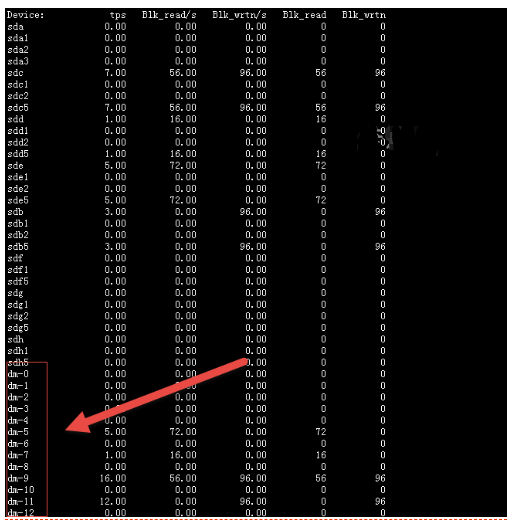
# 背景

在linux系统中你使用一些命令时（例如nmon、iostat如下截图所示），有可能会看到一些名字为dm-xx的设备，那么这些设备到底是什么设备呢，跟磁盘有什么关系呢？

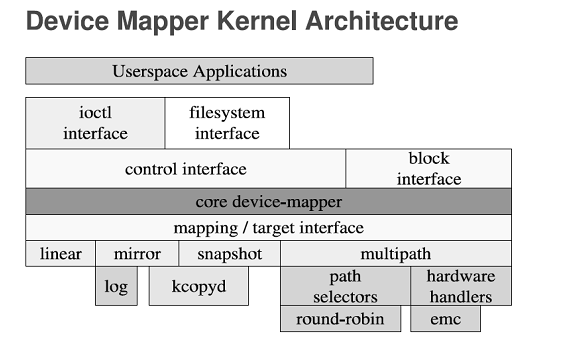
其实dm是Device Mapper的缩写，Device Mapper是Linux 2.6内核中提供的一种**从逻辑设备到物理设备的映射框架机制**，在该机制下，用户可以很方便的根据自己的需要制定实现存储资源的管理策略。

当前比较流行的Linux下的逻辑卷管理器如LVM2(Linux Volume Manager 2 version)、EVMS(Enterprise Volume Management System)、dmraid(Device Mapper Raid Tool)等都是基于该机制实现的。



# 概述

Device Mapper是Linux2.6内核中支持逻辑卷管理的通用设备映射机制，它为实现用于存储资源管理的块设备驱动提供了一个高度模块化的内核架构：



在内核中它通过一个一个模块化的target driver插件实现对IO请求的过滤或者重新定向等工作，当前已经实现的target driver插件包括软raid、软加密、逻辑卷条带、多路径、镜像、快照等，图中linear、mirror、snapshot、multipath表示的就是这些target driver。

Device mapper 进一步体现了在Linux内核设计中策略和机制分离的原则，将所有与策略相关的工作放到用户空间完成，内核中主要提供完成这些策略所需要的机制。Device mapper用户空间相关部分主要负责配置具体的策略和控制逻辑，比如逻辑设备和哪些物理设备建立映射，怎么建立这些映射关系等等，而具体过滤和重定向IO请求的工作由内核中相关代码完成。因此整个device mapper机制由两部分组成：内核空间的device mapper驱动、用户空间的device mapper库以及它提供的dmsetup工具。

# 原理

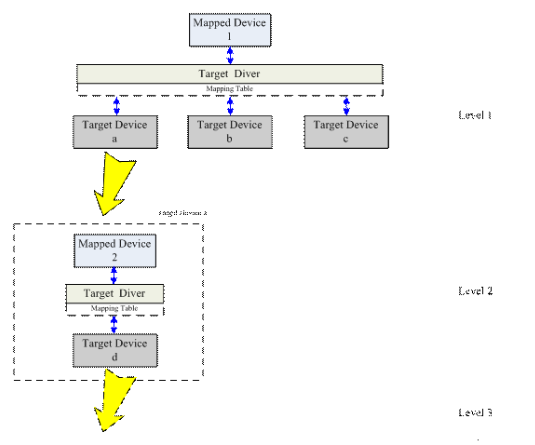
## 内核部分

Device mapper的内核相关代码已经作为Linux 2.6内核发布版的一部分集成到内核源码中了，相关代码在内核源码的driver/md/目录中，其代码文件可以划分为实现device mapper内核中基本架构的文件和实现具体映射工作的target driver插件文件两部分。文章下面的分析结果主要是基于上述源码文件得到的。

**重要概念**

Device mapper在内核中作为一个块设备驱动被注册的，它包含三个重要的对象概念，mapped device、映射表、target device。Mapped device是一个逻辑抽象，可以理解成为内核向外提供的逻辑设备，它通过映射表描述的映射关系和target device建立映射。从Mapped device到一个target device的映射表由一个多元组表示，该多元组由表示mapped device逻辑的起始地址、范围、和表示在target device所在物理设备的地址偏移量以及target类型等变量组成（这些地址和偏移量都是以磁盘的扇区为单位的，即512个字节大小）。Target device 表示的是mapped device所映射的物理空间段，对mapped device所表示的逻辑设备来说，就是该逻辑设备映射到的一个物理设备。Device mapper中这三个对象和target driver插件一起构成了一个可迭代的设备树。在该树型结构中的顶层根节点是最终作为逻辑设备向外提供的 mapped device，叶子节点是target device所表示的底层物理设备。最小的设备树由单个mapped device和target device组成。每个target device都是被mapped device独占的，只能被一个mapped device使用。一个mapped device可以映射到一个或者多个target device上，而一个mapped device又可以作为它上层mapped device的target device被使用，该层次在理论上可以在device mapper架构下无限迭代下去。

图2 Device mapper 内核中各对象的层次关系



在图2中我们可以看到mapped device1通过映射表和a、b、c三个target device建立了映射关系，而target device a又是通过mapped device 2演化过来，mapped device 2通过映射表和target device d建立映射关系。

我们进一步看一下上述三个对象在代码中的具体实现，dm.c 文件定义的 mapped\_device 结构用于表示 mapped device，它主要包括该 mapped device 相关的锁，注册的请求队列和一些内存池以及指向它所对应映射表的指针等域。Mapped device 对应的映射表是由 dm\_table.c 文件中定义的 dm\_table 结构表示的，该结构中包含一个 dm\_target结构数组，dm\_target 结构具体描述了 mapped\_device 到它某个 target device 的映射关系。而在 dm\_table 结构中将这些 dm\_target 按照 B 树的方式组织起来方便 IO 请求映射时的查找操作。Dm\_target 结构具体记录该结构对应 target device 所映射的 mapped device 逻辑区域的开始地址和范围，同时还包含指向具体 target device 相关操作的 target\_type 结构的指针。Target\_type 结构主要包含了 target device 对应的 target driver 插件的名字、定义的构建和删除该类型target device的方法、该类target device对应的IO请求重映射和结束IO的方法等。而表示具体的target device的域是dm\_target中的private域，该指针指向mapped device所映射的具体target device对应的结构。表示target device的具体结构由于不同的target 类型而不同，比如最简单的线性映射target类型对应target device的结构是dm-linear.c文件中定义的linear\_c结构。其定义如下：

struct linear\_c {

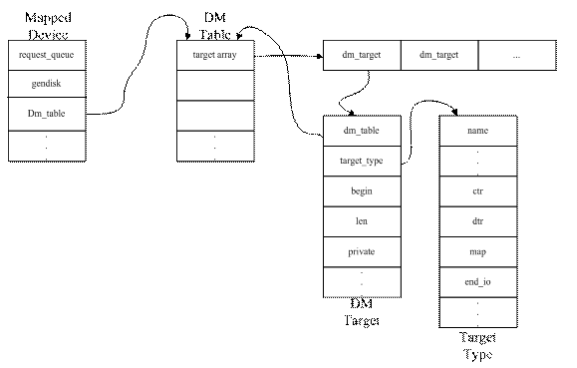
struct dm\_dev \*dev;

sector\_t start;

};

该target device的定义相当简单，就只包括了表示对应物理设备的dm\_dev结构指针和在该物理设备中以扇区为单位的偏移地址start。上述几个数据结构关系如图3所示：

图3 device mapper中几个重要数据结构的关系



**内核中建立过程**

在下面我们结合具体的代码简要介绍下在内核中创建一个mapped device的过程：

1、根据内核向用户空间提供的ioctl接口传来的参数，用dm-ioctl.c文件中的dev\_create函数创建相应的mapped device结构。这个过程很简单，主要是向内核申请必要的内存资源，包括mapped device和为进行IO操作预申请的内存池，通过内核提供的blk\_queue\_make\_request函数注册该mapped device对应的请求队列dm\_request。并将该mapped device作为磁盘块设备注册到内核中。

2、调用dm\_hash\_insert将创建好的mapped device插入到device mapper中的一个全局hash表中，该表中保存了内核中当前创建的所有mapped device。

3、用户空间命令通过ioctl调用table\_load函数，该函数根据用户空间传来的参数构建指定mapped device的映射表和所映射的target device。该函数先构建相应的dm\_table、dm\_target结构，再调用dm-table.c中的dm\_table\_add\_target函数根据用户传入的参数初始化这些结构，并且根据参数所指定的target类型，调用相应的target类型的构建函数ctr在内存中构建target device对应的结构，然后再根据所建立的dm\_target结构更新dm\_table中维护的B树。上述过程完毕后，再将建立好的dm\_table添加到mapped device的全局hash表对应的hash\_cell结构中。

4、最后通过ioctl调用do\_resume函数建立mapped device和映射表之间的绑定关系，事实上该过程就是通过dm\_swap\_table函数将当前dm\_table结构指针值赋予mapped\_device相应的map域中，然后再修改mapped\_device表示当前状态的域。

通过上述的4个主要步骤，device mapper在内核中就建立一个可以提供给用户使用的mapped device逻辑块设备。

**IO流**

Device mapper本质功能就是根据映射关系和target driver描述的IO处理规则，将IO请求从逻辑设备mapped device转发相应的target device上。Device mapper处理所有从内核中块一级IO子系统的generic\_make\_request和submit\_bio接口[两个接口具体的描述可以查看参考文献[1]和[2]，这两本书对内核中的块IO层有比较详尽的讲解。] 中定向到mapped device的所有块读写IO请求。IO请求在device mapper的设备树中通过请求转发从上到下地进行处理。当一个bio请求在设备树中的mapped deivce向下层转发时，一个或者多个bio的克隆被创建并发送给下层target device。然后相同的过程在设备树的每一个层次上重复，只要设备树足够大理论上这种转发过程可以无限进行下去。在设备树上某个层次中，target driver结束某个bio请求后，将表示结束该bio请求的事件上报给它上层的mapped device，该过程在各个层次上进行直到该事件最终上传到根mapped device的为止，然后device mapper结束根mapped device上原始bio请求，结束整个IO请求过程。

Bio在device mapper的设备树进行逐层的转发时，最终转发到一个或多个叶子target节点终止。因为一个bio请求不可以跨多个target device(亦即物理空间段)， 因此在每一个层次上，device mapper根据用户预先告知的mapped device 的target映射信息克隆一个或者多个bio，将bio进行拆分后转发到对应的target device上。这些克隆的bio先交给mapped device上对应的target driver上进行处理，根据target driver中定义的IO处理规则进行IO请求的过滤等处理，然后再提交给target device完成。上述过程在dm.c文件中的dm\_request函数中完成。Target driver可以对这些bio做如下处理：

1、将这些bio在本驱动内部排队等待以后进行处理；

2、将bio重新定向到一个或多个target device上或者每个target device上的不同扇区；

3、向device mapper返回error 状态。

IO请求就按照上文中描述的过程在图2中所示的设备树中逐层进行处理，直到IO请求结束。

**小结**

Device mapper在内核中向外提供了一个从逻辑设备到物理设备的映射架构，只要用户在用户空间制定好映射策略，按照自己的需要编写处理具体IO请求的target driver插件，就可以很方便的实现一个类似LVM的逻辑卷管理器。Device mapper以ioctl的方式向外提供接口，用户通过用户空间的device mapper库，向device mapper的字符设备发送ioctl命令，完成向内的通信。它还通过ioctl提供向往的事件通知机制，允许target driver将IO相关的某些事件传送到用户空间。

## 用户部分

Device mapper在用户空间相对简单，主要包括device mapper库和dmsetup工具。Device mapper库就是对ioctl、用户空间创建删除device mapper逻辑设备所需必要操作的封装，dmsetup是一个提供给用户直接可用的创建删除device mapper设备的命令行工具。因为它们的功能和流程相对简单，在本文中对它们的细节就不介绍了，用户空间主要负责如下工作：

1、发现每个mapped device相关的target device；

2、根据配置信息创建映射表；

3、将用户空间构建好的映射表传入内核，让内核构建该mapped device对应的dm\_table结构；

4、保存当前的映射信息，以便未来重新构建。

以下我们主要通过实例来说明dmsetup的使用，同时进一步说明device mapper这种映射机制。用户空间中最主要的工作就是构建并保存映射表，下面给出一些映射表的例子：

1)0 1024 linear /dev/sda 204

1024 512 linear /dev/sdb 766

1536 128 linear /dev/sdc 0

2) 0 2048 striped 2 64 /dev/sda 1024 /dev/sdb 0

3) 0 4711 mirror core 2 64 nosync 2 /dev/sda 2048 /dev/sdb 1024

例子1中将逻辑设备0~1023扇区、1024~1535扇区以及1536~1663三个地址范围分别以线形映射的方式映射到/dev/sda设备第204号扇区、/dev/sdb设备第766号扇区和/dev/sdc设备的第0号扇区开始的区域。

例子2中将逻辑设备从0号扇区开始的，长度为2048个扇区的段以条带的方式映射的到/dev/sda设备的第1024号扇区以及/dev/sdb设备的第0号扇区开始的区域。同时告诉内核这个条带类型的target driver存在2个条带设备与逻辑设备做映射，并且条带的大小是64个扇区，使得驱动可以该值来拆分跨设备的IO请求。

例子3中将逻辑设备从0号扇区开始的，长度为4711个扇区的段以镜像的方式映射到/dev/sda设备的第2048个扇区以及/dev/sdb设备的第1024号扇区开始的区域。

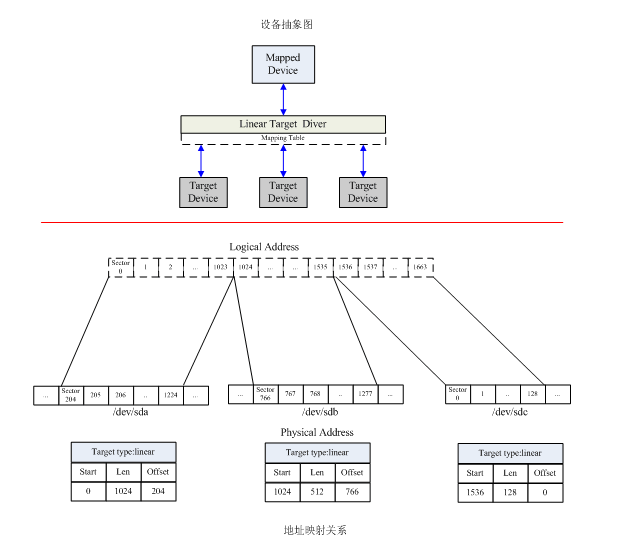
映射表确定后，创建、删除逻辑设备的操作就相对简单，通过dmsetup如下命令就可以完成相应的操作。

dmsetup create 设备名 映射表文件 /\* 根据指定的映射表创建一个逻辑设备 \*/

dmsetup reload 设备名 映射表文件 /\* 为指定设备从磁盘中读取映射文件，重新构建映射关系 \*/

dmsetup remove 设备名 /\* 删除指定的逻辑设备 \*/

图4根据例子1中映射表在内核中建立的逻辑设备



当用户空间根据映射表下达创建逻辑设备命令后，device mapper在内核中就根据传入的参数和映射关系建立逻辑地址到物理地址的映射关系。根据映射表例子1中的映射关系建立的设备如图4所示，图中的下半部分就抽象地描绘出了按照该映射表在内核中建立的逻辑地址到物理地址的映射关系。

Device mapper的用户空间部分对开发者要实现自己的存储管理工具来说是可选的，事实上，很多我们常见的逻辑卷管理器，比如LVM2、dmraid等工具都利用device mapper的提供的device mapper用户空间库，根据自己的管理需求建立独立的一套管理工具，而并没有使用它提供的dmsetup工具，甚至IBM的开源项目企业级的逻辑卷管理系统-EVMS，在实现中都没有采用device mapper的用户空间库，完全根据内核中的ioctl定义实现了一套自己的函数库。

## Target Driver

Device mapper提供了一个统一的架构，通过target driver插件的方式允许用户根据实际的需要指定自己的IO处理规则，因此target driver充分体现了device mapper的灵活性。在上文中我们已经不止一次的提到过target driver，也描述过target driver的功能，在这里我们结合最简单的linear target driver具体介绍target driver的实现。

Target driver主要定义对IO请求的处理规则，在device mapper中对target driver的操作已定义好了统一的接口，在实现中该接口由我们上文提到的target\_type结构中定义，它定义了以下target driver的方法：

１、构建target device 的方法；

２、删除target device 的方法；

３、Target的映射IO请求的方法；

４、Target结束IO请求的方法；

５、暂停target device读写的方法；

６、恢复target device读写的访问；

７、获取当前target device状态的访问；

８、 Target 处理用户消息的方法；

用户可以根据具体需求选择性地实现上述方法，但一般最少要实现前3种方法，否则在device mapper下不能够正常的工作。linear target driver就只实现了前3种方法和方法7，它完成逻辑地址空间到物理地址空间的线性映射，可以将多个物理设备以线性连接的方式组成一个逻辑设备，就如图4中描述的那样，通过linear target driver将/dev/sda、/dev/sdb、/dev/sdc的三段连续空间组成了一个大的逻辑块设备。Linear target的实现很简单，它的创建和删除方法主要完成申请和释放描述linear target device所用结构的内存资源；IO映射处理方法的实现更是简单，如下代码所示：

static int linear\_map(struct dm\_target \*ti, struct bio \*bio,

union map\_info \*map\_context)

{

struct linear\_c \*lc = (struct linear\_c \*) ti->private;

bio->bi\_bdev = lc->dev->bdev;

bio->bi\_sector = lc->start + (bio->bi\_sector - ti->begin);

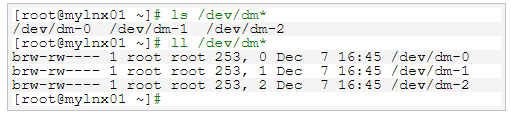
return 1;

}

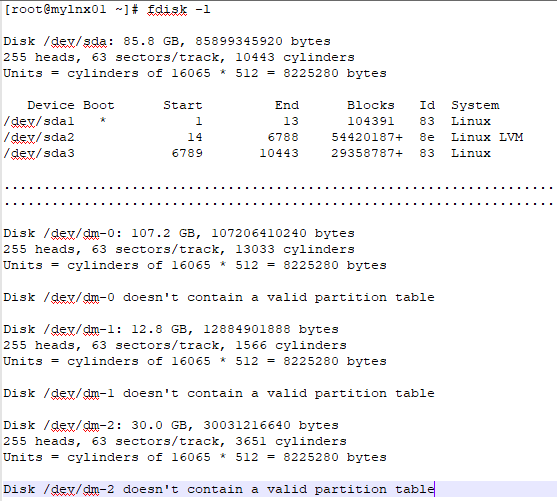
该映射方法就是将发送给逻辑设备mapped device的bio请求，根据映射关系以线性的方式重新定向到linear target device所表示物理设备的相应位置，如代码所示具体实现方法就是修改bio的bi\_bdev设备指针为target device对应的设备指针，并根据target device的起始地址和该bio请求在mapped device设备上的偏移值改变IO请求开始的扇区号bi\_sector，从而完成IO请求的重定向。其他target driver的实现也都大同小异，按照device mapper所定义的接口规范，结合自己需要的功能进行实现即可。

# 实例

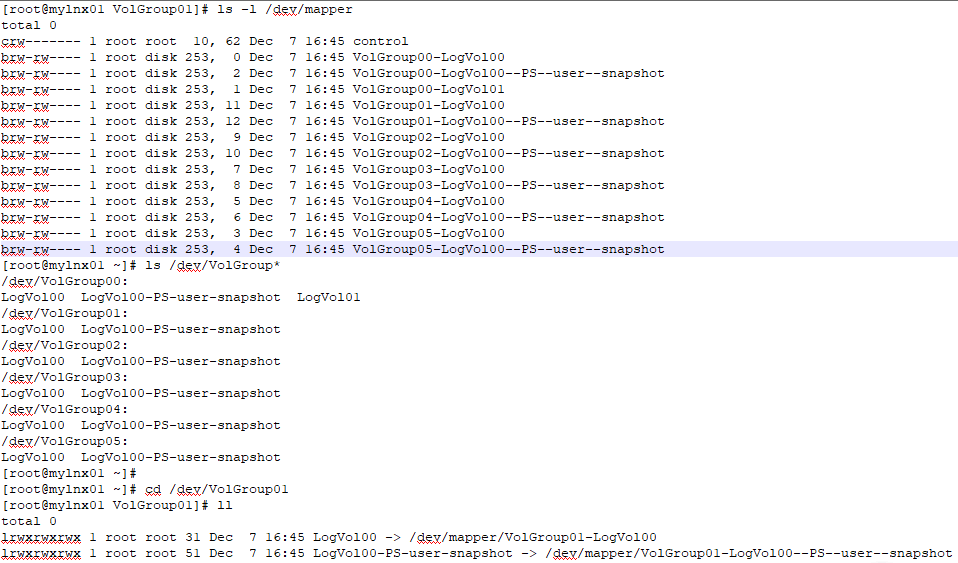
例如上图所示，dm-0、dm-1、dm-xx各自与那些磁盘映射呢？各自又代表的啥意义呢？首先我们来看看dm-0，dm-1，dm-2这个三个文件设备：



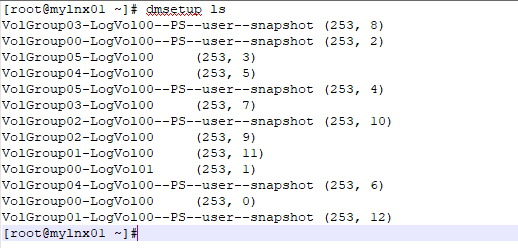
或者你使用fdisk -l命令也能看到：

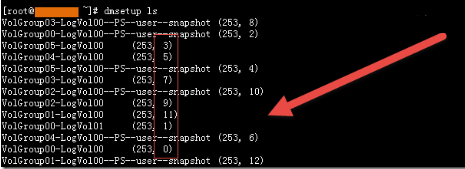


虽然/dev/下面没有所谓的dm-3、dm-4....，一般你用nmon、iostat之类的命令就能看到，其实他也可以查看这些dm对应的那些设备，一般都位于/dev/mapper下面。

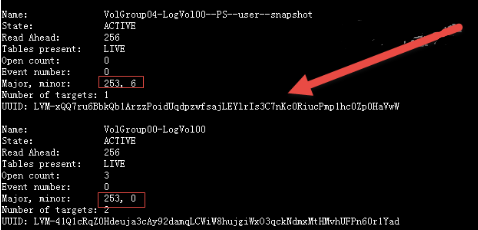


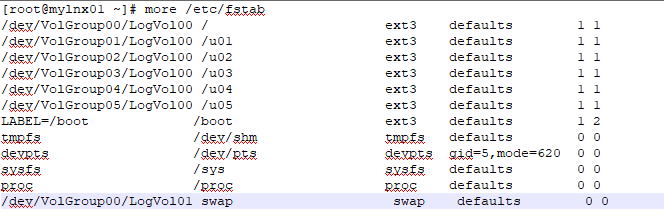
其实我们可以使用命令dmsetup ls查看dm-0、dm-1、dm-x对应的设备，其中dm后面的数字对应（253，xx）后面的数字。如下所示：





253后面的数字就对应dm后面的数字，如果你要查看具体的信息就使用命令dmsetup info就能看到下面详细信息。





有了上面信息我们就能知道dm-0、dm-1、dm-2分别对应下面的一些设备

dm-0对应LVM的VolGroup00-LogVol00对应根目录/

dm-1对应LVM的VolGroup00-LogVol01对应swap

dm-2对应LMV的VolGroup00-LogVol00--PS--user--snapshot

# 总结

Device Mapper是Linux操作系统中块设备一级提供的一种主要映射机制，现在已被多数Linux下的逻辑卷管理器所采用。在该机制下，实现用户自定义的存储资源管理策略变得极其方便。理解device mapper所提供的映射机制，也是进一步理解Linux下一些常见逻辑卷管理器实现的基础。