链滴

rust 中的 DST 和 ZST

作者：zqliang

原文链接：https://ld246.com/article/1539826769170

来源网站：链滴

类型的大小在 Rust 中很重要，Sized trait 是 std::marker 模块中的四大特殊 trait 之一。本文主要介
DST 和 ZST。

DST

DST 是 Dynamic Sized Type 的缩写，意思是动态大小类型，表示在编译阶段无法确定大小的类型
在讲这种类型之前，我们先从数组开始谈起。

数组是一个容器，它在一块连续内存空间中，存储了一系列的同样类型的数据。数组中的元素的占用
间大小必须是编译期确定的，数组本身所容纳的元素个数也必须是编译期确定的。如果需要使用变长
容器，可以使用标准库中的Vec / LinkedList等，原始数组类型是不支持动态改变大小的。数组类型
表示方式为[T; n]，T代表元素类型，n代表元素个数。中间用分号隔开。在Rust中，对于两个数组类
，只有元素类型和元素个数都完全相同，这两个数组才是同类型的。示例如下：

```
fn  modify_array(mut arr: [i32; 5]) {
    arr[0] = 100;

   println!("modified array {:?}", arr);
 }

   fn  main() {

     let xs: [i32; 5] = [1, 2, 3, 4, 5];    modify_array(xs);

     println!("origin array {:?}", xs);
   }
```

编译执行，结果为：

modified array [100, 2, 3, 4, 5] origin array [1, 2, 3, 4, 5]

我们可以看到，把数组 xs 作为参数传给一个函数，这个数组并不会退化成一个指针，而是会将这个
组完整拷贝进入这个函数。函数体内对数组的改动，不会影响到外面的数组。

如果我们把数组的长度改变一下，会发现 [i32; 4] 类型的数组和 [i32; 5] 类型的数组是不同的类型，
能赋值。

数组切片

对数组取borrow操作，可以生成一个“数组切片(Slice)”。数组切片对数组没有“所有权”，我们可
把数组切片看做是专门用于指向数组的指针，是对数组的另外一个“视图”。比如，我们有一个数组[T
n]，它的借用指针的类型就是&[T; n]。它可以通过编译器内部魔法，转换为数组切片类型&[T]。数
切片实质上还是指针，它不过是在类型系统中丢弃了编译阶段定长数组类型的长度信息，而将此长度
息存储为运行期的值。示例如下：

```
// 注意参数类型

fn  mut_array(a : &mut [i32]) {    a[2] = 5;

   println!("len {}", a.len()); }



fn  main() {
```

```
  let  mut v :  [i32; 3] = [1,2,3];
  {

     let s : &mut [i32; 3] = &mut v;        mut_array(s);
   }

  println!("{:?}", v);
 }
```

变量v是[i32; 3]类型，变量s是&mut; [i32; 3]类型。它可以自动转换为&mut; [i32]数组切片类型传入数mut_array。在函数内部，通过这个指针，修改了外部的数组v的值。而且我们可以看到，这个 &mt; [i32] 类型的指针，它不仅包含了指向数组的地址信息，还包含了指向数组的长度信息。

那它是怎么实现的呢？原因就在于 &mut; [i32; 3] 和 &mut; [i32] 的内部表示是有区别的。&mut; [i3; 3] 这种指针，就是普通指针，数组长度信息是编译期确定的。&mut; [i32] 这种指针，是“胖指针（at pointer）”，它既可以指向 [i32; 3]，也可以指向 [i32; 4]，还能指向一个数组的某一个部分。示如下：

```
use std::mem::transmute;

use std::mem::size_of;

fn  main() {

  println!("{:?}", size_of::<&[i32; 3]>());

  println!("{:?}", size_of::<&[i32]>());

  let v : [i32; 5] = [1,2,3,4,5];

  let p : &[i32] = &v;[2..4];

  unsafe {

     let (ptr, len) : (usize, isize) = transmute(p);

     println!("{} {}", ptr, len);

     let ptr = ptr as *const  i32;

     for i in  0..len {

        println!("{}", *ptr.offset(i));
      }
   }

}
```

由此可见，对于 &[i32] 型指针，它是普通指针大小的两倍，这也是为什么它叫做“胖指针”的原因它里面同时存储了所指向的地址，以及长度信息。所以它避免了C/C++里面出现的，数组作为函数参的时候，退化为裸指针的问题。

Sized

为什么Rust编译器会把 &[i32] 这种类型的指针当成胖指针处理呢？因为在Rust眼里，[i32]也是一个理的类型。它代表由 i32 类型组成的数组，然而长度在编译阶段不确定。对于编译阶段大小不定的类，Rust将其称之为 Dynamic Sized Type。我们不能直接声明 DST 类型的变量绑定，因为编译器根没办法知道，怎么为它分配内存。但是，指向这种类型的指针是可以存在的，因为指针的大小是固定。

Rust中有一个重要的 trait Sized，可以用于区分一个类型是不是 DST。所有的 DST 类型都不满足 Sized 约束。我们可以在泛型约束中使用 Sized、!Sized、?Sized 三种写法。其中 T:Sized 代表类型必须编译期确定大小的，T:!Sized 代表类型必须是编译期不确定大小的，T:?Sized 代表以上两种情况都可。在泛型代码中，泛型类型参数默认携带了 Sized 约束，因为这是最普遍最常见的情况。如果我们希这个泛型参数也可以支持 DST 类型，那么就应该为它专门加上 ?Sized 约束，示例如下：

```
use std::fmt::Debug;

fn  call(p : &T;)

  where T:Debug {

  println!("{}", p); }

fn  main() {

  let x : &[i32] = &[1,2,3,4];    call(x); }
```

以上写法，等同于默认有一个 T:Sized 约束。当参数是 &[i32] 类型的时候，编译器推理出来泛型参是 [i32]，不符合 Sized 约束，就会报错。修复方案是，加上 T: ?Sized 约束：

```
use std::fmt::Debug;

fn  call(p : &T;)

  where T: Debug {

  println!("{:?}", p); }

fn  main() {

  let x : &[i32] = &[1,2,3,4];    call(x);
 }
```

如果我没记错的话，这个 ?Sized 表示法，是知乎网友 @Liigo 提出来的建议。

直接在语言中加入对 DST 的支持是有好处的。虽然这种类型无法直接实例化，但是可以被用在 impl，以及泛型代码中。比如，我们可以为 [i32] 类型 impl 一个 trait。再比如， Rc<[i32]> 也是一个合的类型。我们为 [i32] 类型添加的方法，自然而然就可以被 Rc<[i32]> 使用。

Rust 中的 str 类型也是一种典型的 DST 类型。它跟不定长数组是一样的，它内部就是一个 u8 类型不定长数组。&str;也是一个胖指针，跟数组切片一模一样。还有一种常见的 DST 类型就是 trait。trait 仅仅规定了类型需要实现的方法，而对具体类型的大小没有限制，因此实现同一个 trait 的具体类型小是不定的，所以我们不能直接声明 trait 类型的变量。同理，把 trait 放到指针后面是合法的。此时指针也是胖指针，其中包含了指向真实数据结构的指针以及指向虚函数表的 vtable 指针。这种胖指，也叫做 trait object，在后面讲解泛型和动态分派的时候再详细介绍。

DST 的故事到现在为止还没有结束。目前编译器只支持上面介绍的这几种固定的 DST 及其对应的胖针类型。按照Rust设计者的想法，用户应该有权自定义自己的 DST 类型以及各种智能指针类型。只

过这些问题目前不是很紧急，以后再来慢慢设计。

ZST

Rust 还支持 0 大小类型（Zero Sized Type）。比如，在前面的文章中提到过的 () 类型和空结构体型，都是 0 大小类型。示例如下：

```
use std::mem::size_of;

fn  main() {

  println!("{}", size_of::<()>());

  println!("{}", size_of::<[(); 100]>());

  let boxed_unit = Box::new(());

  println!("{:p}", boxed_unit); }
```

执行结果为：

0 0 0x1

由此可见，unit 类型确实是 0 大小的类型，而且由它组成的数组，也是 0 大小类型。而如果我们为 0 大小的类型申请动态分配内存，我们可以得到，指针指向的地址是 1。这个 1 是怎么回事呢？

当碰到 0 大小类型需要动态分配空间的时候，在标准库里面会直接返回一个 EMPTY 出去。这个 EMPTY 定义在 liballoc/heap.rs 模块中：

```
/// An arbitrary non-null address to represent

/// zero-size allocations.

/// This preserves the non-null invariant for

/// types like Box. The address may overlap

/// with non-zero-size memory allocations.

pub  const EMPTY: *mut () = 0x1  as *mut ();
```

为什么选 1 这个值呢？首先，1 不可能是内存分配器正常返回的地址，其次，0 已经用于表示空指针 ull 的情况，所以选择另外一个不同的值来表示这种情况。那么这两种“空”有什么区别呢，我们继用示例说明：

```
use std::mem::transmute;

fn  main() {

let x : Box<()> = Box::new(());

let y : Option> = None;

let z : Option> = Some(Box::new(()));
```

```
unsafe {

 let value1 : usize = transmute(x);

 let value2 : usize = transmute(y);

 let value3 : usize = transmute(z);println!("{} {} {}", value1, value2, value3);    } }
```

其中的 transmute 函数是强制类型转换的作用。编译执行，结果为：“1 0 1”。所以，解释起来就：非空指针指向 0 大小的类型，指向的是地址 1；空指针都是指向的是地址 0。