的线性查找。组成路径的每个部分都在目录缓存中保存一个dentry对象，并且通过它的i节点查找到后续的路径元素的目录项，直到找到真正的文件i节点。

例如，要通过绝对路径名来查找一个文件（如：/usr/ast/file），需要经过如下步骤。首先，系统定位根目录，它通常使用2号i节点（特别是当1号i节点被用来处理磁盘坏块的时候）。系统在目录缓存中存放一条记录以便将来对根目录的查找。然后，在根目录中查找字符串“usr”，得到/usr目录的i节点号。/usr目录的i节点号同样也存入目录缓存。然后这个i节点被取出，并从中解析出磁盘块，这样就可读取/usr目录并查找字符串“ast”。一旦找到这个目录项，目录/usr/ast的i节点号就可以从中获得。有了/usr/ast的i节点号，就可以读取i节点并确定目录所在的磁盘块。最后，从/usr/ast目录查找“file”并确定其i节点号。因此，使用相对地址不仅对用户来说更加方便，而且也为系统节省了大量的工作。

如果文件存在，那么系统提取其i节点号并以它为索引在i节点表（在磁盘上）中定位相应的i节点，并装入内存。i节点被存放在i节点表中，其中i节点表是一个内核数据结构，用于保存所有当前打开的文件和目录的i节点。i节点表项的格式至少要包含stat系统调用返回的所有域，以保证stat正常运行（见图10-28）。图10-33中列出了i节点结构中由Linux文件系统层支持的一些域。实际的i节点结构包含更多的域，这是由于该数据结构也用于表示目录、设备以及其他特殊文件。i节点结构中还包含了一些为将来的应用保留的域。历史已经表明未使用的位不会长时间保持这种方式。

| 域 | 字节数 | 描　述 |
|---|---|---|
| Mode | 2 | 文件类型、保护位、setuid和setgid位 |
| Nlinks | 2 | 指向该i节点的目录项的数目 |
| Uid | 2 | 文件属主的UID |
| Gid | 2 | 文件属主的GID |
| Size | 4 | 文件大小（以字节为单位） |
| Addr | 60 | 12个磁盘块及其后面3个间接块的地址 |
| Gen | 1 | generation数（每次i节点被重用时增加） |
| Atime | 4 | 最近访问文件的时间 |
| Mtime | 4 | 最近修改文件的时间 |
| Ctime | 4 | 最近改变i节点的时间（除去其他时间） |

图10-33　Linux的i节点结构中的一些域

现在来看看系统如何读取文件。对于调用了read系统调用的库函数的一个典型使用是：

```
n = read(fd, buffer, nbytes);
```

当内核得到控制权时，它需要从这三个参数以及内部表中与用户有关的信息开始。内部表中的项目之一是文件描述符数组。文件描述符数组用文件描述符作为索引并为每一个打开的文件保存一个表项（最多达到最大值，通常默认是32个）。

这里的思想是从一个文件描述符开始，找到文件对应的i节点为止。考虑一个可能的设计：在文件描述符表中存放一个指向i节点的指针。尽管这很简单，但不幸的是这个方法不能奏效。其中存在的问题是：与每个文件描述符相关联的是用来指明下一次读（写）从哪个字节开始的文件读写位置，它该放在什么地方？一个可能的方法是将它放到i节点表中。但是，当两个或两个以上不相关的进程同时打开同一个文件时，由于每个进程有自己的文件读写位置，这个方法就失效了。

另一个可能的方法是将文件读写位置放到文件描述符表中。这样，每个打开文件的进程都有自己的文件读写位置。不幸的是，这个方法也是失败的，但是其原因更加微妙并且与Linux的文件共享的本质有关。考虑一个shell脚本s，它由顺序执行的两个命令p1和p2组成。如果该shell脚本在命令行

```
s >x
```

下被调用，我们预期p1将它的输出写到x中，然后p2也将输出写到x中，并且从p1结束的地方开始。

当shell生成p1时，x初始是空的，从而p1从文件位置0开始写入。然而，当p1结束时就必须通过某种机制使得p2看到的初始文件位置不是0（如果将文件位置存放在文件描述符表中，p2将看到0），而是p1