

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

- [無眠月照無情門. 失去自由的歌手](#) [點擊此前往github在線閱讀]

本地模式 [html網頁版](#) [pdf版本](#)

- 心学新解：<https://github.com/brianwchh/worldofheart>

本地模式 [html網頁版](#) [pdf版本](#)

智能指針box<T>

阿柄

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，因為寫着寫着，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是沒有喲。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！因為在多線程的時候，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，因為訪問不到該內存了。

在介紹box之前，先複習下move,clone,pointer和reference。

- move, 轉移所有權

用一張圖來解釋move（淺層拷貝）和clone之深層拷貝。

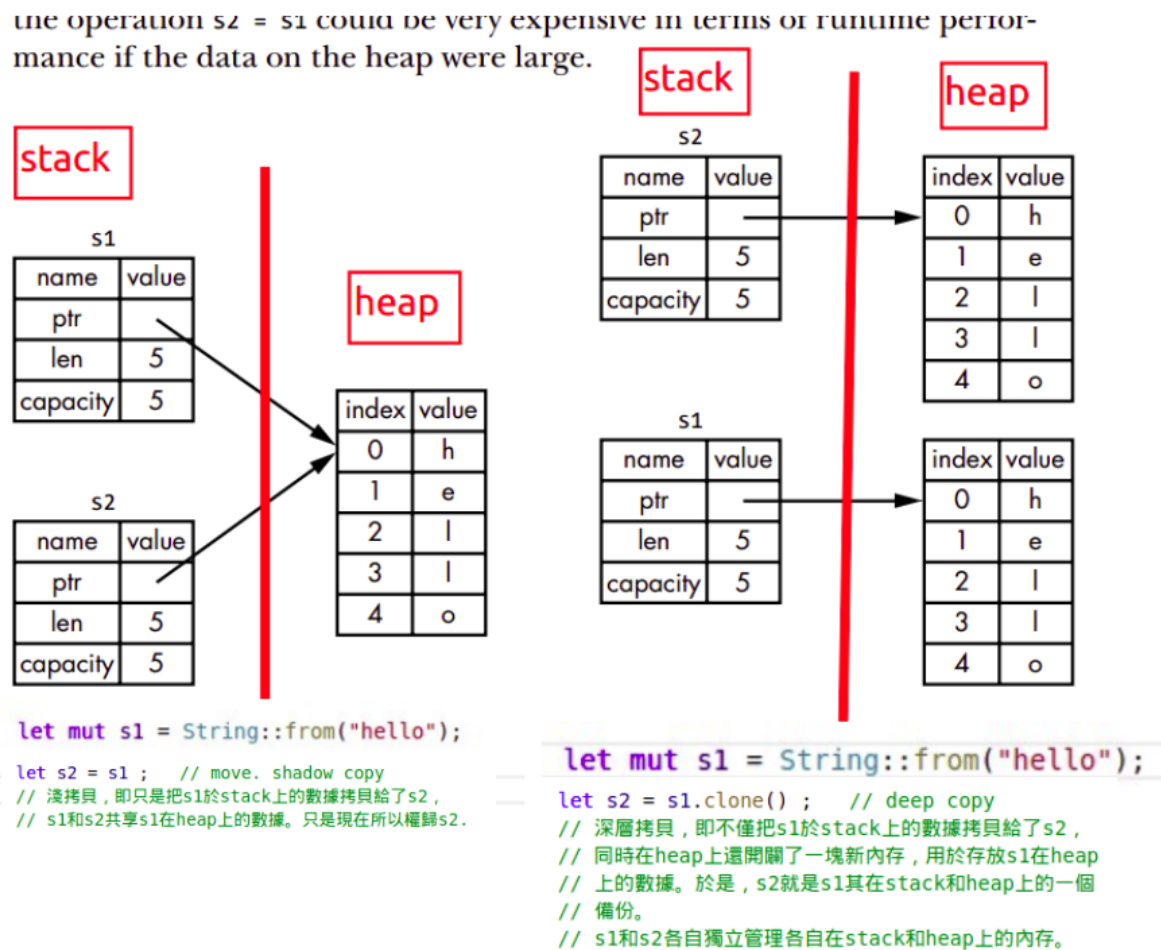


圖1

具體解釋看上圖1中的文字。

- 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?

舉一個栗子來說明變量被referenced的時候,無法使用,只有在reference完成之後,才能使用:

```
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,因為s2尚未退出作用域,後面還用到了s2

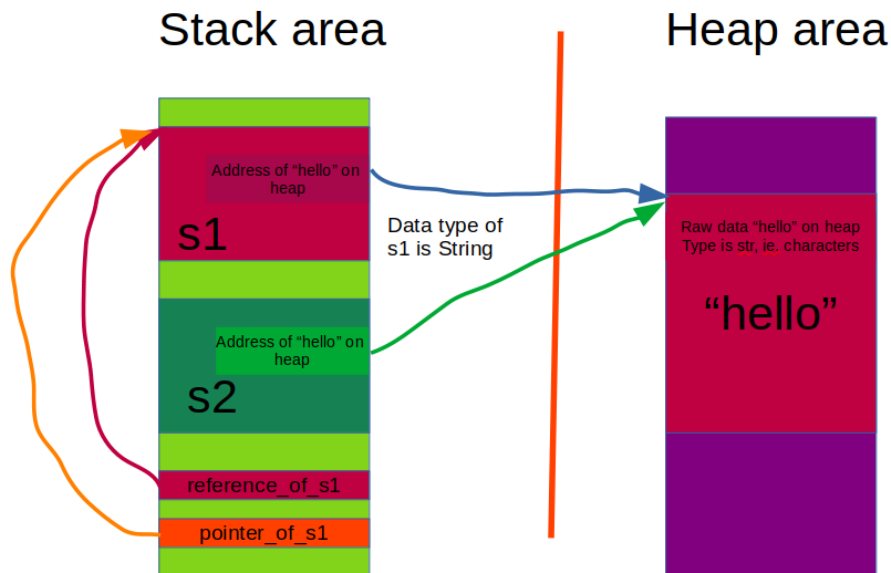
    print_type_of(&s2);
}
```

```
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,因為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`.
```

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



```
let s2 = s1 ; // move,淺層拷貝,只發生在stack上,heap數據共享
let reference_of_s1 = &s1; // reference
```

```
let pointer_of_s1 = &s1 as * const String ;
// or
// let pointer_of_s1 : * const String = &s1 ;
```

注意 move 和 reference 的區別

圖4

從上圖4可以看出，reference同move的區別。本質上圖中reference_of_s1和raw pointer功能其實一樣，只是reference_pointer有compiler參與內存管理，在被s2 referenced階段，無法對s1進行操作。s2和指針一樣，指向s1的首地址。

而

```
let pointer_of_s1 = &s1 as * const String ;
```

注意上面的語法，compiler是如何生成raw pointer的。與reference的區別是，s1在被pointer_of_s1指向的階段，依然可以操作。這裏就需要自己管理內存，也是被compiler認為不安全的代碼區段，你需要自己非常注意管理內存。(後面有更詳細介紹)

這裏吐曹下rust reference語法之怪！在賦值的時候，居然和pointer一樣的使用，即 * reference_of_s1 = something...,與 * pointer_of_s1 = something 的dereference是一樣的。用過opencv C++代碼的人，或許對Mat& reference_of_Aimage = instance_of_Aimage ;與Mat * pointer_of_Aimage = &instance_of_Aimage ;應有所瞭解。在C++中，reference和pointer在本質上是一樣，只是方便語法，有人認為 * pointer_of_Aimage是C語言的寫代碼模式，應盡量不使用pointer，因此可能有了reference，本質還pointer，只是在訪問class/object成員時，不需要像pointer那樣，因為pointer的是用pointer_of_Aimage->memberA, pointer_of_Aimage->methodA(),而在opencv的庫裏面，我們大部分代碼是instance_of_Aimage.methodA(), 或instance_of_Aimage.memberA, 在復用代碼的時候，你如果用pass value by pointer，在復用的函數內，就像逐個把"."換成"->",代碼一多，就顯得很麻煩，而用reference則可以繼續使用reference_of_Aimage.memberA,reference_of_Aimage.methodA().而rust則使用* pointer_of_Aimage.memberA 和 * reference_of_Aimage.memberA一樣的語法方式。這裏自己就要自己記住某個變量是reference還是pointer，雖然本質上兩者都是指針。只是在rust里，

reference是compiler管理內存，而pointer則是要程序員自己注意管理內存，提供了更多的自定義的靈活度。

通過下面一個小程序可以驗證：reference的優點是，即可以當成是指針一樣，可以dereference，同時也可以在寫代碼時直接當成是取所reference的變量一樣使用，而不需要在前面加上*，比如：*

reference_of_Aimage.memberA 其實是Aimage.memberA，但也也可以

reference_of_Aimage.memberA，這樣直接獲取到memberA，編譯器會轉換成指針的方式，這樣的好處就是和C++一樣，不要總是用*在變量之前。具體請看下面這個栗子：

```
15 fn main() {
16
17     let mut s1 = String::from("hello");
18
19     /*
20      * reference examples
21      */
22     let immutable_reference_of_s1 = &s1 ; //immutable reference
23     println!("{}",immutable_reference_of_s1);
24     println!("{}",*immutable_reference_of_s1); //相當於println!("{}",s1);
25
26     let mutable_reference_of_s1 = & mut s1 ;
27     // (*mutable_reference_of_s1).push('a'); //這個也可以,相當於s1.push(a)
28     mutable_reference_of_s1.push('a'); //這個也行。
29     println!("{}",*mutable_reference_of_s1);
30     println!("{}",mutable_reference_of_s1);//相當於println!("{}",s1);
31
32     /*
33      * pointer examples
34      */
35     let immutable_pointer_of_s1 = &s1 as * const String ;
36     unsafe {
37         println!("{}",*immutable_pointer_of_s1); //即println!("{}",s1)
38         // println!("{}",immutable_pointer_of_s1); // can not use pointer in this way,
39         // use reference instead
40     }
41
42     let mutable_pointer_of_s1 : * mut String = &mut s1 ;
43     // let mutable_pointer_of_s1 = &mut s1 as * mut String ;
44
45     unsafe {
46         (*mutable_pointer_of_s1).push('b');
47         println!("{}",*mutable_pointer_of_s1); //即println!("{}",s1)
48     }
49 }
```

圖5

* **【總結】**：reference和pointer在本質上都是指針，在生成assembly代碼時應該是一樣的，其用reference的目的大概就如上面舉的opencv的栗子一樣，寫高級語言的人可能認為在每個指針變量之前加一個*號比較麻煩，於是在語法上設計出一個用於指代(reference)某個變量（以指針的方式指向其首地址，而避免數據拷貝），方便在函數之間做參數傳遞，而且也方便代碼共享，而不用在pointer，reference和變量本身instance三者之間用不同的語法來訪問變量內部成員和方法，典型的就c++中，如下：

```
int b    = pointer_variable->memberA    ;
int b1   = (*pointer_variable).memberA ; //等同於variable.memberA
int bb   = reference_variable.memberA ;
int bbb  = variable.memberA;
// 由於reference本質上也是指針，所以也可以這麼用：
// bb2 = (*reference_variable).memberA ;等同於variable.memberA
```

由上可以看出，在C++語言中，使用pointer比較麻煩，而使用reference則在代碼書寫形式上都一樣，這樣就非常方便代碼復用。

而Rust的reference變量跟pointer除了有上面類似的區別之外，最關鍵的是，compiler無法對pointer進行內存垃圾回收的檢查。但在自己能保證少量代碼的內存管理的前提下，程序員用pointer又有更多的靈活度。只需要如下把這段代碼用unsafe包含起來

```
unsafe {  
  
    *mutable_pointer_variableA = some_value ;  
  
    // 值得注意的是，此時的variableA變量，在多個地方可以同時讀取和賦值，  
    // 與rust的每個變量某時刻只能有一個所有者矛盾，所以  
    // 無法做垃圾自動回收的內存檢查，需要程序員  
    // 自己小心處理變量的讀寫，要保證該變量不會被其他程序刪除。  
}
```

• 什麼情況下用Box<T>

前面介紹了clone（深層拷貝），move（淺層拷貝），reference和pointer，我們應該對計算機如何存放大小的變量有了個大致的瞭解了。小的變量，如int32，之類的，直接存放在stack上，而數據量比較大的變量，raw data（淨數據）放在heap上，而把描述性的meta data放在stack上，其描述數據類型，大小，在heap上的存放位置等等。我們自定義的struct，其存放也無非就是要告訴compiler如何存放。小的數據我們大概不需要過多去關係，讓compiler的規則自己決定。比如如下

```
// A struct with two fields  
struct Point {  
    x: f32,  
    y: f32,  
}  
  
fn main() {  
  
    // Instantiate a `Point`, Point的實例  
    let point: Point = Point { x: 10.3, y: 0.4 };  
  
}
```

以上小的struct，即使compiler像String那樣分別存放在stack和heap上，也問題不大，全放在stack上也可以。

Box<T>的應用場景

當我們想手動規定compiler將數據存放在heap上該如何實現？在C/C++語言中，可以用malloc（memory allocate）函數，去heap上申請一塊內存。Rust似乎沒有這種底層的接口（API）供我們去操作底層的內存。

雖然沒有malloc這個api，但有一個Box<T>的智能指針。用法如下

```
let boxed_variable: Box<u32> = Box::new(5);
```

以上就是把u32類型的5放在了heap上，然後在stack上生成了一個描述型的智能指針boxed_variable來管理這個heap上開闢的內存，當然，這裏是殺雞用宰牛刀了，只放了一個數5. 只是用來說明box是如何存放數據的。

我們再用一個栗子來說明。

```
fn print_type_of<T>(string: String, _: &T) {
    //不必太糾結此處的語法問題，這個只能是在compile
    //階段才有用的，是跟compiler通信的。
    //在compile完之後，類型已經確定了，寫成str，保存在
    //某個變量中。
    //初學者不要認為是在程序運行階段查詢數據類型。對於機器而言
    //不管是int32還是float，還是其他類型，都是32bit或64bit二進制數而已
    //因此無須糾結此處的語法，我們拿來用就是了，反正我們的目的是為了方便調試
    //用它提供的方法來打印compiler理解的數據類型和我們期望的是否一致。
    println!("{}", string, std::any::type_name::<T>())
}

struct MyPoint {
    x: f32,
    y: f32,
}

fn main() {
    let point: MyPoint = MyPoint { x: 10.3, y: 0.4 };
    let boxed_variable: Box<MyPoint> = Box::new(point);
    print_type_of("the type of boxed_variable is ".to_string(), &boxed_variable); // Box<MyPoint>
    print_type_of("the type of *boxed_variable is ".to_string(), &(*boxed_variable)); // MyPoint
    println!("{}", boxed_variable.x); // 智能指針居然還能當reference用
    println!("{}", (*boxed_variable).x); //dereference之後，相當於 println!("{}", point.x )
}
```

圖6 注意Box智能指針居然也能當reference用，因此在讀寫其成員時，無須要先dereference，直接就能用.的方式

```
the type of boxed_variable is alloc::boxed::Box<box::MyPoint>
the type of *boxed_variable is box::MyPoint
10.3
10.3
```

圖7 運行結果

Stack area

Heap area

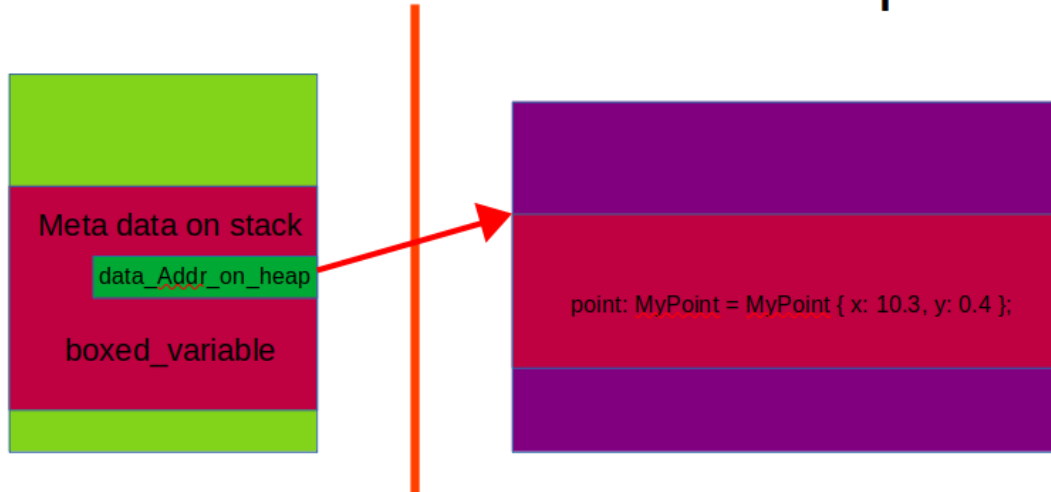


圖8 數據在stack和heap中的存放示意圖

尤其是數據像String或者vector那樣在編譯階段無法確定大小，其數據是在程序運行階段不斷變化的，由用戶決定，這種情況下，只能將數據存放在heap上，這時就需要用Box來處理了。

為何稱之為智能指針中的一種

因其在退出作用域時，會調用其drop()的方法，在drop()的方法裏面去刪除其指向的heap內存數據。

這個就有點類似於C++的deconstruct方法，在class instance（實例）退出函數時，就會自動調用deconstruct來刪除這個instance所申請的heap上的內存空間。

不同之處是，我們調用box的模板，它裏面自動寫好了drop()方法，用於刪除已申請的內存。

而在C++的class中，我們就是 **經常** 忘記寫deconstruct，或者忘記在裏面寫刪除heap內存的代碼。這就是box的一個優點，為偷懶健忘人士寫好了box的模板。

我們寫一個自定義的box來說明下box是如何智能工作的。

以下代碼演示如何讓自定義的MyBox<T>能和rust庫自帶的Box<T>一樣可以同時當作reference和pointer使用，其關鍵就是要為MyBox<T>實現rust庫中的Deref trait 的deref方法。看完下面這個例子，就可以明白這句話了。


```

struct MyBox<T>(T) ;

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x) //如果語法很難懂的話，沒必要去細究，反正這個是寫給
                  //rust compiler的話。按照這個格式來寫即可。
                  //我們只需記住，其內部做的事情，就是在stack生成一個
                  //管理數據T（在heap上）的智能pointer。
    }
}

fn main() {
    let x = 5 ;
    let y = MyBox::new(x);

    print_type_of("the type of y is ".to_string(),&y);

    assert_eq!(5,x);
    assert_eq!(5,*y);
}

```

圖9 自定義box，嘗試做dereference操作，即讀取其所管理的數

圖10 報錯，因為struct沒有dereference的方法，需要額外impl此dereference的trait，具體實現如下圖

```

error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
--> box.rs:35:18
   |
35 |         assert_eq!(5,*y);
   |                        ^^
error: aborting due to previous error

```

圖10 報錯，因為struct沒有dereference的方法，需要額外impl此dereference的trait，具體實現如下圖

*y 應是dereference 智能指針y，來獲得其指向heap上的數據5

```

use std::ops::Deref;
struct MyBox<T>(T) ;

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x) //如果語法很難懂的話，沒必要去細究，反正這個是寫給
                //rust compiler的話。按照這個格式來寫即可。
                //我們只需記住，其內部做的事情，就是在stack生成一個
                //管理數據T（在heap上）的智能pointer。
    }
}

impl<T> Deref for MyBox<T> {
    type Target = T ;

    // deref 即為* dereference的運算符號（一種方法），此處即把此trait的方法之
    // 內功大成的功夫傳授給MyBox，然後 *y 的錯誤就不會再出現了，因為此時MyBox<T>
    // 能自動識別 * 運算符，然後調用此deref方法，因此，*y就等效於 *(&self.0)，即
    // T的實例（instance），此處為5
    fn deref(&self) -> &T {
        &self.0 // 物理意義：返回一個指向此T實例的reference。
    }
}

fn main() {
    let x = 5 ;
    let y = MyBox::new(x);

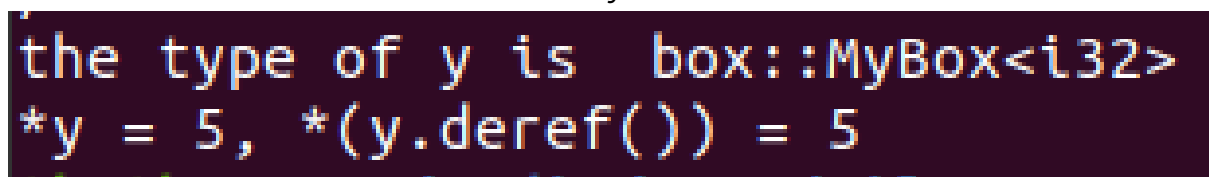
    print_type_of("the type of y is ".to_string(),&y);

    assert_eq!(5,x);
    assert_eq!(5,*y); // 等同於下面這句，compiler自動會幫我們生成如下實現。
    assert_eq!(5,*(&y.deref()));

    println! ("*y = {}, *(&y.deref()) = {}", *y, *(&y.deref()))
}

```

圖11 為MyBox實現



```

the type of y is box::MyBox<i32>
*y = 5, *(&y.deref()) = 5

```

圖12 運行結果

以上實現了如何用MyBox<T>在heap上存放數據，接下來，我們用另一個例子來說明，如何做到和rust系統庫自帶的Box<T>一樣，自動刪除compiler為我們申請的heap內存空間。

```

14 use std::ops::Deref;
15 use std::fmt;
16
17 struct MyBox<T>(T) ;
18
19 impl<T> MyBox<T> {
20     fn new(x: T) -> MyBox<T> {
21         MyBox(x) //如果語法很難懂的話，沒必要去細究，反正這個是寫給
22                 //rust compiler的話。按照這個格式來寫即可。
23                 //我們只需記住，其內部做的事情，就是在stack生成一個
24                 //管理數據T（在heap上）的智能pointer。
25     }
26 }
27
28 impl<T> Drop for MyBox<T> {
29     fn drop(&mut self) {
30         println!("Dropping MyBox with data ! "); //先不去實現T類型的Display trait
31                                                    // 因為泛型T數據的打印無法實現，只能
32                                                    // 針對具體類型數據，比如strut，我們可以
33                                                    // 選擇需要給println這樣的函數顯示什麼數值。
34                                                    // 畜生，大腦是我自己的，別tmd這麼變態，老催眠人，
35                                                    // 讓人無法思考。死變態的。
36     }
37 }
38
39 }
40
41 > impl<T> Deref for MyBox<T> { ...
51 }
52
53 fn main() {
54     let x = 5 ;
55     let y = MyBox::new(x);
56
57 }

```

圖13 實現Drop trait來自動刪除和釋放heap內存



圖14 程序輸出結果，表明drop在main函數退出之前被調用了。

注意，因為rust沒有提供申請和銷毀heap內存的API，此處實現了Drop trait中的drop方法，我們無須在內部自己去刪除某個內存，你想操作，也沒辦法，這裏我們實現了drop方法，但內部怎麼實現drop，那些代碼，rust compiler會添加。除此之外，你還可以加入一些不是刪除heap內存的代碼，如果你需要在MyBox退出之前自己處理一些事情的話，可以在此加入自定義代碼

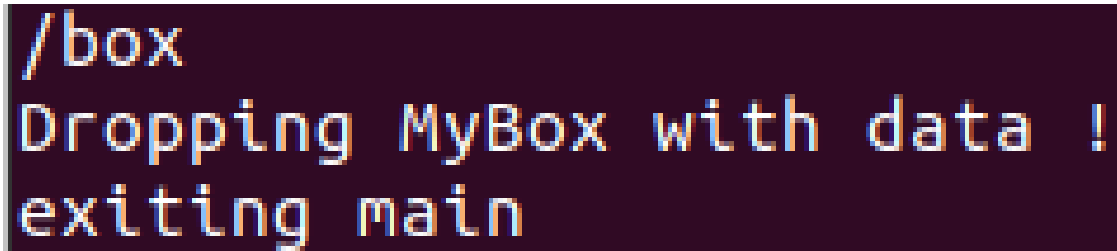
有些變量，你想在程序退出之前，自己手動刪除可否？

當然可以，但不是調用MyBox_instance.drop()，上例子中，即y.drop()。因為編譯器compiler不允許用戶調用Drop trait中的drop方法，因為在程序退出的時候，又會調用drop()方法一次，因為內容之前已經刪除了，在此刪除，會造成程序崩潰！

解決方法是利用系統提供的另一個API函數，即std::mem::drop，調用了這個函數之後，程序在退出的時候，就不會再調用MyBox_instance.drop()，上例子中，即y.drop()。

```
53 fn main() {  
54     let x = 5 ;  
55     let y = MyBox::new(x);  
56     std::mem::drop(y);  
57     println!("exiting main");  
58  
59 }
```

圖15 調用std::mem::drop 方法



```
/box  
Dropping MyBox with data !  
exiting main
```

圖16 輸出結果，可見是先執行了std::mem::drop會調用系統的Drop trait的drop方法，之後才運行println!("exiting main")，之後在main退出時，沒有再次調用Drop trait中的drop方法。這樣就不會出現重複刪除的問題。