**TJ Red Alert需求与设计报告**

[**开始界面（GameMenuScene）**](#开始界面)

[**Class GameMenu类**](#ClassGameMenu)

[**Class StartMenu类**](#ClassStartMenu类)

[**Class ServerMenu类**](#ClassServerMenu类)

[**Class ClientMenu类**](#ClassClientMenu类)

[**游戏场景**](#_游戏场景_1)

[**游戏场景数据结构**](#数据结构)

[**GamingScene类**](#GamingScene类)

[**Money类**](#money类)

[**Electricity类**](#Electricity类)

[**MouseRect类**](#mouseRect)

[**ManufactureMenu类**](#ManufactureMenu类)

[**游戏场景重要算法**](#游戏场景重要算法)

[**鼠标滚动**](#鼠标滚动)

[**键盘滚动与跳转**](#键盘滚动与跳转)

[**鼠标框选**](#鼠标框选)

[**地图相关数据结构**](#地图相关数据结构)

[**GridVec2类、GridDimen类、GridRect类**](#GridVec2类、GridDimen类、GridRect类)

[**GridMap类**](#GridMap类)

[**单位**](#单位)

[**单位相关数据结构分析**](#单位)

[**Unit类**](#Unit类)

[**BuildingUnit类**](#BuildingUnit类)

[**FightUnit类**](#FightUnit类)

[**UnitManager类**](#UnitManager类)

[**建筑类（Building）**](#建筑类（Building）)

[**Base类**](#Base类)

[**Class PowerPlant类**](#ClassPowerPlant类)

[**Class OreRefinery类**](#ClassOreRefinery类)

[**Class Barracks类**](#ClassBarracks类)

[**Class WarFactory类**](#ClassWarFactory类)

[**子弹类（Bullet)**](#子弹类（Bullet）)

[**攻击类（Fighter）**](#攻击类（Fighter）)

[**Soldier类**](#Soldier类)

[**Tank类**](#Tank类)

[**AttackDog类**](#AttackDog类)

[**单位功能实现逻辑**](#单位功能实现逻辑)

[**建筑单位生产**](#建筑单位生产：)

[**Fight类单位生产**](#Fight类单位生产：)

[**单位攻击**](#单位攻击：)

[**网络**](#网络)

[**GameMessages消息格式**](#GameMessage)

[**socket\_message类**](#socket_message)

[**talk\_to\_client类**](#talk_to_client类)

[**talk\_to\_server类**](#talk_to_server类)

[**SocketClient实现**](#实现：)

[**发送**](#发送)

[**接收**](#接受)

[**游戏消息**](#游戏消息)

[**socket\_server类**](#s_s)

[**寻路**](#寻路)

[**Grid类**](#Grid类)

[**PathFinder类**](#PathFinder类)

[**A\*寻路算法**](#A寻路算法)

[**实现原理**](#实现原理)

[**寻路步骤**](#寻路步骤)

**开始界面（GameMenuScene）**

开始界面为用户进入游戏后的第一个界面，主要在GameMenuScene.h和GameMenuScene.cpp中实现，主菜单逻辑,游戏菜单逻辑,服务器端菜单,客户端菜单分别在Class GameMenu, Class StartMenu, Class ServerMenu Class, Class ClientMenu中实现。

**Class GameMenu类**

Class GameMenu公有继承自cocos2d::Layer类

**Class GameMenu类中的成员变量如下：**

|  |  |  |
| --- | --- | --- |
| 变量名 | 变量类型 | 作用 |
| Title | cocos2d::LabelTTF | 游戏名称 |
| Back | cocos2d::Sprite | 背景图片 |
| start\_label | cocos2d::MenuItemImage | start按钮 |
| credits\_label | cocos2d::MenuItemImage | credits按钮 |
| quit\_label | cocos2d::MenuItemImage | quit按钮 |
| menu | cocos2d::Menu | 按钮菜单 |

**Class GameMenu类中的成员函数如下**

* static cocos2d::Scene\* createScene();按照cocos2d内存管理方式创建GameMenu对象
* virtual bool init();初始化开始界面的各精灵以及控件
* void menuItemStartCallback(cocos2d::Ref\*pSender);//Start按钮触发时的回调函数
* void menuItemSetCallback(cocos2d::Ref\*pSender);//Set按钮触发时的回调函数
* void menuItemQuitCallback(cocos2d::Ref\*pSender);//Quit按钮触发时的回调函数
* void menuItemHelpCallback(cocos2d::Ref\*pSender);//Help按钮触发时的回调函数

**Class StartMenu类**

Class StartMenu公有继承自cocos2d::Layer类，用来实现客户端与服务端的选择。

**Class StartMenu类中的成员变量如下：**

|  |  |  |
| --- | --- | --- |
| 变量名 | 变量类型 | 作用 |
| background | cocos2d::Sprite | 背景图片 |
| server\_label | cocos2d::MenuItemImage | 服务端开始的按钮 |
| client\_label | cocos2d::MenuItemImage | 客户端开始的按钮 |
| back\_label | cocos2d::MenuItemImage | 返回时的按钮 |
| menu | cocos2d::Menu | 按钮菜单 |

**Class StartMenu 类中的成员函数如下：**

* static cocos2d::Scene\* createScene();按照cocos2d内存管理方式创建StartMenu对象
* bool init() override;初始化开始界面的各精灵以及控件
* void menuServerCallback(cocos2d::Ref\* pSender);以服务端身份游戏的回调函数
* void menuClientCallback(cocos2d::Ref\* pSender);以客户端身份游戏的回调函数
* void menuBackCallback(cocos2d::Ref\* pSender);返回时的回调函数

**Class ServerMenu类**

Class StartMenu公有继承自cocos2d::Layer类,实现服务端的创建，界面由背景图片，Start server按钮，Start game按钮，Back按钮构成

**Class StartMenu类中的成员函数如下：**

* static cocos2d::Scene\* createScene();按照cocos2d内存管理方式创建ServerMenu对象
* bool init() override;初始化开始界面的控件
* void menuStartServerCallback(cocos2d::Ref\* pSender);Start server按钮的回调函数，用来判断游戏是否建立了服务端，如果没有则根据端口号建立服务端，然后建立本机客户端，并将客户端添加到服务端
* void menuStartGameCallback(cocos2d::Ref\* pSender);Start game按钮的回调函数，用来判断游戏是否建立了服务端，如果已建立则触发服务端开启游戏操作，服务端向每个客户端发出开始游戏消息，随后进入游戏界面
* void menuBackCallback(cocos2d::Ref\* pSender);Back按钮的回调函数
* void editBoxReturn(cocos2d::ui::EditBox\* editBox) override;输入框的回调函数
* void connectionSchdeule(float f);连接情况自动更新函数

**Class StartMenu类中的成员变量如下：**

|  |  |  |  |
| --- | --- | --- | --- |
| 变量名 | 变量类型 | 初值 | 作用 |
| connection\_msg\_ | cocos2d::Label \* | nullptr | 连接信息 |
| socket\_server\_ | socket\_server \* | nullptr | 服务端套接字实例 |
| socket\_client\_ | boost::shared\_ptr<talk\_to\_server> | nullptr | 客户端套接字实例 |
| inputbox | cocos2d::ui::EditBox | 未初始化 | 端口输入框 |
| background | cocos2d::Sprite | 未初始化 | 背景图片 |
| start\_label | cocos2d::MenuItemImage | 未初始化 | Start as server按钮 |
| start\_game\_label | cocos2d::MenuItemImage | 未初始化 | Start as client按钮 |
| back\_label | cocos2d::MenuItemImage | 未初始化 | 返回时的按钮 |
| menu | cocos2d::Menu | 未初始化 | 按钮菜单 |

**Class ClientMenu类**

Class ClientMenu公有继承自cocos2d::Layer类，实现客户端的创建。

**Class ClientMenu类中的成员函数如下：**

* static cocos2d::Scene\* createScene();按照cocos2d内存管理方式创建ClientMenu对象
* bool init() override;初始化游戏场景及控件
* void menuStartGameCallback(cocos2d::Ref\* pSender); Join game按钮的回调函数，如果已经触发定时更新连接情况，等待服务端发回开始游戏消息，连接失败则显示失败消息
* void menuBackCallback(cocos2d::Ref\* pSender);Back按钮的回调函数
* void wait\_start();等待开始的函数
* void startSchedule(float f);连接情况更新函数

**Class ClientMenu类中的成员变量如下：**

|  |  |  |  |
| --- | --- | --- | --- |
| 变量名 | 变量类型 | 初值 | 作用 |
| connection\_msg\_ | cocos2d::Label \* | nullptr | 连接信息 |
| timer | int | 0 | 计时器用来更新连接信息 |
| socket\_client\_ | boost::shared\_ptr<talk\_to\_server> | nullptr | 客户端套接字实例 |
| ip\_box | cocos2d::ui::EditBox | 未初始 | ip地址输入框 |
| port\_box | cocos2d::ui::EditBox | 未初始 | 端口号输入框 |
| background | cocos2d::Sprite | 未初始 | 背景图片 |
| start\_label | cocos2d::MenuItemImage | 未初始 | Join game按钮 |
| back\_label | cocos2d::MenuItemImage | 未初始 | 返回时的按钮 |
| menu | cocos2d::Menu | 未初始 | 按钮菜单 |
| connection\_msg\_ | cocos2d::Label | 未初始 | 连接情况显示标签 |

## 游戏场景

**游戏场景数据结构**

游戏场景的主要逻辑在class GamingScene类中实现，与游戏场景相关的类还有class Money，class Electricity，class MouseRect，class ManufactureMenu。

**GamingScene类**

定义在GamingScene.h中，实现在GamingScene.cpp中。GamingScene继承自cocos2d::Layer，是一个完整的cocos2d图层，便于初始化、更新以及场景切换。

GamingScene类是游戏场景的主要部分，主要包含消息处理，战场地图，信息地图，单位管理，鼠标操作，键盘操作，生产菜单，金钱显示，电力显示等功能部件，GamingScene实现了这些部件的初始化和调度。

**GamingScene类中的成员变量如下：**

|  |  |  |
| --- | --- | --- |
| 变量名 | 变量类型 | 变量含义 |
| \_socketServer | socket\_server \* | 服务端套接字 |
| \_socketClient | shared\_ptr<talk\_to\_server> | 客户端套接字 |
| \_tiledMap | TMXTiledMap \* | 瓦片地图 |
| \_gridMap | GridMap \* | 格点地图 |
| \_unitManager | UnitManager \* | 单位管理器 |
| \_msgs | GameMessageGroup \* | 消息集 |
| \_mouseRect | MouseRect \* | 鼠标选框 |
| \_mousePosition | Point | 鼠标位置 |
| \_money | Money \* | 金钱 |
| \_electricity | Electricity \* | 电力 |
| \_manufactureMenu | Manufacture \* | 生产菜单 |
| \_menuSpriteLayer | Layer \* | 生产图标 |
| \_soldierWaitingNum | LabelBMFont | 待生产士兵数 |
| \_tankWaitingNun | LabelBMFont | 待生产坦克数 |
| \_attackDogWaitingNum | LabelBMFont | 待生产狗数 |
| \_keyStatus | map<KeyCode,bool> | 键盘状态 |
| \_keyUpdate | bool | 键盘操控视角 |

**GamingScenne类中的成员函数如下：**

* static Scene \* createScene(socket\_server \* SocketServer, shared\_ptr<talk\_to\_client>)创建场景，返回指向新场景的指针
* static GamingScene \* create(socket\_server \* socketServer, shared\_ptr<talk\_to\_server> socketClient); 按照cocos2d的内存管理模式创建GamingScene对象，返回指向新对象的指针
* virtual bool init(socket\_server \* socketServer, shared\_ptr<talk\_to\_server> socketClient);

初始化战场场景和各个功能控件

* virtual void update(float f)override; 战场场景更新函数
* void mapScroll(); 地图滚动函数
* bool onTouchBegan(cocos2d::Touch\* touch, cocos2d::Event\* event)override; 触摸开始事件回调函数，获取触摸位置并更新\_mouseRect指向的对象中的成员变量
* void onTouchMoved(cocos2d::Touch\* touch, cocos2d::Event\* event)override; 触摸移动事件回调函数，随着鼠标移动更新\_mouseRect指向的对象
* void onTouchEnded(cocos2d::Touch\* touch, cocos2d::Event\* event)override; 触摸结束事件回调函数，获取数模结束位置并调用单位框选函数
* void onKeyPressed(cocos2d::EventKeyboard::KeyCode keyCode, cocos2d::Event \* event)override;

键盘按压事件回调函数，根据按键执行对应操作，更新\_keyStatus

* void onKeyReleased(cocos2d::EventKeyboard::KeyCode keyCode, cocos2d::Event \* event)override;键盘释放事件回调函数，更新\_keyStatus
* void keyPressedToMove(EventKeyboard::KeyCode keyCode); 根据按键不同，向不同方向滚动地图
* void updateBuildingMenu(float f); 更新建筑生产菜单
* void updateArmyMenu(float f); 更新攻击单位生产菜单
* void updateFightUnitWaitingNum(float dt); 更新攻击单位待生产数量
* CC\_SYNTHESIZE(UnitManager\*, \_unitManager, unitManager);获取与设置\_unitManager
* CC\_SYNTHESIZE(Money\*, \_money, money);获取与设置\_money

**Money类**

定义在Money.h中，实现在Money.cpp中，公共继承自cocos2d::LabelTTF类

Money类用于存储和管理用户当前持有的金钱，并在战场场景中更新和显示金钱的数值。

Money类公共继承自cocos2d::LabelBMFont ，便于金钱数值的显示和快速更新。

**Money类中的成员函数如下：**

* bool init() override;初始化金钱
* void update(float f);更新金钱数目
* void updateMoneyLabel(); 更新金钱显示
* bool checkMoney(int cost) const;检查金钱是否足够
* void costMoney(int cost);金钱减少
* void addMoney(int add);金钱增加
* void setMoneyInPeriod(int moneyInPeriod);设置每段时间内金钱增加的数目
* void addMoneyInPeriod(int add);在原基础上增加金钱
* void cutMoneyInPeriod(int cut); 在原基础上减少金钱

**Money类类中的成员变量如下：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 变量名 | 类型 | 种类 | 初值 | 含义 |
| \_money | int | private | 0 | 金钱 |
| \_time | int | private | 0 | 时间 |
| \_moneyString | std::string | private | 未初始化 | 金钱字符 |
| \_moneyInPeriod | int | private | MONEY\_IN\_PERIOD | 单位时间内金钱增加数目 |
| \_period | int | private | PERIOD | 金钱增加的时间 |

**Electricity类**

声明在Electricity.h中定义在Electricity.cpp中，公有继承自cocos2d::LabelTTF类。

**Electricity中的成员变量如下：**

|  |  |  |  |
| --- | --- | --- | --- |
| 变量名 | 变量类型 | 初值 | 含义 |
| \_electricity | int | 0 | 总电力值 |
| \_usedElectricity | int | 0 | 已用电力值 |
| \_electricityString | std::string | 未初始化 | 电量上限字符 |
| \_usedElectricityString | std::string | 未初始化 | 已用电力字符 |
| \_showElectricity | std::string | 未初始化 | 电力字符集合 |
| \_tempMoneyInPeriod | int | 未初始化 | 每个period增加的量 |
| powerOff | int | 0 | 电力丧失 |

**Electricity中的成员函数如下：**

* bool init() override;初始化设置电力
* void updateLabel();更新电力标签
* bool checkElectricity(int cost) const;检查电力是否足够
* void costElectricity(int cost);建筑消耗电力函数
* void freeElectricity(int free);当建筑被毁坏时释放它所使用的电力
* void addElectricity(int add);当建造发电厂或基地时增加电力
* void cutElectricity(int cut);当发电厂或基地被毁坏时减少电力

**MouseRect类**

定义在MouseRect.h中，实现在MouseRect.cpp中。MouseRect类继承自cocos2d::DrawNode，cocos2d::DrawNode是cocos2d引擎提供的基础绘图类，可以灵活地绘制出空心、实心矩形多边形等基础形状。MouseRect类是用户通过触摸移动或者按住拖动鼠标的操作，在地图上拖动绘制出矩形这一功能的抽象，MouseRect需要在视角变化时跟随变化绘制点的坐标，才能保证绘制出符合用户想法的矩形。

**MouseRect类中的成员变量如下：**

|  |  |  |
| --- | --- | --- |
| 变量名 | 变量类型 | 变量含义 |
| \_touchStartToGL | Point | 触摸开始点的世界坐标 |
| \_touchEndToGL | Point | 触摸结束点的世界坐标 |
| \_touchStartToMap | Point | 触摸开始点的地图坐标 |
| \_touchEndToMap | Point | 触摸结束点的地图坐标 |

**MouseRect类中的成员函数如下：**

* void setTouchStartToGL(cocos2d::Point point); 设置触摸开始点的世界坐标
* void setTouchEndToGL(cocos2d::Point point); 设置触摸结束点的世界坐标
* void setTouchStartToMap(cocos2d::Point mapPoint); 将触摸开始点的世界坐标转换为地图坐标
* void setTouchEndToMap(cocos2d::Point mapPoint); 将触摸结束点的世界坐标转换为地图坐标
* Rect getMouseRect(); 获取由触摸开始点的地图坐标和触摸结束点的地图坐标构成的矩形
* void update(float f) override; 鼠标移动过程中更新绘制矩形
* void reset(); 重置MouseRect对象

**ManufactureMenu类**

定义在ManufacturMenu.h中, 实现在ManufactureMenu.cpp中。ManufactureMenu类继承自cocosd::Menu，实现按钮功能。ManufactureMenu类用来实现单位的生产按键，并做简单的电力，金钱检测。

**ManufactureMenu类中的成员变量如下：**

|  |  |  |
| --- | --- | --- |
| 变量名 | 变量类型 | 变量含义 |
| \_buildingButton | MenuItemImage \* | 建筑建造按钮 |
| \_armyButton | MenuItemImage \* | 战斗单位建造按钮 |

**ManufactureMenu类中的成员函数如下：**

* bool init()override; 初始化生产菜单
* void setBuildingCallBack(std::function<void(Ref\*)>); 设置建筑按钮回调函数
* void setArmyCallBack(std::function<void(Ref\*)>); 设置战斗单位回调函数

**游戏场景重要算法**

**鼠标滚动**

当鼠标在蓝色区域，地图以两倍向鼠标所靠近窗口的边缘一侧移动；当鼠标在灰色区域时，地图以一倍速向鼠标所靠近窗口的边缘一侧移动；当在鼠标在橙色区域时，地图以两个方向上的两倍速移动；当鼠标在黄色区域时，地图以两个方向上的一倍速移动；当鼠标在绿色区域时，地图在鼠标靠近窗口边缘一侧以两倍速移动，另一侧以一倍速移动。

**键盘滚动与跳转**

空格跳转至基地位置，如果还未建造基地则跳转至默认出生点。

W—向上；S—向下；A—向右；D—向左。

键盘按下时，void onKeyPressed(KeyCode keyCode, Event \* event);响应键盘事件执行对应操作，void keyPressedToMove(KeyCode keyCode);在update函数中检测被按下按键，向对应的方向移动，void onKeyReleased(KeyCode keyCode, Event \* event);在按键松开时，设置按键状态。

**鼠标框选**

坐标轴为屏幕坐标，蓝色框Map1为地图初始位置，橙色框Map2为地图最终位置，绿色框即为鼠标框选范围，绿色框由相对于地图的触摸开始与结束两位置绘制而成。

\_touchBeganToMap = \_touBeganToGL – map1Position;

\_touchEndToMap = \_touEndToGL – map2Position;

mouseRect

Map1

\_touchEndToMap

\_touchBeganToMap

\_touchBeganToGL

Map1Position

Map2Position

\_touchEndToGL

Map2

**鼠标框选矩形绘制说明**

**地图相关数据结构**

**GridVec2类、GridDimen类、GridRect类**

定义在GridMap.h中,在GridMap.cpp中实现。

作用：分别作为cocos自带的Vec2和Size类和Rect类的替换类，其所有参数都为int类型，方便格点地图、寻路算法的设计和实现，加快游戏运算速度。

**GridRect类重要成员函数：**

* friend bool operator ==(const GridRect & rect1,const GridRect &rect2);重载==运算符
* bool containsPoint(const GridVec2 & point)const; 判断point是否在该GridRect内
* bool intersectsRect(const GridRect & rect)const; 判断rect是否与该GridRect相交
* bool insideRect(const GridRect &rect)const; 判断rect是否在该GridRect内

**GridMap类**

定义在GridMap.h中,在GridMap.cpp中实现。

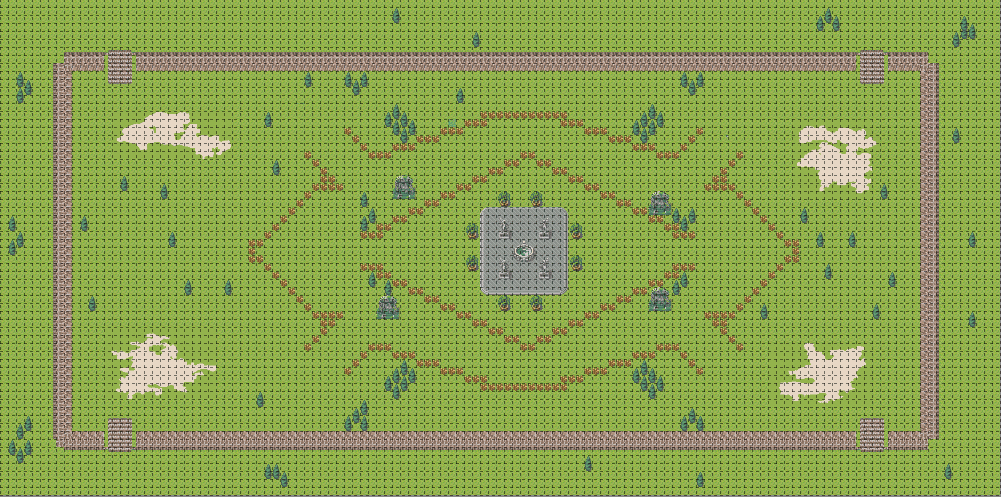
作用：继承cocos::layer,作为一个网格信息层,记录每个网格的信息，用于寻路，搜寻空位、范围内单位id获取.。

**重要成员变量：**

|  |  |  |
| --- | --- | --- |
| 变量名 | 类型 | 含义 |
| \_unitCoord | map<int, GridVec2> | 储存各单位的网格坐标 |
| \_barrierMap | vector <vector<int>> | 存储网格地图上每个格子的状态0为空 其余的代表unit的id或者地图障碍物体 |
| \_worldRect | GridRect | 整个地图的范围 |
| \_ladderRect1 | GridRect | 一号梯子的位置信息 |
| \_ladderRect2 | GridRect | 二号梯子的位置信息 |
| \_ladderRect3 | GridRect | 三号梯子的位置信息 |
| \_ladderRect4 | GridRect | 四号梯子的位置信息 |
| \_findPathMap | vector <vector<int>> | 寻路地图 |

**重要成员函数：**

* bool unitCoordStore(int unitId, const GridVec2 &position); 存储单位的位置信息
* bool unitCoordStore(int unitId, const GridRect & rect); 存储单位的位置信息
* void unitCoordRemove(int unitId,GridRect unitRect); 移除单位信息（一般是在单位灭亡时调用）
* int getUnitIdAt(const GridVec2& position)const; 获取position位置上的单位信息
* set <int> getUnitIdAt(const GridRect & range)const; 获取一定范围内的所有单位id
* bool checkRectPosition(const GridRect & rect)const; 判断rect在网格地图上是否为空
* GridVec2 getEmptyPointNearby(const GridVec2 & position)const; 获取point附近空的网格的坐标
* GridRect getEmptyRectNearby(const GridVec2 & point, const GridDimen & size)const; 获取point附近大小为size且为空的区域的rect
* friend void setCollisionPos(TMXTiledMap\* map,GridMap \* gmap); 设置瓦片地图中障碍物的信息
* bool checkBuildingRectPosition(const GridRect & rect)const; 判断建筑单位在这个范围是否可以生产
* GridVec2 getNewDestination(const GridVec2 & position,const map<GridVec2,int> & destination) const; 获取就目的地附近的空位作为单位移动的新目的地



**单位**

**Unit类**

定义在Unit.h中，实现在Unit.cpp中，公共继承自Sprite类。

Unit类是各种单位的基类，预留了单位的各项数值属性，设置了创建、移动、攻击等基础接口，实现了寻路和重新寻路策略。

**Unit类中的成员函数如下：**

* void setUnitManager(UnitManager \* uM);初始化用户状态
* virtual bool \_stdcall init(CampTypes camp,UnitTypes Type,GridVec2 point,TMXTiledMap\* map ,GridMap \*gridmap , int id=0);添加单位到瓦片地图
* bool getDamage(int hurt);单位受到伤害
* void removeFromMap();将单位信息从地图信息层移除
* void hideHpBar();隐藏血条
* void displayHpBar();显示血条
* virtual void deleteUnit();删除单位
* virtual void tryToFindPath();尝试进行寻路
* virtual std::vector<GridVec2> findPath(const GridVec2 & destination);寻找路径
* virtual void optimizePath();优化路径
* virtual void move();移动
* virtual void attack();攻击
* void initHpBar(UnitTypes type);初始化血条
* virtual void attackUpdate(float dt);攻击更新
* virtual void startAttackUpdate();开始攻击
* virtual void stopAttackUpdate();停止攻击
* virtual GridVec2 findEmptyPosToProduce();寻找空位生产单位（步兵和警犬）
* virtual GridVec2 findEmptyPosToProduceTank();寻找空位生产坦克单位
* virtual void startProduceUnit(UnitTypes proUnitType;开始建造单位
* virtual void stopProduceUnit(); 停止建造单位
* virtual void buildingUpdate();建造更新
* virtual void produceUpdate(float fd);生产fight类单位更新

**Unit类中的成员变量如下：**

|  |  |  |  |
| --- | --- | --- | --- |
| 变量名 | 类型 | 初值 | 含义 |
| \_unitType | UnitTypes | 未初始化 | 单位类型 |
| \_id | int | 未初始化 | 单位ID |
| \_camp | CampTypes | 未初始化 | 阵营 |
| \_health | int | 未初始化 | 单位生命值 |
| \_moneyCost | int | 未初始化 | 建造单位的价格 |
| \_currentHp | int | 未初始化 | 单位当前血量 |
| \_underAttack | bool | 未初始化 | 单位是否在攻击范围内 |
| \_unitSize | GridDimen | GridDimen默认初始化 | 单位所占的格点矩形大小 |
| \_unitRect | GridRect | GridRect默认初始化 | 单位当前所占的格点矩形 |
| \_unitCoord | GridVec2 | GridVec2默认初始化 | 单位坐标 |
| \_destination | Vec2 | 未初始化 | 单位移动目的地 |
| \_attackID | int | 未初始化 | 单位攻击目标的ID |
| \_producingUnitType | UnitTypes | 未初始化 | 所生产单位的类型 |
| \_producingState | int | 未初始化 | 单位生产状态 |
| \_enermyId | int | 未初始化 | 敌人ID |
| \_gridPath | std::vector<GridVec2> | 未初始化 | 格点路径 |
| \_isBuilding | bool | 未初始化 | 单位是否建造 |
| \_autoAttack | bool | 未初始化 | 单位能否自动攻击 |
| \_unitManager | UnitManager | nullptr | 单位管理器 |
| \_hpBar | LoadingBar | nullptr | 血条 |
| \_hpBGSprite | Sprite | nullptr | 血条背景 |
| \_battleMap | GridMap \* | nullptr | 格点地图 |
| \_tiledMap | TMXTiledMap \* | nullptr | 瓦片地图 |

**BuildingUnit类**

定义在Unit.h中，实现在Unit.cpp中，公共继承自Unit类。

BuildingUnit类是建筑单位管理器，用于统一管理当前战斗场景中的建筑单位，完成了建筑单位的建造等基础接口。

**BuildingUnit 类中的成员函数如下：**

* virtual void deleteUnit();删除单位
* void startProduceUnit(UnitTypes proUnitType);开始生产单位
* void stopProduceUnit();停止生产单位
* void produceUpdate(float fd);生产更新
* void updateBuildingBar(float dt);更新建造条

**BuildingUnit类中的成员变量如下：**

|  |  |  |  |
| --- | --- | --- | --- |
| 变量名 | 类型 | 初值 | 含义 |
| \_produceProcess | int | 未初始化 | fight类单位在该单位上的生产进度 |
| \_produceTime | int | 未初始化 | 单位建造所需时间 |
| \_buildingProgressfloat | float | 未初始化 | 建造进度 |
| \_progressbgSprite | Sprite \* | nullptr | 建造进度条背景 |
| progress | Sprite \* | nullptr | 建造进度条 |
| \_powerCost | int | 未初始化 | 所消耗的能量 |
| \_moneyProduce | int | 未初始化 | 所花费的金钱 |
| \_buildingType | UnitTypes | 未初始化 | 建筑种类 |

**FightUnit类**

定义在Unit.h中，实现在Unit.cpp中，公共继承自Unit类。

FightUnit类是攻击单位管理器，用于统一管理当前战斗场景中的攻击单位，完成了攻击单位的创建、移动、攻击等基础接口。完成了自动搜索敌人，自动攻击等高级功能的实现。

**FightUnit类中的成员函数如下：**

* void move();移动
* void moveEndedCallBack(); 移动结束后的回调函数
* virtual void shootBullet();射击子弹
* virtual void attack();攻击
* int searchNearEnemy();搜索附近敌人

**FightUnit类中的成员变量如下：**

|  |  |  |
| --- | --- | --- |
| 变量名 | 类型 | 含义 |
| **\_fighterType** | UnitTypes | 攻击单位类型 |
| **\_attacking** | bool | 是否进行攻击 |
| **\_moveSpeed** | float | 单位移动速度 |
| **\_attackForce** | int | 单位攻击力 |
| **\_attackSpeed** | int | 单位攻击速度 |
| **\_atkIDPosition** | int | 攻击目标位置 |
| **\_manualAttackScope** | int | 手动攻击范围 |
| **\_autoAttackScope** | GridDimen | 自动攻击范围 |

**UnitManager类**

定义在UnitManager.h中，在UnitManager.cpp中实现

作用：作为单位管理类，用于统一管理地图中的所有单位，负责所有单位在地图上的生成和消灭，提供鼠标框选和点击选中单位、创建单位、移动单位、攻击单位、查找所有单位信息、产生各类消息的接口，并实现网络模块、所有单位、游戏场景之间的消息通信。

**重要成员变量：**

|  |  |  |
| --- | --- | --- |
| 变量名 | 类型 | 含义 |
| \_money | Money \* | 指向金钱的指针 |
| \_electricity | Electricity \* | 指向电力的指针 |
| \_msgGroup | GameMessageGroup \* | 指向游戏消息组的指针 |
| \_playerCamp | CampTypes | 玩家阵营 |
| \_gridMap | GridMap \* | 指向格点信息地图的指针 |
| \_tiledMap | TMXTiledMap \* | 指向瓦片地图的指针 |
| \_nextId | Int | 下一个生成单位的id |
| \_waitingGINum | Int | 等待生产的士兵个数 |
| \_waitingTankNum | Int | 等待生产的坦克的个数 |
| \_waitingAttackDogNum | Int | 等待生产的警犬的个数 |
| \_selectedUnitID | vector <int> | 玩家被选中的单位的id |
| \_barracksId | map<int, int> | 我方所有的兵营id及生产状态 |
| \_warFactoryId | map<int, int> | 我方所有的战车工厂id及生产状态 |
| \_producingFightUnit | map<vector<UnitTypes>::iterator, UnitTypes> |  |
| \_endFlag | map<CampTypes, bool> | 每个阵营的游戏状态 |
| \_fighterProduceSeq | vector<UnitTypes> | Fight单位生产序列 |
| \_unitIdMap | map<int, Unit \*> | 每个i单位id及其指针 |
| \_destinationMap | map<GridVec2, int> | 我方单位移动目的地及该单位id |
| \_socketClient | boost::shared\_ptr<talk\_to\_server> | 指向客户端对象指针 |
| \_socketServer | socket\_server \* | 指向服务端对象的指针 |

**重要成员函数：**

* + GridVec2 getUnitPos(int id); 通过单位id获取单位格点坐标
  + CampTypes getUnitCamp(int id); 通过单位id获取单位阵营
  + Unit \* getUnitPtr(int id); 通过单位id获取单位指针
  + void updateUnitState(); 把游戏消息传给客户端和接收客户端的消息并根据客户端的消息更新单位状态
  + void destoryUnit(int id); 销毁单位（所有信息）
  + void checkWinOrLose(); 判断玩家的输赢
  + Unit \* creatUnit(CampTypes camp, UnitTypes type, const GridVec2& pos, int id = 0); 生成一个单位并将其初始化
  + void creatProduceMessage(UnitTypes unitType, const GridVec2 & pos); 产生生产消息
  + void creatMoveMessage(int id, vector<GridVec2> &path); 产生移动消息
  + void createAttackMessage(int id1,int id2 ,int damage); 产生攻击消息
  + void selectUnits(const GridRect &range); 通过矩形范围选中单位
  + void choosePosOrUnit(const GridVec2 & pos); 通过格点选中单位或空地
  + void deselectAllUnits(); 取消所选的所有单位
  + friend GridRect transferRectToGridRect(const Rect & rect); 将像素矩形范围转化为格点矩形范围
  + void fighterUnitProductionUpdate(float fd); 士兵生产队列更新
  + Vec2 getMyBasePos(); 获得玩家基地坐标
  + void stopAllBuildingUnitUpdate(); 停止所有单位的生产（金币和fight单位）
  + void startAllBuildingUnitUpdate(); 启动所有单位的生产
  + void win(); 玩家赢了
  + void lose(); 玩家输了

**建筑类（Building）**

**Base类**

声明在Building.h中实现在Building.cpp中，公有继承自BuildingUnit类。

**Base类中的成员函数如下：**

* + static Base\* Base::create(const std::string& filename);Base类的构造函数
  + bool \_stdcall init( CampTypes camp, UnitTypes buildingType, GridVec2 point, TMXTiledMap\* map, GridMap \*gridmap, int id=0);用于初始化阵营类型，单位类型，单位在地图上的格点，单位的ID

**Class PowerPlant类**

声明在Building.h中实现在Building.cpp中，公有继承自BuildingUnit类。

**重要成员函数：**

* + static PowerPlant\*PowerPlant::create(const std::string& filename);PowerPlant类的构造函数

**Class OreRefinery类**

声明在Building.h中实现在Building.cpp中，公有继承自BuildingUnit类。

**重要成员函数：**

* + static OreRefinery \* OreRefinery::create(const std::string& filename); OreRefinery类的构造函数

**Class Barracks类**

声明在Building.h中实现在Building.cpp中，公有继承自BuildingUnit类。

**重要成员函数：**

* + GridVec2 findEmptyPosToProduce();寻找空地进行生产士兵
  + static Barracks \* Barracks::create(const std::string & filename);Barracks类的构造函数

**Class WarFactory类**

声明在Building.h中实现在Building.cpp中，公有继承自BuildingUnit类。

**重要成员函数：**

* + GridVec2 findEmptyPosToProduce();寻找空地进行生产坦克
  + static WarFactory \* WarFactory::create(const std::string &filename); WarFactory类的构造函数

**子弹类（Bullet）**

声明在Bullet.h中定义在Bullet.cpp中，公有继承自cocos2d::Sprite类。用来实现游戏场景中的子弹射击的效果。

**重要成员函数：**

* static Bullet\* create(const std::string& filename);构造函数
* bool initWithTarget(FightUnit\* sender, int target);为子弹初始化发射单位及攻击目标
* void setMap(TMXTiledMap \*map);
* void update(float dt);

**重要成员变量：**

|  |  |  |  |
| --- | --- | --- | --- |
| 变量名 | 变量类型 | 初值 | 作用 |
| \_tiledMap | TMXTiledMap \* | nullptr | 生成地图 |
| \_bullet | Sprite \* | 未初始化 | 生成子弹精灵 |
| \_target | Unit\* | nullptr | 生成单位 |
| \_speed | int | 未初始化 | 子弹速度 |
| \_targetId | int | 未初始化 | 子弹的目标 |
| \_sender | FightUnit \* | 未初始化 | 指向发射单位对象的指针 |
| \_unitManager | UnitManager \* | 未初始化 | 指向单位管理对象的指针 |

**攻击类（Fighter）**

**Soldier类**

定义在Fighter.h中，实现在Fighter.cpp中，公共继承自FightUnit类。

Soldier类是高级单位步兵，它没有什么特殊能力。

**Soldier类中的成员函数如下：**

* static Solider \*Solider::create(const std::string& filename); Soldier类的构造函数

**Tank类**

定义在Fighter.h中，实现在Fighter.cpp中，公共继承自FightUnit类。

Tank类是高级单位坦克，攻击距离更长，攻击力更大。

**Tank类中的成员函数如下：**

* static Tank\*Tank::create(const std::string& filename);Tank类的构造函数

**AttackDog类**

定义在Fighter.h中，实现在Fighter.cpp中，公共继承自FightUnit类。

AttackDog类是警犬单位，能攻击临近的士兵，但无法攻击坦克。

**AttackDog类中的成员函数如下：**

* static AttackDog\* AttackDog::create(const std::string &filename); AttackDog类的构造函数

**单位功能实现逻辑**

**建筑单位生产：**

在建筑图片上方点住鼠标左键

鼠标在地图上移动

释放鼠标左键

该位置是否可建

产生生产消息并发­送

所有客户端接收消息

产生建筑虚影，并初始化建筑对象进行建筑建造

建造进度条结束，变实

结束

否

是

**Fight类单位生产：**

Fight类单位的生产安排是在unitmanager对象中进行每帧的检测更新

结束

点击一下soldier图片

该单位加入所有fight类单位的生产序列

在unitManager类中遍历所有兵营和战车工厂

加入该兵营或战车工厂的生产中

若其state为0即为空闲状态

经过一定的生产时间后，兵营或战车工厂会搜寻临近空位，并产生生产消息

这部分为一次fight类单位检测更新的流程

否

是

全部客户端接收消息，产生单位

遍 历 结 束

**单位攻击：**

在Fight类单位的autoAttack()更新函数中

选择我方单位

结束

点击敌方单位

遍历我方单位

判断敌方单位是否在攻击范围内

是否收到攻击且处于能反击状态

以敌方单位位置为目的地，找到路径

产生移动消息，移动到目的地

move的回调函数

产生攻击消息并发送

客户端接收消息，并实现伤害和攻击效果

searchNearEnemy()，获取最近的敌人

是否处于自动攻击状态

设置攻击目标

有

startAttackUpdate()

判断该目标单位是否可以攻击

设置攻击目标

是

否

否

是

否

是

是

否

**从上方的流程图可以所示单位攻击分为两种：**

**1指定目标进行攻击**

选择我方单位，点击敌方单位，判断敌方单位是否在攻击范围内，是则我方被选中的fight类单位进行开始攻击更新，（在攻击更新函数调用刷新函数产生攻击消息，所有客户端接收消息，然后实现伤害和攻击效果），不则先以敌方单位位置为目的地，找到路径，产生移动消息，移动到目的地后，在move的回调函数中启动单位的攻击更新函

数。

**2.自动攻击**

Fight类单位在自动攻击开启的状态，searchNearEnemy()，获取最近的敌人，若有则启动单位的攻击更新函数，若无则结束

**网络**

**GameMessages消息格式：**

**我们使用的是proto3语法，所有字段的默认值都是optional类型的。**

**message GameMessage：**

|  |  |  |
| --- | --- | --- |
| 变量名 | 类型 | 含义 |
| Cmd | Enum{ CRT = 0;  MOV = 1;  ATK = 2; } | 消息的类型（生产、移动、攻击） |
| unit\_id1 | int32 | 单位id1 |
| unit\_id2 | int32 | 单位id2 |
| camp | int32 | 消息所属阵营 |
| unit\_type | int32 | unit\_id1单位的类型 |
| damage | int32 | unit\_id2单位收到的伤害 |
| position | GridPoint | 单位的坐标或是目的地 |
| path | GridPath | 单位移动格点路径 |

**message GridPoint：**

|  |  |  |
| --- | --- | --- |
| 变量名 | 类型 | 含义 |
| x | int32 | 格点x坐标 |
| y | int32 | 格点y坐标 |

**message GridPath：**

|  |  |  |
| --- | --- | --- |
| 变量名 | 类型 | 含义 |
| path\_point | repeated GridPoint | 路径上的每个格点 |

**message GameMessageGroup：**

|  |  |  |
| --- | --- | --- |
| 变量名 | 类型 | 含义 |
| game\_message | repeated GameMessage | 消息集中的每条消息 |

**消息类型：**

|  |  |
| --- | --- |
| 名称 | 含义 |
| CRT | camp阵营在position位置产生了类型为unit\_type ，id为unit\_id1的新单位 |
| MOV | unit\_id1单位的移动路径为path |
| ATK | 阵营camp的unit\_id1单位对unit\_id2单位造成了大小为damage的伤害 |

**GameMessages常用的重要函数：**

* inline void set\_camp(::google::protobuf::int32 value); 设置阵营
* inline void set\_cmd(::GameMessage\_Cmd value); 设置消息类型
* inline void set\_unit\_id1(::google::protobuf::int32 value); 设置单位id
* inline void set\_damage(::google::protobuf::int32 value); 设置伤害值
* inline ::GameMessage\*add\_game\_message()；向游戏消息集中加入新消息
* inline ::GridPoint\* mutable\_position(); 手动创建message position分量
* inline ::GridPath\*mutable\_path()； 手动创建message path 分量
* string SerializeAsString() const; 将消息集序列化为string
* inline bool ParseFromString(const string& data); 将消息集反序列化

**socket\_message类**

定义并实现在socket\_message.hpp中。为了实现在网络传输中发送任意长度的socket信息，采用了一个简单的协议来完成，即在要发送的信息前加入一个四个字节长度的消息头来标识消息的长度。

**socket\_message类中的成员变量如下：**

|  |  |  |
| --- | --- | --- |
| 变量名称 | 变量类型 | 变量含义 |
| data\_ | Char类型数组 | 存储消息 |
| Body\_length\_ | size\_t | 消息体长度 |

**socket\_message类中的成员函数如下：**

* const char\* data() const; 获取消息
* char\* data(); 获取可改变消息
* size\_t length() const;获得消息长度
* const char\* body() const;获得消息体
* char\* body(); 获得可改变消息体
* size\_t body\_length() const; 获得消息体长度
* void set\_body\_length(size\_t new\_length); 设置消息体长度
* bool decode\_header(); 解码消息头，获取消息长度
* void encode\_header(); 编码消息头；写入消息长度

**talk\_to\_client类**

定义在SocketServer.h中，实现在SocketServer.cpp中。talk\_to\_client类管理缓存区，并利用队列实现异步接受和同步发送。

**talk\_to\_client类中的成员变量如下：**

|  |  |  |
| --- | --- | --- |
| 变量名称 | 变量类型 | 变量含义 |
| socket\_ | boost::asio::ip::tcp::socket | 套接字 |
| socket\_server\_ | socket\_server \* | 服务端指针 |
| error\_flag\_ | bool | 错误标志 |
| read\_message | socket\_message | 从消息队列中取出的消息 |
| read\_message\_deque | std::deque<socket\_message> | 从客户端读取的消息队列 |
| mut\_ | boost::mutex | 互斥锁 |
| data\_conditioner\_ | boost::condition\_variable | 条件变量 |

**talk\_to\_client类中的成员函数如下：**

* static ptr create(io\_service & io\_service, socket\_server \* socket\_server); 创建talk\_to\_client对象，返回其指针
* void start(); 开始异步读取消息
* ip::tcp::socket & socket(); 返回套接字
* std::string read\_data(); 处理读取到的消息
* void write\_data(std::string s); 处理接收到的消息，并同步发送给客户端
* bool error();返回连接状况
* void do\_close(); 关闭连接
* void handle\_read\_header(const error\_code& error); 处理消息头
* void handle\_read\_body(const error\_code& error); 处理消息体
* talk\_to\_client(io\_service & io\_service, socket\_server \* socket\_server); 私有成员函数，创建talk\_to\_client对象
* void delete\_from\_server(); 从服务端删除连接

**talk\_to\_server类**

该类是enable\_shared\_from\_this 类，由共享指针控制;自定义在SocketClient.h中，在SocketClient.cpp中实现

作用：向服务端发送消息，从服务端接收消息。

**重要成员变量：**

|  |  |  |
| --- | --- | --- |
| 变量名 | 类型 | 含义 |
| error\_flag\_ | bool | 是否出现错误的标志 |
| start\_flag\_ | bool | 客户端是否异步连接的成功标志 |
| io\_service\_ | boost::asio::io\_service | 该对象是asio框架中的调度器，所有异步io事件都是通过它来分发处理的 |
| socket\_ | tcp::socket | 套接字，网络通信中不同主机之间的进程进行双向通信的端点，简单来说就是Ip address+ TCP + port |
| endpoint\_ | tcp::endpoint | 端点，使用该端口连接到一个服务器的地址 |
| read\_msg\_ | socket\_message | 从服务端读取的消息 |
| data\_cond\_ | std::condition\_variable | 条件变量对象，用于阻塞和唤醒线程 |
| mut | std::mutex | 互斥锁，能调用线程将锁住该互斥量或解锁 |
| read\_msg\_deque\_ | std::deque<socket\_message> | 从服务端获取的消息存在该队列中 |
| \*thread\_ | std::thread | 该客户端的线程（主要用于异步读取） |
| total\_ | Int | 总玩家数 |
| camp\_ | int | 该客户端的所属的玩家阵营 |

**重要成员函数：**

* static socket\_client\_ptr start(std::string ip= "127.0.0.1", int port= 8008); 被调用将创建一个新的客户端，并为该客户端分配新的线程以及调用异步连接函数
* talk\_to\_server(boost::asio::ip::tcp::endpoint ep); 客户端的构造函数，初始化ip address和port
* void start(boost::asio::ip::tcp::endpoint ep); 开始异步连接
* void on\_connect(const error\_code & err); 连接成功的回调函数，从服务端获取阵营
* void handle\_read\_header(const error\_code& error); 异步读取及处理从服务端读取到的socketMessage的头部信息
* void handle\_read\_body(const error\_code& error); 异步读取及处理从服务端读取到的socketMessage的主体部分信息
* void do\_write\_string(std::string s); 向服务端同步发送消息
* std::string get\_string\_from\_server(); 将read\_msg\_deque\_的消息转为string并返回
* void do\_close(); 关闭该客户端及线程
* int getCamp()const; 获取玩家阵营
* int getTotalPlayer()const ; 获取玩家总数

**实现**

在游戏的GameMeunScene中调用static socket\_client\_ptr start()函数，该函数中设置了服务端连接端点endpoint\_，创建了客户端的实例，并用share\_ptr来管理，分配一个新的线程，并使\_io\_service::run()能够调用该线程，接着启动异步连接，async\_connect异步连接到endpoint\_地址，成功后回调on\_connect()函数，在该函数中使用read\_some获取玩家的camp和 total，使用异步读取，读取完成的回调函数handle\_read\_header，获取此次消息的主体长度，然后调用异步读取获取body，完成处理方法为handle\_read\_body，存储进read\_msg\_deque\_中，再次启动异步读取header，如此循环。

GameMeunScene中调用静态的start()函数

Do\_close()

设置连接端点endpoint\_，分配新线程，并使\_io\_service::run()能够调用该线程

async\_connect（）

read\_some()获取玩家的camp和 total

async\_read()读取消息头

handle\_read\_header(),获得消息长度，异步读取body

handle\_read\_body()，将消息存储进read\_msg\_deque\_中

on\_connect中判断是否error

Error

Error

**发送**

采用asio中

std::size\_t write(SyncWriteStream& s, const  ConstBufferSequence& buffers) 函数。此函数是包装后的socket发送，可以保证发送的终能成功到达接收端，同时阻塞发送，使得在大数据的情况下不会有发送延时问题导致的错误接收顺序。

**接收**

在异步接收时，将一次异步接收分为两次实现，首先接收前4个字节，读出整个消息长度，然后再读整个消息的内 容，从而实现变长消息的接收。而在一个完整消息读取完毕后，将所读消息存入消息队列中，供外部读取所用。

在完成读取和接收消息的同步时，条件变量方式，总而避免了多线程之间错误的锁定导致的死锁。

**游戏消息**

在TjRedAlert中使用Google Protocol Buﬀer（简称Protobuf）作为我们的游戏消息，在我们编写完GameMessages.proto后，通过Protobuf工具可以生成GameMessages.pb.h 和 GameMessages.pb.cc,将这两个文件直接包含在项目库中即可使用。

**socket\_server类**

socket\_server类定义在SocketServer.h中，实现在SocketServer.cpp中。实现了异步连接、连接管理以及分连接的的收发。

**socket\_server类中的成员变量如下：**

|  |  |  |
| --- | --- | --- |
| 变量名称 | 变量类型 | 变量含义 |
| acceptor\_ | boost::asio::ip::tcp::acceptor | 异步连接客户端 |
| connections\_ | std::vector<talk\_to\_client::ptr> | 已连接客户端指针 |
| connection\_num\_ | int | 已连接客户端数量 |
| io\_service\_ | static io\_service\* | 调度器，处理分发io事件 |
| thread\_ | boost::thread \* | 异步接收线程 |
| button\_thread\_ | boost::thread \* | 同步发送线程 |
| delete\_mutex\_ | std::mutex | 互斥锁 |
| error\_flag\_ | bool | 错误标志，代表服务端连接状态 |

**socket\_server类中的成员函数如下：**

* static socket\_server \* create(int port = 8008); 创建服务端
* void close(); 关闭服务端
* std::vector<talk\_to\_client::ptr> get\_connection() const; 获取已连接客户端指针数组
* void remove\_connection(talk\_to\_client::ptr p); 移除已连接客户端p
* void button\_start(); client连接后，向client发送开始消息并开始游戏
* bool error() const; 返回server连接状况
* int connection\_num() const; 返回已连接客户端数量
* socket\_server(int port); 私有构造函数
* void start\_accept(); 开始异步连接客户端
* void handle\_accept(talk\_to\_client::ptr new\_connection, consttalk\_to\_client::error\_code& error);处理新连接的客户端，将其加入connections\_
* void loop\_process(); 循环同步发送

**寻路**

**Grid类**

定义在PathFinder.h中，实现在PathFinder.cpp中。表示寻路算法中的一个网格，包含各种寻路算法中要用到的信息。

**Grid类中的成员变量如下：**

|  |  |  |
| --- | --- | --- |
| 变量名称 | 变量类型 | 变量含义 |
| \_status | int | 格点状态 |
| \_xPosition | int | 格点X坐标 |
| \_yPosition | int | 格点Y坐标 |
| \_GValue | int | G值，欧几里得距离 |
| \_HValue | int | H值，曼哈顿距离 |
| \_FValue | int | F值，寻路网格评估总值 |
| \_parentGrid | Grid \* | 父节点，寻路算法中的上一步 |

**Grid类中的成员函数如下：**

* int getStatus(); 获取网格状态
* void setStatus(int status); 设置网格状态
* int getXPosition(); 获取网格X坐标
* void setXPosition(int xPosition); 设置网格X坐标
* int getYPosition(); 获取网格Y坐标
* void setYPosition(int xPosition); 设置网格Y坐标
* int getGValue(); 获得G值
* void setGValue(int GValue); 设置G值
* int getHValue(); 获得H值
* void setHValue(int HValue); 设置H值
* int getFValue(); 获得F值
* void setFValue(int FValue); 设置F值
* Grid\* getParent(); 获取父节点
* void setParent(Grid \* parentGrid); 获取父节点

**PathFinder类**

定义在PathFinder.h中，实现在PathFinder.cpp中。PathFinder类是寻路算法的包装类。

**PathFinder类的成员变量如下：**

|  |  |  |
| --- | --- | --- |
| 变量名称 | 变量类型 | 变量含义 |
| \_mapWidth | int | 地图宽度 |
| \_mapHeight | int | 地图高度 |
| \_start | Grid \* | 寻路起点 |
| \_end | Grid \* | 寻路中点 |
| \_openList | std::vector<Grid \*> | 开放列表，待检测格点 |
| \_closeList | std::vector<Grid \*> | 关闭列表，不需检测的格点 |
| \_path | std::vector<GridVec2> | 路径 |
| \_mapGrid | std::vector<std::vector<Grid>> | 网格地图 |

**PathFinder类的成员函数如下：**

* PathFinder(std::vector<std::vector<int>> & barrierMap, int x1, int y1, int x2, int y2);

构造函数，创建PathFinder对象

* void searchPath(); 寻路算法实现，设置网格地图上的路径信息
* void generatePath(); 回溯获得路径
* std::vector<GridVec2> getPath(); 获取路径
* Grid\* selectNextGrid(); 选择下一个待检测格点
* bool isInOpenList(Grid & grid); 判断格点是否在OpenList中
* bool isInCloseList(Grid & grid); 判断格点是否在CloseList中
* void removeFromOpenList(Grid \* grid); 从OpenList中移除某格点
* int calculateEuclideanDistance(Grid & grid1, Grid & grid2); 计算欧几里得距离
* int calculateManhattanDistance(Grid & grid1, Grid & grid2); 计算曼哈顿距离
* bool isAvailable(Grid & grid); 判断格点是否可到达
* bool checkCorner(Grid & grid1, Grid & grid2); 检测单位是否处于拐角处
* void checkSurroundGrid(Grid & grid); 在周围的格点中寻找下一步

**A\*寻路算法**

**实现原理**

1. 从起点开始找它周围可以走的格子，算出可走格子中F值最小的格子，再就以这个格子作为新的中心点，又同样找其周围可以走的格子，以此类推，直至找到目的点。
2. 确定最优格子是取F值最小为依据，那么F值怎么算？F = G + H，G：表示从起点移动到指定格子的耗费(一般横竖着的格子耗费为10(or 1)，斜着的格子耗费为14(or 1.4)，等于其父节点G + 自己的G的和)。H：表示从指定格子移动到终点的预计耗费(一般就是指定点到终点相差的X,Y绝对值的和)。
3. 每次找的周围格子都会放入到“开放列表”中，如果下次周围点某几个“开放列表”中已经存在，是否修改这些存在的点属性，是看新计算的G值，如果G值 > 原来的G值，则什么也不做，否则，就以新的G值替换，同时改变其父节点。
4. 每个格子都以其中心点作为父节点，这样链接起来，就像一个链表结构，这样直到找到目的点后，就可以通过目的点的父节点这样一级级还原出路径了。
5. 每个格子的属性：x坐标，y坐标，G值，H值，F值(F=G + H)，父格子。

**寻路步骤**

1，从起点A开始, 把它作为待处理的方格存入一个OpenList，OpenList就是一个等待检查方格的列表.

2，寻找起点A周围可以到达的方格, 将它们放入OpenList, 并设置它们的”父方格”为A.

3，从OpenList中删除起点 A, 并将起点 A 加入CloseList, CloseList中存放不需要再次检查的方格.

4， 从 OpenList 中选择 F 值最低的方格C.

5，把它从 OpenList 中删除, 并放到CloseList 中.

6，检查它所有相邻并且可以到达 (障碍物和 “关闭列表” 的方格都不考虑) 的方格. 如果这些方格还不在 OpenList里的话, 将它们加入OpenList, 计算这些方格的 G, H 和 F 值各是多少, 并设置它们的 “父方格”为C .

7， 如果某个相邻方格 D 已经在 OpenList 里了, 检查如果用新的路径 (就是经过C 的路径) 到达它的话, G值是否会更低一些, 如果新的G值更低, 那就把它的 “父方格” 改为目前选中的方格 C, 然后重新计算它的 F 值和 G 值 (H 值不需要重新计算, 因为对于每个方块, H 值是不变的). 如果新的 G 值比较高, 就说明经过 C 再到达 D 不是一个明智的选择, 因为它需要更远的路, 这时我们什么也不做.

8，然后继续找F值最小的，如此循环下去…

