简单的类 UFS 文件系统的设计与实现

一 选题背景

文件系统是组织管理文件的重要手段，通过为一块磁盘空间挂载磁盘系统，能够以常见的规范接口访问磁盘中的文件。

如今，随着技术的迭代，已经出现了不少文件系统，常见的如 fat, NFTS, UFS 等，它们具备不同的特点，同时又都实现了打开文件、读取文件、写入文件等公有接口。

可以这么理解，文件系统可定义为一个实现了打开文件、读取文件、写入文件等常见文件操作接口的组织管理系统。

本文将介绍一个基于 libfuse 实现的简单的类 UFS 文件系统——*SampleFS*. 实现本系统，有助于洞察文件系统的底层实现，对文件系统有更深的理解。

二 方案论证(设计理念)

本系统基于 libfuse 实现，libfuse 是一个用户级文件系统设计框架，可以用于编写一个用户级的文件系统，其中的文件操作接口将借由 libfuse 框架的内核处理，并重定向到文件系统开发者提供的文件操作接口，所以，只要实现了这些文件操作接口，就可以借由 libfuse 框架实现一个用户级文件系统。

为了实现一个简单的文件系统，本文提出的文件系统实现了以下接口函数（通过填充 libfuse 提供的接口结构）：

```
static const struct fuse_operations hello_oper = {
    .init       = SFS_init,
    .getattr    = SFS_getattr,
    .readdir    = SFS_readdir,
    .open       = SFS_open,
    .read       = SFS_read,
    .mkdir      = SFS_mkdir,
    .rmdir      = SFS_rmdir,
    .mknod      = SFS_mknod,
    .create     = SFS_create,
    .utimens    = SFS_utimens,
    .write      = SFS_write,
    .unlink     = SFS_unlink
};
```

其中创建普通文件时，会优先调用 SFS_create 函数；创建设备文件时，会调用 SFS_mknod 函数；通过 touch 命令修改文件时间时，会调用 SFS_utimesns；其他函数功能都可简单地根据函数名推测功能。

另外，为了组建一个文件系统，需要实现 inode 等数据结构的管理；需要定义文件系统磁盘块的大小；需要定义文件（目录视为一种特殊文件）的头部用以表明文件的类型等信息（众所周知，

扩展名等手段只是一种约定俗成的规则，难以完全区分文件的类型，另外，文件头往往还需要包括更多的信息，故定义文件头结构是必须的）。文件系统还需要定义超级块、位图块等结构。这些结构都定义在 SampleFS.h 头文件中。头文件中还包含了一些用于操作文件的辅助函数。

还有一点，由于文件系统一般是一块磁盘空间的组织管理系统，因此有必要为文件系统分配一块空间，即将文件系统挂载到磁盘空间上，其中磁盘空间路径在 SampleFS.h 中定义：

```
    char imgPath[] = "/home/krxk/fuse-3.16.2/build/example/SFS_Img"; // 设备载体
```

该路径为绝对路径，为了在其他地方使用该系统，可修改此路径后重新构建。

三 过程论述

本节将介绍 *SampleFS* 的设计总体步骤。

1.1 定义文件系统结构

文件系统结构由需求文档导出，结构如下：

SB	Inode 位图	数据块位图	Inode 区	数据区

SampleFS 以块为单位，每个区包含的块数在 SampleFS.h 中定义：

```
// 下列为各个区域所用块数
const int sb_count = 1;
const int inodeBitmap_count = 1;
const int dataBitmap_count = 4;
const int inode_count = 512;
const int dataBlock_count = 15866; // 8*1024*1024/512-1-1-4-512=15866
```

在文件管理中，*SampleFS*, 每个文件由一个 inode 结构管理，而对于大文件，往往需要多个块来存储数据，这样通过在 inode 中保存文件数据所在磁盘块的块直接地址可能是不够的，所以需要通过多级间接地址的方式来完成目的。直接地址、间接地址的相关结构可在 *SampleFS.h* 头文件中找到：

```
// 定义地址level 的起始地址，按顺序读写，从level_0 开始，填满一个层级再到下一级
const int level_0 = 0; // 直接
const int level_1 = 4; // 一级间接
const int level_2 = 5; // 二级间接
const int level_3 = 6; // 三级间接
```

代码中的 level_k 表示 k 号的 inode 地址指针是某一级类型的磁盘块地址。

为了简单，*SampleFS* 对于文件夹的子文件（包括目录）数量做了限制，在 *SampleFS.h* 中定义：

```
#define max_child_count 20 // 目录下最大子文件数量
```

值得一提的是，由于 libfuse 对上层系统的磁盘块大小在读取文件时存在一定的耦合关系，在 *SampleFS.h* 中在定义了上层文件系统（一般为系统默认）的磁盘块大小：

```
const Native_block_size = 4096; //原生系统的块大小
```

为了分别支持普通文件/设备文件与目录，*SampleFS* 通过定义文件头的方式实现：

```
// 为了简化且实现文件系统的易扩展性，目录头/文件头 设计为单独占一块，如此考虑是因为可以可以在往后很方便地实现扩展而不影响文件内容
struct dentry // 由于目录本质上也是一种文件，故文件也采用此结构体
{
    char fileName[9]; // 文件名(预留一个终止符位置)
    char postFix[4]; // 扩展名(未支持，可考虑扩展)
    short int inodeNo; // inode 号，实际使用12位
    short int childInodeNo[max_child_count]; // 子文件iNode 号
};

struct fileObj { // 文件头部
    char fileName[9]; // 文件名
    char postFix[4]; // 扩展名
    short checksum; // 根据文件头部信息生成的校验和，用于区分是否为一个文件或目录
};
```

另外还在 SampleFS.c 中实现了一个校验算法用以区分普通文件与目录，下列函数通过为普通文件生成校验码并在外部辅助函数中填入 fileObj 文件头结构的 checksum 中，用以区分目录与普通文件。此种方式具备良好的扩展性，不仅有利于在未来支持更多的文件类型，也有助于实现文件的加密功能：

```
short HelpGenFileObjHeadChecksum(struct fileObj* ptrFileObj);
```

1.2 模拟分配磁盘设备

libfuse 框架需要将文件系统挂载到一块磁盘空间上，可通过执行 *CreateDiskFile.sh* 完成，命令如下：

```
#!/bin/sh
dd if=/dev/zero of=./SFS_Img bs=8M count=1
```

通过该代码完成分配还有一个好处：磁盘空间默认置零，可为后期使用省去一些繁琐的置零。

1.3 初始化文件系统

初始化文件系统大致需要完成的工作如下：

1. 根据文件系统结构划分对应的区块（如超级块、inode 位图、数据位图等）。
2. 将根目录的相关信息写入磁盘块。

大致代码执行步骤如下（具体见 *init_disk.c*）：

```
int main(int argc, char *argv[])
{
  // 打开磁盘
  // 划分超级块： 1 块
  // 划分inode 位图块: 1 块 1*512*8=4K 个inode标记，即文件系统支持最多4K个文件
  // 划分数据位图块： 4块 4*512*8=16K 个数据块标记，16K*512=8M 的数据空间总支持大小
  // 划分inode块: 需要 4K 个inode，每个inode 64字节，共需 4K*64/512 = 512 块
  // 初始化第一个inode（根目录）
  // 初始化第一个dataBlock
  // 关闭磁盘
  return 0;
}
```

1.4 文件系统相关接口

本节将简要介绍 *SampleFS* 的相关文件操作接口（如 SFS_read），在此之前，先介绍一些较为底层且重要的辅助函数。

1.4.1 重要辅助函数的简要介绍

1.4.1.1 在 *SampleFS.h* 文件中定义的重要的辅助函数

a) 位图操作辅助函数

由于 inode 位图与数据位图结构极为相似，同时又涉及位运算，适合封装为一个统一的操作以简单操作，降低错误率。可通过如下辅助函数方式实现：

```
struct bitmap_inode {
    char available[4 * 1024 / 8];
};

struct bitmap_dblock {
```

```
    char available[2048]; // 8 * 1024* 1024 / 512 / 8 = 2048
};
// 辅助函数： 读取,设置第 n 个bit的标志。为了简化，不进行越界检查。
// 由于 char* 位于位图结构体的头部，所以根据 C语言规则， 可直接向下列函数传递结构体指针
// 注意： bit 从 1 开始编号，否则将越界
int getBitmapValue(char* bytes,int bit); // 输入字节数组，获得指定节点的值
void setBitmapValue(char* bytes ,int bit ,int Value);
```

由于 *bitmap_inode*、*bitmap_dblock* 的首个成员都是一个 char 数组，所以可很方便地直接为 **BitmapValue* 函数传入一个 *bitmap_inode/bitmap_dblock* 类型的指针，如：

```
struct bitmap_inode* ptrBi = malloc(sizeof(struct bitmap_inode));
struct bitmap_dblock* ptrBd = malloc(sizeof(struct bitmap_dblock));
…
setBitmapValue(ptrBi, avail_InodeNo, 1);
setBitmapValue(ptrBd, avail_dataBlockNo, 1);
```

b) 获取区块偏移辅助函数

在 *SampleFS* 中，频繁地涉及区块的定位操作，通过以下辅助函数实现，函数功能可通过函数名轻易推测得到：

```
// 辅助函数： 返回区域头部距离文件系统头部的偏移量：单位（字节）.采用 inline 提高效率减少递归并保持扩展性
inline int getSuperBlockOffset()
inline int getInodeBitmapOffset()
inline int getDataBitmapOffset()
inline int getInodeOffset()
inline int getDataOffset()
// 二层封装
inline int getDataOffsetByNum(int nBlock/*数据块编号*/)
inline int getInodeOffsetByNum(int iNodeNo/*inode编号*/)
```

c) 多级地址与链表相互转换辅助函数

这是一个特别的设计，由于 SampleFS 支持一到三级间接地址的实现，这在数据结构上往往需要树结构来维护，而树是一个相对复杂的结构，且优化过程比较复杂。同时，如果需要未来扩展 SampleFS，可能不可避免地修改底层树结构，这将使文件操作接口相关代码与复杂的树操作代码高度耦合，这将不是一个好的设计模式，纵使 SampleFS 使用 C 语言开发，但仍然力求做到：**高内聚，低耦合**。

为了隔离复杂的树算法与文件系统操作接口，SampleFS 采用简单的线性表结构（内存中管理）与文件操作接口相连，并提供了链表与树结构的转换接口辅助函数。

```
// 辅助结构，将间接地址转换为链式结构方便读取
#define AddrNodeCapacity 20 /*10K*/
// 重定义结构体，避免多次声明struct类型
```

```
typedef struct AddrNode {
    short addr[AddrNodeCapacity];
    struct AddrNode* next;
} AddrNode;

//工厂函数，创建链表头
AddrNode* CreateListHead();
//添加链式节点，请在外部保留链表头部指针.注意：函数将修改pCurrent,
*ptrNewAddrNumIndex应初始化为0
void AddAddrToListTail(AddrNode** pCurrent ,short* ptrNewAddrNumIndex/*加入
的新地址的索引*/, short addrNum/*磁盘地址值*/);

/*返回下一个块地址号，若无更多，返回-1. *ptrCurAddrIndex应初始化为 -1*/
int ReadAddrList(AddrNode** pCurrent/*标记当前使用的list Node*/, int*
      ptrCurAddrIndex/*上一次迭代时使用的节点addr索引<AddrNodeCapacity*/);
void FreeAddrList(AddrNode* pHead);

/*辅助遍历间接地址树，转换为链表*/
AddrNode* HelpWalkInodeTable(FILE* fp ,struct inode* pIonde);

/*辅助将链表转换回间接地址树*/
int HelpTidyAddr(FILE* fp, AddrNode* pHead, struct inode* ptrInode);

/*在文件写入过程中，用于申请一块新磁盘块用作数据写入，新的块号将通过链表保存*/
short HelpAllocDataBlock(FILE* fp, AddrNode** ppCurrent, int*
ptrNewAddNumIndex);
```

有了以上辅助函数，就有了以下相对低耦合的文件系统结构，未来若需要修改文件系统接口，也只需要与简单的链表结构打交道，若需要优化间接地址树状结构相关算法，也不需要修改文件系统相关接口，理想情况下，也为团队并行开发提供可能。

1.4.1.2 在 *SampleFS.c* 中定义的重要辅助函数

d) 路径分解遍历辅助函数

文件系统中，路径的遍历操作极为频繁，理所当然应该被封装为接口：

```
int HelpWalkPath(const char *customPath, short int startInodeNum/*遍历的起始
InodeNum*/, char** pNext/*返回下一层遍历路径*/);
```

```
void HelpSplitFileName(const char* customPath, const char* outSplitFileName);

int IsReachPathEnd(char* pNext);

int HelpFindFile(const char* fileName, short int startInodeNum, int goDeep/*是否继续下一层*/);
```

e) 文件校验和生成辅助函数

为了区分普通文件/设备文件与目录，也为了未来扩展为具备加密功能的文件系统，提供了文件校验函数，函数通过一定规则将普通文件/设备文件的文件名+扩展名转为一串校验和，目前可以用于进一步区分普通文件/设备文件与目录。

```
short HelpGenFileObjHeadChecksum(struct fileObj* ptrFileObj) /*函数只生成校验码，不修改文件头部校验码，如需要修改请在外部修改*/
```

f) 文件/目录的 Inode 获取辅助函数

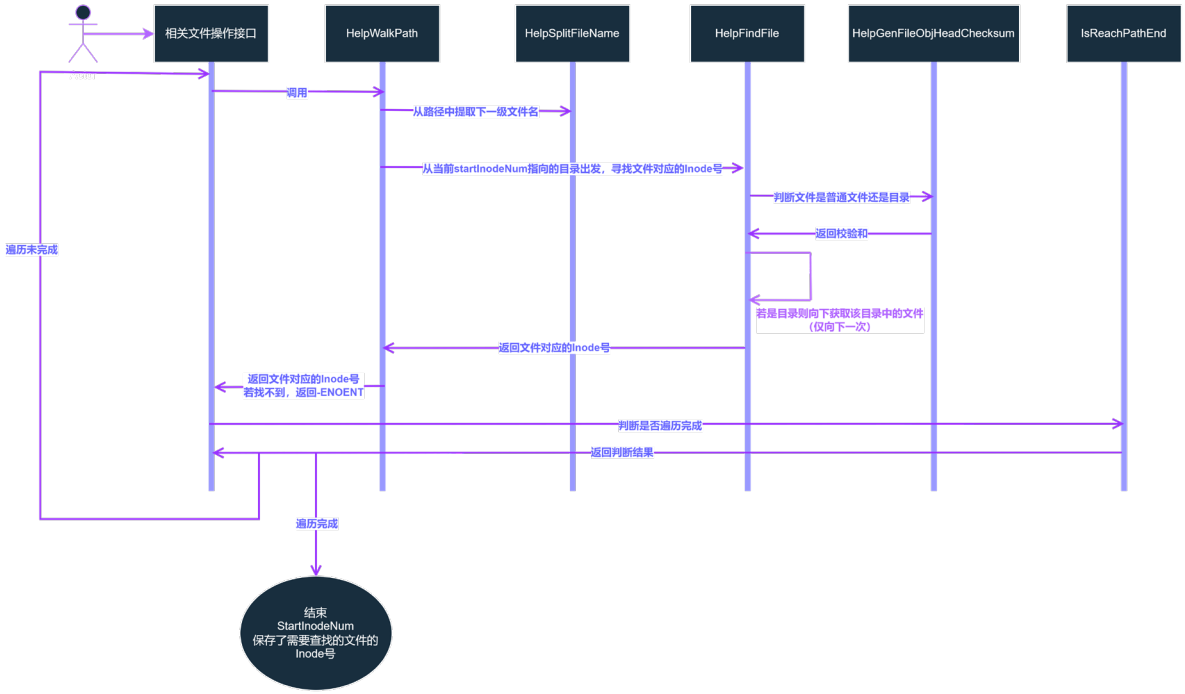
通过上文介绍的 *HelpWalkPath* 函数即可完成。

g) 其余辅助函数

重要的辅助函数已经在上文介绍，但为了完整性与方便理解，以下列出文件中剩余辅助函数原型并在下一节给出依赖关系。

```
void HelpFillStat(struct stat* stbuf, struct inode* ptrInode); // 填充属性结构辅助函数
void HelpGetFileNameFromInodeNum(int inodeNum, char* pFileName); // 通过Inode号获取文件名
void HelpConcatFileName(char* fileName, char* postFix, char* concat); // 连接文件的文件名与扩展名
```

h) 重要辅助函数间的依赖关系



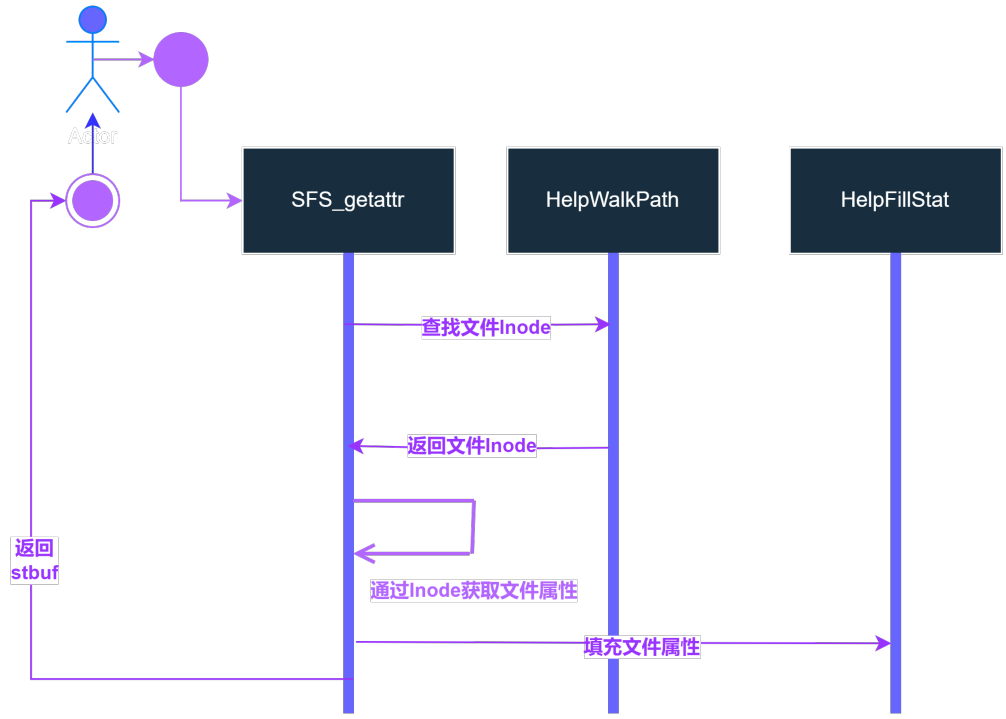
上图便是下列高频代码段的流程图，下列代码执行后，若查找路径的文件存在，*startInodeNum* 将保存该文件的 *Inode* 号：

```
// 按路径查找，直到路径末尾
int startInodeNum = 1;
for(char* pNext = path; !IsReachPathEnd(pNext);) {
    startInodeNum =  HelpWalkPath(pNext, startInodeNum, &pNext);
    if(startInodeNum == -ENONET) {
        // 找不到
        return -ENOENT;
    }
}
```

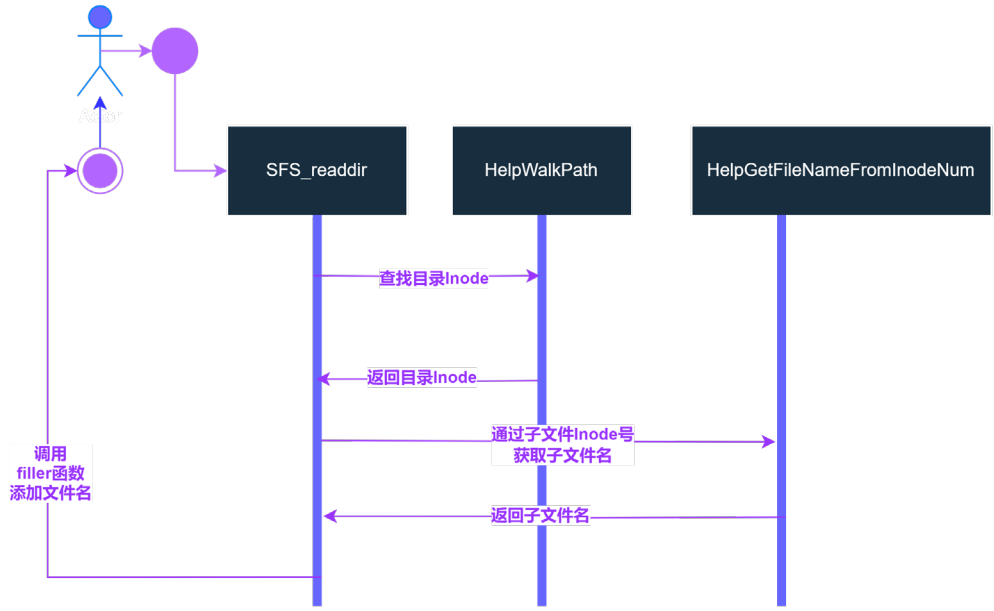
1.4.2 相关文件操作接口的设计思路介绍

本节将通过 *UML* 图的方式介绍相关文件操作接口的设计思路，为了简单且易于理解，图中只展示主线操作，异常情况不展示。具体实现代码可见 *SampleFS.c* 文件。

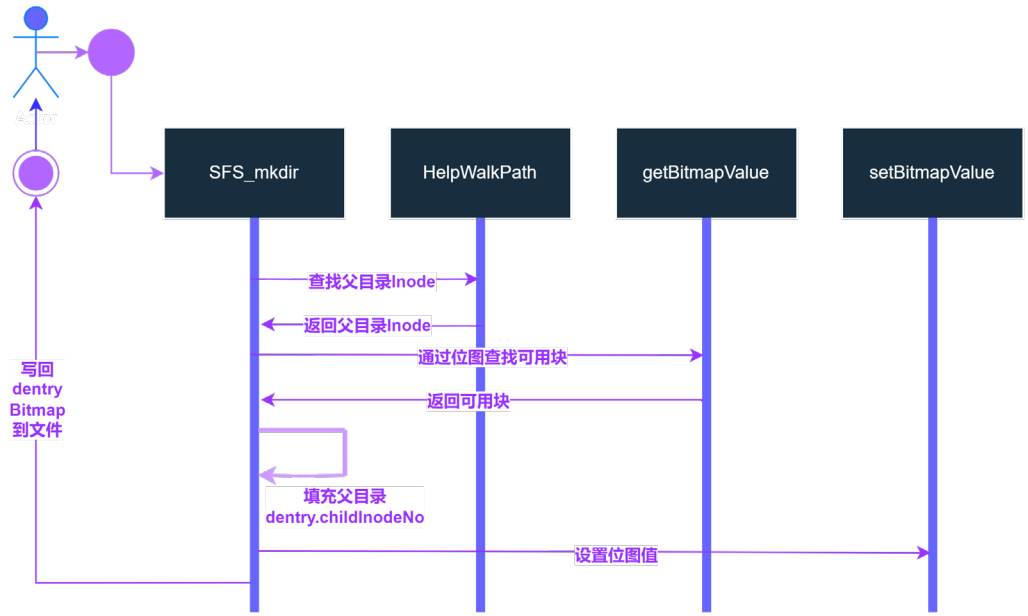
1.4.2.1 SFS_getattr



1.4.2.2 SFS_readdir



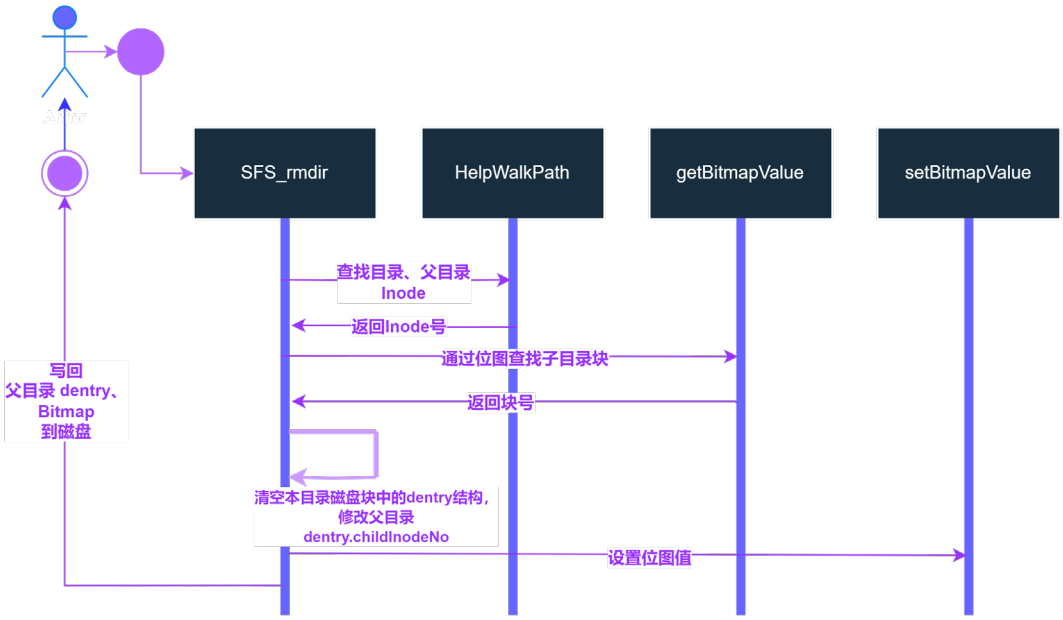
1.4.2.3 SFS_mkdir



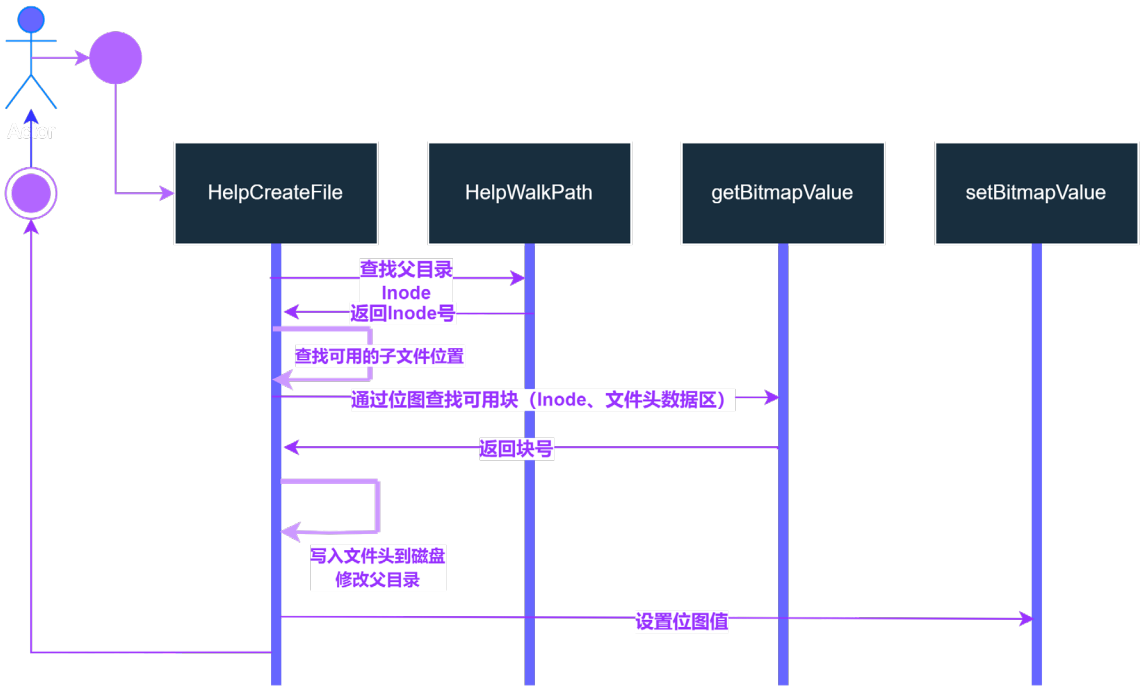
1.4.2.4 SFS_rmdir

代码中的实现与图中顺序有细微区别，使实质上是等同的，另外，判断要删除的目录实际上是普通文件还是目录的操作并未在图中展示，主要由下列代码实现（即依赖于先前介绍的 HelpGenFileObjHeadChecksum 辅助函数）：

```
if(ptrfileObj->checksum == HelpGenFileObjHeadChecksum(ptrfileObj)) {
        // 普通文件非目录
        free(ptrfileObj);
        free(ptrInode);
        fclose(fp);
        return -ENOTDIR;
}
```

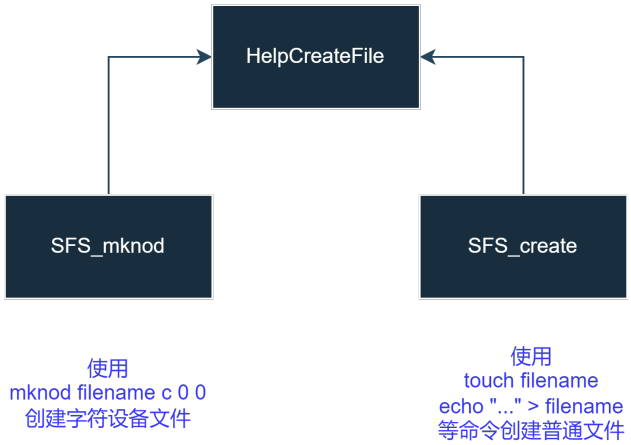


1.4.2.5 SFS_mknod & SFS_create



SFS_mknod 、 SFS_create 实际上仅为上述 HelpCreateFile 辅助函数的套壳。主要区别主要是如下代码中两个函数传入的 *attach_mode* 不同，这是因为对于不同的 shell 命令，libfuse 框架会检测其创建文件的属性（通过调用提供的 SFS_getattr），若属性不同，会提示创建失败。

```
ptrNewInode->st_mode = mode | attach_mode; // 赋予 __S_IFREG 或 __ S_IFCHR 权限，以通过权限检测
```

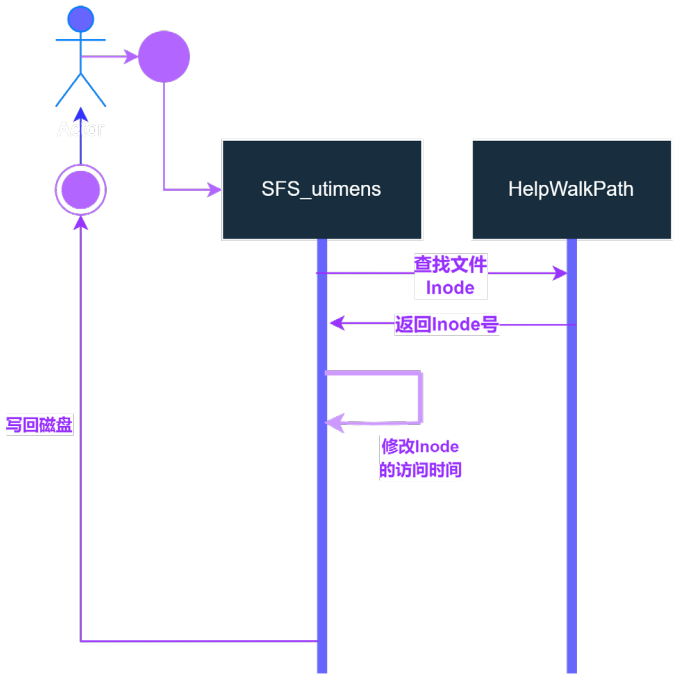


其中根目录下不允许创建文件，图中未展示，主要由下列代码实现：

```
int count = 0;
for(char *temp = path; *temp!='\0'; temp++) {
    if(*temp == '/') {
        count++;
    }
}
if(count <= 1) {
    //直接在根目录创建文件，不应该赋予权限
    return -EPERM;
}
```

1.4.2.6 SFS_utimens

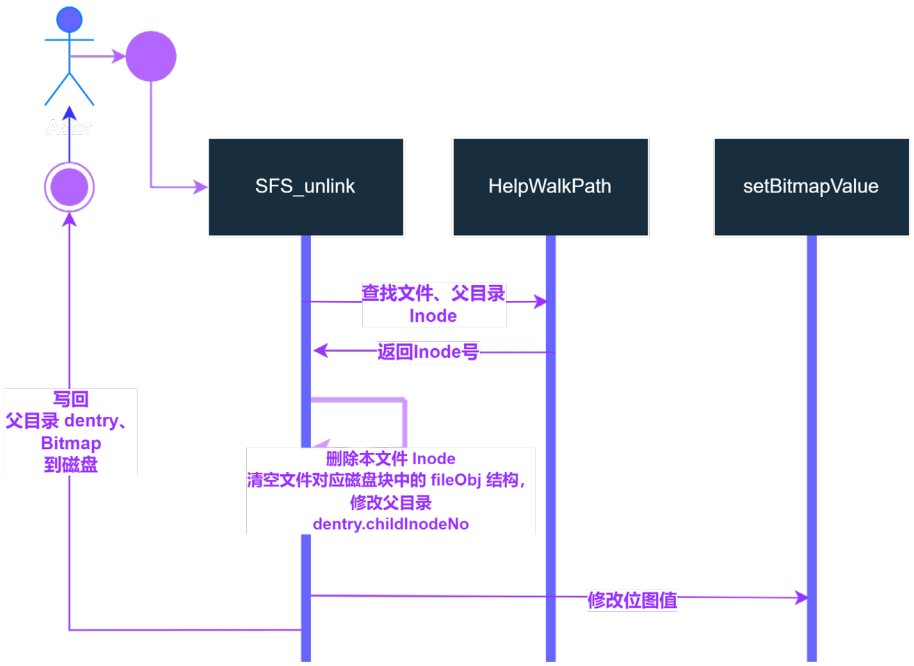
该函数用于支持 *touch* 的 SHELL 命令修改文件时间（由于 *Inode* 中只保存了**访问时间**，故只能修改访问时间）



1.4.2.7 SFS_unlink

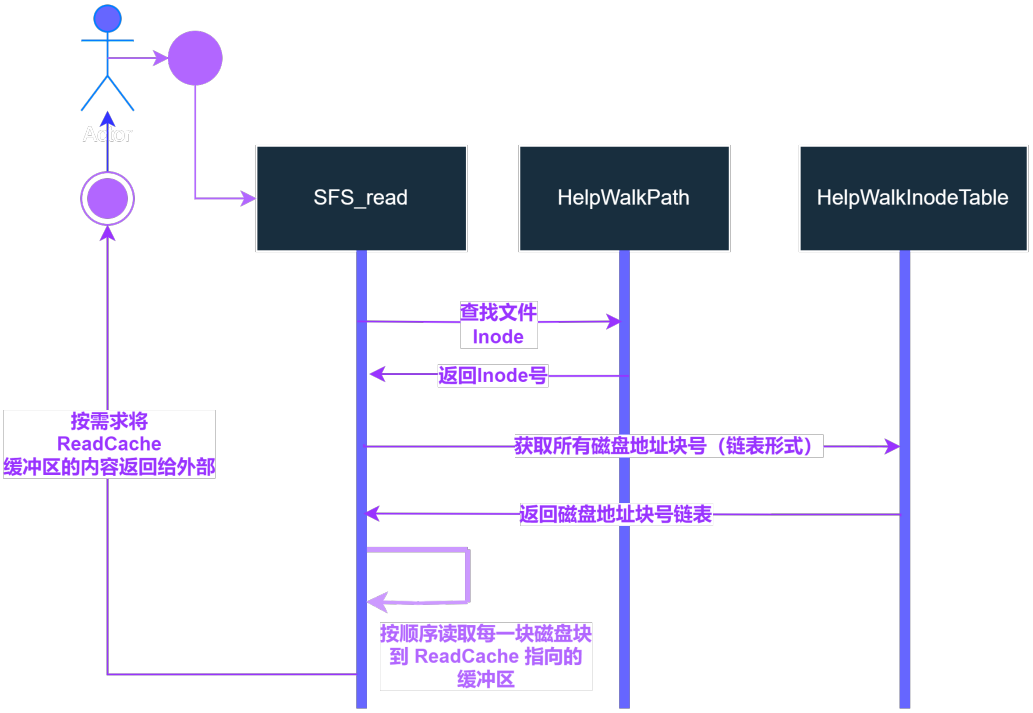
主要操作如下图，图中未展示的获得父目录路径与判断待删除的对象是普通文件还是目录主要由下列代码实现：

```
int pathLength = sizeof(char) * (strlen(path) + 1); // 保存原目录长度

if(HelpGenFileObjHeadChecksum(ptrFileObj) != ptrFileObj->checksum) {
        //非文件，为目录
        free(ptrInode);
        free(ptrFileObj);
        fclose(fp);
        return -EISDIR;
}

// 分离得到父目录
char *parent = malloc(pathLength);
strcpy(parent, path);
char *temp = strrchr(parent, '/');
*temp = '\0';
```



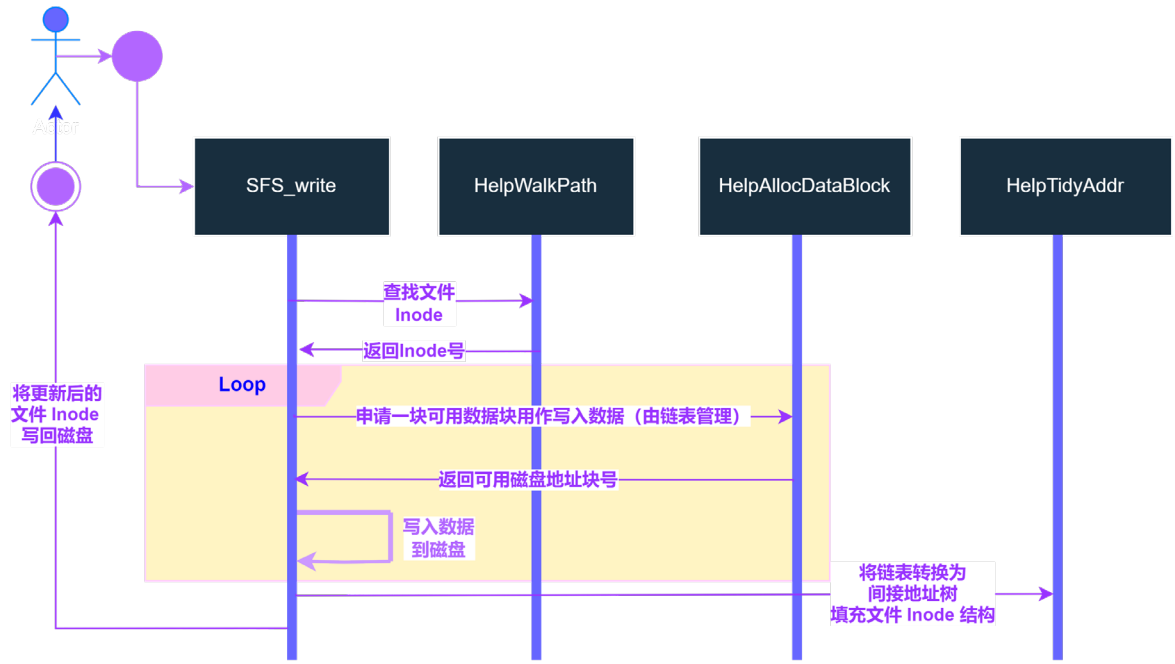
1.4.2.8 SFS_read

剩余两个文件接口函数相对复杂，这是因为它们涉及间接地址树的复杂数据结构，当然，由于加了简单的链表作为隔离层，这两个接口的实现也变得相对简单。



值得注意的是：使用了 ReadCache 缓冲区，这是因为每次读取的字节流是由外部命令（如 Shell 命令）指定的，而 Shell 命令又一般受原生系统的默认块大小的影响，所以在 SFS_read 中建立了缓冲区，一次性将整个文件读到缓冲区，再根据外部调用者需要返回缓冲区的数据。

1.4.2.9 SFS_write



值得注意的是：图中没有直接观察到位图操作，这是由于 HelpAllocDataBlock 辅助函数将自动调用 getBitmapValue 与 setBitmapValue 管理位图。

四 结果分析

SampleFS 支持多级目录的创建，支持文件、空目录的删除；支持创建普通文件与字符设备文件；支持文件的读取与写入；支持修改文件与目录的最后访问时间；支持查看文件的属性；支持查看目录内的文件列表。基本实现了文件系统的常用功能。

五 课程设计总结

通过设计并实现 *SampleFS*，让我进一步加深了对操作系统文件系统的理解，同时也促进我对软件设计模式有了更多的思考，在设计本项目过程中，为了设计更易用，更利于未来维护与扩展的文件系统，在编写代码过程中，贯彻了一些设计模式，尽可能地做到**高内聚、低耦合**，我认为：即使是作为一门面向过程的设计语言 *C*，设计模式、面向对象的思想仍然有助于指导代码的编写。

另外，一个项目的编写过程，往往不可避免地需要经历调试、渐进式开发，故调试器与代码托管工具的使用是很重要的。通过设计 *SampleFS*，我更加熟练地掌握了 *VSCode* 调试器的使用，同时，也进一步熟练了远程开发的（实际上，我是连接 Linux Mint 虚拟机进行开发的，在此过程中，我还掌握了如何将安装了桌面系统的 Linux 以命令行方式启动，这可以节省大量内存），为什么不在虚拟机里安装 *VSCode* 直接开发呢？因为相比实体机，虚拟机的性能真的相形见绌。

六 参考文献

[1] https://github.com/libfuse/libfuse.git

[2] http://libfuse.github.io/doxygen/index.html