System Własności i Zapożyczeń w Rust

Norbert Olkowski

April 25, 2024

Contents

1 Wstep

Cel zajeć: Celem tych zajeć jest wprowadzenie w mechanizmy zarzadzania pamiecia w jezyku programowania Rust, ze szczególnym uwzglednieniem systemu własności i zapożyczeń.

2 Podstawowe pojecia

2.1 Własność (Ownership)

W Rust każda wartość ma swojego **właściciela**, a zasieg tej wartości jest ograniczony przez zasieg jej właściciela. Kiedy właściciel wychodzi poza zasieg, wartość jest automatycznie usunieta. Ten mechanizm zarzadzania pamiecia jest nazywany *systemem własności* i jest jedna z najważniejszych cech jezyka Rust, zapewniajaca bezpieczeństwo pamieci i eliminacje wielu typów błedów zwiazanych z zarzadzaniem pamiecia.

Zasady własności W Rust, zasady własności obejmuja trzy główne aspekty:

1. Każda wartość w Rust ma jednego właściciela.
2. W każdym czasie może istnieć tylko jeden właściciel wartości.
3. Gdy właściciel (zmienna) wychodzi poza zakres działania, wartość jest zwalniana.

Przykład przeniesienia własności Przeniesienie własności wartości z jednej zmiennej do innej jest powszechna praktyka w Rust, która pozwala unikać niepotrzebnego kopiowania danych. Oto przykład demonstrujacy przeniesienie własności:

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

W powyższym przykładzie, po przeniesieniu własności zmiennej `s1` do `s2`, próba użycia `s1` spowoduje błąd kompilacji. Jest to efekt zasady, która zapewnia, że w każdym momencie tylko jedna zmienna może zarzadzać dana wartością, co eliminuje problem podwójnego zwalniania pamieci.

Zwalnianie zasobów Automatyczne zwalnianie zasobów, gdy zmienna wychodzi poza zakres, jest znaczacym atutem Rust, zwłaszcza w kontekście zarzadzania pamiecia w aplikacjach. System ten jest czesto porównywany do deterministycznego zarzadzania pamiecia w innych jezykach, takich jak C++, ale bez ryzyka zwiazanych z błedami programisty.

Podsumowanie System własności Rusta jest kluczowym elementem, który pozwala programistom na pisanie kodu bezpieczniejszego, wydajniejszego i łatwiejszego do zarzadzania. Zapobiega on powszechnym problemom, takim jak wycieki pamieci, dostep do już zwolnionej pamieci, czy wyścigi danych, które sa czesto spotykane w innych jezykach programowania.

2.2 Zapożyczenia (Borrowing)

Zapożyczenia w Rust pozwalaja na dostep do danych bez przejmowania ich własności. Jest to istotna cecha jezyka, która umożliwia bezpieczne i wydajne zarzadzanie pamiecia. W Rust istnieja dwa typy zapożyczeń:

- Niemutowalne zapożyczenia (`&`), które pozwalaja na wielokrotne odczyty danych.
- Mutowalne zapożyczenia (`&mut`), które pozwalaja na modyfikacje danych, ale sa wyłaczne w danym zasiegu.

Niemutowalne zapożyczenia Niemutowalne zapożyczenia sa najcześciej używane w Rust. Pozwalaja one wielu cześciom programu na odczyt danych, ale zapobiegaja ich modyfikacji. To zapewnia, że dane sa bezpieczne przed przypadkowymi zmianami w trakcie ich używania, co jest szczególnie ważne w wielowatkowych aplikacjach.

```
fn print_length(s: &String) {
    println!("Dlugosc stringa: {}", s.len());
}

fn main() {
    let s = String::from("Hello, world!");
    print_length(&s);
    println!("String s nadal dostepny: {}", s);
}
```

W powyższym przykładzie, funkcja `print_length` zapożycza string `s` niemutowalnie, co pozwala na jego odczyt, ale nie modyfikacje. Zmienna `s` jest dalej używana po wywołaniu funkcji, co demonstruje, jak Rust umożliwia bezpieczne współdzielenie danych.

Mutowalne zapożyczenia Mutowalne zapożyczenia pozwalaja na zmiane danych, ale Rust zapewnia, że tylko jedna mutowalna referencja do danych może istnieć w danym zakresie. To eliminuje ryzyko wyścigów danych i innych problemów zwiazanych z równoczesnym dostepem do danych.

```
fn modify(vec: &mut Vec<i32>) {
    vec.push(42);
}

fn main() {
    let mut v = vec![1, 2, 3];
    modify(&mut v);
    println!("Zmodyfikowany wektor: {:?}", v);
}
```

W przykładzie powyżej funkcja `modify` zapożycza wektor `v` mutowalnie, co pozwala na dodanie elementu do wektora. Kompilator Rust gwarantuje, że żadne inne zapożyczenie nie bedzie mogło jednocześnie modyfikować `v`, zapewniajac tym samym bezpieczeństwo operacji.

Podsumowanie Zapożyczenia w Rust sa fundamentalne dla bezpiecznego i efektywnego zarzadzania pamiecia. Zapewniaja one elastyczność w dostepie do danych, jednocześnie chroniac przed wieloma typowymi problemami zwiazanymi z dostepem do pamieci, które wystepuja w innych jezykach programowania. Mechanizm ten, łacznie z systemem własności, stanowi o unikalności i wydajności Rust jako jezyka programowania.

3 Przykłady kodu

3.1 Przeniesienie własności

Mechanizm przeniesienia własności (ownership) w Rust jest kluczowy dla zarzadzania pamiecia. Kiedy wartość jest przypisana do nowej zmiennej, właściwość tej wartości jest "przenoszona" do nowej zmiennej, a oryginalna zmienna przestaje mieć do niej dostep. Oto prosty przykład, demonstrujacy ten mechanizm:

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

W tym przykładzie, 's1' przestaje być dostepne po przeniesieniu jego wartości do 's2'. Próba użycia 's1' po przeniesieniu jej własności spowoduje blad kompilacji, co zapobiega błedom zwiazanym z dostepem do nieistniejacych danych.

Dodatkowy przykład z funkcja Przeniesienie własności można również zaobserwować w kontekście funkcji, gdzie argumenty sa przekazywane przez wartość:

```
fn take_ownership(s: String) {
    println!("{}", s);
}

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

3.2 Zapożyczenie

Zapożyczenia w Rust pozwalaja na korzystanie z wartości bez przejmowania ich własności. Istnieja dwa rodzaje zapożyczeń: niemutowalne (tylko do odczytu) i mutowalne (do odczytu i zapisu). Poniżej przedstawiamy przykład niemutowalnego zapożyczenia:

```
fn print_length(s: &String) {
    println!("Dlugos stringa: {}", s.len());
}

fn main() {
    let s = String::from("Hello, world!");
    print_length(&s);
    println!("String s nadal dostepny: {}", s);
}
```

Niemutowalne zapożyczenie umożliwia wielu cześciom programu dostep do tej samej wartości dla odczytu, zapewniajac, że wartość ta nie zostanie przypadkowo zmodyfikowana.

Przykład z mutowalnym zapożyczeniem Mutowalne zapożyczenie pozwala na modyfikacje zapożyczonej wartości. W tym przypadku tylko jedna mutowalna referencja może istnieć w danym zakresie:

```
fn change(s: &mut String) {
    s.push_str(", world!");
}

fn main() {
    let mut s = String::from("Hello");
    change(&mut s);
    println!("{}", s);
}
```

W tym przykładzie funkcja 'change' przyjmuje mutowalne zapożyczenie stringa 's' i modyfikuje go, dodajac ", world!". Dzieki zapożyczeniu mutowalnemu, funkcja może bezpiecznie zmodyfikować oryginalna wartość bez ryzyka konfliktów dostepu.

4 Ćwiczenia

4.1 Ćwiczenie 1: Modyfikacja Wektora

Celem tego ćwiczenia jest napisanie funkcji w jezyku Rust, która przyjmuje mutowalne zapożyczenie wektora i dodaje element do tego wektora. Ćwiczenie to pomoże zrozumieć, jak funkcje moga modyfikować dane przekazane jako argumenty bez konieczności zwracania nowych wartości.

Specyfikacja funkcji Funkcja powinna przyjmować jeden argument:

- `vec: &mut Vec<i32>` - mutowalne zapożyczenie wektora liczb całkowitych.

Funkcja powinna dodawać określona wartość do końca wektora. Wartość ta powinna być zdefiniowana w ciele funkcji.

```
fn add_element(vec: &mut Vec<i32>) {
    let new_element = 10;
    vec.push(new_element);
}

fn main() {
    let mut my_vec = vec![1, 2, 3];
    println!("Wektor przed dodaniem elementu: {:?}", my_vec);
    add_element(&mut my_vec);
    println!("Wektor po dodaniu elementu: {:?}", my_vec);
}
```

Instrukcje 1. Skopiuj powyższy kod do swojego środowiska Rust. 2. Zmodyfikuj funkcje `add_element` tak, aby przyjmowała dodatkowy argument typu `i32`, który określa, jaki element ma być dodany do wektora. 3. Testuj funkcje, dodajac różne wartości do wektora i obserwujac wyniki.

Pytania do rozważenia

1. Jak zachowuje sie wektor, gdy dodajemy do niego elementy?
2. Co sie stanie, gdy użyjemy niemutowalnego zapożyczenia zamiast mutowalnego w funkcji `add_element`?
3. Jakie inne operacje mogłyby być wykonane na wektorze w kontekście mutowalnych zapożyczeń?

4.2 Ćwiczenie 2: Usuwanie Elementów z Wektora

Cel Nauczyć sie zarzadzać pamiecia poprzez usuwanie elementów z wektora.

Specyfikacja funkcji Funkcja powinna przyjmować mutowalne zapożyczenie wektora i indeks elementu do usuniecia.

```
fn remove_element(vec: &mut Vec<i32>, index: usize) {
    vec.remove(index);
}

fn main() {
    let mut my_vec = vec![10, 20, 30, 40];
    println!("Wektor przed usunieciem elementu: {:?}", my_vec);
    remove_element(&mut my_vec, 2);
    println!("Wektor po usunieciu elementu: {:?}", my_vec);
}
```

Instrukcje 1. Zaimplementuj funkcje `remove_element` w swoim środowisku Rust. 2. Przetestuj działanie funkcji z różnymi indeksami i wektorami. 3. Zwróć uwage na zmiany w rozmiarze i zawartości wektora po każdym usunieciu.

Pytania do rozważenia 1. Jak zachowanie funkcji zmienia sie, gdy próbujemy usunać element spoza zakresu wektora? 2. Jakie sa potencjalne problemy zwiazane z usuwaniem elementów w środku wektora w aplikacjach wielowatkowych?

4.3 Ćwiczenie 3: Weryfikacja Własności i Zapożyczeń

Cel Zrozumienie, jak Rust zarzadza własnością i zapożyczeniami w bardziej złożonych scenariuszach.

Specyfikacja Napisz funkcje demonstrujace prawidłowe i błedne użycie własności i zapożyczeń.

```
fn main() {
    let s = String::from("hello");

    let s1 = &s;
    let s2 = &s;
    println!("{}, world!", s1);
    // let s3 = &mut s;

    println!("U ycie s2: {}", s2);
}
```

Instrukcje 1. Uruchom kod i zobacz, jak kompilator Rust reaguje na błedne zapożyczenia. 2. Modyfikuj kod, dodajac i usuwajac komentarze, aby zobaczyć, kiedy Rust pozwoli na kompilacje kodu.

Pytania do rozważenia 1. Co sie stanie, gdy próbujemy jednocześnie używać mutowalnego i niemutowalnego zapożyczenia? 2. Jak system typów w Rust zapobiega równoczesnemu mutowalnemu i niemutowalnemu zapożyczeniu?

5 Podsumowanie

Podczas tych zajeć skupiliśmy sie na dwóch kluczowych mechanizmach Rusta: systemie własności i zapożyczeniach. Te koncepty sa nie tylko fundamentalne dla zrozumienia tego jezyka programowania, ale także stanowia rdzeń jego bezpieczeństwa i efektywności. Dzieki nim Rust eliminuje wiele problemów zwiazanych z zarzadzaniem pamiecia, które sa obecne w innych jezykach, zapewniajac przy tym wyjatkowe mechanizmy kontroli nad dostepem do danych.

Znaczenie własności i zapożyczeń Zrozumienie systemu własności i zapożyczeń pozwala programistom pisać kod, który jest nie tylko wydajny, ale również bezpieczny przed typowymi błedami zwiazanymi z dostepem do pamieci. Własność zapewnia, że zasoby sa zwalniane automatycznie, eliminujac wycieki pamieci, podczas gdy zapożyczenia umożliwiaja bezpieczne i kontrolowane dzielenie sie danymi w aplikacji.

Praktyczne znaczenie Przez nauke i praktykowanie zastosowań własności i zapożyczeń, uczestnicy moga lepiej projektować aplikacje w Rust, które sa zarówno skalowalne, jak i łatwe w utrzymaniu. Umiejetność teoretyczna w połaczeniu z praktycznym doświadczeniem pozwala na głebsze zrozumienie, jak te mechanizmy współgraja z wielowatkowościa i zarzadzaniem zasobami, co jest kluczowe w nowoczesnym oprogramowaniu.

Dalsze kroki Zachecamy uczestników do dalszej nauki i eksploracji Rusta. Praktykowanie przez tworzenie własnych projektów, rozwiazywanie problemów i eksperymentowanie z zaawansowanymi funkcjami jezyka może znacznie przyspieszyć proces nauki. Dodatkowo, społeczność Rust jest aktywna i pomocna, a dostepne sa liczne zasoby, takie

jak oficjalna dokumentacja, fora i grupy dyskusyjne, które moga wspierać dalszy rozwój umiejetności programistycznych.