

Rust编程语言

核心优势和核心竞争力

庄晓立，北京，[QCon2016](#)

liigo@qq.com



新人分享新语言

2015年5月Rust语言刚刚发布1.0版本

2013年末我开始关注Rust，但尚未深入实践

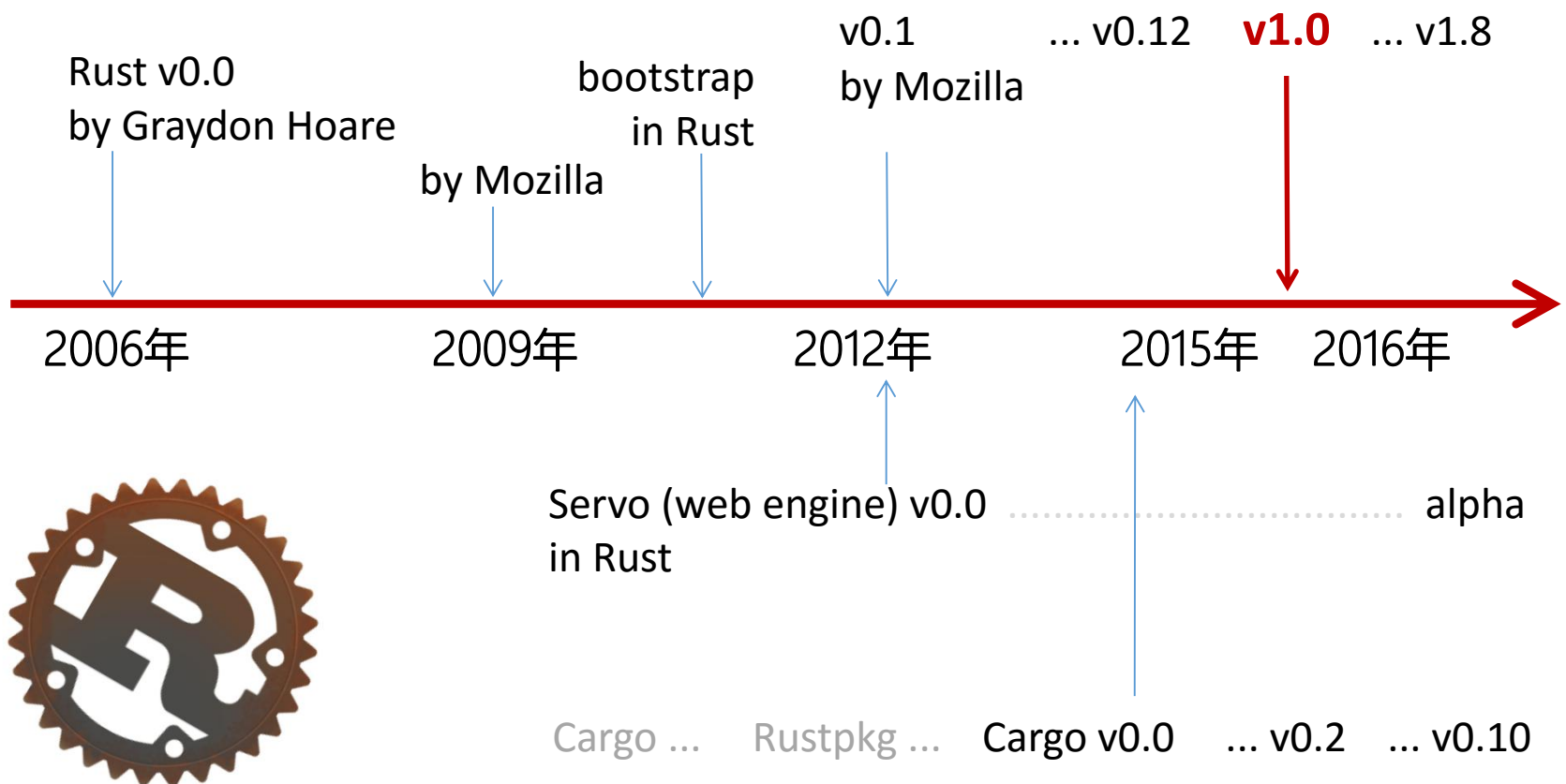
在重大场合正式演讲这是人生第一次

2016年春节至今认真准备讲稿近三个月

如有错误或纰漏，敬请谅解！

感谢臧秀涛主编盛情邀请！

Rust 安全、高效、并发的系统编程语言



系统编程+零运行时+内存安全

系统编程+零运行时+内存安全

（应用领域）

（运行效率）

（系统安全）

系统编程+零运行时+内存安全

系统编程

Systems Programming



WIKIPEDIA
The Free Encyclopedia

System programming

- The programmer will make assumptions about the hardware and other properties of the system that the program runs on, and will often exploit those properties, for example by using an algorithm that is known to be efficient when used with specific hardware.
- Usually a low-level programming language or programming language dialect is used that:
 - can operate in resource-constrained environments
 - is very efficient and has little runtime overhead
 - has a small runtime library, or none at all
 - allows for direct and "raw" control over memory access and control flow
 - lets the programmer write parts of the program directly in assembly language
- Often systems programs cannot be run in a debugger. Running the program in a simulated environment can sometimes be used to reduce this problem.

https://en.wikipedia.org/wiki/System_programming

系统编程

- 对硬件的控制（嵌入式, OS）
- 对系统底层的控制（OS, kernel, driver）
- 对CPU和内存的高效利用（Server, OS）
- 对运算性能的高要求
- 对系统安全和内存安全的强需求

重点项目&热门领域

- 操作系统
- 虚拟机/容器
- 数据库
- 3D游戏引擎
- 网络服务器
- 浏览器引擎
- 编译器、解释器
- 三维建模/动画/渲染
- 大数据
- 云计算
- 物联网
- 航空航天
- 超级计算机
- 科学运算/机器学习
- 图形图像处理
- 虚拟现实

.....都有系统编程的身影

数据中心

- CPU/GPU
- 内存/硬盘
- 电力
- 网络流量
- 其他设备和人员维护费用

都是白花花的银子，“硬件很便宜”的说法不靠谱
你能买最新硬件，对手也能，无助于提升竞争力

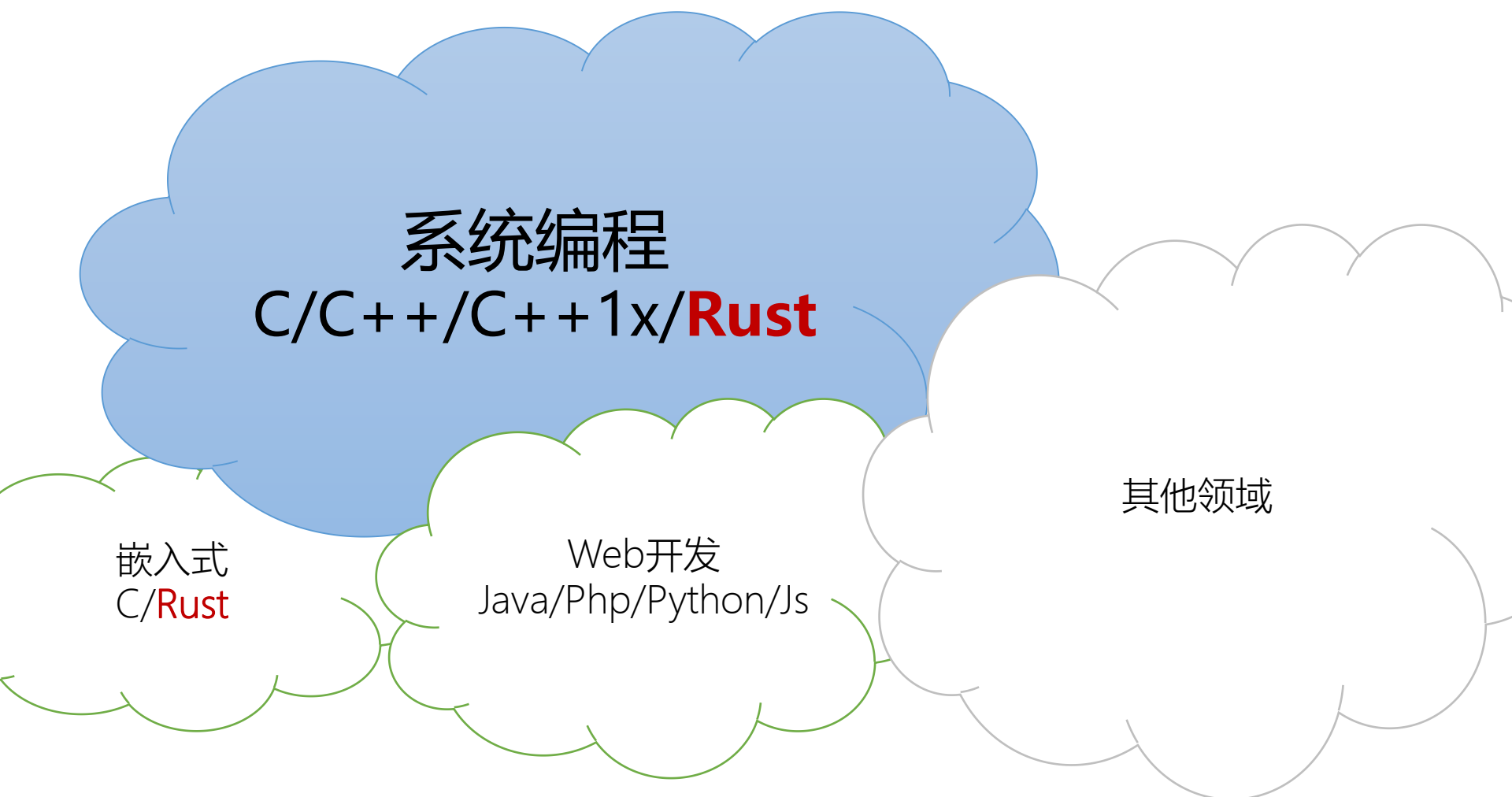
物联网

- 需要大批量部署，必须控制硬件成本
- 受限于成本控制，硬件性能不强
- 受限于电池供电，功耗不能高

这就要求系统和应用软件要高效利用硬件

程序运行在VM上，或后台跑GC
白白浪费了宝贵的CPU和内存资源

Rust在系统编程领域 面临极其强大的竞争对手



(通俗地说)

Rust = 传统 C/C++ 语言
+
内存安全 (没有GC)

传统 VS 现代

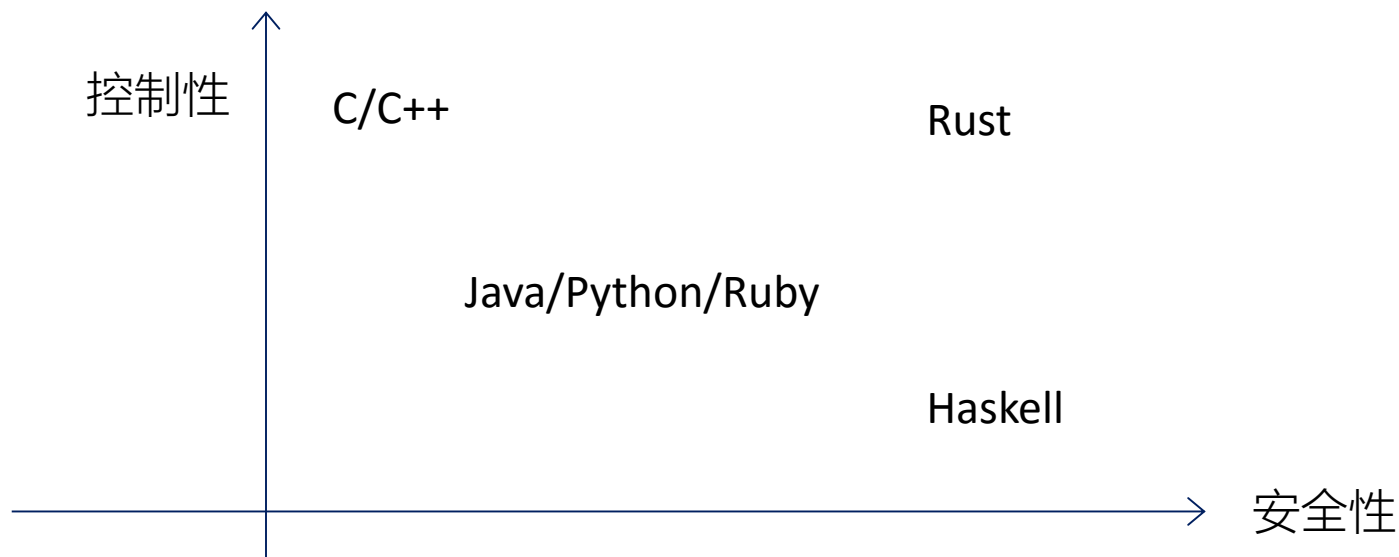
- 传统C语言 (C99)
- 传统C++语言 (C++ 98)
- 现代C++语言 (C++ 1x)

这是3门不同的语言

现代C++被普遍视为全新的语言
现代C++尚在进化革新之中.....

Rust V.S. 传统C/C++

- C/C++有很强的控制性、很弱的安全性
- Rust同时拥有很强的控制性和很强的安全性
- Rust和C/C++一样：运行效率很高，无GC无VM



Rust V.S. 现代C++ (1x)

- 都是对传统C++语言的革新
- 二者应用场景相似，发展目标 and 方向大体相当
- 在互相竞争和互相学习中前进
- Rust暂时领先（内存安全），步伐较快
- C++ 1x 暂时落后，步伐稍慢，多一个历史包袱
- C++ 11 14 17 2x （路还很长）
- 姗姗来迟的11, 小幅改进的14, 眼前的17, 未来的2x...
- 本文后面不再涉及现代C++，因为对其了解有限

系统编程小结

- 系统编程是软件行业的基石
- 很多基础性的、平台性的大中型项目.....
-或隶属于系统编程，或依赖于系统编程
- 系统编程强调底层控制、运行性能和系统安全
- 当前主流的系统编程语言C/C++在内存安全方面有重大欠缺

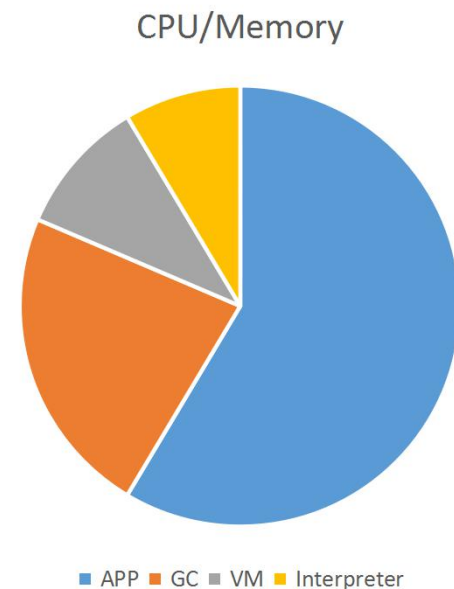
系统编程+零运行时+内存安全

零运行时

Minimal Runtime Overhead

零运行时

- 极小的运行时开销（与C语言相当）
- Zero-cost abstractions
- 无垃圾收集器（GC）
- 无虚拟机（JVM/.Net）
- 无解释器（Python/JS）
- 运行效率很高（与C语言相当）
- 充分高效利用CPU和内存等系统资源



零开销原则/zero-overhead principle

- What you don't use, you don't pay for
- What you do use, you couldn't hand code any better

– Stroustrup, 1994

不为用不到的特性付出代价
仅为用到的特性付出最低代价

这是C++的设计原则，也是Rust的设计原则

No GC

- “没有GC”居然被当作一项特性（20年来GC几乎是标配）
- GC的优势：简化内存管理，基本保证内存安全
- GC的劣势：运行时开销较大，占用CPU和内存较多
- GC不能管理内存以外的其他资源(file/socket/stream)
- 在系统编程领域，GC的运行时开销几乎难以容忍
- GC的终极目的是安全地释放内存
- Rust通过Ownership和RAII也能达到此目的，且性能更好

本次大会涉GC演讲

- Monica Beckwith 《性能工程师指南：玩转OpenJDK HotSpot垃圾收集器》
- 庄振运 《OS造成的长时间非典型JVM GC停顿：深度分析和解决》
- 陶春华 《Golang在BaiduFrontEnd的应用》
- 俞育才 《最优化 Spark 应用的性能—使用低成本的层次化方案加速大数据处理》

.....与其研究怎样优化GC、提高运行性能

.....不如考虑干掉GC，永绝后患

Static and dynamic dispatch

```
trait Run {  
    fn run(&self);  
}
```

泛型/Generics , 静态分派

```
fn static_run<T: Run>(t: &T) {  
    t.run();  
}
```

```
fn dynamic_run(r: &Run) {  
    r.run();  
}
```

Trait Object, 动态分派

Static dispatch

```
define internal void
@_ZN10static_run21h15769893599831004998E
(i32* noalias readonly dereferenceable(4))
unnamed_addr #0 {
    call void @_ZN7i32.Run3run20h405221161c9b7850maaE(i32* noalias readonly dereferenceable(4) %1)
}
```

```
define internal void
@_ZN10static_run21h11253382082138717288E
(float* noalias readonly dereferenceable(4))
unnamed_addr #0 {
    call void @_ZN7f32.Run3run20h26afec01cb8edf9bwaaE(float* noalias readonly dereferenceable(4) %1)
}
```

<http://is.gd/Q2YkAU>

Dynamic dispatch

```
define internal void @_ZN11dynamic_run20h4365854efa082ca1RaaE  
(i8* nonnull, void (i8*)** nonnull) unnamed_addr #0 {
```

entry-block:

```
    %r = alloca { i8*, void (i8*)** }  
    %2 = getelementptr inbounds { i8*, void (i8*)** }, { i8*, void (i8*)** }* %r, i32 0, i32 0  
    store i8* %0, i8** %2  
    %3 = getelementptr inbounds { i8*, void (i8*)** }, { i8*, void (i8*)** }* %r, i32 0, i32 1  
    store void (i8*)** %1, void (i8*)*** %3  
    call void @llvm.dbg.declare(metadata { i8*, void (i8*)** }* %r, metadata !40, metadata !34), !dbg !41  
    %4 = getelementptr inbounds { i8*, void (i8*)** }, { i8*, void (i8*)** }* %r, i32 0, i32 0, !dbg !42  
    %5 = load i8*, i8** %4, !dbg !42  
    %6 = getelementptr inbounds { i8*, void (i8*)** }, { i8*, void (i8*)** }* %r, i32 0, i32 1, !dbg !42  
    %7 = load void (i8*)**, void (i8*)*** %6, !dbg !42  
    %8 = getelementptr inbounds void (i8*)*, void (i8*)** %7, i32 3, !dbg !42  
    %9 = load void (i8*)*, void (i8*)** %8, !dbg !42  
    call void %9(i8* %5), !dbg !42  
    ret void, !dbg !44  
}
```

//trait object, fat pointer, vtable, 虚函数

<http://is.gd/Q2YkAU>

Iterator

```
#[inline(never)]
```

```
fn doit(v: &mut [u8]) {  
    v.iter_mut()  
        .map(|x| *x = 66)    // Lazy iteration / 懒惰遍历  
        .all(|_| true);  
}
```

```
define internal fastcc void @_ZN4doit20h5077d4eddc5f4032eaaE(i8* nonnull) unnamed_addr  
#0 {
```

entry-block:

```
    call void @llvm.memset.p0i8.i64(i8* nonnull %0, i8 66, i64 32, i32 1, i1 false)  
    ret void  
}
```

<http://is.gd/oJ2XcT>

Iterator

```
#[derive(Debug)]
```

```
struct MyData {
```

```
    inner: [u8; 32],
```

```
}
```

```
struct IterMut<'a>(...);
```

```
impl<'a> Iterator for IterMut<'a> { ... }
```

```
#[inline(never)]
```

```
fn doit(v: &mut MyData) {
```

```
    v.iter_mut()
```

```
        .map(|x| *x = 66)
```

```
        .all(|_| true);
```

```
}
```

```
define internal fastcc void @_ZN4doit20h73d357c5d419e850SbaE(%MyData*  
dereferenceable(32)) unnamed_addr #3 {
```

```
entry-block:
```

```
    %1 = getelementptr inbounds %MyData, %MyData* %0, i64 0, i32 0, i64 0
```

```
    call void @llvm.memset.p0i8.i64(i8* %1, i8 66, i64 32, i32 1, i1 false)
```

```
    ret void
```

```
}
```

- FFI
直接调用C库，运行时零损耗
- Thread
直接使用OS本地线程，无额外管理调度损耗
- FileSystem Network等IO操作
薄薄一层封装，通常唯一对应某个系统API调用
- Closure
Inline, Unboxed (No runtime allocation)

零运行时小结

- Rust语言的设计和实现十分重视运行时性能
 - 尽力避免任何非必要的运行时开销
 -仅当与内存安全产生冲突时才有所妥协
 - Rust的性能跟C/C++在同一个数量级上
-
- 零运行时并非指绝对的零，而是尽可能的小
 -仍然跟C/C++的运行时在同一个数量级上

系统编程+零运行时+内存安全

内存安全

Memory Safety

为什么强调内存安全？

- 系统编程对内存安全的需求十分强烈
- 内存不安全的后果非常严重
-“心脏出血”漏洞(Heartbleed)重创全球IT行业
-源于OpenSSL 【越界访问内存】
- OS/GLIBC/JAVA/浏览器等频繁爆出重大安全漏洞
-多与错误使用内存有关
- 传统C/C++语言放弃解决内存安全问题
- 程序员因疏忽或犯错很容易制造内存安全漏洞
- GC能基本保证内存安全，但牺牲了运行时性能



内存安全？

安全地读写内存

- 在限定时间和空间范围内读写内存
- 防止被他人意外修改或释放
- 避免访问空指针和野指针

安全地释放内存

- 在恰当的时机释放
- 确保释放，不遗漏
- 仅释放一次

内存不安全？

- 指针越界访问，意外修改别处内存
- 内存被提前释放，形成野指针，非法读写内存
- 野指针又转化为合法指针，意外修改别处内存
- NULL指针解引用，非法操作
- 并发读写同一内存地址，数据竞争
- 缓冲区溢出、段错误.....

.....等等许多危险操作，在C语言里触目惊心

Move by default

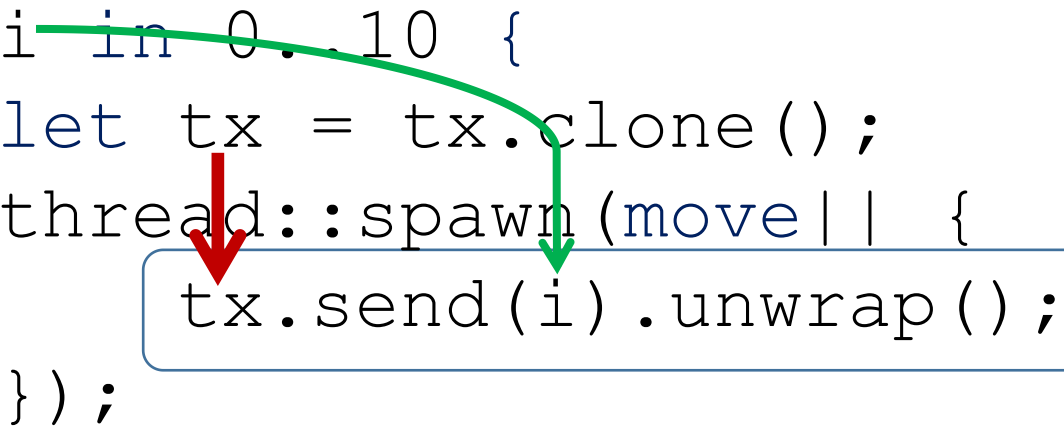
```
struct Value {  
    x: i32,  
}  
  
fn process(v: Value) { /* ... */ }  
  
fn main() {  
    let v = Value{ x: 1 };  
    process(v);  
    println!("{}", v.x);  
}
```

error: use of moved value: `v.x` [E0382]

Move & Copy

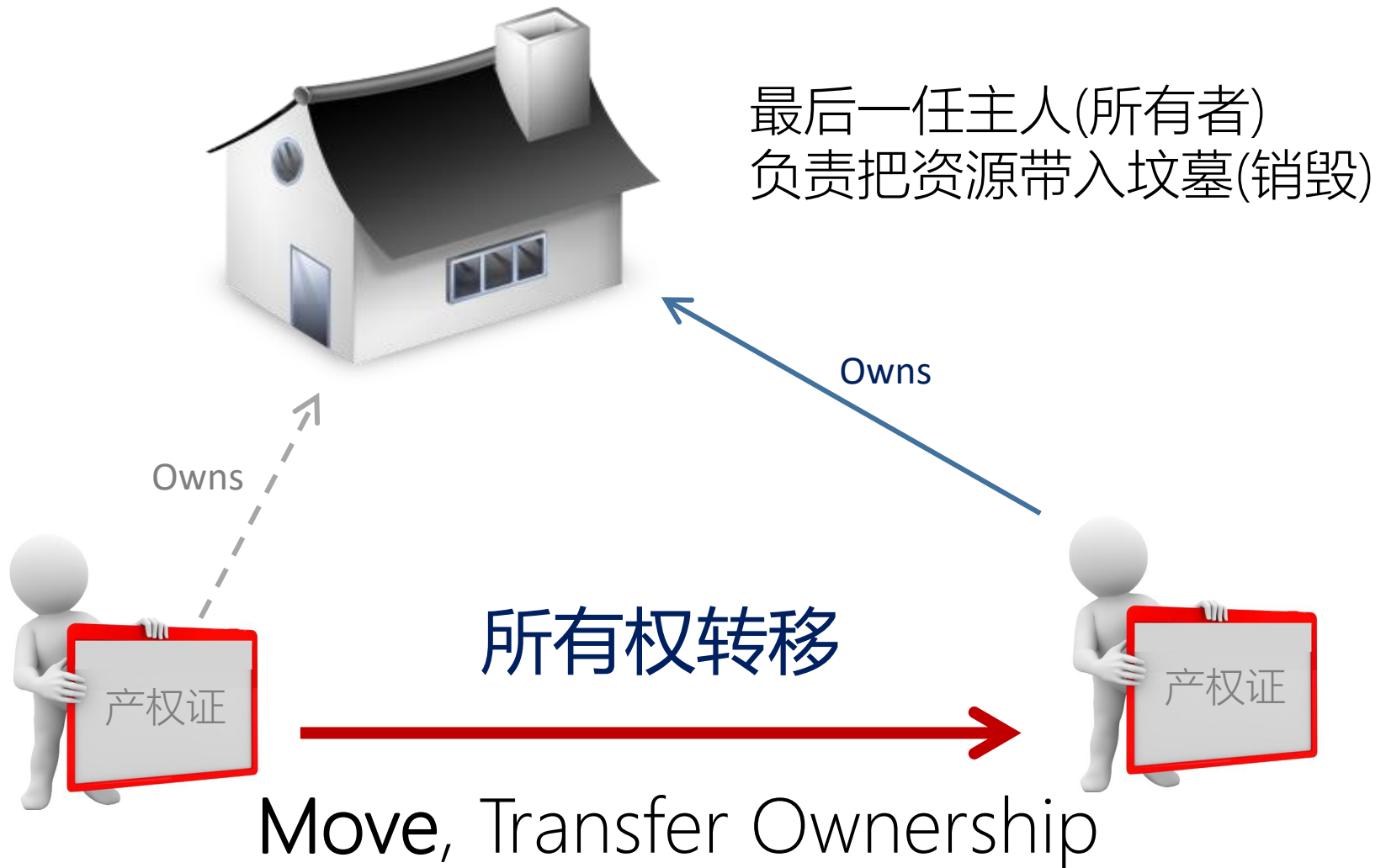
默认是Move，除非类型实现了Copy接口(POD)

```
let (tx, rx) = channel();  
for i in 0..10 {  
    let tx = tx.clone();  
    thread::spawn(move || {  
        tx.send(i).unwrap();  
    });  
}
```



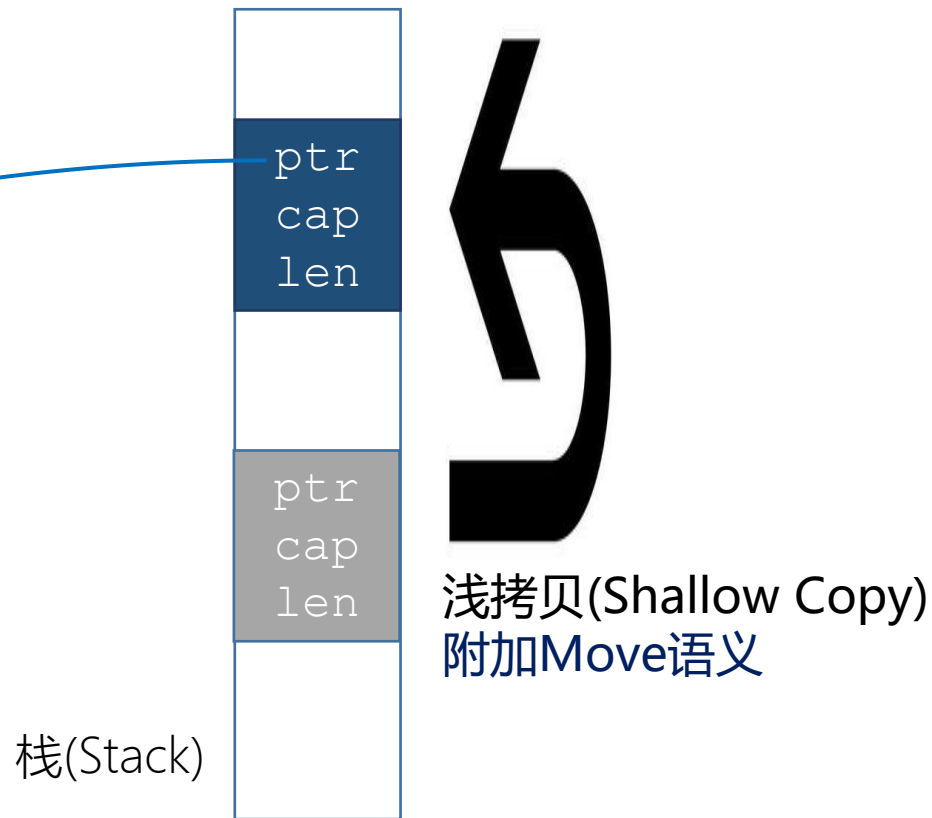
tx (类型是Sender) 被move到新的线程；
i (类型是i32) 被copy到新的线程 (逐字节拷贝)。

Ownership (所有权)



Ownership (所有权)

```
pub struct Vec<T> {  
    ptr: Unique<T>,  
    cap: usize,  
    len: usize,  
}
```



Very big data in the heap ...

堆(Heap)

Drop & RAII

变量(bindings)和资源(resources)的关系

Resources: memories, objects, connections, files, sockets, threads, ...

因为有“所有权转移”的存在：

同一个资源可能由变量A持有或变量B持有；

同一个变量可能持有资源也可能不持有资源；

同一时刻有且只有一个变量唯一持有(Owns)某个资源。

持有资源的变量超出作用域或被另赋新值时，自动调用资源析构函数(Drop)，无论该资源在堆上还是栈上。

Rust的RAII青出于蓝而胜于蓝（蓝=C++）（std::unique_ptr）

Borrowing (租借使用权)



Owners

好借好还再借不难
我死之前必须还我



$\&T$



Shared borrow (多人共享/只读)



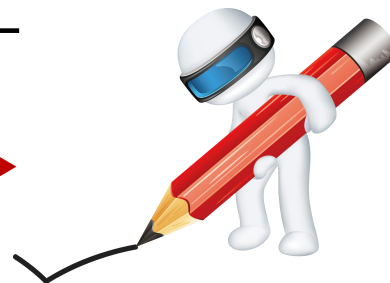
$\&T$

Clients

$\&\text{mut } T$



Mutable borrow (独家修改和使用权)



Lifetime

- Lifetime ('a) 修饰Reference , 是引用的属性
- 限制Reference的生效范围 , 避免无效引用

```
fn get<'a>(&'a self) -> &'a i32 {  
    &self.x  
}
```

```
struct Foo<'a, T: 'a> {  
    x: &'a T,  
}
```


Borrow Checker

Borrowck是编译器内部组件，负责在“编译期”追踪审查引用的有效性，是保证内存安全的重要功臣。运行时零开销。

```
fn main() {  
    let mut v = vec![0, 1, 2, 3, 4, 5, 6];  
    let deleted = v.drain(1..3);  
    println!("{}", v.len());  
}
```

error: cannot borrow `v` as immutable
because it is also borrowed as mutable [E0502]

Borrowck是初学Rust的一大障碍，很多人都有与之抗争的经历，并最终受益

Borrowck的局限性：Rc<Cell<T>>, Rc<RefCell<T>>, Arc<Mutex<T>> ...

Borrow Checker

```
fn main() {  
    let mut v = Vec::new();  
    let t = T::new();  
    v.push(&t); // error: `t` does  
                // not live long enough  
}
```

因为t将先于v被析构（这一点易被忽视）

v的析构函数可能访问到无效的t引用！

此处潜在的内存“不安全”代码没能逃过borrowck的法眼。

```
fn test1(s: String) {}  
fn test2(s: &String) {}  
  
fn main() {  
    let mut s = String::from("QCon");  
    let x = &mut s;  
    // s.push_str("2016"); // error: cannot borrow `s` as mutable  
                           // more than once at a time  
    // s = String::new(); // error: cannot assign to `s`  
                           // because it is borrowed  
    // test1(s); // error: cannot move out of `s`  
                  // because it is borrowed  
    // test2(&s); // error: cannot borrow `s` as immutable  
                  // because it is also borrowed as mutable  
    x.push_str("...");  
}
```

NULL指针安全

```
FILE* fopen(const char* name, const char* mode)
```

C语言无法从类型系统上区分有效指针和无效指针(NULL)，却又习惯把NULL指针用作特殊标记（空缺的参数或返回值）。一旦程序员忽略NULL指针检查，往往会触发很严重的内存错误。人难免因疏忽而犯错，编译期检查对此无能为力。

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

```
match fopen(...) {  
    Some(file) => ... // 必然得到有效的file  
    None => ...      // 必然得不到file  
}
```

并发安全

一个入口：`std::thread::spawn()`

两大门神：`Send`、`Sync`

编译期保证：没有数据竞争（No Data Race）



Safe V.S. Unsafe

// A

Rust类型系统和编译器从机制上保证，unsafe代码块以外的代码（A区），一定是内存安全的。

unsafe {

// B

不能被编译器确认是内存安全的代码，必须且只能写在unsafe代码块内（B区）。

}

尽量把unsafe代码块限制在最小范围内，并且用关键字unsafe突出标识出来，便于集中审查review。

// A

Redox OS kernel: 16.52%, userspace: 0.3%

Safe V.S. Unsafe

有很多人望文生义、想当然的以为unsafe代码块内都是“不安全”的代码。其实不然。

unsafe代码块内，经过专业资深程序员谨慎编写的代码，同样是能保证内存安全的。（通过人脑保证，而非机器）

机器通过执行一系列冰冷的规则拒绝不安全代码。但是机器有机的局限。unsafe代码充分发挥人的主观能动性，允许专家在不受机器束缚的前提下写出安全的代码。

神舟十号与天宫一号有自动对接的同时还有人工对接。

内存安全小结

Rust语言通过：

- 优秀的类型系统设计
 - 严格的编译器静态审查
 - 配合程序员局部核对
 - 加上少量的运行时校验
-保障了内存安全。

（作为对比，传统C/C++语言把包袱完全丢给程序员，还美其名曰“绝对信任程序员”，实质上是放弃。）

因为新概念的引入，增加了Rust语言的复杂性，导致学习曲线比较陡峭，初学难度较大。但学通之后将终生受益。

总 结

系统编程 + 零运行时 + 内存安全

(通俗地说)

Rust = 传统 C/C++ 语言
+
内存安全 (没有GC)

Q&A

谢谢



[@Liigo](#) , [blog](#)

