Rust高性能编程指南

第一版发布于2020年11月

作者：Nicholas Nethercote及其他贡献者

Source code

中文版 翻译 by 鸟窝

下载本书的PDF版本：中文版

介绍

对于许多 Rust 程序来说，性能至关重要。

本书介绍了一些技术，可以改善 Rust 程序的性能相关特性，如运行时速度、内存使用和二进制大小。编译时间 部分还包括一些可以缩短 Rust 程序编译时间的技术。有些技术只需要修改构建配置，但很多需要改动代码。

部分技术是 Rust 特有的，而其他一些则包含了可以（经过修改）应用到其他编程语言程序中的思想。一般提示 部分也包括了适用于任何编程语言的一般原则。尽管如此，本书主要关注 Rust 程序的性能，不能取代一本通用的性能分析与优化指南。

本书还专注于实用且经过验证的技术：许多技术都有链接到拉取请求或其他资源，展示了这些技术如何在真实的 Rust 程序中使用。它反映了主要作者的背景，偏向于编译器开发，而不太涉及其他领域，比如科学计算。

本书的内容故意写的很简练，本书更注重广度而非深度，使得阅读更加快捷。在适当的情况下，会提供链接到提供更深入内容的外部资源。

本书的目标读者是中级和高级 Rust 用户。初学者 Rust 用户有很多需要学习的内容，这些技术可能会给他们带来不必要的困扰。

基准测试

基准测试通常涉及比较两个或多个执行相同任务的程序的性能。有时可能涉及比较两个或多个不同的程序，比如 Firefox vs Safari vs Chrome。有时则涉及比较同一个程序的不同版本。后一种情况可以让我们可靠地回答问题：“这个变动是否加快了速度？”

基准测试是一个复杂的主题，全面覆盖超出了本书的范围，但以下是基础知识。

首先，您需要工作负载来进行测量。理想情况下，您将拥有一系列能够代表程序实际使用情况的工作负载。使用真实世界的输入的工作负载最佳，但微基准测试和压力测试在适度使用时也可以有所帮助。

其次，您需要一种运行工作负载的方法，这也将决定所使用的度量标准。

- Rust 内置的基准测试是一个简单的起点，但它们使用不稳定的功能，因此只能在夜间版 Rust 上使用。
- Criterion和Divan是更复杂的替代方案。
- Hyperfine是一个出色的通用基准测试工具。
- 也可以使用自定义基准测试工具。例如，rustc-perf是用于对 Rust 编译器进行基准测试的工具。

在选择度量标准时，有许多选择，而正确的选择取决于正在进行基准测试的程序的性质。例如，对于批处理程序来说有意义的度量标准可能对于交互式程序来说没有意义。在许多情况下，墙上时间(wall-time)是一个显而易见的选择，因为它对应于用户的感知。然而，它可能受到很高的方差影响。特别是，内存布局的微小变化可能导致显著但短暂的性能波动。因此，其他方差较低的度量标准（如周期或指令计数）可能是一个合理的选择。

总结多个工作负载的测量结果也是一个挑战，有许多方法可以做到这一点，并没有单一的方法显然是最佳的。

良好的基准测试很难。话虽如此，在开始优化程序时，不要过分强调拥有完美的基准测试设置。一般的基准测试要比没有基准测试好得多。对您正在测量的内容保持开放的心态，随着时间的推移，您可以根据您对程序性能特征的了解进行基准测试改进。

构建配置

你可以通过改变 Rust 程序的构建配置，而不改变其代码，极大地改变其性能。每个 Rust 程序都有许多可能的构建配置。所选择的配置将影响编译代码的多个特性，如编译时间、运行时速度、内存使用、二进制大小、调试性、可分析性，以及编译后的程序将在哪些架构上运行。

大多数配置选择会在改善一个或多个特性的同时，恶化一个或多个其他特性。例如，一个常见的权衡是为了获得更高的运行时速度而接受更差的编译时间。对于你的程序来说，正确的选择取决于你的需求和你的程序的具体情况，并且与性能相关的选择（这是大多数选择）应该通过基准测试来验证。

请注意，Cargo 只查看工作空间根目录中 `Cargo.toml` 文件中的配置。在依赖项中定义的配置将被忽略。因此，这些选项大多数情况下只与二进制crate相关，而不是库crate。

发布构建

最重要的构建配置选择是简单但易于忽略的：确保在需要高性能时使用发布构建，而不是开发构建。通常可以通过向 Cargo 指定 `--release` 标志来完成这一点。

开发构建是默认选项。它们适用于调试，但没有经过优化。如果你运行 `cargo build` 或 `cargo run`，则会生成它们。（另外，运行 `rustc` 时不带额外选项也会生成一个未经优化的构建。）

考虑一下从 `cargo build` 运行的输出的最后一行。

```
Finished dev [unoptimized + debuginfo] target(s) in 29.80s
```

这个输出表明已经生成了一个开发构建。编译后的代码将被放置在 `target/debug/` 目录中。 `cargo run` 将运行开发构建。

相比之下，发布构建要更加优化，省略了调试断言和整数溢出检查，并且省略了调试信息。相对于开发构建，10-100 倍的速度提升是常见的！如果你运行 `cargo build`

`--release` 或 `cargo run --release`，就会生成它们。（另外，`rustc` 有多个选项用于优化构建，比如 `-O` 和 `-C opt-level`。）这通常会比开发构建需要更长的时间，因为需要额外的优化。

考虑一下从 `cargo build --release` 运行的输出的最后一行。

```
Finished release [optimized] target(s) in 1m 01s
```

这个输出表明已经生成了一个发布构建。编译后的代码将被放置在 `target/release/` 目录中。`cargo run --release` 将运行发布构建。

请查看 Cargo 配置文件 中的文档以了解更多关于开发构建（使用 `dev` 配置）和发布构建（使用 `release` 配置）之间的区别。

在发布构建中使用的默认构建配置选择提供了编译时间、运行时速度和二进制大小等特性之间的良好平衡。但是有许多可能的调整，如下面的部分所述。

最大化运行时速度

以下构建配置选项主要设计用于最大化运行时速度。其中一些可能也会减小二进制大小。

代码生成单元

Rust 编译器将 crates 分成多个 代码生成单元 来并行化（从而加快）编译。然而，这可能导致它错过一些潜在的优化。通过将单元数设置为一个，你可能能够提高运行时速度并减小二进制大小，但代价是增加编译时间。将以下行添加到 `Cargo.toml` 文件中：

```
[profile.release]
codegen-units = 1
```

示例 1, **示例 2**。

链接时优化

链接时优化（LTO）是一种整个程序的优化技术，可以通过 10-20% 或更多来提高运行时速度，并减小二进制大小，但代价是更差的编译时间。它有几种形式。

LTO 的第一种形式是 *thin local LTO*，是一种轻量级的 LTO。默认情况下，编译器会对任何涉及非零级别优化的构建使用它。这包括发布构建。要显式请求此级别的 LTO，请将以下行放入 `Cargo.toml` 文件中：

```
[profile.release]
lto = false
```

LTO 的第二种形式是 *thin LTO*，这是一种稍微激进一些的形式，很可能会提高运行时速度和减小二进制大小，同时还会增加编译时间。在 `Cargo.toml` 中使用 `lto = "thin"` 来启用它。

LTO 的第三种形式是 *fat LTO*，这是一种更加激进的形式，可能会进一步提高性能并减小二进制大小，但再次增加构建时间。在 `Cargo.toml` 中使用 `lto = "fat"` 来启用它。

最后，完全禁用 LTO 也是可能的，这可能会使运行时速度变慢，并增加二进制大小，但会减少编译时间。在 `Cargo.toml` 中使用 `lto = "off"` 来实现。请注意，这与 `lto = false` 选项不同，如上所述，后者保留了 thin local LTO。

替换分配器

可以用另一个分配器替换 Rust 程序使用的默认（系统）堆分配器。确切的影响将取决于个别程序和选择的替换分配器，但在实践中已经看到了运行时速度的大幅提升和内存使用的大幅减少。该效果在不同平台上也会有所不同，因为每个平台的系统分配器都有其优点和缺点。使用替换分配器还可能增加二进制大小和编译时间。

jemalloc

Linux 和 Mac 上一个流行的替换分配器是 jemalloc，可以通过 `tikv-jemallocator` crate 使用。要使用它，请将依赖项添加到你的 `Cargo.toml` 文件中：

```
[dependencies]
tikv-jemallocator = "0.5"
```

然后将以下内容添加到你的 Rust 代码中，例如在 `src/main.rs` 的顶部：

```
#[global_allocator]
static GLOBAL: tikv_jemallocator::Jemalloc =
tikv_jemallocator::Jemalloc;
```

此外，在 Linux 上，jemalloc 可以配置为使用 [transparent huge pages][THP]（THP）。这可以进一步加速程序，可能以更高的内存使用为代价。

在构建程序之前，通过在 `MALLOC_CONF` 环境变量中适当设置来完成这个目标，例如：

```
MALLOC_CONF="thp:always,metadata_thp:always" cargo build --release
```

运行编译后的程序的系统也必须配置为支持 THP。更多详情请参阅 此博客文章。

mimalloc

另一个适用于许多平台的替换分配器是 mimalloc，可以通过 `mimalloc` crate使用。要使用它，请将依赖项添加到你的 `Cargo.toml` 文件中：

```
[dependencies]
mimalloc = "0.1"
```

然后将以下内容添加到你的 Rust 代码中，例如在 `src/main.rs` 的顶部：

```
#[global_allocator]
static GLOBAL: mimalloc::MiMalloc = mimalloc::MiMalloc;
```

CPU 特定指令

如果你不在乎二进制在旧的（或其他类型的）处理器上的兼容性，你可以告诉编译器生成特定于某个 [特定的 CPU 架构]（如 x86-64 CPU 的 AVX SIMD 指令）的最新

（可能也是最快的）指令。

要从命令行请求这些指令，请使用 `-C target-cpu=native` 标志。例如：

```
RUSTFLAGS="-C target-cpu=native" cargo build --release
```

或者，要从 `config.toml` 文件（针对一个或多个项目）中请求这些指令，添加以下行：

```
[build]
rustflags = ["-C", "target-cpu=native"]
```

这可以提高运行时速度，特别是如果编译器在你的代码中发现了向量化机会。

如果你不确定 `-C target-cpu=native` 是否正常工作，请比较 `rustc --print cfg` 和 `rustc --print cfg -C target-cpu=native` 的输出，看看 CPU 特性是否在后一种情况下被正确检测到。如果没有，你可以使用 `-C target-feature` 来定位特定的特性。

基于配置文件的优化

配置文件是 Rust 的 nightly 版本中的一个实验性功能，可以为你的项目启用或禁用一些实验性功能，从而影响编译时间和生成的代码。要了解更多，请查看 RFC 2994。

自定义配置文件

除了 `dev` 和 `release` 配置文件外，Cargo 还支持 自定义配置文件。例如，如果你发现开发构建的运行时速度不够，并且发布构建的编译时间对于日常开发来说太慢，那么创建一个介于 `dev` 和 `release` 之间的自定义配置文件可能会有用。

总结

在构建配置方面有很多选择。以下几点总结了以上信息并给出了一些建议。

- 如果要最大化运行时速度，请考虑以下所有内容：`codegen-units = 1`、`lto = "fat"`、替换分配器和 `panic = "abort"`。
- 如果要最小化二进制大小，请考虑 `opt-level = "z"`、`codegen-units = 1`、`lto = "fat"`、`panic = "abort"` 和 `strip = "symbols"`。
- 无论哪种情况，如果不需要广泛的架构支持，请考虑 `-C target-cpu=native`，并且如果它符合你的发布机制，还有 `cargo-pgo`。
- 如果你的平台支持的话，一定要使用更快的链接器，因为这样做没有任何坏处。
- 对所有更改进行基准测试，一个一个地进行，以确保它们有预期的效果。

最后，这个问题跟踪了 Rust 编译器自身的构建配置的演变。Rust 编译器的构建系统比大多数 Rust 程序的构建系统更奇怪、更复杂。尽管如此，这个问题可能有助于说明构建配置选择如何应用于一个大型程序。

代码检查

Clippy 是一个用于捕捉 Rust 代码中常见错误的一组检查项。它是一个在 Rust 代码中运行的优秀工具。它还可以帮助提高性能，因为其中一些检查项涉及可能导致次优性能的代码模式。

鉴于自动检测问题优于手动检测，本书的其余部分将不提及 Clippy 默认检测到的性能问题。

基础

安装完成后，运行起来很简单：

```
cargo clippy
```

可以通过访问 检查项列表 并取消选中所有检查项组，除了 “Perf”，来查看所有性能检查项的完整列表。

除了使代码更快外，性能检查项建议通常会导致更简洁、更符合习惯的代码，因此即使是不经常执行的代码，也值得遵循。

相反，一些非性能检查项建议可以改善性能。例如， `ptr_arg` 风格检查项建议将各种容器参数更改为切片，例如将 `&mut Vec<T>` 参数更改为 `&mut [T]` 。这里的主要动机是切片提供了更灵活的 API，但也可能由于减少间接性和为编译器提供更好的优化机会而导致更快的代码。 **示例**。

禁止使用特定类型

在接下来的章节中，我们将看到有时值得避免使用某些标准库类型，而选择更快的替代方案。如果决定使用这些替代方案，则很容易出错地在某些地方意外使用标准库类型。

您可以使用 Clippy 的 `disallowed_types` 检查项来避免这个问题。例如，为了禁止使用标准哈希表（原因在 哈希 部分有解释），请向您的代码添加一个 `clippy.toml` 文件，并包含以下行。

```
disallowed-types = ["std::collections::HashMap",
"std::collections::HashSet"]
```

性能分析

在优化程序时，你也需要一种方法来确定程序的哪些部分是“热点”（被频繁执行以影响运行时）并且值得修改。这最好通过性能分析来完成。

性能分析工具

有许多不同的性能分析工具可用，每种都有其优点和缺点。以下是已成功用于 Rust 程序的性能分析工具的不完全列表。

- perf 是一种使用硬件性能计数器的通用性能分析工具。Hotspot 和 Firefox Profiler 适用于查看 perf 记录的数据。它在 Linux 上运行。
- Instruments 是一种通用性能分析工具，随 Xcode 一起提供在 macOS 上。
- Intel VTune Profiler 是一种通用性能分析工具。它在 Windows、Linux 和 macOS 上运行。
- AMD μProf 是一种通用性能分析工具。它在 Windows 和 Linux 上运行。
- samply 是一种采样分析器，可生成可以在 Firefox Profiler 中查看的分析文件。它在 Mac 和 Linux 上运行。
- flamegraph 是一个 Cargo 命令，使用 perf/DTrace 来分析您的代码，然后在火焰图中显示结果。它在 Linux 上运行，以及支持 DTrace 的所有平台（macOS、FreeBSD、NetBSD 和可能的 Windows）。
- Cachegrind 和 Callgrind 提供全局、每个函数和每个源代码行的指令计数以及模拟缓存和分支预测数据。它们在 Linux 和其他一些 Unix 上运行。
- DHAT 适用于查找代码中哪些部分导致了大量分配，并提供有关峰值内存使用情况的见解。它还可以用于识别对 `memcpy` 的热点调用。它在 Linux 和其他一些 Unix 上运行。dhat-rs 是一个实验性的替代方案，功能稍弱，需要对您的 Rust 程序进行轻微更改，但在所有平台上都运行。
- heaptrack 和 bytehound 是堆分析工具。它们在 Linux 上运行。
- `counts` 支持即时分析，将 `eprintln!` 语句与基于频率的后处理结合使用，非常适合获取对代码部分的领域特定见解。它在所有平台上运行。
- Coz 执行*因果分析*以测量优化潜力，并通过 coz-rs 支持 Rust。它在 Linux 上运行。

调试信息

为了有效地对发布版本进行性能分析，您可能需要启用源代码行调试信息。要做到这一点，请将以下行添加到您的 `Cargo.toml` 文件中：

```
[profile.release]
debug = 1
```

有关 `debug` 设置的更多详细信息，请参阅 Cargo 文档。

不幸的是，即使执行了上述步骤，您也不会得到有关标准库代码的详细性能分析信息。这是因为 Rust 标准库的发布版本未使用调试信息构建。

绕过这个问题最可靠的方法是构建您自己版本的编译器和标准库，按照 这些说明 进行操作，并将以下行添加到 `config.toml` 文件中：

```
[rust]
debuginfo-level = 1
```

虽然这很麻烦，但在某些情况下可能值得一试。

另外，不稳定的 build-std 功能允许您将标准库作为程序的正常编译的一部分进行编译，并使用相同的构建配置。然而，标准库调试信息中的文件名不会指向源代码文件，因为此功能不会下载标准库源代码。因此，这种方法不会帮助像 Cachegrind 和 Samply 这样需要源代码才能完全运行的性能分析工具。

符号解析

Rust 使用一种名称编码形式来在编译代码中编码函数名称。如果性能分析工具不了解这一点，其输出可能包含以 `_ZN` 或 `_R` 开头的符号名称，例如 `_ZN3foo3barE` 或 `_ZN28_$u7b$$u7b$closure$u7d$$u7d$E` 或 `_RMCsno73SFvQKx_1cINtB0_3StrKRe616263_E`

类似这样的名称可以使用 [`rustfilt`] 手动解析。

如果在性能分析时遇到符号解析的问题，将默认的旧版格式更改为新版 v0 格式可能值得一试。

要从命令行使用 v0 格式，可以使用 `-C symbol-mangling-version=v0` 标志。例如：

```
RUSTFLAGS="-C symbol-mangling-version=v0" cargo build --release
```

或者，要求在 `config.toml` 文件中（对于一个或多个项目）添加以下指令：

```
[build]
rustflags = ["-C", "symbol-mangling-version=v0"]
```

内联

对于热点、未内联的函数的进入和退出往往占据了执行时间的相当大的一部分。将这些函数内联可以带来一些小但容易的速度提升。

在 Rust 函数中有四种可以使用的内联属性。

- **None**。编译器会自行决定是否应该将函数内联。这将取决于优化级别和函数的大小等因素。非泛型函数永远不会跨 crate 边界内联，除非使用了链接时优化；而泛型函数可能会。
- `#[inline]` 。这表示该函数应该内联，包括跨 crate 边界。
- `#[inline(always)]` 。这强烈建议该函数应该内联，包括跨 crate 边界。
- `#[inline(never)]` 。这强烈建议该函数不应该内联。

内联属性并不保证函数是否内联或不内联，但实际上 `#[inline(always)]` 会导致内联，除非在极端情况下。

内联不是传递性的。如果函数 `f` 调用函数 `g`，并且您希望这两个函数在调用 `f` 的调用点一起内联，那么这两个函数都应该标记为内联属性。

简单情况

最适合内联的候选函数是：(a) 很小的函数，或者 (b) 只有一个调用点的函数。即使没有内联属性，编译器通常也会自动内联这些函数。但编译器并不能总是做出最佳选择，因此有时需要属性。 **示例 1**， **示例 2**， **示例 3**， **示例 4**， **示例 5**。

Cachegrind 是一个确定函数是否内联的好工具。查看 Cachegrind 的输出时，你可以判断一个函数是否已经内联，只有当它的第一行和最后一行*没有*标记事件计数时才是。例如：

```
       .   #[inline(always)]
       .   fn inlined(x: u32, y: u32) -> u32 {
 700,000       eprintln!("inlined: {} + {}", x, y);
 200,000       x + y
       .   }
       .
       .   #[inline(never)]
 400,000   fn not_inlined(x: u32, y: u32) -> u32 {
 700,000       eprintln!("not_inlined: {} + {}", x, y);
 200,000       x + y
 200,000   }
```

添加内联属性后，您应该再次进行测量，因为效果可能是不可预测的。有时不会产生影响，因为以前内联的附近函数不再内联。有时会使代码变慢。内联还可能会影响编译时间，特别是涉及跨 crate 内联的情况，这会涉及到函数的内部表示的重复。

更困难的情况

有时您有一个大型函数，并且有多个调用点，但只有一个调用点是热点。您希望为了速度内联热点调用点，但不希望内联冷调用点以避免不必要的代码膨胀。处理这种情况的方法是将函数分割为始终内联和从不内联的变体，后者调用前者。

例如，这个函数：

```
fn my_function() {
    one();
    two();
    three();
}
```

会变成这两个函数：

```rust
// 在热点调用点使用这个。
#[inline(always)]
fn inlined_my_function() {
    one();
    two();
    three();
}

// 在冷调用点使用这个。
#[inline(never)]
fn uninlined_my_function() {
    inlined_my_function();
}
```

示例 1， **示例 2**。

哈希

`HashSet` 和 `HashMap` 是两种广泛使用的类型。默认的哈希算法未指定，但在撰写本文时，默认的是一种称为 SipHash 1-3 的算法。这个算法质量很高——它提供了很高的碰撞保护——但相对较慢，特别是对于像整数这样的短键。

如果性能分析显示哈希操作很热门，并且哈希拒绝服务攻击不是您的应用程序的关注点，那么使用具有更快哈希算法的哈希表可以提供很大的速度优势。

- `rustc-hash` 提供了 `FxHashSet` 和 `FxHashMap` 类型，它们是 `HashSet` 和 `HashMap` 的即插即用替代品。其哈希算法质量低，但非常快，特别是对于整数键，已经被发现在 `rustc` 中优于所有其他哈希算法。(`fxhash` 是相同算法和类型的一个较旧、维护程度较低的实现。)
- `fnv` 提供了 `FnvHashSet` 和 `FnvHashMap` 类型。其哈希算法比 `rustc-hash` 更高质量，但稍微慢一些。
- `ahash` 提供了 `AHashSet` 和 `AHashMap` 。其哈希算法可以利用一些处理器上可用的 AES 指令支持。

如果哈希性能在您的程序中很重要，值得尝试多种替代方案。例如，在 `rustc` 中观察到以下结果。

- 从 `fnv` 切换到 `fxhash` 可以实现高达 6% 的加速。
- 尝试从 `fxhash` 切换到 `ahash` 导致1-4% 的减速。
- 尝试从 `fxhash` 切换回默认哈希器导致4-84% 的减速!

如果决定普遍使用其中一种替代方案，比如 `FxHashSet` / `FxHashMap` ，很容易在某些地方意外使用 `HashSet` / `HashMap` 。您可以使用 Clippy 来避免这个问题。

有些类型不需要哈希。例如，您可能有一个包装整数的新类型，而整数值是随机的，或者接近随机的。对于这种类型，哈希值的分布与值本身的分布不会有太大的不同。在这种情况下， `nohash_hasher` crate 可以很有用。

哈希函数设计是一个复杂的话题，超出了本书的范围。 `ahash` 文档 中有很好的讨论。

堆分配

堆分配成本适中。具体细节取决于使用的分配器，但每次分配（和释放）通常涉及获取全局锁、执行一些非平凡的数据结构操作，以及可能执行系统调用。小型分配并不一定比大型分配更便宜。值得了解的是，哪些 Rust 数据结构和操作会引起分配，因为避免它们可以极大地提高性能。

Rust 容器速查表 提供了常见 Rust 类型的可视化，并且是以下部分的绝佳参考。

分析

如果通用性能分析器显示 `malloc` 、 `free` 和相关函数为热点，则尝试减少分配速率和/或使用替代分配器可能是值得的。

DHAT 是减少分配速率时使用的优秀分析器。它适用于 Linux 和其他一些 Unix 系统。它可以精确地识别热点分配位置及其分配速率。确切的结果会有所不同，但在 rustc 中的经验表明，将每百万条指令的分配速率降低 10 次可能会带来可衡量的性能改进（例如，约为 1%）。

以下是 DHAT 的一些示例输出。

```
AP 1.1/25 (2 children) {
  Total:     54,533,440 bytes (4.02%, 2,714.28/Minstr) in 458,839
blocks (7.72%, 22.84/Minstr), avg size 118.85 bytes, avg lifetime
1,127,259,403.64 instrs (5.61% of program duration)
  At t-gmax: 0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  At t-end:  0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  Reads:     15,993,012 bytes (0.29%, 796.02/Minstr), 0.29/byte
  Writes:    20,974,752 bytes (1.03%, 1,043.97/Minstr), 0.38/byte
  Allocated at {
    #1: 0x95CACC9: alloc (alloc.rs:72)
    #2: 0x95CACC9: alloc (alloc.rs:148)
    #3: 0x95CACC9:
reserve_internal<syntax::tokenstream::TokenStream,alloc::alloc::Glob
al> (raw_vec.rs:669)
    #4: 0x95CACC9:
reserve<syntax::tokenstream::TokenStream,alloc::alloc::Global>
(raw_vec.rs:492)
    #5: 0x95CACC9: reserve<syntax::tokenstream::TokenStream>
(vec.rs:460)
    #6: 0x95CACC9: push<syntax::tokenstream::TokenStream>
(vec.rs:989)
    #7: 0x95CACC9: parse_token_trees_until_close_delim
(tokentrees.rs:27)
    #8: 0x95CACC9: syntax::parse::lexer::tokentrees::<impl
syntax::parse::lexer::StringReader<'a>>::parse_token_tree
(tokentrees.rs:81)
  }
}
```

本书不涵盖此示例中的所有内容，但显然 DHAT 提供了大量关于分配的信息，比如它们发生在何处、发生频率如何、大小如何、存活多久以及被访问多频繁等等。

Box

`Box` 是最简单的堆分配类型。`Box<T>` 值是在堆上分配的 `T` 值。

有时在结构体或枚举字段中使用一个或多个 `Box` 来使类型更小是值得的。（有关此内容的更多信息，请参阅 类型大小 章节。）

除此之外，`Box` 是直接的，没有太多优化的余地。

Rc/Arc

`Rc` / `Arc` 与 `Box` 类似，但堆上的值伴随着两个引用计数。它们允许值共享，这可以是减少内存使用的有效方式。

然而，如果用于很少被共享的值，它们可能会增加分配速率，因为会为本来可能不会被堆分配的值进行堆分配。 **示例**。

与 `Box` 不同，在 `Rc` / `Arc` 上调用 `clone` 不涉及分配。相反，它只是增加引用计数。

Vec

`Vec` 是一个具有大量优化空间的堆分配类型，可以优化分配数量和/或最小化浪费空间的量。要做到这一点，需要了解其元素是如何存储的。

`Vec` 包含三个字：长度、容量和指针。如果容量为非零且元素大小为非零，则指针将指向堆分配的内存；否则，它将不指向分配的内存。

即使 `Vec` 本身不是堆分配的，但元素（如果存在且大小非零）总是会被堆分配。如果存在大小非零的元素，则保存这些元素的内存可能比必要的更大，以提供额外的未来元素的空间。存在的元素数量称为长度，而无需重新分配的元素数量称为容量。

当向量需要增长超出当前容量时，元素将被复制到一个更大的堆分配中，旧的堆分配将被释放。

Vec 的增长

通过常见方式创建的新的空的 `Vec` （ `vec![]` 或 `Vec::new` 或 `Vec::default` ）的长度和容量为零，不需要堆分配。如果你反复将单个元素推送到 `Vec` 的末尾，它会周期性地重新分配。增长策略没有明确规定，但在撰写本文时，它使用准加倍策略，导致以下容量：0、4、8、16、32、64 等等。（它直接从 0 跳到 4，而不是经过 1 和 2，因为这样在实践中避免了许多分配。）随着向量的增长，重新分配的频率将按指数方式减少，但可能浪费的多余容量将呈指数增长。

这种增长策略对于可增长数据结构来说是典型的，并且在一般情况下是合理的，但如果你事先知道向量的可能长度，通常可以做得更好。如果你有一个热点向量分配站点（例如热 `Vec::push` 调用），值得使用 `eprintln!` 打印该站点的向量长度，然后进行一些后处理（例如，使用 `counts` ）来确定长度分布。例如，你可能有许多短向量，或者你可能有较少数量的非常长的向量，最佳的优化分配站点的方法将相应地变化。

短 Vec

如果你有许多短向量，你可以使用 `smallvec` crate 中的 `SmallVec` 类型。`SmallVec<[T; N]>` 是 `Vec` 的一个可插拔替代品，可以在 `SmallVec` 本身中存储 `N` 个元素，然后在元素数量超过这个值时切换到堆分配。（注意， `vec![]` 文字必须被替换为 `smallvec![]` 文字。）

示例 1，**示例 2**。

当适当使用时， `SmallVec` 可靠地减少了分配速率，但其使用并不保证性能的改进。对于普通操作，它比 `Vec` 稍慢，因为它必须始终检查元素是否已经堆分配。此外，如果 `N` 很高或 `T` 很大，则 `SmallVec<[T; N]>` 本身可能比 `Vec<T>` 更大，并且复制 `SmallVec` 值将会更慢。像往常一样，需要进行基准测试以确认优化是否有效。

如果你有许多短向量 *并且* 你精确知道它们的最大长度，那么使用 `arrayvec` crate 中的 `ArrayVec` 比 `SmallVec` 更好。它不需要回退到堆分配，这使得它稍微快一点。

示例。

更长的 Vec

如果你知道向量的最小或确切大小，你可以使用 `Vec::with_capacity` 、`Vec::reserve` 或 `Vec::reserve_exact` 预留特定的容量。例如，如果你知道向量将至少增长到具有 20 个元素，这些函数可以立即提供至少容量为 20 的向量，只需一个分配，而逐个推送项目则会导致四个分配（容量分别为 4、8、16 和 32）。

示例。

如果你知道向量的最大长度，上述函数也让你不必不必要地分配多余的空间。同样，`Vec::shrink_to_fit` 可用于最小化浪费的空间，但请注意，它可能导致重新分配。

String（字符串）

`String` 包含堆分配的字节。`String` 的表示和操作与 `Vec<u8>` 非常相似。许多与 `Vec` 的增长和容量相关的方法在 `String` 中也有等效方法，例如 `String::with_capacity`。

`smallstr` 包中的 `SmallString` 类型类似于 `SmallVec` 类型。

`smartstring` 包中的 `String` 类型是 `String` 的一种可替代类型，对于包含少于三个字的字符串避免了堆分配。在 64 位平台上，这意味着任何长度小于 24 字节的字符串，包括所有包含 23 个或更少 ASCII 字符的字符串。**示例**。

请注意，`format!` 宏会生成一个 `String`，这意味着它会执行一次分配。如果可以通过使用字符串字面值避免调用 `format!`，那么就可以避免此分配。**示例**。`std::format_args` 和/或 `lazy_format` 包可能有助于实现这一点。

哈希表

`HashSet` 和 `HashMap` 是哈希表。它们的表示和操作类似于 `Vec`，在分配方面：它们有一个单一的连续堆分配，保存键和值，并在表增长时根据需要重新分配。许多与 `Vec` 的增长和容量相关的方法在 `HashSet` / `HashMap` 中也有等效方法，例如 `HashSet::with_capacity`。

clone（克隆）

对包含堆分配内存的值调用 `clone` 通常会涉及额外的分配。例如，在非空 `Vec` 上调用 `clone` 需要为元素进行新的分配（但请注意，新 `Vec` 的容量可能与原始 `Vec`

的容量不同）。唯一的例外是 `Rc` / `Arc`，在这种情况下，`clone` 调用只会增加引用计数。

`clone_from` 是 `clone` 的一种替代方法。`a.clone_from(&b)` 等效于 `a = b.clone()`，但可能会避免不必要的分配。例如，如果要将一个 `Vec` 克隆到现有 `Vec` 的顶部，则会尝试重用现有 `Vec` 的堆分配，如以下示例所示。

虽然 `clone` 通常会导致分配，但在许多情况下，这是合理的使用方式，而且通常可以使代码更简洁。使用性能分析数据来查看哪些 `clone` 调用是热点，并值得付出努力来避免。

有时 Rust 代码会包含不必要的 `clone` 调用，原因是（a）程序员错误，或者（b）代码更改使以前必要的 `clone` 调用变得不必要。如果看到一个热点 `clone` 调用似乎不必要，有时可以简单地将其删除。

to_owned（转为拥有的）

[`ToOwned::to_owned`] 实现了许多常见类型。它从借用数据创建拥有的数据，通常通过克隆，并因此经常导致堆分配。例如，它可以用于从 `&str` 创建 `String`。

有时可以通过将借用数据的引用存储在结构体中而不是拥有的副本来避免 `to_owned` 调用（以及相关调用，如 `clone` 和 `to_string`）。这需要在结构体上添加生命周期注释，使代码复杂化，并且只有在分析和基准测试表明这样做是值得的时候才应该这样做。

Cow（借用或拥有）

有时候代码会处理借用和拥有数据的混合。想象一下错误消息的向量，其中一些是静态字符串文字，而另一些是使用 `format!` 构造的。显而易见的表示是 `Vec<String>`，如下例所示。

```
let mut errors: Vec<String> = vec![];
errors.push("something went wrong".to_string());
errors.push(format!("something went wrong on line {}", 100));
```

这需要一个 `to_string` 调用将静态字符串文字提升为 `String`，这会导致分配。

相反，可以使用 `Cow` 类型，它可以保存借用或拥有的数据。借用值 `x` 被包装在 `Cow::Borrowed(x)` 中，而拥有值 `y` 被包装在 `Cow::Owned(y)` 中。`Cow` 还为各种字符串、切片和路径类型实现了 `From<T>` trait，因此通常也可以使用 `into`。（或者 `Cow::from`，它更长但更易读，因为它使类型更清晰。）以下示例将所有内容结合在一起。

```
use std::borrow::Cow;
let mut errors: Vec<Cow<'static, str>> = vec![];
errors.push(Cow::Borrowed("something went wrong"));
errors.push(Cow::Owned(format!("something went wrong on line {}",
100)));
errors.push(Cow::from("something else went wrong"));
errors.push(format!("something else went wrong on line {}",
101).into());
```

`errors` 现在保存了借用和拥有的数据，而不需要任何额外的分配。此示例涉及 `&str` / `String`，但其他配对，如 `&[T]` / `Vec<T>` 和 `&Path` / `PathBuf` 也是可能的。

重复使用集合

有时您需要逐步构建集合，比如 `Vec`。通常最好通过修改单个 `Vec` 来完成，而不是构建多个 `Vec` 然后将它们组合起来。

例如，如果您有一个可能被多次调用的函数 `do_stuff`，它会产生一个 `Vec`：

```
fn do_stuff(x: u32, y: u32) -> Vec<u32> {
    vec![x, y]
}
```

修改传入的 `Vec` 可能更好：

```
fn do_stuff(x: u32, y: u32, vec: &mut Vec<u32>) {
    vec.push(x);
    vec.push(y);
}
```

有时值得保留一个“工作集合”，以便重复使用。例如，如果每次循环迭代都需要一个 `Vec`，您可以在循环外声明 `Vec`，在循环体内使用它，然后在循环体末尾调用 `clear`（清空 `Vec` 而不影响其容量）。这样做可以避免分配，但会使每次迭代对 `Vec` 的使用与其他迭代无关的事实变得不明显。 **示例 1**, **示例 2**。

类似地，有时值得在结构体内保留一个工作集合，以便在重复调用的一个或多个方法中重复使用。

从文件中读取行

[`BufRead::lines`] 使逐行读取文件变得容易：

```
use std::io::{self, BufRead};
let mut lock = io::stdin().lock();
for line in lock.lines() {
    process(&line?);
}
```

但它生成的迭代器返回 `io::Result<String>`，这意味着它为文件中的每一行进行分配。

使用循环遍历 [`BufRead::read_line`] 中的工作集合 `String` 是一种替代方法：

```
use std::io::{self, BufRead};
let mut lock = io::stdin().lock();
let mut line = String::new();
while lock.read_line(&mut line)? != 0 {
    process(&line);
    line.clear();
}
```

这样可以将分配数量减少到最多一小撮，甚至可能只有一个。（确切的数量取决于需要多少次重新分配 `line`，这取决于文件中行长度的分布。）

只有当循环体可以处理 `&str` 而不是 `String` 时，这才有效。

示例。

使用替换分配器

还可以通过使用不同的分配器来提高堆分配性能，而不必更改代码。有关详细信息，请参阅[替代分配器]部分。

避免退化

为了确保您的代码执行的分配次数和/或大小不会意外增加，您可以使用 *堆使用测试* 功能来编写测试，检查特定代码片段分配了预期的堆内存量。

类型大小

缩小经常实例化的类型可以提高性能。

例如，如果内存使用量很高，像 DHAT 这样的堆分析器可以识别热点分配点和涉及的类型。缩小这些类型可以减少峰值内存使用量，并可能通过减少内存流量和缓存压力来提高性能。

此外，大于 128 字节的 Rust 类型会使用 `memcpy` 而不是内联代码进行复制。如果在分析中大量出现 `memcpy`，DHAT 的 “copy profiling” 模式将告诉您热点 `memcpy` 调用的确切位置和涉及的类型。将这些类型缩小到 128 字节或更小可以通过避免 `memcpy` 调用和减少内存流量来使代码更快。

测量类型大小

`std::mem::size_of` 给出了类型的大小（以字节为单位），但通常您也想了解确切的布局。例如，由于单个超大变体，枚举可能会很大。

`-Zprint-type-sizes` 选项正是如此。它未在 rustc 的发布版本中启用，因此您需要使用 rustc 的夜间版本。以下是通过 Cargo 可能的一种调用：

```
RUSTFLAGS=-Zprint-type-sizes cargo +nightly build --release
```

以下是 rustc 的一个可能调用：

```
rustc +nightly -Zprint-type-sizes input.rs
```

它将打印出所有正在使用的类型的大小、布局和对齐方式的详细信息。例如，对于此类型：

```
enum E {
    A,
    B(i32),
    C(u64, u8, u64, u8),
    D(Vec<u32>),
}
```

它打印出以下内容，以及有关一些内置类型的信息。

```
print-type-size type: `E`: 32 bytes, alignment: 8 bytes
print-type-size     discriminant: 1 bytes
print-type-size     variant `D`: 31 bytes
print-type-size         padding: 7 bytes
print-type-size         field `.0`: 24 bytes, alignment: 8 bytes
print-type-size     variant `C`: 23 bytes
print-type-size         field `.1`: 1 bytes
print-type-size         field `.3`: 1 bytes
print-type-size         padding: 5 bytes
print-type-size         field `.0`: 8 bytes, alignment: 8 bytes
print-type-size         field `.2`: 8 bytes
print-type-size     variant `B`: 7 bytes
print-type-size         padding: 3 bytes
print-type-size         field `.0`: 4 bytes, alignment: 4 bytes
print-type-size     variant `A`: 0 bytes
```

输出显示以下内容。

- 类型的大小和对齐。
- 对于枚举，判别式的大小。
- 对于枚举，每个变体的大小（从大到小排序）。
- 所有字段的大小、对齐和顺序。（请注意，编译器已重新排序变体 `C` 的字段，以最小化 `E` 的大小。）
- 所有填充的大小和位置。

或者，可以使用 top-type-sizes crate 以更紧凑的形式显示输出。

一旦您了解了热门类型的布局，就有多种方法可以缩小它们。

字段排序

Rust 编译器会自动对结构体和枚举的字段进行排序，以最小化它们的大小（除非指定了 `#[repr(C)]` 属性），因此您不必担心字段的顺序。但是，还有其他方法可以最小化热门类型的大小。

更小的枚举

如果一个枚举有一个超大的变体，请考虑将一个或多个字段放入 Box 中。例如，您可以将此类型更改为：

```
type LargeType = [u8; 100];
enum A {
    X,
    Y(i32),
    Z(i32, LargeType),
}
```

变为：

```
enum A {
    X,
    Y(i32),
    Z(Box<(i32, LargeType)>),
}
```

这会减小类型大小，但需要为 `A::Z` 变体进行额外的堆分配。如果 `A::Z` 变体相对较少，则更有可能实现净性能提升。 `Box` 还会使得 `A::Z` 在 `match` 模式中稍微不那么方便。 **示例 1**, **示例 2**, **示例 3**, **示例 4**, **示例 5**, **示例 6**。

更小的整数

通常可以通过使用较小的整数类型来缩小类型。例如，虽然使用 `usize` 作为索引最自然，但通常可以将索引存储为 `u32` 、 `u16` 或甚至 `u8` ，然后在使用点进行强制转

换为 `usize`。 **示例 1**, **示例 2**。

Box 包装的切片

Rust 向量包含三个字：长度、容量和指针。如果您有一个向量在未来不太可能更改，您可以使用 `Vec::into_boxed_slice` 将其转换为*boxed slice*。Boxed slice 仅包含两个字，一个长度和一个指针。任何多余的元素容量都会被丢弃，这可能会导致重新分配。

```
let v: Vec<u32> = vec![1, 2, 3];
assert_eq!(size_of_val(&v), 3 * size_of::<usize>());

let bs: Box<[u32]> = v.into_boxed_slice();
assert_eq!(size_of_val(&bs), 2 * size_of::<usize>());
```

可以使用 `slice::into_vec` 将 boxed slice 转换回向量，而无需克隆或重新分配。

ThinVec

与 boxed slice 的替代方案是 [`thin_vec`] crate 中的 `ThinVec`。它在功能上等同于 `Vec`，但将长度和容量存储在元素相同的分配中（如果有的话）。这意味着 `size_of::<ThinVec<T>>` 仅为一个字。

`ThinVec` 在经常为空的向量类型中是一个不错的选择。如果枚举的最大变体包含一个 `Vec`，它也可以用于缩小。 [`thin_vec`]: https://crates.io/crates/thin-vec

避免退化

如果一个类型很热门，以至于它的大小会影响性能，最好使用静态断言来确保它不会意外退化。以下示例使用了 `static_assertions` crate 中的宏。

```
// 这个类型经常使用。确保它不会意外增大。
#[cfg(target_arch = "x86_64")]
static_assertions::assert_eq_size!(HotType, [u8; 64]);
```

`cfg` 属性很重要，因为类型大小在不同的平台上可能会有所不同。将断言限制为 `x86_64`（通常是最常用的平台）很可能足以防止实际中的退化。

标准库类型

值得阅读常见标准库类型的文档，比如 `Box` 、 `Vec` 、 `Option` 、 `Result` 以及 `Rc` / `Arc` ，以发现有时可以用来提高性能的有趣函数。

值得了解的还有高性能替代标准库类型的选择，比如 `Mutex` 、 `RwLock` 、 `Condvar` 和 `Once` 。

Box

表达式 `Box::default()` 的效果与 `Box::new(T::default())` 相同，但可能更快，因为编译器可以直接在堆上创建值，而不是在栈上构造然后复制。 **示例**。

Vec

创建长度为 `n` 的零填充 `Vec` 的最佳方法是使用 `vec![0; n]` 。这简单且可能比其他替代方法更快，比如使用 `resize` 、 `extend` 或涉及 `unsafe` 的任何方法，因为它可以使用操作系统的辅助。

`Vec::remove` 移除特定索引处的元素，并将所有后续元素向左移动一个位置，时间复杂度为 O(n)。 `Vec::swap_remove` 用最后一个元素替换特定索引处的元素，这不保留顺序，但时间复杂度为 O(1)。

`Vec::retain` 高效地从 `Vec` 中移除多个项目。其他集合类型如 `String` 、 `HashSet` 和 `HashMap` 也有类似的方法。

Option 和 Result

`Option::ok_or` 将 `Option` 转换为 `Result` ，如果 `Option` 值为 `None` ，则传递一个 `err` 参数，该参数将用于错误处理。 `err` 会立即计算。如果其计算成本很高，

应改用 `Option::ok_or_else`，通过闭包惰性计算错误值。例如，这样：

```
let r = o.ok_or(expensive()); // 总是评估 `expensive()`
```

应更改为：

```
let r = o.ok_or_else(|| expensive()); // 仅在需要时评估 `expensive()`
```

示例。

对于 `Option::map_or`、`Option::unwrap_or`、`Result::or`、`Result::map_or` 和 `Result::unwrap_or` 也有类似的替代方法。

Rc/Arc

`Rc::make_mut` / `Arc::make_mut` 提供写时复制语义。它们使 `Rc` / `Arc` 引用可变。如果引用计数大于一，它们将 `clone` 内部值以确保唯一所有权；否则，它们将修改原始值。它们不经常需要，但在某些情况下可能非常有用。**示例 1**，**示例 2**。

Mutex、RwLock、Condvar 和 Once

`parking_lot` crate 提供了这些同步类型的替代实现。`parking_lot` 类型的 API 和语义与标准库中等价类型的相似但不相同。

以前，`parking_lot` 版本通常比标准库中的版本更小、更快、更灵活，但在某些平台上，标准库版本已经有了很大的改进。因此，在切换到 `parking_lot` 之前应该先进行测量。

如果决定普遍使用 `parking_lot` 类型，很容易在某些地方意外使用标准库的等价类型。您可以使用 Clippy 来避免此问题。

迭代器

collect 和 extend

`Iterator::collect` 将迭代器转换为诸如 `Vec` 之类的集合，这通常需要进行分配。如果转换成集合之后只是再次进行迭代，应避免调用 `collect` 。

因此，通常最好从函数返回像 `impl Iterator<Item=T>` 这样的迭代器类型，而不是 `Vec<T>` 。请注意，有时这些返回类型需要额外的生命周期，正如这篇博文所解释的那样。 **示例**。

类似地，你可以使用 `extend` 来扩展现有集合（如 `Vec` ）的迭代器，而不是将迭代器收集到 `Vec` 中，然后使用 `append` 。

最后，当你编写一个迭代器时，如果可能的话，通常值得实现 `Iterator::size_hint` 或 `ExactSizeIterator::len` 方法。使用迭代器的 `collect` 和 `extend` 调用可能会减少分配，因为它们提前了解了迭代器产生的元素数量。

链式调用

`chain` 可能非常方便，但与单个迭代器相比，它也可能更慢。如果可能的话，对于热迭代器，最好避免使用它。 **示例**。

类似地， `filter_map` 可能比先使用 `filter` 再使用 `map` 更快。

块 (chunks)

当需要分块迭代器且块大小已知确切地能整除切片长度时，使用更快的 `slice::chunks_exact` 而不是 `slice::chunks` 。

当块大小未知确切地能整除切片长度时，仍然可以更快地使用 `slice::chunks_exact` 结合 `ChunksExact::remainder` 或手动处理多余元素。 **示例 1**, **示例 2**。

对于相关的迭代器也是如此：

- `slice::rchunks`, `slice::rchunks_exact`, 和 `RChunksExact::remainder`;
- `slice::chunks_mut`, `slice::chunks_exact_mut`, 和 `ChunksExactMut::into_remainder`;
- `slice::rchunks_mut`, `slice::rchunks_exact_mut`, 和 `RChunksExactMut::into_remainder`.

边界检查

在 Rust 中，默认情况下，对切片（slices）和向量（vectors）等容器类型的访问涉及边界检查。这些检查可能会影响性能，在热循环(hot loop, 经常执行的循环代码)内，尽管不太频繁，但仍然存在。

有几种安全的方法可以更改代码，以便编译器了解容器长度并优化掉边界检查。

- 在循环中用迭代替换直接元素访问。
- 在循环中索引到 `Vec` 时，先制作 `Vec` 的切片，然后在循环内部对切片进行索引。
- 在索引变量的范围上添加断言。 **示例 1**，**示例 2**。

让这些方法起作用可能有些棘手。边界检查指南 对这个主题进行了更详细的讨论。

作为最后的手段，还有不安全的方法 `get_unchecked` 和 `get_unchecked_mut` 。

输入/输出

锁定

Rust 的 `print!` 和 `println!` 宏在每次调用时都会锁定标准输出(stdout)。如果你重复调用这些宏，手动锁定标准输出可能会更好。

例如，将这段代码改为：

```
for line in lines {
    println!("{}", line);
}
```

改为这样：

```
use std::io::Write;
let mut stdout = std::io::stdout();
let mut lock = stdout.lock();
for line in lines {
    writeln!(lock, "{}", line)?;
}
// 当 `lock` 被丢弃时，标准输出会解锁
```

当对标准输入(stdin)和标准错误(stderr)进行重复操作时，同样可以进行锁定。

缓冲

Rust 的文件 I/O 默认是非缓冲的。如果你对文件或网络套接字进行了许多小的重复读取或写入调用，可以使用 `BufReader` 或 `BufWriter` 。它们会维护一个内存缓冲区用于输入和输出，最大限度地减少所需的系统调用次数。

例如，将这段非缓冲写入器的代码：

```
use std::io::Write;
let mut out = std::fs::File::create("test.txt")?;
for line in lines {
    writeln!(out, "{}", line)?;
}
```

改为这样：

```
use std::io::{BufWriter, Write};
let mut out = BufWriter::new(std::fs::File::create("test.txt")?);
for line in lines {
    writeln!(out, "{}", line)?;
}
out.flush()?;
```

示例 1, **示例 2**。

对 `flush` 的显式调用并非绝对必要，因为当 `out` 被丢弃时，会自动执行刷新。然而，在这种情况下，任何刷新时发生的错误都会被忽略，而显式刷新会使该错误显式化。

忘记进行缓冲在写入时更为常见。非缓冲和缓冲写入器都实现了 `Write` 特质，这意味着对于向非缓冲写入器和缓冲写入器写入的代码基本相同。相反，非缓冲读取器实现了 `Read` 特质，但缓冲读取器实现了 `BufRead` 特质，这意味着从非缓冲读取器和缓冲读取器读取的代码是不同的。例如，使用 `BufRead::read_line` 或 `BufRead::lines` 在非缓冲读取器中逐行读取文件是困难的，但在缓冲读取器中却很简单。因此，很难像上面的写入器那样为读取器编写一个示例，其中前后版本如此相似。

最后，注意缓冲也适用于标准输出(stdout)，因此在向标准输出进行多次写入时，可能需要结合手动锁定和缓冲。

从文件读取行

本节 解释了如何在使用 `BufRead` 逐行读取文件时避免过多的分配。

将输入读取为原始字节

内置的 String 类型在内部使用 UTF-8，这会在将输入读入其中时增加一些小但非零的开销，因为需要进行 UTF-8 验证。如果你只想处理输入字节，而不必担心 UTF-8（例如，如果你处理的是 ASCII 文本），可以使用 `BufRead::read_until` 。

还有专门用于读取 字节导向的数据行 和处理 字节字符串 的 crates。

日志记录与调试

有时，日志记录代码或调试代码可能会显著减慢程序的运行速度。无论是日志记录/调试代码本身慢，还是提供给日志记录/调试代码的数据收集代码慢，都可能导致这种情况。确保在未启用日志记录/调试时不要执行不必要的工作。 **示例 1**, **示例 2**。

请注意， `assert!` 调用始终会执行，但 `debug_assert!` 调用仅在开发构建中执行。如果您有一个热点断言，但对安全性不是必需的，请考虑将其改为 `debug_assert!` 。 **示例 1**, **示例 2**。

封装类型

Rust 拥有各种“封装”类型，比如 `RefCell` 和 `Mutex`，它们为值提供了特殊的行为。访问这些值可能需要一定的时间。如果通常一起访问多个这样的值，将它们放在单个封装内可能更好。

例如，像这样的结构体：

```
struct S {
    x: Arc<Mutex<u32>>,
    y: Arc<Mutex<u32>>,
}
```

可能更好地表示为：

```
struct S {
    xy: Arc<Mutex<(u32, u32)>>,
}
```

这是否有助于性能取决于值的确切访问模式。 **示例**。

机器码

当您有一个非常热点的小代码片段时，检查生成的机器码是否存在任何效率低下的地方，比如可移除的边界检查，可能是值得的。在处理小片段时，Compiler Explorer 网站是一个很好的资源。 `cargo-show-asm` 是另一个可用于完整 Rust 项目的工具。

相关地， `core::arch` 模块提供了访问特定架构的内部函数，其中许多与 SIMD 指令相关。

并行性

Rust 提供了出色的安全并行编程支持，可以带来很大的性能提升。有多种方法可以将并行性引入程序中，对于任何程序来说，最佳方法将极大地取决于其设计。

深入讨论并行性超出了本书的范围。如果您对此主题感兴趣，可以从 `rayon` 和 `crossbeam` 的文档开始。

你也可以访问我的 深入理解Rust并发编程 一书。

一般性建议

本书前面的章节讨论了 Rust 特定的技术。本节简要概述了一些一般性能原则。

只要避免明显的陷阱（例如，使用非发布版本），Rust 代码通常是快速的且使用内存少的。特别是如果你习惯于动态类型语言如 Python 和 Ruby，或者带有垃圾回收器的静态类型语言如 Java 和 C#。

优化后的代码通常比未优化的代码更复杂，编写起来需要更多的工作。因此，只有值得优化的热点代码才值得优化。

最大的性能改进通常来自于对算法或数据结构的更改，而不是低级优化。 **示例 1**, **示例 2**。

编写能够很好地与现代硬件配合的代码并不总是容易的，但值得努力。例如，尽量减少缓存未命中和分支预测错误。

大多数优化都会带来小幅度的加速。虽然单个小幅度的加速不会被注意到，但如果你能做足够多的话，它们的确会累加起来。

不同的性能分析工具有不同的优势。最好使用多个。

当性能分析表明某个函数很热时，通常有两种常见的方法可以加速：(a) 加速函数本身，和/或者 (b) 尽量减少对它的调用。

消除愚蠢的减速往往比引入聪明的加速更容易。

除非必要，避免计算。延迟/按需计算通常是有利的。 **示例 1**, **示例 2**。

复杂的一般情况通常可以通过乐观地检查更简单的常见特殊情况来避免。 **示例 1**, **示例 2**, **示例 3**。 特别是，在小尺寸占主导地位时，特别处理包含 0、1 或 2 个元素的集合通常是有效的。 **示例 1**, **示例 2**, **示例 3**, **示例 4**。

类似地，当处理重复数据时，通常可以使用简单的数据压缩形式，通过使用常见值的紧凑表示，然后对于不寻常的值，采用次要表作为回退。 **示例 1**, **示例 2**, **示例 3**。

当代码涉及多个情况时，测量各种情况的频率，并首先处理最常见的情况。

当处理具有高局部性的查找时，将一个小型缓存放在数据结构前面可能是一个好主意。

优化后的代码通常具有非显而易见的结构，这意味着解释性注释是有价值的，特别是那些引用了性能分析测量的注释。像“99% 的情况下，这个向量有 0 或 1 个元素，所以首先处理这些情况”的注释可能会很有启发性。

编译时间

虽然这本书主要是关于提高 Rust 程序性能的，但本节讨论的是减少 Rust 程序的编译时间，因为这是许多人感兴趣的相关话题。

最小化编译时间 部分讨论了通过构建配置选择来减少编译时间的方法。本节的其余部分讨论了需要修改程序代码来减少编译时间的方法。

可视化

Cargo 具有一项功能，可以让您可视化程序的编译过程。使用以下命令构建：

```
cargo build --timings
```

完成后，它将打印一个 HTML 文件的名称。在 Web 浏览器中打开该文件。它包含一个 甘特图，显示了程序中各个 crate 之间的依赖关系。这显示了您的 crate 图中有多少并行性，这可能表明是否应该拆分任何序列化编译的大型 crate。有关如何阅读图表的详细信息，请参阅 文档。

LLVM 中间代码

Rust 编译器使用 LLVM 作为其后端。LLVM 的执行可能是编译时间的一个很大部分，特别是当 Rust 编译器的前端生成大量 IR 时，LLVM 需要很长时间来优化。

这些问题可以通过 `cargo llvm-lines` 进行诊断，它显示了哪些 Rust 函数导致生成最多的 LLVM IR。通常最重要的是泛型函数，因为它们在大型程序中可能被实例化几十甚至几百次。

如果一个泛型函数导致 IR 膨胀，有几种修复方法。最简单的方法就是使函数变小。**示例 1**, **示例 2**。

另一种方法是将函数的非泛型部分移动到一个单独的、非泛型的函数中，这个函数只会被实例化一次。是否可能取决于泛型函数的细节。当可能时，非泛型函数通常可以写成泛型函数内部的内部函数，如下面 `std::fs::read` 的代码所示：

```
pub fn read<P: AsRef<Path>>(path: P) -> io::Result<Vec<u8>> {
    fn inner(path: &Path) -> io::Result<Vec<u8>> {
        let mut file = File::open(path)?;
        let size = file.metadata().map(|m| m.len()).unwrap_or(0);
        let mut bytes = Vec::with_capacity(size as usize);
        io::default_read_to_end(&mut file, &mut bytes)?;
        Ok(bytes)
    }
    inner(path.as_ref())
}
```

示例。

有时常见的实用函数，如 `Option::map` 和 `Result::map_err`，会被实例化多次。将它们替换为等效的 `match` 表达式可以帮助编译时间。

这些改变对编译时间的影响通常会很小，尽管偶尔可能会很大。 示例。

这些改变还可以减少二进制文件的大小。