# 理解Rust的所有权

所有权实现了既要保障内存安全又要无GC,运行时高性能的目标

苏林



# 自我介绍

- 前折800互联网研发团队负责人, 10余年一线研发经验
- 目前是多点Dmall技术Leader
- 具有多年的软件开发经验, 精通Ruby、Java、Rust等开发语言
- 同时也参与过Rust中文社区日报维护工作.

# 分享内容

- 什么是内存安全?
- 堆和栈
- 理解所有权
- 移动和借用
- 通过Rust代码, 加深上面的理解

# 什么是内存安全

- 悬垂指针

指向无效数据的指针（当我们了解数据在内存中如何存储之后，这个就很有意义）

- 重复释放

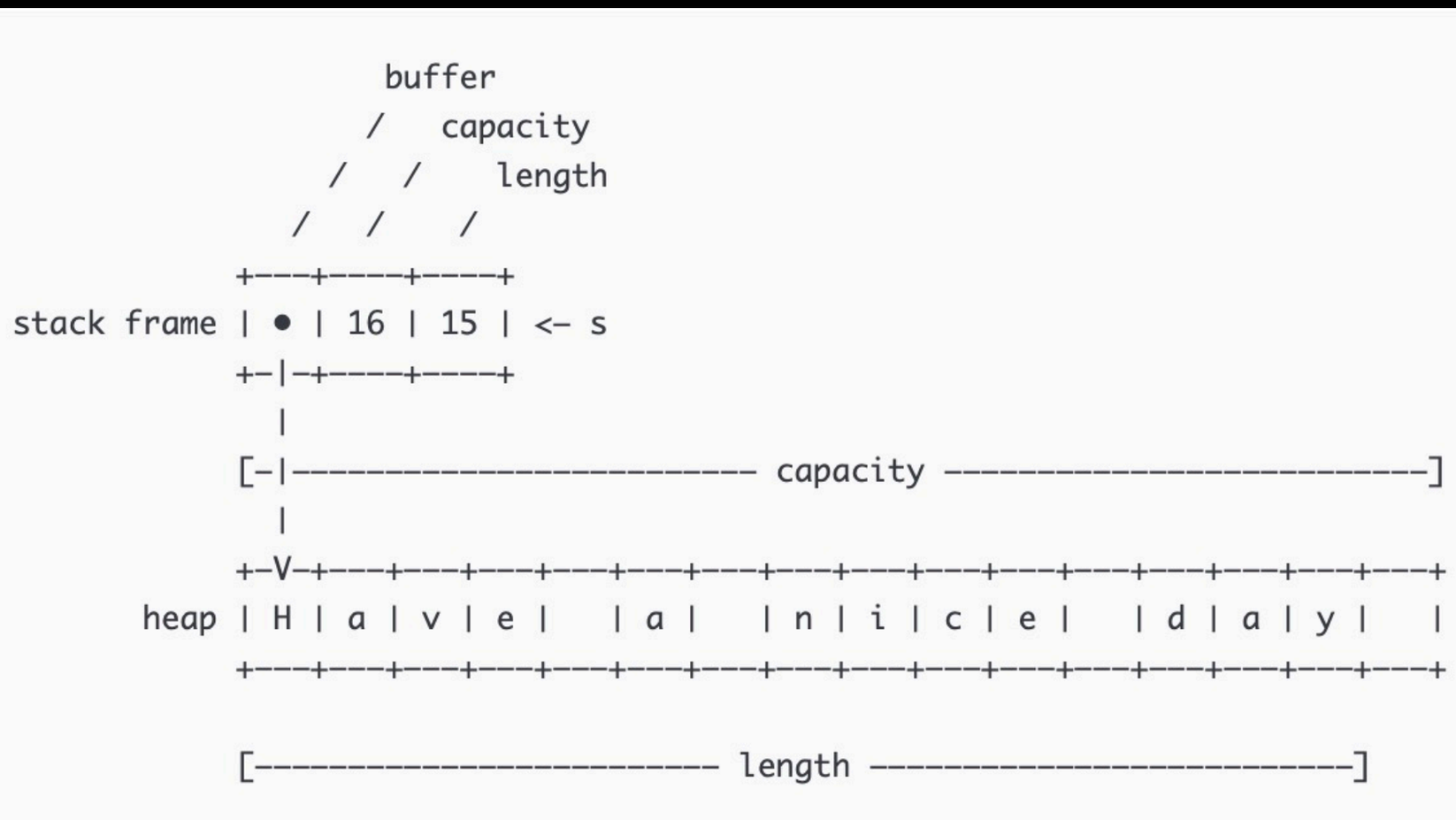
试图对同一块内存地址释放两次，这会导致“未定义行为”。

-

# 什么是内存安全

```
fn main() {
   {
      let s = String::from(“Have a nice day”);
   }
}
```

初始化的字符串通常是在内存中使用堆和栈进行表示的, 像下面这样:



# 什么是内存安全

Rust没有垃圾回收器，取而代之的是，它使用所有权和借用来解决保证内存安全的问题。当我们说Rust是内存安全的，我们是指，在默认情况下，Rust的编译器根本不允许我们写出内存不安全的代码。这是多么酷！

# 堆和栈

堆和栈都是内存的一部分但是以不同的数据数据结构来表示。栈是按照数据进来的顺序进行存储的，但是移除数据的时候是以相反的顺序（这样操作速度比较快）。堆更像是一个树结构，但是在进行数据读写时就需要多进行一些计算。

哪些数据存放在栈上，哪些数据存放在堆上，这取决于我们要处理的数据。在Rust里，任何固定大小(在编译期可以知道的大小)，比如机器整数(machine integers)，浮点数类型，指针类型和一些其他类型会被存储在栈上。动态的和“不确定大小(unsized)”数据被存储在堆上。这是因为这些不知道大小的类型，会经常地要么需要能够动态增长，要么需要在被析构时执行准确地清理工作（这不仅仅是从栈上弹出一个值）。

# 理解所有权

1、Rust里，每一个值都有一个决定其生命周期的唯一的所有者(owner).

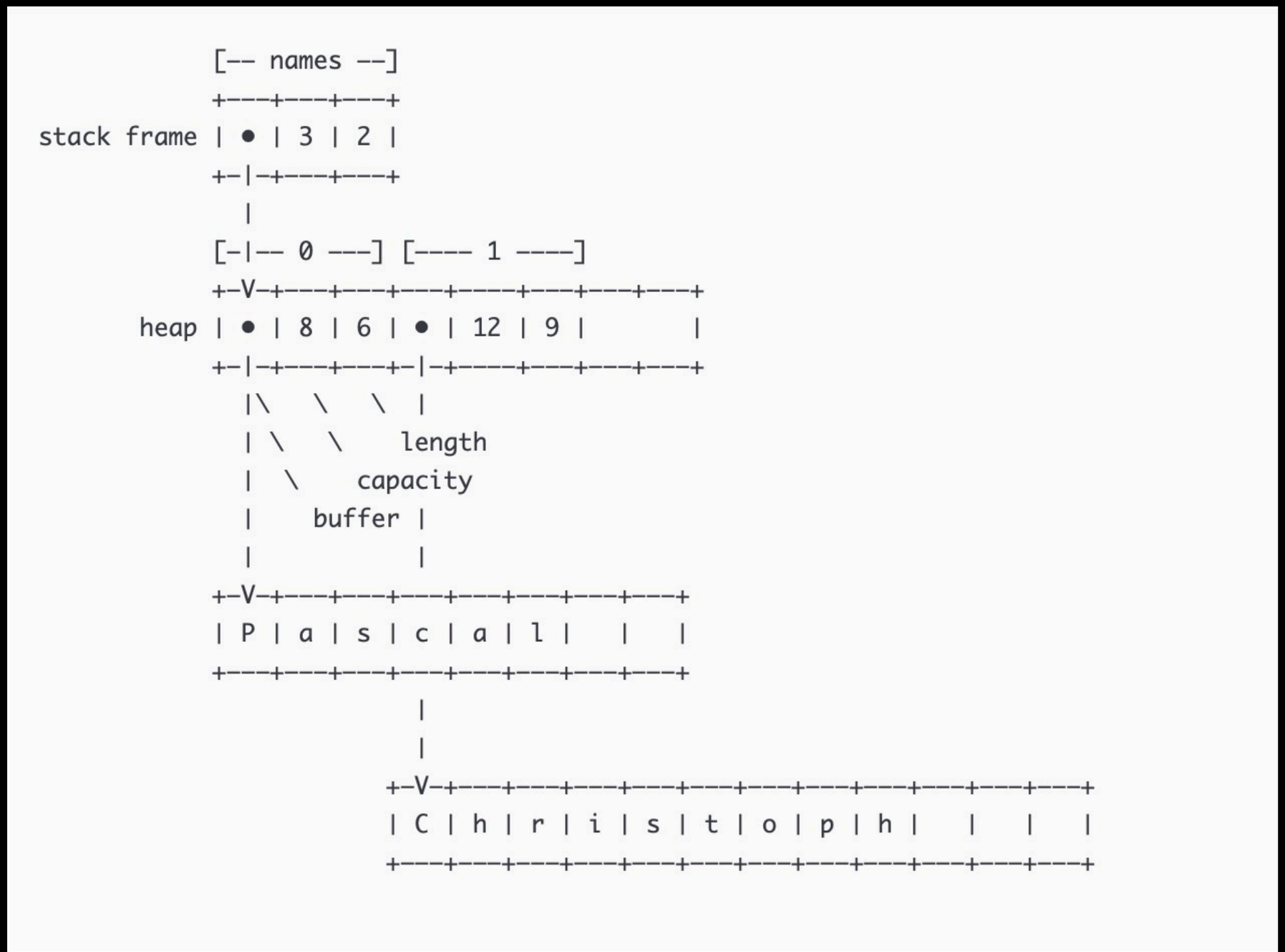
let s = "Have a nice day”.to_string();

假定我们有下面的函数：

```
fn greeting() {
  let s = "Have a nice day".to_string();
  println!("{}", s); // `s` is dropped here
}
```

# 理解所有权

let names = vec!["Pascal".to_string(), "Christoph".to_string()];

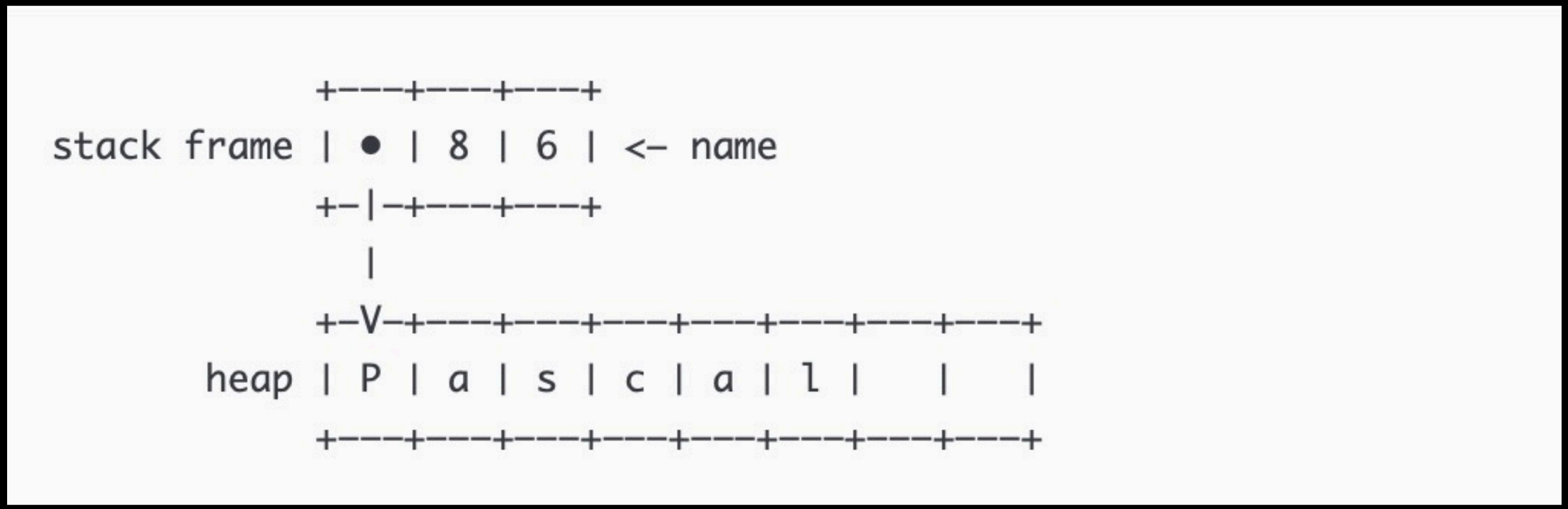


# 移动和借用

```
fn main() {
  {
    let name = “Pascal”.to_string();
    let a = name;
    let b = name;
  }
}
```

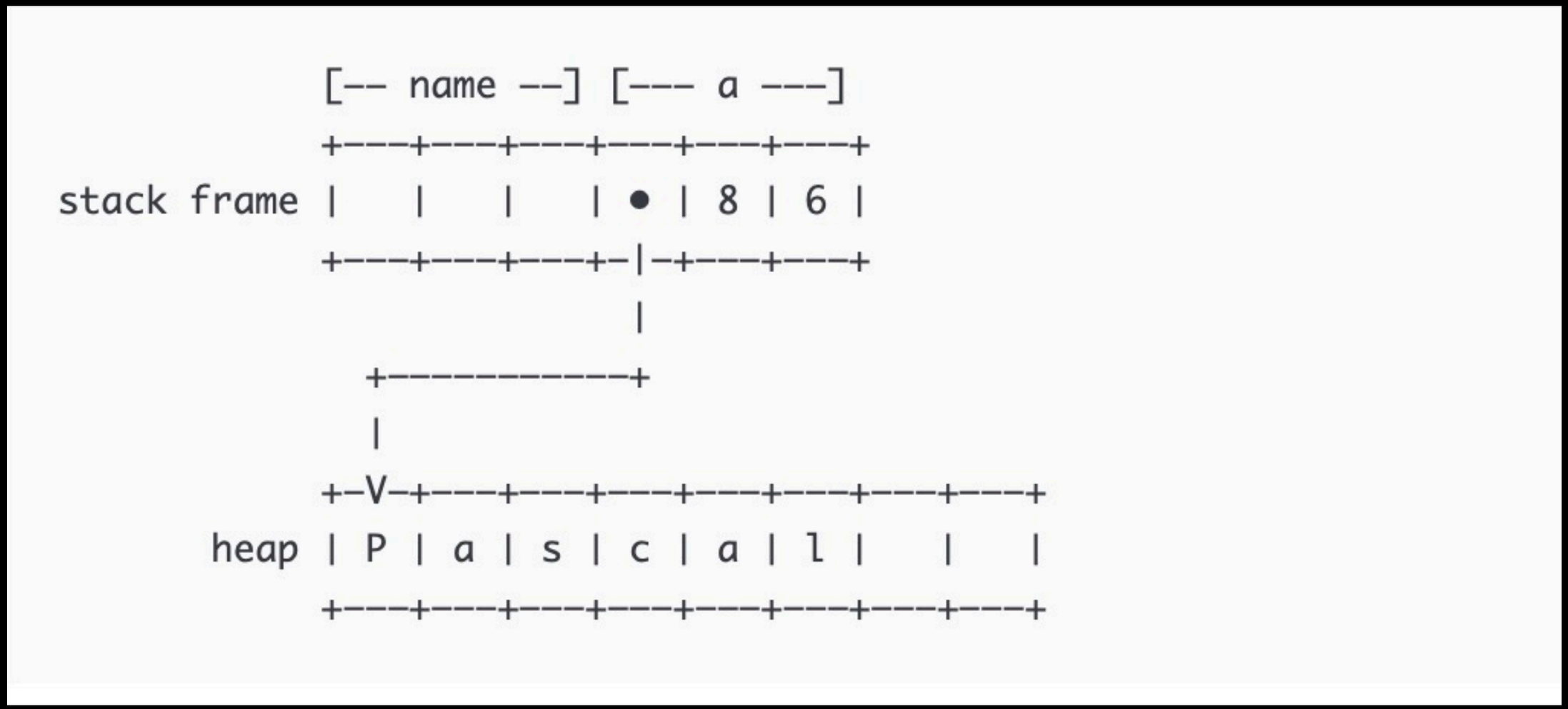
```
error[E0382]: use of moved value: `name`
 --> src/main.rs:4:11
  |
2 |   let name = "Pascal".to_string();
  |       ---- move occurs because `name` has type `std::string::String`, which does
not implement the `Copy` trait
3 |   let a = name;
  |           ---- value moved here
4 |   let b = name;
  |           ^^^^ value used here after move
```

# 移动和借用




但是，当我们把name的值赋值给a的时候，我们也把所有权交给了a，这时候的name是未初始化的。

# 移动和借用



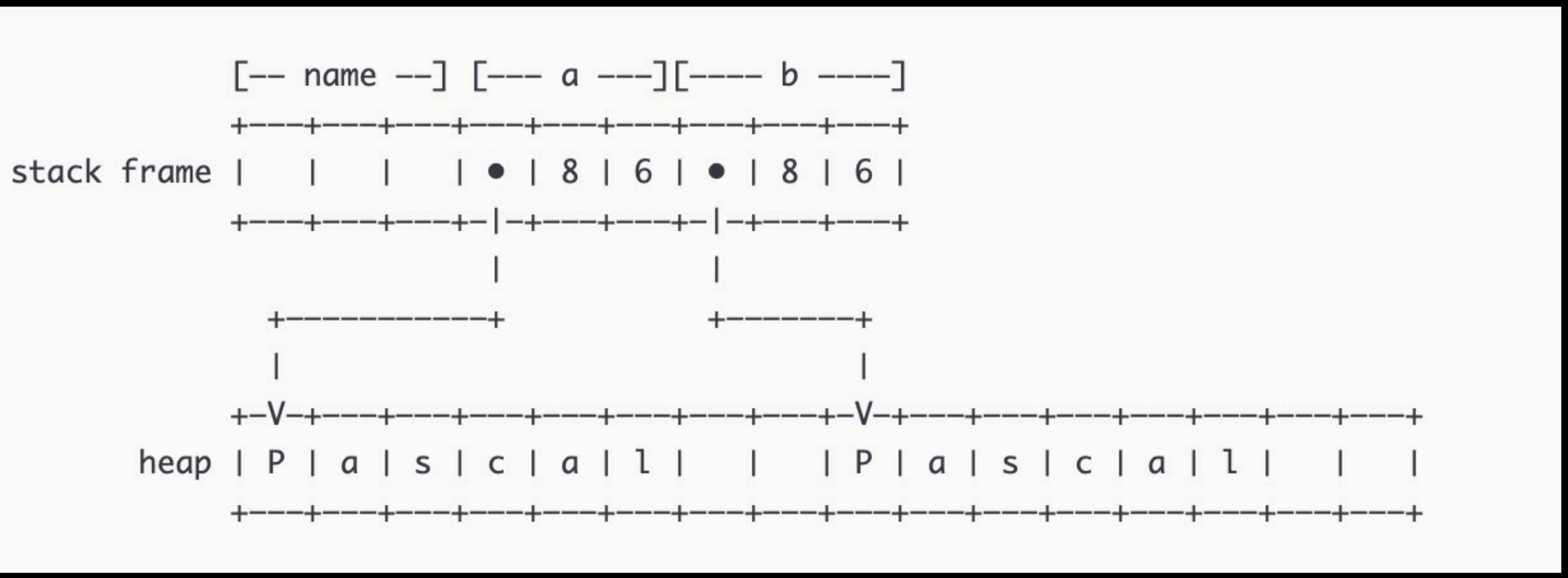

此时，表达式let b = name会产生一个错误就不足为奇了。这里很重要的一点是，所有的这种静态分析都是由编译器完成，而实际上并没有运行我们的代码。

# 移动和借用

```
let name = "Pascal".to_string();

let a = name;

let b = a.clone();
```



# 移动和借用

```
fn greet(name: String) {
  println!("Hello, {}!", name);
}
```

这个函数并不需要传入值的所有权才能输出。而且，这还会阻止我们对这个传入相同变量的函数进行多次调用：

```
let name = "Pascal".to_string();
greet(name);
greet(name); // Move happened earlier so this won't compile
```

# 移动和借用

```
fn greet(name: &String) {
  println!("Hello, {}!", name);
}
```

明确地说，我们可能会由于各种原因使用&str取而代之来设计这个API，但是这里不想让它变得太复杂，因为我们现在只需要一个&String。

```
let name = "Pascal".to_string();
greet(&name);
greet(&name);
```

# 移动和借用

```
let name = "Pascal".to_string();

let a = &name;

let b = &name;
```

使用上面的代码，name就不会失去所有权而a和b只是执行相同数据的指针。下面的表达也是一样的：

```
let name = "Pascal".to_string();

let a = &name;

let b = a;
```

# Rustlings项目上move_semantics的5道题

```rust
fn main() {
    let vec0 = Vec::new();

    let vec1 = fill_vec(vec0);

    println!("{} has length {} content `{:?}`", "vec1", vec1.len(), vec1);

    vec1.push(88);

    println!("{} has length {} content `{:?}`", "vec1", vec1.len(), vec1);
}

fn fill_vec(vec: Vec<i32>) -> Vec<i32> {
    let mut vec = vec;

    vec.push(22);
    vec.push(44);
    vec.push(66);

    vec
}
```

# Rustlings项目上move_semantics的5道题

```rust
fn main() {
    let vec0 = Vec::new();

    let mut vec1 = fill_vec(vec0);

    // Do not change the following line!
    println!("{} has length {} content `{:?}`", "vec0", vec0.len(), vec0);

    vec1.push(88);

    println!("{} has length {} content `{:?}`", "vec1", vec1.len(), vec1);
}

fn fill_vec(vec: Vec<i32>) -> Vec<i32> {
    let mut vec = vec;

    vec.push(22);
    vec.push(44);
    vec.push(66);

    vec
}
```

# Rustlings项目上move_semantics的5道题

```rust
fn main() {
    let vec0 = Vec::new();

    let mut vec1 = fill_vec(vec0);

    println!("{} has length {} content `{:?}`", "vec1", vec1.len(), vec1);

    vec1.push(88);

    println!("{} has length {} content `{:?}`", "vec1", vec1.len(), vec1);
}

fn fill_vec(vec: Vec<i32>) -> Vec<i32> {
    vec.push(22);
    vec.push(44);
    vec.push(66);

    vec
}
```

# Rustlings项目上move_semantics的5道题

```rust
fn main() {
    let vec0 = Vec::new();

    let mut vec1 = fill_vec(vec0);

    println!("{} has length {} content `{:?}`", "vec1", vec1.len(), vec1);

    vec1.push(88);

    println!("{} has length {} content `{:?}`", "vec1", vec1.len(), vec1);
}

// `fill_vec()` no longer takes `vec: Vec<i32>` as argument
fn fill_vec() -> Vec<i32> {
    let mut vec = vec;

    vec.push(22);
    vec.push(44);
    vec.push(66);

    vec
}
```

# Rustlings项目上move_semantics的5道题

```
8   fn main() {
9       let mut x = 100;
10      let y = &mut x;
11      let z = &mut *y;
12      *y += 100;
13      *z += 1000;
14      assert_eq!(x, 1200);
15  }
```

# 小结

这些只是冰山一角。关于数据的所有权，借用以及移动，还有很多东西需要考虑，但是希望这次公开课能够让你对Rust是如何保证内存安全的背后原理有一个基本的理解.

# QA环节

加群一起交流Rust & Datafuse

Datafuse

Rust语言中文社区

知数堂
zhishutang.com