rCore_tutorial 文件系统代码分析

整体架构

设备驱动层 Device Driver

抽象设备层 Device

? 块缓存层 Block Cache

虚拟文件系统层 Virtual File System （其具体实现为 Simple File System）

文件描述符层 File Descriptor

系统调用层 System Call

rcore-fs crate

一共包含如下子模块：

- *util*，只包含一个迭代器 `BlockIter`，可以返回一个 `BlockRange` 序列。
- *file*，类似于 xv6 文件描述符层的 `File` 结构体，表示进程里面打开的文件的信息，但是比较简单里面含有对应的 `INode` 的指针，调用 `INode` 的接口
- *vfs*，包含有以下内容

  首先是 `INode` 接口，我们必须实现以下接口：

  ```
  /// Read bytes at `offset` into `buf`, return the number of bytes read.
  fn read_at(&self, offset: usize, buf: &mut [u8]) -> Result<usize>;
  /// Write bytes at `offset` from `buf`, return the number of bytes written.
  fn write_at(&self, offset: usize, buf: &[u8]) -> Result<usize>;
  /// Poll the events, return a bitmap of events.
  fn poll(&self) -> Result<PollStatus>;

  #[derive(Debug, Default)]
  pub struct PollStatus {
      pub read: bool,
      pub write: bool,
      pub error: bool,
  }
  ```

  注意到 `INode` 继承了 `core::any::Any` 接口，于是 `INode` 指针可以被 `as &dyn Any`，并动态转化为其他类型的指针。注意 `as_any_ref` 函数。

  里面提供了 `create, create2` 两函数的实现，不过目前看上去有些诡异。

  接下来，为 `INode trait object` 实现了下列方法：

  ```
  /// Downcast the INode to specific struct
  pub fn downcast_ref<T: INode>(&self) -> Option<&T>;

  /// Get all directory entries as a Vec
  ```

```rust
pub fn list(&self) -> Result<Vec<String>>;
/// 注意，该实现用到了 INode 的 get_entry 方法
/// Get the name of directory entry
/// fn get_entry(&self, _id: usize) -> Result<String>;
/// 这里的 _id 指的是父目录的第几个 dirent
/// 不过原版 INode 尚未实现

/// Lookup path from current INode, and do not follow symlinks
/// 从当前 INode 开始 traverse ，无视符号链接，返回最终得到的 INode trait object 指针 (Arc)
pub fn lookup(&self, path: &str) -> Result<Arc<dyn INode>>;
/// 它是使用下面的函数实现的

/// Lookup path from current INode, and follow symlinks at most `follow_times` times
pub fn lookup_follow(&self, path: &str, mut follow_times: usize) -> Result<Arc<dyn INode>>;
/// 这里需要注意的是：

/// 在处理绝对路径时，即 139 行， result = self.fs().root_inode();
/// INode 并未实现 fs ，而是留给具体实现
/// Get the file system of the INode
/// fn fs(&self) -> Arc<dyn FileSystem> { unimplemented!(); }

/// 从 156 行开始的对于符号链接的处理
/// 可见，对于符号链接而言，内容只有 256 字节，表示它链接到的位置
```

后面还给出了 `INode` 的元数据描述结构体 `Metadata` ，值得一提的是要弄懂 `nlinks` 的含义。

文件类型 `FileType` ，但其中应该有很多没用到就是了。

```rust
#[derive(Copy, Clone, Debug, Eq, PartialEq)]
pub enum FileType {
    File,
    Dir,
    SymLink,
    CharDevice,
    BlockDevice,
    NamedPipe,
    Socket,
}
```

如果真的能把 `Socket` 也归类进去那就太强了。

文件系统的元数据描述结构体 `FsInfo` 。

文件操作的错误类型 `FsError` 。

值得一提的是，这里定义了类型

```rust
pub type Result<T> = result::Result<T, FsError>;
```

最终给出了文件系统原型，都是一些很好理解的功能

```rust
/// Abstract file system
pub trait FileSystem: Sync + Send {
    /// Sync all data to the storage
    fn sync(&self) -> Result<()>;
    /// Get the root INode of the file system
    fn root_inode(&self) -> Arc<dyn INode>;
    /// Get the file system information
    fn info(&self) -> FsInfo;
}
```

- *dirty* ，实现了一个 wrapper ，将类型 `T` 包裹起来，在其被回收时确保里面的内容已被写回。
- *dev* ，首先提供了若干接口：

```rust
/// mod.fs

/// A current time provider
/// 返回当前的时间
pub trait TimeProvider: Send + Sync {
    fn current_time(&self) -> Timespec;
}
/// Interface for FS to read & write
/// 基于 offset 进行 I/O 的设备原型
pub trait Device: Send + Sync {
    fn read_at(&self, offset: usize, buf: &mut [u8]) -> Result<usize>;
    fn write_at(&self, offset: usize, buf: &[u8]) -> Result<usize>;
    fn sync(&self) -> Result<()>;
}
/// Device which can only R/W in blocks
/// 以块为单位进行 I/O 的设备原型
pub trait BlockDevice: Send + Sync {
    const BLOCK_SIZE_LOG2: u8;
    fn read_at(&self, block_id: BlockId, buf: &mut [u8]) -> Result<()>;
    fn write_at(&self, block_id: BlockId, buf: &[u8]) -> Result<()>;
    fn sync(&self) -> Result<()>;
}
```

后面，使用 `BlockDevice` 提供的接口实现了 `Device` 所要求的接口，可见相比 `BlockDevice` ，`Device` 是更为通用的一种设备，需要基于 offset 进行 I/O 。

在 `block_cache.rs` 中，为块设备 `BlockDevice` 实现了一个块缓存层。

核心数据结构如下：

```rust
/// 块缓存 BlockCache
pub struct BlockCache<T: BlockDevice> {
    device: T, /// 块设备
    bufs: Vec<Mutex<Buf>>, /// Buf 数组
    lru: Mutex<LRU>, /// 缓存管理器 LRU
}
struct Buf {
    status: BufStatus,
    data: Vec<u8>,
}
enum BufStatus {
    /// buffer is unused
    Unused,
```

```
    /// buffer has been read from disk
    Valid(BlockId),
    /// buffer needs to be written to disk
    Dirty(BlockId),
}
```

`BlockCache` 对外提供的接口是：

```
/// 给定块设备，设定缓存中容纳多少块，初始化
pub fn new(device: T, capacity: usize) -> Self;

/// Get a buffer for `block_id` with any status
/// 这里的 MutexGuard 基于 RAII ，即在创建时上锁，等到该结构被释放时才解锁，进行保护
/// 因此，语义为，给定扇区号，返回一个已经被上了锁的 Buf ，里面含有从块设备中读到的数据，以及修改状态
fn get_buf(&self, block_id: BlockId) -> MutexGuard<Buf>;
/// 实现中用到了 _get_buf, get_unused 两个函数
/// 先查找该块是否已经在块缓存中，如果存在直接返回；否则使用 LRU 找一个牺牲品，将修改写回并进行替换
/// 从而，返回的 Buf 里面的数据未必与块设备同步

/// Write back data if buffer is dirty
/// 将一个 Buf 中的数据写回块设备则使用 write_back 函数
fn write_back(&self, buf: &mut Buf) -> Result<()> {
    if let BufStatus::Dirty(block_id) = buf.status {
        self.device.write_at(block_id, &buf.data)?;
        buf.status = BufStatus::Valid(block_id);
    }
    Ok(())
}
```

当 `BlockCache` 被回收时，也会进行写回：

```
impl<T: BlockDevice> Drop for BlockCache<T> {
    fn drop(&mut self) {
        // 将块设备整体进行写回
        BlockDevice::sync(self).expect("failed to sync");
    }
}
```

同时，为 `BlockCache` 实现了块设备的接口试图将其作为一个块设备使用。

以 `read_at` 为例分析下层接口的调用范式：

```
fn read_at(&self, block_id: BlockId, buffer: &mut [u8]) -> Result<()> {
    // 获取 MutexGuard<Buf> 类型
    let mut buf = self.get_buf(block_id);
    match buf.status {
        // 如果还未从块设备上读取过，读取内容并修改状态
        // 读取到 buf 中
        BufStatus::Unused => {
            // read from device
            self.device.read_at(block_id, &mut buf.data)?;
            buf.status = BufStatus::Valid(block_id);
        }
        _ => {}
```

```
        }
        let len = 1 << Self::BLOCK_SIZE_LOG2 as usize;
        buffer[..len].copy_from_slice(&buf.data);
        Ok(())
        // 在 buf 被回收时解锁
    }
```

LRU 是使用链表实现的。

注意迭代器的下述用法

```
prev: (size - 1..size).chain(0..size - 1).collect(),
```

在 `std_impl.rs` 中为 `Mutex<File>` 实现了 `Device` 接口，还实现了 `StdTimeProvider` 。它们使用 `std crate` 实现，想必是用来生成 Simple File System 镜像。

rcore-fs-sfs crate

包含了 `rcore-fs` 中各接口的具体实现。主要是描述磁盘布局以及如何实现。

这里的 sfs 实现是继承 ucore 的，因此可以从那里来获得帮助。

首先是在磁盘上的数据结构超级块 `SuperBlock` 和索引节点 `DiskINode` ：

```
/// On-disk superblock
#[repr(C)]
#[derive(Debug)]
pub struct SuperBlock {
    /// magic number, should be SFS_MAGIC
    pub magic: u32,
    /// number of blocks in fs
    pub blocks: u32,
    /// number of unused blocks in fs
    pub unused_blocks: u32,
    /// information for sfs
    pub info: Str32,
    /// number of freemap blocks
    pub freemap_blocks: u32,
}

/// inode (on disk)
#[repr(C)]
#[derive(Debug)]
pub struct DiskINode {
    /// size of the file (in bytes)
    /// undefined in dir (256 * #entries ?)
    /// 文件字节数
    pub size: u32,

    /// one of SYS_TYPE_* above
    /// 文件类型
    pub type_: FileType,

    /// number of hard links to this file
    /// Note: "." and ".." is counted in this nlinks
    /// 原来 nlink 是指该文件的硬链接数
```

```
    pub nlinks: u16,

    /// 为了拓展单个文件的大小
    /// number of blocks
    pub blocks: u32,
    /// direct blocks
    pub direct: [u32; NDIRECT],
    /// indirect blocks
    pub indirect: u32,
    /// double indirect blocks
    pub db_indirect: u32,

    /// device inode id for char/block device (major, minor)
    ///?
    pub device_inode_id: usize,
}
```

这里将 `INode trait object` 重命名为 `DeviceINode`。

```
pub type DeviceINode = dyn vfs::INode;
```

此外，

```
/// file entry (on disk)
/// 给出了 dirent 的描述
#[repr(C)]
#[derive(Debug)]
pub struct DiskEntry {
    /// inode number
    pub id: u32,
    /// file name
    pub name: Str256,
}
/// 两种定长字符串的描述
#[repr(C)]
pub struct Str256(pub [u8; 256]);
#[repr(C)]
pub struct Str32(pub [u8; 32]);
/// 后面提供了 &Str256, &Str32 与 &str 互相转化的函数
```

之后，为 `DiskINode` 实现了若干函数：

```
pub const fn new_file() -> Self;
// 新建一个文件型 on-disk INode
pub const fn new_symlink() -> Self;
// 新建一个符号链接型 on-disk INode
pub const fn new_dir() -> Self;
// 新建一个目录型 on-disk INode
pub const fn new_chardevice(device_inode_id: usize) -> Self;
// 新建一个字符设备型 on-disk INode
```

通过 `AsBuf` 接口，可以将上述提到的 `SuperBlock, DiskINode, DiskEntry, u32` 转为 `&[u8]`。

定义了一系列类型与常量：

`BlockId, INodeId` 均为 `usize`。

```rust
/// SFS 信息
pub const MAGIC: u32 = 0x2f8dbe2b; /// magic number for sfs
pub const DEFAULT_INFO: &str = "simple file system"; /// default sfs infomation
string
/// 块大小
pub const BLKSIZE: usize = 1usize << BLKSIZE_LOG2; /// size of block，每块 4KiB
pub const BLKSIZE_LOG2: u8 = 12; /// log2( size of block )
pub const BLKBITS: usize = BLKSIZE * 8; /// number of bits in a block
pub const ENTRY_SIZE: usize = 4; /// size of one entry，指单个扇区号大小为 4 字节
pub const BLK_NENTRY: usize = BLKSIZE / ENTRY_SIZE; /// number of entries in a
block
pub const DIRENT_SIZE: usize = MAX_FNAME_LEN + 1 + ENTRY_SIZE; /// size of a
dirent used in the size field
/// 基于 Indirect Block 的文件大小
pub const NDIRECT: usize = 12; /// number of direct blocks in inode
pub const MAX_NBLOCK_DIRECT: usize = NDIRECT; /// max number of blocks with
direct blocks
pub const MAX_NBLOCK_INDIRECT: usize = NDIRECT + BLK_NENTRY; /// max number of
blocks with indirect blocks
pub const MAX_NBLOCK_DOUBLE_INDIRECT: usize = NDIRECT + BLK_NENTRY + BLK_NENTRY
* BLK_NENTRY; /// max number of blocks with double indirect blocks
/// 文件限制
pub const MAX_INFO_LEN: usize = 31; /// max length of infomation
pub const MAX_FNAME_LEN: usize = 255; /// max length of filename
pub const MAX_FILE_SIZE: usize = 0xffffffff; /// max file size in theory (48KB +
4MB + 4GB)
/// however, the file size is stored in u32
/// 磁盘布局，可以看到块 0 为超级块；块 1 为根 INode 块；块 2 开始为 freemap 区域
pub const BLKN_SUPER: BlockId = 0; /// block the superblock lives in
pub const BLKN_ROOT: BlockId = 1; /// location of the root dir inode
pub const BLKN_FREEMAP: BlockId = 2; /// 1st block of the freemap
```

后面又提供了一个文件类型？

```rust
/// file types
#[repr(u16)]
#[derive(Debug, Eq, PartialEq, Copy, Clone)]
pub enum FileType {
    Invalid = 0,
    File = 1,
    Dir = 2,
    SymLink = 3,
    CharDevice = 4,
    BlockDevice = 5,
}
```

注意到 `rcore-fs crate` 中也有一个 `FileType` ，实际使用的到底是哪个需要看情况。

事实上是 `FileType` 与 `vfs::FileType` 之间的区别。

其具体实现就都放在 `lib.fs` 中了。明天再看吧。

首先，实现了一个接口 `DeviceExt` ，给基于 offset 进行 I/O 的 `Device` 提供了基于块 I/O 的能力。

它实现了下面几个函数：

```rust
fn read_block(&self, id: BlockId, offset: usize, buf: &mut [u8]) ->
vfs::Result<()>;
fn write_block(&self, id: BlockId, offset: usize, buf: &[u8]) ->
vfs::Result<()>;
fn load_struct<T: AsBuf>(&self, id: BlockId) -> vfs::Result<T>;
// 这里的 offset 指的是块内偏移
impl DeviceExt for dyn Device {}
// 为 Device trait object 实现 DeviceExt
```

接下来，是 `INode` 的具体实现 `INodeImpl` 。

```rust
/// INode for SFS
pub struct INodeImpl {
    id: INodeId, /// INode number
    disk_inode: RwLock<Dirty<DiskINode>>, /// On-disk INode
    fs: Arc<SimpleFileSystem>, /// Reference to SFS, used by almost all
operations
    /// Char/block device id (major, minor)
    /// e.g. crw-rw-rw- 1 root wheel 3, 2 May 13 16:40 /dev/null
    device_inode_id: usize,
}
```

为 `INodeImpl` 实现了以下函数：

```rust
/// Map file block id to disk block id
/// 传入的 file_block_id 表示 data block 在该文件中的编号
/// 我们要返回该 data block 在块设备中的编号
/// 主要是通过 DiskINode 里面的 direct[], indirect, db_indirect 字段
fn get_disk_block_id(&self, file_block_id: BlockId) -> vfs::Result<BlockId>;
/// 与 get 相对应，实现类似
fn set_disk_block_id(&self, file_block_id: BlockId, disk_block_id: BlockId) ->
vfs::Result<()>;

/// Only for Dir
/// 仅对目录项有效，根据名字查找 dirent ，返回其 INode number 以及在 dirent[] 中的位置
fn get_file_inode_and_entry_id(&self, name: &str) -> Option<(INodeId, usize)> {
    (0..self.disk_inode.read().size as usize / DIRENT_SIZE)
    .map(|i| (self.read_direntry(i as usize).unwrap(), i))
    .find(|(entry, _)| entry.name.as_ref() == name)
    .map(|(entry, id)| (entry.id as INodeId, id as usize))
}
/// 只返回 INode number
fn get_file_inode_id(&self, name: &str) -> Option<INodeId> {
    self.get_file_inode_and_entry_id(name)
    .map(|(inode_id, _)| inode_id)
}
/// 插入 . .. 两个 dirent
fn init_direntry(&self, parent: INodeId) -> vfs::Result<()>;
/// 给定 dirent id ，返回对应的 dirent
fn read_direntry(&self, id: usize) -> vfs::Result<DiskEntry>;
/// 修改对应位置的 dirent
fn write_direntry(&self, id: usize, direntry: &DiskEntry) -> vfs::Result<()>;
/// 追加 dirent
fn append_direntry(&self, direntry: &DiskEntry) -> vfs::Result<()>;
/// 删除 dirent
/// 实现方法是将位置最靠后的 dirent 覆盖到相应位置，并 resize
```

```rust
fn remove_direntry(&self, id: usize) -> vfs::Result<()>;

/// 一些常用的工具类函数
/// 修改 INode 的大小，单位为字节
/// 这里姑且不去看其具体实现
fn _resize(&self, len: usize) -> vfs::Result<()>;
/// 按照块迭代调用 f 函数，为后面的实现提供方便
/// 核心代码 f(&self.fs.device, &range, buf_offset)?;
/// 这里的 [begin, end) 指 INode 内部的字节区间
/// 传入的闭包给出了块设备，块设备中的 BlockRange (扇区号，扇区内区间)，对应的 &[u8] 偏移量
/// 实现中遍历 [begin, end) 中的所有块，调用相应的 F
fn _io_at<F>(&self, begin: usize, end: usize, mut f: F) -> vfs::Result<usize>
    where
        F: FnMut(&Arc<dyn Device>, &BlockRange, usize) -> vfs::Result<()>;
/// Read content, no matter what type it is
fn _read_at(&self, offset: usize, buf: &mut [u8]) -> vfs::Result<usize> {
    self._io_at(offset, offset + buf.len(), |device, range, offset| {
        device.read_block(range.block, range.begin, &mut buf[offset..offset +
range.len()])
    })
}
/// Write content, no matter what type it is
fn _write_at(&self, offset: usize, buf: &[u8]) -> vfs::Result<usize> {
    self._io_at(offset, offset + buf.len(), |device, range, offset| {
        device.write_block(range.block, range.begin, &buf[offset..offset +
range.len()])
    })
}
/// Clean content, no matter what type it is
fn _clean_at(&self, begin: usize, end: usize) -> vfs::Result<usize> {
    static ZEROS: [u8; BLKSIZE] = [0; BLKSIZE];
    self._io_at(begin, end, |device, range, _| {
        device.write_block(range.block, range.begin, &ZEROS[..range.len()])
    })
}

/// 链接相关函数
fn nlinks_inc(&self) {
    self.disk_inode.write().nlinks += 1;
}
fn nlinks_dec(&self);
/// 将 当前目录/name 与 other 对应的文件 硬链接
/// 只需更新自身的 dirent ，将 other 的 INode 编号指向过去，并更新 other 的 nlinks
pub fn link_inodeimpl(&self, name: &str, other: &Arc<INodeImpl>) ->
vfs::Result<()>;
```

还为 `INodeImpl` 实现了 `INode` 接口：

```rust
/// 基于 offset 进行文件 I/O
/// 注意如果是字符设备类型，要进行特殊处理
/// 否则只需调用 _read_at
fn read_at(&self, offset: usize, buf: &mut [u8]) -> vfs::Result<usize>;
fn write_at(&self, offset: usize, buf: &[u8]) -> vfs::Result<usize>
/// 目前看上去并没用到？
fn poll(&self) -> vfs::Result<vfs::PollStatus> {
    Ok(vfs::PollStatus {
```

```
        read: true,
        write: true,
        error: false,
    })
}

/// 这里返回的元数据中的 size 字段并不是实际字节数？
fn metadata(&self) -> vfs::Result<vfs::Metadata>;
/// 由于 DiskINode 被 Dirty 包裹，当发生了更改可以进行写回
/// sync_data 与 sync_all 相同
fn sync_all(&self) -> vfs::Result<()>;

/// 调用 _resize 之前先保证文件类型为 FileType::File 或 FileType::SymLink
///? 那么这里的 SymLink 是硬链接还是软链接(符号链接)呢？
///? 我猜是硬链接，但是 SymLink emmm
fn resize(&self, len: usize) -> vfs::Result<()>;

/// 在当前目录下新建一个 INode
/// 目前支持以下类型
/// vfs::FileType::File => self.fs.new_inode_file()?,
/// vfs::FileType::SymLink => self.fs.new_inode_symlink()?,
/// vfs::FileType::Dir => self.fs.new_inode_dir(self.id)?,
/// vfs::FileType::CharDevice => self.fs.new_inode_chardevice(data)?
/// 最后是对于 dirent 和 nlinks 的更新
fn create2(
        &self,
        name: &str,
        type_: vfs::FileType,
        _mode: u32,
        data: usize,
    ) -> vfs::Result<Arc<dyn vfs::INode>>;

/// 硬链接，实现与之前的 link_inodeimpl 基本相同
fn link(&self, name: &str, other: &Arc<dyn INode>) -> vfs::Result<()>;
/// 删除链接
/// 实现思路：查找其 dirent 可以找到源文件的 INode
/// 随后更新 nlinks ，删除 dirent 即可
/// 注意这里首次使用到了 let inode = self.fs.get_inode(inode_id);
/// 即通过 inode_id 来获取 fs 中的 INode
fn unlink(&self, name: &str) -> vfs::Result<()>;
/// 我猜语义是 mv ./old_name target.path/new_name ？
/// 支持目录的 move
/// 实现方法不过是删除一个 dirent ，再新增一个 dirent ，在此过程中注意修改 nlinks
/// 感觉这个 nlinks 可以被定义为：有多少种不同的绝对路径可以对应到该 INode
fn move_(&self, old_name: &str, target: &Arc<dyn INode>, new_name: &str) ->
vfs::Result<()>;
/// 典型的 dirent 查询：从名字获取 INode
fn find(&self, name: &str) -> vfs::Result<Arc<dyn vfs::INode>>;
/// 从 dirent id 获取名字
/// 这里可以知道 DiskINode 中的 size 字段存的是字节大小
fn get_entry(&self, id: usize) -> vfs::Result<String>;
/// 目前的 io_control 只针对字符设备
/// 核心功能是调用 device_inode.io_control(_cmd, _data)
/// 两个参数分别代表 设备的操作类型与数据
fn io_control(&self, _cmd: u32, _data: usize) -> vfs::Result<()>;
/// 注意这里是使用 Arc clone 出来的
fn fs(&self) -> Arc<dyn vfs::FileSystem> {
    self.fs.clone()
```

```
}
fn as_any_ref(&self) -> &dyn Any {
    self
}
```

后面就进入文件系统的实现 `SimpleFileSystem`，首先是结构体定义：

```
pub struct SimpleFileSystem {
    super_block: RwLock<Dirty<SuperBlock>>, /// 磁盘上的超级块
    free_map: RwLock<Dirty<BitVec>>, /// Bitmap ，被占用则为 0
    inodes: RwLock<BTreeMap<INodeId, Weak<INodeImpl>>>, /// INodeImpl 的 BTreeMap
    /// 这里使用 Weak 仅仅是为了避免引用循环溢出内存？貌似没有看到内存回收。
    device: Arc<dyn Device>, /// 文件系统底层的 Device
    self_ptr: Weak<SimpleFileSystem>, /// 指向自身的指针， 被 INodes 使用
    device_inodes: RwLock<BTreeMap<usize, Arc<DeviceINode>>>, /// 字符设备使用的
BTreeMap
}
```

具体实现：

```
/// 从设备打开文件系统，返回的是 Arc<SimpleFileSystem>
/// 从磁盘上载入超级块还有 freemap
pub fn open(device: Arc<dyn Device>) -> vfs::Result<Arc<Self>>;
/// 从空磁盘创建 sfs
pub fn create(device: Arc<dyn Device>, space: usize) -> vfs::Result<Arc<Self>>;
/// 在上面两个函数返回时，均使用 wrap 函数来完成 Arc 包裹
fn wrap(self) -> Arc<Self>;

/// 块分配
/// 只需修改 freemap 的对应位，并更新 superblock 元数据
fn alloc_block(&self) -> Option<usize>;
/// 块回收
fn free_block(&self, block_id: usize);
```