分类号：TP393.02　　U D C：D10621-408-(2023)3731-0
密　级：公 开　　编 号：2019081190

成 都 信 息 工 程 大 学
学 位 论 文

元宇宙分布式服务端框架设计与实现

论 文 作 者 姓 名：徐绿国
申 请 学 位 专 业：信息安全
申 请 学 位 类 别：工学学士
指导教师姓名(职称)：彭承宗（副教授）
论 文 提 交 日 期：2023 年 6 月 7 日

元宇宙分布式服务端框架设计与实现

摘要：近年来，随着互联网飞速发展，元宇宙引发了全球科技产业的高度关注和投资热潮，在这个元宇宙技术的领域里，服务端技术的高低决定了元宇宙发展的成熟度，一个高度可扩展、高性能、高可用的服务端框架用以支持元宇宙应用程序的开发和部署是本次研究的方向。

本论文提出了一种面向元宇宙的分布式服务器框架设计与实现方法，为构建高性能、可扩展的元宇宙系统提供了理论基础和技术支持。

通过对需求和特点进行了深入分析，本文明确了分布式服务端框架的设计目标和相关技术。采用 C++和 Lua 语言共同编写此框架，将框架拆分成了内核层、节点层和业务层，提升程序的拓展性和可维护性。

将业务功能拆分到不同节点上运行，以实现不同功能的解耦和资源的高效利用，利用动态负载均衡算法，可以在不同节点之间有效地分配任务，确保系统的稳定性和性能。

最后，本文采用科学的方式对框架进行测试且给相应的实验数据，并将该框架运用在商业的两个 VR 项目中，对所做的内容进行了总结和展望。

关键词：服务端框架；C++；分布式；高性能；元宇宙

Design and Implementation of Metaverse Distributed Server Framework

Abstract: In recent years, with the rapid development of the Internet, the metaverse has attracted high attention and investment from the global technology industry. In this field of metaverse technology, the level of server technology determines the maturity of metaverse development. A highly scalable, high-performance, and usable server framework to support the development and deployment of metaverse applications is the direction of this study.

This paper proposes a distributed server framework design and implementation method for the metaverse, providing theoretical basis and technical support for building high-performance and scalable metaverse systems.

Through in-depth analysis of requirements and characteristics, this civilization has identified the design goals and related technologies of a distributed server framework. The framework is jointly written in C++ and Lua language, which divides the framework into kernel layer, node layer and business layer, improving the extensibility and maintainability of the program.

Splitting business functions into different nodes for operation to achieve decoupling of different functions and efficient utilization of resources. By utilizing dynamic load balancing algorithms, tasks can be effectively allocated between different nodes, ensuring system stability and performance.

Finally, this article adopts a scientific approach to test the framework and provides corresponding experimental data. The framework is applied to two VR projects in the business, and the results are summarized and prospected.

Key words： Server side framework; C++; Distributed; High performance; Metaverse

目 录

论文总页数：47 页

1 绪 论 1
1.1 研究背景与意义 1
1.1.1 研究背景 1
1.1.2 研究意义 1
1.2 国内外研究现状 2
1.3 设计目标 2
2 服务端技术背景 4
2.1 模块组合 4
2.2 Reactor 网络模型 4
2.3 数据传输 5
2.3.1 REST 和 RPC 通信风格 5
2.3.2 数据交换格式 6
2.4 数据分离 7
3 框架设计 8
3.1 开发语言的选择 8
3.2 结构设计 9
4 框架内核层 10
4.1 内核层设计 10
4.1 服务管理器 11
4.1.1 插件与模块 11
4.1.2 插件与模块生命周期 11
4.2 动态数据容器 12
4.3 基础插件 13
4.3.1 配置插件 13
4.3.2 网络插件 13
4.3.3 内核插件 16
4.3.4 日志插件 17
4.3.5 Lua 插件 17
4.4 基础工具 19
4.4.1 Generator 19
4.4.2 sqkctl 20
5 框架节点层 21
5.1 节点介绍 21
5.2 节点基础功能 23
5.2.1 动态连接与管理 23
5.2.2 负载均衡 23
5.2.3 转发表 23
5.2.4 心跳检测 24

5.2.3 断线重连 25
5.3 通信机制 26
6 框架业务层 28
6.1 业务节点 28
6.2 业务项目解耦 29
7 打包工作流程 30
7.1 打包工作流程介绍 30
7.2 基础环境 31
7.3 生成配置 32
7.4 打包部署 33
8 压力测试 34
8.1 压力测试介绍 34
8.2 登录节点压力测试 35
8.3 代理节点压力测试 37
9 商业运用 41
9.1 VR 丧尸卡牌下棋游戏 41
9.2 多人在线射击游戏 42
10 工作总结与展望 43
10.1 总结 43
10.2 展望 44
参考文献 45
致 谢 46
声 明 47

1 绪 论

1.1 研究背景与意义

1.1.1 研究背景

元宇宙概念的兴起：元宇宙的概念自 20 世纪 80 年代以来就已经存在，但近年来随着虚拟现实、增强现实、人工智能和区块链等技术的迅速发展，元宇宙的概念受到了广泛的关注。大型科技公司如 Meta、谷歌和腾讯等都宣布了自己的元宇宙计划，这引发了对元宇宙服务端框架的研究需求。

技术基础设施的发展：元宇宙是需要庞大而复杂的技术基础设施来支持其各种功能和应用。这包括高性能的计算、实时的数据处理、网络通信、存储系统等。研究元宇宙服务端框架有助于构建稳定、可扩展、高效和安全的技术基础设施，以满足元宇宙的需求。

多维度的体验和创造：提供了丰富多样的体验和创造机会。用户可以在虚拟世界中扮演不同的角色，探索各种场景和任务，与其他玩家互动，甚至创造自己的虚拟内容、物品或经济体系。这为用户提供了更多选择和自由度，满足了他们的多样化需求。

虚拟世界的互联互通：元宇宙游戏打破了传统游戏的边界，创造了一个虚拟世界，不同游戏、应用和用户可以在其中互联互通。这使得用户可以在一个共享的虚拟空间中进行交互、合作和创造，增强了社交互动的可能性。

1.1.2 研究意义

可以推动技术创新和发展,元宇宙服务端框架的研究促进了虚拟现实、增强现实、人工智能和区块链等关键技术的创新和发展。通过解决元宇宙服务端框架所面临的挑战，为了更快速、稳定、低成本开发元宇宙产品，需要有一个比较成熟、功能丰富、快速接入，性能强的框架。目前在开源社区中，暂未拥有一款比较成熟的框架，大部分服务端框架局限性都比较高，功能单一，跨平台性差，容灾性差，难以达到商业级别的要求，绝大部分成熟的商业框架都属于公司个人的资产，不会开放源代码。本次项目会伴随着两款商业游戏不断完善和更新，核心框架源代码进行 Github 开放，免费供给个人或供给商业公司学习使用，那么实现一个比较完善的游戏服务端框架且满足以上需求是本次研究所要做的内容。
开源地址: https://github.com/pwnsky/squick

1.2 国内外研究现状

目前在国内外，有许多流行的支持多人实时在线的游戏服务端框架。在游戏领域中，技术是通用的，国内外使用的技术是差不多，主要从两个。

国外研发比较知名的有 BigWorld[1]、Unreal Engine Networking[5]、Unity Networking[4]等等。BigWorld 是一个游戏引擎和服务端框架，用于开发和运行大型多人在线游戏。它性能高，支持分布式，支持大规模地图和世界管理，数据库和持久化。Unreal Engine Networking 是 Unreal 引擎的网络模块，提供了多人游戏开发所需的功能，包括同步、复制、预测等。Unity Networking 是 Unity 引擎的内置网络功能，适用于创建多人游戏和实时互动体验。

国内研发比较知名的有 ET[2]、NoahGameFrame[3]、Pomelo[6]。ET 是一款采用 C#研发的双端框架，是 Unity 客户端研发框架又是.net 服务端框架，可以让服务端与客户端跑同一块逻辑代码，降低开发成本，目前也有较好的性能和商业项目支持。NoahGameFrame 是国内的海外留学生基于 BigWorld 的思想研发出来的产品，主要采用 C++实现的，拥有很好的性能和分布式的支持，腾讯游戏中有一部分游戏是采用该框架进行研发的。Pomelo 是网易采用 nodejs 研发的产品，比较适合的应用领域是网页游戏、社交游戏、移动游戏的服务端，Pomelo 可以用如此少的代码达到强大的扩展性和伸缩性。不幸的是，Pomelo 用于大型的 MMORPG 游戏开发，尤其是 3d 游戏是比较吃性能的，还是需要像 BigWorld 这样的商用引擎来支撑。

BigWorld 各方面都很优秀，由于是商业闭源项目，对研究不利。虽然 Unreal Engine Networking 和 Unity Networking 都是开源的，功能丰富，解决了很多模拟物理效果的问题，但由于采用的是 DS 服务架构，占用 CPU 和内存过高问题一直是个头疼的问题，提高了很大的运营成本，公司一般都不会采取该方案进行研发项目。ET 是一款比较优秀的开源框架，性能也不错，支持 KCP 协议，多数商业公司也在运用该框架。NoahGameFrame 拥有丰富的功能支持，也有很强的性能表现，但其代码耦合度太高，不方便对新功能进行拓展或对就功能进行分离。Pomelo 对扩展性有很大的优势，劣势主要是性能问题，不方便对 MMORPG 游戏进行开发。

1.3 设计目标

本次的设计目标是借鉴于 Bigworld 和 NoahGameFrame、UE4 相关特性综合确定的。一下是涉及目标的点。

1. 插件模块组合，让开发功能变成开发拓展的插件模块。
2. 解耦框架代码与业务项目代码。
3. 日志系统，能够将日志异步写入到文件里，且在 Linux 环境下程序崩溃时，自动 Dump Stack 调用链写入崩溃文件。

4. 分布式服务，各服务之间通过网络来进行沟通，可通过分布式+集群方式减轻服务器压力。

5. 支持业务代码的热更新。

6. 支持主流的 Mysql、MongoDb、Redis 等数据库。

7. 支持跨平台，可以在 Windows、Linux 上编译和开发，在 Windows 上可以支持 VS 进行项目开发，不同平台的服务可以进行连接。

8. 支持 1 变 N 或 N 变 1，一个物理机上单个进程启动全部服务器，方便开发调试。不同物理机上启动单个或多个服务器。

9. 拥有比较完整的自动化工具，如自动化代码生成、编译、打包、部署、清理等等。

10. 支持主流的协议，如 HTTP/HTTPS、Websocket、TCP、UDP、KCP 等等。

11. 为了保证包的正确处理与不丢失，基于 TCP 之上，自定义双向 RPC 协议。

2 服务端技术背景

服务端技术是指在网络环境中运行的计算机程序或服务。它负责处理来自客户端的请求，并提供相应的响应处理，它的技术栈庞大且复杂，这里列出几个比较经典的技术，模块组合、Reactor 网络模型[14]、数据传输、数据分离等，可以借助这些技术为框架打下基础。

2.1 模块组合

继承是一种通过扩展已有类，称为基类或父类来创建新类，称为派生类或子类的机制。子类可以继承父类的属性和方法，并可以添加自己的额外属性和方法。继承关系形成了一种层次结构，子类可以重用父类的代码，并可以根据需要进行修改或扩展。继承可以实现代码的重用和继承关系的表达，但它也会引入类之间的紧密耦合关系，使得代码的维护和扩展更加困难。

组合是一种通过将多个类组合成一个更大的类来实现代码复用的机制。在组合中，一个类包含其他类的实例作为其成员变量，这样该类就可以通过调用成员变量的方法来使用其他类的功能。组合关系更加灵活，类之间的耦合度较低，可以通过组合不同的类来构建出各种复杂的对象结构。组合可以实现更加灵活的代码复用和对象组合，同时也使得代码的修改和扩展更加容易。

结合继承和组合的优点，可以将服务端各种功能拆分为模块，通过组合来实现程序的完整性，模块之间互相使用接口的时候，通过定义接口来决定一个模块开放了什么功能，实现这个功能只需继承于该接口或者抽象类来实现实例，让其发挥组合与继承的各个优势。在插件系统中将会使用继承和组合来实现各种模块的管理和接口调用。

2.2 Reactor 网络模型

本次采取的网络模型基于 Reactor 模型来驱动管理 IO 数据，用于更快速实时的处理并发的处理 IO 事件。其该模型的基本组成包括以下几个要素：

事件循环：事件循环是 Reactor 模型的核心部分，它负责监听事件并分发事件给相应的处理器。事件循环在一个无限循环中等待事件的发生，一旦有事件发生，就将事件交给相应的处理器进行处理。

事件处理器：事件处理器是负责处理具体事件的组件。它包含了事件发生时需要执行的逻辑代码。事件处理器可以根据事件的类型进行相应的操作，比如读取数据、写入数据、接收连接等。

事件分派器：事件分派器负责将事件分发给对应的事件处理器。它维护了一个事件注册表，用于记录不同类型的事件和对应的处理器。当事件发生时，事件分派器根据事件的类型找到相应的处理器，并将事件传递给处理器进行处理。

文件描述符集合：文件描述符集合用于存储需要监听的事件。在事件循环开始前，应用程序会将需要监听的文件描述符添加到文件描述符集合中，事件循环会根据集合中的文件描述符来等待事件的发生。

Reactor 模型的工作流程

1. 应用程序初始化，创建事件循环和事件分派器。

2. 应用程序将需要监听的文件描述符添加到文件描述符集合中，并关联相应的事件处理器。

3. 事件循环开始运行，进入事件监听状态。

4. 当有事件发生时，事件循环将事件分发给对应的事件处理器。

5. 事件处理器执行相应的逻辑操作，比如读取数据、写入数据等。

6. 处理完事件后，事件处理器返回并等待下一个事件的到来。

7. 通过使用 Reactor 模型，应用程序可以实现高效的事件驱动 IO，同时避免了阻塞式的 IO 操作，提高了并发处理能力。

Reactor 模型的核心思想是事件驱动和非阻塞 I/O。通过事件驱动，服务器可以高效地处理大量并发连接，避免了传统的一线程一连接模型的性能瓶颈。通过非阻塞 I/O，可以充分利用系统资源，实现高吞吐量和低延迟的网络通信。

2.3 数据传输

数据传输涉及到协议和数据交换格式，协议通常分两种风格，REST 和 RPC，数据交换格式常用的有 JSON、XML、Protobuf。在设计网络协议的时候，选择通信风格和数据交换格式很大程度上决定了服务器网络的包的传输速率和性能。

2.3.1 REST 和 RPC 通信风格

REST 和 RPC 是两种不同的通信协议或风格，用于构建分布式系统中的服务通信。REST 是一种基于 HTTP 协议的软件架构风格，它使用统一的接口定义和约束来构建网络应用程序。REST 服务通过 HTTP 方法，如 GET、POST、PUT、DELETE 对资源进行操作，并使用 URI（统一资源标识符）来标识资源。它使用基于状态的转移的概念，即客户端通过操作资源的表现形式，例如使用 JSON 或 XML 表示来实现与服务端的交互。REST 强调无状态性、可伸缩性和可缓存性，并支持多种数据格式。它广泛应用于 Web 服务开发和构建 API。RPC 是一种用于远程通信的协议或模式，它允许应用程序通过网络调用远程的方法或函数，并获取返回结果。RPC 隐藏了底层网络通信的细节，使得远程调用就像本地调用一样简单。通常情况下，RPC 使用特定的序列化协议，如 JSON、Protocol Buffers 等，来编码参数和返回结果，并使用传输协议，如 TCP、HTTP 等，在客户端和

服务器之间进行通信。RPC 可以使用不同的通信模式，如单向调用、双向调用或请求-响应模式。

REST 和 RPC 在设计和使用上是有一些区别的，REST 是一种基于 HTTP 的架构风格，更关注资源和统一接口的概念，以及面向资源的操作。它通过 URL 和 HTTP 方法进行通信，并支持多种数据格式。REST 倾向于更简单、灵活和可扩展的设计。RPC 是一种通信协议或模式，更关注方法调用和过程的概念，强调远程调用的透明性。它通常使用特定的序列化协议，并使用底层的传输协议进行通信。RPC 可以提供更紧密的语言级别的集成和较低的通信开销。在选择使用 REST 还是 RPC，取决于具体的需求和场景。如果需要构建面向资源和简单易用的 Web 服务或 API，REST 是一个不错的选择。如果需要实现更紧密的方法调用和透明性，RPC 可能更适合。

REST 和 RPC 并不是相互排斥的，实际上可以结合使用。可以使用 RPC 作为底层的通信协议，同时在 RPC 之上使用 REST 风格定义接口和资源。这种混合使用可以根据具体需求来设计和实现分布式系统，在本次框架设计中，就同时使用了两种通信协议方式。

2.3.2 数据交换格式

数据传输中，为了进行数据的交换，主要涉及三种常见的格式，分别是 JSON、XML 和 Protobuf 是三种常见的数据序列化和交换格式，它们在不同的场景中有各自的特点和应用。JSON 是一种轻量级的数据交换格式，以易于阅读和编写的文本形式表示结构化数据。它使用键值对的方式表示数据，支持基本数据类型，字符串、数字、布尔值和复杂数据类型，如对象、数组等。JSON 的优点在于简洁、易于理解和处理，广泛应用于 Web 开发和 API 数据传输。XML 是一种通用的标记语言，用于表示和传输结构化数据。XML 使用标签来定义数据的结构和元素之间的关系。它具有良好的扩展性和兼容性，并且支持自定义标签和命名空间。XML 的优点在于可读性强，能够表达复杂的数据结构和元数据，广泛应用于数据交换和配置文件等领域。Protobuf 是一种由 Google 开发的二进制数据序列化格式，用于高效地序列化结构化数据。Protobuf 使用简洁的消息描述语言来定义数据的结构，然后根据描述生成对应的数据访问类。与 JSON 和 XML 相比，Protobuf 生成的数据大小更小，解析速度更快，同时具有跨平台和语言的特性。Protobuf 适用于高性能和网络传输效率要求较高的场景，比如大规模分布式系统和通信协议定义。

在选择适合的数据格式时，还需要考虑数据大小和传输效率、可读性和理解性、拓展性和兼容性、开发工具和生态系统支持。如果数据量较大或网络传输效率要求较高，可以选择二进制格式，如 Protobuf 以减少数据大小和传输时间。如侧重可读性和可理解性，需要人类可读和编辑的数据格式，JSON 和 XML 是更

好的选择，因为它们以文本形式呈现，易于阅读和编辑。若考虑扩展性和兼容性，需要灵活的数据模型和自定义标签，XML 具有更好的扩展性和兼容性。从开发工具和生态系统支持考虑，JSON 和 XML 拥有广泛的支持和成熟的工具链，Protobuf 在某些语言和平台上可能需要额外的代码生成工具和库。在选择适合的数据格式应该根据具体的需求、场景和优势来决定，考虑数据大小、传输效率、可读性、扩展性和开发工具支持等因素。

2.4 数据分离

数据分离是一种设计原则，旨在将数据存储和处理逻辑分离开来，以提高系统的可维护性、灵活性和可扩展性。在数据分离的设计中，数据被视为独立的实体，独立于应用程序的逻辑。它强调将数据存储在一个单独的层或组件中，通常称为数据访问层或数据存储层。该层负责管理数据的读取、写入和持久化，提供一组接口或服务供上层的应用程序使用。

通过将数据与应用程序逻辑分离，在可维护性上，数据的独立性使得修改数据结构或存储方式变得更加容易，而不会对应用程序的逻辑产生太大的影响。这样可以降低系统维护和演化的成本。在灵活性上，通过数据分离，可以灵活地切换或扩展底层的数据存储技术，而不会对应用程序的逻辑造成太大的改动。这样可以根据实际需求选择适合的数据库或存储方案。在可扩展性上，数据分离使得系统可以采用分布式架构，其中数据存储层可以被水平扩展，以满足高并发和大规模数据处理的需求。同时，应用程序层可以独立扩展，以满足不同的性能和可用性要求。在重用性上，通过数据分离，可以将数据访问逻辑封装成可重用的组件或服务，供多个应用程序共享。这样可以避免重复编写相同的数据访问代码，提高开发效率。

数据分离是一种推崇将数据存储和处理逻辑分离的设计原则，通过提供独立的数据访问层来实现。这种设计可以带来可维护性、灵活性、可扩展性和重用性等优势，有助于构建可持续发展和易于管理的系统。

3 框架设计

在框架设计需要涉及到语言的选择和结构设计，开发语言对性能、效率影响重大，需综合衡量。框架结构设计对决定了以后的拓展性，对功能拓展和开发效率有着直接的相关。

3.1 开发语言的选择

开发语言有很多选择，选择开发语言对产品的发展、质量、性能、效率、拓展、灵活、迭代、调试、定制、可配置有着非常大的影响。采用 C++与 Lua 结合一起开发能够很好的平衡以上问题，以下主要会以性能和效率、灵活性和可扩展性、快速迭代和调试、可定制性和可配置性、社区支持和成熟度等问题去逐个分析选择的原因。

在性能和效率上，C++是一种高性能的编程语言，适合处理底层的网络、存储和计算等任务。它可以提供高效的算法和数据结构，直接操作内存，充分发挥硬件的性能。因此，使用 C++可以在关键性能部分实现高效的服务器逻辑。在灵活性和可扩展性上，Lua 作为一种脚本语言，它提供了灵活的脚本编程和动态性的特点。它可以用于编写逻辑和配置脚本，而不需要重新编译和重启服务器。通过将 C++与 Lua 结合，可以在 C++中实现高性能的核心逻辑，而将灵活的业务逻辑和配置放在 Lua 脚本中，实现逻辑和数据的分离，提高服务器的可扩展性和可维护性。在快速迭代和调试上，使用 Lua 作为服务器逻辑的脚本语言，可以实现快速迭代和调试的优势。通过修改 Lua 脚本，可以实时调整服务器的行为和业务逻辑，而无需重新编译和重启服务器。这样可以加快开发速度，方便测试和调试，并且可以更好地与策划、运营等非技术团队进行协作。可定制性和可配置性上，将服务器逻辑实现为 Lua 脚本，使得业务逻辑可以以可配置的方式存在。不同的服务器实例可以加载不同的 Lua 脚本，从而实现不同的逻辑和行为。这样可以根据项目需求和业务变化进行灵活的定制和调整，而无需修改和重新编译底层的 C++代码。社区支持和成熟度上，C++和 Lua 都拥有活跃的开源社区，提供了丰富的库和工具，可以加快开发进度和提供支持。许多成功的项目和引擎都采用了 C++与 Lua 的组合，积累了丰富的经验和最佳实践，使得开发者能够更好地利用现有资源和经验。

采用 C++与 Lua 结合一起开发服务器可以充分发挥 C++的性能和效率，同时利用 Lua 的灵活性、快速迭代和可定制性。这种组合能够实现高性能的底层逻辑与灵活的业务逻辑的结合，提高服务器的性能、可扩展性和可维护性。

3.2 结构设计

在服务端的结构设计上分为三层，内核层、节点层和业务层，如图 3.1。

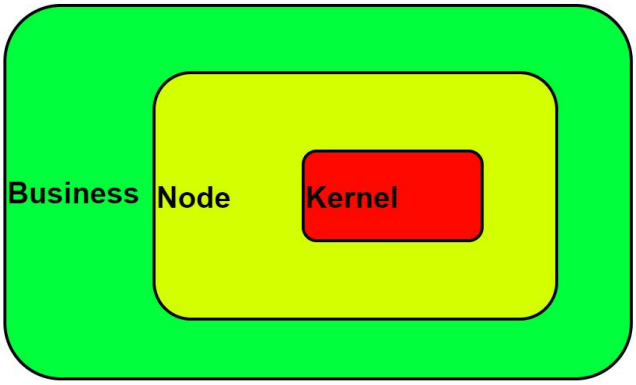

图 3.1 框架结构层级

在内核层中主要是基本功能的提供，如网络、日志、插件与模块的管理器、配置、数据交换格式、事件系统以及大量的工具脚本等等。在节点层提供分布式支持、负载均衡、节点通信等基本功能的支持。业务层是业务上功能的开发，比如游戏玩法逻辑，登录逻辑等。每个层级从里到外依次包含，内层功能不依赖于外层，外层向内层调用时，采用面向接口编程，内层会以接口方式提供给外层。下文会对各层级做出较详细的说明。

4 框架内核层

内核层是基础功能的提供，如网络、日志、配置、插件与模块的管理器、数据格式、基础工具脚本等等，是提供给业务开发人员使用的，它可以直接影响到业务开发人员的开发效率，也决定了该框架的性能效率和功能的拓展性。

4.1 内核层设计

在设计过程参考了 Bigworld 与 UE4 从中引入了一些基本的设计概念，Module、Plugin、Application、Property、Record、Object、GUID、Event。Module 是一类逻辑业务的合集，相对来说功能比较集中，可以做到低耦合，并且可以通过 IOP(面向接口编程)的方式来给其他模块提供耦合功能。Plugin 是一系列 Module 的集合，按照更大的业务来分类。Application 表示一个独立的完整功能的进程，它包含大量插件。Property 表示一维数据，通常用来表示 Object 附带的任意一维动态数据数据结构，当前可以为常用内置数据类型。Record 表示二维数据，通常用来表示 Object 附带的任意二维动态数据结构,结构与 Excel 的二维结构类似，包含 Row 和 Column，并且结构可以通过 Excel 动态传入，记录值可以为常用内置数据类型。Object 表示游戏内动态创建的任意对象，该对象可携带有 Property 和 Record。GUID 用于区别玩家连接或游戏对象的唯一 ID。Event 是游戏逻辑监听和产生事件，用来解耦游戏逻辑。

通过上述引入的设计概念的基础之上，将其内核层的功能设计如下图 4.1 所示

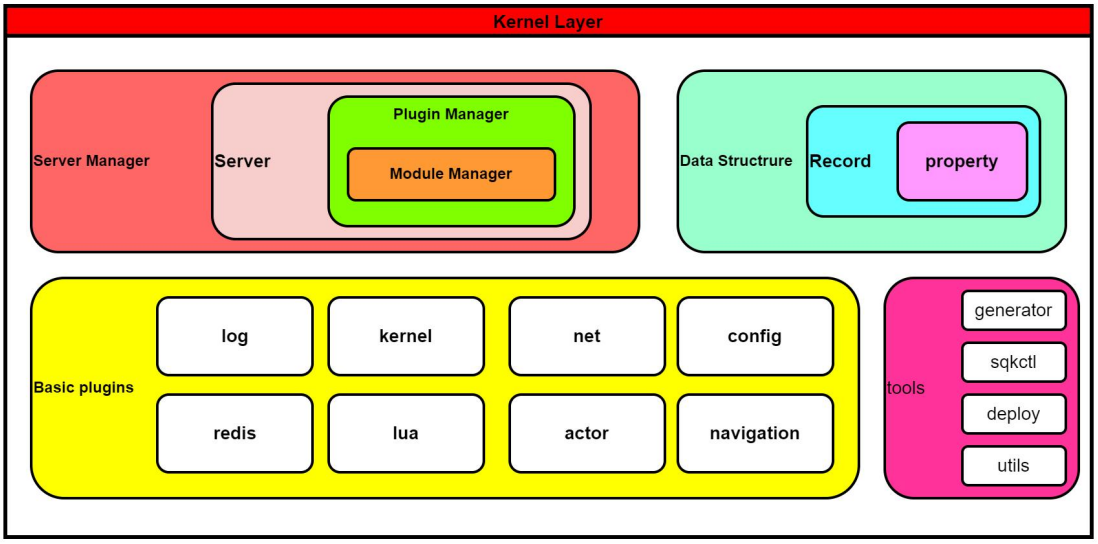

图 4.1 框架内核层结构

在内核层中可以将提供的功能分成四大类，服务管理器、动态数据容器、基础的插件、工具。

4.1 服务管理器

服务管理器是用于管理服务，一个服务管理器中可以管理多个服务，每个服对应一个插件管理器，插件中的模块会执行在插件管理器中。

4.1.1 插件与模块

模块是一类逻辑业务的合集，插件是一系列模块的集合，如下图 4.2 所示，通常插件是一个特定独立的动态链接库。

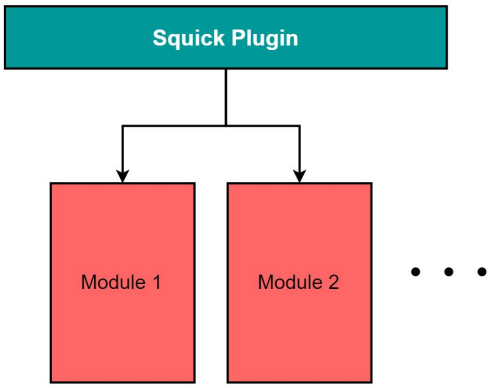

图 4.2 插件与模块

模块管理器负责运行各个模块，它又是嵌入在插件管理器中的。插件可做为一个工程项目，可单独编译成为一个动态链接库文件，类似于 PHP 或者 Apache 拓展的的插件系统，可将功能进行组合拓展，方便单独维护和更新。插件中又以模块为单位进行运行，插件可以管理哪些模块可以运行，哪些模块可以不运行，由开发者自行决定。

虚函数是一种面向对象编程的特性，用于实现多态性，它允许在基类和派生类之间进行动态绑定，使得通过基类指针或引用调用虚函数时，根据实际对象的类型来确定执行哪个版本的函数。虚函数的运用使得 C++实现了面向对象编程中的多态性和动态绑定，提供了灵活性和可扩展性，可以借助这一特性，让模块之间的调用可以通过虚函数来实现。

4.1.2 插件与模块生命周期

插件管理器主要功能是可以管理整个插件的生命周期，包含插件的加载、注册、运行、注销、卸载等，一个插件的生命周期如下图 4.3。

图 4.3 插件生命周期

模块管理器用于管理插件中已经注册模块，可管理的模块整个生命周期，模块的生命周期如下图 4.4。

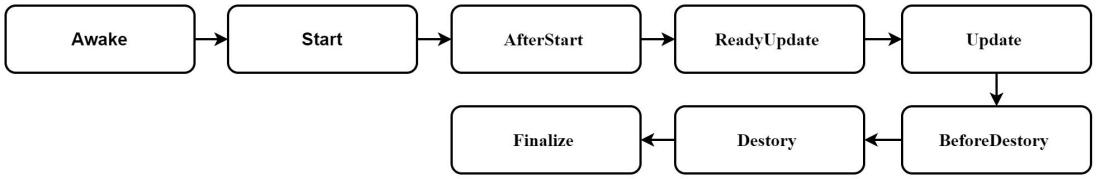

图 4.4 模块生命周期

4.2 动态数据容器

动态数据容器主要包含两种 Property、Record。支持数据的动态类型存储以解决配置表中数据动态类型的存储问题。

Property，是属性的意思，与玩游戏的各种角色属性意思差不多，它是 Key 和 Value 的形式，Key 和 Value 的可以是动态时决定的，可以从配置文件中读取数据并存储在内存中。Property 的 key 和 value 支持的类型有，int、float、long long、string、object、vector2、vector3 等。还支持该存储变量的权限和作用是什么，如公有、私有、保存、缓存、引用、上传等等，它的这些权限，可以在动态时就决定的，比如设置变量 A 是私有的，在创建使，变量 A 需要绑定在一个 Object 之上，那么别的 Object 就不能读取到该 Object 的 A 变量值。

Record，是记录的意思，就像记录账单上，有很多表格，每个表格中有相关的一些重要信息。Record 是 Property 的一个集合，包含了很多个 Property 存储的数据，可以通过 row 和 col 来取出相应的 Property 元素变量。

定义 Property 和 Record 结构，是为了解决动态运行时可以动态改变一个变量类型和权限，它还可以根据 Excel 所配置变量类型与值还有权限从配置中读取到内存中，以达到动态更新内存中的值，这就与之后的配置表息息相关了，在游戏服务器的研发中是不可缺少的一个重要角色。通常游戏公司会分为客户端、策划、服务端，而策划主要的工作就是修改策划表，调参数，客户端和服务端会用到策划给的策划表生成出相应的配置文件，而策划所改的内容更变频率都很高，

为了解决这个问题，就需要程序按照策划或策划遵循程序这边的要求去定义一个变量，该变量的类型、值、由策划表决定。

4.3 基础插件

基础插件包括了配置插件、网络插件、内核插件、Lua 插件，是框架内核层提供给上层的基本功能。

4.3.1 配置插件

配置插件主要功能是能够从 Excel 转化成的 XML 配置文件中读取数据并解析 XML 文件。在解析 XML 文件的时候，可以借助比较流行的开源 XML 解析第三方库 RapidXML 进行实现。RapidXML 是一个快速且易于使用的 C++ XML 解析器库，用于解析和操作 XML 文档。它具有轻量级、高效和简单的特点，适用于处理小到中等规模的 XML 数据。

在配置插件中主要提供了两个模块，ClassModule 和 ElementModule，ClassModule 能够对配置文件进行加载，重加载或配置动态更新。ElementModule 是从内存配置表中读取数据，存储的数据类型是以 Property 或 Record 进行存储，在获取的时候可以动态的去转化类型。如下图 4.5 可以从配置中读取相应的配置信息。

```
std::shared_ptr<IClass> xLogicClass = m_class_->GetElement(excel::Server::ThisName());
if (xLogicClass) {
    const std::vector<std::string> &strIdList = xLogicClass->GetIDList();
    for (int i = 0; i < strIdList.size(); ++i) {
        const std::string &strId = strIdList[i];

        const int serverID = m_element_->GetPropertyInt32(strId, excel::Server::ServerID());
        if (pm_->GetAppID() == serverID) {
            info_.type = (ServerType)m_element_->GetPropertyInt32(strId, excel::Server::Type());
            info_.port = m_element_->GetPropertyInt32(strId, excel::Server::Port());
            info_.max_connect = m_element_->GetPropertyInt32(strId, excel::Server::MaxOnline());
            info_.cpu_count = m_element_->GetPropertyInt32(strId, excel::Server::CpuCount());
            info_.area = m_element_->GetPropertyInt32(strId, excel::Server::Area());
            info_.name = m_element_->GetPropertyString(strId, excel::Server::ID());
            info_.ip = m_element_->GetPropertyString(strId, excel::Server::IP());      You, 4 weeks a
```

图 4.5 配置数据动态读取的部分代码

4.3.2 网络插件

网络插件需要支持大量的网络通信协议，以及拥有顽强的稳定性，本次基于 Libevent 来做该插件的封装, Libevent 是一个开源的事件通知库，它提供了一个轻量级的、跨平台的事件驱动编程接口。它允许开发者编写高性能、可扩展的网络服务器和应用程序，以处理并发的网络连接和事件。且提供了一个抽象层，隐藏了底层操作系统的特定细节，使开发者能够编写与操作系统无关的事件驱动程序。

它支持多种 I/O 事件，包括套接字、管道、定时器和信号等，可以在事件就绪时触发回调函数进行相应的处理。可以编写高效的、可伸缩的网络服务器，而无需手动管理并发连接和事件循环。它提供了事件驱动的编程模型，允许程序响应异步事件，从而避免了阻塞式 I/O 操作带来的性能问题。

在网络插件中，所有协议都是基于 Libevent 提供的协议之上进行，TCP 协议采用的是 Epoll 非阻塞 Reactor 模型，相关代码如下图 4.6。

```
bufferevent_setcb(bev, conn_readcb, conn_writecb, conn_eventcb, (void *)pObject);
bufferevent_enable(bev, EV_READ | EV_WRITE | EV_CLOSED | EV_TIMEOUT | EV_PERSIST);

event_set_log_callback(&Net::log_cb);       You, 2 months ago • Frist Commit

bufferevent_set_max_single_read(bev, SQUICK_BUFFER_MAX_READ);
bufferevent_set_max_single_write(bev, SQUICK_BUFFER_MAX_READ);
```

图 4.6 网络 IO 事件设置的部分代码

网络插件的自定义协议中，是基于 TCP 协议之上做的封装，采用 header+body 形式进行定义新的消息包，header 大小占用 6 字节，头部的前两字节是消息 id 值，用于表示该消息的含义，它由 proto 文件的枚举 ID 来决定，头部的后 4 字节是整个消息的大小（包含头部大小），剩余部分也就是包的 body，它是真正传输的数据内容，消息包的定义如下图 4.7 所示：

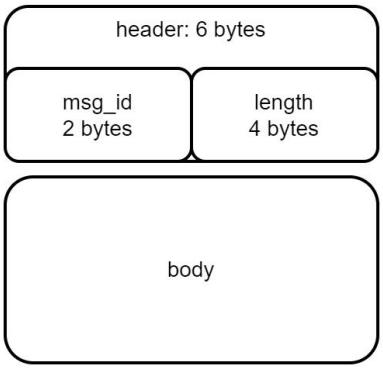

图 4.7 自定义协议结构

在 body 中传输的数据是经过 Protobuf 进行序列化的数据，在这一层，的结构体为：MsgBase，它里面记录了客户端的 ID，传输的数据，还有客户端列表，和一个 Hash 对象 GUID，如下图 4.8

```
message MsgBase {
    Ident player_id = 1;
    bytes msg_data = 2;
    repeated Ident player_client_list = 3;
    Ident hash_ident = 4;
}
```

图 4.8 自定义协议 proto 结构体

平时的包中，后两个字段在客户端与服务端之间的数据包中是不会被用到的，之后服务端与服务端之间转发包的时候，可能会运用到后两个字段。而 msg_data 是 bytes 类型的，里面存储的数据是 Protobuf 序列化后的数据。消息 ID 是由 proto 文件中定义的枚举决定的，用于代表该消息是什么，如下图 4.9

```
// 8000 ~ 10000
enum ProxyRPC {
    PROXY_RPC_NONE = 0;
    REQ_HEARTBEAT = 8001; // 心跳包
    ACK_HEARTBEAT = 8002; // 代理服务器响应

    REQ_CONNECT_PROXY = 8003;
    ACK_CONNECT_PROXY = 8004;
    REQ_DISCONNECT_PROXY = 8005;      You, 2 month
    ACK_DISCONNECT_PROXY = 8006;

    ACK_KICK_OFF = 8010; // 被踢下线
}
```

图 4.9 自定义协议消息 id

消息 ID 就是一个 proto 文件生产相应语言的一个枚举值，只要解析包头部的 msg_id 就知道该包意义。在服务端中服务与服务之间往往都是长连接的，需要实现双向的 RPC，也就是收发包都可以是同样的包体，而不是像 REST 架构，收发包，结构不一样了。消息包已经定义好，只需去监听包的消息 ID 去调用相应的回调函数即可。如图 4.10 所示。

```
bool PlayerManagerModule::ReadyUpdate() {
    m_net_->AddReceiveCallBack(rpc::LobbyBaseRPC::REQ_ENTER, this, &PlayerManagerModule::OnReqPlayerEnter);
    m_net_->AddReceiveCallBack(rpc::LobbyBaseRPC::REQ_LEAVE, this, &PlayerManagerModule::OnReqPlayerLeave);
    m_net_->AddReceiveCallBack(rpc::LobbyBaseRPC::REQ_RECONNECT, this, &PlayerManagerModule::OnReqPlayerReconnect);
```

图 4.10

目前网络插件中支持的协议有 HTTP/HTTPS、TCP、UDP、KCP、Websocket、自定义的消息协议，拥有双端的功能，既可以充当服务端，也可以充当客户端，为业务功能提供了核心的支持，如下图 4.11。

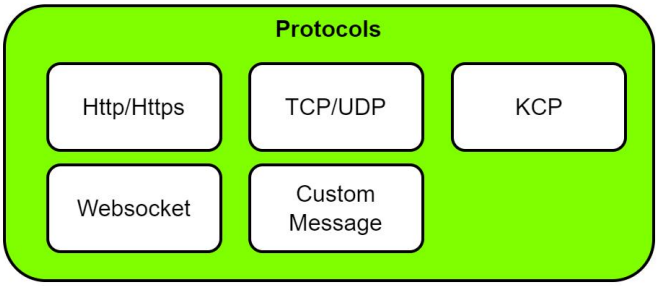

图 4.11 网络插件支持的协议库

4.3.3 内核插件

内核插件是对游戏逻辑和基本库的一些支持，它可以用于管理和处理游戏或应用程序中复杂的虚拟场景的系统。在许多多人在线游戏、虚拟现实应用和模拟系统中，场景系统负责管理游戏世界中的各种对象、地形、物理效果和交互逻辑等元素。以下是内核插件提供的特点和功能：

场景管理：场景系统负责管理游戏中的不同场景或地图，包括场景的创建、加载、销毁和切换。它能够维护多个场景的状态，并支持场景之间的平滑过渡和无缝切换。

对象管理：场景系统管理游戏中的各种对象，如玩家、NPC、怪物、道具等。它负责对象的创建、销毁、位置更新和状态同步等操作。场景系统还需要处理对象之间的交互、碰撞检测和碰撞响应等。

AI 和路径规划：场景系统支持 NPC 和怪物等非玩家对象的人工智能和路径规划。它可以处理 NPC 的行为决策、寻路和动作控制等，使得这些对象在场景中具有自主性和交互性。

客户端自动同步：场景系统负责与客户端进行数据同步，确保玩家在不同客户端上的视觉和交互一致。它可以处理对象的位置、状态、动画和效果等的同步，并提供有效的网络通信和数据压缩。

事件和触发器：场景系统支持场景中的事件和触发器机制。它可以处理玩家的触发行为，触发剧情事件或触发特定的游戏逻辑。场景系统还可以响应系统级的事件，如时间流逝和任务完成等。

4.3.4 日志插件

日志插件采用的是 Easylogging++基础之上封装的一个模块插件，它是一个轻量级的 C++日志库，旨在提供简单易用的日志记录功能。具有简洁的 API 接口和灵活的配置选项，可以方便地集成到 C++应用程序中，帮助开发者进行日志记录和调试。其特点包括，简单易用，它提供了简洁的 API 接口，使得日志记录变得非常简单。只需使用一个宏或函数调用，就可以记录日志消息，并且可以指定不同级别的日志，如调试、信息、警告和错误等。拥有灵活的配置选项的特性，支持了丰富的配置选项，可以通过配置文件或编程方式进行灵活的自定义。可以根据需求定义日志格式、输出目标（如终端、文件）、日志级别过滤、时间戳格式等。跨平台的支持，它可在多个平台上使用，包括 Windows、Linux、Mac OS 等。拥有高性能的 IO，经过精心设计，追求高性能的日志记录。它使用了多种优化技术，如懒惰日志写入、异步日志记录等，以最小化对应用程序性能的影响。在扩展性上支持自定义日志处理器，可以将日志消息发送到自定义的目标，如数据库、远程服务器等。还可以根据需要编写自定义的格式化器和过滤器。

日志模块中提供的接口主要有 5 种类型的日志写入，Debug、Info、Warning、Error、Fatal 等，如下图 4.12 所示。

```
virtual bool LogDebug(const std::string &strLog, const char *func = "", int line = 0);
virtual bool LogInfo(const std::string &strLog, const char *func = "", int line = 0);
virtual bool LogWarning(const std::string &strLog, const char *func = "", int line = 0);
virtual bool LogError(const std::string &strLog, const char *func = "", int line = 0);
virtual bool LogFatal(const std::string &strLog, const char *func = "", int line = 0);
```

图 4.12 日志接口的部分代码

通过不同的类型打印日志会将日志写入到不同的文件，为日志记录提供了基础的保障。

4.3.5 Lua 插件

Lua 虚拟机的设计目标是简洁、轻量且高效，它提供了一种灵活的脚本执行环境，适用于嵌入式系统和应用程序中。通过 Lua 虚拟机，开发者可以利用 Lua 的强大脚本能力来扩展和定制应用程序的行为。Lua 的虚拟机采用的是官方 5.3 版本，该版本在很多项目经过商业验证过，稳定性有较好的保证。而 Lua 插件是基于官方提供的源码之上再次做封装。让其更好的适应到框架里。

由于其他插件采用的是 C++进行开发的，为了达到让 C++对象里的代码与 Lua 层的对象里的代码进行绑定达到互相调用和互相访问变量的要求，在开源库

中，lua-intf 很好的满足了这一需求。lua-intf 是一个 C++库，用于在 C++应用程序中方便地集成 Lua 脚本解释器。它提供了简单易用的 API，使 C++代码能够与 Lua 脚本进行交互，并且支持将 C++对象和函数暴露给 Lua 环境，实现双向的数据和函数调用。如下图 4.13 和图 4.14 是 Lua 层的 log_info 调用 C++层的 LogInfo 函数。

```
You, 2 months ago | 1 author (You)
if true == need_load then
    script_module:log_info("start to reload game scripts as old_version_code == "

    script_module:set_version_code(version_code);
    load_script_file(ReloadList, true);
end
```

图 4.13 Lua 日志打印的部分代码

```
.addFunction("log_info", &LuaScriptModule::LogInfo)
.addFunction("log_error", &LuaScriptModule::LogError)
.addFunction("log_warning", &LuaScriptModule::LogWarning)
.addFunction("log_debug", &LuaScriptModule::LogDebug)
```

图 4.14　C++绑定 Lua 的部分代码

Lua 还有个很好的特性是能够热重载，通过热重载能够对 Lua 代码实现热更。Lua 热更机制是指在游戏运行时，可以动态地更新 Lua 脚本代码，而无需重新启动游戏或重新编译整个游戏程序。这种机制使得开发者能够在不中断游戏的情况下修改和调试游戏逻辑，提高了开发效率。

脚本加载器的实现是游戏启动时，通过一个脚本加载器模块加载初始的 Lua 脚本代码。加载器可以根据配置文件或其他方式确定要加载的脚本文件。如图 4.15 是 C++层加载 Lua 脚本文件并去执行 Lua 中的 init_script_system 函数，图 4.16 是 Lua 层 init_script_system 函数的实现。

```
std::string strRootFile = scriptPath + "/main.lua";

TRY_LOAD_SCRIPT_FLE(strRootFile.c_str());

TRY_RUN_GLOBAL_SCRIPT_FUN1("init_script_system", this);

TRY_RUN_GLOBAL_SCRIPT_FUN0("module_awake");
```

图 4.15　C++中设置 Lua 入口脚本和函数的部分代码

```lua
function init_script_system(xLuaScriptModule)
    script_module = xLuaScriptModule;
    script_module:log_info("Hello Lua, awake script_module, " .. tostring(script_module));
    scirpt_path = script_module:get_script_path()
    print('path: ', scirpt_path)
    package.path = scirpt_path .. '/?.lua;'
    .. scirpt_path .. '/lib/json/?.lua;'
    .. scirpt_path .. '/config/?.lua;'
    .. scirpt_path .. '/server/?.lua;'
    .. scirpt_path .. '/common/?.lua;'
    require("common.init");
```

图 4.16　Lua 的入口函数的部分代码

热更时期，常用方案是在游戏运行期间，可以定期或根据需要检测脚本文件的更新。可以通过检查文件的修改时间戳或使用其他机制来判断是否有新的脚本代码可用。该方案可能会导致脚本文件上传到一半就更新了代码，导致不正常代码的运行。目前方案是主动触发热更，需要人为去确定是否需要热更这一步操作做。这样可以保证在热更前，可以人工检测是否有异常。

数据的共享，为了保持游戏状态的一致性，可以在热更过程中考虑数据的持久化和共享。这可以通过将关键数据保存在外部文件或通过全局变量的方式来实现。

在 Lua 中也有相应的 Lua 模块，它与 C++层的模块稍微有些区别，Lua 层的模块是基于 Lua 插件之上再次运行的一个 Lua 层的模块，为了方便解耦 Lua 层代码而设计出来的结构。Lua 模块的生命周期如下图 4.17。

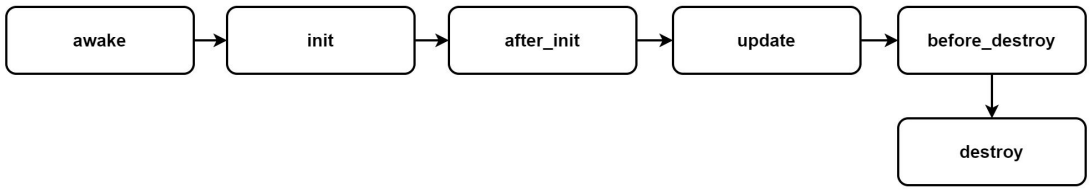

图 4.17　Lua 模块的生命周期

4.4 基础工具

基础工具可以帮助开发者将复杂的流程变简单化，基础的工具主要已自动化脚本编写实现，简化开发者的繁琐步骤。基础工具主要分为 Generator 和 sqkctl。

4.4.1 Generator

Generator 是一个生成器工具的集合，它一个进行代码的生成，比如将 proto 代码转化成 C++、C#、Lua 代码。也可以将 CmakeFile.txt 转化成 VS 能够打开的.sln 文件。也可以将编译出来的的二进制文件进行打包，到单独的文件夹里，方便在

无依赖的情况下放在其他主机上进行运行。它主要是由一系列脚本或工具组成，方便进行批量化或自动化的运行。如下图 4.18：

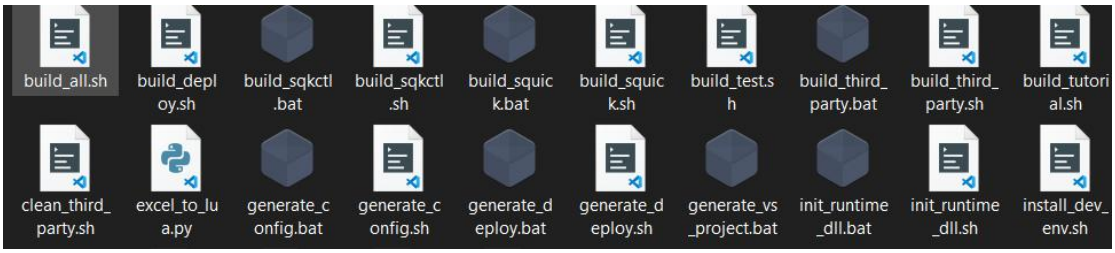

图 4.18　Generator 工具脚本集

4.4.2 sqkctl

sqkctl 是项目管理工具，为了方便让框架核心代码与项目工程代码实现解耦管理，提供了部分命令在工程中使用，例如项目工程的初始化，框架代码更新，对框架代码打 patch，对比等等，类似于 git 的操作对版本进行管控，目的不是为了版本的管控，而是让项目代码与框架代码进行在管理层面进行解耦。这样对于后期升级框架核心代码的时候就比较方便，diff 下存储的就是改动的代码，只要改动的核心文件不是升级中改动的文件，框架代码就可以升级了，采用 update 命令即可实现，该命令也会校验文件是否冲突，如果出现文件冲突，那么说明升级的文件和改动的文件是同一个文件，可能强制升级之后会出现报错。

5 框架节点层

节点层是将各个服务功能划分到不同节点上运行，可以让工作节点能够横向拓展，降低单点服务器的压力，从而达到最大负载的上限，节点层的设计可以使得工作计算分布式化，必要时可以达到弹性伸缩，动态扩容的目的，是分布式计算服务的前提基础。

5.1 节点介绍

节点可以称之为一个独立的服务，它允许遵循它所提供的协议进行连接，若授权之后，可以与它进行正常交互。节点类型（角色）有很多种，主要还是总体的游戏服务功能进行划分的，如 master、login、world、lobby、micro、game、gameplay、gameplay_manager、proxy、db_proxy 等等。部分节点可以支持横向拓展，支持负载均衡的运算与任务分配。如下图 5.1 所示是各种类型的节点连接结构图。

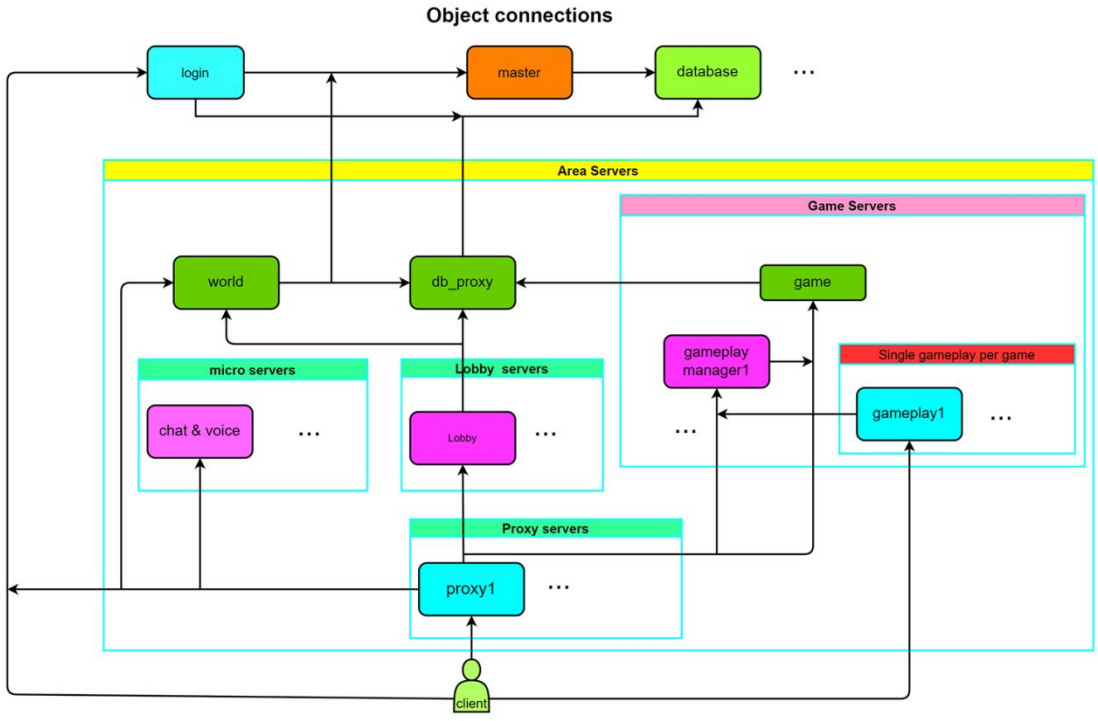

图 5.1 对象连接关系

客户端能连接的节点主要是涉及三个，login、proxy、gameplay，而 gameplay 节点是否能够连接还取决于 gameplay_manager 节点中定义是独立进程启动 gameplay 还是运行在 gameplay_manager 管理机中的。

master 节点主要是用于管理分布式的子节点和后台管理的，它一般情况下是单开的。

login 节点，是为了给客户端提供登录服务，在登录成功后，会将 token 和登录信息记录到 Redis 缓存中，以便 proxy 节点节点进行 token 验证和获取登录者基本信息，还根据负载均衡算法匹配到最佳的 proxy 节点，将节点信息发送给客户端进行连接，它一般情况下是单开的。

proxy 节点，它提供代理服务，与玩家和内部节点进行连接，客户端初始连接的时候需要进行授权，也就是先需要 login 节点的登录验证，proxy 才能对消息进行转发到内部节点，它主要职责就是消息验证、消息转发、心跳检测，这样内部节点就可以不用对消息包进行验证，能够转发到内部节点的消息都是可信的。它可以跨物理机进行多节点部署以达到提升总体服务程序的最大承受的负载能力。它对客户端支持 HTTP、TCP、UDP、KCP、还有自定义消息的 RPC 协议。

world 节点，相当于是一个游戏的区服，它也可以管理 proxy、lobby、micro、game、game、gameplay、gameplay_manager 等节点。同步整个区所涉及连接的所有节点。单区只能部署一个 world 节点。

db_proxy 节点，它是一个数据库的代理节点，是 Mysql、Redis、MongoDB 数据库接口的一个封装，存储操作主要集中在该节点上进行，对外已网络接口的形式提供给其他内部节点使用。一个区一般情况下只开一个 db_proxy 节点。

lobby 节点，它主要是为游戏提供基本的大厅逻辑功能，比如商店的交易、活动、背包物品、抽奖这些，它是玩家登录成功后在大厅界面中主要提供的服务。它可以支持多开部署。

game 节点，它提供了房间相关的逻辑，比如组队，还有匹配等等，它与 gameplay_manager 进行紧密交互，在游戏开局的时候，由 game 节点通知 gameplay_manager 创建新的 gameplay 实体运行，在结束的时候是反向通知的一个行为。它在一个区中只能支持单开。

micro 节点，它是提供一些基本的微服务，比如语音、聊天、或者其他可以解耦的服务都可以放在 micro 节点之中，它支持多开运行。

gameplay_manager 节点，用于管理 gameplay 节点或自主运行 gameplay 部分的逻辑，可负责创建 gameplay 节点实例、销毁、监控，也充当 gameplay 节点与 game 节点之间的代理。在 gameplay 节点连接 gameplay_manager 后，根据 key、account、server_id 来与 game 节点进行连接。

gameplay 节点，是一个独立出来专门为游戏逻辑运行的节点，它可以是任意语言编写的服务，也可以是由 UE4 或 Unity3d 进行开发专用服务器，它由 gameplay_manager 节点进行管理，在每一场对局中需要由 gameplay_manager 启动一个独立进程的 gameplay 节点用于跑该场游戏逻辑，它的占用的网络端口是

开放的，玩家客户端可以直接连接。它可以多部署，具有临时性，游戏结束即会销毁该实例。

5.2 节点基础功能

节点的基础功能是节点所拥有的必要功能，支持动连接与管理，转发表，负载均衡，星跳检测和断线重连是每一

5.2.1 动态连接与管理

动态连接是在节点运行的时候，能够进行连接，并在对方的记录自己的节点信息，方便被连接的节点能够请求所连接的节点。在连接的时候，会通过统一实现的函数 INodeBaseModule::AddServer 进行连接。

在连接的时候，节点会将自己的基本信息告诉给被连接的节点，授权成功后，被连接（上游）的节点上也会保存连接节点的基本信息，比如负载情况，连接情况，CPU，节点 ID，设备信息等等，之后可以通过节点 ID 实现双向的通信，连接的节点也会定时的报告自己的状态给上游节点。上游的节点会有权限管理下游的节点，并且可以操控下游节点去连接其他节点。

5.2.2 负载均衡

负载均衡算法是用于在分布式系统中将请求合理地分配给多个服务器以实现负载平衡的算法。

在 proxy 节点分配上，login 分配 proxy 的负载均衡算法是根据最少连接数来分配 proxy 节点，proxy 节点在玩家连接成功后会保存玩家的连接数，该连接数会定时上报给 login 节点，在新玩家授权登录的时候，login 节点会根据该连接数判断最少连接数的 proxy 节点给新玩家。

在其他内部节点上分配上，proxy 会定时的去同步 world 节点的信息，知道内部所有的节点的状态，会根据最少负载量来决定让玩家的请求由谁去处理。在新玩家连接 proxy 节点授权成功后，proxy 将会计算最小负载相应类型的节点，并存储匹配到节点的 id 与玩家 id 进行绑定，之后玩家的包就会转发至相应的节点上去处理请求。

5.2.3 转发表

分布式节点转发表是分布式系统中的一个重要组成部分，它的作用是记录和管理节点之间的通信路由信息。它支持节点之间的转发，布式系统中的节点之间需要相互通信和交换信息。分布式节点转发表记录了节点之间的路由信息，包括源节点和目标节点的地址信息以及相应的通信路径。通过转发表，节点可以根据目标节点的地址将消息正确地路由到目标节点，实现节点之间的通信和信息交换。

一般分布式系统中存在大量的节点和任务，而每个节点的处理能力和负载情况可能不同。分布式节点转发表可以根据节点的负载情况和性能指标，将请求和任务合理地分配到不同的节点上，实现负载均衡。通过动态调整转发表中的路由

信息，系统可以根据节点的负载情况自动选择最合适的节点进行转发，从而提高系统的整体性能和吞吐量。转发表可以记录节点之间的备用路径和故障恢复策略，以应对节点故障和网络中断等情况。当某个节点发生故障时，转发表可以自动切换到备用路径，将消息转发到可用的节点，从而保证系统的可靠性和可用性。

转发表还可以用于管理和维护分布式系统的网络拓扑结构。通过记录和更新节点之间的通信路径和连接信息，转发表可以帮助系统管理员了解系统的网络拓扑结构，并进行拓扑管理、故障排查和网络优化等工作。如下图 5.2 是节点的玩家表和服务表，这两个表都是转发表。

```
// 玩家表
map<Guid, PlayerProxyInfo> players_;

// 服务表
map<int, ServerData> servers_;
```

图 5.2 转发表

5.2.4 心跳检测

心跳检测是一种常见的机制，用于监测设备或服务的存活状态。在框架中，心跳检测通常定期发送和接收心跳信包来实现。以下是的心跳检测机制：

1.发送心跳包：客户端或服务端定期发送心跳包。心跳包通常是一个简短的消息或数据包。

2.接收心跳：接收端，如客户端或服务端，监听来自发送端的心跳包，并更新对方状态。

3.超时检测：接收端每隔一段时间进行检测，以确认发送端的存活状态。如果在规定的时间内没有收到心跳包，则可以认为发送端可能存在问题或不可用。

4.响应心跳：接收端可以发送响应包作为回应，通知发送端心跳包已收到。这有助于验证通信的双向性，并在必要时进行故障排查和故障切换。

5.故障处理：当发送端的心跳包未能及时到达或未收到接收端的响应时，可以认为发送端可能存在故障或不可用。接收端可以根据心跳检测结果采取相应的离线或故障处理措施。

心跳检测机制的作用包括：存活状态监测：通过心跳检测，可以实时监测设备或服务的存活状态，及时发现故障或不可用的情况。

在部分节点与节点之间拥有心跳检测可用于探测对方的一个状态，在节点出现故障时，可进行故障切换和容错处理。

在内网节点上，每个节点会定时的发心跳给上游节点，报告当前的节点状态，上游根据心跳报告来判断下游节点是否断开，及时的更新自己的转发表，而节点也通过心跳检测保证上游节点的存活，在收到任务时可以选择性的交给存活工作

负载最小的节点去计算。在 proxy 节点上，对象多了个玩家，也需要对玩家进行心跳检测，如果玩家没有在规定时间内发送心跳包，则可以判断该玩家网络不正常或已断开连接。可以判断连接是否断开或失效，客户端也可以进行相应的重连或重建连接操作。心跳处理逻辑，收到心跳包是，会更新 last_ping 为当前时间，如图 5.3 所示。心跳检测，每隔 10 秒回检测一次上一次心跳时间是否超时，如图 5.4 所示。

```
void LogicModule::OnHeartbeat(const socket_t sock, const int msg_id, const char *msg, const uint32_t len) {

    std::string msgData(msg, len);
    NetObject *pNetObject = m_net_->GetNet()->GetNetObject(sock);
    if (pNetObject) {
        auto iter = clients_.find(pNetObject->GetClientID().ToString());
        if (iter != clients_.end()) {
            iter->second.last_ping = SquickGetTimeMS();
            // dout << "heatbeat: " << iter->second.last_ping << std::endl;
        }
    }

    m_net_->SendMsgWithOutHead(rpc::ProxyRPC::ACK_HEARTBEAT, msgData, sock);
}
```

图 5.3 心跳包处理代码

```
int LogicModule::OnHeatbeatCheck(const Guid &self, const std::string &heartBeat, const float time, const int count) {
    auto iter = clients_.find(self.ToString());
    if (iter == clients_.end()) {
        dout << "No this player to heatbeat check\n";
        return 1;
    }
    // dout << "heatbeatcheck : " << self.ToString() << "  " << iter->second.last_ping << "  now " << SquickGetTimeMS(
    time_t now = SquickGetTimeMS();
    if (now - iter->second.last_ping > 10000) { // 大于8秒即断线
        OnClientDisconnect(iter->second.sock);
    }
    return 0;
}
```

图 5.4 心跳定时检测代码

5.2.3 断线重连

在节点与节点之间出现断线，在通过心跳检测到断线时，节点会根据配置表 3 秒后再次重连，直到连接成功为止。

玩家与 proxy 层的断线情况就稍许复杂，主要分两种情况，运行中和意外退出。运行中可能出现弱网络或后台运行，弱网络是网络特别卡，卡网络断开连接了，后台运行网络没有断线，但没有进行收发消息；第二种是意外退出，客户端直接关闭掉或闪退。

弱网络或后台运行，若已进入游戏大厅或正在游戏中网络断开连接了，需要客户端自行判断网络是否断开，这里分三种情况讨论。原因 1：网络已断开，已进入游戏大厅。方案 1：通过连接秘钥请求连接 proxy 节点，请求进入大厅。原因 2：已进入游戏网络已断开，方案 2：通过连接秘钥请求连接 proxy 节点，请求加

入游戏。原因 3: 后台运行游戏网络没有断开，方案 3: 该情况下网络没有断开连接，该过程不需要重新连接 proxy 节点，服务器采用非阻塞发送数据包，为了发送效率，在多次未成功发送的数据包，会舍弃掉该包，客户端在网络差的情况下有一定概率收不到数据包，需要重新调用进入大厅或加入游戏实现同步。

客户端意外退出时，客户端是直接关闭了，客户端重新进入时先判断登录秘钥是否过期，过期重新登录，没过期再判断代理连接秘钥是否过期，过期重新选择选择区服进入区服，没过期直接连接 proxy 节点，请求进入大厅。如果存在游戏对局，那么响应包中会有游戏对局的信息，再次加入游戏即可。

TCP 意外断开连接的情况分析，在做断线重连测试的时候，发现客户端如果采用关闭网卡这种方式来断开连接，服务端并没有立即断开连接，而是向正常连接一样保持连接，差不多过了几十秒才断开连接，这种会导致在等待的这几十秒，为了让服务器更快速感应客户端状态，应采用心跳包来解决此问题。

理想状态下，一个 TCP 连接可以被长期保持，但是实际情况下，一个看似正常的 TCP 连接，可能已经断连。两个主机之间通讯，往往需要通过多个中间节点，如：路由器、防火墙等。因此两个主机 TCP 连接保持同样受中间环节影响。断连的 TCP 连接已经没有意义了，但是维护这样的连接，可能会浪费服务器的系统资源，尤其是内存和 socket 资源，客户端一般还好，因此需要服务器采取相应的探测措施，也就是心跳检测。

为了避免以上情况的发生，客户端与 proxy 节点采用心跳包来保持连接，如果超过一定时间，客户端未向服务端发送心跳包，说明客户端已断开连接。在客户端连接 proxy 节点成功后，proxy 节点会每 10 秒检测客户端有没有刷新心跳时间搓，如果上一次心跳请求时间距现在超过 10 秒，则默认客户端已断开网络，这种办法是避免客户端网络严重卡顿，或者直接关掉网卡的这种情况，默认 TCP KeepLiveTime 是 2 小时，这个时间对于实时检测状态不是很友好。

5.3 通信机制

对象之间的通信类型主要分为两种，第一种是节点与节点之间的通信，另一种是节点与玩家之间的通信。

节点与节点之间的通信，也可以分成两种，一种是直接通信，只需知道对方节点的 id 即可通信，但这种通信的前提条件是，需要两个节点先建立连接。另一种通信是通过代理中转，代理模拟了玩家进行请求另一个节点，该方式两个节点之间可以不用直接连接，但都需要与代理连接上的。如下图 5.5。

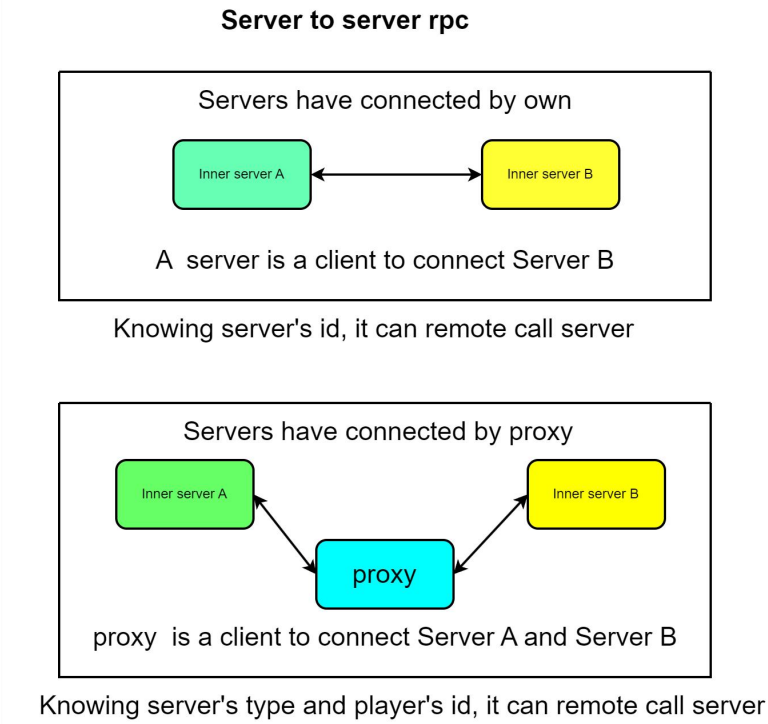

图 5.5 节点间的 RPC 通信

节点与玩家之间的通信，如果玩家是主动与内部节点通信，则需要代理进行转发，代理会查找玩家表玩家所绑定的节点，转发给相应的节点去处理请求。如果是节点主动发包给玩家，节点只需将玩家 ID 和包发给代理，由代理去找玩家表找到玩家所绑定的 socket id，通过 socket 发包给玩家。

6 框架业务层

业务层是指在软件系统中负责处理业务逻辑和业务流程的一层。它位于架构中的节点层之上。它主要职责包括，负责处理与具体业务相关的逻辑，从数据访问层获取的原始数据进行转换和处理，使其符合业务需求。它可能需要进行数据的组合、过滤、排序、计算等操作，以提供给用户界面层或其他业务逻辑使用。在涉及到多个数据库操作的场景中，在开发中，如 gameplay，背包系统，大厅活动逻辑等等，这些逻辑，是需要在业务层进行实现，业务层强调的是根据业务需求实现的逻辑，不需要深入了解或修改内核层节点层上的逻辑即可完成的需求，一般这种业务需求都具有普遍性，都是可以在建设好基础框架之上，业务需求都能基于该框架较容易的实现出来。如下图 6.1 是一款 VR 下棋类游戏截图（该业务代码没有开源）。

图 6.1　VR 下棋类游戏

6.1 业务节点

业务层的逻辑需要基于节点层提供的节点之上进行开发，节点层提供了常用的业务场景节点，如 lobby、game、db_proxy、gameplay 等节点。开发者可根据节点职责去开发相应的业务逻辑。这里列出几个业务逻辑开发中会用到的节点。

lobby 节点，它职责主要是为游戏提供基本的大厅逻辑功能，可以商店的交易、活动、背包物品、抽奖相关逻辑。

db_proxy 节点，是数据库的代理节点，是 Mysql、Redis、MongoDB 数据库接口的一个封装，玩家的数据读取、存储这些操作集中在该节点上实现。

game 节点，它提供了房间相关的逻辑，比如组队，还有匹配等等。

micro 节点，比如语音、聊天、或者其他可以解耦的服务都可以放在这上面

gameplay_manager 和 gameplay 节点，专为游戏玩法而设定的节点，游戏玩法中的所有逻辑都基于这两个节点上进行的，里面提供了 IGameplay 基类，很方便的位开发者自定义游戏状态机，轻松实现游戏玩法。

6.2 业务项目解耦

在游戏公司中，一般游戏项目有很多个，而框架采用单一的框架，如果每个项目单独维护一套框架，维护成本是比较高的，必要时，可以将框架从业务项目中独立出来，让业务层代码与节点层和内核层进行分离，将一个服务端项目分离为业务代码和框架代码，业务代码是根据当前的业务需求而写的代码，框架代码是提供基础设施的代码。在内核层的工具之中，有一个工具 sqkctl 可以很容易的方便解耦业务层代码与基本框架的代码，它可以不止是解耦，还可以在业务项目之中对内核层和节点层代码进行打 patch 操作，这样达到了从外部修改框架内的代码。这么做的目的是因为商业项目中，项目有很多个，但真正里面涉及的核心技术都是单一的，就像在游戏开发中，目前普遍游戏都是用 Untiy 或 Unreal 引擎来制作的，本次研究的框架就好比这两款引擎，制作的游戏代码呢就好比业务层的代码。如图 6.2 和图 6.3 分别是业务项目和框架项目。

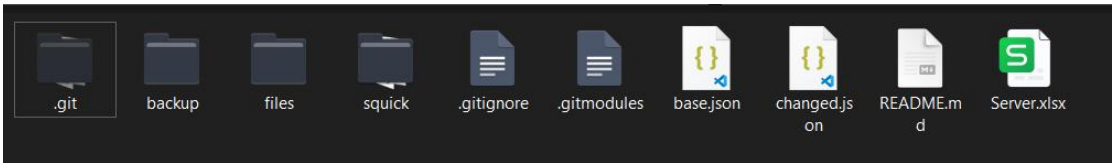

图 6.2 业务层项目

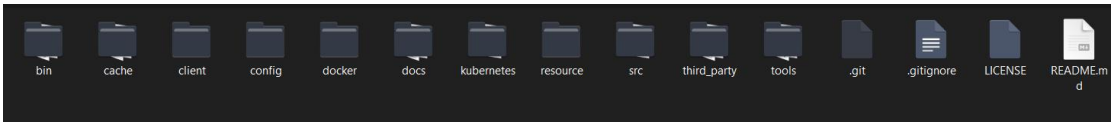

图 6.3 框架层项目

7 打包工作流程

7.1 打包工作流程介绍

打包工作流程是将项目代码转化成为所需要的程序产品，之前了解的设计、开发方面的知识，现在以打包或者测试者的身份去完成任务，也就是编译、修改配置、打包、运行等步骤，就像工厂里，不需要关注生产机器的原理，只需要按步骤操作，也可以将最原始的测试东西生产出来。无需关注产品是怎么设计与实现的，只需将产品按流水步骤生产出来即可。

本框架的项目工程文件如下图 7.1，作为流水线打包测试，只需关注 deploy、config、tools、resource 文件夹即可。

```
deploy:   // 服务端生成可部署文件
config:   // 服务端配置
data:     // 服务程序储存数据
bin:      // 服务端程序
tools:    // 工具
src:          // 主要源码文件夹
        lua:      // lua脚本代码
    server:   // 各服务器代码
    squick:   // 核心代码
    tester:   // 测试代码
    tools:    // 工具代码
    tutorial: // 教学示例代码
    test:     // 测试代码
    proto:   // protobuf代码
    www:          // 网站系统代码
        admin:    // 后台前端代码
        server:   // web服务端代码
        website:  // 官网前端代码
third_party:  // 第三方代码
cache:        // 编译时的临时文件
others:       // 其他
```

图 7.2 框架目录结构

deploy 是部署目录，服务端生成可独立运行的文件集，其中包含了可执行的文件、脚本、配置文件等等，用于直接上传到服务器上运行，好比 Unity 或 UE4 打包出来的文件一样。

config 是配置文件目录，里面包含了日志配置文件、插件配置文件、Excel 生成的配置表等等。struct: sqkctl 将{project_path}/resource/excel 下的所有 xlsx 文件转化成的 xml 文件，主要记录配置表中字段的属性，其中包含了，名称、描述等这些信息。config/ini 目录是 sqkctl 将{project_path}/resource/excel 下的所有

xlsx 文件转化成的 XML 文件，主要记录配置表中目前有哪些内容。config/plugin 目录是各种服务器的插件配置文件，config/log 目录是服务器的日志配置文件。

tools 是工具集目录，里面包含了各种自动化的脚本，方便对程序进行 proto 代码生成、vs 工程生成、编译文件清理、打包、还有部署等自动化脚本。

resource 目录是资源目录，里面也包含了基本的配置表信息，还有数据库的 sql 文件以及打包部署为提供程序运行时的基本管理脚本文件。

7.2 基础环境

这里演示的是在 Windows 操作系统下的项目基础环境搭建，Linux 搭建与部署步骤请查看开源的详细文档介绍。推荐在 Windows10 以上环境，采用 Virsual Studio 2019 以上 编辑器进行开发。在开发或打包之前，需要下载相应的工具 CMake。首先先进行第三方库编译，这里用已编译第三方库的文件进行拷贝即可。将 https://github.com/pwnsky/SquickThirdPartyBuild/tree/main/Windows/ 下的 build 拷贝到 {project_path}/third_party 目录下即可不用编译第三方源码。

先编译 sqkctl 工具，进入到工具目录{project_path}/tools，点击 build_sqkctl.bat 即可编译。编译完毕后，点击 generate_config.bat 生成相应的配置文件和代码文件。采用管理员权限运行 register_env.bat 注册 squick 的环境变量（注：其余的脚本都是普通权限运行，只有 register_env.bat 需要管理员权限）。

之后再点击 generate_vs_project.bat 生成 vs 项目工程在{project_path}/cache 下，打开{project_path}/cache/Project.sln 进行全部编译。将 node/squick_exe 项目设置为启动项，并修改调试的工作目录为 {project_path}/bin，也可以设定相应参数调试不同服务器，之后就可以采用 VS 来启动调试全部服务器了。实现 N 变 1 进行调试，启动界面如下图 7.3。

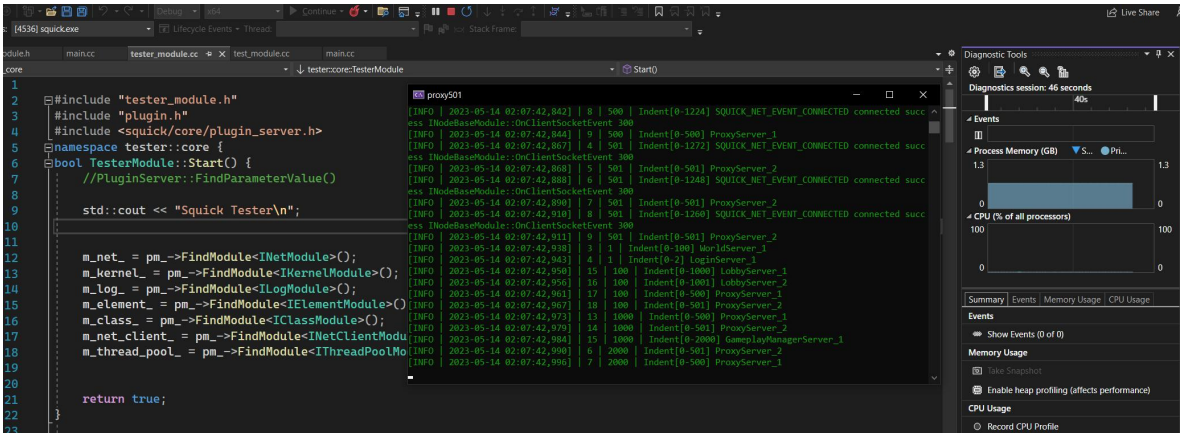

图 7.3 调试运行界面

7.3 生成配置

生成配置之前，可以在 resource/execl 目录下通过 Excel 编辑器修改 xlsx 文件，比如打开图 7.3 中的 Server.xlsx 文件后为图 7.5 所示。

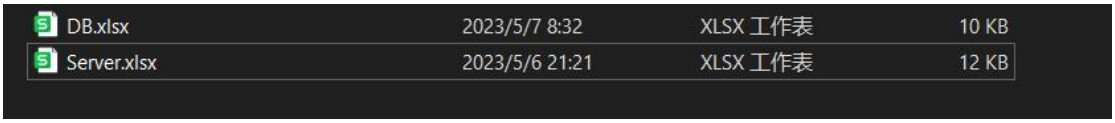

图 7.4 服务配置表文件

MasterServer_1	1	5000	1	127.0.0.1	192.168.150.100	10001
LoginServer_1	2	5000	1	127.0.0.1	192.168.150.100	10010
WorldServer_1	100	5000	1	127.0.0.1	192.168.150.100	10101
DbProxyServer_1	300	5000	1	127.0.0.1	192.168.150.100	10301
ProxyServer_1	500	5000	1	127.0.0.1	192.168.150.100	10531
ProxyServer_2	501	5000	1	127.0.0.1	192.168.150.100	10505
LobbyServer_1	1000	5000	1	127.0.0.1	192.168.150.100	11000
LobbyServer_2	1001	5000	1	127.0.0.1	192.168.150.100	11001
GameplayManagerServer_1	2000	5000	1	127.0.0.1	192.168.150.100	12000
GameplayManagerServer_2	2001	5000	1	127.0.0.1	192.168.150.100	12001
CdnServer_1	3000	5000	1	127.0.0.1	192.168.150.100	13000
CdnServer_2	3001	5000	1	127.0.0.1	192.168.150.100	13002
RobotServer_1	4001	5000	1	127.0.0.1	192.168.150.100	14000

图 7.5 服务配置表

在上图 7.5 所示中，里面信息存储得有各节点的 IP 还有绑定端口等信息，那么可以通过修改配置表的方式达到各节点服务的基本配置。在修改完毕之后，还需要将 Excel 的 xlsx 文件转化成 XML 文件。需要执行 tools/generate_config.bat 脚本。

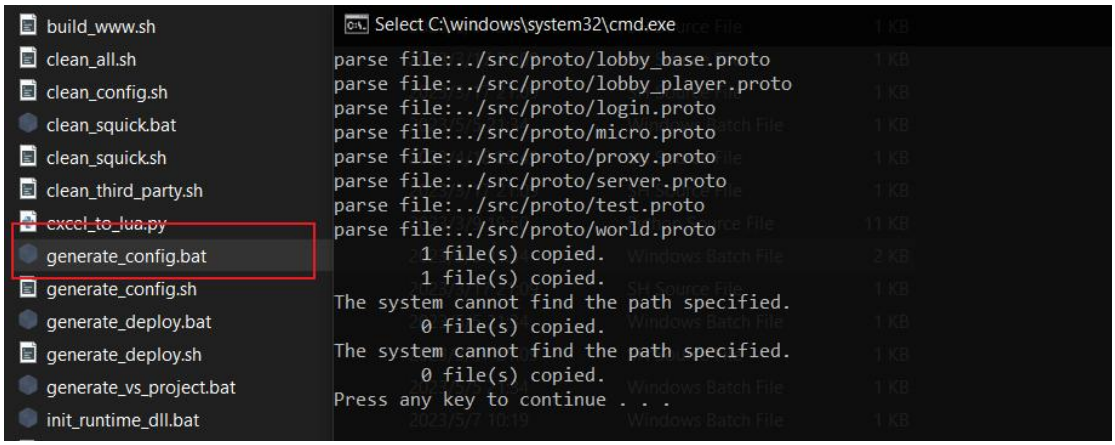

图 7.6 打包生成

生成出来的 XML 文件如图 7.7 所示。

```
<?xml version='1.0' encoding='utf-8' ?>
<XML>
        <Object Id="CdnServer_1" Area="1" CpuCount="1" IP="127.0.0.1" Key="server_wrold_key" MaxOnline="5000" Port="13000" P
        <Object Id="CdnServer_2" Area="1" CpuCount="1" IP="127.0.0.1" Key="server_wrold_key" MaxOnline="5000" Port="13002" P
        <Object Id="DbProxyServer_1" Area="1" CpuCount="1" IP="127.0.0.1" Key="server_wrold_key" MaxOnline="5000" Port="1036
        <Object Id="GameplayManagerServer_1" Area="1" CpuCount="1" IP="127.0.0.1" Key="server_wrold_key" MaxOnline="5000" Pc
        <Object Id="GameplayManagerServer_2" Area="1" CpuCount="1" IP="127.0.0.1" Key="server_wrold_key" MaxOnline="5000" Pc
        <Object Id="LobbyServer_1" Area="1" CpuCount="1" IP="127.0.0.1" Key="server_wrold_key" MaxOnline="5000" Port="11000"
        <Object Id="LobbyServer_2" Area="1" CpuCount="1" IP="127.0.0.1" Key="server_wrold_key" MaxOnline="5000" Port="11001"
        <Object Id="LoginServer_1" Area="1" CpuCount="1" IP="127.0.0.1" Key="server_wrold_key" MaxOnline="5000" Port="10010"
        <Object Id="MasterServer_1" Area="1" CpuCount="1" IP="127.0.0.1" Key="server_wrold_key" MaxOnline="5000" Port="10001
        <Object Id="ProxyServer_1" Area="1" CpuCount="1" IP="127.0.0.1" Key="server_wrold_key" MaxOnline="5000" Port="10531"
        <Object Id="ProxyServer_2" Area="1" CpuCount="1" IP="127.0.0.1" Key="server_wrold_key" MaxOnline="5000" Port="10505"
        <Object Id="RobotServer_1" Area="1" CpuCount="1" IP="127.0.0.1" Key="server_wrold_key" MaxOnline="5000" Port="14000"
        <Object Id="WorldServer_1" Area="1" CpuCount="1" IP="127.0.0.1" Key="server_wrold_key" MaxOnline="5000" Port="10101"
</XML>
```

图 7.7 服务配置文件

7.4 打包部署

打包部署是将编译出来的文件在无依赖的情况下可以让打包出来的文件放在其他同操作系统类型的机器上运行，在生产线中，它好比是最终做出来的产品一样，可以独立于工厂设备环境发挥出产品的价值。在编译成功之后，可以运行 tools/generate_deploy.bat 脚本，会自动拷贝所依赖的所有文件在 deploy 目录下，打包出来的程序，也就是 deploy 目录下的所有文件。

图 7.8 打包出来的文件

这样就可以在同一类型的机器上独立运行这些程序了，这里运行 multi_start.bat 脚本，每一个节点服务启动一个独立的进程，实现 1 变 N 的操作。如下图 7.9。

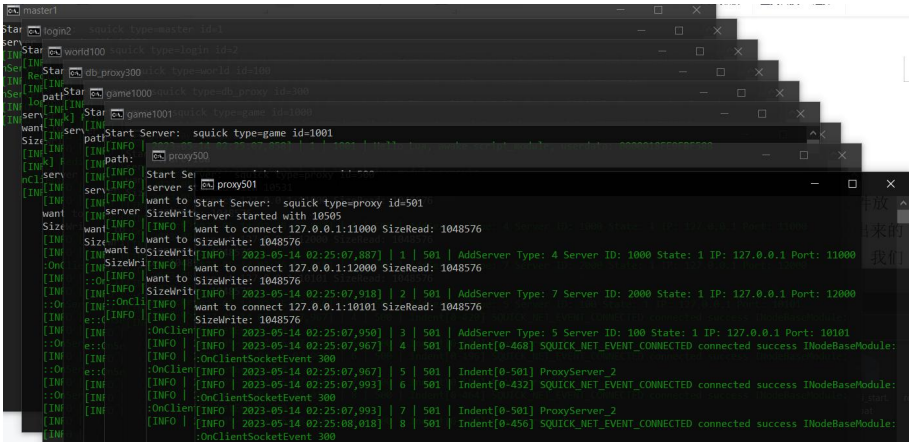

图 7.9 多进程启动

8 压力测试

8.1 压力测试介绍

服务端压力测试是评估服务器在高负载情况下的性能和稳定性的过程。通过模拟多个并发用户请求，可以确定服务器在处理大量请求时的性能极限和瓶颈。以下是进行服务端压力测试的一般步骤：

1.定义测试目标：明确测试的目标和需求。确定要测试的服务器的哪些方面，比如处理能力、并发连接数、响应时间等。

2.创建测试环境：搭建一个与生产环境类似的测试环境，包括服务器、网络设置和所需的软件配置。确保测试环境能够准确反映实际生产环境的特征。

3.选择合适的测试工具：选择适合的压力测试工具，例如 webbench 模拟 HTTP 请求，它提供了模拟多用户并发请求的功能，并能够收集和分析性能数据。如果是自定义协议，还需要开发者自行去开发测试工具来模拟请求。

4.定义测试场景：根据实际应用场景和预期负载，创建一个或多个测试场景。场景可以包括模拟不同类型的请求、并发用户数量、请求频率等。

5.配置测试参数：根据测试需求，设置测试工具的参数，如并发用户数、循环次数、请求延迟等。确保测试参数能够反映实际使用情况。

6.运行测试：运行压力测试工具，模拟并发用户的请求。观察服务器的响应情况，包括请求成功率、响应时间、错误率等指标。

7.监控和分析：在测试运行期间，监控服务器的资源利用率、内存使用情况、网络带宽等指标。收集和分析测试结果，查找性能瓶颈和问题。

8.优化和重复测试：根据测试结果进行性能优化，可能需要调整服务器配置、代码优化或其他措施。然后，重新运行测试，比较优化后的性能指标。

9.结果报告：根据测试结果生成详细的报告，包括性能指标、瓶颈分析、建议改进等。报告应该清晰地概述服务器在压力下的表现和可能的限制。

我的测试环境是在联想 y9000p 2022 款机器上，操作系统为 Windows10，内存 16GB，CPU 为 Intel i7-12700H，显卡为 3070TI。

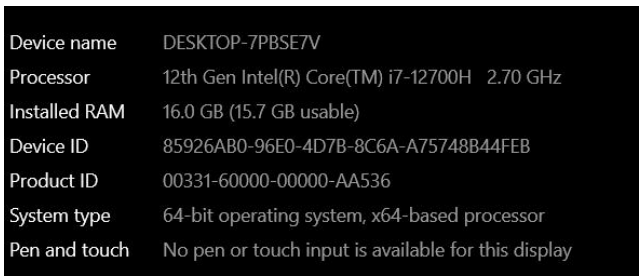

图 8.1 电脑配置

网络环境为 mix4 手机热点 WIFI。

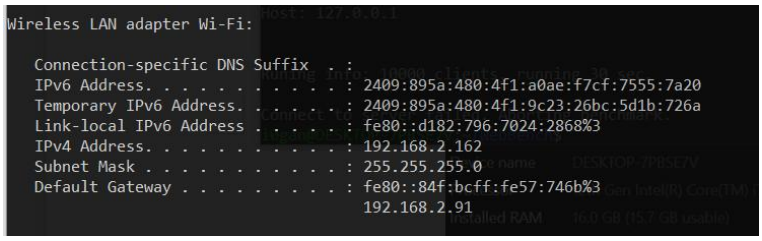

图 8.2 网卡信息

未运行测试程序与被测试程序时系统资源占用情况，如下图 8.3。

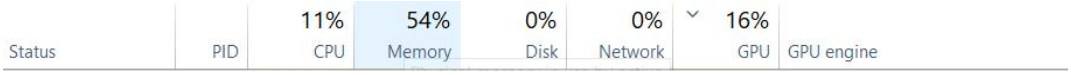

图 8.3 系统资源信息

8.2 登录节点压力测试

登录节点采用的通信协议是 HTTP 协议，那么可以通过 webbench 来模拟客户端，发起请求，测试 URL 为 http://192.168.2.162:10088/cdn

模拟两个客户端并发请求测试。

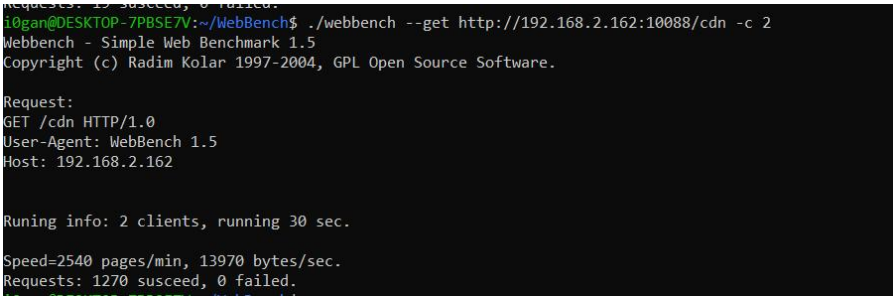

图 8.4 登录节点 2 个客户端压力测试

模拟 100 个客户端并发请求测试。

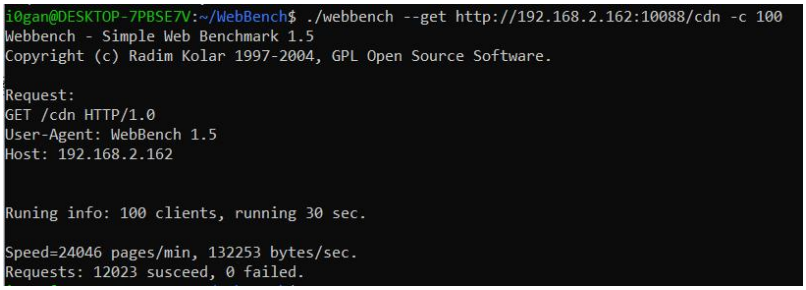

图 8.5 登录节点 100 个客户端压力测试

模拟 1000 个客户端并发请求测试，

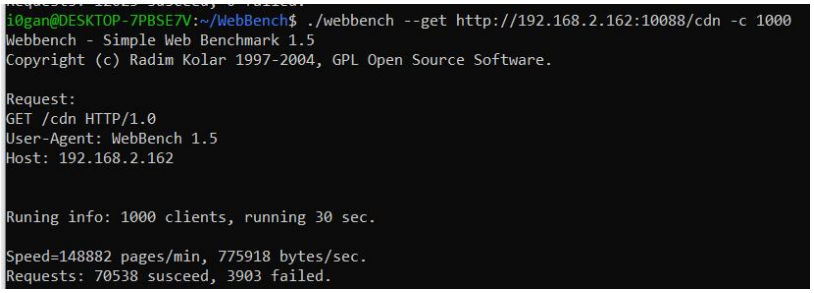

图 8.6 登录节点 1000 个客户端压力测试

模拟 5000 个客户端并发请求测试，

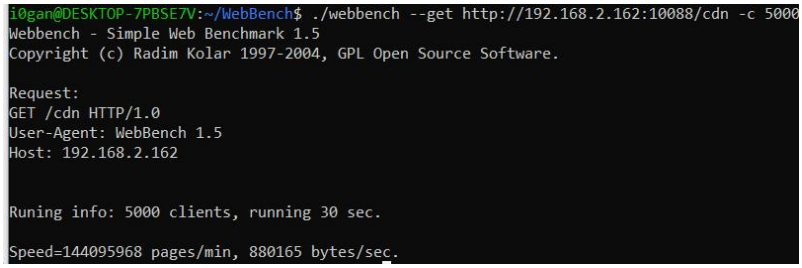

图 8.7 登录节点 5000 个客户端压力测试

在模拟 5000 个客户端进行测试的时候，cpu、内存、gpu 有一点的升高，但都不是很明显。

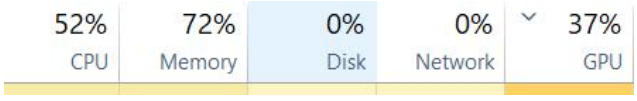

图 8.8 登录节点 5000 个客户端压力测试时资源占用情况

在模拟 1000 个客户端以上的时候，开始出现部分请求丢失的情况，这是由于 HTTP 1.0 模拟的，采用的是短连接方式，每一次请求存在 socket 的连接与断开，这个过程十分消耗资源。通过模拟 1000 个客户端请求的时候，平均每秒能够处理 2351 次请求，这个处理速率稍许比较慢，服务器在处理请求时，CPU 没有并充分发挥出来，后续将会对 HTTP 模块进行优化。

8.3 代理节点压力测试

由于交互协议是自定义协议，需要自己去实现测试工具，测试工具一个进程与服务器只保持一个连接，需要通过多进程方式来模拟客户端，在测试的机器上可能对 CPU 还有内存消耗比较大。多进程启动脚本如下图 8.9 所示。

```
cd bin

for /l %%i in (1,1,100) do (
echo start tester_%%i
start /b .\tester type=tester test=proxy hide=true id=100%%i
)
```

图 8.9 多进程启动脚本

由测试程序会在每一次 update 的时候尽快的发一次包，在测试程序上，检测速率的与响应时间的代码如图 8.10，在被测试的代理节点上，检测代码如图 8.11。测试之前的资源占用情况如图 8.12 所示。

```
INT64 now_time = SquickGetTimeMSEx();
static INT64 last_ack_time = 0;
static int last_index = 0;

if (now_time - last_ack_time > 10000000) {

    if(!is_hide_)
        std::cout << "Test:" << "\n  req_times: " << (ack.index() - last_index)  << " times/10 second \n  last_req_ack_time: " <<
    last_index = ack.index();
    last_ack_time = now_time;

}
```

图 8.10 测试工具上的检测代码

```cpp
void LogicModule::OnReqTestProxy(const socket_t sock, const int msg_id, const char* msg, const uint32_t len) {
    INT64 now_time = SquickGetTimeMSEx();
    static INT64 last_time = 0;
    static INT64 request_time = 0;
    static INT64 last_request_times = 0;
    request_time++;


    if (now_time - last_time > 1000000) {
        rpc::Test req;
        Guid guid;
        INetModule::ReceivePB(msg_id, string(msg, len), req, guid);
        std::cout << "Proxy Test:\n" << "handle quests: " << request_time - last_request_times << " times/second \n req network time: "
        last_time = now_time;
        last_request_times = request_time;
    }

    m_net_->SendMsgWithOutHead(rpc::TestRPC::ACK_TEST_PROXY, string(msg, len), sock);
}
```

图 8.11 代理节点上的请求处理监测代码

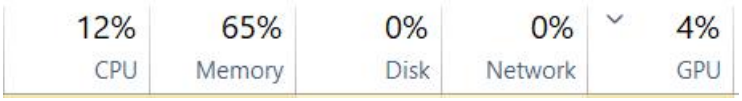

图 8.12

开启 1 个测试程序进行测试

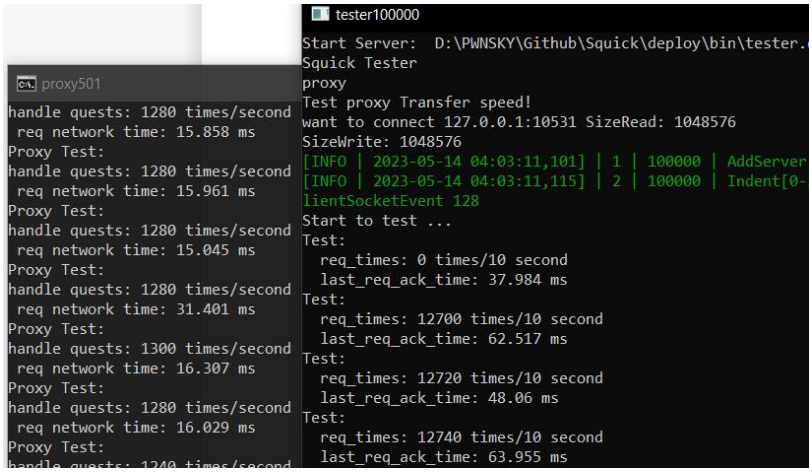

图 8.13 代理节点 1 个测试客户端压力测试

开启 100 个测试程序进行测试

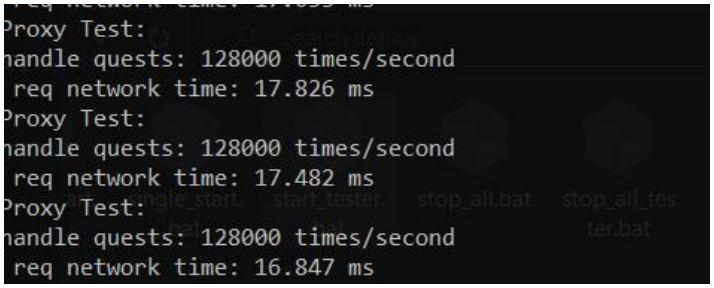

图 8.14 代理节点 100 个测试客户端压力测试

图 8.15 代理节点 100 个测试客户端压力测试资源占用情况

开启 200 个测试程序进行测试

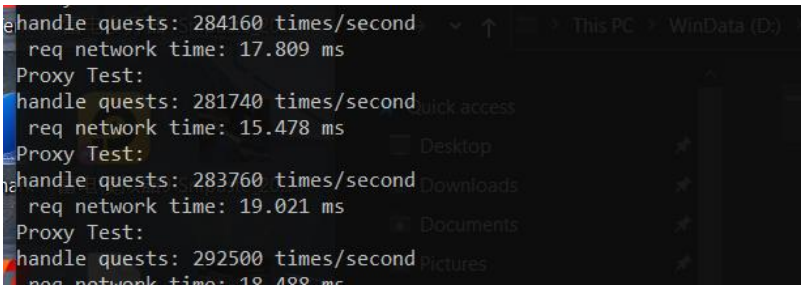

图 8.16 代理节点 200 个测试客户端压力测试

图 8.17 代理节点 200 个测试客户端压力测试资源占用情况

开启 400 个测试程序进行测试

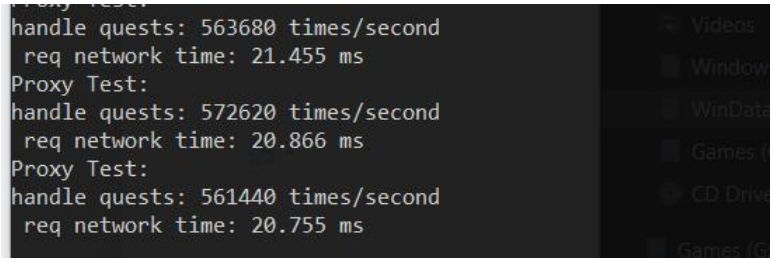

图 8.18 代理节点 400 个测试客户端压力测试

图 8.19 代理节点 400 个测试客户端压力测试资源占用情况

开启 600 个测试程序进行测试

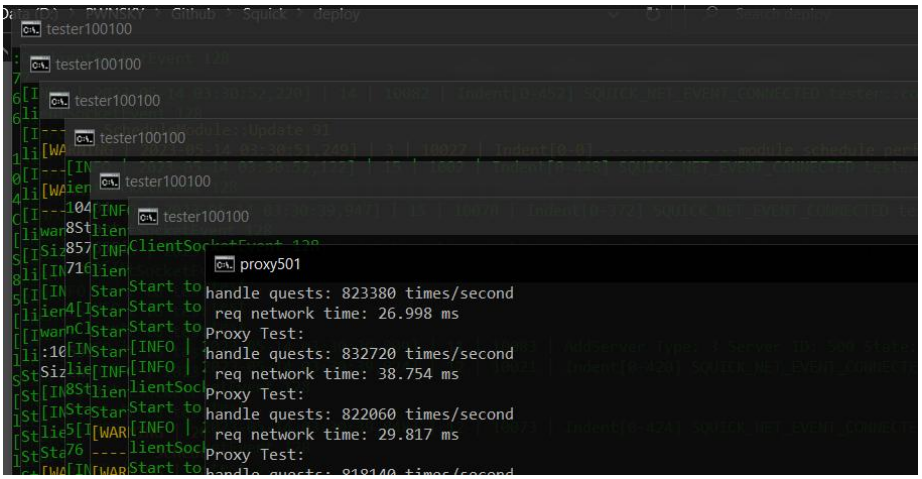

图 8.20 代理节点 600 个测试客户端压力测试

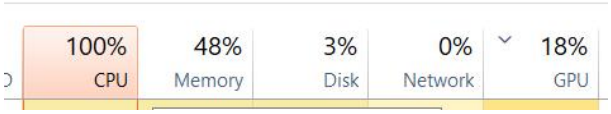

图 8.21 代理节点 600 个测试客户端压力测试资源占用情况

开启 1 个客户端测试程序进行测试的时候，平均每秒要处理 1280 个请求，当开启 100 个测试程序测试的时候，每秒平均处理 128000 个请求，处理请求速率成线性增长，在开启 600 个客户端模拟请求的时候，代理节点平均每秒钟差不多可以处理 82w 次请求，平均每秒能处理每一个测试客户端 1366 请求，不降反增。这个处理速度也是十分惊人的，如果去除测试客户端程序的影响，测试结果可能会更高。在开启 600 个客户端进行模拟的时候，CPU 基本已经跑满，如果再多开的话，会对服务程序造成处理速率方面的降低。

9 商业运用

该框架在商业项目上有多个项目都在使用，且部分公司为了想使用该框架进行引入到他们自己的项目，也多次催促项目文档和客户端 SDK 例子的完善，有着较强的商业发展潜力。本框架目前主要运用在两款商业游戏之中，一款是 VR 丧尸卡牌下棋游戏，另一款是多人在线射击游戏。

9.1 VR 丧尸卡牌下棋游戏

VR 丧尸卡牌下棋游戏是，多人实时在线共同下棋的 VR 游戏，支持创建房间进行对战，还支持语音聊天，在游戏开局后，玩家可以互相看见对方，以面具+双手的形式进行呈现，面具和手的动作取决于现实玩家的动作。每一个人第一轮会抽取一张卡牌，然后摇色子，摇到的什么号，就将棋子放置在对应号数的房间里，直到将手上的 4 个棋子全部放置。从第二轮开始，会有丧尸来攻击，玩家需要借助卡牌的效果、棋子的属性、以及玩家投票来决定棋子怎么走，该牺牲哪一个棋子。考验的是玩家对棋子属性了解、卡牌的技能如何使用、关键时期怎么逃脱等等，趣味性强。在该游戏中，游戏的服务端框架采用的是本次课题研究的框架，在业务开发上，玩法代码采用 C++实现的，并没有采取用 Lua 的原因是，C++执行效率高，相比之下这款游戏实时性强，采用 C++是优先的选择。下图 9.1 是游戏的大厅界面和游玩界面的截图。

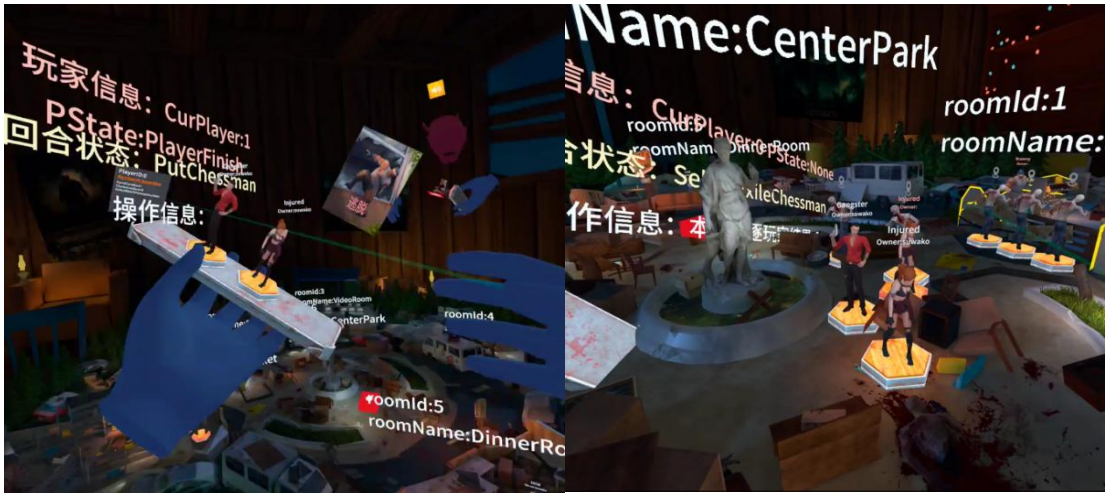

图 9.1 VR 丧尸卡牌下棋游戏截图

9.2 多人在线射击游戏

多人在线射击游戏，是支持移动端、PC 端、VR 平台（Pico、Quest2）的跨平台游戏，采用的是 UE4 来开发的，拥有 PVP 和 PVE 还有单人模式，目前采用的是创建房间组队方式来进行开局，在 PVP 模式中，玩家分为红队和蓝队，每个队伍击杀对方的敌人数高，哪一只队伍就胜利。PVE 模式主要是合作一起去打丧尸闯关，单人模式是去做击杀或打丧尸闯关任务。该游戏服务端的 gameplay 部分采用的是 UE4 的专用服务器，该专用服务器由本框架进行管理，只有在游戏开局的时候，专用服务器才派上用场去计算当前对局中所有的计算，包括人物移动、射击、敌人 AI、丧尸 AI、命中判定、动作同步这些。其他非游戏玩法部分是由本框架来进行处理的。下图 9.2 是游戏的大厅界面和游玩界面的截图。

图 9.2 多人在线射击游戏截图

10 工作总结与展望

10.1 总结

在过去的一段时间里，我致力于设计和实现一个元宇宙分布式服务端框架，采用了分布式架构，使用了微服务的概念，并结合了现有的分布式计算和通信技术实现出该框架，并开源，已在多个商业项目中得已运用。以下是我在这个项目中的工作总结。

在框架设计中，首先进行了框架的整体设计, 分析了元宇宙的需求和特点，并基于这些需求和特点定义了框架的功能和组件，将框架拆分为三个层，内核层、节点层和业务层。

在内核层设计中，参考了 Bigworld 与 UE4 从中引入了一些基本的设计概念，Module、Plugin、Application、Property、Record、Object、GUID、Event。

在内核层中可以将提供的功能分成四大类，服务管理器、动态数据容器、基础的插件、工具。服务管理器是对服务进行管理的程序，它里面包含了很多服务，每一个服务都可以独立运行，服务中又包含了插件管理器和模块管理器。动态数据容器主要是包含了两个 Property 和 Record。基础插件包含有日志插件、配置插件、内核插件还有 Lua 插件。在内核层中主要是基本功能的提供，如网络、日志、插件与模块的管理器、配置、数据交换格式、事件系统以及大量的工具脚本等等。

在节点层设计中，将节点可以称之为一个独立的服务，它允许遵循它所提供的协议进行连接，若授权之后，可以与它进行正常交互。节点类型（角色）有很多种，主要还是总体的游戏服务功能进行划分的，如 master、login、world、lobby、micro、game、gameplay、gameplay_manager、proxy、db_proxy 等等。部分节点可以支持横向拓展，支持负载均衡的运算与任务分配。每一个节点都会提供分布式支持、负载均衡、节点通信等基本功能的支持。

在业务层设计中，业务层是业务上的逻辑开发，如 gameplay，背包系统，大厅活动逻辑等等，这些逻辑，需要在业务层进行实现，业务层强调的是根据业务需求实现的逻辑，不需要深入了解或修改内核层节点层上的逻辑即可完成的需求，一般这种业务需求都具有普遍性，都是可以在建设好基础框架之上，业务需求都能基于该框架较容易的实现出来。每个层级从里到外依次包含，内层功能不依赖于外层，外层向内层调用时，采用面向接口编程，内层会以接口方式提供给外层。

在性能和可扩展性上，注重了性能和可扩展性的考虑, 采用了一些优化技术和策略，以提高框架的性能，并设计了一套可扩展的架构，以支持元宇宙程序的快速增长和扩展。

在压力测试上，开启 600 个客户端进行模拟测试，代理节点平均每秒钟差不多可以处理 82w 次请求，平均每秒能处理每一个测试客户端 1366 请求。这处理速率是非常惊人的，CPU 基本已经跑满，如果将客户端测试程序放在其他机器进去运行，想必测试结果会更好。

在商业项目中，框架在商业项目上有多个项目都在使用，且部分公司为了想使用该框架进行引入到他们自己的项目，也多次催促项目文档和客户端 SDK 例子的完善，有着较强的商业发展潜力。

在实现框架的过程中，遇到了一些挑战和困难，但通过不断的努力和尝试，成功地克服了这些问题，并完成了一个功能完备且可靠的分布式服务端框架。

通过这个项目，深入了解了服务端相关的概念和技术，并获得了设计和实现分布式服务端框架的经验。相信这个框架将为元宇宙的开发和应用提供有力的支持，并为用户提供。

10.2 展望

完善该框架使其更健壮，该框架虽然得到部分商业项目的验证，但想要达到一个高可用、高性能的大型商业项目的需求，还有比较长的路要走，比如模拟 UE4 或 Unity3d 的物理效果，大世界地图碰撞检测，gameplay 节点 KCP 协议的支持等等，相信在之后我会将以上提到的内容给逐渐弥补上。

强化安全和隐私保护，安全和隐私保护成为重要的关注点，目前我的安全性比较低，并未采用严格的权限管理机制去管理每一个分布式节点，只是在 proxy 节点做了一层校验，节点之间通信也需要不断加强安全性，提供更加可靠的身份验证和访问控制机制，保护用户的个人信息和敏感数据。

能够运用到更多的商业项目中，如果一款项目不能运用到产生价值的项目中，那么该项目的研究就毫无意义，希望，随着对该框架的不断完善，让越来越多的公司注意到该框架的优点，能够将其框架运用到他们的项目之中，这是一个框架发展内驱源源不断的动力。

参考文献

[1] Wikipedia.RiotSydney[EB/OL].US.Wikipedia.[2023,04,14].https://en.wikipedia.org/wiki/Riot_Sydney.

[2] egametang.ET[EB/OL].US:Github,[2023,04,20].https://github.com/egametang/ET.

[3] ketoo.NoahGameFrame/ET[EB/OL].US:Github,[2021,05,20].https://github.com/ketoo/NoahGameFrame.

[4] Unity.Multiplayer and Networking[EB/OL].US:Unity,[2021.3.04].https://docs.unity3d.com/Manual/UNet.html.

[5] UnrealEngine.Networking and Multiplayer[EB/OL].US:UnrealEngine,[2023].https://docs.unrealengine.com/5.2/en-US/networking-and-multiplayer-in-unreal-engine/.

[6] NetEase.Pomelo-a fast, scalable game server framework for nodejs[EB/OL].HangZhou: NetEase,[2018.5].https://github.com/NetEase/pomelo.

[7] Tencent Cloud. Game Server Elastic-scaling[M].ShengZhen Tencent,2019.

[8] Jani Ahde. Real-time Unity Multiplayer Server Implementation[D].South-Eastern Finland, South-Eastern Finland University of Applied Sciences, XAMK, Kouvola Campus, 2017.

[9] Huy Pham. A JavaScript Finite-state Game Engine for Multiplayer Turn-based Games[D]. Finland,Vaasa University of Applied Sciences VAMK, 2022.

[10] Wanyun Xie. A Game-theoretical Framework for Byzantine-Robust Federated Learning[D]. Stockholm, Sweden, KTH Royal Institute of Technology, 2022.

[11] Luis Omar Alpala Darío J, Quiroga-Parra ,Juan Carlos Torres and Diego H. Peluffo-Ordóñez. Smart Factory Using Virtual Reality and Online Multi-User: Towards a Metaverse for Experimental Frameworks[R]. Morocco, Mohammed VI Polytechnic University, 2022.

[12] Patric Kabus, Alejandro P. Buchmann and Databases and Distributed Systems Group. A FRAMEWORK FOR NETWORK-AGNOSTIC MULTIPLAYER GAMES[R]. Darmstadt, Germany, Technical University of Darmstadt2007.

[13] Open Mobile Alliance Ltd. Game Services Architecture[R]. Worldwide, Open Mobile Alliance Ltd,2006.

[14] 聂凡杰.基于 Reactor 模式的高性能端框架技术的研究与实例分析[C]浙江理工大学,2020.

[15] 章俊.TCP 技术在内容分发网络上的应用研究[D].清华大学,2016.

[16] 杨明极，王鹤，赵加凤.基于 CPU 和内存利用率的负载均衡算法的研究[J].科技通报，2016,32(04): 160-164.

[17] 罗剑锋. Boost 程序库完全开发指南[M].北京:电子工业出版社,2017:491--530.

[18] 唐富强,于鸿洋，张萍. Linux 下通用线程池的改进与实现[J].计算机工程与应用,2012,48(28):77-83.

致 谢

这篇论文顺利的完成，首先我要得感谢导师彭承宗教授，从选题的确定到论文的安排都给了我非常多的建议和指导。我也非常感谢王娟、段光明、吴春旺、杨帆、张仕斌、万武南老师们的教导和支持，在我学习和成长过程中起着至关重要的作用，他们的知识、经验和指导对我的学术和职业发展中起到了积极的影响。王娟、张仕斌老师也曾告诉我该如何做人做事、让我受益匪浅，衷心的感谢他们。感谢老师们对我四年来的培养，他们的激励和鼓励也使我更有动力去追求自己的目标，他们专业知识和指导帮助我更好地理解和应用相关领域的知识，使我能够取得学习上的进步和成就。我对老师它们表示衷心感谢，他们的支持对我来说无疑是非常宝贵的。

如今，我的论文已顺利完成，我的本科生涯也接近尾声，在这期间，我有幸处在一个十分融洽的道格安全研究实验室中，无论是研究上的困难和生活上的不便，实验室都给予了我很大的帮助。十分感谢陆辉学长带我加入实验室，让我从小白逐渐成长起来，也很感谢实验室里的每一位伙伴，让我在技术成长给予了很大的帮助。

希望老师们身体健康，万事如意，也祝母校和道格实验室发展越来越好。

作者简介：道格安全研究实验室成员、2021 四川省大学生信息安全技术大赛第一名、鸣潮游戏项目-安全开发工程师。

姓　　名：徐绿国　　性别：　男

出生年月：2002 年　6 月　　民族：　苗族

E-mail: 418894113@qq.com

声 明

本论文的工作是2022年11月至2023年6月在成都信息工程大学网络空间安全学院完成的。文中除了特别加以标注地方外，不包含他人已经发表或撰写过的研究成果，也不包含为获得成都信息工程大学或其他教学机构的学位或证书而使用过的材料。

关于学位论文使用权和研究成果知识产权的说明：

本人完全了解成都信息工程大学有关保管使用学位论文的规定，其中包括：

（1）学校有权保管并向有关部门递交学位论文的原件与复印件。

（2）学校可以采用影印、缩印或其他复制方式保存学位论文。

（3）学校可以学术交流为目的复制、赠送和交换学位论文。

（4）学校可允许学位论文被查阅或借阅。

（5）学校可以公布学位论文的全部或部分内容（保密学位论文在解密后遵守此规定）。

除非另有科研合同和其他法律文书的制约，本论文的科研成果属于成都信息工程大学。

特此声明！

作者签名：

2023年6月7日