The concept of Rust

Roman Zaynetdinov

Rust

Empowering everyone to build reliable and efficient software.

- Performance
- Reliability
- Productivity

Performance

Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.

Reliability

Rust’s rich type system and ownership model guarantee memory-safety and thread-safety — and enable you to eliminate many classes of bugs at compile-time.

Productivity

Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling — an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more.

Ownership

Ownership is Rust’s most unique feature, and it enables Rust to make memory safety guarantees without needing a garbage collector.

- Each value in Rust has a variable that’s called its owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

```
{                       // a is not valid here, it's not yet declared
    let a = 123;    // a is valid from this point forward

    // do stuff with a
}                       // this scope is now over, and a is no longer valid
```

```
{
    let s = String::from("hello"); // s is valid from this point forward

    // do stuff with s
}                                  // this scope is now over, and s is no
                                   // longer valid
```

```rust
let x = 5;
let y = x;
println!("{}", x);

// Output: 5
```

```
let s1 = String::from("hello");
let s2 = s1;
println!("{}, world!", s1);
```

```rust
let s1 = String::from("hello");
let s2 = s1;
println!("{}, world!", s1);
```

```
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:4:24
  |
2 | let s1 = String::from("hello");
  |     -- move occurs because `s1` has type `std::string::String`,
          which does not implement the `Copy` trait
3 | let s2 = s1;
  |          -- value moved here
4 | println!("{}, world!", s1);
  |                        ^^ value borrowed here after move
```

```rust
let s1 = String::from("hello");
let s2 = s1.clone();
println!("{}, world!", s1);

// Output: hello, world!
```

References and Borrowing

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

The Rules of References

- At any given time, you can have either one mutable reference or any number of immutable references.
- References must always be valid.

```
> webpack --display-error-details --config webpack/prod.config.js

[clean-webpack-plugin: /home/node/dist has been removed.
*** Error in `node': double free or corruption (fasttop):
0x00007f94b4002420 ***
Aborted
npm ERR! code ELIFECYCLE
npm ERR! errno 134
npm ERR! example@1.0.0 postinstall: `webpack --display-error-details
--config webpack/prod.config.js`
npm ERR! Exit status 134
```

Type System: No Nulls

```
public class HelloWorld {
     static enum Pet {
          Dog, Cat;

          String getName() {
               switch (this) {
                    case Dog:
                              return "I am a Dog";
                    case Cat:
                              return "I am a Cat";
                    default:
                              throw new RuntimeException("This could never happen :'(");
               }
          }
     }

     public static void main(String []args) {
          final Pet pet = Pet.Dog;
          System.out.println(pet.getName());
     }
}
```

```rust
enum Pet {
    Dog,
    Cat,
}

impl Pet {
    fn name(&self) -> String {
        match self {
            Pet::Dog => "I am a Dog".to_string(),
            Pet::Cat => "I am a Cat".to_string(),
        }
    }
}

fn main() {
    let pet = Pet::Dog;
    println!("{}", pet.name());
}
```

Type System: Scopes

```go
type Set struct {
    lock  sync.Mutex
    inner map[string]struct{}
}

func (s *Set) Update(d string) {
    s.lock.Lock()
    defer s.lock.Unlock()

    s.inner[d] = struct{}{}
}

func main() {
    s := Set{
      inner: make(map[string]struct{}),
    }
    s.Update("one")
}
```

```
struct Set {
    inner: Mutex<HashSet<String>>,
}

impl Set {
    pub fn update(&mut self, d: &str) {
        let mut data = self.inner.lock().unwrap();
        data.insert(d.to_string());
    }
}

fn main() {
    let mut s = Set {
        inner: Mutex::new(HashSet::new()),
    };
    s.update("one");
}
```

Type System: Thread Safety

```go
func main() {
    var wg sync.WaitGroup
    result := 1

    wg.Add(1)
    go func() {
      defer wg.Done()
      result += 2
    }()

    wg.Add(1)
    go func() {
      defer wg.Done()
      result *= 2
    }()

    wg.Wait()

    fmt.Println("Result", result)
}
```

```rust
fn main() {
    let mut result = 1;

    let handle = thread::spawn(|| {
        result += 2;
    });

    let handle2 = thread::spawn(|| {
        result *= 2;
    });

    handle.join();
    handle2.join();

    println!("Result {}", result);
}
```

```rust
fn main() {
    let mut result = 1;

    let handle = thread::spawn(|| {
        result += 2;
```

```
error[E0499]: cannot borrow `result` as mutable more than once at a time
  --> src/main.rs:11:33
   |
7  |       let handle = thread::spawn(|| {
   |                    -             -- first mutable borrow occurs here
   |  ___________________|
   | |
8  | |         result += 2;
   | |         ------ first borrow occurs due to use of `result` in closure
9  | |     });
   | |_____- argument requires that `result` is borrowed for `'static`
10 |
11 |       let handle2 = thread::spawn(|| {
   |                                    ^^ second mutable borrow occurs here
12 |           result *= 2;
   |           ------ second borrow occurs due to use of `result` in closure
```

The Rules of References

- At any given time, you can have either one mutable reference or any number of immutable references.
- References must always be valid.

```rust
fn main() {
    let result = Arc::new(Mutex::new(1));

    let handle = {
        let cloned = result.clone();
        thread::spawn(move || {
            let mut result = cloned.lock().unwrap();
            *result += 2;
        })
    };

    let handle2 = {
        // ... The same as for handle
         *result *= 2;
        // ...
    };

    handle.join();
    handle2.join();

    println!("Result {}", result.lock().unwrap());
}
```

Reality of Rust

- Annoying things:
  - Official error handling docs are not clear
  - A lot of flexibility and options
  - Some of the tooling and language features are not fully completed
- Great things:
  - Cargo
  - Type system
  - You can build anything
  - Community

References

- https://www.rust-lang.org/
- https://doc.rust-lang.org/book/
- https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html