

# Rust语言高级特性讲解（上）

---

李睿/Ads Infra

2023/07/22

# README

博客: <https://lirui.org> GitHub: @KernelErr

实习经历:

- R&D Intern - WasmEdge
- Software Engineer Intern - Microsoft
- 云&后端实习生 - 太极图形
- 广告系统架构开发实习生 - 字节跳动

开源项目经历:

- Linux文档翻译
- Vector
- Foxear
- Fourth
- ...

01

# Generic Data Types

Rust中的泛型

# 为什么需要泛型

## 从A + B出发

对于 $a + b$ 问题，我们可能下意识写下来这么一个函数：

```
fn add(a: i32, b: i32) -> i32 {  
    return a + b;  
}
```

但这样代码只能在i32下工作，但假如a和b是i64、f64甚至是struct呢？

```
fn add_i32(a: i32, b: i32) -> i32  
fn add_i64(a: i64, b: i64) -> i64  
fn add_f32(a: f32, b: f32) -> f32
```

😔显然不太优雅，对于每个类型都需要专门的函数，而且需要知道一点：Rust中没有函数overload。

# 为什么需要泛型

## 在工程中

```
struct Padded<T> {
    value: T,
    width: usize,
}

impl<T: fmt::Display> fmt::Display for Padded<T> {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "{: <width$}", self.value, width = self.width)
    }
}
```

我们在实现面向其他对象使用的中间组件等时候，很可能不知道调用者会给我们传过来什么类型，这时候泛型就可以帮我们解决问题。😊

# 泛型的基本概念

## 泛型概念

“Generics is the topic of generalizing types and functionalities to broader cases. This is extremely useful for reducing code duplication in many ways, but can call for rather involved syntax.” —— Rust By Example

泛型可以用来将类型和功能概括到更广的范围，代价：会引入更复杂的语法。

对于需要使用泛型的地方，我们可以使用camel case命名的类型参数：

<Foo, Bar, ...> 通常只有一个泛型的时候使用<T>

```
fn function<Foo, Bar>(foo: Foo, bar: Bar)
```

```
struct Foo<T> {  
    bar: T,  
}
```

# 泛型的基本概念

## 泛型的使用

修改版的a + b:

```
fn add<T: std::ops::Add<Output = T>>(a: T, b: T) -> T {  
    return a + b;  
}  
  
println!("1 + 2 = {}", add(1, 2));  
println!("1.1 + 2.2 = {}", add(1.1, 2.2));
```

包含泛型的结构体:

```
struct Foo<T> {  
    bar: T,  
}
```

# 泛型的基本概念

## 泛型的使用

```
struct Graph<T> {  
    pub x: T,  
    pub y: T,  
}  
  
impl<T> Graph<T> {  
    pub fn new(x: T, y: T) -> Graph<T> {  
        Graph { x, y }  
    }  
}  
  
fn main() {  
    let graph: Graph<i32> = Graph::new(x: 1, y: 2);  
}
```

```
struct Graph<T> {  
    pub x: T,  
    pub y: T,  
}  
  
impl Graph<i32> {  
    pub fn new(x: i32, y: i32) -> Graph<i32> {  
        Graph { x, y }  
    }  
}
```

impl后的<T>表示之后的方法都是对于泛型T而言  
Impl Graph<i32>仍然是有效的，但是定义的方法只能用于  
Graph<i32>, 不能用于Graph<f64>

## 泛型的基本概念

```
trait distance {
    fn calculate(&self, other: &Self) -> f64;
}

struct Line<T: distance> {
    pub x: T,
    pub y: T,
}

struct Point {
    pub x: f64,
    pub y: f64,
}

impl distance for Point {
    fn calculate(&self, other: &Self) -> f64 {
        (self.x - other.x).abs() + (self.y - other.y).abs()
    }
}
```

在泛型之后可以加trait约束条件，只有满足了约束条件的泛型才能作为参数。多个trait使用+连接，如：  
<Success: Interest + Hardwork>

# 泛型的基本概念

## 泛型会对性能造成影响么？

Ref: <https://rustc-dev-guide.rust-lang.org/backend/monomorph.html>

许多语言都支持泛型，但是如何去实现和处理泛型，有不同的方法。

对于Java来说，几乎全部变量都是引用，这样提供了灵活性但是牺牲了性能，访问对象的时候都需要去解引用指针。

而Rust采用了不同的方法，即单态化。对于每一个使用泛型的场景，如`Vec<i32>`和`Vec<String>`，Rust编译器会生成两份使用具体类型代码。这样的好处是执行时间快，但是代价是编译时间和二进制大小会有所增加。

## Exercise

- 尝试自己实现一个Buffer<T>, Buffer只有一个成员Vec<T>, 并实现了一个方法sum, 这个方法会返回全部成员的和。

# 02

## String

str、String、&'static str

# str

str类型表示为一个字符串切片，其不可变、不定长，通常使用&str的形式来使用。

```
the size for values of type `str` cannot be known at compilation time
the trait `Sized` is not implemented for `str`
all local variables must have a statically known size
unsized locals are gated as an unstable feature rustc(Click for full compiler diagnostic)
lib.rs(21, 16): consider borrowing here: `&`
```

Example:

```
let s1: &str = "呜呜呜";
let s2: &String = &String::from("wow");
let s3: &str = &String::from("wow");
let s4: &str = &s2[0..2];
```

# &'static str

& 'static str是一种特殊的&str类型，&' static表明&str拥有静态生命周期，在整个程序运行期间都有效。



The screenshot shows a terminal window with the following content:

```
src > main.rs > THURSDAY
1 static THURSDAY: &'static str = "V50!!!";
2
3 fn main() {
4     println!("{}", THURSDAY);
5 }
```

Below the code editor, there is a navigation bar with tabs: 问题, 输出, 调试控制台, 终端, 端口. The '终端' tab is currently selected.

```
$ cargo build --release
Compiling hello v0.1.0 (/data00/home/lirui.dev/hello)
    Finished release [optimized] target(s) in 0.21s
$ strings ./target/release/hello | grep V50
V50!!!
$
```

# str

str的一些常用方法：

```
let s: &str = "我能吞下玻璃而不伤身体";
assert_eq!(s.len(), 33);
assert!(s.contains("玻璃"));
assert_eq!(s.find("能"), Some(3));
assert_eq!(s.split("而").next(), Some("我能吞下玻璃"));
let chars: Vec<char> = s.chars().collect();
assert_eq!(chars.len(), 11);
assert!(s.starts_with("我"));
assert!(s.ends_with("身体"));
```

# String

str表示一个不可变的切片，假如我们拥有可以动态创建和修改的字符串的话，可以使用String类型。

String表示一个UTF-8编码的字符串，其本身只包含一个指针指向堆中保存的数据和length、capacity。当length增长到超过其capacity时重新分配更大的内存空间，并做一次拷贝。

创建String:

```
let s1 = String::new();
let s2 = String::from("滴滴叭叭鸣");
let name = "Cloud";
let s3 = format!("{}WeGo", name);
```

修改String:

```
let mut s = String::from("a");
s.push_str("bc");
s.push('d');
s.pop();
s.insert(0, ' ');
assert_eq!(s, " abc");
```

## Exercise

- 尝试实现一个函数`fn compareString(x: &str, y: &str) -> bool`, 返回true表示x比y在字典序上更大, 否则返回false。不要使用现成的比较函数和库。

# 03

## Closure & Iterator

Rust中的闭包和迭代器

## 闭包是什么

在以前一场面试中，我被问到了这么一个问题：你知道闭包么？为什么需要闭包？

```
fn foo(i: i32) -> i32 { i + 1 }
let bar = |i| i + 1;

assert_eq!(foo(1), bar(1));
```

Rust中的闭包是可以捕获环境的匿名函数，可以用来存入变量或者给其他函数传递参数。

闭包需要注意两点：

- Rust中的闭包捕获变量和返回值是可以被推断或者明确指定的，在被推断后，我们不能再传入其他类型，如bar(1.1)。
- 需要指定捕获变量的名称。

# 闭包是什么

```
let a: Vec<i32> = vec![1, 2, 3];
let func: impl Fn() -> usize = || {
    return a.len();
};
assert_eq!(func(), 3);
```

闭包可以捕获环境中的变量（不可变借用）。

```
let mut a: Vec<i32> = vec![1, 2, 3];
let mut func: impl FnMut() = || {
    a.push(4);
};
func();
assert_eq!(a.len(), 4);
```

闭包可以可变借用环境中的变量。

```
let mut a = vec![1, 2, 3];
----- move occurs because `a` has type `Vec<i32>`, which does not implement the `Copy` trait
let mut func = move || {
    ----- value moved into closure here
    a.push(4);
    - variable moved due to use in closure

assert_eq!(a.len(), 4);
    ^^^^^^^^ value borrowed here after move
```

也可以将值转移到闭包中。

# 闭包是什么

```
let idx = vec![5, 4, 3, 2, 1];

let mut data = vec![Foo { bar: 0 }, Foo { bar: 1 }, Foo { bar: 2 }, Foo { bar: 3 },
    Foo { bar: 4 }];

data.sort_by(|a, b| {
    idx[a.bar].cmp(&idx[b.bar])
});
```

对于许多数据结构提供的方法，其中F往往指的是一个闭包，我们可以传递闭包来控制排序、去重、查找等。

▷ eonnect(...)	fn(&self, Separator) → <[T] as Join<Separat...
▷ contains(...)	fn(&self, &T) → bool
▷ copy_from_slice(...) (alias memcpy)	fn(&mut self, &[T])
▷ copy_within(...)	fn(&mut self, R, usize)
▷ dedup_by(...)	fn(&mut self, F)
▷ dedup_by_key(...)	fn(&mut self, F)
▷ drain(...)	fn(&mut self, R) → Drain<'_, T, A>
▷ ends_with(...)	fn(&self, &[T]) → bool
▷ extend(...)(as Extend)	fn(&mut self, T)
▷ fill(...)(alias memset)	fn(&mut self, T)
▷ fill_with(...)	fn(&mut self, F)
▷ first()	const fn(&self) → Option<&T>

# 闭包是什么

```
struct Cacher<T>
where
    T: Fn(u32) -> u32,
{
    calculation: T,
    value: HashMap<u32, u32>,
}

impl<T> Cacher<T>
where
    T: Fn(u32) -> u32,
{
    fn new(calculation: T) -> Cacher<T> {
        Cacher {
            calculation,
            value: HashMap::new(),
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        if let Some(ret) = self.value.get(&arg) {
            *ret
        } else {
            let ret: u32 = (self.calculation)(arg);
            self.value.insert(k: arg, v: ret);
            ret
        }
    }
}
```

闭包可以结合泛型使用，示例代码实现了一个取值时再进行计算的Cache。

Rust中有三种闭包类型：

Fn: 不可变借用当前环境的变量，可以被调用多次。

FnMut: 可变借用当前环境的变量，可以被调用多次，但不能并行调用。

FnOnce: 转移了变量的所有权，因此只能调用一次。

```
fn main() {
    let mut cacher = Cacher::new(|x| {
        println!("Calculating...");
        thread::sleep(Duration::from_secs(2));
        x * 2
    });
    println!("{} {}", cacher.value(2), cacher.value(2));
    println!("{} {}", cacher.value(2), cacher.value(3));
}
```

# 迭代器是什么

```
#[test]
▶ Run Test | Debug
fn test_iterator(){
    let a: Vec<i32> = vec![1, 2, 3];
    let mut iter: IntoIter<i32> = a.into_iter();
    assert_eq!(iter.next(), Some(1));
    assert_eq!(iter.next(), Some(2));
    assert_eq!(iter.next(), Some(3));
    assert_eq!(iter.next(), None);
}
```

```
Implementation
struct Foo{};

impl Iterator for Foo {
    type Item = Foo;

    fn next(&mut self) -> Option<Self::Item> {
        return Some(Foo{})
    }
}

#[cfg(test)]
▶ Run Tests | Debug
mod test {
    use super::*;

    #[test]
    ▶ Run Test | Debug
    fn test_custom_iterator(){
        let mut bar: Foo = Foo{};
        assert!(bar.next().is_some());
    }
}
```

迭代器可以让我们对一个序列的值进行处理，同时 Rust的迭代器是惰性执行的，在执行前不会有效果。

本质上，迭代器是实现了Iterator这个trait的对象。

# 迭代器是什么

有许多可以消费迭代器的方法，如：

- Sum
- Filter
- Collect
- Flatten
- ...

完整列表：<https://doc.rust-lang.org/stable/std/iter/trait.Iterator.html>

而一个迭代器可以通过闭包产生另外一个迭代器：

```
#![test]
▶ Run Test | Debug
fn test_iterator(){
    let a: Vec<i32> = vec![1, 2, 3];
    let iter: impl Iterator<Item = i32> = a.iter().map(|x: &i32| *x + 1);
    let res: Vec<i32> = iter.collect();
    assert_eq!(res, vec![2, 3, 4]);
}
```

# 闭包是什么

```
0 implementations
1 struct Item {
2     quantity: usize,
3 }
4
5 fn item_filter(items: Vec<Item>) -> Vec<Item> {
6     items.into_iter().filter(|x: &Item| x.quantity >= 5).collect()
7 }
8
9 #[cfg(test)]
10 // Run Tests | Debug
11 mod test {
12     use super::*;

13     #[test]
14     // Run Test | Debug
15     fn test_filter() {
16         let mut items: Vec<Item> = vec![Item{quantity: 1}, Item{quantity: 5}, Item{quantity: 6}];
17         items = item_filter(items);
18         assert_eq!(items.len(), 2)
19     }
20 }
```

闭包配合迭代器可以简化过滤元素等操作。

## Exercise

- 尝试将一个Vec<char>内容为a, b, c, d, e, 通过闭包+迭代器生成一个新的Vec<char>, 内容是b,c,d,e,f。

# 04

## Unsafe

Unsafe in safe

# 什么是安全，什么是不安全

下面情况是“安全的”：

- 死锁
- 有一个数据竞争
- 内存泄漏
- 未能调用解构器
- 整数溢出
- 中止程序
- 删 除生产数据库

Rust语言本身关心的“安全”是防止引起数据竞争、产生无效的值、解引用悬空或不对齐指针等。如果开发者仅仅编写Safe Rust是永远不必担心内存安全的，然而有些时候我们不得不涉足Unsafe Rust:

- 与操作系统、C链接库交互
- 追求更加高效的数据结构和运算
- ...

# Unsafe case

```
unsafe {
    let num_pages = HEAP_SIZE / PAGE_SIZE;
    if pages > num_pages {
        panic!("Not enough memory, {}/{}", pages, num_pages);
    }
    let p = HEAP_START as *mut Page;
    for i in 0..num_pages - pages {
        let mut found = false;
        if (*p.add(i)).is_free() {
            found = true;
            for j in i..i + pages {
                if (*p.add(j)).is_taken() {
                    found = false;
                    break;
                }
            }
        }
        if found {
            for k in i..=i + pages - 1 {
                (*p.add(k)).set_flag(PageBits::Taken);
            }
            (*p.add(i + pages - 1)).set_flag(PageBits::Last);
            return (i * PAGE_SIZE + ALLOC_START) as *mut u8;
        }
    }
}
```

```
pub fn blue_led_on() {
    let pd2_controller = 0x02000098 as *mut u32;
    let mut pd2_controller_val: u32;
    unsafe {
        pd2_controller_val = ptr::read(pd2_controller);
    }
    pd2_controller_val &= 0xf0ffff;
    pd2_controller_val |= 0x01000000;
    unsafe {
        ptr::write_volatile(pd2_controller, pd2_controller_val);
    }
}

pub fn blue_led_off() {
    let pd2_controller = 0x02000098 as *mut u32;
    let mut pd2_controller_val: u32;
    unsafe {
        pd2_controller_val = ptr::read(pd2_controller);
    }
    pd2_controller_val &= 0xf0ffff;
    pd2_controller_val |= 0x0f000000;
    unsafe {
        ptr::write_volatile(pd2_controller, pd2_controller_val);
    }
}
```

# Unsafe case

```
// a0 contains a unique per-hart ID
// a1 contains a pointer to the device tree
// We should save the value ASAP
let a0: usize;
let a1: usize;
unsafe {
    asm!(
        "",
        out("x10") a0,
        out("x11") a1,
    );
}
```

借助Unsafe Rust我们可以像C一样去直接和内存中的值交互，也可以内联汇编。Unsafe并不意味着不安全，在Unsafe Rust中，安全由开发者自己负责。

Ref: [Rust 秘典（死灵书）](#)

## Exercise

- 阅读：[Rust 秘典（死灵书）](#)，并提前预习生命周期的概念。

# 05 Test

测试

# Program without/with test

- 上线没有信心
- 随时可能出现panic
- 可能出现非预期的行为
- 开发过程比较痛苦，对着代码逻辑反复分析，不知道正确性

```
while(true)  
{  
    ;  
}  
反复分析
```

- 对于代码逻辑输入输出清晰
- 可以有良好的睡眠
- 配合其他QA手段，对上线有信心

# Unit test

```
use crate::define_operation;
use anyhow::{anyhow, Result};
use image::DynamicImage;

define_operation!(
    #[doc = "Tile an image into a new image."]
    unsharpen(image),
    sigma: f32,
    threshold: i32,
    { Ok(Some(image.unsharpen(sigma, threshold))) }
);

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_unsharpen_image() {
        let img = image::open("tests/images/ryan-yao-VURwPtZqyF4-unsplash.jpg").unwrap();
        let img_clone = img.clone();
        let unsharpened_img = execute(&img, OperationArg(1.5, 1)).unwrap().unwrap();

        assert_eq!(img, img_clone);
        assert_ne!(img, unsharpened_img);
    }
}
```

Rust中的测试是一些特殊函数调用待测函数，并判断结果是否符合预期。

执行结果后，我们可以通过

assert!	assert!(optional.is_some())
assert_eq!	assert_eq!(1, 1)
assert_ne!	assert_ne!(1, 2)

去断言结果，如果不符合预期则会panic。

执行测试可以使用

\$ cargo test <test name>

<https://github.com/KernelErr/imgtool/blob/main/src/process/unsharpen.rs>

# Unit test

Rust中的测试是一些特殊函数调用待测函数，并判断结果是否符合预期。

执行结果后，我们可以通过

assert! assert!(optional.is\_some())

assert\_eq! assert\_eq!(1, 1)

assert\_ne! assert\_ne!(1, 2)

去断言结果，如果不符预期则会panic。

执行测试可以使用

```
$ cargo test <test name>
```

# Unit test

```
#[test]
#[should_panic]
► Run Test | Debug
fn test_panic() {}
```

个人认为Rust中应该“panic as it should be”，对于恶意输入等情况，我们可以理性看待panic，应该panic时就该panic。Rust注重安全，对于应该panic的情况，也提供了测试方法，我们可以给测试函数加上should\_panic属性。

```
---- test::test_panic stdout ----
note: test did not panic as expected

failures:
    test::test_panic

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 2 filtered out; finished in 0.00s

error: test failed, to rerun pass `--pass`
```

```
5: core::ops::function::FnOnce::call_once
    at /rustc/eb26296b556cef10fb713a38f3d16b9886080f26/library/core/src/ops/function.rs:250:5
6: core::ops::function::FnOnce::call_once
    at /rustc/eb26296b556cef10fb713a38f3d16b9886080f26/library/core/src/ops/function.rs:250:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
test test::test_panic - should panic ... ok
```

# Document test

```
▶ Run Doctest
1 /// Add two i32 numbers together
2 ///
3 /// ````rust
4 /// let res = course::add(1, 2);
5 /// assert_eq!(res, 3);
6 /// ``
7 pub fn add(x: i32, y: i32) -> i32 {
8     x + y
9 }
10
```

对于Rust中的library代码，可以在文档注释中加入测试代码，cargo test时会测试其中的代码。

Rust的文档注释遵循CommonMark Markdown规范。

## Function `course::add`

[source](#) · [-]

```
pub fn add(x: i32, y: i32) -> i32
```

[-] Add two i32 numbers together

```
let res = course::add(1, 2);
assert_eq!(res, 3);
```

# C

```
name: Rust

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

env:
  CARGO_TERM_COLOR: always

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Upgrade Rust
      run: rustup update
    - name: Build
      run: cargo build --verbose
    - name: Run tests
      run: cargo test --verbose
```

除了在本地进行测试外，我们可以使用GitHub Action在每次Pull Request创建、main分支有新commit时自动触发流水线执行，进行编译和测试。在实际工程中，可以保证合入主分支的代码都已经通过测试，对代码质量进行保证。

Doc: <https://docs.github.com/en/actions>

Ref: <https://github.com/KernelErr/fourth/tree/main/.github/workflows>

All workflows			
Showing runs from all workflows			
Event	Status	Branch	Actor
Rust #5: Commit 1ffd8e3 pushed by KernelErr	main	last year	3m 28s
Rust #4: Commit d2491a5 pushed by KernelErr	main	last year	2m 9s
Rust #3: Commit 1cb596d pushed by KernelErr	main	last year	2m 32s
Rust #2: Commit 0b2e396 pushed by KernelErr	main	2 years ago	2m 36s
Rust #1: Commit 584a9c5 pushed by KernelErr	main	2 years ago	2m 37s

## Exercise

- 在GitHub上创建一个Repo，写一个简单的Rust小项目，并设置好GitHub action，在每次main分支有新commit时自动编译、测试。
- 进阶：尝试实现打标签后自动编译Linux、Windows、Mac的二进制并创建GitHub release上传二进制。

# 06

# Profiling

性能分析

# Measure performance of your code

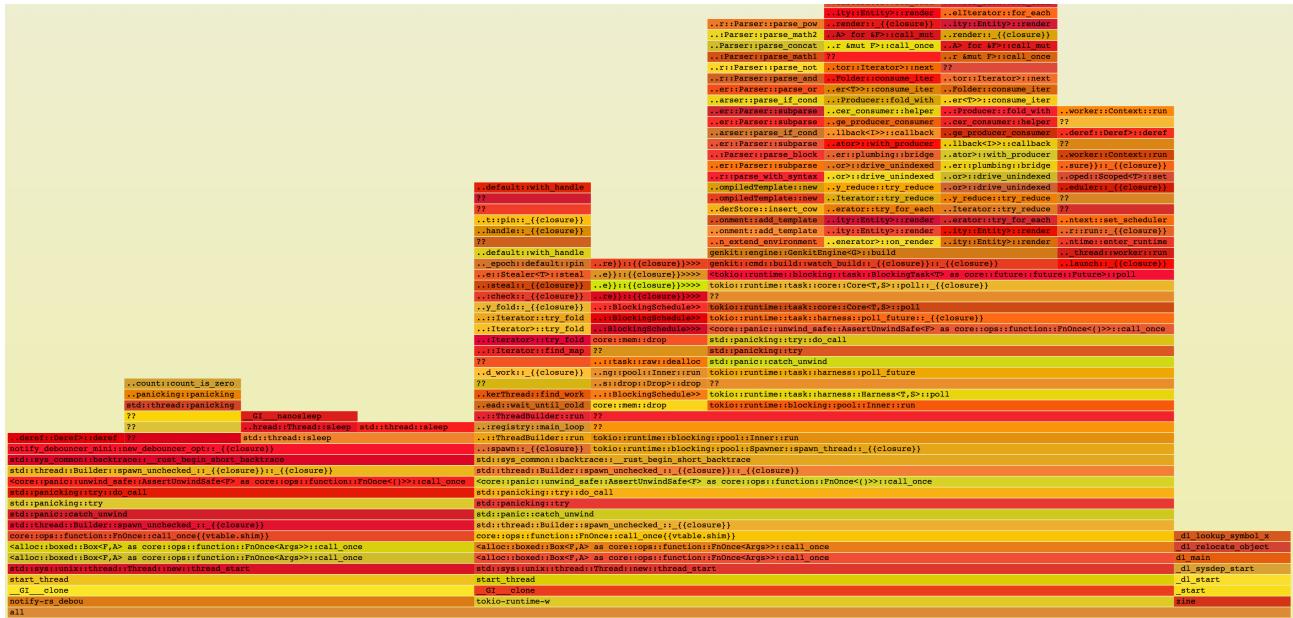
有些时候我们写了一个后端或是一个客户端程序，发现随着输入大小的增加，程序性能开始劣化，有哪些手段去测量代码性能？

- 打点，将调用用时交给类似Prometheus等服务收集起来，之后使用Grafana可视化
- 使用\*nix系统提供的time命令获取执行用时和CPU耗时
- 使用\*nix系统提供的sar命令查看进程相关数据，如IO、CPU等
- 使用profiling工具生成火焰图，分析函数执行时间占比
- 分析复杂度  胡乱分析
- 使用ebpf技术进行trace
- 进行benchmark
- 查看function call graph
- ...



# Flame graph

火焰图可用于生成程序用时分布，其原理是在程序线程执行时多次中断线程，对其进行堆栈跟踪（堆栈采样），从而获知函数用时占整个采样用时的部分。



火焰图函数本身颜色仅作区分，不代表耗时程度。  
火焰图从左到右也不代表时间推移。  
火焰图每一格的长度取决于采样到的次数。

# Flame graph with [cargo-]flamegraph

GitHub: <https://github.com/flamegraph-rs/flamegraph>

现在已经有许多开箱即用的火焰图采样生成工具，这里介绍flamegraph。和其他Rust生态的二进制一样，可以使用cargo安装： cargo install flamegraph

在cargo.toml中添加相关配置保留debug信息：

```
[profile.release]
debug = true

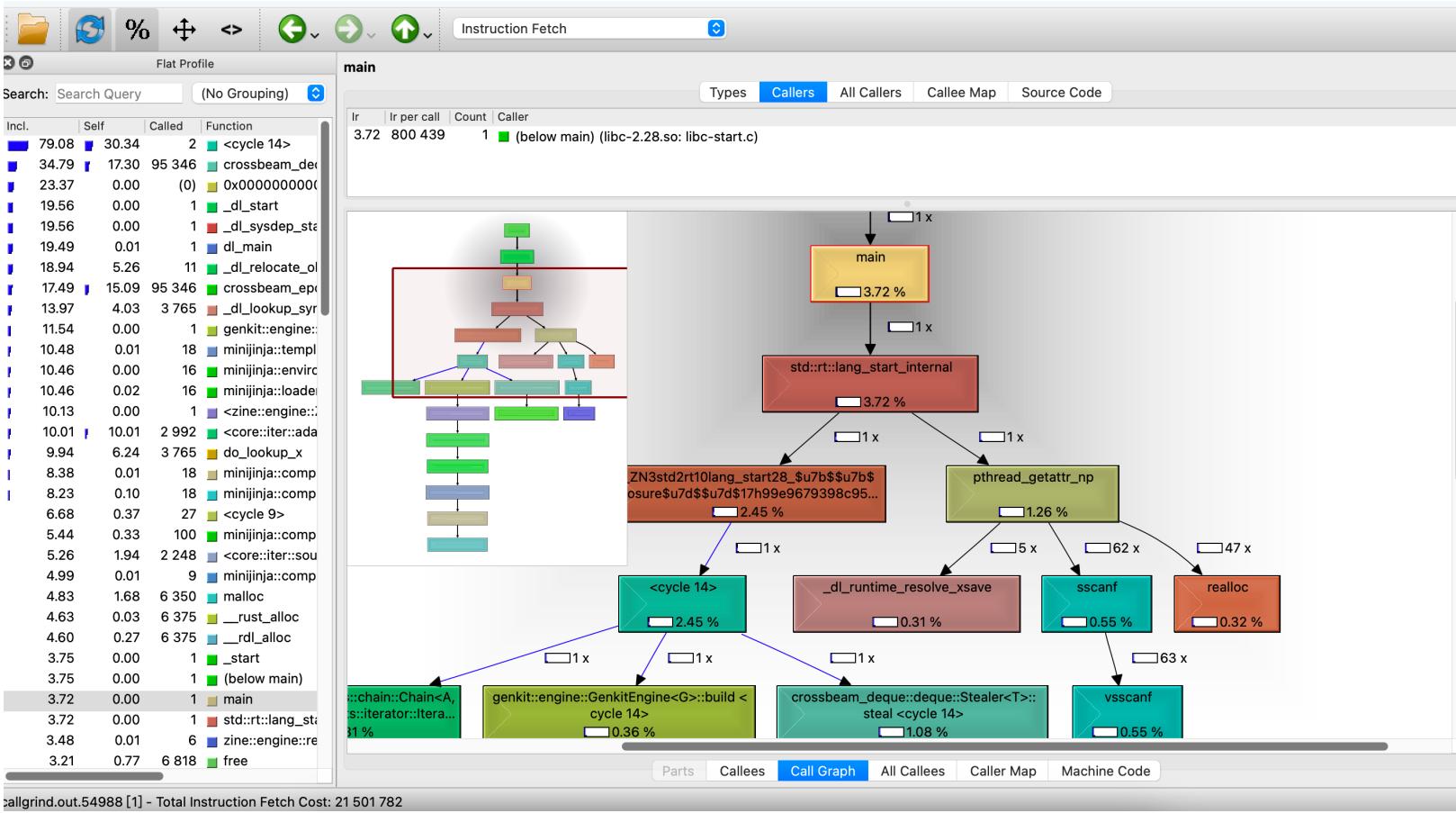
[target.x86_64-unknown-linux-gnu]
linker = "/usr/bin/clang"
rustflags = ["-Clink-arg=-fuse-ld=lld", "-Clink-arg=-Wl,--no-rosegment"]
```

之后只需要执行cargo flamegraph即可。

# Valgrind – callgrind

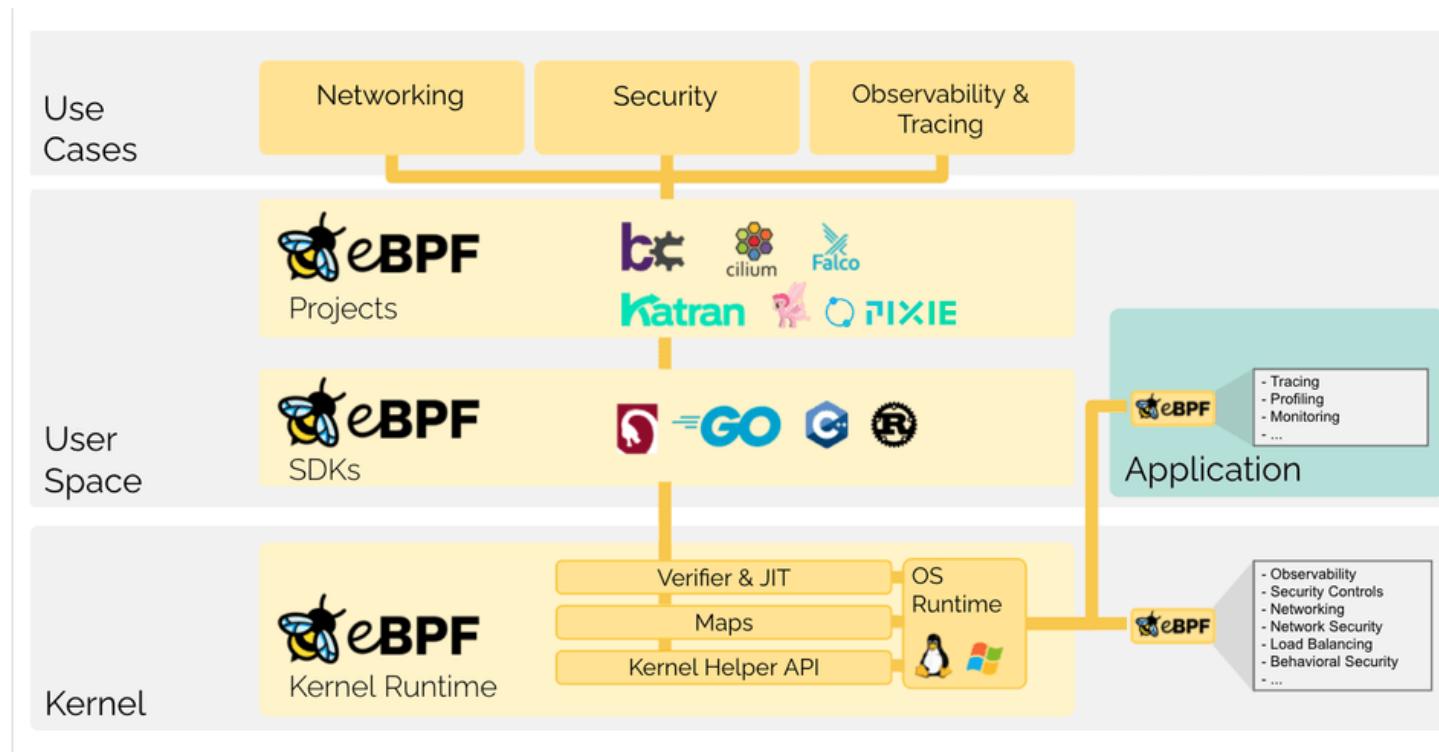
Valgrind是一款用于性能分析、内存调试、内存泄露检测的软件。我们可以使用其中的callgrind工具生成调用图，调用图可以直观看到每个函数耗时的百分比。

valgrind --tool=callgrind [callgrind options] your-program [program options]



# eBPF

eBPF和WebAssembly是新时代的两大虚拟化技术。eBPF是运行在Linux内核中的虚拟机，开发者可以使用C/C++/Rust等语言编写代码，通过工具链编译到eBPF二进制并载入到Linux内核中。Linux在载入eBPF模块时会对模块进行验证（保证没有死循环、非法内存访问等情况），如果没有问题就可以载入内核交给JIT编译器。简单来说，可以认为eBPF模块“附加(attach)”到了内核中代码执行路径，从而可以：



- 对网络包进行处理
- 创建过滤器允许/阻止系统调用  
([https://docs.kernel.org/translations/zh\\_CN/userspace-api/seccomp\\_filter.html](https://docs.kernel.org/translations/zh_CN/userspace-api/seccomp_filter.html))
- 对系统调用进行监控
- ...

# bpftrace

GitHub: <https://github.com/iovisor/bpftrace>

```
#!/usr/bin/env bpftrace
/*
 * opensnoop  Trace open() syscalls.
 *      For Linux, uses bpftrace and eBPF.
 *
 * Also a basic example of bpftrace.
 *
 * USAGE: opensnoop.bt
 *
 * This is a bpftrace version of the bcc tool of the same name.
 *
 * Copyright 2018 Netflix, Inc.
 * Licensed under the Apache License, Version 2.0 (the "License")
 *
 * 08-Sep-2018 Brendan Gregg Created this.
 */

BEGIN
{
    printf("Tracing open syscalls... Hit Ctrl-C to end.\n");
    printf("%-6s %-16s %4d %3d %s\n", "PID", "COMM", "FD", "ERR", "PATH");
}

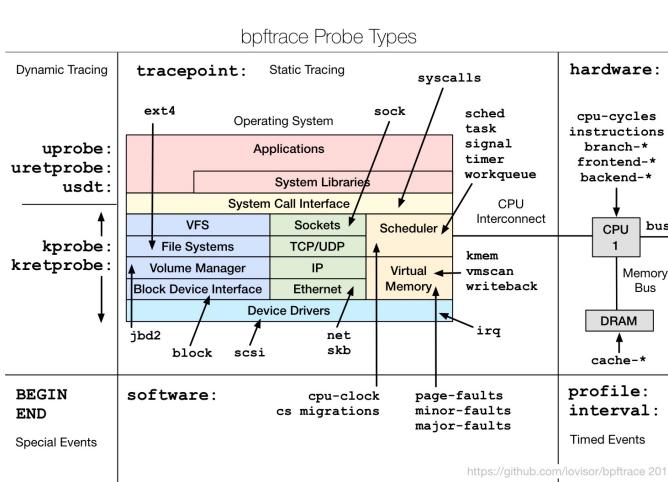
tracepoint:syscalls:sys_enter_open,
tracepoint:syscalls:sys_enter_openat
{
    @filename[tid] = args.filename;
}

tracepoint:syscalls:sys_exit_open,
tracepoint:syscalls:sys_exit_openat
/@filename[tid]/
{
    $ret = args.ret;
    $fd = $ret >= 0 ? $ret : -1;
    $errno = $ret >= 0 ? 0 : - $ret;

    printf("%-6d %-16s %4d %3d %s\n", pid, comm, $fd, $errno,
           str(@filename[tid]));
    delete(@filename[tid]);
}

END
{
    clear(@filename);
}
```

左图是bpftrace监控open系统调用的例子。



```
sudo bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @ = count(); } interval:s:1 { print(@); clear(@); }'
```

```
Attaching 2 probes...
```

```
@: 25842
```

```
@: 5807
```

```
@: 10657
```

```
@: 6157
```

```
@: 5725
```

```
@: 6795
```

```
@: 7123
```

## Exercise

- 尝试对之前写的find工具做profiling。

# 07

## OSS contribution

开源贡献

# From where to start

不要为了开源而开源。

从哪儿开始？

- 给知名的项目做贡献，如Linux还有大量文档没有翻译，不要做cleanup。
- 在使用开源项目的时候发现bug，需要新feature。
- 自己想写一些软件、库贡献给开源社区。
- 找一份开源项目团队的实习工作，或者参加开源之夏等项目。

THANKS

