

Rust Concurrency

Multithreading 免驚啦！



[@weihanglo](https://github.com/weihanglo)

Outline

- What is Concurrency
- Why Concurrency is Hard
- How to Control Concurrency
- Concurrency Model in Rust
 - Threading Model
 - Send and Sync Traits
 - Synchronization Primitives
 - Higher-level Synchronization Objects
- Patterns of Concurrent Programming
- References

What is Concurrency

From Oxford Dictionary

concurrent

/kən'kʌr(ə)nt/

adjective

Existing, happening, or done at the same time.

✓ at the same time

From Wikipedia

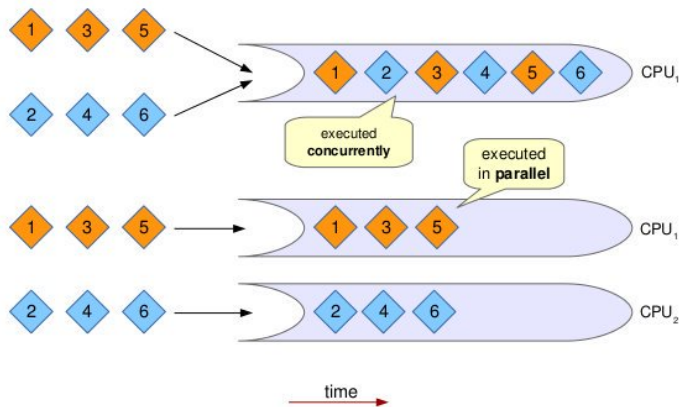
...the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome.

✓ executed out-of-order or in partial order

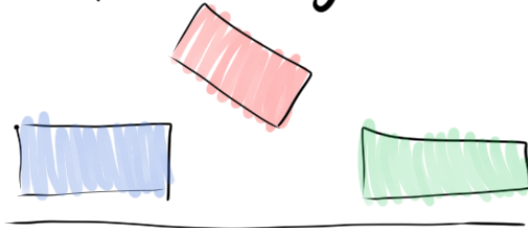
Concurrency

- 多個運算可依任意順序執行。
- 每個運算可以交錯，甚至重疊。
- Concurrent \iff Sequential
- Concurrency \neq Parallelism.

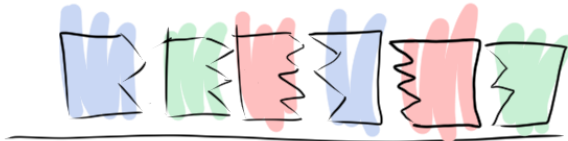
Parallelism vs. Concurrency



Parallel Parking



Concurrent Parking



[By Omar Ferrer](#)

Concurrency 的優點

- **降低 latency:** 工作可以被分成小單元再並行運算。
- **隱藏 latency:** long-running 可以並行運算（例如磁碟或網路 I/O）。
- **增加 throughput:** 不考慮其他開銷，大部分並行運算 throughput 會增加。

Why Concurrency is Hard

Multithreaded programming



Out-of-order Execution

- Compiler reorder instructions
 - Compiler 會最佳化，執行 CPU 指令看起來就會和程式碼順序不一致，例如先把讀取記憶體搬到 code block 最前面，讓 CPU 可以 prefetch 記憶體。
 - 單執行緒的程式中也可能發生，例如寫 signal 或 interrupt handler 這種 low-level 程式。
 - 這次不細講，有興趣請參考 [Linux Kernel memory barriers](#)。
- Single processor 下的 [Out-of-order Execution \(OoOE\)](#)
 - 不同指令所需執行時間長度不同，當前幾條指令尚未完成，其他提前完成的 CPU 指令就是 OoOE。
 - Rust 目前不處理這種 concurrency，由 CPU 處理。
 - Meltdown [就是透過 OoOE 來攻擊](#)。
- Multiprocessor 下多個執行緒同時執行，導致以非預期的順序存取資源。

Data Race and Race Condition

- **Data Race**：兩個存取記憶體的操作同時
 1. 存取同一個記憶體位址，並由兩個執行緒執行
 2. 不是只讀操作，也不是同步操作（synchronization operations）
- **Race Condition**：不同的程式執行順序影響輸出結果，通常由外部影響，例如 context switch、OS signal、hardware interrupt 或 multiprocessor。
- Data Race 可以導致 Race Condition，但不必然；反之 Race Condition 也不一定由 Data Race 產生。

[Data Race vs Race Condition 小品文](#)



How to Control Concurrency

Concurrency Control

- 目標：維持最終輸出的正確性，並越快輸出越好。
- 手段：
 1. 同步（Synchronization）：同步執行單元間的資訊，例如 shared memory 或 message passing。
 2. 調節（Coordination）：調節執行單元間的執行順序，如使用 Semaphores 等待執行緒執行，或使用 priority weight 決定執行先後順序。

實際上 Synchronization 和 Coordination 常常融合在一起，不易區分。

Concurrency Model in Rust

Threading Model

由於 Rust 最初定位在 system programming language，所以

- 沒有 green threads，也不存在任何 lightweight threads，
- `std::thread` 會產生 1:1 的 OS thread 與 user thread。

Threads 之間溝通可以透過

- 共享記憶體的 synchronization primitives，例如[所有 atomic 類型](#)。
- 稍微高級，但依舊底層的 Synchronization Objects，如 [Mutex](#)、[Condvar](#)。
- 還有內建使用 channel 的 message passing 溝通，不過只有單一方向的 [std::sync::mpsc](#)（多訊息生產者 producer，單一接收者 consumer）。

簡單的 thread 操作

建立一個 thread (thread::spawn)

```
use std::thread;

thread::spawn(move || {
    // some work here
});
```

建立一個 thread，並透過 JoinHandle 等待 thread 執行完畢

```
use std::thread;

let child = thread::spawn(move || {
    // some work here
});
// some work here
let res = child.join();
```


簡單的 thread 操作

- `std::thread::yield_now()`：放棄當前的 thread，通知 OS scheduler 執行其他 thread。通常會避免此操作，改用 synchronization object。
- `std::thread::park()`：暫停當前的 thread，可以再呼叫 `std::thread::Thread::unpark(&self)` 恢復執行緒運作。

Borrow check in thread

Rust 的 Concurrency Programming 沒能擺脫 borrow checker 的魔爪，依然活在 lifetime 的陰影之下中，因此，資料在 thread 之間的傳遞共享仍然受「**共享不可變，可變不共享**」的原則保護。

Borrow checker in thread

```
use std::thread;
fn main() {
    let v = vec![1, 2, 3];
    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });
    drop(v); // oh no!
    handle.join().unwrap();
}
```

error[E0373]: closure may outlive the current function, but it borrows `v`, which is owned by the current function

--> src/main.rs:6:32

```
6 |         let handle = thread::spawn(|| {
    |                                     ^^ may outlive borrowed value `v`
7 |             println!("Here's a vector: {:?}", v);
    |                                     - `v` is borrowed here
```

help: to force the closure to take ownership of `v` (and any other referenced variables), use the `move` keyword

```
6 |         let handle = thread::spawn(move || {
```

Borrow checker in thread (add move)

```
use std::thread;
fn main() {
    let v = vec![1, 2, 3];
    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });
    drop(v); // oh no!
    handle.join().unwrap();
}
```

error[E0382]: use of moved value: `v`

--> src/main.rs:7:10

```
4 |         let handle = thread::spawn(move || {
    |                                     ----- value moved into closure here
5 |             println!("Here's a vector: {:?}", v);
    |                                     - variable moved due to use in
6 |         });
7 |         drop(v); // oh no!
    |           ^ value used here after move
```

Borrow checker in thread (remove drop)

```
use std::thread;
fn main() {
    let v = vec![1, 2, 3];
    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });
    handle.join().unwrap();
}
```

Borrow checker in thread (push values)

```
use std::thread;
fn main() {
    let mut v = vec![1, 2, 3];
    let handle = thread::spawn(move || {
        v.push(4);
    });
    handle.join().unwrap();
}
```

Borrow checker in thread (push values)

```
use std::thread;
fn main() {
    let mut v = vec![1, 2, 3];
    let handle = thread::spawn(move || {
        v.push(4);
    });
    dbg!(&v);
    handle.join().unwrap();
}
```

error[E0382]: borrow of moved value: `v`

--> src/main.rs:7:10

```
4 |         let handle = thread::spawn(move || {
    |                                     ----- value moved into closure here
    |                                     v.push(4);
    |                                     - variable moved due to use in closure
6 |     });
7 |     dbg!(&v);
    |         ^^ value borrowed here after move
```

Borrow checker in thread

無論是在 single-threaded 還是 multi-threaded 的環境之下：

The Rust compiler still prevents all data races.

Send and Sync Traits

除此之外，Rust 還定義了兩個 marker trait Send 與 Sync，有以下特色：

- 放在 `std::marker` 底下，表示這只是給編譯器看的標註，不需手動實作。
- 編譯器會自動推導，決定何時替你的型別添加它們，絕大部分不需要手動添加。

這兩個 trait 決定了型別能否跨執行緒溝通，而 borrow checker 則決定了跨執行緒的變數會不會造成 Race condition。這兩個 feature 共同打造 Rust 多執行緒程式的安全區。

Send

1. 可以在 thread 之間轉移所有權（move ownership）的型別。
2. 如果可行，編譯器會自動推導實作。
3. non-Send 的例子是 Rc
 - Rc 在 thread 間傳遞會呼叫 `Rc::clone`，造成引用計數增加，但引用計數並沒有使用原子操作，在多執行緒環境恐有誤。

```
impl<T: ?Sized> !marker::Send for Rc<T> {}
```

Sync

1. 可以在 thread 之間共享（borrowing）的型別。
2. 承上，若 T 是 Sync，則 &T 就是 Send。
3. 如果可行，編譯器會自動推導實作。
4. non-Sync 的例子是 Rc、Cell、RefCell
 - 具有「interior mutability」但不使用原子操作的型別。
 - Rc 引用計數並沒有使用原子操作，在多執行緒執行恐有誤。
 - Cell、RefCell 寫入不遵守原子操作。

```
impl<T: ?Sized> !marker::Sync for Rc<T> {}  
  
impl<T: ?Sized> !Sync for Cell<T> {}  
  
impl<T: ?Sized> !Sync for RefCell<T> {}
```

Send and Sync Takeaways

- 一般開發者知道這些 trait 的型別可在執行緒間爽用即可，編譯器會自動推導。
- Concurrency library 的開發者才需要注意實作 Send Sync
- 介接 FFI 時可能要注意型別是否符合，不行就要自己加類似的實作：

```
unsafe impl<T: ?Sized> !Send for MyExternalType {}
```

Synchronization Primitives

(a.k.a Atomic Type)

- 和 C++ 一樣，Rust 定義了許多基礎型別的 Atomic 版本，
- 比 Rust std 提供的鎖更有效地共享記憶體溝通。
- AtomicUsize，AtomicBool 等 Atomic- 開頭的型別都支援原子操作，
- Atomic operation：透過[編譯器（llvm）定義的 memory model](#)，產生特殊的 CPU 原子操作的指令，榨乾 CPU 最後一點性能。
- [查看更多](#)

Synchronization Primitives

一個簡單的自旋鎖🔒

```
use std::sync::Arc;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::thread;

fn main() {
    let spinlock = Arc::new(AtomicUsize::new(1));



    let spinlock_clone = spinlock.clone();
    let thread = thread::spawn(move || {
        spinlock_clone.store(0, Ordering::SeqCst);
    });

    // Wait for the other thread to release the lock
    while spinlock.load(Ordering::SeqCst) != 0 {}

    if let Err(panic) = thread.join() {
        println!("Thread had an error: {:?}", panic);
    }
}
```

Higher-level Synchronization Objects

Rust 還提供較高階（但仍然很底層）的執行緒同步操作型別，由於內部型別不需要是 Atomic，所以有較高的 runtime overhead。

- Arc：Rc 引用計數的原子操作版本。
- Mutex： 互斥鎖，透過 lock 方法取得內部型別，drop 時自動 unlock。
- RwLock： 讀寫鎖，和 Mutex 類似，不過讀與寫各有 read write 來處理，多讀取單寫入。
- Barrier：雖然叫路障，可以想像成餐廳訂位，人數達到才會放行（執行到 wait 處的執行緒超過定量就放行）。
- Condvar：阻塞執行緒但不耗 CPU 資源的條件變量，通常和 Mutex 成對出現。
- Once：用來確保全域初始化等操作只會執行唯一一次，例如 singleton initialization。
- thread_local!：一個 macro，透過它產生的變數在每個執行緒中都是互不干擾的一份 copy。
- mpsc：透過 channel 以 FIFO queue 傳送訊息溝通，而非共享記憶體，是這些型別之中較人性化的 API，之後會詳細介紹。

Arc + Mutex

如果一個單執行緒的 `Rc<RefCell<T>>` 要改寫支援多執行緒，定義為

- `Arc<Mutex<T>>` 或是
- `Arc<RwLock<T>>`

是最容易的方式。

```
use std::sync::{Arc, Mutex};
use std::thread;

let mutex = Arc::new(Mutex::new(0));
let c_mutex = mutex.clone();

let join_handler = thread::spawn(move || {
    *c_mutex.lock().unwrap() = 10;
});
join_handler.join().expect("thread::spawn failed");
assert_eq!(*mutex.lock().unwrap(), 10);
```


Patterns of Concurrent Programming



Patterns of Concurrent Programming

Concurrent programming 有很多種設計模式與典範，前面介紹的 Synchronization primitives 都是基於 shared-state，雖然好理解，但隨著系統擴張，各種狀態往往複雜到難以管理，因此需要抽象程度更高的設計模式，將狀態與運算解耦合。

接下來介紹常見的並行程式設計模式。

Asynchronous Tasks

又叫做 future 或是 promise，運作邏輯如下：

1. 提供一個 proxy entity (task)，將運算與它最終的結果解耦合。
2. 當 task 發送出去後，會直接返回，讓 caller 繼續往下執行（和結果解耦）。
3. 運算結果一出爐，這個 proxy 可以被 poll 拉取出該結果。
4. 如果結果尚未揭曉，proxy 可以阻塞，或是提供通知不阻塞（依實作而定）。

通常這些 task 的運算會在另一個執行緒中調度，讓最初的執行緒持續往下運作。

- Actor-based system 中的 actor 通常就是一個類似 future proxy。
- Event-driven system 中的 event 也是類似 future proxy 的存在，運算結果被 callback handler 處理。

Rust 最近如火如荼在討論的 feature 之一就是 [future](#) 與 [async/await](#)。

Actor-based

<https://github.com/actix/actix>

Channels and Message Passing

- Channel 兩端分別是 Sender 和 Receiver，互相透過 channel 傳送訊息。
- 可以是雙工，或是單工。
- Rust 的 `std::sync::mpsc` 實際上是一個單工的 FIFO queue channel，多個 Sender 傳送訊息給 Receiver。
- mpsc 建立兩個 channel，透過[傳遞 sender 實現雙向溝通](#)。

Communicate by sharing (clones of) your Sender, and keep the Receiver to yourself.

Event-driven

<https://tokio.rs/>

Coroutines

- Coroutines 可以視為 subroutine 的一般化。
- subroutine 通常是循序執行，而 coroutine 可以暫停（suspend），並從暫停的程式位置恢復（resume）執行。
- 很適合 cooperative task handling。
- Coroutines 通常被當作 low-level primitives，可以用來構建 event-driven 或 actor-based system。
- 知名的 [greenlet](#) Python coroutine framework 就是 [gevent](#) event loop 的低層實作。
- 而 [May](#) 則是 Rust 的 Stackful Coroutine Library。

Random Good Stuff

- [Rayon: A data parallelism library for Rust](#)
- [crossbeam: Lock-free concurrent programming](#)

References

- Erb, B. (2012). [Concurrent programming for scalable web architectures](#).
- Factotum, P. (2019). [Rust concurrency patterns: communicate by sharing your Sender](#).
- Documentation of Rust Standard Library 1.32.0 (9fda7c223 2019-01-16).

AMA Time!

We are from

 Hahow 好學校

Ask us anything!