1. 请根据具体的例子说明RPC的主要过程。
    1. 定义服务接口：这里定义了一个`Hello`服务，其中有`SayHello`的rpc方法定义

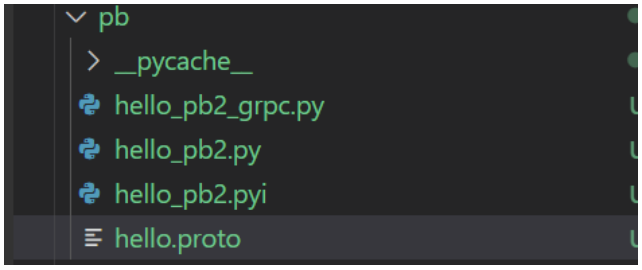
       定义消息类型`HelloRequest`和`HelloResponse`：使RPC框架能将数据序列化为二进制格式，从而通过网络传输，并在接收端反序列化回原始数据

```
syntax="proto3";
service Hello{
 rpc SayHello(HelloRequest)
  returns (HelloResponse){}
}

message HelloRequest{
    string name=1;
}

message HelloResponse{
  string response = 1;
}
```

    2. 生成服务器端代码和客户端存根代码：



    3. 在服务器端定义方法`SayHello`，并且开启服务器

```
class HelloServer(HelloServicer):
    def SayHello(self, request, context):
        name=request.name
        return HelloResponse(response="hello: {}".format(name))


def serve():
    port="40000" #监视端口为40000
    server=grpc.server(futures.ThreadPoolExecutor(max_workers=10))#实例化易感染具有10个工作线程的线程池执行器
    hello_pb2_grpc.add_HelloServicer_to_server(HelloServer(),server)#将Helloerver实例添加到gRPC服务器，为了处理定义的RPC调用
    server.add_insecure_port('[::]:' + port)  # 为服务器添加一个不安全的监听端口。[::] 表示服务器将监听所有可用的网络接口
    server.start() #启动服务器，使其开始监听请求
    print("Server started, listening on " + port)
    server.wait_for_termination() #使服务器保持运行状态，直到它被明确地关闭或终止
```

    4. 在客户端构造消息`dxq`，并且调用客户端存根代码`sayHello`：

```
#初始化HelloStub实例
stub=HelloStub(channel)
#构造一个消息
request=HelloRequest(name="dxq")
#调用存根的sayHello,会阻塞知道服务器返回响应
response=stub.SayHello(request)
```

       存根定义如下：客户端构造`request`并且调用存根，存根将请求消息序列化。
       存根构造消息，存根调用本地OS的网络功能，将消息发送到远程服务器

```
class HelloStub(object):

    def __init__(self, channel):
        """Constructor.
```

```
        Args:
            channel: A grpc.Channel.
        """
        self.SayHello = channel.unary_unary(
                '/Hello/SayHello',#服务名称和方法名称

 request_serializer=pb_dot_hello__pb2.HelloRequest.SerializeToString,#将请求消息对象序列化为字节字符串

 response_deserializer=pb_dot_hello__pb2.HelloResponse.FromString,#将接收到的字节字符串反序列化为响应消息对象
                _registered_method=True)
```

5. 消息通过网络传输到远程服务器，远程服务OS接收消息，并将其传递给服务器存根：服务器存根解析网络信息，转化为对服务器实际过程的调用
    - 自动生成的服务器存根如下：

```
class HelloServicer(object):
    """Missing associated documentation comment in .proto file."""

    def SayHello(self, request, context):
        """Missing associated documentation comment in .proto file."""
        context.set_code(grpc.StatusCode.UNIMPLEMENTED)
        context.set_details('Method not implemented!')
        raise NotImplementedError('Method not implemented!')

def add_HelloServicer_to_server(servicer, server):
    rpc_method_handlers = {
            'SayHello': grpc.unary_unary_rpc_method_handler(
                    servicer.SayHello,

 request_deserializer=pb_dot_hello__pb2.HelloRequest.FromString,

 response_serializer=pb_dot_hello__pb2.HelloResponse.SerializeToString,
            ),
    }
```

    - 服务器端代码覆盖定义 `SayHello` 方法:

```
class HelloServer(HelloServicer):
    def SayHello(self, request, context):
        name=request.name
        return HelloResponse(response="hello: {}".format(name))
```

6. 服务器完成调用方法过程后，将结果返回给存根，存根构建消息，调用本地服务器的网络服务
7. 本地OS通过网络传输消息给客户端OS
8. 客户端OS将接收到的信息给客户端存根
9. 客户端存根解包和反序列消息

2. 描述一下客户端和服务器端之间使用套接字的有连接通信是如何进行的?

- 服务器创建监听套接字，绑定服务器地址，启动监听

```
int passiveTCP(int qlen) {
    int server_socket;
    struct sockaddr_in server_addr;
    // 创建 TCP 套接字
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    // 初始化服务器地址结构
    server_addr.sin_family = AF_INET; //指定IPV4地址簇
    server_addr.sin_addr.s_addr = INADDR_ANY; //监听来自任何网络接口的连接请求
    server_addr.sin_port = htons(9999);//绑定的服务器端口
    // 绑定套接字到服务器地址
    bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr));
    // 启动监听
    listen(server_socket, qlen); // 设置监听队列的最大长度
    return server_socket;
}
```

- 客户端创造套接字，向服务器对应的监听端口用`connect`函数发送连接请求

```
    //创造一个套接字
        SOCKET sockfd;
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == INVALID_SOCKET) {
        fprintf(stderr, "Error creating socket\n");
        WSACleanup();
        exit(1);
    }

    char *host = "172.26.105.168"; /* host to use if none supplied */
    struct sockaddr_in server;
   //定义发送到的服务器的信息
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = inet_addr(host);//发送给本机就是特殊的addr
    server.sin_port = htons(9999); //不是随机的，是监听到的端口
// 向服务器发送连接请求
if (connect(sockfd, (struct sockaddr *)&server, sizeof(server)) != 0) {
    fprintf(stderr, "Error connecting to server\n");
    closesocket(sockfd);
    WSACleanup();
    exit(1);
}
```

- 服务器用`accepct`函数从在监听端口排队的连接请求中接收一个请求，并且创建一个新的接收数据的套接字

```
while(1){ //外层循环监听链接
    alen=sizeof(struct sockaddr); //accept 函数会接受这个连接
    ssock=accept(msock,(struct sockaddr*)&fsin,&alen);
    if (ssock == INVALID_SOCKET) {
        // errexit("accept failed\n");
        continue; // 或者选择退出循环
    }
```

- 客户端用 `send` 函数通过套接字发送数据：

```
// Send "123"
int i=10;
const char *data1 = "123";
 while(i>0){
 i--;
send(sockfd, data1, strlen(data1), 0);
 sleep(1);
 }
```

- 服务器端通过 `recv` 函数用新的套接字接收数据，并且把数据存储在缓冲区 `buffer` 中

```
while(1){ //内层循环接连接的数据
int bytes_received = recv(ssock, buffer, sizeof(buffer), 0); // 接收数据包
if (bytes_received == SOCKET_ERROR) {
    break;
}
if (bytes_received == 0) {
    break;
}
buffer[bytes_received]='\0';
```

3. 当在基于DHT(分布式Hash表)的系统中解析一个键值时，递归查询的主要缺点是什么？
   - 通信代价高：递归查询需要多次访问网络中的多个节点，消耗大量的资源，增加网络负载
   - 响应时间长：将搜索请求发给邻居节点，邻居节点没有搜索到还会将请求继续传递下去，需要的搜索时间长。且请求也具有生存时间TTL限制，如果在限定时间内没有搜索到键值，则请求失效，解析失败
4. 维护到客户的TCP/IP链接的服务器是状态相关的还是状态无关的？说明理由。

   无状态服务器在处理完请求时，不保存关于客户端的精确信息。

   维护到客户的TCP/IP连接的服务器是传输层保存了客户端状态，由TCP/IP链接的协议特性决定的，而服务器没有保存客户信息，是状态无关的