dbfs设计

```
key-bucket
        |key-value
        |key-value
key-bucket
        |key-bucket |
                    |
```

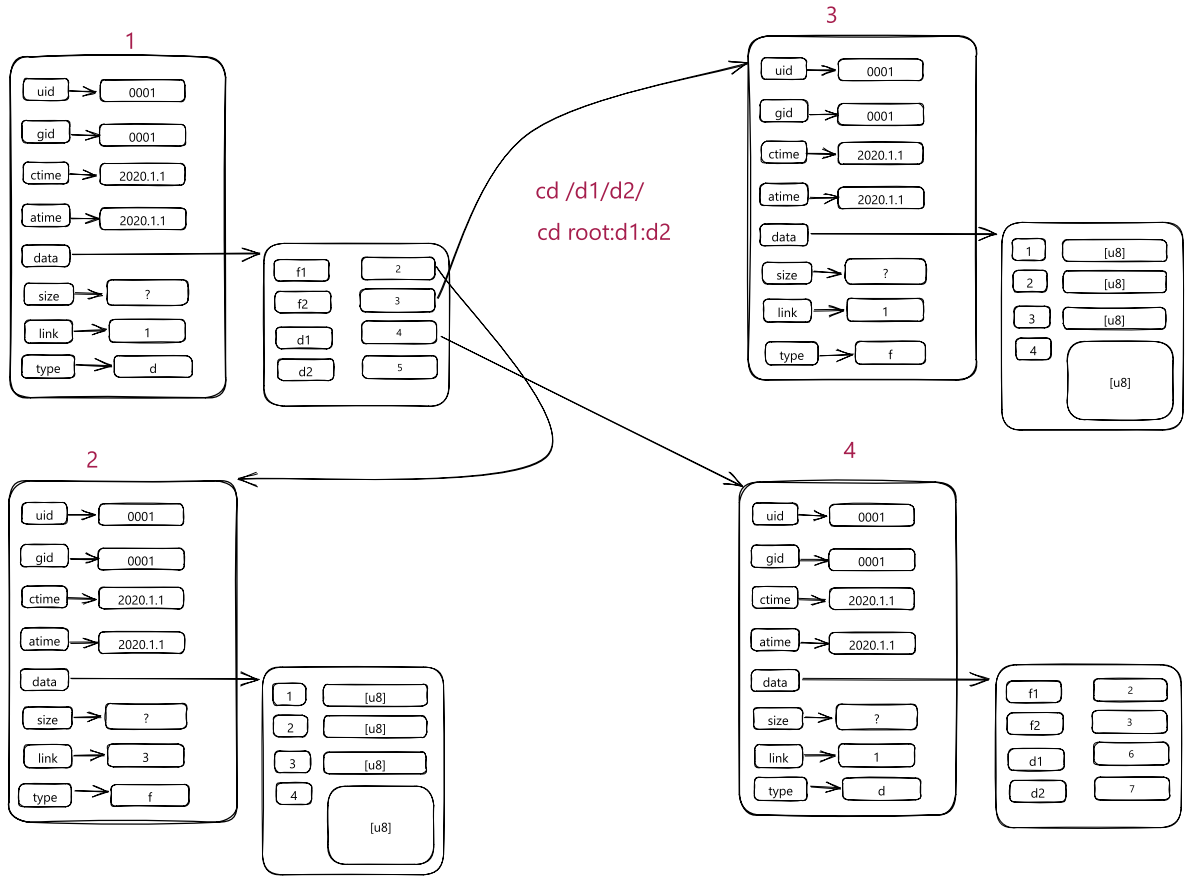
目录与文件的差别：

- 目录只存储子文件名和子目录名
- 目录是否可以按照数据库的嵌套结构存储子目录和子文件
    - 如何处理链接文件？
    - 传统的文件系统只需要inode number 就可以查找到对应的目录或文件，dbfs需要从顶级开始往下查找？

- [ ] 数据库的存储结构
- [ ] 重新修改文档

设计1：

这种设计是为了满足linux中VFS的路径解析算法，对于一个路径`/d1/dd1/f1`来说，VFS会依次解析`/`、`d1`、`dd1`、`f1`，即从根目录下查找得到`d1`，再从`d1`中查找得到`d2`，如果dbfs需要在vfs的框架下运行，就需要有类似于传统文件系统的结构可以依次地去查找文件/目录。



在key-value数据库中，存在两种结构，一个是key-data,一种是key-bucket。这个bucket就像一张表，又可以存储key-data和key-bucket从而形成一种嵌套的结构。

在上面所示的图片中,将文件或者目录都使用bucket这个数据结构表示，这个bucket的编号是唯一的，就像inode_number一样，文件的属性用key-data字段来存储，同样的，文件数据或者目录数据也是key-data字段**(注: 数据也可以使用key-bucket的结构存储，但如果引入这个结构，访问数据时将会发生跨页面的跳转，导致性能下降**)，将其作为key-data字段也需要进一步分析其形式。

对于目录或者文件来说，其本质上来说没有区别，都只是存储数据的一个容器，只是按照传统的说法，目录存储的数据是其子目录或者子文件的索引信息，文件存储的是用户实际想要存储的数据。bucket这个数据结构将文件和目录统一起来，两者存储的数据都是key-value对，一种直观的想法是:

1. 对于目录来说，其保存的是子文件/子目录，假设其包含文件f1,f2,子目录d1,那么它可以包含三个字段:

   `f1: 1` `f2: 2` `f3: 3`, 包含的文件名称作为key, 其对应的bucket 编号为value

2. 对于文件来说，其保存的是用户实际数据，那么它的字段可以是

   `data: [u8]`, 但是如果这样存储文件数据，当文件发生写操作的时候，由于一般key-value数据库不会提供修改的功能，只有删除和添加的操作，因此对于大文件来说，修改就意味着先删除在插入的动作，这是非常降低性能的操作，这里的一种优化方式是对数据进行分片,将数据存储为

   `data1: [u8]` `data2: [u8]` `data3: [u8]` , 每次对文件进行追加就插入一条`data{i} :[u8]`, 但是这也会造成一种现象，即如果追加的数据很少，只有几个byte,那么新建一个key-value对就不划算了， 所以再进一步的优化是 规定一个`data` 存储一个固定大小的数据，比如512byte, 当追加数据的时候，如果最后一个`data{i}`未满512bytes 就在此基础上追加，否则就插入新的key-value对。在这种设计下，当文件的读写发生在中间部分时，会出现数据跨key-value对的情况，但比起直接一个key-data存储文件数据，这种设计会减少很多不必要的开销

如果想要将文件和目录的形式统一，可以将目录存储数据格式也修改为`data{i}: [u8]`的形式， 这时value就可以是`f1:2` 。

上述的这种设计也可以方便地完成硬链接/软连接的实现。

一些常见操作的实现

create file/dir : 数据库根据bucket的标识号找到对应的bucket, 添加一条`data{i}: f:number`键值对，并创建一个新的bucket，填充属性信息与控制信息键值对。

mv file/dir： 数据库根据旧目录的标识号找到对应bucket，从bucket中删除对应的`data{i}: f:number`,在新目录下插入一条新的键值对

cp file/dir： 数据库根据源目录的标识号找到对应bucket，查找需要复制的文件的键值对，得到其唯一标识符后，再查找标识符得到文件的bucket。 在目标目录下插入一条键值对，新建一个bucket,填充信息，并将复制的文件的数据拷贝至新的bucket中。

这些操作与传统的文件系统没有太大的差别。只是在数据库中这些操作得到了简化，因为只是键值对的删除和插入操作。

对文件/目录统一的一些思考

上述的设计中，将普通文件/目录文件/链接文件都抽象为bucket的形式，文件相关的内容全部表示为键值对的形式，此时不同文件的表示已经初步得到了统一。此时可以更近一步，不再区分传统意义上文件/目录，**一个bucket就对应一个存储数据的载体**,用户可以认为这是一个普通文件，或是一个目录，或是一个链接文件，只需要在这个bucket存储一条key-value对，比如`type:Normal file` `type:Directory` `type:Symlink` , 而在传统文件系统中权限控制和属性列表 此时也全部交给用户指定(注: 也可以交给系统指定，但此时系统又得根据类型进行区分，但此时我们已经去掉了这种传统的类型区别)， 比如用户可以在一个bucket中创建`uid: 001` `ctime:2023.1.1` 键值对以表明这个bucket的拥有者是001,创建的时间是2023.1.1。如果用户想要插入数据，那么就在bucket中插入一条键值对，如果想要插入多条数据，那么用户就可以对这些数据做一个预处理保证键的唯一性（bucket的键是唯一的，一个简单的方法是唯一键生成方式可以是当前的时间），将这些数据表达为一批键值对，插入到bucket中。

bucket存储的内容全部交给用户管理，那么传统的文件系统结构就不存在了，那么传统的关于文件系统的系统调用语义也就不同了。当然我认为这是必然出现的事情，传统的文件系统相关调用，比如`ls` `cat` `read` `write`等隐含地表示了要操作的对象是文件或者是目录，而现在已经不存在这种区别了，那么统一的bucket表达形式如何完成这些系统调用想要完成的功能呢？这里有两种方式可以选择:

1. 使用统一的bucket模拟传统文件系统

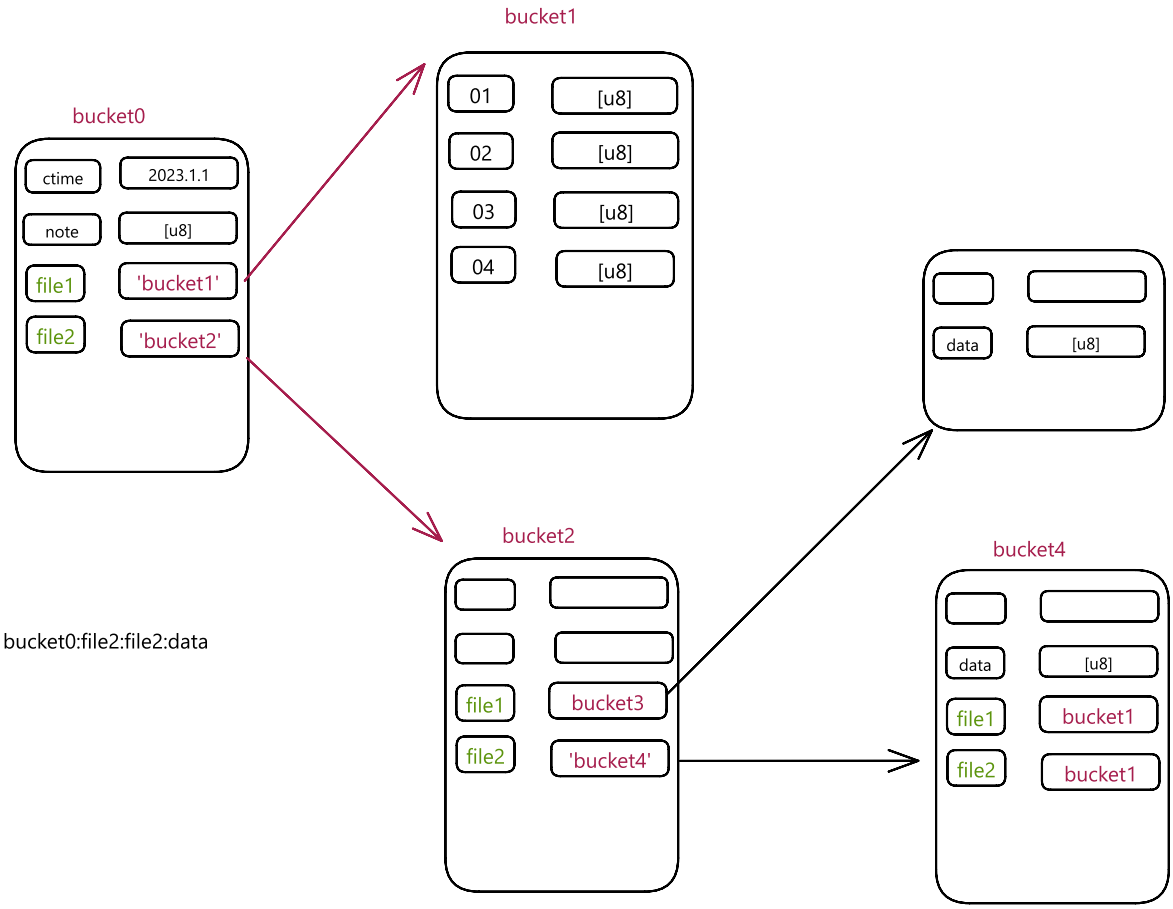
这种方式就是上述dbfs的设计方式，这种设计方式可以完全按照传统文件系统的运行方式运行。但是可以看到，这个时候bucket存储的数据已经被系统介入，因为里面不同类型的bucket代表了不同的文件类型，用户只能对bucket进行有限的控制。

2.抛弃原有的系统调用以及其语义

如果不关注原有的系统调用，而是根据这种统一的设计实现新的系统调用，那么就应该关注用户在使用文件时想要做什么？需要做什么？在传统意义上，用户使用文件来保存数据，并且为了更好组织这些文件引入了目录，为了便利和资源的有效利用又引入了链接文件，这就是用户想要做的事情，保存数据很简单，任何一种简单的数据结构都可以成为文件用来保存数据，而引入目录之后，磁盘上存储的东西就不只是单纯的文件数据了，需要增加新的数据结构来适应这种变化，因此在目录中存储的数据就变成了目录项，用来指向文件的所在位置。而硬链接的引入则需要使得多个文件名指向同一份文件数据，这就需要一种标识使得文件数据变得唯一，inode的设计很好地完成了这个工作。这些用户需要的东西，在统一模型上如何实现呢？

思路

上述提到，key-value数据库提供了`key-data` (注: data 就是一个[u8]数组) 和 `key-bucket` 的数据结构，这两种数据结构可以形成嵌套的结构。但一个key只能有两种value。



- bucket不分类型，数据全部由key-value对构成，value可以是data([u8])，也可以是bucket，这个bucket可以是**全局空间的bucket.**，也可以是**局部嵌套的bucket**，如何创建，怎么创建由用户决定。
- key-value对由用户进行解释，由于比如对一个`bucket0:file1`， 用户需要传递一个函数以解释如何对这个key对应的value进行解释，用户提供的函数形式为

```
enum Para<'a>{
    Data(&'a [u8]),
    Bucket(bucket),
}
fn (p:Para)
```

那么在查找到这个key时，根据其value构建para并调用函数执行。

对于这样bucket数据由用户进行解释的需求，是否就可以将原来那些文件系统相关系统调用全部删除掉？并添加一个系统调用，其形式大致是

```
enum Para<'a>{
    Data(&'a [u8]),
    Bucket(bucket),
}
fn syscall_interpretate(path:&str,func:fn(p:Para))
// func: 桶/[u8]的解析函数
```

此时原来那些系统调用完成的事情由提供的`func`完成。

这里牵扯到的一个问题是是否保留传统文件系统中路径游走和缓存的概念，当然这里路径游走的概念与传统的有点细微差别，这里的含义是指用户解释一个级联键值对时是否需要一个类似vfs层的结构来缓存遇到的bucket结构，从而在下一次在访问这个bucket时加速查找而不是从头开始查找。因为在这个统一模型下，如果出现buckt级联的情况，那么当用户需要解释一个较长的键值对时，比如根据上图所示的: `bucket0:file2:fil2:data`

1. 不需要路径游走和缓存

   对于这个级联的键值对，从第一个bucket0开始查找，找到其key {file2} 对应的value， 上图中是'bucket2'，那么再查找全局空间中bucket2, 再找到其key {fil2} 对应的value {bucket4}, 再到全局空间查找到bucket4， 最后找到 key{data} 指向的数据

2. 存在路径游走和缓存的概念

   内核空间中存在类似于vfs的数据结构，用以缓存bucket的信息，缓存的信息就是bucket中的键值对，在第一次访问时从磁盘上读取信息并缓存。那么对这个级联的键值对，就不需要每次到磁盘进行查找，可能只需要在内存中遍历就可以得到。

更近一步的，对于这些级联的key-value对，每一个分量的处理是一样的吗？从上述所说肯定是不一样的，比如 级联对`bucket0:file2:fil2:data`，如果我们提供的处理函数是对应这个data对应的数据，那么前面的`:file2:file2`用户又是如何解释呢，这里就存在一个**默认实现** 的问题，因为要存在这种级联的键值对处理，说明用户将这些bucket使用一些键值对连接了起来，那么在对这种级联键值对处理时，中间的键值对处理按照进入下一个bucket处理。用户提供的处理函数只针对最后一个键值对分量。

那么考虑到默认实现，我们是否可以为用户提供一些现成的，简单的处理函数使用呢，因为很多时候用户可能就是想看一下这个key对应value是什么，那么只需要一个简单的打印字符串或者打印[u8]数组函数。比如我们可以提供下面这种简单的函数:

```
fn print_value_with_string(para) //将value解析为字符串打印
fn print_value_with_u8array(para) //将value直接打印不解析
fn step_into(para) // 进入value所对应的bucket
```

对于级联对`bucket0:file2:fil2:data`来说，处理时中间分量默认使用`step_into`，末尾分量由用户决定。

- 权限控制是统一模型中比较难处理的一部分，这一部分有待继续分析和设计。

一些传统操作在这个统一模型上的体现

create : 传统的文件系统中这个操作一般用于创建文件，统一模型中则对应于创建一个key-value对，这个键值对可以表达用户的任意想法，可以提供一个系统调用

```
fn sys_create(key:&str,para:Para,is_global:bool)
```

1. 如果用户想创建一个新的bucket来存储一些其它数据，并且其想在一个bucket中存储一个名字指向这个bucket，那么可能的调用就是

```
sys_create("bucket0:mykey:mykey1",Para::Bucket(),true)
```

这个操作将会进行在bucket0找到mykey的值，根据这个值查找其对应的bucket ，再在此bucket中插入键值对,其key为mykey1, value将会是在全局空间中创建的bucket的唯一编号。

2. 如果用户只是单纯地想为这个bucket添加一点内容信息，则则可能的调用就是

```
sys_create("bucket0:mykey:mykey1",Para::Data([u8]),false)
```

这个操作只会在bucket中添加一个简单的key-data键值对，data就是用户想要输入的数据。

move : 传统文件系统中move用户移动文件或者目录，在统一模型下，则可能表达了用户想要从一个bucket中将数据移到另一个bucket中，为这个操作提供的系统调用可能是

```
fn sys_move(old_key:&str,new_key:&str)
```

具体而言，就是将old_key对应的value移动到new_key中，这与传统的操作非常类似。

rename : rename在传统文件系统中，因为有inode和目录项的抽象，rename操作只会修改目录项的内容，而在统一模型下，rename表达了修改key的需求，如果key-value模型不提供修改功能，即只有删除和添加操作时，rename操作将会损害性能，尤其在其存储的数据很大时更为严重，因为这个操作将等价于删除和添加两个操作。为了提供这个的性能，需要添加**额外的优化措施**

设计2:

.....

参考资料

[数据库作为文件系统 视频] | 黑客新闻 (ycombinator.com)

zboxfs/zbox: Zero-details, privacy-focused in-app file system. (github.com)

redisfs：Replication-Friendly Redis-based filesystem (steve.fi)

filesystems - Why do hard links exist? - Unix & Linux Stack Exchange

linux文件系统权限管理 - 简书 (jianshu.com)