Rust

Das umfassende Handbuch

DAS VORWORT

Kapitel 1
Über dieses Buch

Rust hat es in den vergangenen Jahren weit gebracht, seit diese Programmiersprache 2015 als stabil gekennzeichnet und veröffentlicht wurde. Bis 2020 beschäftigte Mozilla das Kernteam, das die Sprache von Graydon Hoare übernommen hatte. Hoare war damals ebenfalls Angestellter bei Mozilla und erdachte Rust in den Jahren 2006 bis 2009 als Privatprojekt neben der Arbeit. Später schied er aus privaten Gründen aus dem Rust-Team aus und wechselte anschließend nach einiger Zeit zu Apple, um an Swift zu arbeiten.

2020 führten Unternehmensentscheidungen dazu, dass die Rust-Entwicklung innerhalb von Mozilla zum Ende kam. Das Rust-Team gründete daraufhin die Rust Foundation, der sich mehrere Unternehmen anschlossen, darunter Mozilla, Amazon AWS, Huawei, Google und Microsoft.

Seit 2015 erscheinen größere Rust-Veröffentlichung im Dreijahrestakt (2015, 2018, 2021, 2024 ...) und werden als *Edition* bezeichnet. Eine herausragende Zwischenveröffentlichung stellte Rust 1.39.0 dar, mit der die Programmiersprache die asynchrone Programmierung mithilfe von `async` und `await` implementierte. Das vereinfachte die nebenläufige Programmierung immens und gab in der Folge dem Interesse an Rust großen Anschub.

Graydon Hoare erdachte Rust als Alternative zu C oder C++. Er sah die Nutzung der Sprache daher in der Systemprogrammierung, etwa von Betriebssystemen, oder als Werkzeug in der eingebetteten Programmierung. Daher lag der Fokus darauf, größtmögliche Leistung zu erzielen. Rust beeindruckte schon zu Beginn mit einer außerordentlichen Ausführungsgeschwindigkeit. Dabei setzte die Sprache in den ersten Jahren sogar auf einen Garbage Collector. Die Implementierung von Referenzzählern wie `Rc<T>` oder `Arc<T>` zeigte aber schon bald, dass Rust ohne den Garbage Collector auskommen würde.

Rust teilte mittlerweile einige Aspekte mit C++, insbesondere das Exception- und Threading-Modell. Das vereinfacht die Verbesserung der Sprache, da Forscher sich auf C++ oder Rust konzentrieren können und beide von den Ergebnissen profitieren. Zudem schafft diese Anlehnung auch ein vertrautes Gefühl: Mit den Konzepten hat Rust auch Eigenheiten oder Fehler übernommen. Wenn Sie als C++-Entwickler*in zu Rust wechseln und die Syntax erlernt haben, treffen Sie dementsprechend auf viele Gemeinsamkeiten.

Rust begeistert Entwickler und Entwicklerinnen weltweit, weil die Sprache den ausgereiften Paket-Manager Cargo mitbringt und sich mit dem Konsolenprogramm `rustup` sowohl installieren als auch verwalten lässt. Sie können Ihre Rust-Umgebung und die Ihres Teams daher umfassend konfigurieren, ohne sich in Einstellungsdateien oder Registrierungen zu verlieren. Zudem erweitern Sie Ihr Rust-Projekt mit nur einem Kommando um neue Pakete. Das kann ein Zufallszahlengenerator sein oder ein ganzes HTTP-Framework! Je leichter der Paket-Manager sich in Ihren Arbeitsfluss integriert, desto effektiver setzen Sie neue und vorhandene Bausteine zusammen.

1.1 Was Sie in diesem Buch lernen werden

Rust ist eine vielseitige Programmiersprache. Das Linux-Projekt und Microsoft setzen die Sprache im Kernel-Umfeld ein. Andere setzen Rust in der Robotik oder in Datenbank-Technologien ein, etwa für InfluxDB 3.0. Web-Frameworks wie Rocket oder Actix Web implementieren HTTP-Server, die Sie mit nur einem Kommando und wenigen Zeilen Code starten. Dank des *Foreign Function Interfaces* können Sie Rust-Projekte mit anderen Programmiersprachen und Technologien zusammenschalten. Einige Projekte verknüpfen etwa Rust als Core-Komponente für Anwendungen oder mobile Apps mit dem UI-Framework Flutter, in dem die Teams Clients für Desktop und mobile Plattformen entwickeln.

Diese Bandbreite kann dieses Buch unmöglich abdecken. Stattdessen möchte ich Ihnen einen umfassenden und tiefen Einblick in die Programmiersprache selbst geben. Die Lektüre wird Sie dafür rüsten, jeden der Anwendungsfälle anzugehen. Viel wichtiger als die Erklärung einer spezifischen Schnittstelle finde ich, dass Sie die fundamentalen Konzepte von Rust zu verstehen, die die Sprache kennzeichnen und so beliebt machen.

Genau diese Erfahrung habe ich auf meiner eigenen Reise durch Rust gemacht: Viele Online-Ressourcen und Bücher konzentrieren sich darauf, die Sprache mehr oder weniger abzubilden. Die Besonderheiten der Syntax, darunter generische Lebenszeiten, erwähnen diese Quellen dann, als seien sie gängig oder ohne jede Komplexität. Das liegt nicht selten daran, dass ein Buch ein Framework, eine Schnittstelle oder ein ganzes Themenfeld mit Rust erklären möchte – die Sprache schrumpft dann zum Beiwerk zusammen.

Stattdessen möchte ich Sie an den entsprechenden Stellen tief in den Kaninchenbau führen. Wir besprechen das Konzept des *Eigentums*, das den Zugriff auf Variablen und Referenzen in einer Art bestimmt, sodass der Rust-Compiler jede Ungültigkeit verhindern kann. Sie werden in Ihrem Rust-Programm nie eine Referenz benutzen, die auf ein Speicherobjekt zeigt, das die Laufzeitumgebung zuvor freigegeben hat!

Dazu macht Rust die *Lebenszeit* zum Teil der Syntax. Die Lebenszeit beschreibt, wie lange eine Variable einen Wert im Speicher bindet. Dieser Begriff ist nicht neu. Sie begegnen ihm auch in C oder C++. Innovativ ist jedoch, dass Sie die Lebenszeiten von Referenzen ins Verhältnis zueinander stellen. Damit sagen Sie dem Compiler, dass eine Referenz mindestens so lange leben muss, wie die andere, auf die sie sich bezieht. Viele dieser Lebenszeiten wird der Compiler bereits für Sie erkennen und einsetzen.

Das Eigentum spielt bei der Zuweisung eine essenzielle Rolle. Der Compiler bewegt einen Wert entweder von einer Variable zur anderen oder von einer Variable beim Aufruf in eine Funktion. Oder er erstellt eine bitweise Kopie des Speicherobjekts, sodass es fortan zwei gibt. Durch solcherlei Regeln garantiert Rust nicht nur die Abwesenheit von Speicherfehlern, sondern auch die von Data Races!

Alle syntaktischen Elemente von Rust bauen auf diesen Grundgesetzen der Sprache auf: Eigentum und Lebenszeit. Manche Eigenschaften der Sprache erscheinen im Schnelldurchflug beliebig, verwirrend oder im schlimmsten Fall ärgerlich. Dieses Buch macht sich dagegen zur Aufgabe, Ihnen den Hintergrund zu vermitteln, sodass Sie die Sprache nicht nur kennen, sondern verstehen lernen!

1.2 Was dieses Buch Ihnen zeigen möchte

Im ersten Teil dieses Buchs führe ich Sie durch das Konzept Eigentum. Sie erlernen in Kapitel 3 den Umgang mit Variablen, insbesondere mit der Veränderlichkeit und dem *Move*. Nachdem Sie dort den fundamentalen Datentypen von Rust begegnet sind, führt Kapitel 4 Sie durch verschiedene Speichersegmente und Referenzen. Sie lernen den Unterschied zwischen geteilten und veränderlichen Referenzen sowie die Regeln des Borrow-Checkers. Kapitel 5 fußt auf den Erkenntnissen zu Speicher und Referenz und stellt Ihnen die zwei String-Datentypen von Rust vor. In Kapitel 6 begegnen Sie den sequenziellen und assoziativen Collections von Rust, die auch als Container bezeichnet wurden.

Nachdem Sie die fundamentalen Mechaniken und Datentypen von Rust bereist haben, besprechen wir im zweiten Teil, wie Sie syntaktische Elemente deklarieren. Wir betrachten zunächst in Kapitel 7 die Funktionen. Dann diskutieren wir die Begriffe Anweisung und Ausdruck. Sie lernen in Kapitel 8, dass nahezu die gesamte Sprache aus Ausdrücken besteht und somit Werte produziert. Das können ebenfalls Muster, die jedoch eine eigene syntaktische Klasse bilden. In Kapitel 8 behandeln wir abschließend den Musterabgleich und sehen uns an, wie er sich durch Schleifen und konditionale Ausdrücke hindurchzieht.

Die Fehlerbehandlung ist Thema von Kapitel 9. Mit Strukturen, Traits und Enumerationen deklarieren oder erweitern Sie Datentypen in Rust. Kapitel 10, Kapitel 11 und

Kapitel 12 geben Ihnen alles mit auf den Weg, um zusammengesetzte Datentypen zu implementieren. Als *Traits* bezeichnet Rust abstrakte Basisklassen, die man in anderen Sprachen auch als *Protokoll* oder *Interface* bezeichnet. Traits durchfließen die gesamte Sprache und haben einen erheblichen Einfluss darauf, wie Sie mit einem Wert umgehen können.

Der dritte Teil führt Ihnen die Projektstruktur und den Paket-Manager vor. Nachdem Sie Kapitel 13 durchgearbeitet haben, kennen Sie alle Details zu *Crates*, Modulen und Pfaden. Kapitel 14 vermittelt Ihnen, wie Sie die generische Programmierung in Rust einsetzen. Danach betrachten wir das Themenfeld des Iterators in Kapitel 15.

Kapitel 16 stellt Ihnen die nebenläufige Programmierung in Rust vor. Sie erlernen das Thread-Modell und erfahren, wie Sie die Lebenszeiten von Werten oder Referenzen über mehrere Ausführungsfäden hinweg teilen. Wir besprechen insbesondere, warum es zu keinem Data Race, jedoch zu Race Conditions kommen kann. Wir behandeln überdies atomare Datentypen, besprechen Smart Pointer und Referenzzähler. Anschließend nehmen wir uns Zeit, um die asynchrone Programmierung in Rust anhand von `async` und `await` zu verfolgen.

Rust implementiert mit Makros eine wirkmächtige Meta-Sprache. Die Standardbibliothek und viele Pakete nutzen diese Möglichkeiten, um Ihnen die Arbeit mit der Codegenerierung zu erleichtern. Darunter die automatische Implementierung von Traits. Kapitel 17 führt Sie durch deklarative und prozedurale Makros. Wir leuchten dieses Thema aus und zeigen, dass ihm keinerlei Magie innewohnt.

Im letzten Teil schließen wir die Betrachtung der Sprache Rust mit automatischen Tests und der Dokumentation von Code in Kapitel 18 ab. Beides ist in den Paket-Manager `cargo` integriert. Sobald Sie Rust installiert haben, verfügen Sie daher schon über einen Test-Runner und einen Generator für die Projektdokumentation. Wie Sie mit Bibliotheken oder Programmen zusammenarbeiten, die in anderen Sprachen programmiert wurden, zeigt Kapitel 19 auf. Es diskutiert das Foreign Function Interface von Rust, das jedoch nicht ohne `cargo` denkbar ist. Nur in Unsafe Rust dürfen Sie primitive Zeiger einsetzen und Code schreiben, der auf Verträgen basiert. Der Rust-Compiler lässt Ihnen in diesen Codeblöcken freie Hand, überträgt damit gleichzeitig aber die Verantwortung auf Ihre Schultern.

1.3 Noch mehr Informationen und Guides

Rust ist eine unglaublich gut dokumentierte Sprache. Wenn Sie in einen Themenbereich tiefer einsteigen möchten, müssen Sie dazu nur das entsprechende Rust-Book öffnen. Tabelle 1.1 liefert eine Übersicht aus Quellen, die ich selber immer wieder gerne konsultiere:

Quelle	Beschreibung
https://doc.rust-lang.org/std/	Die Dokumentation der Standardbibliothek von Rust
https://doc.rust-lang.org/std/	Die Referenz der Sprache
https://github.com/rust-lang/rust	Das quelloffene Projekt auf GitHub
https://rust-lang.github.io/rustfmt	Dokumentation aller Lint-Regeln und deren Konfiguration
https://doc.rust-lang.org/book/	Das Online-Rust-Buch
https://doc.rust-lang.org/nomicon/intro.html	Das *Rustonomicon* steigt tief in den Maschinenraum ein und behandelt die Sprache vor dem Hintergrund von Unsafe Rust.
https://www.lurklurk.org/effective-rust/cover.html	Dieses Rust-Book fasst vielerlei Tipps zusammen, mit denen Sie noch besseren und idiomatischen Rust-Code schreiben.
https://rust-lang.github.io/async-book/01_getting_started/01_chapter.html	Hier finden Sie weitergehende Informationen zur asynchronen Programmierung.
https://veykril.github.io/tlborm/	»The Little Book of Rust Makros«: Wenn Sie sich nach der Lektüre dieses Buchs weiter mit Makros befassen möchten, sollten Sie dieses Rust-Book öffnen.
https://doc.rust-lang.org/stable/rustdoc/what-is-rustdoc.html	Erklärung von `rustdoc`, mit dem Sie Ihre Projekte dokumentieren und Rust-Books schreiben.
https://rust-lang.github.io/api-guidelines/about.html	Der API-Design-Guide; er speist sich aus den Erfahrungen des Rust-Teams.
https://rustc-dev-guide.rust-lang.org/getting-started.html	Architektur und Verhalten des Compilers
https://rust-lang.github.io/rustup/	Die Dokumentation des Werkzeugs `rustup`, mit dem Sie die Rust-Installation verwalten
https://doc.rust-lang.org/cargo/index.html	Der Paket-Manager `cargo` mit allen Details zum Werkzeug und Manifestdateien

Tabelle 1.1 Eine Liste von weiterführenden Rust-Dokumentationen

1.4 Danksagung

Mein besonderer Dank gebührt Frau Almut Poll für Ihre Hilfe und Unterstützung sowie dem Rheinwerk Verlag. Außerdem danke ich Torsten T. Will dafür, dass er seinen aufmerksamen Blick auf das Fachliche gerichtet und mir zahlreiche Hinweise und Empfehlungen geliefert hat.

Ein Buch zu schreiben, erfordert einen erheblichen Zeitaufwand. In diesem Fall haben sich die Arbeiten am vorliegenden Buch über Jahre erstreckt. Ich bin daher unfassbar dankbar für das Verständnis und die Nachsicht, die mir meine Frau Anne und meine Tochter Thalea über diese Zeit entgegengebracht haben.

Marc Marburger
Berlin, April 2024