Tankle-Adventure设计和实现浅谈

试玩地址

> *Tankle Adventure是一个基于 WASM-4 引擎的2D俯视角过关游戏。游戏的主要玩法是通过操控坦克，躲避敌人的攻击，击败敌人，最终到达终点。游戏的整体风格是像素风格，游戏的画面和音效都是由我自己制作的。*

我是谁

我是布丁大魔王，这个博客的作者。我在计算技术研究所的主要工作是研究分布式系统和云计算。闲暇时刻，我是一个 PL 人（Programming Language 爱好者），我喜欢研究各种编程语言的设计和实现。同时，我也是一个游戏爱好者，喜欢玩各种类型的游戏，是任天堂的忠实粉丝。

为什么会有 Tankle Adventure

我一直在关注 MoonBit，当 MoonBit 公众号发布游戏编程比赛的消息时，我就决定要参加了。我一直想尝试游戏开发，之前用 RPG Maker MV 和 Godot 制作过一些小游戏，但都只做出了一些简单的 demo。WASM-4 对编程的限制很大，但这其实正好是我需要的，它能让我专注于游戏本身的内容，而不是复杂的素材制作和收集。

2024年7月时我用 Rust 编写过一个 WASM-4 的游戏，叫作 Tankle，这是我第一次尝试在 WASM-4 上制作游戏，主要是为了熟悉平台，测试各种接口。因此没有对游戏内容进行太多设计。不久之后这个比赛公布，我立刻将 Tankle 移植到了 MoonBit，基于官方提供的 wasm4 库。这个过程很顺利。后来 MoonBit 问我能否将这个作品作为比赛的样例挂在比赛展示页面，我很愉快地同意了。在国庆假期期间，我对 Tankle 的关卡进行了一些修改，增加了射击和爆炸的音效。从完成度上而言，游戏已经可以发布了，于是就挂在了官网。现在大家都可以在 Code JAM 页面看到这个游戏。

尽管我也可以直接将这个游戏作为我的比赛作品提交，然后把时间花在另一个比赛 MiniMoonBit 的实现上。但是我觉得那样实在有些可惜了。编程语言的设计和实现对于我而言是很熟悉的领域，我曾用 Scheme、Java、Haskell 实现过几门自己的编程语言，熟悉诸多特性的实现方式。但是对于游戏开发，我还有太多想法未曾实现过。所以我想做一款新的游戏，这样我就可以在比赛中学习到更多的东西。

最初我打算制作一款银河恶魔城游戏，就像《空洞骑士》一样，这是我最喜欢的游戏类型之一。事实上，我已经为这个游戏撰写了完整的关卡设计、故事背景，甚至实现了一个 ECS 框架和更完善的碰撞检测代码。但是止步于图片素材的绘制。我不擅长绘画，画一个像模像样的人物花费我很多时间。我仍然希望更专注于 gameplay 的设计，而不是素材的制作。所以我决定放弃这个游戏，转而将游戏类型转换到 Tankle 的俯视角射击。对于这种风格的游戏，画面的重要性较小，关卡的设计和战斗体验更为重要。

制作一个类似《血腥大地》或《吸血鬼幸存者》的游戏是一个很好的选择，省事又耐玩，只要在 Tankle 的基础上增加一些 Rogue 元素就可以了。但就像我反复强调的，我有太多关卡设计上的想法想要实现在游戏中，这些想法只有在一个全新的冒险游戏中才能实现。所以 Tankle Adventure 就诞生了。

名字是什么意思

没有什么含义。玩过 Tankle 的朋友很容易联想到 FC 游戏 Battle City，中文翻译《坦克大战》。这个游戏对于我的童年有很大的影响，Tankle 基本上是对 Battle City 的致敬，正如 Tankle Adventure 是对 Jackal（中文名《赤色要塞》）的致敬。之所以叫 Tankle 是因为我不想直接管一款游戏叫 Tank，那样过于没有辨识度了。所以我在 Tank 后面加了一个 le，这样就变成了一个新的单词，而且还有一种可爱的感觉。翻译成中文就是《坦克儿》，你可以理解为这就是主角的名字。

自然的，Tankle Adventure 就是 Tankle 的冒险。这个名字也是我在制作游戏时想到的，没有什么特别的含义。

关卡设计体现在哪里

Tankle Adventure 是一款完成度很高的游戏（相对于其它 WASM-4 项目而言），我几乎用尽了 64KB 的 ROM 和 64KB 的 wasm 代码体积。但即便如此，这个游戏的关卡也只完成了我最初设想的50%左右。

我原本计划制作5个关卡：丛林、山道、工厂、码头、天堂号战舰。由于时间和空间（代码体积）关系，最终呈现出的关卡是四个：丛林、山洞、隧道、沼泽。而且四个关卡的长度都非常短，完全无法和我最初设想的关卡相比。这是我最大的遗憾。

因此，我想表达的关卡设计内容就被紧紧塞进了这么四个关卡构建的很小的区域内，这使得玩起来感觉很紧凑，也很短暂。有些支线也不得不因此砍掉。

具体来说，整个游戏的4个关卡并不是独立的，而是在一张连续的大地图上。在丛林，玩家基本上只有一条路可以走，在这里熟悉各种敌人和地形，而且能捡到第一个强化道具装甲包。

通过 Boss 站进入沼泽地后，眼前有三条路可以选择：向下进入山洞、向右下进入隧道、向右进入沼泽深处。由于向下走是最近的道路，这条路是推荐行进的，也就是可以让玩家获得强化、但非必要的岔路。而隧道此时被反向传送带挡住了入口，无法进入。

进入山洞需要通过一个正向传送带，玩家进去之后就出不来了，被困在了里面，被逼迫和两座炮塔战斗。此时玩家会知道自己的处境，但是也只能继续向下，遭遇蜘蛛坦克 Boss 站。这个 Boss 会周期性发出噪声，此时玩家的移动速度会减慢几秒。经过这个 Boss 战，玩家可以获得一个至关重要的强化道具氮气。获得氮气后可以进行冲刺，冲刺可以灵活地躲避敌人的攻击，也可以快速穿过一些狭窄的地形。获得氮气后玩家马上就能学会氮气的用法，因为不用它就无法反向通过传送带回到沼泽地入口。

回到沼泽地后，玩家仍然面临两个选择：向右下进入隧道或者向右进入沼泽深处。自然，这两个选择都是可以通向最终 Boss 的。但想要完整体验游戏，隧道是推荐的选择。在这里，关卡的设计意图很明显：玩家刚刚获得氮气冲刺，并学会了反向通过传送带，所以会很自然地选择这个之前无法进入但现在可以进入的隧道。

隧道的主要内容是突破数个激光阵列。玩家在丛林已经见识过激光阵列，但这里更加凶险：第一个激光阵列伴随正向传送带，在这里掉血几乎是必然的；第二个激光阵列伴随反向传送带，玩家的移动速度需要高于传送带的传动速度才能前行，所以这里会捡到强化道具轮胎。最后一个激光阵列伴随着一台射速极快的炮塔。这个部分考验玩家管理血条的能力，选择和炮塔硬碰硬可能不明智，保留更多血量去面对 Boss 是更好的选择。也正因为如此，关底 Boss 的难度是很低的。

通过隧道也可以到达沼泽深处的最终 Boss 战。最终 Boss 是乌龟战车和雄鹰直升机的组合。战胜 Boss 后就能看见通关后的致谢画面。

不同的 Boss 战就是堆叠不同的数值吗

显然不是。每个 Boss 都有自己独特的攻击方式和对应打法。

我在丛林设计双炮塔 Boss 是为了测试玩家躲避子弹、找间隙攻击的能力，这是整个游戏的基础。

蜘蛛坦克的设计则更多考虑了蜘蛛这一形象的特点，它会干扰玩家的移动，会没有规则地跑来跑去。玩家需要进行精确的瞄准，在移动能力被限制时躲避敌人。

眼镜蛇列车炮是为了让玩家学会使用刚刚获得的氮气冲刺能力，和体验升级后的火箭武器的强大之处。

最终 Boss 的设计是为了让玩家在保持血量的情况下，同时面对两个不同的敌人。乌龟血厚，移动很慢；雄鹰血量低，但移动很快，这是整个游戏的高潮。如果玩家集齐了道具，战胜它们应该不成问题。但追求速通，跳过山洞和隧道两关直接来到这里，可能就会有一定的挑战性，但也不至于不可能。我希望把这场 Boss 战的难度控制在这个范围内。

这就是全部了吗

正如我所说的，以上内容只是我最初设想的50%左右。我还有很多想法没有实现。比如

- 蜘蛛的减速效果应当是通过发射特殊子弹蛛网实现，而不是通过尖叫
- 氮气装置和轮胎对通关都有帮助，但是遇见障碍——获取道具——使用道具跨越障碍的链路太短。如果中间过程可以更丰富一点，玩家能体验到更多乐趣
- 隧道中三个道具的防止明显不合理。隧道应该更长，装甲包应当放在后面，轮胎应当经过一番额外的战斗才能获得
- 小体型的敌人太多。一般来说，小体型且会移动的敌人应当是少数，有一个蜘蛛坦克就够了；乌龟和雄鹰应当设计成大体型，这样玩家更容易击中它们

- 丛林不应该是完全线性的；或者说，每个关卡应该都有一些可选的分支路线
- 最终关本来希望是一艘战舰，而最终 Boss 是一个受到攻击后会分裂的敌人
- 氮气冲刺过程应该伴随无敌效果。但关卡体量支撑不了加入这个强大的能力了

很可惜，由于时间和空间限制，这些想法没有实现。即便如此，我仍然对这个游戏感到满意。

遇到过什么困难

主要是程序设计过程中的困难。和基于成熟的工业级引擎来构建游戏不同，WASM-4 只提供了非常基础的 API，这意味着很多功能都需要自己实现。我在 Tankle 的开发过程中解决了部分问题，但是 Tankle Adventure 更复杂，遇到的问题也更多。

在阐述具体问题之前，我必须说明，我几乎没有考虑过性能问题。尽管内存资源和显示屏非常受限，我们的 CPU 一定是够用的。基本上我们不需要为游戏的帧率担心，尽情在每一帧堆叠一些很复杂的逻辑吧。

代码架构

如果写多了编译器或解释器，习惯了那种非常结构化、非常优美的代码架构，在做游戏开发时就会被恶心到。在程序设计层面，一个游戏的所有元素可以归为两类：实体和系统。实体是游戏中的所有对象，比如玩家、敌人、子弹、道具等等。系统是游戏中的所有逻辑，比如碰撞检测、动画效果、敌人 AI 等等。需要考虑清楚的是，实体和系统之间的关系是什么，如何组织它们，如何让它们协同工作。我会从一个很简单的场景开始，考虑如何扩展它，最后说说为什么我选择了现在这样的实现方式。

首先，让我们考虑一个只有坦克的场景。每个坦克都有一个自己的位置坐标和一个 sprite。我们可以用以下代码来表示一个坦克：

```
struct Tank {
    x : Int
    y : Int
    sprite : Sprite
}
```

我们需要考虑的是每一帧，每个坦克都要被渲染到屏幕上。我们可以在 update 函数中填充如下代码：

```
pub fn update() -> Unit {
    for tank in tanks {
        tank.sprite.draw(tank.x, tank.y)
    }
}
```

现在考虑给坦克添加一个移动的功能。我们可以在坦克上添加一个速度属性，然后在 update 函数中更新坦克的位置。这设计到两个层面的扩展：坦克的数据结构和新系统。目前来说，直接向已有的数据结构和系统中添加新的属性和功能是可行的，

接下来，再考虑加入一个新的实体，比如子弹。子弹和坦克有很多相似的地方，都有位置、速度、sprite。或许我们可以用同一个结构体来表示这两种不同的实体：

```
struct Entity {
    pos : Position
    vel : Velocity
    sprite : Sprite
}
```

那么，怎么区分坦克和子弹呢？这只涉及到这两种实体的行为。坦克的移动是受玩家或 AI 控制的，而子弹的移动则是直线运动。为了表示这种区别，在设计 controller system 时，这个系统将只对坦克实体生效。或者，换句话说，这个系统只对那些有 controller 组件的实体生效。现在，我们至少有这样几种实现方式：

方法一：为每个实体添加一个可选的 controller 组件，然后在 controller system 中处理这些实体。

```
struct Entity {
    pos : Position
    vel : Velocity
    sprite : Sprite
    controller : Controller?
}

fn make_tank(pos : Position, controller : Controller) -> Entity {
    ...
}

fn make_bullet(pos : Position, vel : Velocity) -> Entity {
    ...
}

fn control_system(entities : Array[Entity]) -> {
    for entity in entities {
        if entity.controller.map(fn (ctrl) { ... })
    }
}
```

这种写法下，当实体种类变多时，Entity 结构体下会有非常多的 Option 类型的属性。每增加一个新的属性，你都需要在旧的实体初始化代码中添加一个 None.

方法二：区分这两种实体，然后系统面向不同的实体进行处理。

```
struct Tank {
    pos : Position
    vel : Velocity
    sprite : Sprite
    controller : Controller
}

struct Bullet {
    pos : Position
    vel : Velocity
    sprite : Sprite
}

fn control_system(tanks : Array[Tank]) -> {
    for tank in tanks {
        ...
    }
}
```

这种写法下，移动系统要处理两种实体，而控制系统只处理坦克。当实体种类增加时，会出现很多重复的代码。

方法三：认识到系统只作用于组件，所以用 Map 数据结构把实体和组件关联，系统通过 query 的方式获取组件。

```
type Entity Int
struct Components {
    positions : Map[Entity, Position]
    velocities : Map[Entity, Velocity]
    sprites : Map[Entity, Sprite]
    controllers : Map[Entity, Controller]
}

fn move_system(components : Components) -> {
    let query_result : Array[(Pos, Vel)] = components.query()
    for (pos, vel) in query_result {
        ...
    }
}

fn control_system(components : Components) -> {
    let query_result : Array[(Controller, Vel)] = components.query()
    for (ctrl, vel) in query_result {
```

```
            ...
        }
    }
```

这或许是可扩展性最好的架构，也就是 ECS 架构：实体关联组件、系统处理组件。在这种架构下，无论我们要添加实体类型、组件类型还是系统，都可以几乎不修改原有代码。

我最初就是使用了这样一个架构。它在组织我的代码时，这种优美让我感到非常愉悦。但是，当我想要将 ECS 框架完全包装成一个库时，MoonBit 的 trait object 要求对象安全这一点成了一个难题。暂且不去深入这个问题，我只能说，我在这个问题上花费了很多时间，最终采取了一种笨拙的妥协。

然而，更严重的问题出现了。由于每个系统都对不同的组件进行操作，query 函数的诸多泛型实现使得代码体积膨胀得非常大。当我的项目 cart 体积达到 84KB 时，我不得不切换回第一种架构，老老实实写一堆 Option 类型，并且不再奢望使用将架构代码单独构建成一个库。优化后我的代码体积缩减到了 40KB。

素材导入

用 Rust 编写 Tankle 时，我用了一个宏来绘制 sprite. 这很方便，它在编译时就能将字符串转化成 2BPP 字节序列。但 MoonBit 没有这个能力（我并没有怪罪 MoonBit 不支持宏，我认为绝大多数情况下，程序员不应当使用且没有必要使用宏，尽管它们看起来很酷），所以我只能另辟蹊径。

我用 pixel studio 软件绘制了一个 96 * 96 的像素画面，然后调用 wasm4 的 png2src 功能将图片转化为字节序列。我使用了如下模板：

```
{{#sprites}}
let assets : Bytes = b"{{bytes}}"

let assets_stride : Int = {{width}}
{{/sprites}}
```

但是，png2src 生成的 bytes 格式和 MoonBit 定义 bytes 的语法稍有区别，所以我还需要进行一次转换。为了图省事，我直接用了 sed：

```
sed -i "" -e "s/0x\([0-9a-fA-F]\{2\}\)/\\\\x\1/g\" lib/assets.mbt && sed -
i "" -e "s/,//g" lib/assets.mbt
```

然后将这些命令集成到 Makefile 或者 npm script 中，就很容易地将图片转化为 MoonBit 代码了。

碰撞检测

当一个可以被碰撞的实体进行移动时，需要根据其它碰撞物体的形状和位置，来计算最终移动的位置。也就是写出一个这样的函数供使用：

```
struct Collision {
    pos : Position
    shape : Shape
}

fn move_with_collide(object : Collision, velocity : Velocity, colliders :
Array[Collision]) -> Collision {
    ...
}
```

当考虑椭圆、圆形和其它凸多边形，「移动时沿着碰撞体滑动」、「碰撞后推动」等其它运动逻辑时，这个问题的复杂性又将再上升一个台阶。在这个游戏中，我们只考虑矩形和「碰撞即停下」移动逻辑，此时要面临的问题是简单的，不会用到区域划分、法线判断等高端算法。

首先，考虑一维的情况：当一个物体 a 移动时会碰到其它物体吗？假设物体 a 的位置是 p，速度是 v（指一帧移动的位置），那么下一帧应当移动到 p + v. 所以我们可以考虑移动后的碰撞体有没有和其它碰撞体发生重叠，以此来判断会不会发生碰撞。

但是这样是不够的。想象一个速度很大的物体，它在下一帧会移动到很远的距离，这样就会穿过本应发生碰撞的碰撞体，发生「隧穿效应」。在一维情形下，一个简单的处理方法是，在移动物体 a 时将 a 的碰撞体积按速度 v 拉长，成为一个线段，然后判断这个线段和其它碰撞体是否相交。如果相交，那么就发生了碰撞。

我们还要计算发生碰撞后，移动的位置应该在何处。这倒不难，我们只需获取到发生碰撞的目标碰撞体，然后将物体 a 的位置移动到目标碰撞体的边缘即可。

现在考虑二维的情况。物体 a 的碰撞体是一个矩形，而它的速度可能是斜向的，这样我们如果还按刚刚的方法将 a 的碰撞体按 v 拉长，得到的就是一个多边形，后续碰撞检测的逻辑就不能直接套用我们实现的矩形-矩形碰撞检测逻辑了。此时我们可以泛化碰撞检测逻辑，写一个适用于凸多边形的碰撞检测函数。还有另一种办法，也是我采用的办法，那就是将物体 a 的速度分解为两个分量，然后分别进行碰撞检测。这样我们就可以将二维碰撞检测问题转化为两个一维碰撞检测问题。这种做法需要仔细处理发生碰撞后物体的移动位置。

敌人 AI

在射击游戏中，想要实现一个正常工作的 AI 并不困难，困难的是策略类游戏的 AI 设计。

在制作 Tankle 时，我为敌人设计的 AI 非常简单：生成一个随机数，敌人有一定概率前行，有一定概率转弯，有一定概率开火。就这么简单，但很管用。

在 Tankle Adventure 中，普通小型敌人基本上也是这个思路，只不过改成了：在一个周期，例如120帧内，第0帧转向玩家，然后持续向前移动60帧，再站立不动，第60帧再转向玩家，第70帧开火射击。

对于 Boss，每个 Boss 的 AI 不尽相同，根据它们各自的特色进行变化。以下是代码片断：

```
fn get_action_set(ai : Ai) -> ActionSet {
  match ai {
    Rhino =>
      ActionSet::{
        actions: [
          Once(OrientPlayer, 0),
          Last(MoveForward(0.6), 0, 60),
          Once(Stand, 60),
          Once(OrientPlayer, 60),
          Once(ShootForward(40, 0.6, 1), 70),
        ],
        period: 120,
      }
    Turret =>
      {
        actions: [
          Once(OrientPlayer, 0),
          Once(ShootForward(60, 0.8, 1), 10),
          Once(OrientPlayer, 20),
          Once(ShootForward(60, 0.8, 1), 30),
          Once(OrientPlayer, 40),
          Once(ShootForward(60, 0.8, 1), 50),
        ],
        period: 100,
      }
     ...
  }
}
```

动画效果

Tankle 中的所有实体都是静态的。制作 Tankle Adventure 时，我非常想要加入一些动画效果。例如坦克移动时，轮胎应该是能看见滚动的；蜘蛛在移动时几只爪子应该是在摆动的；直升机的桨叶是不停转动的。

实现这些效果并不困难，我只需为每个动作序列填充一系列 sprite，并增加一个递增的 frame_counter 用来选取 sprite 即可。

```
type AnimeSequence Array[Sprite]

enum AnimeSet {
  Car8(~udlr : AnimeSequence, ~diag : AnimeSequence, mut ~frame_counter :
Int)
  ...
}

fn AnimeSet::draw(self : AnimeSet, pos : Pos2D, ori : Orientation) -> Unit
{
  match self {
    Car8(_) as car => {
      if ori.moving {
        car.frame_counter += 1
      }
      let (dir4, diag) = ori.dir.to_direction4()
      let sequence = if diag { car.diag } else { car.udlr }
      let frame = sequence._[car.frame_counter % sequence._.length()]
      frame.draw(pos, dir4)
    }
    ...
  }
}
```

平滑镜头处理

在 Tankle 中，玩家的视角是固定的，但在 Tankle Adventure 中，我希望玩家的视角能够随着坦克的移动而移动。我设计了一个摄像机：

```
let camera : Camera = Camera::new()

struct Camera {
  pos : Pos2D
}

fn set_camera(x : Double, y : Double) -> Unit {
  camera.pos.x = x
  camera.pos.y = y
}
```

`Sprite::draw` 会根据摄像机的位置来调整实体绘制在屏幕上的位置。那么怎么让 camera 跟着玩家的坦克移动呢？最简单的方法当然是直接将 camera 的位置设置为玩家的位置。但这样会使得镜头的移动非常生硬，玩家总在画面中央。此时有几个选择：让摄像机的位置在玩家坦克的前

方一定距离，这样玩家可以看到更远处的敌人；让摄像机的位置在玩家坦克的后方一定距离，这样画面会更加平滑，玩家小幅度移动时摄像机不会移动。

对于射击游戏而言，视野是非常重要的，因此我选择了第一个方案。玩家在移动时，摄像机的「应该位置」在玩家坦克的正前方不远处。然后，随着玩家移动，镜头会缓慢地移动到这个位置。实现过程如下：

```
fn holdcamera_system(_entities : Array[Entity]) -> Unit {
  let supposed = supposed_camera()
  let target = Velocity2D::{
    x: smooth_drag(camera.pos.x, supposed.x),
    y: smooth_drag(camera.pos.y, supposed.y),
  }
  set_camera(target.x, target.y)
}
```

这还是不够。当进入 Boss 战后，摄像机的位置应该固定在 Boss 战场地中央，这样玩家才能意识到此处和其它地方的不同，并且集中注意力和 Boss 进行战斗。因此，我添加了几个摄像机锚点，用于标注 Boss 战的位置：

```
let camera_anchors : Array[Pos2D] = [
  norm_pos(55, -47),
  norm_pos(70, -32),
  norm_pos(100, -34),
  norm_pos(99, -50),
]

let anchor_radius : Double = 72
```

函数 `supposed_camera` 中会对玩家的位置、摄像机当前位置、摄像机锚点位置进行计算，返回摄像机应该在的位置。这样，摄像机就能在玩家移动时平滑地移动，而在 Boss 战时固定在 Boss 位置。

整体感受

非常有趣。代码编写、关卡设计、自己玩自己的游戏，都相当有趣。我在这个过程中收获了很多。

MoonBit 怎么样

MoonBit 是长在我审美上的语言。我曾经在多个平台和人探讨过编程语言的审美问题，我认为这非常重要。好的语言设计应该是兼具正交性、一致性和可组合性。正交性意味着功能丰富，且同一个功能尽可能只有一种对应的特性支持；一致性意味着语言设计中很少有特例；可组合性是根

本，特性的可组合性直接影响由这门语言编写的应用的可组合性。MoonBit 在这三个方面都做得很好，它选择吸收了诸多有用的现代语言特性，同时还足够简单和足够实用。

在现代语言中，我一直将 Rust 作为我的「御用语言」，我在编写小工具的时候第一个会想到的是 Rust. 但 Rust 的生命周期和借用检查给我带来的心智负担过于重了。对于系统编程而言，手动管理内存是必要的，但奇妙的事情是，在非系统编程的领域，几乎没有一门语言的使用体验能跟 Rust 相比：Haskell 和 OCaml 生态太小众、Python 和 Ruby 没有静态类型系统、C++ 是一个丑陋且易出错的 Rust、Swift 离开苹果平台也几乎毫无用处、Go 语言的设计根本没有审美可言。在 MoonBit 进入 Beta 阶段后，它便替代 Rust 成为了我最常来写写小工具的语言。

抛开上面这些笼统和理念化的优势，MoonBit 还有一些切实的优点：

- MoonBit 的编译速度非常快。这对用惯了 Rust 的用户来说非常明显
- MoonBit 编译到 wasm 的体积小到惊人，性能也很优秀。可以说目前 MoonBit 是最适合用来编写 wasm 应用的语言
- 足够年轻，社区氛围良好，发展速度快

要说目前的 MoonBit 有哪些让我不满的地方：

- 我不太喜欢目前这个错误类型和错误处理机制。我的感受是错误类型似乎破坏了我先前提到的语言设计的一致性
- 包管理相关的语法设计我不喜欢。我总觉得这个设计有点怪异，每次调用别的包时都要出轨一会儿，去写一写 JSON
- 目前的接口系统我觉得少了点什么，例如泛型和关联类型。但这个问题很难把握，我想过或许我需要一个像 OCaml 那样强大的模块系统，或者比 Haskell 稍微逊色一点的类型系统，但那样必然会背离让这门语言保持简单的初衷。我也不知道应该怎么设计一个又好用、又让人满意的接口系统

如果以后会有异步编程相关的特性，我希望是 async-await，而不是 goroutine。

了解更多

1. WASM-4
2. MoonBit Code JAM 2024
3. MoonBit
4. 通关流程录像
5. 源代码

（本文不收费）