## 文件系统特性

我们都知道磁盘分区完毕后还需要进行格式化(format)，之后操作系统才能够使用这个分割槽。 为什么需要进行『格式化』呢？这是因为每种操作系统所配置的文件属性/权限并不相同， 为了存放这些文件所需的数据，因此就需要将分割槽进行格式化，以成为操作系统能够利用的『文件系统格式(filesystem)』。

由此我们也能够知道，每种操作系统能够使用的文件系统并不相同。 举例来说，windows 98 以前的微软操作系统主要利用的文件系统是 FAT (或 FAT16)，windows 2000 以后的版本有所谓的 NTFS 文件系统，至于 Linux 的正统文件系统则为 Ext2 (Linux second extended file system, ext2fs)这一个。此外，在默认的情况下，windows 操作系统是不会认识 Linux 的 Ext2 的。

传统的磁盘与文件系统之应用中，一个分割槽就是只能够被格式化成为一个文件系统，所以我们可以说一个 filesystem 就是一个 partition。但是由于新技术的利用，例如我们常听到的LVM与软件磁盘阵列(software raid)， 这些技术可以将一个分割槽格式化为多个文件系统(例如LVM)，也能够将多个分割槽合成一个文件系统(LVM, RAID)！ 所以说，目前我们在格式化时已经不再说成针对 partition 来格式化了， 通常我们可以称呼一个可被挂载的数据为一个文件系统而不是一个分割槽喔！

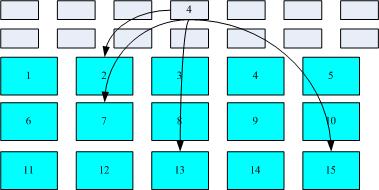
那么文件系统是如何运行的呢？这与操作系统的文件数据有关。较新的操作系统的文件数据除了文件实际内容外， 通常含有非常多的属性，例如 Linux 操作系统的文件权限(rwx)与文件属性(拥有者、群组、时间参数等)。 文件系统通常会将这两部份的数据分别存放在不同的区块，权限与属性放置到 inode 中，至于实际数据则放置到 data block 区块中。 另外，还有一个超级区块 (superblock) 会记录整个文件系统的整体信息，包括 inode 与 block 的总量、使用量、剩余量等。

每个 inode 与 block 都有编号，至于这三个数据的意义可以简略说明如下：

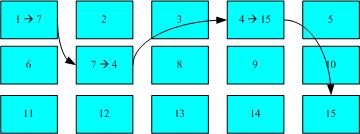
* superblock：记录此 filesystem 的整体信息，包括inode/block的总量、使用量、剩余量， 以及文件系统的格式与相关信息等；
* inode：记录文件的属性，一个文件占用一个inode，同时记录此文件的数据所在的 block 号码；
* block：实际记录文件的内容，若文件太大时，会占用多个 block 。

由于每个 inode 与 block 都有编号，而每个文件都会占用一个 inode ，inode 内则有文件数据放置的 block 号码。 因此，我们可以知道的是，如果能够找到文件的 inode 的话，那么自然就会知道这个文件所放置数据的 block 号码， 当然也就能够读出该文件的实际数据了。这是个比较有效率的作法，因为如此一来我们的磁盘就能够在短时间内读取出全部的数据， 读写的效能比较好啰。

我们将 inode 与 block 区块用图解来说明一下，如下图所示，文件系统先格式化出 inode 与 block 的区块，假设某一个文件的属性与权限数据是放置到 inode 4 号(下图较小方格内)，而这个 inode 记录了文件数据的实际放置点为 2, 7, 13, 15 这四个 block 号码，此时我们的操作系统就能够据此来排列磁盘的阅读顺序，可以一口气将四个 block 内容读出来！ 那么数据的读取就如同下图中的箭头所指定的模样了。

  
图1.2.1、inode/block 数据存取示意图

这种数据存取的方法我们称为索引式文件系统(indexed allocation)。那有没有其他的惯用文件系统可以比较一下啊？ 有的，那就是我们惯用的闪盘(闪存)，闪盘使用的文件系统一般为 FAT 格式。FAT 这种格式的文件系统并没有 inode 存在，所以 FAT 没有办法将这个文件的所有 block 在一开始就读取出来。每个 block 号码都记录在前一个 block 当中， 他的读取方式有点像底下这样：

  
图1.2.2、FAT文件系统数据存取示意图

上图中我们假设文件的数据依序写入1->7->4->15号这四个 block 号码中， 但这个文件系统没有办法一口气就知道四个 block 的号码，他得要一个一个的将 block 读出后，才会知道下一个 block 在何处。 如果同一个文件数据写入的 block 分散的太过厉害时，则我们的磁盘读取头将无法在磁盘转一圈就读到所有的数据， 因此磁盘就会多转好几圈才能完整的读取到这个文件的内容！

常常会听到所谓的『碎片整理』吧？ 需要碎片整理的原因就是文件写入的 block 太过于离散了，此时文件读取的效能将会变的很差所致。 这个时候可以透过碎片整理将同一个文件所属的 blocks 汇整在一起，这样数据的读取会比较容易啊！ 想当然尔，FAT 的文件系统需要经常的碎片整理一下，那么 Ext2 是否需要磁盘重整呢？

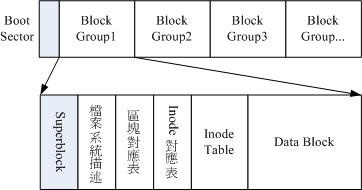
由于 Ext2 是索引式文件系统，基本上不太需要常常进行碎片整理的。但是如果文件系统使用太久， 常常删除/编辑/新增文件时，那么还是可能会造成文件数据太过于离散的问题，此时或许会需要进行重整一下的。 不过，老实说，鸟哥倒是没有在 Linux 操作系统上面进行过 Ext2/Ext3 文件系统的碎片整理说！似乎不太需要啦！^\_^

## Linux 的 EXT2 文件系统(inode)：

Linux 的文件除了原有的数据内容外，还含有非常多的权限与属性，这些权限与属性是为了保护每个用户所拥有数据的隐密性。 而前一小节我们知道 filesystem 里面可能含有的 inode/block/superblock 等。为什么要谈这个呢？因为标准的 Linux 文件系统 Ext2 就是使用这种 inode 为基础的文件系统啦！

而如同前一小节所说的，inode 的内容在记录文件的权限与相关属性，至于 block 区块则是在记录文件的实际内容。 而且文件系统一开始就将 inode 与 block 规划好了，除非重新格式化(或者利用 resize2fs 等命令变更文件系统大小)，否则 inode 与 block 固定后就不再变动。但是如果仔细考虑一下，如果我的文件系统高达数百GB时， 那么将所有的 inode 与 block 通通放置在一起将是很不智的决定，因为 inode 与 block 的数量太庞大，不容易管理。

为此之故，因此 Ext2 文件系统在格式化的时候基本上是区分为多个区块群组 (block group) 的，每个区块群组都有独立的 inode/block/superblock 系统。感觉上就好像我们在当兵时，一个营里面有分成数个连，每个连有自己的联络系统， 但最终都向营部回报连上最正确的信息一般！这样分成一群群的比较好管理啦！整个来说，Ext2 格式化后有点像底下这样：



在整体的规划当中，文件系统最前面有一个启动扇区(boot sector)，这个启动扇区可以安装启动管理程序， 这是个非常重要的设计，因为如此一来我们就能够将不同的启动管理程序安装到个别的文件系统最前端，而不用覆盖整颗硬盘唯一的 MBR， 这样也才能够制作出多重引导的环境啊！至于每一个区块群组(block group)的六个主要内容说明如后：

* **data block (数据区块)**

data block 是用来放置文件内容数据地方，在 Ext2 文件系统中所支持的 block 大小有 1K, 2K 及 4K 三种而已。在格式化时 block 的大小就固定了，且每个 block 都有编号，以方便 inode 的记录啦。 不过要注意的是，由于 block 大小的差异，会导致该文件系统能够支持的最大磁盘容量与最大单一文件容量并不相同。 因为 block 大小而产生的 Ext2 文件系统限制如下：

|  |  |  |  |
| --- | --- | --- | --- |
| Block 大小 | 1KB | 2KB | 4KB |
| 最大单一文件限制 | 16GB | 256GB | 2TB |
| 最大文件系统总容量 | 2TB | 8TB | 16TB |

你需要注意的是，虽然 Ext2 已经能够支持大于 2GB 以上的单一文件容量，不过某些应用程序依然使用旧的限制， 也就是说，某些程序只能够捉到小于 2GB 以下的文件而已，这就跟文件系统无关了！ 举例来说，鸟哥在环工方面的应用中有一套秀图软件称为PAVE， 这套软件就无法捉到鸟哥在数值模式仿真后产生的大于 2GB 以上的文件！害的鸟哥常常还要重跑数值模式...

除此之外 Ext2 文件系统的 block 还有什么限制呢？有的！基本限制如下：

* 原则上，block 的大小与数量在格式化完就不能够再改变了(除非重新格式化)；
* 每个 block 内最多只能够放置一个文件的数据；
* 承上，如果文件大于 block 的大小，则一个文件会占用多个 block 数量；
* 承上，若文件小于 block ，则该 block 的剩余容量就不能够再被使用了(磁盘空间会浪费)。

如上第四点所说，由于每个 block 仅能容纳一个文件的数据而已，因此如果你的文件都非常小，但是你的 block 在格式化时却选用最大的 4K 时，可能会产生一些容量的浪费喔！我们以底下的一个简单例题来算一下空间的浪费吧！

|  |
| --- |
| 例题：  假设你的Ext2文件系统使用 4K block ，而该文件系统中有 10000 个小文件，每个文件大小均为 50bytes， 请问此时你的磁盘浪费多少容量？  答：  由于 Ext2 文件系统中一个 block 仅能容纳一个文件，因此每个 block 会浪费『 4096 - 50 = 4046 (byte)』， 系统中总共有一万个小文件，所有文件容量为：50 (bytes) x 10000 = 488.3Kbytes，但此时浪费的容量为：『 4046 (bytes) x 10000 = 38.6MBytes 』。想一想，不到 1MB 的总文件容量却浪费将近 40MB 的容量，且文件越多将造成越多的磁盘容量浪费。 |

什么情况会产生上述的状况呢？例如 BBS 网站的数据啦！如果 BBS 上面的数据使用的是纯文本文件来记载每篇留言， 而留言内容如果都写上『如题』时，想一想，是否就会产生很多小文件了呢？

好，既然大的 block 可能会产生较严重的磁盘容量浪费，那么我们是否就将 block 大小订为 1K 即可？ 这也不妥，因为如果 block 较小的话，那么大型文件将会占用数量更多的 block ，而 inode 也要记录更多的 block 号码，此时将可能导致文件系统不良的读写效能。

所以我们可以说，在您进行文件系统的格式化之前，请先想好该文件系统预计使用的情况。 以鸟哥来说，我的数值模式仿真平台随便一个文件都好几百 MB，那么 block 容量当然选择较大的！至少文件系统就不必记录太多的 block 号码，读写起来也比较方便啊！

* **inode table (inode 表格)**

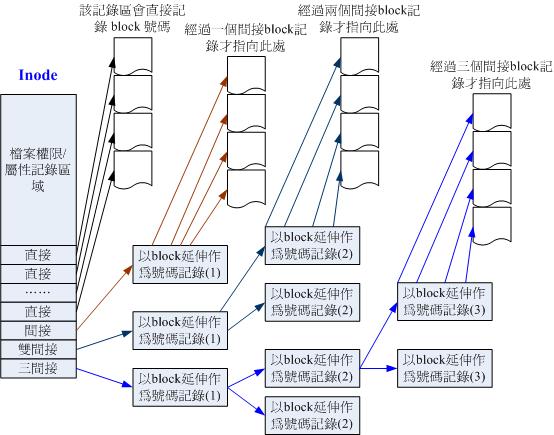
再来讨论一下 inode 这个玩意儿吧！如前所述 inode 的内容在记录文件的属性以及该文件实际数据是放置在哪几号 block 内！ 基本上，inode 记录的文件数据至少有底下这些：

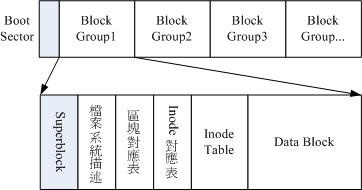
* 该文件的存取模式(read/write/excute)；
* 该文件的拥有者与群组(owner/group)；
* 该文件的容量；
* 该文件创建或状态改变的时间(ctime)；
* 最近一次的读取时间(atime)；
* 最近修改的时间(mtime)；
* 定义文件特性的旗标(flag)，如 SetUID...；
* 该文件真正内容的指向 (pointer)；

inode 的数量与大小也是在格式化时就已经固定了，除此之外 inode 还有些什么特色呢？

* 每个 inode 大小均固定为 128 bytes；
* 每个文件都仅会占用一个 inode 而已；
* 承上，因此文件系统能够创建的文件数量与 inode 的数量有关；
* 系统读取文件时需要先找到 inode，并分析 inode 所记录的权限与用户是否符合，若符合才能够开始实际读取 block 的内容。

我们约略来分析一下 inode / block 与文件大小的关系好了。inode 要记录的数据非常多，但偏偏又只有 128bytes 而已， 而 inode 记录一个 block 号码要花掉 4byte ，假设我一个文件有 400MB 且每个 block 为 4K 时， 那么至少也要十万笔 block 号码的记录呢！inode 哪有这么多可记录的信息？为此我们的系统很聪明的将 inode 记录 block 号码的区域定义为12个直接，一个间接, 一个双间接与一个三间接记录区。这是啥？我们将 inode 的结构画一下好了。

  
图1.3.2、inode 结构示意图



上图最左边为 inode 本身 (128 bytes)，里面有 12 个直接指向 block 号码的对照，这 12 笔记录就能够直接取得 block 号码啦！ 至于所谓的间接就是再拿一个 block 来当作记录 block 号码的记录区，如果文件太大时， 就会使用间接的 block 来记录编号。如上图 1.3.2 当中间接只是拿一个 block 来记录额外的号码而已。 同理，如果文件持续长大，那么就会利用所谓的双间接，第一个 block 仅再指出下一个记录编号的 block 在哪里， 实际记录的在第二个 block 当中。依此类推，三间接就是利用第三层 block 来记录编号啦！

这样子 inode 能够指定多少个 block 呢？我们以较小的 1K block 来说明好了，可以指定的情况如下：

* 12 个直接指向： 12\*1K=12K  
  由于是直接指向，所以总共可记录 12 笔记录，因此总额大小为如上所示；
* 间接： 256\*1K=256K  
  每笔 block 号码的记录会花去 4bytes，因此 1K 的大小能够记录 256 笔记录，因此一个间接可以记录的文件大小如上；
* 双间接： 256\*256\*1K=2562K  
  第一层 block 会指定 256 个第二层，每个第二层可以指定 256 个号码，因此总额大小如上；
* 三间接： 256\*256\*256\*1K=2563K  
  第一层 block 会指定 256 个第二层，每个第二层可以指定 256 个第三层，每个第三层可以指定 256 个号码，因此总额大小如上；

总额：将直接、间接、双间接、三间接加总，得到 12 + 256 + 256\*256 + 256\*256\*256 (K) = 16GB

此时我们知道当文件系统将 block 格式化为 1K 大小时，能够容纳的最大文件为 16GB，比较一下[文件系统限制表](file:///C:\My%20Websites\%E9%B8%9F%E5%93%A5%E7%9A%84linux\cn.linux.vbird.org\linux_basic\0230filesystem_1.html#filesystem_limit)的结果可发现是一致的！但这个方法不能用在 2K 及 4K block 大小的计算中， 因为大于 2K 的 block 将会受到 Ext2 文件系统本身的限制，所以计算的结果会不太符合之故。

* **Superblock (超级区块)**

Superblock 是记录整个 filesystem 相关信息的地方， 没有 Superblock ，就没有这个 filesystem 了。他记录的信息主要有：

* block 与 inode 的总量；
* 未使用与已使用的 inode / block 数量；
* block 与 inode 的大小 (block 为 1, 2, 4K，inode 为 128 bytes)；
* filesystem 的挂载时间、最近一次写入数据的时间、最近一次检验磁盘 (fsck) 的时间等文件系统的相关信息；
* 一个 valid bit 数值，若此文件系统已被挂载，则 valid bit 为 0 ，若未被挂载，则 valid bit 为 1 。

Superblock 是非常重要的，因为我们这个文件系统的基本信息都写在这里，因此，如果 superblock 死掉了， 你的文件系统可能就需要花费很多时间去挽救啦！一般来说， superblock 的大小为 1024bytes。相关的 superblock 信息我们等一下会以 [dumpe2fs](file:///C:\My%20Websites\%E9%B8%9F%E5%93%A5%E7%9A%84linux\cn.linux.vbird.org\linux_basic\0230filesystem_1.html#dumpe2fs) 命令来呼叫出来观察喔！

此外，每个 block group 都可能含有 superblock 喔！但是我们也说一个文件系统应该仅有一个 superblock 而已，那是怎么回事啊？ 事实上除了第一个 block group 内会含有 superblock 之外，后续的 block group 不一定含有 superblock ， 而若含有 superblock 则该 superblock 主要是做为第一个 block group 内 superblock 的备份咯，这样可以进行 superblock 的救援呢！

* **Filesystem Description (文件系统描述说明)**

这个区段可以描述每个 block group 的开始与结束的 block 号码，以及说明每个区段 (superblock, bitmap, inodemap, data block) 分别介于哪一个 block 号码之间。这部份也能够用 [dumpe2fs](file:///C:\My%20Websites\%E9%B8%9F%E5%93%A5%E7%9A%84linux\cn.linux.vbird.org\linux_basic\0230filesystem_1.html#dumpe2fs) 来观察的。

* **block bitmap (区块对照表)**

如果你想要新增文件时总会用到 block 吧！那你要使用哪个 block 来记录呢？当然是选择『空的 block 』来记录新文件的数据啰。 那你怎么知道哪个 block 是空的？这就得要透过 block bitmap 的辅助了。从 block bitmap 当中可以知道哪些 block 是空的，因此我们的系统就能够很快速的找到可使用的空间来处置文件啰。

同样的，如果你删除某些文件时，那么那些文件原本占用的 block 号码就得要释放出来， 此时在 block bitmap 当中相对应到该 block 号码的标志就得要修改成为『未使用中』啰！这就是 bitmap 的功能。

* **inode bitmap (inode 对照表)**

这个其实与 block bitmap 是类似的功能，只是 block bitmap 记录的是使用与未使用的 block 号码， 至于 inode bitmap 则是记录使用与未使用的 inode 号码啰！

了解了文件系统的概念之后，再来当然是观察这个文件系统啰！刚刚谈到的各部分数据都与 block 号码有关！ 每个区段与 superblock 的信息都可以使用 dumpe2fs 这个命令来查询的！查询的方法与实际的观察如下：

|  |
| --- |
| [root@www ~]# **dumpe2fs [-bh] 装置文件名**  选项与参数：  -b ：列出保留为坏轨的部分(一般用不到吧！？)  -h ：仅列出 superblock 的数据，不会列出其他的区段内容！  范例：找出我的根目录磁盘文件名，并观察文件系统的相关信息  [root@www ~]# **df** <==这个命令可以叫出目前挂载的装置  Filesystem 1K-blocks Used Available Use% Mounted on  /dev/hdc2 9920624 3822848 5585708 41% / <==就是这个光！  /dev/hdc3 4956316 141376 4559108 4% /home  /dev/hdc1 101086 11126 84741 12% /boot  tmpfs 371332 0 371332 0% /dev/shm  [root@www ~]# **dumpe2fs /dev/hdc2**  dumpe2fs 1.39 (29-May-2006)  Filesystem volume name: /1 <==这个是文件系统的名称(Label)  Filesystem features: has\_journal ext\_attr resize\_inode dir\_index  filetype needs\_recovery sparse\_super large\_file  Default mount options: user\_xattr acl <==默认挂载的参数  Filesystem state: clean <==这个文件系统是没问题的(clean)  Errors behavior: Continue  Filesystem OS type: Linux  Inode count: 2560864 <==inode的总数  Block count: 2560359 <==block的总数  Free blocks: 1524760 <==还有多少个 block 可用  Free inodes: 2411225 <==还有多少个 inode 可用  First block: 0  Block size: 4096 <==每个 block 的大小啦！  Filesystem created: Fri Sep 5 01:49:20 2008  Last mount time: Mon Sep 22 12:09:30 2008  Last write time: Mon Sep 22 12:09:30 2008  Last checked: Fri Sep 5 01:49:20 2008  First inode: 11  Inode size: 128 <==每个 inode 的大小  Journal inode: 8 <==底下这三个与下一小节有关  Journal backup: inode blocks  Journal size: 128M  Group 0: (Blocks 0-32767) <==第一个 data group 内容, 包含 block 的启始/结束号码  Primary superblock at 0, Group descriptors at 1-1 <==超级区块在 0 号 block  Reserved GDT blocks at 2-626  Block bitmap at 627 (+627), Inode bitmap at 628 (+628)  Inode table at 629-1641 (+629) <==inode table 所在的 block  0 free blocks, 32405 free inodes, 2 directories <==所有 block 都用完了！  Free blocks:  Free inodes: 12-32416 <==剩余未使用的 inode 号码  Group 1: (Blocks 32768-65535)  ....(底下省略)....  # 由于数据量非常的庞大，因此鸟哥将一些信息省略输出了！上表与你的屏幕会有点差异。  # 前半部在秀出 supberblock 的内容，包括标头名称(Label)以及inode/block的相关信息  # 后面则是每个 block group 的个别信息了！您可以看到各区段数据所在的号码！  # 也就是说，基本上所有的数据还是与 block 的号码有关就是了！很重要！ |

如上所示，利用 dumpe2fs 可以查询到非常多的信息，不过依内容主要可以区分为上半部是 superblock 内容， 下半部则是每个 block group 的信息了。从上面的表格中我们可以观察到这个 /dev/hdc2 规划的 block 为 4K， 第一个 block 号码为 0 号，且 block group 内的所有信息都以 block 的号码来表示的。 然后在 superblock 中还有谈到目前这个文件系统的可用 block 与 inode 数量喔！

至于 block group 的内容我们单纯看 Group0 信息好了。从上表中我们可以发现：

* Group0 所占用的 block 号码由 0 到 32767 号，superblock 则在第 0 号的 block 区块内！
* 文件系统描述说明在第 1 号 block 中；
* block bitmap 与 inode bitmap 则在 627 及 628 的 block 号码上。
* 至于 inode table 分布于 629-1641 的 block 号码中！
* 由于 (1)一个 inode 占用 128 bytes ，(2)总共有 1641 - 629 + 1(629本身) = 1013 个 block 花在 inode table 上， (3)每个 block 的大小为 4096 bytes(4K)。由这些数据可以算出 inode 的数量共有 1013 \* 4096 / 128 = 32416 个 inode 啦！
* 这个 Group0 目前没有可用的 block 了，但是有剩余 32405 个 inode 未被使用；
* 剩余的 inode 号码为 12 号到 32416 号。

如果你对文件系统的详细信息还有更多想要了解的话，那么请参考本章最后一小节的介绍喔！ 否则文件系统看到这里对于基础认知您应该是已经相当足够啦！底下则是要探讨一下， 那么这个文件系统概念与实际的目录树应用有啥关连啊？

## 与目录树的关系

由前一小节的介绍我们知道在 Linux 系统下，每个文件(不管是一般文件还是目录文件)都会占用一个inode，且可依据文件内容的大小来分配多个 block 给该文件使用。而且我们知道目录的内容在记录文件名，一般文件才是实际记录数据内容的地方。那么目录与文件在 Ext2 文件系统当中是如何记录数据的呢？

### 目录

当我们在 Linux 下的 ext2 文件系统创建一个目录时， ext2 会分配一个 inode 与至少一块 block 给该目录。其中，inode 记录该目录的相关权限与属性，并可记录分配到的那块 block 号码； 而 block 则是记录在这个目录下的文件名与该文件名占用的 inode 号码数据。也就是说目录所占用的 block 内容在记录如下的信息：



如果想要实际观察 root 家目录内的文件所占用的 inode 号码时，可以使用 ls -i 这个选项来处理：

[root@www ~]# **ls -li**

total 92

654683 -rw------- 1 root root 1474 Sep 4 18:27 anaconda-ks.cfg

648322 -rw-r--r-- 1 root root 42304 Sep 4 18:26 install.log

648323 -rw-r--r-- 1 root root 5661 Sep 4 18:25 install.log.syslog

由于每个人所使用的计算机并不相同，系统安装时选择的项目与 partition 都不一样，因此你的环境不可能与我的 inode 号码一模一样！上表的左边所列出的 inode 仅是鸟哥的系统所显示的结果而已！而由这个目录的 block 结果我们现在就能够知道， 当你使用『 ll / 』时，出现的目录几乎都是 1024 的倍数，为什么呢？因为每个 block 的数量都是 1K, 2K, 4K 嘛！ 看一下鸟哥的环境：

|  |
| --- |
| [root@www ~]# **ll -d / /bin /boot /proc /lost+found /sbin**  drwxr-xr-x 23 root root 4096 Sep 22 12:09 / <==一个 4K block  drwxr-xr-x 2 root root 4096 Sep 24 00:07 /bin <==一个 4K block  drwxr-xr-x 4 root root 1024 Sep 4 18:06 /boot <==一个 1K block  drwx------ 2 root root 16384 Sep 5 01:49 /lost+found <==四个 4K block  dr-xr-xr-x 96 root root 0 Sep 22 20:07 /proc <==此目录不占硬盘空间  drwxr-xr-x 2 root root 12288 Sep 5 12:33 /sbin <==三个 4K block |

由于鸟哥的根目录 /dev/hdc2 使用的 block 大小为 4K ，因此每个目录几乎都是 4K 的倍数。 其中由于 /sbin 的内容比较复杂因此占用了 3 个 block ，此外，鸟哥的系统中 /boot 为独立的 partition ， 该 partition 的 block 为 1K 而已，因此该目录就仅占用 1024 bytes 的大小啰！

|  |
| --- |
| **Tips:** 由上面的结果我们知道目录并不只会占用一个 block 而已，也就是说： 在目录底下的文件数如果太多而导致一个 block 无法容纳的下所有的档名与 inode 对照表时，Linux 会给予该目录多一个 block 来继续记录相关的数据； |

### 文件：

当我们在 Linux 下的 ext2 创建一个一般文件时， ext2 会分配一个 inode 与相对于该文件大小的 block 数量给该文件。例如：假设我的一个 block 为 4 Kbytes ，而我要创建一个 100 KBytes 的文件，那么 linux 将分配一个 inode 与 25 个 block 来储存该文件！ 但同时请注意，由于 inode 仅有 12 个直接指向，因此还要多一个 block 来作为区块号码的记录喔！

### 目录树读取：

好了，经过上面的说明你也应该要很清楚的知道 inode 本身并不记录文件名，文件名的记录是在目录的 block 当中。 因此在[第六章文件与目录的权限](file:///C:\My%20Websites\%E9%B8%9F%E5%93%A5%E7%9A%84linux\cn.linux.vbird.org\linux_basic\0210filepermission.html#filepermission_dir)说明中， 我们才会提到『新增/删除/更名文件名与目录的 w 权限有关』的特色！那么因为文件名是记录在目录的 block 当中， 因此当我们要读取某个文件时，就务必会经过目录的 inode 与 block ，然后才能够找到那个待读取文件的 inode 号码， 最终才会读到正确的文件的 block 内的数据。

由于目录树是由根目录开始读起，因此系统透过挂载的信息可以找到挂载点的 inode 号码(通常一个 filesystem 的最顶层 inode 号码会由 2 号开始喔！)，此时就能够得到根目录的 inode 内容，并依据该 inode 读取根目录的 block 内的文件名数据，再一层一层的往下读到正确的档名。

举例来说，如果我想要读取 /etc/passwd 这个文件时，系统是如何读取的呢？

[root@www ~]# **ll -di / /etc /etc/passwd**

2 drwxr-xr-x 23 root root 4096 Sep 22 12:09 /

1912545 drwxr-xr-x 105 root root 12288 Oct 14 04:02 /etc

1914888 -rw-r--r-- 1 root root 1945 Sep 29 02:21 /etc/passwd

在鸟哥的系统上面与 /etc/passwd 有关的目录与文件数据如上表所示，该文件的读取流程为(假设读取者身份为 vbird 这个一般身份使用者)：

1. / 的 inode：  
   透过挂载点的信息找到 /dev/hdc2 的 inode 号码为 2 的根目录 inode，且 inode 规范的权限让我们可以读取该 block 的内容(有 r 与 x) ；
2. / 的 block：  
   经过上个步骤取得 block 的号码，并找到该内容有 etc/ 目录的 inode 号码 (1912545)；
3. etc/ 的 inode：  
   读取 1912545 号 inode 得知 vbird 具有 r 与 x 的权限，因此可以读取 etc/ 的 block 内容；
4. etc/ 的 block：  
   经过上个步骤取得 block 号码，并找到该内容有 passwd 文件的 inode 号码 (1914888)；
5. passwd 的 inode：  
   读取 1914888 号 inode 得知 vbird 具有 r 的权限，因此可以读取 passwd 的 block 内容；
6. passwd 的 block：  
   最后将该 block 内容的数据读出来。

### filesystem 大小与磁盘读取效能：

另外，关于文件系统的使用效率上，当你的一个文件系统规划的很大时，例如 100GB 这么大时， 由于硬盘上面的数据总是来来去去的，所以，整个文件系统上面的文件通常无法连续写在一起(block 号码不会连续的意思)， 而是填入式的将数据填入没有被使用的 block 当中。如果文件写入的 block 真的分的很散， 此时就会有所谓的文件数据离散的问题发生了。

如前所述，虽然我们的 ext2 在 inode 处已经将该文件所记录的 block 号码都记上了， 所以数据可以一次性读取，但是如果文件真的太过离散，确实还是会发生读取效率低落的问题。 因为磁盘读取头还是得要在整个文件系统中来来去去的频繁读取！ 果真如此，那么可以将整个 filesystme 内的数据全部复制出来，将该 filesystem 重新格式化， 再将数据给他复制回去即可解决这个问题。

此外，如果 filesystem 真的太大了，那么当一个文件分别记录在这个文件系统的最前面与最后面的 block 号码中， 此时会造成硬盘的机械手臂移动幅度过大，也会造成数据读取效能的低落。而且读取头在搜寻整个 filesystem 时， 也会花费比较多的时间去搜寻！因此， partition 的规划并不是越大越好， 而是真的要针对您的主机用途来进行规划才行！^\_^

## EXT2/EXT3 文件的存取与日志式文件系统的功能

上一小节谈到的仅是读取而已，那么如果是新建一个文件或目录时，我们的 Ext2 是如何处理的呢？ 这个时候就得要 block bitmap 及 inode bitmap 的帮忙了！假设我们想要新增一个文件，此时文件系统的行为是：

1. 先确定用户对于欲新增文件的目录是否具有 w 与 x 的权限，若有的话才能新增；
2. 根据 inode bitmap 找到没有使用的 inode 号码，并将新文件的权限/属性写入；
3. 根据 block bitmap 找到没有使用中的 block 号码，并将实际的数据写入 block 中，且升级 inode 的 block 指向数据；
4. 将刚刚写入的 inode 与 block 数据同步升级 inode bitmap 与 block bitmap，并升级 superblock 的内容。

一般来说，我们将 inode table 与 data block 称为数据存放区域，至于其他例如 superblock、 block bitmap 与 inode bitmap 等区段就被称为 metadata (中介数据) 啰，因为 superblock, inode bitmap 及 block bitmap 的数据是经常变动的，每次新增、移除、编辑时都可能会影响到这三个部分的数据，因此才被称为中介数据的啦。

### 数据的不一致 (Inconsistent) 状态

在一般正常的情况下，上述的新增动作当然可以顺利的完成。但是如果有个万一怎么办？ 例如你的文件在写入文件系统时，因为不知名原因导致系统中断(例如突然的停电啊、 系统核心发生错误啊～等等的怪事发生时)，所以写入的数据仅有 inode table 及 data block 而已， 最后一个同步升级中介数据的步骤并没有做完，此时就会发生 metadata 的内容与实际数据存放区产生不一致 (Inconsistent) 的情况了。

既然有不一致当然就得要克服！在早期的 Ext2 文件系统中，如果发生这个问题， 那么系统在重新启动的时候，就会藉由 Superblock 当中记录的 valid bit (是否有挂载) 与 filesystem state (clean 与否) 等状态来判断是否强制进行数据一致性的检查！若有需要检查时则以 [e2fsck](file:///C:\My%20Websites\%E9%B8%9F%E5%93%A5%E7%9A%84linux\cn.linux.vbird.org\linux_basic\0230filesystem_1.html#fsck) 这支程序来进行的。

不过，这样的检查真的是很费时～因为要针对 metadata 区域与实际数据存放区来进行比对， 呵呵～得要搜寻整个 filesystem 呢～如果你的文件系统有 100GB 以上，而且里面的文件数量又多时， 哇！系统真忙碌～而且在对 Internet 提供服务的服务器主机上面， 这样的检查真的会造成主机复原时间的拉长～真是麻烦～这也就造成后来所谓日志式文件系统的兴起了。

* **日志式文件系统 (Journaling filesystem)**

为了避免上述提到的文件系统不一致的情况发生，因此我们的前辈们想到一个方式， 如果在我们的 filesystem 当中规划出一个区块，该区块专门在记录写入或修订文件时的步骤， 那不就可以简化一致性检查的步骤了？也就是说：

1. 预备：当系统要写入一个文件时，会先在日志记录区块中纪录某个文件准备要写入的信息；
2. 实际写入：开始写入文件的权限与数据；开始升级 metadata 的数据；
3. 结束：完成数据与 metadata 的升级后，在日志记录区块当中完成该文件的纪录。

在这样的程序当中，万一数据的纪录过程当中发生了问题，那么我们的系统只要去检查日志记录区块， 就可以知道哪个文件发生了问题，针对该问题来做一致性的检查即可，而不必针对整块 filesystem 去检查， 这样就可以达到快速修复 filesystem 的能力了！这就是日志式文件最基础的功能啰～

那么我们的 ext2 可达到这样的功能吗？当然可以啊！ 就透过 ext3 即可！ ext3 是 ext2 的升级版本，并且可向下兼容 ext2 版本呢！ 所以啰，目前我们才建议大家，可以直接使用 ext3 这个 filesystem 啊！ 如果你还记得 [dumpe2fs](file:///C:\My%20Websites\%E9%B8%9F%E5%93%A5%E7%9A%84linux\cn.linux.vbird.org\linux_basic\0230filesystem_1.html#dumpe2fs) 输出的信息，可以发现 superblock 里面含有底下这样的信息：



看到了吧！透过 inode 8 号记录 journal 区块的 block 指向，而且具有 128MB 的容量在处理日志呢！ 这样对于所谓的日志式文件系统有没有比较有概念一点呢？^\_^。如果想要知道为什么 Ext3 文件系统会更适用于目前的 Linux 系统， 我们可以参考 Red Hat 公司中，首席核心开发者 Michael K. Johnson 的话：

|  |
| --- |
| 『为什么你想要从ext2转换到ext3呢？有四个主要的理由：可利用性、数据完整性、速度及易于转换』 『可利用性』，他指出，这意味着从系统中止到快速重新复原而不是持续的让e2fsck运行长时间的修复。ext3 的日志式条件可以避免数据毁损的可能。他也指出： 『除了写入若干数据超过一次时，ext3往往会较快于ext2，因为ext3的日志使硬盘读取头的移动能更有效的进行』 然而或许决定的因素还是在Johnson先生的第四个理由中。  『它是可以轻易的从ext2变更到ext3来获得一个强而有力的日志式文件系统而不需要重新做格式化』。『那是正确的，为了体验一下 ext3 的好处是不需要去做一种长时间的，冗长乏味的且易于产生错误的备份工作及重新格式化的动作』。 |

**Linux 文件系统的运行：**

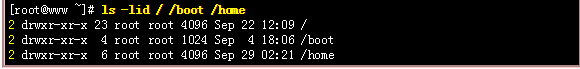
我们现在知道了目录树与文件系统的关系了，但是由[第零章](file:///C:\My%20Websites\%E9%B8%9F%E5%93%A5%E7%9A%84linux\cn.linux.vbird.org\linux_basic\0105computers.html)的内容我们也知道， 所有的数据都得要加载到内存后 CPU 才能够对该数据进行处理。想一想，如果你常常编辑一个好大的文件， 在编辑的过程中又频繁的要系统来写入到磁盘中，由于磁盘写入的速度要比内存慢很多， 因此你会常常耗在等待硬盘的写入/读取上。真没效率！

为了解决这个效率的问题，因此我们的 Linux 使用的方式是透过一个称为异步处理 (asynchronously) 的方式。所谓的异步处理是这样的：

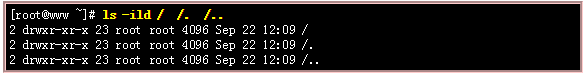
当系统加载一个文件到内存后，如果该文件没有被更动过，则在内存区段的文件数据会被配置为干净(clean)的。 但如果内存中的文件数据被更改过了(例如你用 nano 去编辑过这个文件)，此时该内存中的数据会被配置为脏的 (Dirty)。此时所有的动作都还在内存中运行，并没有写入到磁盘中！ 系统会不定时的将内存中配置为『Dirty』的数据写回磁盘，以保持磁盘与内存数据的一致性。 你也可以利用[第五章谈到的 sync](file:///C:\My%20Websites\%E9%B8%9F%E5%93%A5%E7%9A%84linux\cn.linux.vbird.org\linux_basic\0160startlinux.html#sync)命令来手动强迫写入磁盘。

我们知道内存的速度要比硬盘快的多，因此如果能够将常用的文件放置到内存当中，这不就会添加系统性能吗？ 没错！是有这样的想法！因此我们 Linux 系统上面文件系统与内存有非常大的关系喔：

* 系统会将常用的文件数据放置到主存储器的缓冲区，以加速文件系统的读/写；
* 承上，因此 Linux 的物理内存最后都会被用光！这是正常的情况！可加速系统效能；
* 你可以手动使用 sync 来强迫内存中配置为 Dirty 的文件回写到磁盘中；
* 若正常关机时，关机命令会主动呼叫 sync 来将内存的数据回写入磁盘内；
* 但若不正常关机(如跳电、死机或其他不明原因)，由于数据尚未回写到磁盘内， 因此重新启动后可能会花很多时间在进行磁盘检验，甚至可能导致文件系统的损毁(非磁盘损毁)。
* **挂载点的意义 (mount point)：**
* 每个 filesystem 都有独立的 inode / block / superblock 等信息，这个文件系统要能够链接到目录树才能被我们使用。 将文件系统与目录树结合的动作我们称为『挂载』。 关于挂载的一些特性我们在[第三章](file:///C:\My%20Websites\%E9%B8%9F%E5%93%A5%E7%9A%84linux\cn.linux.vbird.org\linux_basic\0130designlinux.html#partition_install)稍微提过， 重点是：挂载点一定是目录，该目录为进入该文件系统的入口。 因此并不是你有任何文件系统都能使用，必须要『挂载』到目录树的某个目录后，才能够使用该文件系统的。
* 举例来说，如果你是依据鸟哥的方法[安装你的 CentOS 5.x](file:///C:\My%20Websites\%E9%B8%9F%E5%93%A5%E7%9A%84linux\cn.linux.vbird.org\linux_basic\0157installcentos5.html) 的话， 那么应该会有三个挂载点才是，分别是 /, /boot, /home 三个 (鸟哥的系统上对应的装置文件名为 /dev/hdc2, /dev/hdc1, /dev/hdc3)。 那如果观察这三个目录的 inode 号码时，我们可以发现如下的情况：



看到了吧！由于 filesystem 最顶层的目录之 inode 一般为 2 号，因此可以发现 /, /boot, /home 为三个不同的 filesystem 啰！ (因为每一行的文件属性并不相同，且三个目录的挂载点也均不相同之故。) 我们在[第七章一开始的路径](file:///C:\My%20Websites\%E9%B8%9F%E5%93%A5%E7%9A%84linux\cn.linux.vbird.org\linux_basic\0220filemanager.html)中曾经提到根目录下的 . 与 .. 是相同的东西， 因为权限是一模一样嘛！如果使用文件系统的观点来看，同一个 filesystem 的某个 inode 只会对应到一个文件内容而已(因为一个文件占用一个 inode 之故)， 因此我们可以透过判断 inode 号码来确认不同文件名是否为相同的文件喔！所以可以这样看：



上面的信息中由于挂载点均为 / ，因此三个文件 (/, /., /..) 均在同一个 filesystem 内，而这三个文件的 inode 号码均为 2 号，因此这三个档名都指向同一个 inode 号码，当然这三个文件的内容也就完全一模一样了！ 也就是说，根目录的上一级 (/..) 就是他自己！这么说，看的懂了吗？ ^\_^

**其他 Linux 支持的文件系统与 VFS**

虽然 Linux 的标准文件系统是 ext2 ，且还有添加了日志功能的 ext3 ，事实上，Linux 还有支持很多文件系统格式的， 尤其是最近这几年推出了好几种速度很快的日志式文件系统，包括 SGI 的 XFS 文件系统， 可以适用更小型文件的 Reiserfs 文件系统，以及 Windows 的 FAT 文件系统等等， 都能够被 Linux 所支持喔！常见的支持文件系统有：

* 传统文件系统：ext2 / minix / MS-DOS / FAT (用 vfat 模块) / iso9660 (光盘)等等；
* 日志式文件系统： ext3 / ReiserFS / Windows' NTFS / IBM's JFS / SGI's XFS
* 网络文件系统： NFS / SMBFS

想要知道你的 Linux 支持的文件系统有哪些，可以察看底下这个目录：

|  |
| --- |
| [root@www ~]# **ls -l /lib/modules/$(uname -r)/kernel/fs** |

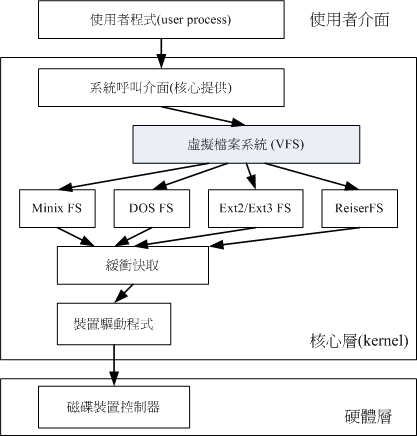
系统目前已加载到内存中支持的文件系统则有：

|  |
| --- |
| [root@www ~]# **cat /proc/filesystems** |

* **Linux VFS (Virtual Filesystem Switch)**

了解了我们使用的文件系统之后，再来则是要提到，那么 Linux 的核心又是如何管理这些认识的文件系统呢？ 其实，整个 Linux 的系统都是透过一个名为 Virtual Filesystem Switch 的核心功能去读取 filesystem 的。 也就是说，整个 Linux 认识的 filesystem 其实都是 VFS 在进行管理，我们使用者并不需要知道每个 partition 上头的 filesystem 是什么～ VFS 会主动的帮我们做好读取的动作呢～

假设你的 / 使用的是 /dev/hda1 ，用 ext3 ，而 /home 使用 /dev/hda2 ，用 reiserfs ， 那么你取用 /home/dmtsai/.bashrc 时，有特别指定要用的什么文件系统的模块来读取吗？ 应该是没有吧！这个就是 VFS 的功能啦！透过这个 VFS 的功能来管理所有的 filesystem， 省去我们需要自行配置读取文件系统的定义啊～方便很多！整个 VFS 可以约略用下图来说明：



**实体链接与符号链接： ln**

在 Linux 底下的连结档有两种，一种是类似 Windows 的快捷方式功能的文件，可以让你快速的链接到目标文件(或目录)； 另一种则是透过文件系统的 inode 连结来产生新档名，而不是产生新文件！这种称为实体链接 (hard link)。 这两种玩意儿是完全不一样的东西呢！现在就分别来谈谈。

* **Hard Link (实体链接, 硬式连结或实际连结)**

在前一小节当中，我们知道几件重要的信息，包括：

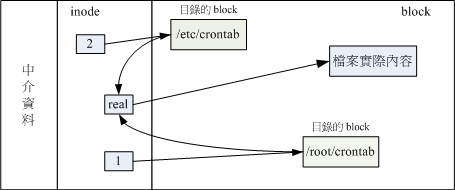
* 每个文件都会占用一个 inode ，文件内容由 inode 的记录来指向；
* 想要读取该文件，必须要经过目录记录的文件名来指向到正确的 inode 号码才能读取。

也就是说，其实文件名只与目录有关，但是文件内容则与 inode 有关。那么想一想， 有没有可能有多个档名对应到同一个 inode 号码呢？有的！那就是 hard link 的由来。 所以简单的说：hard link 只是在某个目录下新增一笔档名链接到某 inode 号码的关连记录而已。

举个例子来说，假设我系统有个 /root/crontab 他是 /etc/crontab 的实体链接，也就是说这两个档名连结到同一个 inode ， 自然这两个文件名的所有相关信息都会一模一样(除了文件名之外)。实际的情况可以如下所示：

|  |
| --- |
| [root@www ~]# **ln /etc/crontab .** <==创建实体链接的命令  [root@www ~]# **ll -i /etc/crontab /root/crontab**  1912701 -rw-r--r-- 2 root root 255 Jan 6 2007 /etc/crontab  1912701 -rw-r--r-- 2 root root 255 Jan 6 2007 /root/crontab |

你可以发现两个档名都连结到 1912701 这个 inode 号码，所以您瞧瞧，是否文件的权限/属性完全一样呢？ 因为这两个『档名』其实是一模一样的『文件』啦！而且你也会发现第二个字段由原本的 1 变成 2 了！ 那个字段称为『连结』，这个字段的意义为：『有多少个档名链接到这个 inode 号码』的意思。 如果将读取到正确数据的方式画成示意图，就类似如下画面：



上图的意思是，你可以透过 1 或 2 的目录之 inode 指定的 block 找到两个不同的档名，而不管使用哪个档名均可以指到 real 那个 inode 去读取到最终数据！那这样有什么好处呢？最大的好处就是『安全』！如同上图中， 如果你将任何一个『档名』删除，其实 inode 与 block 都还是存在的！ 此时你可以透过另一个『档名』来读取到正确的文件数据喔！此外，不论你使用哪个『档名』来编辑， 最终的结果都会写入到相同的 inode 与 block 中，因此均能进行数据的修改哩！

一般来说，使用 hard link 配置链接文件时，磁盘的空间与 inode 的数目都不会改变！ 我们还是由图 2.2.1 来看，由图中可以知道， hard link 只是在某个目录下的 block 多写入一个关连数据而已，既不会添加 inode 也不会耗用 block 数量哩！

|  |  |
| --- | --- |
| **Tips:** hard link 的制作中，其实还是可能会改变系统的 block 的，那就是当你新增这笔数据却刚好将目录的 block 填满时，就可能会新加一个 block 来记录文件名关连性，而导致磁盘空间的变化！不过，一般 hard link 所用掉的关连数据量很小，所以通常不会改变 inode 与磁盘空间的大小喔！ |  |

由图 2.2.1 其实我们也能够知道，事实上 hard link 应该仅能在单一文件系统中进行的，应该是不能够跨文件系统才对！ 因为图 2.2.1 就是在同一个 filesystem 上嘛！所以 hard link 是有限制的：

* 不能跨 Filesystem；
* 不能 link 目录。

不能跨 Filesystem 还好理解，那不能 hard link 到目录又是怎么回事呢？这是因为如果使用 hard link 链接到目录时， 链接的数据需要连同被链接目录底下的所有数据都创建链接，举例来说，如果你要将 /etc 使用实体链接创建一个 /etc\_hd 的目录时，那么在 /etc\_hd 底下的所有档名同时都与 /etc 底下的檔名要创建 hard link 的，而不是仅连结到 /etc\_hd 与 /etc 而已。 并且，未来如果需要在 /etc\_hd 底下创建新文件时，连带的， /etc 底下的数据又得要创建一次 hard link ，因此造成环境相当大的复杂度。 所以啰，目前 hard link 对于目录暂时还是不支持的啊！

* **Symbolic Link (符号链接，亦即是快捷方式)**

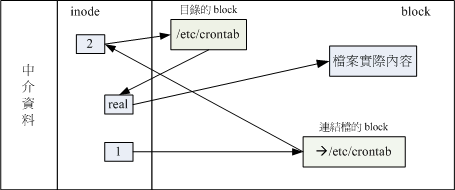
相对于 hard link ， Symbolic link 可就好理解多了，基本上， Symbolic link 就是在创建一个独立的文件，而这个文件会让数据的读取指向他 link 的那个文件的档名！由于只是利用文件来做为指向的动作， 所以，当来源档被删除之后，symbolic link 的文件会『开不了』， 会一直说『无法开启某文件！』。实际上就是找不到原始『档名』而已啦！

举例来说，我们先创建一个符号链接文件链接到 /etc/crontab 去看看：

|  |
| --- |
| [root@www ~]# **ln -s /etc/crontab crontab2**  [root@www ~]# **ll -i /etc/crontab /root/crontab2**  1912701 -rw-r--r-- 2 root root 255 Jan 6 2007 /etc/crontab  654687 lrwxrwxrwx 1 root root 12 Oct 22 13:58 /root/crontab2 -> /etc/crontab |

由上表的结果我们可以知道两个文件指向不同的 inode 号码，当然就是两个独立的文件存在！ 而且连结档的重要内容就是他会写上目标文件的『文件名』， 你可以发现为什么上表中连结档的大小为 12 bytes 呢？ 因为箭头(-->)右边的档名『/etc/crontab』总共有 12 个英文，每个英文占用 1 个 byes ，所以文件大小就是 12bytes了！

关于上述的说明，我们以如下图示来解释：



由 1 号 inode 读取到连结档的内容仅有档名，根据档名链接到正确的目录去取得目标文件的 inode ， 最终就能够读取到正确的数据了。你可以发现的是，如果目标文件(/etc/crontab)被删除了，那么整个环节就会无法继续进行下去， 所以就会发生无法透过连结档读取的问题了！

这里还是得特别留意，这个 Symbolic Link 与 Windows 的快捷方式可以给他划上等号，由 Symbolic link 所创建的文件为一个独立的新的文件，所以会占用掉 inode 与 block 喔！

由上面的说明来看，似乎 hard link 比较安全，因为即使某一个目录下的关连数据被杀掉了， 也没有关系，只要有任何一个目录下存在着关连数据，那么该文件就不会不见！举上面的例子来说，我的 /etc/crontab 与 /root/crontab 指向同一个文件，如果我删除了 /etc/crontab 这个文件，该删除的动作其实只是将 /etc 目录下关于 crontab 的关连数据拿掉而已， crontab 所在的 inode 与 block 其实都没有被变动喔！

不过由于 Hard Link 的限制太多了，包括无法做『目录』的 link ， 所以在用途上面是比较受限的！反而是 Symbolic Link 的使用方面较广喔！好了， 说的天花乱坠，看你也差不多快要昏倒了！没关系，实作一下就知道怎么回事了！要制作连结档就必须要使用 ln 这个命令呢！

|  |
| --- |
| [root@www ~]# **ln [-sf] 来源文件 目标文件**  选项与参数：  -s ：如果不加任何参数就进行连结，那就是hard link，至于 -s 就是symbolic link  -f ：如果 目标文件 存在时，就主动的将目标文件直接移除后再创建！  范例一：将 /etc/passwd 复制到 /tmp 底下，并且观察 inode 与 block  [root@www ~]# **cd /tmp**  [root@www tmp]# **cp -a /etc/passwd .**  [root@www tmp]# **du -sb ; df -i .**  18340 . <==先注意一下这里的容量是多少！  Filesystem Inodes IUsed IFree IUse% Mounted on  /dev/hdc2 2560864 149738 2411126 6% /  # 利用 du 与 df 来检查一下目前的参数～那个 du -sb  # 是计算整个 /tmp 底下有多少 bytes 的容量啦！  范例二：将 /tmp/passwd 制作 hard link 成为 passwd-hd 文件，并观察文件与容量  [root@www tmp]# **ln passwd passwd-hd**  [root@www tmp]# **du -sb ; df -i .**  18340 .  Filesystem Inodes IUsed IFree IUse% Mounted on  /dev/hdc2 2560864 149738 2411126 6% /  # 仔细看，即使多了一个文件在 /tmp 底下，整个 inode 与 block 的容量并没有改变！  [root@www tmp]# **ls -il passwd\***  586361 -rw-r--r-- 2 root root 1945 Sep 29 02:21 passwd  586361 -rw-r--r-- 2 root root 1945 Sep 29 02:21 passwd-hd  # 原来是指向同一个 inode 啊！这是个重点啊！另外，那个第二栏的连结数也会添加！  范例三：将 /tmp/passwd 创建一个符号链接  [root@www tmp]# **ln -s passwd passwd-so**  [root@www tmp]# **ls -li passwd\***  586361 -rw-r--r-- 2 root root 1945 Sep 29 02:21 passwd  586361 -rw-r--r-- 2 root root 1945 Sep 29 02:21 passwd-hd  586401 lrwxrwxrwx 1 root root 6 Oct 22 14:18 passwd-so -> passwd  # passwd-so 指向的 inode number 不同了！这是一个新的文件～这个文件的内容是指向  # passwd 的。passwd-so 的大小是 6bytes ，因为 passwd 共有六个字符之故  [root@www tmp]# **du -sb ; df -i .**  18346 .  Filesystem Inodes IUsed IFree IUse% Mounted on  /dev/hdc2 2560864 149739 2411125 6% /  # 呼呼！整个容量与 inode 使用数都改变啰～确实如此啊！  范例四：删除源文件 passwd ，其他两个文件是否能够开启？  [root@www tmp]# **rm passwd**  [root@www tmp]# **cat passwd-hd**  ......正常显示完毕！  [root@www tmp]# **cat passwd-so**  cat: passwd-so: No such file or directory  [root@www tmp]# **ll passwd\***  -rw-r--r-- 1 root root 1945 Sep 29 02:21 passwd-hd  lrwxrwxrwx 1 root root 6 Oct 22 14:18 passwd-so -> passwd  # 怕了吧！符号链接果然无法开启！另外，如果符号链接的目标文件不存在，  # 其实档名的部分就会有特殊的颜色显示喔！ |

|  |  |
| --- | --- |
| **Tips:** 还记得[第六章](file:///C:\My%20Websites\%E9%B8%9F%E5%93%A5%E7%9A%84linux\cn.linux.vbird.org\linux_basic\0210filepermission.html#dir)当中，我们提到的 /tmp 这个目录是干嘛用的吗？是给大家作为缓存盘用的啊！ 所以，您会发现，过去我们在进行测试时，都会将数据移动到 /tmp 底下去练习～ 嘿嘿！因此，有事没事，记得将 /tmp 底下的一些怪异的数据清一清先！ |  |

要注意啰！使用 ln 如果不加任何参数的话，那么就是 Hard Link 啰！如同范例二的情况，添加了 hard link 之后，可以发现使用 ls -l 时，显示的 link 那一栏属性添加了！而如果这个时候砍掉 passwd 会发生什么事情呢？passwd-hd 的内容还是会跟原来 passwd 相同，但是 passwd-so 就会找不到该文件啦！

而如果 ln 使用 -s 的参数时，就做成差不多是 Windows 底下的『快捷方式』的意思。当你修改 Linux 下的 symbolic link 文件时，则更动的其实是『原始档』， 所以不论你的这个原始档被连结到哪里去，只要你修改了连结档，原始档就跟着变啰！ 以上面为例，由于你使用 -s 的参数创建一个名为 passwd-so 的文件，则你修改 passwd-so 时，其内容与 passwd 完全相同，并且，当你按下储存之后，被改变的将是 passwd 这个文件！

此外，如果你做了底下这样的连结：

ln -s /bin /root/bin

那么如果你进入 /root/bin 这个目录下，『请注意呦！该目录其实是 /bin 这个目录，因为你做了连结档了！』所以，如果你进入 /root/bin 这个刚刚创建的链接目录， 并且将其中的数据杀掉时，嗯！ /bin 里面的数据就通通不见了！这点请千万注意！所以赶紧利用『rm /root/bin 』 将这个连结档删除吧！

基本上， Symbolic link 的用途比较广，所以您要特别留意 symbolic link 的用法呢！未来一定还会常常用到的啦！

* **关于目录的 link 数量：**

或许您已经发现了，那就是，当我们以 hard link 进行『文件的连结』时，可以发现，在 ls -l 所显示的第二字段会添加一才对，那么请教，如果创建目录时，他默认的 link 数量会是多少？ 让我们来想一想，一个『空目录』里面至少会存在些什么？呵呵！就是存在 . 与 .. 这两个目录啊！ 那么，当我们创建一个新目录名称为 /tmp/testing 时，基本上会有三个东西，那就是：

* /tmp/testing
* /tmp/testing/.
* /tmp/testing/..

而其中 /tmp/testing 与 /tmp/testing/. 其实是一样的！都代表该目录啊～而 /tmp/testing/.. 则代表 /tmp 这个目录，所以说，当我们创建一个新的目录时， 『新的目录的 link 数为 2 ，而上一级目录的 link 数则会添加 1 』 不信的话，我们来作个测试看看：

|  |
| --- |
| [root@www ~]# **ls -ld /tmp**  drwxrwxrwt 5 root root 4096 Oct 22 14:22 /tmp  [root@www ~]# **mkdir /tmp/testing1**  [root@www ~]# **ls -ld /tmp**  drwxrwxrwt 6 root root 4096 Oct 22 14:37 /tmp  [root@www ~]# **ls -ld /tmp/testing1**  drwxr-xr-x 2 root root 4096 Oct 22 14:37 /tmp/testing1 |

瞧！原本的所谓上一级目录 /tmp 的 link 数量由 5 添加为 6 ，至于新目录 /tmp/testing 则为 2 ，这样可以理解目录的 link 数量的意义了吗？ ^\_^