

# Rust语言入门、关键技术与实战经验

原创：唐刘 高可用架构 2016-03-25

编者按：高可用架构分享及传播在架构领域具有典型意义的文章，本文由唐刘在高可用架构群分享。转载请注明来自高可用架构公众号「ArchNotes」。



唐刘，PingCAP 首席架构师，现致力于下一代分布式数据库 TiDB、分布式存储 TiKV 的开发。开源爱好者，Go、Rust 等语言爱好者和实 践者。

大家好，我是 PingCAP 的唐刘，今天很荣幸跟大家来分享一下 Rust 相关知识以及我们团队使用 Rust 的实战经验。

## 为什么选择Rust

首先来说说最近 Go 社区讨论比较多的事情，Dropbox 将底层的类 S3 服务改用 Rust 来重写了，一下子让 Rust 增加了很多知名度，谁叫 Dropbox 这种公司通常都是技术架构上面的风向标。大家普遍关注的一个问题：

**为什么 Dropbox 不用 Go，反而用一门学习曲线比较陡峭的 Rust 来进行开发？**

其实这个问题也同样适用于我们，一个自认为 Go 经验非常丰富的团队，为什么不用 Go 反而要选择 Rust？

介绍一下我们在做的事情。我们团队从事的是下一代分布式数据库的开发，也就是俗称的 NewSQL，整个理论基础基于 Google 的 F1 以及 Spanner，所以自然我们的 NewSQL 也是分成了两块，一个是无状态的 SQL 层，也就是现阶段已经开源出来的 TiDB (<http://dwz.cn/2XZkSm>)，另一个就是分布式 KV 层，我们叫做 TiKV，预计会在 4 月份开源。



PingCAP

TiDB 是使用 Go 编写的，熟悉 Go 的同学都应该知道，用 Go 来写分布式应用那是非常的快捷方便的，而且我们团队的成员都有非常深厚的 Go 编程经验，但是在决定做 TiKV 的时候，我们没有使用 Go，或者使用 C++，Java 这些更流行的静态语言，反而是选择了一门我们自身完全不熟悉的语言 Rust，Why？

先来说说使用 Go 会遇到的问题：

- GC，虽然 1.6 之后 Go 的 GC 已经改善的很好了，而且我们觉得以后会越来越好，但是对于一个对性能要求非常高的分布式应用，我们倾向选择一门没有 GC 的语言，这样在内存上面更加可控。
- Cgo，TiKV 使用 RocksDB 作为其底层存储 engine，如果用 Go，我们需要使用 Cgo 来进行 RocksDB 的调用，而 Cgo 对性能还是有比较大的损耗的。

再来说说不选择 C++ 或者 Java，C++ 主要是大规模开发对整个团队要求非常高，而我们自己也觉得用 C++ 不可能 hold 住在短时间内做出产品，所以自然放弃。而 Java，我们团队没几个人精通 Java，也直接放弃了。

所以，我们最终将目光落到了 Rust 上面，主要在于 Rust 有几个很 cool 的 feature，就是 type safety，memory safety 以及 thread safety，这个后续详细说明。

## Rust 的基础入门

---

在 Rust 的官网上面，我们可以看到 Rust 的介绍，它是一门系统编程语言，性能好，同时在编译阶段就能帮你检测出内存，多线程数据访问等问题，保证程序安全健壮。

也就是说，使用 Rust 写程序，如果能通过编译，你就不用担心类似 C++ 里面很多 memory leak，segment fault，data race 的问题了，但这一切都是有代价的。Rust 上手非常不容易，难度可以跟 C++ 媲美，如果是 Go，没准学习一个星期都能开始给项目贡献代码，但换成 Rust，可能一个月都还在跟编译器作斗争，研究为啥自己的代码编译不过。

因为我不清楚大家有多少人接触过 Rust，所以这里列出来一些例子，让大家对 Rust 的语法有一个基本了解。

首先就是最通用的 Hello world：

```
fn main() {  
    println!("Hello world!");  
}
```

(点击图片可全屏缩放图片)

fn 是 Rust 的关键字，用来定义函数，函数名字是 main, println! 是一个 macro，在 rust style 里面，macro 都是在末尾使用“!”来表示的。

我们接下来定义几个变量，下面就开始显示出 rust 跟其他语言的不一样的地方了：

```
fn main() {  
    let x:u32;  
    println!("Hello {}", x);  
}
```

(点击图片可全屏缩放图片)

上面我们声明了一个 u32 类型的变量，但是没有初始化，然后打印它的值，在一些语言里面，譬如 Go，会打印出 0，但是在 Rust 里面，这没法编译通过，编译器会提示：

```
<anon>:5:27: 5:28 error: use of possibly uninitialized variable: `x`  
[E0381]  
<anon>:5    println!("Hello {}", x);
```

(点击图片可全屏缩放图片)

上面的错误告诉我们使用了一个没有初始化的变量，我们先来初始化一下，变成这样：

```
fn main() {  
    let x:u32 = 0;  
    x = 10;  
    println!("Hello {}", x);  
}
```

(点击图片可全屏缩放图片)

上面我们首先定义了一个初始值为 0 的变量，然后改成 10 打印，编译，发现如下错误：

```
<anon>:4:5: 4:11 error: re-assignment of immutable variable `x` [E0384]  
<anon>:4    x = 10;  
          ^~~~~~
```

(点击图片可全屏缩放图片)

这次编译器告诉我们对一个 immutable 的变量进行了更改。在 Rust 里面，变量是分为 immutable 和 mutable 的，我们必须显示地定义这个变量到底能不能改动。我们使用

mut 关键字来告诉 Rust 这个变量是 mutable 的，如下：

```
fn main() {  
    let mut x = 0;  
  
    x = 10;  
    println!("Hello {}", x);  
}
```

(点击图片可全屏缩放图片)

这下就能正常编译了。

通过上面一些简单的例子，大家应该对 Rust 有了一个初步的印象，更加详细的了解可以去参考官网。

## 让 Rust 开发变 easy 的关键技术点

前面在为什么选择 Rust 里面，我提到了因为 Rust 有几个很 cool 的特性，能让我们写出不容易出错的并发程序。而且我认为，只要理解掌握了这个关键点，用 Rust 进行程序开发就很 easy 了。

### Type safety

Rust 是一门严格要求类型安全的语言，在 C/C++ 的世界里面，我们可以无拘无束的进行类型转换，譬如：

```
char a[4] = {0x11, 0x22, 0x33, 0x44};  
int b = *(int*)&a;
```

(点击图片可全屏缩放图片)

这种在 C/C++ 里面很常见的处理方式，在 Rust 里面是不被允许的，譬如：

```
let a = [11u8, 22u8, 33u8, 44u8];  
let b = a as u32;
```

我们会得到如下编译错误：

```
<anon>:4:13: 4:21 error: non-scalar cast: `[u8; 4]` as `u32`  
<anon>:4    let b = a as u32;
```

如果强制做这种内存转换，我们可以使用 `unsafe`，显示的告诉 Rust 这么做是不安全的，但是你别管了：

```
let mut b: u32;
unsafe {
    let a = [11u8, 22u8, 33u8, 44u8];
    b = *(&a as *const [u8; 4] as *const u32);
}
```

(点击图片可全屏缩放图片)

通常情况下面，Rust 是不允许写上面这样的代码的，所以它将 `unsafe` 的选择权显示的交给了程序员，而我们通过 `unsafe` 也能清晰的知道哪里是不安全的代码，需要注意的。

## Memory safety

下面进入 Rust 最令人抓狂的一个关键点了。Rust 是能够保证程序的 `memory safety` 的，那么是如何保证的呢？首先我们需要了解的是 Rust 里面 `ownership` 以及 `move` 的概念。

## Ownership + move

在 Rust 里面，任何资源只可能有一个 `ownership`，譬如一个最简单的例子：

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

这里我们使用 `let` 将一个 `vector` 给绑定到 `a` 这个变量上面了，我们就可以认为 `a` 现在就是这个 `vector` 的 `ownership`。然后对于这个 `vector` 的 `resouce`，同一个时间只允许一个 `ownership`。我们来看下面这个代码：

```
let a = vec![1, 2, 3];
let b = a;
println!("{}", a[0]);
```

(点击图片可全屏缩放图片)

在上面的例子中，我们将 `a` 赋值给了 `b`，同时继续打印 `a[0]` 的值，这个在大多数语言都没有任何问题的操作，在 Rust 里面，是会报错的，如下：

```
<anon>:4:20: 4:21 error: use of moved value: `a` [E0382]
<anon>:4    println!("{}", a[0]);
```

为什么呢？在打印 `a[0]` 之前，我们进行了 `let b = a` 这样的操作，这个操作，在 Rust 里面叫做 `move`，含义就是把 `a` 对 `vector` 的 `ownership` 移交给了 `b`，`a` 放弃了对 `vector` 的 `ownership`。因为 `a` 已经对这个 `vector` 没有 `ownership` 了，自然就不能访问相关的数据了。

`ownership` 和 `move` 的概念应该算是学习 Rust 第一个坑，我们也很容易写出如下代码：

```
fn do_vec(v: Vec<u32>) {}

fn main() {
    let a = vec![1, 2, 3];
    do_vec(a);
    println!("{}", a[0]);
}
```



(点击图片可全屏缩放图片)

这个代码照样是编译不过的，因为 `do_vec` 这个函数，`a` 已经将对 `vector` 的 `ownership` 给 `move` 了。

所以通常我们只要看到 `let b = a` 这样的代码，就表明 `a` `move` 掉了 `ownership` 了，但有一个例外，如果 `a` 的类型实现了 `copy trait`，`let b = a` 就不是 `move`，而是 `copy` 了，下面的代码是能正常编译的：

```
let a = 1;
let b = a;
println!("{}", a);
```



上面的代码里面，`let b = a`，`a` 并没有 `move`，而是将自己的数据 `copy` 了一份给 `b` 使用。通常基本的数据类型都是实现了 `copy trait`，当然我们也可以将我们自定义的类型实现 `copy`，只是就需要权衡下 `copy` 的性能问题了。

## Borrow

前面我们举了一个 `do_vec` 的例子，如果真的需要在调用这个函数之后继续使用这个 `vector`，怎么办呢？

这里我们就开始接触到了第二个坑，`borrow`。上面说了，`move` 是我给你了 `ownership`，而 `borrow` 则是我借给你这个 `resource` 的使用权，到时候还要还给我：

```
fn do_vec(v: &Vec<u32>) {
}

fn main() {
    let a = vec![1, 2, 3];
    do_vec(&a);
    println!("{}", a[0]);
}
```

 高可用架构

(点击图片可全屏缩放图片)

上面的例子中，我们在参数里面使用了 & 来表示 borrow，do\_vec 这个函数只是借用了 a，然后函数结束之后，还了回来，这样后续我们就能继续使用 a 了。

既然 borrow，当然就能用了，于是我们就可能写这样的代码：

```
fn do_vec(v: &Vec<u32>) {
    v[0] = 1;
}
```

 高可用架构

然后 Rust 又华丽丽的报错了，输出：

```
<anon>:2:4: 2:5 error: cannot borrow immutable borrowed content `*v`
as mutable
<anon>:2   v[0] = 1;
```

 高可用架构

因为我们的 borrow 只是 immutable 的 borrow，并不能改数据。在前面我们也提到过，如果要对一个变量进行修改，必须显示的用 mut 进行声明，borrow 也是一样，如果要对一个 borrow 的东西显示修改，必须使用 mutable borrow，也就是这样：

```
fn do_vec(v: &mut Vec<u32>) {
    v[0] = 1;
}

fn main() {
    let mut a = vec![1, 2, 3];
    do_vec(&mut a);
    println!("{}", a[0]);
}
```

 高可用架构

(点击图片可全屏缩放图片)

borrow 还有 scope 的概念，有时候我们写这样的代码：



```
fn main() {  
    let mut x = 5;  
    let y = &mut x;  
  
    *y += 1;  
  
    println!("{}", x);  
}
```

 高可用架构

(点击图片可全屏缩放图片)

发现编译器又报错了，输出：

```
<anon>:7:20: 7:21 error: cannot borrow `x` as immutable because it is  
also borrowed as mutable [E0502]  
<anon>:7    println!("{}", x);
```

 高可用架构

因为我们之前用 y 来对 x 进行了 mutable 的 borrow，但是还没还回去，所以后面 immutable 的 borrow 就不允许。这个我们可以通过 scope 来显示的控制 mutable 的生存周期：

```
fn main() {  
    let mut x = 5;  
    {  
        let y = &mut x;  
        *y += 1;  
    }  
  
    println!("{}", x);  
}
```

 高可用架构

(点击图片可全屏缩放图片)

这样在 println! 进行 immutable 的 borrow 的时候，y 这个 mutable 的 borrow 已经还回来了。

一个变量，可以同时进行多个 immutable 的 borrow，但只允许一个 mutable 的 borrow，这个其实跟 read-write lock 很相似，同时允许多个读锁，但一次只允许一个写锁。

## Lifetime

在 C++ 里面，相信大家对野指针都印象深刻，有时候，我们会引用了一个已经被 delete 的对象，然后再次使用的时候就 panic 了。在 Rust 里面，是通过 lifetime 来解决这个问题的，不过引入了 lifetime 之后，代码看起来更丑了。一个简单的例子：



```

struct A<'a> {
    b: &'a u32,
}

fn main() {
    let b = 10;
    let mut a = A { b: &b };

    {
        let c = 11;
        a.b = &c;
    }

    println!("{}", a.b);
}

```



(点击图片可全屏缩放图片)

我们定义了一个 struct，里面的 field b 是对外面一个 u32 变量的引用，而这个引用的 lifetime 是 'a。使用 lifetime 能保证 b 引用的 u32 数据的生存周期一定是大于 A 的。

但是在上面的例子中，我们在一个 scope 里面，让 b 引用了 c，但是 c 在 scope 结束之后就没了，这时候 b 就是一个无效的引用了，所以 Rust 会编译报错：

```

<anon>:11:16: 11:17 error: `c` does not live long enough
<anon>:11      a.b = &c;

```



## Thread safety

前面我们提到，Rust 使用 move，borrow 以及 lifetime 这些机制来保证 memory safety，虽然这几个概念不怎么好理解，而且很容易大家写代码的时候就会陷入与编译器的斗争，但是我个人觉得只要理解了这些概念，写 Rust 就不是问题了。

好了，说完了 memory safety，我们马上进入 thread safety 了。大家都知道，多线程的程序很难写，有且稍微不注意，就会出现 data race 等情况，导致数据错误，而且偏偏这样的 bug 还能难查出来。

譬如在 Go 里面，我们可以这样：

```
var n = 10;

go func() {
    n += 1
}()

go func() {
    n += 1
}()
```

 高可用架构

(点击图片可全屏缩放图片)

上面的例子很极端，大家应该也不会这么写代码，但实际中，我们仍然可能会面临 data race 的问题。虽然 Go 可以通过 `--race` 打开 data race 的检查，可通常只会用于 test，而不会在线上使用。

而 Rust 则是在源头上完全让大家没法写出 data race 的代码。首先我们先来了解 Rust 两个针对并发的 trait，Send 和 Sync：

- **Send**

当一个类型实现了 Send，我们就可以认为这个类型可以安全的从一个线程 move 给另一个线程去使用。

- **Sync**

当一个类型实现了 Sync，我们就可以认为这个类型可以在多线程里面通过 shared reference（也就是 Arc）安全的使用。

上面的概念看起来比较困惑，简单一点就是如果一个类型实现了 Send + Sync，那么这个就能在多线程下面安全的使用。

先来看一个简单的例子：


```
fn main() {
    let mut a = vec![1, 2, 3];
    for _ in 0..10 {
        thread::spawn(move || a[0] += 1);
    }

    thread::sleep(Duration::from_millis(50));
}
```

 高可用架构

(点击图片可全屏缩放图片)

上面就是一个典型的多线程 data race 的问题了，Rust 是编译不过的，报错：

```
<anon>:8:31: 8:32 error: capture of moved value:  [E0382] 架构  
<anon>:8      thread::spawn(move || a[0] += 1);
```

前面说了，我们可以通过 Arc 来保证自己的类型在多线程下面安全使用，我们加上 Arc。

```
fn main() {  
    let mut a = Arc::new(vec![1, 2, 3]);  
    for _ in 0..10 {  
        let a = a.clone();  
        thread::spawn(move || a[0] += 1);  
    }  
  
    thread::sleep(Duration::from_millis(50));  
}
```

 高可用架构

(点击图片可全屏缩放图片)

现在我们的 vector 能多线程访问了，但是仍然出错：

```
<anon>:9:31: 9:32 error: cannot borrow immutable borrowed content as mutable  
<anon>:9      thread::spawn(move || a[0] += 1);
```

 高可用架构

(点击图片可全屏缩放图片)

因为我们不光是要多线程 read，而且还需要多线程去 write，自然 Rust 不允许，所以我们需要显示的进行加锁保护，如下：

```
fn main() {  
    let a = Arc::new(Mutex::new(vec![1, 2, 3]));  
    for _ in 0..10 {  
        let a = a.clone();  
        thread::spawn(move || {  
            let mut a = a.lock().unwrap();  
            a[0] += 1  
        });  
    }  
  
    thread::sleep(Duration::from_millis(50));  
}
```

 高可用架构

(点击图片可全屏缩放图片)

所以，如果我们要对一个数据进行安全的多线程使用，最通用的做法就是使用 Arc<Mutex<T>> 或者 Arc<RwLock<T>>进行封装使用。

当然，除了 Arc + Lock 之外，Rust 还提供了 channel 机制方便进行线程之间的数据通讯，channel 类似于 Go 的 channel，一端 send，一端 recv，这里就不详细说明了。

这里在提一点，因为在 Rust 里面，对于多线程使用的数据，我们必须明确的进行 lock 保护，这个编程风格直接带到了后来我们写 Go。在 Go 里面，我以前写 lock，通常会这样：

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



(点击图片可全屏缩放图片)

使用一个 mutex 变量 m 来对数据 v1, v2 进行多线程保护，但这种写法其实很容易就容易忘记这个 lock 到底要保护哪些数据。自从受 Rust 影响了之后，我就喜欢写成这样了：

```
type A struct {  
    m sync.Mutex  
    v1 int  
    v2 int  
}
```

(点击图片可全屏缩放图片)

上面就是显示的将 lock 以及需要保护的数据放到一个 struct 里面，大家一看代码就知道这个 lock 要保护哪些数据了。

## Rust 开发实战经验

前面说了是 Rust 的一些基本 feature，这里开始说下我们项目中用 Rust 的相关经验。

### Cargo

如果要用 Rust 进行项目开发，首先就需要了解的就是 Cargo，Cargo 是 Rust 一个构建以及包管理工具，现在应该已经成了 Rust 开发项目的规范了。Cargo 的使用还是很简单的，大家可以直接去看浏览官网 (<https://crates.io/>)。

### quick\_error!

最开始，我们在写 C 程序的时候，通过定义不同的 int 返回值来表示一个函数是不是有 error。然后到了 C++，我们就可以通过 exception 来处理 error 了。不过到底采用哪种标准，都是没有定论的。

到了 Go，直接约定了函数最后一个返回参数是 error，官方还有一篇 blog 来介绍了 Go 的 error handling (<http://blog.golang.org/error-handling-and-go>)。

在 Rust 里面，error 也有相应的处理规范，就是 Result，Result 是一个 enum，定义是这样的：

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



(点击图片可全屏缩放图片)

也就是说，我们的函数都可以返回 Result，外面去判断，如果是 Ok，那么就是正确的处理，如果是 Err 则是错误了。

这里有篇 error handling 的详细说明 (<https://doc.rust-lang.org/book/error-handling.html>)。

通常大家都会按照上面的规范来处理 error，也就是定义自己的 error，实现其他 error 转成自己 error 的 from 函数，然后在使用 try! 在代码里面简化 error 的处理。

但是很快我们就发现了一个很严重的问题，定义自己的 error，以及将其他 error 转成我们对应的 error 是一件非常冗余复杂的事情，所以我们使用 quick\_error! (<http://dwz.cn/2XZpNo>) 来简化整个流程。

## Clippy

当我第一次写 Go 代码的时候，我对 Go 的 fmt 印象特别深刻，以后再也不用担心编码风格的争论了，Rust 也有相关的 rust fmt，但是更令我惊奇的是 Rust 的 Clippy 这个工具。Clippy 已经不是纠结于编码风格了，而是直接告诉你代码要这么写，那么写不对。

一个很简单的例子：

```
fn main() {  
    let a = "Hello world";  
    let b = a.to_string();  
    println!("{}", b);  
}
```



(点击图片可全屏缩放图片)

这个代码是能编译通过的，但是如果我们打开了 Clippy 的支持，直接会提示：

```
src/main.rs:7:13: 7:26 warning: `a.to_owned()` is faster, #[warn(str_to_string)] on by default
src/main.rs:7      let b = a.to_string();
```



(点击图片可全屏缩放图片)

也就是告诉你，别用 `to_string`，用 `to_owned`。

我们都知道，要开发一个高性能的网络服务，通常的选择就是 `epoll` 这种基于事件触发的网络模型，在 Rust，现阶段成熟的库就是 MIO。MIO 是一个异步 IO 库，对不同的操作系统提供了统一抽象支持，譬如 Linux 下面就是 `epoll`，UNIX 下面就是 `kqueue`，Windows 下是 `IOCP`。不过 MIO 为了统一扩平台，在一些实现上面做了妥协。

## MIO

譬如在 Linux 下面，系统直接提供了 `event fd` 的支持，但 MIO 为了兼容 UNIX，使用了传统的 `pipe` 而不是 `event fd` 来进行 `event loop` 的 `awake` 处理。

这里在单独说下 MIO 提供的另一种线程通讯 `channel` 机制，虽然我们可以用 Rust 自己的 `thread channel` 来进行线程通讯，但如果引入 MIO，我更喜欢用 MIO 自己的 `channel`，主要原因是采用的 `lock free queue`，性能更好，但有 `queue size` 限制问题，发送太频繁但接受端没处理过来，就会造成发送失败的问题了。

## Rust 语言的美中不足

我们团队已经使用 Rust 进行了几个月的开发，当然也遇到了一些很不爽的地方。

首先就是库的不完善，相比于 Go，Rust 的库真的太不完备了。我觉得现阶段 Rust 仍然没有大规模的应用，lib 不完备占了很大一个原因。

TiKV 是一个服务器程序，自然就会涉及到网络编程，官方现阶段的 `net mod` 里面，只有 `block socket` 的支持，这个完全没法用来开发高性能网络程序的。幸好有 MIO，但光有 MIO 是远远不够，在 Go 里面，我们很方便的使用 `gRPC` 来进行 `RPC` 的编写，但是在 Rust 里面，我觉得还得在等很长一段时间，看能不能有开源的实现。

再来就是在 Mac OS X 下面，panic 出来的堆栈完全没法看，没有 file 和 line number 的信息，根本没法方便的查 bug。

当然，毕竟 Rust 是一门比较新的语言，还在不断的完善发展，我们还是很有信心它能越来越好的。

## Q & A

---

### 1. Go 的 Cgo 在效率上面与 Rust FFI 有啥区别？

唐刘：我自己写过一个简单的测试，就是都循环调用 Snappy 的 MaxCompressedLength 这个函数 10000 次，发现 Rust 的 FFI 比 Go 的 Cgo 要快上一个数量级，虽然这么测试不怎么精确，但至少证明了 Rust 的 FFI 性能更好。

### 2. 在官方的介绍中，Rust 的首选平台是 Windows，那是否可以生成 dll. 你们的 IDE 用的是什么？

唐刘：我没用过 Windows，所以也不知道怎么生成 dll，开发 Rust 的 ide 也就是常用的那几个，譬如 Vim，Emacs，Sublime 这些，反正都有 Rust 的插件支持。

### 3. Rust 调用 C 的库方便吗？

唐刘：Rust 通过 FFI 调用 C，很方便的，这里有相关文档（<https://doc.rust-lang.org/book/ffi.html>），但毕竟这涉及到跨语言，代码写起来就不怎么好看了。而且 FFI 需要 unsafe 保护，所以通常我们会在外面在 wrap 一层 Rust 的函数。

### 4. Rust 性能指标如何？

唐刘：Rust 性能这个不怎么好衡量，因为我们只是在一些特定的环境下面做过跟 Go 的对比，譬如 Cgo vs FFI 的测试，这方面性能是比 Go 要好的。另外，Rust 是一门静态语言，没有 GC，所以我觉得他的性能不会是问题，不然 Dropbox 也不可能将类 S3 的应用用 Rust 写了。

### 5. Rust 周边生态如何？如跟常用的 DBSQL/MQ 等第三方系统的 binding？

唐刘：Rust 的生态只能呵呵来形容了，跟 Go，Java 这些的没法比。虽然有常用的 MySQL 等的 binding，但我没用过。主要原因在于官方的网络 IO 是同步的，所以必须借助多线程来进行这些处理，而用 MIO 这种异步模式，大家也知道写出来的代码逻辑切割很厉害。所以通常我们也不会拿 Rust 来做这些复杂的业务系统开发，感觉还是 Go 更合适。

---

本文策划刘芸，海报唐端荣，编辑尤茜、郝亚奇，转播尹雯玉、尹学罡，想讨论更多 Rust 语言开发，请关注公众号获取进群机会。转载请注明来自高可用架构「ArchNotes」微信公众号及包含以下二维码。