Oracle PL/SQL高级编程

编者语：

作者对Oracle开发管理有多年的经验，并在Oracle 数据库的基础上开发了自己的交易控制中间件，适用于金融、电信、交通等多个行业，现就主要开发资料参考资料共享给大家。

祝大家在Oracle平台上更上一层楼，共同进步。

2006-2-14

第一章 集合

1.1 索引表

索引表是将数据保存在内存中！！！

1.1.1 定义索引表

```
-- 定义记录集
TYPE yang_rec IS RECORD( ename varchar2(30),  eid NUMBER );

-- 定义索引表类型
TYPE yang_tab IS TABLE OF yang_rec INDEX BY BINARY_INTEGER;

-- 定义索引表对象的实例
test_tab yang_tab;
```

1.1.2 将条目插入到索引表中

索引表中的每个元素都由一个唯一的整型值(索引)标识。引用表中的值时，必须提供该值的索引。
索引的范围：1 ---- 2147483647，索引值可以不连续，同时PL/SQL并不为没有使用的条目预留空间。
例如：

```
DECLARE
   CURSOR all_emps IS SELECT * FROM employee ORDER by emp_id;
   TYPE emp_table IS TABLE OF employee%ROWTYPE INDEX BY BINARY_INTEGER;
   emps emp_table;
   emps_max BINARY_INTEGER;
BEGIN
   emps_max := 0;
   FOR emp IN all_emps LOOP
      emps_max := emps_max + 1;
      emps(emps_max).emp_id := emp.emp_id;
      emps(emps_max).emp_name := emp.emp_name;
   END LOOP;
END;
/
```

1.1.3 对索引表中进行操作

1) 插入：见上例。
2) 引用：

```
   IF emps.EXIST(10) THEN
      DBMS_OUTPUT.PUT_LINE('存在第10条记录。');
   END IF;
```

3) 修改：

```
   修改emps 表中的第100个条目：
   emps(100).emp_name := 'yang linker';
```

4) 删除：

```
   -- 删除emps 表中的第100个条目：
   emps.DELETE(100);
```

```
-- 删除emps 表中的从1到100的条目：
emps.DELETE(1, 100);

-- 删除emps 表中的的所有条目：
emps.DELETE;
```

1.1.4 索引表中的函数

1) count：返回表的条目数量：
```
num_rows := emps.COUNT;
```
2) EXISTS：如果指定的条目存在，则返回为真；否则为假。
```
IF emps.EXIST(10) THEN
    DBMS_OUTPUT.PUT_LINE('存在第10条记录。');
END IF;
```
3) LIMIT：该方法返回集合可以包含的最大元素数目。只有变长数组才有上限。将LIMIT 用于嵌套表和索引表时，其返回为NULL。
4) FRIST：该方法返回集合中使用的最小的索引值。
5) LAST：该方法返回集合中使用的最大的索引值。
6) NEXT：该方法返回集合中当前使用的下一个索引值。
7) PRIOR：该方法返回集合中当前使用的上一个索引值。
8) DELETE：删除集合中的条目，见前例。
9) TRIM：从集合的尾部删除一个或多个条目，无返回值，只适用于变长数组和嵌套表。
```
emps.TRIM(1);-- 从集合的尾部删除一个条目
emps.TRIM(3);-- 从集合的尾部删除三个条目
```
10) EXTEND：在集合的尾部添加条目或复制已有的条目，只适用于变长数组和嵌套表。
```
emps.EXTEND(1);  -- 从集合的尾部添加一个条目
emps.EXTEND(3);  -- 从集合的尾部添加三个条目
emps.EXTEND(1, 3);-- 复制集合的第三个条目，并将其添加到表的末尾。
```

1.2 嵌套表

将数据保存在内存中！！！
嵌套表是一个无序记录集合。
检索数据库中的嵌套表时，条目的索引是连续的，不能象索引表那样随意跳过索引值。
需要使用构造函数初始化嵌套表。

嵌套表不能是以下数据类型：
BOOLEAN, NCHAR, NCLOB, NVARCHAR2, REF CURSOR, TABLE, VARRAY, NOT NULL。

嵌套表的定义和索引表类似，但不能使用INDEX BY 子句。

1.2.1 初始化嵌套表

必须使用构造函数初始化嵌套表后，才能给它添加条目！

```
-- 定义索引表类型
TYPE emp_tab IS TABLE OF emp%ROWTYPE;

-- 定义索引表对象的实例
```

```
emps emp_tab;

-- 初始化嵌套表
emps := emp_tab();
```

1.2.2 扩展嵌套表

利用 EXTEND 来扩展嵌套表的数据于内存。

```
DECLARE
   CURSOR all_emps IS SELECT * FROM emp ORDER BY empno;
   TYPE emp_table IS TABLE OF emp%ROWTYPE;
   emps      emp_table;
   I      PLS_INTEGER;
   l_count PLS_INTEGER;
BEGIN
   l_count := 0;
   emps := emp_table();       -- 初始化嵌套表并产生一条空记录
   FOR c1 IN all_emps LOOP
      l_count := l_count + 1;
      emps.EXTEND;
      emps(l_count).empno := c1.empno;
      emps(l_count).ename := c1.ename;
      emps(l_count).job := c1.job;
      emps(l_count).mgr := c1.mgr;
      emps(l_count).hiredate := c1.hiredate;
      emps(l_count).sal := c1.sal;
      emps(l_count).comm := c1.comm;
      emps(l_count).deptno := c1.deptno;
   END LOOP;

   -- clone the first entry five times
   emps.EXTEND(5,1);

   FOR i IN 1..l_count+5 LOOP
      DBMS_OUTPUT.PUT_LINE(TO_CHAR(emps(i).empno) || '     ' || mps(i).ename);
   END LOOP;
END;
/
```

1.2.3 删除嵌套表中的条目

1) DELETE 方法：

emps.DELETE(10); -- 删除嵌套表中的第10个条目。

注意：在删除嵌套表中的条目后，嵌套表中的条目并没有重新编号，还可以继续使用。

2) TRIM 方法：

TRIM方法是在表的末尾删除指定数目的条目。

TRIM方法只能用于嵌套表和变长数组。

```
DECLARE
   CURSOR all_emps IS SELECT * FROM emp ORDER BY empno;
   TYPE emp_table IS TABLE OF emp%ROWTYPE;
   emps emp_table;
   i    PLS_INTEGER;
   l_count  PLS_INTEGER;
BEGIN
```

```
    l_count := 0;

    -- 初始化嵌套表并产生一条空记录
    emps := emp_table();
    FOR c1 IN all_emps LOOP
       l_count := l_count + 1;
       emps.EXTEND;
       emps(l_count).empno := c1.empno;
       emps(l_count).ename := c1.ename;
       emps(l_count).job := c1.job;
       emps(l_count).mgr := c1.mgr;
       emps(l_count).hiredate := c1.hiredate;
       emps(l_count).sal := c1.sal;
       emps(l_count).comm := c1.comm;
       emps(l_count).deptno := c1.deptno;
    END LOOP;

    -- clone the first entry five times
    emps.EXTEND(5,1);

    -- Trim off the five clones of entry #1
    emps.TRIM(5);

    -- Delete the first entry
    emps.DELETE(1);

    FOR i IN 1..l_count+5 LOOP
       DBMS_OUTPUT.PUT_LINE(TO_CHAR(emps(i).empno) || '  ' || emps(i).ename);
    END LOOP;
END;
/
```

注意：调试以上代码，并注意错误表达!!!

1.3 变长数组

变长数组与嵌套表类似，但变长数组的最大长度是固定的。
变长数组与嵌套表一样需要初始化。

1.3.1 定义变长数组

```
-- 定义最大长度为100 的变长数组
TYPE type_name IS VARRAY(100) OF VARCHAR2(20);
```

1.3.2 扩展变长数组

类似于嵌套表，但不能超过最大长度。例程类似于嵌套表的例程。

1.4 批绑定

PL/SQL 批绑定是Oracle8i中的新功能。
1) 使用 BULK COLLECT
2) 使用 FORALL

例程：

```
DECLARE
   CURSOR c1 IS SELECT empno, ename FROM emp;
   TYPE eno_tab IS TABLE OF emp.empno%TYPE;
   TYPE ename_tab IS TABLE OF emp.ename%TYPE;
   l_enos    eno_tab;
   l_enames  ename_tab;
BEGIN
   OPEN c1;
   FETCH c1 BULK COLLECT INTO l_enos, l_enames;
   CLOSE C1;

   FOR i IN 1..l_enos.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE(TO_CHAR(l_enos(i)) || '    ' || l_enames(i));
   END LOOP;

   FORALL i IN l_enos.FIRST .. l_enos.LAST
      UPDATE emp SET ename = l_enames(i) WHERE empno = l_enos(i);
END;
/
```

1.5 集合的异常处理

异 常	原 因
COLLECTION_IS_NULL	在构造函数初始化集合之前试图使用它
NO_DATA_FOUND	试图访问集合中不存在的条目
SUBCRIPT_BEYOND_COUNT	使用的下标超过了集合当前的元素数目
SUBCRIPT_OUTSIDE_LIMIT	变长数组中使用的下标大于该变长数组声明中规定的最大值
VALUE_ERROR	使用一个不能转换为整形的下标

第二章 触发器

2.1 触发器的创建

```
CREATE TRIGGER [schema.]trigger_name
   {BEFORE|AFTER} {UPDATE|INSERT|DELETE} ON [schema.]table_name
   [ [REFERENCING correlation_names] FOR EACH ROW [WHEN (condition)] ]
DECLARE
   declaration
BEGIN
   pl/sql code
END;
/
```

2.2 触发器的管理

1) 查看触发器：

```
SQL> desc ALL_TRIGGERS;
   名称          空?    类型
   ------------------- -------- ----------
```

```
owner                   VARCHAR2(30)
trigger_name            VARCHAR2(30)
trigger_type            VARCHAR2(16)
triggering_event        VARCHAR2(216)
table_owner             VARCHAR2(30)
base_object_type        VARCHAR2(16)
table_name              VARCHAR2(30)
column_name             VARCHAR2(4000)
referencing_names       VARCHAR2(128)
when_clause             VARCHAR2(4000)
status                  VARCHAR2(8)
description             VARCHAR2(4000)
action_type             VARCHAR2(11)
trigger_body            LONG
```

2）查看触发器的代码：

触发器的源代码被存贮在 ALL_TRIGGERS 的 trigger_body 字段中。抽取触发器定义的命令：

```
SET ECHO OFF
SET MAXDATA 20000
SET LONG 20000
SET LONGCHUNKSIZE 1000
SET PAGESIZE 0
SET HEADING OFF
SET TRIMSPOOL ON
SET TRIMOUT ON
SET RECSEP OFF

ACCEPT trigger_name    CHAR PROMPT 'please input the trigger to lookup:'
ACCEPT trigger_owner   CHAR PROMPT 'please input the trigger owner:'
ACCEPT file_name       CHAR PROMPT 'please enter the output file path:'

SET TERMOUT OFF
SET FEEDBACK OFF

-- 设置when_clause字段格式并使之折行
COLUMN when_clause FORMAT A60 WORD_WRAPPED

SPOOL &file_name
SELECT 'CREATE OR REPLACE TRIGGER ' || description FROM all_triggers
   WHERE trigger_name = UPPER('&trigger_name')
   AND owner = UPPER('&trigger_owner');

SELECT 'WHEN (' || when_clause || ')' when_caluse FROM all_triggers
   WHERE trigger_name = UPPER('&trigger_name')
   AND owner = UPPER('&trigger_owner')
   AND when_clause IS NOT NULL;

SELECT trigger_body FROM all_triggers
   WHERE trigger_name = UPPER('&trigger_name')
   AND owner = UPPER('&trigger_owner');

SELECT '/' FROM DUAL;

SPOOL OFF
SET TERMOUT ON
SET FEEDBACK ON
```

```
SET VERIFY ON
SET HEADING ON
SET PAGESIZE 24
```

3）打开和关闭触发器：

```
-- 打开触发器
ALTER TRIGGER trigger_name DISABLE;

-- 关闭触发器
ALTER TRIGGER trigger_name ENABLE;
```

2.3 触发器的新功能

创建作用于特定数据库与数据定义事件的触发器。事件包括：

- 关闭数据库(startup)；
- 启动数据库(shutdown)；
- 登陆数据库(logon)；
- 退出数据库(logoff)；
- 发生服务器错误(server error)；
- 创建对象(create命令)；
- 修改对象(alter命令)；
- 删除对象(delete命令)；

1）数据库事件触发器：

数据库事件触发器可以定义为数据库级和模式级。

要创建数据库级的触发器，必须有administrator database trigger 的系统权限，该权限提供给DBA角色，通常只有数据库管理员有此权限。

事　件	触发器关键词	模 式 级	数据库级
启动数据库	STARTUP	否	是
关闭数据库	SHUTDOWN	否	是
服务器错误	SERVERERROR	否	是
登陆数据库	LOGON	是	是
退出数据库	LOGOFF	是	是
创建对象	CREATE	是	否
修改对象	ALTER	是	否
删除对象	DELETE	是	否

2）事件属性：

编写触发器时，可以象变量一样使用这些属性。

事件属性	描　述	适应的触发器类型
sys.sysevent	返回一个20个字符的字符串，描述导致触发器激发的事件	所有类型
sys.intance_num	返回Oracle 实例编号	所有类型
sys.database_name	返回数据库品牌，通常为“Oracle”	所有类型
sys.server_error (stack_position)	从错误堆栈的指定位置返回错误号，使用sys.server_error(1) 查找最近的错误	SERVERERROR
sys.login_user	返回导致触发器激发的用户名	所有类型
sys.dictionary_obj_type	返回当DDL触发器激发时涉及到的对象类型	CREATE、ALTER、DROP
sys.dictionary_obj_name	返回当DDL触发器激发时涉及到的对象名称	CREATE、ALTER、DROP

sys.des_encrypted_password	创建或修改用户时，返回加密后的该用户的密码	CREATE、ALTER、DROP

3）创建数据库事件触发器实例：

```
CREATE TABLE uptime_log(
   database_name       VARCHAR2(30),
   event_name          VARCHAR2(30),
   event_time          DATE,
   triggered_by_user   VARCHAR2(30));


-- 创建启动数据库时的事件触发器。
CREATE OR REPLACE TRIGGER log_startup
AFTER STARTUP ON DATABASE
BEGIN
   INSERT INTO uptime_log VALUES(sys.database_name,
        sys.sysevent, sysdate, sys.login_user);
END;
/

-- 创建关闭数据库时的事件触发器。
CREATE OR REPLACE TRIGGER log_shutdown
AFTER SHUTDOWN ON DATABASE
BEGIN
   INSERT INTO uptime_log VALUES(sys.database_name,
        sys.sysevent, sysdate, sys.login_user);
END;
/
```

4）创建DDL事件触发器实例：

```
CREATE TABLE alter_audit_trail(
   object_owner     VARCHAR2(30),
   object_name      VARCHAR2(30),
   object_type      VARCHAR2(30),
   alter_by_user    VARCHAR2(30),
   sys_event        VARCHAR2(30),
   alteration_time  DATE);

-- 当SCOTT用户创建、修改或删除时的模式触发器。
CREATE OR REPLACE TRIGGER tri_alter_audit_trail
AFTER ALTER OR CREATE OR DROP ON scott.SCHEMA
BEGIN
   IF sys.sysevent = 'ALTER' THEN
      INSERT INTO alter_audit_trail VALUES
         (sys.dictionary_obj_owner, sys.dictionary_obj_name,
          sys.dictionary_obj_type, sys.login_user, 'my_alter', sysdate);
   ELSIF sys.sysevent = 'CREATE' THEN
      NULL;
   ELSIF sys.sysevent = 'DROP' THEN
      NULL;
   END IF;
END;
/
```

2.4 替代触发器

语法同建立表触发器，使用关键词INTEAD_OF代替BEFORE或AFTER。

替代触发器解决了不能修改视图的问题，它只能在视图上创建，执行一个PL/SQL代码块，而不是一条DML语句(如INSERT、UPDATE)。

编写替代触发器时，主要是编写更新视图底层的表的代码。同时也可以针对视图中的某个字段或所有字段。语法：

```
CREATE OR REPLACE TRIGGER tri_emp_view
INSTEAD OF UPDATE ON emp_view
BEGIN
   PL/SQL code;
END;
```

2.5 触发器的局限性

1. 触发器无法完成的工作：
   1) 查询或修改变化的表；
   2) 执行数据定义语言(CREATE、DROP、ALTER)的语句；
   3) 执行COMMIT、ROLLBACK、SAVEPOINT 语句；
   注：变化的表是指执行触发器时正在被修改的表。
   触发器中不能执行COMMIT、ROLLBACK、SAVEPOINT 语句，以确保对事务控制的有效性及数据的完整性。

2. 如果使用触发器来实现商务规则或引用完整性规则，不会影响在触发器创建之前的记录。创建声明性约束实际上是指出数据必须为真，如果出现了违反了约束的数据，则不能建立约束。
3. 不能对特定模式对象(例如：表)来定义DDL触发器。

第三章 对象

3.1 对象的定义

对象可以嵌套，而且嵌套级数不限。

1. 例程：

```
-- 创建对象头
CREATE OR REPLACE TYPE test_obj IS OBJECT(
   m_Name VARCHAR2(30),
   m_id   NUMBER,
   MEMBER PROCEDURE GetName(p_id IN NUMBER, p_name OUT VARCHAR2),
   MEMBER FUNCTION  GetName(p_id NUMBER) RETURN VARCHAR2,
   MEMBER PROCEDURE SetName(p_id IN NUMBER, p_name IN VARCHAR2)
);
/

-- 创建对象包体
CREATE OR REPLACE TYPE BODY test_obj IS
   MEMBER PROCEDURE GetName(p_id IN NUMBER, p_name OUT VARCHAR2) IS
   BEGIN
          SELECT ename INTO p_name FROM emp WHERE empno = p_id;
   END;
```

```
    MEMBER FUNCTION  GetName(p_id IN NUMBER) RETURN VARCHAR2 IS
        l_Name VARCHAR2(30);
        l_sql  VARCHAR2(100);
    BEGIN
        -- 不能使用 m_Name 来代替 l_name
        l_sql := 'SELECT ename FROM emp WHERE empno = :1';
        EXECUTE IMMEDIATE l_sql INTO l_Name USING p_id;
        RETURN l_Name;
    END;

    MEMBER PROCEDURE SetName(p_id IN NUMBER, p_name IN VARCHAR2) IS
        l_sql  VARCHAR2(100);
    BEGIN
        l_sql := 'UPDATE emp SET ename = :1 WHERE empno = :2';
        EXECUTE IMMEDIATE l_sql USING p_name, p_id;
        COMMIT;
    END;
END;
/

-- 删除对象：
DROP TYPE my_object;
```

2．调用上述例程，注意调用方法：

```
DECLARE
    l_name VARCHAR2(30);
    mo1    test_obj;
    mo2    test_obj;
BEGIN
    mo1 := test_obj('test', 10);
    mo2 := mo1;

    mo1.GetName(7369, l_name);
    DBMS_OUTPUT.PUT_LINE(l_name);

    l_name := mo2.GetName(7369);
    DBMS_OUTPUT.PUT_LINE(l_name);

    mo2.SetName(7369, 'yang');
END;
/
```

3.2 对象的存贮和检索

1．存贮和检索对象：

Oracle 的对象关系模型允许对象作为数据库表中的一个字段存贮。例如：

```
CREATE TABLE yang_test(test test_obj);
```

注意：在定义表之后，若想改变对象test_obj的内容，则必须首先删除表。

要求：如何应用该字段？如何调用此字段中包含的函数和过程？

第四章 调试

4.1 编写 DEBUG 程序包(例程)

```
CREATE OR REPLACE PACKAGE yang_debug AS
   PROCEDURE out(p_comments IN VARCHAR2, p_varible IN VARCHAR2);
   PROCEDURE Erase;
END yang_debug;

CREATE OR REPLACE PACKAGE BODY yang_debug AS
   PROCEDURE out(p_comments IN VARCHAR2, p_varible IN VARCHAR2) IS
      l_file UTL_FILE.FILE_TYPE;
   BEGIN
      l_file := UTL_FILE.FOPEN('d:\test', 'debug.log', 'a');
      UTL_FILE.PUT_LINE(l_file, TO_CHAR(sysdate, 'yyyy-mm-dd 24HH:MM:SS ') ||
           ',Comment: ' || p_comments || ', Varible: ' || p_varible);
      UTL_FILE.FCLOSE(l_file);
   EXCEPTION
      WHEN OTHERS THEN
         DBMS_OUTPUT.PUT_LINE('ERROR:' || TO_CHAR(SQLCODE) || SQLERRM);
      NULL;
   END out;

   PROCEDURE Erase IS
      l_file UTL_FILE.FILE_TYPE;
   BEGIN
      l_file := UTL_FILE.FOPEN('d:\test', 'debug.log', 'w');
      UTL_FILE.FCLOSE(l_file);
   EXCEPTION
      WHEN OTHERS THEN
         DBMS_OUTPUT.PUT_LINE('ERROR:' || TO_CHAR(SQLCODE) || SQLERRM);
         NULL;
   END Erase;
END yang_debug;
/
```

4.2 调用函数

```
CREATE OR REPLACE PROCEDURE assert(
   condition IN BOOLEAN,
   message IN VARCHAR2 ) AS
BEGIN
   IF NOT condition THEN
      RAISE_AAPLICATION_ERROR(-20000, message);
   END IF;
END assert;
/
```

第五章 大对象类型

5.1 大对象数据类型

1．大对象数据类型(LOB)可以存贮高达4GB的非结构化数据(如文本、图像、视频剪辑、声音信号等)。

对象	位置	描述
CLOB	内部	字符大型对象，保存多达4GB的单字节字符，其中的字符与数据库当前的字符集对应
NCLOB	内部	国家特有字符大型对象，保存多达4GB的单字节字符或多字节字符，其中的字符符合ORACLE数据库定义的国家特有字符集
BLOB	内部	二进制大型对象，保存多达4GB的原始(未结构化)的数据
BFILE	外部	二进制文件，存贮为操作系统可存取的文件，使ORACLE可以操纵它，这些文件可以位于各种存贮设备上，包括DVD_ROW、CD_ROW和磁盘驱动器

外部指的是一个物理文件，即操作系统可以访问该文件，该文件并不存贮于数据库中，ORACLE使用它来存贮文件名和文件位置；内部对象将一个定位器存贮在表的一个大型对象字段中，定位器指向数据在表中的实际位置。因此，检索LOB对象时，只返回数据定位器，而不返回实际的数据项。

LOB 可以在整个会话期间都存在，或者是临时的。

局限性：

1) LOB 不能用于ORACLE高级表存贮技术，如群集；
2) LOB 不能出现在 GROUP BY、ORDER BY、SELECT DISTINCT、JOIN 语句或群集中。然而LOB可以出现在使用 UNION ALL 的语句中；
3) 禁止分布式LOB；
4) LOB 不允许出现在VARRAY中。

2．大对象数据类型与LONG型的区别：

1) 可以在单条记录中存贮多个LOB，而每条记录只能存贮一个LONG或LONG RAW数据；
2) LOB 能够具有用户定义的数据类型的属性，而LONG或LONG RAW不可以；
3) 只有LOB定位器能存贮在表字段中，BLOB和CLOB数据能够存贮在独立的表空间中，而BFILE 数据作为外部文件存贮。对于LONG和LONG RAW 而言，整个值被存贮在表字段中；如果LOB小于3964字节，则整个LOB被存贮在表字段中。
4) 访问LOB字段时，返回的是定位器；访问LONG或LONG RAW 时，返回的是整个值；
5) LOB 可多达4GB，而BFILE 的最大尺寸取决于操作系统，但不能超过 4GB，其有效取值范围为 1--2**32-1；而LONG 或LONG RAW 不能超过2GB。
6) LOB 以随机、准确的方式操纵数据，其灵活性比 LONG 或 LONG RAW 数据更大；可以以随机的偏移访问LOB，而LONG 必须从头开始访问，再到需要的位置。
7) 在本地和分布式环境中都可以使用LOB；而对LONG 或 LONG RAW 而言，这是不可能的。
8) TO_LOB 函数将已有的LONG 字段转换为LOB；

3．定位器：

定位器被存贮在大型对象字段中，并指向数据存贮的实际位置。理解使用Oracle定位器时，在事务级发生了什么事情非常重要。将LOB从一条记录中拷贝到下一条记录时，将创建一个定位器，并拷贝和贮存来自源记录的所有数据。

如果删除一条记录，而没有将所有的内容拷贝到新记录中，则LOB中的所有数据都将丢失。删除内部LOB时，定位器和LOB的内容都将被删除；删除外部BFILE时，文件依然存在，但是定位器被删除。

将内部LOB添加到表中时，需要创建一个定位器，方法是将数据赋给LOB字段，或者使用EMPTY_BLOB或 EMPLY_CLOB；将外部 BFILE 添加到表中时，应该使用BFILENAME将定位器赋给相应的字段。

使用 LOB 时，应该锁定它，以防止其他用户访问此 LOB。

5.2 在 Oracle8i 数据库中使用外部文件：

1. 创建目录对象：

在能够访问外部文件之前，需要创建目录对象。目录对象将一个名称映射为一个指定的路径。语法如下：

```
CREATE OR REPLACE DIRECTORY dir_name AS src_path;
其中：
dir_name是与路径相关的目录对象的名称；
src_path是文件的物理路径。

SELECT * FROM all_directories; --查看路径信息；
```

2. BFILE 的局限性：

BFILE 数据类型不为 commit 和 rollback 提供事务性支持。同时，文件以只读方式打开，因此不能以任何方式写或修改这些外部文件。

编辑init.ora 文件，并修改 session_max_open_files= 20语句以适应要求。当收到“打开文件过多”的消息后，程序异常结束而不关闭所有已打开的文件。

3. BFILE 的属性：

BFILE数据类型是RAW型。

5.3 DBMS_LOB 包

BFILE 是Oracle 中唯一的外部数据类型，因此Oracle 提供了只能用于外部 LOB的函数：

名称	访问的对象	描述
BFILENAME	BFILE	在PL/SQL块表中创建一个指向文件位置的指针(定位器)
COMPARE	所有的LOB	比较两个文件的全部或部分内容
FILECLOSE	BFILE	关闭与BFILE定位器关联的文件
FILECLOSEALL	BFILE	关闭打开的所有文件
FILEEXISTS	BFILE	检查文件是否位于定位器指定是位置
FILEGETNAME	BFILE	返回目录对象和BFILE的路径
FILEISOPEN	BFILE	检查文件是否已经打开
FILEOPEN	BFILE	真正打开文件
GETLENGTH	所有的LOB	返回LOB 的实际长度
INSTR	所有的LOB	在LOB 中搜索与指定的字符串匹配的模式
READ	所有的LOB	将LOB 中指定的内容读取到缓冲区中
SUBSTR	所有的LOB	返回参数指定的LOB的全部或部分内容

5.3.1 函数说明

1) BFILENAME 函数：

使用外部文件或将BFILE 的定位器插入到表中时，需要调用BFILENAME 函数，该函数创建一个指向外部文件的定位器。BFILENAME 函数的语法如下：

```
FUNCTION BFILENAME( Directory_Object IN VARCAHR2, Filename IN VARCHAR2)
   RETUTN BFILE_Locator;
```

- 参数含义：
  - Directory_Object 是已经创建的对象，它存贮了文件的路径；
  - Filename 是文件的实际名称；

- 返回值：
  返回一个指向文件的定位器。在PL/SQL 块中就可以使用该值，也可以将其插入至包含有BFILE类型字段的表中。

- 注意：
  如果文件被移动或删除时，而Oracle 不会自动更新，此文件定位器仍然指向原来的位置，则当试图打开此文件时，Oracle 会引发一个错误。

2) COMPARE 函数：

COMPARE 函数用于比较LOB 的部分或全部内容。可以检查是否有两个相同的外部文件。

```
FUNCTION COMPARE( Lob1 IN BFILE, Lob2 IN BFILE,
   Number_Bytes_to_Compare IN INTEGER,
   Origin_Lob1 IN INTEGER := 1, Origin_Lob2 IN INTEGER := 1 )
RETURN Compare_Result_Integer;
```

- 参数含义：
  - Lob1：是要比较的第一个LOB；Lob2 是要比较的第二个LOB；
  - Number_Bytes_to_Compare：是要比较的 Lob1 和 Lob2 中的字节总数；
  - Origin_Lob1：是要比较的文件的开始位置，1 表示从头开始比较。

- 返回值：
  - 0: Lob1和Lob2 数据相同；
  - >0: Lob1和Lob2 数据不同；
  - NULL: 其中一个参数无效(如：起始位置不正确、字节越界等)；

3) FILECLOSE 函数：

关闭一个已打开的文件。语法如下：

```
PROCEDURE FILECLOSE(BFILE_Locator);
```

4) FILECLOSEALL 函数：

关闭所有已打开的文件。语法如下：

```
PROCEDURE FILECLOSEALL;
```

5) FILEEXISTS 函数：

检查文件本身是否在目录对象和文件名指定的位置。

```
FUNCTION FILEEXISTS(BFILE_Locator) RETURN Status_Integer;
```

- 参数含义：
  BFILE_Locator 是BFILENAME 函数给文件指定的定位器。

- 返回值：
  - 1: 文件存在于特定的位置；
  - 0: 文件不存在
  - NULL: 发生系统错误、没有访问该文件或路径的权限、定位器的值为NULL等。

6) FILEGETNAME 函数：

得到与定位器相关联的目录对象和文件名。

```
PROCEDURE FILEGETNAME (BFILE_Locator, Directory_Object OUT VARCHAR2,
   Filename OUT VARCHAR2);
```

- 参数含义：
  - BFILE_Locator 是BFILENAME 函数给文件指定的定位器;

- Directory_Object 是CREATE DIRECTORY 命令创建的、与路径关联的目录对象；
- Filename 是与BFILE 定位器关联的文件名。

7) FILEISOPEN 函数：

检查文件是否打开。

```
FUNCTION FILEISOPEN(BFILE_Locator) RETURN Status_Integer;
```

- 参数含义：
  - BFILE_Locator 是BFILENAME 函数给文件指定的定位器。
- 返回值：
  - 1: 文件已经打开；
  - 其他整形值： 文件不存在
  - Exception： 文件或目录不存在、权限不足等。

8) FILEOPEN 函数：

打开文件。

```
PROCEDURE FILEOPEN(BFILE_Locator, DBMS_LOB.FILE_READONLY);
```

- 参数含义：
  - BFILE_Locator 是BFILENAME 函数给文件指定的定位器；
  - DBMS_LOB.FILE_READONLY 是目前唯一可用的打开文件的模式。

9) GETLENGTH 函数：

返回文件的实际长度，以字节我单位。

```
FUNCTION GETLENGTH(BFILE_Locator) RETURN Length_Integer;
```

- 参数含义：
  - BFILE_Locator 是BFILENAME 函数给文件指定的定位器。
- 返回值：
  - >=0: 文件长度；
  - NULL: 定位器为空、文件没有打开、发生操作系统错误、权限不足等。

10) INSTR函数：

从指定的偏移开始，在LOB 中查找第 N 个模式匹配的字符串。

```
FUNCTION INSTR(BFILE_Locator, Patten IN RAW,
   Starting_location IN INTEGER := 1,
   Nth_Occurrence IN INTEGER := 1)
RETURN Status_Integer;
```

- 参数含义：
  - BFILE_Locator 是BFILENAME 函数给文件指定的定位器；
  - RAW 类型的 patten 是要匹配的模式；
  - Starting_location 是要搜索匹配的文件的开始位置；
  - Nth_Occurrence 是要进行的第N次匹配。
- 返回值：
  - 0: 没有找到模式；
  - >0: 找到的模式离头文件的偏移；
  - NULL: 某个参数为空或者无效。

11) READ 函数：

将文件的部分或全部内容读取到计算机的本地内存中。

```
PROCEDURE READ(BFILE_Locator, Read_Amount IN BENARY_INTEGER,
   Starting_Location IN INTEGER, Buffer OUT RAW);
```

- 参数含义：
  - BFILE_Locator 是BFILENAME 函数给文件指定的定位器；
  - Read_Amount 是要从文件中读取到缓冲区中的字节数；
  - Starting_location 是要从文件中读取数据的开始位置；
  - Buffer 是用于存取从文件中读取的数据的位置；

- 注意：
  - 如果任何一个参数为NULL，则会引发VALUE_ERROR异常；
  - 如果任何一个参数无效，则会引发INVALID_AVGVAL异常；
  - 如果达到文件结尾，则将引发NO_DATA_FOUND 异常；
  - 如果文件没有打开，则将引发UNOPEN_FILE异常；

12) SUBSTR 函数：

从文件中抽取指定数目的字节。

```
FUNCTION SUBSTR(BFILE_Locator, Read_Amount IN BINARY_INTEGER,
   Starting_Location IN INTEGER := 1);
RETURN RAW;
```

- 参数含义：
  - BFILE_Locator 是BFILENAME 函数给文件指定的定位器；
  - Read_Amount 是要从文件中抽取的字节数；
  - Starting_location 是指从文件中读取数据的开始位置；

5.3.2 应用举例

```
CREATE ANY DIRECTORY TO scott;
CONN scott/tiger;
DROP DIRECTORY yang_dir;
CREATE OR REPLACE DIRECTORY yang_dir AS 'd:\test';

DECLARE
   v_file          BFILE;
   v_dirname       VARCHAR(30);
   v_location      VARCHAR(2000);
   v_fileisopen    INTEGER;
   v_fileisexists  INTEGER;
BEGIN
   v_file := BFILENAME('YANG_DIR', test01.jpg');  -- 必须为大写
   v_fileisexists := DBMS_LOB.FILEEXISTS(v_file);

   IF v_fileisexists = 1 THEN
      DBMS_OUTPUT.PUT_LINE('The File exists!');
      v_fileisopen := DBMS_LOB.FILEISOPEN(v_file);
      IF v_fileisopen = 1 THEN
         DBMS_OUTPUT.PUT_LINE('The File is open!');
      ELSE
         DBMS_OUTPUT.PUT_LINE('Opening the file!');
         DBMS_LOB.FILEOPEN(v_file);
      END IF;
```

```
            DBMS_LOB.FILEGETNAME(v_file, v_dirname, v_location);
               DBMS_OUTPUT.PUT_LINE('The Diectory Object is: ' || v_dirname ||
                  'The File Name is: ' || v_Location);
               DBMS_LOB.FILECLOSE(v_file);
         ELSE
            DBMS_OUTPUT.PUT_LINE('The File can not be found!');
         END IF;
      END;
      /
```

5.3.3 内部 LOB 的函数和过程

名称	访问的对象	描　述
APPEND	内部BLOB	将一个LOB附加在另一个LOB的后面
COMPARE	内部BLOB	比较两个LOB的部分或全部内容
COPY	内部BLOB	将一条记录中的LOB拷贝到另一条记录中
EMPTY_BLOB	BLOB	在BLOB字段中创建一个定位器
EMPTY_CLOB	CLOB	在CLOB字段中创建一个定位器
ERASE	内部BLOB	删除内部LOB的部分或全部内容
GETLENGTH	所有的LOB	返回LOB的长度
INSTR	所有的LOB	使用指定的字符串在LOB中搜索匹配模式
READ	所有的LOB	将LOB 中指定数目的字节读取到缓冲区中
SUBSTR	所有的LOB	返回参数指定的LOB的全部或部分内容
TRIM	内部 LOB	将一个 LOB 裁剪到指定的长度
WRITE	内部 LOB	将输出写入到LOB中

1) APPEND：

将一个LOB附加在另一个LOB的后面。

```
PROCEDURE APPEND(Dest_Locator IN OUT BLOB, Src_Locator IN BLOB);
PROCEDURE APPEND(Dest_Locator IN OUT CLOB CHARACTER SET Set_Destid,
          Src_Locator IN OUT CLOB CHARACTER SET Dest_Locator%CHARSET);
```

- 参数含义：
  - Dest_Locator 是被原LOB 附加的目标LOB的定位器；
  - Src_Locator 指出表示源LOB；当使用CLOB时，Set_Destid用于指定字符集，如果没有字符集，则使用Oracle服务器本身的字符集。

2) COMPARE：

比较LOB部分或全部内容。

```
FUNCTION COMPARE( Lob1 IN BLOB, Lob2 IN BLOB,
   Number_Bytes_to_Compare IN INTEGER,
   Origin_Lob1 IN INTEGER := 1, Origin_Lob2 IN INTEGER := 1 )
RETURN Compare_Result_Integer;

FUNCTION COMPARE( Lob1 IN CLOB CHARACTER SET Set_Desired,
   Lob2 IN CLOB CHARACTER SET Set_Destid%CHARSET,
   Number_Bytes_to_Compare IN INTEGER,
   Origin_Lob1 IN INTEGER := 1, Origin_Lob2 IN INTEGER := 1 )
RETURN Compare_Result_Integer;
```

- 参数含义：
  - Lob1 是要比较的第一个LOB；Lob2 是要比较的第二个LOB；
  - Number_Bytes_to_Compare 是要比较的 Lob1 和 Lob2 中的字节总数；

- Origin_Lob1 是要比较的文件的开始位置，1 表示从头开始比较。当使用CLOB时，Set_Desired用于指定字符集，如果没有字符集，则使用Oracle服务器本身的字符集。

◆ 返回值：
- 0: Lob1和Lob2 数据相同；
- 非0: Lob1和Lob2 数据不同；
- NULL: 其中一个参数无效(如：起始位置不正确、字节越界等)；

3) COPY：

复制一条记录中的LOB至另一条记录中。

```
PROCEDURE COPY(Dest_Locator IN OUT BLOB, Src_Locator IN BLOB,
   Amount IN OUT INTEGER,
   Dest_start_pos IN INTEGER :=1,
   Src_start_pos IN INTEGER :=1 );

PROCEDURE COPY(Dest_Locator IN OUT CLOB CHARACTER SET Set_Desired,
   Src_Locator IN CLOB CHARACTER SET Dest_Locator%CHARSET,
   Amount IN OUT INTEGER,
   Dest_start_pos IN INTEGER :=1,
   Src_start_pos IN INTEGER :=1 );
```

◆ 参数含义：
- Dest_Locator 是被原LOB 附加的目标LOB的定位器；
- Src_Locator 指出表示源LOB；
- Amount 指定了要拷贝多少数据；
- Dest_start_pos 指出了从源LOB的什么位置开始拷贝；
- Src_start_pos 指出了将拷贝的数据放到目标LOB的什么位置 ;当使用CLOB时,Set_Desired用于指定字符集，如果没有字符集，则使用Oracle服务器本身的字符集。

4) EMPLY_BLOB:

要将一个BLOB 添加到表中，需要使用EMPTY_BLOB给BLOB指定一个定位器。

```
FUNCTION EMPLY_BLOB() RETURN Locator;
```

◆ 返回值：
- Locator 是函数返回的BLOB 的定位器。

5) EMPLY_CLOB:

要将一个CLOB 添加到表中，需要使用EMPTY_CLOB给CLOB指定一个定位器。

```
FUNCTION EMPLY_CLOB() RETURN Locator;
```

◆ 返回值：
- Locator 是函数返回的CLOB 的定位器。

6) ERASE:

删除LOB 的部分或全部内容。

```
PROCEDURE ERASE(BLOB_Locator IN OUT BLOB, Amount IN OUT INTEGER,
   Start_Pos IN INTEGER := 1);
PROCEDURE ERASE(CLOB_Locator IN OUT CLOB, Amount IN OUT INTEGER,
   Start_Pos IN INTEGER := 1);
```

◆ 参数含义：
- BLOB_Locator和CLOB_Locator是给LOB指定的定位器；
- Amount 指定了要删除LOB中的多少数据；

- Src_start_pos 指出了从什么位置开始删除数据，1表示LOB的开头。

◆ 注意：

- 使用 GETLENGTH 函数得到 LOB 的长度，将Amount 指定为此值，以删除 LOB的所有内容。

7) GETLENGTH:

以字节为单位返回 LOB 的长度。

```
FUNCTION GETLENGTH(BLOB_Locator) RETURN INTEGER;
FUNCTION GETLENGTH(CLOB_Locator CHARACTER SET Set_desired) RETURN INTEGER;
```

◆ 参数含义：

- BLOB_Locator和CLOB_Locator是给LOB指定的定位器；当使用CLOB时，Set_Desired用于指定字符集，如果没有字符集，则使用Oracle服务器本身的字符集。

◆ 返回值：

- 返回 LOB 的长度；如果定位器为空，则返回NULL：

8) INSTR:

从指定的偏移开始，在LOB 中查找第 N 个模式匹配的字符串。

```
FUNCTION INSTR(BLOB_Locator, Patten IN RAW,
   Starting_location IN INTEGER := 1,
   Nth_Occurrence IN INTEGER := 1) RETURN Status_Integer;

FUNCTION INSTR(CLOB_Locator CHARACTER SET Set_Desired,
   Patten IN VARCHAR2 CHARACTER SET CLOB_Locator%CHARSET,
   Starting_location IN INTEGER := 1,
   Nth_Occurrence IN INTEGER := 1)
RETURN Status_Integer;
```

◆ 参数含义：

- BLOB_Locator和CLOB_Locator是给LOB指定的定位器；
- RAW 或 VARCHAR2 类型的 patten 是要在 LOB 中匹配的模式；
- Starting_location 是要搜索匹配的 LOB 的开始位置；
- Nth_Occurrence 是要进行的第N次匹配。当使用CLOB时，Set_Desired用于指定字符集，如果没有字符集，则使用Oracle服务器本身的字符集。

◆ 返回值：

- 0: 没有找到模式；
- >0: 找到的模式离头文件的偏移；
- NULL: 某个参数为空或者无效。

9) READ 函数：

将 LOB 的部分或全部内容读取到计算机的本地内存中。

```
PROCEDURE READ(BLOB_Locator, Read_Amount IN BENARY_INTEGER,
   Starting_Location IN INTEGER, Buffer OUT RAW);

PROCEDURE READ(CLOB_Locator CHARACTER SET Set_Desired,
   Read_Amount IN BENARY_INTEGER,
   Starting_Location IN INTEGER,
   Buffer OUT VARCHAR2 CHARACTER SET CLOB_Locator%CHARSET);
```

◆ 参数含义：

- BLOB_Locator和CLOB_Locator是给LOB指定的定位器；
- Read_Amount 是要从 LOB 中读取到缓冲区中的字节数；

- RAW 或 VARCHAR2 类型的 patten 是要在 LOB 中匹配的模式；
- Starting_location 是要从 LOB 中读取数据的开始位置；
- Buffer 是用于存取从 LOB 中读取的数据的位置；当使用CLOB时，Set_Desired用于指定字符集，如果没有字符集，则使用Oracle服务器本身的字符集。

◆ 注意：
- 如果任何一个参数为NULL，则会引发VALUE_ERROR异常；
- 如果任何一个参数无效，则会引发INVALID_AVGVAL异常；
- 如果达到 LOB 结尾，则将引发NO_DATA_FOUND 异常；

10) SUBSTR 函数：

从 LOB 中抽取指定数目的字节。

```
FUNCTION SUBSTR(BLOB_Locator, Read_Amount IN BINARY_INTEGER,
   Starting_Location IN INTEGER := 1);
RETURN RAW;

FUNCTION SUBSTR(CLOB_Locator  CHARACTER SET Set_Desired,
   Read_Amount IN BINARY_INTEGER, Starting_Location IN INTEGER := 1);
RETURN VARCHAR2 CHARACTER SET CLOB_Locator%CHARSET;
```

◆ 参数含义：
- BLOB_Locator和CLOB_Locator是给LOB指定的定位器；
- Read_Amount 是要从 LOB 中读取到缓冲区中的字节数；
- RAW 或 VARCHAR2 类型的 patten 是要在 LOB 中匹配的模式；
- Starting_location 是要从 LOB 中读取数据的开始位置；当使用CLOB时，Set_Desired 用于指定字符集，如果没有字符集，则使用Oracle服务器本身的字符集。

11) TRIM:

使 LOB 裁剪到指定的长度。

```
PROCEDURE TRIM(BLOB_Locator, New_Len IN INTEGER);
PROCEDURE TRIM(CLOB_Locator, New_Len IN INTEGER);
```

◆ 参数含义：
- BLOB_Locator和CLOB_Locator是给LOB指定的定位器；
- New_Len 是 LOB 的新长度。

12) WRITE:

将数据写入到 LOB 中。

```
PROCEDURE WRITE(BLOB_Locator, Amount IN OUT BENARY_INTEGER,
   Starting_Location IN INTEGER, Buffer IN RAW);

PROCEDURE WRITE(CLOB_Locator CHARACTER SET Set_Desired,
   Read_Amount IN OUT BENARY_INTEGER,
   Starting_Location IN INTEGER,
   Buffer IN VARCHAR2 CHARACTER SET CLOB_Locator%CHARSET);
```

◆ 参数含义：
- BLOB_Locator和CLOB_Locator是给LOB指定的定位器；
- Amount 是要从缓冲区写入到LOB中的字节数；
- Starting_location 指定将数据写入到LOB的什么位置；
- Buffer 是写入到LOB中的数据缓冲区；当使用CLOB时，Set_Desired用于指定字符集，如果没有字符集，则使用Oracle服务器本身的字符集。

5.3.4 内部 LOB 的函数和过程的应用举例

```
CREATE TABLE yang_test(eid NUMBER, ephoto CLOB);
INSERT INTO yang_test VALUES(1, 'test01');
INSERT INTO yang_test VALUES(2, EMPTY_CLOB());
INSERT INTO yang_test VALUES(3, 'welcome to my host. and thanks for your kind.');

DECLARE
   clob1     CLOB;
   clob2     CLOB;
   clob3     CLOB;
   l_count       INTEGER;
   l_start_p INTEGER;
   v_patten  VARCHAR2(6) := 'Oracle';
   v_Nth_Occ INTEGER := 1;
BEGIN
   SELECT ephoto INTO clob1 FROM yang_test WHERE eid = 1 FOR UPDATE;
   SELECT ephoto INTO clob2 FROM yang_test WHERE eid = 2 FOR UPDATE;
   SELECT ephoto INTO clob3 FROM yang_test WHERE eid = 3 FOR UPDATE;

   -- 得到 eid = 1 的CLOB 的长度
   l_count := DBMS_LOB.GETLENGTH(clob1);

   -- 将 eid = 1 的CLOB 拷贝到 eid = 2
   DBMS_LOB.COPY(clob2, clob1, l_count);

   -- 将 eid = 1 的CLOB 与 eid = 2 的相加并附值给 eid = 1
   DBMS_LOB.APPEND(clob1, clob2);
   COMMIT;

   l_start_p := DBMS_LOB.SUBSTR(colb1, v_patten, l_start_p, v_Nth_Occ);
   DBMS_LOB.ERASE(colb1, l_start_p, 1);

   COMMIT;
END;
/
```

5.3.5 临时 LOB

类似于局部变量，不会永久性的存在于数据库中，默认情况下，它们的寿命是整个会话期。最常用于转换 LOB 数据。

与持久性 LOB 相比，临时 LOB不会记录重复的记录，默认情况下，LOB 都是永久性的，可以显式的删除临时 LOB，以释放额外的资源和空间。

PL/SQL 通过定位器操作临时 LOB，方法同永久性 LOB。

由于临时 LOB 不会存在于数据库中，因此不能使用SQL 数据操纵语言(DML) 来操纵它们，而必须使用 DBMS_LOB包来操纵它们，方法与持久性 LOB 相同。

安全性是通过 LOB 定位器来提供，只有创建临时 LOB 的用户可以访问它。定位器不能从一个会话传递到另外的会话。

1）管理临时 LOB：

所有的临时 LOB 都被记录到V$TEMPORARY_LOBS 视图中。

2）创建临时的 LOB，例如：

```
DECLARE
   Dest_Blob BLOB;
BEGIN
   DBMS_LOB.CREATETEMPORARY(Dest_Blob, TRUE, DBMS_LOB.SESSION);
END;
```

注意：会话结束后，临时LOB将消失，并归还分配给它的所有内存和表空间。

第六章 管理事务和锁定

6.1 事务

事务是一个工作逻辑单元，由一条或多条数据操纵语句(DML)或数据定义语句(DDL)组成。Oracle 提供了两种通用的事务：只读事务和读写事务。只读事务规定，查询到的数据以及该事务中的查询将不受发生在数据库中的任何其他事务的影响。而读写事务保证查询返回的数据与查询开始的数据一致。

只读事务实现事务级读取的一致性，这种事务只能包含查询语句，而不能包含任何DML 语句。在这种情况下，只能查询到事务开始之前提交的数据。因此，查询可以执行多次，并且每次返回的结果都相同。

读写事务提供语句级读取的一致性，这种事务将看不到这查询执行期间提交的事务所做的修改。

1) 开始事务：

当第一条SQL 语句开始时，事务开始；当事务的效果被撤消或提交时，事务结束。SET TRANSACTION 也启动一个事务。语法：

```
SET TRANSACTION parameter
```

- 参数含义：
  - READ_ONLY：建立事务级读取的一致性；
  - READ WRITE：建立语句级读取的一致性；
  - USE ROLLBACK SEGMENT：定义要使用合适的撤消段；
  - ISOLATION LEVEL：规定如何处理DML事务，有两个选项：SERIALIZABLE 和 READ COMMITTED。SERIALIZABLE选项导致任何试图操纵已修改，但没有提交的数据对象的DML事务失败；READ COMMITTED 导致上述DML事务等待前面的DML锁定消失，这是Oracle的默认特性。

只读事务是所有事务的默认模式。对于这种模式，不用指定撤消段。另外，在事务期间，不能执行INSERT、UPDATE、DELETE 或 SELECT FOR UPDATE 子句命令。读写事务模式对事务中可以使用的DML语句没有限制。

SET TRANSACTION 命令可以给读写事务指定一个特定的撤消段。当执行撤消时，该撤消段被用于撤消当前事务所做的所有修改。如果没有指定撤消段，Oracle 将给事务指定一个撤消段。

2) 结束事务：

结束事务就是保存事务所做的修改(COMMIT)或所做的修改都被撤消(ROLLBACK)。

可以将 COMMENT 子句和 COMMIT 一起使用，把一个文本字符串和事务 ID 放到数据库字典中，可以通过查看视图 DBA_2PC_PENDING 来查看这些信息。通常，使用该视图以获取分布式环境中处于可疑状态的事务的额外信息。

要通过执行合适的命令来实现显式提交，必须有FORCE TRANSACTION 系统权限。要手工提交分布式环境中其他用户发起的事务，必须要有FORCE ANY TRANSACTION 系统权限。在每条 DDL语句之前或之后，Oracle 隐式的执行提交，Oracle 将自动执行这种提交。

3) 撤消事务：

执行 ROLLBACK 命令以撤消整个事务，执行 ROLLBACK TO SAVEPOINT 命令以撤消部分事务。

- 撤消整个事务时：
  - ① 使用相应的撤消段撤消当前事务所做的修改；

② 解除事务导致的对记录的锁定；
③ 结束事务；

◆ 撤消部分事务时：
① 只撤消最后的保存点后面执行的SQL 语句所做的修改；
② ROLLBACK 命令中指定的保存点不变，从数据库删除该保存后面的所有保存点；
③ 解除指定的保存点后面建立的锁定；
④ 事务仍然是活动的，而且可以继续执行；

注意：如果主机或应用程序发生严重故障，Oracle 隐式的执行撤消。

4）两步提交：

Oracle 通过两步提交机制来管理分布式事务的提交和撤消。

两步提交是一种确保参与分布式事务的数据库服务器要么都提交事务中的语句，要么都撤消事务中的语句的机制。两步提交机制还能保护完整性约束、远程过程调用和触发器执行的隐式DML操作。

在非分布式环境中，所有的事物都作为一个单元，要么被提交，要么被撤消。而在分布式环境中，提交和撤消一个分布式事务必须通过网络进行协调，以便所有参与的数据库要么都提交事务，要么都撤消事务，即使在分布式事务期间网络出现故障也需如此。两步提交机制确保参与事务的所有节点提交或撤消事务，以维护整个数据库的数据完整性。

5）使用保存点创建书签：

保存点用于将一个大型事务分成小块，这样就可以撤消到事务的中间点，而不是撤消整个事务。

在给定的事务中，保存点的名称必须是唯一的。如果创建一个名称与以前创建的保存点相同，则以前的保存点被删除。例如：

```
INSERT INTO emp(empno, ename) VALUES (10, 'test01');
SAVEPOINT sp1;
INSERT INTO emp(empno, ename) VALUES (11, 'test02');
ROLLBACK TO sp1;
```

说明：上述例子中，撤消了事务中对empno=11的插入操作；在撤消到保存点时，插入emp=10是未决数据。

6）release 选项：

通常当程序成功完成时，所有的锁定、游标以及一些内存将归还给系统。然而，如果程序异常终止，则一些锁定和游标在一段时间内仍然处于活动状态，这将给数据库带来不必要的开销，直到数据库发现终止并进行清理。例如：

```
EXEC SQL COMMIT RELEASE;
EXEC SQL ROLLBACK RELEASE;
```

以上语句强制程序结束时进行清理，以使得锁定、内存和游标被释放。

6.2 锁定

Oracle 通过锁记录让用户暂时拥有并控制诸如表、记录等数据对象。

1）锁定表：

DML 操作会对特定的记录和特定的表进行数据锁定。当表被多个用户同时访问时，这些锁用于保护表中的数据，并防止可能发生的 DDL 操作。但是，表锁定不能防止其他事务也锁定相同的表。

当表被下列DML 操作语句修改时，表被锁定：

```
INSERT、UPDATE、DELETE、SELECT..FOR UPDATE、LOCK TABLE。
```

对于所有的 SQL 语句，将自动发生隐式数据库锁定，因此数据库用户不用显式的锁定任何记录。默认情况下， Oracle 尽量以最低的级别锁定资源。

在多用户数据库中，有两种不同的锁定级别：

级 别	描 述
EXCLUSIVE	禁止共享相关资源，在解除锁定之前，最先获得资源的事务是唯一可以修改资源的事务
SHARE	根据涉及到的操作，这种锁定允许共享相关资源。多个事务可以获得对一个资源的共享锁定，提供了更高程度的数据并发性

表锁定可以以5种不同的模式执行：

模 式	描 述
ROW SHARE	是限制性最小的表锁定，它允许其他并发事务查询、插入、更新、删除或锁定同一个表中的记录。但不允许对同一个表进行独占式写访问
ROW EXCLUSIVE	当一个表中的多条记录被更新时发生这种表锁定。它允许其他并发事务查询、插入、更新、删除或锁定同一个表中的记录。不能防止任何对同一个表的手工锁定或独占式读写访问
SHARE LOCK	只允许其他用户查询或锁定特定的记录。这种锁定防止对同一个表的任何更新、插入和删除操作
SHARE ROW EXCLUSIVE	这种锁定只允许用于UPDATE 语句的查询和选择
EXCLUSIVE	这种锁定允许事务写一个表，其他事务只能查询

2）锁定记录：

当记录被下列DML 操作语句修改时，将自动获得记录锁定：

```
INSERT、UPDATE、DELETE、SELECT..FOR UPDATE。
```

这些记录锁定在事务完成或撤消之前一直有效。记录锁定总是独占式的，它禁止其他事务修改相同的记录。当记录锁定执行后，将执行相应的表锁定，以防止任何冲突的DDL语句起作用。

3）显式锁定：

```
LOCK TABLE table_name1, table_name2 IN lock_mode MODE NOWAIT;
```

其中：table_name是将要锁定的表名称；lock_mode是要锁定的模式。NOWAIT是可选项，如果指定它，则如果数据库已被锁定，将立即把控制返回给事务；如果省略它，则事务等待已有的锁定被解除后，锁定该数据对象。

当执行 LOCK TABLE 语句并覆盖默认的锁定机制时，事务显式获得特定表的锁定。当对视图执行LOCK TABLE 语句时，将锁定底层的基础表。

4）其他锁定(次要锁定)：

次要锁定是用户通常不直接打交道的锁定。

名 称	描 述
字典锁定	防止事务期间修改数据库对象。当DDL 需要这种锁定，Oracle 自动获得这种锁定。字典锁定可以是独占或共享方式
内部锁定	保护数据库和内存中的内部组件。终端用户不能访问这些组件，例如：可以对记录文件、控制文件、数据字典缓冲文件和归档文件进行锁定。
分布式锁定	确保多个实例间的数据一致性。当需要时，Oracle 自动创建这种锁定

5）监视锁定：

① 可以通过 **V$LOCK** 数据库视图来查找锁定的信息；

② 可以通过 **DBA_DDL_LOCKS、DBA_DML_LOCKS、DBA_LOCKS** 数据库表来查找锁定的信息；

6）DBMS_LOCK 包：

DBMS_LOCK 包用来管理数据库锁定。通过此包可以请求特定的锁定、给它指定唯一的名称、改变锁模式以及解除锁定。

① ALLOCATE_UNIQUE：

提供一个锁定名称时，ALLOCATE_UNIQUE函数分配一个唯一的锁定标识符，此锁定标识符界于 1073741824 到 1999999999之间。

如果选择用名称标识锁定，可以使用ALLOCATE_UNIQUE为这些已命名的锁生成一个唯一的锁定标识符。

第一个会话使用一个新的锁定名调用ALLOCATE_UNIQUE创建一个唯一的锁定ID，并存贮在DBMS_LOCK_ALLOCATED 表中，接下来调用(通常由另一个会话调用)返回前面创建的锁定ID。其语法如下：

```
ALLOCATE_UNIQUE(lockname IN VARCAHR2, lockhandle OUT VARCHAR2,
   expiration_secs IN INTEGER DEFAULT 864000)
```

- 参数含义：
  - lockname是要生成一个唯一的ID 的锁定名；
  - lockhandle为过程生成的唯一的标识符；
  - expiration_secs 是给定的锁定执行最后的ALLOCATE_UNIQUE 后，将该锁定从DBMS_LOCK_ALLOCATED表中删除该锁定之前需要等待的秒数。

② REQUEST:

请求一个给定模式的锁定。

```
FUNCTION REQUEST(id IN INTEGER || lockhandle IN VARCHAR2,
   lockmode IN INTEGER DEFAULT X_MODE,
   timeout IN INTEGER DEFAULT MAXWAIT,
   release_on_commit IN BOOLEAN DEFAULT FALSE)
RETURN INTEGER;
```

- 参数含义：
  - id 是指定的锁定标识符或 ALLOCATE_UNIQUE 过程返回的lockhandle
  - Lockmode 是请求的锁定模式，其可能值为：
    - 1 ------ NULL 模式
    - 2 ------ 记录共享模式
    - 3 ------ 记录独占模式
    - 4 ------ 共享模式
    - 5 ------ 共享记录独占模式
    - 6 ------ 独占模式
  - timeout 是在因错误而超时之前授予锁定应该尝试的秒数；
  - release_on_commit 指出是在执行提交还是撤消时解除锁定；
- 返回值：
  - 0 ------- 成功执行
  - 1 ------- 因超时而失败
  - 2 ------- 检测到死锁
  - 3 ------- 语法或参数错误
  - 4 ------- 已经拥有指定的锁定 ID 或句柄
  - 5 ------- 非法的锁定语句

③ CONVERT:

将锁定从一种模式转换为另外一种模式。

```
FUNCTION CONVERT( id IN INTEGER || lockhandle IN VARCHAR2,
   lockmode IN INTEGER,
   timeout IN NUMBER DEFAULT MAXWAIT)
RETURN INTEGER;
```

- 参数含义：
  - id 是指定的锁定标识符或 ALLOCATE_UNIQUE 过程返回的lockhandle
  - Lockmode 是请求的锁定模式，其可能值为：

- 1 ------ NULL 模式
- 2 ------ 记录共享模式
- 3 ------ 记录独占模式
- 4 ------ 共享模式
- 5 ------ 共享记录独占模式
- 6 ------ 独占模式

➢ timeout 是在因错误而超时之前授予锁定应该尝试的秒数；

◆ 返回值：

➢ 0 ------- 成功执行
➢ 1 ------- 函数超时
➢ 2 ------- 检测到死锁
➢ 3 ------- 语法或参数错误
➢ 4 ------- 指定的锁定 ID 或句柄不属于当前用户
➢ 5 ------- 非法的锁定句柄 或 ID

④ RELEASE:

解除通过REQUEST 函数显式获得的锁定。

```
FUNCTION RELEASE( id IN INTEGER || lockhandle IN VARCHAR2 )
   RETURN INTEGER;
```

◆ 参数含义：

➢ id 是指定的锁定标识符或 ALLOCATE_UNIQUE 过程返回的lockhandle

◆ 返回值：

➢ 0 ------- 成功执行
➢ 3 ------- 语法或参数错误
➢ 4 ------- 指定的锁定 ID 或句柄不属于当前用户
➢ 5 ------- 非法的锁定句柄 或 ID

⑤ SLEEP:

将当前的会话挂起一段时间。

```
PROCEDURE SLEEP(Seconds IN NUMBER);
```

◆ 参数含义：

➢ Seconds 是会话将要挂起的秒数。

第七章 动态 SQL

执行动态SQL有两种方式：一是 DBMS_SQL 程序包；二是本机动态 SQL。

7.1 DBMS_SQL 程序包

功能强大，函数众多，很少使用。

7.2 本机动态 SQL

7.2.1 执行 DDL 语句

```
BEGIN
```

```
    EXECUTE IMMEDIATE 'CREATE TABLE yang_test (eid VARCHAR2(30) PRIMARY KEY)';
    EXECUTE IMMEDIATE 'ALTER TABLE yang_test ADD (ePhoto CLOB)';
    EXECUTE IMMEDIATE 'ALTER TABLE yang_test MODIFY (eid NUMBER)';
    EXECUTE IMMEDIATE 'DROP TABLE yang_test';
END;
/
```

7.2.2 使用绑定变量

```
DECLARE
    l_sql     VARCHAR2(200);
    l_photo       yang_test.ePhoto%TYPE;
    l_id          yang_test.eid%TYPE;
BEGIN
    l_sql := 'INSERT INTO yang_test VALUES(:1, :2)';
    EXECUTE IMMEDIATE l_sql USING 10, 'welcome to our couty';
    COMMIT;

    l_sql := 'SELECT ePhoto, eid FROM yang_test WHERE eid = :1';
    EXECUTE IMMEDIATE l_sql INTO l_photo, l_id USING 10;
    DBMS_OUTPUT.PUT_LINE(l_id);
    DBMS_OUTPUT.PUT_LINE(l_photo);
END;
/

注意：运行以上代码，查看结果！！！
```

7.2.3 执行 PL/SQL 块

```
DECLARE
    l_id  NUMBER;
    l_sql VARCHAR2(200) := 'BEGIN  SELECT eid INTO :1 FROM yang_test; END;';
BEGIN
    EXECUTE IMMEDIATE l_sql INTO l_id;
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(l_id));
END;
/

注意：运行以上代码，查看结果！！！
```

第八章 显示数据

DBMS_OUTPUT包是一个可以用SQL*Plus将输出显示到屏幕上的包；UTL_FILE是一个服务器端的内置读写文件包，可以在服务器端读写数据；TEXT_IO是一个客户端的读写包，可以在客户端读写数据。

8.1 DBMS_OUTPUT 程序包

利用SET SERVEROUTPUT ON 开启屏幕显示时，实际上是命令SQL*Plus 在每一条语句后检查缓冲区中的数据，以取回并显示它。同时，DBMS_OUTPUT也可以用于两个PL/SQL 过程之间交换数据。

8.1.1 开启屏幕显示

1. 使用命令行：

```
SET SERVEROUTPUT ON;              (Oracle 8.0 以前版本)
SET SERVEROUTPUT ON SIZE 20000;   (Oracle 8.0 以后版本)
```

说明：上述语句隐含的调用了DBMS_OUTPUT.ENABLE，并为缓冲区分配了20000个字节的空间。
注意：使用DBMS_OUTPUT将数据发送到SQL*Plus 时，在所有数据都发送之前，不能开始读取数据!

2. 使用过程：

```
BEGIN
   DBMS_OUTPUT.ENABLE(100000);
END;
```

8.1.2 关闭屏幕显示

1. 使用命令行：

```
SET SERVEROUTPUT OFF;
```

2. 使用过程：

```
BEGIN
   DBMS_OUTPUT.DISABLE;
END;
```

8.1.3 其他函数

1) DBMS_OUTPUT.PUT_LINE(type);
2) DBMS_OUTPUT.PUT(type); -- 其中 type 可以是NUMBER、VARCHAR2、DATE数据类型。
3) DBMS_OUTPUT.GUT_LINE(line OUT VARCHAR2, status OUT INEGER);
4) DBMS_OUTPUT.GUT_LINES(line OUT VARCHAR2, numlines IN OUT INEGER);
   - line 是取回的行；
   - status 指出是否取回一行，1表示从缓冲区取回一行，0表示不取回数据；
   - numlines 作为输入参数表示要取回的行数，作为输出参数表示实际取回的行数；
5) DBMS_OUTPUT.NEW_LINE; -- 输出一个空行；

8.1.4 引发的异常

异 常 码	错 误 描 述	修 改 措 施
ORU_10027	缓冲区溢出	增大缓冲区
ORU_10028	行长溢出，每行限制为255个字符	确保每行都小于255个字符

8.2 UTL_FILE 程序包

8.2.1 概述

使用 UTL_FILE 有两个前提条件：

- 必须有执行 UTL_FILE 包的权限；

```
CONN sys/change_on_install;
GRANT EXECUTE ON UTL_FILE TO scott;
```

- 数据库管理员必须设置一个名为 UTL_FILE_DIR 参数；
  在init.ora 文件中添加 UTL_FILE_DIR = c:\oracle,c:\tmp,c:\test

UTL_FILE 只能在所添加的目录下写文件，不能在其他目录（包含其下的子目录）下写文件

说明：

在大多数系统中，Oracle 以特权模式运行，使它能够访问所有的系统文件；当调用UTL_FILE 时，实际上就是 Oracle 为我们读写文件，这样，就带来了安全性的风险，因此在使用UTL_FILE之前，设置 UTL_FILE_DIR 参数，使之指定一固定目录，这样所有的 UTL_FILE 的 I/O 操作都必须在其中的一个目录中进行。

8.2.2 函数描述

1) FOPEN：

打开文件，返回一个指向被打开文件的句柄。

```
FUNCTION FOPEN(loc IN VARCHAR2, fname IN VARCHAR2, openmode IN VARCHAR2 )
RETURN FILE_TYPE;

FUNCTION FOPEN(loc IN VARCHAR2, fname IN VARCHAR2,
   openmode IN VARCHAR2, MaxLine IN BINARY_INTEGER )
RETURN FILE_TYPE;
```

- 参数含义：
  - loc 是包含文件的目录名称，必须与为UTL_FILE_DIR参数列出的目录之一匹配；
  - filename 是文件名称，也可以包含扩展名。
  - openmode 是打开文件的模式。对于读文件用'R'，对于写文件用'W'，对于附加到已有文件的末尾，用'A'；
  - MaxLine 指定默认的行的大小。允许的范围为1--32767；缺省值为1023；

- 可能引发的异常

引发的异常	描 述
UTL_FILE.INVALID_PATH	目录无效，应该核对UTL_FILE_DIR
UTL_FILE.INVALID_MODE	指定的模式无效，打开模式必须是 R、W、A
UTL_FILE.INTERNAL_ERROR	发生内部错误
UTL_FILE.INVALID_OPERATION	由于某些其他原因不能打开文件，应该核对访问此目录的权限

2) FCLOSE：

关闭文件，如果被关闭的文件使用的缓冲区不为空，则在关闭文件之前，将其中的内容写入到磁盘中。

```
PROCEDURE FCLOSE(filehandle IN OUT FILE_TYPE);
```

- 参数含义：
  - filehandle 是 调用FOPEN 时返回的文件句柄。

3) FCLOSE_ALL：

关闭所有文件，不为空的所有缓冲区。

```
PROCEDURE FCLOSE_ALL;
```

- FCLOSE_ALL 和 FCLOSE可能引发的异常

引发的异常	描 述
UTL_FILE.INVALID_FILEHANDLE	无效的文件句柄(文件可能没有打开)
UTL_FILE.WRITE_ERROR	操作系统不能写此文件
UTL_FILE.INTERNAL_ERROR	发生内部错误

4) GET_LINE：

将数据从文件输入到缓冲区中。

```
PROCEDURE GET_LINE(filehandle IN FILE_TYPE, Buffer OUT VARCHAR2);
```

- 参数含义：
  - filehandle 是 调用FOPEN 时返回的文件句柄。
  - Buffer 是将数据从文件输入到缓冲区中的位置。
- 可能引发的异常

引发的异常	描 述
UTL_FILE.INVALID_FILEHANDLE	无效的文件句柄(文件可能没有打开)
UTL_FILE.VALUES_ERROR	缓冲区不够大，小于从文件中读取的行。增大缓冲区
UTL_FILE.INVALID_MODE	指定的模式无效，打开模式必须是 R、W、A
UTL_FILE.INTERNAL_ERROR	发生内部错误
UTL_FILE.NO_DATA_FOUND	到达文件的末尾
UTL_FILE.INVALID_OPERATION	由于某些其他原因不能打开文件，应该核对访问此目录的权限

- 注意：

  使用GET_LINE读取文件时，它能处理的最大行长度是在打开文件时指定的。其默认值为1023个字节，不包括换行符。换行符被裁掉并不返回。

5) IS_OPEN：

检查函数是否打开。

```
FUNCTION IS_OPEN(filehandle IN FILE_TYPE) RETURN BOOLEAN;
```

- 参数含义：
  - filehandle 是 调用FOPEN 时返回的文件句柄。
- 返回值：
  - true：文件已经打开；
  - false：文件没有打开；

6) NEW_LINE：

将一个或多个换行符写入到文件中。文件必须以输出模式(A或W)打开。不能将换行符写入到正在读取的文件中。

```
FUNCTION NEW_LINE(filehandle IN FILE_TYPE, lines IN NATURAL := 1);
```

- 参数含义：
  - filehandle 是 调用FOPEN 时返回的文件句柄。
  - lines 是写入到文件的换行符总数，默认值为 1。

7) PUT：

将字符串写入到输出文件中，但不在后面添加换行符。

```
FUNCTION PUT(filehandle IN FILE_TYPE, Buffer IN VARCHAR2);
```

- 参数含义：
  - filehandle 是 调用FOPEN 时返回的文件句柄。
  - Buffer 是写入到文件中的文本。一次写入的字符数不能超过该文件OPEN时指定的行大小，默认值为1023。

8) PUT_LINE：

将字符串写入到输出文件中，后面添加换行符。

```
FUNCTION PUT_LINE(filehandle IN FILE_TYPE, Buffer IN VARCHAR2);
```

- 参数含义同 PUT 函数。

9) PUTF：

类似于 C 语言中的printf 函数，格式化输出。

```
PROCEDURE PUTF(filehandle IN FILE_TYPE,
```

```
    format IN VARCHAR2,
    arg1 IN VARCHAR2 DEFAULT NULL,
    .....
    arg5 IN VARCHAR2 DEFAULT NULL);
```

- 参数含义：
  - filehandle 是 调用FOPEN 时返回的文件句柄。
  - format 是要写入到文件中的字符串。

10) FFLUSH：

使用任何一个PUT命令时，数据都被存贮在UTL_FILE包的缓冲区中，当缓冲区添满数据后，内容被写到文件中。如果需要立刻清洗缓冲区的内容，可以调用FFLUSH过程。

```
PROCEDURE FFLUSH(filehandle IN FILE_TYPE);
```

- 参数含义：
  - filehandle 是 调用FOPEN 时返回的文件句柄。
- NEW_LINE、PUT、PUT_LINE、PUTF、FFLUSH 可能引发的异常：

引发的异常	描 述
UTL_FILE.INVALID_FILEHANDLE	无效的文件句柄(文件可能没有打开)
UTL_FILE.WRITE_ERROR	操作系统不能写此文件
UTL_FILE.INTERNAL_ERROR	发生内部错误
UTL_FILE.INVALID_OPERATION	试图写一个不是以写模式(A或W)打开的文件

8.2.3 例程

```
DECLARE
    l_file UTL_FILE.FILE_TYPE;
BEGIN
    -- Open the file
    l_file := UTL_FILE.FOPEN('C:\TEST', 'TEST.TXT', 'W');

    -- Close the file
    UTL_FILE.FCLOSE(l_file);

EXCEPTION
    WHEN UTL_FILE.INTERNAL_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('UTL_FILE: An internal error occurred.');
        UTL_FILE.FCLOSE_ALL;
    WHEN UTL_FILE.INVALID_FILEHANDLE THEN
        DBMS_OUTPUT.PUT_LINE('UTL_FILE: The file handle is invalid.');
        UTL_FILE.FCLOSE_ALL;
    WHEN UTL_FILE.INVALID_MODE THEN
        DBMS_OUTPUT.PUT_LINE('UTL_FILE: An invalid mode is given.');
        UTL_FILE.FCLOSE_ALL;
    WHEN UTL_FILE.INVALID_OPERATION THEN
        DBMS_OUTPUT.PUT_LINE('UTL_FILE: An invalid operation is attemped.');
        UTL_FILE.FCLOSE_ALL;
    WHEN UTL_FILE.INVALID_PATH THEN
        DBMS_OUTPUT.PUT_LINE('UTL_FILE: An invalid path was given.');
        UTL_FILE.FCLOSE_ALL;
    WHEN UTL_FILE.READ_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('UTL_FILE: An read error occurred.');
        UTL_FILE.FCLOSE_ALL;
    WHEN UTL_FILE.WRITE_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('UTL_FILE: An write error occurred.');
```

```
        UTL_FILE.FCLOSE_ALL;
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('OTHERS: Some other error occurred.');
        UTL_FILE.FCLOSE_ALL;
END;
/
```

8.3 TEXT_IO 程序包

TEXT_IO 包和 UTL_FILE 包类似，但是TEXT_IO 包只能读写客户机上的文件。TEXT_IO包通常不是 Oracle 数据库的组成部分，而是 Oracle Develop 2000 上自带的。例程：

```
CREATE OR REPLACE PROCEDURE yang_text IS
    l_file TEXT_IO.FILE_TYPE;
BEGIN
    TEXT_IO.PUT_LINE('I am about to open a file');
    TEXT_IO.FOPEN('d:\test.txt', 'W');
    TEXT_IO.PUT_LINE('File is opened.');
    TEXT_IO.PUT_LINE(l_file, 'Yang');
    TEXT_IO.PUT_LINE('Write data to the file.');
    TEXT_IO.FCLOSE(l_file);
    TEXT_IO.PUT_LINE('The file has been closed.');
END;
/
注意： 编译有错误，如何设置系统参数！
```

第九章 管理数据作业

作业是一个存贮程序，它被安排在特定的时间运行，或者在特定的事件发生后运行。

9.1 DBMS_JOB 包

DBMS_JOB 包是将作业提交到作业队列。作业队列是一个保存安排的作业的地方。通过作业队列，可以安排这些作业执行的时间以及执行的频度，还可以查找当前的关于当前运行的作业、终止的队列、作业调度的信息或其他作业的信息。

过 程 名	描述
BROKEN	禁止作业运行，如果代码被标记为损坏，Oracle 将不执行它
CHANGE	修改指定作业的详细资料，如作业描述、作业运行时间或作业执行的间隔
INTERVAL	修改指定作业执行的间隔
NEXT_DATE	修改指定作业下一次执行的时间
REMOVE	从队列中删除特定的作业
RUN	强制执行特定的作业
SUBMIT	将作业提交给作业队列
WHAT	修改特定作业的作业描述

9.2 使用后台进程

Oracle 不是为每个用户调度或运行作业而运行同样的多个程序，从而浪费大量的资源；而是提供了SNP(快照刷新进程)后台进程，这些后台进程共享相同的函数和代码，这使它们能够监控可能的并发执行的 Oracle 进程。

SNP 的一个关键功能是：作业失败而不降低数据库的性能；而且，SNP 进程按用户指定的时间间隔监控作业，

启动所有需要执行的进程，然后等待下一个时间间隔。

Oracle 最多可提供 10 个SNP 进程，它们被标识为 SNP0 到 SNP9。一些参数在文件init.ora 中定义。

SNP 参数

参 数	取值范围	默认值	描 述
JOB_QUEUE_PROCESS	0 — 10	0	决定为每个实例启动的后台进程数
JOB_QUEUE_INTERVAL	1 — 3600	60	SNP搜索要执行的作业的时间间隔(以秒为单位)
JOB_QUEUE_KEEP_CONNECTIONS	TRUE/FALSE	FALSE	如果为TRUE，则在作业完成之前所有的数据库连接都保持打开；否则，根据需要打开和关闭连接

注意：每一个作业使用一个进程，因此不能有跨越多个进程执行的作业。如果JOB_QUEUE_PROCESS 被设置为0(默认值)，则作业不能执行。一定要在init.ora中修改此参数。

9.3 执行作业

执行作业的方法有两种：

1) 通过将作业提交到作业队列中实现定时执行；

2) 立刻执行。

9.3.1 使用 SUBMIT 将作业提交给作业队列

```
PROCEDURE SUBMIT(
   job_name         OUT BINARY_INTEGER,
   job_to_submit    IN VARCHAR2,
   next_run         IN DATE DEFAULT SYSDATE,
   interval         IN VARCAHR2 DEFAULT NULL,
   job_parsing      IN BOOLEAN DEFAULT FALSE,
   instance         IN BINARY_INTEGER DEFAULT ANY_INSTANCE,
   force            IN BOOLEAN DEFAULT FALSE);
```

- 参数含义：
  - job_name 是给进程指定的作业号，在作业存在期间作业号不变。一个进程只能指定一个作业号。
  - job_to_submit 是要提交的PL/SQL 代码；
  - next_run 是下一个作业运行的日期；
  - interval 是下一个作业运行的时间；
  - job_parsing: 如果设置为FALSE，则Oracle将分析作业，确保所有的对象都存在。如果并不是所有的对象都存在，应该将其设置为TRUE，否则该作业成为损坏的作业。
  - instance 指定哪个实例可以运行这个作业；
  - force：如果为TRUE，则任何正整数都可以被当作作业实例接收；如果为FALSE(默认值)，则指定的实例必须正在运行，否则例程将引发一个异常。

- 例程：

```
DECLARE
   v_JobNum  BINARY_INTEGER;
BEGIN
   DBMS_JOB.SUBMIT(v_JobNum, 'Weekly(''maintenance'', 1000, ''Friday'');',
                   SYSDATE, 'NEXT_DAY(TRUNC(SYSDATE),''Friday'') + 22/24);
END;
/
```

说明：上例调用一个存贮过程，该存贮过程传递三个参数。注意参数上的引号和函数后的逗号。

9.3.2 使用 RUN 立即执行作业：

使用RUN 过程可以在作业被发送到作业队列之后立即执行它。

```
PROCEDURE RUN(job_num_specifiec IN BINARY_INTEGER);
```

为使用RUN 过程，必须知道指定给要执行的作业的作业号，当作业被执行时，作业的下一次执行时间被重置。

9.3.3 作业环境

执行作业时，下述变量将被存贮在Oracle中：

1) 当前用户；
2) 提交或修改作业的用户；
3) 当前用户模式
4) NLS_LANGUAGE;
5) NLS_CURRENCY;
6) NLS_ISO_CURRENCY;
7) NLS_NUMBERIC_CHARACTERS;
8) NLS_DATE_FORMAT;
9) NLS_DATA_LANGUAGE;
10) NLS_SORT;

注意： 执行作业后，NLS 参数被存贮。可以使用ALTER 过程修改这些特性。

作业一旦递交，Oracle 便将提交作业的用户指定为该作业的拥有者，只有作业的拥有者可以修改作业、根据需要执行作业、将作业从作业队列中删除。

Oracle 将下一个顺序作业号指定为存贮值 SYS.JOBSEQ。在删除作业之前，不能修改作业号，也不能将作业号指定给其他作业。总是可以使用ISUBMIT指定自己的作业号，但如果该作业号已经存在，则指定的作业不能执行。如果试图使用相同的作业号，将产生错误。

作业定义是要通过DBMS_JOB.SUBMIT包执行的PL/SQL 代码的标识符，这通常是一个存贮过程。对于作为普通的 PL/SQL参数时需要使用单引号的参数，都必须使用双引号；否则，当Oracle 在处理作业的过程中删除单引号时，参数将是无效的。下面列出了另外一些Oracle能够识别的特殊参数。

特殊的作业定义参数

参数	模式	描述
Job	IN	当前的作业
Next_date	IN/OUT	作业下一次执行的日期，默认值为SYSDATE
Broken	IN/OUT	作业状态：IN 值总为FALSE；对于OUT值，如果作业已损坏，则为TRUE；否则为FALSE

9.4 查看作业

视图是关于特定主题的信息的信息集合。

作业的数据字典视图

视 图	描 述
DBA_JOBS	显示数据库中的所有作业
USER_JOBS	与 DBA_JOBS 视图的结构相同
DBA_JOBS_RUNNING	显示数据库中当前正在运行的所有作业
DBA_JOBS	显示用户拥有的所有作业，用户ID为PRIV_USER

注意：在屏幕上输入 SET ARRAYSIZE 10 来减小数组的大小。

9.4.1 DBA_JOBS 视图的结构

字段名称	含 义
JOB	作业号
LOG_USER	与作业相关联的用户
PRIV_USER	提交并拥有作业的用户
LAST_DATE	最后一次成功执行的日期
LAST_SEC	最后一次成功执行的时间
THIS_DATE	当前正在执行的作业开始执行的日期
THIS_SEC	当前正在执行的作业开始执行的时间
NEXT_DATE	下一次作业开始执行的日期
NEXT_SEC	下一次作业开始执行的时间
TOTAL_TIME	执行作业所需的总时间(以秒为单位)
BROKEN	作业是否损坏的指示符，如果已损坏，则为 'Y'
WHAT	给SUBMIT 或ISUBMIT 提供的WHAT参数
INTERNAL	作业两次执行的时间间隔
FAILUSERS	最后一次成功执行后，作业启动并失败的次数

9.4.2 DBA_JOBS_RUNNING 视图的结构：

字段名称	含 义
JOB	作业号
SID	执行作业的进程列表
LAST_DATE	最后一次成功执行的日期
LAST_SEC	最后一次成功执行的时间
THIS_SEC	当前正在执行的作业开始执行的时间
FAILUSERS	最后一次成功执行后，作业启动并失败的次数

9.5 管理作业

作业管理包括删除作业、修改作业、在数据库间导入和导出作业、修复损坏的作业。
注意：如果作业启动的时间较长，请检查该作业。

9.5.1 删除作业

如果作业当前处于运行状态，则不能删除该作业。必须等到作业完成之后才可以删除。

```
PROCEDURE REMOVE( job_num IN BINARY_INTEGER);
```

- 参数含义：
  - job_num 是要删除的作业号。

  执行此过程后，如果该进程还没有执行，则将它从作业队列中永久删除。

9.5.2 修改作业

作业被提交以后，可以使用CHANGE过程修改参数。如果要修改作业的待定参数，则使用WHAT、NEXT_DATE 或 INTERVAL。

```
PROCEDURE CHANGE(job_num IN BINARY_INTEGER, process_name IN VARCHAR2,
   next_run in DATE, interval IN VARCHAR2,
   instance IN BINARY_INTEGER DEFAULT ANY_INSTANCE,
   froce IN BOOLEAN DEFAULT FALSE);

PROCEDURE WHAT(job_num IN BINARY_INTEGER, process_name IN VARCAHR2);
PROCEDURE NEXT_DATE(job_num IN BINARY_INTEGER, Next_run IN DATE);
PROCEDURE INTERVAL(job_num IN BINARY_INTEGER, interval IN DATE);
```

9.5.3 导入和导出作业

在数据库之间导入和导出作业时，源数据库中指定的作业号将成为目标数据库中的作业号。当目标数据库存在相同作业号时，作业号之间就产生冲突，只有在源数据库中再次提交该作业，并给它指定目标数据库中没有的作业号，才能导出该作业。

```
PROCEDURE USER_EXPORT(job_num IN BINARY_INTEGER, Des_database OUT VARCHAR2);
```

9.5.4 处理损坏的作业

损坏的作业是指连续16次执行失败的作业。对于这种作业，Oracle 在BROKEN字段中用一个标记标识它，并将TRUE存贮在这个字段中。只有用下面的方式才能执行：

1）使用DBMS_JOB.RUN来执行；

2）将BROKEN字段的标记改为'Fixed'，即BROKEN等于FALSE；

```
PROCEDURE BROKEN( Job_num IN BINARY_INTEGER,Broken_status IN BOOLEAN,
                  Next_date IN DATE DEFAULT SYSDATE);
```

使用这种方式将作业标记为已修复。

9.5.5 例程

注意：确保init.ora 文件中的UTL_FILE_DIR被设置为 = D:\Test，否则将得到一个USER_EXECPTION 错误。

1）编写三个过程：

```
CREATE OR REPLACE PROCEDURE yang_test1 AS
BEGIN
   DBMS_OUTPUT.PUT_LINE('Hai test1 ...' ||
          TO_CHAR(SYSDATE, 'YYYY-MM-DD HH24:MI:SS'));
END;
/

CREATE OR REPLACE PROCEDURE yang_test2 AS
   v_file UTL_FILE.FILE_TYPE;
BEGIN
   v_file := UTL_FILE.FOPEN('D:\TEST', 'test.txt', 'A');
   UTL_FILE.PUT_LINE(v_file, 'Hai test2 ...' ||
        TO_CHAR(SYSDATE, 'YYYY-MM-DD HH24:MI:SS'));
   UTL_FILE.FCLOSE(v_file);
EXCEPTION
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error: ' || TO_CHAR(SQLCODE) || SQLERRM);
      NULL;
END;
/
```

```
CREATE OR REPLACE PROCEDURE yang_test3 AS
   v_file UTL_FILE.FILE_TYPE;
BEGIN
   v_file := UTL_FILE.FOPEN('D:\TEST', 'test.txt', 'A');
   UTL_FILE.PUT_LINE(v_file, 'Hai test2 again ...' ||
        TO_CHAR(SYSDATE, 'YYYY-MM-DD HH24:MI:SS'));
   UTL_FILE.FCLOSE(v_file);
EXCEPTION
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error: ' || TO_CHAR(SQLCODE) || SQLERRM);
      NULL;
END;
/
```

2）编写提交作业给作业队列匿名包：

```
DECLARE
   v_jobnum  BINARY_INTEGER;
BEGIN
   DBMS_OUTPUT.ENABLE;
   DBMS_JOB.SUBMIT(v_jobnum, 'YANG_TEST1;', SYSDATE,
          'SYSDATE+(1/(24*60*60))');
   DBMS_OUTPUT.PUT_LINE('Frist Job Number assigned to test1 is:'|| v_jobnum);
   DBMS_JOB.SUBMIT(v_jobnum, 'YANG_TEST2;', SYSDATE,
          'SYSDATE+(1/(24*60*60))');
   DBMS_OUTPUT.PUT_LINE('Second Job Number assigned to test2 is:' ||
         v_jobnum);
   DBMS_JOB.ISUBMIT(110, 'YANG_TEST3;', SYSDATE, 'SYSDATE+(1/(24*60*60))');
   DBMS_OUTPUT.PUT_LINE('Third Job Number assigned to test3 is: 110');
END;
/
```

3）编写立即运行作业的匿名包：

```
DECLARE
   l_jobnum  NUMBER;
BEGIN
   SELECT job INTO l_jobnum FROM USER_JOBS WHERE what = UPPER('yang_test1;');
   DBMS_JOB.RUN(l_jobnum);

   SELECT job INTO l_jobnum FROM USER_JOBS WHERE what = UPPER('yang_test2;');
   DBMS_JOB.RUN(l_jobnum);

   SELECT job INTO l_jobnum FROM USER_JOBS WHERE what = UPPER('yang_test3;');
   DBMS_JOB.RUN(l_jobnum);
EXCEPTION
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error: ' || TO_CHAR(SQLCODE) || SQLERRM);
      NULL;
END;
/
```

4）编写删除作业的匿名包：

```
DECLARE
   l_jobnum  NUMBER;
BEGIN
   SELECT job INTO l_jobnum FROM USER_JOBS WHERE what = UPPER('yang_test1;');
   DBMS_JOB.REMOVE(l_jobnum);
```

```
        SELECT job INTO l_jobnum FROM USER_JOBS WHERE what = UPPER('yang_test2;');
        DBMS_JOB.REMOVE(l_jobnum);

        SELECT job INTO l_jobnum FROM USER_JOBS WHERE what = UPPER('yang_test3;');
        DBMS_JOB.REMOVE(l_jobnum);
    EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('Error: ' || TO_CHAR(SQLCODE) || SQLERRM);
            NULL;
    END;
    /
```

第十章 过程通信

10.1 报警(DBMS_ALERT 程序包)

DBMS_ALERT通常是一种在提交事务时触发的单向异步通信。除非事务被提交，否则不向报警发送任何消息，因此在事务被提交以前，等待的过程或应用程序将一直处于空闲状态。

由于DBMS_ALERT使用COMMIT，因此不能在Orcle Forms 中使用这个包。

由于DBMS_ALERT包是基于事务的，因此任何ROLLBACK 都将删除所有的报警。

要使用SYS.DBMS_ALERT包，要对此包有 EXECUTE 权限。

10.1.1 建立报警的次序

1) 使用 REGISTER 记录特定的报警；
2) 使用 WAITONE 过程，等待特定的报警；
3) 使用 WAITANY 过程，等待任何已注册了的报警；
4) 使用 SIGNAL 对已提交的事务并满足报警条件的进行报警。

10.1.2 函数应用和说明

1) 注册报警：

一个会话可以注册任意数量的报警。可以监视所有的已注册的报警。

```
PROCEDURE REGISTER(alert_name IN VARCHAR2);
```

- 参数含义：
  - alert_name 是要注册的报警的名称。

2) 等待特定的报警：

要监视一个特定的报警，使用WAITONE过程。

```
PROCEDURE WAITONE(alert_name IN VARCHAR2, alert_msg OUT VARCHAR2,
    alert_status OUT INTEGER, Timeout IN NUMBER DEFAULT MAXWAIT);
```

- 参数含义：
  - alert_name 是要注册的报警的名称；
  - alert_msg 报警被发送时收到的消息，此消息是通过SIGNAL 来调用发送的；
  - alert_status: 0--报警在超时之前被发送；1-- 超时之前没有收到任何报警；
  - Timeout：如果没有收到报警时，继续执行过程之前等待报警的时间(单位为秒)；
  - MAXWAIT 的缺省值为 1000 天。

注意：如果指定的alert_name 没有注册，将收到一个错误消息。

3）等待所有已注册的报警：

监视在当前会话中已注册的所有报警，用WAITANY过程。

```
PROCEDURE WAITANY(alert_name OUT VARCHAR2, alert_msg OUT VARCHAR2,
   alert_status OUT INTEGER, Timeout IN NUMBER DEFAULT MAXWAIT);
```

- 参数含义：
  - alert_name 返回收到被发送的首先注册为alert_name的报警；
  - alert_msg 报警被发送时收到的消息，此消息是通过SIGNAL 来调用发送的；
  - alert_status：0-- 报警在超时之前被发送；1-- 超时之前没有收到任何报警；
  - Timeout：如果没有收到报警时，继续执行过程之前等待报警的时间(单位为秒)；
  - MAXWAIT 的缺省值为 1000 天。

  注意：如果指定的alert_name 没有注册，将收到一个错误消息。

4）发布报警：

要发送报警时，需要使用SIGNAL过程。只有发布COMMIT后，此过程才会执行。

在多个会话向同一个报警发送信号的情况下，当每个会话发布报警时，它阻止其他并发的会话，直到它被提交为止。这种行为的实际效果是：报警可能导致事务序列化。

发送报警后，Oracle 将报警的状态从没有被发送改为已经被发送。此信息被记录在SYS.DBMS_ALERT_INFO 数据字典中。由于每个报警只有一条记录，因此在报警　被收到之前，任何试图发送报警的会话都被阻断。

如果没有会话注册这个报警，则这个报警将一直处于已被发送状态，直到事务注册它为止。如果有多个事务注册了这个报警，则报警被发送后，所有会话都将收到这个报警，报警返回到没有被发送的状态。

```
PROCEDURE SIGNAL(alter_name IN VARCHAR2, msg_sent IN VARCHAR2);
```

- 参数含义：
  - alert_name 最多可包含30个字符，并且不区分大小写。而且名称必须以ORA$开始，ORA$是为Oracle 保留的。
  - msg_sent 最多可以可以包含1800个字符，可以包含文本、变量等信息；此消息发送给等待的会话。

5）删除报警：

要删除注册列表中的一个特定报警，使用REMOVE过程。

不管是否等待一个报警，在报警注册后，报警将试图通知所有已注册的过程。因此，删除报警以节约资源是非常必要的。

```
PROCEDURE REMOVE(alert_name IN VARCHAR2);
```

- 参数含义：
  - alert_name 是要从注册列表中删除的报警名称。

6）删除所有的报警：

要删除注册列表中的所有报警，使用REMOVEALL过程。此调用隐含的执行了COMMIT。

```
PROCEDURE REMOVE;
```

7）轮讯：

当 Oracle 事件发生时，它将被放到系统中进行处理。WAITONE过程等待特定的事件发生，当报警发生时通知或者超时。但是在下列情况下，需要轮讯或者专门搜索一个报警：

① 当数据库的共享实例能够发布报警时，需要轮讯所有实例的报警。

② 当使用WAITANY过程时，需要搜索特定的报警。WAITANY过程进入循环轮讯模式，以搜索任何已注册的报警；在WAITANY 轮讯报警后进入睡眠模式时，如果在睡眠期间通知了三个报警，它只选择最后通知的报警。默认的轮讯从 1 秒开始，并按指数模式增加到 30 秒。

由于使用共享实例或WAITANY可能导致错过报警，因此可以用SET_DEFAULTS修改轮讯时间(以秒我单位)。

```
PROCEDURE SET_DEFAULTS(polling_interval IN NUMBER);
```

- 参数含义：
  - polling_interval 是以秒我单位指出轮讯的时间间隔，默认值为 5 秒。

10.1.3 应用举例

第一步：设置相应的权限；

```
启动SQL* PLus;
conn sys/change_on_install;
GRANT EXECUTE ON DBMS_ALERT TO scott;
conn scott/tiger;
```

第二步：对添加、删除或者修改员工表(scott.emp)进行报警设置；

```
CREATE OR REPLACE TRIGGER tri_emp
AFTER INSERT OR UPDATE OR DELETE ON emp
FOR EACH ROW
BEGIN
   IF INSERTING THEN
      DBMS_ALERT.SIGNAL('emp_changing', USER || ': inserting some data');
   ELSIF UPDATING THEN
      DBMS_ALERT.SIGNAL('emp_changing', USER || ': updating some data');
   ELSIF DELETING THEN
      DBMS_ALERT.SIGNAL('emp_changing', USER || ': deleting some data');
   END IF;
END;
```

第三步：注册并等待报警，并最终等待报警发生；

```
DECLARE
   l_msg     VARCHAR2(1800); -- 用于保存报警发送的消息；
   l_status      INTEGER;    -- 用于保存WAITONE过程的状态；
BEGIN
   DBMS_ALERT.REGISTER('emp_changing');
   DBMS_ALERT.WAITONE('emp_changing', l_msg, l_status, 60);
   DBMS_OUTPUT.PUT_LINE(l_msg);
   DBMS_ALERT.REMOVE('emp_changing');
END;
```

注意：名字要和第二步声明中的名字相同。

第四步：测试报警；

注意：因为报警属于会话间通讯，所以要启动两个SQL*Plus。

1) 在其中一个SQL*Plus着输入：

```
INSERT INTO emp(empno, ename) VALUES(8888, 'test01');
COMMIT;
DELETE FROM emp WHERE empno = 8888;
COMMIT;
```

2) 在另外一个SQL*Plus看输出的内容。

10.2 DBMS_PIPE 程序包

DBMS_PIPE 包可以在同一个数据库实例中的多个会话间通讯。它是通过管道发送和接收消息进行通讯。发送的消息称为写程序，接收的消息称为读程序，每一个管道都可以有一个或多个写程序、一个或多个读程序。任何能够访问数据库实例并执行PL/SQL代码的用户都可以访问管道。

管道的一个重要特性是它的异步性，即不必使用COMMIT便可以访问管道。另外，ROLLBACK 命令不使用

管道，这样就可以将管道用作一个功能强大的调试工具以及审计跟踪工具。如果需要对会话实现事务性控制，则DBMS_ALERT是有用的选择。

两个或多个终端可以使用相同的形式将数据发送到同一管道。

常见的错误：

1）没有安装DBMS_PIPE 包；

2）没有执行DBMS_PIPE 包的权限；

10.2.1 公有管道和私有管道

公有管道可以被数据库中的任何用户访问。

私有管道可以被DBA、管道的创建者、任何拥有者创建的存贮过程访问。

10.2.2 使用管道

操作管道的步骤：

1) 如果要创建私有管道，首先执行CREATE_PIPE函数，将隐式的创建一个公有管道。当不再包含数据时，这种隐式管道将消失；
2) 不管管道是私有的还是公有的，都是通过执行PACK_MESSAGE过程将要通过管道传输的数据发送到消息缓冲区；
3) 在缓冲区被添满之前，执行SEND_MESSAGE过程将数据发送到管道；如果创建的是公有管道，则默认情况下，SEND_MESSAGE将创建管道；
4) 为接收数据准备好后，调用RECEIVE_MESSAGE过程。每次调用此过程时，它首先读取管道中未读取的数据，并将数据转储存到消息缓冲区中。每当需要抽取下一条消息时，都需要调用RECEIVE_MESSAGE过程。如果需要知道数据的类型(可能变化)，可以调用函数NEXT_ITEM_TYPE。
5) 使用UNPACK_MESSAGE 从缓冲区中检索下一个消息。

在默认情况下，管道保留数据的时间长达1000天；然而，当实例被关闭时，被缓存在管道中的数据都将丢失。在 Oracle 中，此常数被定义为：

```
MAXWAIT CONSTANT INTEGER := 86400000
```

命名管道时，不能以ORA$开头，ORA$是保留给Oracle 使用的。管道命名不能超过128个字符。当然，还要确保管道名称是唯一的。当存在疑问时，使用函数UNIQUE_SESSION_NAME 给管道指定一个Oracle 定义的名称。

可以修改管道的默认值8192字节，同时，消息缓冲区的最大空间为4096个字节。

10.2.3 DBMS_PIPE 包的函数

名 称	类 型	描 述
CREATE_PIPE	函数	主要用于创建私有管道，但也能用于创建公有管道
NEXT_ITEM_TYPE	函数	抽取消息缓冲区中下一个项目的类型，主要用于拆开收到的消息
PACK_MESSAGE	过程	将数据发送到缓冲区，最后发送到管道
PURSE	过程	删除管道中的所有数据
RECEIVE_MESSAGE	函数	从管道接收一条消息并将它写入到消息缓冲区
REMOVE_PIPE	函数	从内存中删除一个管道
RESET_BUFFER	过程	清除消息缓冲区中的数据
SEND_MESSAGE	函数	将消息缓冲区中的所有数据发送到指定的管道，如果指定的管道不存在，则创建一个公有管道
UNIQUE_SESSION_NAME	函数	返回唯一的会话名
UNPACK_MESSAGE	过程	从消息管道中检索下一个条目

1) CREATE_PIPE函数：

创建私有管道。

```
FUNCTION CREATE_PIPE(pipe_name IN VARCAHR2,
   pipesize IN INTEGER DEFAULT 8192,
   private IN BOOLEAN DEFAULT TRUE);
RETURN INTEGER;
```

- 参数含义：
  - pipe_name: 给管道指定的名称；
  - pipesize：是管道的最大空间，默认值为8192字节；
  - private：FALSE 表示是公有管道；TRUE 表示是私有管道。
- 返回值：
  - 0：成功的创建了该管道；
  - ORA-23322：用户没有创建管道的权限或者此管道已存在。
- 注意：

  调用SEND_MESSAGE函数时，就不必用此函数来创建公有管道。

2) PACK_MESSAGE函数：

创建管道后，可以将数据发送到消息缓冲区，以便以后使用PACK_MESSAGE过程传输到管道。

```
PROCEDURE PACK_MESSAGE(data IN VARCHAR2);
PROCEDURE PACK_MESSAGE(data IN DATE);
PROCEDURE PACK_MESSAGE(data IN NUMBER);
```

- 参数含义：
  - data: 是要发送到缓冲区的数据，可以是VARCHAR2、DATE和NUMBER数据类型。
- 注意：

  缓冲区只有4096个字节可用。

3) SEND_MESSAGE函数：

在消息缓冲区充满之前，应该使用SEND_MESSAGE函数将数据发送到管道。此函数将消息缓冲区中的数据移到函数调用指定的管道中。

```
FUNCTION SEND_MESSAGE(pipe_name IN VARCAHR2,
   Timeout IN INTEGER DEFAULT MAXWAIT,
   pipesize IN INTEGER DEFAULT 8192);
RETURN INTEGER;
```

- 参数含义：
  - pipe_name: 给管道指定的名称；如果指定的管道不存在，则执行此函数后自动创建该管道。
  - Timeout: 是尝试将消息放到管道中的时间(以秒我单位)，默认值为1000天。
  - pipesize：是管道的最大空间，默认值为8192字节；
- 返回值：
  - 0：成功的发送了消息；
  - 1：在等待RECEIVE_MESSAGE函数从管道中清理出一些空间时，已超时；
  - 3：发送的消息被中断；

4) RECEIVE_MESSAGE函数：

此函数将消息从管道移到消息缓冲区中；然后，可以使用NEXT_ITEM_TYPE确定数据类型，或者使用UNPACK_MESSAGE读取消息，并将其用于进程中

```
FUNCTION RECEIVE_MESSAGE(pipe_name IN VARCAHR2,
   Timeout IN INTEGER DEFAULT MAXWAIT);
RETURN INTEGER;
```

- 参数含义：
  - pipe_name: 给管道指定的名称；如果指定的管道不存在，则执行此函数后自动创建该管道。

- Timeout：是尝试将消息放到管道中的时间(以秒为单位)，默认值为1000天。

◆ 返回值：

- 0：成功的发送了消息；
- 1：在等待RECEIVE_MESSAGE函数从管道中清理出一些空间时，已超时；
- 2：管道中的消息大于消息缓冲区的最大字节，即4096个字节；
- 3：发送的消息被中断；

5) UNPACK_MESSAGE函数：

将消息到缓冲区以后，需要使用此函数将消息从缓冲区中移到变量中。

```
PROCEDURE UNPACK_MESSAGE(data IN VARCHAR2);
PROCEDURE UNPACK_MESSAGE(data IN DATE);
PROCEDURE UNPACK_MESSAGE(data IN NUMBER);
```

◆ 参数含义：

- data：是要发送到缓冲区的数据，可以是VARCHAR2、DATE和NUMBER数据类型。

◆ 注意：

① 缓冲区只有4096个字节可用。

② 当试图拆开消息时，可能受到两种错误：ORA-06556 和 ORA-06559。

- ORA-06556 是试图读取空的消息缓冲区；
- ORA-06559 是请求的类型与消息缓冲区中的类型不同；

因此，在接收消息之前，应该使用NWXT_ITEM_TYPE函数确定缓冲区下一个项目的数据类型。

6) NEXT_ITEM_TYPE 函数：

```
FUNCTION NEXT_ITEM_TYPE RETURN INTEGER;
```

◆ 返回值：

- 0： 没有项目；
- 6： NUMBER；
- 9： VARCHAR2
- 11： ROWID；
- 12： DATE
- 23： RAW

◆ 注意：

可以将此函数用于处理管道中没有数据的异常以及确定从消息缓冲区传递的数据类型。

7) REMOVE_PIPE函数：

```
FUNCTION REMOVE_PIPE(pipe_name IN VARCHAR2) RETURN INTEGER;
```

◆ 返回值：

- 无论管道是否存在，返回值都为0；
- 异常ORA-23322 指出用户没有创建管道的权限。

◆ 注意：

管道被删除后，存贮在此管道中的所有消息也被删除。

8) PACK_MESSAGE_RAW函数：

对于将数据写入到消息缓冲区，利用PACK_MESSAGE_RAW函数来处理RAW 数据；但是，因为消息缓冲区的最大空间仍然是4096，所以不能使用LONG RAW 数据类型。

```
PROCEDURE PACK_MESSAGE_RAW(data IN VARCHAR2);
```

9) UNPACK_MESSAGE_RAW函数：

对消息缓冲区的RAW数据进行解码。

```
PROCEDURE UNPACK_MESSAGE_RAW(data OUT VARCHAR2);
```

10) PACK_MESSAGE_ROWID函数：

PACK_MESSAGE_ROWID函数可以将ROWID型数据类型发送到消息缓冲区。

```
PROCEDURE PACK_MESSAGE_ROWID(data IN VARCHAR2);
```

11) UNPACK_MESSAGE_ROWID函数：

对消息缓冲区的ROWID数据进行解码。

```
PROCEDURE UNPACK_MESSAGE_ROWID(data OUT VARCHAR2);
```

12) RESET_BUFFER函数：

在引发异常时清除缓冲区。

```
PROCEDURE RESET_BUFFER;
```

13) PURGE 函数：

删除指定管道中的所有数据，主要用于对数据处理之前清理管道和发生错误重置管道。

```
PROCEDURE PURGE(pipe_name IN VARCHAR2);
```

14) UNIQUE_SESSION_NAME 函数：

如果担心管道名已经存在,可以使用此函数给管道指定名称,此函数提供一个Oracle没有使用的名称。

```
PROCEDURE UNIQUE_SESSION_NAME RETURN VARCHAR2;
```

◆ 返回值：

➢ 返回一个最多可包含30个字符的唯一的名称。

10.2.4 例程

1) 授权：

```
启动SQL* PLus;
conn sys/change_on_install;
GRANT EXECUTE ON DBMS_PIPE TO scott;
conn scott/tiger;
```

2) 创建管道，并向管道中写入数据：

```
DECLARE
   v_statpipe1  INTEGER;
   v_statpipe2  INTEGER;
   v_pubchar VARCHAR2(100) := 'Yang test pipe in';
   v_pubdate DATE := SYSDATE;
   v_pubnum  NUMBER := 110;
BEGIN
   -- create private pipe
   v_statpipe1 := DBMS_PIPE.CREATE_PIPE('YANG_PRIV_PIPE');

   -- 检查是否被成功创建 **
   IF v_statpipe1 = 0 THEN
      DBMS_PIPE.PACK_MESSAGE('privateline1');
      DBMS_PIPE.PACK_MESSAGE('privateline2');
      -- Send message buffer to private pipe
      v_statpipe1 := DBMS_PIPE.SEND_MESSAGE('YANG_PRIV_PIPE');
   END IF;

   DBMS_PIPE.PACK_MESSAGE(v_pubchar);
   DBMS_PIPE.PACK_MESSAGE(v_pubdate);
   DBMS_PIPE.PACK_MESSAGE(v_pubnum);

   -- create public pipe and send message buffer to the pipe
   v_statpipe2 := DBMS_PIPE.SEND_MESSAGE('YANG_PUB_PIPE');

   -- Check status of both pipes to make sure they're created properly
   DBMS_OUTPUT.PUT_LINE('The Status of private pipe is: ' || v_statpipe1);
   DBMS_OUTPUT.PUT_LINE('The Status of public pipe is: ' || v_statpipe2);
END;
```

注意：在检查管道是否被成功创建时，最理想的是检查消息缓冲区是否溢出、是否被发送到管道、管

道是否被填满等。

3）从管道中读取数据：

```
DECLARE
   v_statpipe1  INTEGER;
   v_statpipe2  INTEGER;
   v_pubtype INTEGER;
   v_pubchar VARCHAR2(100);
   v_pubdate DATE := SYSDATE;
   v_pubnum  NUMBER := 110;
BEGIN
   -- 设置在放弃之前等待响应的时间为15秒
   v_statpipe1 := DBMS_PIPE.RECEIVE_MESSAGE('YANG_PRIV_PIPE', 15);
   DBMS_PIPE.UNPACK_MESSAGE(v_pubchar);
   DBMS_OUTPUT.PUT_LINE(v_pubchar);

   -- 设置在放弃之前等待响应的时间为15秒
   v_statpipe2 := DBMS_PIPE.RECEIVE_MESSAGE('YANG_PUB_PIPE', 15);
   LOOP
      v_pubtype := DBMS_PIPE.NEXT_ITEM_TYPE;
      IF v_pubtype = 0 THEN EXIT;   -- 没有数据
      ELSIF v_pubtype = 6 THEN   -- 数字
         DBMS_PIPE.UNPACK_MESSAGE(v_pubnum);
      ELSIF v_pubtype = 9 THEN   -- 字符串
         DBMS_PIPE.UNPACK_MESSAGE(v_pubchar);
      ELSIF v_pubtype = 12 THEN  -- 日期
         DBMS_PIPE.UNPACK_MESSAGE(v_pubdate);
      END IF;
   END LOOP;

   DBMS_OUTPUT.PUT_LINE(v_pubnum);
   DBMS_OUTPUT.PUT_LINE(v_pubchar);
   DBMS_OUTPUT.PUT_LINE(v_pubdate);
END;
```

4）删除两个管道：

```
DECLARE
   v_status  NUMBER;
BEGIN
   v_status := DBMS_PIPE.REMOVE_PIPE('YANG_PRIV_PIPE');
   DBMS_OUTPUT.PUT_LINE('The status for removing from private' ||
          ' pipe is : ' || v_status);

   v_status := DBMS_PIPE.REMOVE_PIPE('YANG_PUB_PIPE');
   DBMS_OUTPUT.PUT_LINE('The status for removing from public' ||
          ' pipe is : ' || v_status);
END;
```

10.3 DBMS_ALERT 与 DBMS_PIPE 的比较

- 相同点：
  1. 都使用异步通讯；
  2. 都在同一个实例中的不同消息间发送消息；
  3. 都能与 C 语言进行通讯；

4．都是PL/SQL 函数包。

◆ 不同点：

1．DBMS_ALERT 包使用COMMIT，而DBMS_PIPE包不使用COMMMIT。由于报警是基于事务的，因此使用提交，同时ROLLBACK 可以删除没有收到的所有等待警报；而对于管道，消息被发送后，则无法取消。

2．报警通常用于单向通信，而管道用于多向通讯。

3．DBMS_ALERT 包允许多个会话等待同一个报警，而且报警发送以后，所有的会话都将收到此报警，这与网络中的广播类似；对于DBMS_PIPE包，如果多个会话等待来自管道的消息，则只有一个会话接收到消息，然后就将清除管道。

第十一章 PL/SQL 和 JAVA

将 JAVA 引擎加到 Oracle 数据库中是 Oracle8i 版本中最激动人心的功能之一。因此：

1）将 JAVA 类和源代码装载到数据库中；

2）在 PL/SQL 中调用 JAVA 代码；

3）在 JAVA 中调用 PL/SQL 代码；

11.1 Oracle JAVA

1．Jserver：

由以下四部分组成：

1）JAVA 虚拟机(JVM)

2）JDBC 支持

3）SQLJ 运行支持

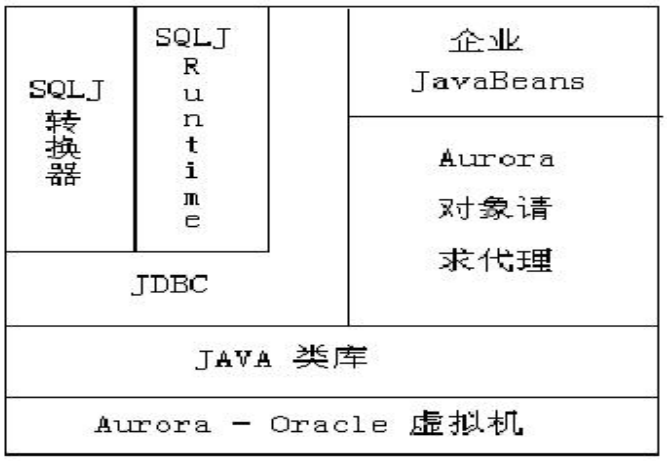
4）SQLJ 转换器

Oracle 公司的JVM的名称为Aurora，同时通过浏览新创建的Oracle8i数据库，将看到一个名为AURORAORBUNAUTHENTICATED 的用户，此用户用于到ORB的连接。

所有的JAVA组件都运行在与PL/SQL 相同的内存空间中，这使得它能够与数据库紧密的结合在一起。

Oracle 的Jserver完全是用C语言编写的，与SUN 公司的JAVA 1.1.6 兼容性包能很好的兼容。同时，由于Jserver是在数据库服务器范围内运行，因此Oracle不支持GUI 库，任何试图在服务器上使用GUI界面都将产生异常。

2．组成结构图：



3．配置系统环境变量：

1) 在WINNT 下的系统环境变量CLASSPATH中，将Oracle 数据库安装目录下的translator.zip加入到环境变量中。例如：

```
D:\Oracle817\sqlj\lib\translator.zip
```

4．装载JAVA类：

```
loadjava {-user|-u} username/password[@database]
   [-option [-option...]]  filename [filename...];
```

参数说明：

1) {-user|-u} username/password[@database] 指定了数据库的用户名称和密码，JAVA将被装载到此用户中。
2) [-option [-option...]] 的选项有：
   ① {andresove|a} 编辑源文件并在装载每个类时解析它，通常不使用。
   ② debug 产生调试信息。
   ③ {definer|d} 规定使用定义者的权限执行此类；否则使用调用者的执行权限。
   ④ {encodeing|e} 指定一种标准的JDK解码 模式，该解码模式必须与文件中使用的解码模式匹配。默认值为latin1
   ⑤ {force|f} 强行装载JAVA 类，即使以前装载过。
   ⑥ {grant|g} {usename|rolename}[{usename|rolename}...] 将执行被装载的类的权限授予给列出的用户和角色。
   ⑦ {help|h} 产生简短的屏幕帮助，以解释所有这些选项。
   ⑧ {oci8|O} 指出loadjava在与Oracle通信时使用基于Oracle调用接口(OCI)的JDBC驱动程序(对应于瘦驱动程序)。
   ⑨ Oracleresolver 解析对象引用的方式首先在用户的模式中查找，然后考虑公有对象。
   ⑩ {resolve|r} 为装载的类解析所有的外部引用。否则直到运行时才解析。
   (11).{resolve|R} 允许指定自己的解析器范围，控制如何解析其他对象的引用。
   (12).{schema|S} 将对象装载到指定的模式中。默认情况下，对象被装载到用户自己的模式中。
   (13).{synonym|s} 创建被装载的类的同义词。必须要有此权限。

```
GRANT CREATE ANY SYNONYM TO SCOTT;
```

   (14).{thin|t} 指出javaload 使用瘦JDBC驱动程序与数据库相连。
   (15).{verbose|v} 让loadjava 在装载文件时显示进度信息。

3) filename：
   可以是以下的扩展名称： JAVA类文件(.class)、

```
JAVA源文件(.java)、SQL文件(.sql)。
```

   注意：可以同时加载多个文件。

5．删除装载的JAVA类：

```
dropjava {-user|-u} username/password[@database]
   [-option [-option...]]  filename [filename...];
```

参数说明：

1) {-user|-u} username/password[@database] 指定了数据库的用户名称和密码，JAVA将被从此用户中删除。
2) [-option [-option...]] 的选项有：
   ① {help|h} 产生简短的屏幕帮助，以解释所有这些选项。
   ② {oci8|O} 指出loadjava在与Oracle通信时使用基于Oracle调用接口(OCI)的JDBC驱动程序(对应于瘦驱动程序)。

③ {schema|S} 将对象装载到指定的模式中。默认情况下，对象被装载到用户自己的模式中。
④ {thin|t} 指出javaload 使用瘦JDBC驱动程序与数据库相连。
⑤ {verbose|v} 让loadjava 在装载文件时显示进度信息。

3) filename：
可以是以下的扩展名称： JAVA类文件(.class)、
JAVA源文件(.java)、SQL文件(.sql)。
注意：可以同时删除多个文件，但文件原位置保持不变，仅从数据库中删除。

11.2 装载、应用、删除 JAVA

- 装载JAVA例程:
  - 第一步：创建如下JAVA类：

```
public class test{
    public static int test()
    {
        return 100;
    }
}
```

  - 第二步：将上述类加入到Oracle database 中：

```
    进入DOS状态，输入：
    loadjava -user scott/tiger@yang d:\test.java

注意： 如果发生java.lang.NoClassDefFounfError 异常，则可能是CLASSPATH 变量没有指向translator.zip文件。
```

  - 第三步：在PL/SQL中的调用方法：

```
CREATE OR REPLACE FUNCTION test_java RETURN NUMBER AS
LANGUAGE JAVA NAME 'test.test() return int';
/

DECLARE
    l_test NUMBER;
BEGIN
    SELECT test_java INTO l_test FROM DUAL;
    DBMS_OUTPUT.PUT_LINE(l_test);
    l_test := test_java;
    DBMS_OUTPUT.PUT_LINE(l_test);
END;
/
```

  - 第四步：从Oracle中删除此函数：

```
DROP FUNCTION test_java;
```

  - 第五步：从Oracle中删除此类：

```
    进入DOS状态，输入：
    dropjava -user scott/tiger@yang -verbose d:\test.java
```