线程管理分析

RUST的线程主要由以下几部分组成：

1. 线程属性设置，线程创建，join，销毁等,是操作系统系统调用的RUST延伸
2. 线程局部存储, 是操作系统系统调用的RUST延伸
3. 线程运行时及panic管理, 是RUST的异常处理方案一部分
4. RUST运行时
5. 为在线程中借用环境变量的Scope方案，是RUST语言自身的特性

基于操作系统调用封装的底层线程结构

均以linux为例，wasi与linux基本相同

线程类型结构

```
// Thread结构，pthread函数需要用此结构作为参数调用pthread的API
// 这里 id没有象fd那样实现RawFd, OwnedFd, BorrowedFd
pub struct Thread {
    id: libc::pthread_t,
}

// Thread当然应该支持Send 及 Sync
unsafe impl Send for Thread {}
unsafe impl Sync for Thread {}

impl Thread {
    // 熟悉C语言的会发现这个函数很容易理解，基本和C的同类型函数
    // 可以一一映射。大量的libc的调用直接导致用unsafe标记这个函数
    pub unsafe fn new(stack: usize, p: Box<dyn FnOnce()>) -> io::Result<Thread>
{
        //申请一个堆内存存放Box<dyn FnOnce()>，并解封，将p直接置为申请的堆内存地址
        // p就是线程函数指针
        let p = Box::into_raw(box p);
        //等于pthread_t native = 0;
        let mut native: libc::pthread_t = mem::zeroed();
        let mut attr: libc::pthread_attr_t = mem::zeroed();
        //pthread_attr_init出错是有可能的，此处标准库偷了懒
        assert_eq!(libc::pthread_attr_init(&mut attr), 0);

        //对线程栈进行设置，一般不必设置
        {
            //线程栈不能小于允许最小的栈，实际上最大栈也应该有限制
            let stack_size = cmp::max(stack, min_stack_size(&attr));
```

```
            //设置线程栈大小
            match libc::pthread_attr_setstacksize(&mut attr, stack_size) {
                0 => {}
                n => {
                    assert_eq!(n, libc::EINVAL);
                    // 仅在参数不是内存页整数倍的情况会执行下面代码,重新调整栈空间,并
设置,
                    // 此时的设置不应该再出错
                    let page_size = os::page_size();
                    let stack_size =
                        (stack_size + page_size - 1) & (-(page_size as isize -
1) as usize - 1);
                    assert_eq!(libc::pthread_attr_setstacksize(&mut attr,
stack_size), 0);
                }
            };
        }

        //创建线程，thread_start是线程主函数，见后面分析
        //输入的闭包p作为thread_start的参数，attr当前只处理栈大小，成功后native会被
赋值
        let ret = libc::pthread_create(&mut native, &attr, thread_start, p as
*mut _);
        // attr任务完成，释放其申请的资源，C编程的时候这一步经常被忽略，这也是RUST的
一个安全体现
        // 此处RUST认为不会失败
        assert_eq!(libc::pthread_attr_destroy(&mut attr), 0);

        return if ret != 0 {
            // 失败
            // 重新建立Box以便释放申请的堆内存
            drop(Box::from_raw(p));
            //获取操作系统的错误
            Err(io::Error::from_raw_os_error(ret))
        } else {
            //成功，创建Thread返回
            Ok(Thread { id: native })
        };

        //所有RUST线程的主函数
        extern "C" fn thread_start(main: *mut libc::c_void) -> *mut
libc::c_void {
            unsafe {
                // 这个是线程栈保护机制，如果线程出现栈溢出，可以用这个机制探测到，
                // 这个是C语言编写大的服务器应用如数据库等积累下来的经验
                // 对C来说必要性是很大的，具体的代码分析略
                let _handler = stack_overflow::Handler::new();
                // 先将传入的堆内存重组为Box，然后自动解双层引用消费掉两个Box并运行真
正的
                //线程函数
                Box::from_raw(main as *mut Box<dyn FnOnce()>)();
```

```
            }
            //C函数的返回值
            ptr::null_mut()
        }
    }

    pub fn yield_now() {
        //让出CPU
        let ret = unsafe { libc::sched_yield() };
        debug_assert_eq!(ret, 0);
    }

    pub fn set_name(name: &CStr) {
        const PR_SET_NAME: libc::c_int = 15;
        // 更改线程名字，具体的系统调用请参考libc库
        unsafe {
            libc::prctl(
                PR_SET_NAME,
                name.as_ptr(),
                0 as libc::c_ulong,
                0 as libc::c_ulong,
                0 as libc::c_ulong,
            );
        }
    }

    //线程睡眠,可以认为是glibc的sleep函数的RUST版本
    pub fn sleep(dur: Duration) {
        let mut secs = dur.as_secs();
        let mut nsecs = dur.subsec_nanos() as _;

        unsafe {
            //一次睡不到位就多睡几次，另外也可能被中途打断，那需要再接着睡，
            // 以下这段代码值得注意，nanosleep让以前简单的调用一次usleep的我深感惭愧
            while secs > 0 || nsecs > 0 {
                //准备C语言时间变量
                let mut ts = libc::timespec {
                    tv_sec: cmp::min(libc::time_t::MAX as u64, secs) as
libc::time_t,
                    tv_nsec: nsecs,
                };
                secs -= ts.tv_sec as u64;
                let ts_ptr = &mut ts as *mut _;
                if libc::nanosleep(ts_ptr, ts_ptr) == -1 {
                    assert_eq!(os::errno(), libc::EINTR);
                    //中途被打断的话，ts_ptr会放置还剩余的时间
                    //因此要把时间重新加入，下一次循环再睡
                    //真的是容易被忽视的返回值
                    secs += ts.tv_sec as u64;
                    nsecs = ts.tv_nsec;
                } else {
```

```
                    //重新置值
                    nsecs = 0;
                }
            }
        }
    }

    //等待目标线程结束
    pub fn join(self) {
        unsafe {
            let ret = libc::pthread_join(self.id, ptr::null_mut());
            //默认是非join，此处的作用是取消drop导致的默认pthread_detach调用
            mem::forget(self);
            assert!(ret == 0, "failed to join thread: {}",
io::Error::from_raw_os_error(ret));
        }
    }

    pub fn id(&self) -> libc::pthread_t {
        //此处所有权没有转移。一般用于做pthread的系统调用临时使用
        //但不能用这个返回的id调用pthread_detach或者pthread_join及类似功能的pthread
        //C函数
        self.id
    }

    pub fn into_id(self) -> libc::pthread_t {
        let id = self.id;
        //pthread_detach应该由调用此函数的代码负责。
        //所有权已经转移
        mem::forget(self);
        id
    }
}

impl Drop for Thread {
    //主要用于父线程不必等待子线程结束的情况，默认为不等待
    fn drop(&mut self) {
        let ret = unsafe { libc::pthread_detach(self.id) };
        debug_assert_eq!(ret, 0);
    }
}
```

线程堆栈溢出守卫及运行时

RUST在操作系统线程的基础上，实现了线程栈内存的溢出检查:

```
//线程栈溢出守卫
pub mod guard {
    // 内存页大小
    static PAGE_SIZE: AtomicUsize = AtomicUsize::new(0);
```

```rust
    pub type Guard = Range<usize>;

    //获取线程栈栈底(栈溢出)内存地址,
    unsafe fn get_stack_start() -> Option<*mut libc::c_void> {
        //以下实际上可以认为是一个使用了RUST语法的C函数,
        //具体的pthread函数请参观相关手册
        let mut ret = None;
        let mut attr: libc::pthread_attr_t = crate::mem::zeroed();
        let e = libc::pthread_getattr_np(libc::pthread_self(), &mut attr);
        if e == 0 {
            let mut stackaddr = crate::ptr::null_mut();
            let mut stacksize = 0;
            assert_eq!(libc::pthread_attr_getstack(&attr, &mut stackaddr, &mut
stacksize), 0);
            ret = Some(stackaddr);
        }
        if e == 0 {
            assert_eq!(libc::pthread_attr_destroy(&mut attr), 0);
        }
        ret
    }

    // 获取从线程栈栈底(地址小的一侧)找到的第一个内存页对齐的地址
    unsafe fn get_stack_start_aligned() -> Option<*mut libc::c_void> {
        let page_size = PAGE_SIZE.load(Ordering::Relaxed);
        assert!(page_size != 0);
        let stackptr = get_stack_start()?;
        let stackaddr = stackptr.addr();

        // 以下计算从栈底向栈顶方向找到第一个对齐地址
        let remainder = stackaddr % page_size;
        Some(if remainder == 0 {
            stackptr
        } else {
            stackptr.with_addr(stackaddr + page_size - remainder)
        })
    }

    pub unsafe fn init() -> Option<Guard> {
        //获得内存页大小
        let page_size = os::page_size();
        PAGE_SIZE.store(page_size, Ordering::Relaxed);

        {
            // Linux 内核已经做了栈守护，所以使用内核的机制
            let stackptr = get_stack_start_aligned()?;
            let stackaddr = stackptr.addr();
            Some(stackaddr - page_size..stackaddr)
        }
    }
```

```
    //获取当前栈守护地址
    pub unsafe fn current() -> Option<Guard> {
        let mut ret = None;
        let mut attr: libc::pthread_attr_t = crate::mem::zeroed();
        let e = libc::pthread_getattr_np(libc::pthread_self(), &mut attr);
        if e == 0 {
            let mut guardsize = 0;
            assert_eq!(libc::pthread_attr_getguardsize(&attr, &mut guardsize),
0);
            if guardsize == 0 {
                    panic!("there is no guard page");
            }
            let mut stackptr = crate::ptr::null_mut::<libc::c_void>();
            let mut size = 0;
            assert_eq!(libc::pthread_attr_getstack(&attr, &mut stackptr, &mut
size), 0);

            let stackaddr = stackptr.addr();
            let ret = {
                Some(stackaddr - guardsize..stackaddr + guardsize)
            }
        }
        ret
    }
}

fn min_stack_size(attr: *const libc::pthread_attr_t) -> usize {
    //用动态链接获取库函数
    dlsym!(fn __pthread_get_minstack(*const libc::pthread_attr_t) ->
libc::size_t);

    match __pthread_get_minstack.get() {
        None => libc::PTHREAD_STACK_MIN,
        Some(f) => unsafe { f(attr) },
    }
}
//以上的代码对于C程序员，是比较容易理解的，因为涉及到大量的pthread的库函数

// 专用于处理栈溢出的结构及实现
pub struct Handler {
    data: *mut libc::c_void,
}

impl Handler {
    pub unsafe fn new() -> Handler {
        //主要用于完成溢出处理函数的栈处置
        make_handler()
    }

    fn null() -> Handler {
        Handler { data: crate::ptr::null_mut() }
```

```
    }
}

impl Drop for Handler {
    fn drop(&mut self) {
        unsafe {
            drop_handler(self.data);
        }
    }
}

//stack_overflow模块
mod imp {
    // 对SIGSEGV及SIGBUS的信号处理函数。这两个函数会在线程出现栈溢出时被触发
    unsafe extern "C" fn signal_handler(
        signum: libc::c_int,
        info: *mut libc::siginfo_t,
        _data: *mut libc::c_void,
    ) {
        let guard = thread_info::stack_guard().unwrap_or(0..0);
        let addr = (*info).si_addr() as usize;

        // 判断是否访问了栈保护端的地址，如果是，则输出告警信息
        if guard.start <= addr && addr < guard.end {
            rtprintpanic!(
                "\nthread '{}' has overflowed its stack\n",
                thread::current().name().unwrap_or("<unknown>")
            );
            rtabort!("stack overflow");
        } else {
            // 否则执行默认操作.
            let mut action: sigaction = mem::zeroed();
            action.sa_sigaction = SIG_DFL;
            sigaction(signum, &action, ptr::null_mut());
        }
    }

    static MAIN_ALTSTACK: AtomicPtr<libc::c_void> =
AtomicPtr::new(ptr::null_mut());
    static NEED_ALTSTACK: AtomicBool = AtomicBool::new(false);

    //初始化信号函数注册，此函数似乎,在sys::init中被调用
    pub unsafe fn init() {
        let mut action: sigaction = mem::zeroed();
        for &signal in &[SIGSEGV, SIGBUS] {
            sigaction(signal, ptr::null_mut(), &mut action);
            // 配置保护内存访问的信号处理函数.
            if action.sa_sigaction == SIG_DFL {
                action.sa_flags = SA_SIGINFO | SA_ONSTACK;
                action.sa_sigaction = signal_handler as sighandler_t;
                sigaction(signal, &action, ptr::null_mut());
```

```
                NEED_ALTSTACK.store(true, Ordering::Relaxed);
            }
        }

        let handler = make_handler();
        MAIN_ALTSTACK.store(handler.data, Ordering::Relaxed);
        mem::forget(handler);
    }

    pub unsafe fn cleanup() {
        drop_handler(MAIN_ALTSTACK.load(Ordering::Relaxed));
    }

    //下面这段函数是将段保护的内存设置成用户态写入会触发缺页中断，从而触发信号
    unsafe fn get_stackp() -> *mut libc::c_void {
        let flags = MAP_PRIVATE | MAP_ANON | libc::MAP_STACK;
        //mmap一段内存作为信号处理函数的栈，额外一个page用作保护
        let stackp =
            mmap(ptr::null_mut(), SIGSTKSZ + page_size(), PROT_READ |
PROT_WRITE, flags, -1, 0);
        if stackp == MAP_FAILED {
            panic!("failed to allocate an alternative stack: {}",
io::Error::last_os_error());
        }
        // 最低的一个page用来作为保护
        let guard_result = libc::mprotect(stackp, page_size(), PROT_NONE);
        if guard_result != 0 {
            panic!("failed to set up alternative stack guard page: {}",
io::Error::last_os_error());
        }
        // 真正的栈从底部向上一个page开始
        stackp.add(page_size())
    }

    unsafe fn get_stack() -> libc::stack_t {
        libc::stack_t { ss_sp: get_stackp(), ss_flags: 0, ss_size: SIGSTKSZ }
    }

    //用于对每个线程设置线程信号处理的栈
    pub unsafe fn make_handler() -> Handler {
        if !NEED_ALTSTACK.load(Ordering::Relaxed) {
            return Handler::null();
        }
        let mut stack = mem::zeroed();
        sigaltstack(ptr::null(), &mut stack);
        // 设置信号处理函数的栈
        if stack.ss_flags & SS_DISABLE != 0 {
            //设置用于信号处理的栈
            stack = get_stack();
            // 设置栈，长度为SIGSTKSZ
            sigaltstack(&stack, ptr::null_mut());
```

```
            Handler { data: stack.ss_sp as *mut libc::c_void }
        } else {
            Handler::null()
        }
    }

    pub unsafe fn drop_handler(data: *mut libc::c_void) {
        if !data.is_null() {
            let stack = libc::stack_t {
                ss_sp: ptr::null_mut(),
                ss_flags: SS_DISABLE,
                ss_size: SIGSTKSZ,
            };
            //删除信号处理函数专用栈
            sigaltstack(&stack, ptr::null_mut());
            // unmap用于信号处理的内存.
            munmap(data.sub(page_size()), SIGSTKSZ + page_size());
        }
    }
}
```

线程局部变量类型

线程的本地全局变量Thread Local Key的实现。线程的本地存储解决一类问题如下： 在线程代码中，有时希望多个线程共享同一个变量名的全局变量，以简化编码。但希望这个变量在不同的线程有各自的拷贝，彼此不影响。典型的例子就是前文的线程栈guard空间。如果每个线程都共享同一个变量名，那代码会少很多啰嗦。 具体的代码分析如下:

```
//pthread_key_t请参考libc的pthread编程手册
pub type Key = libc::pthread_key_t;

//dtor用于对创建的key做释放操作，返回的Key可以被进程中的线程共享使用
pub unsafe fn create(dtor: Option<unsafe extern "C" fn(*mut u8)>) -> Key {
    let mut key = 0;
    assert_eq!(libc::pthread_key_create(&mut key, mem::transmute(dtor)), 0);
    key
}

// 各线程可以将key设置成自己需要的内存块，这个内存块的所有权属于Key，
// 这是个代码规定，编译器不知道，所以安全上需要程序员负责
pub unsafe fn set(key: Key, value: *mut u8) {
    let r = libc::pthread_setspecific(key, value as *mut _);
    debug_assert_eq!(r, 0);
}

// 用key将内存块获得，实际上是获得一个引用
pub unsafe fn get(key: Key) -> *mut u8 {
    libc::pthread_getspecific(key) as *mut u8
```

```
}

// 删除掉key，需要所有线程都删除，调用此函数会导致调用内存块的dtor函数，也即drop
pub unsafe fn destroy(key: Key) {
    let r = libc::pthread_key_delete(key);
    debug_assert_eq!(r, 0);
}
```

标准库线程支持层代码分析

代码路径：library/std/src/sys_common/thread.rs
library/std/src/sys_common/thread_local.rs library/std/src/sys_common/thread_info.rs

在操作系统的线程概念与RUST作为API提供的线程之间的一层代码。主要处理一些RUST的语法导致的一些需要额外在操作系统的线程结构做一些包装的基础层。

对Thread Local Key做类型封装，以屏蔽不同操作系统除API外的差异。
代码如下：

```
//适用与作为静态变量的Thread Local Key结构
pub struct StaticKey {
    /// 仅仅是一个数值，为0的时候代表此时无意义
    key: AtomicUsize,
    ///对key的析构函数
    dtor: Option<unsafe extern "C" fn(*mut u8)>,
}

//一般作为StaticKey的初始化赋值
pub const INIT: StaticKey = StaticKey::new(None);

impl StaticKey {
    pub const fn new(dtor: Option<unsafe extern "C" fn(*mut u8)>) -> StaticKey
{
        //key为0，代表此时没有创建thread local key
        StaticKey { key: atomic::AtomicUsize::new(0), dtor }
    }

    //如果没有创建thread local key，此方法会创建一个
    pub unsafe fn get(&self) -> *mut u8 {
        //获取key的指针
        //调用self.key会在无thread local key时创建一个
        imp::get(self.key())
    }

    //如果没有创建thread local key，此方法会创建一个
    pub unsafe fn set(&self, val: *mut u8) {
        imp::set(self.key(), val)
    }
```

```
    //获得thread local key的key值
    unsafe fn key(&self) -> imp::Key {
        match self.key.load(Ordering::Relaxed) {
            //如果为0，表示thread local key没有创建，需要创建一个
            0 => self.lazy_init() as imp::Key,
            //不为0，则返回key
            n => n as imp::Key,
        }
    }

    //创建一个thread local key
    unsafe fn lazy_init(&self) -> usize {
        // 为特殊的操作系统准备
        if imp::requires_synchronized_create() {
            // 需要加锁保护，因为保护静态变量，所以要使用StaticMutex
            // INIT_LOCK所有线程共享
            static INIT_LOCK: StaticMutex = StaticMutex::new();
            let _guard = INIT_LOCK.lock();
            let mut key = self.key.load(Ordering::SeqCst);
            if key == 0 {
                //创建Key
                key = imp::create(self.dtor) as usize;
                self.key.store(key, Ordering::SeqCst);
            }
            rtassert!(key != 0);
            return key;
            //_guard生命周期结束会释放INIT_LOCK
        }

        //unix系统有可能分配为0的thread local key
        let key1 = imp::create(self.dtor);
        let key = if key1 != 0 {
            key1
        } else {
            //如果是0，需要重新再申请一个新的key
            let key2 = imp::create(self.dtor);
            imp::destroy(key1);
            key2
        };
        rtassert!(key != 0);
        match self.key.compare_exchange(0, key as usize, Ordering::SeqCst,
Ordering::SeqCst) {
            //这里也作为方法的返回
            Ok(_) => key as usize,
            // 如果有其他的值，那就用那个值，
            Err(n) => {
                imp::destroy(key);
                n
            }
        }
```

```
    }
}

//非静态变量的Thread Local Key
pub struct Key {
    key: imp::Key,
}

impl Key {
    // 创建一个thread local key
    pub fn new(dtor: Option<unsafe extern "C" fn(*mut u8)>) -> Key {
        Key { key: unsafe { imp::create(dtor) } }
    }

    // 获取key相关的内存，可能为空，
    pub fn get(&self) -> *mut u8 {
        unsafe { imp::get(self.key) }
    }

    //设置key相关的内存
    pub fn set(&self, val: *mut u8) {
        unsafe { imp::set(self.key, val) }
    }
}

impl Drop for Key {
    fn drop(&mut self) {
        // Right now Windows doesn't support TLS key destruction, but this also
        // isn't used anywhere other than tests, so just leak the TLS key.
        // unsafe { imp::destroy(self.key) }
    }
}
```

标准库线程局部变量外部接口(Thread Local)

操作系统的Thread Local Key使用起来明显非常繁琐，且很容易出错。RUST标准库对其进行了符合`rust`的类型封装。这一类型需要完成的工作如下：

1. 对所有的Thread Local Key存放的真正的数据类型需要声明key时做定义。
2. 向使用者屏蔽key的创建及销毁过程，key与真实数据的捆绑与获取过程，使得key的使用类似于通用类型结构的使用。

RUST采用了宏及数据类型相结合的方案，下面的代码说明了RUST的local key的使用。

```
use std::cell::RefCell;
thread_local! {
    pub static FOO: RefCell<u32> = RefCell::new(1);
```

```
    static BAR: RefCell<f32> = RefCell::new(1.0);
}
fn main() {FOO.with(|f|{*f.borrow_mut() = 2})}
```

以上的`thread_local`将一个普通的变量定义转换为Thread Local Key的变量. 并且可以在随后的with方法内可以正常的的使用该普通变量。用这种方法，RUST使得Thread Local Key的变量使用与普通变量基本上做到了相一致。 (其他语言多用set(),get()方法完成Thread local的操作，RUST采用了更近一步的设计)
具体的代码实现如下：

```
//LocalKey只能用于静态变量
pub struct LocalKey<T: 'static> {
    //只能用于静态变量
    //inner是一个支持泛型的函数类型
    inner: unsafe fn(Option<&mut Option<T>>) -> Option<&'static T>,
}

pub struct AccessError;

impl Error for AccessError {}

impl<T: 'static> LocalKey<T> {
    // 这里仅仅做一个内存占位
    pub const unsafe fn new(
        inner: unsafe fn(Option<&mut Option<T>>) -> Option<&'static T>,
    ) -> LocalKey<T> {
        LocalKey { inner }
    }

    //所有的Thread Local的操作都在witch参数的闭包中,
    // with会将Thread Local的可变引用输入闭包的参数
    pub fn with<F, R>(&'static self, f: F) -> R
    where
        F: FnOnce(&T) -> R,
    {
        self.try_with(f).expect(
            "cannot access a Thread Local Storage value \
             during or after destruction",
        )
    }

    pub fn try_with<F, R>(&'static self, f: F) -> Result<R, AccessError>
    where
        F: FnOnce(&T) -> R,
    {
        unsafe {
            //获取Thread Local内存指针后，调用闭包
            let thread_local = (self.inner)(None).ok_or(AccessError)?;
```

```
            Ok(f(thread_local))
        }
    }

    //对Thread Local做初始化，然后再执行操作
    fn initialize_with<F, R>(&'static self, init: T, f: F) -> R
    where
        F: FnOnce(Option<T>, &T) -> R,
    {
        unsafe {
            let mut init = Some(init);
            let reference = (self.inner)(Some(&mut init)).expect(
                "cannot access a Thread Local Storage value \
                 during or after destruction",
            );
            f(init, reference)
        }
    }
}

//LocalKey通常会与内部可变性变量配合,设计方法来简化使用者的代码
impl<T: 'static> LocalKey<Cell<T>> {
    //对内部可变性变量赋值
    pub fn set(&'static self, value: T) {
        self.initialize_with(Cell::new(value), |value, cell| {
            if let Some(value) = value {
                // value输入的Cell变量参数，cell是Thread Local的引用
                // 对cell做出更新,并消费掉value.
                cell.set(value.into_inner());
            }
        });
    }

    //只能在T实现Copy trait的情况下支持，否则会出现
    //双份所有权
    pub fn get(&'static self) -> T
    where
        T: Copy,
    {
        self.with(|cell| cell.get())
    }

    //获取Thread Local的变量所有权，并将Thread Local置为默认
    pub fn take(&'static self) -> T
    where
        T: Default,
    {
        self.with(|cell| cell.take())
    }

    //替换
```

```
    pub fn replace(&'static self, value: T) -> T {
        self.with(|cell| cell.replace(value))
    }
}

//提供Thread Local是内部可变性的基础
impl<T: 'static> LocalKey<RefCell<T>> {
    //borrow的对应简化
    pub fn with_borrow<F, R>(&'static self, f: F) -> R
    where
        F: FnOnce(&T) -> R,
    {
        self.with(|cell| f(&cell.borrow()))
    }

    //borrow_mut的对应简化
    pub fn with_borrow_mut<F, R>(&'static self, f: F) -> R
    where
        F: FnOnce(&mut T) -> R,
    {
        self.with(|cell| f(&mut cell.borrow_mut()))
    }

    //修改值
    pub fn set(&'static self, value: T) {
        self.initialize_with(RefCell::new(value), |value, cell| {
            if let Some(value) = value {
                *cell.borrow_mut() = value.into_inner();
            }
        });
    }

    //获取所有权
    pub fn take(&'static self) -> T
    where
        T: Default,
    {
        self.with(|cell| cell.take())
    }

    //替换
    pub fn replace(&'static self, value: T) -> T {
        self.with(|cell| cell.replace(value))
    }
}

/// 惰性初始化.
mod lazy {
    use crate::cell::UnsafeCell;
    use crate::hint;
    use crate::mem;
```

```rust
    pub struct LazyKeyInner<T> {
        //None作为未初始化的标志
        inner: UnsafeCell<Option<T>>,
    }

    impl<T> LazyKeyInner<T> {
        //new一个未初始化变量
        pub const fn new() -> LazyKeyInner<T> {
            LazyKeyInner { inner: UnsafeCell::new(None) }
        }

        pub unsafe fn get(&self) -> Option<&'static T> {
            // 返回内部变量的引用
            unsafe { (*self.inner.get()).as_ref() }
        }

        // 真正的初始化
        pub unsafe fn initialize<F: FnOnce() -> T>(&self, init: F) -> &'static
T {
            let value = init();
            let ptr = self.inner.get();

            // 如果用*ptr = Some(value)，会导致编译器对上一个变量做drop处理,对
Thread Local
            // 的drop实际上有些复杂，所以此处用一个replace
            unsafe {
                let _ = mem::replace(&mut *ptr, Some(value));
            }

            unsafe {
                // 返回Some内变量的引用.
                match *ptr {
                    Some(ref x) => x,
                    None => hint::unreachable_unchecked(),
                }
            }
        }

        //take语义
        pub unsafe fn take(&mut self) -> Option<T> {
            unsafe { (*self.inner.get()).take() }
        }
    }
}

//利用llvm的Thread Local方案，不直接使用操作系统系统调用的thread local key
pub mod fast {
    use super::lazy::LazyKeyInner;
    use crate::cell::Cell;
    use crate::fmt;
```

```rust
    use crate::mem;
    use crate::sys::thread_local_dtor::register_dtor;

    #[derive(Copy, Clone)]
    enum DtorState {
        //没有初始化
        Unregistered,
        //已经初始化完成
        Registered,
        //Local Key已经被destroy
        RunningOrHasRun,
    }

    pub struct Key<T> {
        //  放置key存储的变量，None表示变量没有初始化。与dtor_state配合完成对
        //  Key的状态判断
        inner: LazyKeyInner<T>,

        // Local Key的destroy函数状态
        dtor_state: Cell<DtorState>,
    }


    impl<T> Key<T> {
        pub const fn new() -> Key<T> {
            //实际做内存占位
            Key { inner: LazyKeyInner::new(), dtor_state:
Cell::new(DtorState::Unregistered) }
        }

        // 证明仍然使用操作系统的thread local key的destory机制
        pub unsafe fn register_dtor(a: *mut u8, dtor: unsafe extern "C" fn(*mut
u8)) {
            unsafe {
                register_dtor(a, dtor);
            }
        }

        pub unsafe fn get<F: FnOnce() -> T>(&self, init: F) -> Option<&'static
T> {
            // 如果已经初始化，则取用值
            // 如果没有初始化，则进行初始化
            unsafe {
                match self.inner.get() {
                    Some(val) => Some(val),
                    None => self.try_initialize(init),
                }
            }
        }

        #[inline(never)]
```

```
        unsafe fn try_initialize<F: FnOnce() -> T>(&self, init: F) ->
Option<&'static T> {
            if !mem::needs_drop::<T>() || unsafe { self.try_register_dtor() } {
                // 只用变量不需要drop，或者注册destroy函数成功的情况下才做初始化.
                Some(unsafe { self.inner.initialize(init) })
            } else {
                None
            }
        }

        unsafe fn try_register_dtor(&self) -> bool {
            match self.dtor_state.get() {
                DtorState::Unregistered => {
                    // 注册destroy函数
                    unsafe { register_dtor(self as *const _ as *mut u8,
destroy_value::<T>) };
                    self.dtor_state.set(DtorState::Registered);
                    true
                }
                DtorState::Registered => {
                    // 被递归初始化
                    true
                }
                DtorState::RunningOrHasRun => false,
            }
        }
    }

    unsafe extern "C" fn destroy_value<T>(ptr: *mut u8) {
        let ptr = ptr as *mut Key<T>;

        unsafe {
            //将变量所有权获得
            let value = (*ptr).inner.take();
            //设置destroy状态
            (*ptr).dtor_state.set(DtorState::RunningOrHasRun);
            //对变量做drop
            drop(value);
        }
    }
}

//利用操作系统的StaticKey
pub mod os {
    use super::lazy::LazyKeyInner;
    use crate::cell::Cell;
    use crate::fmt;
    use crate::marker;
    use crate::ptr;
    use crate::sys_common::thread_local_key::StaticKey as OsStaticKey;
```

```rust
    pub struct Key<T> {
        // 操作系统的静态Key.
        os: OsStaticKey,
        //指示本结构有一个Cell<T>的所有权
        marker: marker::PhantomData<Cell<T>>,
    }

    unsafe impl<T> Sync for Key<T> {}

    struct Value<T: 'static> {
        inner: LazyKeyInner<T>,
        key: &'static Key<T>,
    }

    impl<T: 'static> Key<T> {
        pub const fn new() -> Key<T> {
            //创建一个StaticKey
            Key { os: OsStaticKey::new(Some(destroy_value::<T>)), marker:
marker::PhantomData }
        }

        pub unsafe fn get(&'static self, init: impl FnOnce() -> T) ->
Option<&'static T> {
            // thread local key的get操作.
            let ptr = unsafe { self.os.get() as *mut Value<T> };
            if ptr.addr() > 1 {
                //有值
                if let Some(ref value) = unsafe { (*ptr).inner.get() } {
                    return Some(value);
                }
            }
            // 进行初始化.
            unsafe { self.try_initialize(init) }
        }

        unsafe fn try_initialize(&'static self, init: impl FnOnce() -> T) ->
Option<&'static T> {
            let ptr = unsafe { self.os.get() as *mut Value<T> };
            if ptr.addr() == 1 {
                // 被destroy了
                return None;
            }

            let ptr = if ptr.is_null() {
                // 从堆上申请内存.
                let ptr: Box<Value<T>> = box Value { inner:
LazyKeyInner::new(), key: self };
                //获取申请的堆内存地址
                let ptr = Box::into_raw(ptr);
                // 将地址与操作系统的key相关联
                unsafe {
```

```
                    self.os.set(ptr as *mut u8);
                }
                ptr
            } else {
                // 递归初始化，返回已有的ptr
                ptr
            };

            // 初始化变量.
            unsafe { Some((*ptr).inner.initialize(init)) }
        }
    }

    unsafe extern "C" fn destroy_value<T: 'static>(ptr: *mut u8) {
        unsafe {
            //恢复Box以便释放堆内存
            let ptr = Box::from_raw(ptr as *mut Value<T>);
            let key = ptr.key;
            //将thread local key设置为1
            key.os.set(ptr::invalid_mut(1));
            //释放Box
            drop(ptr);
            // key可以重新用于与新的内存相关
            key.os.set(ptr::null_mut());
        }
    }
}
pub use self::local::fast::Key as __FastLocalKeyInner;
pub use self::local::os::Key as __OsLocalKeyInner;

// Thread Local声明宏
macro_rules! thread_local {
    // empty (base case for the recursion)
    () => {};

    // init 是一个const 修饰的block
    ($(#[$attr:meta])* $vis:vis static $name:ident: $t:ty = const { $init:expr
}; $($rest:tt)*) => (
        $crate::__thread_local_inner!($(#[$attr])* $vis $name, $t, const
$init);
        $crate::thread_local!($($rest)*);
    );

    ($(#[$attr:meta])* $vis:vis static $name:ident: $t:ty = const { $init:expr
}) => (
        $crate::__thread_local_inner!($(#[$attr])* $vis $name, $t, const
$init);
    );

    //  init不是block
    ($(#[$attr:meta])* $vis:vis static $name:ident: $t:ty = $init:expr;
```

```
    $($rest:tt)*) => (
            $crate::__thread_local_inner!($(#[$attr])* $vis $name, $t, $init);
            $crate::thread_local!($($rest)*);
        );

        ($(#[$attr:meta])* $vis:vis static $name:ident: $t:ty = $init:expr) => (
            $crate::__thread_local_inner!($(#[$attr])* $vis $name, $t, $init);
        );
    }

    macro_rules! __thread_local_inner {
        ($(#[$attr:meta])* $vis:vis $name:ident, $t:ty, $($init:tt)*) => {
            //定义了一个LocalKey的变量
            $(#[$attr])* $vis const $name: $crate::thread::LocalKey<$t> =
                $crate::__thread_local_inner!(@key $t, $($init)*);
        }
        //对const init的处理
        (@key $t:ty, const $init:expr) => {{
            //LocalKey的创建函数
            unsafe fn __getit(
                _init: $crate::option::Option<&mut $crate::option::Option<$t>>,
            ) -> $crate::option::Option<&'static $t> {
                //不可变变量定义
                const INIT_EXPR: $t = $init;

                {
                    #[thread_local]
                    //静态全局变量，应用llvm的Thread Local Storage，需要操作系统支持
                    static mut VAL: $t = INIT_EXPR;

                    // 判断是否需要有drop函数
                    if !$crate::mem::needs_drop::<$t>() {
                        //如果不需要drop，返回VAL的引用即可
                        unsafe {
                            return $crate::option::Option::Some(&VAL)
                        }
                    }

                    // 0 == dtor not registered
                    // 1 == dtor registered, dtor not run
                    // 2 == dtor registered and is running or has run
                    #[thread_local]
                    //释放函数注册状态
                    static mut STATE: $crate::primitive::u8 = 0;

                    //释放函数
                    unsafe extern "C" fn destroy(ptr: *mut $crate::primitive::u8) {
                        let ptr = ptr as *mut $t;

                        unsafe {
                            $crate::debug_assert_eq!(STATE, 1);
```

```
                        STATE = 2;
                        $crate::ptr::drop_in_place(ptr);
                    }
                }

                unsafe {
                    match STATE {
                        // 0 == 需要注册释放函数.
                        0 => {
                            //fast::Key::register_dtor，见下文分析
                            $crate::thread::__FastLocalKeyInner::
<$t>::register_dtor(
                                $crate::ptr::addr_of_mut!(VAL) as *mut
$crate::primitive::u8,
                                destroy,
                            );
                            STATE = 1;
                            $crate::option::Option::Some(&VAL)
                        }
                        // 1 == 释放函数已经注册，直接返回Key
                        1 => $crate::option::Option::Some(&VAL),
                        // 释放函数已经运行，返回.
                        _ => $crate::option::Option::None,
                    }
                }
            }
        }

        unsafe {
            //生成LocalKey
            $crate::thread::LocalKey::new(__getit)
        }
    }};

    // 非const的init的处理
    (@key $t:ty, $init:expr) => {
        {
            fn __init() -> $t { $init }

            //LocalKey的创建函数
            unsafe fn __getit(
                init: $crate::option::Option<&mut $crate::option::Option<$t>>,
            ) -> $crate::option::Option<&'static $t> {
                //利用llvm编译器属性定义变量为thread local变量
                #[thread_local]
                static __KEY: $crate::thread::__FastLocalKeyInner<$t> =
                    $crate::thread::__FastLocalKeyInner::new();


                //初始化
                unsafe {
```

```
                    __KEY.get(move || {
                        if let $crate::option::Option::Some(init) = init {
                            if let $crate::option::Option::Some(value) =
init.take() {
                                return value;
                            } else if $crate::cfg!(debug_assertions) {
                                $crate::unreachable!("missing default value");
                            }
                        }
                        __init()
                    })
                }
            }

            unsafe {
                $crate::thread::LocalKey::new(__getit)
            }
        }
    };
}
```

Thread Local的RUST标准库内容颇为复杂，但提供了非常方便的对外使用。

Thread Info 分析

Thead Info利用Thread Local存储一些Thread的信息。

```
struct ThreadInfo {
    stack_guard: Option<Guard>,
    thread: Thread,
}

thread_local! { static THREAD_INFO: RefCell<Option<ThreadInfo>> = const {
RefCell::new(None) } }

impl ThreadInfo {
    fn with<R, F>(f: F) -> Option<R>
    where
        F: FnOnce(&mut ThreadInfo) -> R,
    {
        THREAD_INFO
            .try_with(move |thread_info| {
                let mut thread_info = thread_info.borrow_mut();
                let thread_info = thread_info.get_or_insert_with(|| ThreadInfo
{
                    stack_guard: None,
                    thread: Thread::new(None),
                });
                f(thread_info)
```

```
            })
            .ok()
    }
}

pub fn current_thread() -> Option<Thread> {
    ThreadInfo::with(|info| info.thread.clone())
}

pub fn stack_guard() -> Option<Guard> {
    ThreadInfo::with(|info| info.stack_guard.clone()).and_then(|o| o)
}

pub fn set(stack_guard: Option<Guard>, thread: Thread) {
    THREAD_INFO.with(move |thread_info| {
        let mut thread_info = thread_info.borrow_mut();
        rtassert!(thread_info.is_none());
        *thread_info = Some(ThreadInfo { stack_guard, thread });
    });
}
```

标准库线程外部接口分析

RUST在操作系统的线程支持之上，实现了语言自身的线程概念，首要的，就是为每一个线程分配一个ID做标识。 即ThreadId, 代码如下：

```
//用一个非零的64位整数
pub struct ThreadId(NonZeroU64);

impl ThreadId {
    // 分配一个新的线程ID
    fn new() -> ThreadId {
        // 线程ID是一个全局静态变量，且是一个临界区访问，此处只能用StaticMutex锁
        static GUARD: mutex::StaticMutex = mutex::StaticMutex::new();
        //声明静态变量,是全局的静态变量，但声明在这里限制对其的访问，初始值为1
        static mut COUNTER: u64 = 1;

        unsafe {
            //防止竞争
            let guard = GUARD.lock();

            // 加入到达最大值，panic处理。这里默认一个进程不可能创建超过u64::MAX的线程.
            // 这个稍微有些不严谨，因为即使线程终止，ID也不能重用。可以认为，一个进程不可能运行到这个时间
            if COUNTER == u64::MAX {
                drop(guard); //panic之前显式drop，以避免影响其他线程
                //错误处理
```

```
                panic!("failed to generate unique thread ID: bitspace
exhausted");
            }

            //分配新线程ID
            let id = COUNTER;
            COUNTER += 1;

            ThreadId(NonZeroU64::new(id).unwrap())
            //guard生命周期结束，会调用drop()
        }
    }
    ...
}
```

Thread park的实现，Thread park是一种将线程自身陷入阻塞，等待别的线程做唤醒的机制。是一种比较简单的多个线程间的同步机制，通常用于线程指令执行过程中有顺序要求但不必临界区的情况

```
const PARKED: i32 = -1;
const EMPTY: i32 = 0;
const NOTIFIED: i32 = 1;

pub struct Parker {
    state: AtomicI32,
}

// Parker 利用原子变量操作中的内存顺序规则完成.
impl Parker {
    #[inline]
    pub const fn new() -> Self {
        Parker { state: AtomicI32::new(EMPTY) }
    }

    //只被本线程调用
    pub unsafe fn park(&self) {
        // 利用Acquire顺序获取当前状态
        if self.state.fetch_sub(1, Acquire) == NOTIFIED {
            return;
        }
        loop {
            // 如果state是PARKED，阻塞等待
            futex_wait(&self.state, PARKED, None);
            // 被唤醒，将状态重新置为EMPTY,并检测是否为NOTIFIED.
            if self.state.compare_exchange(NOTIFIED, EMPTY, Acquire,
Acquire).is_ok() {
                return;
            } else {
                // 应该是语法要求，逻辑上不应该进入此分支，因为不应该有其他线程调用
```

```
park.
            }
        }
    }

    // 超时.
    pub unsafe fn park_timeout(&self, timeout: Duration) {
        if self.state.fetch_sub(1, Acquire) == NOTIFIED {
            return;
        }
        // 直接等待，设置超时，此时不再循环，因为循环会导致超时不准.
        futex_wait(&self.state, PARKED, Some(timeout));
        if self.state.swap(EMPTY, Acquire) == NOTIFIED {
            // 被unpark()唤醒
        } else {
            // 超时或者其他唤醒.
        }
    }

    pub fn unpark(&self) {
        // 将state更换到NOTIFIED
        if self.state.swap(NOTIFIED, Release) == PARKED {
            //唤醒阻塞的线程
            futex_wake(&self.state);
        }
    }
}
```

RUST标准库的Thread的对外结构:

```
/// 事实上的Thread结构
struct Inner {
    name: Option<CString>, //需要与外部语言库交互
    id: ThreadId,
    // 用于park
    parker: Parker,
}

//Thread的管理类型
#[derive(Clone)]
pub struct Thread {
    //需要被多个线程共享
    inner: Arc<Inner>,
}

impl Thread {
    // 仅创建一个结构用于管理，此时尚没有与线程相关联.
    pub(crate) fn new(name: Option<CString>) -> Thread {
        Thread { inner: Arc::new(Inner { name, id: ThreadId::new(), parker:
Parker::new() }) }
```

```
    }

    //对thread做unpark操作，使得park的thread结束阻塞。
    pub fn unpark(&self) {
        self.inner.parker.unpark();
    }

    pub fn id(&self) -> ThreadId {
        self.inner.id
    }

    pub fn name(&self) -> Option<&str> {
        self.cname().map(|s| unsafe { str::from_utf8_unchecked(s.to_bytes()) })
    }

    fn cname(&self) -> Option<&CStr> {
        self.inner.name.as_deref()
    }
}
```

RUST的进程创建返回类型 JoinHandle：

```
struct JoinInner<'scope, T> {
    native: imp::Thread,
    thread: Thread,
    packet: Arc<Packet<'scope, T>>,
}

impl<'scope, T> JoinInner<'scope, T> {
    fn join(mut self) -> Result<T> {
        //等待线程退出
        self.native.join();
        //获取线程退出的结果或者异常信息
        Arc::get_mut(&mut
self.packet).unwrap().result.get_mut().take().unwrap()
    }
}

//调用spawn后返回JoinHandle，JoinHandle作为线程外部对该线程操作的标识类型结构,如
park，join等
pub struct JoinHandle<T>(JoinInner<'static, T>);

unsafe impl<T> Send for JoinHandle<T> {}
unsafe impl<T> Sync for JoinHandle<T> {}


impl<T> JoinHandle<T> {
    //获取线程的Thread结构引用
    pub fn thread(&self) -> &Thread {
        &self.0.thread
```

```
    }

    //等待线程结束
    pub fn join(self) -> Result<T> {
        self.0.join()
    }

    //判断线程是否已经终止
    pub fn is_finished(&self) -> bool {
        Arc::strong_count(&self.0.packet) == 1
    }
}
```

RUST线程创建工厂类型：

```
//用于非默认属性的线程创建
pub struct Builder {
    // 名字，线程默认没有名字
    name: Option<String>,
    // 线程堆栈大小，默认堆栈为2M bytes
    stack_size: Option<usize>,
}

// 线程创建方法实现
impl Builder {
    //创建一个默认的builder，一般的，用于需要对name及stack_size做修改
    pub fn new() -> Builder {
        Builder { name: None, stack_size: None }
    }

    //给线程设置名称，目前仅用于线程panic时的信息输出
    pub fn name(mut self, name: String) -> Builder {
        self.name = Some(name);
        self
    }

    //设置线程的堆栈空间
    pub fn stack_size(mut self, size: usize) -> Builder {
        self.stack_size = Some(size);
        self
    }

    // 利用Builder属性参数创建一个新线程。如果不是在主线程执行这个函数，
    // 新线程的生命周期可能长于创建它的线程，此时创建线程可以用JoinHandle来等待
    // 新线程结束。
    pub fn spawn<F, T>(self, f: F) -> io::Result<JoinHandle<T>>
    where
        F: FnOnce() -> T,
        F: Send + 'static,
```

```
        T: Send + 'static,
    {
        unsafe { self.spawn_unchecked(f) }
    }

    //不安全的spawn
    pub unsafe fn spawn_unchecked<'a, F, T>(self, f: F) ->
io::Result<JoinHandle<T>>
    where
        F: FnOnce() -> T,
        F: Send + 'a,
        T: Send + 'a,
    {
        Ok(JoinHandle(unsafe { self.spawn_unchecked_(f, None) }?))
    }

    //真正的spawn执行函数，此处与进程的spawn函数有些类似
    unsafe fn spawn_unchecked_<'a, 'scope, F, T>(
        self,
        f: F,
        scope_data: Option<&'scope scoped::ScopeData>,
    ) -> io::Result<JoinInner<'scope, T>>
    where
        F: FnOnce() -> T,
        F: Send + 'a,
        T: Send + 'a,
        'scope: 'a,
    {
        let Builder { name, stack_size } = self;

        //不能小于规定的最小堆栈
        let stack_size = stack_size.unwrap_or_else(thread::min_stack);

        //创建一个Thread的变量
        let my_thread = Thread::new(name.map(|name| {
            CString::new(name).expect("thread name may not contain interior
null bytes")
        }));
        //增加Arc计数，用于转移到创建的线程代码
        let their_thread = my_thread.clone();

        let my_packet: Arc<Packet<'scope, T>> =
            Arc::new(Packet { scope: scope_data, result: UnsafeCell::new(None)
});
        //Arc计数增加，子线程写，父线程读
        let their_packet = my_packet.clone();

        //捕获panic输出的缓存空间设置,是Thread Local变量
        let output_capture = crate::io::set_output_capture(None);
        crate::io::set_output_capture(output_capture.clone());
```

```
        //所有线程的主函数，可以认为是线程的runtime
        let main = move || {
            //their_thread已经转移到创建线程
            if let Some(name) = their_thread.cname() {
                //设置线程名字
                imp::Thread::set_name(name);
            }

            //设置本线程的panic捕获空间
            crate::io::set_output_capture(output_capture);

            // 完成thread_info的线程本地初始化.
            thread_info::set(unsafe { imp::guard::current() }, their_thread);
            // 执行f，如果f内部发生panic，调用栈会输出
            let try_result = panic::catch_unwind(panic::AssertUnwindSafe(|| {
                crate::sys_common::backtrace::__rust_begin_short_backtrace(f)
            }));
            //  将线程退出信息设置到their_packet中
            unsafe { *their_packet.result.get() = Some(try_result) };
        };

        if let Some(scope_data) = scope_data {
            scope_data.increment_num_running_threads();
        }

        //真正的创建线程
        Ok(JoinInner {
            //创建线程
            native: unsafe {
                imp::Thread::new(
                    stack_size,
                    mem::transmute::<Box<dyn FnOnce() + 'a>, Box<dyn FnOnce() +
'static>>(
                        Box::new(main),
                    ),
                )?
            },
            thread: my_thread,
            packet: my_packet,
        })
    }
}

//无须指定参数的简易线程启动函数
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
{
    Builder::new().spawn(f).expect("failed to spawn thread")
```

```rust
}

//线程自身的结构变量获取
pub fn current() -> Thread {
    thread_info::current_thread().expect(
        "use of std::thread::current() is not possible \
         after the thread's local data has been destroyed",
    )
}

//出让CPU
pub fn yield_now() {
    imp::Thread::yield_now()
}

//本线程是否已经panic
pub fn panicking() -> bool {
    panicking::panicking()
}

//阻塞，等待其他线程唤醒
pub fn park() {
    unsafe {
        current().inner.parker.park();
    }
}

pub fn park_timeout(dur: Duration) {
    unsafe {
        current().inner.parker.park_timeout(dur);
    }
}

pub type Result<T> = crate::result::Result<T, Box<dyn Any + Send + 'static>>;

// 用来获取线程的退出值
// 需要在线程间共享
struct Packet<'scope, T> {
    scope: Option<&'scope scoped::ScopeData>,
    result: UnsafeCell<Option<Result<T>>>,
}

// 使用了UnsafeCell， 需要声明实现Sync
unsafe impl<'scope, T: Sync> Sync for Packet<'scope, T> {}

impl<'scope, T> Drop for Packet<'scope, T> {
    fn drop(&mut self) {
        // If this packet was for a thread that ran in a scope, the thread
        // panicked, and nobody consumed the panic payload, we make sure
        // the scope function will panic.
        let unhandled_panic = matches!(self.result.get_mut(), Some(Err(_)));
```

```
        // Drop the result without causing unwinding.
        // This is only relevant for threads that aren't join()ed, as
        // join() will take the `result` and set it to None, such that
        // there is nothing left to drop here.
        // If this panics, we should handle that, because we're outside the
        // outermost `catch_unwind` of our thread.
        // We just abort in that case, since there's nothing else we can do.
        // (And even if we tried to handle it somehow, we'd also need to handle
        // the case where the panic payload we get out of it also panics on
        // drop, and so on. See issue #86027.)
        if let Err(_) = panic::catch_unwind(panic::AssertUnwindSafe(|| {
            *self.result.get_mut() = None;
        })) {
            rtabort!("thread result panicked on drop");
        }
        // Book-keeping so the scope knows when it's done.
        if let Some(scope) = self.scope {
            // 在scope spawn线程时，在此处保证唤醒park的主线程
            scope.decrement_num_running_threads(unhandled_panic);
        }
    }
}
```

RUST线程 scope 结构,因为线程的主函数是闭包函数，对所有环境变量都是以借用引入，这会导致 因为线程不知道何时结束而出现环境变量的生命周期问题，利用scope使得线程闭包可以正常借用环境变量

```
pub struct Scope<'scope, 'env: 'scope> {
    //主要用来做生命周期的保证
    data: ScopeData,
    //指示线程的生命周期
    scope: PhantomData<&'scope mut &'scope ()>,
    //指示环境变量的生命周期
    env: PhantomData<&'env mut &'env ()>,
}

/// JoinHandle的scoped 版本
pub struct ScopedJoinHandle<'scope, T>(JoinInner<'scope, T>);

pub(super) struct ScopeData {
    num_running_threads: AtomicUsize,
    a_thread_panicked: AtomicBool,
    main_thread: Thread,
}

impl ScopeData {
    pub(super) fn increment_num_running_threads(&self) {
        //spawn线程时增加计数
        if self.num_running_threads.fetch_add(1, Ordering::Relaxed) >
```

```
usize::MAX / 2 {
            self.decrement_num_running_threads(false);
            panic!("too many running threads in thread scope");
        }
    }
    pub(super) fn decrement_num_running_threads(&self, panic: bool) {
        if panic {
            self.a_thread_panicked.store(true, Ordering::Relaxed);
        }
        //减少线程计数
        if self.num_running_threads.fetch_sub(1, Ordering::Release) == 1 {
            //唤醒主线程
            self.main_thread.unpark();
        }
    }
}

// socpe内部创建的线程可以借用非静态变量
// 其中，'env是环境变量的生命周期，'scope是线程的生命周期
pub fn scope<'env, F, T>(f: F) -> T
where
    F: for<'scope> FnOnce(&'scope Scope<'scope, 'env>) -> T,
{
    let scope = Scope {
        data: ScopeData {
            num_running_threads: AtomicUsize::new(0),
            main_thread: current(),
            a_thread_panicked: AtomicBool::new(false),
        },
        env: PhantomData,
        scope: PhantomData,
    };

    // Run `f`, but catch panics so we can make sure to wait for all the
threads to join.
    let result = catch_unwind(AssertUnwindSafe(|| f(&scope)));

    // 等待所有的线程都退出，保证线程的生命周期小于本函数的生命周期.
    while scope.data.num_running_threads.load(Ordering::Acquire) != 0 {
        park();
    }

    // Throw any panic from `f`, or the return value of `f` if no thread
panicked.
    match result {
        Err(e) => resume_unwind(e),
        Ok(_) if scope.data.a_thread_panicked.load(Ordering::Relaxed) => {
            panic!("a scoped thread panicked")
        }
        Ok(result) => result,
    }
```

```rust
}

impl<'scope, 'env> Scope<'scope, 'env> {
    //用于在scope中创建新线程
    pub fn spawn<F, T>(&'scope self, f: F) -> ScopedJoinHandle<'scope, T>
    where
        F: FnOnce() -> T + Send + 'scope,
        T: Send + 'scope,
    {
        Builder::new()
            .spawn_scoped(self, f)
            .expect("failed to spawn thread")
    }
}

impl Builder {
    // 在scope情况下创建线程
    pub fn spawn_scoped<'scope, 'env, F, T>(
        self,
        scope: &'scope Scope<'scope, 'env>,
        f: F,
    ) -> io::Result<ScopedJoinHandle<'scope, T>>
    where
        //设置了生命周期，使得f可以使用环境变量引用
        F: FnOnce() -> T + Send + 'scope,
        T: Send + 'scope,
    {
        Ok(ScopedJoinHandle(unsafe {
            self.spawn_unchecked_(f, Some(&scope.data))
        }?))
    }
}

//利用生命周期的标注来使用环境变量引用
impl<'scope, T> ScopedJoinHandle<'scope, T> {

    pub fn thread(&self) -> &Thread {
        &self.0.thread
    }

    pub fn join(self) -> Result<T> {
        self.0.join()
    }

    pub fn is_finished(&self) -> bool {
        Arc::strong_count(&self.0.packet) == 1
    }
}
```