RUST的RUNTIME

RUST的runtime及程序的主线程初始化
路径：library/std/src/rt.rs：
library/std/src/unix/mod.rs
library/std/src/panic.rs
library/std/src/panicking.rs
library/std/src/panic/*.rs
RUST程序execv以后，最初是由std::rt::lang_start进入RUST的RUNTIME : 代码如下：

```
//RUST应用的代码入口点
fn lang_start<T: crate::process::Termination + 'static>(
    main: fn() -> T,
    argc: isize,
    argv: *const *const u8,
) -> isize {
    //调用了lang_start_internal
    let Ok(v) = lang_start_internal(
        //__rust_begin_short_backtrace(main)标识栈顶,同时也调用了main
        &move ||
crate::sys_common::backtrace::__rust_begin_short_backtrace(main).report().to_i3
2(),
        argc,
        argv,
    );
    v
}

fn lang_start_internal(
    main: &(dyn Fn() -> i32 + Sync + crate::panic::RefUnwindSafe),
    argc: isize,
    argv: *const *const u8,
) -> Result<isize, !> {
    use crate::{mem, panic};
    let rt_abort = move |e| {
        mem::forget(e);
        rtabort!("initialization or cleanup bug");
    };
    //完成执行main之前的准备,具体见后面的init函数，用catch_unwind捕获init函数执行中
的panic信息
    panic::catch_unwind(move || unsafe { init(argc, argv)
}).map_err(rt_abort)?;
    //执行main函数，同样，用catch_unwind捕获所有可能的panic信息
    let ret_code = panic::catch_unwind(move ||
panic::catch_unwind(main).unwrap_or(101) as isize)
```

```
        .map_err(move |e| {
            mem::forget(e);
            rtabort!("drop of the panic payload panicked");
        });
    //完成所有的清理工作,一样的catch_unwind
    panic::catch_unwind(cleanup).map_err(rt_abort)?;
    ret_code
}
```

进入main函数之前的初始化内容

```
//此函数在main函数之前被调用完成标准输入/输出/错误，线程栈保护等设置，
//然后控制权交给main
unsafe fn init(argc: isize, argv: *const *const u8) {
    unsafe {
        //见下面的代码分析，完成进入main的各项初始化
        sys::init(argc, argv);

        //以下是对主线程的线程runtime的初始化,可对比线程的spawn函数
        //设置主线程的栈保护
        let main_guard = sys::thread::guard::init();
        //设置当前的线程为主线程
        let thread = Thread::new(Some(rtunwrap!(Ok, CString::new("main"))));
        //设置栈保护地址与线程的信息， 使用了thread_local_key的方式使得此info仅与当前
线程相关
        thread_info::set(main_guard, thread);
    }
}

//linux系统的上文sys::init实现
pub unsafe fn init(argc: isize, argv: *const *const u8) {
    // 见下文说明.
    sanitize_standard_fds();

    // 将 SIGPIPE 设置为ignore
    reset_sigpipe();

    //进程栈溢出初始化,系统调用sigaltstack()支持设置一个内存空间，当访问这个空间地址
的时候
    //发送一个信号给进程，stack_overflow即利用这个机制完成了对当前线程的该信号的设置
及处理
    //这个对所有线程的堆栈溢出的处理做了初始化
    stack_overflow::init();
    //对命令行的输入完成RUST的结构转化
    args::init(argc, argv);

    unsafe fn sanitize_standard_fds() {
        //仅linux
        {
            {
```

```
                use crate::sys::os::errno;
                //轮询stdin,stdout,stderr的文件描述符
                let pfds: &mut [_] = &mut [
                    libc::pollfd { fd: 0, events: 0, revents: 0 },
                    libc::pollfd { fd: 1, events: 0, revents: 0 },
                    libc::pollfd { fd: 2, events: 0, revents: 0 },
                ];
                //从poll结果获得文件描述符是否已经关闭
                while libc::poll(pfds.as_mut_ptr(), 3, 0) == -1 {
                    if errno() == libc::EINTR {
                        continue;
                    }
                    //此处说明未知错误需要退出
                    libc::abort();
                }
                for pfd in pfds {
                    if pfd.revents & libc::POLLNVAL == 0 {
                        //文件描述符已经打开
                        continue;
                    }
                    //文件描述符关闭，则用/dev/null作为文件描述符，注意下面直接用str
转换为CStr的
                    //代码,因为此循环的fd最小，所以下面这个open如果调用成功，返回的
fd即为当前的//被关闭的fd.从而达到了重新将标准输入/输出/错误文件描述符打开的目的
                    if libc::open("/dev/null\0".as_ptr().cast(), libc::O_RDWR,
0) == -1 {
                        // 无法打开文件，则应退出程序
                        libc::abort();
                    }
                }
            }
        }
    }

    //设置对SIGPIPE的处理为IGNORE
    unsafe fn reset_sigpipe() {
        rtassert!(signal(libc::SIGPIPE, libc::SIG_IGN) != libc::SIG_ERR);
    }
}
```

对panic的捕获函数：

```
//对f的panic做unwind操作并捕获
pub fn catch_unwind<F: FnOnce() -> R + UnwindSafe, R>(f: F) -> Result<R> {
    //编译器的try catch机制
    unsafe { panicking::r#try(f) }
}

//常用于前面已经调用过catch_unwind，但需要继续panic过程
pub fn resume_unwind(payload: Box<dyn Any + Send>) -> ! {
```

```
    panicking::rust_panic_without_hook(payload)
}
```

RUST的RUNTIME主要是完成一些安全机制及异常处理机制。了解RUNTIME可以使得我们对如何构建一个强健的，易于排查错误的应用有更深的了解。