飞哥带你揭秘：为什么HugePage能让Oracle数据库如虎添翼？

原创 张彦飞allen 开发内功修炼 2025年01月09日 09:01 北京

大家如果有人部署过 Oracle 数据库的话，一定也看到过 Oracle 为了性能考虑，是推荐开启大页（HugePage）的。

那么为什么开启大页 能有性能提升，它的优化原理是啥，又是如何实现的呢？今天飞哥就来和你一起深入地聊聊这个 Topic。

一、 内核四级页表之殇

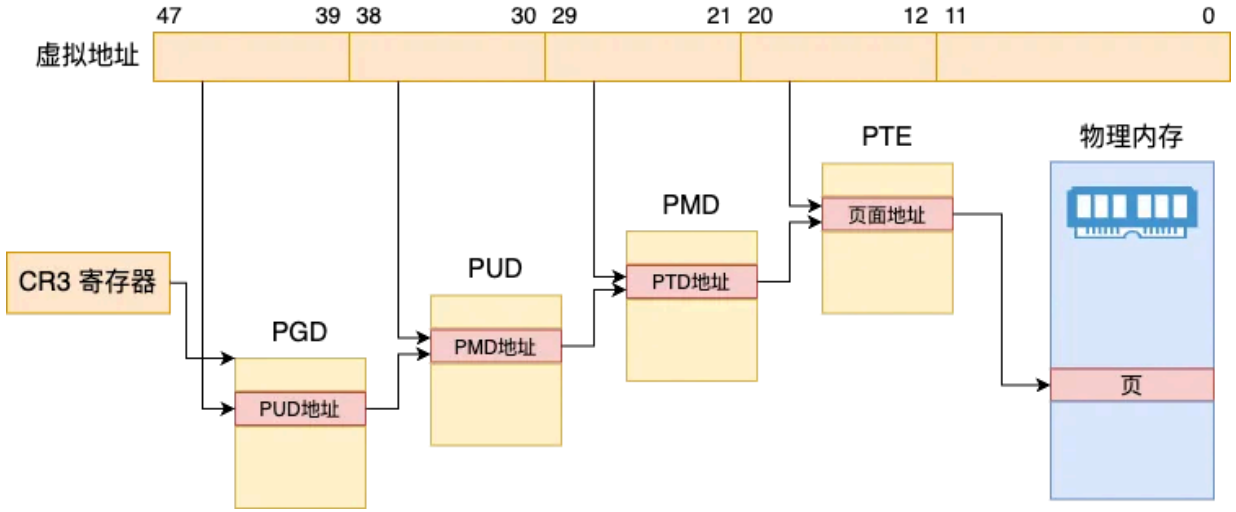
为了更好了解 HugePage，我们需要温习一下内核的页表机制。

在这个机制中有两个前提知识点，那就是

- 第一、应用程序申请内存时不会分配物理内存，访问触发缺页中断时才分配！
- 第二、页是内核分配物理内存的最小单位！

我们应用程序使用的都是虚拟内存地址。在程序实际运行的时候，需要转换成实际的物理地址。如果转换后的物理地址所在的页面正好存在，那么直接访问就可以了。如果页面不存在，那么需要触发缺页中断并申请一个完整的页面后再供应用程序继续访问。页的最小单位是 4 KB。

在《深入理解Linux进程与内存》里的第六章「进程如何使用内存」中，我们提到过 Linux 将虚拟地址到物理地址中用到的四级页表机制。



内核四级页表机制把 64 位的内存地址范围分成了几段。

- 第 63-48 位，额。。64位内存地址太大了，这段属于废弃不用的。
- 第 39-47（9）位指定在一级页表 PGD(Page Global Directory, 页全局目录)中索引位置
- 第 30-38（9）位指定在二级页表 PUD(页上级目录)中索引位置
- 第 21-29（9）位指定在对应三级页表 PMD 中索引位置
- 第 12-20（9）位指定在四级页表 PTE 中索引位置

> 大家注意下，每一级页表管理的地址范围都是 9 个位。为啥是 9 ，不是 8 ，也不是 10。原因是为了将数据结构对齐到 4 KB。这样具体的一个 PGD/PUD/PMD/PTE，保存着 2 的 9 次方， 512 个 64 位物理地址（8个字节）。512 * 8 = 正好是 4 KB。

在将某进程的一个具体的 64 位的虚拟内存地址转换为物理地址时，首先按照上述地址范围把虚拟地址切分成几段。然后经过下面几步转换成物理地址。

- 第一步：从 CPU 中名为 CR3 的寄存器中找到当前进程的一级页表 PGD 的地址
- 第二步：以虚拟地址中的 39 ~ 47 位作为索引，找到 PUD 所在的内存地址
- 第三步：再以虚拟地址中的 30 ~ 38 位作为索引，找到 PMD 所在的内存地址

- 第四步：再以虚拟地址中的 21 ~ 29 位作为索引，找到 PTE 所在的内存地址
- 第五步：再以虚拟内存地址的 0 ~ 11 位作为物理内存页的偏移量，得到最终的物理地址

Linux分页机制就带领大家简单回忆这么一下。今天我们的重点是想说页表机制带来的额外的问题。

页表是存在内存里的。完成一个虚拟地址转换的过程中需要把当前虚拟地址对应的四个页表全部找出来才能完成虚拟地址到物理地址的转换。**那就是一次内存 IO 光是虚拟地址到物理地址的转换就要去内存查 4 次页表**。再算上真正的内存访问，最坏情况下需要 5 次内存 IO 才能获取一个内存数据!

为了提升地址转换效率。既然进行地址转换需要的内存 IO 次数多，且耗时。那么干脆就和 CPU 的 L1、L2、L3 的缓存思想一样，在 CPU 里把页表中的数据尽可能地缓存起来不就行了么，

所以 CPU 硬件中有个 TLB(Translation Lookaside Buffer) 模块，专门用于加速虚拟地址到物理地址转换速度的缓存。其访问速度非常快，和寄存器相当，比 L1 访问还快。

虽然有了 TLB 加速的方案，但这个方案并不是万能的。最大的缺点是 TLB 太小了。一般的 CPU 中 L1 TLB 一般也就几十个条目容量，L2 TLB 一般也就小几千。

再看需求端，我们假设每个进程需要 40 GB 物理内存，那换算成 4 KB 页面的话就是大约 1000 万个页面，也就对应 1000 万个页表条目。TLB 里这点点容量还是捉襟见肘。

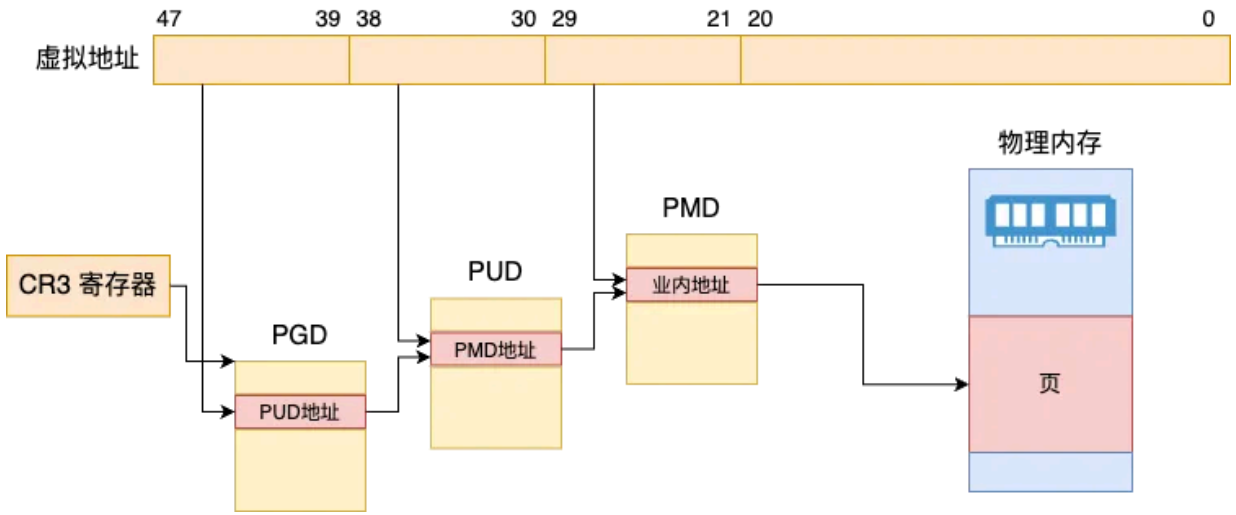
正因为在四级页表下有这样潜在的性能隐患。所以 Oracle 这种内存密集型的应用就推荐配置 HugePage 来提高它的运行性能了。

二、HugePage 如何使用

可见，四级页表最大的问题是在于页面太多时性能较差。页面一多，管理这些页面的页表项就多，TLB缓存命中率就会很差。那如果能把页面数量给降下来，TLB 缓存命中率一定会有大幅度的提升。

假如说我们把 4 KB 的页面换成 2 MB 的页面，那么同样对于 40 GB 物理内存消耗，那仅仅只需要 2 万个页面就够了。相比于原来的 1000 万 降低到了 500 之一。

另外这样不光是 TLB 缓存命中率会有大幅度的提升。内核的虚拟地址转换时的页表机制也可以简化成下面这样的三级页表，少了一次转换开销。



所以，一个结论是**把 4 KB 的页面换成 2 MB 的页面，可以大幅度提升虚拟地址转换物理地址时的性能！！**

那么，如果你想获取这个性能提升的话，该如何操作呢？

第一步首先是大页的预留。

预留的方式分为启动阶段预留和运行时预留。

对于启动阶段预留，需要修改 Linux 内核的启动参数。编辑/boot/grub/grub.cfg 文件找到启动参数行（不同的发行版可能修改方式会有一些出入）。添加以下内容，指定 HugePage 的页面大小，指定预留的大页数量。：

```
hugepagesz=2M hugepages=512
```

对于运行时预留，直接修改内核 hugetlbfs 暴露出来的伪文件即可。

```
```

```
// 预留特定size的大页
echo 5 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

第二步是大页的申请

申请的时候，先打开通过 open 打开 hugepage 伪文件句柄，再通过 mmap 来申请即可。

```
int main(){
 // 打开 hugepage 句柄
 fd = open("/mnt/huge/hugepage...", O_CREAT|O_RDWR);

    // 申请大页
    addr = mmap(0, MAP_LENGTH, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
}
```

这样，你的应用程序就能享受 TLB 缓存命中率提升带来的飞翔感觉了。

三、内核启动时 HugePage 处理

咱们「开发内功修炼」公众号的风格是不仅要会用，还要懂内部原理。接下来飞哥再来带你看下内核是如何管理 HugePage 的！

3.1 回顾普通页的伙伴系统

在《深入理解Linux进程与内存》里的第五章「系统物理内存初始化」中介绍过，

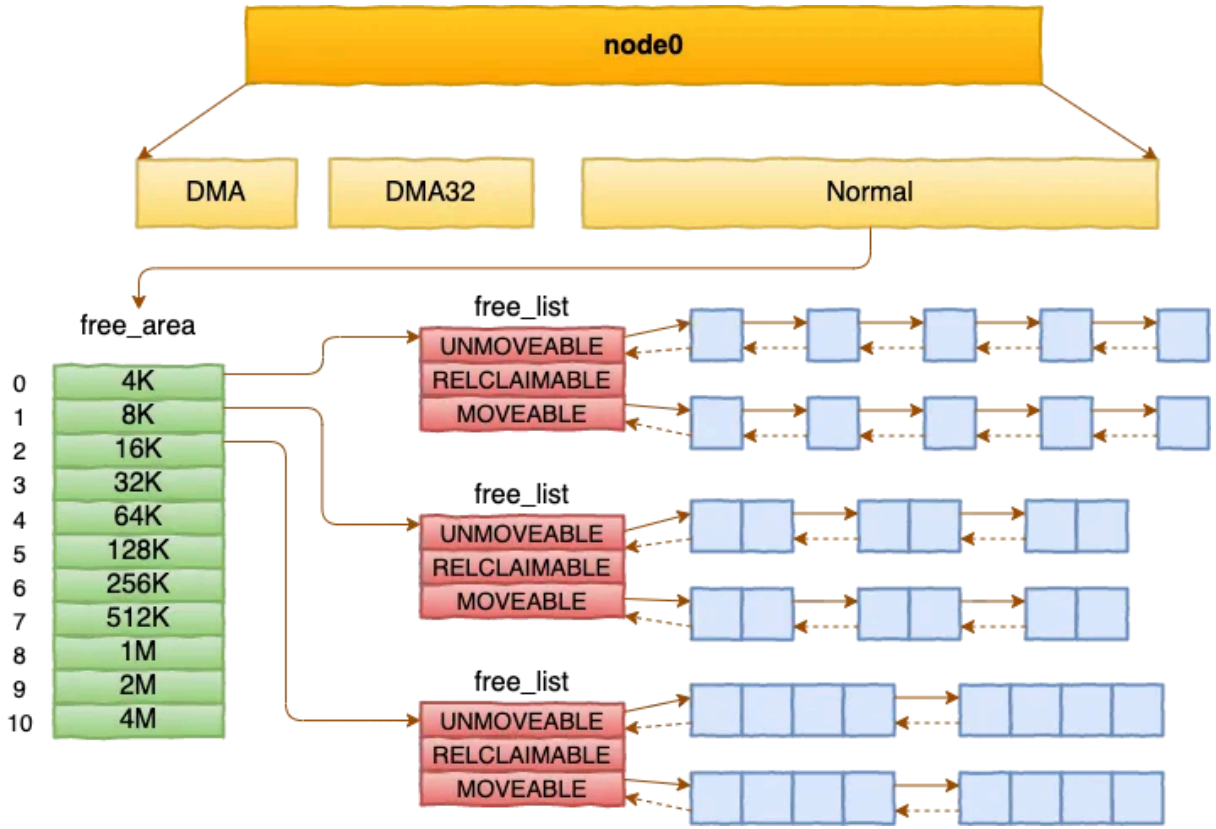
- 内核先是通过固件 ACPI E820 规范探测安装的内存的物理地址范围
- 将探测到的内存交给 memblock 初期内存分配器来管理，同时会再读取 ACPI 中的 SRAT 表获取 NUMA 信息

- 接着在初期内存分配器中申请管理所有页面的 struct page 对象（一个 struct page 一般是 64 字节）
- 最后释放其余的可用内存交给伙伴系统来管理

```
start_kernel
-> setup_arch
---> e820__memory_setup   // 内核把物理内存检测保存从boot_params.e820_table保存到e820_table中，并打印出来
---> e820__memblock_setup // 根据e820信息构建memblock内存分配器，开启调试能打印
---> initmem_init         // 内存中 NUMA 机制初始化）
---> x86_init.paging.pagetable_init（native_pagetable_init）
-----> paging_init        // 页管理机制的初始化
-> mm_init
---> mem_init
-----> memblock_free_all  // 向伙伴系统移交控制权
```

```
// file:include/linux/mmzone.h
struct zone {
 ......
 // zone的名称
 const char  *name;


 // 管理zone下面所有页面的伙伴系统
 struct free_area free_area[MAX_ORDER];
 ......
}
```

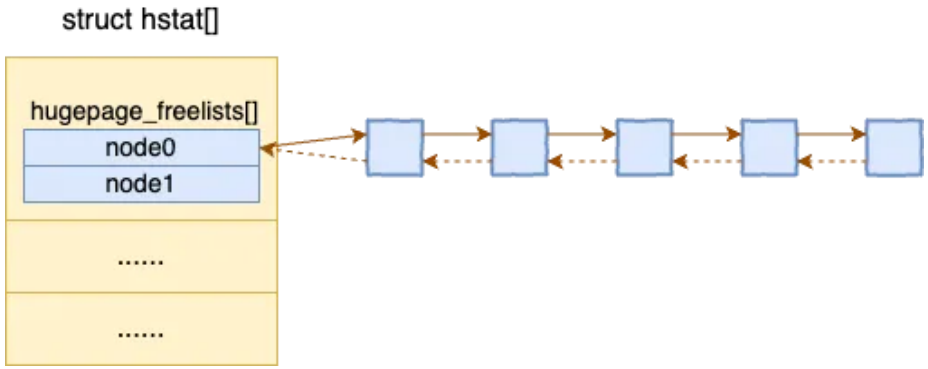


3.2 空闲 HugePage 的管理

相比伙伴系统中 4KB 页面的管理，内核对 HugePage 页面的管理要简单许多。内核中维持一个各种 HugePage 页面（内核支持多种大小的 HugePage，不仅仅只有 2 MB）的 struct hstate 数组。

```
// file:mm/hugetlb.c
struct hstate hstates[HUGE_MAX_HSTATE];
```

在每一个 hstate 成员内，有一个空闲链表 hugepage_freelists，会把所有的空闲页面给串起来。



我们来看大致看下空闲页面的初始化过程。内核启动过程中，还会按照一定的顺序执行初始化函数。HugePage 的初始化函数 hugetlb_init 通过 subsys_initcall 注册。

```
// file:mm/hugetlb.c
subsys_initcall(hugetlb_init);
```

这样内核启动的时候，就会执行到 hugetlb_init 进行 HugePage 的初始化。

```
// file:mm/hugetlb.c
static int __init hugetlb_init(void)
{
 ...
 // 初始化默认大页 state，空闲大内存页链表 hugepage_freelists
 hugetlb_add_hstate(HUGETLB_PAGE_ORDER);
 // 申请大内存页，并且保存到 hugepage_freelists 链表中
 hugetlb_init_hstates();
 ...

 // 创建/sys/kernel/mm/hugepages相关目录文件
 hugetlb_sysfs_init();
```

```
 // 创建/sys/device/system/node/node*/hugepages相关目录文件
 hugetlb_register_all_nodes();
 ...
}
```

hugetlb_init 函数主要完成两个工作：

第一：初始化默认大页 state。在 Linux 中是支持多种规格的大页的，存在一个全局变量 states 数组，其中每一个元素都对应一个规格的大页的管理数据结构，包括所有空闲页面管理用的链表 hugepage_freelists。

第二：为系统申请空闲的大内存页，并且保存到空闲链表 hugepage_freelists 中。

第三：创建 hugetlbfs 相关伪文件，如 /sys/kernel/mm/hugepages、/sys/device/system/node/node*/hugepages。用户后续可以通过这些伪文件来和内核交互。

我们来重点看下申请空闲大内存页的逻辑，这是依次调用 hugetlb_init_hstates -> hugetlb_hstate_alloc_pages，在执行到 hugetlb_hstate_alloc_pages_onenode 中完成的。

```
// file:mm/hugetlb.c
static void __init hugetlb_hstate_alloc_pages_onenode(struct hstate *h, int nid)
{
 ...
 for (i = 0; i < h->max_huge_pages_node[nid]; ++i) {
  page = alloc_fresh_huge_page(h, gfp_mask, nid,
     &node_states[N_MEMORY], NULL);
  if (page)
   break;
 }

 free_huge_page(page);
```

```
 return 1;
}
```

其中 alloc_fresh_huge_page 是在申请页面，free_huge_page 会将其放到空闲链表 hugepage_freelists 中。

四、mmap 申请内存

大页的内存申请内核工作原理大概分三步：

- 第一先是要打开 HugePage 伪文件句柄，
- 第二是通过 mmap 申请大页
- 第三是在访问缺页中断时实际申请真正的物理大页

```
int main(){
 // 打开 hugepage 句柄
 fd = open("/mnt/huge/hugepage...", O_CREAT|O_RDWR);

    // 申请大页
    addr = mmap(0, MAP_LENGTH, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
}
```

4.1 打开 HugePage 伪文件句柄

调用 open 打开 hugetlbfs 下的文件时，会执行到 hugetlb_file_setup 函数，在这里会给申请文件内核对象，为它指定它所绑定的各种 operations 方法。

```
```

```
// file:fs/hugetlbfs/inode.c
struct file *hugetlb_file_setup(const char *name, ...)
{
 ...
 file = alloc_file_pseudo(inode, mnt, name, O_RDWR,
     &hugetlbfs_file_operations);
 ...
}
```

其中 hugetlbfs_file_operations 指定了这类文件的各种具体的方法。

```
const struct file_operations hugetlbfs_file_operations = {
 .read_iter  = hugetlbfs_read_iter,
 .mmap    = hugetlbfs_file_mmap,
 .fsync    = noop_fsync,
 .get_unmapped_area = hugetlb_get_unmapped_area,
 ......
};
```

这样当对该文件执行 mmap 操作时，就会调用到内核中的 hugetlbfs_file_mmap 函数。

4.2 mmap 分配虚拟内存

mmap 系统调用执行经过如下的复杂调用链后，最终会调用到 file 内核对象的 map 方法。

```
mmap                        // offset转成页为单位
+-- sys_mmap_pgoff          // 通过fd获取file
```

```
    +-- vm_mmap_pgoff       // 信号量保护，映射完成后populate
        +-- do_mmap_pgoff   // 简单封装
            +-- do_mmap     // 映射长度页对齐，prot和flags检查，设置vm_flags，获取映射虚拟地址
                +-- mmap_region      // 地址空间检查，vma_merge，vma分配及初始化
                    |-- call_mmap    // 文件映射，简单封装
                    |   +-- file->f_op->mmap    // 调用实际文件的mmap方法
                    ....
```

执行到的 file->f_op->mmap 是一个函数指针。在上一小节我们看到对于 hugetlbfs 下的文件，其 mmap 函数指针对应的是 hugetlbfs_file_mmap 函数。

```
// file:fs/hugetlbfs/inode.c
static int hugetlbfs_file_mmap(struct file *file, struct vm_area_struct *vma)
{
 ...
 // 为映射分配所需的大页框
 hugetlb_reserve_pages(inode,
    vma->vm_pgoff >> huge_page_order(h),
    len >> huge_page_shift(h), vma,
    vma->vm_flags)
 ...
}
```

在该函数中主要做的就是调用 hugetlb_reserve_pages 预留大页。

4.3 缺页中断处理

当缺页中断发生时，内核会调用到 handle_mm_fault 函数。在这里对于 HugePage、普通缺页、透明大页的处理都是不一样的。

```
// file:mm/memory.c
vm_fault_t handle_mm_fault(struct vm_area_struct *vma, ...)
{
 ...
 // 是否是大页缺页
 if (is_vm_hugetlb_page(vma))
  ret = hugetlb_fault(vma->vm_mm, vma, address, flags);
 else
  // 普通的缺页中断，包括透明大页也都在这里
  ret = __handle_mm_fault(vma, address, flags);
 ...
}
```

HugePage 缺页会执行到 hugetlb_fault 函数，然后再调用 hugetlb_no_page。

```
static vm_fault_t hugetlb_no_page(struct mm_struct *mm, ...)
{
 page = find_lock_page(mapping, idx);
 if (!page) {
  ...
  // 1. 从空闲大内存页链表 hugepage_freelists 中申请一个大内存页
  page = alloc_huge_page(vma, haddr, 0);
 }

 // 2. 通过大内存页的物理地址生成页表表项
```

```
    new_pte = make_huge_pte(vma, page, ((vma->vm_flags & VM_WRITE)
                && (vma->vm_flags & VM_SHARED)));
    // 3. 将页表表项挂到页表中
    set_huge_pte_at(mm, haddr, ptep, new_pte);
    ...
    return ret;
}
```

在 hugetlb_no_page 中主要做了两件事：

- 第一件：调用 alloc_huge_page 从空闲链表中 hugepage_freelists 摘一个页面下来
- 第二件：设置页表。先是通过大内存页的物理地址生成页表表项，再将页表表项挂到页表中

这样，应用程序就申请到了大页物理内存了。

五、总结

我们应用程序使用的都是虚拟内存地址。在程序实际运行的时候，需要转换成实际的物理地址。

为了提升地址转换效率。CPU 硬件中设计有 TLB 模块，用于缓存内存中的页表项，加速访问。这样 CPU 在执行虚拟地址转换时，就可以避免很多的内存访问，极大地提升效率。

但可惜的是 TLB 缓存容量都不大，一般 CPU 中 L1 TLB 一般也就几十个条目容量，L2 TLB 一般也就小几千，我手头的一台服务器 L2 TLB 才是 1500 个条目。

如果使用 4 KB 的小页面。假设每个进程需要 40 GB 物理内存，每个页面 4 KB，那就是大约 1000 万个页面，也就要管理 1000 万个页表条目。区区 1500 个 TLB 缓存条目空间，显然是捉襟见肘。

如果使用 2 MB 的 HugePage， 40 GB / 2 MB，只需要 2 万个页面。管理的页表条目一下子从 1000 万下降到了 2万，这样 1500 个条目就挺充裕的了。

使用 HugePage 能帮助 TLB 缓存命中率得到了大大的提升。应用程序在执行虚拟地址到物理地址的转换过程中就会节约许多开销。

Oracle 数据库是一个存储密集型的应用，会申请大量的内存，也会涉及到大量的内存访问。那么用 HugePage 优化一下性能的话，对于它来讲再合适不过了。

要补充提的一点是，如果你的应用程序使用的内存很小，例如只有几百 M，那建议你还是不要费这个劲儿了，提升不了多少。