标准库文件系统模块分析

文件模块将集中在操作系统中除文件读写之外的操作：包括创建，删除，属性查看及修改，目录操作等 文件读写的部分将统一放在后继的IO部分去分析

linux的操作系统的文件系统实现

以下是rust在linux适配层提供的文件操作API, 基本上与C语言的标准库实现了一一对应：

目录相关的类型结构及方法，函数

```
//创建目录类型，准备读
pub fn readdir(p: &Path) -> io::Result<ReadDir> {
    let root = p.to_path_buf();
    let p = cstr(p)?;
    unsafe {
        //调用libc函数获得libc::DIR的指针
        let ptr = libc::opendir(p.as_ptr());
        if ptr.is_null() {
            Err(Error::last_os_error())
        } else {
            //对C函数的返回值完成RUST的初步封装
            let inner = InnerReadDir { dirp: Dir(ptr), root };
            //创建多线程安全的RUST目录读类型结构ReadDir
            Ok(ReadDir {
                inner: Arc::new(inner),
            })
        }
    }
}

//类似OwnedFd，拥有了C语言返回的目录的所有权
struct Dir(*mut libc::DIR);
//将目录类型与路径字符串相联系
struct InnerReadDir {
    //C函数返回的目录句柄
    dirp: Dir,
    //目录路径字符串
    root: PathBuf,
}

//对目录结构的多线程封装结构,用于读目录
pub struct ReadDir {
    inner: Arc<InnerReadDir>,
}
```

```rust
//目录下的每个条目的类型结构
pub struct DirEntry {
    dir: Arc<InnerReadDir>,
    entry: dirent64_min,
    // We need to store an owned copy of the entry name on platforms that use
    // readdir() (not readdir_r()), because a) struct dirent may use a flexible
    // array to store the name, b) it lives only until the next readdir() call.
    name: CString,
}

// 针对linux的dirent的部分的存储
struct dirent64_min {
    d_ino: u64,
    d_type: u8,
}

//针对目录读实现Iterator以简化操作
impl Iterator for ReadDir {
    type Item = io::Result<DirEntry>;

    //将复杂的C语言操作用next自然的呈现
    fn next(&mut self) -> Option<io::Result<DirEntry>> {
        unsafe {
            loop {
                //linux已经保证了readdir的线程安全性
                //readdir64会读取下一个
                super::os::set_errno(0);
                let entry_ptr = readdir64(self.inner.dirp.0);
                if entry_ptr.is_null() {
                    //系统调用出错处理
                    return match super::os::errno() {
                        0 => None,
                        e => Some(Err(Error::from_raw_os_error(e))),
                    };
                }

                // 以下从C调用返回的结构创建DirEntry结构.
                // 具体细节请参考相关的C语言dirent64的手册
                // 因为不能直接对entry_ptr做解引用，所以用一个
                // 局部变量将需要的内容拷贝出来
                let mut copy: dirent64 = mem::zeroed();

                let copy_bytes = &mut copy as *mut _ as *mut u8;
                let copy_name = &mut copy.d_name as *mut _ as *mut u8;
                let name_offset = copy_name.offset_from(copy_bytes) as usize;
                let entry_bytes = entry_ptr as *const u8;
                let entry_name = entry_bytes.add(name_offset);
                ptr::copy_nonoverlapping(entry_bytes, copy_bytes, name_offset);

                //获取需要的值
```

```rust
                let entry = dirent64_min {
                    d_ino: copy.d_ino as u64,
                    d_type: copy.d_type as u8,
                };

                let ret = DirEntry {
                    entry,
                    name: CStr::from_ptr(entry_name as *const _).to_owned(),
                    dir: Arc::clone(&self.inner),
                };
                //去掉目录中的 ./及../
                if ret.name_bytes() != b"." && ret.name_bytes() != b".." {
                    return Some(Ok(ret));
                }
            }
        }
    }

}

//实现对目录的关闭，进行资源释放
impl Drop for Dir {
    fn drop(&mut self) {
        let r = unsafe { libc::closedir(self.0) };
        debug_assert_eq!(r, 0);
    }
}

//针对目录下每个条目的操作
impl DirEntry {
    //用目录名及entry的名字连接形成新的path字符串
    pub fn path(&self) -> PathBuf {
        self.dir.root.join(self.file_name_os_str())
    }

    //获取Entry的名称字符串
    pub fn file_name(&self) -> OsString {
        self.file_name_os_str().to_os_string()
    }

    //利用文件属性操作获取Entry的属性数据
    pub fn metadata(&self) -> io::Result<FileAttr> {
        let fd = cvt(unsafe { dirfd(self.dir.dirp.0) })?;
        let name = self.name_cstr().as_ptr();

            //见try_statx解析
            if let Some(ret) = unsafe { try_statx(
                fd,
                name,
                libc::AT_SYMLINK_NOFOLLOW | libc::AT_STATX_SYNC_AS_STAT,
                libc::STATX_ALL,
```

```rust
            ) } {
                return ret;
            }

        let mut stat: stat64 = unsafe { mem::zeroed() };
        cvt(unsafe { fstatat64(fd, name, &mut stat, libc::AT_SYMLINK_NOFOLLOW)
})?;
        Ok(FileAttr::from_stat64(stat))
    }

    //entry的文件类型
    pub fn file_type(&self) -> io::Result<FileType> {
        match self.entry.d_type {
            //以下是设备文件
            libc::DT_CHR => Ok(FileType { mode: libc::S_IFCHR }),
            libc::DT_FIFO => Ok(FileType { mode: libc::S_IFIFO }),
            libc::DT_LNK => Ok(FileType { mode: libc::S_IFLNK }),
            libc::DT_REG => Ok(FileType { mode: libc::S_IFREG }),
            libc::DT_SOCK => Ok(FileType { mode: libc::S_IFSOCK }),
            libc::DT_DIR => Ok(FileType { mode: libc::S_IFDIR }),
            libc::DT_BLK => Ok(FileType { mode: libc::S_IFBLK }),
            //DirEntry的文件类型
            _ => self.metadata().map(|m| m.file_type()),
        }
    }

    //以下为entry结构内部成员获取

    pub fn ino(&self) -> u64 {
        self.entry.d_ino as u64
    }

    fn name_bytes(&self) -> &[u8] {
        self.name_cstr().to_bytes()
    }

    fn name_cstr(&self) -> &CStr {
        &self.name
    }

    pub fn file_name_os_str(&self) -> &OsStr {
        OsStr::from_bytes(self.name_bytes())
    }
}
//没有实现对DirEntry的Drop trait

//删除目录下所有项目的实现
pub use remove_dir_impl::remove_dir_all;

mod remove_dir_impl {
    use super::{cstr, lstat, Dir, DirEntry, InnerReadDir, ReadDir};
```

```rust
    use crate::ffi::CStr;
    use crate::io;
    use crate::os::unix::io::{AsRawFd, FromRawFd, IntoRawFd};
    use crate::os::unix::prelude::{OwnedFd, RawFd};
    use crate::path::{Path, PathBuf};
    use crate::sync::Arc;
    use crate::sys::{cvt, cvt_r};

    use libc::{fdopendir, openat, unlinkat};

    //将目录按照文件打开
    pub fn openat_nofollow_dironly(parent_fd: Option<RawFd>, p: &CStr) ->
io::Result<OwnedFd> {
        let fd = cvt_r(|| unsafe {
            openat(
                parent_fd.unwrap_or(libc::AT_FDCWD),
                p.as_ptr(),
                libc::O_CLOEXEC | libc::O_RDONLY | libc::O_NOFOLLOW |
libc::O_DIRECTORY,
            )
        })?;
        //返回一个文件描述符
        Ok(unsafe { OwnedFd::from_raw_fd(fd) })
    }

    //用已有的目录的文件描述符打开目录
    fn fdreaddir(dir_fd: OwnedFd) -> io::Result<(ReadDir, RawFd)> {
        //映射到C语言调用
        let ptr = unsafe { fdopendir(dir_fd.as_raw_fd()) };
        if ptr.is_null() {
            return Err(io::Error::last_os_error());
        }

        //以下形成RUST的目录类型结构

        let dirp = Dir(ptr);
        // 这里容易出错，因为Dir会关闭fd，所以此次OwnedFd不应再存在
        // 否则其生命周期终结会导致也调用fd的关闭操作。
        // 这里是RUST底层编程因为其所有权额外增加程序负担的情况
        let new_parent_fd = dir_fd.into_raw_fd();
        // 无法获取完整路径，此函数不能用于需要获取完整路径的操作
        let dummy_root = PathBuf::new();
        Ok((
            ReadDir {
                inner: Arc::new(InnerReadDir { dirp, root: dummy_root }),
            },
            new_parent_fd,
        ))
    }

    //判断目录下的条目是否为目录
```

```rust
    fn is_dir(ent: &DirEntry) -> Option<bool> {
        match ent.entry.d_type {
            libc::DT_UNKNOWN => None,
            libc::DT_DIR => Some(true),
            _ => Some(false),
        }
    }

    //递归的删除目录下所有条目
    fn remove_dir_all_recursive(parent_fd: Option<RawFd>, path: &CStr) ->
io::Result<()> {
        // 用文件描述符打开目录
        let fd = match openat_nofollow_dironly(parent_fd, &path) {
            Err(err) if err.raw_os_error() == Some(libc::ENOTDIR) => {
                return match parent_fd {
                    // 删除文件 unlink...
                    Some(parent_fd) => {
                        cvt(unsafe { unlinkat(parent_fd, path.as_ptr(), 0)
}).map(drop)
                    }
                    // ...unless this was supposed to be the deletion root
directory
                    None => Err(err),
                };
            }
            result => result?,
        };

        // 打开目录
        let (dir, fd) = fdreaddir(fd)?;
        // 用Iterator遍历
        for child in dir {
            let child = child?;
            let child_name = child.name_cstr();
            //判断child是否为目录
            match is_dir(&child) {
                Some(true) => {
                    //递归调用
                    remove_dir_all_recursive(Some(fd), child_name)?;
                }
                Some(false) => {
                    //删除文件
                    cvt(unsafe { unlinkat(fd, child_name.as_ptr(), 0) })?;
                }
                None => {
                    //有些操作系统需要
                    remove_dir_all_recursive(Some(fd), child_name)?;
                }
            }
        }
```

```
        cvt(unsafe {
            //删除本身
            unlinkat(parent_fd.unwrap_or(libc::AT_FDCWD), path.as_ptr(), 
libc::AT_REMOVEDIR)
        })?;
        Ok(())
    }

    fn remove_dir_all_modern(p: &Path) -> io::Result<()> {
        let attr = lstat(p)?;
        //判断是否是link
        if attr.file_type().is_symlink() {
            crate::fs::remove_file(p)
        } else {
            //能够满足文件及目录需求
            remove_dir_all_recursive(None, &cstr(p)?)
        }
    }

    pub fn remove_dir_all(p: &Path) -> io::Result<()> {
        remove_dir_all_modern(p)
    }
}
```

操作系统的每一个资源都是文件，这些资源即满足文件的统一操作，又具备自己的独特性。目录是一个典型的例子。

文件相关的类型结构及方法，函数： 创建一个文件的函数代码如下：

```
//以创建的方式打开一个文件并设置权限
fn open_to_and_set_permissions(
    to: &Path,
    reader_metadata: crate::fs::Metadata,
) -> io::Result<(crate::fs::File, crate::fs::Metadata)> {
    use crate::fs::OpenOptions;
    use crate::os::unix::fs::{OpenOptionsExt, PermissionsExt};

    let perm = reader_metadata.permissions();
    //利用OpenOptions打开一个文件
    let writer = OpenOptions::new()
        //见OpenOptions说明
        .mode(perm.mode())
        //可写
        .write(true)
        //没有则创建
        .create(true)
        .truncate(true)
        .open(to)?;
    let writer_metadata = writer.metadata()?;
```

```rust
    if writer_metadata.is_file() {
        // 设置文件权限
        writer.set_permissions(perm)?;
    }
    Ok((writer, writer_metadata))
}

//打开文件
fn open_from(from: &Path) -> io::Result<(crate::fs::File, crate::fs::Metadata)>
{
    use crate::fs::File;
    use crate::sys_common::fs::NOT_FILE_ERROR;

    //此File::open是RUST标准库对外接口，与下面的File不是一个
    //实质是OpenOptions::new().read(true).open(from.as_ref());
    let reader = File::open(from)?;
    //获取文件属性
    let metadata = reader.metadata()?;
    //判断是否是文件
    if !metadata.is_file() {
        //不是，返回错误
        return Err(NOT_FILE_ERROR);
    }
    Ok((reader, metadata))
}
```

以上实际上是文件类型结构及方法的应用起始点，相关的类型结构及方法如下：

```rust
//此类型结构主要用于设置文件open的选择项
//此类型结构对libc的open的属性参数做了总结，
//更友好的生成open的属性参数。否则，每次调用open都需要重新看man手册
//此类型结构可以处理一些文件打开时的矛盾选项，提升安全
pub struct OpenOptions {
    // 可读
    read: bool,
    //可写
    write: bool,
    //添加到尾部
    append: bool,
    //删除文件内容
    truncate: bool,
    //创建文件
    create: bool,
    //创建一个新的文件
    create_new: bool,
    //具体操作系统相关
    custom_flags: i32,
    //
    mode: mode_t,
}
```

```rust
impl OpenOptions {
    pub fn new() -> OpenOptions {
        //默认的属性
        OpenOptions {
            // generic
            read: false,
            write: false,
            append: false,
            truncate: false,
            create: false,
            create_new: false,
            // system-specific
            custom_flags: 0,
            //linux的文件权限
            mode: 0o666,
        }
    }

    //以下设置文件打开属性

    pub fn read(&mut self, read: bool) {
        self.read = read;
    }
    pub fn write(&mut self, write: bool) {
        self.write = write;
    }
    pub fn append(&mut self, append: bool) {
        self.append = append;
    }
    pub fn truncate(&mut self, truncate: bool) {
        self.truncate = truncate;
    }
    pub fn create(&mut self, create: bool) {
        self.create = create;
    }
    pub fn create_new(&mut self, create_new: bool) {
        self.create_new = create_new;
    }

    pub fn custom_flags(&mut self, flags: i32) {
        self.custom_flags = flags;
    }
    pub fn mode(&mut self, mode: u32) {
        self.mode = mode as mode_t;
    }

    //文件属性转化为libc的open函数中的文件模式参数读写位
    //可以看到，此函数将以前的经验做了总结。
    fn get_access_mode(&self) -> io::Result<c_int> {
        match (self.read, self.write, self.append) {
            (true, false, false) => Ok(libc::O_RDONLY),
```

```rust
            (false, true, false) => Ok(libc::O_WRONLY),
            (true, true, false) => Ok(libc::O_RDWR),
            (false, _, true) => Ok(libc::O_WRONLY | libc::O_APPEND),
            (true, _, true) => Ok(libc::O_RDWR | libc::O_APPEND),
            (false, false, false) =>
Err(Error::from_raw_os_error(libc::EINVAL)),
        }
    }

    //文件属性转化为libc的open函数中的文件模式参数创建位
    fn get_creation_mode(&self) -> io::Result<c_int> {
        //矛盾判断
        match (self.write, self.append) {
            (true, false) => {}
            //不允许写即不允许创建文件
            (false, false) => {
                if self.truncate || self.create || self.create_new {
                    return Err(Error::from_raw_os_error(libc::EINVAL));
                }
            }
            //与truncate矛盾
            (_, true) => {
                if self.truncate && !self.create_new {
                    return Err(Error::from_raw_os_error(libc::EINVAL));
                }
            }
        }

        Ok(match (self.create, self.truncate, self.create_new) {
            (false, false, false) => 0,
            //创建文件
            (true, false, false) => libc::O_CREAT,
            //原有文件内容清零
            (false, true, false) => libc::O_TRUNC,
            //没有文件就创建，文件存在则清零
            (true, true, false) => libc::O_CREAT | libc::O_TRUNC,
            //没有文件就创建，文件存在则返回失败
            (_, _, true) => libc::O_CREAT | libc::O_EXCL,
        })
    }
}
```

OpenOption在细节上体现了RUST的程序员友好，将原本libc::open函数中的属性参数的学习负担清除掉了。

```rust
// 操作系统无关界面接口File类型结构
pub struct File(FileDesc);

//创建文件的方法实现
impl File {
```

```rust
    //打开文件，创建新文件也用此函数
    pub fn open(path: &Path, opts: &OpenOptions) -> io::Result<File> {
        //linux中，需要把Path转换成C字符串
        let path = cstr(path)?;
        File::open_c(&path, opts)
    }

    //利用libc::open打开及创建文件
    pub fn open_c(path: &CStr, opts: &OpenOptions) -> io::Result<File> {
        //创建文件时最复杂的是flags的生成，
        //RUST利用OpenOptions比较直观的完成了这个工作
        let flags = libc::O_CLOEXEC
            | opts.get_access_mode()?
            | opts.get_creation_mode()?
            | (opts.custom_flags as c_int & !libc::O_ACCMODE);
        //不同的操作系统还是有些区别，但不必关注这个细节了
        let fd = cvt_r(|| unsafe { open64(path.as_ptr(), flags, opts.mode as
c_int) })?;
        //创建File变量，unsafe表明了fd的不安全的特性
        Ok(File(unsafe { FileDesc::from_raw_fd(fd) }))
    }

    ...
    ...
}

//路径名类型结构，必须用OsStr来实现
//此处没有repr(transparent)，但Path的内存布局与OsStr是一致的
pub struct Path {
    inner: OsStr,
}

impl Path {
    //能够转换为OsStr引用的类型都能够转换为Path的引用
    //Path的内存布局与OsStr是一致的。
    pub fn new<S: AsRef<OsStr> + ?Sized>(s: &S) -> &Path {
        unsafe { &*(s.as_ref() as *const OsStr as *const Path) }
    }
}

//Path一般用于引用，PathBuf拥有所有权
//本质上Path与PathBuf的关系就是OsStr与OsString的关系
pub struct PathBuf {
    inner: OsString,
}

impl PathBuf {
    pub fn new() -> PathBuf {
        PathBuf { inner: OsString::new() }
    }
    ...
```

```
    ...
}
```

以上是RUST中文件类型结构典型的创建的过程。 下面是文件拷贝函数:

```
//拷贝文件
pub fn copy(from: &Path, to: &Path) -> io::Result<u64> {
    let (mut reader, reader_metadata) = open_from(from)?;
    let max_len = u64::MAX;
    //注意这个文件属性的传递
    let (mut writer, _) = open_to_and_set_permissions(to, reader_metadata)?;

    use super::kernel_copy::{copy_regular_files, CopyResult};

    match copy_regular_files(reader.as_raw_fd(), writer.as_raw_fd(), max_len) {
        CopyResult::Ended(bytes) => Ok(bytes),
        CopyResult::Error(e, _) => Err(e),
        CopyResult::Fallback(written) => match io::copy::generic_copy(&mut
reader, &mut writer) {
            Ok(bytes) => Ok(bytes + written),
            Err(e) => Err(e),
        },
    }
}
```

其他RUST的文件操作，操作系统相关模块提供的对外接口, 基本都是直接调用libc的同名函数，解释略:

```
//删除文件及连接
pub fn unlink(p: &Path) -> io::Result<()> {
    let p = cstr(p)?;
    cvt(unsafe { libc::unlink(p.as_ptr()) })?;
    Ok(())
}

pub fn rename(old: &Path, new: &Path) -> io::Result<()> {
    let old = cstr(old)?;
    let new = cstr(new)?;
    cvt(unsafe { libc::rename(old.as_ptr(), new.as_ptr()) })?;
    Ok(())
}

pub fn symlink(original: &Path, link: &Path) -> io::Result<()> {
    let original = cstr(original)?;
    let link = cstr(link)?;
    cvt(unsafe { libc::symlink(original.as_ptr(), link.as_ptr()) })?;
```

```
    Ok(())
}

pub fn chown(path: &Path, uid: u32, gid: u32) -> io::Result<()> {
    let path = cstr(path)?;
    cvt(unsafe { libc::chown(path.as_ptr(), uid as libc::uid_t, gid as
libc::gid_t) })?;
    Ok(())
}

pub fn fchown(fd: c_int, uid: u32, gid: u32) -> io::Result<()> {
    cvt(unsafe { libc::fchown(fd, uid as libc::uid_t, gid as libc::gid_t) })?;
    Ok(())
}

pub fn lchown(path: &Path, uid: u32, gid: u32) -> io::Result<()> {
    let path = cstr(path)?;
    cvt(unsafe { libc::lchown(path.as_ptr(), uid as libc::uid_t, gid as
libc::gid_t) })?;
    Ok(())
}

pub fn chroot(dir: &Path) -> io::Result<()> {
    let dir = cstr(dir)?;
    cvt(unsafe { libc::chroot(dir.as_ptr()) })?;
    Ok(())
}

//创建一个文件链接
pub fn link(original: &Path, link: &Path) -> io::Result<()> {
    let original = cstr(original)?;
    let link = cstr(link)?;
        {
            // Where we can, use `linkat` instead of `link`; see the comment
above
            // this one for details on why.
            cvt(unsafe { libc::linkat(libc::AT_FDCWD, original.as_ptr(),
libc::AT_FDCWD, link.as_ptr(), 0) })?;
        }
    Ok(())
}
```

以下接口函数需要对libc的函数做些适配工作

```
//设置文件权限， FilePermission见下面结构
pub fn set_perm(p: &Path, perm: FilePermissions) -> io::Result<()> {
    let p = cstr(p)?;
    cvt_r(|| unsafe { libc::chmod(p.as_ptr(), perm.mode) })?;
    Ok(())
}
```

```rust
//linux的文件权限
pub struct FilePermissions {
    mode: mode_t,
}


//获取文件属性，FileAttr见后面的代码分析
pub fn stat(p: &Path) -> io::Result<FileAttr> {
    let p = cstr(p)?;

        //try_statx将后面的分析
        if let Some(ret) = unsafe { try_statx(
            libc::AT_FDCWD,
            p.as_ptr(),
            libc::AT_STATX_SYNC_AS_STAT,
            libc::STATX_ALL,
        ) } {
            //如果成功，已经用statx方式获取
            //返回
            return ret;
        }

    //否则，用stat64方式获取属性
    let mut stat: stat64 = unsafe { mem::zeroed() };
    cvt(unsafe { stat64(p.as_ptr(), &mut stat) })?;
    Ok(FileAttr::from_stat64(stat))
}


//相关的
pub struct FileType {
    mode: mode_t,
}

pub struct FileAttr {
    stat: stat64,
    statx_extra_fields: Option<StatxExtraFields>,
}

impl FileAttr {
    fn from_stat64(stat: stat64) -> Self {
        Self { stat, statx_extra_fields: None }
    }

    pub fn size(&self) -> u64 {
        self.stat.st_size as u64
    }
    pub fn perm(&self) -> FilePermissions {
        FilePermissions { mode: (self.stat.st_mode as mode_t) }
    }
```

```rust
    pub fn file_type(&self) -> FileType {
        FileType { mode: self.stat.st_mode as mode_t }
    }
}

impl FileAttr {
    //修改时间
    pub fn modified(&self) -> io::Result<SystemTime> {
        Ok(SystemTime::from(libc::timespec {
            tv_sec: self.stat.st_mtime as libc::time_t,
            tv_nsec: self.stat.st_mtime_nsec as _,
        }))
    }

    //创建时间
    pub fn created(&self) -> io::Result<SystemTime> {
            if let Some(ext) = &self.statx_extra_fields {
                return if (ext.stx_mask & libc::STATX_BTIME) != 0 {
                    Ok(SystemTime::from(libc::timespec {
                        tv_sec: ext.stx_btime.tv_sec as libc::time_t,
                        tv_nsec: ext.stx_btime.tv_nsec as _,
                    }))
                } else {
                    Err(io::const_io_error!(
                        io::ErrorKind::Uncategorized,
                        "creation time is not available for the filesystem",
                    ))
                };
            }

        Err(io::const_io_error!(
            io::ErrorKind::Unsupported,
            "creation time is not available on this platform \
                            currently",
        ))
    }
}


struct StatxExtraFields {
    stx_mask: u32,
    stx_btime: libc::statx_timestamp,
}

//linux上，statx包含了最全面的信息
unsafe fn try_statx(
    fd: c_int,
    path: *const c_char,
    flags: i32,
    mask: u32,
```

```rust
) -> Option<io::Result<FileAttr>> {
    use crate::sync::atomic::{AtomicU8, Ordering};

    syscall! {
        fn statx(
            fd: c_int,
            pathname: *const c_char,
            flags: c_int,
            mask: libc::c_uint,
            statxbuf: *mut libc::statx
        ) -> c_int
    }

    let mut buf: libc::statx = mem::zeroed();
    if let Err(err) = cvt(statx(fd, path, flags, mask, &mut buf)) {
        return Some(Err(err));
    }

    // 需要用stat64返回，以下从stat翻译到stat64.
    let mut stat: stat64 = mem::zeroed();
    // `c_ulong` on gnu-mips, `dev_t` otherwise
    stat.st_dev = libc::makedev(buf.stx_dev_major, buf.stx_dev_minor) as _;
    stat.st_ino = buf.stx_ino as libc::ino64_t;
    stat.st_nlink = buf.stx_nlink as libc::nlink_t;
    stat.st_mode = buf.stx_mode as libc::mode_t;
    stat.st_uid = buf.stx_uid as libc::uid_t;
    stat.st_gid = buf.stx_gid as libc::gid_t;
    stat.st_rdev = libc::makedev(buf.stx_rdev_major, buf.stx_rdev_minor) as _;
    stat.st_size = buf.stx_size as off64_t;
    stat.st_blksize = buf.stx_blksize as libc::blksize_t;
    stat.st_blocks = buf.stx_blocks as libc::blkcnt64_t;
    stat.st_atime = buf.stx_atime.tv_sec as libc::time_t;
    // `i64` on gnu-x86_64-x32, `c_ulong` otherwise.
    stat.st_atime_nsec = buf.stx_atime.tv_nsec as _;
    stat.st_mtime = buf.stx_mtime.tv_sec as libc::time_t;
    stat.st_mtime_nsec = buf.stx_mtime.tv_nsec as _;
    stat.st_ctime = buf.stx_ctime.tv_sec as libc::time_t;
    stat.st_ctime_nsec = buf.stx_ctime.tv_nsec as _;

    let extra = StatxExtraFields {
        stx_mask: buf.stx_mask,
        stx_btime: buf.stx_btime,
    };

    Some(Ok(FileAttr { stat, statx_extra_fields: Some(extra) }))
}


//link的属性获取,与stat类似
pub fn lstat(p: &Path) -> io::Result<FileAttr> {
    let p = cstr(p)?;
```

```rust
        if let Some(ret) = unsafe { try_statx(
            libc::AT_FDCWD,
            p.as_ptr(),
            libc::AT_SYMLINK_NOFOLLOW | libc::AT_STATX_SYNC_AS_STAT,
            libc::STATX_ALL,
        ) } {
            return ret;
        }

    let mut stat: stat64 = unsafe { mem::zeroed() };
    cvt(unsafe { lstat64(p.as_ptr(), &mut stat) })?;
    Ok(FileAttr::from_stat64(stat))
}


//相关的类型结构的方法实现
impl FilePermissions {
    pub fn readonly(&self) -> bool {
        // check if any class (owner, group, others) has write permission
        self.mode & 0o222 == 0
    }

    pub fn set_readonly(&mut self, readonly: bool) {
        if readonly {
            // remove write permission for all classes; equivalent to `chmod a-
w <file>`
            self.mode &= !0o222;
        } else {
            // add write permission for all classes; equivalent to `chmod a+w
<file>`
            self.mode |= 0o222;
        }
    }
    pub fn mode(&self) -> u32 {
        self.mode as u32
    }
}

impl FileType {
    pub fn is_dir(&self) -> bool {
        self.is(libc::S_IFDIR)
    }
    pub fn is_file(&self) -> bool {
        self.is(libc::S_IFREG)
    }
    pub fn is_symlink(&self) -> bool {
        self.is(libc::S_IFLNK)
    }

    pub fn is(&self, mode: mode_t) -> bool {
```

```
        self.mode & libc::S_IFMT == mode
    }
}
```

以下两个函数涉及到了从外部C函数传入的字符串与RUST字符串的转换，值得仔细学习。

```
//读取链接的path
pub fn readlink(p: &Path) -> io::Result<PathBuf> {
    let c_path = cstr(p)?;
    let p = c_path.as_ptr();

    //因为需要用C语言的字符串存放读回的内容，
    //所以用Vec来申请内存
    let mut buf = Vec::with_capacity(256);

    loop {
        //读到buf里，限制了读的长度
        let buf_read =
            cvt(unsafe { libc::readlink(p, buf.as_mut_ptr() as *mut _,
buf.capacity()) })? as usize;

        //读成功，设置Vec，使得Vec正确反映读的内容
        //此处是RUST与C交互的额外的设置内容，很易出错
        unsafe {
            //直接用set_len完成Vec的len初始化
            buf.set_len(buf_read);
        }

        //将Vec转化为OsString
        if buf_read != buf.capacity() {
            //不能有额外的容量
            buf.shrink_to_fit();

            //创建PathBuf并返回
            return Ok(PathBuf::from(OsString::from_vec(buf)));
        }

        // 如果正好是vec的容量，证明link的内容可能长过容量，
        // reserve(1)后再次读
        buf.reserve(1);
    }
}

//返回绝对路径
pub fn canonicalize(p: &Path) -> io::Result<PathBuf> {
    //这里需要自行申请内存，防止不安全
    let path = CString::new(p.as_os_str().as_bytes())?;
    let buf;
    unsafe {
```

```
            //返回绝对路径
            let r = libc::realpath(path.as_ptr(), ptr::null_mut());
            if r.is_null() {
                return Err(io::Error::last_os_error());
            }
            //将返回的C字符串用以生成Vec
            buf = CStr::from_ptr(r).to_bytes().to_vec();
            //负责释放
            libc::free(r as *mut _);
        }
        //生成PathBuf
        Ok(PathBuf::from(OsString::from_vec(buf)))
    }
```

文件的其他属性：

```
impl File {
    //文件属性获取
    pub fn file_attr(&self) -> io::Result<FileAttr> {
        let fd = self.as_raw_fd();

            if let Some(ret) = unsafe { try_statx(
                fd,
                b"\0" as *const _ as *const c_char,
                libc::AT_EMPTY_PATH | libc::AT_STATX_SYNC_AS_STAT,
                libc::STATX_ALL,
            ) } {
                return ret;
            }

        let mut stat: stat64 = unsafe { mem::zeroed() };
        cvt(unsafe { fstat64(fd, &mut stat) })?;
        Ok(FileAttr::from_stat64(stat))
    }

    //完成文件内存与磁盘同步
    pub fn fsync(&self) -> io::Result<()> {
        cvt_r(|| unsafe { os_fsync(self.as_raw_fd()) })?;
        return Ok(());

        unsafe fn os_fsync(fd: c_int) -> c_int {
            libc::fsync(fd)
        }
    }

    //仅完成文件的数据与磁盘同步，不包括文件属性
    pub fn datasync(&self) -> io::Result<()> {
        cvt_r(|| unsafe { os_datasync(self.as_raw_fd()) })?;
        return Ok(());
```

```
        unsafe fn os_datasync(fd: c_int) -> c_int {
            libc::fdatasync(fd)
        }
    }

    //删除文件内容
    pub fn truncate(&self, size: u64) -> io::Result<()> {
        use crate::convert::TryInto;
        let size: off64_t =
            size.try_into().map_err(|e|
io::Error::new(io::ErrorKind::InvalidInput, e))?;
        cvt_r(|| unsafe { ftruncate64(self.as_raw_fd(), size) }).map(drop)
    }

    //复制fd，并形成新的File
    pub fn duplicate(&self) -> io::Result<File> {
        self.0.duplicate().map(File)
    }

    //设置文件权限
    pub fn set_permissions(&self, perm: FilePermissions) -> io::Result<()> {
        cvt_r(|| unsafe { libc::fchmod(self.as_raw_fd(), perm.mode) })?;
        Ok(())
    }
}

//用于创建目录
pub struct DirBuilder {
    mode: mode_t,
}


impl DirBuilder {
    pub fn new() -> DirBuilder {
        DirBuilder { mode: 0o777 }
    }

    //创建一个目录
    pub fn mkdir(&self, p: &Path) -> io::Result<()> {
        let p = cstr(p)?;
        cvt(unsafe { libc::mkdir(p.as_ptr(), self.mode) })?;
        Ok(())
    }

    //设置创建目录的权限
    pub fn set_mode(&mut self, mode: u32) {
        self.mode = mode as mode_t;
    }
}
```

操作系统无关文件系统模块分析

引入linux的fs类型结构及实现

```
use crate::sys::fs as fs_imp;
```

RUST标准库对外接口的类型结构:

```
//及linux文件系统中File的封装
pub struct File {
    inner: fs_imp::File,
}

//文件元数据，即FileAttr的封装
pub struct Metadata(fs_imp::FileAttr);

//打开的目录类型结构，即linux的fs同名结构封装
pub struct ReadDir(fs_imp::ReadDir);

//目录中的项目类型结构，也即简单封装
pub struct DirEntry(fs_imp::DirEntry);

//创建/打开文件的执行者类型结构，也即简单封装
pub struct OpenOptions(fs_imp::OpenOptions);

//文件权限
pub struct Permissions(fs_imp::FilePermissions);

//文件类型
pub struct FileType(fs_imp::FileType);

//目录创建执行类型结构
pub struct DirBuilder {
    inner: fs_imp::DirBuilder,
    //是否为多级目录
    recursive: bool,
}
```

相关的与文件读写无关的实现:

```
impl File {
    //只读文件打开，RUST的文件打开，打开模式的输入，
    //用不同的函数表示不同的打开方式
    pub fn open<P: AsRef<Path>>(path: P) -> io::Result<File> {
        //见后继的OpenOptions的分析
        OpenOptions::new().read(true).open(path.as_ref())
    }
```

```rust
    //创建一个文件
    pub fn create<P: AsRef<Path>>(path: P) -> io::Result<File> {
        //文件设置为可写，文件存在则删除内容，文件不在就创建

OpenOptions::new().write(true).create(true).truncate(true).open(path.as_ref())
    }

    //创建一个文件打开选项
    pub fn options() -> OpenOptions {
        OpenOptions::new()
    }

    //同步文件到磁盘
    pub fn sync_all(&self) -> io::Result<()> {
        self.inner.fsync()
    }

    //只同步文件数据到磁盘
    pub fn sync_data(&self) -> io::Result<()> {
        self.inner.datasync()
    }

    //设置文件为指定大小
    pub fn set_len(&self, size: u64) -> io::Result<()> {
        self.inner.truncate(size)
    }

    //获取文件属性
    pub fn metadata(&self) -> io::Result<Metadata> {
        self.inner.file_attr().map(Metadata)
    }

    //复制文件描述符，生成新的File变量
    pub fn try_clone(&self) -> io::Result<File> {
        Ok(File { inner: self.inner.duplicate()? })
    }

    //设置文件权限
    pub fn set_permissions(&self, perm: Permissions) -> io::Result<()> {
        self.inner.set_permissions(perm.0)
    }
}

//文件创建/打开的执行类型结构
impl OpenOptions {
    //对操作系统相关的同名结构的Adapter
    pub fn new() -> Self {
        OpenOptions(fs_imp::OpenOptions::new())
    }
```

```rust
    pub fn read(&mut self, read: bool) -> &mut Self {
        self.0.read(read);
        self
    }

    pub fn write(&mut self, write: bool) -> &mut Self {
        self.0.write(write);
        self
    }

    pub fn append(&mut self, append: bool) -> &mut Self {
        self.0.append(append);
        self
    }

    pub fn truncate(&mut self, truncate: bool) -> &mut Self {
        self.0.truncate(truncate);
        self
    }

    pub fn create(&mut self, create: bool) -> &mut Self {
        self.0.create(create);
        self
    }

    pub fn create_new(&mut self, create_new: bool) -> &mut Self {
        self.0.create_new(create_new);
        self
    }

    //利用一个类型结构完成打开文件的各种选项，比用一个参数表达更清晰
    //易掌握
    pub fn open<P: AsRef<Path>>(&self, path: P) -> io::Result<File> {
        self._open(path.as_ref())
    }

    fn _open(&self, path: &Path) -> io::Result<File> {
        fs_imp::File::open(path, &self.0).map(|inner| File { inner })
    }
}

// ReadDir适配设计模式
impl Iterator for ReadDir {
    type Item = io::Result<DirEntry>;

    fn next(&mut self) -> Option<io::Result<DirEntry>> {
        self.0.next().map(|entry| entry.map(DirEntry))
    }
}
```

```rust
impl DirBuilder {
    pub fn new() -> DirBuilder {
        DirBuilder { inner: fs_imp::DirBuilder::new(), recursive: false }
    }

    pub fn recursive(&mut self, recursive: bool) -> &mut Self {
        self.recursive = recursive;
        self
    }

    //创建一个目录
    pub fn create<P: AsRef<Path>>(&self, path: P) -> io::Result<()> {
        self._create(path.as_ref())
    }

    fn _create(&self, path: &Path) -> io::Result<()> {
        //如果是多级，则进入下一级，否则创建新目录
        if self.recursive { self.create_dir_all(path) } else {
self.inner.mkdir(path) }
    }

    //创建多级目录
    fn create_dir_all(&self, path: &Path) -> io::Result<()> {
        if path == Path::new("") {
            return Ok(());
        }

        match self.inner.mkdir(path) {
            Ok(()) => return Ok(()),
            Err(ref e) if e.kind() == io::ErrorKind::NotFound => {}
            Err(_) if path.is_dir() => return Ok(()),
            Err(e) => return Err(e),
        }
        match path.parent() {
            Some(p) => self.create_dir_all(p)?,
            None => {
                return Err(io::const_io_error!(
                    io::ErrorKind::Uncategorized,
                    "failed to create whole tree",
                ));
            }
        }
        match self.inner.mkdir(path) {
            Ok(()) => Ok(()),
            Err(_) if path.is_dir() => Ok(()),
            Err(e) => Err(e),
        }
    }
}
```

RUST标准库对外文件操作函数：

```rust
//删除文件
pub fn remove_file<P: AsRef<Path>>(path: P) -> io::Result<()> {
    fs_imp::unlink(path.as_ref())
}

//获取文件属性
pub fn metadata<P: AsRef<Path>>(path: P) -> io::Result<Metadata> {
    fs_imp::stat(path.as_ref()).map(Metadata)
}

//链接属性
pub fn symlink_metadata<P: AsRef<Path>>(path: P) -> io::Result<Metadata> {
    fs_imp::lstat(path.as_ref()).map(Metadata)
}

//重命名
pub fn rename<P: AsRef<Path>, Q: AsRef<Path>>(from: P, to: Q) -> io::Result<()>
{
    fs_imp::rename(from.as_ref(), to.as_ref())
}

//创建硬链接
pub fn hard_link<P: AsRef<Path>, Q: AsRef<Path>>(original: P, link: Q) ->
io::Result<()> {
    fs_imp::link(original.as_ref(), link.as_ref())
}

//创建软链接
pub fn soft_link<P: AsRef<Path>, Q: AsRef<Path>>(original: P, link: Q) ->
io::Result<()> {
    fs_imp::symlink(original.as_ref(), link.as_ref())
}

//读取链接内容
pub fn read_link<P: AsRef<Path>>(path: P) -> io::Result<PathBuf> {
    fs_imp::readlink(path.as_ref())
}

//生成绝对路径
pub fn canonicalize<P: AsRef<Path>>(path: P) -> io::Result<PathBuf> {
    fs_imp::canonicalize(path.as_ref())
}

//创建一个目录
pub fn create_dir<P: AsRef<Path>>(path: P) -> io::Result<()> {
    DirBuilder::new().create(path.as_ref())
}
```

```rust
//创建多级目录
pub fn create_dir_all<P: AsRef<Path>>(path: P) -> io::Result<()> {
    DirBuilder::new().recursive(true).create(path.as_ref())
}

//删除空目录
pub fn remove_dir<P: AsRef<Path>>(path: P) -> io::Result<()> {
    fs_imp::rmdir(path.as_ref())
}

//删除整个目录
pub fn remove_dir_all<P: AsRef<Path>>(path: P) -> io::Result<()> {
    fs_imp::remove_dir_all(path.as_ref())
}

//打开目录，准备用Iterator的方式读
pub fn read_dir<P: AsRef<Path>>(path: P) -> io::Result<ReadDir> {
    fs_imp::readdir(path.as_ref()).map(ReadDir)
}

//设置文件权限
pub fn set_permissions<P: AsRef<Path>>(path: P, perm: Permissions) ->
io::Result<()> {
    fs_imp::set_perm(path.as_ref(), perm.0)
}
```