最近，看到很多文章都在介绍 Linux 中的文件系统，其中就包括：inode 节点、软链接、硬链接等重要的概念。

于是就有小伙伴私信问我：这些概念我都懂，但是我能利用他们来完成什么工作呢？

或者说，在哪些情况下，软链接和硬链接能够提供提供更好的解决方案呢？

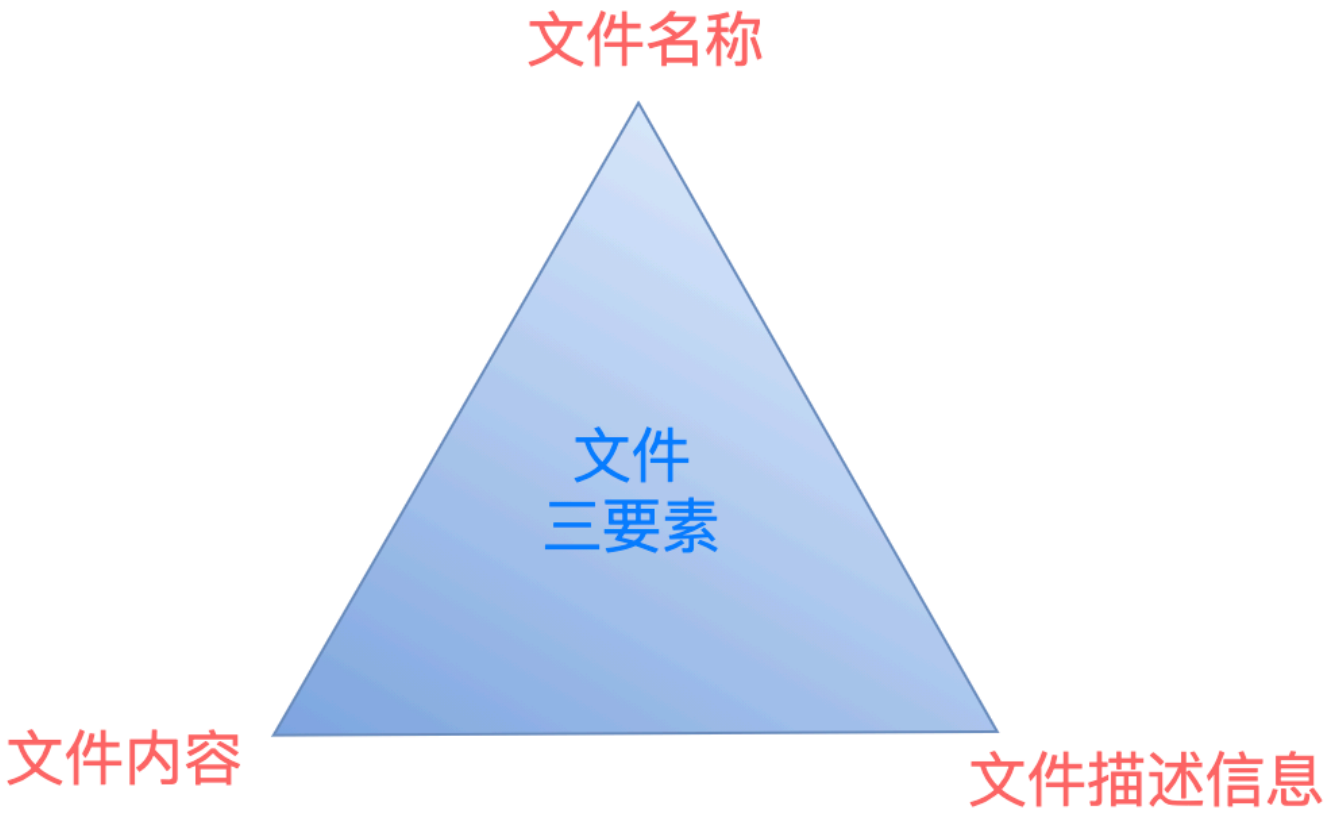
这篇文章我们就来简单梳理一下，软链接和硬链接的几个使用场景。

> 什么是索引节点
> 什么是硬链接
> 什么是软链接

> 软链接应用之：灵活切换不同版本的目标程序
> 软链接应用之：动态库版本管理
> 软链接应用之：快捷方式

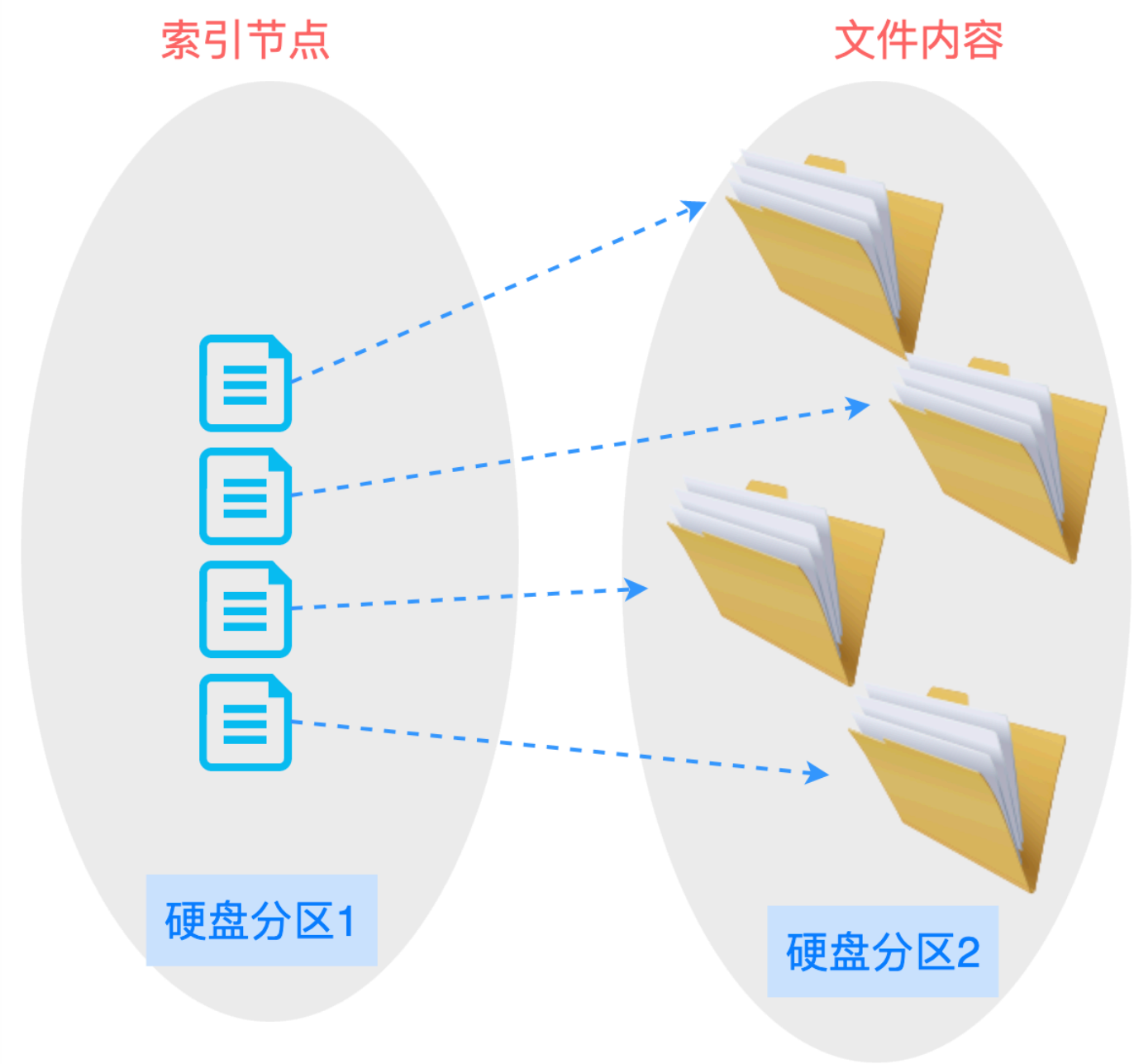
> 硬链接之应用：从不同角度对文件进行分类
> 硬链接应用之：文件多人共享
> 硬链接之应用：文件备份

文件和索引节点 inode



在 Linux 系统中，我们可以把一个文件看做 3 个组成部分：

1. 文件名：从用户角度来描述一个文件;
2. 文件内容：也就是文件中存储的那些数据;
3. 文件的描述信息：文件的类型、所有者、创建时间等等，可以称之为元信息;



可以简单的做一个类比：

文件本身的内容，可以看做一个实实在在的人。

文件的描述信息，可以看做是派出所里的户籍卡。

类似
文件索引节点

类似
文件内容

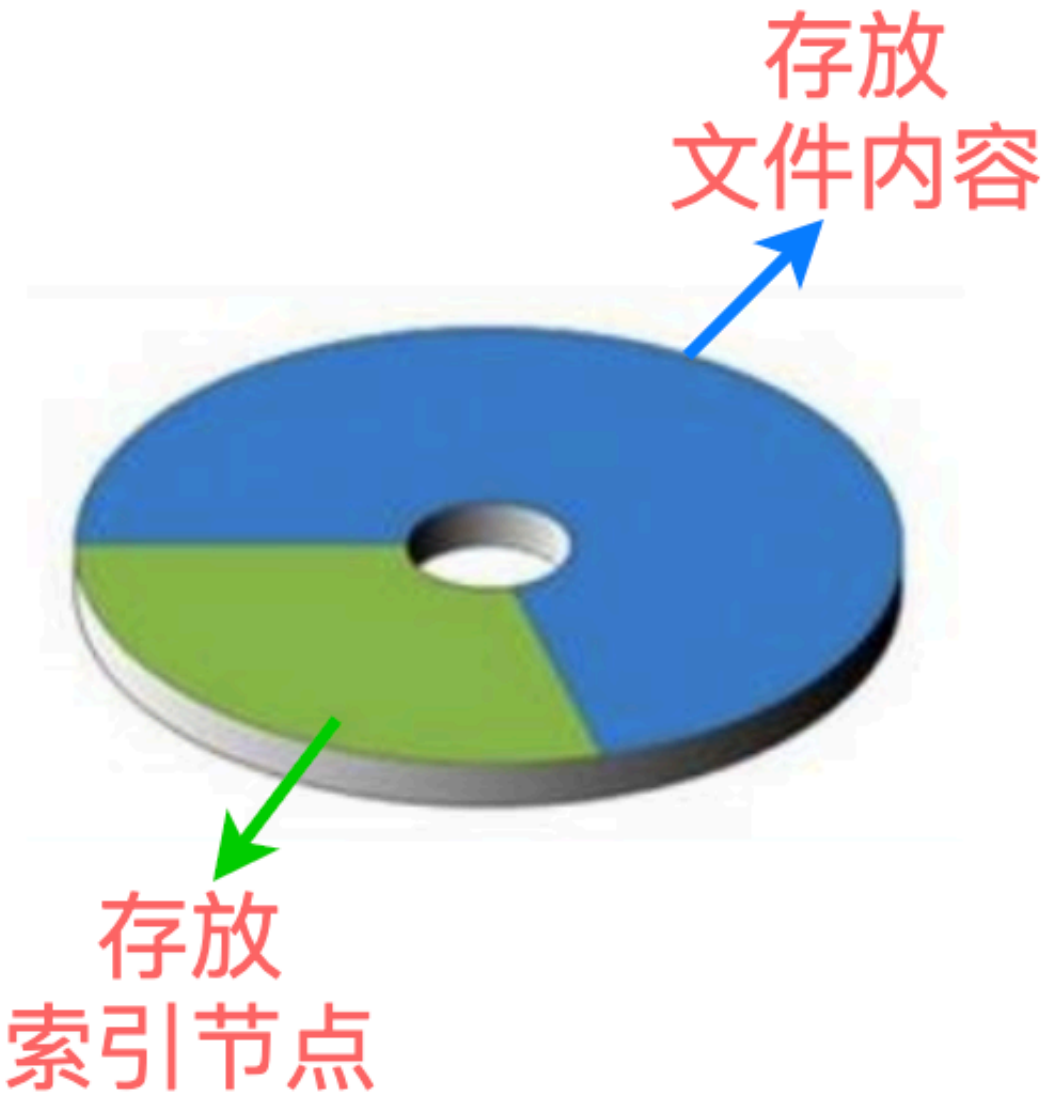
户籍卡上记录了一个人的姓名、年龄、住址等信息，警察叔叔通过这个户籍卡，就知道这个人的一切描述信息，除了你脑袋里的知识。

回到计算机中，文件的所有信息都需要存储在硬盘上，因此就要对硬盘进行区域划分：不同的区域存储不同类型的数据，这就是文件系统的重要作用。

在 Linux 系统使用的 ext2/ext3 文件系统中，从硬盘上划分一块区域，用来存放文件本身的内容(数据)，这块区域按照一个最小单位：块(block)来进行划分。

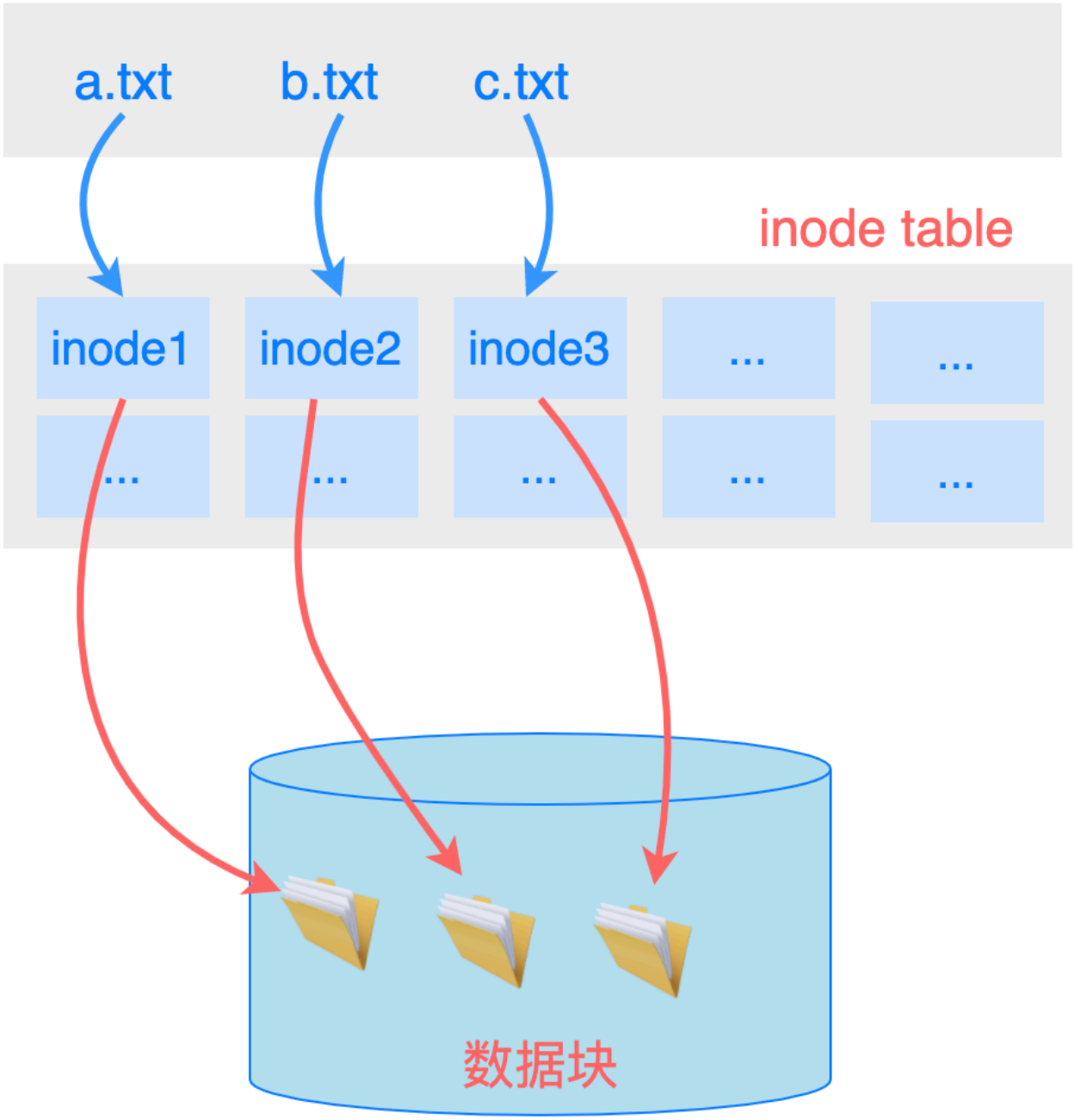
然后从硬盘上划分出另一块区域，专门用来存放所有文件的描述信息。

每一个文件的描述信息，都用一个名为索引节点(inode)的数据结构来表示，所有文件的 inode 就统一放在这块硬盘区域中。



就像户籍卡上记录了一个人的住址一样，一个文件的索引节点(inode)中，也记录了这个文件的所有描述信息，包括：文件类型、所有者、创建时间等待，当然也包括文件内容存储在硬盘的哪些块(block)中。

当我们调用打开文件 API 函数的时候，操作系统首先根据传入的文件路径，找到这个文件的 inode，然后进行一系列的权限检查操作，最后从 inode 中获得这个文件的内容存储在哪些块(block)中，从而可以对文件的内容进行读取、写入操作。



文件名称只是给我们用户来使用的，操作系统只是通过 inode 节点，来对文件进行管理的。

当我们创建一个新文件的时候，就同时创建了这个文件对应的 inode 节点。

当我们删除一个文件的时候，就同时删除了这个文件对应的 inode 节点。

此时，文件本身内容所在的那个块中，数据并不会被抹除掉，因此有些数据恢复软件就是利用这个特点来进行数据找回。

一句话总结：索引节点(inode)就像户籍卡，操作系统通过 inode 来管理所有的文件。

硬链接

刚才已经说到，每一个文件都对应一个 inode 节点。

例如有一个文件 a.txt，文件内容长度是 1024 个字节，存放在硬盘上的某个块(block)中，假设就是第 10000 个块吧。

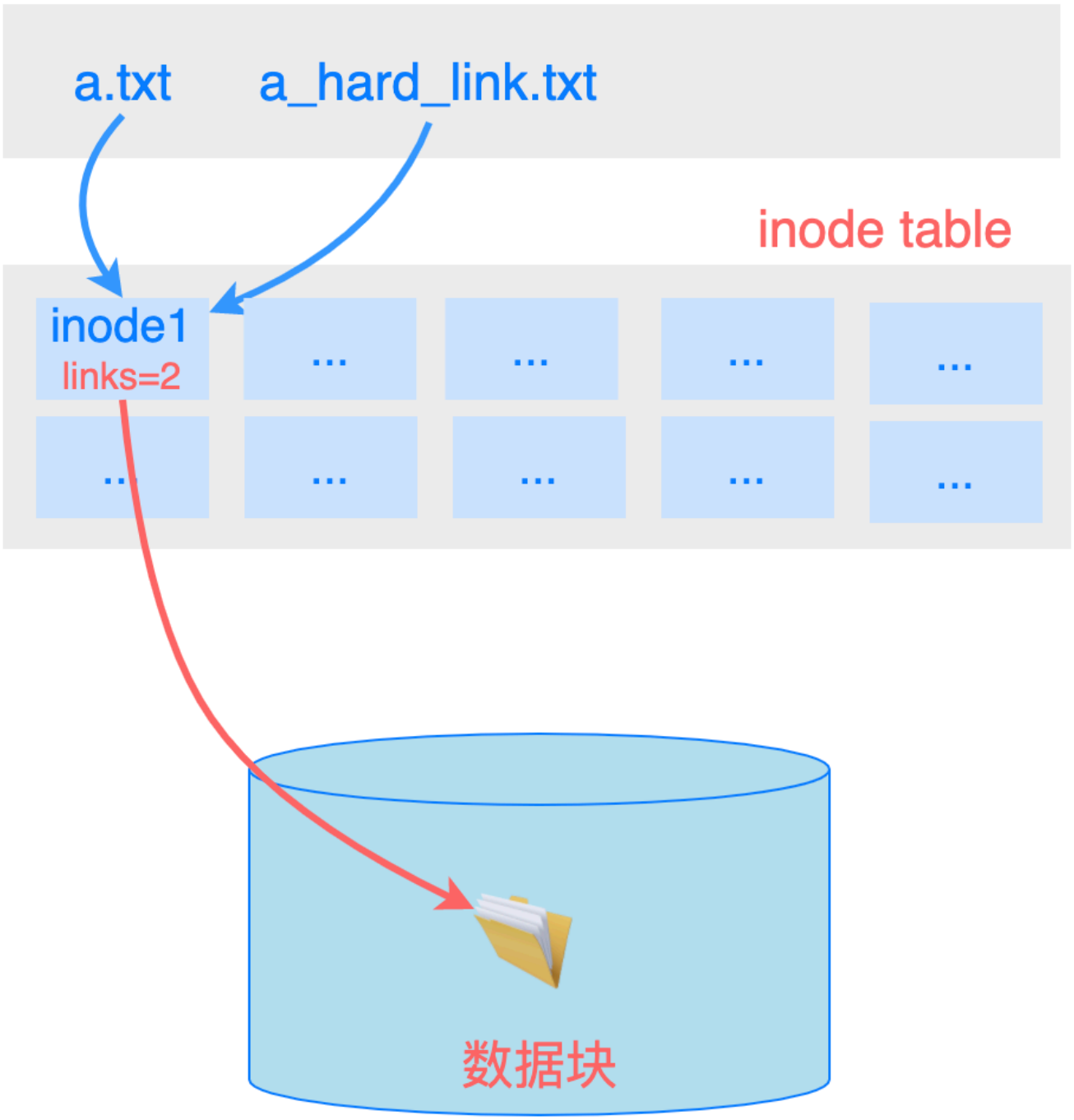
那么这个文件对应的 inode 节点中，就会把 10000 这个块记录下来。

同时，它还有一个 links 字段，表示：当前这个 inode 对应一个文件，此时 inode.links 的值为 1。

此时，如果我们用另一个文件名 a_hard_link.txt，也来表示 a.txt 这个文件。

也就是说：虽然我们用了 2 个文件名称，但是本质上指向同一个文件，内容都指向第 10000 个块中存储的文件内容。

Linux 系统中提供了硬链接来支持这样的目的，它仅仅是把 inode 节点中的 links 字段的值 加1 即可，也就是 inode.links 的值变成了 2。



硬链接的操作指令是：

```
$ ln a.txt b.txt
```

基于硬链接，用户就可以用不同的文件名来访问同一个文件，所有的操作最终修改的都是同一个文件。

如果仅仅从用户的角度来看，好像我们是在操作不同的文件，但是这些文件具有自动同步的功能。

这个行为有点类似于网盘：

在云存储中有一个文件 `hello.txt`，然后我有两台电脑 A 和 B，这两台电脑会把云端的文件 `hello.txt` 都创建一个镜像文件在本地，就好像这个文件就在自己的硬盘上一样。

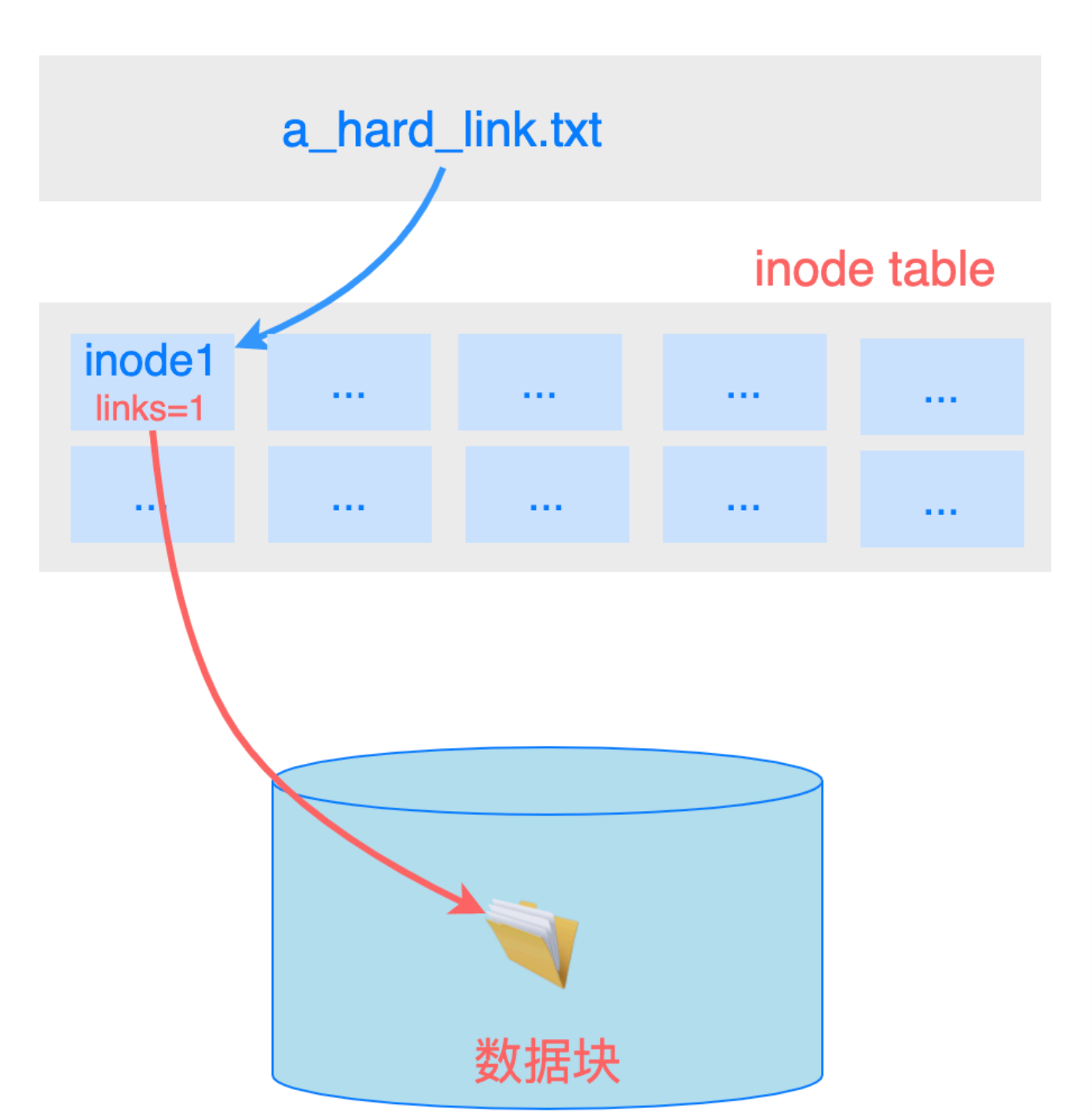
当我在电脑 A 上操作 hello.txt 时，电脑 B 中的同名文件会自动更新。

因此，从行为上来看，硬链接就相当于是：文件拷贝 + 自动同步。

再来看一下硬链接文件的删除操作。

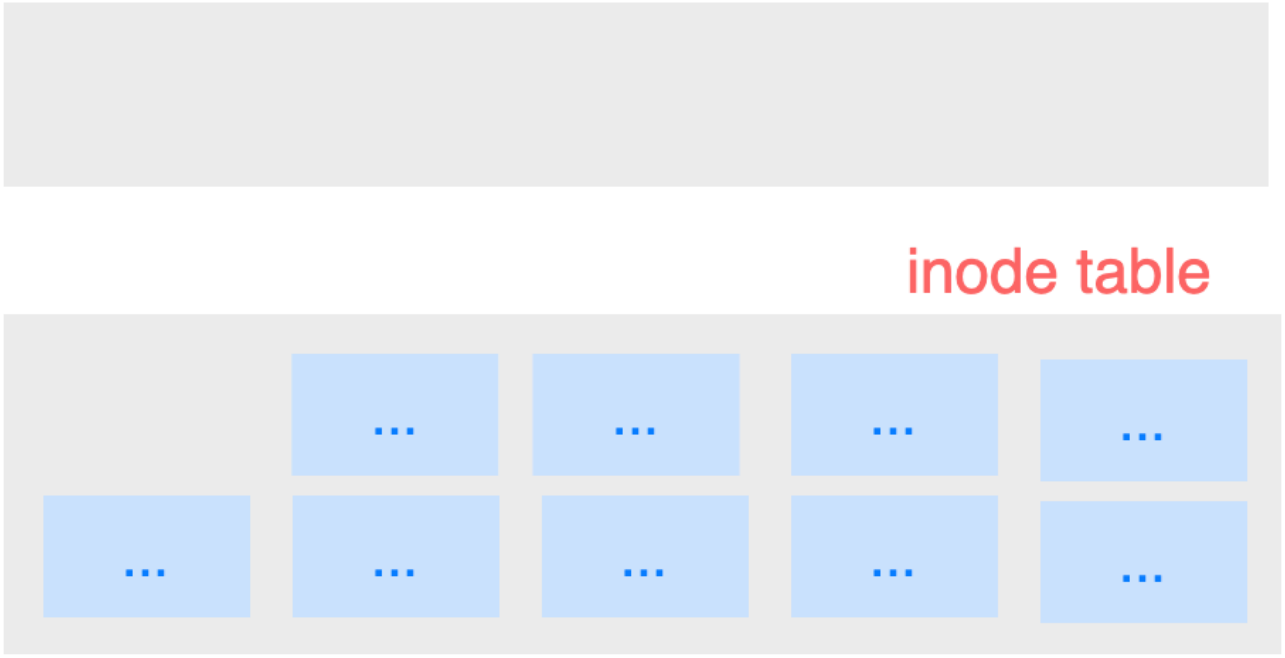
在执行 $ ln a.txt a_hard_link.txt 指令之后，该文件对应的 inode 节点中，links 的值为 2。

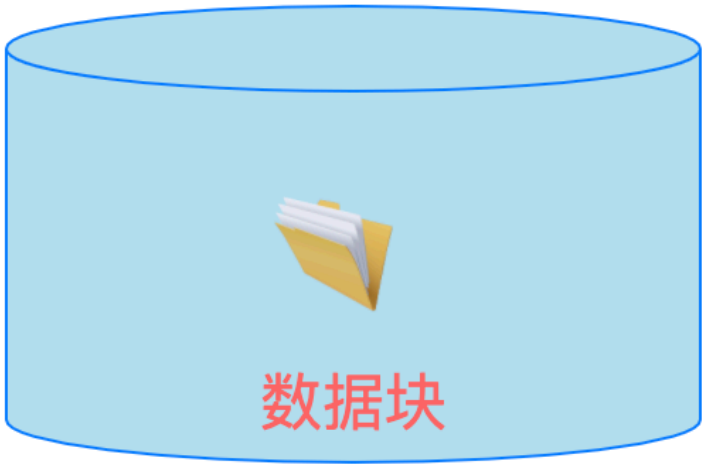
如果我们删除 a.txt，操作系统会把该文件对应的 inode 中的 links 值减1，结果为 1，操作系统发现不为 0，因此并不会删掉这个 inode。



如果我们再删除 a_hard_link.txt，操作系统再次执行 inode.links 减1 动作，发现值变成了 0，于是就把这个 inode 删除了，于是这个文件就彻底不存在了。

这就相当于把一个人的户籍卡给注销掉了，从户籍管理角度看，这个人就不存在了。即使存在，也是一个黑户。





硬链接存在 2 个限制：

> 1. 不允许用户给目录创建硬链接，即：用户不可以，操作系统可以(想一下每个目录下的 . 和 ..);
> 2. 只有在同一个文件系统中的文件，才能创建硬链接，也就是说：不能跨文件系统;

软链接

为了克服硬链接的 2 个限制，软链接被引入进来了。

软链接也叫符号链接，它是一个独立的文件。

软链接文件的内容是一个文本字符串，存储的是目标文件(即：链接到的文件)的路径名。

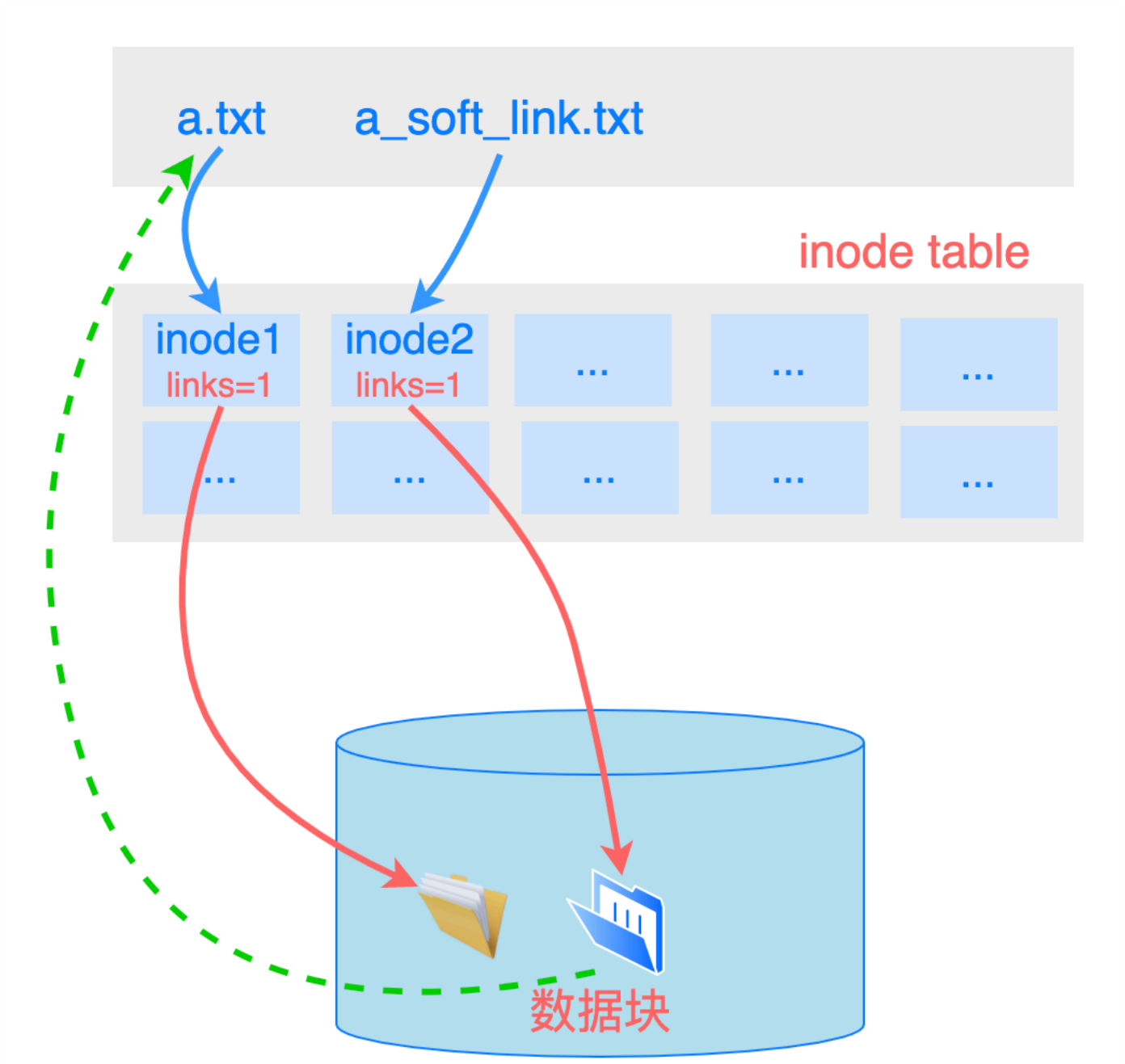
这个路径名可以指向任意一个文件系统的任意文件或者目录，甚至可以指向一个不存在的文件。

与创建硬链接不同的是：当我们创建了一个软链接之后，操作系统会创建一个新的 inode 来表示这个软链接文件。

例如有一个文件 `a.txt`，我们创建一个软链接 `a_soft_link.txt` 来指向它：

```
$ ln -s a.txt a_soft_link.txt
```

此时，`a.txt` 和 `a_soft_link.txt` 各自都有自己的 `inode` 节点。



图中的绿色虚线，就表示软链接文件中的文件路径。

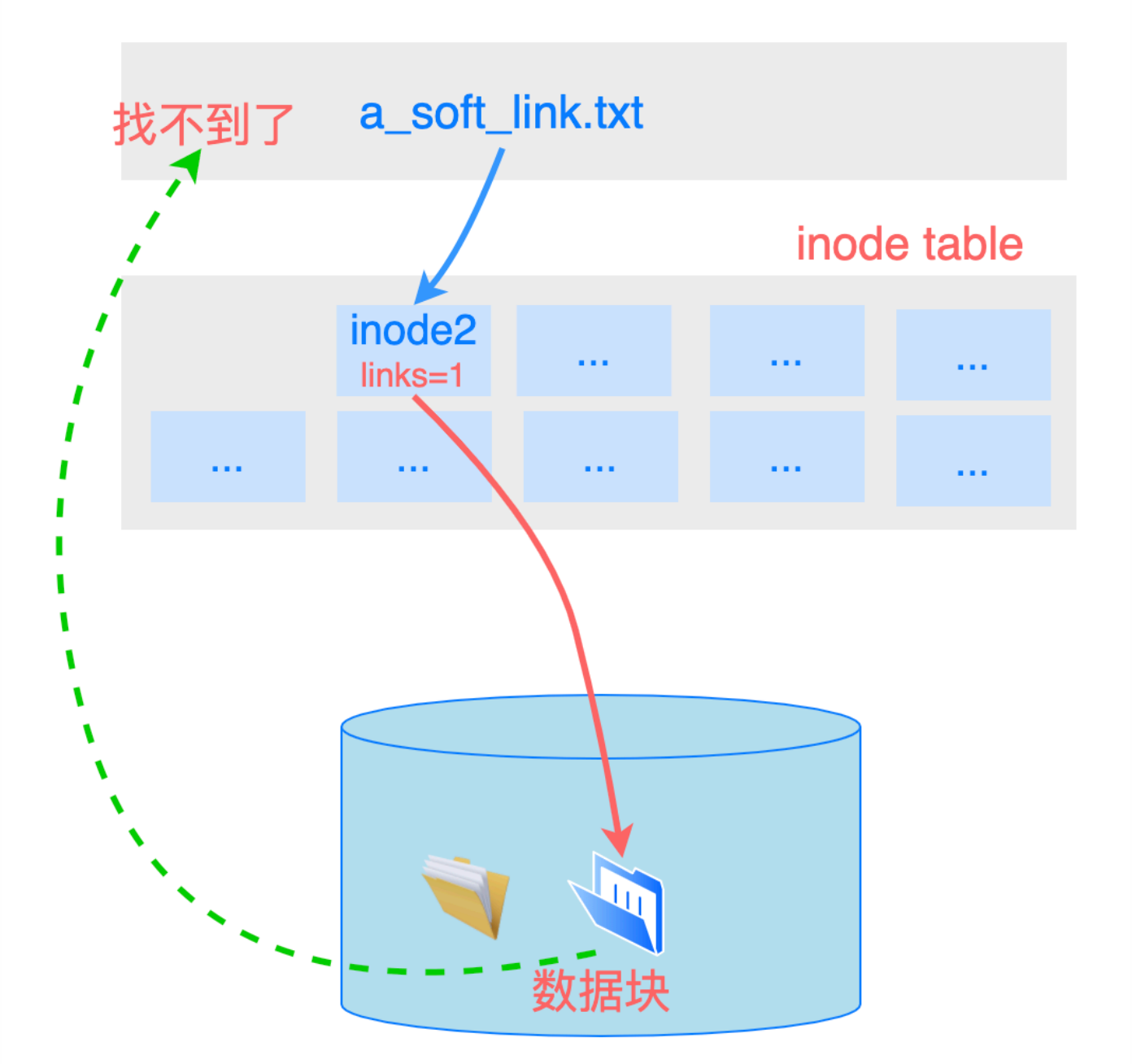
正因为软链接文件中存储的仅仅是目标文件的路径字符串，所以可以表示任意一个文件系统中的文件，或者是目录。

当我们打开文件软链接 a_soft_link.txt 时，操作系统从 a_soft_link.txt 对应的 inode 数据结构中发现：这是一个软链接文件。

于是操作系统就根据其中的路径信息，找到 a.txt 的 inode 节点，从而对最终的目标文件进行操作。

再来看一下软链接文件的删除操作。

如果我们把目标文件 a.txt 删除掉之后，inode 节点会被删除掉，就相当于它的户籍卡被注销掉了。



此时再次打开软链接 a_soft_link.txt 时，虽然其中的路径信息仍然存在，但是系统此时却找不到 a.txt 对应的 inode 节点了。

因此，软链接就类似于与 Windows 系统中的快捷方式。

当真正的目标文件被删除之后，快捷方式也就没有存在的意义了。

软链接应用之：灵活切换不同版本的目标程序

在开发的过程中，对于同一个工具软件，可能要安装多个不同的版本，例如：Python2 和 Python3， JDK8 和 JDK9 等等。

此时就可以通过软链接来指定当前使用哪个版本。例如在我的电脑中：

```
$ ll -l /usr/bin/python*
lrwxrwxrwx 1 root root       9 12月 31 08:19 /usr/bin/python -> python2.7*
lrwxrwxrwx 1 root root       9 12月 31 08:19 /usr/bin/python2 -> python2.7*
-rwxr-xr-x 1 root root 3492624 3月   2 04:47 /usr/bin/python2.7*
lrwxrwxrwx 1 root root       9 12月 31 08:19 /usr/bin/python3 -> python3.5*
-rwxr-xr-x 2 root root 4456208 1月  27 02:48 /usr/bin/python3.5*
```

当在终端窗口中输入：python 时，启动的是 python2.7 版本。

如果有一天我需要使用 python3.5 版本，只需要把软链接 python 指向 python3.5 即可。

软链接应用之：动态库版本管理

在 Linux 系统的动态库版本管理中，有一个 SONAME 的概念。

我们在编译一个动态链接库时，一般使用如下编译命令：

```
$ gcc -fPIC -shared -o libhello.so hello.c
```

在使用这个动态库时，需要链接这个库：-llibhello。

简单的 demo 可以这么来写，但是如果遇到一些比较大的项目，需要执行严格的版本管理，那应该怎么来操作呢?

Linux 系统已经为我们想到了问题的解决方案，利用 SO-NAME。

首先，在编译动态链接库文件时，就指定产生 SO-NAME，它会被存储在动态链接库 ELF 文件中。

我们来直接看一个优秀的开源工具 libevent 的例子：

```
$ ll /usr/lib/libevent-2.1.so*
lrwxrwxrwx 1 root root     17 Jul 27  2020 /usr/lib/libevent-2.1.so -> libevent-
2.1.so.7
lrwxrwxrwx 1 root root     21 Jul 27  2020 /usr/lib/libevent-2.1.so.7 -> libevent-
2.1.so.7.0.1
-rw-r--r-- 1 root root 412016 Jul 27  2020 /usr/lib/libevent-2.1.so.7.0.1
```

此时使用 readelf 命令来查看生成的动态库文件 libevent-2.1.so.7.0.1：

```
$ readelf -a libevent-2.1.so.7.0.1 | grep SONAME
 0x000000000000000e (SONAME)             Library soname: [libevent-2.1.so.7]
```

它这么做有什么好处呢?

Linux 系统在查找动态链接库文件时，会到下面这 3 个默认目录下查找(当然然还有其他目录，比如：当前目录，LD_LIBRARY_PATH 指定的目录)

> /lib: 存放操作系统最关键和基础的库文件;
>
> /usr/lib: 存放一些非系统运行时所需要的关键库文件;
>
> /usr/local/lib: 存放用户自己安装的一些第三方库文件;

系统中安装的所有动态链接库，借助 ldconfig 这个程序，会自动的创建、更新或者删除对应的 SONAME(它是一个软链接，链接到 实际的库文件)，并把这些 SONAME 汇总到一个文件 /etc/ld.so.cache 中缓存起来。

这样，当动态库加载器查找动态库文件时，就可以直接在这个缓存文件中进行查找，加快了动态库的查找速度。

软链接应用之：快捷方式

利用软链接的快捷方式功能就比较好理解了，想一想：我们为什么在 Windows 的桌面上创建很多软件的快捷方式啊?

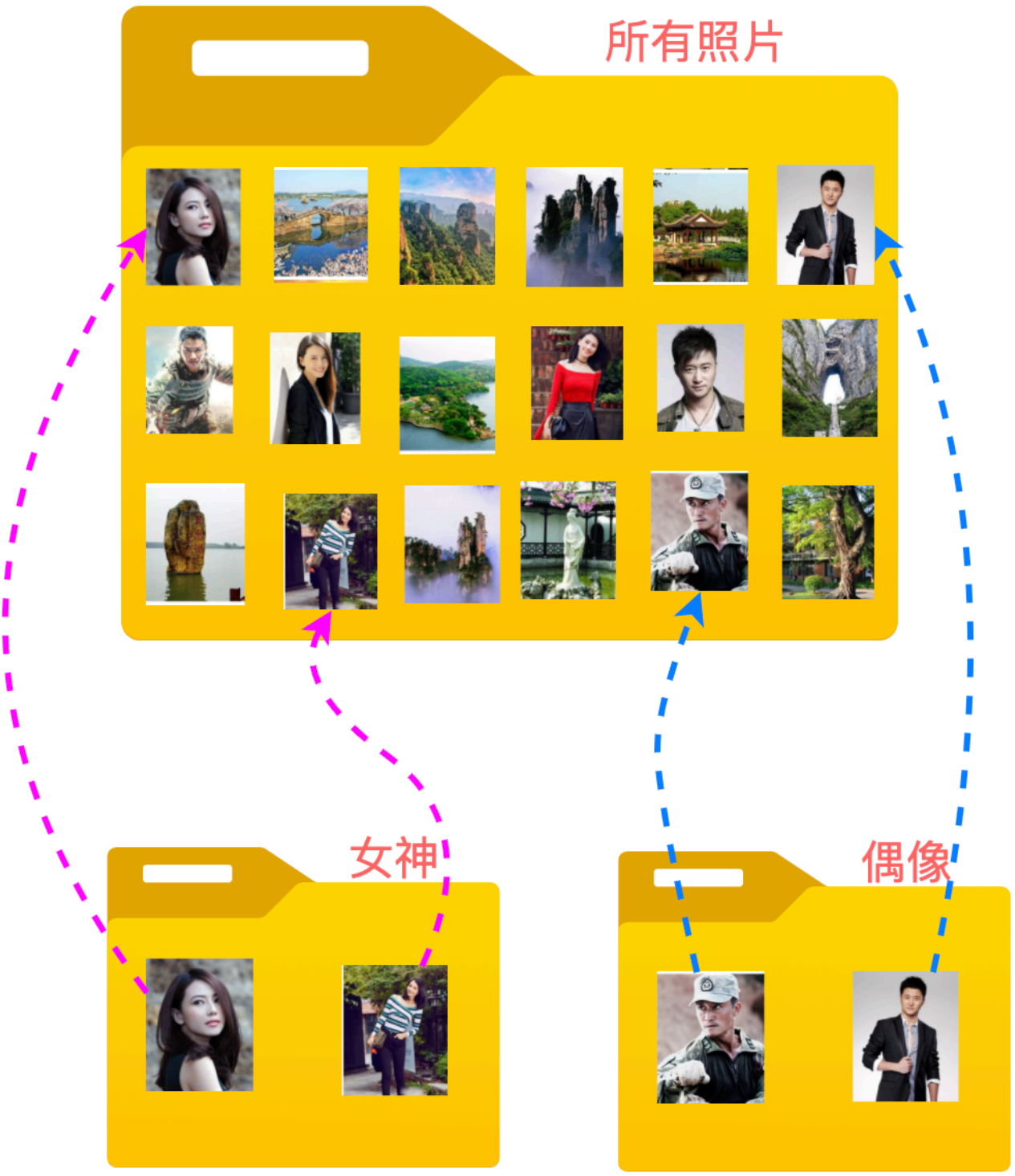
在 Linux 中同样如此！

比如：最近一段时间的工作，每次都要打开一个路径很深的文件。

如果在资源管理器中，一层一层的点击鼠标，是不是比较浪费时间。

此时，就可以在桌面上创建一个软链接，每次直接双击就打开所链接的目标文件了。

硬链接之应用：从不同角度对文件进行分类



比如我有一个文件夹，存储了10 个G的照片。

这些照片中的人物、拍照地点、拍照时间都是不一样的。

现在，我既想根据照片中的人物进行分类，也想根据拍照地点进行分类，还想根据拍照时间进行分类，那该怎么办？

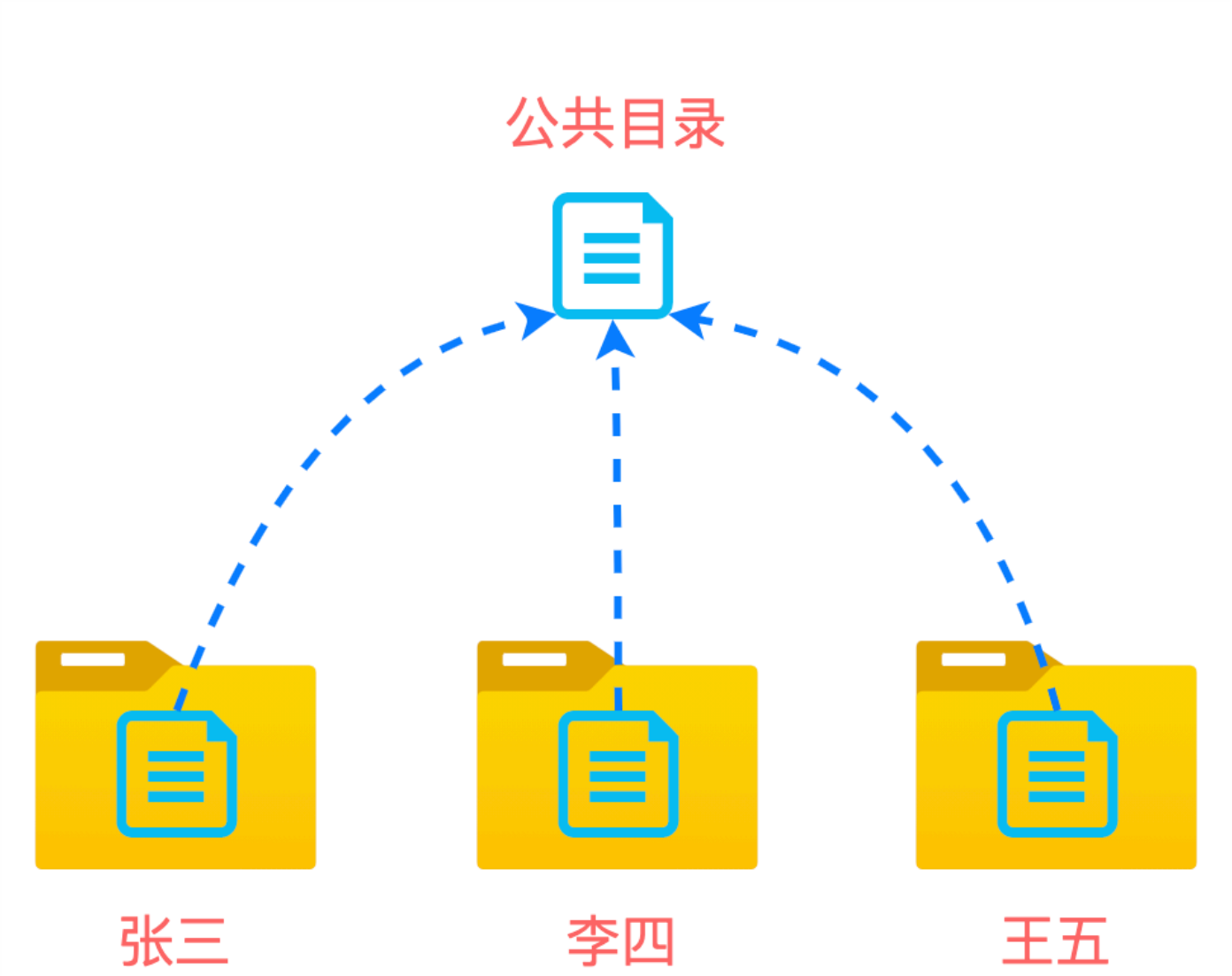
因为一张照片可能同时属于多个不同的分类，难道每个分类中都复制一张照片？这样也太浪费硬盘空间了！

解决方案是：

所有的照片仍旧放在一个总的文件夹中，然后创建不同的分类文件夹，在每个分类文件夹中，创建硬链接到目标照片文件。

这样的话，不仅对照片进行了分类，而且一点都不占用硬盘空间。

硬链接应用之：文件多人共享



当很多人同时对同一个文件进行维护的时候，如果大家都直接操作这个文件，万一不小心把文件删除了，大家就都玩完了！

此时，可以在每个人自己的私人目录中，创建一个硬链接。

每次只需要对这个硬链接文件进行操作，所有的改动会自动同步到目标文件中。

由于每个人都是操作硬链接文件，即使不小心删除了，也不会导致文件的丢失。

因为删除硬链接文件，仅仅是把该文件的 inode 节点中的 links 值减 1 而已，只要不为 0，就不会真正的删除文件。

硬链接之应用：文件备份

一些小伙伴有定期备份文件、清理文件的好习惯。

在备份的时候，如果是实实在在的拷贝一份，那真的是太浪费磁盘空间，特别是对于我这种只有 256G 硬盘空间的笔记本。

此时，就可以利用硬链接功能，既实现文件备份的目的，又节省了大量的硬盘空间，一举两得！

很多备份工具利用的就是硬链接的功能，包括 `git` 工具，当克隆本地的一个仓库时，执行 `clone` 指令：

```
git clone --reference <repository>
```

`git` 并不会把仓库中的所有文件拷贝到本地，而仅仅是创建文件的硬链接，几乎是零拷贝！

------ End ------

推荐阅读

【1】C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻

【2】一步步分析-如何用C实现面向对象编程

【3】原来gdb的底层调试原理这么简单

【4】内联汇编很可怕吗？看完这篇文章，终结它！

【5】都说软件架构要分层、分模块，具体应该怎么做

星标公众号，能更快找到我！

C/C++、物联网、嵌入式、Lua语言

Linux 操作系统、应用程序开发设计

扫码关注公众号

道哥 个人微信

喜欢请分享，满意点个赞，最后点在看。