简介

中文翻译注（The Chinese translation of The Rust Edition Guide）：

- 👉 查看更多 **Rust 官方文档中英文双语教程**，包括双语版《Rust 程序设计语言》（出版书名为《Rust 权威指南》），本站还提供了 Rust 标准库中文版。
- 《Rust 版本指南》（The Rust Edition Guide 中文版）翻译自 The Rust Edition Guide，内容已全部翻译完成，查看此书的 Github 翻译项目和源码。本文版最后更新时间：2019-05-05。
- 本文档已加入到 Rust 中文翻译项目组，主要译者：*Sun*，Rust 中文翻译项目组成员。
- 本站支持文档中英文切换，点击页面右上角语言图标可切换到相同章节的英文页面，**英文版每天都会自动同步一次官方的最新版本。**
- 若发现当前页表达错误或帮助我们改进翻译，可点击右上角的编辑按钮打开该页对应源码文件进行编辑和修改，Rust 中文资源的开源组织发展离不开大家，感谢您的支持和帮助！
- 注意：**此文档已较长时间没更新，内容可能比英文滞后较多。**期待您加入 Rust 中文翻译项目组，协助我们，一起更新完善中文版，感激不尽！

欢迎来到 Rust 版本(Edition)使用指南！ "Editions" 是通过编写 Rust 代码来传达巨大改变的一种方式。

在指南中，我们将讨论：

- 什么是版本(editions)
- 每个版本什么样
- 如何将你的代码从一个版本迁移到另一个版本

请注意，标准库随每个Rust版本的增长而增长; 标准库中有**许多**添加的内容，本指南未对其进行说明。 只包含那些主要的变化，当然同时也有大量的中小型的改变也很棒。 您可能还想查看标准库文档。

什么是版本(Editions)?

Rust 六周发布一次新版本。这意味着用户可以获得不断的新功能。 这比其他语言的更新要快得多，但这也意味着每次更新都会更小。 一段时间之后，所有这些微小的变化都加进来了。 但是，从正式发布到正式发布，很难回头看看 *哇，在 Rust 1.10和 Rust 1.20之间，Rust已经发生了很大变化!*

每隔两三年，我们将制作一个新 *版本* 的 Rust。 每个版本都将功能集成到一个清晰的包中，并提供全面更新的文档和工具。 新版本通过正常的发布流程发布。

这为不同的人提供不同的目的：

- 对于活跃的 Rust 用户，它将增量更改集成到易于理解的包中。
- 对于非用户而言，它表明一些重大进步已经落地，这可能使 Rust 值得再看一眼。
- 对于那些开发 Rust 本身的人来说，它为整个项目提供了一个集结点。

兼容性

当编译器中出现新版本时，crates 必须明确选择使用它才能充分利用它。 此选择允许版本包含不兼容的更改，例如添加可能与代码中的标识符冲突的新关键字，或将警告转换为错误。 Rust 编译器将支持编译器发布之前存在的所有版本，并且可以将任何受支持版本的 crates 链接在一起。 版本更改仅影响编译器最初解析代码的方式。 因此，如果您正在使用 Rust 2015，并且您的某个依赖项使用 Rust 2018，那么一切正常。相反的情况也适用。

需要明确的是：大多数功能都适用于所有版本。随着新的稳定版本的发布，使用任何版本的 Rust 的人将继续看到改进。 但是，在某些情况下，主要是在添加新关键字时，但有时由于其他原因，可能会有新功能仅在以后的版本中提供。如果要利用此类功能，则只需升级。

试一下2018版本

在撰写本文时，有两个版本：2015和2018。2015是现在的 Rust版本; Rust 2018将于今年晚些时候发布。 要从2015版本过渡到2018版本，您需要开始使用迁移说明。

迁移你的代码到新版本

新版本可能会改变您编写 Rust 的方式 - 它们会添加新的语法，语言和库功能，但也会删除功能。 例如，`try`，`async`和`await`是 Rust 2018 中的关键字，但不在 Rust 2015中。 尽管如此，我们试图尽可能顺利地迁移到新版本。 如果很难将您的 crates 升级到新版本，那么这可能是一个 bug 。如果您遇到困难，那么应该向 Rust 提交一个 bug。

版本之间的迁移是围绕编译器标签(lints)构建的。从根本上说，这个过程是这样的：

- 打开 lints 以指示代码与新版本不兼容的位置
- 在没有警告的情况下编译代码。
- 选择加入新版本，代码应该编译。 *（可选）在新版本中启用有关 *idiomatic* 代码的 lints。

幸运的是，我们一直致力于 Cargo 帮助完成这一过程，最终推出了一个新的内置子命令`cargo fix`。 它可以从编译器中获取建议并自动重新编写代码以符合新功能和习惯用法，从而大大减少手动修复所需的警告数量！

> `cargo fix`仍然很早期，而且非常重要。但它已经适用于基础部分！我们正在努力使其变得更好，更强大，但暂时不必使用。

预览期

在发布版本之前，它将有一个“预览”阶段，让您可以在发布之前在 nightly 的 Rust 中试用新版本。 目前 Rust 2018 正处于预览阶段，因此，您需要采取额外的步骤来选择加入。 将此功能标志添加到您的`lib.rs`或`main.rs`以及任何示例中。如果你有一个项目的`examples`目录：

```
#![feature(rust_2018_preview)]
```

这将启用 特性状态 页面中列出的不稳定功能。请注意，某些功能需要最小的 Rust 2018，这些功能需要 Cargo.toml 拥有更改权限才能启用（在下面的部分中描述）。 另请注意，在预览可用期间，我们可能会继续使用此标志来添加/启用新功能！

对于 Rust 2018 预览版2中，我们还测试了新模块路径变体，“统一路径”，我们想要获得进一步测试和反馈。 请尝试将以下内容添加到`lib.rs`或`main.rs`中：

```
#![feature(rust_2018_preview, uniform_paths)]
```

Rust 2018 的发布时候将会选择两个模块路径变体中的一个，并放弃另一个。 The release of Rust 2018 will stabilize one of the two module path variants and drop the other.

修复版本兼容性警告

接下来是启用有关与新 2018 版本 不兼容的代码的编译器警告。 这是方便 `cargo fix` 这个工具进入图片的地方。 要为项目启用兼容性lints，请运行：

```
$ cargo fix --edition
```

如果 nightly 不是你的默认选择的话，你需要运行下面的这个命令：

```
$ cargo +nightly fix --edition
```

这将指示 Cargo 编译项目中的所有目标（库，二进制文件，测试等），同时启用所有 Cargo 功能并为2018版本做好准备。 Cargo 可能会自动修复一些文件，并在其发生时通知您。 请注意，这不会启用任何新的 Rust 2018 功能; 它只能确保您的代码与 Rust 2018 兼容。

如果Cargo无法自动修复所有内容，它将打印出剩余的警告。继续运行上述命令，直到所有警告都已解决。

你可以获取更多 `cargo fix` 信息，运行：

```
$ cargo fix --help
```

切换到下一个版本

一旦您对这些更改感到满意，就可以使用新版本了。 将其添加到您的 `Cargo.toml`：

```
cargo-features = ["edition"]

[package]
edition = '2018'
```

那个 `cargo-features` 行应该排在最前面; `edition` 进入 `[package]` 部分。 如上所述，现在这是 Cargo 的 nightly 特征，因此您需要启用它才能使其工作。

此时，您的项目应该使用常规的 `cargo build` 进行编译。 如果没有，这是一个错误！ 请提交问题。

在新版本中编写惯用代码

你的 crate 现在已经进入了2018版的 Rust，恭喜！ 回想一下，Rust 中的 Editions 表示随着时间的推移，习惯用语的转变。 虽然很多旧代码将继续编译，但今天可能会用不同的习惯用语编写。

您可以采取的可选的后续步骤是将代码更新为新版本中的惯用语。 这是通过一组不同的“习惯用语 lints”完成的。 就像之前我们使用 `cargo fix` 来推动这个过程一样：

```
$ cargo fix --edition-idioms
```

与之前一样，这是一个 *简单* 的步骤。 在这里 `cargo fix` 将自动修复任何可能的lint，所以你只会得到 `cargo fix` 无法修复的情况下的警告。 如果您发现难以完成警告，那就是一个错误！

一旦你用这个命令警告没有了，你就可以继续了。

> `--edition-idioms` 标志仅适用于“当前 crate”，如果你想在工作空间运行它是必要的，使用 `RUSTFLAGS` 的解决方法，以便在所有工作区中执行它。
>
> ```
> $ RUSTFLAGS='-Wrust_2018_idioms' cargo fix --all
> ```

享受新版本吧！

Rust 2015

Rust 2015 的主题是“稳定性”。 它从1.0版本开始，是“默认版”。 该版本系统于2017年底构思，但 Rust 1.0 于2015年5月发布。因此，2015年是您未指定任何特定版本时获得的版本，出于向后兼容性原因。

“稳定性”是 Rust 2015 的主题，因为1.0标志着 Rust 开发的巨大变化。 在 Rust 1.0 之前，Rust 每天都在变化。 这使得在 Rust 中编写大型软件变得非常困难，并且难以学习。 随着 Rust 1.0 和 Rust 2015 的发布，我们致力于向后兼容，确保为人们构建项目奠定坚实的基础。

由于它是默认版本，因此无法将代码移植到 Rust 2015; 它 *就是*。 你将从 2015 开始 *过渡*，但从未真正 *到* 2015版。因此，没有什么可说的了！

Rust 2018

该版本系统是为 Rust 2018 的发布而创建的. Rust 2018 的主题是*生产力*。 Rust 2018 通过新功能，在某些情况下更简单的语法，更智能的借用检查器以及许多其他东西来改进 Rust 2015。 这些都是为了提高生产力目标。 Rust 2015 是一个基础; Rust 2018 使粗糙边缘平滑，使编写代码更简单，更容易，并消除了一些不一致性。

模块系统

在本指南的这一章中，我们将讨论模块系统的一些变化。 其中最值得注意的是 路径清晰度变化。

原始标识符

Minimum Rust Version nightly

与许多编程语言一样，Rust 具有“关键字”的概念。 这些标识符对语言有意义，因此你不能在变量名，函数名和其他位置使用它们。 原始标识符允许你使用通常不允许的关键字。

举个例子，`match` 是一个关键字。如果你试图编译这个方法：

```
fn match(needle: &str, haystack: &str) -> bool {
    haystack.contains(needle)
}
```

你将得到以下错误：

```
error: expected identifier, found keyword `match`
 --> src/main.rs:4:4
  |
4 | fn match(needle: &str, haystack: &str) -> bool {
  |    ^^^^^ expected identifier, found keyword
```

你可以使用原始标识符来实现：

```
#![feature(rust_2018_preview)]
#![feature(raw_identifiers)]

fn r#match(needle: &str, haystack: &str) -> bool {
    haystack.contains(needle)
}

fn main() {
    assert!(r#match("foo", "foobar"));
}
```

注意 `r#` 不仅在定义的时候有，在调用的时候也得有。

更多的细节

此功能还是有一些用处的，但主要动机是版本间的情况。 例如，`try` 不是2015版的关键字，而是2018版的。 因此，如果你有一个用 Rust 2015 编写并具有 `try` 函数的库，要在 Rust 2018 中调用它，你需要使用原始标识符。

新的关键字

2018 中新定义的关键字:

`async` and `await`

这里, 保留 `async` 用来实现 `async fn` 或者 `async ||` 闭包 和 `async { .. }` 块。 同时， 保留 `await` 用来保持 `await!(expr)` 这种语法是一个开放的选项。有关详细信息，请参阅 RFC 2394。

`try`

`do catch { .. }` 块已经被重命名为 `try { .. }` 并已经得到支持, 关键字 `try` 在2018版中将被保留. 有关详细信息，请参阅 RFC 2388。

路径清晰化

Minimum Rust Version nightly

对于刚接触 Rust 的人来说，模块系统通常是最困难的事情之一。 当然，每个人掌握东西的时间都不同，但是有一个根本原因，导致了为什么对许多人来说模块系统如此混乱： 尽管模块系统有了简单而一致的定义规则，但它们给人的感觉可能不一致，甚至是违反直觉的，神秘的。

因此，Rust 2018 引入了一些新的模块系统功能，它们将*简单化*模块系统，使其更加清晰。

注意：在2018版预览期间，正在考虑的模块系统有两种变体，“统一路径(uniform paths)”变体和“锚定使用路径(anchored use paths)”变体。 这些变化大多数是适用于两种变体的; 两个变体部分也列出了两者之间的差异。 我们鼓励使用预览2版本的用户，引入新的“统一路径”变体。Rust 2018 的稳定发布时，将只会选择其一。

要使用新的“统一路径”变体测试 Rust 2018 ， 请将 `#![feature(rust_2018_preview, uniform_paths)]` 放在 `lib.rs` 或 `main.rs` 的顶部。

这是一个简短的总结：

- `extern crate` 不再需要。
- `crate` 关键字指的是当前的 crate。
- 统一路径变体： 路径在 `use` 声明和其他代码中统一工作。路径在顶级模块和子模块中统一工作。任何路径都可以以crate开头，包括 `crate` ， `super` 或 `self` ，或者具有相对于当前模块的本地名称。
- 锚定使用路径变量： `use` 声明中的路径始终以包名称开头，或者以 `crate` ， `super` 或 `self` 开头。 除了 `use` 声明之外的代码中的路径也可以从相对于当前模块的名称开始。
- `foo.rs` 和 `foo /` 子目录可以共存; 将子模块放在子目录中时不再需要 `mod.rs` 。

这样看起来就像是新的规则，但现在心理模型整体上大大简化了。 请阅读以获得更多详情！

更多的细节

让我们依次讨论每个新功能。

不再需要 `extern crate`

这个非常简单：您不再需要编写 `extern crate` 来将crate导入到您的项目中。 之前：

```
// Rust 2015

extern crate futures;

mod submodule {
    use futures::Future;
}
```

现在:

```
// Rust 2018

mod submodule {
    use futures::Future;
}
```

现在，要为项目添加一个新的包，你可以将它添加到你的 `Cargo.toml`，然后没有第二步。 如果你没有使用Cargo，你必须通过 `--extern` 标志给 `rustc` 提供外部crate的位置，然后继续做其他的事前吧。

`extern crate` 的另一个用途是导入宏; 那也不再需要了。 查看宏章节以获取更多信息。

`crate` 指当前crate.

在 `use` 声明和其他代码中，您可以使用 `crate::` 前缀来引用当前包的根。 例如，`crate::foo::bar` 将始终引用模块 `foo` 中的名称 `bar`，来自同一个crate中的任何其他位置。

前缀 `::` 以前称为crate root或外部crate; 它现在毫无疑问的是指外部crate。 例如，`::foo::bar` 总是指外部crate中 `foo` 中的 `bar`。

统一路径变体

与 Rust 2015 相比，Rust 2018 的统一路径变体简化并统一了路径处理。 在 Rust 2015 中，路径在 `use` 声明中的工作方式与在其他地方的工作方式不同。 特别地，`use` 声明中的路径总是从包根开始，而其他代码中的路径隐含地从当前模块开始。 这些差异在顶级模块中没有任何影响，这意味着在处理足够大的子模块项目之前，所有内容都会显得简单明了。

在 Rust 2018 的统一路径变体中，`use` 声明和其他代码中的路径始终以相同的方式工作，无论是在顶级模块还是在任何子模块中。 您始终可以使用当前模块的相对路径，从外部包名称开始的路径，或以 `crate`， `super` 或 `self` 开头的路径。

代码长这样：

```rust
// Rust 2015

extern crate futures;

use futures::Future;

mod foo {
    pub struct Bar;
}

use foo::Bar;

fn my_poll() -> futures::Poll { ... }

enum SomeEnum {
    V1(usize),
    V2(String),
}

fn func() {
    let five = std::sync::Arc::new(5);
    use SomeEnum::*;
    match ... {
        V1(i) => { ... }
        V2(s) => { ... }
    }
}
```

在 Rust 2018 中看起来完全一样，除了删除 `extern crate` 行：

```
// Rust 2018 (uniform paths variant)

use futures::Future;

mod foo {
    pub struct Bar;
}

use foo::Bar;

fn my_poll() -> futures::Poll { ... }

enum SomeEnum {
    V1(usize),
    V2(String),
}

fn func() {
    let five = std::sync::Arc::new(5);
    use SomeEnum::*;
    match ... {
        V1(i) => { ... }
        V2(s) => { ... }
    }
}
```

但是，使用 Rust 2018，相同的代码也可以在子模块中完全不修改：

```
// Rust 2018 (uniform paths variant)

mod submodule {
    use futures::Future;

    mod foo {
        pub struct Bar;
    }

    use foo::Bar;

    fn my_poll() -> futures::Poll { ... }

    enum SomeEnum {
        V1(usize),
        V2(String),
    }

    fn func() {
        let five = std::sync::Arc::new(5);
        use SomeEnum::*;
        match ... {
            V1(i) => { ... }
            V2(s) => { ... }
        }
    }
}
```

这样可以轻松地在项目中移动代码，并避免为引入多模块项目增加额外的复杂性。

如果路径不明确，例如，如果您有外部包和本地模块或具有相同名称的项目，您将收到错误，并且您需要重命名其中一个冲突名称或明确消除路径歧义。 要明确消除路径歧义，请使用 `::name` 作为外部包名，或使用 `self::name` 作为本地模块或项目。

锚定使用路径

在 Rust 2018 的锚定使用路径变体中， `use` 声明 *必须* 必须以包名开头，开头包括 `crate`， `self` 或 `super`。

以前代码长这样：

```
// Rust 2015

extern crate futures;

use futures::Future;

mod foo {
    pub struct Bar;
}

use foo::Bar;
```

现在长这样:

```
// Rust 2018 (anchored use paths variant)

// 'futures' is the name of a crate
use futures::Future;

mod foo {
    pub struct Bar;
}

// 'crate' means the current crate
use crate::foo::Bar;
```

此外，所有这些路径形式也可以在 `use` 声明之外使用，这消除了许多混淆的来源。 在Rust 2015中考虑以下代码：

```
// Rust 2015

extern crate futures;

mod submodule {
    // this works!
    use futures::Future;

    // so why doesn't this work?
    fn my_poll() -> futures::Poll { ... }
}

fn main() {
    // this works
    let five = std::sync::Arc::new(5);
}

mod submodule {
    fn function() {
        // ... so why doesn't this work
        let five = std::sync::Arc::new(5);
    }
}
```

在 `futures` 示例中， `my_poll` 函数签名不正确，因为 `submodule` 不包含名为 `futures` 的项目; 也就是说，这条路径被认为是相对的。 但是因为 `use` 是锚定的， `use futures::` 即使单独的 `futures::` 也不行！ 使用 `std` 它可能会更加令人困惑，因为你从来没有写过 `extern crate std;` 行。 那么为什么它在 `main` 中工作但不在子模块中工作？ 同样的事情：它是一个相对路径，因为它不在 `use` 声明中。 `extern crate std;` 被插入到crate root中，所以它在 `main` 中很好，但它根本不存在于子模块中。

让我们来看看这种变化如何影响：

```
// Rust 2018 (anchored use paths variant)

// no more `extern crate futures;`

mod submodule {
    // 'futures' is the name of a crate, so this is anchored and works
    use futures::Future;

    // 'futures' is the name of a crate, so this is anchored and works
    fn my_poll() -> futures::Poll { ... }
}

fn main() {
    // 'std' is the name of a crate, so this is anchored and works
    let five = std::sync::Arc::new(5);
}

mod submodule {
    fn function() {
        // 'std' is the name of a crate, so this is anchored and works
        let five = std::sync::Arc::new(5);
    }
}
```

更加的直截了当。

不再需要 `mod.rs`

在 Rust 2015 中，子模块如下：

```
mod foo;
```

它可以是 `foo.rs` 或者 `foo/mod.rs`。如果是一个子模块，那么它 *必须* 有一个 `foo/mod.rs`。 这样的化， `bar` 存在于子模块 `foo` 中，将表现为 `foo/bar.rs`。

在 Rust 2018 中， `mod.rs` 不再需要， `foo.rs` 仅仅表示 `foo.rs`，子模块仍然是 `foo/bar.rs`。 这消除了特殊名称，如果你在编辑器中打开了一堆文件，你可以清楚地看到它们的名字，而不是有一堆名为 `mod.rs` 的标签。

更加可见的修饰符

Minimum Rust Version 1.18

您可以使用 `pub` 关键字将某些内容作为模块公共接口的一部分。 但此外，还有一些新形式：

```
pub(crate) struct Foo;

pub(in a::b::c) struct Bar;
```

第一种形式使 `Foo` 结构公开在整个crate中，但不是外部的。 第二种形式是类似的，只在另一种模块 `a::b::c` 中， `Bar` 是公开的。

用 use 进行导入嵌套

Minimum Rust Version 1.25

在 Rust 中： 嵌套导入中添加了一种编写 `use` 语句的新方法。 如果您曾编写过这样的一组导入：

```
use std::fs::File;
use std::io::Read;
use std::path::{Path, PathBuf};
```

可以这样写了：

```
// on one line
use std::{fs::File, io::Read, path::{Path, PathBuf}};

// with some more breathing room
use std::{
    fs::File,
    io::Read,
    path::{
        Path,
        PathBuf
    }
};
```

这可以减少一些重复，并使事情更清晰。

错误处理与崩溃

在本章节中，我们将讨论 Rust 中一些关于错误处理的改进。 最值得注意的是 `?` 操作符介绍.

? 操作符对于早期错误的处理

Minimum Rust Version 1.13 for `Result<T, E>`

Minimum Rust Version 1.22 for `Option<T>`

Rust 已经有了一个新的操作符 `?`，它通过减少视觉干扰来使错误处理变得更加愉快。 它通过解决了一个简单问题来做到这一点。 为了说明，假设我们有段代码，从文件中读取一些数据：

```
fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("username.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

> 注意: 此代码更简单，通过只需一次调用 `std::fs::read_to_string`，但是我们在这里手动编写所有内容以获得多个错误的示例。

此代码有两个可能失败的可能，打开文件和从中读取数据。 如果其中任何一个无法工作，我们想从 `read_username_from_file` 返回错误。 这样做涉及对匹配I/O操作的结果。 在这种简单的情况下，我们只是在调用堆栈中传播错误， 匹配只是样板-它每次都是以相同的模式写出来，但是这并不会为读者提供更多有用的信息。

利用 `?`，上面的代码可以写成这样：

```
fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("username.txt")?;
    let mut s = String::new();

    f.read_to_string(&mut s)?;

    Ok(s)
}
```

`?` 是我们之前写的整个匹配语句的简写。 换句话说，`?` 适用于 `Result` 值，如果它是 `Ok`，它会解开它并给出内部值。 如果它是一个 `Err`，它将从您当前所处的函数返回。在视觉上，它更直接。 现在我们只使用单个 `?` 而不是整个匹配语句。 "?" 字符表示我们在这里以标准方式处理错误，将它们传递给调用堆栈。

经验丰富的 Rustaceans 可能会认识到这与自 Rust `1.0` 以来一直可用的 `try!` 宏相同。 事实上，他们是一样的。以前，`read_username_from_file` 可能是这样实现的：

```
fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = try!(File::open("username.txt"));
    let mut s = String::new();

    try!(f.read_to_string(&mut s));

    Ok(s)
}
```

那么为什么在我们已经拥有宏了，还要扩展语言呢？ 原因有很多。 首先，`try!` 已被证明非常有用，并且常用于惯用的 Rust中。 因为经常被使用，所以我们认为这是值得拥有的语法糖。 这种演化是强大的宏系统的巨大优势之一： 语言语法的推测性扩展可以在不修改语言本身的情况下进行原型化和迭代， 反过来，那些表现出特别有用的宏，可以用来指导制定缺少的语言特征。这种演变，从 `try!` 到 `?` 就是一个很好的例子。

其中一个原因 `try!` 需要一个更甜的语法，当连续使用 `try!` 的多次调用时，这是非常没有吸引力的。 考虑：

```
try!(try!(try!(foo()).bar()).baz())
```

作为对比：

```
foo()?.bar()?.baz()?
```

第一个是非常难以直观阅读的，每个错误处理层都在表达式前加上一个额外的 `try!` 调用。 这会引起过度关注琐碎的错误传播，模糊主代码执行流程，在本例中调用 `foo`， `bar` 和 `baz`。 这种与错误处理链接的方法发生在构建器模式等情况下。

最后，专用语法将使以后更容易生成专门针对 `?` 定制的更好的错误消息，而一般来说很难为宏扩展代码产生好的错误。

您可以将 `?` 与 `Result <T, E>` 一起使用，也可以使用 `Option <T>`。 在这种情况下，`?` 将为 `Some(T)` 返回一个值，并为 `None` 返回 `None`。 一个当前的限制是你不能在同一个函数中反复使用 `?`，因为返回类型需要匹配你使用的类型 `?`。 将来，这种限制将是有限的。

`?` 在 `main` 和 tests 中

Minimum Rust Version 1.26

Rust的错误处理围绕返回 `Result <T, E>` 并使用 `?` 传播错误。 对于那些编写许多小程序并且希望进行许多测试的人来说，更关注于那些复杂的入口，例如`main`和`#[test]`中的错误处理。

举个例子，你将尝试这样写：

```
use std::fs::File;

fn main() {
    let f = File::open("bar.txt")?;
}
```

因为 `?` 通过处理 `Result` 并提前返回函数来工作，所以上面的代码不起作用，并且导致以下错误：

```
error[E0277]: the `?` operator can only be used in a function that returns
`Result`
              or `Option` (or another type that implements `std::ops::Try`)
 --> src/main.rs:5:13
  |
5 |     let f = File::open("bar.txt")?;
  |             ^^^^^^^^^^^^^^^^^^^^^^ cannot use the `?` operator in a function
that returns `()`
  |
  = help: the trait `std::ops::Try` is not implemented for `()`
  = note: required by `std::ops::Try::from_error`
```

在 Rust 2015 中，处理这种问题，需要这样：

```
// Rust 2015


fn run() -> Result<(), Box<Error>> {
    // real logic..
    Ok(())
}

fn main() {
    if let Err(e) = run() {
        println!("Application error: {}", e);
        process::exit(1);
    }
}
```

但是，在这种情况下，`run` 函数具有所有有趣的逻辑，而 `main` 只是样板。 问题更糟糕的是 `#[test]`，因为它们往往会有更多这种情况。

在 Rust 2018 中，你可以使得你的 `#[test]` 和 `main` 函数返回一个 `Result`：

```
// Rust 2018

use std::fs::File;

fn main() -> Result<(), std::io::Error> {
    let f = File::open("bar.txt")?;

    Ok(())
}
```

在这种情况下，如果说文件不存在并且某处有一个 `Err(err)`，那么 `main` 将以错误代码（不是 `0`）退出并打印出 `Debug` 表示 `err`。

更多的细节

使 `-> Result <..>` 在 `main` 和 `#[test]` 的上下文中工作并不神奇。 它全部由 `Termination` 特征支持，所有有效的返回类型的 `main` 和测试函数必须实现。 特征定义为：

```

pub trait Termination {
    fn report(self) -> i32;
}
```

在为应用程序设置入口点时，编译器将使用此特征并在您编写的 `main` 函数的 `Result` 上调用 `.report()`。

`Result` 和 `()` 的这个特性的两个简化示例实现是：

```
impl Termination for () {
    fn report(self) -> i32 {
        ExitCode::SUCCESS.report()
    }
}

impl<E: fmt::Debug> Termination for Result<(), E> {
    fn report(self) -> i32 {
        match self {
            Ok(()) => ().report(),
            Err(err) => {
                eprintln!("Error: {:?}", err);
                ExitCode::FAILURE.report()
            }
        }
    }
}
```

正如您在 `()` 中看到的那样，只返回成功代码。 在 `Result` 的情况下，成功的话交给 `()` 来执行，错误的话，交给 `Err(..)`，打印出错误消息并退出代码。

要了解有关更细节的信息，请参阅跟踪问题 或者 the RFC.

通过 `std::panic` 处理崩溃

Minimum Rust Version 1.9

有一个 `std::panic` 模块，其中包含崩溃，停止和启动的展开过程的方法：

```
use std::panic;

let result = panic::catch_unwind(|| {
    println!("hello!");
});
assert!(result.is_ok());

let result = panic::catch_unwind(|| {
    panic!("oh no!");
});
assert!(result.is_err());
```

通常，Rust区分操作失败的两种方式：

- 由于 *预期的问题*，就像找不到文件一样。
- 由于 *意外问题*，就像索引超出数组范围一样。

预期的问题通常来自您无法控制的情况; 应该为其环境可能抛出的任何内容准备健壮的代码。 在Rust中，预期的问题通过 `Result` 类型 来处理，它允许函数将有关问题的信息返回给调用者，然后调用者可以以细粒度的方式处理错误。

意外问题是*错误*：它们是由于合同或断言被违反而产生的。由于它们是意料之外的，因此以细粒度的方式处理它们是没有意义的。 相反，Rust通过*崩溃*采用“快速失败”方法，默认情况下解除发现错误的线程的堆栈（运行析构函数但没有其他代码）。 其他线程继续运行，但每当他们尝试与崩溃线程（无论是通过通道还是共享内存）进行通信时，都会发现崩溃。 因此，崩溃将执行中止到一些“隔离边界”，边界另一侧的代码仍然可以运行，并且可能以某种非常粗粒度的方式从崩溃中“恢复”。 例如，服务器不一定因为其中一个线程中的断言失败而需要关闭。

同样值得注意的是，程序可能会选择*中止*而不是放松，因此捕捉崩溃可能无效。如果你的代码依赖于 `catch_unwind`，你应该将它添加到你的Cargo.toml：

```
[profile.debug]
panic = "unwind"

[profile.release]
panic = "unwind"
```

如果您的任何用户选择中止，他们将遇到编译时失败。

`catch_unwind` API提供了一种在线程中*引入新的隔离边界*的方法。 有几个关键的刺激例子：

- 在其他语言中嵌入 Rust
- 管理线程的抽象
- 测试框架，因为测试可能会引起崩溃，你不希望它会杀死测试运行器

对于第一种情况，跨语言边界展开是未定义的行为，并且经常导致实践中的段错误。 允许捕获崩溃意味着您可以通过 C API 安全地公开 Rust 代码，并将展开转换为C侧的错误。

对于第二种情况，请考虑一个线程池库。如果池中的线程发生混乱，您通常不希望杀死线程本身，而是抓住崩溃并将其传递给池的客户端。 `catch_unwind` API 与 `resume_unwind` 配对，然后可以用它来重新启动它所属的池的客户端上的崩溃过程。

在这两种情况下，您都在一个线程中引入了一个新的隔离边界，然后将崩溃转换为其他地方的其他形式的错误。

中止崩溃

Minimum Rust Version 1.10

默认情况下，当发生 `panic!` 时，Rust 程序将展开堆栈。如果你更喜欢立即中止，你可以在 `Cargo.toml` 中配置它：

```
[profile.debug]
panic = "abort"

[profile.release]
panic = "abort"
```

你为什么选择这样做？通过删除对展开的支持，你将获得更小的二进制文件。你将失去捕捉崩溃的能力。哪种选择取决于你正在做什么。

流程控制

在本章节中，我们将讨论流程控制的改进，更多的关注点在 `async` 和 `await`。

`loop` 可以 break 并携带返回值

Minimum Rust Version 1.19

`loop` 可以 break 并携带返回值

```
// old code
let x;

loop {
    x = 7;
    break;
}

// new code
let x = loop { break 7; };
```

Rust 传统上将自己定位为“面向表达式的语言”，也就是说，大多数事物都是评估价值而不是陈述表达。 `loop` 以这种方式突然变得奇怪，因为它之前是一个声明。

现在，这只适用于 `loop`，而不适用于 `while` 或 `for`。 目前尚不清楚，但我们可能会将此添加到未来。

async/await 早期的并发

Minimum Rust Version nightly

Rust 2018 的初始版本不会附带 `async` / `await` 支持，但是我们保留了关键字，以便将来的版本包含它们。我们将在接近发布的时候更新此页面！

Trait 系统

在本章，我们将讨论关于 trait 系统的改进，需要特别关注 `impl Trait`。

`impl Trait` 轻松返回复杂的类型

Minimum Rust Version 1.26

`impl Trait` 是指定实现特定特征的未命名但有具体类型的新方法。 你可以把它放在两个地方：参数位置和返回位置。

```
trait Trait {}

// argument position
fn foo(arg: impl Trait) {
}

// return position
fn foo() -> impl Trait {
}
```

参数位置

在参数位置上，这个特性是十分简单的，下面这两种写法几乎相同：

```
trait Trait {}

fn foo<T: Trait>(arg: T) {
}

fn foo(arg: impl Trait) {
}
```

也就是说，它是泛型类型参数的简短的语法。这意味着，“ `arg` 是一个参数，它可以是实现了 `Trait` 特征的任何类型。”

但是，在技术上， `T: Trait` 和 `impl Trait` 有着一个很重要的不同点。 当你编写前者时，可以使用turbo-fish语法在调用的时候指定 `T` 的类型，如 `foo::<usize>(1)` 。 在 `impl Trait` 的情况下，只要它在函数定义中使用了，不管什么地方，都不能再使用turbo-fish。 因此，您应该注意，更改两者和切换到 `impl Trait` 都会对代码的用户构成重大变化。

返回参数

在返回位置，此功能更有趣。这意味着“我正在返回一些实现了 `Trait` 特征的类型，但我不打算告诉你究竟是什么类型。” 在 `impl Trait` 之前，你可以用特征对象做到这一点：

```
trait Trait {}

impl Trait for i32 {}

fn returns_a_trait_object() -> Box<dyn Trait> {
    Box::new(5)
}
```

但是，这会产生一些开销： `Box <T>` 表示这里有堆分配，这将使用动态分配。 有关此语法的说明，请参阅 `dyn Trait` 部分。 但是我们在这里只返回一个可能的东西，即 `Box <i32>` 。 这意味着我们即使我们不使用动态分配，但是依旧为它而付出了代价！

使用 `impl Trait` ，上面的代码如下：

```
trait Trait {}

impl Trait for i32 {}

fn returns_a_trait_object() -> impl Trait {
    5
}
```

在这里，我们没有 `Box <T>` ，没有特征对象，也没有动态分配。但我们仍然可以实现 `i32` 的返回类型。

使用 `i32` ，这看起来并不是非常有用。但 Rust 中有一个主要运用的地方，它更有用： 闭包。

`impl Trait` 和闭包

> 如果你想要了解闭包，参阅 their chapter in the book.

在 Rust 中，闭包具有独特的，不可写的类型。然而，他们确实实现了 `Fn` 系列的特征。 这意味着，在以前，从函数处返回闭包的唯一方法是，使用trait对象：

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

你不能写明闭包的类型，仅仅是使用 `Fn` 特性。这意味着特征对象是必须的，但是，利用 `impl Trait` 呢：

```
fn returns_closure() -> impl Fn(i32) -> i32 {
    |x| x + 1
}
```

我们现在可以直接返回闭包类型，就像其他的返回值那样！

更多的细节

以上是你需要了解和使用 `impl Trait` 的所有内，但是还有一些更细微的细节： 类型参数和参数位置 `impl Trait` 的普遍性（普遍量化的类型）。 同时， `impl Trait` 在返回位置是存在的（存在量化的类型）。 好吧，也许这有点太行话了。 我们退一步吧。

考虑下面这个函数:

```
fn foo<T: Trait>(x: T) {
```

当你调用它时，你设置类型， `T` 。 “你”是这里的调用者。 这个签名说“我接受任何实现Trait的类型”。 （“任何类型” == 行话中的*通用*）

这个版本:

```
fn foo<T: Trait>() -> T {
```

相似但是有些不同，你这个调用者，提供一个你想要的返回类型 `T` ，你可以看一下现在 Rust 中的 parse 和 collect :

```
let x: i32 = "5".parse()?;
let x: u64 = "5".parse()?;
```

这里， `.parse` 有如下的签名：

```
pub fn parse<F>(&self) -> Result<F, <F as FromStr>::Err> where
    F: FromStr,
```

如你所想，虽然结果类型和 `FromStr` 有一个相关的类型... 无论如何，你可以看到 `F` 在这里的返回位置。所以你有能力去选择了。

使用 `impl Trait`，你会说“嘿，有些类型存在实现这个特性，但我不会告诉你它是什么。”（术语中的“存在主义”，“某种类型存在”）。 所以现在，调用者无法选择，函数本身可以选择。如果我们尝试使用 `Result <impl F, ...` 定义解析作为返回类型，它将无效。

使用 `impl Trait` 在更多的地方

如前所述，作为一个开始，您将只能使用 `impl Trait` 作为自由或固有函数的参数或返回类型。 但是，现在 `impl Trait` 不能在traits的实现中使用，也不能用作let绑定的类型或类型别名。未来其中一些限制将被取消。 获取更多的信息，查看tracking issue on `impl Trait`.

`dyn Trait` trait 对象

Minimum Rust Version 1.27

`dyn Trait` 是使用 trait 对象的新语法，简而言之：

- `Box<Trait>` becomes `Box<dyn Trait>`
- `&Trait` and `&mut Trait` become `&dyn Trait` and `&mut dyn Trait`

因此，代码中:

```
trait Trait {}

impl Trait for i32 {}

// old
fn function1() -> Box<Trait> {
}

// new
fn function2() -> Box<dyn Trait> {
}
```

这就是了!

更多细节

仅仅使用特征对象的特征名其实是个糟糕的决定。目前的语法通常含糊不清，即使对于老一批人来来说也是如此， 而且竟然没有它的替代品使用的更频繁，有时速度较慢，而且当其替代品可以使用时，它将根本不会被使用。

此外，随着 `impl Trait` 的到来， `impl Trait` vs `dyn Trait` 比 `impl Trait` vs `Trait` 更好更对称。 `impl Trait` 将在下一节进一步解释。

因此，在新版本中，选择使用 trait 对象时，你应该选 `dyn Trait` 而不是 `Trait`。

支持 trait 对象的更多容器类型

Minimum Rust Version 1.2

在 Rust 1.0 中，只有某些特殊的类型可以创建成 trait objects.

在 Rust 1.2 中，这种限制被解除了，更多的类型可以做到这一点。 例如， `Rc<T>`，Rust 的引用计数类型之一：

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

trait Foo {}

impl Foo for i32 {

}

fn main() {
    let obj: Rc<dyn Foo> = Rc::new(5);
}
```

这段代码在 Rust 1.0 中无法执行，但是现在可以了。

> 如果您之前没有看过 `dyn` 语法，请参阅相关章节。对于不支持它的版本，将 `Rc <dyn Foo>` 替换为 `Rc <Foo>`。

相关常数

Minimum Rust Version 1.20

你可以定义具有“关联函数”的 traits, structs, enums：

```
struct Struct;

impl Struct {
    fn foo() {
        println!("foo is an associated function of Struct");
    }
}

fn main() {
    Struct::foo();
}
```

这个叫做“关联函数”，因为它关联了相关的类型，也就是说，它们附加到类型本身，而不是任何特定的实例。

Rust 1.20 中为关联函数增加了新的功能：

```
struct Struct;

impl Struct {
    const ID: u32 = 0;
}

fn main() {
    println!("the ID of Struct is: {}", Struct::ID);
}
```

其中，常量 `ID` 关联到 `Struct` 上，如果函数一样，关联常量也可以工作在 trait 和 enum 上。

Trait 具有额外的能力和相关的常数，为他们提供额外的力量。使用特征，您可以像使用关联类型一样 使用关联常量： 通过声明它，但不给它一个值。然后，特征的实现者在实现时声明其值：

```
trait Trait {
    const ID: u32;
}

struct Struct;

impl Trait for Struct {
    const ID: u32 = 5;
}

fn main() {
    println!("{}", Struct::ID);
}
```

在此功能之前，如果要创建表示浮点数的特征，则必须如下：

```
trait Float {
    fn nan() -> Self;
    fn infinity() -> Self;
    // ...
}
```

这有点笨拙，但更重要的是，因为它们是函数，所以它们不能用于常量表达式，即使它们只返回常量。 因此，`Float` 的设计也必须包含常量：

```
mod f32 {
    const NAN: f32 = 0.0f32 / 0.0f32;
    const INFINITY: f32 = 1.0f32 / 0.0f32;

    impl Float for f32 {
        fn nan() -> Self {
            f32::NAN
        }
        fn infinity() -> Self {
            f32::INFINITY
        }
    }
}
```

关联常量让你可以更清晰的表达，如下：

```
trait Float {
    const NAN: Self;
    const INFINITY: Self;
    // ...
}
```

继续实现如下：

```
mod f32 {
    impl Float for f32 {
        const NAN: f32 = 0.0f32 / 0.0f32;
        const INFINITY: f32 = 1.0f32 / 0.0f32;
    }
}
```

更加清晰，更通用。

切片模式

Minimum Rust Version 1.26

你有没有试过，用模式匹配去匹配切片的内容和结构？ Rust 2018 将让你做到这一点。

例如，我们想要接受一个名单列表并回复问候语。使用切片模式，我们可以用以下方式轻松完成：

```
fn main() {
    greet(&[]);
    // output: Bummer, there's no one here :(
    greet(&["Alan"]);
    // output: Hey, there Alan! You seem to be alone.
    greet(&["Joan", "Hugh"]);
    // output: Hello, Joan and Hugh. Nice to see you are at least 2!
    greet(&["John", "Peter", "Stewart"]);
    // output: Hey everyone, we seem to be 3 here today.
}

fn greet(people: &[&str]) {
    match people {
        [] => println!("Bummer, there's no one here :("),
        [only_one] => println!("Hey, there {}! You seem to be alone.", only_one),
        [first, second] => println!(
            "Hello, {} and {}. Nice to see you are at least 2!",
            first, second
        ),
        _ => println!("Hey everyone, we seem to be {} here today.", people.len()),
    }
}
```

现在，你不必检查长度了。

你也可以匹配 array 如下：

```
let arr = [1, 2, 3];

assert_eq!("ends with 3", match arr {
    [_, _, 3] => "ends with 3",
    [a, b, c] => "ends with something else",
});
```

更多的细节

穷举模式

在第一个例子中，注意匹配的 `_ => ...`。 如果开始匹配，那么将会匹配一切情况，所以有"穷尽所有模式"的处理方式。 如果我们忘记使用 `_ => ...` 或者 `identifier => ...` 模式，我们会得到如下的错误提醒：

```
error[E0004]: non-exhaustive patterns: `&[_, _, _]` not covered
```

如果我们再增加一项，我们将得到如下：

```
error[E0004]: non-exhaustive patterns: `&[_, _, _, _]` not covered
```

如此。

数组和精确的长度

在第二个例子中，数组是有固定长度的，我们需要匹配所有长度项，如果只匹配2，4项的话，会报错：

```
error[E0527]: pattern requires 2 elements but array has 3
```

和

```
error[E0527]: pattern requires 4 elements but array has 3
```

管道中

在切片模式方面，计划采用更先进的形式，但尚未稳定。要了解更多信息，请跟踪 the tracking issue。

所有权和生命周期

在本章节，我们讨论所有权和生命周期的改进，最值得关注的是 默认 match 绑定模式。

默认匹配绑定

Minimum Rust Version 1.26

你有过借用 `Option<T>` 然后进行匹配的经历嘛？你可能写成下面这样：

```
let s: &Option<String> = &Some("hello".to_string());

match s {
    Some(s) => println!("s is: {}", s),
    _ => (),
};
```

在 Rust 2015，这样写是错的，你必须这样写：

```
// Rust 2015

let s: &Option<String> = &Some("hello".to_string());

match s {
    &Some(ref s) => println!("s is: {}", s),
    _ => (),
};
```

在 Rust 2018， 对比之下，会自动推断 `&` 和 `ref`，然后你再这样写就没问题了。

这个并不仅仅影响 `match`，也会影响到很多地方，比如 `let` 表达式，闭包以及 `for` 循环。

更多的细节

模式的心理预期模型随着这种变化而略有改变，使其与语言的其他方面保持一致。 例如，在编写 `for` 循环时，您可以通过借用集合本身来迭代集合的借用内容：

```
let my_vec: Vec<i32> = vec![0, 1, 2];

for x in &my_vec { ... }
```

这个想法是 `&T` 可以被理解为 `T` *的借用视图*，所以当你迭代，匹配或以其他方式构造一个 `&T` 时，你可以借用它的内部视图。

更正式地说，模式具有“绑定模式”，它可以是值（ `x` ），引用（ `ref x` ），也可以是可变引用（ `ref mut x` ）。 在 Rust 2015 中， `match` 总是以by-value模式启动，并要求你在模式中显式写

`ref` 或 `ref mut` 以切换到借用模式。 在 Rust 2018 中，匹配的值的类型通知绑定模式，因此如果您使用 `Some` 变量匹配 `&Option <String>`，您将自动进入 `ref` 模式， 为您提供借用查看内部数据。 类似地， `&mut Option <String>` 会给你一个 `ref mut` 视图。

'_ — 匿名生命周期

Minimum Rust Version nightly

Rust 2018 允许你明确标记生命周期被省略的地方，对于此省略可能不清楚的类型。 要做到这一点，你可以使用特殊的生命周期 `'_`，就像你可以用语法 `let x: _ = ..;` 明确标记一个类型一样。

要我们说的话，无论出于什么原因，我们在 `&'a str` 周围有一个简单的封装：

```
struct StrWrap<'a>(&'a str);
```

在 Rust 2015，你可能写成这样：

```
// Rust 2015

use std::fmt;


fn make_wrapper(string: &str) -> StrWrap {
    StrWrap(string)
}

impl<'a> fmt::Debug for StrWrap<'a> {
    fn fmt(&self, fmt: &mut fmt::Formatter) -> fmt::Result {
        fmt.write_str(self.0)
    }
}
```

在 Rust 2018，你可以替换成这样：

```
#![feature(rust_2018_preview)]


// Rust 2018

fn make_wrapper(string: &str) -> StrWrap<'_> {
    StrWrap(string)
}

impl fmt::Debug for StrWrap<'_> {
    fn fmt(&self, fmt: &mut fmt::Formatter<'_>) -> fmt::Result {
        fmt.write_str(self.0)
    }
}
```

更多的细节

在上面的 Rust 2015 片段中，我们使用了 `-> StrWrap`。 但是，除非你看一下 `StrWrap` 的定义，否则返回的值实际上是借用了什么并不清楚。 因此，从 Rust 2018 开始，不推荐使用它来省去非引用类型 的生命周期参数（除了 `&` 和 `&mut` 之外的类型）。 相反，你之前写过 `-> StrWrap` 的地方， 你现在应该写 `-> StrWrap <'_>`，明确说明正在进行借用。

`'_` 究竟是什么意思？这取决于具体情况！在输出上下文中，与 `make_wrapper` 的返回类型一样， 它指的是所有“输出”位置的单个生命周期。在输入上下文中，为每个“输入位置”生成新的生命周期。 更具体地说，要了解输入上下文，请考虑以下示例：

```
// Rust 2015

struct Foo<'a, 'b: 'a> {
    field: &'a &'b str,
}

impl<'a, 'b: 'a> Foo<'a, 'b> {
    // some methods...
}
```

我们可以重写为：

```
#![feature(rust_2018_preview)]


// Rust 2018

impl Foo<'_, '_> {
    // some methods...
}
```

这是相同的，因为对于每个`'_`，会产生一个新的生命周期。最后，必须坚持结构所需的关系 `'a: 'b`。

更多的细节，参阅tracking issue on In-band lifetime bindings。

在 impl 中省略生命周期

Minimum Rust Version nightly

在编写 `impl` 时，你可以提及生命周期而不将它们绑定在参数列表中。

在 Rust 2015 中：

```
impl<'a> Iterator for MyIter<'a> { ... }
impl<'a, 'b> SomeTrait<'a> for SomeType<'a, 'b> { ... }
```

在 Rust 2018 中：

```
impl Iterator for MyIter<'iter> { ... }
impl SomeTrait<'tcx> for SomeType<'tcx, 'gcx> { ... }
```

`T: 'a` 结构体中的推导

Minimum Rust Version nightly

一个注释形式为 `T: 'a`，其中 `T` 可以是一个类型或另一个生命周期，被称为 *“outlives”* 要求。 注意 *“outlives”* 也意味着 `'a: 'a`。

2018版在编写程序时帮助您保持流程的一种方法是，不需要在 `struct` 定义中明确注释这些 `T: 'a` 的要求。 相反，这些要求将从定义中的字段推断出来。

考虑下面这个 `struct` 定义，在 Rust 2015 中：

```
// Rust 2015

struct Ref<'a, T: 'a> {
    field: &'a T
}

// or written with a `where` clause:

struct WhereRef<'a, T> where T: 'a {
    data: &'a T
}

// with nested references:

struct RefRef<'a, 'b: 'a, T: 'b> {
    field: &'a &'b T,
}

// using an associated type:

struct ItemRef<'a, T: Iterator>
where
    T::Item: 'a
{
    field: &'a T::Item
}
```

在 Rust 2018 中， 这种需求是可以被推导的，你可以这样写：

```
// Rust 2018

struct Ref<'a, T> {
    field: &'a T
}

struct WhereRef<'a, T> {
    data: &'a T
}

struct RefRef<'a, 'b, T> {
    field: &'a &'b T,
}

struct ItemRef<'a, T: Iterator> {
    field: &'a T::Item
}
```

如果您希望在某些情况下更明确，那仍然是可能的。

更多的细节

更多信息，查阅 the tracking issue 和 the RFC.

在 `static` 和 `const` 中的更简单的生命周期

Minimum Rust Version 1.17

在以往的 Rust 中，在需要的时候，你必须更加明确的在 `static` 或者 `const` 上写明 `'static` 生命周期。

```
const NAME: &'static str = "Ferris";
static NAME: &'static str = "Ferris";
```

但是，在这里 `'static` 是唯一一种可能的生命周期，所以现在你可以不用再写 `'static` 了：

```
const NAME: &str = "Ferris";
static NAME: &str = "Ferris";
```

在某些场景下，这个可以消除很多累赘：

```
// old
const NAMES: &'static [&'static str; 2] = &["Ferris", "Bors"];

// new
const NAMES: &[&str; 2] = &["Ferris", "Bors"];
```

数据类型

在本章节中，我们讨论数据类型的改进，值得关注的是 字段初始化简写。

字段初始化简写

Minimum Rust Version 1.17

在以往的 Rust 中，当初始化一个结构体的时候，总是需要完全按照 `key: value` 对的写法：

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

let a = 5;
let b = 6;

let p = Point {
    x: a,
    y: b,
};
```

但是，这些字段通常会是相同的名字，所以你可以把它写成这样：

```
let p = Point {
    x: x,
    y: y,
};
```

现在，如果变量名和结构体字段名相同，可以省略写成这样：

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

let x = 5;
let y = 6;

// new
let p = Point {
    x,
    y,
};
```

..= 包含取值范围

Minimum Rust Version 1.26

在以前的 Rust 1.0 中，我们像下面这样写一个取值范围：

```
for i in 1..3 {
    println!("i: {}", i);
}
```

这将会打印 `i: 1` 然后是 `i: 2`，现在你可以这么写：

```
for i in 1..=3 {
    println!("i: {}", i);
}
```

这也会打印 `i: 1` 然后是 `i: 2`，最后是 `i: 3`；最后的也将会包含在范围取值中， 当你需要包含取值范围的时候，这会十分有用。下面是一个令人惊奇的例子：

```
fn takes_u8(x: u8) {
    // ...
}

fn main() {
    for i in 0..256 {
        println!("i: {}", i);
        takes_u8(i);
    }
}
```

这个程序做了什么？回答是：没有任何东西。编译器将会报出警告：

```
warning: literal out of range for u8
 --> src/main.rs:6:17
  |
6 |     for i in 0..256 {
  |                 ^^^
  |
  = note: #[warn(overflowing_literals)] on by default
```

这是正常的，因为 `i` 作为一个 `u8`，256超出了范围，这就导致效果和 `for i in 0..0` 一样, 这段代码将执行0次。

但是，我们现在可以这么写：

```rust
fn takes_u8(x: u8) {
    // ...
}

fn main() {
    for i in 0..=255 {
        println!("i: {}", i);
        takes_u8(i);
    }
}
```

这个将会执行256次你需要执行的内容。

128位整型

Minimum Rust Version 1.26

这是一个简单的特性： Rust 现在支持128位的整型了！

```
let x: i128 = 0;
let y: u128 = 0;
```

这将是 u64的两倍大小，并且可以存放更多的值，十分方便。

- `u128`：`0` - `340,282,366,920,938,463,463,374,607,431,768,211,455`
- `i128`：`-170,141,183,460,469,231,731,687,303,715,884,105,728` - `170,141,183,460,469,231,731,687,303,715,884,105,727`

哈!

"Operator-equals" 现在实现了

Minimum Rust Version 1.8

各种各样的 “等价操作符” 已经被各种各样的trait实现了, 比如 `+=` 和 `-=`。 举个例子：下面是一个 `+=` 操作符：

```
use std::ops::AddAssign;

#[derive(Debug)]
struct Count {
    value: i32,
}

impl AddAssign for Count {
    fn add_assign(&mut self, other: Count) {
        self.value += other.value;
    }
}

fn main() {
    let mut c1 = Count { value: 1 };
    let c2 = Count { value: 5 };

    c1 += c2;

    println!("{:?}", c1);
}
```

这将打印 `Count { value: 6 }`.

`union`：一个非安全的 `enum`

Minimum Rust Version 1.19

Rust 现在支持 `unions`：

```
union MyUnion {
    f1: u32,
    f2: f32,
}
```

Unions 是一种如同 enums 的结构，但是，它*没有标签*。 Enums 是带有 “标签” 的，用来存储哪个变种在运行时，正在被使用。unions并没有这个。

由于我们可以使用错误的变体解释 union 中保存的数据，并且 Rust 无法为 我们检查这一点，这意味着读取或写入 union 字段是不安全的：

```
let mut u = MyUnion { f1: 1 };

unsafe { u.f1 = 5 };

let value = unsafe { u.f1 };
```

模式匹配也一样工作：

```
fn f(u: MyUnion) {
    unsafe {
        match u {
            MyUnion { f1: 10 } => { println!("ten"); }
            MyUnion { f2 } => { println!("{}", f2); }
        }
    }
}
```

unions 什么时候有用？ 一个主要的用例是与 C 的互操作性。 C API 可以（并且取决于区域，经常这样做）公开 unions， 因此这使得为这些库编写 API 包装器变得非常容易。 此外，unions 还简化了依赖于值表示的 节省空间或高效缓存的结构的 Rust 实现，例如使用对齐指针的最低有效位来区分情况的机器字大小的 unions。

还有更多改进。 目前，unions 只能包含 `Copy` 类型，可能不会实现 `Drop`。 我们希望将来能够解除这些限制。

repr 属性的对齐方式

Minimum Rust Version 1.25

Wikipedia:

> 现代计算机硬件中的 CPU 在数据自然对齐时，可以最有效地执行对存储器的读写，这通常意味着数据地址是数据大小的倍数。 数据对齐是指根据元素的自然对齐来对齐元素。为了确保自然对齐，可能需要在结构元素之间或结构的最后一个元素之后插入一些填充值。

`#[repr]` 属性有一个新参数 `align`，用于设置结构体的对齐方式：

```
struct Number(i32);

assert_eq!(std::mem::align_of::<Number>(), 4);
assert_eq!(std::mem::size_of::<Number>(), 4);

#[repr(align(16))]
struct Align16(i32);

assert_eq!(std::mem::align_of::<Align16>(), 16);
assert_eq!(std::mem::size_of::<Align16>(), 16);
```

如果您正在使用底层级别的东西，控制这些事情可能非常重要！

一般来说，类型的对齐并不担心，因为编译器将为一般用例选择适当的对齐“做正确的事情”。 但是，在使用外部系统操作时，可能需要非标准对齐。例如，通过自定义对齐，这些情况往往需要或更容易：

- 当硬件实际上仅由4字节值组成时，硬件通常具有模糊的要求，例如“此结构对齐到32字节”。虽然这通常可以手动计算和管理，但将它表达为类型的属性通常也很有用，可以让编译器做一些额外的工作。
- 像 `gcc` 和 `clang` 这样的C编译器提供了为结构指定自定义对齐的能力，如果 Rust 也可以镜像自定义对齐的请求（例如将结构传递给C），Rust 可以更容易地与这些类型进行互操作。正确的更容易）。
- 自定义对齐通常可以在这里和那里用于各种技巧，并且通常很方便，因为“让我们使用实现”工具。例如，这可用于在内核中静态分配页表，或者为并发编程轻松创建至少缓存行大小的结构。

此功能的目的是提供轻量级注释，以更改结构的编译器推断对齐，从而更轻松地启用这些情况。

SIMD 更快的计算

Minimum Rust Version 1.27

SIMD 的基础部分已经可用了！ SIMD 代表“单指令，多数据”。考虑这样的函数：

```
pub fn foo(a: &[u8], b: &[u8], c: &mut [u8]) {
    for ((a, b), c) in a.iter().zip(b).zip(c) {
        *c = *a + *b;
    }
}
```

在这里，我们采用两个切片，并将数字加在一起，将结果放在第三个切片中。最简单的方法是完成代码所做的工作，循环遍历每组元素，将它们添加到一起，并将其存储在结果中。 但是，编译器通常可以做得更好。 LLVM 通常会“自动向量化”这样的代码，这是“使用 SIMD ”的一个奇特术语。 想象一下， `a` 和 `b` 都是16个元素长。每个元素都是一个“u8”，这意味着每个切片都是128位数据。 使用SIMD，我们可以将 `a` 和 `b` 放入128位寄存器，将它们一起添加到*single*指令中，然后将得到的128位复制到 `c` 中。那要快得多！

虽然稳定的Rust总是能够利用自动向量化，但有时候，编译器并不够聪明，不能意识到我们可以做这样的事情。 此外，并非每个CPU都有这些功能，因此 LLVM 可能不会使用它们，因此您的程序可以在各种硬件上使用。 `std::arch` 模块允许我们直接使用这些指令，这意味着我们不需要依赖智能编译器。 此外，它还包含一些功能，允许我们根据各种标准选择特定的实现。例如：

```
#[cfg(all(any(target_arch = "x86", target_arch = "x86_64"),
      target_feature = "avx2"))]
fn foo() {
    #[cfg(target_arch = "x86")]
    use std::arch::x86::_mm256_add_epi64;
    #[cfg(target_arch = "x86_64")]
    use std::arch::x86_64::_mm256_add_epi64;

    unsafe {
        _mm256_add_epi64(...);
    }
}
```

在这里，我们使用 cfg 标志根据我们定位的机器选择正确的版本; 在 x86 上我们使用该版本，在 x86_64 上我们使用它的版本。 我们也可以在运行时选择：

```
fn foo() {
    #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
    {
        if is_x86_feature_detected!("avx2") {
            return unsafe { foo_avx2() };
        }
    }

    foo_fallback();
}
```

在这里，我们有两个版本的功能：一个使用 AVX2，一种特定的 SIMD 功能，可以让你进行256位操作。 `is_x86_feature_detected!` 宏将生成检测 CPU 是否支持 AVX2 的代码，如果是，则调用 foo_avx2 函数。如果没有，那么我们回到非 AVX 实现 foo_fallback。 这意味着我们的代码将在支持 AVX2 的CPU上运行得非常快，但仍然可以在不支持 AVX2 的CPU上运行，尽管速度较慢。

如果所有这一切看起来都有点低级和狡猾，那就好了！ `std::arch` 特别适用于构建这类东西。 我们希望最终能够在更高级别的东西中稳定一个 `std::simd` 模块。 但从现在开始，这些基础点可以让生态系统开始尝试更高级别的库。

举个例子： 查阅 faster 库. 这是一个没有 SIMD 的代码片段：

```
let lots_of_3s = (&[-123.456f32; 128][..]).iter()
    .map(|v| {
        9.0 * v.abs().sqrt().sqrt().recip().ceil().sqrt() - 4.0 - 2.0
    })
    .collect::<Vec<f32>>();
```

使用 SIMD 的代码将会更快，你需要改成这样：

```
let lots_of_3s = (&[-123.456f32; 128][..]).simd_iter()
    .simd_map(f32s(0.0), |v| {
        f32s(9.0) * v.abs().sqrt().rsqrt().ceil().sqrt() - f32s(4.0) - f32s(2.0)
    })
    .scalar_collect();
```

这看起来差不多： `simd_iter` 取代 `iter`，`simd_map` 取代 `map`，`f32s(2.0)` 取代 `2.0`。但是你需要一个 SIMD-ified 版本。

除此之外，您可能永远不会自己编写任何内容，但与往常一样，您依赖的库可能。 例如，正则表达式包含这些 SIMD 加速，而您根本不需要做任何事情！

宏

在本章节，主要讨论宏系统的改进，特别需要关注的是 自定义 derive。

自定义 Derive

Minimum Rust Version 1.15

在 Rust 中，你始终可以能够通过derive属性来自动实现一些特性：

```
#[derive(Debug)]
struct Pet {
    name: String,
}
```

`Pet` 实现了 `Debug` 特性， 使用了相当少的代码，非常醒目。举个例子，如果没有 `derive`，你需要这样写：

```
use std::fmt;

struct Pet {
    name: String,
}

impl fmt::Debug for Pet {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            Pet { name } => {
                let mut debug_trait_builder = f.debug_struct("Pet");

                let _ = debug_trait_builder.field("name", name);

                debug_trait_builder.finish()
            }
        }
    }
}
```

哈!

但是，这仅适用于作为标准库的一部分提供的特征; 它不可定制。 但是现在，当有人想要推导出你的特质时，你可以告诉Rust要做什么。 这在serde， Diesel等流行的crate中大量使用。

获取更多信息，包括如果构建你自己的derive，查阅 The Rust Programming Language.

宏的变化

Minimum Rust Version nightly

在 Rust 2018 中，您可以通过 `use` 语句从外部包中导入特定的宏，而不是旧的 `#[macro_use]` 属性。

举个例子，考虑 `bar` 包中实现了一个 `!bar` 宏，在 `src/lib.rs` 中：

```
#[macro_export]
macro_rules! baz {
    () => ()
}
```

在你的包中，你可以这样写：

```
// Rust 2015

#[macro_use]
extern crate bar;

fn main() {
    baz!();
}
```

现在你可以这样：

```
// Rust 2018
#![feature(rust_2018_preview)]

use bar::baz;

fn main() {
    baz!();
}
```

这会使 `macro_rules` 宏更接近其他类型的项目。

程序宏

使用过程宏来派生特征时，您必须命名提供自定义派生的宏。 这通常与特征的名称相匹配，但请查阅提供派生的包的文档以确认。

举个例子，在 Serde 中，你可以这样写：

```
// Rust 2015
extern crate serde;
#[macro_use] extern crate serde_derive;

#[derive(Serialize, Deserialize)]
struct Bar;
```

现在你可以这样：

```
// Rust 2018
#![feature(rust_2018_preview)]
use serde_derive::{Serialize, Deserialize};

#[derive(Serialize, Deserialize)]
struct Bar;
```

更多细节：

这仅适用于外部包中定义的宏。 对于本地定义的宏，`#[macro_use] mod foo;` 还是需要的，如同 Rust 2015 一样。

编译器

在本章节中，主要讨论n编译器的改进，主要关注点是 改进报错信息。

改进报错信息

Minimum Rust Version 1.12

我们一直致力于改进错误，几乎每个 Rust 版本都没有什么改进，但在 Rust 1.12 中，创建了错误消息系统的重大改进。

例如，这里有一些产生错误的代码：

```
fn main() {
    let mut x = 5;

    let y = &x;

    x += 1;
}
```

这是 Rust 1.11：

```
foo.rs:6:5: 6:11 error: cannot assign to `x` because it is borrowed [E0506]
foo.rs:6     x += 1;
             ^~~~~~
foo.rs:4:14: 4:15 note: borrow of `x` occurs here
foo.rs:4     let y = &x;
                     ^
foo.rs:6:5: 6:11 help: run `rustc --explain E0506` to see a detailed explanation
```

这是 Rust 1.28：

```
error[E0506]: cannot assign to `x` because it is borrowed
 --> foo.rs:6:5
  |
4 |     let y = &x;
  |             - borrow of `x` occurs here
5 |
6 |     x += 1;
  |     ^^^^^^ assignment to borrowed `x` occurs here

error: aborting due to previous error
```

这个错误并没有太大的不同，但展示了格式的变化。它在上下文中显示您的代码，而不是仅显示行本身的文本。

增量编译

Minimum Rust Version 1.24

早在2016年9月，关于增量编译的博客。 虽然这篇文章详细介绍了，但这个想法基本上是这样的：当你在一个项目上工作时，你经常编译它，然后改变一些小的东西，然后重新编译。 从历史上看，无论您更改代码的程度如何，编译器都会编译整个项目。渐进式编译的想法是，您只需要编译实际更改的代码，这意味着第二次构建更快。

现在默认情况下已启用此功能。这意味着您的构建应该更快！在尝试获得尽可能低的构建时间时，不要忘记 crates 检查。

这通常不是编译器性能的最终故事，也不是具体的增量编译。我们计划在未来做更多的工作。

关于这种变化的一个小注意事项：它使构建更快，但使最终的二进制文件慢一点。出于这个原因，它在发布版本中没有打开。

弃用属性

Minimum Rust Version 1.9

如果您正在编写库，并且想要弃用某些内容，则可以使用 `deprecated` 属性：

```
#[deprecated(
    since = "0.2.1",
    note = "Please use the bar function instead"
)]
pub fn foo() {
    // ...
}
```

如果用户使用已弃用的功能，则会向您的用户发出警告：

```
   Compiling playground v0.0.1 (file:///playground)
warning: use of deprecated item 'foo': Please use the bar function instead
  --> src/main.rs:10:5
   |
10 |     foo();
   |     ^^^
   |
   = note: #[warn(deprecated)] on by default
```

`since` 和 `note` 都是可选的。

`since` 可以是将来的; 你可以在那放任何东西，因为那儿并没有检查。

Rustup — Rust 的版本管理器

Minimum Rust Version various (this tool has its own versioning scheme and works with all Rust versions)

Rustup 工具已成为推荐的安装 Rust 的方式，并在我们的网站上有。它的功能远不止于此，允许您管理各种版本，组件和平台。

安装 Rustup

要通过 Rustup 安装 Rust，您可以访问 https://www.rust-lang.org/install.html，它将告诉您如何在您的平台上执行此操作。 这将安装 `rustup` 本身和 `rustc` 和 `cargo` 的 `stable` 版本。

安装 Rust 的其他版本，执行 `rustup install`：

```
$ rustup install 1.30.0
```

对每夜版也是有效的：

```
$ rustup install nightly-2018-08-01
```

三种最新的版本:

```
$ rustup install stable
$ rustup install beta
$ rustup install nightly
```

升级

升级所有安装的版本，你可以执行：

```
$ rustup update
```

这将查看您已安装的所有内容，如果有新版本，将会更新。

版本管理

设置非 `stable` 的为默认版本：

```
$ rustup toolchain default nightly
```

使用一个 toolchain 而不是默认的，`rustup run`：

```
$ rustup run nightly cargo build
```

还有一个别名，这个更短一些：

```
$ cargo +nightly build
```

如果您希望每个目录具有不同的默认值，那也很容易！ 如果你在项目中运行它：

```
$ rustup override set nightly
```

然后当你在那个目录中时，`rustc` 或 `cargo` 的任何调用都将使用该工具链。 要与其他人共享，可以使用工具链的内容创建一个 `rust-toolchain` 文件，并将其检入源代码管理中。 现在，当有人克隆您的项目时，他们将获得正确的版本，而无需自己“覆盖集合”。

安装其他目标 (target)

Rust 支持交叉编译到其他平台，Rustup 可以帮助您管理它们。 例如，要使用 MUSL：

```
$ rustup target add x86_64-unknown-linux-musl
```

然后，你可以：

```
$ cargo build --target=x86_64-unknown-linux-musl
```

查看所有安装的目标：

```
$ rustup target list
```

安装组件

组件用于安装某些类型的工具。虽然大多数工具都提供了“cargo-install”，但有些工具需要深入集成到编译器中。 Rustup 确切地知道您正在使用的编译器版本，因此它只具有这些工具所需的信息。

组件是每个工具链，因此如果您希望它们可用于多个工具链，则需要多次安装它们。 在下面的示例中，添加一个 `--toolchain` 标志，设置为您要安装的工具链，例如 `nightly`。 如果没有此标志，它将安装默认工具链的组件。

要查看可以安装的完整组件列表：

```
$ rustup component list
```

接下来，让我们谈谈一些流行的组件以及何时需要安装它们。

`rust-docs`, 本地文档

安装工具链时，默认情况下会安装此第一个组件。 它包含 Rust 的文档副本，以便您可以脱机阅读。

此组件暂时无法删除; 如果感兴趣，请对 this issue 发表评论。

`rust-src` 标准库代码的拷贝

`rust-src` 组件可以为您提供 Rust 的源代码的本地副本。你为什么需要这个？好吧，像 Racer 这样的自动完成工具使用这些信息来了解你要调用的函数的更多信息。

```
$ rustup component add rust-src
```

“预览”组件

“预览”阶段有几个组件。 这些组件的名称目前都有 `-preview`，这表明它们还没有100%准备好进行一般使用。 请尝试一下并给我们反馈，但要知道他们不遵循 Rust 的稳定性保证，并且仍然在积极地改变，可能是以向后不兼容的方式。

`rustfmt-preview` 自动代码格式化

Minimum Rust Version 1.24

如果您希望自动格式化代码，可以安装此组件：

```
$ rustup component add rustfmt-preview
```

这将安装两个工具，`rustfmt` 和 `cargo-fmt`，它们将为您自动格式化代码！ 例如：

```
$ cargo fmt
```

将重新格式化您的整个 cargo 项目。

`rls-preview` 为了 IDE 集成

Minimum Rust Version 1.21

许多 IDE 功能都是基于 `langserver` 协议 构建的。要使用这些 IDE 获得对 Rust 的支持，您需要安装 Rust 语言服务器，即“RLS”：

```
$ rustup component add rls-preview
```

你的 IDE 应该从那拿到它。

`clippy-preview` 更多的 lints

要获得更多的 lints 来帮助你编写 Rust 代码，你可以安装 `clippy`：

```
$ rustup component add clippy-preview
```

This will install `cargo-clippy` for you:

```
$ cargo clippy
```

更多信息，查阅 clippy's documentation.

`llvm-tools-preview` 使用额外的LLVM工具

如果您想使用 `lld` 链接器或其他工具，如 `llvm-objdump` 或 `llvm-objcopy`，您可以安装此组件：

```
$ rustup component add llvm-tools-preview
```

这是最新的组件，因此目前没有良好的文档。

Cargo 和 crates.io

在本章中，我们将讨论关于 `cargo` 和 crates.io 的改进。更多的关注在 `cargo check`。

`cargo check` 用于快速检查

Minimum Rust Version 1.16

`cargo check` 是一个新的子命令，可以在很多情况下加快开发工作流程。

它有什么作用？让我们退一步说，讨论 `rustc` 如何编译代码。编译有许多“过程”，也就是说，编译器在从源代码到生成最终二进制文件的过程中有许多不同的步骤。 但是，您可以通过两个重要步骤来考虑这个过程：首先，`rustc` 执行所有安全检查，确保您的语法正确，所有这些。其次，一旦满足一切顺序，就会生成最终执行的实际二进制代码。

事实证明，第二步需要花费很多时间。而且大多数时候，这不是必要的。也就是说，当您处理一些 Rust 代码时，许多开发人员将进入这样的工作流程：

1. 写一些代码。
2. 运行 `cargo build` 以确保它编译。
3. 根据需要重复1-2。
4. 运行 `cargo test` 以确保测试通过。
5. 亲自尝试二进制文件
6. GOTO 1。

在第二步中，您实际上从未运行过您的代码。您正在寻找编译器的反馈，而不是实际运行二进制文件。 `cargo check` 正好支持这个用例：它运行所有编译器的检查，但不生成最终的二进制文件。要使用它：

```
$ cargo check
```

在那里你通常可以 `cargo build`。 工作流现在看起来像：

1. 写一些代码。
2. 运行 `cargo check` 以确保它编译。
3. 根据需要重复1-2。
4. 运行 `cargo test` 以确保测试通过。
5. 运行 `cargo build` 来构建二进制文件并自己尝试
6. GOTO 1。

那么你实际获得了多少加速？与大多数相关的问题一样，答案是“它取决于”。在撰写本文时，这里有一些非科学的基准。

build	performance	check performance	speedup
initial compile	11s	5.6s	1.96x

build	performance	check performance	speedup
second compile (no changes)	3s	1.9s	1.57x
third compile with small change	5.8s	3s	1.93x

`cargo install` 用于自动安装

Minimum Rust Version 1.5

Cargo 已经发展了一种新的 `install` 命令。 这旨在用于为 Cargo 安装新的子命令，或者为 Rust 开发人员安装工具。 这并不能取代为您支持的平台上的最终用户构建真实的本机程序包的需要。

例如，本指南是使用 `mdbook`。 你可以将它安装在你的系统上:

```
$ cargo install mdbook
```

然后使用:

```
$ mdbook --help
```

作为扩展 Cargo 的示例，你可以使用 `cargo-update` 包。 要安装它:

```
$ cargo install cargo-update
```

这将允许你使用此命令，该命令检查你 `cargo install` 的所有内容并将其更新为最新版本:

```
$ cargo install-update -a
```

`cargo new` 创建一个默认可执行项目

Minimum Rust Version 1.25

`cargo new` 现在默认生成二进制文件，而不是库。我们试图保持 Cargo 的 CLI 非常稳定，但这种变化很重要，不太可能导致破损。

对于某些背景，cargo new 接受两个标志： `--lib` 用于创建库， `--bin` 用于创建二进制文件或可执行文件。 如果你没有传递其中一个标志，它曾经默认为 `--lib` 。 当时，我们做出了这个决定，因为每个二进制文件（通常）都依赖于许多库，因此我们认为库案例会更常见。但是，这是不正确的; 每个包都依赖于许多二进制文件。 此外，在开始使用时，你经常需要的是一个可以运行和使用的程序。而且，不仅仅是新 Rustaceans们， 甚至是很长时间的社区成员都说他们发现这个默认值令人惊讶。因此，我们已经改变它，它现在默认为 `--bin` 。

`cargo rustc` 用于传递标记至 rustc

Minimum Rust Version 1.1

`cargo rustc` 是Cargo的一个新的子命令，它允许你通过 cargo 传递任意标记到 `rustc`。

例如，Cargo 没有办法传递内置的不稳定标志。但是如果我们想使用 `print-type-sizes` 来查看我们的类型有哪些布局信息。我们可以运行这个：

```
$ cargo rustc -- -Z print-type-sizes
```

我们将得到一堆描述我们类型大小的输出。

注意

`cargo rustc` 只会将这些标记传递给你的 crate 的调用，而不是用于构建依赖项的任何 `rustc` 调用。 如果你想这样做，请参阅 `$RUSTFLAGS`。

Cargo workspaces 用于有多个子包的项目

Minimum Rust Version 1.12

Cargo 曾经有两个组织层次：

- 一个 *package* 有一个或多个 crates
- 一个 *crate* 有一个或多个 modules

Cargo 现在有一个额外的层次：

- 一个 *workspace* 包含一个或多个 packages

这对于大型项目非常有用。例如，the `futures` package 是一个 *workspace*，包含许多相关的包：

- futures
- futures-util
- futures-io
- futures-channel

还有其他。

Workspaces 允许单独开发这些包，但它们共享一组依赖项，因此只有单个 target 目录和单个 `Cargo.lock`。

更多有关 workspaces, 请查阅 the Cargo documentation.

多个演示用例

Minimum Rust Version 1.22

Cargo有一个 `examples` 功能，用于向人们展示如何使用您的包裹。通过将单个文件放在顶级 `examples` 目录中，您可以创建多个示例。

但是如果你的例子对于单个文件来说太大了怎么办？Cargo支持在 `examples` 中添加子目录，并在其中查找 `main.rs` 来构建示例。它看起来像这样：

```
my-package
 └──src
     └── lib.rs // code here
 └──examples
     └── simple-example.rs // a single-file example
     └── complex-example
        └── helper.rs
        └── main.rs // a more complex example that also uses `helper` as a
submodule
```

patch 替换依赖

Minimum Rust Version 1.21

当你想要覆盖依赖图的某些部分时，可以使用你的 `Cargo.toml` 的 `[patch]` 部分。

> cargo 有一个类似的 `[replace]` 功能; 虽然我们不打算弃用或删除 `[replace]`，但在任何情况下都应该更喜欢 `[patch]`。

那么它看起来像什么？ 假设我们有一个看起来像这样的 Cargo.toml：

```
[dependencies]
foo = "1.2.3"
```

另外，我们的 `foo` 包依赖于 `bar` 包，我们在 `bar` 中发现了一个错误。为了测试这个，我们下载了 `bar` 的源代码，然后更新我们的 `Cargo.toml`：

```
[dependencies]
foo = "1.2.3"

[patch.crates-io]
bar = { path = '/path/to/bar' }
```

现在，当你 `cargo build` 时，它将使用本地版本的 `bar`，而不是来自 crates.io 的 `foo` 所依赖的版本。然后，您可以尝试更改，并修复该错误！

更多细节，查阅 the documentation for `patch`.

Cargo 更改源

Minimum Rust Version 1.12

Cargo 在“源”中找到它的包。默认源是 crates.io。但是，您可以在 `.cargo/config` 中选择其他源：

```
[source.crates-io]
replace-with = 'my-awesome-registry'

[source.my-awesome-registry]
registry = 'https://github.com/my-awesome/registry-index'
```

这种配置意味着不是使用 crates.io，而是 Cargo 将查询 `my-awesome-registry` 源（在此处配置为不同的索引）。此备用源 *必须与 crates.io 索引完全相同*。 Cargo 假设替换源在这方面是精确的 1：1 镜像，并且围绕该假设设计了以下支持。

使用替换源为 crate 生成锁定文件时，原始源将编码到锁定文件中。例如，在上面的配置中，所有锁定文件仍然会提到 crates.io 作为包源的信息。 这在语义上代表了 crates.io 如何成为所有 crates 的真实来源，并且由于所有替换都具有 1：1 的对应性，因此这是坚持的。

总的来说，这意味着无论你使用何种替换源，您都可以将锁文件发送给其他任何人，并且你仍然可以获得可验证的可重现的构建！

这个工具有 `cargo-vendor` 和 `cargo-local-registry`，这对于“离线构建”通常很有用。他们提前准备所有 Rust 依赖项的列表，这使您可以轻松地将它们发送到构建计算机。

Crates.io 不允许使用通配符

Minimum Rust Version 1.6

Crates.io 不允许您上传具有通配符依赖关系的包。 换句话说，这些：

```
[dependencies]
regex = "*"
```

通配符依赖性意味着您可以使用任何可能的依赖项版本。 这极不可能是真实的，并且会在生态系统中造成不必要的破坏。

相反，取决于版本范围。例如，`^`是默认值，因此您可以使用：

```
[dependencies]
regex = "1.0.0"
```

相应的，`>`，`<=`，和所有其他的非`*`范围都是可以的。

文档

在这章节中，我们将要讨论文档的改进，关注点在 second edition of "the book"。

新版本的 "the book"

Minimum Rust Version 1.18 第二版的草稿

Minimum Rust Version 1.26 第二版的最后一版

Minimum Rust Version 1.28 2018版的草稿

我们分发了一份 “Rust编程语言”，亲切地昵称为 “书”，以及 Rust 自 Rust 1.0 以来的每个版本。

但是，因为它是在 Rust 1.0 之前编写的，所以它开始显示它的年龄。本书的许多部分都含糊不清，因为它是在真正的细节被确定为1.0版本之前编写的。它在教学生涯中没有做出出色的工作。

从 Rust 1.18 开始，我们发布了本书第二版的草稿。 最终版本随 Rust 1.26 一起提供。新版本是从头开始的完整重写，使用我们从 Rust 中获得的最近两年的知识。 你会发现很多 Rust 的核心概念，要构建的新项目以及各种其他好东西的全新解释。请[查看]check it out 并告诉我们你的想法！

您也可以从Starch Press购买。现在印刷版已发货，第二版已冻结。

这些名字有点令人困惑，因为这本书的“第二版”是该书的第一版。因此，我们认为本书的新版本将与 Rust 本身的新版本相对应，因此从 1.28 开始，我们一直在发送下一版本的草案，2018年版。它仍然非常接近第二版，但包含有关新书功能的信息，因为该书的内容已被冻结。我们将继续更新此版本，直到我们决定在纸上打印第二版。

The Rust Bookshelf

Minimum Rust Version various, each book is different. 随着Rust的文档不断发展，我们获得的远远超过了“The book”和参考资料。 我们现在收集了各种长篇文档，绰号“Rust书架”。 在不同的时间添加不同的资源，并且随着更多的写入，我们将添加新的资源。

The Cargo book

Minimum Rust Version 1.21 从历史上看，Cargo 的文档是在 http://doc.crates.io 上托管的，它不遵循发布模型，即使 Cargo 本身也是如此。 这导致了一个功能会在cargo nightly中出现，文档会更新，然后长达12周，用户会认为它应该可以工作，但它还没有。 https://doc.rust-lang.org/cargo 是 Cargo 文档的新家，http://doc.crates.io 现在重定向到那里。

The `rustdoc` book

Minimum Rust Version 1.21

Rustdoc, 我们文档工具, 现在位于 https://doc.rust-lang.org/rustdoc.

Rust By Example

Minimum Rust Version 1.25

Rust by Example 过去住在 https://rustbyexample.com，但现在是 Bookshelf 的一部分！ 它可以在 https://doc.rust-lang.org/rust-by-example/ 找到。 RBE 允许您通过简短的代码示例和练习来学习 Rust，而不是本书的冗长散文。

The Rustonomicon

我们现在有了一个草稿书, The Rustonomicon: the Dark Arts of Advanced and Unsafe Rust Programming.

从标题来看，我相信你可以猜到：这本书讨论了一些高级主题，包括“不安全”。 对于那些在 Rust 最低级别工作的人来说，这是必读的。

`std::os` 所有平台的文档

Minimum Rust Version 1.21

`std::os` 模块包含特定于操作系统的功能。您现在将看到的不仅仅是 linux，而是所有我们构建文档的平台。

我们很遗憾该文档的版本目前暂定在 Linux; 这是第一步。这是标准库特有的，不适用于一般用途; 我们希望将来能够进一步改进这一点。

`rustdoc`

在本章节中，我们将讨论 `rustdoc` 的改进，特别关注 文档测试现在可以编译失败。

文档测试现在可以编译失败

Minimum Rust Version 1.22

现在可以创建 `compile-fail` 测试在 Rustdoc 中，如下:

```
/// ```compile_fail
/// let x = 5;
/// x += 2; // shouldn't compile!
/// ```
# fn foo() {}
```

请注意，这些类型的测试可能比其他测试更脆弱，因为 Rust 的添加可能导致代码在以前不会在编译时编译。 考虑使用 `?` 的第一个版本，例如：使用 `?` 的代码将无法在 Rust 1.21 上编译，但在 Rust 1.22 上成功编译，导致您的测试套件开始失败。

Rustdoc uses CommonMark

for support by default

for support via a flag

Rustdoc 允许您在文档注释中，使用 Markdown 编写。 在 Rust 1.0 中，我们使用了 `hoedown` markdown实现，用 C 编写。 Markdown 更像是一个想法的实现系列，因此 `hoedown` 有自己的方言，就像许多解析器一样。 CommonMark project 试图定义更严格的Markdown版本，现在，Rustdoc默认使用它。

从 Rust 1.23 开始，我们仍然默认为 `hoedown`，但你可以通过标志 `--enable-commonmark` 启用 Commonmark。 今天，我们只支持 CommonMark。

平台支持和目标（target）支持

在本章中，我们讨论平台和目标的改进，主要关注点是 that the `libcore` library now works on stable Rust。

libcore 给低层 Rust 使用

Minimum Rust Version 1.6

Rust 的标准库是双层的： 有一个小的核心库， `libcore` ，以及构建在它之上的完整标准库 `libstd` 。 `libcore` 完全与平台无关，只需要定义少量外部符号。Rust 的 `libstd` 建立在 `libcore` 之上，增加了对内存分配和I/O等内容的支持。 在嵌入式空间中使用 Rust 的应用程序以及编写操作系统的应用程序通常只使用 `libcore` 来避免 `libstd` 。

另外需要注意的是，虽然今天支持使用 `libcore` 构建 *库*，但构建完整的应用程序还不稳定。

要使用 `libcore` ，请将此标志添加到你的 crate 根目录：

```
#![no_std]
```

这将删除标准库，并将 `core` crate 带入您的命名空间以供使用：

```
#![no_std]

use core::cell::Cell;
```

你可以找到 `libcore` 的文档 here。

WebAssembly support

Minimum Rust Version 1.14 for `emscripten`

Minimum Rust Version nightly for `wasm32-unknown-unknown`

Rust 已经有了对 WebAssembly 的支持，这意味着你可以在浏览器客户端中运行 Rust 代码。

在 Rust 1.14 中，我们通过 emscripten 获得了支持。 安装它后，您可以编写 Rust 代码并生成 asm.js（the precusor to wasm）或 WebAssembly。

以下是使用此支持的示例：

```
$ rustup target add wasm32-unknown-emscripten
$ echo 'fn main() { println!("Hello, Emscripten!"); }' > hello.rs
$ rustc --target=wasm32-unknown-emscripten hello.rs
$ node hello.js
```

然而，与此同时，Rust 也增加了自己的支持，独立于 Emscripten。 这被称为 “未知目标”，因为它不是 `wasm32-unknown-emscripten`，而是 `wasm32-unknown-unknown`。 这将是它准备好后首选使用的目标，但就目前而言，它实际上只能在 nightly版 中得到很好的支持。

全局分配符

Minimum Rust Version 1.28

分配器是 Rust 中的程序在运行时从系统获取内存的方式。以前，Rust 不允许改变获取内存的方式，这阻止了一些用例。 在某些平台上，这意味着在其他平台上使用 jemalloc，系统分配器，但用户无法控制此关键组件。 在1.28中，`#[global_allocator]` 属性现在是稳定的，它允许 Rust 程序将它们的分配器设置为系统分配器，并通过实现 `GlobalAlloc` 特性来定义新的分配器。

某些平台上 Rust 程序的默认分配器是 jemalloc。标准库现在提供了系统分配器的句柄，可以在需要时通过声明静态并使用 `#[global_allocator]` 属性标记它来切换到系统分配器。

```
use std::alloc::System;

#[global_allocator]
static GLOBAL: System = System;

fn main() {
    let mut v = Vec::new();
    // This will allocate memory using the system allocator.
    v.push(1);
}
```

但是，有时您希望为给定的应用程序域定义自定义分配器。通过实现 `GlobalAlloc` 特性，这也相对容易。 您可以在 the documentation。

MSVC toolchain 支持

Minimum Rust Version 1.2

在 Rust 1.0 的发布中，我们只支持 Windows 上的 GNU 工具链。 随着 Rust 1.2 的发布，我们引入了对 MSVC 工具链的初始支持。 之后，随着支持的成熟，我们最终将其作为 Windows 用户的默认选择。

与 C 交互的两个问题之间的区别。如果您使用的是使用一个工具链或另一个工具链构建的库，则需要将其与相应的 Rust 工具链相匹配。 如果您不确定，请使用 MSVC; 这是有充分理由的默认值。

要使用此功能，只需在 Windows 上使用 Rust，安装程序将默认使用它。 如果您更愿意切换到 GNU 工具链，可以使用 Rustup 进行安装：

```
$ rustup toolchain install stable-x86_64-pc-windows-gnu
```

MUSL 支持完全静态二进制文件

Minimum Rust Version 1.1

默认情况下，Rust 将静态链接所有 Rust 代码。但是，如果使用标准库，它将动态链接到系统的 `libc` 实现。

如果您想要100%静态二进制文件，可以在 Linux 上使用 `MUSL libc`。

安装MUSL支持

要添加对MUSL的支持，您需要选择正确的目标。 这个页面 有完整目标支持列表，其中一些是 `musl`。

如果你不确定你想要什么，对于64位 Linux，它可能是 `x86_64-unknown-linux-musl`。 我们将在本指南中使用此目标，但其他目标的说明保持不变，只需在我们提及目标的位置更改名称。

要获得对此目标的支持，请使用 `rustup`：

```
$ rustup target add x86_64-unknown-linux-musl
```

这将安装对默认工具链的支持; 要安装其他工具链，请添加 `--toolchain` 标志。例如：

```
$ rustup target add x86_64-unknown-linux-musl --toolchain=nightly
```

使用MUSL构建

要使用这个新目标，请将 `--target` 标志传递给 Cargo：

```
$ cargo build --target x86_64-unknown-linux-musl
```

生成的二进制文件现在将使用 MUSL 构建！

cdylib 与 C 交互

Minimum Rust Version 1.10 for `rustc`

Minimum Rust Version 1.11 for `cargo`

如果你正在生成一个打算从 C（或其他语言通过 C FFI）使用的库，则 Rust 不需要在最终目标代码中包含特定于 Rust 的内容。 对于像这样的库，你需要在你的 `Cargo.toml` 中使用 `cdylib` crate 类型：

```
[lib]
crate-type = ["cdylib"]
```

这将生成一个较小的二进制文件，其中没有特定于 Rust 的信息。

不稳定的特性状态

Language

Feature	Status	Minimum Edition
`impl Trait`	Shipped, 1.26	2015
Basic slice patterns	Shipped, 1.26	2015
Default match bindings	Shipped, 1.26	2015
Anonymous lifetimes	Shipped, 1.26	2015
`dyn Trait`	Shipped, 1.27	2015
SIMD support	Shipped, 1.27	2015
`?` in `main`/tests	Shipping, 1.26 and 1.28	2015
In-band lifetimes	Unstable; tracking issue	2015
Lifetime elision in `impl`s	Unstable; tracking issue	2015
Non-lexical lifetimes	Implemented but not ready for preview	2015
`T: 'a` inference in `struct`s	Unstable; tracking issue	2015
Raw identifiers	Unstable; tracking issue	?
Import macros via `use`	Unstable; tracking issue	?
Module system path changes	Unstable; tracking issue	2018

虽然 Rust 2015 中已经提供了其中一些功能，但是它们在这里被跟踪，因为它们是作为 Rust 2018 版本的一部分进行推广的。 因此，它们将在本指南的后续章节中讨论。标记为 “Shipped” 的功能现在都可以在稳定的 Rust 中使用，因此您可以立即开始使用它们！

Standard library

Feature	Status
Custom global allocators	Will ship in 1.28

Tooling

Tool	Status
RLS 1.0	Feature-complete; see 1.0 milestone
rustfmt 1.0	Finalizing spec; 1.0 milestone, style guide RFC, stability RFC
Clippy 1.0	RFC

Documentation

Tool	Status
Edition Guide	Initial draft complete
TRPL	Updated as features stabilize

Web site

视觉设计正在最终确定，早期的内容头脑风暴已经完成。