注： 本文档内容解释多数截自网络。。 Leonid\_Wan

PRO#DD 设备中查看lowmemorykiller参数：

root@android:/sys/module/lowmemorykiller/parameters # ls

adj

check\_filepages

cost

debug\_level

fork\_boost

minfile

minfree

root@android:/sys/module/lowmemorykiller/parameters # cat \*

0,2,4,7,9,12

1

32

2

1

2048,3072,4096,6144,10854,14132

2048,3072,4096,6144,10854,14132

本文档中所涉及到的文件路径如下：把这些文件打开先。。

kernel/drivers/staging/android/lowmemorykiller.c

kernel/mm/oom\_kill.c

system/core/rootdir/init.rc

customize/base/services/java/com/android/server/am/ActivityManagerService.java

customize/base/services/java/com/android/server/am/ProcessList.java

kernel/include/linux/mm.h

kernel/mm/vmscan.c

**Android Low memory killer**

 Android中，进程的生命周期都是由系统控制的，即使用户关掉了程序，进程依然是存在于内存之中。这样设计的目的是为了下次能快速启动。当然，随着系统运行时间的增长，内存会越来越少。Android Kernel 会定时执行一次检查，杀死一些进程，释放掉内存。

      那么，如何来判断，那些进程是需要杀死的呢？答案就是我们的标题：Low memory killer机制。

Android 的Low memory killer是基于linux的OOM（out of memory）  规则改进而来的。 OOM通过一些比较复杂的评分机制，对进程进行打分，然后将分数高的进程判定为bad进程，杀死并释放内存。

android的low memory killer和标准的oom killer很多想法是一致的，只不过low memory killer作为一个shrinker实现；而oom killer则在分配内存时候被调用（如果内存资源很紧张）

     Low memory killer 主要是通过进程的oom\_adj 来判定进程的重要程度。oom\_adj的大小和进程的类型以及进程被调度的次序有关。

     对于Android4.0以前，Low memory killer 的具体实现可参看：kernel/drivers/misc/lowmemorykiller.c

    对于Android4.0后，Low memory killer 的具体实现可参看kernel/drivers/staging/lowmemorykiller.c

     其原理很简单，在linux中，存在一个kswapd的内核线程，当linux回收存放分页的时候，kswapd线程将会遍历一张shrinker链表，并执行回调。shrinker是在中kernel\include\linux\mm.h定义的，定义如下：

/\*

 \* A callback you can register to apply pressure to ageable caches.

 \*

 \* 'sc' is passed shrink\_control which includes a count 'nr\_to\_scan'

 \* and a 'gfpmask'.  It should look through the least-recently-used

 \* 'nr\_to\_scan' entries and attempt to free them up.  It should return

 \* the number of objects which remain in the cache.  If it returns -1, it means

 \* it cannot do any scanning at this time (eg. there is a risk of deadlock).

 \*

 \* The 'gfpmask' refers to the allocation we are currently trying to

 \* fulfil.

 \*

 \* Note that 'shrink' will be passed nr\_to\_scan == 0 when the VM is

 \* querying the cache size, so a fastpath for that case is appropriate.

 \*/

struct shrinker {

int (\*shrink)(struct shrinker \*, struct shrink\_control \*sc);

int seeks; /\* seeks to recreate an obj \*/

/\* These are for internal use \*/

struct list\_head list;

long nr; /\* objs pending delete \*/

};

#define DEFAULT\_SEEKS 2 /\* A good number if you don't know better. \*/

extern void register\_shrinker(struct shrinker \*);

extern void unregister\_shrinker(struct shrinker \*);

register\_shrinker()和unregister\_shrinker()则是在kernel\mm\Vmscan.c中定义的

/\*

 \* Add a shrinker callback to be called from the vm

 \*/

void register\_shrinker(struct shrinker \*shrinker)

{

shrinker->nr = 0;

down\_write(&shrinker\_rwsem);

list\_add\_tail(&shrinker->list, &shrinker\_list);

up\_write(&shrinker\_rwsem);

}

EXPORT\_SYMBOL(register\_shrinker);

/\*

 \* Remove one

 \*/

void unregister\_shrinker(struct shrinker \*shrinker)

{

down\_write(&shrinker\_rwsem);

list\_del(&shrinker->list);

up\_write(&shrinker\_rwsem);

}

EXPORT\_SYMBOL(unregister\_shrinker);

static inline int do\_shrinker\_shrink(struct shrinker \*shrinker,

    struct shrink\_control \*sc,

    unsigned long nr\_to\_scan)

{

sc->nr\_to\_scan = nr\_to\_scan;

return (\*shrinker->shrink)(shrinker, sc);

}

所以只要注册一个Shrinker结构体，在内存分页回收时，系统将调用Shrinker结构体的函数指针

int (\*shrink)(struct shrinker \*, struct shrink\_control \*sc);,

下面我们来看看其实现Low memory killer的实现。

在lowmemorykiller.c中，首先定义一个以lowmem\_shrink为回调函数的指针的shrinker结构体lowmem\_shrinker

static struct shrinker lowmem\_shrinker = {

.shrink = lowmem\_shrink,

.seeks = DEFAULT\_SEEKS \* 16

};

然后对lowmem\_shrink进行注册

static int \_\_init lowmem\_init(void)

{

task\_free\_register(&task\_nb);

register\_shrinker(&lowmem\_shrinker);

return 0;

}

static void \_\_exit lowmem\_exit(void)

{

unregister\_shrinker(&lowmem\_shrinker);

task\_free\_unregister(&task\_nb);

}

module\_param\_named(cost, lowmem\_shrinker.seeks, int, S\_IRUGO | S\_IWUSR);

module\_param\_array\_named(adj, lowmem\_adj, int, &lowmem\_adj\_size,

S\_IRUGO | S\_IWUSR);

module\_param\_array\_named(minfree, lowmem\_minfree, uint, &lowmem\_minfree\_size,

S\_IRUGO | S\_IWUSR);

module\_param\_named(debug\_level, lowmem\_debug\_level, uint, S\_IRUGO | S\_IWUSR);

module\_init(lowmem\_init);

module\_exit(lowmem\_exit);

通过module\_init(lowmem\_init)注册，在内存分页回收时，系统将调用lowmem\_shrink（）函数。

在lowmemorykiller.c中，定义了两对数组，以表示当系统内存低于某一值，就在低于某一优先级别的线程中，选择一占用内存最大的进程，然后杀死该进程，以释放其占用的内存。

这两对数组的定义如下：

static int lowmem\_adj[6] = {

0,

1,

6,

12,

};

static int lowmem\_adj\_size = 4;

static size\_t lowmem\_minfree[6] = {

3 \* 512, /\* 6MB \*/

2 \* 1024, /\* 8MB \*/

4 \* 1024, /\* 16MB \*/

16 \* 1024, /\* 64MB \*/

};

static int lowmem\_minfree\_size = 4;

这就说明，当系统的空闲空间下降到16 \* 1024个页面时，oom\_adj值为12或者更大的进程将被Kill掉;当系统的空闲空间下降到4 \* 1024个页面时，oom\_adj值为6或者更大的进程将被Kill掉；依此类推。

其实更简明就是说：task\_struct->signal\_struct->oom\_adj越大的越优先被Kill，

同oom\_adj的进程占用物理内存最多的那个进程会被优先Kill。

进程描述符中的signal\_struct->oom\_adj表示当内存短缺时进程被选择并Kill的优先级，值越大越可能被选中。当某个进程被选中后，内核会发送SIGKILL信号将其Kill掉。

实际上，Low Memory Killer驱动程序会认为被用于缓存的存储空间都要被释放，但是，如果很大一部分缓存存储空间处于被锁定的状态，那么这将是一个非常严重的错误，并且当正常的oom killer被触发之前，进程是不会被Kill掉的。

lowmem\_adj和lowmem\_minfree上面的值只是个默认值，我们还可以通过以写文件的方式进行重写。我想该功能应该是通过lowmemorykiller.c中的如下代码开启该功能的

module\_param\_named(cost, lowmem\_shrinker.seeks, int, S\_IRUGO | S\_IWUSR);

module\_param\_array\_named(adj, lowmem\_adj, int, &lowmem\_adj\_size,

S\_IRUGO | S\_IWUSR);

module\_param\_array\_named(minfree, lowmem\_minfree, uint, &lowmem\_minfree\_size,

S\_IRUGO | S\_IWUSR);

module\_param\_named(debug\_level, lowmem\_debug\_level, uint, S\_IRUGO | S\_IWUSR);

在Android4.0以前，一直是通过system/rootdir/init.rc进行重写，重写配置的，

比如：

# Define the oom\_adj values for the classes of processes that can be

# killed by the kernel.  These are used in ActivityManagerService.

    setprop ro.FOREGROUND\_APP\_ADJ 0

    setprop ro.VISIBLE\_APP\_ADJ 1

    setprop ro.SECONDARY\_SERVER\_ADJ 2

    setprop ro.BACKUP\_APP\_ADJ 2

    setprop ro.HOME\_APP\_ADJ 4//<category android:name="android.intent.category.HOME"/>

    setprop ro.HIDDEN\_APP\_MIN\_ADJ 7

    setprop ro.CONTENT\_PROVIDER\_ADJ 14

    setprop ro.EMPTY\_APP\_ADJ 15

# Define the memory thresholds at which the above process classes will

# be killed.  These numbers are in pages (4k).

    setprop ro.FOREGROUND\_APP\_MEM 1536

    setprop ro.VISIBLE\_APP\_MEM 2048

    setprop ro.SECONDARY\_SERVER\_MEM 4096

    setprop ro.BACKUP\_APP\_MEM 4096

    setprop ro.HOME\_APP\_MEM 4096

    setprop ro.HIDDEN\_APP\_MEM 5120

    setprop ro.CONTENT\_PROVIDER\_MEM 5632

    setprop ro.EMPTY\_APP\_MEM 6144

# Write value must be consistent with the above properties.

# Note that the driver only supports 6 slots, so we have HOME\_APP at the

# same memory level as services.

    write /sys/module/lowmemorykiller/parameters/adj 0,1,2,7,14,15

    write /proc/sys/vm/overcommit\_memory 1

    write /proc/sys/vm/min\_free\_order\_shift 4

    write /sys/module/lowmemorykiller/parameters/minfree 1536,2048,4096,5120,5632,6144

# Set init its forked children's oom\_adj.

    write /proc/1/oom\_adj -16

在com.android.server.am.ActivityManagerService.java中，通过以下方式读取这些值。

可以在ActivityManagerService.java中清楚的看到：

    static final int EMPTY\_APP\_ADJ;

    static final int HIDDEN\_APP\_MAX\_ADJ;

    static final int HIDDEN\_APP\_MIN\_ADJ;

    static final int HOME\_APP\_ADJ;

    static final int BACKUP\_APP\_ADJ;

    static final int SECONDARY\_SERVER\_ADJ;

    static final int HEAVY\_WEIGHT\_APP\_ADJ;

    static final int PERCEPTIBLE\_APP\_ADJ;

    static final int VISIBLE\_APP\_ADJ;

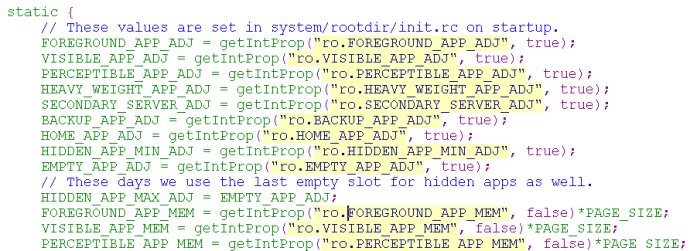
    static final int FOREGROUND\_APP\_ADJ;

    static final int CORE\_SERVER\_ADJ = -12;

    static final int SYSTEM\_ADJ = -16;

   ActivityManagerService定义各种进程的oom\_adj,CORE\_SERVER\_ADJ代表一些核心的服务的omm\_adj,数值为-12，由前面的分析可知道，这类进程永远也不会被杀死。

其他未赋值的都在static块中进行了初始化，是通过system/rootdir/init.rc进行配置的：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=4d66a3cb0100prfe&url=http://s9.sinaimg.cn/orignal/4d66a3cbt9fc6f0a21868)

以上ADJ在customize/base/services/java/com/android/server/am/ProcessList.java中有定义

/\*\*

\* Activity manager code dealing with processes.

\*/

class ProcessList {

// The minimum time we allow between crashes, for us to consider this

// application to be bad and stop and its services and reject broadcasts.

static final int MIN\_CRASH\_INTERVAL = 60\*1000;

// OOM adjustments for processes in various states:

// This is a process only hosting activities that are not visible,

// so it can be killed without any disruption.

**static final int HIDDEN\_APP\_MAX\_ADJ = 15;**

**static int HIDDEN\_APP\_MIN\_ADJ = 9;**

// The B list of SERVICE\_ADJ -- these are the old and decrepit

// services that aren't as shiny and interesting as the ones in the A list.

**static final int SERVICE\_B\_ADJ = 8;**

// This is the process of the previous application that the user was in.

// This process is kept above other things, because it is very common to

// switch back to the previous app. This is important both for recent

// task switch (toggling between the two top recent apps) as well as normal

// UI flow such as clicking on a URI in the e-mail app to view in the browser,

// and then pressing back to return to e-mail.

**static final int PREVIOUS\_APP\_ADJ = 7;**

//[framework] 20120324, Leo\_Hsu, begin, swap SERVICE\_ADJ and HOME\_APP\_ADJ so that Home will not be killed easier than Services

// This is a process holding the home application -- we want to try

// avoiding killing it, even if it would normally be in the background,

// because the user interacts with it so much.

//static final int HOME\_APP\_ADJ = 6;

**static final int HOME\_APP\_ADJ = 5;**

// This is a process holding an application service -- killing it will not

// have much of an impact as far as the user is concerned.

// static final int SERVICE\_ADJ = 5;

**static final int SERVICE\_ADJ = 6;**

//[framework] 20120324, Leo\_Hsu, end

// This is a process currently hosting a backup operation. Killing it

// is not entirely fatal but is generally a bad idea.

**static final int BACKUP\_APP\_ADJ = 4;**

// This is a process with a heavy-weight application. It is in the

// background, but we want to try to avoid killing it. Value set in

// system/rootdir/init.rc on startup.

**static final int HEAVY\_WEIGHT\_APP\_ADJ = 3;**

// This is a process only hosting components that are perceptible to the

// user, and we really want to avoid killing them, but they are not

// immediately visible. An example is background music playback.

**static final int PERCEPTIBLE\_APP\_ADJ = 2;**

// This is a process only hosting activities that are visible to the

// user, so we'd prefer they don't disappear.

**static final int VISIBLE\_APP\_ADJ = 1;**

// This is the process running the current foreground app. We'd really

// rather not kill it!

**static final int FOREGROUND\_APP\_ADJ = 0;**

// This is a system persistent process, such as telephony. Definitely

// don't want to kill it, but doing so is not completely fatal.

**static final int PERSISTENT\_PROC\_ADJ = -12;**

// The system process runs at the default adjustment.

**static final int SYSTEM\_ADJ = -16;**

// Memory pages are 4K.

**static final int PAGE\_SIZE = 4\*1024;**

// The minimum number of hidden apps we want to be able to keep around,

// without empty apps being able to push them out of memory.

**static final int MIN\_HIDDEN\_APPS = 2;**

// The maximum number of hidden processes we will keep around before

// killing them; this is just a control to not let us go too crazy with

// keeping around processes on devices with large amounts of RAM.

//[framework] 20120229, Leo\_Hsu, begin, Per request from Azar Huang, set MAX\_HIDDEN\_APPS to 10 for Sense 4.0a

**static int MAX\_HIDDEN\_APPS = 15;**

在Android4.0中，则是通过com.android.server.am.ProcessList进行重写，重行配置的.

摘要如下：

    // These are the various interesting memory levels that we will give to

    // the OOM killer.  Note that the OOM killer only supports 6 slots, so we

    // can't give it a different value for every possible kind of process.

    private final int[] mOomAdj = new int[] {

            FOREGROUND\_APP\_ADJ, VISIBLE\_APP\_ADJ, PERCEPTIBLE\_APP\_ADJ,

            BACKUP\_APP\_ADJ, HIDDEN\_APP\_MIN\_ADJ, EMPTY\_APP\_ADJ

    };

    // These are the low-end OOM level limits.  This is appropriate for an

    // HVGA or smaller phone with less than 512MB.  Values are in KB.

    private final long[] mOomMinFreeLow = new long[] {

            8192, 12288, 16384,

            24576, 28672, 32768

    };

    // These are the high-end OOM level limits.  This is appropriate for a

    // 1280x800 or larger screen with around 1GB RAM.  Values are in KB.

    private final long[] mOomMinFreeHigh = new long[] {

            32768, 40960, 49152,

            57344, 65536, 81920

    };

...................................................................

  ProcessList() {

        MemInfoReader minfo = new MemInfoReader();

        minfo.readMemInfo();

        mTotalMemMb = minfo.getTotalSize()/(1024\*1024);

        updateOomLevels(0, 0, false);

    }

..............................................................................

    private void updateOomLevels(int displayWidth, int displayHeight, boolean write) {

        // Scale buckets from avail memory: at 300MB we use the lowest values to

        // 700MB or more for the top values.

        float scaleMem = ((float)(mTotalMemMb-300))/(700-300);

        // Scale buckets from screen size.

        int minSize = 320\*480;  //  153600

        int maxSize = 1280\*800; // 1024000  230400 870400  .264

        float scaleDisp = ((float)(displayWidth\*displayHeight)-minSize)/(maxSize-minSize);

        //Slog.i("XXXXXX", "scaleDisp=" + scaleDisp + " dw=" + displayWidth + " dh=" + displayHeight);

        StringBuilder adjString = new StringBuilder();

        StringBuilder memString = new StringBuilder();

        float scale = scaleMem > scaleDisp ? scaleMem : scaleDisp;

        if (scale < 0) scale = 0;

        else if (scale > 1) scale = 1;

        for (int i=0; i<mOomAdj.length; i++) {

            long low = mOomMinFreeLow[i];

            long high = mOomMinFreeHigh[i];

            mOomMinFree[i] = (long)(low + ((high-low)\*scale));

            if (i > 0) {

                adjString.append(',');

                memString.append(',');

            }

            adjString.append(mOomAdj[i]);

            memString.append((mOomMinFree[i]\*1024)/PAGE\_SIZE);

        }

        //Slog.i("XXXXXXX", "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MINFREE: " + memString);

        if (write) {

            writeFile("/sys/module/lowmemorykiller/parameters/adj", adjString.toString());

            writeFile("/sys/module/lowmemorykiller/parameters/minfree", memString.toString());

        }

        // GB: 2048,3072,4096,6144,7168,8192

        // HC: 8192,10240,12288,14336,16384,20480

}

在了解了Low Memory Killer的原理之后，我来看其具体实现，lowmemorykiller.c的lowmem\_shrink函数。

static int lowmem\_shrink(int nr\_to\_scan, gfp\_t gfp\_mask)  
{  
    struct task\_struct \*p;  
    struct task\_struct \*selected = NULL;  
    int rem = 0;  
    int tasksize;  
    int i;  
    int min\_adj = OOM\_ADJUST\_MAX + 1;  
    int selected\_tasksize = 0;  
    int array\_size = ARRAY\_SIZE(lowmem\_adj);  
    int other\_free = global\_page\_state(NR\_FREE\_PAGES);  
    int other\_file = global\_page\_state(NR\_FILE\_PAGES);  
    if(lowmem\_adj\_size < array\_size)  
        array\_size = lowmem\_adj\_size;  
    if(lowmem\_minfree\_size < array\_size)  
        array\_size = lowmem\_minfree\_size;  
    for(i = 0; i < array\_size; i++) {  
        if (other\_free < lowmem\_minfree[i] &&  
            other\_file < lowmem\_minfree[i]) {  
            min\_adj = lowmem\_adj[i];  
            break;  
        }  
    }  
    if(nr\_to\_scan > 0)  
        lowmem\_print(3, "lowmem\_shrink %d, %x, ofree %d %d, ma %d\n", nr\_to\_scan,   
                 gfp\_mask, other\_free, other\_file, min\_adj);  
    rem = global\_page\_state(NR\_ACTIVE\_ANON) +  
        global\_page\_state(NR\_ACTIVE\_FILE) +  
        global\_page\_state(NR\_INACTIVE\_ANON) +  
        global\_page\_state(NR\_INACTIVE\_FILE);  
    if (nr\_to\_scan <= 0 || min\_adj == OOM\_ADJUST\_MAX + 1) {  
        lowmem\_print(5, "lowmem\_shrink %d, %x, return %d\n", nr\_to\_scan, gfp\_mask,   
                 rem);  
        return rem;  
    }  
  
    read\_lock(&tasklist\_lock);  
    for\_each\_process(p) {  
        if (p->oomkilladj < min\_adj || !p->mm)  
            continue;  
        tasksize = get\_mm\_rss(p->mm);  
        if (tasksize <= 0)  
            continue;  
        if (selected) {  
            if (p->oomkilladj < selected->oomkilladj)  
                continue;  
            if (p->oomkilladj == selected->oomkilladj &&  
                tasksize <= selected\_tasksize)  
                continue;  
        }  
        selected = p;  
        selected\_tasksize = tasksize;  
        lowmem\_print(2, "select %d (%s), adj %d, size %d, to kill\n",  
                     p->pid, p->comm, p->oomkilladj, tasksize);  
    }  
    if(selected != NULL) {  
        lowmem\_print(1, "send sigkill to %d (%s), adj %d, size %d\n",  
                     selected->pid, selected->comm,  
                     selected->oomkilladj, selected\_tasksize);  
        force\_sig(SIGKILL, selected);  
        rem -= selected\_tasksize;  
    }  
    lowmem\_print(4, "lowmem\_shrink %d, %x, return %d\n", nr\_to\_scan, gfp\_mask, rem);  
    read\_unlock(&tasklist\_lock);  
    return rem;  
}

首先通过global\_page\_state获取当前剩余内存大小；接着检测lowmem\_adj和lowmem\_minfree数组的大小(元素个数)是否一致，如果不一致则以最小数组的大小为基准；然后根据剩余内存和内存阈值数组lowmem\_minfree查找当前的内存警戒数min\_adj。接着遍历所有进程，找到oom\_adj大于min\_adj并且oom\_adj最大的进程。

进程的oom\_adj小于警戒阈值，则无视。进程的oom\_adj大于等于于警戒阈值，则获取这个进程所占用的内存大小tasksize，如果小于比我们当前选出进程的内存，则无视。如果大于则选中这个进程。

经过for\_each的遍历，selected 就是我们选出要释放掉的bad进程，它具有下面两个条件：

第一、Oom\_adj大于当前警戒阈值并且最大。

第二、在同样大小的oom\_adj中，占用内存最多。

最后，我们释放掉这个进程的内存，通过force\_sig(SIGKILL, selected)来向进程发送一个不可以忽略或阻塞的SIGKILL信号。

在lowmem\_shrink函数中多处用到了global\_page\_state函数。

它被定义在了common/include/linux/vmstat.h中，

static inline unsigned long global\_page\_state(enum zone\_stat\_item item)

{

long x = atomic\_long\_read(&vm\_stat[item]);

#ifdef CONFIG\_SMP

if (x < 0)

x = 0;

#endif

return x;

}

global\_page\_state函数的参数NR\_FREE\_PAGES等使用zone\_stat\_item枚举，被定义在kernel/include/linux/mmzone.h中，具体代码如下：

enum zone\_stat\_item {  
    NR\_FREE\_PAGES,  
    NR\_LRU\_BASE,  
    NR\_INACTIVE\_ANON = NR\_LRU\_BASE,  
    NR\_ACTIVE\_ANON,  
    NR\_INACTIVE\_FILE,  
    NR\_ACTIVE\_FILE,  
#ifdef CONFIG\_UNEVICTABLE\_LRU  
    NR\_UNEVICTABLE,  
    NR\_MLOCK,  
#else  
    NR\_UNEVICTABLE = NR\_ACTIVE\_FILE, /\* 避免编译错误\*/  
    NR\_MLOCK = NR\_ACTIVE\_FILE,  
#endif  
    NR\_ANON\_PAGES,        /\* 匿名映射页面\*/  
    NR\_FILE\_MAPPED,        /\*映射页面\*/  
    NR\_FILE\_PAGES,  
    NR\_FILE\_DIRTY,  
    NR\_WRITEBACK,  
    NR\_SLAB\_RECLAIMABLE,  
    NR\_SLAB\_UNRECLAIMABLE,  
    NR\_PAGETABLE,  
    NR\_UNSTABLE\_NFS,  
    NR\_BOUNCE,  
    NR\_VMSCAN\_WRITE,  
    NR\_WRITEBACK\_TEMP,    /\* 使用临时缓冲区\*/  
#ifdef CONFIG\_NUMA  
    NUMA\_HIT,            /\* 在预定节点上分配\*/  
    NUMA\_MISS,            /\* 在非预定节点上分配\*/  
    NUMA\_FOREIGN,  
    NUMA\_INTERLEAVE\_HIT,  
    NUMA\_LOCAL,            /\* 从本地页面分配\*/  
    NUMA\_OTHER,            /\* 从其他节点分配 \*/  
#endif  
    NR\_VM\_ZONE\_STAT\_ITEMS };

到此结束！