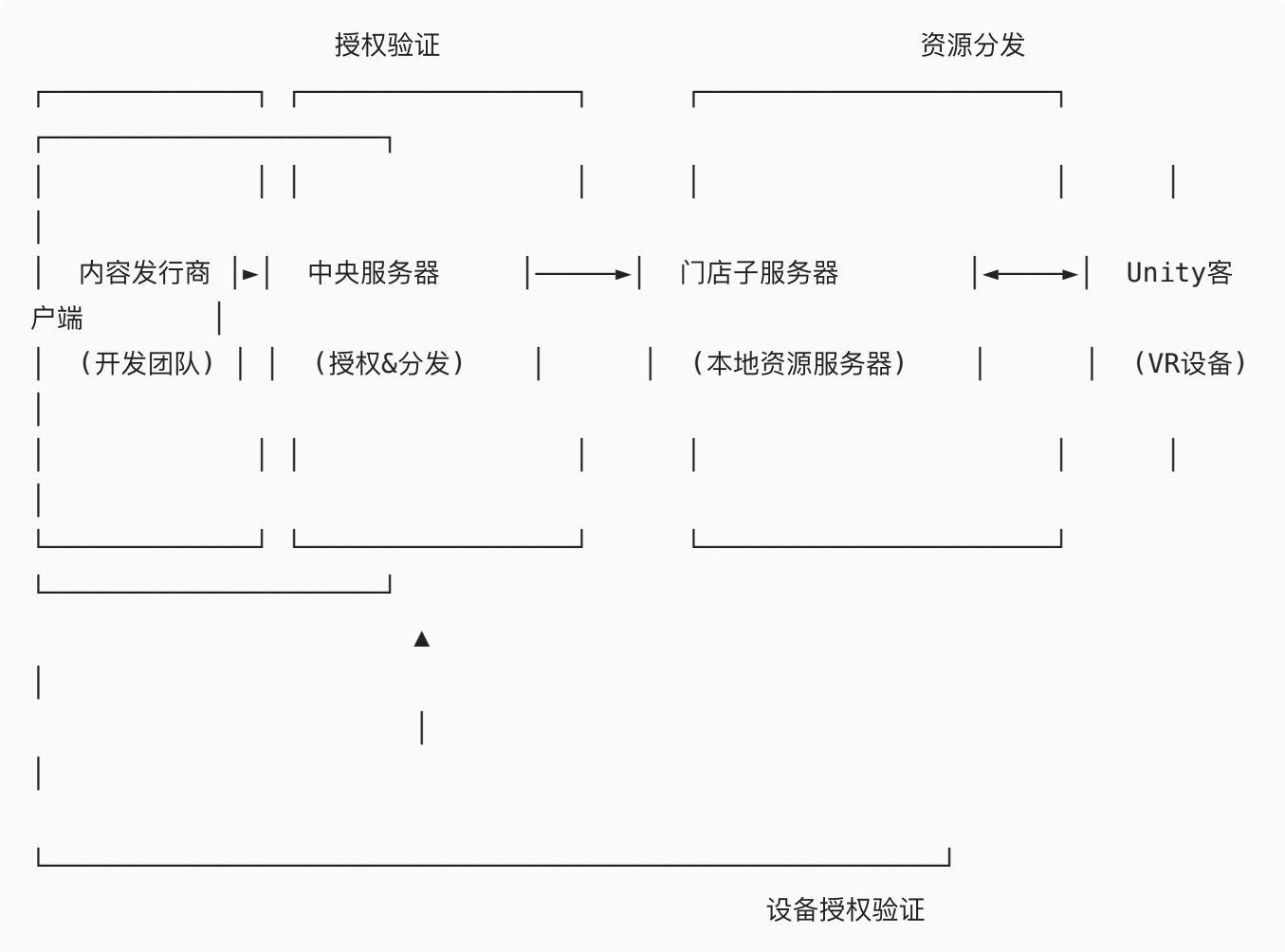
浸趣玩后端资源服务对接说明文档

1. 概述

本文档描述浸趣玩沉浸式剧场项目的Unity客户端与后端资源服务器的对接方案，重点说明内容发行、门店分发及客户端资源加载流程。Unity客户端仅与门店子服务器通信获取资源，而授权验证则通过中央服务器完成。

2. 系统架构（修订）

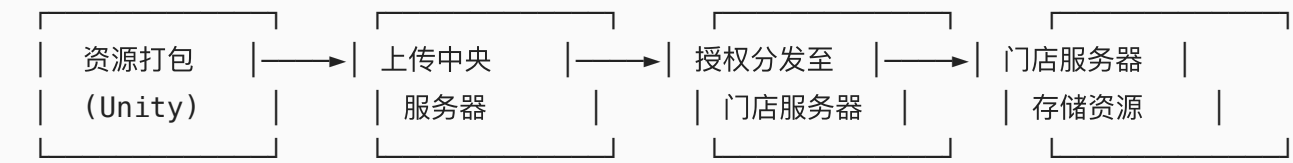


2.1 组件说明（修订）

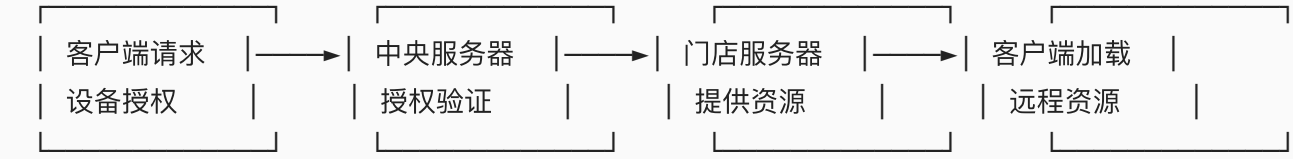
- **内容发行商**：开发团队，负责内容创作、打包和上传资源
- **中央服务器**：负责内容管理、授权验证和向门店分发资源，不直接向客户端提供资源下载
- **门店子服务器**：存储该门店获得授权的资源，为本门店VR设备提供资源下载服务
- **Unity客户端**：VR设备上运行的应用，从门店子服务器获取资源，向中央服务器验证授权

3. 资源流通路径

3.1 资源发布流程



3.2 资源获取流程



4. 资源打包策略

4.1 Addressable资源分组

基于StoryDefinition类，将资源按以下结构组织和打包：

```
远程资源分组结构:
├── Stories
│   ├── [StoryId1] (如GeminiScroll)
│   │   ├── story_definition.json  (故事定义文件)
│   │   ├── thumbnail.bundle       (故事缩略图资源包)
│   │   ├── shared_assets.bundle   (故事共享资源包)
│   │   │
│   │   ├── [SceneId1].bundle      (场景主包，如Lvl_TimeSpaceRiver)
│   │   ├── [SceneId1]_env.bundle  (场景环境包)
│   │   ├── [SceneId1]_char.bundle (场景角色包)
│   │   ├── [SceneId1]_audio.bundle(场景音频包)
│   │   │
│   │   └── [其他场景相关包]
│   │
│   └── [StoryId2] (类似结构)
│
└── Catalog
```

```
    ├── catalog_[timestamp].json   (资源目录文件)
    └── catalog_[timestamp].hash   (资源目录哈希值)
```

5. 服务器API规范（修订）

5.1 中央服务器API

中央服务器主要负责授权和资源分发管理：

5.1.1 设备授权验证API

```
POST /api/auth/device
Content-Type: application/json

Body:
{
  "deviceId": "VR_DEVICE_SN_12345",
  "storeId": "store_001",
  "timestamp": "2025-04-01T10:15:30Z",
  "signature": "hash_signature" // 安全验证
}
```

响应格式：

```
{
  "code": 200,
  "data": {
    "isAuthorized": true,
    "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
    "expiresIn": 86400,
    "authorizedStories": ["GeminiScroll", "MoonStation"],
    "serverConfig": {
      "resourceServerUrl": "http://192.168.1.100:8080",
      "tokenRefreshInterval": 43200
    }
  }
}
```

5.1.2 资源上传API（内容发行商使用）

```
POST /api/admin/upload
Content-Type: multipart/form-data

Form参数:
- file: 资源文件
- storyId: 故事ID
- resourceType: 资源类型
- version: 版本号
- hash: 文件哈希值
```

5.1.3 资源分发API（系统内部使用）

```
POST /api/admin/distribute
Content-Type: application/json

Body:
{
  "storyId": "GeminiScroll",
  "version": "1.0.0",
  "storeIds": ["store_001", "store_002"],  // 目标门店
  "resources": ["main", "shared", "scenes"] // 分发的资源类型
}
```

5.2 门店子服务器API

门店子服务器负责为本门店VR设备提供资源：

5.2.1 获取可用故事列表

```
GET /api/local/stories/list
Authorization: Bearer {accessToken} // 中央服务器颁发的访问令牌
```

响应格式：

```
{
  "code": 200,
  "data": {
    "stories": [
      {
        "storyId": "GeminiScroll",
```

```
        "displayName": "清明上河珏",
        "version": "1.0.0",
        "thumbnailUrl": "/assets/GeminiScroll/thumbnail.jpg",
        "description": "姑苏繁华:清明上河珏",
        "size": 1024000000,
        "scenes": 6,
        "lastUpdated": "2025-03-15T08:00:00Z"
      },
      // 其他授权故事...
    ]
  }
}
```

5.2.2 获取故事详情

```
GET /api/local/stories/{storyId}
Authorization: Bearer {accessToken}
```

响应格式：

```
{
  "code": 200,
  "data": {
    "storyDefinition": {
      // 完整的StoryDefinition对象JSON
    },
    "resourceInfo": {
      "baseUrl": "/assets/GeminiScroll/",
      "catalogUrl": "/catalogs/catalog_20250315.json",
      "size": {
        "total": 1024000000,
        "scenes": {
          "Lvl_TimeSpaceRiver": 150000000,
          // 其他场景大小...
        }
      }
    }
  }
}
```

5.2.3 资源下载端点

```
GET /assets/{storyId}/{bundleName}
Authorization: Bearer {accessToken}
```

6. 门店资源分发与管理

6.1 门店资源同步机制

中央服务器将定期或按需向门店子服务器推送授权资源：

1. **定期同步**：每天凌晨自动同步
2. **按需同步**：新资源发布或门店授权变更时触发
3. **增量更新**：仅同步变更的资源包，节省带宽

6.2 门店资源存储策略

1. **分级存储**：
   - 高频访问内容存储在SSD
   - 低频访问内容存储在HDD
   - 自动根据访问频率迁移
2. **内容缓存管理**：
   - 设置最大缓存容量
   - LRU(最近最少使用)策略淘汰不常用内容
   - 保留基础内容永不淘汰

6.3 门店服务器状态监控

门店子服务器应定期向中央服务器报告状态：

```
{
  "storeId": "store_001",
  "timestamp": "2025-04-01T12:00:00Z",
  "status": "online",
  "storageUsage": {
    "total": 500000000000,
    "available": 350000000000,
    "used": 150000000000
  },
  "resourceVersions": {
    "GeminiScroll": "1.0.0",
    "MoonStation": "1.2.1"
  },
  "activeDevices": 8,
```

```
  "networkStatus": {
    "uplink": 50000000,  // bps
    "downlink": 100000000  // bps
  }
}
```

7. Unity客户端实现

7.1 资源加载流程（修订）

Unity客户端在加载资源时需遵循以下流程：

```
// 初始化资源系统
async Task InitializeAddressables()
{
    // 1. 获取设备授权（与中央服务器通信）
    var authResult = await CentralAuthService.AuthorizeDevice(deviceId,
storeId);
    if (!authResult.isAuthorized) {
        // 处理授权失败
        return;
    }

    // 2. 使用返回的门店服务器URL
    string storeServerUrl = authResult.serverConfig.resourceServerUrl;

    // 3. 设置Addressables远程加载路径
    Addressables.InitializeAsync().WaitForCompletion();
    Addressables.InternalIdTransformFunc = (id) => {
        // 转换资源ID为门店服务器URL
        return id.Replace("{RemoteLoadPath}", storeServerUrl);
    };

    // 4. 更新资源目录
    var catalogUpdateHandle = Addressables.CheckForCatalogUpdates();
    await catalogUpdateHandle.Task;

    List<string> catalogs = catalogUpdateHandle.Result;
    if (catalogs != null && catalogs.Count > 0)
    {
        var updateHandle = Addressables.UpdateCatalogs(catalogs);
        await updateHandle.Task;
    }

    // 5. 预加载公共资源
```

```
    await Addressables.DownloadDependenciesAsync("BuiltIn", false).Task;
}
```

7.2 故事资源管理

```
// 故事管理类
public class StoryManager
{
    // 获取可用故事列表
    public async Task<List<StoryInfo>> GetAvailableStories()
    {
        // 从门店服务器获取故事列表
        HttpClient client = new HttpClient();
        client.DefaultRequestHeaders.Authorization =
            new AuthenticationHeaderValue("Bearer",
AuthService.CurrentToken);

        var response = await client.GetAsync($"
{ResourceConfig.StoreServerUrl}/api/local/stories/list");
        var content = await response.Content.ReadAsStringAsync();

        return JsonUtility.FromJson<StoriesResponse>(content).data.stories;
    }

    // 加载故事资源
    public async Task LoadStory(string storyId)
    {
        // 1. 获取故事详情
        var storyDetails = await GetStoryDetails(storyId);

        // 2. 加载故事定义文件
        var loadDefinitionHandle = Addressables.LoadAssetAsync<TextAsset>($"
{storyId}/story_definition.json");
        await loadDefinitionHandle.Task;
        string definitionJson = loadDefinitionHandle.Result.text;

        StoryDefinition storyDef = new StoryDefinition();
        storyDef.FromJson(definitionJson);

        // 3. 预加载共享资源
        await Addressables.DownloadDependenciesAsync($"
{storyId}/shared_assets").Task;

        // 4. 加载初始场景
        var initialScene = storyDef.scenes.Find(s => s.isInitialScene);
```

```
        if (initialScene != null)
        {
            await LoadScene(storyId, initialScene.sceneId);
        }
    }

    // 加载场景资源
    public async Task LoadScene(string storyId, string sceneId)
    {
        // 加载场景主包
        await Addressables.DownloadDependenciesAsync($"
{storyId}/{sceneId}").Task;

        // 按优先级加载场景分包
        // 注：实际实现应根据ScenePackageDefinition的priority和loadAtStart属性进
行筛选
        await Addressables.DownloadDependenciesAsync($"
{storyId}/{sceneId}_env").Task;
        await Addressables.DownloadDependenciesAsync($"
{storyId}/{sceneId}_char").Task;
        await Addressables.DownloadDependenciesAsync($"
{storyId}/{sceneId}_audio").Task;
    }
}
```

7.3 授权管理实现

```
// 设备授权服务
public static class AuthService
{
    private static string _currentToken;
    private static DateTime _tokenExpiration;
    private static string _deviceId;
    private static string _storeId;

    public static string CurrentToken => _currentToken;

    // 初始化授权服务
    public static void Initialize(string deviceId, string storeId)
    {
        _deviceId = deviceId;
        _storeId = storeId;
    }

    // 向中央服务器验证设备授权
```

```
    public static async Task<AuthResult> AuthorizeDevice()
    {
        // 检查是否需要刷新令牌
        if (!string.IsNullOrEmpty(_currentToken) && DateTime.Now <
_tokenExpiration)
        {
            return new AuthResult {
                isAuthorized = true,
                accessToken = _currentToken
            };
        }

        HttpClient client = new HttpClient();

        // 构建授权请求
        var authRequest = new AuthRequest
        {
            deviceId = _deviceId,
            storeId = _storeId,
            timestamp = DateTime.UtcNow.ToString("o"),
            signature = GenerateSignature(_deviceId, _storeId)
        };

        var jsonContent = JsonUtility.ToJson(authRequest);
        var content = new StringContent(jsonContent, Encoding.UTF8,
"application/json");

        // 发送授权请求到中央服务器
        var response = await client.PostAsync(
            $"{ResourceConfig.CentralServerUrl}/api/auth/device", content);

        if (response.IsSuccessStatusCode)
        {
            var responseJson = await response.Content.ReadAsStringAsync();
            var authResponse = JsonUtility.FromJson<AuthResponse>
(responseJson);

            if (authResponse.data.isAuthorized)
            {
                _currentToken = authResponse.data.accessToken;
                _tokenExpiration =
DateTime.Now.AddSeconds(authResponse.data.expiresIn);

                // 更新资源服务器配置
                ResourceConfig.StoreServerUrl =
authResponse.data.serverConfig.resourceServerUrl;
```

```
                return new AuthResult {
                    isAuthorized = true,
                    accessToken = _currentToken,
                    serverConfig = authResponse.data.serverConfig
                };
            }
        }

        return new AuthResult { isAuthorized = false };
    }

    // 生成签名
    private static string GenerateSignature(string deviceId, string storeId)
    {
        // 安全签名生成逻辑
        // 可使用HMAC等算法确保请求的合法性
        return "signature_placeholder";
    }
}
```

8. 资源更新与热修复

8.1 资源更新流程

1. 内容发行商上传更新的资源到中央服务器
2. 中央服务器向门店子服务器推送更新
3. 客户端启动或定期检查catalog更新
4. 检测到更新后，下载新版本资源

8.2 增量更新机制

利用Addressable系统的内置增量更新功能：

1. 仅下载hash值变更的bundle
2. 版本之间共享不变的资源
3. 支持资源依赖分析和冗余处理

8.3 紧急修复机制

对于发现严重问题需要紧急修复的情况：

1. 内容发行商上传修复资源
2. 标记为"紧急更新"

3. 中央服务器立即推送到所有门店
4. 客户端接收推送通知强制更新

9. 安全与性能

9.1 安全策略

1. **设备身份验证**：使用设备ID和签名验证设备合法性
2. **令牌认证**：使用JWT令牌控制资源访问权限
3. **资源完整性**：下载后验证资源哈希值
4. **数据传输**：使用HTTPS加密传输

9.2 性能优化

1. **缓存策略**：
   - 客户端缓存已下载资源
   - 门店服务器缓存热门内容
   - 避免重复下载相同资源
2. **并行下载**：
   - 多线程下载不同资源包
   - 按优先级调度下载任务
3. **预加载**：
   - 预测用户行为提前加载资源
   - 空闲时间预热下一场景资源

10. 实施计划

10.1 前期准备

1. 评估每个门店的网络环境和子服务器需求
2. 准备中央服务器基础设施
3. 设计并测试资源分发机制

10.2 开发阶段

1. 开发中央服务器授权和分发系统
2. 开发门店子服务器资源管理功能
3. 实现Unity客户端Addressable配置和资源加载流程

10.3 测试阶段

1. 模拟门店环境测试资源分发
2. 验证授权机制和安全策略
3. 测试大规模资源更新和热修复
4. 性能压测和网络波动测试

10.4 部署阶段

1. 分批部署门店子服务器
2. 门店网络环境配置和优化
3. 初始内容分发
4. 操作人员培训

附录：数据结构和类型定义

A. 授权相关数据结构

```
// 授权请求
public class AuthRequest
{
    public string deviceId;
    public string storeId;
    public string timestamp;
    public string signature;
}

// 授权响应
public class AuthResponse
{
    public int code;
    public AuthData data;

    public class AuthData
    {
        public bool isAuthorized;
        public string accessToken;
        public int expiresIn;
        public List<string> authorizedStories;
        public ServerConfig serverConfig;
    }

    public class ServerConfig
    {
        public string resourceServerUrl;
        public int tokenRefreshInterval;
```

```
    }
}

// 授权结果
public class AuthResult
{
    public bool isAuthorized;
    public string accessToken;
    public AuthResponse.ServerConfig serverConfig;
}
```

B. 资源相关数据结构

```
// 故事列表响应
public class StoriesResponse
{
    public int code;
    public StoriesData data;

    public class StoriesData
    {
        public List<StoryInfo> stories;
    }
}

// 故事基本信息
public class StoryInfo
{
    public string storyId;
    public string displayName;
    public string version;
    public string thumbnailUrl;
    public string description;
    public long size;
    public int scenes;
    public string lastUpdated;
}

// 资源服务器配置
public static class ResourceConfig
{
    public static string CentralServerUrl = "https://central.example.com";
    public static string StoreServerUrl = "http://localhost:8080"; // 默认
值，将被更新
}
```

此对接文档反映了修订后的架构，明确了内容发行商、中央服务器、门店子服务器和Unity客户端之间的交互关系和职责分工。特别强调了Unity客户端仅与门店子服务器通信获取资源，而授权验证则与中央服务器进行，以减轻中央服务器的带宽压力并提高资源访问效率。