游戏服务器伸缩

开发指南

产品文档

版权所有：腾讯云计算（北京）有限责任公司

【版权声明】

©2013-2019 腾讯云版权所有

本文档著作权归腾讯云单独所有，未经腾讯云事先书面许可，任何主体不得以任何形式复制、修改、抄袭、传播全部或部分本文档内容。

【商标声明】

及其它腾讯云服务相关的商标均为腾讯云计算（北京）有限责任公司及其关联公司所有。本文档涉及的第三方主体的商标，依法由权利人所有。

【服务声明】

本文档意在向客户介绍腾讯云全部或部分产品、服务的当时的整体概况，部分产品、服务的内容可能有所调整。您所购买的腾讯云产品、服务的种类、服务标准等应由您与腾讯云之间的商业合同约定，除非双方另有约定，否则，腾讯云对本文档内容不做任何明示或模式的承诺或保证。

文档目录

开发指南
整体流程
服务器集成 gRPC 框架
gRPC C++ 教程
gRPC C# 教程
gRPC Go 教程
gRPC Java 教程
gRPC Lua 教程
gRPC Nodejs 教程
gRPC Unity 教程
获取服务器地址
云 API 调用方式
创建游戏服务器会话
放置游戏服务器会话
本地调试工具
测速工具
游戏进程启动配置

版权所有：腾讯云计算（北京）有限责任公司

开发指南
整体流程

最近更新时间：2020-08-25 09:55:02

整体流程图

Client (PC, mobile device, or game console)
Client service
GSE
Provide the runtime environment
Developer
Provide the game framework and the business logic
4.Access the server through“Ip:port”
3. Request the TencentCloud API and get the server address (IP:port)
CVM instance
Process
Game server session
Process
Game server session
Process
Game server session
1. Integrate the gRPC framework for gRPC bidirectional streaming
Tencent Cloud GSE Scheduling Center
2. Publish the program: C++ or C# engine/custom framework
Tencent Cloud GSE Console

接入步骤

步骤1：服务器集成 gRPC 框架

游戏服务器和 GSE 通过 gRPC 通信，游戏服务器程序集成 gRPC 框架，实现多语言接入，生成游戏服务器可执行文件。各语言服务端接入 GSE 的具体教程请您参考腾讯云 gRPC-C++ 教程、gRPC-C# 教程、gRPC-Go 教程、gRPC-Java 教程、gRPC-Lua 教程、gRPC-Nodejs 教程 文档。其他语言请您参考 gRPC 官方文档。

步骤2：发布程序

1. 上传生成包
   生成包包括游戏服务器可执行文件、依赖包、安装脚本，将其打包成 zip 包后上传。详情请参见 创建生成包。
2. 创建服务器舰队
   将上传的生成包部署在新建的服务器舰队上，并完成进程管理、部署配置、扩缩容配置等。 详情请参见 创建服务器舰队。

步骤3：调用云 API ，获取服务器地址（IP:port）

通过创建游戏服务器会话或放置游戏服务器会话，即可获取服务器地址（IP:port）。

方式一：创建游戏服务器会话

调用云 API：
根据不同的游戏服务器会话支持方式，有不同的客户端云 API 调用流程。

- 当一个游戏服务器会话仅支持一局游戏时 ：
  - 创建游戏服务器会话（CreateGameServerSession）；
  - 加入游戏服务器会话（JoinGameServerSession）。
- 当一个游戏服务器会话支持多局游戏或一个服务（如登录服）时：
  - 查询游戏服务器会话列表（DescribeGameServerSessions）或搜索游戏服务器会话列表（SearchGameServerSessions）；
  - 当已有游戏服务器会话时，则加入游戏服务器会话（JoinGameServerSession）；
  - 当没有游戏服务器会话时，则先创建游戏服务器会话（CreateGameServerSession），再加入游戏服务器会话（JoinGameServerSession）。

调用云 API 的具体操作请参见 创建游戏服务器会话 文档。

方式二：放置游戏服务器会话

调用云 API：

- 开始放置游戏服务器会话（StartGameServerSessionPlacement）；
- 查询游戏服务器会话的放置（DescribeGameServerSessionPlacement）；
- 停止放置游戏服务器会话（StopGameServerSessionPlacement）。

调用云 API 的具体操作请参见 放置游戏服务器会话 文档。

版权所有：腾讯云计算（北京）有限责任公司

步骤4：客户端通过 IP:port访问服务器

步骤3已返回服务端 IP:port，客户端可通过该 IP:port连接至目标服务器。

工作流程图

Developer
Client
Console
Game Server
Tencent Cloud backend
Register the service and publish the code
Register configuration files, models, code packages, and dependencies
Deploy
Initialize
ProcessReady
Access the server
OnHealthCheck
Return whether the request has been successfully sent
CreateGameServerSession
OnStartGameServerSession
Request the server address
ActivateGameServerSession
Assign
JoinGameServerSession
Return the "IP:port"
Call the service based on the address
A player establishes a connection or exits
Access the server
AcceptPlayerSession
RemovePlayerSession
Return
Game server session ends
TerminateGameServerSession
Actively initiate the expansion
Deploy
Expand
Actively initiate the reduction
OnProcessTerminate
ProcessEnding
Automatically/Manually reduce
Destroy

版权所有：腾讯云计算（北京）有限责任公司

服务器集成 gRPC 框架
gRPC C++ 教程

最近更新时间：2021-11-17 18:06:09

安装 gRPC

1. 前提条件--安装 CMake。
   - Linux
     ```
     $ sudo apt install -y cmake
     ```
   - MAC OS
     ```
     $ brew install cmake
     ```
2. 在本地安装 gRPC 和 protocol buffers。

> 说明
> 具体安装流程请参考 安装 CMake，安装 gRPC C++ 的说明，安装 protocol buffers 的说明。

定义服务

gRPC 通过 protocol buffers 实现定义一个服务：一个 RPC 服务通过参数和返回类型来指定可以远程调用的方法。

> 说明
> 我们提供定义服务的 proto 文件，请您直接下载使用，无需自己生成。

生成 gRPC 代码

1. 定义好服务后，通过 protocol buffer 编译器 protoc 生成客户端和服务端的代码（任意 gRPC 支持的语言）。
2. 生成的代码包括客户端的存根和服务端要实现的抽象接口。

版权所有：腾讯云计算（北京）有限责任公司

3. 生成 gRPC 代码步骤：
   在 proto 目录下执行：
   ```
   protoc --cpp_out=. *.proto
   ```
   生成 pb.cc 和 pb.h 文件。

```
protoc --grpc_out=. --plugin=protoc-gen-grpc=`which grpc_cpp_plugin` *.proto
```

生成对应的 gRPC 代码。

将生成的8个文件移到项目合适的位置。

游戏进程集成流程

版权所有：腾讯云计算（北京）有限责任公司

Client
Game Server
gRPC framework
GSE
ProcessReady
OnHealthCheck
CreateGameServerSession
OnStartGameServerSession
ActivateGameServerSession
JoinGameServerSession
Return the "IP:port"or "domain name"
AcceptPlayerSession
RemovePlayerSession
TerminateGameServerSession
OnProcessTerminate
ProcessEnding
Destroy

Game Server 回调接口列表

接口名称	接口功能
OnHealthCheck	健康检查
OnStartGameServerSession	接收游戏服务器会话
OnProcessTerminate	结束游戏进程

Game Server 主调接口列表

接口名称	接口功能
ProcessReady	进程准备就绪
ActivateGameServerSession	激活游戏服务器会话
AcceptPlayerSession	接收玩家会话
RemovePlayerSession	移除玩家会话
DescribePlayerSessions	获取玩家会话列表
UpdatePlayerSessionCreationPolicy	更新玩家会话的创建策略
TerminateGameServerSession	结束游戏服务器会话
ProcessEnding	结束进程
ReportCustomData	上报自定义数据

其他

请求 meta，在游戏进程通过 gRPC 调用 Game Server 主调接口时，需要在 gRPC 请求的 meta 里添加两个字段。

字段	含义	类型
pid	当前游戏进程的 pid	string
requestId	当前请求的 requestId，已使用唯一标记一次请求	string

1. 一般在服务端初始化后，进程检查自身是否可对外提供服务，Game Server 调用 ProcessReady 接口，告知 GSE 进程准备就绪，已准备好托管游戏服务器会话，GSE 接收到后，将服务器实例状态更改为“活跃”。

```
Status GseManager::ProcessReady(std::vector<std::string> &amp;logPath, int clientPort, int grpcPort, GseResponse&amp; reply)
{
ProcessReadyRequest request;
//日志路径
for (auto iter = logPath.begin(); iter != logPath.end(); iter++)
{
request.add_logpathstoupload(*iter);
}

GConsoleLog-&gt;PrintOut(true, "ProcessReady clientPort is %d¥n", clientPort);
GConsoleLog-&gt;PrintOut(true, "ProcessReady grpcPort is %d¥n", grpcPort);
```

```
//设置端口
request.set_clientport(clientPort);
request.set_grpcport(grpcPort);

ClientContext context;
AddMetadata(context);

//准备就绪，可对外提供服务
return stub_-&gt;ProcessReady(&amp;context, request, &amp;reply);
}
```

2. 进程准备就绪后，GSE 调用 OnHealthCheck 接口，对 Game Server 进行健康检查，每1分钟检查1次，连续3次失败就判定该进程不健康，不会分配游戏服务器会话至该进程。

```
Status GameServerGrpcSdkServiceImpl::OnHealthCheck(ServerContext* context, const HealthCheckRequest* request, HealthCheckResponse* reply)
{
reply->set_healthstatus(healthStatus);
return Status::OK;
}
```

3. 因为 Client 调用 CreateGameServerSession 接口创建一个游戏服务器会话，将该游戏服务器会话分配给一个进程，所以触发 GSE 调用该进程的 onStartGameServerSession 接口，并且将 GameServerSession 状态更改为“激活中”。

```
Status GameServerGrpcSdkServiceImpl::OnStartGameServerSession(ServerContext* context, const StartGameServerSessionRequest* request, GseResponse* reply)
{
auto gameServerSession = request->gameserversession();
GGseManager->SetGameServerSession(gameServerSession);
GseResponse processReadyReply;
Status status = GGseManager->ActivateGameServerSession(gameServerSession.gameserversessionid()
, gameServerSession.maxplayers(), processReadyReply);
// 对status和replay来判断，激活是否成功
```

```
return Status::OK;
}
```

4. 当 Game Server 收到 onStartGameServerSession，您自行处理一些逻辑或资源分配，准备就绪后，Game Server 就调用 ActivateGameServerSession 接口,通知 GSE 游戏服务器会话已分配给一个进程，现在已准备好接收玩家请求，将服务器状态更改为“活跃”。

```
Status GseManager::ActivateGameServerSession(std::string gameServerSessionId, int maxPlayers, GseResponse& reply)
{
GConsoleLog->PrintOut(true, "ActivateGameServerSession gameServerSessionId is %s¥n", gameServerSessionId.c_str());
GConsoleLog->PrintOut(true, "ActivateGameServerSession maxPlayers is %d¥n", maxPlayers);
ActivateGameServerSessionRequest request;
request.set_gameserversessionid(gameServerSessionId);
request.set_maxplayers(maxPlayers);

ClientContext context;
AddMetadata(context);

return stub_->ActivateGameServerSession(&context, request, &reply);
}
```

5. 当 Client 调用 JoinGameServerSession 接口玩家加入后，Game Server 调用 AcceptPlayerSession 接口验证玩家合法性，如果连接被接受，则将 PlayerSession 状态设置为“活跃”。如果 Client 调用 JoinGameServerSession 接口在60秒内未收到响应，则将 PlayerSession 状态更改为“超时”，然后重新调用 JoinGameServerSession。

```
Status GseManager::AcceptPlayerSession(std::string playerSessionId, GseResponse& reply)
{
AcceptPlayerSessionRequest request;
request.set_gameserversessionid(gameServerSession.gameserversessionid());
request.set_playersessionid(playerSessionId);
ClientContext context;
AddMetadata(context);
```

```
return stub_->AcceptPlayerSession(&context, request, &reply);
}
```

6. 游戏结束或者玩家退出后，Game Server 调用 RemovePlayerSession 接口移除玩家，将 playersession 状态更改为“已完成”，并预留游戏服务器会话中的玩家位置。

```
Status GseManager::RemovePlayerSession(std::string playerSessionId, GseResponse& reply)
{
GConsoleLog->PrintOut(true, "RemovePlayerSession playerSessionId is %s¥n", playerSessionId.c_str
());
RemovePlayerSessionRequest request;
request.set_gameserversessionid(gameServerSession.gameserversessionid());
request.set_playersessionid(playerSessionId);
ClientContext context;
AddMetadata(context);

return stub_->RemovePlayerSession(&context, request, &reply);
}
```

7. 当一个游戏服务器会话（一组游戏对局或一个服务）结束后，Game Server 调用 TerminateGameServerSession 接口结束 GameServerSession，将 GameServerSession 状态更改为"已终止"。

```
Status GseManager::TerminateGameServerSession(GseResponse& reply)
{
GConsoleLog->PrintOut(true, "start to TerminateGameServerSession¥n");
TerminateGameServerSessionRequest request;
request.set_gameserversessionid(gameServerSession.gameserversessionid());
ClientContext context;
AddMetadata(context);
```

```
return stub_->TerminateGameServerSession(&context, request, &reply);
}
```

8. 当健康检查失败或缩容时，GSE 调用 OnProcessTerminate 接口结束游戏进程，缩容时依据是您在 GSE 控制台配置的 保护策略。

```
Status GameServerGrpcSdkServiceImpl::OnProcessTerminate(ServerContext* context, const ProcessT
erminateRequest* request, GseResponse* reply)
{
auto terminationTime = request->terminationtime();
GGseManager->SetTerminationTime(terminationTime);
//调以下两个接口，会立即结束游戏服务器会话，建议无玩家或无游戏服务器会话后，再调用ProcessEndin
g结束进程
//不调用以下两个接口，根据保护策略调用ProcessEnding结束进程，建议配置时限保护
```

```
//结束游戏服务器会话
GseResponse terminateGameServerSessionReply;
GGseManager->TerminateGameServerSession(terminateGameServerSessionReply);

// 结束进程
GseResponse processEndingReply;
GGseManager->ProcessEnding(processEndingReply);

 return Status::OK;
}
```

9. Game Server 调用 ProcessEnding 接口会立刻结束进程，将服务器进程状态更改为“已终止”，并回收资源。

```
//主动调用：一局游戏对应一个进程，当一局游戏结束后主动调用ProcessEnding接口
//被动调用：当缩容或进程异常健康检查失败时，根据保护策略被动调用ProcessEnding接口，配置完全保护和时限保护策略时需要先判断游戏服务器会话上有无玩家，再被动调用
Status GseManager::ProcessEnding(GseResponse& reply)
{
GConsoleLog->PrintOut(true, "start to ProcessEnding¥n");
ProcessEndingRequest request;
ClientContext context;
AddMetadata(context);

 return stub_->ProcessEnding(&context, request, &reply);
}
```

10. Game Server 调用 DescribePlayerSessions 接口获取游戏服务器会话下的玩家信息（根据业务可选）。

```
Status GseManager::DescribePlayerSessions(std::string gameServerSessionId, std::string playerId, std::string playerSessionId,std::string playerSessionStatusFilter, std::string nextToken, int limit, DescribePlayerSessionsResponse& reply)
{
GConsoleLog->PrintOut(true, "start to DescribePlayerSessions¥n");
DescribePlayerSessionsRequest request;
request.set_gameserversessionid(gameServerSessionId);
```

版权所有：腾讯云计算（北京）有限责任公司

```
request.set_playerid(playerId);
request.set_playersessionid(playerSessionId);
request.set_playersessionstatusfilter(playerSessionStatusFilter);
request.set_nexttoken(nextToken);
request.set_limit(limit);
ClientContext context;
AddMetadata(context);

return stub_->DescribePlayerSessions(&context, request, &reply);
}
```

1. Game Server 调用 UpdatePlayerSessionCreationPolicy 接口更新玩家会话的创建策略，设置是否接受新玩家，即游戏会话里是否允许加入人（根据业务可选）。

```
Status GseManager::UpdatePlayerSessionCreationPolicy(std::string newpolicy, GseResponse& reply
)
{
GConsoleLog->PrintOut(true, "UpdatePlayerSessionCreationPolicy, newpolicy is %s¥n", newpolicy.
c_str());
UpdatePlayerSessionCreationPolicyRequest request;
request.set_gameserversessionid(gameServerSession.gameserversessionid());
request.set_newplayersessioncreationpolicy(newpolicy);
ClientContext context;
AddMetadata(context);
```

```
return stub_->UpdatePlayerSessionCreationPolicy(&context, request, &reply);
}
```

12. Game Server 调用 ReportCustomData 接口告知 GSE 的自定义数据，在查询游戏服务器会话时可以查到（根据业务可选）。

```
Status GseManager::ReportCustomData(int currentCustomCount, int maxCustomCount, GseResponse& repl
y)
{
GConsoleLog->PrintOut(true, "ReportCustomData, currentCustomCount is %d¥n", currentCustomCount);
GConsoleLog->PrintOut(true, "ReportCustomData, maxCustomCount is %d¥n", maxCustomCount);
ReportCustomDataRequest request;
request.set_currentcustomcount(currentCustomCount);
request.set_maxcustomcount(maxCustomCount);

ClientContext context;
```

```
AddMetadata(context);

return stub_->ReportCustomData(&context, request, &reply);
}
```

启动服务端，供 GSE 调用

服务端运行：将 GrpcServer 启动起来。

```
GameServerGrpcSdkServiceImpl::GameServerGrpcSdkServiceImpl() : serverAddress("127.0.0.1:0"), heal
thStatus(true)
{
sem_init(&sem, 0, 0);
}
void GameServerGrpcSdkServiceImpl::StartGrpcServer()
{
ServerBuilder builder;
builder.AddListeningPort(serverAddress, grpc::InsecureServerCredentials(), &grpcPort);
builder.RegisterService(this);
std::unique_ptr<Server> server(builder.BuildAndStart());
sem_post(&sem);
server->Wait();
}
```

客户端连接 GSE 的 gRPC 服务端

连接服务端：创建一个 gRPC 频道，指定我们要连接的主机名和服务器端口，然后用这个频道创建存根实例。

```
void GseManager::InitStub()
{
auto channel = grpc::CreateChannel("127.0.0.1:5758", grpc::InsecureChannelCredentials());
stub_ = GseGrpcSdkService::NewStub(channel);
}
```

C++ DEMO

1. 单击这里，您可下载 C++ DEMO 代码。

版权所有：腾讯云计算（北京）有限责任公司

2. 生成 gRPC 代码。
   C++ DEMO 代码示例里已生成 gRPC 代码，在 cpp-demo/source/grpcsdk 目录下，不需要额外生成。
3. 启动服务端，供 GSE 调用。
   - 服务端实现。
     在 cpp-demo/source/api 目录下的 grpcserver.cpp，实现了服务端的三个接口。
   - 服务端运行。
     在 cpp-demo/source/api 目录下的 grpcserver.cpp，将 GrpcServer 启动起来。
4. 客户端连接 GSE 的 gRPC 服务端。
   - 客户端实现。
     在 cpp-demo/source/gsemanager 目录下的 gsemanager.cpp，实现了客户端的九个接口。
   - 连接服务端。
     创建一个 gRPC 频道，指定我们要连接的主机名和服务器端口，然后用这个频道创建存根实例。
5. 编译运行。
   i. 安装 cmake。
   ii. 安装 gcc，版本要求4.9以上。
   iii. 将代码下载，在 cpp-demo 目录下，执行以下命令：

```
mkdir build
cmake ..
make
```

   会生成对应的 cpp-demo 可执行文件。
   iv. 将 cpp-demo 可执行文件打包为 生成包，启动路径配置 cpp-demo，无启动参数。
   v. 然后 创建服务器舰队，将生成包部署在服务器舰队上，后续可进行 扩缩容 等一系列操作。

gRPC C# 教程

最近更新时间：2021-11-17 18:06:09

安装 gRPC

1. 使用 gRPC C# 时，需要先安装 .Net Core 3.1 SDK。以 CentOS 操作系统为例，版本不得低于 CentOS 7 或 CentOS 8。
   - 添加签名密钥
   ```
   sudo rpm -Uvh https://packages.microsoft.com/config/centos/7/packages-microsoft-prod.rpm
   ```
   - 安装 .NET Core SDK
   ```
   sudo yum install dotnet-sdk-3.1
   ```
2. 除此之外，您还可以以下运行环境 /IDE 中使用 gRPC C#：
   - Windows：.NET Framework 4.5或更高版本，Visual Studio 2013或更高版本，Visual Studio Code。
   - Linux：Mono 4或更高版本，Visual Studio Code。
   - Mac OS X：Mono 4或更高版本，Visual Studio Code，Visual Studio for Mac。

> 说明
> 具体流程请您参考 安装 gRPC C# 操作步骤。

定义服务

gRPC 通过 protocol buffers 实现定义一个服务：一个 RPC 服务通过参数和返回类型来指定可以远程调用的方法。

> 说明：
> 我们提供定义服务的 proto 文件，请您在 proto文件 里下载使用，无需自己生成。

生成 gRPC 代码

1. 定义好服务后，通过 protocol buffer 编译器 protoc 生成客户端和服务端的代码（任意 gRPC 支持的语言）。
2. 生成的代码包括客户端的存根和服务端要实现的抽象接口。
3. 生成 gRPC 代码步骤：
    - 下载代码，在 csharp-demo 目录下，执行如下指令：
      ```
      dotnet run
      ```
      即可自动编译并运行服务。
    - 程序正确编译运行后，会在 csharp-demo/obj/Debug/netcoreapp3.1 文件夹生成项目依赖的库、二进制文件以及 proto 文件编译后的 .cs 文件。
    - proto 文件的引入是在 csharp-demo/csharpdemo.csproj 中：
      ```
      <Protobuf Include="..¥proto¥csharp-demo¥GameServerGrpcSdkService.proto" Link="GameServerGrpcSdkService.proto"/>
      <Protobuf Include="..¥proto¥csharp-demo¥GseGrpcSdkService.proto" Link="GseGrpcSdkService.proto" />
      ```
      项目依赖于 proto/csharp-demo 文件夹中的 GameServerGrpcSdkService.proto 和 GseGrpcSdkService.proto 两个 proto 文件。

游戏进程集成流程

Client
Game Server
gRPC framework
GSE
ProcessReady
OnHealthCheck
CreateGameServerSession
OnStartGameServerSession
ActivateGameServerSession
JoinGameServerSession
Return the "IP:port"or "domain name"
AcceptPlayerSession
RemovePlayerSession
TerminateGameServerSession
OnProcessTerminate
ProcessEnding
Destroy

Game Server 回调接口列表

接口名称	接口功能
OnHealthCheck	健康检查
OnStartGameServerSession	接收游戏服务器会话
OnProcessTerminate	结束游戏进程

Game Server 主调接口列表

接口名称	接口功能
ProcessReady	进程准备就绪
ActivateGameServerSession	激活游戏服务器会话
AcceptPlayerSession	接收玩家会话
RemovePlayerSession	移除玩家会话
DescribePlayerSessions	获取玩家会话列表
UpdatePlayerSessionCreationPolicy	更新玩家会话的创建策略
TerminateGameServerSession	结束游戏服务器会话
ProcessEnding	结束进程
ReportCustomData	上报自定义数据

其他

请求 meta，在游戏进程通过 gRPC 调用 Game Server 主调接口时，需要在 gRPC 请求的 meta 里添加两个字段。

字段	含义	类型
pid	当前游戏进程的 pid	string
requestId	当前请求的 requestId，已使用唯一标记一次请求	string

1. 一般在服务端初始化后，进程检查自身是否可对外提供服务，Game Server 调用 ProcessReady 接口，告知 GSE 进程准备就绪，已准备好托管游戏服务器会话，GSE 接收到后，将服务器实例状态更改为“活跃”。

```
public static GseResponse ProcessReady(string[] logPath, int clientPort, int grpcPort)
{
logger.Println($"Getting process ready, LogPath: {logPath}, ClientPort: {clientPort}, GrpcPort: {grpcPort}");
//设置端口
var req = new ProcessReadyRequest{
ClientPort = clientPort,
GrpcPort = grpcPort,
};
//日志路径
req.LogPathsToUpload.Add(logPath); //repeated类型解析pb后，是只读类型，需要Add加入
//准备就绪，可对外提供服务
```

```
return GrpcClient.GseClient.ProcessReady(req, meta);
}
```

2. 进程准备就绪后，GSE 调用 OnHealthCheck 接口，对 Game Server 进行健康检查，每1分钟检查1次，连续3次失败就判定该进程不健康，不会分配游戏服务器会话至该进程。

```
public override Task<HealthCheckResponse> OnHealthCheck(HealthCheckRequest request, ServerCallContext context)
{
logger.Println($"OnHealthCheck, HealthStatus: {GseManager.HealthStatus}");
return Task.FromResult(new HealthCheckResponse{
HealthStatus = GseManager.HealthStatus
});
}
```

3. 因为 Client 调用 CreateGameServerSession 接口创建一个游戏服务器会话，将该游戏服务器会话分配给一个进程，所以触发 GSE 调用该进程的 onStartGameServerSession 接口，并且将 GameServerSession 状态更改为“激活中”。

```
public override Task<GseResponse> OnStartGameServerSession(StartGameServerSessionRequest request, ServerCallContext context)
{
logger.Println($"OnStartGameServerSession, request: {request}");
GseManager.SetGameServerSession(request.GameServerSession);
var resp = GseManager.ActivateGameServerSession(request.GameServerSession.GameServerSessionId, request.GameServerSession.MaxPlayers);
return Task.FromResult(resp);
}
```

4. 当 Game Server 收到 onStartGameServerSession，您自行处理一些逻辑或资源分配，准备就绪后，Game Server 就调用 ActivateGameServerSession 接口，通知 GSE 游戏服务器会话已分配给一个进程，现在已准备好接收玩家请求，将服务器状态更改为“活跃”。

```
public static GseResponse ActivateGameServerSession(string gameServerSessionId, int maxPlayers)
{
logger.Println($"Activating game server session, GameServerSessionId: {gameServerSessionId}, MaxPlayers: {maxPlayers}");
var req = new ActivateGameServerSessionRequest{
GameServerSessionId = gameServerSessionId,
MaxPlayers = maxPlayers,
};
```

```
return GrpcClient.GseClient.ActivateGameServerSession(req, meta);
}
```

5. 当 Client 调用 JoinGameServerSession 接口玩家加入后，Game Server 调用 AcceptPlayerSession 接口验证玩家合法性，如果连接被接受，则将 PlayerSession 状态设置为“活跃”。如果 Client 调用 JoinGameServerSession 接口在60秒内未收到响应，则将 PlayerSession 状态更改为“超时”，然后重新调用 JoinGameServerSession。

```
public static GseResponse AcceptPlayerSession(string playerSessionId)
{
logger.Println($"Accepting player session, PlayerSessionId: {playerSessionId}");
var req = new AcceptPlayerSessionRequest{
GameServerSessionId = gameServerSession.GameServerSessionId,
PlayerSessionId = playerSessionId,
};
return GrpcClient.GseClient.AcceptPlayerSession(req, meta);
}
```

6. 游戏结束或者玩家退出后，Game Server 调用 RemovePlayerSession 接口移除玩家，将 playersession 状态更改为“已完成”， 并预留游戏服务器会话中的玩家位置。

```
public static GseResponse RemovePlayerSession(string playerSessionId)
{
logger.Println($"Removing player session, PlayerSessionId: {playerSessionId}");
var req = new RemovePlayerSessionRequest{
GameServerSessionId = gameServerSession.GameServerSessionId,
PlayerSessionId = playerSessionId,
};
return GrpcClient.GseClient.RemovePlayerSession(req, meta);
}
```

7. 当一个游戏服务器会话（一组游戏对局或一个服务）结束后，Game Server 调用 TerminateGameServerSession 接口结束 GameServerSession，将 GameServerSession 状态更改为“已终止”。

```
public static GseResponse TerminateGameServerSession()
{
logger.Println($"Terminating game server session, GameServerSessionId: {gameServerSession.GameServerSessionId}");
var req = new TerminateGameServerSessionRequest{
GameServerSessionId = gameServerSession.GameServerSessionId
```

```
};
return GrpcClient.GseClient.TerminateGameServerSession(req, meta);
}
```

8. 当健康检查失败或缩容时，GSE 调用 OnProcessTerminate 接口结束游戏进程，缩容时依据是您在 GSE 控制台配置的 保护策略。

```
public override Task<gseresponse> OnProcessTerminate(ProcessTerminateRequest request, ServerCallContext context)
{
logger.Println($"OnProcessTerminate, request: {request}");
// 设置进程终止时间
GseManager.SetTerminationTime(request.TerminationTime);
//调以下两个接口，会立即结束游戏服务器会话，建议无玩家或无游戏服务器会话后，再调用ProcessEnding结束进程
//不调用以下两个接口，根据保护策略调用ProcessEnding结束进程，建议配置时限保护
// 终止游戏服务器会话
GseManager.TerminateGameServerSession();
// 进程退出
GseManager.ProcessEnding();
return Task.FromResult(new GseResponse{
Status = GseResponse.Types.Status.Ok,
ResponseData = "SUCCESS",
});
}
```

9. Game Server 调用 ProcessEnding 接口会立刻结束进程，将服务器进程状态更改为“已终止”，并回收资源。

```
//主动调用：一局游戏对应一个进程，当一局游戏结束后主动调用ProcessEnding接口
//被动调用：当缩容或进程异常健康检查失败时，根据保护策略被动调用ProcessEnding接口，配置完全保护和时限保护策略时需要先判断游戏服务器会话上有无玩家，再被动调用
public static GseResponse ProcessEnding()
{
logger.Println($"Process ending, pid: {pid}");
var req = new ProcessEndingRequest();
return GrpcClient.GseClient.ProcessEnding(req, meta);
}
```

0. Game Server 调用 DescribePlayerSessions 接口获取游戏服务器会话下的玩家信息（根据业务可选）。

```
public static DescribePlayerSessionsResponse DescribePlayerSessions(string gameServerSessionId, string playerId, string playerSessionId, string playerSessionStatusFilter, string nextToke
```

```
n, int limit)
{
logger.Println($"Describing player session, GameServerSessionId: {gameServerSessionId}, Player
Id: {playerId}, PlayerSessionId: {playerSessionId}, PlayerSessionStatusFilter: {playerSessionS
tatusFilter}, NextToken: {nextToken}, Limit: {limit}");
var req = new DescribePlayerSessionsRequest{
GameServerSessionId = gameServerSessionId,
PlayerId = playerId,
PlayerSessionId = playerSessionId,
PlayerSessionStatusFilter = playerSessionStatusFilter,
NextToken = nextToken,
Limit = limit,
};
return GrpcClient.GseClient.DescribePlayerSessions(req, meta);
}
```

1. Game Server 调用 UpdatePlayerSessionCreationPolicy 接口更新玩家会话的创建策略，设置是否接受新玩家，即游戏会话里是否允许加入人（根据业务可选）。

```
public static GseResponse UpdatePlayerSessionCreationPolicy(string newPolicy)
{
logger.Println($"Updating player session creation policy, newPolicy: {newPolicy}");
var req = new UpdatePlayerSessionCreationPolicyRequest{
GameServerSessionId = gameServerSession.GameServerSessionId,
NewPlayerSessionCreationPolicy = newPolicy,
};
return GrpcClient.GseClient.UpdatePlayerSessionCreationPolicy(req, meta);
}
```

2. Game Server 调用 ReportCustomData 接口告知 GSE 的自定义数据（根据业务可选）。

```
public static GseResponse ReportCustomData(int currentCustomCount, int maxCustomCount)
{
logger.Println($"Reporting custom data, CurrentCustomCount: {currentCustomCount}, MaxCustomCou
nt: {maxCustomCount}");
var req = new ReportCustomDataRequest{
CurrentCustomCount = currentCustomCount,
MaxCustomCount = maxCustomCount,
};
return GrpcClient.GseClient.ReportCustomData(req, meta);
}
```

版权所有：腾讯云计算（北京）有限责任公司

启动服务端，供 GSE 调用

服务端运行：将 GrpcServer 启动起来。

```
public class Program
{
public static int ClientPort = PortServer.GenerateRandomPort(2000, 6000);
public static int GrpcPort = PortServer.GenerateRandomPort(6001, 10000);

public static void Main(string[] args)
{
CreateHostBuilder(args).Build().Run();
}
public static IHostBuilder CreateHostBuilder(string[] args) =&gt;
Host.CreateDefaultBuilder(args)
.ConfigureWebHostDefaults(webBuilder =&gt;
{
webBuilder.ConfigureKestrel(options =&gt;
{
// gRPC 端口 (设置不带TLS证书的 HTTP/2 端点)
options.ListenAnyIP(GrpcPort, o =&gt; o.Protocols =
HttpProtocols.Http2);

// HTTP 端口
options.ListenAnyIP(ClientPort);
});

webBuilder.UseStartup<startup>();
});
}
```

客户端连接 GSE 的 gRPC 服务端

连接服务端：创建一个 gRPC 频道，指定我们要连接的主机名和服务器端口，然后用这个频道创建存根实例。

```
public class GrpcClient
{
private static string agentAdress = "127.0.0.1:5758";
public static GameServerGrpcSdkService.GameServerGrpcSdkServiceClient GameServerClient
{
get
{
Channel channel = new Channel(agentAdress, ChannelCredentials.Insecure);
```

```
return new GameServerGrpcSdkService.GameServerGrpcSdkServiceClient(channel);
}
}
public static GseGrpcSdkService.GseGrpcSdkServiceClient GseClient
{
get
{
Channel channel = new Channel(agentAdress, ChannelCredentials.Insecure);
return new GseGrpcSdkService.GseGrpcSdkServiceClient(channel);
}
}
}
```

C# DEMO

1. 单击这里，您可下载 C# DEMO 代码。
2. 生成 gRPC 代码。
   C# DEMO 代码示例里已生成 gRPC 代码，在 proto/csharp-demo 目录下，不需要额外生成。
3. 启动服务端，供 GSE 调用。
   - 服务端实现。
     在 csharp-demo/api 目录下的 gameserversdk.cs，实现了服务端的三个接口。
   - 服务端运行。
     在 csharp-demo 目录下的 Program.cs，将 GrpcServer 启动起来。
4. 客户端连接 GSE 的 gRPC 服务端。
   - 客户端实现。
     在 csharp-demo/Models 目录下的 GseManager.cs，实现了客户端的九个接口。
   - 连接服务端。
     创建一个 gRPC 频道，指定我们要连接的主机名和服务器端口，然后用这个频道创建存根实例。
5. 编译运行。
   i. 生成可执行文件及依赖

      ```
      dotnet publish -c Release -r linux-x64 --self-contained true
      ```

      以上会在 csharp-demo/bin/Release/netcoreapp3.1/linux-x64 目录下生成打包生成包所需要的所有依赖文件，其中即包含运行该服务的可执行文件csharpdemo。
   - 拷贝前置脚本 install.sh

      ```
      chmod u+x install.sh
      cp install.sh bin/Release/netcoreapp3.1/linux-x64
      ```

   - 打包 GSE 生成包

```
cd csharp-demo/bin/Release/netcoreapp3.1/linux-x64
zip -r csharpdemo.zip *
```

打包好的 csharpdemo.zip 即 GSE 需要的 生成包，启动路径填写 csharpdemo，无启动参数。

- 然后 创建服务器舰队，将生成包部署在服务器舰队上，后续可进行 扩缩容 等一系列操作。

gRPC Go 教程

最近更新时间：2021-11-17 18:06:09

安装 gRPC

1. 使用 gRPC Go 时，需要先安装 Go 的最新主要版本。
2. 安装 Protocol buffer 编辑器，版本为 protoc3。
3. 安装 Protocol buffer 编辑器里 Go 插件。
   - 使用以下命令为 Go（protoc-gen-go）安装 Protocol buffer 编译器插件：

```
$ export GO111MODULE=on # Enable module mode
$ go get github.com/golang/protobuf/protoc-gen-go
```

   - 更新路径以便 Protocol buffer 编译器找到 Go 插件：

```
$ export PATH="$PATH:$(go env GOPATH)/bin"
```

说明
具体流程请您参考 安装 Go 的说明，安装 Protocol buffer 编辑器的说明。

定义服务

gRPC 通过 protocol buffers 实现定义一个服务：一个 RPC 服务通过参数和返回类型来指定可以远程调用的方法。

说明
我们提供定义服务的 proto 文件，请您在 proto 文件 里下载使用，无需自己生成。

生成 gRPC 代码

1. 定义好服务后，通过 protocol buffer 编译器 protoc 生成客户端和服务端的代码（任意 gRPC 支持的语言）。

版权所有：腾讯云计算（北京）有限责任公司

2. 生成的代码包括客户端的存根和服务端要实现的抽象接口。

3. 生成 gRPC 代码步骤：
   在 proto 目录下执行：

```
protoc --go_out=plugins=grpc:. *.proto
```

会自动生成包含 proto 的 go_package 路径，而用户可以根据需要修改成适合自己的 go_package 路径，但不能修改 package。

```
## 游戏进程集成流程
![](https://main.qcloudimg.com/raw/96018551bc88c71a02333b1f197b3111.png)
#### Game Server 回调接口列表
| 接口名称 | 接口功能|
|-----|----|
|[OnHealthCheck](https://intl.cloud.tencent.com/document/product/1055/37422)| 健康检查|
|[OnStartGameServerSession](https://intl.cloud.tencent.com/document/product/1055/37423)|接收游戏服务器会话|
|[OnProcessTerminate](https://intl.cloud.tencent.com/document/product/1055/37424)|结束游戏进程|
#### Game Server 主调接口列表
| 接口名称 | 接口功能 |
|-----|----|
|[ProcessReady](https://intl.cloud.tencent.com/document/product/1055/37426)|进程准备就绪|
|[ActivateGameServerSession](https://intl.cloud.tencent.com/document/product/1055/37427)|激活游戏服务器会话|
|[AcceptPlayerSession](https://intl.cloud.tencent.com/document/product/1055/37428)|接收玩家会话|
|[RemovePlayerSession](https://intl.cloud.tencent.com/document/product/1055/37429)|移除玩家会话|
|[DescribePlayerSessions](https://intl.cloud.tencent.com/document/product/1055/37430)|获取玩家会话列表|
|[UpdatePlayerSessionCreationPolicy](https://intl.cloud.tencent.com/document/product/1055/37431)|更新玩家会话的创建策略|
|[TerminateGameServerSession](https://intl.cloud.tencent.com/document/product/1055/37432)|结束游戏服务器会话|
|[ProcessEnding](https://intl.cloud.tencent.com/document/product/1055/37434)|结束进程|
|[ReportCustomData](https://intl.cloud.tencent.com/document/product/1055/37435)|上报自定义数据|
#### 其他
请求 meta，在游戏进程通过 gRPC 调用 Game Server 主调接口时，需要在 gRPC 请求的 meta 里添加两个字段。
| 字段 | 含义 | 类型 |
| --------- | ----------------------------------------- | ------ |
| pid | 当前游戏进程的 pid | string |
```

```
| requestId | 当前请求的 requestId，已使用唯一标记一次请求 | string |
1. 一般在服务端初始化后，进程检查自身是否可对外提供服务，Game Server 调用 ProcessReady 接口，告知 GSE 进程准备就绪，已准备好托管游戏服务器会话，GSE 接收到后，将服务器实例状态更改为“活跃”。
```

Go

func (g *gsemanager) ProcessReady(logPath []string, clientPort int32, grpcPort int32) error {
logger.Info("start to processready", zap.Any("logPath", logPath), zap.Int32("clientPort", clientPort),

```
zap.Int32("grpcPort", grpcPort))
```

req := &grpcsdk.ProcessReadyRequest{

```
//日志路径
LogPathsToUpload: logPath,
//设置端口
ClientPort: clientPort,
GrpcPort: grpcPort,
```

}

_, err := g.rpcClient.ProcessReady(g.getContext(), req)
if err != nil {

```
logger.Info("ProcessReady fail", zap.Error(err))
return err
```

}

//准备就绪，可对外提供服务
logger.Info("ProcessReady success")
return nil
}

```
2. 进程准备就绪后，GSE 调用 OnHealthCheck 接口，对 Game Server 进行健康检查，每1分钟检查1次，连续3次失败就判定该进程不健康，不会分配游戏服务器会话至该进程。
```

Go

func _GameServerGrpcSdkService_OnHealthCheck_Handler(srv interface{}, ctx context.Context, dec func(interface{}) error, interceptor grpc.UnaryServerInterceptor) (interface{}, error) {
in := new(HealthCheckRequest)
if err := dec(in); err != nil {

```
return nil, err
```

}
if interceptor == nil {

```
return srv.(GameServerGrpcSdkServiceServer).OnHealthCheck(ctx, in)
```

}
info := &grpc.UnaryServerInfo{

```
Server: srv,
FullMethod: "/tencentcloud.gse.grpcsdk.GameServerGrpcSdkService/OnHealthCheck",
```

}
handler := func(ctx context.Context, req interface{}) (interface{}, error) {

```
return srv.(GameServerGrpcSdkServiceServer).OnHealthCheck(ctx, req.(*HealthCheckRequest))
```

}
return interceptor(ctx, in, info, handler)
}

```
3. 因为 Client 调用 [CreateGameServerSession](https://intl.cloud.tencent.com/document/product/1055/37139) 接口创建一个游戏服务器会话，将该游戏服务器会话分配给一个进程，所以触发 GSE 调用该进程的 onStartGameServerSession 接口，并且将 GameServerSession 状态更改为“激活中”。
```

Go

func _GameServerGrpcSdkService_OnStartGameServerSession_Handler(srv interface{}, ctx context.Context, dec func(interface{}) error, interceptor grpc.UnaryServerInterceptor) (interface{}, error) {

in := new(StartGameServerSessionRequest)

if err := dec(in); err != nil {

```
return nil, err
```

}

if interceptor == nil {

```
return srv.(GameServerGrpcSdkServiceServer).OnStartGameServerSession(ctx, in)
```

}

info := &grpc.UnaryServerInfo{

```
Server: srv,
FullMethod: "/tencentcloud.gse.grpcsdk.GameServerGrpcSdkService/OnStartGameServerSession",
```

}

handler := func(ctx context.Context, req interface{}) (interface{}, error) {

```
return srv.(GameServerGrpcSdkServiceServer).OnStartGameServerSession(ctx, req.(*StartGameServerSessionRequest))
```

}

return interceptor(ctx, in, info, handler)

}

```
4. 当 Game Server 收到 onStartGameServerSession，您自行处理一些逻辑或资源分配，准备就绪后，Game Server 就调用 ActivateGameServerSession 接口，通知 GSE 游戏服务器会话已分配给一个进程，现在已准备好接收玩家请求，将服务器状态更改为“活跃”。
```

Go

func (g *gsemanager) ActivateGameServerSession(gameServerSessionId string, maxPlayers int32) error {

logger.Info("start to ActivateGameServerSession", zap.String("gameServerSessionId", gameServerSessionId),

```
zap.Int32("maxPlayers", maxPlayers))
```

req := &grpcsdk.ActivateGameServerSessionRequest{

```
GameServerSessionId: gameServerSessionId,
MaxPlayers: maxPlayers,
```

}

_, err := g.rpcClient.ActivateGameServerSession(g.getContext(), req)
if err != nil {

```
logger.Error("ActivateGameServerSession fail", zap.Error(err))
return err
```

}

logger.Info("ActivateGameServerSession success")
return nil
}

```
5. 当 Client 调用 [JoinGameServerSession](https://intl.cloud.tencent.com/document/product/1055/39130) 接口玩家加入后，Game Server 调用 AcceptPlayerSession 接口验证玩家合法性，如果连接被接受，则将 PlayerSession 状态设置为“活跃”。如果 Client 调用 JoinGameServerSession 接口在60秒内未收到响应，则将 PlayerSession 状态更改为“超时”，然后重新调用 JoinGameServerSession。
```

func (g *gsemanager) AcceptPlayerSession(playerSessionId string) (*grpcsdk.GseResponse, error) {
logger.Info("start to AcceptPlayerSession", zap.String("playerSessionId", playerSessionId))
req := &grpcsdk.AcceptPlayerSessionRequest{

```
GameServerSessionId: g.gameServerSession.GameServerSessionId,
PlayerSessionId: playerSessionId,
```

}

return g.rpcClient.AcceptPlayerSession(g.getContext(), req)
}

```
6. 游戏结束或者玩家退出后，Game Server 调用 RemovePlayerSession 接口移除玩家，将 playersession 状态更改为“已完成” ，并预留游戏服务器会话中的玩家位置。
```

Go

func (g *gsemanager) RemovePlayerSession(playerSessionId string)* (grpcsdk.GseResponse, error) {
logger.Info("start to RemovePlayerSession", zap.String("playerSessionId", playerSessionId))
req := &grpcsdk.RemovePlayerSessionRequest{

```
GameServerSessionId: g.gameServerSession.GameServerSessionId,
PlayerSessionId: playerSessionId,
```

}

return g.rpcClient.RemovePlayerSession(g.getContext(), req)
}

```
7. 当一个游戏服务器会话（一组游戏对局或一个服务）结束后，Game Server 调用 TerminateGameServerSession 接口结束 GameServerSession，将 GameServerSession 状态更改为“已终止”。
```

Go

func (g *gsemanager) TerminateGameServerSession()* (grpcsdk.GseResponse, error) {
logger.Info("start to TerminateGameServerSession")
req := &grpcsdk.TerminateGameServerSessionRequest{

```
GameServerSessionId: g.gameServerSession.GameServerSessionId,
```

}

return g.rpcClient.TerminateGameServerSession(g.getContext(), req)
}

```
8. 当健康检查失败或缩容时，GSE 调用 OnProcessTerminate 接口结束游戏进程，缩容时依据是您在 GSE 控制台配置的 [保护策略](https://intl.cloud.tencent.com/document/product/1055/36675)。
```

Go

func _GameServerGrpcSdkService_OnProcessTerminate_Handler(srv interface{}, ctx context.Context, dec func(interface{}) error, interceptor grpc.UnaryServerInterceptor) (interface{}, error) {
in := new(ProcessTerminateRequest)
if err := dec(in); err != nil {

```
return nil, err
```

}
if interceptor == nil {

```
return srv.(GameServerGrpcSdkServiceServer).OnProcessTerminate(ctx, in)
```

}
info := &grpc.UnaryServerInfo{

```
Server: srv,
FullMethod: "/tencentcloud.gse.grpcsdk.GameServerGrpcSdkService/OnProcessTerminate",
```

}
handler := func(ctx context.Context, req interface{}) (interface{}, error) {

```
return srv.(GameServerGrpcSdkServiceServer).OnProcessTerminate(ctx, req.(*ProcessTerminateRequest))
```

}
return interceptor(ctx, in, info, handler)
}

```
9. Game Server 调用 ProcessEnding 接口会立刻结束进程，将服务器进程状态更改为“已终止”，并回收资源。
```

Go

//主动调用：一局游戏对应一个进程，当一局游戏结束后主动调用ProcessEnding接口
//被动调用：当缩容或进程异常健康检查失败时，根据保护策略被动调用ProcessEnding接口，配置完全保护和时限保护策略时需要先判断游戏服务器会话上有无玩家，再被动调用

func (g *gsemanager) ProcessEnding()* (grpcsdk.GseResponse, error) {
logger.Info("start to ProcessEnding")
req := &grpcsdk.ProcessEndingRequest{
}

return g.rpcClient.ProcessEnding(g.getContext(), req)
}

```
10. Game Server 调用 DescribePlayerSessions 接口获取游戏服务器会话下的玩家信息（根据业务可选）。
```

Go

func (g *gsemanager) DescribePlayerSessions(gameServerSessionId, playerId, playerSessionId, playerSessionStatusFilter, nextToken string,limit int32)* (grpcsdk.DescribePlayerSessionsResponse, error) {
logger.Info("start to DescribePlayerSessions", zap.String("gameServerSessionId", gameServerSessionId),

```
zap.String("playerId", playerId), zap.String("playerSessionId", playerSessionId),
zap.String("playerSessionStatusFilter", playerSessionStatusFilter), zap.String("nextToken", nextToken),
zap.Int32("limit", limit))
```

req := &grpcsdk.DescribePlayerSessionsRequest{

```
GameServerSessionId: gameServerSessionId,
PlayerId: playerId,
PlayerSessionId: playerSessionId,
PlayerSessionStatusFilter: playerSessionStatusFilter,
NextToken: nextToken,
Limit: limit,
```

}

return g.rpcClient.DescribePlayerSessions(g.getContext(), req)
}

```
11. Game Server 调用 UpdatePlayerSessionCreationPolicy 接口更新玩家会话的创建策略，设置是否接受新玩家，即游戏会话里是否允许加入人（根据业务可选）。
```

Go

func (g *gsemanager) UpdatePlayerSessionCreationPolicy(newPolicy string) (*grpcsdk.GseResponse, error) {
logger.Info("start to UpdatePlayerSessionCreationPolicy", zap.String("newPolicy", newPolicy))
req := &grpcsdk.UpdatePlayerSessionCreationPolicyRequest{

```
GameServerSessionId: g.gameServerSession.GameServerSessionId,
NewPlayerSessionCreationPolicy: newPolicy,
```

}

return g.rpcClient.UpdatePlayerSessionCreationPolicy(g.getContext(), req)
}

```
12. Game Server 调用 ReportCustomData 接口告知 GSE 的自定义数据（根据业务可选）。
```

Go

func (g *gsemanager) ReportCustomData(currentCustomCount, maxCustomCount int32) (*grpcsdk.GseResponse, error) {
logger.Info("start to UpdatePlayerSessionCreationPolicy", zap.Int32("currentCustomCount", currentCustomCount),

```
zap.Int32("maxCustomCount", maxCustomCount))
```

req := &grpcsdk.ReportCustomDataRequest{

```
CurrentCustomCount: currentCustomCount,
MaxCustomCount: maxCustomCount,
```

}

return g.rpcClient.ReportCustomData(g.getContext(), req)
}

```
## 启动服务端，供 GSE 调用
服务端运行：将 GrpcServer 启动起来。
```

Go

func (s *rpcService) StartGrpcServer() {
listen, err := net.Listen("tcp", "127.0.0.1:")
if err != nil {

```
logger.Fatal("grpc fail to listen", zap.Error(err))
```

}

addr := listen.Addr().String()
portStr := strings.Split(addr, ":")[1]
s.grpcPort, err = strconv.Atoi(portStr)
if err != nil {

```
logger.Fatal("grpc fail to get port",zap.Error(err))
```

}

logger.Info("grpc listen port is", zap.Int("port", s.grpcPort))

grpcServer := grpc.NewServer()
grpcsdk.RegisterGameServerGrpcSdkServiceServer(grpcServer, s)
logger.Info("start grpc server success")
go grpcServer.Serve(listen)
}

```
## 客户端连接 GSE 的 gRPC 服务端
连接服务端：创建一个 gRPC 频道，指定我们要连接的主机名和服务器端口，然后用这个频道创建存根实例。
```

Go

const (

```
localhost = "127.0.0.1"
agentPort = 5758
```

)
type gsemanager struct {

```
pid string
gameServerSession *grpcsdk.GameServerSession
terminationTime int64
rpcClient grpcsdk.GseGrpcSdkServiceClient
```

}

```
## Go DEMO
1. [单击这里](https://gsegrpcdemo-1301007756.cos.ap-guangzhou.myqcloud.com/go-demo.zip), 您可下载 Go DEMO 代码。
2. 生成 gRPC 代码。
Go DEMO 代码示例里已生成 gRPC 代码, 在 go-demo/grpcsdk 目录下, 不需要额外生成。
3. 启动服务端, 供 GSE 调用。
- 服务端实现。
在 go-demo/api 目录下的 grpcserver.go, 实现了服务端的三个接口。
- 服务端运行。
在 go-demo/api 目录下的grpcserver.go, 将 GrpcServer 启动起来。
4. 客户端连接GSE的gRPC服务端。
- 客户端实现。
在 go-demo/gsemanager 目录下的 gsemanager.go, 实现了客户端的九个接口。
- 连接服务端。
创建一个 gRPC 频道, 指定我们要连接的主机名和服务器端口, 然后用这个频道创建存根实例。
5. 编译运行。
1. 在 go-demo 目录下, 执行
```

go mod vendor

```
生成 vendor 目录。



◦ 编译命令:
```

go build -mod=vendor main.go

```
会生成对应的 go-demo 可执行文件 main.go。
```

4. 将可执行文件 main.go 打包为 [生成包](https://intl.cloud.tencent.com/document/product/1055/36674), 启动路径配置 main, 无启动参数。

5. 然后 [创建服务器舰队](https://intl.cloud.tencent.com/document/product/1055/36675)，将生成包部署在服务器舰队上，后续可进行 [扩缩容](https://intl.cloud.tencent.com/document/product/1055/37445) 等一系列操作。

gRPC Java 教程

最近更新时间：2021-11-17 18:06:09

安装 gRPC

1. Java gRPC 除了 JDK 外不需要其他工具。
2. 在本地安装 gRPC Java 库 SNAPSHOT，包括代码生成插件。

> 说明：
> 具体安装流程请您参考 安装 gRPC Java的说明。

定义服务

gRPC 通过 protocol buffers 实现定义一个服务：一个 RPC 服务通过参数和返回类型来指定可以远程调用的方法。

> 说明
> 我们提供定义服务的 proto 文件，请您在 proto 文件 里下载使用，无需自己生成。

生成 gRPC 代码

1. 定义好服务后，通过 protocol buffer 编译器 protoc 生成客户端和服务端的代码（任意 gRPC 支持的语言）。
2. 生成的代码包括客户端的存根和服务端要实现的抽象接口。
3. 生成 gRPC 代码步骤：
   - 方法一：在 java-demo/src/main/proto下执行脚本，需要从 gprc 官网下载 protoc 和 protoc-gen-grpc-java 生成工具：

```
sh gen_pb.sh
```

```
protoc --java_out=../java --proto_path=. GameServerGrpcSdkService.proto
protoc --plugin=protoc-gen-grpc-java=`which protoc-gen-grpc-java` --grpc-java_out=../java --proto_path=. GameServerGrpcSdkService.proto
```

```
protoc --java_out=../java --proto_path=. GseGrpcSdkService.proto
protoc --plugin=protoc-gen-grpc-java=`which protoc-gen-grpc-java` --grpc-java_out=../java --
proto_path=. GseGrpcSdkService.proto
```

- 方法二：使用 maven 工具生成 gRPC 代码，在 maven 中增加编译 grpc 代码的 maven 插件，详细信息请您参考 gRPC-Java-RPC 库和框架。

```
<build>
<extensions>
<extension>
<groupid>kr.motd.maven</groupid>
<artifactid>os-maven-plugin</artifactid>
<version>1.6.2</version>
</extension>
</extensions>
<plugins>
<plugin>
<groupid>org.xolstice.maven.plugins</groupid>
<artifactid>protobuf-maven-plugin</artifactid>
<version>0.6.1</version>
<configuration>
<protocartifact>com.google.protobuf:protoc:3.12.0:exe:${os.detected.classifier}</protocartif
act>
<pluginid>grpc-java</pluginid>
<pluginartifact>io.grpc:protoc-gen-grpc-java:1.30.2:exe:${os.detected.classifier}</pluginart
ifact>
</configuration>
<executions>
<execution>
<goals>
<goal>compile</goal>
<goal>compile-custom</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
```

游戏进程集成流程

Client
Game Server
gRPC framework
GSE
ProcessReady
OnHealthCheck
CreateGameServerSession
OnStartGameServerSession
ActivateGameServerSession
JoinGameServerSession
Return the "IP:port"or "domain name"
AcceptPlayerSession
RemovePlayerSession
TerminateGameServerSession
OnProcessTerminate
ProcessEnding
Destroy

Game Server 回调接口列表

接口名称	接口功能
OnHealthCheck	健康检查
OnStartGameServerSession	接收游戏服务器会话
OnProcessTerminate	结束游戏进程

Game Server 主调接口列表

版权所有：腾讯云计算（北京）有限责任公司

接口名称	接口功能
ProcessReady	进程准备就绪
ActivateGameServerSession	激活游戏服务器会话
AcceptPlayerSession	接收玩家会话
RemovePlayerSession	移除玩家会话
DescribePlayerSessions	获取玩家会话列表
UpdatePlayerSessionCreationPolicy	更新玩家会话的创建策略
TerminateGameServerSession	结束游戏服务器会话
ProcessEnding	结束进程
ReportCustomData	上报自定义数据

其他

请求 meta，在游戏进程通过 gRPC 调用 Game Server 主调接口时，需要在 gRPC 请求的 meta 里添加两个字段。

字段	含义	类型
pid	当前游戏进程的 pid	string
requestId	当前请求的 requestId，已使用唯一标记一次请求	string

1. 一般在服务端初始化后，进程检查自身是否可对外提供服务，Game Server 调用 ProcessReady 接口，告知 GSE 进程准备就绪，已准备好托管游戏服务器会话，GSE 接收到后，将服务器实例状态更改为“活跃”。

```
public GseResponseBo processReady(ProcessReadyRequestBo request) {
logger.info("processReady request=" + new Gson().toJson(request));
GseResponseBo responseBo = new GseResponseBo();
GseGrpcSdkServiceOuterClass.ProcessReadyRequest rpcRequest = GseGrpcSdkServiceOuterClass.ProcessReadyRequest
//设置端口。
.newBuilder().setClientPort(request.getClientPort())
.setGrpcPort(request.getGrpcPort())
//日志路径。
.addAllLogPathsToUpload(request.getLogPathsToUploadList()).build();
GseGrpcSdkServiceOuterClass.GseResponse rpcResponse;
try {
```

```
rpcResponse = getGseGrpcSdkServiceClient().processReady(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return createRpcFailedResponseBo(e.getStatus());
}
//准备就绪，可对外提供服务。
logger.info("processReady response=" + rpcResponse.toString());
return createResponseBoByRpcResponse(rpcResponse);
}
```

2. 进程准备就绪后，GSE 调用 OnHealthCheck 接口，对 Game Server 进行健康检查，每1分钟检查1次，连续3次失败就判定该进程不健康，不会分配游戏服务器会话至该进程。

```
public boolean onHealthCheck() {
//添加游戏服务器逻辑以进行健康检查。
boolean res = getGrpcServiceConfig().getGseGrpcSdkServiceClient().isProcessHealth();
logger.info("onHealthCheck status=" + res);
return res;
}
```

3. 因为 Client 调用 CreateGameServerSession 接口创建一个游戏服务器会话，将该游戏服务器会话分配给一个进程，所以触发 GSE 调用该进程的 onStartGameServerSession 接口，并且将 GameServerSession 状态更改为“激活中”。

```
public GseResponseBo onStartGameServerSession(GameServerSessionBo gameServerSessionBo) {
logger.info("onStartGameServerSession gameServerSession=" + new Gson().toJson(gameServerSessio
nBo));
//添加用于启动游戏服务器会话的游戏服务端逻辑。
//保存游戏服务器会话。
getGrpcServiceConfig().getGseGrpcSdkServiceClient().onStartGameServerSession(gameServerSession
Bo);
//激活游戏服务器会话。
ActivateGameServerSessionRequestBo activateRequest = new ActivateGameServerSessionRequestBo();
activateRequest.setGameServerSessionId(gameServerSessionBo.getGameServerSessionId());
activateRequest.setMaxPlayers(gameServerSessionBo.getMaxPlayers());
getGrpcServiceConfig().getGseGrpcSdkServiceClient().activateGameServerSession(activateReques
t);
//在此处添加最终逻辑。
return createResponseBo(0, "SUCCESS");
}
```

版权所有：腾讯云计算（北京）有限责任公司

4. 当 Game Server 收到 onStartGameServerSession，您自行处理一些逻辑或资源分配，准备就绪后，Game Server 就调用 ActivateGameServerSession 接口,通知 GSE 游戏服务器会话已分配给一个进程，现在已准备好接收玩家请求，将服务器状态更改为“活跃”。

```
public GseResponseBo activateGameServerSession(ActivateGameServerSessionRequestBo request) {
logger.info("activateGameServerSession request=" + new Gson().toJson(request));
if (gameServerSessionBo == null) {
return createResponseBo(Constants.gameServerSessionExpectCode, "no game server session found."
);
}
GseResponseBo responseBo = new GseResponseBo();
GseGrpcSdkServiceOuterClass.ActivateGameServerSessionRequest rpcRequest = GseGrpcSdkServiceOut
erClass.ActivateGameServerSessionRequest
.newBuilder().setMaxPlayers(request.getMaxPlayers())
.setGameServerSessionId(gameServerSessionBo.getGameServerSessionId()).build();
GseGrpcSdkServiceOuterClass.GseResponse rpcResponse;
try {
rpcResponse = getGseGrpcSdkServiceClient().activateGameServerSession(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return createRpcFailedResponseBo(e.getStatus());
}
logger.info("activateGameServerSession response=" + rpcResponse.toString());
return createResponseBoByRpcResponse(rpcResponse);
}
```

5. 当 Client 调用 JoinGameServerSession 接口玩家加入后，Game Server 调用 AcceptPlayerSession 接口验证玩家合法性，如果连接被接受，则将 PlayerSession 状态设置为“活跃”。如果 Client 调用 JoinGameServerSession 接口在60秒内未收到响应，则将 PlayerSession 状态更改为“超时”，然后重新调用 JoinGameServerSession。

```
public GseResponseBo acceptPlayerSession(PlayerSessionRequestBo request) {
logger.info("acceptPlayerSession request=" + new Gson().toJson(request));
if (gameServerSessionBo == null) {
return createResponseBo(Constants.gameServerSessionExpectCode, "no game server session found."
);
}
GseResponseBo responseBo = new GseResponseBo();
GseGrpcSdkServiceOuterClass.AcceptPlayerSessionRequest rpcRequest = GseGrpcSdkServiceOuterClas
s.AcceptPlayerSessionRequest
.newBuilder().setGameServerSessionId(gameServerSessionBo.getGameServerSessionId())
.setPlayerSessionId(request.getPlayerSessionId()).build();
GseGrpcSdkServiceOuterClass.GseResponse rpcResponse;
try {
```

```
rpcResponse = getGseGrpcSdkServiceClient().acceptPlayerSession(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return createRpcFailedResponseBo(e.getStatus());
}
logger.info("acceptPlayerSession response=" + rpcResponse.toString());
return createResponseBoByRpcResponse(rpcResponse);
}
```

6. 游戏结束或者玩家退出后，Game Server 调用 RemovePlayerSession 接口移除玩家，将 playersession 状态更改为“已完成”，并预留游戏服务器会话中的玩家位置。

```
public GseResponseBo removePlayerSession(PlayerSessionRequestBo request) {
logger.info("removePlayerSession request=" + new Gson().toJson(request));
if (gameServerSessionBo == null) {
return createResponseBo(Constants.gameServerSessionExpectCode, "no game server session found."
);
}
GseResponseBo responseBo = new GseResponseBo();
GseGrpcSdkServiceOuterClass.RemovePlayerSessionRequest rpcRequest = GseGrpcSdkServiceOuterClas
s.RemovePlayerSessionRequest
.newBuilder().setGameServerSessionId(gameServerSessionBo.getGameServerSessionId())
.setPlayerSessionId(request.getPlayerSessionId()).build();
GseGrpcSdkServiceOuterClass.GseResponse rpcResponse;
try {
rpcResponse = getGseGrpcSdkServiceClient().removePlayerSession(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return createRpcFailedResponseBo(e.getStatus());
}
logger.info("removePlayerSession response=" + rpcResponse.toString());
return createResponseBoByRpcResponse(rpcResponse);
}
```

7. 当一个游戏服务器会话（一组游戏对局或一个服务）结束后，Game Server 调用 TerminateGameServerSession 接口结束 GameServerSession，将 GameServerSession 状态更改为“已终止”。

```
public GseResponseBo terminateGameServerSession(String gameServerSessionId) {
logger.info("terminateGameServerSession request=" + gameServerSessionId);
if (StringUtils.isEmpty(gameServerSessionId) && gameServerSessionBo != null
&& !StringUtils.isEmpty(gameServerSessionBo.getGameServerSessionId())) {
gameServerSessionId = gameServerSessionBo.getGameServerSessionId();
}
```

```
if (StringUtils.isEmpty(gameServerSessionId)) {
return createResponseBo(Constants.gameServerSessionExpectCode, "no game server session found."
);
}
GseResponseBo responseBo = new GseResponseBo();
GseGrpcSdkServiceOuterClass.TerminateGameServerSessionRequest rpcRequest = GseGrpcSdkServiceOu
terClass.TerminateGameServerSessionRequest
.newBuilder().setGameServerSessionId(gameServerSessionId).build();
GseGrpcSdkServiceOuterClass.GseResponse rpcResponse;
try {
rpcResponse = getGseGrpcSdkServiceClient().terminateGameServerSession(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return createRpcFailedResponseBo(e.getStatus());
}
logger.info("terminateGameServerSession response=" + rpcResponse.toString());
return createResponseBoByRpcResponse(rpcResponse);
}
```

8. 当健康检查失败或缩容时，GSE 调用 OnProcessTerminate 接口结束游戏进程，缩容时依据是您在 GSE 控制台配置的 保护策略。

```
public GseResponseBo onProcessTerminate(long terminationTime) {
logger.info("onProcessTerminate terminationTime=" + terminationTime);
//现在可能结束游戏服务端。
return createResponseBo(0, "SUCCESS");
}
```

9. Game Server 调用 ProcessEnding 接口会立刻结束进程，将服务器进程状态更改为“已终止”，并回收资源。

```
//主动调用：一局游戏对应一个进程，当一局游戏结束后主动调用ProcessEnding接口
//被动调用：当缩容或进程异常健康检查失败时，根据保护策略被动调用ProcessEnding接口，配置完全保
护和时限保护策略时需要先判断游戏服务器会话上有无玩家，再被动调用
public GseResponseBo processEnding() {
logger.info("processEnding begin");
GseResponseBo responseBo = new GseResponseBo();
GseGrpcSdkServiceOuterClass.ProcessEndingRequest rpcRequest = GseGrpcSdkServiceOuterClass.Proc
essEndingRequest
.newBuilder().build();
GseGrpcSdkServiceOuterClass.GseResponse rpcResponse;
try {
rpcResponse = getGseGrpcSdkServiceClient().processEnding(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
```

版权所有：腾讯云计算（北京）有限责任公司

```
return createRpcFailedResponseBo(e.getStatus());
}
logger.info("processEnding response=" + rpcResponse.toString());
return createResponseBoByRpcResponse(rpcResponse);
}
```

0. Game Server 调用 DescribePlayerSessions 接口获取游戏服务器会话下的玩家信息（根据业务可选）。

```
public DescribePlayerSessionsResponseBo describePlayerSessions(DescribePlayerSessionsRequestBo
request) {
logger.info("describePlayerSessions request=" + new Gson().toJson(request));
if (StringUtils.isEmpty(request.getGameServerSessionId()) &&
gameServerSessionBo != null && !StringUtils.isEmpty(gameServerSessionBo.getGameServerSessionId
())) {
request.setGameServerSessionId(gameServerSessionBo.getGameServerSessionId());
}
GseGrpcSdkServiceOuterClass.DescribePlayerSessionsRequest rpcRequest = GseGrpcSdkServiceOuterC
lass.DescribePlayerSessionsRequest
.newBuilder().setGameServerSessionId(request.getGameServerSessionId())
.setLimit(request.getLimit())
.setNextToken(request.getNextToken())
.setPlayerId(request.getPlayerId())
.setPlayerSessionId(request.getPlayerSessionId())
.setPlayerSessionStatusFilter(request.getPlayerSessionStatusFilter()).build();
GseGrpcSdkServiceOuterClass.DescribePlayerSessionsResponse rpcResponse;
try {
rpcResponse = getGseGrpcSdkServiceClient().describePlayerSessions(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return null;
}
logger.info("describePlayerSessions response=" + rpcResponse.toString());
return toPlayerSessionsResponseBo(rpcResponse);
}
```

1. Game Server 调用 UpdatePlayerSessionCreationPolicy 接口更新玩家会话的创建策略，设置是否接受新玩家，即游戏会话里是否允许加入人（根据业务可选）。

```
public GseResponseBo updatePlayerSessionCreationPolicy(UpdatePlayerSessionCreationPolicyReques
tBo request) {
logger.info("updatePlayerSessionCreationPolicy request=" + new Gson().toJson(request));
if (gameServerSessionBo == null) {
return createResponseBo(Constants.gameServerSessionExpectCode, "no game server session found."
);
```

```
}
GseResponseBo responseBo = new GseResponseBo();
GseGrpcSdkServiceOuterClass.UpdatePlayerSessionCreationPolicyRequest rpcRequest = GseGrpcSdkSe
rviceOuterClass.UpdatePlayerSessionCreationPolicyRequest
.newBuilder().setGameServerSessionId(gameServerSessionBo.getGameServerSessionId())
.setNewPlayerSessionCreationPolicy(request.getNewPlayerSessionCreationPolicy()).build();
GseGrpcSdkServiceOuterClass.GseResponse rpcResponse;
try {
rpcResponse = getGseGrpcSdkServiceClient().updatePlayerSessionCreationPolicy(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return createRpcFailedResponseBo(e.getStatus());
}
logger.info("updatePlayerSessionCreationPolicy response=" + rpcResponse.toString());
return createResponseBoByRpcResponse(rpcResponse);
}
```

2. Game Server 调用 ReportCustomData 接口告知 GSE 的自定义数据（根据业务可选）。

```
public GseResponseBo reportCustomData(ReportCustomDataRequestBo request) {
logger.info("reportCustomData request=" + new Gson().toJson(request));
GseResponseBo responseBo = new GseResponseBo();
GseGrpcSdkServiceOuterClass.ReportCustomDataRequest rpcRequest = GseGrpcSdkServiceOuterClass.R
eportCustomDataRequest
.newBuilder()
.setCurrentCustomCount(request.getCurrentCustomCount())
.setMaxCustomCount(request.getMaxCustomCount()).build();
GseGrpcSdkServiceOuterClass.GseResponse rpcResponse;
try {
rpcResponse = getGseGrpcSdkServiceClient().reportCustomData(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return createRpcFailedResponseBo(e.getStatus());
}
logger.info("reportCustomData response=" + rpcResponse.toString());
return createResponseBoByRpcResponse(rpcResponse);
}
```

启动服务端，供 GSE 调用

服务端运行：将 GrpcServer 启动起来。

```
@Bean(name = "grpcService", initMethod = "startup", destroyMethod = "shutdown")
public GrpcService getGrpcService() {
GrpcServiceConfig grpcServiceConfig = new GrpcServiceConfig();
grpcServiceConfig.setGseGrpcSdkServiceClient(gseGrpcSdkServiceClient);
grpcServiceConfig.setGameServerGrpcPort(gameServerGrpcPort);
grpcServiceConfig.setGameServerToClientPort(gameServerToClientPort);
grpcServiceConfig.setTargetAddress(targetAddress);
grpcServiceConfig.setGameServerUploadLogPath(gameServerUploadLogPath);
GrpcService grpcService = new GrpcService(grpcServiceConfig);
return grpcService;
}
```

客户端连接 GSE 的 gRPC 服务端

连接服务端：创建一个 gRPC 频道，指定我们要连接的主机名和服务器端口，然后用这个频道创建存根实例。

```
public GseGrpcSdkServiceGrpc.GseGrpcSdkServiceBlockingStub getGseGrpcSdkServiceClient() {
// 这里的 “channel” 是一个频道，而不是 ManagedChannel，因此，此代码的职责不是关掉它。
//将频道传递给代码，使代码更易于测试和重用频道。
if (blockingStub == null) {
managedChannel = getGrpcChannel(targetAddress);
blockingStub = GseGrpcSdkServiceGrpc.newBlockingStub(managedChannel);
}
if (managedChannel.isShutdown() || managedChannel.isTerminated()) {
managedChannel.shutdownNow();
managedChannel = getGrpcChannel(targetAddress);
blockingStub = GseGrpcSdkServiceGrpc.newBlockingStub(managedChannel);
}
return blockingStub;
}
```

Java DEMO

1. 单击这里，您可下载 Java DEMO 代码。
2. 生成 gRPC 代码。
   Java DEMO 代码示例里已生成 gRPC 代码，在 java-demo/src/main/java/tencentcloud 目录下，不需要额外生成。
3. 启动服务端，供 GSE 调用。
   - 服务端实现。
     在 java-demo/src/main/java/com/tencentcloud/gse/gameserver/service/gamelogic/impl 目录下的

GameServerGrpcCallbackImpl.java，实现了服务端的三个接口。
- 服务端运行。
  在 java-demo/src/main/java/com/tencentcloud/gse/gameserver/config 目录下的 GameServerConfig.java，将 GrpcServer 启动起来。

4. 客户端连接 GSE 的 gRPC 服务端。
   - 客户端实现。
     在 java-demo/src/main/java/com/tencentcloud/gse/gameserver/service/gsegrpc/impl 目录下的 GseGrpcSdkServiceClientImpl.java，实现了客户端的九个接口。
   - 连接服务端。
     创建一个 gRPC 频道，指定我们要连接的主机名和服务器端口，然后用这个频道创建存根实例。
5. 编译运行。
   i. 安装 java 版本要求1.8及以上，linux 下可以使用 yum 安装 openjdk：

   ```
   yum install -y java-1.8.0-openjdk
   ```

   - 将代码下载，在 java-demo 目录下，使用 maven 构建生成 gse-gameserver-demo.jar，使用如下命令启动：

   ```
   java -jar gse-gameserver-demo.jar
   ```

   - 将可执行文件 gse-gameserver-demo.jar 打包为 生成包，启动路径配置 java，启动参数配置 jar gse-gameserver-demo.jar。
   - 然后 创建服务器舰队，将生成包部署在服务器舰队上，后续可进行 扩缩容 等一系列操作。

gRPC Lua 教程

最近更新时间：2021-11-17 18:06:10

安装 gRPC

1. 安装 gRPC，安装完后会生成 grpc_cpp_plugin 可执行程序，该程序在生成 gRPC 代码时需要。
2. 安装 protocol buffers。

> 说明
> 具体安装流程请您参考 安装 gRPC Lua 的说明，安装 protocol buffers 的说明。

定义服务

gRPC 通过 protocol buffers 实现定义一个服务：一个 RPC 服务通过参数和返回类型来指定可以远程调用的方法。

> 说明
> 我们提供定义服务的 proto 文件，请您在 proto 文件 里下载使用，无需自己生成。

生成 gRPC 代码

1. 定义好服务后，通过 protocol buffer 编译器 protoc 生成客户端和服务端的代码（任意 gRPC 支持的语言）。
2. 生成的代码包括客户端的存根和服务端要实现的抽象接口。
3. 生成 gRPC 代码步骤：
   Lua DEMO 依赖 C++ 框架，步骤同 C++ DEMO 一样，在 proto 目录下执行：
   ```
   protoc --cpp_out=. *.proto
   ```
   生成 pb.cc 和 pb.h 文件。
   ```
   protoc --grpc_out=. --plugin=protoc-gen-grpc=`which grpc_cpp_plugin` *.proto
   ```

生成对应的 gRPC 代码。

将生成的8个文件移到项目合适的位置。

游戏进程集成流程

Client

Game Server

gRPC framework

GSE

ProcessReady

OnHealthCheck

CreateGameServerSession

OnStartGameServerSession

ActivateGameServerSession

JoinGameServerSession

Return the "IP:port"or "domain name"

AcceptPlayerSession

RemovePlayerSession

TerminateGameServerSession

OnProcessTerminate

ProcessEnding

Destroy

Game Server 回调接口列表

接口名称	接口功能

接口名称	接口功能
OnHealthCheck	健康检查
OnStartGameServerSession	接收游戏服务器会话
OnProcessTerminate	结束游戏进程

Game Server 主调接口列表

接口名称	接口功能
ProcessReady	进程准备就绪
ActivateGameServerSession	激活游戏服务器会话
AcceptPlayerSession	接收玩家会话
RemovePlayerSession	移除玩家会话
DescribePlayerSessions	获取玩家会话列表
UpdatePlayerSessionCreationPolicy	更新玩家会话的创建策略
TerminateGameServerSession	结束游戏服务器会话
ProcessEnding	结束进程
ReportCustomData	上报自定义数据

其他

请求 meta，在游戏进程通过 gRPC 调用 Game Server 主调接口时，需要在 gRPC 请求的 meta 里添加两个字段。

字段	含义	类型
pid	当前游戏进程的 pid	string
requestId	当前请求的 requestId，已使用唯一标记一次请求	string

1. 一般在服务端初始化后，进程检查自身是否可对外提供服务，Game Server 调用 ProcessReady 接口，告知 GSE 进程准备就绪，已准备好托管游戏服务器会话，GSE 接收到后，将服务器实例状态更改为“活跃”。

```
static bool luaProcessReady(std::vector <std::string> &logPath, int clientPort, int grpcPort)
{
```

```
GseResponse reply;
//日志路径，设置端口
Status status = GGseManager->ProcessReady(logPath, clientPort, grpcPort, reply);
//准备就绪，可对外提供服务
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

2. 进程准备就绪后，GSE 调用 OnHealthCheck 接口，对 Game Server 进行健康检查，每1分钟检查1次，连续3次失败就判定该进程不健康，不会分配游戏服务器会话至该进程。

```
Status GameServerGrpcSdkServiceImpl::OnHealthCheck(ServerContext* context, const HealthCheckRequest* request, HealthCheckResponse* reply)
{
healthStatus = GSESDK()->exec("return OnHealthCheck()");
std::cout << "healthStatus=" << healthStatus << std::endl;
reply->set_healthstatus(healthStatus);
return Status::OK;
}
```

3. 因为 Client 调用 CreateGameServerSession 接口创建一个游戏服务器会话，将该游戏服务器会话分配给一个进程，所以触发 GSE 调用该进程的 onStartGameServerSession 接口，并且将 GameServerSession 状态更改为“激活中”。

```
Status GameServerGrpcSdkServiceImpl::OnStartGameServerSession(ServerContext* context, const StartGameServerSessionRequest* request, GseResponse* reply)
{
auto gameServerSession = request->gameserversession();
GGseManager->SetGameServerSession(gameServerSession);
std::ostringstream o;
o << "return OnStartGameServerSession('" << gameServerSession.gameserversessionid() << "'," <<
gameServerSession.maxplayers() << ")";
std::string luaCmd = o.str();
bool res = GSESDK()->exec(luaCmd);
return Status::OK;
}
```

4. 当 Game Server 收到 onStartGameServerSession，您自行处理一些逻辑或资源分配，准备就绪后，Game Server 就调用ActivateGameServerSession 接口，通知 GSE 游戏服务器会话已分配给一个进程，现在已准

版权所有：腾讯云计算（北京）有限责任公司

备好接收玩家请求，将服务器状态更改为“活跃”。

```
static bool luaActivateGameServerSession(const std::string &gameServerSessionId, int maxPlayer
s) {
GseResponse reply;
Status status = GGseManager->ActivateGameServerSession(gameServerSessionId, maxPlayers, repl
y);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

5. 当 Client 调用 JoinGameServerSession 接口玩家加入后，Game Server 调用 AcceptPlayerSession 接口验证玩家合法性，如果连接被接受，则将 PlayerSession 状态设置为“活跃”。如果 Client 调用 JoinGameServerSession 接口在60秒内未收到响应，则将 PlayerSession 状态更改为“超时”，然后重新调用 JoinGameServerSession。

```
static bool luaAcceptPlayerSession(const std::string &gameServerSessionId, const std::string &
playerSessionId) {
GseResponse reply;
Status status = GGseManager->AcceptPlayerSession(gameServerSessionId, playerSessionId, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

6. 游戏结束或者玩家退出后，Game Server 调用 RemovePlayerSession 接口移除玩家，将 playersession 状态更改为“已完成”， 并预留游戏服务器会话中的玩家位置。

```
static bool luaRemovePlayerSession(const std::string &gameServerSessionId, const std::string &
playerSessionId) {
GseResponse reply;
Status status = GGseManager->RemovePlayerSession(gameServerSessionId, playerSessionId, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

7. 当一个游戏服务器会话（一组游戏对局或一个服务）结束后，Game Server 调用 TerminateGameServerSession 接口结束 GameServerSession，将 GameServerSession 状态更改为“已终止”。

```
static bool luaTerminateGameServerSession(const std::string &gameServerSessionId) {
GseResponse reply;
Status status = GGseManager->TerminateGameServerSession(gameServerSessionId, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

8. 当健康检查失败或缩容时，GSE 调用 OnProcessTerminate 接口结束游戏进程，缩容时依据是您在 GSE 控制台配置的 保护策略。

```
Status GameServerGrpcSdkServiceImpl::OnProcessTerminate(ServerContext* context, const ProcessTerminateRequest* request, GseResponse* reply)
{
auto terminationTime = request->terminationtime();
std::to_string(terminationTime));
std::ostringstream o;
o << "OnProcessTerminate(" << terminationTime << ")";
std::string luaCmd = o.str();
GSESDK()->execWithNilResult(luaCmd);
return Status::OK;
}
```

9. Game Server 调用 ProcessEnding 接口会立刻结束进程，将服务器进程状态更改为“已终止”，并回收资源。

```
//主动调用：一局游戏对应一个进程，当一局游戏结束后主动调用ProcessEnding接口
//被动调用：当缩容或进程异常健康检查失败时，根据保护策略被动调用ProcessEnding接口，配置完全保护和时限保护策略时需要先判断游戏服务器会话上有无玩家，再被动调用
static bool luaProcessEnding() {
GseResponse reply;
Status status = GGseManager->ProcessEnding(reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
```

```
return true;
}
```

0. Game Server 调用 DescribePlayerSessions 接口获取游戏服务器会话下的玩家信息（根据业务可选）。

```
static bool luaDescribePlayerSessions(const std::string &gameServerSessionId,
const std::string &playerId,
const std::string &playerSessionId,
const std::string &playerSessionStatusFilter, const std::string &nextToken,
int limit) {
DescribePlayerSessionsResponse reply;
Status status = GGseManager->DescribePlayerSessions(gameServerSessionId,playerId, playerSessio
nId, playerSessionStatusFilter, nextToken, limit, reply);
GSESDK()->setDescribePlayerSessionsResponse(reply);
if (!status.ok()) {
return false;
}
return true;
}
```

1. Game Server 调用 UpdatePlayerSessionCreationPolicy 接口更新玩家会话的创建策略，设置是否接受新玩家，即游戏会话里是否允许加入人（根据业务可选）。

```
static bool luaUpdatePlayerSessionCreationPolicy(const std::string &newpolicy) {
GseResponse reply;
Status status = GGseManager->UpdatePlayerSessionCreationPolicy(newpolicy, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

2. Game Server 调用 ReportCustomData 接口告知 GSE 的自定义数据（根据业务可选）。

```
static bool luaReportCustomData(int currentCustomCount, int maxCustomCount) {
GseResponse reply;
Status status = GGseManager->ReportCustomData(currentCustomCount, maxCustomCount, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
```

```
    return true;
    }
```

启动服务端，供 GSE 调用

服务端运行：将 GrpcServer 启动起来。

```
// 启动grpc server
std::thread tGrpc(&GameServerGrpcSdkServiceImpl::StartGrpcServer, GGameServerGrpcSdkService);
sem_wait(&(GGameServerGrpcSdkService->sem));
auto grpcPort = GGameServerGrpcSdkService->GetGrpcPort();
```

客户端连接 GSE 的 gRPC 服务端

连接服务端：创建一个 gRPC 频道，指定我们要连接的主机名和服务器端口，然后用这个频道创建存根实例。

```
void GseManager::InitStub() {
auto channel = grpc::CreateChannel("127.0.0.1:5758", grpc::InsecureChannelCredentials());
stub_ = GseGrpcSdkService::NewStub(channel);
}
```

Lua DEMO

1. 单击这里，您可下载 Lua DEMO 代码。
2. 生成 gRPC 代码。
   Lua DEMO 依赖 C++ 框架，已生成的 gRPC 代码在 cpp-demo/source/grpcsdk 目录下，不需要额外生成。
3. 启动服务端，供 GSE 调用。
   - 服务端实现。
     在 lua-demo/source/api 目录下的 grpcserver.cpp，实现了服务端的三个接口。
   - 服务端运行。
     在 lua-demo 目录下的 main.cpp，将 GrpcServer 启动起来。
4. 客户端连接 GSE 的 gRPC 服务端。
   - 客户端实现。
     在 lua-demo/source/lua 目录下的 GSESdkHandleWrapper.cpp，实现了客户端的九个接口。
   - 连接服务端。
     创建一个 gRPC 频道，指定我们要连接的主机名和服务器端口，然后用这个频道创建存根实例。

5. 编译运行。
   i. 安装 cmake。
   - 安装 gcc，版本要求4.9以上。
   - 安装 luajit 开发包和 boost 开发包：

```
yum install -y luajit-devel
yum install -y boost-devel
yum install -y cmake
```

   - 将代码下载，在 lua-demo 目录下，执行以下命令：

```
mkdir build
cd build
cmake ..
make
cp ../source/lua/gse.lua .
```

     会生成对应的 lua-demo 可执行文件，执行 ./lua-demo 启动。
   - 将可执行文件 lua-demo.cpp 打包为 生成包，启动路径配置 lua-demo，无启动参数。
   - 然后 创建服务器舰队，将生成包部署在服务器舰队上，后续可进行 扩缩容 等一系列操作。

gRPC Nodejs 教程

最近更新时间：2022-01-21 11:48:44

安装 gRPC

1. 安装 gRPC，安装完后会生成 grpc_cpp_plugin 可执行程序，该程序在生成 gRPC 代码时需要。
2. 安装 protocol buffers。

> 说明
> 具体安装流程请您参考 安装 gRPC Lua 的说明，安装 protocol buffers 的说明。

定义服务

gRPC 通过 protocol buffers 实现定义一个服务：一个 RPC 服务通过参数和返回类型来指定可以远程调用的方法。

> 说明
> 我们提供定义服务的 proto 文件，请您在 proto 文件 里下载使用，无需自己生成。

生成 gRPC 代码

1. 定义好服务后，通过 protocol buffer 编译器 protoc 生成客户端和服务端的代码（任意 gRPC 支持的语言）。

2. 生成的代码包括客户端的存根和服务端要实现的抽象接口。

3. 生成 gRPC 代码步骤：

   i. Lua DEMO 依赖 C++ 框架，步骤同 C++ DEMO 一样，在 proto 目录下执行：

   ```
   protoc --cpp_out=. *.proto
   ```

版权所有：腾讯云计算（北京）有限责任公司

ii. 生成 pb.cc 和 pb.h 文件。

```
protoc --grpc_out=. --plugin=protoc-gen-grpc=`which grpc_cpp_plugin` *.proto
```

iii. 生成对应的 gRPC 代码。

iv. 将生成的8个文件移到项目合适的位置。

游戏进程集成流程

Client

Game Server

gRPC framework

GSE

ProcessReady

OnHealthCheck

CreateGameServerSession

OnStartGameServerSession

ActivateGameServerSession

JoinGameServerSession

Return the "IP:port"or "domain name"

AcceptPlayerSession

RemovePlayerSession

TerminateGameServerSession

OnProcessTerminate

ProcessEnding

Destroy

Game Server 回调接口列表

接口名称	接口功能
OnHealthCheck	健康检查
OnStartGameServerSession	接收游戏服务器会话
OnProcessTerminate	结束游戏进程

Game Server 主调接口列表

版权所有：腾讯云计算（北京）有限责任公司

接口名称	接口功能
ProcessReady	进程准备就绪
ActivateGameServerSession	激活游戏服务器会话
AcceptPlayerSession	接收玩家会话
RemovePlayerSession	移除玩家会话
DescribePlayerSessions	获取玩家会话列表
UpdatePlayerSessionCreationPolicy	更新玩家会话的创建策略
TerminateGameServerSession	结束游戏服务器会话
ProcessEnding	结束进程
ReportCustomData	上报自定义数据

其他

请求 meta，在游戏进程通过 gRPC 调用 Game Server 主调接口时，需要在 gRPC 请求的 meta 里添加两个字段。

字段	含义	类型
pid	当前游戏进程的 pid	string
requestId	当前请求的 requestId，已使用唯一标记一次请求	string

1. 一般在服务端初始化后，进程检查自身是否可对外提供服务，Game Server 调用 ProcessReady 接口，告知 GSE 进程准备就绪，已准备好托管游戏服务器会话，GSE 接收到后，将服务器实例状态更改为“活跃”。

```
static bool luaProcessReady(std::vector <std::string> &logPath, int clientPort, int grpcPort)
{
GseResponse reply;
//日志路径，设置端口
Status status = GGseManager->ProcessReady(logPath, clientPort, grpcPort, reply);
//准备就绪，可对外提供服务
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

2. 进程准备就绪后，GSE 调用 OnHealthCheck 接口，对 Game Server 进行健康检查，每1分钟检查1次，连续3次失败就判定该进程不健康，不会分配游戏服务器会话至该进程。

```
Status GameServerGrpcSdkServiceImpl::OnHealthCheck(ServerContext* context, const HealthCheckRequest* request, HealthCheckResponse* reply)
{
healthStatus = GSESDK()->exec("return OnHealthCheck()");
std::cout << "healthStatus=" << healthStatus << std::endl;
reply->set_healthstatus(healthStatus);
return Status::OK;
}
```

3. 因为 Client 调用 CreateGameServerSession 接口创建一个游戏服务器会话，将该游戏服务器会话分配给一个进程，所以触发 GSE 调用该进程的 onStartGameServerSession 接口，并且将 GameServerSession 状态更改为“激活中”。

```
Status GameServerGrpcSdkServiceImpl::OnStartGameServerSession(ServerContext* context, const StartGameServerSessionRequest* request, GseResponse* reply)
{
auto gameServerSession = request->gameserversession();
GGseManager->SetGameServerSession(gameServerSession);
std::ostringstream o;
o << "return OnStartGameServerSession('" << gameServerSession.gameserversessionid() << "'," <<
gameServerSession.maxplayers() << ")";
std::string luaCmd = o.str();
bool res = GSESDK()->exec(luaCmd);
return Status::OK;
}
```

4. 当 Game Server 收到 onStartGameServerSession，您自行处理一些逻辑或资源分配，准备就绪后，Game Server 就调用ActivateGameServerSession 接口，通知 GSE 游戏服务器会话已分配给一个进程，现在已准备好接收玩家请求，将服务器状态更改为“活跃”。

```
static bool luaActivateGameServerSession(const std::string &gameServerSessionId, int maxPlayers) {
GseResponse reply;
Status status = GGseManager->ActivateGameServerSession(gameServerSessionId, maxPlayers, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
```

```
}
return true;
}
```

5. 当 Client 调用 JoinGameServerSession 接口玩家加入后，Game Server 调用 AcceptPlayerSession 接口验证玩家合法性，如果连接被接受，则将 PlayerSession 状态设置为“活跃”。如果 Client 调用 JoinGameServerSession 接口在60秒内未收到响应，则将 PlayerSession 状态更改为“超时”，然后重新调用 JoinGameServerSession。

```
static bool luaAcceptPlayerSession(const std::string &gameServerSessionId, const std::string &
playerSessionId) {
GseResponse reply;
Status status = GGseManager->AcceptPlayerSession(gameServerSessionId, playerSessionId, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

6. 游戏结束或者玩家退出后，Game Server 调用 RemovePlayerSession 接口移除玩家，将 playersession 状态更改为“已完成”，并预留游戏服务器会话中的玩家位置。

```
static bool luaRemovePlayerSession(const std::string &gameServerSessionId, const std::string &
playerSessionId) {
GseResponse reply;
Status status = GGseManager->RemovePlayerSession(gameServerSessionId, playerSessionId, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

7. 当一个游戏服务器会话（一组游戏对局或一个服务）结束后，Game Server 调用 TerminateGameServerSession 接口结束 GameServerSession，将 GameServerSession 状态更改为“已终止”。

```
static bool luaTerminateGameServerSession(const std::string &gameServerSessionId) {
GseResponse reply;
Status status = GGseManager->TerminateGameServerSession(gameServerSessionId, reply);
```

版权所有：腾讯云计算（北京）有限责任公司

```
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

8. 当健康检查失败或缩容时，GSE 调用 OnProcessTerminate 接口结束游戏进程，缩容时依据是您在 GSE 控制台配置的 保护策略。

```
Status GameServerGrpcSdkServiceImpl::OnProcessTerminate(ServerContext* context, const ProcessTerminateRequest* request, GseResponse* reply)
{
auto terminationTime = request->terminationtime();
std::to_string(terminationTime));
std::ostringstream o;
o << "OnProcessTerminate(" << terminationTime << ")";
std::string luaCmd = o.str();
GSESDK()->execWithNilResult(luaCmd);
return Status::OK;
}
```

9. Game Server 调用 ProcessEnding 接口会立刻结束进程，将服务器进程状态更改为“已终止”，并回收资源。

```
//主动调用：一局游戏对应一个进程，当一局游戏结束后主动调用ProcessEnding接口
//被动调用：当缩容或进程异常健康检查失败时，根据保护策略被动调用ProcessEnding接口，配置完全保护和时限保护策略时需要先判断游戏服务器会话上有无玩家，再被动调用
static bool luaProcessEnding() {
GseResponse reply;
Status status = GGseManager->ProcessEnding(reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

0. Game Server 调用 DescribePlayerSessions 接口获取游戏服务器会话下的玩家信息（根据业务可选）。

```
static bool luaDescribePlayerSessions(const std::string &gameServerSessionId,
const std::string &playerId,
const std::string &playerSessionId,
const std::string &playerSessionStatusFilter, const std::string &nextToken,
```

```
int limit) {
DescribePlayerSessionsResponse reply;
Status status = GGseManager->DescribePlayerSessions(gameServerSessionId,playerId, playerSessionId, playerSessionStatusFilter, nextToken, limit, reply);
GSESDK()->setDescribePlayerSessionsResponse(reply);
if (!status.ok()) {
return false;
}
return true;
}
```

1. Game Server 调用 UpdatePlayerSessionCreationPolicy 接口更新玩家会话的创建策略，设置是否接受新玩家，即游戏会话里是否允许加入人（根据业务可选）。

```
static bool luaUpdatePlayerSessionCreationPolicy(const std::string &newpolicy) {
GseResponse reply;
Status status = GGseManager->UpdatePlayerSessionCreationPolicy(newpolicy, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

2. Game Server 调用 ReportCustomData 接口告知 GSE 的自定义数据（根据业务可选）。

```
static bool luaReportCustomData(int currentCustomCount, int maxCustomCount) {
GseResponse reply;
Status status = GGseManager->ReportCustomData(currentCustomCount, maxCustomCount, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

启动服务端，供 GSE 调用

服务端运行：将 GrpcServer 启动起来。

版权所有：腾讯云计算（北京）有限责任公司

```
// 启动grpc server
std::thread tGrpc(&GameServerGrpcSdkServiceImpl::StartGrpcServer, GGameServerGrpcSdkService);
sem_wait(&(GGameServerGrpcSdkService->sem));
auto grpcPort = GGameServerGrpcSdkService->GetGrpcPort();
```

客户端连接 GSE 的 gRPC 服务端

连接服务端：创建一个 gRPC 频道，指定我们要连接的主机名和服务器端口，然后用这个频道创建存根实例。

```
void GseManager::InitStub() {
auto channel = grpc::CreateChannel("127.0.0.1:5758", grpc::InsecureChannelCredentials());
stub_ = GseGrpcSdkService::NewStub(channel);
}
```

Lua DEMO

1. 单击这里，您可下载 Lua DEMO 代码。
2. 生成 gRPC 代码。
   Lua DEMO 依赖 C++ 框架，已生成的 gRPC 代码在 cpp-demo/source/grpcsdk 目录下，不需要额外生成。
3. 启动服务端，供 GSE 调用。
   - 服务端实现。
     在 lua-demo/source/api 目录下的 grpcserver.cpp，实现了服务端的三个接口。
   - 服务端运行。
     在 lua-demo 目录下的 main.cpp，将 GrpcServer 启动起来。
4. 客户端连接 GSE 的 gRPC 服务端。
   - 客户端实现。
     在 lua-demo/source/lua 目录下的 GSESdkHandleWrapper.cpp，实现了客户端的九个接口。
   - 连接服务端。
     创建一个 gRPC 频道，指定我们要连接的主机名和服务器端口，然后用这个频道创建存根实例。
5. 编译运行。
   i. 安装 cmake。
   - 安装 gcc，版本要求4.9以上。
   - 安装 luajit 开发包和 boost 开发包：

   ```
   yum install -y luajit-devel
   yum install -y boost-devel
   yum install -y cmake
   ```

- 将代码下载，在 lua-demo 目录下，执行以下命令：

  ```
  mkdir build
  cd build
  cmake ..
  make
  cp ../source/lua/gse.lua .
  ```

  会生成对应的 lua-demo 可执行文件，执行 ./lua-demo 启动。
- 将可执行文件 lua-demo.cpp 打包为 生成包，启动路径配置 lua-demo，无启动参数。
- 然后 创建服务器舰队，将生成包部署在服务器舰队上，后续可进行 扩缩容 等一系列操作。

版权所有：腾讯云计算（北京）有限责任公司

gRPC Unity 教程

最近更新时间：2021-12-08 15:50:09

本文介绍使用 Unity 接入 GSE SDK 的完整过程，主要包括两方面工作：
1.Unity 接入 gRPC。
2.Unity 接入 GSE SDK。

前提条件

已安装 Unity Hub、Unity IDE。

> 注意：
> 本文基于 Unity 引擎版本： 2018.3.5f1、2019.4.9f1，操作系统：MacOS。

Unity 接入 gRPC

gRPC 对 Unity 的支持仍处于实验阶段，更多信息可参见 README。具体操作步骤如下：

步骤1：创建 Unity 项目

需要创建一个针对 `.NET 4.x` 等效版本的 Unity 项目，由于 gRPC 使用的 API 仅在 `.NET 4.5+` 可用，所以这一步是必需的，通过**Edit**>**Project Setting**>**Player**>**Configuration**>**Scripting Runtime Version**进行

版权所有：腾讯云计算（北京）有限责任公司

设置。

Project Settings

Audio
Editor
Graphics
Input
Physics
Physics 2D
Player
Preset Manager
Quality
Script Execution Order
Tags and Layers
TextMesh Pro
Time
VFX

Player

Settings for PC, Mac & Linux Standalone

Icon

Resolution and Presentation

Splash Image

Other Settings

Rendering

Color Space* Gamma
Auto Graphics API for Windows
Auto Graphics API for Mac
Auto Graphics API for Linux
Color Gamut For Mac*
sRGB
Metal Editor Support*
Metal API Validation*
Metal Write-Only Backbuffer
Memoryless Depth Unused
Static Batching
Dynamic Batching
GPU Skinning*
Graphics Jobs (Experimental)*
Lightmap Encoding High Quality
Lightmap Streaming Enabled
Streaming Priority 0

Vulkan Settings

SRGB Write Mode*

Mac App Store Options

Bundle Identifier com.Company.ProductName
Build 0
Category public.app-category.games
Mac App Store Validation

Configuration

Scripting Runtime Version* .NET 4.x Equivalent
Scripting Backend Mono
Api Compatibility Level* .NET 4.x
C++ Compiler Configuration Release
Disable HW Statistics*
Scripting Define Symbols
Allow 'unsafe' Code
Active Input Handling* Input Manager

步骤2：下载grpc_unity_package

版权所有：腾讯云计算（北京）有限责任公司

下载 `grpc_unity_package.VERSION.zip` 的 最新开发版本。单击 `BuildId ID` 跳转到下载页面。

gRPC Packages

Official gRPC Releases

Commits corresponding to official gRPC release points and release candidates are tagged on GitHub.

To maximize usability, gRPC supports the standard way of adding dependencies in your language of choice (if there is one). In most languages, the gRPC runtime comes in form of a package available in your language's package manager.

For instructions on how to use the language-specific gRPC runtime in your project, please refer to the following:

- C++: follow the instructions under the `src/cpp` directory
- C#: NuGet package `Grpc`
- Dart: pub package `grpc`
- Go: `go get google.golang.org/grpc`
- Java: Use JARs from gRPC Maven Central Repository
- Kotlin: Use JARs from gRPC Maven Central Repository
- Node: `npm install grpc`
- Objective-C: Add `gRPC-ProtoRPC` dependency to podspec
- PHP: `pecl install grpc`
- Python: `pip install grpcio`
- Ruby: `gem install grpc`
- WebJS: follow the instructions in `grpc-web` repository

Daily Builds of `master` Branch

gRPC packages are built on a daily basis at the `HEAD` of the `master` branch and are archived here.

The current document (view source) is an XML feed pointing to the packages as they get built and uploaded. You can subscribe to this feed and fetch, deploy, and test the precompiled packages with your continuous integration infrastructure.

For stable release packages, please consult the above section and the common package manager for your language.

Timestamp	Commit	Build ID
2020-09-28T05:23:42-0700	80c98971dda8c944b9d5bb2af0c8e6dc4e408421	0d426dde-91b4-4c25-b5bf-e3da7b7433cd
2020-09-26T05:15:16-0700	80c98971dda8c944b9d5bb2af0c8e6dc4e408421	21a92080-db55-4c89-a409-646366b94363
2020-09-25T05:01:26-0700	303ce9ea3d9dd68be9d86cf62167565cefc27e65	a8c9edab-99ab-4171-ad2b-ec2478a348f4
2020-09-24T05:26:42-0700	5604d4d4414a76b83249b75c9278ffa65db480b4	50ba2730-08f2-4822-b6c9-6b2f9f5b4367
2020-09-18T05:01:46-0700	1f964e3b24f32b57255ced1e864d3386b98b89c2	edba4667-d461-4d79-957f-117bd615a53c
2020-09-17T05:04:34-0700	a11e4df082b6c72ff6a791d20f00ada3f34338be	2f15b256-1994-43dc-954c-5d822b518544
2020-09-16T05:10:42-0700	e834d0b34b43745b0839b6205e04cfb1a5924c3a	0334328e-2982-4a3a-8919-27957d52c97e
2020-09-15T05:34:27-0700	58b0233aebc7ea894799c532cf5356de46f900cc	a21124d2-c952-49b2-ad67-67fe67917c0c

单击 `c#` 目录下的 `grpc_unity_package.VERSION.zip` 即可下载成功。

Build: `0d426dde-91b4-4c25-b5bf-e3da7b7433cd` [`invocation`]
Timestamp: `2020-09-28T05:23:42-0700`
Branch: `master`
Commit: `80c98971dda8c944b9d5bb2af0c8e6dc4e408421`

gRPC `protoc` Plugins

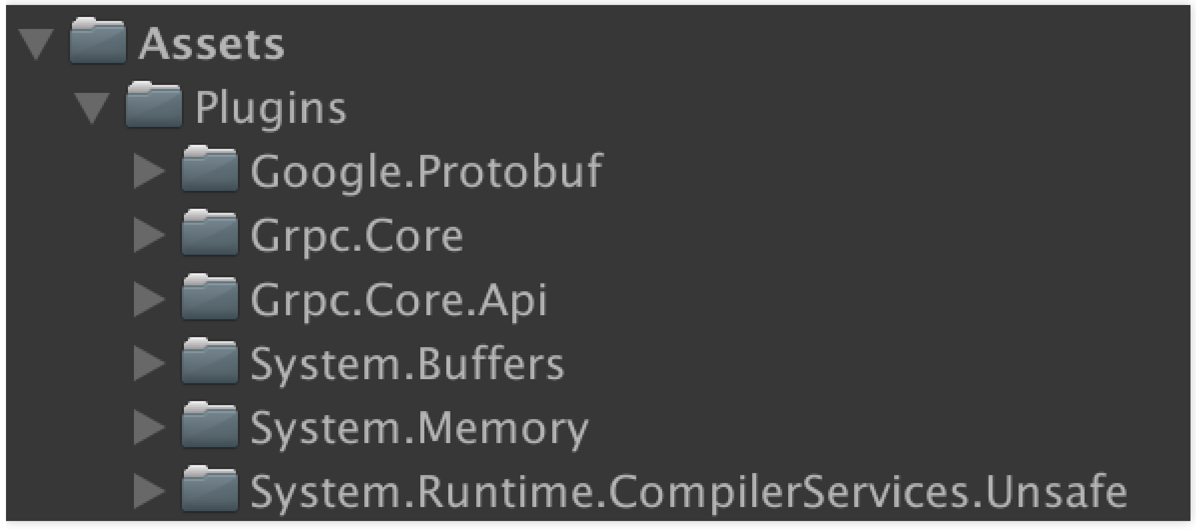
grpc-protoc_linux_x64-1.33.0-dev.tar.gz	c4b4a354363613bf011e85a4635ed6a193f710f849df8c109c093d7c022d5f72
grpc-protoc_linux_x86-1.33.0-dev.tar.gz	1fa6c3f6afb5cca9e5a70ff920324184b33302a48d61a259cf796434e34deaea
grpc-protoc_macos_x64-1.33.0-dev.tar.gz	e527428c70a8cd081ce515c6bbd045b787c61750c55e7d21e9555c0a181819fa
grpc-protoc_macos_x86-1.33.0-dev.tar.gz	fa13b3a80b5a0ab8966cd8020314529470f41788f7db18508005a01132578306
grpc-protoc_windows_x64-1.33.0-dev.zip	8dcdcb9563dd8dd0762c24aefabab56e197725eda181c3e2551df30d2e9d0eaa
grpc-protoc_windows_x86-1.33.0-dev.zip	9ca0e1f8e557545f4a873b791fd1e70d403add85fa2571fd666109c6da66cb02

C#

grpc_unity_package.2.33.0-dev.zip	18b5313758b2b7451e41d37bfc12992e6b61b055017a11b33b7a953b7e58b1a8
Grpc.2.33.0-dev202009281202.nupkg	01ca8739a97fbab5536036f16f2e30f17a0c1d9b409ee719b2991d58b6b929ac
Grpc.Auth.2.33.0-dev202009281202.nupkg	8d0b7294a82b093739f470cfab636bbfe08e62ed3177d463033acc75420793e4
Grpc.Core.2.33.0-dev202009281202.nupkg	57739eac5f4b995e16a570b202b45e10c315a9e85f036882b8c8a47294ce1409
Grpc.Core.Api.2.33.0-dev202009281202.nupkg	95ae31673348685bcfa11005123fd68fe33ee5428224c2090f4f407187ee8597
Grpc.Core.NativeDebug.2.33.0-dev202009281202.nupkg	824e20d42d3066e3ceaa17b3240f5ef8a605eac3258dff2c42f3ba6619489260
Grpc.Core.Testing.2.33.0-dev202009281202.nupkg	3bb4565eee60f6bc433107e62df5beff6442eafdb1a2d5742e46b9597cc898f1
Grpc.HealthCheck.2.33.0-dev202009281202.nupkg	546ac6a534496e67f0eac49a583feff0389fbb74225af93d88fc13d0256dab21
Grpc.Reflection.2.33.0-dev202009281202.nupkg	5f6f7ba895e1da7e6ddce02340d95bce0ce0133853b9ff28c5636f33b20fff87
Grpc.Tools.2.33.0-dev202009281202.nupkg	4a70f6bbf67a8b39a6470aa902ff2d424dee893b11a602a83a964d1de367443d

步骤3： 解压

将下载的 `.zip` 文件解压到 Unity 项目的 `Assets` 目录中，如下图所示



步骤4：测试

Unity Editor 将取出文件并自动添加到项目中，您即可在代码中使用 gRPC 和 Protobuf，如果 Unity Editor 提示错误，详情可参见 常见问题。

Unity 接入 GSE SDK

Unity 接入 GSE SDK 包括以下几个步骤：

步骤1： 获取 GSE SDK Protobuf 文件

获取 GSE SDK Protobuf 文件 `GameServerGrpcSdkService.proto` 和 `GseGrpcSdkService.proto`，详情可参见 proto 文件。

步骤2： 根据 Protobuf 生成 C# 代码

1. 下载 gRPC protoc Plugin，再次访问下载 grpc_unity_package.VERSION.zip 页面，下载对应操作系统的 protoc 压缩包。

版权所有：腾讯云计算（北京）有限责任公司

Build: edd81ac6-e3d1-461a-a263-2b06ae913c3f [invocation]
Timestamp: 2019-12-02T03:56:08-0800
Branch: master
Commit: a02d6b9be81cbadb60eed88b3b44498ba27bcba9

gRPC protoc Plugins

grpc-protoc_linux_x64-1.26.0-dev.tar.gz	534030f048c54b6d1c4788a7694a92fde11b36f1ef6386fe71fbc5ea6be73117
grpc-protoc_linux_x86-1.26.0-dev.tar.gz	8be18206322c8702fb3d3fd631f9a5218495bee856a7f65cb2693923175d550d
grpc-protoc_macos_x64-1.26.0-dev.tar.gz	51a1f7ba55dacd7f9ee7ddd5b445433b4d9ff9ca0166a08e99573827a379165e
grpc-protoc_macos_x86-1.26.0-dev.tar.gz	ac3419dea1589e367e6245cfd68489890928d6250de15b3d591ff48d3e9aecbc
grpc-protoc_windows_x64-1.26.0-dev.zip	8a76893e05a8d838572c6c4ff0be877029f50b6cc9de93cc3e9b045040784fdd
grpc-protoc_windows_x86-1.26.0-dev.zip	2b552c652d97cee13bd7356a38c642c8578b4c26444c20c09cd9c8e37faecb51

C#

grpc_unity_package.2.26.0-dev.zip	509af7278e725cc6a291d354365db930ea1bc96fe53bd96dc88cb33ed1c3e797
Grpc.2.26.0-dev201912021138.nupkg	5ac6ef42a6dbb17f15073abfc986764a18872437f0f357c6a17abfc93aaf20f7
Grpc.Auth.2.26.0-dev201912021138.nupkg	d0e2bb6538478ced3319d962dd1ac4519f0ab93ee7b16b87cafc903cebbda4c3
Grpc.Core.2.26.0-dev201912021138.nupkg	5efd93d0519ec9ccb91f04d0a6fa0f7c596fc7b40fe2f1506785c5de008de1fb
Grpc.Core.Api.2.26.0-dev201912021138.nupkg	822baa7faef5caa6b04666f8bd926c3d316e245f4960f351887f034df4a53541
Grpc.Core.NativeDebug.2.26.0-dev201912021138.nupkg	1211e8bd5336b612c866d84dc85bb32257a7907796da49b0f827000a7bbdd73e
Grpc.Core.Testing.2.26.0-dev201912021138.nupkg	4e5faa71af895c476418376fab48c48f65ac73d469049d490ec5dd4674201a91
Grpc.HealthCheck.2.26.0-dev201912021138.nupkg	a07b5e1399fcf8dcf94b0ca90f96bfdf84d582d69a7ddd03ec81baa79c460ef3
Grpc.Reflection.2.26.0-dev201912021138.nupkg	823c4dbece26be83a210c778b95bfa07e43d0a439851d67a247cbd048cea009c
Grpc.Tools.2.26.0-dev201912021138.nupkg	adb038ceac8b820f2875d66118775a42b2055fb9a0bb77fef3535a80cc56bb5e

版权所有：腾讯云计算（北京）有限责任公司

2. 将压缩包解压得到 `protoc` 和 `grpc_csharp_plugin` 可执行程序。

3. 拷贝 `protoc` 和 `grpc_csharp_plugin` 可执行程序到和 Protobuf 文件同一目录下，并在该目录下执行以下两条命令生成 `C#` 代码：
    - **MAC 和 Linux 环境命令如下：**
        - `protoc -I ./ --csharp_out=. GseGrpcSdkService.proto --grpc_out=. --plugin=protoc-gen-grpc=grpc_csharp_plugin`
        - `protoc -I ./ --csharp_out=. GameServerGrpcSdkService.proto --grpc_out=. --plugin=protoc-gen-grpc=grpc_csharp_plugin`
    - **Windows 环境命令如下：**
        - `./protoc -I ./ --csharp_out=. GseGrpcSdkService.proto --grpc_out=. --plugin=protoc-gen-grpc=grpc_csharp_plugin.exe`
        - `./protoc -I ./ --csharp_out=. GameServerGrpcSdkService.proto --grpc_out=. --plugin=protoc-gen-grpc=grpc_csharp_plugin.exe`

版权所有：腾讯云计算（北京）有限责任公司

如下图所示生成四个 `.cs` 代码文件。

步骤3： Unity 服务端开发使用 GSE SDK

将 步骤2 中生成的四个 `.cs` 文件拷贝到 Unity 项目中（可以拷贝到 Assets/Scripts/目录下单独的文件夹中），便可使用 GSE SDK 进行开发，详情可参见 Unity DEMO。

1. 实现 `gameserver_grpcsdk_service.proto` 定义的三个接口 `OnHealthCheck` 、 `OnStartGameServerSession` 和 `OnProcessTerminate` 。

```
public class GrpcServer : GameServerGrpcSdkService.GameServerGrpcSdkServiceBase
{
private static Logs logger
{
get
{
return new Logs();
```

版权所有：腾讯云计算（北京）有限责任公司

```
}
}
// 健康检查
public override Task<HealthCheckResponse> OnHealthCheck(HealthCheckRequest request, ServerCall
Context context)
{
logger.Println($"OnHealthCheck, HealthStatus: {GseManager.HealthStatus}");
logger.Println($"OnHealthCheck, GameServerSession: {GseManager.GetGameServerSession()}");
return Task.FromResult(new HealthCheckResponse
{
HealthStatus = GseManager.HealthStatus
});
}
// 接收游戏会话
public override Task<GseResponse> OnStartGameServerSession(StartGameServerSessionRequest reque
st, ServerCallContext context)
{
logger.Println($"OnStartGameServerSession, request: {request}");
GseManager.SetGameServerSession(request.GameServerSession);
var resp = GseManager.ActivateGameServerSession(request.GameServerSession.GameServerSessionId,
request.GameServerSession.MaxPlayers);
logger.Println($"OnStartGameServerSession, resp: {resp}");
return Task.FromResult(resp);
}
// 结束游戏进程
public override Task<GseResponse> OnProcessTerminate(ProcessTerminateRequest request, ServerCa
llContext context)
{
logger.Println($"OnProcessTerminate, request: {request}");
// 设置进程终止时间
GseManager.SetTerminationTime(request.TerminationTime);
// 终止游戏服务器会话
GseManager.TerminateGameServerSession();
// 进程退出
GseManager.ProcessEnding();
return Task.FromResult(new GseResponse());
}
}
```

2. 开发 Unity 服务端程序（以 ChatServer 为例）。

```
public static void StartChatServer(int clientPort)
{
RegisterHandlers();
logger.Println("ChatServer Listen at " + clientPort);
```

版权所有：腾讯云计算（北京）有限责任公司

```
NetworkServer.Listen(clientPort);
}
```

3. 开发 gRPC 服务端。

```
public static void StartGrpcServer(int clientPort, int grpcPort, string logPath)
{
try
{
Server server = new Server
{
Services = { GameServerGrpcSdkService.BindService(new GrpcServer()) },
Ports = { new ServerPort("127.0.0.1", grpcPort, ServerCredentials.Insecure) },
};
server.Start();
logger.Println("GrpcServer Start On localhost:" + grpcPort);
GseManager.ProcessReady(new string[] { logPath }, clientPort, grpcPort);
}
catch (System.Exception e)
{
logger.Println("error: " + e.Message);
}
}
```

4. 启动开发者本身实现的服务端和 gRPC 服务端。

```
public class StartServers : MonoBehaviour
{
private int grpcPort = PortServer.GenerateRandomPort(2000, 6000);
private int chatPort = PortServer.GenerateRandomPort(6001, 10000);
private const string logPath = "./log/log.txt";
// Start is called before the first frame update
[Obsolete]
void Start()
{
// Start ChatServer By UNet's NetWorkServer, Listen on UDP protocol
MyChatServer.StartChatServer(chatPort);
// Start GrpcServer By Grpc, Listen on TCP protocol
MyGrpcServer.StartGrpcServer(chatPort, grpcPort, logPath);
}
[Obsolete]
void OnGUI()
{
}
}
```

版权所有：腾讯云计算（北京）有限责任公司

Unity DEMO

1. 单击这里，您可以下载 Unity DEMO代码。
2. 导入 grpc unity package。
   将 步骤2 中 grpc_unity_package 解压到 Demo 工程 unity-demo/Assets 目录下。
3. 根据 Protobuf 文件生成 C# 代码。
4. 启动服务端，供 GSE 调用。
   - 服务端实现，在 `unity-demo/Assets/Scripts/Api` 目录下的 `GrpcServer.cs` 文件中实现服务端的三个接口。
   - 服务端运行，在 `unity-demo/Assets/Scripts` 目录下的 `MyGrpcServer.cs` 文件中，创建 `gRPC Server` ，`StartServers.cs` 从而启动 `gRPC Server` 。
5. 客户端连接 GSE 的 gRPC 服务端。
   - 客户端实现，在 `unity-demo/Assets/Scripts/Gsemanager` 目录下的 `Gsemanager.cs` 文件实现客户端的九个接口。
   - 连接服务端，创建一个 gRPC channel，指定要连接的主机和服务器端口，然后使用此 channel 创建存根实例。
6. 编译运行。
   使用 Unity Editor 打包目标系统的可执行程序，并打包为生成包，启动路径配置可执行程序名（需根据实际的可执行程序名称填写）。

获取服务器地址

云 API 调用方式

最近更新时间：2021-04-20 15:04:29

客户端 API 以云 API 的形式提供，具体调用方式有以下三种。

1. SDK 调用

通过腾讯云开发者工具套件（SDK）3.0 调用客户端云 API，支持语言包括 PHP、Python、Java、Go、.NET、Node.js、C++。

> **说明：**
>
> GSE 已支持 SDK 3.0版本，具体操作请参见 SDK 简介。

2. 在线调试

通过 API Explorer 工具调用客户端云 API，该工具提供了在线调用、签名验证、SDK 代码生成和快速检索接口等能力。

> **说明：**
>
> 在 API 3.0 Explorer 里，选择“游戏服务器伸缩”产品，选择“控制台相关接口”或“服务管理相关接口”里的云 API，可进行在线调试。

3. 直接封装

通过域名、接口名的 HTTP 请求方法调用客户端云 API。

> **说明：**
>
> GSE 的云 API 已升级到3.0版本，具体操作请参见 云 API 调用方式。

版权所有：腾讯云计算（北京）有限责任公司

创建游戏服务器会话

最近更新时间：2021-12-08 15:45:19

简介

- 游戏开发者可通过客户端云 API 创建游戏服务器会话，有两种创建方式：
  - 通过服务器舰队创建，可实现弹性伸缩、健康检查的功能；
  - 通过别名创建，可实现不停服更新的功能。
- 一个服务器进程放置一个游戏服务器会话，但根据不同的游戏服务器会话支持方式，有不同的客户端 API 调用流程。

客户端 API 调用流程

一个游戏服务器会话支持一局游戏

当一个游戏服务器会话仅支持一局游戏时，客户端 API 的调用流程如下：

1. 首先通过服务器舰队或别名创建一个游戏服务器会话，具体操作请参见 创建游戏服务器会话 API 文档。

> 说明：
> 以下示例均基于 Java 语言。

```
public class CreateGameServerSession
{
public static void main(String [] args) {
try{
Credential cred = new Credential("", "");
HttpProfile httpProfile = new HttpProfile();
httpProfile.setEndpoint("gse.tencentcloudapi.com");
ClientProfile clientProfile = new ClientProfile();
clientProfile.setHttpProfile(httpProfile);
GseClient client = new GseClient(cred, "", clientProfile);
String params = "{}";
CreateGameServerSessionRequest req = CreateGameServerSessionRequest.fromJsonString(params, CreateGameServerSessionRequest.class);
CreateGameServerSessionResponse resp = client.CreateGameServerSession(req);
System.out.println(CreateGameServerSessionRequest.toJsonString(resp));
```

```
} catch (TencentCloudSDKException e) {
System.out.println(e.toString());
}
}
}
```

2. 其次加入已创建好的游戏服务器会话，具体操作请参见 加入游戏服务器会话 API 文档。

```
public class JoinGameServerSession
{
public static void main(String [] args) {
try{
Credential cred = new Credential("", "");
HttpProfile httpProfile = new HttpProfile();
httpProfile.setEndpoint("gse.tencentcloudapi.com");
ClientProfile clientProfile = new ClientProfile();
clientProfile.setHttpProfile(httpProfile);
GseClient client = new GseClient(cred, "", clientProfile);
String params = "{}";
JoinGameServerSessionRequest req = JoinGameServerSessionRequest.fromJsonString(params, JoinGameSe
rverSessionRequest.class);
JoinGameServerSessionResponse resp = client.JoinGameServerSession(req);
System.out.println(JoinGameServerSessionRequest.toJsonString(resp));
} catch (TencentCloudSDKException e) {
System.out.println(e.toString());
}
}
}
```

一个游戏服务器会话支持多局游戏或一个服务

当一个游戏服务器会话支持多局游戏或一个服务（如登录服）时，客户端 API 的调用流程如下：

1. 首先进行查询游戏服务器会话列表，判断游戏服务器会话是否存在，具体操作请参见 查询游戏服务器会话列表 API 文档。

```
public class DescribeGameServerSessions
{
public static void main(String [] args) {
try{
Credential cred = new Credential("", "");
HttpProfile httpProfile = new HttpProfile();
httpProfile.setEndpoint("gse.tencentcloudapi.com");
```

```
ClientProfile clientProfile = new ClientProfile();
clientProfile.setHttpProfile(httpProfile);
GseClient client = new GseClient(cred, "", clientProfile);
String params = "{}";
DescribeGameServerSessionsRequest req = DescribeGameServerSessionsRequest.fromJsonString(param
s, DescribeGameServerSessionsRequest.class);
DescribeGameServerSessionsResponse resp = client.DescribeGameServerSessions(req);
System.out.println(DescribeGameServerSessionsRequest.toJsonString(resp));
} catch (TencentCloudSDKException e) {
System.out.println(e.toString());
}
}
}
```

也可通过搜索游戏服务器会话列表，判断游戏服务器会话是否存在，具体操作请参见 搜索游戏服务器会话列表 API 文档。

```
public class SearchGameServerSessions
{
public static void main(String [] args) {
try{
Credential cred = new Credential("", "");
HttpProfile httpProfile = new HttpProfile();
httpProfile.setEndpoint("gse.tencentcloudapi.com");
ClientProfile clientProfile = new ClientProfile();
clientProfile.setHttpProfile(httpProfile);
GseClient client = new GseClient(cred, "", clientProfile);
String params = "{}";
SearchGameServerSessionsRequest req = SearchGameServerSessionsRequest.fromJsonString(params, S
earchGameServerSessionsRequest.class);
SearchGameServerSessionsResponse resp = client.SearchGameServerSessions(req);
System.out.println(SearchGameServerSessionsRequest.toJsonString(resp));
} catch (TencentCloudSDKException e) {
System.out.println(e.toString());
}
}
}
```

2. 当游戏服务器会话已存在时，可直接加入游戏服务器会话，具体操作请参见 加入游戏服务器会话 API 文档或本文 代码示例。

3. 当游戏服务器会话不存在时，需要先创建游戏服务器会话，具体操作请参见 创建游戏服务器会话 API 文档或本文 代码示例。再加入游戏服务器会话，具体操作请参见 加入游戏服务器会话 API 文档或本文 代码示例。

版权所有：腾讯云计算（北京）有限责任公司

说明：

可通过 API 3.0 Explorer 进行在线调试，选择左侧“游戏服务器伸缩”/“服务管理相关接口”里相应的云 API，然后可进行“代码生成”、“在线调用”等操作。

放置游戏服务器会话

最近更新时间：2021-04-20 15:08:38

简介

游戏开发者可通过客户端云 API 放置游戏服务器会话，即通过游戏服务器队列，实现就近调度、跨地域容灾的功能。

客户端 API 调用流程

1. 首先查询游戏服务器会话的放置，判断游戏服务器会话是否放置在进程里，具体操作请参见 查询游戏服务器会话的放置 API 文档。

> **说明：**
> 以下示例均基于 Java 语言。

```
public class DescribeGameServerSessionPlacement
{
public static void main(String [] args) {
try{

Credential cred = new Credential("", "");

HttpProfile httpProfile = new HttpProfile();
httpProfile.setEndpoint("gse.tencentcloudapi.com");

ClientProfile clientProfile = new ClientProfile();
clientProfile.setHttpProfile(httpProfile);

GseClient client = new GseClient(cred, "", clientProfile);

String params = "{}";
DescribeGameServerSessionPlacementRequest req = DescribeGameServerSessionPlacementRequest.fromJsonString(params, DescribeGameServerSessionPlacementRequest.class);

DescribeGameServerSessionPlacementResponse resp = client.DescribeGameServerSessionPlacement(req);

System.out.println(DescribeGameServerSessionPlacementRequest.toJsonString(resp));
} catch (TencentCloudSDKException e) {
```

```
System.out.println(e.toString());
}
}
}
```

2. 其次可开始放置游戏服务器会话，具体操作请参见 开始放置游戏服务器会话 API 文档。

```
public class StartGameServerSessionPlacement
{
public static void main(String [] args) {
try{

Credential cred = new Credential("", "");

HttpProfile httpProfile = new HttpProfile();
httpProfile.setEndpoint("gse.tencentcloudapi.com");

ClientProfile clientProfile = new ClientProfile();
clientProfile.setHttpProfile(httpProfile);

GseClient client = new GseClient(cred, "", clientProfile);

String params = "{}";
StartGameServerSessionPlacementRequest req = StartGameServerSessionPlacementRequest.fromJsonStrin
g(params, StartGameServerSessionPlacementRequest.class);

StartGameServerSessionPlacementResponse resp = client.StartGameServerSessionPlacement(req);

System.out.println(StartGameServerSessionPlacementRequest.toJsonString(resp));
} catch (TencentCloudSDKException e) {
System.out.println(e.toString());
}
}
}
```

3. 最后停止放置游戏服务器会话，具体操作请参见 停止放置游戏服务器会话 API 文档。

```
public class StopGameServerSessionPlacement
{
public static void main(String [] args) {
try{

Credential cred = new Credential("", "");

HttpProfile httpProfile = new HttpProfile();
httpProfile.setEndpoint("gse.tencentcloudapi.com");
```

```
ClientProfile clientProfile = new ClientProfile();
clientProfile.setHttpProfile(httpProfile);

GseClient client = new GseClient(cred, "", clientProfile);

String params = "{}";
StopGameServerSessionPlacementRequest req = StopGameServerSessionPlacementRequest.fromJsonString(
params, StopGameServerSessionPlacementRequest.class);

StopGameServerSessionPlacementResponse resp = client.StopGameServerSessionPlacement(req);

System.out.println(StopGameServerSessionPlacementRequest.toJsonString(resp));
} catch (TencentCloudSDKException e) {
System.out.println(e.toString());
}
}
}
```

说明：

可通过 API 3.0 Explorer 进行在线调试，选择左侧“游戏服务器伸缩”/“服务管理相关接口”里相应的云 API，然后可进行“代码生成”、“在线调用”等操作。

版权所有：腾讯云计算（北京）有限责任公司

本地调试工具

最近更新时间：2021-06-28 10:10:16

GSE Local 说明

GSE Local 是一个命令行工具，可启动托管服务 GSE 的独立版本。GSE Local 还提供了服务器进程初始化、运行状况检查以及 API 调用和响应的运行事件日志。

使用 GSE Local 可以在本地设备上运行托管服务 GSE 的有限版本，并在其上测试您的游戏集成。使用该工具，能够在游戏进行迭代开发时极大减少调试时间，提升效率。其替代方法是将每个新生成的游戏包上传到 GSE 并配置服务器舰队来托管游戏。

通过 GSE，您可以验证以下内容：

- 您的游戏服务器与 GSE 服务端开发工具包正确集成，并且正确与 GSE 服务通信，启动新游戏会话、接受新玩家和报告运行状况及状态。
- 您的游戏客户端与适用于 GSE 的云 API 正确集成，可以检索现有游戏会话的信息，启动新游戏会话，让玩家加入游戏并连接到游戏会话。

设置 GSE Local

GSE Local 可以在 Windows、Linux 和 Mac上运行，并可用于任何 GSE 支持的语言。本地调试工具安装包如下：

- 下载 Windows 本地调试工具
- 下载 Linux 本地调试工具
- 下载 Mac 本地调试工具

> 说明：
> 以下示例代码适用于 Linux 和 macos 操作系统，在 Windows 操作系统下执行 curl 命令时推荐使用 gitbash 命令行工具。

测试游戏服务器

如果您只需测试游戏服务器，则可以直接使用 curl 方式来模拟游戏客户端对 GSE Local 服务的调用。这将验证您的游戏服务器是否按预期执行以下操作：

1. 在启动过程中，游戏服务器通知 GSE（调用 ProcessReady），服务器已准备好托管游戏服务器会话。
2. 在运行时，游戏服务器每分钟将运行状况发送到 GSE（onHealthCheak 回调）。
3. 游戏服务器响应请求，启动新游戏会话（触发 onStartGameServerSession 回调，回调中调用 activateGameServerSession）。

步骤1：启动 GSE Local

打开命令提示符窗口，导航到包含 gselocal_windows 或 gselocal_linux 或 gselocal_mac 文件的目录并运行它。本文以 mac 为例，本地启动 `./gselocal_mac`，程序启动将会自动连接到 GSE Local 上。
在终端输入以下命令：

```
./gselocal_mac
```

命令提示符窗口展示以下信息，则表示启动日志成功：

```
{"level":"info","ts":"2020-10-20T09:16:09.364+0800","msg":"start grpc v3 server success"}
```

步骤2：启动游戏服务器

在编程工具中或命令行工具里启动游戏进程， 游戏进程启动后将会调用 ProcessReady，表示进程已经做好托管会话的准备，进程将打印出以下日志信息：

```
Getting process ready, LogPath: System.String[], ClientPort: 3237, GrpcPort: 6224
Process ready succeed, resp: { }
Server Start On Locolhost:6224
```

Gse Local 此时收到 ProcessReady 请求，也打印出日志信息，并进行健康检查：

```
{"level":"info","ts":"2020-10-20T09:27:03.172+0800","msg":"ProcessReady Info is","pid":"41688","requestId":"3b38495b38bc4ef8a59ae8****a8256d","info":"clientPort:3237 grpcPort:6224 "}
{"level":"info","ts":"2020-10-20T09:27:03.172+0800","msg":"set runner success","pid":"41688","processUUID":"527bf89b-d128-4b5d-bfea-****3d22ede7"}
{"level":"info","ts":"2020-10-20T09:28:03.276+0800","msg":"onHealthCheck received","pid":"41688","health":true}
{"level":"info","ts":"2020-10-20T09:29:03.256+0800","msg":"onHealthCheck received","pid":"41688","health":true}
{"level":"info","ts":"2020-10-20T09:30:03.261+0800","msg":"onHealthCheck received","pid":"41688","health":true}
```

步骤3：使用 curl 调试创建游戏服务器会话和玩家会话

本地通过 curl 模拟客户端的调用。具体参数可参考 云 API 接口。

创建游戏服务器会话

对于 Local，FleetId 参数可以设置为任意有效字符串 `(^fleet-¥S+)`。执行以下命令配置 FleetId 参数：

```
curl -d '{"Action":"CreateGameServerSession", "FleetId":"fleet-1235", "MaximumPlayerSessionCount"
:5}' http://127.0.0.1:8080/capi
```

在 Local 命令提示符窗口中，日志消息指示 GSE Local 已向您的游戏服务器发送 onStartGameServerSession 回调。如果成功创建游戏服务器会话，您的游戏服务器将通过调用 ActivateGameServerSession 来响应。日志消息如下：

```
{"level":"info","ts":"2020-10-20T09:37:08.580+0800","msg":"API to use: GSE.CreateGameServerSessio
n, with input","req":"FleetId:<value:¥"fleet-1235¥" > MaximumPlayerSessionCount:<value:5 > "}
{"level":"info","ts":"2020-10-20T09:37:08.580+0800","msg":"Reserved process: 41688 for GameServer
Session: qcs::gse:local::gameserversession/fleet-1235/gssess-c648654a-293b-4f1f-b71f-****6a09bff
e"}
{"level":"info","ts":"2020-10-20T09:37:08.580+0800","msg":"start to call StartGameSessionByGrpc t
o game server","gameServerSessionId":"qcs::gse:local::gameserversession/fleet-1235/gssess-c648654
a-293b-4f1f-b71f-****6a09bffe"}
{"level":"info","ts":"2020-10-20T09:37:08.597+0800","msg":"onGameSessionActivate received","pid":
"4****","gameServerSessionId":"qcs::gse:local::gameserversession/fleet-1235/gssess-c648654a-293b-
4f1f-b71f-****6a09bffe","requestId":"de1a678dea364db4b487ff84ad****31"}
{"level":"info","ts":"2020-10-20T09:37:08.598+0800","msg":"call StartGameSessionByGrpc to game se
rver success","gameServerSessionId":"qcs::gse:local::gameserversession/fleet-1235/gssess-c648654a
-293b-4f1f-b71f-****6a09bffe"}
```

查询游戏服务器会话

通过 curl 命令，GSE 使用包含游戏服务器会话 ID 的游戏服务器会话对象进行响应。请注意，新游戏服务器会话的状态为“Activating”。游戏服务器会话调用 ActivateGameServerSession 后，状态将更改为“Active”。如果您希望查看更改后的状态，请使用 curl 调用 DescribeGameServerSessions，执行以下命令：

```
curl -d '{"Action":"DescribeGameServerSessions", "FleetId":"fleet-1235"}' http://127.0.0.1:8080/c
api
```

查询结果如下：

```
{"Response":{"GameServerSessions":[{"AvailabilityStatus":"Enable","CreationTime":"2020-10-20T01:3
7:08Z","CreatorId":"","CurrentCustomCount":0,"CurrentPlayerSessionCount":0,"DnsName":"","FleetId"
:"fleet-1235","GameProperties":[],"GameServerSessionData":"","GameServerSessionId":"qcs::gse:loca
l::gameserversession/fleet-1235/gssess-c648654a-293b-4f1f-b71f-2fa56a09bffe","InstanceType":"loca
lhost","IpAddress":"127.0.0.1","MatchmakerData":"","MaxCustomCount":0,"MaximumPlayerSessionCount"
```

```
:5,"Name":"","PlayerSessionCreationPolicy":"ACCEPT_ALL","Port":3237,"Status":"ACTIVE","StatusReason":"","TerminationTime":null,"Weight":0}],"NextToken":"","RequestId":"s1603158295201357000"}}
```

测试游戏服务器和客户端

前提条件

已完成 游戏服务器测试 。

步骤1：加入玩家

执行以下命令加入玩家，其中 GameServerSessionId 为 创建游戏服务器会话 接口返回的 GameServerSessionId。

```
curl -d '{"Action":"JoinGameServerSession", "GameServerSessionId":"qcs::gse:local::gameserversession/fleet-1235/gssess-c648654a-293b-4f1f-b71f-****6a09bffe", "PlayerId":"k****111"}' http://127.0.0.1:8080/capi
```

在 Local 命令提示符窗口中，日志消息将显示游戏服务器已发送 AcceptPlayerSession 请求来验证新玩家连接。

```
{"level":"info","ts":"2020-10-20T10:03:43.096+0800","msg":"API to use: GSE.JoinGameServerSession, with input","req":"GameServerSessionId:¥"qcs::gse:local::gameserversession/fleet-****/gssess-c648654a-293b-4f1f-b71f-****6a09bffe¥" PlayerId:¥"ka****11¥" "}
{"level":"info","ts":"2020-10-20T10:03:43.096+0800","msg":"Creating player session with id: kadin111 for gameServersessionId: qcs::gse:local::gameserversession/fleet-****/gssess-c648654a-293b-4f1f-b71f-****6a09bffe"}
{"level":"info","ts":"2020-10-20T10:03:43.096+0800","msg":"Created player session with PlayerId: kadin111 and PlayerSessionId: psess-56dd6f48-08d4-4a11-9330-****09784977"}
```

步骤2：查询玩家会话

调用 DescribePlayerSessions 查询玩家会话，玩家会话初始状态为“Reserved”：

- 如果1分钟内客户端成功连接游戏服务器，状态将更改为“Active”。
- 如果1分钟后客户端未连接游戏服务器，状态将更改为“TIMEDOUT”。

```
curl -d '{"Action":"DescribePlayerSessions", "GameServerSessionId":"qcs::gse:local::gameserversession/fleet-1235/gssess-c648654a-293b-4f1f-b71f-2fa56a09bffe", "PlayerId":"kadin111"}' http://127.0.0.1:8080/capi
```

查询结果：

版权所有：腾讯云计算（北京）有限责任公司

```
{"Response":{"NextToken":"","PlayerSessions":[{"CreationTime":"2020-10-20T02:03:43Z","DnsName":""
,"FleetId":"fleet-****","GameServerSessionId":"qcs::gse:local::gameserversession/fleet-1235/gsses
s-c648654a-293b-4f1f-b71f-****6a09bffe","IpAddress":"127.*.*.1","PlayerData":"","PlayerId":"ka***
*11","PlayerSessionId":"psess-56dd6f48-08d4-4a11-9330-****09784977","Port":3237,"Status":"TIMEDOU
T","TerminationTime":"1970-01-01T00:00:00Z"}],"RequestId":"s16031596094****2000"}}%
```


步骤3：客户端玩家连接服务端

创建游戏会话和玩家会话之后，建立与游戏会话的直接连接。玩家客户端将通过 `localhost：port` 进行连接。

在 Local 命令提示符窗口中，日志消息将显示游戏服务器已发送 AcceptPlayerSession 请求来验证新玩家连接。如果您使用 curl 调用 DescribePlayerSessions，玩家会话状态应从“Reserved”更改为 “Active”。

步骤4：验证报告发送至 GSE

验证您的游戏服务器正在将游戏和玩家状态发送至 GSE 服务。需要让 GSE 管理玩家需求，并正确报告指标内容，您的游戏服务器必须将各种状态发送至 GSE。验证 Local 正在记录与以下操作相关的事件。您可能还希望使用 curl 跟踪状态的更改。

- **玩家从游戏会话断开连接**
  GSE Local 日志消息应显示游戏服务器调用了 RemovePlayerSession。对 DescribePlayerSessions() 的状态从“Active”更改为“Completed”。您还可以调用 DescribeGameServerSessions 来检查游戏会话的当前玩家数减少了一个。
- **游戏会话结束**
  GSE Local 日志消息将显示游戏服务器调用了 TerminateGameServerSession。对 DescribeGameServerSessions 的 状态从“Active”更改为“Terminated”（或“Terminating”）。
- **服务器进程终止**
  GSE Local 日志消息将显示游戏服务器调用了 ProcessEnding。

测试游戏客户端对 GSE 服务的调用

测试游戏服务器 、 测试游戏服务器和客户端 中涉及到调用游戏会话和玩家会话的 API，都是通过 curl 来进行调用 GSE Local，在游戏服务中调用时，可以通过代码调用，本地调试需通过调用 `http://127.0.0.1:8080/capi` ，验证您的游戏服务是否成功运行，可进行以下 API 调用：

- CreateGameServerSession
- DescribeGameServerSessions
- JoinGameServerSession
- JoinGameServerSessionBatch
- DescribePlayerSessions

在 Local 命令提示符窗口中，只有对 CreateGameServerSession 的调用才会产生日志消息。日志消息显示 GSE Local 提示您的游戏服务器何时启动游戏会话（onStartGameServerSession 回调），并在您的游戏服务器调用它时，获取成功的 ActivateGameServerSession。其他 API 调用可以通过以上 curl 查看状态。

其他说明

使用 GSE Local 时，请记住以下内容：

1. 与 GSE Web 服务不同，Local 不跟踪服务器的运行状况和启动 onProcessTerminate 回调。Local 仅停止记录游戏服务器的运行状况报告。
2. 对于面向腾讯云开发工具包的调用，不验证队组 ID，该 ID 可以满足 FleetId 参数可以设置为任意有效字符串 `(^fleet-¥S+)` 。
3. 使用 Local 创建的游戏会话 ID 具有不同结构。它们包括字符串 local，具体示例如下所示：

```
arn:gse:local::gamesession/fleet-****/gsess-56961f8e-db9c-4173-97e7-****82f0daa6
```

测速工具

最近更新时间：2021-04-12 14:22:16

本文为您介绍各地域的延时测试地址及示例。测速域名，支持 HTTPS 和 UDP 测速。

各地域 HTTPS 和 UDP 测速地址如下表所示

地域	HTTPS 测速地址	UDP 测速地址
北京	https://ap-beijing.speed.tencentgse.com	ap-beijing.speed.tencentgse.com
上海	https://ap-shanghai.speed.tencentgse.com	ap-shanghai.speed.tencentgse.com
香港	https://ap-hongkong.speed.tencentgse.com	ap-hongkong.speed.tencentgse.com
广州	https://ap-guangzhou.speed.tencentgse.com	ap-guangzhou.speed.tencentgse.com
成都	https://ap-chengdu.speed.tencentgse.com	ap-chengdu.speed.tencentgse.com
新加坡	https://ap-singapore.speed.tencentgse.com	ap-singapore.speed.tencentgse.com
孟买	https://ap-mumbai.speed.tencentgse.com	ap-mumbai.speed.tencentgse.com
硅谷	https://na-siliconvalley.speed.tencentgse.com	na-siliconvalley.speed.tencentgse.com
弗吉尼亚	https://na-ashburn.speed.tencentgse.com	na-ashburn.speed.tencentgse.com
法兰克福	https://eu-frankfurt.speed.tencentgse.com	eu-frankfurt.speed.tencentgse.com
首尔	https://ap-seoul.speed.tencentgse.com	ap-seoul.speed.tencentgse.com
东京	https://ap-tokyo.speed.tencentgse.com	ap-tokyo.speed.tencentgse.com

示例说明

以下以广州地域为示例进行说明：

- **HTTPS**

```
ping ap-guangzhou.speed.tencentgse.com
curl https://ap-guangzhou.speed.tencentgse.com/v1/ping
```

- **UDP**

```
域名 + PORT (8888)
ap-guangzhou.speed.tencentgse.com + PORT (8888)
```

游戏进程启动配置

最近更新时间：2021-04-12 14:22:16

1. Linux 环境以 root 或 user_00 用户启动游戏程序

目前 GSE 在 Linux 环境下默认使用 root 用户启动游戏进程，如果用户需要以非 root 用户启动游戏进程，需要进行以下配置：

1. 在游戏生成包的根目录添加文件 gse.yaml，即在游戏服务器舰队实例上解压后的路径为 /local/game/gse.yaml；
2. gse.yaml 文件内容如下，表示在用户组 users 中新增用户 user_00（目前暂不支持配置其他用户及用户组）；

```
User: user_00:users
```

生成包中成功放置 gse.yaml 文件后，GSE 将使用 user_00:users 启动游戏进程，并将 /local/game 路径下所有文件的用户及用户组设置为 user_00:users

示例如下图所示：

```
[root@VM-0-200-centos local]# tree -u -g game
game
├── [user_00  users    ]  gse_5dbe4ee9-e4ca-c7e1-6e8c-3bc856a3ba60.zip
├── [user_00  users    ]  gse.yaml
└── [user_00  users    ]  tarke
    └── [user_00  users    ]  linux
        ├── [user_00  users    ]  linux
        │   └── [user_00  users    ]  linux
        │       ├── [user_00  users    ]  log_2810.txt
        │       ├── [user_00  users    ]  log_2822.txt
        │       ├── [user_00  users    ]  log_2841.txt
        │       ├── [user_00  users    ]  log_2862.txt
        │       ├── [user_00  users    ]  log_2876.txt
        │       └── [user_00  users    ]  perfasset
        └── [user_00  users    ]  user00muli.zip

4 directories, 9 files
[root@VM-0-200-centos local]# ps -ef |grep perfasset
user_00    2810    2754  0 19:43 ?        00:00:00 /local/game/tarke/linux/linux/linux/perfasset
user_00    2822    2754  0 19:43 ?        00:00:00 /local/game/tarke/linux/linux/linux/perfasset
user_00    2841    2754  0 19:43 ?        00:00:00 /local/game/tarke/linux/linux/linux/perfasset
user_00    2862    2754  0 19:43 ?        00:00:00 /local/game/tarke/linux/linux/linux/perfasset
user_00    2876    2754  0 19:43 ?        00:00:00 /local/game/tarke/linux/linux/linux/perfasset
root       4035    3220  0 19:49 pts/0    00:00:00 grep --color=auto perfasset
```

2. Linux环境游戏进程启动前执行 install.sh

在游戏进程启动之前，用户可能需要在 CVM 实例上安装一些软件或配置一些环境变量等，具体操作步骤如下：

1. 用户新建 install.sh 脚本，将游戏进程启动前的操作写入 install.sh 中；
2. 将 install.sh 放置到游戏生成包的根目录下，即在游戏服务器舰队实例上解压后的路径为 /local/game/install.sh。

3. Java语言开发的游戏进程启动配置

Linux 环境下启动 Java 程序的命令如：java -jar XXXX.jar，为确保 Java 语言开发的游戏进程正确启动，需要做以下配置：

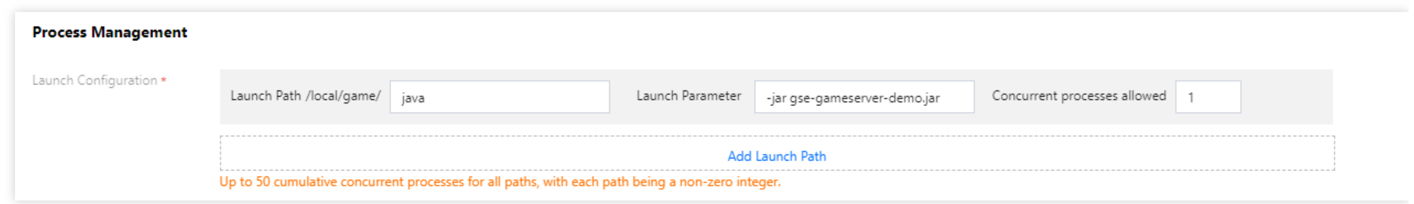
1. 编写 install.sh 脚本

```
#!/bin/bash

# 安装JDK 1.8环境 -y 表示 answer yes for all questions
yum install java-1.8.0-openjdk.x86_64 -y
# 将java命令软链到 /local/game 路径下
ln -s /usr/bin/java /local/game/java
```

2. 将 install.sh 脚本放置到游戏生成包的根目录下，即在游戏服务器舰队实例上解压后的路径为 /local/game/install.sh。

3. 创建游戏服务器舰队时，启动路径填写 /local/game/java，启动参数填写 -jar 用户指定的 jar 包。



4. 游戏进程成功启动后，/local/game 路径内容示意如下：

```
[root@VM-8-206-centos game]# ll
total 77032
-rw-r--r-- 1 root root 41270086 Jan 21 15:35 gse-gameserver-demo.jar
-rw-r--r-- 1 root root 37599574 Jan 21 23:31 gse_08063e02-f884-e503-e88c-c483673ac879.zip
-rwxr-xr-x 1 root root       70 Jan 21 15:35 install.sh
lrwxrwxrwx 1 root root       13 Jan 21 15:35 java -> /usr/bin/java
drwxr-xr-x 2 root root     4096 Jan 25 00:00 logs
```

版权所有：腾讯云计算（北京）有限责任公司