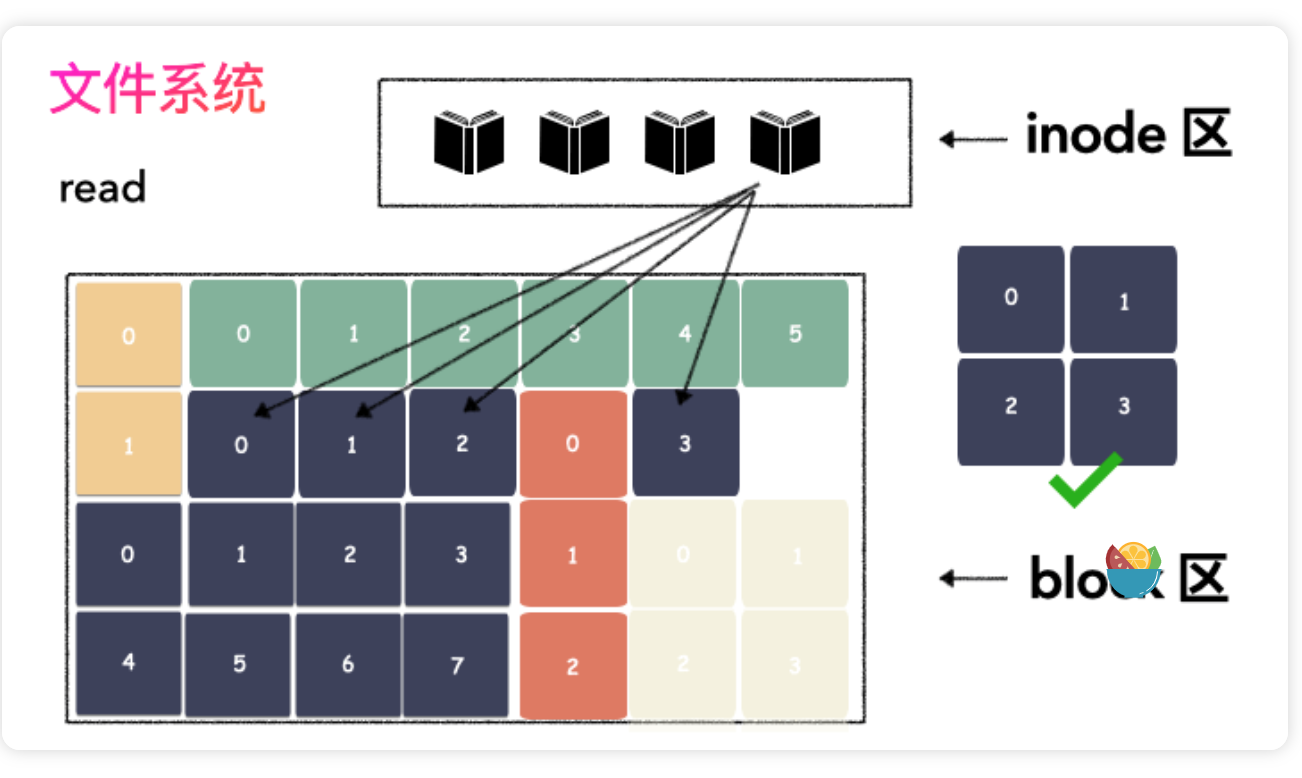
**inode/block 概念**

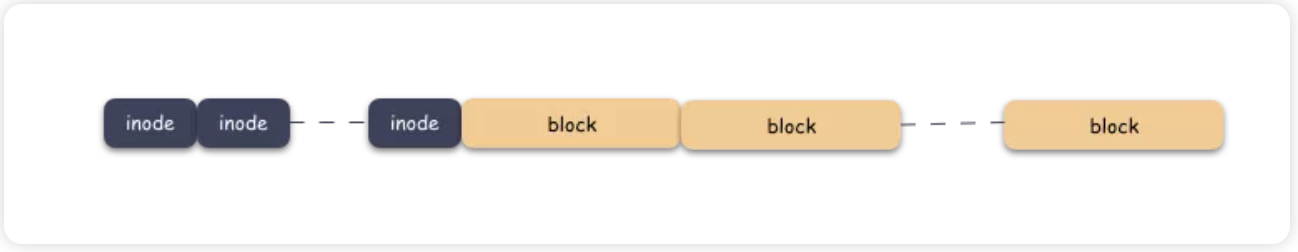
1. 磁盘空间是按照 Block 粒度来划分空间的，存储数据的区域全都是 Block，我们叫做数据区域；
2. 文件存储不再连续存储在磁盘上，所以需要记录元数据，这个我们叫做 inode；



文件系统中，一个 inode 唯一对应一个文件，inode 的个数则是在文件系统格式化的时候就确定好了的，换言之，**一个 local 文件系统支持的文件数是天然就有上限的**。

block 固定大小，每个 4k（大部分文件系统都是，这里不做纠结），block 意图存储打散的用户数据。

无论是 inode 区，还是 block 区，本质上都是在线性的磁盘空间上。文件系统的空间层次如下：



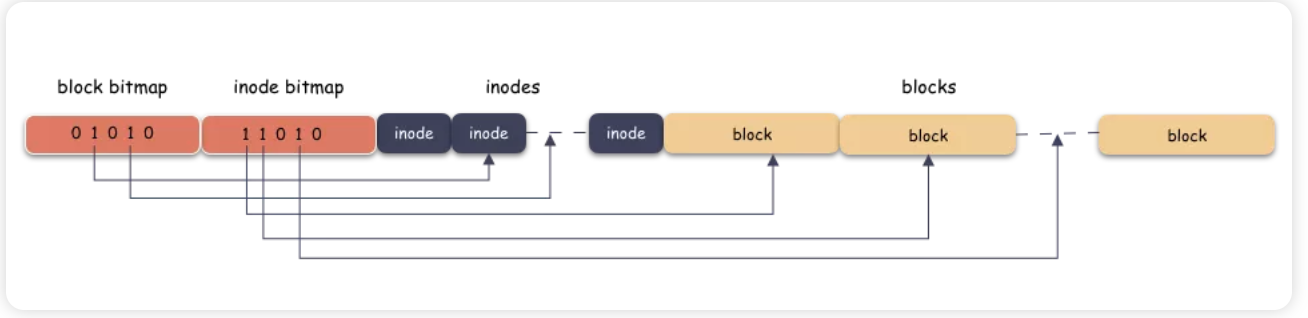
一个文件的对应一个 inode，这个文件需要按照 Block 切分存储在磁盘上，存储的位置则由 inode 记录起来，通过 inode 则能找到 block，也就获取到用户数据。

现在有一个新的小问题，inode 区和 block 区都是在初始化就构造好的。存储一个文件的时候，需要取一个空闲的 inode，然后把数据切分成 4k 大小存储到空闲的 block 上，对吧？

**划重点：空闲的inode，空闲的 block。** 这个很关键，已经存储了数据的地方不能再让写，不然会把别人的数据覆盖掉。

那么，怎么区分空闲和已经在用的 inode ，block 呢？

**答案是 ：inode 区和 block 区分别需要另一张表，用来表示 inode 是否在用，block 是否在用，这个表的名字我们叫做 bitmap 表**。bitmap 是一个 bit 数组，用 0 表示空闲，1 表示在用，如下：



bitmap 什么时候用呢？自然是写的时候，也就是分配 inode 或者 block 的时候，因为只有分配的时候，你才需要找空闲的空间。

**小结一下**：

1. bitmap 本质是个 bit 数组，占用空间极其少，用 0 来表示空闲，1 表示在用。使用时机是在创建文件，或者写数据的时候；
2. inode 则对应一个文件，里面存储的是元数据，主要是数据 block 的位置信息；
3. block 里面存储的是用户数据，用户数据按照 block 大小（4k）切分，离散的分布在磁盘上。读的时候只有依赖于 inode 里面记录的位置才能恢复出完整的文件；
4. inode 和 block 的总个数在文件系统格式化的时候就确定了，所以文件数和文件大小都是有上限的；

**一个文件真实的模样**

上面是抽象的样子，现在我们看一个真实的 inode -> block 的样子。一个文件除了数据需要存储之外，一些元信心也需要存储，例如文件类型，权限，文件大小，创建/修改/访问时间等，这些信息存在 inode 中，每个文件唯一对应一个inode 。

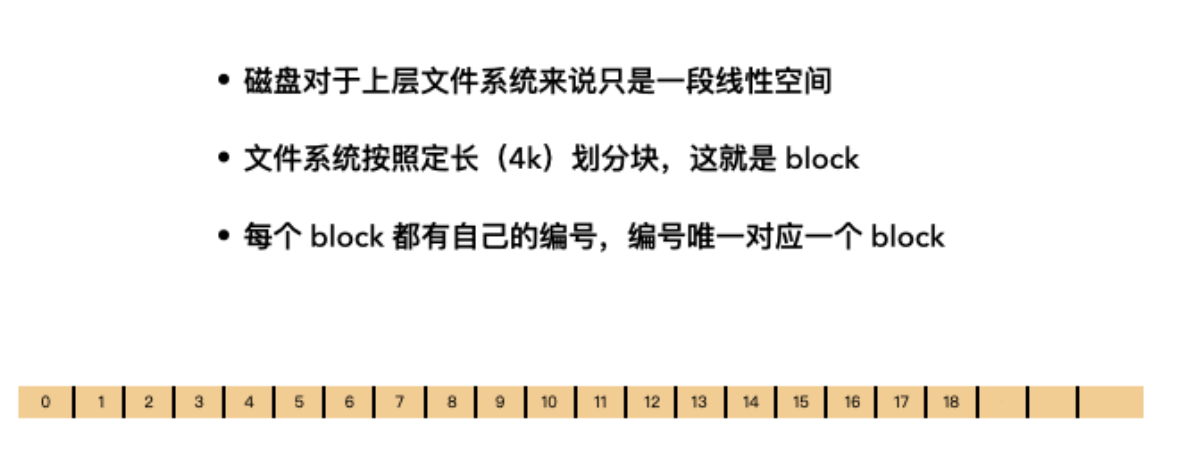
看一下 inode 的数据结构（就以 linxu ext2 为例，该结构定义在 linux/fs/ext2/ext2.h 头文件中 ）：

struct ext2\_inode {  
    \_\_le16  i\_mode;     /\* File mode \*/  
    \_\_le16  i\_uid;      /\* Low 16 bits of Owner Uid \*/  
    \_\_le32  i\_size;     /\* Size in bytes \*/  
    \_\_le32  i\_atime;    /\* Access time \*/  
    \_\_le32  i\_ctime;    /\* Creation time \*/  
    \_\_le32  i\_mtime;    /\* Modification time \*/  
    \_\_le32  i\_dtime;    /\* Deletion Time \*/  
    \_\_le16  i\_gid;      /\* Low 16 bits of Group Id \*/  
    \_\_le16  i\_links\_count;  /\* Links count \*/  
    \_\_le32  i\_blocks;   /\* Blocks count \*/  
    \_\_le32  i\_flags;    /\* File flags \*/  
  
    \_\_le32  i\_block[EXT2\_N\_BLOCKS];/\* Pointers to blocks \*/  
    \_\_le32  i\_file\_acl; /\* File ACL \*/  
    \_\_le32  i\_dir\_acl;  /\* Directory ACL \*/  
    \_\_le32  i\_faddr;    /\* Fragment address \*/  
};

**重点**：

* 上面的结构 mode，uid，size，time 等信息就是我们常说的文件类型，大小，创建修改等时间元数据；
* 注意到 i\_block[EXT2\_N\_BLOCKS]  这个字段，这个字段将会带你找到数据, 因为里面存储的就是 block 所在的位置，也就是 block 的编号；

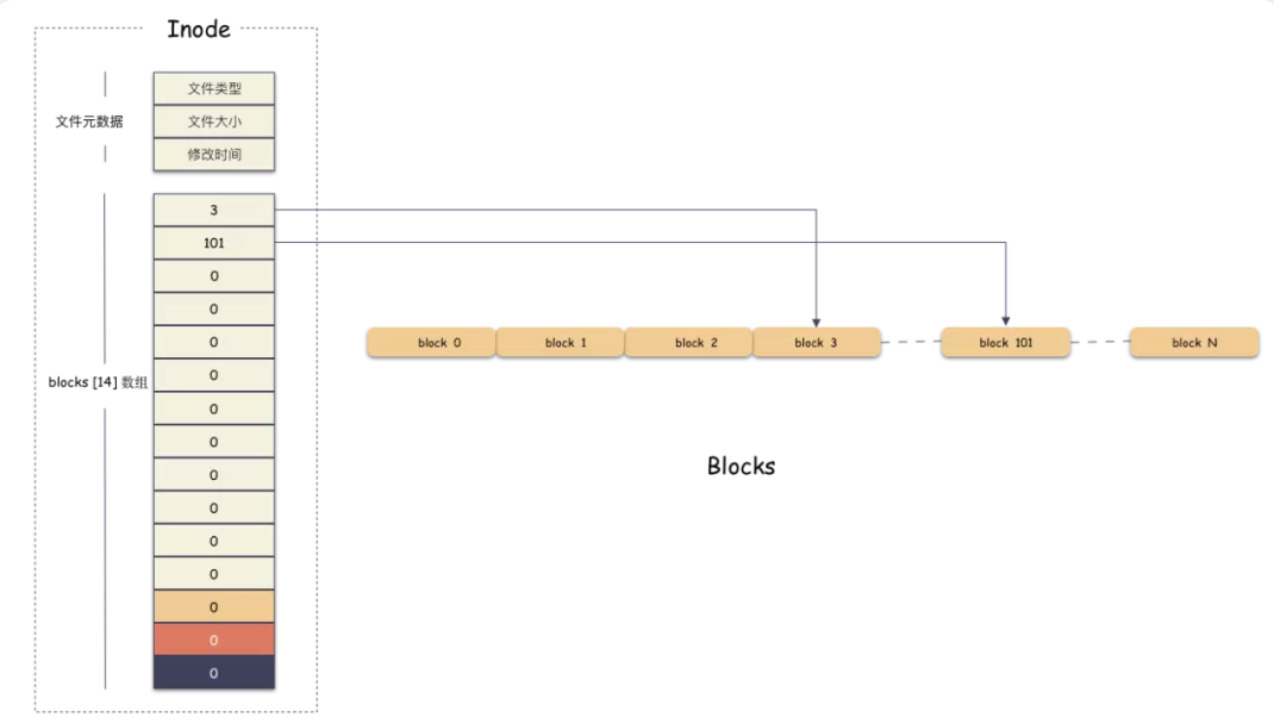
再来，理解下什么叫做 block 的位置（编号）。



**位置就是编号，记录位置就是记录编号，编号就是索引。**

我们看到有一个数组：i\_block[EXT2\_N\_BLOCKS]，这个数组是存储 block 位置的数组。其中 EXT2\_N\_BLOCKS 是一个宏定义，值为 15 。也就是说，i\_block 是一个 15 个元素的数组，每个元素是 4 字节（32 bit）大小。

举个例子，假设我们现在有一个 6k 的文件，那么只需要 2 个 block 就可以存下了，假设现在数据就存储在编号为 3 和 101  这两个 block 上，那么如下图：



i\_block[15] 第一个元素存的是 3，第二个存储的是 101，其他槽位没用用到，**由于 inode 的内存是置零分配的，所以里面的值为 0，表示没有在使用** . 我们通过 [3, 101] 这两个 block 就能拼装出完整的用户数据了。用户的 6k 文件组成如下：

1. 第一个 4k 数据在 [3\*4K, 4\*4K] 范围；
2. 第二个 2k 数据在 [ 101\*4K, 101\*4K+2K] 范围；

好，现在我们知道了**每个定长 block 都有唯一编号**，我们的 i\_block[15]**数组 通过有序存储这个编号找到文件数据所在的位置**，并且拼装出完整文件。

**思考问题：区分文件的切分成 4k 块的编号和 磁盘上物理 4k 块的编号的区别。**

举个栗子，一个文件 12K 的大小，那么按照 4K 切分会存储到 3 个 物理 block 上。

文件第 0 个 4k 存储到了 101 这个物理 block 上；文件第 1 个 4k 存储到了 30  这个物理 block 上；文件第 2 个 4k 存储到了 11  这个物理 block 上；

文件逻辑空间上的编号是从 0 开始，到 2 结束，对应存储的物理块编号分别是 101，30，11 。

**思考问题：这么一个 inode 结构能够表示多大的文件？**

我们看到 inode->i\_block[15] 是一个一维数组，里面能存 15 个元素。也就是能存 15 个 block 的编号，那么如果直接存储文件的 block 编号最大能表示 60K (15\*4K) 的文件。换句话说，如果我拿着 15 个槽位全部用来存储文件的编号，这个文件系统支撑的最大文件却就是 60K。惊呆了？（注意：ext2 文件系统是可以创建 4T 以内的文件的！！）

那我们自然会思考，怎么解决呢？怎么才能支撑更大的文件？

最直接思考就是用更大的数组，把 inode->i\_block  数组变得更大。比如，如果你想要支持 100G 的文件：

那么，需要 i\_block 数组大小为 26214400 （计算公式：100\\*1024\\*1024/4），也就是要分配一个 i\_block[26214400]  的数组。

每个编号占用 4 字节，这个数组就占用 100M 的空间（计算公式：(26214400\\*4)/1024/1024）。100M ！这里就有点夸张了，注意到 i\_block 只是一个 inode 内部的字段，是一个静态分配的数组，也就是说，**这个文件系统为了支持最大 100G 的文件存入，每一个 inode 都要占用 100M 的内存，就算你是一个 1K 的文件，inode 也会占用这么大的内存空间**。并且，这种方案扩展性差，支持的文件 size 越大，i\_block[N] 消耗内存情况越严重。这是无法接受的。

**思考问题：怎么才能让你既能表示更大的文件，又能不浪费占用空间？**

我们仔细分析这个问题，你会发现，这里有 2 个核心问题：

1. 第一点，核心在于浪费内存空间（关键点是要保证 inode 内存结构的稳定，无论文件怎么变，inode 结构本身不能变）；
2. 第二点，仔细思考你会发现，无论是什么神仙方案，如果你要存储一个按照 4k 切分的 100G 文件，都是需要 100M 的空间来存储索引（ block 编号），但是 99.99% 的文件可能都没有这么大；

我们前面用一个大数组来一把存储 block 编号的方案固然简单，但是问题在于太过死板。核心问题在于存储 block 编号的数组是预分配的，为了还没有发生并且 99% 场景都不会发生的事情（文件大小达到 100G），却不管三七二十一，提前准备好了完整的 block 索引数组，**预分配就是浪费的根源**。

**问题一的解决：索引存磁盘**：

既然问题在于浪费内存，inode 内存分配不灵活，那就可以看把 inode->i\_block 下放到磁盘。

为什么？

因为磁盘的空间比内存大了不止一个量级。100M 对内存来说**很大**，对磁盘来说**很小**。换句话说，用把用户数据所在的 block 编号存到磁盘上去，这个也需要物理空间，使用的也是 block 来存储，只不过这种 block 存储的是 block 编号信息，而不是用户数据。

那么我们怎么通过 inode 找到用户数据呢？

因为这个 block 本身也有编号，我们则需要把这个存储用户 block 编号的 block 所在块的编号存储在 inode->i\_block[15] 里，当读数据的时候，我们需要先找到这个存储编号的 block，然后再通过里面存储的用户数据所在的 block 编号找到用户所在的 block ，去读数据。

这个**存储用户 block 编号的 block 所在块的编号**我们叫做**间接索引**，然后我们根据跳转的次数可以分类成一级索引，二级索引，三级索引。顾名思义，一级索引就是跳转 1 次就能定位到用户数据，二级索引就是跳转 2 次，三级索引就是跳转 3 次才能定位到用户数据。那么 inode->i\_block[15]  里面存储的可以直接定位到用户数据的 block 就是**直接索引**。

终于可以说回 ext2 的使用了，ext2 的 inode->i\_block[15] 数组。知识点来了，按照约定，这 15 个槽位分作 4 个不同类别来用：

1. 前 12 个槽位（也就是 0 - 11 ）我们成为**直接索引**；
2. 第 13 个位置，我们称为 **1 级索引**；
3. 第 14 个位置，我们称为 **2 级索引**；
4. 第 15 个位置，我们称为 **3 级索引**；



**直接索引**：能存 12 个 block 编号，每个 block 4K，就是 48K，也就是说，48K 以内的文件，只需要用到 inode->i\_block[15]  前 12 个槽位存储编号就能完全 hold 住。

**一级索引**：

inode->i\_block[12] 这个位置存储的是一个一级索引，也就是说这里存储的编号指向的 block 里面存储的也是 block 编号，里面的编号指向用户数据。一个 block 4K，每个元素 4 字节，也就是有 1024 个编号位置可以存储。

所以，一级索引能寻址 4M（1024 \* 4K）空间 。

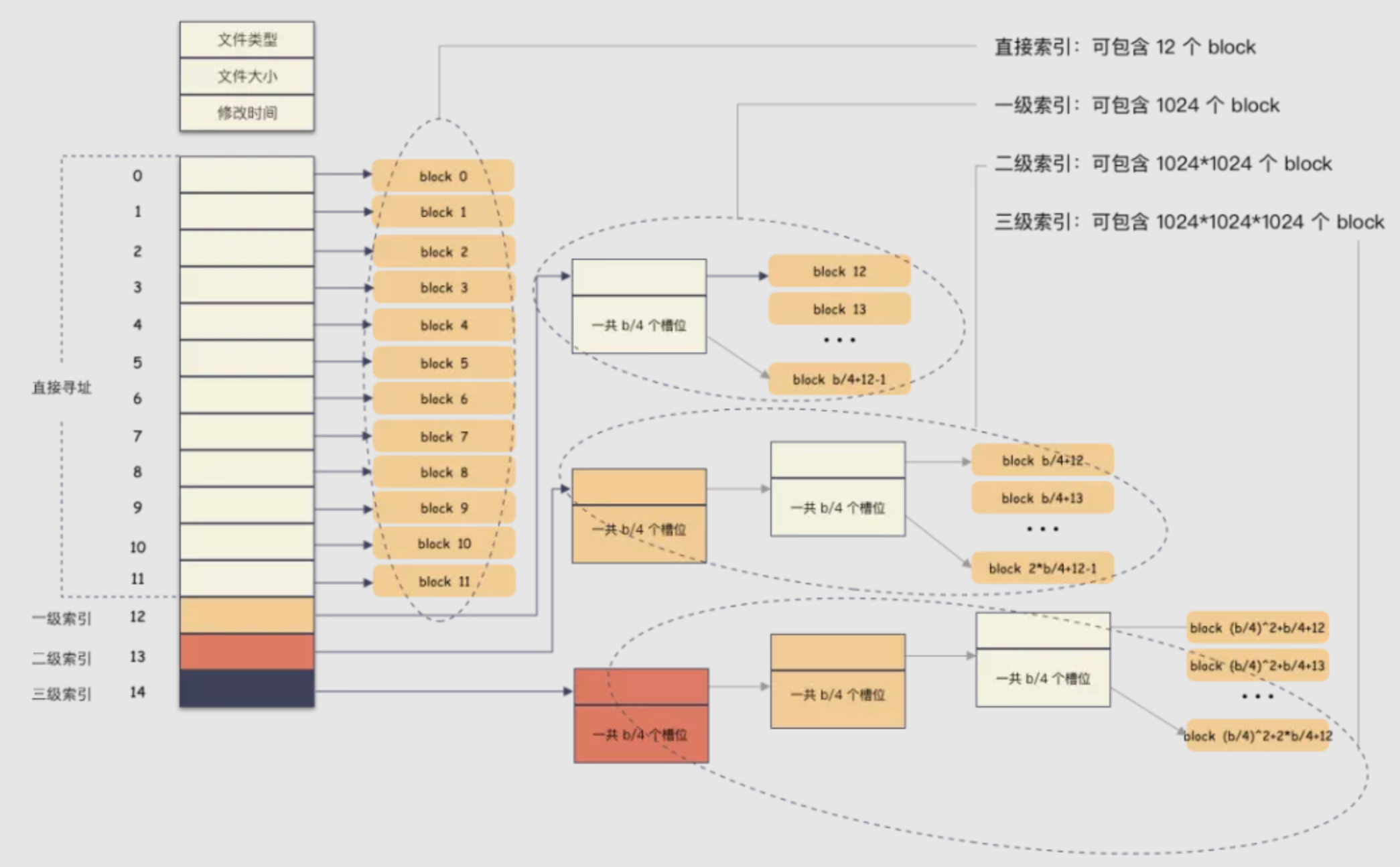
**二级索引**：

二级索引是在一级索引的基础上多了一级而已，换算下来，有了 4M 的空间用来存储用户数据的编号。所以二级索引能寻址 4G (4M/4 \* 4K) 的空间。

**三级索引**：

三级索引是在二级索引的基础上又多了一级，也就是说，有了 4G 的空间来存储用户数据的 block 编号。所以二级索引能寻址 4T （4G/4 \* 4K） 的空间。

**最后，看一眼完整的表示图：**



所以，在我们 ext2 的文件系统上，通过这种间接块索引的方式，最大能支撑的文件大小 = 48K + 4M + 4G + 4T ，约等于 4 T。文件系统最大支撑 16T 空间，因为 4 Byte 的整形最大数就是 2^32=4294967296 ， 乘以 4K 就等于 16 T。

ext2 文件系统支持的最大单文件大小和文件系统最大容量就是这么算出来的（温馨提示：ext4 文件系统不仅兼容间接块的实现，还使用的是 extent 模式来管理的空间，最大支持单文件 16 TB ，文件系统最大 1 EB）。

**思考：这种多级索引寻址性能表现怎么样？**

在不超过 12 个数据块的小文件的寻址是最快的，访问文件中的任意数据理论只需要两次读盘，一次读 inode，一次读数据块。访问大文件中的数据则需要最多五次读盘操作：inode、一级间接寻址块、二级间接寻址块、三级间接寻址块、数据块。

**一级索引不够，表现力太差，预留空间又太浪费，不预留空间又无法扩展，怎么解决？**

既然问题在于预分配，我们使用后分配（瘦分配，或精简分配）解决。**也就是说用户文件数据有多大，我才分配出多大的数组**。举个例子，我们存储 100 G 的文件，那么就要用到三级索引块，最多分配 26214400 个槽位的数组（因为要 26214400 个 block）。如果是存储 6K 的文件，那么只需要 2 个槽位的数组。

**索引数组的后分配**

后分配这里说的是 block 索引编号数组的后分配，需要用到的时候才分配，而不是说，现在用户存储一个 1k 的文件，我上来就给他分配一个 100M 的索引数组，只是为了以后这个文件可能增长到 100 G。

**数据的后分配**

既然这里说到，关于后分配还有一个层面，就是数据所占的空间也是用到了才分配。

**实际的栗子**

先看下下正常的文件写入要做的事情（注意这里只描述主干，实际流程可能，有优化）：

1. 创建一个文件，这个时候分配一个 inode；
2. 在 [ 0，4K ] 的位置写入 4K 数据，这个时候只需要 一个 block 假设编号 102，把这个编号写到 inode->i\_block[0] 这个位置保存起来；
3. 在 [ 1T，1T+4K ] 的位置写入 4K 数据，这个时候需要分配一个 block 假设编号 7，因为这个位置已经落到三级索引才能表现的空间了，所以需要还需要分配出 3 个索引块；
4. 写入完成，close 文件；

这里解释下文件偏移位置 [1T, 1T+4K] 为什么落到三级索引。

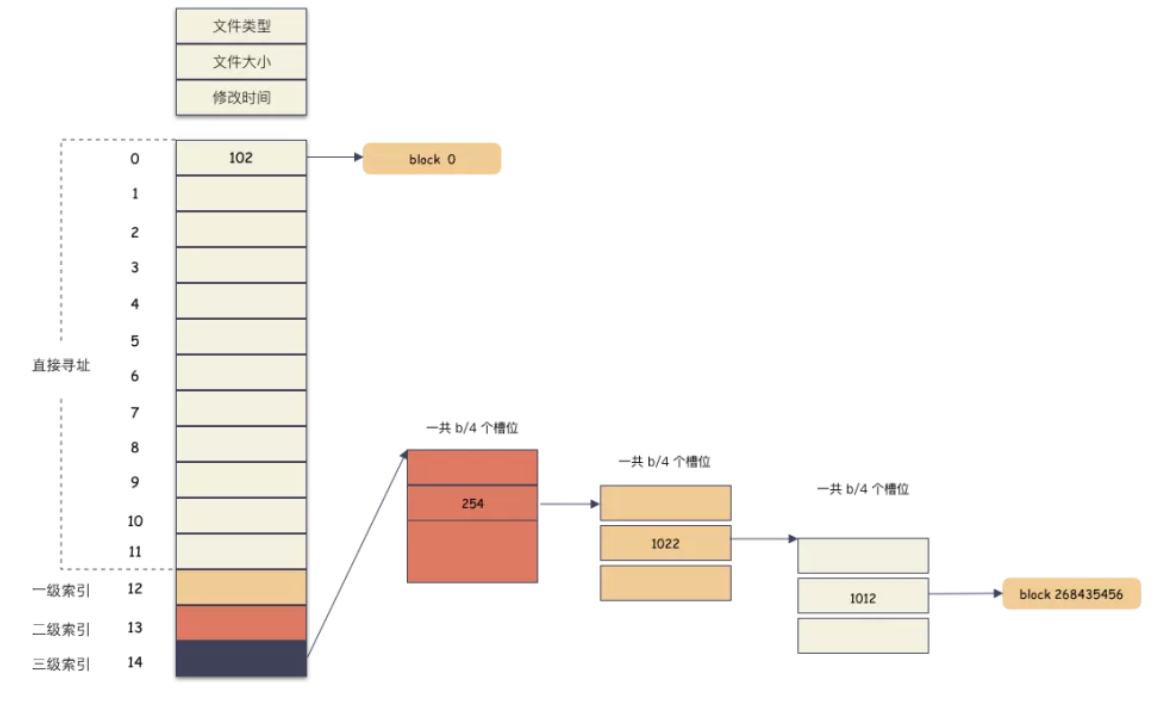
1. offset 为 1T，按照 4K 切分，也就是 block 268435456 块（注意这个是虚拟文件块，不是物理位置）；
2. 先算出范围：直接索引的范围是 [0, 11] 个，一级索引 [12, 1035]，二级索引 [1036, 1049611], 三级索引 [1049612, 1074791435]，（有人如果不知道怎么来的话，可以往前看看 inode 的结构，直接索引 12个，一级索引 1024 个，二级 1M 个，三级 1G 个，然后算出来的）；
3. 268435456 落在三级索引 [1049612, 1074791435] 这个范围；

**实际存储如图**：

计算索引：

12 + 1024 + 1024 \* 1024 + 1024 \* 1024 \* 254 + 1024 \* 1022 + 1012 =  268435456

实际的物理分配如图：



因为偏移已经用到了 3 级索引，所以除了用户数据的两个 block ，中间还需要 3 个间接索引 block 分配出来。

如果要读 [1T, 1T+4K] 这个位置的数据怎么办？

**流程如下**：

1. 计算 offset 得出在第 268435456 的位置；
2. 读出三级索引 inode->i\_block[14] 里存储的 block 编号，找到对应的物理 block，这个是第一级的 block；
3. 然后读该 block 的第 254+1 个槽位里的数据，里面存储的是第二级的 block 编号，把这个编号读出来，通过这个编号找到对应的物理 block；
4. 读该 block 的第 1022 +1 个操作的数据，里面存储的是第三级的 block 编号，通过这个编号可以找到物理 block 的数据，里面存储的是用户数据所在 block 的编号；
5. 读该 block 第 1012+1 个槽位里存储的编号，找到物理 block，这个 block 里存的就是用户数据了；

这个时候，我们的文件看起来是超大文件，size 等于 1T+4K ，但里面实际的数据只有 8 K，位置分别是  [ 0，4K ] ，[ 1T，1T+4K ]。

**重点：文件 size 只是 inode 里面的一个属性，实际物理空间占用则是要看用户数据放了多少个 block 。**

**划重点：没写数据的地方不用分配物理 block 块**。

**没写数据不分配物理块？那是什么？那就是我们下面要说的稀疏文件。**

稀疏文件本质上就是计算机文件，用户不感知，文件系统支持稀疏文件只是为了更有效率的使用磁盘空间而已。稀疏文件就是后分配空间的一种实现形式，做到真正用时才分配，最大效率的利用磁盘空间。

就以上面举的栗子，文件大小 1T，但是实际数据只有 8K，这种就是稀疏文件，**逻辑大小和实际物理空间是可以不等的**。文件大小只是一个属性，文件只是数据的容器，没有用户数据的位置可以不分配空间。

**为什么要支持稀疏语义？**

还是以上面 1T 的文件举例，如果这 1T 的文件只有首尾分别写了 4K 的数据，而文件系统却要分配 1T 的物理空间，**这里将带来巨大的浪费**。何不等存了用户数据的时候再分配了，实际数据有多少，才去分配多大的 block ，何必着急的预分配呢？

**后分配本着用多少给多少的原则，尽量有效的利用空间。**

后分配还有一个优点，这也减少了首次写入的时间，怎么理解？

因为，如果文件大小 1T，就要分配 1T 的空间，那么初始分配需要写入全零到空间，否则上面的数据可能是随机数。

**对于稀疏文件空洞的地方，不占用物理空间，但要保证读的时候返回全 0 数据的语义，即可。**

又一个知识点：有时候稀疏文件的空洞和用户真正的全 0 数据是无法区分的，因为对外表现是一样的。

**稀疏文件也要文件系统支持，并不是所有的文件系统都支持稀疏语义，比如 ext2 就没有，ext4 才有稀疏语义，支持的标志是实现文件系统的 fallocate 接口。**

**稀疏语义接口**

为了知识的完整性，简要介绍稀疏语义的几个接口：

1. preallocate（预分配）：提供接口可以让用户预占用文件内指定范围的物理空间；
2. punch hole（打洞）：提供接口可以让用户释放文件内指定范围的物理空间；

这两个操作刚好相反。

**预分配的意思是？**

就是说，当你创建一个 1T的文件，如果你没写数据，这个时候其实没有分配物理空间的，支持稀疏语义的文件系统会提供一个 fallocate 接口给你，让你实现预分配，也就是说把这 1T 的物理空间现在就分配出来。

**思考：这个有什么好处呢？**

* 第一，如果你命中注定要 1T 的空间，预分配是有好处的，把空间分配的工作量集中在初始化的时候一把做了，避免了实时现场分配的开销；
* 第二，如果不提前占坑，很有可能等你想要的时候已经没有空间可占用了。所以你把物理空间先占好，就可以安心使用了；

**打洞（punch hole） 是干啥的呢？**

这个调用允许你把已经占用的物理空间释放掉，从而达到快速释放的目的。这种操作在虚拟机镜像的场景用得多，通常用于快速释放空间，punch hole 能够让业务更有效的利用空间。

这里有一个知识点：punch hole 的调用要保证 4k 对齐才能释放空间。

举个例子，比如：

punch hole [0, 6k] 的数据，你会发现只有 [0, 4k] 的数据物理块被释放了，[4k, 6k] 所占的 4k 物理块还占着空间呢。

这个很容易理解，因为磁盘的物理空间是划分成 4k 的 block，这个是最小单位了，不能再分了，你无法切割一个最小的单位。

**值得注意的是，就算你没有 4k 对齐的发送调用，fallocate 也不会报错，这个请注意了。**

**cp 快速的原因**

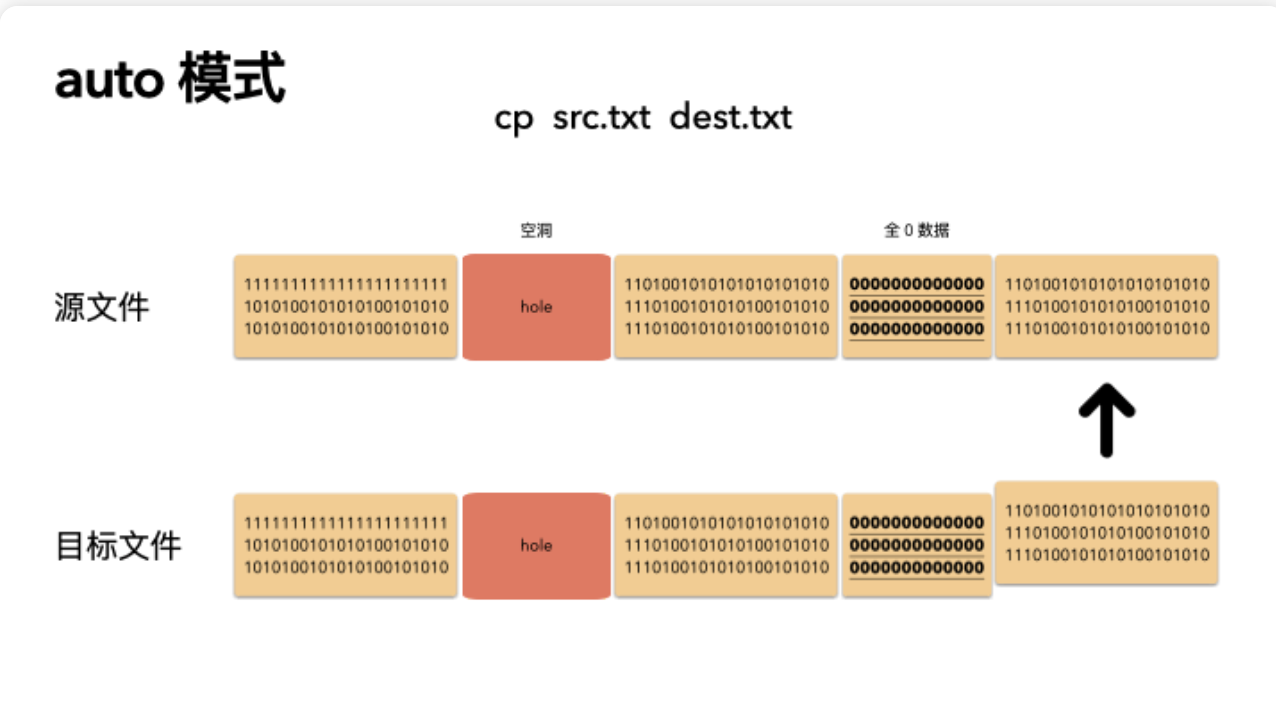
梳理到这里，cp 的秘密已经彻底揭开了，cp 一个 100G 的文件为什么那么快？

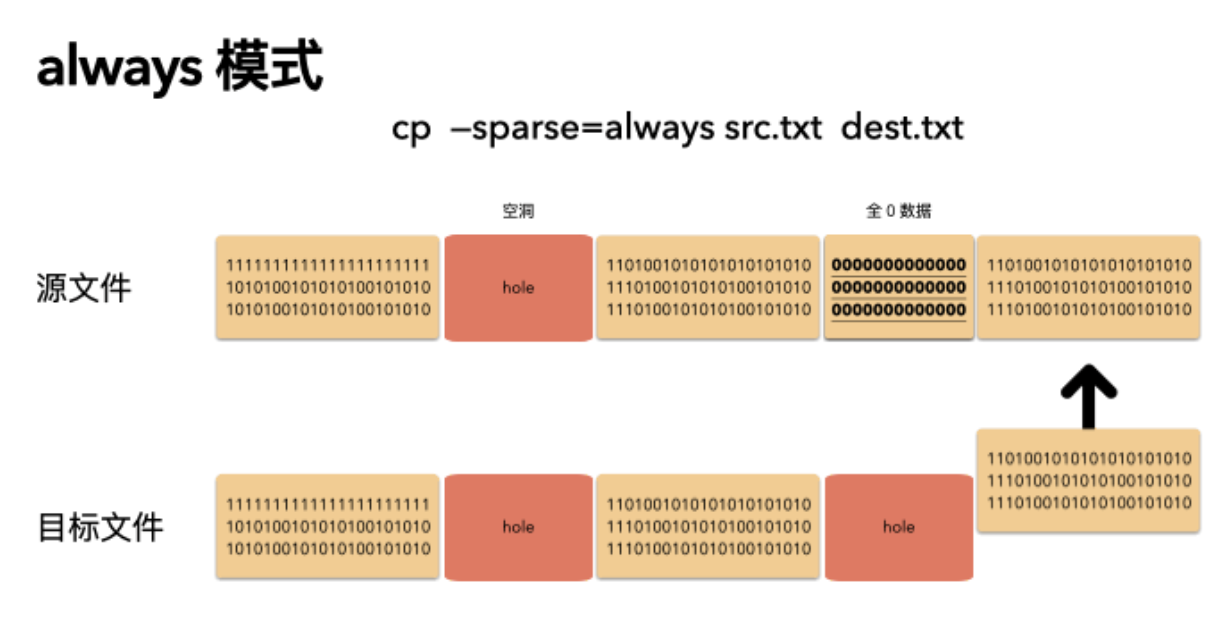
因为源文件是稀疏文件啊，文件看似 100G，实际只占用了 2M 的物理空间。文件系统将文件大小和物理空间占用这两个概念解耦，使得有更灵活的使用姿势，更有效的使用物理空间。

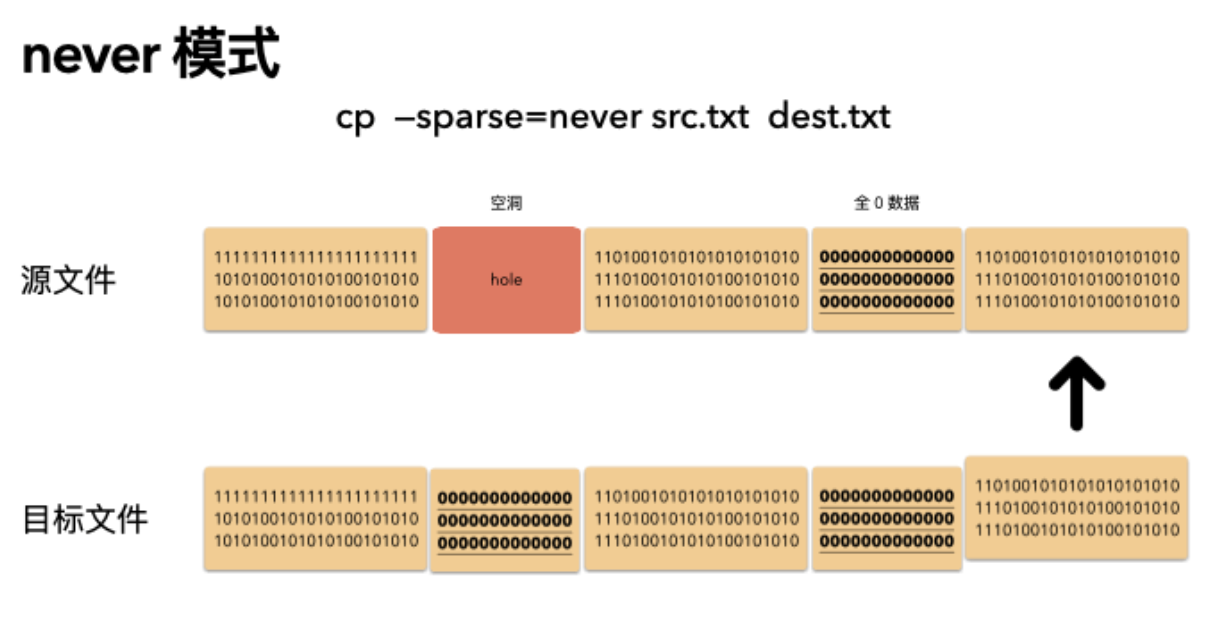
cp 默认的情况下，通过文件系统提供的 fiemap 接口，获取到文件所有的空洞信息，然后跳过这些空洞，只 copy 有效的数据，极大的减少了磁盘 io 的数据量，所以才那么快。

总结下 cp --sparse 三个参数的特点：

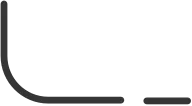
1. auto 模式：默认模式，最一致的模式（如果没有用户全0 块数据，那么可能也是速度最快的），会根据源文件的实际空间占用复制数据，目标文件和源文件一致。无论是文件 size 还是物理 blocks；
2. always 模式：追求最小空间占用的模式，就算源文件不是稀疏文件，而仅仅是有些连续大块的全 0 数据，也会尝试在目标文件上 punch hole，从而节省空间，**这种方式会导致目标文件的物理 blocks 可能比源文件要小**；
3. never 模式：最低效，速度最慢的方式。这种方式无论源文件是啥，全都是实打实的复制，不管是空洞还是全 0 数据，都会在目标文件写入；







**知识点总结**



1. 文件系统对外提供文件语义，本质只是管理磁盘空间的软件而已；
2. 经典的文件系统主要划分 3 大块 superblock 区，inode 区，block 区（块描述区，bitmap区这里暂不介绍）。一个文件在文件系统的内部形态由一个 inode 记录元数据加上 block 存储用户存储用户数据样子；
3. **文件系统的 size 是文件大小，是逻辑空间大小，文件大小 size 和真实的物理空间并不是一个概念**；
4. 稀疏语义是文件系统提供的一种特性，根本用途是用来更有效的利用磁盘空间；
5. **后分配空间是空间利用最有效的方式**，公有云的云盘靠什么赚钱？就是后分配，你买了 2T 的云盘，在没有写入数据的时候，一个字节都没给你分配，你却是付出 2T 的价格；
6. stat 命令能够查看物理空间占用，Blocks 表示的是扇区（512字节）个数；
7. 稀疏文件的空洞和用户真正的全 0 数据是无法区分的，因为对外表现是一样的（这点非常重要）；
8. cp 命令通过调用 ioctl（fiemap）系统调用，可以获取到文件空洞的分布情况，cp 过程中跳过这些空洞，极大的提高了效率（100G 的源文件，cp 只做了十几次 io 搞定了，所以 1 秒足以）；
9. cp 的 sparse 参数从速度最快，空间最省，数据最拷贝最多，各有特点，小小的 cp 命令出来的目标文件，其实和源文件并不相同，只不过你没注意到；
10. 预分配和 punch hole 其实都是fallocate 调用，只是参数不同而已，调用的时候，**注意要 4k 对齐才能达到目的**；
11. 稀疏文件的 punch hole 应用有很多场景，通常是用来快速释放空间，比如镜像文件；