链滴

Rust 宏笔记

作者：plus7wist

原文链接：https://ld246.com/article/1592390738395

来源网站：链滴

这篇文章说的是?

Rust 的宏。

宏按照来源分类:

声明宏（Declarative Macro）和过程宏（Procedural Macro）。前者指的是用某种语法直接声明出宏。后者是对应直接生成抽象语法树的过程的宏。

直觉上过程宏更隐式，更全能；声明宏更可读，更直接。

如何定义声明宏?

现在用 macro_rules!。以后可能还有别的办法。

如何定义过程宏?

以后再说。

宏按照使用方式分类:

属性宏：给声明添加属性的宏，例如 #[derive(Debug)] 和 #[test]。

调用宏：像函数一样的宏，例如 println!。

来源分类和使用方式分类之间的关系如何？

目前的声明宏都是用 macro_rules! 声明出的，它声明出的一定是调用宏。过程宏可以产生属性宏，可以产生调用宏。

也就是说，属性宏都是过程宏，调用宏可能是声明宏或者过程宏。

---

println! 宏大概是什么样子？

```
macro_rules! println {
  () => (println!("\n"));
  ($fmt: expr) => (print!(concat!($fmt, "\n")));
  ($fmt: expr, $($arg:tt)*) =>
    (print!(concat!($fmt, "\n"), $($arg)*));
}
```

这个宏有几部分？

有三个部分，输入分别是 ()、($fmt:expr) 和 ($fmt: expr, $($args:tt)*)，依次扩展成 => 后，圆括号部的部分。每个部分是一条**规则**，每条规则以 ; 结尾。

=> 后的圆括号是必须的吗？

不能省略，但是可以换成 {} 或者 []。

$fmt: expr 是什么？

$fmt 是对宏参数的**捕获**，类似于函数的参数。expr 表示这个捕获的类型是**表达式**，也就是会它会生具体的值。具体到此处，它代表生成 println! 的格式字符串的表达式。

捕获有什么用？

宏的替换结果里可以用 $fmt 代表要替换这个捕获。例如 println!("Hello") 中，$fmt: expr 捕获了 "Hllo"，所以 print!(concat!($fmt, "\n)) 中的 $fmt 会被替换为 "Hello"，所以展开成 print!(concat!("Hllo", "\n"))。

展开的宏中如果还有宏，还会继续展开吗？

会，上面 println! 展开之后的内容中有 print! 和 concat!，它们都会再次展开。这是理所当然的行为

这个问题只是为了让 Rust 的宏跟 C++ 的宏划清界限。

都有什么捕获类型?

类型	意义
item	语言项，模块、定义、声明等
block	代码块，花括号限定的代码
stmt	语句，分号结尾的代码
expr	表达式，会生成具体的值
pat	模式
ty	类型
ident	标识符
path	路径，指从crate到mod的定位
meta	元信息
tt	TokenTree的缩写，词条树
vis	可见性，例如pub
lifetime	生命周期参数

$($arg:tt)* 是什么意思?

单独看 $arg:tt 表示匹配一个词条树的捕获，在外面套上 $()* 表示匹配若干次词条树。

$($arg)* 是什么意思?

单看 $arg，表示在宏里替换捕获 $arg，外面套上 $()* 表示使用所有匹配的捕获。这个用法跟它的捕语法是对应的。

$($arg:tt)* 能匹配什么?

println! 在第一个参数之后的所有东西。这个宏不止可以传递像函数一样的参数，还可以像 Python 样传递命名参数，例如:

println!("Hello, {name}", name="Luna");

这样的参数 $($arg:tt)* 也能捕获到。

如果想只捕获（不定个数个）函数参数应该如何做?

用 $($arg: expr),* 或者 $($arg: expr,)*。

这两个有什么不同?

后者也匹配逗号结尾的参数列表。

Rust 的函数参数列表最后可以添加逗号，也可以不加。如果想让宏表现的尽量接近函数，应该两种况都处理。

举个例子。

```
macro_rules! hash_map {
  ($($key:expr => $value:expr),*) => {{
    let mut map = ::std::collections::HashMap::new();
    $( map.insert($key, $value); )*
    map
  }};
  ($($key:expr => $value:expr),*) =>
    (hash_map! ($($key => $value),*));
}
```

怎么使用?

```
let map = hash_map! (1 => "one", 2 => "two", 3 => "three");
```

如何获取宏可变参数的长度?

没有直接的办法，但可以想些技巧。

```
macro_rules! unit {
  ($($x:tt)*) => (());
}

macro_rules! count {
  ($($x:expr),*) => (<[()]>::len(&[$(unit!($x)),*]));
}
```

它是如何工作的?

unit! 接受任意什么东西，返回一个 unit（()）。count! 宏把参数填给 unit!，构造了一个 unit 数组数组长度就是参数的个数。

好处是，unit 不占空间，unit 的数组也是。

---

count! 宏需要 unit! 宏才能工作，但 unit! 宏本身没什么用，能不能把 unit! 变成私有的?

也没有直接的办法。基本上的技巧是换成一个比较难直接用到的规则：

```
macro_rules! count {
  (@unit $($x:tt)*) => (());

  ($($x:expr),*) => (<[()]>::len(&[$(count!(@unit $x)),*]));
}
```

要是有人非要写 count!(@unit ...)，也没法阻止。但那个奇奇怪怪的 @ 已经暗示了这是一个内部实，这就足够了。

#[macro_export] 有什么作用?

标记一个宏可以在其他包中使用。也就是说，默认情况下，宏不能在定义的包外使用。

#[macro_use] 有什么用?

在 Rust 2015 中，在外部包的声明语前，用它标记要导入另一个包的宏。另外，它还标记一个 mod 宏可以在外面使用。

宏重名会怎样?

后导入的会覆盖先前导入的，不会发生错误。

$crate 有什么用?

它看起来像一个捕获，但不是在宏参数列表里捕获的。它会扩展成当前包的名字。需要这个捕获的原是，当前包无法决定此包被其他包导入的时候，使用的是什么名字。所以如果要使用当前包的函数或别的东西，就需要从 $crate 开始写路径。

完。