测试

在上面的基础上，先把数据删除，然后插入数据。

```
SQL> delete from long_tab;

40 rows deleted.

SQL> commit;

Commit complete.
```

插入数据

```
begin
  for i in 1 .. 40 loop
    insert into long_tab values (i, rpad(i, 100, 'a'), rpad(i, 100, 'b'),
      rpad(i, 100, 'c'), rpad(i, 30000, 'l'));
  end loop;
  commit;
end;
/
```

checkpoint 数据

用前面的方法导出数据，生成文件 ROCK_LONG_TAB.dat。

在用户 jake 下，先把 long_tab 中的数据删除

用下面的命令导入数据

```
[road@CentOS data]$ ../bin/myimp jake/jake file=ROCK_LONG_TAB.dat touser=jake ignore=y
```

用前面的方法比较数据，先比较非 long 字段，再单独比较 long 字段数据。

```
SQL> select id, f1, f2, f3 from rock.long_tab
  2  minus
  3  select id, f1, f2, f3 from jake.long_tab;

no rows selected

SQL> declare
  2    match  boolean;
  3    v_id1  number;
  4    v_id2  number;
  5    v_fl1  varchar2(32000);
  6    v_fl2  varchar2(32000);
  7    cursor c1 is select id, fl from rock.long_tab order by id;
  8    cursor c2 is select id, fl from jake.long_tab order by id;
  9  begin
 10    match := true;
 11    open c1;
 12    open c2;
 13    loop
 14      fetch c1 into v_id1, v_fl1;
 15      exit when c1%NOTFOUND;
 16      fetch c2 into v_id2, v_fl2;
 17      if v_fl1 <> v_fl2 then
 18        match := false;
 19        dbms_output.put_line('row id='||v_id1||' long column not matach.');
 20      end if;
 21    end loop;
 22    close c1;
 23    close c2;
 24    if match then
 25      dbms_output.put_line('all long column data matched.');
 26    end if;
 27  end;
 28  /
all long column data matched.

PL/SQL procedure successfully completed.
```

两个表中的数据一致。

long 字段不在最后

在上面测试的基础上，在表中添加两个字段，这样 LONG 字段后面就有了新字段。
准备数据：

```
alter table long_tab add (f4 varchar2(200), f5 varchar2(200));
```

插入新数据

```
begin
  for i in 41 .. 60 loop
    insert into long_tab values (i, rpad(i, 100, 'a'), rpad(i, 100, 'b'),
      rpad(i, 100, 'c'), rpad(i, 6000, 'l'), rpad(i, 40, 'd'), rpad(i, 40, 'e'));
  end loop;
  commit;
end;
/
```

checkpoint 数据

导出数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini

MYDUL> export dict
MYDUL> load dict
MYDUL> unload table rock.long_tab
```

导入数据

```
[road@CentOS data]$ ls
ROCK_LONG_TAB.dat
[road@CentOS data]$ ../bin/myimp jake/jake file=ROCK_LONG_TAB.dat touser=jake
ignore=y
```

用前面的方法比较数据，先比较非 long 字段，再单独比较 long 字段数据。

```
SQL> select id, f1, f2, f3, f4, f5 from rock.long_tab
  2  minus
  3  select id, f1, f2, f3, f4, f5 from jake.long_tab;

no rows selected
```

比较 LONG 字段数据。

```
SQL> set serveroutput on size 10000;
SQL> declare
```

```
  2    match  boolean;
  3    v_id1  number;
  4    v_id2  number;
  5    v_fl1  varchar2(32000);
  6    v_fl2  varchar2(32000);
  7    cursor c1 is select id, fl from rock.long_tab order by id;
  8    cursor c2 is select id, fl from jake.long_tab order by id;
  9  begin
 10    match := true;
 11    open c1;
 12    open c2;
 13    loop
 14      fetch c1 into v_id1, v_fl1;
 15      exit when c1%NOTFOUND;
 16      fetch c2 into v_id2, v_fl2;
 17      if v_fl1 <> v_fl2 then
 18        match := false;
 19        dbms_output.put_line('row id='||v_id1||' long column not matach.');
 20      end if;
 21    end loop;
 22    close c1;
 23    close c2;
 24    if match then
 25      dbms_output.put_line('all long column data matched.');
 26    end if;
 27  end;
 28  /
all long column data matched.

PL/SQL procedure successfully completed.
```

结论

原始数据与 mydul 导出的数据一致，测试通过。

BASIC LOB 测试

本节测试 basicfile 的 LOB 字段表的导出。共有四个方面要测试，第一，测试 in row 的 LOB 数据；第二，chunk 小于 12 个的测试；第三，chunk 大于 12 个的测试；第四，out row LOB 的测试。

in row LOB

创建表，仍然使用 rock 用户。

```
create table in_lob (id number, f1 varchar2(200), fc clob);
```

插入数据

```
begin
  for i in 1 .. 20 loop
    insert into in_lob values (i, rpad(i, 20, 'a'), rpad(i, 600, 'l'));
  end loop;
  commit;
end;
/
```

导出数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini

MYDUL> export dict
MYDUL> load dict
MYDUL> unload table rock.in_lob
```

导入数据

```
[road@CentOS data]$ ls
ROCK_IN_LOB.dat
[road@CentOS data]$ ../bin/myimp jake/jake file=ROCK_IN_LOB.dat touser=jake
```

比较数据

先比较非 LOB 字段

```
SQL> select id, f1 from rock.in_lob minus select id, f1 from jake.in_lob;

no rows selected
```

用下面的 PL/SQL 比较 LOB 字段

```
set serveroutput on size 10000;

declare
  match  boolean;
  v_id1  number;
  v_id2  number;
  lob1   clob;
  lob2   clob;
  cursor c1 is select id, fc from rock.in_lob order by id;
```

```
  cursor c2 is select id, fc from jake.in_lob order by id;
begin
  match := true;
  open c1;
  open c2;
  loop
    fetch c1 into v_id1, lob1;
    exit when c1%NOTFOUND;
    fetch c2 into v_id2, lob2;
    if dbms_lob.compare(lob1, lob2) <> 0 then
      match := false;
      dbms_output.put_line('row id='||v_id1||' lob column not matach.');
    end if;
  end loop;
  close c1;
  close c2;
  if match then
    dbms_output.put_line('');
    dbms_output.put_line('all lob column data matched.');
  end if;
end;
/
```

比较结果如下

```
SQL> declare
  2    match  boolean;
  3    v_id1  number;
  4    v_id2  number;
  5    lob1   clob;
  6    lob2   clob;
  7    cursor c1 is select id, fc from rock.in_lob order by id;
  8    cursor c2 is select id, fc from jake.in_lob order by id;
  9  begin
 10    match := true;
 11    open c1;
 12    open c2;
 13    loop
 14      fetch c1 into v_id1, lob1;
 15      exit when c1%NOTFOUND;
 16      fetch c2 into v_id2, lob2;
 17      if dbms_lob.compare(lob1, lob2) <> 0 then
 18        match := false;
 19        dbms_output.put_line('row id='||v_id1||' lob column not matach.');
 20      end if;
```

```
 21    end loop;
 22    close c1;
 23    close c2;
 24    if match then
 25      dbms_output.put_line('');
 26      dbms_output.put_line('all lob column data matched.');
 27    end if;
 28  end;
 29  /
all lob column data matched.

PL/SQL procedure successfully completed.
```

小于 12 个 chunk

在上面创建的表中，一个 chunk 占用一个数据块，一个块是 2048 字节，减去 LOB 块的头和尾的 60 字节，每个块最多能存储 1988 个字节，12 个块最多存储 23856 字节，每个字符用两个字节表示，那么最多存储 11928 个字符，只要我们插入 LOB 的字符数少于 11928，那么 chunk 地址都在行内存储。

我们还是使用上面已经创建的 in_lob 表，把数据删除，重新插入，LOB 字符插入 8000 个。由于 LOB 字段只能直接插入 4000 个字符，所以需要用到 DBMS_LOB 包来写入 LOB 数据。

```
declare
  v_data  varchar2(32767);
  v_clob  clob;
begin
  for i in 1 .. 20 loop
    insert into in_lob values (i, rpad(i, 20, 'a'), empty_clob)
      returning fc into v_clob;
    v_data := rpad(i, 8000, 'l');
    dbms_lob.writeappend(v_clob, 8000, v_data);
  end loop;
  commit;
end;
/

SQL> conn rock/rock
Connected.
SQL> delete from in_lob;

20 rows deleted.
```

```
SQL> commit;

Commit complete.

SQL> declare
  2    v_data  varchar2(32767);
  3    v_clob  clob;
  4  begin
  5    for i in 1 .. 20 loop
  6      insert into in_lob values (i, rpad(i, 20, 'a'), empty_clob)
  7        returning fc into v_clob;
  8      v_data := rpad(i, 8000, 'l');
  9      dbms_lob.writeappend(v_clob, 8000, v_data);
 10    end loop;
 11    commit;
 12  end;
 13  /

PL/SQL procedure successfully completed.
```

依照前面的步骤导出数据，然后导入到用户 jake 下
比较数据

```
SQL> select id, f1 from rock.in_lob minus select id, f1 from jake.in_lob;

no rows selected

SQL> set serveroutput on size 10000;
SQL> declare
  2    match  boolean;
  3    v_id1  number;
  4    v_id2  number;
  5    lob1   clob;
  6    lob2   clob;
  7    cursor c1 is select id, fc from rock.in_lob order by id;
  8    cursor c2 is select id, fc from jake.in_lob order by id;
  9  begin
 10    match := true;
 11    open c1;
 12    open c2;
 13    loop
 14      fetch c1 into v_id1, lob1;
 15      exit when c1%NOTFOUND;
 16      fetch c2 into v_id2, lob2;
```

```
 17      if dbms_lob.compare(lob1, lob2) <> 0 then
 18        match := false;
 19        dbms_output.put_line('row id='||v_id1||' lob column not matach.');
 20      end if;
 21    end loop;
 22    close c1;
 23    close c2;
 24    if match then
 25      dbms_output.put_line('');
 26      dbms_output.put_line('all lob column data matched.');
 27    end if;
 28  end;
 29  /
all lob column data matched.

PL/SQL procedure successfully completed.
```

大于 12 个 chunk

根据对小于 12 个 chunk 的 lob 存储的分析，加大字符个数就能使 chunk 数大于 12，需要通过 LOB index 才能完全导出 LOB 数据。

还是使用 rock 用户下的 in_lob 表，插入 LOB 字符数为 25000 个。

```
SQL> show user
USER is "ROCK"
SQL> delete from in_lob;

20 rows deleted.

SQL> commit;

Commit complete.

SQL> declare
  2    v_data  varchar2(32767);
  3    v_clob  clob;
  4  begin
  5    for i in 1 .. 20 loop
  6      insert into in_lob values (i, rpad(i, 20, 'a'), empty_clob)
  7        returning fc into v_clob;
  8      v_data := rpad(i, 25000, 'l');
```

```
  9      dbms_lob.writeappend(v_clob, 25000, v_data);
 10    end loop;
 11    commit;
 12  end;
 13  /

PL/SQL procedure successfully completed.
```

依照前述步骤导出和导入数据。
比较数据，结果如下：

```
SQL> select id, f1 from rock.in_lob minus select id, f1 from jake.in_lob;

no rows selected

SQL> set serveroutput on size 10000;
SQL>
SQL> declare
  2    match  boolean;
  3    v_id1  number;
  4    v_id2  number;
  5    lob1   clob;
  6    lob2   clob;
  7    cursor c1 is select id, fc from rock.in_lob order by id;
  8    cursor c2 is select id, fc from jake.in_lob order by id;
  9  begin
 10    match := true;
 11    open c1;
 12    open c2;
 13    loop
 14      fetch c1 into v_id1, lob1;
 15      exit when c1%NOTFOUND;
 16      fetch c2 into v_id2, lob2;
 17      if dbms_lob.compare(lob1, lob2) <> 0 then
 18        match := false;
 19        dbms_output.put_line('row id='||v_id1||' lob column not matach.');
 20      end if;
 21    end loop;
 22    close c1;
 23    close c2;
 24    if match then
 25      dbms_output.put_line('');
 26      dbms_output.put_line('all lob column data matched.');
 27    end if;
 28  end;
```

```
 29  /
all lob column data matched.

PL/SQL procedure successfully completed.
```

out row LOB

仍然在 rock 用户下测试，创建 out row 的 LOB 字段表。

```
create table out_lob (id number, f1 varchar2(100), fco clob)
  lob (fco) store as (disable storage in row);
```

插入数据，在 CLOB 字段中插入 12000 个字符。

```
declare
  v_data  varchar2(32767);
  v_clob  clob;
begin
  for i in 1 .. 20 loop
    insert into out_lob values (i, rpad(i, 20, 'a'), empty_clob)
      returning fco into v_clob;
    v_data := rpad(i, 12000, 'l');
    dbms_lob.writeappend(v_clob, 12000, v_data);
  end loop;
  commit;
end;
/
```

依照前面的方法导出和导入数据。

比较数据一致性，结果如下：

```
SQL> select id, f1 from rock.in_lob minus select id, f1 from jake.in_lob;

no rows selected

SQL> set serveroutput on size 10000;

SQL> SQL> declare
  2    match  boolean;
  3    v_id1  number;
  4    v_id2  number;
  5    lob1   clob;
  6    lob2   clob;
  7    cursor c1 is select id, fco from rock.out_lob order by id;
  8    cursor c2 is select id, fco from jake.out_lob order by id;
  9  begin
 10    match := true;
```

```
 11    open c1;
 12    open c2;
 13    loop
 14      fetch c1 into v_id1, lob1;
 15      exit when c1%NOTFOUND;
 16      fetch c2 into v_id2, lob2;
 17      if dbms_lob.compare(lob1, lob2) <> 0 then
 18        match := false;
 19        dbms_output.put_line('row id='||v_id1||' lob column not matach.');
 20      end if;
 21    end loop;
 22    close c1;
 23    close c2;
 24    if match then
 25      dbms_output.put_line('');
 26      dbms_output.put_line('all lob column data matched.');
 27    end if;
 28  end;
 29  /
all lob column data matched.

PL/SQL procedure successfully completed.
```

Secure LOB 测试

本节测试 SecureFile LOB 字段表的导出，测试分三个部分，第一，In row LOB；第二，In row LOB 超出 4000 字节，chunk 地址在行内；第三，In row LOB 数据比较大，头块地址在行内。至于 Out row 的 LOB 存储方式与 In Row LOB 的后两种方式一样，就不测试了。

测试中继续使用 rock 用户。

in row LOB

先创建一个 SecureFile LOB 字段的表。

```
create table in_sec_lob (id number, f1 varchar2(100), fsc clob)
  lob (fsc) store as securefile;
```

插入数据

```
begin
  for i in 1 .. 20 loop
```

```
    insert into in_sec_lob values (i, rpad(i, 20, 'a'), rpad(i, 400, 'l'));
  end loop;
  commit;
end;
/
```

这儿插入数据时产生错误，ORA-60019: Creating initial extent of size 5 in tablespace of extent size 14，用户 rock 所在的表空间初始 extent 太小。重新建一个表空间，重建一个新用户测试 SecureFile LOB。

创建表空间 ts_secure 作为测试默认表空间

```
create tablespace ts_secure
  datafile 'ts_secure_01.dbf' size 100M autoextend on
  blocksize 2048
  extent management local uniform size 32768;
```

创建用户 secu 作为测试用户

```
create user secu identified by secu default tablespace ts_secure;
```

为 secu 用户赋权限

```
grant connect, resource to secu;
```

创建包含 SecureFile LOB 字段的测试表

```
create table in_sec_lob (id number, f1 varchar2(100), fsc clob)
  lob (fsc) store as securefile;
```

插入行内数据

```
begin
  for i in 1 .. 20 loop
    insert into in_sec_lob values (i, rpad(i, 20, 'a'), rpad(i, 400, 'l'));
  end loop;
  commit;
end;
/
```

checkpoint 数据

```
SQL> conn / as sysdba
Connected.
SQL> alter system checkpoint;

System altered.
```

导出数据，导出前要把新的数据文件 ts_secure_01.dbf 加到配置中

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini
```

```
MYDUL> export dict
MYDUL> load dict
MYDUL> unload table secu.in_sec_lob
```

导入数据

```
[road@CentOS data]$ ls
SECU_IN_SEC_LOB.dat
[road@CentOS data]$ ../bin/myimp jake/jake file=SECU_IN_SEC_LOB.dat touser=jake
```

比较数据，与 Basic LOB 比较方法一样

```
SQL> conn / as sysdba
Connected.
SQL> select id, f1 from secu.in_sec_lob minus select id, f1 from jake.in_sec_lob;

no rows selected
```

比较 LOB 字段

```
SQL> set serveroutput on size 10000;

declare
  match  boolean;
SQL> SQL>   2    3    v_id1  number;
  4    v_id2  number;
  5    lob1   clob;
  6    lob2   clob;
  7    cursor c1 is select id, fsc from secu.in_sec_lob order by id;
  8    cursor c2 is select id, fsc from jake.in_sec_lob order by id;
  9  begin
 10    match := true;
 11    open c1;
 12    open c2;
 13    loop
 14      fetch c1 into v_id1, lob1;
 15      exit when c1%NOTFOUND;
 16      fetch c2 into v_id2, lob2;
 17      if dbms_lob.compare(lob1, lob2) <> 0 then
 18        match := false;
 19        dbms_output.put_line('row id='||v_id1||' lob column not matach.');
 20      end if;
 21    end loop;
 22    close c1;
 23    close c2;
```

```
 24    if match then
 25      dbms_output.put_line('');
 26      dbms_output.put_line('all lob column data matched.');
 27    end if;
 28  end;
 29  /
all lob column data matched.

PL/SQL procedure successfully completed.
```

in row chunk

把 in_sec_lob 表中的数据删除，然后插入超过 4000 字节的数据。

准备数据

```
delete from in_sec_lob
commit;

declare
  v_data  varchar2(32767);
  v_clob  clob;
begin
  for i in 1 .. 20 loop
    insert into in_sec_lob values (i, rpad(i, 40, 'a'), empty_clob)
      returning fsc into v_clob;
    v_data := rpad(i, 16000, 'l');
    dbms_lob.writeappend(v_clob, 16000, v_data);
  end loop;
  commit;
end;
/

SQL> conn / as sysdba
Connected.
SQL> alter system checkpoint;

System altered.
```

dump 出一个数据块来，看一下是否 LOB chunk 地址在行内。

```
select
  dbms_rowid.rowid_relative_fno(rowid) rfn,
  dbms_rowid.rowid_block_number(rowid) blk#
```

```
from secu.in_sec_lob
where id=1;

SQL> select
  2    dbms_rowid.rowid_relative_fno(rowid) rfn,
  3    dbms_rowid.rowid_block_number(rowid) blk#
  4  from secu.in_sec_lob
  5  where id=1;

       RFN       BLK#
---------- ----------
         9        521

SQL> alter system dump datafile 9 block 521;

System altered.

tab 0, row 0, @0x736
tl: 98 fb: --H-FL-- lb: 0x1  cc: 3
col  0: [ 2]  c1 02
col  1: [40]
 31 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61
 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61
col  2: [50]
 00 54 00 01 02 0c 80 80 00 02 00 00 00 01 00 00 00 0b e7 45 00 1e 40 90 00
 18 21 00 7d 00 01 02 01 02 40 02 1f 01 01 02 40 02 51 0f 01 02 40 02 61 01
LOB
Locator:
  Length:        84(50)
  Version:        1
  Byte Length:    2
  LobID: 00.00.00.01.00.00.00.0b.e7.45
  Flags[ 0x02 0x0c 0x80 0x80 ]:
    Type: CLOB
    Storage: SecureFile
    Characterset Format: IMPLICIT
    Partitioned Table: No
    Options: VaringWidthReadWrite
  SecureFile Header:
    Length:   30
    Old Flag: 0x40 [ SecureFile ]
    Flag 0:   0x90 [ INODE Valid ]
    Layers:
      Lengths Array: INODE:24
```

```
INODE:
  21 00 7d 00 01 02 01 02 40 02 1f 01 01 02 40 02 51 0f 01 02
  40 02 61 01
```

从上面看到 inode 的 flag1 标志是 0x21，表示行内存储的是 chunk 地址，在后面还可以看到一共有 3 个 chunk。

导出数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini

MYDUL> export dict
MYDUL> load dict
MYDUL> unload table secu.in_sec_lob
```

导入数据

```
[road@CentOS data]$ ../bin/myimp jake/jake file=SECU_IN_SEC_LOB.dat touser=jake
ignore=y
```

比较数据

```
SQL> conn / as sysdba
Connected.
SQL> select id, f1 from secu.in_sec_lob minus select id, f1 from jake.in_sec_lob;

no rows selected

SQL> set serveroutput on size 10000;
SQL> declare
  2    match  boolean;
  3    v_id1  number;
  4    v_id2  number;
  5    lob1   clob;
  6    lob2   clob;
  7    cursor c1 is select id, fsc from secu.in_sec_lob order by id;
  8    cursor c2 is select id, fsc from jake.in_sec_lob order by id;
  9  begin
 10    match := true;
 11    open c1;
 12    open c2;
 13    loop
 14      fetch c1 into v_id1, lob1;
 15      exit when c1%NOTFOUND;
 16      fetch c2 into v_id2, lob2;
 17      if dbms_lob.compare(lob1, lob2) <> 0 then
 18        match := false;
 19        dbms_output.put_line('row id='||v_id1||' lob column not matach.');
```

```
 20      end if;
 21    end loop;
 22    close c1;
 23    close c2;
 24    if match then
 25      dbms_output.put_line('');
 26      dbms_output.put_line('all lob column data matched.');
 27    end if;
 28  end;
 29  /
all lob column data matched.

PL/SQL procedure successfully completed.
```

LOB Head Block

LOB 头块的测试，我们知道当 LOB 数据大到一定程度，在 LOB locator 的行内就不再存放 chunk 地址，而是存放一个管理 LOB chunk 地址的头块地址。我们要让 LOB 数据多大才能出现头块呢？在 securefile LOB 存储时，chunk 的大小不再固定，计算有些困难，但我们可以取它的最大值，也就是我们要计算出一个数据量，当达到这个数据大小时肯定会出现 LOB 头块。下面我们来计算这个大小，我们知道 LOB chunk 中的块必须是连续的，在 oracle 中，只有 extent 中的块能保证连续性，所以一个 LOB chunk 包含的块个数最多也就是 extent 中块的个数，我们在上面创建表空间时，指定了 extent 的大小是 32768，并且每个 extent 的大小都是一样的，表空间的块大小是 2K，那么一个 extent 中块的个数=32768/2048=16 个，除去一个管理块，所以一个 LOB chunk 最多有 15 个块。由于行内存储的限制，一个行内最多能存储 6 个 chunk 地址，超过 6 个就会变成存储头块地址，那么我们能得到这个 LOB 数据的值=6*15*2048=184320 字节，由于一个块还有管理用的空间，每个 chunk 也不可能都用去 15 个块的空间，所以如果插入 184320 字节的数据，肯定就会出现 LOB 头块了，如果是 CLOB，双字节存储，那么插入 184320/2=92160 个字符可以了。我们通过下面的实验验证一下。

准备数据

```
declare
  v_data  varchar2(32767);
  v_clob  clob;
begin
```

```
  for i in 1 .. 20 loop
    insert into in_sec_lob values (i, rpad(i, 40, 'a'), empty_clob)
      returning fsc into v_clob;
    dbms_lob.open(v_clob, DBMS_LOB.LOB_READWRITE);
    for j in 1 .. 8 loop
      v_data := rpad(i, 11520, 'l');
      dbms_lob.writeappend(v_clob, 11520, v_data);
    end loop;
    dbms_lob.close(v_clob);
  end loop;
  commit;
end;
/
```

dump 出一行数据看一下，是否行内存储的是头块。

```
SQL> select
  2    dbms_rowid.rowid_relative_fno(rowid) rfn,
  3    dbms_rowid.rowid_block_number(rowid) blk#
  4  from secu.in_sec_lob
  5  where id=1;

       RFN       BLK#
---------- ----------
        12        521

SQL> conn / as sysdba
Connected.
SQL> alter system checkpoint;

System altered.

SQL> alter system dump datafile 12 block 521;

System altered.

tab 0, row 0, @0x70b
tl: 141 fb: --H-FL-- lb: 0x1  cc: 3
col  0: [ 2]  c1 02
col  1: [40]
 31 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61
 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61
col  2: [93]
 00 54 00 01 02 0c 80 80 00 02 00 00 00 01 00 00 00 0c be e5 00 49 40 90 00
 43 22 00 02 d0 00 02 09 01 03 00 02 1f 01 01 03 00 02 51 0f 01 03 00 02 61
```

```
 0f 01 03 00 02 35 0b 01 03 00 02 4d 03 01 03 00 02 7d 03 01 03 00 02 81 0f
 01 03 00 02 71 0c 01 03 00 02 91 0f 01 03 00 02 a1 07
LOB
Locator:
  Length:         84(93)
  Version:         1
  Byte Length:     2
  LobID: 00.00.00.01.00.00.00.0c.be.e5
  Flags[ 0x02 0x0c 0x80 0x80 ]:
    Type: CLOB
    Storage: SecureFile
    Characterset Format: IMPLICIT
    Partitioned Table: No
    Options: VaringWidthReadWrite
  SecureFile Header:
    Length:   73
    Old Flag: 0x40 [ SecureFile ]
    Flag 0:   0x90 [ INODE Valid ]
    Layers:
      Lengths Array: INODE:67
      INODE:
        22 00 02 d0 00 02 09 01 03 00 02 1f 01 01 03 00 02 51 0f 01
        03 00 02 61 0f 01 03 00 02 35 0b 01 03 00 02 4d 03 01 03 00
        02 7d 03 01 03 00 02 81 0f 01 03 00 02 71 0c 01 03 00 02 91
        0f 01 03 00 02 a1 07
```

从上面看到，行内存储的还是 chunk 地址，不是头块地址，为什么呢？原来我们这个表创建时，LOB 是 Enable store in row 的，这样这个字段一共能存储 4000 字节，去掉 Lob locator，Secure File Header 和 Inode 还能存储三千多字节，Oracle 在 SecureFile LOB 存储时改变了策略，剩下的这三千多字节都可以用来存储 chunk 地址，这样在一定的数据量范围内就不用通过头块索引了，减小了 I/O，真是个聪明的做法。

如果用 in row 的 LOB，为了测试 LOB 头块读取数据，就要插入大量的 LOB 数据才能出现，为了尽量用少的数据创建测试场景，我们也只有改变策略了，用 out row 的 LOB，那么我们上面分析的就生效了。

创建一个 out row 的 LOB 表

```
create table out_sec_lob (id number, f1 varchar2(100), fsc clob)
  lob (fsc) store as securefile (disable storage in row);
```

插入数据

```
declare
```

```
  v_data  varchar2(32767);
  v_clob  clob;
begin
  for i in 1 .. 20 loop
    insert into out_sec_lob values (i, rpad(i, 40, 'a'), empty_clob)
      returning fsc into v_clob;
    dbms_lob.open(v_clob, DBMS_LOB.LOB_READWRITE);
    for j in 1 .. 8 loop
      v_data := rpad(i, 11520, 'l');
      dbms_lob.writeappend(v_clob, 11520, v_data);
    end loop;
    dbms_lob.close(v_clob);
  end loop;
  commit;
end;
/
```

dump 出来检查一下。

```
select
  dbms_rowid.rowid_relative_fno(rowid) rfn,
  dbms_rowid.rowid_block_number(rowid) blk#
from secu.out_sec_lob
where id=1;

SQL> conn / as sysdba
Connected.
SQL> alter system checkpoint;

System altered.

SQL> select
  2    dbms_rowid.rowid_relative_fno(rowid) rfn,
  3    dbms_rowid.rowid_block_number(rowid) blk#
  4  from secu.out_sec_lob
  5  where id=1;

       RFN       BLK#
---------- ----------
        12       2905

SQL> alter system dump datafile 12 block 2905;

System altered.
```

```
tab 0, row 0, @0x744
tl: 84 fb: --H-FL-- lb: 0x1  cc: 3
col  0: [ 2]  c1 02
col  1: [40]
 31 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61
 61 61 61 61 61 61 61 61 61 61 61 61 61 61
col  2: [36]
 00 54 00 01 02 0c 80 80 00 02 00 00 00 01 00 00 00 0c bf 17 00 10 40 90 00
 0a 42 00 02 d0 00 02 03 00 0c 21
LOB
Locator:
  Length:        84(36)
  Version:        1
  Byte Length:    2
  LobID: 00.00.00.01.00.00.00.0c.bf.17
  Flags[ 0x02 0x0c 0x80 0x80 ]:
    Type: CLOB
    Storage: SecureFile
    Characterset Format: IMPLICIT
    Partitioned Table: No
    Options: VaringWidthReadWrite
  SecureFile Header:
    Length:   16
    Old Flag: 0x40 [ SecureFile ]
    Flag 0:   0x90 [ INODE Valid ]
    Layers:
      Lengths Array: INODE:10
      INODE:
        42 00 02 d0 00 02 03 00 0c 21
```

这次我们看到行内存储的是头块了，地址是 0x03000c21。

导出数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini
MYDUL> export dict
MYDUL> load dict
MYDUL> unload table secu.out_sec_lob
```

导入数据

```
[road@CentOS bin]$ cd ../data
[road@CentOS data]$ ls
SECU_OUT_SEC_LOB.dat
```

```
[road@CentOS data]$ ../bin/myimp jake/jake file=SECU_OUT_SEC_LOB.dat touser=jake
```

比较数据一致性

先比较非 LOB 数据

```
SQL> select id, f1 from secu.out_sec_lob minus select id, f1 from jake.out_sec_lob;

no rows selected
```

再比较 LOB 数据

```
set serveroutput on size 10000;

declare
  match  boolean;
  v_id1  number;
  v_id2  number;
  lob1   clob;
  lob2   clob;
  cursor c1 is select id, fsc from secu.out_sec_lob order by id;
  cursor c2 is select id, fsc from jake.out_sec_lob order by id;
begin
  match := true;
  open c1;
  open c2;
  loop
    fetch c1 into v_id1, lob1;
    exit when c1%NOTFOUND;
    fetch c2 into v_id2, lob2;
    if dbms_lob.compare(lob1, lob2) <> 0 then
      match := false;
      dbms_output.put_line('row id='||v_id1||' lob column not matach.');
    end if;
  end loop;
  close c1;
  close c2;
  if match then
    dbms_output.put_line('');
    dbms_output.put_line('all lob column data matched.');
  end if;
end;
/
```

比较结果

```
SQL> declare
  2    match  boolean;
```

```
  3    v_id1  number;
  4    v_id2  number;
  5    lob1   clob;
  6    lob2   clob;
  7    cursor c1 is select id, fsc from secu.out_sec_lob order by id;
  8    cursor c2 is select id, fsc from jake.out_sec_lob order by id;
  9  begin
 10    match := true;
 11    open c1;
 12    open c2;
 13    loop
 14      fetch c1 into v_id1, lob1;
 15      exit when c1%NOTFOUND;
 16      fetch c2 into v_id2, lob2;
 17      if dbms_lob.compare(lob1, lob2) <> 0 then
 18        match := false;
 19        dbms_output.put_line('row id='||v_id1||' lob column not matach.');
 20      end if;
 21    end loop;
 22    close c1;
 23    close c2;
 24    if match then
 25      dbms_output.put_line('');
 26      dbms_output.put_line('all lob column data matched.');
 27    end if;
 28  end;
 29  /
all lob column data matched.

PL/SQL procedure successfully completed.
```

分区表测试

本节测试分区表的导出，分区表有几种，我们只测试最常用的范围（range）分区表，其他种类分区表的存储与范围分区表一样，只是分区的方法不同而已，不影响导出测试。

本节测试中使用用户 andy。

准备数据

创建分区表

```
create table range_part
  (id number, f1 varchar2(1000), f2 varchar2(1000), f3 varchar2(1000))
partition by range (id)
(
  partition r_part1 values less than (1000),
  partition r_part2 values less than (2000),
  partition r_part3 values less than (3000),
  partition r_part4 values less than (maxvalue)
);
```

插入数据

```
begin
  for i in 1 .. 200 loop
    insert into range_part values
      (i, rpad(i, 100, 'a'), rpad(i, 200, 'b'), rpad(i, 300, 'c'));
  end loop;
  commit;

  for i in 1001 .. 1200 loop
    insert into range_part values
      (i, rpad(i, 100, 'a'), rpad(i, 200, 'b'), rpad(i, 300, 'c'));
  end loop;
  commit;

  for i in 2001 .. 2200 loop
    insert into range_part values
      (i, rpad(i, 100, 'a'), rpad(i, 200, 'b'), rpad(i, 300, 'c'));
  end loop;
  commit;

  for i in 3001 .. 3200 loop
    insert into range_part values
      (i, rpad(i, 100, 'a'), rpad(i, 200, 'b'), rpad(i, 300, 'c'));
  end loop;
  commit;
end;
/

SQL> conn / as sysdba
Connected.
SQL> alter system checkpoint;

System altered.
```

导出数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini
MYDUL> export dict
MYDUL> load dict
MYDUL> unload table andy.range_part
```

导入数据

```
[road@CentOS bin]$ cd ../data
[road@CentOS data]$ ls
ANDY_RANGE_PART.dat
[road@CentOS data]$ ../bin/myimp jake/jake file=ANDY_RANGE_PART.dat touser=jake
```

比较数据

```
SQL> select * from andy.range_part minus select * from jake.range_part;

no rows selected
```

子分区表测试

准备数据

创建子分区表

```
create table comp_part
  (lid number, rid number, f1 varchar2(1000), f2 varchar2(1000), f3 varchar2(1000))
partition by list (lid)
subpartition by range (rid)
(
  partition l_part1 values (100)
  (
    subpartition r_subpart11 values less than (1000),
    subpartition r_subpart12 values less than (2000),
    subpartition r_subpart13 values less than (maxvalue)
  ),
  partition l_part2 values (200)
  (
```

```
    subpartition r_subpart21 values less than (3000),
    subpartition r_subpart22 values less than (4000),
    subpartition r_subpart23 values less than (maxvalue)
  )
);
```

插入数据

```
begin
  --part 1, subpart 1
  for i in 1 .. 200 loop
    insert into comp_part values
      (100, i, rpad(i, 100, 'a'), rpad(i, 200, 'b'), rpad(i, 300, 'c'));
  end loop;
  commit;

  -- part 1, subpart 2
  for i in 1001 .. 1200 loop
    insert into comp_part values
      (100, i, rpad(i, 100, 'a'), rpad(i, 200, 'b'), rpad(i, 300, 'c'));
  end loop;
  commit;

  -- part 1, subpart 3
  for i in 2001 .. 2200 loop
    insert into comp_part values
      (100, i, rpad(i, 100, 'a'), rpad(i, 200, 'b'), rpad(i, 300, 'c'));
  end loop;
  commit;

  --part 2, subpart 1
  for i in 1 .. 200 loop
    insert into comp_part values
      (200, i, rpad(i, 100, 'a'), rpad(i, 200, 'b'), rpad(i, 300, 'c'));
  end loop;
  commit;

  -- part 2, subpart 2
  for i in 3001 .. 3200 loop
    insert into comp_part values
      (200, i, rpad(i, 100, 'a'), rpad(i, 200, 'b'), rpad(i, 300, 'c'));
  end loop;
  commit;

  -- part 2, subpart 3
```

```
  for i in 4001 .. 4200 loop
    insert into comp_part values
      (200, i, rpad(i, 100, 'a'), rpad(i, 200, 'b'), rpad(i, 300, 'c'));
  end loop;
  commit;
end;
/

SQL> conn / as sysdba
Connected.
SQL> alter system checkpoint;

System altered.
```

导出数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini
MYDUL> export dict
MYDUL> load dict
MYDUL> unload table andy.comp_part
```

导入数据

```
[road@CentOS data]$ ls
ANDY_COMP_PART.dat
[road@CentOS data]$ ../bin/myimp jake/jake file=ANDY_COMP_PART.dat touser=jake
```

比较数据

```
SQL> select * from andy.comp_part minus select * from jake.comp_part;

no rows selected
```

国际字符集测试

准备数据

```
create table tab_in
  (id number, f1 char(40), f2 nchar(40), f3 varchar2(100), f4 nvarchar2(100));


insert into tab_in values
  (1, '中文测试 1', '中文测试 2', '中文测试 3', '中文测试 4');
commit;
```

导出数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini
MYDUL> export dict
MYDUL> load dict
MYDUL> unload table andy.tab_in
```

导入数据

```
[road@CentOS data]$ ls
ANDY_TAB_IN.dat
[road@CentOS  data]$  ../bin/myimp  jake/jake  file=ANDY_TAB_IN.dat  touser=jake
ignore=y
```

比较数据

```
select id, trim(f1), trim(f2), f3, f4 from andy.tab_in
minus
select id, trim(f1), trim(f2), f3, f4 from jake.tab_in;

SQL> select id, trim(f1), trim(f2), f3, f4 from andy.tab_in
  2  minus
  3  select id, trim(f1), trim(f2), f3, f4 from jake.tab_in;

no rows selected
```

```
SQL> select id, trim(f1), trim(f2), f3, f4 from jake.tab_in;

        ID TRIM(F1)
---------- ------------------------------------
TRIM(F2)
--------------------------------------------------------------------------------
F3
--------------------------------------------------------------------------------
F4
--------------------------------------------------------------------------------
         1 中文测试 1
中文测试 2
中文测试 3
中文测试 4
```

测试 CLOB 和 NCLOB 的中文

```
create table tab_in_lob (id number, f1 clob, f2 nclob);

insert into tab_in_lob values (1,'这是一条中文字符的 LOB 数据','这是一条 national
LOB 的中文数据');
commit;
```

导出数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini
MYDUL> export dict
MYDUL> load dict
MYDUL> unload table andy.tab_in_lob
```

导入数据

```
[road@CentOS data]$ ls
ANDY_TAB_IN_LOB.dat
[road@CentOS data]$ ../bin/myimp jake/jake file=ANDY_TAB_IN_LOB.dat touser=jake
ignore=y
```

比较数据

不能用 minus 直接比较数据，我们查询出来看看，人工比对一下吧。

```
SQL> select * from andy.tab_in_lob;

        ID
----------
F1
--------------------------------------------------------------------------------
```

```
F2
--------------------------------------------------------------------------------
         1
这是一条中文字符的 LOB 数据
这是一条 national LOB 的中文数据

SQL> select * from jake.tab_in_lob;

        ID
----------
F1
--------------------------------------------------------------------------------
F2
--------------------------------------------------------------------------------
         1
这是一条中文字符的 LOB 数据
这是一条 national LOB 的中文数据
```

可见，数据时一致的。

ASSM Extent Map 测试

本节测试是为了在表的数据量比较大时，Segment Header 块的 extent map 用完后，要使用下一个 extent map 来存储 extent 的地址。本节测试要验证 extent map 的读取没有问题。

我们使用 ts_small_block 表空间，rock 用户来做测试，这个表空间使用 2K 的块大小，一个 extent 包含 5 个块，所以不用很大的数据量，就能出现下一个 extent map。

准备数据

创建表

```
create table tab_extmap (id number, f1 varchar2(2000));
```

插入数据

我们在每行数据的 f1 字段插入 1600 字符，这样就让每一行数据占用一个块，我们来粗略计算一下需要插入多少行数据，才能出现下一个 extent map。一个块 2048 字节，一半的空间用于辅助 map，每个 extent rdba 占用 4 字节，那么一个块最多能存储的 extent 个数是 2048/2/4=256，一个 extent 包含 5 个块，那么一共能管理 256*5=1280 个块，也就是说

插入 1280 行数据就能出现 extent map，我们来试一下。

```
begin
  for i in 1 .. 1280 loop
    insert into tab_extmap values (i, rpad(i, 1600, 'a'));
  end loop;
  commit;
end;
/

SQL> conn / as sysdba
Connected.
SQL> alter system checkpoint;

System altered.
```

我们把段头的块 dump 出来看看。

```
SQL> select header_file, header_block from dba_segments where owner='ROCK' and
segment_name='TAB_EXTMAP';

HEADER_FILE HEADER_BLOCK
----------- ------------
          8           34

SQL> alter system dump datafile 8 block 34;

System altered.

Block dump from disk:
buffer tsn: 9 rdba: 0x02000022 (8/34)
scn: 0x0000.0040de88 seq: 0x03 flg: 0x04 tail: 0xde882303
frmt: 0x02 chkval: 0x709a type: 0x23=PAGETABLE SEGMENT HEADER
Hex dump of block: st=0, typ_found=1
  Extent Control Header
  -----------------------------------------------------------------
  Extent Header:: spare1: 0      spare2: 0      #extents: 267    #blocks: 1335
                  last map  0x0200038b  #maps: 2      offset: 668
      Highwater::  0x02000557  ext#: 266    blk#: 5      ext size: 5
  #blocks in seg. hdr's freelists: 0
  #blocks below: 1283
  mapblk  0x0200038b  offset: 91
                   Unlocked
  -----------------------------------------------------------
  Low HighWater Mark :
```

```
    Highwater::  0x02000557  ext#: 266    blk#: 5      ext size: 5
  #blocks in seg. hdr's freelists: 0
  #blocks below: 1283
  mapblk  0x0200038b  offset: 91
  Level 1 BMB for High HWM block: 0x0200052a
  Level 1 BMB for Low HWM block: 0x0200052a
  ------------------------------------------------
  Segment Type: 1 nl2: 1      blksz: 2048   fbsz: 0
  L2 Array start offset:  0x00000434
  First Level 3 BMB:  0x00000000
  L2 Hint for inserts:  0x02000021
  Last Level 1 BMB:  0x0200052a
  Last Level II BMB:  0x02000021
  Last Level III BMB:  0x00000000
    Map Header:: next  0x02000120  #extents: 51   obj#: 76009  flag: 0x10000000
  Inc # 0
```

从上面看到下一个 extent map 的地址是 0x02000120，转换后就是相对文件号 8，块号 288 的地址，再 dump 出来看一下。

```
Block dump from disk:
buffer tsn: 9 rdba: 0x02000120 (8/288)
scn: 0x0000.0040ddce seq: 0x01 flg: 0x04 tail: 0xddce2401
frmt: 0x02 chkval: 0x7afd type: 0x24=PAGETABLE EXTENT MAP BLOCK
Hex dump of block: st=0, typ_found=1
EMB Dump:
    Map Header:: next  0x0200038b  #extents: 124  obj#: 76009  flag: 0x10000000
Inc # 0
```

这里看到还有下一个 extent map，原来我们在计算时犯了一个小错误，一个 extent map 不止有 rdba，还有一个 4 字节的长度，那么一个块中能存储的最大 extent 个数就是 2048/2/8=128 个，实际我们看到是#extents: 124，相去不远。

那么我们就拿这批数据来测试吧。

导出数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini
MYDUL> export dict
MYDUL> load dict
MYDUL> unload table rock.tab_extmap
```

导入数据

```
[road@CentOS data]$ ls
ANDY_TAB_IN_LOB.dat
[road@CentOS data]$ ls
ROCK_TAB_EXTMAP.dat
[road@CentOS data]$ ../bin/myimp jake/jake file=ROCK_TAB_EXTMAP.dat touser=jake
```

比较数据

```
SQL> select * from rock.tab_extmap minus select * from jake.tab_extmap;

no rows selected
```

MSSM Extent Map 测试

MSSM 的段头块，我们在导出数据字典的时候就已经遇到了，为了向下兼容，Oracle 数据字典的表空间还是手动管理的。我们在这里主要是测试 extent map 的导出。

准备环境

创建表空间

```
create tablespace ts_mssm
  datafile 'ts_mssm_01.dbf' size 100M autoextend on
  blocksize 2048
  extent management local uniform size 10K
  segment space management manual;
```

创建用户

```
create user uman identified by uman default tablespace ts_mssm;
```

为用户授权

```
grant connect, resource to uman;
```

准备数据

创建测试表

```
create table man_extmap (id number, f1 varchar2(2000));
```

插入数据

```
begin
  for i in 1 .. 1280 loop
    insert into man_extmap values (i, rpad(i, 1600, 'a'));
  end loop;
  commit;
end;
/

SQL> conn / as sysdba;
Connected.
SQL> alter system checkpoint;

System altered.
```

检查是否出现了下一个 extent map

```
SQL> select header_file, header_block from dba_segments
  2    where owner='UMAN' and segment_name='MAN_EXTMAP';

HEADER_FILE HEADER_BLOCK
----------- ------------
         10           32

SQL> alter system dump datafile 10 block 32;

System altered.
```

看一下 dump 出来的内容。

```
Block dump from disk:
buffer tsn: 11 rdba: 0x02800020 (10/32)
scn: 0x0000.0040e813 seq: 0x03 flg: 0x04 tail: 0xe8131003
frmt: 0x02 chkval: 0x1f74 type: 0x10=DATA SEGMENT HEADER - UNLIMITED
Hex dump of block: st=0, typ_found=1
  Extent Control Header
  -----------------------------------------------------------------
  Extent Header:: spare1: 0      spare2: 0      #extents: 257    #blocks: 1283
                  last map  0x0280027d  #maps: 1      offset: 1056
      Highwater::  0x02800525  ext#: 256    blk#: 5      ext size: 5
  #blocks in seg. hdr's freelists: 4
  #blocks below: 1283
  mapblk  0x0280027d  offset: 135
                   Unlocked
     Map Header:: next  0x0280027d  #extents: 121  obj#: 76011  flag: 0x40000000
```

从上面看到，已经出现了下一个 extent map，用这批数据测试。

导出数据

```
[road@CentOS bin]$ ./mydul config=../etc/config.ini
MYDUL> export dict
MYDUL> load dict
MYDUL> unload table uman.man_extmap
```

导入数据

```
[road@CentOS data]$ ls
UMAN_MAN_EXTMAP.dat
[road@CentOS data]$ ../bin/myimp jake/jake file=UMAN_MAN_EXTMAP.dat touser=jake
```

比较数据

```
SQL> select * from uman.man_extmap minus select * from jake.man_extmap;

no rows selected
```