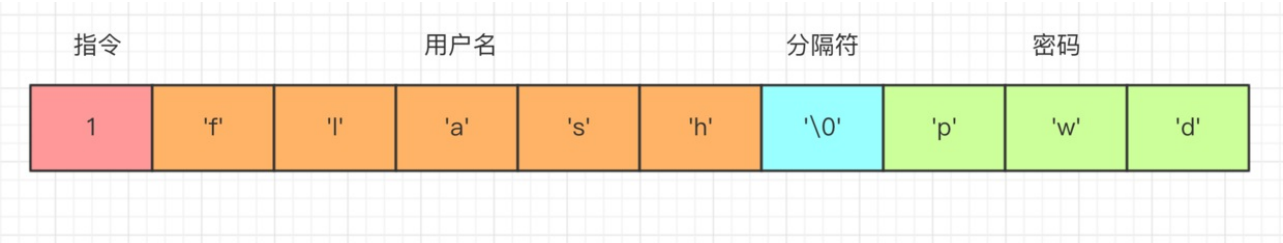
客户端与服务端通信协议编解码

> 在学习了 ByteBuf 的 API 之后，这小节我们来学习如何设计并实现客户端与服务端的通信协议

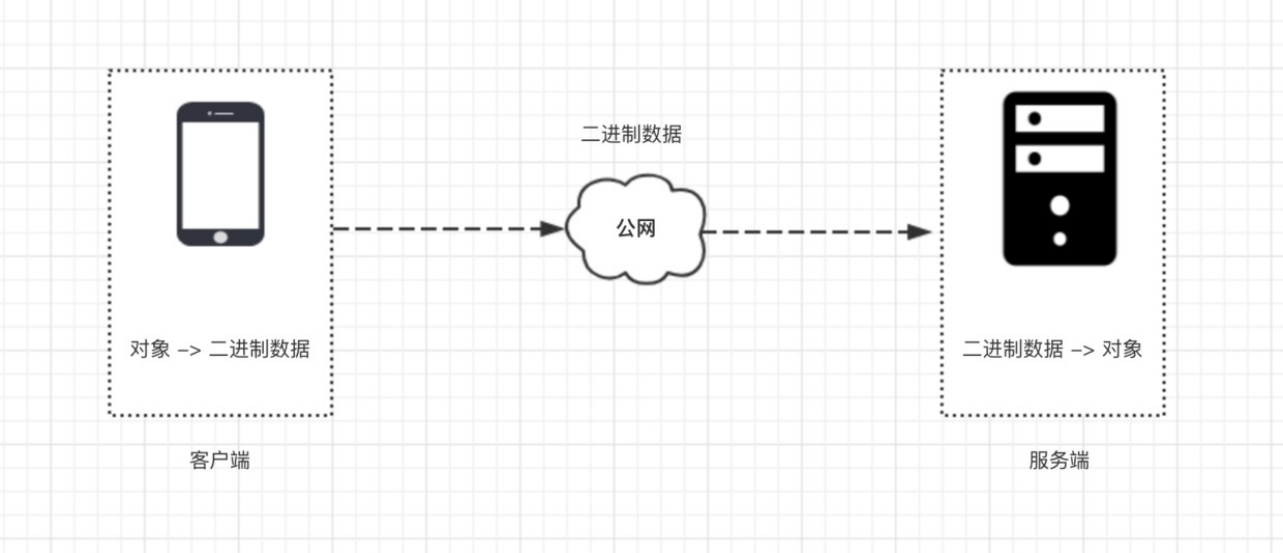
什么是服务端与客户端的通信协议

无论是使用 Netty 还是原始的 Socket 编程，基于 TCP 通信的数据包格式均为二进制，协议指的就是客户端与服务端事先商量好的，每一个二进制数据包中每一段字节分别代表什么含义的规则。如下图的一个简单的登录指令



在这个数据包中，第一个字节为 1 表示这是一个登录指令，接下来是用户名和密码，这两个值以 \0 分割，客户端发送这段二进制数据包到服务端，服务端就能根据这个协议来取出用户名密码，进行登录逻辑。实际的通信协议设计中，我们会考虑更多细节，要比这个稍微复杂一些。

那么，协议设计好之后，客户端与服务端的通信过程又是怎样的呢?

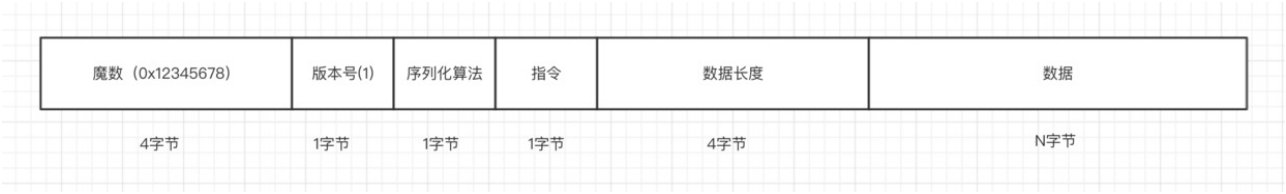


如上图所示，客户端与服务端通信

1. 首先，客户端把一个 Java 对象按照通信协议转换成二进制数据包。
2. 然后通过网络，把这段二进制数据包发送到服务端，数据的传输过程由 TCP/IP 协议负责数据的传输，与我们的应用层无关。
3. 服务端接受到数据之后，按照协议取出二进制数据包中的相应字段，包装成 Java 对象，交给应用逻辑处理。
4. 服务端处理完之后，如果需要吐出响应给客户端，那么按照相同的流程进行。

在本小册子的第一节
(https://juejin.im/book/5b4bc28bf265da0f60130116/sectior
中，我们已经列出了实现一个支持单聊和群聊的 IM 的指令集合，我们设计协议的目的就是为了客户端与服务端能够识别出这些具体的指令。
接下来，我们就来看一下如何设计这个通信协议。

通信协议的设计



1. 首先，第一个字段是魔数，通常情况下为固定的几个字节（我们这边规定为4个字节）。 为什么需要这个字段，而且还是一个固定的数？假设我们在服务器上开了一个端口，比如 80 端口，如果没有这个魔数，任何数据包传递到服务器，服务器都会根据自定义协议来进行处理，包括不符合自定义协议规范的数据包。例如，我们直接通过 http://服务器ip 来访问服务器（默认为 80 端口），
   服务端收到的是一个标准的 HTTP 协议数据包，但是它仍然会按照事先约定好的协议来处理 HTTP 协议，显然，这是会解析出错的。而有了这个魔数之后，服务端首先取出前面四个字节进行比对，能够在第一时间识别出这个数据包并非是遵循自定义协议的，也就是无效数据包，为了安全考虑可以直接关闭连接以节省资源。在 Java 的字节码的二进制文件中，开头的 4 个字节为0xcafebabe 用来标识这是个字节码文件，亦是异曲同工之妙。

2. 接下来一个字节为版本号，通常情况下是预留字段，用于协议升级的时候用到，有点类似 TCP 协议中的一个字段标识是 IPV4 协议还是 IPV6 协议，大多数情况下，这个字段是用不到的，不过为了协议能够支持升级，我们还是先留着。

3. 第三部分，序列化算法表示如何把 Java 对象转换二进制数据以及二进制数据如何转换回 Java 对象，比如 Java 自带的序列化，json，hessian 等序列化方式。

4. 第四部分的字段表示指令，关于指令相关的介绍，我们在前面已经讨论过，服务端或者客户端每收到一种指令都会有相应的处理逻辑，这里，我们用一个字节来表示，最高支持256种指令，对于我们这个 IM 系统来说已经完全足够了。

5. 接下来的字段为数据部分的长度，占四个字节。

6. 最后一个部分为数据内容，每一种指令对应的数据是不一样的，比如登录的时候需要用户名密码，收消息的时候需要用户标识和具体消息内容等等。

通常情况下，这样一套标准的协议能够适配大多数情况下的服务端与客户端的通信场景，接下来我们就来看一下我们如何使用 Netty 来实现这套协议。

通信协议的实现

我们把 Java 对象根据协议封装成二进制数据包的过程成为编码，而把从二进制数据包中解析出 Java 对象的过程成为解码，在学习如何使用 Netty 进行通信协议的编解码之前，我们先来定义一下客户端与服务端通信的 Java 对象。

Java 对象

我们如下定义通信过程中的 Java 对象

```
@Data
public abstract class Packet {
    /**
     * 协议版本
     */
    private Byte version = 1;

    /**
    * 指令
    * /
    public abstract Byte getCommand();
}
```

1. 以上是通信过程中 Java 对象的抽象类，可以看到，我们定义了一个版本号（默认值为 1 ）以及一个获取指令的抽象方法，所有的指令数据包都必须实现这个方法，这样我们就可以知道某种指令的含义。

2. @Data 注解由 lombok (https://www.projectlombok.org/) 提供，它会自动帮我们生产 getter/setter 方法，减少大量重复代码，推荐使用。

接下来，我们拿客户端登录请求为例，定义登录请求数据包

```
public interface Command {

    Byte LOGIN_REQUEST = 1;
}

@Data
public class LoginRequestPacket extends Packet {
    private Integer userId;

    private String username;

    private String password;

    @Override
    public Byte getCommand() {

        return LOGIN_REQUEST;
    }
}
```

登录请求数据包继承自 Packet，然后定义了三个字段，分别是用户 ID，用户名，密码，这里最为重要的就是覆盖了父类的 getCommand() 方法，值为常量 LOGIN_REQUEST。

Java 对象定义完成之后，接下来我们就需要定义一种规则，如何把一个 Java 对象转换成二进制数据，这个规则叫做 Java 对象的序列化。

序列化

我们如下定义序列化接口

```
public interface Serializer {

    /**
     * 序列化算法
     */
    byte getSerializerAlgorithm();

    /**
     * java 对象转换成二进制
     */
    byte[] serialize(Object object);

    /**
     * 二进制转换成 java 对象
     */
    <T> T deserialize(Class<T> clazz, byte[]
bytes);
}
```

序列化接口有三个方法，getSerializerAlgorithm() 获取具体的序列化算法标识，serialize() 将 Java 对象转换成字节数组，deserialize() 将字节数组转换成某种类型的 Java 对象，在本小册中，我们使用最简单的 json 序列化方式，使用阿里巴巴的 [fastjson (https://github.com/alibaba/fastjson)](https://github.com/alibaba/fastjson) 作为序列化框架。

```
public interface SerializerAlgorithm {
    /**
     * json 序列化标识
     */
    byte JSON = 1;
}


public class JSONSerializer implements Serializer
{

    @Override
    public byte getSerializerAlgorithm() {

        return SerializerAlgorithm.JSON;
    }

    @Override
    public byte[] serialize(Object object) {

        return JSON.toJSONBytes(object);
    }

    @Override
    public <T> T deserialize(Class<T> clazz,
byte[] bytes) {

        return JSON.parseObject(bytes, clazz);
    }
}
```

然后，我们定义一下序列化算法的类型以及默认序列化算法

```
public interface Serializer {
    /**
     * json 序列化
     */
    byte JSON_SERIALIZER = 1;

    Serializer DEFAULT = new JSONSerializer();

    // ...
}
```

这样，我们就实现了序列化相关的逻辑，如果想要实现其他序列化算法的话，只需要继承一下 Serializer，然后定义一下序列化算法的标识，再覆盖一下两个方法即可。

序列化定义了 Java 对象与二进制数据的互转过程，接下来，我们就来学习一下，如何把这部分的数据编码到通信协议的二进制数据包中去。

编码：封装成二进制的过程

PacketCodeC.java

```
private static final int MAGIC_NUMBER =
0x12345678;

public ByteBuf encode(Packet packet) {
    // 1. 创建 ByteBuf 对象
    ByteBuf byteBuf =
ByteBufAllocator.DEFAULT.ioBuffer();
    // 2. 序列化 Java 对象
    byte[] bytes =
Serializer.DEFAULT.serialize(packet);

    // 3. 实际编码过程
    byteBuf.writeInt(MAGIC_NUMBER);
    byteBuf.writeByte(packet.getVersion());

byteBuf.writeByte(Serializer.DEFAULT.getSerialize
rAlgorithm());
    byteBuf.writeByte(packet.getCommand());
    byteBuf.writeInt(bytes.length);
    byteBuf.writeBytes(bytes);

    return byteBuf;
}
```

编码过程分为三个过程

1. 首先，我们需要创建一个 ByteBuf，这里我们调用 Netty 的 ByteBuf 分配器来创建，ioBuffer() 方法会返回适配 io 读写相关的内存，它会尽可能创建一个直接内存，直接内存可以理解为不受 jvm 堆管理的内存空间，写到 IO 缓冲区的效果更高。
2. 接下来，我们将 Java 对象序列化成二进制数据包。

3. 最后，我们对照本小节开头协议的设计以及上一小节 ByteBuf 的 API，逐个往 ByteBuf 写入字段，即实现了编码过程，到此，编码过程结束。

一端实现了编码之后，Netty 会将此 ByteBuf 写到另外一端，另外一端拿到的也是一个 ByteBuf 对象，基于这个 ByteBuf 对象，就可以反解出在对端创建的 Java 对象，这个过程我们称作为解码，下面，我们就来分析一下这个过程。

解码：解析 Java 对象的过程

PacketCodeC.java

```
public Packet decode(ByteBuf byteBuf) {
    // 跳过 magic number
    byteBuf.skipBytes(4);

    // 跳过版本号
    byteBuf.skipBytes(1);

    // 序列化算法标识
    byte serializeAlgorithm = byteBuf.readByte();

    // 指令
    byte command = byteBuf.readByte();

    // 数据包长度
    int length = byteBuf.readInt();

    byte[] bytes = new byte[length];
    byteBuf.readBytes(bytes);

    Class<? extends Packet> requestType =
getRequestType(command);
    Serializer serializer =
getSerializer(serializeAlgorithm);

    if (requestType != null && serializer !=
null) {
        return
serializer.deserialize(requestType, bytes);
    }

    return null;
}
```

解码的流程如下：

1. 我们假定 decode 方法传递进来的 ByteBuf 已经是合法的（在后面小节我们再来实现校验），即首四个字节是我们前面定义的魔数 0x12345678，这里我们调用 skipBytes 跳过这四个字节。
2. 这里，我们暂时不关注协议版本，通常我们在没有遇到协议升级的时候，这个字段暂时不处理，因为，你会发现，绝大多数情况下，这个字段几乎用不着，但我们仍然需要暂时留着。
3. 接下来，我们调用 ByteBuf 的 API 分别拿到序列化算法标识、指令、数据包的长度。
4. 最后，我们根据拿到的数据包的长度取出数据，通过指令拿到该数据包对应的 Java 对象的类型，根据序列化算法标识拿到序列化对象，将字节数组转换为 Java 对象，至此，解码过程结束。

可以看到，解码过程与编码过程正好是一个相反的过程。

总结

本小节，我们学到了如下几个知识点

1. 通信协议是为了服务端与客户端交互，双方协商出来的满足一定规则的二进制数据格式。
2. 介绍了一种通用的通信协议的设计，包括魔数、版本号、序列化算法标识、指令、数据长度、数据几个字段，该协议能够满足绝大多数的通信场景。
3. Java 对象以及序列化，目的就是实现 Java 对象与二进制数据的互转。
4. 最后，我们依照我们设计的协议以及 ByteBuf 的 API 实现了通信协议，这个过程称为编解码过程。

> 本小节完整代码参考 github (https://github.com/lightningMan/flash-netty/tree/%E5%AE%A2%E6%88%B7%E7%AB%AF%E4%B8%8
> 其中 PacketCodeCTest.java 对编解码过程的测试用例，大家可以下载下来跑一跑。

思考题

1. 除了 json 序列化方式之外，还有那些序列化方式？如何实现？
2. 序列化和编码都是把 Java 对象封装成二进制数据的过程，这两者有什么区别和联系？

欢迎留言讨论。