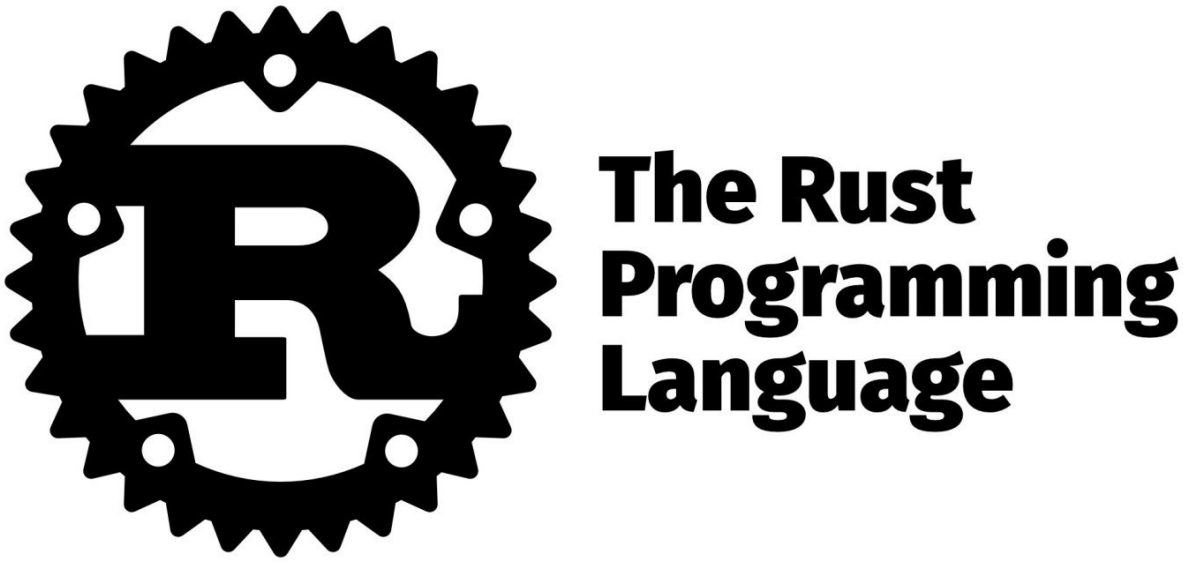
【译】Rust中的Sizedness

zhuanlan.zhihu.com/p/189353352



Praying

> 原文链接: github.com/pretzelhammer
>
> 原文标题: Sizedness in Rust
>
> 公众号: Rust碎碎念

介绍

Sizedness是Rust中需要理解的最重要的概念中最不起眼的一个。它经常以微妙的方式贯穿于众多其他的语言特性之中，并且仅以"x doesn't have size known at compile time"这种每个Rustacean都熟悉的错误信息的方式出现。在本文中，我们将会探讨从确定大小类型(sized type)，到不确定大小类型(unsized type)，再到零大小类型(zero-sized type)等各种类型的sizedness，同时对它们的用例，优势，痛点以及解决方法进行评估。

下面是我所使用的短语表格以及它们的意思:

词语	含义
sizedness	确定大小(sized)或不确定大小(unsized)的特性
sized type	在编译期可以确定大小的类型
1) unsized type or 2) DST	动态大小的类型，例如，在编译期无法确定的大小
?sized type	可能是确定大小也可能是不确定大小的类型
unsized coercion	强制把确定大小类型(sized type)转为不确定大小类型(unsized type)
ZST	零大小(zero-sized)类型，比如，某类型的实例大小为0字节
width	指针宽度的测量单位
1) thin pointer or 2) single-width pointer	1个宽度(width)的指针
1) fat pointer or 2) double-width pointer	2个宽度(width)的指针
1) pointer or 2) reference	指向某种宽度的指针，宽度由上下文确定
slice	指向某个数据的动态大小视图(view)的双宽度(double-width)指针

Sizedness

在Rust里，如果一个类型的字节大小在编译期可以确定，那么这个类型就是确定大小(sized)的。确定类型的大小(size)对于能够在栈(stack)上为实例分配足够的空间是十分重要的。确定大小类型(sized type)可以通过传值(by value)或者传引用(by reference)的方式来传递。如果一个类型的大小不能在编译期确定，那么它就被称为不确定大小类型(unsized type)或者DST，即动态大小类型(Dynamically-Sized Type)。因为不确定大小类型(unsized type)不能存放在栈上，所以它们只能通过传引用(by reference)的方式来传递。下面是确定大小(size)和不确定大小(unsized)类型的一些例子:

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

fn main() {
    // primitives
    assert_eq!(4, size_of::<i32>());
    assert_eq!(8, size_of::<f64>());

    // tuples
    assert_eq!(8, size_of::<(i32, i32)>());

    // arrays
    assert_eq!(0, size_of::<[i32; 0]>());
    assert_eq!(12, size_of::<[i32; 3]>());

    struct Point {
        x: i32,
        y: i32,
    }

    // structs
    assert_eq!(8, size_of::<Point>());

    // enums
    assert_eq!(8, size_of::<Option<i32>>());

    // get pointer width, will be
    // 4 bytes wide on 32-bit targets or
    // 8 bytes wide on 64-bit targets
    const WIDTH: usize = size_of::<&()>();

    // pointers to sized types are 1 width
    assert_eq!(WIDTH, size_of::<&i32>());
    assert_eq!(WIDTH, size_of::<&mut i32>());
    assert_eq!(WIDTH, size_of::<Box<i32>>());
    assert_eq!(WIDTH, size_of::<fn(i32) -> i32>());

    const DOUBLE_WIDTH: usize = 2 * WIDTH;

    // unsized struct
    struct Unsized {
        unsized_field: [i32],
    }

    // pointers to unsized types are 2 widths
    assert_eq!(DOUBLE_WIDTH, size_of::<&str>()); // slice
    assert_eq!(DOUBLE_WIDTH, size_of::<&[i32]>()); // slice
    assert_eq!(DOUBLE_WIDTH, size_of::<&dyn ToString>()); // trait object
    assert_eq!(DOUBLE_WIDTH, size_of::<Box<dyn ToString>>()); // trait object
    assert_eq!(DOUBLE_WIDTH, size_of::<&Unsized>()); // user-defined unsized type

    // unsized types
    size_of::<str>(); // compile error
    size_of::<[i32]>(); // compile error
    size_of::<dyn ToString>(); // compile error
    size_of::<Unsized>(); // compile error
}
```

确定确定大小类型(sized types)的大小的方式是直观的：所有的基本类型和指针拥有确定大小，仅由基本类型和指针或者内嵌的结构体(structs)，元组(tuples)，枚举(enums)以及数组(arrays)组成的结构体(struct)、元组(tuple)、枚举(enum)和数组(array)，我们可以在考虑填充和对齐所需的额外字节数的情况下，递归地把字节数加起来。同样比较直观的原因，我们不能确定不确定大小类型(unsized type)的大小：切片(slice)可以有任意数量的元素从而在运行时的大小也是任意的，trait对象可以被任意数量的结构体(struct)或枚举(enum)实现，因此在运行时也可以是任意的大小。

Pro tips

- 在Rust中，指向数组的动态大小视图(dynamically sized views)被称为切片(slice)。例如，一个 `&str` 是一个"字符串切片(string slice)"，一个 `&[i32]` 是一个"i32切片"。
- 切片(slice)是双宽度(double-width)的，因为他们存储了一个指向数组的**指针**和数组中元素的**数量**。
- trait对象指针是双宽度(double-width)的，因为他们存储了一个指向数据的指针和一个指向vtale的指针。
- 不确定大小(unsized) 结构体指针是双宽度的，因为他们存储了一个指向结构体数据的指针和结构体的大小(size)。
- 不确定大小(unsized) 结构体只能拥有有1个不确定大小(unsized)字段(field)而且它必须是结构体里的最后一个字段(field)。

为了彻底说明不确定大小类型(unsized type)的双宽度(double-width)指针，下面是带有注释的比较数组(array)和切片(slice)代码示例：

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

const WIDTH: usize = size_of::<&()>();
const DOUBLE_WIDTH: usize = 2 * WIDTH;

fn main() {
    // data length stored in type
    // an [i32; 3] is an array of three i32s
    let nums: &[i32; 3] = &[1, 2, 3];

    // single-width pointer
    assert_eq!(WIDTH, size_of::<&[i32; 3]>());

    let mut sum = 0;

    // can iterate over nums safely
    // Rust knows it's exactly 3 elements
    for num in nums {
        sum += num;
    }

    assert_eq!(6, sum);

    // unsized coercion from [i32; 3] to [i32]
    // data length now stored in pointer
    let nums: &[i32] = &[1, 2, 3];

    // double-width pointer required to also store data length
    assert_eq!(DOUBLE_WIDTH, size_of::<&[i32]>());

    let mut sum = 0;

    // can iterate over nums safely
    // Rust knows it's exactly 3 elements
    for num in nums {
        sum += num;
    }

    assert_eq!(6, sum);
}
```

这是另外一个带有注释的比较结构体和trait对象的代码示例:

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

const WIDTH: usize = size_of::<&()>();
const DOUBLE_WIDTH: usize = 2 * WIDTH;

trait Trait {
    fn print(&self);
}

struct Struct;
struct Struct2;

impl Trait for Struct {
    fn print(&self) {
        println!("struct");
    }
}

impl Trait for Struct2 {
    fn print(&self) {
        println!("struct2");
    }
}

fn print_struct(s: &Struct) {
    // always prints "struct"
    // this is known at compile-time
    s.print();
    // single-width pointer
    assert_eq!(WIDTH, size_of::<&Struct>());
}

fn print_struct2(s2: &Struct2) {
    // always prints "struct2"
    // this is known at compile-time
    s2.print();
    // single-width pointer
    assert_eq!(WIDTH, size_of::<&Struct2>());
}

fn print_trait(t: &dyn Trait) {
    // print "struct" or "struct2" ?
    // this is unknown at compile-time
    t.print();
    // Rust has to check the pointer at run-time
    // to figure out whether to use Struct's
    // or Struct2's implementation of "print"
    // so the pointer has to be double-width
    assert_eq!(DOUBLE_WIDTH, size_of::<&dyn Trait>());
}

fn main() {
    // single-width pointer to data
    let s = &Struct;
    print_struct(s); // prints "struct"

    // single-width pointer to data
    let s2 = &Struct2;
```

```
    print_struct2(s2); // prints "struct2"

    // unsized coercion from Struct to dyn Trait
    // double-width pointer to point to data AND Struct's vtable
    let t: &dyn Trait = &Struct;
    print_trait(t); // prints "struct"

    // unsized coercion from Struct2 to dyn Trait
    // double-width pointer to point to data AND Struct2's vtable
    let t: &dyn Trait = &Struct2;
    print_trait(t); // prints "struct2"
}
```

关键点(Key Takeaway)

- 只有确定大小类型(sized type)的实例可以被放到栈上，也就是，可以通过值传递
- 不确定大小类型(unsized type)的实例不能被放置在栈上并且必须通过引用来传递
- 不确定大小类型(unsized type)的指针是双宽度(double-width)的，因为除了指向数据之外，他们还需要做一些额外的记录来追踪数据的长度或者指向一个vtable

Sized Trait

Rust中的 `Sized` trait是自动(auto)trait和标记(marker)trait。

自动trait是能够为满足某些条件的类型自动实现的trait。标记trait是标记一个类型拥有某种特定属性的trait。标记trait没有任何trait项，比如方法，关联函数，关联常量或者关联类型。所有的自动trait都是标记trait但是并非所有的标记trait都是自动trait。自动trait必须是标记trait，这样编译器才能为它们提供一个自动的默认实现，如果trait有任何trait项，这一点就无法做到。

如果一个类型的所有成员都是 `Sized` ，那么这个类型就会自动实现 `Sized` 。 `成员(member)` 的含义取决于具体的类型，例如：一个结构体的字段，枚举的变体(variants)，数组的元素，元组(tuple)的项(item)等等。一旦一个类型被一个 `Sized` 实现“标记(marked)”则意味着在编译期可以确定它的字节数大小。

另外两个自动标记trait的例子是 `Send` 和 `Sync` trait。如果一个类型能够安全地在线程间被发送(send)，那么这个类型是 `Send` 。一个类型如果能够在线程间安全地共享引用，那么这个类型是 `Sync` 。如果一个类型的所有成员都是 `Send` 和 `Sync` ，那么这个类型也会自动实现 `Send` 和 `Sync` 。 `Sized` 比较特殊的一点是，不像其他的自动标记trait可以选择退出(opt-out)， `Sized` 不能选择退出(opt-out)。

```
#![feature(negative_impls)]

// this type is Sized, Send, and Sync
struct Struct;

// opt-out of Send trait
impl !Send for Struct {}

// opt-out of Sync trait
impl !Sync for Struct {}

impl !Sized for Struct {} // compile error
```

这似乎是合理的，因为有时候我们不想我们的类型在线程间能够发送或者共享，但是，很难想象这样一个场景，即我们想要编译器"忘记(forget)"我们的类型的大小而把它作为一个不确定大小类型(unsized type)，因为这样没有任何益处且仅仅是让类型更难使用。

而且，更学究一点儿来讲， `Sized` 从技术上来讲不算是一个自动trait，因为它不是使用 `auto` 关键字定义而是得到了编译器的特殊处理从而使它表现得和自动trait非常相似，所以，在实践中把它看做一个自动trait也是可以的。

关键点(Key Takeaway)

> `Sized` 是一个"自动(auto)"标记trait

泛型中的Sized (Sized in Generic)

当我们写泛型代码的时候，每一个泛型类型参数默认会自动和 `Sized` trait绑定，这件事并不总是显而易见。

```
// this generic function...
fn func<T>(t: T) {}

// ...desugars to...
fn func<T: Sized>(t: T) {}

// ...which we can opt-out of by explicitly setting ?Sized...
fn func<T: ?Sized>(t: T) {} // compile error

// ...which doesn't compile since t doesn't have
// a known size so we must put it behind a pointer...
fn func<T: ?Sized>(t: &T) {} // compiles
fn func<T: ?Sized>(t: Box<T>) {} // compiles
```

Pro tips

- `?Sized` 可以是明显的“可选大小(optionally sized)”或者“可能大小(maybe sized)”，将其添加到类型参数的约束(bound)上，允许该类型是确定大小(sized)或者不确定大小(unsized)
- `?Sized` 一般是指一个“不断扩大的约束(widening bound)”或者一个“宽松约束(relaxed bound)”，因为它是放松(relax)而不是限制了类型参数
- `?Sized` 是Rust中惟一的宽松约束

所以，为什么这很重要？当我们处理泛型参数的时候并且那个类型隐藏在指针背后，我们几乎总是想要选择退出默认的 `Sized` 约束来让我们的函数在其将要接受的参数类型上更加自由。而且，如果我们没有选择退出默认的 `Sized` 约束，我们将最终得到一些令人惊讶和迷惑的编译错误信息。

让我带你踏上我用Rust编写的第一个泛型函数的旅程。我开始学习Rust的时候， `dbg!` 宏还没有被加入到stable版本里，所以我打印调试值的唯一方式就是使用 `println!()" {:?}", some_value);` 。 每次调试都是这么枯燥，所以我决定写一个类似下面这样的 `debug` 辅助函数：

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

fn debug<T: Debug>(t: T) { // T: Debug + Sized
    println!("{:?}", t);
}

fn main() {
    debug("my str"); // T = &str, &str: Debug + Sized ✔
}
```

到目前为止，一切都还不错，但是这个函数会获取传递进来的值的所有权，这有点烦人，所以我把下面的函数改成只获取引用的方式：

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

fn dbg<T: Debug>(t: &T) { // T: Debug + Sized
    println!("{:?}", t);
}

fn main() {
    dbg("my str"); // &T = &str, T = str, str: Debug + !Sized ❌
}
```

这段代码抛出了下面的错误：

```
error[E0277]: the size for values of type `str` cannot be known at compilation
time
 --> src/main.rs:8:9
  |
3 | fn dbg<T: Debug>(t: &T) {
  |        - required by this bound in `dbg`
...
8 |     dbg("my str");
  |         ^^^^^^^^ doesn't have a size known at compile-time
  |
  = help: the trait `std::marker::Sized` is not implemented for `str`
  = note: to learn more, visit <https://doc.rust-lang.org/book/ch19-04-advanced-
types.html#dynamically-sized-types-and-the-sized-trait>
help: consider relaxing the implicit `Sized` restriction
  |
3 | fn dbg<T: Debug + ?Sized>(t: &T) {
  |
```

当我第一次看到这个错误信息，我发现它非常令人困惑。尽管我已经让这个函数在获取参数上比之前更加地严格，但现在却莫名其妙地抛出个错误！发生了什么?

我已经在上面的代码注释中给出了答案，但基本的：Rust在编译时，当把T解析到具体对应的类型时，会执行模式匹配。下面两张表有助于说明：

Type	T	&T
&str	T = &str	T=str

Type	Sized
str	✗
&str	✓
&&str	✓

这就是为什么我在将函数改为传引用之后不得不添加一个 `?Sized` 约束来让它按照预期来工作。正常工作的函数如下:

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

fn debug<T: Debug + ?Sized>(t: &T) { // T: Debug + ?Sized
    println!("{:?}", t);
}

fn main() {
    debug("my str"); // &T = &str, T = str, str: Debug + !Sized ✔
}
```

关键点(Key Takeaway)

- 所有的泛型类型参数默认都是被 `Sized` 自动约束的
- 如果我们有一个泛型函数，该函数接收一个隐藏在指针背后的参数T，比如， `&T` ， `Box<T>` ， `Rc<T>` 等等，那么我们几乎总是要选择退出默认的 `Sized` 约束然后用 `T:?Sized` 来约束

不确定大小类型(Unsized Types)

切片(Slices)

最常见的切片是字符串切片 `&str` 和数组切片 `&[T]` 。切片的好处在于许多其他类型能够强制转换成切片，所以利用切片和Rust的自动类型强制转换(Type coercions)能够让我们写出灵活的API。

类型强制转换(type coercions)可以在几种情况下发生但是主要发生在方法调用的函数参数上。我们感兴趣的一类类型强制转换是解引用强制转换(deref coercions)和不确定大小强制转换(unsized coercions)。一个解引用强制类型转换是在解引用操作之后， `T` 强制转换到 `U` ，即， `T: Deref<Target=U>` ，例如: `String.deref() -> str` 。一个不确定大小强制转换(unsized coercion)是一个确定大小类型(sized type)的 `T` 强制转换为一个不确定大小类型(unsized type)的 `U` ，即， `T:Unsized<U>` ，例如， `[i32;3]->[i32]` 。

```
trait Trait {
    fn method(&self) {}
}

impl Trait for str {
    // can now call "method" on
    // 1) str or
    // 2) String since String: Deref<Target = str>
}
impl<T> Trait for [T] {
    // can now call "method" on
    // 1) any &[T]
    // 2) any U where U: Deref<Target = [T]>, e.g. Vec<T>
    // 3) [T; N] for any N, since [T; N]: Unsize<[T]>
}

fn str_fun(s: &str) {}
fn slice_fun<T>(s: &[T]) {}

fn main() {
    let str_slice: &str = "str slice";
    let string: String = "string".to_owned();

    // function args
    str_fun(str_slice);
    str_fun(&string); // deref coercion

    // method calls
    str_slice.method();
    string.method(); // deref coercion

    let slice: &[i32] = &[1];
    let three_array: [i32; 3] = [1, 2, 3];
    let five_array: [i32; 5] = [1, 2, 3, 4, 5];
    let vec: Vec<i32> = vec![1];

    // function args
    slice_fun(slice);
    slice_fun(&vec); // deref coercion
    slice_fun(&three_array); // unsized coercion
    slice_fun(&five_array); // unsized coercion

    // method calls
    slice.method();
    vec.method(); // deref coercion
    three_array.method(); // unsized coercion
    five_array.method(); // unsized coercion
}
```

关键点(Key Takeaway)

利用切片(slice)和Rust的自动类型强制转换能够让我们写出灵活的API

Trait对象(Trait Objects)

Traits默认是 `?Sized` 。下面的程序:

```
trait Trait: ?Sized {}
```

会抛出这个错误:

```
error: `?Trait` is not permitted in supertraits
 --> src/main.rs:1:14
  |
1 | trait Trait: ?Sized {}
  |              ^^^^^^
  |
  = note: traits are `?Sized` by default
```

我们很快就能知道为什么trait默认是 `?Sized` ，但是首先，让我们问问自己trait是 `?Sized` 意味着什么？让我们把上面的例子进行脱糖(desugar)(译者注：这里指去除语法糖，还原更原始的代码):

```
trait Trait where Self: ?Sized {}
```

所以，默认情况下，trait允许 `self` 可能是一个不确定大小类型(unsized type)。正如我们前面所了解的，我们不能对不确定大小类型(unsized type)进行值传递，因此，这个限制了我们在trait可以定义的方法种类。写一个以传值的方式接收或返回 `self` 的方法应该是不被允许的，然而，令人惊奇的是，这居然可以编译。

```
trait Trait {
    fn method(self); // compiles
}
```

尽管如此，在我们尝试实现这个方法的时候，无论是通过提供的默认实现还是为不确定大小类型(unsized type)实现这个trait，我们都会得到一个编译错误：

```
trait Trait {
    fn method(self) {} // compile error
}

impl Trait for str {
    fn method(self) {} // compile error
}
```

抛出错误:

```
error[E0277]: the size for values of type `Self` cannot be known at compilation
time
 --> src/lib.rs:2:15
  |
2 |     fn method(self) {}
  |               ^^^^ doesn't have a size known at compile-time
  |
  = help: the trait `std::marker::Sized` is not implemented for `Self`
  = note: to learn more, visit <https://doc.rust-lang.org/book/ch19-04-advanced-
types.html#dynamically-sized-types-and-the-sized-trait>
  = note: all local variables must have a statically known size
  = help: unsized locals are gated as an unstable feature
help: consider further restricting `Self`
  |
2 |     fn method(self) where Self: std::marker::Sized {}
  |                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

error[E0277]: the size for values of type `str` cannot be known at compilation
time
 --> src/lib.rs:6:15
  |
6 |     fn method(self) {}
  |               ^^^^ doesn't have a size known at compile-time
  |
  = help: the trait `std::marker::Sized` is not implemented for `str`
  = note: to learn more, visit <https://doc.rust-lang.org/book/ch19-04-advanced-
types.html#dynamically-sized-types-and-the-sized-trait>
  = note: all local variables must have a statically known size
  = help: unsized locals are gated as an unstable feature
```

如果我们决定通过传值的方式传递 `self` ， 我们可通过显式地使用 `Sized` 约束trait来修复第一个错误:

```
trait Trait: Sized {
    fn method(self) {} // compiles
}

impl Trait for str { // compile error
    fn method(self) {}
}
```

现在抛出的错误是:

```
error[E0277]: the size for values of type `str` cannot be known at compilation
time
 --> src/lib.rs:7:6
  |
1 | trait Trait: Sized {
  |              ----- required by this bound in `Trait`
...
7 | impl Trait for str {
  |      ^^^^^ doesn't have a size known at compile-time
  |
  = help: the trait `std::marker::Sized` is not implemented for `str`
  = note: to learn more, visit <https://doc.rust-lang.org/book/ch19-04-advanced-
types.html#dynamically-sized-types-and-the-sized-trait>
```

正如我们所知，使用 `Sized` 约束trait之后，我们就不能够再为像 `str` 这样的不确定大小类型(unsized type)实现这个trait。另一方面，如果我们确实想为 `str` 实现这个trait，一个可行的方案是让这个trait是 `?Sized` 并且通过传引用的方式传递 `self` 。

```
trait Trait {
    fn method(&self) {} // compiles
}

impl Trait for str {
    fn method(&self) {} // compiles
}
```

不同于把整个trait标记为 `?Sized` 或者 `Sized` ， 我们有更细粒度和更精确的选择来标记单个方法为 `Sized` ，像这样:

```
trait Trait {
    fn method(self) where Self: Sized {}
}

impl Trait for str {} // compiles!?

fn main() {
    "str".method(); // compile error
}
```

现在回到最初的问题，为什么trait默认是 `?Sized` ？答案是trait对象。trait对象本质上不确定大小(unsized)的，因为任何大小的任意类型都能实现一个trait，因此，如果是 `Trait:?Sized` ，我们只能为 `dyn Trait` 实现trait。在代码中是这样:

```
trait Trait: ?Sized {}

// the above is REQUIRED for

impl Trait for dyn Trait {
    // compiler magic here
}

// since `dyn Trait` is unsized

// and now we can use `dyn Trait` in our program

fn function(t: &dyn Trait) {} // compiles
```

如果我们尝试编译上面的程序，我们会得到:

```
error[E0371]: the object type `(dyn Trait + 'static)` automatically implements the
trait `Trait`
 --> src/lib.rs:5:1
  |
5 | impl Trait for dyn Trait {
  | ^^^^^^^^^^^^^^^^^^^^^^^^ `(dyn Trait + 'static)` automatically implements
trait `Trait`
```

抛出错误:

```
error[E0038]: the trait `Trait` cannot be made into an object
 --> src/lib.rs:3:18
  |
1 | trait Trait: Sized {}
  |       -----  ----- ...because it requires `Self: Sized`
  |       |
  |       this trait cannot be made into an object...
2 |
3 | fn function(t: &dyn Trait) {}
  |                  ^^^^^^^^^^ the trait `Trait` cannot be made into an object
```

让我们尝试写一个带有 `Sized` 方法的 `?Sized` trait，然后来看看是否可以把它转成一个trait对象：

```
trait Trait {
    fn method(self) where Self: Sized {}
    fn method2(&self) {}
}

fn function(arg: &dyn Trait) { // compiles
    arg.method(); // compile error
    arg.method2(); // compiles
}
```

正如我们之前看到的，只要我们不在trait对象上调用这个 `Sized` 方法，就一切正常。

关键点(Key Takeaway)

- 所有的trait默认都是 `?Sized`
- `impl Trait for dyn Trait` 要求 `Trait:?Sized`
- 我们可以在每一个方法的基础上规定 `Self:Sized`
- 被 `Sized` 约束的trait不能转成trait对象

Trait对象的限制(Trait Object Limitations)

即使一个trait是对象安全的，仍然存在sizeness相关的边界情况，这些情况限制了什么类型可以转成trait对象以及多少种trait和什么样的trait可以通过一个trait对象来表示。

不能把不确定大小类型(unsized type)转成trait对象

```
fn generic<T: ToString>(t: T) {}
fn trait_object(t: &dyn ToString) {}

fn main() {
    generic(String::from("String")); // compiles
    generic("str"); // compiles
    trait_object(&String::from("String")); // compiles, unsized coercion
    trait_object("str"); // compile error, unsized coercion impossible
}
```

抛出下面的错误:

```
error[E0277]: the size for values of type `str` cannot be known at compilation
time
 --> src/main.rs:8:18
  |
8 |     trait_object("str"); // compile error
  |                  ^^^^^ doesn't have a size known at compile-time
  |
  = help: the trait `std::marker::Sized` is not implemented for `str`
  = note: to learn more, visit <https://doc.rust-lang.org/book/ch19-04-advanced-
types.html#dynamically-sized-types-and-the-sized-trait>
  = note: required for the cast to the object type `dyn std::string::ToString`
```

传递一个 `&String` 到一个期望接收参数类型是 `&dyn ToString` 的函数能够工作是因为类型强制转换(type coercion)。 `String` 实现了 `ToString` (这个trait)并且我们通过一个不确定大小强制转换(unsized coercion)可以把像 `String` 这样的确定大小类型(sized type)转换成一个像 `dyn ToString` 这样的不确定大小类型(unsized type)。 `str` 也实现了 `ToString` ，并且把 `str` 转换成一个 `dyn ToString` 也需要一个不确定大小强制转换(unsized coercion)，但是 `str` 已经是不确定大小(unsized)的了！我们怎么样能把一个已经是不确定大小类型转成另一个不确定大小类型?

`&str` 指针是双宽度(double-width)的，存储了一个指向数据的指针和数据长度。 `&dyn ToString` 指针也是双宽度(double-width)，存储了一个指向数据的指针和一个指向vtable的指针。要把一个 `&str` 强制转换成 `&dyn toString` 将会需要一个三倍宽度(triple-width)的指针来存储执行数据的指针、数据的长度、指向vtable的指针。Rust不支持三倍宽度(triple-width)的指针，所以把一个不确定大小类型转为一个trait对象是不可能的。

上面的两段可以用下面这张表来总结:

类型(Type)	指向数据的指针(Pointer to Data)	数据长度(Data Length)	指向VTable的指针(Pointer to VTable)	总长度(Total Width)
&String	✔	✘	✘	1 ✔
&str	✔	✔	✘	2 ✔
&String as &dyn ToString	✔	✘	✔	2 ✔
&str as &dyn ToString	✔	✔	✔	3 ✘

不能创建多Trait的对象(Cannot create Multi-Trait Objects)

```
trait Trait {}
trait Trait2 {}

fn function(t: &(dyn Trait + Trait2)) {}
```

抛出:

```
error[E0225]: only auto traits can be used as additional traits in a trait object
 --> src/lib.rs:4:30
  |
4 | fn function(t: &(dyn Trait + Trait2)) {}
  |                      -----   ^^^^^^
  |                      |       |
  |                      |       additional non-auto trait
  |                      |       trait alias used in trait object type (additional
use)
  |                      first non-auto trait
  |                      trait alias used in trait object type (first use)
```

记住:一个trait对象指针是双宽度的：存储一个指向数据的指针和一个指向vtable的指针。但是，这里有两个trait，所以有两个vtable，这就要求 `&(dyn Trait + Trait2)` 的指针是3个宽度的。像 `Sync` 和 `Send` 这样的自动trait(Auto-trait)可以被允许是因为他们没有方法也就没有vtable。

这种情况的解决方法是通过另一个trait把(多个)trait进行组合的方式来把vtable进行拼接，像下面这样：

```
trait Trait {
    fn method(&self) {}
}

trait Trait2 {
    fn method2(&self) {}
}

trait Trait3: Trait + Trait2 {}

// auto blanket impl Trait3 for any type that also impls Trait & Trait2
impl<T: Trait + Trait2> Trait3 for T {}

// from `dyn Trait + Trait2` to `dyn Trait3`
fn function(t: &dyn Trait3) {
    t.method(); // compiles
    t.method2(); // compiles
}
```

这种方法的一个缺点在于，Rust不支持supertait向上转换。这意味着，如果我们有一个 `&dyn Trait3` ，但是我们不能在需要 `dyn Trait` 和 `dyn Trait2` 的地方使用它。下面的程序无法编译:

```
trait Trait {
    fn method(&self) {}
}

trait Trait2 {
    fn method2(&self) {}
}

trait Trait3: Trait + Trait2 {}

impl<T: Trait + Trait2> Trait3 for T {}

struct Struct;
impl Trait for Struct {}
impl Trait2 for Struct {}

fn takes_trait(t: &dyn Trait) {}
fn takes_trait2(t: &dyn Trait2) {}

fn main() {
    let t: &dyn Trait3 = &Struct;
    takes_trait(t); // compile error
    takes_trait2(t); // compile error
}
```

抛出:

```
error[E0308]: mismatched types
  --> src/main.rs:22:17
   |
22 |     takes_trait(t);
   |                 ^ expected trait `Trait`, found trait `Trait3`
   |
   = note: expected reference `&dyn Trait`
              found reference `&dyn Trait3`

error[E0308]: mismatched types
  --> src/main.rs:23:18
   |
23 |     takes_trait2(t);
   |                  ^ expected trait `Trait2`, found trait `Trait3`
   |
   = note: expected reference `&dyn Trait2`
              found reference `&dyn Trait3`
```

这是因为 `dyn Trait3` 是一个不同于 `dyn Trait` 和 `dyn Trait2` 的类型，因为它们有不同的vtable布局，尽管 `dyn Trait3` 的确包含了 `dyn Trait` 和 `dyn Trait2` 的所有方法。这里的解决办法是添加显式的转换方法:

```
trait Trait {}
trait Trait2 {}

trait Trait3: Trait + Trait2 {
    fn as_trait(&self) -> &dyn Trait;
    fn as_trait2(&self) -> &dyn Trait2;
}

impl<T: Trait + Trait2> Trait3 for T {
    fn as_trait(&self) -> &dyn Trait {
        self
    }
    fn as_trait2(&self) -> &dyn Trait2 {
        self
    }
}

struct Struct;
impl Trait for Struct {}
impl Trait2 for Struct {}

fn takes_trait(t: &dyn Trait) {}
fn takes_trait2(t: &dyn Trait2) {}

fn main() {
    let t: &dyn Trait3 = &Struct;
    takes_trait(t.as_trait()); // compiles
    takes_trait2(t.as_trait2()); // compiles
}
```

这是一种简单且直接的解决办法，似乎可以让编译器来为我们自动化实现。正如我们在解引用和不确定大小强制转换中所见，Rust并不羞于执行强制类型转换，那么，为什么没有一个trait的向上强制转换？这个问题很好，答案也很熟悉:Rust核心团队正在研究其他具有更高优先级和更具影响力的的特性。很好。

关键点(Key Takeaway)

- Rust不支持超过2个宽度的指针，所以
- 我们不能够把不确定大小类型(unsized type)转换为trait对象
- 我们不能有多trait对象，但是我们可以通过把多个trait合并到一个trait里来解决

用户定义的不确定大小类型

```
struct Unsized {
    unsized_field: [i32],
}
```

我们可以通过给结构体添加一个不确定大小(unsized)的字段来定义一个不确定大小结构体。不确定大小结构体只能有1个不确定大小字段并且该字段必须是这个结构体里的最后一个字段。这是第一个必要条件，以便于编译器在编译时确定每个字段在结构体中的起始偏移量，这对于高效快速的字段访问是必要的。而且，使用一个双宽度(double-width)指针最多可以追踪一个不确定大小字段，因为更多的不确定大小字段会需要更多的宽度。

那我们怎么来实例化这个结构体呢？和处理其他不确定大小类型的方式一样：首先构造一个确定性大小的版本，然后把它强制转换为不确定大小的版本。尽管如此，根据定义，`Unsized` 总是不确定大小的，没有办法构造一个它的确定性大小版本。唯一的解决方法是把这个结构体变成泛型(generic)的，这样它就可以存在于确定性大小和不确定性大小的版本里。

```
struct MaybeSized<T: ?Sized> {
    maybe_sized: T,
}

fn main() {
    // unsized coercion from MaybeSized<[i32; 3]> to MaybeSized<[i32]>
    let ms: &MaybeSized<[i32]> = &MaybeSized { maybe_sized: [1, 2, 3] };
}
```

所以这个的使用场景是什么？没有任何特别引人注目的特性，用户定义的不确定大小类型目前是一个相当不成熟的特性，它们的局限性超过了所能带来的益处。这里提到它纯粹了是为了内容的全面性。

有趣的事实: `std::ffi::OsStr` 和 `std::path::Path` 是标准库里的两个不确定大小结构体，你之前可能已经使用过他们但是并没有意识到。

关键点(Key Takeaway)

> 用户定义的不确定大小类型是一个目前尚不成熟的特性并且它们的局限性超过了所能带来的益处。

零大小类型(Zero-Sized Types)

ZST一开始听起来感觉很奇怪，但是几乎到处都在使用它们。(译者注: ZST即Zero-Sized Type的缩写)

单元类型(Unit Type)

最常见的ZST是是单元类型(也叫空元组)： `()` ，所有的空块 `{}` 的计算结果为 `()` ，并且，如果这个代码块是非空但是使用一个分号来丢弃最后的表达式，计算的结果也是 `()` ， 例如:

```
fn main() {
    let a: () = {};
    let b: i32 = {
        5
    };
    let c: () = {
        5;
    };
}
```

没有明确返回类型的函数默认都返回 `()` :

```
// with sugar
fn function() {}

// desugared
fn function() -> () {}
```

因为 `()` 是零字节，所以 `()` 的实例都是相同的，这就使得 `Default` ， `PartialEq` 和 `Ord` 的实现变得相当简单:

```
use std::cmp::Ordering;

impl Default for () {
    fn default() {}
}

impl PartialEq for () {
    fn eq(&self, _other: &()) -> bool {
        true
    }
    fn ne(&self, _other: &()) -> bool {
        false
    }
}

impl Ord for () {
    fn cmp(&self, _other: &()) -> Ordering {
        Ordering::Equal
    }
}
```

编译器理解 `()` 是零大小类型并且会优化和 `()` 实例有关的交互。例如:一个 `Vec<()>` 永远不会执行堆分配，从 `Vec` 里推进(push)和弹出(pop) `()` 只是对它里面的 `len` 字段进行增加或减少。

```
fn main() {
    // zero capacity is all the capacity we need to "store" infinitely many ()
    let mut vec: Vec<()> = Vec::with_capacity(0);
    // causes no heap allocations or vec capacity changes
    vec.push(()); // len++
    vec.push(()); // len++
    vec.push(()); // len++
    vec.pop(); // len--
    assert_eq!(2, vec.len());
}
```

上面的例子没有实际的应用，但是是否存一些解决方案，能够让我们以一种有意义的方式充分利用上面的思想？是的，我们可以通过把 `HashMap<Key, Value>` 的 `Value` 设置为 `()` 来得到一个更高效的 `HashSet<Key>` 实现，这确实也是Rust标准库里 `HashSet` 里的实现方式:

```
// std::collections::HashSet
pub struct HashSet<T> {
    map: HashMap<T, ()>,
}
```

关键点(Key Takeaway)

- 所有ZST的实例都是相等的
- Rust编译器会去优化和ZST相关的操作

用户定义的单元结构体(User-Defined Unit Structs)

单元结构体是没有任何字段的结构体，例如:

```
struct Struct;
```

单元结构比（）更有用的一些属性:

- 我们可以在自己的单元结构体上实现任何我们想要的trait， Rust的trait孤儿规则阻止我们为（）实现trait，因为（）被定义在标准库里
- 在我们程序的上下文环境中，单元结构体可以被赋予有意义名字
- 单元结构体，像所有的结构体一样，默认是非拷贝(non-Copy)的，这在我们程序上下文环境中可能是重要的

Never Type

第二常见的ZST是never类型： ！。它被叫做never类型是因为它表示永远不会产生任何值的计算。

never类型不同于（）的一些有趣的属性:

- ！可以被强制转换到任意其他的类型
- 无法创建一个！的实例

第一个有趣的属性对于对于人体工程学非常有用并且允许我们使用下面这些方便的宏:

```
// nice for quick prototyping
fn example<T>(t: &[T]) -> Vec<T> {
    unimplemented!() // ! coerced to Vec<T>
}

fn example2() -> i32 {
    // we know this parse call will never fail
    match "123".parse::<i32>() {
        Some(num) => num,
        None => unreachable!(), // ! coerced to i32
    }
}

fn example3(bool: someCondition) -> &'static str {
    if (!someCondition) {
        panic!() // ! coerced to &str
    } else {
        "str"
    }
}
```

`break`， `continue`，和 `return` 表达式也有 `!` 类型：

```
fn example() -> i32 {
    // we can set the type of x to anything here
    // since the block never evaluates to any value
    let x: String = {
        return 123 // ! coerced to String
    };
}

fn example2(nums: &[i32]) -> Vec<i32> {
    let mut filtered = Vec::new();
    for num in nums {
        filtered.push(
            if *num < 0 {
                break // ! coerced to i32
            } else if *num % 2 == 0 {
                *num
            } else {
                continue // ! coerced to i32
            }
        );
    }
    filtered
}
```

`!` 的第二个有趣的属性让我们能够让我们在类型级别把特定的状态标记为不可能。让我们看看下面的函数：

```
fn function() -> Result<Success, Error>;
```

我们知道，如果这个函数返回并且是成功的， `Result` 将会持有 `Success` 类型的实例，如果它出错了， `Result` 将会包含 `Error` 类型的某个实例。现在让我们和下面这个函数前面对比一下：

```
fn function() -> Result<Success, !>;
```

我们知道，如果这个函数返回并且是成功的， `Result` 将会持有 `Success` 类型的实例，如果它出错了...等等，它永远不会出错，因为无法创建一个 `!` 的实例。通过上面给出的函数签名，我们知道这个函数永远不会出错。那么下面这个函数签名呢：

```
fn function() -> Result<!, Error>;
```

和前面的恰好相反:如果这个函数返回，我们知道它一定是出错的，因为成功是不可能的。

前面例子的一个实际应用是 `String` 的 `FromStr` 实现，因为它在把 `&str` 转换到 `String` 时是不可能失败的：

```
#![feature(never_type)]

use std::str::FromStr;

impl FromStr for String {
    type Err = !;
    fn from_str(s: &str) -> Result<String, Self::Err> {
        Ok(String::from(s))
    }
}
```

后面一个例子的实际应用是一个运行无限循环的函数，这个函数永远不会返回，比如一个响应客户端请求的服务器，除非出现某些错误:

```
#![feature(never_type)]

fn run_server() -> Result<!, ConnectionError> {
    loop {
        let (request, response) = get_request()?;
        let result = request.process();
        response.send(result);
    }
}
```

feature标志是必要的，因为never类型存在并工作于Rust内部，在用户代码里使用它被当做是实验性质(experimental)的。

关键点(Key Takeaway)

- `!` 可以被强制转到到任何其他的类型
- 无法创建 `!` 的实例，我们可以使用这一点在类型级别把一个状态标记为不可能的

用户定义的伪Never类型(User-Defined Pseudo Never Types)

尽管定义一个能够强制转换到任意其他类型的类型是不可能的，但是定义一个无法创建实例的类型是有可能的，例如一个没有任何variant的 `enum` :

```
enum Void {}
```

这可以让我们从前面的两个例子中移除feature标记并且使用stable版本的Rust来实现它们:

```
enum Void {}

// example 1
impl FromStr for String {
    type Err = Void;
    fn from_str(s: &str) -> Result<String, Self::Err> {
        Ok(String::from(s))
    }
}

// example 2
fn run_server() -> Result<Void, ConnectionError> {
    loop {
        let (request, response) = get_request()?;
        let result = request.process();
        response.send(result);
    }
}
```

这是Rust标准库里使用的技术，因为 `String` 的 `FromStr` 实现里的 `Err` 类型是 `std::convert::Infallible` ， 其定义如下:

```
pub enum Infallible {}
```

PhantomData

第三常见的使用ZST的应该是 `PhantomData` 。 `PhantomData` 是一个零大小标记结构体，可以用于“标记(mark)”一个包含(containing)结构体含有特定的属性。在使用目的上类似于像 `Sized` ， `Send` 和 `Sync` 这样的自动标记trait，但是作为一个标记结构体在使用上略有不同。对 `PhantdomData` 作一个详尽的解释并探讨它的各种用例已经超出了本文的范围，所以，让我们仅仅来看一个简单的示例。回顾一下前面出现的这个代码示例:

```
#![feature(negative_impls)]

// this type is Send and Sync
struct Struct;

// opt-out of Send trait
impl !Send for Struct {}

// opt-out of Sync trait
impl !Sync for Struct {}
```

很不幸地，我们不得不使用一个feature标记，我们能否只使用stable版本的Rust就实现相同的结果？正如我们所知，一个类型只有在它所有的成员都是 `Send` 和 `Sync` 的时候，它才是 `Send` 和 `Sync` ，因此，我们可以在结构体中添加一个 `!Send` 和 `!Sync` 成员，比如 `Rc<()>` :

```
use std::rc::Rc;

// this type is not Send or Sync
struct Struct {
    // adds 8 bytes to every instance
    _not_send_or_sync: Rc<()>,
}
```

这样并不理想，因为它为 `Struct` 的每个实例都增加了大小，而且现在每次要创建一个 `Struct` 的时候，我们不得不凭空构造出一个 `Rc<()>` 。而 `PhantomData` 是一个ZST，同时解决了这两个问题:

```
use std::rc::Rc;
use std::marker::PhantomData;

type NotSendOrSyncPhantom = PhantomData<Rc<()>>;

// this type is not Send or Sync
struct Struct {
    // adds no additional size to instances
    _not_send_or_sync: NotSendOrSyncPhantom,
}
```

关键点(Key Takeaway)

> `PhantomData` 是一个零大小标记结构体，可以用于标记一个包含结构体为拥有特定的属性

译者注:因不确定 `PhantomData` 译为 `幻影数据` 是否合适，故此处保留原词。

总结(Conclusion)

- 只有确定大小类型(sized type)的实例才可以放到栈上，也就是，可以通过传值的方式传递
- 不确定大小类型(unsized tpe)的实例不能放到栈上而且必须通过传引用的方式传递
- 指向不确定大小类型(unsized tpe)的指针是双宽度的，因为除了保存指向数据的指针外，还需要额外的比特位来追踪数据的长度或者指向一个vtable
- `Sized` 是一个"自动(auto)"标记trait
- 所有的泛型类型参数默认是被 `Sized` 自动约束
- 如果我们有一个泛型函数，它接收隐于指针后的类型 `T` 为参数，例如 `&T` ， `Box<T>` ， `Rc<T>` 等，那么我们总是选择退出默认的 `Sized` 约束而选用 `T:?Sized` 约束
- 利用切片和Rust的自动类型强制转换能够让我们写出灵活的API
- 所有的trait默认都是 `?Sized`
- 对于 `impl Trait for dyn Trait` ，要求 `Trait: ?Sized`
- 我们可以在每个方法上要求 `Self:Sized`
- 由 `Sized` 约束的trait不能转为trait对象
- Rust不支持超过2个宽度的指针，因此
- 我们不能把不确定大小类型转为trait对象

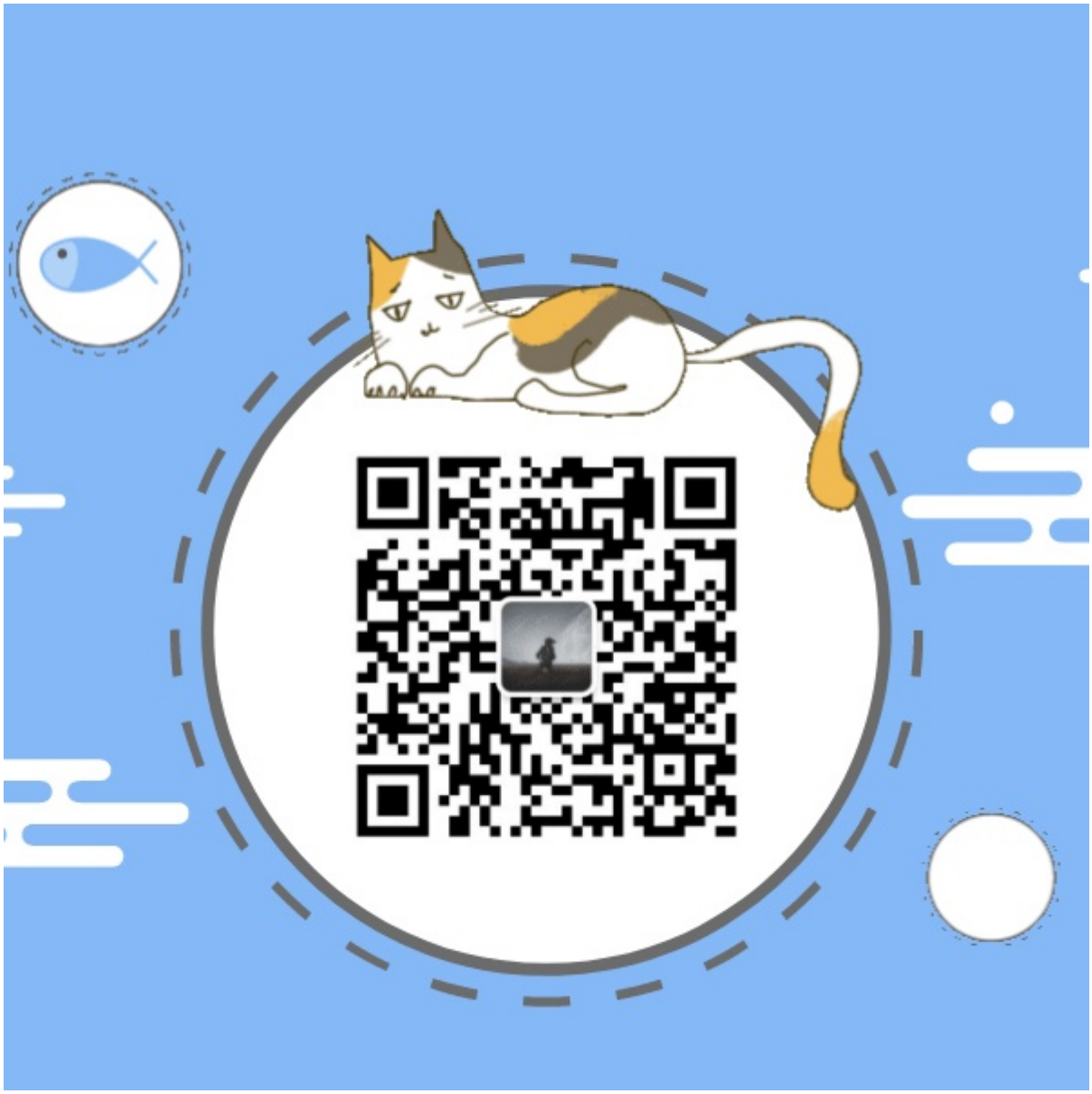
- 我们不能有多trait对象，但是我们可以通过把多个trait合并到一个trait里来解决这个问题
- 用户定义的不确定类型大小类型是个不成熟的特性，现在其局限性超过所能带来的益处
- ZST的所有实例都相等
- Rust编译器会去优化和ZST相关的交互
- `!` 可以被强制转换为其他类型
- 无法创建一个 `!` 的实例，我们可以利用这一点在类型级别把特定状态标记为不可能
- `PhantomData` 是一个零大小标记结构体，可以用于把一个包含结构体标记为含有特定属性

讨论(Discuss)

可以在下面这些地方讨论这篇文章 - official Rust users forum - learnrust subreddit - Twitter - rust subreddit

Further Reading

本文禁止转载，谢谢配合！欢迎关注我的公众号: Rust碎碎念



发布于 2020-08-19

Rust（编程语言）

文章被以下专栏收录

Rust碎碎念

关于Rust的一些杂记，可能很零散，不成体系。