ORACLE 傻 瓜 手 册

To be DBA or not to be, that is NOT the question. ---- Arron

作者 email：zhou_arron@163.com

Oracle8 以 8.1.5 为界分为普通版本和 internet 版本。普通版版本号 8.0.x，接触较多的是 8.0.5；internet 版版本号包括 8.1.5（Release 1），8.1.6（Release 2），8.1.7（Release 3）。普通版简称 Oracle 8，internet 版简称 Oracle 8i。如果不作特别说明，文中凡出现 Oracle 8i 均指 8.1.7 版。

Oracle9i 目前出到第二版，版本号为 9.2，简称 Oracle 9i。如果不作特别说明，文中凡出现 Oracle 9i 均指 9.2 版。

本手册介绍 Oracle 配置的基本方法，描述的是“所然”而不是“所以然”。全部操作以命令行方式出现，不涉及 GUI（只有白刃战才是真正的战斗）。鉴于大家对 Windows 已经十分熟悉，同时为了避免 Windows 和 Unix 两种截然不同的使用和开发风格给描述带来的复杂性，所以本手册不介绍在 Windows 上的 Oracle（上帝的归上帝，恺撒的归恺撒）。

文中所有例子以 oradb 作为数据库实例名，数据库用户 dbuser，口令 oracle。如果不作特别说明，关于 Oracle 8i 所有的例子都在 Solaris 8 Intel Platform+Oracle 8iR3 上通过，关于 Oracle 9i 所有的例子都在 RedHat Linux 7.3+Oracle 9iR2 上通过。附录文件 sample.tar 包含全部示例，简称附录。

大量使用表 emp 作为例子（参见附录 `08_proc/proc/single/emp.sql`）：

```
create table emp
(
        no              number(12) not null,
        name                char(20) not null,
        age             number(6) not null,
        duty                char(1) not null,
        salary              number(12) not null,
        upd_ts              date not null,
       primary key (no)
);
```

开发中对应 emp 表结构，定义其宿主结构（参见附录 `08_proc/proc/single/db.h`）：

```
typedef struct
{
        double          no;
        char            name[21];
        int             age;
        char            duty[2];
        double          salary;
        char            upd_ts[15];
} emp_t;
```

修改历史：

2000/07 版本 1.0

2000/09 版本 1.1

增加 Linux 安装，export,import 使用，数据库监控及优化（utlbstat,utlestat,分析 session），语言时间环境变量设置，Oracle8.0.5 手工建库脚本（wei_dick 提供，稍加修改）

2000/10 版本 1.2

修改 Linux 安装中 RedHat 6.x+Oracle 8.1.6、数据库优化中配置文件和 session 分析、常用技巧中下载上传文本数据和访问他机数据库；增加创建数据库实例中数据字典参考、常用技巧中删除冗余记录、应用开发，常见错误

感谢 liu_freeman,jiao_julian,huang_miles 等人对开发工具所作的努力

2001/03 版本 1.3

修改安装部分、init.ora 配置、常用技巧、应用开发；增加手工建库、MTS 配置；重写开发工具

感谢 li_bo 的大力帮助

2001/09 版本 1.4

修改数据库优化，使之较系统化；增加应用开发中多线程下的数据库连接

2002/04 版本 1.5

修改数据库优化、多线程条件下数据库编程；分离附录的程序范例

2002/12 版本 2.0

重新安排内容，增加 Oracle 9i 安装配置、OCI 开发、mysql 安装配置开发，补充数据库优化、PROC 开发

1 安装

所有参见内容都在附件 01_install_02_create_03_init/下。

1.1 通用设置

文件系统 swap

创建文件系统时应考虑 Oracle 对 swap 的需要，大约每个 oracle 服务进程将占用 10-20Mswap 空间，通常操作系统建议 2 倍于内存的 swap 空间，数据库系统可能要求更多些。

操作系统用户和环境变量

Oracle 文档要求为数据库系统的管理和使用建立 3 个或更多的组，但这个需求是可以忽略的，实践中并没有体现其必要性。为简化操作起见，只建立 dba 组，即拥有更新软件和管理最高权限（SYSDBA）的操作系统用户组，此组称为 OSDBA，属于此组的用户可以 SYSDBA 身份登录进任何一个数据库实例,简单的,只建立一个用户,习惯上使用 oracle 的名称。

```
$ groupadd dba
$ useradd –g dba –d /home/oracle –m –s /bin/bash oracle
```

确定 oracle 系统的根目录 **ORACLE_BASE**，如/opt/oracle，所有的软件和配置都在这个目录下展开，虽然并非一定需要如此，但这是一个良好的习惯。同时确定软件安装的起始点 **ORACLE_HOME**，通常在 ORACLE_BASE 下。

修改 oracle 用户的.profile,加入以下各行，或者修改/etc/profile，使每一个用户都获得环境变量设置

```
umask 022
ORACLE_BASE=/opt/oracle
ORACLE_HOME=$ORACLE_BASE/product/{版本号}（如 8.0.5,8.1.7,9.2.0 等）
ORACLE_SID=oradb
ORACLE_TERM=ansi  ＃仅与 Oracle8 字符界面安装有关
ORA_NLS33=$ORACLE_HOME/ocommon/nls/admin/data  ＃字符集支持
NLS_LANG=American_America.{ZHS16CGB231280(Oracle8 支持)|ZHS16GBK(Oracle8i 支持)|ZHS16GB18030(Oracle9i 支持)}
NLS_DATE_FORMAT=YYYYMMDDHH24MISS
LD_LIBRARY_PATH=$ORACLE_HOME/lib:$LD_LIBRARY_PATH  ＃动态连接路径，Unixware 中要确保/usr/ucb/lib 在/usr/ccs/lib 之后出现
TMPDIR=/tmp  ＃安装中 Oracle 会在此目录下存储相当数量的文件,所以 TMPDIR 所在的磁盘分区要确保空闲空间的大小，至少在 1G 左右
PATH=$PATH:$ORACLE_HOME/bin
export ORACLE_BASE ORACLE_HOME ORACLE_SID ORACLE_TERM ORA_NLS33 NLS_LANG NLS_DATE_FORMAT LD_LIBRARY_PATH TMPDIR
```

参见 profile。

注意：

NLS_LANG=American_America.ZHS16CGB231280(ZHS16GBK)

“American”指显示信息时所用的语言，窃以为凭大家的英语水平足够应付，如改为 SIMPLIFIED CHINESE，在不带中文支持的终端上就没人能看懂了。

“America”指地区

“ZHS16CGB231280”指 Client 工具使用的字符集，一般使用”ZHS16CGB231280”，Oracle8i 已支持到”ZHS16GBK”

NLS_DATE_FORMAT=YYYYMMDDHH24MISS

Oracle 的 date 类型过于灵活，为统一时间格式，利于编程，应将时间的输入输出格式限定为 14 位字符串，如”20000101235959”

据 oracle 文档，此参数可按照 session,操作系统用户环境，init.ora 由高到低的优先级顺序设置，依次覆盖。

相关系统表：

v$nls_parameters v$nls_valid_values

X-Window

Oracle 8 的安装程序是光盘 mount 点/bin/orainst，使用字符界面，不用考虑 X-Window。

Oracle 8i 和 9i 使用光盘 mount 点/runInstaller 进行安装，它是用 Java 编写的图形界面，对中文处理有问题，所以应在进入 X-Window 前确保语言（LANG）和地域（LC_ALL,LC_TYPE,…）环境变量不是中文。

LANG=C

LC_ALL=C

安装选项

Oracle 8i 的主要软件包在安装选项 Enterprise 中，但并不包括 proc，必须进行第二次安装，可选择安装选项 Client 中的 programmer。

Oracle 9i 的主要软件包在安装选项 Enterprise 安装选项中，但并不包括 proc，必须进行第二次安装，一定要选择安装选项 Client 中的 Administrator。runInstaller 的稳定性欠佳，建议每次安装结束后，先退出，再进行下一次安装。

1.2 UnixWare7

1.2.1 Oracle 8

确认操作系统的交换分区 swap 不少于 350M

认为该打的补丁统统打上，宁滥毋缺。UnixWare7.0.1 必须打的补丁为 ptf7033,ptf7051,ptf7052,ptf7068,ptf7096。

将/etc/default/login 中的 ulimit 设为大于 2113674（稍大一点即可，太大会有问题）

将/etc/conf/node.d/async 中的 600 改为 666

修改以下核心参数

核心参数	必需值	解释
SHMMAX	2147483647	共享内存段最大尺寸
SHMMNI	100	系统共享内存段标识最大数目
SHMSEG	15	每个进程所能使用最大共享内存段数目
SEMMNI	100	核心信号量标识最大数目

SEMMSL	150	每个信号量标识包含的信号量个数
SCORLIM	0X7FFFFFFF	Core 文件最大尺寸
HCORLIM	0X7FFFFFFF	
SDATLIM	0X7FFFFFFF	进程堆最大尺寸
HDATLIM	0X7FFFFFFF	
SVMMLIM	0X7FFFFFFF	进程最大映射地址
HVMMLIM	0X7FFFFFFF	
SFSZLIM	0X7FFFFFFF	进程文件最大偏移量
HFSZLIM	0X7FFFFFFF	
SFNOLIM	128	进程能打开的最大文件个数
HFNOLIM	2048	
NPROC	20+(8*MAXUSERS)	MAX:125000
ARG_MAX	1,048,576	
NPBUF	100	I/O 缓冲区数目
MAXUP	1000	用户同时使用的最大进程个数
STRTHRESH	0X500000	流能使用的最大字节数

为优化应用系统修改以下核心参数

核心参数	参考值	解释
MSGMAX	8192	消息最大尺寸
MSGMNB	81920	消息队列尺寸
MSGMNI	2048	系统能并存的最大消息队列数目
MSGSSZ	16384	
MSGTQL	4096	系统能并用的消息头数目
SEMMNI	1024	
SEMMSL	150	

也可通过编辑/etc/conf/cf.d/stune 达到同样效果

重新连接内核,重起或运行/etc/conf/bin/idbuild –B

修改核心参数 SEMMAP 时,注意要同时修改/etc/conf/mtune.d/ipc 中相应的 MAX 值

建立/var/opt/oracle,使 oracle 成为此目录属主

mount oracle 光盘,通常 mount 目录为/SD-CDROM_1

root 用户,ORACLE_OWNER=oracle, 执行光盘上 orainst 中 oratab.sh, 建立 /var/opt/oracle/oratab

安装时,选 custom 方式,安装时不建立数据库,字符集可选 Simplified Chinese

1.3 HP-UX

1.3.1 Oracle 8

流程大致与 unixware 相同，调整 kernel 参数可通过 sam，选择/Kernel Configuration/Actions/Apply Tuned Parameter Set/OLTP Database Server System，另外为提高 I/O 能力，还需调整以下参数：

核心参数	参考值	解释
bufpages	61992	缓冲页
dbc_max_pct	10	动态缓存占内存最大百分比
dbc_min_pct	10	动态缓存占内存最小百分比
nbuf		

设定共享库目录 SHLIB_PATH，不是 LD_LIBRARY_PATH

SHLIB_PATH=$SHLIB_PATH:$ORACLE_HOME/lib;export SHLIB_PATH

1.4 Linux

1.4.1 kernel 2.0 & glibc 2.0

代表产品为 Red Hat Linux 5.1。

Oracle 8 在 RedHat5.1 上能成功安装，安装软件包为 805ship.tgz

一般不会在 RedHat5.1 上安装 Oracle8i 以上的版本

修改共享内存最大尺寸限制：

在系统初始化脚本/etc/rc.d/rc.sysinit 中加入：

echo 2147483648 >/proc/sys/kernel/shmmax

重启计算机。这样做避免了 Oracle 分配的共享内存碎片化，对提高效率有好处。

原$ORACLE_HOME/precomp/admin/pcscfg.cfg 中 sys_include 有误，使 proc 预处理 pc 程序失败，安装结束后，应设为：sys_include=(/usr/include,/usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/include)（视 gcc 版本而定）

1.4.2 kernel 2.2 & glibc 2.1

代表产品为 Red Hat Linux 6.2。

修改共享内存最大尺寸限制：

在系统初始化脚本/etc/rc.d/rc.sysinit 中加入：

echo 2147483648 >/proc/sys/kernel/shmmax

重启计算机。这样做避免了 Oracle 分配的共享内存碎片化，对提高效率有好处。

原$ORACLE_HOME/precomp/admin/pcscfg.cfg 中 sys_include 有误，使 proc 预处理 pc 程序失败，安装结束后，应设为 sys_include=(/usr/include,/usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/include) （视 gcc 版本而定）

1.4.2.1 Oracle 8

本来已经很少有人在 LinuxKernel2.2 的系统中安装 Oracle8.0.5，但笔者实在怀念 8.0.5 纯粹的文本界面和与之相处的无数不眠之夜，故收录如下：

Oracle8 在 kernel 为 2.2.x 的 linux 中是无法正常运行的，运行可执行文件如 svrmgrl,sqlplus 时会导致“Segmentation fault”，原因在于这些 linux 使用了默认的 libc2.1，与 Oracle8 程序重连接所需的 libc2.0 不兼容。Oracle 的补丁程序其实是将 Oracle 可执行程序的重连接脚本中 libc 位置重新定位到 libc2.0 上去，并用旧版的 gcc，ld 重新连接可执行文件。为此必须先在系统中安装兼容库和相应工具。这是权宜之计，而且仅对 RedHat 有效。

root 用户

```
rpm –ivh tcl-8.0.3-20.i386.rpm  Oracle 的 Intelligent Agent 要使用
rpm –ivh compat-binutils-5.2-2.9.1.0.23.1.i386.rpm
rpm –ivh compat-glibc-5.2-2.0.7.1.i386.rpm
rpm –ivh compat-egcs-5.2-1.0.3a.1.i386.rpm
rpm –ivh compat-egcs-c++-5.2-1.0.3a.1.i386.rpm
rpm –ivh compat-libs-5.2-1.i386.rpm
```

版本号可略有差异

oracle 用户安装 Oracle8.0.5 但不创建 instance，如选择安装文档，则会产生如下错误：

A write error occurred while try to copy '/home/oracle/setup_oracle/unixdoc/server.805/install/lnx_server.805.map' to '/oracle/product/8.0.5/doc/server.805/install/lnx_server.805'(No such file or directory).

这是安装程序的一个 bug—不能创建目录。可进入$ORACLE_HOME/doc，mkdir –p server.805/install，再选择 Retry

从ftp.oracle.com/pub/www/otn/linux下载 glibcpatch.tgz，在某一目录（如~/patch）下展开

cd ~/patch

glibcpatch.sh

经过一段时间后，看到"Applied glibc patch for Oracle 8.0.5.x successfully"，表明补丁成功。此时就能成功创建 instance。

1.4.2.2 Oracle 8i

推荐使用典型安装，否则会产生难以预料的错误。

1.4.3 kernel 2.4 & glibc 2.2

代表产品为 Red Hat Linux 7.3,SuSE Linux 7.3。

1.4.3.1 Oracle 8i

与 Oracle8 在 RedHat Linux 6.2 上安装所遇到的问题一样，Oracle 8i 使用的 glibc 2.1 与操作系统自带的 glibc 2.2 不能兼容，解决的方法也一样，要安装 glibc 2.1 的兼容库，并重新连接 Oracle 各组件。

除非万不得已，不建议使用兼容方式，因此省略安装步骤，可参阅网上有关文档。

1.4.3.2 Oracle 9i

RedHat 7.3

修改共享内存最大尺寸限制：

在系统初始化脚本/etc/rc.d/rc.sysinit 中加入：

echo 2147483648 >/proc/sys/kernel/shmmax。

修改信号量参数：

在系统初始化脚本/etc/rc.d/rc.sysinit 中加入：

echo 250 32000 100 128 >/proc/sys/kernel/sem。

这 4 个参数依次为 SEMMSL(每个用户拥有信号量最大数量),SEMMNS(系统信号量最大数量),SEMOPM(每次 semop 系统调用操作数),SEMMNI(系统信号量集最大数量)，事实上只有 SEMOP 是需要调整的。

重启计算机。

在连接可执行文件过程中，会发生中断，打开$ORACLE_HOME/ctx/lib/env_ctx.mk，找到 INSO_LINK，在-L$(CTXLIB) -L$(LDLIBFLAG)m 后加入-L$(LDLIBFLAG)dl，重试。

SuSE 7.3

与 RedHat 类似，但 SuSE 没有/etc/rc.d/rc.sysinit，笔者选择/etc/rc.d/rc，将核心参数修改添加到最后 exit 语句之前。

安装过程中没有发生任何问题。

1.5 Solaris

Oracle 8i 在 Solaris 7,8 Intel Platform 上均能顺利安装，未测试 Solaris Sparc Platform。

Oracle 9i 目前无 Solaris Intel Platform 上的版本，由于条件所限，未测试在 Solaris Sparc Platform 上的 Oracle 9i。

修改下列核心参数：

核心参数	参考值	解释
shmmax	物理内存/2	共享内存段最大尺寸
shmmin	1	共享内存段最小尺寸
shmmni	100	系统共享内存段标识最大数目
shmseg	10	每个进程所能使用最大共享内存段数目
semmni	100	系统信号量标识最大数目
semmsl	init.ora.processes+10	每个信号量标识包含的信号量数目
semmns	sum(init.ora.processes)*10+max(init.ora.processes)+count(init.ora)*10	系统信号量最大数目
semopm	100	每个 semop 调用最大操作数目
rlim_fd_max	4096	系统文件句柄最大数目
rlim_fd_cur	1024	每个进程文件句柄最大数目

修改/etc/system，并重启使核心参数生效

例：

```
set shmsys:shminfo_shmmax=2147483648
set shmsys:shminfo_shmmin=1
set shmsys:shminfo_shmmni=100
set shmsys:shminfo_shmseg=10

set semsys:seminfo_semmni=200
set semsys:seminfo_semmsl=200
set semsys:seminfo_semmns=1000
set semsys:seminfo_semopm=100
set semsys:seminfo_semmap=200
set semsys:seminfo_semmnu=250
set semsys:seminfo_semvmx=32767

set msgsys:msginfo_msgmni=200
set msgsys:msginfo_msgmap=200
set msgsys:msginfo_msgmax=65536
set msgsys:msginfo_msgmnb=655360
set msgsys:msginfo_msgssz=64
set msgsys:msginfo_msgtql=1000
```

set msgsys:msginfo_msgseg=16384

set rlim_fd_max=4096
set rlim_fd_cur=1024

参见 solaris_7_8/system

注意：

一定要先重建好 kernel 后再安装，因为 oracle 安装时根据 kernel 动态连接程序，如果先安装 oracle，即使随后正确调整 kernel，也会带来许多问题，如 oracle 进程不能拉起，instance 创建失败等。

在 kernel 参数中，对数据库运行影响最大的主要是 SHMMAX,SEMMNS,SEMMNI,SEMMSL，SHMMAX 取内存一半即可，SEMMNS 理论上应等于 SEMMNI*SEMMSL，实际取一个较大值即可。

SEMMNS：信号量最大个数，有些系统可忽略，因为他与 SEMMNI,SEMMSL 有关。

2 创建

所有参见内容都在附件 01_install_02_create_03_init/下。

以 oracle 用户进行操作，设定数据库实例名为 oradb（长度建议不要超过 8 个字符）。

2.1 Oracle 8 & 8i

2.1.1 工具创建

Oracle 8

运行$ORACLE_HOME/bin/orainst（安装数据库时必须选中 oracle installer），选择 create database object，安装界面中选 Oracle Enterprise Server(RDBMS)

mount point 暂为$ORACLE_BASE，字符集为 ZHS16CGB231280 或 ZHS16GBK，调整 system,tools,users,rbs,temp,redolog 等尺寸。

创建过程中会提示输入 osdba,osoper 的 UNIX 组，这是向 instance 表明此组的成员享有角色 sysdba 或 sysoper 的权限，从而用 connect / as sysdba 替换掉 connect internal

Oracle 8i

进入 X WINDOW，运行 dbassist

2.1.2 手工创建

任何工具都有其局限性，熟练的数据库管理员可采用手工方法创建数据库，以增加对系统的灵活控制。

对于手工建库 Oracle 8 与 Oracle 8i 的区别主要是建立的数据字典和存储过程有些不同，Oracle8i 的 dbassistant 可以生成建库脚本供以后使用。

取得/8i/initoradb.ora，编辑如 db_name,control_file,dump_dest 等参数，以符合实际情况。如不需要生成 remote_login_passwordfile，可在 initoradb.ora 中设 remote_login_passwordfile=none；如需要，在 initoradb.ora 中设 remote_login_passwordfile=exclusive，运行 orapwd file=<file> password=<password>

必须创建新生成文件所要用到的目录，如在配置文件中指定的 bdump,cdump,udump 等目录，以及数据文件存储目录。

将 initoradb.ora 转移到 $ORACLE_BASE/admin/oradb/pfile/，并连接到 $ORACLE_HOME/dbs/initoradb.ora。

```
ln –s $ORACLE_BASE/admin/oradb/pfile/initoradb.ora
$ORACLE_HOME/dbs/initoradb.ora
```

取得 8i/createdb.sh，编辑如 pfile,数据文件目录等参数，以符合实际情况，并转移到$ORACLE_BASE/admin/oradb/create/下，执行。

相关系统表：

v$database
v$datafile(file#,ts#,name)
v$tablespace(ts#,name)

v$parameter（SQL>show parameter）
v$sga（SQL>show sga）

2.1.3 MTS（multi-threaded server）

Oracle8 使用两种配置模式：dedicated server（专用模式）和 shared server（即 multi-threaded server 共享模式），缺省使用专用模式。在连接数不很大且保持长期连接的情况下，专用模式为每个连接设立一个专用 oracle 服务进程，以保持较高的性能和稳定性。而当连接数上升到非常高的数目且不保持长期连接时，数据库管理开销增大，并且占用大量系统资源，给操作系统形成带来极大的压力。在这种情况下，共享模式更为有利，它通过缓冲池和预先设定数目的 server 提供服务，每个连接不再有专用的 oracle 服务进程，每次 SQL 操作由分配器（dispatcher）确定 oracle 服务进程。

multi-thread 仅表示分配器展开的多个服务流程，并非操作系统意义上的多线程

配置：

- initoradb.ora

  加入

  mts_dispatchers = “(address=(protocol=TCP))(dispatchers=10)” #初始分配器数量
  mts_max_dispatchers = 15　#最大分配器数量
  mts_servers = 50　#初始服务进程数量
  mts_max_servers = 80　#最大服务进程数量
  mts_service = oradb3　　#MTS 方式下对外提供的数据库服务，非 service_name

  表明 instance 能够提供 MTS 服务，不意味着取消 dedicated 方式

- listener.ora

  应删除所有 SID_LIST，SID_LIST 的存在决定 LISTENER 以 dedicated 还是 shared 方式启动 oracle 连接。如 SID_LIST 存在，LISTENER 不再接受 instance 的登记，以 dedicated 方式启动 oracle 连接； 如 SID_LIST 不存在，LISTENER 启动时不为任何 instance 服务，由 instance 来登记 MTS service，以 shared 方式启动 oracle 连接

- client

  MTS 在 client 端配置颇为怪诞，在 tnsnames.ora 中的 host 一定要写数据库 server 的名字，而且必须作全名解析，似乎 server 端接收到 client 端请求后会将主机字符串返回，应此 client 端必须能够解析，否则会报出诸如“ database service not exist ”的错误

  tnsnames.ora

```
dbserver.soar.com =
(DESCRIPTION =
        (ADDRESS_LIST =
                (ADDRESS=(PROTOCOL=TCP)(HOST= dbserver)(PORT = 1521))
        )
        (CONNECT_DATA =(SERVICE_NAME = oradb))
)
```

  /etc/hosts

```
10.0.0.1    dbserver.soar.com dbserver
```

启动：先起 LISTENER，后起 instance

以下步骤均在数据库 open 状态下，由 system 用户完成

2.1.4 调整临时表空间

alter tablespace temp temporary; #Oracle8 的 orainst 没有将 temp 的缺省值 permanent 改为 temporary，这样用户在 temp 上暂存的数据均为永久对象，很快将 temp 空间耗完。Oracle8i 已修正。

SQL>alter tablespace temp default storage (initial 128k next 128k maxextents 5000 pctincrease 0)；

SQL 查询操作如 group by,order by,distinct，join 等需要在临时段上展开数据，须充分考虑临时段的大小。

如果实例启动参数指定 hash_join_enabled=true（缺省为 true），当 oracle 选择以 hash join 方式进行表与表的联接，oracle 根据查询操作的实际情况计算出 hash_multiblock_io_count，此参数从属于 session，平时显示为 0，即 hash join 一次 I/O 读写需要的连续数据空间。这样当此参数大于临时段的 next 扩展块时，hash join 操作会中断。如果预知联接表的规模比较巨大，可使用 alter tablespace temp default storage(next …)将 next 值设为较大值，待全部操作完成后，再恢复正常。

2.1.5 调整回滚表空间

先将建库工具缺省设定的若干个回滚段删除

SQL>alter rollback segment r01 offline；

SQL>drop rollback segment r01；

根据实际需要创建回滚段（如 r01-r10），供联机处理和批处理使用

SQL>create rollback segment r01 storage（initial 128k next 128k maxextents 5000 optimal 5M） tablespace rbs；

SQL>alter rollback segment r01 online；

注意修改$ORACLE_HOME/dbs/initoradb.ora 中的激活回滚段段名

另创建一个尺寸无限制的回滚段（r99），供特殊用途

SQL>create rollback segment r99 storage（initial 128k next 128k maxextents 5000） tablespace rbs；

如果在创建回滚段时使用 create ***public*** rollback segment，则不需要在 $ORACLE_HOME/dbs/initoradb.ora 中用 rollback_segment=(…)选项激活，推荐使用 public 方式

相关系统表：

SQL>select segment_name, initial_extent, next_extent, max_extents, extents,bytes from ***dba_segments*** where segment_type='ROLLBACK'; #回滚段占用空间状况

SQL>select segment_name, status from ***dba_rollback_segs***; #回滚段状态

2.1.6 调整日志

建立日志组

SQL>alter database add logfile group x('log1a','log1b') size 10M;

增加日志组成员

SQL>alter database add logfile member 'log1c' to group x；

删除日志

数据库实例至少需要 2 个日志组，只有状态为 inactive 的日志组才能被删除，而当前日志组状态为 current，上一个切换的日志组状态为 active，这就意味着至少存在 3 个日志组才能删除其中的一个，如果要更新全部日志组，只能删除一个，再创建一个，直至全部被更新。

SQL>alter database drop logfile group x；

如果要删除的日志组是当前日志组，必须先将其切换至状态为 inactive，再删除。

SQL>alter system switch logfile；

删除日志组成员

SQL>alter database drop logfile member 'log1c';

相关系统表

v$log ＃日志组状态、占用空间、顺序号等

v$logfile ＃日志组文件

2.1.7 调整用户表空间

创建表空间

假定表数据在 ts_data，索引在 ts_index

SQL>create tablespace ts_data default storage（initial 10M next 10M maxextents 5000 pctincrease 0） datafile 'path/data_01.dbf' size 500M;

SQL>create tablespace ts_index default storage（initial 5M next 5M maxextents 5000 pctincrease 0） datafile 'path/index_01.dbf' size 500M;

参考命令：删除表空间

SQL>drop tablespace data including contents; ＃删除表空间及其包含的所有数据对象

相关系统表：

user(dba)_tablespaces

增加表空间尺寸

假定表空间 ts_data 由 path/data_01.dbf 和 path/data_02.dbf（500M）组成

增加一个数据文件：

SQL>alter tablespace ts_data add datafile 'path/data_03.dbf' size 500M;

扩大原有文件大小：

SQL>alter database datafile 'path/data_01.dbf' resize 1000M;

移动表空间数据文件

假如要求为：将 path1 下 data_01.dbf 移至 path2 下，并把文件名改为 data01.dbf

实例处于关闭状态

sqlplus "/ as sysdba"

SQL>startup mount

回到 shell 环境下

$ mv path1/data_01.dbf path2/data01.dbf

$ mv path1/data_02.dbf path2/data02.dbf

再到 sqlplus 环境中

SQL>alter database rename file ‘path1/data_01.dbf’ to ‘path2/data01.dbf’;

或

SQL>alter tablespace tbsdata rename datafile ‘path/data_01.dbf’ to ‘path2/data01.dbf’;

SQL>alter database open;

查看剩余空间

SQL>select tablespace_name,sum(bytes),max(bytes) from dba_free_space group by tablespace_name;

注意:空闲数据块总和 sum(bytes)够用并不意味每个空闲块都满足分配需要，所以当表空间不够分配扩展块的时候，还要查看最大空闲数据块 max(bytes)的大小。

合并空闲块

如果表空间上的数据对象经常发生类似 drop-create 的变动，加之未采用统一的扩展块尺寸，使那些采用较大扩展块的数据对象不能利用较小的空间碎片，造成空间浪费。可通过将较小的空闲块合并成较大的空闲块的方法，减少空间浪费。

SQL>alter tablespace tbsdata coalesce;

2.1.8 创建用户

SQL>create user dbuser identified by oracle default tablespace data temporary tablespace temp quota unlimited on data quota 0 on system quota 0 on tools quota 0 on users；

SQL>grant connect to dbuser；

SQL>grant create procedure to dbuser； ＃这些权限足够用于开发及生产环境

SQL>grant select on dba_pending_transactions to dbuser; ＃二阶段提交过程中类似 Tuxedo 的软件需要检索挂起交易的状态，所以必须得到对此视图的 select 权限，以 sys 用户身份赋予

修改用户可使用 alter user dbuser ...

参考命令：

drop user dbuser cascade; ＃删除用户及其所有的数据对象

revoke connect from dbuser; ＃取消用户角色权限

相关系统表：

user(dba)_users

user(dba)_role_privs 角色权限

user(dba)_sys_privs 系统权限

user(dba)_tab_privs 对其他用户表操作的权限

user_ts_quotas 表空间限额

2.1.9 创建数据对象

相关系统表：

user_catalog(cat)

user_objects(obj)

表和索引建立在表空间上，如果不指定表空间，使用本用户的缺省表空间（default tablespace）；如果不指定本对象的存储参数，使用建于其上的表空间的缺省存储参数（default storage）。

表（table）

建表脚本通常是以下形式：

create table emp （no number(12), name char(20), …,constraint emp_x00 primary key(no)） storage（initial 100M next 100M pctincrease 0 maxextents 5000） pctused 70 pctfree 10 tablespace tbs_data enable primary key using index tablespace tbs_index;

然而从简化数据对象配置、减少表空间碎片的角度考虑，不推荐为每张表单独指定 storage 选项，存储参数使用建于其上的表空间的缺省存储参数。不同表对扩展块大小的要求，可以通过分析归类，建立相应具有不同缺省存储参数的表空间的方法解决。这样数据库设计就能变得简洁明了。

命令简化为：

create table emp （no number(12), name char(20), …, constraint emp_x00 primary key(no)）pctused 70 pctfree 10 tablespace tbs_data enable primary key using index tablespace tbs_index;

primary key 关键字建立同名的 primary key constraint 和 unique index，表的每个域都有自身的 constraint。

相关系统表：

`user_tables(tabs)，dba_tables` #表属性

`user_tab_columns(cols),dba_tab_columns` #表各列属性

索引（index）

create index emp_x01 on emp(name) storage(initial 10M next 10M pctincrease 0 maxextents 5000) pctfree 10 tablespace tbs_index;

可参照表对 storage 的处理方式。

create index emp_x01 on emp(name) pctfree 10 tablespace tbs_index;

相关系统表：

`user_indexes(ind),dba_indexes` #索引属性

`user_ind_columns,dba_ind_columns` #索引各列属性，以 `index_position` 为顺序

序列（sequence）

create sequence emp_seq increment by 1 start with 1 nomaxvalue nocycle;

相关系统表：

user(dba)_sequences(seq) 序列属性

视图（view）

create emp_depart_view as select emp.name,emp_duty.name from emp,emp_duty where emp.duty=emp_duty.duty;

相关系统表：

user(dba)_views 视图属性

`Oracle` 将 `view,sequence`，用户参数等定义均存放于系统表空间，而用户创建的表空间仅存放 `table,index` 实体，因此可以大胆删除用户表空间，再用备份重新恢复，不必担心 `view,sequence` 等会被一并删去。

2.1.10 创建只读用户

假定数据库用户 dbbrsr 需要对 dbuser 的表 emp 拥有 select 权力

connect dbuser
grant select on emp to dbbrsr
connect dbbrsr
create synonym emp for dbuser.emp;

这样，dbbrsr 就能象使用自己的表一样对 dbuser 的表执行 select 操作

2.1.11 启动及关闭数据库实例

oracle 用户，dbstart 和 dbshut 启动及关闭/var/opt/oracle/oratab 或/etc/oratab 中设定的数据库实例，dbstart 采用 normal 方式，dbshut 采用 immediate 方式。

或者使用手工方式

sqlplus "/ as sysdba"

启动

normal

SQL>startup

mount

SQL>startup mount; #启动实例进程,载入数据库文件,允许 DBA 权限的某些操作，但禁止对数据库文件的一般性操作

SQL>完成某些操作

SQL>alter database open;

nomount

SQL>startup nomount; #启动实例进程,但不允许访问数据库,常用于创建数据库、介质恢复或创建 controlfile

SQL>完成某些操作

SQL>alter database open;

关闭

normal

SQL>shutdown 或 SQL>shutdown transactional; #等待每个连接交易完成后，切断连接，再关闭数据库

immediate

SQL>shutdown immediate; #立刻中止每个连接，交易回滚

abort

SQL>shutdown abort; #立刻关闭数据库，不保证交易完整性，在下一次启动打开数据库文件时会进行介质恢复

2.1.12 网络配置

假定某一台机器为 client，ORACLE_SID 为 oraclient，数据库用户为 dbclient；另一台机器为 server，ORACLE_SID 为 oraserver，数据库用户为 dbserver 在 server 上 $ORACLE_HOME/dbs/initoraserver.ora 中有以下设定：

db_name = oraserver
instance_name = oraserver

Oracle 8i

service_names=oraserver

2.1.12.1TNS

Client 端配置

修改$ORACLE_HOME/network/admin/tnsnames.ora，增加一条 PROTOCOL=TCP 的记录。

Oracle8

db_server

(DESCRIPTION =

(ADDRESS = (PROTOCOL= TCP)**(Host= server**)(Port= 1521))

(CONNECT_DATA = (**SID** = **oraserver**))

)

Oracle8i

db_server

(DESCRIPTION =

(ADDRESS = (PROTOCOL= TCP)(**HOST= server**)(PORT= 1521))

(CONNECT_DATA = (**SERVICE_NAME**=**oraserver**

)

HOST 可在/etc/hosts 或 DNS 中配置，或直接写上 IP 地址

sqlplus dbserver/passwd@db_server

Server 端配置

修改$ORACLE_HOME/network/admin/listener.ora

在 LISTENER 中增加 ADDRESS 的记录

LISTENER =

(DESCRIPTION_LIST =

(DESCRIPTION =

(ADDRESS_LIST =

(ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC0))

(ADDRESS = (PROTOCOL = TCP)(**HOST = server**)(PORT = 1521))

)

)

)

在 SID_LIST_LISTENER 中增加 SID_DESC 记录

SID_LIST_LISTENER =

(SID_LIST =

(SID_DESC =

(SID_NAME = PLSExtProc)

(ORACLE_HOME = /opt/oracle/app/oracle/product/8.1.5)

(PROGRAM = extproc)

)

(SID_DESC =

(GLOBAL_DBNAME = oraserver)

(ORACLE_HOME = /opt/oracle/app/oracle/product/8.1.7)

(SID_NAME = oraserver)

)

)

HOST 可在/etc/hosts 或 DNS 中配置，或直接写上 IP 地址

注意：LISTENER 和 SID_LIST_LISTENER 是成对出现的，可配置多个监听服务进程和相应的 SID_LIST，如 LISTENER_1 和 SID_LIST_LISTENER_1

2.1.12.2DB Link

如果 client 的 instance 需要在访问本地数据对象同时访问 server 中的数据对象，可在 instance 中创建对 server 的数据库连接，实现间接访问

在 tnsnames.ora 中建立“ db_server ”配置

sqlplus dbclient/passwd1

SQL>create database link **server_link** connect to dbserver identified by passwd using '**db_server**';

使用 emp@server_link 访问 server 上的 emp，如同访问本地 instance 中的数据对象一样。为了更方便的使用，可建立 synonym

2.2 Oracle 9i

Oracle 9i 相较于 Oracle 8&8i，在兼容 Oracle 8&8i 的基础上，回滚和临时表空间配置发生比较大的变化，导致建库操作出现一些不同。在数据库配置文件 initoradb.ora 中有关于回滚表空间的选项，详细情况在“ 数据库配置 ”中解释。而且 Oracle 9i 简化了表空间的创建。所以此小节主要描述Oracle 9i相对于Oracle 8i的差异，其它相同的操作可参考 Oracle 8i。

2.2.1 手工创建

由于在 Oracle 9i 中工具 dbassist 的使用方法与在 Oracle 8i 中类似，因此工具建库过程省略，只记录手工建库过程

Oracle 9i 中的建库过程已经变得极为简洁，大致如下：

```
create database ${ORACLE_SID}
user sys identified by sys
user system identified by system
logfile group 1 ('${ORACLE_BASE}/oradata/${ORACLE_SID}/redo01.log') size 10M,
group 2 ('${ORACLE_BASE}/oradata/${ORACLE_SID}/redo02.log') size 10M,
group 3 ('${ORACLE_BASE}/oradata/${ORACLE_SID}/redo03.log') size 10M
maxlogfiles 5
maxlogmembers 5
maxloghistory 1
maxdatafiles 254
maxinstances 1
archivelog
character set ZHS32GB18030
national character set AL16UTF16
datafile '${ORACLE_BASE}/oradata/${ORACLE_SID}/system01.dbf' size 300M
default temporary tablespace tbstemp tempfile '${ORACLE_BASE}/oradata/${ORACLE_SID}/temp01.dbf' size 500M
```

undo tablespace tbsundo datafile '${ORACLE_BASE}/oradata/${ORACLE_SID}/undo01.dbf' size 500M;

其特点为使用专用的回滚和临时表空间，而不象 Oracle 8i 中的那样，回滚和临时表空间与普通表空间没有差异，这样既简化了配置也有利于效能提高。要注意临时表空间的指定文件关键字是 tempfile 而不是通用的 datafile，而且临时表空间的存储选项必须为 uniform，由 Oracle 系统决定。同样回滚表空间也是由 Oracle 系统决定。不必人工干预。

Oracle 9i 在$ORACLE_HOME/dbs 下可使用二进制配置文件，缺省为 spfile{实例名}.ora，如 spfileoradb.ora，支持 Oracle 系统进程在不重启的情况下动态调整参数，这对要求不间断运行的系统是有利的。在建库阶段就可将此配置文件创建起来。

create spfile from pfile=
'${ORACLE_BASE}/admin/${ORACLE_SID}/init${ORACLE_SID}.ora

完整步骤见/9i/createdb.sh，编辑如 pfile,数据文件目录等参数，以符合实际情况，并转移到$ORACLE_BASE/admin/oradb/create/下，执行。

2.2.2 创建用户表空间

Oracle 9i 对于表空间管理一个明显的变化是改**数据字典管理（extent management dictionary）**为**表空间本地管理（extent management local）**，还可以根据建立的数据对象对空间的要求自动确定扩展块的大小**（autoallocate）**，最小为 64K，这两项都是创建表空间的缺省选项。

create tablespace tbsdata datafile '…' **[extent management local] [autoallocate]**;

而对于指定每个扩展块大小的创建策略，设立了新选项：**统一扩展块大小（uniform [size xxx[K|M]]）**，可覆盖 autoallocate 选项，如果不加上具体的 size xxx[K|M]，缺省为 1M，这样就不必考虑 Oracle 8i 中的如 initial,next,pctincrease,maxextents 等 default storage 参数应如何组合，事实上 Oracle 8i 的这些设置原本就没有什么意义。

不能够同时指定 extent management local 和 default storage，换言之，default storage 只能和 extent management dictionary 一起显式指定。

如果未指定 extent management 的类型，Oracle 9i 缺省使用 local 方式，如果又同时使用 default storage 选项，就有以下的判断：

如果使用 minimun extent，Oracle 检查是否 minumum extent=initial=next 且 pctincrease＝0,如是，Oracle 使用 uniform 选项，size=initial；如不是，Oracle 忽略指定选项，使用 autoallocate。

如果未指定 minimum extent，Oracle 检查是否 initial=next 且 pctincrease＝0，如是 Oracle 使用 uniform 选项，size=initial；如不是 Oracle 忽略指定选项，使用 autoallocate。

为了避免与 Oracle 8i 的习惯做法混淆，建议只使用 Oracle 9i 较简洁的方法。

对于存储少量静态数据的表空间来说，如配置信息等，可简单地写为：

create tablespace tbsdata datafile '…';

对于必须关心其扩展块大小的表空间，如大批量的记录或索引，可简单地写为：

create tablespace tbsdata datafile '…' uniform size 10M;

3 初始化文件配置

所有参见内容都在附件 01_install_02_create_03_init/下。

描述 initoradb.ora 中各选项。

3.1 Oracle 8 & 8i

具体参见 8i/initoradb.ora。

db_block_size

数据库基本数据块尺寸，字节为单位。

当涉及到大量数据交换时，例如 export/import 操作时，此参数对数据库性能有非常大的影响，设定一个较大的值，有利于提高数据吞吐量，但由于 db block 是文件和内存之间交换的基本单位，过大的值反而会交换不需要的记录，增加额外的 I/O。

一般取 8k 就已能获得较满意效果。

db_block_buffers

数据缓冲区，db_block_size 为单位，不超过 1/4 内存

计算查询缓冲命中率：

SELECT name, value FROM v$sysstat WHERE name IN ('db block gets', 'consistent gets', 'physical reads');

Hit Ratio = 1 – (physical reads / (db block gets + consistent gets))

SELECT name, phyrds, phywrts FROM v$datafile df, v$filestat fs WHERE df.file# = fs.file#

db block gets:在内存 buffer 中的命中次数

consistent gets:一致性命中次数，指在内存 buffer 中未命中，但从回滚段或数据文件中获得命中

physical reads:在数据文件中的读次数

注意：一般 HitRatio 达到 90%以上就可以认为已达到优化，这个数值应在系统运行稳定后进行统计。

shared_pool_size

数据字典和 SQL 操作缓冲区，字节为单位，不超过 1/4 内存

select (sum(pins – reloads)) / sum(pins) "Lib Cache" from v$librarycache;

select (sum(gets – getmisses – usage – fixed)) / sum(gets) "Row Cache" from v$rowcache;

select * from v$sgastat where name = 'free memory'

注意：Cache 命中率达到 95%以上就可以认为已达到优化，这个数值应在系统运行稳定后进行统计

log_checkpoint_interval

日志提交点数据量间隔

以操作系统 block(通常 512-byte)为单位，当日志累计至此参数，会使 sga 中 dirty buffer 被同步至数据文件，日志切换时也会引起此操作，如设为 0，则相当于无限大，此参数失去作用，日志提交仅依靠日志文件的切换。

应选择适当大小的日志文件，同时使 log_checkpoint_interval 略大于日志文件或设为 0。

原则上应该避免过于频繁的 checkpoint 操作，控制在 30 分钟以上为好。

推荐此参数设为 0。

log_buffer

在线日志缓冲，字节为单位，512K 或 128K*CPU 数量，取较大值

processes 和 sessions

dedicated server 模式下每一个连接都有一个 Oracle 服务进程（process）为之服务，这个连接本身也就是一个会话（session）。

shared server 模式下所有连接共享一个 Oracle 服务进程池，这样 process 和 session 就不再是一一对应，sessions 要大于 processes。

sort_area_size 和 sort_area_retained_size

排序缓冲区，字节为单位。

当排序记录被全部取走后，缓冲区缩减到 sort_area_retained_size，为减少缓冲区缩放的开销，可使 sort_area_size 和 sort_area_retained_size 取相同值。

hash_area_size

hash join 缓冲区，字节为单位，缺省为 2*sort_area_size。

db_file_multiblock_read_count

每次读取的 db block 数，对大规模查询性能有提高，特别是表扫描效率。在线系统应避免这种类型的查询。

db_writer_processes

同步数据进程数，与 checkpoint 的频率和数据量有关。

db_block_lru_latches

LRU 锁集，一般设为 CPU 数目。RedHat Linux 6.x 下的 Oracle 8.1.6 设此参数会导致系统挂起，疑对 smp 支持有问题

log_archive_start

系统启动时是否同时启动归档进程（archive）。

log_archive_dest_1

归档日志目录，最后的标号表明归档线程编号，一般只用 1。

log_archive_format

归档日志名称，%t 指归档线程编号， %s 指归档日志序列号

rollback_segments

如果创建回滚段（rollback segment）时不使用 public 选项，那就是使用私有的回滚段，这样就必须在系统启动时激活。

推荐使用 public rollback segment 的做法，这个选项可以废弃。

background_dump_dest

Oracle 系统进程记录 log 和 trc 目录。

alert_{实例名}.log 以文本方式记录系统启动、关闭、出错、存储变化、日志切换等 log 信息。

系统进程以各自名称和进程号记录错误信息，文件以 trc 为后缀，文本格式。

core_dump_dest

Oracle 服务进程的 core dump 目录。

user_dump_dest

Oracle 服务进程以各自名称和进程号记录错误信息，文件以 trc 为后缀，文本格式。

3.2 Oracle 9i

参见 9i/initoradb.ora。

pga_aggregate_target

以 K、M、G 为单位

sort, group-by, hash-join, bitmap merge, bitmap create 等对内存有一定需求的 SQL 操作，都由此选项统一动态分配内存区域大小，因此 Oracle 8i 中如 sort_area_size,sort_area_retained_size,hash_area_size，bitmap_merge_area_size 等选项可以废弃。

db_cache_size

数据缓冲区，以 K、M、G 为单位，自动对齐到粒度单位。

取代 Oracle 8i 的 db_block_buffers 选项。

undo_management

回滚空间管理模式，缺省为 manual，使用回滚段（rollback segment），如设为 auto，则使用 Oracle 9i 的回滚表空间。此选项决定了以下关于 undo 的其它选项。

undo_retention

已提交数据在回滚表空间中保留时间，以秒为单位，缺省 900。

当某些较长时间的查询需要通过回滚数据重建老数据块的时候，此选项可使新事务尽可能使用空闲的回滚表空间，这样就减少了查询过程因 snapshot too old 而失败的几率。

然而当空闲回滚表空间不足以应付新事务时，系统仍然会重用此选项保留的空间，因此不能保证长查询一定能成功执行完毕。

undo_tablespace

指定系统启动时的回滚表空间。

4 工具

所有参见内容都在附件 04_tool/下。
在《优化》一节中讨论以下工具使用的效率。

4.1 sqlldr

参见 sqlldr/。
用于将格式化的文本数据上载到表中去
以表 emp 为例
首先编写一个控制命令的脚本文件，通常以 ctl 结尾，内容如下：

emp.ctl

```
load data
append
into table emp
fields terminated by '|'
(
    no          float external,
    name        char(20),
    age         integer external,
    duty        char(1),
    salary      float external,
    upd_ts      date(14) 'YYYYMMDDHH24MISS'
)
```

括号里对数据文件里每个数据域进行解释，以此在上载时与目标表进行比对。
除了 append 外，还有 insert、replace、truncate 等方式，与 append 大同小异，不作更多的解释。

再将上载数据组织成数据文件，通常以 dat 结尾，内容如下：

emp.dat

```
100000000001|Tom|000020|1|000000005000|20020101000000
100000000002|Jerry|000025|2|000000008000|20020101235959
```

分隔符要与 ctl 文件中 fields terminated by 指定的一致，这个例子中为“|”
ctl 和 dat 文件就绪后可以执行上载，命令为：

sqlldr dbuser/oracle control=emp.ctl data=emp.dat

也可以将 dat 文件合并在 ctl 文件中，ctl 文件改写为：

emp2.ctl

```
load data
infile *
append
into table emp
fields terminated by '|'
(
    no          float external,
    name        char(20),
```

```
    age         integer external,
    duty        char(1),
    salary      float external,
    upd_ts      date(14) 'YYYYMMDDHH24MISS'
)
begindata
100000000003|Mulder|000020|1|000000005000|20020101000000
100000000004|Scully|000025|2|000000008000|20020101235959
```

控制文件中 infile 选项跟 sqlldr 命令行中 data 选项含义相同，如使用 infile * 则表明数据在本控制文件以 begin data 开头的区域内。

这样命令变成：

sqlldr dbuser/oracle control=emp2.ctl

conventional path

通过常规通道方式上载。

rows：每次提交的记录数

bindsize：每次提交记录的缓冲区

readsize：与 bindsize 成对使用，其中较小者会自动调整到较大者

sqlldr 先计算单条记录长度，乘以 rows，如小于 bindsize，不会试图扩张 rows 以填充 bindsize；如超出，则以 bindsize 为准。

命令为：

sqlldr dbuser/oracle control=emp.ctl log=emp.log rows=10000 bindsize=8192000

direct path

通过直通方式上载，不进行 SQL 解析。

命令为：

sqlldr dbuser/oracle control=emp.ctl log=emp.log direct=true

4.2 exp

参见 dmp/exp_demo.sh。

将数据库内的各对象以二进制方式下载成 dmp 文件，方便数据迁移。

buffer：下载数据缓冲区，以字节为单位，缺省依赖操作系统

consistent：下载期间所涉及的数据保持 read only，缺省为 n

direct：使用直通方式，缺省为 n

feeback：显示处理记录条数，缺省为 0，即不显示

file：输出文件，缺省为 expdat.dmp

filesize：输出文件大小，缺省为操作系统最大值

indexes：是否下载索引，缺省为 n，这是指索引的定义而非数据，exp 不下载索引数据

log：log 文件，缺省为无，在标准输出显示

owner：指明下载的用户名

query：选择记录的一个子集

rows：是否下载表记录

tables：输出的表名列表

下载整个实例

exp dbuser/oracle file=oradb.dmp log=oradb.log full=y consistent=y direct=y

user 应具有 dba 权限

下载某个用户所有对象

exp dbuser/oracle file=dbuser.dmp log=dbuser.log owner=dbuser buffer=4096000 feedback=10000

下载一张或几张表

exp dbuser/oracle file=dbuser.dmp log=dbuser.log tables=table1,table2 buffer=4096000 feedback=10000

下载某张表的部分数据

exp dbuser/oracle file=dbuser.dmp log=dbuser.log tables=table1 buffer=4096000 feedback=10000 query=\"where col1=\'…\' and col2 \<…\"

不可用于嵌套表

以多个固定大小文件方式下载某张表

exp dbuser/oracle file=1.dmp,2.dmp,3.dmp,… filesize=1000m tables=emp buffer=4096000 feedback=10000

这种做法通常用在：表数据量较大，单个 dump 文件可能会超出文件系统的限制

直通路径方式

direct=y，取代 buffer 选项，query 选项不可用

有利于提高下载速度

consistent 选项

自 export 启动后，consistent=y 冻结来自其它会话的对 export 操作的数据对象的更新，这样可以保证 dump 结果的一致性。但这个过程不能太长，以免回滚段和联机日志消耗完

4.3 imp

参见 dmp/imp_demo.sh。

将 exp 下载的 dmp 文件上载到数据库内。

buffer：上载数据缓冲区，以字节为单位，缺省依赖操作系统

commit：上载数据缓冲区中的记录上载后是否执行提交

feeback：显示处理记录条数，缺省为 0，即不显示

file：输入文件，缺省为 expdat.dmp

filesize：输入文件大小，缺省为操作系统最大值

fromuser：指明来源用户方

ignore：是否忽略对象创建错误，缺省为 n，在上载前对象已被建立往往是一个正常现象，所以此选项建议设为 y

indexes：是否上载索引，缺省为 n，这是指索引的定义而非数据，如果上载时索引已建立，此选项即使为 n 也无效，imp 自动更新索引数据

log：log 文件，缺省为无，在标准输出显示

rows：是否上载表记录

tables：输入的表名列表

touser：指明目的用户方

上载整个实例

imp dbuser/oracle file=oradb.dmp log=oradb.log full=y buffer=4096000 commit=y ignore=y feedback=10000

上载某个用户所有对象

imp dbuser/oracle file=dbuser.dmp log=dbuser.log fromuser=dbuser touser=dbuser2 buffer=2048000 commit=y ignore=y feedback=10000

上载一张或几张表

imp dbuser2/oracle file=user.dmp log=user.log tables=table1,table2 fromuser=dbuser touser=dbuser2 buffer=2048000 commit=y ignore=y feedback=10000

以多个固定大小文件方式上载某张表

imp dbuser/oracle file=\(1.dmp,2.dmp,3.dmp,…\) filesize=1000m tables=emp fromuser=dbuser touser=dbuser2 buffer=4096000 commit=y ignore=y feedback=10000

4.4 sqlplus

参见 sqlplus/download.sh。
仅列出常用的选项，对复杂的应用不作深究

4.4.1 命令行参数

/ as {sysdba|sysopr}：使用操作系统用户验证，以 osdba 或 osopr 一员的身份登录，如验证通过，被赋予 sysdba 或 sysopr 的权限

使用格式：`sqlplus "/ as sysdba"`

/nolog：不执行 connect 操作，直接进入 sqlplus 操作界面

-s：silent 模式，不显示 sqlplus 启动信息和提示符

<：接受 sql 脚本从标准输入重定向

<<：立即文档

4.4.2 提示符命令

accept *variable* [number|char|date] [format *format*] [default *default*] [prompt *text*] [hide]：接受输入变量

例子：`accept pwd char format a8 prompt 'Password:' hide`

column *column* [format *format*] [heading *heading*]：设定对某个域的显示格式

如果要同时改变某域的输出长度和标题，必须使用 column 命令

见 emp 的定义，name 本为 char(20)，输出缩为 10 位，duty 本为 char(1)，扩张为 6 位，以便有足够的空间显示中文标题。

```
SQL>column name format a10 heading '姓名';
SQL>column duty format a6 heading '职位';
SQL>column age format 999999 heading '年龄';
SQL>column upd_ts format a14 heading '更新时间';
SQL>select name,duty,age,upd_ts from emp;
```

show *option*：显示 SET 的选项

spool [*filename*|off]：输出重定向文件

timing [start *text*|show|stop]：定时器

4.4.3 SET 选项

autocommit：自动提交 insert、update、delete 带来的记录改变，缺省为 off

colsep：域输出分隔符

define：识别命令中的变量前缀符，缺省为 on，也就是'&'，碰到变量前缀符，后面的字符串作为变量处理

如果待更新内容包含'&'（在 URL 中很常见），而 define 非设为 off，sqlplus 会把'&'后面紧跟的字符串当成变量，提示输入，这里必须重新输入'&'和那个字符串，才能实现正常更新。将 define 设为 off，就不再进行变量判断。

```
SQL>set define off;
SQL>update bbs_forum set url='http://www.xxx.com/bbs/show.php&forum_id=1' where forum_id=1;
```

echo：显示 start 启动的脚本中的每个 sql 命令，缺省为 on

feedback：回显本次 sql 命令处理的记录条数，缺省为 on

heading：输出域标题，缺省为 on

linesize：输出一行字符个数，缺省为 80

如果一行输出超过 linesize，会回车到第二行，这样格式就会混乱。

markup html：html 格式输出，缺省为 off

通常需要与 spool 配合，否则 html 输出就没有意义。

numwidth：输出 number 类型域长度，缺省为 10

长 number 类型的域常常因为输出长度的问题，引起误会。

pagesize：输出每页行数，缺省为 24

为了避免分页，可设定为 0。

termout：显示脚本中的命令的执行结果，缺省为 on

timing：显示每条 sql 命令的耗时，缺省为 off

trimout：去除标准输出每行的拖尾空格，缺省为 off

trimspool：去除重定向（spool）输出每行的拖尾空格，缺省为 off

4.4.4 例子

以文本形式下载表数据

oracle 缺乏将表中数据输出至文本文件的工具，因此只能利用 sqlplus 和 unix 工具做变通的处理

```
sqlplus –s dbuser/oracle <<EOF >/dev/null
set colsep |;
set echo off;
set feedback off;
set heading off;
set pagesize 0;
set linesize 1000;
set numwidth 12;
set termout off;
set trimout on;
set trimspool on;
```

```
spool tmp.txt;
select * from emp;
spool off;
exit
EOF
tr –d ‘ ‘ < tmp.txt >emp.txt
```

删除空格，可选

注意：一定要用 spool，如果在命令行中直接用>tmp.txt 可能会造成数据缺失，至少在 Unixware7 上如此

假定某域是 char(n)，如中间出现回车\n，则下载出的这条记录的格式将会错乱，不宜采用此方法

5 备份及恢复

所有参见内容都在附件 05_backup/下。

5.1 export 与 import 方式

参见 dmp/backup.sh。

见《工具》对 exp 和 imp 的描述

数据库中的对象是比较多的，但除了表以外占用的空间不大，所以当表中记录数量达到一定规模后，以用户的方式一下子把数据 exp 出来就显得不够灵活。考虑以下的策略，先 exp 出除表数据以外的所有对象，再分别 exp 出每张表的数据。

exp dbuser 所有的数据对象

```
exp dbuser/oracle file=dbuser.dmp log=user.log owner=user buffer=2048000 rows=n
```

exp 单张表的数据

```
sqlplus -s dbuser/oracle <<EOF >/dev/null
set colsep |;
set echo off;
set feedback off;
set heading off;
set pagesize 0;
set linesize 1000;
set termout off;
set trimout on;
set trimspool on;
spool tables.txt;
select table_name from user_tables;
spool off;
exit;
EOF
for table in $(cat tables.txt)
do
    exp dbuser/oracle file=${table}_$(date '+%Y%m%d').dmp tables=$table direct=y
done
```

5.2 冷备份

shutdown 数据库，将所有和本实例有关的文件，包括 datafile,controlfile,redolog,archived redolog,initora.ora 等全部备份。恢复时只要将这些文件放回从前的目录，startup 数据库即可。

5.3 联机全备份+日志备份

5.3.1 设置

如果数据库实例原来没有使用归档日志功能，则必须进行配置修改

initoradb.ora：

log_archive_start = true　#实例启动时同时启动归档进程。

log_archive_dest_1= “location=/appl/oracle/oradata/orafe/arch/arch”　#归档日志目录。

打开归档日志功能：

shutdown 数据库

sqlplus “/ as sysdba”

SQL>startup mount

SQL>alter database archivelog;

SQL>alter database open;

可用 archive log list 查看状态，去除归档日志功能的命令为 alter database noarchivelog。

5.3.2 步骤

参见 online/full.sh、daily.sh，以 osdba 组的用户执行

联机全备份：

数据库处于 open 状态，依次对各个表空间备份

sqlplus “/ as sysdba”

SQL>alter tablespace system begin backup;

复制此 tablespace 各个 datafile

SQL>alter tablespace system end backup;

注意：据推测，begin backup 是对 tablespace 冻结写入，end backup 是解除冻结，因此复制 datafile 的过程不宜过长

备份 controlfile

SQL>alter database backup controlfile to ‘…….’;

日志备份：

sqlplus “/ as sysdba”

SQL>alter system archive log stop;

移去日志目录下的所有 archived redolog

SQL>alter system archive log start;

5.3.3 恢复

数据库处于 shutdown 状态

最差情况：磁盘全部损坏，仅保存上次联机全备份和每天日志备份

解决硬件故障，配置系统软件及环境

oracle 用户，将全备份和日志备份转移至相应目录，根据 initoradb.ora 中 controlfile 的配置，将备份控制文件复制到响应目录下

sqlplus “/ as sysdba”

SQL>startup mount

SQL>recover database until cancel using backup controlfile;

逐个确认待恢复的 archived redolog，待最后一个完成后，键入 cancel，使恢复结束

SQL>alter database open resetlogs;

注意：由于日志已经重置，所以应尽快做一次联机全备份

丢失某数据文件

只要将此文件从上次联机全备份中复制至其目录，并将自上次联机全备份以来所有日

志备份移至归档目录

sqlplus “/ as sysdba”

SQL>startup mount

SQL>alter database recover datafile ‘path/file’;或者简单些 recover database;

SQL>alter database open;

如果此文件损坏或丢失，又无备份，则只能将此文件脱机，将数据 exp 出来，重建表空间，再 imp 进去

sqlplus “/ as sysdba”

SQL>connect internal

SQL>startup mount

SQL>alter database datafile ‘path/file’ offline;

SQL>alter database open;

5.4 注意要点

无论有多少把握，恢复前先做冷备份，此为第一原则

不这样做，便是无路可退，一旦失误，后果不必多说。

rollback 段损坏

这是非常严重的问题，可在 initora.ora 中写入_corrupted_rollback_segments=(rxx)，启动时避开损坏的 rollback 段，这只是权宜之计。如数据库处于 archivelog，应从上一次全备份起利用备份的日志进行恢复；如数据库处于 noarchivelog，应尽快将全部数据 export 出来，重建数据库，再 import 进去。所有操作之前，应做冷备份。

数据库异常中止处理

通过手工 shutdown abort 操作中止数据库，不会产生大的问题，通常直接 startup 无需使用介质恢复命令

如果由于机器崩溃引起的中止，则情况严重得多，有可能要使用到上面提到的恢复方法，不过这种现象并不多见。一般需要显式使用介质恢复命令，如下：

sqlplus “/ as sysdba”

SQL>startup mount;

SQL>recover database;

SQL>alter database open;

6 数据库优化

所有参见内容都在附件 05_optimize 下。

一个以数据库为核心的应用系统，其 80%以上的效率决定于应用软件架构是否合理、简洁、高效，支撑软件和硬件终究是放在第二位考虑的问题，这是系统分析员和程序员们的价值所在和使命所系。

6.1 通用设置

对于每个数据库系统，以下设置是固定的，企图在此仅通过某种设置优化实现神奇的效果，不是解决问题的正确途径，效果极为有限。

6.1.1 硬件配置

I/0 负载均衡

按照一般要求，在没有 RAID 的情况下，将 I/O 频繁的数据文件分配在不同磁盘上，使磁盘负载均衡。

临时表空间、回滚表空间、在线日志的 I/O 操作都比较密集，同时应用数据对象在表空间上的分布也决定 I/O 操作的分布。

RAID

关于 RAID 5 的写效率一直存在争论，在实际应用中表现并不突出。如果有条件，可使用 RAID 0+1。RAID 5 适用于磁盘空间有限，又要求保证一定容错功能的中小型应用。

增加 RAID 卡缓存，并将 Cache 选项置为 WriteBack 而非 WriteThrough。此两项对系统 I/0 影响之大，令人印象极其深刻。

6.1.2 应用配置

表与索引分在不同表空间中，分配适宜的数据块尺寸

为表和索引建立不同的表空间，并在创建表空间时指定于此建立的表或索引的缺省存储参数，这样此表空间上的所有对象使用统一的数据块尺寸、扩展率和最大可用数据块数量等参数，简化了配置过程，减少了存储空间的碎片

根据应用确定索引

检查应用程序 SQL 语句，尽可能在 where 的查询条件中使用整个索引，如不能满足，至少用到索引首列。

如在 emp 中建立索引：`create index emp_x01 on emp(name,age)`。

如果查询条件包含索引首列 name：

```
select salary from emp where name like 'Tom%'
```

这样的查询能够利用索引 emp_x01。

如果查询条件不包含索引首列 name，即使使用了索引的某个域，也不能利用到索引 emp_x01:

```
select salary form emp where age=25;
```

索引首列的选择应综合考虑查询条件的组合情况，同时为使索引 B+树尽可能产生多的分叉，应使用值变化较多的域，否则可考虑建立位图索引（bitmap index）

查询条件中对列的函数运算，列与列之间的比较会使执行效率大大降低，原因还是在于不能充分利用索引：

```
select salary from emp where substr(name,2,1)='o';  #名字第二个字母是'o'
```

慎用 order by，group by，它们会选出所有符合条件的记录，在 temp 表空间上进行处理，再输出。

评估应用程序中 SQL 语句的合理性

explain 可以获得 SQL 语句的执行策略，从而提供改进的思路，但由于执行环境的不同而无法准确推断执行效率。

执行$ORACLE_HOME/rdbms/admin/utlxplan.sql 建立 plan_table

SQL>explain plan select salary from emp where age>25；

SQL>start $ORACLE_HOME/rdbms/admin/utlxpls.sql 或 utlxplp.sql

查看分析结果，主要看是否出现表扫描、涉及的记录条数和耗费时间

6.1.3 日常性能监控

utlbstat 和 utlestat

编写以下两个脚本：

run_utlbstat.sh

```
sqlplus "/ as sysdba" << EOF
@${ORACLE_HOME}/rdbms/admin/utlbstat.sql
exit
EOF
```

run_utlestat.sh

```
cd /export/oracle/tuning_report
sqlplus "/ as sysdba" << EOF
@$ORACLE_HOME/rdbms/admin/utlestat.sql
exit
EOF
```

选择适当时间，如 4:00 运行 run_utlbstat.sh，22:00 运行 run_utlestat.sh，可通过 crontab 方式执行。在/export/oracle/tuning_report 下生成 report.txt，并分析：

stats$lib PINHITRATIO 应高

stats$waitstat 如 undo header 较高，则应加入更多回滚空间

如 segment header 较高，则应加入更多 freelist(待查)

stats$roll 如 TRANS_TBL_WAITS 较高，则应加入更多回滚空间

stats$dc GET_MISS, SCAN_MISS 应较低

stats$files I/O 应平均分布于各磁盘

READS 和 BLKS_READ 相差较大表明表扫描，相差较小表明 index 充分被利用

6.2 实战分析

当应用系统出现联机处理或批处理程序执行时间变长、反应缓慢的现象，就应该积极寻找减少消耗增加效率的方法。硬件升级是最后的终极的手段，也是最不能体现人的能力的手段，矛盾的暂时平息不意味未来不会再次爆发。

6.2.1 总体分析

分析一个高负荷、低效率的数据库系统先从分析操作系统的表现开始。

cpu 利用率

使用 sar 2 10 发现：

a) %wio 保持在 50 以上，%usr 和%sys 数值偏低

b) 或者%usr 异常高，保持 80 以上，%wio 几乎为 0

而%idle 总是接近 0。

情形 a）说明数据的检索量极大，无法从缓冲区中及时得到，oracle 系统进程被迫从数据文件上将数据读入缓冲区替换掉一些陈旧的数据，从而使oracle服务进程处于等待状态。这是缓冲区不足的表现，可以通过适当增加缓冲区大小来缓解矛盾，但不能根本上解决问题，首先内存区域有限，不可能一直增加下去；其次即使缓冲区足够容纳下所需数据，矛盾会向 b）转化。

情形 b）说明 oracle 服务进程已能从缓冲区中得到几乎所有需要的数据，但由于冗余数据多，分析时间非常漫长，使 cpu 计算能力过多地被分配到 oracle 服务进程上。

两者的发生多为检索策略失当，特别在对索引的设计和利用上，从而导致表扫描。

必须结合应用程序的设计来进行改进，仅在数据库层面上很难有所作为。

曾经发生过一种情形类似 a），只是%wio 不算太高，而%idle 很高。无法用常理分析，后更换存储设备后解决，推测是存储设备的问题。

内存和交换空间

每个 oracle 服务进程大约占用 5M 左右内存（mem），内存不足时，容易引起物理页的频繁换入换出，使系统 I/O 活动增加；还会占用 20M 左右交换空间（swap），当交换空间不足时，就不能增加新的 oracle 连接。

vmstat 2 10 观察内存和交换空间的空闲情况，注意 pi（kilobytes page in），po(kilobytes page out)，结合 sar –d 2 10 观察磁盘的繁忙程度，及时调整硬件。

top 也能对内存和交换空间进行监控。

改进措施无非就是增加内存和交换空间，以及控制 oracle 服务进程的数目。

联机日志

sar –d 2 10 观察磁盘 I/0，注意 read 和 write 的比例

如 write 比例过高，打开$ORACLE_BASE/admin/oradb/bdump/alert.log，检查日志文件切换间隔，如小于 15 分钟，说明日志文件太小，导致切换频率过高，引起缓冲区和数据文件的同步（checkpoint）发生过于频繁。也可能由于数据库同步写进程（db_writer_processes）数量太少。

改进措施是创建新的日志文件组，适当扩大日志文件的尺寸，使同步间隔保持在 30 分钟以上为宜。

参见《配置》中的 **log_checkpoint_interval** 和 **db_writer_processes**，作综合考虑。

6.2.2 详细分析

排除了上述效率问题，就要开始对数据库本身进行考察。

6.2.2.1 察看 session 当前执行的 SQL 语句

使用 top 观察 oracle 服务进程 cpu 占用情况，通常情况下单个进程不应超过 5%，如果超过 10%，一定发生了严重的检索策略失当

在此情况下，可认为有问题的 oracle 服务进程代表的 session 当前执行的 SQL 语句耗费了大量的时间，通过分析此 SQL 语句，能获得改进效率的线索

记录下待分析的 oracle 服务进程的进程号 **PID**

获得 oracle 服务进程的 session id，对应的应用进程信息，如进程号、主机名、程序名，以及正执行的 SQL 语句在缓冲池中的编号

sqlplus "/ as sysdba"

SQL>select s.**sid**,s.**serial#**,s.**process**,s.**machine**,s.**program**,s.**sql_hash_value** from v$session s, v$process p where p.spid='**PID**' and s.**paddr**=p.**addr**;

获得正执行的 SQL 语句的文本

SQL>select q.**sql_text** from v$session s, v$process p,v$sqlarea q where p.spid='**PID**' and s.**paddr**=p.**addr** and s.**sql_hash_value**=q.**hash_value**;

这个方法显示此 oracle 服务进程的会话当前使用的 sql 命令，并不一定是导致效率低下的 sql 命令，但低效率的命令往往在会话中停留较长的时间，所以大部分情况还是能够找出来的。

6.2.2.2 某段间隔 session SQL 语句执行情况

接上节，为更精确地分析此 session 中 SQL 语句的执行情况，需要获得某段间隔各 SQL 语句所占的比重，从而可以针对耗时较长的语句进行改进

记录下待分析的 oracle 服务进程的进程号 **PID**

sqlplus "/ as sysdba"

SQL>select s.**sid**, s.**serial#** from v$session s, v$process p where p.spid='**PID**' and s.paddr=p.addr;

SQL>alter system set timed_statistics=true;

SQL>execute dbms_system.set_sql_trace_in_session(**sid**,**serial#**, **TRUE**);

或 execute dbms_session.set_sql_trace(TRUE)或 alter session set sql_trace=true #对当前 session 做 trace 分析，分析文件生成于 initoradb.ora 中 user_dump_dest 指定的目录下

一段时间后……

SQL>alter system set timed_statistics=false;

SQL> execute dbms_system.set_sql_trace_in_session(**sid**, **serial#**, FALSE);

利用工具 tkprof 将 report.trc 文件转化为可读形式

tkprof report.trc report.txt sort=EXECPU explain=system/manager ＃选择 CPU 占用时间进行排序

举例：

call	count	cpu	elapsed
parse	5	**0.15**	**2.08**
execute	12	**6.30**	**41.04**
fetch	0	0.00	0.00

update work_user_skill set work_level=:b0,skill_value=:b1 where (mb_loginname=:b2 and work_type=:b3) ＃**mb_loginname 和 work_type 恰好为主键**

Rows Execution Plan

```
-------  ---------------------------------------------------
0       UPDATE STATEMENT   GOAL: CHOOSE
0       UPDATE OF 'WORK_USER_SKILL'
```

```
0       TABLE ACCESS   GOAL: ANALYZED (FULL) OF ‘WORK_USER_SKILL’
```

分析：

cpu 与 elapse 相差很大说明 I/O 繁忙，请求等待时间长，有可能进行了表扫描

TABLE ACCESS 出现 FULL，即表扫描，说明：

可能一：未使用 index。

可能二：使用了 index，而未用首列。

可能三：使用了 index，而由于其检索信息（主要是 B+树）过于陈旧，使 oracle 在决定检索策略时，错误地选择了表扫描。

解决：

对策一：根据应用的实际需要酌情建立索引。

对策二：修改查询条件，或者修改索引的组成域和顺序，确保索引首列被使用。

对策三：更新相关索引的陈旧检索信息

sqlplus dbuser/oracle

SQL>analyze index emp_x01 validate structure;

或者简单些

SQL>analyze table emp validate structure cascade; #不仅重新检查了 table 的存储情况，还更新了它的所有 index 的检索信息

另外可重新组织 index

SQL>alter index emp_x01 rebuild [online]; #建议使用 online 选项，可以与更新此索引的其它操作同时进行

参考命令：

生成 table 的统计信息至 dba_tables,user_tables

```
analyze table emp compute|estimate statistics for table [ all columns
| columns col1,col2 | all indexed columns ]
```

生成 index 的统计信息至 dba_indexes,user_indexes

```
analyze index emp_x01 compute|estimate statistics
```

6.2.2.3 检查被锁的对象

由于某种原因操作的对象被加锁，而又不能在短时间内解锁，会在此对象上形成 sql 操作等待队列，导致部分操作不得不超时退出，系统效率下降到难以接受的程度。这种现象可能是死锁，也可能因为某些事务耗时太长，影响操作同一对象的短事务。通常是应用程序组织不当的结果。

检查被长时间锁的对象：

sqlplus “/ as sysdba”

SQL>select a.session_id,a.process,a.locked_mode,b.object_name,b.object_type,b.status from v$locked_object a,dba_objects b where a.object_id=b.object_id;

6.3 专题分析

6.3.1 巨表查询

参见 hugh,脚本都在 script 下，程序都在 program 下。。

问题的提出

在数据库应用的开发中常常需要记录用户历史、流水等这样的数据，开发者的一般思路是仅建立一张表，以一个不重复的递增的序列号作为主键，配合查询的需要建立若干索引。

然而在数据条数积累到相当多的时候，比如几千万这样的级别，查询的效率问题就会成为令人头痛的瓶颈，因为查询凭借的索引本身也达到了相当复杂庞大的程度，任何一次查询都会带来很高的开销。由此还给批处理、数据备份、数据清理带来相当大的困扰。

尽管数据库系统在提高查询效率上已经提供了相当强劲的功能，但单纯依赖数据库往往不能应对复杂多变的现实情况。不得不采用一些专用设计，但需要跟通用做法进行权衡。

试图寻找解决巨表查询的方法，同时考虑巨表备份、清理。为形象地说明，建立 user_log 表，这张表储存典型的用户历史，包括序列号、日期、用户名、历史信息等。

步骤：

在某数据库实例中建立用户 dbuser，口令为 oracle
进入 script 目录
根据 create_tablespace.sql 建立各表空间
根据 create_table.sql 建立表和索引
运行 genlog 生成测试用数据
运行 loadlog 将数据上载到数据库中
进入 program 目录
运行 make

数据组织方面，设想有一万个用户访问，每天留下 20 万条记录，延续 15 天，总共 300 万条记录，平均每人 300 条，每天 20 条，访问的数据随机生成

硬件设备为 PIII800EB+512M+Quantum FireBall 10 代

分割建表

由于 user_log 中 log_dt（记录日期）是一个划分数据范围的天然标志，所以很自然地想到把日期作为表名的一部分，从而将巨大的表分割成较小的表，期望从较小的表和索引中获得较高的查询效率。能否实现这个想法？于是有以下测试。

program 目录下有 user_log_static.c 和 user_log_dyna.c，前者对整个 user_log 查询，后者对已分割成 user_log_yyyymmdd 的表查询，随机生成某个用户名，选出在 log 中**这个用户**的**全部**记录，共 100 次。运行 7 次，前 2-3 次由于数据缓冲的原因与后面的相差较大，可以忽略掉。

选出全部记录 （以秒为单位）

未分割	198	122	118	71	53	46	46
分割	255	129	79	58	50	48	48

从结果上看，分割几乎没有任何效果。根据理论分析，表分割后，应用不得不使用动态 SQL 对每张表分别操作，原先的一次查询被分成多次串行，尽管每次查询的数据量大为减少，但叠加上去也很可观。查询的日期范围越大，叠加的效应越强。所以在这种情况下，分割表的做法不能带来效率上的提高，甚至可能更糟。

但在通常的应用中，不需要一下子选出全部的查询所得，更多的是分批提供，比如显示用户历史记录，一屏可能就只有 10 条，向前或向后翻页时再次向服务器提交请求。试想在分割表的情况下，有可能在取到足够记录时，只需要查询数目很少的表，这样效率提高就很明显。修改 user_log_static.c 和 user_log_dyna.c，将 Process 函数中关于变量 count 的两行注释去掉，只选出 **100 条记录**就中止查询，大致估计访问 5 张分割表。

100 条记录 （以秒为单位）

未分割	221	111	58	41	41	36	35
分割	96	51	30	25	18	15	14

可以作出这样的结论：只要将访问的表的数目控制在较小的范围，分割表的查询效率就能得到明显的提升。

分区建表

在 oracle8 中可以使用 partition 选项将表或索引存放在不同的区域中，这样的做法与上面的分割表方法似乎是殊途同归。

program 目录下有 user_log_partition.c，对 user_log_partition 进行查询，user_log_partition 的数据域和 user_log 完全一致，数据也是使用同样的规则随产生，可以认为 user_log_partition 比起 user_log 差别仅在于 partition 选项。

索引的分区是否能带来性能上的提升？先测试数据、查询使用的索引都分区配置的情况，第一行选出***全部记录***，第二行选出 ***100 条记录***

（以秒为单位）

全部	236	121	73	48	46	42	40
100 条	47	45	44	40	38	38	37

数据分区配置，而查询使用的索引非分区配置，第一行选出***全部记录***，第二行选出 ***100 条记录***

（以秒为单位）

全部	238	120	69	48	45	42	41
100 条	37	37	39	37	35	35	33

无论数据还是索引使用或不使用分区，都没能带来预期中的效率提升，也许是测试案例不够全面，测试用机器不够专业的原因吧。

总结

应用层次上的表分割，还是数据库层次上的表分区，是值得考虑的选择。前者以缩小单张表的规模为目标，企图在较小的数据范围内以较高的效率完成工作，同时数据备份、清理也能以表为单位进行处理，代价是应用必须围绕着设计调整，通用化的程度弱，编程方面考虑得多，特别在应用某些成型软件时，几乎不能作这样的调整。而表分区能在不改动应用的前提下，透明地提高查询效率（见 Oracle 文档，尽管在本次有限测试中未能体现），缺点在于配置过于繁琐，数据库管理员需要对应用的非常了解，根本上讲只是在巨表内部作了有限的优化。

6.3.2 对比测试

参见 benchmark，脚本都在 script 下，程序都在 program 下。

这组测试的目的是为了搞清楚对表的不同处理方式会导致效率上怎样的差异，同时希望以数字代替长久以来的猜测。

结合笔者曾经参与过的一个项目，为测试设立了 3 张表，参见 script/create_table.sql：

charge_fee：费用

charge_fee_dtl：费用明细

charge_fee_cfg：费用配置

charge_fee 模拟了某种费用，如水、电、煤、电话等每月的缴付记录，包括缴费状态、费用月份、用户名、缴费日期等信息。生成 1,000,000 条记录，分布在 10,000 个用户名上，平均每个用户 100 条记录。

charge_fee_dtl 进一步说明某笔费用的具体情况，如电话费常常细分为固定费用、市话、长话、漫游、国际等。对应 charge_fee，每条费用记录拥有 5 个分项（假定用户不会在每个分项上都产生费用，所以从 10 个分项中随机选择了 5 个），共有 5,000,000 条记录。它采用 charge_fee 的序列号，也就是主键，作为分项和费用**多对一**关系的凭据。

charge_fee_cfg 对 charge_fee_dtl 中每个分项进行文字说明，是一张配置表，共有 10 条记录，代表 10 个分项。

这 3 张表大致反映了大（charge_fee）－大（charge_fee_dtl）－小（charge_fee_cfg）的模式，下面分析在它们上面使用嵌入式 SQL 进行查询的效率。

测试服务器硬件配置：PII400，512M SDRAM，昆腾火球 10 代 10G。

软件配置：Solaris 8 Intel Platform Edition 10/01，Oracle 8iR3

Join vs Not Join

在开发程序的时候，表与表间的连接是经常的选择，这样做可以很简洁地写出代码。然而，将涉及的表连接起来和顺序选出之间的效率差异，哪些场合可以使用或者不适合使用连接，等等诸如此类的问题往往容易为开发者忽略。

大（charge_fee）－大（charge_fee_dtl）连接：

模拟需求：选取某个用户的费用记录，并选出每笔费用的分项记录（常用于查询、缴纳费用的场合）。

这个需求将 2 张记录都很多的表 charge_fee 和 charg_fee_dtl 联系起来，以比较采用连接与否的效率差异。

引入程序 bb_join.c 和 bb_nojoin.c（bb—big table & big table），前者使用一个连接 2 张表的游标，直接选出费用和分项记录，后者仅使用对 charge_fee 的游标，选出费用记录后，再根据费用序列号，从 charge_fee_dtl 中选出分项记录。两者都使用了 order by 选项。

为更好地显示查询效果，每次程序运行随机选取 100 个用户，记录总耗时。同种测试进行 7 次。

上面说过平均每个用户拥有 100 条费用记录，而每条费用记录拥有 5 个分项。

测试一：选出用户全部 100 条费用记录

（以秒为单位）

连接（bb_join）	96	78	77	76	77	76	77
不连接（bb_nojoin）	92	96	93	97	95	95	95

结果有点出乎笔者的意料，因为笔者长久以来认为 2 张记录都很多的表的连接会带来很大的系统开销，然而事实证明，连接和不连接（顺序查询）的效率近似，而且连接的表现更好一些！

联系到上一节《巨表查询》的结果，笔者很快就想到：不连接方式由于对每笔费用记录（charge_fee）都要重新展开一个对分项记录（charge_fee_dtl）的游标，总计大约 100（每个用户费用记录条数）*100（用户数）=10,000 次之多，耗费很大，而实际应用中很难有一次选出某用户上百条记录的情况，如果减少一点，比如每个用户只选出 20 条，游标开启次数降为约 2,000 次，结果会怎样？

基于这个想法，在 2 个程序中加进了对费用记录条数的控制，见注释部分。

测试二：选出用户前 20 条费用记录

（以秒为单位）

连接（bb_join）	82	75	67	67	66	68	72
不连接（bb_nojoin）	24	17	13	10	9	8	8

结果同样令笔者吃惊！性能提高得很明显，由此可以得出与《巨表查询》相似的结论：**在涉及 2 张记录都很多的表操作中，如果采用不连接的方式，只要将顺序完成的查询控制在一定数量内，一定能取得比连接方式高的效率。**

大（charge_fee_dtl）－小（charge_fee_cfg）连接：

模拟需求：选取某笔费用的分项记录，并显示每笔分项的文字说明（常用于打印单据的场合）。

这个需求将 1 张记录都很多的表 charge_fee_dtl 和另 1 张记录极少的表 charg_fee_cfg 联系起来，以比较采用连接与否的效率差异。

引入程序 bs_join.c 和 bs_nojoin.c（bs—big table & small table），前者使用一个连接 2 张表的游标，直接选出分项和配置记录，后者事先将 charge_fee_cfg 全部读取到内存中，仅使用对 charge_fee_dtl 的游标，选出分项记录后，再根据分项类型，从内存中选出分项的文字说明。两者都使用了 order by 选项。

为更好地显示查询效果，每次程序运行随机选取 30,000 笔费用记录，记录总耗时。同种测试进行 7 次。

（以秒为单位）

连接（bs_join）	77	75	76	74	76	74	75
不连接（bs_nojoin）	67	66	66	65	67	65	66

结果差不多，可以得出结论：**在涉及 1 张记录很多的表和另 1 张记录很少的表操作中，采用连接与否，效率相差不大。从灵活的角度考虑，建议对记录很少的表（往往是静态配置）多利用内存缓冲。**

Static vs Dynamic

这里比较静态嵌入式 SQL 和动态嵌入式 SQL 语句的效率差异，动态方式多了一个对 SQL 语句的 prepare 过程。

引入程序 bs_nojoin_dyna.c，其实就在 bs_nojoin.c 的基础上，把静态嵌入式 SQL 语句的部分代码改成动态嵌入式 SQL 语句。

（以秒为单位）

静态（bs_nojoin）	67	66	66	65	67	65	66
动态（bs_nojoin_dyna）	97	96	95	96	94	95	95

动态方法比起静态方法效率上有一定的差距，但如果考虑到动态方法的灵活性，笔者认为可以接受。

Local vs LAN

实际应用中经常把数据库和应用程序安装在独立的服务器上，这样应用进程调用数据库接口时，会受到一些因素的影响，如网络接口系统调用开销、操作系统内核对发起网络请求进程的调度、网络传输延迟等等，效率下降是不可避免的，下面对这样的效率损失作一个有趣的统计。

将程序 bs_nojoin.c 移到另一服务器上，以此服务器为客户机，数据库所在服务器为服务机。

客户机配置：PIII Xeon 550，1G SDRAM，IBM DNES-318350Y 硬盘，服务机维持不变。网络为 10M 以太网。

（以秒为单位）

本地（bs_nojoin）	67	66	66	65	67	65	66
局域网（bs_nojoin）	171	165	165	165	166	165	165

效率差距令人震惊！由于条件限制，没有对 100M 以太网的情况进行测试，相信会有

较大的改善。

结论：**尽可能本地访问数据库，如果必须跨机访问，应使用高速局域网。**

6.3.3 上下载数据

参见 upload_download。

一直认为把数据从数据库里倒进倒出是一件体力活，仅仅是利用一些数据库工具的简单任务。然而在经历了无数个不眠之夜后，渐渐体会到，这个简单任务居然已经成为 DBA 的一项主要工作，而且是直接关系到 DBA 身心健康的大问题。所以觉得有必要对此作一点研究了。

测试服务器硬件配置：PIII800EB，512M SDRAM，Maxtor DiamondMax VL 40 33073H3 30G 硬盘。

软件配置：RedHat Linux 7.3，Oracle 9iR2

参见 create_tablespace.sql 和 create_table.sql，建立测试所需表空间和表。

表 user_info 模仿了某种用户信息记录，建立一些索引方便查询。建表语句没有包括 primary key 关键字，具体原因到讨论 imp 时会提到。

上载文本--sqlldr

关于下载文本，在《工具》中的 sqlplus 里已经介绍过了，而且基本上没有提高效率的方法，因此不作深入。

首先产生 3,000,000 条记录，运行 genrec，本目录下生成 ctl 目录，里面有一个叫 user_info.ctl 的文件，即为 sqlldr 的控制文件，数据都在此文件中，具体选项参见《工具》中的 sqlldr。运行 loadrec 进行测试。

测试一：不使用任何参数

sqlldr dbuser/oracle control= ctl/user_info.ctl log=ctl/user_info.log

缺省情况下，sqlldr 采用常规路径（conventional path）方式上载，rows=64，bindsize=65535

耗时：41:07

测试二：常规路径（conventional path）方式，采用较大的缓冲参数

sqlldr dbuser/oracle control=ctl/user_info.ctl log=ctl/user_info.log rows=10000 bindsize=8192000

参见 loadrec 中的 conventional path

耗时：24:31

测试三：直通路径（direct path）方式

sqlldr dbuser/oracle control=ctl/user_info.ctl log=ctl/user_info.log direct=y

参见 loadrec 中的 direct path

耗时：6:15

由此可见直通方式能够大幅度提高上载效率。

测试四：直通路径（direct path）方式，关闭表和索引 logging

因为数据插入会记录日志，如果关闭表和索引的 logging 选项，应该可以进一步提高上载效率。

对于常规路径方式，无论 logging 选项是否关闭，都会记录日志，所以在测试一和测试二中没有调整 logging 选项。

对表和索引使用 alter table|index … nologging 命令。

sqlldr dbuser/oracle control=ctl/user_info.ctl log=ctl/user_info.log direct=y

参见 loadrec 中的 direct path

耗时：3:25

结论：使用直通方式上载，并关闭相关表和索引的 logging 选项。

注意：采用直通方式，如果被操作的表的索引上载前已被建立，在上载过程中不对索引进行操作，待数据全部进入后，再执行更新索引的操作，奇怪的是，进行此操作时数据库使用的是回滚空间，所以应注意回滚空间的大小。

曾尝试过使用不同大小的日志文件，以求降低检查点（checkpoint）发生的频率，但几乎没有产生任何效果。

导出 dmp 文件--exp

假定表 user_info 中已有了 3,000,000 条记录。

运行 exprec 进行测试，在本目录下新建 dmp 目录，在此目录下产生 user_info.dmp 文件。

测试一：常规路径（conventional path）方式

exp dbuser/oracle file=dmp/user_info.dmp log=dmp/user_info.log tables=user_info buffer=4096000 feedback=10000

参见 exprec 中的 conventional path

耗时：2:17

测试二：直通路径（direct path）方式

exp dbuser/oracle file=dmp/user_info.dmp log=dmp/user_info.log tables=user_info direct=y feedback=10000

参见 exprec 中的 direct path

耗时：1:37

结论：使用直通方式导出 dmp 文件。

导入 dmp 文件--imp

利用已经导出的 user_info.dmp。

运行 imprec 进行测试。

测试一：不作任何调整

imp dbuser/oracle file=dmp/user_info.dmp tables=user_info fromuser=dbuser touser=dbuser buffer=4096000 commit=y ignore=y feedback=10000

参见 imprec 中的 index

耗时：19:29

联系 sqlldr 的表现，很自然地想到将表和索引的 logging 选项关闭，然而 imp 没有直通方式！因此 logging 选项在此无所作为。

在导入过程中 imp 缺省更新索引，当数据导入量逐渐增大，对索引的更新是一个逐渐变慢的过程，所以一个很自然的想法是：先导入数据，再建立索引。可以设置选项 indexes=n 使 imp 在导入数据同时不进行更新索引的操作，然而如果导入前索引已被建立，此选项无效（参见《工具》中的 imp），因此必须在导入前：**1.先建立表（imp 会缺省建表和索引）2.确保所有的索引没有建立。**这也是建表脚本不使用 primary key 的原因：primary key 会建立 unique index，而且不能去除，就 primary key 的功能而言，代之以 unique index 没有问题。

测试二：使用 indexes=n，先不建立任何索引，导入结束后，再建立索引

参见 imprec 中的 no index

imp dbuser/oracle file=dmp/user_info.dmp tables=user_info fromuser=dbuser touser=dbuser buffer=4096000 indexes=n commit=y ignore=y feedback=10000

耗时：7:55

SQL>create unique index user_info_x00 on user_info(loginname) tablespace tbsindex;

SQL>create unique index user_info_x01 on user_info(nickname) tablespace tbsindex;

SQL>create index user_info_x02 on user_info(register_dt) tablespace tbsindex;

耗时：2:20

总耗时：10:15

结论：先建表，不建索引，使用不更新索引的导入方式，最后统一建立索引。

先导入数据还有一个额外的好处：数据尽早安全地进库，无论对 DBA 还是项目的其他成员心理上都是一个极大的安慰，反正有了数据就有了一切，至于后面的建立索引，相对从容许多了，即使时间拖得长一些，关系也不大。这是笔者多次倒数据的切身体会。

6.3.4 回滚空间快照陈旧（snapshot too old）

有这样的两种情况：

1. 根据某些条件选取一批记录，在游标不关闭的情况下，作一定处理，再更新回去
2. 根据某些条件选取一批记录，输出到某地。

这是再平常不过的操作。但如果这个选取过程变得比较漫长，往往在某个时刻的 fetch 会返回 snapshot too old 错误，导致选取过程中断。尤其是第一种情况，发生错误的概率很大。

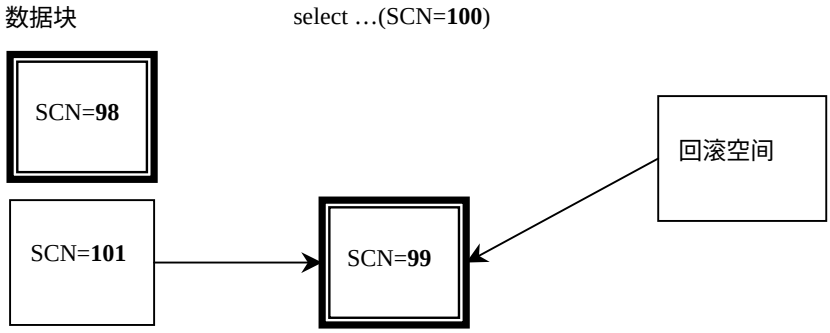
这是 Oracle 特有的现象，跟它对回滚空间的利用策略有关。

Oracle 自动使用语句级别的读取一致性策略，保证返回的数据都是查询发起时刻的模样，对查询过程中结果集发生的变动不加理会。

为了达到这个目的，Oracle 使用回滚空间中维持的信息结合数据文件提供一致性视图，回滚空间包含已被未提交或已提交事务修改过的数据的原值。

Oracle 使用 System Change Number(SCN)区分时刻。当查询语句进入查询阶段以后，系统分配一个 SCN，产生结果集后，对应到数据文件，只有 SCN 比查询 SCN 旧的数据块才会被读取，如果在查询过程中某数据块被更新，即 SCN 被改变，比查询 SCN 新，这样必须检索回滚空间，重建旧数据块的内容。

如下所示，加黑显示的数据块才是最终选取的。



如果回滚空间中相应的内容被覆盖，Oracle 就认为旧数据块过于陈旧，无法重建出来，

这就是 snapshot too old 的由来。针对这种错误，Oracle 建议使用更大的回滚空间，这是有道理的，但更大的回滚空间仍有不敷使用的时候，不足以应付多变的应用，只能算一种消极应付的对策。

第一种情况：

假定不能重建的数据块为 X，当 X 中的记录被更新（无论来自于本次游标操作，还是其它数据库连接，更多可能是来自于自身），SCN 同时刷新，如果经过某段间隔再要读取 X 中记录就必须根据回滚空间相关内容重建旧 X，而一旦这段间隔中发生较多的数据库更新操作，回滚空间有可能在消耗完空闲内容后被迫覆盖旧内容，这样 X 的重建就无法完成。

比较好的解决方法是：游标展开后将记录主键和更新所需用到的域读至内存，不执行更新操作，随即关闭游标，再处理这些记录，根据主键分批更新回数据库。

第二种情况：

与第一种情况类似，只不过更新操作全部来自于其它数据库连接。

没有太好的解决方法，尽量加快查询过程，保证其它连接不发生或少发生与本次查询相关的更新操作。

7 常用技巧

所有参见内容都在附件 07_skill/下。

7.1 增加、更改和删除域

增加域

SQL>alter table emp add(column_x char(10) not null);

域被增加到表结构的末尾。

更改域

SQL>alter table emp modify(column_x char (20));

对于 char 类型的域，只能扩展，不能缩小；char 与 varchar2 可互为转换。

删除域

如待删除域属于某个索引，则不允许删除操作，必须将此域先设置为 NULL。

SQL>alter table emp modify(column_x null);

SQL>update emp set column_x=null;

SQL>commit;

SQL>alter table emp drop(column_x);

在原表基础上改动结构有一定的局限性，比如“增加域”，新域只能添加到表结构的末尾，破坏了原有的整齐和协调。建议采用这样的方法：新建表，再将原表数据迁移过去，最后去除旧表。

假设要在表 dbuser.emp 中增加域 colx,(col1,col2,col3)→(col1,col2,colx,col3)

修改 emp.sql

sqlplus dbuser/oracle

SQL>alter table emp rename to emp_temp;　＃原表改名

SQL>start emp　＃建立新表

编辑 SQL 脚本 modi.sql

　　insert into emp(select col1,col2,' ',col3 from emp_temp where …);

sqlplus system/system

SQL>alter rollback segment r99 online;　＃见“调整 rbs 表空间”中关于 r99 的设置，Oracle 9i 中不需要

SQL>connect dbuser

SQL>set transaction use rollback segment r99;　＃见“调整 rbs 表空间”中关于 r99 的设置，Oracle 9i 中不需要

SQL>start modi;　＃迁移数据

SQL>commit;

SQL>connect system

SQL>alter rollback segment r99 offline;　＃见“调整 rbs 表空间”中关于 r99 的设置，Oracle 9i 中不需要

SQL>drop table emp_temp;　＃去除旧表

7.2 删除冗余记录

参见 02/。

需求：user_log 记录用户大量 log，现在需要进行清理，只保留最新一条。

表结构：

```
create table user_log
(
  log_sq          number(12) not null,
  user_nm         char(20) not null,
  user_log_tx     varchar2(255) not null,
  last_upd_ts     date,
  primary key (log_sq)
);
```

log_sq 利用了 sequence 的值。

方法：

```
sqlplus –s dbuser/oracle <<EOF >/dev/null
set heading off;
set term off;
set echo off;
set pagesize 0;
set linesize 1000;
set trimspool on;
set trimout on;
set feedback off;
set colsep |;
spool tmp.txt;
select user_nm,count(*),max(log_sq) from user_log group by user_nm having(count(*)>1);
spool off;
exit
EOF
tr –d ‘ ‘ < tmp.txt | sed –e “s/^/’/
s/|/’|/1” | awk ‘BEGIN { FS=”|” } { printf(“delete from user_log where user_nm=%s and log_sq<%s;\n”, $1, $3); }’ > del.sql
```

执行 del.sql

注意：sed 要用双引号，作用是为 user_nm 加上单引号，因为 awk 命令中的单引号会与其命令引用所用的单引号混淆，无法在 awk 命令行中为 user_nm 添加单引号，使用单独的 awk 脚本就没有这个问题

7.3 更改字符集

条件 新字符集必须为原字符集的超集，如 USASCII7→ZHS16GBK。ZHS16CGB231280 并不是 ZHS16GBK 的子集

假设 oradb 原字符集为 USASCII7，更改为 ZHS16GBK

```
sqlplus “/ as sysdba”
SQL>startup mount
```

SQL>alter system enable restricted session;
SQL>alter system set job_queue_processes=0;
SQL>alter database open;
SQL>alter database character set ZHS16GBK;
SQL>shutdown
SQL>startup

7.4 表数据迁移

需求：将一张表或几张表中的域重新组合后插入新表。

上面的“ 增加、更改、删除域 ”已经描述了一张表迁移到另一张表的情况，这里讨论多张表合并的情形。

假定原先的两张表为 emp,work，现选择部分数据域合并为 emp_work

对回滚段的操作不再重复叙述，Oracle 9i 中不需要

建立 emp_work

SQL>insert into emp_new select a.no, sysdate, a.name, b.service_duration from emp a, work b where a.no=b.no;

SQL>commit;

当连接涉及的数据量较大时，对临时段 next 扩展块大小有一定要求，参见《创建》中的“ 调整临时表空间 ”。

这样的方式仍然要使用回滚段，为加快数据迁移速度，可将 **insert** 替换成 **insert /*+APPEND*/**（大小写不论），指示 oracle 以直通方式直接写数据文件，绕过回滚空间。

SQL>insert /*+APPEND*/ into emp_new select a.no, sysdate, a.name, b.service_duration from emp a, work b where a.no=b.no;

SQL>commit;

7.5 成批生成数据

参见 05/。

笔者曾经为一个类似题库的项目做过数据上载，题目是选择题。需求非常简单，但对于 DBA 来说，此类的数据维护经常会碰到，所以在这里记录下来，仅仅是为了提供一种解决问题的思路。

由于题库的提供者对数据库一无所知，所以跟他们之间的数据文件接口使用了简单而通用的定义。这是一个文本文件，每道题目和答案占用一行，每个数据域以“ | ”分隔，为简单计，第一个答案为正确答案，例如：

GSM 技术最初是哪个地区的标准？|欧洲|美国|中国|日本

对此接口文件要进行详细的描述，最好提供样例文件，非技术人员对“ 数据接口 ”这一概念感觉极为迟钝，如果不作硬性规定，他们会怎么方便怎么来，结果象 word、excel、access 等五花八门的文件都会出现，格式随心所欲，甚至还会口头通知。

下面的问题是将每行记录拆分到 question 和 answer 两张表中去，并在表 question 中记录正确的答案编号，而表 answer 中相对于某问题的数个答案编号必须是打乱的。这一任务交给 awk 完成。

打乱答案编号：

```
choice_qt=NF-1;  #答案的数目
 for(i=0; i<choice_qt; i++)  aAnswer[i]=0;
```

```
        i=0;
        while(i<choice_qt) #依次分配答案编号
        {
            ind=int(rand()*choice_qt);  #随机取得答案下标，注意 rand()的范围是
[0,1)
            if(aAnswer[ind]==0)  #如果此下标代表的答案还没有被赋予编号，就将此
编号赋予它；如果已有了编号，不作任何处理，再去随机选择答案下标。这种方法会浪费
一些效率，但比较简单。
            {
                aAnswer[ind]=i+1;  #编号实际从 1 开始
                i++;
            }
        }
```

拆分成两张表：

```
    printf("insert into question values(question_seq.nextval,'%s',%d);\n",
  $1,aAnswer[0]);
    for(i=0; i<choice_qt; i++)
    {
        printf("insert into answer
     values(question_seq.currval,%d,'%s');\n",
     aAnswer[i],$(i+2));
    }
```

7.6 注意要点

cursor

cusor for update 不能嵌套使用，并且只能在一个 transaction 中完成

sequence

在同一 session 中，必须首先调用 sequence.nextval，如先调用 sequence.currval，oracle 会认为使用了 sequence 未定义的值

有些场合往一张或多张表中插入记录时，需要使用相同的序列号，可先得到 sequence.nextval，代入宿主结构，再对表操作

比如增加一条雇员工资记录（emp_salary_hist）及其明细（emp_salary_hist_dtl）

SQL>select emp_salary_hist_sq.nextval from dual;

SQL>insert into emp_salary_hist values(**seq**,…);

SQL>insert into emp_salary_hist_dtl values(**seq**,…);

rownum

rownum<=N 作为查询条件时，不能与 order by,group by 此类需将记录全部选出来的操作共用，因为 rownum 优先。

选择记录集中第 N 条记录时应参照以下做法：

如此记录集的查询 SQL 语句不含 order by

select name from (select name, rownum rowseq from emp where age>25) where rowseq=N;

如此记录集的查询 SQL 语句包含 order by

select name from (select name,rownum rowseq from (select name from emp where age>25 order by no)) where rowseq=N;

count

获取记录集条数时，count(1)最快，count(某列)得出的记录数目不包括此列中 null 值的数量

8 嵌入式 SQL（C）

所有程序例子都在附件 08_proc/下。

在 C 语言程序代码中直接嵌入 SQL 语句，使数据库编程变得非常简单明了，而且嵌入式 SQL 是一种标准，代码不需要很多的修改就能移植到支持嵌入式 SQL 的数据库系统上去，但这同时也是一个缺点，许多数据库系统不提供嵌入式 SQL 的预编译器。

8.1 编译

编译过程分为两步，第一步，对带有嵌入式 SQL 的 C 代码程序（通常此程序以.pc 结尾，简称 PC 代码）使用 proc 做一次预编译，将里面的嵌入式 SQL 转化为代表数据库功能调用的 C 代码。第二步，使用 C 编译器将 C 代码编译连接成可执行文件。

配置文件

proc 首先读取$ORACLE_HOME/precomp/pcscfg.cfg，里面的选项与命令行选项作用完全相同，这样可以让 proc 使用变得简洁。但从减少系统配置步骤、保持应用迁移的灵活性的角度出发，建议对此文件不作修改，将各种选项写在程序的编译文件中。

命令行选项：

sys_include：系统头文件目录，如/usr/include,/usr/lib/gcc-lib/i486-suse-linux/2.95.3/include,/usr/lib/gcc-lib/i386-redhat-linux/2.96/include ,$ORACLE_HOME/precomp/public,$(ORACLE_HOME)/rdbms/public -I$(ORACLE_HOME)/rdbms/demo，前 3 个通常在配置文件中已经写入

include：自定义的应用头文件目录

sqlcheck：SQL 语句检查方式，语法（SYNTAX）和语义（SEMANTICS）。

语法：仅检查 SQL 语句的语法结构

语义：除语法检查外，还检验数据库对象和宿主变量的有效性

userid：访问数据库途径，在 sqlcheck 为语义检查时必须指定，如 dbuser/oracle@oradb

iname：输入 pc 文件名

oname：输出 c 文件名

如果省略 iname、oname，只需输入 pc 文件名，缺省转化成同名后缀为.c 的 C 文件。

threads：线程支持，yes 或 no

例子

proc sqlcheck=semantic userid=dbuser/oracle threads=yes
sys_include=$(ORACLE_HOME)/precomp/public
sys_include=$(ORACLE_HOME)/rdbms/public
sys_include=$(ORACLE_HOME)/rdbms/demo include=$IN threads=yes iname=emp.pc
oname=emp.c //$IN 是自定义的应用文件目录

gcc -I$(ORACLE_HOME)/precomp/public -I$(ORACLE_HOME)/rdbms/public -I$(ORACLE_HOME)/rdbms/demo –g –c –o emp.o emp.c

参见 proc/single/Makefile。

8.2 SQL 语句

建议设立一个与表结构一一对应的结构定义，作为宿主结构，下面的所有例子几乎都会用到关于表 emp 的宿主结构（emp_t emp）。

参见 proc/single/db.h 中 emp_t 和 emp 建表脚本 proc/single/emp.sql。

8.2.1 内部类型与宿主类型对应

下面是最常用的内部数据类型和宿主变量类型的对应关系：

内部数据类型	宿主变量类型
char(n)	char[n+1]
varchar2(n)	char[n+1]
number(n<=6)	int
number(n>6)	double
number(m,n)	double
date	char[15]

8.2.2 连接和断开

最简单的形式

这是最为常用的形式：应用程序不指明任何数据源，Oracle 库从环境变量 ORACLE_SID 中得到数据库实例名，在本机连接此实例。应用程序只要输入数据库用户名和口令即可。

```
strcpy(dbuser,”dbuser”);  strcpy(password,”oracle”);
EXEC SQL connect :dbuser identified by :password;  //dbuser 和 password 是 char[n]的宿主变量。
```

也可以将用户名和口令写在一起。

```
strcpy(dbuser_password,”dbuser/oracle”);
EXEC SQL connect :dbuser_password;
```

跨机连接

首先在$ORACLE_HOME/network/admin/tnsnames.ora 中添加对方数据库信息，取得一个连接名。

假定已经在 tnsnames.ora 中配好了名为 oradb2 的连接。

```
strcpy(dbstring, “oradb2”);
EXEC SQL connect :dbuser identified by :password using :dbstring;
```

多个连接

可以在一个应用程序中同时使用两个以上的数据库连接，为区分不同的连接，要为每个连接起一个名字。

```
strcpy(dbuser,”dbuser1”);  strcpy(password,”oracle1”);
strcpy(dbconnect,”dbconnect1”);
EXEC SQL connect :dbuser identified by :password at :dbconnect;
strcpy(dbuser,”dbuser2”);  strcpy(password,”oracle2”);
strcpy(dbconnect,”dbconnect2”);
EXEC SQL connect :dbuser identified by :password at :dbconnect;
```

在多连接的情况下，每句 sql 都要指明连接名。

EXEC SQL at :dbconnect select …; //下面不再讨论多连接，仅仅说明其使用方法

断开

应用程序无论正常还是非正常退出，数据库连接自动中断，也可以显式关闭。

EXEC SQL commit work release;

8.2.3 事务

提交

EXEC SQL commit;

回滚

EXEC SQL rollback;

8.2.4 标准 SQL 语句

select

EXEC SQL * into :emp from emp where no=:emp->no;

EXEC SQL select name,to_char(upd_ts,'yyyymmddhh24miss') into :emp->name,:emp->upd_ts from emp where no=:emp->no;

lock

EXEC SQL select * into :emp from emp where no=:emp->no for update;

update

EXEC SQL update emp set duty=:emp->duty,upd_ts=sysdate where no=:emp->no;

delete

EXEC SQL delete from emp where no=:emp->no;

insert

EXEC SQL insert into emp values (:emp);

EXEC SQL insert into emp values (:emp->name,:emp->name,:emp->age,:emp->duty,:emp->salary,:emp->to_date(upd_ts,'yyyymmddhh24miss'));

cursor

EXEC SQL declare emp_cur cursor for select * from emp where age>25;

EXEC SQL open emp_cur;

EXEC SQL fetch emp_cur into :emp;

EXEC SQL close emp_cur;

8.2.5 动态 SQL 语句

在输出域或输入查询条件，甚至表名，不能事先确定的时候，可以引入动态 sql 语句的方法。

非查询语句

专指 insert,update,delete 等无结果集的 sql 语句。

不带宿主变量：

最直接的做法，处理一条纯粹的 sql 语句，不需要任何变化，每次都完整地从 sql

解析走到执行。

```
strcpy(sql,"delete from emp where no=0");
EXEC SQL execute immediate :sql;
```

查询条件使用宿主变量：

目的是为了只进行一次解析，以后只需绑定不同输入就能直接执行。

```
strcpy(sql,"delete from emp where no=:no");
EXEC SQL prepare sql_stmt from :sql;
emp.no=0;
EXEC SQL execute sql_stmt using :emp.no;
emp.no=1;
EXEC SQL execute sql_stmt using :emp.no;
```

查询语句

查询必须使用游标方式。

```
strcpy(sql,"select * from emp where duty=:duty and age>:age");
strcpy(emp.duty,"1");    emp.age=20;
EXEC SQL prepare sql_stmt from :sql;
EXEC SQL declare sql_cur cursor for sql_stmt;
EXEC SQL open sql_cur using :emp.duty,:emp.age;
EXEC SQL fetch sql_cur into :emp;
EXEC SQL close sql_cur;
```

8.2.6 数组操作

利用宿主数组可以用一次 sql 操作处理成批记录，降低了应用程序与 oracle 服务进程之间的通讯开销，也减少了 sql 语句解析次数，这样能提高数据库应用程序的效率，对 update,delete 和 insert 尤为有效。

宿主数组

假定要操作表 emp，定义宿主数组：

```
double a_no[50]
char a_name[50][21];
```

CURSOR

目的在于：试图在一次 fetch 中选出批量数据以减少 fetch 的次数。但 fetch 过程本身消耗资源不大，所以使用宿主数组对效率提高有限。

```
int rec_count;   //本次条数
int accu_count;   //累计条数
EXEC SQL declare emp_cur FOR select no,name from emp where age>25;
EXEC SQL open emp_cur;
EXEC SQL fetch emp_cur into :a_no,:a_name;   //rec_count 保存当前 fetch 的记录条数，最多为 a_no,a_name 中较少的结构个数
rec_count=sqlca.sqlerr[2]-accu_account;   //sqlca.sqlerr[2]记录 cursor 累积读取的记录条数，这样与上次累积数之差就是本次读取的条数
accu_count=sqlca.sqlerr[2];   // accu_count 保存累积 fetch 的记录条数
```

当最后一次取完时，sqlca.sqlcode 为 1403，即记录不存在，注意 count 可能会大于 0，这里的“记录不存在”只是表明“没有填满宿主结构”。

SELECT

EXEC SQL select no,name into :a_no,:a_name from emp where age>25;

不能用 for :rec_count 子句进行操作记录条数控制，即不可指定选出若干条记录，因为有“执行此语句若干次”和“一次选出若干条记录”的歧义，需要条数控制应使用 CURSOR。

DELETE

在 a_no 中准备 N 条待删除记录的 no，也就是主键。

a_no[0]=0;　a_no[1]=1;　a_no[2]=2; …

rec_count=N;

EXEC SQL for :rec_count delete from emp where no=:a_no;

UPDATE

假定要更新一部分记录的 name 域。

在 a_no 中准备 N 条待更新记录的 no，在 a_name 中准备 N 个新的 name，两者必须一一对应。

a_no[0]=0;　strcpy(a_name[0],”职员 0”);

a_no[1]=1;　strcpy(a_name[1],”职员 1”);

…

a_no[N-1]=N-1;　strcpy(a_name[N-1],”职员 N-1”);

rec_count=N;

EXEC SQL for :rec_count update emp set name=:a_name where no=:a_no;

INSERT

为了说明问题，这里对 emp 表作了一些改变，假定除 no 和 name 域外都有缺省值。

在 a_no,a_name 中准备 N 条记录的 no 和 name，两者必须一一对应。

a_no[0]=0;　strcpy(a_name[0],”职员 0”);

a_no[1]=1;　strcpy(a_name[1],”职员 1”);

…

a_no[N-1]=N-1;　strcpy(a_name[N-1],”职员 N-1”);

rec_count=N;

EXEC SQL for :rec_count insert into emp(no,name) values(:a_no,:a_name);

宿主结构数组

宿主结构数组其实是宿主变量组合起来的数组，由于 update set 语句的特殊性，所以宿主结构数组仅能用于 SELECT,FETCH 和 INSERT

定义部分域结构组合 emp_part_def a_emp_part[50]，和 a_no[50]形成混合模式。

```
double a_no[50];
typedef struct
{
    char name [21];
    int age;
} emp_part_def;
emp_part_def a_emp_part[50];
```

SELECT

EXEC SQL select no,name,age into :a_no,:a_emp_part from emp where age>25;

CURSOR

iAccuCount=0;

EXEC SQL DECLARE emp_part_cur FOR select name,age from emp where age>25;

EXEC SQL OPEN emp_part_cur;

EXEC SQL FETCH emp_part_cur into :a_emp_part;

rec_count=sqlca.sqlerr[2]-accu_count;

accu_count=sqlca.sqlerr[2];

INSERT

emp_def a_emp[50];

在 a_emp 中准备 N 条记录。

a_emp[0].no=0; strcpy(a_emp[0].upd_ts,”20020101000000”); …

a_emp[1].no=1; strcpy(a_emp[1].upd_ts,”20020101000000”); …

…

a_emp[N-1].no=N-1; strcpy(a_emp[N-1].upd_ts,”20020101000000”); …

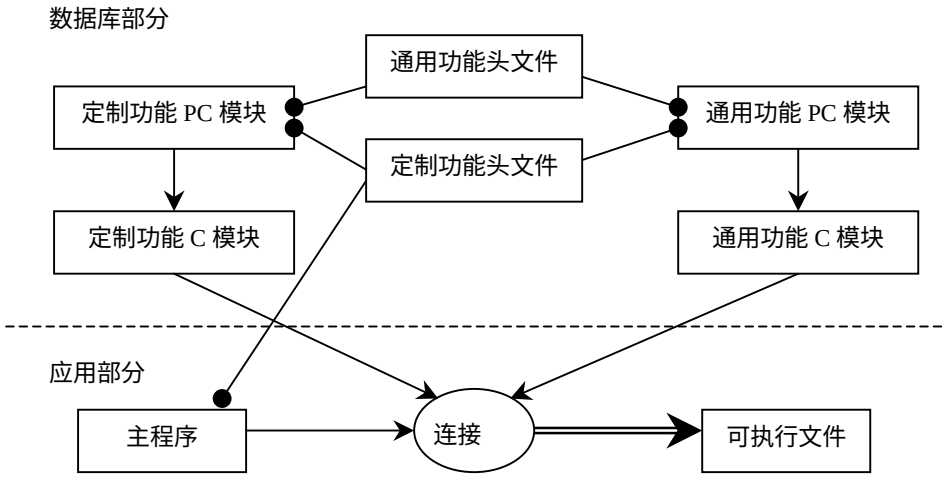
rec_count=N;

EXEC SQL for :rec_count insert into emp values(:a_emp);

8.3 编程框架

8.3.1 总体原则

为了使程序模块划分更加简洁清晰，提高应用程序的可移植性，所以采取将主程序模块和数据库功能模块分离的方法，即不在主程序逻辑中出现嵌入式 sql 语句，而把这些语句归并到数据库功能模块中去，对主程序而言，展现的是某种功能。同时在数据库模块上再进一步把通用功能和定制功能分割开来。大致框架如下：



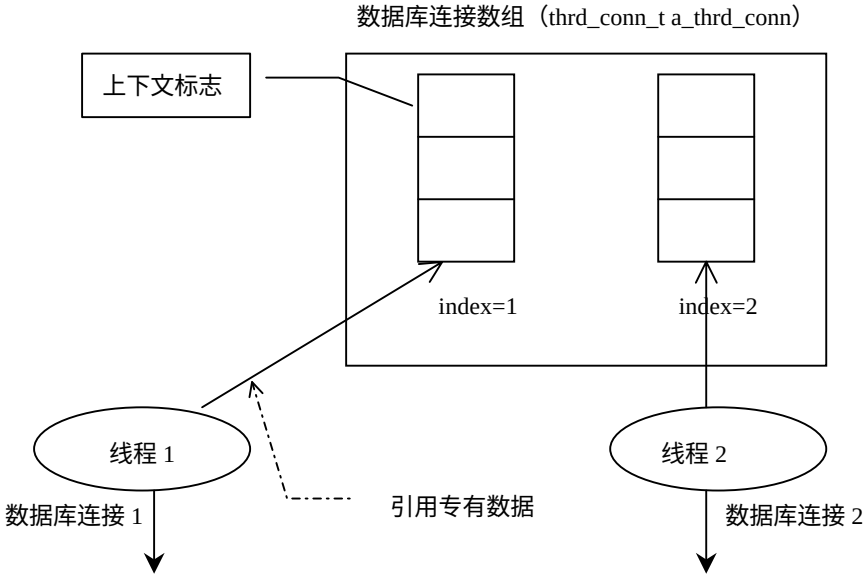
通用功能头文件和通用功能模块参见 proc/single/dbcom.h 和 proc/single/dbcom.pc，定制功能头文件和定制功能模块参见 proc/single/dbfunc.h 和 proc/single/dbfunc.pc，编译连接方法参见 proc/single/Makefile。

8.3.2 单线程和多线程

由于嵌入式 SQL 语句被预编译成全局性的 C 数据结构和函数调用，所以在单线程条件下不会有任何执行问题，程序编写十分简单，参见 proc/single/。

而多线程条件就复杂很多，为了保证多个线程能够安全互斥地使用全局数据结构，必须给每个线程一个表明身份的上下文标志（context），在进行 sql 操作时彼此能区别开来。

每个线程使用一个数据库连接，凭借自己的 id 使用专有的数据结构，包括上下文标志（context）等，此数据结构是数据库连接数组（a_thrd_conn）的一个成员，参见 dbcom.h 中 thrd_conn_t 和 dbcom.c 中 a_thrd_conn。这样做的目的在于：统一存放连接有关数据，使线程引用变得简单。如下所示：



加入多线程支持

在 proc 的命令行或配置文件 pcscfg.cfg 中写入 threads=yes，表明为待编译的程序提供 thread-safe 功能，参见 proc/multi/Makefile。

```
proc thread=yes …
```

应用程序中最前面必须指明本程序支持多线程，参见 proc/multi/multi.c 和 proc/multi/dbcom.pc 中 DbsEnableThread。

```
EXEC SQL enable threads;
```

代码大致流程

每个线程使用线程编号（thrd_index）引用数据库连接数组。

在建立数据库连接的时候，分配上下文标志，参见 proc/multi/dbcom.pc 中 DbsConnectThread。

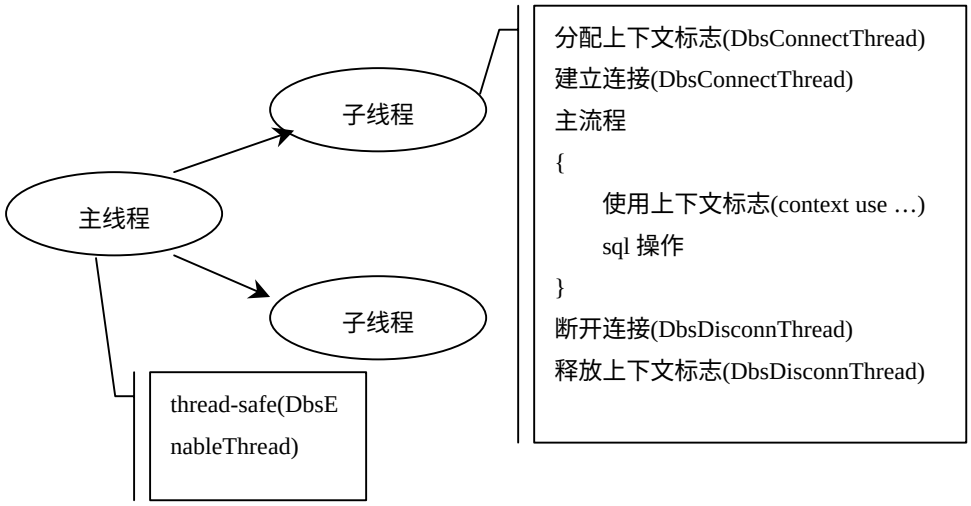
```
EXEC SQL context allocate :a_thrd_conn[thrd_index].context;
```

断开时，释放上下文标志，参见 DbsDisconnThread。

```
EXEC SQL context free :a_thrd_conn[thrd_index].context;
```

在每次 sql 操作前使用上下文标志（由于 proc 的一些原因，threads=yes 条件下的 sql 语句在被 proc 转换时使用上面离其**最近**的 use 所指明的 context，所以保险做法是每句 sql 语句前使用 use）。

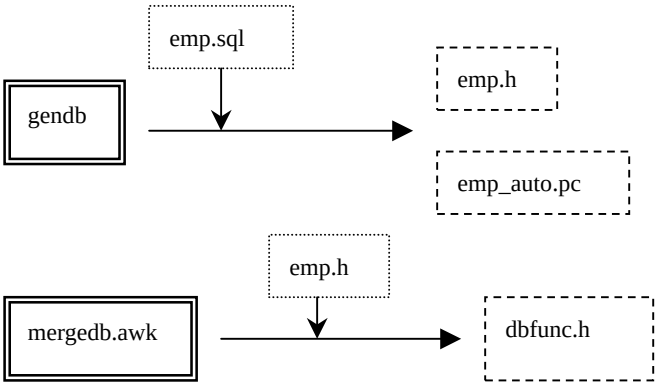
EXEC SQL context use : a_thrd_conn[thrd_index].context
EXEC SQL update emp set …;



8.3.3 开发工具

基于以下事实：宿主结构与表严格对应；基本 SQL 操作，如初始化宿主结构、SELECT(根据主键)、INSERT、DELETE(根据主键)、UPDATE(根据主键)等，流程非常单一。因此根据建表脚本生成宿主结构定义和基本 SQL 操作函数，可大大减少 SQL 操作函数的设计，提高开发速度。

工具由 perl 程序 gendb 和 awk 脚本 mergedb.awk 组成，由 shell 脚本 chgdb 组织流程。假定处理表 emp，gendb 根据表的创建脚本生成宿主结构定义（emp.h）和基本 SQL 操作函数（emp_auto.pc），mergedb.awk 将宿主结构定义融入宿主结构所在头文件（dbfunc.h）。所有的文件或表的名称都在 chgdb 中定义，可以需要根据实际情况改写 chgdb 中部分内容。



使用方法：chgdb 表名，如 chgdb emp
参见 chgdb

9 OCI—Oracle Call Interface

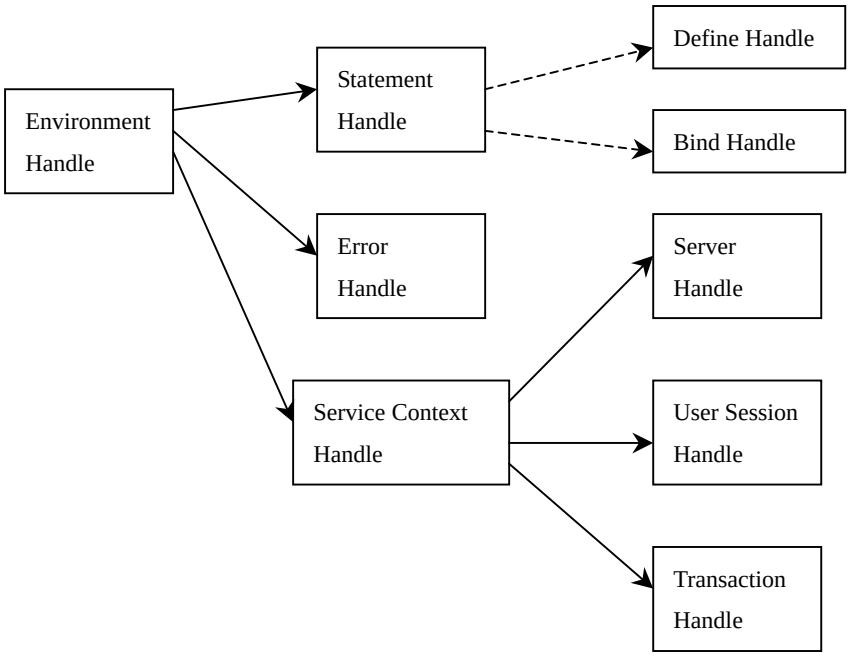
所有程序例子都在附件 09_oci/下

OCI 每个函数的参数都有很严格的类型定义，非常繁琐，下面的函数示例为了简洁说明起见，没有对参数类型作明确说明，也没有对错误进行处理，实际应用中要复杂得多。

9.1 连接和断开

9.1.1 句柄层次

OCI 使用各种句柄操作数据库，环境句柄（Environment Handle）是所有句柄的父句柄下面列出开发所需的基本句柄。



服务器句柄（Server Handle），用户会话句柄（User Session Handle），事务句柄（Transaction Handle）隶属于服务上下文句柄（Service Context Handle），但都是以环境句柄为父句柄分配的。

查询输出定位句柄（Define Handle）和输入输出绑定变量句柄（Bind Handle）在执行具体的 SQL 语句的时候，被隐含创建并连接到表达句柄（Statement Handle）上，当表达句柄释放时，它们也被隐含释放。所以在执行每一个 sql 命令时，先分配表达句柄，执行结束后，释放表达句柄，这样做保证不发生由于定位句柄和绑定变量句柄引起的内存泄漏。

9.1.2 连接流程

参见 dbcom.c 中 DbsConnect。

OCI 连接过程十分复杂，除了分配设置各基本句柄外，还要明确彼此之间的联系，大致流程如下：

创建环境：

OCIEnvCreate(&**handle_env**…);

分配错误句柄：

OCIHandleAlloc(handle_env,&**handle_error**,OCI_HTYPE_ERROR,…);

分配设置服务上下文句柄和服务句柄：

OCIHandleAlloc(handle_env,&**handle_service**, OCI_HTYPE_SVCCTX,…);

OCIHandleAlloc(handle_env,&**handle_server**, OCI_HTYPE_SERVER,…);

OCIServerAttach(**handle_server**, handle_error,**dblink**,…); //初始化服务器句柄，为连接做好准备，如果 dblink 为 NULL，则选择缺省数据库

OCIAttrSet(**handle_service**, OCI_HTYPE_SVCCTX, **handle_server**, 0, OCI_ATTR_SERVER, handle_error); //将服务器句柄连接到服务上下文句柄

分配设置会话句柄：

OCIHandleAlloc(handle_env,&**handle_session**, OCI_HTYPE_ SESSION,…);

OCIAttrSet(**handle_session**, OCI_HTYPE_SESSION, username, strlen(username), OCI_ATTR_USERNAME, handle_error); //向会话句柄填充用户名

OCIAttrSet(**handle_session**, OCI_HTYPE_SESSION, password, strlen(password), OCI_ATTR_PASSWORD, handle_error); //向会话句柄填充口令

建立会话连接：

OCISessionBegin(**handle_service**, handle_error, **handle_session**, OCI_CRED_RDBMS, OCI_DEFAULT); //发起连接

OCIAttrSet(**handle_service**, OCI_HTYPE_SVCCTX, **handle_session**, 0, OCI_ATTR_SESSION, handle_error);

9.1.3 断开流程

参见 dbcom.c 中 DbsDisConnect。

当父句柄被释放时，与之连接的所有子句柄也被释放。大致流程如下：

断开会话连接：

OCISessionEnd(**handle_service**, handle_error,**handle_session**, OCI_DEFAULT);

去除服务器句柄：

OCIServerDetach(**handle_server**, handle_error,OCI_DEFAULT);

释放环境句柄：

OCIHandleFree**(handle_env**, OCI_HTYPE_ENV);

9.2 SQL 语句

9.2.1 事务

提交：

OCITransCommit(**handle_service**,handle_error,OCI_DEFAULT);

回滚：

OCITransRollback(**handle_service**,handle_error,OCI_DEFAULT);

9.2.2 无结果集的 sql 语句

参见 dbfunc.c 中 DbsRESUME_INS。

指 insert,update,delete 语句，执行以后它们只是返回错误值，因此可以用统一的形式完成。流程如下：

如执行这样一条 sql 语句：“ insert into emp values(1,sysdate, ‘职员 1’,’1’,30,2000)”。

OCIHandleAlloc(handle_env,&**handle_stmt**, OCI_HTYPE_STMT,0,NULL); //分配表达句柄

strcpy(sql, “insert into emp values(1,sysdate, ‘职员 1’,’1’,30,2000)”);

OCIStmtPrepare(**handle_stmt**, handle_error, sql, strlen(sql), OCI_NTV_SYNTAX, OCI_DEFAULT); //解析 sql 语句

OCIStmtExecute(**handle_service**,**handle_stmt**,ThrdConn.p_herr,**1**, **0**,NULL, NULL,OCI_DEFAULT); //加亮的 1,0 表示两个参数 iters 和 rowoff,对于非 select 语句执行 iters-rowoff 次

OCIHandleFree**(handle_stmt**, OCI_HTYPE_STMT); //释放表达句柄

可以使用绑定变量的方式执行 sql，例如：

假定输入数据都在指针 p_emp 指向的结构中。

……

strcpy(sql, “insert into emp values(:no,sysdate,:name,:duty,:age,:salary)”);

OCIStmtPrepare(**handle_stmt**, handle_error, sql, strlen(sql), OCI_NTV_SYNTAX, OCI_DEFAULT);

OCIBindByName(**handle_stmt**, &a_handle_bind[0], handle_error,no,&p_emp->no,sizeof(p_emp->no),SQL_FLT,…);

OCIBindByName(**handle_stmt**, &a_handle_bind[1], handle_error,name,p_emp->name,sizeof(p_emp->name),SQL_STR,…);

OCIBindByName(**handle_stmt**, &a_handle_bind[2], handle_error,duty,p_emp->duty,sizeof(p_emp->duty),SQL_STR,…);

OCIBindByName(**handle_stmt**, &a_handle_bind[3], handle_error,age,&p_emp->age,sizeof(p_emp->age),SQL_INT,…);

OCIBindByName(**handle_stmt**, &a_handle_bind[4], handle_error,salary,&p_emp->salary,sizeof(p_emp->salary),SQL_FLT,…);

……

a_handle_bind 是一个数组，成员为 OCIBind *，上面说过，绑定变量句柄是绑定函数执行时隐含分配的，并隶属于表达句柄。对于定位句柄，也采取了相同的方式。

不过这种绑定的方法没有多大必要，因为可以直接把变量的值写作 sql 语句中，这个结论对下面的段落也有效，只不过为了描述的完整性，还是加入了绑定的内容。

9.2.3 有结果集的 sql 语句

选取

参见 dbfunc.h 中 DbsEMP_SEL。

为了作更好的区分，这里的“ 选取 ”定义为根据唯一约束得到一条记录。这样与上面无结果集的 sql 语句相比，只是多了对输出的定义。流程如下：

如执行这样一条 sql 语句：“ select no,upd_ts,name,duty,age,salary from emp where

no=:no ”，假定指针 p_emp 指向对应于表的宿主结构，输入条件在里面，输出也利用它。这句 sql 语句要求查询条件使用 p_emp->no，这就要把它绑定到输入变量上，并且要求输出 no,upd_ts,name,duty,age,salary 等域，所以必须对它们进行输出定位。

OCIHandleAlloc(handle_env,&**handle_stmt**, OCI_HTYPE_STMT,0,NULL); //分配表达句柄

strcpy(sql, “select no,upd_ts,name,duty,age,salary from emp where no=:no”);

OCIStmtPrepare(**handle_stmt**, handle_error, sql, strlen(sql), OCI_NTV_SYNTAX, OCI_DEFAULT); //解析 sql 语句

OCIBindByName(**handle_stmt**, &a_handle_bind[0], handle_error,no,&p_emp->no,sizeof(p_emp->no),SQL_FLT,…); //p_emp->no 被绑定到输入

OCIDefineByPos(**handle_stmt**,&a_handle_define[0],handle_error,**1**,&p_emp->no, sizeof(p_emp->no), SQL_FLT, …); //p_emp->no 同时可作输出定位

OCIDefineByPos(**handle_stmt**,&a_handle_define[1],handle_error,**2**,p_emp->upd_ts, sizeof(p_emp->upd_ts), SQL_STR, …);

OCIDefineByPos(**handle_stmt**,&a_handle_define[2],handle_error,**3**,p_emp->name,sizeof(p_emp->name), SQL_STR, …);

OCIDefineByPos(**handle_stmt**,&a_handle_define[3],handle_error,**4**,p_emp->duty,sizeof(p_emp->duty), SQL_STR, …);

OCIDefineByPos(**handle_stmt**,&a_handle_define[4],handle_error,**5**,&p_emp->age,sizeof(p_emp->age), SQL_INT, …);

OCIDefineByPos(**handle_stmt**,&a_handle_define[5],handle_error,**6**,&p_emp->salary,sizeof(p_emp->salary), SQL_FLT, …);

OCIStmtExecute(**handle_service**,**handle_stmt**,ThrdConn.p_herr,**1**, **0**,NULL, NULL,OCI_DEFAULT); //由于参数 iters 设为 1,所以输出定位必须在执行前完成

OCIHandleFree**(handle_stmt**, OCI_HTYPE_STMT); //释放表达句柄

游标

参见 09_oci/DbsEMP_CUR。

与上面“ 选取 ”相比，游标不强调查询条件的唯一性，而是处理成批的记录，这样就要有循环的结构和对输出区域的管理。流程如下：

如执行这样一条 sql 语句：“ select no,name,age from emp where age<:age order by no ”。

由于成批记录占用空间很难估计，所以在游标检索循环中根据实际情况分配和调整空间，回传给调用者的是指针。

strcpy(sql,” select no,name,age from emp where age<:age order by no”);

OCIStmtPrepare(**handle_stmt**, handle_error, sql, strlen(sql), OCI_NTV_SYNTAX, OCI_DEFAULT); //解析 sql 语句

OCIBindByName(**handle_stmt**, &a_handle_bind[0], handle_error,age,&p_emp->no,sizeof(p_emp->age),SQL_INT,…);

OCIStmtExecute(handle_service,handle_stmt,handle_error,**0**,0, NULL,NULL,OCI_DEFAULT); //对照上面的“ 选取 ”，由于不知道记录数量，所以将参数 iters 设为 0，仅仅打开游标，这也是执行前不进行输出定位的原因。

OCIDefineByPos(**handle_stmt**,&a_handle_define[0],handle_error,**1**,& emp.no, sizeof(emp.no), SQL_FLT, …);

OCIDefineByPos(**handle_stmt**,&a_handle_define[1],handle_error,**2**, emp.name,sizeof(emp.name), SQL_STR, …);

OCIDefineByPos(**handle_stmt**,&a_handle_define[2],handle_error,**3**,& emp.age,sizeof(emp->age), SQL_INT, …);

while(1)

{

OCIStmtFetch2(**handle_stmt**, handle_error, **1**,OCI_DEFAULT,0, OCI_DEFAULT); //加亮的 1 是参数 nrows，意思是一次选取的记录条数，没有特别的要求，一般只取一条。

如果记录已取完，跳出循环。

将暂存的记录 emp 转移到记录数组 a_emp 中去。

判断并调整 a_emp。

}

返回 a_emp 的指针和记录条数。

9.2.4 LOB

这里只描述了 blob 的操作情况，clob 的情况大同小异。blob 以字节为单位处理数据，不理会各种字符集的差异。

插入

参见 dbfunc.c 中 DbsRESUME_INS。

分为两步，第一步与通常插入操作类似，但以 sql 函数 empty_blob()代替类型为 blob 的域，流程如下：

sprintf(sql, "insert into resume values(:no,empty_blob())");

OCIStmtPrepare(**handle_stmt**, handle_error, sql, strlen(sql), OCI_NTV_SYNTAX, OCI_DEFAULT);

OCIBindByName(**handle_stmt**, &a_handle_bind[0], handle_error,no,&p_emp->no,sizeof(p_emp->no),SQL_FLT,…);

OCIStmtExecute(**handle_service**,**handle_stmt**,handle_error,**1**, **0**,NULL, NULL,OCI_DEFAULT);

第二步做一次选取，目的是将 blob 域与大对象定位句柄（OCILobLocate *handle_blob）连接起来，再把通过句柄把内容写到 blob 域中去，流程如下

sprintf(sql, "select resume from resume where no=:no");

OCIStmtPrepare(**handle_stmt**, handle_error, sql, strlen(sql), OCI_NTV_SYNTAX, OCI_DEFAULT);

OCIBindByName(**handle_stmt**, &a_handle_bind[0], handle_error,no,&p_emp->no,sizeof(p_emp->no),SQL_FLT,…);

OCIDefineByPos(**handle_stmt**,&a_handle_define[0],handle_error,**1**,& handle_blob,sizeof(handle_blob), SQL_BLOB, …);

OCIStmtExecute(**handle_service**,**handle_stmt**,handle_error,**1**, **0**,NULL, NULL,OCI_DEFAULT);

写 blob 公用函数（write_blob），参见 dbcom.c 中 write_blob。

为了不一次写入过多数据，首先定义一个缓冲区大小 LOB_BUF_LEN，比如规定为 8192。

如果写入数据大小（size）在 LOB_BUF_LEN 范围内，则可一次写完。

amt=size;

OCILobWrite(**handle_service**,handle_error,**p_blob**, &**amt**, **1**,write_buf, buf_len, **OCI_ONE_PIECE**,…);

//p_blob 是大对象定位句柄

//amt 输入要写的字节数，并输出实际写的字节数，当前情况下，两个数值一样

//加亮的 1 表示缓冲区开始写的位移，1 为起始位置

//OCI_ONE_PIECE 表示一次写完

如果写入数据超过 LOB_BUF_LEN 范围，必须分多次写入。

先写入第一批数据：

offset=1; **amt**=LOB_BUF_LEN; **remainder**=size; //offset 记录已写的位置，remainder 记录剩下的数据字节数

OCILobWrite(handle_service,handle_error,p_blob, &amt, offset,write_buf, buf_len,**OCI_FIRST_PIECE**,…); //OCI_FIRST_PIECE 表示写第一批

remainder-=amt; **offset**+=amt;

展开循环：

```
while(remainder>LOB_BUF_LEN)
{
    amt=LOB_BUF_LEN;  //每次写 LOB_BUF_LEN 大小的数据
    OCILobWrite(handle_service, handle_error,p_blob, &amt, offset,write_buf, buf_len, OCI_NEXT_PIECE,…);  //OCI_NEXT_PIECE 表示继续写，但不写最后一批
    remainder-=amt; offset+=amt;
}
```

写入最后一批数据：

amt=remainder;

OCILobWrite(handle_service,handle_error,p_blob, &amt, offset,write_buf, buf_len,OCI_LAST_PIECE,…); //OCI_LAST_PIECE 表示写最后一批数据

选取

参见 09_oci/DbsRESUME_SEL。

分为两步，第一步与通常选取操作类似，但对应 blob 域的定位句柄是大对象定位句柄，流程如下：

sprintf(sql, "select no,resume from resume where no=:no");

OCIStmtPrepare(**handle_stmt**, handle_error, sql, strlen(sql), OCI_NTV_SYNTAX, OCI_DEFAULT);

OCIBindByName(**handle_stmt**, &a_handle_bind[0], handle_error,no,&p_emp->no,sizeof(p_emp->no),SQL_FLT,…);

OCIDefineByPos(**handle_stmt**,&a_handle_define[0],handle_error,**1**,&p_emp->no, sizeof(p_emp->no), SQL_FLT, …);

OCIDefineByPos(**handle_stmt**,&a_handle_define[1],handle_error,**2**,&handle_blob, sizeof(handle_blob), SQL_BLOB, …); //handle_blob 必须事先分配

OCIStmtExecute(**handle_service**,**handle_stmt**,ThrdConn.p_herr,**1**, **0**,NULL, NULL,OCI_DEFAULT);

第二步通过大对象定位句柄调用读 blob 公用函数（read_blob），参见 *dbcom.c 中 read_blob*。

取得 blob 域数据长度（size）：

OCILobGetLength(handle_service,handle_error,p_blob, &size)

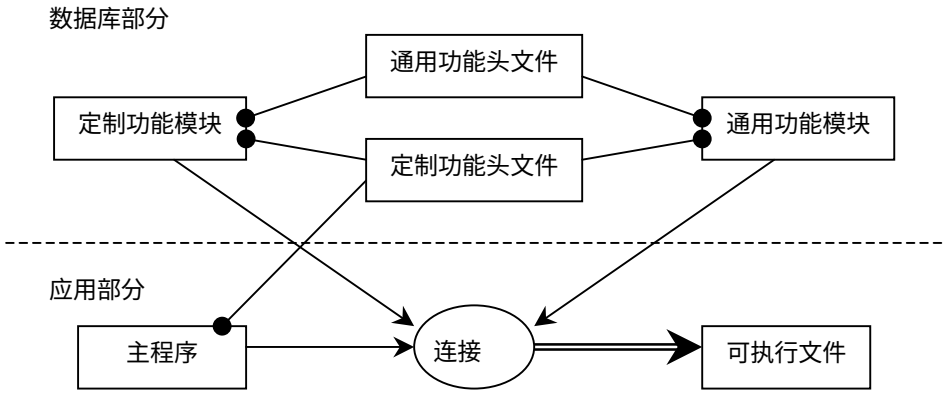
展开读取循环：

```
remainder=size; offset=1;
while(1)
{
    amt=0;   //输入要写的字节数为 0,使用流模式（streamed mode）读取
    OCILobRead(handle_service,handle_error,p_blob,&amt,1,read_buf+offset-1,
(ub4)LOB_BUF_LEN,NULL,NULL …);
    //加亮的 1 表示第一次读的位移，在流模式下，此参数和 amt 一样，只有在第一次读时有效
    //加亮的 NULL 表示回调函数为空，这样流模式使用轮询（polling）方法而不是回调（callback）方法
   //轮询方法下，如果没有读到 blob 末尾，函数返回 OCI_NEED_DATA，读取完毕，返回 OCI_SUCCESS
    如果读取完毕（OCI_NEED_DATA），跳出循环
    offset+=amt;   remainder-=amt;
}
read_buf[size]='\0';
```

9.3 编程框架

9.3.1 总体原则

为了使程序模块划分更加简洁清晰，提高应用程序的可移植性，所以采取将主程序模块和数据库功能模块分离的方法，即不在主程序逻辑中出现 OCI 函数调用，而把这些调用归并到数据库功能模块中去，对主程序而言，展现的是某种功能。同时在数据库模块上再进一步把通用功能和定制功能分割开来。大致框架如下：

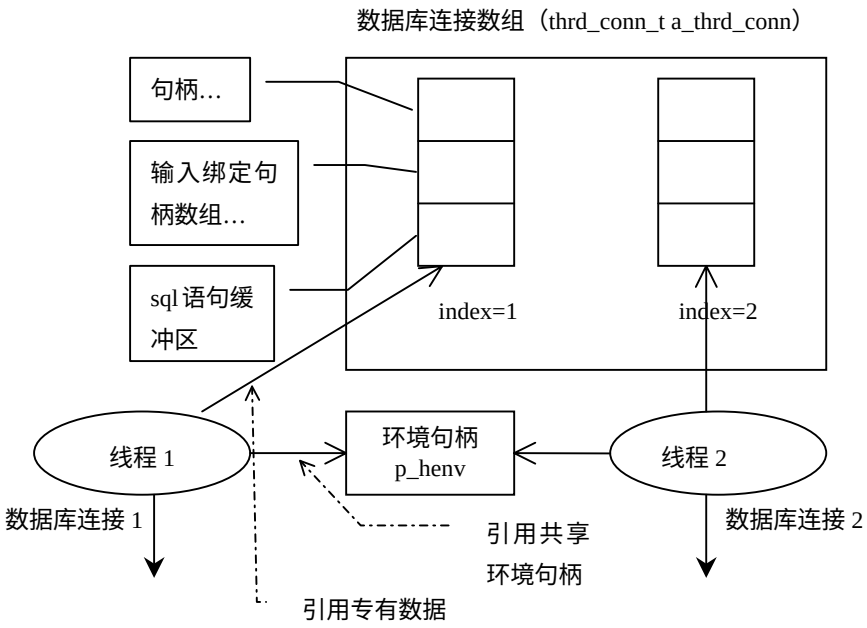


通用功能头文件和通用功能模块参见 dbcom.h 和 dbcom.c，定制功能头文件和定制功

能模块参见 dbfunc.h 和 dbfunc.c。

支持多线程编程，线程共享一个环境句柄（测试中发现环境句柄分配超出 5 个就会出错，因此采用共享方式），参见 dbcom.h 中 p_henv 的定义。

每个线程使用一个数据库连接，凭借自己的 id 使用专有的数据结构，包括错误句柄、上下文句柄、服务器句柄、会话句柄、表达句柄、输入绑定句柄数组、输出定位句柄数组、sql 语句缓冲区等，此数据结构是数据库连接数组（a_thrd_conn）的一个成员，参见 dbcom.h 中 thrd_conn_t 和 dbcom.c 中 a_thrd_conn。这样做的目的在于：避免在每个 sql 函数处作繁琐的定义，通过一些简化的写法，使编程更简洁。如下所示：



9.3.2 sql 语句

参见 dbcom.h。

由于 OCI 函数参数及其繁琐，而使用方式又比较单一，加之引用规范化的数据结构，所以在选取规范化的函数调用的基础上，可用宏对其进行大幅度简化。

首先定义的宏是连接对数据库连接数组的引用：

#define ThrdConn a_thrd_conn[thrd_index]　//thrd_index 是每个连接的编号

prepare

OCIStmtPrepare(ThrdConn.p_hstmt, ThrdConn.p_herr, ThrdConn.sql, (ub4)strlen((char *)ThrdConn.sql), (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT)

简化为 StmtPrepare。

execute

OCIStmtExecute(ThrdConn.p_hsvc, ThrdConn.p_hstmt, ThrdConn.p_herr, (ub4)1, (ub4)0, (CONST OCISnapshot *)NULL, (OCISnapshot *)NULL, OCI_DEFAULT)

简化为 StmtExecute。

open

OCIStmtExecute(ThrdConn.p_hsvc, ThrdConn.p_hstmt,ThrdConn.p_herr, (ub4)0, (ub4)0,(CONST OCISnapshot *)NULL, (OCISnapshot *)NULL,OCI_DEFAULT)

简化为 StmtOpen，注意与 execute 的不同。

fetch

OCIStmtFetch2(ThrdConn.p_hstmt, ThrdConn.p_herr,(ub4)1, (ub2)OCI_DEFAULT, (sb4)0, (ub4)OCI_DEFAULT)

简化为 StmtFetch

define

按输出顺序将输出定位句柄联系至实体缓冲区。

OCIDefineByPos(ThrdConn.p_hstmt, &ThrdConn.a_hdef[Pos-1], ThrdConn.p_herr, (ub4)Pos, (dvoid *)Host, (sb4)Size, (ub2)Type,(dvoid *)NULL, (ub2 *)NULL, (ub2 *)NULL, OCI_DEFAULT)

简化为 DefinePos(Pos,Host,Size,Type)

bind

将宿主变量名（即 sql 语句中:col）将输入绑定句柄联系到实体缓冲区。

OCIBindByName(ThrdConn.p_hstmt, &ThrdConn.a_hbnd[Order-1],ThrdConn.p_herr,(CONST text *)":"#Name, (sb4)-1,(dvoid *)Host, (sb4)Size, (ub2)Type,(dvoid *)NULL, (ub2 *)NULL, (ub2 *)NULL,(ub4)0, (ub4 *)NULL, OCI_DEFAULT)

简化为 BindName(Order,Name,Host,Size,Type)

按输入宿主变量顺序将输入绑定句柄联系到实体缓冲区。

OCIBindByPos(ThrdConn.p_hstmt, &ThrdConn.a_hbnd[Pos-1],ThrdConn.p_herr, (ub4)Pos, (dvoid *)Host, (sb4)Size, (ub2)Type, (dvoid *)NULL, (ub2 *)NULL, (ub2 *)NULL,(ub4)0, (ub4 *)NULL, OCI_DEFAULT)

简化为 BindPos(Pos,Host,Size,Type)

9.3.3 函数

基于以上的简化，可以将 sql 操作归到几个简单的标准化函数模式中去。函数至少必须拥有两个参数：数据库连接编号（int thrd_indx）和 sql 语句（char *sql）。函数体定义部分必须使用 DefaultDefinition，程序部分头尾分别使用 StartFunc 和 EndFunc。

无结果集的 sql

参见 dbcom.c 中 DbsSQLExec。

在 9.2.2 中已经提到，输入绑定是没必要的，这样象 update、delete、insert 的操作都可以用一个统一的执行函数就能解决。

```
int DbsSQLExec(int thrd_index, char *sql)
{
    strcpy(ThrdConn.sql, sql);  //之所以多复制了一次，是因为照顾到各定义的通用性
    StmtPrepare;
    StmtExecute;
}
```

选取一条记录

参见 dbfunc.c 中 DbsEMP_SEL。

输入绑定和输出定位由于情况复杂不能省略，因此选取必须用单独定制的 sql 函数。建议在应用程序中先定义一个对应表结构的宿主结构，将其地址传入 sql 函数，可以将这个宿主结构同时作为输入条件和输出数据的缓冲区。

```
int DbsEMP_SEL(int thrd_index, char *sql, emp_t *p_emp)
{
    strcpy(ThrdConn.sql, sql);
    StmtPrepare;
    BindName(…);
    DefinePos(…);
    StmtExecute;
}
```

游标

游标的流程大致如下：

```
解析 sql 语句
绑定输入
打开游标
定位输出
分配输出空间
展开循环
{
    读取游标
    如已读完，跳出循环
    检查输出空间，如已满，则分配更多的空间
}
返回输出空间的引用
```

参见 dbcom.h，dbfunc.h 中 DbsEMP_CUR。

前 4 步操作由于情况多变，不能成为标准化过程，而后面的步骤则可成为一种模式，定义如下：

```
#define ProcCursor(Table) \
do \
{ \
    InitCurBuf(Table); \   //分配输出空间
    while(1) \
    { \
        CheckErr(StmtFetch); \   //读取游标
        if(status!=OCI_SUCCESS && status!=OCI_NO_DATA) \
            goto ERROR; \
        if(status==OCI_NO_DATA)   break; \   //如已读完，跳出循环
        CurBufLast(Table)=Table; \
        ExpandCurBuf(Table); \   //检查输出空间
    } \
```

```
        *pa_##Table=CurBuf(Table); \   //返回输出空间的引用
        *p_##Table##_qt=Table##_real_qt; \
    } while(0);
```

为了实现流程的标准化，定义使用了一些保留字，假定 Table 为 emp，输出空间指针 **a_emp**，辅助变量 **emp_alloc_qt** 和 **emp_real_qt**，编译时被自动定义出来。同时要求游标所在的函数传入输出空间指针地址（emp_t **pa_emp）和记录条数变量地址（int *p_emp_qt）。

注意加亮的一行，因为输出定位在循环开始之前必须联系到某个结构实体，所以这里利用了那个实体，作了一次转存。这个实体在 DefCurBuf(Table)里定义，名字即为表名。

与“选取”类似，建议使用宿主结构指针传入输入条件。

```
int DbsEMP_CUR(int thrd_index,char *sql,emp_t *p_emp,emp_t **pa_emp,int *p_emp_qt)
{
    DefCurBuf(emp);  //定义输出空间指针和几个辅助变量
    strcpy(ThrdConn.sql, sql);
    StmtPrepare;
    BindName(…);
    StmtOpen;
    DefinePos(1, &emp.no, sizeof(emp.no), SQLT_FLT));
    ProcCursor(emp);
}
```

10 附录—MYSQL

MYSQL 作为一个小巧、灵活、高效的开源数据库，可以作为象 Oracle 这样的大型商业数据库的有益补充。并不是所有场合都需要 Oracle 等数据库的高可靠性和高可扩展性，没有必要盲目支付昂贵的软件费用。所以在本附录里作一点介绍。

10.1 安装配置

笔者选择的安装平台是 RedHat Linux 7.3，在 mysql 的网站www.mysql.com里有 rpm 格式的安装文件，版本号 3.23.52 的标准版，包括 Server、Benchmark/test suites、Client programs、Libraries and Header files for development、Client shared libraries 等数个 rpm 文件。

以 root 用户登录，依次安装 rpm 包，大致顺序先装库，再装 server，client。

安装结束后，mysql 已经启动。下面对数据库作一些调整。

以 root 用户登录。

关闭数据库：

mysqladmin shutdown -uroot

mysql 的启动脚本是/usr/bin/safe_mysqld，打开它，找到 MY_BASEDIR_VERSION, DATADIR,ledir 所在的区域，这是一个 if … then … elif … then … else … fi 的结构，到最后一个 else 块，修改这三个变量为：

MY_BASEDIR_VERSION=/

DATADIR=/opt/mysql #数据库所在目录

ledir=/usr/sbin #mysql 系统程序 mysqld 所在目录

原来的 DATADIR 为/var/lib/mysql，肯定要根据实际情况改变的。

将/var/lib/mysql/下的 mysql 目录复制到新的 DATADIR 即/opt/mysql 下。

在/usr/share/doc/packages/MySQL 下能找到 my.cnf 的不同配置，选一个符合实际情况的配置，复制成/etc/my.cnf。

启动数据库：

safe_mysqld &

mysql 在安装过程中产生/etc/init.d/mysql，设置各/etc/rc.d 对它的符号连接，这样 mysql 的启动关闭可以让操作系统自动完成。

10.2 管理

10.2.1 初始调整

第一次进入 mysql，没有任何验证。

以 root 用户登录 mysql：

mysql -uroot

选择数据库 mysql，即 mysql 的系统数据库：

mysql>use mysql

操作数据字典，重新设置 root 用户口令：

在数据库 mysql 中，db,user 和 host 构成了最基本的数据字典，主要功能为检查数据库连接的有效性。

mysql>delete from user where user<>’root’;

mysql>update user set password=password(‘mysql’) where user=’root’;

建立新数据库：

mysql>create database data;

去除数据库：drop database data;

建立普通用户 dbuser，赋予权限：

mysql>insert into user (host,user,password) values(‘%’,’dbuser’,password(‘mysql’));

#表名和域列表之间一定要有空格，如 user（…）

mysql>grant select,insert,update,delete,create,drop,index,alter on data.* to dbuser;

重载 mysql 系统信息：

mysql>exit

mysqladmin reload –uroot

从此以后，对 root 用户的验证已经建立起来，这样使用 mysqladmin 的操作必须加上口令验证：

mysqladmin shutdown –uroot –pmysql

mysqladmin reload –uroot -pmysql

10.2.2 建立用户对象

登录：

mysql data –udbuser –pmysql

mysql>show databases; #显示现有的数据库

mysql>show tables; #显示本数据库中的表

mysql>show columns from emp; #显示表结构

mysql>desc emp; #与上面相同

mysql>show index from emp; #显示表所有的索引

mysqll>show status; #显示系统配置信息

mysql>show table status; #显示表状态

mysql>show grants for dbuser; #显示用户所有的权限

建立表和索引：

mysql>create table emp(no numeric(12) not null,upd_ts timestamp not null,…);

mysql>drop table emp;

mysql>alter table emp add primary key (no);

mysql>alter table drop primary key;

mysql>alter table emp add index emp_x01 (name);

mysql>alter table drop index emp_x01;

也可以将 sql 语句写成一个脚本，重定向到 mysql 中去：

mysql data –udbuser –pmysql <db.sql

10.3 开发

参见附件 10_mysql/

10.3.1 连接和断开

首先定义数据库连接的数据结构 **mysql** 以及衍生的连接句柄 **sock**（参见 dbcom.h）：

MYSQL **mysql**;

MYSQL ***sock**;

为了简化编程，将本连接的 **res**（资源句柄），**row**（域指针数组）和 **sql**（sql 语句缓冲区）也一起定义好：

```
MYSQL_RES *res;
MYSQL_ROW row;
char *sql;
```

建立连接（参见 dbcom.c 中 DbsConnect）：

```
mysql_init(&mysql);
sock=mysql_real_connect(&mysql, host, username, password, dbname, 0, NULL, 0));
```

如果是本机，则 host 为 NULL。

username （数据库用户名） ,password （口令） ,dbname （数据库名）是”dbuser”,”mysql”,”data”这样的组合。

```
sql=(char *)malloc(…);
```

断开连接（参见 dbcom.c 中 DbsDisConnect）：

```
mysql_close(sock);
free(sql);
```

10.3.2 无结果集的 sql 语句

参见 dbfunc.c 中 DbsRESUME_INS。

指 insert,update,delete 语句，执行以后它们只是返回错误值，因此可以用统一的形式完成。

```
strcpy(sql, “insert into emp values(…)”);
mysql_query(sock, sql)；
```

10.3.3 有结果集的 sql

指 select 语句，为简化编程，这里分两种情况：根据 unique 约束的 select 和根据一般条件的 select，前者可以写得更简洁一些。笔者把前者称之为 “ 选取 ”，后者为 “ 游标 ”。

选取：

参见 dbfunc.c 中 DbsEMP_SEL。

```
strcpy(sql, “select name from emp where no=1”);
mysql_query(sock, sql);
res=mysql_store_result(sock);  #将结果集从 mysql 服务端读取到客户端
row=mysql_fetch_row(res);  #获取本记录域指针
strcpy(name,row[0]);  #根据选取域位置复制到用户数据结构中
mysql_free_result(res);  #释放记录内容
```

游标：

参见 dbfunc.c 中 DbsEMP_CUR。

```
strcpy(sql, “select name from emp where age>20”);
mysql_query(sock, sql);
res=mysql_use_result(sock);
count=0;
```

展开游标循环

```
{
        row=mysql_fetch_row(res);
        如果已经选完，跳出循环;
        strcpy(name[count],row[0]);
        count++;
}
mysql_free_result(res);
```

10.3.4 错误处理

参见 dbcom.c 中 checkerr。

根据文档判断 mysql 函数调用是否错误。

mysql_errno(sock)指明错误码。

mysql_error(sock)显示错误说明文本。

```
        unsigned int result=mysql_errno(sock);
        if(result)
        {
            printf("errno:%d errmsg:%s\n", result, mysql_error(sock));
        }
        return result;
```

有些函数的返回值可能代表正确也可能代表错误，此时必须进行错误判断，如 mysql_fetch_row 取完最后一条记录后返回的是 NULL，而出错也返回 NULL，前者为正常情况。

注意 mysql 错误码处理与其它数据库的不同，如 mysql_fetch_row，取完记录并不产生“记录已取完”的错误信息，而是产生“成功”信息。