RUST的临界区变量实现

代码路径: library/std/src/sync/*.rs

`Mutex<T>` 的实现

`Mutex<T>` 是最典型的临界区变量。RUST的设计目标是在使用 `Mutex<T>` 时代码中不必关注其的跨线程操作的安全性，使用方式与内部可变性类型的使用相类似。这一设计思路实际在 `RefCell<T>`，`Rc<T>`，`Arc<T>` 等类型设计上是一脉相承的：

1. 设计一个基础类型结构，将要操作的真实类型变量封装在其内，并拥有其所有权，
2. 设计一个借用类型结构，由基础类型结构的某一方法生成，在此方法中完成附加安全操作，如计数增加，加锁等。
3. 借用类型结构实现解引用方法，返回真实类型变量的引用或可变引用，由此可以对真实变类型变量进行访问，修改操作
4. 借用类型结构的drop方法会执行安全逆操作，如减少计数或解锁。

`Mutex<T>` 的设计如下：

1. 基本类型 `Mutex<T>`，负责临界区的数据存储及Mutex锁
2. `MutexGuard<'a, T>` 作为 `Mutex<T>` 的借用类型结构, lock()作为借用方法，返回 `MutexGuard<T>`，可以直接对其解引用后获得内部变量的引用/可变引用，随后执行临界区数据操作及读写。生命周期结束后，`MutexGuard<T>` 的drop会解锁操作，从而使得加锁解锁操作实际上代码不必关心。lock()本身完全可以等同于一个borrow()的调用。
3. `Mutex<T>` 本身是一个内部可变型的类型, 实现多处共享且可修改
4. 线程panic时的Poison处理,使得其他语言极少关注的情况在RUST中自然得解。

```
pub struct Mutex<T: ?Sized> {
    //临界区的锁
    inner: sys::MovableMutex,
    //标识Mutex在线程panic时处于锁状态
    poison: poison::Flag,
    //临界区数据，Mutex本身是一个内部可变性的类型
    data: UnsafeCell<T>,
}

unsafe impl<T: ?Sized + Send> Send for Mutex<T> {}
unsafe impl<T: ?Sized + Send> Sync for Mutex<T> {}
```

用于 `Mutex<T>` 配合的借用封装类型结构 `MutexGuard` 如下：

```rust
//用于lock调用后的对原始变量的访问引用。并包含了poison用于在自身生命周期终结的时候
//更新Mutex<T>的Flag
pub struct MutexGuard<'a, T: ?Sized + 'a> {
    lock: &'a Mutex<T>,
    poison: poison::Guard,
}

//标识MutexGuard的当前线程panic状态
pub struct Guard {
    panicking: bool,
}

//支持函数，LockResult请参考14-线程间锁通信
pub fn map_result<T, U, F>(result: LockResult<T>, f: F) -> LockResult<U>
where
    F: FnOnce(T) -> U,
{
    match result {
        Ok(t) => Ok(f(t)),
        Err(PoisonError { guard }) => Err(PoisonError::new(f(guard))),
    }
}

//MutexGuard创建关联函数
impl<'mutex, T: ?Sized> MutexGuard<'mutex, T> {
    unsafe fn new(lock: &'mutex Mutex<T>) -> LockResult<MutexGuard<'mutex, T>>
{
        //如果Mutex<T>的poison为假，即使本线程已经panic，也返回Ok类型
        //因为不是在加锁时遇到panic，所以临界区数据一致性没有受到破坏。
        poison::map_result(lock.poison.borrow(), |guard| MutexGuard { lock,
poison: guard })
    }
}

//deref，返回临界区数据的引用
impl<T: ?Sized> Deref for MutexGuard<'_, T> {
    type Target = T;

    fn deref(&self) -> &T {
        //利用UnsafeCell获得内部可变性
        unsafe { &*self.lock.data.get() }
    }
}

//返回临界区数据的可变引用
impl<T: ?Sized> DerefMut for MutexGuard<'_, T> {
    fn deref_mut(&mut self) -> &mut T {
        unsafe { &mut *self.lock.data.get() }
    }
}
```

```rust
//drop方法
impl<T: ?Sized> Drop for MutexGuard<'_, T> {
    fn drop(&mut self) {
        unsafe {
            //更新Mutex<T>的Flag,一般的，如果在上锁的状态下线程panic
            //会导致对所有栈变量做drop调用，从而此drop被调用
            //self.lock.poison被更新为true
            self.lock.poison.done(&self.poison);
            //解锁
            self.lock.inner.raw_unlock();
        }
    }
}

//获取Mutex
pub fn guard_lock<'a, T: ?Sized>(guard: &MutexGuard<'a, T>) -> &'a
sys::MovableMutex {
    &guard.lock.inner
}

//获取线程panic状态
pub fn guard_poison<'a, T: ?Sized>(guard: &MutexGuard<'a, T>) -> &'a
poison::Flag {
    &guard.lock.poison
}
```

在`Mutex<T>`结构中，poison导致更新一般在发生panic时，线程drop`MutexGuard<T>`时进行标志更新。值得说明的时，上锁后线程异常退出是很少被考虑到的安全问题，RUST的标准库则给出了解决方案。

`Mutex<T>`的代码分析如下：

```rust
//只能创建固定尺寸类型的临界区
impl<T> Mutex<T> {
    //对数据创建一个临界区
    pub fn new(t: T) -> Mutex<T> {
        Mutex {
            //创建系统MovableMutex类型
            inner: sys::MovableMutex::new(),
            //poison为false
            poison: poison::Flag::new(),
            //必须用内部可变性类型
            data: UnsafeCell::new(t),
        }
    }
}
```

```rust
impl<T: ?Sized> Mutex<T> {
    // 获取锁，允许阻塞，返回guard结构用于访问临界区数据及处理锁的释放
    pub fn lock(&self) -> LockResult<MutexGuard<'_, T>> {
        unsafe {
            //先做锁操作
            self.inner.raw_lock();
            //在MutexGuard的new中处理线程panic问题
            MutexGuard::new(self)
        }
    }

    //试图获取锁，不会阻塞
    pub fn try_lock(&self) -> TryLockResult<MutexGuard<'_, T>> {
        unsafe {
            if self.inner.try_lock() {
                //上锁成功，生成MutexGuard
                Ok(MutexGuard::new(self)?)
            } else {
                //失败，提示应该阻塞
                Err(TryLockError::WouldBlock)
            }
        }
    }

    //立即解锁，不希望等待guard生命周期终结，
    pub fn unlock(guard: MutexGuard<'_, T>) {
        drop(guard);
    }

    //是否有线程在panic时锁住了临界区
    pub fn is_poisoned(&self) -> bool {
        self.poison.get()
    }

    //消费Mutex<T>,并获取临界区数据
    pub fn into_inner(self) -> LockResult<T>
    where
        T: Sized,
    {
        //获取临界区数据
        let data = self.data.into_inner();
        //根据是否有线程在panic时加锁，
        poison::map_result(self.poison.borrow(), |_| data)
    }

    //获取临界区数据的可变引用，如果已经执行过lock，
    //此处会编译失败
    pub fn get_mut(&mut self) -> LockResult<&mut T> {
        let data = self.data.get_mut();
        poison::map_result(self.poison.borrow(), |_| data)
    }
```

```
}
```

RUST的`Mutex<T>`再一次揭示了RUST标准库对编程员简化编程的努力及一些标准思维及技巧。

`Condvar`实现分析

RUST的Condvar是与`MutexGuard<'a, T>`临界区变量相关联在一起。RUST的Condvar摆脱了其他语言的那些复杂概念与代码形式，使用方式逻辑上非常顺理成章。

```
//仅仅是对操作系统的Condvar的一个封装
pub struct Condvar {
    inner: sys::Condvar,
}

impl Condvar {

    pub fn new() -> Condvar {
        Condvar { inner: sys::Condvar::new() }
    }

    // 等待信号通知，并可能进入阻塞,因为用MutexGuard，解脱了与Mutex相互关联的复杂概念，
    // 且Mutex一定已经lock，且临界区数据包括在内，使得Condvar更易被理解及使用
    pub fn wait<'a, T>(&self, guard: MutexGuard<'a, T>) ->
LockResult<MutexGuard<'a, T>> {
        let poisoned = unsafe {
            //获取关联的imp::Mutex
            let lock = mutex::guard_lock(&guard);
            self.inner.wait(lock);
            //获取Mutex的poison
            mutex::guard_poison(&guard).get()
        };
        if poisoned { Err(PoisonError::new(guard)) } else { Ok(guard) }
    }

    //函数式编程，将对临界区的操作封装在闭包中
    pub fn wait_while<'a, T, F>(
        &self,
        mut guard: MutexGuard<'a, T>,
        mut condition: F,
    ) -> LockResult<MutexGuard<'a, T>>
    where
        F: FnMut(&mut T) -> bool,
    {
        while condition(&mut *guard) {
            guard = self.wait(guard)?;
        }
```

```
        Ok(guard)
    }

    // 简化以毫秒计数的超时等待
    pub fn wait_timeout_ms<'a, T>(
        &self,
        guard: MutexGuard<'a, T>,
        ms: u32,
    ) -> LockResult<(MutexGuard<'a, T>, bool)> {
        let res = self.wait_timeout(guard, Duration::from_millis(ms as u64));
        poison::map_result(res, |(a, b)| (a, !b.timed_out()))
    }

    // 超时等待
    pub fn wait_timeout<'a, T>(
        &self,
        guard: MutexGuard<'a, T>,
        dur: Duration,
    ) -> LockResult<(MutexGuard<'a, T>, WaitTimeoutResult)> {
        let (poisoned, result) = unsafe {
            let lock = mutex::guard_lock(&guard);
            let success = self.inner.wait_timeout(lock, dur);
            (mutex::guard_poison(&guard).get(), WaitTimeoutResult(!success))
        };
        if poisoned { Err(PoisonError::new((guard, result))) } else {
Ok((guard, result)) }
    }

    //wait_while的超时版本
    pub fn wait_timeout_while<'a, T, F>(
        &self,
        mut guard: MutexGuard<'a, T>,
        dur: Duration,
        mut condition: F,
    ) -> LockResult<(MutexGuard<'a, T>, WaitTimeoutResult)>
    where
        F: FnMut(&mut T) -> bool,
    {
        let start = Instant::now();
        loop {
            if !condition(&mut *guard) {
                return Ok((guard, WaitTimeoutResult(false)));
            }
            let timeout = match dur.checked_sub(start.elapsed()) {
                Some(timeout) => timeout,
                None => return Ok((guard, WaitTimeoutResult(true))),
            };
            guard = self.wait_timeout(guard, timeout)?.0;
        }
    }
```

```
    //信号通知，唤醒一个线程
    pub fn notify_one(&self) {
        self.inner.notify_one()
    }

    //信号通知，唤醒所有线程
    pub fn notify_all(&self) {
        self.inner.notify_all()
    }
}
```

`RWLock<T>` 分析

与 `Mutex<T>` 的设计采用了一致的设计思路：

代码分析如下：

```
//与Mutex<T>几乎同样的成员
pub struct RwLock<T: ?Sized> {
    inner: sys::MovableRwLock,
    poison: poison::Flag,
    data: UnsafeCell<T>,
}

unsafe impl<T: ?Sized + Send> Send for RwLock<T> {}
unsafe impl<T: ?Sized + Send + Sync> Sync for RwLock<T> {}

//用read锁后的借用封装类型结构
pub struct RwLockReadGuard<'a, T: ?Sized + 'a> {
    lock: &'a RwLock<T>,
}

impl<T: ?Sized> !Send for RwLockReadGuard<'_, T> {}

unsafe impl<T: ?Sized + Sync> Sync for RwLockReadGuard<'_, T> {}

//用write锁后的借用封装类型结构
pub struct RwLockWriteGuard<'a, T: ?Sized + 'a> {
    lock: &'a RwLock<T>,
    poison: poison::Guard,
}

impl<T: ?Sized> !Send for RwLockWriteGuard<'_, T> {}

unsafe impl<T: ?Sized + Sync> Sync for RwLockWriteGuard<'_, T> {}

impl<T> RwLock<T> {
    pub fn new(t: T) -> RwLock<T> {
```

```
        RwLock {
            inner: sys::MovableRwLock::new(),
            poison: poison::Flag::new(),
            data: UnsafeCell::new(t),
        }
    }
}

impl<T: ?Sized> RwLock<T> {
    //读上锁，返回读锁的临界区借用封装
    pub fn read(&self) -> LockResult<RwLockReadGuard<'_, T>> {
        unsafe {
            self.inner.read();
            RwLockReadGuard::new(self)
        }
    }

    //不希望阻塞时做调用
    pub fn try_read(&self) -> TryLockResult<RwLockReadGuard<'_, T>> {
        unsafe {
            if self.inner.try_read() {
                Ok(RwLockReadGuard::new(self)?)
            } else {
                Err(TryLockError::WouldBlock)
            }
        }
    }

    //写锁，返回一个写锁的借用封装
    pub fn write(&self) -> LockResult<RwLockWriteGuard<'_, T>> {
        unsafe {
            self.inner.write();
            RwLockWriteGuard::new(self)
        }
    }

    //不希望阻塞时的写锁调用
    pub fn try_write(&self) -> TryLockResult<RwLockWriteGuard<'_, T>> {
        unsafe {
            if self.inner.try_write() {
                Ok(RwLockWriteGuard::new(self)?)
            } else {
                Err(TryLockError::WouldBlock)
            }
        }
    }

    //是否中毒
    pub fn is_poisoned(&self) -> bool {
        self.poison.get()
    }
```

```rust
    //消费掉锁，此时如果有读锁或写锁，编译器会告警
    pub fn into_inner(self) -> LockResult<T>
    where
        T: Sized,
    {
        let data = self.data.into_inner();
        poison::map_result(self.poison.borrow(), |_| data)
    }

    //此时如果有读锁或写锁，编译器会告警,针对self
    pub fn get_mut(&mut self) -> LockResult<&mut T> {
        let data = self.data.get_mut();
        poison::map_result(self.poison.borrow(), |_| data)
    }
}

// 读锁的借用封装结构
impl<'rwlock, T: ?Sized> RwLockReadGuard<'rwlock, T> {
    unsafe fn new(lock: &'rwlock RwLock<T>) ->
LockResult<RwLockReadGuard<'rwlock, T>> {
        poison::map_result(lock.poison.borrow(), |_| RwLockReadGuard { lock })
    }
}

// 写锁的借用封装结构
impl<'rwlock, T: ?Sized> RwLockWriteGuard<'rwlock, T> {
    unsafe fn new(lock: &'rwlock RwLock<T>) ->
LockResult<RwLockWriteGuard<'rwlock, T>> {
        poison::map_result(lock.poison.borrow(), |guard| RwLockWriteGuard {
lock, poison: guard })
    }
}

impl<T: ?Sized> Deref for RwLockReadGuard<'_, T> {
    type Target = T;

    fn deref(&self) -> &T {
        unsafe { &*self.lock.data.get() }
    }
}

impl<T: ?Sized> Deref for RwLockWriteGuard<'_, T> {
    type Target = T;

    fn deref(&self) -> &T {
        unsafe { &*self.lock.data.get() }
    }
}

impl<T: ?Sized> DerefMut for RwLockWriteGuard<'_, T> {
```

```
    fn deref_mut(&mut self) -> &mut T {
        unsafe { &mut *self.lock.data.get() }
    }
}

impl<T: ?Sized> Drop for RwLockReadGuard<'_, T> {
    fn drop(&mut self) {
        unsafe {
            self.lock.inner.read_unlock();
        }
    }
}

impl<T: ?Sized> Drop for RwLockWriteGuard<'_, T> {
    fn drop(&mut self) {
        self.lock.poison.done(&self.poison);
        unsafe {
            self.lock.inner.write_unlock();
        }
    }
}
```

`RwLock<T>`与`Mutex<T>`的代码逻辑基本一致，所以分析基本没有做。

Barrier 类型临界变量

Barrier建立了一个多个线程同步的等待点，当所有的线程都到达这个点后，每个线程才能恢复执行，否则，就在该点等待。
Barrier的实例：初始化的时候每一个线程负责不同的初始化内容，只有所有线程都完成了初始化之后，才能继续执行，否则会出现错误。此时，可以用Barrier建立同步点，每个线程完成初始化后就阻塞等待在这个点上，当所有线程都完成后，所有线程解除阻塞，继续运行。
也可用于多个线程协同工作，需要设置一个协同点，每个线程在完成一部分工作后需要等在协同点等待其他线程也都完成工作，然后才能继续工作。

```
pub struct Barrier {
    //BarrierState临界区数据
    lock: Mutex<BarrierState>,
    //等待及解除等待
    cvar: Condvar,
    //线程计数
    num_threads: usize,
}

struct BarrierState {
    //等待的线程数量
    count: usize,
    //唤醒标志
```

```
    generation_id: usize,
}

//表明线程是否是唤醒其他线程的线程
pub struct BarrierWaitResult(bool);


impl Barrier {
    //指定Barrier能够做多少个线程的同步
    pub fn new(n: usize) -> Barrier {
        Barrier {
            lock: Mutex::new(BarrierState { count: 0, generation_id: 0 }),
            cvar: Condvar::new(),
            num_threads: n,
        }
    }

    //等待在Barrier
    pub fn wait(&self) -> BarrierWaitResult {
        //获取临界区变量
        let mut lock = self.lock.lock().unwrap();
        let local_gen = lock.generation_id;
        //等待线程计数加1
        lock.count += 1;
        //判断是否已经有足够的线程
        if lock.count < self.num_threads {
            //有可能被其他情况唤醒，如收到信号
            //此处用循环来完成此种情况下的再次进入等待
            //判断是否满足等待的条件
            while local_gen == lock.generation_id {
                //进入等待
                lock = self.cvar.wait(lock).unwrap();
                //被唤醒,有可能被信号唤醒
            }
            //线程已经到达数目，返回
            BarrierWaitResult(false)
        } else {
            //线程已经到达规定数目
            lock.count = 0;
            //唤醒标志
            lock.generation_id = lock.generation_id.wrapping_add(1);
            //唤醒所有其他阻塞线程
            self.cvar.notify_all();
            BarrierWaitResult(true)
        }
    }
}

impl BarrierWaitResult {
    //唤醒其他线程的线程
    pub fn is_leader(&self) -> bool {
```

```
        self.0
    }
}
```

Once 类型分析

Once是对全局变量的初始化必须在多个线程中(例如，库)竞争执行且只需要执行一次时的需求的方案。
C的pthread库实现了pthread_once来实现这个特性。RUST实现了自己的方案。Once的call_once方法使得可以用闭包的形式初始化全局变量，闭包内的代码不必考虑竞争，由Once确保线程安全且只初始化只被执行一次。
代码如下：

```
type Masked = ();

pub struct Once {
    //用一个变量即实现状态，又实现了等待队列的头节点
    //最后两位是Once的状态，前面是* const Waiter的裸指针地址
    //Waiter是4字节对齐，因此地址最后两位为0，
    //这个设计技巧不值得倡导,这里是为了效率考虑
    state_and_queue: AtomicPtr<Masked>,
    //state_and_queue中包含了一个等待的头节点的裸指针
    _marker: marker::PhantomData<*const Waiter>,
}

//不能自动生成这两个trait
unsafe impl Sync for Once {}
unsafe impl Send for Once {}

impl UnwindSafe for Once {}

impl RefUnwindSafe for Once {}

pub struct OnceState {
    //闭包执行期间出现panic的标识
    poisoned: bool,
    //给初始化闭包使用，用来标识是否中毒，或者已经顺利完成
    set_state_on_drop_to: Cell<*mut Masked>,
}

//所有的静态变量可以使用ONCE_INIT进行赋值
pub const ONCE_INIT: Once = Once::new();

//闭包没有执行
const INCOMPLETE: usize = 0x0;
//闭包执行时线程panic
const POISONED: usize = 0x1;
```

```rust
//闭包正在执行
const RUNNING: usize = 0x2;
//初始化完成
const COMPLETE: usize = 0x3;

// 用来取出最后两位，用来做INCOMPLETE/POISONED/RUNNING/COMPLETE
const STATE_MASK: usize = 0x3;

//用来作为等待的线程队列节点，这些线程都对once做了闭包初始化的调用
#[repr(align(4))] //确保指针的地址的后2位无意义，可以用来作为状态，这是一个不值得推倡的技巧
struct Waiter {
    //标识自身
    thread: Cell<Option<Thread>>,
    signaled: AtomicBool,
    //next的节点
    next: *const Waiter,
}

// 等待的队列.
struct WaiterQueue<'a> {
    //Once的state_and_queue的引用
    state_and_queue: &'a AtomicPtr<Masked>,
    //初始化闭包的返回结果
    set_state_on_drop_to: *mut Masked,
}

impl Once {
    //一般直接使用ONCE_INIT
    pub const fn new() -> Once {
        Once {
            //初始赋值
            state_and_queue: AtomicPtr::new(ptr::invalid_mut(INCOMPLETE)),
            _marker: marker::PhantomData,
        }
    }

    //例如ONCE_INIT.call_once(|| {}), 在函数体内，可以对全局变量执行初始化，不必考虑
    //线程间安全问题。
    pub fn call_once<F>(&self, f: F)
    where
        F: FnOnce(),
    {
        // 是否已经完成初始化
        if self.is_completed() {
            return;
        }

        //以下将FnOnce()转换为了FnMut(state)
        let mut f = Some(f);
        //需要处理panic情况
```

```
        self.call_inner(false, &mut |_| f.take().unwrap()());
    }

    //不理会panic的初始化
    pub fn call_once_force<F>(&self, f: F)
    where
        F: FnOnce(&OnceState),
    {
        if self.is_completed() {
            return;
        }

        let mut f = Some(f);
        self.call_inner(true, &mut |p| f.take().unwrap()(p));
    }

    pub fn is_completed(&self) -> bool {
        self.state_and_queue.load(Ordering::Acquire).addr() == COMPLETE
    }

    fn call_inner(&self, ignore_poisoning: bool, init: &mut dyn
FnMut(&OnceState)) {
        let mut state_and_queue = self.state_and_queue.load(Ordering::Acquire);
        loop {
            //判断当前状态
            match state_and_queue.addr() {
                //没有等待线程且初始化完毕
                COMPLETE => break,
                //没有等待线程且已经POISONED，且不能忽视panic做初始化
                POISONED if !ignore_poisoning => {
                    // Panic to propagate the poison.
                    panic!("Once instance has previously been poisoned");
                }
                //没有等待线程，没有初始化，或者已经panic但可以初始化
                POISONED | INCOMPLETE => {
                    // 将状态转为RUNNING
                    let exchange_result =
self.state_and_queue.compare_exchange(
                        state_and_queue,
                        ptr::invalid_mut(RUNNING),
                        Ordering::Acquire,
                        Ordering::Acquire,
                    );
                    //判断是否出现竞争
                    if let Err(old) = exchange_result {
                        //有竞争者，再次做循环
                        state_and_queue = old;
                        continue;
                    }
                    // 本线程获得初始化权利，后面这段代码不会有竞争出现
                    // 创建其他线程等待的队列
```

```
                    let mut waiter_queue = WaiterQueue {
                        //设置Once的state_and_que为队列头部
                        state_and_queue: &self.state_and_queue,
                        //默认是POISONED
                        set_state_on_drop_to: ptr::invalid_mut(POISONED),
                    };
                    // 设置初始化状态
                    let init_state = OnceState {
                        poisoned: state_and_queue.addr() == POISONED,
                        //默认为COMPLETE
                        set_state_on_drop_to:
Cell::new(ptr::invalid_mut(COMPLETE)),
                    };
                    //调用初始化函数
                    init(&init_state);
                    //对等待队列中的状态进行更新，如果初始化闭包不关心
init_state(call_once)，则默认为COMPLETE
                    waiter_queue.set_state_on_drop_to =
init_state.set_state_on_drop_to.get();
                    //waiter_queue被释放，调用drop
                    break;
                }
                _ => {
                    // RUNNING，进入阻塞状态
                    assert!(state_and_queue.addr() & STATE_MASK == RUNNING);
                    wait(&self.state_and_queue, state_and_queue);
                    //阻塞被唤醒，重新获取新的状态并再次做判断循环
                    state_and_queue =
self.state_and_queue.load(Ordering::Acquire);
                }
            }
        }
    }
}

//进入等待队列
fn wait(state_and_queue: &AtomicPtr<Masked>, mut current_state: *mut Masked) {

    loop {
        //在不是初次循环的情况下，初始化结束后的竞争赋值
        //可能导致current_state被更新。
        if current_state.addr() & STATE_MASK != RUNNING {
            return;
        }

        // 针对本线程创建一个等待队列的节点
        let node = Waiter {
            thread: Cell::new(Some(thread::current())),
            signaled: AtomicBool::new(false),
            //将地址清零后两位后，得到Waiter节点的地址
            //如果是头节点，此处的偏移为0，next即是0
```

```
            next: current_state.with_addr(current_state.addr() & !STATE_MASK)
as *const Waiter,
        };
        //本身作为下一个节点时的地址
        let me = &node as *const Waiter as *const Masked as *mut Masked;

        //更换当前的state_and_queue，将新创建的node作为队列头
        let exchange_result = state_and_queue.compare_exchange(
            current_state,
            //node的地址与状态做或操作，一个变量即是队列头，又是状态
            me.with_addr(me.addr() | RUNNING),
            Ordering::Release,
            Ordering::Relaxed,
        );
        //判断是否成功
        if let Err(old) = exchange_result {
            //不成功，更新current_state，再次循环
            //此时
            current_state = old;
            //此处node会被drop掉
            continue;
        }

        //下面的代码面对一个非常复杂的竞争冲突分析
        //作为一个课题留给读者

        while !node.signaled.load(Ordering::Acquire) {
            //如果没有发信号，则阻塞
            thread::park();
            //阻塞结束后，进入循环再次判断是否信号已经被接收
        }
        break;
    }
}

impl Drop for WaiterQueue<'_> {
    fn drop(&mut self) {
        //更新Once的state_and_queue的值,并获取老值,
        //这个做法对规避竞争有很大的意义
        //即我的孩子我领走。
        let state_and_queue =
            self.state_and_queue.swap(self.set_state_on_drop_to,
Ordering::AcqRel);

        // 老值应该只可能是RUNNING状态
        assert_eq!(state_and_queue.addr() & STATE_MASK, RUNNING);

        unsafe {
            //获取等待的队列头地址
            let mut queue =
                state_and_queue.with_addr(state_and_queue.addr() & !STATE_MASK)
```

```
as *const Waiter;
            while !queue.is_null() {
                //保存下一个节点信息
                let next = (*queue).next;
                //发送信号，唤醒等待的线程
                let thread = (*queue).thread.take().unwrap();
                (*queue).signaled.store(true, Ordering::Release);
                queue = next;
                thread.unpark();
            }
        }
    }
}

//留给初始化闭包函数使用。
impl OnceState {
    pub fn is_poisoned(&self) -> bool {
        self.poisoned
    }

    pub(crate) fn poison(&self) {
        self.set_state_on_drop_to.set(ptr::invalid_mut(POISONED));
    }
}
```

OnceLock类型分析

`OnceLock<T>`是`OnceCell<T>`在多线程下的版本，也Once的一个具体的实例。 提供了多线程的情况下对变量做一次性初始化的解决方案。

```
pub struct OnceLock<T> {
    //保证多线程情况下仅做一次初始化
    once: Once,
    // 被仅初始化一次的变量
    value: UnsafeCell<MaybeUninit<T>>,
    //因为value是MaybeUninit<T>，所以需要PhantomData向编译器提示
    //本结构负责T的释放，以便编译器进行drop check
    _marker: PhantomData<T>,
}

impl<T> OnceLock<T> {
    //创建函数
    pub const fn new() -> OnceLock<T> {
        OnceLock {
            once: Once::new(),
            //获得正确的内存
            value: UnsafeCell::new(MaybeUninit::uninit()),
            _marker: PhantomData,
        }
```

```
    }

    //直接获取内部变量的引用
    unsafe fn get_unchecked(&self) -> &T {
        debug_assert!(self.is_initialized());
        //请参考UnsafeCell的内容
        (&*self.value.get()).assume_init_ref()
    }

    //直接获取内部变量的可变引用
    unsafe fn get_unchecked_mut(&mut self) -> &mut T {
        debug_assert!(self.is_initialized());
        (&mut *self.value.get()).assume_init_mut()
    }

    //仅在初始化过后才能返回内部引用
    pub fn get(&self) -> Option<&T> {
        if self.is_initialized() {
            // Safe b/c checked is_initialized
            Some(unsafe { self.get_unchecked() })
        } else {
            None
        }
    }

    //仅在初始化过后才能返回内部变量可变引用
    pub fn get_mut(&mut self) -> Option<&mut T> {
        if self.is_initialized() {
            // Safe b/c checked is_initialized and we have a unique access
            Some(unsafe { self.get_unchecked_mut() })
        } else {
            None
        }
    }

    //如果已经初始化，返回内部变量引用，
    //否则，调用f进行初始化，然后返回内部变量引用
    pub fn get_or_init<F>(&self, f: F) -> &T
    where
        F: FnOnce() -> T,
    {
        //Ok::<T,!>(f())将FnOnce()->T转换成了
        //FnOnce()->Result<T,!>,
        //并且只可能返回Ok()的值
        match self.get_or_try_init(|| Ok::<T, !>(f())) {
            Ok(val) => val,
            //编译器分析出不会返回Err,
        }
    }

    //可能不成功
```

```rust
pub fn get_or_try_init<F, E>(&self, f: F) -> Result<&T, E>
where
    F: FnOnce() -> Result<T, E>,
{
    // 如果已经初始化完成，则返回
    if let Some(value) = self.get() {
        return Ok(value);
    }
    //见后继方法的分析
    self.initialize(f)?;

    debug_assert!(self.is_initialized());

    //再次获得内部变量引用并返回
    Ok(unsafe { self.get_unchecked() })
}

//上面方法的支持方法
fn initialize<F, E>(&self, f: F) -> Result<(), E>
where
    F: FnOnce() -> Result<T, E>,
{
    let mut res: Result<(), E> = Ok(());
    let slot = &self.value;

    //利用once实现仅初始化一次
    self.once.call_once_force(|p| {
        match f() {
            Ok(value) => {
                //实现对value的赋值
                unsafe { (&mut *slot.get()).write(value) };
            }
            Err(e) => {
                res = Err(e);

                //设置once状态为POSIONED
                p.poison();
            }
        }
    });
    res
}

//修改内部变量的值,这个方法的编码技巧值得学习
pub fn set(&self, value: T) -> Result<(), T> {
    //用Some来做是否成功的判断
    let mut value = Some(value);
    //此处仅当赋值成功时才会调用value.take().unwrap()
    self.get_or_init(|| value.take().unwrap());
    match value {
        //成功设置了值
```

```
            None => Ok(()),
            //内部变量已经初始化过了
            Some(value) => Err(value),
        }
    }


    //对一个Pin<&Self>做初始化
    pub(crate) fn get_or_init_pin<F, G>(self: Pin<&Self>, f: F, g: G) ->
Pin<&T>
    where
        F: FnOnce() -> T,
        G: FnOnce(Pin<&mut T>),
    {
        //判断是否初始化完毕
        if let Some(value) = self.get_ref().get() {
            //初始化完毕，创建一个Pin返回
            return unsafe { Pin::new_unchecked(value) };
        }

        let slot = &self.value;

        self.once.call_once_force(|_| {
            let value = f();
            let value: &mut T = unsafe { (&mut *slot.get()).write(value) };
            //初始化成功后，调用回调函数g,完成进一步初始化
            g(unsafe { Pin::new_unchecked(value) });
        });

        //创建Pin返回
        unsafe { Pin::new_unchecked(self.get_ref().get_unchecked()) }
    }

    //消费掉self，返回内部变量
    pub fn into_inner(mut self) -> Option<T> {
        //见后继分析
        self.take()
    }

    pub fn take(&mut self) -> Option<T> {
        if self.is_initialized() {
            //重新创建一个Once
            self.once = Once::new();
            //将内部变量读出，并将value设置为默认值
            unsafe { Some((&mut *self.value.get()).assume_init_read()) }
        } else {
            None
        }
    }

    fn is_initialized(&self) -> bool {
```

```
        self.once.is_completed()
    }

}
```

LazyLock类型分析

`LazyLock<T>`是`Lazy<T>`的多线程版本

```
//惰性，在解引用时进行初始化
pub struct LazyLock<T, F = fn() -> T> {
    //初始化的目的类型
    cell: OnceLock<T>,
    //保存初始化闭包
    init: Cell<Option<F>>,
}

impl<T, F> LazyLock<T, F> {
    pub const fn new(f: F) -> LazyLock<T, F> {
        LazyLock { cell: OnceLock::new(), init: Cell::new(Some(f)) }
    }
}

impl<T, F: FnOnce() -> T> LazyLock<T, F> {
    //执行初始化
    pub fn force(this: &LazyLock<T, F>) -> &T {
        //利用OnceLock及闭包进行初始化
        this.cell.get_or_init(|| match this.init.take() {
            Some(f) => f(),
            None => panic!("Lazy instance has previously been poisoned"),
        })
    }
}

//在此方法进行初始化
impl<T, F: FnOnce() -> T> Deref for LazyLock<T, F> {
    type Target = T;
    fn deref(&self) -> &T {
        LazyLock::force(self)
    }
}
```