Rust 入门导学（2）

Mike Tang
mike@cdot.network
2020-8

本节课目的

1. 了解 Rust 语言的所有权与生命周期的概念
2. 学习 Rust 语言的 Trait
3. 学习 Rust 语言的泛型
4. 学习 Rust 语言的迭代器
5. 学习 Rust 语言的宏

Ownership 和 Borrowing

- 由字符串的示例，我们自然引出了两个概念： Ownership 和 Borrowing 。
- Rust 基本思维模型：要分清你对一个东西（资源）是否拥有所有权，或者只是借用状态?

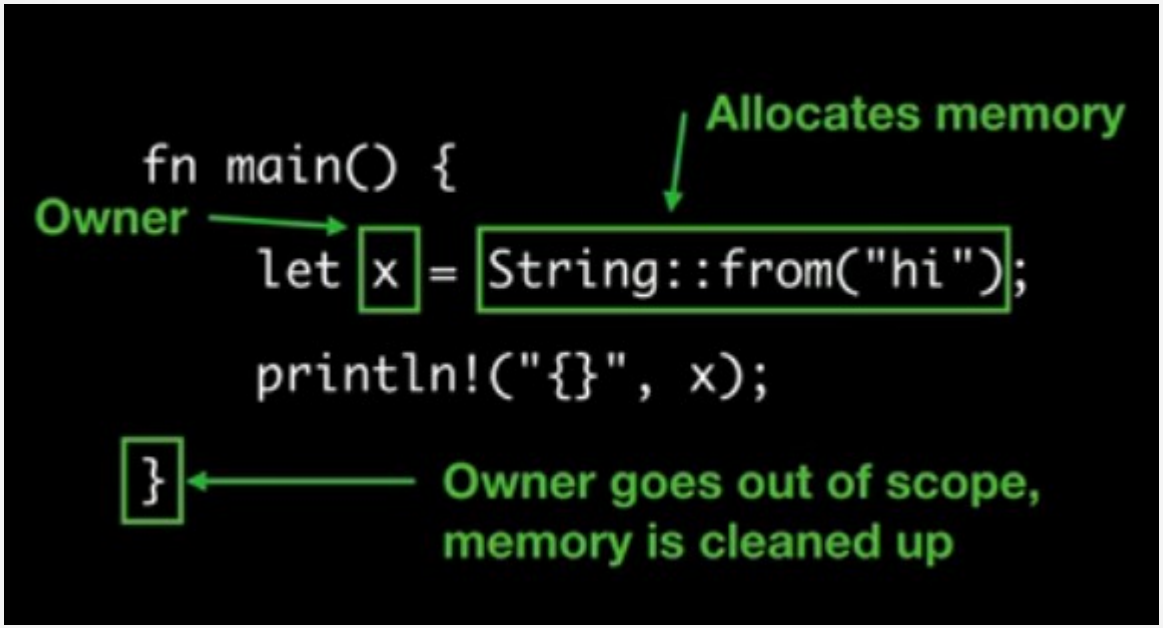
以下示例来自于：《 A Programming Language for the Next 40 Years: (Rust) 》 by Carol Nichols

https://matthew.krupczak.org/2019/09/27/a-programming-language-for-the-next-40-years-rust/

所有权示例 1

```
fn main() {
    let x = String::from("hi");
    println!("{}", x);
}
```

所有权示例 1


Allocates memory
fn main() {
Owner
let x = String::from("hi");
println!("{}", x);
}
Owner goes out of scope,
memory is cleaned up

所有权示例 1

```
fn main() {
    let x = String::from("hi");
    let y = x;
    println!("{}", x);
}
```

Moves ownership

所有权示例 1

```
error[E0382]: borrow of moved value: `x`

fn main() {

    let x = String::from("hi");

    let y = x;
            - value moved here
    println!("{}", x);
                   ^ value borrowed
}                    here after move
```

所有权示例 1

```
fn main() {
    let x = String::from("hi");
    let y = &x; ← Immutable borrow
    println!("{}", x);
}
```

```
fn main() {
    let x = String::from("hi");
    let y = &x;
    println!("{}", x);
    println!("{}", y);
}
```

所有权示例 2

```
fn main() {
    let y = {
        let x = String::from("hi");
        &x
    };
    println!("{}", y);
}
```

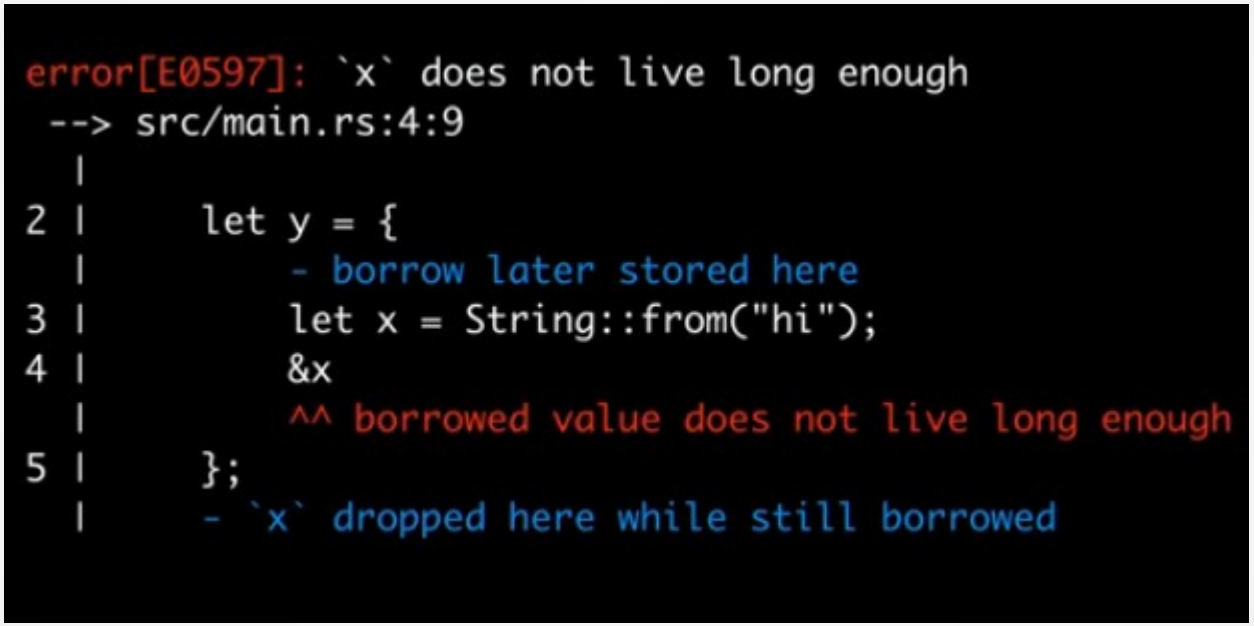
所有权示例 2

```
fn main() {
    let y = {
        let x = String::from("hi");
        &x
    };
    println!("{}", y);
}
```

Returning a reference

x is cleaned up

所有权示例 2



```
error[E0597]: `x` does not live long enough
 --> src/main.rs:4:9
  |
2 |     let y = {
  |         - borrow later stored here
3 |         let x = String::from("hi");
4 |         &x
  |         ^^ borrowed value does not live long enough
5 |     };
  |     - `x` dropped here while still borrowed
```

生命周期

- 由上面的例子可以看出， Ownership 和 Borrowing 自动带出了 “生命周期” 的概念。
- RAII (Resource Acquisition Is Initialization) – 一种可精确计算资源和变量生命的分析机制
- 其实生命周期（ lifetime ）概念一直存在，但是在其它语言中，没有研究得这么显要和精细
- C 完全手动管理资源的生命周期， Java 完全交给 Gc 管理， Rust 通过 Ownership & Borrowing 这一套规则，走出了既不需要 Gc ，也不需要手动管理的第三条路
- 优势：兼具 Java 的安全（整体比 Java 更安全），和 C 的速度及低资源占用率

生命周期管理的实现

- 如何实现？编译器的黑魔法。精准分析，在适当的位置，自动插入析构 (drop) 指令 RAII
- 编译器不够聪明，所以在复杂的情况下，需要人为标注生命周期符号
- 编译器会越来越聪明（论 AI 技术在编译器中的应用）。人工标注的情况会逐渐变少（当然不太可能变为 0 ）

Trait

- Trait – 特征
- 抽象共同行为 – like 类型类（可以理解为比类型本身高一层抽象）
- 操作：给具体的某种类型实现某个 trait，实现这个共同行为
- 两种用法
  - 给某个泛型限定某个 trait，约束泛型的类型范围
  - 作为某个函数的参数或返回值，用以代表一大类的类型
- Trait 可组合： T: TraitA + TraitB+ TraitC 赋予泛型 T 三个 trait 中的方法，同时限制 T 为必须为同时实现这三个 trait 的类型（并与交的概念，仔细理解）
- Trait 的设计： Trait 用于建模，可用来给事物的行为分类，分成多个 Trait

定义一个 trait

```rust
pub trait Summary {
    fn summarize(&self) -> String;
}
```

为一个结构体实现这个 trait

```rust
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}
```

为另一个结构体实现这个 trait

```rust
pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

使用 trait 中实现的方法

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from(
        "of course, as you probably already know, people",
    ),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summarize());
```

Trait Bound 特征限定

- Trait Bound 必须与泛型配合，对泛型的可能类型集进行限定（缩小集合取值空间）

```rust
pub fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

```rust
pub fn notify<T: Summary>(item1: &T, item2: &T) {
```

```rust
pub fn notify(item1: &impl Summary, item2: &impl Summary) {
```

Generics 泛型

- 泛型是用来表示一种类型的代号，这种类型代号在具体使用的时候，会具化成具体的类型
- 常常与 trait 绑定一起使用
- Impl trait 表示法是另一种泛型的写法，很多时候更简洁
- 当泛型的取值类型集固定的时候， enum 是更好的替代

两个范型示例

```
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

```
pub enum Option<T> {
    None,
    Some(T),
}
```

Iterator 迭代器

- 迭代器用于对一个元素序列进行迭代（遍历）
- Rust Std 中，有迭代器的抽象的实现，适用于任何可供迭代的类型（且可按其规范自定义）
- 迭代器是 lazy 的，也就是具体的元素在被请求（被消费）的时候，才会被访问
- 迭代器可由 map/filter/fold 这一套方法进行变换，且并不消费迭代器本身
- for 循环能实现的功能，迭代器都能实现。忘掉传统的 for 循环，都使用迭代器，更安全，也有可能更高效

Std 标准库定义的 Iterator Trait

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // methods with default implementations elided
}
```

实现一个自己的迭代器

```rust
struct Counter {
    count: u32,
}

impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}
```

```rust
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        if self.count < 5 {
            self.count += 1;
            Some(self.count)
        } else {
            None
        }
    }
}
```

使用我们自己的迭代器

```
for item in Counter {}
```

宏

- Rust 中包含两大类宏：声明 (declarative) 宏和过程 (procedural) 宏
- 过程宏又分为三种
  - 自定义 #[derive] 宏，可用在 struct 和 enum 上
  - 属性 (Attribute-like) 宏，可作用在几乎所有条目上
  - 函数 (Function-like) 宏，接收一串 token 作为其参数，也就是说这个可以直接操作代码 token 了（把人升级为编译器，人肉编译器？强大到令人发指）

Substrate 中的宏 - 声明宏示例

https://github.com/paritytech/substrate/blob/master/frame/support/src/dispatch.rs#L277

```
277  #[macro_export]
278  macro_rules! decl_module {
279      // Entry point #1.
280      (
281          $(#[$attr:meta])*
282          pub struct $mod_type:ident<
283              $trait_instance:ident: $trait_name:ident
284              $( <I>, I: $instantiable:path $( = $module_default_instance:path )? )?
285          >
286          for enum $call_type:ident where origin: $origin_type:ty $(, $where_ty:ty: $
287              $( $t:tt )*
288          }
289      ) => {
```

Substrate 中的宏 - 声明宏示例

https://github.com/paritytech/substrate/blob/master/frame/support/src/debug.rs#L130

```
130  #[macro_export]
131  macro_rules! runtime_print {
132      ($($arg:tt)+) => {
133          {
134              use core::fmt::Write;
135              let mut w = $crate::debug::Writer::default();
136              let _ = core::write!(&mut w, $($arg)+);
137              w.print();
138          }
139      }
140  }
```

Substrate 中的宏 - 过程宏示例

https://github.com/paritytech/substrate/blob/master/primitives/sr-api/proc-macro/src/lib.rs#L111

```
111    #[proc_macro]
112    pub fn impl_runtime_apis(input: TokenStream) -> TokenStream {
113        impl_runtime_apis::impl_runtime_apis_impl(input)
114    }
```

Thank You

Q&A