Impart Music 客户端-服务器 封装模型文档

Listen to music with your friends like you are in a fxxkin impart

- 维护：朱玉林（分支Daxia）

快速上手

数据包

首先为了能理解如何使用服务端和客户端之间的通信，我们必须要了解服务端和客户端之间的通信媒介——**数据包**（Datapack）

```
public class Datapack {
    /**用于进行JSON序列化和反序列化的GSON实例 */
    private static final Gson GSON = new Gson();
    /**数据包的标识信息 */
    public String identifier;
    /**数据包的主体内容 */
    public String content;

    public static Datapack toDatapack(String rawJson){
        return GSON.fromJson(rawJson, Datapack.class);
    }

    public String toJson(){
        return GSON.toJson(this);
    }

    public Datapack(String rawJson){
        Datapack source = GSON.fromJson(rawJson, Datapack.class);
        this.identifier = source.identifier;
        this.content = source.content;
    }

    public Datapack(String identifier, String content){
        this.identifier = identifier;
        this.content = content;
    }

    public Datapack(String identifier, Object object){
        this.content = GSON.toJson(object);
        this.identifier = identifier;
    }

    public static final Datapack HEARTBEAT = new Datapack("HEARTBEAT", null);
}
```

在数据包中你会发现两个特殊的单例内容 `GSON` 和 `HEARTBEAT` ， 这两个单例均为**已经封装的系统自动调用的内容**，无需使用。

- 构造方法 `Datapack(String identifier, Object object)`
- 此构造方法接受一个 `identifier` 字符串和一个对象实例，会自动构造一个标识符为 `identifier` ， 内容主体为 `object` 的 `Json` 序列化的内容，这是**最安全的数据包构造方法**

- *其他构造方法最终功能其实与上述方法一致，不过更加依赖调用者手动序列化。*最不安全的方法是 `Datapack(String rawJson)` 和 `toDatapack(String rawJson)` ，这两个方法均会在服务器内部进行自动调用，**除非你已经完全理解了数据包的解析，不要调用这两个方法**
- 对象实例方法 `Datapack.toJson()` 可以将一个数据包实例序列化为 `JSON` 字符串

启动服务端

- 当你需要启动服务端的时候，新建一个 `Server` 实例

```
new Server(25585);
```

- 该构造方法的原型是 `Server(int port)` ，其中 `port` 是服务端监听的端口，在项目要求中**统一为25585。**
- **注意**， 这个构造方法是一个**阻塞线程**的构造方法，因此如果你想要创建服务端但是还希望执行其他操作，你可以将这个构造方法放入单独的线程中运行。
- 这个构造方法内部会实现 `Server` 对象的单例，当你需要对 `Server` 进行操作的时候，使用 `Server.instance()` 即可获得单例。
- 观察终端的输出，当输出： `Done!` 的时候，说明服务端已经启动成功，并且随时就绪等待客户端的连接了。

启动客户端

- *现在，让我们把视线转向客户端*
- 当你需要建立与服务端的连接的时候，新建一个 `Connection` 实例。

```
new Connection("localhost", 25585);
```

- 该构造方法的原型是 `Connection(String host, int port)` ，其中 `host` 是服务端的IP地址，当你在本地进行测试的时候，我们直接使用 `localhost` 即可。 `port` 是服务端监听的端口号，在项目要求中，我们统一为 `25585` 。

- **注意**，这个方法并不是一个阻塞方法，它会自动分配子线程。
- **注意**，与服务端的 `Server` 方法不同的是， `Connection` 并没有实现单例，也就是说你可以创建与多个服务端的连接。**但是不推荐你这样做。**

- 如果你在启动客户端后输出了多条一样的错误信息并最终输出： `Connection failed after 5 tries` ，说明没有成功建立服务端与客户端的连接，这种情况通常是因为**服务端的进程已经被终止**，或者**客户端的连接IP，端口号与实际的服务器IP，端口号不一致。**

- 如果你发现输出了 `Connected to localhost:25585` ，那么恭喜你，你已经**成功建立了一条待命的客户端与服务端的通信。**

从服务端发送数据包到客户端

- 现在，我们已经成功建立了服务端到客户端的连接，我们如何将服务端发送数据到客户端?
- 首先，根据前面的知识，发送类似于 `Hello` 这样的**非JSON格式的字符串**是非法操作，会导致接收方无法解析。
- 我们可以编写一个测试类 `SendTest.java`

```
public class ServerTest{
    public static void main(String[] args){
        new Server(25585);
    }
}
```

- 我们要在这样的基础上设法发送数据包到客户端，我们可以考虑**使用Scanner**测试。

```
public class ServerTest{
    public static void main(String[] args){
        new Thread(() -> new Server(25585)).start();
        Scanner sc = new Scanner(System.in);
        while(true){
            String line = sc.nextLine();...
        }
    }
}
```

- 注意，我这里在创建服务器的位置添加了一些特殊的语法，让我们来挨个解析。
  ps 如果不想看可以直接跳到后面的**正式发送**部分
- `() -> new Server(25585)`，这是一个**Lambda 表达式**，代表了一个匿名方法，有点类似于C语言当中的**函数指针**，`()` 是方法的参数列表，由于根本没有，所以括号留空。 `->` 是**Lambda的基本符号**，在后面跟上匿名方法的方法体。这个方法的内容就是 `new Server(25585)` 。其实 `() -> new Server(25585)` 与：

```
private void __method(){
    new Server(25585);
}
```

当中的 `__method` 方法名的地位和性质是几乎一致的。

- `new Thread(() -> new Server(25585))` 根据我们先前创建的匿名方法新建一个**线程**，对于这个线程来说，它的*主方法*就是我们建立的匿名方法。
- `new Thread(() -> new Server(25585)).start()` 调用了这个新建线程的 `start()` 方法，这个线程便会开始运行。

正式发送

- 要测试发送功能，我们可以使用 `Server.sendToAll()` 方法，这会给与服务器连接的所有客户端发送一个数据包或者JSON。
- 但是**注意**，就如先前所说，直接发送非JSON字符串是不允许的，所以我们可以考虑将字符串进行 `Datapack` 的封装再发送

```
public class ServerTest{
    public static void main(String[] args){
        new Thread(() -> new Server(25585)).start();
        Scanner sc = new Scanner(System.in);
        while(true){
            String line = sc.nextLine();
            Datapack datapack = new Datapack("SimpleString", line);
            Server.instance().sendToAll(datapack);
        }
    }
}
```

- 当你已经建立客户端与服务端的连接后，在**服务端的窗口打下一个字符串，按下回车后发送**，如果在**客户端界面**收到： `message from server: com.tfs.datapack.Datapack@<.....>` ，恭喜你，你已经掌握了往客户端发送数据包的功能。

- 不过，我们明明输入的是 `Hello` ，为什么收到的是这样一条奇怪的信息？
- 原因是**还没有对数据包进行解析。**由于数据包的发送对象是不固定的，所以我们并不能假定所有的发送内容都只是 `String` ，我们可能会需要发送 `User` ， `Instruction` 等其他类，所以这个时候 `SimpleString` 标识符就起作用了，当客户端收到一个数据包，它就可以观察标识符是否是 `SimpleString` ，如果是，就直接取数据包的 `Content` 部分，那就是你需要的内容。

- 例如我们可以把 `ClientTest` 改成

```
import com.tfs.client.Connection;
import com.tfs.datapack.Datapack;
import com.tfs.logger.Logger;

public class ClientTest {
    public static void main(String[] args){
        Connection c = new Connection("localhost", 25585);
        while(true){
            try {
                Thread.sleep(20);
                //为什么这里要有一个间隔呢说实话我也不是非常
                //明白，但是异步的话如果高速运行容易出现一些
                //难以理解的错误，所以还是给出一些充足的等待
                //时间
            } catch (Exception e) {
                // TODO: handle exception
            }
            Datapack pack = c.popReceive();
            if(pack == null){
                continue;
            }
            if(pack.identifier.equals("SimpleString")){
                Logger.logInfo(pack.content);
            }
        }
    }
}
```

- 现在，当你在服务端键入一条信息的时候，例如 `Hello world!` ，客户端输出：
  `[xx:xx:xx][INFO:main] Hello world!` ，恭喜你，你已经掌握了基本的数据包解析。
- 之后，如果发送一些复杂的内容，你可能需要使用 `GSON` 对 `content` 属性进行 `JSON` 解析，获得对象实例等信息。

类详解

Server类

`Server(int port)`

- 摘要：创建一个服务器实例，监听端口为 `port`
- 重要性质：阻塞线程，单例
- 使用： `new Server()`

`boolean isRunning()`

- 摘要：获得服务器的运行状态
- 返回： `true` - 如果服务器正在运行， `false` - 如果服务器已经关闭
- 使用： `Server.instance().isRunning()`

`void kill()`

- 摘要：中断服务器的运行
- 使用： `Server.instance().kill()`

`static Server instance()`

- 摘要：返回服务器的唯一实例
- 使用： `Server.instance()`

`void sendToAll(String message)`

- 摘要：向所有客户端发送数据包（JSON）
- 使用： `Server.instance().sendToAll("...")`
- 不安全：必须发送符合规范的 `Datapack` JSON

`void sendToAll(Datapack datapack)`

- 摘要：向所有客户端发送数据包
- 使用： `Server.instance().sendToAll(Datapack)`

`receivedDatapacks`

- 摘要： `Queue<Datapack>` ，代表了从客户端收取的所有未处理数据包的集合
- 使用： `Server.instance().receivedDatapacks`

`connectedClients`

- 摘要： `List<ClientHandler>` ，代表了与服务器连接的所有客户端的处理器的集合
- 使用： `Server.instance().connectedClients`

`static int tickIntervalMilliseconds`

- 摘要：代表了服务器每两次处理时间 `tick` 之间的间隔（以毫秒为单位）

ClientHandler类

- 类描述：每次服务器接受到一个新的客户端连接的时候会自动创建此类实例

`ClientHandler(Socket clientSocket)`

- 摘要：对连接来自于 `clientSocket` 的客户端进行服务
- 参数： `clientSocket` - 与客户端的连接

`void run`

- 摘要：对客户端进行服务的主方法
- 重要性质：阻塞性，自动调用性
- 注意：不要在外部直接调用 `run()` ，服务器会自动调用

`void onTick()`

- 摘要：对服务端中心一次 `tick` 的响应
- 重要性质：不可阻塞性，自动调用性
- 注意：不要再外部直接调用 `onTick()` ，服务器会自动调用

`boolean isConnected()`

- 摘要：获取与客户端是否仍然连接
- 返回： `true` - 仍然连接。 `false` - 已经断开

`void killConnection()`

- 摘要：强行断开和客户端的连接

`void sendMessage(String message)`

- 摘要：将某个数据包按JSON格式传入发送到客户端
- 重要性质：非即时性，传入的数据会等待并自动发送
- 不安全：必须传入符合 `Datapack` 格式的JSON字符串

`void sendMessage(Datapack datapack)`

- 摘要：将某个数据包直接发送到客户端
- 重要性质：非即时性，传入的数据会等待并自动发送

`static final int HEART_BEAT_INTERVAL_MILLISECONDS`

- 摘要：与客户端每两次进行验证信息交换的时间间隔，单位为毫秒

`NO_RESPONSE_TIMEOUT_TRIES`

- 摘要：与客户端之间连接没有回应的最大尝试次数

Connection类

- 类描述：代表了客户端方的，来自于服务端的连接。

`static final int HEART_BEAT_INTERVAL_MILLISECONDS`

- 摘要：与服务端每两次进行验证信息交换的时间间隔，单位为毫秒

`static final int NO_RESPONSE_TIMEOUT_TRIES`

- 摘要：与服务端之间连接没有回应的最大尝试次数

`Connection(String host, int port)`

- 摘要：根据IP地址和端口号，建立与服务器的连接
- 参数：
  `host` - 服务器的IP
  `port` - 服务器的端口
- 重要性质：非阻塞性，网络依赖性
- 使用： `new Connection("localhost", 25585)`

`void connect(int maxTries, int timeout)`

- 摘要：尝试对服务器进行连接
- 参数：
  `maxTries` - 最大尝试次数
  `timeout` - 尝试的最长等待时间（ms）
- 重要性质：自动调用性
- 使用： `connection.connect(5, 1000)`

`void sendMessage(String message)`

- 摘要：向服务器发送数据包的JSON字符串
- 参数：
  `message` - 数据包JSON字符串
- 不安全： `message` 必须符合 `Datapack` JSON格式
- 重要性质：非即时性
- 使用： `connection.sendMessage("...")`

`void sendMessage(Datapack datapack)`

- 摘要：向服务器发送数据包
- 参数：
  `datapack` - 发送的数据包
- 重要性质：非及时性
- 使用： `connection.sendMessage("...")`

`Datapack popReceive()`

- 摘要：从服务器获取的所有的未处理的数据包中弹出一个
- 返回：数据包，如果没有，返回 `null`
- 使用： `connection.popReceive()`

`void killConnection()`

- 摘要：强行中断与服务器之间的连接
- 使用： `connection.killConnection()`

`boolean isConnected()`

- 摘要：获取与服务器之间的连接状态
- 返回： `boolean` 。 `true` - 与服务器仍然连接， `false` - 已经断开连接

Datapack类

- 类描述：代表服务器和客户端之间通信的媒介

`Datapack(String rawJson)`

- 摘要：根据JSON文本，自动生成一个数据包
- 不安全：传入的文本必须符合 `Datapack` JSON格式
- 使用： `new Datapack("...")`

`static final Datapack HEARTBEAT`

- 摘要： 与服务器之间的验证数据包
- 重要性质：不可改变性，自动调用性

`static Datapack toDatapack(String rawJson)`

- 摘要：根据JSON文本，自动生成一个数据包
- 不安全：传入的文本必须符合 `Datapack` JSON格式
- 使用： `Datapack.toDatapack("...")`

`String toJson()`

- 摘要：把数据包序列化为JSON
- 使用： `datapack.toJson()`

`Datapack(String identifier, String content)`

- 摘要：根据标识信息和内容主体生成一个数据包
- 不安全： `content` 除字符串外，必须符合对象的 `JSON` 格式
- 使用： `new Datapack("...", "...")`

`Datapack(String identifier, Object object)`

- 摘要：根据标识信息和对象生成一个数据包
- 使用： `new Datapack("...", object)`

`deserializeContent(Class<T> targetClass)`

- 摘要：根据传入的目标类，把本数据包持有的 `content` 序列化为某类的示例
- 警告：一定注意 `content` 和目标类的匹配。
- 使用： `datapack.DeserializeContent(TargetClass.class)`

Logger类

- 类描述：用于记录服务器信息的所有工具

void logError(String message)

- 摘要：向控制台发送一个错误信息

void logError(String format, Object... args)

- 摘要：向控制台发送一个格式化错误信息

void logInfo(String message)

- 摘要：向控制台发送一个普通消息

void logInfo(String format, Object... args)

- 摘要：向控制台发送一个格式化普通消息

void logWarning(String message)

- 摘要：向控制台发送一个警告信息

void logWarning(String format, Object... args)

- 摘要：向控制台发送一个格式化警告信息