Coleções Comuns

Diogo Silveira Mendonça

Slides Baseados no Capítulo 8 do Rust Book
https://doc.rust-lang.org/book/ch08-00-common-collections.html

Introdução

- As bibliotecas padrões possuem diversas estruturas de dados conhecidas como Coleções.
- Maioria dos tipos de dados representam um único valor.
- Coleções podem contém diversos valores.
- O tamanho das Coleções não precisa ser definido em tempo de compilação diferentemente de arrays e tuplas.

Introdução

- Coleções possuem tamanho variável em tempo de execução, os dados são armazenados na Heap.

- Exemplo de coleções:
  - Vetores:
  - Strings:
  - Hash map:

- Veremos como utilizar essas coleções.

- Os exemplos de código podem ser encontrados nesse link.

Criando um Vetor

- Vetor permite armazenar mais de um valor em uma estrutura de dados, colocando todos os valores próximos na memória.

- Vetores só conseguem armazenar dados do mesmo tipo.

- Podemos iniciar um valor, como no exemplo abaixo:

```
let v: Vec<i32> = Vec::new();
```

- Explicitamos o tipo i32, para o Rust saber que queremos utilizar i32.

Criando um vetor

- Também podemos escrever a definição do vetor já adicionando os valores.
- Nesse caso não precisamos passar o tipo, pois o Rust já está inferindo que queremos utilizar i32.

Exemplo:

```
fn main() {
    let v = vec![1, 2, 3];
}
```

Atualizando um Vetor

- Podemos adicionar valores no vetor, utilizando push.
- O Rust inferiu que o dado deveria ser i32.
- Lembrando que precisamos definir utilizando let mut, para permitir modificações no vetor.

Exemplo:

```
fn main() {
    let mut v = Vec::new();

    v.push(5);
    v.push(6);
    v.push(7);
    v.push(8);
}
```

Lendo elementos de um vetor

- Existem duas maneiras de ler um vetor:
  - Utilizando o índice.
  - Utilizando get.

- Utilizando & e [ ], recebemos uma referência do valor.

- Quando utilizamos get, recebemos um Option<&T>, que podemos utilizar em um match.

- Se tentarmos acessar um índice que não existe no vetor, o compilador retornará um erro. Isso não ocorre utilizando get, pois ele irá retornar None.

Exemplo:

```
fn main() {
        let v = vec![1, 2, 3, 4, 5];

        let third: &i32 = &v[2];
        println!("The third element is {third}");

        let third: Option<&i32> = v.get(2);
        match third {
                Some(third) =>
                        println!("The third element is {third}"),
                None =>
                        println!("There is no third element."),
        }
}
```

Lendo elementos de um vetor

- Lembrando que não é possível ter uma referência imutável e mutável no mesmo escopo.
- O código ao lado apresenta um problema, já que cria uma referência imutável de um valor do vetor e tenta modificar o vetor.
- Esse problema ocorre, mesmo sem modificar especificamente o valor que é referenciado, por conta que os vetores colocam os valores um ao lado do outro na memória. Podendo alocar todos os valores para outro espaço na memória, caso não tenha espaço na posição atual.

Exemplo:

```
fn main() {
    let mut v = vec![1, 2, 3, 4, 5];

    let first = &v[0];

    v.push(6);

    println!(
        "The first element is: {first}"
    );

}
```

Repetição sobre os valores de um vetor

- Para acessar todos os valores do vetor de forma sequencial, em vez de acessar um índice específico, podemos utilizar o loop for.
- Podemos também realizar mudanças ao vetor, como mostrado no exemplo 2.
- No exemplo 2, temos o operador de desreferência “ * ”. Que pega o valor de i antes de utilizar o operador +=.
- Se tentarmos adicionar ou remover itens do vetor, teremos o mesmo erro visto anteriormente.

Exemplo 1:

```
fn main() {
        let v = vec![100, 32, 57];
        for i in &v {
                println!("{i}");
        }
}
```

Exemplo 2:

```
fn main() {
        let mut v = vec![100, 32, 57];
        for i in &mut v {
                *i += 50;
        }
}
```

Usando Enum para armazenar diversos tipos

- Vetores podem armazenar somente um tipo. Podemos utilizar enum para armazenar uma lista de itens de diferentes tipos.
- No caso ao lado, temos um exemplo, onde temos o vetor que imita uma linha de uma planilha, que pode conter int, string e float.
- O Rust precisa saber em tempo de compilação, o quanto de memória é necessário para cada elemento. No caso do enum, o Rust garante que que todos os casos sejam possíveis.

Exemplo:

```
fn main() {
        enum SpreadsheetCell {
                Int(i32),
                Float(f64),
                Text(String),
        }

        let row = vec![
        SpreadsheetCell::Int(3),
        SpreadsheetCell::Text(String::from("blue")),
        SpreadsheetCell::Float(10.12),
        ];
}
```

Dropping a Vector Drops Its Elements

- Assim como qualquer estrutura, quando o vetor sai do escopo, ele e seu conteúdo é deletado.

Exemplo:

```
fn main() {
    {
        let v = vec![1, 2, 3, 4];

        // do stuff with v
    } // <- v goes out of scope and is freed here
}
```

O que é uma string?

- Rust tem apenas um tipo de string no núcleo da linguagem, que é a string slice str geralmente vista em sua forma emprestada &str

- string slice são referências a alguns dados de string codificados em UTF-8 armazenados em outro lugar.

- Literais de string, por exemplo, são armazenados no binário do programa e, portanto, são string slices.

- Strings são na verdade uma implementação de um vetor de bytes com algumas garantias, restrições e capacidades extras.

Criando uma nova string

- Muitas das operações de um Vec<T> também estão disponíveis em Strings.

- Assim como um vetor, podemos criar uma instância vazia utilizando new().

Exemplo:

```
fn main() {
    let mut s = String::new();
}
```

Criando uma nova string

- Podemos criar uma string que já possua valor, utilizar “.to_string()”. Como no exemplo 1.

- Também podemos utilizar a função String::from, para criar uma string com um valor inicial. Como no exemplo 2.

Exemplo 1:

```
fn main() {
        let data = "initial contents";

        let s = data.to_string();

        // the method also works on a literal directly:
        let s = "initial contents".to_string();
}
```

Exemplo 2:

```
fn main() {
        let s = String::from("initial contents");
}
```

Criando uma nova string

- Como strings são codificadas em UTF-8, podemos escrever de diversas formas, todas os valores ao lado são válidos.

Exemplo:

```
fn main() {
        let hello = String::from("السلام عليكم");
        let hello = String::from("Dobrý den");
        let hello = String::from("Hello");
        let hello = String::from("שלום");
        let hello = String::from("नमस्ते");
        let hello = String::from("こんにちは");
        let hello = String::from("안녕하세요");
        let hello = String::from("你好");
        let hello = String::from("Olá");
        let hello = String::from("Здравствуйте");
        let hello = String::from("Hola");

}
```

Atualizando uma string

- Uma string pode aumentar seu tamanho ou modificar seu conteúdo.

- Podemos acrescentar a uma string, utilizando push_str ou push.

- Podemos concatenar uma string, utilizando o operador + ou format!

Acrescentando com push_str e push

- push_str recebe um slice de string e push recebe char.
- No exemplo 1, é utilizado push_str, para adicionar “bar” em s, formando foobar.
- No exemplo 2, é mostrado que push_str não pega o ownership de s2, permitindo imprimir.
- No exemplo 3, é utilizado push, que só pode adicionar uma letra.

Exemplo 1:

```
fn main() {
        let mut s = String::from("foo");
        s.push_str("bar");
}
```

Exemplo 2:

```
fn main() {
        let mut s1 = String::from("foo");
        let s2 = "bar";
        s1.push_str(s2);
        println!("s2 is {s2}");
}
```

Exemplo 3:

```
fn main() {
        let mut s = String::from("lo");
        s.push('l');
}
```

Concatenando com o operador + e format!

- Após a concatenação, s1 não estará mais disponível. s3 terá “Hello, world!”.

- O operador + utiliza uma assinatura que parece ser:

```
fn add(self, s: &str) -> String {
```

- Por isso na operação +, passamos uma string (s1) e uma referência &string (&s2).

- A função utiliza um &str e não &string. Mas o exemplo compila, por conta que o compilador do rust realiza essa conversão.

Exemplo:

```
fn main() {
    let s1 = String::from("Hello, ");
    let s2 = String::from("world!");
    let s3 = s1 + &s2;
    // note s1 has been moved here
    // and can no longer be used
}
```

Concatenando com o operador + e format!

- O exemplo 1, mostra a concatenação de diversas strings.
- O exemplo 2, mostra a mesma concatenação, utilizando o macro format!, que funciona como println!, porém ele retorna uma string em vez de imprimir.
- O macro format! utiliza referências, para não adquirir nenhum ownership de nenhum parâmetro.

Exemplo 1:

```
fn main() {
        let s1 = String::from("tic");
        let s2 = String::from("tac");
        let s3 = String::from("toe");

        let s = s1 + "-" + &s2 + "-" + &s3;
}
```

Exemplo 2:

```
fn main() {
        let s1 = String::from("tic");
        let s2 = String::from("tac");
        let s3 = String::from("toe");

        let s = format!("{s1}-{s2}-{s3}");
}
```

Acessando strings por index

- Em muitas linguagens de programação é possível acessar caracteres da string, utilizando index.

Exemplo:

```
fn main() {
    let s1 = String::from("hello");
    let h = s1[0];
}
```

- Se tentarmos realizar isso em Rust, teremos um erro.

```
error[E0277]: the type `String` cannot be indexed by `{integer}`
 --> src/main.rs:3:13
  |
3 |   let h = s1[0];
  |           ^^^^^ `String` cannot be indexed by `{integer}`
  |
```

Bytes, valores escalares e agrupamentos de grafemas

- Se tivermos o código let hello = String::from("Hola"); a variável hello, terá tamanho 4.

- Se tivermos o código let hello = String::from("Здравствуйте"); a variável hello não terá tamanho 12, mas sim 24. Porque cada valor escalar do unicode ocupa 2 bytes nesse caso.

- Se pedíssemos para o Rust retornar a primeira letra de “Здравствуйте”, ele vai retorna 208 e não 3. Se for “Hola” que possui somente letras do latim, ele vai retornar 104. Esses números são os valores em UTF-8.

Bytes, valores escalares e agrupamentos de grafemas

- Existem 3 formas relevantes que o Rust vê as strings, que são em bytes, valores escalares e agrupamentos de grafemas.
- Dado a string “नमस्ते” em hindi escrita no script Devanagari.
- Em bytes seria representado por:

  [224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164, 224, 165, 135]

- Em valor escalar unicode: ['न', 'म', 'स', '्', 'त', 'े'] onde o quarto e sexto caracteres não fazem sentido por si só. Pois são diacríticos.
- Em agrupamentos de grafemas: ["न", "म", "स्", "ते"]
- Por conta dessa diferenças de representação da string e o fato do Rust não conseguir retornar a letra em (O(1)), como esperado. O Rust impossibilita acessar a string utilizando o index.

Slicing Strings

- Para realizar um particionamento da string, teremos que ser mais específico, por conta do seu tamanho em bytes não ser igual a quantidade de caracteres.

```
#![allow(unused)]
fn main() {
    let hello = "Здравствуйте";

    let s = &hello[0..4];
}
```

- s será &str, que terá os 4 primeiros bytes, ou seja, “Зд”

- Se tentarmos utilizar &hello[0..1], o programa irá crashar em tempo de execução.

Métodos de repetição sobre string

- Ao operar com strings, devemos ser específicos, se deve ser retornado um byte ou um caracter.
- O Exemplo 1, utilizando .chars() terá como resultado “З д”
- O Exemplo 2, utilizando .bytes() terá como resultado “208 151 208 180”
- A biblioteca padrão do Rust não consegue realizar o mesmo para agrupamentos de grafemas, devido a complexidade, mas existem crates para isso.

Exemplo 1:

```
#![allow(unused)]
fn main() {
    for c in "Зд".chars() {
        println!("{c}");
    }
}
```

Exemplo 2:

```
#![allow(unused)]
fn main() {
    for b in "Зд".bytes() {
        println!("{b}");
    }
}
```

Armazenando chaves com valores associados com hashmap

- O tipo HashMap<K, V> recebe um chave do tipo K e um valor do tipo V.

- Em outras linguagens essa estrutura é conhecido como hash, map, objeto, tabela hash, dicionário ou array associativo.

- Hashmap é útil para procurar dados sem utilizar o índice, assim como vetores, porém com uma chave de qualquer tipo.

Criando um hashmap

- Por exemplo, temos um jogo onde com a chave, o nome da equipe, é possível obter a sua pontuação.
- Podemos criar um HashMap vazio utilizando new.
- Podemos adicionar elementos ao HashMap utilizando insert.

Exemplo:

```
fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert(
        String::from("Blue"),
        10
    );
    scores.insert(
        String::from("Yellow"),
        50
    );
}
```

Criando um hashmap

- Assim como vetores, HashMap tem seus dados armazenados em heap.
- Todas suas chaves devem ser do mesmo tipo.
- Todos os valores devem ser do mesmo tipo.

Exemplo:

```
fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert(
        String::from("Blue"),
        10
    );
    scores.insert(
        String::from("Yellow"),
        50
    );
}
```

Acessando os valores de um hashmap

- Nesse caso, o time Blue tem 10 pontos.
- O método get retorna Option<&V>.
- Nesse caso, get irá retornar 10. Caso get procure um time invalido, ele vai retornar none.
- Esse programa lida com o option, copiando seu valor, em vez de pegar a referência dele.
- Então unwrap_or(0) define score como 0, se a chave não existir.

Exemplo:

```
fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert(
        String::from("Blue"), 10);
    scores.insert(
        String::from("Yellow"), 50);

    let team_name = String::from("Blue");
    let score =
        scores.get(&team_name)
        .copied().unwrap_or(0);
}
```

Acessando os valores de um hashmap

- Assim como vetores, podemos utilizar um loop for para ler cada chave e seu valor no hashmap.

- O exemplo ao lado tem como output:

  Yellow: 50
  Blue: 10

```
fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);

    for (key, value) in &scores {
        println!("{key}: {value}");
    }
}
```

HashMap e Ownership

- Tipo que implementam o traço copy, por exemplo i32, são copiados para o hashmap.
- Valores de propriedade, por exemplo String, são movidos para o hashmap e o hashmap se torna dono desses valores.
- Por isso, após o insert, não conseguimos mais utilizar field_name e field_value.
- Se utilizarmos uma referência para o insert, os valores não serão movidos.

Exemplo:

```
fn main() {
    use std::collections::HashMap;

    let field_name =
        String::from("Favorite color");
    let field_value =
        String::from("Blue");

    let mut map = HashMap::new();
    map.insert(field_name, field_value);

    // field_name and field_value are invalid
    // at this point, try using them and
    // see what compiler error you get!
}
```

Atualizando um hashmap

- Todas as chaves devem ser únicas, diferentemente dos valores, que podem ser repetidos.

- Se utilizarmos insert para um valor já existente, o seu valor será substituído. Como mostra o exemplo ao lado.

Exemplo:

```
fn main() {
    use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Blue"), 25);

    println!("{:?}", scores);
}
```

Atualizando um hashmap

- Podemos adicionar um valor, somente se a chave não existir, como mostra o exemplo ao lado.
- or_insert é definido para retornar uma referência mutável ao valor para a chave correspondente se essa chave existir.
- Se não existir, insere o parâmetro como o novo valor para esta chave e retorna uma referência mutável.
- Nesse caso, no final Blue terá valor 10 e Yellow 50.

Exemplo:

```
fn main() {
        use std::collections::HashMap;

        let mut scores = HashMap::new();
        scores.insert(String::from("Blue"), 10);

        scores.entry(String::from("Yellow"))
        .or_insert(50);

        scores.entry(String::from("Blue"))
        .or_insert(50);

        println!("{:?}", scores);
}
```

Atualizando um hashmap

- Podemos procurar o valor de uma chave e depois atualizá-lo com base no valor antigo.
- O código ao lado conta quantas vezes cada palavra aparece em um texto.
- O método split_whitespace, retorna o texto, separado por espaço.
- or_insert retorna uma referência mutável para o valor da chave especificada, que é armazenada em count.
- Desreferencia count, com “*”. Permitindo a referência mutável sair do escopo no final do loop

Exemplo:

```
fn main() {
        use std::collections::HashMap;

        let text = "hello world wonderful world";

        let mut map = HashMap::new();

        for word in text.split_whitespace() {
                let count = map.entry(word).or_insert(0);
                *count += 1;
        }

        println!("{:?}", map);
}
```

Funções hashing

- Por padrão hashmap utiliza SipHash, que é uma função de hash que proporciona segurança a ataques de de Negação de Serviço (DoS).
- Não é a implementação mais rápida, mas a segurança compensa essa perda de desempenho.
- Podemos abrir mão dessa segurança, especificando a utilização da função hash BuildHasher.
- Falaremos sobre traits no capítulo 10.
- Não é preciso implementar um Hash do zero, pois o crates.io possue diversos algoritmos de hash feito por outros usuários.

Referências

Slides Baseados no Capítulo 8 do Rust Book
https://doc.rust-lang.org/book/ch08-00-common-collections.html