RUST线程间消息通信

在网络操作系统中，线程间使用消息通信被广泛采用，甚至线程间通信仅使用消息机制。主要因为如果线程之间仅使用消息机制的话，即基本可以保证没有临界区，从而减少内存安全问题的情况。一般的，针对每个线程创建一个多个生产者，单个消费者的消息队列，消费者绑定在这个线程上，其他需要与此线程通信的线程是生产者。这种系统一般会确定一个通用的消息协议格式。 消息通信的方式需要尽量规避过长的消息内容。

代码路径：library/std/src/sync/mpsc/.

本书将只讨论mpsc这一机制，spsc的分析留给读者。
RUST将通信分成了三种情况：

1. 最初建立连接时，默认为仅做一次发送，接收，即oneshot通道形式
2. 如果发送多于一个包，但收线程及发线程都固定为同一个，则升级为stream通道形式
3. 如果发送线程多于一个，则升级为shared通道形式

采用如此复杂的情况，虽然有合理的成分，但感觉标准库的作者实际上是在炫技，并且不想被人轻易的理解其思路及想法。实际上，统一使用shared的形式即可靠，又简单。因为升级这个过程实际上极易引发问题。 mpsc模块中复杂的主要结构类型如下：

1. Queue结构，用于消费者及接受者之间存储消息的队列，是满足Sync的类型结构
2. SignalToken/WaitToken结构，用于解除接收线程的阻塞信号
3. `oneshot::Packet<T>` oneshot类型的channel机制
4. `shared::Packet<T>` shared类型的channel机制
5. `Sender<Flavor<T>>`，`Receiver<Flavor<T>>`是接收及发送的端口

我们将分节对其进行介绍

消息队列数据结构实现

多于一个消息包的时候，需要消息队列，RUST用于消息包的队列结构是一个无锁的，无阻塞的临界区队列，非常巧妙的设计，是需要牢记在心的, 充分体现了RUST标准库开发人员高超的编程技巧。

```
//以下是简单的FIFO的队列实现
pub enum PopResult<T> {
    //返回队列成员
    Data(T),
    //队列为空
    Empty,
    //在有些时刻会出现瞬间的不一致情况
```

```
    Inconsistent,
}

//节点结构
struct Node<T> {
    //next指针,利用原子指针实现多线程的Sync，值得牢记
    next: AtomicPtr<Node<T>>,
    value: Option<T>,
}

///  能够被多个线程操作的队列
pub struct Queue<T> {
    //利用原子指针操作实现多线程的Sync，极大简化了代码
    head: AtomicPtr<Node<T>>,
    //从后面的代码看，这里实际上是队列的头部，这个Queue的代码搞得奇怪
    tail: UnsafeCell<*mut Node<T>>,
}

unsafe impl<T: Send> Send for Queue<T> {}
unsafe impl<T: Send> Sync for Queue<T> {}

impl<T> Node<T> {
    unsafe fn new(v: Option<T>) -> *mut Node<T> {
        //申请堆内存后，将堆内存的指针提取出来
        Box::into_raw(box Node { next: AtomicPtr::new(ptr::null_mut()), value:
v })
    }
}

impl<T> Queue<T> {
    pub fn new() -> Queue<T> {
        let stub = unsafe { Node::new(None) };
        //生成一个空元素的节点列表
        Queue { head: AtomicPtr::new(stub), tail: UnsafeCell::new(stub) }
    }

    //在头部
    pub fn push(&self, t: T) {
        unsafe {
            let n = Node::new(Some(t));
            //换成C的话，就是head->next = n; head = n
            //对于空队列来说，是tail = head; head->next = n; head = n;
            //现在tail实际上是队列头部，head是尾部。tail的next是第一个有意义的成员
            let prev = self.head.swap(n, Ordering::AcqRel);
            //要考虑在两个赋值中间加入了其他线程的操作是否会出问题，
            //这里面有一个复杂的分析，
            //假设原队列为head，有两个线程分别插入新节点n,m
            //当n先执行，而m在这个代码位置插入，则m插入前prev_n = pre_head，head =
n
            //m插入后，prev_m = n，head = m。如果n先执行下面的语句，执行完后
            // pre_head->next = n，n->next = null，然后m执行完下面语句
```

```
            // pre_head->next = n, n->next = m, head = m, 队列是正确的。
            // 如果m先执行，执行完后 pre_head->next = null, n->next = m, head =
m;
            // 然后n执行，执行完成后 pre_head->next = n, n->next = m, head =m,  队
列是正确的。
            // 换成多个线程实际上也一样是正确的。这个地方处理十分巧妙，这是系统级编程
语言的魅
            //力，当然，实际上是裸指针编程的魅力
            //当然，在这个过程中会出现Inconsistent
            (*prev).next.store(n, Ordering::Release);

        }
    }

    //仅有一个线程在pop
    pub fn pop(&self) -> PopResult<T> {
        unsafe {
            //tail实际上是队列头，value是None
            let tail = *self.tail.get();
            //tail的next是第一个有意义的成员
            let next = (*tail).next.load(Ordering::Acquire);

            //next如果为空，说明队列是空队列
            if !next.is_null() {
                //此处原tail会被drop，tail被赋成next
                //因为push只可能改变next，所以这里不会有线程冲突问题
                //这个语句完成后，队列是完整及一致的
                *self.tail.get() = next;
                assert!((*tail).value.is_none());
                assert!((*next).value.is_some());
                //将value的所有权转移出来，*next的value又重新置为None
                //当tail == head的时候 就又都是stub了
                let ret = (*next).value.take().unwrap();
                //恢复Box，以便以后释放堆内存
                let _: Box<Node<T>> = Box::from_raw(tail);
                return Data(ret);
            }

            // 此时如果head不是tail，一般说明有线程正在push，出现了不一致的情况,但这
个不一致
            // 随着另一线程插入的结束会终结
            if self.head.load(Ordering::Acquire) == tail { Empty } else {
Inconsistent }
        }
    }
}

impl<T> Drop for Queue<T> {
    fn drop(&mut self) {
        unsafe {
            //空队列的stub也要释放
```

```
            let mut cur = *self.tail.get();
            while !cur.is_null() {
                let next = (*cur).next.load(Ordering::Relaxed);
                //恢复Box并消费掉，释放堆内存
                let _: Box<Node<T>> = Box::from_raw(cur);
                cur = next;
            }
        }
    }
}
```

线程间简单的阻塞及唤醒信号机制

消息通信时，消息发送端需要有一个机制通知消息接收端消息已经发出。Condvar可以完成这一工作，但RUST的消息机制决定用无锁设计，所以做了新的实现。
下面的设计具有通用性，正如上节的Queue。基本思路是：

1. 设计多个线程间的信号结构, 只允许一个线程等待在信号上，可以有多个线程触发信号解锁
2. 利用原子变量的变化来做等待及信号等待

代码如下：

```
//线程间共享的信号结构
struct Inner {
    //指明执行信号等待的线程
    thread: Thread,
    //标志解除等待信号发送
    woken: AtomicBool,
}

unsafe impl Send for Inner {}
unsafe impl Sync for Inner {}

//信号发送端结构
pub struct SignalToken {
    inner: Arc<Inner>,
}

//信号接收端结构
pub struct WaitToken {
    inner: Arc<Inner>,
}

impl !Send for WaitToken {}

impl !Sync for WaitToken {}
```

```rust
//信号对创建函数,由信号等待端线程创建
pub fn tokens() -> (WaitToken, SignalToken) {
    //初始为无信号
    let inner = Arc::new(Inner { thread: thread::current(), woken:
AtomicBool::new(false) });
    // wait由线程本身使用
    let wait_token = WaitToken { inner: inner.clone() };
    // signal由其他线程使用
    let signal_token = SignalToken { inner };
    (wait_token, signal_token)
}

impl SignalToken {
    //发送信号以便唤醒等待线程
    pub fn signal(&self) -> bool {
        //更改原子变量，看是否处于等待信号状态
        let wake = self
            .inner
            .woken
            .compare_exchange(false, true, Ordering::SeqCst, Ordering::SeqCst)
            .is_ok();
        if wake {
            //更改成功，接收线程会调用park阻塞，unpark解除接收线程阻塞
            self.inner.thread.unpark();
        }
        wake
    }

    //传递给其他线程以便用来生成SignalToken，此处只能用
    //裸指针，这里是传递没有被智能指针封装的堆内存指针
    pub unsafe fn to_raw(self) -> *mut u8 {
        Arc::into_raw(self.inner) as *mut u8
    }

    //从to_raw生成的堆内存指针恢复为SignalToken,由发送线程完成
    pub unsafe fn from_raw(signal_ptr: *mut u8) -> SignalToken {
        SignalToken { inner: Arc::from_raw(signal_ptr as *mut Inner) }
    }
}

impl WaitToken {
    //接收线程等待发送端信号
    pub fn wait(self) {
        //必须先对woken做过设置
        while !self.inner.woken.load(Ordering::SeqCst) {
            thread::park()
        }
    }

    //设置超时的等待，请参考线程锁那一节的park内容
    pub fn wait_max_until(self, end: Instant) -> bool {
```

```
        while !self.inner.woken.load(Ordering::SeqCst) {
            let now = Instant::now();
            if now >= end {
                return false;
            }
            thread::park_timeout(end - now)
        }
        true
    }
}
```

oneshot通道机制实现

oneshot专门为收发一次消息包而优化的结构。

```
//以下用于标识通道的状态
//没有数据包
const EMPTY: *mut u8 = ptr::invalid_mut::<u8>(0);
//有数据包等待被接收
const DATA: *mut u8 = ptr::invalid_mut::<u8>(1);
//中断
const DISCONNECTED: *mut u8 = ptr::invalid_mut::<u8>(2);
// 其他值(ptr)代表接收者信号结构变量的指针，说明有接收者在等待接收

//消息包结构，因为只有一次收及一次发，所以结构中除state外
//其他不涉及数据竞争
pub struct Packet<T> {
    // 通道状态，取值为EMPTY/DATA/DISCONNECTED/ptr
    state: AtomicPtr<u8>,
    // 通道内的数据，此数据需要从发送者拷贝到此处，再拷贝到接受者，但因为仅有一个包
    // 所以性能不是关注要点
    data: UnsafeCell<Option<T>>,
    // 当发送第二个包，或者对Sender做clone时，需要进行升级,此处放置新的通道接收
Receiver结构
    // 拥有所有权
    upgrade: UnsafeCell<MyUpgrade<T>>,
}

//接收时发生的错误类型结构
pub enum Failure<T> {
    //空错误
    Empty,
    //连接中断
    Disconnected,
    //升级中,发送线程会把ReceiverT发送过来
    Upgraded(Receiver<T>),
}

pub enum UpgradeResult {
```

```
    //已经成功升级为其他类型的通道
    UpSuccess,
    // 升级遇到Disconnected
    UpDisconnected,
    //接收线程阻塞及期望接收的信号
    UpWoke(SignalToken),
}

enum MyUpgrade<T> {
    //通道内没有包，可以不升级
    NothingSent,
    //通道内已经发送过包，需要考虑升级
    SendUsed,
    //通道已经被通知需要升级，升级后的端口在参数中
    GoUp(Receiver<T>),
}

impl<T> Packet<T> {
    //创建一个通道,所有的内容都是初始化值
    pub fn new() -> Packet<T> {
        Packet {
            data: UnsafeCell::new(None),
            upgrade: UnsafeCell::new(NothingSent),
            state: AtomicPtr::new(EMPTY),
        }
    }

    //发送线程通过Sender端口发送包,发送线程应保证只调用一次
    pub fn send(&self, t: T) -> Result<(), T> {
        unsafe {
            //检查是否已经有包发过了
            match *self.upgrade.get() {
                //没有包发送过，则继续执行
                NothingSent => {}
                //不应该执行到此处，应该先升级再发送
                _ => panic!("sending on a oneshot that's already sent on "),
            }
            assert!((*self.data.get()).is_none());
            //拷贝消息包内容
            ptr::write(self.data.get(), Some(t));
            //设置upgrade为已经发送过包,
            ptr::write(self.upgrade.get(), SendUsed);

            //更新state
            match self.state.swap(DATA, Ordering::SeqCst) {
                // 此时可以正常发送，state设置为有数据状态
                EMPTY => Ok(()),

                // 表明接收端已经destroy通道,
                DISCONNECTED => {
                    //需要state恢复成中断
```

```
                    self.state.swap(DISCONNECTED, Ordering::SeqCst);
                    //需要恢复upgrade,
                    ptr::write(self.upgrade.get(), NothingSent);
                    //需要将消息包数据回收,并返回发送出错
                    Err((&mut *self.data.get()).take().unwrap())
                }

                // 不应该到达这一步
                DATA => unreachable!(),

                // 有线程等待接收.
                ptr => {
                    //通知接收线程解除阻塞
                    SignalToken::from_raw(ptr).signal();
                    Ok(())
                }
            }
        }
    }

    // 测试是否已经发过消息包
    pub fn sent(&self) -> bool {
        unsafe { !matches!(*self.upgrade.get(), NothingSent) }
    }

    //接收线程通过Receiver接收
    pub fn recv(&self, deadline: Option<Instant>) -> Result<T, Failure<T>> {
        // 尽量不阻塞线程
        if self.state.load(Ordering::SeqCst) == EMPTY {
            //消息为空，需要阻塞，生成信号通知对
            let (wait_token, signal_token) = blocking::tokens();
            //获取信号发送端的堆内存
            let ptr = unsafe { signal_token.to_raw() };

            // 设置状态为有线程在等待接收
            if self.state.compare_exchange(EMPTY, ptr, Ordering::SeqCst,
Ordering::SeqCst).is_ok() {
                //设置成功，判断是否有超时
                if let Some(deadline) = deadline {
                    //设置超时，阻塞
                    let timed_out = !wait_token.wait_max_until(deadline);
                    // 判断是否超时
                    if timed_out {
                        //如果超时，做清理，如果发送端通知升级，则形成
Upgraded(Receiver<T>)
                        // 这里的map_err(Upgraded)构建了Upgraded(Receiver<T>)，需
要记住
                        self.abort_selection().map_err(Upgraded)?;
                    }
                    //被接收线程唤醒
                } else {
```

```
                //没有设置超时，一直阻塞等待
                wait_token.wait();
                debug_assert!(self.state.load(Ordering::SeqCst) != EMPTY);
            }
        } else {
            //失败，清理信号
            drop(unsafe { SignalToken::from_raw(ptr) });
        }
        //wait_token及signal_token都生命周期终止
    }

    //此时已经有数据了
    self.try_recv()
}

pub fn try_recv(&self) -> Result<T, Failure<T>> {
    unsafe {
        match self.state.load(Ordering::SeqCst) {
            //数据为空，返回错误
            EMPTY => Err(Empty),

            //发现数据
            DATA => {
                //修改state为EMPTY
                let _ = self.state.compare_exchange(
                    DATA,
                    EMPTY,
                    Ordering::SeqCst,
                    Ordering::SeqCst,
                );
                //将数据读出
                match (&mut *self.data.get()).take() {
                    Some(data) => Ok(data),
                    None => unreachable!(),
                }
            }

            //中断状态时，可能此通道已经被升级，要检查是否还有数据
            DISCONNECTED => match (&mut *self.data.get()).take() {
                //有数据,读出数据即可
                Some(data) => Ok(data),
                //没有数据，更新upgrade状态
                None => match ptr::replace(self.upgrade.get(), SendUsed) {
                    //不是通知升级，则发送端已经关闭，返回Disconnected信息
                    SendUsed | NothingSent => Err(Disconnected),
                    //通知升级,将Receiver<T>包装到返回变量返回
                    GoUp(upgrade) => Err(Upgraded(upgrade)),
                },
            },

            // 不可能的分支
```

```
                _ => unreachable!(),
            }
        }
    }

    // 升级管道到其他类型，由发送线程调用
    pub fn upgrade(&self, up: Receiver<T>) -> UpgradeResult {
        unsafe {
            let prev = match *self.upgrade.get() {
                //可正常升级
                NothingSent => NothingSent,
                SendUsed => SendUsed,
                //其他状态表示已经升级完成
                _ => panic!("upgrading again"),
            };
            // 将升级到的Receiver写入self.upgrade
            ptr::write(self.upgrade.get(), GoUp(up));

            //后继不会再使用self传递消息，更新状态为DISCONNECTED
            match self.state.swap(DISCONNECTED, Ordering::SeqCst) {
                // 原状态为DATA及EMPTY，返回升级成功
                // 此时有可能消息还没有被接收
                // 返回后，发送端端口Sender会生命周期终结
                DATA | EMPTY => UpSuccess,

                //  如果已经DISCONNECT，则需要撤回本次请求
                DISCONNECTED => {
                    ptr::replace(self.upgrade.get(), prev);
                    // 升级时通道已经中断
                    UpDisconnected
                }

                // 如果有线程在等待接收， 需要将唤醒信号返回
                ptr => UpWoke(SignalToken::from_raw(ptr)),
            }
        }
    }

    //删除通道，由发送线程在Sender被drop时调用
    pub fn drop_chan(&self) {
        //更新状态
        match self.state.swap(DISCONNECTED, Ordering::SeqCst) {
            //原状态为下面的值可以不做操作
            DATA | DISCONNECTED | EMPTY => {}

            // 如果有等待线程，则发送信号唤醒
            ptr => unsafe {
                SignalToken::from_raw(ptr).signal();
            },
        }
    }
```

```
    //删除端口,由接收线程在Receiver被drop时调用
    pub fn drop_port(&self) {
        //更新状态
        match self.state.swap(DISCONNECTED, Ordering::SeqCst) {
            DISCONNECTED | EMPTY => {}

            // 如果有数据，需要删除它
            DATA => unsafe {
                (&mut *self.data.get()).take().unwrap();
                //数据包生命周期终止
            },

            // 接收线程才能调用这个函数
            _ => unreachable!(),
        }
    }

    // 阻塞超时处理.
    pub fn abort_selection(&self) -> Result<bool, Receiver<T>> {
        //获取state
        let state = match self.state.load(Ordering::SeqCst) {
            // 这些状态不用处理
            s @ (EMPTY | DATA | DISCONNECTED) => s,

            // ptr是本线程设置的，切换回EMPTY状态，并把信号指针带回
            ptr => self
                .state
                .compare_exchange(ptr, EMPTY, Ordering::SeqCst,
Ordering::SeqCst)
                .unwrap_or_else(|x| x),
        };

        match state {
            //不应该出现这个情况
            EMPTY => unreachable!(),
            //有数据
            DATA => Ok(true),

            // 发送端中断
            DISCONNECTED => unsafe {
                //收到数据
                if (*self.data.get()).is_some() {
                    Ok(true)
                } else {
                    //看是否需要升级
                    match ptr::replace(self.upgrade.get(), SendUsed) {
                        //升级调用，返回升级到的端口Reciver<T>
                        GoUp(port) => Err(port),
                        _ => Ok(true),
                    }
```

```
                }
            },

            // 没有其他线程发送数据
            ptr => unsafe {
                //删除信号
                drop(SignalToken::from_raw(ptr));
                //没有接收数据
                Ok(false)
            },
        }
    }
}

impl<T> Drop for Packet<T> {
    fn drop(&mut self) {
        assert_eq!(self.state.load(Ordering::SeqCst), DISCONNECTED);
    }
}
```

Shared的通道

当oneshot的tx做clone操作时，oneshot的通道升级到Shared类型通道:

```
//用发送包的技术来表示通道的状态
//通道中断计数标志
const DISCONNECTED: isize = isize::MIN;
//最大能支持的通道数
const FUDGE: isize = 1024;
const MAX_REFCOUNT: usize = (isize::MAX) as usize;
//最多能计数的无阻塞收包数目
const MAX_STEALS: isize = 1 << 20;
const EMPTY: *mut u8 = ptr::null_mut(); // initial state: no data, no blocked
receiver

pub struct Packet<T> {
    //消息包的queue
    queue: mpsc::Queue<T>,
    //发送的包总数,每次阻塞或接收包数目到达限值会设置为-1。
    // -1作为有阻塞，需要发送信号的标记
    cnt: AtomicIsize,
    //接收的包总数,每次阻塞，或接收包数目达到限值会清零
    steals: UnsafeCell<isize>,
    //唤醒的信号SingleToken指针

    //接收线程阻塞时期待的信号量
    to_wake: AtomicPtr<u8>,
```

```
    //初始最少有两个使用者,每多一个发送线程就加1
    channels: AtomicUsize,

    //接收端关闭通道的标志
    port_dropped: AtomicBool,
    //发送端发现接收端中断，确定清理线程的辅助结构
    sender_drain: AtomicIsize,

    //使用单元类型的Mutex，将Mutex仅做锁的场景，不包含临界区,通常这个锁的临界区是一段
代码操作
    select_lock: Mutex<()>,
}

pub enum Failure {
    Empty,
    Disconnected,
}

enum StartResult {
    Installed,
    Abort,
}

impl<T> Packet<T> {
    //新建一个通道，随后必须紧跟postinit_lock及inherit_blocker后才能做其他
    //通道操作
    pub fn new() -> Packet<T> {
        Packet {
            //包队列
            queue: mpsc::Queue::new(),
            //发送的包总数,每次阻塞或接收包数目到达限值会清零。
            cnt: AtomicIsize::new(0),
            //接收的包总数,每次阻塞，或接收包数目达到限值会清零
            steals: UnsafeCell::new(0),
            //唤醒接收线程的信号
            to_wake: AtomicPtr::new(EMPTY),
            //初始最少有两个使用者,每多一个发送线程就加1
            channels: AtomicUsize::new(2),
            //接收端关闭通道的标志
            port_dropped: AtomicBool::new(false),
            //发送端发现接收端中断，确定清理线程的辅助结构
            sender_drain: AtomicIsize::new(0),
            //用于创建时的临界区代码保户
            select_lock: Mutex::new(()),
        }
    }

    // 必须在new之后第一时间调用，在封装self的Arc还没有clone之前
    pub fn postinit_lock(&self) -> MutexGuard<'_, ()> {
        self.select_lock.lock().unwrap()
    }
```

```
    // 这个函数处理升级前的通道遗留的阻塞线程场景,guard是调用postinit_lock的返回
    pub fn inherit_blocker(&self, token: Option<SignalToken>, guard:
MutexGuard<'_, ()>) {
        //判断是否有接收线程阻塞
        if let Some(token) = token {
            assert_eq!(self.cnt.load(Ordering::SeqCst), 0);
            assert_eq!(self.to_wake.load(Ordering::SeqCst), EMPTY);
            //将阻塞信号设置到to_wake中
            self.to_wake.store(unsafe { token.to_raw() }, Ordering::SeqCst);
            //有接收线程阻塞，导致发第一个包的时候，才会去唤醒接收线程，接收线程才可能
            //做升级，然后才能接收数据包。这个-1作为阻塞的标志
            //这个设计方式过于复杂，不是一个好的设计，
            self.cnt.store(-1, Ordering::SeqCst);

            unsafe {
                // cnt为-1，steals也需要设置为-1
                *self.steals.get() = -1;
            }
        }

        //解锁
        drop(guard);
    }

    pub fn send(&self, t: T) -> Result<(), T> {
        //看接收端口Receiver是否已经关闭
        if self.port_dropped.load(Ordering::SeqCst) {
            return Err(t);
        }

        //判断通道是否中断,因为每个线程发送都可能会造成计数加1，所以最大值
        //是DISCONNECTED+FUDGE,这个区间可认为通道已经被设置为中断
        if self.cnt.load(Ordering::SeqCst) < DISCONNECTED + FUDGE {
            return Err(t);
        }

        //消息入队列
        self.queue.push(t);
        //增加队列计数,每次push队列都要先对cnt增加值来反映此操作
        //但此时此时接收端口Receiver可能生命周期终止，导致cnt被设置为DISCONNECT
        match self.cnt.fetch_add(1, Ordering::SeqCst) {
            //原值为-1，是发送的第一个包，且接收端在等待信号
            //其他线程不会得到-1，只有-1的发送线程来发送信号
            -1 => {
                //发信号通知接收线程退出阻塞,工作结束,
                //这个机制搞的有些复杂
                self.take_to_wake().signal();
            }

            // 消息入队列后，通道被中断，此时需要把数据包撤回.
```

```
            n if n < DISCONNECTED + FUDGE => {
                //重新设置cnt为中断状态
                self.cnt.store(DISCONNECTED, Ordering::SeqCst);

                //判断我们是否是第一个sender_drain
                if self.sender_drain.fetch_add(1, Ordering::SeqCst) == 0 {
                    //是，负责删除队列里面的所有消息包
                    loop {
                        //循环直到queue为空
                        loop {
                            match self.queue.pop() {
                                mpsc::Data(..) => {}
                                mpsc::Empty => break,
                                mpsc::Inconsistent => thread::yield_now(),
                            }
                        }

                        if self.sender_drain.fetch_sub(1, Ordering::SeqCst) ==
1 {
                            //确定所有线程都已经被处理
                            break;
                        }
                        //还有其他线程做了sender_drain的add，那再循环
                    }

                    // 本线程push到queue的包确定已经删除
                }
            }

            _ => {}
        }

        Ok(())
    }

    pub fn recv(&self, deadline: Option<Instant>) -> Result<T, Failure> {
        //尽量不阻塞
        match self.try_recv() {
            Err(Empty) => {}
            data => return data,
        }

        //需要阻塞
        //生成通知信号
        let (wait_token, signal_token) = blocking::tokens();
        //因为try_recv到此处可能会有其他线程发包，需要做些
        //处理看是否需要阻塞
        if self.decrement(signal_token) == Installed {
            //确定要阻塞
            if let Some(deadline) = deadline {
                //有超时要去,做一个超时等待
```

```
                let timed_out = !wait_token.wait_max_until(deadline);
                if timed_out {
                    //如果超时，需要做清理工作
                    self.abort_selection(false);
                }
            } else {
                //阻塞至包来到
                wait_token.wait();
            }
        }

        //当前已经有数据包
        match self.try_recv() {
            data @ Ok(..) => unsafe {
                //反应阻塞收包统计,try_recv会加1,这里减掉
                //有可能没有阻塞，但按照阻塞来计算
                //这里是为了对冲在阻塞时对cnt多减1
                //无论如何，利用这个来实现对阻塞与否的判断我认为不是一个好主意
                *self.steals.get() -= 1;
                data
            },
            data => data,
        }
    }

    //判断是否应该阻塞
    fn decrement(&self, token: SignalToken) -> StartResult {
        unsafe {
            assert_eq!(
                self.to_wake.load(Ordering::SeqCst),
                EMPTY,
                "This is a known bug in the Rust standard library. See
https://github.com/rust-lang/rust/issues/39364"
            );
            // 设置收线程阻塞信号到通道
            let ptr = token.to_raw();
            self.to_wake.store(ptr, Ordering::SeqCst);

            //进入阻塞时对steals做清零
            let steals = ptr::replace(self.steals.get(), 0);

            //cnt需要把上次阻塞到本次阻塞之间的收包数目减掉，然后再减1,以便cnt成为-1
            match self.cnt.fetch_sub(1 + steals, Ordering::SeqCst) {
                //如果减法之前发送侧已经中断
                DISCONNECTED => {
                    //将cnt恢复为中断
                    self.cnt.store(DISCONNECTED, Ordering::SeqCst);
                }

                //不是中断，原来至少应该发送过一个包，cnt应该不小于0
                n => {
```

```
                    assert!(n >= 0);
                    //在两次取值间可能有其他通道已经发包过来，那不应该阻塞
                    //如果没有其他包，则阻塞
                    //发送的包减掉接收的包不大于0，表示没有线程竞争发包
                    if n - steals <= 0 {
                        //正常阻塞
                        return Installed;
                    }
                }
            }

            //此时队列已经有包或者DISCONNECT了，不需要阻塞
            //撤掉信号
            self.to_wake.store(EMPTY, Ordering::SeqCst);
            drop(SignalToken::from_raw(ptr));
            Abort
        }
    }

    //接收数据包
    pub fn try_recv(&self) -> Result<T, Failure> {
        //从队列取得一个包
        let ret = match self.queue.pop() {
            //成功
            mpsc::Data(t) => Some(t),
            //不成功
            mpsc::Empty => None,

            // 此时处于一个临界状态.可以做个自旋等待一下
            mpsc::Inconsistent => {
                let data;
                //默认为肯定会获得数据
                loop {
                    //这里等待一个操作系统调度周期
                    //试图让发送线程工作
                    //但等待时间不定
                    thread::yield_now();
                    match self.queue.pop() {
                        //收到数据
                        mpsc::Data(t) => {
                            data = t;
                            break;
                        }
                        //不应有这种情况
                        mpsc::Empty => panic!("inconsistent => empty"),
                        //继续等待
                        mpsc::Inconsistent => {}
                    }
                }
                Some(data)
            }
```

```
        };
        match ret {
            //接收到数据
            Some(data) => unsafe {
                //如果非阻塞收包已经大于MAX_STEALS
                if *self.steals.get() > MAX_STEALS {
                    //将cnt清零
                    match self.cnt.swap(0, Ordering::SeqCst) {
                        //原cnt是DISCONNECTED
                        DISCONNECTED => {
                            //重新置为DISCONNECTED
                            self.cnt.store(DISCONNECTED, Ordering::SeqCst);
                        }
                        n => {
                            //这里在cnt及steals上共同减去两者之间小者
                            //取值小者
                            let m = cmp::min(n, *self.steals.get());
                            //steals及cnt都减去最小值
                            *self.steals.get() -= m;
                            //实际上是原cnt减去m
                            self.bump(n - m);
                        }
                    }
                    assert!(*self.steals.get() >= 0);
                }
                //steals增加
                *self.steals.get() += 1;
                Ok(data)
            },

            //没有收到数据
            None => {
                match self.cnt.load(Ordering::SeqCst) {
                    //如果通道没有中断，返回异常的队列空
                    n if n != DISCONNECTED => Err(Empty),
                    //其他 只可能是DISCONNECTED
                    _ => {
                        //再接收一次
                        match self.queue.pop() {
                            //没有对self.steals做操作
                            mpsc::Data(t) => Ok(t),
                            //空，认为已经中断
                            mpsc::Empty => Err(Disconnected),
                            // 不应有这种情况
                            mpsc::Inconsistent => unreachable!(),
                        }
                        //丢弃这个包，所以不必更新计数
                    }
                }
            }
        }
```

```
    }

    // Sender<T>做clone时的支撑函数
    pub fn clone_chan(&self) {
        //channel数目增加
        let old_count = self.channels.fetch_add(1, Ordering::SeqCst);

        if old_count > MAX_REFCOUNT {
            abort();
        }
    }

    // 发送线程关闭通道
    pub fn drop_chan(&self) {
        //减少channel计数
        match self.channels.fetch_sub(1, Ordering::SeqCst) {
            //需要做清理
            1 => {}
            //还有其他发送线程，不必处理
            n if n > 1 => return,
            //不应该发生这种情况
            n => panic!("bad number of channels left {n}"),
        }

        //所有发送线程均已关闭，发端置中断状态
        match self.cnt.swap(DISCONNECTED, Ordering::SeqCst) {
            // 有接收线程阻塞
            -1 => {
                //发信号解除阻塞
                self.take_to_wake().signal();
            }
            DISCONNECTED => {}
            n => {
                assert!(n >= 0);
            }
        }
    }

    //接收线程关闭通道
    pub fn drop_port(&self) {
        //置标志
        self.port_dropped.store(true, Ordering::SeqCst);
        //获取上次阻塞以来接收的数据包
        let mut steals = unsafe { *self.steals.get() };
        while {
            //这个block是while的条件语句
            //当发送数据包与接收数据包相同时，设置中断
            match self.cnt.compare_exchange(
                steals,
                DISCONNECTED,
                Ordering::SeqCst,
```

```
                Ordering::SeqCst,
            ) {
                //成功,退出循环
                Ok(_) => false,
                //old是DISCONNECT时，退出循环，否则进入循环
                Err(old) => old != DISCONNECTED,
            }
        } {
            //这个循环把队列清空
            loop {
                //收包
                match self.queue.pop() {
                    mpsc::Data(..) => {
                        steals += 1;
                    }
                    mpsc::Empty | mpsc::Inconsistent => break,
                }
                //生命周期终结，释放包
            }
        }
    }

    // 重组阻塞信号结构
    fn take_to_wake(&self) -> SignalToken {
        let ptr = self.to_wake.load(Ordering::SeqCst);
        self.to_wake.store(EMPTY, Ordering::SeqCst);
        assert!(ptr != EMPTY);
        unsafe { SignalToken::from_raw(ptr) }
    }

    //一次性给cnt增加若干值
    fn bump(&self, amt: isize) -> isize {
        //一次增加cnt输入参数
        match self.cnt.fetch_add(amt, Ordering::SeqCst) {
            //如果原值是DISCONNECT
            DISCONNECTED => {
                //cnt恢复为DISCONNECT
                self.cnt.store(DISCONNECTED, Ordering::SeqCst);
                DISCONNECTED
            }
            n => n,
        }
    }

    //接收线程阻塞超时时做处理
    pub fn abort_selection(&self, _was_upgrade: bool) -> bool {
        //加锁，保护下面的临界区代码
        {
            let _guard = self.select_lock.lock().unwrap();
        }
```

```
        let steals = {
            //这个程序员愿意用block作为表达式结果
            //获取cnt
            let cnt = self.cnt.load(Ordering::SeqCst);
            //发送端没有中断，cnt应该是阻塞超时的次数
            //只能是-1或者0
            if cnt < 0 && cnt != DISCONNECTED { -cnt } else { 0 }
        };
        //cnt增加，清除超时,每次超时如果有包，则steals加1
        let prev = self.bump(steals + 1);

        //发送端已经中断
        if prev == DISCONNECTED {
            //更新等待信号为空,
            assert_eq!(self.to_wake.load(Ordering::SeqCst), EMPTY);
            //后继退出收包
            true
        } else {
            //当前的发包计数
            let cur = prev + steals + 1;
            assert!(cur >= 0);
            if prev < 0 {
                //没有发包导致，drop掉接收等待信号
                drop(self.take_to_wake());
            } else {
                //发送端马上应该发送信号，等一下
                while self.to_wake.load(Ordering::SeqCst) != EMPTY {
                    thread::yield_now();
                }
            }
            unsafe {
                let old = self.steals.get();
                //steals只可能是0或1
                assert!(*old == 0 || *old == -1);
                //更新self.steals,实际上是steals加1
                *old = steals;
                prev >= 0
            }
        }
    }
}

impl<T> Drop for Packet<T> {
    fn drop(&mut self) {
        //确保Packet已经清理完毕
        assert_eq!(self.cnt.load(Ordering::SeqCst), DISCONNECTED);
        assert_eq!(self.to_wake.load(Ordering::SeqCst), EMPTY);
        assert_eq!(self.channels.load(Ordering::SeqCst), 0);
    }
}
```

shared 类型的通道设计最奇怪的地方是用了复杂的发包计数来作为阻塞标记。导致该处代码不易理解。

mpsc的对外函数及接口

通道相关的类型结构及函数：

```
pub fn channel<T>() -> (Sender<T>, Receiver<T>) {
    //初始时创建oneshot的通道
    let a = Arc::new(oneshot::Packet::new());
    //对onshot通道做clone，然后创建Sender及Receiver
    (Sender::new(Flavor::Oneshot(a.clone())),
Receiver::new(Flavor::Oneshot(a)))
}

//发送端端口
pub struct Sender<T> {
    inner: UnsafeCell<Flavor<T>>,
}

//接收端端口
pub struct Receiver<T> {
    inner: UnsafeCell<Flavor<T>>,
}

//用来实现可升级的通道，因为有带参数的成员，RUST没有使用dyn trait这种设计
//对于认为以后通道类型不会再扩张时，采用enum的设计方式更易控制
//但如果预计后继还会有很多通道方式，则应该采用dyn Packet<T>的设计方式
enum Flavor<T> {
    //只发送单一通信包的通道
    Oneshot(Arc<oneshot::Packet<T>>),
    //一对一的多通信包的通道,当发端发送第二个包的时候
    //要创建并切换到这个通道
    Stream(Arc<stream::Packet<T>>),
    //多对一的通道，当发端做clone操作的时候
    //要创建并切换到这个通道
    Shared(Arc<shared::Packet<T>>),
    //同步通道，本书不分析
    Sync(Arc<sync::Packet<T>>),
}

//Sender及Receiver内部访问支持trait
trait UnsafeFlavor<T> {
    fn inner_unsafe(&self) -> &UnsafeCell<Flavor<T>>;
    unsafe fn inner_mut(&self) -> &mut Flavor<T> {
        &mut *self.inner_unsafe().get()
    }
    unsafe fn inner(&self) -> &Flavor<T> {
        &*self.inner_unsafe().get()
    }
```

```
}
impl<T> UnsafeFlavor<T> for Sender<T> {
    fn inner_unsafe(&self) -> &UnsafeCell<Flavor<T>> {
        &self.inner
    }
}
impl<T> UnsafeFlavor<T> for Receiver<T> {
    fn inner_unsafe(&self) -> &UnsafeCell<Flavor<T>> {
        &self.inner
    }
}
```

Sender的方法：

```
impl<T> Sender<T> {
    //创建包含通道的Sender
    fn new(inner: Flavor<T>) -> Sender<T> {
        Sender { inner: UnsafeCell::new(inner) }
    }

    //发送一个数据包
    pub fn send(&self, t: T) -> Result<(), SendError<T>> {
        //相当于新创建了一个Flavor的变量，此时要注意drop是否发生了两次
        //这里对解引用的match因为没有引发赋值，不会导致所有权转移
        let (new_inner, ret) = match *unsafe { self.inner() } {
            //必须是ref，否则会导致所有权转移
            Flavor::Oneshot(ref p) => {
                //判断是否还能发送包，此时只能发一个包
                if !p.sent() {
                    return p.send(t).map_err(SendError);
                } else {
                    //多于一个包，创建stream的通道来进行升级
                    let a = Arc::new(stream::Packet::new());
                    //基于新的通道创建新的Receiver
                    let rx = Receiver::new(Flavor::Stream(a.clone()));
                    //通知rx端进行升级操作
                    match p.upgrade(rx) {
                        //升级成功
                        oneshot::UpSuccess => {
                            //发送报文
                            let ret = a.send(t);
                            //将新的通道赋值
                            (a, ret)
                        }
                        //接收已经DISCONNECT，将数据包及新通道共同返回
                        oneshot::UpDisconnected => (a, Err(t)),
                        //接收线程阻塞,需要做唤醒
                        oneshot::UpWoke(token) => {
                            //先将包发送
                            a.send(t).ok().unwrap();
```

```
                            //唤醒接收线程
                            token.signal();
                            //返回新通道
                            (a, Ok(()))
                        }
                    }
                }
            }
            //已经是Stream，正常发送包的逻辑，直接返回，不修改self
            Flavor::Stream(ref p) => return p.send(t).map_err(SendError),
            //已经是Shared，正常的发送逻辑，直接返回，不修改self
            Flavor::Shared(ref p) => return p.send(t).map_err(SendError),
            //不可能到达这个代码位置
            Flavor::Sync(..) => unreachable!(),
        };

        unsafe {
            //只有oneshot会进入此处
            //新建Sender，并将新的Sender及老的Sender进行内存替换
            //此处要注意，enum的不同成员不保证内存相同，但这里是没有问题的
            let tmp = Sender::new(Flavor::Stream(new_inner));
            mem::swap(self.inner_mut(), tmp.inner_mut());
            //此处,tmp会生命周期终结，tmp当前是oneshot的类型。
            //要注意收端是怎么终结的
        }
        ret.map_err(SendError)
    }
}

impl<T> Clone for Sender<T> {
    /// clone代表进入了多发一收的模式，需要升级到shared类型的通道
    fn clone(&self) -> Sender<T> {
        let packet = match *unsafe { self.inner() } {
            Flavor::Oneshot(ref p) => {
                //创建shared类型通道
                let a = Arc::new(shared::Packet::new());
                {
                    //创建后首先lock
                    let guard = a.postinit_lock();
                    //创建Receiver
                    let rx = Receiver::new(Flavor::Shared(a.clone()));
                    //进行升级
                    let sleeper = match p.upgrade(rx) {
                        oneshot::UpSuccess | oneshot::UpDisconnected => None,
                        oneshot::UpWoke(task) => Some(task),
                    };
                    //完成通道设置
                    a.inherit_blocker(sleeper, guard);
                }
                //置值
                a
```

```
            }
            //进入一对一的多包发送
            Flavor::Stream(ref p) => {
                //仍然创建shared类型通道
                let a = Arc::new(shared::Packet::new());
                {
                    //首先lock
                    let guard = a.postinit_lock();
                    //创建Receiver
                    let rx = Receiver::new(Flavor::Shared(a.clone()));
                    //升级
                    let sleeper = match p.upgrade(rx) {
                        stream::UpSuccess | stream::UpDisconnected => None,
                        stream::UpWoke(task) => Some(task),
                    };
                    //完成通道设置
                    a.inherit_blocker(sleeper, guard);
                }
                //置值
                a
            }
            Flavor::Shared(ref p) => {
                //先做clone_chan
                p.clone_chan();
                //创建新的Sender,并返回
                return Sender::new(Flavor::Shared(p.clone()));
            }
            //不会到达这个地方
            Flavor::Sync(..) => unreachable!(),
        };

        unsafe {
            //创建新的Sender
            let tmp = Sender::new(Flavor::Shared(packet.clone()));
            //替换现有的Sender
            mem::swap(self.inner_mut(), tmp.inner_mut());
            //原有的Flavor生命周期终止并被drop
        }
        //创建新的Sender,并返回
        Sender::new(Flavor::Shared(packet))
    }
}

impl<T> Drop for Sender<T> {
    fn drop(&mut self) {
        //行为一致，都是中断通道
        match *unsafe { self.inner() } {
            Flavor::Oneshot(ref p) => p.drop_chan(),
            Flavor::Stream(ref p) => p.drop_chan(),
            Flavor::Shared(ref p) => p.drop_chan(),
            Flavor::Sync(..) => unreachable!(),
```

```
        }
    }
}
```

Receiver的方法：

```
impl<T> Receiver<T> {
    fn new(inner: Flavor<T>) -> Receiver<T> {
        Receiver { inner: UnsafeCell::new(inner) }
    }

    //不阻塞的收包
    pub fn try_recv(&self) -> Result<T, TryRecvError> {
        loop {
            let new_port = match *unsafe { self.inner() } {
                Flavor::Oneshot(ref p) => match p.try_recv() {
                    //非升级的情况都直接返回
                    Ok(t) => return Ok(t),
                    Err(oneshot::Empty) => return Err(TryRecvError::Empty),
                    Err(oneshot::Disconnected) => return
Err(TryRecvError::Disconnected),
                    //升级的情况将rx置值到new_port
                    Err(oneshot::Upgraded(rx)) => rx,
                },
                Flavor::Stream(ref p) => match p.try_recv() {
                    //非升级的情况都直接返回
                    Ok(t) => return Ok(t),
                    Err(stream::Empty) => return Err(TryRecvError::Empty),
                    Err(stream::Disconnected) => return
Err(TryRecvError::Disconnected),
                    //升级的情况将rx置值到new_port
                    Err(stream::Upgraded(rx)) => rx,
                },
                Flavor::Shared(ref p) => match p.try_recv() {
                    Ok(t) => return Ok(t),
                    Err(shared::Empty) => return Err(TryRecvError::Empty),
                    Err(shared::Disconnected) => return
Err(TryRecvError::Disconnected),
                    //不应该出现升级的情况
                },
                Flavor::Sync(ref p) => match p.try_recv() {
                    Ok(t) => return Ok(t),
                    Err(sync::Empty) => return Err(TryRecvError::Empty),
                    Err(sync::Disconnected) => return
Err(TryRecvError::Disconnected),
                },
            };
            unsafe {
                //直接用new_port替换原来的Flavor
                mem::swap(self.inner_mut(), new_port.inner_mut());
```

```
            }
            //new_port生命周期终结，原有的Flavor被调用drop
        }
    }

    //阻塞收包，替换逻辑与try_recv相同
    pub fn recv(&self) -> Result<T, RecvError> {
        loop {
            let new_port = match *unsafe { self.inner() } {
                Flavor::Oneshot(ref p) => match p.recv(None) {
                    Ok(t) => return Ok(t),
                    Err(oneshot::Disconnected) => return Err(RecvError),
                    Err(oneshot::Upgraded(rx)) => rx,
                    Err(oneshot::Empty) => unreachable!(),
                },
                Flavor::Stream(ref p) => match p.recv(None) {
                    Ok(t) => return Ok(t),
                    Err(stream::Disconnected) => return Err(RecvError),
                    Err(stream::Upgraded(rx)) => rx,
                    Err(stream::Empty) => unreachable!(),
                },
                Flavor::Shared(ref p) => match p.recv(None) {
                    Ok(t) => return Ok(t),
                    Err(shared::Disconnected) => return Err(RecvError),
                    Err(shared::Empty) => unreachable!(),
                },
                Flavor::Sync(ref p) => return p.recv(None).map_err(|_|
RecvError),
            };
            unsafe {
                mem::swap(self.inner_mut(), new_port.inner_mut());
            }
        }
    }

    //设置超时的阻塞收包
    pub fn recv_timeout(&self, timeout: Duration) -> Result<T,
RecvTimeoutError> {
        // Do an optimistic try_recv to avoid the performance impact of
        // Instant::now() in the full-channel case.
        match self.try_recv() {
            Ok(result) => Ok(result),
            Err(TryRecvError::Disconnected) =>
Err(RecvTimeoutError::Disconnected),
            //没有包的时候才进入超时
            Err(TryRecvError::Empty) => match
Instant::now().checked_add(timeout) {
                //调用超时接收
                Some(deadline) => self.recv_deadline(deadline),
                None => self.recv().map_err(RecvTimeoutError::from),
            },
```

```
        }
    }

    //真正的超时接收,与recv基本相同，仅增加了超时参数
    pub fn recv_deadline(&self, deadline: Instant) -> Result<T,
RecvTimeoutError> {
        use self::RecvTimeoutError::*;

        loop {
            let port_or_empty = match *unsafe { self.inner() } {
                Flavor::Oneshot(ref p) => match p.recv(Some(deadline)) {
                    Ok(t) => return Ok(t),
                    Err(oneshot::Disconnected) => return Err(Disconnected),
                    Err(oneshot::Upgraded(rx)) => Some(rx),
                    Err(oneshot::Empty) => None,
                },
                Flavor::Stream(ref p) => match p.recv(Some(deadline)) {
                    Ok(t) => return Ok(t),
                    Err(stream::Disconnected) => return Err(Disconnected),
                    Err(stream::Upgraded(rx)) => Some(rx),
                    Err(stream::Empty) => None,
                },
                Flavor::Shared(ref p) => match p.recv(Some(deadline)) {
                    Ok(t) => return Ok(t),
                    Err(shared::Disconnected) => return Err(Disconnected),
                    Err(shared::Empty) => None,
                },
                Flavor::Sync(ref p) => match p.recv(Some(deadline)) {
                    Ok(t) => return Ok(t),
                    Err(sync::Disconnected) => return Err(Disconnected),
                    Err(sync::Empty) => None,
                },
            };

            if let Some(new_port) = port_or_empty {
                unsafe {
                    mem::swap(self.inner_mut(), new_port.inner_mut());
                }
            }

            // If we're already passed the deadline, and we're here without
            // data, return a timeout, else try again.
            if Instant::now() >= deadline {
                return Err(Timeout);
            }
        }
    }

    //函数式编程，用iterator来简化rx的动作
    pub fn iter(&self) -> Iter<'_, T> {
        Iter { rx: self }
```

```
    }

    //不会阻塞的iterator
    pub fn try_iter(&self) -> TryIter<'_, T> {
        TryIter { rx: self }
    }
}
```

针对Receiver的迭代器举例：

```
//只是为了函数式编程及利用Iterator的基础设施
pub struct Iter<'a, T: 'a> {
    rx: &'a Receiver<T>,
}

impl<'a, T> Iterator for Iter<'a, T> {
    type Item = T;

    fn next(&mut self) -> Option<T> {
        self.rx.recv().ok()
    }
}
```