第 1 章
Rust 基础

在你决定是否要学习一门新语言 Rust 的时候，先看一个新闻：2022 年的调查结果显示，Android 的安全漏洞从 2019 年的 223 个降低到 2022 年的 85 个，经过分析，谷歌认为内存漏洞减少的情况主要与 Rust 代码的比例增加有关。

Rust 语言考虑了内存的安全性。在编译的时候，Rust 就能够捕捉到大多数的内存安全问题，避免相关漏洞在生产环境中出现。

在 Android 13 中，已经有约 21%的新原生代码以 Rust 开发，官方提到，这些组件大多数是在用户层面的系统服务（即 Linux 中运行），但目前还有许多组件依然使用 C++编写，而其中许多安全关键组件都在 Linux 核心之外的裸机环境中运行，当下谷歌为了强化 Android 设备的安全性，正逐渐提高在裸机环境使用 Rust 的比例。

1.1 Rust概述

1.1.1 Rust 的来源与定义

1. Rust的来源

Rust语言一开始在2006年作为Mozilla员工Graydon Hoare的私人项目出现，而Mozilla于2009年开始赞助这个项目。第一个有版本号的Rust编译器于2012年1月发布。Rust 1.0是第一个稳定版本，于2015年5月15日发布。

Graydon Hoare是一个职业编程语言工程师，其日常工作就是给其他语言开发编译器和工具集，但是不会参与语言本身的设计，由于这种工作性质，他接触过很多编程语言，了解各种语言的优缺点。比如C和C++，性能比较好，但是类型系统和内存都不太安全；一些拥有垃圾回收（Garbage Collection，GC）的语言，比如Java、Golang、Kotlin等，内存是安全的，但是性能却比较低。于是Graydon Hoare萌生了自己开发一门语言的想法，这门语言就是Rust。

Rust的LOGO如图1-1所示。Rust的LOGO承载了创造者对该语言的期望：

图 1-1

（1）Rust这个单词是由Trust和Robust组合而成的，暗示了信任（Trust）和鲁棒性（或健壮性，Robust）。

（2）Rust LOGO的形状与一种叫作锈菌的真菌相似，这种真菌生命力非常顽强，在其生命周期内可产生多达5种孢子类型，这5种生命形态还可以相互转换，也就是健壮性非常强。其LOGO上面的5个小圆孔与锈菌的5种生命形态相对应，暗示Rust语言超强的健壮性。

2. Rust的定义

Rust是一门系统编程语言，专注于安全，尤其是并发安全，是支持函数式、命令式以及泛型等编程范式的多范式语言。Rust在语法上和C++类似，设计者想要在保证性能的同时提供更好的内存安全。

Rust最初是由Mozilla研究院的Graydon Hoare设计创造的，然后在Dave Herman、Brendan Eich以及其他人的贡献下逐步完善。Rust的设计者们通过在研发Servo网站浏览器布局引擎的过程中积累的经验，优化了Rust语言和Rust编译器。

Rust编译器是在MIT License和Apache License 2.0双重协议声明下的免费开源软件。Rust已经连续7年（2016~2022年）在Stack Overflow开发者调查的“最受喜爱编程语言”评选项目中摘取桂冠。

1.1.2 Rust 适合做什么

Rust语言适合做的事情包括以下8个方面：防止数据泄露、数据分析、游戏开发、机器学习、嵌入式设备的开发、网络服务器的开发、编译成 WebAssembly、直接生成目标可执行程序。

1. 防止数据泄露

Rust已经是一种用于生产环境的成熟技术。作为一种系统编程语言，它允许用户保持对低级细节的控制。用户可以选择将数据存储在堆栈上（用于静态内存分配），还是存储在堆上（用于动态内存分配）。在这里，着重提一下RAII技术。RAII也称为“资源获取就是初始化”，是C++等编程语言常用的管理资源、避免内存泄露的方法。它保证在任何情况下，使用对象时先构造对象，再析构对象，这是一个主要与C++相关的代码习语，但该技术现在也存在于Rust中，即每次对象超出范围时，都会调用其析构函数并释放其拥有的资源，程序员不必手动执行此操作，并且可以防止资源泄露错误。

2. 数据分析

高性能和安全性对使用Rust来执行大量数据分析的科学家具有强烈的吸引力。Rust的速度非常快，使其成为计算生物学和机器学习的理想选择，在这些领域的应用中，用户需要非常快速地处理大量数据。

3. 游戏开发

Rust是一种面向性能的语言，它可以通过适当的内存管理有效地执行复杂的任务。此外，Rust不使用垃圾回收器，这是最优化的游戏性能的加分项。

4. 机器学习

Rust预计将在机器学习（Machine Language，ML）领域大放异彩，因为它的低级内存控制。该语言使用高级抽象，这些抽象在构建基于Rust的神经网络时非常有益。Rust具有创建现代算法的巨大潜力，但它仍然远不及其他机器学习语言。程序员目前正在尝试Rust，该语言仍然需要一些时间来成熟到足以创建机器学习算法，就像我们在Python中所做的那样。目前正在创建新的Rust库来开发可靠的神经网络，但这需要一些时间。

5. 嵌入式设备的开发

Rust是一种低级编程语言，可直接访问硬件和内存，这使其成为嵌入式和裸机开发的绝佳解决方案。用户可以使用Rust编写操作系统或微控制器应用程序。事实上，有许多用Rust编写的操作系统，例如vivo自主研发的“蓝河操作系统”（BlueOS）、BlogOS、RustOS、QuiltOS、intermezzOS等。Rust也被用于浏览器（如Mozilla Firefox）、游戏等方面。不少开发者视 Rust 为一种更具创新性的系统级语言，因为它不允许悬空指针或空指针。它是为了在不影响性能和速度的前提下做到安全、可靠而创建的。

6. 网络服务器的开发

Rust用极低的资源消耗做到安全高效，且具备很强的大规模并发处理能力，十分适合开发普通或极端的服务器程序，可以用于开发网络服务器。

7. 编译成WebAssembly

Rust可以被编译成WebAssembly，WebAssembly是一种JavaScript的高效替代品。

8. 直接生成目标可执行程序

Rust编译器可以直接生成目标可执行程序，不需要任何解释程序，可用于传统命令行程序。

1.1.3　Rust 的特点

Rust是一门系统级编程语言，它有如下特点。

1. 类C的语言语法

Rust的具体语法和C/C++类似，都是由花括号限定代码块，还有一样的控制流关键字，例如if、else、while和for。然而，也并非所有的C或者C++关键字都被实现了。尽管与C/C++极其相似，Rust在深层语法上跟元语言家族的语言，比如Haskell（一种通用的纯函数编程语言）更接近。基本上一个函数体的每个部分都是表达式，甚至是控制流操作符。例如，普通的if表达式就取代了C的三元表达式。一个函数不需要以return表达式结束，在这种情况下函数最后的表达式就是返回值。

2. 内存安全

Rust语言系统设计用于保证内存安全，它在安全代码中不允许空指针、悬垂指针和数据竞争。数值只能用一系列固定形式来初始化，要求所有输入已经被初始化。在其他语言中复制函数指针要

么有效、要么为空，比如在链表和二叉树等数据结构中，Rust核心库提供Option类型，用来测试指针是否有值。Rust同时引入添加语法来管理生命周期，而且编译器通过租借检查器来说明相关理由。

3. 高效的内存管理

Rust不像Go、Java以及.NET Framework那样使用自动垃圾回收系统。不同的是Rust通过RAII来管理内存和资源，还可以选用引用计数。Rust以低开销提供资源确定性管理。Rust也支持值的栈分配，并且不表现暗箱。Rust中也有引用概念（用&符号），不包含运行时引用计数，租约检查器编译时已经验证了此类指针的安全性，阻止了悬空指针和其他形式的未定义行为。

4. 引进所有权

所有权（系统）是Rust最为与众不同的特性，对语言的其他部分有着深刻含义。它让Rust无须进行垃圾回收即可保障内存安全，因此理解Rust中的所有权如何工作是十分重要的。所有程序都必须管理其运行时使用计算机内存的方式，有些语言具有垃圾回收机制，在程序运行时有规律地寻找不再使用的内存，而有些语言程序员必须亲自分配和释放内存。

C/C++这样的语言主要通过手动方式管理内存，开发者需要手动申请和释放内存资源。但为了提高开发效率，只要不影响程序功能的实现，许多开发者没有及时释放内存的习惯。所以手动管理内存的方式常常造成资源浪费。

Java语言编写的程序在虚拟机（Java Virtual Machine，JVM）中运行，JVM具备自动回收内存资源的功能。但这种方式常常会降低运行时效率，所以JVM会尽可能少地回收资源，这样也会使程序占用较多的内存资源。

Rust则选择了第三种方式，通过所有权系统管理内存，编译器在编译时会根据一系列的规则进行检查。违反任何规则，程序都不能编译。在运行时，所有权系统的任何功能都不会减慢程序运行速度。

Rust有一个所有权系统，所有的值都有一个唯一的属主，值的有效范围跟属主的有效范围一样。Rust中的每一个值都有一个所有者，值在任一时刻有且只有一个所有者，所有者（变量）离开作用域，这个值将被丢弃。在任何时候，要么有多个不可变引用，要么只有一个可变引用。Rust编译器在编译时执行这些规则，同时检查所有引用的有效性。

5. 类型多态

Rust的类型系统支持一种类似类型类的机制，叫traits，是被Haskell语言激发灵感产生的。这是一种用于特定同质法的设施，通过给类型变量声明添加约束来实现。其他来自Haskell的特性，如更高类型多态还没有支持。

1.1.4 Rust 和其他语言的总体比较

Java、C、Python都是功能强大的编程语言，为什么人们还要设计出一个Rust？笔者当初也很困惑，我们首先将其和其他优秀语言做一下总体比较。

1. Rust与C++比较

Rust和C/C++相比肯定是稍显年轻，最初的开发者只有一位，就是Graydon Hoare，之后得到了Mozilla的赞助。Rust的语法与C++相似，它能提供更高的速度和更好的内存安全，不用自动垃圾回收，也无须手动释放。

在安全的内存管理方面，不少开发者把Rust当作一种更具有创新性的系统级语言，因为它不允许悬空指针或者空指针。

在外媒The Register的文章中写道：或许我们总是可以写出完美安全的C/C++代码，只是大多数情况下这不是一件容易的事情。因为这两种语言都太容易造成内存错误了，比如无效的栈和堆内存访问、内存泄露、不匹配的内存分配和反分配、未初始化的内存访问。

2. Rust与Java比较

对于开发者而言，完美的资源分配和良好的内存管理是Rust突出的优点。使用Rust可以轻易尝试各种类型新颖的复杂项目，之前由于Java语言的复杂性，用户不敢轻易尝试的都可以用Rust尝试。

3. Rust与Python比较

Rust超越Python的一个主要原因是性能。因为Rust是直接编译成机器代码的，所以在代码和计算机之间没有虚拟机或解释器。

与Python相比，另一个关键优势是Rust的线程和内存管理。虽然Rust不像Python那样有垃圾回收功能，但Rust中的编译器会强制检查无效的内存引用泄露和其他危险或不规则行为。

编译语言通常比解释语言要快。但是，使Rust处于不同水平的是，它几乎与C和C++一样快，但是没有开销。

1.2　Rust到底值不值得学

Rust是近两年呼声比较高的一种新型开发语言，市场占有量并不大，但增长速度极为迅猛。

有人统计过，在计算机行业，平均每33.5天就有一种所谓的新型开发语言面世，这还不包括很多企业内部、项目内部的内置简易流程工具。然而大浪淘沙，如今仍然占据着市场地位的，仍然是耳熟能详的有限几种。

作为新来的搅局者，Rust到底值不值得学习并且在工作中应用呢？先说结论，这里粗略地把开发者分为初学者、小有经验的常规工程师和资深开发者三类。

对于初学者，Rust具有比较陡峭的学习曲线，虽然学习Rust能训练良好的编程习惯，从长远来看对提高学习者的开发素养极具价值。但短期的大量付出很容易让初学者心力交瘁。并且尽管官方文档并不欠缺，但学习资料对于初学者来说仍然远远不够，比较而言得不偿失。因此，建议初学者仍然使用久经验证的语言入门加入软件开发的大家庭，比如C、Java、Python、JavaScript都是很好的入门选择。

对于有一定经验的常规工程师，他们已经有了一段时间的开发工作实践，对于软件开发的现状、发展都已经形成了自己的世界观。如果感觉不是很喜欢这个行业，希望将来转行管理岗位或者

产品岗位。那么当前应当倾向于业务领域，了解业务和技术的衔接和互动，完全不需要学习Rust。而如果醉心于技术，并从中获得了自己的乐趣，希望逐步提高自己的技术水平，那么Rust会是一个很好的桥梁，哪怕仅仅学习Rust而并不将其应用于工作，也能让开发者从中获取大量的有益习惯和软件底层经验，从而形成自己良好的代码风格。

对于资深工程师，即便并不从事底层系统级的开发工作，Rust也是一门很优秀的语言。它能弥补当前多种开发语言的不足，形成良好的开发哲学和思想导向，帮助开发者交付高质量的软件产品。因此，及早学习并应用Rust非常有价值。

为了说明这个结论，下面从多个角度，采用同传统语言对比的方式来说一说笔者对Rust的理解。

1.2.1 Rust 是一种全面创新的语言

值不值得学笔者说了不算，我们要评估这门新语言的特点。这几年出现了不少有影响力的语言出现，但大多数都只是关键字或者小范围的语法创新，随后可能会有大量的特色库函数来丰富语言的功能。一个有经验的开发者，可能翻两天资料，就能快速掌握。

而Rust极具自身语言特点，是一种完全创新的语言，而不是简单的语法替换。简单地熟悉几个关键字、判断、循环等语法，远不足以掌握这门语言。

为了证明这一点，下面用Rust的“所有权”（Ownership）机制和“遮蔽”（Shadowing）机制来举例说明。以C++为例，请看下面这段代码：

```
#include using namespace std;
int main(){
    string s1="hello";
    string s2=s1;
    cout << "s1=" << s1 << ",s2=" << s2 << endl;
    return 0;
}
```

编译执行后，程序输出：

```
s1=hello,s2=hello
```

代码再简单不过，首先声明、赋值一个字符串变量s1，然后把变量s1赋值给变量s2，最后输出两者的值。对应地，我们看一个Rust的版本：

```
fn main(){
    let s1=String::from("hello");
    let s2=s1;
    println!("s1={},s2={}",s1,s2);
}
```

除细小的语法差异外，看上去跟C++版本没有什么不同。然而在Rust中，这段代码连编译都无法通过，得益于rustc编译程序详细的输出，我们能看到很细致的错误提示：

```
    2 | let s1=String::from("hello"); | -- move occurs because `s1` has type
`std::string::String`, which does not implement the `Copy` trait 3 | let s2=s1; | --
value moved here 4 | println!("s1={},s2={}",s1,s2); | ^^ value borrowed here after move
```

这个编译错误是指，上面的代码中，当变量s1赋值给s2之后，s1变量名所指向的内存所有权被

“转移”（move）到了s2变量名之下。从此之后，s1变量名就无效了，不再指向任何一块内存。除非重新声明并为s1赋值（Rust中称为Shadow，“遮蔽”原有的s1），s1不能再被使用。

所有权机制可以有效防止内存泄露所导致的程序Bug，是Rust内存管理的核心理念。上面提到的所有权“转移”是所有权管理的重要特征之一。

“遮蔽”也是一个有趣的概念，Rust的处理方式跟很多我们熟悉的语言不同。

请看下面的C语言代码：

```
#include int main(){
    int x = 5;
    x = x+1;
    printf("x=%d\n", x);
}
```

这又是一段很基本的代码。首先声明、赋值一个整数变量x，接着把x的值加1，再赋值回变量x。这是各种开发语言中都常见的用法。编译执行的输出结果为x=6。

下面来看Rust的版本：

```
fn main(){
    let x=5;
    x = x+1;
    println!("x={}", x);
}
```

很不幸，这段代码同样无法编译通过，错误是：

```
    error[E0384]: cannot assign twice to immutable variable `x` --> test-own1.rs:3:5
| 2 | let x=5; | - | | | first assignment to `x` | help: make this binding mutable:
`mut x` 3 | x = x+1; | ^^^^^^^ cannot assign twice to immutable variable
```

编译器rustc这种“图示”型的输出信息让你排查错误更加方便。错误的原因在于，在Rust中，默认所有变量都是只读类型的，除非在变量声明的时候就注明为可变类型mut。因此，两次对于一个只读变量赋值导致编译错误。解决的办法是，要么注明变量为可读写，这样与C语言版本具有完全相同的意义：

```
let mut x=5;
```

要么用前面提到过的“遮蔽”机制：

```
fn main(){
    let x=5;
    let x = x+1;
    println!("x={}", x);
}
```

注意，在上面的x=x+1这一行的开始再次使用let关键字，表示再次声明了变量x。

与大多数语言不允许重复声明变量不同，这个变量x跟第一次声明的变量x同名，并对其做出了“遮蔽”。之后除非再次遮蔽变量x，那么起作用的都将是本次新声明的x。

通过这两个例子可以看出，Rust是从理念上做出了大量创新的一种语言。如果只是像学习其他语言一样对比学习语法和关键字，无法真正掌握这门语言。这些融汇在语言中的理念才是Rust最宝

贵的地方。注意，在这里“理念”可不是什么大而化之的套话，而是实际操作中很重要的原则。

很多语言的设计初衷是“简化”，在Rust中当然也有很多简化的地方，就像直接使用let关键字声明一个变量，而变量的类型可以通过赋值的操作从而推导出变量的类型。比如变量超出作用域，也会被自动回收。但Rust中也大量存在“复杂化”的操作，比如上面举例的所有权机制，再比如使用可读写变量需要额外标注mut。这些“复杂化”的部分，都基于“尽量在程序开发的早期，就将可能会出现问题的部分暴露出来，从而在设计中和编译时就解决掉”这样一个理念。

1.2.2 引用和借用

如果每次都发生所有权的转移，程序的编写就会变得异常复杂。因此，Rust和其他编程语言类似，提供了引用（References）的方式来操作。获取变量的引用称为借用。类似于你借别人的东西来使用，但是这个东西的所有者不是你。引用不会发生所有权的转移。引用类似于C语言中的指针，指向一块已经存在的数据：

```
let mut x = 5;
let y = &x;
```

上例中，y是对变量x的引用，并且没有标注mut，所以是只读引用。写法跟C语言中获取指针的方式类似，就是一个&符号。y此时具有了变量x的一些权限，所以也称为“借用”，本例中因为只借用了读的功能，没有借用写的功能，所以称“一些”。当然也可以借用写的功能，我们后面会再举例。

借用（Borrowing）看起来跟引用是一回事，但“借用”这个词更主要对应的是前面所说的所有权“转移”的概念，转移之后，原来的变量就无效了。而借用之后，原来的变量还有效，或者部分有效，比如只被借用了写权限。在函数参数中，使用引用的方式让函数临时获得数据的访问权，也是典型的借用。事实上，这种方式才是最常用到借用的地方：

```
fn main() {
    fn sum_vec(v: &Vec) -> i32
    { return v.iter().fold(0, |a, &b| a + b); }
    let v1 = vec![1, 2, 3];
    let s1 = sum_vec(v1);
    println!("{}", s1);
}
```

先别管我们使用到的令人困惑的关键字和函数名，那些系统学习之后都不算什么。在函数sum_vec的参数中，我们就使用了借用。顺便还见识了Rust中函数的嵌套写法，当然现在新兴的语言，包括C++11之后的版本，都已经支持这种写法，这在函数式（Functional Programming Paradigm，注意不是函数化Functionalization）编程中是很重要的支持。

引用和借用的概念与C/C++语言中所使用的非常类似，尽管名称不同，主要的区别在于对引用的管理理念，Rust对引用的管理规则如下：

（1）对于一块内存，同时只能有一个可写引用存在。

（2）对于一块内存，同时可以有多个只读引用存在。

（3）对于一块内存，在有一个可写引用存在的时候，不能有其他引用存在，无论只读或者可写。

引用的原始对象必须在引用存在的生命期一直有效，比如：

```
let mut x = 5;
let y = &mut x;
let z = &mut x;
println!("{} {} {}", x,y,z);
```

上面的代码会产生编译错误，因为y已经是可写的引用，而同时存在一个可写的引用z，违反了Rust对引用的管理规则。如果把z变量这一行和后面显示z的部分去掉呢？去掉之后是可以编译通过的，但仍然需要注意，y此时是可写的指针，“借用”了x的写权限。所以x此时只有读的权限，不能再对x进行赋值。因为它已经被“借用走”（Borrowed）了。

这些复杂的规则看起来就跟前面见过的所有权转移一样，似乎极大地限制了程序员的自由度。但这些都是在强迫你，让你成为一位更优秀的程序员，产出更高质量的代码，将Bug消灭在萌芽期。

1.2.3　生命期

通常一个变量的生命期（Lifetime）就是它的作用域。但在引用和借用出现后，这个问题变得复杂了。熟悉C语言的程序员都碰到过数据失效了而指针依然存在的情况，俗称“悬挂指针”。Java为了解决这个问题，干脆取消了指针，并且最终以引用计数器作为内存管理的主要模式。这种情况出现最多的场景，是在某个函数中使用了变量或者申请了内存，并将其引用作为返回值传递到调用者的时候。比如这段C语言代码：

```
int *getSomeData(){
    int c=32767;
    return &c;
}
```

变量c位于栈上，是一个局部变量，当函数返回指针的时候，指针在这个函数的调用者中依然存在，但变量c已经被回收了。在新版本的编译器中，这种情况也会被警告，但可以编译成功。而在Rust中，这种情况是不允许编译通过的，比如下面的类似代码：

```
fn somestr() -> &str {
    let result = String::from("a demo string");
    result.as_str()
}
```

直接使用方法返回值（或者变量），之后没有分号，即将其作为返回值处理，不用像C语言一样要使用return result.as_str()语句返回值。编译的时候会报错“result变量没有足够长的生命期”：

```
    error[E0597]: `result` does not live long enough --> src/main.rs:3:5 | 3 |
result.as_str() | ^^^^^^ does not live long enough 4 | } | - borrowed value only lives
until here
```

如果仅仅是这样断然地禁止返回悬挂引用，也就“不过如此”了。事实上，更复杂的问题在于，如果数据源来自函数的参数，参数本身就是引用的情况。比如下面的Rust代码：

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() { x }
    else { y }
}
```

上面这个函数接受两个字符串的引用，比较其长度，将长的那个字符串作为结果返回调用者。这种返回值的方式一定让你印象深刻。虽然示例简单，但不可否认，这种需求是很正当的。大量的应用场景都需要函数独立于外，处理固定的内存数据，进入和返回的都只是指向内存的指针。当然，尽管合理，但是上面的代码是无法编译通过的，报错是“丢失生命期指定”：

```
error[E0106]: missing lifetime specifier --> src/main.rs:1:33 | 1 | fn longest(x: &str, y: &str) -> &str { | ^ expected lifetime parameter | = help: this function's return type contains a borrowed value, but the signature does not say whether it is borrowed from `x` or `y`
```

Rust引入了生命期的概念，从而保证返回值与给定的参数具有相同的生命期。这既保证了程序的灵活性，又不会造成内存泄露，同时还不会把维护内存安全的责任推给不可靠的人为因素。

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x }
    else { y }
}
```

上面的代码添加了生命期指定。在函数名之后首先声明了生命期a，语法样式跟泛型的类型说明部分实际是一样的，都放在尖括号“<>”中。生命期名称之前附加一个单引号“'”。随后的两个引用参数x/y以及作为返回值的字符串引用都直接在&符号之后标注了生命期'a。这表示，这几个引用具有相同的生命期。

当然，在这个例子中，x/y是调用的参数，是外面传递进来的，所以完整的含义应当是：返回的引用值，同参数x/y一样具有相同的生命期。因此，从调用者的角度来看，当x/y指向的内存超出作用域销毁之后，所获得的函数返回值也同时被销毁。

有一个特殊的生命期'static，用于表示Rust中的全局量或者静态量，专门表示这种引用具有贯穿于整个程序运行时的生命期长度。比如，Rust中通常用字面量赋值的字符串实际都是'static，因为这些字面量实际在编译程序的时候，就放置到了数据区并一直存在，贯穿程序始终：

```
let s = "I have a static lifetime.";
```

1.2.4 编译时检查和运行时开销

通过前面的几个例子，我们对Rust的编译器rustc有了一个初步了解，丰富、详尽的编译错误输出对于排查源码中的错误帮助很大。实际上远不止于此。Rust的编译器包含着Rust语言的另一个核心思想，那就是尽量在编译阶段就暴露出程序的设计错误，而不让这些错误带到生产环境中，从而付出昂贵的代价。

这也是Rust学习曲线陡峭的原因之一，很多在其他语言中可以编译通过的代码，在Rust中都无法编译通过（排除语法错误）。这种更严格的编译时检查，很容易让初学者手足无措。

带来的优点也是显而易见的，除刚才提过的不让程序Bug带入生产环境外，错误能在编译阶段

就消除掉，无须在运行时进行更多不必要的错误检查，这也将大大地减少程序在运行时的消耗。这个消耗包括编译所生成的代码体积和运行时检查所损耗的CPU资源两个方面。比如，Rust中有多种不同功能的智能指针，以常见的Box和Rc为例，前者提供基本的指针功能，后者提供类似Java语言一样，基于引用统计的自动垃圾回收机制。（请注意，这里并不是做语言学习，所以请关注在Rust的设计理念上，先别在意具体的关键字和语法。）

如果在程序中使用Box指针的话，当变量x被赋值给变量y，所有权同时被转移，变量x就不再可用了，这个在开始介绍所有权时就见到了：

```
let x = Box::new(1);
let y = x;                          // x从此无效了
```

与此规则对应的所有操作在程序的编译器都可以做出检查，从而判断是否有错误存在。但毕竟我们也有其他的需求，比如希望同时有多个指针指向同一块存储区域。这时就需要使用Rc指针。

```
let five = Rc::new(5);
let five1 = five.clone();        // 此时five/five1都是有效的
```

但显然，使用Rc指针的时候我们无法在编译过程中发现可能的错误。并且，Rc指针类似于Java，当对一块内存的所有引用都失效之后，系统会释放这部分内存。而这个过程都需要在程序执行的过程中，有对应的管理代码不停地工作，以保证跟踪内存的引用和内存的释放（垃圾回收）。这就产生了运行时开销。

为了对运行时开销能够更精确地掌控，Rust在语言层面增加了许多选择，这些选择在其他语言中本来是不需要的。但一个经验丰富的程序员，可以充分利用这些不同的选择，写出高品质的代码。比如Rc指针并不支持多线程，因为其中的引用计数器操作不是原子级的，所以Rust还提供了Arc用于多线程环境。当然，原子级的操作在运行时需要额外的开销。

与Rust语言的编译设计相映成趣的是Go语言，Go语言提供非常快速的编译过程，从而提供流畅的开发体验，让Go语言易于学习和使用。但Go语言的编译质量早就为人所诟病。

当然，更极端的例子是Python、JS等脚本型的语言，脚本语言完全无须编译。虽然执行效率方面这些年来随着计算机性能的提升已经不是严重问题，但大多错误几乎都只能通过代码的执行来发现，使得脚本语言在商业软件开发中占有率一直不高，更别说操作系统这一类的底层软件了。

总结一下这一部分，Rust提供高级语言所具有的一些特征，比如自动的运行时垃圾回收机制。但同时也提供并且倾向于开发人员通过精细的设计，在开发和程序编译过程中就完成内存的设计和管理，从而及早发现错误，降低运行时开销，提高最终的代码质量。

1.2.5　有限的面向对象特征

面向对象是现代开发语言的基本能力，但Rust只提供了有限的面向对象支持。笔者衷心地认为这是一件好事，笔者一直认为现在很多程序员往往为了面向对象而去进行面向对象开发，把原本很简单的事情做得过于复杂，使得代码量和运行开销高企不下，开发效率和执行效率完全失控。

Linus Torvalds曾经在那场著名的辩论中直呼C++是“糟糕程序员的垃圾语言”，有兴趣的读者可以去看原文：*Re: [RFC] Convert builin-mailinfo.c to use The Better String Library*。

在Rust中没有直接提供“类”（Class）的概念，希望使用“对象”的程序员可以直接在结构（Struct）和枚举（Enum）类型上附加函数方法，比如：

```
// 声明一个“圆”结构类型
struct Circle { x: f64, y: f64, radius: f64, }
// 为结构实现一个方法
area impl Circle {
    fn area(&self) -> f64 { std::f64::consts::PI * (self.radius * self.radius) } }
fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
    println!("{}", c.area());        // 调用结构的内置方法计算圆的面积
}
```

看起来跟Go处理对象的方法很像，其实在面向对象方面，Go语言的理念也是高举了“简化”的大旗。

Rust也没有我们习惯了的构造函数和析构函数。上面代码中对Circle对象的初始化语句如下：

```
let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
```

就是直接对成员变量赋值。这是因为Rust推崇“明确化”（Being Explicit）的代码方式，也就是所有要执行的代码应当清晰地在代码中体现出来，而不是隐藏在一些容易忘记、容易出错的构造函数之后。

与“简化对象”相反，Rust对面向对象中“接口”（Java中的接口，或者C++中的多重继承）的概念做了发扬，贯穿在了Rust类型管理的方方面面。

当然笔者这样说有点不贴切，其实应当先忘记“接口”的概念，从头理解Rust中的trait，因为trait和接口只是在技术实现上有些类似，但在应用理念上还是很有区别的。本质上说，trait也是实现多个对象中共性的方法，比如：

```
trait HasArea {                    //求取对象的面积
fn area(&self) -> f64; }
```

随后多个对象都可以实现这个trait，从而都具有这个方法：

```
struct Circle {                    //定义一个“圆”对象
    x: f64, y: f64, radius: f64, }
impl HasArea for Circle {
    fn area(&self) -> f64 { std::f64::consts::PI * (self.radius * self.radius) }
}
struct Square {                    //定义一个“方形”对象
    x: f64, y: f64, side: f64, }
impl HasArea for Square { fn area(&self) -> f64 { self.side * self.side }
}
```

在Rust中，通过泛型的帮助，根据数据类型实现的不同trait会把类型分为不同的功能和用途。比如，具有Send trait的类型，才可以安全地在多个线程间传递从而共享数据。

比如，具有Copy trait的类型，说明数据保存在栈（Stack）上，数据的复制（赋值给其他变量）不会产生所有权的转移（参考前面所有权的例子）。还有刚才讲过Rust中没有析构函数，但如果有一些数据并没有被Rust所管理，需要自己去释放，则可以为自己定义的对象实现一个Drop trait，在其中的drop方法中释放自己申请的内存：

```
    impl Drop for CustomSmartPointer {
        fn drop(&mut self) { println!("Dropping CustomSmartPointer with data `{}`!",
self.data); }
    }
```

其他面向对象的编程特征，比如泛型、重载，与其他语言并没有很大的区别，这里不再额外介绍。这些相比较其他面向对象语言并不算丰富的语法工具，是保留了面向对象开发模式最精华的部分，并不会对业务的描述造成什么障碍，反而会让建模工作更为简洁、务实，尽可能不造成代码上的晦涩和运行时的低效。

早期出现的开发语言，比如C、Java，本身并没有附加官方的管理工具，比如包管理、测试管理、编译管理。

在编程语言的发展过程中，因为开发工作的需求，往往会出现多个有影响力的工具。在C/C++方面，常见的编译管理工具有Makefile、CMake、AutoMake等，包管理工具往往与系统包管理工具结合在一起，常见的有APT、YUM、Aptitude、Dnf、HomeBrew。Java的情况与之类似。

新近风靡的语言，比如Python，pip工具占了大部分市场，Node.js则是NPM用户最多。Go语言的同名管理工具就更不用说了。这些现象跟语言本身的官方支持密不可分。

Rust也由官方直接发布了Cargo工具，功能涵盖版本升级、项目管理、包管理、测试管理、编译管理等多方面。

大多数初学者的Rust之旅就是由执行cargo new helloworld开始的。

开发语言的综合管理工具对于构建大型的软件项目必不可少。相信在Cargo的帮助下，可以让学习者快速学以致用，把一些项目迁移至Rust能轻松不少。

1.2.6　扩展库支持

一门语言能否被大量用户支持，与语言所提供的扩展库功能密不可分。笔者就见到不少程序员学习Python的原因是，Python能够更好地支持PyTorch、TensorFlow等机器学习工具包。Rust通过Crate（可以翻译为扩展箱）机制支持自己的扩展包，而且通过内置的Cargo工具可以直接使用大量的官方预置扩展包和社区共享的扩展包。此外，Rust还可以通过FFI接口（Foreign Function Interface）直接调用其他语言编写的函数库或者共享Rust函数供其他语言调用。比如，我们在Rust中调用C++写的Snappy压缩、解压功能包。Snappy官方网站为https://google.github.io/snappy/。因为使用了libc扩展库，需要在Cargo.toml中设置库依赖：

```
    [dependencies] libc = "0.2.0"
```

编译的时候，rustc会自动链接libc库和宏定义指明的Snappy压缩解压库。把Rust中定义的函数共享给C语言调用也很类似，请看Rust的代码：

```
    extern crate libc;
    use libc::uint32_t; #[no_mangle]
    pub extern fn add(a: uint32_t, b: uint32_t) -> uint32_t { a + b }
```

上面的代码需要设置Cargo.toml文件的lib参数：

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

从而让rustc将项目编译为.dylib动态链接库文件（macOS）或者.so动态链接库文件（Linux）。对应的C语言代码如下：

```
#include extern "C" uint32_t
add(uint32_t, uint32_t);
int main(){ uint32_t sum = add(5, 5);
    return 0;
}
```

C代码编译的时候，记着使用-l参数链接Rust生成的动态链接库。

综上所述，迁移至Rust完全不用担心扩展库的限制，也完全不用担心同现有软件资源之间的互动和共享。可以从一个小的项目作为切入点，边学边用，在享受Rust安全可靠的同时，逐渐达成软件架构的迁移。

1.2.7 Rust 是一种可以进行底层开发的高级语言

现在流行的开发语言很多，但能够进行操作系统底层开发的选择项并没有几个。除传统的C、新近的Go外，Rust是另一个不错的选择。要做到这一点，除Rust是真正的二进制编译外，Rust还具有非常小并且可控的“脚印”（Footprint）。这代表Rust可以做到在完全没有自己的运行时库支持下运行。

作为新兴的开发语言，Rust在函数式编程、网络编程、多线程、消息同步、锁、测试代码、异常处理等方面都有不俗表现，但本书这里不展开介绍。建议在学习Rust的过程中，根据所选教程的组织结构来逐步了解。在企业应用中，Web框架和ORM是最常用的组件，但这应当说是Rust当前的一个短板。因为毕竟Rust是一个新兴的生态系统，尽管选择很多，但尚没有重量级的选手出现。在性能和规模化的应用方面还有待市场验证。

总之，Rust首先包含长期软件工程中对于高频Bug的经验总结，从而开创性地提出了大量全新编程理念。不同于很多新式语言给予开发者更多的便利和自由，Rust更苛刻地对待程序员的开发工作。尽管在易用方面Rust也下了不少功夫，但相对于繁复的规则，这些努力很容易被忽视。而这些“成长的代价”保证了更高品质的开发输出。比如，自2004年以来，微软安全响应中心（Microsoft Security Response Center，MSRC）已对所有报告过的微软安全漏洞进行了分类。根据其提供的数据，所有微软年度补丁中约有70%是针对内存安全漏洞的修复程序。恐怕没有人再继续做延伸统计，比如这些安全漏洞造成了多少经济损失。所以，甚至已有传闻微软正在探索使用Rust编程语言作为C、C++和其他语言的替代方案，以此来改善应用程序的安全状况。

Rust并不适合初学者，只有经历过大量实践磨炼，甚至被安全漏洞痛苦折磨的资深开发者，才更能理解Rust的价值。自由还是安全，终要有所取舍。