- 查看更多文章請點擊此跳轉到博客目錄頁面 *點擊此查看html網頁格式　pdf格式*

推薦文章<<心學新解>>

這篇文章比我在這裏分享的任何代碼和創業項目都重要，其中的發現關係到每一個人的方方面面。哲學比科學和技術更重要！哲學是人生，科學和技術只是喫飯而已！

心智是可以被操控的！心智是可以被操控的！心智是可以被操控的！你所不知道的5G/6G微波腦機接口技術！

點擊下面鏈接訪問此文章

- https://github.com/brianwchh/worldofheart

智能指針box<T>

你是否也會問，何謂智能指針？慾答此問，應先知rust之所有權機制，及其設計動機！

rust語言設計宗旨之一，就是在 ***變量離開作用區域時，自動刪除其在內存上申請的空間*** ，而且是在compilation階段確保這些變量的內存會被自動清除！不需要自己寫代碼手動去刪除，又沒有像Go語言那種背景線程去檢查和回收的機制，rust是既要C的效率，又不想要有田螺姑娘在背後運行佔用資源，請問，RUST是如何做到的呢！？

其答案就是compiler在編譯的時候，自動幫你生成了這些刪除內存的代碼。在C++/C中，我們總是會出現申請了，卻忘記刪除的情況，因爲寫着寫着，人都難免會忘記，也因此，在golang和其他一些高級語言中有了一個defer語句，就是說讓你申請完一段內存，馬上寫一個刪除該內存的代碼，但在這代碼之前寫上defer，就是告訴編譯器：親愛的，把這個命令在生成彙編assemble指令時，放在該作用域（比如函數）退出之前。這種除了用於刪除變量，也常用在退出某個線程thread程序中。

但是，這麼高級的defer語句，RUST是唔憂啲。咁哩啲話，rust有咩呢？ 它擔心有些懶鬼，連defer都會忘記。所以它就設計了一個飛常嚴格的變量所有權機制。它從每個變量的生成到結束一直在compile階段就排查，在這個變量退出作用域時，就幫你插入一段刪除該變量的程序。

如此，你就可以放心地申請內存，反正最後這個變量都會退出作用域，compile會幫你生成代碼，一一刪除內存。

這種排查機制，又如何工作的呢？

其原則也不難，就是保證塊heap內存某一時刻只有一個所有者，如果這個所有者在退出某作用域時，沒有將所有權轉交給別人，那麼其他所有想訪問該內存的變量都時無效的，你只要訪問了，編譯器就會報錯，告訴你，所有權在哪裏失去了，這樣你就可以去該處將所有權先轉交給需要的繼承人。所以，在heap上的一塊內存，在stack上可能有多個指針，但每一時刻，只有一個指針是有擁有權的，其他的指針，在這擁有者還活着的時候，能讀，也可以寫，只是它們在退出各自作用域的時候，不會刪除該變量。

因此這裏，又引入了一個作用域的 ***生命週期*** ！即上面提到的，你必須要保證，在多線程的時候，有所有權的指針生命週期要比沒有所有權的指針，活得長！因爲在多線程的時候，compiler是沒辦法知道哪個子線程的退出

時間比較早的，因此，這個時候，所有權應該給主線程的變量，在主線程退出的時候，確保所有子線程都退出了，再清除該內存。如果你把所有權給某個線程，在程序運行的時候，如果那個線程在退出之時就把內存清除了，另一個線程就會奔潰了，因爲訪問不到該內存了。

在回到Box<T>。

- ## referencing & borrowing

  感覺這Rust發明了好多亂七八糟的名稱，明明可以沿用C語言的詞彙，華華非要搞個borrowing，找了半天還是reference，中的一種（即只讀）。

  用一個栗子來說明。

```
    let mut s1 = String::from("hello");

    let s2 = & s1 ;  // read only referencing and
                     //s1 pointer is freezed untill s2 is destroyed.
```

  以上是只讀的referencing

```
    let mut s1 = String::from("hello");
    let s2 = & mut s1 ;  // read & write referencing and
                         // s1 pointer is freezed untill s2 is
destroyed.
```

  不管是只讀還是讀寫的referencing,s2只要還在其作用域，s1都將無法讀寫訪問，雖然s1仍然擁有所有權。問題是，多線程時如何處理？這搞定似乎有點複雜了？

  舉一個栗子：

```
fn print_type_of<T>(_: &T) {
    println!("{}", std::any::type_name::<T>())
}

fn main() {

    let mut s1 = String::from("hello");

    let s2 = &s1 ;   // s2 只是只讀reference

    s1.push('a');   // 無法進行操作，因爲s2尚未退出作用域，後面還用到了s2

    print_type_of(&s2);
}
```

```
error[E0502]: cannot borrow `s1` as mutable because it is also borrowed as immutable
  --> box.rs:13:5
   |
11 |     let s2 = &s1 ;   // s2 只是只讀reference
   |              --- immutable borrow occurs here
12 |
13 |     s1.push('a');   // 無法進行操作，因爲s2尚未退出作用域，後面還用到了s2
..
   |     ^^^^^^^^^^^^ mutable borrow occurs here
14 |
15 |     print_type_of(&s2);
   |                   --- immutable borrow later used here

error: aborting due to previous error

For more information about this error, try `rustc --explain E0502`.
```

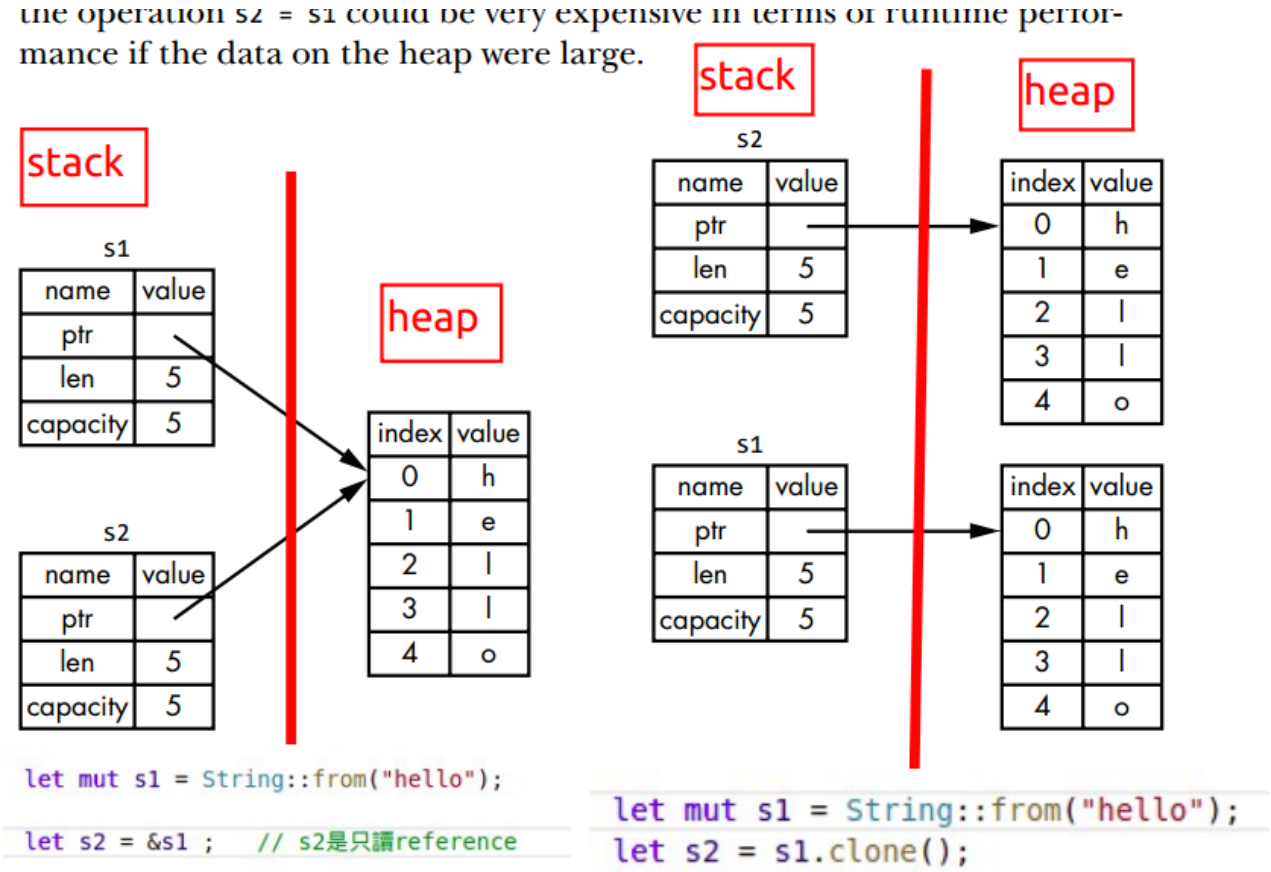

圖1

- referencing 與 raw pointer （* const / * mut)的區別

從上圖可以看出，reference其實就是將stack部分的數值（meta data/描述信息）拷貝了一份，命名爲s2。然後s1和s2都指向heap上的同一塊內存。這就是語句（let s2 = &s1 //s2只讀 ； 或者let s2 = & mut s1 ; //s2 可寫 ）所做的事情。

而raw pointer的值就是其指向的變量首地址。還是以上圖1爲栗子。

```
let raw_ptr_of_s1 = &s1 ; // &s1可以用C語言中的取s1變量的首地址來理解。&在此爲取地址操作符號。
```

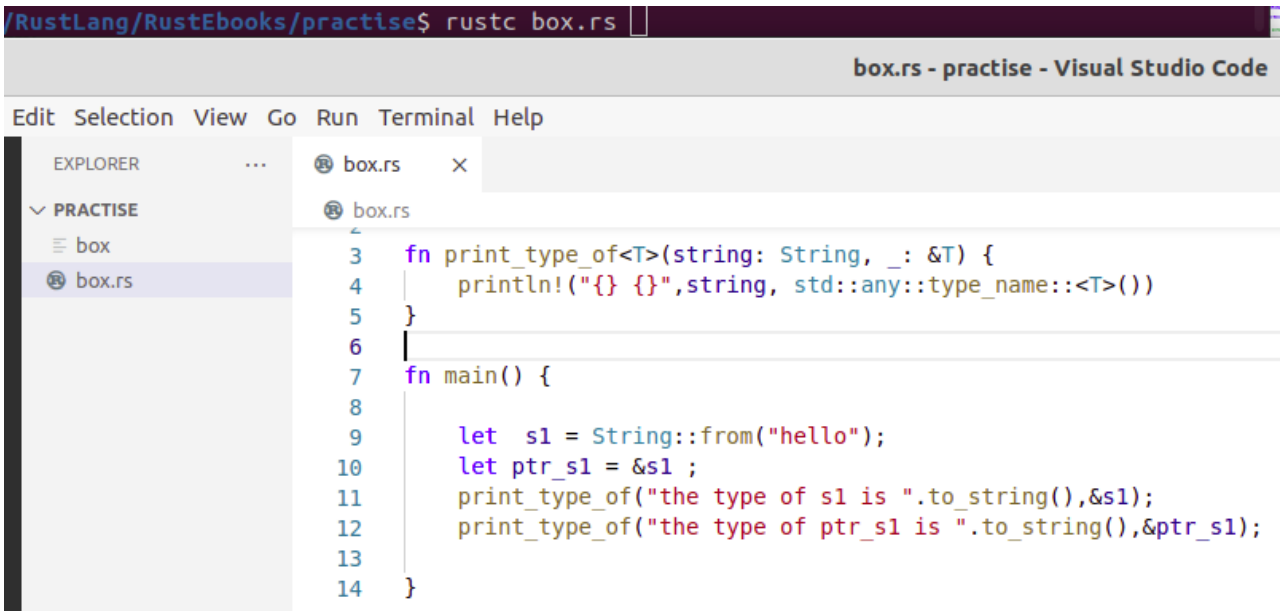

圖2

如果我們打印其類型，compiler自動infer的類型爲如下：

```
the type of s1 is  alloc::string::String
the type of ptr_s1 is  &alloc::string::String
```

圖3

以一張圖來幫助理解raw pointer和reference在stack和heap上的存放方式：

```
  <!-- image area, flex to make it center,it may not work for github,
for html and pdf rendering only -->
```

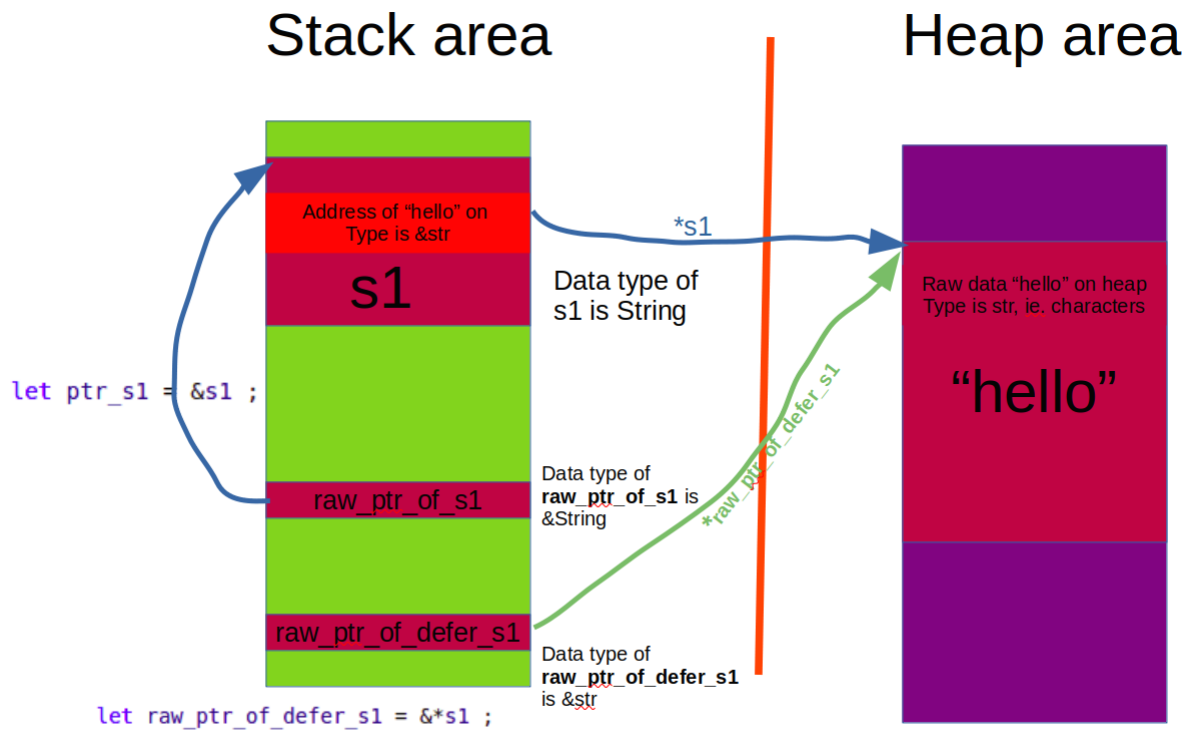

圖4

現在我們來寫一個簡單的程序來驗證下上圖：

- move, 轉移所有權

```
let mut s1 = String::from("hello");

let s2 = s1 ;
```

s1 將無效，當s2退出作用區域後，heap上的字符串hello空間將會被無效清除。除非在s2退出作用域之前將 s1=s2 move回給s1.

- 什麼情況下用Box<T>

簡單來說，就是當你有一段數據，尤其是大小在compiler階段無法確定的數據，這種情況，一般都要把raw data放在heap上，然後在stack上生成像描述信息，如上面提到的String類型。

Box是一種struct，好處是，自己可以添加trait的方法，比如 let GetValue = * some_box_instance ; // dereference 的方法。

當我們稱之爲智能指針時，一般都是說它能夠在退出作用域時，自動刪除heap上的內存（當然還有stack上的）。Box實現這種自動刪除的智能方式就是通過自定義drop方法，在退出作用域時，會自動調用這個drop方法，而我們則需要在這drop方法里寫上刪除某個變量的語句。