Blog Archive 2022

2022 年博客文章合集

目录

Oblivion - A Vitepress Theme

2022-05-18

上一个主题 reco 已经用了不短的时间了，最近关注 **Vitepress**，感觉很不错，于是萌生了自己再做一个主题的想法。

目前的功能

- toc
- Katex
- Tags 和 Collections
- Vitepress 默认主题中支持的几乎全部 Markdown 功能

加入各种功能时遇到的问题

在开发的过程中参考了很多现成主题的写法，包括 Vitepress 的 default-theme、reco、vitepress-blog-zaun 等等。

中间也遇到了好多问题，比如 Vue 动画效果在首次载入时的表现问题、SSR 与客户端代码的区分问题、Katex 构建问题等等，到目前为止这几个问题都得到了解决。

Katex

不得不说，Katex 的加入给我带来了不小的麻烦，自定义标签会在 build 的时候报错：

```
build error:
 TypeError: Invalid value used as weak map key
    at WeakMap.set (<anonymous>)
    at normalizePropsOptions (/Users/dyz/Sites/dyzdyz010.github.io/node_modules/@vue/runtime-core/dist/runtime-core.cjs.prod.js:2755:15
)
    at createComponentInstance (/Users/dyz/Sites/dyzdyz010.github.io/node_modules/@vue/runtime-core/dist/runtime-core.cjs.prod.js:5644:
23)
    at renderComponentVNode (/Users/dyz/Sites/dyzdyz010.github.io/node_modules/@vue/server-renderer/dist/server-renderer.cjs.prod.js:19
4:22)
    at Object.ssrRenderComponent (/Users/dyz/Sites/dyzdyz010.github.io/node_modules/@vue/server-renderer/dist/server-renderer.cjs.prod.
js:626:12)
    at _sfc_ssrRender (/Users/dyz/Sites/dyzdyz010.github.io/.vitepress/.temp/posts_productivity_2020_bulletproof-task-management-priori
ty-formula.md.js:25:24)
    at renderComponentSubTree (/Users/dyz/Sites/dyzdyz010.github.io/node_modules/@vue/server-renderer/dist/server-renderer.cjs.prod.js:
266:13)
    at renderComponentVNode (/Users/dyz/Sites/dyzdyz010.github.io/node_modules/@vue/server-renderer/dist/server-renderer.cjs.prod.js:21
1:16)
    at renderVNode (/Users/dyz/Sites/dyzdyz010.github.io/node_modules/@vue/server-renderer/dist/server-renderer.cjs.prod.js:307:22)
    at renderVNodeChildren (/Users/dyz/Sites/dyzdyz010.github.io/node_modules/@vue/server-renderer/dist/server-renderer.cjs.prod.js:322
:9)
error Command failed with exit code 1.
info Visit https://yarnpkg.com/en/docs/cli/run for documentation about this command.
```

Figure 1: Katex 构建错误

找了好多的资料都没找到点上，最后还是在 Vitepress 自己的配置文件中增加了一项才解决问题。在 .vitepress/config.ts 中加入以下内容：

```
// Katex用到的自定义 Tag 列表。参考：[](https://github.com/KaTeX/KaTeX/blob/main/src/mathMLTree.js#L23)
const mathTags = [
  "math", "annotation", "semantics",
  "mtext", "mn", "mo", "mi", "mspace",
  "mover", "munder", "munderover", "msup", "msub", "msubsup",
  "mfrac", "mroot", "msqrt",
  "mtable", "mtr", "mtd", "mlabeledtr",
  "mrow", "menclose",
  "mstyle", "mpadded", "mphantom", "mglyph"
]

// config 中加入 vue 字段
export default {
  vue: {
    template: {
      compilerOptions: {
          isCustomElement: tag => mathTags.includes(tag)
      }
    }
  },
}
```

参考链接：

1. KaTeX
2. Vitepress

Server-side Rendering(SSR)

Vitepress 使用的是服务端构建，虽然官方文档中说要尽量区分客户端 js 代码和服务端 js 代码，但在实践中因为我之前压根没怎么接触过 Vue，所以对具体的实现方式一头雾水。在查阅大量资料之后，我的解决方案是，如果有只能在客户端执行的 js 代码（比如只有在浏览器中才存在的 window 对象），那我就把他放到 onMounted 方法中去。

例如在 Tags 组件中：

```
<script setup lang="ts">
import { useData } from "vitepress"
import { ref, computed, watchEffect, onMounted } from "vue"
import { getStorageTag, getPostsByTag } from "../helpers/tags.ts"
import { getStoragePage } from "../helpers/pagination.ts"

// 先设定默认值，可在服务端执行
const currentPage = ref(1)
const allPosts = useData().theme.value.posts
const currentTag = ref('')
const postsByTag = computed(() => getPostsByTag(currentTag.value))
const watchFun = ref(() => {})
const show = ref(false)

// 调用浏览器 API，只在客户端执行
onMounted(() => {
  currentPage.value = getStoragePage()
  currentTag.value = getStorageTag()

  watchEffect(() => {
    if (window.location.hash) {
      currentTag.value = window.location.hash.replace('#', '');
    }
  })

  show.value = true
})
</script>
```

这样就避免了构建过程中提示找不到对象的问题。

Transition 动画在第一次载入页面时不生效

在构建成功后运行查看效果时，发现页面中加入的 transition 效果在清除缓存刷新页面时不起作用。后来经过查找资料，我的解决方案是，在每个 transition 元素的子元素上增加 v-if，使其默认为 false 不显示，并在 onMounted 中将其设置为 true 显示：

```
<template>
 <transition appear enter-active-class="transition ease-out duration-300"
    enter-from-class="transform opacity-0 scale-95" enter-to-class="opacity-100 scale-100">
    <h1 v-if="show">Tags</h1>
 </transition>
</template>

<script setup lang="ts">
 const show = ref(false)
 onMounted(() => {
  show.value = true
 })
</script>
```

这样即使是第一次访问页面，动画效果也会存在。

下一步准备增加的功能

目前的功能只能说满足了基本的写作需求，但是类型还不够丰富，还是比较笼统。下一步准备实现更多功能：

1. 时间线 - 这个不用多说，就是仿 Facebook 时间线，给自己回顾的时候用。
2. 贡献热度图 - 就是 Github 贡献图的那个样子，打算自己用 svg 实现一个。

3. 微博 - 方便自己随便记录一些内容用，目前想法是给现有的文章增加一个新类别，结构和正常文章完全一致，后续有了新的想法再说。

给想拿来用的人

如果有人想尝试使用本主题 ,直接将 vitepress-theme-oblivion 的内容复制到自己的仓库里构建发布就可以。也可以参考本人的 Pages 仓库 dyzdyz010.github.io。 有什么问题可以直接提 Issue。

MMO Server From Scratch(0) - Introduction

2022-06-08

引言

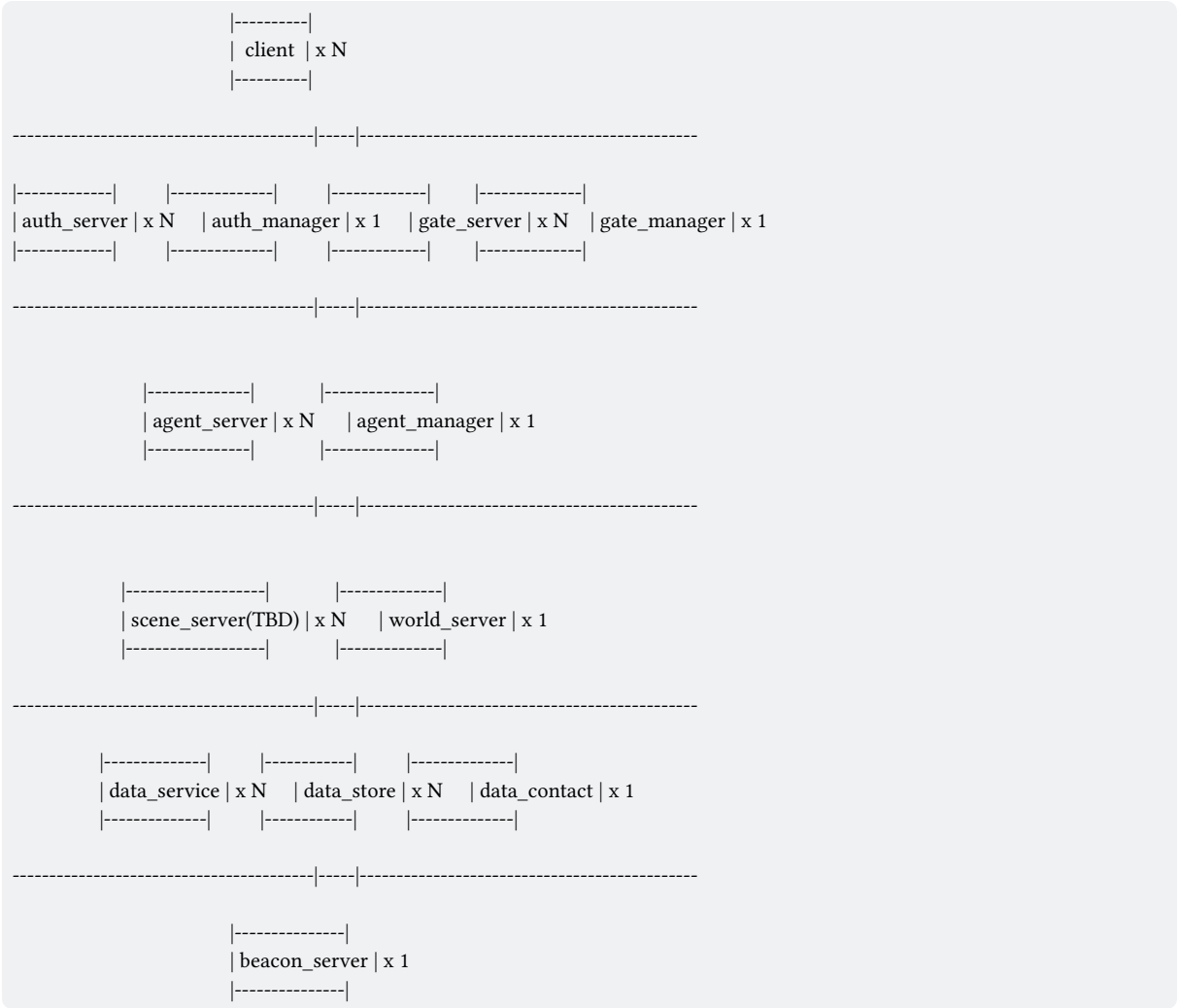
一直想做 MMORPG 游戏，一直在学习，开一个系列把服务器开发的过程记录下来。

一直以来搜索了许多资料 ,但是都支离破碎 ,缺少一个完整详细的概念 ,要么就是只对架构泛泛而谈，要么就是对某个算法的纸上谈兵。我作为纯粹的业外人士 ,这些抽象的资料基本无法对我形成特别重要的指导作用，所以我还是得亲自尝试重构所有内容。

基本构想

查阅过众多资料之后，我对 MMORPG 服务器的整体架构有了一个基本的概念。恰好之前深入了解了一下 Erlang 和 Elixir，它们天然支持高并发的特性以及监督树和尽早崩溃的思想深深打动了我，所以我决定用 Elixir 实现服务器的各个部件。

我的想法中服务器的大致架构如下：



1. auth_server - 用户登录游戏用
2. auth_manager - 用于管理 auth_server，做负载均衡
3. gate_server - 网关服务器，用于接受客户端的 tcp 连接，加解密消息，并转发流量
4. gate_manager - 用于管理 gate_server，做负载均衡
5. agent_server - 用于执行非场景的用户逻辑（待定）
6. agent_manager - 用于管理 agent_server，做负载均衡
7. scene_server - 用于执行场景逻辑，如 AOI、地图区块管理等
8. world_server - 用于统筹管理 scene_server 的分配，以及处理世界级逻辑
9. data_server - 数据库内存存储节点，用来对其他服务器提供服务
10. data_store - 数据库持久化存储节点，用来保存数据库数据
11. data_contact - 数据联系节点，用来联系各个数据库节点，同时为其他服务器分配数据库节点，做负载均衡
12. beacon_server - 灯塔服务器，用于联系各个服务器，交换资源，在节点间建立连接

数据库 打算采用 Mnesia，天然具备集群能力，可以动态横向扩展。

客户端连接打算采用 one process per client，充分利用 BEAM 的轻量级线程能力，提高逻辑编写效率。当然这些都是我猜的，到时候实际性能如何还未可知。

项目地址

https://github.com/dyzdyz010/ex_mmo_cluster

因为找不到同好，只有我一个人在孤军奋战，不知能坚持多久，但是希望能有人一起。且行且珍惜吧。

MMO Server From Scratch(1) - Beacon Server

2022-06-10

今天来实现服务器的第一个部件 - **beacon_server**。

功能解析

为了建立 Elixir 集群，需要所有 Beam 节点在启动之时就已经知道一个固定的节点用来连接，之后 Beam 会自动完成节点之间的链接，即默认的全连接模式，所有节点两两之间均有连接。关于这一点我还没有深入思考过有没有必要进行调整，之后看情况再说

因此，为了让服务器集群内的所有节点在启动时都能够连接一个固定节点从而组成集群，这个固定节点就是 beacon_server。

beacon_server 需要有什么功能呢？在经过一番简单思考后，至少需要具备以下几个功能：

1. 接受其他节点的连接
2. 接受其他节点的注册信息
3. 相应其他节点的需求，返回需求节点的信息

这里有两个重要概念：资源(Resource) 和 需求(Requirement)。资源指某个节点自身的内容类型，也就是在集群中所处的角色，比如网关服务器的资源就是网关(gate_server)；需求指某个节点需要的其他节点，比如网关节点需要网关管理节点**(gate_manager)**来注册自己，数据服务节点需要数据联系节点**(data_contact)**来把数据库同步到自身。

当一个节点向 beacon_server 节点注册时，我们希望它能够向 beacon_server 提供自己的节点名称、资源、需求等数据，方便 beacon_server 在收到别的节点注册时，能够把已经注册过的节点当做需求返回给别的节点。

数据结构

我用一个 GenServer 线程负责上面所说的所有工作，利用线程的 state 来保存来往节点信息。当前粗略想了想，姑且定义信息存储格式如下：

```
%{
  nodes: %{
    "node1@host": :online,
    "node2@host": :offline
  },
  requirements: [
    %{
      module: Module.Interface,
      name: [:requirement_name],
      node: :"node@host"
    }
  ],
  resources: [
    %{
      module: Module.Interface,
      name: :resoutce_name,
      node: :"node@host"
    }
  ]
}
```

我用一个字典存储所有信息，分为 nodes、 requirements 以及 resources 三部分。

nodes 存储所有已经连接的节点和他们的状态，:online 表示在线正常连接，:offline 表示节点断开连接；

requirements 存储每个节点注册时提供的需求信息。使用列表存储，列表中每个项代表一个节点。项使用字典，存储模块(module)、名称(name)、节点(node)信息。其中名称字段，因为有些节点可能会有不只一个需求，因此使用列表存储。模块字段是为了留着以备后用，目前没什么用……节点字段用于获取的节点使用该字段对目标节点发送消息，必不可少。

resources 存储每个节点注册时提供的资源信息，字段与 requirements 完全相同，有一个不同的地方是名称字段的数据类型不再是列表，而是原子，因为每个节点只可能属于唯一的一种资源，不可能属于两种以上，因此用一个单一的原子就可以代表了。

简要实现

建立项目

这是第一个实现，在实现之前，我们先建立一个 umbrella 项目，用来存放之后的所有代码：

```
mix new cluster --umbrella
```

然后创建本节的 beacon_server 项目：

```
cd apps/
mix new beacon_server --sup
```

--sup 用来生成监督树。

有了项目之后，我们需要建立一个 GenServer，用来充当其他节点用来通信的接口，我们就把他叫做 Beacon 好了。

功能函数

根据前面的设想，我们需要下面这么几个函数：

- register(credentials, state) - 用于把注册来的节点信息记录在 state 中，并将新的 state 返回。
- get_requirements(node, requirements, resources) - 用于向已注册的节点返回其需求。

下面贴上我粗略实现的代码，当然这不会是最终版本，未来还有优化的空间：

```
@spec register({node(), module(), atom(), [atom()]}, map()) :: {:ok, map()}
defp register(
       {node, module, resource, requirement},
       state = %{nodes: connected_nodes, resources: resources, requirements: requirements}
     ) do
 Logger.debug("Register: #{node} | #{resource} | #{inspect(requirement)}")

 {:ok,
  %{
    state
    | nodes: add_node(node, connected_nodes),
      resources: add_resource(node, module, resource, resources),
      requirements:
        if requirement != [] do
          add_requirement(node, module, requirement, requirements)
        else
          requirements
        end
  }
 }
end

@spec get_requirements(node(), list(map()), list(map())) :: list(map())
defp get_requirements(node, requirements, resources) do
 req = find_requirements(node, requirements)
 offer = find_resources(req, resources)
 offer
end
```

上面代码中用到的其他私有函数我就不贴了，总之就是利用线程 state 中的数据返回新的数据。

除了这两个必要的函数，我还想添加两个能够监控节点通断的函数。这两个函数通过 handle_info 实现。首先需要在线程初始化的时候开启这项功能：

```
:net_kernel.monitor_nodes(true)
```

之后实现两个 callback：

```
# ========== Node monitoring ==========

@impl true
def handle_info({:nodeup, node}, state) do
 Logger.debug("Node connected: #{node}")

 {:noreply, state}
end

@impl true
def handle_info({:nodedown, node}, state = %{nodes: node_list}) do
 Logger.critical("Node disconnected: #{node}")

 {:noreply, %{state | nodes: %{node_list | node => :offline}}}
end
```

不在 :nodeup 回调中将节点状态修改为 :online 是因为节点在注册的时候，注册函数已经将节点的状态修改为 :online 了。

接口函数

有了功能之后，还需要提供对外接口，GenServer 已经提供了相关的回调函数供我们实现，在这里我使用 handle_call/3，因为注册流程需要是同步的，只有注册完成之后对应节点才能开始正常运行。

同样地，对外接口也是两个，分别是 :register 和 :get_requirements ：

```
@impl true
# Register node with resource and requirement.
def handle_call(
    {:register, credentials},
    _from,
    state
  ) do
 Logger.info("New register from #{inspect(credentials, pretty: true)}.")

 {:ok, new_state} = register(credentials, state)

 Logger.info("Register #{inspect(credentials, pretty: true)} complete.", ansi_color: :green)

 {:reply, :ok, new_state}
end

@impl true
# Reply to caller node with specified requirements
def handle_call(
    {:get_requirements, node},
    _from,
    state = %{nodes: _, resources: resources, requirements: requirements}
  ) do
 Logger.debug("Getting requirements for #{inspect(node)}")

 offer = get_requirements(node, requirements, resources)

 {:reply,
  case length(offer) do
   0 -> nil
   _ ->
    Logger.info("Requirements retrieved: #{inspect(offer, pretty: true)}", ansi_color: :green)
    {:ok, offer}
  end, state}
end
```

至此 ,Beacon 功能模块就基本完整了 ,最后我们需要把它加入到监督树里使其运行起来。在 application.ex 中：

```
def start(_type, _args) do
 children = [
  # Starts a worker by calling: BeaconServer.Worker.start_link(arg)
  {BeaconServer.Beacon, name: BeaconServer.Beacon}
 ]

 # See https://hexdocs.pm/elixir/Supervisor.html
 # for other strategies and supported options
 opts = [strategy: :one_for_one, name: BeaconServer.Supervisor]
 Supervisor.start_link(children, opts)
end
```

像这样把 Beacon 模块加入到监督者的子线程列表中，beacon_server 暂时就算完成了。

效果测试

运行一下试试：

```
iex --name beacon1@127.0.0.1 --cookie mmo -S mix
```

为了让其他节点连接，name 和 cookie 一定好设置好。

我写了点测试代码调用一下试试：

```
iex --name beacon1@127.0.0.1 --vm-args rel/vm.args.eex -S mix
Erlang/OTP 24 [erts-12.1.5] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1] [jit]

Compiling 1 file (.ex)
Interactive Elixir (1.13.0) - press Ctrl+C to exit (type h() ENTER for help)
iex(beacon1@127.0.0.1)1>
11:07:48.858 [debug] Node connected: data_contact1@127.0.0.1

11:07:48.866 [info]  New register from {:"data_contact1@127.0.0.1", DataContact.Interface, :data_contact, []}.

11:07:48.866 [debug] Register: data_contact1@127.0.0.1 | data_contact | []

11:07:48.866 [debug] Add node: data_contact1@127.0.0.1

11:07:48.866 [info]  Register {:"data_contact1@127.0.0.1", DataContact.Interface, :data_contact, []} complete.

11:07:51.547 [debug] Node connected: data_store1@127.0.0.1

11:07:51.547 [info]  New register from {:"data_store1@127.0.0.1", DataStore.Interface, :data_store, [:data_contact]}.

11:07:51.547 [debug] Register: data_store1@127.0.0.1 | data_store | [:data_contact]

11:07:51.547 [debug] Add node: data_store1@127.0.0.1

11:07:51.547 [info]  Register {:"data_store1@127.0.0.1", DataStore.Interface, :data_store, [:data_contact]} complete.

11:07:51.554 [debug] Getting requirements for :"data_store1@127.0.0.1"
%{
  module: DataStore.Interface,
  name: [:data_contact],
  node: :"data_store1@127.0.0.1"
}

11:07:51.555 [info]  Requirements retrieved: [
  %{
    module: DataContact.Interface,
    name: :data_contact,
    node: :"data_contact1@127.0.0.1"
  }
]

nil
iex(beacon1@127.0.0.1)2>
```

Figure 2: Beacon Server Output

最后我们看一下 Beacon 模块的 state 长什么样：

```
iex(beacon1@127.0.0.1)2> :sys.get_state(BeaconServer.Beacon)
%{
  nodes: %{"data_contact1@127.0.0.1": :online, "data_store1@127.0.0.1": :online},
  requirements: [
    %{
      module: DataStore.Interface,
      name: [:data_contact],
      node: :"data_store1@127.0.0.1"
    }
  ],
  resources: [
    %{
      module: DataStore.Interface,
      name: :data_store,
      node: :"data_store1@127.0.0.1"
    },
    %{
      module: DataContact.Interface,
      name: :data_contact,
      node: :"data_contact1@127.0.0.1"
    }
  ]
}
iex(beacon1@127.0.0.1)3>
```

Figure 3: Beacon State

就先这样，后面我们会在此基础上继续实现别的服务器。

向 UE5 项目中集成 Protobuf

2022-10-12

虚幻引擎版本：**5.0.3**

Protobuf 版本：**21.7**

> 作为不懂 C++的人，向 UE 集成 C++库真的是一种折磨。
>
> ——沃兹基·硕德

B 站上人宅的教程说得很详细，从创建插件到 Protobuf 源码的导入，如果严格按照他的方法做的话最终结果应该是可以用的。但是我在跟随的过程中遇到了问题：

视频里有一个把 Protobuf 源代码拷贝到插件代码目录里的步骤，但是从视频中的文件列表可以看出其与官方源码不完全一致。我尝试过将官方源码原封不动放进插件，结果编译无法通过；

我又暂停视频，利用视频中的文件列表删掉多余的官方源码文件，再放进插件，结果插件本身是编译通过了，但是一旦用 proto 文件生成 C++文件并在游戏代码中引入之后，编译又会报一堆的错误。

查阅众多资料之后发现，几乎没有一个最近还在更新的教程或者代码库，基本都失去了参考价值。最终找到了这个代码。代码中作者使用引入预编译好的库文件的方式实现 Protobuf 的集成给了我思路。预编译的库文件可以从官方源码中编译而来，而我只需要包含完整的头文件即可。因此我做了下面的工作：

从官方源码编译所需的包含文件(include)和库文件(.lib)

从官方 Github 仓库下载源代码。在本人写下这篇文章时，最新的 **Release** 是 21.7。下载时选择 C++的压缩包：

Figure 4: 下载源码

下载后解压，按照官方 README 的指示编译和安装。本人当时使用的是 CMake + VS2019 的方式得到最终的包含文件和库文件：

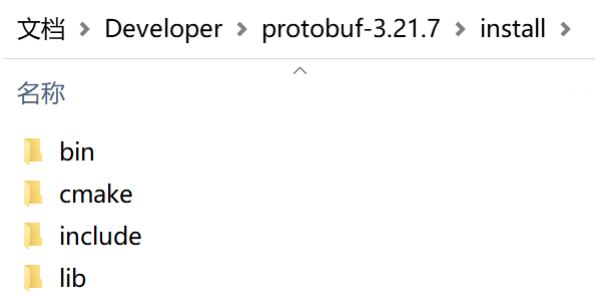


Figure 5: Proto 源码编译结果

- include 文件夹为要包含的头文件
- lib 文件夹内有编译好的.lib 库文件，我只使用了 libprotobuf.lib
- bin 文件夹内有转化.proto 文件用的可执行程序 protoc.exe

将编译结果整合进 UE5 插件

所以需要拷贝的东西如下（为了整洁，保持文件结构）：

- include 文件夹
- lib/libprotobuf.lib
- bin/protoc.exe

把这三个东西放到插件目录下，我的插件名叫 SimpleProto5，我放到了插件/Source/SimpleProto5/ThirdParty/probuf 下：

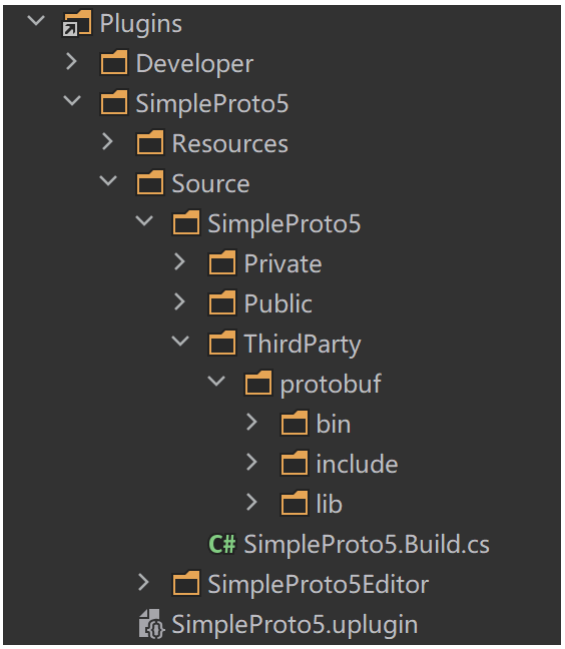


Figure 6: 放置路径

文件到位之后，还需要修改插件的.Build.cs 文件，添加头文件包含路径以及库文件查找路径：

```
// 头文件包含路径
PublicSystemIncludePaths.AddRange(
 new string[]
 {
  Path.Combine(ThridPartyPath, "protobuf/include")
 }
);

// 库文件包含路径
PublicAdditionalLibraries.Add(Path.Combine(ThridPartyPath, "protobuf", "lib", "Win64", "libprotobuf.lib"));
```

其中库文件路径的写法是临时的，因为我的目标是不只支持 Windows 一个平台，因此后续可以根据不同平台添加不同的库文件路径。

后续操作

你可以选择继续跟着人宅的视频把这个插件做完，但是关于 Protobuf 引入的过程到这里就结束了。跟着文章做的小伙伴可以编译一下项目看看，如果没有其他问题的话项目应该是可以正常编译的。

如果编译过程中出现如下错误：

```
inlined_string_field.h(430): [C4668] 没有将“GOOGLE_PROTOBUF_INTERNAL_DONATE_STEAL_INLINE”定义为预处理器宏，用“0”替换“#if/#elif”
```

只需要根据错误定位到 inlined_string_field.h 文件的第 430 行，把 #if 改为 #ifdef 即可。

祝各位在 UE5 中与 Protobuf 玩的开心！

MMO Server From Scratch(2) - Gate Server

2022-10-16

今天实现服务器的第二个部件 - gate_server

功能解析

根据[开篇架构设想](/posts/mmo-server-from-scratch/2022/20220608-mmo-server-from-scratch(0)-introduction.md)中的想法，gate_server 是用来接受用户连接的服务器。客户端通过 TCP Socket 的方式连接到 gate_server 上来，gate_server 负责把客户端发来的消息转发到指定的业务相关服务器上，并将服务器产生的消息发回客户端。

所以，根据上面的描述，gate_server 至少需要具备以下功能：

1. 连接 beacon_server 服务器，注册自身并获取自身需要的其他服务器资源
2. 监听 TCP 端口并接受连接
3. 接受客户端消息并转发至其他服务器
4. 向客户端发送消息

在 Erlang/Elixir 中，进程的使用是极其廉价的，这里说的进程不是系统进程，而是 Beam 虚拟机进程，属于用户进程。因此我打算为每个客户端传入的 Socket 连接分配一个 GenServer，用于消息交换和状态保存。

对于消息协议，我选择了 Protobuf，因此 gate_server 还需要具备消息的编解码能力。

简要实现

建立项目

首先建立本节的 gate_server 项目：

```
cd apps/
mix new gate_server --sup
```

模块划分

为了实现以上功能，需要划分几个模块：一个 Interface 模块负责集群相关操作，如注册、加入集群、获取其他服务器节点等；一个 TcpAcceptor 模块负责监听端口并接受 TCP 连接；一个 TcpConnection 模块负责和客户端进行通信。

按照这个设计，gate_server 应用的监督树如下：

```
                  application
                 /    |    \
                /     |     \
               /      |      \
       TcpAcceptorSup  InterfaceSup  TcpConnectionSup
           |           |           |
           |           |           |
           |           |           |
        Interface   TcpAcceptor   TcpConnection x N
```

其中：

- application - gate_server 主程序
- TcpAcceptorSup - TCP 监听进程监督者进程
- InterfaceSup - Interface 模块监督者进程
- TcpConnectionSup - TcpConnection 模块监督者进程
- Interface - 集群接口模块进程
- TcpAcceptor - Tcp 监听模块进程
- TcpConnection - 用户连接进程

Interface

关于监督者进程的创建我就不说了，查阅各种文档都可以找到方法。首先来实现一下 Interface 的功能。

Interface 进程的初始化流程：

1. 建立 GenServer
2. 连接给定的 beacon_server 节点
3. 连接成功后调用 beacon_server 的 register 接口，注册自身
4. 向 beacon_server 请求自身所需的其他节点

受前面实现的 beacon_server 限制，这个流程姑且就先这样，后续再继续优化。

我不在 init 函数中进行以上动作，而是将逻辑放置到 timeout 消息处理中，使得进程尽快完成初始化开始接收消息。代码如下：

```
@impl true
def init(_init_arg) do
  {:ok, %{auth_server: [], server_state: :waiting_requirements}, 0}
end

@impl true
def handle_info(:timeout, state) do
  send(self(), :establish_links)
  {:noreply, state}
end

@impl true
def handle_info(:establish_links, state) do
  Logger.info("===Starting #{Application.get_application(__MODULE__)} node initialization===", ansi_color: :blue)

  join_beacon()
  register_beacon()
  new_state = get_requirements(state)

  Logger.info("===Server initialization complete, server ready===", ansi_color: :blue)
  {:noreply, %{new_state | server_state: :ready}}
end
```

join_beacon/0、 register_beacon/0、 get_requirements/1 三个函数我就不放了，基本需要的东西就是 Node.connect/1 和 GenServer.call/3。

运行效果：

```
→ iex --name gate1@127.0.0.1 --cookie mmo -S mix
Erlang/OTP 24 [erts-12.3.2.1] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1] [jit]


20:57:46.126 [info]  ===Starting gate_server node initialization===

20:57:46.131 [info]  Joining beacon ...

20:57:46.129 [debug] Accepting connections on port 29000
Interactive Elixir (1.13.4) - press Ctrl+C to exit (type h() ENTER for help)
iex(gate1@127.0.0.1)1>
20:57:46.144 [info]  Joining beacon complete.

20:57:46.144 [info]  Registering to beacon ...

20:57:46.145 [info]  Registering to beacon complete

20:57:46.148 [info]  Getting requirements([]) from beacon ...
{:ok, []}

20:57:46.151 [info]  Getting requirements([]) from beacon complete.

20:57:46.151 [info]  ===Server initialization complete, server ready===
```

Figure 7: Interface 运行效果

此时 beacon_server 的 state 数据：

```
iex(beacon1@127.0.0.1)2> :sys.get_state(BeaconServer.Beacon)
%{
  nodes: %{"gate1@127.0.0.1": :online},
  requirements: [],
  resources: [
    %{
      module: GateServer.Interface,
      name: :gate_server,
      node: :"gate1@127.0.0.1"
    }
  ]
}
```

Figure 8: beacon_server state 内容

此时如果在 iex 中输入：

```
Node.list
```

可以发现我们的 gate_server 已经和 beacon_server 建立连接了：

```
iex(gate1@127.0.0.1)2> Node.list
[:"beacon1@127.0.0.1"]
```

Figure 9: beacon_server state 内容

TcpAcceptor

TcpAcceptor 是用来接收 TCP 传入链接的进程，同样是一个 GenServer。这个进程的逻辑非常简单，直接看代码：

```
defp listen(port) do
  {:ok, socket} = :gen_tcp.listen(port, [:binary, packet: 0, active: true, reuseaddr: true])

  Logger.debug("Accepting connections on port #{port}")
  loop_acceptor(socket)
end

defp loop_acceptor(socket) do
  {:ok, client} = :gen_tcp.accept(socket)

  {:ok, pid} =
    DynamicSupervisor.start_child(
    GateServer.TcpConnectionSup,
    {GateServer.TcpConnection, client}
    )

  :ok = :gen_tcp.controlling_process(client, pid)

  loop_acceptor(socket)
end
```

listen/1 函数用来监听指定的端口，在成功之后开始接受传入 TCP 连接；loop_acceptor/1 函数用来循环接受 TCP 连接，一旦接受一个连接，则为其创建一个 TcpConnection 进程，并将该链接的 socket 控制权转交给新生成的 TcpConnection 进程。在 Elixir 中，尾递归可以被优化，不会无限制占用栈空间，因此在函数的最末尾进行递归调用实现循环接受 TCP 连接。

TcpConnection

这是本节最核心的功能模块，负责与客户端的一切通信。

TcpConnection 同样是一个 GenServer，用于保存一些状态和传输 TCP 数据。其初始化函数如下：

```
@impl true
def init(socket) do
  Logger.debug("New client connected.")
  {:ok, %{socket: socket, status: :waiting_auth}}
end
```

可以看到，我们把 socket 信息存入了进程的 state 中，方便后续调取。此处的 status 属性暂且抛开不管，用于后续用户的鉴权，本节不作讨论。

接下来是本进程的核心函数 - TCP 消息接收函数：

```
@impl true
def handle_info({:tcp, _socket, data}, %{socket: socket} = state) do
  result = "You've typed: #{data}"
  send_data(senddata, socket)

  {:noreply, state}
end
```

此处为了方便测试，先写一个简单的 echo 功能。GenServer 还贴心地提供了对 TCP 连接状态变化的处理，只需实现以下两个函数：

```
@impl true
def handle_info({:tcp_closed, _conn}, state) do
  Logger.error("Socket #{inspect(state.socket, pretty: true)} closed unexpectly.")
  DynamicSupervisor.terminate_child(GateServer.TcpConnectionSup, self())

  {:stop, :normal, state}
end

@impl true
def handle_info({:tcp_error, _conn, err}, state) do
  Logger.error("Socket #{inspect(state.socket, pretty: true)} error: #{err}")
  DynamicSupervisor.terminate_child(GateServer.TcpConnectionSup, self())

  {:stop, :normal, state}
end
```

这里我们就可以对 TCP 的连接建立以及信息收发功能进行测试了。假设我的 gate_server 运行在本机 29000 端口上，我们运行 Telnet：

```
telnet 127.0.0.1 29000
```

即可开始与服务器进行通信。测试结果：

```
→ telnet 127.0.0.1 29000
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
hello
You've typed: hello
echo test
You've typed: echo test
```

Figure 10: beacon_server state 内容

接下来的工作

gate_server 到这里暂告一段落，一个能够接受 TCP 客户端连接并且进行高并发通信的服务器就基本完成了。下一步我们将继续实现消息协议相关内容，引入 Protobuf 消息库并进行消息分发，进一步完善网关服务器功能。此部分功能等到下一步实现场景服务器 scene_server 后再回来继续，敬请期待！

MMO Server From Scratch(3) - Scene Server(0) - 概要

2022-10-19

今天开始实现服务器的第三个部件 - **scene_server**，即场景服务器。本节只关注接下来一段时间将要简单实现的内容，而不是一个完整的场景服务器。场景服务器包含的内容极多，我们将一步一步将其实现。

功能解析

场景服务器 scene_server 用于为玩家提供场景服务，即玩家在进入地图之后的一切与场景相关的内容均由场景服务器提供，如移动同步、周围玩家状态更新、战斗结算、玩家行为验证等。

scene_server 的消息可以来自其他不同服务器，如网关服务器 gate_server、世界服务器 world_server、用户代理服务器 agent_server 等。但是如果把全部的功能点一次性全部纳入考虑的话我怕会顾不过来，所以目前让我们把注意力集中在场景服务器最基本的职能上：移动同步。

当然我们不能丢到最基础的功能：玩家进入/离开场景

移动同步

什么是移动同步？简单地说，在多人游戏里，你需要看到你身边的其他玩家，队友也好、敌人也好，你希望能够看到身边其他玩家的各种操作，角色的位置和移动也是各种操作中的一种。为了实现这种效果，当你的角色在移动时，你需要将自己移动的信息发送给身边所有玩家，以便他们在自己的客户端上更新你的角色表现，使得他们看到的和你看到的内容是 一致 的。

这里涉及两个条件：

1. 将自己的信息发送给其他多个玩家
2. 信息发送的目标是自己周围一定范围内

即：在所有玩家中搜索出自己身边的玩家，并向他们广播自己的移动消息。这个功能有个专有名词，叫做 AOI(Area Of Interest)。顾名思义，每个玩家都具有一个自己感兴趣的范围，在这个范围内发生的事情才会告诉客户端，超出这个范围的动静一概不管（当然也没有这么绝对）。

因此为了实现在游戏中能够看到其他玩家的移动，我们需要实现一个 AOI 系统。经过一番资料搜索，目前实现 AOI 的方式主要有三种：

1. 灯塔
2. 九宫格
3. 十字链表

这三种方式大家可以自行搜索。在这里我选择了 十字链表 作为前期粗略实现，也让自己对相关概念进行了解。

玩家进入/离开场景

玩家进入或离开场景的情况不止一种，目前我能想到以下几种：

1. 玩家进入/离开游戏
2. 玩家移动过程中去往其他场景服务器
3. 玩家因为某些玩法（如副本、传送等）离开/进入大世界

当前为了方便我们只考虑第一种，其他两种待后期代码完善之后再进行实现

简要设计

计算量规划

scene_server 作为场景管理服务器，集多项职能于一身，且多数职能属于计算密集型。Elixir 可能再并发方面比较擅长，但在数学计算方面与 C/C++ 一系的语言相比就差得多了。但幸运的是，Elixir 和 Erlang 一样可以允许通过多种方式与外部程序通信：

1. C Node
2. NIF
3. Port Driver
4. Port

每种方法都有不同的特点和适用环境。在这个项目里，我们需要的是巨大的计算量和相对较低的延迟，而在以上四种方法中，NIF 的消息传递延迟是最低的，而且有一个非常棒的 NIF 库使得 NIF 的开发过程更加高效，程序更加健壮—— Rustler。该库使用 Rust 作为 NIF 函数的开发语言，兼具了内存安全和高效，可谓完美之选。

因此，将重度计算的内容扔给 Rust 来做，如 AOI 数据管理、各种计算验证等；并发消息传递的工作由 Elixir 来完成，如玩家之间的交互、消息广播等。

初步监督树

为了让前期的代码尽量简化，前期的功能规划目前只包括以下几部分：

1. 玩家进程模块 PlayerCharacter，每个玩家为一个进程，玩家交互通过进程间消息传递实现
2. 玩家管理模块 PlayerManager，用来管理玩家 ID 与进程之间的映射，方便其他进程进行查询从而向目标玩家进程发送消息
3. AOI 管理模块 Aoi，用来与 NIF 通信，操纵 AOI 数据结构（十字链表）
4. 集群操作模块 Interface，与其他服务器一样

监督树形态大概如下：

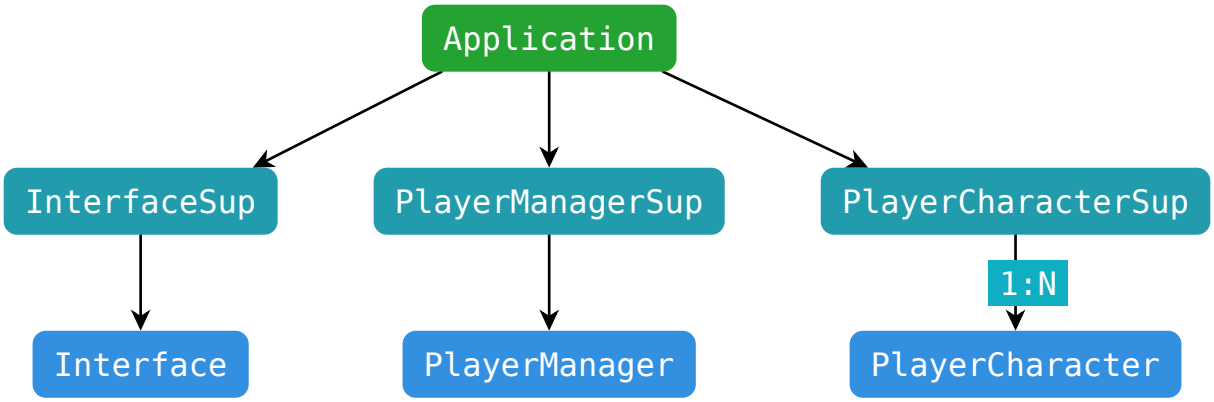


Figure 11: Scene Server Supervision Tree

其中：

- application - scene_server 主程序
- InterfaceSup - Interface 模块监督者进程
- AoiSup - AOI 管理模块监督者进程

- PlayerManagerSup - 玩家管理 PlayerManager 模块监督者进程
- PlayerCharacterSup - 玩家进程 PlayerCharacter 模块监督者进程
- Aoi - AOI 管理模块进程
- PlayerManager - 玩家管理模块进程
- PlayerCharacter - 玩家进程

接下来的工作

前期一个极致简单的 scene_server 就先规划到这里 ,未来还会增加其他更多功能。下一篇将着重实现 Aoi 的 NIF 部分，使用 Rust 编写。敬请期待。

MMO Server From Scratch(4) - Scene Server(1) - AOI 算法与数据结构

2022-10-21

本节着重讨论场景服务器的 AOI 模块中的 NIF 部分，确定接口、算法和数据结构。

功能解析

AOI(Area Of Interest) 是用来管理大世界上各种实体之间位置关系的模块，可以接受某个实体的 查询身边其他实体 的请求，同时负责维护各实体位置关系的数据结构。当实体在大世界上发生移动时，AOI 模块需要更新实体在数据结构中的位置，以便业务进程在请求身边实体列表时返回更新后的结果。

从描述上就可以看出，这个模块需要频繁进行列表的查询、插入、删除操作，计算量不算小，因此如果把这部分功能拿给 Elixir 实现的话恐怕会力不从心，运算速度不足导致客户端之间的同步操作延迟过大，影响游戏体验。因此我们选择使用更快速的办法实现这个模块——NIF(Native Implemented Functions)，使用执行效率更高的语言实现计算量最大的部分，同时在 Elixir 中提供调用接口，这样将两种语言的代码联系起来，各取所长。

经过一段时间的网上搜索，最终选定了 Rustler 库，该库使用 Rust 作为 NIF 函数编写语言，同时提供了运行时安全保障，使得 NIF 函数中的错误不会 Down 掉整个 BEAM 虚拟机，而这个问题是 NIF 的一个重要缺陷，而 Rustler 解决了这个问题；同时 Rust 也提供了安全的内存管理特性，使得我可以不必像在 C++ 中那样时刻关注内存分配和销毁的情况。

算法和数据结构分析

我们需要的操作基本分为三种：查找、插入、删除。可以选择的数据结构基本为两种：数组 和 链表。下面分别讨论两种结构的特点。

数组

数组是一块连续的内存空间，使用索引下标访问元素。

对于 查找 来说，不同情况下的时间复杂度见表：

是否拥有索引	是否有序	时间复杂度
是	-	O(1)
否	否	线性查找 O(n)
否	是	二分查找 O(lg(n))

Table 1: 数组查找时间复杂度

对于 插入 和 删除 来说，假设已经确定元素索引，其复杂度均为 O(n)，因为插入和删除某个元素时，其后的元素均需要整体移动。

可以看出，数组 结构对于 查找 来说效率较高，但 插入/删除 元素则效率较低。

链表

链表是通过元素内指针指向其他元素形成的链式结构。

对于链表来说，由于 查找 只能对结构进行遍历，因此其时间复杂度为 O(n)。但是如果查询方持有元素且链表有序的话则可以相对降低，但在最坏情况下依然为 O(n)。

对于 插入 和 删除 来说，如果不考虑查找过程，链表只需要修改元素本身及前后元素的指针即可，因此其时间复杂度为 O(1)。

总结

对于 AOI 中的场景，一般是 玩家移动 -> **AOI** 更新元素位置 -> 玩家获取元素周围一定半径内其他元素 。当 AOI 更新元素位置时，意味着元素顺序会频繁变化，这样对于数组来说就失去了其最大优势 —— 索引。如果采用数组结构，那么对元素所作的所有操作都将有一个前置操作—— 二分查找。这样的话与 链表 相比，效率上应该略逊一筹。

数组 和 链表 都可以通过建立 层 机制进行优化，使得花费时间进一步减少。至于优化后的结构孰优孰劣，待我有时间全部实现后再做对比。

至于本系列，由于本人对链表很久不用了不甚熟悉，而且在类似问题上找到了 Discord 的一篇文章，该文章使用了类似 跳跃表 的机制，将数组变成了两层，提高了对其进行操作的速度。为了方便起见，在目前本人暂时先选择 数据 的形式，在 Discord 代码的基础上进行改造。

简要实现

首先为我们的 scene_server 添加依赖，将依赖项添加到 mix.exx 文件的 deps 函数中：

```
{:rustler, "~> 0.26.0"}
```

然后使用命令创建 NIF 模块：

```
mix rustler.new
```

根据提示输入 Elixir 和 Rust 中的模块名称。本人使用的是 SceneServer.Native.CoordinateSystem 和 coordinate_system。

该命令会在 scene_server 项目文件夹下生成 native 文件夹，并将 NIF 模块文件放在其下：

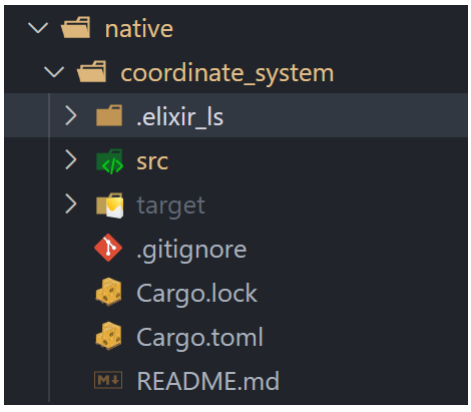


Figure 12: Rustler 结构

SortedSet 分析

上面 Discord 的文章中实现了一个名为 SortedSet 的数组结构，其包含一个元素为 Bucket 的数组，而 Bucket 则包含一个元素为 Item 的数组，如此将一个数组定义为两层。本节中我也将沿用这些名称，对代码进行改造。

Item

Item 即是最基本的元素结构体，其包含 实体 ID 、 坐标 、 所在轴向 三个属性：

```
// 轴向枚举类型
#[derive(NifUnitEnum, Clone, Debug, Copy)]
pub enum OrderAxis {
    X,
    Y,
    Z,
}

// 坐标结构体类型
#[derive(NifTuple, Clone, Debug, Copy)]
pub struct CoordTuple {
    pub x: f64,
    pub y: f64,
    pub z: f64,
}

// Item结构体类型
#[derive(NifStruct, Clone, Debug, Copy)]
#[module = "Item"]
pub struct Item {
    pub cid: i64,
    pub coord: CoordTuple,
    pub order_type: OrderAxis,
}
```

其中的 NifUnitEnum、NifTuple、NifStruct 是为了将 Rust 内的类型转化为 Elixir 类型 ,从而使得其可以被传递。

为了能让 Item 之间能够被比较和排序，我们需要实现几个 Trait ：

1. PartialEq
2. PartialOrd
3. Eq
4. Ord

实现细节此处就不再展示了，各位看官有兴趣自行实现即可。此处之讨论一下 Item 之间该如何比较。

如果是 排序 的话，我们需要一个可以被排大小的值，在这里就是坐标值 coord 了，但是三个方向的顺序怎么确定呢 ?这是 order_type 就派上用场了 ,order_type 是哪个轴 ,那我们就把坐标里哪个轴上的数值拿来比较。

如果是 查找 的话，有了上面的比较大小还不够，别忘了我们的 cid 属性。坐标值相等并不意味着两个 Item 元素就是相等的。想想我们什么时候需要查找元素 ?只有唯一的一个场景 ,那就是找到对应玩家/实体的 Item，而实体之间的区分使用 cid，因此在相等条件中我们还需要加入 cid 相等的逻辑。

除此之外我们还需要一个 distance(距离) 方法，用来计算两个 Item 之间的距离，使用简单的勾股定理即可。

之后可以写几个单元测试用例，试一试 Item 的功能正不正常。

Bucket

Bucket 是一个包含 Item 数组的结构体：

```
#[derive(NifStruct, Clone, Debug)]
#[module = "Bucket"]
pub struct Bucket {
    pub data: Vec<Item>,
}
```

作为包含元素列表的类型，Bucket 需要实现查找、插入、删除元素的方法。但是，贴心的 Rust 已经为我们实现了 二分查找；删除方法同样 Rust 内置类型 Vec 已经实现；对于插入，我们需要自行实现，对 Vec 的插入方法进行上层包装。这是因为我们想要让 Bucket 的长度固定，从 Rust 官方文档得知，Vec 有一个 capacity(容量) 属性，代表 Vec 预分配内存的大小，如果 Vec 长度超过了 capacity，那么 Rust 将需要为其额外分配内存，造成不必要的计算消耗；而且当我们需要 插入/删除 元素时，我们希望数组的长度越短越好。因此综合上面的因素，我们需要 Bucket 为固定长度。

这样一来，在插入元素的方法中，我们需要对列表长度进行判断，如果长度超过了设定的长度，则需要对列表进行 分裂，将一个 Vec 一分为二，成为两个对半分的 Vec。

Bucket 作为 SortedSet 的元素，同样需要具备比较的能力。但是比较特殊的是，Bucket 的比较对象是 Item，这是因为我们一般查找的对象都是 Item，在 SortedSet 进行查找时，需要确定给定 Item 在哪个 Bucket 内部。

最后，Bucket 还需要一个功能，那就是返回给定 Item 一定范围内其他 Item 列表的方法。实现很简单，使用 filter 方法对列表内元素进行遍历即可，该过程可以借助 Rayon 库进行并行优化。

SortedSet

SortedSet 是最外层真正的元素数组。其为一个包含 Bucket 数组的结构体，其同时还包含容量、元素数量等属性：

```
#[derive(Debug, NifStruct, Clone, Copy)]
#[module = "Configuration"]
pub struct Configuration {
    // Bucket最大容量
    pub bucket_capacity: usize,

    // SortedSet最大容量
    pub set_capacity: usize,
}

#[derive(Debug, NifStruct, Clone)]
#[module = "SortedSet"]
pub struct SortedSet {
    configuration: Configuration,
    buckets: Vec<Bucket>,
    size: usize,
}
```

与 Bucket 类似，我们也希望 SortedSet 有一个最大长度避免额外的内存分配开销，但是不需要分裂，只要最大容量即可。

方法实现与 Bucket 类似，需要 新建列表、查找元素、插入元素、删除元素、查找一定范围内 Item 列表 等方法。对于 SortedSet 来说，对元素的所有操作都需要先找到 Bucket 然后在对应 Bucket 中操作 Item。

CoordinateSystem

最后是我们的顶层结构体 CoordinateSystem，包含 X、Y、Z 三个轴向的三个 SortedSet 成员：

```
#[derive(Debug, NifStruct, Clone)]
#[module = "CoordinateSystem"]
pub struct CoordinateSystem {
    configuration: Configuration,
    axes: Vec<SortedSet>,
}
```

此处 SortedSet 以列表形式存储，是为了后期对其进行并行优化。例如 插入元素 方法：

```rust
pub fn add(&mut self, item: &Item) -> AddResult {
    let mut jobs: Vec<SetAddResult> = Vec::with_capacity(3);

    self.axes
        .par_iter_mut()
        .enumerate()
        .map(|(idx, ss)| {
            return ss.add(Item {
                cid: item.cid,
                coord: item.coord.clone(),
                order_type: OrderAxis::axis_by_index(idx),
            });
        })
        .collect_into_vec(&mut jobs);

    let result = match (jobs[0], jobs[1], jobs[2]) {
        (SetAddResult::Added(ix), SetAddResult::Added(iy), SetAddResult::Added(iz)) => {
            AddResult::Added(ix, iy, iz)
        }
        (rx, ry, rz) => AddResult::Error((rx, ry, rz)),
    };

    result
}
```

其中的 par_iter_mut() 方法将一个 Vec 中的元素转化为并行迭代器，使得 map 方法可以在每个元素上并行运行。

CoordinateSystem 要实现的方法就是接口所需的所有方法，目前实现的有：

1. 建立结构
2. 插入元素
3. 删除元素
4. 更新元素位置
5. 查找一定范围内 Item 列表

接下来的工作

Nif 部分到这里基本完成，我们拥有了一个简陋但相对完整的 AOI NIF 库，下一步我们将继续将其接入到 Elixir 程序里，使其可以被玩家进程调用。