# **Rust是不是就相当于新时代的C语言？**

Rust 语言是一门通用系统级编程语言，无 GC 且能保证内存安全、并发安全和高性能而著称。

自发布以来，截止到 2021 年的今天，经历六年的发展，Rust 得到稳步上升，已逐渐趋于成熟稳定。

至 2016 年开始，截止到 2021 年，Rust 连续五年成为 [StackOverflow 语言榜上最受欢迎的语言](https://link.zhihu.com/?target=https://insights.stackoverflow.com/survey/2020" \t "_blank)。

作者：盼盼编程  
链接：https://www.zhihu.com/question/496992250/answer/2222924757  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

2021 年 2 月 9 号，**Rust 基金会宣布成立**。**华为、AWS、Google、微软、Mozilla、Facebook 等科技行业领军巨头加入 Rust 基金会**，成为白金成员，以致力于在全球范围内推广和发展 Rust 语言。

那 Rust 语言到底有何魅力，能让广大开发者和巨头公司这么感兴趣呢？

本文打算从 Rust 语言自身特性 和 Rust 行业应用盘点两个方面的社区调研来尝试来解答这个问题。供想选择 Rust 的公司参考。

其实 Rust 生态还有很多内容，等待大家挖掘。本文内容还未覆盖完全 Rust 生态的方方面面。

注明： 本文中所罗列数据均来源互联网公开内容。

## **认识 Rust 语言**

编程语言设计在两个看似不可调和的愿望之间长期存在着矛盾对立。

* 安全 ( safe )。 我们想要强类型系统来静态地排除大量错误。 我们要自动内存管理。我们想要数据封装， 这样我们就可以对私有变量执行不变的对象的表示形式，并确保它们将不会被不受信任的代码破坏。
* 控制 (control )。 至少对于 Web 浏览器，操作系统，或游戏引擎这样的 系统编程 (system programming) 程序，约束它们性能或资源是一个重要的问题，我们想了解数据的字节级表示。 我们想要用底层语言 (low-level programming) 的编程技术优化我们程序的时间和空间的使用。 我们希望在需要时使用 裸机 。

然而，按照传统的看法，鱼和熊掌不能兼得。 Java 之类的语言使我们极大的安全保证，但代价是牺牲对底层的控制。结果，对于许多系统编程应用程序，唯一现实的选择是使用一种像 C 或 C++ 提供细粒度的语言控制资源管理。 但是，获得这种控制需要很高的成本。例如，微软最近报告说，他们修复的 70％ 安全漏洞都归因于内存安全违规行为 [33](https://link.zhihu.com/?target=https://rustmagazine.github.io/rust_magazine_2021/chapter_4/safe_system.html%2333" \t "_blank)，并且都是能被强类型系统排除的问题。同样，Mozilla 报告指出，绝大多数关键 他们在 Firefox 中发现的错误是内存有关的[16](https://link.zhihu.com/?target=https://rustmagazine.github.io/rust_magazine_2021/chapter_4/safe_system.html%2316" \t "_blank) 。

如果可以以某种方式两全其美: 安全系统编程的同时对底层有控制权，岂不美哉。因此，Rust 语言应运而生。

官方网如此介绍 Rust : 一门赋予每个人 构建可靠且高效软件能力的语言。

Rust 语言有三大优势值得大家关注：

1. 高性能。Rust 速度惊人且内存利用率极高。由于没有运行时和垃圾回收，它能够胜任对性能要求特别高的服务，可以在嵌入式设备上运行，还能轻松和其他语言集成。
2. 可靠性。Rust 丰富的类型系统和所有权模型保证了内存安全和线程安全，让您在编译期就能够消除各种各样的错误。
3. 生产力。Rust 拥有出色的文档、友好的编译器和清晰的错误提示信息， 还集成了一流的工具——包管理器和构建工具， 智能地自动补全和类型检验的多编辑器支持， 以及自动格式化代码等等。

Rust 足够底层，如果有必要，它可以像 C 一样进行优化，以实现最高性能。

抽象层次越高，内存管理越方便，可用库越丰富，Rust 程序代码就越多，做的事情越多，但如果不进行控制，可能导致程序膨胀。

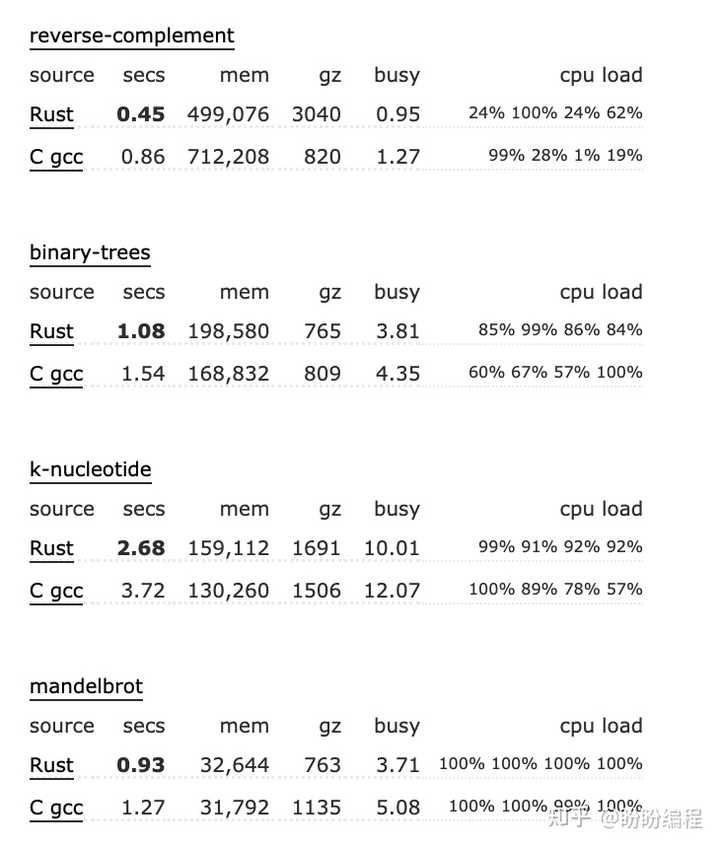
然而，Rust 程序的优化也很不错，有时候比 C 语言更好，C 语言适合在逐个字节逐个指针的级别上编写最小的代码，而 Rust 具有强大的功能，能够有效地将多个函数甚至整个库组合在一起。

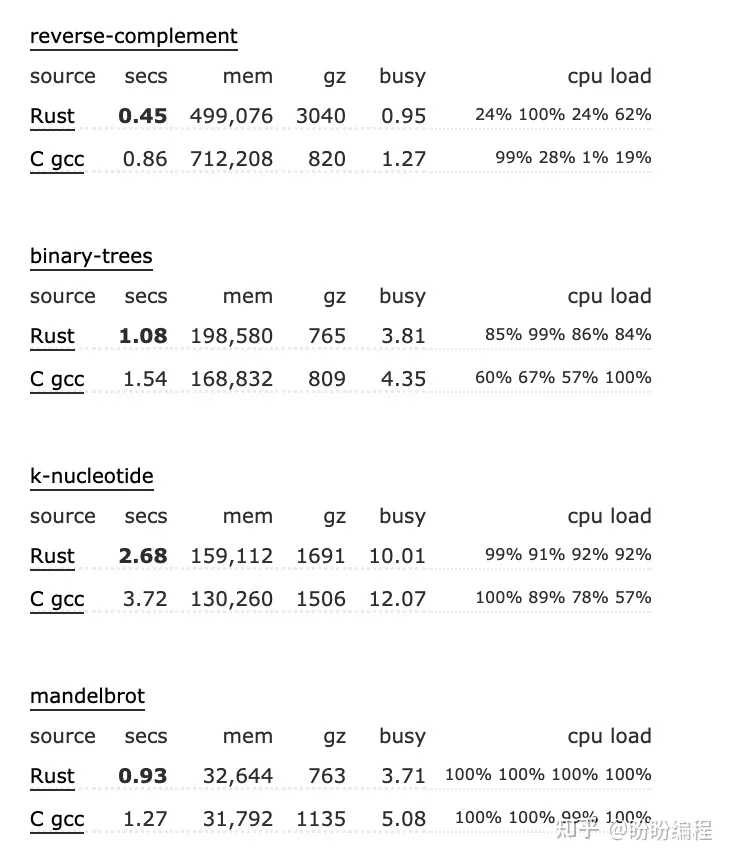
但是，最大的潜力是可以无畏（fearless）地并行化大多数 Rust 代码，即使等价的 C 代码并行化的风险非常高。在这方面，Rust 语言是比 C 语言更为成熟的语言。

Rust 语言也支持高并发零成本的[异步编程](https://www.zhihu.com/search?q=%E5%BC%82%E6%AD%A5%E7%BC%96%E7%A8%8B&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":2222924757}" \t "_blank)，Rust 语言应该是首个支持异步编程的系统级语言。

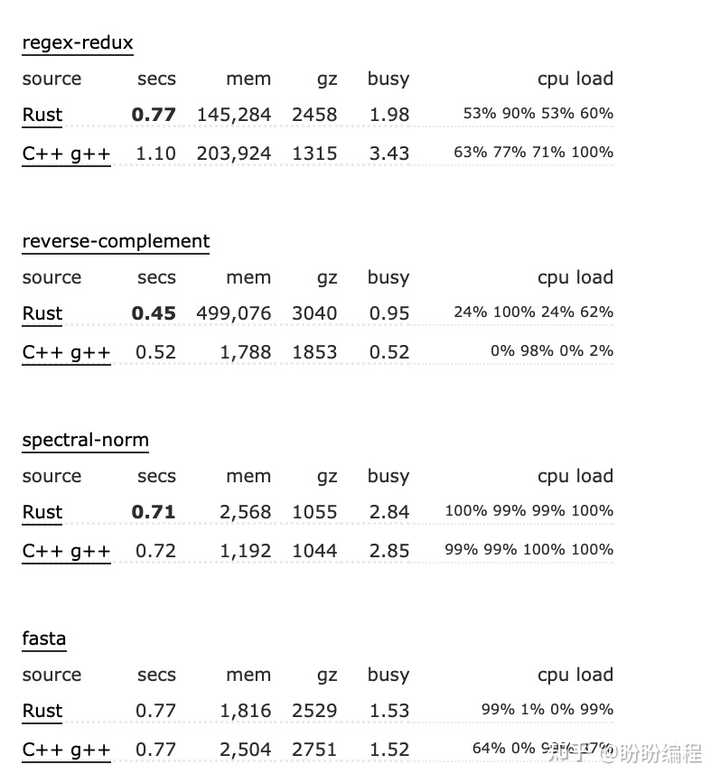
### **媲美 C / Cpp 的高性能**

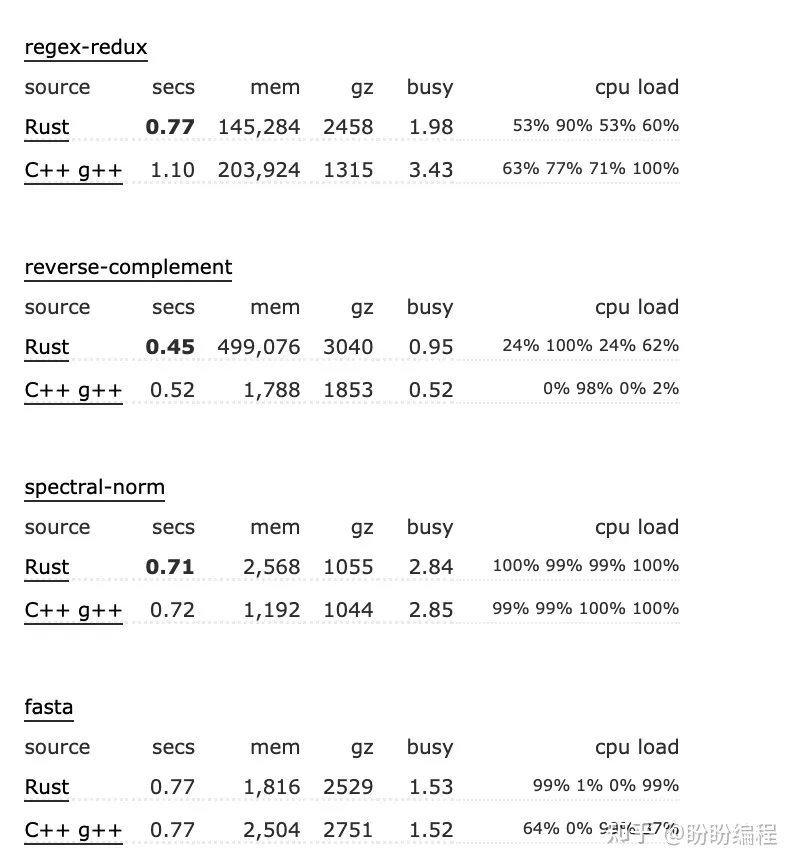
**Rust vs C**



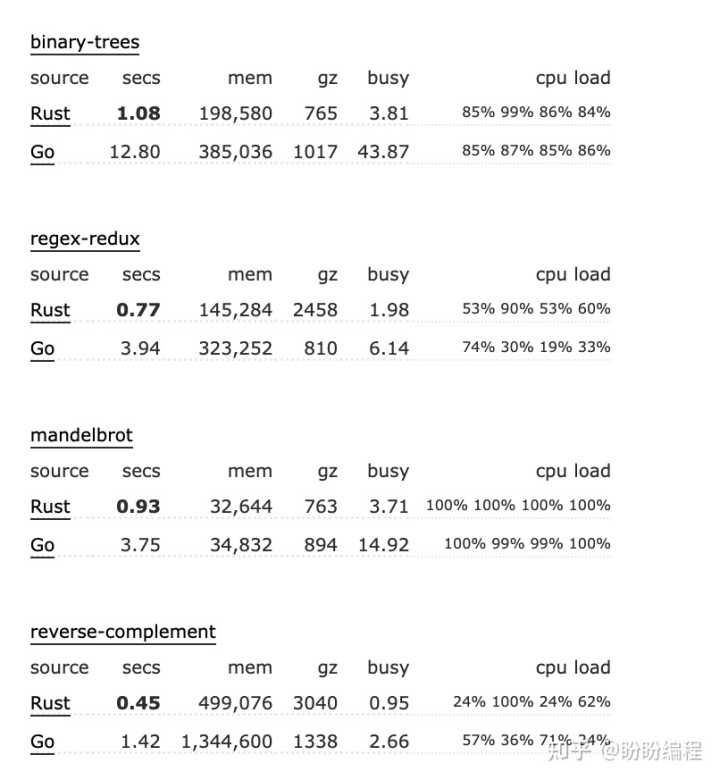


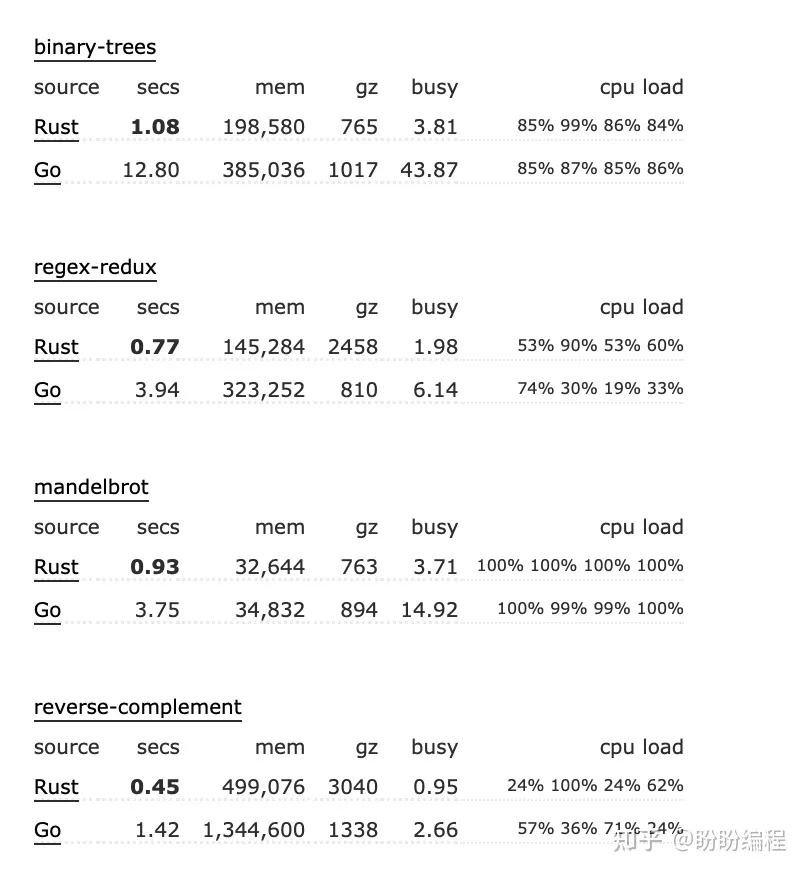
**Rust vs Cpp**





**Rust vs Go**





用 Rust 编写的程序的运行时速度和内存使用量应该和用 C 编写的程序差不多，但这两种语言的总体编程风格不同，很难去概括它们的性能。

总的来说：

1. 抽象是一把双刃剑。Rust 语言抽象程度比 C 语言更高，抽象会隐藏一些不是那么优化的代码，这意味着，默认实现的 Rust 代码性能不是最好的。所以，你的 Rust 代码必须经过优化才能达到媲美 C 的性能。Unsafe Rust 就是高性能出口。
2. Rust 默认线程安全，消除数据竞争，让多线程并发编程更具实用价值。
3. Rust 在有些方面确实比 C 快。理论上，C 语言什么都可以做。 但在实践中，C 抽象能力比较低，不那么现代化，开发效率比较低。只要开发者有无限时间和精力，就可以让 C 语言在这些方面比 Rust 更快。

因为 C 语言足以代表高性能，下面就分别谈一下 C 和 Rust 的异同。如果你熟悉 C/Cpp，也可以根据此对比来评估 Cpp 和 Rust。

### **Rust 和 C 都是硬件直接抽象**

Rust 和 C 都是直接对硬件的抽象，都可看作一种「[可移植汇编程序](https://www.zhihu.com/search?q=%E5%8F%AF%E7%A7%BB%E6%A4%8D%E6%B1%87%E7%BC%96%E7%A8%8B%E5%BA%8F&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":2222924757}" \t "_blank)」。

Rust 和 C 都能控制数据结构的内存布局、整数大小、栈与堆内存分配、指针间接寻址等，并且一般都能翻译成可理解的机器代码，编译器很少插入 "魔法"。

即便 Rust 比 C 有更高层次的结构，如迭代器、特质（trait）和[智能指针](https://www.zhihu.com/search?q=%E6%99%BA%E8%83%BD%E6%8C%87%E9%92%88&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":2222924757}" \t "_blank)，它们也被设计为可预测地优化为简单的机器代码（又称 "零成本抽象"）。

Rust 的类型的内存布局很简单，例如，可增长的字符串String 和 Vec<T> 正好是{byte\*, capacity, length}。Rust 没有任何像 Cpp 里的 移动 或 [复制构造函数](https://www.zhihu.com/search?q=%E5%A4%8D%E5%88%B6%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":2222924757}" \t "_blank) 这样的概念，所以对象的传递保证不会比传递指针或 memcpy 更复杂。

Rust 借用检查只是编译器对代码中引用的静态分析。生命周期（lifetime）信息早就在 中级中间语言（MIR） 生成前完全抽离了。

Rust 中不使用传统的异常处理，而是使用基于返回值的错误处理。但你也可以使用 恐慌（Panic）来处理像 Cpp 中那样的异常行为。它可以在编译时被禁用（panic = abort），但即便如此，Rust 也不喜欢 与 Cpp 异常 或 longjmp 混在一起。

### **同样的 LLVM 后端**

Rust 与 LLVM 有很好的整合，所以它支持链接时间优化，包括 ThinLTO，甚至是跨越 C/C++/Rust 语言边界的内联。也有按配置优化（Profile-guided Optimization，PGO）的支持。尽管 rustc 比 clang 生成的 LLVM IR 更加冗长，但优化器仍然能够很好地处理它。

C 语言用 GCC 编译比用 LLVM 更快，现在 Rust 社区也有人在开发 GCC 的 Rust 前端。

理论上，因为 Rust 有比 C 更严格的不可变和别名规则，应该比 C 语言有更好的性能优化，但实际上并没有起到这样的效果。目前在 LLVM 中，超越 C 语言的优化是一项正在进行的工作，所以 Rust 仍然没有达到其全部潜力。

### **都允许手工优化，但有一些小例外**

Rust 的代码足够底层和可预测，可以通过手工调整它的优化到什么样的汇编代码。

Rust 支持 SIMD ，对内联和调用约定有很好的控制。

Rust 和 C 语言足够相似，C 语言的一些分析工具通常可以用于 Rust 。

总的来说，如果性能绝对关键，并且需要手工优化压榨最后一点性能，那么优化 Rus t 与 优化 C 语言并没有什么不同。

但是在一些比较底层的特性，Rust 没有特别好的替代方法。

* goto。Rust 中没有提供goto，不过你可以使用循环的 break 标签来代替。C 语言中一般使用 goto 来清理内存，但是 Rust 因为有确定性析构功能，所以不需要 goto。然而有一个 非标准的 goto 扩展，对性能优化比较有用。
* 栈内存分配alloca和C99可变长度数组，可以节省内存空间，减少内存分配次数。但这些即使在 C 语言中也是有争议的，所以 Rust 远离了它们。

### **Rust 相比 C 语言的一些开销**

如果没有经过手工优化，Rust 因为其抽象表达也会有一些开销。

* Rust 缺乏隐式类型转换和只用 usize 的索引，这导致开发者只能使用这种类型，哪怕只需要更小的数据类型。64 位平台上用 usize 做索引更容易优化，而不需要担心未定义行为，但多余的 bit 位可能会给寄存器和内存带来更大的压力。而在 C 中，你可以选择 32 位类型。
* Rust 中的字符串，总是会携带指针和长度。但是很多 C 代码中的函数只接收指针而不管大小。
* 像 for i in 0...len {arr[i]} 这样的迭代，性能取决于 LLVM 优化器能否证明长度匹配。有时候，它不能，并且边界检查也会抑制自动矢量化。
* C 语言比较自由，对于内存有很多“聪明”的使用技巧，但在 Rust 里就没这么自由了。但 Rust 仍然给了内存分配很多控制权，并且可以做一些基本的事情，比如内存池、将多个分配合并为一个、预分配空间等等。
* 在不熟悉 Rust 借用检查的情况下，可能会用 Clone 来逃避使用引用。
* Rust 的标准库中 I/O 是不带缓存的，所以需要使用 **BufWriter** 来包装。这就是为什么有些人说 Rust 写的代码还不如 Python 快的原因，因为 99% 的时间都用在 I/O 上了。

### **[可执行文件](https://www.zhihu.com/search?q=%E5%8F%AF%E6%89%A7%E8%A1%8C%E6%96%87%E4%BB%B6&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":2222924757}" \t "_blank)大小**

每个操作系统都有一些内置的标准 C 库，其中有大约 30MB 的代码。C 语言的执行文件，可以“免费”使用这些库。

一个小的 "Hello World " 级 C 可执行文件实际上不能打印任何东西，它只调用操作系统提供的 printf。

而 Rust 则不可以，Rust 可执行文件会捆绑自己的标准库（300KB 或更多）。幸运的是，这只是一次性的开销，可以减少。

对于[嵌入式开发](https://www.zhihu.com/search?q=%E5%B5%8C%E5%85%A5%E5%BC%8F%E5%BC%80%E5%8F%91&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":2222924757}" \t "_blank)，可以关闭标准库，使用 "no-std"，Rust 将生成 "裸 "代码。

在每个函数的基础上，Rust 代码的大小与 C 差不多，但有一个 "泛型膨胀 "的问题。泛型函数为它们所使用的每一种类型都有优化的版本，所以有可能出现同一个函数有 8 个版本的情况，[cargo-bloat](https://link.zhihu.com/?target=https://lib.rs/cargo-bloat" \t "_blank) 库有助于发现这些问题。

在 Rust 中使用依赖关系是非常容易的。与 JS/npm 类似，现在推荐使用小型且单用途的包，但它们确实在不断增加。cargo-tree 命令对于删减它们非常有用。

### **Rust 略胜 C 的一些地方**

1. 为了隐藏实现细节，C 库经常返回不透明的数据结构指针，并确保结构的每个实例只有一个副本。它会消耗堆分配和指针间接寻址的成本。Rust 内置的隐私、单一所有权规则和编码惯例允许库暴露其对象，而不需要间接性，这样，调用者可以决定将其放入堆（heap）上还是栈（stack）中。可以主动或彻底地优化栈上的对象。
2. 缺省情况下，Rust 可以将来自标准库、依赖项和其他[编译单元](https://www.zhihu.com/search?q=%E7%BC%96%E8%AF%91%E5%8D%95%E5%85%83&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":2222924757}" \t "_blank)的函数内联。
3. Rust 会对结构体字段进行重排，以优化内存布局。
4. 字符串携带大小信息，使得长度检查速度很快。并允许就地生成子串。
5. 与 C++ 模板类似，Rust 中泛型函数会单态化，生成不同类型的副本，因此像 sort 这样的函数和 HashMap 这样的容器总是针对相应的类型进行优化。对于 C 语言，则必须在修改宏或者处理void\*和运行时变量大小的效率较低的函数之间做出选择。
6. Rust 的迭代器可以组合成链状，作为一个单元一起被优化。因此，你可以调用it.buy().use().break().change().mail().upgrade()，而不是对同一个缓存区多次写入的一系列调用。
7. 同样，通过 Read 和 Write 接口，接收一些未缓存的流数据，在流中执行 CRC 校验，然后将其转码、压缩，再写入网络中，所有这些都可以在一次调用中完成。虽然 C 语言中应该也可以做到，但它没有泛型和特质（trait），将很难做到。
8. Rust 标准库中内置高质量的容器和优化过的数据结构，比 C 使用起来更方便。
9. Rust 的 [serde](https://www.zhihu.com/search?q=serde&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":2222924757}" \t "_blank) 是世界上最快的 JSON 解析器之一，使用体验非常棒。

### **Rust 比 C 明显优越的地方**

主要是两点：

1. Rust 消除数据竞争，天生线程安全，解放多线程生产力，是 Rust 明显比 C / Cpp 等语言优越的地方。
2. Rust 语言支持异步高并发编程。
3. Rust 支持 安全的编译期计算。

**线程安全**

即使是在第三方库中，Rust 也会强制实现所有代码和数据的线程安全，哪怕那些代码的作者没有注意线程安全。一切都遵循一个特定的线程安全保证，或者不允许跨线程使用。当你编写的代码不符合线程安全时，编译器会准确地指出不安全之处。

Rust 生态中已经有了很多库，如数据并行、线程池、队列、任务、无锁数据结构等。有了这类组件的帮助，再加上类型系统强大的安全网，完全可以很轻松地实现并发/并行化 Rust 程序。有些情况下，用 par\_iter 代替 iter 是可以的，只要能够进行编译，就可以正常工作！这并不总是线性加速（ 阿姆达尔定律（Amdahl's law）很残酷），但往往是相对较少的工作就能加速 2~3 倍。

延伸：[阿姆达尔定律](https://www.zhihu.com/search?q=%E9%98%BF%E5%A7%86%E8%BE%BE%E5%B0%94%E5%AE%9A%E5%BE%8B&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":2222924757}" \t "_blank)，一个计算机科学界的经验法则，因 Gene Amdahl 而得名。它代表了处理器并行计算之后效率提升的能力。

在记录线程安全方面，Rust 和 C 有一个有趣的不同。

Rust 有一个术语表用于描述线程安全的特定方面，如 Send 和 Sync、guards 和 cell。

对于 C 库，没有这样的说法：“可以在一个线程上分配它，在另一个线程上释放它，但不能同时从两个线程中使用它”。

根据数据类型，Rust 描述了线程安全性，它可以泛化到所有使用它们的函数。

对于 C 语言来说，线程安全只涉及单个函数和配置标志。

Rust 的保证通常是在编译时提供的，至少是无条件的。

对于 C 语言，常见的是“仅当 turboblub 选项设置为 7 时，这才是线程安全的” （这句话并不是真的让你去设置 turboblub 选项，因为本来就没有这个选项，所以 C 语言不会保证你线程安全 ）。

**异步并发**

Rust 语言支持 async/await异步编程模型。

该编程模型，基于一个叫做 Future 的概念，，在 JavaScript 中也叫做 Promise。Future 表示一个尚未得出的值，你可以在它被解决（resolved）以得出那个值之前对它进行各种操作。在许多语言中，对 Future 所做的工作并不多，这种实现支持很多特性比如组合器（Combinator），尤其是能在此基础上实现更符合人体工程学的 async/await 语法。

Future 可以表示各种各样的东西，尤其适用于表示异步 I/O ：当你发起一次网络请求时，你将立即获得一个 Future 对象，而一旦网络请求完成，它将返回任何响应可能包含的值；你也可以表示诸如“超时”之类的东西，“超时”其实就是一个在过了特定时间后被解决的 Future ；甚至不属于 I/O 的工作或者需要放到某个线程池中运行的 CPU 密集型的工作，也可以通过一个 Future 来表示，这个 Future 将会在线程池完成工作后被解决。

**Future** **存在的问题** 是它在大多数语言中的表示方式是这种基于回调的方法，使用这种方式时，你可以指定在 Future 被解决之后运行什么回调函数。也就是说， Future 负责弄清楚什么时候被解决，无论你的回调是什么，它都会运行；而所有的不便也都建立在此模型上，它非常难用，因为已经有很多开发者进行了大量的尝试，发现他们不得不写很多分配性的代码以及使用动态派发；实际上，你尝试调度的每个回调都必须获得自己独立的存储空间，例如 crate 对象、堆内存分配，这些分配以及动态派发无处不在。这种方法没有满足零成本抽象的第二个原则，如果你要使用它，它将比你自己写要慢很多，那你为什么还要用它。

Rust 中的方案有所不同。不是由 Future 来调度回调函数，而是由一个被称为执行器（executor）的组件去轮询 Future。而 Future 可能返回“尚未准备就绪（Pending）”，也可能被解决就返回“已就绪（Ready）”。该模型有很多优点。其中一个优点是，你可以非常容易地取消 Future ，因为取消 Future 只需要停止持有 Future。而如果采用基于回调的方法，要通过调度来取消并使其停止就没这么容易了。

同时它还能够使我们在程序的不同部分之间建立真正清晰的抽象边界，大多数其他 Future 库都带有事件循环（[event loop](https://www.zhihu.com/search?q=event loop&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":2222924757}" \t "_blank)），这也是调度 你的Future 执行 I/O 的方法，但实际上你对此没有任何控制权。

而在 Rust 中，各组件之间的边界非常整洁，执行器（executor）负责调度你的 Future ，反应器（reactor）处理所有的 I/O ，然后是你的实际代码。因此最终用户可以自行决定使用什么执行器，使用他们想使用的反应器，从而获得更强的控制力，这在系统编程语言中真的很重要。

而此模型最重要的真正优势在于，它使我们能够以一种真正零成本的完美方式实现这种状态机式的 Future 。也就是当你编写的 Future 代码被编译成实际的本地（native）代码时，它就像一个状态机；在该状态机中，每次 I/O 的暂停点都有一个变体（[variant](https://www.zhihu.com/search?q=variant&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":2222924757}" \t "_blank)），而每个变体都保存了恢复执行所需的状态。

而这种 Future 抽象的真正有用之处在于，我们可以在其之上构建其他 API 。可以通过将这些组合器方法应用于 Future 来构建状态机，它们的工作方式类似于迭代器（Iterator）的适配器（如 filter、map）。但是这种方式是有一些缺点的，尤其是诸如嵌套回调之类，可读性非常差。所以才需要实现 async / await异步语法。

目前 Rust 生态中，已经有了成熟的 [tokio](https://link.zhihu.com/?target=https://tokio.rs/" \t "_blank) 运行时生态，支持 epoll 等异步 I/O。如果你想用 io\_uring ，也可以使用 [Glommio](https://link.zhihu.com/?target=https://github.com/DataDog/glommio" \t "_blank) ，或者等待 tokio 对 io\_uring 的支持。甚至，你可以使用 smol 运行时提供的 [async\_executor](https://link.zhihu.com/?target=https://github.com/smol-rs/async-executor" \t "_blank) 和 [async-io](https://link.zhihu.com/?target=https://github.com/smol-rs/async-io" \t "_blank) 来构建你自己的运行时。

**编译期计算**

Rust 可以支持类似于 Cpp 那样的 编译期常量求值。这一点是明显比 C 优越的。

虽然目前功能还不如 Cpp 那样强大，但还在不断的维护中。

为什么 Rust 中支持 编译期计算这么谨慎呢？因为 Rust [编译期求值](https://www.zhihu.com/search?q=%E7%BC%96%E8%AF%91%E6%9C%9F%E6%B1%82%E5%80%BC&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":2222924757}" \t "_blank)是必须要保证安全的，所以有很多考虑。Rust 编译期求值不像 Cpp 那样自由且容易滥用。

作者：盼盼编程  
链接：https://www.zhihu.com/question/496992250/answer/2222924757  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。