1，长连接的心跳机制

如果应用是基于 TCP 的，则可以简单的通过 SO_KEEPALIVE 来实现心跳，具体为：TCP 在设置的 KeepAlive 定时器到时向对端发送一个 TCP segment，如果尝试几次后都没有收到 ACK 或 RST 就认为对端已经不存在，则可以将该消息通知给应用程序。对于服务器来说缺陷是 server 主动发送检测包对于性能有些影响。

如果在应用层上自己实现，可以使用以下方案：

Client 启动心跳定时器，每隔一段时间发送一个心跳，同时启动超时计时器。

Server 端在收到心跳后对 client 发送 ACK。

Client 如果在超时计时器超时之前收到 ACK，则说明服务器正常，否则，说明服务器失效。

Server 端启动判定定时器，用于判断 client 端是否存在，有以下两种判断策略：

(1)时间差策略：

Server 在收到一个心跳后，记录该心跳发送端的 receivedTime，判定计时器时间到达时，计算当前时间减去 receivedTime，若该差值大于某个设定值，则判定该 client 失效。

(2)简单标志：

Server 收到一个心跳后，标记该 client 为连接状态为 true。在判定计时器到达时，查看所有标志。true 的没有超时，false 的超时。该方法较为简单但误差较大。

2，消息的不遗漏

Client 发送的消息对应一个唯一的 ID，服务端在收到后会发送包含该 ID 的一个确认消息给 client，如果 client 在一定的时间内没有收到该确认消息，则会选择再次发送，在重试到达一个上限次数后，给用户提示发送失败。收到确认消息则认为发送成功。

3，消息不重复

Client 和 server 端都会对处理过的消息 ID 进行存储，但是没有收到确认的消息不会被标识为处理过。两端对收到消息的 ID 进行检查是否已经被处理过。

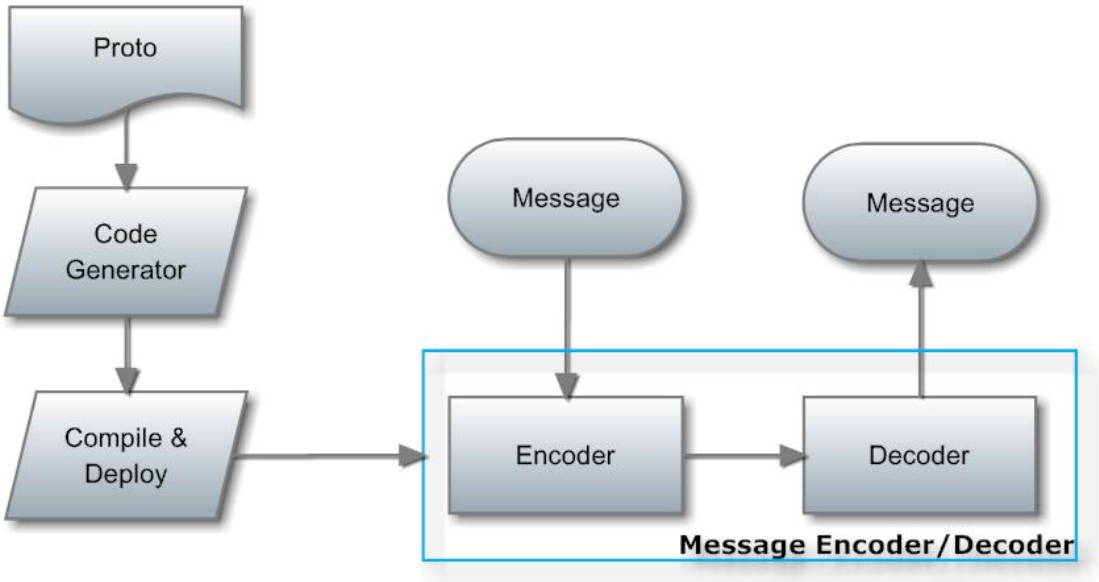
4，消息压缩

在进行消息传输时，pomelo 实现了基于 protobuf 的数据编码协议，与其他的编码协议如 xml，json 相比，protobuf 有着更好的传输效率和压缩比率。在 lordofpomelo 项目中，使用 protobuf 进行数据编码后的消息大小只有基于 Json 的编码的 20%左右。

protobuf 协议介绍：

protobuf 协议是由 google 制定的，主要用于其内部的 rpc 调用和文件编码。原生的 protobuf 包括两部分内容：基于二进制的数据编码协议和基于 proto 元数据的代码生成器。首先，需要根据每条消息来编写对应的 proto 文件，然后使用 google 提供的代码生成器，基于 proto

文件来生成相应的编码器和解码器，然后使用生成的编/解码器来进行编/解码操作，对应的流程如下图：



这种方式的优势是代码静态生成，运行时不需要 proto 文件信息，而且可以根据具体的信息内容对代码进行优化，编解码的时候不需要类型元信息，效率很高。但缺点也十分明显：使用复杂（涉及到代码生成，编译，部署），改动成本高昂（需要重新生成，编译代码，并对代码进行部署），需要生成大量新代码（每个消息都需要一个独立的编码/解码器）。