RUST的IO操作

IO是一门语言中内容最多，最繁杂的一个课题，其关心的主要内容:

1. 操作系统文件同步IO操作
2. 操作系统网络/设备同步IO操作
3. 操作系统多路异步IO操作，包括文件/设备/网络
4. 采用一套编程模型抽象与IO相关的缓存操作
5. 程序不同模块间通信采取与4相同的抽象接口
6. ...

值得注意，在服务器后端的应用中，实际上采用多路异步IO是标准的实现方式，异步IO内容稍微复杂，将在后继的异步IO章节里单独说明。 RUST的标准库中的IO内容实际上仅仅提供同步IO的实现。要注意的是，虽然标准库仅提供了同步IO，但异步IO实际上仍然可以将这些同步IO的实现作为基本组件来简化工作。

在同步IO设计中，符合自然视角的IO对象设计，线程安全设计，缓存设计是难点。

IO对象设计：
最自然的IO对象设计是针对每一个不同的IO分类设计不同的IO对象类型，不同的IO对象实现相同的操作 trait，仅在独特之处进行方法扩充。

线程安全设计：
每一个IO对象实际上都存在多线程操作的可能，IO对象的类型结构应该是一个RUST线程安全类型结构。

缓存设计： 不是所有的IO对象类型都需要缓存，设计缓存的作用主要是：

1. 可以将一些底层的IO操作封装在缓存实现中，简化上层模块IO实现。
2. 提升IO效率，对某些非实时IO操作，可以达到一定数目后批量性写入，或批量性读出
3. 更好的模块性，可以将缓存作为不同模块的IO管道，重用已有模块，例如重用压缩/解压缩模块
4. 用作数据序列化格式转换的执行类型，以及数据序列化的内存存储，方便各种操作

缓存设计的一些需求：

1. 缓存自身应该作为一种IO对象，
2. 缓存封装原始IO对象，使用adapter模式完成对原始IO对象的IO操作
3. 针对不同的IO对象的缓存基础设施结构，支持不同的IO对象的缓存设计
4. 迭代器设计以应用函数式编程。

先以标准输入的IO对象来阐明RUST的同步IO实现的代码:

RUST标准库Stdin的代码分析

RUST语言库实现了线程安全的标准输入。

```
//路径: library/std/src/io/stdio.rs
pub struct Stdin {
    //标准输入可认为是静态的
    inner: &'static Mutex<BufReader<StdinRaw>>,
}
```

`Mutex<T>`请参考前文的分析。 原始的标准输入源IO对象类型StdinRaw定义相关代码如下:

```
//linux系统的标准输入的类型结构
//因为标准输入的文件描述符不必关闭,
//所以此处用了单元类型
//路径: library/std/src/sys/unix/stdio.rs
pub struct Stdin(());

impl Stdin {
    //创建函数
    pub const fn new() -> Stdin {
        Stdin(())
    }
}

//RUST对操作系统的扩展
//路径: library/std/src/io/stdio.rs
//此处stdio是sys::stdio
struct StdinRaw(stdio::Stdin);

//StdinRaw的工厂函数
const fn stdin_raw() -> StdinRaw {
    StdinRaw(stdio::Stdin::new())
}
```

RUST专门为读入设计的缓存类型结构`BufReader<R>`定义如下:

```
//路径: library/std/src/io/buffer/bufreader.rs
//在实现了Read trait的输入源IO对象类型基础上创建读缓存结构
pub struct BufReader<R> {
    //输入源IO对象类型,
    //BufReader拥有其所有权
    inner: R,
```

```
    //缓存，在self创建的时候一般没有初始化
    //位于堆内存
    buf: Box<[MaybeUninit<u8>]>,
    //缓存中未被读取的数据起始位置
    pos: usize,
    //从输入源已经读入缓存的数据终止位置
    cap: usize,
    //buf中已经初始化过的数据的终止位置
    init: usize,
}

impl<R: Read> BufReader<R> {
    //创建一个默认空间的缓存
    pub fn new(inner: R) -> BufReader<R> {
        //DEFAULT_BUF_SIZE RUST当前定义为8*1024
        BufReader::with_capacity(DEFAULT_BUF_SIZE, inner)
    }

    pub fn with_capacity(capacity: usize, inner: R) -> BufReader<R> {
        //从堆中申请相应空间的内存
        let buf = Box::new_uninit_slice(capacity);
        //创建BufReader类型变量
        BufReader { inner, buf, pos: 0, cap: 0, init: 0 }
    }
}
```

对于所有的输入IO对象类型，必须实现Read trait: 定义如下：

```
//路径: library/std/src/io/mod.rs
//在异步IO时，此Read可以用于最底层的支持
pub trait Read {
    //从输入IO对象类型中读出数据到buf中，成功则返回读到的长度
    //否则返回IO错误，IO错误的情况下，buf中一定没有数据
    //此函数可能被阻塞，如果需要阻塞又没办法时，会返回Err
    //返回0一般表示已经读到文件尾部或fd已经关闭，或者buf空间为0
    fn read(&mut self, buf: &mut [u8]) -> Result<usize>;

    //利用向量读的方式读，除此之外，与read相同，IoSliceMut见后继说明
    fn read_vectored(&mut self, bufs: &mut [IoSliceMut<'_>]) -> Result<usize> {
        //默认不支持iovec的方式，使用read来模拟实现
        default_read_vectored(|b| self.read(b), bufs)
    }

    //是否实现向量读的方式,一般应优选向量读
    fn is_read_vectored(&self) -> bool {
        false
    }

    //此方法会循环调用read直至读到文件尾(EOF)
    //一直读到文件尾部，此方法内部可以自由扩充Vec，Vec中的有效内容代表已经读到
```

```
    //的数据。
    //遇到错误会立刻返回，读到的数据仍然在Vec中
    fn read_to_end(&mut self, buf: &mut Vec<u8>) -> Result<usize> {
        default_read_to_end(self, buf)
    }

    //类似与read_to_end，但这里确定读到的是字符串,且符合utf-8的编码
    //其他与read_to_end相同
    fn read_to_string(&mut self, buf: &mut String) -> Result<usize> {
        default_read_to_string(self, buf)
    }

    //精确读与buf长度相同的字节，否则返回错误
    //如果长度不够且到达尾部，会返回错误
    fn read_exact(&mut self, buf: &mut [u8]) -> Result<()> {
        default_read_exact(self, buf)
    }

    //在有缓存的情况下，用以下函数将数据读到缓存里
    //一般ReadBuf由缓存类型结构创建
    fn read_buf(&mut self, buf: &mut ReadBuf<'_>) -> Result<()> {
        default_read_buf(|b| self.read(b), buf)
    }

    //精确的将要求容量字节到缓存里面
    fn read_buf_exact(&mut self, buf: &mut ReadBuf<'_>) -> Result<()> {
        while buf.remaining() > 0 {
            let prev_filled = buf.filled().len();
            match self.read_buf(buf) {
                Ok(()) => {}
                Err(e) if e.kind() == ErrorKind::Interrupted => continue,
                Err(e) => return Err(e),
            }

            if buf.filled().len() == prev_filled {
                return Err(Error::new(ErrorKind::UnexpectedEof, "failed to fill
buffer"));
            }
        }

        Ok(())
    }

    //借用的一种实现方式，专为Read使用
    fn by_ref(&mut self) -> &mut Self
    where
        Self: Sized,
    {
        self
    }
```

```
    //将本身转换为一个字节流的迭代器
    //后继用迭代器的方法完成读
    fn bytes(self) -> Bytes<Self>
    where
        Self: Sized,
    {
        Bytes { inner: self }
    }

    //将两个读的源串接
    fn chain<R: Read>(self, next: R) -> Chain<Self, R>
    where
        Self: Sized,
    {
        Chain { first: self, second: next, done_first: false }
    }

    //以self为基础生成一个字节数有限制的输入源
    fn take(self, limit: u64) -> Take<Self>
    where
        Self: Sized,
    {
        Take { inner: self, limit }
    }
}
```

对于文件类及缓存类IO对象类型，一般也都实现了在IO流中定位特定位置的 Seek trait:

```
pub enum SeekFrom {
    /// 从头部开始向尾部偏移字节数.
    Start(u64),

    //从尾部开始向头部偏移字节数
    End(i64),

    //从当前位置开始向尾部偏移字节数
    Current(i64),
}
pub trait Seek {
    //定位到IO流的指定偏移位置，如果需要从当前位置向头部偏移
    //则返回错误
    //成功则返回从头部计算的的偏移字节数
    fn seek(&mut self, pos: SeekFrom) -> Result<u64>;

    //重新定位到头部
    fn rewind(&mut self) -> Result<()> {
        self.seek(SeekFrom::Start(0))?;
        Ok(())
    }
```

```
    //返回IO流的总长度
    fn stream_len(&mut self) -> Result<u64> {
        //保存当前位置
        let old_pos = self.stream_position()?;
        //重定位到尾
        let len = self.seek(SeekFrom::End(0))?;

        if old_pos != len {
            //返回到当前位置
            self.seek(SeekFrom::Start(old_pos))?;
        }

        Ok(len)
    }

    //返回当前位置
    fn stream_position(&mut self) -> Result<u64> {
        self.seek(SeekFrom::Current(0))
    }
}
```

针对std::sys::Stdio的Read trait实现：

```
//路径：library/std/src/sys/unix/stdio.rs
//实现Read trait
impl io::Read for Stdin {
    fn read(&mut self, buf: &mut [u8]) -> io::Result<usize> {
        //标准输入不必关闭，因此这里生成的OwnedFd不能调用drop
        //所以用ManuallyDrop来实现这一点
        //自动解引用调用FileDesc::read方法
        unsafe {
ManuallyDrop::new(FileDesc::from_raw_fd(libc::STDIN_FILENO)).read(buf) }
    }

    fn read_vectored(&mut self, bufs: &mut [IoSliceMut<'_>]) ->
io::Result<usize> {
        unsafe {
ManuallyDrop::new(FileDesc::from_raw_fd(libc::STDIN_FILENO)).read_vectored(bufs
) }
    }

    fn is_read_vectored(&self) -> bool {
        true
    }
}
```

针对StdioRaw的Read trait实现：

```rust
//路径: library/std/src/io/stdio.rs
//支持函数,处理输入输出的错误
fn handle_ebadf<T>(r: io::Result<T>, default: T) -> io::Result<T> {
    match r {
        //如果错误是fd无效，则返回默认值
        Err(ref e) if stdio::is_ebadf(e) => Ok(default),
        r => r,
    }
}

//RUST的IO对象类型通常是一个逐级封装的结构
//StdinRaw采用了adapter的模式实现Read trait
impl Read for StdinRaw {
    fn read(&mut self, buf: &mut [u8]) -> io::Result<usize> {
        //直接调用内部封装的stdin同名方法
        handle_ebadf(self.0.read(buf), 0)
    }

    fn read_vectored(&mut self, bufs: &mut [IoSliceMut<'_>]) ->
io::Result<usize> {
        handle_ebadf(self.0.read_vectored(bufs), 0)
    }

    fn is_read_vectored(&self) -> bool {
        self.0.is_read_vectored()
    }

    fn read_to_end(&mut self, buf: &mut Vec<u8>) -> io::Result<usize> {
        handle_ebadf(self.0.read_to_end(buf), 0)
    }

    fn read_to_string(&mut self, buf: &mut String) -> io::Result<usize> {
        handle_ebadf(self.0.read_to_string(buf), 0)
    }
}
```

RUST语言的Stdin实质是 `BufReader<StdinRaw>` 的线程安全版本。
BufReader需要实现基于缓存的 BufRead trait, 以充分利用缓存:

```rust
//路径: library/std/src/io/mod.rs
pub trait BufRead: Read {
    //从输入源IO对象读入并填充缓存，并将内部的缓存
    //以字节切片引用方式返回
    fn fill_buf(&mut self) -> Result<&[u8]>;

    //有amt的字节被从缓存读出，对self的参数做针对性改变
    fn consume(&mut self, amt: usize);

    //缓存是否还存在未被读出的数据
```

```
    fn has_data_left(&mut self) -> Result<bool> {
        self.fill_buf().map(|b| !b.is_empty())
    }

    //将buf读到buf中，直到有数据为输入的参数
    fn read_until(&mut self, byte: u8, buf: &mut Vec<u8>) -> Result<usize> {
        read_until(self, byte, buf)
    }

    //从缓存中读出一行
    fn read_line(&mut self, buf: &mut String) -> Result<usize> {
        //借助read_until简单实现
        unsafe { append_to_string(buf, |b| read_until(self, b'\n', b)) }
    }

    //返回一个迭代器，将buf按输入的参数做分离
    fn split(self, byte: u8) -> Split<Self>
    where
        Self: Sized,
    {
        Split { buf: self, delim: byte }
    }

    //返回一个迭代器，将buf按行进行迭代
    fn lines(self) -> Lines<Self>
    where
        Self: Sized,
    {
        Lines { buf: self }
    }
}

//上面trait的支持函数
fn read_until<R: BufRead + ?Sized>(r: &mut R, delim: u8, buf: &mut Vec<u8>) ->
Result<usize> {
    let mut read = 0;
    loop {
        let (done, used) = {
            //先将数据读入r的缓存中，available是新读入的内容
            let available = match r.fill_buf() {
                Ok(n) => n,
                Err(ref e) if e.kind() == ErrorKind::Interrupted => continue,
                Err(e) => return Err(e),
            };
            //在buf中定位第一个分隔符
            match memchr::memchr(delim, available) {
                //找到
                Some(i) => {
                    //将分隔符之前的内容置入buf中
                    buf.extend_from_slice(&available[..=i]);
                    (true, i + 1)
```

```
                }
                //没找到
                None => {
                    //将所有内容置入buf中
                    buf.extend_from_slice(available);
                    (false, available.len())
                }
            }
        };
        //更新r以反应已经读出的内容
        r.consume(used);
        //得到读到的字节总数
        read += used;
        //看是否已经读到分隔符，或者内容已经读空
        if done || used == 0 {
            //返回读到的字节总数目
            return Ok(read);
        }
    }
}

//BufRead::read_lines的Iterator类型支持结构
pub struct Lines<B> {
    buf: B,
}

impl<B: BufRead> Iterator for Lines<B> {
    type Item = Result<String>;

    fn next(&mut self) -> Option<Result<String>> {
        let mut buf = String::new();
        //调用read_line
        match self.buf.read_line(&mut buf) {
            Ok(0) => None,
            Ok(_n) => {
                if buf.ends_with('\n') {
                    //将'\n'删除
                    buf.pop();
                    if buf.ends_with('\r') {
                        //将'\r'删除
                        buf.pop();
                    }
                }
                Some(Ok(buf))
            }
            Err(e) => Some(Err(e)),
        }
    }
}
```

利用缓存从IO输入源读取数据时，RUST专门设计了ReadBuf的类型结构： 对于不同的数据，可能设计不同的缓存结构，BufReader仅仅是其中的一种。
但从输入源读入到缓存涉及的数据基本是固定不变的，即要读入的字节切片，字节切片已经读入的字节数，及RUST语法带来的字节切片中已经初始化的字节数，因此设计了ReadBuf以完成缓存读入的抽象类型结构体。

```
//路径:  library/std/src/io/readbuf.rs
pub struct ReadBuf<'a> {
    //用作缓存的内存块切片引用，由
    //外部的缓存类型提供，此块内存
    //第一次总是MaybeUninit
    buf: &'a mut [MaybeUninit<u8>],
    //已经读入的数据，buf[0..filled]是读入的数据
    filled: usize,
    //已经assume_init的数据,buf[0..initialized]
    //是初始化过的数据
    //代码中需要保证filled应该小于initialized
    initialized: usize,
}

impl<'a> ReadBuf<'a> {
    //用一个已经初始化的内存块创建一个ReadBuf，
    //此内存块应该已经初始化完毕
    pub fn new(buf: &'a mut [u8]) -> ReadBuf<'a> {
        let len = buf.len();

        ReadBuf {
            //强制转换为[MaybeUninit<u8>]类型
            buf: unsafe { (buf as *mut [u8]).as_uninit_slice_mut().unwrap() },
            //没有读入数据
            filled: 0,
            //此buf实际上已经初始化
            initialized: len,
        }
    }

    //用未初始化的内存块创建ReadBuf
    pub fn uninit(buf: &'a mut [MaybeUninit<u8>]) -> ReadBuf<'a> {
        ReadBuf { buf, filled: 0, initialized: 0 }
    }

    pub fn capacity(&self) -> usize {
        self.buf.len()
    }

    //返回已经读到的字节切片引用
    pub fn filled(&self) -> &[u8] {
        unsafe { MaybeUninit::slice_assume_init_ref(&self.buf[0..self.filled])
}
    }
```

```
    //返回已经读到的字节切片可变引用
    pub fn filled_mut(&mut self) -> &mut [u8] {
        unsafe { MaybeUninit::slice_assume_init_mut(&mut
self.buf[0..self.filled]) }
    }

    //返回已经初始化的字节切片引用
    pub fn initialized(&self) -> &[u8] {
        unsafe {
MaybeUninit::slice_assume_init_ref(&self.buf[0..self.initialized]) }
    }

    //返回已经初始化的字节切片可变引用
    pub fn initialized_mut(&mut self) -> &mut [u8] {
        unsafe { MaybeUninit::slice_assume_init_mut(&mut
self.buf[0..self.initialized]) }
    }

    //返回没有读入字节的缓存部分的可变引用切片
    pub unsafe fn unfilled_mut(&mut self) -> &mut [MaybeUninit<u8>] {
        &mut self.buf[self.filled..]
    }

    //返回没有做assume_init的缓存部分的可变引用切片
    pub fn uninitialized_mut(&mut self) -> &mut [MaybeUninit<u8>] {
        &mut self.buf[self.initialized..]
    }

    //对所有的未读入字节的缓存做assume_init
    pub fn initialize_unfilled(&mut self) -> &mut [u8] {
        self.initialize_unfilled_to(self.remaining())
    }

    //从未读到字节的起始字节开始设置若干个字节assume_init
    pub fn initialize_unfilled_to(&mut self, n: usize) -> &mut [u8] {
        assert!(self.remaining() >= n);

        //获取没有读入内容却已经初始化的字节数
        let extra_init = self.initialized - self.filled;
        //判断是否需要额外做初始化
        if n > extra_init {
            //获取需要初始化的字节数
            let uninit = n - extra_init;
            //获取需要初始化的字节切片
            let unfilled = &mut self.uninitialized_mut()[0..uninit];

            //完成初始化为0
            for byte in unfilled.iter_mut() {
                byte.write(0);
            }
```

```
            unsafe {
                //设置为已经初始化
                self.assume_init(n);
            }
        }

        let filled = self.filled;

        //返回初始化但没有读到内容的字节切片
        &mut self.initialized_mut()[filled..filled + n]
    }

    //空闲的字节数目
    pub fn remaining(&self) -> usize {
        self.capacity() - self.filled
    }

    //清除已读的内容，仅需要设置filled数值即可
    pub fn clear(&mut self) {
        self.set_filled(0); // The assertion in `set_filled` is optimized out
    }

    //增加已读内容字节数
    pub fn add_filled(&mut self, n: usize) {
        self.set_filled(self.filled + n);
    }

    //设置已经读入内容的字节数
    pub fn set_filled(&mut self, n: usize) {
        assert!(n <= self.initialized);

        self.filled = n;
    }

    //设置已经初始化的字节数目
    pub unsafe fn assume_init(&mut self, n: usize) {
        self.initialized = cmp::max(self.initialized, self.filled + n);
    }

    //将内容拷贝入已读内容之后作为新读入的内容
    pub fn append(&mut self, buf: &[u8]) {
        assert!(self.remaining() >= buf.len());

        unsafe {
            //需要用MaybeUninit的方法来完成内容更新
            MaybeUninit::write_slice(&mut self.unfilled_mut()[..buf.len()],
buf);
        }

        //更新初始化的字节数及读入内容的字节数
```

```
            unsafe { self.assume_init(buf.len()) }
            self.add_filled(buf.len());
        }

        //获取filled参数
        pub fn filled_len(&self) -> usize {
            self.filled
        }

        //获取初始化的参数
        pub fn initialized_len(&self) -> usize {
            self.initialized
        }
    }
```

BufRead的基础方法，BufRead trait, Read trait实现如下：

```
//路径:  library/std/src/io/buffer/bufreader.rs
impl<R> BufReader<R> {
    //获取内部的输入源引用
    pub fn get_ref(&self) -> &R {
        &self.inner
    }

    //获取内部输入源的可变引用
    pub fn get_mut(&mut self) -> &mut R {
        &mut self.inner
    }

    pub fn buffer(&self) -> &[u8] {
        //将已经读入缓存，但未从缓存读出的内容以切片返回，且完成初始化操作
        unsafe {
MaybeUninit::slice_assume_init_ref(&self.buf[self.pos..self.cap]) }
    }

    //缓存内存空间大小
    pub fn capacity(&self) -> usize {
        self.buf.len()
    }

    //消费self并取出内部输入源
    pub fn into_inner(self) -> R {
        self.inner
    }

    //丢弃已经读入缓存的内容,此处似乎用drop更符合rust
    //用discar实际上是类似C的方式了
    fn discard_buffer(&mut self) {
        self.pos = 0;
        self.cap = 0;
```

```
    }
}

//为数据读入缓存设计的trait
impl<R: Read> BufRead for BufReader<R> {
    fn fill_buf(&mut self) -> io::Result<&[u8]> {
        //判断缓存中是否还有未读的内容
        if self.pos >= self.cap {
            //没有，则清理缓存，并从输入源读入新的内容
            debug_assert!(self.pos == self.cap);

            //利用self.buf创建ReadBuf类型结构体变量完成读
            let mut readbuf = ReadBuf::uninit(&mut self.buf);

            unsafe {
                //传递buf中已经assmue_init过的字节数
                readbuf.assume_init(self.init);
            }

            //调用输入源IO对象的read_buf完成缓存读
            self.inner.read_buf(&mut readbuf)?;

            //根据readbuf的参数修改self参数
            self.cap = readbuf.filled_len();
            self.init = readbuf.initialized_len();

            //更新初始位置
            self.pos = 0;
        }
        //返回缓存内的有效内容
        Ok(self.buffer())
    }

    //对已经从缓存读出的数据完成参数调整
    fn consume(&mut self, amt: usize) {
        self.pos = cmp::min(self.pos + amt, self.cap);
    }
}

impl<R: Read> Read for BufReader<R> {
    fn read(&mut self, buf: &mut [u8]) -> io::Result<usize> {
        //判断缓存是否为空且要读出的数据长度大于缓存容量
        if self.pos == self.cap && buf.len() >= self.buf.len() {
            //是，将缓存参数复位
            self.discard_buffer();
            //旁路缓存，直接将数据读入参数中的buf
            return self.inner.read(buf);
        }
        //缓存内有数据，或者要读出的数据长度小于缓存容量
        let nread = {
            //先填充缓存
```

```
            let mut rem = self.fill_buf()?;
            //实质是&[u8] as Read::read(buf),
            rem.read(buf)?
        };
        //调整参数反应已经从缓存读出的字节数
        self.consume(nread);
        Ok(nread)
    }

    fn read_buf(&mut self, buf: &mut ReadBuf<'_>) -> io::Result<()> {
        //见read的逻辑
        if self.pos == self.cap && buf.remaining() >= self.buf.len() {
            self.discard_buffer();
            return self.inner.read_buf(buf);
        }

        //获取原有的已读字节
        let prev = buf.filled_len();

        //填充缓存
        let mut rem = self.fill_buf()?;
        //&[u8] as Read::read_buf()
        rem.read_buf(buf)?;

        //获得本次读的字节数，更新参数
        self.consume(buf.filled_len() - prev); //slice impl of read_buf known
to never unfill buf

        Ok(())
    }

    //精确读取给定长度的内容
    fn read_exact(&mut self, buf: &mut [u8]) -> io::Result<()> {
        //判断缓存中是否已经有足够的已读字节
        if self.buffer().len() >= buf.len() {
            //有，从缓存拷贝到buf
            buf.copy_from_slice(&self.buffer()[..buf.len()]);
            //调整本身参数
            self.consume(buf.len());
            return Ok(());
        }

        //没有，用默认精确读
        crate::io::default_read_exact(self, buf)
    }

    //向量读方法
    fn read_vectored(&mut self, bufs: &mut [IoSliceMut<'_>]) ->
io::Result<usize> {
        //获取总体要读的字节数
        let total_len = bufs.iter().map(|b| b.len()).sum::<usize>();
```

```
        //判断缓存是否为空,且读取总字节数大于缓存长度
        if self.pos == self.cap && total_len >= self.buf.len() {
            //清空缓存
            self.discard_buffer();
            //直接读入参数给出的buf
            return self.inner.read_vectored(bufs);
        }
        //缓存不为空或读取总长度小于缓存长度
        let nread = {
            //填充缓存
            let mut rem = self.fill_buf()?;
            //&[u8]::read_vectored
            rem.read_vectored(bufs)?
        };
        //更新缓存参数
        self.consume(nread);
        Ok(nread)
    }

    fn is_read_vectored(&self) -> bool {
        self.inner.is_read_vectored()
    }

    fn read_to_end(&mut self, buf: &mut Vec<u8>) -> io::Result<usize> {
        //先将缓存内容读到buf中
        let nread = self.cap - self.pos;
        buf.extend_from_slice(&self.buffer());
        //清空缓存
        self.discard_buffer();
        //再将内部输入源的内容全部读出到输入的buf中,
        //返回本次操作的总长度
        Ok(nread + self.inner.read_to_end(buf)?)
    }

    fn read_to_string(&mut self, buf: &mut String) -> io::Result<usize> {
        //判断是否为空字符串
        if buf.is_empty() {
            //空字符串，则直接用append_to_string完成即可
            unsafe { crate::io::append_to_string(buf, |b| self.read_to_end(b))
}
        } else {
            //不是空字符串
            //先将内容读入创建的缓存中
            let mut bytes = Vec::new();
            self.read_to_end(&mut bytes)?;
            //从缓存生成字符串,并连接到输入字符串尾部
            let string = crate::str::from_utf8(&bytes).map_err(|_| {
                io::const_io_error!(
                    io::ErrorKind::InvalidData,
                    "stream did not contain valid UTF-8",
                )
```

```
            })?;
            *buf += string;
            Ok(string.len())
        }
    }
}

//Seek trait的实现
impl<R: Seek> Seek for BufReader<R> {
    fn seek(&mut self, pos: SeekFrom) -> io::Result<u64> {
        let result: u64;
        if let SeekFrom::Current(n) = pos {
            //从当前位置偏移
            //获取剩余未读的字节数
            let remainder = (self.cap - self.pos) as i64;

            //如果偏移字节大于缓存内未读的字节
            if let Some(offset) = n.checked_sub(remainder) {
                //需要对输入源进行偏移
                result = self.inner.seek(SeekFrom::Current(offset))?;
            } else {
                //偏移字节小于缓存内未读的字节
                //此时需要将缓存内已读的字节清空，
                self.inner.seek(SeekFrom::Current(-remainder))?;
                self.discard_buffer();
                //将输入源偏移到新的位置
                result = self.inner.seek(SeekFrom::Current(n))?;
            }
        } else {
            //不是从当前位置偏移，则直接在输入源做偏移
            result = self.inner.seek(pos)?;
        }
        //偏移后需要清空缓存
        self.discard_buffer();
        Ok(result)
    }

    fn stream_position(&mut self) -> io::Result<u64> {
        //获得未被读出的字节数
        let remainder = (self.cap - self.pos) as u64;
        //从底层IO对象获得当前位置，再减掉未被读出的字节
        self.inner.stream_position().map(|pos| {
            pos.checked_sub(remainder).expect(
                "overflow when subtracting remaining buffer size from inner
stream position",
            )
        })
    }
}
```

以上涉及的一些支持函数代码如下：

```
//路径:  library/std/src/io/mod.rs
//Read trait精确读若干字节的默认实现。精确读的例子是处理有总长度字段的协议包头。只有读出包头才能知道整个数据包的长度
//因此一般先读一个固定长度的数据包报文头
pub(crate) fn default_read_exact<R: Read + ?Sized>(this: &mut R, mut buf: &mut [u8]) -> Result<()> {
    //循环直到读到要求的字节数目
    while !buf.is_empty() {
        match this.read(buf) {
            //输入源已经没有内容
            Ok(0) => break,
            //根据读到的内容更新buf
            Ok(n) => {
                //这个交换是比较经典的编码技巧
                let tmp = buf;
                buf = &mut tmp[n..];
            }
            //如果是操作系统的原因造成中断，则继续循环
            Err(ref e) if e.kind() == ErrorKind::Interrupted => {}
            //其他错误返回
            Err(e) => return Err(e),
        }
    }
    //判断是否读到字节数目
    if !buf.is_empty() {
        //否，仅因为输入源已经没有内容的错误
        Err(error::const_io_error!(ErrorKind::UnexpectedEof, "failed to fill whole buffer"))
    } else {
        Ok(())
    }
}

//Read trait中read_buf的默认函数
pub(crate) fn default_read_buf<F>(read: F, buf: &mut ReadBuf<'_>) -> Result<()>
where
    F: FnOnce(&mut [u8]) -> Result<usize>,
{
    //对buf中没读入的空间先全部初始化，然后做读操作
    let n = read(buf.initialize_unfilled())?;
    //根据读入字节更新ReadBuf参数
    buf.add_filled(n);
    Ok(())
}

//Read trait中默认的read_to_end的方法实现
pub(crate) fn default_read_to_end<R: Read + ?Sized>(r: &mut R, buf: &mut Vec<u8>) -> Result<usize> {
```

```
    let start_len = buf.len();
    let start_cap = buf.capacity();

    //初始化但没有读入内容的字节数为0
    let mut initialized = 0; // Extra initialized bytes from previous loop
iteration
    loop {
        if buf.len() == buf.capacity() {
            //buf已经没有空间,
            //对buf进行扩充
            buf.reserve(32); // buf is full, need more space
        }

        //将buf中没有填充内容的部分生成切片，并创建ReadBuf
        let mut read_buf = ReadBuf::uninit(buf.spare_capacity_mut());

        // 将read_buf的参数设置正确
        unsafe {
            //设置ReadBuf的初始化字节数
            read_buf.assume_init(initialized);
        }

        //调用输入源的read_buf读入内容
        match r.read_buf(&mut read_buf) {
            Ok(()) => {}
            //操作系统的原因中断，继续循环
            Err(e) if e.kind() == ErrorKind::Interrupted => continue,
            //出错则返回
            Err(e) => return Err(e),
        }

        //判断输入源是否已经全部被读入
        if read_buf.filled_len() == 0 {
            //已经全部读入，则返回
            return Ok(buf.len() - start_len);
        }

        //输入源仍然可能有数据没有被读入
        //根据ReadBuf更新已经初始化但没有读入内容的字节数
        initialized = read_buf.initialized_len() - read_buf.filled_len();
        //根据读入字节的数目修改Vec的参数
        let new_len = read_buf.filled_len() + buf.len();

        unsafe {
            //设置Vec的参数反应已经读入的内容
            buf.set_len(new_len);
        }

        //前面最多只能读到buf.capacity()
        //判断初始传入的buf是否已经读满
        if buf.len() == buf.capacity() && buf.capacity() == start_cap {
```

```
            let mut probe = [0u8; 32];

            loop {
                //每次循环最多读取32个字节的额外内容
                match r.read(&mut probe) {
                    //输入源已经没有内容
                    Ok(0) => return Ok(buf.len() - start_len),
                    //输入源还有内容
                    Ok(n) => {
                        //对buf做扩展并放置新的内容
                        buf.extend_from_slice(&probe[..n]);
                        //重新进入上级循环
                        break;
                    }
                    Err(ref e) if e.kind() == ErrorKind::Interrupted =>
continue,
                    Err(e) => return Err(e),
                }
            }
        }
    }
}

//Read trait的默认read_to_string的函数
pub(crate) fn default_read_to_string<R: Read + ?Sized>(
    r: &mut R,
    buf: &mut String,
) -> Result<usize> {
    //对读入做是否为字符的判断，并且在判断非字符的时候将String恢复为初始值
    unsafe { append_to_string(buf, |b| default_read_to_end(r, b)) }
}

//上个函数的支持函数
pub(crate) unsafe fn append_to_string<F>(buf: &mut String, f: F) ->
Result<usize>
where
    F: FnOnce(&mut Vec<u8>) -> Result<usize>,
{
    //利用Guard保证错误处理
    let mut g = Guard { len: buf.len(), buf: buf.as_mut_vec() };
    //对g.buf做更新
    let ret = f(g.buf);
    //对g.buf新增的内容判断是否为字符串
    if str::from_utf8(&g.buf[g.len..]).is_err() {
        //不是字符串,返回错误
        ret.and_then(|_| {
            Err(error::const_io_error!(
                ErrorKind::InvalidData,
                "stream did not contain valid UTF-8"
            ))
        })
```

```
            //此处用Guard的结构保证了g.buf会被恢复成输入时的状态
        } else {
            //是字符串，对g做更新，返回读到的字节数
            g.len = g.buf.len();
            ret
        }
        //Guard保证了buf里面内容的正确，但有些不够直接
}
//上面函数的Guard相关内容
//这个结构是为了使用drop
struct Guard<'a> {
    buf: &'a mut Vec<u8>,
    len: usize,
}

impl Drop for Guard<'_> {
    fn drop(&mut self) {
        unsafe {
            //对buf的len做修改
            self.buf.set_len(self.len);
        }
    }
}

//直接从输入源内容创建String，
pub fn read_to_string<R: Read>(mut reader: R) -> Result<String> {
    let mut buf = String::new();
    reader.read_to_string(&mut buf)?;
    Ok(buf)
}
//操作系统不支持向量读写的方式时的默认实现
//用输入源提供的read来实现
pub(crate) fn default_read_vectored<F>(read: F, bufs: &mut [IoSliceMut<'_>]) ->
Result<usize>
where
    F: FnOnce(&mut [u8]) -> Result<usize>,
{
    //[]实际上是[u8;0]
    let buf = bufs.iter_mut().find(|b| !b.is_empty()).map_or(&mut [][..], |b|
&mut **b);
    read(buf)
}
```

以上完成了所有RUST的对外类型结构Stdin的相关类型的代码分析。现在回到Stdin本身：再次看一下Stdin的类型结构的相关实现：

```
//路径:  library/std/src/io/stdio.rs
//RUST的对外的标准输入结构
pub struct Stdin {
    inner: &'static Mutex<BufReader<StdinRaw>>,
```

```rust
}

//Stdin的Mutex.lock返回的借用类型结构
pub struct StdinLock<'a> {
    inner: MutexGuard<'a, BufReader<StdinRaw>>,
}

//获取标准输入
pub fn stdin() -> Stdin {
    //在本文写作的时候SyncOnceCell已经改为OnceLock
    //但内容基本没有变化，SyncOnceCell是适配在多线程的情况下只完成
    //一次初始化的类型结构，是OnceCell的线程安全版本
    //INSTANCE保证一个进程内只有一个Stdin变量被初始化
    static INSTANCE: SyncOnceCell<Mutex<BufReader<StdinRaw>>> =
SyncOnceCell::new();
    Stdin {
        //如果未初始化，则进行初始化，如果已经初始化，则获取Mutex的引用
        //并返回基于此引用创建的Stdin变量
        inner: INSTANCE.get_or_init(|| {
            Mutex::new(BufReader::with_capacity(stdio::STDIN_BUF_SIZE,
stdin_raw()))
        }),
    }
}

impl Stdin {
    pub fn lock(&self) -> StdinLock<'static> {
        //对标准输入上锁，并获取临界变量
        StdinLock { inner: self.inner.lock().unwrap_or_else(|e| e.into_inner())
}
    }

    pub fn read_line(&self, buf: &mut String) -> io::Result<usize> {
        //StdinLock的适配器实现,见下面分析
        self.lock().read_line(buf)
    }

    //返回一个每次读入一行的迭代器
    pub fn lines(self) -> Lines<StdinLock<'static>> {
        //StdinLock的适配器实现
        self.lock().lines()
    }
}

//实现Read trait,即StdinLock的适配器实现
impl Read for Stdin {
    fn read(&mut self, buf: &mut [u8]) -> io::Result<usize> {
        //直接是BufReader.read
        self.lock().read(buf)
    }
```

```
    //以下与read的实现类似，函数体略
    fn read_vectored(&mut self, bufs: &mut [IoSliceMut<'_>]) ->
io::Result<usize>;
    fn is_read_vectored(&self) -> bool;
    fn read_to_end(&mut self, buf: &mut Vec<u8>) -> io::Result<usize>;
    fn read_to_string(&mut self, buf: &mut String) -> io::Result<usize> ;
    fn read_exact(&mut self, buf: &mut [u8]) -> io::Result<()> ;
}

//Mutex加锁后的返回
impl StdinLock<'_> {
    pub(crate) fn as_mut_buf(&mut self) -> &mut BufReader<impl Read> {
        &mut self.inner
    }
}

//以下是BufReader的Read trait的适配实现
impl Read for StdinLock<'_> {
    fn read(&mut self, buf: &mut [u8]) -> io::Result<usize> {
        //BufReader::read
        self.inner.read(buf)
    }

    //以下实现与read的实现形式基本类似，函数体略

    fn read_vectored(&mut self, bufs: &mut [IoSliceMut<'_>]) ->
io::Result<usize>;
    fn is_read_vectored(&self) -> bool;
    fn read_to_end(&mut self, buf: &mut Vec<u8>) -> io::Result<usize> ;
    fn read_to_string(&mut self, buf: &mut String) -> io::Result<usize> ;
    fn read_exact(&mut self, buf: &mut [u8]) -> io::Result<()> ;
}

//以下是BufReader的BufRead trait的适配实现
impl BufRead for StdinLock<'_> {
    //以下函数体略
    fn fill_buf(&mut self) -> io::Result<&[u8]> ;
    fn consume(&mut self, n: usize) ;
    fn read_until(&mut self, byte: u8, buf: &mut Vec<u8>) -> io::Result<usize>
;
    fn read_line(&mut self, buf: &mut String) -> io::Result<usize> ;
}
```

常用函数:

```
//直接将输入源读入一个string
pub fn read_to_string<R: Read>(mut reader: R) -> Result<String> {
    let mut buf = String::new();
    reader.read_to_string(&mut buf)?;
```

```
    Ok(buf)
}
```

一些其他的支持实现：

```
//标准输入/输出/错误不能形成OwnedFd，但可以通过借用
//生成BorrowedFd
impl AsFd for io::Stdin {
    fn as_fd(&self) -> BorrowedFd<'_> {
        unsafe { BorrowedFd::borrow_raw(libc::STDIN_FILENO) }
    }
}

impl<'a> AsFd for io::StdinLock<'a> {
    fn as_fd(&self) -> BorrowedFd<'_> {
        unsafe { BorrowedFd::borrow_raw(libc::STDIN_FILENO) }
    }
}

//针对字节切片的Read trait实现
impl Read for &[u8] {
    //本质上是完成两个字节数组的拷贝，
    //完成了字节数组的长度处理
    //函数执行后，self会更新反应内容已经读出
    fn read(&mut self, buf: &mut [u8]) -> io::Result<usize> {
        //长度小的作为拷贝长度
        let amt = cmp::min(buf.len(), self.len());
        //将本身依据拷贝长度分成两个部分
        let (a, b) = self.split_at(amt);

        if amt == 1 {
            //提高效率
            buf[0] = a[0];
        } else {
            //将self拷贝到buf
            buf[..amt].copy_from_slice(a);
        }

        //更新self，此处易忽略
        *self = b;
        Ok(amt)
    }

    ...
    ...
}
```

标准输入使用的代码例如下：

```
let stdin = stdio::stdin();
let mut first_string = read_to_string(stdin);
let line = stdin.read_line(first_string);
```

linux的向量读写相关类型结构:

```
//路径: library/std/src/sys/unix/io.rs
//libc中的iovec的RUST封装,
//iovec用于多个缓存在一次读操作或写操作完成,
//减少将多个缓存移动到一个缓存造成的性能下降
//IoSlice通常用于写
//内存中IoSlice等同于iovec
#[repr(transparent)]
pub struct IoSlice<'a> {
    //libc中用于io读写的结构
    vec: iovec,
    //拥有读写的buf的所有权
    _p: PhantomData<&'a [u8]>,
}

impl<'a> IoSlice<'a> {
    //简化libc中的iovec的结构生成代码
    pub fn new(buf: &'a [u8]) -> IoSlice<'a> {
        IoSlice {
            vec: iovec { iov_base: buf.as_ptr() as *mut u8 as *mut c_void,
iov_len: buf.len() },
            _p: PhantomData,
        }
    }

    //向前至还未使用的buf的第一个字节
    pub fn advance(&mut self, n: usize) {
        if self.vec.iov_len < n {
            panic!("advancing IoSlice beyond its length");
        }

        unsafe {
            //调整iovec参数
            self.vec.iov_len -= n;
            self.vec.iov_base = self.vec.iov_base.add(n);
        }
    }

    //生成读写的缓存buf
    pub fn as_slice(&self) -> &[u8] {
        unsafe { slice::from_raw_parts(self.vec.iov_base as *mut u8,
self.vec.iov_len) }
    }
}
```

```rust
//通常应用于读入
pub struct IoSliceMut<'a> {
    vec: iovec,
    _p: PhantomData<&'a mut [u8]>,
}

//见IoSlice的相关结构的分析
impl<'a> IoSliceMut<'a> {
    pub fn new(buf: &'a mut [u8]) -> IoSliceMut<'a> {
        IoSliceMut {
            vec: iovec { iov_base: buf.as_mut_ptr() as *mut c_void, iov_len:
buf.len() },
            _p: PhantomData,
        }
    }

    pub fn advance(&mut self, n: usize) {
        if self.vec.iov_len < n {
            panic!("advancing IoSliceMut beyond its length");
        }

        unsafe {
            self.vec.iov_len -= n;
            self.vec.iov_base = self.vec.iov_base.add(n);
        }
    }

    pub fn as_slice(&self) -> &[u8] {
        unsafe { slice::from_raw_parts(self.vec.iov_base as *mut u8,
self.vec.iov_len) }
    }

    pub fn as_mut_slice(&mut self) -> &mut [u8] {
        unsafe { slice::from_raw_parts_mut(self.vec.iov_base as *mut u8,
self.vec.iov_len) }
    }
}

//IoSlice/IoSliceMut的使用如下
impl FileDesc {
    ...

    //一次读入多个buf
    pub fn read_vectored(&self, bufs: &mut [IoSliceMut<'_>]) ->
io::Result<usize> {
        let ret = cvt(unsafe {
            libc::readv(
                self.as_raw_fd(),
                bufs.as_ptr() as *const libc::iovec,
                cmp::min(bufs.len(), max_iov()) as c_int,
```

```
            )
        })?;
        Ok(ret as usize)
    }

    //一次写入多个buf
    pub fn write_vectored(&self, bufs: &[IoSlice<'_>]) -> io::Result<usize> {
        let ret = cvt(unsafe {
            libc::writev(
                self.as_raw_fd(),
                bufs.as_ptr() as *const libc::iovec,
                cmp::min(bufs.len(), max_iov()) as c_int,
            )
        })?;
        Ok(ret as usize)
    }
    ...
}

//RUST对linux向量读写的扩展
//直接对操作系统的基础类型结构封装
pub struct IoSliceMut<'a>(sys::io::IoSliceMut<'a>);

unsafe impl<'a> Send for IoSliceMut<'a> {}
unsafe impl<'a> Sync for IoSliceMut<'a> {}


impl<'a> IoSliceMut<'a> {
    //见上面的linux的IoSliceMut的分析
    pub fn new(buf: &'a mut [u8]) -> IoSliceMut<'a> {
        IoSliceMut(sys::io::IoSliceMut::new(buf))
    }

    //见上面的linux的IoSliceMut的分析
    pub fn advance(&mut self, n: usize) {
        self.0.advance(n)
    }

    pub fn advance_slices(bufs: &mut &mut [IoSliceMut<'a>], n: usize) {
        // 需要前移的IoSliceMut成员数目
        let mut remove = 0;
        // 计算前移的总体字节数
        let mut accumulated_len = 0;
        for buf in bufs.iter() {
            //是否应该前移到此成员
            if accumulated_len + buf.len() > n {
                //找到
                break;
            } else {
                //否，找下一个
                accumulated_len += buf.len();
```

```
                remove += 1;
            }
        }

        //此处的逻辑是:
        //必须获得&mut [IoSliceMut]的所有权，但不能用 = *bufs的方式实现
        //此时只能用replace来完成
        //要获得*bufs的所有权，只能用replace的方法,
        *bufs = &mut replace(bufs, &mut [])[remove..];
        if !bufs.is_empty() {
            bufs[0].advance(n - accumulated_len)
        }
    }
}

impl<'a> Deref for IoSliceMut<'a> {
    type Target = [u8];

    fn deref(&self) -> &[u8] {
        self.0.as_slice()
    }
}

impl<'a> DerefMut for IoSliceMut<'a> {
    fn deref_mut(&mut self) -> &mut [u8] {
        self.0.as_mut_slice()
    }
}
```

以上是Stdin相关的IO类型结构及其函数，方法

RUST标准库Stdout代码分析

RUST实现了线程安全的标准输入类型结构：

```
pub struct Stdout {
    //可重入的内部可变性类型
    //LineWriter是缓存类型结构
    inner: Pin<&'static ReentrantMutex<RefCell<LineWriter<StdoutRaw>>>>,
}
```

`ReentrantMutex<T>`，`RefCell<T>`请参考前文 原始的标准输出目的IO对象类型StdoutRaw定义相关代码如下：

```
//linux系统的标准输出的类型结构
//因为标准输出的文件描述符不必关闭,
//所以此处用了单元类型
```

```rust
//路径: library/std/src/sys/unix/stdio.rs
pub struct Stdout(());

impl Stdout {
    //创建函数
    pub const fn new() -> Stdout {
        Stdout(())
    }
}

//RUST对操作系统的扩展
//路径: library/std/src/io/stdio.rs
//此处stdio是sys::stdio
struct StdoutRaw(stdio::Stdout);

//StdoutRaw的工厂函数
const fn stdout_raw() -> StdoutRaw {
    StdoutRaw(stdio::Stdout::new())
}
```

RUST专门为行输出设计的缓存类型IO类型结构`LineWriter<W>`定义如下：

```rust
//LineWriter是BufWriter的一个adapter
//针对行输出做出优化
pub struct LineWriter<W: Write> {
    inner: BufWriter<W>,
}

//输出缓存类型结构
pub struct BufWriter<W: Write> {
    //输出目的IO对象类型结构变量
    //本结构体拥有其所有权
    inner: W,
    //缓存
    buf: Vec<u8>,
    //是否在输出的过程中出现线程panic
    panicked: bool,
}
//BufWriter创建函数
impl<W: Write> BufWriter<W> {
    //基于输出目的IO对象变量创建缓存
    pub fn new(inner: W) -> BufWriter<W> {
        //DEFAULT_BUF_SIZE是8*1024
        BufWriter::with_capacity(DEFAULT_BUF_SIZE, inner)
    }

    //创建指定容量的BufWriter
    pub fn with_capacity(capacity: usize, inner: W) -> BufWriter<W> {
        BufWriter { inner, buf: Vec::with_capacity(capacity), panicked: false }
    }
```

```
    ...
}
//LineWriter的创建函数
impl<W: Write> LineWriter<W> {
    pub fn new(inner: W) -> LineWriter<W> {
        //创建一个缓存为1024的LineWriter
        LineWriter::with_capacity(1024, inner)
    }

    //创建指定容量的LineWriter
    pub fn with_capacity(capacity: usize, inner: W) -> LineWriter<W> {
        //指定内部的BufWriter的容量
        LineWriter { inner: BufWriter::with_capacity(capacity, inner) }
    }
    ...
}
```

所有的输出IO对象类型都必须实现Write trait,定义如下：

```
pub trait Write {
    //将buf写入输出目的，返回写入的字节数
    //此写操作可能阻塞，或返回错误，
    //此函数完成后，输出可能保存在操作系统的缓存中
    //需要调用flush才能确保真正的完成输出
    fn write(&mut self, buf: &[u8]) -> Result<usize>;

    //以向量的形式写
    fn write_vectored(&mut self, bufs: &[IoSlice<'_>]) -> Result<usize> {
        //默认不支持向量，使用write方法来进行模拟
        default_write_vectored(|b| self.write(b), bufs)
    }

    //是否支持向量写
    fn is_write_vectored(&self) -> bool {
        false
    }

    //确保缓存内的输出已经输出到输出目的
    fn flush(&mut self) -> Result<()>;

    //确保缓存内的所有内容都已经写入输出目的
    fn write_all(&mut self, mut buf: &[u8]) -> Result<()> {
        //主要因为单次write可能会在没有写完时返回
        //所以循环以确保写完后再返回
        while !buf.is_empty() {
            match self.write(buf) {
                //写入0字节代表未知错误
                Ok(0) => {
                    return Err(error::const_io_error!(
                        ErrorKind::WriteZero,
```

```
                        "failed to write whole buffer",
                    ));
                }
                //写入n字节，则调整未写入的缓存
                Ok(n) => buf = &buf[n..],
                //被中断打断的错误继续做循环
                Err(ref e) if e.kind() == ErrorKind::Interrupted => {}
                //其他错误返回
                Err(e) => return Err(e),
            }
        }
        //已经写入了缓存的所有内容
        Ok(())
    }

    //vetor的write_all版本
    fn write_all_vectored(&mut self, mut bufs: &mut [IoSlice<'_>]) ->
Result<()> {
        //确保vector中有内容
        IoSlice::advance_slices(&mut bufs, 0);
        //循环防止单次写入被中断
        while !bufs.is_empty() {
            match self.write_vectored(bufs) {
                Ok(0) => {
                    return Err(error::const_io_error!(
                        ErrorKind::WriteZero,
                        "failed to write whole buffer",
                    ));
                }
                //写入则调整未写入的vetor
                Ok(n) => IoSlice::advance_slices(&mut bufs, n),
                Err(ref e) if e.kind() == ErrorKind::Interrupted => {}
                Err(e) => return Err(e),
            }
        }
        //已经完全写入
        Ok(())
    }

    //写入格式化内容
    fn write_fmt(&mut self, fmt: fmt::Arguments<'_>) -> Result<()> {
        struct Adapter<'a, T: ?Sized + 'a> {
            inner: &'a mut T,
            error: Result<()>,
        }

        impl<T: Write + ?Sized> fmt::Write for Adapter<'_, T> {
            fn write_str(&mut self, s: &str) -> fmt::Result {
                match self.inner.write_all(s.as_bytes()) {
                    Ok(()) => Ok(()),
                    Err(e) => {
```

```
                        self.error = Err(e);
                        Err(fmt::Error)
                    }
                }
            }
        }

        let mut output = Adapter { inner: self, error: Ok(()) };
        //见“智能指针(四)”中关于fmt的章节
        match fmt::write(&mut output, fmt) {
            Ok(()) => Ok(()),
            Err(..) => {
                // check if the error came from the underlying `Write` or not
                if output.error.is_err() {
                    output.error
                } else {
                    Err(error::const_io_error!(ErrorKind::Uncategorized,
"formatter error"))
                }
            }
        }
    }

    //获取输入目的类型变量的引用
    fn by_ref(&mut self) -> &mut Self
    where
        Self: Sized,
    {
        self
    }
}
```

io::sys::unix::Stdout的Write的实现

```
//路径: library/std/src/sys/unix/stdio.rs
impl io::Write for Stdout {
    fn write(&mut self, buf: &[u8]) -> io::Result<usize> {
        //标准输出文件不必关闭，因此不能调用FileDesc的drop
        unsafe {
ManuallyDrop::new(FileDesc::from_raw_fd(libc::STDOUT_FILENO)).write(buf) }
    }

    fn write_vectored(&mut self, bufs: &[IoSlice<'_>]) -> io::Result<usize> {
        unsafe {

ManuallyDrop::new(FileDesc::from_raw_fd(libc::STDOUT_FILENO)).write_vectored(bu
fs)
        }
    }
```

```
    fn is_write_vectored(&self) -> bool {
        true
    }

    fn flush(&mut self) -> io::Result<()> {
        Ok(())
    }
}
```

针对StdoutRaw的Read trait实现：

```
//即sys::Stdout的Write trait的adapter
impl Write for StdoutRaw {
    fn write(&mut self, buf: &[u8]) -> io::Result<usize> {
        handle_ebadf(self.0.write(buf), buf.len())
    }

    fn write_vectored(&mut self, bufs: &[IoSlice<'_>]) -> io::Result<usize> {
        let total = bufs.iter().map(|b| b.len()).sum();
        handle_ebadf(self.0.write_vectored(bufs), total)
    }

    fn is_write_vectored(&self) -> bool {
        self.0.is_write_vectored()
    }

    fn flush(&mut self) -> io::Result<()> {
        handle_ebadf(self.0.flush(), ())
    }

    fn write_all(&mut self, buf: &[u8]) -> io::Result<()> {
        handle_ebadf(self.0.write_all(buf), ())
    }

    fn write_all_vectored(&mut self, bufs: &mut [IoSlice<'_>]) ->
io::Result<()> {
        handle_ebadf(self.0.write_all_vectored(bufs), ())
    }

    fn write_fmt(&mut self, fmt: fmt::Arguments<'_>) -> io::Result<()> {
        handle_ebadf(self.0.write_fmt(fmt), ())
    }
}
```

BufWriter的相关实现：

```
impl<W: Write> BufWriter<W> {
    //将缓存中的内容输出到内部的输入目的IO对象变量中,即使在调用此方法前,
```

```rust
//缓存中的内容对调用者都被认为已经成功输出
pub(in crate::io) fn flush_buf(&mut self) -> io::Result<()> {
    struct BufGuard<'a> {
        //BufWriter中的buf
        buffer: &'a mut Vec<u8>,
        //写入内部输出目的IO对象的字节数
        written: usize,
    }

    impl<'a> BufGuard<'a> {
        fn new(buffer: &'a mut Vec<u8>) -> Self {
            Self { buffer, written: 0 }
        }

        /// The unwritten part of the buffer
        fn remaining(&self) -> &[u8] {
            &self.buffer[self.written..]
        }

        /// 记录已经输出的字节数
        fn consume(&mut self, amt: usize) {
            self.written += amt;
        }

        /// 是否所有的缓存数据都已经输出
        fn done(&self) -> bool {
            self.written >= self.buffer.len()
        }
    }

    impl Drop for BufGuard<'_> {
        fn drop(&mut self) {
            if self.written > 0 {
                //确保缓存删除已经输出的数据
                self.buffer.drain(..self.written);
            }
        }
    }

    //利用BufGuard来更新self.buf
    let mut guard = BufGuard::new(&mut self.buf);
    //缓存如果还有数据则一直循环
    while !guard.done() {
        //如果写的过程遇到线程panic,
        //此处提取做标记
        self.panicked = true;
        //调用内部输出目的对象完成写
        let r = self.inner.write(guard.remaining());
        //清除panic标记
        self.panicked = false;
```

```
            match r {
                //没有输出内容
                Ok(0) => {
                    //必须返回IO出错，因为缓存的内容已经被
                    //认为写成功
                    return Err(io::const_io_error!(
                        ErrorKind::WriteZero,
                        "failed to write the buffered data",
                    ));
                }
                //buffer中需要反映已经写出的内容
                Ok(n) => guard.consume(n),
                Err(ref e) if e.kind() == io::ErrorKind::Interrupted => {}
                Err(e) => return Err(e),
            }
        }
        Ok(())
    }

    //将字节切片写入缓存
    pub(super) fn write_to_buf(&mut self, buf: &[u8]) -> usize {
        //得到缓存空闲空间,并取空闲空间与切片长度小者
        let available = self.spare_capacity();
        let amt_to_buffer = available.min(buf.len());

        unsafe {
            //将字节切片写入缓存
            self.write_to_buffer_unchecked(&buf[..amt_to_buffer]);
        }

        //返回写入字节
        amt_to_buffer
    }

    //获得内部输出目的对象的引用
    pub fn get_ref(&self) -> &W {
        &self.inner
    }

    //获得内部输出目的对象的可变引用
    pub fn get_mut(&mut self) -> &mut W {
        &mut self.inner
    }

    //获得内部缓存的字节切片引用
    pub fn buffer(&self) -> &[u8] {
        &self.buf
    }

    //获得内部缓存的字节切片的可变引用
    pub(in crate::io) fn buffer_mut(&mut self) -> &mut Vec<u8> {
```

```
        &mut self.buf
    }

    //缓存总容量
    pub fn capacity(&self) -> usize {
        self.buf.capacity()
    }

    //消费self，获取内部输出目的IO对象
    pub fn into_inner(mut self) -> Result<W, IntoInnerError<BufWriter<W>>> {
        //需要先将缓存内容全部输出
        match self.flush_buf() {
            Err(e) => Err(IntoInnerError::new(self, e)),
            //此处获得self.buf的所有权，然后生命周期终结
            Ok(()) => Ok(self.into_parts().0),
        }
    }

    //into_inner的支持方法，处理panic
    pub fn into_parts(mut self) -> (W, Result<Vec<u8>, WriterPanicked>) {
        //获取self.buf的所有权
        let buf = mem::take(&mut self.buf);
        //将buf所有权返回，由调用者处理
        let buf = if !self.panicked { Ok(buf) } else { Err(WriterPanicked { buf
}) };

        //获取self.inner的所有权
        let inner = unsafe { ptr::read(&mut self.inner) };
        //不调用self的drop，防止对self.inner的drop调用
        mem::forget(self);

        (inner, buf)
    }

    #[cold]
    #[inline(never)]
    fn write_cold(&mut self, buf: &[u8]) -> io::Result<usize> {
        //判断输出是否大于缓存可用空间
        if buf.len() > self.spare_capacity() {
            //大于，则先将缓存输出
            self.flush_buf()?;
        }

        //再次判断是否大于缓存空间
        if buf.len() >= self.buf.capacity() {
            //大于,则直接输出到内部IO对象
            self.panicked = true;
            let r = self.get_mut().write(buf);
            self.panicked = false;
            r
        } else {
```

```
            //不大于，则输入到内部的缓存中
            unsafe {
                self.write_to_buffer_unchecked(buf);
            }

            Ok(buf.len())
        }
    }

    #[cold]
    #[inline(never)]
    fn write_all_cold(&mut self, buf: &[u8]) -> io::Result<()> {
        if buf.len() > self.spare_capacity() {
            self.flush_buf()?;
        }

        if buf.len() >= self.buf.capacity() {
            self.panicked = true;
            //直接调用的情况下必须用write_all
            let r = self.get_mut().write_all(buf);
            self.panicked = false;
            r
        } else {
            unsafe {
                self.write_to_buffer_unchecked(buf);
            }

            Ok(())
        }
    }

    unsafe fn write_to_buffer_unchecked(&mut self, buf: &[u8]) {
        debug_assert!(buf.len() <= self.spare_capacity());
        //就是直接做拷贝
        let old_len = self.buf.len();
        let buf_len = buf.len();
        let src = buf.as_ptr();
        let dst = self.buf.as_mut_ptr().add(old_len);
        ptr::copy_nonoverlapping(src, dst, buf_len);
        //拷贝完毕后设置内部的buf参数
        self.buf.set_len(old_len + buf_len);
    }

    fn spare_capacity(&self) -> usize {
        //获取内部可用空间
        self.buf.capacity() - self.buf.len()
    }
}


//实现Write trait
```

```rust
impl<W: Write> Write for BufWriter<W> {
    fn write(&mut self, buf: &[u8]) -> io::Result<usize> {
        //判断输出的字符切片长度是否小于缓存空间
        if buf.len() < self.spare_capacity() {
            //小于，直接写入缓存
            unsafe {
                self.write_to_buffer_unchecked(buf);
            }

            Ok(buf.len())
        } else {
            //否则，调用write_cold
            self.write_cold(buf)
        }
    }

    fn write_all(&mut self, buf: &[u8]) -> io::Result<()> {
        if buf.len() < self.spare_capacity() {
            unsafe {
                self.write_to_buffer_unchecked(buf);
            }

            Ok(())
        } else {
            //调用write_all_code
            self.write_all_cold(buf)
        }
    }

    fn write_vectored(&mut self, bufs: &[IoSlice<'_>]) -> io::Result<usize> {
        //判断内部IO对象是否支持向量写
        if self.get_ref().is_write_vectored() {
            //支持，
            // We have to handle the possibility that the total length of the
buffers overflows
            // `usize` (even though this can only happen if multiple `IoSlice`s
reference the
            // same underlying buffer, as otherwise the buffers wouldn't fit in
memory). If the
            // computation overflows, then surely the input cannot fit in our
buffer, so we forward
            // to the inner writer's `write_vectored` method to let it handle
it appropriately.
            //获取总的字节长度
            let saturated_total_len =
                bufs.iter().fold(0usize, |acc, b| acc.saturating_add(b.len()));

            //总字节长度是否大于可用空间
            if saturated_total_len > self.spare_capacity() {
                self.flush_buf()?;
            }
```

```
            //再次判断是否能用缓存
            if saturated_total_len >= self.buf.capacity() {
                //不能，则直接用write_vectored输出
                self.panicked = true;
                let r = self.get_mut().write_vectored(bufs);
                self.panicked = false;
                r
            } else {
                //否则，将内容写入缓存
                unsafe {
                    bufs.iter().for_each(|b|
self.write_to_buffer_unchecked(b));
                };

                Ok(saturated_total_len)
            }
        } else {
            //不支持向量写
            let mut iter = bufs.iter();
            //找到第一个不为空的IoSlice
            let mut total_written = if let Some(buf) =
iter.by_ref().find(|&buf| !buf.is_empty()) {
                if buf.len() > self.spare_capacity() {
                    self.flush_buf()?;
                }
                if buf.len() >= self.buf.capacity() {
                    // so it's better to write it directly, bypassing the
buffer.
                    self.panicked = true;
                    let r = self.get_mut().write(buf);
                    self.panicked = false;
                    //return的情况下，不必保证两个分支
                    //类型一致
                    return r;
                } else {
                    unsafe {
                        //写入buf
                        self.write_to_buffer_unchecked(buf);
                    }

                    buf.len()
                }
            } else {
                return Ok(0);
            };
            debug_assert!(total_written != 0);
            //iter位置已经移动
            for buf in iter {
                //判断buf是否有足够空间
                if buf.len() <= self.spare_capacity() {
```

```
                    unsafe {
                        self.write_to_buffer_unchecked(buf);
                    }

                    total_written += buf.len();
                } else {
                    //没有，则结束
                    break;
                }
            }
            //返回已经写入的总长度
            Ok(total_written)
        }
    }

    fn is_write_vectored(&self) -> bool {
        true
    }

    fn flush(&mut self) -> io::Result<()> {
        //先完成buf写入内部IO对象，再调用内部Io对象的flush方法。
        self.flush_buf().and_then(|()| self.get_mut().flush())
    }
}

//Seek trait实现
impl<W: Write + Seek> Seek for BufWriter<W> {
    fn seek(&mut self, pos: SeekFrom) -> io::Result<u64> {
        //先将缓存内容写入
        self.flush_buf()?;
        //再调用底层IO对象的seek
        self.get_mut().seek(pos)
    }
}

impl<W: Write> Drop for BufWriter<W> {
    fn drop(&mut self) {
        if !self.panicked {
            //生命周期终止时将buf写入内部输出IO对象
            let _r = self.flush_buf();
        }
    }
}
```

LineWrite的所有实现如下：

```
impl<W: Write> LineWriter<W> {
    //获取内部IO对象的引用
    pub fn get_ref(&self) -> &W {
        self.inner.get_ref()
```

```
    }

    //获取内部IO对象的可变引用
    pub fn get_mut(&mut self) -> &mut W {
        self.inner.get_mut()
    }

    //消费掉self，获得内部IO对象的引用
    pub fn into_inner(self) -> Result<W, IntoInnerError<LineWriter<W>>> {
        self.inner.into_inner().map_err(|err| err.new_wrapped(|inner|
LineWriter { inner }))
    }
}

//Write trait实现， 对BufReader的Adapter
impl<W: Write> Write for LineWriter<W> {
    fn write(&mut self, buf: &[u8]) -> io::Result<usize> {
        LineWriterShim::new(&mut self.inner).write(buf)
    }

    fn flush(&mut self) -> io::Result<()> {
        self.inner.flush()
    }

    fn write_vectored(&mut self, bufs: &[IoSlice<'_>]) -> io::Result<usize> {
        LineWriterShim::new(&mut self.inner).write_vectored(bufs)
    }

    fn is_write_vectored(&self) -> bool {
        self.inner.is_write_vectored()
    }

    fn write_all(&mut self, buf: &[u8]) -> io::Result<()> {
        LineWriterShim::new(&mut self.inner).write_all(buf)
    }

    fn write_all_vectored(&mut self, bufs: &mut [IoSlice<'_>]) ->
io::Result<()> {
        LineWriterShim::new(&mut self.inner).write_all_vectored(bufs)
    }

    fn write_fmt(&mut self, fmt: fmt::Arguments<'_>) -> io::Result<()> {
        LineWriterShim::new(&mut self.inner).write_fmt(fmt)
    }
}

//支持类型LineWriterShim
pub struct LineWriterShim<'a, W: Write> {
    buffer: &'a mut BufWriter<W>,
}
```

```rust
impl<'a, W: Write> LineWriterShim<'a, W> {
    pub fn new(buffer: &'a mut BufWriter<W>) -> Self {
        Self { buffer }
    }

    fn inner(&self) -> &W {
        self.buffer.get_ref()
    }

    fn inner_mut(&mut self) -> &mut W {
        self.buffer.get_mut()
    }

    fn buffered(&self) -> &[u8] {
        self.buffer.buffer()
    }

    fn flush_if_completed_line(&mut self) -> io::Result<()> {
        //判断buf的尾部字符
        match self.buffered().last().copied() {
            //如果是行结束符，将缓存输出
            Some(b'\n') => self.buffer.flush_buf(),
            _ => Ok(()),
        }
    }
}

impl<'a, W: Write> Write for LineWriterShim<'a, W> {
    fn write(&mut self, buf: &[u8]) -> io::Result<usize> {
        //查找buf中的分行符
        let newline_idx = match memchr::memrchr(b'\n', buf) {
            //没找到
            None => {
                //将buf中可能的行输出
                self.flush_if_completed_line()?;
                //将内容写入缓存
                return self.buffer.write(buf);
            }
            //找到，设置新行的启动位置
            Some(newline_idx) => newline_idx + 1,
        };

        //输出缓存内容
        self.buffer.flush_buf()?;

        //获取本行的内容
        let lines = &buf[..newline_idx];

        //本行内容直接写入底层的IO对象
        let flushed = self.inner_mut().write(lines)?;
```

```
        //判断是否写入了内容
        if flushed == 0 {
            //没有写入，返回0
            return Ok(0);
        }

        //判断本行是否完全输出，tail为需要写入缓存的字节切片
        let tail = if flushed >= newline_idx {
            //完全输出,tail为没有输出的字节切片
            &buf[flushed..]
        } else if newline_idx - flushed <= self.buffer.capacity() {
            //没有完全输出，且未输出的内容小于缓存容量
            //tail为本行没有输出的内容
            &buf[flushed..newline_idx]
        } else {
            //没有完全输出，且未输出内容大于缓存容量
            //tail为本行缓存容量的字节切片
            let scan_area = &buf[flushed..];
            let scan_area = &scan_area[..self.buffer.capacity()];
            match memchr::memrchr(b'\n', scan_area) {
                Some(newline_idx) => &scan_area[..newline_idx + 1],
                None => scan_area,
            }
        };

        //将tail写入缓存
        let buffered = self.buffer.write_to_buf(tail);
        Ok(flushed + buffered)
    }

    //adapter
    fn flush(&mut self) -> io::Result<()> {
        self.buffer.flush()
    }

    fn write_vectored(&mut self, bufs: &[IoSlice<'_>]) -> io::Result<usize> {
        //判断是否支持向量读写
        if !self.is_write_vectored() {
            //不支持，将第一个向量写入self并返回
            return match bufs.iter().find(|buf| !buf.is_empty()) {
                Some(buf) => self.write(buf),
                None => Ok(0),
            };
        }

        //找到buf中最后一个换行符
        //最后一个换行符前的数据都可以输出，
        //符合行输出的规则
        let last_newline_buf_idx = bufs
            .iter()
            .enumerate()
```

```
            .rev()
            .find_map(|(i, buf)| memchr::memchr(b'\n', buf).map(|_| i));


        let last_newline_buf_idx = match last_newline_buf_idx {
            //没有换行符
            None => {
                //先输出可能的行
                self.flush_if_completed_line()?;
                //将bufs写入内部缓存
                return self.buffer.write_vectored(bufs);
            }
            //有，获得切片下标
            Some(i) => i,
        };

        //先将缓存已有内容输出
        self.buffer.flush_buf()?;

        //将切片分为带输出及存入缓存的两个部分
        let (lines, tail) = bufs.split_at(last_newline_buf_idx + 1);

        //直接调用底层IO对象将内容输出,此处认为换行应该是IoSlice的尾部?
        let flushed = self.inner_mut().write_vectored(lines)?;

        //如果无法输出
        if flushed == 0 {
            //通知调用者
            return Ok(0);
        }

        //判断是否全部的slice都已经输出
        let lines_len = lines.iter().map(|buf| buf.len()).sum();
        if flushed < lines_len {
            //没有，则返回输出长度
            return Ok(flushed);
        }

        //将剩余的内容写入缓存中
        let buffered: usize = tail
            .iter()
            .filter(|buf| !buf.is_empty())
            .map(|buf| self.buffer.write_to_buf(buf))
            .take_while(|&n| n > 0)
            .sum();

        Ok(flushed + buffered)
    }

    fn is_write_vectored(&self) -> bool {
        self.inner().is_write_vectored()
```

```
    }

    fn write_all(&mut self, buf: &[u8]) -> io::Result<()> {
        match memchr::memrchr(b'\n', buf) {
            //没有换行符
            None => {
                //将buffer的行输出，然后将内容写入缓存
                self.flush_if_completed_line()?;
                self.buffer.write_all(buf)
            }
            //有换行符
            Some(newline_idx) => {
                //将换行符前的内容都输出，换行符后的内容写入缓存
                let (lines, tail) = buf.split_at(newline_idx + 1);

                if self.buffered().is_empty() {
                    self.inner_mut().write_all(lines)?;
                } else {
                    self.buffer.write_all(lines)?;
                    self.buffer.flush_buf()?;
                }

                self.buffer.write_all(tail)
            }
        }
    }
}
```

以上完成了所有RUST的对外类型结构Stdout的相关类型的代码分析。现在回到Stdout本身： 再次看一下Stdout的类型结构的相关实现：

```
//路径： library/std/src/io/stdio.rs
pub struct Stdout {
    //可重入的内部可变性类型
    //LineWriter是缓存类型结构
    inner: Pin<&'static ReentrantMutex<RefCell<LineWriter<StdoutRaw>>>>,
}
//Mutex返回的借用结构
pub struct StdoutLock<'a> {
    inner: ReentrantMutexGuard<'a, RefCell<LineWriter<StdoutRaw>>>,
}

//保证Stdout在一个进程中只初始化一次
static STDOUT: SyncOnceCell<ReentrantMutex<RefCell<LineWriter<StdoutRaw>>>> =
SyncOnceCell::new();

pub fn stdout() -> Stdout {
    Stdout {
        //如果没有初始化，则进行初始化，如果已经初始化，则加锁，获得引用
        //并构建Stdout
```

```
        inner: Pin::static_ref(&STDOUT).get_or_init_pin(
            || unsafe {
ReentrantMutex::new(RefCell::new(LineWriter::new(stdout_raw()))) },
            |mutex| unsafe { mutex.init() },
        ),
    }
}

pub fn cleanup() {
    if let Some(instance) = STDOUT.get() {
        if let Some(lock) = Pin::static_ref(instance).try_lock() {
            //将前个LineWriter做drop操作
            //生成一个缓存空间为0的LineWriter
            *lock.borrow_mut() = LineWriter::with_capacity(0, stdout_raw());
        }
    }
}

impl Stdout {
    //获得一个借用
    pub fn lock(&self) -> StdoutLock<'static> {
        //创建StdoutLock
        StdoutLock { inner: self.inner.lock() }
    }
}

//Write trait实现,
//是StdoutLock的adapter
//但是有一个线程安全操作
impl Write for &Stdout {
    fn write(&mut self, buf: &[u8]) -> io::Result<usize> {
        self.lock().write(buf)
    }
    fn write_vectored(&mut self, bufs: &[IoSlice<'_>]) -> io::Result<usize> ;
    fn is_write_vectored(&self) -> bool ;
    fn flush(&mut self) -> io::Result<()> ;
    fn write_all(&mut self, buf: &[u8]) -> io::Result<()> ;
    fn write_all_vectored(&mut self, bufs: &mut [IoSlice<'_>]) ->
io::Result<()> ;
    fn write_fmt(&mut self, args: fmt::Arguments<'_>) -> io::Result<()> ;
}

//LineWriter的adapter
impl Write for StdoutLock<'_> {
    fn write(&mut self, buf: &[u8]) -> io::Result<usize> {
        self.inner.borrow_mut().write(buf)
    }
    fn write_vectored(&mut self, bufs: &[IoSlice<'_>]) -> io::Result<usize> ;
    fn is_write_vectored(&self) -> bool ;
    fn flush(&mut self) -> io::Result<()> ;
    fn write_all(&mut self, buf: &[u8]) -> io::Result<()> ;
```

```
    fn write_all_vectored(&mut self, bufs: &mut [IoSlice<'_>]) ->
io::Result<()> ;
}
```

Stderr与Stdout类似，请自行分析。

以下可以作为stdout的使用例。是测试时使用的线程局部信息输出流的实现。

```
//线程局部流
type LocalStream = Arc<Mutex<Vec<u8>>>;

//线程局部流定义
thread_local! {
    static OUTPUT_CAPTURE: Cell<Option<LocalStream>> = {
        Cell::new(None)
    }
}

//已经使用了OUTPUT_CAPTURE机制
static OUTPUT_CAPTURE_USED: AtomicBool = AtomicBool::new(false);

//设置输出的缓存
pub fn set_output_capture(sink: Option<LocalStream>) -> Option<LocalStream> {
    if sink.is_none() && !OUTPUT_CAPTURE_USED.load(Ordering::Relaxed) {
        // OUTPUT_CAPTURE is definitely None since OUTPUT_CAPTURE_USED is
false.
        return None;
    }
    OUTPUT_CAPTURE_USED.store(true, Ordering::Relaxed);
    OUTPUT_CAPTURE.with(move |slot| slot.replace(sink))
}

//测试信息打印函数
fn print_to<T>(args: fmt::Arguments<'_>, global_s: fn() -> T, label: &str)
where
    T: Write,
{
    if OUTPUT_CAPTURE_USED.load(Ordering::Relaxed)
        && OUTPUT_CAPTURE.try_with(|s| {
            s.take().map(|w| {
                let _ = w.lock().unwrap_or_else(|e|
e.into_inner()).write_fmt(args);
                s.set(Some(w));
            })
        }) == Ok(Some(()))
    {
        // Successfully wrote to capture buffer.
        return;
    }
```

```
    if let Err(e) = global_s().write_fmt(args) {
        panic!("failed printing to {label}: {e}");
    }
}

//向标准输出打印
pub fn _print(args: fmt::Arguments<'_>) {
    print_to(args, stdout, "stdout");
}
//向标准错误打印
pub fn _eprint(args: fmt::Arguments<'_>) {
    print_to(args, stderr, "stderr");
}
```

网络IO

因为异步编程成为网络IO的共识，而异步框架如tokio等对网络IO库做了重写，标准库中的网络IO显得不再那么重要，本书将省略网络IO的分析。