一方面，如果在更新日志记录的过程中（图 7-12 中的步骤②）被强制切断电源，就只需丢弃日志区域的数据即可，数据本身依旧是开始处理前的状态，如图 7-13 所示。

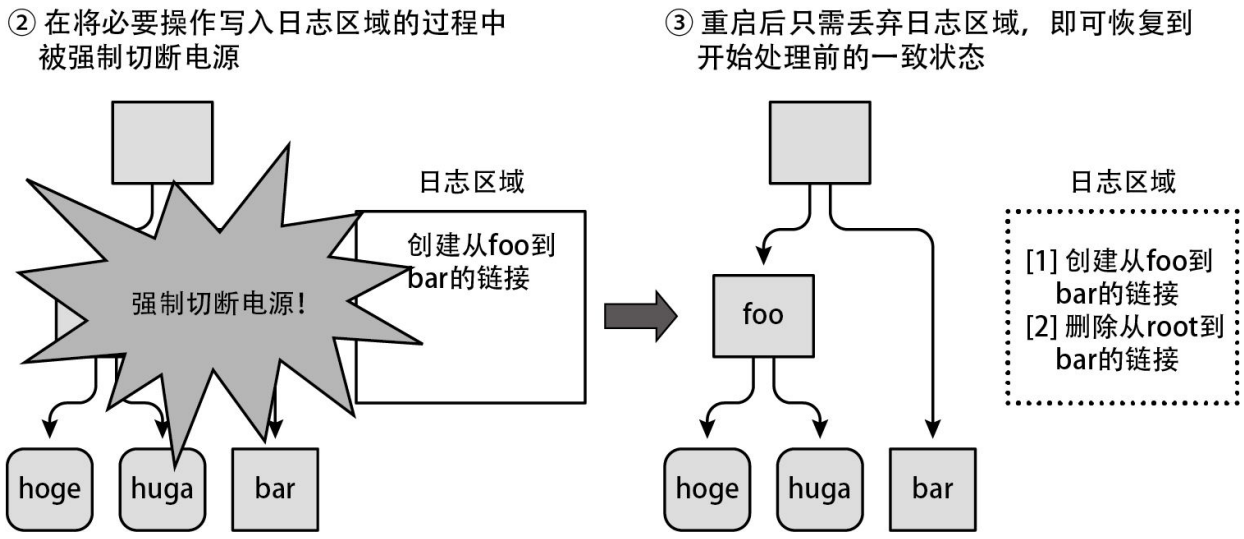

图 7-13　利用日志防止不一致（1）

另一方面，如果在实际执行数据更新的过程中（图 7-12 中的步骤④）被强制切断电源，那么只需按照日志记录从头开始执行一遍操作，即可完成文件系统的处理，如图 7-14 所示。

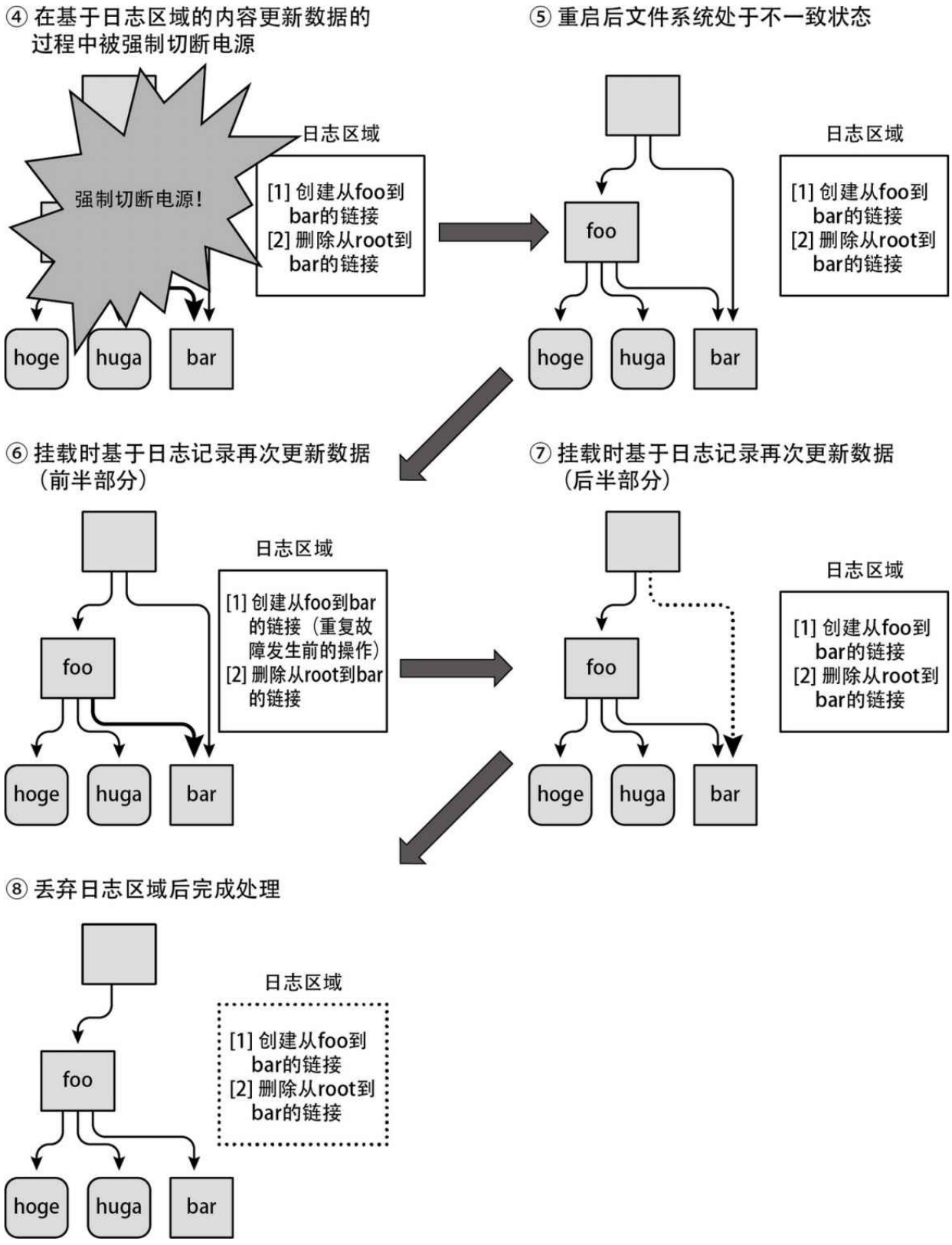

图 7-14　利用日志防止不一致（2）

在这两种情况下，都能避免文件系统出现不一致的情况，且能恢复到处理前的状态，或者正常执行处理后的状态。

7.6 写时复制

在介绍写时复制如何防止发生不一致之前，首先需要介绍一下文件系统是如何收纳数据的。

在 ext4 和 XFS 等传统的文件系统上，文件一旦被创建，其位置原则上就不会再改变了。即便在更新文件内容时，也只会在外部存储器的同一位置写入新的数据，如图 7-15 所示。

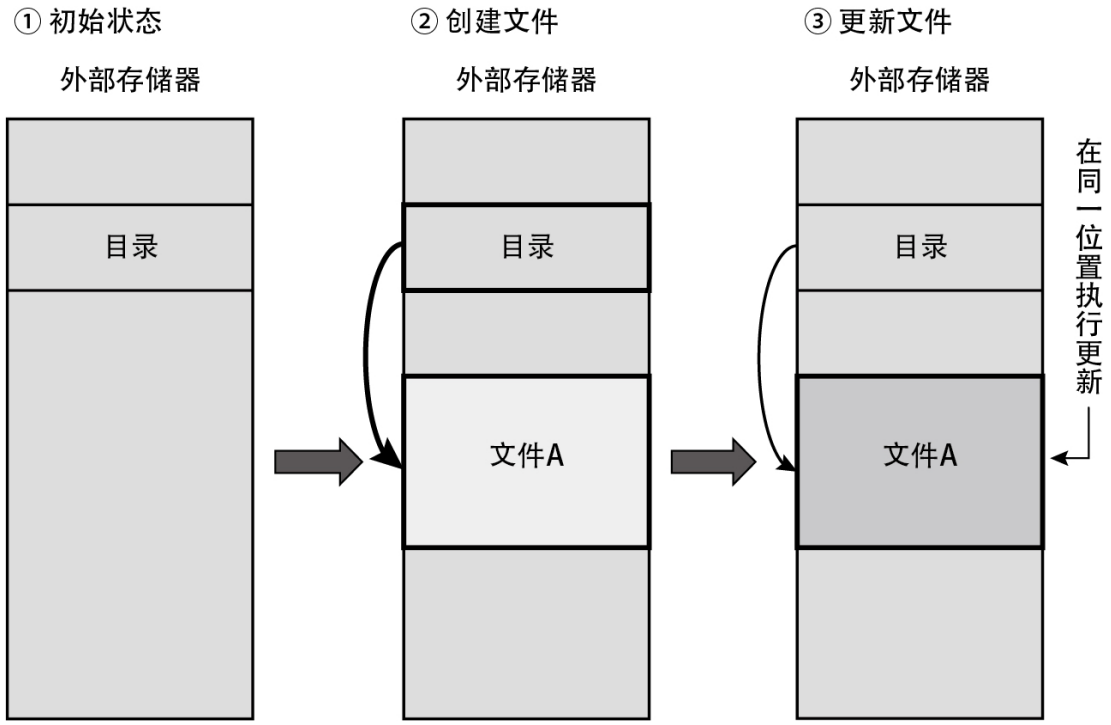

图 7-15 非写时复制方式的文件系统更新处理

与此相对，在 Btrfs 等利用写时复制的文件系统上，创建文件后的每一次更新处理都会把数据写入不同的位置[4]，如图 7-16 所示。

[4]在图 7-16 中，为了便于说明，我们更新了整个文件，但实际上只有更新后的数据会被复制到别的位置。

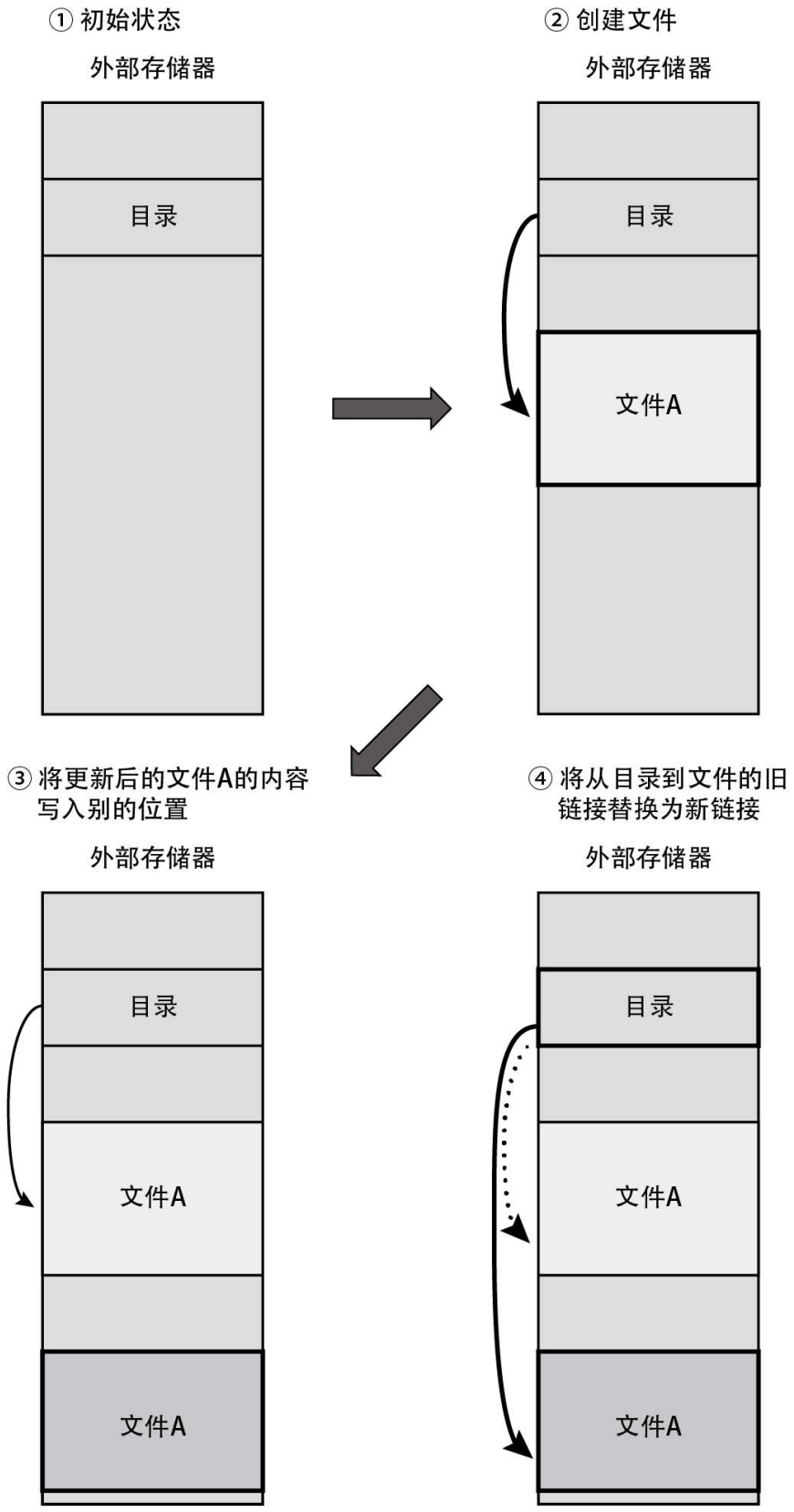
① 初始状态
外部存储器
目录
② 创建文件
外部存储器
目录
文件A
③ 将更新后的文件A的内容
写入别的位置
外部存储器
目录
文件A
文件A
④ 将从目录到文件的旧
链接替换为新链接
外部存储器
目录
文件A
文件A

图 7-16　写时复制方式的文件系统中的简单的更新处理

图 7-16 所示为更新单个文件时的情况，在执行作为原子操作的多个处理时，也同样是先把更新后的数据写入别的位置，然后替换旧的链接，如图 7-17 所示。

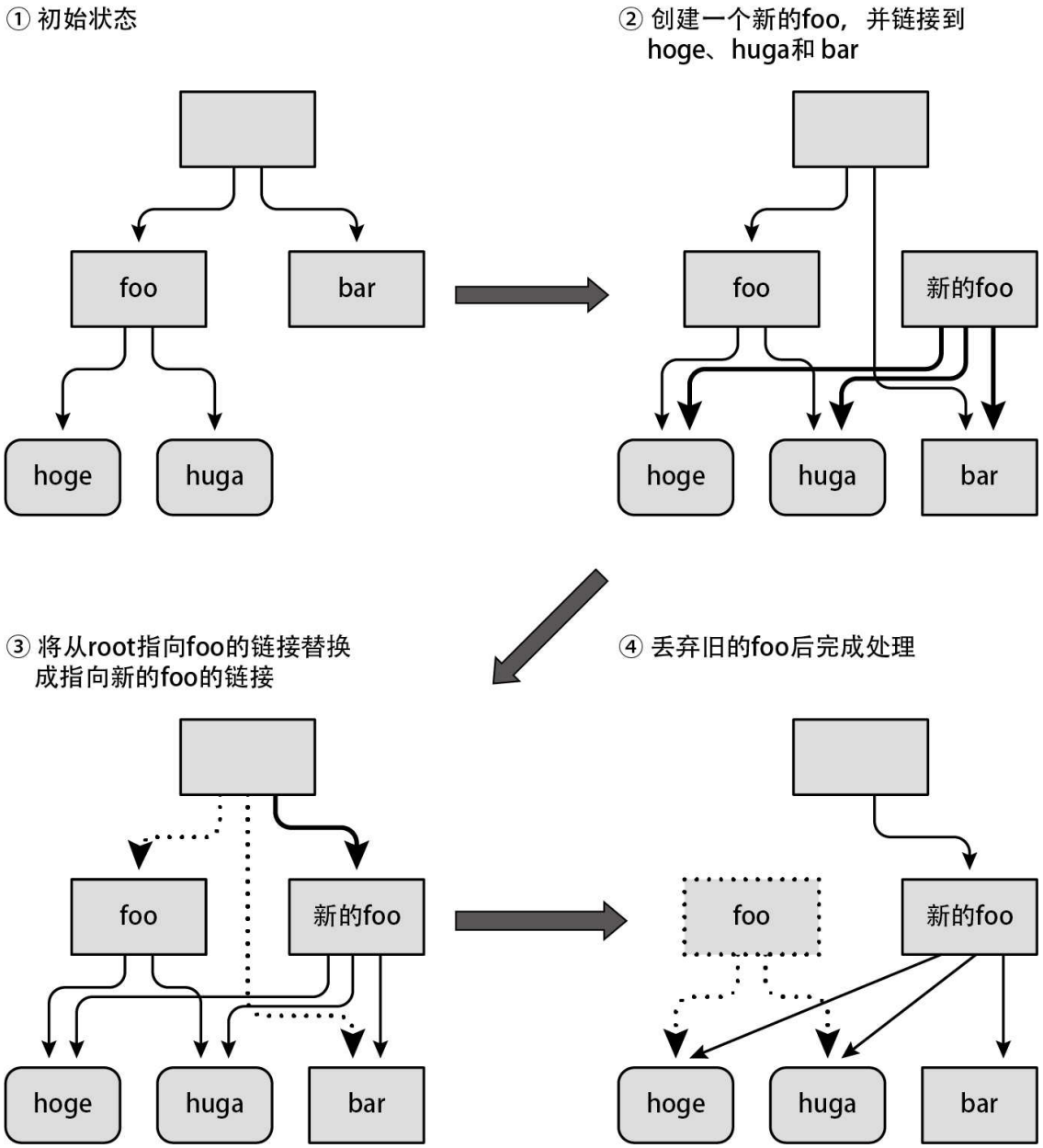

图 7-17　写时复制方式的文件系统中的复杂的更新处理

即使在执行步骤②时被强制切断电源，只要在重启后删除未处理完的数据，也就不会导致不一致的情况出现，如图 7-18 所示。

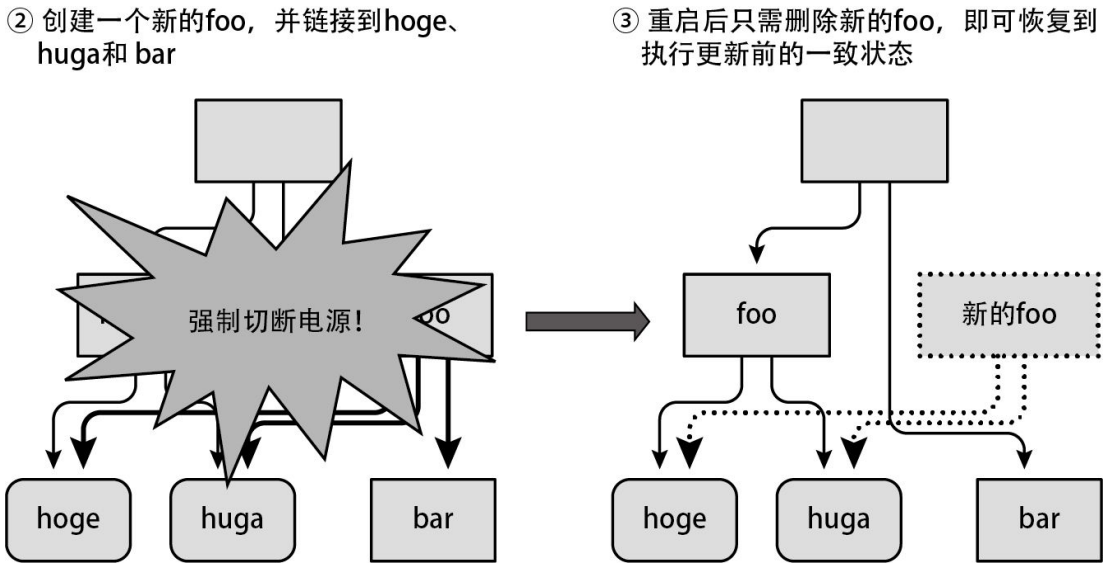

图 7-18　利用写时复制防止不一致

7.7 防止不了的情况

借助前面介绍的机制，近几年已经很少发生文件系统不一致的情况了，但是由文件系统的 Bug 导致的不一致问题依旧偶尔会发生。

7.8 文件系统不一致的对策

万一文件系统上出现了不一致的情况，要怎样应对呢？

一般的做法是定期备份文件系统，当发生文件系统不一致时，可以直接还原到最近备份的状态。

如果平常出于某些原因没有执行定期备份，可以利用各文件系统提供的恢复命令。

各文件系统提供的恢复命令都不一样，但是所有文件系统都会提供一个通用的 `fsck` 命令（在 ext4 上为 `fsck.ext4`，在 XFS 上为 `xfs_repair`，在 Btrfs 上为 `btrfs check`）。该命令有可能将文件系统还原到一致的状态。但笔者不太推荐使用 `fsck`，原因如下。

- 该命令会遍历整个文件系统，以检查文件系统的一致性，并修复不一致的地方，因此随着文件系统使用量的增加，运行时间也会不断增加。如果对一个非常大的文件系统执行该命令，将可能耗费几个小时甚至几天时间
- 耗费这么长时间进行的修复工作也经常以失败告终
- 即便修复成功，也不一定能恢复到用户期待的状态。说到底，**`fsck`** 命令也只是将发生数据不一致的文件系统强行改变到可以挂载的状态而已。在这个过程中，一切不一致的数据与元数据都将被强制删除，如图 7-19 所示

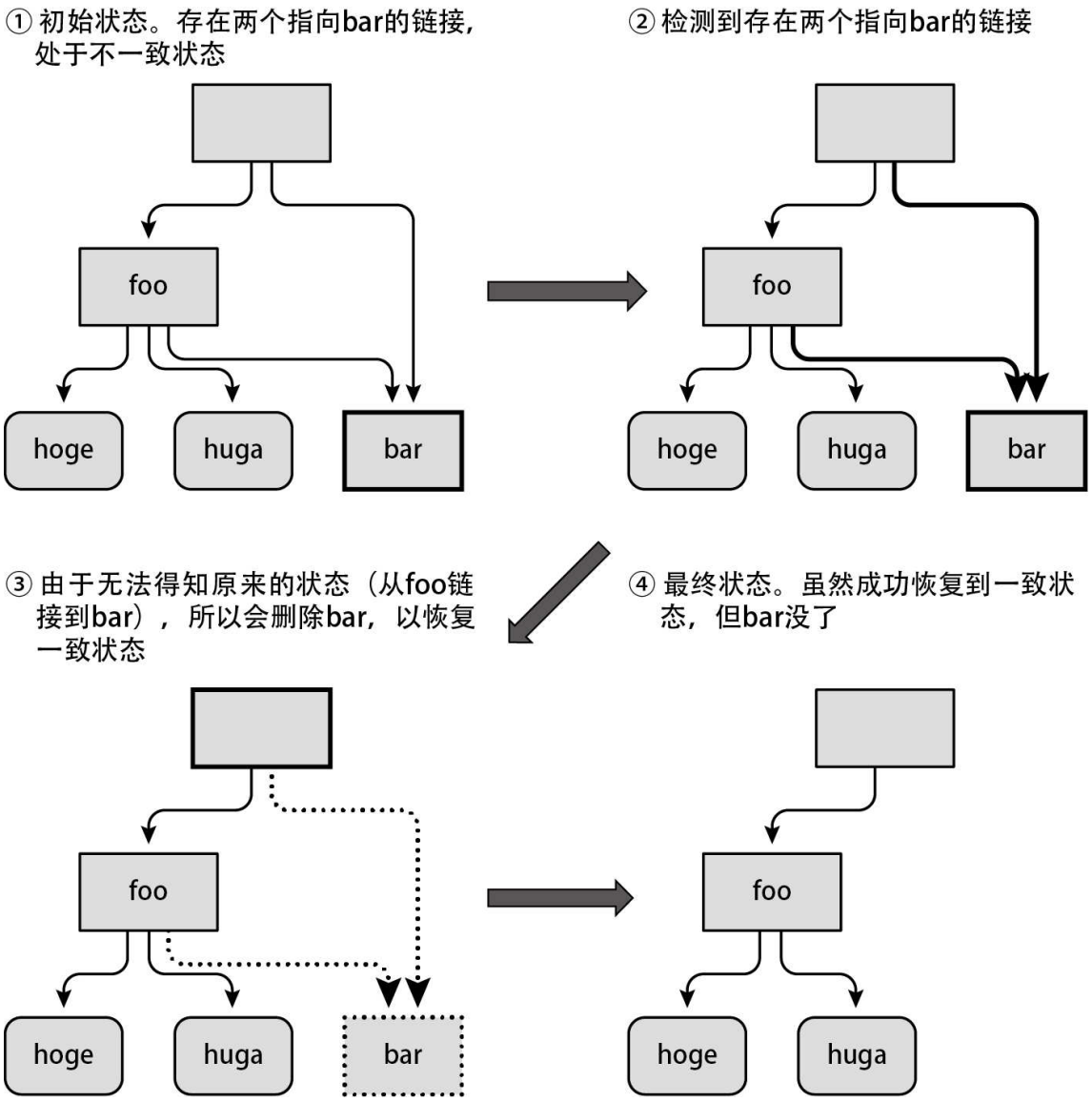

图 7-19　fsck 的处理

由于在使用 fsck 时会存在上述几个问题，所以最佳解决方案还是**执行定期备份**。

7.9 文件的种类

我们在前面提到了文件的两种类型：保存用户数据的普通文件，以及保存其他文件的目录。在 Linux 中还有一种文件，称为**设备文件**。

Linux 会将自身所处的硬件系统上几乎所有的设备呈现为文件形式[5]。因此在 Linux 上，设备如同文件一般，可以通过 open()、read()、write() 等系统调用进行访问。在需要执行设备特有的复杂操作时，就使用 ioctl() 系统调用。在通常情况下，只有 root 用户可以访问设备文件。

[5]网络适配器是例外，并不存在与之对应的文件。

虽然设备也存在很多种类，但 Linux 将以文件形式存在的设备分为两种类型，分别为**字符设备**与**块设备**。所有设备文件都保存在 /dev 目录下。通过设备文件的元数据中保存的以下信息，我们可以识别各个设备。

- **文件的种类（字符设备或块设备）**
- **设备的主设备号**
- **设备的次设备号**

现在不需要在意主设备号与次设备号的区别。

接下来，让我们尝试列出 /dev 下的所有文件。

```
$ ls -l /dev
total 0
crw-rw-rw-1 root tty    5,   0 Dec 18 11:39 tty
...
brw-rw---- 1 root disk   8,   0 Dec 17 09:49 sda
...
```

在 ls -l 的输出中，行首字母为 c 的是字符设备，为 b 的是块设备。第 5 个字段显示的是主设备号，紧接其后的第 6 个字段为次设备号。根据这些信息可以得知，*dev*tty 是字符设备，而 *dev*sda 是块设备。

接下来将逐一介绍这两种设备。

7.10 字符设备

字符设备虽然能执行读写操作，但是无法自行确定读取数据的位置。下面列出了几个比较具有代表性的字符设备。

- 终端
- 键盘
- 鼠标

其中，终端也分为很多种类，很难给出一个准确的定义，但现在只需将其理解为通过 bash 等 shell 程序执行命令的、充满字符的黑白画面或窗口即可。以终端为例，我们可以对其设备文件执行下列操作。

- **write() 系统调用：向终端输出数据**
- **read() 系统调用：从终端输入数据**

下面来实际执行这些操作。首先需要寻找当前进程对应的终端，以及该终端对应的设备文件。查看 ps ax 命令的输出结果中的第 2 个字段，即可得知各个进程关联的终端。

```
$ ps ax | grep bash
 6417 pts/9    Ss     0:00 -bash
 6432 pts/9    S+     0:00 grep bash
$
```

可以看到，bash 关联的终端对应的设备文件名为 *dev*pts/9。接着尝试向该文件写入一个字符串。

```
$ sudo su
#  echo hello >devpts/9
hello
#
```

通过向终端写入字符串 hello（准确来说，是以设备文件为对象请求 write() 系统调用），即可在终端上输出该字符串。执行结果与 echo hello 命令的执行结果一样，这是因为 echo 命令会把 hello 写入标准输出，而 Linux 上的标准输出是指向终端的。

接下来，尝试对系统上的其他终端进行操作。首先在刚才的状态下再启动一个新的终端，然后再次执行 ps ax 命令。

```
$ ps ax | grep bash
 6417 pts/9    Ss+     0:00 -bash
 6648 pts/10   Ss      0:00 -bash
 6663 pts/10   S+      0:00 grep bash
$
```

从执行结果可知，第 2 个终端对应的设备文件名为 *dev*pts/10。接下来，尝试向该文件写入一个字符串。

```
$ sudo su
# echo hello >dev*pts/10
#
```

在执行该命令后查看第 2 个终端，可以发现，这个终端上明明没有写入任何东西，却输出了在前一个终端上写入设备文件的字符串。

```
$ hello
```

现实中其实很少有应用程序会直接操作终端的设备文件，取而代之的是操作 Linux 提供的 shell 程序或者库。应用程序将利用它们提供的更易于使用的接口。通过上面的实验，笔者希望大家至少能明白，平时用惯了的 bash 上的操作都会在底层被转换成对设备文件的操作。

7.11 块设备

块设备除了能执行普通的读写操作以外，还能进行随机访问，比较具有代表性的块设备是 HDD 与 SSD 等外部存储器。只需像读写文件一样读写块设备的数据，即可访问外部存储器中指定的数据。

正如之前提到的那样，通常不会直接访问块设备，而是在设备上创建一个文件系统并将其挂载，然后通过文件系统进行访问，但在以下几种情况下，需要直接操作块设备。

- 更新分区表（利用 **parted** 命令等）
- 块设备级别的数据备份与还原（利用 **dd** 命令等）
- 创建文件系统（利用各文件系统的 **mkfs** 命令等）
- 挂载文件系统（利用 **mount** 命令等）
- **fsck**

下面，我们尝试直接对块设备进行操作。由此，大家将能看见文件系统的“真面目”，而非平常使用的抽象为树状结构的状态。

首先，选择一个合适的分区，在上面创建一个 ext4 文件系统。

```
# mkfs.ext4 devsdc7
mke2fs 1.42.13 (17-May-2015)
Creating filesystem with 244224 4k blocks and 61056 inodes
Filesystem UUID: ele22ad6-a569-47aa-9242-af61b11ee1a3
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376

Allocating group tables: done
Writing inode tables: done
Creating journal (4096 blocks): done
Writing superblocks and filesystem accounting information: done

#
```

然后，挂载创建好的文件系统，并在上面创建一个文件，随意写入一些内容。

```
# mount devsdc7 mnt
# echo "hello world" >mnttestfile
```