三年后，我们退出了 Rust 游戏开发

2024-05-01 MicroBlock #Rust

这是一篇翻译自 Loglog Games 的 Leaving Rust gamedev after 3 years 的文章，由 MicroBlock, H4M5TER, Claude 翻译，并已获得转载许可。
Loglog Games 开发很不错的游戏，可以去看看~
若有翻译问题 请在 我的 Telegram 群组 联系我

免责声明：本文是对多年来我遇到的思考和问题的长篇集合，同时也回应了我一再听到的一些论点。本文表达的观点来自我使用 Rust 游戏开发多年来数千小时的 Rust 开发经验和多个完成的游戏。这并不是为了炫耀或表明成功，而只是展示了在 Rust 上付出了足够的努力，以反驳常说的"一旦你获得足够的经验，一切都会明了"的论点。

本文不是一项科学评估或 A/B 研究。这是我个人的观点，我们尝试使 Rust 游戏开发适用于我们这样的小型独立开发团队（2人），并试图通过它来筹集足够的资金来支持我们的开发。我们不是那些有无限资金和多年时间来烧掉的开发者。如果你属于这个类别，并且愿意花几年时间构建系统，下面的内容都不适用。我从"我希望在最多 3-12 个月内制作一款游戏并发布，以便人们可以玩它，并从中赚一些钱"的角度来看待这些问题。这并不是从"我想学习 Rust，游戏开发似乎很有趣"的角度来写的，虽然这是一个有效的目标，但与我们的目标完全不一致，我们希望以商业可行和自给自足的方式进行游戏开发。

我们已经发布了几款游戏，使用过的技术包括 Rust、Godot、Unity 和 Unreal Engine，许多人在 Steam 上玩过它们。我们还制作了我们自己的 2D 游戏引擎，使用从头编写的简单渲染引擎，并在多年的项目中使用了 Bevy 和 Macroquad，其中一些项目非常复杂。我还全职在工作中使用 Rust 作为后端开发人员。本文不是刚刚通过一些教程或在 Game Jam 上制作小游戏之后产生的短浅观点。我们已经编写了超过 10 万行 Rust 代码，历时 3 年以上。

本文的目标是作为一个参考，反驳一再重复的常见论点。但再次强调，这是一种主观观点，很大程度上是为了我不必一遍又一遍地解释同样的事情。当别人问起时，我希望这可以成为一个参考，解释我们为什么希望放弃使用 Rust 开发游戏。我们不会停止游戏开发，只是停止在 Rust 中进行游戏开发。

如果你的目标是学习 Rust,因为它看起来很有趣,你喜欢技术挑战,那完全没问题。但我想通过这篇文章反思的一部分是，人们经常在不知道对方是在构建技术演示还是试图交付某些东西的情况下，建议使用 Rust 开发游戏。整个社区压倒性地专注于技术,以至于游戏开发中的"游戏"部分成了次要的。举个例子,我记得有一次关于 Rust Gamedev Meetup 的讨论,虽然可能是开玩笑的，但我认为仍然说明了这个问题，大致是"有人想在聚会上展示一个游戏,可以吗?".....我不是想说人们应该和我们有相同的目标,但我认为也许有些事情的传达方式可以更清晰,人们应该更诚实地面对他们正在做的事情。

一旦你精通Rust，所有这些问题都会消失

学习Rust是一种有趣的体验，一开始很多事情似乎是"只有我遇到的特殊问题"，随后人们意识到存在一些普遍的基本模式，每个学习者都必须重新发现和内化某些问题，以提高生产力。这可能包括简单的事情，比如 `&str` 和 `String` 之间的区别，`.iter()` 和 `.into_iter()` 的使用，以及对某些抽象的部分借用常常会产生冲突的认识。

这些问题中的许多只是学习过程中的阵痛,一旦积累了足够的经验,用户就可以不加思索地完全预见到这些问题，并保持高效。我非常享受用 Rust 编写各种实用程序和命令行工具的时光，在仅仅几行代码内，它便比 Python 更高效。

话虽如此，Rust 社区有一种压倒性的力量：当有人提到他们在基本层面上遇到了 *Rust 语言* 的问题时，答案就是"你只是还没理解，相信我，一旦你变得足够熟练，一切都会变得清晰明了"。这不仅仅是在Rust中如此，如果你尝试使用ECS，你会得到同样的回答。如果你尝试使用Bevy，你也会得到同样的回答。如果你尝试使用任何框架来制作GUI（无论是一种响应式解决方案还是即时模式），你也会得到同样的回答。*你遇到的问题之所以是问题，只是因为你的努力还不够。*

我多年来一直相信这一点。我尝试了很多年，非常努力。我确实在语言的许多方面看到了这种情况，并发现自己在某些领域非常高效。我学会了如何避免语言和类型系统的某些问题。

然而我能这么说是因为，我花了过去大约3年的时间，在整个框架/引擎生态系统中编写了超过 10 万行与游戏相关的代码，并制作了自己的游戏引擎，但是许多问题（如果一个人不愿意不断重构代码并将编程视为解决难题的过程，而不仅仅是完成任务的工具）并没有消失。

最根本的问题是借用检查器在最不方便的时候_强制_进行重构。Rust用户认为这是件好事，因为它使他们"编写优秀的代码"，但是我花在这门语言上的时间越多，我越怀疑这句话的真实性。优秀的代码是通过不断迭代一个想法并尝

试各种方法来编写的，虽然借用检查器可以强制进行更多迭代，但并不意味着这是一种理想的编码方式。我经常发现，无法暂时地_继续前进_并解决问题，然后稍后再修复它，真正妨碍了我编写优秀代码。

在其他语言中，人们可以以"我以后可以抛弃这个"为前提来编写代码，我发现这是在获得优秀代码方面最有用的方法。比如，我正在实现一个玩家控制器。我只想让玩家移动和执行操作，以便我可以开始构建我的关卡和敌人。我不需要一个优秀的控制器，我只需要它能工作。我可以删除它然后稍后再制作一个更好的。在 Rust 中，有时候只是做一件事是不可能的，因为你可能需要的东西在你正在做事情的地方不可用，编译器最终会_迫使_你进行重构，即使你知道代码基本上是可以丢弃的。

Rust 在大规模重构方面的优势解决了借用检查器造成的大部分问题

人们经常说 Rust 最大的优点之一是易于重构。这绝对是真实的，我曾多次有这样的经历，可以毫不担心地重构代码库的重要部分，之后一切都能正常工作。是这样吧?

问题是，Rust 是一门相对于其他语言而言，更经常迫使用户进行重构的语言。只需要稍微写一点点东西，你就可能陷入和借用检查器打架的困境，意识到"等等，我不能添加这个新东西，因为会出现编译错误。我除了重构代码，没有其他解决办法"。

在这种情况下，有经验的人通常会说，一旦你更熟悉 Rust，这个问题就不再是问题了。我的观点是，虽然这是完全正确的，但是游戏是复杂的状态机，并且需求经常变化，这是一个根本性的问题。用 Rust 编写 CLI 或服务器与编写独立游戏完全不同。目标是为玩家构建良好的体验，而不是预先设计好的通用系统，需求可能会在有人玩游戏后的每一天都发生根本性的变化。Rust 极其静态和过度检查的特性直接与此冲突。

许多人可能会反驳说，在你与借用检查器作斗争并不得不重构代码之后，你的代码变得更好了，这实际上是件好事。我认为，当你知道自己在构建的东西是什么的时候，这是一个有效的观点。但在大多数情况下，我不想要"更好的代码"，我想要"更快地制作游戏"，以便我可以尽快测试它，检查我的点子是否有趣。我经常不得不在"我是应该打破思路并花2个小时重构代码以测试一个想法，还是让代码库变得更糟？"之间做出选择。

对于独立游戏来说，*可维护性不是一个正确的价值观*，因为我们应该追求的是迭代速度。其他语言可以更容易地解决突然出现的问题，而不必牺牲代码质量。而在Rust中，总是需要在_给这个函数添加第11个参数，或者添加另一个`Lazy<AtomicRefCell<T>>`，或者将其放入另一个全能对象，或者添加间接性并恶化迭代体验，或者花时间重新设计代码的这一部分_之间做出选择。

间接引用只解决了一些问题，却以DX为代价

Rust非常喜欢并经常使用的一种基本解决方案是添加间接引用层。一个经典的例子是 Bevy 的事件系统，它是"我的系统需要17个参数来完成任务"相关问题的首选解决方案。我尝试过解决这个问题的两个方案，甚至是在使用Bevy时: 尝试更多地使用事件，或者将所有内容放在一个单一的系统中。话虽如此，这只是一个例子。

使用间接方式可以简单地解决许多借用检查器的问题。或是通过复制/移动某些内容，然后执行操作，再将其移动回来。或是将其存储在命令缓冲区中，稍后再执行。这通常会导致在设计模式方面发现有趣的内容。例如，我发现非常有趣的一点是，通过预先保留实体ID（例如hecs中的World::reserve方法，注意是`&world`而不是`&mut world`），再结合命令缓冲区，可以解决很大一部分问题。这些模式在起作用时非常出色，它们解决了其他非常困难的问题。另一个例子是看似非常特殊的`thunderdome`中get2_mut方法，起初可能看起来像是一个随意的想法，直到人们意识到这是一个经常出现且解决了许多意外问题的方法。

我不打算争论学习曲线是否合理。它当然不合理，但本文整体讨论的是即使在积累足够经验之后，这些问题仍然存在于开发的基本层面。

回到重点，虽然上述某些方法可以解决特定问题，但通常会有一些无法用专门设计和深思熟虑的库函数解决的情况。这就是许多人会建议使用命令缓冲区或事件队列来解决"稍后再执行有问题的操作"的情况，而且这确实有效。

游戏特别关注事件之间的相互关联、特定的时间安排以及同时管理大量状态。在事件之间移动数据意味着代码逻辑会突然分成两部分，即使业务逻辑可能是"一个块"，但在认知上仍然需要将其视为两个块。

在社区中待得足够久的人都会被告知 *这实际上是一件好事*，关注点分离、代码更"清晰"等等。你看，Rust 的设计很聪明，如果某些事情无法做到，那是因为你的设计有问题，它只是想强迫你走上正确的道路...对吗?

有时候，在 C# 中可能只需要 3 行代码的事情，突然变成了分散在两个地方的 30 行 Rust 代码。最经典的例子是像这样的："当我遍历这个查询时，我想要检查另一件东西上的一个组件，并触发一系列相关的系统"（生成粒子、播放音频等）。我已经能听到有人告诉我："嗯，这显然是一个 `Event`，你不应该将这段代码写成内联的形式。"

想象一下，如果我想要做类似的事情（以Unity代码为例，请做好准备，或者假装它是Godot）：

```
if (Physics.Raycast(..., out RayHit hit, ...)) {
  if (hit.TryGetComponent(out Mob mob)) {
    Instantiate(HitPrefab, (mob.transform.position + hit.point) / 2).GetComponent<AudioSource>().clip = mob.HitSoun
```

```
    }
}
```

这是一个相对简单的例子，但确实是我们可能想要编写的代码。尤其是在实现新机制和测试时，你可以直接这么写。无需考虑“可维护性”，我只想做一些非常简单的事情，并且希望在它们应该发生的地方完成。我不想要一个 `MobHitEvent`，因为我可能还有其他5个事物需要检查光线投射。

我也不想检查 "`Mob` 上是否有 `Transform`"。当然有一个，我正在制作游戏。我所有的实体都有一个变换组件。但 Rust 不允许我直接访问 `.transform`，更不用说以一种即使我不小心在有重叠原型的查询中使用，也绝不会导致双重借用错误崩溃的方式了。

我可能也不想检查音频源是否存在。当然，我可以使用 `.unwrap().unwrap()`，但更加细心的🦀们会注意到 `world` 没有被传递，我们是默认使用全局世界吗？我们在使用依赖注入将查询作为系统的另一个参数写出来并提前布局吗？`.Choose` 是否假设了一个全局随机数生成器？线程呢??? 而物理世界具体在哪里，我们真的假设它也是全局的吗？

如果你在想“这样没法规模化”或者“它可能以后会崩溃”或者“有XYZ的原因你不能假设全局世界”或者“如果是多人游戏怎么办”或者“这就是糟糕的代码”.....我明白你的意思。但是在你向我解释我错了的时候，我已经完成了我的功能实现并继续前进了。我一次性编写代码而不考虑代码本身，我在编写代码时思考的是我正在实现的游戏功能以及它对玩家的影响。我没有考虑“在这里获取随机生成器的正确方式是什么”或者“我可以假设这是单线程的吗”或者“我是否在嵌套查询中，如果我的原型重叠会怎么样”，而且我之后也没有遇到编译器错误，也没有运行时借用检查器崩溃。我在愚蠢的引擎中使用愚蠢的语言编写代码，在编写代码的整个过程中一直思考的是游戏本身。

ECS解决了错误类型的问题

由于Rust的类型系统和借用检查器的工作方式，ECS成为了解决“如何使东西引用其他东西”的问题的一种自然出现的解决方案。不幸的是，我认为术语混淆相当严重，不仅不同的人意味着不同的事情，而且社区的大部分人将一些并非真正是ECS的东西归因于ECS。让我们试着将这些事情分开来。

首先，这里是一些由于各种原因我们实际上无法做到的事情（由于这篇文章已经太长，这里有一些细微的差别，但简化起见）：

- 使用实际指针的指针型数据。问题很简单，如果角色 A 跟随 B，而 B 被删除（并且被释放），指针将无效。
- 结合弱指针的 `Rc<RefCell<T>>`。虽然这可能有效，但在游戏中性能很重要，由于内存局部性的原因，这些的开销是不可忽视的。
- 实体数组的索引。在第一种情况下，我们将有一个无效的指针，在这种情况下，如果我们有一个索引并且删除一个元素，索引可能仍然有效，但会指向其他内容。

现在出现了一个神奇的解决方案，摆脱了所有这些问题，即 Generational Arenas Allocator，就像由`thunderdome`所展示的。顺便说一句，这是一个我强烈推荐的库，因为它小巧轻便，做到了它应该做的事情，同时保持了代码的可读性，这在Rust生态系统中相当罕见。

Generational Arenas Allocator 基本上只是一个数组，只不过我们的键不是索引，而是一个包含 `(索引, 时代)` 的元组。数组本身存储了 `(时代, 值)` 的元组，为了简单起见，我们可以假设每次在某个索引处删除东西时，我们只需增加该索引处的 `时代` 计数器。然后我们只需要确保对竞技场的索引总是检查提供的索引的时代是否与数组中的时代匹配。如果该项已被删除，则对应索引将具有较高的时代，并且该索引将被视为“无效”，就好像该项不存在一样。还有一些其他相当简单的问题需要解决，例如保持一个空闲插槽列表，以使插入更快，但这都不是用户需要担心的。

关键点在于，这使得像Rust这样的语言完全可以绕过借用检查器，允许我们进行“使用 arenas 进行手动内存管理”，而不实际触及任何复杂的指针，同时保证 0% unsafe。如果我喜欢 Rust 哪个地方，我会提到这个。特别是使用像`thunderdome`这样的库，它，和它的数据结构，都非常符合这门语言的初衷。

现在到了有趣的部分。大多数人认为 ECS 带来的的好处实际上大部分是 Generational Arenas Allocator 带来的。当人们说 “ECS 让我拥有很好的内存局部性”，但他们只是像 `Query<Mob, Transform, Health, Weapon>` 这样查询怪物时，他们实际上做的基本上相当于`Arena<Mob>`，其中结构定义如下：

```
struct Mob {
  typ: MobType,
  transform: Transform,
  health: Health,
  weapon: Weapon
}
```

当然，以这种方式定义东西并没有ECS的所有好处，但我觉得应该非常明确地指出，仅仅因为我们在使用 Rust，仅仅因为我们不想让一切都成为 `Rc<RefCell<T>>` 并不意味着我们需要 ECS，实际上我们真正想要的可能只是一个 Generational Arenas Allocator。

回到ECS，有几种非常不同的看待ECS的方式：

ECS作为动态组合，允许组件的组合被存储、查询和修改，而无需被绑定在单个类型中。一个明显的例子是在 Rust 中许多人最终会使用的方式（因为没有其他好的方法）是给实体打上"状态"组件的标签。一个例子可能是我们想查询所有的 `Mob`，但是可能其中一些已经变形成了不同的类型。我们可以简单地执行 `world.insert(entity, MorphedMob)`，然后在查询中可以查询 `(Mob, MorphedMob)`，或者类似 `(Mob, Not<MorphedMob>)` 或 `(Mob, Option<MorphedMob>)`，或者在代码中检查该组件的存在。根据不同的ECS实现，这些可能会有不同的结果，但实际上我们使用这种方式来"标记"或"分离"实体。

组合可以比这个更丰富。前面的例子也适用于这种情况，我们可以将其作为单独的 `Transform`、`Health`、`Weapon` 等组件，而不是一个大的 `Mob` 结构体。一个没有武器的怪物没有 `Weapon` 组件，一旦它拾起武器，我们就将其插入到实体中。这将允许我们在一个单独的系统中遍历所有带有武器的怪物。

我会把 Unity 的 "EC" 方法包括在动态组合中，尽管它可能不是最纯粹的传统 "带系统的ECS"，但它在很大程度上使用组件进行组合，抛开性能问题不谈，它最终允许的功能与"纯 ECS" 非常相似。我还要特别提到 Godot 的节点系统，其中子节点经常被用作"组件"，虽然这与 ECS 无关,但它与"动态组合"有着千丝万缕的联系，因为它允许在运行时插入/删除节点，并因此改变实体的行为。

还应该注意到，"将组件拆分为尽可能小以实现最大复用"经常被视为一种好的做法。我参与过无数次争论，有人试图说服我，如果我不将`Position`和`Health`从我的对象中分离出来，我的代码就是一团乱麻。

在多次尝试了这些方法之后，我现在完全不同意这种观点，除了在关心性能的时候。所以，我只会在这个性能问题对这些实体很重要的情况下才会让步。在尝试"分离"的方法之前和之后，我也尝试过另一种"胖组件"方法，我觉得"胖组件"的方法更适合具有场景限定逻辑的 *游戏*。例如，将 `Health` 建模为通用机制可能在简单的模拟中有用，但在每个游戏中，我最终都希望玩家的生命和敌人的生命具有非常不同的逻辑。我也经常希望不同类型的非玩家实体有不同的逻辑，例如墙的生命和怪物的生命。如果有什么不同，我发现试图将其归纳为"相同的生命值"导致我的生命系统中充斥着`if player { ... } else if wall { ... }`这样的不清晰代码，而不是将它们作为大而全的玩家或墙壁系统的一部分。

ECS 作为数组的动态结构，由于组件在 ECS 中的存储方式，我们可以获得迭代"Health"组件并将它们在内存中彼此相邻的好处。对于外行来说，这意味着我们不再使用 `Arena<Mob>`，而是使用：

```
struct Mobs {
  typs: Arena<MobType>,
  transforms: Arena<Transform>,
  healths: Arena<Health>,
  weapons: Arena<Weapon>,
}
```

而在同一索引处的值将属于同一个"实体"。手动完成这个过程是很麻烦的，根据您的背景和之前使用过的语言，您可能手动进行过这个过程。但是由于现代ECS 的出现，我们只需将类型写成一个元组的形式，然后底层存储机制将正确的组件自动放在一起。

我还将这种用法称为 **ECS 作为性能**，因为这种做法的目的不是"因为我们想要组合"，而是"因为我们想要更好的内存局部性"。这可能确实有一些有用的地方，但我认为对于绝大多数实际发布的独立游戏来说，这是不必要的。我有意地说"实际发布的"，是因为当然很容易构建出需要这种方式的令人惊叹的复杂原型，但这些原型的复杂度使它们离被其他人"玩"还有无限的距离，对于本文来说不是关注的重点。

ECS 作为 Rust 借用检查器的解决方案，这是我认为大多数使用ECS的人实际上正在做的事情，或者说他们使用 ECS 的原因。ECS 是 Rust 中非常流行的解决方案和建议，因为它往往可以解决很多问题。如果我们只是传递 `struct Entity(u32, u32)` 这样的东西，我们就不需要关心它们的生命周期，这一切都是拷贝的，很好，就像 Rust 喜欢的那样。

我之所以将这作为一个单独的观点，是因为很多时候人们使用ECS是因为它解决了"我应该把对象放在哪里"的特定问题，而不是真正使用它进行组合，也不需要其性能。这没有什么问题，只是当这些人最终在整个互联网上争论，试图说服其他人他们的做事方式是错误的，并且他们应该出于上述原因以某种方式使用ECS时，事情就变得有些不对劲了，因为实际上他们根本不需要它。

以动态创建的生成型竞技场（ECS），这是我希望存在并尝试过合并的东西，只是意识到要真正获得我想要的东西，我必须重新发明许多我一开始想要避免的丑陋的内部可变性相关的东西，只是为了允许同时做`storage.get_mut::<Player>()`和`storage.get_mut::<Mob>()`这样的事情。Rust在按照它预期的方式工作时，一切都很有趣和美好，但一旦你想要一些它不太喜欢的东西，事情很快会变成"我需要重新实现自己的`RefCell`来做这个特定的事情"或者更糟。

我所指的是，虽然生成型竞技场很好，但其中一个令人讨厌的缺点是，每个要使用的竞技场都必须定义一个变量和类型。当然，如果只在每个查询中使用一个组件，ECS可以解决这个问题，但如果不需要一个完整的原型ECS，只是想按需获取每个类型的竞技场，那将是非常好的。当然有办法做到这一点，但是我已经对尝试重新发明生态系统的部分已经过度疲劳，也不再关心是否强迫自己去做了。

之所以选择ECS是因为Bevy，这部分是一个玩笑，但我认为由于Bevy的流行和它的全面性，应该将其作为ECS的另一种观点提及。对于大多数引擎/框架来说，ECS是一种选择，是人们决定使用的库。但对于Bevy游戏而言，这不是一

种可选项，你的整个游戏都将是基于ECS的。

必须以最积极的方式指出，尽管我可能对许多事情持有不同意见，但很难否认Bevy对ECS API和ECS本身人机工程学的改进有多大。任何看过或甚至使用过像`specs`这样的东西的人都明白Bevy在使ECS易于使用和接近方面取得了多大的改进，并且多年来取得了多大的进步。

话虽如此，我认为这也是我对Rust生态系统如何看待ECS的问题的核心原因，尤其是Bevy。ECS是一种工具，一种解决特定问题的非常特定的工具，并且这并非是免费的。

我想在这里稍微转个弯，谈谈Unity。无论其许可证、领导层还是商业模式发生了什么，认为Unity不是使独立游戏开发取得成功的主要因素是愚蠢的。看看SteamDB统计数据，现在有近44,000个Unity游戏在Steam上，第二名是Unreal引擎的12,000个，其他引擎远远落后。

任何关注Unity的人都知道Unity DOTS，它本质上就是他们的“ECS”（以及其他数据导向的东西）。作为Unity的过去、现在和未来的用户，我对此非常兴奋，我发现它令人兴奋的主要原因之一是它与现有的游戏对象方法并存。虽然有许多复杂之处，但其核心是可以预期的。一个游戏可以在某些方面使用DOTS，同时仍然使用标准的游戏对象场景树，这两者可以很好地结合使用。

我认为在Unity领域不会找到一个了解DOTS的人认为它是一个不应该存在的糟糕功能。但我也认为没有人认为DOTS就是未来的全部，认为游戏对象应该从存在中抹去，将Unity的所有内容都转移到DOTS上。即使忽略维护和向后兼容性，这也是非常愚蠢的，因为有许多工作流程自然适合于游戏对象。

那些使用Godot的人可能会看到类似的观点，尤其是那些使用`gdnative`（例如通过`godot-rust`）的人，其中虽然节点树可能不是适用于所有情况的最佳数据结构，但对于许多事情来说确实非常方便。

将这一点回到Bevy，我认为许多人没有意识到“ECS一切”的方法有多么糟糕。一个明显的例子，也是我认为是一个巨大的失败点的例子，是Bevy的UI系统，这已经成为一个长期的痛点，尤其是与“我们今年肯定会开始开发编辑器！”这类承诺相结合。如果你看一下Bevy的UI示例库，很快就会明显地发现那里没有太多东西，而且查看一下简单的在鼠标悬停和点击时更改颜色的按钮的源代码很快就会揭示原因。我实际上尝试过使用Bevy UI来做一些非平凡的事情，我可以确认，与看起来一样，由ECS完成任何与UI相关的事情所需的仪式感就是完全疯狂的。结果，Bevy最接近编辑器的东西就是一个使用`egui`的第三方crate。我有点简化了这件事，当然，制作编辑器需要更多的东西，不仅仅是UI，但我确实认为将一切都放入ECS中，包括UI，绝对没有帮助。

Rust中的ECS有这种趋势，从其他语言中被视为工具的东西变成几乎是一种宗教信仰。应该使用它，因为它是纯粹和正确的，因为这是正确的方式。

编程语言社区通常有一些倾向性，多年来我一直是一个连续的语言跳槽者，我发现这很有趣。我能想到与Rust对ECS的看法最接近的是Haskell。虽然很暴论，但是我还是要说，我觉得Haskell的整体社区更加成熟，人们普遍对其他方法的存在更加理性，并将Haskell视为“解决问题的有趣工具”。

另一方面，Rust往往感觉就像是与青少年谈论他们对任何事情的偏好一样。他们表达出来的通常是非常强烈的观点，没有太多细微差别。编程是一种非常微妙的活动，在其中人们经常不得不做出次优选择以及及时完成工作。Rust生态系统中的完美主义和对“正确方式”的迷恋常常让我觉得这门语言吸引了那些对编程较新的、容易受到影响的人。再次强调，我明白这并不适用于每个人，但我认为对ECS的整体痴迷在某种程度上是这种情况的产物。

广义系统并不带来有趣的游戏玩法

解决许多问题的一个常见方案是通过系统进行更多的泛化。如果只是将组件更细分，并使用适当的系统，那么所有那些特例问题肯定会被避免，对吧？ 这是个很强有力的论点，除了“通用解决方案会导致无聊的游戏玩法”之外，很难反驳。作为Rust游戏开发社区的活跃成员，我见过很多其他人正在构建的项目，当然，他们提供的建议通常与他们正在开发的游戏相关。那些倾向于拥有完全通用的精心设计的系统的人往往拥有的游戏实际上并不是游戏，它们是最终将成为游戏的模拟器，其中常常将“我有一个移动的角色”视为游戏玩法的一部分，核心关注点可能是以下之一：

- 过程生成的世界、行星、太空、地下城。
- 基于体素的任何东西，深入研究体素本身、渲染体素、世界大小和性能。
- 通用交互，“任何东西都可以与其他任何东西做XXX”。
- 以最优方式进行渲染：“如果不使用间接绘制，你算是在制作游戏吗？”
- 拥有良好的类型和用于构建游戏的“框架”。
- 构建一个引擎，用于制作更多类似即将建立的游戏。
- 多人游戏。
- 大量GPU粒子，粒子越多，视觉效果越好。
- 结构良好的ECS和清晰的代码。
- ...等等

从技术探索和学习Rust的角度来看，所有这些目标都是可以接受的，但我要再次强调本文开头所说的。我不是从技术的好奇心或“满足大脑的渴望”角度评估Rust。我想制作真正的游戏，以合理的时间内发送给真正的玩家（而不是开发

者），这些人将为之付费并进行游戏，并有可能登上Steam的首页。澄清一下，这不是冷血的“不惜一切代价赚钱”的计划，但也不是“我只是为了玩乐而这样做”。整篇文章都是从一个关心游戏、游戏玩法和玩家的认真游戏开发者的角度来写的，而不仅仅是技术的热情。

再次强调，技术的热情没有错，但我认为人们应该非常谨慎地确定他们的实际目标，最重要的是要诚实地说明他们为什么要做自己正在做的事情。有时候我觉得一些项目展示自己的方式以及人们谈论这些项目的方式是虚假宣传，它创造了一种商业目标可以通过这些方法实现的错觉，而不是更明确地表明“我只是为了技术本身而这样做”。

现在回到通用系统。以下是我认为能够创造出好游戏的一些因素，它们直接或间接地与通用ECS方法相悖：

- 大多数关卡都是手工设计的。这并不意味着“线性”或“故事”，但确实意味着“对于玩家何时看到什么有很大的控制权”。
- 在整个关卡中精心设计的各种交互。
- 不基于大量相似粒子的视觉效果，而是基于时间同步事件（例如，多个不同的发射器按照手工设计的时间表触发）在游戏的所有系统中工作。
- 反复进行游戏特色的游戏测试，进行实验并放弃不起作用的部分。
- 尽快将游戏发送给玩家，以便进行测试和迭代。没有人看到它的时间越长，当它发布时就越有可能没有人关心它。
- 独特而令人难忘的体验。

我理解，阅读到这里，很多人可能会认为我在想象一个由画家制作的艺术氛围的游戏，而不是一个真正想制作类似于Factorio的游戏的程序员，但这并不正确。我仍然喜欢系统性的游戏，我喜欢编码，我想制作一个由编程驱动的东西，因为我觉得自己主要是一个程序员。

我认为大多数人的误区在于,他们错误地将仔细思考玩家互动和设计玩家互动视为一种艺术创作。我想说,这才是游戏开发的本质。游戏开发不是构建物理模拟,不是构建渲染器,不是构建游戏引擎,不是设计场景树,也不是设计具有数据绑定的响应式UI。

一个很好的例子是《以撒的结合》（The Binding of Isaac），这是一个非常简单的类似地下城探险的游戏，有着数百种修改游戏的升级方式。它是一个拥有多个相互作用系统的游戏，但它也不是一个通用的游戏。它不是一款拥有500种“+15%伤害”升级的游戏，而是许多升级是“炸弹粘在敌人身上”或“你发射激光而不是子弹”或“每个关卡你杀死的第一个敌人不会再次生成”之类的升级方式。 回顾这样的游戏可能会让人觉得，这是一种可以事先使用通用系统设计的东西，但我认为这也是大多数人在游戏开发中完全走错了方向的地方。你不能通过一年坐在地牢里，思考所有边缘情况，构建一个通用系统，然后通过生成升级来制作一个好游戏。你需要使用一些基本机制构建一个原型，并让人们玩一下，确保核心要素运作正常，然后再添加更多内容，让人们再次玩游戏。其中一些互动必须通过深入了解游戏后，玩了许多小时并尝试了许多不同的事物后才能发现。

Rust 是这样一种语言：想要进行新类型的升级可能会让你走上重构所有系统的道路，许多人甚至会说“太好了，现在我的代码更好了，可以适应更多的事情！！！”。这听起来像一个非常有说服力的论点，我听过很多次，也因此浪费了很多时间去追求解决错误问题的解决方案。

一种更灵活的语言将允许游戏开发者立即以一种糟糕的方式实现新功能，然后玩游戏，测试它并查看该功能是否真的有趣，并在短时间内进行许多此类迭代。当Rust开发人员完成重构时，C++/C#/Java/JavaScript开发人员已经实现了许多不同的游戏功能，玩了很多次，并尝试了所有功能，对游戏的发展方向有了更好的理解。

乔纳斯·泰罗尔在他的游戏设计搜索视频中非常好地解释了这一点，我强烈推荐每个游戏开发者观看，因为它感觉是对为什么许多人制作的游戏（包括我自己）如此糟糕的最佳解释。一个好游戏不是在实验室中精心制作的，而是由一位是该类型的高手玩家的开发者制作的，他了解设计的每个方面，并在达到最终设计之前尝试了许多失败的尝试。一个好游戏是通过摒弃许多不好的想法，通过非线性的过程制作出来的。

制作有趣的游戏是关于快速原型和迭代，Rust的价值观与此截然不同

为了更好地定义本文中的“游戏开发”这一点，我们需要定义一下这个概念。我们不谈论AAA游戏，或者一般来说规模庞大、非常长期的项目。我认为除非已经具有大量游戏开发和所使用工具的先前经验，否则没有人能够真实地认为他们能够建立一个成功的五年游戏项目。我们谈论的是由个人或小团队制作、在相对紧张的预算和时间限制下的独立游戏。

其次，制作游戏的原因有很多，但我们的意图是制作其他人会玩并认为好的东西，而并不在意使用了哪种技术/引擎/框架/思想，也不了解或与作者有任何先前接触。我觉得这一点尤其需要强调，因为尽管Rust社区总体上非常支持游戏开发，但它经常会营造出一个非常错误的观念：“这太酷了，人们一定会喜欢这样的游戏。”这不仅是Rust所面临的问题，许多游戏开发者最终会向其他游戏开发者和游戏开发社区展示他们的游戏，并因此陷入同样的谬误。

由于Rust社区的一般氛围，人们普遍会得到积极的反馈。这对于心理健康和短期动力来说是好的，但我曾多次经历过在Steam上公开发布东西的过程，我觉得一旦不属于他们的朋友群体/社区的人看到他们的游戏，很多人将会有一种

痛苦的认识。我之所以说这个是因为我认为整个社区已经接受了这种对一切与Rust相关的东西无止境的积极评价和赞扬的想法，完全将自己与外界隔离开来。

但现实世界的玩家并不那么友善。Steam上的玩家不关心游戏是否使用Rust制作，他们不关心制作花了5年时间，也不关心代码是否开源。**他们关心的是看两眼游戏，在几秒钟内能够判断这是否会浪费时间，或者是否有趣。**

我见过很多人将这些事情视为"年轻一代"、"注意力不集中"、"多动症这/那"和"人们**应该**欣赏XYZ"的观点。我不认为这些观点有任何帮助，因为每个人都会这样做，只是当谈到_我们的游戏_时，我们作为游戏开发者有偏见。当你在杂货店购物时，看到香蕉，其中一些颜色稍微有点丑陋或有点破损，你会选择看起来更好的那些。当去餐厅时，你会选择一个看起来会有好食物和好价格的餐厅，或者至少提供你在乎的体验。

我甚至可以说，玩家不关心开发者，只是看游戏几秒钟，这是正确和可取的，至少这让我们保持诚实。这使得游戏只关注游戏本身，而不是其他任何东西，因为最重要的是游戏本身和玩游戏的体验。

这也揭示了作为游戏开发者应该迎合的价值观。如果你展示你的游戏，但回应不是"我想玩这个！"，那么你向他展示的游戏对他来说不是有趣的。至少从商业成功的角度来看，不是真正有趣的。

人们经常争论Rust的吸引力是"可维护性"以及这如何导致更好的游戏不会崩溃，但我认为这里的问题完全不同。我们都可以同意，当有人按下播放按钮时游戏崩溃是不好的，当保存文件损坏并导致玩家失去进度时也是绝对不好的。

但我认为所有这些完全忽略了对玩家来说重要的是什么。有很多情况下，人们的进度可能会被清除，但他们仍会回到游戏中并再次玩游戏，因为游戏非常好。作为玩家，我不止一次这样做过。

Rust作为一种如此专注于不惜一切代价避免问题语言和社区，完全忽视了最重要的事情，即提供一个如此出色的体验，以至于存在的任何问题都不重要。这并不意味着"发布垃圾游戏"，而是专注于使游戏成为一款游戏层面上的好游戏，而不仅仅是代码层面上的好代码。

宏不是反射的简单解决方案

作为编程领域的游戏开发通常需要编写多种类型的代码。我们需要用于处理碰撞、物理、粒子等系统级代码。我们还需要编写用于"脚本化"实体行为的游戏代码。我们还有用户界面、视觉效果、音频等。此外，我们还有一些工具。根据正在构建的游戏的类型，每个类别的大小可能会有所不同，但经过足够多不同类型游戏的开发后，我可以说的是，通常来说，每个方面都需要花费一定的精力。

Rust非常适合在低级算法领域中使用，其中我们完全了解问题所在，只需要解决问题即可。不幸的是，许多游戏开发需要更动态的方法，尤其是在关卡编辑、工具和调试方面，这变得尤为困难。

即使是像"打印这个对象"这样简单的问题，也无法合理地解决，除非编写代码或创建过程宏。现在许多语言都有宏，对于那些没有足够长时间使用Rust的人来说，他们可能不知道Rust中有两种类型的宏：

- **声明式宏：** 这些宏相对简单易创建且非常有用，但遗憾的是，它们的功能相当有限。在Rust中，许多事情都以"安全性"为首要考虑，而在C预处理器宏中完全可以接受的事情在这里就变成了无法解决的问题。最简单的例子就是连接标记，现在有一个著名的`paste`包，它使用过程宏提供了部分解决方案。从表面上看，你可能会认为_太好了，问题解决了，对吧？_...但遗憾的是，甚至连这都远远不够，例如，嵌套和混合过程宏和声明式宏并不总是能够工作，而且在花费了_大量_时间研究技术细节之前，甚至无法明确知道什么是可能的，为什么是可能的。
- **过程宏：** 作为一个核心概念，过程宏基本上是：允许程序员在编译时运行代码，使用Rust的AST，并生成新的代码。遗憾的是，这方面有_许多_问题。首先，过程宏并没有真正被缓存，而是在重新编译时重新运行。这最终迫使你的代码被分割成多个包，这并不总是可能的，而且如果你更重度地依赖过程宏，你的编译时间将会因此大幅度增加。有许多方便的过程宏，如`profiling`的`function`宏，这些宏非常非常有用，但最终却无法被使用，因为它们破坏了增量构建时间。其次，过程宏非常难以编写，大多数人最终都会使用非常重的辅助包，如`syn`，这是一个非常重的Rust解析器，它会急切地解析所有有用了它的东西。例如，如果你想在你的宏中注解一个函数并只解析它的名字，`syn`将会解析整个函数体。还有一种情况是，`syn`的作者也是`serde`的作者，这是一个流行的Rust序列化包，去年的某个时候开始在补丁发布中附带一个二进制blob，拒绝了社区的反对。这并不是针对Rust的一个案例，但我觉得应该提一下，因为它展示了大部分生态系统是如何建立在由单个开发者制作的库上的，这些开发者可能会做出潜在的危险决策。当然，这在任何语言中都可能发生，但在过程宏方面，这是_非常_重要的，因为生态系统中几乎所有的东西都使用了这个特定作者制作的包（`syn`，`serde`，`anyhow`，`thiserror`，`quote`，...）。

即使忽略上述问题，过程宏的学习曲线非常陡峭，而且必须在一个单独的crate中进行定义。这意味着与声明宏不同，你不能轻松地创建一个新的过程宏。

相比之下，使用C＃中的反射非常容易，如果不考虑性能（在使用反射的情况下通常不重要），它可以是构建工具或调试的快速且有用的选择。Rust没有提供任何类似的功能，而去年的一场Rust争端基本上取消了编译时反射的最后方法。

由于本文旨在保持技术性，我不认为详细解释这场争议或试图站队有太多价值，因为尽管这些对不同的人来说重要程度不同，但社区的共识实际上是在可预见的将来不会再出现编译时反射，这对所有与该语言相关的人来说都是非常令人沮丧的。过程宏是一个重要且强大的工具，但它们对独立游戏开发的实用性非常低，因为它们的开发成本和复杂性有点高，无法用于解决可以轻松通过反射解决的次要问题。

热重载对于迭代速度的重要性被低估了

在讨论Rust和热重载之前，我想提一些事情。

首先，如果你还没有看过Tomorrow Corporation Tech Demo，我强烈建议每个游戏开发者观看这个视频，看看在热重载、可逆调试和游戏开发工具方面有什么可能。即使你认为自己知道这些东西是什么，也请观看这个视频。我一直觉得热重载至少在某种程度上是重要的，但看到这些人构建的东西真的让我为我以前觉得某些工作流程对于DX是足够的感到羞愧。

对于那些还没有观看过视频的人，以下是Tomorrow Corporation的团队所做的事情：

- 构建了自己的编程语言、代码编辑器、游戏引擎、调试器和游戏。
- 在整个技术栈中构建了热重载支持。
- 提供了可逆的时间旅行调试功能，可以在游戏状态之间切换。
- ...只需观看视频 :) 我保证你不会后悔

我理解在现有平台（如.NET）或原生语言（如C++或Rust）中普遍的构建类似的东西几乎是不可能的，但我也不接受这样的论点：即仅因为它很困难并且无法完美运行，我们就不应该努力追求这些东西。 有_许多_现有的平台/语言支持不同程度的热重载。在我的探索过程中，我甚至使用Common Lisp制作了一个游戏，以了解其热重载能力。我不一定建议其他人这样做，但也不必走得那么远。

自从.NET 6推出以来，我们现在可以在任何C#项目中进行热重载。我听说过人们对此报告了不同的体验，我自己也尝试过，对于一些最近的争论，特别是那些不是“在它推出时我就试过”的观点，我很难认真对待。在Unity的环境中，现在有一个名为hotreload.net的自定义实现，专门为Unity开发，我已经使用了大约4个月，它在提高生产效率方面非常出色。这实际上是我们回到Unity的主要原因。这不是我们放弃Rust的原因，但这是我们选择Unity而不是Godot或UE5的原因。（截至撰写本文时，Godot不支持.NET热重载，UE仍然只支持蓝图和C++。）

在本文中，我们可以专注于热重载函数体，即在函数内部更改代码，并进行热重载。不知怎么的，在Rust生态系统中，这是一个有争议的话题，许多人乐意争论如果它不能做到一切就没有用，或者它的限制太多以至于无用，或者潜在的错误可能性超过了任何可能的好处。

在游戏开发的背景下，我很难理解其中的任何观点。游戏绝不是无状态的数据处理器。

热重载变得非常有用的几种情况：

- 无论是UI还是绘图的即时模式。即使编译时间很快，迭代速度也得到了显着提高，因为不必不断重新输入相同的状态。
- 使用即时模式绘图/几何进行调试。这可能是我最喜欢的用例，特别是在调试角色控制器和物理效果时，我可能会进入意外/有错误的状态，使用热重载，我可以简单地添加几行代码来在游戏中绘制相关值，以查看发生了什么，而无需再次复现问题。
- 调整影响游戏玩法的常量。虽然在某些情况下，用新值重新启动游戏可能会导致不同的结果，但游戏并不是科学实验。我们不需要可重复性，我们需要的是_乐趣_。当我可以在游戏进行时调整值时，优化乐趣要容易得多。像`inline_tweak`这样的包在这里很有用，但它们需要预见。热重载允许我在处理其他功能时突然想到一个想法“我想知道如果...”，然后就可以随意进行尝试，而不会打乱我之前的工作。

值得注意的是，Rust实际上已经有了解决方案，即`hot-lib-reloader`，但我尝试过，它并不完美，即使只是重新加载函数的简单用例也会出现很多奇奇怪怪的问题，最终我放弃了，因为它给我带来的麻烦比节省的时间还多。即使这个包没有任何问题，它也无法解决随时调整的问题，因为它仍然需要计划和预见，这降低了潜在的创造性用途。

许多人用“但编译器做了XYZ”来反驳热重载，我很想建议一些永远不会合并的东西，但是这样做会很好。如果有一个编译器标志...是的，我已经能够看到人们大喊“可怜的编译器团队”...我猜我们永远不会有这个。

存在许多部分解决方案，但没有一个能接近_真正的热重载_的实用性，这就是我称之为.NET和Unity目前具备的。脚本语言是部分解决方案，由于许多原因，在Rust中存在问题，手动实现的动态库热重载受到限制，任何形式的状态序列化和重新启动只适用于大型代码更改，而不仅仅是可调整性。并不是说这些东西没有用，但我认为作为游戏开发者，我们应该期望比“我可以重新加载代码中的一些结构”更高级的工具。

抽象不是一个选择

这一部分是由我在开发我们的游戏时刚写的一个非常简单的代码示例所激发的。我有一个UI，其中包含一个角色列表和一个在选择角色（鸭子）时出现的详细页面。

在我的UI代码组织方式里，UI的每个状态都有一个帮助函数，因为我们使用egui，而即时模式要求大多数东西在大多数地方都可用。这实际上非常好使，因为像这样的东西可以正常工作

```
egui::SidePanel::left("left_panel").frame(frame).show_inside(
    ui,
    |ui| {
        ui.vertical_centered(|ui| {
            character_select_list_ducks(egui, ui, gs, self);
        });
    },
);

egui::SidePanel::right("right_panel").frame(frame).show_inside(
    ui,
    |ui| {
        character_select_recover_builds(ui, gs, self);
    },
);

egui::TopBottomPanel::bottom("bottom_panel")
    .frame(frame)
    .show_inside(ui, |ui| {
        character_select_missing_achievements(
            egui, ui, gs, self,
        );
    });
```

但是，假设其中一些具有条件状态，并且它们的实现实际上相当复杂。这种情况出现在选择特定的duck时。最初，我的代码如下所示：

```
egui::CentralPanel::default().frame(frame).show_inside(
    ui,
    |ui| {
        character_select_duck_detail(ui, gs, self);
    }
});

fn character_select_duck_detail(..., state: ...) {
        if let Some(character) = state.selected_duck {
            // some UI
        } else {
                // other UI
        }
}
```

这样做是可以的，但问题是 `egui` 经常需要非常深的嵌套，因为几乎每个布局操作都是一个闭包。如果我们能够减少嵌套并将 `if` 移到外面会很好。最终，我们也会分离出两个明确的不同的东西... 我的第一个想法是这样的：

```
if let Some(character) = &self.selected_duck {
     character_select_duck_detail(.., character, self);
} else {
   character_select_no_duck(...);
}
```

但是这样我们当然会遇到问题，我不可用在传递 `self` 的同时借用 `self` 上的字段。

即使使用 Rust 已经有几年了，我有时仍然会过多地思考 UI 或游戏，而过少地思考如何组织我的代码，最终导致出现这样的问题。Rust 人的本能反应是 "显然你需要将状态分离，而不是传递一个大的结构体"。这是对的，但这是一个很好的例子，说明了 Rust 与最自然的做事方式之间的冲突。

因为在这种情况下，我们正在构建一个单独的 UI 窗口。我不想花费任何脑细胞去考虑 UI 的哪些部分需要状态的哪些部分，我只想传递我的状态，它并不大。当我在之后的 15 分钟里添加更多字段时，我也不想花费额外的时间来修改代码以传递更多字段，我几乎可以肯定我会这样做。我也不想将数据分成多个结构体，因为我可能想对多个数据进行 `if` 操作。而且，我在之前就已经尝试过 "拆分结构体" 的方法，很少能一次成功。

解决方案？和 Rust 中的许多事情一样，我们会感到自己仿佛是个🤡（小丑），然后将代码更改为：

```
if let Some(character) = &self.selected_duck.clone() {
     character_select_duck_detail(.., character, self);
} else {
   character_select_no_duck(...);
}
```

现在一切都正常工作了，借用检查器很高兴，我们每一帧都在克隆一个字符串。它甚至不会在性能分析器中显示出来，所以从总体上来说并不重要。但是，对于一个旨在快速和高效的语言来说，不得不经常重新分配内存来保持编程效率的高效，是一件令人遗憾的事情。

我之所以提到这个特定的情况，是因为它相当典型地反映了我编写Rust代码的整体经验：许多问题只是通过额外的复制或克隆来解决的。这是大多数Rust开发人员熟悉的情况，但对于我帮助学习Rust的许多人来说，这是一个令人惊讶

的发现。他们通常的反应是："等等，我以为Rust应该非常快和高效"，对此我们只能回答："哦，它确实很快，不用担心，在这种情况下，每一帧都克隆字符串是完全无害的"，然后再次感到自己像个🤡。

Rust中的GUI情况糟糕透了

就像有一个笑话说Rust有5个游戏和50个游戏引擎一样，我们可能还需要另一个笑话来描述GUI框架的情况。人们正在尝试许多不同的方法，这在Rust作为一种完全通用语言的情况下是有道理的。但在本文中，我们将重点关注游戏开发，并且我感觉在这方面我们不仅严重缺乏，而且我甚至看不到出路。

现在，当我说UI时，我指的是游戏内的UI，而不是构建编辑器的UI。这是一种必须高度样式化和可视化的UI。至少根据我的经验，构建游戏UI最困难的部分不是弄清楚如何进行数据绑定，或者如何使事物以响应方式更新，甚至不是如何最好地描述布局。而是自定义UI的外观和感觉。

这甚至没有涉及到像_UI中的粒子_或用户可能想要的各种效果这样的事情。显然，一个完全不考虑任何事情的GUI库无法拥有花哨的着色器效果和粒子，但我认为这也是整体方法问题的一部分。GUI库将所有这些都推给用户去解决，然后每个用户都要在他们选择的框架/引擎中重新发明轮子。

我们最终在`egui`中完成了大部分的UI工作，尽管在许多方面都不是最优的，甚至是令人困惑的，但它至少提供了一个`Painter`接口，用于完全自定义的UI。

当提到这一点并说Unity或Godot中的UI情况要好得多时，人们总是会说类似于"哦！我试过Unity，糟糕透了，我在代码中做UI比在编辑器里更加开心"。这是一个非常常见的回答，我以前也会这样说，但这种说法完全忽视了构建UI是一种技能，而在像Unity或Godot这样的复杂UI工具包中进行构建是复杂而烦人的，因为需要学习。

响应式UI并不是解决高度可视化、独特和交互式游戏UI的答案

在 Rust 中 有 许多 GUI 库，他们采用了许多不同的方法。有些是对现有GUI库的绑定，有些是即时模式，有些是响应式的，甚至还有保留模式。有些尝试使用flexbox，而其他一些甚至根本不涉及布局。

问题是，就游戏开发而言，我不确定我们是否有任何适合的方法。我们拥有如此多的库的原因与我们拥有如此多的游戏引擎的原因是一样的，那就是在Rust生态系统中实际上几乎没有人在制作游戏。

至少在我看来，游戏GUI并不是那么关心数据更新速度最快，具有响应式重新渲染、数据绑定或最先进的声明性布局描述的问题。

正相反，我只是希望拥有一个非常漂亮的GUI，具有许多定制的精灵、动画、矢量形状、粒子、效果、闪光等。当点击按钮时，我希望它能摇摆，当悬停在文本上时，我希望文本能动起来，我希望能够使用自定义着色器并用噪声纹理扭曲它。当选择角色框时，我希望粒子飞来飞去。 我理解有些游戏可能希望渲染一个拥有百万个元素的表格，但我不认为这应该是一个游戏 GUI 库的目标。我也知道，上面列出的许多（如果不是全部）GUI库都没有将自己定位为游戏 GUI 库，这也就是我想在本节中点出的部分问题。

据我所知，在 Rust 生态系统中，没有一个解决方案的目标是"擅长制作游戏 GUI"。我理解，在一个可能希望与引擎无关的库中实现"粒子和着色器"之类的功能并不容易，但这也可能是情况难以改善的另一个原因。

我确实认为，大多数游戏希望拥有摇摆的按钮、动画化的文本、以及以各种奇怪方式旋转的框，甚至可能还会出现某种不可思议的模糊效果。希望有这些东西并不是疯狂的一件事。

孤儿规则应该是可选的

这一部分可能会很简短，因为我认为任何试图编写大量用户层 Rust 代码的人都会感受到孤儿规则的痛苦。它是一个对我称之为"Rust 对安全的追求"的很好的例子，即对完美和完全避免所有问题的渴望，即使这意味着开发人员的工作体验显著下降。

对于希望将库上传到 crates.io 等地方的库来说，希望有孤儿规则的原因大多是合理的，我承认这一点。

但是对于开发端产品中的应用程序和库来说，我很难同意这个规则。我当然不是在说"二进制库"，因为大多数较大的项目由多个 crate 组成，其中许多是一个工作区中的多个 crate。

实际上，我认为即使对于已发布的库，我们也应该能够有办法禁用这个规则，因为其中一些并不是真正被其他下游库使用的库。游戏引擎和框架就是一个很好的例子，使用类似 Macroquad 或 Comfy 这样的库的人在他们的代码库中并不需要遵守孤儿规则。对于"框架型"库来说，能够在不分叉的情况下扩展现有功能，并为最终用户提供更统一的体验将非常有益。

但不幸的是，就像 Rust 中的许多事情一样，"完美只存在于绝对中"，只因为有可能有人可能实现了一个冲突的 trait，我们就必须在任何情况下禁止所有人使用，而且不能禁用这个规则。

- 协程/异步和闭包与高级语言相比，DX非常差劲
- 无论使用什么工具或者在什么操作系统上，Rust 的调试都很糟糕

编译时间有所改善，但编译过程宏依旧很慢

自从 Rust 的编译时间变得非常糟糕以来已经过去了几年，整体情况在某种程度上肯定有所改善，至少在 Linux 上是如此。在 Windows 上进行增量构建仍然明显较慢，以至于我们最初转向了 Linux（差距为 3-5 倍），但至少在购买了一台新的高端台式机之后，构建我们的 10k 行代码库只需要几秒钟的时间。

这是在经过大量时间优化编译时间、移除处理过程宏以及将事物移入各自的 crate 后的结果。

一个很好的例子是 `comfy-ldtk` 的存在只是为了封装一个单一文件，并确保 `serde` 的单态化发生在一个单独的 crate 中。这可能看起来是一个琐碎的细节，但至少在我的台式机上，这导致了增量构建时间从 2 秒增加到了 10 秒。对于 1600 行结构定义来说，这是一个相当大的差异。

现在我明白，序列化并不是一件简单的事情，我也理解 `serde` 有很多功能。但我也不认为在编译 1600 行代码时花费 8 秒是合理的任何情况。特别是当你看到代码并发现它只是一些简单的结构体时。这里没有复杂的通用魔法，所有的问题都归结为 `serde` 的速度慢。 我见过很多人对这类事情不在乎，而且我个人在许多不同的情境中多次提出增量编译时间的问题，总有一部分人会说服我这没问题，他们的构建时间需要20-30秒甚至更长，但他们仍然能保持高效率。

冒昧地说，我只能将这归因于缺乏对更好工具的经验，或者他们的游戏还没有达到真正需要快速迭代的阶段。或者至少，我觉得有些人并没有意识到，如果编译时间从30秒减少到0.5秒，他们的游戏能更加精细。GUI等东西本质上是需要微调的，除了`godot-rust`的用户外，其他人都不得不多次重启游戏以达到良好的外观效果。如果你的经验与此不同，我很愿意看到一个使用超过30秒增量构建时间构建了非常精细且不平凡的GUI的例子。

Rust游戏开发生态圈充满了炒作

毫无疑问，Rust游戏开发生态圈还很年轻。当你在社区里询问问题时，大多数人在提到问题时都会承认这一点，至少在2024年，我们对此已经不再缺乏认识。

然而，我认为外部世界的观点却大不相同，这要归功于Bevy和其他一些项目在营销方面的非常好的推广。就在几天前，Brackeys发布了他们关于回归游戏开发并使用Godot开发的视频。当我观看这个视频并开始听到所有这些令人惊叹的开源游戏引擎时，我已经有了一种感觉。在大约5:20的时候，视频中出现了一张_游戏引擎市场地图_的图片，我只能说当我看到三个Rust游戏引擎时，我真的感到非常震惊，尤其是这三个引擎：Bevy、Arete和Ambient。

现在我想要特别明确一下，这篇博文并不是对任何特定项目的抨击，我理解这些项目并不对其他人在他们的视频中做的事情负责。但与此同时，这已经成为了Rust世界中的一个主题，甚至可以说是一个梗，我觉得应该谈谈这个问题。

在Rust生态系统中，通常的运作方式是，哪个项目能做出最多的承诺，展示最好的网站/自述文件，有最炫酷的动态图，最重要的是，能吸引到正确的抽象价值观，就会得到广泛的赞誉，无论该项目的实用性如何。然后还有一些其他的项目，它们通常不为人知，因为它们并不引人注目，也不承诺无法实现的功能，而只是试图以一种可行的方式做某件事，这些项目几乎从未被提及，或者当它们被提及时，它们被视为下策。

第一个例子是Macroquad，这是一个非常实用的2D游戏库，可以运行在几乎所有平台上，具有非常简单的API，编译速度非常快，几乎没有依赖。这个库是由一个人构建的。还有一个配套库`miniquad`，它在Windows/Linux/MacOS/Android/iOS和WASM上提供了一个图形抽象层。然而，Macroquad犯了Rust生态系统中最严重的错误之一，那就是使用全局状态，甚至可能存在潜在的不安全性。我说可能是因为我知道纯粹主义者会说"不，这不是一个问题，它是错误的"，因为就所有的目的而言，除非你决定使用最底层的API来触及OpenGL上下文，否则完全可以安全使用。我使用Macroquad已经将近两年了，从来没有遇到过这个问题。然而，每当有人建议使用Macroquad时，这个问题永远会被提到，因为它并不符合Rust的最终价值观，即100%的安全性和正确性。

第二个例子是Fyrox，这是一个具有完整3D场景编辑器、动画系统和几乎满足制作游戏所需的一切的3D游戏引擎。这个项目也是由一个人完成的，他也在该引擎中制作了一个完整的3D游戏。个人而言，我没有使用过Fyrox，因为就像这个部分提到的，我个人也曾经犯过追逐炒作并选择那些有漂亮网站、很多GitHub星标并以某种方式展示自己的项目的错误。Fyrox最近在Reddit上开始引起一些关注，但令我非常遗憾的是，几乎没有人在任何视频中提到它，尽管它拥有一个完整的编辑器，而Bevy多年来一直承诺要做到这一点。

第三个例子是`godot-rust`，这是Rust对Godot Engine的绑定。这个库犯下的最严重的罪行是它不是一个纯Rust的解决方案，而是对一个肮脏的C++引擎的绑定。我有点夸张，但是那些从外面看Rust的人可能会惊讶于这有时候是多么接近现实。Rust是纯粹的，Rust是正确的，Rust是安全的。C++是糟糕的，旧的，丑陋的，不安全的，复杂的。这就是为什么在Rust游戏开发中我们不使用SDL，我们有`winit`，我们不使用OpenGL，我们有`wgpu`，我们不使用Box2D或PhysX，我们有`rapier`，我们有`kira`用于游戏音频，我们不使用Dear ImGUI，我们有`egui`，最重要的是我们肯定不能使用一个用C++写的现有游戏引擎。那将是对每个使用`rustup default nightly`来获得更快编译时间的人在许可证中同意的神圣的螃蟹代码的忤逆（我们被禁止使用的由Rust基金会官方认可的标志）。

如果有人真的想在Rust中制作一个真正的游戏，尤其是3D游戏，我的第一推荐是使用Godot和`godot-rust`，因为至少它们有机会提供他们需要的所有功能，因为他们可以依靠真正的引擎来帮助他们实现。我们花了一年时间使用Godot 3和`godot-rust`的gdnative构建了BITGUN，虽然在很多方面这个经历都很痛苦，但这并不是绑定的错，而是因为我们试图以各种可能和动态的方式混合大量的GDScript和Rust。这是我们的第一个也是最大的Rust项目，也是我们走上Rust之路的原因，最终我会说，我们用Rust制作的每个游戏都不再是一个游戏，仅仅因为我们花了很多时间去解决与Rust语言、生态系统的某些部分或者某些设计决策相关的不相关的技术问题，这些问题由于语言的严格性而变得困难。我不会说GDScript和Rust的互操作很容易，它确实不容易。但至少Godot提供了"只需做好事情并继续前进"的选择。我觉得大多数尝试纯代码解决方案的人并不重视这一点，尤其是在Rust中，这种语言以诸多不便阻碍了我们的创造力。

关于Ambient，我没有太多可说的，因为它比较新，而且我没有用过它，但同样，我也不知道有没有其他人使用过它，不过它却出现在了Brackeys的视频中。

Arete几个月前发布了0.1版本，由于它对其声明非常模糊并且同时是闭源的，实际上在Rust社区中收到了相对负面的反应。尽管如此，我在许多场合看到外部人员提到它，而且经常有很大胆的说法。

至于Bevy，我确实认为将其展示为"主要"的Rust游戏引擎在很大程度上是合理的，至少是因为该项目的规模和参与人数。他们成功地建立了一个非常庞大的社区，虽然我可能不同意他们的承诺和一些领导层的选择，但我不能否认Bevy很受欢迎。

这一部分的目的只是为了让人们对事情的奇怪状态有所认识，外部人员通常只会看每个引擎在市场营销方面表现如何以及他们在发布博客文章时说了什么。我之所以觉得有必要提到所有这些事情，是因为我曾经走过这条路，多次看到人们会说一些令人信服的话，但后来才意识到他们只是擅长画饼，而不擅长把饼做出来。

还有一个值得一提的例子不是游戏引擎，而是`rapier`，一个经常被推荐的物理引擎，因为它承诺成为物理的纯Rust解决方案，是Box2D、PhysX和其他丑陋外部库的一个很好的替代品。毕竟，Rapier是用纯Rust编写的，因此享受到了WASM支持的所有好处，同时也非常快速，核心部分是并行的，当然也非常安全...对吗？ 我的经验主要来自于2D领域，在基本功能方面，一切都正常运作，但一些更高级的API却存在根本性的问题，例如凸包分解在相对简单的数据上崩溃，或者多体关节在移除时导致崩溃。后者尤其有趣，因为这让我觉得我是第一个尝试移除关节的人，而这似乎并不是那么高级的用法。这些可能看起来像是极端情况，但总体而言，我也发现这个模拟器相当不稳定，以至于我最终写了自己的2D物理引擎，并在我的测试中至少发现它在像"防止敌人重叠"的简单问题上引起的问题没那么多。

这并不是为了推广我的物理库，请不要使用它，因为它没有经过充分的测试。关键是，如果一个刚接触Rust的人问有什么推荐的物理库，他们会被推荐rapier。很多人会说它是一个很好且流行的库，有一个漂亮的网站，在社区中广为人知。作为曾经的那个人，我真的为此苦苦挣扎了几个月，并且一直认为"肯定是我做错了什么"，我之所以感觉"找到了答案"，只是因为我试图重新实现它。

Rust生态系统中的很多东西会让用户感觉自己在做一些根本上错误的事情，他们不应该希望做某些事情，他们想要构建的项目是不可取的或不正确的。这种感觉类似于使用Haskell并且希望进行副作用...这不是"你应该想要的"事情。

不过，在Rust的情况下，问题在于很多时候，让用户产生这种感觉的库会得到普遍的赞誉和认可，因为大部分的生态系统都在画饼，而不是干活。

因为编译器对单线程游戏的过高要求，全局状态做起来很不方便

我知道仅仅说"全局状态"就会立即引发许多对此持有强烈意见的人的反感。我觉得这是Rust社区在项目/人员上创造的非常有害和不切实际的规则之一。不同的项目有着截然不同的要求，在游戏开发的背景下，我觉得很多人对实际问题有了错误的判断。对全局状态的整体"厌恶"是一个连续的事情，大多数人不会完全反对它，但我仍然觉得整个社区在走向错误的方向。再次强调，我们不谈论制作引擎、工具包、库、模拟器或类似的东西。我们谈论的是"游戏"。

对于一个游戏来说，只有一个音频系统、一个输入系统、一个物理世界、一个`deltaTime`、一个渲染器、一个资源加载器。也许在某些极端情况下，如果某些东西不是全局的话会稍微方便一些，如果你正在制作一个基于物理的大型多人在线游戏，你的需求可能不同。但是大多数人要么在构建2D平台游戏，要么是俯视射击游戏，要么是基于体素的行走模拟器。

多年来，我实际上多次尝试过使用纯粹的方法，将一切作为参数注入（从Bevy 0.4开始，到0.10），并尝试构建我的自己的引擎，在那里一切都是全局的，播放声音只需`play_sound("beep")`，我对哪种方法更有用非常清楚。

这并不是特别针对Bevy，我认为整个生态系统的很大一部分都有这个问题，唯一的例外是macroquad，但我选择以Bevy为例，因为它处于另一个极端，在那里一切都是显式传递的。

以下是在Comfy中对我们的游戏非常有用并且我一直在使用的一些功能，它们使用全局状态：

- `play_sound("beep")` 用于播放一次性音效。如果需要更多控制，可以使用`play_sound_ex(id: &str, params: PlaySoundParams)`。

- `texture_id("player")` 用于创建一个`TextureHandle`来引用一个资源。没有资源服务器需要传递，因为在最坏的情况下，我可以使用路径作为标识符，而且由于路径是唯一的，显然标识符也将是唯一的。
- `draw_sprite(texture, position, ...)` 或者 `draw_circle(position, radius, color)` 用于绘制。由于每个非玩具引擎都会批量绘制调用，所以这些调用不会比仅仅将绘制命令推送到某个队列中多做任何事情。我很乐意拥有一个全局队列，因为仅仅是将"绘制圆形"推送到队列中，我为什么还要关心传递任何东西呢？

如果你作为一个Rust开发者，而不是一个游戏开发者，来阅读这篇文章，你可能会想到"但是线程呢？？？"是的，这也是Bevy作为一个很好的例子的原因。因为Bevy提出了这个问题，并试图以最一般的方式回答，"如果我们让所有的系统并行运行会怎么样"。

这是一个很好的理论想法，对于许多新手来说可能很有吸引力，因为就像在后端领域一样，所有的事情都是异步的并在线程池上运行，这似乎会带来免费的性能。

但不幸的是，我觉得这是Bevy犯下的最大错误之一，并且在询问过这个问题之后，我觉得很多人开始意识到这一点，尽管很少有人真正承认。Bevy的并行系统模型非常灵活，甚至在帧之间也没有保持一致的顺序（至少在我上次检查时是这样）。如果想要保持顺序，就需要指定约束条件。 这起初似乎合理，但在尝试使用Bevy制作一个非平凡的游戏时（几个月的开发时间，数万行代码），最终发生的情况是用户仍然需要指定大量依赖项，因为游戏中的事物往往需要按特定顺序进行，以避免一帧帧的随机延迟，取决于先运行什么，或者更糟糕的是，有时会出现奇怪的行为，因为你得到了AB而不是BA。当你提出这个问题时，你会遭到激烈的反对，因为Bevy所做的是"技术上正确的"，但对于实际制作游戏来说，这只是一堆毫无意义的仪式。

现在肯定有好处吧？肯定，所有这些并行执行当然一定都很有用，能使游戏运行得更快吧？

不幸的是，在整理系统所需的所有工作完成后，并没有太多可以并行化的余地。实际上，从中可能获得的一点好处将等同于使用`rayon`进行数据并行的纯数据驱动系统的并行化。

回顾多年的游戏开发经验，我在Unity中使用Burst/Jobs编写的并行代码比在Rust游戏中（无论是在Bevy中还是在自定义代码中）都要多得多，这仅仅是因为大部分游戏工作最终都是"游戏"本身，剩下足够的精力来解决有趣的问题。而在几乎每个Rust项目中，我感觉自己的大部分精力都花在与语言斗争，或者围绕语言设计东西，或者至少确保我不会因为某些事情以特定方式完成而失去太多开发体验。

全局状态就是这个类别的一个完美例子，虽然这一部分很长，但我觉得还需要进一步解释一下。让我们首先定义这个问题。在Rust语言中，通常有几种选择：

- `static mut`，这是不安全的，意味着每次使用都需要`unsafe`，这非常丑陋，而且在意外误用的情况下会导致未定义行为。
- `static X: AtomicBool`（或`AtomicUsize`或任何其他支持的类型）...这是一个不错的解决方案，虽然有点烦人，但至少使用起来不太烦人，但只适用于简单的类型。
- `static X: Lazy<AtomicRefCell<T>> = Lazy::new(|| AtomicRefCell::new(T::new()))`...这对于大多数类型来说是必要的，不仅在定义和使用上很烦人，而且由于双重借用，还可能导致运行时崩溃。
- ...当然，或者"只需传递它，不要使用全局状态"

我数不清有多少次是因为对某个东西进行了双重借用而导致意外崩溃。并不是因为代码"一开始就设计得不好"，而是因为代码库中的其他东西强迫我进行了重构，而在我重构时，我还需要重构我的全局状态使用方式，这导致了意外的崩溃。

Rust用户会说，这意味着我的代码做错了什么，并且它实际上为我捕获了一个错误，这是全局状态不好并应该避免的一个很好的例子。这并不完全错误，有些错误可能会发生，并且可以通过此类检查来防止。但实际上，在使用像C#这样具有易于使用的全局状态的语言时，从实际情况和出现的错误类型来看，在游戏开发的背景下，这些问题实际上很少发生。

另一方面，由于动态借用检查，任何与动态借用检查有关的双重借用导致的崩溃都是非常容易发生的，而且通常是出于错误的原因。一个例子是在ECS中对重叠原型进行查询。对于未经了解的人来说，这样的代码在Rust中会成为一个问题（稍微简化以提高可读性）：

```
for (entity, mob) in world.query::<&mut Mob>().iter() {
  if let Some(hit) = physics.overlap_query(mob.position, 2.0) {
    println!("hit a mob: {}", world.get::<&mut Mob>(hit.entity));
  }
}
```

问题在于，我们从两个不同的位置访问了相同的东西。一个更简单的例子是通过执行类似以下操作来迭代对：

```
for mob1 in world.query::<&mut Mob>() {
  for mob2 in world.query::<&Mob>() {
    // ...
  }
}
```

Rust的规则禁止同时拥有对同一对象的两个可变引用，任何可能导致这种情况的操作都是不允许的。在上述情况下，我们将会遇到运行时崩溃。一些ECS解决方案可以解决这个问题，例如在Bevy中，当查询不重叠时，可以进行部分重叠，例如`Query<(Mob, Player)>`和`Query<(Mob, Not<Player>)>`，但这只解决了没有重叠的情况。

我在全局状态的部分提到了这一点，因为当事物变为全局时，这种限制的存在变得特别明显，因为很容易通过一些全局引用意外接触到其他代码库的`RefCell<T>`。再次强调，Rust开发者会说"这很好，你防止了潜在的错误！"但是我要再次说，我认为这种情况并没有真正拯救我，使我免于犯错的先例。在没有这种限制的语言中这样做也不会引发问题。

还有线程的问题。我认为主要的谬误在于Rust游戏开发者认为游戏与后端服务相同，为了性能而必须以异步方式运行。在游戏代码中，人们不得不将事物包装在`Mutex<T>`或`AtomicRefCell<T>`中，不是为了"避免如果他们使用C++编写并忘记同步访问将遇到的问题"，而只是为了满足编译器对"使一切线程安全"的全面要求，即使整个代码库中没有哪怕一个`thread::spawn`。

动态借用检查在重构后导致意外崩溃

在我写这篇文章的时候，我刚刚发现我们的游戏由于重叠的`World::query_mut`而崩溃。我们使用`hecs`已经有大约两年的时间了，这些并不是当你初次使用该库时遇到的那种微不足道的"哦，我不小心嵌套了两个查询，糟糕"的问题。而是代码的一部分是顶层运行的系统执行某些操作，然后代码的另一部分独立地在某个深层次的ECS中进行一些简单的操作，然后通过大规模的重构意外地发生了重叠。

这并不是我第一次遇到这种情况，而常见的解决方案是"你的代码结构不佳，这就是你遇到这些问题的原因，你需要重构并正确设计它"。反驳这样的论点相对困难，因为他们的核心观点并没有错，这种情况的发生是因为代码库的某些部分设计得不够理想。问题在于，这又是一个Rust强制进行重构的情况，而其他语言则不会这样。重叠的原型并不一定是错误的，非Rust的ECS解决方案，如flecs，很乐意允许这样做。

而且，这个问题不仅限于ECS。我们在使用`RefCell<T>`时一次又一次地遇到了两个`.borrow_mut()`重叠并导致意外崩溃。

问题是，这些并不总是因为"糟糕的代码"。人们会说"尽可能短地借用"，以解决这个问题，但这并不是免费的。显然，这又取决于代码的正确结构，但此时我们已经确定了游戏开发与服务器开发并不相同，代码并不总是优化过的。有时候，我们可能需要在循环中使用`RefCell`的某个值，并且在整个循环中延长借用的时间而不仅仅在需要的地方借用。如果循环足够大并调用了可能在内部需要相同`RefCell`的系统，通常还带有一些条件逻辑，这可能立即导致问题。人们可能会再次争论"只需使用间接引用，并通过事件执行条件操作"，但这样我们将权衡的是将游戏逻辑分散在代码库中，而不仅仅是拥有20行明显可读的代码。

在一个完美的世界里，每次重构都会进行测试，每个分支都会被评估，代码流程会被精心设计成线性且自上而下的方式，这样就不会出现这些问题。人们不需要使用`RefCell`，而是会精心设计他们的函数，以便他们可以传递正确的上下文对象，或者只传递所需的参数。

很遗憾，我认为这对于独立游戏开发来说甚至不太现实。花费时间重构可能在两周后就会被移除的功能是浪费时间，这使得`RefCell`成为了部分借用的理想解决方案，否则数据将不得不重新组织为不同形状的上下文结构，或者必须在各处更改函数参数以深入提取正确的参数，或者必须使用间接手段来分离事物。

上下文对象的灵活性不够

由于Rust对程序员有一套相对独特的约束，它最终会产生许多自找麻烦的问题，而这些问题的解决方案在其他语言中并不经常出现。 这个例子中有一个传递的上下文对象。在几乎所有其他编程语言中，引入全局状态并不是一个大问题，可以是全局变量或单例形式。然而，由于上述各种原因，在Rust中这样的事变得非常困难。

首先，人们可能会想到的解决方案是"只需存储以后需要的引用"，但是任何使用过Rust几天的人都会意识到这是不可能的。借用检查器要求对每个引用字段进行生命周期跟踪，因为生命周期变成了泛型，这会影响到该类型的每个使用点，因此无法轻易进行实验。

这里存在不止一个问题，但我觉得有必要更明确地指出这一点，因为对于没有尝试过的人来说可能不太明显。表面上看，似乎可以使用这样的生命周期，例如：

```
struct Thing<'a> {
  x: &'a i32,
}
```

问题是，如果我们现在想要一个 `fn foo(t: &Thing) ...` 当然不行，`Thing` 是泛型的生命周期，所以必须变成 `fn foo<'a>(t: &Thing<'a>)` 或者更糟糕。如果我们尝试在另一个结构体中存储 `Thing`，情况就会变得一样糟：

```
struct Potato<'a> {
  size: f32,
  thing: Thing<'a>,
}
```

尽管 `Potato` 可能并不真正关心 `Thing`，但在Rust中，生命周期必须被极其严肃地对待，我们不能只是忽略它们。事情上，这比他们看起来还要糟糕得多，因为假设你确实走上这条路，并试图弄清楚生命周期的事情。

Rust也不允许未使用的生命周期，所以假设你有：

```
struct Foo<'a> {
    x: &'a i32,
}
```

但是当你重构代码时，你最终想要将其更改为

```
struct Foo<'a> {
    x: i32,
}
```

现在，这当然是完全禁止的，因为你会有一个未使用的生命周期，而我们不能容忍这种情况。这可能看起来非常微小，在某些语言中，甚至在更简单的情况下，这种情况似乎是期望的（参见此处）。但问题是，生命周期常常需要相当多的"问题解决"和"调试"，人们需要尝试一些不同的方法，而尝试使用生命周期通常意味着添加或删除生命周期，而删除生命周期往往意味着"哦，现在这个未使用了，你必须在每个地方都删除它"，从而导致大规模的级联重构。多年来，我尝试过几次这样的方法，老实说，其中最令人恼火的事情之一就是尝试对带有生命周期的非常简单的更改进行迭代，但被迫在每次更改时在10个不同的地方进行更改。

但即使上述情况不成立，在许多情况下，我们也不能只是"存储对某个东西的引用"，因为生命周期并不好用。

Rust 在这里提供的一种替代方案是共享所有权，即 `Rc<T>` 或 `Arc<T>`。当然，这是可行的，但是这种做法受到了严厉的批评。在使用 Rust 一段时间后，我意识到使用这些方法实际上可以节省很多脑细胞，虽然你需要不再向 Rust 朋友们展示你编写的代码，或者至少要隐藏它，并假装它不存在。

不幸的是，仍然有许多情况下，共享所有权只是一个糟糕的解决方案，有时可能是出于性能原因，但有时是因为你只能控制引用而无法控制所有权。

Rust 游戏开发中的第一个技巧是"如果你每帧从上到下传递引用，你所有的生命周期/引用问题都会消失"。这实际上非常有效，并且类似于 React 中自顶向下传递的 props。只有一个问题，那就是现在你需要将 *每个* 需要它的函数中的 *每个* 东西都传递进去。

起初似乎很明显和容易，只要正确设计你的代码，就不会有任何问题，哈哈。或者至少许多人会这样说，特别是"如果你在这方面遇到问题，你的代码就是丑陋/错误/糟糕/面条代码"，你知道的，常见的说法。

幸运的是，这里有一个真正的解决方案，那就是创建一个传递的 *上下文* 结构体，其中包含所有这些引用。这最终会有一个生命周期，但只有一个，并且看起来像这样：

```
struct Context<'a> {
  player: &'a mut Player,
  camera: &'a mut Camera,
  // ...
}
```

每个游戏中的函数现在只需要接受一个简单的 `c: &mut Context`，并获取所需的内容。很棒，对吧？

好吧，只要你不借用任何东西就没问题。想象一下，你想运行一个玩家系统，但同时也想保留相机系统。`player_system` 就像游戏中的其他所有内容一样，需要 `c: &mut Context`，因为你希望保持一致，并避免将10个不同的参数传递给函数。但当你尝试这样做时：

```
let cam = c.camera;

player_system(c);

cam.update();
```

你将会遇到通常的 "无法借用 `c`，因为它已经被借用"，因为我们触及了一个字段，而部分借用规则说如果你触及了某个东西，那么整个东西都被借用了。

无论 `player_system` 是否只访问了 `c.player`，Rust 都不关心内部是什么，它只关心类型，而类型表示它需要 `c`，所以它必须得到 `c`。这个例子可能看起来有点愚蠢，但在更大的项目中，希望在某些地方使用一些字段子集，同时方便地将其余字段传递到其他地方，这种情况变得非常普遍。

幸运的是，Rust 并不完全愚蠢，它允许我们这样做 `player_system(c.player)`，因为部分借用允许我们借用不相交的字段。

此时，借用检查器的捍卫者会说你的上下文对象设计得不对，你应该将它分解成多个上下文对象，或者根据它们的使用情况将你的字段分组，这样就可以利用部分借用。也许所有的相机相关的东西都在一个字段中，所有的玩家相关的东西都在另一个字段中，然后我们只需要将那个字段传递给`player_system`，而不是整个`c`，每个人都会感到满意，对吧?

不幸的是，这属于本文试图解决的问题之一，即我想要做的是开发我的游戏。我制作游戏不是为了享受类型系统并找出最佳的组织结构方式来让编译器满意。当我重新组织我的上下文对象时，从单线程代码的可维护性方面来说，我没有任何收益。我已经做过这件事很多次，我非常确定下一次进行游戏测试并获得对游戏的新建议时，我可能还需要再次更改设计。

这里的问题是，代码的更改并不是因为业务逻辑的变化，而是因为编译器对本质上正确的某些内容表示不满。只因为这可能不符合借用检查器的工作方式，因为它只看类型，但从我们传递所有我们正在使用的字段的角度来看，它是正确的，如果我们这样做，它就可以正常编译。Rust让我们在 `传递7个不同的参数` 或者 `在需要在结构体中移动某些东西时重构我们的结构体` 之间做出选择，而这两种选择都很烦人，浪费时间。

Rust没有一种结构化的类型系统，使得我们可以说“有这些字段的类型”，也没有对这个问题有任何其他的，不需要重新定义结构体和使用它的所有东西的解决方案。它只是强迫程序员做它所谓“正确”的事情。

Rust 的优点

尽管整篇文章对 Rust 非常不满意，但我想列举一些我认为是积极的事情，这些事情在开发过程中真正帮助了我们。

如果编译通过，它通常就能正常工作。这既是一个梗，但在某种程度上也是真的。有很多次我对 "编译驱动的开发" 能走得这么远感到惊讶。Rust 最大的优势是，当你编写适合 Rust 的代码时，事情进展得非常顺利，语言会引导用户走向正确的路径。 从我的角度来看，这里最大的优势是CLI工具、数据处理和算法。我花了相当多的时间在用Rust编写“Python脚本”，也就是通常大多数人会选择使用Python或Bash的小工具，而我选择使用Rust（既为了学习，也为了看看它是否有效），而且很多时候我都会惊讶地发现这实际上是有效的。我绝对不想在C++中做同样的事情。

默认性能优越。由于我们正在转回C#，我开始更深入地研究Rust与C#在更细粒度上的性能，尝试在两种语言之间一对一地匹配特定算法，并尽量使性能更接近。然而，经过一些努力，Rust的性能仍然大致优于C#的1:1.5-2.5倍。对于那些经常参考基准测试的人来说，这可能并不令人意外，但我亲自经历并真正尝试过后，我非常惊喜地发现Rust代码在自然情况下确实非常快。

我要指出的是，Unity的Burst编译器在提高C#性能方面做得相当不错，但我没有足够的A/B数据提供具体数字，只是观察到了C#的显著加速。

话虽如此，在这些年里，我一直对Rust的代码运行情况感到非常满意，即使我经常做一些非常愚蠢的事情，我也很满意。我要注意的是，这一切的前提是在`Cargo.toml`中具有以下设置：

```
[profile.dev]
opt-level = 1
[profile.dev.package."*"]
opt-level = 1
```

因为我看到很多很多人问为什么速度慢，结果发现他们只是在构建调试版本。就像Rust在开启优化时非常快速一样，关闭优化后速度也非常慢。我使用`opt-level = 1`而不是`3`，因为在我的测试中，我没有注意到速度上的差异，但是`3`编译速度稍慢，至少在我测试的代码上是这样的。

枚举类型的实现非常好。所有使用Rust的人可能都知道这一点，我要说的是，随着时间的推移，我倾向于更多地使用动态结构而不是严格使用枚举和模式匹配，但至少对于适合使用枚举的情况，它们非常好用，可能是我使用过的语言中我最喜欢的实现方式。

Rust Analyzer。我不确定是否应该将其列为优点还是缺点。我将其列为优点，因为我现在离了它就已经不会写 Rust了。自从2013年左右开始接触Rust以来，围绕这门语言的工具链已经大大改进，已经变得非常非常有用。

我考虑将其列为缺点的原因是，它仍然是我使用过的一些比较有问题的LSP服务器之一。我理解这是因为Rust是一门复杂的语言。我也与很多人讨论过这个问题，最后我认为我的项目可能有点“被诅咒”（可能是我的错），因为它经常崩溃和不工作（是的，我已经更新了，这种情况已经持续了一年多，涉及多台机器/项目）。但尽管如此，它仍然非常有用，并且对于编写Rust代码非常有帮助。

Traits。虽然我不完全支持完全放弃继承，但我认为Traits系统非常好，非常适合Rust。如果我们能在孤儿规则上放松一些限制，情况将会变得更好。尽管如此，能够使用扩展Traits是我最喜欢的语言特性之一。

总结

自从2021年中期以来，我们基本上在所有的游戏中都使用了Rust。这是当时BITGUN最初作为一个仅使用Godot/GDScript的项目开始的，当我们遇到Godot的寻路问题（无论是性能还是功能方面）时，我开始寻找替代方案，找到了`gdnative`，然后被推荐使用了`godot-rust`。这不是我第一次见到或使用Rust，但这是我第一次在游戏开发中进行严肃的使用，之前只是进行过一些游戏松散项目。

从那时起，Rust成为我用于一切的语言。我对构建自己的渲染器/框架/引擎等事物感到兴奋，早期版本的Comfy应运而生。随之而来的是很多其他的事情，从CPU光线追踪的小型游戏松散原型，到玩弄简单的2D IK，尝试编写物理引擎，实现行为树，实现单线程协程为重点的异步执行器，再到构建模拟类的NANOVOID，最后是我们将发布的第一款也是最后一款使用Comfy开发的游戏Unrelaxing Quacks，该游戏将与本文同时发布在Steam上。 本文在很大程度上受到我们在开发NANOVOID和Unrelaxing Quacks时遇到的困难的启发，因为在最初开发BITGUN时，我们缺乏Rust知识。这些项目还有一个好处，就是让我们多次尝试了Rust的游戏开发生态。我们尝试了使用Bevy，BITGUN是我们首次尝试移植的游戏，而Unrelaxing Quacks则是最后一次。在开发Comfy的两年时间里，渲染器从OpenGL重写为wgpu，又重写为OpenGL，再从OpenGL重写为wgpu。撰写本文时，我已经编程大约20年了，从C++开始，逐渐涉猎各种语言，包括PHP、Java、Ruby、JavaScript、Haskell、Python、Go、C#，并使用Unity、Unreal Engine 4和Godot各制作并发布了一款游戏。我是那种喜欢尝试各种方法的人，只为确保自己没有错过任何东西。从大多数标准来看，我们的游戏可能不是最好的，但我们已经彻底探索了可用的选择，希望找到最好的解决方案。

我说这些是为了消除任何关于没有付出足够努力尝试Rust或本文不是出于无知或未尝试"正确方法"的观点的想法。当有人指出Rust作为一种语言的问题时，我听到的第一个论点是"你只是没有足够的经验来欣赏这一点"，这是一个玩笑。我们反复尝试了更动态和完全静态的方法来进行各种操作。我们尝试了纯ECS，也尝试了非ECS。

对于那些关心Comfy未来的人，这是我对它的看法。

从2D游戏的角度来看，Comfy在很大程度上已经"完成"。这可以从我们正在发布的完整游戏中看出来，我想澄清的是，我们的游戏是针对主分支运行的。如果你的目标是构建类似复杂性和质量的东西，显然你可以做到。

话虽如此，目前Comfy还没有提供一些期望的功能，主要是在自定义着色器和后处理通道方面的改进。还有一个问题是"维护未来"，因为我不会再开发更多的Rust游戏了。

那些在我们的Discord上积极参与的人已经知道计划是将Comfy的渲染器移植到Macroquad上，这意味着完全删除所有与`wgpu`和`winit`相关的代码，而是使用Macroquad进行窗口管理、输入和渲染。我们从中获得了一些原因和好处：

- 很大一部分代码可以被删除，顺便解决了许多奇怪的边缘情况。用户不会从"拥有自定义wgpu渲染器"中获得任何好处，重要的是功能，这不会改变。
- 通过Macroquad，着色器和后处理变得更加灵活，因为Macroquad已经具备了所有必要的功能。
- 支持更多平台，因为Macroquad/Miniquad在生态系统中具有最广泛的覆盖范围，而Comfy目前只能在wgpu支持的设备上运行。
- 稳定的未来，Comfy可以成为一个高级便利层，构建在由现有社区维护的东西之上，这些人知道自己在做什么。

有些人可能会问为什么一开始就不这样做。最初，我们的一些Comfy项目是在Macroquad之上编写的，但是在某个时候，我想要HDR和Bloom效果，而Macroquad不支持。Comfy是通过复制粘贴Macroquad的API并在其基础上进行z轴索引和y轴排序来创建的，并重新实现了整个渲染器。

但最近，Miniquad/Macroquad现在支持`f16`纹理了，这意味着我们可以在不需要自定义渲染器的情况下获得所有这些功能。已经有一个正在进行的移植工作，但由于我们试图及时发布Unrelaxing Quacks，所以进展相对较慢。然而，我计划在发布后继续这项工作，并且考虑到基本功能已经可用，我对移植工作不会过于复杂感到有希望。

最后，我要为我们的最新游戏做个无耻的推广，因为我们已经花了将近一年的时间在这上面，不这样做太愚蠢了 :)

Unrelaxing Quacks是一款速度很快的幸存者游戏。这款游戏让你直接进入行动，不浪费你的时间。多亏了Rust，我们成功地拥有了许多敌人和抛射物，同时实现了出色的性能。

我们还在对核心机制（如移动和射击）进行了大量的努力，以确保一切都"感觉良好"。

如果您喜欢这篇文章并且想要支持我们，请购买这款游戏并在Steam上进行评价。评价可以是正面的也可以是负面的，只要您对游戏的感受是真实的！ 评价对开发者非常有帮助，因为一旦游戏获得10个评价，Steam就会提高游戏的曝光度，无论这些评价是正面还是负面。