rHamal

一次绝少人知的 Rust 学习历程

目录

1 原点 1
1.1 最简单的程序 1
1.2 你好啊，原点! 1
1.3 结构体 4
1.4 成就感来自 Pov-Ray 5
1.5 小结 9

2 很多点 11
2.1 路径 11
2.2 打开文件 12
2.3 字符串变量 13
2.4 枚举 14
2.5 模式匹配 15
2.6 读取文件内容 17
2.7 所有权、转移和借用 20
2.8 灵异事件 26
2.9 逐行读取文件 28
2.10 字符串分割 30
2.11 很多点 31
2.12 unwrap 33
2.13 小结 34

3 点集 35
3.1 字符串 → 数字 35
3.2 数组和向量 36
3.3 String 和 &str 37
3.4 点集 39
3.5 Pov-Ray：模型与视图的分离 41
3.6 写文件 41
3.7 包围盒 43
3.8 小结 44

4 外界 45
4.1 命令行参数 45
4.2 文件名 46
4.3 Option 46
4.4 不含扩展名的文件名 47
4.5 小结 48

5 rHamal v0.1 49
5.1 Pov-Ray 场景文件概览 49
5.2 复合模型 50
5.3 rHamal v0.1 源码 52
5.4 编译、安装及用法 57
5.5 若求美好，要自己动手 58
5.6 小结 61

6 方法 63
6.1 静态方法 63
6.2 实例方法 64
6.3 收纳 65
6.4 字符串续行符 66
6.5 小结 67

7 玄之又玄 69
7.1 泛型类型 69
7.2 泛型函数 70
7.3 Trait 72
7.4 小结 73

1 原点

事情，总要有个起点。学习 Rust，我从三维世界的原点开始。

1.1 最简单的程序

最简单的程序是 Hello world 吗？不是。最简单的程序是什么也不做的程序。下面是用 Rust 语言写的一个什么也不做的程序：

```
fn main() {}
```

事实上，它现在还不能称为程序，只能称为一段 Rust 代码。假设这段代码保存在 foo.rs 文件里，可使用 Rust 语言编译器 rustc 将其编译为程序 foo：

```
$ rustc foo.rs
```

因为程序 foo 什么也不做，所以执行它，

```
$ ./foo
```

自然也是什么都没做，便结束了。

1.2 你好啊，原点！

三维世界的原点，就是在三个数轴上的投影为 0 的点。假设三个数轴为 x，y 和 z，则原点在它们上的投影可表示为

$$x = 0$$

$$y = 0$$

$$z = 0$$

这些投影，用 Rust 代码可表示为

```
fn main() {
    let x: f64 = 0;
    let y: f64 = 0;
    let z: f64 = 0;
}
```

但是，上述代码却无法通过编译，rustc 报错如下：

```
error[E0308]: mismatched types
 --> foo.rs:2:18
  |
2 |     let x: f64 = 0;
  |            ---   ^
  |            |     |
  |            |     expected `f64`, found integer
  |            |     help: use a float literal: `0.0`
  |            expected due to this

error[E0308]: mismatched types
 --> foo.rs:3:18
  |
3 |     let y: f64 = 0;
  |            ---   ^
  |            |     |
  |            |     expected `f64`, found integer
  |            |     help: use a float literal: `0.0`
  |            expected due to this

error[E0308]: mismatched types
 --> foo.rs:4:18
  |
4 |     let z: f64 = 0;
  |            ---   ^
  |            |     |
  |            |     expected `f64`, found integer
  |            |     help: use a float literal: `0.0`
  |            expected due to this

error: aborting due to 3 previous errors

For more information about this error, try `rustc --explain E0308`.
```

rustc 认为，我将一个整型（Integer）类型的值 0 赋给了 64 位的浮点类型（f64），类型不匹配。既然如此，那么

```rust
fn main() {
    let x: f64 = 0.0;
    let y: f64 = 0.0;
    let z: f64 = 0.0;
}
```

总该能行吧？的确可以，但是 rustc 依然给出了以下警告：

```
warning: unused variable: `x`
 --> foo.rs:2:9
  |
2 |     let x: f64 = 0.0;
  |         ^ help: if this is intentional, prefix it with an underscore: `_x`
  |
  = note: `#[warn(unused_variables)]` on by default

warning: unused variable: `y`
 --> foo.rs:3:9
  |
3 |     let y: f64 = 0.0;
  |         ^ help: if this is intentional, prefix it with an underscore: `_y`

warning: unused variable: `z`
 --> foo.rs:4:9
  |
4 |     let z: f64 = 0.0;
  |         ^ help: if this is intentional, prefix it with an underscore: `_z`

warning: 3 warnings emitted
```

之所以出现这些警告，是因为代码里定义的三个变量 x、y 和 z 未被使用。rustc 的眼睛里，竟是如此地揉不得沙子。

将 x、y 和 z 的值输出到终端，算不算使用了它们？试试看，

```
fn main() {
    let x: f64 = 0.0;
    let y: f64 = 0.0;
    let z: f64 = 0.0;
    println!("你好啊，({0}, {1}, {2})！", x, y, z);
}
```

依然假设上述代码是 foo.rs 文件的全部内容，编译 foo.rs 得到 foo 程序，再执行后者，该过程及 foo 程序的输出如下：

```
$ rustc foo.rs
$ ./foo
你好啊，(0, 0, 0)！
```

1.3 结构体

倘若将 x、y 和 z 绑定在一起，就有了一个真正的抽象意义的三维原点。此愿可基于 Rust 的结构体类型予以达成，例如：

```
struct point {
    x: f64, y: f64, z: f64
}

fn main() {
    let origin: point = point {x: 0.0, y: 0.0, z: 0.0};
    println!("你好啊，({0}, {1}, {2})！",
             origin.x, origin.y, origin.z);
}
```

我自认为已将代码写得无懈可击了，但 rustc 依然警告：

```
warning: type `point` should have an upper camel case name
 --> foo.rs:1:8
  |
1 | struct point {
  |        ^^^^^ help: convert the identifier to upper camel case: `Point`
  |
  = note: `#[warn(non_camel_case_types)]` on by default

warning: 1 warning emitted
```

rustc 希望我将上述代码写为

```
struct Point {
    x: f64, y: f64, z: f64
}

fn main() {
    let origin: Point = Point {x: 0.0, y: 0.0, z: 0.0};
    println!("你好啊，({0}, {1}, {2})！",
             origin.x, origin.y, origin.z);
}
```

听它的吧！结构体类型（struct）是为用户定义自己的类型而设。用户自定义的类型的名称——如 point，Rust 语言建议使用骆驼命名法，即每个单词的首字母大写——如 Point。

由于 rustc 能够根据值的语法形式自动推断出变量的类型，因此将

```
let origin: Point = Point {x: 0.0, y: 0.0, z: 0.0};
```

写为

```
let origin = Point {x: 0.0, y: 0.0, z: 0.0};
```

rustc 不以为忤。

1.4 成就感来自 Pov-Ray

事实上，仅现在所学的 Rust 编程技术已经足以写出一个实用的程序了，只要认同陆游所说的，汝果欲学诗，工夫在诗外。倘若在 Rust 语言之外，对 Pov-Ray 也略微熟悉，那么便可在计算机里创造三维世界。

对于阅读本文档而言，不熟悉 Pov-Ray 也不要紧，但是需要系统里已经安装了 Pov-Ray。我想，对于一个有志于学习 Rust 的人而言，在自己日常使用的系统里安装 Pov-Ray 并非难事。如果是 Linux 系统，多数 Linux 发行版的软件仓库里提供了 Pov-Ray，例如在 Ubuntu 系统里，只需

```
$ sudo apt install povray
```

在 Gentoo 系统里，只需

```
$ sudo emerge -avt povray
```

倘若是 Windows 系统，需要去

```
https://www.povray.org/download/
```

下载安装包。至于 Mac OS 系统，我没用过，可参考

```
http://megapov.inetart.net/povrayunofficial_mac/
```

有了 Pov-Ray，便可编为其写场景文件。例如，绘制三维原点的场景文件 foo.pov：

```
1  #version 3.7;
2  #include "colors.inc"
3  global_settings {assumed_gamma 1.0}

4  background {color White}
5  sphere {<0, 0, 0>, 0.3 pigment {color Gray50}}
6  light_source {<3, 3, -3> color White}
7  camera {location <0, 0, -3> look_at <0, 0, 0>}
```

使用 `povray` 命令可基于 foo.pov 描述的三维场景生成图片 foo.png：

```
$ povray +A foo.pov
```

生成的 foo.png，如图 1.1 所示。

图 1.1　三维世界的原点

文件 foo.pov 的前三行内容可视为 Pov-Ray 3.7 版本所需要的固定格式，真正描述三维场景的是 4 - 7 行，其中

```
background {color White}
```

将三维场景的背景颜色设为白色。继而，代码

```
sphere {<0, 0, 0>, 0.3 pigment {color Gray50}}
```

以三维原点为中心，半径为 0.3，颜色为中度灰色，绘制球体。代码

```
light_source {<3, 3, -3> color White}
```

在三维坐标为 (3, 3, -3) 的地方安置一个白色的光源。代码

```
camera {location <0, 0, -3> look_at <0, 0, 0>}
```

在三维坐标为 (0, 0, -3) 的地方安置一个相机（不妨理解为场景的观察者），对准原点。

Pov-Ray 的三维世界坐标系是左手系，其 X 轴由屏幕左边指向右边，Y 轴由屏幕下方指向上方，Z 轴由屏幕外部指向内部。只需要熟悉这个坐标系，再结合上述对 foo.pov 所描述的场景的解释，应该不难理解为什么能由 foo.pov 得到图 1.1。

文件 foo.pov 里有 3 个三维世界的点，倘若在 Rust 程序里定义它们，然后 foo.pov 便可由 Rust 程序生成。试试看，

```
struct Point {
    x: f64, y: f64, z: f64
}
```

```rust
fn main() {
    let origin = Point {x: 0.0, y: 0.0, z: 0.0};
    let r = 0.3;
    let light_source = Point {x: 3.0, y: 3.0, z: -3.0};
    let camera = Point {x: 0.0, y: 0.0, z: -3.0};
    println!("#version 3.7;");
    println!("#include "colors.inc"");
    println!("global_settings {assumed_gamma 1.0}");
    println!("background {color White}");
    println!("sphere {<{0}, {1}, {2}>, {3} pigment {color Gray50}}",
             origin.x, origin.y, origin.z, r);
    println!("light_source {<{0}, {1}, {2}> color White}",
             light_source.x, light_source.y, light_source.z);
    println!("camera {location <{0}, {1}, {2}> look_at <{3}, {4}, {5}>}",
             camera.x, camera.y, camera.z, origin.x, origin.y, origin.z);
}
```

rustc 觉得上述代码存在的问题实在是太多了……第一个问题是

```
error: suffixes on a string literal are invalid
  --> foo.rs:11:14
   |
11 |     println!("#include "colors.inc"");
   |              ^^^^^^^^^^^^^^^^^^ invalid suffix `colors`
```

这个问题是 Rust 的字符串之要用引号予以表示，如果字符串内含有引号，必须使用 \ 进行转义。因此上述代码里的

```rust
println!("#include "colors.inc"");
```

需修改为

```rust
println!("#include \"colors.inc\"");
```

\ 能用于许多特殊字符的转移，包括它自身，即 \\。第二个问题是

```
error: invalid format string: expected `'}'`, found `'1'`
  --> foo.rs:12:46
   |
12 |     println!("global_settings {assumed_gamma 1.0}");
   |                               -              ^ expected `}` in format string
   |                               |
   |                               because of this opening brace
```

```
   |
   = note: if you intended to print `{`, you can escape it using `{{`
```

字符串里如果含有 { 和 }，也需要进行转义，因为这两个符号在 Rust 语言里用于字符串的格式化。不过，它们的转义不能用 \。Rust 语法单独为它们提供了转义方式，即 {{ 和 }}。例如，上述代码里的

```
println!("global_settings {assumed_gamma 1.0}");
```

需修改为

```
println!("global_settings {{assumed_gamma 1.0}}");
```

因此，能够通过 rustc 编译的最终代码如下：

```
struct Point {
    x: f64, y: f64, z: f64
}

fn main() {
    let origin = Point {x: 0.0, y: 0.0, z: 0.0};
    let r = 0.3;
    let light_source = Point {x: 3.0, y: 3.0, z: -3.0};
    let camera = Point {x: 0.0, y: 0.0, z: -3.0};
    println!("#version 3.7;");
    println!("#include \"colors.inc\"");
    println!("global_settings {{assumed_gamma 1.0}}");
    println!("background {{color White}}");
    println!("sphere {{<{0}, {1}, {2}>, {3} pigment {{color Gray50}}}}",
             origin.x, origin.y, origin.z, r);
    println!("light_source {{<{0}, {1}, {2}> color White}}",
             light_source.x, light_source.y, light_source.z);
    println!("camera {{location <{0}, {1}, {2}> look_at <{3}, {4}, {5}>}}",
             camera.x, camera.y, camera.z, origin.x, origin.y, origin.z);
}
```

依然假设上述代码是 foo.rs 的全部内容，编译 foo.rs，运行所得程序 foo，通过输出重定向，便可将 foo 的输出写入 foo.pov，亦即

```
$ rustc foo.rs
$ ./foo > foo.pov
$ povray +A foo.pov
```

最终可得 foo.png，即图 1.1。

1.5 小结

通过 Rust 程序生成了一个能绘制三维原点的 Pov-Ray 场景文件，真的很有成就感吗？现在还没有。显然直接写 foo.pov，比写一个 Rust 程序 foo 并用它生成 foo.pov 更为直接有效。但是，倘若要绘制的不止原点，而是成千上万的点，用于生成 Pov-Ray 场景文件的 Rust 程序——当然不是本文的 foo 程序——便非常有必要。

2 很多点

有很多点的时候，应当将它们保存在文件里，文件可以是很简单的纯文本文件，例如 foo.asc：

```
1.0 2.0 3.0
0.1 0.5 0.7
1.3 4.6 7.1
... ... ...
```

每一行的内容是一个点的三维坐标。这样的文件格式，是几乎所有三坐标测量设备事实上的标准输出格式。换言之，有成千上万的点，它们可以来自现实世界。如何编写一个 Rust 程序，令它基于 foo.asc 这样的文件生成 Pov-Ray 场景文件，从而直观地将 foo.asc 记录的三维点集呈现出来？例如

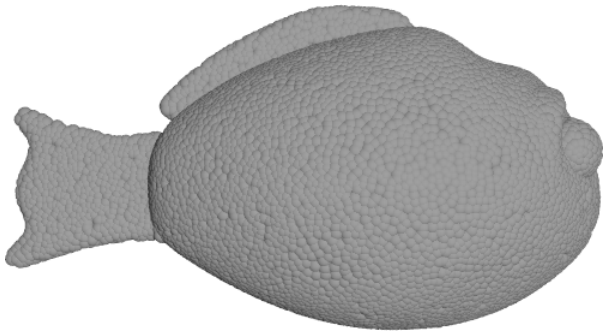

图 2.1　鱼形点集（点数：4967）

2.1 路径

要打开一个文件，访问其内容，前提是要有一条指向它的路径。假设 foo.rs 和 foo.asc 文件皆在同一目录下，使用 Rust 标准库提供的 path 模块可创造指向 foo.asc 的路径。例如

```rust
use std::path::Path;

fn main() {
    let path = Path::new("foo.asc");
    println!("点集文件: {}", path.display());
}
```

假设上述代码是 foo.rs 的全部内容，编译 foo.rs，运行所得程序 foo，便可输出文件路径：

```
$ rustc foo.rs
$ ./foo
```

```
点集文件: foo.asc
```

倘若 foo.asc 位于 foo.rs 所在目录的 data 目录内，那么 foo.rs 需要写成

```
use std::path::Path;

fn main() {
    let path = Path::new("data/foo.asc");
    println!("点集文件: {}", path.display());
}
```

注意，在 `println!` 的字符串格式化输出里，格式化参数的编号可以省略。

事实上，直接用字符串作为文件路径，并无不可，但 path 模块有助于消除操作系统的差异，例如在 Windows 系统里，路径 `data/foo.asc` 需要表示为 `data\foo.asc`。

`Path` 是 `path` 模块的一个结构体，只是这个结构体还拥有一些与它密切相关的函数，这样的函数，在 Rust 语言里称为方法（Method），例如 `Path` 的 `display` 方法，可将 `Path` 结构体信息转化为字符串。

2.2 打开文件

通过打开一个文件，便可知道文件的路径是否正确了。对于文件的打开、读取和写入，可基于 Rust 标准库的 fs 模块提供的结构体 `Path` 以及与之相关的函数予以实现。

用于打开文件的函数是 `std::fs::File::open`。此刻我并不知道这个函数如何使用，但是总可以一试：

```
use std::path::Path;
use std::fs::File;

fn main() {
    let path = Path::new("data/foo.asc");
    let file = File::open(path);
}
```

rustc 对上述代码给出了以下警告

```
warning: unused variable: `file`
 --> foo.rs:6:9
  |
6 |     let file = File::open(path);
  |         ^^^^ help: if this is intentional, prefix it with an underscore: `_file`
  |
  = note: `#[warn(unused_variables)]` on by default
```

```
warning: 1 warning emitted
```

因为变量 `file` 创建了，却没使用，我是想使用的，但是现在尚不知如何用。为了验证上述代码里 `File::open` 是否将文件打开，可考虑将文件的内容读入到字符串变量，然后将后者输出到终端。

2.3 字符串变量

能够用于存储不确定的字符串信息的变量，可通过 Rust 标准库提供的 String 模块里的函数予以创建或构造。例如，使用 `String::new` 方法可构造一个空的字符串，然后为其增加内容：

```
fn main() {
    let s = String::new();
    s = s + "Hello world!";
    println!("{}", s);
}
```

上述代码只是几乎正确，因为 rustc 会报错：

```
error[E0384]: cannot assign twice to immutable variable `s`
 --> foo.rs:3:5
  |
2 |     let s = String::new();
  |         -
  |         |
  |         first assignment to `s`
  |         help: make this binding mutable: `mut s`
3 |     s = s + "Hello world!";
  |     ^ cannot assign twice to immutable variable

error: aborting due to previous error
```

在 Rust 语言里，变量一旦赋值，默认不再能再变，除非在定义变量时，明确表示它是可变的，即增加 `mut` 标记：

```
fn main() {
    let mut s = String::new();
    s = s + "Hello";
    s = s + " world!";
    println!("{}", s);
}
```

编译上述修正后的代码，运行所得程序，可在终端输出

```
Hello world!
```

2.4 枚举

在 2.2 节尝试打开文件的代码里，`file` 的类型是结构体 `File`，该类型有个方法 `read_to_string`，可将文件的内容读入到一个字符串变量，例如：

```
use std::path::Path;
use std::fs::File;

fn main() {
    let path = Path::new("data/foo.asc");
    let file = File::open(path);
    let mut s = String::new();
    file.read_to_string(s);
}
```

果然不出所料，上述代码让 rustc 似乎有些绝望：

```
error[E0599]: no method named `read_to_string` found for enum
`Result<File, std::io::Error>` in the current scope
 --> foo.rs:8:10
  |
8 |     file.read_to_string(s);
  |          ^^^^^^^^^^^^^^ method not found in `Result<File, std::io::Error>`

error: aborting due to previous error
```

rustc 认为 `file` 变量并没有一个叫作 `read_to_string` 的方法。这意味着什么呢？这意味着上文里，我说 `file` 的类型是 `File`，仅仅是一厢情愿，rustc 认为它的类型是

```
Result<File, std::io::Error>
```

这是一个什么类型呢？是 Rust 标准库定义的枚举类型。

枚举（enum）与结构体（struct）相似，是 Rust 语言为用户定义自己的类型而设，通常用于制作一些类似于标签的东西。例如，

```
enum MyResult {
    ok,
    error
}
```

不过，未卜先知，rustc 会抱怨 ok 和 error 未使用骆驼命名法。因此，上述代码应修改为

```
enum MyResult {
    Ok,
    Error
}
```

rustc 管得太多了，但人在屋檐下，怎能不低头？

枚举能用来做什么呢？若想从直觉上理解枚举的价值，需要学习如何定义函数，确切地说，是如何定义一个有参数或者有返回值的函数。

2.5 模式匹配

事实上，我已经定义过 Rust 函数，即

```
fn main() {
    ... ... ...
}
```

更一般的 Rust 函数，有参数，也有返回值，例如

```
fn foo (a: i32) -> MyResult {
    if a == 0 {
        return MyResult::Ok;
    } else {
        return MyResult::Error;
    }
}
```

函数 foo 接受一个 32 位整型类型的参数值，返回枚举类型 MyResult 的值，除此以外，趁机学了 Rust 语言的条件语法。

倘若将 MyResult::Ok 和 MyResult::Error 视为两种状态，前者表示函数执行成功，后者表示函数执行失败，则函数能够用一种更贴近直觉的方式，以一种用户自定义的类型，返回此两种状态之一。不过，使用枚举类型作为函数的返回值类型，存在一个问题，即执行函数后，如何判断它返回哪种状态呢？

对于上述代码定义的函数 foo，运用刚学的条件语法，可写出以下代码：

```
let a = 0;
let b = foo(a);
if b == MyResult::Ok {
    println!("Ok!");
}
if b == MyResult::Error {
```

```
    println!("Error!");
}
```

不过，上述代码无法通过编译，rustc 报错说，运算符 == 不能用于类型为 MyResult 的值的比较。不要沮丧，因为 Rust 语言为此事提供了更好的语法——模式匹配。与上述条件语法所表达的逻辑相符的模式匹配语句为

```
let a = 0;
match foo(a) {
    MyResult::Ok => println!("Ok!"),
    MyResult::Error => println!("Error!")
}
```

显然，模式匹配语法不仅更简洁，甚至可以省却条件语句里的中间变量 b。

枚举类型的分量（或成员）可以带有特定类型。例如，

```
enum MyResult {
    Ok(i32),
    Error(String)
}
```

基于这个新的 MyResult，函数 foo 可定义为

```
fn foo (a: i32) -> MyResult {
    if a == 0 {
        return MyResult::Ok(1);
    } else {
        return MyResult::Error(String::from("Error!"));
    }
}
```

事实上，其中的条件语句也可以改为模式匹配语句：

```
fn foo (a: i32) -> MyResult {
    match a {
        0 => return MyResult::Ok(1),
        _ => MyResult::Error(String::from("Error!"))
    }
}
```

模式匹配里的 _ 表示除了之前出现的模式之外，其它的所有模式。

为了展现这个健壮的 foo 的全貌，下面给出能够被 rustc 编译为程序的全部代码：

```
enum MyResult {
```

```
    Ok(i32),
    Error(String)
}

fn foo (a: i32) -> MyResult {
    match a {
        0 => return MyResult::Ok(1),
        _ => MyResult::Error(String::from("Error!"))
    }
}

fn main() {
    let a = 1;
    match foo(a) {
        MyResult::Ok(v) => println!("{}", v),
        MyResult::Error(e) => println!("{}", e)
    }
}
```

编译所得程序的输出结果是 `Error!`

2.6 读取文件内容

Rust 标准库里也定义了一个与上一节定义的 `MyResult` 类似但更强大的枚举类型 `Result`。以下代码，

```
use std::path::Path;
use std::fs::File;

fn main() {
    let path = Path::new("data/foo.asc");
    let file = File::open(path);
    let mut s = String::new();
    file.read_to_string(s);
}
```

`File::open` 函数的返回值 `file` 便是 `Result` 类型，因此它并没有类型为 `File` 的变量所拥有的 `read_to_string` 方法，但是 `File::open` 函数的返回值里的确包含了 `File` 的变量，只是需要用模式匹配进行提取，即

```
let mut file = match File::open(path) {
    Result::Ok(v) => v,
```

```
    Result::Err(e) => panic!("{}", e),
};
```

由于 Result 是 Rust 标准库里的类型，rustc 会自动载入它的路径，即

```
use Result::*;
```

因此在上述的模式匹配语句里，Result:: 可省略，亦即

```
let mut file = match File::open(path) {
    Ok(v) => v,
    Err(e) => panic!("{}", e),
};
```

在上述代码里，当文件打开出错时，使用 panic! 令程序报错后终止。

现在，将上述模式匹配代码植入 main 函数里，可得到最新的 foo.rs：

```
use std::path::Path;
use std::fs::File;

fn main() {
    let path = Path::new("data/foo.asc");
    let mut file = match File::open(path) {
        Ok(v) => v,
        Err(e) => panic!("Error: {}", e),
    };
    let mut s = String::new();
    file.read_to_string(s);
}
```

此时，rustc 的编译结果依然很不乐观，报错信息如下

```
error[E0599]: no method named `read_to_string` found for struct `File` in the current
scope
   --> foo.rs:11:10
    |
11  |     file.read_to_string(s);
    |          ^^^^^^^^^^^^^^ method not found in `File`
    |
    = help: items from traits can only be used if the trait is in scope
help: the following trait is implemented but not in scope; perhaps add a `use`
for it:
    |
```

```
1   | use std::io::Read;
    |

error: aborting due to previous error
```

不过，rustc 给出了指导，让我在代码里增加

```
use std::io::Read;
```

姑且一试，于是 foo.rs 内容变为

```
use std::path::Path;
use std::fs::File;
use std::io::Read;

fn main() {
    let path = Path::new("data/foo.asc");
    let mut file = match File::open(path) {
        Ok(v) => v,
        Err(e) => panic!("Error: {}", e),
    };
    let mut s = String::new();
    file.read_to_string(s);
}
```

新的 foo.rs 虽然依然导致 rustc 报错，但是刚才的错误消失了。这意味着，`File` 类型的值的 `read_to_string` 方法是在 `std::io::Read` 里实现的。虽然对此非常不解，但是现在还是装作理解了吧，不拘小节。

rustc 给出的新的错误信息是

```
error[E0308]: mismatched types
  --> foo.rs:12:25
   |
12 |     file.read_to_string(s);
   |                         ^
   |                         |
   |                         expected `&mut String`, found struct `String`
   |                         help: consider mutably borrowing here: `&mut s`
```

rustc 似乎在建议我，将 `file.read_t_string` 的参数 `s` 更改为 `&mut s`，那么 `&mut` 是什么？

2.7 所有权、转移和借用

若想理解何为 &mut，前提是要理解变量的所有权及其转移。

首先，每个变量都诞生于一个作用域。例如

```
fn main() {
    let x = 3;
}
```

x 的作用域就是 main 函数。倘若将 x 作为参数传给一个函数，例如

```
fn foo(a: i32) {
    println!("{}", a);
}

fn main() {
    let x = 3;
    foo(x);
}
```

那么，x 的所有权便由函数 main 移交于函数 foo 了。由于 Rust 语言规定，无论何时，变量的所有权是唯一的，亦即仅能有一个作用域拥有变量的所有权。因此，函数 main 将 x 的所有权移交于函数 foo 之后，它便不能再继续使用 x 了。例如，以下代码

```
fn foo(a: i32) {
    println!("{}", a);
}

fn main() {
    let x = 3;
    foo(x);
    println!("{}", x);
}
```

结果很打脸，上述代码完美地通过了 rustc 的编译，程序也如预想一致，输出了

```
3
3
```

因为这是例外。对于基本类型的变量，诸如布尔类型、字符类型、各种整型或浮点型变量，不需要考虑所有权问题，无论是将它们的值赋予其它变量，还是作为参数传递给函数，rustc 传递的只是它们的值的副本。对于复杂的数据类型，就需要考虑变量的所有权了。以结构体类型为例，以下代码

```rust
struct Point {x: f64, y: f64, z: f64}

fn foo(a: Point) {
    println!("({}, {}, {})", a.x, a.y, a.z);
}

fn main() {
    let a = Point {x: 1.0, y: 2.0, z: 3.0};
    foo(a);
    println!("({}, {}, {})", a.x, a.y, a.z);
}
```

便会令 rustc 报错：

```
error[E0382]: borrow of moved value: `x`
  --> foo.rs:10:40
   |
8  |     let x = Point {x: 1.0, y: 2.0, z: 3.0};
   |         - move occurs because `x` has type `Point`, which does not implement
the `Copy` trait
9  |     foo(x);
   |         - value moved here
10 |     println!("({}, {}, {})", x.x, x.y, x.z);
   |                                        ^^^ value borrowed here after move
```

同理，将一个结构体变量的值赋给另一个变量，也会发生所有权转移，例如

```rust
struct Point {x: f64, y: f64, z: f64}

fn main() {
    let a = Point {x: 1.0, y: 2.0, z: 3.0};
    let b = a;
    println!("({}, {}, {})", a.x, a.y, a.z);
    println!("({}, {}, {})", b.x, b.y, b.z);
}
```

可令 rustc 给出以下错误信息：

```
error[E0382]: borrow of moved value: `a`
 --> foo.rs:6:40
  |
4 |     let a = Point {x: 1.0, y: 2.0, z: 3.0};
  |         - move occurs because `a` has type `Point`, which does not implement
```

```
the `Copy` trait
5 |     let b = a;
  |             - value moved here
6 |     println!("({}, {}, {})", a.x, a.y, a.z);
  |                                        ^^^ value borrowed here after move
```

倘若我想让当前的作用域保留一个变量的所有权，但是又允许其他变量或函数使用该变量，该如何实现？基于借用。类似于，你有一本书，我可以借阅，但书依然是你的。Rust 语言提供了 & 用于表示变量的借用，例如

```
struct Point {x: f64, y: f64, z: f64}

fn main() {
    let a = Point {x: 1.0, y: 2.0, z: 3.0};
    let b = &a;
    println!("({}, {}, {})", a.x, a.y, a.z);
    println!("({}, {}, {})", b.x, b.y, b.z);
}
```

上述代码能够完美地通过 rustc 的编译，且程序运行结果符合预期。再例如

```
struct Point {x: f64, y: f64, z: f64}

fn foo(a: &Point) {
    println!("({}, {}, {})", a.x, a.y, a.z);
}

fn main() {
    let a = Point {x: 1.0, y: 2.0, z: 3.0};
    foo(&a);
    println!("({}, {}, {})", a.x, a.y, a.z);
}
```

也是正确的代码。

符号 &，在 Rust 术语里，称为引用。将上述最后一份示例代码「翻译」为 C 语言，或许有助于理解变量的所有权、转移、引用以及借用，例如

```
#include <stdio.h>

typedef struct Point {
    double x;
    double y;
    double z;
```

```
} Point;

void foo(const Point *a) { /* 注意参数 a 的类型 */
    printf("(%f, %f, %f)\n", a->x, a->y, a->z);
}

int main(void) {
    Point a = (Point){.x = 1.0, .y = 2.0, .z = 3.0};
    foo(&a);
    printf("(%f, %f, %f)\n", a.x, a.y, a.z);
    return 0;
}
```

倘若不熟悉 C 语言，可自行忽视这些代码。

现在，回到上一节最后提出的问题，`&mut` 是什么？是可变引用。上述示例 Rust 代码里给出的变量皆是不可变的，因此它们的引用也是不可变的，亦即引用这些变量的变量或函数，它们只能访问自己所引用的变量，却不能修改它们的值。例如，以下代码

```
struct Point {x: f64, y: f64, z: f64}

fn main() {
    let a = Point{x: 1.0, y: 2.0, z: 3.0};
    let b = &a;
    b.x = 3.0;
    println!("({}, {}, {})", a.x, a.y, a.z);
}
```

无法通过 rustc 的编译，出错信息是

```
error[E0594]: cannot assign to `b.x` which is behind a `&` reference
 --> foo.rs:6:5
  |
5 |     let b = &a;
  |             -- help: consider changing this to be a mutable reference: `&mut
a`
6 |     b.x = 3.0;
  |     ^^^^^^^^^ `b` is a `&` reference, so the data it refers to cannot be written
```

之所以出错，是因为 `b` 仅仅是对 `a` 的不可变引用。若想消除该错误，必须令 `a` 具有可变性，且令 `b` 为 `a` 的可变引用，即

```
let mut a = Point{x: 1.0, y: 2.0, z: 3.0};
let b = &mut a;
```

```
b.x = 3.0;
println!("({}, {}, {})", a.x, a.y, a.z);
```

结果是，a 的成员被修改，输出 (3, 2, 3)。又例如

```
struct Point {x: f64, y: f64, z: f64}

fn foo(a: &mut Point) {
    a.x = 3.0;
}

fn main() {
    let mut a = Point {x: 1.0, y: 2.0, z: 3.0};
    foo(&mut a);
    println!("({}, {}, {})", a.x, a.y, a.z);
}
```

与上述 Rust 代码近乎等价的 C 代码是

```
#include <stdio.h>

typedef struct Point {
    double x;
    double y;
    double z;
} Point;

void foo(Point *a) { /* 注意参数 a 的类型 */
    a->x = 3.0;
}

int main(void) {
    Point a = (Point){.x = 1.0, .y = 2.0, .z = 3.0};
    foo(&a);
    printf("(%f, %f, %f)\n", a.x, a.y, a.z);
    return 0;
}
```

弄清楚上述的一切，如何从文件里读取内容，存入一个可变的字符串变量，就很容易了，以下代码

```
use std::path::Path;
use std::fs::File;
```

```rust
use std::io::Read;

fn main() {
    let path = Path::new("data/foo.asc");
    let mut file = match File::open(path) {
        Ok(v) => v,
        Err(e) => panic!("Error: {}", e),
    };
    let mut s = String::new();
    file.read_to_string(&mut s);
    println!("{}", s);
}
```

能够通过 rustc 的编译，但 rustc 会给出以下警告：

```
warning: unused `Result` that must be used
  --> foo.rs:12:5
   |
12 |     file.read_to_string(&mut s);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: `#[warn(unused_must_use)]` on by default
   = note: this `Result` may be an `Err` variant, which should be handled
```

之所以出现这样的警告，是因为 `file.read_to_string` 的返回值也是 `Result` 类型，因此需要用模式匹配，亦即

```rust
match file.read_to_string(&mut s) {
    Ok(_) => println!("{}", s),
    Err(e) => panic!("Error: {}", e)
}
```

变量 `_` 在 Rust 代码里通常表示不关心，亦即不会使用这个变量。

至此，foo.rs 完整的代码如下：

```rust
use std::path::Path;
use std::fs::File;
use std::io::Read;

fn main() {
    let path = Path::new("data/foo.asc");
    let mut file = match File::open(path) {
        Ok(v) => v,
```

```
        Err(e) => panic!("Error: {}", e),
    };
    let mut s = String::new();
    match file.read_to_string(&mut s) {
        Ok(_) => println!("{}", s),
        Err(e) => panic!("Error: {}", e)
    }
}
```

rustc 编译所得程序，能够从自身所在目录的 data 目录内发现 foo.asc 文件，并将其内容读取到字符串变量 s 里，最后输出

```
1.0 2.0 3.0
0.1 0.5 0.7
1.3 4.6 7.1
```

似乎……真是一个了不起的微小成就啊！

2.8 灵异事件

事实上，在上一节的经历里，我遇到了一个灵异事件。以下代码

```
struct Point {x: f64, y: f64, z: f64}

fn main() {
    let mut a = Point {x: 1.0, y: 2.0, z: 3.0};
    let b = &mut a;
    println!("({}, {}, {})", a.x, a.y, a.z);
    println!("({}, {}, {})", b.x, b.y, b.z);
}
```

无法通过 rustc 的编译，错误信息是

```
error[E0502]: cannot borrow `a.x` as immutable because it is also borrowed as mutable
 --> foo.rs:6:30
  |
5 |     let b = &mut a;
  |             ------ mutable borrow occurs here
6 |     println!("({}, {}, {})", a.x, a.y, a.z);
  |                              ^^^ immutable borrow occurs here
7 |     println!("({}, {}, {})", b.x, b.y, b.z);
  |                              --- mutable borrow later used here
```

... ...
... ...

倘若将上述代码修改为

```
struct Point {x: f64, y: f64, z: f64}

fn main() {
    let mut a = Point {x: 1.0, y: 2.0, z: 3.0};
    let b = &mut a;
    println!("({}, {}, {})", b.x, b.y, b.z);
    println!("({}, {}, {})", a.x, a.y, a.z);
}
```

便可完美地通过 rustc 的编译。

事实上，上述示例可进一步简化——出现问题的示例越简单，越有助于分析问题的成因。以下代码

```
1  fn main() {
2      let mut a = 1;
3      let b = &mut a;
4      println!("{}", a);
5      println!("{}", b);
6  }
```

无法通过编译，但是倘若将第 4 行和第 5 行代码交换，便可通过编译。

之所以出现这种灵异现象，是因为 b 借用了 a，但 rustc 并不知道借期，在这种情况下访问 a，便会出问题。这就类似于，我将一本书借给你看，虽然我依然保留此书的所有权，但是在你借阅期间，我是没有这本书的。为何将上述代码的第 4 行和第 5 行交换，便可通过编译呢？ rustc 认为，b 在 println! 语句里被访问了，因此它觉得 b 对 a 的借期已过，亦即，在 rustc 看来，上述代码的第 4 行和第 5 行交换之后，它将其全部代码理解为

```
1  fn main() {
2      let mut a = 1;
3      {
4          let b = &mut a;
5          println!("{}", b);
6      }
7      println!("{}", a);
8  }
```

亦即，rustc 自作主张，为 b 创造了一个作用域，并据此判断 b 对 a 的借期已过。暂时，暂且如此不求甚解吧。

2.9 逐行读取文件

现在，回归正题，即如何基于 foo.asc 文件里的点集数据信息，生成 Pov-Ray 场景文件。仅仅是写出一个能够将 foo.asc 的内容读取至字符串变量，尚不足以解决该问题，需要逐行读取 foo.asc，欲达成此愿，仍需借助标准库提供的工具。

Rust 标准库提供的 `std::io::BufReader` 类型，有一个方法 `lines`，可创造一个可读取文件每行内容的迭代器——不要怕，写这句话的时候的我，也不知道什么是迭代器——，但是要使用这个方法，需要以下 `use` 语句：

```
use std::io::BufRead;
use std::io::BufReader;
```

Rust 允许将前缀相同的类型合并，因此上述代码可以写为

```
use std::io::{BufRead, BufReader};
```

下面是一个有待填充的代码框架：

```
use std::path::Path;
use std::fs::File;
use std::io::{BufRead, BufReader};

fn main() {
    let path = Path::new("data/foo.asc");
    let file = match File::open(path) {
        Ok(v) => v,
        Err(e) => panic!("Error: {}", e)
    };
    let reader = BufReader::new(file);
    for line in reader.lines() {
        ... 待定代码 ...
    }
}
```

`for ... in` 是 Rust 的迭代（或循环）语法。`reader.lines()` 便是迭代器。在它能返回文件的当前行，然后将文件的下一行作为下一次访问的对象。此外，迭代器通常知道如何终止迭代过程。例如，读取文件，当读取到最后一行之后，迭代器便会终止对文件的读取。

试试在迭代过程将 `line` 输出，

```
for line in reader.lines() {
    println!("{}", line);
}
```

rustc 报错：

```
error[E0277]: `Result<String, std::io::Error>` doesn't implement `std::fmt::Display`
  --> foo.rs:13:24
   |
13 |         println!("{}", line);
   |                        ^^^^ `Result<String, std::io::Error>` cannot be formatted
with the default formatter
```

很好……rustc 告诉了我 `line` 的类型是 `Result`，因此可以用模式匹配：

```
for line in reader.lines() {
    let p = match line {
        Ok(v) => v,
        Err(e) => panic!("Error: {}", e)
    };
    println!("{}", p);
}
```

于是，现在便有了一个可以逐行读取文件 foo.asc 的 Rust 程序了，其源文件 foo.rs 的全部内容如下：

```
use std::path::Path;
use std::fs::File;
use std::io::{BufRead, BufReader};

fn main() {
    let path = Path::new("data/foo.asc");
    let file = match File::open(path) {
        Ok(v) => v,
        Err(e) => panic!("Error: {}", e)
    };
    let reader = BufReader::new(file);
    for line in reader.lines() {
        let p = match line {
            Ok(v) => v,
            Err(e) => panic!("Error: {}", e)
        };
        println!("{}", p);
    }
}
```

2.10 字符串分割

虽然 foo.rs 已经能够从 foo.asc 文件里获得每一行信息，但是倘若将其转化为 Pov-Ray 的球体语句，例如

```
sphere {<0.1, 0.2, 0.3>, 0.01 pigment {color Gray50}}
```

尚需将每一行信息进行分割，获得三维坐标分量的字符串表示，亦即，例如将

```
0.1 0.2 0.3
```

分割为 0.1，0.2 和 0.3 三部分。

字符串类型提供了 split_whitespace 方法，可基于空白字符对字符串进行分割，得到一组字段。例如

```
fn main() {
    let a = String::from("0.1 0.2 0.3");
    let split = a.split_whitespace();
    for s in split {
        println!("{}", s);
    }
}
```

变量 split 是 a.split_whitespace 返回的迭代器，可直接将后者放在 for ... in 语句，因此上述代码可简化为

```
fn main() {
    let a = String::from("0.1 0.2 0.3");
    for s in a.split_whitespace() {
        println!("{}", s);
    }
}
```

上述代码使用了 String::from 基于字符串的字面量构造字符串变量，等价于

```
let a = String::new();
a = a + "0.1 0.2 0.3";
```

现在，考虑如何将字符串

```
0.1 0.2 0.3
```

转换为

```
sphere {<0.1, 0.2, 0.3> 0.3 pigment {color 50Gray}}
```

呢？

倘若不引入更多的知识，解决上述问题，唯有对字符串进行两次分割。第一次分割用于确定字段的个数，第二次分割获取字段并加以应用。例如

```
fn main() {
    let a = String::from("0.1 0.2 0.3");
    let mut n = 0;
    for _ in a.split_whitespace() {
        n = n + 1;
    }
    print!("sphere {{<");
    let mut i = 0;
    for s in a.split_whitespace() {
        i = i + 1;
        if i < n {
            print!("{}, ", s);
        } else {
            print!("{}", s);
        }
    }
    println!("> 0.3 pigment {{color 50Gray}}}}");
}
```

上述代码唯一引入的新事物是 `print!`，它与 `println!` 相似，只是后者会在输出内容末尾增加换行符。

2.11 很多点

综合利用上一章以及上述的全部知识，终于能基于 foo.asc 生成 Pov-Ray 场景文件了。为了避免代码过长而令人不适，我分段给出，并略作讲述吧。

首先，

```
use std::path::Path;
use std::fs::File;
use std::io::{BufRead, BufReader};
```

然后定义了函数 `print_prelude`，

```
fn print_prelude() {
    println!("#version 3.7;");
    println!("#include \"colors.inc\"");
    println!("global_settings {{assumed_gamma 1.0}}");
```

```
    println!("background {{color White}}");
}
```

用于输出 Pov-Ray 场景文件里固定不变的内容。

为了便于输出表示任意一个三维点的球体，将字符串两次分割过程以及球体渲染语句封装为一个专门的函数：

```
fn print_sphere(a: &String, b: f64) {
    let mut n = 0;
    for _ in a.split_whitespace() {
        n = n + 1;
    }
    if n != 3 {return}
    print!("sphere {{<");
    let mut i = 0;
    for s in a.split_whitespace() {
        i = i + 1;
        if i < n {
            print!("{}, ", s);
        } else {
            print!("-{}", s); //右手坐标系向左手坐标系的变换
        }
    }
    println!("> {} pigment {{color Gray50}}}}", b);
}
```

需要注意的是，由于点集文件里的数据通常源于采用右手坐标系的设备，而 Pov-Ray 场景为了渲染的方便，采用了左手坐标系，因此需要将前者变换至左手坐标系，变换过程仅仅是将前者的第三个坐标值取相反数。为了实现该变换，上述代码里我只是简单粗暴地为字符串分割结果的最后一个字段增加了负号前缀——这显然不尽合理，但是此刻我尚不知该如何将表示数字的字符串转化为数字类型。

最后是程序的入口函数 `main` 的定义：

```
fn main() {
    let path = Path::new("data/foo.asc");
    let file = match File::open(path) {
        Ok(v) => v,
        Err(e) => panic!("Error: {}", e)
    };
    let reader = BufReader::new(file);
    let r = 1.5;
```

```
    print_prelude();
    for line in reader.lines() {
        let p = match line {
            Ok(v) => v,
            Err(e) => panic!("Error: {}", e)
        };
        print_sphere(&p, r);
    }
    // 光源和相机的参数，需要根据 foo.asc 的点集分布情况精心设定
    println!("light_source {{<100, 50, -200> color White}}");
    println!("camera {{location <55, 35, -150> look_at <55, 35, 0>}}");
}
```

上述代码对 Pov-Ray 场景中的光源和相机参数的设定，是结合具体作为示例的 foo.asc 文件里点集信息进行的。我使用了一个鱼形的点集作为 foo.asc 文件，将其放在 foo.rs 同一目录下的 data 目录。

编译 foo.rs，运行 foo，并使用 povray 将场景渲染为图片的过程如下：

```
$ rust foo.rs ; ./foo > foo.pov ; povray +A foo.pov
```

结果如图 2.1 所示。

2.12 unwrap

上一节给出的 foo.rs 的代码里，有两处类似的代码片段，即

```
let file = match File::open(path) {
    Ok(v) => v,
    Err(e) => panic!("Error: {}", e)
};
```

和

```
let p = match line {
    Ok(v) => v,
    Err(e) => panic!("Error: {}", e)
};
```

事实上，Rust 标准库里有许多函数，返回值类型皆为 `Result` 类型，而针对它们的返回值所作的处理通常具有共性，即倘若返回 `Ok(v)`，就提取 `v`，倘若返回 `Err(e)`，就让程序报错终止，并输出错误信息。为了简化这部分代码，`Result` 类型提供了 `unwrap` 方法，基于该方法，上述两个代码片段可等价地简化为

```
let file = File::open(path).unwrap();
```

和

```
let p = line.unwrap();
```

2.13 小结

文件打开了，不用关闭吗？不用。文件变量所在的作用域结束时，rustc 会基于文件变量自动关闭文件。文件和内存一样，属于程序资源，而 Rust 语言支持资源自动回收。

本章实现的 foo 程序尚有诸多不足，所以它远称不上是 rhamal，更确切地说，连后者的雏形都不是。最大的不足是，foo 程序无法根据读取的点集信息在一定程度上自动确定光源和相机的参数，此外也未能为点集实现真正的坐标变换。

3 点集

点集数据是存放在文本文件里的。上一章仅实现了从文件里读取每一行文本，然后对文本进行分割，获取包含着每个点的坐标分量信息的字段，但这些字段依然是文本，需要将其转化为数字，使得点集数据能够参与数值运算。

3.1 字符串 → 数字

字符串类型有 `parse` 方法，可将字符串解析为指定类型，前提是后者已经实现了相应的转换函数[1]，Rust 标准库为数字类型（整型、浮点型）实现了这样的转换函数。

以下代码可将内容为数字的文本转换为 64 位浮点数：

```
fn main() {
    let a = String::from("3.2");
    let b: f64 = a.parse().unwrap();
    println!("{}", b);
}
```

看到 `unwrap`，就该知道 `a.parse` 的返回值的类型是 `Result`。

配合字符串分割函数，可将形如

```
1.2 2.3 3.4
```

这样的文本转化为

```
struct Point {x: f64, y: f64, z: f64}
```

结构。例如

```
struct Point {x: f64, y: f64, z: f64}

fn main() {
    let mut p = Point{x: 0.0, y: 0.0, z: 0.0};
    let a = String::from("1.2 2.3 3.4");
    let mut i = 0;
    for s in a.split_whitespace() {
        let t: f64 = s.parse().unwrap();
        if i == 0 {
            p.x = t;
```

[1] 确切地说，是 Trait），但是现在我不是很想过早地使用它，不然会让事情变得过于复杂。

```
        } else if i == 1 {
            p.y = t;
        } else if i == 2 {
            p.z = t;
        } else {
            break;
        }
        i = i + 1;
    }
    println!("({}, {}, {})", p.x, p.y, p.z);
}
```

这段代码写得很笨，因为对 Rust 语言过于无知。

3.2 数组和向量

使用数组，能将上一节最后给出的那段代码有所简化。例如，

```
struct Point {x: f64, y: f64, z: f64}

fn main() {
    let mut p = Point{x: 0.0, y: 0.0, z: 0.0};
    let a = String::from("1.2 2.3 3.4");
    let mut coord: [f64; 3] = [0.0; 3];
    let mut i = 0;
    for s in a.split_whitespace() {
        let t: f64 = s.parse().unwrap();
        coord[i] = t;
        i = i + 1;
    }
    p.x = coord[0]; p.y = coord[1]; p.z = coord[2];
    println!("({}, {}, {})", p.x, p.y, p.z);
}
```

其中，

```
let mut coord: [f64; 3] = [0.0; 3];
```

便是数组的构造语句。该数组的各个元素类型为 `f64`，元素个数或数组长度为 3。等号右边是将数组的 3 个元素初始化为0.0。数组里的各个元素可基于从 0 开始的下标访问或赋值，例如上述代码里的

```
coord[i] = t;
```

数组一旦构造，其长度便不可再变。Rust 语言有可变数组——向量。基于向量可将上述基于数组的文本解析代码进一步简化，例如

```
struct Point {x: f64, y: f64, z: f64}

fn main() {
    let mut p = Point{x: 0.0, y: 0.0, z: 0.0};
    let a = String::from("1.2 2.3 3.4");
    let mut coord: Vec<f64> = Vec::new();
    for s in a.split_whitespace() {
        let t: f64 = s.parse().unwrap();
        coord.push(t);
    }
    p.x = coord[0]; p.y = coord[1]; p.z = coord[2];
    println!("({}, {}, {})", p.x, p.y, p.z);
}
```

其中，

```
let mut coord: Vec<f64> = Vec::new();
```

构造了一个空的向量，它不包含任何元素，但是向量元素的类型已经给出，即 `f64`。虽然 `coord` 是空向量，但是可以通过 `coord.push` 方法为其增加元素，例如上述代码里的

```
coord.push(t);
```

向量各个元素，与数组相似，可通过从 0 开始的下标予以访问或赋值。

3.3 String 和 &str

向量可否存储 `String` 类型的元素呢？试试看吧，即使做得不对，rustc 也会给予不错的指导。于是，有以下代码

```
fn main() {
    let a = String::from("1.2 2.3 3.4");
    let mut coord: Vec<String> = Vec::new();
    for s in a.split_whitespace() {
        coord.push(s);
    }
    println!("({}, {}, {})", coord[0], coord[1], coord[2]);
}
```

rustc 给出的错误信息是

```
error[E0308]: mismatched types
 --> foo.rs:7:20
  |
7 |         coord.push(s);
  |                    ^
  |                    |
  |                    expected struct `String`, found `&str`
  |                    help: try using a conversion method: `s.to_string()`

error: aborting due to previous error
```

意思是，s 的类型是 &str，而不是 String，因此代码无法通过编译，而且 rustc 也给出了解决方案，使用 s.to_string 构造 String 类型的值。

可以证实，rustc 给出的方案切实可行。问题是 &str 是什么类型呢？它的功能看上去像 String，譬如它有 parse 方法，但它又不是 String。

&str 是 str 类型的借用形式。str 是什么类型呢？是一种在 Rust 代码里似乎永远也用不到的类型，只能使用它的借用形式。事实上，在 C 语言里也是如此，譬如，下面是 C 语言表示字符串的例子：

```
char *s = "Hello world!";
```

C 语言里有字符串类型吗？有，但也没有，因为它的字符串类型，不过是一个指向字符串常量首地址的 char 类型的指针。Rust 语言里的 &str 颇类似 C 语言里的 char *。这种相似性，在以下 Rust 代码里有所体现，

```
let s = "Hello world!";
```

s 的类型是 &str。

String 类型实际上是向量 Vec<u8>，即每个元素为 1 个字节长度的值。&str 能够用于访问 String 类型的值的某一段元素，因此 &str 也被称为「字符串切片」。例如，

```
let a = String::from("Hello world!");
let b = &a[2..4]; // b 的值为 "ll"
let c = &a[6..];  // c 的值为 "world!"
let d = &a[..3];  // d 的值为 "Hel"
```

理解了上述知识，可写出以下代码：

```
fn main() {
    let a = String::from("1.2 2.3 3.4");
    let mut coord: Vec<&str> = Vec::new();
    for s in a.split_whitespace() {
```

```
        coord.push(s);
    }
    println!("({}, {}, {})", coord[0], coord[1], coord[2]);
}
```

它必能通过 rustc 的编译!

也许，将以下 Rust 代码

```
fn main() {
    let mut a = "Hello world!";
    let b = String::from(a);
    a = &b[6..];
    println!("{}\n{}", b, a);
}
```

翻译为 C 语言代码，更有助于从本质上理解 `String` 和 `&str` 的关系。与上述代码近乎等价的 C 代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
        char *a = "Hello world!";
        char *b = malloc((strlen(a) + 1) * sizeof(char));
        for (char *p = a, *q = b; ; p++, q++) {
                *q = *p;
                if (*p == '\0') break;
        }
        a = b + 6;
        printf("%s\n%s\n", b, a);
        free(b);
        return 0;
}
```

3.4 点集

利用向量 `Vec<Point>`，可将文本文件里存储的点集信息转化为真正的点集——每个点表示为 `Point`，点的每个坐标分量为 `f64`。完成此事，现在已不费吹灰之力：

```
use std::path::Path;
use std::fs::File;
```

```rust
use std::io::{BufRead, BufReader};

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

fn main() {
    let path = Path::new("data/holly.asc");
    let file = File::open(path).unwrap();
    let reader = BufReader::new(file);
    let mut points: Vec<Point> = Vec::new();
    for line in reader.lines() {
        let p = line.unwrap();
        let mut coord: Vec<f64> = Vec::new();
        for s in p.split_whitespace() {
            let t: f64 = s.parse().unwrap();
            coord.push(t);
        }
        if coord.len() == 3 {
            points.push(Point{x: coord[0], y: coord[1], z: coord[2]});
        }
    }
    println!("点数: {}", points.len());
}
```

其中，coord.len 和 points.len 皆是利用了 Vec 的方法 len 获得向量中的元素个数。

上述代码能够通过 rustc 的编译，但是会收到它的以下警告：

```
warning: field is never read: `x`
 --> foo.rs:5:15
  |
5 | struct Point {x: f64, y: f64, z: f64}
  |               ^^^^^^
  |
  = note: `#[warn(dead_code)]` on by default

warning: field is never read: `y`
 --> foo.rs:5:23
  |
5 | struct Point {x: f64, y: f64, z: f64}
  |                       ^^^^^^

warning: field is never read: `z`
 --> foo.rs:5:31
```

```
  |
5 | struct Point {x: f64, y: f64, z: f64}
  |                                ^^^^^^

warning: 3 warnings emitted
```

rustc 认为，这个程序虽然构造了一个点集，但是每个点的坐标并未使用。从下一节开始，考虑如何使用它们。

3.5 Pov-Ray：模型与视图的分离

假设点集信息存储在文本文件 foo.asc 里，Rust 程序 foo 解析了每个点的坐标从而获得了点集之后，为了利用 Pov-Ray 将点集直观地呈现出来，需要将每个点转化为 Pov-Ray 场景里的球体模型，例如

```
sphere {<x, y, z> r pigment {color Gray50}}
```

foo 可将这种形式的文本输出到 Pov-Ray 场景文件里，例如 foo.pov。倘若 foo.asc 存储了大量的三维点信息，则 foo.pov 文件里也要相应存储同样数量的球体模型构造语句。但是，作为 foo.pov 文件里不止需要含有模型，也要有用于构造光源和相机的语句，大量的 `sphere` 语句会导致 foo.pov 可读性极差。

倘若利用 Pov-Ray 支持的场景文件 `#include` 机制，将全部的 `sphere` 语句存放在扩展名为.inc 的文件里，例如 foo.inc，则在.pov 文件了可通过 `#include` 语句将其载入场景。例如：

```
#version 3.7;
#include "colors.inc"
global_settings {assumed_gamma 1.0}
background {color White}

#include "foo.inc"

light_source {<100, 50, -200> color White}
camera {location <55, 35, -150> look_at <55, 35, 0>}
```

类似 foo.inc 这样的文件，可称为模型文件，而 foo.pov 这样的文件可称为视图文件，因为它不需要关心模型具体是什么，仅需要关心光源和相机如何设定。

3.6 写文件

倘若对 Pov-Ray 场景文件进行模型与视图的分离，并且用 Rust 程序输出这样的场景文件，那么就需要 Rust 程序能够生成.inc 和.pov 文件的能力，之前用的程序输出重定向的方法现在没法再用了。

倘若是将一些文本写入文件，可使用 `File::creat` 函数创建一份文件，然后使用 `File` 类型的 `write_all` 方法将文本写入。例如

```
use std::path::Path;
use std::fs::File;
use std::io::Write;

fn main() {
    let path = Path::new("foo.txt");
    let mut file = File::create(path).unwrap();

    file.write_all("Hello\n".as_bytes()).unwrap();
    file.write_all("world\n".as_bytes()).unwrap();
    file.write_all("!\n".as_bytes()).unwrap();
}
```

可以创建 foo.txt 文件，并向其写入以下文本：

```
Hello
world
!
```

由于 `File` 的 `write_all` 方法不能直接将字符串写入文件，因此在上述代码里，需要使用 `&str` 类型的 `as_bytes` 将字符串转化为 UTF-8 字节序列。

现在可以写一个能够生成 foo.inc 的 Rust 程序了，例如：

```
use std::path::Path;
use std::fs::File;
use std::io::{BufRead, BufReader, Write};

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

fn main () {
    let path = Path::new("data/foo.asc");
    let file = File::open(path).unwrap();
    let reader = BufReader::new(file);
    let mut points: Vec<Point> = Vec::new();
    for line in reader.lines() {
        let p = line.unwrap();
        let mut coord: Vec<f64> = Vec::new();
        for s in p.split_whitespace() {
            let t: f64 = s.parse().unwrap();
            coord.push(t);
```

```
        }
        if coord.len() == 3 {
            coord[2] = -coord[2]; // 坐标变换：右手坐标系 -> 左手坐标系
            points.push(Point{x: coord[0], y: coord[1], z: coord[2]});
        }
    }

    let inc_path = Path::new("foo.inc");
    let mut inc_file = File::create(inc_path).unwrap();
    let r = 0.1;
    for p in points.iter() {
        inc_file.write_all("sphere {".as_bytes()).unwrap();
        inc_file.write_all(format!("<{:.3}, {:.3}, {:.3}> {}",
                                   p.x, p.y, p.z, r).as_bytes()).unwrap();
        inc_file.write_all(" pigment {color Gray50}}\n".as_bytes()).unwrap();
    }
}
```

上述代码里，`Vec` 的 `iter` 方法用于构造向量迭代器，`format!` 用于构造格式化的字符串。在 `format!` 语句里，`{:.3}` 表示将浮点类型的格式化参数的小数点后保留 3 位。

现在，有一个问题。上述代码在输出 `sphere` 语句时，将每一个用于表示三维点的小球的半径值设为 0.1，有什么理由吗？没有，这只是很随意的取值，结果会导致，有的时候，Pov-Ray 显示的点（球体）过大了，有的时候又会过小。有没有办法，基于点集自动确定一个不大不小的小球半径？没有。有许多星星比太阳大得多，但是看上去，太阳何其大，星辰何其小。不过，倘若能够计算出点集的包围盒，可将包围盒的尺寸作为参考，从而确定一个较为合理的小球半径。

3.7 包围盒

所谓点集的包围盒，可定义为

```
struct BBox {llc: Point, urc: Point}
```

即两个点。`llc` 和 `urc` 分别是点集所有点的坐标值的最小值和最大值，它们是一个三维盒子的左下角和右上角顶点，基于它们便可确定盒子的长、宽、高以及中心。

函数 `bbox_of_points` 可计算点集的包围盒，其定义如下：

```
fn bbox_of_points(points: &Vec<Point>) -> BBox {
    let mut bbox = BBox{llc: Point{x: f64::INFINITY,
                                   y: f64::INFINITY,
                                   z: f64::INFINITY},
                        urc: Point{x: f64::NEG_INFINITY,
```

```
                                    y: f64::NEG_INFINITY,
                                    z: f64::NEG_INFINITY}};
    for p in points.iter() {
        if bbox.llc.x > p.x {bbox.llc.x = p.x;}
        if bbox.llc.y > p.y {bbox.llc.y = p.y;}
        if bbox.llc.z > p.z {bbox.llc.z = p.z;}
        if bbox.urc.x < p.x {bbox.urc.x = p.x;}
        if bbox.urc.y < p.y {bbox.urc.y = p.y;}
        if bbox.urc.z < p.z {bbox.urc.z = p.z;}
    }
    return bbox;
}
```

`f64::INFINITY` 和 `f64::NEG_INFINITY` 分别表示 `f64` 类型的浮点数的无穷大和负无穷大。

以下代码片段可基于 `bbox_of_points` 的返回值，计算包围盒的尺寸和中心：

```
let bbox = bbox_of_points(&points);
println!("包围盒尺寸 ({:.3}, {:.3}, {:.3})",
         bbox.urc.x - bbox.llc.x,
         bbox.urc.y - bbox.llc.y,
         bbox.urc.z - bbox.llc.z);
println!("包围盒中心 ({:.3}, {:.3}, {:.3})",
         0.5 * (bbox.llc.x + bbox.urc.x),
         0.5 * (bbox.llc.y + bbox.urc.y),
         0.5 * (bbox.llc.z + bbox.urc.z));
```

基于包围盒尺寸，可以大致确定小球半径，例如可取包围盒最长尺寸的 $\frac{1}{2000}$，这意味着在这个尺寸所在的维向上能恰好放置 1000 个点。此外，Pov-Ray 场景里的相机和光源的参数也能基于包围盒的中心和尺寸予以估算。即使这些参数估计结果不尽准确，但是只需在它们的基础上再做调整，便可定义出更好的场景。

3.8 小结

又前进了一小步。

4 外界

迄今为止，所实现的 Rust 程序的任何一个版本，涉及点集文件读取和 Pov-Ray 场景文件输出时，文件路径是在程序里硬性给出的。现在，到了通过与外界交互的方式消除 Rust 程序与特定的文件耦合的时候了。

4.1 命令行参数

点集文件的路径可在程序的命令行参数里给定。例如，

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    println!("{}", args[0]);
    println!("{}", args[1]);
}
```

假设上述代码是 foo.rs 的全部内容，编译 foo.rs，运行所得程序 foo，

```
$ ./foo data/foo.asc
```

可在终端输出

```
./foo
data/foo.asc
```

将下标为 1 的参数作为路径，

```
use std::env;
use std::path::Path;

fn main() {
    let args: Vec<String> = env::args().collect();
    let path = Path::new(args[1].as_str());
    println!("{}", path.display());
}
```

便可以让 Rust 程序打开指定文件了。在上述代码里，`String` 类型的值——以后简称为 `String` 实例，其他类型同——需要通过 `as_str` 方法转化为 `&str` 类型。

4.2 文件名

在终端里，倘若像下面这样执行程序 foo，

```
$ ./foo data/foo.asc
```

基于 data/foo.asc 构造 Path 实例之后，如何从中提取文件名 foo.asc 呢？Path 类型有 file_name 方法，可解决该问题，例如

```
use std::env;
use std::path::Path;

fn main() {
    let args: Vec<String> = env::args().collect();
    let path = Path::new(args[1].as_str());
    let file_name = path.file_name().unwrap();
    println!("{}", file_name.to_str().unwrap());
}
```

4.3 Option

上述代码出现了两次 unwrap，但是与它们相关的函数返回的值的类型并非 Result，而是 Option，后者也可以使用 unwrap 方法，从而避免繁琐的模式匹配。

以下代码

```
let file_name = match path.file_name() {
    Some(v) => v,
    None => panic!("No file name!")
};
```

展现了针对 Option 类型的模式匹配。

Option 类型与 Result 有相似性，但前者可以匹配更多的模式，例如：

```
fn foo(x: Option<&str>) {
    match x {
        Some("Hello") => println!("Hello world!"),
        Some(v) => println!("{} world!", v),
        None => println!("Bad!")
    }
}

fn main() {
```

```
    let a = Some("Hello");
    let b = Some("Hi");
    foo(a);
    foo(b);
}
```

编译上述代码生成的程序执行后，输出

```
Hello world!
Hi world!
```

4.4 不含扩展名的文件名

倘若从路径里提取不含扩展名的文件名，例如从 `data/foo.asc` 路径提取 `foo`，需要使用 `Path` 的 `file_stem` 方法：

```
use std::env;
use std::path::Path;

fn main() {
    let args: Vec<String> = env::args().collect();
    let path = Path::new(args[1].as_str());
    let file_name = path.file_stem().unwrap();
    println!("{}", file_name.to_str().unwrap());
}
```

有了无扩展名的文件名，便可基于它构造同名但不同扩展名的文件了，例如

```
use std::env;
use std::path::Path;

fn main() {
    let args: Vec<String> = env::args().collect();
    let path = Path::new(args[1].as_str());
    let stem = path.file_stem().unwrap().to_str().unwrap();
    let inc = format!("{}.inc", stem);
    let pov = format!("{}.pov", stem);
    let inc_path = Path::new(inc.as_str());
    let pov_path = Path::new(pov.as_str());
    println!("{}, {}", inc_path.display(), pov_path.display());
}
```

4.5 小结

Rust 标准库提供的这些函数和方法，唯一让我不太舒适的是，后缀级别的零碎太多了。虽然我知道它们有助于提高程序的安全性，但是这并非它们存在的理由。

5 rHamal v0.1

现在，只需要对 Pov-Ray 场景文件再多一些了解，便可以创造 rhamal 0.1 版了，它能基于点集文件生成 Pov-Ray 模型文件和视图文件，前者由一组小球表示点集，后者能为点集的渲染提供光源和相机。

5.1 Pov-Ray 场景文件概览

写这份文档之时，Pov-Ray 版本为 3.7，针对该版本，场景文件通常以

```
#version 3.7; // Pov-Ray 版本
#include "colors.inc" // Pov-Ray 预定义的颜色和纹理集
global_settings {assumed_gamma 1.0} // 将颜色空间设置为线性
background {color White} // 场景的背景色默认为黑色，现在设为白色，可省墨
```

作为开头，其中 // 是 Pov-Ray 场景文件的注释符。

接下来，按照 Pov-Ray 场景文件的传统，应该是设置相机、光源，最后给出要渲染的几何模型，例如

```
// 位于 <0, 0, -3> 瞄准 <0, 0, 0> 的相机
camera {location <0, 0, -3> look_at <0, 0, 0>}

// 位于 <3, 3, -3> 的白色光源
light_source {<3, 3, -3> color White}

// 中心为原点，半径为 0.5，颜色为红色的球体
sphere {<0, 0, 0>, 0.5 pigment {color Red}}
```

Pov-Ray 场景的坐标系是左手系，原点位于屏幕中心，x 轴水平指向屏幕右侧，y 轴竖直屏幕上方，z 轴垂直指向屏幕内部。与左手系相反的是右手系，区别仅仅是后者的 z 轴指向与前者相反。rHamal 所处理的点集来自于右手系，因此在构造 Pov-Ray 场景时，需要对其进行坐标变换——对 z 轴坐标取相反数。

rHamal 所处理的点集，点的坐标值并不确定，相机和光源需根据具体的点集的空间分布方能确定，因此 rHamal 生成的场景文件是先定义待渲染的模型，然后设置光源和相机。又由于 rHamal 所生成的几何形体通常是大量的球体，为了让场景文件保持简洁，这些球体保存在.inc 文件，然后在场景文件使用 #include 载入。例如有许多小球的 foo.inc 文件，

```
sphere {<0, 0, 0>, 0.5 pigment {color Red}}
sphere {<0.1, 1.2, 3.1>, 0.5 pigment {color Red}}
sphere {<2.1, 1.7, 0.9>, 0.5 pigment {color Red}}
... ... ...
```

其中每个小球可表示一个点。可渲染这些小球的场景文件 foo.pov 的全部内容为

```
#version 3.7;
#include "colors.inc"
global_settings {assumed_gamma 1.0}
background {color White}

#include "foo.inc"
camera {location <0, 0, -3> look_at <0, 0, 0>}
light_source {<3, 3, -3> color White}
```

rHamal 除了生成.inc 和.pov 文件之外，也需要基于点集的分布，为小球半径、相机和光源的参数给出近似合理的参考值。

从现在开始，将 rHamal 生成的.inc 文件称为模型文件，将.pov 文件称为视图文件。

5.2 复合模型

Pov-Ray 支持一种完备的编程语言，即 SDL（Scene Description Language），为 Pov-Ray 编写场景文件，使用的便是该语言。通常情况下，并不需要太多 SDL 编程知识，但是为了简化 rHamal 输出的模型文件，有必要熟悉 Pov-Ray 的复合模型。

Pov-Ray 在几何建模方面支持 CSG 运算，即以一组简单的三维形体——球体、圆柱、锥体、长方体——通过并（union）、交（intersection）、差（difference）以及合并（merge）等运算，构造更为复杂的三维形体。例如，通过 union 运算，可将许多小球合并为一个复合模型：

```
#version 3.7;
#include "colors.inc"
global_settings {assumed_gamma 1.0}
background {color White}
camera {location <0, 0, -10> look_at <0, 0, 0>}
light_source {<30, 30, -30> color White}

union {
  sphere {<-3, 0, 0>, 1}
  sphere {<0, 0, 0>, 1}
  sphere {<3, 0, 0>, 1}
  pigment {color Red}
}
```

上述场景的渲染结果如图 5.1 所示。

图 5.1　CSG union 运算

虽然将三个球体合并之后渲染与直接渲染三个球体，结果并无不同，但是合并的结果在于构造了一个几何对象。对该对象作整体性的旋转、平移或其他几何变换，而不需要将这些变化逐一施加到构成该对象的基本形体。例如

```
object {
  union {
    sphere {<-3, 0, 0>, 1}
    sphere {<0, 0, 0>, 1}
    sphere {<3, 0, 0>, 1}
    pigment {color Red}
  }
  rotate 30 * z
}
```

可将三个球体的合并结果绕 z 轴逆时针旋转 30 度角，渲染结果如图 5.2 所示。

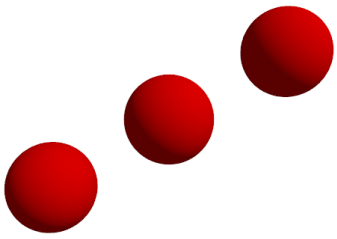

图 5.2　复合对象的旋转变换

在 Pov-Ray 场景文件里使用 `object` 语句有助于将复合模型仅专注于形体的构造，例如上述代码的 `object` 语句可等价修改为

```
object {
  union {
    sphere {<-3, 0, 0>, 1}
    sphere {<0, 0, 0>, 1}
    sphere {<3, 0, 0>, 1}
  }
  pigment {color Red}
  rotate 30 * z
}
```

Pov-Ray 的 SDL 支持变量。可将复合模型定义为变量，然后在 `object` 语句里使用，例如：

```
#declare points = union {
  sphere {<-3, 0, 0>, 1}
  sphere {<0, 0, 0>, 1}
  sphere {<3, 0, 0>, 1}
}

object {
  points
  pigment {color Red}
  rotate 30 * z
}
```

倘若将上述代码里的 `points` 的定义放到模型文件里，例如 foo.inc，然后在视图文件里将其载入，即

```
#include "foo.inc"
object {
  points
  pigment {color Red}
  rotate 30 * z
}
```

这就是 rHamal 0.1 版所要实现的目标的雏形。

5.3 rHamal v0.1 源码

从现在开始，逐步实现 rHamal 0.1 版本。该版本所需要的 Rust 标准库功能如下：

```
use std::env;
use std::path::Path;
```

```
use std::fs::File;
use std::io::{BufRead, BufReader, Write};
```

5.3.1 读取点集文件

点采用结构体类型表示：

```
struct Point {x: f64, y: f64, z: f64}
```

定义一个函数 read_points，用于读取点云文件，返回 Vec<Point> 实例，即

```
fn read_points(path: &Path) -> Vec<Point> {
    let file = File::open(path).unwrap();
    let reader = BufReader::new(file);
    let mut points: Vec<Point> = Vec::new();
    for line in reader.lines() {
        let p = line.unwrap();
        let mut coord: Vec<f64> = Vec::new();
        for s in p.split_whitespace() {
            let t: f64 = s.parse().unwrap();
            coord.push(t);
        }
        if coord.len() == 3 {
            coord[2] = -coord[2]; // 坐标变换: 右手坐标系 -> 左手坐标系
            points.push(Point{x: coord[0], y: coord[1], z: coord[2]});
        }
    }
    return points;
}
```

5.3.2 点集包围盒

点集包围盒表示为以下结构体类型：

```
struct BBox {llc: Point, urc: Point}
```

bbox_of_points 用于计算点集的包围盒：

```
fn bbox_of_points(points: &Vec<Point>) -> BBox {
    let mut bbox = BBox{llc: Point{x: f64::INFINITY,
                                   y: f64::INFINITY,
                                   z: f64::INFINITY},
```

```
                        urc: Point{x: f64::NEG_INFINITY,
                                   y: f64::NEG_INFINITY,
                                   z: f64::NEG_INFINITY}};
    for p in points.iter() {
        if bbox.llc.x > p.x {bbox.llc.x = p.x;}
        if bbox.llc.y > p.y {bbox.llc.y = p.y;}
        if bbox.llc.z > p.z {bbox.llc.z = p.z;}
        if bbox.urc.x < p.x {bbox.urc.x = p.x;}
        if bbox.urc.y < p.y {bbox.urc.y = p.y;}
        if bbox.urc.z < p.z {bbox.urc.z = p.z;}
    }
    return bbox;
}
```

center_of_bbox 用于计算包围盒的中心：

```
fn center_of_bbox(bbox: &BBox) -> Point {
    let center = Point{
        x: 0.5 * (bbox.llc.x + bbox.urc.x),
        y: 0.5 * (bbox.llc.y + bbox.urc.y),
        z: 0.5 * (bbox.llc.z + bbox.urc.z)
    };
    return center;
}
```

5.3.3 输出模型文件

```
fn output_model(path: &Path, points: &Vec<Point>, object_name: &str) {
    let mut file = File::create(path).unwrap();
    file.write_all(format!("#declare {} = union {{\n",
                           object_name).as_bytes()).unwrap();
    for p in points.iter() {
        file.write_all(format!("  sphere {{<{:.3}, {:.3}, {:.3}> point_size}}\n",
                               p.x, p.y, p.z).as_bytes()).unwrap();
    }
    file.write_all("}".as_bytes()).unwrap();
}
```

5.3.4 输出视图

```
fn output_view(path: &Path, bbox: &BBox, object_name: &str) {
    let mut file = File::create(path).unwrap();
    output_prelude(&mut file);
    output_bbox(&mut file, bbox);
    output_object(&mut file, object_name);
    output_camera(&mut file);
    output_light_source(&mut file);
}

fn output_prelude(file: &mut File) {
    file.write_all("#version 3.7;\n".as_bytes()).unwrap();
    file.write_all("#include \"colors.inc\"\n".as_bytes()).unwrap();
    file.write_all("global_settings {assumed_gamma 1.0}\n".as_bytes()).unwrap();
    file.write_all("background {color White}\n\n".as_bytes()).unwrap();
}

fn output_bbox(file: &mut File, bbox: &BBox) {
    file.write_all(format!("#declare bbox_llc = <{:.3}, {:.3}, {:.3}>;\n",
                           bbox.llc.x, bbox.llc.y, bbox.llc.z).as_bytes()).un-
wrap();
    file.write_all(format!("#declare bbox_urc = <{:.3}, {:.3}, {:.3}>;\n",
                           bbox.urc.x, bbox.urc.y, bbox.urc.z).as_bytes()).un-
wrap();
    let center = center_of_bbox(bbox);
    file.write_all(format!("#declare bbox_center = <{:.3}, {:.3}, {:.3}>;\n",
                           center.x, center.y, center.z).as_bytes()).unwrap();
    file.write_all(format!("#declare bbox_len_x = {:.3};\n",
                           bbox.urc.x - bbox.llc.x).as_bytes()).unwrap();
    file.write_all(format!("#declare bbox_len_y = {:.3};\n",
                           bbox.urc.y - bbox.llc.y).as_bytes()).unwrap();
    file.write_all(format!("#declare bbox_len_z = {:.3};\n\n",
                           bbox.urc.z - bbox.llc.z).as_bytes()).unwrap();
}

fn output_object(file: &mut File, object_name: &str) {
    file.write_all("#declare point_size = 0.005 * ".as_bytes()).unwrap();
    let point_size_if = "#if (bbox_len_x > bbox_len_y) bbox_len_x; ";
    let point_size_else = "#else bbox_len_y; #end\n";
    file.write_all(point_size_if.as_bytes()).unwrap();
    file.write_all(point_size_else.as_bytes()).unwrap();
```

```
    let inc = format!("#include \"{}.inc\"\n", object_name);
    file.write_all(inc.as_bytes()).unwrap();

    file.write_all("object {\n".as_bytes()).unwrap();
    file.write_all(format!("  {}\n", object_name).as_bytes()).unwrap();
    file.write_all("  pigment {\n".as_bytes()).unwrap();
    file.write_all("    color Gray50\n".as_bytes()).unwrap();
    file.write_all("  }\n}\n\n".as_bytes()).unwrap();
}

fn output_camera(file: &mut File) {
    file.write_all("#declare how_far = 5 * bbox_len_z;\n".as_bytes()).unwrap();
    file.write_all("camera {\n".as_bytes()).unwrap();
    file.write_all("  location bbox_center - how_far * z\n".as_bytes()).un-
wrap();
    file.write_all("  look_at bbox_center\n}\n".as_bytes()).unwrap();
}

fn output_light_source(file: &mut File) {
    file.write_all("light_source {\n".as_bytes()).unwrap();
    file.write_all("  bbox_center + 5 * how_far * <1, 1, -1>\n".as_bytes()).un-
wrap();
    file.write_all("  color White\n}\n".as_bytes()).unwrap();
}
```

5.3.5 主函数

```
fn main () {
    let args: Vec<String> = env::args().collect();
    let path = Path::new(args[1].as_str());
    let stem = path.file_stem().unwrap().to_str().unwrap();
    let points = read_points(&path);

    let model = format!("{}.inc", stem);
    let model_path = Path::new(model.as_str());
    output_model(&model_path, &points, stem);

    let view = format!("{}.pov", stem);
    let view_path = Path::new(view.as_str());
    let bbox = bbox_of_points(&points);
```

```
    output_view(&view_path, &bbox, stem);
}
```

5.4 编译、安装及用法

假设上一节的所有代码存放在 rhamal.rs 文件，开启 rustc 的二级优化选项编译 rhamal.rs：

```
$ rustc -C opt-level=2 rhamal.rs
```

可将编译所得程序 rhamal 放置在系统路径定义的某个目录内，例如在 Linux 系统，可存放在 /usr/local/bin 目录。

rhamal 的用法如下：

```
$ rhamal 点集文件
```

下面给出 rhamal 的用法示例，所用的点集文件可从

```
https://gitee.com/garfileo/rhamal/raw/master/data/fish.asc
```

下载。

执行以下命令，可在当前目录内生成模型文件 fish.inc 和视图文件 fish.pov 文件：

```
$ rhamal fish.asc
```

fish.inc 文件的内容形如：

```
#declare fish = union {
  sphere {<3.468, 15.881, -4.030> point_size}
  sphere {<3.758, 16.779, -3.993> point_size}
  sphere {<4.310, 16.006, -4.224> point_size}
  ... ... ...
}
```

视图文件 fish.pov 的内容形如：

```
#version 3.7;
#include "colors.inc"
global_settings {assumed_gamma 1.0}
background {color White}

#declare bbox_llc = <3.468, 3.567, -24.538>;
#declare bbox_urc = <119.442, 65.334, -2.996>;
#declare bbox_center = <61.455, 34.450, -13.767>;
```

```
#declare bbox_len_x = 115.975;
#declare bbox_len_y = 61.767;
#declare bbox_len_z = 21.542;

#declare point_size =
  0.005 * #if (bbox_len_x > bbox_len_y) bbox_len_x; #else bbox_len_y; #end
#include "fish.inc"
object {
  fish
  pigment {
    color Gray50
  }
}

#declare how_far = 5 * bbox_len_z;
camera {
  location bbox_center - how_far * z
  look_at bbox_center
}
light_source {
  bbox_center + 5 * how_far * <1, 1, -1>
  color White
}
```

使用 povray 命令，便可在当前目录将 fish.pov 渲染为 fish.png：

```
$ povray +A +P fish.pov
```

其中，`+A` 选项用于打开 Pov-Ray 图形抗锯齿优化功能，`+P` 选项打开渲染结果预览窗口——如图 5.3 所示。

5.5 若求美好，要自己动手

rHamal 已经尽了自己最大的努力，尽管这努力仅仅是 0.1 版，但是无论 rHamal 如何迭变，它也不可能生成更好的视图文件了。倘若追求更好的点集渲染结果，需要在粗通 Pov-Ray SDL 的基础上对视图文件进行一些调整。以下是我常用的一些手段。

以上一节由 rhamal 命令生成的视图文件 fish.pov，若想让模型在默认的位姿绕某个轴向旋转一个角度，例如让 fish 模型以包围盒中心为基准点，绕 z 州方向旋转 90 度，需将 fish.pov 里的 `object` 语句修改为：

```
object {
  fish
```

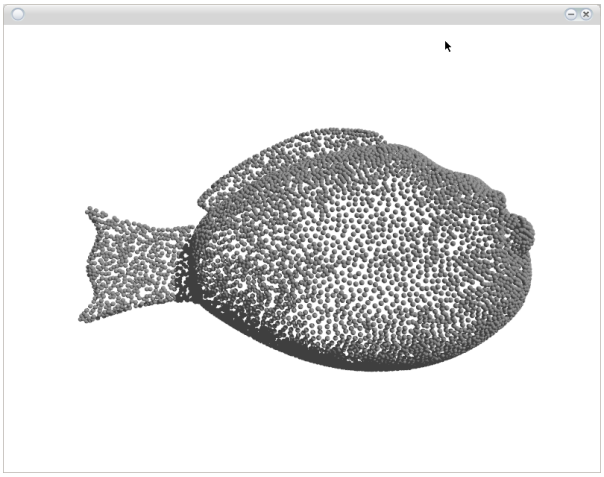

图 5.3　Pov-Ray 渲染结果预览窗口

```
  translate -bbox_center
  rotate 90 * z
  translate bbox_center
  pigment {
    color Gray50
  }
}
```

渲染结果如图 5.4 所示。

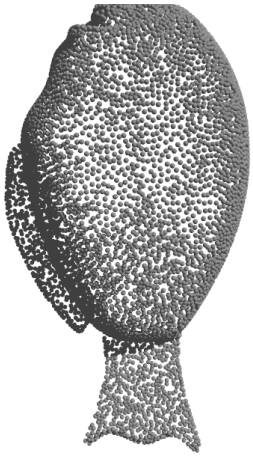

图 5.4　模型旋转结果

显然，图 5.4 所示的旋转结果，图形的顶部有一小部分超出了相机的视野，因而丢失。对于这种情况，需要增令相机距离模型再远一些，因此需要将 fish.pov 文件里的 how_far 的值增大一些，例如：

```
#declare how_far = 6 * bbox_len_z; // 由原来 5 倍的 bbox_len_z 修改为 6 倍
camera {
  location bbox_center - how_far * z
```

```
  look_at bbox_center
}
```

渲染结果如图 5.5 所示。

图 5.5　相机位置修正后的渲染结果

若想让模型的颜色——确切地说，是纹理（Texture）——更夺目，这需要对 Pov-Ray 纹理设计方面的知识有所了解。基于 Pov-Ray SDL 完全能够创造出一些具有艺术性的纹理特效。例如，渲染如图 5.6 所示的热带鱼模型——当然，我可不是说它是艺术品——，

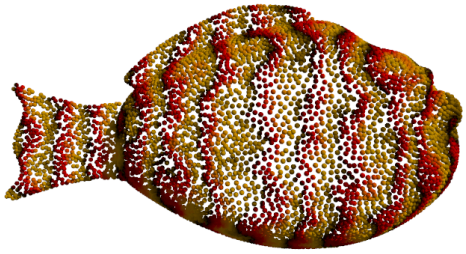

图 5.6　热带鱼

仅需要将 fish.pov 文件里的 `object` 语句做以下修改：

```
object {
  fish
  pigment {
    gradient x
    turbulence .625
    color_map {
      [0.00 color Red]
```

```
      [0.33 color Orange]
      [0.66 color Black]
      [1.00 color Red]
    }
    scale 10
  }
}
```

5.6 小结

至此，完成了 Rust 语言学习的初级阶段。在下一阶段，尝试学习和使用 Rust 语言更高级的语法，对 rHamal 0.1 版本的代码有所改进，令其多少能体现些许优雅。

6 方法

依附于某种类型的函数，称为方法（Method）。到现在为止，已经用过了许多方法，譬如依附于字符串实例的 as_bytes，依附于 Result 或 Option 实例的 unwrap 方法。在 rHamal 的源码里，倘若能为自定义的类型 Point 和 BBox 定义一些方法，也许能有助于提高 rHamal 源码的优雅性。

6.1 静态方法

之前，构造 Point 类型的实例，使用的是结构体实例化语法，例如

```
let p = Point {x: 0.0, y: 0.0, z: 0.0};
```

如果 Point 类型有一个 origin 方法，能够构造原点，那么上述代码可简化为

```
let p = Point::origin();
```

这种方法叫类型的静态方法（Static Method），只能通过类型直接调用，主要用于构造类型实例。

以下代码为 Point 类型定义了四种静态方法：

```
struct Point {x: f64, y: f64, z: f64}

impl Point {
    fn origin() -> Point {
        Point {x: 0.0, y: 0.0, z: 0.0}
    }
    fn inf() -> Point {
        Point{x: f64::INFINITY,
              y: f64::INFINITY,
              z: f64::INFINITY}
    }
    fn neg_inf() -> Point {
        Point{x: f64::NEG_INFINITY,
              y: f64::NEG_INFINITY,
              z: f64::NEG_INFINITY}
    }
    fn new(x: f64, y: f64, z: f64) -> Point {
        Point {x: x, y: y, z: z}
    }
}
```

基于上述方法，能够为 BBox 类型实现两种静态方法：

```
struct BBox {llc: Point, urc: Point}

impl BBox {
    fn universe() -> BBox {
        BBox {llc: Point::inf(), urc: Point::neg_inf()}
    }
    fn new(llc: Point, urc: Point) -> BBox {
        BBox {llc: llc, urc: urc}
    }
}
```

6.2 实例方法

对于 Point 实例，例如

```
let a = Point::new(1.0, 2.0, 3.0);
```

如何能够让它像基本类型那样，能够在 println! 语句里输出呢？例如

```
println!("{}", a);
```

现在，为 Point 类型实现上述的格式化输出的时机尚不成熟，因为需要引入 Rust 的 Trait 机制。但是，倘若为 Point 类型实现 to_string 方法，也能让 Point 实例的格式化输出语句优雅一些：

```
println!("{}", a.to_string());
```

至少比

```
println!("({}, {}, {}", a.x, a.y, a.z);
```

优雅 1/4 吧。

像 a.to_string 这样的方法称为实例方法（Instance Method）。以下代码，为 Point 类型定义了 to_string 方法：

```
impl Point {
    fn to_string(self: &Point) -> String {
        format!("<{}, {}, {}>", self.x, self.y, self.z)
    }
}
```

Rust 语言提供了语法糖，能够让实例方法的参数更为简洁，与上述代码等价的代码如下：

```
impl Point {
    fn to_string(&self) -> String {
        format!("<{}, {}, {}>", self.x, self.y, self.z)
    }
}
```

同理，也能为 BBox 实现 to_string：

```
impl BBox {
    fn to_string(&self) -> String {
        format!("{}, {}",
                self.llc.to_string(),
                self.urc.to_string())
    }
}
```

上述为 Point 和 BBox 实现的静态方法和实例方法有助于为 Pov-Ray 场景文件构造格式化字符串，例如

```
let a = Point::new(1.0, 2.0, 3.0);
let b = Point::new(4.0, 5.0, 6.0);
let c = BBox::new(a, b);
println!("box {{{}}}", c.to_string());
```

可输出

```
box {<1, 2, 3>, <4, 5, 6>}
```

同理，输出球体也容易许多：

```
let a = Point::new(1.0, 2.0, 3.0);
let r = 0.1;
println!("sphere {{}, {}}", a.to_string(), r);
```

可输出

```
sphere {<1, 2, 3>, 0.1}
```

6.3 收纳

在 rHamal v0.1 的源码里，有一些像

```
file.write_all(format!("#declare bbox_llc = <{}, {}, {}>;\n",
                       bbox.llc.x, bbox.llc.y, bbox.llc.z).as_bytes()).unwrap();
```

这样的代码。下面为 Point 类型定义 write 方法，以简化此类代码。

```
impl Point {
    fn write(&self, file: &mut File, prefix: &str, suffix: &str) {
        let s = format!("{}{}{}", prefix, self.to_string(), suffix);
        file.write_all(s.as_bytes()).unwrap();
    }
}
```

基于 Point 类型的 to_file 方法，上述将 bbox_llc 的定义语句写入 Pov-Ray 场景文件的语句可简化为

```
bbox.llc.write(&mut file, "#declare bbox_llc = ", ";\n");
```

于是，那些充斥于 rHamal v0.1 源码里的 as_bytes 和 unwrap 的后缀就得到了很好的收纳。

但是，rHamal v0.1 的源码也有许多像

```
file.write_all("#version 3.7;\n".as_bytes()).unwrap();
file.write_all(format!("#declare {} = union {{\n", object_name).as_bytes()).unwrap();
```

这样的代码。虽然没有办法为 Rust 标准库的 String 或 &str 类型定义 write 方法，但是可以定义一个普通的函数用于简化此类代码，例如

```
fn write_str(file: &mut File, content: &str) {
    file.write_all(content.as_bytes()).unwrap();
}

fn write_string(file: &mut File, content: String) {
    file.write_all(content.as_bytes()).unwrap();
}
```

这两个函数的用法如下：

```
write_str(&mut file, "#version 3.7;\n");
write_string(&mut file, format!("#declare {} = union {{\n", object_name));
```

6.4 字符串续行符

通过 write_str 函数，可将 rHamal v0.1 里的 output_prelude 函数

```
fn output_prelude(file: &mut File) {
    file.write_all("#version 3.7;\n".as_bytes()).unwrap();
```

```
    file.write_all("#include \"colors.inc\"\n".as_bytes()).unwrap();
    file.write_all("global_settings {assumed_gamma 1.0}\n".as_bytes()).unwrap();
    file.write_all("background {color White}\n\n".as_bytes()).unwrap();
}
```

简化为

```
fn output_prelude(file: &mut File) {
    write_str(file, "#version 3.7;\n");
    write_str(file, "#include \"colors.inc\"\n");
    write_str(file, "global_settings {assumed_gamma 1.0}\n");
    write_str(file, "background {color White}\n\n");
}
```

由于 Rust 的字符串语法提供了续行符 \，因此上述的简化结果可进一步简化为

```
fn output_prelude(file: &mut File) {
    write_str(file, "#version 3.7;\n\
                     #include \"colors.inc\"\n\
                     global_settings {assumed_gamma 1.0}\n\
                     background {color White}\n\n");
}
```

6.5 小结

我对 rHamal v0.1 的修改比本章所述内容略多，详见：

https://gitee.com/garfileo/rhamal/blob/7955ab01/rhamal.rs

7 玄之又玄

如果一些类型或一些函数，看上去极为相似，它们的差异也许仅仅在一些类型上，这意味着可将这些类型抽离出来，于是原本有差异的类型或函数便会相同。反过来，如果有一些类型，它们之间存在着很大的差异，倘若希望能够为它们建立相似性，就需要规划一组行为，让这些类型去遵守。

7.1 泛型类型

rHamal v0.1 的源码里，`Point` 类型的定义为

```
struct Point {x: f64, y: f64, z: f64}
```

倘若希望 `Point` 类型的各个坐标分量的类型可变，需要将其定义为泛型类型：

```
struct Point<T> {x: T, y: T, z: T}
```

倘若为泛型类型定义一些方法，需要像下面这样定义：

```
impl <T> Point<T> {
    fn to_string(&self) -> String {
        format!("<{}, {}, {}>", self.x, self.y, self.z)
    }
}
```

不过，上述代码无法通过 rustc 的编译，因为 rustc 认为类型 `T` 未实现 `std::fmt::Display`——用于字符串格式化的 Trait。按照 rustc 的建议，对类型 `T` 给出约束：

```
impl <T: std::fmt::Display> Point<T> {
    fn to_string(&self) -> String {
        format!("<{}, {}, {}>", self.x, self.y, self.z)
    }
}
```

下面是泛型版本的 `Point` 类型完整的示例：

```
use std::fmt::Display;

struct Point<T> {x: T, y: T, z: T}

impl <T: Display> Point<T> {
    fn to_string(&self) -> String {
```

```
        format!("<{}, {}, {}>", self.x, self.y, self.z)
    }
}

fn main() {
    let a = Point::<i32>{x: 3, y: 2, z: 1};
    println!("{}", a.to_string());
}
```

7.2 泛型函数

以下代码定义的两个函数：

```
fn write_str(file: &mut File, content: &str) {
    file.write_all(content.as_bytes()).unwrap();
}

fn write_string(file: &mut File, content: String) {
    file.write_all(content.as_bytes()).unwrap();
}
```

这两个函数非常相似，区别仅仅在于它们的第二个参数的类型，基于 Rust 的泛型语法可将二者统一为：

```
fn write_text<T>(file: &mut File, text: T) {
    file.write_all(text.as_bytes()).unwrap();
}
```

然而，上述代码却无法通过 rustc 的编译，出错信息为

```
error[E0599]: no method named `as_bytes` found for type parameter `T`
in the current scope
 --> foo.rs:6:25
  |
6 |     file.write_all(text.as_bytes()).unwrap();
  |                         ^^^^^^^^ method not found in `T`
  |
  = help: items from traits can only be used if the type parameter is
bounded by the trait
help: the following trait defines an item `as_bytes`, perhaps you need
to restrict type parameter `T` with it:
  |
```

```
5 | fn write_text<T: OsStrExt>(file: &mut File, text: T) {
  |               ^^^^^^^^^^^
```

这一次，rustc 成了狗头军师，它建议的修改方案，即

```
fn write_text<T: OsStrExt>(file: &mut File, text: T) {
    file.write_all(text.as_bytes()).unwrap();
}
```

一度领我进了死胡同，因为，以下调用 `wrote_text` 语句

```
fn main() {
    let path = Path::new("foo.txt");
    let mut file = File::create(path).unwrap();
    write_text::<&str>(&mut file, "Hello");
}
```

会导致 rustc 给出新的错误信息：

```
error[E0277]: the trait bound `&str: OsStrExt` is not satisfied
  --> foo.rs:13:27
   |
6  | fn write_text<T: OsStrExt>(file: &mut File, text: T) {
   |                  -------- required by this bound in `write_text`
...
13 |     write_text(&mut file, "Hello");
   |                           ^^^^^^^ the trait `OsStrExt` is not
implemented for `&str`
```

即，`&` 未实现 `OsStrExt` Trait。

我猜，rustc 发现泛型 `T` 的实例使用了 `as_bytes` 方法，它认为只有实现了 `OsStrExt` Trait 的类型有此方法，然而标准库里的 `&str` 和 `String` 类型也有 `as_bytes` 方法，但是它们的这个方法仅仅是普通的方法，并非某个 Trait 的实现。

Rust 语言虽然在语法层面能够对泛型予以约束，例如 `T: OsStrExt`，但是只有 Trait 可作为约束条件。事实上，这种机制甚为合理。上面我遇到的问题，应当归咎为 Rust 标准库没能认真地对 Trait 予以规划，导致类型的方法和 Trait 的方法名同但实离。

最终我还是解决了 `write_text` 函数的泛型定义，因为我发现有一个 `Into<String>` Trait，可将泛型约束为可转化为 `String` 实例的类型。能通过编译的 `write_text` 函数的定义如下：

```
fn write_text<T: Into<String>>(file: &mut File, text: T) {
    file.write_all(text.into().as_bytes()).unwrap();
}
```

它之所以能通过编译，是因为 rustc 能够判断出代码里使用 as_bytes 方法的实例，其类型一定是 String，因为 Into Trait 的 into 方法返回值的类型是 String。

下面给出完整的示例：

```
use std::path::Path;
use std::fs::File;
use std::io::Write;

fn write_text<T: Into<String>>(file: &mut File, text: T) {
    file.write_all(text.into().as_bytes()).unwrap();
}

fn main() {
    let path = Path::new("foo.txt");
    let mut file = File::create(path).unwrap();
    write_text::<&str>(&mut file, "Hello");
}
```

由于 rustc 能够根据参数的字面量自动推导出它的类型，因此上述代码里的

```
write_text::<&str>(&mut file, "Hello");
```

可简化为

```
write_text(&mut file, "Hello");
```

问题虽然得以解决，但是 write_text 的第 2 个参数，其类型若为 &str，会导致程序的性能有所损失，因为需要将其转换为 String 类型，而此事原本并无必要。

7.3 Trait

不同的类型，但是可以有相似的行为。人要吃东西，睡觉，生孩子。动物不是人，也要吃东西，睡觉，生孩子。道路上，可以走人，也可以行车。在 Rust 语言里，不同的类型拥有的相似的行为，称为 Trait。

下面定义了一个叫作 Text 的 Trait：

```
trait Text {
    fn write(&self, file: &mut File);
}
```

该 Trait 仅由一个方法的声明（或签名）构成，如果某些类型实现了该 Trait，意味着它们皆有方法 write。

以下代码分别为 &str 和 String 类型实现了 Text Trait：

```rust
impl Text for &str {
    fn write(&self, file: &mut File) {
        file.write_all(self.as_bytes()).unwrap();
    }
}

impl Text for String {
    fn write(&self, file: &mut File) {
        file.write_all(self.as_bytes()).unwrap();
    }
}
```

于是，将 `&str` 和 `String` 实例写入文件会简洁甚多，例如

```rust
let path = Path::new("foo.txt");
let mut file = File::create(path).unwrap();
"Hello".write(&mut file);
format!(" world!").write(&mut file);
```

7.4 小结

老子说，玄之又玄，众妙之门。