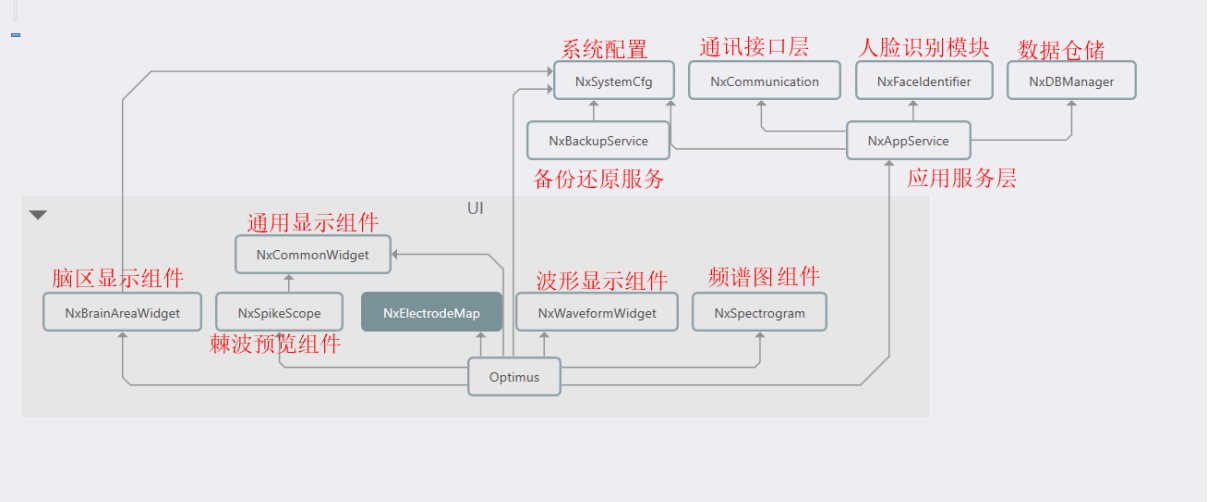
Optimus

前端



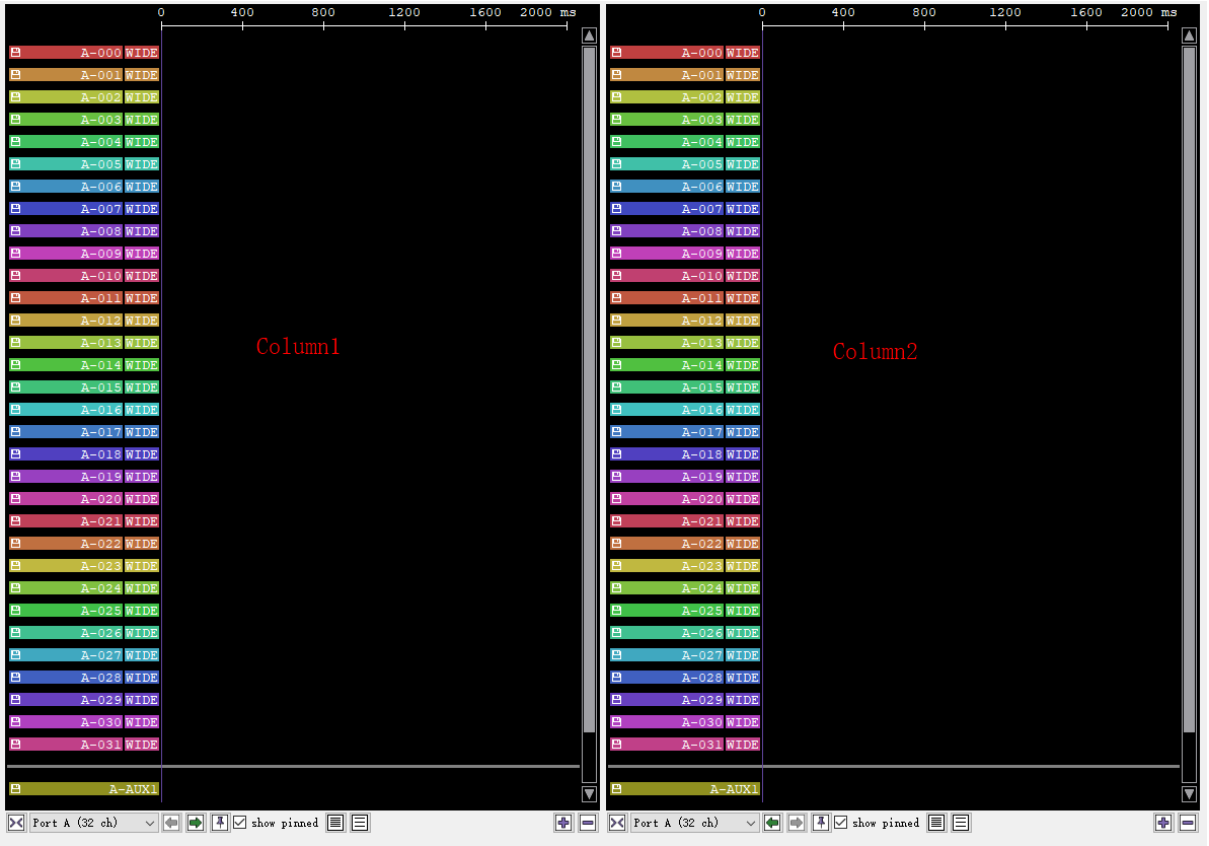
Optimus前端工程中主要模块如上图所示，应用主要包含UI层、应用服务层、业务层，其中UI层用于展示数据以及和用户交互；应用层用于整合业务层所有功能，暴露接口给UI层；业务层主要为各个具体业务模块，包含通讯、数据仓储、人脸识别等基础功能。

在UI层中，脑区显示组件用于脑区、电极、探头植入情况预览；SpikeScope用于预览单个通道中Spike情况；WaveformWidget为波形显示组件；Spectrogram为频谱图

显示组件，这几个显示组件最多只依赖系统配置，不依赖其他任何业务模块，并提供单一的接口给上层模块调用。

业务层中，通讯接口模块负责与Server端进行交互，该模块中封装了GRPC接口的调用，所有具体功能实现都放在RecorderClientImpl中进行。考虑到在某些特定业务场景下，可能会不再使用GRPC接口进行前后端交互(比如通过动态库方式去调用)，此时只需要重新定义新的Client实现即可，将新的接口给RecorderClient调用。

WaveformWidget功能实现

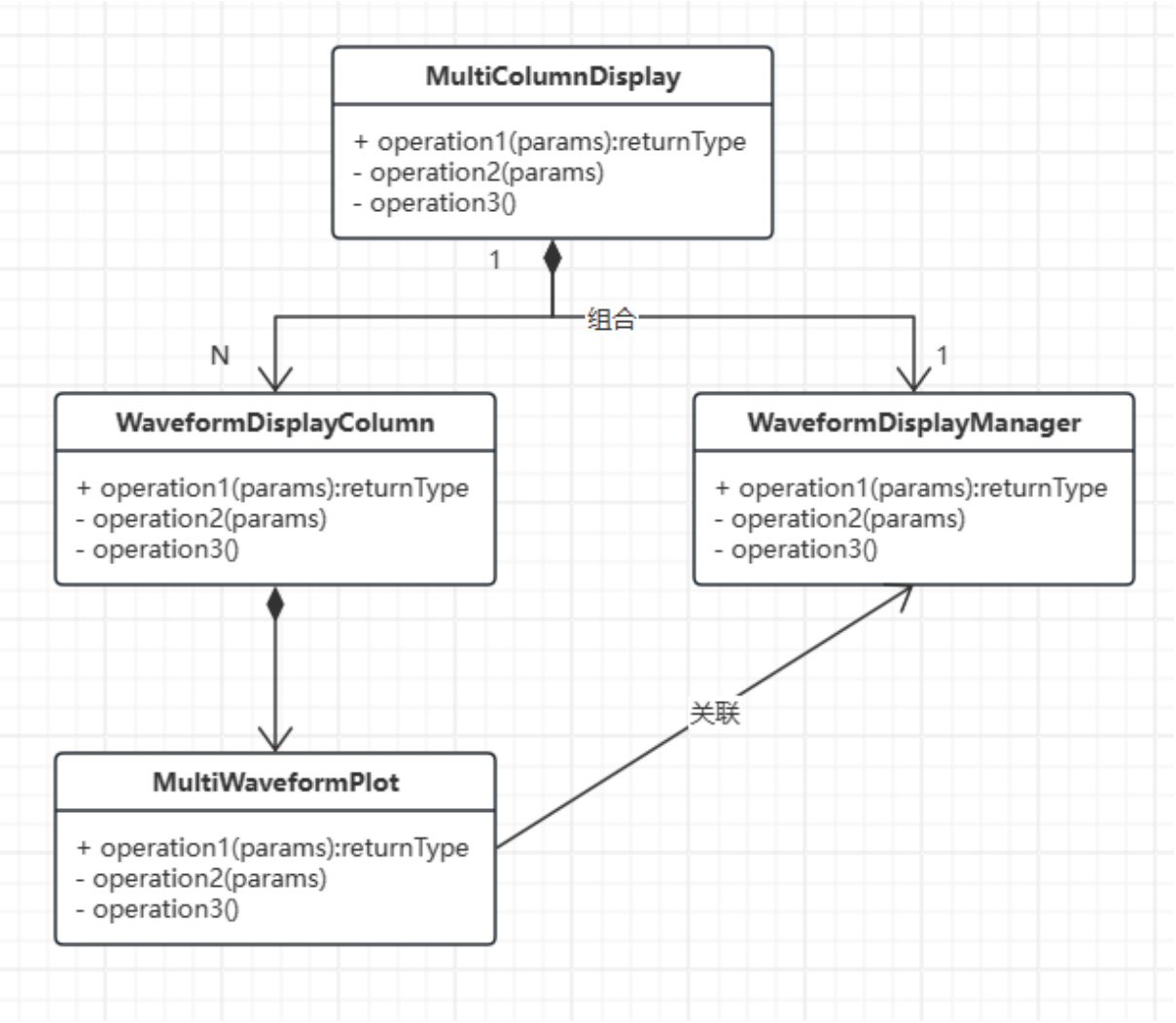


波形显示组件主要显示内容如上图所示，其可以显示多列的波形显示区域，每个显示区域中包含label、波形、标尺等功能，波形显示组件中核心类包括:

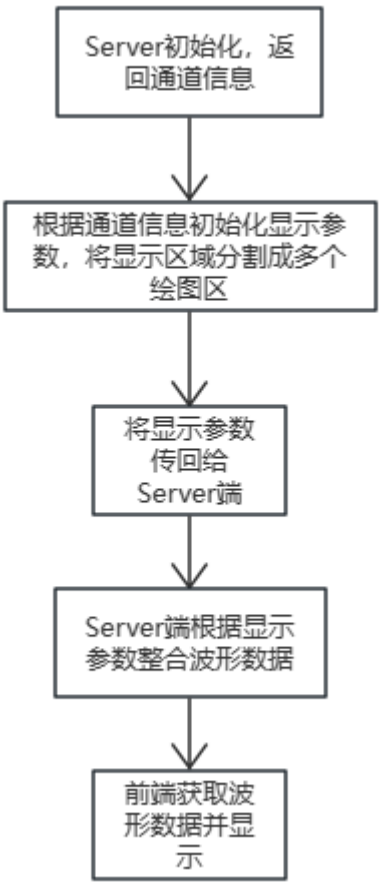
- MultiWaveformPlot 画图功能类，绘制显示区域内所有元素，包括波形、标签、标尺等等
- WaveformDisplayManager 用于保存波形显示波形数据和波形颜色，以及提供波形绘制功能
- WaveformDisplayColumn 波形显示列，其中包含波形显示中所有的元素
- MultiColumnDisplay 对外暴露的接口类，包含多个WaveformDisplayColumn

- DisplayStatus 记录波形绘制起始位置、波形分段数量、时间戳、背景颜色、绘制方式等显示参数
- SignalSources 记录信号参数，包括通道数、通道名称、通道采样率、分组信息等

关键类图如下图：



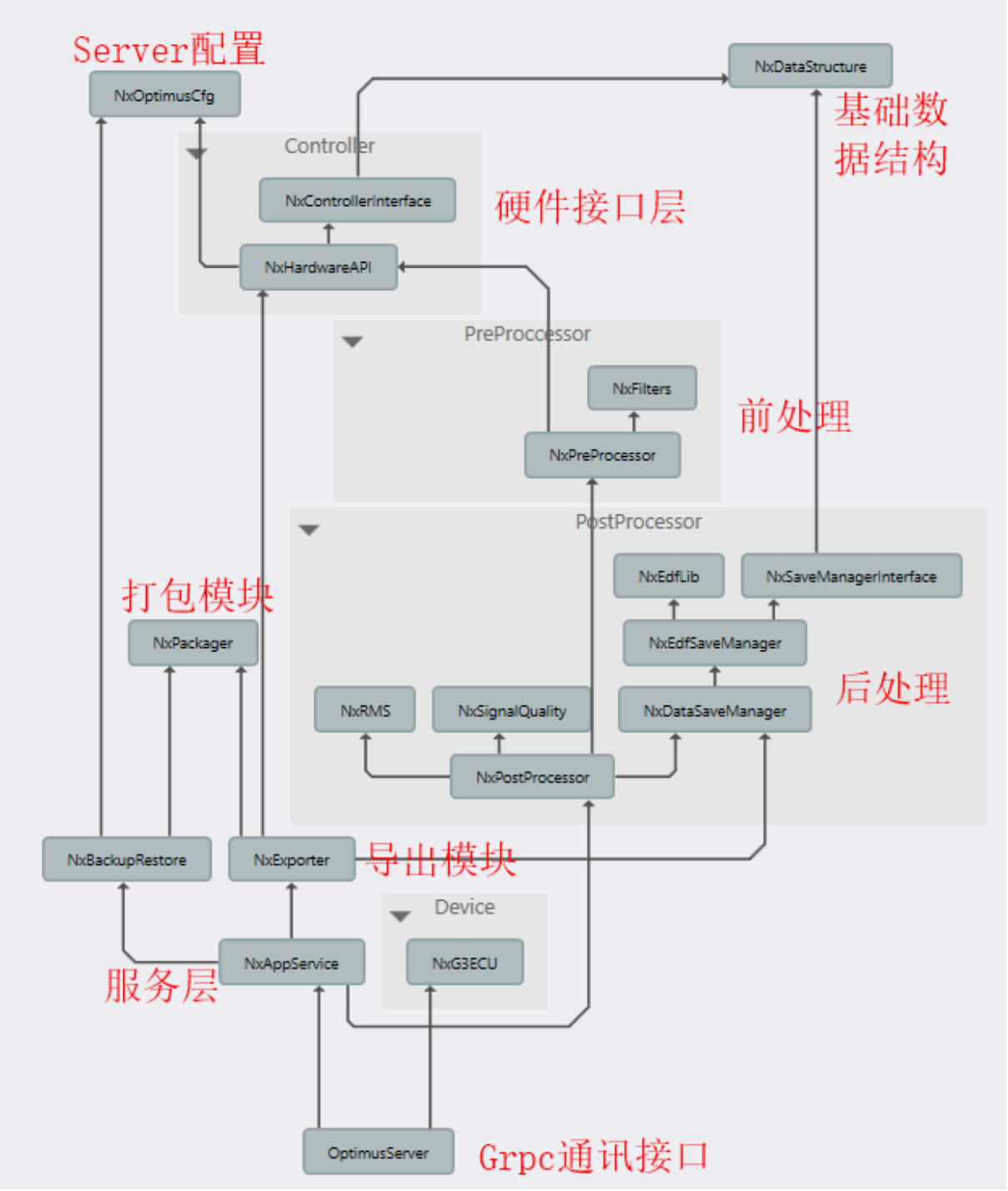
绘图过程：



在Server端初始化后，返回通道列表信息，包括通道名称、采样率等。前端获取到通道列表后初始化显示的通道列表、时间线位置、显示窗口的像素数、每次刷新的采样点和像素个数等显示参数，并将显示区域划分为多个绘制区域，用于后面刷新时每次只刷新一个显示区域；然后前端将显示参数以及显示的通道列表更新到Server端，在开始采集以后，Server端根据显示参数会去获取波形数据，然后将所有需要绘制的波形返回给前端，前端获取到波形数据将波形绘制到波形显示区域当中。

Server端

Server端模块依赖关系如下图所示，其主要包括硬件接口层、前处理层、后处理层、应用服务层、其他基础模块等。



硬件接口层：

硬件接口层用于处理与硬件设备的底层交互，其中ControllerInterface抽象了操作控制器的功能接口，所有控制器都应实现该接口供HardwareAPI模块调用。特殊的，回放的Controller并不是直接操作硬件，而是读取本地的文件实现的，其依赖DataFileReader。

在HardwareAPI中，提供单独的线程用于从硬件设备或者文件系统获取脑电数据，然后将数据脑电数据内容缓存只UsbFIFO当中，在当前实现中，每个Port接口都拥有单

独的UsbFifo用于缓存数据。

前处理层：

前处理层主要用于对从硬件采集上来的数据进行基础的滤波操作，同样在该模块中包含单独WaveFormProcessorThread进行单独的线程处理，该线程从UsbFIFO中获取数据，然后将处理后的数据缓存至WaveformFIFO当中，与UsbFIFO类似，每个Port接口都拥有单独的WaveformFIFO用于缓存滤波后的数据和原始数据。

后处理层：

后处理层用于对前处理后的数据进行进一步加工，处理成业务所需的数据，其中主要包括：信号质量计算、波形显示数据处理、数据保存管理、均方根计算等等，其中每一个功能均在一个单独的线程中进行处理，所有功能均继承自WorkerThreadBase类，该模块还会根据后期业务需求进行不断扩展。

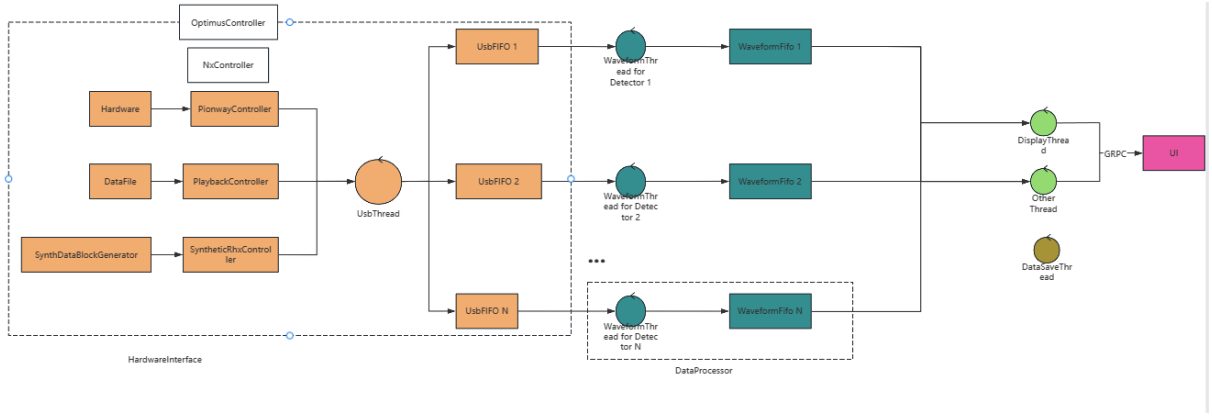
应用服务层:

应用服务层用于整合前面所有模块的业务，用于提供供上层调用的应用接口，通过应用服务层可以实现所有的业务功能。必要时，前端代码可直接通过Communication中增加实现类来调用该模块实现相关功能。

GRPC接口层:

grpc接口层为OptimusServer，用于封装应用服务层为grpc接口，供前端界面进行远程调用。

Server端数据流图如下图所示：



Grpc编译

简介

gRPC 是一个高性能、开源的远程过程调用（RPC）框架，由 Google 开发并开源。它基于 HTTP/2 协议，使用 Protocol Buffers（protobuf）作为接口定义语言

（IDL），支持多种编程语言。
gRPC 的主要特点包括高性能、跨平台、强类型、可插拔和安全性。它基于 HTTP/2，使用二进制协议传输数据，并采用多路复用技术，提高了性能和效率。gRPC 支持多种编程语言和操作系统，如 C、C++、Java、Python、Go，以及 Linux、Windows、macOS 等平台。它使用 Protocol Buffers 作为接口定义语言，可以对消息进行强类型约束，避免了数据类型不匹配的问题。同时，gRPC 提供了多种扩展点，可以添加自定义功能和特性。另外，gRPC 还提供了多种安全机制，如 SSL/TLS 加密、认证和授权，以保证通信的安全性。
gRPC 的应用场景包括微服务架构、分布式系统和移动应用。在微服务架构中，gRPC 支持多种服务发现和负载均衡机制，方便实现服务间的通信。对于分布式系统，gRPC 的高性能和可扩展性使得它非常适合用于大规模数据处理和分布式计算。而在移动应用中，gRPC 的轻量级和跨平台特性使得它适合于构建服务端和客户端通信。
总之，gRPC 是一个高性能、跨平台、强类型的远程过程调用框架，可以帮助开发者快速构建分布式系统和微服务架构。

gRPC的基本使用包括以下几个步骤：

1. 定义服务接口：使用Protocol Buffers（protobuf）定义服务接口的消息格式和方法。通过编写.proto文件来定义消息和服务，并使用protobuf编译器生成对应的代码。
2. 实现服务端：根据定义的.proto文件，编写服务端代码来实现服务接口中定义的方法。可以使用任何支持的编程语言来实现服务端。
3. 实现客户端：根据生成的代码，编写客户端代码来调用服务端提供的方法。客户端可以使用与服务端相同的编程语言，也可以使用其他支持的编程语言。
4. 编译和构建：使用相应的构建工具来编译和构建服务端和客户端的代码。根据编程语言和平台的不同，可以使用不同的构建工具。
5. 启动服务端：在服务端主机上启动服务端程序，监听指定的端口，等待客户端发起请求。
6. 发起请求：在客户端代码中，创建一个 gRPC 客户端对象，并使用该对象调用服务端提供的方法。
7. 处理请求和响应：服务端接收到客户端发起的请求后，根据请求的方法和参数进行相应的处理，并返回一个对应的响应。
8. 客户端接收响应：客户端接收到服务端的响应后，可以进行相应的处理，得到所需的结果。

以上是gRPC的基本使用步骤，希望能对你有所帮助。如果你有更多关于gRPC的问题，请继续提问。

- 服务端代码实现：

```
#include <iostream>
#include <memory>
#include <string>
#include <grpcpp/grpcpp.h>
#include "helloworld.grpc.pb.h"

using grpc::Server;
using grpc::ServerBuilder;
using grpc::ServerContext;
using grpc::Status;
using helloworld::Greeter;
using helloworld::HelloRequest;
using helloworld::HelloReply;

// Logic and data behind the server's behavior.
class GreeterServiceImpl final : public Greeter::Service {
  Status SayHello(ServerContext* context, const HelloRequest*
                  HelloReply* reply) override {
    std::string prefix("Hello ");
    reply->set_message(prefix + request->name());
    return Status::OK;
  }
};

void RunServer() {
  std::string server_address("0.0.0.0:50051");
  GreeterServiceImpl service;

  ServerBuilder builder;
  // Listen on the given address without any authentication m
  builder.AddListeningPort(server_address, grpc::InsecureServ
  // Register "service" as the instance through which we'll c
```

```
  // clients. In this case it corresponds to an *synchronous*
  builder.RegisterService(&service);
  // Finally assemble the server.
  std::unique_ptr<Server> server(builder.BuildAndStart());
  std::cout << "Server listening on " << server_address << st

  server->Wait();
}

int main(int argc, char** argv) {
  RunServer();

  return 0;
}
```

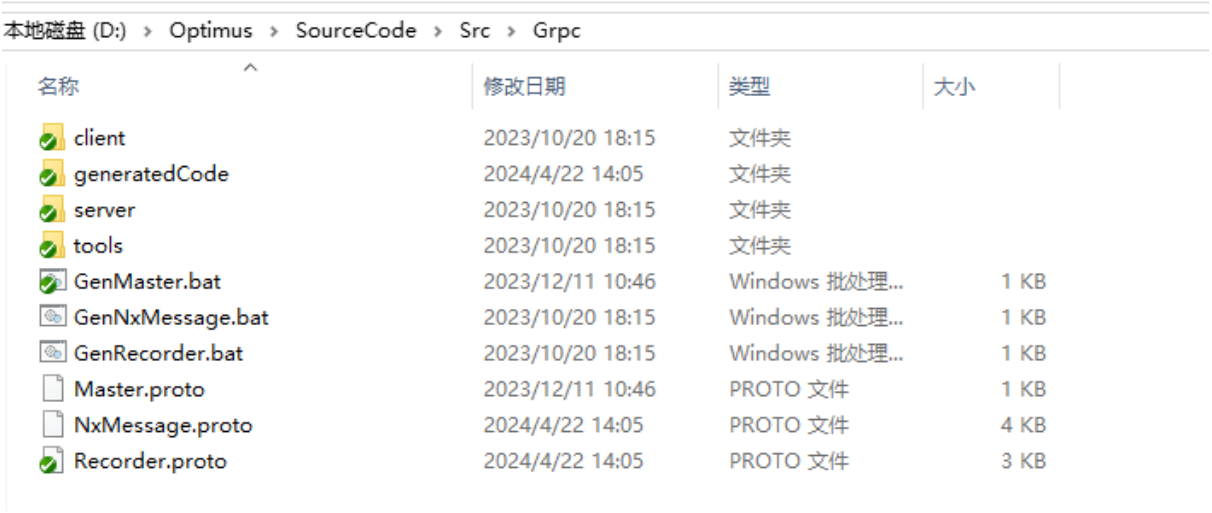
- 客户端代码实现实例

```
#include <iostream>
#include <memory>
#include <string>
#include <grpcpp/grpcpp.h>
#include "helloworld.grpc.pb.h"

using grpc::Channel;
using grpc::ClientContext;
using grpc::Status;
using helloworld::Greeter;
using helloworld::HelloRequest;
using helloworld::HelloReply;

class GreeterClient {
public:
  GreeterClient(std::shared_ptr<Channel> channel) : stub_(Gre

  // Assembles the client's payload, sends it and presents th
  // from the server.
  std::string SayHello(const std::string& user) {
    // Data we are sending to the server.
```

```
    HelloRequest request;
    request.set_name(user);

    // Container for the data we expect from the server.
    HelloReply reply;

    // Context for the client. It could be used to convey ext
    // the server and/or tweak certain RPC behaviors.
    ClientContext context;

    // The actual RPC.
    Status status = stub_->SayHello(&context, request, &reply

    // Act upon its status.
    if (status.ok()) {
      return reply.message();
    } else {
      std::cout << "RPC failed" << std::endl;
      return "RPC failed";
    }
  }

private:
  std::unique_ptr<Greeter::Stub> stub_;
};

int main(int argc, char** argv) {
  // Instantiate the client. It requires a channel, out here
  // created with the server address and set as insecure (no
  GreeterClient greeter(grpc::CreateChannel("localhost:50051"
  std::string user("world");
  std::string reply = greeter.SayHello(user);
  std::cout << "Greeter received: " << reply << std::endl;

  return 0;
}
```

Quick start | C++ | gRPC

实现



GRPC生成工具及对应协议文件放在SourceCode\Src\Grpc文件夹下，其中proto文件为协议文件，其中Recorder.proto文件为Server端Rpc协议接口，Master.proto为Master端服务接口，NxMessage.proto为公共的消息类型定义接口，所有协议更新后，只需要执行对应批处理生成对应的grpc 代码文件即可，代码文件默认生成到generatedCode文件夹下，生成成功后，需要将代码文件拷贝到前端的NxCommunication工程下和Server端的OptimusServer工程文件夹下。