Marco Amann · Joachim Baumann · Marcel Koch

Rust

Konzepte und Praxis
für die sichere Anwendungsentwicklung

dpunkt.verlag

Marco Amann hat Softwaretechnik studiert und arbeitet bei Digital-Frontiers als Consultant. Er ist als einer der Experten der Digital Frontiers für das Thema Rust verantwortlich und hat Schwerpunkte in den Bereichen systemnaher Programmierung und robuster Systeme.

Dr. Joachim Baumann ist Management Consultant und Geschäftsführer der Digital Frontiers GmbH & Co. KG. Er verfügt über mehr als 30 Jahre Erfahrung in der IT, als Entwickler, Architekt, Projektleiter, Scrum-Master und Berater und beschäftigt sich seit dem Jahr 2000 mit agilen Vorgehensweisen. Sein Wissen gibt er gerne in Form von Büchern, aber auch als Hochschuldozent und in Schulungen weiter, er ist aber auch immer noch Committer in Open-Source-Projekten.

Marcel Koch vermittelt – ob zwischen Technologien oder Menschen. Er versteht es, in verschiedenste Technologien und Gebiete einzutauchen, die Vorteile zu nutzen und die Essenzen zu erklären. Als Kommunikationscoach setzt er auf gewaltfreie Kommunikation, Transaktionsanalyse und Radical Candor. Als Softwarearchitekt baut er auf konservative Technologien, wie zum Beispiel Kotlin oder Spring, und ergänzt diese bedarfsgerecht mit neueren wie WebAssembly oder Rust. *www.marcelkoch.net*

Marco Amann · Joachim Baumann · Marcel Koch

Rust

Konzepte und Praxis für die sichere Anwendungsentwicklung

dpunkt.verlag

Marco Amann · Joachim Baumann · Marcel Koch

Lektorat: René Schönfeldt
Projektkoordinierung: Anja Weimer
Copy-Editing: Annette Schwarz, Ditzingen
Satz: Veronika Schnabel
Herstellung: Stefanie Weidner
Umschlaggestaltung: Helmut Kraus, *www.exclam.de*

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:
Print 978-3-86490-878-1
PDF 978-3-96910-614-3
ePub 978-3-96910-615-0
mobi 978-3-96910-616-7

1. Auflage 2022

Wieblinger Weg 17
69123 Heidelberg

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: *hallo@dpunkt.de*.

5 4 3 2 1 0

Inhaltsübersicht

Inhaltsverzeichnis

Vorwort

Zielgruppe für dieses Buch

Dieses Buch richtet sich an Entwickler, die über den Tellerrand normaler Programmierung hinausschauen wollen. Mit Rust bietet sich die Möglichkeit, eine neue und interessante Programmiersprache kennenzulernen, die einiges anders macht als die gängigen Programmiersprachen.

Notwendige Vorkenntnisse für das Buch

Sie sollten als Entwickler bereits Erfahrung in der objektorientierten Programmierung haben. Wissen um typische Konzepte gerade objektorientierter Sprachen sollte vorhanden sein. Wir erklären auch keine gängigen Programmierkonstrukte wie `if` oder `while`, sondern setzen die dafür notwendigen Kenntnisse voraus.

Was dieses Buch nicht enthält

Dieses Buch wiederholt weder die leicht zu findenden Tutorials noch die exzellente Online-Dokumentation zu

Rust. Es ist auch keine Einführung für Menschen, die das Programmieren erst lernen wollen (das Buch wäre sonst mehr als doppelt so dick).

Was dieses Buch enthält

Unser Ziel ist es, Sie mit den Inhalten dieses Buchs in die Lage zu versetzen, auch komplexere praktische Programme mit Rust zu schreiben.

Natürlich hoffen wir außerdem, dass Sie genau wie wir die Freude dabei verspüren, Programme zu schreiben, bei denen Speicherlecks der Vergangenheit angehören.

Zum Schluss ist die Auseinandersetzung mit dem Ownership-Modell von Rust eine interessante Erfahrung, die uns auch in anderen Sprachen zu besseren Programmierern macht.

Begleitrepository

Zu diesem Buch haben wir ein git-Repository eingerichtet, in dem Sie viele Codebeispiele direkt zum Klonen und Ausprobieren finden:

https://rust-buch.de/repository

Danksagung

Das Schreiben eines Buches ist, wenig überraschend, ein langwieriges und aufwendiges Unterfangen, das unsere Familien sehr stark unterstützt haben. Und dies, obwohl wir dadurch weniger aufmerksam waren, mehr Zeit mit unverständlichen Dingen zubrachten, ständig »Oh, da muss ich auch noch dran denken …« vor uns hinmurmelten und generell leicht geistig abwesend waren. Vielen Dank für die Geduld und das Verständnis.

Marco, Joachim

Auch wenn dies eher unüblich ist, möchten wir ganz explizit unserer Firma Digital Frontiers danken, die mit ihrem ungewöhnlichen Arbeitsmodell das Schreiben eines Buches in der Arbeitszeit ermöglicht und derartige inhaltliche Arbeit stark fördert.

Marcel

Ich möchte mich bei meinen beiden Co-Autoren Marco und Joachim für ihr Wissen, ihren Humor und ihre Geduld bedanken. Das Privatleben gab mir nicht immer so viel Zeit frei, wie wir uns alle gewünscht hätten.

Ich danke ausdrücklich meiner Frau Johanna und meinem Sohn Niklas. Wie oben schon erwähnt, mussten sie

sehr oft auf mich verzichten. Vielleicht lest Ihr, Niklas und Larissa, das Buch auch mal im Informatikgeschichtsunterricht.

Ihr drei seid meine Sonne und Hoffnung!

1 Rust – Einführung

In diesem Kapitel werfen wir einen ersten Blick auf Rust-Programme, betrachten die Installation von Rust und der Sprachunterstützung in verschiedenen Entwicklungsumgebungen, sodass wir möglichst schnell praktische Schritte mit der Sprache unternehmen, ein Beispielprogramm schreiben und mit dem Rust-eigenen Build-System übersetzen und starten können.

1.1 Warum Rust?

Rust ist eine moderne Sprache, die sehr stark auf Geschwindigkeit und Parallelverarbeitung ausgelegt ist. Vielfach wird Rust als Systemprogrammiersprache und Ersatz für C dargestellt, der Anwendungsbereich ist aber sehr viel breiter. Betrachten wir ein paar der interessanten Eigenschaften von Rust.

1.1.1 Rust und der Speicher

Das absolute Alleinstellungsmerkmal ist die Art, wie Rust mit Speicher umgeht. Rust kann garantieren, dass durch die Verwaltung des Speichers zur Übersetzungszeit keine Fehler zur Laufzeit auftreten können. Damit braucht Rust auch keinen Garbage Collector. Das verhindert

unbeabsichtigte Unterbrechungen im Programmablauf, um den Speicher aufzuräumen. Wir haben also nicht nur korrektere Programme, die schneller laufen, sie verhalten sich auch deterministischer.

Um dies zu erreichen, wird für jeden Wert ein Eigentümer festgelegt. Dies kann ein primitiver Wert sein oder eine beliebig komplexe Struktur. Ein Wert lebt, solange der Eigentümer lebt.

Der Eigentümer kann wechseln, und für den Zugriff auf ein Objekt können Referenzen ausgeliehen werden (*Borrowing*). Ausgeliehene Referenzen sind im Normalfall Lesereferenzen, es kann aber alternativ auch maximal eine Schreib-/Lese-Referenz auf einen Wert definiert werden. Dies impliziert, dass wir keine aktive Lesereferenz haben. Die Beschränkung auf eine einzige schreibende Instanz sorgt bei Neulingen meist für Überraschungen, hat aber den großen Vorteil, dass es keine undefinierten Zustände durch gleichzeitiges Schreiben oder nicht synchronisiertes Lesen geben kann.

Dieses *Ownership* genannte Konzept ist extrem mächtig, braucht aber zum vollständigen Verinnerlichen etwas Zeit und Übung. Wir werden dies in Abschnitt 7.2 kennenlernen und in Kapitel 15 im Detail beleuchten.

1.1.2 Rust und Objektorientierung

Rust ist eine Programmiersprache, die mit der Kapselung von Daten und Funktionen und Methoden auf diesen Daten objektorientierte Konzepte unterstützt.

Rust erreicht dies durch die Einführung von Modulen, die private und öffentliche Daten und Funktionen enthalten. Polymorphismus wird durch das Konzept der *Traits* erreicht, die inzwischen in vielen anderen Programmiersprachen wie Kotlin oder Scala auch verwendet werden. Eine

vergleichbare Funktionalität gibt es in Java seit der Version 8 mit den Default-Methoden in Interface-Spezifikationen.

Rust bietet allerdings anders als die gewohnten objektorientierten Sprachen keine Vererbung. Dies mag im ersten Moment überraschen und ist eine Abkehr vom normalen objektorientierten Denken, hat aber gute Gründe.

Aus konzeptioneller Sicht ist es problematisch, dass wir bei der Vererbung nicht kontrollieren können, welche Teile unserer Elternklasse wir erben möchten. Dies kann dazu führen, dass wir in abgeleiteten Klassen Funktionalität haben, die dort nicht gewollt ist.

Das praktischere Argument ist aber, dass durch Verzicht auf Vererbung ein hoher Aufwand zur Identifikation der richtigen auszuführenden Methode/Funktion wegfällt. Dies macht Rust-Programme deutlich laufzeiteffizienter.

Wir werden uns mit objektorientierten Konzepten in Kapitel 9 auseinandersetzen.

1.1.3 Rust und funktionale Programmierung

Zur Unterstützung funktionaler Programmierung bietet Rust *Closures*, anonyme Funktionen, die auf ihre Umgebung zur Zeit der Definition zugreifen können. Dieses vielseitige Konstrukt findet sich in mehr und mehr Sprachen und erlaubt eine sehr elegante Kapselung von Funktionalität und Daten.

Zusammen mit Iteratoren, die die Verarbeitung von Sammlungen von Daten kapseln, erlauben Closures sehr mächtige funktionale Abstraktionen. Iteratoren und Closures werden wir in Kapitel 11 kennenlernen.

1.1.4 Rust und Parallelverarbeitung

Rust bietet eine direkte Abstraktion der Thread-Funktionalität des unterliegenden Betriebssystems. Dies

sorgt für den geringstmöglichen Mehraufwand zur Laufzeit, beschränkt aber natürlich die Flexibilität in der Verwendung von Threads auf die Unterstützung durch das unterliegende System. Bei Bedarf können allerdings auch Thread-Module verwendet werden, die eine unabhängige und damit flexiblere Implementierung anbieten. Dies erlaubt uns, von Fall zu Fall zu entscheiden, ob wir die größere Flexibilität oder den geringeren Speicherbedarf bevorzugen. Während die Entscheidung in vielen Fällen in Richtung der Flexibilität getroffen werden wird, gibt es eingeschränkte Umgebungen (wie zum Beispiel Mikro-Controller), in denen die Möglichkeit der expliziten Wahl sehr vorteilhaft ist.

Viele der Probleme, die bei der normalen Programmierung von paralleler Verarbeitung zu sehr hoher Komplexität und damit zu schwer auffindbaren Fehlern führen, finden wir in Rust nicht. Dies entsteht durch das *Ownership*-Modell, das dafür sorgt, dass der Compiler problematische Stellen im Quelltext sehr früh identifizieren und damit entfernen kann. Das heißt nicht, dass Rust alle Probleme im Zusammenhang mit Parallelprogrammierung löst. Es erlaubt uns aber, uns auf die wirklich schwierigen Probleme zu konzentrieren.

Threads in Rust können kommunizieren, indem sie Nachrichten in verschiedene Kanälen senden oder aus diesen empfangen. Zusätzlich können sie Teile ihres Zustands geschützt durch eine Mutex-Abstraktion mit anderen Threads teilen.

Parallelprogrammierung in Rust ist sehr mächtig, und wir werden uns in Kapitel 16 eingehend damit beschäftigen.

1.2 Ein Beispielprogramm

Als ein erstes Beispiel, um Ihren Appetit für Rust zu wecken, betrachten wir ein kleines Programm, das bereits viele