trait object

阿柄

用不嚴謹的話概括：trait object就類似與其他語言中的接口(interface)，比如Golang。如果你沒有學過GoLang語言，也沒關係，我在這裏嘗試站在語言設計者的角度來介紹

1. 設計接口interface的動機。
2. 實現原理
3. 注意事項
4. 加dyn這個關鍵字原因

參考文檔: rust-programming-language-steve-klabnik, page 407, Using Trait Objects That Allow for Values of Different Types

因爲個人覺得這本好像是官方推薦的書本關於這塊寫的不容易理解，所以希望寫一篇博客，從自己理解的角度來介紹rust的trait object。希望能幫助初學者形象理解這方面的內容。

1. 設計接口interface的動機，如何在vector里放入不同類型的元素

    - **假設讀者已經理解了stack（棧）和heap（堆）的硬件知識**

      這個是理解Rust語言的關鍵，因爲rust語言的發明動機之一，就是如何在編譯之時防止stack and heap overflow（堆棧內存溢出），其優於C/C++之處就是因爲C/C++項目很大，或團隊成員之間合作時，往往會忘記手動刪除在heap上申請的內存，變成了垃圾碎片，導致電腦或嵌入式系統出現heap內存溢出，Rust的變量所有權機制很聰明地在編譯階段就幫我們排除了這樣的錯誤出現，相較於其他語言，比如有垃圾回收進程的GoLang等等，優點就是不需要額外的進程消耗資源來檢測和回收垃圾內存。這裏只簡單提及和複習下Rust語言發明的動機之一。

    - **發明rust的另一個主要動機 （非必要閱讀小節）**

當然其發明的另一個動機就是，相較於C/C++,在保證性能和其差不多時，又能實現非常方便部署和遷移，可以理解成生產的自動化，不需要像C/C++那樣，手動一個一個去下載依賴包，CMake File雖然比Makefile方便許多，但還是沒法做到一鍵自動化實現生產的環境重現，嘗試過opencv編譯的讀者應該有親身體會。爲了實現像javascript那樣實現nodemoudle的一鍵自動下載和安裝，Rust用Cargo來管理依賴的包，順便提及下Golang的包管理系統是go module，有興趣的可以看下我github上的pdf介紹文章：https://github.com/brianwchh/web3-full-stack-design-tutorial/tree/main/05-go%20module/slice。這裏也不展開，有興趣可以之後去瞭解，和對比這兩種語言的包管理方法，原理上都差不多。不理解這個包管理系統，不方案理解接口(interface)，這裏只是順帶提及下rust語言發明的另一個動機。

- **必要前提，瞭解vector相關知識**

  除了必要的stack和heap知識，希望讀者也瞭解了vector. 這裏簡要回顧下。

  ```
  let v: Vec<i32> = Vec::new();
  //  Creating a new, empty vector to hold values of type i32 ,
  ```

  簡單瞭解下在rust如何定義一個i32，即32bit的integer類型的vector v。

  這裏請注意下，我們在用高級語言時，往往把它當成黑匣子，很多人也不去深究這一條短短的語句後面編譯器幫我們做了多少事情！如果不去深究和瞭解，即，站在底層電腦/嵌入式處理器的角度去看自己寫的代碼，你可能不知道自己寫的代碼，在compile之後會變成怎麼樣。

  很多人也可能都沒寫過C語言，人們發明rust這些高級語言，有一個動機當然是希望程序員能少寫很多代碼，有些代碼可以通過compiler去生成，但不代表我們不要去稍微瞭解下，其背後做了那些工作。

  比如，vector，這是一個大小可變型的內存，可以比較大，真正的數據放在stack肯定是不可能的，stack上只能放指針和描述型數據。這裏用下面一張圖來輔佐理解。

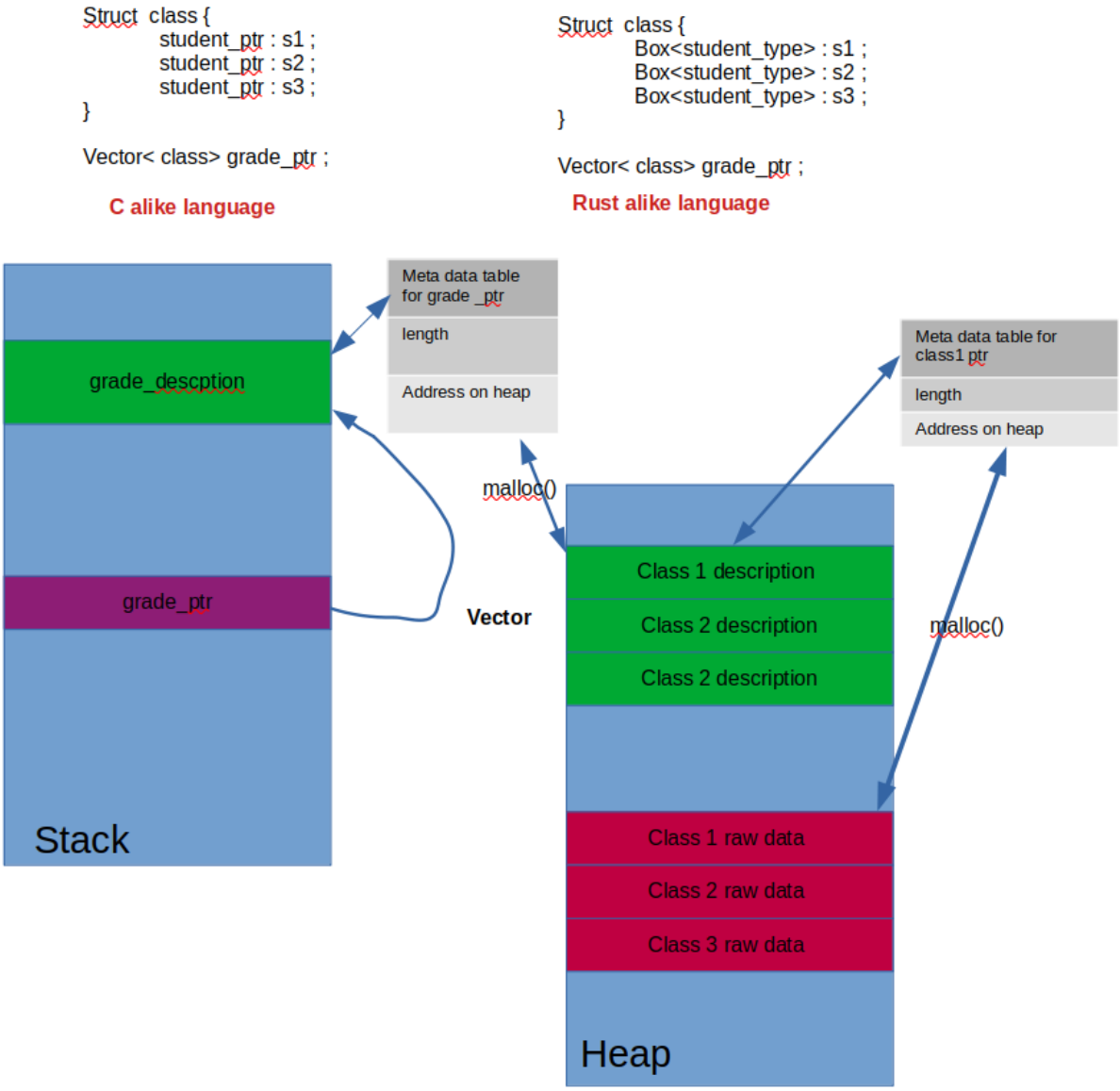

圖1，vector示意圖

暫時先不要去深究Box<Type>,其實也不難，以後會介紹，簡單理解就是獲得一個heap上數據的指針，尤其是當這數據大小在compile階段無法知道時，用Box，其他的可以用其他例如reference和move等方式，這裏不深入介紹Box。放注意力放在我們如何處理在heap上的大數據內存，尤其是像vector這種大小在compile階段無法知道的，即在程序運行階段動態可變的。

在C語言中，或者是彙編assemblly語言中，我們用鏈表的方式來實現vector，其實rust compiler也是這麼幫我們我們自動實現的，在C語言中，自己寫鏈表成員的添加和刪除還是蠻麻煩的，所謂鏈表，就是在heap的一段一段的小內存（vector成員在heap上的數據空間），裏面有指向上一個成員和下一個成員的地址，這樣，就像鏈表一樣，把vector內的各個成員連在一塊了，在查找的時候，就能循着這個鏈條尋邊所有成員。

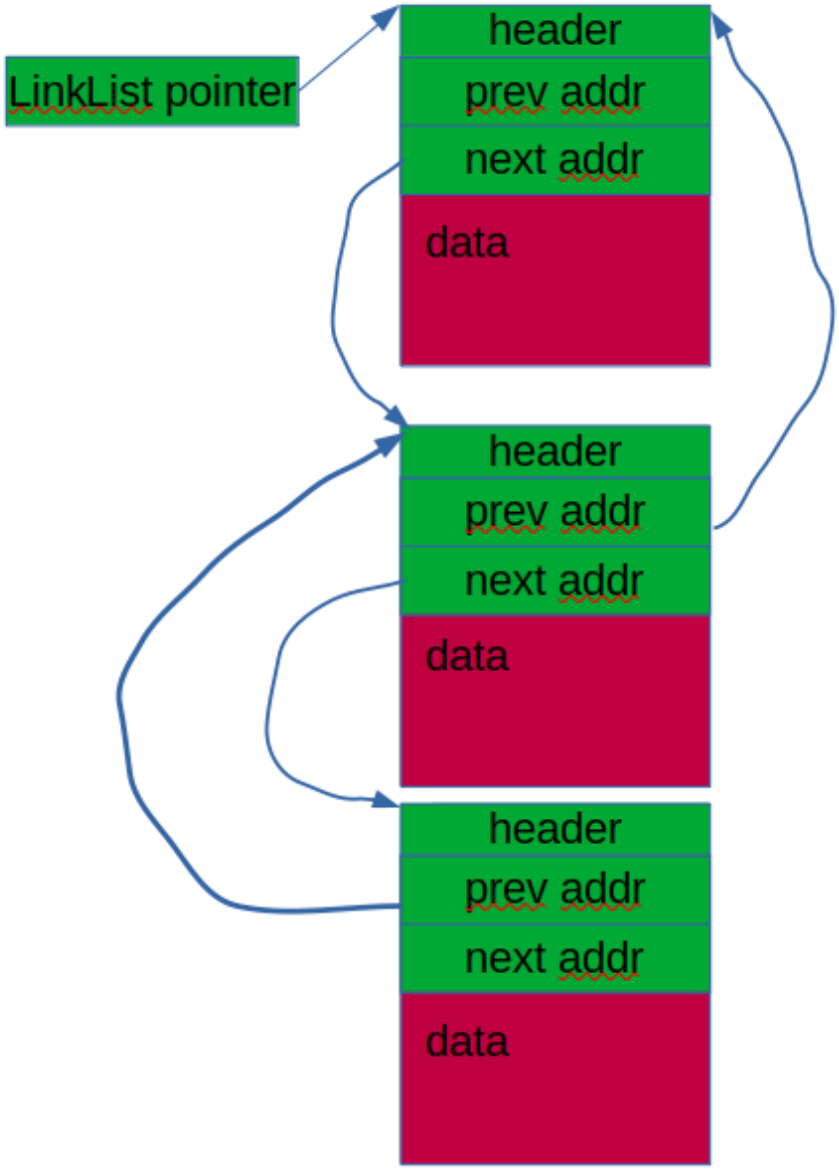

圖2，鏈表意圖

所有注意上圖heap上綠色和紅色部分的內存是不一定連續的，因爲，每次加入一個vector成員時，我們從高級語言用戶的角度，只是一小段語句，就像大老闆動動嘴，如下

```
    v.push(6);
```

comopiler,卻要像C底層程序員那樣，去幫你生成往heap上申請內容的函數，比如c語言的是類似於int* faddr = malloc()，然後返回一個heap上的地址，如果某些原因申請不成功，faddr是負數，我們只要判斷下是否小於0，就知道是否成功了，這就是C語言程序的做法，但在寫程序的時候，我們常常忘記去判斷是否小於0，是人都會犯下這種很難查的失誤，所有常常會出現程序奔潰的很難查的bug。而rust編譯器幫我們生成的底層代碼，都是模塊化的，安全的代碼。

```
    vector<class> grade_ptr
```

注意上面的vecotor里裝的成員class是指針，heap綠色部分，而每個指針指向heap上紅色的raw data。這就是大概數據的存放情況（物理，甚至虛擬內存都可能不連續）。

- **vector<T>缺點，與interface的發明動機**

vector的缺點是，內部成員的所有類型必須一樣，比如vector<int32>, 那麼內部成員必須是全部都是32bit的integer。如何能將不同類型的object放到一個vector裏面呢？

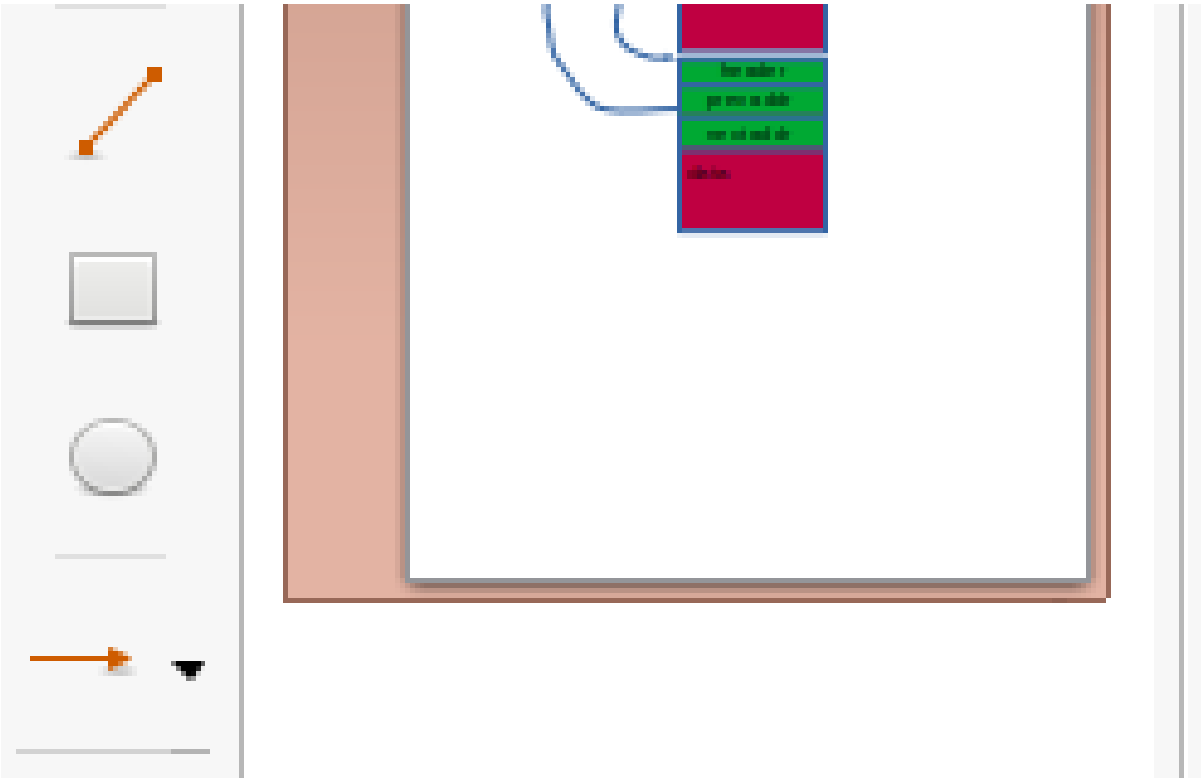

圖3，繪製圖形list in vector

假設我們有一個應用場景，如上圖3所示，我們設計一個繪圖工具，在左邊欄中用一個vector<Shape>把所有可以畫的圖形放在左邊，而且還能讓客戶自定義圖形，放入該欄中。

這樣自己定義不同的shape struct / class / object 放入vector中，編譯器肯定會報錯的，因爲類型不一樣，即使是指向該object的指針，夜不行！雖然可能都是32bit或64bit的2進制數，但編譯器是理解數據類型的。如何解決呢？ 可否強制轉化成像c語言那樣void*類型的忘能指針呢？但這樣放進去的時候，似乎可以騙過編譯器，說你看，這些成員在內存中的長度都一樣，類型都一樣，不就是32bit嘛。編譯器說：ok！但你在自己調用的時候，還知道哪個是哪個嗎？難道要自己去記住各個順序分別放了什麼類型的指針，然後再強制轉換回來？要知道，這可是動態的，是在程序運行時要做的，而不是你寫程序的時候，靜態預測的。所以這樣的方法顯然能騙得了編譯器，卻弄暈了自己。當然你也可以不轉換回原來的類型，如果你能自己知道各個object成員在內存在的地址的話，你可以和assemble一樣，自己用地址去訪問這些成員，只是你可以想像這複雜程度。

山不轉，水轉！

上面的思想實驗給我們指了條明路。我不強制轉換成萬能指針void*，況且rust裏面也似乎沒有這種萬能指針，倒是goLang裏面用interface{}的另一個用途來實現了類似c語言中的void*，這不在這裏展開，這是golang獨有的interface功能。這裏我們介紹interface的主要用途，也就是所有高級語言中，如有interface，***其目的和用途都是一樣的，就是爲了解決如何在vector里放入不同類型的元素***。

2. interface係咩

那我們能不能發明創造一個一種指針，用於指向一些不同類型的object，class，或struct，相當於在語法層面告訴compiler說，這些不同類型的成員是我的同族，我可以用這種指針去指向它們，其實嘛就是32bit或64bit的2進制數嘛，只是高級語言嘛，都要語法規則，那我們就創造這種規則，然後在vector<interface_pointer>中把這些同族成員放進去。然後在調用時，就可以用interface_pointer.shape_type,或者interface_pointer.width,又或者調用其方法interface_pointer.draw_shape(),如此等等遍歷所有成員和方法，然後顯示在GUI（界面）上。而我們在寫程序用每個成員時，先判斷具體是什麼類型的，這樣就不會調用到不存在的變量或方法，兒導致程序出現動態的錯誤，並且，這一段程序，我們必須告訴compiler，我們自己像寫C語言一樣，自己搞定heap內存變量的管理，因爲compiler沒法管理動態的存儲分配。

ok，思路有了，無法就是搞一套語法，和compiler商量好，那32bit或64bit二進制數是interface的指針，然後要compiler去生成訪問成員和方法的assembly code.

接下來，我們來講講rust是如何實現interface這功能的，不像golang和其他高級語言，有關鍵詞interface。rust是用trait來實現的，即稱之爲trait object,就跟trait的名字一樣怪，背了一輩子的英語單詞，背不完的單詞，一輩子看文章像是離不開英語字典這個尿不溼一樣，我也懶得去查trait的單詞意思了，哈哈。我下面就用人話解釋下咩係trait！

我們知道，不知道你知不知道？rust的class定義是成員（member）和方法（method）分開的。其也不叫class，而是struct，閣下可以自行查下rust是如何用impl關鍵字來隱式爲某個struct定義方法（method，以下不再重複寫了，方法即method，海峽兩岸五湖四海大家應該都是這麼稱呼C++中的method爲方法的吧？）的，如果多個struct共享一些方法，可以用trait關鍵字把它們包起來，如果不做interface的功能，有沒必要包起來組織起來給一個trait的別名，我不知道，大概也許可能用一個別名方便代碼閱讀和管理吧。

如果你要實現interface pointer的功能，必須要綁起來的。然後實現了這個trait內部**_全部_**方法的struct們就是這個trait的同族，可以用這個trait object來做指針，賦值啊，或傳遞如別的函數的調用中。當然也解決了“ ***如何將不同類型的struct放入vector<T>中的世紀難題*** ”。

3. 我舉一個栗子

下面用一個栗子來解釋下上面的人話。

```
 pub trait Draw {
    fn draw(&self);
    } // 這裏定義了一個Draw的組件interface，方法只有draw()

pub struct Button {
        pub width: u32,
        pub height: u32,
        pub label: String,
    }  // 這裏定義一個Button的組件

impl Draw for Button {
    fn draw(&self) {
        // code to actually draw a button
    }
    } // Button組件實現Draw組件interface的的draw()方法，
```

```
        // 然後就成了同類。就可以被Draw object指針指向。

struct SelectBox {
        width: u32,
        height: u32,
        options: Vec<String>,
    }  // 這裏定義一個SelectBox的組件

impl Draw for SelectBox {
    fn draw(&self) {
        // code to actually draw a select box
    }
    }// SelectBox組件實現Draw組件interface的的draw()方法，
     //然後就成了同類。就可以被Draw object指針指向。

pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
    } // 在screen中用Draw object指針把button
      // 和select組件一起放進components的vector中。
      // Box<T>是將數據類型爲T的變量實例放在heap上，並在stack上
      // 生成一個指向之的智能指針
      // 此處dyn就是告訴編譯器，Draw的類型是object safe的
      // dynamically dispatched trait。
      // a dyn Trait reference 其成員只有兩個指針，一個指向
      // 被動態分配的struct instance，另一個指向所有方法的總表頭。
      // 也因此，dyn trait object可以作爲interface的指針指向
      // 其同族的struct instance（實例）。
      // 類比下Box<u32>,eg let stack_pointer_to_8 = Box<8> ;
      // 即可明白dyn Draw是一種data type，而非實例。說得有點羅嗦:)
```

4. 注意事項

    就是在使用的時候，一定要先判斷類型，然後才能調用其成員和方法，要知道，這部分的代碼是我們自己管理內存。不要出現成員或方法不存在，以及內存忘記清除的錯誤。

以上就是從比較容易理解的和不是那麼嚴謹的指針的角度來理解Rust的trait作爲interface時的原理。

注意一點，

```
pub fn notify(item: impl Summary) {
      // item 是參數，是trait object/interface type 的一個instance，即實例

      println!("Breaking news! {}", item.summarize());

   }
```

注意到pub fn notify(item: impl Summary) 函數定義沒有像下面這個vector一樣動態

```
pub struct Screen {
      pub components: Vec<Box<dyn Draw>>,
      // 此處Draw是trait object/interface的type
      // Box<Type>簡單理解就是獲得其中Type元素（在heap）上的地址，就是說
      // Box<Type> 就成了指向該Type實例（instance）在heap上的地址。
      eg :
      let b : Box<int32> = Box::new(5);
      b就成了指向在heap上的5，雖然這個有點誇張，把一個int32數放在heap，
      但可以說明下Box的原理。所以無法就是類似於圖1，理解了，就也能理解很多
      其他的智能指針的內容。
```

加dyn這個關鍵字原因

```
        // dynamically dispatched trait。
        // a dyn Trait reference 其成員只有兩個指針，一個指向
        // 被動態分配的struct instance，另一個指向所有方法的總表頭。
        // 也因此，dyn trait object可以作爲interface的指針指向
        // 其同族的struct instance（實例）。
        // 類比下Box<u32>,eg let stack_pointer_to_8 = Box<8> ;
        // 即可明白dyn Draw是一種data type，而非實例。說得有點羅嗦:)
```

所以概括上面的解釋就是，一個是函數的參數（parameter),在C語言里也腳argment，pub fn notify(item: impl Summary)，即item是impl了Summary這個trait object的是struct。
而pub components: Vec<Box<dyn Draw > >,即components是vector類型，vector成員的類型是Box<dyn Draw>, 即指向了在heap上的trait object類型Struct的實例。
以上兩者做得事情都差不多，就是如何把指針指向heap上的trait object同族的struct，只是在表達意思時，函數參數和變量類型重實現這個trait object指針的 ***語法*** 不一樣。

2022年12月21日