HZ BOOKS
华章科技

Oracle数据库领域传奇人物、前阿里B2B最高级别Oracle DBA吕海波（VAGE）10余年职业生涯的集大成之作

深入分析和挖掘Oracle数据库内核中的精髓与秘密，揭示了大量鲜为人知的原理和算法，并详细阐释了如何建立一套自己的调优排故模型

数据库技术丛书

Oracle Core Revealed

Oracle内核技术揭密

吕海波◎著

机械工业出版社
China Machine Press

数据库技术丛书

Oracle 内核技术揭密

吕海波　著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Oracle 内核技术揭密 / 吕海波著 . —北京：机械工业出版社，2014.7
（数据库技术丛书）

ISBN 978-7-111-46931-5

I. O… II. 吕… III. 关系数据库系统 IV. TP311.138

中国版本图书馆 CIP 数据核字（2014）第 115943 号

Oracle 内核技术揭密

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：陈佳媛 责任校对：董纪丽
印 刷： 版 次：2014 年 9 月第 1 版第 1 次印刷
开 本：186mm×240mm 1/16 印 张：23
书 号：ISBN 978-7-111-46931-5 定 价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

Preface 前　言

美国有一句著名的谚语：如果上帝关闭了一扇门，他会为你打开一扇窗。美国还有一个有名的关于 Oracle 的笑话：上帝和埃里森的区别就是，上帝不认为自己是埃里森。

无论上帝怎么想，埃里森肯定认为自己是上帝，至少，是数据库界的上帝。这位数据库界的上帝所开创的著名的 Oracle 数据库软件是闭源的，对于想研究 Oracle 的 DBA 来说，相当于关上了一扇门。但同时 Oracle 中提供大量的 DUMP 命令，这又相当于为 DBA 打开了一扇窗。但现在，这扇窗正在慢慢关闭。

很早之前，有很多从 Oracle 公司流向外界的“内部资料”，对这些内部资料的研究甚至成为学习 Oracle 的一个专门分支：Oracle Internal。当时很多 DBA 都会在简历中加上一行：“精通 Oracle Internal”。当然，笔者也不例外。但是，近五六年来，不知是因为 Oracle 公司加强控制，还是因为众 DBA 研究热情下降，或者二者兼而有之，市面上所见的“内部资料”越来越少，特别是近两三年，基本已经绝迹。

从笔者来看，造成这种情况是“二者兼而有之”。“ Oracle 有意控制”论并非空穴来风，伴随着 Oracle 数据库应用得越来越广泛，第三方维护市场的发展也是如火如荼。如果有了疑难问题只能找 Oracle 原厂的售后团队解决，那么第三方维护公司将很难与 Oracle 竞争，这是控制这块市场的最好方法。实现这一目标的捷径，当然就是控制非原厂 DBA 对 Oracle 数据库的了解程度。但是另一方面，Oracle 对 Oracle 技术社区的支持还是有些力度的。所以个人感觉是二者兼而有之，但愿我是“以小人之心，度君子之腹”。无论 Oracle 是否有意控制，现在 Internal 爱好者越来越少已是不争的事实，这与 Oracle 的闭源策略有很大关系。

笔者早年也是“ Oracle Internal”研究的爱好者，个人认为，对 Internal 的研究分为以下 3 个阶段：

- ❑ 好奇。
- ❑ 学以致用。
- ❑ 麻木。

好奇之心，人皆有之。对 IT 技术有兴趣的 DBA，在接触 Oracle 不久后，总会对

Oracle 内部算法产生强烈的好奇心：它是如何计算 HASH 值的？它是如何搜索 HASH 链表的？检查点到底完成了什么操作？如果你有这种好奇心，并且有强烈的探索欲望，恭喜你，你将有望成为高级 DBA。跟着你的好奇心去探索吧，在网上查找资料，再加上自己动手做测试，很快你就会对各种 Internal 有朦胧的了解。当出现一些等待事件、一些性能问题或故障时，你就可以从你所了解的原理出发，去分析问题了。这就是第二阶段——“学以致用”，到这一步，“高级 DBA”这个头衔就在向你招手了。

但是很快，你会发现，看不到 Oracle 的源代码，仅从各种 DUMP 结果中分析算法，无异于“隔靴搔痒”，有个最恰当的成语描述这种情况：雾里看花。你会发现，我们雾中看出来的“花”，在解决实际问题时，作用极为有限。一些疑难问题，还是无法理清头绪。这样的事情经历多了，就进入了“麻木”阶段。自然也会得出结论，对 Internal 的研究，现实意义有限，主要是满足好奇心。然后，有可能还会语重心长地告诫初学者，Internal 意义不大，浅尝辄止即可，不必浪费太多精力。当这样的情况越来越多后，人们的“研究”欲望自然会越来越低。所以，现在基本上已经很少有人再去研究 Internal 了。

反观开源领域，虽然源代码的改动并不是“想改你就改”那么简单，因为有各种各样的管理委员会控制，如果不能成为委员，改一个 Bug 都难，但由于可以看到源代码，只要下工夫钻研代码，想做到“明明白白我的心”并不难。这样一来，在遇到一些奇怪问题时，进行诊断、分析将更有依据。这其实是开源数据库在国内流行的主要原因之一。国内的 Committer 并不多，就算能读懂源码也不能修改，在这种情况下，开源除了“便宜”、成本低之外，还有什么优势呢？优势就是“用得明白”、“用得清楚”。闭源的 Oracle 虽然 Bug 更少、更稳定，但出了奇怪的问题后就很难解决。开源则不一样了。一边是 Oracle 的“雾里看花”，你看不到隐藏在暗处的原理；一边是开源领域的“明明白白我的心”，众多技术人员当然选择“弃暗投明 ”了。

而且近些年，由于前面说的两点原因，Oracle 这场雾更浓了，已经变成“雾霾”了。以前著名的内部资料 DSI 也止步于 Oracle 9i，鲜见 Oracle 10g 版本的，更别说 Oracle 11g 的了。外界 DBA 由于缺乏对原理的了解，很多基本操作都要依赖原厂工程师。比如 Exadata，如果没有原厂工程师协助，连安装都很难完成，更别谈运维了。Oracle 的大门从来就没有敞开过，现在，连“窗”也在逐渐关闭。

变革正在到来。在“门”、“窗”都没有的大环境下，或许可以选择把墙给凿了。凿穿墙壁，一样能看到 Internal。不过，工欲善其事，必先利其器。如果没有适当的工具，想要打开 Oracle 这样庞然大物般的软件无疑是以卵击石。幸运的是，现在凿墙利器已经有了，那就是“动态性能跟踪”语言，比如，Linux 下的 System Tap，Solaris 下的 DTrace，等等。经过笔者试用比较，Solaris 下的 DTrace 更适合用来研究 Oracle 原理。虽然我们只能在 Solaris 平台使用，但 Oracle 的原理在所有 OS 下是一样的（除了在 Windows 下略有不同）。在 Solaris 下研究出的原理，一样可以用于其他平台，可方便大家进行性能调优、故障诊断。笔者书中大多数 Oracle 内部原理的结论，都是使用 DTrace 加 MDB 分析而得出的。

但由于笔者精力有限，而且部分结论还要对 Oracle 的核心代码反汇编才能得出，这将耗费更多精力，因此难免个别结论分析有误，如果各位读者朋友在阅读时发现有误之处，敬请告知，笔者不胜感谢。笔者的 BLOG 地址为：www.mythdata.com，有问题大家可以到该博客留言探讨。

DTrace 跟踪，再加上调试工具 MDB，笔者称为 DBA 中新的领域：“调试 Oracle”。调试技术的引入，加上我们对 Oracle 的理解，可以让我们把 Oracle 原理看得更加清晰，可以达到与阅读开源代码同样的效果。这如同吹散“雾霾”的北风，让我们不再“雾里看花”。对原理把握得更加清晰，也使得调优、故障诊断更加精准。“ Change”，是这两年最流行的词汇。“我们一直身处变革之中而不自知”。变革正在悄然进入 DBA 领域，传统的 Oracle 运维日渐式微，这已然是不争事实。众多处于“麻木”阶段的 DBA 纷纷转行。“调试 Oracle”领域的出现，将是一条新的发展之路。希望本书能给大家提供一种新的思路。

目　录 *Contents*

第 1 章 Chapter 1

存储结构

存储结构，其实就是表空间、数据文件中的空间组织和使用形式，也许有人会认为存储结构不过是基础知识，相对简单，其实这里面隐藏了很多 Oracle 的秘密，如果不注意挖掘，将无法把 Oracle 提供的性能特性完全发挥出来，为我们所用。本章将会抽丝剥茧地为大家全面介绍表空间、数据文件的知识点，除此以外，在本章的最后，还会提供一个精彩的实际案例，帮助大家进一步理解相关知识点，但希望大家不要急着直接学习案例，这样没有太大意义。那么，下面就让我们来扮演一次福尔摩斯，亲手揭开存储结构的谜团吧！

1.1 区：表空间中的基本单位

区，Extent，逻辑上连续的空间。它是表空间中空间分配的基本单位。如果在某表空间中创建一个表，哪怕只插入一行，这个表至少也会占一个区。

具体来讲，在 Oracle 10g 中，如果创建一个新表，初始至少为这个表分配一个区。而在 Oracle 11.2.0.3 以上版本中，创建新表时默认一个区都不会分配，也就是说，这个表此时不占存储空间。只有在向表中插入第一行数据时，才会默认为表分配第一个区。

无论是 Oracle 10g 还是 Oracle 11GR2，如果表原有区中的空间用完了，Oracle 就会默认为表一次分配一个区的空间。

可以通过 DBA_EXTENTS 数据字典视图查看表所属区。假设有一个表 Table1，想要查询它所在的区，可通过如下方式：

```
select extent_id, file_id, block_id, blocks from dba_extents where
segment_name='TABLE1' order by extent_id;
```

上面语句中，每个列的意义都很简单，这里不再介绍，如有问题，可以查看 Oracle 联机文档 Oracle Database Reference 中的视图介绍。

说了这么多，大家会不会有一个疑问：既然区这么重要，是空间分配的基本单位，那么，区的大小是如何定义的呢？

Oracle 专门设定了两种类型的表空间：统一区大小表空间和系统管理区大小表空间。区的大小就是由这两种表空间决定的。下面，先从统一区大小讲起。

1.1.1 统一区大小表空间和区的使用规则

统一区大小的表空间理解起来很简单，顾名思义，就是创建表空间时，设定区大小为一个统一的值。如下命令创建一个区大小为 1MB 的表空间：

```
create tablespace tbs_ts1 datafile '/u01/Disk1/tbs_ts1_01.dbf' size 50m uniform size 1m;
```

tbs_ts1 表空间包含一个 50MB 的数据文件，区大小为 1MB。在此表空间中创建一个测试表：table1，观察一下区大小。

```
SQL> create table table1(id int,name varchar2(20)) tablespace tbs_ts1;
Table created.

SQL> insert into table1values(1,'VAGE');

1 row created.

SQL> commit;

Commit complete.

SQL> select extent_id, file_id, block_id, blocks from dba_extents where
segment_name='TABLE1' order by extent_id;
EXTENT_ID    FILE_ID   BLOCK_ID     BLOCKS
---------- ---------- ---------- ----------
0            4         128          128
```

可以看到，table1 表目前只包含一个区，它从 4 号文件的第 128 号块开始，大小为 128 个块。笔者这里的块大小为 8KB，128 个块，正好就是 1MB。

从上面的结果可以看到，表 table1 从 4 号文件的 128 号块开始占用空间。从 128 ~ 257 号块是 table1 的第一个区，那么，0 ~ 127 号块又是干什么用的呢？

事实上，每个文件的前 128 个块，都是文件头，被 Oracle 留用了。在 Oracle 10g 中是 0 至 8 号块被 Oracle 留用。而从 Oracle 11GR2 开始，一下就留用 128 个块，真是大手笔，不是吗？

这一部分文件头又分两部分，其中 0 号、1 号块是真正的文件头，2 ~ 127 号块是位图块。而在 Oracle10g 中，2 ~ 8 号块则是位图块。

这个位图块又是干什么用的呢？

很容易理解，是用来记录表空间中区的分配情况的。位图块中的每一个二进制位对应一个区是否被分配给某个表、索引等对象。如果第一个二进制位为 0 说明表空间中第一个区未分配，如果为 1 说明已分配；第二个二进制位对应第二个区，以此类推，如图 1-1 所示。

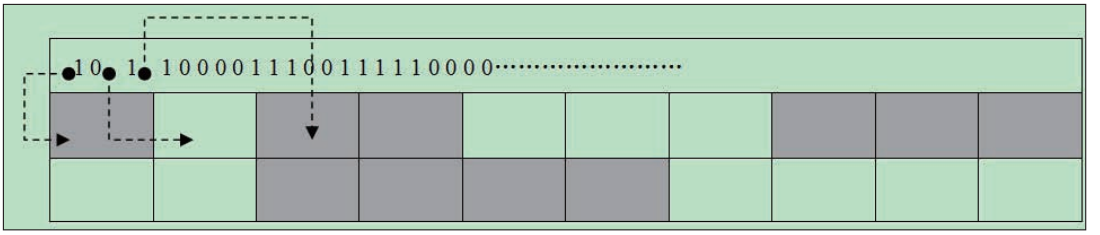

图 1-1　位图块示意图

在图 1-1 中，上面的二进制位就是位图块中的数据，下面表格表示区，其中灰色区是已分配区，白色区是未分配区。第 1 个区对应的二进制位是 1，代表此区已分配，第 2 个区对应的二进制位是 0，表示此区未分配，等等。

位图块又分两部分，其中第一个位图块又被当作位图段头，可以在 DUMP 文件中找到 Oracle 对此块类型的说明：Bitmapped File Space Header。从第二个位图块也就是 3 号块开始，就是真正的位图数据了，DUMP 文件中这些块的类型说明为：Bitmapped File Space Bitmap。

关于位图块的 DUMP 格式描述，大家可以在网上找一下，此处不赘述。下面讨论一个网上较少讨论的话题：当要分配区时，Oracle 如何在位图块中搜索可用区。

大家可以考虑一个问题，如果块大小为 8KB，0 号、1 号块是文件头，2 号块是位图头，在 Oracle 10g 中，3 ~ 8 号块是位图数据块，共 6 个位图块，大小是 48K 字节，每个字节 8 个二进制位，一共 393216 个二进制位。每个二进制位对应一个区，一共就 393216 个区。如果是 Oracle 11GR2，这个数字又要多出好多倍。那么，如果某个表要在数据文件中分配一个新区，Oracle 如何在这 30 多万个二进制位中，确定哪个二进制位对应的区可以分配给表呢？

Oracle 用的方法其实很简单，在位图块中，找一个标记位，如果 0~2 号区被占用了，标记位的值为 3；如果 3~4 号区又被占用了，标记位增加为 5。假设此时 2 号区被释放了，标记位变为 2。

如果需要分配新的区，从这个标记位处开始查找即可。假设目前标记位值为 5，有进程需要 4 个未分配区，Oracle 就从 5 号区开始向下查找。

需要注意的是，如果开启了闪回 Drop，而且 Drop 了表，那么，区并不会被释放，因此标记位不会下降。因为 Drop 只是改名，而并不会真正删除表。

1.1.2 系统管理区大小

刚才介绍了统一区大小，并且测试了区的分配规则。在系统管理区中，区的分配规则是一样的，但是，区大小的设定不再由人为决定，Oracle 会根据表大小自动设置。

Oracle 如何设定区大小呢？只需要创建一个表空间，并进行很简单的测试，就能搞明白这点。命令如下所示：

```
create tablespace tbs_ts2 datafile '/u01/Disk1/tbs_ts2_01.dbf' size 50m reuse;
```

上面的命令创建了一个 tbs_ts2 表空间，至于区大小的管理方式，这里没有指定，这样 Oracle 默认创建的，就会是系统管理区的大小。

现在在 tbs_ts2 中创建表，并观察它的区大小。

```
SQL> create table table_lhb1(id int,name varchar2(20)) tablespace tbs_ts2;
Table created.

SQL> select extent_id, file_id, block_id, blocks from dba_extents where segment_
name='TABLE_LHB1' order by extent_id;

 EXTENT_ID    FILE_ID   BLOCK_ID     BLOCKS
---------- ---------- ---------- ----------
         0          5        128          8
```

可以看到 TABLE_LHB1 目前有一个区，大小只有 8 个块，也就是 64KB。插入一些数据，将表撑大点再观察。

```
SQL> insert into table_lhb1 select rownum,'aaa' from dba_objects;

12650 rows created.

SQL> select extent_id, file_id, block_id, blocks from dba_extents where segment_
name='TABLE_LHB1' order by extent_id;

 EXTENT_ID    FILE_ID   BLOCK_ID     BLOCKS
---------- ---------- ---------- ----------
         0          5        128          8
         1          5        136          8
         2          5        144          8
         3          5        152          8
```

上面向表中插入了 12650 行，表目前涉及 4 个区，每个区都是 8 个块 64KB。提交后继续插入，命令如下：

```
SQL> insert into table_lhb1 select * from table_lhb1;

12650 rows created.

SQL> insert into table_lhb1 select * from table_lhb1;
```

```
25300 rows created.

SQL> insert into table_lhb1 select * from table_lhb1;

50600 rows created.

SQL> insert into table_lhb1 select * from table_lhb1;

101200 rows created.

SQL> commit;

Commit complete.

SQL> select extent_id, file_id, block_id, blocks from dba_extents where segment_
name='TABLE_LHB1' order by extent_id;

 EXTENT_ID    FILE_ID   BLOCK_ID     BLOCKS
---------- ---------- ---------- ----------
         0          5        128          8
         1          5        136          8
         2          5        144          8
         3          5        152          8
         4          5        160          8
         5          5        168          8
         6          5        176          8
         7          5        184          8
         8          5        192          8
         9          5        200          8
        10          5        208          8
        11          5        216          8
        12          5        224          8
        13          5        232          8
        14          5        240          8
        15          5        248          8
        16          5        256        128
        17          5        384        128

18 rows selected.
```

多次插入后，表已经占了很多空间，我们来看一下现在区的使用情况。0 ~ 15 号区，大小为 8 个块 64KB，从第 16 号区开始，区大小变为了 128 个块 1MB 大小。也就是说，表的大小小于 1MB 时，表的每个区都是 64KB，当表的大小超过 1MB，再分配新区时，区的大小将是 1MB。读者可以继续测试，当表进一步变大，区大小将会变成 8MB，表继续扩大，更大的区也有，这里就不测了。

可见，在系统管理区大小表空间中，区的大小随表的增大而增大。

到这里，我们已经发现了系统管理区大小的秘密。很简单吧？有时候探索 Oracle 也不

是件复杂的事情，只要多动手就行了。我在做培训的时候，也常跟学员讲：探索的过程，就是熟悉 Oracle 的过程。你探索出的结果不一定很有用，但探索的过程，会加深对 Oracle 的熟悉程度，这才是研究、探索 Oracle 的真正目的。研究 Oracle，很多时候结果不是目的，过程更有意义。

我们已经了解了统一区大小和系统管理区大小的不同，那么，什么时候使用统一区大小，什么时候使用系统管理区大小呢？

从空间的利用率上讲，小区节省空间，大区可能会浪费空间。比如，当区大小是 10MB 时，为一个表分配了一个 10MB 的区，哪怕它只使用了这 10MB 中的 1 个字节，这 10MB 空间也完全属于这个表了，其他表无法再使用这部分空间。从这个角度上讲，小区的空间利用率无疑是高的。

但从性能角度上讲，对于随机访问，大区、小区没有影响。但对于全表扫描这样的操作，大区又是更合适的。因为连续空间更多，可以减少磁头在区间的定位。

在系统管理区大小的方式下，当表比较小时，区也比较小，当表大时，区也随之变大，这种方式无疑可以在空间的利用率、全扫描的性能之间找到一种平衡。因此建议大多数情况下，都可以采用系统管理区大小的方式。除非有某个表，已明确地知道它会很大，为了保证全扫描的性能，直接建一个统一区大小，并且区比较大的表空间，以便将表存放其中。

如果使用统一区大小，几百 KB 甚至 1MB 的区，都有点小。其实可以参考系统管理型的表空间，当段的大小超过 64MB 时，区大小为 8MB。在使用统一区大小时，也可以将所有区都固定为 8MB。

我见到过很多数据库都使用统一区大小，而且其大小为 1MB。原因是在大部分的操作系统中，一次 I/O 操作的最大的读、写数据量是 1MB。即使使用 8MB 的区，一个区也必须分 8 次进行 I/O 操作，超过 1MB 的区大小，并不能减少 I/O 操作的次数。

但是，我们要考虑一点，8MB 的区连续的空间更多。读取 8MB 内的第 1MB 和第 2MB 数据虽然必须要分两次 I/O 操作，但这两次 I/O 操作很可能是连续 I/O，因为第 1MB 和第 2MB 数据有可能是相连的。如果区大小仅为 1MB，虽然读取表的第 1 区和第 2 区也是两次 I/O 操作，但这两次 I/O 操作很可能不相连，是随机 I/O 操作。连续 I/O 操作的性能当然比随机 I/O 操作的要高。

因此，出于全表扫描性能的考虑，即使使用统一区大小，大点的区（如 8MB 大小）是很合适的选择。

还有一个问题，不知道大家有没考虑到。这个问题涉及统一区大小表空间的位图块。每个二进制位对应一个区的使用情况，这是没问题的，但系统管理区大小呢？就比如刚才创建的 TABLE_LHB1 表，前 16 个区大小为 64KB，之后的区大小为 1MB。区的大小不同，如何用二进制位来反映区的使用情况呢？

Oracle 的处理方法是这样的，以 64KB（也就是 8 个块）为准，每个二进制位对应 64KB。1MB 的区，对应 16 个二进制位。每分配一个 1MB 的区，Oracle 将对应的 16 个二

进制位（也就是两个字节）设置为 1。释放一个区也同样，将 16 个二进制位设置为 0。这样就解决了区大小不统一的问题，Oracle 的解决方法还是很巧妙的！

1.1.3　碎片：少到可以忽略的问题

最后，来想这样一个问题：在表空间级别有碎片吗？

答案是：有，但也要看情况。在统一区大小表空间中，因为区的大小一致，不会出现碎片问题。但在系统管理区中，由于区的大小不一致，仍会存在碎片。比如说，有很多个 64KB 的区，互相不连续，分布在数据文件的各个角落。当需要 1MB、8MB 大小的区时，这些不连续的 64KB 区无法被重用，这就是典型的碎片了。但是，这种情况很少出现。因为区不会被频繁地分配、释放。一个表创建之后，很少会去对它进行 Drop、Truncate 操作。

没有频繁的分配、释放操作，碎片也就很少出现了。所以，在表空间层，碎片已经是一个可以忽略的问题了。当然，在表层、索引层，还有可能存在碎片。

1.2　段中块的使用

在讲解本节主题前，我们先来理清一个概念，什么是段。在 Oracle 中，表和段是两个截然不同的概念。表从逻辑上说明表的形式，比如表有几列，每列的类型、长度，这些信息都属于表。而段只代表存储空间，比如，上节中提到的区，就是属于段。一个段中至少要包含一个区。

Oracle 中，每个对象都有一个 ID 值，表有表的 ID，段有段的 ID。在 DBA_OBJECTS 数据字典视图中，object_id 列是表 ID，data_object_id 列是段 ID，下面查看了某个表的表 ID 和段 ID：

```
SQL> create table lhb.table_lhb2 (id int,name varchar2(20)) tablespace tbs_ts2;
Table created.

SQL> select object_id,data_object_id from dba_objects where owner='LHB' and
 object_name='TABLE_LHB2';

 OBJECT_ID DATA_OBJECT_ID
---------- --------------
 13039     13039
```

从上面信息可知，这里创建了一个表 TABLE_LHB2，初始情况下，它的表 ID 和段 ID 是一样的，都是 13039。

表 ID 一旦创建，就不会再改变。但段 ID 是会变化的，比如，当 Truncate 表时，Oracle 会将表原来的段删除，再为表新建一个段。也就是将表原来的存储空间释放，再重新分配新的区。这个过程完毕后，表就换了一个段，所以，表 ID 不变，但段 ID 却变了。如下所示：

```
SQL> insert into lhb.table_lhb2 values(1,'abc');
1 row created.

SQL> commit;
Commit complete.

SQL> truncate table lhb.table_lhb2;
Table truncated.

SQL> select object_id,data_object_id from dba_objects where owner='LHB' and
object_name='TABLE_LHB2';

 OBJECT_ID DATA_OBJECT_ID
---------- --------------
 13039     13040
```

可以看到，在 Truncate 表后，OBJECT_ID 不变，DATA_OBJECT_ID 变了。基本上，每 Truncate 一次，段 ID 都会加 1。

注意，上面的测试是在 Oracle 11GR2 中做的，如果是在 Oracle 10g 中，创建表后不需要插入一行，直接 Truncate，就可以观察到段 ID 的变化。

1.2.1 块中空间的使用

一个块的大小最常见是 8KB。对于这 8KB 空间的使用，网上已经有很多描述，这里简单说一下。块中信息分两部分：管理信息和用户数据，其中，管理信息包括块头的 SCN、ITL 槽等。

块的结构相信很多人也研究过，下面讨论一个常见问题：如果删除了一行，再回滚，行的位置会变吗？

测试如下：

```
SQL> select dbms_rowid.ROWID_RELATIVE_FNO(rowid) fno,dbms_rowid.rowid_block_
number(rowid) block_id,dbms_rowid.ROWID_ROW_NUMBER(rowid) row_id,id from lhb.
table_lhb2;
       FNO   BLOCK_ID     ROW_ID         ID
---------- ---------- ---------- ----------
         5        517          0          1
         5        517          12<---（删除此行再回滚）
         5        517          23
         5        517          34
```

这里使用了一个包，dbms_rowid，它的作用是从 ROWID 中将对象 ID、文件号、块号、行号分解出来。或者把对象 ID、文件号、块号、行号合并成 ROWID，具体使用方法这里不再列出，可以参考 Oracle 官方文档 PL/SQL Reference，其中有详细的说明。这里，使用它的第一种功能，从 ROWID 中解析出块号、行号等信息。如果向 lhb.table_lhb2 表中依次插入 ID

为 1、2、3、4 的 4 行数据，观察 ROW_ID 列，可以看到，这 4 行的行编号分别是 0、1、2、3。

下面将 ID 为 2 的行（行编号是 1）删除，再回滚，然后再次查看。

```
SQL> delete lhb.table_lhb2 where id=2;
1 row deleted.

SQL> rollback;
Rollback complete.

SQL> select dbms_rowid.ROWID_RELATIVE_FNO(rowid) fno,dbms_rowid.rowid_block_
number(rowid) block_id,dbms_rowid.ROWID_ROW_NUMBER(rowid) row_id,id from lhb.
table_lhb2;

       FNO   BLOCK_ID     ROW_ID         ID
---------- ---------- ---------- ----------
         5        517          0          1
         5        517          12<---(回滚后行号不变)
         5        517          23
         5        517          34
```

结果不变，ID 为 2 的行，还是在行号为 1 的位置。

道理很简单，删除某行，其实只是在行上加个删除标志，声明此行所占的空间可以被覆盖。在没有提交时，事务加在行上的锁并没有释放，此行虽然已经打上了删除标志，但空间仍不会被其他行覆盖。而删除行的回滚，其实就是将被删除的行重新插入一次。但回滚时的插入和普通插入一行还是有一定区别的。因为被删除行的空间不会被覆盖，所以回滚时的插入，不需进行寻找空间的操作，而是行原来在哪儿，就还插入到那里。这也就是它和普通插入的区别。

因此，删除的回滚，不会改变行原来的位置。

但如果删除后提交再插入呢？行的位置肯定就会发生变化了。

1.2.2　典型问题：堆表是有序的吗

曾经有位开发人员跟我聊到，他曾做过测试，插入几万行，删掉，再插入，发现原来 Oracle 中堆表是按插入顺序安排行的位置的，而且这个测试他做了好多遍，都是这个结果。现在他们有个应用，显示数据时，要求先插入的行在前，后插入的行在后，其实 Oracle 已经帮他们实现了这个功能。

事实上，堆表是无序的，堆表的特点就是无序、插入快速。

Oracle 在插入行时是如何在数据块内查找可用空间的呢？这有点类似于上节中提到的区的分配过程。Oracle 会在数据块中设立一个标记位，记录空间使用到哪儿了。

块中用户数据所占空间是从下往上分配的。假设，在 8192 字节的块中插入了 5 行，每行 100 字节，也就是说，空间已经使用到了（8192−500）7692 字节处，那么，标记位的值就是 7692。

如果删除了其中一行并提交，标记位的值不会变，还是 7692。再重新插入被删除行，或插入新行，将会从 7692 处向上查找可用空间，删除行释放出的空间不会被使用。

当标记位的值越来越小，向上到达管理性信息的边界时，标记位会再变为 8192。

我们可以测试一下。

```
SQL> delete lhb.table_lhb2 where id=2;
1 row deleted.

SQL>commit;
Commit complete.

SQL> insert into lhb.table_lhb2 values (2,'ABC');
1 row created.

SQL> commit;
Commit complete.

SQL> select dbms_rowid.ROWID_RELATIVE_FNO(rowid) fno,dbms_rowid.rowid_block_
number(rowid) block_id,dbms_rowid.ROWID_ROW_NUMBER(rowid) row_id,id from lhb.
table_lhb2;

       FNO   BLOCK_ID     ROW_ID         ID
---------- ---------- ---------- ----------
         5        517          0          1
         5        517          2          3
         5        517          3          4
         5        517          4          2<------ 删除提交后再插入，
                                              被分配到了新的位置
```

在上面的测试中，先删除 ID 为 2 的行，提交后接着又插入 ID 为 2 的行。不过，新插入的行并没有使用刚刚删除行的空间。

如果只测试到这一步，很容易得出结论，行的位置就是插入顺序。但别忘了，我们只在一个块内进行了观察，查找了可用空间。在众多的块中，Oracle 是如何选择要向哪个块中插入的呢？情况会不会有变化呢？我们还不知道。

所以，现在还不能完全回答“堆表是有序的吗”这个问题，继续向下看，据说 ASSM 对插入的影响是巨大的，那接下来看看 ASSM。

1.2.3 ASSM 与 L3、L2、L1 块的意义

ASSM 的目的是大并发插入，这应该是 DBA 要掌握的基本知识。在输入输出能力满足的情况下，使用 ASSM 就能有大并发插入吗？这可不一定。工具再好，还要看我们如何使用工具。

在了解 ASSM 的使用注意事项之前，先来分析一下 ASSM。为什么 Oracle 对外宣称 ASSM 可以支撑大并发插入应用呢？

ASSM 的整体结构是 3 层位图块 + 数据块，即共 4 层的树状结构。

第一层位图块称为 L3 块，一个 L3 块中可以存放多个 L2 块的地址，一个 L2 块中可以存放多个 L1 块地址，一个 L1 块中可以存放多个数据块地址，如图 1-2 所示。

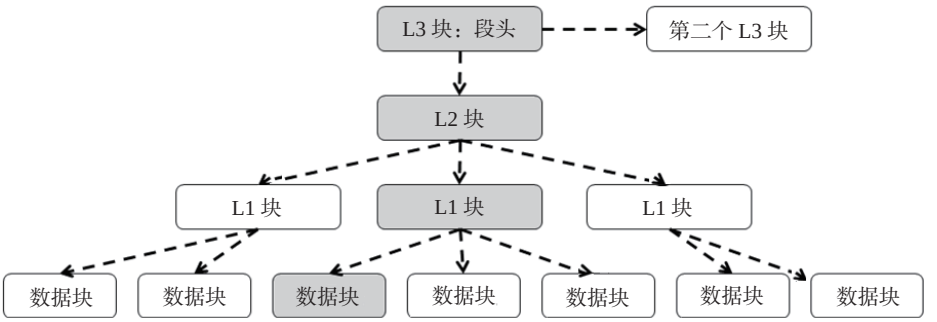

图 1-2　ASSM 的整体结构

第一个 L3 块一般是段头。如果段头中存放了太多 L2 块的信息，空间不足，Oracle 会再分配第二个 L3 块。当然，段头中会有第二个 L3 块的地址。如果第二个 L3 块空间也用完了，会再分配第三个。第二个 L3 块中会存放第三个 L3 块的地址。通常情况下，一个 L3 块就够了。有两个 L3 块就已经是非常罕有的情况了，基本上不会出现需要 3 个 L3 块的情况。

Oracle 是如何使用 4 层树状结构（3 层位图块 + 数据块）来确定向哪个块中插入的呢?

第一步，查找数据字典（就是 dba_segments 数据字典视图的基表），确定段头位置。

第二步，在段头中找到第一个 L2 块位置信息。

第三步，到 L2 块中根据执行插入操作进程的 PID 号，做 HASH 运算，得到一个随机数 N，在 L2 中，找到第 N 个 L1 块的位置信息。

第四步，到第三步中确定的 L1 块中，再根据执行插入操作进程的 PID 号，做 HASH 运算，得到一个随机数 M，在 L1 中找到第 M 号数据块。

第五步，向第 M 号数据块中插入。

L3 块中虽然可以有多个 L2 块，但插入操作不会选择多个 L2 块，每次只会选择同一个 L2 块。直到这个 L2 块下面的所有数据块都被插满了，才会选择下一个 L2 块。

在 L2 中选择某个 L1 的时候，就是随机的了。不同 Session，只要有可能，就会被分配到不同的 L1 中。在 L1 中找数据块时也是一样。

现在我们可以回答这个问题了：Oracle 为什么宣称 ASSM 可以支持大并发插入。

假设一个 L2 中有 100 个 L1，每个 L1 中有 64 个数据块，可以算一下，100×64，如果 Oracle 的随机算法真的够随机，如果有 6400 个进程一起执行插入操作，Oracle 会随机地将它们分配到 6400 个数据块中。

Oracle 的随机算法一向都是值得信赖的。

所以，在 Oracle 的所有资料中，都宣称 ASSM 可以支撑大并发插入。

但实际情况往往不像想象中的这么简单。

1.2.4 值得注意的案例：ASSM 真的能提高插入并发量吗

这个案例很有代表性，如果不深入到细节中，很容易在中途得出错误的结论。下面详细描述思考过程，希望能给大家带来些启发。

曾经遇到过这样的应用，要求对用户的登录、退出行为做记录。此部分的逻辑很简单，用户每登录一次应用，向数据库中一个日志表中插入一行，退出应用的时候再向日志表中插入一行。

此日志表是个日分区表，每天一个分区。每天大约会插入千万行，除了插入并发很高以外，就没有其他的大并发操作。另外，每天晚上会将当天的数据推送到数据仓库，在数据仓库中再进行分析、对比。

项目上线后，有些用户反映登录变慢了。而且，只有上午八九点钟左右的时候慢，过了这一段时间就没有用户反映有问题。经过对比 AWR，发现变慢是不定时的，从 8 点开始，到 9 点左右为止，在半小时一次的报告中，偶尔会有那么一两份 AWR 会显示 Buffer Busy Waits 比较高，然后就正常了。

看到这个情况，很容易让人认为是某个时间段有很多人一起在访问同一张表，其他时间又不一起访问了。究竟是不是这么回事呢？

先来确定一下等待是针对哪个对象。通过 V$SEGMENT_STATISTICS，查找 STATISTIC_NAME 列为 buffer busy waits 的，或者，查看 V$ACTIVE_SESSION_HISTORY 中的历史等待事件，根据 P1、P2 列的值，就可以定位争用是针对哪个对象的。

根据文件号、块号查找的结果来看，绝大多数的 Buffer Busy Waits 都出现在日志表上。

日志表每天分区的数据量最高接近千万行，就按每天 1000 万行算，除以 3600 × 24，平均每秒 116 个并发插入。当然，还要考虑高低峰的问题，晚上应用基本上没什么人用的，这几百万行大部分都是白天插入的。所以，再乘个 2，每秒 232 的插入量，这是最高的了。也并不是很多，这点量和 Oracle 宣称的 ASSM 支持的高并发插入相比，应该不会有 Buffer Busy Waits。

但无论如何，Buffer Busy Waits 是产生了，有可能以主机的硬件来论，现在已经是并发插入量的极限了。但奇怪的是，这种情况每天只会在刚上班后不久（8 ~ 9 点）出现，其他时段正常。

难道是刚上班时向日志表的插入量高？

但统计的结果显示，白天有好几个时段，日志表的插入量都很大，并不是早上上班时段特别大，有时下午还会比上午插入的稍多些，但没有发现下午日志表上有 Buffer Busy Waits，下午也从来没人反映过慢，而且整库的压力上下午基本差不多。

如果全天都有 Buffer Busy Waits，我想我也会放弃进一步调查。但有时下午的插入量多，反而没有等待。那说明 ASSM 是足以支撑这个量级的并发插入的。想解决问题的话，第一步是定位问题，这我们都知道。可如何定位这个问题呢？

遇到这样的疑难杂症，一般的方法是在测试环境中详细地分析相关操作，甚至可以使用 DTrace 加 MDB/GDB 这种底层分析工具。总之，只有清楚地了解底层操作，才能分析出问题在哪儿。

如何发现现在遇到的这个问题出在哪儿呢?

很简单，还是从最基本的测试做起。先建一个表，验证一下 Oracle 插入时，是否会随机地选择块。如下所示：

```
SQL> drop tablespace tbs_ts1 INCLUDING CONTENTS;

Tablespace dropped.

SQL> create tablespace tbs_ts1 datafile '/u01/Disk1/tbs_ts1_01.dbf' size 50m reuse
uniform size 1m;

Tablespace created.

SQL> create table table1(id int,name varchar2(20)) tablespace tbs_ts1;

Table created.
```

由于线上环境表空间区大小是 1MB，因此在测试环境，我也创建了个区大小为 1MB 的表空间。

在 Oracle 10g 以后，Oracle 默认的表空间类型就是 ASSM 了，所以，不需要专门指定了。

接着，在 tbs_ts1 表空间中创建一个测试表 TABLE1，下面来看看它的区占用情况。

```
SQL> select extent_id, file_id, block_id, blocks from dba_extents where
owner='LHB' and segment_name='TABLE1' order by extent_id;

 EXTENT_ID    FILE_ID   BLOCK_ID     BLOCKS
---------- ---------- ---------- ----------
         0          4        128        128
```

可以看到，TABLE1 在 4 号文件中，第一个区开始自 128 号块处。可以 DUMP 一下 128 号块看看，它是一个 L1 块。129 号块也是一个 L1 块，130 号块是 L2 块，131 号块是段头，也是 L3 块。

128 号和 129 号块中，各自有 64 个数据块信息。这一点，可以通过 DUMP 来确认。

下面，插入一行，试试看这一行将被插入哪个块中。

```
SQL> insert into table1 values(1,'AAAAAA');

1 row created.

SQL> commit;

Commit complete.
```

```
SQL> select dbms_rowid.ROWID_RELATIVE_FNO(rowid) fno,dbms_rowid.rowid_block_
number(rowid) block_id,dbms_rowid.ROWID_ROW_NUMBER(rowid) row_id,id from lhb.table1;

       FNO   BLOCK_ID     ROW_ID         ID
---------- ---------- ---------- ----------
         4        155          0          1
```

在插入这一行并提交后，可以用之前介绍过的语句，查看这一行的位置。可以看到，它被插入在了 4 号文件 155 号块中。换个会话再插入一行试试。

```
SQL> insert into table1 values(2,'BBBBBB');

1 row created.

SQL> commit;

Commit complete.

SQL> select dbms_rowid.ROWID_RELATIVE_FNO(rowid) fno,dbms_rowid.rowid_block_
number(rowid) block_id,dbms_rowid.ROWID_ROW_NUMBER(rowid) row_id,id from lhb.table1;

       FNO   BLOCK_ID     ROW_ID         ID
---------- ---------- ---------- ----------
         4        155          0          1
         4        156          0          2
```

在另一个会话中，插入了 ID 为 2 的行，它被插入在了 156 号块中。

不同的会话，Oracle 会将行插入到不同块中。Oracle 是根据 PID 计算出的随机数，随机地将行插入在不同的块中。只要 PID 不一样，行就会被插入在不同的块中。在 PID 一样的情况下，行会被插入在同一块中。

比如，在第一个会话中再插入一个 ID 为 3 的行。

```
SQL> insert into table1 values(3,'AAAAAA');

1 row created.

SQL> commit;

Commit complete.

SQL> select dbms_rowid.ROWID_RELATIVE_FNO(rowid) fno,dbms_rowid.rowid_block_
number(rowid) block_id,dbms_rowid.ROWID_ROW_NUMBER(rowid) row_id,id from lhb.table1;

       FNO   BLOCK_ID     ROW_ID         ID
---------- ---------- ---------- ----------
         4        155          0          1
         4        155          1          3
         4        156          0          2
```

ID 为 3 的行也被插入到 155 号块中。因为它和 ID 为 1 的行是在同一会话中插入的，会话对应进程的 PID 相同，两行就被插入了同一个块中。

另外，我们可以发现，后插入的 ID 为 3 的行，在显示时被排在先插入的 ID 为 2 的行前了。这说明堆表中行的排列也并非是插入顺序。

现在我们终于可以对前面提出的“堆表是有序的吗”问题给出一个明确的回复了。那就是：完全无序。因为插入时有个根据 PID 计算随机数的过程，这就会导致行被插入哪个块是随机的。因此，堆表是无序的。

继续观察行被插入的位置。但如果我们老是通过 sqlplus lhb/a 建立一个会话，在会话中插入，这样太麻烦了，还是写个脚本吧。

```
$ cat assm_test.sh
sqlplus lhb/a <<EOF
insert into lhb.table1 values($1,'aaabbbcccd');
commit;
exec dbms_lock.sleep(10000);
EOF
```

关于 Shell 脚本的编写，这里不再解释。下面只说一点，为什么最后要加如下语句：

```
exec dbms_lock.sleep(10000);
```

如果没有这个暂停操作，会话将立即结束。在 Oracle 中，如果前一个会话结束，下一个会话马上建立，则下一个会话将会有和前一个会话相同的 Session ID 和 PID（注意，PID 不是 SPID，PID 是 Oracle 对进程的编号）。如果两个会话的 PID 相同，行将被插入在同一块中。所以，这里专门加个“暂停”操作，让会话停 10000 秒后再退出。这样，再新建一个会话，它将有一个新的 PID。

按如下方法，将上述脚本执行 10 次：

```
./assm_test.sh 4&
./assm_test.sh 5&
…
```

加一个 &，表示放在后台执行，要不然要等 10000 秒才能结束。

查看一下这些行都被插到哪儿了。

```
SQL> select dbms_rowid.ROWID_RELATIVE_FNO(rowid) fno,dbms_rowid.rowid_block_number
(rowid) block_id,dbms_rowid.ROWID_ROW_NUMBER(rowid) row_id,id from lhb.table1;

       FNO   BLOCK_ID     ROW_ID         ID
---------- ---------- ---------- ----------
         4        155          0          1
         4        155          1          3
         4        156          0          2
         4        157          0          4
         4        158          0          5
```

```
         4         159           0           6
         4         160           0           8
         4         161           0           7
         4         162           0           9
         4         163           0          10
         4         164           0          11
         4         165           0          13
         4         166           0          12

13 rows selected.
```

还是很平均的，每个块一行。我们看一下 ROW_ID 列，这是行在块中的行号。除了刚才做测试的 ID 为 3 的行，其他行都是块中的第一行（行号为 0）。

平均是很平均，但我们应该也注意到了一个问题，在后面所做的 10 次插入，虽然这 10 行的确被插到了 10 个块中，但是，这些块未免有点太集中了。

table1 表现在共有 128 个块，块编号从 128 到 255。但这些行都被插到了 155 ~ 166 号块中。

这应该是 Oracle 的算法不够随机吧。

一开始我觉得，是区不够多，只有一个区，128 个块，Oracle 选择面太窄了。我们知道，表在扩展时，也都是一个区一个区地扩展的。每次占满了 128 个块后，再扩展下一个区。但下一个区也还是 128 块，还是只在 128 个块中选择。由于随机算法不够随机，导致在 128 选一时，很多行被同时插到了同一个块中，这时，就会出现 Buffer Busy Waits。

一切都是合乎情理，我马上将发现告知应用方。解决方案就是，在晚上数据库空闲时，为日志表手动分很多个区。

第二天，客户依然反映，运行速度慢。查看数据库，还是有 Buffer Busy Waits。

为什么？

看来是第一次的实验做得不够彻底。为什么使用的块是 155 号、156 号、157 号等，这么有顺序，而且不够分散呢？

继续前面的测试。这次，我调用 ./assm_test.sh N&，每 10 次观察一下行的分配情况，终于发现了问题。

```
SQL> select dbms_rowid.ROWID_RELATIVE_FNO(rowid) fno,dbms_rowid.rowid_block_number
(rowid) block_id,dbms_rowid.ROWID_ROW_NUMBER(rowid) row_id,id from lhb.table1;


       FNO   BLOCK_ID     ROW_ID         ID
---------- ---------- ---------- ----------
         4        132          0         39
           ..........................
         4        159          0          6
         4        159          1         67
         4        160          0          8
```

```
4         160          1          66
4         161          0           7
4         161          1          68
4         162          0           9
4         162          1          69
  ....................
4         190          0          38
4         191          0          37

69 rows selected.
```

一共插入了 69 行，最小的块号是 132。这个可以理解，因为表的第一个块编号是 128，128 号、129 号块是 L1，130 号是 L2，131 号是段头兼 L3。第一个可用数据块是从 132 开始的。但是到 150 号块后，就开始有重复，两行被插入同一块中。还有一点就是，最大使用的块是 191 号。用 192−128，正好等于 64。

继续分析下去有个关键点，要看之前对细节的挖掘程度了。前面我们一再地提过，对于 1MB 大小的区，每个区最前面的两个块，大多数情况下是 L1 块。在 8KB 的块大小下，1MB 共 128 个块，两个 L1，正好每个 L1 记录 64 个数据块。

好了，答案基本上已经浮出水面。

Oracle 只使用了第一个 L1 块中的数据块，而没有使用第二个 L1 中的块。

其实还有一个知识点，如果不具备，可能分析就到这里为止了。前面也提过了，Oracle 在 L3、L2、L1、数据块中这个树状图中选择要插入的块时，从 L3 中选择 L2 并不是随机的，每次都只选某一个。但从 L2 中选择 L1 是随机的。关于这一点，我已经做了测试。

现在 L2 中有两个 L1，会什么 Oracle 只选择第一个 L1 呢？

你想到原因了吗？

我是这样想到原因的，我曾经做过直接路径插入的测试，这个测试验证了如果进行直接路径插入，每次会在高水点之上分配空间，如果提交，则修改高水点。如果不提交，则不修改高水点，通过这种方式可减少 UNDO 的耗用。而普通的插入则是在高水点之下寻找空间。

我们一直没有提过高水点。直接路径都是在高水点之上插入的，那么间接路径呢？肯定是在高水点之下了。

好，答案已经见分晓了。高水点肯定在第 192 号块。因为第二个 L1 块中的数据块，都在高水点之上，因此，第二个 L1 块中的数据块不会被插入算法选择到。

DUMP 一下段头验证一下吧。

```
  Extent Control Header
-----------------------------------------------------------------
Extent Header:: spare1: 0      spare2: 0     #extents: 1      #blocks: 128
last map        0x00000000     #maps: 0      offset: 2716
Highwater::     0x010000c0      ext#: 0      blk#: 64         ext size: 128
```

代码中加下划线的就是高水点了。0x010000c0，这个是 DBA（Data Block Address，数据块地址）。它的前 10 个二进制位是文件号，后面的是块号。0x010000c0 也就是 4 号文件 192 号块。

看来 Oracle 的高水点每次向后移动时，是以 L1 块中的数据块数量为单位的啊。

水落石出了，原来是高水点太低的问题。

Oracle 只告诉我们，ASSM 可以增大插入并发量，但没告诉我们，并发插入量还要受高水点限制。

以前曾经有人讨论过在 MSSM 表空间中高水点的移动规则，而 ASSM 下高水点的推移规则还很少有人注意过。

当在区中插入第一行时，高水点移到区的第一个 L1 块中最大的数据块后。这句话有点绕，还是以我们的测试表 TABLE1 为例吧：插入第一行时，高水点移到了第一个 L1 块（128 号块）中最大的数据块后，128 号块中最大的数据块是 191，那么高水点就是 192 了，其实也就是第二个 L1 块中的第一个数据块。

简单总结一下，高水点的移动，在 ASSM 下，是以 L1 中数据块的数量为准的。

如果块大小是 8KB，区大小是 1MB，L1 中有 64 个数据块，高水点就是以 64 个块为单位，依次往后挪的。也就是说，我们的并发插入，每次都只是向 64 个块中插入。可以想象，如果同时有 100 个进程插入，但只有 64 个块接收，将有 36 个进程不得不和另一个进程同时向一个块中插入。

两个进程同时修改一个块，会有什么等待时间呢？ Buffer Busy Waits（当然也会有少量的 Cache Buffer Chain Latch）。

问题已经找到一大半了，ASSM 表空间仍有可能因为高水点不高，可用于插入的块不多，造成 Buffer Busy Waits。但另一半问题隐藏得更深，为什么只会在刚上班那会儿出现这个等待，而其他时间则没有呢？

注意，白天的时候，压力是差不多的。有时下午比上午还要高。

要解答这个问题，就看你对 Oracle 的内部机制有多大的好奇心了。

我挖掘出这个问题纯属意外。

其实在发现了高水点问题后，我建议使用抬高高水点的方式解决争用问题。

当然，抬高高水点后，将对全表扫描不利。全表扫描只扫描高水点之下的块，如果高水点太高，要扫描的块也多了。

但这个日志型应用，平常没有全表扫描，只有在每天晚上向数据仓库传数据时，需要全表扫描。因此，对全表扫描的影响不是主要考虑的因素。

如何抬高呢？手动分配区是无法抬高高水点的。只有一种方法，先插入行再删除。

因为日志表是一个日分区表，按照日期，每天一个分区。考虑到每天的插入量不会高于 1000 万行，因此决定对未来的每个分区，先插入 1000 万行，再用 Delete 删除。

具体的方案是这样的，先使用 APPEND 向一张中间表中插入 1000 万行，采用直接路

径方式，这样产生的 UNDO 量较少。再用 Delete 慢慢删除，根据 ROWID 来删除，一次删除一个区的所有行，然后提交。将整个表删除完后，高水点就已经被抬高了，但表中是没有行的。再使用分区交换命令，将被抬高高水点的中间表交换到日志表中。

这种方法听起来有点不太规范，但没办法，暂时只能这样解决了。

实事上，我用上面的方式调高了几个分区的高水点，第二天观察，果然在全天任意时候，都不再有 Buffer Busy Waits 了。

其实如此交差也可以，就是加分区的时候麻烦点。若用脚本实现，只是在 Delete 的时候慢点，不占太多回滚段就不会有任何问题。

但还有一个问题一直困扰着我，但这个问题和应用已经无关了，我只是好奇：一个 L1 中有 64 个数据块，64 这个数字是固定的吗？

我分别用 40KB（5 个 8K 的块，已经是 Oracle 中最小的区了）、1MB、10MB、30MB 大小的区测试，40KB 的区中，一个 L1 中可以只有 5 个数据块，是最少的。但 1MB、10MB、30MB 的区，都是一个 L1 中有 64 个块。64 个块应该就是 L1 中数据块的最大数量了。

Oracle 的系统管理区大小是随着段的不断变大而不断变大的，L1 会不会也是这样呢？我决定再试一下。

用手动分配区的命令，为 TABLE1 多分配些区。我为 TABLE1 每次分配 30MB 空间，每次 DUMP 一下最后一个区的第一个块（每个区第一个块通常都是 L1 块）。

当分配的总空间到 90MB 时，我发现 L1 中的数据块数量从 64 增加到了 256 个。测试如下：

```
SQL> drop tablespace tbs_ts1 INCLUDING CONTENTS;

Tablespace dropped.

SQL> create tablespace tbs_ts1 datafile '/u01/Disk1/tbs_ts1_01.dbf' size 100m
 =reuse uniform size 1m;

Tablespace created.

SQL> create table table1(id int,name varchar2(20)) tablespace tbs_ts1;

Table created.

SQL> alter table table1 allocate extent (size 90m);

Table altered.

SQL> set pagesize 1000
SQL> select extent_id, file_id, block_id, blocks from dba_extents where
owner='LHB' and segment_name='TABLE1' order by extent_id;

 EXTENT_ID    FILE_ID   BLOCK_ID     BLOCKS
```

```
---------- ---------- ---------- ----------
         0          4        128        128
         1          4        256        128
         2          4        384        128
..............................
        88          4      11392        128
        89          4      11520        128
        90          4      11648        128

91 rows selected.
```

上面删除了表空间，重新建了个全空的，区大小 1MB。又创建了个新表，TABLE1，手动分配 90MB 空间。它一共有 91 个区。

分别 DUMP 一下第 128 号块和 11520 号块。以下是 4 号文件 128 号块的 DUMP 结果：

```
mapblk  0x00000000  offset: 0
  -------------------------------------------------------
  DBA Ranges :
  -------------------------------------------------------
0x01000080  Length: 64     Offset: 0

   0:Metadata         1:Metadata        2:Metadata        3:Metadata
   4:unformatted      5:unformatted     6:unformatted     7:unformatted
   8:unformatted      9:unformatted    10:unformatted      ...............
    ......................
   60:unformatted    61:unformatted    62:unformatted    63:unformatted
  -------------------------------------------------------
```

可以看到，这个 L1 中共有 64 个数据块。以下是 11520 号块的 DUMP 结果：

```
  -------------------------------------------------------
  DBA Ranges :
  -------------------------------------------------------
0x01002d00  Length: 128    Offset: 0
0x01002d80  Length: 128    Offset: 128

   0:Metadata           1:unformatted     2:unformatted     3:unformatted
   4:unformatted        5:unformatted     6:unformatted     7:unformatted
..............................
   252:unformatted   253:unformatted   254:unformatted   255:unformatted
  -------------------------------------------------------
```

在这个 L1 块中，数据块的数量增加到了 256 个。

这证明了 L1 块中记录的数据块个数也是随着表的不断增大而增多的。

这个证明有何意义呢？还记得上面遇到的问题吧，每天总是在刚上班时会有 Buffer Busy Waits，而其他时间则没有。现在有答案了。

因为日志表每天一个分区，也就是每天一个段。刚上班时，段还比较小，L1 块中只有 64 个数据块，因此并发插入每次都只针对 64 个块。随着表增大，当表超过 90MB 时，一

个 L1 就有 256 个数据块了，即使所有并发都只针对一个 L1 中的数据块，256 个块也足以支撑这套应用的所有并发了。因此，每天总是在最开始不长一段的时间内，会有 Buffer Busy Waits，再往后就正常了。

这个奇怪的问题终于找到了原因。其实我研究 L1 中数据块的数量，本来只是为了满足好奇心，没想到可以查找出这个问题的原因。

更进一步，可以再试一下不同区大小、不同段大小下，L1 块中数据块的数量。

我测试的结果是，10MB 区大小，从第 4 个区开始，L1 块中数据块的数量就已经是 256 个了。10MB 的区好像有点大了，我只测试了一下 4MB 或 8MB 的区，在段大小超过 64MB 后，L1 块中数据块的数量会达到 256 个。

好，研究得差不多了。可问题该如何解决呢？方法还和刚才一样，先插入，再删除。只不过，不需要插入 1000 万行了。我选择建立 8MB 区大小的表空间，日志表新的分区都建到新表空间中。每个分区只需插入 50 万行再删除就可以了。

只需要将前 8 个分区，插入满行，再删除，将高水点推到第 8 个分区后，因为第 8 个分区后，每个 L1 块中都是 256 个数据块，足够支撑并发插入量了。

该问题终于有了一个比较好的解决方案。但后面经过观察又发现，在 L1 块上出现了争用，但不严重，没有造成反应延迟。Oracle 的高水点每次以 L1 块中数据块的量为单位向后扩，始终会有问题。如果同一时刻的并发超过了 256 个，一样会有争用，而且，这么大的量，L1 块的竞争也会大大加剧。这样的话，解决方法只有一个，就是像我最初的方案一样，插入很多行（比如 1000 万行），将高水点拉得很高，再删除。

好了，ASSM 的问题就说到这儿。看来随便建个 ASSM 表空间，再建个表上去，就想支撑大并发插入，这种想法有点简单了。

实际案例就先介绍到这儿。希望通过这个案例读者能有所收获。

补充一句：对 Oracle 越熟悉，面临的疑难杂症就越少。

关于表空间和存储结构，还有两个疑问：全表扫描时，Oracle 是如何找到表的块在哪儿的？索引扫描 Oracle 是如何找到 Root 块的？

1.2.5　段头与 Extent Map

上一节提到了，段头是第一个 L3 块，就是说段头中包含 L3 信息。其实，段头中的重要信息，除了 L3 外，还有 Extent Map，将其直译过来就是区地图。

顾名思义，区地图就是记录一个段中所有区都在哪儿的地图。全表扫描操作，就是按图索骥，按区地图逐个读取所有区。

让我们来看看区地图是什么样子，同时，也模拟一下全表扫描的执行流程。

第一步，确定段头位置。

```
SQL> select header_file ,header_block from dba_segments where segment_name='TABLE1';

HEADER_FILE HEADER_BLOCK
```

```
---------- ------------
         4          131
```

当然，Oracle 肯定不会读 dba_segments 这个数据字典视图，它会读 dba_segments 低层 seg$ 这样的数据字典表。会先到共享池中的字典缓存中查找 seg$ 相关的行，如果没有找到，再到 Buffer Cache 中读 seg$ 相关的块，如果还没有，就到磁盘上 SYSTEM 表空间中读 seg$ 表。

当找到 TABLE1 的段头位置时，Oracle 会读取它里面的区地图，我们来 DUMP 一下。

执行下面的命令 DUMP：

```
exit
sqlplus / as sysdba
alter system dump datafile 4 block 131;
```

就是先退出 sqlplus，再重新连接，然后去 DUMP。因为同一服务器进程会把 DUMP 信息写到一个 DUMP 文件中。如果你 DUMP 多次，被会写进一个文件，这样观察起来不方便。我退出再登录，服务器进程会换一个的，SPID 也会不同，这样 DUMP 信息会被写到不同的文件中，便于查看。

下面就是段头中的区地图信息：

```
  Extent Map
  -----------------------------------------------------------------
0x01000080  length: 128
0x01000100  length: 128
0x01000180  length: 128
.............
0x01002d00  length: 128
0x01002d80  length: 128
```

第一个区，开始自 0x01000080 处，前 10 个二进制位是文件号，后面是块号，前面已经提到过的，也就是 4 号文件 128 号块处。这个区的大小是 128 个块，最后一个区，开始自 4 号文件 11648 号块处（就是最后一行 0x01002d80），大小也是 128 个块。

我们已经看到区地图了，很简单是吧？但全表扫描时 Oracle 读取的并不是这里的区地图，还要往下看。

```
Auxillary Map
-------------------------------------------------------
 Extent 0     :  L1 dba:  0x01000080 Data dba:  0x01000084
 Extent 1     :  L1 dba:  0x01000100 Data dba:  0x01000102
 Extent 2     :  L1 dba:  0x01000180 Data dba:  0x01000182
            ..................
 Extent 89    :  L1 dba:  0x01002d00 Data dba:  0x01002d01
 Extent 90    :  L1 dba:  0x01002d00 Data dba:  0x01002d80
-------------------------------------------------------
```

在上面的信息中，出现了 Auxillary Map，直译过来是辅助地图。这一部分信息更详细。L1 dba: 0x01000080，说明了此区内第一个 L1 块开始的地方，即 4 号文件的 128 号块。Data dba: 0x01000084，说明用户数据开始的地方，即 132 号块。这里说明了真正的用户数据开始自哪里，Oracle 全扫描时，是按照“Data dba：*******”后的 DBA 查找区的。但这里没有区长度，所以，上面那部分区地图信息还是要读的。

另外，我们看最后两行：

```
Extent 89    :  L1 dba:  0x01002d00 Data dba:  0x01002d01
Extent 90    :  L1 dba:  0x01002d00 Data dba:  0x01002d80
```

这两行的 L1 Dba 一样，都是 0x01002d00，即 4 号文件 11520 号块。为什么这样？因为 11520 号块中有 256 个数据块，所以这两个区只需要有一个 L1 块就行了。可以观察一下从什么地方开始两个区只要一个 L1 块，这里是从 8192 号块开始的。

```
Extent 61    :  L1 dba:  0x01001f00 Data dba:  0x01001f02
Extent 62    :  L1 dba:  0x01001f80 Data dba:  0x01001f82
Extent 63    :  L1 dba:  0x01002000 Data dba:  0x01002001
Extent 64    :  L1 dba:  0x01002000 Data dba:  0x01002080
```

可以看到，61 号、62 号区，还各自有不同的 L1 号块，而 63 号、64 号区，已经只有 63 区头的一个 L1 块了。63 号区也就是第 64 个区，每个区 1MB，也就是当段大小超过 64MB 时，一个 L1 将放存 256 个数据块。

好了，这就是区地图，通过研究它，全表扫描操作的流程我们应该也都清楚了。很简单，找到段头，读取区地图信息，根据区地图的顺序，读取每一个区。所以，全表扫描的显示顺序，就是区地图中区的顺序，其实也就是 dba_extents 中区的顺序。

下面再来看一下全表扫描的逻辑读。

```
SQL> drop tablespace tbs_ts2 INCLUDING CONTENTS;
Tablespace dropped.

SQL> create tablespace tbs_ts2 datafile '/u01/Disk1/tbs_ts2_01.dbf' size 20m reuse
uniform size 40k;
Tablespace created.

SQL> drop table table2;
Table dropped.

SQL> create table table2(id int,name varchar2(20)) tablespace tbs_ts2;
Table created.

SQL> insert into table2 values(1,'ABC');
1 row created.

SQL> commit;
Commit complete
```

```
SQL> set autot trace
SQL> select * from table2;

.....
Statistics
----------------------------------------------------------
0  recursive calls
0  db block gets
4  consistent gets
0  physical reads
0  redo size
594  bytes sent via SQL*Net to client
520  bytes received via SQL*Net from client
2  SQL*Net roundtrips to/from client
0  sorts (memory)
0  sorts (disk)
1  rows processed
```

注意，做观察逻辑读的测试时，对测试 SQL 语句 select * from table2，要反复多次执行。

这里重新创建了一个 TBS_TS2 表空间，它的区大小只有 40KB，也就是 5 个块。然后建了一个表，随便插入一行，插入的这一行将会使高水点被抬升到区的最后一个块之后。

这个区只有 5 个块，前三个块分别是 L1、L2 和段头，可以存放用户数据的只有第 4、5 两个块，那么高水点将在第 5 个块之后。

为什么逻辑读是 4 次呢？全表扫描，要跳过 L1、L2，只读段头和高水点下的所有块，也就是读段头和第 4、5 个块。但是段头要读两次，所以，逻辑读为 4。至于段头读两次的原因，根据前面 DUMP 的段头来看，段头中的 Extent Map、Auxillary Map 信息是分开存放的，要一次读 Extent Map，一次读 Auxillary Map，所以就要读两次了。

如何确定段头读两次的问题呢？Oracle 10G 以前的版本，可以观察 Latch 的 Gets 次数，但在 11GR2 后，就只有使用 DTrace 跟踪才能知道了。本书后面章节会有些这方面的内容，我们会逐步深入到 Oracle 内部，揭开 Oracle 之谜。

1.2.6 索引范围扫描的操作流程

索引范围扫描，网上已经有很多讨论了，就是按照根、枝、叶的顺序读取。叶块的地址在枝块，枝块地址在根块。找到枝块就可以找到叶块，找到根块就可以找到枝块。那么，如何找到根块呢？

其实很简单，在 Oracle 中，根块永远在索引段头的下一个块处。因此，索引扫描是不必读取索引段头的。先在数据字典表中找到段头位置，块号加 1 就是根块位置了。

对索引范围扫描时的逻辑读，可以做如下测试：

```
SQL> insert into table1 select rownum,'abcde' from dba_objects;
12691 rows created.
```

```
SQL> commit;
Commit complete.

SQL> create index table1_id on table1(id) tablespace tbs_ts1;
Index created.

SQL> exec dbms_stats.gather_table_stats('LHB','TABLE1');
PL/SQL procedure successfully completed.

SQL> select BLEVEL from dba_INDEXES where index_name='TABLE1_ID' and owner='LHB';

    BLEVEL
----------
         1
```

上面先向表中插入了 10000 多行，再创建了一个 1 层高的索引，索引只有 Root 块和叶块。下面看看索引访问一次的逻辑读：

```
SQL> set autot trace
SQL> select * from table1 where id=10;
Statistics
----------------------------------------------------------
0  recursive calls
0  db block gets
4  consistent gets
0  physical reads
0  redo size
596  bytes sent via SQL*Net to client
520  bytes received via SQL*Net from client
2  SQL*Net roundtrips to/from client
0  sorts (memory)
0  sorts (disk)
1  rows processed
```

将测试 SQL 语句 select * from table1 where id=10 多执行几次，观察到的逻辑读为 4。这 4 次逻辑读分别是：Root 块一次，叶块两次，数据块一次。

叶块之所以需要两次，是因为索引是非唯一的。第一次读叶块是为了取出目标行 ROWID，第二次读叶块是判断此叶块中还有没有满足条件的行。

如果建成了唯一索引，不需要判断叶块是否还有满足条件的行，叶块就只需要读一次，一共只需要 3 次逻辑读。

```
SQL> drop index table1_id ;
Index dropped.

SQL> create UNIQUE index table1_id on table1(id) tablespace tbs_ts1;
Index created.

SQL> set autot trace
SQL> select * from table1 where id=10;
```

```
Statistics
----------------------------------------------------------
0  recursive calls
0  db block gets
3  consistent gets
0  physical reads
0  redo size
460  bytes sent via SQL*Net to client
509  bytes received via SQL*Net from client
1  SQL*Net roundtrips to/from client
0  sorts (memory)
0  sorts (disk)
1  rows processed
```

表空间和存储结构这就说到这儿。本章中的例子，都是在 Oracle 11GR2 中做的，在 Oracle 10g 中做同样例子时的注意事项也都随例子说明了。

另外，本章的测试都是以 8KB 块大小为例的，其他块大小下的情况，留给读者亲自动手测试。

第 2 章 Chapter 2

调优排故方法论

一旦 Oracle 出现问题，DBA 会忙碌起来了。一般情况下，他们会从几个固定的地方入手，查找、确定问题。但要真正地挖掘问题原因，特别是那些隐藏得很深的问题，不了解工作原理是不行的。

如果在你骑单车锻炼身体时，骑到半路上车不动了，如何检查单车出了什么问题？首先，我们要知道单车之所以可以动起来，是因为转动脚踏板时带动了轮盘，轮盘上的链子又带动了后轮上的轮盘，这个轮盘将带动后车轮旋转，而后车轮的旋转将使整个车向前运动。知道了单车的工作原理，我们就可以检查单车为什么不走了。比如，首先转动脚踏板，测试一下轮盘是否带动链子运动。如果没有，说明轮盘和链子的结合出了问题。如果没问题，继续观察链子的运动，是否带动后轮盘……如此追根溯源，最终一定能找出问题所在。

对应到 Oracle，了解原理同样重要。以 SQL 语句执行为例，总的来讲，语句执行过程分三大步骤：解析、执行和抓取。解析又分硬解析、软解析。硬解析要生成执行计划，并将执行计划传到共享池中。软解析直接在共享池中取出执行计划就可以了。

假设数据库出现了许多 Shared Pool Latch 竞争，会是哪个环节出问题了呢？

如果我们对 Oracle 原理有基本了解，就能得出判断。Shared Pool Latch 的申请主要出现在进程从共享池中分配内存时。再来分析 SQL 语句的执行过程，在解析、执行、抓取，还有软解析、硬解析的过程中，什么时候进程需要从共享池分配内存呢？

执行的时候不会，抓取的时候也不会。抓取主要是从磁盘或 Buffer Cache 中读数据，与共享池关系不大。再看解析呢？软解析不会，软解析只是查找、读取共享池中的执行计划。而硬解析需要将新生成的执行计划存入共享池。在存入共享池前，肯定先要在共享池中分配一块内存，然后才能将信息存进去。

如果我们对原理有所了解，对于 Shared Pool Latch 的竞争，很快就能分析出来原因：只有在硬解析时，才会从共享池分配内存，因此，此 Latch 竞争很有可能是过多的硬解析造成的。

但除了硬解析，还有其他情况也会持有 Shared Pool Latch，比如自动调节内存时，如果启用了内存自动调节，当 Oracle 觉得共享池内存不足或太多时，也会持有 Shared Pool Latch、调节共享池内存大小，有时这也会导致 Shared Pool Latch 的竞争。如果忽视了这一块，在诊断 Shared Pool Latch 问题时，将所有问题都归为硬解析，就有可能会误判方向。

这就是 Oracle 问题的诊断思路，即按照 Oracle 工作原理，判断问题出在哪个环节。如果对工作原理了解不全面，问题的判断就可能偏离方向。

2.1 调优排故的一般步骤

总的来说，调优、排故大都是从原理的角度分析问题可能出现在哪个环节。但 Oracle 不是单车，其原理庞杂，单是 Oracle 编译过的可执行文件，在 Oracle 11g 中就已经达到 250MB 左右，在 Oracle 最新版 12C 中，可执行文件大小已达 340MB 左右（在不同操作系统下，稍微会有些差异），自定义函数有十几万个。

这么庞大的软件，在实际遇到问题时，不可能从头到尾把所有原理分析一遍。事实上，Oracle 为我们提供了很多工具和信息，可以帮助确定问题的大概方向。

其实，Oracle 在各种资料（包括联机文档）中多次宣称 Oracle 是一个可观测、可调节的数据库。Oracle 不单有 DBA_ 系列视图，可以查阅“数据定义”类元数据信息，还有丰富的 V$ 系列视图，可以查阅 SGA 中的状态。如果这还不够，V$ 的底层 X$ 还提供了更详细的信息，另外还有各种各样的跟踪事件、DUMP 命令。善于使用这些工具，将使你能体会“发现”的乐趣。利用这些工具，一步步地挖掘 Oracle 工作原理，将会拨开网络上各种信息的“迷雾”，一步步接近真相。正如我一直宣称的，研究 Internal，结果不是目的，过程更有意义。在这个过程中，你会收获很多。

下面，把最重要、最常用的工具介绍一下。

2.1.1 常见 DUMP 和 Trace 文件介绍

Oracle 的问题通常可以分为两大类：性能问题和故障。性能问题通常都是对用户的反应慢，而故障多指异常的宕机或其他数据库的异常情况。

如果你遇到了故障，DUMP 文件和 Trace 文件将成为你调查故障的最佳入口。

按照存放位置不同，DUMP、Trace 文件共分 3 类：后台进程 DUMP 文件、核心 DUMP 文件和用户 DUMP 文件。它们分别对应参数 background_dump_dest、core_dump_dest、user_dump_dest 所在的位置。在 Oracle 11g 之后，这 3 个参数通常是一个位置。

其中最重要的就是 background_dump_dest 中的告警日志文件了。

在阿里巴巴，Oracle 数据库的所有监控中，最重要的一个就是将告警日志中大部分以 ORA- 开头的错误信息，以短信的方式发送到 DBA 的手机上。

至于这些文件的阅读示例，网络中有很多相关描述，本章不再论述。

2.1.2　等待事件

如果遇到故障，在分析完 DUMP、Trace 文件后，就得关注等待事件了。

而对于性能问题，等待事件将是我们判断问题首先要关注的。

其实 Oracle 称呼等待事件为 Event，也就是事件，并不一定发生等待时才有 Event。Event 目的是告诉我们 Oracle 此刻正在做什么的，不能单纯只把它看作等待。

比如，当你看到某个 Session 当前的 Event 是 db file sequential read 时，说明此 Session 对应的进程，正在完成物理 I/O，也就是正在从磁盘上读一些块。如果这个 Event 一共用了 15 毫秒，那就是说 Oracle 完成这批物理 I/O，用时 15 毫秒。

如果在过去的半小时中，数据库一共产生过 777375 次 db file sequential read 事件，这 70 多万次事件共占用时间 3308 秒，用 3308 除以 777375，可以得到每个事件的平均占用时间约 4 毫秒。这个值可以作为磁盘的随机 I/O 响应时间。也就是说，在过去的半小时内，Oracle 随机 I/O 响应时间为 4 毫秒。

这个值可以很方便地在 AWR 或老版的 Statspack 中查到。在评估存储是否正常时，这个值是很重要的值。从经验上来说，通常高端存储的随机 I/O 响应时间都可以控制在 10 毫秒以下，中端的控制在 10 毫秒左右。响应时间超过 20 毫秒的 I/O，一般认为是比较缓慢的 I/O。

I/O 响应时间慢一般有两种原因：存储问题或者 I/O 太多。这里不再对这一事件展开描述，在后面的章节中会详细讲述 I/O。

我一直觉得 Oracle 中的等待事件很奇妙，在 Oracle 11g 中，一共有 1118 个等待事件，几乎覆盖了 Oracle 工作的方方面面，它是如何实现为每个操作记录等待事件的呢？机制是怎样的呢？网络中关于这方面的资料很少。有一段时间我一直认为等待事件是超过一定时间就记录，但其实不是。经过用 gdb、mdb 等调试工具调试 Oracle 可执行文件发现，Oracle 中的事件，按照工作原理一共可以分为两类，一类是主动触发事件，另一类是被动触发事件。

比如前文一直提到的 db file sequential read 就是一个主动触发事件。Oracle 在完成一个 I/O 时，它知道 I/O 不会很快完成，于是会主动登记一个事件，然后再开始进行 I/O 操作。

所有 I/O 类相关的等待事件，包括网络 I/O，都是主动触发事件。除 db file sequential read 外，db file sequential read、direct path read、direct path write、log file parallel write 等这些与 I/O 相关的事件，都是主动触发的。

主动触发事件还有一个特点：只要发生一次 I/O，一定会对应一次 I/O 相关事件。仍以 db file sequential read 为例，无论 I/O 的完成速度有多快，在读 I/O 操作开始前，Oracle 都会登记一次 db file sequential read 事件。

Oracle 明知 I/O 操作会很慢，因此会主动登记一个等待事件，告诉用户“我在完成 I/O”。但有些动作，比如一个 Latch 的获取，会在极短的时间内完成。在获取一个 Latch 前，进程不会主动登记等待事件。只有当遇到阻塞、获取不到时，才会记录等待事件，这种就是被动事件。

除 I/O 相关、网络相关的等待之外的事件，基本上都是被动事件。

仍以 Latch 为例，再次强调一下被动事件的主要特点，如果没有遇到阻塞，哪怕 Latch 的获取过程非常慢，也不会有任何等待事件。

对各种事件以及其背后原理的深入挖掘是十分必要的。因为一旦遇到问题，Oracle 就会用事件告诉你此刻它正在做什么，如果你对事件不理解或者理解错了，就会错失解决问题的良机。

一般遇到性能问题时，通过查看事件即可解决，所以相对来说还是很简单的。但如果遇到故障，特别是异常宕机的故障，则很难知道最后时刻 Oracle 登记的事件是什么。如果能找到这个事件，对于诊断宕库类故障将很有意义，因为根据事件可以推测出 Oracle 最后时刻的动作。下面说一下如何挖掘宕库时数据库最后时刻的等待事件。

正常情况下，等待事件的查询可以通过 v$session、v$session_evnt、v$system_event、v$session_wait 等视图来查看，这里不再详述。对于异常故障，Oracle 通常会产生一些 DUMP 文件或 Trace 文件，有时，我们可以从中挖掘出等待事件。下面通过案例来介绍一种在异常情况下挖掘等待事件的方法。

首先查看告警日志文件。Oracle 生成的重要 Trace 文件会在告警日志文件中有记录。找到告警日志后，再在其中找 Trace 文件，因为在 Trace 文件中，通常可以找到 Call Stack Trace，也就是运行堆栈。

运行堆栈是当出问题时 Oracle 进程自身调用的函数信息。Oracle 开发人员在确诊代码 Bug 时，这部分函数内容是很重要的信息，通常我们在查看 Trace 文件时，会把这部分信息略去。但如果想找到数据库异常宕掉时的等待事件，这部分信息就不能略去。

还有一点要知道，Oracle 10g 的全部事件和 Oracle 11g 的大部分事件，都是用一个 Oracle 内部自定义函数 kslwait 登记的。在 Trace 文件的运行堆栈中，找一下是否有 kslwait 函数的调用，如果有，就能确定在数据库宕掉时，最后的等待事件是什么，或者说，进程发生异常时，等待事件是什么。下面我们通过一个案例了解一下这种情况。

首先，以 Oracle 11.2.0.4、Linux 为测试环境，深入了解一下等待事件的相关知识。

步骤 1：打开一个测试 Session，如图 2-1 所示。

测试会话 SID 为 140，进程号是 4718。

步骤 2：使用 gdb 调试它。

这一步的结果很多，分成两张图显示，如图 2-2 和图 2-3 所示。

从图 2-3 可以看到，gdb 命令运行成功后，最终会显示 gdb 的提示符 (gdb)。

```
[oracle@ocp admin]$ sqlplus lhb/a

SQL*Plus: Release 11.2.0.3.0 Production on Sat Dec 14 19:48

Copyright (c) 1982, 2011, Oracle.  All rights reserved.


Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.3.0 -
With the Partitioning, OLAP, Data Mining and Real Applicati

SQL>
SQL>select c.sid,spid,pid,a.SERIAL# from (select sid from v

       SID SPID                            PID    SERIAL#
---------- ------------------------ ---------- ----------
       140 4718                             23          9

SQL>
SQL>
```

图 2-1　打开测试 Session

```
[root@ocp ~]# su - oracle
[oracle@ocp ~]$
[oracle@ocp ~]$
[oracle@ocp ~]$ gdb $ORACLE_HOME/bin/oracle 4718
GNU gdb Fedora (6.8-27.el5)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
(no debugging symbols found)
Attaching to program: /u01/app/oracle/product/11.2.0/bin/oracle, process 4718
Reading symbols from /u01/app/oracle/product/11.2.0/lib/libodm11.so...(no debuggin
Loaded symbols for /u01/app/oracle/product/11.2.0/lib/libodm11.so
Reading symbols from /u01/app/oracle/product/11.2.0/lib/libcell11.so...done.
Loaded symbols for /u01/app/oracle/product/11.2.0/lib/libcell11.so
Reading symbols from /u01/app/oracle/product/11.2.0/lib/libskgxp11.so...done.
Loaded symbols for /u01/app/oracle/product/11.2.0/lib/libskgxp11.so
Reading symbols from /lib64/librt.so.1...done.
Loaded symbols for /lib64/librt.so.1
Reading symbols from /u01/app/oracle/product/11.2.0/lib/libnnz11.so...done.
Loaded symbols for /u01/app/oracle/product/11.2.0/lib/libnnz11.so
Reading symbols from /u01/app/oracle/product/11.2.0/lib/libclsra11.so...done.
Loaded symbols for /u01/app/oracle/product/11.2.0/lib/libclsra11.so
Reading symbols from /u01/app/oracle/product/11.2.0/lib/libdbcfg11.so...done.
Loaded symbols for /u01/app/oracle/product/11.2.0/lib/libdbcfg11.so
```

图 2-2　调试结果一

```
Loaded symbols for /usr/lib64/libaio.so.1
Reading symbols from /lib64/libdl.so.2...done.
Loaded symbols for /lib64/libdl.so.2
Reading symbols from /lib64/libm.so.6...done.
Loaded symbols for /lib64/libm.so.6
Reading symbols from /lib64/libpthread.so.0...done.
[Thread debugging using libthread_db enabled]
[New Thread 0x2ab830a4d910 (LWP 4718)]
Loaded symbols for /lib64/libpthread.so.0
Reading symbols from /lib64/libnsl.so.1...done.
Loaded symbols for /lib64/libnsl.so.1
Reading symbols from /lib64/libc.so.6...done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
Reading symbols from /usr/lib64/libnuma.so.1...done.
Loaded symbols for /usr/lib64/libnuma.so.1
Reading symbols from /lib64/libnss_files.so.2...done.
Loaded symbols for /lib64/libnss_files.so.2
Reading symbols from /u01/app/oracle/product/11.2.0/lib/libnque11.so...done.
Loaded symbols for /u01/app/oracle/product/11.2.0/lib/libnque11.so
0x0000003b2ea0d290 in __read_nocancel () from /lib64/libpthread.so.0
(gdb)
```

图 2-3　调试结果二

步骤 3：在 kslwtbctx 函数处设置断点，如图 2-4 所示。

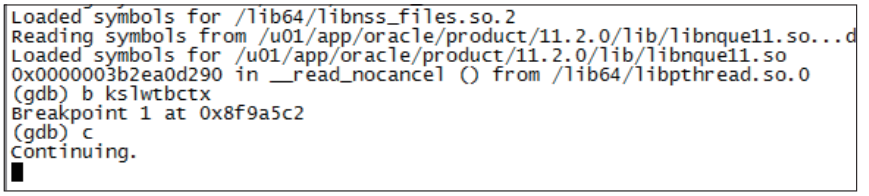

图 2-4　设置断点

kslwtbctx 函数是等待事件的起始函数。在它的入口处设置断点，接着使用了“c”命令，让进程继续处理指令流。

由于现在 4718 进程没有任何动作，因此没有执行到 kslwtbctx 函数处，所以在“c”命令后，gdb 显示 continuing，表示进程正在运行中，没有触发断点。

步骤 4：执行测试语句。

在测试会话中，随便执行一条语句（比如 select 语句），Oracle 都会产生等待事件。比如 select 会产生 SQL*Net 类的等待事件。下面执行测试 SQL，如图 2-5 所示。

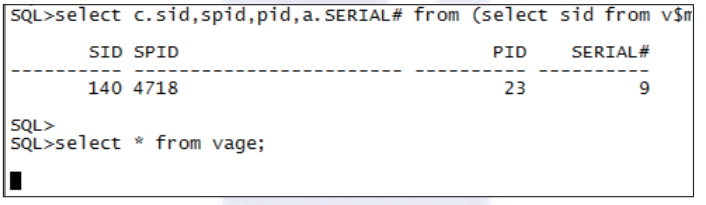

图 2-5　执行测试语句

140 会话中的 SQL 被 Gdb Hang 住了，说明断点已经被触发。

步骤 5：在 gdb 中查看断点被触发的情况。

测试会话的执行流，停在 kslwtbctx 函数处，如图 2-6 所示。

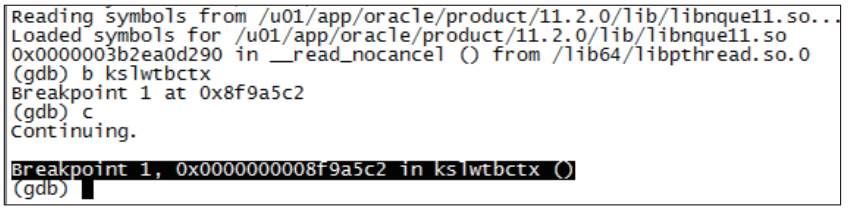

图 2-6　查看断点

此函数的第一个参数是一个指向进程自身堆栈内存的地址，它是一个 Struct 的指针。在进程自身堆栈空间中，可以认为 Struct 的内存属于 PGA。此 struct 中有等待事件的信息。

这些内部信息是使用 Dtrace 和 mdb 调试出来的。我在 Solaris 下调试 Oracle 时，曾经

研究过等待事件的相关函数，因此了解这些信息。

如何查看 kslwtbctx 第一个参数的值呢？如果是 32 位系统，函数参数会放在堆栈中，可以通过 rbp、rsp 寄存器找到函数参数。而在 64 位系统中，OS 已经做了优化，函数参数直接放在 rdi 寄存器中，即函数的第一个参数在 rdi 寄存器中。

关于这些信息，可以查阅 Intel CPU 手册和 Linux 内核分析的书籍。一名精通调试技术的 DBA，不是普通的 DBA，他必然会对 CPU、OS 内核非常了解。将来我们调试的不一定是 Oracle，也可以是其他重要的系统。

下面显示一下寄存器的值，如图 2-7 所示。

```
Breakpoint 1, 0x0000000008f9a5c2 in kslwtbctx ()
(gdb)
Continuing.

Breakpoint 1, 0x0000000008f9a5c2 in kslwtbctx ()
(gdb) info register
rax            0x0        0
rbx            0x2000     8192
rcx            0x0        0
rdx            0x0        0
rsi            0x0        0
rdi            0x7fff7ba0eb88    140735267531656
rbp            0x7fff7ba0e8c0    0x7fff7ba0e8c0
rsp            0x7fff7ba0e8c0    0x7fff7ba0e8c0
r8             0x19f37c80        435387520
r9             0xb619b6 11934134
r10            0x52ac59b4        1387026868
r11            0x2ab8310db7a8    46970585331624
r12            0x834f5f20        2203016992
r13            0x0        0
r14            0x0        0
r15            0x836bdcf8        2204884216
rip            0x8f9a5c2         0x8f9a5c2 <kslwtbctx+4>
```

图 2-7　显示寄存器

这里特意将 rdi 这行涂黑。它保存的数据是 0x7fff7ba0eb88。前文说了，这是一个地址。下面，显示它所指向的数据。使用命令 x/32 0x7fff7ba0eb88，从 0x7fff7ba0eb88 开始，显示 32 个字，如图 2-8 所示。

```
fioff          0x9394e95         154750613
foseg          0x7fff    32767
fooff          0x7ba0f590        2074146192
fop            0x0       0
mxcsr          0x1fa1    [ IE PE IM DM ZM OM UM PM ]
(gdb) x/32 0x7fff7ba0eb88
0x7fff7ba0eb88: 0x08f97a59      0x00000000      0x7ba0ec10      0x00007fff
0x7fff7ba0eb98: 0x09329a7d      0x00000000      0x00073fff      0x00000000
0x7fff7ba0eba8: 0x00000000      0x00000000      0x312e02f4      0x00002ab8
0x7fff7ba0ebb8: 0x01eb1fd9      0x00000000      0x00000000      0x00000000
0x7fff7ba0ebc8: 0x00003000      0x00000000      0x00000017      0x20000000
0x7fff7ba0ebd8: 0x00000000      0x00000000      0x00000001      0x00000000
0x7fff7ba0ebe8: 0x0a0fcd50      0x00000000      0x00000092      0x00000000
0x7fff7ba0ebf8: 0x7fffffff      0x00000000      0x00000001      0x00000000
(gdb)
```

图 2-8　显示数据

注意被涂黑的数据 0x00000092，这个值就是被调试进程的等待事件。这个等待事件是什么呢？

```
SQL>select event#,name from v$event_name where event#=to_number('92','xxxx');
    EVENT# NAME
---------- ----------------------------------------------------------------
       146 db file sequential read
```

如何验证当前会话正在等的事件的确是 db file sequential read 呢？直接查询 v$session 视图是不行的。

```
SQL>select event from v$session where sid=140;
EVENT
----------------------------------------------------------------
Disk file operations I/O
```

因为等待事件信息还没有完全写到资料视图中。其实验证方法很简单。

如图 2-9 所示是 140 会话曾经发生过的等待事件。

```
SQL>
SQL>select sid,EVENT,TOTAL_WAITS from V$SESSION_EVENT where sid in (140);

       SID EVENT                                    TOTAL_WAITS
---------- ---------------------------------------- -----------
       140 Disk file operations I/O                           1
       140 SQL*Net message to client                         18
       140 SQL*Net message from client                       18

SQL>
```

图 2-9　等待事件

在 gdb 中，输入命令“c”，让进程的执行流停在下一个等待事件上，如图 2-10 所示。

```
(gdb) x/32 0x7fff7ba0eb88
0x7fff7ba0eb88: 0x08f97a59      0x00000000      0x7ba0ec10      0x00007fff
0x7fff7ba0eb98: 0x09329a7d      0x00000000      0x00073fff      0x00000000
0x7fff7ba0eba8: 0x00000000      0x00000000      0x312e02f4      0x00002ab8
0x7fff7ba0ebb8: 0x01eb1fd9      0x00000000      0x00000000      0x00000000
0x7fff7ba0ebc8: 0x00003000      0x00000000      0x00000017      0x20000000
0x7fff7ba0ebd8: 0x00000000      0x00000000      0x00000001      0x00000000
0x7fff7ba0ebe8: 0x0a0fcd50      0x00000000      0x00000092      0x00000000
0x7fff7ba0ebf8: 0x7fffffff      0x00000000      0x00000001      0x00000000
(gdb) c
Continuing.

Breakpoint 1, 0x0000000008f9a5c2 in kslwtbctx ()
(gdb)
```

图 2-10　停在等待事件上

再次查看 140 会话当前曾经发生过的等待事件，如图 2-11 所示。

和刚才相比，多了一次 db file sequential read。

好，现在已经验证了 kslwtbctx 函数的第一个参数，向下 104 字节处的一个字，就是等待事件的 event#。

```
SQL>select sid,EVENT,TOTAL_WAITS from V$SESSION_EVENT where sid in (140);

       SID EVENT                                    TOTAL_WAITS
---------- ---------------------------------------- -----------
       140 Disk file operations I/O                           1
       140 SQL*Net message to client                         18
       140 SQL*Net message from client                       18

SQL>
SQL>/

       SID EVENT                                    TOTAL_WAITS
---------- ---------------------------------------- -----------
       140 Disk file operations I/O                           1
       140 db file sequential read                            1
       140 SQL*Net message to client                         18
       140 SQL*Net message from client                       18

SQL>
```

图 2-11　140 会话等待事件

有了这些信息，使用 dtrace 或 Linux 下的 ptrace 等调试工具，非常容易就能编写出显示进程运行过程中所有等待事件的跟踪脚本。但这已经超出本章主题，以后再讲。

下面看一个案例。情况很简单，用户的 RAC 系统最近一段时间老是莫名其妙地宕机。下面来分析一下宕机时的 DUMP 文件。

在 CRS_HOME/log 的日志中，只能看到节点 DOWN 了，没有进一步的信息。但告警日志中显示，宕机时 LMS 进程产生一个 TRC 文件。下面就尝试从这个文件挖掘线索。

先从 DUMP 文件开头开始，开头显示 LMS 进程会话号为 449。

```
*** SESSION ID:(449.1)
```

下面用会话号“449”，在文件内搜索，找到如下内容：

```
----------------------------------------
 SO: 0xc00000123065dcc8, type: 4, owner: 0xc00000123000a0c8, flag: INIT/-/-/0x00
 if: 0x3 c: 0x3
proc=0xc00000123000a0c8, name=session, file=ksu.h LINE:10719 ID:, pg=0
    (session) sid: 449 ser: 1 trans: 0x0000000000000000, creator:
    0xc00000123000a0c8
ksuxds FALSE at location: 0
service name: SYS$BACKGROUND
    Current Wait Stack:
     0: waiting for 'gcs remote message'
Wait State:
     auto_close=0 flags=0x22 boundary=0x0000000000000000/-1
    Session Wait History:
     0: waited for 'gcs remote message'
     1: waited for 'gcs remote message'
     2: waited for 'gcs remote message'
     3: waited for 'gcs remote message'
.....
```

这里显示 449 会话最后的等待事件是 gcs remote message。知道这点，对于解决问题

没有任何帮助，因为 Oracle 后台进程日常的等待事件就是 gcs remote message。从 Session Wait History 中也能看到这点。

但在进程调用堆栈中，发现有如下的内容：

```
.........
kslwaitctx()+240      call     $cold_ksliwat()       C00000123065F668 ?
C00000123065F668 ?
000000003 ?
600000000013F700 ?
kslwait()+192         call     kslwaitctx()          9FFFFFFFFFFFB710 ?
000000003 ?
.........
```

这段运行堆栈说明，kslwait 函数在偏移 0x192 字节处调用了函数 kslwaitctx。

由于用户数据库版本是 11.1.0.7，而前文中的测试版本是 11.2.0.4，因此 Oracle 内部函数名会略有不同。注意，这里的 kslwaitctx 就是前文测试中的 kslwtbctx，它的第一个参数指向等待事件的具体信息。

此处，它的第一个参数值为 0x 9FFFFFFFFFFFB710。用这个地址在 DUMP 文件中搜索，相关内容如图 2-12 所示。

```
713 Argument/Register addr=0x9fffffffffffb710.
714 Dump of memory from 0x9fffffffffffb6d0 to 0x9fffffffffffb810
715 9FFFFFFFFFFFB6D0 00000000 00136B44 20B747A3 0A7F143C  [......kD .G....<]
716 9FFFFFFFFFFFB6E0 00000000 0001003E 346DC5D6 3886594B  [.......>4m..8.YK]
717 9FFFFFFFFFFFB6F0 9FFFFFFF FFFFB7DC 00000010 00000000  [................]
718 9FFFFFFFFFFFB700 00000000 0001003E 346DC5D6 3886594B  [.......>4m..8.YK]
719 9FFFFFFFFFFFB710 7A590000 307446D2 00000089 870042D1  [zY..0tF.......B.]
720 9FFFFFFFFFFFB720 00000089 870042D1 00000089 870042D1  [......B.......B.]
721 9FFFFFFFFFFFB730 00000089 870042D1 00000000 00000000  [......B.........]
722 9FFFFFFFFFFFB740 00000000 00000000 00000089 87004290  [..............B.]
723 9FFFFFFFFFFFB750 00000089 870042A8 00000000 00000000  [......B.........]
724 9FFFFFFFFFFFB760 00000003 000005A8 00000001 00000000  [................]
725 9FFFFFFFFFFFB770 40000000 019172B0 000000A0 00000000  [@.....r.........]
726 9FFFFFFFFFFFB780 00000003 00000006 00000000 00000018  [................]
727 9FFFFFFFFFFFB790 00000000 00000000 00000000 00000000  [................]
728         Repeat 1 times
```

图 2-12　查找相关内容

注意，9FFFFFFFFFFFB710 向下 104 字节处的值是 000000A0，它就是等待事件的 event#，十进制是 160。在相同的版本下（这个库版本是 11.1.0.7），在 v$event_name 查看 event# 为 160 的等待事件：gc current block lost。

为什么跟踪文件下面的等待事件是 gcs remote message，而我们从调用堆栈挖出的等待事件是 gc current block lost 呢？

具体原因已经是另一个话题了，需要更深一步了解 Oracle 等待事件机制，这里不再详述。

本例中这种 DUMP 文件中等待事件错误的情况是很少见的，大部分时候调用堆栈中的等待事件，和下面 DUMP 的等待事件是一样的。

但如果出现不一样的情况，应该以调用堆栈中的等待事件作为最后的等待事件。

gc current block lost 是一个有关 gc 当前块的等待事件。这个等待事件笔者也没有深究过。不过，什么时候会有“当前块”的需求呢？很简单，进程在修改某个块时。这和单实例下的当前读一样。

普通 DML 所引发的 gc current 类等待，等待进程大多是服务器进程，很少是后台进程。这个 gc current 既然是后台进程，很有可能是 DDL 引起的，而且应该是频繁执行的 DDL。根据这个猜想对数据库应用进行排查，发现有一个 truncate 被频繁地执行。和应用部分协商后，将此表改为临时表后，节点就不再 DOWN 掉了。

这个例子先介绍到这里，总之，在确定问题时，等待事件是非常重要的。

等待事件的查看、等待事件的意义是 DBA 必须掌握的内容。但我们不能只关注等待事件，还有一些其他的信息可以帮助我们确定问题，比如资料视图。

2.1.3　各种资料视图介绍

Oracle 提供了很多 V$*STAT 资料视图，以便我们可以及时了解数据库的运行状态。比如 v$undostat 记录 UNDO 的使用情况，v$segstat 记录段级访问资料，等等。各种资料类视图中，有一类使用最为广泛，就是 v$sysstat、v$sesstat、v$statname。

在 AWR 中，许多信息其实都来自于 v$sysstat，比如报告最前面的 Profile 中的逻辑读次数、物理读次数、执行次数、解析次数等。

除了 Profile，AWR 中的 Instance Activity Statistics 其实展示的就是 v$sysstat 中的所有信息，这对于诊断问题还是很有帮助的。

阿里巴巴有一套核心数据库，曾经出现过一个奇怪的问题。某天中午 11 点多，应用服务器上的数据库连接数突然大增，数据库进程数迅速超过 Processes 参数值，新的连接无法建立，客服不断收到用户投诉。

这套库的 Processes 参数值设定为 4000，平常连接数每天低峰时 2000 左右，高峰时 3000 左右，出问题时是上午 11 点多，正是业务高峰期，连接数本来接近 3000。但在短短几分钟内，从这个数字上升到 4000，用完了所有的 Processes。

随后几天，这样的问题每天定时出现，每次都是 11 点左右。

由于当时使用的 Oracle 版本还是 9i，没有 ASH 可用，无法查询历史等待事件，而在以半小时为粒度的 Statspack 中又查不到任何异常，只能暂时 Kill 掉一些不重要的进程，暂时缓解一下问题。

至于对进程重要性的判定，则是通过看进程来自哪个应用服务器，也就是 v$session 的 MACHINE 列，还有 PROGRAM 列，和开发人员一起来判断的。

除了 Kill 进程外，我们一直在观察等待事件，因为既然进程数大增，说明一定是哪里出现了阻塞，导致用户操作完成得慢，而此时新的用户又在不断地连接，于是造成了进程

数突增。但很可惜，等待事件非常正常，没有发现任何异常的等待事件，一些关键等待事件的等待时间也都很正常。

几分钟后，也就是过了问题发生的时段，进程数滑落到正常范围内。虽然通过等待事件没有确定任何问题，不过幸好，我们以 10 秒为单位从数据库中抓取了一些关键资料，而这些资料的数据来源就是 v$sysstat。

如图 2-13 所示就是出问题时，执行次数、用户调用数、解析次数的曲线图。

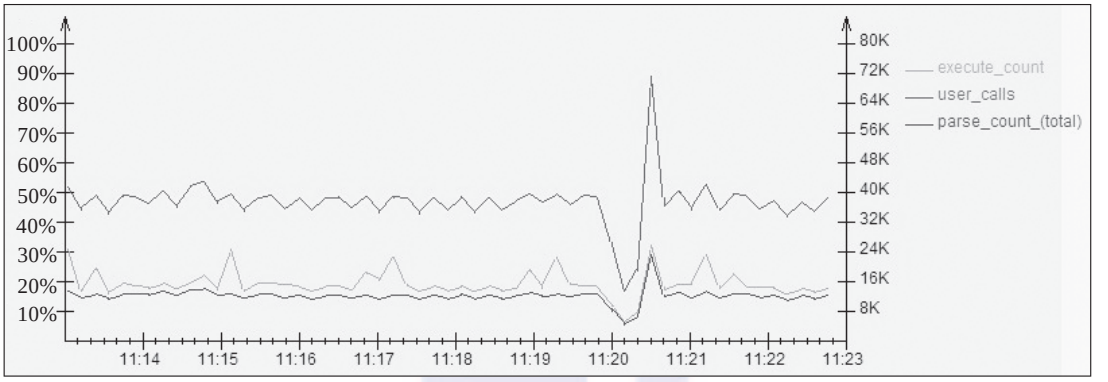

图 2-13　v$sysstat 资料视图使用示例（一）

图中这 3 个指标最上面的曲线是执行次数（execute count），中间的是调用次数（user calls），最下面的曲线是解析次数（parse count）。

虽然该图有些简陋，但仍可以看到问题，这 3 个指标在 11 点 20 分前，突然下滑，之后快速升高，再之后，又迅速恢复正常。而连接数（也就是进程数）的高涨，发生在 11 点 20 分左右，到 11 点 21 分之后恢复正常，正好是在这 3 个指标下滑到升高之间。

可以看到这 3 个指标都是先下降，后升高，而在此期间没有任何可以观察到的异常等待，甚至连总的等待次数、等待时间都没有太大波动。

所以，单纯依赖等待事件是无法判断此问题的。根据执行次数等资料值的波动倒是可以分析一下。

调用次数、执行次数、解析次数的下降说明用户都不再执行 SQL 操作了，又没有等待事件阻塞着用户，那么，是什么原因让用户同时都不再执行 SQL 了？很明显，是用户突然无法将需求发送给数据库了。又是什么原因导致用户无法将需求发送给数据库的呢？只能是网络。

但是网络的状态和资料的查看时间粒度比较粗，是以 10 分钟为周期的，所以并没有发现异常情况。

数据库端资料显示，每次都是在 11 点 20 分前执行次数开始下滑时，应用服务器上的需求无法传到数据库，到 11 点 20 分后，网络恢复正常，大量积压的需求突然一下传到数据库，导致执行次数等资料值突升。

另外，开发人员也检查了应用服务器上的程序，逻辑设计还是有点问题的：在网络异常阶段，程序有连接或执行 SQL 需求无法完成，会不断循环，尝试创建新的连接。这导致一旦无法建立连接，或 SQL 无法完成，应用服务器会疯狂地向数据库发送连接请求。这可能造成网络一旦恢复正常，数据库瞬间收到大量连接和 SQL 执行请求。

由于网络那边无法检查以分钟以下为单位的异常，最终我们将目光瞄向了和数据库服务器同一网段、共享网络的其他主机。经过排查，发现有一组十几台 MySQL 数据库，每天会定时向另外一组 MySQL 同步数据，而在出问题的那天，同步数据的时间变了，本来应该是晚上同步，而变到了 11 点 20 分左右。而且，在刚开始同步时，流量还是很大的。

看来问题已经找出来了，11 点 20 分左右 MySQL 突然同步，导致网络带宽被大量占用，应用服务器上的连接、SQL 请求无法传到数据库，导致此时数据库的执行次数等资料突然下降。然后，应用服务器上的程序开始不断创建新的连接。接着，MySQL 同步数据量下降，网络空闲，大量新的连接、和命令请求同时传到数据库，使得进程数瞬间增加至很高。

将 MySQL 的同步数据时间改变后，数据库恢复正常，再没有出现过连接数突增的情况。

从这个案例可以看出，如果在确定问题时只关注等待事件是无法找出产生问题的根源的，因为网络只是被别的程序占用，并不是完全断开，仍有部分请求可以传到数据库上执行，而且虽然新的 SQL 请求无法及时传到数据库，但原来已有的语句还在执行。所以，从等待事件上看，数据库完全正常，都是一些 I/O 类等待事件。

执行次数指标的先低后高，再加上没有异常的等待事件，使我们可以确定，问题一定不在 Oracle 范围内。

其实对于这种情况，还有两个资料也可以帮助确定问题：bytes received via SQL*Net from client 和 bytes sent via SQL*Net to client。对这两个网络相关的资料，大部分 DBA 并不怎么重视，因为数据库的主要瓶颈就是 I/O、CPU 和内存，网络通常并不会造成竞争和等待。特别是网络相关等待事件 SQL*Net message from client（等待从客户端接收命令），更是相关进程空闲的代表。但网络相关的资料是十分有意义的，网络相当于 Oracle 的大门，用户需求通过网络传送到 Oracle，Oracle 处理完需求后，结果数据还是要通过网络传送给客户端。这一进一出都要经过网络这道大门，统计进、出网络的流量，对于了解数据库负载是十分有意义的事。其实几乎数据库的任何风吹草动都会在网络上体现出来。

比如在该示例中，在执行次数先低后高时，bytes received via SQL*Net from client 资料值也出现突然的下降。这些都可以帮助我们确定问题。

之后又遇到过一个类似案例：一个 DBA 所维护的数据库被投诉应用不正常。在出问题时段，从 AWR 报告、ASH 视图中的等待事件查不到任何异常。我对比了 AWR 报告中的 bytes received via SQL*Net from client 和 bytes sent via SQL*Net to client，发现通过网络收到的信息量在出现问题时段比平常少一个量级。收到的信息减少，说明数据库根本收不到应用程序发送的 SQL 请求。我把结果告诉那个 DBA，让他将问题交给网络人员定位，结果果真找出了问题所在。

其实，只要是数据出问题，网络资料都会有些异常。用图形工具软件可以将网络资料转成曲线图，只要网络曲线图有任何风吹草动，数据库中必会有所反应。

图 2-14 至图 2-17 是我的一些总结，依次来看看。

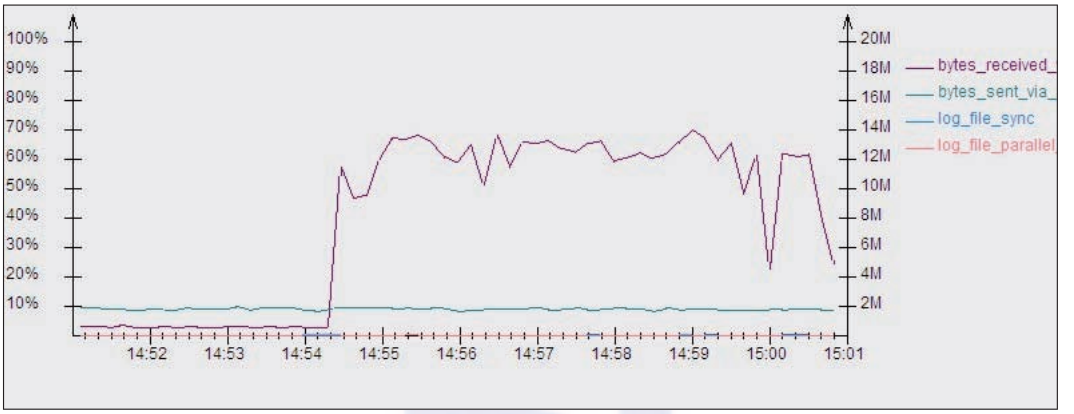

图 2-14　v$sysstat 资料视图使用示例（二）

从图 2-14 中可以看到，在 14 点 54 分多一点的时候，有一条曲线波动剧烈，这条曲线就是 bytes received via SQL*Net from client，即收到字节数。它突然大幅升高，表示数据库定有异常。再查看其他资料，如图 2-15 所示。

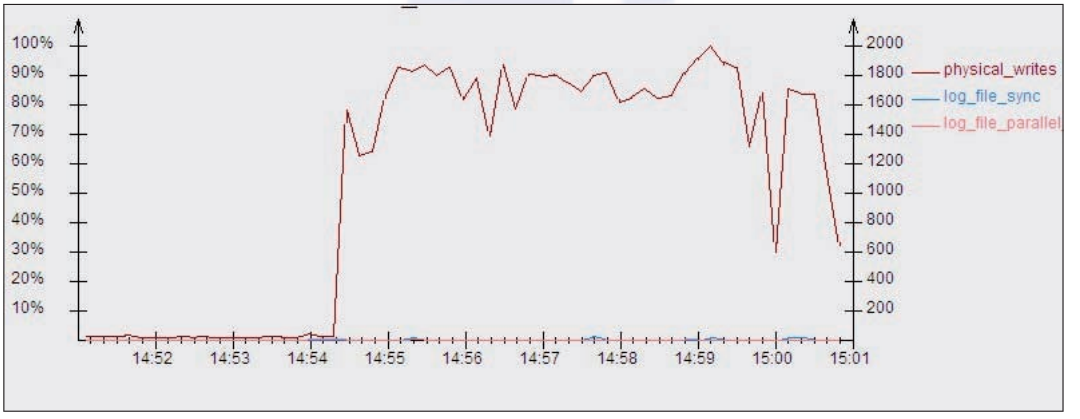

图 2-15　v$sysstat 资料视图使用示例（三）

在这幅图中，波动剧烈的资料是物理写。在同一时间，它也大幅增加，它的曲线图和 bytes received via SQL*Net from client 基本吻合。更进一步排查原因，发现是由大量的排序操作所导致的，调整 SQL 后变得正常，未再出现物理写数据量飙升的情况。

再来看一个例子，如图 2-16 所示。

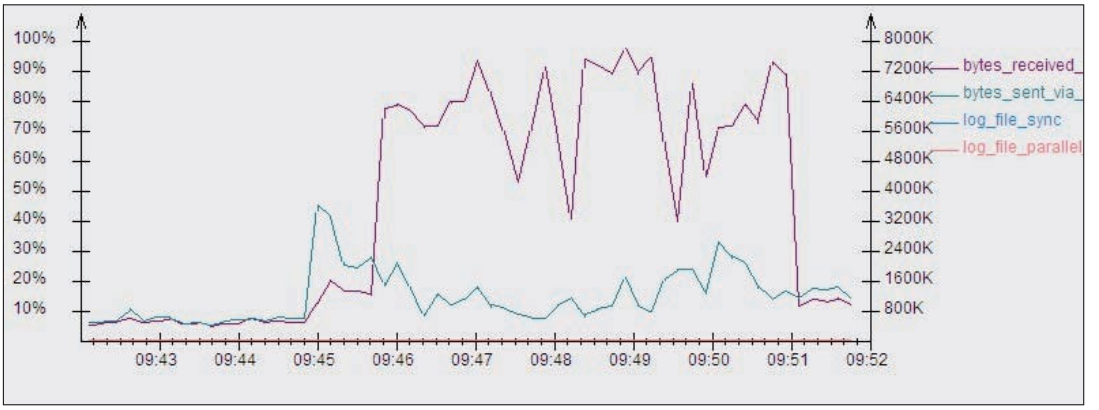

图 2-16　v$sysstat 资料视图使用示例（四）

在图 2-16 中，从 9 点 46 分到 9 点 51 分出现了收到字节数异常的情况。经过对比，发现这次是出现了日志资料异常，如图 2-17 所示。

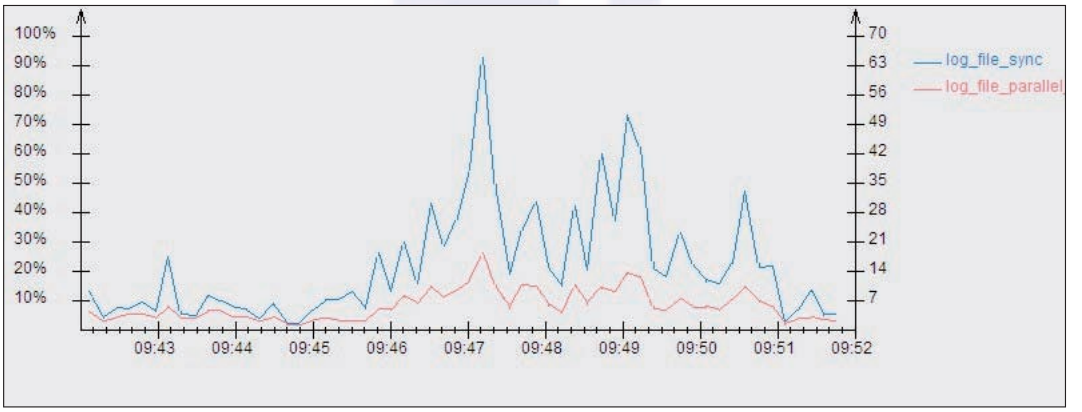

图 2-17　v$sysstat 资料视图使用示例（五）

图 2-17 中波动剧烈的曲线是 Log File Sync 的响应时间，在 9 点 46 分到 9 点 51 分之间，响应时间慢了将近 10 倍。最终的问题是由于存储控制器后端口压力不平衡，在 I/O 压力大时，响应时间出现抖动。存储工程师对此进行调整后该问题得到解决。

更多例子这里不再列举。在阿里巴巴的所有数据库中，重要的资料都会用曲线图展现。在每日查看数据库健康状态时，我都会从网络曲线图入手。如果网络资料没有大幅度波动，数据库基本上都是正常的。如果看到网络资料波动很大，就要进一步查看数据库是否有问题。

资料视图中的有些资料，其本身虽有一定含义，但从名字上却不容易看出来，比如 Redo entries（重做条目数）。一条 Redo Recoder，又可以称为一个 Redo entrie。因此这项资料是用于记录 Oracle 一共生成了多少条 Redo Recoder 的。这个资料看似意义不大，但是，如果进一步发掘 Redo 的原理，可以发现 Redo Recoder 的数目可以告诉我们其他信息。

分析一下 Redo 信息的生成流程：通常 Oracle 会先在 PGA 中记录后映像，在 IMU 方式下，后映像还会传送到共享池中暂存，并且最终在 Log Buffer 中根据后映像组装成一条 Redo Recoder。

这里暂时不对 IMU 相关问题及日志流程详细讨论，后面有专门章节描述相关内容。

注意一点，只有在 Redo 相关的数据被传送到 Log Buffer 后，这些数据才会被组装成 Redo Recoder。换句话说，当 Redo 相关的数据被传送到 Log Buffer 时，这些数据才会被称为 Redo Recoder。

可见，有一条 Redo Recoder 则说明向 Log Buffer 传送过一次 Redo 数据，有 100 条 Redo Recoder 则说明向 Log Buffer 传送过 100 次数据。也就是说，Redo Recoder 的数量代表了向 Log Buffer 中写数据的次数，这就是 Redo entries 资料背后的意义。

如果出现 Log Buffer Space 类的等待事件，或者 Redo Copy Latch、Redo allocate Latch 竞争严重，可以对比一下正常和不正常时段 Redo entries 资料的值。

如果 Redo entries 的值很高，则说明向 Log Buffer 中写数据的次数很多。

每向 Log Buffer 中写一次数据，都会申请一次 Redo Copy Latch、Redo allocate Latch 等相关的 Latch。向 Log Buffer 中写数据次数太多，必然会导致这些 Latch 有竞争。IMU 的出现，就是为了缓解这些竞争的。

2.1.4 等待事件的注意事项

在查看等待事件的时候，有一个小的注意事项，很多人平常并不注意，在此用一小测试，提醒一下各位读者。

步骤 1：在两个会话中分别执行类似下面的 PL/SQL 匿名块。

```
declare
m_id number;
begin
fori in 1..10000000 loop
select id into m_id from vage where rownum=1;
end loop;
end;
/
```

此匿名块什么工作都不做，只是在循环中反反复复查询表 T 的第一行数据。

如果在两个会话中同时执行此段程序，会遇到什么等待事件？

步骤 2：在另一会话中观察等待事件。

```
col event for a45
setlinesize 1000
select sid,seq#,event from v$session where wait_class<>'Idle' order by event;

SID SEQ# EVENT
```

```
--- ---- -------------------------
237 769 SQL*Net message to client
256 231 latch: cache buffers chains
253 253 latch: cache buffers chains
```

可以看到，由于反复逻辑读同样的块，会有块上的 Cache Buffer Chain Latch 竞争。还有，由于两个会话中会反复解析同样的 SQL，因此有可能会有解析时的 Mutex 等待。如果重复执行上面的 SQL，可以观察到 Cache Buffer Chain Latch 或 Cursor:pin S 等待事件。

出现等待事件的原因是两个会话对应的服务器进程同时请求同样的资源，如果此时将一个会话中的 PL/SQL 程序块终止掉会怎样呢？想必只剩一个进程解析、逻辑读，就不会有等待了。是否真是这样呢？

步骤 3：将一个会话中正在执行的 PL/SQL 终止，再次观察等待事件。

终止 PL/SQL 的执行是很简单，在一个会话中使用快捷键 Ctrl+C 即可。现在只剩一个会话反反复复解析、逻辑读了，这样应该就不会再有等待了。使用步骤 2 中的 SQL，再一次观察等待事件。结果如下：

```
SQL> select sid,seq#,event from v$session where wait_class<>'Idle' order by event;

SID        SEQ# EVENT
---------- ---------- ----------------------------------------------
237        825 SQL*Net message to client
256        350 latch: cache buffers chains
```

还是有等待。一个进行正在执行的程序已经被终止了，只剩另一个进程在运行，但查询结果还是有等待。

原因很简单，我们在用 v$session 查看当前等待事件时，记得要把 STATE 列带上。下面是修改后的显示结果：

```
SQL> select sid,seq#,event,state from v$session where wait_class<>'Idle' order by event;

SID SEQ# EVENT                          STATE
--- ---- ---------------------------- -------------------
237  829 SQL*Net message to client    WAITED SHORT TIME
256  350 latch: cache buffers chains  WAITED SHORT TIME
```

STATE 列如果为 WAITING，说明会话正在等待此事件。如果不是 WAITING，说明当前会话已经不再等待某个事件了，v$session 中的 EVENT 列，显示的只是会话最后一次等待的事件。

新的显示结果中，STATE 为 WAITED SHORT TIME，所以，此处的 Cache Buffer Chain Latch 等待，是会话最后一次等待时的事件。会话现在已经不等待了。

不等待这是进程工作的理想状态，即没有任何等待完成工作就仿佛开车上班，一路无红灯，畅通无阻地前行。

这个例子很简单，但越是简单的地方，越容易被忽视。

2.2 AWR 概览

2.2.1 AWR 报告的注意事项

AWR 报告是进行健康检查时必不可少的工具。关于报告前面的 Load Profile、TOP5 等待等这些东西不再叙述，已经有太多相关资料。AWR 的中后部有些信息值得注意，比如 Instance Activity Stats。上一节讲了很多资料类视图相关的内容，如果 AWR 报告是 30 分钟产生一份，那么 Instance Activity Stats 其实就是这 30 分钟内各种资料的值。

不过这一部分内容很多人在阅读时往往忽略，但定位 Oracle 问题是这一部分内容中必不可少的。

除这一部分外，还有一些容易被忽视的，如 I/O Stats 部分。I/O 始终是数据库的“命门”，I/O 压力的不均衡，可能导致奇怪的问题。看如图 2-18 所示的这份 AWR 报告。

Top 5 Timed Foreground Events

Event	Waits	Time(s)	Avg wait (ms)	% DB time	Wait Class
db file sequential read	2,471,268	2,672	1	74.74	User I/O
DB CPU		735		20.57	
db file scattered read	14,910	52	3	1.44	User I/O
unspecified wait event	2,492,034	32	0	0.90	Other
control file sequential read	21,757	25	1	0.71	System I/O

图 2-18　AWR 报告（一）

其中，unspecified wait event 是一个奇怪的 Event。从 TOP 5 来看，I/O 响应时间都很正常，db file sequential read 是 1 毫秒，db file scattered read 是 3 毫秒，这两个值不仅正常，而且应该说是很不错。查看所有 I/O 相关的等待，Control file sequential read、Log file parallel write 等，响应时间为几毫秒，可以排除是 I/O 问题造成的 unspecified wait event。

但真的是这样吗？继续查看 IO Stats 部分，在 Tablespace IO Stats 部分，发现有一个表空间的 I/O 响应延时远远高于正常水平，如图 2-19 所示。

从这部分可以看到，有一个表空间的 I/O 响应时间已经是 1491 毫秒了。再进一步查看 File IO Stats 部分，如图 2-20 所示。

还是刚才那个表空间，它有两个数据文件，I/O 平均响应时间已经高达四五千毫秒了。也就是在此份 AWR 报告期间，这两个文件要平均四五秒才能完成一个 I/O。

Tablespace	Reads	Av Reads/s	Av Rd(ms)	Av Blks/Rd	Writes	Av Writes/s	Buffer Waits	Av BufWt(ms)
.....	.....	.....	.....	.....	.....	.....	.....	.....
**********	2,628	1	1491.66	1.00	86	0	2	5.00
.....	.....	.....	.....	.....	.....	.....	.....	.....

图 2-19　AWR 报告（二）

注：出于保密原因，表空间名称部分省去。

File IO Stats

ordered by Tablespace, File

Tablespace	Filename	Reads	Av Reads/s	Av Rd(ms)	Av Blks/Rd	Writes	Av Writes/s	Buffer Waits	Av BufWt(ms)
.....	.....	316	0	5.89	1.00	13	0	0	0.00
********	+DG1/********.dbf	375	0	5202.24	1.00	13	0	0	0.00
********	+DG1/********.dbf	407	0	4793.98	1.00	15	0	0	0.00
.....	.....	36	0	8.89	1.00	21	0	0	0.00

图 2-20　AWR 报告（三）

这只是平均值，说不定在某个时刻，I/O 的响应时间比这个还要长。如果 I/O 过分的缓慢，产生一些奇怪的等待事件，如 unspecified wait event，也就不足为奇了。

这份报告是晚上 23 点至 23 点 30 分产生的，当时这个数据库正在同时进行大概 10 个左右的大表加载数据操作，I/O 量非常大，超过了存储的瓶颈，因此 I/O 响应时间很长。数据加载完成后，像 unspecified wait event 之类的奇怪等待事件再没有出现过。

在进一步诊断问题时，AWR 报告后面的这部分内容其实为我们列出了很多有用的信息，平常在诊断问题时可以注意挖掘。

但说白了，AWR 其实也只是把各种运行资料、等待事件组合起来显示而已，如何阅读 AWR 报告，建立在对这些资料、等待事件的理解基础上。如果不知道这些资料、等待事件的含义，一份报告根本无从看起。

在后面的章节中，每讲一部分内容，都先给出原理，然后根据原理介绍相关运行资料、等待事件的含义。

当然，Oracle 中有很多等待事件和运行资料，本书不可能每个都详细讲到。实际上，学习挖掘、分析 Oracle 等待事件和运行资料含义的方法，比了解某个事件、资料的含义更有意义。授人以鱼，不如授人以渔。直接告诉你某个等待事件的意义，远不如告诉如何发

掘等待事件的意义更重要。这将是本书后面章节的主要内容。

2.2.2 AWR 类视图

AWR 报告的底层有一系列以 DBA_HIST_ 为前缀的视图，用于保存 AWR 的历史资料，Oracle 每隔一定时间，写一份所有资料、等待事件类视图的快照到此类视图中，AWR 报告中的大部分内容都来自这些快照。所有快照的信息都保存在 DBA_HIST_SNAPSHOT 视图中。如下语句可以查看最早的和最近的快照：

```
selectmax(BEGIN_INTERVAL_TIME),
    min(BEGIN_INTERVAL_TIME),
    max(SNAP_ID),
    min(snap_id)
from DBA_HIST_SNAPSHOT ;
```

下面的语句可以查看快照的时间间隔和最早的快照编号：

```
setlinesize 1000
col BEGIN_INTERVAL_TIME for a40
select * from (select BEGIN_INTERVAL_TIME,SNAP_ID  from DBA_HIST_SNAPSHOT order
by BEGIN_INTERVAL_TIME) where rownum<=10;

SQL> set linesize 1000
SQL> col BEGIN_INTERVAL_TIME for a40
SQL> select * from (select BEGIN_INTERVAL_TIME,SNAP_ID  from DBA_HIST_SNAPSHOT
order by BEGIN_INTERVAL_TIME) where rownum<=20;

BEGIN_INTERVAL_TIME                         SNAP_ID
------------------------------------------- ----------
07-AUG-11 12.00.58.249 AM                     26659
07-AUG-11 12.30.58.283 AM                     26660
07-AUG-11 01.00.00.680 AM                     26661
07-AUG-11 01.30.03.632 AM                     26662
.....
```

可以看到，Oracle 半小时产生一份快照。目前最早的一份快照是 8 月 7 日上午 12 点产生的，编号 26659。

所有其他的 DBA_HIST_ 视图基本上都有 SNAP_ID 列，可以根据此列关联。比如，如下语句查看从实例启动到 8 月 10 日 9 点时的物理读信息：

```
setlinesize 1000
col BEGIN_INTERVAL_TIME for a40
selecta.BEGIN_INTERVAL_TIME,
    a.SNAP_ID,
    b.stat_name,
    b.value
from DBA_HIST_SNAPSHOT a,DBA_HIST_SYSSTAT b
```

```
wherea.BEGIN_INTERVAL_TIME>=to_date('2011-08-10 09:00:00')
    anda.BEGIN_INTERVAL_TIME<=to_date('2011-08-10 09:29:00')
    and b.SNAP_ID=a.SNAP_ID
    andb.stat_name like 'physical reads' ;

BEGIN_INTERVAL_TIME   SNAP_ID STAT_NAME VALUE
--------------------------------------------------- ---------
10-AUG-11 09.00.53.960 AM 26816 physical reads   6285630462
```

因为 AWR 是半小时产生一次快照，有时快照的时间点不是整点，所以上述条件是从 9 点至 9 点 29 分。

数据库已经累计物理读 6285630462 块，但查看累计值没有意义。如果要查看 8 月 10 日上午 8 点 30 分到 9 点的物理读，可以很简单地扩展如上语句为如下语句：

```
selecta.BEGIN_INTERVAL_TIME,
    b.BEGIN_INTERVAL_TIME,
    a.value,
    b.value,
   (b.value-a.value)/1800
from
   (selecta.BEGIN_INTERVAL_TIME,
           a.SNAP_ID,
           b.stat_name,
           b.value
   from DBA_HIST_SNAPSHOT a,DBA_HIST_SYSSTAT b
wherea.BEGIN_INTERVAL_TIME>=to_date('2011-08-10 08:30:00')
   anda.BEGIN_INTERVAL_TIME<=to_date('2011-08-10 08:59:00')
   andb.SNAP_ID=a.SNAP_ID and b.stat_name like 'physical reads'
) a,
(selecta.BEGIN_INTERVAL_TIME,
    a.SNAP_ID,
    b.stat_name,
    b.value
from DBA_HIST_SNAPSHOT a,DBA_HIST_SYSSTAT b
wherea.BEGIN_INTERVAL_TIME>=to_date('2011-08-10 09:00:00')
    anda.BEGIN_INTERVAL_TIME<=to_date('2011-08-10 09:29:00')
    andb.SNAP_ID=a.SNAP_ID and b.stat_name like 'physical reads'
) b;

BEGIN_INTERVAL_TIMEBEGIN_INTERVAL_TIMEVALUE  VALUE(B.VALUE-A.VALUE)/1800
--------------------------------------------------- ---------- ------------ ------
10-AUG-11 08.30.52.268 AM  10-AUG-11 09.00.53.960 AM  6274864874 6285630462    5980.88222
```

因为快照间隔时间是半个小时，共 1800 秒，因此在 SQL 语句中有 (b.value-a.value)/1800，总半个小时的总量除以 1800 秒，得到 8 月 10 日 8 点 30 分到 9 点，物理读每秒约 5980 块。

除 SNAP_ID 列以外，有些视图也有时间列，像 DBA_HIST_UNDOSTAT、DBA_HIST_ACTIVE_SESS_HISTORY 等。它们的值并不累加，不像 DBA_HIST_SYSSTAT、DBA_

HIST_FILESTATXS 等这些资料的视图，它们的值是累加的。而 DBA_HIST_ACTIVE_SESS_HISTORY 类型的视图和前面所述又不一样，在一个快照周期（比如 30 分钟）内，可以有各种各样的等待事件。对于这样的情况，Oracle 在一个 AWR 快照周期内，又会以固定的间隔时间抓取此类视图的快照。就以 DBA_HIST_ACTIVE_SESS_HISTORY 为例，Oracle 每秒将 V$Session 快照写入 V$ACTIVE_SESSION_HISTORY，每 10 秒写入 DBA_HIST_ACTIVE_SESS_HISTORY。因此，这类视图除了有一个 SNAP_ID 外，还会有个 SAMPLE_ID 列。SNAP_ID 是 AWR 快照 ID，SAMPLE_ID 则是此类视图自己的快照 ID。

此类视图的查看不需要将两份快照的值相减，以半小一次 AWR 快照为例，8 点 30 分到 9 点的等待事件，在 DBA_HIST_ACTIVE_SESS_HISTORY 中 SNAP_ID 统一取 8 点 30 分的值。因此，查看 8 月 10 日 8 点 30 分到 9 点的等待事件的语句如下：

```
setlinesize 1000
col BEGIN_INTERVAL_TIME for a40
selecta.BEGIN_INTERVAL_TIME,
    a.SNAP_ID,
    b.SAMPLE_TIME,
    b.event,
    b.p1,
    b.p2
from DBA_HIST_SNAPSHOT a,DBA_HIST_ACTIVE_SESS_HISTORY b
wherea.BEGIN_INTERVAL_TIME>=to_date('2011-08-10 08:30:00')
    anda.BEGIN_INTERVAL_TIME<=to_date('2011-08-10 08:59:00')
    andb.SNAP_ID=a.SNAP_ID
    andb.wait_class<> 'Idle'
order by sample_time;
```

当然，也可以不管什么 SNAP_ID 了，因为此类视图自带时间，直接以时间为单位查询也可以。

```
select SNAP_ID,
     SAMPLE_ID,
     SAMPLE_TIME,
     session_id,
     USER_ID,
     event,
     p3,
     module
from DBA_HIST_ACTIVE_SESS_HISTORY
where SAMPLE_TIME>=to_date('2011-08-10 08:30:00','yyyy-mm-dd hh24:mi:ss')
and SAMPLE_TIME<=to_date('2011-08-10 09:00:00','yyyy-mm-dd hh24:mi:ss')
order by sample_time;
```

如果想查看 SQL 会麻烦一些。

```
setlinesize 1000
col BEGIN_INTERVAL_TIME for a40
selecta.BEGIN_INTERVAL_TIME,
```

```
    a.SNAP_ID,
    b.sql_id,
    b.EXECUTIONS_TOTAL,
    b.EXECUTIONS_DELTA,
    DISK_READS_TOTAL,
    DISK_READS_DELTA
from DBA_HIST_SNAPSHOT a,DBA_HIST_SQLSTAT b
wherea.BEGIN_INTERVAL_TIME>=to_date('2011-08-10 08:30:00')
    anda.BEGIN_INTERVAL_TIME<=to_date('2011-08-10 08:59:00')
    andb.SNAP_ID=a.SNAP_ID ;

select * from
(
    selecta.BEGIN_INTERVAL_TIME,
           a.SNAP_ID,
           b.sql_id,
           b.EXECUTIONS_TOTAL,
           b.EXECUTIONS_DELTA,
           DISK_READS_TOTAL,
           DISK_READS_DELTA
    from DBA_HIST_SNAPSHOT a,DBA_HIST_SQLSTAT b
    wherea.BEGIN_INTERVAL_TIME>=to_date('2011-08-10 09:00:00')
    anda.BEGIN_INTERVAL_TIME<=to_date('2011-08-10 09:29:00')
    andb.SNAP_ID=a.SNAP_ID
)
wheresql_id='aqb23gd02krbd';
```

在 SQL 的 AWR 视图中，有关资料的列通常有两列：***_TOTAL，***_DELTA。以物理读为例，DISK_READS_TOTAL 表示累计物理读，DISK_READS_DELTA 表示两次 AWR 快照的物理读增量值。如果半小时抓一次 AWR 快照，那么 9 点时候的 DISK_READS_DELTA 中保存的是 8 点 30 分到 9 点间某条 SQL 的物理读。

以下语句查看 8 月 10 日 8 点 30 分到 9 点间 SQL 的物理读和执行次数，以增量物理读排序（注意，8 点 30 分到 9 点间的增量数据，在 9 点的快照中）。

```
setlinesize 1000
col BEGIN_INTERVAL_TIME for a40
selecta.BEGIN_INTERVAL_TIME,
    a.SNAP_ID,
    b.sql_id,
    b.EXECUTIONS_TOTAL,
    b.EXECUTIONS_DELTA,
    DISK_READS_TOTAL,
    DISK_READS_DELTA
from DBA_HIST_SNAPSHOT a,DBA_HIST_SQLSTAT b
wherea.BEGIN_INTERVAL_TIME>=to_date('2011-08-10 09:00:00')
    anda.BEGIN_INTERVAL_TIME<=to_date('2011-08-10 09:29:00')
    andb.SNAP_ID=a.SNAP_ID
order by DISK_READS_DELTA;
```

查询结果如下：

```
    BEGIN_INTERVAL_TIME    SNAP_ID SQL_ID    EXECUTIONS_TOTAL EXECUTIONS_DELTA DISK_
READS_TOTAL DISK_READS_DELTA
    ------------------------------------ ---------- ------------- ----------------
    10-AUG-11 09.00.53.960   AM   26816  cqda35pgjhxc4    1  1   2743922    495307
    10-AUG-11 09.00.53.960   AM   26816  c3amcasx93pvb  190  1  82661833    500732
    10-AUG-11 09.00.53.960   AM   26816  4qk3ay7bq4pab 1250  1 571678928    508307
    10-AUG-11 09.00.53.960   AM   26816  2skr3f87yx371    0  0   1648390   1648390
    10-AUG-11 09.00.53.960   AM   26816  7d36vg5ntu4x4    1  1   1651535   1651535
    10-AUG-11 09.00.53.960   AM   26816  b2q33zan1bk40    1  1   2239386   2239386
    10-AUG-11 09.00.53.960   AM   26816  3u19w33vtv358    1  1   4043676   2473630
```

DBA_HIST_SQLSTAT 中没有 SQL 语句，要查看 SQL 语句，还要和 DBA_HIST_SQLTEXT 关联。此视图中没有时间、SNAP_ID 这些列，只有 SQL_ID 列。以上面的查询结果为例，查看物理读最多的 SQL 是哪条的语句如下：

```
    selectDBID,SQL_ID,SQL_TEXT from DBA_HIST_SQLTEXT where sql_id='3u19w33vtv358';

    DBID SQL_ID        SQL_TEXT
    ---------- ----------------------------------------
     978291946 3u19w33vtv358 insert /*+append*/ into
       ***.*******
    select * from *****.en_
```

这是条同步语句。

另外，在 DBA_HIST_SQL_PLAN 中，还有历史执行计划，此视图也是只有 SQL_ID 列，没有时间和快照 ID 列。

另外，还有一个视图必须交代一下，即 ASH（ActiveSessionHistory）。从名字就可以知道，这些视图着重反映数据库的“历史”情况，对于诊断数据库的“历史”问题很有帮助。通常从发现数据库有问题，到 DBA 登录到数据库查看情况中间，至少会有十几分钟的时间，有可能 DBA 登录数据库查看时，问题已经没有了。通过 ASH 类历史视图，可以查看几分钟前，甚至几个小时、几天前的等待事件等信息，从而帮助 DBA 诊断问题是如何产生的。

ASH 对应的视图就是 V$ACTIVE_SESSION_HISTORY，Oracle 每秒会将所有会话非 Idle 类的等待事件记录到此视图中。此视图的所有数据都在内存中，Oracle 会定期将数据写到 DBA_HIST_ACTIVE_SESS_HISTORY 中，此数据字典视图存储在磁盘中。

Oracle内核技术揭密

Oracle Core Revealed

海波是Oracle DBA领域出道较晚而又能脱颖而出的异类，他在几乎被钻研通透的Oracle数据库技术领域独辟蹊径，找到了一条属于他自己的兴趣盎然之路。深入内核分析数据库的精髓，既需技艺，又需耐心，如无兴趣与毅力则必无数年如一日的决心，海波做到了他人极难做到的坚持，这非常值得钦佩。书中来自实证实验的剖析，将Oracle数据库的原理精烹而细饪，识者请自取之。

—— **盖国强** 云和恩墨创始人，Oracle ACE总监

海波，我数据库学习道路上的领路人之一。遥想当年，拿着海波精心准备的各种介绍Oracle功能实现、隐含参数、内核剖析的小册子，边阅读边做实验，受益匪浅。现如今，小册子变成了这本大部头，内容更是增加了作者在最近几年对Oracle新的研究体会，相信读者能从此书中，汲取更多的营养。

—— **何登成** 网易杭州研究院 技术专家

吕海波，网名VAGE。曾混迹于杭州多年，圈内同行称之为“瓦鸡”。他是我多年好友，也是我OCM导师，其Oracle水平，一直是我最佩服的三人之一。他待Oracle始终如初恋，安全地度过了七年之痒。VAGE与Oracle长时间的耳鬓厮磨，让Oracle在其面前再无秘密可言。故VAGE在不同场合炫耀其独享大众“情人”之后的回味，着实令我等羡慕不已。在闲暇之余，我经常以各种借口向其讨教“调教”之法，然始终未得其真传，只可远观而不可亵玩。今日，VAGE耗时2年，终于将其“调教大法”公布于众，对广大DBA来说，幸矣！

—— **周亮** Oracle ACE、《Oracle DBA实战攻略》作者、美创科技技术部经理

上架指导：计算机/数据库

ISBN 978-7-111-46931-5

9 787111 469315 >

定价：69.00元

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn