Visually Turing-Complete Event Action State-Machine

可视化图灵完备事件驱动状态机

WEB3 世界赋予 NFT 以生命

1、概念

21 世纪是信息化爆炸的时代，我们生活中充斥着各种智能设备，移动电话、智能家电、个人电脑、无人机、自动驾驶，仿佛这些高科技产品都和程序相关，这个人均 CPU 占有率几乎是 100%的社会，程序员在总人口的比率确少的可怜，对于一个普通人来说，写代码是遥不可及的事情。根据 2021 年工信部测算，我国的程序员大约在 600~700 万左右，占总人口的 0.46%。让更多的不懂代码和技术细节的人会写程序是这个系统要解决的问题。

本文主要介绍事件驱动的概念，事件驱动与状态机可视化编程，事件驱动在现实程序框架的接入和实际应用。以可视化方式，不需要写代码，只需要有基本的逻辑思维，就可以去实现程序代码相关的工具。在实际生活应用中，该体系框架可以帮助没有程序编程经验的人实现自己想要的逻辑构架。可应用于区块链智能合约、游戏行为控制、单片机机器人控制、工业自动化控制、家用电器用户自定义流程等应用领域。

2、事件

事件这个定义在单片机系统里可以称为传感器硬件中断，在游戏开发领域叫做触发器，在区块链智能合约里叫做交易（Transaction）。即某种情况下，告之计算机系统输入端，某个事情正在发生，并且包含这个事件的一些参数变量。事件可以描述成该状态机的入口函数。

3、动作

即执行一系列操作的程序流程控制指令流程，流程语句可以实现各种根据状态值的流程跳转，程序输出，设备控制等一系列操作。动作可以描述成该状态机的实体函数。比如一个无人机，当陀螺仪传感器发来某边倾斜的事件后，动作模块可以通过指令调整倾斜端的螺旋桨转数变量来维持平衡。

4、状态

该状态机存储的一系列变量，变量分为可读变量、可写变量。可读变量是程序动作指令只

能读取的变量，比如某个无人机当前 GPS 坐标或者陀螺仪矢量参数，该变量由传感器传入系统，并且可供 读取以作为逻辑判断条件。可写变量是程序动作指令可以改写的变量，该变量可以作为一个临时状态或者全局状态，也可以映射为某个控制部件的参数，如某个无人机当前的四个螺旋桨转数。

状态机是维护一系列状态变量的管理器，在一个系统中可以存在无限种状态也可以存在有限种状态，状态机由行为树负责调度，根据若干变量来选择执行某种状态。行为树可以结合 AI 深度学习，在系统初期我们通过各项变量数值来投喂给 AI 系统，用于改变行为树逻辑节点的权重值，将改进行为树执行逻辑来达到更合理的选择。

在我们的系统中，行为树通过可视化界面交互来编写。

5、图灵完备

图灵完备是指机器执行任何其他可编程计算机能够执行计算的能力。一切可计算的问题都能计算，这样的虚拟机或者编程语言就叫图灵完备的。在应用场景中图灵完备需要限制语言，有循环执行语句，判断分支语句等逻辑结构。使用图灵完备的脚本语言，可以在逻辑上做到和其他编程语言兼容，并在理论上能够实现任何其他语言所能实现的逻辑，以及最大限度的复制现实的商业逻辑。在我们的系统中，图灵完备不以脚本代码作为基础，而是通过一种图形化可交互界面来编写。而且写好的通用模块可以被封装、继承、打包输出。

6、可视化

对于所有高级程序语言的类拓补结构，都可以理解为一个树形结构。在面向对象的程序语言中，一个类通常可以分为：属性列表，方法列表。属性本身可以是原始的的 Primitive（int,float,bool,string）,也可以是复杂类型 Complex（struct,class）。然而方法本身作为方法的根节点，其子语句也可以抽象为，赋值、条件、循环、函数调用，等逻辑节点。所以高级程序语言通常可以转换为一个树形结构的拓补图。

7、应用场景

除了以上举例说明的无人机利用陀螺仪事件控制平衡，在生产生活中有各种各样应用场景，以下举几个可应用场景。

7.1、以太坊智能合约

数字自动售货机

也许对于智能合约最恰当的比喻是自动售货机，就像 Nick Szabo 描述的那样。 有了正确的投入，就保证了某些产出。

要从售货机中获取零食：

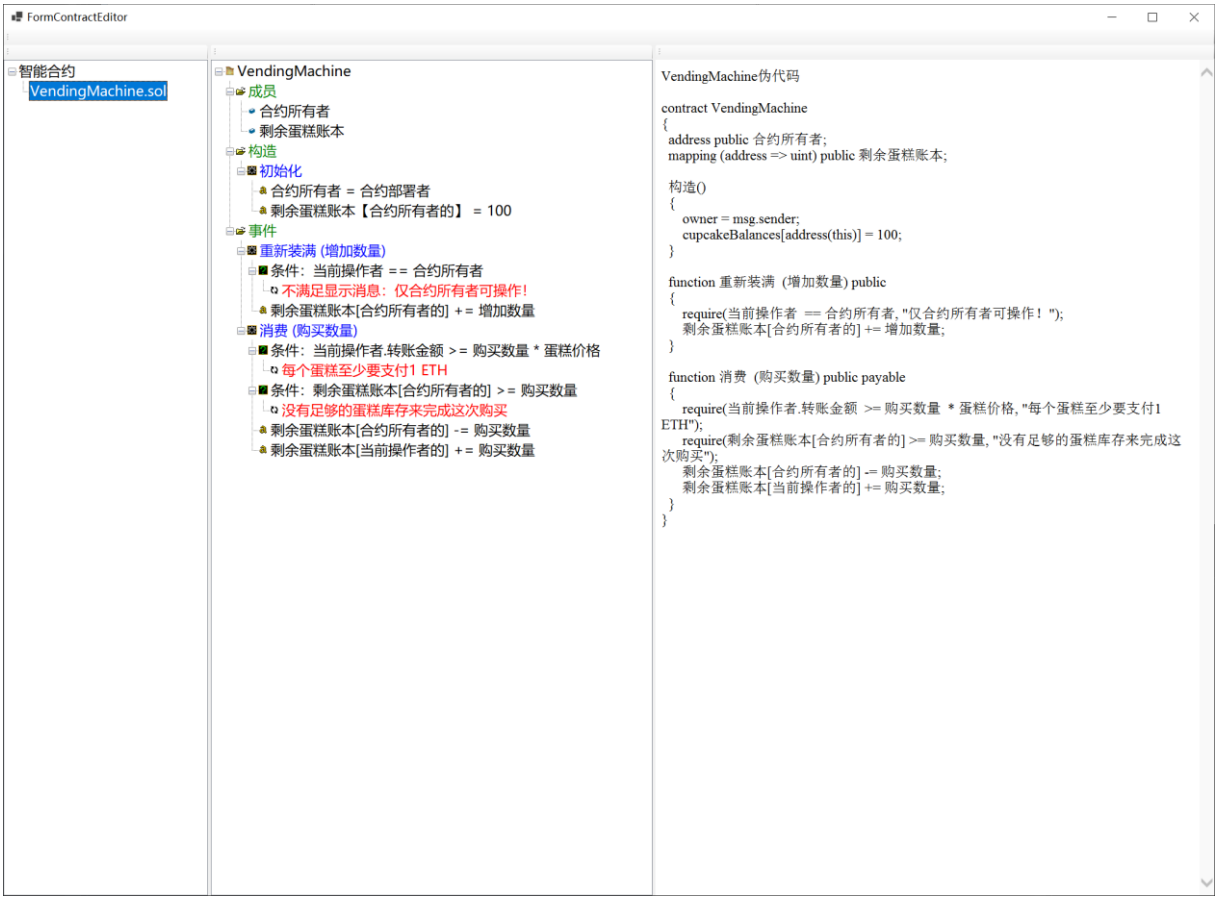
```
    money + snack selection = snack dispensed
```

这种逻辑以程序的形式写入自动售货机。

像自动售货机一样，智能合约也有逻辑编程到其中。 下面的简单示例展示了自动售货机如何看上去如同智能合约一样：

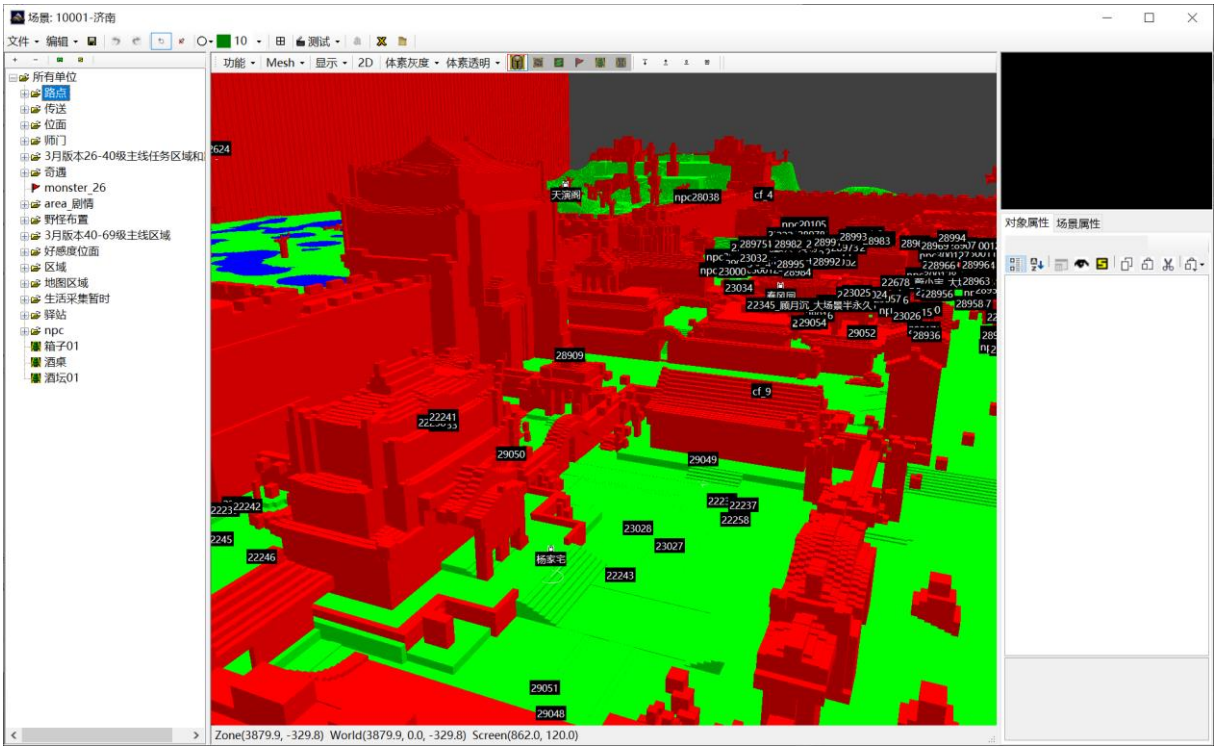
```
pragma solidity 0.8.7;

contract VendingMachine {

    // 合约持有者
    address public owner;
    // 蛋糕账本键值对
    mapping (address => uint) public cupcakeBalances;

    // When 'VendingMachine' contract is deployed:
    // 1. set the deploying address as the owner of the contract
    // 2. set the deployed smart contract's cupcake balance to
100
    constructor() {
        // 合约持有者为合约部署者
        owner = msg.sender;
        // 合约持有者初始蛋糕数量 100
        cupcakeBalances[address(this)] = 100;
    }

    // 允许蛋糕持有者填充蛋糕存量
    function refill(uint amount) public {
        require(msg.sender == owner, "Only the owner can
refill.");
        cupcakeBalances[address(this)] += amount;
    }

    // 允许任何人消费蛋糕
    function purchase(uint amount) public payable {
        require(msg.value >= amount * 1 ether, "You must pay at
least 1 ETH per cupcake");
        require(cupcakeBalances[address(this)] >= amount, "Not
enough cupcakes in stock to complete this purchase");
        cupcakeBalances[address(this)] -= amount;
        cupcakeBalances[msg.sender] += amount;
    }
}
```

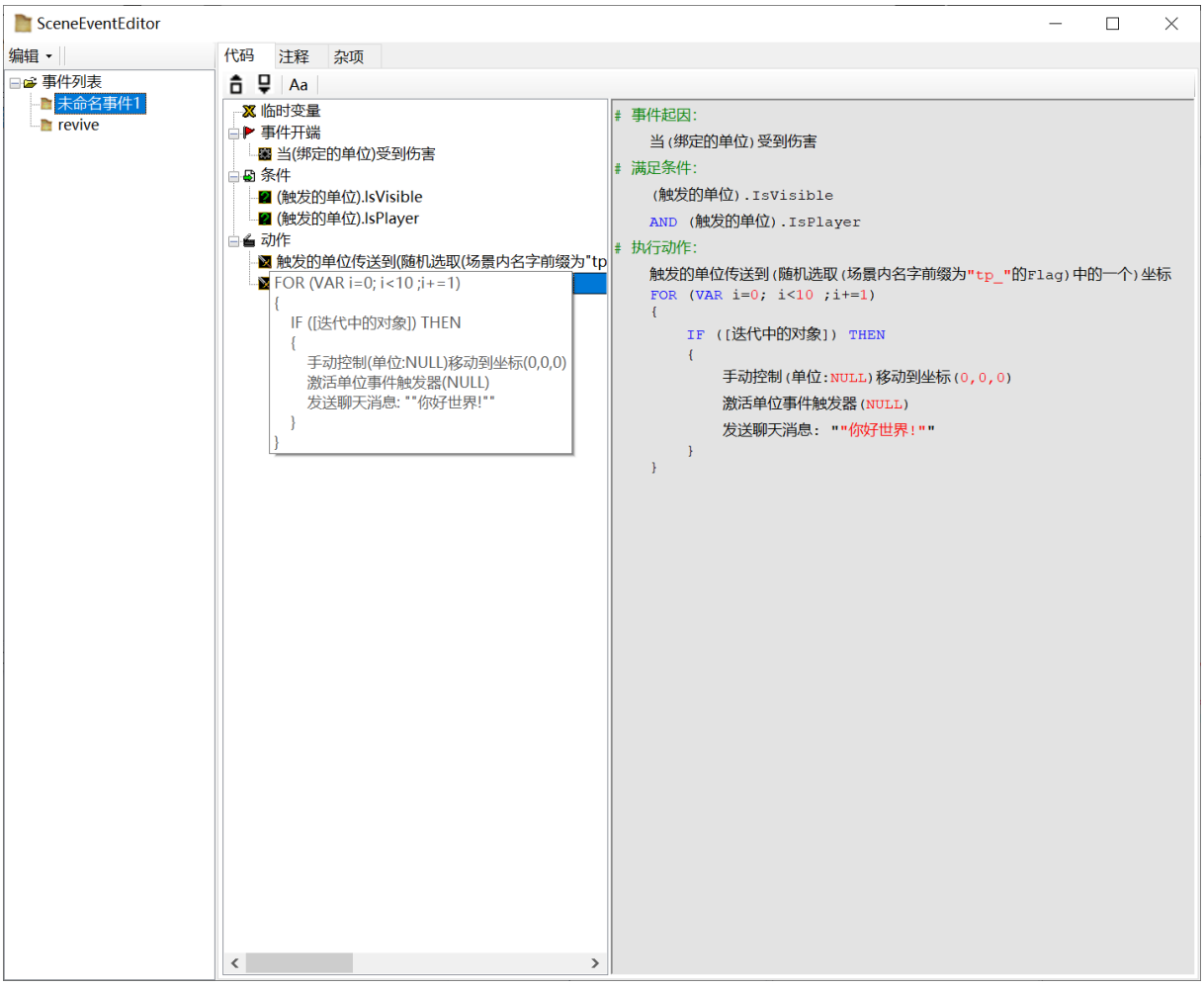
在图形化交互界面，普通用户可以动态的构建，部署，更新区块链智能合约。不用去研究代码怎么写，一键生成智能合约。



7.2、游戏 AI 与事件控制

基于游戏单位状态机，通过事件编辑系统，可对游戏中的单位进行控制。下面为在研发项目游戏核心引擎的截图。


场景: 10001-济南
文件 编辑 测试
功能 Mesh 显示 2D 体素灰度 体素透明
所有单位
路点
传送
位面
帅门
3月版本26-40级主线任务区域和
奇遇
monster_26
area_剧情
野怪布置
3月版本40-69级主线区域
好感度位面
区域
地图区域
生活采集暂时
驿站
npc
箱子01
酒桌
酒坛01
对象属性 场景属性
Zone(3879.9, -329.8) World(3879.9, 0.0, -329.8) Screen(862.0, 120.0)


SceneEventEditor
编辑
事件列表
未命名事件1
revive
代码 注释 杂项
Aa
临时变量
事件开端
当(绑定的单位)受到伤害
条件
(触发的单位).IsVisible
(触发的单位).IsPlayer
动作
触发的单位传送到(随机选取(场景内名字前缀为"tp
FOR (VAR i=0; i<10 ;i+=1)
{
IF ([迭代中的对象]) THEN
{
手动控制(单位:NULL)移动到坐标(0,0,0)
激活单位事件触发器(NULL)
发送聊天消息: ""你好世界!""
}
}
# 事件起因:
当(绑定的单位)受到伤害
满足条件:
(触发的单位).IsVisible
AND (触发的单位).IsPlayer
执行动作:
触发的单位传送到(随机选取(场景内名字前缀为"tp_"的Flag)中的一个)坐标
FOR (VAR i=0; i<10 ;i+=1)
{
IF ([迭代中的对象]) THEN
{
手动控制(单位:NULL)移动到坐标(0,0,0)
激活单位事件触发器(NULL)
发送聊天消息: ""你好世界!""
}
}

8、社会价值

提前发掘社会中有计算机天赋的人才，让大学刚毕业甚至再读学生们，通过一个平台+工具，迅速掌握商业软件开发技能。官方提供大量模板可以帮助开发者迅速搭建自己的程序流程。

结合 NFT 市场，使得 NFT 不再仅仅只是一张静态图片，NFT 融合到游戏场景中，可以通过可视化编程工具，让普通用户赋予 NFT 以生命。

Distributed Meta-Service Scene

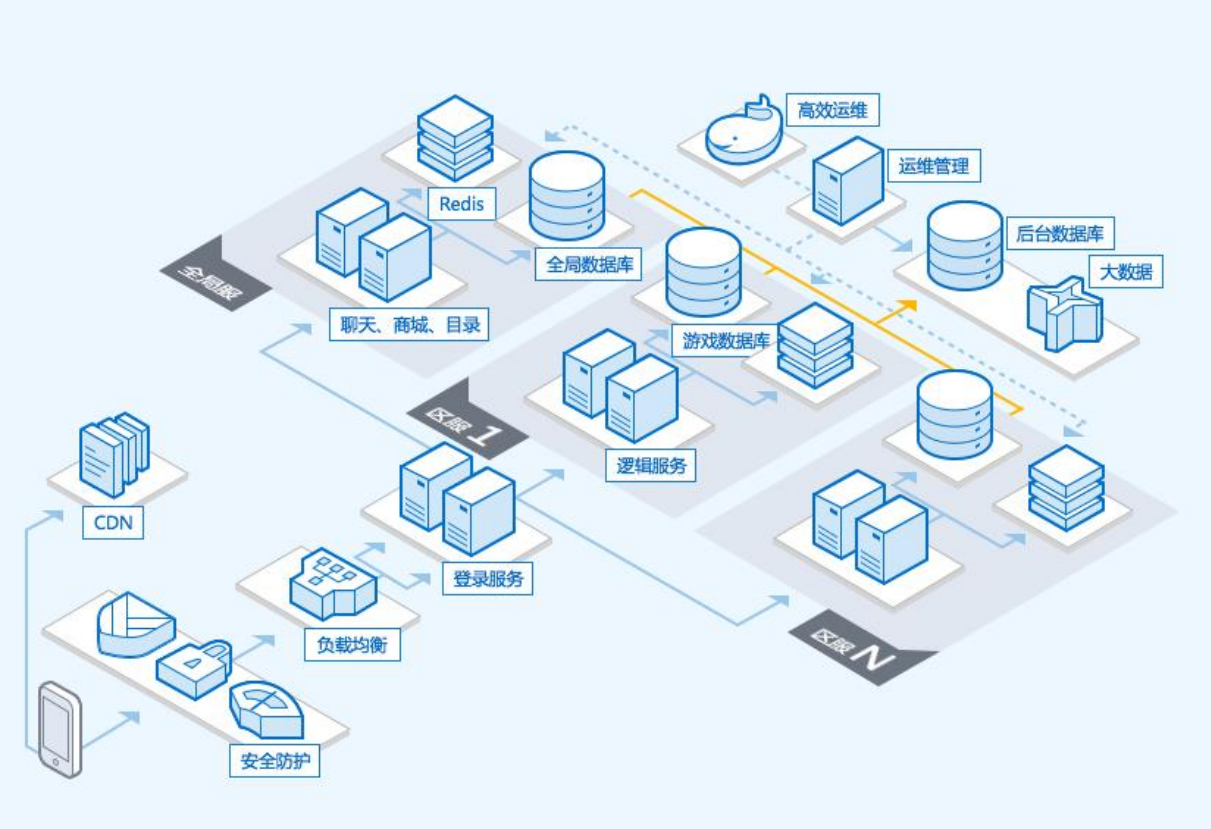
分布式元服务场景

WEB3 世界基础架构

为 WEB3 打造轻量级去中心化服务解决方案，为 NFT 提供远程服务器支持。微服务编程依旧按照可视化图灵完备事件驱动的思路，通过界面来组织程序流程。

服务端引擎

没开发过游戏的人会觉得游戏服务器是很神秘的东西。但事实上它并不比 web 服务器复杂，无非是给客户端提供网络请求服务，本质上它只是基于长连接的 socket 服务器。当然在逻辑复杂性、消息量、实时性方面有更高的要求，下面从 web 服务器与游戏服务器的对比中来说明游戏服务器的一些特点：



A. 复杂的 socket 服务器

如果说 web 服务器的本质是 http 服务器，那么游戏服务器的本质就是 socket 服务器。它利用 socket 通讯来实现服务器与客户端之间的交互。事实上有不少游戏是直接基于原生 socket 来开发的。相对于简单的 socket 服务器，它承受着更加繁重的任务：

- 后端承载着极复杂的游戏逻辑。
- 网络流量与消息量巨大，且实时性要求高。
- 通常一台 socket 服务器无法支撑复杂的游戏逻辑，因此往往使用一个服务器集群来提供服务。

B. 长连接和实时响应

web 应用都是基于 request/response 的短连接模式,占用的资源要比一直 hold 长连接的游

戏服务器要少很多，因此 web 应用可以使用基于 http 的短连接来达到最大的可扩展性，Web 应用能使用短连接模式的原因如下：

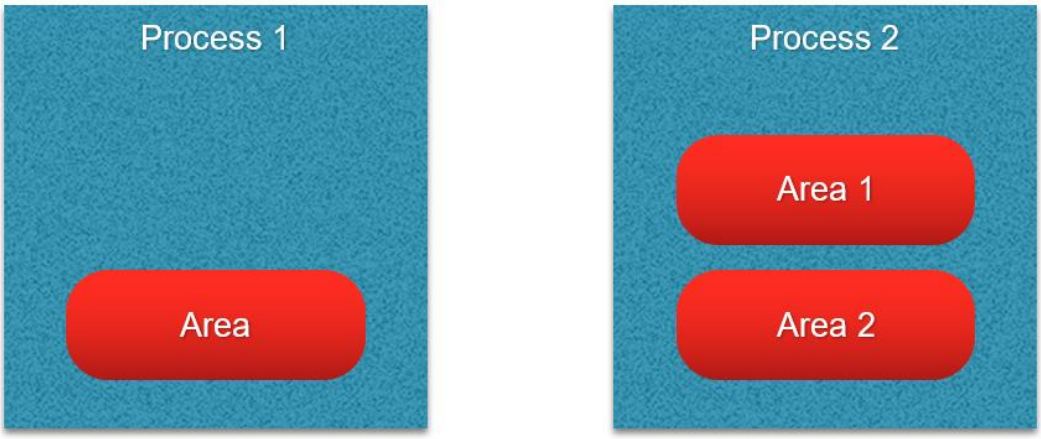
- 通讯的单向性，普通 web 应用一般只有拉模式
- 响应的实时性要求不高，一般 web 应用的响应时间在 3 秒以内都算响应比较及时的。

而游戏应用只能使用长连接，原因如下：

- 通讯的双向性，游戏应用不仅仅是推拉模式，而且推送的数据量要远远大于拉的数据量
- 响应的实时性要求极高，一般游戏应用要求推送的消息实时反应，而实时响应的最大时间是 100 毫秒。

C. 分区策略与负载均衡

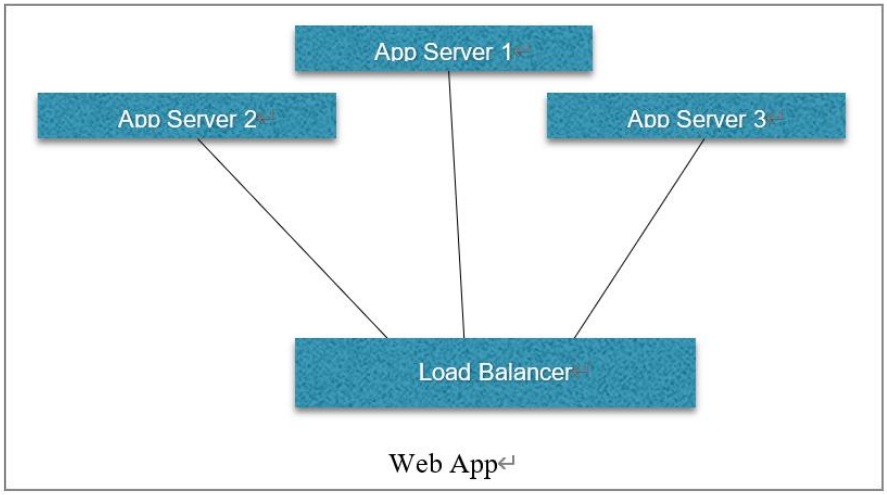
普通的 web 应用在交互上没有相邻性的概念，所有用户之间的交互都是平等，交互频率也不受地域限制。而游戏则不然，游戏交互跟玩家所在地图（场景）上的位置关系非常大，如两个玩家在相邻的地方可以互相 PK 或组队打怪。这种相邻的交互频率非常高，对实时性的要求也非常高，这就必须要求相邻玩家在分布在同一个进程里。于是就有了按场景分区的策略，如图所示： 普通的 web 应用在交互上没有相邻性的概念，所有用户之间的交互都是平等，交互频率也不受地域限制。 而游戏则不然，游戏交互跟玩家所在地图（场景）上的位置关系非常大，如两个玩家在相邻的地方可以互相 PK 或组队打怪。这种相邻的交互频率非常高，对实时性的要求也非常高，这就必须要求相邻玩家在分布在同一个进程里。于是就有了按场景分区的策略，如图所示：

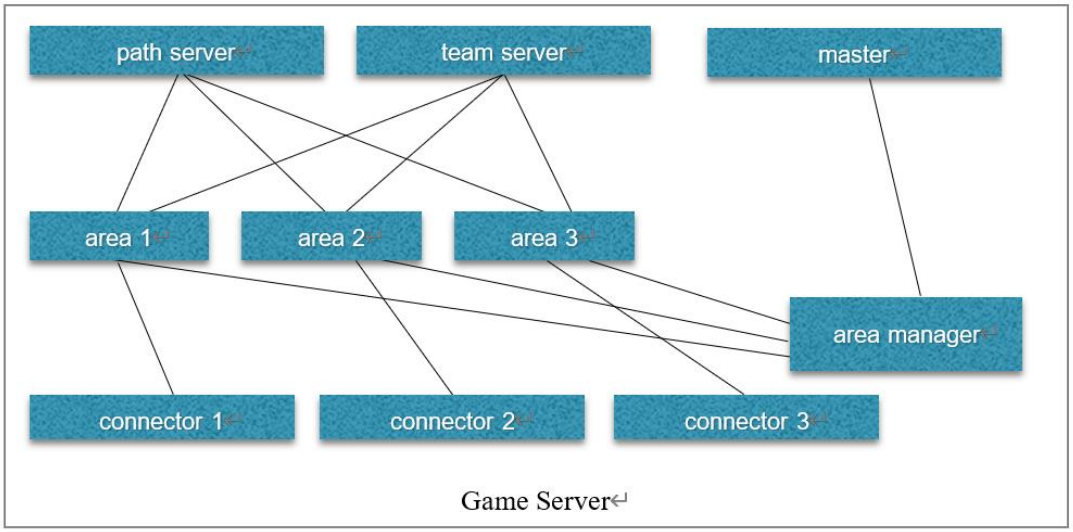


一个进程里可以有一个场景，也可以有多个场景。这种实现带来了游戏的可伸缩性受到场景进程的限制，如果某个场景过于烦忙可能会把进程撑爆，也就把整个游戏撑爆。场景服务器是有状态的，每个用户请求必须发回原来的场景服务器。服务器的有状态带来一系列的问题：场景进程的可伸缩，高可用性等都比不上 web 服务器。目前只能通过游戏服务器的隔离来缓解这些问题。web 应用的分区可以根据负载均衡自由决定， 而游戏则是基于场景(area)的分区模式， 这使同场景的玩家跑在一个进程内， 以达到最少的跨进程调用。

D. 可伸缩性与分布式开发

不管是 web 应用还是游戏服务器，可伸缩性始终是最重要的指标，也是最棘手的问题，它涉及到系统运行架构的搭建，各种优化策略。只有把可伸缩性设计好了，游戏的规模、同时在线人数、响应时间等参数才能得到保证。最初的网络服务器是单进程的架构，所有的逻辑都在单台服务器内完成，这对于同时在线要求不高的游戏是可以这么做的。由于同时在线人数的上升，单服务器的可伸缩性必然受到挑战。随着网络游戏对可伸缩性要求 的增加，分布式是必然的趋势的。 下面是一个 web 服务器和游戏服务器架构对比的示意图:





可以看到由于 web 服务器的无状态性，只需要通过前端的负载均衡器可以导向任意一个进程，因此运行架构相对简单，而且很少需要分布式开发。而游戏服务器是蜘蛛网式的架构，每个进程都有各自的职责，这些进程的交织在一起共同完成一件任务。因此游戏服务器是一个标准的分布式开发架构。

服务器引擎开发难点

从上面的分析可知，游戏服务器是蜘蛛网式的架构，每个进程都有各自的职责，这些进程的交织在一起共同完成一件任务。这些需求也决定了游戏服务器开发的难度。这些难题有：

A. 实时性保证

对实时游戏服务器来说，常见的实时性很高的任务有：

- **实时 Tick**

实时游戏的服务端一般都需要一个定时 tick 来执行定时任务，为了游戏的实时性，一般要求这个 tick 时间在 100ms 之内。这些任务一般包括以下逻辑:

遍历场景中的实体(包括玩家、怪物等)，进行定时操作，如移动、复活、消失等逻辑。

定期补充场景中被杀掉的怪的数量。

定期执行 AI 操作，如怪物的攻击、逃跑等逻辑。

由于实时 100ms 的限制，这个实时 tick 的执行时间必须要远少于 100ms。

- **广播**

由于玩家在游戏里的行动要实时地通知场景中的其它玩家，必须通过广播的模式实时发送。这也使游戏在网络通信上的要求高于 web 应用。 游戏中广播的代价是非常大的。玩家的输入与输出是不对等的，玩家自己简单地动一下，就需要将这个消息实时推送给所有看到这个玩家的其他玩家。假如场景里面人较少，广播发送的消息数还不多，但如果人数达到很密集的程度，则广播的频度将呈平方级增长。如图所示：

B. 分布式开发

几乎在很多书、演讲和文章中都可以看到这样的观点： 分布式开发是很难的，分布式开发的难点主要有：

- **多进程（服务器）的管理**

通常的游戏服务器要由很多进程共同去完成任务。当这些进程交织在一起的时候，多进程的管理并不那么容易。如果没有统一的抽象与管理，光把这些开发环境的进程启动起来就是非常复杂的工作，进程的启动与重启就将严重影响开发效率。重量级的进程消耗大量的机器资源，普通的开发机支撑不了那么多进程，可能一个人的开发环境就需要多台机器。多进程间的调试并不容易，我们发现一个 bug 就要跨好几个进程。

- **RPC 调用**

RPC 调用的解决方案已经有 n 多年的历史了，但 rpc 在分布式开发效率上仍然没有明显提升。 以当前最流行的开发框架 thrift 为例，它在调用代码前需要经过以下步骤：写一个.thrift 文件从.thrift 文件生成源码 thrift --gen <language> <thrift_filename>

在程序中使用生成的源码如果发生接口改动，我们又需要重新修改描述文件，重新生成

stub 接口。对于接口不稳定的开发环境， 这种方式对开发效率影响较大。要想让 rpc 调用的开发达到最简，不需生成 stub 接口， 无需描述文件，我们需要一种很巧妙的方法。

- **分布式事务、异步化操作**

尽管我们尽量把逻辑放在一个进程里处理，但分布式事务仍然是不可避免的。两阶段提交的代码，异步化的操作在普通的开发语言里并不是容易的事。

- **负载均衡，高可用**

由于游戏服务器的有状态性，很多请求需要通过特定的路由规则导到某台服务器；对于有些无状态的服务器，我们则可以把请求路由到负载最低的服务器。通常对于无状态的服务器，高可用是比较好做的。对于有状态的服务器，要做高可用会非常困难，但也不是完全没有办法，常见的两招：（1）将状态引出到外存；例如 redis， 这样进程本身就可以无状态了。但由于所有的操作都通过 redis 可能带来性能损耗，有些场景是不能承受这些损耗的。（2）通过进程互备；将状态通过日志等方式同步到另一进程， 但这可能存在着瞬间数据丢失的问题，这种数据丢失在一些应用场景可能毫无问题，但在另外一些应用场景可能引起严重的数据不一致。

C. 原生 socket 开发的问题

除了上述所讲的分布式开发方面存在的难点外，使用原生的 socket 开发也会有很多问题：

- **抽象程度**

原生的 socket 抽象程度过低，接口过于底层，很多机制都需要自己封装，如 Session、filter、请求抽象、广播等机制都要自已实现，工作量很大，容易出错，且有很多的重复劳动。

- **可伸缩性**

高可伸缩性需考虑很多问题，消息密度、存储策略、进程架构等因素都需要考虑。用原生的 socket 要达到高可伸缩性，需要在架构上花费大量的功夫，而且效果也未必能达到开源框架的水准。

- **服务端的监控管理**

很多服务器的数据需要监控，例如消息密度、在线人数、机器压力、网络压力等，如果采用原生 socket，所有这些都要自己开发，代价很大。

微服务概念

微服务是系统或应用程序中的自包含独立组件。每个微服务都应该有明确的作用域和责任，理想情况下，一个微服务只做一件事。它应该是无状态的或有状态的，如果它是有状态

的，它应该带有自己的持久层（即数据库），不与其他服务共享。软件开发团队基于微服务架构以更分散的方式开发可重用的独立组件。他们可以为每个微服务使用自定义框架、依赖关系集，甚至是完全不同的编程语言。微服务也有助于实现可扩展性，因为它们本质上是分布式的，并且每个微服务都可以独立增长或复制。

容器和微服务

容器是在操作系统中建立隔离上下文的一种方法。实际上，这意味着它们中的每一个都有一个单独的包含了一组已安装的软件和相关配置的虚拟文件系统。由于它们是相互隔离的，因此任何容器都不能直接访问或影响其他容器或底层宿主操作系统。

创建容器的能力已经成为 Linux 操作系统的一部分，这种能力已经存在了很长一段时间，但直到 2013 年 Docker 的推出，容器才成为一种流行的技术。

当我们在谈论定义时，需要注意的是微服务和容器其实是不一样的东西，但这两个概念经常被放在一起谈论，就像 API 和微服务一样。如果没有容器，要么把服务器配置成可以运行多个微服务，让这些微服务不可避免地相互产生负面干扰，要么每个微服务都需要一个单独的服务器或自己的虚拟机，导致不必要的开销。因此，微服务通常被部署在一组由容器集群软件（如 Kubernetes）管理的一组容器中。可以肯定地说，容器和微服务的崛起其实是相互影响、相互促进的结果。

微服务之间的通信

基于微服务架构构建的应用程序或 API 不仅要把自己完全暴露出来，还需要在内部组件（微服务）之间建立连接。由于每个微服务都可以使用不同的编程语言实现，我们需要依赖标准协议（如 HTTP）来建立微服务之间的连接。这个时候我们就回到了 API 上。

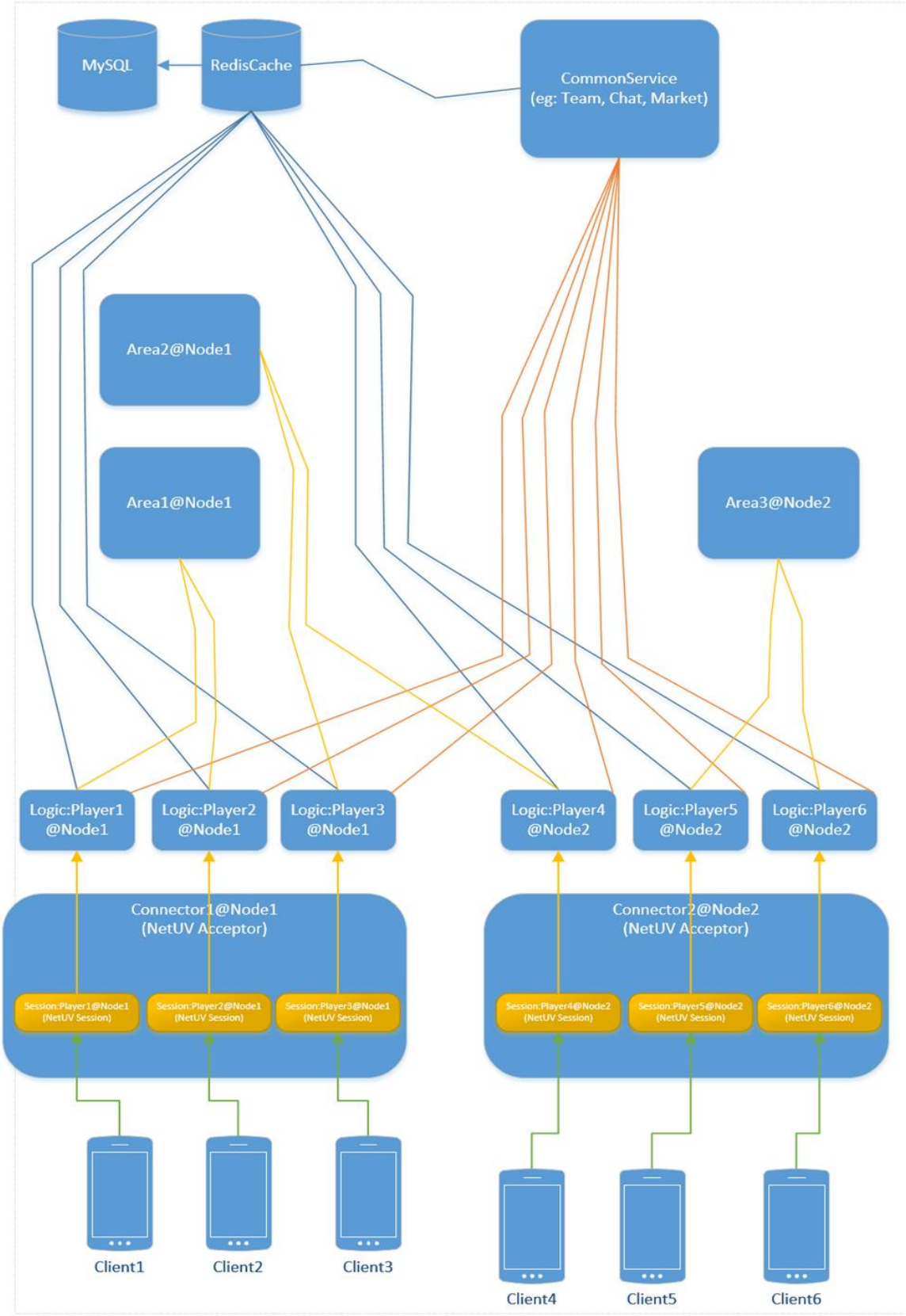
最基本的形式是每个微服务都公开一个 API，让其他服务可以向这个 API 发出请求并获取数据。也可以使用其他不同的方法，比如消息队列。微服务 API 是私有 API，仅限用在单个应用程序中。它通常不提供公共 URL，而是使用组织内部专用网络的私有 IP 或主机名，甚至是单个服务器集群内的 IP 或主机名。不过，这些 API 可以遵循类似公共 API 那样的设计范式或协议。而且，尽管它们的消费者数量有限，也应该遵循开发者体验的基本规则。也就是说，它们应该拥有相关的、一致的、可演化的 API 设计和文档，让其他团队（甚至是你自己）知道如何使用这些微服务。因此，你可以而且应该使用类似的工具来创建你的微服务 API。

当然，与更面向外部的 API 相比，在设计微服务 API 时有不同的侧重点。

微服务和 API 是不同的东西，就像微服务和容器也不是同一种东西一样。不过，这两个概念以两种不同的方式协同工作：首先，微服务可以作为部署内部、合作伙伴或公共 API 后端的一种方法。其次，微服务通常依赖 API 作为与语言无关的通信手段，以便在内部网络中相互通信。开发团队可以使用相似的方法和工具来创建公开 API 和微服务 API。

常见的微服务组件及概念

1. **服务注册**，服务提供方将自己调用地址注册到服务注册中心，让服务调用方能够方便地找到自己。
2. **服务发现**，服务调用方从服务注册中心找到自己需要调用的服务的地址。
3. **负载均衡**，服务提供方一般以多实例的形式提供服务，负载均衡功能能够让服务调用方连接到合适的服务节点。并且，节点选择的工作对服务调用方来说是透明的。
4. **服务网关**,服务网关是服务调用的唯一入口,可以在这个组件是实现用户鉴权、动态路由、灰度发布、A/B 测试、负载限流等功能。
5. **配置中心**，将本地化的配置信息（properties, xml, yaml 等）注册到配置中心，实现程序包在开发、测试、生产环境的无差别性，方便程序包的迁移。
6. **API 管理**，以方便的形式编写及更新 API 文档，并以方便的形式供调用者查看和测试。
7. **集成框架**，微服务组件都以职责单一的程序包对外提供服务，集成框架以配置的形式将所有微服务组件（特别是管理端组件）集成到统一的界面框架下，让用户能够在统一的界面中使用系统。
8. **分布式事务**，对于重要的业务，需要通过分布式事务技术（TCC、高可用消息服务、最大努力通知）保证数据的一致性。
9. **调用链**，记录完成一个业务逻辑时调用到的微服务，并将这种串行或并行的调用关系展示出来。在系统出错时，可以方便地找到出错点。
10. **支撑平台**，系统微服务化后，系统变得更加碎片化，系统的部署、运维、监控等都比单体架构更加复杂，那么，就需要将大部分的工作自动化。现在，可以通过 Docker 等工具来中和这些微服务架构带来的弊端。 例如持续集成、蓝绿发布、健康检查、性能健康等等。严重点，以我们两年的实践经验，可以这么说，如果没有合适的支撑平台或工具，就不要使用微服务架构。


MySQL
RedisCache
CommonService
(eg: Team, Chat, Market)
Area2@Node1
Area1@Node1
Area3@Node2
Logic:Player1
@Node1
Logic:Player2
@Node1
Logic:Player3
@Node1
Logic:Player4
@Node2
Logic:Player5
@Node2
Logic:Player6
@Node2
Connector1@Node1
(NetUV Acceptor)
Connector2@Node2
(NetUV Acceptor)
Session:Player1@Node1
(NetUV Session)
Session:Player2@Node1
(NetUV Session)
Session:Player3@Node1
(NetUV Session)
Session:Player4@Node2
(NetUV Session)
Session:Player5@Node2
(NetUV Session)
Session:Player6@Node2
(NetUV Session)
Client1
Client2
Client3
Client4
Client5
Client6