

# Oracle 内存全面分析

作者: fuyuncat

来源: <http://WWW.HelloDBA.COM>

## 作者简介

黄玮, 男, 99年开始从事 DBA 工作, 有多年的水利、军工、电信及航运行业大型数据库 Oracle 开发、设计和维护经验。

曾供职于南方某著名电信设备制造商——H 公司。期间, 作为 DB 组长, 负责设计、开发和维护彩铃业务的数据库系统。目前, H 公司的彩铃系统是世界上终端用户最多的彩铃系统。最终用户数过亿。

目前供职于某世界著名物流公司, 负责公司的电子物流系统的数据库开发、维护工作。

msn: fuyuncat@hotmail.com

Email: fuyuncat@gmail.com

Oracle 的内存配置与 oracle 性能息息相关。而且关于内存的错误（如 4030、4031 错误）都是十分令人头疼的问题。可以说, 关于内存的配置, 是最影响 Oracle 性能的配置。内存还直接影响到其他两个重要资源的消耗: CPU 和 IO。

首先, 看看 Oracle 内存存储的主要内容是什么:

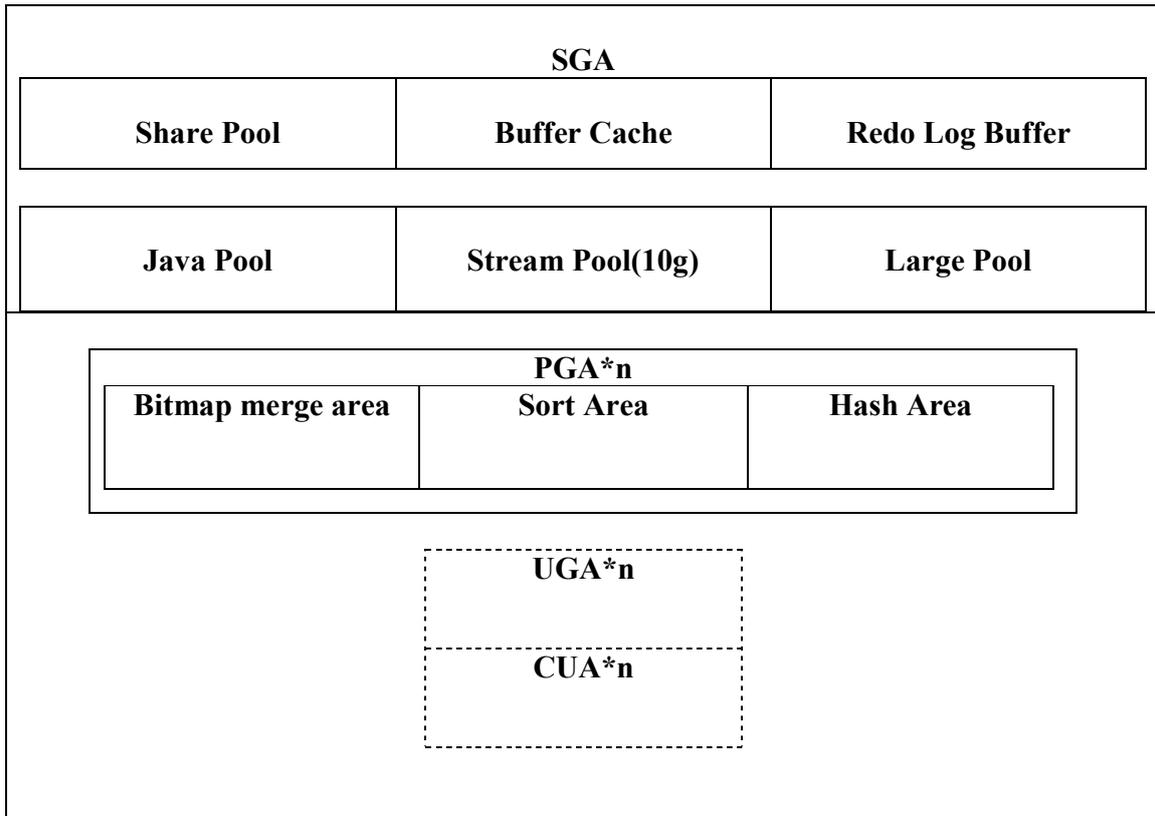
- 程序代码 (PLSQL、Java);
- 关于已经连接的会话的信息, 包括当前所有活动和非活动会话;
- 程序运行时必须的相关信息, 例如查询计划;
- Oracle 进程之间共享的信息和相互交流的信息, 例如锁;
- 那些被永久存储在外围存储介质上, 被 cache 在内存中的数据 (如 redo log 条目, 数据块)。

此外, 需要记住的一点是, Oracle 的内存是与实例对应的。也就是说, 一个实例就有一个独立的内存结构。

先从 Oracle 内存的组成架构介绍。

## 1. Oracle 的内存架构组成

Oracle 的内存, 从总体上讲, 可以分为两大块: 共享部分 (主要是 SGA) 和进程独享部分 (主要是 PGA 和 UGA)。而这两部分内存里面, 根据功能不同, 还分为不同内存池 (Pool) 和内存区 (Area)。下面就是 Oracle 内存构成框架图:



下面分别介绍这两块内存区。

### 1.1. SGA (System Global Area)

SGA (System Global Area 系统全局区域) 是一组包含一个 Oracle 实例的数据和控制信息的共享内存结构。这句话可以说是 SGA 的定义。虽然简单，但其中阐述了 SGA 几个很重要的特性：1、SGA 的构成——数据和控制信息，我们下面会详细介绍；2、SGA 是共享的，即当有多个用户同时登录了这个实例，SGA 中的信息可以被它们同时访问（当涉及到互斥的问题时，由 latch 和 enqueue 控制）；3、一个 SGA 只服务于一个实例，也就是说，当一台机器上有多个实例运行时，每个实例都有一个自己的 SGA，尽管 SGA 来自于 OS 的共享内存区，但实例之间不能相互访问对方的 SGA 区。

Oracle 进程和一个 SGA 就构成了一个 Oracle 实例。当实例启动时，Oracle 会自动从系统中分配内存给 SGA，而实例关闭时，操作系统会回收这些内存。下面就是当实例启动后，显示已经分配了 SGA：

```
SQL> startup
ORACLE instance started.

Total System Global Area  289406976 bytes
Fixed Size                  1248576 bytes
Variable Size              117441216 bytes
Database Buffers          163577856 bytes
Redo Buffers                7139328 bytes
Database mounted.
```

```
Database opened.
```

```
SQL>
```

SGA 区是可读写的。所有登录到实例的用户都能读取 SGA 中的信息，而在 oracle 做执行操作时，服务进程会将修改的信息写入 SGA 区。

SGA 主要包括了以下的数据结构：

- 数据缓冲 (Buffer Cache)
- 重做日志缓冲 (Redo Log Buffer)
- 共享池 (Shared Pool)
- Java 池 (Java Pool)
- 大池 (Large Pool)
- 流池 (Streams Pool --- 10g 以后才有)
- 数据字典缓存 (Data Dictionary Cache)
- 其他信息 (如数据库和实例的状态信息)

最后的两种内存信息会被实例的后台进程所访问，它们在实例启动后就固定在 SGA 中了，而且不会改变，所以这部分又称为固定 SGA (Fixed SGA)。这部分区域的大小一般小于 100K。

此外，用于并非进程控制的锁 (latch) 的信息也包含在 SGA 区中。

Shared Pool、Java Pool、Large Pool 和 Streams Pool 这几块内存区的大小是相应系统参数设置而改变的，所以有通称为可变 SGA (Variable SGA)。

### 1.1.1. SGA 的重要参数和特性

在设置 SGA 时，有一些很重要的参数，它们设置正确与否，会直接影响到系统的整体性能。下面一一介绍他们：

#### • SGA\_MAX\_SIZE

SGA 区包括了各种缓冲区和内存池，而大部分都可以通过特定的参数来指定他们的大小。但是，作为一个昂贵的资源，一个系统的物理内存大小是有限的。尽管对于 CPU 的内存寻址来说，是无需关系实际的物理内存大小的 (关于这一点，后面会做详细的介绍)，但是过多的使用虚拟内存导致 page in/out，会大大影响系统的性能，甚至可能会导致系统 crash。所以需要有一个参数来控制 SGA 使用虚拟内存的最大大小，这个参数就是 SGA\_MAX\_SIZE。

当实例启动后，各个内存区只分配实例所需要的最小大小，在随后的运行过程中，再根据需要扩展他们的大小，而他们的总和大小受到了 SGA\_MAX\_SIZE 的限制。

当试图增加一个内存的大小，并且如果这个值导致所有内存区大小总和大于 SGA\_MAX\_SIZE 时，oracle 会提示错误，不允许修改。

当然，如果在设置参数时，指定区域为 spfile 时 (包括修改 SGA\_MAX\_SIZE 本身)，是不会受到这个限制的。这样就可能出现这样的情况，在 spfile 中，SGA 各个内存区设置大小总和大于 SGA\_MAX\_SIZE。这时，oracle 会如下处理：当实例再次启动时，如果发现 SGA 各个内存总和大于 SGA\_MAX\_SIZE，它会将 SGA\_MAX\_SIZE 的值修改为 SGA 各个内存区总和的值。

SGA 所分配的是虚拟内存，但是，在我们配置 SGA 时，一定要使整个 SGA 区都在物理内存中，否则，会导致 SGA 频繁的页入/页出，会极大影响系统性能。

对于 OLTP 系统，我个人建议可以如下配置 SGA\_MAX\_SIZE（一般有经验的 DBA 都会有自己的默认配置大小，你也可以通过一段时间的观察、调整自己的系统来得到适合本系统的参数配置）：

系统内存	SGA_MAX_SIZE 值
1G	400-500M
2G	1G
4G	2500M
8G	5G

SGA 的实际大小可以通过以下公式估算：

SGA 实际大小 = DB\_CACHE\_SIZE + DB\_KEEP\_CACHE\_SIZE + DB\_RECYCLE\_CACHE\_SIZE + DB\_nk\_CACHE\_SIZE + SHARED\_POOL\_SIZE + LARGE\_POOL\_SIZE + JAVA\_POOL\_SIZE + STREAMS\_POOL\_SIZE（10g 中的新内存池） + LOG\_BUFFERS+11K (Redo Log Buffer 的保护页) + 1MB + 16M (SGA 内部内存消耗，适合于 9i 及之前版本)

公式种涉及到的参数在下面的内容种会一一介绍。

## • PRE\_PAGE\_SGA

我们前面提到，oracle 实例启动时，会只载入各个内存区最小的大小。而其他 SGA 内存只作为虚拟内存分配，只有当进程 touch 到相应的页时，才会置换到物理内存中。但我们也许希望实例一启动后，所有 SGA 都分配到物理内存。这时就可以通过设置 PRE\_PAGE\_SGA 参数来达到目的了。

这个参数的默认值为 FALSE，即不将全部 SGA 置入物理内存中。当设置为 TRUE 时，实例启动会将全部 SGA 置入物理内存中。它可以使实例启动达到它的最大性能状态，但是，启动时间也会更长（因为为了使所有 SGA 都置入物理内存中，oracle 进程需要 touch 所有的 SGA 页）。

我们可以通过 TopShow 工具（本站原创工具，可在 <http://www.HelloDBA.com/Download/TopShow.html> 中下载）来观察 windows（Unix 下的内存监控比较复杂，这里暂不举例）下参数修改前后的对比。

PRE\_PAGE\_SGA 为 FALSE:

```
SQL> show parameter sga

NAME                                TYPE                                VALUE
-----                                -
lock_sga                            boolean                             FALSE
pre_page_sga                         boolean                             FALSE
sga_max_size                         big integer                         276M
sga_target                           big integer                         276M
SQL> startup force
ORACLE instance started.

Total System Global Area  289406976 bytes
Fixed Size                1248576 bytes
Variable Size             117441216 bytes
Database Buffers         163577856 bytes
Redo Buffers              7139328 bytes
Database mounted.
Database opened.
SQL>
```

启动后，Oracle 的内存情况

Processes:

Name	Running Time	Kernel CPU(...)	User CPU( 2...)	MEM (722M)	Page File (...
ORACLE.EXE	2d 23h 59m 51s	0.00	0.00	168	340
WINWORD.EXE	5d 0h 16m 47s	0.00	0.00	43	48
iexplore.exe	3d 22h 44m 39s	0.00	0.00	40	41
Rtvscan.exe	5d 0h 57m 57s	0.00	0.00	33	23
iexplore.exe	2d 18h 51m 26s	0.00	0.00	32	29
services.exe	5d 1h 0m 17s	0.00	0.00	29	4
svchost.exe	5d 1h 0m 10s	0.00	0.00	26	17
MsnMsgr.Exe	5d 0h 58m 45s	0.00	0.00	20	22
OUTLOOK.EXE	4d 19h 29m 40s	0.00	0.78	20	58
plsqldev.exe	5d 0h 50m 2s	0.00	0.00	19	38
Explorer.EXE	5d 0h 59m 34s	0.00	0.00	15	19
Communicator.exe	5d 0h 58m 51s	0.00	0.00	14	25
lsass.exe	5d 1h 0m 17s	0.00	0.00	13	6
iexplore.exe	5d 0h 55m 18s	0.78	0.00	13	24
ccApp.exe	5d 0h 59m 4s	0.00	0.00	11	6
tg4msql.exe	5d 0h 55m 17s	0.00	0.00	11	8
tg4msql.exe	3d 21h 13m 17s	0.00	0.00	11	8

Threads:

TID	Name	Running Time	Kernel CPU(...)	User CPU(P...
5304		2d 23h 59m 51s	0.00	0.00
5584		2d 23h 59m 50s	0.00	0.00
5644		2d 23h 59m 50s	0.00	0.00
1300		0d 0h 0m 8s	0.00	0.00
2432		0d 0h 0m 40s	0.00	0.00
3656		0d 0h 0m 40s	0.00	0.00
4188		0d 0h 0m 40s	0.00	0.00
4832		0d 0h 0m 40s	0.00	0.00
2040		0d 0h 0m 40s	0.00	0.00
1156		0d 0h 0m 40s	0.00	0.00
4948		0d 0h 0m 40s	0.00	0.00
2760		0d 0h 0m 40s	0.00	0.00

可以看到，实例启动后，oracle 占用的物理内存只有 168M，远小于 SGA 的最大值 288M（实际上，这部分物理内存中还有一部分进程的 PGA 和 Oracle Service 占用的内存），而虚拟内存则为 340M。

将 PRE\_PAGE\_SGA 修改为 TRUE，重启实例：

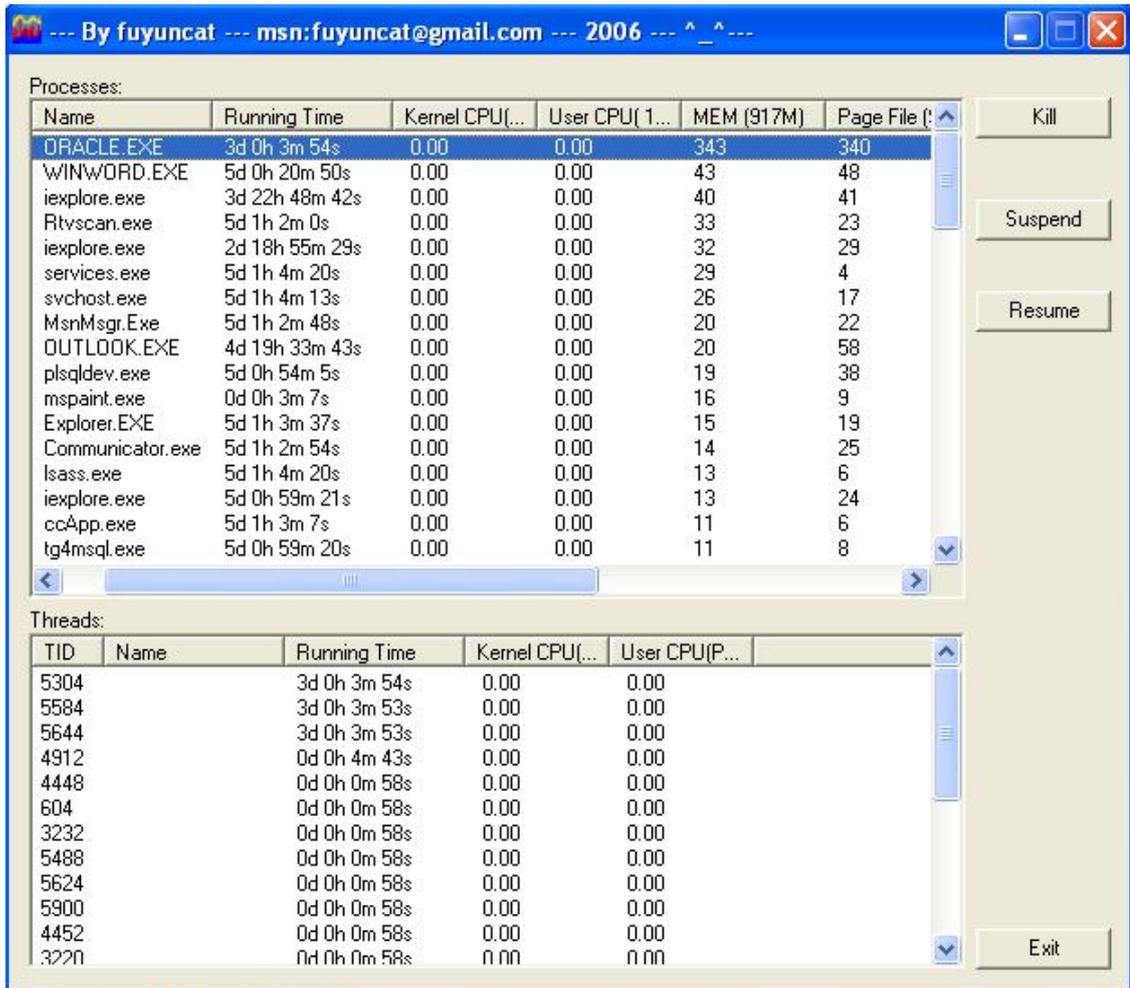
```
SQL> alter system set pre_page_sga=true scope=spfile;

System altered.

SQL> startup force
ORACLE instance started.

Total System Global Area  289406976 bytes
Fixed Size                  1248576 bytes
Variable Size               117441216 bytes
Database Buffers            163577856 bytes
Redo Buffers                 7139328 bytes
Database mounted.
Database opened.
```

再观察启动后 Oracle 的内存分配情况：



这时看到，实例启动后物理内存达到了最大 343M，于虚拟内存相当。这时，oracle 实例已经将所有 SGA 分配到物理内存。

当参数设置为 TRUE 时，不仅在实例启动时，需要 touch 所有的 SGA 页，并且由于每个 oracle 进程都会访问 SGA 区，所以每当一个新进程启动时（在 Dedicated Server 方式中，每个会话都会启动一个 Oracle 进程），都会 touch 一遍该进程需要访问的所有页。因此，每个进程的启动时间页增长了。所以，这个参数的设置需要根据系统的应用情况来设定。

在这种情况下，进程启动时间的长短就由系统内存的页的大小来决定。例如，SGA 大小为 100M，当页的大小为 4K 时，进程启动时需要访问  $100000/4=25000$  个页，而如果页大小为 4M 时，进程只需要访问  $100/4=25$  个页。页的大小是由操作系统指定的，并且是无法修改的。

但是，要记住一点：PRE\_PAGE\_SGA 只是在启动时将物理内存分配给 SGA，但并不能保证系统在以后的运行过程不会将 SGA 中的某些页置换到虚拟内存中，也就是说，尽管设置了这个参数，还是可能出现 Page In/Out。如果需要保障 SGA 不被换出，就需要由另外一个参数 LOCK\_SGA 来控制了。

## • LOCK\_SGA

上面提到，为了保证 SGA 都被锁定在物理内存中，而不必页入/页出，可以通过参数 LOCK\_SGA 来控制。这个参数默认值为 FALSE，当指定为 TRUE 时，可以将全部 SGA 都锁定在物理内存中。当然，有些系统不支持内存锁定，这个参数也就无效了。

## • SGA\_TARGET

这里要介绍的时 Oracle10g 中引入的一个非常重要的参数。在 10g 之前，SGA 的各个内存区的大小都需要通过各自的参数指定，并且都无法超过参数指定大小的值，尽管他们之和可能并没有达到 SGA 的最大限制。此外，一旦分配后，各个区的内存只能给本区使用，相互之间是不能共享的。拿 SGA 中两个最重要的内存区 Buffer Cache 和 Shared Pool 来说，它们两个对实例的性能影响最大，但是就有这样的矛盾存在：在内存资源有限的情况下，某些时候数据被 cache 的需求非常大，为了提高 buffer hit，就需要增加 Buffer Cache，但由于 SGA 有限，只能从其他区“抢”过来——如缩小 Shared Pool，增加 Buffer Cache；而有时又有大块的 PLSQL 代码被解析驻入内存中，导致 Shared Pool 不足，甚至出现 4031 错误，又需要扩大 Shared Pool，这时可能又需要人为干预，从 Buffer Cache 中将内存夺回来。

有了这个新的特性后，SGA 中的这种内存矛盾就迎刃而解了。这一特性被称为自动共享内存管理 (Automatic Shared Memory Management ASMM)。而控制这一特性的，也就仅仅是这一个参数 SGA\_TARGET。设置这个参数后，你就不需要为每个内存区来指定大小了。SGA\_TARGET 指定了 SGA 可以使用的最大内存大小，而 SGA 中各个内存的大小由 Oracle 自行控制，不需要人为指定。Oracle 可以随时调节各个区域的大小，使之达到系统性能最佳状态的个最合理大小，并且控制他们之和在 SGA\_TARGET 指定的值之内。一旦给 SGA\_TARGET 指定值后（默认为 0，即没有启动 ASMM），就自动启动了 ASMM 特性。

设置了 SGA\_TARGET 后，以下的 SGA 内存区就可以由 ASMM 来自动调整：

- 共享池 (Shared Pool)
- Java 池 (Java Pool)
- 大池 (Large Pool)
- 数据缓存区 (Buffer Cache)
- 流池 (Streams Pool)

对于 SGA\_TARGET 的限制，它的大小是不能超过 SGA\_MAX\_SIZE 的大小的。

```
SQL> show parameter sga
```

NAME	TYPE	VALUE
lock_sga	boolean	FALSE
pre_page_sga	boolean	FALSE
sga_max_size	big integer	276M
sga_target	big integer	276M

```
SQL>
SQL>
SQL> alter system set sga_target=280M;
alter system set sga_target=280M
*
ERROR at line 1:
ORA-02097: parameter cannot be modified because specified value is
invalid
ORA-00823: Specified value of sga_target greater than sga_max_size
```

另外，当指定 SGA\_TARGET 小于 SGA\_MAX\_SIZE，实例重启后，SGA\_MAX\_SIZE 就自动变为和 SGA\_TARGET 一样的值了。

```
SQL> show parameter sga

NAME                                TYPE                                VALUE
-----                                -
lock_sga                            boolean                             FALSE
pre_page_sga                         boolean                             FALSE
sga_max_size                         big integer                         276M
sga_target                           big integer                         276M

SQL> alter system set sga_target=252M;

System altered.

SQL> show parameter sga

NAME                                TYPE                                VALUE
-----                                -
lock_sga                            boolean                             FALSE
pre_page_sga                         boolean                             FALSE
sga_max_size                         big integer                         276M
sga_target                           big integer                         252M

SQL> startup force
ORACLE instance started.

Total System Global Area  264241152 bytes
Fixed Size                 1248428 bytes
Variable Size             117441364 bytes
Database Buffers         138412032 bytes
Redo Buffers              7139328 bytes
Database mounted.
Database opened.
SQL> show parameter sga

NAME                                TYPE                                VALUE
-----                                -
lock_sga                            boolean                             FALSE
pre_page_sga                         boolean                             FALSE
sga_max_size                         big integer                         252M
sga_target                           big integer                         252M
SQL>
```

对于 SGA\_TARGET，还有重要一点就是，它的值可以动态修改（在 SGA\_MAX\_SIZE 范围内）。在 10g 之前，如果需要修改 SGA 的大小（即修改 SGA\_MAX\_SIZE 的值）需要重启实例才能生效。当然，在 10g 中，修改 SGA\_MAX\_SIZE 的值还是需要重启的。但是有了 SGA\_TARGET 后，可以将 SGA\_MAX\_SIZE 设置偏大，再根据实际需要调整 SGA\_TARGET 的值（我个人不推荐频繁修改 SGA 的大小，SGA\_TARGET 在实例启动时设置好，以后不要再修改）。

SGA\_TARGET 带来一个重要的好处就是，能使 SGA 的利用率达到最佳，从而节省内存成本。因为 ASMM 启动后，Oracle 会自动根据需要调整各个区域的大小，大大减少了某些区域内内存紧张，而某些区域又有内存空闲的矛盾情况出现。这也同时大大降低了出现 4031 错误的几率。

- **use\_indirect\_data\_buffers**

这个参数使 32 位平台使用扩展缓冲缓存基址，以支持支持 4GB 多物理内存。设置此参数，可以使 SGA 突破在 32 位系统中的 2G 最大限制。64 位平台中，这个参数被忽略。

### 1.1.2. 关于 SGA 的重要视图

要了解和观察 SGA 的使用情况，并且根据统计数据来处理问题和调整性能，主要有以下的几个系统视图。

- **v\$sga**

这个视图包括了 SGA 的的总体情况，只包含两个字段：name（SGA 内存区名字）和 value（内存区的值，单位为字节）。它的结果和 show sga 的结果一致，显示了 SGA 各个区的大小：

```
SQL> select * from v$sga;

NAME                                VALUE
-----                                -
Fixed Size                          1248428
Variable Size                        117441364
Database Buffers                    138412032
Redo Buffers                         7139328

4 rows selected.

SQL> show sga

Total System Global Area  264241152 bytes
Fixed Size                 1248428 bytes
Variable Size             117441364 bytes
Database Buffers         138412032 bytes
Redo Buffers              7139328 bytes
SQL>
```

- **v\$sgastat**

这个视图比较重要。它记录了关于 sga 的统计信息。包含三个字段：Name（SGA 内存区的名字）；Bytes（内存区的大小，单位为字节）；Pool（这段内存所属的内存池）。

这个视图尤其重要的是，它详细记录了个各个池（Pool）内存分配情况，对于定位 4031 错误有重要参考价值。

以下语句可以查询 Shared Pool 空闲率：

```
SQL> select to_number(v$parameter.value) value, v$sgastat.BYTES,
2         (v$sgastat.bytes/v$parameter.value)*100 "percent free"
3         from v$sgastat, v$parameter
4         where v$sgastat.name= 'free memory'
5         and v$parameter.name = 'shared_pool_size'
6         and v$sgastat.pool='shared pool'
7         ;

        VALUE          BYTES percent free
-----
503316480  141096368  28.033329645

SQL>
```

## • v\$sga\_dynamic\_components

这个视图记录了 SGA 各个动态内存区的情况，它的统计信息是基于已经完成了的，针对 SGA 动态内存区大小调整的操作，字段组成如下：

字段	数据类型	描述
COMPONENT	VARCHAR2 (64)	内存区名称
CURRENT_SIZE	NUMBER	当前大小
MIN_SIZE	NUMBER	自从实例启动后的最小值
MAX_SIZE	NUMBER	自从实例启动后的最大值
OPER_COUNT	NUMBER	自从实例启动后的调整次数
LAST_OPER_TYPE	VARCHAR2 (6)	最后一次完成的调整动作，值包括： <ul style="list-style-type: none"><li>• GROW (增加)</li><li>• SHRINK (缩小)</li></ul>
LAST_OPER_MODE	VARCHAR2 (6)	最后一次完成的调整动作的模式，包括： <ul style="list-style-type: none"><li>• MANUAL (手动)</li><li>• AUTO (自动)</li></ul>
LAST_OPER_TIME	DATE	最后一次完成的调整动作的开始时间
GRANULE_SIZE	NUMBER	GRANULE 大小 (关于 granule 后面详细介绍)

## • V\$SGA\_DYNAMIC\_FREE\_MEMORY

这个视图只有一个字段，一条记录：当前 SGA 可用于动态调整 SGA 内存区的空闲区域大小。它的值相当于 (SGA\_MAX\_SIZE - SGA 各个区域设置大小的总和)。当设置了 SGA\_TARGET 后，它的值一定为 0 (为什么就不需要我再讲了吧^\_^)。

下面的例子可以很清楚的看到这个视图的作用：

```
SQL> select * from v$sga_dynamic_free_memory;

CURRENT_SIZE
-----
0

SQL> show parameter shared_pool

NAME                                TYPE                                VALUE
-----                                -                                -
shared_pool_size                    big integer                        50331648

SQL> alter system set shared_pool_size=38M;

system altered.

SQL> show parameter shared_pool
```

```

NAME                                     TYPE          VALUE
-----
shared_pool_size                         big integer 41943040

SQL> select * from v$sga_dynamic_free_memory;

CURRENT_SIZE
-----
8388608

```

### 1.1.3. 数据库缓冲区 (Database Buffers)

Buffer Cache 是 SGA 区中专门用于存放从数据文件中读取的数据块拷贝的区域。Oracle 进程如果发现需要访问的数据块已经在 buffer cache 中，就直接读写内存中的相应区域，而无需读取数据文件，从而大大提高性能（要知道，内存的读取效率是磁盘读取效率的 14000 倍）。Buffer cache 对于所有 oracle 进程都是共享的，即能被所有 oracle 进程访问。

和 Shared Pool 一样，buffer cache 被分为多个集合，这样能够大大降低多 CPU 系统中的争用问题。

#### 1.1.1.1. Buffer cache 的管理

Oracle 对于 buffer cache 的管理，是通过两个重要的链表实现的：写链表和最近最少使用链表 (the Least Recently Used LRU)。写链表所指向的是所有脏数据块缓存（即被进程修改过，但还没有被回写到数据文件中去的的数据块，此时缓冲中的数据和数据文件中的数据不一致）。而 LRU 链表指向的是所有空闲的缓存、pin 住的缓存以及还没有来的及移入写链表的脏缓存。空闲缓存中没有任何有用的数据，随时可以使用。而 pin 住的缓存是当前正在被访问的缓存。LRU 链表的两端就分别叫做最近使用端 (the Most Recently Used MRU) 和最近最少使用端 (LRU)。

#### • Buffer cache 的数据块访问

当一个 Oracle 进程访问一个缓存是，这个进程会将这块缓存移到 LRU 链表中的 MRU。而当越来越多的缓冲块被移到 MRU 端，那些已经过时的脏缓冲（即数据改动已经被写入数据文件中，此时缓冲中的数据和数据文件中的数据已经一致）则被移到 LRU 链表中 LRU 端。

当一个 Oracle 用户进程第一次访问一个数据块时，它会先查找 buffer cache 中是否存在这个数据块的拷贝。如果发现这个数据块已经存在于 buffer cache（即命中 cache hit），它就直接读从内存中取该数据块。如果在 buffer cache 中没有发现该数据块（即未命中 cache miss），它就需要先从数据文件中读取该数据块到 buffer cache 中，然后才访问该数据块。命中次数与进程读取次数之比就是我们一个衡量数据库性能的重要指标：buffer hit ratio (buffer 命中率)，可以通过以下语句获得自实例启动至今的 buffer 命中率：

```

SQL> select 1-(sum(decode(name, 'physical reads', value, 0))/
2           (sum(decode(name, 'db block gets', value, 0))+
3           (sum(decode(name, 'consistent gets', value, 0))))) "Buffer
Hit Ratio"
4   from v$sysstat;

Buffer Hit Ratio
-----
.926185625

```

```
1 row selected.
```

```
SQL>
```

根据经验，一个良好性能的系统，这一值一般保持在 95%左右。

上面提到，如果未命中（missed），则需要先将数据块读取到缓存中去。这时，oracle 进程需要从空闲列表中找到一个适合大小的空闲缓存。如果空闲列表中没有适合大小的空闲 buffer，它就会从 LRU 端开始查找 LRU 链表，直到找到一个可重用的缓存块或者达到最大查找块数限制。在查找过程中，如果进程找到一个脏缓存块，它将这个缓存块移到写链表中去，然后继续查找。当它找到一个空闲块后，就从磁盘中读取数据块到缓存块中，并将这个缓存块移到 LRU 链表的 MRU 端。

当有新的对象需要请求分配 buffer 时，会通过内存管理模块请求分配空闲的或者可重用的 buffer。“free buffer requested”就是产生这种请求的次数；

当请求分配 buffer 时，已经没有适合大小的空闲 buffer 时，需要从 LRU 链表上获取到可重用的 buffer。但是，LRU 链表上的 buffer 并非都是立即可重用的，还会存在一些块正在被读写或者已经被别的用户所等待。根据 LRU 算法，查找可重用的 buffer 是从链表的 LRU 端开始查找的，如果这一段的前面存在这种不能理解被重用的 buffer，则需要跳过去，查找链表中的下一个 buffer。“free buffer inspected”就是被跳过去的 buffer 的数目。

如果 Oracle 用户进程达到查找块数限制后还没有找到空闲缓存，它就停止查找 LRU 链表，并且通过信号同志 DBWO 进程将脏缓存写入磁盘去。

下面就是 oracle 用户进程访问一个数据块的伪代码：

```
user_process_access_block(block)
{
    if (search_lru(block))
    {
        g_cache_hit++;
        return read_block_from_buffer_cache(block);
    }
    else
    {
        g_cache_missed++;
        search_count = 1;
        searched = FALSE;
        set_lru_latch_context();
        buffer_block = get_lru_from_lru();

        do
        {
            if (block == buffer_block)
            {
                set_buffer_block(buffer_block,
read_block_from_datafile(block);
                move_buffer_block_to_mru(buffer_block);
                searched = TRUE;
            }

            search_count++;
            buffer_block = get_next_from_lru(buffer_block);
        }while(!searched && search_count < BUFFER_SEARCH_THRESHOLD)

        free_lru_latch_context();

        if (!searched)
        {
            buffer_block = signal_dbw0_write_dirty_buffer();
        }
    }
}
```

```

        set_buffer_block(buffer_block,
read_block_from_datafile(block);
        move_buffer_block_to_mru(buffer_block);
    }

    return buffer_block;
}
}

```

## • 全表扫描

当发生全表扫描 (Full Table Scan) 时, 用户进程读取表的数据块, 并将他们放在 LRU 链表的 LRU 端 (和上面不同, 不是放在 MRU 端)。这样做的目的是为了使全表扫描的数据尽快被移出。因为全表扫描一般发生的频率较低, 并且全表扫描的数据块大部分在以后都不会被经常使用到。

而如果你希望全表扫描的数据能被 cache 住, 使之在扫描时放在 MRU 端, 可以通过在创建或修改表 (或簇) 时, 指定 CACHE 参数。

## • Flush Buffer

回顾一下前面一个用户进程访问一个数据块的过程, 如果访问的数据块不在 buffer cache 中, 就需要扫描 LRU 链表, 当达到扫描块数限制后还没有找到空闲 buffer, 就需要通知 DBWO 将脏缓存回写到磁盘。分析一下伪代码, 在这种情况下, 用户进程访问一个数据块的过程是最长的, 也就是效率最低的。如果一个系统中存在大量的脏缓冲, 那么就可能导致用户进程访问数据性能下降。

我们可以通过人工干预将所有脏缓冲回写到磁盘去, 这就是 flush buffer。

在 9i, 可以用以下语句:

```
alter system set events = 'immediate trace name flush_cache'; --9i
```

在 10g, 可以用以下方式 (9i 的方式在 10g 仍然有效):

```
alter system flush buffer cache; -- 10g
```

另外, 9i 的设置事件的方式可以是针对系统全部的, 也可以是对会话的 (即将该会话造成的脏缓冲回写)。

### 1.1.3.1. Buffer Cache 的重要参数配置

Oracle 提供了一些参数用于控制 Buffer Cache 的大小等特性。下面介绍一下这些参数。

## • Buffer Cache 的大小配置

由于 Buffer Cache 中存放的是从数据文件中来的数据块的拷贝, 因此, 它的大小的计算也是以块的尺寸为基数的。而数据块的大小是由参数 db\_block\_size 指定的。9i 以后, 块的大小默认是 8K, 它的值一般设置为和操作系统的块尺寸相同或者它的倍数。

而参数 db\_block\_buffers 则指定了 Buffer Cache 中缓存块数。因此, buffer cache 的大小就等于 db\_block\_buffers \* db\_block\_size。

在 9i 以后, Oracle 引入了一个新参数: db\_cache\_size。这个参数可以直接指定 Buffer Cache 的大小, 而不需要通过上面的方式计算出。它的默认值 48M, 这个数对于一个系统来说一般是够用的。

注意: db\_cache\_size 和 db\_block\_buffers 是不能同时设置的, 否则实例启动时会报错。

```
SQL> alter system set db_block_buffers=16384 scope=spfile;
system altered.

SQL> alter system set db_cache_size=128M scope=spfile;
system altered.

SQL> startup force

ORA-00381: cannot use both new and old parameters for buffer cache size
specification
```

9i 以后，推荐使用 db\_cache\_size 来指定 buffer cache 的大小。

在 OLTP 系统中，对于 DB\_CACHE\_SIZE 的设置，我的推荐配置是：

$DB\_CACHE\_SIZE = SGA\_MAX\_SIZE/2 \sim SGA\_MAX\_SIZE*2/3$

最后，DB\_CACHE\_SIZE 是可以联机修改的，即实例无需重启，除非增大 Buffer Cache 导致 SGA 实际大小大于 SGA\_MAX\_SIZE。

### • 多种块尺寸系统中的 Buffer Cache 的配置

从 9i 开始，Oracle 支持创建不同块尺寸的表空间，并且可以为不同块尺寸的数据块指定不同大小的 buffer cache。

9i 以后，除了 SYSTEM 表空间和 TEMPORARY 表空间必须使用标准块尺寸外，所有其他表空间都可以最多指定四种不同的块尺寸。而标准块尺寸还是由上面的所说的参数 db\_block\_size 来指定。而 db\_cache\_size 则是标准块尺寸的 buffer cache 的大小。

非标准块尺寸的块大小可以在创建表空间（CREATE TABLESPACE）是通过 BLOCKSIZE 参数指定。而不同块尺寸的 buffer cache 的大小就由相应参数 DB\_nK\_CACHE\_SIZE 来指定，其中 n 可以是 2, 4, 8, 16 或者 32。例如，你创建了一个块大小为 16K 的非标准块尺寸的表空间，你就可以通过设置 DB\_16K\_CACHE\_SIZE 为来指定缓存这个表空间数据块的 buffer cache 的大小。

任何一个尺寸的 Buffer Cache 都是不可以缓存其他尺寸的数据块的。因此，如果你打算使用多种块尺寸用于你的数据库的存储，你必须最少设置 DB\_CACHE\_SIZE 和 DB\_nK\_CACHE\_SIZE 中的一个参数（10g 后，指定了 SGA\_TARGET 就可以不需要指定 Buffer Cache 的大小）。并且，你需要给你要用到的非标准块尺寸的数据块指定相应的 Buffer Cache 大小。这些参数使你可以为系统指定多达 4 种不同块尺寸的 Buffer Cache。

另外，请注意一点，DB\_nK\_CACHE\_SIZE 参数不能设定标准块尺寸的缓冲区大小。举例来说，如果 DB\_BLOCK\_SIZE 设定为 4K，就不能再设定 DB\_4K\_CACHE\_SIZE 参数。

### • 多缓冲池

你可以配置不同的 buffer cache，可以达到不同的 cache 数据的目的。比如，可以设置一部分 buffer cache 缓存过的数据在使用后马上释放，使后来的数据可以立即使用缓冲池；还可以设置数据进入缓冲池后就被 keep 住不再释放。部分数据库对象（表、簇、索引以及分区）可以控制他们的数据缓存的行为，而这些不同的缓存行为就使用不同缓冲池。

- 保持缓冲池（Keep Buffer Pool）用于缓存那些永久驻入内存的数据块。它的大小由参数 DB\_KEEP\_CACHE\_SIZE 控制；
- 回收缓冲池（Recycle Buffer Pool）会立即清除那些不在使用的数据缓存块。它的大小由参数 DB\_RECYCLE\_CACHE\_SIZE 指定；

- 默认的标准缓存池，也就是上面所说的 DB\_CACHE\_SIZE 指定。

这三个参数相互之间是独立的。并且他们都只适用于标准块尺寸的数据块。与 8i 兼容参数 DB\_BLOCK\_BUFFERS 相应的，DB\_KEEP\_CACHE\_SIZE 对应有 BUFFER\_POOL\_KEEP、DB\_RECYCLE\_CACHE\_SIZE 对应有 BUFFER\_POOL\_RECYCLE。同样，这些参数之间是互斥的，即 DB\_KEEP\_CACHE\_SIZE 和 BUFFER\_POOL\_KEEP 之间只能设置一个。

## • 缓冲池建议器

从 9i 开始，Oracle 提供了一些自动优化工具，用于调整系统配置，提高系统性能。建议器就是其中一种。建议器的作用就是在系统运行过程中，通过监视相关统计数据，给相关配置在不同情况下的性能效果，提供给 DBA 做决策，以选取最佳的配置。

9i 中，Buffer Cache 就有了相应的建议器。参数 db\_cache\_advice 用于该建议器的开关，默认值为 FALSE（即关）。当设置它为 TRUE 后，在系统运行一段时间后，就可以查询视图 v\$db\_cache\_advice 来决定如何使之 DB\_CACHE\_SIZE 了。关于这个建议器和视图，我们会在下面的内容中介绍。

## • 其他相关参数

### DB\_BLOCK\_LRU\_LATCHES

LRU 链表作为一个内存对象，对它的访问是需要进行锁 (latch) 控制的，以防止多个用户进程同时使用一个空闲缓存块。DB\_BLOCK\_LRU\_LATCHES 设置了 LUR latch 的数量范围。Oracle 通过一系列的内部检测来决定是否使用这个参数值。如果这个参数没有设置，Oracle 会自动为它计算出一个值。一般来说，oracle 计算出来的值是比较合理，无需再去修改。

9i 以后这个参数是隐含参数。对于隐含参数，我建议在没有得到 Oracle 支持的情况下不要做修改，否则，如果修改了，Oracle 是可以拒绝为你做支持的。

### DB\_WRITER\_PROCESSES

在前面分析 Oracle 读取 Buffer Cache 时，提到一个 Oracle 重要的后台进程 DBW0，这个（或这些）进程负责将脏缓存块写回到数据文件中去，称为数据库书写器进程（Database Writer Process）。DB\_WRITER\_PROCESSES 参数配置写进程的个数，各个进程以 DBWn 区分，其中 n>=0，是进程序号。一般情况下，DB\_WRITER\_PROCESSES = MAX(1, TRUNC(CPU 数/8))。也就是说，CPU 数小于 8 时，DB\_WRITER\_PROCESSES 为 1，即只有一个写进程 DBW0。这对于一般的系统来说也是足够用。当你的系统的修改数据的任务很重，并且已经影响到性能时，可以调整这个参数。这个参数不要超过 CPU 数，否则多出的进程也不会起作用，另外，它的最大值不能超过 20。

DBWn 进程除了上面提到的在用户进程读取 buffer cache 时会被触发，还能被 Checkpoint 触发（Checkpoint 是实例从 redo log 中做恢复的起始点）。

## 1.1.3.2. Buffer Cache 的重要视图

关于 Buffer Cache，oracle 提供一些重要视图，用于查询关于 Buffer Cache 的重要信息，为调整 Buffer Cache、提高性能提供参考。下面一一介绍它们

## • v\$db\_cache\_advice

上面我们提到了 Oracle 的建议器，其中有一个针对 Buffer Cache 的建议器。在我们设置了参数 db\_cache\_advice 为 TRUE 后，经过一段时间的系统运行，Oracle 收集到相关

统计数据，并根据一定的数学模型，预测出 DB\_CACHE\_SIZE 在不同大小情况下的性能数据。我们就可以由视图 V\$DB\_CACHE\_ADVICE 查出这些数据，并根据这些数据调整 DB\_CACHE\_SIZE，使系统性能最优。

下面是关于这个视图的结构描述：

字段	数据类型	描述
ID	NUMBER	缓冲池标识号（从 1 到 8，1-6 对应于 DB_nK_CACHE_SIZE，DB_CACHE_SIZE 与系统标准块尺寸的序号相关，如 DB_BLOCK_SIZE 为 8K，则 DB_CACHE_SIZE 的标识号为 3（2, 4, 8...）。7 是 DB_KEEP_CACHE_SIZE，8 是 DB_RECYCLE_CACHE_SIZE）
NAME	VARCHAR2 (20)	缓冲池名称
BLOCK_SIZE	NUMBER	缓冲池块尺寸（字节为单位）
ADVICE_STATUS	VARCHAR2 (3)	建议器状态：ON 表示建议器在运行；OFF 表示建议器已经关闭。当建议器关闭了，视图中的数据是上一次打开所统计得出的。
SIZE_FOR_ESTIMATE	NUMBER	预测性能数据的 Cache 大小（M 为单位）
SIZE_FACTOR	NUMBER	预测的 Cache 大小因子（即与当前大小的比例）
BUFFERS_FOR_ESTIMATE	NUMBER	预测性能数据的 Cache 大小（缓冲块数）
ESTD_PHYSICAL_READ_FACTOR	NUMBER	这一缓冲大小时，物理读因子，它是如果缓冲大小为 SIZE_FOR_ESTIMATE 时，建议器预测物理读数与当前实际物理读数的比率值。如果当前物理读数为 0，这个值为空。
ESTD_PHYSICAL_READS	NUMBER	如果缓冲大小为 SIZE_FOR_ESTIMATE 时，建议器预测物理读数。

下面是从这个视图中查询出来的数据：

```
SQL> select size_for_estimate, estd_physical_read_factor,
estd_physical_reads
  2   from v$db_cache_advice
  3   where name = 'DEFAULT';
```

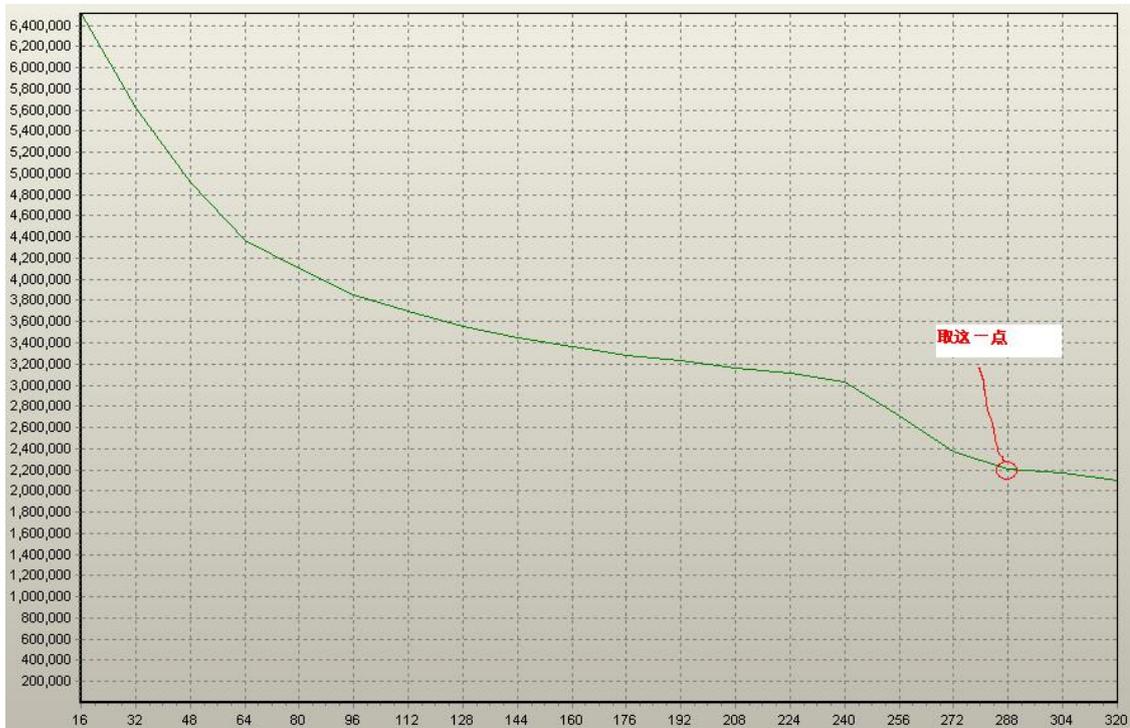
SIZE_FOR_ESTIMATE	ESTD_PHYSICAL_READ_FACTOR	ESTD_PHYSICAL_READS
16	2.0176	6514226
32	1.7403	5619048
48	1.5232	4917909
64	1.3528	4367839
80	1.2698	4099816
96	1.1933	3852847
112	1.1443	3694709
128	1.1007	3553685
144	1.0694	3452805
160	1.0416	3362964
176	1.0175	3285085
192	1	3228693
208	0.9802	3164754
224	0.9632	3109920
240	0.9395	3033427
256	0.8383	2706631

272	0.7363	2377209
288	0.682	2202116
304	0.6714	2167888
320	0.6516	2103876

20 rows selected

当前我们的 DB\_CACHE\_SIZE 为 192M，可以看到，它的物理读因子为 1，物理读数为 3228693。那么如何根据这些数据调整 DB\_CACHE\_SIZE 呢？给出一个方法，找到变化率较平缓的点作为采用值。因为建议器做预测是，DB\_CACHE\_SIZE 的预测值的的增长步长是相同的，是 16M。我们按照这一步长增加 DB\_CACHE\_SIZE，如果每次增加物理读降低都很明显，就可以继续增加，直到物理读降低不明显，说明继续增加 DB\_CACHE\_SIZE 没有太大作用。当然，性能和可用资源是天平的两端，你需要根据自己系统的实际情况调整。

上面的例子中，我们可以考虑将 DB\_CACHE\_SIZE 调整到 288M。因为在 288M 之前，物理读因子变化都比较大，而从 288M 到 304M 以后，这个因子变化趋缓。用一个二维图可以更容易看出这个变化来：



这一视图作为调整 DB\_CACHE\_SIZE 以提高性能有很大参考价值。但衡量 Buffer Cache 是否合适的重要指标还是我们前面提到的缓存命中率 (Buffer Hit)，而影响缓存命中率往往还有其他因素，如性能极差的 SQL 语句。

### • V\$BUFFER\_POOL

这一视图显示了当前实例中所有缓冲池的信息。它的结构如下：

字段	数据类型	描述
ID	NUMBER	缓冲池 ID，和上面视图描述相同。
NAME	VARCHAR2 (20)	缓冲池名称

字段	数据类型	描述
BLOCK_SIZE	NUMBER	缓冲池块尺寸（字节为单位）
RESIZE_STATE	VARCHAR2(10)	缓冲池当前状态。 STATIC：没有被正在调整大小 ALLOCATING：正在分配内存给缓冲池（不能被用户取消） ACTIVATING：正在创建新的缓存块（不能被用户取消） SHRINKING：正在删除缓存块（能被用户取消）
CURRENT_SIZE	NUMBER	缓冲池大小（M 为单位）
BUFFERS	NUMBER	当前缓存块数
TARGET_SIZE	NUMBER	如果正在调整缓冲池大小（即状态不为 STATIC），这记录了调整后的大小（M 为单位）。如果状态为 STATIC，这个值和当前大小值相同。
TARGET_BUFFERS	NUMBER	如果正在调整缓冲池大小（即状态不为 STATIC），这记录了调整后的缓存块数。否则，这个值和当前缓存块数相同。
PREV_SIZE	NUMBER	前一次调整的缓冲池大小。如果从来没有调整过，则为 0。
PREV_BUFFERS	NUMBER	前一次调整的缓存块数。如果从来没有调整过，则为 0。
LO_BNUM	NUMBER	9i 后已经废弃字段
HI_BNUM	NUMBER	9i 后已经废弃字段
LO_SETID	NUMBER	9i 后已经废弃字段
HI_SETID	NUMBER	9i 后已经废弃字段
SET_COUNT	NUMBER	9i 后已经废弃字段

- ***v\$buffer\_pool\_statistics***

V\$BUFFER\_POOL\_STATISTICS 视图记录了所有缓冲池的统计数据。它的结构如下：

字段	数据类型	描述
ID	NUMBER	缓冲池 ID，和上面视图描述相同。
NAME	VARCHAR2(20)	缓冲池名称
SET_MSIZE	NUMBER	缓冲池中缓存块的最大数
CNUM_REPL	NUMBER	在置换列表中的缓存块数
CNUM_WRITE	NUMBER	在写列表中的缓存块数
CNUM_SET	NUMBER	当前的缓存块数
BUF_GOT	NUMBER	读取过的缓存块数
SUM_WRITE	NUMBER	被写过的缓存块数
SUM_SCAN	NUMBER	被扫描过的缓存块数
FREE_BUFFER_WAIT	NUMBER	等待空闲块统计数

字段	数据类型	描述
WRITE_COMPLETE_WAIT	NUMBER	等待完成写统计数
BUFFER_BUSY_WAIT	NUMBER	忙（正在被使用）等待统计数
FREE_BUFFER_INSPECTED	NUMBER	确认了的空闲缓存块数（即可用的）
DIRTY_BUFFERS_INSPECTED	NUMBER	确认了的脏缓存块数
DB_BLOCK_CHANGE	NUMBER	被修改过的数据块数
DB_BLOCK_GETS	NUMBER	读取过的数据块数
CONSISTENT_GETS	NUMBER	一致性读统计数
PHYSICAL_READS	NUMBER	物理读统计数
PHYSICAL_WRITES	NUMBER	物理写统计数

查看当前的 Buffer Cache 命中率:

```
SQL> select 1-(physical_reads)/(consistent_gets+db_block_gets)
2 from v$buffer_pool_statistics;

1-(PHYSICAL_READS)/(CONSISTENT
-----
0.967658520581074

SQL>
```

### • v\$dbh

这一视图在深入定位缓冲区问题时很有用。它记录了缓冲区中所有数据块对象。粒度非常细。这个视图最初的目的是用于 OPS (Oracle Parallel Server Oracle 平行服务器, 9i 后称为 RAC) 的, 是用于保证 RAC 中各个节点的数据一致性的。但是, 我们可以通过它来查询 Buffer Cache 的使用情况, 找出大量消耗 Buffer Cache 的对象。下面的语句就可以完成这一工作:

```
SQL> column c0 heading 'Owner' format a15
SQL> column c1 heading 'Object|Name' format a30
SQL> column c2 heading 'Number|of|Buffers' format 999,999
SQL> column c3 heading 'Percentage|ofData|Buffer' format 999,999,999
SQL> select
2 owner c0,
3 object_name c1,
4 count(1) c2,
5 (count(1)/(select count(*) from v$dbh)) *100 c3
6 from
7 dba_objects o,
8 v$dbh bh
9 where
10 o.object_id = bh.objd
11 and
12 o.owner not in ('SYS','SYSTEM')
13 group by
14 owner,
15 object_name
16 order by
17 count(1) desc
18 ;
```

```
C0 C1 C2 C3
-----
```

```

PLSQLDEV          STANDARD_CITY          17290          72.5860621
DBOWNER           MSG_LOG              2              0.00839630
DBOWNER           COUNTRY_PK          1              0.00419815
DBOWNER           PARAMETER           1              0.00419815
DBOWNER           PARAMETER_PK        1              0.00419815
DBOWNER           MSG_LOG_IDX1        1              0.00419815

6 rows selected

SQL>

```

更重要的是，这个视图记录的粒度非常细，一条记录对应了一个数据块。这对于我们做内存问题分析或分析 Oracle 行为时很有帮助。

下面是这个视图的结构：

字段	数据类型	说明
FILE#	NUMBER	缓存块对应的数据块所在的数据文件号。可以通过视图 DBA_DATA_FILES 或 V\$DBFILES 查询
BLOCK#	NUMBER	缓存块对应的数据块编号
CLASS#	NUMBER	分类编号
STATUS	VARCHAR2(1)	缓存块的状态 FREE: 空闲，没有被使用 XCUR: 排斥（正在被使用） SCUR: 可被共享 CR: 一致性读 READ: 正在从磁盘读入 MREC: 处于从存储介质恢复状态 IREC: 处于实例恢复状态
XNC	NUMBER	缓存块上由于和其他实例争用导致的 PCM (Parallel Cache Management 并行缓存管理) x to null 锁的数量。这一字段已经被废弃。
LOCK_ELEMENT_ADDR	RAW(4   8)	缓存块上 PCM 锁的地址。如果多个缓存块的 PCM 锁地址相同，说明他们被同一锁锁住。
LOCK_ELEMENT_NAME	NUMBER	缓存块上 PCM 锁的地址。如果多个缓存块的 PCM 锁地址相同，说明他们被同一锁锁住。
LOCK_ELEMENT_CLASS	NUMBER	缓存块上 PCM 锁的地址。如果多个缓存块的 PCM 锁地址相同，说明他们被同一锁锁住。
FORCED_READS	NUMBER	由于其他实例的 PCM 锁锁住了该缓存块，导致当前实例尝试重新请求读该缓冲块的次数。
FORCED_WRITES	NUMBER	由于其他实例的 PCM 锁锁住了该缓存块，导致当前实例尝试重新请求写该缓冲块的次数。
DIRTY	VARCHAR2(1)	脏标志: Y - 块被修改过，是脏块; N - 不是脏块
TEMP	VARCHAR2(1)	是否为临时块: Y - 是; N - 否。
PING	VARCHAR2(1)	是否被 ping 住: Y - 是; N - 否。
STALE	VARCHAR2(1)	是否是陈旧块: Y - 是; N - 否。

字段	数据类型	说明
DIRECT	VARCHAR2(1)	是否为直接读写块: Y - 是; N - 否。
NEW	VARCHAR2(1)	字段被废弃, 始终为 N
OBJD	NUMBER	数据块所属对象的对象标号, 可以查询 dba_objects
TS#	NUMBER	数据块所在的表空间号, 可以查询 v\$tablespaces

#### 1.1.4. 共享池 (Shared pool)

SGA 中的共享池由库缓存 (Library Cache)、字典缓存 (Dictionary Cache)、用于并行执行消息的缓冲以及控制结构组成。

Shared Pool 的大小由参数 SHARED\_POOL\_SIZE 决定。在 32 位系统中, 这个参数的默认值是 8M, 而 64 位系统中的默认值位 64M。最大为 4G。

对于 Shared Pool 的内存管理, 是通过修正过的 LRU 算法表来实现的。

下面分别介绍 Shared Pool 的几个组成部分。

##### 1.1.4.1. 库缓存 (Library Cache)

Library Cache 中包括共享 SQL 区 (Shared SQL Areas)、PL/SQL 存储过程和包以及控制结构 (如锁、库缓存句柄)。

任何用户都可以访问共享 SQL 区 (可以通过 v\$sqlarea 访问, 随后会介绍这个重要视图)。因此库缓存存在于 SGA 的共享池中。

##### • 共享 SQL 区和私有 SQL 区

Oracle 会为每一条 SQL 语句运行 (每运行一条语句 Oracle 都会打开一个游标) 提供一个共享 SQL 区 (Shared SQL Areas) 和私有 SQL 区 (Private SQL Areas 属于 PGA)。当发现两个 (或多个) 用户都在运行同一 SQL 语句时, Oracle 会重新组织 SQL 区, 使这些用户能重用共享 SQL 区。但他们还会在私有 SQL 区中保存一份这条 SQL 语句的拷贝。

一个共享 SQL 区中保存了一条语句的解析树和查询计划。在多用户系统中, Oracle 通过为 SQL 语句使用同一共享 SQL 区多次运行来节省内存。

当一条新的 SQL 语句被解析时, Oracle 从共享池中分配一块内存来存储共享 SQL 区。这块内存的大小与这条语句的复杂性相关。如果 Shared Pool 不够空间分配给共享 SQL 区, Oracle 将释放从 LRU 链表中查找到最近最少使用的内存块, 直到有足够空间给新的语句的共享 SQL 区。如果 Oracle 释放的是一个共享 SQL 区的内存, 那么相应的语句在下次执行时需要再次解析并重新分配共享 SQL 区。而从解析语句到分配共享 SQL 区是一个比较消耗 CPU 的工程。这就是为什么我们提倡使用绑定变量的原因了。在没有使用绑定变量时, 语句中的变量的数值不同, oracle 就视为一条新的语句 (9i 后可以通过 cursor\_sharing 来控制), 重复上面的解析、内存分配的动作, 将大大消耗系统资源, 降低系统性能。

##### • PL/SQL 程序单元

Oracle 对于 PL/SQL 程序单元 (存储过程、函数、包、匿名 PL/SQL 块和触发器) 的处理过程和对单个的 SQL 语句的处理过程相似。它会分配一个共享区来存储被解析、编译过的程序单元。同时分配一个私有区域来存放运行程序单元的会话所指定的程序单元的参数值 (包括本地变量、全局变量和包变量——这也叫做包的实例化) 和用于执行程序所需的

内存。如果多个用户运行同一个程序单元，则他们共享同一个共享区域，并且各自保持一份私有区域，用于用户会话中指定的变量值。

而一个 PL/SQL 程序单元中的每条单个 SQL 语句的处理过程则和上面描述的 SQL 语句的处理过程相同。要注意一点，尽管这些语句是从 PL/SQL 程序单元中来的，但是 Oracle 还是会为这些语句分配一块共享 SQL 区，同时为每个用户分配一个相应的私有 SQL 区。

### 1.1.4.2. 字典缓存 (Dictionary Cache)

数据字典是有关于数据库的参考信息、数据库的结构信息和数据库中的用户信息的一组表和视图的集合，如我们常用到的 V\$视图、DBA\_视图都属于数据字典。在 SQL 语句解析的过程中，Oracle 可以非常迅速的访问（如果需要的话）这些数据字典，在 SQL Trace 中，这种对数据字典的访问就被统计为回调 (recursive calls)。看下面例子：

第一调用语句，需要做硬解析：

```
SQL> select * from T_COMPANY;

9999 rows selected.

Execution Plan
-----
Plan hash value: 3356521258

-----
| Id | Operation          | Name          | Rows  | Bytes | Cost (%CPU)| Time |
|----|-----|-----|-----|-----|-----|-----|-----|
| 0  | SELECT STATEMENT  |               | 10000 | 156K  | 9   (0)| 00:00:01 |
| 1  | TABLE ACCESS FULL| T_COMPANY    | 10000 | 156K  | 9   (0)| 00:00:01 |
-----

Statistics
-----
355 recursive calls
  0 db block gets
 764 consistent gets
  39 physical reads
 116 redo size
305479 bytes sent via SQL*Net to client
 7711 bytes received via SQL*Net from client
  668 SQL*Net roundtrips to/from client
   5 sorts (memory)
   0 sorts (disk)
 9999 rows processed
```

可以看到，Recursive Calls 高达 355。第二次调用，无需解析，直接使用共享 SQL 区中缓存：

```
SQL> /

9999 rows selected.
```

```

Execution Plan
-----
Plan hash value: 3356521258

-----
| Id | Operation          | Name          | Rows  | Bytes | Cost (%CPU)| Time
|-----|-----|-----|-----|-----|-----|-----|
| 0  | SELECT STATEMENT   |               | 10000 | 156K | 9 (0) |
00:00:01 |
| 1  | TABLE ACCESS FULL| T_COMPANY    | 10000 | 156K | 9 (0) |
00:00:01 |
-----

Statistics
-----
          0 recursive calls
          0 db block gets
         705 consistent gets
          0 physical reads
          0 redo size
    305479 bytes sent via SQL*Net to client
         7711 bytes received via SQL*Net from client
          668 SQL*Net roundtrips to/from client
           0 sorts (memory)
           0 sorts (disk)
         9999 rows processed

```

由于没做解析，这时 recursive calls 为 0。

当然，recursive calls 并不仅仅发生在解析的时候。由于数据字典记录了所有对象的结构、数据信息，因此在对象结构、数据发生变化时都会访问数据字典：

```

SQL> delete from t_company where rownum=1;

1 row deleted.

...

Statistics
-----
          360 recursive calls
...

SQL> /

1 row deleted.

...

Statistics
-----
           4 recursive calls
...

SQL> /

```

```

...
Statistics
-----
          4 recursive calls
...

```

可以看到，上面的 delete 语句在第一次执行时，包括因解析和数据改动导致对数据字典的访问，因此 recursive calls 较高，为 360。在随后的执行中，因为没有做解析，所以 recursive calls 大大减少，只有 4，而这 4 个 recursive calls 是因为数据改变而需要对数据字典的访问。

因为 Oracle 对数据字典访问如此频繁，因此内存中有两处地方被专门用于存放数据字典。一个地方就是数据字典缓存 (Data Dictionary Cache)。数据字典缓存也被称为行缓存 (Row Cache)，因为它是以记录行为单元存储数据的，而不像 Buffer Cache 是以数据块为单元存储数据。内存中另外一个存储数据字典的地方是库缓存。所有 Oracle 的用户都可以访问这两个地方以获取数据字典信息。

### 1.1.4.3. 共享池的内存管理

通常来说，共享池是根据修正过的 LRU 算法来是否其中的对象（共享 SQL 区和数据自动记录行）的，否则这些对象就一直保持在共享池中。如果共享池需要为一个新对象分配内存，并且共享池中没有足够内存时，内存中那些不经常使用的对象就被释放掉。一个被许多会话使用过的共享池对象，即使最初创建它的进程已经结束，只要它是有用的，都会被修正过的 LRU 算法一直保持在共享池中。这样就使一个多用户的 Oracle 系统对 SQL 语句的处理和内存消耗最小。

注意，即使一个共享 SQL 区与一个打开的游标相关，但如果它长时间没有被使用，它还是可能会被从共享池中释放出来。而此时如果打开的游标还需要运行它的相关语句，Oracle 就会重新解析语句，并分配新的共享 SQL 区。

当一条 SQL 语句被提交给 Oracle 执行，Oracle 会自动执行以下的内存分配步骤：

1. Oracle 检查共享池，看是否已经存在关于这条语句的共享 SQL 区。如果存在，这个共享 SQL 区就被用于执行这条语句。而如果不存在，Oracle 就从共享池中分配一块新的共享 SQL 区给这条语句。同时，无论共享 SQL 区存在与否，Oracle 都会为用户分配一块私有 SQL 区以保存这条语句相关信息（如变量值）。
2. Oracle 为会话分配一个私有 SQL 区。私有 SQL 区的所在与会话的连接方式相关。

下面是 Oracle 执行一条语句时共享池内存分配过程的伪代码：

```

execute_sql(statement)
{
    if ((shared_sql_area = find_shared_sql_area(statement)) == NULL)
    {
        if (!allocate_from_shared_pool(&new_buffer))
        {
            if (new_buffer = find_age_area_by_lru(size_of(statement)) == NULL)
            {
                raise(4031);
                return 0;
            }
        }
    }

    shared_sql_area = set_shared_sql_area(new_buffer);
    parse_sql(statement, shared_sql_area);
}

```

```
private_sql_area = allocate_private_sql_area(statement);
run_sql(statement, shared_sql_area, private_sql_area);
return 1;
}
```

在以下情况下，Oracle 也会将共享 SQL 区从共享池中释放出来：

- 当使用 ANALYZE 语句更新或删除表、簇或索引的统计信息时，所有与被分析对象相关的共享 SQL 区都被从共享池中释放掉。当下一次被释放掉的语句被执行时，又重新在一个新的共享 SQL 区中根据被更新过的统计信息重新解析。
- 当对象结构被修改过后，与该对象相关的所有共享 SQL 区都被标识为无效（invalid）。在下次运行语句时再重新解析语句。
- 如果数据库的全局数据库名（Global Database Name）被修改了，共享池中的所有信息都会被清空掉。
- DBA 通过手工方式清空共享池：

```
ALTER SYSTEM FLUSH SHARED_POOL;
```

Shared Pool 能被分成几个区域，分别被不同的 latch（latch 数最大为 7，可以通过隐含参数 `_kgghdsidx_count` 设置）保护。

表 `x$kgghlu` 可以查看 shared pool 中的 LRU 列表。当满足以下条件之一时，shared pool 会分为多个区，分别有不同的 LRU 链表管理：

- 在 10g 之前版本，如果 shared pool 大于 128M、CPU 数量大于 4；
- Oracle 数据库版本为 10g

这时，在 `x$kgghlu` 中就会对应不同记录。

#### 1.1.4.4. 保留共享池

前面提到，如果 Oracle 解析一个 PL/SQL 程序单元，也需要从共享池中分配内存给这些程序单元对象。由于这些对象本一般比较大（如包），所以分配的内存空间也相对较大。系统经过长时间运行后，共享池可能存在大量内存碎片，导致无法满足对于大块内存段的分配。

为了使有足够空间缓存大程序块，Oracle 专门从共享池内置出一块区域来分配内存保持这些大块。这个保留共享池的默认大小是共享池的 5%。它的大小也可以通过参数 `SHARED_POOL_RESERVED_SIZE` 来调整。保留区是从共享池中分配，不是直接从 SGA 中分配的，它是共享池的保留部分，用于存储大块段。

Shared Pool 中内存大于 5000 字节的大段就会被存放在共享池的保留部分。而这个大小限制是通过隐含参数 `_SHARED_POOL_RESERVED_MIN_ALLOC` 来设定的（如前面所说，隐含参数不要去修改它）。除了在实例启动过程中，所有小于这个数的内存段永远都不会放到保留部分中，而大于这个值的大内存段也永远不会存放在非保留区中，即使共享池的空间不够用的情况下也是如此。

保留区的空闲内存也不会被包含在普通共享池的空闲列表中。它会维护一个单独的空闲列表。保留池也不会它的 LRU 列表中存放可重建（Recreatable 关于内存段的各种状态我们在后面的内容中再介绍）段。当释放普通共享池空闲列表上的内存时是不会清除这些大段的，同样，在释放保留池的空闲列表上的大内存段时也不会清除普通共享池中内存。

通过视图 `V$SHARED_POOL_RESERVED` 可以查到保留池的统计信息。其中字段 `REQUEST_MISSES` 记录了没有立即从空闲列表中得到可用的大内存段请求次数。这个值要为

0。因为保留区必须要有足够个空闲内存来适应那些短期的内存请求，而无需将那些需要长期 cache 住的没被 pin 住的可重建的段清除。否则就需要考虑增大 SHARED\_POOL\_RESERVED\_SIZE 了。

你可以通过观察视图 V\$SHARED\_POOL\_RESERVED 的 MAX\_USED\_SPACE 字段来判断保留池的大小是否合适。大多数情况下，你会观察到保留池是很少被使用的，也就是说 5% 的保留池空间可能有些浪费。但这需要经过长期观察来决定是否需要调整保留池大小。

保留区使用 shared pool 的 LRU 链表来管理内存块，但是在做扫描时，相互是不受影响的。例如，内存管理器扫描 shared pool 的 LRU 链表，清出空间以分配给一个小于 5000 字节的内存请求，是不会清出保留区的内存块的，相反亦然。

#### 1.1.4.5. 将重要、常用对象保持 (Keep) 在共享池中

前面提到，根据 LRU 算法，一些一段时间没有使用到的内存块会被情况释放。这就可能导致一些重要的对象（如一个含有大量通用算法函数的包、被 cache 的序列）被从内存中清除掉。这些对象可能只是间歇被使用，但是因为他们的处理过程复杂（不仅包本身重新分配内存、解析，还要检查里面的所有语句），要在内存中重建他们的代价非常大。

我们可以通过调用存储过程 DBMS\_SHARED\_POOL.KEEP 将这些对象保持在共享池中以降低这种风险。这个存储过程立即将对象及其从事对象载入 library cache 中，并将他们都标记为保持 (Keeping) 状态。对于这种对象，我们建议在实例启动时就 Keep 住，以减少内存碎片的几率。

有一种观点认为那些大对象（如包）是没有必要被 Keep 住的，因为他们会被保持在共享池的保留区（如前所述，这个区通常使用率很低），所以一般不可能被清出。这个观点是错误的！因为大多数大对象实际上是被分为多个小的内存段被载入共享池的，因此根本不会因为对象的大小而受到特别的保护。

另外，也不要通过频繁调用某些对象以防止他们被从共享池中清出。如果共享池大小设置合理，在系统运行的高峰时期，LRU 链表会相对较短，那些没有被 pin 住的对象会很快被清出，除非他们被 keep 住了。

#### 1.1.4.6. 关于 Shared Pool 的重要参数

这里再介绍与共享池相关的一些重要参数。

##### • SHARED\_POOL\_SIZE

这个参数我们前面已经提到，它指定了 Shared Pool 的大小。在 32 位系统中，这个参数的默认值是 8M，而 64 位系统中的默认值位 64M。

但是，在 SGA 中还存在一块叫内部 SGA 消耗 (Internal SGA Overhead) 的内存被放置在共享池中。在 9i 及之前版本，共享池的统计大小（通过 v\$sgastat 视图统计）为 SHARED\_POOL\_SIZE + 内部 SGA 消耗大小。而 10g 以后，SHARED\_POOL\_SIZE 就已经包含了这部分内存大小。因此在 10g 中，共享池的实际使用大小就是 SHARED\_POOL\_SIZE - 内部 SGA 消耗大小，这在配置共享池大小时需要考虑进去，否则，扶过 SHARED\_POOL\_SIZE 设置过小，在实例启动时就会报 ORA-00371 错误。

看 9i 中的结果：

```
SQL> show parameter shared_pool_size
```

NAME	TYPE	VALUE
shared_pool_size	big integer	41943040

```
SQL> select sum(bytes) from v$$sgastat where pool = 'shared pool';

SUM(BYTES)
-----
58720256

SQL>
```

- ***SHARED\_POOL\_RESERVED\_SIZE***

这个参数前面已经提到，指定了共享池中缓存大内存对象的保留区的大小。这里不再赘述。

- ***\_SHARED\_POOL\_RESERVED\_MIN\_ALLOC***

这个参数前面也已经介绍，设置了进入保留区的对象大小的阈值。

### 1.1.4.7. 共享池的重要视图

最后，我们再介绍关于共享池的一些重要视图

- ***v\$\$shared\_pool\_advice***

这个视图与 Oracle 的另外一个优化建议器——共享池建议器——相关。我们可以根据这个视图里面 oracle 所做的预测数据来调整共享池大小。它的预测范围是从当前值的 10% 到 200%之间。视图的结构如下

字段	数据类型	描述
SHARED_POOL_SIZE_FOR_ESTIMATE	NUMBER	估算的共享池大小（M 为单位）
SHARED_POOL_SIZE_FACTOR	NUMBER	估算的共享池大小与当前大小比
ESTD_LC_SIZE	NUMBER	估算共享池中用于库缓存的大小（M 为单位）
ESTD_LC_MEMORY_OBJECTS	NUMBER	估算共享池中库缓存的内存对象数
ESTD_LC_TIME_SAVED	NUMBER	估算将可以节省的解析时间。这些节省的时间来自于请求处理一个对象时，重新将它载入共享池的时间消耗和直接从库缓存中读取的时间消耗的差值。
ESTD_LC_TIME_SAVED_FACTOR	NUMBER	估算的节省的解析时间与当前节省解析时间的比。
ESTD_LC_MEMORY_OBJECT_HITS	NUMBER	估算的可以直接从共享池中命中库缓存的内存对象的命中次数。

关于如何根据建议器采用合理的共享池大小的方法，和前面提到的缓冲区建议器的使用方法类似，不再赘述。

- ***V\$\$SHARED\_POOL\_RESERVED***

前面提到了这个视图。这个视图存放了共享池保留区的统计信息。可以根据这些信息来调整保留区。视图结构如下：

Column	Datatype	Description
以下字段只有当参数 SHARED_POOL_RESERVED_SIZE 设置了才有效。		

Column	Datatype	Description
FREE_SPACE	NUMBER	保留区的空闲空间数。
AVG_FREE_SIZE	NUMBER	保留区的空闲空间平均数。
FREE_COUNT	NUMBER	保留区的空闲内存块数
MAX_FREE_SIZE	NUMBER	最大的保留区空闲空间数。
USED_SPACE	NUMBER	保留区使用空间数。
AVG_USED_SIZE	NUMBER	保留区使用空间平均数。
USED_COUNT	NUMBER	保留区使用内存块数。
MAX_USED_SIZE	NUMBER	最大保留区使用空间数
REQUESTS	NUMBER	请求再保留区查找空闲内存块的次数。
REQUEST_MISSES	NUMBER	无法满足查找保留区空闲内存块请求，需要从 LRU 列表中清出对象的次数。
LAST_MISS_SIZE	NUMBER	请求的内存大小，这次请求是最后一次需要从 LRU 列表清出对象来满足的请求。
MAX_MISS_SIZE	NUMBER	所有需要从 LRU 列表清出对象来满足的请求中的内存最大大小
以下字段无论参数 SHARED_POOL_RESERVED_SIZE 是否设置了都有效。		
REQUEST_FAILURES	NUMBER	没有内存能满足的请求次数（导致 4031 错误的请求）
LAST_FAILURE_SIZE	NUMBER	没有内存能满足的请求所需的内存大小（导致 4031 错误的请求）
ABORTED_REQUEST_THRESHOLD	NUMBER	不清出对象的情况下，导致 4031 错误的最小请求大小。
ABORTED_REQUESTS	NUMBER	不清出对象的情况下，导致 4031 错误的请求次数。。
LAST_ABORTED_SIZE	NUMBER	不清出对象的情况下，最后一次导致 4031 错误的请求大小。

我们可以根据后面 4 个字段值来决定如何设置保留区的大小以避免 4031 错误的发生。

- ***v\$db\_object\_cache***

这一视图显示了所有被缓存在 library cache 中的对象，包括表、索引、簇、同义词、PL/SQL 存储过程和包以及触发器。

字段	数据类型	说明
OWNER	VARCHAR2(64)	对象所有者
NAME	VARCHAR2(1000)	对象名称
DB_LINK	VARCHAR2(64)	如果对象存在 db link 的话，db link 的名称
NAMESPACE	VARCHAR2(28)	库缓存的对象命名空间，包括：TABLE/PROCEDURE, BODY, TRIGGER, INDEX, CLUSTER, OBJECT, CURSOR, INVALID NAMESPACE, JAVA SHARED DATA, PUB_SUB, RSRG CONSUMER GROUP

字段	数据类型	说明
TYPE	VARCHAR2(28)	对象类型，包括：INDEX, TABLE, CLUSTER, VIEW, SET, SYNONYM, SEQUENCE, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER, CLASS, OBJECT, USER, DBLINK, CURSOR, JAVA CLASS, JAVA SHARED DATA, NON-EXISTENT, NOT LOADED, PUB_SUB, REPLICATION OBJECT GROUP, TYPE
SHARABLE_MEM	NUMBER	对象消耗的共享池中的共享内存
LOADS	NUMBER	对象被载入次数。即使对象被置为无效了，这个数字还是会增长。
EXECUTIONS	NUMBER	对象执行次数，但本视图中没有被使用。可以参考视图 v\$sqlarea 中执行次数。
LOCKS	NUMBER	当前锁住这个对象的用户数（如正在调用、执行对象）。
PINS	NUMBER	当前 pin 住这个对象的用户数（如正在编译、解析对象）。
KEPT	VARCHAR2(3)	对象是否被保持，即调用了 DBMS_SHARED_POOL.KEEP 来永久将对象 pin 在内存中。 (YES   NO)
CHILD_LATCH	NUMBER	正在保护该对象的子 latch 的数量。

- **v\$sql、v\$sqlarea、v\$sqltext**

这三个视图都可以用于查询共享池中已经解析过的 SQL 语句及其相关信息。

V\$SQL 中列出了共享 SQL 区中所有语句的信息，它不包含 GROUP BY 字句，并且为每一条 SQL 语句中单独存放一条记录；

V\$SQLAREA 中一条记录显示了一条共享 SQL 区中的统计信息。它提供了有在内存中、解析过的和准备运行的 SQL 语句的统计信息；

V\$SQLTEXT 包含了库缓存中所有共享游标对应的 SQL 语句。它将 SQL 语句分片显示。

下面介绍一下我常用的 V\$SQLAREA 的结构：

字段	数据类型	说明
SQL_TEXT	VARCHAR2(1000)	游标中 SQL 语句的前 1000 个字符。
SHARABLE_MEM	NUMBER	被游标占用的共享内存大小。如果存在多个子游标，则包含所有子游标占用的共享内存大小。
PERSISTENT_MEM	NUMBER	用于一个打开这条语句的游标的生命过程中的固定内存大小。如果存在多个子游标，则包含所有子游标生命过程中的固定内存大小。
RUNTIME_MEM	NUMBER	一个打开这条语句的游标的执行过程中的固定内存大小。如果存在多个子游标，则包含所有子游标执行过程中的固定内存大小。
SORTS	NUMBER	所有子游标执行语句所导致的排序次数。
VERSION_COUNT	NUMBER	缓存中关联这条语句的子游标数。
LOADED_VERSIONS	NUMBER	缓存中载入了这条语句上下文堆 (KGL heap 6) 的子游标数。

字段	数据类型	说明
OPEN_VERSIONS	NUMBER	打开语句的子游标数。
USERS_OPENING	NUMBER	打开这些子游标的用户数。
FETCHES	NUMBER	SQL 语句的 fetch 数。
EXECUTIONS	NUMBER	所有子游标的执行这条语句次数。
USERS_EXECUTING	NUMBER	通过子游标执行这条语句的用户数。
LOADS	NUMBER	语句被载入和重载入的次数
FIRST_LOAD_TIME	VARCHAR2(19)	语句被第一次载入的时间戳。
INVALIDATIONS	NUMBER	所以子游标的非法次数。
PARSE_CALLS	NUMBER	所有子游标对这条语句的解析调用次数。
DISK_READS	NUMBER	所有子游标运行这条语句导致的读磁盘次数。
BUFFER_GETS	NUMBER	所有子游标运行这条语句导致的读内存次数。
ROWS_PROCESSED	NUMBER	这条语句处理的总记录行数。
COMMAND_TYPE	NUMBER	Oracle 命令类型代号。
OPTIMIZER_MODE	VARCHAR2(10)	执行这条的优化器模型。
PARSING_USER_ID	NUMBER	第一次解析这条语句的用户的 ID。
PARSING_SCHEMA_ID	NUMBER	第一次解析这条语句所用的 schema 的 ID。
KEPT_VERSIONS	NUMBER	所有被 DBMS_SHARED_POOL 包标识为保持 (Keep) 状态的子游标数。
ADDRESS	RAW(4   8)	指向语句的地址
HASH_VALUE	NUMBER	这条语句在 library cache 中 hash 值。
MODULE	VARCHAR2(64)	在第一次解析这条语句是通过调用 DBMS_APPLICATION_INFO.SET_MODULE 设置的模块名称。
MODULE_HASH	NUMBER	模块的 Hash 值
ACTION	VARCHAR2(64)	在第一次解析这条语句是通过调用 DBMS_APPLICATION_INFO.SET_ACTION 设置的动作名称。
ACTION_HASH	NUMBER	动作的 Hash 值
SERIALIZABLE_ABORTS	NUMBER	所有子游标的事务无法序列化的次数，这会导致 ORA-08177 错误。
IS_OBSOLETE	VARCHAR2(1)	游标是否被废除 (Y 或 N)。当子游标数太多了时可能会发生。
CHILD_LATCH	NUMBER	为了包含此游标的子 latch 数。

查看当前会话所执行的语句以及会话相关信息：

```
SQL> select a.sid||'|'||a.SERIAL#, a.username, a.TERMINAL, a.program,
s.sql_text
  2  from v$sqlarea s
  3  where a.sql_address = s.address(+)
  4  and a.sql_hash_value = s.hash_value(+)
```

```

5 order by a.username, a.sid;
... ..
SQL>

```

- ***v\$sql\_plan***

视图 V\$SQL\_PLAN 包含了 library cache 中所有游标的执行计划。通过结合 v\$sqlarea 可以查出 library cache 中所有语句的查询计划。先从 v\$sqlarea 中得到语句的地址，然后在由 v\$sql\_plan 查出它的查询计划：

```

SQL> select lpad(' ', 2*(level-1))||operation "Operation",
2          options "Options",
3          decode(to_char(id), '0', 'Cost='||nvl(to_char(position),
'n/a'), object_name) "Object Name",
4          substr(optimizer, 1, 6) "Optimizer"
5  from v$sql_plan a
6  start with address = 'C0000000FCCDEDA0'
7  and id = 0
8  connect by prior id = a.parent_id
9  and prior a.address = a.address
10 and prior a.hash_value = a.hash_value;

```

Operation	Options	Object Name	Optimizer
-----	-----	-----	-----
SELECT STATEMENT		Cost=0	CHOOSE
NESTED LOOPS			
INDEX	RANGE SCAN	CSS_BL_CNTR_IDX1	ANALYZ
INDEX	RANGE SCAN	CSS_BKG_BL_ASSN_UQ1	ANALYZ

```
SQL>
```

- ***v\$librarycache***

这个视图包含了关于 library cache 的性能统计信息，对于共享池的性能调优很有帮助。它是按照命名空间分组统计的，结构如下：

字段	数据类型	说明
NAMESPACE	VARCHAR2(15)	library cache 的命名空间
GETS	NUMBER	请求 GET 该命名空间中对象的次数。
GETHITS	NUMBER	请求 GET 并在内存中找到了对象句柄的次数（锁定命中）。
GETHITRATIO	NUMBER	请求 GET 的命中率。
PINS	NUMBER	请求 pin 住该命名空间中对象的次数。
PINHITS	NUMBER	库对象的所有元数据在内存中被找到的次数（pin 命中）。
PINHITRATIO	NUMBER	Pin 命中率。
RELOADS	NUMBER	Pin 请求需要从磁盘中载入对象的次数。
INVALIDATIONS	NUMBER	命名空间中的非法对象（由于依赖的对象被修改所导致）数。
DLM_LOCK_REQUESTS	NUMBER	GET 请求导致的实例锁的数量。

字段	数据类型	说明
DLM_PIN_REQUESTS	NUMBER	PIN 请求导致的实例锁的数量。
DLM_PIN_RELEASES	NUMBER	请求释放 PIN 锁的次数。
DLM_INVALIDATION_REQUESTS	NUMBER	GET 请求非法实例锁的次数。
DLM_INVALIDATIONS	NUMBER	从其他实例那的得到的非法 pin 数。

其中 PIN 的命中率（或未命中率）是我们系统调优的一个重要依据：

```
SQL> select sum(pins) "hits",
2          sum(reloads) "misses",
3          sum(pins)/(sum(pins)+sum(reloads)) "Hits Ratio"
4  from v$librarycache;
```

```
      hits      misses Hits Ratio
-----
84962803      288 0.99999661
```

```
SQL>
SQL> select sum(pins) "hits",
2          sum(reloads) "misses",
3          ((sum(reloads)/sum(pins))*100) "Reload%"
4  from v$librarycache;
```

```
      hits      misses   Reload%
-----
84963808      288 0.00033896
```

```
SQL>
```

当命中率小于 99%或未命中率大于 1%时，说明系统中硬解析过多，要做系统优化（增加 Shared Pool、使用绑定变量、修改 cursor\_sharing 等措施，性能优化不是本文重点，不再赘述）。

### • *v\$library\_cache\_memory*

这个视图显示了各个命名空间中的库缓存内存对象的内存分配情况。一个内存对象是为了高效管理而组织在一起的一组内部内存。一个库对象可能包含多个内存对象。

字段	数据类型	说明
LC_NAMESPACE	VARCHAR2(15)	Library cache 命名空间
LC_INUSE_MEMORY_OBJECTS	NUMBER	属于命名空间并正被在共享池使用的内存对象数。
LC_INUSE_MEMORY_SIZE	NUMBER	正在使用的内存对象的大小总（M 未单位）。
LC_FREEABLE_MEMORY_OBJECTS	NUMBER	共享池中空闲的内存对象数。
LC_FREEABLE_MEMORY_SIZE	NUMBER	空闲内存对象的大小总和（M 为单位）。

### • *v\$sgastat*

这个视图前面介绍过，是关于 SGA 使用情况的统计。其中，关于 Shared Pool 有详细的统计数据。

### 1.1.5. 重做日志缓存 (Redo Log Buffer)

Redo Log Buffer 是 SGA 中一段保存数据库修改信息的缓存。这些信息被存储在重做条目 (Redo Entry) 中。重做条目中包含了由于 INSERT、UPDATE、DELETE、CREATE、ALTER 或 DROP 所做的修改操作而需要对数据库重新组织或重做的必须信息。在必要时，重做条目还可以用于数据库恢复。

重做条目是 Oracle 数据库进程从用户内存中拷贝到 Redo Log Buffer 中去的。重做条目在内存中是连续相连的。后台进程 LGWR 负责将 Redo Log Buffer 中的信息写入到磁盘上活动的重做日志文件 (Redo Log File) 或文件组中去的。

参数 LOG\_BUFFER 决定了 Redo Log Buffer 的大小。它的默认值是 512K (一般这个大小都是足够的)，最大可以到 4G。当系统中存在很多的大事务或者事务数量非常多时，可能会导致日志文件 IO 增加，降低性能。这时就可以考虑增加 LOG\_BUFFER。

但是，Redo Log Buffer 的实际大小并不是 LOG\_BUFFER 的设定大小。为了保护 Redo Log Buffer，oracle 为它增加了保护页 (一般为 11K)：

```
SQL> select * from v$sgastat where name = 'log_buffer';
```

POOL	NAME	BYTES
	log_buffer	7139328

```
1 row selected.
```

```
SQL> show parameter log_buffer
```

NAME	TYPE	VALUE
log_buffer	integer	7028736

```
SQL>
```

### 1.1.6. 大池 (large pool)

大池是 SGA 中的一块可选内存池，根据需要时配置。在以下情况下需要配置大池：

- 用于共享服务 (Shared Server MTS 方式中) 的会话内存和 Oracle 分布式事务处理的 Oracle XA 接口
- 使用并行查询 (Parallel Query Option PQO) 时
- IO 服务进程
- Oracle 备份和恢复操作 (启用了 RMAN 时)

通过从大池中分配会话内存给共享服务、Oracle XA 或并行查询，oracle 可以使用共享池主要来缓存共享 SQL，以防止由于共享 SQL 缓存收缩导致的性能消耗。此外，为 Oracle 备份和恢复操作、IO 服务进程和并行查询分配的内存一般都是几百 K，这么大的内存段从大池比从共享池更容易分配得到 (所以叫“大”池嘛^^)。

参数 LARGE\_POOL\_SIZE 设置大池的大小。大池是属于 SGA 的可变区 (Variable Area) 的，它不属于共享池。对于大池的访问，是受到 large memory latch 保护的。大池中只有两种内存段：空闲 (free) 和可空闲 (freeable) 内存段 (关于不同类型内存段我们在后面介绍)。它没有可重建 (recreatable) 内存段，因此也不用 LRU 链表来管理 (这和其他内存区的管理不同)。大池最大大小为 4G。

为了防止大池中产生碎片，隐含参数 \_LARGE\_POOL\_MIN\_ALLOC 设置了大池中内存段的最小大小，默认值是 16K (同样，不建议修改隐含参数)。

此外，large pool 是没有 LRU 链表的。

### 1.1.7. Java 池 (Java Pool)

Java 池也是 SGA 中的一块可选内存区，它也属于 SGA 中的可变区。

Java 池的内存是用于存储所有会话中特定 Java 代码和 JVM 中数据。Java 池的使用方式依赖与 Oracle 服务的运行模式。

Java 池的大小由参数 JAVA\_POOL\_SIZE 设置。Java Pool 最大可到 1G。

在 Oracle 10g 以后，提供了一个新的建议器——Java 池建议器——来辅助 DBA 调整 Java 池大小。建议器的统计数据可以通过视图 V\$JAVA\_POOL\_ADVICE 来查询。如何借助建议器调整 Java 池的方法和使用 Buffer Cache 建议器类似，可以参考 Buffer Cache 中关于建议器部分。

### 1.1.8. 流池 (Streams Pool)

流池是 Oracle 10g 中新增加的。是为了增加对流（流复制是 Oracle 9iR2 中引入的一个非常吸引人的特性，支持异构数据库之间的复制。10g 中得到了完善）的支持。

流池也是可选内存区，属于 SGA 中的可变区。它的大小可以通过参数 STREAMS\_POOL\_SIZE 来指定。如果没有被指定，oracle 会在第一次使用流时自动创建。如果设置了 SGA\_TARGET 参数，Oracle 会从 SGA 中分配内存给流池；如果没有指定 SGA\_TARGET，则从 buffer cache 中转换一部分内存过来给流池。转换的大小是共享池大小的 10%。

Oracle 同样为流池提供了一个建议器——流池建议器。建议器的统计数据可以通过视图 V\$STREAMS\_POOL\_ADVICE 查询。使用方法参看 Buffer Cache 中关于优化器部分。

## 1.2. PGA (The Process Global Area)

PGA (Program Global Area 程序全局区) 是一块包含一个服务进程的数据和控制信息的内存区域。它是 Oracle 在一个服务进程启动是创建的，是非共享的。一个 Oracle 进程拥有一个 PGA 内存区。一个 PGA 也只能被拥有它的那个服务进程所访问，只有这个进程中的 Oracle 代码才能读写它。因此，PGA 中的结构是不需要 Latch 保护的。

我们可以设置所有服务进程的 PGA 内存总数受到实例分配的总体 PGA (Aggregated PGA) 限制。

在专有服务器 (Dedicated Server) 模式下，Oracle 会为每个会话启动一个 Oracle 进程；而在多线程服务 (Multi-Thread Server MTS) 模式下，由多个会话共享通一个 Oracle 服务进程。

PGA 中包含了关于进程使用到的操作系统资源的信息，以及一些关于进程状态的信息。而关于进程使用的 Oracle 共享资源的信息则是在 SGA 中。这样做可以使在进程以外中止时，能够及时释放和清除这些资源。

### 1.1.2. PGA 的组成

PGA 由两组区域组成：固定 PGA 和可变 PGA (或者叫 PGA 堆，PGA Heap【堆——Heap 就是一个受管理的内存区】)。固定 PGA 和固定 SGA 类似，它的大小时固定的，包含了大量原子变量、小的数据结构和指向可变 PGA 的指针。

可变 PGA 时一个内存堆。它的内存段可以通过视图 X\$KSMPP (另外一个视图 X\$KSMSP 可以查到可变 SGA 的内存段信息，他们的结构相同) 查到。PGA 堆包含用于存放 X\$表的的内存 (依赖与参数设置，包括 DB\_FILES、CONTROL\_FILES)。

总的来说，PGA 的可变区中主要分为以下三部分内容：

- 私有 SQL 区；

- 游标和 SQL 区
- 会话内存

### 1.1.2.1. 私有 SQL 区 (Private SQL Area)

前面已经说过，私有 SQL 区包含了绑定变量值和运行时期内存结构信息等数据。每一个运行 SQL 语句的会话都有一个块私有 SQL 区。所有提交了相同 SQL 语句的用户都有各自的私有 SQL 区，并且他们共享一个共享 SQL 区。因此，一个共享 SQL 区可能和多个私有共享区相关联。

一个游标的私有 SQL 区又分为两个生命周期不同的区：

- 永久区。包含绑定变量信息。当游标关闭时被释放。
- 运行区。当执行结束时释放。

创建运行区是一次执行请求的第一步。对于 INSERT、UPDATE 和 DELETE 语句，Oracle 在语句运行结束时释放运行区。对于查询操作，Oracle 只有在所有记录被 fetch 到或者查询被取消时释放运行区。

### 1.1.2.2. 游标和 SQL 区 (Cursors and SQL Areas)

一个 Oracle 预编译程序或 OCI 程序的应用开发人员能够很明确的打开一个游标，或者控制一块特定的私有 SQL 区，将他们作为程序运行的命名资源。另外，oracle 隐含的为一些 SQL 语句产生的递归调用（前面有介绍，读取数据字典信息）也使用共享 SQL 区。

私有 SQL 区是由用户进程管理的。如何分配和释放私有 SQL 区极大的依赖与你所使用的应用工具。而用户进程可以分配的私有 SQL 区的数量是由参数 OPEN\_CURSORS 控制的，它的默认值是 50。

在游标关闭前或者语句句柄被释放前，私有 SQL 区将一直存在（但其中的运行区是在语句执行结束时被释放，只有永久区一直存在）下去。应用开发人员可以通过将所有打开的不再使用的游标都关闭来释放永久区，以减少用户程序所占用的内存。

### 1.1.2.3. 会话内存 (Session Memory)

会话内存是一段用于保存会话变量（如登录信息）和其他预会话相关信息的内存。对于共享服务器模式下，会话内存是共享的，而不是私有的。

对于复杂的查询（如决策支持系统中的查询），运行区的很大一部分被那些内存需求很大的操作分配给 SQL 工作区 (SQL Work Area)。这些操作包括：

- 基于排序的操作 (ORDER BY、GROUP BY、ROLLUP、窗口函数)；
- Hash Join
- Bitmap merge
- Bitmap create

例如，一个排序操作使用工作区（这时也可叫排序区 Sort Area）来将一部分数据行在内存排序；而一个 Hash Join 操作则使用工作区（这时也可以叫做 Hash 区 Hash Area）来建立 Hash 表。如果这两种操作所处理的数据量比工作区大，那就会将输入的数据分成一些更小的数据片，使一些数据片能够在内存中处理，而其他的就在临时表空间的磁盘上稍后处理。尽管工作区太小时，Bitmap 操作不会将数据放到磁盘上处理，但是他们的复杂性是和工作区大小成反比的。因此，总的来说，工作区越大，这些操作就运行越快。

工作区的大小是可以调整的。一般来说，大的工作区能让一些特定的操作性能更佳，但也会消耗更多的内存。工作区的大小足够适应输入的数据和相关的 SQL 操作所需的辅助的内存就是最优的。如果不满足，因为需要将一部分数据放到临时表空间磁盘上处理，操作的响应时间会增长。

### 1.1.3. PGA 内存自动管理

SQL 工作区可以是自动的、全局的管理。DBA 只要设置参数 `PGA_AGGREGATE_TARGET` 给一个实例的 PGA 内存指定总的大小。设置这个参数后，Oracle 将它作为一个总的全局限制值，尽量使所有 Oracle 服务进程的 PGA 内存总数不超过这个值。

在这个参数出现之前，DBA 要调整参数 `SORT_AREA_SIZE`、`HASH_AREA_SIZE`、`BITMAP_MERGE_AREA_SIZE` 和 `CREATE_BITMAP_AREA_SIZE`（关于这些参数，我们会在后面介绍），使性能和 PGA 内存消耗最佳。对这些参数的调整是非常麻烦的，因为即要考虑所有相关的操作，使工作区适合它们输入数据大小，又要使 PGA 内存不消耗过大导致系统整体性能下降。

9i 以后，通过设置了参数 `PGA_AGGREGATE_TARGET`，使所有会话的工作区的大小都是自动分配。同时，所有 `*_AREA_SIZE` 参数都会失效。在任何时候，实例中可用于工作区的 PGA 内存总数都是基于参数 `PGA_AGGREGATE_TARGET` 的。工作区内存总数等于参数 `PGA_AGGREGATE_TARGET` 的值减去系统其他组件（如分配给会话的 PGA 内存）的内存消耗。分配给 Oracle 进程的 PGA 内存大小是根据它们对内存的需求情况来的。

参数 `WORKAREA_SIZE_POLICY` 决定是否使用 `PGA_AGGREGATE_TARGET` 来管理 PGA 内存。它有两个值：`AUTO` 和 `MANUAL`。默认是 `AUTO`，即使用 `PGA_AGGREGATE_TARGET` 来管理 PGA 内存。其实，从参数 `WORKAREA_SIZE_POLICY` 的名字上可以看出，Oracle 的 PGA 内存自动管理只会调整工作区部分，而非工作区部分（固定 PGA 区）则不会受影响。

还有注意一点就是：10g 之前，`PGA_AGGREGATE_TARGET` 只在专用服务模式下生效。而 10g 以后，PGA 内存自动管理在专有服务模式（Dedicated Server）和 MTS 下都有效。另外，9i 在 OpenVMS 系统上还不支持 PGA 内存自动管理，但 10g 支持。

设置了 `PGA_AGGREGATE_TARGET` 以后，每个进程 PGA 内存的大小也是受限制的：

- 串行操作时，每个进程可用的 PGA 内存为  $\text{MIN}(\text{PGA\_AGGREGATE\_TARGET} * 5\%, \text{\_pga\_max\_size}/2)$ ，其中隐含参数 `\_pga_max_size` 的默认值是 200M，同样不建议修改它。
- 并行操作时，并行语句可用的 PGA 内存为  $\text{PGA\_AGGREGATE\_TARGET} * 30\% / \text{DOP}$ （Degree Of Parallelism 并行度）。

### 1.1.4. 专有服务（Dedicated Server）和共享服务（Shared Server）

对 PGA 内存的管理和分配，很大程度上依赖与服务模式。下面这张表显示了在不同模式下，PGA 内存不同部分的分配的异同：

内存区	专有服务	共享服务
会话内存	私有的	共享的
永久区所在区域	PGA	SGA
SELECT 语句的运行区所在区域	PGA	PGA
DML/DDL 语句的运行区所在区域	PGA	PGA

### 1.1.5. 重要参数

PGA 的管理和分配是由多个系统参数控制的，下面介绍一下这些参数：

#### 1.1.5.1. PGA\_AGGREGATE\_TARGET

这个参数前面介绍了。它控制了所有进程 PGA 内存的总的大小。

在专有服务模式下，推荐使用 PGA\_AGGREGATE\_TARGET。

PGA\_AGGREGATE\_TARGET 的取值范围是 10M~ (4096G - 1) bytes。

对于 PGA\_AGGREGATE\_TARGET 大小的设置，Oracle 提供了一个以下建议方案（参见 Metalink Note: 223730.1）：

- 对于 OLTP 系统， $PGA\_AGGREGATE\_TARGET = (\text{物理内存大小} * 80\%) * 20\%$
- 对于 DSS 系统， $PGA\_AGGREGATE\_TARGET = (\text{物理内存大小} * 80\%) * 50\%$

例如，你的系统是一个 OLTP 系统，物理内存为 8G，那么推荐 PGA\_AGGREGATE\_TARGET 设置为  $(8 * 80\%) * 20\% = 1.28G$ 。

#### 1.1.5.2. WORKAREA\_SIZE\_POLICY

参数 WORKAREA\_SIZE\_POLICY 决定是否使用 PGA\_AGGREGATE\_TARGET 来管理 PGA 内存。

它有两个值：AUTO 和 MANUAL。默认是 AUTO，即使用 PGA\_AGGREGATE\_TARGET 来管理 PGA 内存。

#### 1.1.5.3. sort\_area\_size

Oracle 在做排序操作（ORDER BY、GROUP BY、ROLLUP、窗口函数）时，需要从工作区中分配一定内存区域对数据记录做内存排序。在排序完成后，数据返回之前，Oracle 会释放这部分内存。SORT\_AREA\_SIZE 指定了这部分内存的大小。设置了 PGA\_AGGREGATE\_TARGET 后，该参数无效。

除非在共享服务模式下，一般不推荐设置这个参数，而推荐使用 PGA\_AGGREGATE\_TARGET 进行 PGA 内存自动管理。如果需要设置此参数，可以考虑设置在 1M~3M。

Oracle 也许会为一个查询分配多个排序区。通常情况下，一条语句只有 1、2 个排序操作，但是对于复杂语句，可能存在多个排序操作，每个排序操作都有自己的排序区。因此，语句的复杂性也影响到每个进程 PGA 内存的大小。

#### 1.1.5.4. sort\_area\_retained\_size

这个参数与 SORT\_AREA\_SIZE 配合使用。它指定了在排序操作完成后，继续保留用户全局区（User Global Area UGA，关于 UGA 与 PGA、SGA 关系在 UGA 部分介绍）内存的最大大小，以维护内存中的排序，直到所有数据行被返回后才释放（上面提到，SORT\_AREA\_SIZE 的内存排序完成、数据行返回之前被释放）回 UGA（注意：是释放回 UGA，而不会被操作系统回收）。

SORT\_AREA\_RETAINED\_SIZE 在共享服务中是从 SGA 中分配的（因为此时 UGA 从 SGA 中分配），在专有服务模式中是从 PGA 中分配的。而 SORT\_AREA\_SIZE 无论在那种模式下都从 PGA 中分配。

同样，设置了 PGA\_AGGREGATE\_TARGET 后，该参数无效。

### 1.1.5.5. hash\_area\_size

HASH\_AREA\_SIZE 设置了在做 Hash Join 时, hash 内存表可占用的内存空间。同样, 设置了 PGA\_AGGREGATE\_TARGET 后, 该参数无效。它的默认值大小是 sort\_area\_size 的 1.5 倍。

此外, 由于 Hash Join 只有在优化器为 CBO (Cost-Base Optimizer) 模式下才有效, 因此这个参数也只有 CBO 模式下才有意义。

### 1.1.5.6. hash\_join\_enable

这个参数决定是否启用 Hash Join。默认为 TRUE。

由于 Hash Join 只有在优化器为 CBO (Cost-Base Optimizer) 模式下才有效, 因此这个参数也只有 CBO 模式下才有意义。

10g 中, 这个参数是隐含参数。

### 1.1.5.7. bitmap\_merge\_area\_size

在使用位图索引 (Bitmap Index) 时, oracle 为索引位图段建立一张位图。在进行位图索引扫描时, 需要将扫描到的位图索引排序后与位图合并 (Merge), Oracle 会在 PGA 中开辟一片区域用于排序和合并。参数 BITMAP\_MERGE\_AREA\_SIZE 指定了这篇区域的大小。默认值是 1M。

同样, 设置了 PGA\_AGGREGATE\_TARGET 后, 该参数无效。

### 1.1.5.8. create\_bitmap\_area\_size

在字段的集的势 (Cardinality 参照记录行数, 字段的不同值的一个因子。记录数越多, 不同值越少, 则集的势越小) 很小, 并且表的数据变化不大时, 可以考虑为字段建立位图索引以提高对该字段的检索效率。这个参数指定可在创建位图索引时的内存空间占用大小。它的默认大小是 8M。

同样, 设置了 PGA\_AGGREGATE\_TARGET 后, 该参数无效。

### 1.1.5.9. open\_cursors

这个参数设置一个会话可以同时打开的游标数。由于每打开一个游标, 都需要一部分 PGA 内存分配出来作为私有 SQL 区。因此这个参数也影响了每个进程的 PGA 内存的占用大小。

### 1.1.5.10. \_pga\_max\_size

这是一个隐含参数。它规定了一个 PGA 的最大大小。可参见 1.2.2。

## 1.1.6. 重要视图

### 1.1.6.1. V\$PGASTA

V\$PGASTAT 提供了 PGA 内存使用情况的统计信息和当自动 PGA 内存管理启动时的统计信息。视图里面的累加数据是自从实例启动后开始累加的。

字段	数据类型	说明
NAME	VARCHAR2(64)	统计的名称, 包括: <b>aggregate PGA target parameter</b> - 当前参数 PGA_AGGREGATE_TARGET 的

字段	数据类型	说明
		<p>值。如果参数没有设置，则值为 0 并且 PGA 内存自动管理被关闭。</p> <p><b>aggregate PGA auto target</b> - 在自动管理模式下，可用于工作区的总的 PGA 内存数。这个数值是动态的，和 PGA_AGGREGATE_TARGET 的值以及当前工作区的负载有关，Oracle 会动态调整它。</p> <p>这个值相对与 PGA_AGGREGATE_TARGET 来说很小。其他很大一部分的 PGA 内存都被用于系统的其他组件（如 PLSQL 和 Java 的内存）。DBA 必须保证在自动模式下有足够的 PGA 内存用于工作区。</p> <p><b>global memory bound</b> - 自动模式下可用的工作区的最大大小。Oracle 根据当前工作区的负载动态调整这个值。当系统中活动的工作区数量增加时，<b>global memory bound</b> 一般会下降。如果 global bound 降低到低于 1M，则要考虑增加 PGA_AGGREGATE_TARGET 了。</p> <p><b>total PGA allocated</b> - 当前实例分配的总的 PGA 内存大小。Oracle 会试图保持这个值在 PGA_AGGREGATE_TARGET 以内。然而，当工作区负载增加得非常快或者 PGA_AGGREGATE_TARGET 被设置得很小时，这个值也许会在一段时间内超过 PGA_AGGREGATE_TARGET。</p> <p><b>total PGA used</b> - 当前被工作区消耗得 PGA 内存。这个数值也可以用于计算有多少 PGA 内存被其他组件（如 PLSQL 或 Java）所消耗。</p> <p><b>total PGA used for auto workareas</b> - 自动模式下，当前多少 PGA 内存被工作区所消耗。这个数值也可以用于计算有多少 PGA 内存被其他组件（如 PLSQL 或 Java）所消耗。</p> <p><b>total PGA used for manual workareas</b> - 手动模式下，当前多少 PGA 内存被工作区所消耗。这个数值也可以用于计算有多少 PGA 内存被其他组件（如 PLSQL 或 Java）所消耗。</p> <p><b>over allocation count</b> - 这个数值是自从实例启动后累加的。当 PGA_AGGREGATE_TARGET 设置非常小或工作区负载增长很快时，会超额分配 PGA 内存（分配的值大于 PGA_AGGREGATE_TARGET）。这种情况发生时，Oracle 不能限制 PGA 内存小于 PGA_AGGREGATE_TARGET，只能分配实际需要的 PGA 内存。此时，建议通过建议器视图 V\$PGA_TARGET_ADVICE 来增加 PGA_AGGREGATE_TARGET 的大小。</p> <p><b>bytes processed</b> - 自从实例启动后，被内存 SQL 操作处理的字节数。</p> <p><b>extra bytes read/written</b> - 自从实例启动后，需要额外输入数据所处理的字节数。当工作区无法在最佳状态下运行时，就需要进行这个额外处理。</p> <p><b>cache hit percentage</b> - Oracle 计算出来的一个与 PGA 内存组件性能相关的数据，是自从实例启动后累加的。如果这个值是 100%，则表示实例启动后，所有系统使用到的工作区都分配了最佳的 PGA 内存。</p> <p>当工作区无法在最佳状态下运行，就需要进行额外的数据输入处理，这将会降低 cache hit percentage。</p>
VALUE	NUMBER	统计数据
UNITS	VARCHAR2(12)	数据的单位 (microseconds, bytes, or percent)

### 1.1.6.2. V\$PGA\_TARGET\_ADVICE

这个视图是可以显示 PGA 优化建议器的估算预测结果，它显示了在各种 PGA\_AGGREGATE\_TARGET 值时，V\$PGASTAT 可能会显示的 PGA 性能统计数据。选取所用来预

测的 PGA\_AGGREGATE\_TARGET 值是当前 PGA\_AGGREGATE\_TARGET 左右的值。而估算出的统计值是根据实例启动后的负载模拟出来的。

只有当建议器打开（隐含参数 `smm_advice_enabled` 为 TRUE），并且参数 `STATISTICS_LEVEL` 值不是 BASIC 时，视图中才会有内容。实例重启后，所有预测数据都会被重写。

字段	数据类型	说明
PGA_TARGET_FOR_ESTIMATE	NUMBER	用于预测的 PGA_AGGREGATE_TARGET 值。
PGA_TARGET_FACTOR	NUMBER	预测的 PGA_AGGREGATE_TARGET 与当前 PGA_AGGREGATE_TARGET 的比。
ADVICE_STATUS	VARCHAR2(3)	建议器状态（ON 或 OFF）
BYTES_PROCESSED	NUMBER	预测的被所有工作区处理的字节数。
ESTD_EXTRA_BYTES_RW	NUMBER	当 PGA_AGGREGATE_TARGET 设置为预测值时，需要额外读写的字节数。
ESTD_PGA_CACHE_HIT_PERCENTAGE	NUMBER	当 PGA_AGGREGATE_TARGET 设置为预测值时，缓存命中率。这个值等于 $\text{BYTES\_PROCESSED} / (\text{BYTES\_PROCESSED} + \text{ESTD\_EXTRA\_BYTES\_RW})$
ESTD_OVERALLOC_COUNT	NUMBER	当 PGA_AGGREGATE_TARGET 设置为预测值时，需要超额分配的 PGA 内存。如果非 0 则说明 PGA_AGGREGATE_TARGET 设置得太小。

### 1.1.6.3. V\$SYSSTAT、V\$SESSTAT

这两个视图显示了系统（会话）的统计数据。他们的统计项目基本相同，但不同之处在于一个是系统级的、一个是会话级的。

通过这两个视图我们可以查出像 `sort` 这样操作对工作区的使用情况：

```
SQL> select * from V$SYSSTAT
      2 where name like '%sort%';
```

```
STATISTIC# NAME
CLASS      VALUE
-----
          245 sorts (memory)
64         2876455
          246 sorts (disk)
64           483
          247 sorts (rows)
64        116554720
SQL>
```

### 1.1.6.4. V\$SQL\_WORKAREA

这个视图显示了被 SQL 游标使用的工作区的信息。存储在 Shared Pool 中的每条 SQL 语句都有一个或多个子游标，它们能被 V\$SQL 显示。而 V\$SQL\_WORKAREA 显示需要被这些游标所使用的工作区信息。可以将它与 V\$SQL 进行 join 查询。

通过这个视图可以解决以下一些问题：

- 1、请求最多的工作区；
- 2、在自动模式下，占用内存最多的工作区。

字段	数据类型	说明
ADDRESS	RAW(4   8)	游标句柄的地址。
HASH_VALUE	NUMBER	游标句柄的 Hash 值。这个字段和 ADDRESS 字段 join V\$SQLAREA 可以定位出相关语句。
CHILD_NUMBER	NUMBER	使用此工作区的子游标数。
WORKAREA_ADDRESS	RAW(4   8)	工作区句柄的地址。唯一定位了一条记录
OPERATION_TYPE	VARCHAR2(20)	工作区的操作类型(SORT, HASH JOIN, GROUP BY, BUFFERING, BITMAP MERGE, or BITMAP CREATE)
OPERATION_ID	NUMBER	唯一定位查询计划中的一个操作的值，可以和视图 V\$SQL_PLAN join。
POLICY	VARCHAR2(10)	工作区的模式(MANUAL 或 AUTO)
ESTIMATED_OPTIMAL_SIZE	NUMBER	估计需要此工作区来执行内存中操作的所需的大小。
ESTIMATED_ONEPASS_SIZE	NUMBER	估计需要此工作区来一次执行内存中操作的所需的大小。
LAST_MEMORY_USED	NUMBER	最后一次执行游标所使用的工作区大小。
LAST_EXECUTION	VARCHAR2(10)	最后一次执行游标，工作区请求内存的方式，OPTIMAL, ONE PASS, ONE PASS 或 MULTI-PASS。
LAST_DEGREE	NUMBER	最后一次执行并行操作的并行度 (Degree of parallelism DOP)。
TOTAL_EXECUTIONS	NUMBER	此工作区激活的次数。
OPTIMAL_EXECUTIONS	NUMBER	此工作区运行于 optimal 模式的次数
ONEPASS_EXECUTIONS	NUMBER	此工作区运行于 one-pass 模式的次数
MULTIPASSES_EXECUTIONS	NUMBER	此工作区运行于 one-pass 内存请求情况下的次数
ACTIVE_TIME	NUMBER	此工作区激活的评价时间数。
MAX_TEMPSEG_SIZE	NUMBER	实例化此工作区所创建的临时段的最大大小。
LAST_TEMPSEG_SIZE	NUMBER	最后一次实例化此工作区所创建的临时段的大小。

### 1.1.6.5. V\$SQL\_WORKAREA\_ACTIVE

这个视图包含了系统当前分配的工作区的瞬间信息。可以通过字段 WORKAREA\_ADDRESS join V\$SQL\_WORKAREA 来查询工作区信息。如果工作区溢出到磁盘，则这个视图就包含了这个工作区所溢出的临时段的信息。通过与视图 V\$TEMPSEG\_USAGE join，可以得到更多的临时段信息。

这个视图可以解决以下问题：

- 1、当前系统分配最大的工作区；

- 2、超额分配内存的百分比 (EXPECTED\_SIZE < ACTUAL\_MEM\_USED) , 和未超额分配内存的百分比 (EXPECTED\_SIZE > ACTUAL\_MEM\_USED) ;
- 3、哪个活动的工作区使用了超出内存管理预期的内存大小;
- 4、那个活动的工作区溢出到磁盘了。

字段	数据类型	说明
WORKAREA_ADDRESS	RAW(4   8)	工作区句柄的地址。唯一定位了一条记录。
OPERATION_TYPE	VARCHAR2(20)	使用此工作区的操作 (SORT, HASH JOIN, GROUP BY, BUFFERING, BITMAP MERGE, 或 BITMAP CREATE)
OPERATION_ID	NUMBER	唯一定位查询计划中的一个操作的值, 可以和视图 V\$SQL_PLAN join。
POLICY	VARCHAR2(6)	工作区的模式 (MANUAL 或 AUTO)
SID	NUMBER	会话 ID
QCINST_ID	NUMBER	查询协调 (查询协调在并行查询中出现) 者实例 ID。
QCSID	NUMBER	查询协调者的会话 ID。
ACTIVE_TIME	NUMBER	此工作区激活的平均时间 (厘秒为单位)
WORK_AREA_SIZE	NUMBER	被当前操作使用的最大工作区大小。
EXPECTED_SIZE	NUMBER	工作区的预期大小。预期大小是内存管理器设置的。当 WORK_AREA_SIZE 大于 EXPECTED_SIZE 时, 内存会超额分配。
ACTUAL_MEM_USED	NUMBER	当前分配个工作区的 PGA 内存大小 (KB)。这个值在 0 和 WORK_AREA_SIZE 之间。
MAX_MEM_USED	NUMBER	这个工作区使用的最大大小 (KB)。
NUMBER_PASSES	NUMBER	这个工作区的通道数 (如果在 OPTIMAL 模式下为 0)
TEMPSEG_SIZE	NUMBER	用于此工作区的临时段大小。
TABLESPACE	VARCHAR2(31)	创建临时段给这个工作区的表空间名字。
SEGRFNO#	NUMBER	创建临时段的表空间的文件 ID。
SEGBLK#	NUMBER	给工作区创建临时段的 block 数。

### 1.1.6.6. V\$PROCESS

这个视图显示了所有 Oracle 进程的信息。其中以下几个字段则说明了进程 PGA 内存的使用情况。

- PGA\_USED\_MEM: 进程使用的 PGA 内存
- PGA\_ALLOCATED\_MEM: 分配给进程的 PGA 内存
- PGA\_MAX\_MEM: 进程使用的最大的 PGA 内存。

## 1.3. UGA (The User Global Area)

PGA 是一段包含一个 Oracle 服务或后台进程的数据和控制信息的内存。PGA 的大小依赖于系统的配置。在专用服务 (Dedicated Server) 模式下, 一个服务进程与一个用户进程相关, PGA 就包括了堆空间和 UGA。而 UGA (User Global Area 用户全局区) 由用户会

话数据、游标状态和索引区组成。在共享服务 (MTS) 模式下, 一个共享服务进程被多个用户进程共享, 此时 UGA 是 Shared Pool 或 Large Pool 的一部分 (依赖与配置)。

许多 DBA 都不理解 PGA 和 UGA 之间的区别。其实这种区别可以简单的理解为进程和会话直接的区别。在专用服务模式下, 进程和会话是一对一的; 而在 MTS 模式下, 进程和会话是一对多的关系。PGA 是服务于进程的, 它包含的是进程的信息; 而 UGA 是服务于会话的, 它包含的是会话的信息。因此, MTS 模式下, PGA 和 UGA 之间的关系也是一对多的。

UGA 中包含了一个会话的信息, 包括:

- 打开游标的永久区和运行区;
- 包的状态信息, 特别是包的变量;
- Java 会话的信息;
- 激活的角色;
- 激活的跟踪事件 (ALTER SESSION SET EVENT ...);
- 起作用的 NLS 参数 (SELECT \* FROM NLS\_SESSION\_PARAMETERS);
- 所有打开的 db link;
- 会话对于信任的 Oracle 的托管访问标记 (mandatory access control (MAC))

和 PGA 一样, UGA 也由两组区组成, 固定 UGA 和可变 UGA (或者说 UGA 堆)。固定 UGA 包含了大概 70 个原子变量、小的数据结构以及指向 UGA 堆的指针。

UGA heap 中的段可以通过表 X\$KSMUP 查到 (它的结构和 X\$KSMSP 相同)。UGA 堆包含了存储一些固定表 (X\$表) 的永久内存 (依赖与特定参数的设置, 如 OPEN\_CURSORS, OPEN\_LINKS 和 MAX\_ENABLED\_ROLES)。除此以外, 大部分的 UGA 用于私有 SQL 区。UGA 内存的所在依赖于会话的设置。在专用服务模式下, 会话和进程是一对一的关系, UGA 位于 PGA 中。固定 UGA 是 PGA 中的一段内存段, 而 UGA 堆是 PGA 的子堆。在 MTS 模式下, 固定 UGA 是 shared pool 中的一段内存段, 而 UGA 堆是 Large Pool 的子堆, 如果从 large pool 分配失败, 则从 shared pool 中分配。

MTS 模式下, 可以通过 Profile 中的 PRIVATE\_SGA 项 (通过 dba\_profiles 查看) 来控制每个 UGA 占用的 SGA 的总的大小, 但是不建议这样做。

Oracle 9.2 以后, 有一个新的隐含参数: `_use_realfree_heap`。当设置这个参数为 true 时, Oracle 会为 CGA、UGA 单独分配堆, 而不从 PGA 中分配。它的默认值为 false, 而当设置了 `pga_aggregate_target` 后, 它的值自动被改为 true。

## 1.4. CGA (The Call Global Area)

与其他的全局区不同, CGA (Call Global Area 调用全局区) 的存在是瞬间的。它只存在于一个调用过程中。对于实例的一些低层次的调用需要 CGA, 包括:

- 解析一条 SQL 语句;
- 执行一条 SQL 语句;
- 取一条 SELECT 语句的输出值。

如果语句产生了递归调用, 则需要为每个递归调用分配一个 CGA。如上所述, 递归调用是在语句解析、优化器产生语句查询计划、DML 操作时需要查询或修改数据字典信息的调用。

无论 UGA 存在于 PGA 还是 SGA, CGA 都是 PGA 的 subheap。因为无论那种模式, 会话在做调用时总需要一个进行进行处理。这一点很重要, 特别是在 MTS 模式下时, 如果发现一次调用很久没有响应, 则可能需要增加 PGA 的大小。

当然, 调用并不是只通过 CGA 中的数据结构来工作。实际上, 调用所需要的大部分的重要数据结构都来自于 UGA。例如私有 SQL 取和排序区都存放在 UGA 中, 因为调用结束后, 它们是被保留的。CGA 中只包含了那些调用结束后可以被释放的数据。例如, CGA 中

包含了直接 IO 缓存、关于递归调用的信息、用于表达式评估（产生查询计划时）的堆空间和其他一些临时数据。

Java 调用内存也分配在 CGA 中。它被分为三部分空间：堆空间、新空间和老空间。在调用期间（调用长短依赖于使用期长短和大小），在新空间和老空间中的内存段不再使用的内存段将被垃圾收集器回收。

## 1.5. 软件代码区 (Software Code Area)

软件代码区是一部分用于存放那些正在运行和可以被运行的代码（Oracle 自身的代码）的内存区。Oracle 代码一般存储在一个不同于用户程序存储区的软件代码区，而用户程序存储区是排他的、受保护的区域。

软件区的大小一般是固定的，只有 Oracle 软件升级或重装后才会改变。在不同操作系统下，这部分区域所要求的大小也不同。

软件区是只读的，可以被安装成共享的或非共享的。可能的情况下，Oracle 代码是共享的，这样所有 Oracle 用户都可以直接访问这些代码，而不需要各自保存一份拷贝在自己的内存中。这样可以节省大量内存并提高整体性能。

而用户程序也可以是共享的或非共享的。一些 Oracle 工具（如 SQL Plus）能被安装成共享的，但有些不能。如果一台机器运行多个实例，这些实例可以使用同一个 Oracle 代码区。

另外要注意的是：并不是所有操作系统都能将软件区安装成共享的，如 Windows。

## 2. Oracle 的内存管理

在这部分章节中，我们将从更底层的角度来了解 Oracle 的内存管理。当然，涉及到内存管理，就不可避免的要了解 OS 对内存的管理，在这以章节中，将谈到不少关于 OS 内存的管理知识。如果概念记得不清除了，可以找一本《操作系统》的书看看先。

### 2.1. Oracle 内存管理基础

要了解内存管理，首先需要知道虚拟内存。而要了解虚拟内存，就先要知道 CPU 寻址。我们知道，目前主流的 CPU 都是 32 位或者 64 位的。那么这个位数指的是什么呢？就是指 CPU 的最大寻址能力。在 CPU 中，所有一切都是二进制表示的，那么一个 32 位 CPU 的寻址范围就是  $2^{32}=4G$ 。但有可能一台机器的实际物理内存没有这么大，比如说只有 2G。而程序员在编程的时候根本不用考虑实际物理内存多大，还是按照 4G 的内存来分配。这时，OS 就提出了一个虚拟内存的概念，如果所寻址的数据实际上不在物理内存中，那就从“虚拟内存”中来获取。这个虚拟内存可以是一个专门文件格式的磁盘分区（比如 UNIX 下的 swap 分区），也可以是硬盘上的某个足够大的文件（比如 win 下的那个 i386 文件，好像是这个名字）。物理内存中长期不用的数据，也可以转移到虚拟内存中。这样的交换由 OS 来控制，用户看起来就好像物理内存大了一样。

虚拟内存寻址的一个好处就是可以时进程使用很大的虚拟内存地址，而无需考虑实际的物理内存的大小。这使得进程内存可以基于使用需要，从逻辑上分为几个不同段。这些段可能映射到不连续的虚拟内存地址上，以使内存能够增加。

为了共享 RAM，需要有一个叫做交换磁盘（swap disk）的特殊磁盘空间。交换磁盘的重要目的是保存程序的通过 LRU 算法换出的内存，这样可以使很多程序能够共享有限的 RAM。一旦非活动程序的 RAM 页被写入交换磁盘（Page Out），操作系统可以使空出来的内存用于其他活动的程序。如果非活动程序稍后又继续执行，被 page out 的 RAM 页又重

新从交换磁盘中载入 RAM (Page in)。这种重新载入 RAM 页的动作就叫交换，而交换是非常消耗时间的，并且会降低相关程序的性能。

尽管交换磁盘确保并发的 RAM 使用量能大于实际的 RAM 总量，但是为了确保最佳性能，交换空间绝不要被活动程序所使用。这是因为从交换磁盘中读取 page out 的 RAM 页要比直接从 RAM 中读取内存页要慢 14000 倍。磁盘访问都是以毫秒计的，而 RAM 访问是以 10 亿份之一秒计的。

### 2.1.1. Oracle 的内存段类型

段 (Segment) 在 OS 上是对不同内存的使用目的和存放位置不同的区分。和一般的程序一样，Oracle 使用以下几种段类型：

- **程序文本 (Program Text)**

文本段包括了程序本身的可执行的机器代码 (除动态链接库以外)。文本段一般标识为只读，因此它不能被多个进程共享来跑同一个程序。

- **初始化全局数据 (Initialized Global Data)**

这一段包括了被编译器初始化的全局数据，比如用于跟踪数据的字符串。初始化数据能被修改，因此它不能被运行同一程序的多个进程共享。Oracle 很少使用这个段。

- **未初始化全局数据 (Uninitialized Global Data)**

未初始化全局数据一般称为 BSS (Block Started by Symbol 以符号开始的块) 段。这一段包括了静态分配的全局数据，这些数据在进程运行时被进程初始化。Oracle 也很少使用这个段。

- **数据堆 (Data Heap)**

数据堆被用于进程在运行时，通过使用系统调用 `malloc()` 或 `sbrk()` 动态分配内存。Oracle 将数据 heap 用于 PGA。

- **执行堆栈 (Execution Stack)**

无论什么时候一个函数被调用，它的参数和返回上下文被 push 到一个执行堆栈中。返回上下文实际上是一组 CPU 注册值，这些注册值描述了进程在调用函数时那一刻的状态。当调用结束后，堆栈被 POP 而上下文被保留，以使执行能从函数调用时的结构状态立即执行下去。堆栈同时还保留了代码块的本地变量。堆栈大小依赖于函数嵌套或递归调用的深度、参数和本地变量所需的内存大小。

- **共享库 (Shared Libraries)**

共享库是一个与位置无关的可执行代码集，这个集合实现了许多程序——特别是系统调用功能——所需要的功能。共享库段也是只读的，它被所有的进程 (包括 Oracle 进程) 共享。共享库无需保存一份在内存中。当调用了共享库中的一个函数后，进程需要打开共享库文件，然后通过系统调用 `mmap()` 将它映射到它的地址空间去。

使用共享库的另外一种方法是在程序文本段本身将需要的系统调用 `include` 进去。在那些不支持共享库的操作系统中或用上面方式有问题时就需要这样做。在大多数操作系统中，Oracle 使用共享库作为来实现系统调用而不是实现 Oracle 代码本身。然而，Java 类库都是编译好的，并且作为共享库动态链接的。

- **共享内存段 (Shared Memory Segment)**

共享内存允许关联的进程共同读写内存中的同样数据。每个需要在共享内存段中寻址的进程都需要先将这段内存附到它自己的虚拟内存地址中去 (一般通过 `shmat()` 系统调用实现)。Oracle SGA 就是使用的共享内存段。

## 2.1.2. Oracle 的内存管理模块

Oracle 中有两个内存管理模块。一个是内核服务内存管理模块 (Kernel Service Memory KSM)；一个是内核通用堆管理模块 (Kernel Generic Heap KGH)

在 X\$表中，有两种用于这两个模块的表，它们就是以 KSM 和 KGH 开头的表。这两个模块相互非常紧密。内存管理模块是负责与操作系统进行接口以获取用于 Oracle 的内存，同时还负责静态内存的分配。这个模块中比较重要的 X\$表是 X\$ksmfs，它记录了固定 sga、buffer cache、log buffer 在内核服务内存中的分配。而堆管理模块则负责动态内存的管理。这也就是为什么 SGA 和 PGA 中堆又叫可变内存区了。Shared Pool、Library cache 和 PGA 的堆都是由这个模块管理的。

一个堆 (Heap) 包括一个堆描述符和一个或多个内存扩展段 (extent)。一个堆还可以包含子堆 (Subheap)。这种情况下，堆描述符和子堆的扩展段可以被视为其父堆的大块 (chunk)。堆描述符的大小依赖于堆的类型和堆的空闲列表和 LRU 列表所包含的列表 (Header) 头的多少。一个扩展段又一个包含指向前一个和后一个扩展段指针 (Pointer) 的小的头部，扩展段的其他内存就是堆可用于动态分配的内存。

除了还包含一个保留列表的这一特性外，Shared Pool 中的子堆具有与 Shared Pool 本身相同的结构。内存是以 Chunk 为单位分配的。空闲的 chunk 按照大小来组织在相应的空闲列表 (Free List) 中。而未 pin 住的、可重建 (unpinned recreatable) 的 chunk 被维护在两个分别用于周期性 chunk 和短期 chunk 的 LRU 链表中。子堆还有一个包含少许空闲内存的主永久内存 chunk。子堆也许还包含子堆，一共可以嵌套到四层。

子堆的概念非常重要，因为大多数被缓存在 shared pool 中的对象实际上都被保存在子堆中，而不是保存在最上一层的堆中。在一个子堆中寻找空闲空间就相当于在 shared pool 中寻找空闲空间。它们的不同处就在于子堆可以通过分配新的扩展段 (extent) 来增长，而 shared pool 具有固定的扩展段数 (10g 引入 SGA 自动管理特性后，shared pool 的扩展段数也是可变的)。为子堆分配新的扩展段受到扩展段大小的限制，因此会存在这样的可能，因为没有任何一个父堆可以分配一个所需最小扩展段大小的 chunk 导致在子堆中查找一个小的 chunk 失败 (抛 4031 错误)。

为了减少内存错误，在 10g 中，引入了多个隐含参数对扩展段进行控制，

`_bct_buffer_allocation_min_extents` — 每次 buffer cache 分配时的最小扩展段数 (1)。

`_compilation_call_heap_extent_size` 编译调用时分配堆的扩展段的大小 (16384) 10gR2 引入。

`_kgl_fixed_extents` library cache 内存分配时是否使用固定的扩展段大小 (TRUE)，10gR2 引入。

`_mem_std_extent_size` 固定扩展段大小的堆的标准扩展段大小 (4096)，10gR2 引入。

`_minimum_extents_to_shrink` 当内存收缩时，收缩的最少扩展段数 (1)

`_total_large_extent_memory` 分配大扩展段的内存数 (0)，10gR1 引入，10gR2 中已经废除。

`_pga_large_extent_size` (1048576) 和 `_uga_cga_large_extent_size` (262144) 控制了当使用系统函数 `mmap()` 初始化时，PGA、UGA 和 CGA 扩张段的最大大小。

### 2.1.3. 内存分配颗粒 Granule

在 Oracle 的内存分配和管理中，有一个重要的单位：Granule（颗粒）。granule 是连续虚拟内存分配的单位。Granule 的大小依赖于 SGA 的总的大小（即 SGA\_MAX\_SIZE 大小）。当 SGA 小于 128M 时，granule 为 4M，SGA 大于 128M 时，granule 为 16M。

Buffer Cache、Shared Pool、Large Pool 和 Java Pool（在 10g 中，还包括 Streams Pool、KEEP buffer cache、RECYCLE buffer cache、nK Buffer Cache、ASM Buffer Cache）的增长和收缩都是以 granule 为单位的。SGA 的各个组件的大小增长、收缩情况可以通过视图 `v$sga_dynamic_components`（这个视图在 1.1.2 中有介绍）来观察。

在实例启动时，Oracle 先分配 granule 条目（Entry），使所有 granule 能支持到 SGA\_MAX\_SIZE 的空间大小。如果没有设置 PRE\_PAGE\_SGA 和 LOCK\_SGA，在实例启动时，每个组件请求它所需要的最少 granule。

因此最小 SGA（占用物理内存）是 3 个 granule，包括：

- 固定 SGA（包括 redo buffer）一个 granule
- Buffer Cache 一个 granule
- Shared Pool 一个 granule

我们可以通过“ALTER SYSTEM”命令来修改分配给各个组件的 granule 数。当 DBA 想要给组件增加 granule 时，需要考虑实例中是否还有足够的 granule 来分配给新增加的组件大小（即增大后，各个组件之和小于 SGA\_MAX\_SIZE）。有一点要注意，ALTER SYSTEM 指定的是新的组件大小，是内存的大小，不是 granule 数。而 Oracle 在分配内存时，会以 granule 为单位，如果新分配的大小不是 granule 大小的倍数，则会使用最接近且大于分配数的 granule 的倍数值。例如，Unix 中，当 granule 为 16M，新分配组件大小为 120M（不是 16 的倍数），Oracle 则会实际分配 128M（16×8）；32 位 windows 下，SGA 小于 1G 时 granule 是 4M，大于 1G 时 granule 是 8M。

执行 ALTER SYSTEM 扩展组件大小后，前台进程（即执行 ALTER SYSTEM 命令）的进程会将 SGA 中的可用的 granule（按照扩展新增大小计算）先保留下来。当这些 granule 被保留后，前台进程将后续处理交给后台进程。后台进程负责将这些保留的 granule 加到指定的组件的 granule 列表中去。这就完成了一次对 SGA 组件的扩展。

Granule 的默认大小是根据以上规则规定的，但也可以通过隐含参数 `_ksm_granule_size` 来修改（强烈不建议修改）。另外，隐含参数 `_ksmg_granule_locking_status` 可以设置内存分配是否强制按照 granule 为单位进行分配。

### 2.1.4. SGA 内存

当一个 Oracle 实例启动后，主要的 SGA 区的大小一开始基于初始化参数计算得出。这些大小可以通过 `show sga` 显示（实例启动后也会显示出这些信息）。但是，在共享内存段分配之前，每个区（Area）的大小都只有大概一个内存页大小。当需要时，这些区被分为一些子区（sub-area），因此没有一个子区会大于操作系统所限制的共享内存段（UNIX 下受 SHMMAX 限制）的大小。对于可变区，有一个操作系统规定的最小子区大小，因此可变区的大小是最小子区大小的倍数。

如果可能，Oracle 会为整个 SGA 分配一个单独的共享内存段。然而，如果 SGA 大于操作系统限制的单个共享内存段的大小时，Oracle 会使用最佳的算法来将所有子区组织在多个共享段中，并且不超过 SGA 最大大小的限制。

严重的页入/页出会导致很严重的系统性能问题。然而，大内存消耗导致的间断的页入/页出是没有影响的。大多数系统都有大量的非活动内存被 page out 而没有什么性能影响。但少量的页出也是有问题的，因为这会导致 SGA 中那些中度活性的页会经常 page out。大多数操作系统提供了一个让 Oracle 锁住 SGA（设置 lock\_sga 参数为 TRUE）到物理内存中的机制以防止 page out。在某些操作系统中，oracle 需要有特定的系统权限来使用这一特性。

当共享池分配了一个 chunk，代码会返回给执行分配的函数一个注释。这些注释可以通过表 X\$KSMSP 的字段 KSMCHCOM 查到。它们同时描述了这块内存分配的目的。如以下语句：

```
select ksmchcom, ksmchcls, ksmchsiz from x$ksmsp;
```

### 2.1.5. 进程内存

进程包括程序代码（文本）、本地数据域（进程堆栈、进程堆【主要是 PGA】和进程 BSS【未初始化的全局数据】）和 SGA。程序代码的大小由基本内核、内核、联机的网络情况以及所使用的操作系统决定的。SGA 大小是由 Oracle 初始化参数决定的。而这两部分是共享。随着用户的增加，它们与单个 Oracle 服务进程的关系越来越小。它们是可以修改 Oracle 配置参数来改变的。

本地数据域中的堆栈是根据需要来增大、缩小的。然而，堆就不会释放内存了。堆的主要组成部分是 PGA。而影响 PGA 的主要因素是 sort\_area\_size（如果没有配置 PGA\_AGGREGATE\_TARGET 的话）。因此，非自动 PGA 内存管理模式下，可以通过控制 sort\_area\_size 来控制 PGA 的大小。

因此，总的来说，可以有以下方法来限制进程内存大小：

- 1、降低相关的内核参数；
- 2、通过 Oracle 参数来降低 SGA；
- 3、通过减小 sort\_area\_size 来降低 PGA。

在 UNIX 平台中，一般通过操作系统命令如“ps”或“top”来定位一个经常的内存大小。这些工具可以预警那些大量占用内存的进程。但是，这些工具统计出来的 Oracle 进程的内存情况往往是实际 PGA 内存是有出入的。

这是为什么呢？有两种原因会导致错误的进程内存报告错误：将那些非进程私有的（如共享内存）计算进去了；操作系统没有回收空闲内存。下面详细解释它们。

- **统计了非私有内存**

一个内存中的进程包含以下几个部分：

- 共享内存（SGA）
- 共享库（包括公用和私有的）
- 私有数据（指数据段【DATA】或堆）
- 可执行部分（指文本【TEXT】段）

而 SGA 和 TEXT 部分是被所有 Oracle 进程共享的。它们只会被映射到内存中一次，而不会未每个进程做映射。因此，这些内存不是一个新的 Oracle 进程导致的内存增加部分。

- **操作系统没有回收空闲内存**

通常，一部分内存被一个进程释放了后，并没有立即返回到操作系统的空闲池中，而是继续保持与进程的关联，直到操作系统内存不足时，才会回收这些空闲页。所以操作系统工具报告的进程内存大小可能会被实际大。

从 Oracle 的角度来看，一个服务进程的私有内存包括多个 Oracle “堆（heap）”。在 Oracle 的术语中，堆就是一个受管理的内存区。而对于操作系统来说，这仅仅时分配给一个应用程序的另外一块内存而已。PGA 和 UGA 中都关联到堆。

Oracle 当前或者曾经在哪些堆中拥有的内存总数可以通过以下语句统计出来：

```
SQL> select statistic#, name, value
      2   from v$sysstat
      3   where name like '%ga memory%';
```

STATISTIC#	NAME	VALUE
20	session uga memory	8650004156
21	session uga memory max	778811244
25	session pga memory	50609488
26	session pga memory max	58007200

查询所有会话的堆大小可以用以下语句实现：

```
select value, n.name|| '('||s.statistic#||')' , sid
from v$sesstat s , v$statname n
where s.statistic# = n.statistic#
and n.name like '%ga memory%'
order by value;
```

但是，查询出来大的 PGA 或 UGA 并不一定说明有问题。它们的大小受到以下参数影响：

- SORT\_AREA\_SIZE
- SORT\_AREA\_RETAINED\_SIZE
- HASH\_AREA\_SIZE

另外，过多的使用 PL/SQL 结构体（如 PL/SQL TABLE、ARRAY）也会导致会话内存增大。

## 2.2. Oracle 的内存的分配、回收

Oracle 中的共享内存区的分配都是以 chunk 为最小单位的。Chunk 不是一个固定值，它是一个扩展段（extent）中一块连续的内存。而 Oracle 的内存（其他存储，如磁盘也是）的增长是以扩展段为基础的。

### 2.2.1. 空闲列表和 LRU 链表

空闲的 chunk 按照大小来组织在相应的空闲列表（Free List）中。而未 pin 住的、可重建（unpinned recreatable）的 chunk 被维护在两个分别用于周期性 chunk 和短期 chunk 的 LRU 链表中。子堆还有一个包含少许空闲内存的主永久内存 chunk。

### 2.2.2. 空闲内存分配和回收

空闲内存都是由空闲列表（free list）统一管理、分配的。每个空闲的 chunk（大块）都会属于也只属于一个空闲列表。空闲列表上的 chunk 的大小范围是由 bucket 来划分的。Bucket 直译为“桶”，在西方，往往用桶来盛装一定品质范围物品，以便查找。比如，在采矿时，用不同的桶来装不同纯度的矿石，在桶上标明矿石的纯度范围，以便在提炼时可以采用不同工艺。在这里，我们也可以把 bucket 视为一种索引，使 Oracle 在查找空闲块时，先定位所需的空闲块在哪个 bucket 的范围内，然后在相应的空闲列表中查找。

一次内存的分配过程如下：当一个进程需要一个内存的大块（chunk）时，它会先扫描目标空闲列表（每个空闲列表对应有一个 bucket，bucket 是这个空闲列表的中 chunk 的

大小范围)以查找最适合大小的 chunk。如果找不到一个大小正好合适的 chunk,则继续扫描空闲列表中更大的 chunk。如果找到的可用 chunk 比所需的大小大 24 字节或者更多,则这个 chunk 就会被分裂,剩余的空闲 chunk 又被加到空闲列表的合适位置。如果空闲列表中没有一个 chunk 能满足要求的大小,则会从非空的相邻 bucket 的空闲列表中取最小的 chunk。如果所有空闲列表都是空的,就需要扫描 LRU 链表释放最近最少使用的内存。当 chunk 空闲时,如果相邻的 chunk 也是空闲的,它们可能会结合(coalesce)起来。

当所有空闲列表都没有合适的空闲 chunk 可用时,就开始扫描 LRU 链表,将最近最少使用的 chunk 释放掉(如有空闲的相邻 chunk,则结合),放到相应的空闲列表中去,直到找到合适的 chunk 或者达到最大查找数限制。如果在释放了 LRU 链表中最近最少使用的 chunk 后没有找到合适空闲 chunk,系统就抛 4031 错误。

如果找到了可用空闲 chunk,就将它从空闲列表中移出,放到 LRU 链表中去。

下面介绍一下 Shared Pool 的分配和回收。

### 2.2.3. Shared Pool 的分配和回收

在 Shared Pool 中,空闲列表扫描、管理和 chunk 分配的操作都是受到 shared pool latch 保护的。显然,如果 shared pool 含有大量的非常小的空闲 chunk,则扫描空闲列表时间将很长,而 shared pool latch 则会要保持很久。这就是导致 shared pool latch 争用的主要原因了。有些 DBA 通过增加 shared pool 来减少 shared pool latch 争用,这种方法是没有用的,可能反倒会加剧争用(作为短期解决方法,可以 flush shared pool;而要真正解决问题,可以采取在实例启动时 keep 住那些可能会断断续续使用的对象【这种对象最容易导致 shared pool 碎片】)。

只有执行了 ALTER SYSTEM FLUSH SHARED\_POOL 才会使 shared pool 的空闲 chunk 全结合起来。因此,即使 shared pool 空闲内存之和足够大,也可能出现内存请求失败(空闲内存都是一些很小的碎片 chunk)。

实际上,oracle 实例启动时,会保留大概一半的 Shared Pool,当有内存压力时逐渐释放它们。Oracle 通过这种方法限制碎片的产生。Oracle 的少量空袭内存(spare free memory)和 X\$表即其他持久内存结构一起,隐含在 shared pool 的主要持久内存 chunk 中。这些内存不在 shared pool 的空闲列表中,因此能够立即被分配。但是,却包含在视图 V\$SGASTAT 的 free memory 的统计数据中。

```
SQL> select * from v$sgastat
      2  where pool = 'shared pool'
      3  and name = 'free memory';
```

POOL	NAME	BYTES
shared pool	free memory	18334468

```
SQL>
```

而 spare free memory 可以用以下方式查出:

```
select
  avg(v.value) shared_pool_size,
  greatest(avg(s.ksmsslcn) - sum(p.ksmchsiz), 0) spare_free,
  to_char(
    100 * greatest(avg(s.ksmsslcn) - sum(p.ksmchsiz), 0) / avg(v.value),
    '99999'
  ) || '%' wastage
from
  sys.x$ksmss s,
  sys.x$ksmsp p,
  sys.v$parameter v
where
```

```
s.inst_id = userenv('Instance') and
p.inst_id = userenv('Instance') and
p.ksmchcom = 'free memory' and
s.ksmssnam = 'free memory' and
v.name = 'shared_pool_size';
```

注意：如果 10g 中用了 SGA 内存自动管理。以上语句可能无法查出。

当需要时，少量的空闲内存 chunk 会被释放到 shared pool 中。除非所有这些少量空闲内存倍耗尽了，否则不会报 4031 错误。如果实例在负载高峰运行了一段时期之后还有大量的少量空闲内存，这就说明 shared pool 太大了。

而未 pin 住的、可重建 (unpinned recreatable) 的 chunk 被维护在两个分别用于周期性 chunk 和短期 chunk 的 LRU 链表中。这两个 LRU 链表的长度可以通过表 X\$KGHLU 查到，同时还能查到被 flush 的 chunk 数、由于 pin 和 unpin 而加到和从 LRU 链表中移出的 chunk 数。X\$KGHLU 还能显示 LRU 链表没有被成功 flush 的次数，以及最近一次这样的请求失败的请求大小。

```
SQL> column kghlurcr heading "RECURRENT|CHUNKS"
SQL> column kghlutrn heading "TRANSIENT|CHUNKS"
SQL> column kghlufsh heading "FLUSHED|CHUNKS"
SQL> column kghluops heading "PINS AND|RELEASES"
SQL> column kghlunfu heading "ORA-4031|ERRORS"
SQL> column kghlunfs heading "LAST ERROR|SIZE"
SQL> select
  2 kghlurcr "RECURRENT_CHUNKS",
  3 kghlutrn "TRANSIENT_CHUNKS",
  4 kghlufsh "FLUSHED_CHUNKS",
  5 kghluops "PINS AND_RELEASES",
  6 kghlunfu "ORA-4031_ERRORS",
  7 kghlunfs "LAST_ERROR_SIZE"
  8 from
  9 sys.x$kgghlu
 10 where
 11 inst_id = userenv('Instance');
```

RECURRENT_CHUNKS	TRANSIENT_CHUNKS	FLUSHED_CHUNKS	PINS AND_RELEASES	ORA-4031_ERRORS	LAST_ERROR_SIZE
327	368	0	7965	0	0
865	963	2960	102138	0	0
1473	5657	96	20546	1	540
0	0	0	0	0	0

如果短期链表的长度大于周期链表的长度 3 倍以上，说明 Shared Pool 太大，如果 chunk flush 对 LRU 操作的比例大于 1/20，则说明 Shared Pool 可能太小。

### 2.3. Oracle 在 UNIX 下的内存管理

在 UNIX 下，Oracle 是以多进程的方式运行的。除了几个后台进程外，Oracle 为每个连接起一个进程（进程名称中，LOCAL = NO）。而 UNIX 下的内存分为两类，一种叫共享内存，即可以被多个进程共享；还有一种就是私有进程，只能被所分配的进程访问。更加 Oracle 内存区的不同特性，SGA 内存可以被所有会话进程共享，因此是从共享内存中分配的；而 PGA 是进程私有的，因而是从私有内存中分配的。

### 2.3.1. 共享内存和信号量

在 Unix 下, Oracle 的 SGA 是保存在共享内存中的 (因为共享内存是可以被多个进程共享的, 而 SGA 正是需要能被所有 Oracle 进程所共享)。因此, 系统中必须要有足够的共享内存用于分配 SGA。而信号量则是在共享内存被多个进程共享时, 防止发生内存冲突的一种锁机制。

UNIX 下, 对于内存的管理配置, 是由许多内核参数控制, 在安装使用 Oracle 时, 这些参数一定要配置正确, 否则可能导致严重的性能问题, 甚至 Oracle 无法安装、启动。涉及共享内存段和信号量的参数:

参数名称	建议大小 (各个平台的 Oracle 建议值可以去 metalink 上找)	参数描述
SHMMAX	可以得到的物理内存 (0.5*物理内存)	定义了单个共享内存段能分配的最大数。 SHMMAX 必须足够大, 以在一个共享内存段中能足够分配 Oracle 的 SGA 空间。这个参数设置过低会导致创建多个共享内存段, 这会导致 Oracle 性能降低。
SHMMIN		定义了单个共享内存段能分配的最小值。
SHMNI	512	定义了整个系统共享内存段的最大数。
NPROC	4096	系统的进程总数
SHMSEG	32	定义了一个进程能获取的共享内存段的最大数。
SEMMNS	$(NPROC * 2) * 2$	设置系统中的信号数。 SEMMNS 的默认值是 128
SEMMNI	$(NPROC * 2)$	定义了系统中信号集的最大数
SEMMSL	与 Oracle 中参数 processes 大小相同	一个信号集中的信号最大数
SEMMAP	$((NPROC * 2) + 2)$	定义了信号映射入口最大数
SEMMNU	$(NPROC - 4)$	定义了信号回滚结构数
SEMVMX	32768	定义了信号的最大值

**信号量(Semaphore):** 对于每个相关的 process 都给予一个信号来表示其目前的状态。主要的目地在于确保 process 间能同步, 避免 process 存取 shared data 时产生碰撞( collisions)的情况。可以把信号量视为操作系统级的用于管理共享内存的钥匙, 每个进程需要有一把, 一个信号量集就是一组钥匙, 这组钥匙可以打开共同一个共享内存段上的锁。当一个进程需要从共享内存段中获取共享数据时, 使用它自己的钥匙打开锁, 进入后, 再反馈以防止其他进程进来。使用完毕后, 将锁释放, 这样其他进程就可以存取共享数据了。

**共享内存 (Shared Memory)** 是指同一块记内存段被一个以上的进程所共享。这是我们所知速度最快的进程间通讯方式。使用共享内存存在使用多 CPU 的机器上，会使机器发挥较佳的效能。

可以通过以下命令检查你当前的共享内存和信号量的设置：

```
$ sysdef | more
```

当 Oracle 异常中止，如果怀疑共享内存没有被释放，可以用以下命令查看：

```
$ ipcs -mop
IPC status from /dev/kmem as of Thu Jul 6 14:41:43 2006
T      ID      KEY      MODE      OWNER      GROUP NATTCH  CPID  LPID
Shared Memory:
m      0 0x411c29d6 --rw-rw-rw-   root      root        0    899   899
m      1 0x4e0c0002 --rw-rw-rw-   root      root        2    899   901
m      2 0x4120007a --rw-rw-rw-   root      root        2    899   901
m 458755 0x0c6629c9 --rw-r-----   root      sys         2   9113 17065
m      4 0x06347849 --rw-rw-rw-   root      root        1   1661  9150
m 65541 0xffffffff --rw-r--r--   root      root        0   1659 1659
m 524294 0x5e100011 --rw-----   root      root        1   1811 1811
m 851975 0x5fe48aa4 --rw-r-----   oracle    oinstall    66   2017 25076
```

然后它 ID 号清除共享内存段：

```
$ ipcrm -m 851975
```

对于信号量，可以用以下命令查看：

```
$ ipcs -sop
IPC status from /dev/kmem as of Thu Jul 6 14:44:16 2006
T      ID      KEY      MODE      OWNER      GROUP
Semaphores:
s      0 0x4f1c0139 --ra-----   root      root
... ..
s     14 0x6c200ad8 --ra-ra-ra-   root      root
s     15 0x6d200ad8 --ra-ra-ra-   root      root
s     16 0x6f200ad8 --ra-ra-ra-   root      root
s     17 0xffffffff --ra-r--r--   root      root
s     18 0x410c05c7 --ra-ra-ra-   root      root
s     19 0x00446f6e --ra-r--r--   root      root
s     20 0x00446f6d --ra-r--r--   root      root
s     21 0x00000001 --ra-ra-ra-   root      root
s 45078 0x67e72b58 --ra-r-----   oracle    oinstall
```

当 Oracle 异常中止，可以根据信号量 ID，用以下命令清除信号量：

```
$ ipcrm -s 45078
```

一个共享内存段可以分别由以下两个属性来定位：

- Key：一个 32 位的整数
- 共享内存 ID：系统分配的一个内存

```
$ ipcs -mop
IPC status from /dev/kmem as of Thu Jul 6 10:32:12 2006
T      ID      KEY      MODE      OWNER      GROUP NATTCH  CPID  LPID
Shared Memory:
m      0 0x411c29d6 --rw-rw-rw-   root      root        0    899   899
m      1 0x4e0c0002 --rw-rw-rw-   root      root        2    899   901
m      2 0x4120007a --rw-rw-rw-   root      root        2    899   901
m 458755 0x0c6629c9 --rw-r-----   root      sys         2   9113 17065
m      4 0x06347849 --rw-rw-rw-   root      root        1   1661  9150
m 65541 0xffffffff --rw-r--r--   root      root        0   1659 1659
m 524294 0x5e100011 --rw-----   root      root        1   1811 1811
```

### 2.3.2. 私有内存

对于 PGA，由于是分配的私有内存，不存在争用问题，因而 OS 也没有相应的信号量来控制（同理，PGA 中也没有 latch）。在 10g 之前，PGA 的私有内存是通过函数 `malloc()` 和 `sbrk()` 进行分配扩展的，10g 之后，私有内存是通过 `mmap()` 进行初始化的。

另外，还有一点要注意的是，和 SGA 不同，PGA 内存的大小不是固定的，是可以扩展的（前者的大小是固定的，无法增长的）。因而进程的私有内存是会增长的。因此，一个规划好的系统发生内存不足的情况通常是由于进程的私有内存或进程数量突然增长造成的（PGA\_AGGREGATE\_TARGET 参数能尽量控制但不保证 PGA 内存总量被控制在一定数值范围内）。

隐含参数 `_pga_large_extent_size` (1048576) 和 `_uga_cga_large_extent_size` (262144) 就控制了当初初始化时，PGA、UGA 和 CGA 扩张段的最大大小。

### 2.3.3. SWAP 的保留区

swap（交换）区，是 UNIX 中用来分配虚拟内存的一块特殊的磁盘分区。UNIX 启动每一个进程，都需要在 swap 区预留一块和内存一样大小的区域，以防内存不够时作数据交换。当预留的 swap 区用完时，系统就不能再启动新的进程。比如，系统物理内存是 4G，而设置的交换区只有 1G，那么可以计算得出大概 3G 的内存会浪费（Buffer Cache 除外，可能有 2G 浪费）。

在 HP-UX 中，参数 `swapmen_on` 可以让系统创建一个 *pseudo-swap*（伪交换区），大小为系统物理内存的 3/4，但是这个伪交换区并不占用任何内存和硬盘资源。只是说，让系统认为，交换区的大小是  $1+4*3/4=4G$ ，而不是 1G，就是说可以启动更多的进程，避免内存的浪费。

一般系统物理内存不大的时候，设置交换区是物理内存的 2-4 倍，`swapmen_on` 设置为 1 或 0 都没什么影响，但是当系统内存很大如 8G 时，因为 swap 一般不设为 16G-32G，这时开启 `swapmen_on` 就很必要了。

hp 建议，即使设置了 `swapmen_on`，也将你的 swap 为物理内存的 1-1.5 倍。

swap 大小设置不当，也同样会造成系统的性能问题。因为，swap 中首先会为各个进程留出一个保留区，这部分区去掉后，swap 的可用大小就比较小了（这就是为什么用 `swapinfo` 可能看到 Total PCT USED 为 100% 而 dev PCT USED 为 0%）。当 swap 可用区不足，而由内存需要被 page out 到 swap 区中，就需要先将 swap 区中一些页被 page in 到物理内存中去，因而导致发生交换，产生性能问题。

swap 的使用情况可以通过 `swapinfo` 查看：

```
> swapinfo -mt
      Mb      Mb      Mb  PCT  START/      Mb
TYPE   AVAIL   USED   FREE  USED  LIMIT RESERVE  PRI  NAME
dev     4096      0   4096   0%    0      -    1
/dev/vg00/lvol12
dev     8000      0   8000   0%    0      -    1
/dev/vg00/swap2
reserve -   12026 -12026
memory  20468  13387   7081  65%
total   32564  25413   7151  78%    -      0    -
```

## 2.4. Oracle 在 windows 下的内存管理

### 2.4.1. Windows 内存系统概述

Windows NT 使用一个以页为基础的虚拟内存系统，该系统使用 32 位线性地址。在内部，系统管理被称为页的 4096 字节段中的所有内存。每页的物理内存都被备份。对于临时的内存页使用页文件 (pagefile)，而对于只读的内存页，则使用磁盘文件。在同一时刻，最多可以有 16 个不同的页文件。代码、资源和其它只读数据都是通过它们创建的文件直接备份。

Windows NT 为系统中的每一个应用程序（进程）提供一个独立的、2 GB 的用户地址空间。对于应用程序来说，好象是有 2 GB 的可用内存，而不用考虑实际可用的物理内存的量。如果某个应用程序要求的内存比可用的内存更多时，Windows NT 是这样满足这种要求的，它从这个和/或其他进程把非关键内存分页 (paging) 到一个页文件，并且释放这些物理内存页。结果，在 Windows NT 中，全局堆不再存在。相反，每一个进程都有其自己的 32 位地址空间，在其中，该进程的所有内存被分配，包括代码、资源、数据、DLL（动态链接库），和动态内存。实际上，系统仍然要受到可用的硬件资源的限制，但是实现了与系统中应用程序无关的、对于可用资源的管理。

Windows NT 在内存和地址空间之间作出了区分。每个进程分配到 2 GB 的用户地址空间，而不管对于该进程的实际可用物理内存有多少。而且，所有进程都使用相同范围的线性 32 位地址，范围从 0000000016-7FFFFFFF16，而不考虑可用内存的地址。Windows NT 负责在适当的时间把内存页映射 (paging) 到磁盘以及从磁盘页映射回内存，使得每个进程都确保能够寻址到它所需要的内存。尽管有可能出现两个进程试图同时访问同一虚拟地址上的内存，但是，实际上 Windows NT 虚拟内存管理程序是在不同的物理位置描述这两个内存的位置。而且这两个地址都不见得与原始的虚拟地址一致。这就是虚拟内存。

Win32 环境下，32 位的地址空间转化为 4GB 的虚拟内存。默认情况下，将一半（2GB）分配给用户进程（因而一个进程的最大可用虚拟内存为 2G，oracle 进程同样受此限制），另一半（2GB）分配给操作系统。

因为虚拟内存的存在，一个应用程序能够管理它自己的地址空间，而不必考虑在系统中对于其它进程的影响。在 Windows NT 中的内存管理程序负责查看在任何给定的时间里，所有的应用程序是否有足够的物理内存进行有效的操作。Windows NT 操作系统下的应用程序不必考虑和其它应用程序共享系统内存这个问题。并且，即使在应用程序自己的地址空间内，它们仍能够与其它的应用程序共享内存。

区分内存和地址空间的一个好处是，为应用程序提供了将非常大的文件加载到内存的能力。不必将一个大的文件读进内存中，Windows NT 为应用程序保留该文件所需的地址范围提供了支持。然后，在需要的时候，该文件部分就可以被浏览了（物理性地读进内存）。通过虚拟内存的支持，对于大段的动态内存的分配同样可以做到这一点。

在任意给定的时间，进程中每个地址都可以被当作是空闲的 (free)、保留的 (reserved) 或已提交的 (committed)。进程开始时，所有地址的都是空闲的，意味着它们都是自由空间并且可以被提交到内存，或者为将来使用而保留起来。在任何空闲的地址能够被使用前，它必须首先被分配为保留的或已提交的。试图访问一个保留的或已提交的地址都将产生一个访问冲突异常 (access violation exception)。

一个进程中的所有 2 GB 的地址要么为了使用而是空闲的、要么为了将来的使用而是保留的、要么已提交到特定的内存（在使用的）。

一旦地址被以保留的或者已提交的形式分配，VirtualFree 是唯一可以释放它们的方法。那就是，将它们返回到自由的地址。VirtualFree 还可以用来对已提交的页解除提交，同时，返回这些地址到保留状态。当解除地址的提交时，所有与该地址相关的物理内存和页文件空间都被释放。

在 Windows NT 中的进程有一个被称为**工作组**（working set）的最小页，是为了进程能够顺利地运行，在运行时在内存中必须被提供。Windows NT 在启动时为一个进程分配了默认数量的页数，并且逐渐地调整该数，使得系统中所有激活的进程的性能达到一种平衡的最优。当一个进程正在运行时（实际上是，是一个进程的线程正在运行时），Windows NT “尽量”确保该进程的工作组页总是驻留在物理内存中。工作集即在物理内存中保持的虚拟页面的子集，分进程工作集和系统工作集。

### 2.4.2. Windows 下 Oracle 的内存配置

在 windows 下，Oracle 实例作为一个单独的进程运行，这个进程是一个标准的 Win32 应用程序，它能够申请最大为 2G 的虚拟内存地址空间，所有用户连接后台线程的分配内存（包括像 buffer cache 那些全局内存）必须小于 2G（64 位平台中无此限制，32 位平台中可以通过设置参数 `use_indirect_data_buffers` 来突破这一限制，不详述）。

Oracle 可以运行于 windows NT 下的任何版本，但是 Oracle 一般不推荐将 Oracle 数据库运行在 PDC（主域控服务器）或 BDC（备域控服务器）下。这是因为域控服务器会需要大量的文件缓存（这会消耗大量内存，影响到 Oracle）和网络资源。

而文件缓存带来的另外一个问题就是，这个机器到底是作为一个专用的数据库服务器还是一个混合服务器。因为 Oracle 数据库不会用到文件缓存（log buffer 就相当于 Oracle 自己的文件缓存），它通过直接写磁盘来避免文件缓冲。

在专用数据库服务器系统上，用户一定要确保没有使用到页文件（pagefile 即虚拟内存文件）。否则，可以通过修改 Oracle 参数或者增大物理内存来避免。如果大量额页被持续的移入、移出到虚拟内存中，会严重影响到性能。

如果是专用服务器系统，有以下建议：

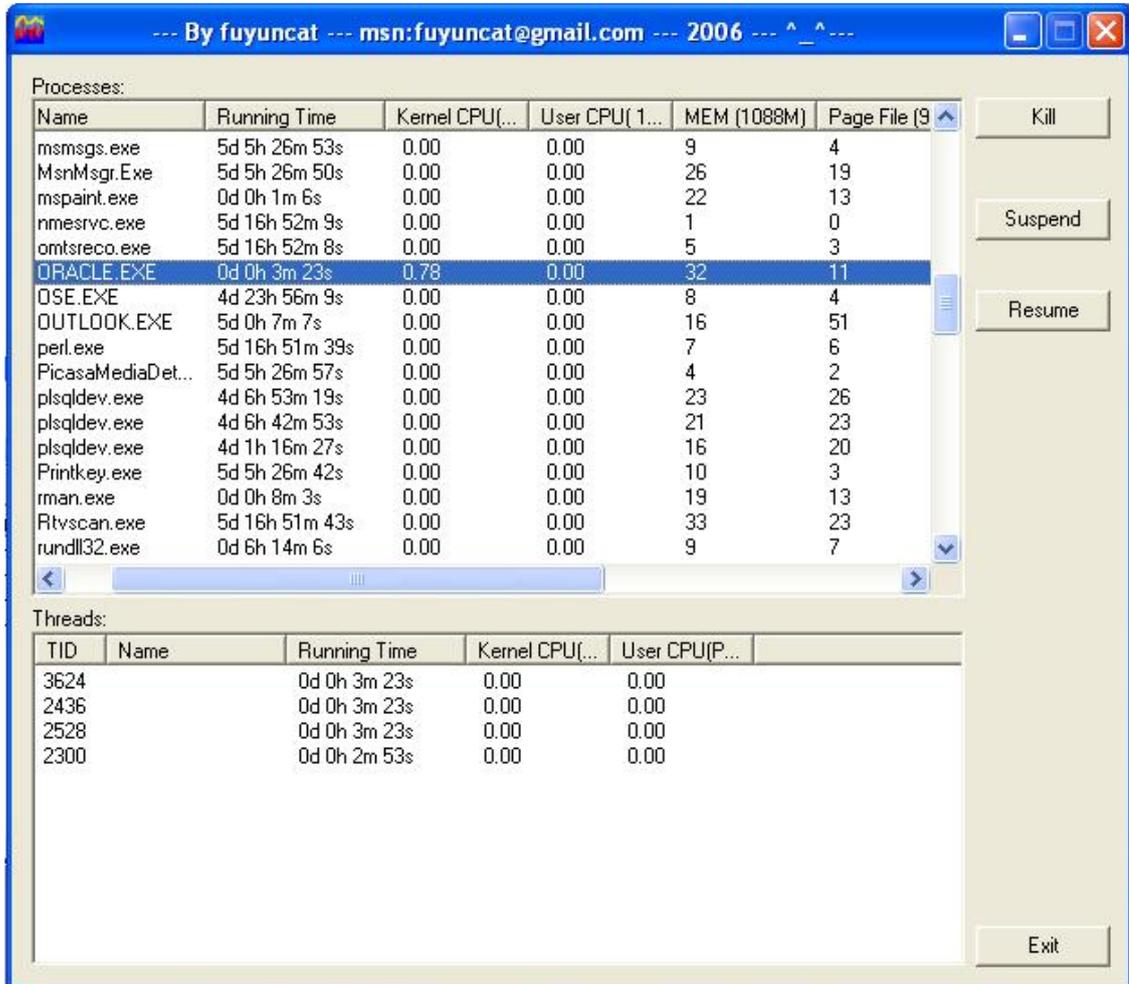
- 如果分配给 Oracle 的总内存能保证不会超过物理内存，则虚拟内存页可以被设置位物理内存的 50%，并且可以增长到物理内存的 100% 大小（在 my computer => properties => Advanced => Performance => Settings => Advanced => Virtual Memory => Change 中设置，设置 Initial size 为物理内存的 50%，Maximum Size 和物理内存大小相同）；
- 对于主要是运行纯 Oracle 数据库的系统（但不是专用），一般推荐虚拟内存页大小在 1 倍到 1.5 倍于物理内存大小之间；
- 对于物理内存大于 2G 的机器，要求虚拟内存页最少为 2G。

一个机器上能被用于分配的总内存等于物理内存加上扩展前的虚拟内存页大小。你一定要避免设置参数如 `buffer_cache_size` 或其他相关参数导致 Oracle 要求分配内存大于物理内存。尽管 Oracle 的分配内存大小是限制在总内存（物理内存+最小虚拟内存）之内，但是，对虚拟内存页的访问是非常慢的，会直接影响到系统性能，因此 Oracle 分配内存要小于物理内存大小以避免发生内存交换。

如果系统是混合应用，除了 Oracle 数据库外还运行了其他程序，这时就需要考虑设置虚拟内存页大于物理内存了。那些当前不活动的进程可以减少它们的工作区（working set 即物理内存）以使活动进程能增加工作区。如果 Oracle 数据库运行在这样的机器上，则推荐虚拟内存最少 1.5 到 2 倍的物理内存大小，特别是在内存大于 2G 时。

在 Oracle 8.1.x 之前，启动 Oracle 服务时不会启动 Oracle 实例。此时（即只启动了 Oracle 服务，没有启动实例），分配给 Oracle.EXE 的主要内存是给相关 DLL 的，大概

20M（各个版本的 DLL 不同，因此占用内存情况也不同）。9i 之后，Oracle 实例会随着服务启动，不过我们可以在服务启动后再关闭实例，这时就可以观察出 Oracle 服务所占用的内存大小了：



这是 windows 下一个 Oracle 10g 的实例关闭后内存占用情况。我们看到此时 Oracle 服务占用的内存是 32M。

Windows 下，可以使用以下语句来计算 Oracle 占用的虚拟内存大小：

```
select sum(bytes)/1024/1024 + 22*DLL 占用内存*/ Mb
from (select bytes from v$sgastat - SGA 内存
union
select value bytes from - 会话内存
v$sesstat s,
v$statname n
where
n.STATISTIC# = s.STATISTIC# and
n.name = 'session pga memory'
union
select 1024*1024*count(*) bytes - 线程堆栈
```

```
from v$process  
);
```

在实例启动时，所有全局内存页都被保留和提交（所有共享全局区、Buffer Cache 和 Redo Buffer）——可以通过 [TopShow](#) 观察到实例启动后，所需要的 Page File 都已经被分配。但只有一小部分内存页（如果没有设置 PRE\_PAGE\_SGA 的话；这小部分内存以 granule 为单位，固定 SGA 【包括 redo buffer】一个、Buffer Cache 一个、Shared Pool 一个）被触及（touch）而已经分配到工作组(working set)中，而其他更多的页需要使用时才分配到工作组中。

通过设置注册表可以设置 Oracle 进程的最小工作组大小和最大工作组大小：

- ORA\_WORKINGSETMIN 或 ORA\_%SID%\_WORKINGSETMIN: Oracle.EXE 进程的最小工作组大小（M 为单位）
- ORA\_WORKINGSETMAX 或 ORA\_%SID%\_WORKINGSETMAX: Oracle.EXE 进程的最大工作组大小（M 为单位）

这些注册项需要加在 HKEY\_LOCAL\_MACHINE -> SOFTWARE -> ORACLE 或者 HKEY\_LOCAL\_MACHINE -> SOFTWARE -> ORACLE -> HOMEn 下。

在混合服务器下，这种设置能防止 Oracle 进程的内存被其他进程争用。设置这些注册项时，需要考虑 PRE\_PAGE\_SGA 的设置。如前所述，PRE\_PAGE\_SGA 使 Oracle 实例启动时“触及”所有的 SGA 内存页，使它们都置入工作组中，但同时会增长实例启动时间。

ORA\_WORKINGSETMIN 是一个非常有用的参数，它能防止 Oracle 进程的工作组被缩小到这一限制值之下：

- 如果设置了 PRE\_PAGE\_SGA，实例启动后，工作组就大于这个限制值。在实例关闭之前，就不会低于这个值；
- 如果没有 PRE\_PAGE\_SGA，当实例的工作组一旦达到这个值后，就不会再低于这个值。

另外，在 10g 之前，存在一个 Bug(642267)，导致在 windows 系统中设置了 LOCK\_SGA 后，实例启动报 ORA-27102 错误。可以通过设置 ORA\_WORKINGSETMIN 最小为 2M 来解决这个问题。但是，windows 中，LOCK\_SGA 只影响 Oracle 进程中的 SGA 部分，而分配给用户会话的内存不会受影响，这部分内存还是可能会产生内存交换。

### 2.4.3. SGA 的分配

当实例启动时，oracle 在虚拟内存地址空间中创建一段连续的内存区，这个内存区的大小与 SGA 所有区相关参数有关。通过调用 Win32 API 接口“VirtualAlloc”，在接口函数的参数中指定 MEM\_RESERVE | MEM\_COMMIT 内存分配标识和 PAGE\_READWRITE 保护标识，这部分内存始终会被保留和提交。这就保证所有线程都能访问这块内存（SGA 是被所有线程共享的），并且这块内存区始终有物理存储（内存或磁盘）所支持。

VirtualAlloc 函数不会触及这块区域内的内存页，这就是说分配给 SGA 组件（如 buffer cache）的内存存在没有触及之前是不会到工作组中去的。

#### ***VirtualAlloc***

*VirtualAlloc* 函数保留或提交调用此函数的进程的虚拟内存地址空间中的一段内存页。如果没有指定 MEM\_RESET，被这个函数分配的内存自动初始化为 0，

Buffer Cache 通常被创建为一个单一连续内存区。但这并不是必须的，特别是当使用了非常大的 Buffer Cache 时。因为 dll 内存和线程分配的内存会导致虚拟地址空间产生碎片。

当一个进程创建后，windows NT 会在进程的地址空间中创建一个堆（heap），这个堆被称为进程的默认堆。许多 Win32 API 调用接口和 C 运行调用接口（如 malloc、localalloc）都会使用这个默认堆。当需要时，进程能在虚拟内存地址空间中创建另外的命名堆。默认堆创建为 1M 大小（被保留和提交的）的内存区，当执行分配或释放这个堆的操作时，堆管理器提交或撤销这个区。而访问这个区是通过临界区来串行访问的，所以多个线程不能同时访问这个区。

当进程创建了一个线程后，windows NT 会为线程堆栈（每个现场都有自己的堆栈）保留一块地址空间区域，并提交一部分这些保留区。当一个进程连接到标准区时，系统为堆栈保留一个 1M 大小的虚拟地址空间并提交这个区顶部的两个页。当一个线程分配了一个静态或者全局变量时，多个线程可以同时访问这个变量，因此存在变量内容被破坏的潜在可能。本地和自动变量被创建在线程的堆栈中，因而变量被破坏的可能性很小。堆栈的分配是从上至下的。例如，一个地址空间从 0x08000000 到 0x080FF000 的堆栈的分配是从 0x080FF000 到 0x08001000 来提交内存页，如果访问在 0x08001000 的页就会导致堆栈溢出异常。堆栈不能增长，任何试图访问堆栈以外的地址的操作都可能会导致进程被中止的致命错误。

#### 2.4.4. 会话内存的分配

当监听创建了一个用户会话（线程和堆栈）时，Oracle 服务进程就通过调用 Win32 API 函数创建用户连接所必须的内存结构，并且指定 MEM\_RESERVE | MEM\_COMMIT 标识，以保持和提交给线程私有的地址空间区域。

当调用了 VirtualAlloc 来在指定地址保留内存，它会返回一个虚拟地址空间到下一个 64K 大块（chunk，windows 内存分配最小单位）的地址。Oracle 调用 VirtualAlloc 时不指定地址，只传一个 NULL 在相应参数中，这会返回到下一个 64K 大块的地址。因此用户会话在分配 PGA、UGA 和 CGA 时同时也遵循 64K 的最小粒度，来提交倍数于这个粒度值的内存页。许多用户会话经常分配许多小于 64K 的内存，这就导致地址空间出现碎片，因为许多 64K 区只有两个页被提交。

一旦地址空间被用户会话占满了后，如果要再创建一个新会话，就会存在无法分配到地址空间的危险。将可能报以下错误：

```
ORA-12500 / TNS-12500
```

```
TNS:listener failed to start a dedicated server process
```

也可能报这些错误：

- ORA-12540 / TNS-12540 TNS:internal limit restriction exceeded
- NT-8 Not enough storage is available to process this command
- skgspawn failed:category = ....
- ORA-27142 could not create new process
- ORA-27143 OS system call failure
- ORA-4030 out of process memory when trying to allocate ....

因为地址空间碎片问题和 DLL 被载入了 oracle 服务进程的地址空间，这些错误很可能发生再当 Oracle 进程占用大概 1.6G~1.7G（可以通过任务管理器或 [TopShow](#) 查看）时。

### 2.4.4.1. 会话内存大小设置

我们前面说了，一个进程的全部内存大小被限制在 2G 以内。因此，对于一个有许多用户同时访问的系统，要考虑这些会话总内存小于 2G - SGA 的大小。

下列参数会影响每个会话的内存大小（这些参数前面都有介绍）：

- o bitmap\_merge\_area\_size
- o create\_bitmap\_area\_size
- o hash\_area\_size
- o open\_cursors
- o sort\_area\_size (sort\_area\_retained\_size)

在没有设置 PGA\_AGGREGATE\_TARGET（这个参数能尽量但不一定使所有会话 PGA 之和在指定范围内）参数时，需要调整这些参数，以使所有会话占用内存与 SGA 之和小于 2G。过多的使用 PL/SQL 结构体（如 PL/SQL TABLE、ARRAY）也会导致会话内存增大。

### 2.4.4.2. ORASTACK 修改线程堆栈大小

Oracle 提供了 ORASTACK 工具让用户内修改 Oracle 执行程序创建会话、线程时的默认堆栈大小。当 ORASTACK 应用于一个可执行程序时，它会修改程序头部的、定义使用创建线程 API 函数所指定默认堆栈大小的二进制区，以修改默认堆栈的大小。没有必要去修改线程提交页数的默认值，因为它们是从堆栈中请求到的。当用户非常多时，通过减少每个创建在 Oracle 中会话堆栈大小，可以节省大量内存。比如，一个 1000 用户的系统，将堆栈从 1M 降为 500K 后，能节省出  $1000 * 500K = 500M$  的地址空间。

在需要使用 ORASTACK 来降低现场堆栈大小时，你需要测试你的系统以保证新的堆栈大小能确保系统正常运行。如果堆栈大小被缩小到 Oracle 服务端所必须的堆栈大小以下，就会产生堆栈溢出错误，用户进程就失败（通常报 ORA-3113 错误），并且在 alert log 中不会有报错而且页不产生 trace 文件。Oracle 一般不推荐将堆栈该到 500K 以下（尽管不少系统在 300K 时也能正常运行）。

ORASTACK 必须修改所有能在 oracle 中创建线程的进程，使用语法如下：

```
orastack executable_name new_stack_size_in_bytes
```

下面的例子将堆栈改为 500K：

```
orastack oracle.exe 500000
orastack tnslsnr.exe 500000
orastack svrmgrl.exe 500000
orastack sqlplus.exe 500000
```

在使用 ORASTACK 之前必须保证没有任何 oracle 进程正在运行（可以通过任务管理器或者 [TopShow](#)）。

此外，如果有程序在本地连接（没有通过 SQL\*NET）了 Oracle，也要先停止（如在本地运行 sqlplus 连接了实例）。

### 2.4.4.3. 会话内存如何释放、线程如何结束

当会话成功结束后，它会按用 Win32 API 函数 VirtualFree 来释放它的内存，调用此函数时，需要指定 MEM\_DECOMMIT | MEM\_RELEASE 标识。当所有内存被释放后，堆栈也被释放，将 Oracle 进程中指向完成的会话的地址空间空闲出来。

如果一个用户会话被异常结束，它将不会释放它所分配的内存，这些内存页会被继续保留在 Oracle 进程的地址空间中，知道进程结束。会话的异常结束可能由以下原因导致的：

- o Shutdown abort.

- Alter session kill session.
- orakill 杀掉的会话.
- Oracle 管理助手 for Windows: kill session.
- 其他杀线程的工具 ([TopShow](#) 工具提供了杀线程的功能)

Oracle 建议尽量少使用以上方式 (命令), 特别是 shutdown abort (这种停止实例的方法所带来的问题还不止这个)。当调用 shutdown abort 时, Oracle 会调用 Win32 API 函数 TerminateThread 来中止每个用户会话, 这个命令将直接杀掉线程而不释放它们的内存。如果系统已经接近 2G 地址空间的限制, Oracle 实例再次启动要分配内存时就会产生问题。唯一释放 Oracle 全部内存的方法是停止和启动 Oracle 服务 (服务 OracleService<SID>)。

如果 windows NT 下的系统需要能被许多用户访问, 可以通过以下措施来优化内存的使用:

- 降低相关的 PGA、UGA 内存参数 (如 SORT\_AREA\_SIZE);
- 降低 SGA 的参数;
- 使数据库作业队列数 (参数 job\_queue\_processes 控制) 和并行查询 slave (参数 parallel\_max\_servers 控制) 最小, 因为它们也会导致 Oracle 进程创建线程;
- 使用 ORASTACK 降低会话、线程堆栈大小到 500K;
- 考虑使用 MTS 模式;
- 考虑将 Windows NT 升级到 Windows NT 企业版;
- 考虑升级硬件以支持 Intel ESMA (Extended. Server Memory Architecture, 扩展服务内存架构)

### 3. 内存错误处理

Oracle 中最常见的内存错误就是 4030 和 4031 错误。这两个错误分别是在分配 PGA 和 SGA 时, 没有足够内存分配导致的。经过我们以上对 Oracle 内存的了解以及对内存管理机制的浅析, 可以总结在这两种错误发生时, 我们该如何分析和处理。

#### 3.1. 分析、定位 ORA-4030

4030 错误是由于 oracle 进程在内存扩展时, 无法从 OS 获取到所需的内存而产生的报错。在专有服务模式下, Oracle 进程的内存包括堆栈、PGA、UGA (从 PGA 中分配) 和有些进程信息; 而 MTS 下, UGA 是从 SGA 中分配, 不包括在进程的内存范围内。

##### 3.1.1. 4030 错误产生的原因

PGA 的大小不是固定的, 是可以扩展的。PGA 通过系统调用扩展堆数据段时, 操作系统分配新的虚拟内存给进程作为 PGA 扩展段。这些扩展段一般是几个 KB。只要需要, oracle 会分配几千个扩展段。然而, 操作系统限制了一个进程的堆数据段的增长。在 UNIX 中, 这个限制一般受到 OS 内核参数 MAXDSIZ 限制, 这是限制单个进程的。还有一个堆所有进程的虚拟内存的总的大小的限制。这个限制和 swap 交换空间 (虚拟内存) 大小有关。如果在扩展 PGA 内存时达到这些限制, 就会抛 4030 错误。

### 3.1.2. 4030 错误分析

既然知道了 4030 错误产生的可能原因，我们在分析 4030 错误时，就可以从这几个方面分别收集信息进行分析，并结合 Oracle 进程内存的使用情况来解决问题。

#### 3.1.2.1. 操作系统是否由足够的内存

在不同的操作系统下，我们可以使用相应的工具来收集系统的内存使用情况，以判断内存是否足够：

- OpenVMS systems

可以使用 show memory 查看物理内存和虚拟内存页的使用情况

```
Physical Memory Usage (pages):  Total      Free      In Use    Modified
Main Memory
(256.00Mb)                      32768     24849     7500     419
.....

Paging File Usage (blocks):      Free    Reservable  Total
DISK$BOBBIEAXPSYS:[SYS0.SYSEXE]SWAPFILE.SYS  30720   30720       39936
DISK$BOBBIEAXPSYS:[SYS0.SYSEXE]PAGEFILE.SYS  226160 201088     249984
DISK$BOBBIE_USER3:[SYS0.PAGEFILE]PAGEFILE.SYS 462224 405296     499968
```

一般情况下，空闲 pagefile 的之和不能小于它的总数之和的一半。而且 SWAPFILE 必须是始终没有被使用的，它的空闲页必须和总页数相同。

- Windows

Windows 下可以通过任务管理器的性能页来查看内存的使用情况，也可以使用 [TopShow](#) 来观察系统进程的内存使用情况

- UNIX

不同厂商的 UNIX 下，各种工具的使用和统计结果可能有所不同。常用的对内存的查看工具主要有：

- TOP —— 查看物理内存和交换空间
- vmstat —— 查看物理内存和交换空间状况，以及观察是否有 page out/page in
- swapon -s —— 查看交换空间情况
- swapinfo -mt —— 查看交换空间使用情况

下面是 swapinfo 的一个输出：

```
> swapinfo -mt
      Mb      Mb      Mb    PCT  START/      Mb
TYPE   AVAIL  USED   FREE  USED  LIMIT RESERVE  PRI  NAME
dev     4096    0   4096   0%    0      -    1
/dev/vg00/lvol2
dev     8000    0   8000   0%    0      -    1
/dev/vg00/swap2
reserve - 12026 -12026
memory 20468 13387  7081  65%
total  32564 25413  7151  78%    -      0    -
```

此外，在一些操作系统中，还可以通过 Oracle 自己提供的工具 maxmem 来检查一个进程能够分配的最大堆数据段的大小。

```
> maxmem
Memory starts at: 6917529027641212928 (6000000000020000)
Memory ends at:   6917529031936049152 (6000000100000000)
Memory available: 4294836224 (fffe0000)
```

### 3.1.2.2. 是否受到系统限制

在操作系统，往往会有对单个进程或者所有进程能够分配的内存大小做了限制。当 Oracle 分配进程内存时，如果达到这些限制，也会导致 4030 错误。在不同操作系统中，可以用不同方式检查系统是否有做限制。

- OpenVMS systems:

show process/id=<process id>/quota 可以显示一个进程的可用配额是多少。

- Windows

如前所述，在 window 32 位系统中，进程的可用内存限制为 2G（可以通过其他方式突破此限制）。而 windows 下，oracle 是以一个单独进程方式运行的，它的内存包括了堆栈、SGA、PGA。我们可以通过任务管理器或 [TopShow](#) 来检查 Oracle 进程是否达到此限制。

- UNIX

可以使用命令 ulimit 来查看 unix 下的限制：

```
> ulimit -a
time(seconds)          unlimited
file(blocks)           unlimited
data(kbytes)           1048576
stack(kbytes)          131072
memory(kbytes)         unlimited
coredump(blocks)      4194303
```

### 3.1.2.3. 哪个 Oracle 进程请求了过多的内存

有些进程会做某些操作时会需要分配大量内存，如使用了 PL/SQL TABLE 或者做排序时。如果这样的进程在系统中运行一段时间后，就可能导致 4030 错误的产生。我们可以用以下语句来查看各个进程的内存使用情况：

```
select
  sid,name,value
from
  v$statname n,v$sesstat s
where
  n.STATISTIC# = s.STATISTIC# and
  name like '%ga %'
order by 3 asc;
```

同时，我们还可以从操作系统的角度来确认这些大量消耗内存的进程。

- OpenVMS systems:

show process/continous 可以查看各个进程的物理和虚拟内存的使用情况。

- Windows

在 windows 中，由于 Oracle 是以一个单独进程运行的，而由线程来服务于会话的。因此无法查看单个会话的内存占用情况。

- UNIX

UNIX 中，可以通过 ps -lef|grep ora 来查看 oracle 进程占用的内存情况。

### 3.1.2.4. 收集进程正在进行的操作

在解决 4030 问题时，有一点很重要，抛出 4030 错误的进程并不一定是导致内存不足的进程。只不过在它请求分配内存时，内存已经不足了。很有可能在此之前就已经有大量消耗内存的进程导致内存不足。你需要找出内存消耗不断增长的进程，观察它锁进行的操作。这条语句可以查出会话进程正在执行的语句：

```
select sql_text
from v$sqlarea a, v$session s
where a.address = s.sql_address and s.sid = <SID>;
```

另外，可以做一个 heapdump，将结果发给 Oracle 进行分析，

```
SQL> oradebug unlimit
SQL> oradebug setorapid <PID> (通过 v$process 查到的 pid, 用 setospid 来设置 OS
中的 PID 【或者 v$process 中的 spid】)
SQL> oradebug dump heapdump 7 (1-PGA; 2-Shared Pool; 4-UGA; 8-CGA; 16-top
CGA; 32-large pool)
SQL> alter session set events '4030 trace name heapdump level 25';
```

### 3.1.3. 解决 4030 错误的建议

如果问题是由于 swap 空间不足造成的，并且由中度或者严重的 page in/page out（可以用 vmstat 查看），你就需要尝试降低系统整体的虚拟内存的使用（如调整 SGA 大小），或者降低单个进程内存的使用（如调整 sort\_area\_size），或者减少进程数量（如限制 processes 参数，使用 MTS）。而如果 page in/page out 很少或者根本没有，就可以考虑增大 swap 空间。某些系统中，可以考虑使用伪交换区（如 hp-ux 中，可以考虑设置 swapmen\_on）。

如果问题和 PLSQL 操作有关，可以，1、检查 PLSQL 中的 TABLE，看看其中的数据是否全都必要，是否可以减少数据放入 TABLE 中；2、优化相关语句（比如通过调整优化器策略，使查询计划走 sort 比较少的访问路径），减少 sort 操作，或者减少 sort\_area\_size（代价就是一部分 sort 操作会放在磁盘上进行，降低性能）。

9i 以后可以考虑设置 PGA 内存自动管理。即设置 PGA\_AGGREGATE\_TARGET 在一定数值范围内，WORKAREA\_SIZE\_POLICY 设置为 AUTO。但是注意，9i 在 OpenVMS 系统上、或者在 MTS 模式下不支持 PGA 内存自动关联。

如果是因为进程数过多导致的内存大量消耗，首先可以考虑调整客户端，减少不必要的会话连接，或者采用连接池等方式，以保持系统有稳定的连接数。如果会话非常多，且无法降低的话，可以考虑采用 MTS，以减少 Oracle 进程数。

检查 SGA 中的内存区是否分配过多（如 shared pool、large pool、java pool）等，尝试减少 SGA 的内存大小。

在 windows 下，可以尝试使用 ORASTACK 来减少线程的堆栈大小，以释放更多的内存。考虑增加物理内存。

## 3.2. 分析、定位 ORA-4031

4031 错误是 Oracle 在没有足够的连续空闲空间分配给 Shared Pool 或者 Large Pool 时抛出的错误。

### 3.2.1. 4031 错误产生的原因

前面我们描述 Shared Pool 的空闲空间的请求、分配过程。在受到空闲空间请求时，内存管理模块会先查找空闲列表，看是否有合适的空闲 chunk，如果没有，则尝试从 LRU 链表中寻找可释放的 chunk，最终还未找到合适的空闲 chunk 就会抛出 4031 错误。

在讨论 4031 问题之前，可以先到第一章中找到与 shared pool (shared\_pool\_size、shared\_pool\_reserved\_size、shared\_pool\_reserved\_min\_alloc) 和 large pool (large\_pool\_size) 的参数描述，再了解一下这些参数的作用。这对于理解和分析 4031

错误会很有帮助。此外，还需要再回顾以下 10g 以后的 SGA 内存自动关联部分（相关参数是 SGA\_TARGET），因为使用这一特性，能大大减少 4031 错误产生的几率。

### 3.2.2. 4031 错误分析

通常，大多数的 4031 错误都是和 shared pool 相关的。因此，4031 错误的分析，主要是对 shared pool 的分析。

#### 3.2.2.1. 对 shared pool 的分析

当 4031 错误提示是 shared pool 无足够连续内存可分配时，有可能是由于 shared pool 不足或者 shared pool 中严重的碎片导致的。

- Shared pool 不足分析

视图 V\$SHARED\_POOL\_RESERVED 中可以查询到产生 4031 的一些统计数据、以及 shared pool 中保留区（前面说了，保留区是用来缓存超过一定大小的对象的 shared pool 区）的统计信息。

如果字段 REQUEST\_FAILURES >= 0 并且字段 LAST\_FAILURE\_SIZE < \_SHARED\_POOL\_RESERVED\_MIN\_ALLOC，可以考虑减小 \_SHARED\_POOL\_RESERVED\_MIN\_ALLOC，以使更多的对象能放到保留区中（当然，你还需要观察字段 MAX\_USED\_SPACE 以确保保留区足够大）。如果还没有效果，就需要考虑增加 shared\_pool\_size 了。

- 碎片问题分析

Library cache 和 shared pool 保留区的碎片也会导致 4031 错误的产生。

还是观察上面的视图，如果字段 REQUEST\_FAILURES > 0 并且字段 LAST\_FAILURE\_SIZE > \_SHARED\_POOL\_RESERVED\_MIN\_ALLOC，就可以考虑增加 \_SHARED\_POOL\_RESERVED\_MIN\_ALLOC 大小以减少放入保留区的对象，或者增加 SHARED\_POOL\_RESERVED\_SIZE 和 shared\_pool\_size（因为保留区是从 shared pool 中分配的）的大小。

此外，要注意有一个 bug 导致 REQUEST\_FAILURES 在 9.2.0.7 之前所有版本和 10.1.0.4 之前的 10g 版本中统计的数据是错误的，这时可以观察最后一次 4031 报错信息中提示的无法分配的内存大小。

#### 3.2.2.2. 对 large pool 的分析

Large pool 是在 MTS、或并行查询、或备份恢复中存放某些大对象的。可以通过视图 v\$sgastat 来观察 large pool 的使用情况和空闲情况。

而在 MTS 模式中，sort\_area\_retained\_size 是从 large pool 中分配的。因此也要检查和调整这个参数的大小，并找出产生大量 sort 的会话，调整语句，减少其中的 sort 操作。

MTS 中，UGA 也是从 large pool 中分配的，因此还需要观察 UGA 的使用情况。不过要注意一点的是，如果 UGA 无法从 large pool 获取到足够内存，会尝试从 shared pool 中去分配。

### 3.2.3. 解决 4031 错误

根据 4031 产生的不同原因，采取相应办法解决问题。

### 3.2.3.1. bug 导致的错误

有很多 4031 错误都是由于 oracle bug 引起的。因此，发生 4031 错误后，先检查是否你的系统的 4031 错误是否是由 bug 引起的。下面是已经发现的会引起 4031 错误的 bug。相关信息可以根据 bug 号或 note 号到 metalink 上查找。

BUG	说明	修正版本
Bug 1397603	ORA-4031 由于缓存句柄导致的 SGA 永久内存泄漏	8172, 901
Bug 1640583	ORA-4031 due to leak / 由于查询计划中 AND-EQUAL 访问路径导致缓冲内存链争用，从而发生内存泄漏。	8171, 901
Bug:1318267 (未公布)	如果设置了 TIMED_STATISTICS 可能导致 INSERT AS SELECT 无法被共享。	8171, 8200
Bug:1193003 (未公布)	Oracle 8.1 中，某些游标不共享。	8162, 8170, 901
Bug 2104071	ORA-4031 太多 PIN 导致 shared pool 消耗过大。	8174, 9013, 9201
Note 263791.1	许多与 4031 相关的错误在 9205 补丁集中修正。	9205

### 3.2.3.2. Shared pool 太小

大多数 4031 错误都是由 shared pool 不足导致的。可以从以下几个方面来考虑是否调整 shared pool 大小：

- Library cache 命中率

通过以下语句可以查出系统的 library cache 命中率：

```
SELECT SUM(PINS) "EXECUTIONS",  
       SUM(RELOADS) "CACHE MISSES WHILE EXECUTING",  
       1 - SUM(RELOADS)/SUM(PINS)  
FROM V$LIBRARYCACHE;
```

如果命中率小于 99%，就可以考虑增加 shared pool 以提高 library cache 的命中率。

- 计算 shared pool 的大小

以下语句可以查看 shared pool 的使用情况

```
select sum(bytes) from v$sgastat  
where pool='shared pool'  
and name != 'free memory';
```

专用服务模式下，以下语句查看 cache 在内存中的对象的大小，

```
select sum(sharable_mem) from v$db_object_cache;
```

专用服务模式下，以下语句查看 SQL 占用的内存大小，

```
select sum(sharable_mem) from v$sqlarea;
```

Oracle 需要为保存每个打开的游标分配大概 250 字节的内存，以下语句可以计算这部分内存的占用情况，

```
select sum(250 * users_opening) from v$sqlarea;
```

此外，在我们文章的前面部分有多处提到了如何分析 shared pool 是否过大或过小，这里就不在赘述。

### 3.2.3.3. Shared pool 碎片

每当需要执行一个 SQL 或者 PLSQL 语句时，都需要从 library cache 中分配一块连续的空闲空间来解析语句。Oracle 首先扫描 shared pool 查找空闲内存，如果没有发现大小

正好合适的空闲 chunk，就查找更大的 chunk，如果找到比请求的大小更大的空闲 chunk，则将它分裂，多余部分继续放到空闲列表中。这样就产生了碎片问题。系统经过长时间运行后，就会产生大量小的内存碎片。当请求分配一个较大的内存块时，尽管 shared pool 总空闲空间还很大，但是没有有一个单独连续空闲块能满足需要。这时，就可能产生 4031 错误。

如果检查发现 shared\_pool\_size 足够大，那 4031 错误一般就是由于碎片太多引起的。

如果 4031 是由碎片问题导致的，就需要弄清楚导致碎片的原因，采取措施，减少碎片的产生。以下是可能产生碎片的一些潜在因素：

- 没有使用共享 SQL；
- 过多的没有必要的解析调用（软解析）；
- 没有使用绑定变量。

以下表/视图、语句可以查询 shared pool 中没有共享的 SQL

- 通过 V\$SQLAREA 视图

前面我们介绍过这个视图，它可以查看到每一个 SQL 语句的相关信息。以下语句可以查出没有共享的语句，

```
SELECT substr(sql_text,1,40) "SQL",
count(*) ,
sum(executions) "TotExecs"
FROM v$sqlarea
WHERE executions < 5 --语句执行次数
GROUP BY substr(sql_text,1,40)
HAVING count(*) > 30 --所有未共享的语句的总的执行次数
ORDER BY 2;
```

- X\$KSMLRU 表

这张表保存了对 shared pool 的分配所导致的 shared pool 中的对象被清出的记录。可以通过它来查找是什么导致了大的 shared pool 分配请求。

如果有许多对象定期会被从 shared pool 中被清出，会导致响应时间太长和 library cache latch 争用问题。

不过要注意一点，每当查询过表 X\$KSMLRU 后，它的内容就会被删除。因此，最好将查出的数据保存在一个临时的表中。以下语句查询 X\$KSMLRU 中的内容，

```
SELECT * FROM X$KSMLRU WHERE ksmlrsiz > 0;
```

- X\$KSMSP 表

从这张表中可以查到当前分配了多少空闲空间，这对于分析碎片问题很有帮助。一些语句可以查询 shared pool 的空闲列表中 chunk 的统计信息，

```
select '0 (<140)' BUCKET, KSMCHCLS, KSMCHIDX, 10*trunc(KSMCHSIZ/10)
"From",
count(*) "Count" , max(KSMCHSIZ) "Biggest",
trunc(avg(KSMCHSIZ)) "AvgSize", trunc(sum(KSMCHSIZ)) "Total"
from x$ksmsp
where KSMCHSIZ<140
and KSMCHCLS='free'
group by KSMCHCLS, KSMCHIDX, 10*trunc(KSMCHSIZ/10)
UNION ALL
select '1 (140-267)' BUCKET, KSMCHCLS, KSMCHIDX, 20*trunc(KSMCHSIZ/20) ,
count(*) , max(KSMCHSIZ) ,
trunc(avg(KSMCHSIZ)) "AvgSize", trunc(sum(KSMCHSIZ)) "Total"
from x$ksmsp
where KSMCHSIZ between 140 and 267
and KSMCHCLS='free'
group by KSMCHCLS, KSMCHIDX, 20*trunc(KSMCHSIZ/20)
UNION ALL
```

```

select '2 (268-523)' BUCKET, KSMCHCLS, KSMCHIDX, 50*trunc(KSMCHSIZ/50) ,
count(*) , max(KSMCHSIZ) ,
trunc(avg(KSMCHSIZ)) "AvgSize", trunc(sum(KSMCHSIZ)) "Total"
from x$ksmsp
where KSMCHSIZ between 268 and 523
and KSMCHCLS='free'
group by KSMCHCLS, KSMCHIDX, 50*trunc(KSMCHSIZ/50)
UNION ALL
select '3-5 (524-4107)' BUCKET, KSMCHCLS, KSMCHIDX,
500*trunc(KSMCHSIZ/500) ,
count(*) , max(KSMCHSIZ) ,
trunc(avg(KSMCHSIZ)) "AvgSize", trunc(sum(KSMCHSIZ)) "Total"
from x$ksmsp
where KSMCHSIZ between 524 and 4107
and KSMCHCLS='free'
group by KSMCHCLS, KSMCHIDX, 500*trunc(KSMCHSIZ/500)
UNION ALL
select '6+ (4108+)' BUCKET, KSMCHCLS, KSMCHIDX,
1000*trunc(KSMCHSIZ/1000) ,
count(*) , max(KSMCHSIZ) ,
trunc(avg(KSMCHSIZ)) "AvgSize", trunc(sum(KSMCHSIZ)) "Total"
from x$ksmsp
where KSMCHSIZ >= 4108
and KSMCHCLS='free'
group by KSMCHCLS, KSMCHIDX, 1000*trunc(KSMCHSIZ/1000);

```

如果使用 ORADEBUG 将 shared pool 信息 dump 出来，就会发现这个查询结果和 trace 文件中空闲列表信息一直。

如果以上查询结果显示大多数空闲 chunk 都在 bucket 比较小的空闲列表中，则说明系统存在碎片问题。

### 3.2.3.4. 编译 java 代码导致的错误

当编译 java（用 loadjava 或 deployjb）代码时产生了 4031 错误，错误信息一般如下：

```

A SQL exception occurred while compiling: : ORA-04031: unable to allocate
bytes of shared memory ("shared pool","unknown object","joxlod: init h",
"JOX: ioc_allocate_pal")

```

这里提示时 shared pool 不足，其实是错误，实际应该是 java pool 不足导致的。解决方法将 JAVA\_POOL\_SIZE 加大，然后重启实例。

### 3.2.3.5. Large pool 导致的错误

Large pool 是在 MTS、或并行查询、或备份恢复中存放某些大对象的。但和 shared pool 中的保留区（用于存放 shared pool 的大对象）不同，large pool 是没有 LRU 链表的，而后者使用的是 shared pool 的 LRU 链表。

在 large pool 中的对象永远不会被清出的，因此不存在碎片问题。当由于 large pool 不足导致 4031 错误时，可以先通过 v\$sgastat 查看 large pool 的使用情况，

```

SELECT pool,name,bytes FROM v$sgastat where pool = 'large pool';

```

或者做一个 dump，看看 large pool 中空闲 chunk 的大小情况。

进入 large pool 的大小条件是由参数 LARGE\_POOL\_MIN\_ALLOC 决定的，根据以上信息，可以适当调整 LARGE\_POOL\_MIN\_ALLOC 的大小。

Large pool 的大小是由 LARGE\_POOL\_SIZE 控制的，因此当 large pool 空间不足时，可以调整这个参数。

### 3.2.4. SGA 内存自动管理

10g 以后，Oracle 提供了一个非常有用的特性，即 SGA 内存自动管理。通过设置 SGA\_TARGET 可以指定总的 SGA 大小，而无需固定每个区的大小。这就是说，当分配 shared pool 或 large pool 时，只要 SGA 区足够大，就能获取到足够内存，因而可以大大减少 4031 错误发生的几率。

### 3.2.5. FLUSH SHARED POOL

使用绑定变量是解决 shared pool 碎片的最好方法。此外，9i 以后，可以设置 CURSOR\_SHARING 为 FORCE，强行将没有使用绑定变量的语句使用绑定变量，从而共享 SQL 游标。当采用以上措施后，碎片问题并不会马上消失，并可能还会长时间存在。这时，可以考虑 flush shared pool，将内存碎片结合起来。但是，在做 flush 之前，要考虑以下问题。

- Flush 会将所有没有使用的游标从 library cache 中清除出去。因此，这些语句在被再次调用时会被重新硬解析，从而提高 CPU 的占用率和 latch 争用；
- 如果应用没有使用绑定变量，即使 flush 了 shared pool 以后，经过一段时间运行，仍然会出现大量碎片。因此，这种情况下，flush 是没有必要的，需要先考虑优化应用系统；
- 如果 shared pool 非常大，flush 操作可能会导致系统被 hung 住。

因此，如果要 flush shared pool，需要在系统不忙的时候去做。Flush 的语法为，

```
alter system flush shared_pool;
```

### 3.2.6. TRACE 4031 错误

如果问题比较复杂（比如由于内存泄漏导致），或者你不幸遇上了 oracle 的 bug，这时就需要考虑设置 4031 事件来 trace 并 dump 出相关内存信息。

以下语句在整个系统设置 4031 事件，

```
SQL> alter system set events '4031 trace name errorstack level 3';  
SQL> alter system set events '4031 trace name HEAPDUMP level 3';
```

这个事件也可以在会话中设置，只要将以上语句中的“system”改为“session”就行了。

然后将 dump 出来的 trace 文件发给 oracle 吧。

不过注意一点，9205 以后就无需设置这个事件了，因为一旦发生 4031 错误时，oracle 会自动 dump 出 trace 文件。

## 4. Dump 内存解析

下面以 shared pool 为例，解释一下 dump 出来的内存结构。

```
SQL> conn sys/sys as sysdba  
Connected.  
SQL> oradebug setmypid  
Statement processed.  
SQL> oradebug dump heapdump 2  
Statement processed.  
SQL>
```

以下时 trace 文件的内容，我们分别解释各个部分：

```

Dump file
c:\oracle\product\10.2.0\admin\fuyuncat\udump\fuyuncat_ora_4032.trc
Tue Jul 11 16:03:26 2006
ORACLE V10.2.0.1.0 - Production vsnsta=0
vsnsql=14 vsnxtr=3
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining options
Windows XP Version V5.1 Service Pack 2
CPU : 2 - type 586
Process Affinity : 0x00000000
Memory (Avail/Total): Ph:885M/2039M, Ph+PgF:2702M/3890M, VA:1590M/2047M
Instance name: fuyuncat

Redo thread mounted by this instance: 1

Oracle process number: 18

Windows thread id: 4032, image: ORACLE.EXE (SHAD)

*** SERVICE NAME:(SYS$USERS) 2006-07-11 16:03:26.322
*** SESSION ID:(159.7) 2006-07-11 16:03:26.322

```

这部分是关于 trace 文件的基本信息，oracle 版本、资源情况、用户和会话等。

```

KGH Latch Directory Information
ldir state: 2 next slot: 75
Slot [ 1] Latch: 03C3D280 Index: 1 Flags: 3 State: 2 next:
00000000
Slot [ 2] Latch: 1EC9D4B0 Index: 1 Flags: 3 State: 2 next:
00000000
Slot [ 3] Latch: 1EC9D540 Index: 1 Flags: 3 State: 2 next:
00000000
Slot [ 4] Latch: 03C3E100 Index: 1 Flags: 3 State: 2 next:
00000001
Slot [ 5] Latch: 1ED65CE4 Index: 1 Flags: 3 State: 2 next:
00000000
Slot [ 6] Latch: 1ED65F14 Index: 1 Flags: 3 State: 2 next:
00000000
... ..

```

这部分记录的是 shared pool 中的 latch 信息。每个 latch 的具体信息可以通过视图 V\$LATCH、V\$LATCH\_PARENT、V\$LATCH\_CHILDREN 或者表 x\$ksllt 查出

```

*****
HEAP DUMP heap name="sga heap" desc=03C38510
extent sz=0x32c8 alt=108 het=32767 rec=9 flg=-126 opc=0
parent=00000000 owner=00000000 nex=00000000 xsz=0x10
*****

```

这是堆 dump 信息的头部，heap name 说明了内存所述的堆，shared pool 是属于 SGA 区的，因此，这里是“sga heap”；

extent sz 记录的是所有扩展段的大小。

```

HEAP DUMP heap name="sga heap(1,0)" desc=04EC131C
extent sz=0xfc4 alt=108 het=32767 rec=9 flg=-126 opc=0
parent=00000000 owner=00000000 nex=00000000 xsz=0x400000
EXTENT 0 addr=1CC00000
  Chunk 1cc00038 sz=          24 R-freeable "reserved stoppe"
  Chunk 1cc00050 sz=       212888 R-free     "                "
  Chunk 1cc33fe8 sz=          24 R-freeable "reserved stoppe"
  Chunk 1cc34000 sz=     3977544 perm      "perm            " alo=3977544

```

```

Chunk 1cfff148 sz=      3768   free   "           "
EXTENT 1 addr=1D000000
Chunk 1d000038 sz=         24 R-freeable "reserved stoppe"
Chunk 1d000050 sz=    212888   R-free   "           "
Chunk 1d033fe8 sz=         24 R-freeable "reserved stoppe"
Chunk 1d034000 sz=    2097168   perm   "perm       " alo=2097168

```

这部分信息是 trace 文件中的主要部分，它详细记录了 shared pool 中各个 chunk 的信息。

首先看它的头部信息，注意到这里 heap name 是“sga heap(1,0)”。这是什么意思呢？我们前面提到，oracle 10g 会将 shared pool 分为几个区来管理，这里就是其中的一个区。共有 4 个区。通过表 X\$KGHLU 可以看到对应的 LRU 链表。

```
EXTENT 0 addr=1CC00000
```

这一行说明下面的 chunk 都属于这个扩展段 (extent)，0 是它的编号，addr 是它的起始地址。

```
Chunk 1cc00038 sz=         24 R-freeable "reserved stoppe"
```

这是一个 chunk 的信息，sz 是这个 chunk 的大小 (24 字节)。R-freeable 是这个 chunk 的状态，“reserved stoppe”是这个 chunk 的用途。Chunk 有 4 种可能状态，以下是这四种状态的含义：

**free:** 即空闲 chunk，可以随时分配给适合大小的请求；

**freeable:** 这种状态的 chunk 表示它当前正在被使用，但是这种使用是短期的，比如在一次调用中或者一个会话中，会话或者调用解释就可以被释放出来。这种状态的 chunk 是不放在 LRU 链表中的，一旦使用结束，自动成为 free 状态，放到空闲列表中；

**recreatable:** 这种状态的 chunk 正在被使用，但是它所包含的对象是可以被暂时移走、重建，比如解析过的语句。它是被 LRU 链表管理的。

**permanent:** 顾名思义，这种状态的 chunk 所包含的对象是永远不会被释放的。即使 flush shared pool 也不会释放。

我们注意到，这里还有一些状态是有前缀“R-”的。带有这种前缀的 chunk 说明是 shared pool 中的保留区的 chunk。

```
Total heap size = 41942480
```

最后是这一 shared pool 区的总的大小。

```
FREE LISTS:
```

```

Bucket 0 size=16
Bucket 1 size=20
Chunk 166ed050 sz=         20   free   "           "
Chunk 167de068 sz=         20   free   "           "
Chunk 164b9c10 sz=         20   free   "           "
Chunk 1f2776f8 sz=         20   free   "           "

```

接下来便是这个 shared pool 区的空闲列表。Bucket 是一个空闲列表的范围，例如 Bucket 1，它的最小值是上一个 Bucket 的最大值，即 16，最大值为 20。Bucket 下面是空闲列表中 chunk，后面的信息和前面解释 chunk 的信息一样，8 位的 16 进制数字是它的地址；sz 是 chunk 的大小；free 是 chunk 的状态，因为是空闲列表中的 chunk，这里只有一个状态；最后是 chunk 的用途，因为都是 free，所以肯定为空。

```
Total free space = 1787936
```

最后是这块 shared pool 区中空闲 chunk 的总的大小。

```
RESERVED FREE LISTS:
```

```

Reserved bucket 0 size=16
Reserved bucket 1 size=4400

```

```

Reserved bucket 2 size=8204
Reserved bucket 3 size=8460
Reserved bucket 4 size=8464
Reserved bucket 5 size=8468
Reserved bucket 6 size=8472
Reserved bucket 7 size=9296
Reserved bucket 8 size=9300
Reserved bucket 9 size=12320
Reserved bucket 10 size=12324
Reserved bucket 11 size=16396
Reserved bucket 12 size=32780
Reserved bucket 13 size=65548
Chunk 1b800050 sz= 212888 R-free " "
Chunk 16c00050 sz= 212888 R-free " "
Chunk 1ac00050 sz= 212888 R-free " "
Total reserved free space = 638664

```

Shared pool 的普通区的空闲列表下面就是关于这块 shared pool 区中保留区的空闲列表的描述，其中除了在名字上 bucket 前面都有一个 Reserved 标识，和状态前面有“R-”前缀外，含义和普通空闲列表相同。

```

UNPINNED RECREATABLE CHUNKS (lru first):
  Chunk 1aee99c0 sz= 4096 recreate "sql area "
latch=1D8BDD48
  Chunk 1ae4aeec sz= 4096 recreate "sql area "
latch=1D8BDDB0
... ..
SEPARATOR
  Chunk 166e8384 sz= 540 recreate "KQR PO "
latch=1DD7F138
  Chunk 1f333a5c sz= 284 recreate "KQR PO "
latch=1DC7DFC8
  Chunk 166e9340 sz= 540 recreate "KQR PO "
latch=1DE00A70
  Chunk 1f0fe058 sz= 284 recreate "KQR PO "
latch=1DC7DFC8
  Chunk 1f2116b4 sz= 540 recreate "KQR PO "
latch=1DE81910
  Chunk 1f21127c sz= 540 recreate "KQR PO "
latch=1DE81910
... ..
Unpinned space = 1611488 rcr=645 trn=864

```

空闲列表后面就是 LRU 链表了。LRU 链表不是按照大小分的，因而没有 Bucket。它的 chunk 是按照最近最少使用的顺序排列。其中 chunk 的信息和前面解释的一样。但是要注意一点，因为 LRU 链表中的 chunk 都是使用的，因为每个 chunk 根据用途不同，都会有一个 latch 来保护，Chunk 信息最后便是 latch 的地址。

注意，我们前面提到，shared pool 中是有两种 LRU 链表的，一种循环 LRU 链表；另外一种暂时 LRU 链表。在这里 LRU 信息中前面部分是循环 LRU 链表，SEPARATOR 后面部分是暂时 LRU 链表信息。

最后是 LRU 链表中 chunk 的总的大小，rcr 是循环 LRU 链表中的 chunk 数，trn 是暂时 LRU 链表中的 chunk 数。

此外，有一点提示，如果是有多多个 shared pool 区，第一个区是不含 LRU 链表信息的。

```

PERMANENT CHUNKS:
  Chunk 1d234010 sz= 1884144 perm "perm " alo=1728440
  Chunk 1cc34000 sz= 3977544 perm "perm " alo=3977544
  Chunk 1d034000 sz= 2097168 perm "perm " alo=2097168

```

```
Chunk 1d434000 sz= 3117112 perm "perm" alo=3117112
... ..
Chunk 1f434000 sz= 3917704 perm "perm" alo=3917704
Permanent space = 38937696
```

最后是永久 chunk 的信息。Chunk 部分解释和前面一致。alo 表示已经分配的大小。如果有多个 shared pool 区，永久 chunk 信息则只存在于第一个 shared pool 区。