技术预研报告

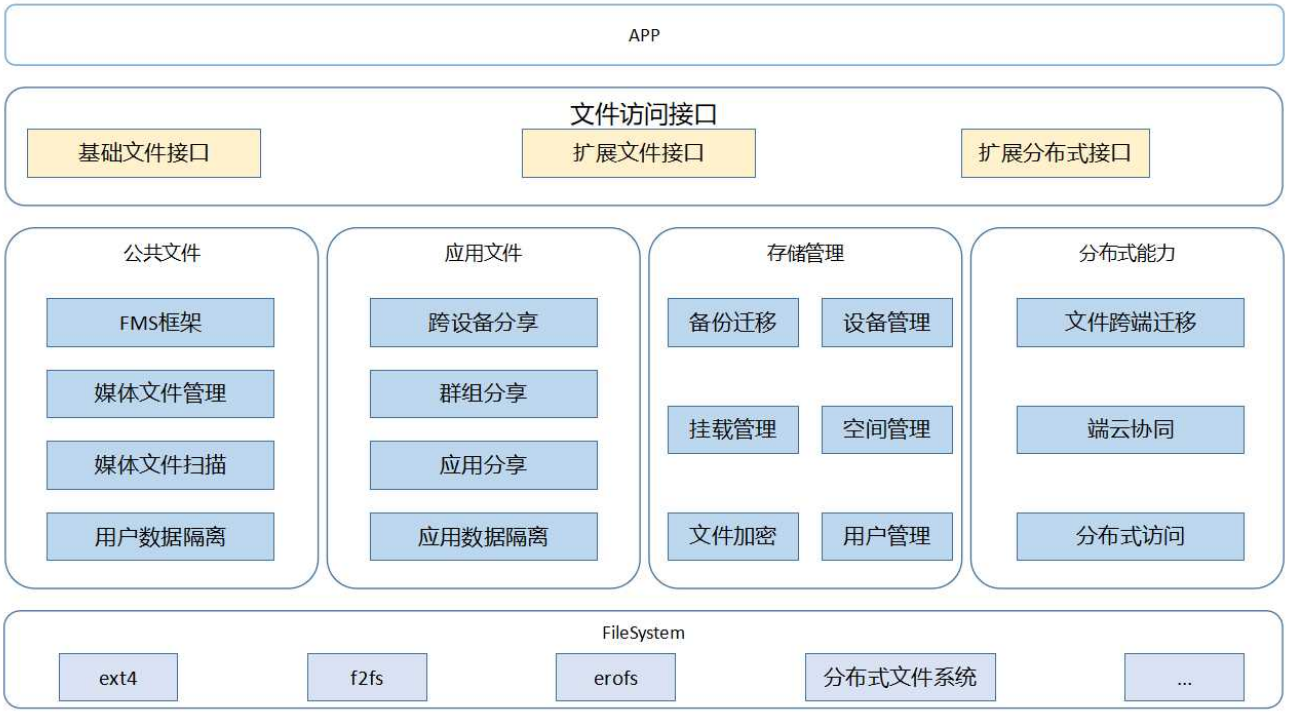
OpenHarmony文件管理子系统

简介

文件管理子系统为OpenHarmony提供一套完整的文件数据管理解决方案, 提供安全, 易用的文件访问能力和完善的文件存储管理能力, 包括:

1. 为应用提供安全的沙箱隔离技术, 保证应用数据安全基础上权限最小化;
2. 统一的公共文件管理能力, 统一公共数据访问入库, 保证用户数据安全, 纯净;
3. 分布式文件系统和云接入文件系统访问框架, 应用可以像使用本地文件一样使用分布式和云端文件;
4. 支持公共数据, 跨应用, 跨设备的系统级文件分享能力;
5. 提供系统的存储管理能力和基础文件系统能力.

图 1 文件管理子系统架构图



文件管理子系统对应用提供文件访问框架, 文件分享框架, 存储管理框架能力.

模块	详细描述
文件访问接口	1. 提供完整文件JS 接口, 支持基础文件访问能力; 2. 提供本地文件, 分布式文件扩展接口.
存储管理	1. 提供数据备份恢复框架能力, 支持系统和应用数据备份克隆等场景; 2. 提供应用空间清理和统计, 配额管控等空间管理能力; 3. 提供挂载管理, 外卡管理, 设备管理及多用户管理等存储存储管理能力.
公共文件	1. 公共数据沙箱隔离, 保证用户数据安全, 纯净; 2. 统一公共数据访问入口, 仅medialibrary;

模块	详细描述
	3. 提供统一的FMF的文件管理框架.
应用文件	1. 为应用提供安全的沙箱隔离技术, 保证应用数据安全基础上权限最小化; 2. 支持应用间文件分享和文件跨设备分享, 支持群组分享.
分布式能力	1. 提供基础分布式跨端访问能力, 支持同账号分布式访问和异账号临时访问; 2. 支持文件跨端迁移能力, 支撑应用迁移, 分布式剪切板等分布式场景.
基础文件系统	1. 支持ext4, f2fs, exfat, ntfs等本地文件系统; 2. 支持分布式文件系统, nfs等网络文件系统; 3. 文件系统相关工具.

本赛题的目标是应用数据目录的备份, 故主要分析应用文件管理.

应用文件管理

应用文件服务

应用文件服务是为应用提供文件分享和管理能力的服务, 包含应用间文件分享, 跨设备同应用文件分享以及跨设备跨应用文件分享的能力.

目录

```
/foundation/filemanagement/app_file_service
|── interfaces                          // 接口声明
|   ├── innerkits                       // 对内接口声明
|   └── kits                            // 对外接口声明
```

备份恢复

备份恢复是为Openharmony设备上三方应用数据、系统应用数据、公共数据提供一套完整的数据备份和数据恢复解决方案。

备份恢复功能主要由三大部分组成：

- 集成在克隆等系统应用中的JS API：负责触发备份/恢复数据。支持获取能力文件，触发备份应用数据，触发恢复应用数据，设置恢复应用数据时安装应用。
- 集成在待备份恢复应用中的备份服务扩展：负责备份恢复具体应用的数据。应用开发者可通过配置备份恢复策略规则，配置备份恢复场景及过滤隐私等目录。
- 具有独立进程的备份服务：主要负责调度备份恢复任务。具体而言，其具体职责包括获取及检查备份恢复能力、管理备份服务扩展的生命周期与并发程度、协调零拷贝传输文件、在恢复时可选择安装应用。

目录

```
/foundation/filemanagement/app_file_service
|── frameworks                          // 框架层
|   └── native
|       └── backup_ext                  // 备份服务扩展
|── interfaces                          // 接口存放目录
|   ├── api
|   ├── inner_api                       // 内部接口声明
```

```
|   └── kits
|       └── js
|           └── backup                  // js外部接口
|── services
|   └── backup_sa                       // 备份恢复服务
|── tests                               // 测试用例
|── tools                               // 备份恢复工具
└── utils                               // 工具套
```

OpenHarmony备份恢复服务

服务组织框架

备份恢复服务入口

```
/foundation/filemanagement/app_file_service/tools/backup_tool/src/main.cpp
```

```
int ParseOpAndExecute(const int argc, char *const argv[])
{
    // 注册下命令
    ToolRegister();
    int flag = -1;
    for (int i = 1; i < argc; i++) {
        // 暂存 {argv[1]...argv[i]};
        vector<string_view> curOp;
        for (int j = 1; j <= i; ++j) {
            curOp.emplace_back(argv[j]);
        }

        // 尝试匹配当前命令，成功后执行
        auto tryOpSucceed = [&curOp](const ToolsOp &op) { return op.TryMatch(curOp); };
        auto &&opeartions = ToolsOp::GetAllOperations();
        auto matchedOp = find_if(opeartions.begin(), opeartions.end(), tryOpSucceed);
        if (matchedOp != opeartions.end()) {
            vector<ToolsOp::CmdInfo> argList = matchedOp->GetParams();
            optional<map<string, vector<string>>> mapNameToArgs = GetArgsMap(argc, argv,
argList);
            if (mapNameToArgs.has_value()) {
                flag = matchedOp->Execute(mapNameToArgs.value());
            }
        }
    }
    if (flag != 0) {
        printf("backup_tool: missing operand\nTry 'backup_tool help' for more
information.\n");
    }
    return flag;
}
```

当调用 `backup_tool` 命令, 命令处理接口会截断 `backup_tool` , 以其第一个参数作为命令, 调用 `main` 函数. 该程序执行时首先注册各个操作, 再从操作表 `ToolsOP::opsAvailable_` 中寻找传入命令对应的操作, 并运行对应的执行函

数.

因此, 备份恢复服务并非一直挂载, 而是直到调用相关操作才加载到内存.

操作表 `ToolsOP::opsAvailable_` 存储了所有的操作, 也可以通过 `ToolsOp::Register` 去注册新的操作, 这需要提供操作, 参数和执行函数. 注册的操作要在 `ToolRegister()` 中调用初始化才能正常运行.

我们只关心其中的备份和恢复操作, OpenHarmony 实现了全量备份, 全量恢复, 增量备份和增量恢复四种相关操作.

以备份操作为例

```
static int32_t InitPathCapFile(const string &pathCapFile, vector<string> bundleNames)
{
    StartTrace(HITRACE_TAG_FILEMANAGEMENT, "InitPathCapFile");
    // SELinux backup_tool工具/data/文件夹下创建文件夹 SA服务因root用户的自定义标签无写入权限 此处调整为软链接形式
    BackupToolDirSoftlinkToBackupDir();

    if (access((BConstants::BACKUP_TOOL_RECEIVE_DIR).data(), F_OK) != 0 &&
        mkdir((BConstants::BACKUP_TOOL_RECEIVE_DIR).data(), S_IRWXU) != 0) {
        throw BError(BError::Codes::TOOL_INVAL_ARG, generic_category().message(errno));
    }

    UniqueFd fdLocal(open(pathCapFile.data(), O_RDWR | O_CREAT | O_TRUNC, S_IRWXU));
    if (fdLocal < 0) {
        fprintf(stderr, "Failed to open file. error: %d %s\n", errno, strerror(errno));
        return -EPERM;
    }
    auto proxy = ServiceProxy::GetInstance();
    if (!proxy) {
        fprintf(stderr, "Get an empty backup sa proxy\n");
        return -EPERM;
    }
    BFile::SendFile(fdLocal, proxy->GetLocalCapabilities());

    auto ctx = make_shared<Session>();
    ctx->session_ = BSessionBackup::Init(
        BSessionBackup::Callbacks {.onFileReady = bind(OnFileReady, ctx, placeholders::_1, placeholders::_2),
                                   .onBundleStarted = bind(OnBundleStarted, ctx, placeholders::_1, placeholders::_2),
                                   .onBundleFinished = bind(OnBundleFinished, ctx, placeholders::_1, placeholders::_2),
                                   .onAllBundlesFinished = bind(OnAllBundlesFinished, ctx, placeholders::_1),
                                   .onBackupServiceDied = bind(OnBackupServiceDied, ctx)});
    if (ctx->session_ == nullptr) {
        printf("Failed to init backup\n");
        FinishTrace(HITRACE_TAG_FILEMANAGEMENT);
        return -EPERM;
    }
    int ret = ctx->session_->AppendBundles(bundleNames);
    if (ret != 0) {
        printf("backup append bundles error: %d\n", ret);
        throw BError(BError::Codes::TOOL_INVAL_ARG, "backup append bundles error");
    }

    ctx->SetBundleFinishedCount(bundleNames.size());
    ctx->Wait();
    FinishTrace(HITRACE_TAG_FILEMANAGEMENT);
```

```
    return 0;
}
```

备份操作并非直接进行, 而是借由 `ServiceProxy` 通过 IPC 发送给服务器进程执行. 备份工具在此时仅起到客户端进程的作用, 向服务器进程发送请求, 并接收服务器进程返回的结果. 这样做允许多进程地执行备份任务, 加速备份过程, 减小命令行响应时间.

为此, 首先创建一个 `ServiceProxy`, 将本地能力写入给定的能力文件 `pathCapFile` 中. 然后初始化一个 `Session`.

`Session` 是用于备份状态管理的类. `Session` 记录了需要备份的应用列表, 与它们对应需要备份和已经备份的文件表. 当收到了一个应用所需要备份的全部文件, 该应用从 `Session` 的记录中删除, 当所有应用全部备份完成, 备份流程结束.

上述功能以文件为单位管理备份. 为了维护单个文件备份的过程, `Session` 中还额外有 `BSessionBackup` 对象. 该对象允许定义回调函数, 来处理备份当中的各种情况(例如备份服务意外死亡), 向命令行输出日志.

`BSessionBackup` 对象还对应一系列备份过程的函数, 每个函数与 `ServiceProxy` 中特定函数对应. 后者向服务器发送消息, 请求服务器执行对应操作.

```
Remote()->SendRequest(static_cast<uint32_t>(IServiceInterfaceCode::SERVICE_CMD_START), data,
reply, option);
```

每个操作对应一个代码, 例如 `BSessionBackup` 中 `Start` 函数, 对应 `ServiceProxy` 中 `Start` 函数, 对应 `SERVICE_CMD_START` 代码.

服务器进程接收到请求后, 将备份任务排入线程池. 线程池中线程启动 `extension` 进程, 这是备份真正发生的进程. 然后线程池中线程向 `extension` 进程发送 `IExtensionInterfaceCode::CMD_HANDLE_BACKUP` 消息. `extension` 进程收到消息并开始处理备份任务, 其准备好需要备份的文件后, 向服务器进程发送 `IServiceInterfaceCode::SERVICE_CMD_APP_FILE_READY` 消息. 服务器进程收到消息后, 执行在 `BSessionBackup` 中定义的回调函数, 进行文件复制, 最终完成备份.

全量备份

全量备份的关键函数是 `extension` 进程执行的 `BackupExtExtension::DoBackup` 函数与服务器进程执行的 `OnFileReady` 函数.

```
int BackupExtExtension::DoBackup(const BJsonEntityExtensionConfig &usrConfig)
{
    HILOGI("Do backup");
    if (extension_->GetExtensionAction() != BConstants::ExtensionAction::BACKUP) {
        return EPERM;
    }

        // /data/storage/el2/backup/backup/
    string path =
string(BConstants::PATH_BUNDLE_BACKUP_HOME).append(BConstants::SA_BUNDLE_BACKUP_BACKUP);
    if (mkdir(path.data(), S_IRWXU) && errno != EEXIST) {
        throw BError(errno);
    }

        //需要备份目录
    vector<string> includes = usrConfig.GetIncludes();
        //排除目录
    vector<string> excludes = usrConfig.GetExcludes();
```

```
    // 获取大小文件信息
    auto [bigFileInfo, smallFiles] = GetFileInfos(includes, excludes);
    for (const auto &item : bigFileInfo) {
                //大文件路径
        auto filePath = std::get<0>(item.second);
        if (!filePath.empty()) {
            excludes.push_back(filePath);
        }
    }

    // 分片打包小文件，单包最多100MB/60000文件，包名 part.[index].tar
    TarMap tarMap {};
    TarFile::GetInstance().Packet(smallFiles, "part", path, tarMap);

        //将打包的小文件加入到大文件信息中
    bigFileInfo.insert(tarMap.begin(), tarMap.end());

    auto proxy = ServiceProxy::GetInstance();
    if (proxy == nullptr) {
        throw BError(BError::Codes::EXT_BROKEN_BACKUP_SA,
std::generic_category().message(errno));
    }

    if (auto ret = IndexFileReady(bigFileInfo, proxy); ret) {
        return ret;
    }
    auto res = BigFileReady(proxy);
    HILOGI("HandleBackup finish, ret = %{public}d", res);
    return res;
}
```

需要备份的应用目录也许包含非常多小文件与一些大文件. 对小文件进行复制非常花费时间. 为此, `DoBackup` 函数首先进行小文件的打包, 它将不大于 `2MB` 的文件分片打包, 放在 `/data/storage/el2/backup/backup/` 下, 包名为 `part.[index].tar`. 分片中每片不超过 `100MB`, 打包不多于 `60000` 个文件. 然后打包的小文件和大文件被同等看待, 它们被当成需要备份的文件, 信息写入 `/data/storage/el2/backup/backup/manage.json`.

```
static void OnFileReady(shared_ptr<Session> ctx, const BFileInfo &fileInfo, UniqueFd fd)
{
    printf("FileReady owner = %s, fileName = %s, sn = %u, fd = %d\n", fileInfo.owner.c_str(),
fileInfo.fileName.c_str(),
           fileInfo.sn, fd.Get());
    string tmpPath = string(BConstants::BACKUP_TOOL_RECEIVE_DIR) + fileInfo.owner;
    if (access(tmpPath.data(), F_OK) != 0 && mkdir(tmpPath.data(), S_IRWXU) != 0) {
        throw BError(BError::Codes::TOOL_INVAL_ARG, generic_category().message(errno));
    }
    if (fileInfo.fileName.find('/') != string::npos) {
        throw BError(BError::Codes::TOOL_INVAL_ARG, "Filename is not valid");
    }
    UniqueFd fdLocal(open((tmpPath + "/" + fileInfo.fileName).data(), O_WRONLY | O_CREAT |
O_TRUNC, S_IRWXU));
    if (fdLocal < 0) {
        throw BError(BError::Codes::TOOL_INVAL_ARG, generic_category().message(errno));
    }
    BFile::SendFile(fdLocal, fd);
    if (fileInfo.fileName == BConstants::EXT_BACKUP_MANAGE) {
        ctx->SetIndexFiles(fileInfo.owner, move(fd));
    } else {
        ctx->UpdateBundleReceivedFiles(fileInfo.owner, fileInfo.fileName);
```

```
    }
    ctx->TryNotify();
}
```

这之后, `extension` 进程将 `manage.json` 文件和每个大文件的信息一一发给服务器进程, 服务器进程调用 `OnFileReady` 函数, 将文件写到 `/data/backup/received/[owner]/[filename]`, 完成备份.

命令行调用命令 `backup_tool backup` 进行备份. 附加参数

名称	可重复	作用	举例
isLocal	否	是否本地备份, 未实装	--isLocal=true
bundle	是	待备份的应用	--bundle com.sample.app2
pathCapFile	否	本地能力文件路径	--pathCapFile /data/backup/tmp

全量恢复

全量备份的关键函数是 `extension` 进程执行的 `BackupExtExtension::DoRestore` 函数与服务器进程执行的 `OnFileReady` 函数.

```
static void OnFileReady(shared_ptr<Session> ctx, const BFileInfo &fileInfo, UniqueFd fd)
{
    printf("FileReady owner = %s, fileName = %s, sn = %u, fd = %d\n", fileInfo.owner.c_str(),
fileInfo.fileName.c_str(),
           fileInfo.sn, fd.Get());
    if (fileInfo.fileName.find('/') != string::npos) {
        throw BError(BError::Codes::TOOL_INVAL_ARG, "Filename is not valid");
    }
    string tmpPath = string(BConstants::BACKUP_TOOL_RECEIVE_DIR) + fileInfo.owner + "/" +
fileInfo.fileName;
    if (access(tmpPath.data(), F_OK) != 0) {
        throw BError(BError::Codes::TOOL_INVAL_ARG, generic_category().message(errno));
    }
    UniqueFd fdLocal(open(tmpPath.data(), O_RDONLY));
    if (fdLocal < 0) {
        throw BError(BError::Codes::TOOL_INVAL_ARG, generic_category().message(errno));
    }
    BFile::SendFile(fd, fdLocal);
    int ret = ctx->session_->PublishFile(fileInfo);
    if (ret != 0) {
        throw BError(BError::Codes::TOOL_INVAL_ARG, "PublishFile error");
    }
    ctx->TryNotify();
}
```

`OnFileReady` 首先将备份目录中的文件写到数据目录. 一些原先打包的小文件需要恢复, 这些文件在 `DoRestore` 中进行解压. 如果用户指定了恢复版本, 这些文件被解压到 `/data/storage/el2/backup/restore/` 文件夹而不是原先的文件夹.

命令行调用命令 `backup_tool restore` 进行恢复.

名称	可重复	作用	举例
depMode	否	是否并行执行恢复	--depMode=true
bundles	是	待恢复的应用	--bundles com.sample.app2

名称	可重复	作用	举例
pathCapFile	否	本地能力文件路径	--pathCapFile /data/backup/tmp

增量备份

OpenHarmony 的增量备份服务器进程部分还未完成, 其服务器接收消息入口

```
int32_t ServiceStub::CmdAppendBundlesIncrementalBackupSession(MessageParcel &data,
MessageParcel &reply)
{
    return BError(BError::Codes::OK);
}
```

处于未完成状态.

优缺点分析

优点

- 使用并行备份方式, 可以对备份进行加速.
- 运用 IPC 实现入口与逻辑进程分离
- 对小文件分片打包, 可以防止备份琐碎的文件花费大量时间, 减小跟踪备份文件所需空间

缺点

- 备份恢复系统大量运用 IPC, 由于开发周期和人员等种种问题
  - 入口和逻辑并未完全分离, 大量信息没有通过 IPC 而是其他方式在进程间传递
  - IPC 通信频繁, 混乱; 系统中存在冗余代码 [1]
  - 在备份中, 备份每个大文件都与客户端进程通信了一遍, 产生了额外的开销, 这些信息可以一次传递.
- 小文件打包之后放到临时目录, 最后才写到备份目录, 进行了冗余的磁盘访问.

[1]: 例如 `DoBackup` 中

```
vector<string> excludes = usrConfig.GetExcludes();

// 获取大小文件信息
auto [bigFileInfo, smallFiles] = GetFileInfos(includes, excludes);
for (const auto &item : bigFileInfo) {
                //大文件路径
    auto filePath = std::get<0>(item.second);
    if (!filePath.empty()) {
        excludes.push_back(filePath);
    }
}
```

局部变量 `excludes` 为排除的文件夹, 它的作用是用于 `GetFileInfos`, 这段代码向其中加入了大文件的路径, 之后就再也没有使用过.