std库文件描述符代码分析

以linux为例，文件描述符实际上是操作系统所有资源的标识。也是对其他模块分析的一个基础。
RUST文件描述符管理的结构设计：

1. 对操作系统文件描述符的适配层。对于RUST来说，操作系统的文件描述符与获取的堆内存指针要处理的安全性类似。需要建立类似智能指针的结构完成对其的管理，从而纳入到RUST的安全体系内。
2. RUST自身的文件描述符类型结构，在1中最重要的是解决安全问题，缺乏文件的逻辑功能。文件的逻辑功能在RUST的文件描述符类型的方法中进行了实现。仍然是操作系统的适配层。
3. 在2的基础上，实现普通的文件，目录文件，Socket，Pipe，IO设备文件等逻辑文件类型。

本章将讨论1及2，3以后在涉及到各模块时再进行详细分析
代码目录： library/src/std/src/os/fd/raw.rs
library/src/std/src/os/fd/owned.rs
library/src/std/src/sys/unix/fd.rs

操作系统的文件描述符的所有权设计

RUST当然要使用操作系统调用返回的fd来操作文件，fd在RUST中被重定义为RawFd类型。不同系统的RawFd可能不一样，但类型名称都是RawFd。
可以把RawFd按照裸指针来理解，RawFd不能作为所有权的载体，但RUST中文件显然需要具备所有权，因此，RUST在RawFd上定义了封装类型OwnedFd来实现针对RawFd的所有权，又定义了类型BorrowedFd作为OwnedFd的借用类型。
理解这两个类型，我们可以把RawFd类比与裸指针* const T， OwnedFd类比于 T，BorrowedFd类比于&T。

打开一个文件后(其他模块定义)，RUST底层会用系统返回的RawFd(fd)类型变量创建OwnedFd并存储到适当的类型中。当需要调用存在fd的系统调用时，从OwnedFd获取RawFd作为参数完成系统调用。

以linux操作系统为基础进行分析：

```
//虽然是int型，但因为表示操作系统资源，所以可以类比于裸指针。
//后继被标识文件所有权的封装类型所封装后才能进入安全的RUST领域。
pub type RawFd = raw::c_int;

//此trait用于从封装RawFd的类型中获取RawFd,
```

```rust
//此时返回的RawFd安全性类似于裸指针。
pub trait AsRawFd {
    fn as_raw_fd(&self) -> RawFd;
}

//从RawFd创建一个封装类型,返回的Self获得了RawFd代表的文件的所有权
pub trait FromRawFd {
    unsafe fn from_raw_fd(fd: RawFd) -> Self;
}

//将封装类型变量消费掉，并返回RawFd，此时RUST中没有其他变量拥有RawFd代表文件的所有权，
后继要由RawFd对close负责，或者将RawFd重新封装入另一个表示所有权的封装类型变量。
pub trait IntoRawFd {
    fn into_raw_fd(self) -> RawFd;
}

//获取标准输入的RawFd
impl AsRawFd for io::Stdin {
    fn as_raw_fd(&self) -> RawFd {
        //libc的标准输入文件标识宏
        libc::STDIN_FILENO
    }
}

//标准输出的RawFd
impl AsRawFd for io::Stdout {
    fn as_raw_fd(&self) -> RawFd {
        //libc的标准输出宏
        libc::STDOUT_FILENO
    }
}

//标准错误的RawFd
impl AsRawFd for io::Stderr {
    fn as_raw_fd(&self) -> RawFd {
        //libc的标准错误宏
        libc::STDERR_FILENO
    }
}
```

拥有RawFd所有权的OwnedFd类型结构及OwnedFd的借用类型结构BorrowedFd。

```rust
#[repr(transparent)]
pub struct BorrowedFd<'fd> {
    fd: RawFd,
    //用OwnedFd作为RawFd的所有权版本，RawFd实际上可认为是对OwnedFd的借用。
    //但仅用fd无法表达出生命周期和借用关系，
    //这里的PhantomData用OwnedFd的引用及生命周期泛型表示了这个关系
    _phantom: PhantomData<&'fd OwnedFd>,
}
```

```rust
#[repr(transparent)]
pub struct OwnedFd {
    //这个封装仅是一个形式，编译器并没有认为OwnedFd已经拥有了fd所代表文件的所有权。
    //所以，OwnedFd拥有所有权这个事情实际上是代码约定，其他代码务必不能导致用RawFd创建另一份
    //OwnedFd，也不能另外调用fd的close。
    //调用操作系统的系统调用获得文件Fd后，应该第一时间用OwnedFd进行封装，后继如果要使用，则应该
    //用borrow的方法来借出BorrowedFd，
    fd: RawFd,
}

impl BorrowedFd<'_> {
    //直接在RawFd上生成BorrowFd，这个函数主要用于不适合获得rawfd的所有权，
    //但需要借用fd时。
    //例如:标准输入/输出/错误， 或者从C语言调用传入的参数fd，不允许关闭，
    //如果在这些rawfd的基础上生成OwnedFd，会导致他们被错误关闭
    pub unsafe fn borrow_raw(fd: RawFd) -> Self {
        assert_ne!(fd, u32::MAX as RawFd);
        //这里的PhantomData的赋值令人疑惑，只能认为是编译器的魔术了
        unsafe { Self { fd, _phantom: PhantomData } }
    }
}

impl OwnedFd {
    //复制，这里是在操作系统内部复制了一个新的fd，需要调用系统调用完成
    pub fn try_clone(&self) -> crate::io::Result<Self> {
        // 设置复制的功能设定标志
        let cmd = libc::F_DUPFD_CLOEXEC;

        //调用libc库完成复制，返回新的fd
        let fd = cvt(unsafe { libc::fcntl(self.as_raw_fd(), cmd, 0) })?;
        //用新的fd创建新的Owned变量
        Ok(unsafe { Self::from_raw_fd(fd) })
    }
}

impl AsRawFd for BorrowedFd<'_> {
    //此方法应该尽量仅用于调用系统调用的时候使用
    fn as_raw_fd(&self) -> RawFd {
        self.fd
    }
}

impl AsRawFd for OwnedFd {
    //此方法应该尽量仅用于调用系统调用时使用
    fn as_raw_fd(&self) -> RawFd {
        self.fd
    }
}
```

```rust
impl IntoRawFd for OwnedFd {
    fn into_raw_fd(self) -> RawFd {
        let fd = self.fd;
        //必须forget，否则会触发drop调用close(fd)
        forget(self);
        fd
    }
}

impl FromRawFd for OwnedFd {
    //应该只能用这个方法创建OwnedFd
    unsafe fn from_raw_fd(fd: RawFd) -> Self {
        assert_ne!(fd, u32::MAX as RawFd);
        unsafe { Self { fd } }
    }
}

//这个方法证明OwnedFd拥有了操作系统返回的fd的所有权
impl Drop for OwnedFd {
    fn drop(&mut self) {
        unsafe {
            let _ = libc::close(self.fd);
        }
    }
}

//对OwnedFd创建借用的trait
pub trait AsFd {
    fn as_fd(&self) -> BorrowedFd<'_>;
}

impl AsFd for OwnedFd {
    fn as_fd(&self) -> BorrowedFd<'_> {
        //BorrowedFd中的PhantomData从&self中获得
        unsafe { BorrowedFd::borrow_raw(self.as_raw_fd()) }
    }
}
//以下为所有的高层视角资源生成OwnedFd的借用
impl AsFd for fs::File {
    fn as_fd(&self) -> BorrowedFd<'_> {
        //实质是OwnedFd.as_fd
        self.as_inner().as_fd()
    }
}

impl From<fs::File> for OwnedFd {
    fn from(file: fs::File) -> OwnedFd {
        //消费了File
        file.into_inner().into_inner().into_inner()
        //此处不涉及对file的forget
```

```
    }
}

impl From<OwnedFd> for fs::File {
    fn from(owned_fd: OwnedFd) -> Self {
        //创建fs::File

Self::from_inner(FromInner::from_inner(FromInner::from_inner(owned_fd)))
    }
}

impl AsFd for crate::net::TcpStream {
    fn as_fd(&self) -> BorrowedFd<'_> {
        //socket在unix与fd没有区别，也使用OwnedFd和BorrowedFd来做所有权的解决方案
        self.as_inner().socket().as_fd()
    }
}

impl From<crate::net::TcpStream> for OwnedFd {
    fn from(tcp_stream: crate::net::TcpStream) -> OwnedFd {
        //消费掉tcp_stream，具体在tcp stream分析
        tcp_stream.into_inner().into_socket().into_inner().into_inner().into()
    }
}

impl From<OwnedFd> for crate::net::TcpStream {
    fn from(owned_fd: OwnedFd) -> Self {
        //后继在TcpStream章节分析

Self::from_inner(FromInner::from_inner(FromInner::from_inner(FromInner::from_in
ner(
            owned_fd,
        ))))
    }
}

impl AsFd for crate::net::TcpListener {
    fn as_fd(&self) -> BorrowedFd<'_> {
        //同TcpStream
        self.as_inner().socket().as_fd()
    }
}

impl From<crate::net::TcpListener> for OwnedFd {
    fn from(tcp_listener: crate::net::TcpListener) -> OwnedFd {

tcp_listener.into_inner().into_socket().into_inner().into_inner().into()
    }
}

impl From<OwnedFd> for crate::net::TcpListener {
```

```
    fn from(owned_fd: OwnedFd) -> Self {

Self::from_inner(FromInner::from_inner(FromInner::from_inner(FromInner::from_in
ner(
            owned_fd,
        ))))
    }
}

impl AsFd for crate::net::UdpSocket {
    fn as_fd(&self) -> BorrowedFd<'_> {
        //UDP与TCP类似
        self.as_inner().socket().as_fd()
    }
}

impl From<crate::net::UdpSocket> for OwnedFd {
    fn from(udp_socket: crate::net::UdpSocket) -> OwnedFd {
        udp_socket.into_inner().into_socket().into_inner().into_inner().into()
    }
}

impl From<OwnedFd> for crate::net::UdpSocket {
    fn from(owned_fd: OwnedFd) -> Self {

Self::from_inner(FromInner::from_inner(FromInner::from_inner(FromInner::from_in
ner(
            owned_fd,
        ))))
    }
}
```

对于需要调用RUST以外语言实现的第三方库时，都会面临一个从第三方库获取的资源如何在RUST设计其所有权的问题。Unix的fd的方案给出了一个经典的设计方式。即把第三方库获取的资源在逻辑上类似于裸指针，如RawFd。然后用一个封装结构封装用于所有权实现，例如OwnedFd。用另一个封装结构用作借用，例如BorrowedFd。这个设计方案在真正的生产环境中会经常被用到。

RUST标准库文件描述符的结构与实现

在OwnedFd的基础上创建的结构，是文件操作的操作系统适配层的实现。以下代码分析linux操作系统的RUST文件描述符模块实现：

```
//RUST的文件描述符类型结构
pub struct FileDesc(OwnedFd);

impl FileDesc {
```

```rust
    //从文件描述符读出字节流
    pub fn read(&self, buf: &mut [u8]) -> io::Result<usize> {
        let ret = cvt(unsafe {
            //调用libc的read函数
            libc::read(
                //按C语言的调用进行转换
                self.as_raw_fd(),
                //转换成void *指针
                buf.as_mut_ptr() as *mut c_void,
                //不能超过buf，也不能超过一次读的最大长度
                cmp::min(buf.len(), READ_LIMIT),
            )
        })?;
        Ok(ret as usize)
    }

    //对应于libc的iovec读的方式,具体请参考libc的说明
    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)
    }

    //一直读到文件结束
    pub fn read_to_end(&self, buf: &mut Vec<u8>) -> io::Result<usize> {
        let mut me = self;
        (&mut me).read_to_end(buf)
    }

    //从文件的某一个位置开始读，请参考libc的pread64的man
    pub fn read_at(&self, buf: &mut [u8], offset: u64) -> io::Result<usize> {
        use libc::pread64;

        unsafe {
            cvt(pread64(
                self.as_raw_fd(),
                buf.as_mut_ptr() as *mut c_void,
                cmp::min(buf.len(), READ_LIMIT),
                offset as i64,
            ))
            .map(|n| n as usize)
        }
    }

    //读到buffer中的某一个位置
```

```rust
    pub fn read_buf(&self, buf: &mut ReadBuf<'_>) -> io::Result<()> {
        let ret = cvt(unsafe {
            libc::read(
                self.as_raw_fd(),
                buf.unfilled_mut().as_mut_ptr() as *mut c_void,
                cmp::min(buf.remaining(), READ_LIMIT),
            )
        })?;

        //原有的空间是MaybeUninit，读到内容后需要进行初始化标注
        unsafe {
            buf.assume_init(ret as usize);
        }
        //更新内容长度
        buf.add_filled(ret as usize);
        Ok(())
    }

    //向文件描述符写入字节流
    pub fn write(&self, buf: &[u8]) -> io::Result<usize> {
        let ret = cvt(unsafe {
            libc::write(
                self.as_raw_fd(),
                buf.as_ptr() as *const c_void,
                cmp::min(buf.len(), READ_LIMIT),
            )
        })?;
        Ok(ret as usize)
    }

    //iovec的方式写入
    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)
    }

    //在文件的某一位置写入字节流
    pub fn write_at(&self, buf: &[u8], offset: u64) -> io::Result<usize> {
        use libc::pwrite64;

        unsafe {
            cvt(pwrite64(
                self.as_raw_fd(),
                buf.as_ptr() as *const c_void,
                cmp::min(buf.len(), READ_LIMIT),
```

```
                offset as i64,
            ))
            .map(|n| n as usize)
        }
    }

    //获取FD_CLOEXEC，具体请参考libc的相关手册
    pub fn get_cloexec(&self) -> io::Result<bool> {
        unsafe { Ok((cvt(libc::fcntl(self.as_raw_fd(), libc::F_GETFD))? &
libc::FD_CLOEXEC) != 0) }
    }

    //设置FD_CLOEXEC的属性，一般会在打开文件时完成设置，否则要注意不同线程竞争问题
    pub fn set_cloexec(&self) -> io::Result<()> {
        unsafe {
            let previous = cvt(libc::fcntl(self.as_raw_fd(), libc::F_GETFD))?;
            let new = previous | libc::FD_CLOEXEC;
            if new != previous {
                cvt(libc::fcntl(self.as_raw_fd(), libc::F_SETFD, new))?;
            }
            Ok(())
        }
    }

    //设置为非阻塞
    pub fn set_nonblocking(&self, nonblocking: bool) -> io::Result<()> {
        unsafe {
            let v = nonblocking as c_int;
            cvt(libc::ioctl(self.as_raw_fd(), libc::FIONBIO, &v))?;
            Ok(())
        }
    }

    //复制文件描述符
    pub fn duplicate(&self) -> io::Result<FileDesc> {
        Ok(Self(self.0.try_clone()?))
    }

    //后继可以加入其他需要的通用文件操作方法
}

impl AsInner<OwnedFd> for FileDesc {
    //不消费FileDesc获取内部引用
    fn as_inner(&self) -> &OwnedFd {
        &self.0
    }
}

impl IntoInner<OwnedFd> for FileDesc {
    fn into_inner(self) -> OwnedFd {
        //消费self，获得内部OwnedFd
```

```
        //不必做其他资源释放操作
        self.0
    }
}

impl FromInner<OwnedFd> for FileDesc {
    //从参数创建FileDesc类型变量
    fn from_inner(owned_fd: OwnedFd) -> Self {
        Self(owned_fd)
    }
}

impl AsFd for FileDesc {
    //创建一个引用
    fn as_fd(&self) -> BorrowedFd<'_> {
        self.0.as_fd()
    }
}

impl AsRawFd for FileDesc {
    //简化代码
    fn as_raw_fd(&self) -> RawFd {
        self.0.as_raw_fd()
    }
}

impl IntoRawFd for FileDesc {
    fn into_raw_fd(self) -> RawFd {
        //见OwnedFd::into_raw_fd
        self.0.into_raw_fd()
    }
}

impl FromRawFd for FileDesc {
    //见OwnedFd::from_raw_fd
    unsafe fn from_raw_fd(raw_fd: RawFd) -> Self {
        Self(FromRawFd::from_raw_fd(raw_fd))
    }
}
```

以上代码要细心体会RUST所有权及借用概念是如何在文件描述符来使用的。

std库进程管理代码分析

在服务器端，服务器程序经常需要创建进程以完成一些任务。在客户端进程管理最显著的应用场景是操作系统的shell。另外，在目前JS成为主导地位的界面编程中，一个桌面应用分为前端进程及后端进程也是一种可行的架构。

描述进程管理的需求以一个linux的shell命令比较合适： 例如： cat 序言.md | more 在shell程序执行这条命令时，做了以下的工作：

1. 创建1个管道
2. fork第一个子进程
3. 指定第一个子进程的标准输入是管道的读出端
4. 第一个子进程用execv执行more的可执行文件
5. fork第二个子进程
6. 第二个子进程用execv执行cat可执行文件, "序言.md"作为参数
7. wait两个子进程结束 上例如果用RUST来实现，代码简略如下：

```
    let child_more = Command::new("more")
                          .stdin(Stdio::piped())
                          .spawn()
                          .expect("error more");
    let child_cat = Command::new("cat")
                         .arg("引言.md")
                         .stdout(child_more.stdin.unwrap())
                         .spawn().expect("cat error");
```

可以看到，RUST代码相当简单及易于理解。注意这里不要与C语言的system函数相比较，system实际上创建一个shell进程执行system的输入命令。属于在以上所述的基础在上一层的逻辑层面。
RUST进程管理的任务基本如上所述。
RUST进程管理在操作系统适配层涉及到：

1. 匿名管道
2. 进程管理

适配层扩展主要是标准输入/输出/错误及重定向的RUST实现

RUST标准库对外提供的进程管理是Command及其的方法和函数。

操作系统无关的代码路径：library/src/std/src/process.rs
library/src/std/src/syscommon/process.rs 操作系统相关的代码路径：
library/src/std/src/sys/unix/process/*
library/src/std/src/os/linux/process.rs
library/src/std/src/sys/unix/pipe.rs

匿名管道

匿名管道被设计用来在父子进程或者同一个父进程创建的子进程之间进行通信。一般只用于标准输入及输出的重定向。以下是Linux的实现。

```rust
//匿名管道的资源用文件描述符表示
pub struct AnonPipe(FileDesc);

//创建管道的函数，管道只能通过这个函数完成创建
pub fn anon_pipe() -> io::Result<(AnonPipe, AnonPipe)> {
    //匿名管道会创建两个文件描述符
    let mut fds = [0; 2];

    unsafe {
        //pipe2系统调用，fds的类型RUST做了推断，O_CLOEXEC表示后继exec调用的时候会自动close
        cvt(libc::pipe2(fds.as_mut_ptr(), libc::O_CLOEXEC))?;
        //返回两个匿名管道,AnonPipe生命周期终结的时候会close此处创建的fd，返回的第一个管道是读出，
        // 第二个管道是写入
        Ok((AnonPipe(FileDesc::from_raw_fd(fds[0])),
AnonPipe(FileDesc::from_raw_fd(fds[1]))))
    }
}
//FileDesc的adapter
//管道创建以后的读写操作
impl AnonPipe {
    pub fn read(&self, buf: &mut [u8]) -> io::Result<usize> {
        //就是对内部FileDesc的同名函数调用，
        //后继的函数逻辑也相同，代码省略
        self.0.read(buf)
    }

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

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

    pub fn write(&self, buf: &[u8]) -> io::Result<usize> {...}

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

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

//为了进程管理专门实现的函数，可以从中看一下RUST的多路通道读的程序
pub fn read2(p1: AnonPipe, v1: &mut Vec<u8>, p2: AnonPipe, v2: &mut Vec<u8>) ->
io::Result<()> {
    // 获取C调用的文件描述符
    let p1 = p1.into_inner();
    let p2 = p2.into_inner();
    //需要设置成非阻塞，后继用异步读的方式
    p1.set_nonblocking(true)?;
    p2.set_nonblocking(true)?;
```

```
    //准备libc的poll调用参数，具体的细节请参考libc的手册
    let mut fds: [libc::pollfd; 2] = unsafe { mem::zeroed() };
    fds[0].fd = p1.as_raw_fd();
    fds[0].events = libc::POLLIN;
    fds[1].fd = p2.as_raw_fd();
    fds[1].events = libc::POLLIN;
    loop {
        // poll调用，当任意一个fd有内容可读事件，则返回，否则阻塞
        cvt_r(|| unsafe { libc::poll(fds.as_mut_ptr(), 2, -1) })?;

        //fds[0]读到v1
        if fds[0].revents != 0 && read(&p1, v1)? {
            // 见下面read返回Ok(true)的说明，
            // 此函数的两个文件是联动的,读到其中一个即认为任务结束
            // 另一个也很快会输出，后继要阻塞等能读到p2为止
            p2.set_nonblocking(false)?;
            //因为返回是Result<()>类型，此处用drop将Result<usize>转换为
            // Result<()>
            return p2.read_to_end(v2).map(drop);
        }
        if fds[1].revents != 0 && read(&p2, v2)? {
            //见上
            p1.set_nonblocking(false)?;
            return p1.read_to_end(v1).map(drop);
        }
    }

    fn read(fd: &FileDesc, dst: &mut Vec<u8>) -> Result<bool, io::Error> {
        //一直读完
        match fd.read_to_end(dst) {
            //读到内容
            Ok(_) => Ok(true),
            Err(e) => {
                if e.raw_os_error() == Some(libc::EWOULDBLOCK)
                    || e.raw_os_error() == Some(libc::EAGAIN)
                {
                    //没有读到内容，但实际没有出错
                    Ok(false)
                } else {
                    //读出错
                    Err(e)
                }
            }
        }
    }
    //必然会在前面返回。这里会对p1及p2做所有权释放，会close掉创建的管道文件
}
```

下节中既可以看到进程管理函数对pipe使用的例子。

标准输入输出重定向类型及实现

在pipe的基础上，对重定向实现的支持类型结构。

```
//当创建子进程时，需要对子进程的标准输入/输出/错误每一个的配置
//指定为此结构中的一种类型。
pub enum Stdio {
    //继承父进程的fd
    Inherit,
    //设置为Null
    Null,
    //创建匿名管道作为标准输入/输出/错误
    MakePipe,
    //标准输入/输出/错误使用给出的文件描述符
    Fd(FileDesc),
}

// 创建子进程时，父进程针对于子进程的标准输入/输出/错误的对应管道配置
// 此时管道已经创建
pub struct StdioPipes {
    //None表示不重定向，Some()表示重定向到匿名管道
    pub stdin: Option<AnonPipe>,
    pub stdout: Option<AnonPipe>,
    pub stderr: Option<AnonPipe>,
}

// 对于子进程的标准输入/输出/错误完成准备后，调用操作系统创建进程时，
// 对子进程的标准输入/输出/错误每一个指定为下面结构中的一种类型
pub enum ChildStdio {
    //继承父进程的标准输入输出错误fd
    Inherit,
    //设置fd为参数
    Explicit(c_int),
    //从RUST的FileDesc中的信息设置fd
    Owned(FileDesc),
}
//准备完毕后，此类型结构具有子进程的标准输入/输出/错误的完整设置
pub struct ChildPipes {
    pub stdin: ChildStdio,
    pub stdout: ChildStdio,
    pub stderr: ChildStdio,
}
//针对Stdio实现方法
impl Stdio {
    // 此方法是父进程与子进程的标准输入/输出/错误的准备方法，每次准备三者其一，
    // 此方法的输出会设置 StdioPipes及childPipes的变量
    pub fn to_child_stdio(&self, readable: bool) -> io::Result<(ChildStdio,
Option<AnonPipe>)> {
        match *self {
            //指定为继承父进程，子进程为继承父进程，父进程没有与之相关的管道
```

```
            Stdio::Inherit => Ok((ChildStdio::Inherit, None)),

            //配置为使用指定的文件描述符
            Stdio::Fd(ref fd) => {
                if fd.as_raw_fd() >= 0 && fd.as_raw_fd() <= libc::STDERR_FILENO
{
                    //如果指定的文件描述符是标准输入/输出/错误，则复制后返回Owned，
                    //父进程没有管道与子进程连接
                    Ok((ChildStdio::Owned(fd.duplicate()?), None))
                } else {
                    //子进程使用指定的文件描述符，此时因为不能获取FileDesc的所有权，
所以只能使用RawFd，父进程没有管道与之连接
                    Ok((ChildStdio::Explicit(fd.as_raw_fd()), None))
                }
            }

            //配置为创建管道连接父子进程
            Stdio::MakePipe => {
                //创建管道，
                let (reader, writer) = pipe::anon_pipe()?;
                //根据读写标志设置自身管道的文件描述符，readable为真指子进程是读方
                let (ours, theirs) = if readable { (writer, reader) } else {
(reader, writer) };
                //返回创建的管道描述符
                Ok((ChildStdio::Owned(theirs.into_inner()), Some(ours)))
            }

            //指定为dev/null
            Stdio::Null => {
                let mut opts = OpenOptions::new();
                opts.read(readable);
                opts.write(!readable);
                let path = unsafe { CStr::from_ptr(DEV_NULL.as_ptr() as *const
_) };
                //需要先打开/dev/null
                let fd = File::open_c(&path, &opts)?;
                //输出为/dev/null的fd
                Ok((ChildStdio::Owned(fd.into_inner()), None))
            }
        }
    }
}

impl From<AnonPipe> for Stdio {
    fn from(pipe: AnonPipe) -> Stdio {
        Stdio::Fd(pipe.into_inner())
    }
}

impl From<File> for Stdio {
    fn from(file: File) -> Stdio {
```

```
            Stdio::Fd(file.into_inner())
        }
    }
    //直接获取RawFd用于操作系统调用
    impl ChildStdio {
        pub fn fd(&self) -> Option<c_int> {
            match *self {
                ChildStdio::Inherit => None,
                ChildStdio::Explicit(fd) => Some(fd),
                ChildStdio::Owned(ref fd) => Some(fd.as_raw_fd()),
            }
        }
    }
    //以下是上面代码的一个使用的实例:
    impl Command {
        ......

        //setup_io作为创建子进程的标准输入/输出/错误的准备函数
        //形成父进程与子进程标准输入/输出/错误的必要的管道创建及
        // 配对返回
        pub fn setup_io(
            &self,
            default: Stdio,
            needs_stdin: bool,
        ) -> io::Result<(StdioPipes, ChildPipes)> {
            let null = Stdio::Null;
            let default_stdin = if needs_stdin { &default } else { &null };
            //没有配置的话就使用默认配置，stdin/stdout/stderr是Stdio类型变量
            let stdin = self.stdin.as_ref().unwrap_or(default_stdin);
            let stdout = self.stdout.as_ref().unwrap_or(&default);
            let stderr = self.stderr.as_ref().unwrap_or(&default);
            //创建标准输入的子进程文件对,their用于子进程，our用于本进程
            let (their_stdin, our_stdin) = stdin.to_child_stdio(true)?;
            //创建标准输出的子进程文件对
            let (their_stdout, our_stdout) = stdout.to_child_stdio(false)?;
            //创建标准错误的子进程文件对
            let (their_stderr, our_stderr) = stderr.to_child_stdio(false)?;
            //完成本进程的设置
            let ours = StdioPipes { stdin: our_stdin, stdout: our_stdout, stderr:
    our_stderr };
            //完成子进程的设置
            let theirs = ChildPipes { stdin: their_stdin, stdout: their_stdout,
    stderr: their_stderr };
            Ok((ours, theirs))
        }
    }
```

管道是类unix系统实现多个进程组合完成一个大的任务的有效特性。

进程管理

使用RUST的创建进程的代码举例如下：

```
use std::process::Command;

Command::new("ls")
        .arg("-l")
        .arg("-a")
        .stdout(Stdio::piped())
        .spawn()
        .expect("ls command failed to start");
```

可以看到，RUST把C语言库中分散的进程准备及执行相关的内容整体组织进了Command结构的实现中，并利用函数式编程的链式调用使其语法易于理解，从后面的实现中也可以看出Command对程序员是一个巨大的福利。 Command的具体使用方式请参考官方标准库文档获得指导。

RUST中操作系统适配层对其他模块提供的接口类型结构及实现：
RUST在进程管理的操作系统适配层的设计中，没有使用trait的设计方式。而是直接使用了各操作系统统一实现相同类型名称的类型，再对类型实现相同名字及参数的方法。
因为在编译中每种操作系统都会单独编译，所以这种方式方便有效，也是C语言跨操作系统经常采用的代码组织方法。

```
//以下以linux为例的Process进程类型结构，
//各操作系统针对Process的结构体成员定义可以不同，但必须是Process类型名并对Process实现同样的方法。
pub struct Process {
    //进程pid
    pid: pid_t,
    //退出的状态
    status: Option<ExitStatus>,
    // Linux上，每个process与一个fd相关联。
    #[cfg(target_os = "linux")]
    pidfd: Option<PidFd>,
}
//Process的方法实现
impl Process {
    //linux的创建方法，应该在fork函数调用以后才能调用此方法
    #[cfg(target_os = "linux")]
    unsafe fn new(pid: pid_t, pidfd: pid_t) -> Self {
        use crate::os::unix::io::FromRawFd;
        use crate::sys_common::FromInner;
        // Safety: If `pidfd` is nonnegative, we assume it's valid and otherwise unowned.
        let pidfd = (pidfd >= 0).then(||
PidFd::from_inner(sys::fd::FileDesc::from_raw_fd(pidfd)));
```

```
        Process { pid, status: None, pidfd }
    }

    //父进程调用此函数杀掉子进程
    pub fn kill(&mut self) -> io::Result<()> {
        // 假如子进程已经是退出状态，那子进程的pid可能已经分配给其他进程使用，此时需要
返回错误。
        if self.status.is_some() {
            Err(io::const_io_error!(
                ErrorKind::InvalidInput,
                "invalid argument: can't kill an exited process",
            ))
        } else {
            //libc库的kill调用
            cvt(unsafe { libc::kill(self.pid, libc::SIGKILL) }).map(drop)
        }
    }

    //父进程等待子进程结束，会引发阻塞
    pub fn wait(&mut self) -> io::Result<ExitStatus> {
        use crate::sys::cvt_r;

        //判断子进程是否已经退出
        if let Some(status) = self.status {
            return Ok(status);
        }

        let mut status = 0 as c_int;
        //调用libc的waitpid等待子进程结束
        cvt_r(|| unsafe { libc::waitpid(self.pid, &mut status, 0) })?;
        //子进程已经退出，设置合适的状态
        self.status = Some(ExitStatus::new(status));
        Ok(ExitStatus::new(status))
    }

    //非阻塞的等待子进程退出
    pub fn try_wait(&mut self) -> io::Result<Option<ExitStatus>> {
        if let Some(status) = self.status {
            return Ok(Some(status));
        }
        let mut status = 0 as c_int;
        //用libc::WNOHANG表示非阻塞
        let pid = cvt(unsafe { libc::waitpid(self.pid, &mut status,
libc::WNOHANG) })?;
        if pid == 0 {
            //没有退出
            Ok(None)
        } else {
            //已经退出
            self.status = Some(ExitStatus::new(status));
            Ok(Some(ExitStatus::new(status)))
```

```
        }
    }
}
```

因为wasi进程管理与unix类似，因此下面再给出windows的代码做一下对比：

```
pub struct Process {
    //windows句柄
    handle: Handle,
}

impl Process {
    pub fn kill(&mut self) -> io::Result<()> {
        //使用windows的系统调用
        cvt(unsafe { c::TerminateProcess(self.handle.as_raw_handle(), 1) })?;
        Ok(())
    }

    pub fn wait(&mut self) -> io::Result<ExitStatus> {
        unsafe {
            //windows使用如下调用来等待进程退出
            let res = c::WaitForSingleObject(self.handle.as_raw_handle(),
c::INFINITE);
            if res != c::WAIT_OBJECT_0 {
                return Err(Error::last_os_error());
            }
            let mut status = 0;
            //额外调用来获取退出码
            cvt(c::GetExitCodeProcess(self.handle.as_raw_handle(), &mut
status))?;
            Ok(ExitStatus(status))
        }
    }

    pub fn try_wait(&mut self) -> io::Result<Option<ExitStatus>> {
        unsafe {
            //不等待
            match c::WaitForSingleObject(self.handle.as_raw_handle(), 0) {
                c::WAIT_OBJECT_0 => {}
                c::WAIT_TIMEOUT => {
                    return Ok(None);
                }
                _ => return Err(io::Error::last_os_error()),
            }
            //获取退出码
            let mut status = 0;
            cvt(c::GetExitCodeProcess(self.handle.as_raw_handle(), &mut
status))?;
            Ok(Some(ExitStatus(status)))
        }
```

```
    }
}
```

由上可见，不同的操作系统实现了相同的类型名及类型方法。
其他进程管理相关的类型结构及方法实现：

```
//Command完成进程准备的所有参数，进程启动等，操作系统适配层及标准库对外接口都有Command名称的类型，
//要注意区别， 此处是操作系统适配层的实现。
pub struct Command {
    //进程的可执行文件名.
    program: CString,
    //进程的命令行参数,同上，是否支持中文?
    args: Vec<CString>,
    /// 传递给`execvp`的参数，第一个参数应该是`program`，然后是
    /// `args`，最后应该是`null`. 修改时需要注意这三个参数的联动性
    argv: Argv,
    env: CommandEnv,

    //当前目录
    cwd: Option<CString>,
    //unix的uid
    uid: Option<uid_t>,
    //unix的gid
    gid: Option<gid_t>,
    // 对CString参数的输入是否存在0做标识
    saw_nul: bool,
    closures: Vec<Box<dyn FnMut() -> io::Result<()> + Send + Sync>>,
    groups: Option<Box<[gid_t]>>,
    //标准输入配置
    stdin: Option<Stdio>,
    //标准输出配置
    stdout: Option<Stdio>,
    //标准错误配置
    stderr: Option<Stdio>,
    #[cfg(target_os = "linux")]
    create_pidfd: bool,
    pgroup: Option<pid_t>,
}

impl Command {
    pub fn new(program: &OsStr) -> Command {
        let mut saw_nul = false;
        //OsStr转换为CStr,并返回OsStr是否存在尾值0
        let program = os2c(program, &mut saw_nul);
        //用program创建默认的Command结构体
        Command {
            //argv尾部必须有一个null指针
            argv: Argv(vec![program.as_ptr(), ptr::null()]),
            args: vec![program.clone()],
```

```
            //同名参数赋值
            program,
            env: Default::default(),
            cwd: None,
            uid: None,
            gid: None,
            saw_nul,
            closures: Vec::new(),
            groups: None,
            stdin: None,
            stdout: None,
            stderr: None,
            create_pidfd: false,
            pgroup: None,
        }
    }

    //这个方法设置进程的执行文件名program作为第一个arg参数
    pub fn set_arg_0(&mut self, arg: &OsStr) {
        // Set a new arg0
        let arg = os2c(arg, &mut self.saw_nul);
        debug_assert!(self.argv.0.len() > 1);
        self.argv.0[0] = arg.as_ptr();
        self.args[0] = arg;
    }

    //此方法增加增加一个进程命令的参数
    pub fn arg(&mut self, arg: &OsStr) {
        let arg = os2c(arg, &mut self.saw_nul);
        //增加参数到argv
        self.argv.0[self.args.len()] = arg.as_ptr();
        self.argv.0.push(ptr::null());

        //增加参数到args
        self.args.push(arg);
    }

    //根据标准输入/输出/错误的配置完成动作
    pub fn setup_io(
        &self,
        default: Stdio,
        needs_stdin: bool,
    ) -> io::Result<(StdioPipes, ChildPipes)> {
        let null = Stdio::Null;
        let default_stdin = if needs_stdin { &default } else { &null };
        //没有配置的话就使用默认配置
        let stdin = self.stdin.as_ref().unwrap_or(default_stdin);
        let stdout = self.stdout.as_ref().unwrap_or(&default);
        let stderr = self.stderr.as_ref().unwrap_or(&default);
        //创建标准输入的子进程文件对,their用于子进程，our用于本进程
        let (their_stdin, our_stdin) = stdin.to_child_stdio(true)?;
```

```
        //创建标准输出的子进程文件对
        let (their_stdout, our_stdout) = stdout.to_child_stdio(false)?;
        //创建标准错误的子进程文件对
        let (their_stderr, our_stderr) = stderr.to_child_stdio(false)?;
        //完成本进程的设置
        let ours = StdioPipes { stdin: our_stdin, stdout: our_stdout, stderr:
our_stderr };
        //完成子进程的设置
        let theirs = ChildPipes { stdin: their_stdin, stdout: their_stdout,
stderr: their_stderr };
        Ok((ours, theirs))
    }
```

以下为创建进程的方法实现，从以下代码可见，在底层面向操作系统调用编程时，RUST可近似的认为C代码，复杂及容易出现问题。

```
    //创建进程的具体执行方法
    pub fn spawn(
        &mut self,
        default: Stdio,
        needs_stdin: bool,
    ) -> io::Result<(Process, StdioPipes)> {
        //父子进程结果通信的魔数
        const CLOEXEC_MSG_FOOTER: [u8; 4] = *b"NOEX";

        //完成环境变量创建
        let envp = self.capture_env();

        //命令行出错处理
        if self.saw_nul() {
            return Err(io::const_io_error!(
                ErrorKind::InvalidInput,
                "nul byte found in provided data",
            ));
        }

        //完成标准输入/输出/错误文件的创建与准备
        let (ours, theirs) = self.setup_io(default, needs_stdin)?;

        //利用posix的api来创建进程
        if let Some(ret) = self.posix_spawn(&theirs, envp.as_ref())? {
            return Ok((ret, ours));
        }

        //创建一个匿名管道,这个匿名管道用来捕捉exec的错误
        let (input, output) = sys::pipe::anon_pipe()?;

        //此时要对环境参数加锁
        let env_lock = sys::os::env_read_lock();
        //fork新的进程，新进程复制所有的老进程的参数和栈
```

```
        let (pid, pidfd) = unsafe { self.do_fork()? };

        if pid == 0 {
            //新创建的子进程，总是abort退出
            crate::panic::always_abort();
            //不用理会env_lock，父进程会处理,这个细节RUST和C是一样容易出错的
            mem::forget(env_lock);
            //子进程不使用input
            drop(input);
            //执行二进制可执行文件,执行成功不会返回，exec执行成功后,因为output设置了
FD_CLOEXEC,
            //所以output会被关闭
            let Err(err) = unsafe { self.do_exec(theirs, envp.as_ref()) };
            //exec执行失败,做错误处理
            let errno = err.raw_os_error().unwrap_or(libc::EINVAL) as u32;
            let errno = errno.to_be_bytes();
            let bytes = [
                errno[0],
                errno[1],
                errno[2],
                errno[3],
                CLOEXEC_MSG_FOOTER[0],
                CLOEXEC_MSG_FOOTER[1],
                CLOEXEC_MSG_FOOTER[2],
                CLOEXEC_MSG_FOOTER[3],
            ];
            // 将exec的错误写入管道，使得父进程能够获得
            // 这里是一个细致的考虑
            rtassert!(output.write(&bytes).is_ok());
            unsafe { libc::_exit(1) }
        }

        //对env_lock进行处理
        drop(env_lock);
        //不需要output，要显式drop
        drop(output);

        //根据fork的返回创建RUST的子进程结构
        let mut p = unsafe { Process::new(pid, pidfd) };
        let mut bytes = [0; 8];

        loop {
            //如果子进程exec成功，则管道对端会关闭，此处read会返回
            match input.read(&mut bytes) {
                //子进程成功执行
                Ok(0) => return Ok((p, ours)),
                //子进程exec失败，返回错误
                Ok(8) => {
                    let (errno, footer) = bytes.split_at(4);
                    assert_eq!(
                        CLOEXEC_MSG_FOOTER, footer,
```

```
                        "Validation on the CLOEXEC pipe failed: {:?}",
                        bytes
                    );
                    let errno = i32::from_be_bytes(errno.try_into().unwrap());
                    //子进程失败时做个等待
                    assert!(p.wait().is_ok(), "wait() should either return Ok
or panic");
                    return Err(Error::from_raw_os_error(errno));
                }
                //以下为其他失败情况
                Err(ref e) if e.kind() == ErrorKind::Interrupted => {}
                Err(e) => {
                    assert!(p.wait().is_ok(), "wait() should either return Ok
or panic");
                    panic!("the CLOEXEC pipe failed: {e:?}")
                }
                Ok(..) => {
                    // pipe I/O up to PIPE_BUF bytes should be atomic
                    assert!(p.wait().is_ok(), "wait() should either return Ok
or panic");
                    panic!("short read on the CLOEXEC pipe")
                }
            }
        }
    }

    //fork系统调用的RUST版本,不做更多分析
    unsafe fn do_fork(&mut self) -> Result<(pid_t, pid_t), io::Error> {
        use crate::sync::atomic::{AtomicBool, Ordering};

        static HAS_CLONE3: AtomicBool = AtomicBool::new(true);
        const CLONE_PIDFD: u64 = 0x00001000;

        #[repr(C)]
        struct clone_args {
            flags: u64,
            pidfd: u64,
            child_tid: u64,
            parent_tid: u64,
            exit_signal: u64,
            stack: u64,
            stack_size: u64,
            tls: u64,
            set_tid: u64,
            set_tid_size: u64,
            cgroup: u64,
        }

        raw_syscall! {
            fn clone3(cl_args: *mut clone_args, len: libc::size_t) ->
libc::c_long
```

```
        }

        // Bypassing libc for `clone3` can make further libc calls unsafe,
        // so we use it sparingly for now. See #89522 for details.
        // Some tools (e.g. sandboxing tools) may also expect `fork`
        // rather than `clone3`.
        let want_clone3_pidfd = self.get_create_pidfd();

        // If we fail to create a pidfd for any reason, this will
        // stay as -1, which indicates an error.
        let mut pidfd: pid_t = -1;

        // Attempt to use the `clone3` syscall, which supports more arguments
        // (in particular, the ability to create a pidfd). If this fails,
        // we will fall through this block to a call to `fork()`
        if want_clone3_pidfd && HAS_CLONE3.load(Ordering::Relaxed) {
            let mut args = clone_args {
                flags: CLONE_PIDFD,
                pidfd: &mut pidfd as *mut pid_t as u64,
                child_tid: 0,
                parent_tid: 0,
                exit_signal: libc::SIGCHLD as u64,
                stack: 0,
                stack_size: 0,
                tls: 0,
                set_tid: 0,
                set_tid_size: 0,
                cgroup: 0,
            };

            let args_ptr = &mut args as *mut clone_args;
            let args_size = crate::mem::size_of::<clone_args>();

            let res = cvt(clone3(args_ptr, args_size));
            match res {
                Ok(n) => return Ok((n as pid_t, pidfd)),
                Err(e) => match e.raw_os_error() {
                    // Multiple threads can race to execute this store,
                    // but that's fine - that just means that multiple threads
                    // will have tried and failed to execute the same syscall,
                    // with no other side effects.
                    Some(libc::ENOSYS) => HAS_CLONE3.store(false,
Ordering::Relaxed),
                    // Fallback to fork if `EPERM` is returned. (e.g. blocked
by seccomp)
                    Some(libc::EPERM) => {}
                    _ => return Err(e),
                },
            }
        }
```

```
        // Generally, we just call `fork`. If we get here after wanting
`clone3`,
        // then the syscall does not exist or we do not have permission to call
it.
        cvt(libc::fork()).map(|res| (res, pidfd))
    }

    // 执行二进制文件,值得注意的是，调用execvp，函数将不再返回，从而导致父进程已经申请
的内存及
    // 文件描述符资源会错误的被处理，因此，这个函数整体的安全规则仍然与C语言调用execvp
的规则一致，
    // RUST的安全规则在此处起不到更大的帮助
    unsafe fn do_exec(
        &mut self,
        stdio: ChildPipes,
        maybe_envp: Option<&CStringArray>,
    ) -> Result<!, io::Error> {
        use crate::sys::{self, cvt_r};

        //用dup2系统调用设置本进程的标准输入/输出/错误
        if let Some(fd) = stdio.stdin.fd() {
            cvt_r(|| libc::dup2(fd, libc::STDIN_FILENO))?;
        }
        if let Some(fd) = stdio.stdout.fd() {
            cvt_r(|| libc::dup2(fd, libc::STDOUT_FILENO))?;
        }
        if let Some(fd) = stdio.stderr.fd() {
            cvt_r(|| libc::dup2(fd, libc::STDERR_FILENO))?;
        }

        //设置进程其他参数，具体请参考libc的编程手册获得相关函数指导
        {
            if let Some(_g) = self.get_groups() {
                cvt(libc::setgroups(_g.len().try_into().unwrap(),
_g.as_ptr()))?;
            }
            if let Some(u) = self.get_gid() {
                cvt(libc::setgid(u as gid_t))?;
            }
            if let Some(u) = self.get_uid() {
                if libc::getuid() == 0 && self.get_groups().is_none() {
                    cvt(libc::setgroups(0, ptr::null()))?;
                }
                cvt(libc::setuid(u as uid_t))?;
            }
        }
        if let Some(ref cwd) = *self.get_cwd() {
            cvt(libc::chdir(cwd.as_ptr()))?;
        }

        if let Some(pgroup) = self.get_pgroup() {
```

```
            cvt(libc::setpgid(0, pgroup))?;
        }

        {
            //对进程接收的信号进行设置
            use crate::mem::MaybeUninit;
            //注意这里用MaybeUninit申请了一个栈变量，后继需要给C函数使用
            //因为不能由RUST做drop，所以用uninit(),这是调用C的通常用法
            let mut set = MaybeUninit::<libc::sigset_t>::uninit();
            //这些设置，execvp执行后，那边的代码会重新设置
            cvt(sigemptyset(set.as_mut_ptr()))?;
            cvt(libc::pthread_sigmask(libc::SIG_SETMASK, set.as_ptr(),
ptr::null_mut()))?;

            {
                //设置PIPE为默认处理
                let ret = sys::signal(libc::SIGPIPE, libc::SIG_DFL);
                if ret == libc::SIG_ERR {
                    return Err(io::Error::last_os_error());
                }
            }
        }

        //用于在执行二进制之前做些提前设置的回调闭包操作,如统计信息和Log日志之类
        //考虑很完善
        for callback in self.get_closures().iter_mut() {
            callback()?;
        }

        //以下用于在exec出错的时候恢复环境变量,这个地方是RUST细心对安全考虑的地方
        //请细心体会以下下面利用生命周期的错误处理方式
        let mut _reset = None;
        if let Some(envp) = maybe_envp {
            struct Reset(*const *const libc::c_char);

            impl Drop for Reset {
                fn drop(&mut self) {
                    unsafe {
                        *sys::os::environ() = self.0;
                    }
                }
            }

            _reset = Some(Reset(*sys::os::environ()));
            *sys::os::environ() = envp.as_ptr();
        }

        //调用execvp执行二进制文件
        libc::execvp(self.get_program_cstr().as_ptr(),
self.get_argv().as_ptr());
        //返回证明出错
```

```
            Err(io::Error::last_os_error())
        }
    }
```

以上是RUST操作系统适配层的代码分析。

RUST标准库对外提供的进程管理类型结构：

```
use crate::sys::process as imp;
// Child用来保存创建的子进程的信息，
// 其实质是Process的封装结构，Child被用来
// 作为单个子进程的管理信息的类型结构。每个Child变量
// 保存一个子进程的所有管理信息
pub struct Child {
    //系统分配的子进程的标识句柄
    pub(crate) handle: imp::Process,

    //父进程用此向子进程标准输入写入通信信息
    pub stdin: Option<ChildStdin>,

    //父进程用此从子进程标准输出读出通信信息
    pub stdout: Option<ChildStdout>,

    //父进程用此从子进程标准错误读出错误信息
    pub stderr: Option<ChildStderr>,
}

//父进程保留的与子进程标准输入/输出/错误相对应的管道信息
pub struct ChildStdin {
    inner: AnonPipe,
}
pub struct ChildStdout {
    inner: AnonPipe,
}
pub struct ChildStderr {
    inner: AnonPipe,
}

//Command用于其他模块对进程管理操作的界面
pub struct Command {
    inner: imp::Command,
}
impl Command {
    //输入进程的二进制可执行文件的路径及名称，
    pub fn new<S: AsRef<OsStr>>(program: S) -> Command {
        Command { inner: imp::Command::new(program.as_ref()) }
    }

    //输入一个进程命令的参数
    pub fn arg<S: AsRef<OsStr>>(&mut self, arg: S) -> &mut Command {
```

```
        self.inner.arg(arg.as_ref());
        self
    }

    //输入若干进程命令的参数
    pub fn args<I, S>(&mut self, args: I) -> &mut Command
    where
        I: IntoIterator<Item = S>,
        S: AsRef<OsStr>,
    {
        for arg in args {
            self.arg(arg.as_ref());
        }
        self
    }

    //针对进程插入或设置一个环境参数
    pub fn env<K, V>(&mut self, key: K, val: V) -> &mut Command
    where
        K: AsRef<OsStr>,
        V: AsRef<OsStr>,
    {
        self.inner.env_mut().set(key.as_ref(), val.as_ref());
        self
    }

    //插入或设置若干个环境参数
    pub fn envs<I, K, V>(&mut self, vars: I) -> &mut Command
    where
        I: IntoIterator<Item = (K, V)>,
        K: AsRef<OsStr>,
        V: AsRef<OsStr>,
    {
        for (ref key, ref val) in vars {
            self.inner.env_mut().set(key.as_ref(), val.as_ref());
        }
        self
    }

    //清除一个环境参数
    pub fn env_remove<K: AsRef<OsStr>>(&mut self, key: K) -> &mut Command {
        self.inner.env_mut().remove(key.as_ref());
        self
    }

    //清除所有环境参数
    pub fn env_clear(&mut self) -> &mut Command {
        self.inner.env_mut().clear();
        self
    }
```

```
    //获取当前目录
    pub fn current_dir<P: AsRef<Path>>(&mut self, dir: P) -> &mut Command {
        self.inner.cwd(dir.as_ref().as_ref());
        self
    }

    //配置标准输入，此处的Stdio见后面的代码，与imp::Stdio不同。
    pub fn stdin<T: Into<Stdio>>(&mut self, cfg: T) -> &mut Command {
        self.inner.stdin(cfg.into().0);
        self
    }

    //配置标准输出
    pub fn stdout<T: Into<Stdio>>(&mut self, cfg: T) -> &mut Command {
        self.inner.stdout(cfg.into().0);
        self
    }

    //配置标准错误
    pub fn stderr<T: Into<Stdio>>(&mut self, cfg: T) -> &mut Command {
        self.inner.stderr(cfg.into().0);
        self
    }

    //正式按照Command的参数创建进程
    pub fn spawn(&mut self) -> io::Result<Child> {
        self.inner.spawn(imp::Stdio::Inherit, true).map(Child::from_inner)
    }

    //创建子进程，并等待子进程结束,返回Output结构变量
    pub fn output(&mut self) -> io::Result<Output> {
        self.inner
            .spawn(imp::Stdio::MakePipe, false)
            .map(Child::from_inner)
            .and_then(|p| p.wait_with_output())
    }

    //创建子进程，等待进程结束，返回进程退出码
    pub fn status(&mut self) -> io::Result<ExitStatus> {
        self.inner
            .spawn(imp::Stdio::Inherit, true)
            .map(Child::from_inner)
            .and_then(|mut p| p.wait())
    }

    ...
}

//进程命令的参数集合
pub struct CommandArgs<'a> {
    inner: imp::CommandArgs<'a>,
```

```rust
}

//保存子进程结束后的输出
pub struct Output {
    //子进程退出的返回码
    pub status: ExitStatus,
    //子进程的标准输出输出的内容
    pub stdout: Vec<u8>,
    //子进程标准错误输出的内容
    pub stderr: Vec<u8>,
}

//标准输入输出的类型结构
pub struct Stdio(imp::Stdio);

impl Stdio {
    //创建一个管道的标准输入输出
    pub fn piped() -> Stdio {
        Stdio(imp::Stdio::MakePipe)
    }
    //继承父进程
    pub fn inherit() -> Stdio {
        Stdio(imp::Stdio::Inherit)
    }

    //使用/dev/null作为进程标准输入输出
    pub fn null() -> Stdio {
        Stdio(imp::Stdio::Null)
    }
}

//对子进程的API
impl Child {
    //杀掉子进程
    pub fn kill(&mut self) -> io::Result<()> {
        self.handle.kill()
    }

    //获取子进程的id值
    pub fn id(&self) -> u32 {
        self.handle.id()
    }

    //等待子进程结束，需要提前释放子进程的标准输入，否则子进程
    //可能不会退出
    pub fn wait(&mut self) -> io::Result<ExitStatus> {
        drop(self.stdin.take());
        self.handle.wait().map(ExitStatus)
    }
```

```
    //尝试等子进程退出
    pub fn try_wait(&mut self) -> io::Result<Option<ExitStatus>> {
        Ok(self.handle.try_wait()?.map(ExitStatus))
    }

    //等待子进程退出并获取所有输出
    pub fn wait_with_output(mut self) -> io::Result<Output> {
        //需要先关闭子进程的标准输入
        drop(self.stdin.take());

        //申请缓存
        let (mut stdout, mut stderr) = (Vec::new(), Vec::new());
        match (self.stdout.take(), self.stderr.take()) {
            (None, None) => {}

            (Some(mut out), None) => {
                //读子进程标准输出到缓存
                let res = out.read_to_end(&mut stdout);
                res.unwrap();
            }
            (None, Some(mut err)) => {
                //读标准错误到缓存
                let res = err.read_to_end(&mut stderr);
                res.unwrap();
            }
            (Some(out), Some(err)) => {
                //读两个，此函数应该专门为这个目的做的设计
                let res = read2(out.inner, &mut stdout, err.inner, &mut
stderr);
                res.unwrap();
            }
        }

        //等待子进程结束
        let status = self.wait()?;
        //创建Output返回
        Ok(Output { status, stdout, stderr })
    }
}
//从进程退出并返回一个值，父进程会获得这个值
pub fn exit(code: i32) -> ! {
    crate::rt::cleanup();
    crate::sys::os::exit(code)
}

//异常退出，与exit相比较，不会处理资源释放操作
pub fn abort() -> ! {
    crate::sys::abort_internal();
}
```

不熟悉操作系统的进程操作，基本就没有办法理解进程管理，因此，最关键的仍然是对操作系统系统调用的理解，而操作系统调用实际上又是C编程的较高级的内容。对RUST的std库的深入学习目前仍然需要有较高的C语言编程水平。