Rust 语言核心概念：Rust 语言架构

极客时间

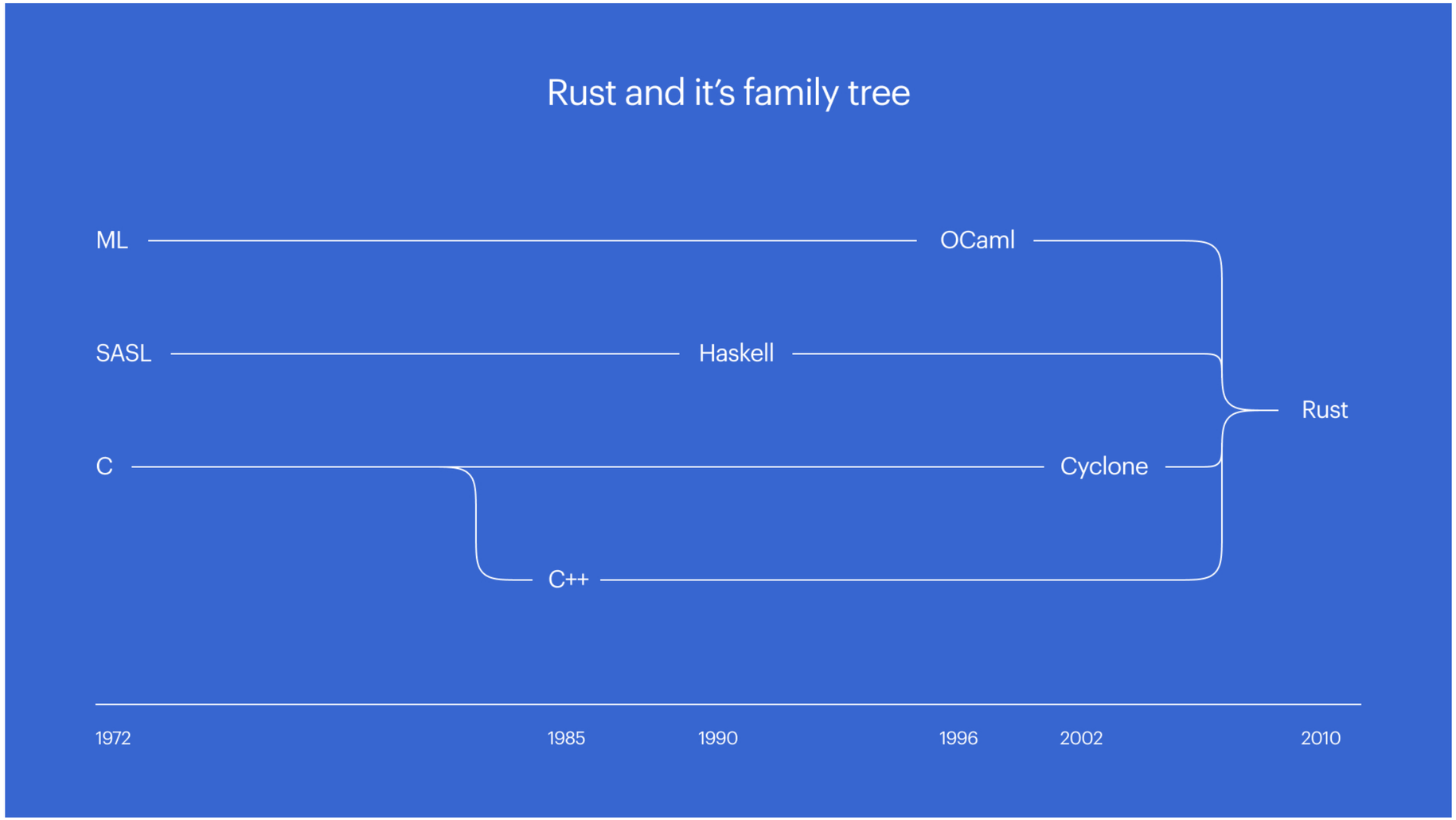
扫码试看/订阅

《张汉东的Rust实战课》视频课程

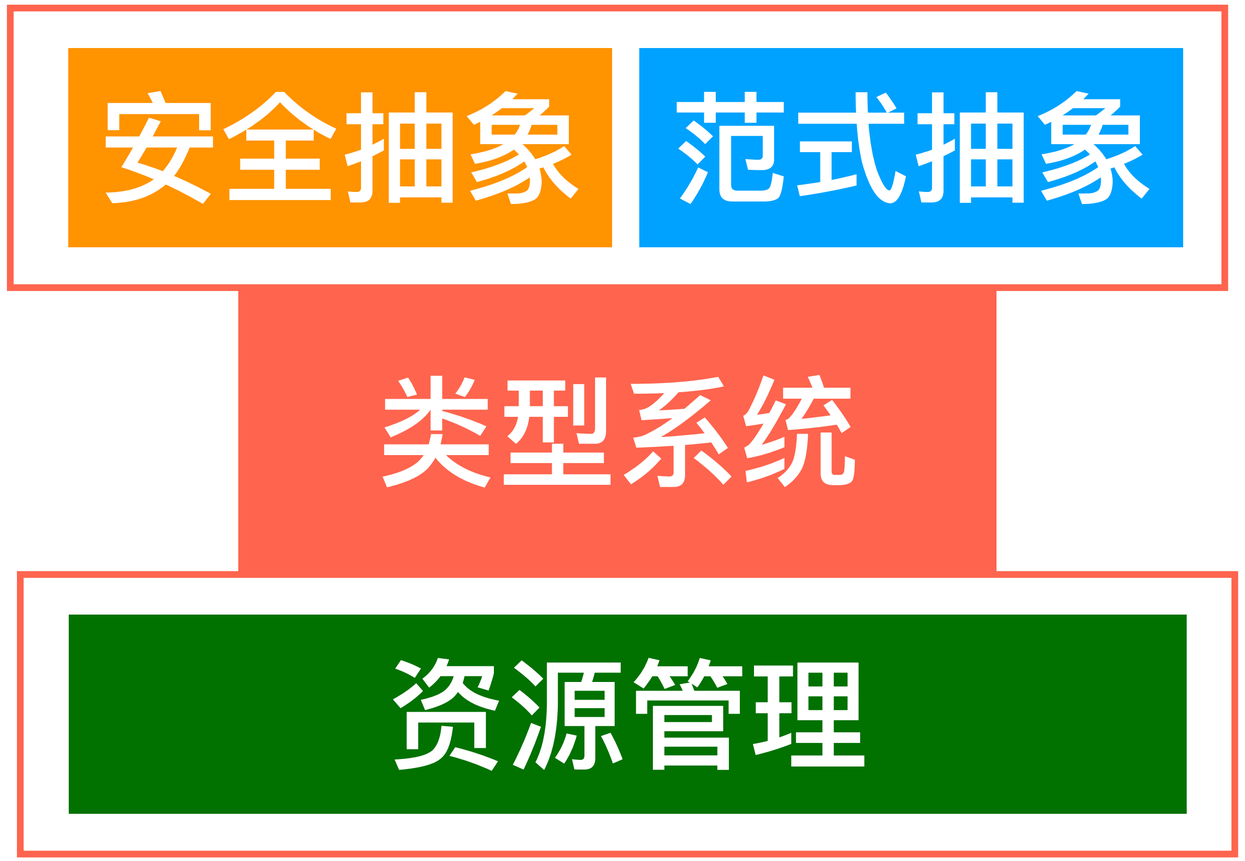
内容包括：

1. Rust 语言架构
2. Rust 语言核心概念介绍

Rust 语言架构


Rust and it's family tree
ML
OCaml
SASL
Haskell
Rust
C
Cyclone
C++
1972
1985
1990
1996
2002
2010

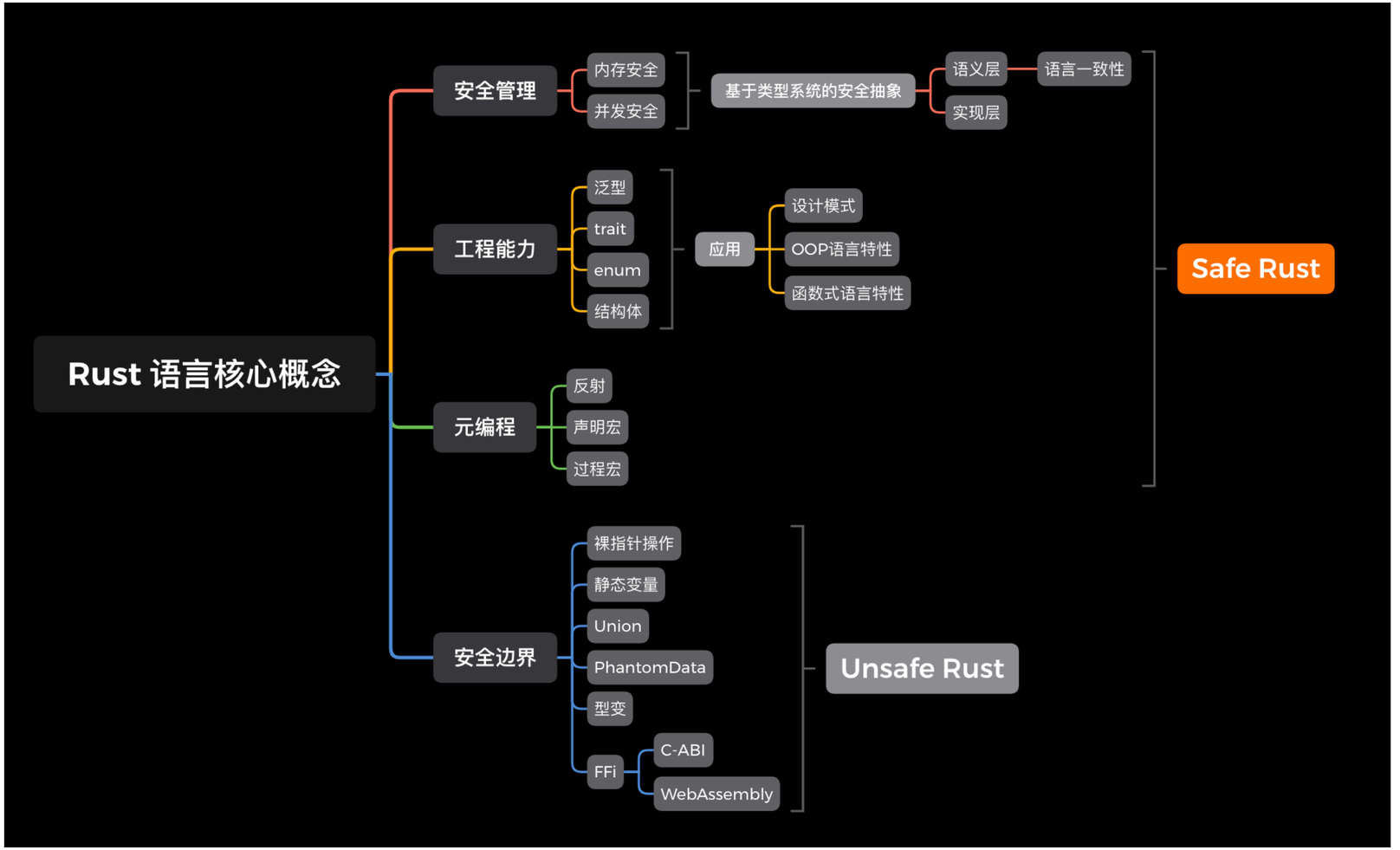
Rust 语言架构



Rust 语言概念核心介绍

1. 掌握所有权语义
2. 领略Rust 的工程能力
3. 掌握元编程能力
4. 正确认识 Unsafe Rust

Rust 语言概念核心介绍



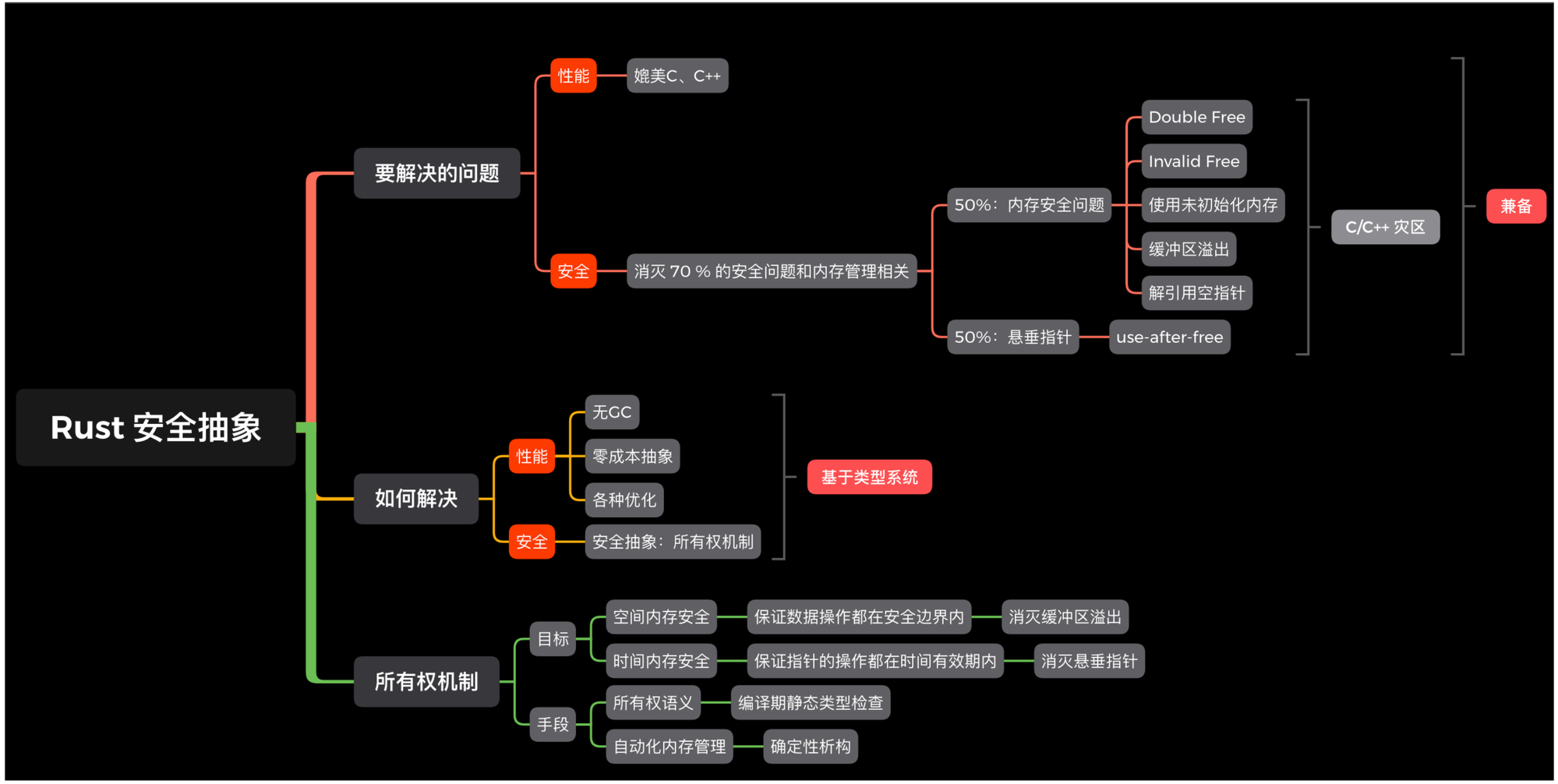
小结

Rust 语言核心概念：内存管理基础知识

内容包括：

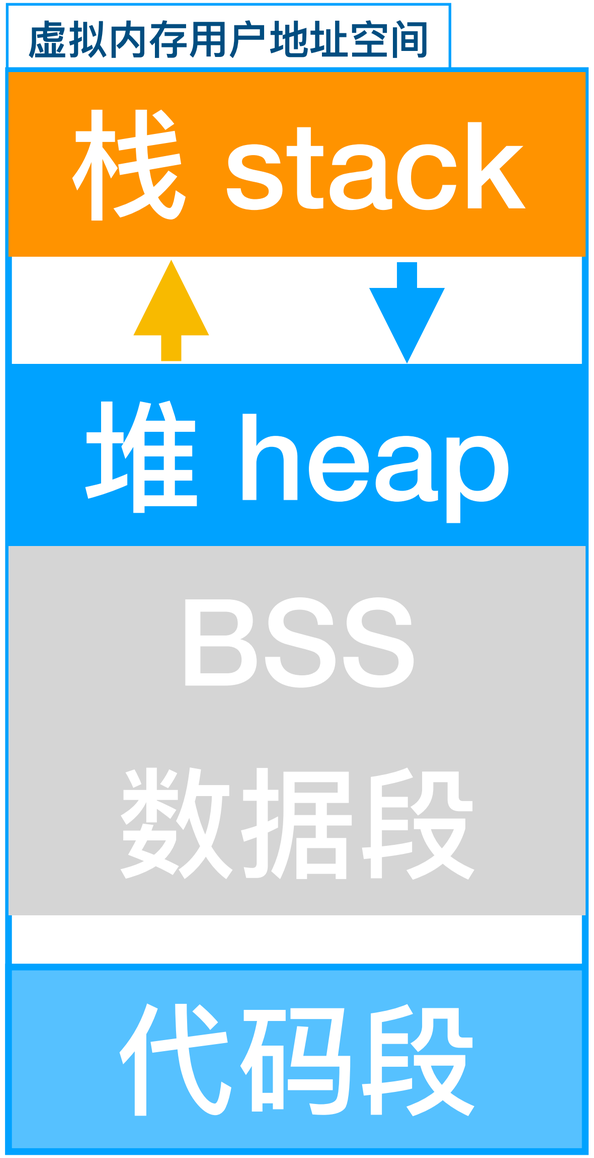
1. Rust 语言的安全抽象概要。
2. OS 内存管理通用知识介绍。
3. Rust 所有权机制介绍。

Rust 语言安全抽象概要



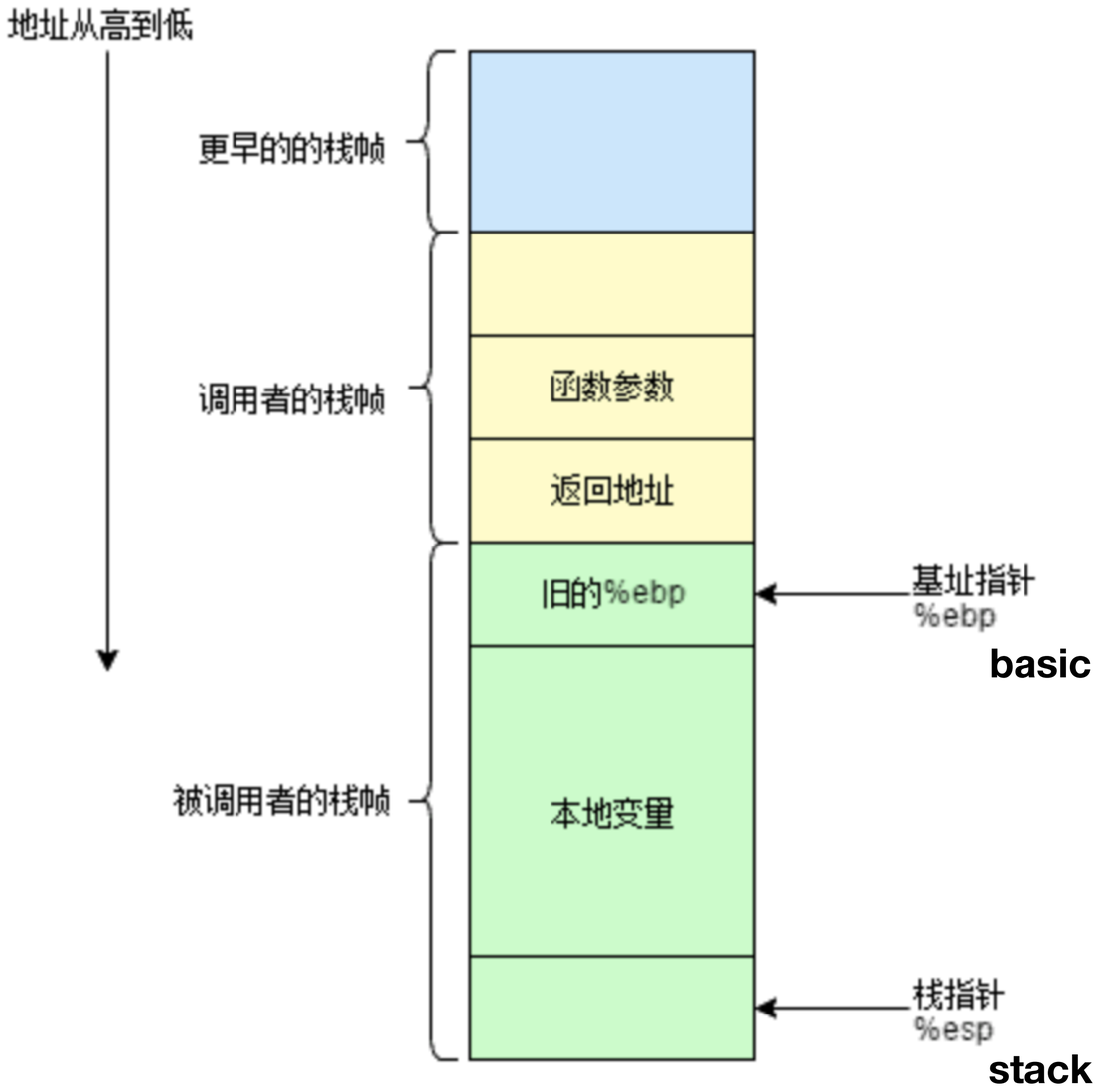
OS 内存管理通用知识

了解虚拟地址空间



OS 内存管理通用知识

了解函数调用栈

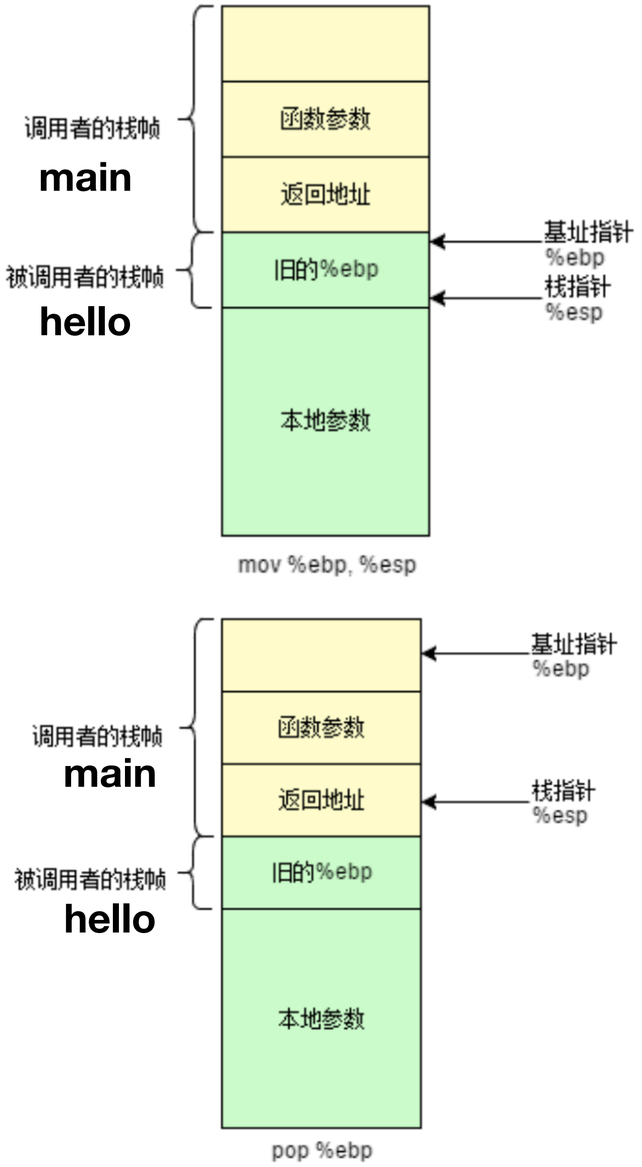


OS 内存管理通用知识

了解函数调用栈

```rust
// 被调用者 (callee)
fn hello(s: &str) {
    println!("Hello {:?}", s);
}

// 调用者 (caller)
fn main() {
    let s = "Rust";
    hello(s);  // print : "Hello Rust"
}
```



Rust 所有权机制

1. Rust 出现之前，其他编程语言的内存管理方式
2. Rust 基于所有权的安全内存管理模型

Rust 所有权机制

其他语言内存管理方式

1. C：纯手工管理内存
2. C++： 手工管理 + 确定性析构

缺乏安全抽象模型

3. GC语言： 垃圾回收

性能差

Rust 所有权机制

Rust 语言内存管理方式

1. 考虑性能： 借鉴 Cpp 的RAII 资源管理方式
2. 考虑安全：增加所有权语义

小结

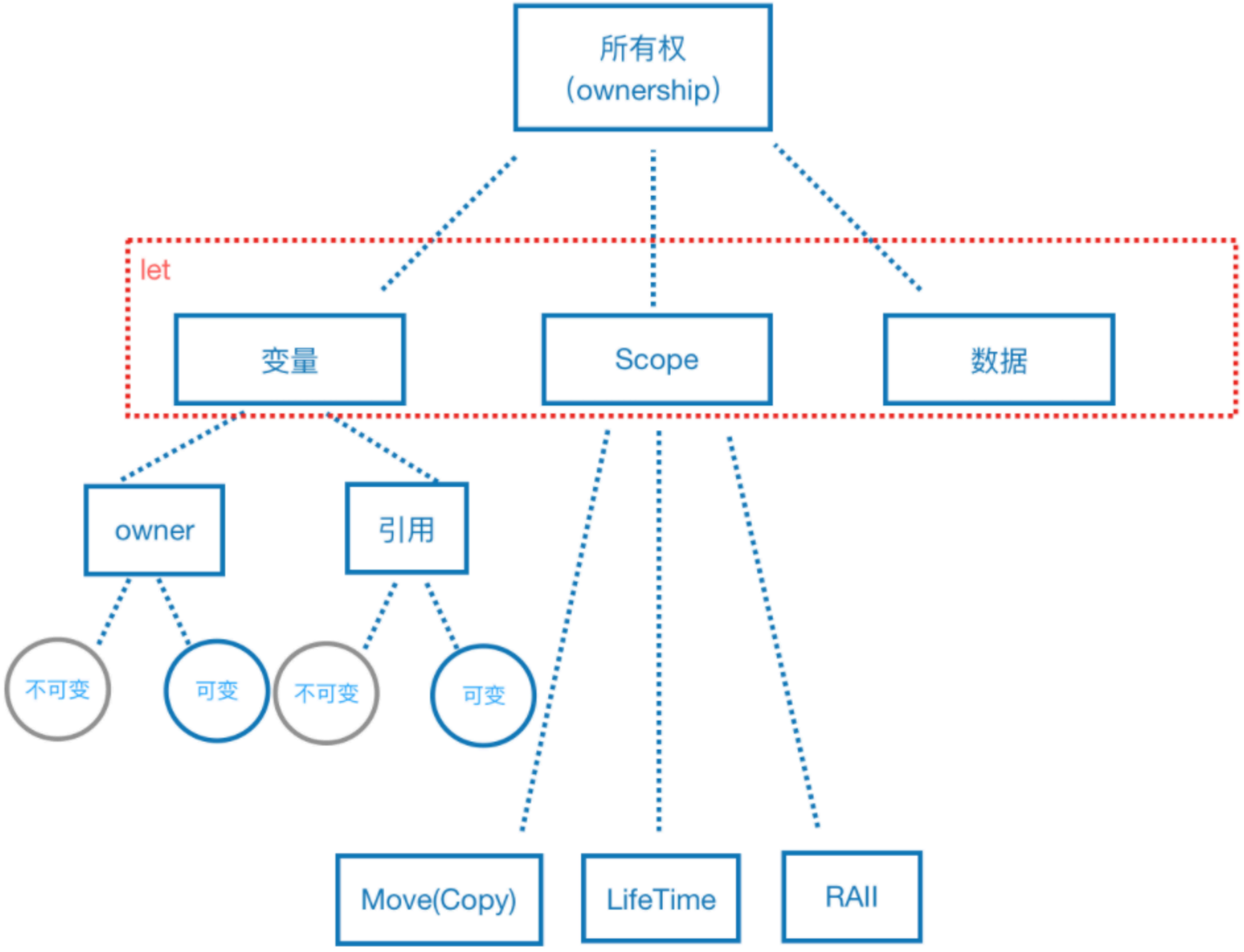
Rust 语言核心概念：安全管理之内存安全

内容包括：

1. Rust 语言所有权 之 语义模型
2. Rust 语言所有权 之 类型系统
3. Rust 语言所有权 之 内存管理
4. Rust 语言所有权之 所有权借用
5. Rust 语言所有权之 所有权共享

所有权之语义模型

```
let answer = "42";
```



所有权之语义模型

```rust
fn main(){
    let answer = "42";
    let no_answer = answer;
    println!("{:?}", answer);  // usable
    let answer = String::from("42");
    let no_answer = answer;
    println!("{:?}", answer);  // unusable
}
```

```
let _1: &str;
scope 1 {
    debug answer => _1;
    let _2: &str;
    scope 2 {
        debug no_answer => _2;
    }
}
```

所有权之类型系统

仿射类型（Affine Type）

子结构类型系统（Substructural Type Systems）

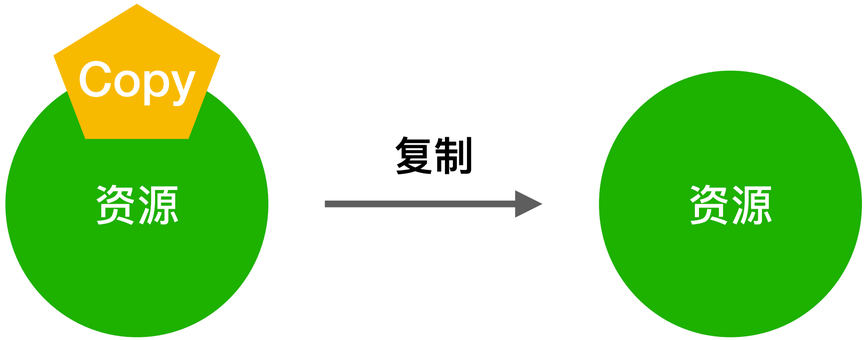
资源最多只能被使用一次

所有权之类型系统

移动语义（Move）

复制语义（Copy）

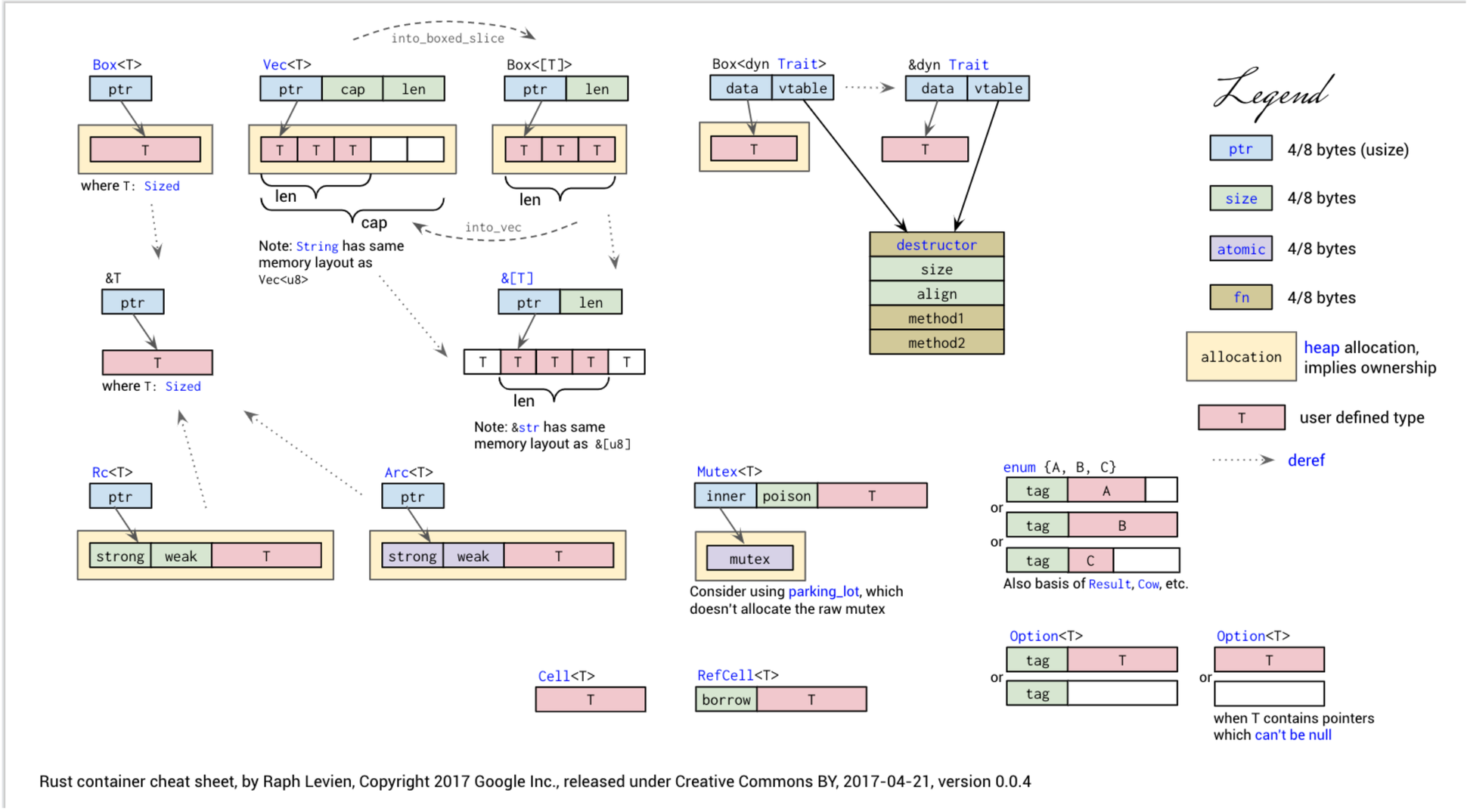
impl Copy for SomeT



所有权之内存管理

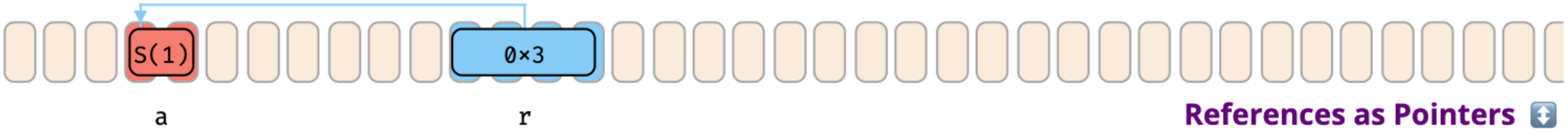
1. 默认存储数据到栈上
2. 利用栈来自动管理堆内存

所有权之内存管理



Rust container cheat sheet, by Raph Levien, Copyright 2017 Google Inc., released under Creative Commons BY, 2017-04-21, version 0.0.4

所有权之所有权借用

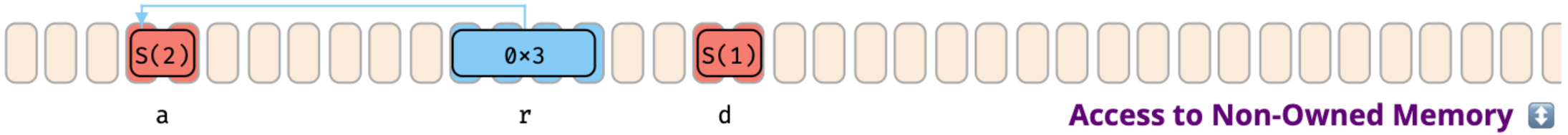



```
let a = S(1);
let r: &S = &a;
```

- A **reference type** such as `&S` or `&mut S` can hold the **location of** some `s`.
- Here type `&S`, bound as name `r`, holds *location of* variable `a` (`0x3`), that must be type `S`, obtained via `&a`.
- If you think of variable `c` as *specific location*, reference **`r` is a *switchboard for locations***.
- The type of the reference, like all other types, can often be inferred, so we might omit it from now on:

```
let r: &S = &a;
let r = &a;
```

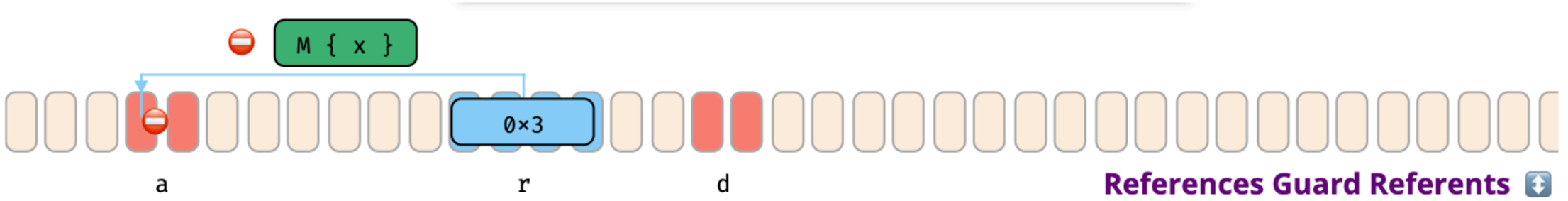
所有权之所有权借用




```
let mut a = S(1);
let r = &mut a;
let d = r.clone();  // Valid to clone (or copy) from r-target.
*r = S(2);          // Valid to set new S value to r-target.
```

- References can **read from** (`&S`) and also **write to** (`&mut S`) location they point to.
- The *dereference* `*r` means to neither use the *location of* or *value within* `r`, but the **location `r` points to**.
- In example above, clone `d` is created from `*r`, and `S(2)` written to `*r`.
    - Method `Clone::clone(&T)` expects a reference itself, which is why we can use `r`, not `*r`.
    - On assignment `*r = ...` old value in location also dropped (not shown above).

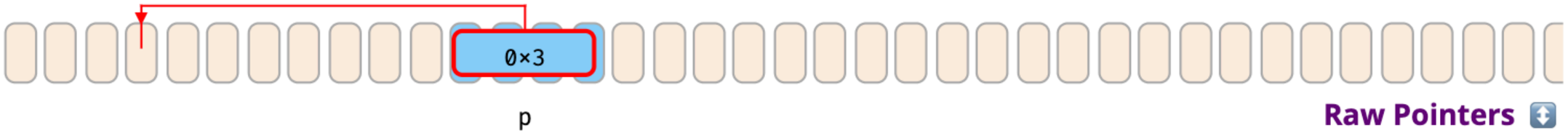
所有权之所有权借用




```
let mut a = ...;
let r = &mut a;
let d = *r;         // Invalid to move out value, `a` would be empty.
*r = M::new();      // invalid to store non S value, doesn't make sense.
```

- While bindings guarantee to always *hold* valid data, references guarantee to always *point to* valid data.
- Esp. `&mut T` must provide same guarantees as variables, and some more as they can't dissolve the target:
  - They do **not allow writing invalid** data.
  - They do **not allow moving out** data (would leave target empty w/o owner knowing).

所有权之所有权借用




```
let p: *const S = questionable_origin();
```

- In contrast to references, pointers come with almost no guarantees.
- They may point to invalid or non-existent data.
- Dereferencing them is `unsafe`, and treating an invalid `*p` as if it were valid is undefined behavior. ↓

所有权之所有权共享

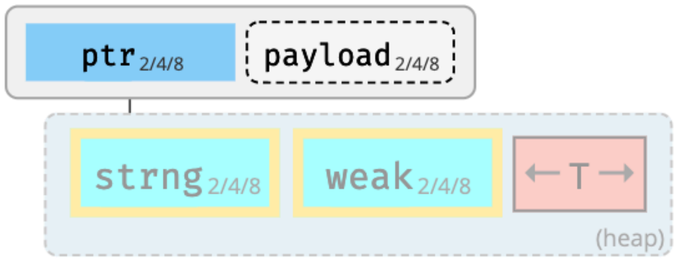
从语义层面把握语言一致性

1. Rust 中 Clone trait 在语义上表示：所有权共享
2. 具体实现层面是不一样的

所有权之所有权共享

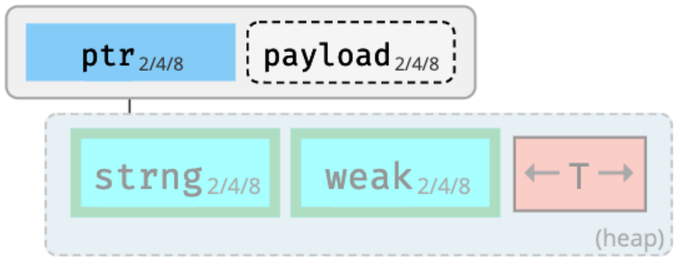
引用计数容器

Rc<T>



Share ownership of `T` in same thread. Needs nested `Cell` or `RefCell` to allow mutation. Is neither `Send` nor `Sync`.

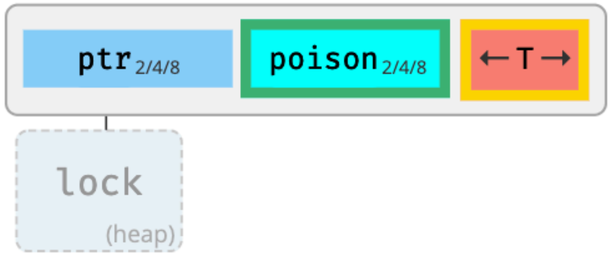
Arc<T>



Same, but allow sharing between threads IF contained `T` itself is `Send` and `Sync`.

多线程需要加锁

Mutex<T> / RwLock<T>



Needs to be held in `Arc` to be shared between threads, always `Send` and `Sync`. Consider using parking_lot instead (faster, no heap usage).

极客时间

扫码试看/订阅

《张汉东的Rust实战课》视频课程

小结