1. 资源

资源允许你存储某种类型的单个全局实例。使用它来获取你应用程序中全局的数据，例如设置。

要创建一个资源类型，只需要定义一个 Rust 结构体或枚举，并派生 Resource Trait ，类似于组件和事件。

```
#[derive(Resource)]
struct Score(f32);
```

资源每种类型只能有一个实例。如果你需要多个，请使用实体/组件。

Bevy 有许多内置资源，你可以使用这些内置资源来访问引擎的各种功能。它们的工作方式与你自定义的资源完全相同。具体的内置资源，使用到时再说。

2. 访问资源

要在系统中访问资源，请使用 `Res` / `ResMut`。

```
fn my_system(
  // 不可变
  resource_1: Res<ResourceOne>,
  // 可变
  mut resource_2: ResMut<ResourceTwo>,
  // 可能不存在的不可变
  resource_3: Option<Res<ResourceThree>>,
  // 可能不存在的可变
  mut resource_4: Option<ResMut<ResourceFour>>,
) {
  // 游戏逻辑
}
```

3. 管理资源

如果你需要在运行时创建/删除资源，可以使用 `Commands`：

```
fn system_1(mut commands: Commands) {
  // 初始化资源
  commands.init_resource::<ResourceOne>();
  // 插入资源
  commands.insert_resource(ResourceTwo);
  // 删除资源
  commands.remove_resource::<ResourceThree>();
}
```

在 App 上初始化和插入资源，这样做可以保证资源一开始就存在。

```
App::new()
  // 初始化资源
  .init_resource::<ResourceOne>()
  // 插入资源
  .insert_resource(ResourceTwo)
  .run()
```

4. 资源初始化

实现了 `FromWorld` Trait 的资源可以被初始化，实现了 `Default` Trait 的资源将自动实现 `FromWorld` Trait。

```
#[derive(Resource, Default)]
struct ResourceOne {
  field_1: f32,
  field_2: u32,
}

#[derive(Resource)]
struct ResourceTwo {
  field_1: f32,
  field_2: i32,
}

impl FromWorld for ResourceTwo {
  fn from_world(world: &mut World) -> Self {
    Self {
      field_1: 0.,
      field_2: 0,
    }
  }
}
```