```
# ls mnt
lost+found testfile              ←在创建 ext4 文件系统时，必定会创建一个
lost+found 文件
# cat mnttestfile
Hello world
# umount /mnt
```

下面来看一下该文件系统的原生数据。

这里利用 strings 命令，从包含文件系统数据的 devsdc7 中提取出字符串信息。通过 string -t x 命令，可以以一行一个字符串的格式列出文件中的字符串，每行中的第 1 个字段为文件偏移量，第 2 个字段为查找到的字符串。

```
# strings -t x devsdc7
( 略 )
 f35020 lost+found
 f35034 testfile
...
803d000 hello world
10008020 lost+found
10008034 testfile
( 略 )
#
```

通过上面的输出结果可以得知，在 devsdc7 中保存着下列信息。

- **lost+found 目录以及文件名 testfile（元数据）**[6]
- **testfile 文件中的内容，即字符串 hello world（数据）**

[6]不需要深究为什么这两个字符串会出现两次。

接下来，尝试直接在块设备上更改 testfile 的内容。

```
$ echo "HELLO WORLD" >testfile-overwrite
# cat testfile-overwrite
HELLO WORLD
# dd if=testfile-overwrite of=devsdc7 seek=$((0x803d000)) ↵
bs=1                             ←把 HELLO WORLD 覆写到 testfile 中

# strings -t x devsdc7
( 略)
803d000 HELLO WORLD              ←直到刚才都还是 hello world
```

```
( 略)
#
```

testfile 中的内容被成功地替换了。通过这一系列的实验，大家应该能明白，通过直接操作块设备的设备文件，即可操作外部存储器，并且隐藏在文件系统下的只是保存在外部存储器中的数据而已。

在本次实验中，为了向大家揭示文件系统与块设备之间的关系，我们直接通过块设备更改了文件系统的内容，但是在这样做时需要谨记以下几点。

- 实验中那样的替换文件内容的方法，只是恰好能在当前版本的 ext4 文件系统上使用而已，不能保证适用于其他种类的文件系统以及以后版本的 ext4
- 随便地通过块设备更改文件系统的内容是很危险的，因此只应在测试用的文件系统上执行这种操作
- 不要在文件系统已挂载的状态下访问保存着该文件系统的块设备，否则将有可能破坏文件系统的一致性，并且损坏其中的数据

7.12 各种各样的文件系统

到现在为止，我们已经介绍了 ext4、XFS 和 Btrfs 这 3 种文件系统，这些都是存在于外部存储器中的文件系统。除此以外，Linux 上还存在各种各样的文件系统，接下来将对其中几个进行介绍。

7.13 基于内存的文件系统

tmpfs 是一种创建于内存（而非外部存储器）上的文件系统。虽然这个文件系统中保存的数据会在切断电源后消失，但由于访问该文件系统时不再需要访问外部存储器，所以能提高访问速度，如图 7-20 所示。

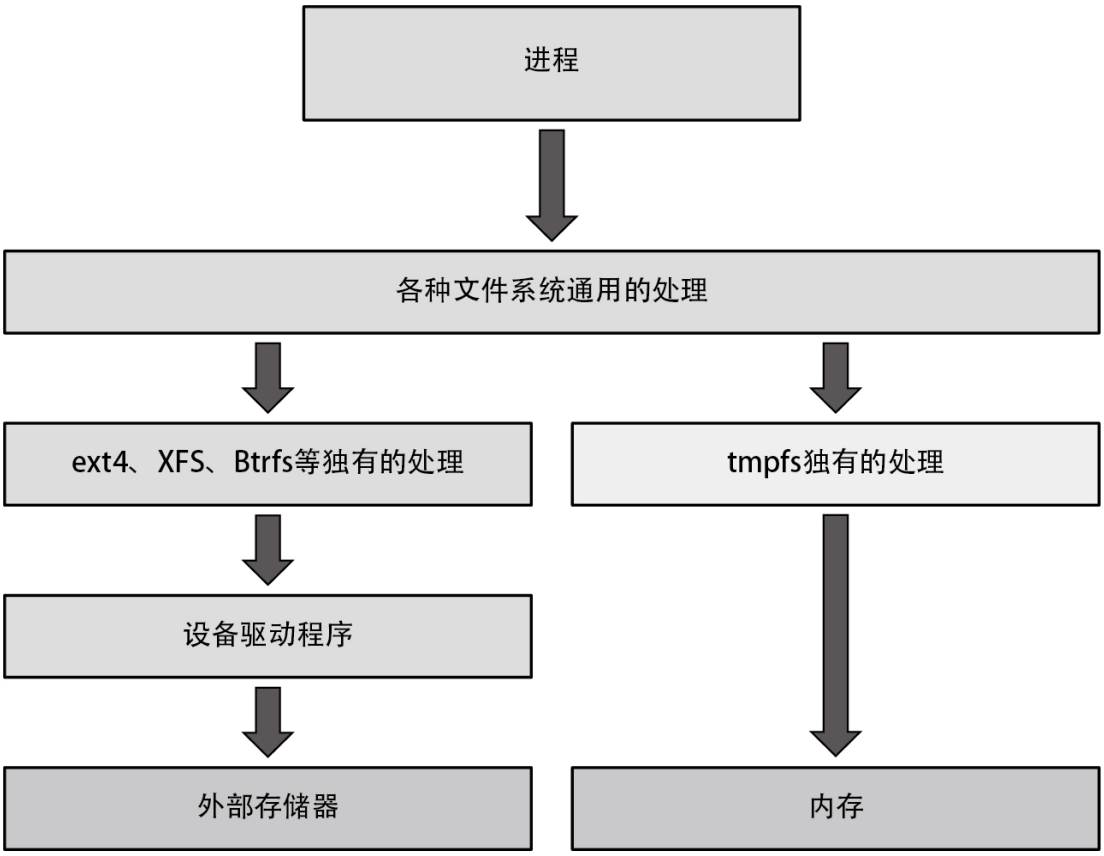

图 7-20 tmpfs

tmpfs 通常被用于 /tmp 与 /var/run 这种“文件内容无须保存到下一次启动时”的文件上。在笔者的计算机上也以各种用途使用了 tmpfs，如下所示。

```
$ mount | grep ^tmpfs
tmpfs on /run type tmpfs (rw,nosuid,noexec,relatime,size ↵
=3294200k,mode=755)
tmpfs on devshm type tmpfs (rw,nosudid,nodev)
tmpfs on runlock type tmpfs (rw,nosudio,nodev,noexec, ↵
```

```
relatime,size=5120k)
tmpfs on sysfs/cgroup type tmpfs (rw,mode=755)
tmpfs on runuser/108 type tmpfs (rw,nosuid,nodev,relatime, ↲
size=3294200k,mode=700,uid=108,gid=114)
tmpfs on runuser/1000 type tmpfs (rw,nosuid,nodev,relatime, ↲
size=3294200k,mode=700,uid=1000,gid=1000)
$
```

tmpfs 创建于挂载的时候。在挂载时，通过 size 选项指定最大容量。不过，并不是说从一开始就直接占用指定的内存量，而是在初次访问文件系统中的区域时，以页为单位申请相应大小的内存。通过查看 free 命令的输出结果中 shared 字段的值，可以得知 tmpfs 实际占用的内存量。

```
$ free
              total        used        free      shared  buff/cache   available
Mem:       32942000      390620    28889232      108552     3662148    31818884
Swap:             0           0           0
$
```

在笔者的计算机中，tmpfs 占用的内存量为 108552KB，即 106 MB 左右。

7.14　网络文件系统

到现在为止介绍的文件系统都是用于查看本地计算机上的数据的，而**网络文件系统**则可以通过网络访问远程主机上的文件，如图 7-21 所示。

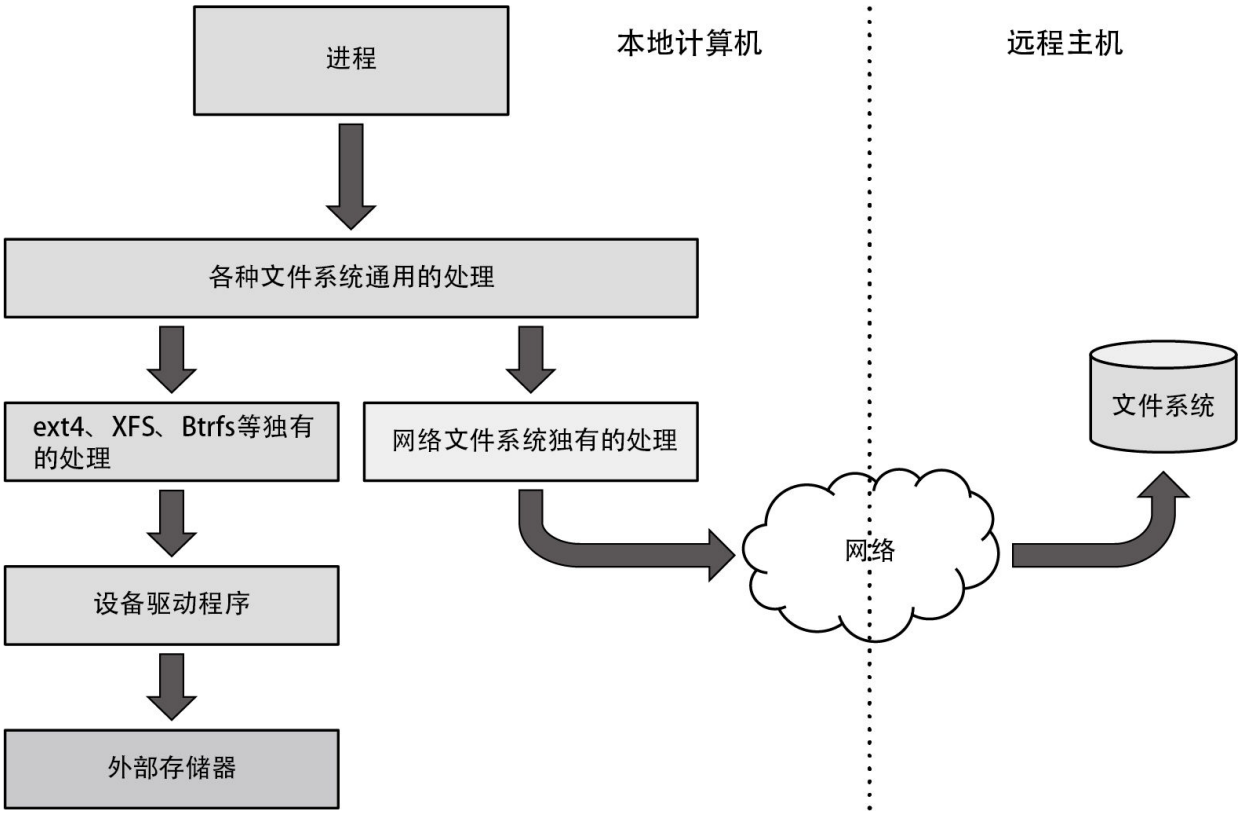

图 7-21　网络文件系统

虽然网络文件系统同样存在许多种类，但基本上，在访问 Windows 主机上的文件时，使用名为 cifs 的文件系统；而在访问搭载 Linux 等类 UNIX 系统的主机上的文件时，则使用名为 nfs 的文件系统。

7.15 虚拟文件系统

系统上存在着各种各样的文件系统，用于获取内核中的各种信息，以及更改内核的行为。有几种文件系统已经在本书中出现过，这里再次介绍一下。

● procfs

procfs 用于获取系统上所有进程的信息。它通常被挂载在 /proc 目录下。通过访问 *proc*pid/ 目录下的文件，即可获取各个进程的信息。在笔者的计算机上，bash 相关的信息如下所示。

```
$ ls proc$$
attr       cgroup      comm       cwd      fd         io  map_files
mountinfo     net   oom_adj   pagemap    root  sessionid      stack
status     timers     wchan   autogroup  clear_refs coredump_filter
environ    fdinfo    limits      maps  mounts         ns  oom_score
personality   sched  setgroups   stat  syscall  timerslack_ns  auxv
cmdline    cpuset    exe gid_map   loginuid       mem    mountstats
numa_maps  oom_score_adj  projid_map  schedstat    smaps      statm
task     uid_map
$
```

这里出现了大量的文件，我们看一下其中一部分。

- ***proc*pid/maps**：进程的内存映射
- ***proc*pid/cmdline**：进程的命令行参数
- ***proc*pid/stat**：进程的状态，比如 CPU 时间、优先级和内存使用量等信息

除了进程的信息之外，使用 procfs 还能得到以下几点信息。

- ***proc*cpuinfo**：搭载于系统上的 CPU 的相关信息
- ***proc*diskstat**：搭载于系统上的外部存储器的相关信息
- ***proc*meminfo**：搭载于系统上的内存的相关信息
- ***proc*sys** 目录下的文件：内核的各种调优参数，与 **sysctl** 命令和 ***etc*sysctl.conf** 中的调优参数一一对应

到此为止使用过的 ps、sar、top 和 free 等用于显示 OS 提供的各种信息的命令，都是从 procfs 中采集信息的。

如需了解各个文件的详细含义，请参考 man proc 的内容。

● sysfs

在 Linux 引入 procfs 后不久，越来越多的乱七八糟的信息也被塞入其中，导致出现了许多与进程无关的信息。为了防止 procfs 被滥用，Linux 又引入了一个名为 sysfs 的文件系统，用来存放那些信息。sysfs 通常被挂载在 /sys 目录下。

sysfs 包括但不限于下列文件。

- ***sys*devices 目录下的文件：搭载于系统上的设备的相关信息**
- ***sysfs* 目录下的文件：系统上的各种文件系统的相关信息**

● cgroupfs

cgroup 用于限制单个进程或者由多个进程组成的群组的资源使用量，它需要通过文件系统 cgroupfs 来操作。另外，只有 root 用户可以操作 cgroup。cgroupfs 通常被挂载在 *sys*fs/cgroup 目录下。

通过 cgroup 添加限制的资源有很多种，举例如下。

- **CPU：设置能够使用的比例，比如令某群组只能够使用 CPU 资源总量的 50% 等。通过读写 *sysfs/cgroup/cpu* 目录下的文件进行控制**
- **内存：限制群组的物理内存使用量，比如令某群组最多只能够使用 1 GB 内存等。通过读写 *sysfs/cgroup/memory* 目录下的文件进行控制**

cgroup 通常被用于通过 Docker 之类的容器管理软件，或者 virt-manager 等虚拟机管理软件来限制各个容器或虚拟机的资源使用量。特别是多租户技术架构的系统上经常使用 cgroup，在这些系统上往往有多个客户的容器与虚拟机同时存在。

7.16 Btrfs

ext4 与 XFS 虽然存在些许差别，但两者都是从 UNIX（Linux 的根基）诞生之时就存在的文件系统，且两者都仅提供了创建、删除和读写文件等基本功能。近年来出现了许多功能更加丰富的新文件系统，其中比较具有代表性的就是 Btrfs。下面将介绍一下 Btrfs 提供的部分功能。

● 多物理卷

在 ext4 与 XFS 上需要为每个分区创建文件系统，但在 Btrfs 中不需要这样。Btrfs 可以创建一个包含多个外部存储器或分区的存储池，然后在存储池上创建可被挂载的区域，该区域称为**子卷**。存储池相当于 LVM 中的卷组，而子卷则类似于 LVM 中的逻辑卷与文件系统的融合。因此，为了便于理解 Btrfs 的机制，与其把它当成传统意义上的文件系统，不如把它当作文件系统与 LVM 等逻辑卷管理器融合后的产物，如图 7-22 所示。

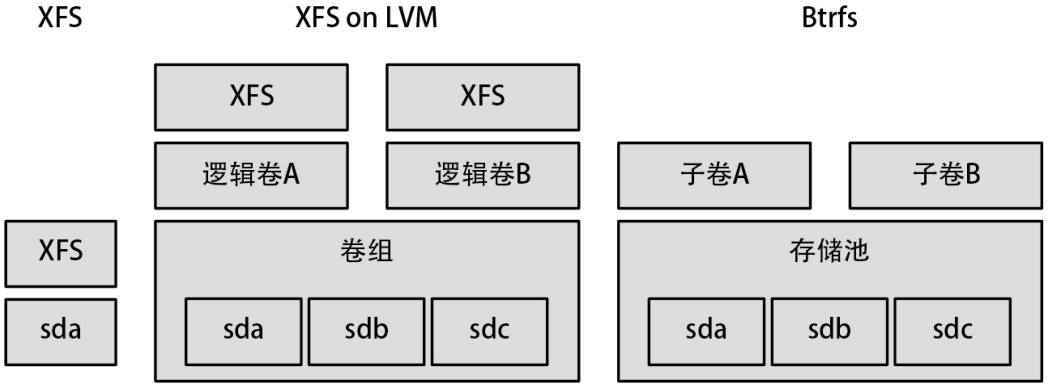

图 7-22 Btrfs 同时具备传统文件系统与 LVM 的功能

在 Btrfs 中，甚至还可以向现存的文件系统添加、删除以及替换外部存储器。即便这些操作导致容量发生变化，也无须调整文件系统大小，如图 7-23 所示。另外，这些操作都能在文件系统已挂载的状态下进行，无须卸载文件系统。

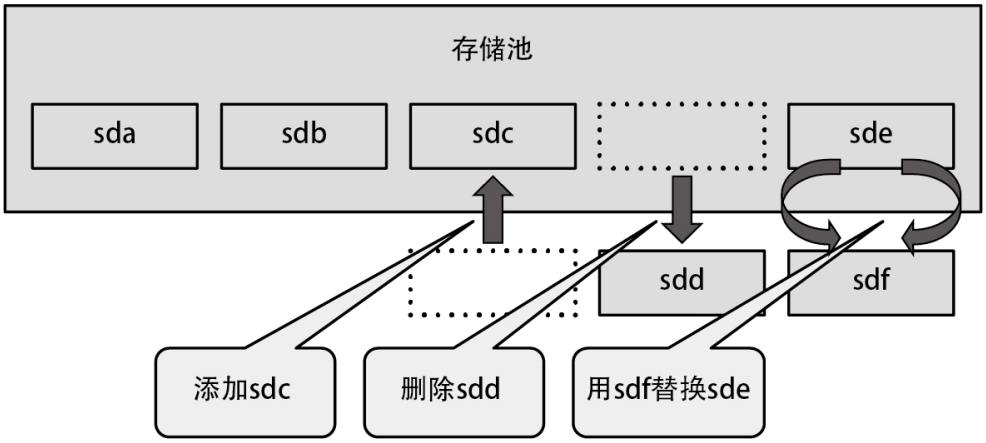

图 7-23　在 Btrfs 文件系统上可以添加、删除以及替换外部存储器

● 快照

Btrfs 可以以子卷为单位创建快照。创建快照并不需要复制所有的数据，只需根据数据创建其元数据，并回写快照内的脏页即可，因此创建快照与平常的复制操作比起来要快得多。而且，由于快照与子卷共享数据，所以创建快照的存储空间成本也很低。

下面就来看一下通常的复制操作与在 Btrfs 中创建子卷快照这两种备份方式在存储空间成本上的差距。首先是复制操作的情形，如图 7-24 所示。

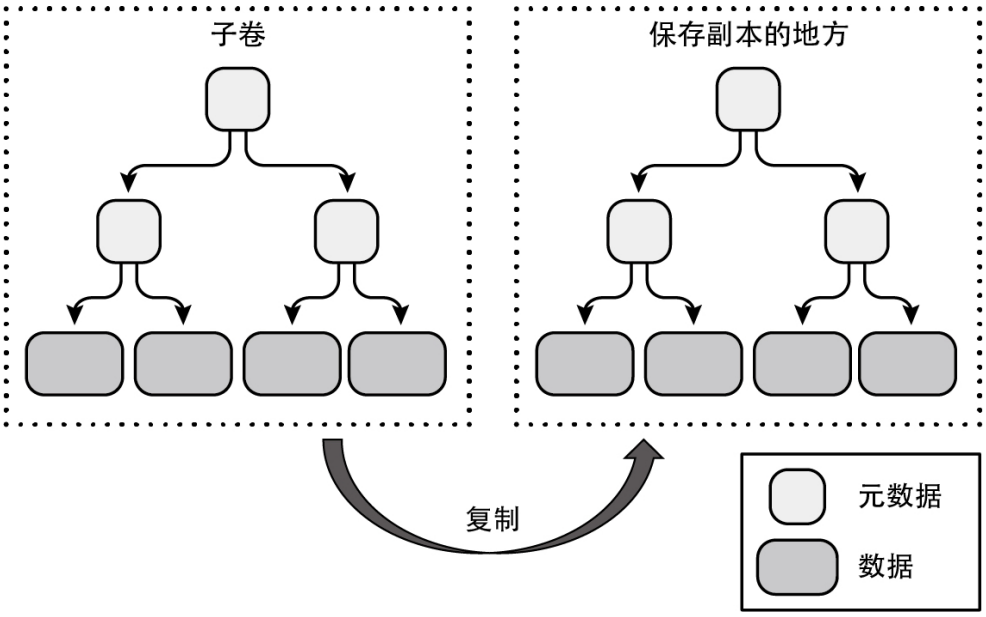

图 7-24　通常的复制操作

可以看到，不但需要创建新的元数据，还需要为所有数据创建一份副本。接着，来看一下在 Btrfs 上是如何创建快照的，如图 7-25 所示。

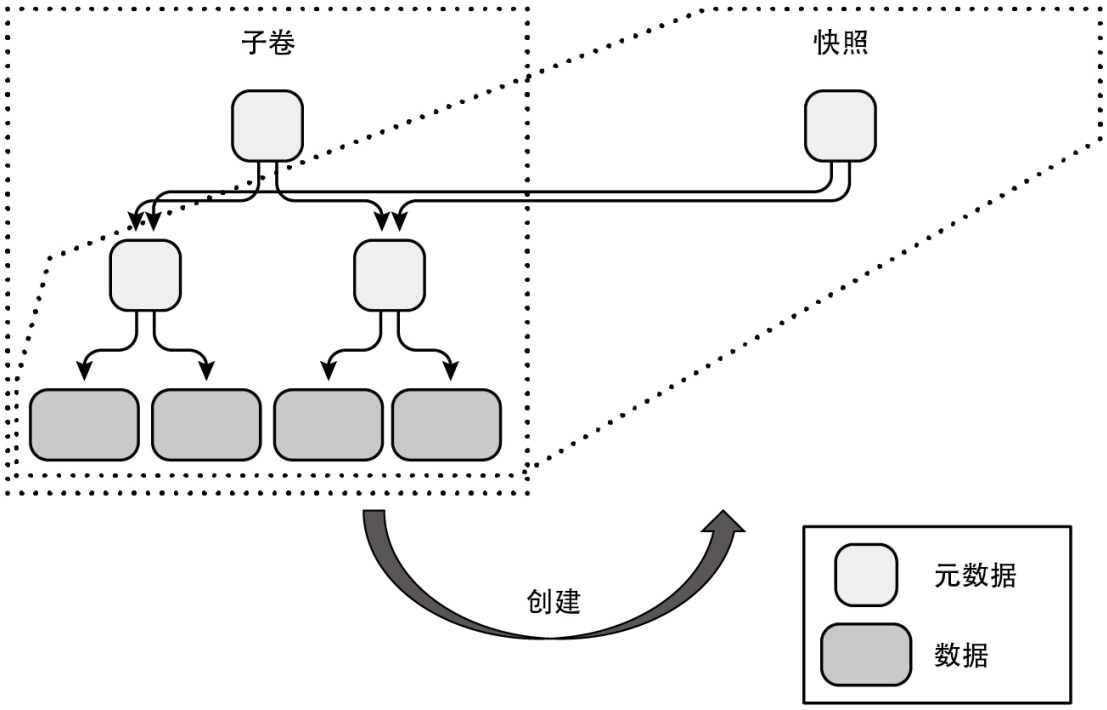

图 7-25　快照

只需创建一个新的根目录节点，然后为其创建指向下一层的节点的链接即可。也就是说，无须复制任何数据，因此比起复制操作要快得多。

● RAID

Btrfs 可以在文件系统级别上配置 RAID，支持的 RAID 级别有 RAID 0、RAID 1、RAID 10、RAID 5、RAID 6 以及 dup（在同一个外部存储器中对一份数据创建两份副本，适用于单设备）。另外，在配置 RAID 时，不是以子卷为单位，而是以整个 Btrfs 文件系统为单位。

首先来看一下没有配置 RAID 时的结构。假设在一个仅由 sda 构成的单设备 Btrfs 文件系统上存在一个子卷 A。在这种情况下，一旦 sda 出现故障，子卷 A 上所有的数据就会丢失，如图 7-26 所示。

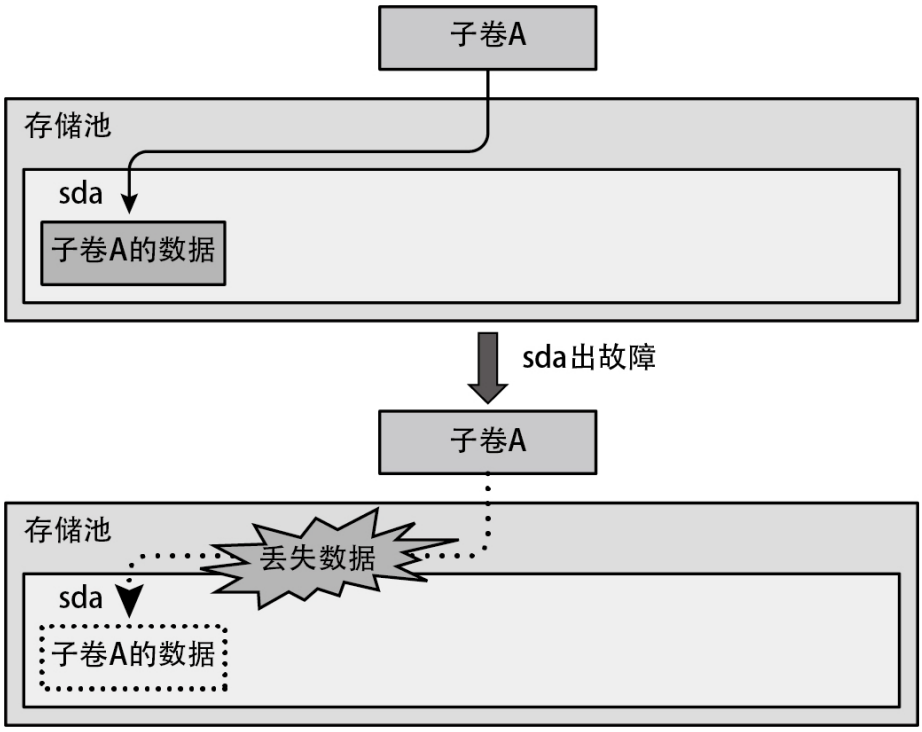

图 7-26　如果没有配置 RAID，则一旦 sda 出现故障，所有数据就会丢失

与此相对，如果为文件系统配置了 RAID 1，那么所有数据都会被写入两台外部存储器（在本例中为 sda 与 sdb），所以即便 sda 出现故障，在 sdb 上也还保留着一份子卷 A 的数据，如图 7-27 所示。

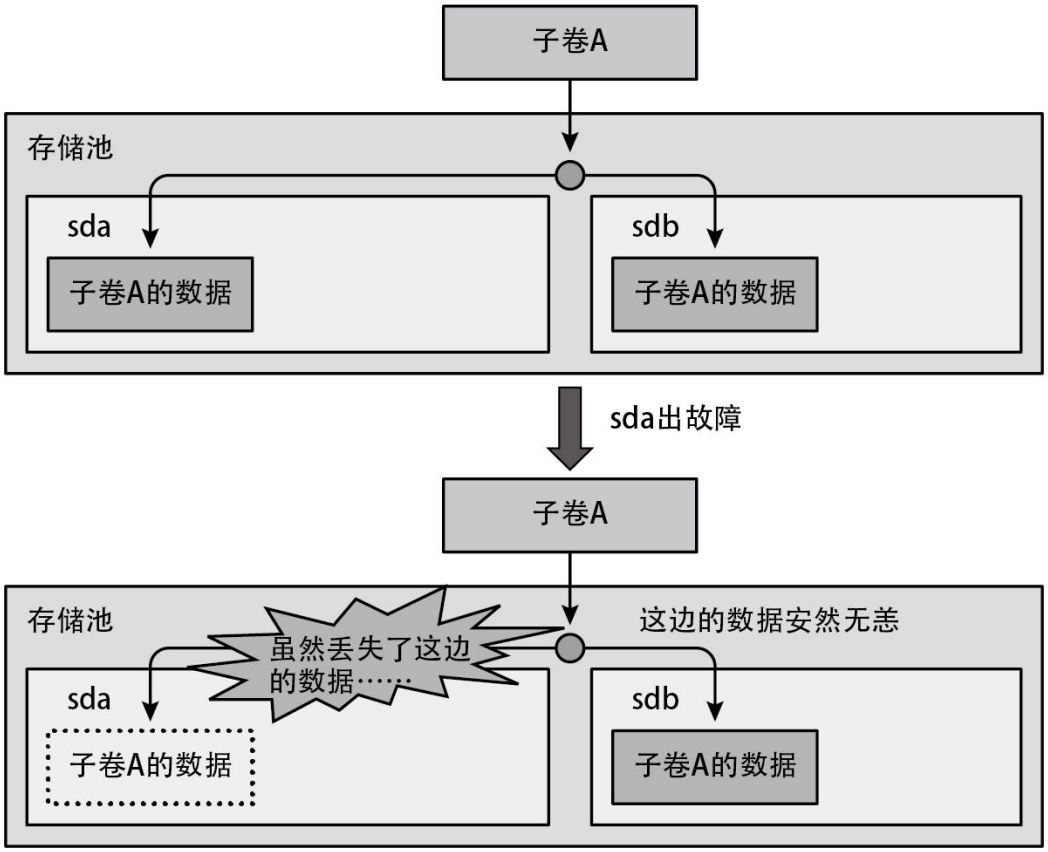

图 7-27　如果配置了 RAID 1，那么即便 sda 出现故障，也不会导致数据丢失

● 数据损坏的检测与恢复

当外部存储器中的部分数据损坏时，Btrfs 能够检测出来，如果配置了某些级别的 RAID，还能修复这些损坏的数据。而在那些没有这类功能的文件系统中，即便在写入时外部存储器中的数据因比特差错等而损坏了，也无法检测出这些损坏的数据，而是在数据损坏的状态下继续运行，如图 7-28 所示。

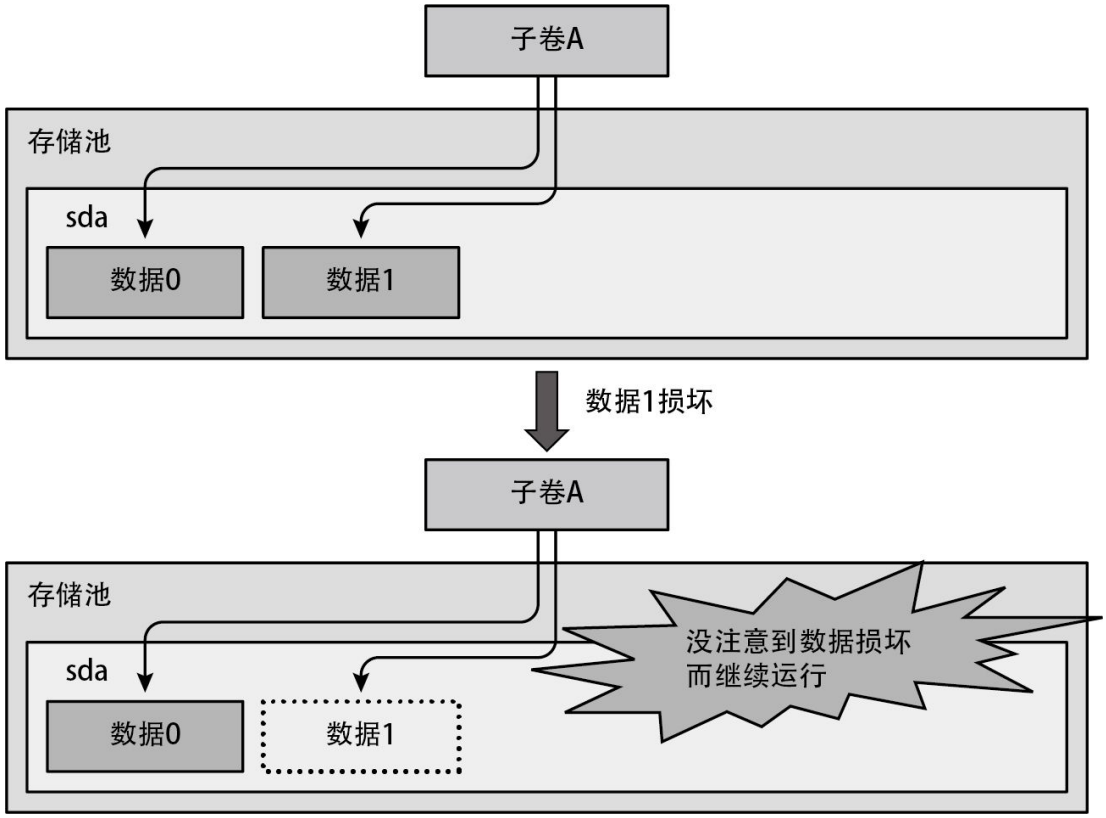

图 7-28　检测不到数据损坏，有可能在损坏的状态下继续运行

这有可能导致更多的数据被损坏，而且我们很难查明这种故障的起因。

与此相对，Btrfs 拥有一种被称为**校验和**（checksum）的机制，用于检测数据与元数据的完整性。通过这种机制，可以检测出数据损坏。如图 7-29 所示，如果在读取数据或者元数据时出现校验和报错，将丢弃这部分数据，并通知发出读取请求的进程出现了 I/O 异常。

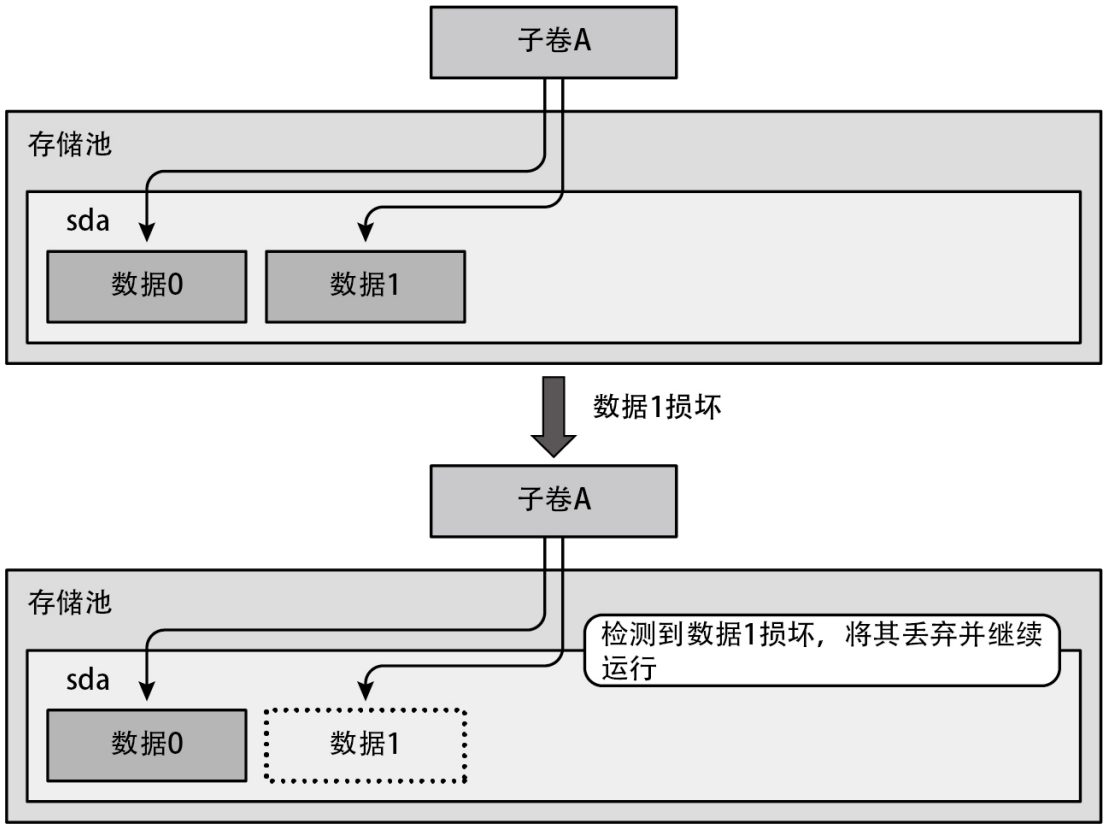

图 7-29　通过校验和检测数据损坏

在检测到损坏时，如果配置的 RIAD 级别是 RAID 1、RAID 10、RAID 5、RAID 6 或者 dup 之一，Btrfs 就能基于校验和正确的另一份数据副本修复破损的数据。在 RAID 5 和 RAID 6 中，还能通过**奇偶校验**（parity check）实现同样的功能。图 7-30 所示为 RAID 1 的数据修复流程。