TJ Red Alert 需求与设计报告

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：

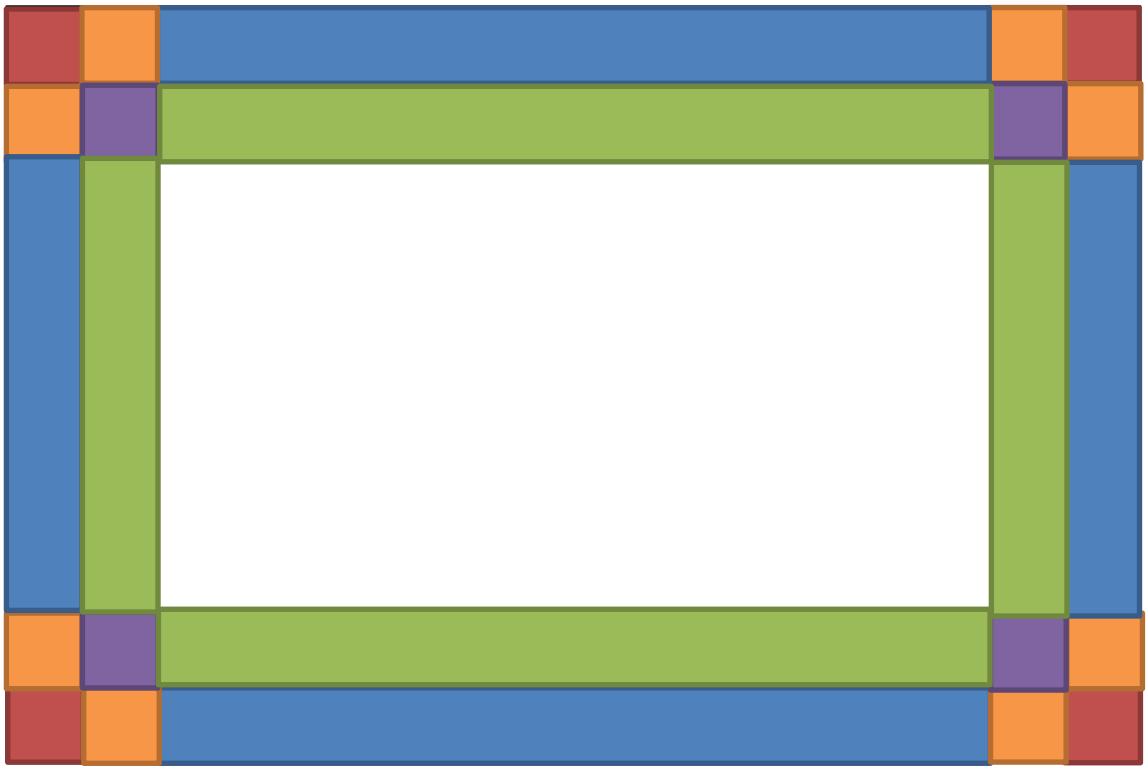
变量名	变量类型	变量含义
_buildingButton	MenuItemImage *	建筑建造按钮
_armyButton	MenuItemImage *	战斗单位建造按钮

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。

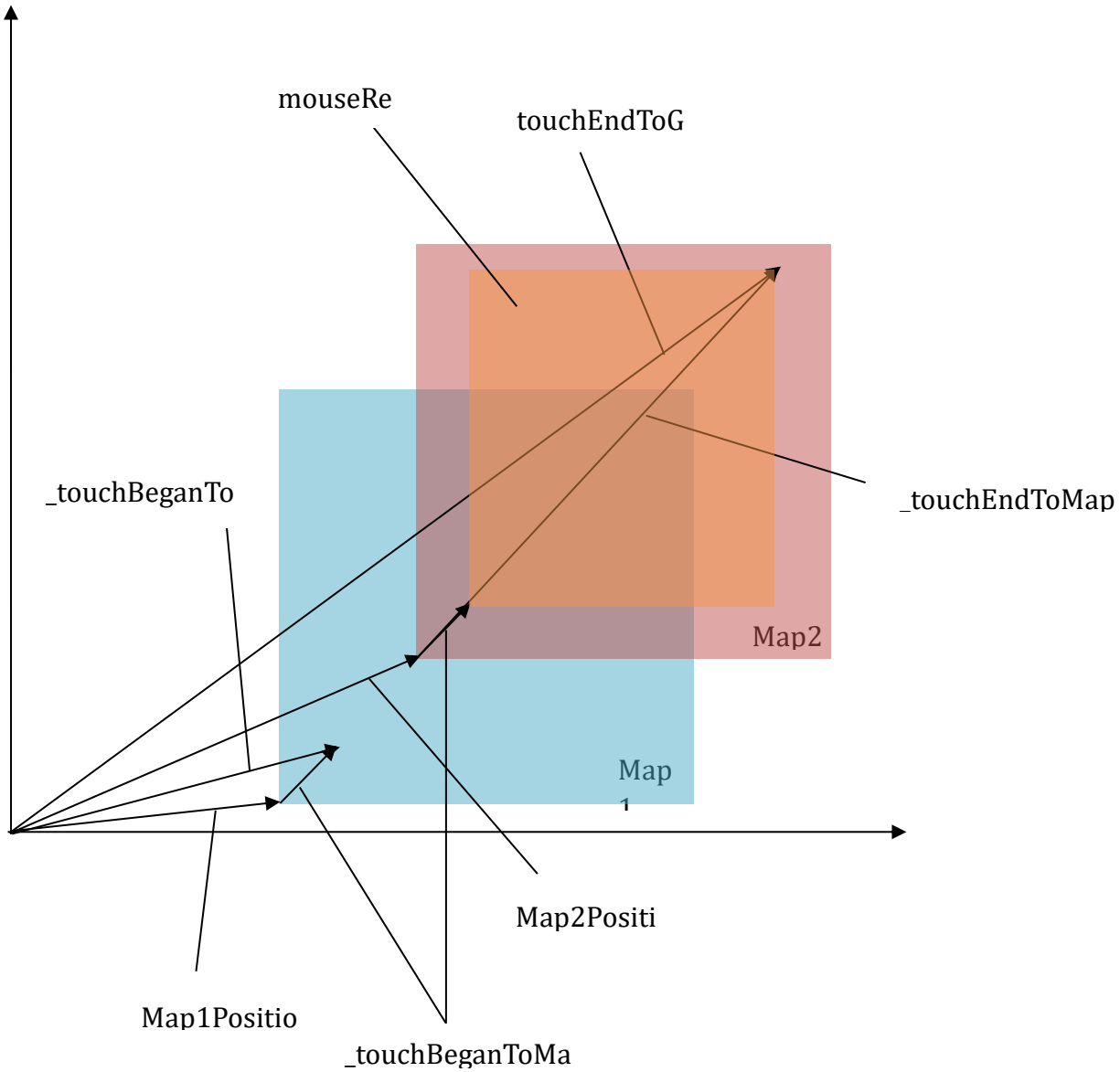
键盘按下时，`void onKeyPressed(KeyCode keyCode, Event * event);`响应键盘事件执行对应操作，`void keyPressedToMove(KeyCode keyCode);`在 update 函数中检测被按下按键，向对应的方向移动，`void onKeyReleased(KeyCode keyCode, Event * event);`在按键松开时，设置按键状态。

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;



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.。

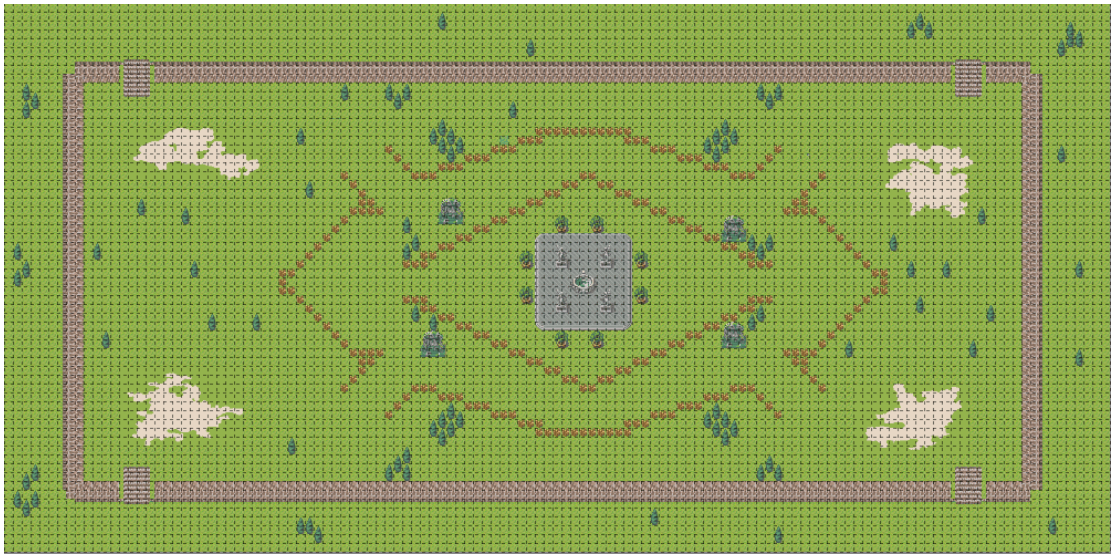
重要成员变量：

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

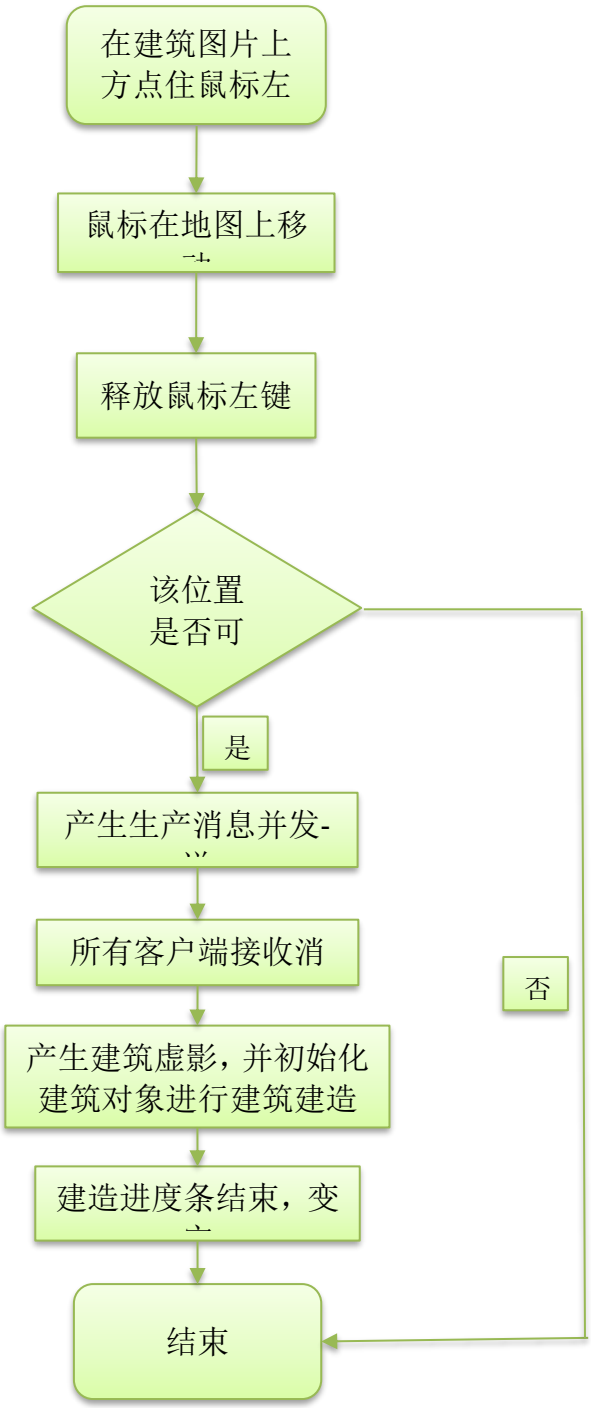
AttackDog 类

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

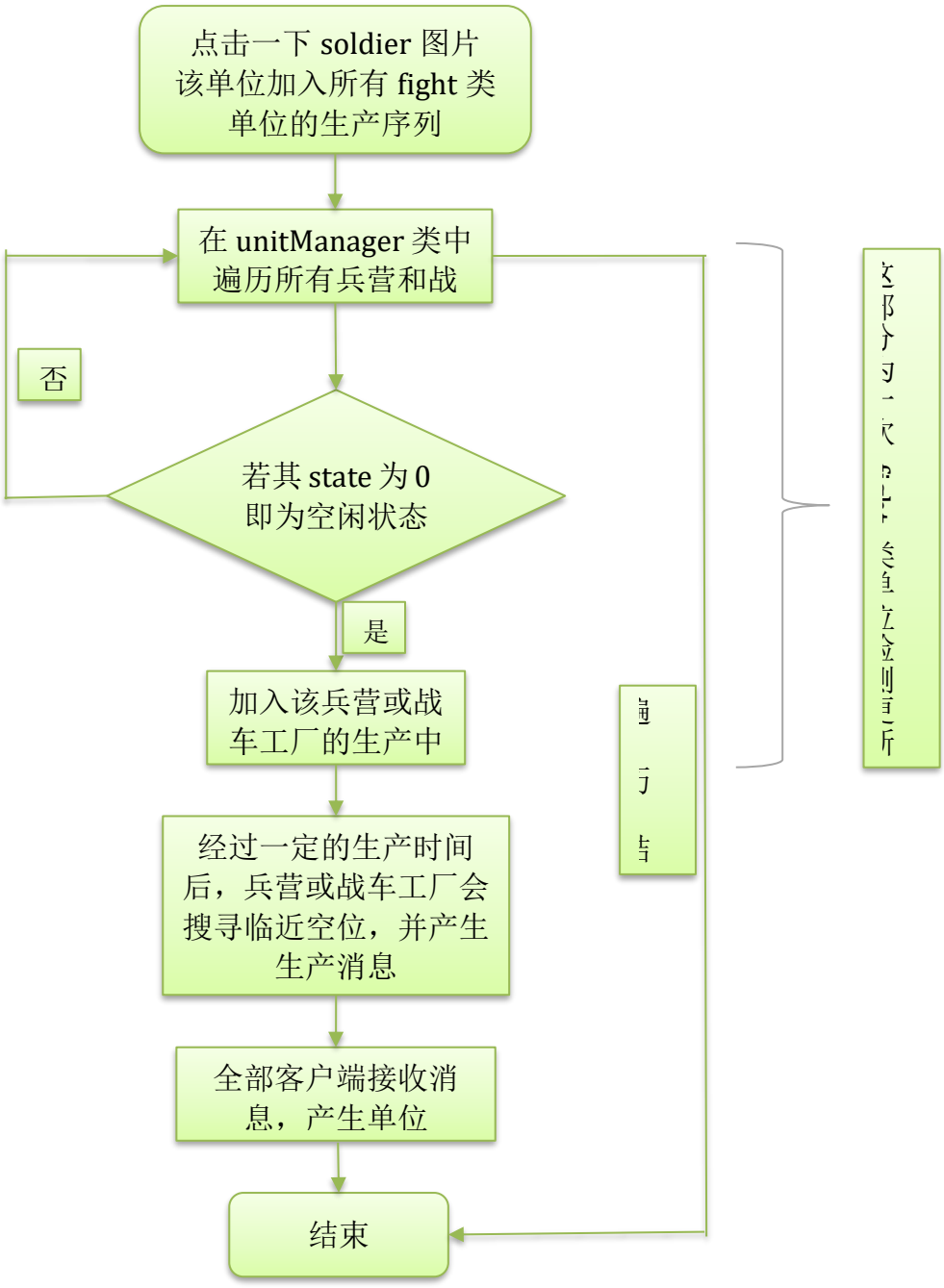
单位功能实现逻辑

建筑单位生产：

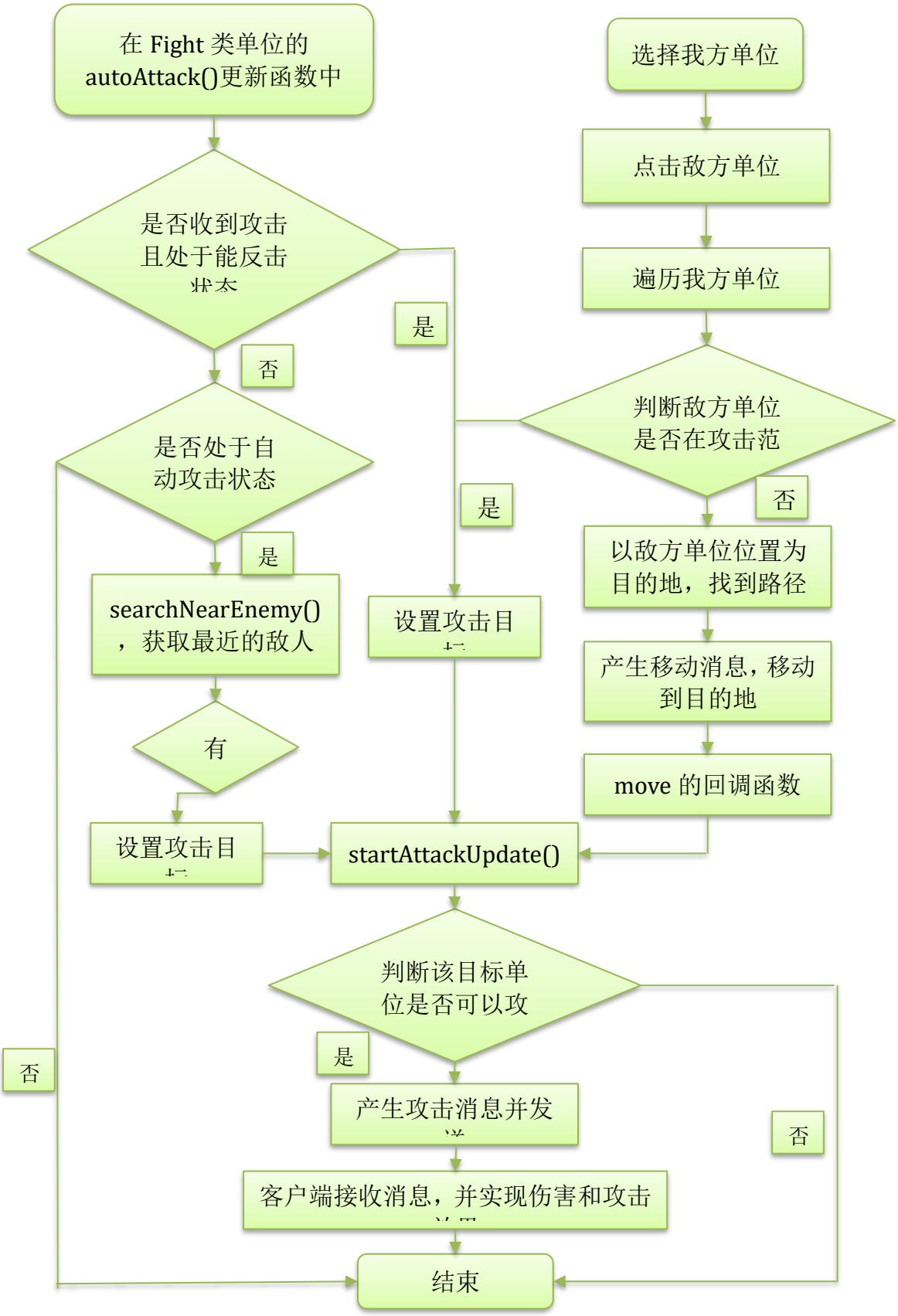


Fight 类单位生产：

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



单位攻击：


在 Fight 类单位的 autoAttack()更新函数中
是否收到攻击且处于能反击状态
是
否
是否处于自动攻击状态
是
searchNearEnemy()，获取最近的敌人
有
设置攻击目标
否
选择我方单位
点击敌方单位
遍历我方单位
判断敌方单位是否在攻击范
否
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
是
设置攻击目标
startAttackUpdate()
判断该目标单位是否可以攻
是
否
产生攻击消息并发送
客户端接收消息，并实现伤害和攻击
否
结束

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

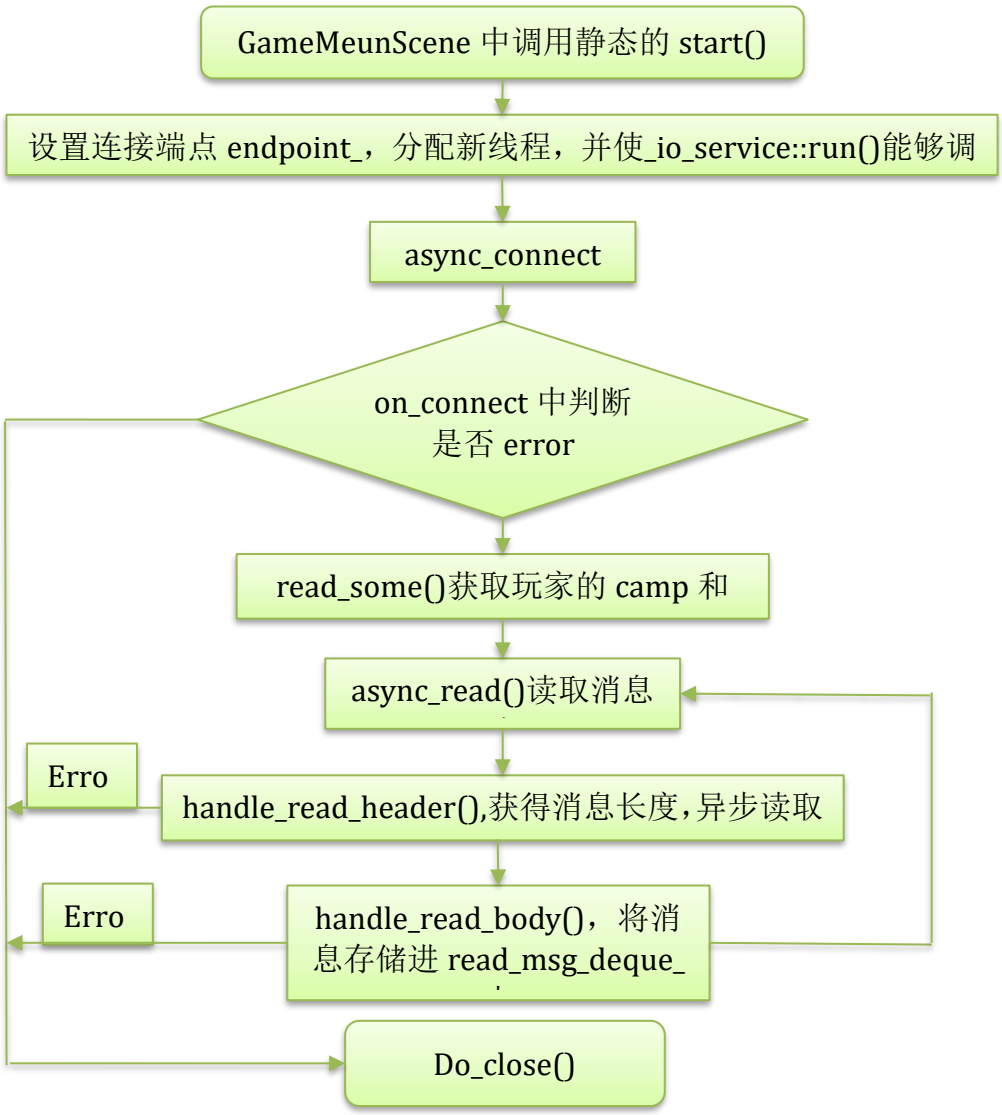
endpoint_	tcp::endpoint	端点，使用该端口连接到一个服务器的地址
read_msg_	socket_message	从服务端读取的消息
data_cond_	std::condition_variable	条件变量对象，用于阻塞和唤醒线程
mut	std::mutex	互斥锁，能调用线程将锁住该互斥量或解锁
read_msg_deque_	std::deque<socket_message>	从服务端获取的消息存在该队列中
***thread_**	std::thread	该客户端的线程（主要用于异步读取）
total_	Int	总玩家数
camp_	int	该客户端的所属的玩家阵营

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。



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...