

## 04 栈和堆

---

通过上一节课的学习，我们对 Rust 中的变量和值的绑定、主要数据类型以及表达式和语句建立了一个相对直观的了解。感觉好像立马可以使用 Rust 编写很多好玩儿的应用了。但是先别急，让我们暂放一下激动的心，颤抖的手，再次恶补一下计算机的基础：栈（Stack）和堆（Heap）。

同学在学习 JavaScript 的时候，大概率了解过“堆”这个概念。比如下面这段代码：

### JavaScript 代码

```
1 function foo() {
2     const obj = {
3         name: "John Wick",
4         age: 40,
5         pet: "Dog"
6     };
7
8     bar(obj);
9 }
10
11 function bar(obj) {
12     console.log(JSON.stringify(obj));
13 }
14
15 foo();
```

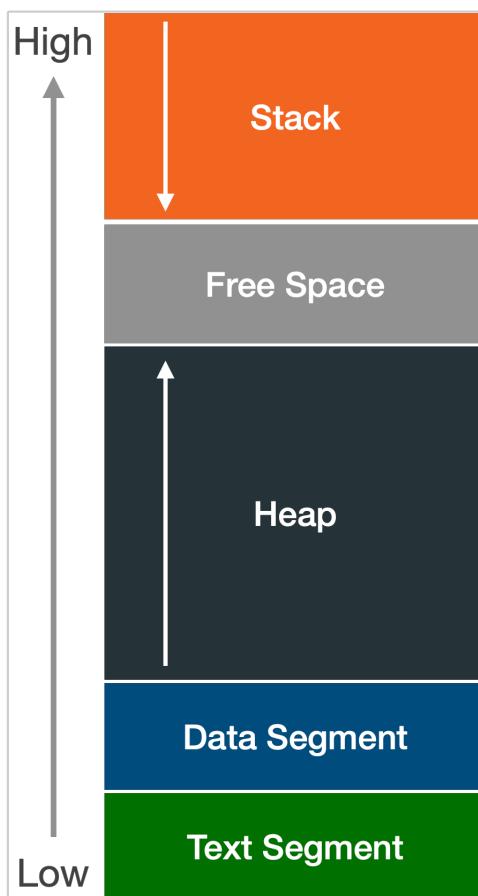
在上面的这段代码中，函数 `foo` 中的对象 `obj` 的“数据”（按照 Rust 的说法，称为“值”）是存储到 JavaScript 引擎的堆上的，变量 `obj` 只是堆上值的一个引用。那么在 `foo` 中调用 `bar(obj)` 的时候，实际上传递的是值的引用。

在学习 JavaScript 的时候，即使不是很了解栈和堆，不区分值到底是存储在哪里，也不影响我们写出非常漂亮的 JavaScript 代码或者 Web 应用。但是学习 Rust 的时候，还是要对栈和堆有一个更加清晰的理解，对于掌握后面的内容有很大的帮助。

本节的内容相对比较抽象，如果第一遍没有特别明白，也不用焦虑，先建立栈和堆的基本概念，随着编写 Rust 代码越来越多，对栈和堆认识也会越来越清晰。

## 程序运行时内存布局

一个可执行的程序运行时，操作系统会给它分配一段内存空间，下图是这段内存空间的结构示意图，自下而上代表内存地址从低到高：



- 栈（Stack），程序执行时的本地变量、函数参数、函数返回地址等
- 预留空间（Free Space）
- 堆（Heap），一段可以伸缩的内存空间，用来存储程序运行时动态生命周期的数据

- 数据段（Data Segment），存储全局变量、静态变量等数据。可以细分为 Initialized Data 和 Uninitialized Data (BSS)
- 代码段（Text Segment），存储程序的机器码

实际上，不同的操作系统的内存布局在细节上可能会有所不同，但是大致上都是上面所说的这几个部分。

## 栈和堆

### 栈（Stack）

- 栈是后进先出的（LIFO，Last In, First Out）
- 在栈上读写数据非常高效
- 栈中的数据由操作系统内核管理
- 栈上的一块内存被称作“栈帧”（Stack Frame），用来存储本地变量、参数、函数的返回地址等信息
- 栈的大小是固定的，并且在程序开始运行的时候就已经固定了
- 栈上的数据只能由当前正在执行的函数访问

栈的空间都不是很大，例如 Windows 上默认分配 1MB。但是很多编译器支持在编译时指定栈的大小，Linux/Unix 操作系统也可以通过修改系统参数来调整栈大小。例如，在我的 macOS 上，栈大小是 8176KB。总结来说，栈的大小和操作系统参数、编译器都有关系。

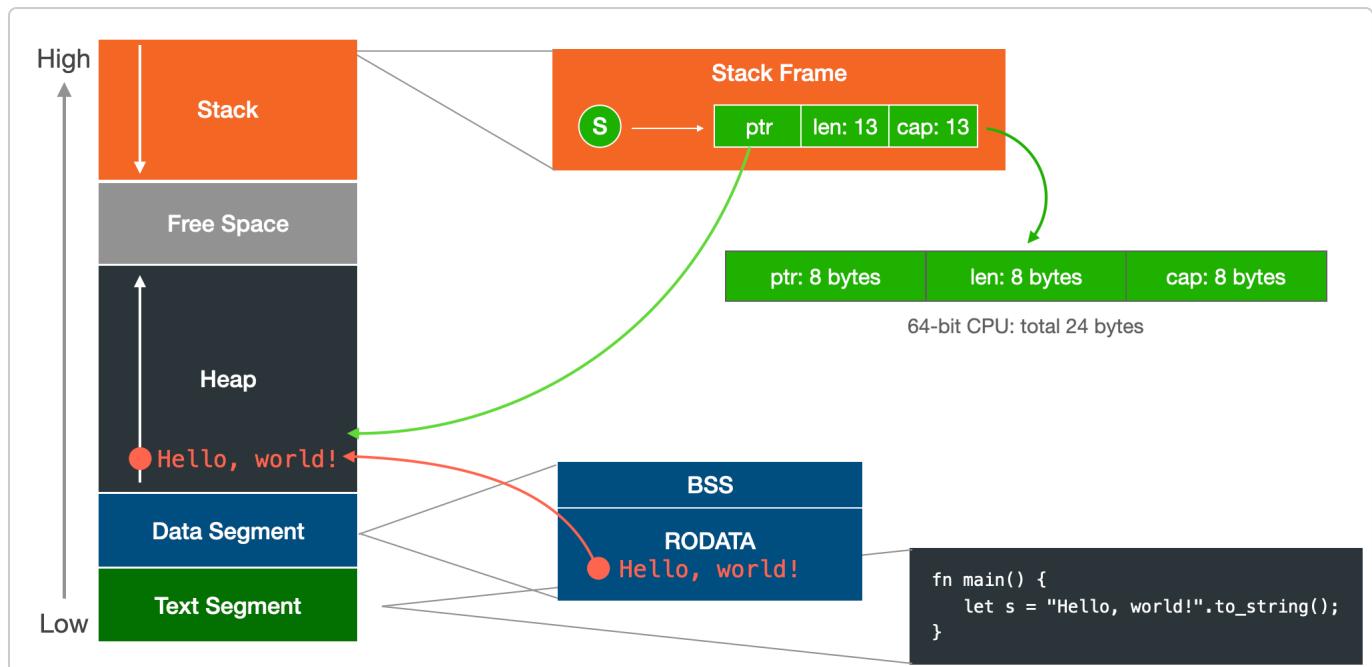
### 堆（Heap）

- 堆是动态大小的
- 程序运行时，从堆上申请内存 (`malloc()`)
- 程序要对自己申请的内存空间负责，用完了要释放 (`free()`)
- 堆内存上的数据访问效率不如栈高
- 堆上可以用来存储较大的数据
- 堆上的数据可以被多个线程共享

例如，下面这段 Rust 代码：

```
1 fn main() {
2     let s = "Hello, world!".to_string();
3 }
```

在运行时栈和堆的变化示意图如下：



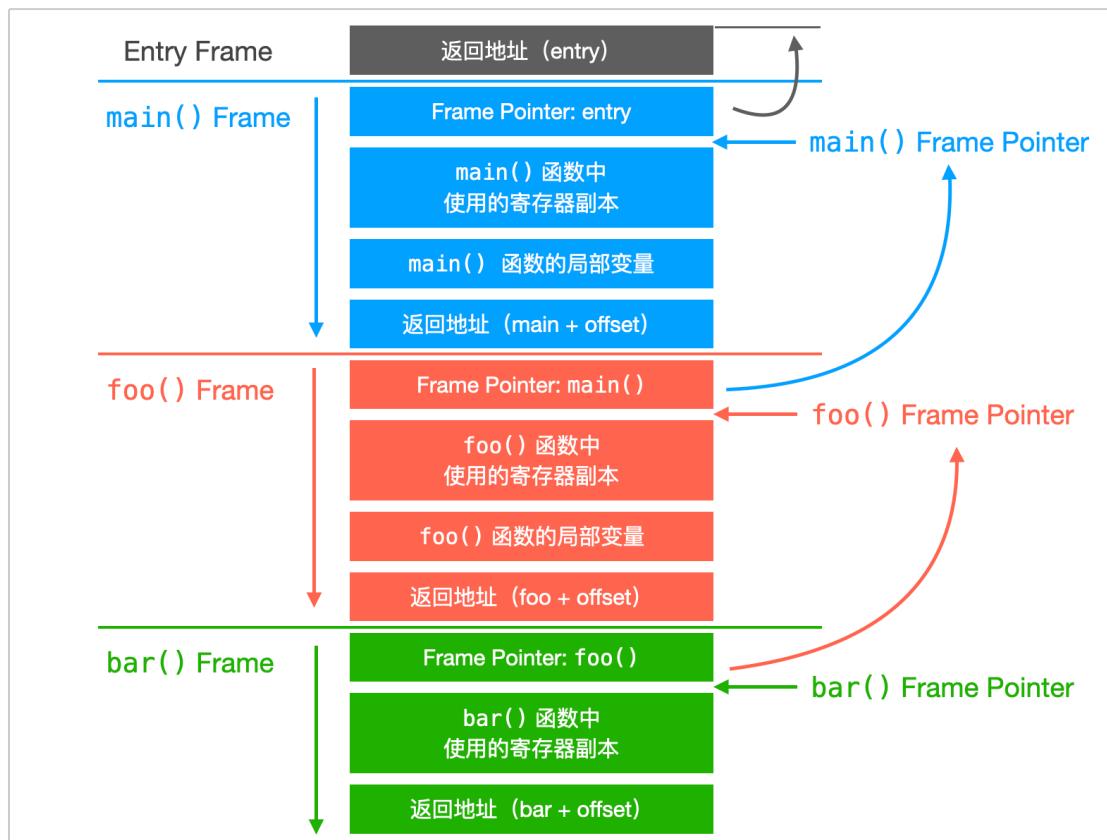
1. 编译后的代码在 Text Segment
2. 代码中的字面量“Hello, world!”存储在 BSS 段
3. `"Hello world!".to_string()` 方法执行的时候，会从 BSS 段复制字符串到堆上
4. 栈上会开辟一个栈帧，栈帧中有一个变量 **s**，占用 24 个字节，第一组 8 字节记录的是堆上值的地址，第二组 8 字节是堆上这段内存空间的长度，最后一组 8 个字节是堆上这段内存的总容量

在函数调用中，操作系统会给程序分配栈帧，栈帧用完之后，就会被释放（LIFO），这样就可以释放掉在栈上占用的空间。例如下面这段示意代码：

```

1 fn main() {
2     foo();
3 }
4
5 fn foo(foo_args) {
6     bar();
7 }
8
9 fn bar(bar_args) {
10}
11}

```



上图是一个示意图，有一些细节并没有体现出来。程序开始的时候，操作系统会在栈上分配入口栈帧，然后进入 `main()` 函数，随着 `main()` 函数调用 `foo()` 函数、`foo()` 函数调用 `bar()` 函数，栈自顶向下增长，当 `bar()` 函数执行完毕，然后 `foo()` 函数执行完毕，栈帧指针依次收缩，释放了栈上的空间。释放栈上空间只需改变栈帧指针的地址，所以它是非常高效的。

那么，操作系统是如何决定栈帧大小的呢？秘密就在于编译器中。函数是编译器的最小编译单元，编译器知道每个函数中需要用到哪些寄存器，有哪些变量，以及这些变量分别需要多大的空间。根据这些信息，编译器知道函数执行时需要开辟多大的栈帧。所以，栈上存放的都是已知大小的值，对于那些编译期未知大小的值，或者所需内存在运行期动态变动的值，都是需要放到帧上的。

虽然栈有诸多的优点，但是栈的空间比较小，所以如果在栈上存储大量的数据，容易造成栈溢出（Stack Overflow）的问题。最常见的场景就是当递归调用的时候，每次调用都会形成一个新的栈帧，栈持续的自顶向下增长，如果不能正确终止递归，就很容易出现栈溢出。

那么，使用堆内存就不会出现问题吗？也不是，使用堆内存最常见的错误就是：

1. 堆越界问题（Heap Out of Bounds）。多个线程并发访问堆上的值，有的读取，有的写入，如果控制不当，会导致堆越界
2. 使用已释放的堆内存（Use after Free）。前面说到过，堆上的内存由程序申请（`malloc()`），同样也是由程序负责释放（`free()`）。如果针对一段已经释放的堆内存，继续访问其上的数据，会导致使用已释放内存的错误

如何让程序员安全便捷的使用内存，是编程语言的一个课题。很多语言提供了自动内存管理方案，省去了程序员手动申请和释放内存的麻烦。常见的自动内存管理方案有两种：

1. 追踪式垃圾收集（Tracing Garbage Collection）。通过标记（Mark）找出可以不再使用的内存，然后再进行清理（Sweep）。代表语言有 Java（JVM）、JavaScript（V8 Engine）。它的优势是效率高，吞吐量大。缺点是对整个堆内存进行清理的时候（Full GC），容易发生 STW（Stop the World）现象，此时，应用对外的表现明显延迟。虽然说可能只是短短的几毫秒或几十毫秒的 STW，依然可以被用户感知到的延迟
2. 自动引用计数（Automatic Reference Counting）。编译器为函数插入语句来维护对象的引用计数，当引用计数为 0 时，释放内存。代表语言有 ObjC、Swift。它的优势是用完即释放，更加平滑。短板就是编译器为了实现 ARC，在编译的过程中，插入了大量的代码

但是，Rust 却另辟蹊径搞出来一个叫做“所有权”（Ownership）的概念，非常独特。切切实实的解决了很多内存使用方便的问题。下一节课我们将详细讲解。

## 回顾

1. 栈和堆的特点
2. 栈上存放已知大小的值
3. 堆上存放动态大小的值
4. 常见的自动内存管理方案：GC 和 ARC