并发编程相关类型结构代码分析

并发编程主要包括线程及线程间通信内容。并发编程的熟练是程序员迈向高级水平的必由之路和标志之一。
本书认为读者已经很清楚并发的基础概念并且已经有了并发编码的基础。因此会直接涉及到一些并发中比较本质的代码。
按照先通用后特殊的原则，本节内容的先后顺序如下：
临界区同步：

1. 分析基于linux的RUST底层同步机制，包括futex, mutex, condvar, rwlock等。从linux系统分析，RUST实际上重新构建了自己的线程锁机制。在操作系统的适配层提供了与其他语言类似的类型，方法，函数。
2. 在1的基础上，分析RUST根据自身需求实现的适配扩展层。主要是匹配RUST的静态全局变量而进行了适配
3. 在1,2的基础上，RUST向标准库的使用者提供了屏蔽锁的概念的临界区数据类型。使用临界区数据类型的代码实际可以把临界区数据看成与智能指针类似的数据，不必额外关注线程间的互斥。

线程管理：

1. 操作系统的线程结构，函数的适配层，
2. 线程局部存储适配层
3. 线程同步原语的适配层
4. 标准库线程外部接口实现
5. 线程间消息通信的实现

临界区同步的linux适配层

代码路径： library/std/src/sys/unix/locks/*.rs
library/std/src/sys/unix/futex.rs

锁的基础设施FUTEX代码分析

FUTEX是一种高效率的互斥机制，其本质是将原先分配给内核的部分锁处理代码转移到用户态处理。
所有的临界区锁保护实际上是基于一个原子变量的检测及赋值，当这个原子变量处于一个范围时，允许代码执行，否则，就进入等待队列。当处于临界区的代码执行完毕后，会恢复原子变量唤醒等待队列等待的线程或进程。
传统上，对锁的原子变量的检测是放在内核态的，内核检测及修改原子变量并完成等待阻

塞操作或唤醒操作。这显然是一种效率很低的做法，因为大部分时候临界区是不会发生冲突的。每次检测都执行一次内核态与用户态的切换没有必要。因此，诞生了将原子变量检测与赋值放在用户态完成的思路。
这个思路需要完成的需求整体如下：

1. 用户态声明一个原子变量作为能否进入临界区的锁。如果该值被设置为某值，代表不能进入临界区，其他值代表可以进入临界区
2. 访问临界区的线程或进程修改原子变量表示要进入临界区，然后探测原子变量，看是否能够进入，能够进入则执行临界区访问
3. 不能进入则调用系统调用进入内核，内核会重新做原子变量判断并决定是否进入等待，这个过程是不可被其他线程打断的。
4. 位于临界区的线程或进程执行完毕临界区代码后，在用户态修改原子变量，并唤醒一个或所有等待的进程或线程
5. 可以满足跨进程

futex即是满足这个需求的设计。显然，传统的锁机制如pthread_mutex可以用futex方案重新设计。所以futex实际上是作为传统锁机制的基础设施存在。但对其做了解对于系统级编程仍然是有必要的。

RUST对futex的linux系统实现的适配如下：

```
pub fn futex_wait(futex: &AtomicI32, expected: i32, timeout: Option<Duration>)
-> bool {
    use super::time::Timespec;
    use crate::ptr::null;
    use crate::sync::atomic::Ordering::Relaxed;

    // 计算超时
    let timespec =
        timeout.and_then(|d|
Some(Timespec::now(libc::CLOCK_MONOTONIC).checked_add_duration(&d)?));

    loop {
        // 判断原子变量是否是期待的值
        if futex.load(Relaxed) != expected {
            //不是，直接返回
            return true;
        }

        // 是期待值，进入阻塞处理
        // 以下说明，RUST目前对futex的应用是线程间的同步，具体的系统调用请参考man手册
        let r = unsafe {
            //futex系统调用
            libc::syscall(
                libc::SYS_futex,
                futex as *const AtomicI32,
                //FUTEX_PRIVATE_FLAG说明仅支持本进程线程间的futex
```

```
                libc::FUTEX_WAIT_BITSET | libc::FUTEX_PRIVATE_FLAG,
                //传入的futex是这个值时就阻塞
                expected,
                //超时
                timespec.as_ref().map_or(null(), |t| &t.t as *const
libc::timespec),
                null::<u32>(), // This argument is unused for
FUTEX_WAIT_BITSET.
                !0u32,         // A full bitmask, to make it behave like a
regular FUTEX_WAIT.
            )
        };

        //错误处理，请参考libc的man手册
        match (r < 0).then(super::os::errno) {
            Some(libc::ETIMEDOUT) => return false,
            Some(libc::EINTR) => continue,
            _ => return true,
        }
    }
}

//唤醒一个等待futex的执行者
pub fn futex_wake(futex: &AtomicI32) -> bool {
    unsafe {
        //futex系统调用
        libc::syscall(
            libc::SYS_futex,
            futex as *const AtomicI32,
            libc::FUTEX_WAKE | libc::FUTEX_PRIVATE_FLAG,
            1,
        ) > 0
    }
}

/// 唤醒所有等待futex的所有者.
pub fn futex_wake_all(futex: &AtomicI32) {
    unsafe {
        //futex系统调用
        libc::syscall(
            libc::SYS_futex,
            futex as *const AtomicI32,
            libc::FUTEX_WAKE | libc::FUTEX_PRIVATE_FLAG,
            i32::MAX,
        );
    }
}
```

FUTEX的令人惊讶在于为什么如此之晚这一特性才被实现。

Mutex源代码分析

Mutex作为传统的临界区保护机制，在linux适配层上，RUST利用futex重新实现了Mutex库而没有使用pthread_mutex_t。

```
pub struct Mutex {
    /// 0: unlocked
    /// 1: locked, no other threads waiting
    /// 2: locked, and other threads waiting (contended)
    /// 用作锁判断的原始变量,值的含义见上面英文注释
    futex: AtomicI32,
}

impl Mutex {
    pub const fn new() -> Self {
        //初始化为不加锁
        Self { futex: AtomicI32::new(0) }
    }

    pub unsafe fn init(&mut self) {}

    pub unsafe fn destroy(&self) {}

    //如果临界区操作的代码短，可以用try_lock，获取到锁后
    //即可快速完成操作，作为整体不进入内核的方案。
    //try_lock如果返回false，需要调用lock等待锁被打开
    pub unsafe fn try_lock(&self) -> bool {
        //利用原子操作试图加锁，如果futex不为0,会失败
        //如果成功，表明获得了锁
        self.futex.compare_exchange(0, 1, Acquire, Relaxed).is_ok()
    }

    //如果临界区操作代码较长，则应该直接调用lock方法
    pub unsafe fn lock(&self) {
        if self.futex.compare_exchange(0, 1, Acquire, Relaxed).is_err() {
            //如果失败，说明已经由其他线程占有了锁，本线程需要等待
            self.lock_contended();
        }
    }

    //执行后会等待其他线程临界区操作结束
    fn lock_contended(&self) {
        // 这里处理另一个线程很快结束操作临界区的情况,
        // 再次试图避免进入内核
        let mut state = self.spin();

        // 自旋等待后，判断临界区是否已经可以进入
        if state == 0 {
            //试图再次获取锁
            match self.futex.compare_exchange(0, 1, Acquire, Relaxed) {
```

```
                //成功，不进入内核
                Ok(_) => return, // Locked!
                //不成功，state应为当前的futex
                Err(s) => state = s,
            }
        }

        //需要调用系统内核进入等待状态
        loop {
            // 如果没有其他线程等待，则修改futex为2来通知有线程在等待锁
            if state != 2 && self.futex.swap(2, Acquire) == 0 {
                //如果此时发现已经获取了锁，则返回
                //出现这个结果表明在代码执行时，另一个线程释放了锁
                return;
            }

            // 如果state是2且futex不是0，则需要阻塞.
            futex_wait(&self.futex, 2, None);

            // 阻塞被解除后，再次做自旋以规避进入阻塞
            state = self.spin();
        }
    }

    // 在临界区操作非常快的情况下，不进入操作系统内核的解决方案
    fn spin(&self) -> i32 {
        let mut spin = 100;
        loop {
            // 读取原子变量
            let state = self.futex.load(Relaxed);

            // 判断是否循环结束或者其他线程已经在等待/其他线程退出临界区
            if state != 1 || spin == 0 {
                //判断真，直接返回
                return state;
            }

            //做个CPU自旋
            crate::hint::spin_loop();
            spin -= 1;
        }
    }

    //解锁，Lock调用后及try_Lock调用返回为真时，需要调用这个函数。
    pub unsafe fn unlock(&self) {
        if self.futex.swap(0, Release) == 2 {
            // 如果是2，说明有线程通过操作系统内核等待，需要wake
            self.wake();
        }
    }
```

```
    #[cold]
    fn wake(&self) {
        //通知操作系统内核，唤醒一个线程
        futex_wake(&self.futex);
    }
}
```

Convar分析

RUST标准库的条件变量Condvar解决方案：
条件变量本身是一种信号机制， 通常用于生产消费的两个线程或多个线程。生产线程完成临界区数据后，向消费线程发出通知。消费线程拿不到临界区数据时，会阻塞等待信号。
条件变量需要与一个Mutex的变量配合，来完成临界区数据及信号自身的保护。

```
pub struct Condvar {
    // 本身是一个原子变量，值的变化表示条件变化
    futex: AtomicI32,
}

impl Condvar {
    //初始化
    pub const fn new() -> Self {
        Self { futex: AtomicI32::new(0) }
    }

    pub unsafe fn init(&mut self) {}

    pub unsafe fn destroy(&self) {}


    //通知一个线程条件已经变化，调用此函数前，应该对关联
    //mutex加锁，调用后，应释放锁
    pub unsafe fn notify_one(&self) {
        //简单的改变值，表示条件已经变化
        self.futex.fetch_add(1, Relaxed);
        //用futex操作系统调用完成通知,仅唤醒一个线程
        futex_wake(&self.futex);
    }

    //调用前应对关联mutex加锁，调用后应释放锁
    pub unsafe fn notify_all(&self) {
        self.futex.fetch_add(1, Relaxed);
        //唤醒所有等待线程
        futex_wake_all(&self.futex);
    }

    pub unsafe fn wait(&self, mutex: &Mutex) {
        self.wait_optional_timeout(mutex, None);
    }
```

```
    pub unsafe fn wait_timeout(&self, mutex: &Mutex, timeout: Duration) -> bool
{
        self.wait_optional_timeout(mutex, Some(timeout))
    }

    //等待条件变化,调用此函数前，应该将关联mutex上锁保护self.futex及其他临界区的操作,
    //调用后，应释放锁
    unsafe fn wait_optional_timeout(&self, mutex: &Mutex, timeout:
Option<Duration>) -> bool {
        // 外部应该先将mutex上锁，防止其他线程改变self.futex，此处获取当前值
        let futex_value = self.futex.load(Relaxed);

        // 释放锁。
        mutex.unlock();

        // 如果条件不变，则进入等待
        let r = futex_wait(&self.futex, futex_value, timeout);

        // 条件已经变化，加锁保护self.futex的值及其他的临界区的操作
        mutex.lock();

        r
    }
}
```

RWLock源代码分析

读写锁适用的场景如下：
临界区允许多个读同时存在，但读写不能同时存在。临界区读的时候要设置锁处于读锁，此时允许读不允许写。如果有线程要写，需要等待。 临界区写的时候要设置写锁，此时和正常的锁是一致的，所有其他试图访问临界区的线程都需要等待。

```
pub struct RwLock {
    // 从0到30位用来作为锁的计数:
    //   0: Unlocked
    //   1..=0x3FFF_FFFE: 作为读线程的计数，并作为读锁
    //   0x3FFF_FFFF: 写锁
    // Bit 30: 有其他读线程在等待，此时应该是写锁.
    // Bit 31: 有其他写线程在等待，此时读锁及写锁都有可能.
    state: AtomicI32,
    // 利用这个值的变化做信号的通知，类似与CondVar的操作.
    writer_notify: AtomicI32,
}

const READ_LOCKED: i32 = 1;
const MASK: i32 = (1 << 30) - 1;
const WRITE_LOCKED: i32 = MASK;
const MAX_READERS: i32 = MASK - 1;
```

```
const READERS_WAITING: i32 = 1 << 30;
const WRITERS_WAITING: i32 = 1 << 31;

fn is_unlocked(state: i32) -> bool {
    state & MASK == 0
}

fn is_write_locked(state: i32) -> bool {
    state & MASK == WRITE_LOCKED
}

fn has_readers_waiting(state: i32) -> bool {
    state & READERS_WAITING != 0
}

fn has_writers_waiting(state: i32) -> bool {
    state & WRITERS_WAITING != 0
}

//这个函数用来判断是否可以进入临界区读
fn is_read_lockable(state: i32) -> bool {
    // 只有在读线程的数量小于最大值，且没有其他线程在等着读或等着写，此时可以更新读锁
    // 否则应该进入读等待队列，
    // 只要有线程等着写，要后继的读线程就需要阻塞在读锁中
    state & MASK < MAX_READERS && !has_readers_waiting(state) &&
!has_writers_waiting(state)
}

fn has_reached_max_readers(state: i32) -> bool {
    state & MASK == MAX_READERS
}

impl RwLock {
    pub const fn new() -> Self {
        Self { state: AtomicI32::new(0), writer_notify: AtomicI32::new(0) }
    }

    pub unsafe fn destroy(&self) {}

    //试图读，一般如果需要做读锁，应直接调用read，此函数用作不希望阻塞的情况
    pub unsafe fn try_read(&self) -> bool {
        // 如果判断可读，则对读锁加1,然后进入临界区做读操作，如果失败，可以调用read等
待读
        self.state
            .fetch_update(Acquire, Relaxed, |s| is_read_lockable(s).then(|| s +
READ_LOCKED))
            .is_ok()
    }

    //获取读锁或阻塞等待到能获取
    pub unsafe fn read(&self) {
```

```
        let state = self.state.load(Relaxed);
        if !is_read_lockable(state)
            //此时更新读锁失败，证明锁已经被改变
            || self
                .state
                .compare_exchange_weak(state, state + READ_LOCKED, Acquire,
Relaxed)
                .is_err()
        {
            //复杂的读锁获取或者进入等待
            self.read_contended();
        }
    }

    //解锁读
    pub unsafe fn read_unlock(&self) {
        //更新读锁
        let state = self.state.fetch_sub(READ_LOCKED, Release) - READ_LOCKED;

        // 除非有写线程在等待，否则此时不应该有线程在等待读. 写线程优先于读
        debug_assert!(!has_readers_waiting(state) ||
has_writers_waiting(state));

        // 如果有线程等着写，那就做唤醒操作，解锁读时，一定是有等着写的线程才能导致读
线程被阻塞.
        if is_unlocked(state) && has_writers_waiting(state) {
            self.wake_writer_or_readers(state);
        }
    }

    //读线程阻塞处理
    fn read_contended(&self) {
        //自旋读，主要处理同时读的读锁更细冲突导致的不一致情况
        let mut state = self.spin_read();

        loop {
            // 再次尝试获取读锁
            if is_read_lockable(state) {
                match self.state.compare_exchange_weak(state, state +
READ_LOCKED, Acquire, Relaxed)
                {
                    //成功
                    Ok(_) => return, // Locked!
                    //如果是读锁不一致，那就再次尝试
                    Err(s) => {
                        state = s;
                        continue;
                    }
                }
            }
```

```
            // 如果达到最大的读线程数目，可能是有线程忘记解锁
            if has_reached_max_readers(state) {
                panic!("too many active read locks on RwLock");
            }

            // 确保等待读的标志已经被设置.
            if !has_readers_waiting(state) {
                if let Err(s) =
                    self.state.compare_exchange(state, state | READERS_WAITING,
Relaxed, Relaxed)
                {
                    //这里，上段的is_read_lockable(state)执行是失败的，所以不会有读
锁被增加的问题
                    //失败说明读锁又在被并发修改，所以此时可能可以读了，要再次尝试
                    state = s;
                    continue;
                }
            }

            // 终于可以阻塞了，如果此时state没变，就会阻塞
            futex_wait(&self.state, state | READERS_WAITING, None);

            // 此处或者是阻塞失败，或者被唤醒，两者都要重新看是否能够获得锁或者继续阻
塞
            state = self.spin_read();
        }
    }

    // 试图读，应该在不希望阻塞的情况下调用，此方法返回成功代表已经获取了写锁
    pub unsafe fn try_write(&self) -> bool {
        self.state
            //读的时候必须处于Unlocked状态
            .fetch_update(Acquire, Relaxed, |s| is_unlocked(s).then(|| s +
WRITE_LOCKED))
            .is_ok()
    }

    //此函数用于获取写锁或阻塞等待到能获取
    pub unsafe fn write(&self) {
        if self.state.compare_exchange_weak(0, WRITE_LOCKED, Acquire,
Relaxed).is_err() {
            //进入更复杂的锁获取或等待
            self.write_contended();
        }
    }

    //解锁写
    pub unsafe fn write_unlock(&self) {
        //更新值
        let state = self.state.fetch_sub(WRITE_LOCKED, Release) - WRITE_LOCKED;
```

```
        //还没有唤醒，应该没有其他线程冲突
        debug_assert!(is_unlocked(state));

        //有等待线程的话，就唤醒
        if has_writers_waiting(state) || has_readers_waiting(state) {
            self.wake_writer_or_readers(state);
        }
    }

    //进入写等待队列的处理
    fn write_contended(&self) {
        let mut state = self.spin_write();

        //假定当前没有线程等待写，后继处理如果更新，
        //则说明有两个以上的线程在竞争获取写锁，所以
        //设置state时需要同时更新写等待标志位
        let mut other_writers_waiting = 0;

        loop {
            //  如果unlocked，那试图获取写锁
            if is_unlocked(state) {
                match self.state.compare_exchange_weak(
                    state,
                    state | WRITE_LOCKED | other_writers_waiting,
                    Acquire,
                    Relaxed,
                ) {
                    //获取成功
                    Ok(_) => return, // Locked!
                    Err(s) => {
                        //获取失败，再次循环
                        state = s;
                        continue;
                    }
                }
            }

            // 不为unlock，进入写等待队列并更新写等待标志
            if !has_writers_waiting(state) {
                if let Err(s) =
                    self.state.compare_exchange(state, state | WRITERS_WAITING,
Relaxed, Relaxed)
                {
                    //更新失败，说明有同时访问者，需要重新试图获取写锁
                    state = s;
                    continue;
                }
            }

            // 不为unlock，且写等待标志位已经设置
            // 已经有其他写线程在等待.
```

```
            other_writers_waiting = WRITERS_WAITING;

            // 获取写锁解除的通知变量
            let seq = self.writer_notify.load(Acquire);

            let s = self.state.load(Relaxed);
            //这个地方错了，本意估计是is_unlocked(s)，state此时已经确定
            //为lock了(注:作者在github提交了issue，目前此错误已经被修改)
            if is_unlocked(state) || !has_writers_waiting(s) {
                //这里如果又有变化，那么再次试图获得写锁
                state = s;
                continue;
            }

            // 阻塞，等待解锁通知
            futex_wait(&self.writer_notify, seq, None);

            // 失败或者唤醒，重新再次试图获取写锁
            state = self.spin_write();
        }
    }

    /// 唤醒等待线程
    fn wake_writer_or_readers(&self, mut state: i32) {
        assert!(is_unlocked(state));

        // 仅有写现程在等.
        if state == WRITERS_WAITING {
            //改变写等待标记，此时可能会形成一个冲突，所以write_contended会等待之前再
做
            //一次判断
            match self.state.compare_exchange(state, 0, Relaxed, Relaxed) {
                Ok(_) => {
                    self.wake_writer();
                    return;
                }
                Err(s) => {
                    // 有冲突，更新state，此时只有可能是读线程在更新.
                    state = s;
                }
            }
        }

        // 即有读线程在等，也有写线程在等
        if state == READERS_WAITING + WRITERS_WAITING {
            //清写等待标志，此时0到30位肯定是0
            if self.state.compare_exchange(state, READERS_WAITING, Relaxed,
Relaxed).is_err() {
                // 不应该出错，如果出错，那锁状态已经不对，无能为力了，而且不知道错误
在哪里.
                // 感觉还是应该panic一下
```

```
                return;
            }
            //唤醒等待的写线程
            if self.wake_writer() {
                return;
            }
            // 执行到这里，证明没有写线程在等，那接下来
            // 处理读线程。此时直接修改state就可以，因为不可能有其他
            // 线程修改state了
            state = READERS_WAITING;
        }

        // 唤醒等待的读线程
        if state == READERS_WAITING {
            if self.state.compare_exchange(state, 0, Relaxed, Relaxed).is_ok()
{
                //唤醒所有读，这里读线程被唤醒后，仍然可能会有写线程插入，但不会出现问
题
                futex_wake_all(&self.state);
            }
        }
    }

    /// 唤醒写线程
    fn wake_writer(&self) -> bool {
        //类似CondVar的处理方式，修改唤醒标志，然后唤醒即可
        self.writer_notify.fetch_add(1, Release);
        futex_wake(&self.writer_notify)
    }

    /// 自旋，处理一些在很短的时间内的状态修改使得锁可以获取，规避进入内核.
    fn spin_until(&self, f: impl Fn(i32) -> bool) -> i32 {
        let mut spin = 100; // Chosen by fair dice roll.
        loop {
            let state = self.state.load(Relaxed);
            //满足函数要求或自旋时间到，退出
            if f(state) || spin == 0 {
                return state;
            }
            crate::hint::spin_loop();
            spin -= 1;
        }
    }

    fn spin_write(&self) -> i32 {
        // 如果为unlock状态或者已经明确有写线程在等待并做了写等待置位.
        self.spin_until(|state| is_unlocked(state) ||
has_writers_waiting(state))
    }

    fn spin_read(&self) -> i32 {
```

```
        // 如果没有写锁，或者写等待或者读等待已经被置位
        self.spin_until(|state| {
            !is_write_locked(state) || has_readers_waiting(state) ||
has_writers_waiting(state)
        })
    }
}
```

可重入的Mutex

如果一个线程调用lock获取锁之后，允许其在临界区的代码又对该锁调用lock，这个锁是Reentrant mutex。用futex处理这种情况效率不高，因为需要多次进入内核获取线程信息。因此，使用已有的libc的pthread_mutex_t的机制.

```
pub struct ReentrantMutex {
    //仅用来给libc使用
    inner: UnsafeCell<libc::pthread_mutex_t>,
}

unsafe impl Send for ReentrantMutex {}
unsafe impl Sync for ReentrantMutex {}

impl ReentrantMutex {
    //因为这个初始化没有完成可重入锁的设置，所以实际上没有初始化
    //但因为libc的限制，必须要先创建变量才能后完成初始化，所以需要此关联函数
    //调用此函数后，再调用init方法完成初始化
    pub const unsafe fn uninitialized() -> ReentrantMutex {
        ReentrantMutex { inner:
UnsafeCell::new(libc::PTHREAD_MUTEX_INITIALIZER) }
    }

    //完成可重入锁的设置
    pub unsafe fn init(&self) {
        //栈中定义一块内存
        let mut attr = MaybeUninit::<libc::pthread_mutexattr_t>::uninit();
        //利用C函数做初始化，
        cvt_nz(libc::pthread_mutexattr_init(attr.as_mut_ptr())).unwrap();
        //创建类型变量
        let attr = PthreadMutexAttr(&mut attr);
        //设置attr属性
        cvt_nz(libc::pthread_mutexattr_settype(attr.0.as_mut_ptr(),
libc::PTHREAD_MUTEX_RECURSIVE))
            .unwrap();
        //完成初始化
        cvt_nz(libc::pthread_mutex_init(self.inner.get(),
attr.0.as_ptr())).unwrap();
    }
```

```
    //lock调用
    pub unsafe fn lock(&self) {
        //简单的调用libc函数
        let result = libc::pthread_mutex_lock(self.inner.get());
        debug_assert_eq!(result, 0);
    }

    //不想阻塞时的调用
    pub unsafe fn try_lock(&self) -> bool {
        //简单的调用libc
        libc::pthread_mutex_trylock(self.inner.get()) == 0
    }

    //解锁
    pub unsafe fn unlock(&self) {
        let result = libc::pthread_mutex_unlock(self.inner.get());
        debug_assert_eq!(result, 0);
    }

    //释放锁
    pub unsafe fn destroy(&self) {
        let result = libc::pthread_mutex_destroy(self.inner.get());
        debug_assert_eq!(result, 0);
    }
}
```

RUST适配层扩展的锁机制

代码路径： library/std/src/sys_common/mutex.rs|condvar.rs|rwlock.rs

适用于静态变量的锁

因为所有权的定义，如果锁作为静态变量存在，则其初始化函数必须在编译期执行，即为const fn。静态锁主要用于保护静态变量形成的临界区。

静态Mutex结构代码如下：

```
//用于static变量
pub struct StaticMutex(imp::Mutex);

unsafe impl Sync for StaticMutex {}

impl StaticMutex {
    /// const 函数可以用于static变量赋值
    pub const fn new() -> Self {
        Self(imp::Mutex::new())
    }
```

```
    //上锁后用锁的引用形成封装结构返回，StaticMutexGuard见下文分析
    pub unsafe fn lock(&'static self) -> StaticMutexGuard {
        self.0.lock();
        StaticMutexGuard(&self.0)
    }
    //没有设计unlock方法
}

//此结构设计主要是充分利用RUST的编译器来简化解锁代码，
//使用StaticMutexGuard后可以不必再考虑解锁这件事
pub struct StaticMutexGuard(&'static imp::Mutex);

impl Drop for StaticMutexGuard {
    //生命周期终止时做unlock
    fn drop(&mut self) {
        unsafe {
            self.0.unlock();
        }
    }
}
```

适用于静态变量的读写锁

```
pub struct StaticRwLock(imp::RwLock);

impl StaticRwLock {
    /// const fn以初始化静态读写锁.
    pub const fn new() -> Self {
        Self(imp::RwLock::new())
    }

    pub fn read(&'static self) -> StaticRwLockReadGuard {
        unsafe { self.0.read() };
        StaticRwLockReadGuard(&self.0)
    }

    pub fn write(&'static self) -> StaticRwLockWriteGuard {
        unsafe { self.0.write() };
        StaticRwLockWriteGuard(&self.0)
    }
}

pub struct StaticRwLockReadGuard(&'static imp::RwLock);

impl Drop for StaticRwLockReadGuard {
    fn drop(&mut self) {
        unsafe {
            self.0.read_unlock();
        }
    }
```

```
}

pub struct StaticRwLockWriteGuard(&'static imp::RwLock);

impl Drop for StaticRwLockWriteGuard {
    fn drop(&mut self) {
        unsafe {
            self.0.write_unlock();
        }
    }
}
```

适用于非静态变量的锁

MovableMutex

代码如下：

```
//imp::MoveableMutex在linux即imp::Mutex，其他系统基本也一样
pub struct MovableMutex(imp::MovableMutex);

unsafe impl Sync for MovableMutex {}

impl MovableMutex {
    /// 创建锁
    pub fn new() -> Self {
        let mut mutex = imp::MovableMutex::from(imp::Mutex::new());
        //需要调用init()，这里区别于StaticMutex
        unsafe { mutex.init() };
        Self(mutex)
    }

    pub(super) fn raw(&self) -> &imp::Mutex {
        &self.0
    }

    //获取锁
    pub fn raw_lock(&self) {
        unsafe { self.0.lock() }
    }

    //不希望阻塞获取锁
    pub fn try_lock(&self) -> bool {
        unsafe { self.0.try_lock() }
    }

    //释放锁
    pub unsafe fn raw_unlock(&self) {
        self.0.unlock()
```

```
    }
}

impl Drop for MovableMutex {
    fn drop(&mut self) {
        //用pthread_mutex_t需要
        unsafe { self.0.destroy() };
    }
}
```

条件变量

```
//对Condvar的关联Mutex做check
type CondvarCheck = <imp::MovableMutex as check::CondvarCheck>::Check;

/// 对操作系统的Condvar做的封装.
pub struct Condvar {
    //就是imp::Condvar
    inner: imp::MovableCondvar,
    check: CondvarCheck,
}

impl Condvar {
    /// 创建新的Condvar.
    pub fn new() -> Self {
        let mut c = imp::MovableCondvar::from(imp::Condvar::new());
        unsafe { c.init() };
        Self { inner: c, check: CondvarCheck::new() }
    }

    /// 发信号唤醒一个等待此Condvar的线程.
    pub fn notify_one(&self) {
        unsafe { self.inner.notify_one() };
    }

    /// 发信号唤醒所有等待此信号的线程.
    pub fn notify_all(&self) {
        unsafe { self.inner.notify_all() };
    }

    /// 等待信号.
    pub unsafe fn wait(&self, mutex: &MovableMutex) {
        //确保始终使用同一个关联Mutex
        self.check.verify(mutex);
        // 阻塞并等待信号
        self.inner.wait(mutex.raw())
    }

    //超时等待
    pub unsafe fn wait_timeout(&self, mutex: &MovableMutex, dur: Duration) ->
```

```
bool {
        self.check.verify(mutex);
        self.inner.wait_timeout(mutex.raw(), dur)
    }
}

impl Drop for Condvar {
    fn drop(&mut self) {
        unsafe { self.inner.destroy() };
    }
}
```

RWLock

```
pub struct MovableRwLock(imp::MovableRwLock);

impl MovableRwLock {
    pub fn new() -> Self {
        Self(imp::MovableRwLock::from(imp::RwLock::new()))
    }

    pub fn read(&self) {
        unsafe { self.0.read() }
    }

    pub fn try_read(&self) -> bool {
        unsafe { self.0.try_read() }
    }

    pub fn write(&self) {
        unsafe { self.0.write() }
    }

    pub fn try_write(&self) -> bool {
        unsafe { self.0.try_write() }
    }

    pub unsafe fn read_unlock(&self) {
        self.0.read_unlock()
    }

    pub unsafe fn write_unlock(&self) {
        self.0.write_unlock()
    }
}

impl Drop for MovableRwLock {
    fn drop(&mut self) {
        unsafe { self.0.destroy() };
```

```
    }
}
```

RUST中锁的杂项

加锁返回的统一类型

```rust
//lock方法的返回结果
pub type LockResult<Guard> = Result<Guard, PoisonError<Guard>>;
//try_lock方法的返回结果
pub type TryLockResult<Guard> = Result<Guard, TryLockError<Guard>>;

//错误类型
pub struct PoisonError<T> {
    guard: T,
}

impl<T> PoisonError<T> {
    //创建一个错误变量
    pub fn new(guard: T) -> PoisonError<T> {
        PoisonError { guard }
    }

    //从Error中获取导致错误的变量
    pub fn into_inner(self) -> T {
        self.guard
    }

    //获取导致错误变量的引用
    pub fn get_ref(&self) -> &T {
        &self.guard
    }

    //获取可变引用
    pub fn get_mut(&mut self) -> &mut T {
        &mut self.guard
    }
}

//Mutex<T> try_lock错误返回
pub enum TryLockError<T> {
    //线程异常返回
    Poisoned(PoisonError<T>),
    //临界区已经被锁，需要阻塞
    WouldBlock,
}
```

Poison状态

如果一个线程在获取一个锁的期间发生了panic，则锁保护的临界区的数据已经不能认为是正确的。因为panic导致锁在一个未期望的代码位置解锁。此时，需要对锁标志一个状态，RUST名词称为锁处于Poison状态，并设计了Flag来表示这个状态。
代码如下：

```
// 用于标识线程在加锁的状态下panic退出
pub struct Flag {
    failed: AtomicBool,
}

impl Flag {
    //初始的状态时候为假
    pub const fn new() -> Flag {
        Flag { failed: AtomicBool::new(false) }
    }

    //加锁的时候被调用，此时如果本线程已经panic，则需要返回错误
    pub fn borrow(&self) -> LockResult<Guard> {
        //获取本线程的panic状态，相关部分在Thread一节会再解释
        let ret = Guard { panicking: thread::panicking() };
        //如果Flag已经是真，则返回Err并给出本线程状态，否则返回Ok
        if self.get() { Err(PoisonError::new(ret)) } else { Ok(ret) }
    }

    //释放锁的时候被调用，如果本线程panic，则会更新Flag为true
    pub fn done(&self, guard: &Guard) {
        if !guard.panicking && thread::panicking() {
            self.failed.store(true, Ordering::Relaxed);
        }
    }

    //获得Flag的值
    pub fn get(&self) -> bool {
        self.failed.load(Ordering::Relaxed)
    }
}
```