ComStar 客户端通信开发指南

1. 如何建立与服务器的连接？

```
try {
    TBCServerConnectionInfo info = new TBCServerConnectionInfo();
    info.userName = "hhh";
    info.password = "hhh";
    info.IP = "fis-ext-dev";
    info.port = "8006";
    info.name = "FIS";
    TBCServerConnection tbcCon = TBCServerConnection
            .createTBCServerConnection(info);
} catch (Exception ex) {
    //创建连接失败
}
```

如果创建成功就正常返回 TBCServerConnection 的一个实例，如果失败则抛出异常。

2. 如何向服务器发送请求？

```
FieldMap para = new FieldMap("Page").putField("Date", "20100908")
        .putField("ID", "89076");
tbcCon.requestRemoteService("CA82", para,new CommonMsgCallback() {
    @Override
    public void onMessage(CommonMsg msg) {
        if (msg.isError()) {
            //错误处理
            String errorMsg = msg.getErrorMsg();
        } else {
            FieldMap fm = msg.getFieldMap();
            //.........
        }
    }
});
```

以上第一个参数是请求的服务名，通常也叫 FID （即功能 ID），第二个参数是请求服务的数据，第三个是回调接口用来接收服务器对这个请求的回应。很明显这是一种异步的方式，执行请求并不会阻塞，等到服务器有回应了自动会调用 onMessage。你可能会问，如果服务器一直不回应会如何？答案是：会超时。如果 30 秒服务器还没回应，通信库会构建一

个超时信息然后调用 onMessage，表示请求已经超时，这个时候即使如果服务器后面消息回来了，那么也不会再理会了，因为一次请求响应的调用已经结束。

3. 如何对一次请求指定超时时间？

```
tbcCon.requestRemoteService("CA82", para,60,new CommonMsgCallback() {
    @Override
    public void onMessage(CommonMsg msg) {
        if (msg.isError()) {
            //错误处理
            String errorMsg = msg.getErrorMsg();
        } else {
            FieldMap fm = msg.getFieldMap();
            //.........
        }
    }
});
```

从上面的代码看出来很简单，请求方法有重载版本支持对这次请求的超时设定，比如上面的 60 就表示这次请求的超时时间是 60 秒，没错单位是秒。提供使用者自己可以定义超时时间还是很有必要的，原因是某些请求你可能预估服务器会执行比较长的时间，而有些预估比较短，所以为了灵活适应各种情况，通信 API 支持你自己设定，当然如果你不设置的话默认也是 60 秒。

4. CommonMsgCallback 和 CommonMsgSwingCallback 的区别是什么?

我们看到上面的例子中接收服务器回应的都用到了 CommonMsgCallback，但是 API 还提供了一个 CommonMsgSwingCallback。看下面的例子：

```
tbcCon.requestRemoteService("CA82",para,
        new CommonMsgSwingCallback(view){
    @Override
    public void onSwingMessage(CommonMsg msg) {
        if(msg.isError()){
            //错误处理
            String errorMsg = msg.getErrorMsg();
            JOptionPane.showMessageDialog(view,errorMsg);
        }else{
            FieldMap fm = msg.getFieldMap();
            //将数据设置到GUI
        }
    }
});
```

看起来与 CommonMsgCallback 差不多，除了名字上多了个 Siwng 以外似乎就是构造 CommonMsgSwingCallback 时多传入了一个 view。没错 CommonMsgSwingCallback 构造时需要传入一个 Component 类型的对象。因为 CommonMsgSwingCallback 比 CommonMsgCallback 多了两个特性，一个是 onSwingMessage 的调用已经保证在 Swing 的 UI 线程上，第二点是如果在 onSwingMessage 发生异常那么会被捕获并且弹出对话框告知某功能模块发生了异常，所以如果要对 GUI 进行操作，那么要使用 CommonMsgSwingCallback。

5. **如何同步的向服务器发送请求？**

```
CommonMsg msg = tbcCon.syncRequestRemoteService("CA82", para);
if (msg.isError()) {
    //错误处理
    String errorMsg = msg.getErrorMsg();
} else {
    FieldMap fm = msg.getFieldMap();
}
```

同步的调用就像普通的函数调用一样，不过同步调用要当心的一点是它可能时间比较长，会阻塞线程，如果是在一个按钮点击引发了这样一个同步调用那么可能会阻塞 UI 线程导致系统看起来像死机不再响应用户的键盘鼠标等事件。所以通常同步调用你需要在后台线

程执行，一般 GUI 开发的话要使用 SwingWorker。这是一个方便的执行长时间任务的 API，在 Java1.6 以后已经属于标准库了，它的使用这里就不展开了。通常一次性的请求响应不会用到同步调用，一般都是像上面例子中那样异步的写法。但是同步仍然有它的用武之地。如果你在循环中不断要向服务器请求数据，如果要严格顺序的执行多个步骤，每个步骤都要和服务器通信，那么异步的写法将是非常难以表达的，而使用同步方法就会非常轻松。另外同步也支持超时设置，这个和异步是一样的。

6. **如果我和服务器之间请求和消息处理要分离该如何做到？**

我们看到上面所有的通信都是典型的请求响应，但是有时候我可能希望发送请求那个地方并不想处理，另外有个地方监听消息去处理。而且也有可能发送之后，服务器有什么变化主动推送消息给我。看一下下面的例子：

```
tbcCon.requestRemoteService("CA82", para);

tbcCon.registerRemoteServiceListener("CA82",
        new CommonMsgCallback() {
    @Override
    public void onMessage(CommonMsg msg) {
        if (msg.isError()) {
            //错误处理
            String errorMsg = msg.getErrorMsg();
        } else {
            FieldMap fm = msg.getFieldMap();
            //.........
        }
    }
});
```

发送请求的时候没有回调接口，但是在另一个地方主动监听了 CA82，所以上面这个请求和处理完全可以分开在不同的地方。

7. **我仅仅想要监听服务器的消息可以吗？**

当然可以，做法其实就是监听而已：

```
tbcCon.registerRemoteServiceListener("CA82",
        new CommonMsgCallback() {
    @Override
    public void onMessage(CommonMsg msg) {
        if (msg.isError()) {
            //错误处理
            String errorMsg = msg.getErrorMsg();
        } else {
            FieldMap fm = msg.getFieldMap();
            //.........
        }
    }
});
```

你不需要请求什么，直接可以监听某个服务，当然这个服务能不能被监听，服务器会不会向你发送什么，这都取决于实际你和服务器方协议上的约定。

我们上面在注册的时候直接构造了匿名类，如果你想要在某个时候移除掉监听，那么这样是不合适的，因为你没有了这个监听器的引用。我们看一下下面的例子：

```
CommonMsgCallback xxxListener = new CommonMsgCallback(){

    @Override
    public void onMessage(CommonMsg msg) {
        //..............
    }
};
tbcCon.registerRemoteServiceListener("CA82",xxxListener);

tbcCon.removeRemoteServiceListener("CA82",xxxListener);
```

上面描述了注册监听和移除监听，在监听模式这种通信方式下什么时候注册监听，什么时候移除监听，都必须使用者自己把握，这和请求响应时不同。

另外要说明一下的是注册的地方也可以使用 CommonMsgSwingCallback ,如果监听到消息后要对 GUI 处理，那么也应该使用 CommonMsgSwingCallback，这是一样的。

8. CommonMsg 的 isError 函数到底能判断哪几种错误？

Comstar 服务器由于某些历史原因，返回的错误信息有些多样化，API 试图屏蔽掉这种差异，但是也不能说支持全部的错误格式，所以这里说明一下支持的错误格式是哪些：

1. 如果服务器返回的消息开头是 F 的，那么通信库立即判断是一个错误。
2. 服务器返回格式<Error>XXXXXX</Error>。
3. 服务器返回格式<Return><Status>Y/N</Status><Message>XX</Message></Return>
4. 服务器返回格式<Return><Result>true/false</Result><Cause>XX</Cause></Return>

其实使用通信 API 的人不应该关心这个细节，通常你确实也不用关心，除非你发现 isError 判断不了的情况，你或需要特殊处理或需要和服务器方沟通然后修改格式。新的协议一般都会满足 isError 的要求。

9. 对于类似于外汇等那种标准键值对的消息监听有没有更为方便的支持？

当然是有的，我们上面讲过消息监听，但是如果你处理过外汇等那种键值对的消息要使用上面的方法还是显得有些麻烦了，所以通信 API 专门为这种数据提供更为便利的支持。请看例子：

```
String [] ids ={"CNY=CFHB/P_MID","CNY=CFHB/P_UPPER"};
OIDListener fxListener = new OIDListener(){
    @Override
    public void onOIDMessage(String key, String value) {

    }
};
tbcCon.registerOIDListener(ids,fxListener);

//不监听了要移除
tbcCon.removeOIDListener(ids,fxListener);
```

上面接收服务器回应的方法和前面的例子有所不同,因为这种消息的格式有它的特殊性和固定性，所以为了方便专门设计了一个监听器接口，上面的例子是说监听了两个 ID，只要这两个 ID 的值发生变化，那么 onOIDMessage 就会收到消息，你可以根据 key 和 value 的状况决定如何处理。当然如果你不监听了也需要移除。这个 API 还支持 List 的重载版本：

```
List<String> ids = new ArrayList<String>();
ids.add("CNY=CFHB/P_MID");
ids.add("CNY=CFHB/P_UPPER");

tbcCon.registerOIDListener(ids, fxListener);
```

这些主要是为了使用者更方便,因为使用者当前最方便的数据结构是不确定的,有可能是数组有可能是 List。

10. 如何监听服务器的状态？

```
tbcCon.addTBCServerConnectionListener(
        new TBCServerConnectionListener() {
    @Override
    public void statusChange(int status) {
        if(status == NetConstants.Connected){
            //已连接上
        }else if(status == NetConstants.ReConnect){
            //重连中
        }
    }
});

tbcCon.addTBCServerConnectionLogoutListener(
        new TBCServerConnectionLogoutListener() {
    @Override
    public void onLogout(String info) {
        //服务器出于某个原因强行退出
    }
});
```

从上面看到有两个可以监听，一个是通常的状态，你可以监听到连接断线重连了或者重连成功又连上了。另一个是监听连接被服务器强行要求退出，这个监听只是告知退出，你无法改变什么，事实上连接通知你的时候已经将自己关闭了，你只是被告知一下退出了，你无法阻拦这件事。

11. 如何关闭与服务器的连接？

```
tbcCon.close();
```

很简单，直接调用 close 方法即可。要注意的是连接一旦被关闭就无法恢复，如果再连接，需要重新创建。

12. 与服务沟通的数据结构除了 FieldMap 之外还支持哪些？

上面的例子中无论是发送的数据还是接收的数据都用了 FieldMap，但是如果数据复杂一点 FieldMap 可能就不够用了，比如我如果要发送或接收一批 FieldMap，也有可能我要接收一个多层次结构的数据等该如何？所以 API 另外提供了 FieldMapSet 和 FieldMapNode。FieldMapSet 可以包含多个 FieldMap，FieldMapNode 是继承 FieldMap，但扩展支持了增加子 FieldMapNode 的能力，这个层次当然可以不断复合下去，所以可以构建一颗复杂的树。如果情况很特殊你甚至可以直接向服务器发送字符串，字符串的格式可以自由定义，但一般不推荐这么做，使用 FieldMap 会清晰和方便很多。下面是一个发送字符串的例子：

```
String strPara = "你的特殊格式";
tbcCon.requestRemoteService("C305",strPara,
        new CommonMsgCallback() {
    @Override
    public void onMessage(CommonMsg msg) {
        if(msg.isError()){
            //错误处理
        }else{
            msg.getMsgString();
        }
    }
});
```

我们看到 CommonMsg 也支持 getMsgString 方法获得服务器发回来的最原始字符串形式。实际上 CommonMsg 支持多种方式：

```
msg.getMsgString();//获得服务器发回来的原始字符串数据
msg.getXmlDocument();//将消息解析为XML文档结构
msg.getFieldMap();//将消息解析为FieldMap
msg.getFieldMapSet();//将消息解析为FieldMapSet
msg.getFieldMapNode();//将消息解析为FieldMapNode
```

实际上 CommonMsg 采用的是延迟解析的方法，只有当你调用 getXXX 时才开始解析成对应的数据结构。因为一开始是无法确定消息可以被解析成什么，客户端和服务端的协议约定上自由度是很大的，最低要求是只要是文本就行，到时候发的时候直接发字符串，收的时候直接 getMsgString 获取字符串然后自己解析。但是非常麻烦，通常我们现在采用 XML 规范，但是解析 XML 仍然是麻烦的，所以客户端还提供了一整套 FieldMap 的数据结构。以大大减轻复杂性。因此，我们通常的通信应该在 FieldMap，FieldMapSet，FieldMapNode 这些层次上。

13. 如何更方便的和服务器进行批量数据处理？

当你成功创建 TBCServerConnection 后其上的 API 可以完成和服务器全部的沟通了，但是假如这样一种情况，服务器的提供的协议中仅仅支持单笔新增或单笔删除，但如果我要发送一批数据我该怎么办？我们可能很快想到如下的写法：

```
FieldMapSet fms = getData();
for(int i = 0; i < fms.getFieldMapCount();i++){
    CommonMsg commonMsg = tbcCon.syncRequestRemoteService
            ("C350",fms.getFieldMap(i));
    //可能的处理
}
```

上面的代码从 getData 拿到了一批数据，现在要发送到服务器，代码采用了同步请求的方式，一笔一笔的发送。这里其实就是上面讲的同步请求的典型用武之地。但是很明显这样

一段代码放到 UI 线程是不现实的，一定会卡住 GUI。在 TBCDataIO 包下提供了方便的 API 处理批量数据，看一个例子：

```
FieldMapSet fms = getData();
CommonDataManager.uploadData(tbcCon,"C350",fms,
        new BatchDataProcessingListener() {
    @Override
    public void onEnd() {
        //全部上传完成会调用
    }

    @Override
    public void onFailure(FieldMap data, String errorMsg) {
        //失败会调用
    }

    @Override
    public void onSuccess(FieldMap data) {
        //成功会调用
    }
});
```

使用者只要实现好三个回调函数便可，整个发送过程由 CommonDataManager 全权负责。如果你想起 CommonMsgSwingCallback 你可能会问 BatchDataProcessingListener 是否也有 Swing 版？的确如此，也有 BatchDataProcessingSwingListener。而且通常你应该用 Swing 版的，因为对客户端来说批量处理过程中意味着要不断更新进度条等，所以在回调方法中一般都要和 GUI 打交道。你可能又会问在整个处理过程中能不能停止上传？这个是可以的。事实上 uploadData 方法会返回一个 Work 对象，Work 上提供了 stop()方法可以停止处理。