PingCAP

Rust Quick Tutorial

Wenxuan Shi @ PingCAP

PingCAP

About Me

- Infrastructure Engineer @ PingCAP
- TiKV team

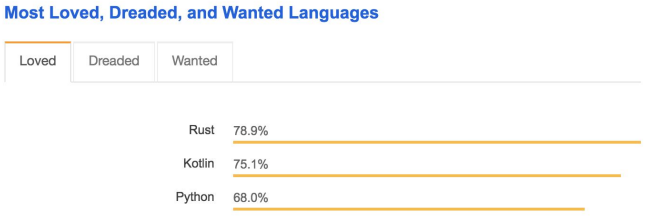
PingCAP

Rust

Wikipedia: “Rust is a systems programming language *sponsored by Mozilla* which describes it as a "safe, concurrent, practical language," supporting functional and imperative-procedural paradigms. Rust is syntactically similar to C++, but its designers intend it to provide better memory safety while still maintaining performance.”

PingCAP

The Most Loved Language



PingCAP

Rust Applications

- A good replacement for C / C++
- Performance critical applications
- Suitable for system programming
  - Databases
  - Web Servers
  - Browsers → Firefox Servo
  - Game Engines
  - Web Assembly 🔥
  - Operating Systems → CS140e
  - Compilers

PingCAP

Disadvantages

- Steep learning curve. Writing a A+B in Rust is much harder than in C++.
- Maybe too rigorous and too explicit for toy projects.
- Develop not fast as script languages.
- Community is not mature.
- Documentations and materials are limited.
- ……

PingCAP

Let's Getting Started

1. Use rustup to install Rust toolchains (compilers, docs, cargo, etc): https://rustup.rs/
2. Use cargo to manage your project:
    - Create project directory: `cargo new my_fancy_project`
    - Specify dependencies: `Cargo.toml`
    - Build: `cargo build`
    - Run tests: `cargo test`
    - Run application: `cargo run`

Rust A+B

```rust
fn main() {
    let mut line = String::new();
    ::std::io::stdin().read_line(&mut line).unwrap();

    let sum: i32 = line
        .split_whitespace()
        .map(|x| x.parse::<i32>().unwrap())
        .sum();

    println!("{}", sum);
}
```

PingCAP

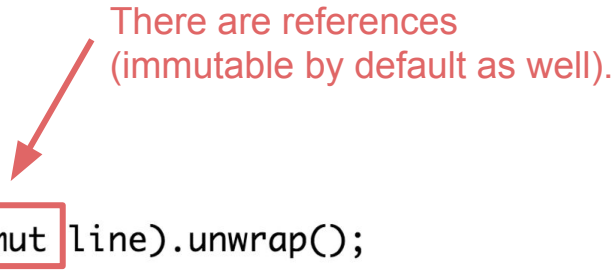
Rust A+B


```
fn main() {
    let mut line = String::new();
    ::std::io::stdin().read_line(&mut line).unwrap();

    let sum: i32 = line
        .split_whitespace()
        .map(|x| x.parse::<i32>().unwrap())
        .sum();

    println!("{}", sum);
}
```

Rust A+B


```
fn main() {
    let mut line = String::new();
    ::std::io::stdin().read_line(&mut line).unwrap();

    let sum: i32 = line
        .split_whitespace()
        .map(|x| x.parse::<i32>().unwrap())
        .sum();

    println!("{}", sum);
}
```

Rust A+B




```
fn main() {
    let mut line = String::new();
    ::std::io::stdin().read_line(&mut line).unwrap();

    let sum: i32 = line
        .split_whitespace()
        .map(|x| x.parse::<i32>().unwrap())
        .sum();

    println!("{}", sum);
}
```

Rust A+B

```rust
fn main() {
    let mut line = String::new();
    ::std::io::stdin().read_line(&mut line).unwrap();

    let sum: i32 = line
        .split_whitespace()
        .map(|x| x.parse::<i32>().unwrap())
        .sum();

    println!("{}", sum);
}
```

Functional programming style.

Rust A+B

```
fn main() {
    let mut line = String::new();
    ::std::io::stdin().read_line(&mut line).unwrap();

    let sum: i32 = line
        .split_whitespace()
        .map(|x| x.parse::<i32>().unwrap())
        .sum();

    println!("{}", sum);
}
```

Closure.

Rust A+B

Generic trait: Parse to what type?

```rust
fn main() {
    let mut line = String::new();
    ::std::io::stdin().read_line(&mut line).unwrap();

    let sum: i32 = line
        .split_whitespace()
        .map(|x| x.parse::<i32>().unwrap())
        .sum();

    println!("{}", sum);
}
```

```rust
pub fn parse<F>(&self) -> Result<F, <F as FromStr>::Err>
where
    F: FromStr,
```

```rust
impl FromStr for i8
  type Err = ParseIntError;

impl FromStr for char
  type Err = ParseCharError;

impl FromStr for f32
  type Err = ParseFloatError;

impl FromStr for i16
  type Err = ParseIntError;

impl FromStr for u16
  type Err = ParseIntError;

impl FromStr for isize
  type Err = ParseIntError;

impl FromStr for usize
  type Err = ParseIntError;

impl FromStr for f64
  type Err = ParseFloatError;

impl FromStr for i32
  type Err = ParseIntError;

impl FromStr for i128
  type Err = ParseIntError;
```

Rust A+B

```
fn main() {
    let mut line = String::new();
    ::std::io::stdin().read_line(&mut line).unwrap();

    let sum: i32 = line
        .split_whitespace()
        .map(|x| x.parse::<i32>().unwrap())
        .sum();

    println!("{}", sum);
}
```

Output to stdout.

Rust A+B

```
fn main() {
    let mut line = String::new();
    ::std::io::stdin().read_line(&mut line).unwrap();

    let sum: i32 = line
        .split_whitespace()
        .map(|x| x.parse::<i32>().unwrap())
        .sum();

    println!("{}", sum);
}
```

The macro enables string formatting at compile time.

Rust A+B

```rust
fn main() {
    let mut line = String::new();
    ::std::io::stdin().read_line(&mut line).unwrap();

    let sum: i32 = line
        .split_whitespace()
        .map(|x| x.parse::<i32>().unwrap())
        .sum();

    println!("{}", sum);
}
```

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

Rust's error handling style.

PingCAP

Rust Features

- zero-cost abstractions
- move semantics
- guaranteed memory safety
- threads without data races
- trait-based generics
- pattern matching
- type inference
- minimal runtime
- efficient C bindings

PingCAP

Ownership

- Rust’s most unique feature
- Achieve memory safe without GC
- Move semantics
- Rules:
    - 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.

PingCAP

Move Ownership

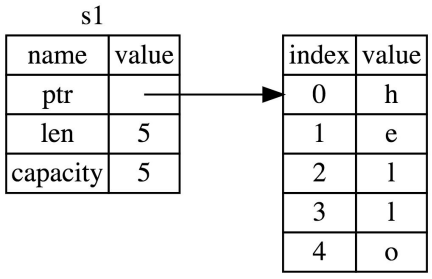
```
let v = vec![1, 2, 3];

let v2 = v;

println!("v[0] is: {}", v[0]);
```

```
error: use of moved value: `v`
println!("v[0] is: {}", v[0]);
                        ^
```

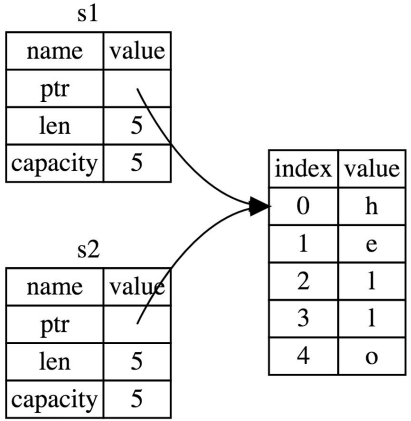
```
fn take(v: Vec<i32>) {
    // What happens here isn't important.
}

let v = vec![1, 2, 3];

take(v);

println!("v[0] is: {}", v[0]);
```

Move Ownership

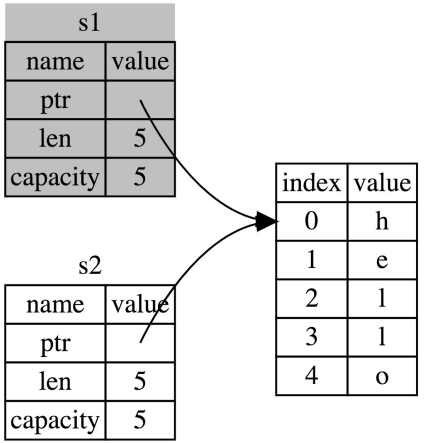
```
let s1 = String::from("hello");
let s2 = s1;
```



(1)



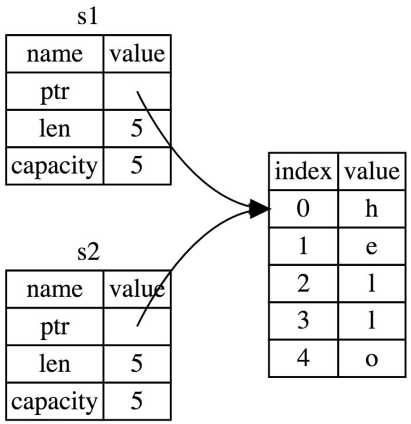
(2)



(3)

PingCAP

Typical C++ World Disaster (Double Free)




```
class String {
    char * ptr;
    int len;
    int capacity;
}
```

What if we..

```
free(s1);
free(s2);
```

This will not happen in Rust.

Protected by Rust Compiler ™.

PingCAP

Borrow

- Borrow by using reference operator.
- Rules:
  - At any given time, you can have *either* one mutable reference *or* any number of immutable references.
  - References must always be valid.

PingCAP

Typical C++ World Disaster: Data Race

Compile Error:

```
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;
```

```
let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM
```

This will not happen in Rust.

Protected by Rust Compiler ™.

PingCAP

Lifetime

- A variable's lifetime begins when it is created and ends when it is destroyed.
- Each reference is bounded to a lifetime.
- Ensures that all references are valid.
- Long lifetime references cannot be made from a shorter lifetime variable.
- Short lifetime references cannot be passed to a longer lifetime scope.

PingCAP

Typical C++ World Disaster: Dangling Pointer

Compile Error:

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s
}
```

Life time too short!

This will not happen in Rust.

Protected by Rust Compiler ™.

PingCAP

Typical C++ World Disaster: Dangling Pointer

Compile Error:

```
{
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

Life time too short!

This will not happen in Rust.

Protected by Rust Compiler ™.

PingCAP

Specify Lifetime

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

← *Cannot infer life time*

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

PingCAP

Learn More

Follow the official Rust book:

https://doc.rust-lang.org/stable/book/second-edition/

Thank You !

We are hiring!

hire@pingcap.com

Contact me:

breezewish@pingcap.com

PingCAP