**Markdown协同编辑器**

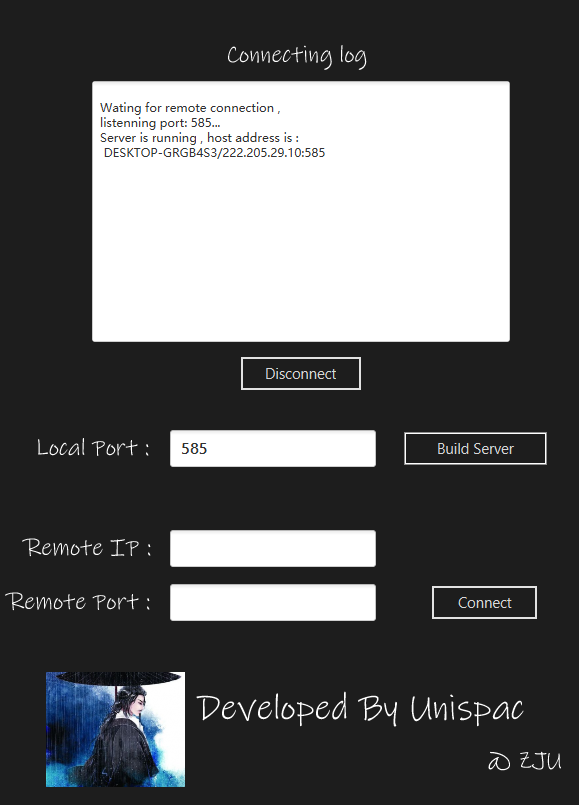
**developed by Unispac @ ZJU**

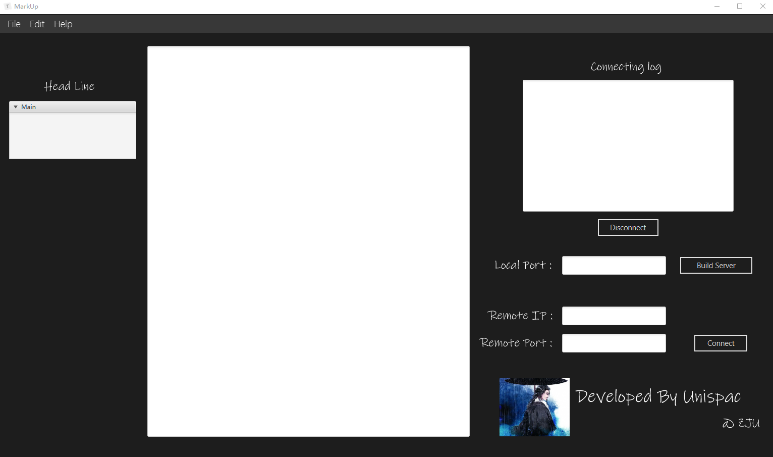
**漆翔宇 3170104557**

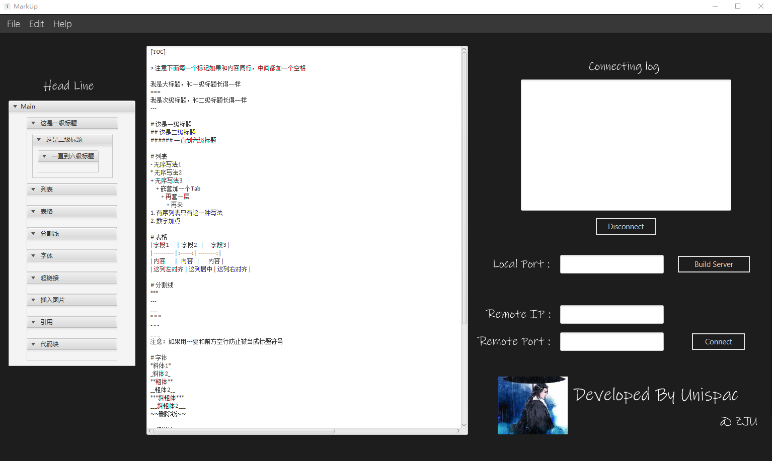
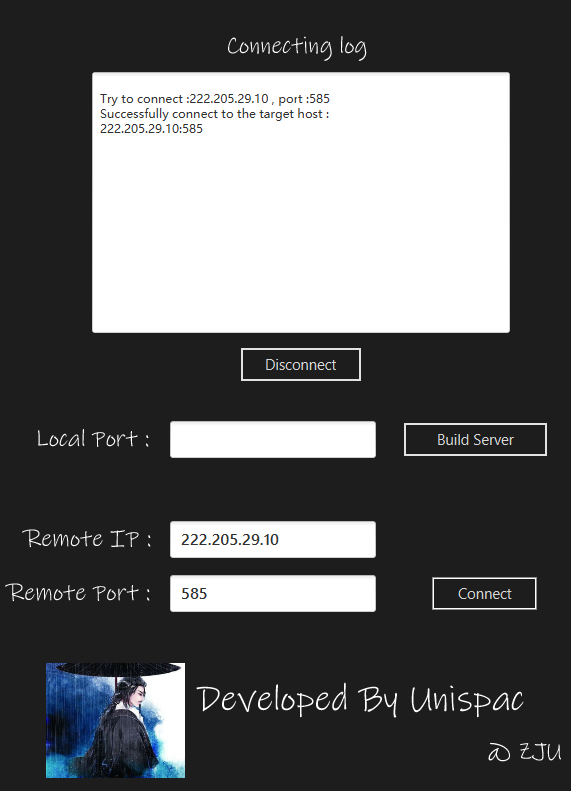
**浙江大学计算机科学与技术学院**

**Tel：17342017090**

**Email：**[**3170104557@zju.edu.cn**](mailto:3170104557@zju.edu.cn)







**Chapter1 项目结构概述**

这一部分，我们主要介绍一下项目的基本构成，各代码模块的承担的功能和一些实现的原理。

**一．JavaFX**

JavaFX平台是java客户端设计演进，使应用开发者易于创建和部署跨平台且表现一致的RIA。JavaFX是由Java技术构建，基于高性能硬件加速的媒体和图形引擎，提供了一套丰富的图形和媒体API。现代版本中，JavaFX已经整合到了Java的标准包库中。可以像使用swing和awt那样直接使用。

本项目是在JavaFX平台上构建的。利用JavaFX Scene Builder，我们可以可视化编辑GUI的前端，完成GUI前端的快速布局。JavaFX Scene Builder会自动为我们生成前端布局的fxml文件，我们在自己的Java程序中载入它，即可和我们的java程序连接起来，生成完整的程序。

使用JavaFX开发桌面GUI应用非常的方便，它使得桌面应用开发变得和网站开发一样。我们可以轻松的实现前后端分离，用类似于HTML的FXML来做前端设计，还能用CSS来做前端样式的编辑。同时在可视化编辑平台的帮助下，前端的设计变得简单直接。大大减省了前端的工作。

**二．md2x**

我们的编辑器从markdown转换成html的模块是借助开源的三方库完成的。我们选择了md2x来帮助我们完成这一转换。库文件可以在<https://github.com/touchface/md2x> 获取，对应于项目的本地lib目录下的md2x-1.0.1.jar。

**三．项目结构**

1. markUp.java

项目根目录下，属于默认包，作为用户调用的接口，它在main方法中调用Main.class中的静态方法启动JavaFX窗口程序。

2. Main.java

属于包Main，其中的Main作为窗口的底层类，继承自JavaFx中的Application类，负责初始化窗口环境。同时提供了一些静态的公有方法，作为controller和net模块之间互通有无的通道。

3. Controller.java

含控制器类，属于Main.Controller。我们设计好的前端.fxml文件在被载入的时候，会和这个类建立联系，实例化一个这个类的对象，并且把前端中定义的各种控件和事件处理器与Java中相应的控件对象和事件处理方法绑定起来。

Controller类中是一些用于接收来自fxml文件定义的控件对象和一系列控件处理方法。它完全承担了整个前端的事件响应。

4. editorNet

Main.editorNet

下属三个子包，分别含三个公有类Connection.class，editorClient.class，editorServer.class。

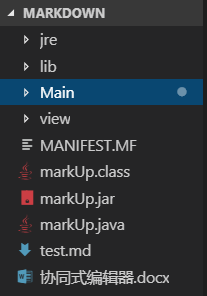
Connection类是editorClient和editorServer的父类。这使得我们可以用同一个接口来管理客户端和服务端的连接。

editorCilent对象被实例化的时候，就会启用一个客户端线程。同样的editorServer对象被实例化时，就启用一个服务端线程。

上述的两个线程分别是通过editorClientThread类和editorServerThread类实现的。

5. view

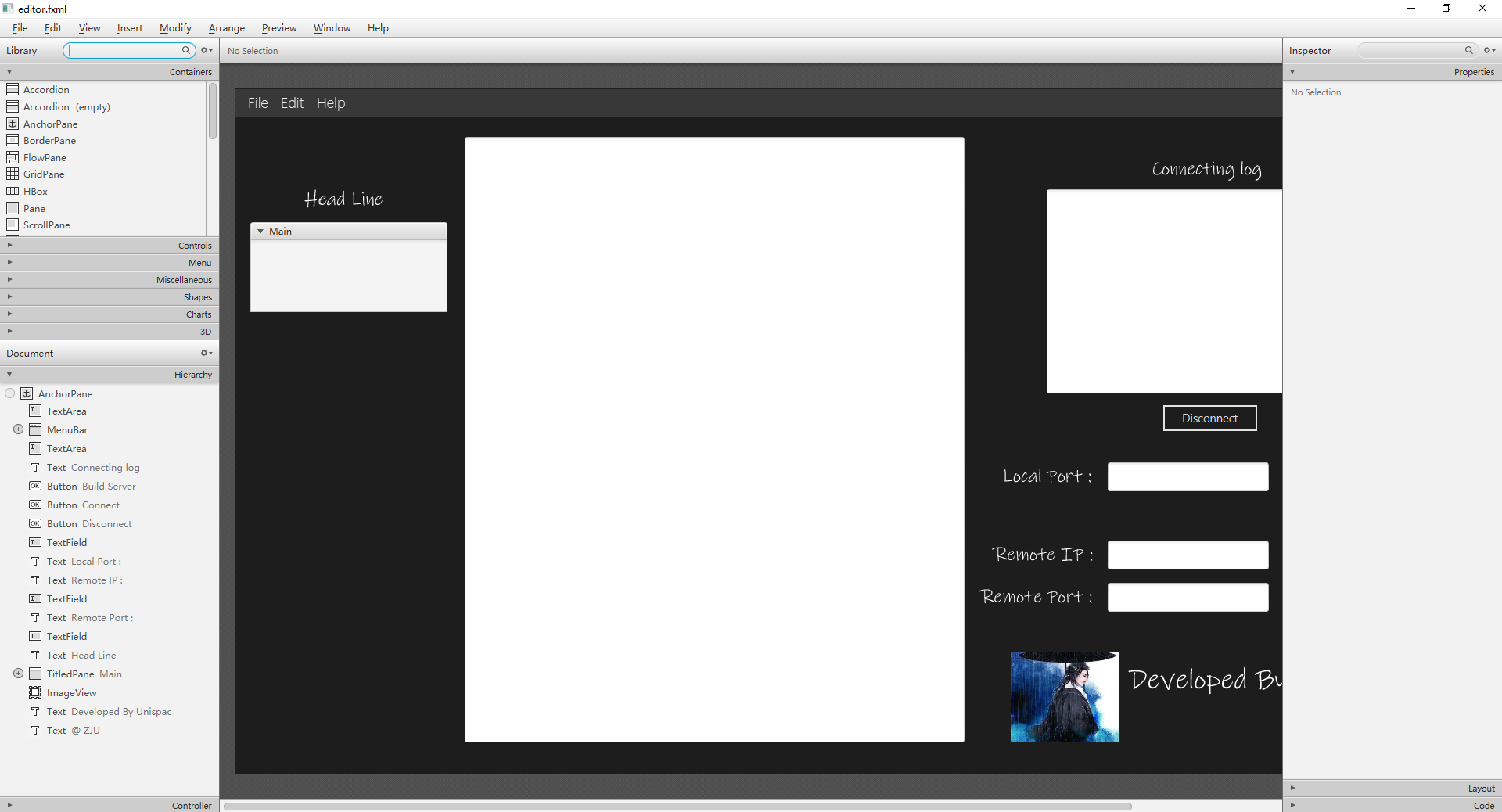
根目录下的view是我们的前端内容。含一个fxml文件editor.fxml，一个css文件DarkTheme.css，两张图片资源文件。



**Chapter2 GUI设计介绍**

这一部分我们主要挑选了GUI前后端中几个重要的设计细节来介绍。

1. **前端设计**
2. 可视化编辑



前端的布局已经在Scene Builder中实现了。同时，我们也在这个可视化的设计平台中，为控件设置了ID，设置了事件处理器的名称。Scene Builder为我们生成了fxml文件，在加载fxml文件的时候，这些ID和时间处理器名称就会与我们控制器里面定义的对象和方法映射和绑定。

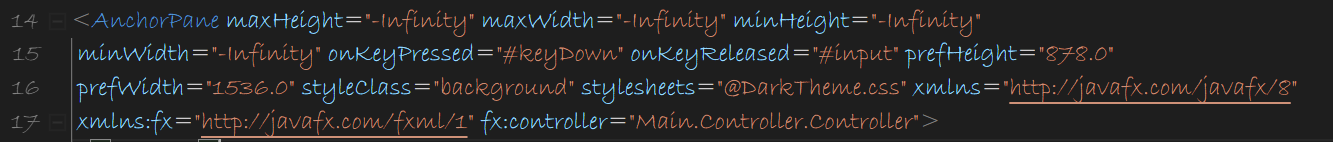
我们同时也为前端指定了css文件。我们从互联网上下载了DarkTheme.css这个灰黑风格的层叠样式表文件，然后在设计平台中将它绑定。

1. 控制器映射

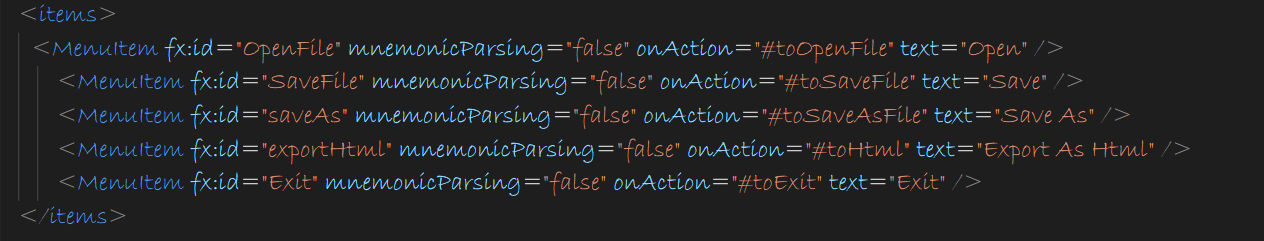
在Main中初始化窗口的时候，我们调用了

Parent root = FXMLLoader.load(getClass().getResource("/view/editor.fxml"));

这样一来fxml文件就被装载进入了我们的java程序中。背后的加载器会解析fxml文件，将组织成fxml格式的信息与java系统中的信息映射起来。



在顶层的AnchorPane中，我们在最后用fx:controller=”Main.Controller.Controller”将前端文件和Controller类绑定在类一起。

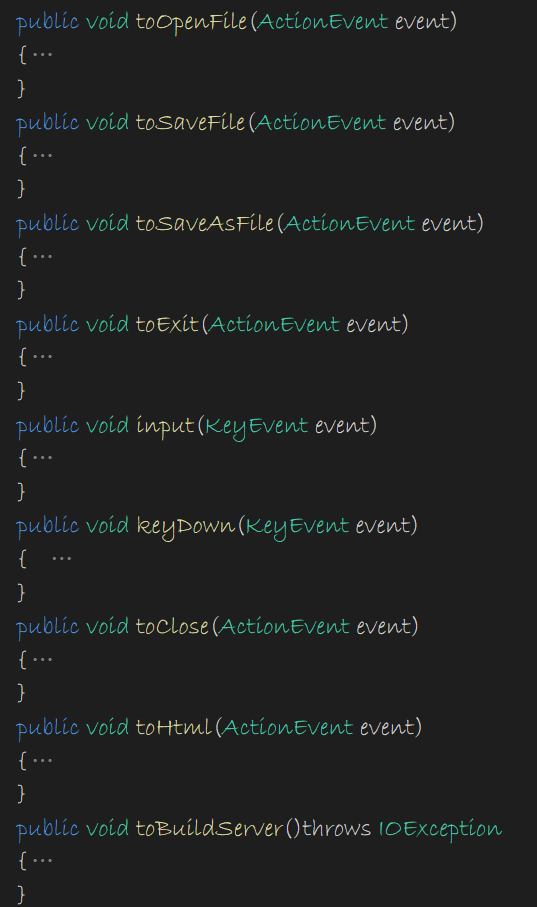


同时，可以看到，fxml文件中的控件，有一些我们为它指定了id，对应于fxml代码中的fx:id=”xxx”。

我们也为控件指定了事件处理器，对应于fxml代码中onAction=”#xxxx”等。



在Controller.class类中，我们定义了与id对应的相应的控件对象。用@FXML作为编译提示。



同时也定义了一系列方法，这些方法也分别与fxml中定义的事件处理器名称同名。

在fxml文件被装载后，那些以fxml格式表示的控件就会被映射到我们定义的同名的java数据结构中，同时根据fxml中指定的事件处理器名会自动与定义的同名方法绑定。

这样一来GUI的前端界面就和后端的控制器逻辑连接起来了。

1. **导出为HTML**

public void toHtml(ActionEvent event)

{

File myFile=exportHtml.showSaveDialog(Main.currentStage);

if(myFile!=null)

try

{

Md2x md2x=new Md2x();

BufferedWriter out = new BufferedWriter(new OutputStreamWriter

(new FileOutputStream(myFile.getAbsolutePath()),"UTF-8"));

out.write(md2x.parse(inputArea.getText()));

out.close();

}catch(Exception e)

{

e.printStackTrace();

}

}

调用Md2x库提供的方法，把markdown文本导出为html格式，然后再输出到指定目标。读写统一都使用”UTF-8”，避免了中文乱码。

1. **实时更新标题栏**



编辑器会以一定时间间隔，实时的扫描文本输入框中的内容，对它进行标题分析，然后在左侧的标题栏将各级标题的层次结构展示出来。

可以看到，这其实就是一个层层嵌套的递归框架。每一个标题对应的都是javafx中的一个titlePane，每个titlePane下有一个FlowPane，子标题的titlePane可以直接push到里面去。

我们实现了一个checkHeadLine方法来实现标题解析与渲染。

public void checkHeadLine()

{

myHeadLine.clear();

String content=inputArea.getText();

if(lastText!=null&&lastText.equals(content))return;

else lastText=content;

int index=content.indexOf('#');

int ed;

int level;

while(index!=-1)

{

level=0;

while(index<content.length()&&content.charAt(index)=='#')

{

level++;

index++;

}

if(index<content.length()&&content.charAt(index)==' ')

{

ed=content.indexOf('\n',index);

if(ed<0)ed=content.length()-1;

if(ed>index)myHeadLine.add

(new headLine(content.substring(index,ed),level));

index=ed;

}

if(index>=0&&index<content.length())index=content.indexOf('#',index);

else index=-1;

}

ObservableList<Node> temp=headLineField.getChildren();

temp.clear();

TitledPane x;

int lastlevel=100;

index=0;

ed=0;

headLine tempHead=null;

FlowPane lastPane=null,tempPane;

if(myHeadLine.size()>0)tempHead=myHeadLine.get(0);

for(int i=0;i<myHeadLine.size();i++)

{

tempPane=new FlowPane();

x=new TitledPane(""+index, tempPane);

x.setText(tempHead.content);

x.setPrefWidth(200-20\*tempHead.level);

if(index==0)temp.add(x);

else levelStack[index-1].myPane.getChildren().add(x);

lastlevel=tempHead.level;

lastPane=tempPane;

if(i+1<myHeadLine.size())

{

tempHead=myHeadLine.get(i+1);

if(tempHead.level>lastlevel)

{

levelStack[index].level=lastlevel;

levelStack[index].myPane=lastPane;

index++;

}

else

{

while(index>0&&levelStack[index-1].level>=tempHead.level)index--;

}

}

}

}

}

每次这个方法被调用，都会从输入框中获得最新的文本串，然后开始扫描’#’符号，看是否是markdown中标题格式，并且判定标题的等级。

我们维护了一个名为levelStack的栈，这个栈被定义为Controller类的一个私有成员。

private headLine levelStack[]=new headLine[12];

当前扫描到的标题：

如果比上一个标题的等级高显然它属于上一个标题的子标题，它的titlePane显然应该放在上一个标题的pane中。

如果等级相同，则是平行的，依然放在上一个pane放的那个父pane中。

如果等级更高，则应该向上一个等级的pane回溯跳出。

这样一个递归和嵌套的过程，我们用一个stack来维护非常合适。

嵌套关系的处理直接作用在了GUI界面的主titlePane上。我们定时的调用这个方法，更新GUI控件来实时的反应标题目录。

JavaFx要求对GUI控件的操作必须在javafx同一个线程中进行。所以我们不能直接在Controller中设置定时器。这样的异步处理是不支持的。

解决方法是在Main类的start方法中建立一个定时器，通过设置Platform.runLater让定时器触发的事件合并到窗口线程中执行。

1. **与其他模块通信**

Controller是处理GUI后端事件的模块，它起到的作用是维护整个GUI逻辑。

类似于服务器建立，服务器连接等功能的实现，是在其他模块中完成的。而其他模块要修改GUI的内容又必须通过Controller模块，我们以主类Main为通信的中继点。

前面说了，Controller对象是在装载FXML文件时自动实例化的，而非我们在代码中实例化的。好在Controller对象提供了一个initialize的方法，在对象创建的时候被调用。我们将它重载，把自己的句柄传递到Main的静态成员中去就可以获得这个对象的引用了。其他模块就可以通过Main的这个静态成员获取对Controller的访问。

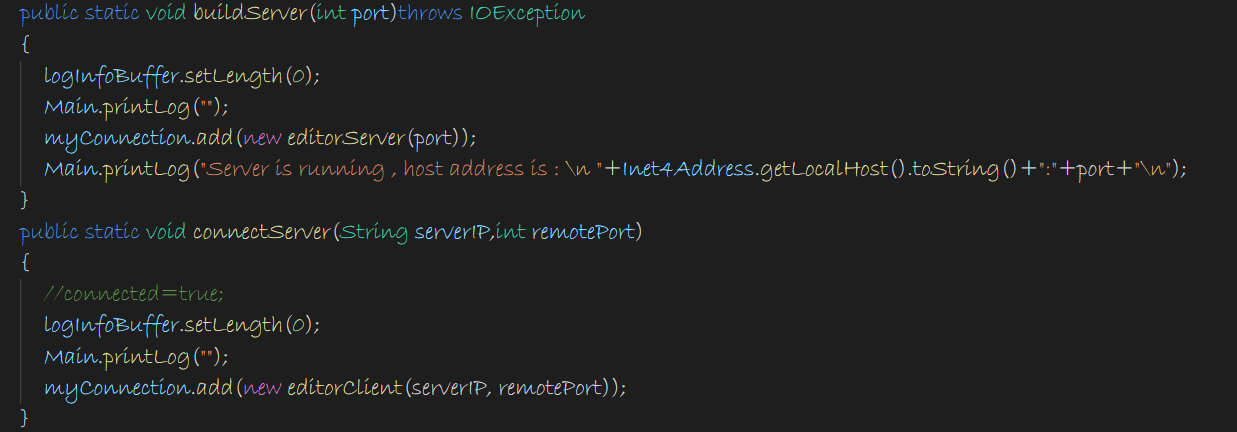
同时像服务器建立和连接等接口我们都是放在Main的静态方法中的，它们会创建相应的服务器对象和客户端对象。

Controller处理服务器创建/客户端连接请求的时候，也是通过访问Main的静态方法来间接完成的。

**Chapter3 实现协同编辑**

这一部分我们主要介绍协同编辑的实现，包括连接的建立，通信的规范，防止用户争抢的惩罚算法。

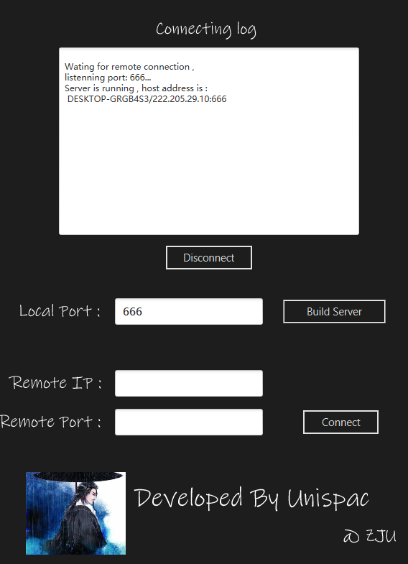
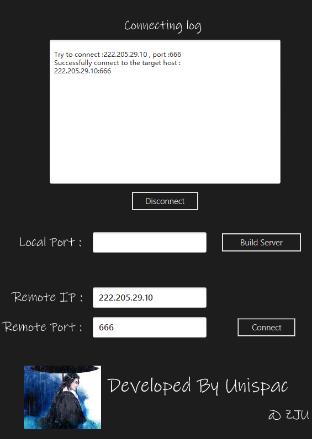
1. **建立连接**
2. Main中的连接接口

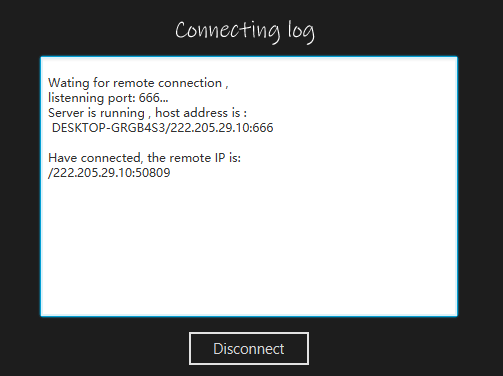


我们在Main中提供了两个静态方法作为服务器建立和客户端连接的接口。这两个接口将相应的信息输出到我们界面中的连接日志区域，同时会实例化相应的连接对象，并将相应的句柄加入名为myConnection的容器中。

editorServer对象和editorClient对象被创建的时候，就会自动的创建连接线程。服务器会在端口守候监听，客户端会尝试向目标地址发送连接请求。myConnection容器负责管理所有的连接，这样只要我们点下disconnect，所有的连接线程都会被执行interrupt。

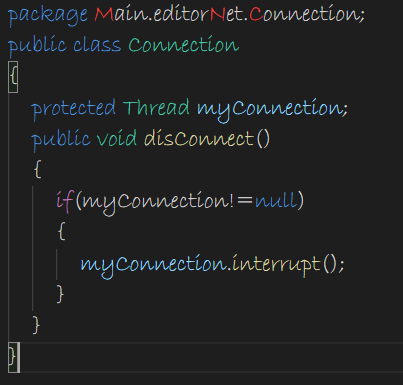
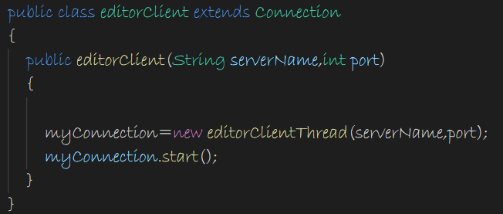
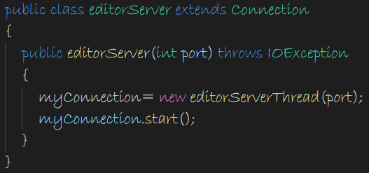
如图，输入本地端口，点击Build Server。控制器中相应的事件处理被触发，然后它会再调用Main中的buildServer接口建立起服务器。log中会输出相应的连接信息，表明自己的本地IP和监听端口。此时开启另一个编辑器，输入远程服务器的IP和端口，点击Connect就会调用connectServer方法来建立连接。下面展示了一个连接过程。





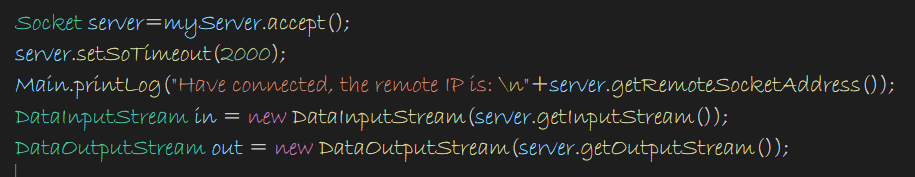
1. 两个Connection线程

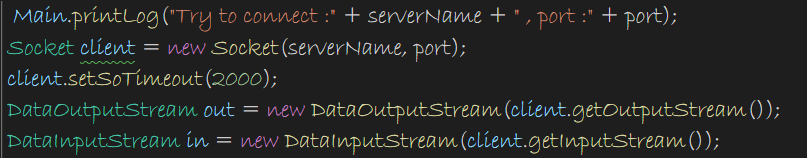
在抽象的层次上，我们只需要实例化一个editorClient或者editorServer就能完成客户端连接或者服务器建立。具体的实现其实是通过分别建立线程实现的。



editorClient和editorServer都是Connection类的子类，Connection类主要是为管理这两个连接类提供了统一的接口，非常简单。一个成员保存连接线程对象的引用，一个方法用来中断这个线程。两个分别为客户端和服务端的子类做的其实就是在构造函数里实例化相应的线程对象，然后启动它们。

最底层的editorServerThread和editorClientThread中是我们的通信的底层实现。在我们的底层实现中，只实现了点对点通信。即建立一个服务器，这个服务器监听，当有一个客户端连接后，它们就开始互相保持持续的通信，其它的客户端就无法再连接了。但是可以通过再build一个新的服务器，这样就会又出现一个线程监听了。。但由于我们的协同编辑逻辑主要是针对二人的，所以那样的多方通信会导致整个协同编辑的体验比较差。





上面分别是服务端和客户端建立连接时的调用。客户端线程启动后，调用accept方法开始阻塞，等待客户端连接。客户端线程启动后，新建一个Socket对象和远端建立连接。

连接建立后，它们通过socket对象获得互相的输入输出流。双发的通信通过in/out这个管道进行。

双方建立连接后，会互相发送一些确认消息，如ready，initial等，这里不描述细节。这些确认消息主要是为了确认连接的稳定和身份确认。

上述的确认流程过后，由服务端率先将自己的文本框中的内容发送到客户端，客户端接收然后更新到自己的文本框中去。之后，双方会轮流的向对方发送自己文本框中的内容。一方发送，一方接收，实现内容的同步。发送和接收都采用的阻塞式IO，这样可以让这个轮流制度有序的进行。同时，阻塞式IO也是我们的防抢占算法得以实现的基础。

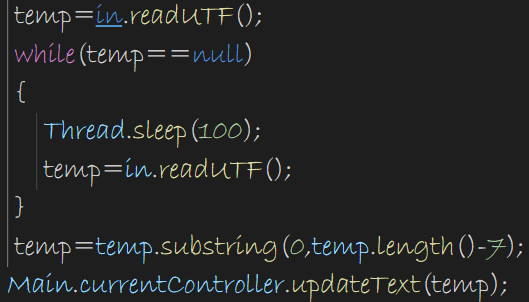
两个线程中都有一个interrupt事件的异常处理，被interrupt时会结束循环，终结连接。

1. 防抢占算法

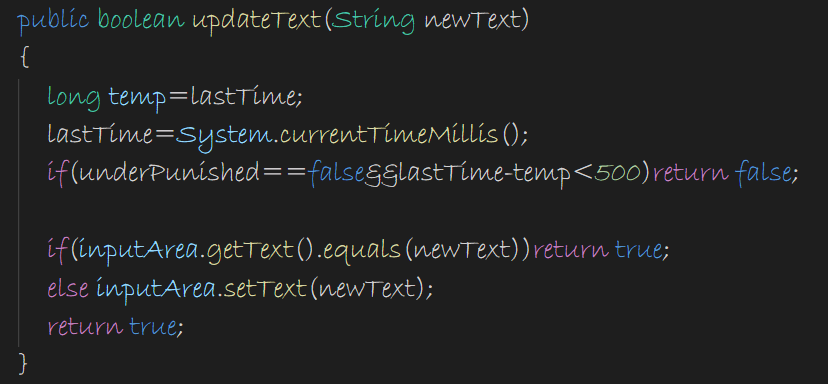
协同编辑中，最大的问题就是，如何让协同编辑有序的进行。双方都在对同一个区块进行编辑，对方的内容传来会覆盖自己的，自己的内容传过去会覆盖对方的。这样的编辑不是协同编辑，变成了争夺。无序的合作会导致1+1=0。

比较精细化的设计是可以把文档划分成若干块，同步更新是每一块分别更新。这样双方可以在对不同块进行同时编辑时不会遇到矛盾。但是这样的设计需要更精细的数据结构来管理。而我们只是简单的采用了javafx提供的文本输入框，整个框内的内容被作为整体处理和传送。这实在是因为到了期末，时间和精力有限，做的非常粗糙的处理。

我们采用的防抢占算法是，只允许一个人同时的编辑，另一个可以实时的查看对方编辑的内容。对方停止输入后，另一个人则可以接替他输入。即，当有一个人开始输入和编辑的时候，另一个人不能贸然的也随便往内容里面插入。同一时刻，只能有一个人获得文档的主导权。这很像桌面的远程操作一样，如果两个人都在争夺键盘和鼠标的控制权，那么会变得非常混乱。我们引入了惩罚机制来防止两个人同时操作的情况发生。



如上所示，接收到一个字符串后，更新文本框内容是通过Main获得controller的引用，然后再调用了controller中的一个用于更新文本框内容的接口进行的。



可以看到这个接口会做一个判断。只有对方在未出于被惩罚状态，以及距离自己最后一次输入文本已经过了0.5s，对方对自己的更新才有效，否则会返回false。

当一方发现了对方对自己的更新为false时，说明对方想要抢占自己对文本框的主导权。这个时候更新文本失败的一方会被惩罚，而处于对文本框控制的一方会获得主导权。在这个状态中，处于惩罚的一方将不再发送自己的文本内容，出于主导权的一方也不会再接收对方的内容。以收发一次为一轮，这个状态会持续4轮，总计约两秒。

这样以来，当一个人在连续的对一个文档进行编辑的时候，另一方就难以插手来进行编辑，因为它一动手就会立刻遭到系统的制裁。

在和室友使用这个算法进行测试的时候，我们发现这样的机制有效性非常好。抢占的问题在一定程度上被解决了。

PS:

由于临近期末，时间和精力非常有限。最后我只实现了前面基本要求中的所有内容。所以最后的版本中未能实现实时渲染，这是一个非常大的遗憾。同时，文本输入框也是用的javafx自带的，对于文本编辑框不能做到特别精细的控制。服务器通讯也是通过粗暴的整个文本串传输实现的，而不是基于类似git那样的对修改细节做检查。协同方面，也只实现了点对点的通信。总的来说，还有很多需要改进的地方，希望以后能对这些细节进行进一步的弥补和修改。