Rust宏小册 中文版

本文档为The Little Book of Rust Macros的中文翻译。本书试图提炼出一份Rust社区对Rust宏知识的集锦。

目　录

致谢

当前文档 《Rust宏小册 中文版》 由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建，生成于 2020-02-22。

内容来源: DaseinPhaos https://github.com/DaseinPhaos/tlborm-chinese

文档地址: http://www.bookstack.cn/books/DaseinPhaos-tlborm-chinese

书栈官网: https://www.bookstack.cn

书栈开源: https://github.com/TruthHun

Rust宏小册 中文版

本书试图提炼出一份Rust社区对Rust宏知识的集锦。因此，我们欢迎社区成员进行内容添补（通过pull）或提出需求（通过issue）。

本文档为The Little Book of Rust Macros的中文翻译。如果希望为原文作出贡献，请移步至原版的repository。中文版的repo在这里。

致谢

感谢下列成员作出的建议及修正: IcyFoxy, Rym, TheMicroWorm, Yurume, akavel, cmr, eddyb, ogham, and snake_case.

许可证

宏，彻底剖析

本章节将介绍Rust的“示例宏”(Macro-By-Example)系统： `macro_rules` 。我们不会通过实际的示例来介绍它，而将尝试对此系统的运作方式给出完备且彻底的解释。因此，本章的目标读者应是那些想要理清这整个系统的人，而非那些仅仅想要了解它一般该怎么用的人。

在Rust官方教程中也有一章讲解宏(中文版)，它更易理解，提供的解释更加高层。本书也有一章实践介绍，旨在阐释如何在实践中实现一个宏。

语法扩展

在谈及宏之前，我们首先应当讨论语法扩展这一一般性机制。宏正是在它之上构建的。而想要弄明白语法扩展，我们则应该首先阐述编译器处理Rust源代码的机制。

源码解析过程

Rust程序编译过程的第一阶段是标记解析(tokenization)。在这一过程中，源代码将被转换成一系列的标记(token，即无法被分割的词法单元；在编程语言世界中等价于“单词”)。Rust包含多种标记，比如：

- 标识符(identifiers): `foo`, `Bambous`, `self`, `we_can_dance`, `LaCaravane`, …
- 整数(integers): `42`, `72u32`, `0_______0`, …
- 关键词(keywords): `_`, `fn`, `self`, `match`, `yield`, `macro`, …
- 生命周期(lifetimes): `'a`, `'b`, `'a_rare_long_lifetime_name`, …
- 字符串(strings): `""`, `"Leicester"`, `r##"venezuelan beaver"##`, …
- 符号(symbols): `[`, `:`, `::`, `->`, `@`, `<-`, …

…等等。

上面的叙述中有些地方值得注意。

首先， `self` 既是一个标识符又是一个关键词。几乎在所有情况下它都被视作是一个关键词，但它有可能被视为标识符。我们稍后会（带着咒骂）提到这种情况。

其次，关键词里列有一些可疑的家伙，比如 `yield` 和 `macro` 。它们在当前的Rust语言中并没有任何含义，但编译器的确会把它们视作关键词进行解析。这些词语被保留作语言未来扩充时使用。

第三，符号里也列有一些未被当前语言使用的条目。比如 `<-` ，这是历史残留：目前它被移除了Rust语法，但词法分析器仍然没丢掉它。

最后，注意 `::` 被视作一个独立的标记，而非两个连续的 `:` 。这一规则适用于Rust中所有的多字符符号标记。^逝去的@

作为对比，某些语言的宏系统正扎根于这一阶段。Rust并非如此。举例来说，从效果来看，C/C++的宏就是在这里得到处理的。^其实不是这也正是下列代码能够运行的原因:^这看起来不错

```
#define SUB void
#define BEGIN {
#define END }

SUB main() BEGIN
    printf("Oh, the horror!\n");
END
```

编译过程的下一个阶段是语法解析(parsing)。这一过程中，一系列的标记将被转换成一棵抽象语法树

(Abstract Syntax Tree, AST)。此过程将在内存中建立起程序的语法结构。举例来说，标记序列 `1+2` 将被转换成某种类似于：

```
┌──────────┐   ┌──────────┐
│ BinOp    │ ┌─│ LitInt   │
│ op: Add  │ │ │ val: 1   │
│ lhs: ○   │─┘ └──────────┘
│ rhs: ○   │─┐ ┌──────────┐
└──────────┘ └─│ LitInt   │
               │ val: 2   │
               └──────────┘
```

的东西。生成出的AST将包含整个程序的结构，但这一结构仅包含词法信息。举例来讲，在这个阶段编译器虽然可能知道某个表达式提及了某个名为 `a` 的变量，但它并没有办法知道 `a` 究竟是什么，或者它在哪儿。

在AST生成之后，宏处理过程才开始。但在讨论宏处理过程之前，我们需要先谈谈标记树(token tree)。

标记树

标记树是介于标记与AST之间的东西。首先明确一点，几乎所有标记都构成标记树。具体来说，它们可被看作标记树叶节点。另有一类存在可被看作标记树叶节点，我们将在稍后提到它。

只有一种基础标记不是标记树叶节点，“分组”标记： `(...)` ， `[...]` 和 `{...}` 。这三者属于标记树内节点， 正是它们给标记树带来了树状的结构。给个具体的例子，这列标记：

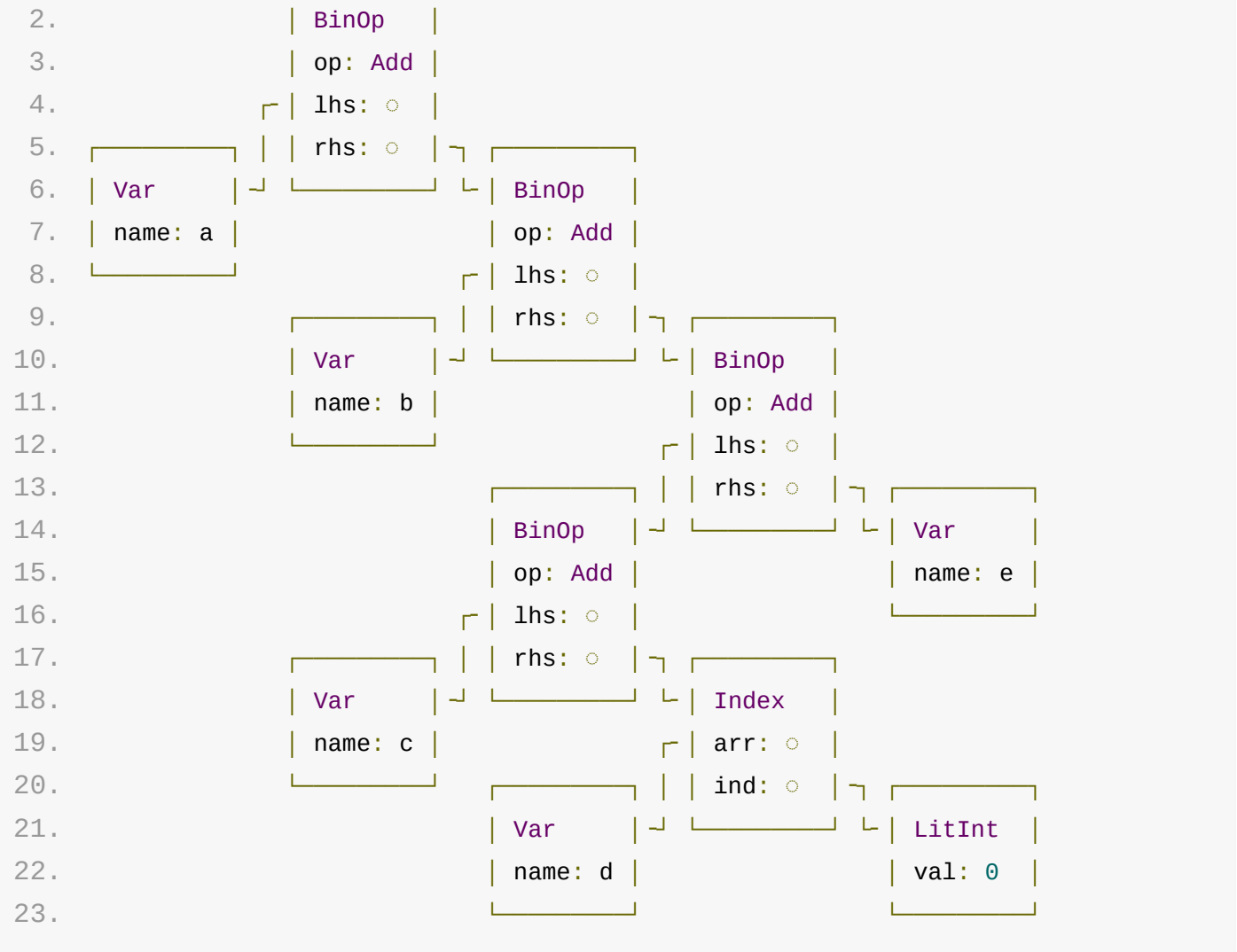
```
a + b + (c + d[0]) + e
```

将被转换为这样的标记树：

```
«a» «+» «b» «+» «(    )» «+» «e»
           ┌──────────┴──────────┐
            «c» «+» «d» «[   ]»
                          ┌─┴─┐
                           «0»
```

注意它跟最后生成的AST并没有关联。AST将仅有一个根节点，而这棵标记树有九（原文如此）个。作为参照，最后生成的AST应该是这样：

```
               ┌──────────┐
```



理解AST与标记树间的区别至关重要。写宏时，你将同时与这两者打交道。

还有一条需要注意：不可能出现不匹配的小/中/大括号，也不可能存在包含错误嵌套结构的标记树。

AST中的宏

如前所述，在Rust中，宏处理发生在AST生成之后。因此，调用宏的语法必须是Rust语言语法中规整相符的一部分。实际上，Rust语法包含数种“语法扩展”的形式。我们将它们同用例列出如下：

- `# [ $arg ]`；如 `#[derive(Clone)]`，`#[no_mangle]`，…
- `# ! [ $arg ]`；如 `#![allow(dead_code)]`，`#![crate_name="blang"]`，…
- `$name ! $arg`；如 `println!("Hi!")`，`concat!("a", "b")`，…
- `$name ! $arg0 $arg1`；如 `macro_rules! dummy { () => {}; }`.

头两种形式被称作“属性(attribute)”，被同时用于语言特属的结构(比如用于要求兼容C的ABI的 `#[repr(C)]`)以及语法扩展(比如 `#[derive(Clone)]`)。当前没有办法定义这种形式的宏。

我们感兴趣的是第三种：我们通常使用的宏正是这种形式。注意，采用这种形式的并非只有宏：它是一种一般性的语法扩展形式。举例来说，`format!` 是宏，而 `format_args!` (它被用于 `format!`) 并不是。

第四种形式实际上宏无法使用。事实上，这种形式的唯一用例只有 `macro_rules!` 我们将在稍后谈到它。

将注意力集中到第三种形式 (`$name ! $arg`)上，我们的问题变成，对于每种可能的语法扩展，Rust的语法分析器(parser)如何知道 `$arg` 究竟长什么样?答案是它不需要知道。其实，提供给每次语法扩展调用的参数，是一棵标记树。具体来说，一棵非叶节点的标记树；即 `(...)`，`[...]`，或 `{...}`。拥有这一知识后，语法分析器如何理解如下调用形式，就变得显而易见了：

```
1.  bitflags! {
2.      flags Color: u8 {
3.          const RED    = 0b0001,
4.          const GREEN  = 0b0010,
5.          const BLUE   = 0b0100,
6.          const BRIGHT = 0b1000,
7.      }
8.  }
9.
10. lazy_static! {
11.     static ref FIB_100: u32 = {
12.         fn fib(a: u32) -> u32 {
13.             match a {
14.                 0 => 0,
15.                 1 => 1,
```

```
                a => fib(a-1) + fib(a-2)
            }
        }

        fib(100)
    };
}

fn main() {
    let colors = vec![RED, GREEN, BLUE];
    println!("Hello, World!");
}
```

虽然看起来上述调用包含了各式各样的Rust代码，但对语法分析器来说，它们仅仅是堆毫无意义的标记树。为了让事情变得更清晰，我们把所有这些句法“黑盒”用⬚代替，仅剩下：

```
bitflags! ⬚

lazy_static! ⬚

fn main() {
    let colors = vec! ⬚;
    println! ⬚;
}
```

再次重申，语法分析器对⬚不作任何假设；它记录黑盒所包含的标记，但并不尝试理解它们。

需要记下的点：

- Rust包含多种语法扩展。我们将仅仅讨论定义在 `macro_rules!` 结构中的宏。
- 当遇见形如 `$name! $arg` 的结构时，该结构并不一定是宏，可能是其它语言扩展。
- 所有宏的输入都是非叶节点的单个标记树。
- 宏(其实所有一般意义上的语法扩展)都将作为抽象语法树的一部分被解析。

> 脚注：接下来(包括下一节)将提到的某些内容将适用于一般性的语法扩展。^作者很懒

最后一点最为重要，它带来了一些深远的影响。由于宏将被解析进AST中，它们将仅仅只能出现在那些支持它们出现的位置。具体来说，宏能在如下位置出现：

- 模式(pattern)中
- 语句(statement)中
- 表达式(expression)中
- 条目(item)中

- `impl` 块中

一些并不支持的位置包括:

- 标识符(identifier)中
- `match` 臂中
- 结构体的字段中
- 类型中[^类型宏]

[^类型宏]: 在非稳定Rust中可以通过 `#![feature(type_macros)]` 使用类型宏。见Issue #27336。

绝对没有任何在上述位置以外的地方使用宏的可能。

展开

展开相对简单。编译器在生成AST之后，对程序进行语义理解之前的某个时间点，将会对所有宏进行展开。

这一过程包括，遍历AST，定位所有宏调用，并将它们用其展开进行替换。在非宏的语法扩展情境中，此过程具体如何发生根据具体情境各有不同。但所有语法扩展在展开完成之后所经历的历程都与宏所经历的相同。

每当编译器遇见一个语法扩展，都会根据上下文决定一个语法元素集。该语法扩展的展开结果应能被顺利解析为集合中的某个元素。举例来说，如果在模组作用域内调用了宏，那么编译器就会尝试将该宏的展开结果解析为一个表示某项条目(item)的AST节点。如果在需要表达式的位置调用了宏，那么编译器就会尝试将该宏的展开结果解析为一个表示表达式的AST节点。

事实上，语义扩展能够被转换成以下任意一种：

- 一个表达式，
- 一个模式，
- 0或多个条目，
- 0或多个 `impl` 条目，
- 0或多个语句。

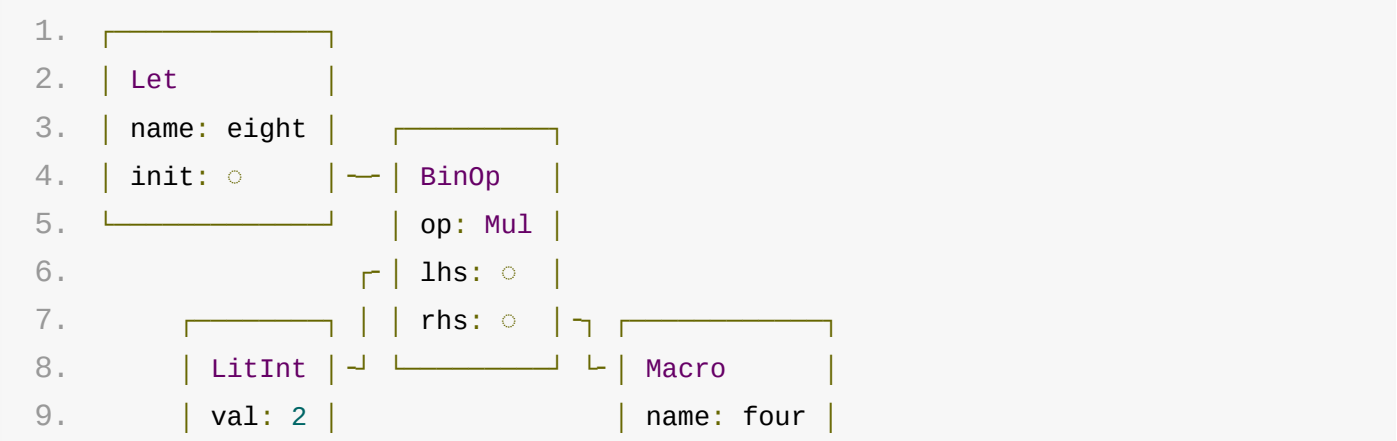
换句话讲，宏调用所在的位置，决定了该宏展开之后的结果被解读的方式。

编译器将把AST中表示宏调用的节点用其宏展开的输出节点完全替换。这一替换是结构性(structural)的，而非织构性(textural)的。

举例来说：

```
let eight = 2 * four!();
```

我们可将这部分AST表示为：



```
     └────────┘                │ body: ()  │
                               └───────────┘
```

根据上下文， `four!()` 必须展开成一个表达式（初始化语句只可能是表达式）。因此，无论实际展开结果如何，它都将被解读成一个完整的表达式。此处我们假设， `four!` 的定义保证它被展开为表达式 `1 + 3` 。故而，展开这一宏调用将使整个AST变为

```
┌─────────────┐
│ Let         │
│ name: eight │   ┌─────────┐
│ init: ○     ├╌╌╌┤ BinOp   │
└─────────────┘   │ op: Mul │
                ┌╌┤ lhs: ○  │
     ┌────────┐ ╎ │ rhs: ○  ├╌┐ ┌─────────┐
     │ LitInt ├╌┘ └─────────┘ └╌┤ BinOp   │
     │ val: 2 │                 │ op: Add │
     └────────┘               ┌╌┤ lhs: ○  │
               ┌────────┐     ╎ │ rhs: ○  ├╌┐ ┌────────┐
               │ LitInt ├╌╌╌╌╌┘ └─────────┘ └╌┤ LitInt │
               │ val: 1 │                     │ val: 3 │
               └────────┘                     └────────┘
```

这又能被重写成

```
let eight = 2 * (1 + 3);
```

注意到虽然表达式本身不包含括号，我们仍加上了它们。这是因为，编译器总是将宏展开结果作为完整的AST节点对待，而不是仅仅作为一列标记。换句话说，即便不显式地把复杂的表达式用括号包起来，编译器也不可能“错意”宏替换的结果，或者改变求值顺序。

理解这一点——宏展开被当作AST节点看待——非常重要，它表明：

- 宏调用不仅可用的位置有限，其展开结果也只可能跟语法分析器在该位置所预期的AST节点种类相符合。
- 因此，宏必定无法展开成不完整或不合语法的架构。

有关展开还有一条值得注意：如果某个语法扩展的展开结果包含了另一次语法扩展调用，那会怎么样？例如，上述 `four!` 如果被展开成了 `1 + three!()` ，会发生什么？

```
let x = four!();
```

展开成：

```
let x = 1 + three!();
```

编译器将会检查扩展结果中是否包含更多的宏调用；如果有，它们将被进一步展开。因此，上述AST节点将被再次展开成：

```
let x = 1 + 3;
```

此处我们了解到，展开是按“趟”发生的；要多少趟才能完全展开所有调用，那就会展开多少趟。

嗯，也不全是如此。事实上，编译器为此设置了一个上限。它被称作宏递归上限，默认值为32.如果第32次展开结果仍然包含宏调用，编译器将会终止并返回一个递归上限溢出的错误信息。

此上限可通过属性 `#![recursion_limit="…"]` 被改写，但这种改写必须是crate级别的。 一般来讲，可能的话最好还是尽量让宏展开递归次数保持在默认值以下。

macro_rules!

有了这些知识，我们终于可以引入 `macro_rules!` 了。 如前所述， `macro_rules!` 本身就是一个语法扩展，也就是说它并不是Rust语法的一部分。它的形式如下：

```
macro_rules! $name {
    $rule0 ;
    $rule1 ;
    // …
    $ruleN ;
}
```

至少得有一条规则，最后一条规则后面的分号可被省略。

每条“规则”(`rule`)都形如：

```
    ($pattern) => {$expansion}
```

实际上，分组符号可以是任意一种，选用这种(`pattern` 外小括号、 `expansion` 外花括号)只是出于传统。

如果你好奇的话， `macro_rules!` 的调用将被展开为空。至少可以说，在AST中它被展开为空。它所影响的是编译器内部的结构，以将该宏注册进系统中去。因此，技术上讲你可以在任何一个空展开合法的位置插入 `macro_rules!` 的调用。

匹配

当一个宏被调用时，对应的 `macro_rules` 解释器将一一依序检查规则。对每条规则，它都将尝试将输入标记树的内容与该规则的 `pattern` 进行匹配。某个模式必须与输入完全匹配才能被选中为匹配项。

如果输入与某个模式相匹配，则该调用项将被相应的 `expansion` 内容所取代；否则，将尝试匹配下条规则。如果所有规则均匹配失败，则宏展开会失败并报错。

最简单的例子是空模式：

```
macro_rules! four {
    () => {1 + 3};
}
```

它将且仅将匹配到空的输入(即 `four!()` , `four![]` 或 `four!{}`)。

注意调用所用的分组标记并不需要匹配定义时采用的分组标记。也就是说，你可以通过 `four![]` 调用上述宏，此调用仍将被视作匹配。只有调用时的输入内容才会被纳入匹配考量范围。

模式中也可以包含字面标记树。这些标记树必须被完全匹配。将整个对应标记树在相应位置写下即可。比如，为匹配标记序列 `4 fn ['spang "whammo"] @_@` ，我们可以使用：

```
1. macro_rules! gibberish {
2.     (4 fn ['spang "whammo"] @_@) => {...};
3. }
```

捕获

宏模式中还可以包含捕获。这允许输入匹配在某种通用语法基础上进行，并使得结果被捕获进某个变量中。此变量可在输出中被替换使用。

捕获由 `$` 符号紧跟一个标识符(identifier)紧跟一个冒号(`:`)紧跟捕获种类组成。捕获种类须是如下之一：

- `item` : 条目，比如函数、结构体、模组等。
- `block` : 区块(即由花括号包起的一些语句加上/或是一项表达式)。
- `stmt` : 语句
- `pat` : 模式
- `expr` : 表达式
- `ty` : 类型
- `ident` : 标识符
- `path` : 路径 (例如 `foo` , `::std::mem::replace` , `transmute::<_, int>` , …)
- `meta` : 元条目，即被包含在 `#[...]` 及 `#![...]` 属性内的东西。
- `tt` : 标记树

举例来说，下列宏将其输入捕获为一个表达式：

```
1. macro_rules! one_expression {
2.     ($e:expr) => {...};
3. }
```

Rust编译器的语法转义器将保证捕获的“准确性”。一个 `expr` 捕获总是会捕获到一个对当前Rust版本来说完整、有效的表达式。

你可以将字面标记树与捕获混合使用，但有些限制(接下来将阐明它们)。

在扩展过程中，对于某捕获 `$name:kind` ，我们可以通过在 `expansion` 中写下 `$name` 来使用它。比如：

```
macro_rules! times_five {
    ($e:expr) => {5 * $e};
}
```

如同宏扩展本身一样，每一处捕获也都将被替换为一个完整的AST节点。也就是说，在上例中无论 `$e` 所捕获的是怎样的标记序列，它总会被解读成一个完整的表达式。

在一条模式中也可以出现多次捕获：

```
macro_rules! multiply_add {
    ($a:expr, $b:expr, $c:expr) => {$a * ($b + $c)};
}
```

重复

模式中可以包含重复。这使得匹配标记序列成为可能。重复的一般形式为 `$ ( ... ) sep rep` 。

- `$` 是字面标记。
- `( ... )` 代表了将要被重复匹配的模式，由小括号包围。
- `sep` 是一个可选的分隔标记。常用例子包括 `,` 和 `;` 。
- `rep` 是重复控制标记。当前有两种选择，分别是 `*` (代表接受0或多次重复)以及 `+` (代表1或多次重复)。目前没有办法指定“0或1”或者任何其它更加具体的重复计数或区间。

重复中可以包含任意有效模式，包括字面标记树，捕获，以及其它的重复。

在扩展部分，重复也采用相同的语法。

举例来说，下述宏将每一个 `element` 都通过 `format!` 转换成字符串。它将匹配0或多个由逗号分隔的表达式，并分别将它们展开成一个 `Vec` 的 `push` 语句。

```
macro_rules! vec_strs {
    (
        // 重复开始：
        $(
            // 每次重复必须有一个表达式...
            $element:expr
        )
        // ...重复之间由“,”分隔...
        ,
```

```
        // ...总共重复0或多次.
        *
    ) => {
        // 为了能包含多条语句,
        // 我们将扩展部分包裹在花括号中...
        {
            let mut v = Vec::new();

            // 重复开始：
            $(
                // 每次重复将包含如下元素，其中
                // “$element”将被替换成其相应的展开...
                v.push(format!("{}", $element));
            )*

            v
        }
    };
}
#
# fn main() {
#     let s = vec_strs![1, "a", true, 3.14159f32];
#     assert_eq!(&*s, &["1", "a", "true", "3.14159"]);
# }
```

% 细枝末节

本节将介绍宏系统的一些细枝末节。你最少应该记住有这些东西存在。

再探捕获与展开

一旦语法分析器开始消耗标记以匹配某捕获，整个过程便无法停止或回溯。这意味着，下述宏的第二项规则将永远无法被匹配到，无论输入是什么样的：

```
1.  macro_rules! dead_rule {
2.      ($e:expr) => { ... };
3.      ($i:ident +) => { ... };
4.  }
```

考虑当以 `dead_rule!(x+)` 形式调用此宏时，将会发生什么。解析器将从第一条规则开始试图进行匹配：它试图将输入解析为一个表达式；第一个标记(`x`)作为表达式是有效的，第二个标记—作为二元加的节点—在表达式中也是有效的。

至此，由于输入中并不包含二元加的右手侧元素，你可能会以为，分析器将会放弃尝试这一规则，转而尝试下一条规则。实则不然：分析器将会 `panic` 并终止整个编译过程，返回一个语法错误。

由于分析器的这一特点，下面这点尤为重要：一般而言，在书写宏规则时，应从最具体的开始写起，依次写至最不具体的。

为防范未来宏输入的解读方式改变所可能带来的句法影响， `macro_rules!` 对各式捕获之后所允许的内容施加了诸多限制。在Rust1.3下，完整的列表如下：

- `item` ：任何标记
- `block` ：任何标记
- `stmt` ： `=>` 、 `,` 、 `;`
- `pat` ： `=>` 、 `,` 、 `=` 、 `if` 、 `in`
- `expr` ： `=>` 、 `,` 、 `;`
- `ty` ： `,` 、 `=>` 、 `:` 、 `=` 、 `>` 、 `;` 、 `as`
- `ident` ：任何标记
- `path` ： `,` 、 `=>` 、 `:` 、 `=` 、 `>` 、 `;` 、 `as`
- `meta` ：任何标记
- `tt` ：任何标记

此外， `macro_rules!` 通常不允许一个重复紧跟在另一重复之后，即便它们的内容并不冲突。

有一条关于替换的特征经常引人惊奇：尽管看起来很像，但替换并非基于标记(token-based)的。下例展示了这一点：

```
1.  macro_rules! capture_expr_then_stringify {
2.      ($e:expr) => {
```

```
        stringify!($e)
    };
}

fn main() {
    println!("{:?}", stringify!(dummy(2 * (1 + (3)))));
    println!("{:?}", capture_expr_then_stringify!(dummy(2 * (1 + (3)))));
}
```

注意到 `stringify!` ，这是一条内置的语法扩充，将所有输入标记结合在一起，作为单个字符串输出。

上述代码的输出将是：

```
"dummy ( 2 * ( 1 + ( 3 ) ) )"
"dummy(2 * (1 + (3)))"
```

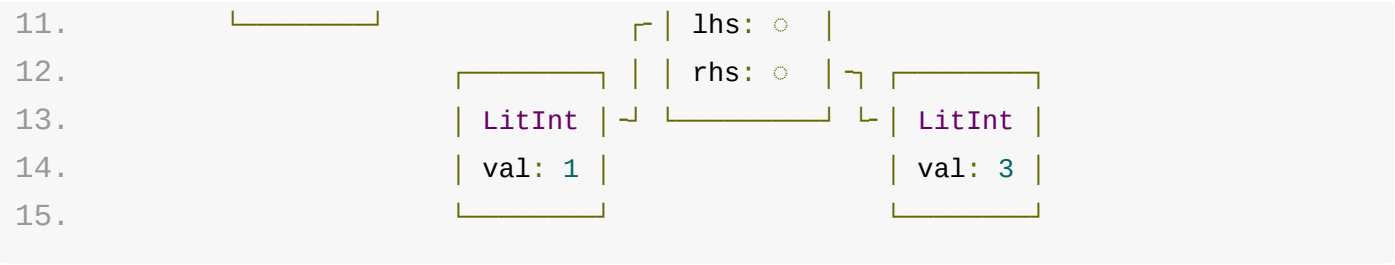
尽管二者的输入完全一致，它们的输出并不相同。这是因为，前者字符串化的是一列标记树，而后者字符串化的则是一个AST表达式节点。

我们另用一种方式展现二者的不同。第一种情况下， `stringify!` 被调用时，输入是：

```
«dummy» «(    )»
    ┌─────────┴─────────┐
     «2» «*» «(    )»
        ┌──────┴──────┐
         «1» «+» «(    )»
                  ┌─┴─┐
                   «3»
```

…而第二种情况下， `stringify!` 被调用时，输入是：

```
« »
  │ ┌──────────────┐
  └─┤ Call         │
    │ fn: dummy    │   ┌──────────┐
    │ args: ○      ├─┬─┤ BinOp    │
    └──────────────┘ │ │ op: Mul  │
                     │ │ lhs: ○   │
          ┌──────────┐ │ │ rhs: ○   ├─┐ ┌──────────┐
          │ LitInt   ├─┘ └──────────┘ └─┤ BinOp    │
          │ val: 2   │                 │ op: Add  │
```



如图所示，第二种情况下，输入仅有一棵标记树，它包含了一棵AST，这棵AST则是在解析 `capture_expr_then_stringify!` 的调用时，调用的输入经解析后所得的输出。因此，在这种情况下，我们最终看到的是字符串化AST节点所得的输出，而非字符串化标记所得的输出。

这一特征还有更加深刻的影响。考虑如下代码段：

```
macro_rules! capture_then_match_tokens {
    ($e:expr) => {match_tokens!($e)};
}

macro_rules! match_tokens {
    ($a:tt + $b:tt) => {"got an addition"};
    (($i:ident)) => {"got an identifier"};
    ($($other:tt)*) => {"got something else"};
}

fn main() {
    println!("{}\n{}\n{}\n",
        match_tokens!((caravan)),
        match_tokens!(3 + 6),
        match_tokens!(5));
    println!("{}\n{}\n{}",
        capture_then_match_tokens!((caravan)),
        capture_then_match_tokens!(3 + 6),
        capture_then_match_tokens!(5));
}
```

其输出将是：

```
got an identifier
got an addition
got something else

got something else
got something else
got something else
```

因为输入被解析为AST节点，替换所得的结果将无法析构。也就是说，你没办法检查其内容，或是再按原先相符的匹配匹配它。

接下来这个例子尤其可能让人感到不解：

```
macro_rules! capture_then_what_is {
    (#[$m:meta]) => {what_is!(#[$m])};
}

macro_rules! what_is {
    (#[no_mangle]) => {"no_mangle attribute"};
    (#[inline]) => {"inline attribute"};
    ($($tts:tt)*) => {concat!("something else (", stringify!($($tts)*), ")")};
}

fn main() {
    println!(
        "{}\n{}\n{}\n{}",
        what_is!(#[no_mangle]),
        what_is!(#[inline]),
        capture_then_what_is!(#[no_mangle]),
        capture_then_what_is!(#[inline]),
    );
}
```

输出将是：

```
no_mangle attribute
inline attribute
something else (# [ no_mangle ])
something else (# [ inline ])
```

得以幸免的捕获只有 `tt` 或 `ident` 两种。其余的任何捕获，一经替换，结果将只能被用于直接输出。

卫生性

Rust宏是部分卫生的。具体来说，对于绝大多数标识符，它是卫生的；但对泛型参数和生命周期来算，它不是。

之所以能做到“卫生”，在于每个标识符都被赋予了一个看不见的“句法上下文”。在比较两个标识符时，只有在标识符的明面名字和句法上下文都一致的情况下，两个标识符才能被视作等同。

为阐释这一点，考虑下述代码：

```
macro_rules! using_a {
    ($e:expr) => {
        {
            let a = 42;
            $e
        }
    }
}

let four = using_a!(a / 10);
```

我们将采用背景色来表示句法上下文。现在，将上述宏调用展开如下：

```
let four = {
    let a = 42;
    a / 10
};
```

首先记起， `macro_rules!` 的调用在展开过程中等同于消失。

其次，如果我们现在就尝试编译上述代码，编译器将报如下错误：

```
<anon>:11:21: 11:22 error: unresolved name `a`
<anon>:11 let four = using_a!(a / 10);
```

注意到宏在展开后背景色(即其句法上下文)发生了改变。每处宏展开均赋予其内容一个新的、独一无二的上下文。故而，在展开后的代码中实际上存在两个不同的 `a` ，分别有不同的句法上下文。
即， `a` 与 `a` 并不相同，即它们便看起来很像。

尽管如此，被替换进宏展开中的标记仍然保持着它们原有的句法上下文(因它们是被提供给这宏的，并非这宏本身的一部分)。因此，我们作出如下修改：

```
macro_rules! using_a {
    ($a:ident, $e:expr) => {
        {
            let $a = 42;
            $e
        }
    }
}

let four = using_a!(a, a / 10);
```

此宏在展开后将变成：

```
let four = {
    let a = 42;
    a / 10
};
```

因为只用了一种 `a` ，编译器将欣然接受此段代码。

不是标识符的标识符

有两个标记，当你撞见时，很有可能最终认为它们是标识符，但实际上它们不是。然而正是这些标记，在某些情况下又的确是标识符。

第一个是 `self` 。毫无疑问，它是一个关键词。在一般的Rust代码中，不可能出现把它解读成标识符的情况；但在宏中这种情况则有可能发生：

```
macro_rules! what_is {
    (self) => {"the keyword `self`"};
    ($i:ident) => {concat!("the identifier `", stringify!($i), "`")};
}

macro_rules! call_with_ident {
    ($c:ident($i:ident)) => {$c!($i)};
}

fn main() {
    println!("{}", what_is!(self));
    println!("{}", call_with_ident!(what_is(self)));
}
```

上述代码的输出将是：

```
the keyword `self`
the keyword `self`
```

但这没有任何道理！ `call_with_ident!` 要求一个标识符，而且它的确匹配到了，还成功替换了！所以， `self` 同时是一个关键词，但又不是。你可能会想，好吧，但这鬼东西哪里重要呢？看看这个：

```
macro_rules! make_mutable {
    ($i:ident) => {let mut $i = $i;};
}

struct Dummy(i32);

impl Dummy {
    fn double(self) -> Dummy {
        make_mutable!(self);
        self.0 *= 2;
```

```
        self
    }
}
#
# fn main() {
#     println!("{:?}", Dummy(4).double().0);
# }
```

编译它会失败，并报错：

```
<anon>:2:28: 2:30 error: expected identifier, found keyword `self`
<anon>:2     ($i:ident) => {let mut $i = $i;};
                                    ^~
```

所以说，宏在匹配的时候，会欣然把 `self` 当作标识符接受，进而允许你把 `self` 带到那些实际上没办法使用的情况中去。但是，也成吧，既然得同时记住 `self` 既是关键词又是标识符，那下面这个讲道理应该可行，对吧？

```
macro_rules! make_self_mutable {
    ($i:ident) => {let mut $i = self;};
}

struct Dummy(i32);

impl Dummy {
    fn double(self) -> Dummy {
        make_self_mutable!(mut_self);
        mut_self.0 *= 2;
        mut_self
    }
}
#
# fn main() {
#     println!("{:?}", Dummy(4).double().0);
# }
```

实际上也不行，编译错误变成：

```
<anon>:2:33: 2:37 error: `self` is not available in a static method. Maybe a
`self` argument is missing? [E0424]
<anon>:2     ($i:ident) => {let mut $i = self;};
                                        ^~~~
```

这同样也没有任何道理。它明明不在静态方法里。这简直就像是在抱怨说，它看见的两个 `self` 不是同一个 `self` … 就搞得像关键词 `self` 也有卫生性一样，类似…标识符。

```
macro_rules! double_method {
    ($body:expr) => {
        fn double(mut self) -> Dummy {
            $body
        }
    };
}

struct Dummy(i32);

impl Dummy {
    double_method! {{
        self.0 *= 2;
        self
    }}
}
#
# fn main() {
#     println!("{:?}", Dummy(4).double().0);
# }
```

还是报同样的错。那这个如何：

```
macro_rules! double_method {
    ($self_:ident, $body:expr) => {
        fn double(mut $self_) -> Dummy {
            $body
        }
    };
}

struct Dummy(i32);

impl Dummy {
    double_method! {self, {
        self.0 *= 2;
        self
    }}
```

```
}
#
# fn main() {
#     println!("{:?}", Dummy(4).double().0);
# }
```

终于管用了。所以说， `self` 是关键词，但当它想的时候，它同时也能是一个标识符。那么，相同的道理对类似的其它东西有用吗？

```
macro_rules! double_method {
    ($self_:ident, $body:expr) => {
        fn double($self_) -> Dummy {
            $body
        }
    };
}

struct Dummy(i32);

impl Dummy {
    double_method! {_, 0}
}
#
# fn main() {
#     println!("{:?}", Dummy(4).double().0);
# }
```

```
<anon>:12:21: 12:22 error: expected ident, found _
<anon>:12     double_method! {_, 0}
                             ^
```

哈，当然不行。 `_` 是一个关键词，在模式以及表达式中有效，但不知为何，并不像 `self` ，它并不是一个标识符；即便它—如同 `self` —从定义上讲符合标识符的特性。

你可能觉得，既然 `_` 在模式中有效，那换成 `$self_:pat` 是不是就能一石二鸟了呢？可惜了，也不行，因为 `self` 不是一个有效的模式。真棒。

如果你真想同时匹配这两个标记，仅有的办法是换用 `tt` 来匹配。

调试

`rustc` 提供了一些工具用来调试宏。其中，最有用的之一是 `trace_macros!` 。它会指示编译器，在每一个宏调用被展开之前将其转印出来。例如，给定下列代码：

```
# // Note: make sure to use a nightly channel compiler.
#![feature(trace_macros)]

macro_rules! each_tt {
    () => {};
    ($_tt:tt $($rest:tt)*) => {each_tt!($($rest)*);};
}

each_tt!(foo bar baz quux);
trace_macros!(true);
each_tt!(spim wak plee whum);
trace_macros!(false);
each_tt!(trom qlip winp xod);
#
# fn main() {}
```

编译输出将包含：

```
each_tt! { spim wak plee whum }
each_tt! { wak plee whum }
each_tt! { plee whum }
each_tt! { whum }
each_tt! {  }
```

它在调试递归很深的宏时尤其有用。同时，它可以在命令提示符中被打开，在编译指令中附加 `-Z trace-macros` 即可。

另一有用的宏是 `log_syntax!` 。它将使得编译器输出所有经过编译器处理的标记。举个例子，下述代码可以让编译器唱一首歌：

```
# // Note: make sure to use a nightly channel compiler.
#![feature(log_syntax)]

macro_rules! sing {
    () => {};
```

```
    ($tt:tt $($rest:tt)*) => {log_syntax!($tt); sing!($($rest)*);};
}

sing! {
    ^ < @ < . @ *
    '\x08' '{' '"' _ # ' '
    - @ '$' && / _ %
    ! ( '\t' @ | = >
    ; '\x08' '\'' + '$' ? '\x7f'
    , # '"' ~ | ) '\x07'
}
#
# fn main() {}
```

比起 `trace_macros!` 来说，它能够做一些更有针对性的调试。

有时问题出在宏展开后的结果里。对于这种情况，可用编译命令 `--pretty` 来勘察。给出下列代码：

```
// Shorthand for initialising a `String`.
macro_rules! S {
    ($e:expr) => {String::from($e)};
}

fn main() {
    let world = S!("World");
    println!("Hello, {}!", world);
}
```

并用如下编译命令进行编译，

```
rustc -Z unstable-options --pretty expanded hello.rs
```

将输出如下内容(略经修改以符合排版)：

```
#![feature(no_std, prelude_import)]
#![no_std]
#[prelude_import]
use std::prelude::v1::*;
#[macro_use]
extern crate std as std;
// Shorthand for initialising a `String`.
fn main() {
```

```
 9.     let world = String::from("World");
10.     ::std::io::_print(::std::fmt::Arguments::new_v1(
11.         {
12.             static __STATIC_FMTSTR: &'static [&'static str]
13.                 = &["Hello, ", "!\n"];
14.             __STATIC_FMTSTR
15.         },
16.         &match (&world,) {
17.              (__arg0,) => [
18.                 ::std::fmt::ArgumentV1::new(__arg0, ::std::fmt::Display::fmt)
19.             ],
20.         }
21.     ));
22. }
```

`--pretty` 还有其它一些可用选项，可通过 `rustc -Z unstable-options --help -v` 来列出。此处并不提供该选项表；因为，正如指令本身所暗示的，表中的一切内容在任何时间点都有可能发生改变。

作用域

宏作用域的决定方式可能有一点反直觉。首先就与语言剩下的所有部分都不同的是，宏在子模组中仍然可见。

```
macro_rules! X { () => {}; }
mod a {
    X!(); // 已被定义
}
mod b {
    X!(); // 已被定义
}
mod c {
    X!(); // 已被定义
}
# fn main() {}
```

> 注意：即使子模组的内容处在不同文件中，这些例子中所述的行为仍然保持不变。

其次，同样与语言剩下的所有部分不同，宏只有在其定义之后可见。下例展示了这一点。同时注意到，它也展示了宏不会“漏出”其定义所在的域：

```
mod a {
    // X!(); // 未被定义
}
mod b {
    // X!(); // 未被定义
    macro_rules! X { () => {}; }
    X!(); // 已被定义
}
mod c {
    // X!(); // 未被定义
}
# fn main() {}
```

需要阐明的是，即便宏定义被移至外围域，此顺序依赖行为仍旧不变：

```
mod a {
    // X!(); // 未被定义
}
```

```
macro_rules! X { () => {}; }
mod b {
    X!(); // 已被定义
}
mod c {
    X!(); // 已被定义
}
# fn main() {}
```

然而，对于宏们自身来说，此依赖行为不存在：

```
mod a {
    // X!(); // 未被定义
}
macro_rules! X { () => { Y!(); }; }
mod b {
    // X!(); // 已被定义，但Y!未被定义
}
macro_rules! Y { () => {}; }
mod c {
    X!(); // 均已被定义
}
# fn main() {}
```

可通过 `#[macro_use]` 属性将宏导出模组：

```
mod a {
    // X!(); // 未被定义
}
#[macro_use]
mod b {
    macro_rules! X { () => {}; }
    X!(); // 已被定义
}
mod c {
    X!(); // 已被定义
}
# fn main() {}
```

注意到这一特性可能会产生一些奇怪的后果，因为宏中的标识符只有在宏展开的过程中才会被解析。

```
mod a {
```

```
    // X!(); // 未被定义
}
#[macro_use]
mod b {
    macro_rules! X { () => { Y!(); }; }
    // X!(); // 已被定义，但Y!并未被定义
}
macro_rules! Y { () => {}; }
mod c {
    X!(); // 均已被定义
}
# fn main() {}
```

让情形变得更加复杂的是，当 `#[macro_use]` 被作用于 `extern crate` 时，其行为又会发生进一步变化：此类声明从效果上看，类似于被放在了整个模组的顶部。因此，假设在某个 `extern crate mac` 中定义了 `X!` ，则有：

```
mod a {
    // X!(); // 已被定义，但Y!并未被定义
}
macro_rules! Y { () => {}; }
mod b {
    X!(); // 均已被定义
}
#[macro_use] extern crate macs;
mod c {
    X!(); // 均已被定义
}
# fn main() {}
```

最后，注意这些有关作用域的行为同样适用于函数，除了 `#[macro_use]` 以外(它并不适用)：

```
macro_rules! X {
    () => { Y!() };
}

fn a() {
    macro_rules! Y { () => {"Hi!"} }
    assert_eq!(X!(), "Hi!");
    {
        assert_eq!(X!(), "Hi!");
        macro_rules! Y { () => {"Bye!"} }
```

```
11.         assert_eq!(X!(), "Bye!");
12.     }
13.     assert_eq!(X!(), "Hi!");
14. }
15.
16. fn b() {
17.     macro_rules! Y { () => {"One more"} }
18.     assert_eq!(X!(), "One more");
19. }
20. #
21. # fn main() {
22. #     a();
23. #     b();
24. # }
```

由于前述种种规则，一般来说，建议将所有应对整个 `crate` 均可见的宏的定义置于根模组的最顶部，借以确保它们一直可用。

导入/导出

有两种将宏暴露给更广范围的方法。第一种是采用 `#[macro_use]` 属性。它不仅适用于模组，同样适用于 `extern crate` 。例如：

```
#[macro_use]
mod macros {
    macro_rules! X { () => { Y!(); } }
    macro_rules! Y { () => {} }
}

X!();
#
# fn main() {}
```

可通过 `#[macro_export]` 将宏从当前 `crate` 导出。注意，这种方式无视所有可见性设定。

定义库包 `macs` 如下：

```
mod macros {
    #[macro_export] macro_rules! X { () => { Y!(); } }
    #[macro_export] macro_rules! Y { () => {} }
}

// X!和Y!并非在此处定义的，但它们**的确**被
// 导出了，即便macros并非pub。
```

则下述代码将成立：

```
X!(); // X!已被定义
#[macro_use] extern crate macs;
X!();
#
# fn main() {}
```

注意只有在根模组中，才可将 `#[macro_use]` 用于 `extern crate` 。

最后，在从 `extern crate` 导入宏时，可显式控制导入哪些宏。可利用这一特性来限制命名空间污染，或是覆写某些特定的宏。就像这样：

```
// 只导入`X!`这一个宏
#[macro_use(X)] extern crate macs;

// X!(); // X!已被定义，但Y!未被定义

macro_rules! Y { () => {} }

X!(); // 均已被定义

fn main() {}
```

当导出宏时，常常出现的情况是，宏定义需要其引用所在 `crate` 内的非宏符号。由于 `crate` 可能被重命名等，我们可以使用一个特殊的替换变量： `$crate` 。它总将被扩展为宏定义所在的 `crate` 在当前上下文中的绝对路径(比如 `:: macs`)。

注意这招并不适用于宏，因为通常名称的决定进程并不适用于宏。也就是说，你没办法采用类似 `$crate::Y!` 的代码来引用某个自己 `crate` 里的特定宏。结合采用 `#[macro_use]` 做到的选择性导入，我们得出：在宏被导入进其它 `crate` 时，当前没有办法保证其定义中的其它任一给定宏也一定可用。

推荐的做法是，在引用非宏名称时，总是采用绝对路径。这样可以最大程度上避免冲突，包括跟标准库中名称的冲突。

宏，实践介绍

本章节将通过一个相对简单、可行的例子来介绍Rust的“示例宏”系统。我们将不会试图解释整个宏系统错综复杂的构造；而是试图让读者能够舒适地了解宏的书写方式，以及为何如斯。

在Rust官方教程中也有一章讲解宏(中文版)，同样提供了高层面的讲解。同时，本书也有一章更富条理的介绍，旨在详细阐释宏系统。

一点背景知识

> 注意：别慌！我们通篇只会涉及到下面这一点点数学。如果想直接看重点，本小节可被安全跳过。

如果你不了解，所谓“递推(recurrence)关系”是指这样一个序列，其中的每个值都由先前的一个或多个值决定，并最终由一个或多个初始值完全决定。举例来说，Fibonacci数列可被定义为如下关系：

$$F_n = 0, 1, \ldots, F_{n-1} + F_{n-2}$$

即，序列的前两个数分别为0和1，而第3个则为 $F_0 + F_1 = 0 + 1 = 1$ ，第4个为 $F_1 + F_2 = 1 + 1 = 2$ ，依此类推。

由于这列值可以永远持续下去，定义一个 `fibonacci` 的求值函数略显困难。显然，返回一整列值并不实际。我们真正需要的，应是某种具有怠惰求值性质的东西—只在必要的时候才进行运算求值。

在Rust中，这样的需求表明，是 `Iterator` 派上用场的时候了。实现迭代器并不十分困难，但比较繁琐：你得自定义一个类型，弄明白该在其中存储什么，然后为它实现 `Iterator` `trait`。

其实，递推关系足够简单；几乎所有的递推关系都可被抽象出来，变成一小段由宏驱动的代码生成机制。

好了，说得已经足够多了，让我们开始干活吧。

构建过程

通常来说，在构建新宏时，我所做的第一件事，是决定宏调用的形式。在我们当前所讨论的情况下，我的初次尝试是这样：

```
let fib = recurrence![a[n] = 0, 1, ..., a[n-1] + a[n-2]];

for e in fib.take(10) { println!("{}", e) }
```

以此为基点，我们可以向宏的定义迈出第一步，即便在此时我们尚不了解该宏的展开部分究竟是什么样子。此步骤的用处在于，如果在此处无法明确如何解析输入语法，那就可能意味着，整个宏的构思需要改变。

```
macro_rules! recurrence {
    ( a[n] = $($inits:expr),+ , ... , $recur:expr ) => { /* ... */ };
}
# fn main() {}
```

假装你并不熟悉相应的语法，让我来解释。上述代码块使用 `macro_rules!` 系统定义了一个宏，称为 `recurrence!` 。此宏仅包含一条解析规则，它规定，此宏必须依次匹配下列项目：

- 一段字面标记序列， `a` `[` `n` `]` `=` ；
- 一段重复 (`$( ... )`)序列，由 `,` 分隔，允许重复一或多次(`+`)；重复的内容允许：
  - 一个有效的表达式，它将被捕获至变量 `inits` (`$inits:expr`)
- 又一段字面标记序列， `...` `,` ；
- 一个有效的表达式，将被捕获至变量 `recur` (`$recur:expr`)。

最后，规则声明，如果输入被成功匹配，则对该宏的调用将被标记序列 `/* ... */` 替换。

值得注意的是， `inits` ，如它命名采用的复数形式所暗示的，实际上包含所有成功匹配进此重复的表达式，而不仅是第一或最后一个。不仅如此，它们将被捕获成一个序列，而不是—举例说—把它们不可逆地粘贴在一起。还注意到，可用 `*` 替换 `+` 来表示允许“0或多个”重复。宏系统并不支持“0或1个”或任何其它更加具体的重复形式。

作为练习，我们将采用上面提及的输入，并研究它被处理的过程。“位置”列将揭示下一个需要被匹配的句法模式，由“⌂”标出。注意在某些情况下下一个可用元素可能存在多个。“输入”将包括所有尚未被消耗的标记。 `inits` 和 `recur` 将分别包含其对应绑定的内容。

位置	输入	inits	recur
`a[n] = $($inits:expr),+ , … , $recur:expr` ⌂	`a[n] = 0, 1, …, a[n-1] + a[n-2]`		

a[n] = $($inits:expr),+ , … , $recur:expr ⌂	[n] = 0, 1, …, a[n-1] + a[n-2]		
a[n] = $($inits:expr),+ , … , $recur:expr ⌂	n] = 0, 1, …, a[n-1] + a[n-2]		
a[n] = $($inits:expr),+ , … , $recur:expr ⌂	] = 0, 1, …, a[n-1] + a[n-2]		
a[n] = $($inits:expr),+ , … , $recur:expr ⌂	= 0, 1, …, a[n-1] + a[n-2]		
a[n] = $($inits:expr),+ , … , $recur:expr ⌂	0, 1, …, a[n-1] + a[n-2]		
a[n] = $($inits:expr),+ , … , $recur:expr ⌂	0, 1, …, a[n-1] + a[n-2]		
a[n] = $($inits:expr),+ , … , $recur:expr ⌂ ⌂	, 1, …, a[n-1] + a[n-2]	0	
注意：这有两个 ⌂，因为下个输入标记既能匹配 重复元素间的分隔符逗号，也能匹配 标志重复结束的逗号。宏系统将同时追踪这两种可能，直到决定具体选择为止。			
a[n] = $($inits:expr),+ , … , $recur:expr ⌂ ⌂	1, …, a[n-1] + a[n-2]	0	
a[n] = $($inits:expr),+ , … , $recur:expr ⌂ ⌂ ⏏	, …, a[n-1] + a[n-2]	0 , 1	
注意：第三个被划掉的记号表明，基于上个被消耗的标记，宏系统排除了一项先前存在的可能。			
a[n] = $($inits:expr),+ , … , $recur:expr ⌂ ⌂	…, a[n-1] + a[n-2]	0 , 1	
a[n] = $($inits:expr),+ , … , $recur:expr ⏏ ⌂	, a[n-1] + a[n-2]	0 , 1	
a[n] = $($inits:expr),+ , … , $recur:expr ⌂	a[n-1] + a[n-2]	0 , 1	
a[n] = $($inits:expr),+ , … , $recur:expr ⌂		0 , 1	a[n-1] + a[n-2]
注意：这一步表明，类似$recur:expr的绑定将消耗一个完整的表达式。此处，究竟什么算是一个完整的表达式，将由编译器决定。稍后我们会谈到语言其它部分的类似行为。			

从此表中得到的最关键收获在于，宏系统会依次尝试将提供给它的每个标记当作输入，与提供给它的每条规则进行匹配。我们稍后还将谈回到这一“尝试”。

接下来我们首先将写出宏调用完全展开后的形态。我们想要的结构类似：

```
let fib = {
    struct Recurrence {
        mem: [u64; 2],
        pos: usize,
    }
```

它就是我们实际使用的迭代器类型。其中， `mem` 负责存储最近算得的两个斐波那契值，保证递推计算能够顺利进行； `pos` 则负责记录当前的 `n` 值。

> 附注：此处选用 `u64` 是因为，对斐波那契数列来说，它已经“足够”了。先不必担心它是否适用于其它数列，我们会提到这一点的。

```
    impl Iterator for Recurrence {
        type Item = u64;
```

```
3.
4.          #[inline]
5.          fn next(&mut self) -> Option<u64> {
6.              if self.pos < 2 {
7.                  let next_val = self.mem[self.pos];
8.                  self.pos += 1;
9.                  Some(next_val)
```

我们需要这个 `if` 分支来返回序列的初始值，没什么花哨。

```
1.              } else {
2.                  let a = /* something */;
3.                  let n = self.pos;
4.                  let next_val = (a[n-1] + a[n-2]);
5.
6.                  self.mem.TODO_shuffle_down_and_append(next_val);
7.
8.                  self.pos += 1;
9.                  Some(next_val)
10.             }
11.         }
12.     }
```

这段稍微难办一点。对于具体如何定义 `a` ，我们稍后再提。 `TODO_shuffle_down_and_append` 的真面目也将留到稍后揭晓；我们想让它做到：将 `next_val` 放至数组末尾，并将数组中剩下的元素依次前移一格，最后丢掉原先的首元素。

```
1.
2.      Recurrence { mem: [0, 1], pos: 0 }
3.  };
4.
5.  for e in fib.take(10) { println!("{}", e) }
```

最后，我们返回一个该结构的实例。在随后的代码中，我们将用它来进行迭代。综上所述，完整的展开应该如下：

```
1.  let fib = {
2.      struct Recurrence {
3.          mem: [u64; 2],
4.          pos: usize,
5.      }
```

```
6.
7.      impl Iterator for Recurrence {
8.          type Item = u64;
9.
10.         #[inline]
11.         fn next(&mut self) -> Option<u64> {
12.             if self.pos < 2 {
13.                 let next_val = self.mem[self.pos];
14.                 self.pos += 1;
15.                 Some(next_val)
16.             } else {
17.                 let a = /* something */;
18.                 let n = self.pos;
19.                 let next_val = (a[n-1] + a[n-2]);
20.
21.                 self.mem.TODO_shuffle_down_and_append(next_val.clone());
22.
23.                 self.pos += 1;
24.                 Some(next_val)
25.             }
26.         }
27.     }
28.
29.     Recurrence { mem: [0, 1], pos: 0 }
30. };
31.
32. for e in fib.take(10) { println!("{}", e) }
```

> 附注：是的，这样做的确意味着每次调用该宏时，我们都会重新定义并实现一个 `Recurrence` 结构。如果 `#[inline]` 属性应用得当，在最终编译出的二进制文件中，大部分冗余都将被优化掉。

在写展开部分的过程中时常检查，也是一个有效的技巧。如果在过程中发现，展开中的某些内容需要根据调用的不同发生改变，但这些内容并未被我们的宏语法定义囊括；那就要去考虑，应该怎样去引入它们。在此示例中，我们先前用过一次 `u64` ，但调用端想要的类型不一定是它；然而我们的宏语法并没有提供其它选择。因此，我们可以做一些修改。

```
1.  macro_rules! recurrence {
2.      ( a[n]: $sty:ty = $($inits:expr),+ , ... , $recur:expr ) => { /* ... */ };
3.  }
4.
5.  /*
6.  let fib = recurrence![a[n]: u64 = 0, 1, ..., a[n-1] + a[n-2]];
```

```
7.
8.  for e in fib.take(10) { println!("{}", e) }
9.  */
10. # fn main() {}
```

我们加入了一个新的捕获 `sty` ，它应是一个类型(type)。

> 附注：如果你不清楚的话，在捕获冒号之后的部分，可是几种语法匹配候选项之一。最常用的包括 `item` ， `expr` 和 `ty` 。完整的解释可在宏，彻底解析- `macro_rules!` -捕获部分找到。
>
> 还有一点值得注意：为方便语言的未来发展，对于跟在某些特定的匹配之后的标记，编译器施加了一些限制。这种情况常在试图匹配至表达式(expression)或语句(statement)时出现：它们后面仅允许跟进 `=>` ， `,` 和 `;` 这些标记之一。
>
> 完整清单可在宏，彻底解析-细枝末节-再探捕获与展开找到。

索引与移位

在此节中我们将略去一些实际上与宏的联系不甚紧密的内容。本节我们的目标是，让用户可以通过索引 `a` 来访问数列中先前的值。 `a` 应该如同一个切口，让我们得以持续访问数列中最近几个(在本例中，两个)值。

通过采用封装类，我们可以相对简单地做到这点：

```
struct IndexOffset<'a> {
    slice: &'a [u64; 2],
    offset: usize,
}

impl<'a> Index<usize> for IndexOffset<'a> {
    type Output = u64;

    #[inline(always)]
    fn index<'b>(&'b self, index: usize) -> &'b u64 {
        use std::num::Wrapping;

        let index = Wrapping(index);
        let offset = Wrapping(self.offset);
        let window = Wrapping(2);

        let real_index = index - offset + window;
        &self.slice[real_index.0]
    }
}
```

> 附注：对于新接触Rust的人来说，生命周期的概念经常需要一番思考。我们给出一些简单的解释： `'a` 和 `'b` 是生命周期参数，它们被用于记录引用(即一个指向某些数据的借用指针)的有效期。在此例中， `IndexOffset` 借用了一个指向我们迭代器数据的引用，因此，它需要记录该引用的有效期，记录者正是 `'a` 。
>
> 我们用到 `'b` ，是因为 `Index::index` 函数(下标句法正是通过此函数实现的)的一个参数也需要生命周期。 `'a` 和 `'b` 不一定在所有情况下都相同。我们并没有显式地声明 `'a` 和 `'b` 之间有任何联系，但借用检查器(borrow checker)总会确保内存安全性不被意外破坏。

`a` 地定义将随之变为：

```
let a = IndexOffset { slice: &self.mem, offset: n };
```

如何处理 `TODO_shuffle_down_and_append` 是我们现在剩下的唯一问题了。我没能在标准库中寻得可以直接使用的方法，但自己造一个出来并不难。

```
{
    use std::mem::swap;

    let mut swap_tmp = next_val;
    for i in (0..2).rev() {
        swap(&mut swap_tmp, &mut self.mem[i]);
    }
}
```

它把新值替换至数组末尾，并把其他值向前移动一位。

> 附注：采用这种做法，将使得我们的代码可同时被用于不可拷贝(non-copyable)的类型。

至此，最终起作用的代码将是：

```
macro_rules! recurrence {
    ( a[n]: $sty:ty = $($inits:expr),+ , ... , $recur:expr ) => { /* ... */ };
}

fn main() {
    /*
    let fib = recurrence![a[n]: u64 = 0, 1, ..., a[n-1] + a[n-2]];

    for e in fib.take(10) { println!("{}", e) }
    */
    let fib = {
        use std::ops::Index;

        struct Recurrence {
            mem: [u64; 2],
            pos: usize,
        }

        struct IndexOffset<'a> {
            slice: &'a [u64; 2],
            offset: usize,
        }

        impl<'a> Index<usize> for IndexOffset<'a> {
```

```
25.             type Output = u64;
26.
27.             #[inline(always)]
28.             fn index<'b>(&'b self, index: usize) -> &'b u64 {
29.                 use std::num::Wrapping;
30.
31.                 let index = Wrapping(index);
32.                 let offset = Wrapping(self.offset);
33.                 let window = Wrapping(2);
34.
35.                 let real_index = index - offset + window;
36.                 &self.slice[real_index.0]
37.             }
38.         }
39.
40.         impl Iterator for Recurrence {
41.             type Item = u64;
42.
43.             #[inline]
44.             fn next(&mut self) -> Option<u64> {
45.                 if self.pos < 2 {
46.                     let next_val = self.mem[self.pos];
47.                     self.pos += 1;
48.                     Some(next_val)
49.                 } else {
50.                     let next_val = {
51.                         let n = self.pos;
52.                         let a = IndexOffset { slice: &self.mem, offset: n };
53.                         (a[n-1] + a[n-2])
54.                     };
55.
56.                     {
57.                         use std::mem::swap;
58.
59.                         let mut swap_tmp = next_val;
60.                         for i in (0..2).rev() {
61.                             swap(&mut swap_tmp, &mut self.mem[i]);
62.                         }
63.                     }
64.
65.                     self.pos += 1;
66.                     Some(next_val)
```

```
                }
            }
        }

        Recurrence { mem: [0, 1], pos: 0 }
    };

    for e in fib.take(10) { println!("{}", e) }
}
```

注意我们改变了 `n` 与 `a` 的声明顺序，同时将它们(与递推表达式一同)用一个新区块包裹了起来。改变声明顺序的理由很明显(`n` 得在 `a` 前被定义才能被 `a` 使用)。而包裹的理由则是：如果不，借用引用 `&self.mem` 将会阻止随后的 `swap` 操作(在某物仍存在其它别名时，无法对其进行改变)。包裹区块将确保 `&self.mem` 产生的借用在彼时过期。

顺带一提，将交换 `mem` 的代码包进区块里的唯一原因，正是为了缩减 `std::mem::swap` 的可用范畴，以保持代码整洁。

如果我们直接拿上段代码来跑，将会得到：

```
0
1
1
2
3
5
8
13
21
34
```

成功了！现在，让我们把这段代码复制粘贴进宏的展开部分，并把它们原本所在的位置换成一次宏调用。这样我们得到：

```
macro_rules! recurrence {
    ( a[n]: $sty:ty = $($inits:expr),+ , ... , $recur:expr ) => {
        {
            /*
                What follows here is *literally* the code from before,
                cut and pasted into a new position.  No other changes
                have been made.
            */
```

```
9.
10.             use std::ops::Index;
11.
12.             struct Recurrence {
13.                 mem: [u64; 2],
14.                 pos: usize,
15.             }
16.
17.             struct IndexOffset<'a> {
18.                 slice: &'a [u64; 2],
19.                 offset: usize,
20.             }
21.
22.             impl<'a> Index<usize> for IndexOffset<'a> {
23.                 type Output = u64;
24.
25.                 #[inline(always)]
26.                 fn index<'b>(&'b self, index: usize) -> &'b u64 {
27.                     use std::num::Wrapping;
28.
29.                     let index = Wrapping(index);
30.                     let offset = Wrapping(self.offset);
31.                     let window = Wrapping(2);
32.
33.                     let real_index = index - offset + window;
34.                     &self.slice[real_index.0]
35.                 }
36.             }
37.
38.             impl Iterator for Recurrence {
39.                 type Item = u64;
40.
41.                 #[inline]
42.                 fn next(&mut self) -> Option<u64> {
43.                     if self.pos < 2 {
44.                         let next_val = self.mem[self.pos];
45.                         self.pos += 1;
46.                         Some(next_val)
47.                     } else {
48.                         let next_val = {
49.                             let n = self.pos;
```

```
                                let a = IndexOffset { slice: &self.mem, offset: n
50. };
51.                             (a[n-1] + a[n-2])
52.                         };
53.
54.                         {
55.                             use std::mem::swap;
56.
57.                             let mut swap_tmp = next_val;
58.                             for i in (0..2).rev() {
59.                                 swap(&mut swap_tmp, &mut self.mem[i]);
60.                             }
61.                         }
62.
63.                         self.pos += 1;
64.                         Some(next_val)
65.                     }
66.                 }
67.             }
68.
69.             Recurrence { mem: [0, 1], pos: 0 }
70.         }
71.     };
72. }
73.
74. fn main() {
75.     let fib = recurrence![a[n]: u64 = 0, 1, ..., a[n-1] + a[n-2]];
76.
77.     for e in fib.take(10) { println!("{}", e) }
78. }
```

显然，宏的捕获尚未被用到，但这点很好改。不过，如果尝试编译上述代码， `rustc` 会中止，并显示：

```
    recurrence.rs:69:45: 69:48 error: local ambiguity: multiple parsing options:
1.  built-in NTs expr ('inits') or 1 other options.
    recurrence.rs:69     let fib = recurrence![a[n]: u64 = 0, 1, ..., a[n-1] + a[n-
2.  2]];
3.                                                              ^~~
```

这里我们撞上了 `macro_rules` 的一处限制。问题出在那第二个逗号上。当在展开过程中遇见它时，编译器无法决定是该将它解析成 `inits` 中的又一个表达式，还是解析成 `...` 。很遗憾，它不够聪

明，没办法意识到 `...` 不是一个有效的表达式，所以它选择了放弃。理论上来说，上述代码应该能奏效，但当前它并不能。

> 附注：有关宏系统如何解读我们的规则，我之前的确撒了点小谎。通常来说，宏系统确实应当如我前述的那般运作，但在这里它没有。 `macro_rules` 的机制，由此看来，是存在一些小毛病的；我们得记得偶尔去做一些调控，好让它我们期许的那般运作。
>
> 在本例中，问题有两个。其一，宏系统不清楚各式各样的语法元素(如表达式)可由什么样的东西构成，或不能由什么样的东西构成；那是语法解析器的工作。其二，在试图捕获复合语法元素(如表达式)的过程中，它无法不100%地首先陷入该捕获中去。
>
> 换句话说，宏系统可以向语法解析器发出请求，让后者试图把某段输入当作表达式来进行解析；但此间无论语法解析器遇见任何问题，都将中止整个进程以示回应。目前，宏系统处理这种窘境的唯一方式，就是对任何可能产生此类问题的情境加以禁止。
>
> 好的一面在于，对于这摊子情况，没有任何人感到高兴。关键词 `macro` 早已被预留，以备未来更加严密的宏系统使用。直到那天来临之前，我们还是只得该怎么做就怎么做。

还好，修正方案也很简单：从宏句法中去掉逗号即可。出于平衡考量，我们将移除 `...` 双边的逗号：[^译注1]

```
1.  macro_rules! recurrence {
2.      ( a[n]: $sty:ty = $($inits:expr),+ ... $recur:expr ) => {
3.  //                                    ^~~ changed
4.          /* ... */
5.  #         // Cheat :D
6.  #         (vec![0u64, 1, 2, 3, 5, 8, 13, 21, 34]).into_iter()
7.      };
8.  }
9.
10. fn main() {
11.     let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-2]];
12. //                                        ^~~ changed
13.
14.     for e in fib.take(10) { println!("{}", e) }
15. }
```

成功！现在，我们该将捕获部分捕获到的内容替代进展开部分中了。

[^译注1]: 在目前的稳定版本(Rust 1.14.0)下，去掉双边逗号后的代码无法通过编译。 `rustc` 报错“error: `$inits:expr` may be followed by `...`, which is not allowed for `expr` fragments”。解决方案是将这两处逗号替换为其它字面值，如分号；与之前的捕获所用分隔符不同即可。

替换

在宏中替换你捕获到的内容相对简单，通过 `$sty:ty` 捕获到的内容可用 `$sty` 来替换。好，让我们换掉那些 `u64` 吧：

```
macro_rules! recurrence {
    ( a[n]: $sty:ty = $($inits:expr),+ ... $recur:expr ) => {
        {
            use std::ops::Index;

            struct Recurrence {
                mem: [$sty; 2],
//                    ^~~~ changed
                pos: usize,
            }

            struct IndexOffset<'a> {
                slice: &'a [$sty; 2],
//                         ^~~~ changed
                offset: usize,
            }

            impl<'a> Index<usize> for IndexOffset<'a> {
                type Output = $sty;
//                            ^~~~ changed

                #[inline(always)]
                fn index<'b>(&'b self, index: usize) -> &'b $sty {
//                                                      ^~~~ changed
                    use std::num::Wrapping;

                    let index = Wrapping(index);
                    let offset = Wrapping(self.offset);
                    let window = Wrapping(2);

                    let real_index = index - offset + window;
                    &self.slice[real_index.0]
                }
            }

            impl Iterator for Recurrence {
                type Item = $sty;
```

```
38. //                          ^~~~ changed
39. 
40.                 #[inline]
41.                 fn next(&mut self) -> Option<$sty> {
42. //                                        ^~~~ changed
43.                     /* ... */
44. #                   if self.pos < 2 {
45. #                       let next_val = self.mem[self.pos];
46. #                       self.pos += 1;
47. #                       Some(next_val)
48. #                   } else {
49. #                       let next_val = {
50. #                           let n = self.pos;
    #                           let a = IndexOffset { slice: &self.mem, offset: n
51. };
52. #                           (a[n-1] + a[n-2])
53. #                       };
54. #
55. #                       {
56. #                           use std::mem::swap;
57. #
58. #                           let mut swap_tmp = next_val;
59. #                           for i in (0..2).rev() {
60. #                               swap(&mut swap_tmp, &mut self.mem[i]);
61. #                           }
62. #                       }
63. #
64. #                       self.pos += 1;
65. #                       Some(next_val)
66. #                   }
67.                 }
68.             }
69. 
70.             Recurrence { mem: [1, 1], pos: 0 }
71.         }
72.     };
73. }
74. 
75. fn main() {
76.     let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-2]];
77. 
78.     for e in fib.take(10) { println!("{}", e) }
```

```
79. }
```

现在让我们来尝试更难的：如何将 `inits` 同时转变为字面值 `[0, 1]` 以及数组类型 `[$sty; 2]` 。首先我们试试：

```
1.              Recurrence { mem: [$($inits),+], pos: 0 }
2. //                              ^~~~~~~~~~ changed
```

此段代码与捕获的效果正好相反：将 `inits` 捕得的内容排列开来，总共有1或多次，每条内容之间用逗号分隔。展开的结果与期望一致，我们得到标记序列： `0, 1` 。

不过，通过 `inits` 转换出字面值 `2` 需要一些技巧。没有直接可行的方法，但我们可以通过另一个宏做到。我们一步一步来。

```
1. macro_rules! count_exprs {
2.     /* ??? */
3. #     () => {}
4. }
5. # fn main() {}
```

先写显而易见的情况：未给表达式时，我们期望 `count_exprs` 展开为字面值 `0` 。

```
1. macro_rules! count_exprs {
2.     () => (0);
3. //  ^~~~~~~~~~ added
4. }
5. # fn main() {
6. #     const _0: usize = count_exprs!();
7. #     assert_eq!(_0, 0);
8. # }
```

> 附注：你可能已经注意到了，这里的展开部分我用的是括号而非花括号。 `macro_rules` 其实不关心你用的是什么，只要它成对匹配即可： `( )` ， `{ }` 或 `[ ]` 。实际上，宏本身的匹配符(即紧跟宏名称后的匹配符)、语法规则外的匹配符及相应展开部分外的匹配符都可以替换。
>
> 调用宏时的括号也可被替换，但有些限制：当宏被以 `{...}` 或 `(...);` 形式调用时，它总是会被解析为一个条目(item，比如， `struct` 或 `fn` 声明)。在函数体内部时，这一特征很重要，它将消除“解析成表达式”和“解析成语句”之间的歧义。

有一个表达式的情况该怎么办？应该展开为字面值 `1` 。

```
1. macro_rules! count_exprs {
```

```
    () => (0);
    ($e:expr) => (1);
//  ^~~~~~~~~~~~~~~~ added
}
# fn main() {
#     const _0: usize = count_exprs!();
#     const _1: usize = count_exprs!(x);
#     assert_eq!(_0, 0);
#     assert_eq!(_1, 1);
# }
```

两个呢？

```
macro_rules! count_exprs {
    () => (0);
    ($e:expr) => (1);
    ($e0:expr, $e1:expr) => (2);
//  ^~~~~~~~~~~~~~~~~~~~~~~~~~ added
}
# fn main() {
#     const _0: usize = count_exprs!();
#     const _1: usize = count_exprs!(x);
#     const _2: usize = count_exprs!(x, y);
#     assert_eq!(_0, 0);
#     assert_eq!(_1, 1);
#     assert_eq!(_2, 2);
# }
```

通过递归调用重新表达，我们可将扩展部分“精简”出来：

```
macro_rules! count_exprs {
    () => (0);
    ($e:expr) => (1);
    ($e0:expr, $e1:expr) => (1 + count_exprs!($e1));
//                           ^~~~~~~~~~~~~~~~~~~~ changed
}
# fn main() {
#     const _0: usize = count_exprs!();
#     const _1: usize = count_exprs!(x);
#     const _2: usize = count_exprs!(x, y);
#     assert_eq!(_0, 0);
#     assert_eq!(_1, 1);
```

```
#     assert_eq!(_2, 2);
# }
```

这样做可行是因为，Rust可将 `1 + 1` 合并成一个常量。那么，三种表达式的情况呢？

```
macro_rules! count_exprs {
    () => (0);
    ($e:expr) => (1);
    ($e0:expr, $e1:expr) => (1 + count_exprs!($e1));
    ($e0:expr, $e1:expr, $e2:expr) => (1 + count_exprs!($e1, $e2));
//  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ added
}
# fn main() {
#     const _0: usize = count_exprs!();
#     const _1: usize = count_exprs!(x);
#     const _2: usize = count_exprs!(x, y);
#     const _3: usize = count_exprs!(x, y, z);
#     assert_eq!(_0, 0);
#     assert_eq!(_1, 1);
#     assert_eq!(_2, 2);
#     assert_eq!(_3, 3);
# }
```

> 附注：你可能会想，我们是否能翻转这些规则的排列顺序。在此情境下，可以。但在有些情况下，宏系统可能会对此挑剔。如果你发现自己有一个包含多项规则的宏系统老是报错，或给出期望外的结果；但你发誓它应该能用，试着调换一下规则的排序吧。

我们希望你现在已经能看出规律。通过匹配至一个表达式加上0或多个表达式并展开成1+a，我们可以减少规则列表的数目：

```
macro_rules! count_exprs {
    () => (0);
    ($head:expr) => (1);
    ($head:expr, $($tail:expr),*) => (1 + count_exprs!($($tail),*));
//  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ changed
}
# fn main() {
#     const _0: usize = count_exprs!();
#     const _1: usize = count_exprs!(x);
#     const _2: usize = count_exprs!(x, y);
#     const _3: usize = count_exprs!(x, y, z);
#     assert_eq!(_0, 0);
```

```
#     assert_eq!(_1, 1);
#     assert_eq!(_2, 2);
#     assert_eq!(_3, 3);
# }
```

> 仅对此例：这段代码并非计数仅有或其最好的方法。若有兴趣，稍后可以研读计数一节。

有此工具后，我们可再次修改 `recurrence` ，确定 `mem` 所需的大小。

```
// added:
macro_rules! count_exprs {
    () => (0);
    ($head:expr) => (1);
    ($head:expr, $($tail:expr),*) => (1 + count_exprs!($($tail),*));
}

macro_rules! recurrence {
    ( a[n]: $sty:ty = $($inits:expr),+ ... $recur:expr ) => {
        {
            use std::ops::Index;

            const MEM_SIZE: usize = count_exprs!($($inits),+);
//          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ added

            struct Recurrence {
                mem: [$sty; MEM_SIZE],
//                       ^~~~~~~~ changed
                pos: usize,
            }

            struct IndexOffset<'a> {
                slice: &'a [$sty; MEM_SIZE],
//                              ^~~~~~~~ changed
                offset: usize,
            }

            impl<'a> Index<usize> for IndexOffset<'a> {
                type Output = $sty;

                #[inline(always)]
                fn index<'b>(&'b self, index: usize) -> &'b $sty {
                    use std::num::Wrapping;
```

```
34.
35.                     let index = Wrapping(index);
36.                     let offset = Wrapping(self.offset);
37.                     let window = Wrapping(MEM_SIZE);
38. //                                        ^~~~~~~~ changed
39.
40.                     let real_index = index - offset + window;
41.                     &self.slice[real_index.0]
42.                 }
43.             }
44.
45.             impl Iterator for Recurrence {
46.                 type Item = $sty;
47.
48.                 #[inline]
49.                 fn next(&mut self) -> Option<$sty> {
50.                     if self.pos < MEM_SIZE {
51. //                                  ^~~~~~~~ changed
52.                         let next_val = self.mem[self.pos];
53.                         self.pos += 1;
54.                         Some(next_val)
55.                     } else {
56.                         let next_val = {
57.                             let n = self.pos;
                                let a = IndexOffset { slice: &self.mem, offset: n
58. };
59.                             (a[n-1] + a[n-2])
60.                         };
61.
62.                         {
63.                             use std::mem::swap;
64.
65.                             let mut swap_tmp = next_val;
66.                             for i in (0..MEM_SIZE).rev() {
67. //                                      ^~~~~~~~ changed
68.                                 swap(&mut swap_tmp, &mut self.mem[i]);
69.                             }
70.                         }
71.
72.                         self.pos += 1;
73.                         Some(next_val)
74.                     }
```

```
                }
            }

            Recurrence { mem: [$($inits),+], pos: 0 }
        }
    };
}
/* ... */
#
# fn main() {
#     let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-2]];
#
#     for e in fib.take(10) { println!("{}", e) }
# }
```

完成之后，我们开始替换最后的 `recur` 表达式。

```
# macro_rules! count_exprs {
#     () => (0);
#     ($head:expr $(, $tail:expr)*) => (1 + count_exprs!($($tail),*));
# }
# macro_rules! recurrence {
#     ( a[n]: $sty:ty = $($inits:expr),+ ... $recur:expr ) => {
#         {
            const MEMORY: uint = count_exprs!($($inits),+);
            struct Recurrence {
                mem: [$sty; MEMORY],
                pos: uint,
            }
            struct IndexOffset<'a> {
                slice: &'a [$sty; MEMORY],
                offset: uint,
            }
            impl<'a> Index<uint, $sty> for IndexOffset<'a> {
                #[inline(always)]
                fn index<'b>(&'b self, index: &uint) -> &'b $sty {
                    let real_index = *index - self.offset + MEMORY;
                    &self.slice[real_index]
                }
            }
            impl Iterator<u64> for Recurrence {
/* ... */
```

```
                #[inline]
                fn next(&mut self) -> Option<u64> {
                    if self.pos < MEMORY {
                        let next_val = self.mem[self.pos];
                        self.pos += 1;
                        Some(next_val)
                    } else {
                        let next_val = {
                            let n = self.pos;
                            let a = IndexOffset { slice: &self.mem, offset: n
};
                            $recur
//                          ^~~~~ changed
                        };
                        {
                            use std::mem::swap;
                            let mut swap_tmp = next_val;
                            for i in range(0, MEMORY).rev() {
                                swap(&mut swap_tmp, &mut self.mem[i]);
                            }
                        }
                        self.pos += 1;
                        Some(next_val)
                    }
                }
/* ... */
#             }
#             Recurrence { mem: [$($inits),+], pos: 0 }
#         }
#     };
# }
# fn main() {
#     let fib = recurrence![a[n]: u64 = 1, 1 ... a[n-1] + a[n-2]];
#     for e in fib.take(10) { println!("{}", e) }
# }
```

现在试图编译的话…

```
recurrence.rs:77:48: 77:49 error: unresolved name `a`
recurrence.rs:77     let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-
2]];
                                                                ^
```

```
recurrence.rs:7:1: 74:2 note: in expansion of recurrence!
recurrence.rs:77:15: 77:64 note: expansion site
recurrence.rs:77:50: 77:51 error: unresolved name `n`
recurrence.rs:77     let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-
2]];
                                                  ^
recurrence.rs:7:1: 74:2 note: in expansion of recurrence!
recurrence.rs:77:15: 77:64 note: expansion site
recurrence.rs:77:57: 77:58 error: unresolved name `a`
recurrence.rs:77     let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-
2]];
                                                        ^
recurrence.rs:7:1: 74:2 note: in expansion of recurrence!
recurrence.rs:77:15: 77:64 note: expansion site
recurrence.rs:77:59: 77:60 error: unresolved name `n`
recurrence.rs:77     let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-
2]];
                                                          ^
recurrence.rs:7:1: 74:2 note: in expansion of recurrence!
recurrence.rs:77:15: 77:64 note: expansion site
```

…等等，什么情况？这没道理…让我们看看宏究竟展开成了什么样子。

```
$ rustc -Z unstable-options --pretty expanded recurrence.rs
```

参数 `--pretty expanded` 将促使 `rustc` 展开宏，并将输出的AST再重转为源代码。此选项当前被认定为是 `unstable` ，因此我们还要添加 `-Z unstable-options` 。输出的信息(经过整理格式后)如下；特别留意 `$recur` 被替换掉的位置：

```
#![feature(no_std)]
#![no_std]
#[prelude_import]
use std::prelude::v1::*;
#[macro_use]
extern crate std as std;
fn main() {
    let fib = {
        use std::ops::Index;
        const MEM_SIZE: usize = 1 + 1;
        struct Recurrence {
            mem: [u64; MEM_SIZE],
            pos: usize,
```

```
        }
        struct IndexOffset<'a> {
            slice: &'a [u64; MEM_SIZE],
            offset: usize,
        }
        impl <'a> Index<usize> for IndexOffset<'a> {
            type Output = u64;
            #[inline(always)]
            fn index<'b>(&'b self, index: usize) -> &'b u64 {
                use std::num::Wrapping;
                let index = Wrapping(index);
                let offset = Wrapping(self.offset);
                let window = Wrapping(MEM_SIZE);
                let real_index = index - offset + window;
                &self.slice[real_index.0]
            }
        }
        impl Iterator for Recurrence {
            type Item = u64;
            #[inline]
            fn next(&mut self) -> Option<u64> {
                if self.pos < MEM_SIZE {
                    let next_val = self.mem[self.pos];
                    self.pos += 1;
                    Some(next_val)
                } else {
                    let next_val = {
                        let n = self.pos;
                        let a = IndexOffset{slice: &self.mem, offset: n,};
                        a[n - 1] + a[n - 2]
                    };
                    {
                        use std::mem::swap;
                        let mut swap_tmp = next_val;
                        {
                            let result =
                                match
::std::iter::IntoIterator::into_iter((0..MEM_SIZE).rev()) {
                                    mut iter => loop {
                                        match ::std::iter::Iterator::next(&mut
iter) {
                                            ::std::option::Option::Some(i) => {
```

```
                                                swap(&mut swap_tmp, &mut
54. self.mem[i]);
55.                                             }
                                                ::std::option::Option::None =>
56. break,
57.                                         }
58.                                     },
59.                                 };
60.                             result
61.                         }
62.                     }
63.                     self.pos += 1;
64.                     Some(next_val)
65.                 }
66.             }
67.         }
68.         Recurrence{mem: [0, 1], pos: 0,}
69.     };
70.     {
71.         let result =
72.             match ::std::iter::IntoIterator::into_iter(fib.take(10)) {
73.                 mut iter => loop {
74.                     match ::std::iter::Iterator::next(&mut iter) {
75.                         ::std::option::Option::Some(e) => {
76.                             ::std::io::_print(::std::fmt::Arguments::new_v1(
77.                                 {
                                        static __STATIC_FMTSTR: &'static [&'static
78. str] = &["", "\n"];
79.                                     __STATIC_FMTSTR
80.                                 },
81.                                 &match (&e,) {
                                        (__arg0,) =>
82. [::std::fmt::ArgumentV1::new(__arg0, ::std::fmt::Display::fmt)],
83.                                 }
84.                             ))
85.                         }
86.                         ::std::option::Option::None => break,
87.                     }
88.                 },
89.             };
90.         result
91.     }
92. }
```

呃..这看起来完全合法！如果我们加上几条 `#![feature(...)]` 属性，并把它送去给一个nightly版本的 `rustc` ，甚至真能通过编译…究竟什么情况？！

> 附注：上述代码无法通过非nightly版 `rustc` 编译。这是因为， `println!` 宏的展开结果依赖于编译器内部的细节，这些细节尚未被公开稳定化。

保持卫生

这儿的问题在于，Rust宏中的标识符具有卫生性。这就是说，出自不同上下文的标识符不可能发生冲突。作为演示，举个简单的例子。

```
# /*
macro_rules! using_a {
    ($e:expr) => {
        {
            let a = 42i;
            $e
        }
    }
}

let four = using_a!(a / 10);
# */
# fn main() {}
```

此宏接受一个表达式，然后把它包进一个定义了变量 `a` 的区块里。我们随后用它绕个弯子来求 `4` 。这个例子中实际上存在2种句法上下文，但我们看不见它们。为了帮助说明，我们给每个上下文都上一种不同的颜色。我们从未展开的代码开始上色，此时仅看得见一种上下文：

```
macro_rules! using_a {
    ($e:expr) => {
        {
            let a = 42;
            $e
        }
    }
}

let four = using_a!(a / 10);
```

现在，展开宏调用。

```
let four = {
    let a = 42;
    a / 10
};
```

可以看到，在宏中定义的 `a` 与调用所提供的 `a` 处于不同的上下文中。因此，虽然它们的字母表示一致，编译器仍将它们视作完全不同的标识符。

宏的这一特性需要格外留意：它们可能会产出无法通过编译的AST；但同样的代码，手写或通过 `--pretty expanded` 转印出来则能够通过编译。

解决方案是，采用合适的句法上下文来捕获标识符。我们沿用上例，并作修改：

```
macro_rules! using_a {
    ($a:ident, $e:expr) => {
        {
            let $a = 42;
            $e
        }
    }
}

let four = using_a!(a, a / 10);
```

现在它将展开为：

```
let four = {
    let a = 42;
    a / 10
};
```

上下文现在匹配了，编译通过。我们的 `recurrence!` 宏也可被如此调整：显式地捕获 `a` 与 `n` 即可。调整后我们得到：

```
macro_rules! count_exprs {
    () => (0);
    ($head:expr) => (1);
    ($head:expr, $($tail:expr),*) => (1 + count_exprs!($($tail),*));
}

macro_rules! recurrence {
```

```
        ( $seq:ident [ $ind:ident ]: $sty:ty = $($inits:expr),+ ... $recur:expr )
 8. => {
 9. //    ^~~~~~~~~~   ^~~~~~~~~~ changed
10.         {
11.             use std::ops::Index;
12.
13.             const MEM_SIZE: usize = count_exprs!($($inits),+);
14.
15.             struct Recurrence {
16.                 mem: [$sty; MEM_SIZE],
17.                 pos: usize,
18.             }
19.
20.             struct IndexOffset<'a> {
21.                 slice: &'a [$sty; MEM_SIZE],
22.                 offset: usize,
23.             }
24.
25.             impl<'a> Index<usize> for IndexOffset<'a> {
26.                 type Output = $sty;
27.
28.                 #[inline(always)]
29.                 fn index<'b>(&'b self, index: usize) -> &'b $sty {
30.                     use std::num::Wrapping;
31.
32.                     let index = Wrapping(index);
33.                     let offset = Wrapping(self.offset);
34.                     let window = Wrapping(MEM_SIZE);
35.
36.                     let real_index = index - offset + window;
37.                     &self.slice[real_index.0]
38.                 }
39.             }
40.
41.             impl Iterator for Recurrence {
42.                 type Item = $sty;
43.
44.                 #[inline]
45.                 fn next(&mut self) -> Option<$sty> {
46.                     if self.pos < MEM_SIZE {
47.                         let next_val = self.mem[self.pos];
48.                         self.pos += 1;
```

```
                        Some(next_val)
                    } else {
                        let next_val = {
                            let $ind = self.pos;
//                              ^~~~ changed
                            let $seq = IndexOffset { slice: &self.mem, offset:
$ind };
//                              ^~~~ changed
                            $recur
                        };

                        {
                            use std::mem::swap;

                            let mut swap_tmp = next_val;
                            for i in (0..MEM_SIZE).rev() {
                                swap(&mut swap_tmp, &mut self.mem[i]);
                            }
                        }

                        self.pos += 1;
                        Some(next_val)
                    }
                }
            }

            Recurrence { mem: [$($inits),+], pos: 0 }
        }
    };
}

fn main() {
    let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-2]];

    for e in fib.take(10) { println!("{}", e) }
}
```

通过编译了！接下来，我们试试别的数列。

```
# macro_rules! count_exprs {
#     () => (0);
#     ($head:expr) => (1);
```

```
#     ($head:expr, $($tail:expr),*) => (1 + count_exprs!($($tail),*));
# }
#
# macro_rules! recurrence {
#     ( $seq:ident [ $ind:ident ]: $sty:ty = $($inits:expr),+ ... $recur:expr )
=> {
#         {
#             use std::ops::Index;
#
#             const MEM_SIZE: usize = count_exprs!($($inits),+);
#
#             struct Recurrence {
#                 mem: [$sty; MEM_SIZE],
#                 pos: usize,
#             }
#
#             struct IndexOffset<'a> {
#                 slice: &'a [$sty; MEM_SIZE],
#                 offset: usize,
#             }
#
#             impl<'a> Index<usize> for IndexOffset<'a> {
#                 type Output = $sty;
#
#                 #[inline(always)]
#                 fn index<'b>(&'b self, index: usize) -> &'b $sty {
#                     use std::num::Wrapping;
#
#                     let index = Wrapping(index);
#                     let offset = Wrapping(self.offset);
#                     let window = Wrapping(MEM_SIZE);
#
#                     let real_index = index - offset + window;
#                     &self.slice[real_index.0]
#                 }
#             }
#
#             impl Iterator for Recurrence {
#                 type Item = $sty;
#
#                 #[inline]
#                 fn next(&mut self) -> Option<$sty> {
```

```
#                     if self.pos < MEM_SIZE {
#                         let next_val = self.mem[self.pos];
#                         self.pos += 1;
#                         Some(next_val)
#                     } else {
#                         let next_val = {
#                             let $ind = self.pos;
#                             let $seq = IndexOffset { slice: &self.mem,
offset: $ind };
#                             $recur
#                         };
#
#                         {
#                             use std::mem::swap;
#
#                             let mut swap_tmp = next_val;
#                             for i in (0..MEM_SIZE).rev() {
#                                 swap(&mut swap_tmp, &mut self.mem[i]);
#                             }
#                         }
#
#                         self.pos += 1;
#                         Some(next_val)
#                     }
#                 }
#             }
#
#             Recurrence { mem: [$($inits),+], pos: 0 }
#         }
#     };
# }
#
# fn main() {
for e in recurrence!(f[i]: f64 = 1.0 ... f[i-1] * i as f64).take(10) {
    println!("{}", e)
}
# }
```

运行上述代码得到：

```
1
1
```

```
2
6
24
120
720
5040
40320
362880
```

成功了！

常用模式

解析与扩展中的常用套路。

回调

```
macro_rules! call_with_larch {
    ($callback:ident) => { $callback!(larch) };
}

macro_rules! expand_to_larch {
    () => { larch };
}

macro_rules! recognise_tree {
    (larch) => { println!("#1, 落叶松。") };
    (redwood) => { println!("#2, THE巨红杉。") };
    (fir) => { println!("#3, 冷杉。") };
    (chestnut) => { println!("#4, 七叶树。") };
    (pine) => { println!("#5, 欧洲赤松。") };
    ($($other:tt)*) => { println!("不懂，可能是种桦树？") };
}

fn main() {
    recognise_tree!(expand_to_larch!());
    call_with_larch!(recognise_tree);
}
```

由于宏展开的机制限制，(在Rust1.2中)不可能做到把一例宏的展开结果作为有效信息提供给另一例宏。这为宏的模组化工作施加了难度。

使用递归并传递回调是条出路。作为演示，上例两处宏调用的展开过程如下：

```
recognise_tree! { expand_to_larch ! (  ) }
println! { "I don't know; some kind of birch maybe?" }
// ...

call_with_larch! { recognise_tree }
recognise_tree! { larch }
println! { "#1, the Larch." }
// ...
```

可以使用 `tt` 的重复来将任意参数转发给回调：

```
macro_rules! callback {
    ($callback:ident($($args:tt)*)) => {
        $callback!($($args)*)
    };
}

fn main() {
    callback!(callback(println("Yes, this *was* unnecessary.")));
}
```

如有需要，当然还可以在参数中增加额外的标记。

标记树撕咬机

```
macro_rules! mixed_rules {
    () => {};
    (trace $name:ident; $($tail:tt)*) => {
        {
            println!(concat!(stringify!($name), " = {:?}"), $name);
            mixed_rules!($($tail)*);
        }
    };
    (trace $name:ident = $init:expr; $($tail:tt)*) => {
        {
            let $name = $init;
            println!(concat!(stringify!($name), " = {:?}"), $name);
            mixed_rules!($($tail)*);
        }
    };
}
#
# fn main() {
#     let a = 42;
#     let b = "Ho-dee-oh-di-oh-di-oh!";
#     let c = (false, 2, 'c');
#     mixed_rules!(
#         trace a;
#         trace b;
#         trace c;
#         trace b = "They took her where they put the crazies.";
#         trace b;
#     );
# }
```

此模式可能是最强大的宏解析技巧。通过使用它，一些极其复杂的语法都能得到解析。

“标记树撕咬机”是一种递归宏，其工作机制有赖于对输入的顺次、逐步处理。处理过程的每一步中，它都将匹配并移除(“撕咬”掉)输入头部的一列标记，得到一些中间结果，然后再递归地处理输入剩下的尾部。

名称中含有“标记树”，是因为输入中尚未被处理的部分总是被捕获在 `$($tail:tt)*` 的形式中。之所以如此，是因为只有通过使用 `tt` 的重复才能做到无损地捕获住提供给宏的部分输入。

标记树撕咬机仅有的限制，也是整个宏系统的局限：

- 你只能匹配 `macro_rules!` 允许匹配的字面值和语法结构。
- 你无法匹配不成对的标记组(unbalanced group)。

然而，需要把宏递归的局限性纳入考量。 `macro_rules!` 没有做任何形式的尾递归消除或优化。在写标记树撕咬机时，推荐多花些功夫，尽可能地限制递归调用的次数。对于输入的变化，增加额外的匹配分支(而非采用中间层并使用递归)；或对输入句法施加限制，以便于对标准重复的记录追踪。

%内用规则

```
#[macro_export]
macro_rules! foo {
    (@as_expr $e:expr) => {$e};

    ($($tts:tt)*) => {
        foo!(@as_expr $($tts)*)
    };
}
#
# fn main() {
#     assert_eq!(foo!(42), 42);
# }
```

宏并不参与标准的条目可见性与查找流程，因此，如果一个公共可见宏在其内部调用了其它宏，那么被调用的宏也将必须公共可见。这会污染全局命名空间，甚至会与来自其它 `crate` 的宏发生冲突。那些想对宏进行选择性导入的用户也会因之感到困惑；他们必须导入所有宏—包括公开文档并未记录的—才能使代码正常运转。

将这些本不该公共可见的宏封装进需要被导出的宏内部，是一个不错的解决方案。上例展示了如何将常用的 `as_expr!` 宏移至另一个宏的内部，仅有后者公共可见。

之所以用 `@` ，是因为在Rust 1.2下，该标记尚无任何在前缀位置的用法；因此，我们的语法定义不会与任何东西撞车。想用的话，别的符号或特有前缀都可以；但 `@` 的用例已被传播开来，因此，使用它可能更容易帮助读者理解你的代码。

> 注意：标记 `@` 先前曾作为前缀被用于表示被垃圾回收了的指针，那时的语言还在采用各种记号代表指针类型。现在的标记 `@` 只有一种用法：将名称绑定至模式中。而在此用法中它是中缀运算符，与我们的上述用例并不冲突。

还有一点，内用规则通常应排在“真正的”规则之前。这样做可避免 `macro_rules!` 错把内规调用解析成别的东西，比如表达式。

如果导出内用规则无法避免(比如说，有一干效用性的宏规则，很多应被导出的宏都同时需要用到它们)，你仍可以采用此规则，将所有内用规则封装到一个“究极”效用宏里去：

```
macro_rules! crate_name_util {
    (@as_expr $e:expr) => {$e};
    (@as_item $i:item) => {$i};
    (@count_tts) => {0usize};
    // ...
}
```

下推累积

```
macro_rules! init_array {
    (@accum (0, $_e:expr) -> ($($body:tt)*))
        => {init_array!(@as_expr [$($body)*])};
    (@accum (1, $e:expr) -> ($($body:tt)*))
        => {init_array!(@accum (0, $e) -> ($($body)* $e,))};
    (@accum (2, $e:expr) -> ($($body:tt)*))
        => {init_array!(@accum (1, $e) -> ($($body)* $e,))};
    (@accum (3, $e:expr) -> ($($body:tt)*))
        => {init_array!(@accum (2, $e) -> ($($body)* $e,))};
    (@as_expr $e:expr) => {$e};
    [$e:expr; $n:tt] => {
        {
            let e = $e;
            init_array!(@accum ($n, e.clone()) -> ())
        }
    };
}

let strings: [String; 3] = init_array![String::from("hi!"); 3];
# assert_eq!(format!("{:?}", strings), "[\"hi!\", \"hi!\", \"hi!\"]");
```

在Rust中，所有宏最终必须展开为一个完整、有效的句法元素(比如表达式、条目等等)。这意味着，不可能定义一个最终展开为残缺构造的宏。

有些人可能希望，上例中的宏能被更加直截了当地表述成：

```
macro_rules! init_array {
    (@accum 0, $_e:expr) => {/* empty */};
    (@accum 1, $e:expr) => {$e};
    (@accum 2, $e:expr) => {$e, init_array!(@accum 1, $e)};
    (@accum 3, $e:expr) => {$e, init_array!(@accum 2, $e)};
    [$e:expr; $n:tt] => {
        {
            let e = $e;
            [init_array!(@accum $n, e)]
        }
    };
}
```

他们预期的展开过程如下：

```
1.              [init_array!(@accum 3, e)]
2.              [e, init_array!(@accum 2, e)]
3.              [e, e, init_array!(@accum 1, e)]
4.              [e, e, e]
```

然而，这一思路中，每个中间步骤的展开结果都是一个不完整的表达式。即便这些中间结果对外部来说绝不可见，Rust仍然禁止这种用法。

下推累积则使我们得以在完全完成之前毋需考虑构造的完整性，进而累积构建出我们所需的标记序列。顶端给出的示例中，宏调用的展开过程如下：

```
1.  init_array! { String:: from ( "hi!" ) ; 3 }
2.  init_array! { @ accum ( 3 , e . clone (  ) ) -> (  ) }
3.  init_array! { @ accum ( 2 , e.clone() ) -> ( e.clone() , ) }
4.  init_array! { @ accum ( 1 , e.clone() ) -> ( e.clone() , e.clone() , ) }
    init_array! { @ accum ( 0 , e.clone() ) -> ( e.clone() , e.clone() , e.clone()
5.  , ) }
6.  init_array! { @ as_expr [ e.clone() , e.clone() , e.clone() , ] }
```

可以看到，每一步都在累积输出，直到规则完成，给出完整的表达式。

上述过程的关键点在于，使用 `$($body:tt)*` 来保存输出中间值，而不触发其它解析机制。采用 `($input) -> ($output)` 的形式仅是出于传统，用以明示此类宏的作用。

由于可以存储任意复杂的中间结果，下推累积在构建标记树撕咬机的过程中经常被用到。

重复替代

```
macro_rules! replace_expr {
    ($_t:tt $sub:expr) => {$sub};
}
```

在此模式中，匹配到的重复序列将被直接丢弃，仅留用它所带来的长度信息；原本标记所在的位置将被替换成某种重复要素。

举个例子，考虑如何为一个元素多于12个(Rust 1.2下的最大值)的 `tuple` 提供默认值。

```
macro_rules! tuple_default {
    ($($tup_tys:ty),*) => {
        (
            $(
                replace_expr!(
                    ($tup_tys)
                    Default::default()
                ),
            )*
        )
    };
}
#
# macro_rules! replace_expr {
#     ($_t:tt $sub:expr) => {$sub};
# }
#
# assert_eq!(tuple_default!(i32, bool, String), (0, false, String::new()));
```

> 仅对此例：我们其实可以直接用 `$tup_tys::default()` 。

上例中，我们并未真正使用匹配到的类型。实际上，我们把它抛开了，并用用一个表达式重复替代。换句话说，我们实际关心的不是有哪些类型，而是有多少个类型。

尾部分隔符

```
macro_rules! match_exprs {
    ($($exprs:expr),* $(,)*) => {...};
}
```

Rust语法在很多地方允许尾部分隔符存在。一列(举例说)表达式的常见匹配方式有两种(`$($exprs:expr),*`和`$($exprs:expr,)*`);一种可处理无尾部分隔符的情况，一种可处理有的情况；但没办法同时匹配到。

不过，在主重复的尾部放置一个`$(,)*`重复，则可以匹配到任意数量(包括0或1)的尾部分隔符。

注意此模式并非对所有情况都适用。如果被编译器拒绝，可以尝试增加匹配臂和/或使用逐条匹配。

标记树聚束

```
macro_rules! call_a_or_b_on_tail {
    ((a: $a:expr, b: $b:expr), 调a $($tail:tt)*) => {
        $a(stringify!($($tail)*))
    };

    ((a: $a:expr, b: $b:expr), 调b $($tail:tt)*) => {
        $b(stringify!($($tail)*))
    };

    ($ab:tt, $_skip:tt $($tail:tt)*) => {
        call_a_or_b_on_tail!($ab, $($tail)*)
    };
}

fn compute_len(s: &str) -> Option<usize> {
    Some(s.len())
}

fn show_tail(s: &str) -> Option<usize> {
    println!("tail: {:?}", s);
    None
}

fn main() {
    assert_eq!(
        call_a_or_b_on_tail!(
            (a: compute_len, b: show_tail),
            规则的 递归部分 将 跳过 所有这些 标记
            它们 并不 关心 我们究竟 调b 还是 调a
            只有 终结规则 关心
        ),
        None
    );
    assert_eq!(
        call_a_or_b_on_tail!(
        (a: compute_len, b: show_tail),
        而现在 为了 显式 可能的路径 有两条
        我们也 调a 一哈: 它的 输入 应该
```

```
39.         自我引用 因此 我们给它 一个 72),
40.         Some(72)
41.     );
42. }
```

在十分复杂的递归宏中，可能需要非常多的参数，才足以在每层调用之间传递必要的标识符与表达式。然而，根据实现上的差异，可能存在许多这样的中间层，它们转发了这些参数，但并没有用到。

因此，将所有这些参数聚成一束，通过分组将其放进单独一棵标记树里；可以省事许多。这样一来，那些用不到这些参数的递归层可以直接捕获并替换这棵标记树，而不需要把整组参数完完全全准准确确地捕获替换掉。

上面的例子把表达式 `$a` 和 `$b` 聚束，然后作为一棵 `tt` 交由递归规则转发。随后，终结规则将这组标记打开，并访问其中的表达式。

可见性

在Rust中，因为没有类似 `vis` 的匹配选项，匹配替换可见性标记比较难搞。

匹配与忽略

根据上下文，可由重复做到这点：

```
macro_rules! struct_name {
    ($(pub)* struct $name:ident $($rest:tt)*) => { stringify!($name) };
}
#
# fn main() {
#     assert_eq!(struct_name!(pub struct Jim;), "Jim");
# }
```

上例将匹配公共可见或本地可见的 `struct` 条目。但它还能匹配到 `pub pub` （十分公开?)甚至是 `pub pub pub pub` （真的非常非常公开)。防止这种情况出现的最好方法，只有祈祷调用方没那么多毛病。

匹配和替换

由于不能将重复的内容和其自身同时绑定至一个变量，没有办法将 `$(pub)*` 的内容直接拿去替换使用。因此，我们只好使用多条规则：

```
macro_rules! newtype_new {
    (struct $name:ident($t:ty);) => { newtype_new! { () struct $name($t); } };
    (pub struct $name:ident($t:ty);) => { newtype_new! { (pub) struct
$name($t); } };

    (($($vis:tt)*) struct $name:ident($t:ty);) => {
        as_item! {
            impl $name {
                $($vis)* fn new(value: $t) -> Self {
                    $name(value)
                }
            }
        }
    };
```

```
}

macro_rules! as_item { ($i:item) => {$i} }
#
# #[derive(Debug, Eq, PartialEq)]
# struct Dummy(i32);
#
# newtype_new! { struct Dummy(i32); }
#
# fn main() {
#     assert_eq!(Dummy::new(42), Dummy(42));
# }
```

> 参考: AST强转.

这里，我们用到了宏对成组的任意标记的匹配能力，来同时匹配 `()` 与 `(pub)` ，并将所得内容替换到输出中。因为在此处解析器不会期望看到一个 `tt` 重复的展开结果，我们需要使用AST强转来使代码正常运作。

临时措施

本节将留给那些价值有待探讨，以及那些可能有缺陷因而不适合出现在正文中的模式与技巧。

算盘计数

> 临时信息：需要更合适的例子。虽然它是 `Ook!` 宏的重要组成部分之一，但该用例——匹配Rust分组机制无法表示的嵌套结构——实在是过于特殊，因此不适作为例子使用。
>
> 注意：此节预设读者了解下推累积以及标记树撕咬机。

```
macro_rules! abacus {
    ((- $($moves:tt)*) -> (+ $($count:tt)*)) => {
        abacus!(($($moves)*) -> ($($count)*))
    };
    ((- $($moves:tt)*) -> ($($count:tt)*)) => {
        abacus!(($($moves)*) -> (- $($count)*))
    };
    ((+ $($moves:tt)*) -> (- $($count:tt)*)) => {
        abacus!(($($moves)*) -> ($($count)*))
    };
    ((+ $($moves:tt)*) -> ($($count:tt)*)) => {
        abacus!(($($moves)*) -> (+ $($count)*))
    };

    // 检查最终结果是否为零
    (() -> ()) => { true };
    (() -> ($($count:tt)+)) => { false };
}

fn main() {
    let equals_zero = abacus!((++-+-+++--++---++----+) -> ());
    assert_eq!(equals_zero, true);
}
```

当需要记录的计数会发生变化，且初始值为零或在零附近，且必须支持如下操作：

- 增加一；
- 减少一；
- 与0(或任何其它固定有限值)相比较；

时，可以使用次技巧。

数值n将由一组共n个相同的特定标记来表示。对数值的修改操作将采用下推累积模式由递归调用完成。假设所采用的特定标记是 `x` ，则上述操作可实现为：

- 增加一：匹配 `($($count:tt)*)` 并替换为 `(x $($count)*)` 。
- 减少一：匹配 `(x $($count:tt)*)` 并替换为 `($($count)*)` 。
- 与0相比较：匹配 `()` 。
- 与1相比较：匹配 `(x)` 。
- 与2相比较：匹配 `(x x)` 。
- *(依此类推…)*

作用于计数值的操作将所选的标记来回摆动，如同算盘摆动算子。[^abacus]

[^abacus]: 在这句极度单薄的辩解下，隐藏着选用此名称的真实理由：避免造出又一个名含“标记”的术语。今天就该跟你认识的作者谈谈规避语义饱和吧！公平来讲，本来也可以称它为“一元计数 (unary counting)”。

在想表示负数的情况下，值-*n*可被表示成*n*个相同的其它标记。在上例中，值+*n*被表示成*n*个 `+` 标记，而值-*m*被表示成*m*个 `-` 标记。

有负数的情况下操作起来稍微复杂一些，增减操作在当前数值为负时实际上互换了角色。给定 `+` 和 `-` 分别作为正数与负数标记，相应操作的实现将变成：

- 增加一：
  - 匹配 `()` 并替换为 `(+)`
  - 匹配 `(- $($count:tt)*)` 并替换为 `($($count)*)`
  - 匹配 `($($count:tt)+)` 并替换为 `(+ $($count)+)`
- 减少一：
  - 匹配 `()` 并替换为 `(-)`
  - 匹配 `(+ $($count:tt)*)` 并替换为 `($($count)*)`
  - 匹配 `($($count:tt)+)` 并替换为 `(- $($count)+)`
- 与0相比较：匹配 `()`
- 与+1相比较：匹配 `(+)`
- 与-1相比较：匹配 `(-)`
- 与+2相比较：匹配 `(++)`
- 与-2相比较：匹配 `(--)`
- *(依此类推…)*

注意在顶部的示例中，某些规则被合并到一起了(举例来说，对 `()` 及 `($($count:tt)+)` 的增加操作被合并为对 `($($count:tt)*)` 的增加操作)。

如果想要提取出所计数目的实际值，可再使用普通的计数宏。对上例来说，终结规则可换为：

```
macro_rules! abacus {
    // ...

    // 下列规则将计数替换成实际值的表达式
    (() -> ()) => {0};
    (() -> (- $($count:tt)*)) => {
        {(-1i32) $(- replace_expr!($count 1i32))*}
    };
    (() -> (+ $($count:tt)*)) => {
        {(1i32) $(+ replace_expr!($count 1i32))*}
    };
}

macro_rules! replace_expr {
    ($_t:tt $sub:expr) => {$sub};
}
```

仅限此例：严格来说，想要达到此例的效果，没必要做的这么复杂。如果你不需要在宏中匹配所计的值，可直接采用重复来更加高效地实现：

```
macro_rules! abacus {
    (-) => {-1};
    (+) => {1};
    ($($moves:tt)*) => {
        0 $(+ abacus!($moves))*
    }
}
```

轮子

可复用的宏代码片段。

AST强转

在替换 `tt` 时，Rust的解析器并不十分可靠。当它期望得到某类特定的语法构造时，如果摆在它面前的是一坨替换后的 `tt` 标记，就有可能出现问题。解析器常常直接选择死亡，而非尝试去解析它们。在这类情况中，就要用到AST强转。

```
# #![allow(dead_code)]
#
macro_rules! as_expr { ($e:expr) => {$e} }
macro_rules! as_item { ($i:item) => {$i} }
macro_rules! as_pat  { ($p:pat) =>  {$p} }
macro_rules! as_stmt { ($s:stmt) => {$s} }
#
# as_item!{struct Dummy;}
#
# fn main() {
#     as_stmt!(let as_pat!(_) = as_expr!(42));
# }
```

这些强制变换经常与下推累积宏一同使用，以使解析器能够将最终输出的 `tt` 序列当作某类特定的语法构造对待。

注意，之所以只有这几种强转宏，是因为决定可用强转类型的点在于宏可展开在哪些地方，而不是宏能够捕捉哪些东西。也就是说，因为宏没办法在· `type` 处展开issue-27245，所以就没办法实现什么 `as_ty!` 强转。

计数

重复替代

在宏中计数是一项让人吃惊地难搞的活儿。最简单的方式是采用重复替代。

```
macro_rules! replace_expr {
    ($_t:tt $sub:expr) => {$sub};
}

macro_rules! count_tts {
    ($($tts:tt)*) => {0usize $(+ replace_expr!($tts 1usize))*};
}
#
# fn main() {
#     assert_eq!(count_tts!(0 1 2), 3);
# }
```

对于小数目来说，这方法不错，但当输入量到达500标记附近时，编译器将被击溃。想想吧，输出的结果将类似：

```
0usize + 1usize + /* ~500 `+ 1usize`s */ + 1usize
```

编译器必须把这一大串解析成一棵AST，那可会是一棵完美失衡的500多级深的二叉树。

递归

递归是个老套路。

```
macro_rules! count_tts {
    () => {0usize};
    ($_head:tt $($tail:tt)*) => {1usize + count_tts!($($tail)*)};
}
#
# fn main() {
#     assert_eq!(count_tts!(0 1 2), 3);
# }
```

> 注意：对于 `rustc` 1.2来说，很不幸，编译器在处理大数量的类型未知的整型字面值时将会出现性能问题。我们此处显式采用 `usize` 类型就是为了避免这种不幸。
>
> 如果这样做并不合适(比如说，当类型必须可替换时)，可通过 `as` 来减轻问题。(比如， `0 as $ty` ， `1 as $ty` 等)。

这方法管用，但很快就会超出宏递归的次数限制。与重复替换不同的是，可通过增加匹配分支来增加可处理的输入面值。

```
macro_rules! count_tts {
    ($_a:tt $_b:tt $_c:tt $_d:tt $_e:tt
     $_f:tt $_g:tt $_h:tt $_i:tt $_j:tt
     $_k:tt $_l:tt $_m:tt $_n:tt $_o:tt
     $_p:tt $_q:tt $_r:tt $_s:tt $_t:tt
     $($tail:tt)*)
        => {20usize + count_tts!($($tail)*)};
    ($_a:tt $_b:tt $_c:tt $_d:tt $_e:tt
     $_f:tt $_g:tt $_h:tt $_i:tt $_j:tt
     $($tail:tt)*)
        => {10usize + count_tts!($($tail)*)};
    ($_a:tt $_b:tt $_c:tt $_d:tt $_e:tt
     $($tail:tt)*)
       => {5usize + count_tts!($($tail)*)};
    ($_a:tt
     $($tail:tt)*)
        => {1usize + count_tts!($($tail)*)};
    () => {0usize};
}

fn main() {
    assert_eq!(700, count_tts!(
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

```

```
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        // 重复替换手段差不多将在此处崩溃
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
    ));
}
```

我们列出的这种形式可以处理差不多1,200个标记。

切片长度

第三种方法，是帮助编译器构建一个深度较小的AST，以避免栈溢出。可以通过新建数列，并调用其 `len` 方法来做到。

```
macro_rules! replace_expr {
    ($_t:tt $sub:expr) => {$sub};
}

macro_rules! count_tts {
    ($($tts:tt)*) => {<[()]>::len(&[$(replace_expr!($tts ())),*])};
}
#
# fn main() {
#     assert_eq!(count_tts!(0 1 2), 3);
# }
```

经过测试，这种方法可处理高达10,000个标记数，可能还能多上不少。缺点是，就Rust 1.2来说，没法拿它生成常量表达式。即便结果可以被优化成一个简单的常数(在 `debug build` 里得到的编译结果仅是一次内存载入)，它仍然无法被用在常量位置(如 `const` 值或定长数组的长度值)。

不过，如果非常量计数够用的话，此方法很大程度上是上选。

枚举计数

当你需要计互不相同的标识符的数量时，可以用到此方法。结果还可被用作常量。

```
macro_rules! count_idents {
    ($($idents:ident),* $(,)*) => {
        {
            #[allow(dead_code, non_camel_case_types)]
            enum Idents { $($idents,)* __CountIdentsLast }
            const COUNT: u32 = Idents::__CountIdentsLast as u32;
            COUNT
        }
    };
}
#
# fn main() {
#     const COUNT: u32 = count_idents!(A, B, C);
#     assert_eq!(COUNT, 3);
# }
```

此方法的确有两大缺陷。其一，如上所述，它仅能被用于数有效的标识符(同时还不能是关键词)，而且它不允许那些标识符有重复。其二，此方法不具备卫生性；就是说如果你的末位标识符(在 `__CountIdentsLast` 位置的标识符)的字面值也是输入之一，宏调用就会失败，因为 `enum` 中包含重复变量。

枚举解析

```
macro_rules! parse_unitary_variants {
    (@as_expr $e:expr) => {$e};
    (@as_item $($i:item)+) => {$($i)+};

    // Exit rules.
    (
        @collect_unitary_variants ($callback:ident ( $($args:tt)* )),
        ($(,)*) -> ($($var_names:ident,)*)
    ) => {
        parse_unitary_variants! {
            @as_expr
            $callback!{ $($args)* ($($var_names),*) }
        }
    };

    (
        @collect_unitary_variants ($callback:ident { $($args:tt)* }),
        ($(,)*) -> ($($var_names:ident,)*)
    ) => {
        parse_unitary_variants! {
            @as_item
            $callback!{ $($args)* ($($var_names),*) }
        }
    };

    // Consume an attribute.
    (
        @collect_unitary_variants $fixed:tt,
        (#[$_attr:meta] $($tail:tt)*) -> ($($var_names:tt)*)
    ) => {
        parse_unitary_variants! {
            @collect_unitary_variants $fixed,
            ($($tail)*) -> ($($var_names)*)
        }
    };

    // Handle a variant, optionally with an with initialiser.
    (
```

```
        @collect_unitary_variants $fixed:tt,
        ($var:ident $(= $_val:expr)*, $($tail:tt)*) -> ($($var_names:tt)*)
    ) => {
        parse_unitary_variants! {
            @collect_unitary_variants $fixed,
            ($($tail)*) -> ($($var_names)* $var,)
        }
    };

    // Abort on variant with a payload.
    (
        @collect_unitary_variants $fixed:tt,
        ($var:ident $_struct:tt, $($tail:tt)*) -> ($($var_names:tt)*)
    ) => {
        const _error: () = "cannot parse unitary variants from enum with non-
unitary variants";
    };

    // Entry rule.
    (enum $name:ident {$($body:tt)*} => $callback:ident $arg:tt) => {
        parse_unitary_variants! {
            @collect_unitary_variants
            ($callback $arg), ($($body)*,) -> ()
        }
    };
}
#
# fn main() {
#     assert_eq!(
#         parse_unitary_variants!(
#             enum Dummy { A, B, C }
#             => stringify(variants:)
#         ),
#         "variants : ( A , B , C )"
#     );
# }
```

此宏展示了如何使用标记树撕咬机与下推累积来解析类C枚举的变量。完成后的 `parse_unitary_variants!` 宏将调用一个回调宏，为后者提供枚举中的所有选择(以及任何附加参数)。

经过改造后，此宏将也能用于解析 `struct` 的域，或为枚举变量计算标签值，甚至是将任意一个枚举

中的所有变量名称都提取出来。

实例注解

此章节包含一些实际代码环境^*中的宏，并附上说明其设计与构造的注解。

Ook!

此宏是对Ook!密文的实现，该语言与Brainfuck密文同构。

此语言的执行模式非常简单：内存被表示为一列总量不定(通常至少包括30,000个)的“单元”(每个单元至少8bit)；另有一个指向该内存的指针，最初指向位置0；还有一个执行栈(用来实现循环)和一个指向程序代码的指针。最后两个组件并未暴露给程序本身，它们属于程序的运行时性质。

语言本身仅由三种标记，`Ook.`、`Ook?`及`Ook!`构成。它们两两组合，构成了八种运算符：

- `Ook. Ook?` - 指针增。
- `Ook? Ook.` - 指针减。
- `Ook. Ook.` - 所指单元增。
- `Ook! Ook!` - 所指单元减。
- `Ook! Ook.` - 将所指单元写至标准输出。
- `Ook. Ook!` - 从标准输入读至所指单元。
- `Ook! Ook?` - 进入循环。
- `Ook? Ook!` - 如果所指单元非零，跳回循环起始位置；否则，继续执行。

Ook!之所以有趣，是因为它图灵完备。这意味着，你必须要在同样图灵完备的环境中才能实现它。

实现

```
1.  #![recursion_limit = "158"]
```

实际上，这个值将是我们随后给出的示例程序编译成功所需的最低值。如果你很好奇究竟是怎样的程序，会如此这般复杂，以至于必须要把递归极限调至默认值的近五倍大… 请大胆猜测。

```
1.  type CellType = u8;
2.  const MEM_SIZE: usize = 30_000;
```

加入这些定义，以供宏展开使用^*

```
1.  macro_rules! Ook {
```

可能应该取名`ook!`以符合Rust标准的命名传统。然而良机不可错失，我们就用本名吧！

我们使用了内用规则模式；此宏的规则因而可被分为几个模块。

第一块是`@start`规则，负责为之后的展开搭建舞台。没什么特别的地方：先定义一些变量、效用函

数，然后处理展开的大头。

一些脚注：

- 我们之所以选择展开为函数，很大原因在于，这样以来就可以采用 `try!` 来简化错误处理流程。
- 有些时候，举例来说，如果用户想写一个不做I/O的程序，那么I/O相关的名称就不会被用到。我们让部分名称以 `_` 起头，正是为了使编译器不对此类情况产生抱怨。

```
1.      (@start $($Ooks:tt)*) => {
2.          {
3.              fn ook() -> ::std::io::Result<Vec<CellType>> {
4.                  use ::std::io;
5.                  use ::std::io::prelude::*;
6.
7.                  fn _re() -> io::Error {
8.                      io::Error::new(
9.                          io::ErrorKind::Other,
10.                         String::from("ran out of input"))
11.                 }
12.
13.                 fn _inc(a: &mut [u8], i: usize) {
14.                     let c = &mut a[i];
15.                     *c = c.wrapping_add(1);
16.                 }
17.
18.                 fn _dec(a: &mut [u8], i: usize) {
19.                     let c = &mut a[i];
20.                     *c = c.wrapping_sub(1);
21.                 }
22.
23.                 let _r = &mut io::stdin();
24.                 let _w = &mut io::stdout();
25.
26.                 let mut _a: Vec<CellType> = Vec::with_capacity(MEM_SIZE);
27.                 _a.extend(::std::iter::repeat(0).take(MEM_SIZE));
28.                 let mut _i = 0;
29.                 {
30.                     let _a = &mut *_a;
31.                     Ook!(@e (_a, _i, _inc, _dec, _r, _w, _re); ($($Ooks)*));
32.                 }
33.                 Ok(_a)
34.             }
35.             ook()
```

```
        }
    };
```

解析运算符码

接下来，是一列“执行”规则，用于从输入中解析运算符码。

这列规则的通用形式是 `(@e $syms; ($input))` 。从 `@start` 规则种我们可以看出， `$syms` 包含了实现Ook程序所必须的符号：输入、输出、内存等等。我们使用了标记树聚束来简化转发这些符号的流程。

第一条规则是终止规则：一旦没有更多输入，我们就停下来。

```
    (@e $syms:tt; ()) => {};
```

其次，是一些适用于绝大部分运算符码的规则：我们剥下运算符码，换上其相应的Rust代码，然后继续递归处理输入剩下的部分。教科书式的标记树撕咬机。

```
    // 指针增
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook. Ook? $($tail:tt)*))
    => {
        $i = ($i + 1) % MEM_SIZE;
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };

    // 指针减
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook? Ook. $($tail:tt)*))
    => {
        $i = if $i == 0 { MEM_SIZE } else { $i } - 1;
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };

    // 所指增
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook. Ook. $($tail:tt)*))
    => {
        $inc($a, $i);
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };

```

```
    // 所指减
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook! Ook! $($tail:tt)*))
    => {
        $dec($a, $i);
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };

    // 输出
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook! Ook. $($tail:tt)*))
    => {
        try!($w.write_all(&$a[$i .. $i+1]));
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };

    // 读入
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook. Ook! $($tail:tt)*))
    => {
        try!(
            match $r.read(&mut $a[$i .. $i+1]) {
                Ok(0) => Err($re()),
                ok @ Ok(..) => ok,
                err @ Err(..) => err
            }
        );
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };
```

现在要弄复杂的了。运算符码 `Ook! Ook?` 标记着循环的开始。Ook!中的循环，翻译成Rust代码的话，类似：

> 注意：这不是宏定义的组成部分。
>
> ```
> while memory[ptr] != 0 {
>     // 循环内容
> }
> ```

显然，我们无法为中间步骤生成不完整的循环。这一问题似可用下推累积解决，但更根本的困难在于，我们无论如何都没办法生成类似 `while memory[ptr] != {` 的中间结果，完全不可能。因为它引入了不匹配的花括号。

解决方法是，把输入分成两部分：循环内部的，循环之后的。前者由 `@x` 规则处理，后者由 `@s` 处理。

```
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook! Ook? $($tail:tt)*))
    => {
        while $a[$i] != 0 {
            Ook!(@x ($a, $i, $inc, $dec, $r, $w, $re); (); (); ($($tail)*));
        }
        Ook!(@s ($a, $i, $inc, $dec, $r, $w, $re); (); ($($tail)*));
    };
```

提取循环区块

接下来是“提取”规则组 `@x` 。它们负责接受输入尾，并将之展开为循环的内容。这组规则的一般形式为： `(@x $sym; $depth; $buf; $tail)` 。

`$sym` 的用处与上相同。 `$tail` 表示需要被解析的输入；而 `$buf` 则作为下推累积的缓存，循环内的运算符码在经过解析后将被存入其中。那么， `$depth` 代表什么？

目前为止，我们还未提及如何处理嵌套循环。 `$depth` 的作用正在于此：我们需要记录当前循环在整个嵌套之中的深度，同时保证解析不会过早或过晚终止，而是刚好停在恰当的位置。^justright

由于在宏中没办法进行计算，而将数目匹配规则一一列出又不太可行(想想下面这一整套规则都得复制粘贴一堆不算小的整数的话，会是什么样子)，我们将只好回头采用最古老最珍贵的计数方法之一：亲手去数。

当然了，宏没有手，我们实际采用的是算盘计数模式。具体来说，我们选用标记 `@` ，每个 `@` 都表示新的一层嵌套。把这些 `@` 们放进一组后，我们就可以实现所需的操作了：

- 增加层数：匹配 `($($depth:tt)*)` 并用 `(@ $($depth)*)` 替换。
- 减少层数：匹配 `(@ $($depth:tt)*)` 并用 `($($depth)*)` 替换。
- 与0相比较：匹配 `()` 。

规则组中的第一条规则，用于在找到 `Ook? Ook!` 输入序列时，终止当前循环体的解析。随后，我们需要把累积所得的循环体内容发给先前定义的 `@e` 组规则。

注意，规则对于输入所剩的尾部不作任何处理(这项工作将由 `@s` 组的规则完成)。

```
    (@x $syms:tt; (); ($($buf:tt)*);
        (Ook? Ook! $($tail:tt)*))
    => {
        // 最外层的循环已被处理完毕，现在转而处理缓存到的标记。
```

```
        Ook!(@e $syms; ($($buf)*));
    };
```

紧接着，是负责进出嵌套的一些规则。它们修改深度计数，并将运算符码放入缓存。

```
    (@x $syms:tt; ($($depth:tt)*); ($($buf:tt)*);
        (Ook! Ook? $($tail:tt)*))
    => {
        // 嵌套变深
        Ook!(@x $syms; (@ $($depth)*); ($($buf)* Ook! Ook?); ($($tail)*));
    };

    (@x $syms:tt; (@ $($depth:tt)*); ($($buf:tt)*);
        (Ook? Ook! $($tail:tt)*))
    => {
        // 嵌套变浅
        Ook!(@x $syms; ($($depth)*); ($($buf)* Ook? Ook!); ($($tail)*));
    };
```

最后剩下的所有情况将交由一条规则处理。注意到它用的 `$op0` 和 `$op1` 两处捕获；对于Rust来说，Ook!中的一个标记将被视作两个标记：标识符 `Ook` 与剩下的符号。因此，我们用此规则来处理其它任何Ook!的非循环运算符，将 `!` ， `?` 和 `.` 作为 `tt` 匹配，并捕获之。

我们放置 `$depth` ，仅将运算符码推至缓存区中。

```
    (@x $syms:tt; $depth:tt; ($($buf:tt)*);
        (Ook $op0:tt Ook $op1:tt $($tail:tt)*))
    => {
        Ook!(@x $syms; $depth; ($($buf)* Ook $op0 Ook $op1); ($($tail)*));
    };
```

跳过循环区块

这组规则与循环提取大致相同，不过它们并不关心循环的内容(也因此不需要累积缓存)。它们仅仅关心循环何时被完全跳过。彼时，我们将恢复到 `@e` 组规则中并继续处理剩下的输入。

因此，我们将不加进一步说明地列出它们：

```
    // End of loop.
    (@s $syms:tt; ();
        (Ook? Ook! $($tail:tt)*))
    => {
```

```
        Ook!(@e $syms; ($($tail)*));
    };

    // Enter nested loop.
    (@s $syms:tt; ($($depth:tt)*);
        (Ook! Ook? $($tail:tt)*))
    => {
        Ook!(@s $syms; (@ $($depth)*); ($($tail)*));
    };

    // Exit nested loop.
    (@s $syms:tt; (@ $($depth:tt)*);
        (Ook? Ook! $($tail:tt)*))
    => {
        Ook!(@s $syms; ($($depth)*); ($($tail)*));
    };

    // Not a loop opcode.
    (@s $syms:tt; ($($depth:tt)*);
        (Ook $op0:tt Ook $op1:tt $($tail:tt)*))
    => {
        Ook!(@s $syms; ($($depth)*); ($($tail)*));
    };
```

入口

这是唯一一条非内用规则。

需注意的一点是，由于此规则单纯地匹配所有提供的标记，它极其危险。任何错误输入，都将造成其上的内用规则匹配完全失败，进而又落至匹配它(成功)的后果；引发无尽递归。

当在写、改及调试此类宏的过程中，明智的做法是，在此类规则的匹配头部加上临时性前缀，比如给此例加上一个 `@entry` ；以防止无尽递归，并得到更加恰当有效的错误信息。

```
    ($($Ooks:tt)*) => {
        Ook!(@start $($Ooks)*)
    };
}
```

用例

现在终于是时候上测试了。

```
fn main() {
    let _ = Ook!(
        Ook. Ook?  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook! Ook?  Ook? Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook?  Ook! Ook!  Ook? Ook!  Ook? Ook.
        Ook! Ook.  Ook. Ook?  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook! Ook?  Ook? Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook?
        Ook! Ook!  Ook? Ook!  Ook? Ook.  Ook. Ook.
        Ook! Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook! Ook.  Ook! Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook! Ook.  Ook. Ook?  Ook. Ook?
        Ook. Ook?  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook! Ook?  Ook? Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook?
        Ook! Ook!  Ook? Ook!  Ook? Ook.  Ook! Ook.
        Ook. Ook?  Ook. Ook?  Ook. Ook?  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook! Ook?  Ook? Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook?  Ook! Ook!  Ook? Ook!  Ook? Ook.
        Ook! Ook!  Ook! Ook!  Ook! Ook!  Ook! Ook.
        Ook? Ook.  Ook? Ook.  Ook? Ook.  Ook? Ook.
        Ook! Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook! Ook.  Ook! Ook!  Ook! Ook!  Ook! Ook!
        Ook! Ook!  Ook! Ook!  Ook! Ook!  Ook! Ook.
        Ook! Ook!  Ook! Ook!  Ook! Ook!  Ook! Ook!
        Ook! Ook!  Ook! Ook!  Ook! Ook!  Ook! Ook!
        Ook! Ook.  Ook. Ook?  Ook. Ook?  Ook. Ook.
        Ook! Ook.  Ook! Ook?  Ook! Ook!  Ook? Ook!
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook! Ook.
    );
```

```
43. }
```

运行(在编译器进行数百次递归宏展开而停顿相当长一段时间之后)的输出将是:

```
1. Hello World!
```

由此，我们揭示出了令人惊恐的真相: `macro_rules!` 是图灵完备的！

附注

此文所基的宏，是一个名为“Hodor!”的同构语言实现。Manish Goregaokar后来用那个宏实现了一个Brainfuck的解析器。也就是说，那个Brainfuck解析器用 `Hodor!` 宏写成，而后者本身则又是由 `macro_rules!` 实现的！

传说在把递归极限提至三百万，并让之编译了四天后，整个过程终于得以完成。

…收场的方式是栈溢出中止。时至今日，Rust宏驱动的密文编程语言仍然绝非可行的开发手段。