

1. GameController

- **继承自:** `QWidget`
- **功能:** 管理游戏资源。显示主菜单界面、显示关卡选择界面、加载关卡、暂停/恢复游戏、退出游戏、控制玩家生命与金币. 控制游戏内背景音乐, 难度等级.
- **主要成员变量:**
 - `currentLevel` : 当前待加载关卡 (平时就存放最大关卡)
 - `maxLevel` : 最大解锁关卡
 - `hardLevel` : 游戏难度. 此量需要
 - `volumn` : 游戏音量等级
 - `gameBgm` : 游戏内背景音乐
 - `mainMenu` : 主菜单界面
 - `levelSelectMenu` : 关卡选择界面
 - `settingsMenu` : 设置界面
 - `gameScene` : 游戏场景界面
- **主要方法:**
 - **构造函数与析构函数** : 需要创建一个存档,或者从存档中读取一些基本数值.
`maxLevel`;`hardLevel`;`volumn`;`gameBgm`等都需要存放.
 - `loadMap(int level)` : 加载地图, 需要调用地图的加载关卡功能, 并且场景内绘制地图
- **槽:**
 - `startGame()` : 开始游戏. 在点击 `GameScene` 下的开始游戏按钮之后会开始游戏, 新建一个新的 `GameScene`, 然后同时启动各个定时器. (默认从目前解锁的最大关卡开始)
 - `endGame()` : 结束游戏. 点击 `GameScene` 下的关闭游戏按钮后, 会关闭 `GameScene`. 然后回到主菜单界面.
 - `exitGame()` : 退出游戏, 退出整个游戏. (包括主菜单)
 - `showMainMenu()` : 显示主菜单页面
 - `showSettingMenu()` : 显示设置页面
 - `showLevelSelectMenu()` : 显示关卡选择菜单页面
 - `loadLevel(int level)` : 加载指定关卡 (实现上可以将 `currentLevel` 更新为 `level`, 然后调用 `startGame`. 记得在退出游戏之后恢复到 `maxLevel`)
 - `onEnemyArrived(int damage)`

2. MainMenu

- **继承自:** `QWidget`

- **功能:** 显示主菜单界面, 提供开始游戏、选择关卡、设置和退出等功能。
- **主要成员变量:**
 - `startButton` : 开始游戏按钮
 - `levelSelectButton` : 选择关卡按钮
 - `settingsButton` : 设置按钮
 - `exitButton` : 退出按钮
- **主要方法:**
 - `show()` : 显示主菜单
 - `hide()` : 隐藏主菜单
- **信号:**
 - `startNewGame` : 开始一局新游戏
 - `openLevelMenu` : 打开菜单界面
 - `openSettingMenu` : 打开设置页面
 - `exitGame` : 退出游戏
- **槽:**
 - `onStartButtonClicked` : 用户点击了开始游戏按键
 - `onLevelSelectButtonClicked` : 类似上面
 - `onSettingButtonClicked`
 - `onExitButtonClicked`

3. LevelSelectMenu

- **继承自:** `QWidget`
- **功能:** 显示关卡选择界面, 允许玩家选择不同的关卡。注意未解锁的关卡不可进入, 在 UI 上要显示锁起来, 用户点击就没有反应, 不能返回对应的 level 值。
- **主要成员变量:**
 - `levelButtons` : 关卡按钮列表
- **主要方法:**
 - `show()` : 显示关卡选择菜单
 - `hide()` : 隐藏关卡选择菜单
- **信号:**
 - `selectLevel(int level)` : 发送关卡信息给 `gameController`
- **槽:**
 - `onLevelButtonClicked(int level)` : 用户点击已解锁的关卡按钮

4. GameScene

- **继承自:** QGraphicsView
- **功能:** 显示游戏场景, 管理游戏中的所有图形元素. 管理玩家的血量与金钱. 需要实现放置塔的功能. 管理 towers, enemies, projectiles. 需要实现按一定的规律放置新的敌人的功能. 需要实现游戏的暂停与继续功能. 需要实时显示血量与金钱 (货币叫做 shilling 先令). 需要管理好各个元素, 实现添加, 删除, 并且链接槽与信号.
- **主要成员变量:**
 - `player`: 玩家. 控制其金钱和 health 的花费. 每关开始血量回满, 钱币归到一个初始数值.
 - `map`: 地图信息. 从中获取各类关卡信息.
 - `pauseGameButton`: 暂停按钮
 - `resumeGameButton`: 继续按钮
 - `scene`: QGraphicsScene 对象, 管理游戏中的图形元素
 - `healthTextItem`: QGraphicsTextItem 对象, 用于显示血量文本. 要求实时更新
 - `moneyTextItem`: QGraphicsTextItem 对象, 用于显示血量文本. 要求实时更新
 - `towers`: 防御塔列表
 - `enemies`: 敌人列表
 - `obstacles`: 障碍物列表
 - `towerSelectMenu` 的菜单, 选择里面的按钮会触发对应的信号
 - `pausedMenu` 的界面, 会使得玩家无法点击原界面, 并且这个界面里有 `resumeButton`. 效果如下图. 也可见 UI 文档中的简图



- **主要方法:**
 - `addTower(Tower*)`: 添加防御塔. 在鼠标点击可放置位置之后 (这个判断直接调用 `map` 的函数), 在点击位置对应的方格上先弹出选择防御塔的小窗(不够买的显示成灰色, 可以买的再显示成正常颜色). 然后用户再通过点击对应的塔的按钮来放置防御塔. 放置好以后, 将塔添加到 `towers` 的列表. 并且花费对应的钱.

- `addEnemy()` : 添加敌人. 每隔一段时间(比如说 1.5 秒)调用一下这个函数, 从 map 中获取待添加的敌人类型(0 表示不添加敌人,-1 表示没有新的敌人, 游戏胜利), 然后添加敌人, 同时加入到敌人列表中, 并且给所有可以攻击到此敌人的塔设置 `setTarget`, 让他们可以攻击.
- `addObstacles()` : 添加障碍物. 从 map 中获取障碍物列表, 然后添加到障碍物列表中.
- `updateScene()` : 更新场景. 重绘. 每 50 ms 重绘一次 (20 帧).
- `pauseScene()` : 暂停游戏 (实现可以调用每一个 tower 和 enemy 的暂停函数)
- `mousePressEvent` : 重写的鼠标事件, 用于捕捉用户鼠标点击位置. 主要用在放置新的防御塔上. (点击防御塔进行拆除或者升级的鼠标事件, 在 Tower 中来处理, GameScene 不用研究此鼠标事件. 但是需要处理好塔升级完, 或者拆除之后对应 towers 列表的管理与图像的重绘.)
- **信号:**
 - `gameEnd()` : 由槽函数发出的信号, 发送给 GameController, 然后关闭 GameScene.
- **槽:**
 - `onPauseButtonClicked()` 按下暂停按钮之后反馈的槽函数. 它将调用每一个对象 (塔, 敌人, 投掷物) 的暂停函数. 同时显示出继续的按钮, 并且创造出一个新的界面, 防止用户继续访问 GameScene, 并且将 GameScene 的背景虚化.
 - `onResumeButtonClicked()` 按下继续的按钮之后的槽函数. 它将调用每一个对象的继续函数. 同时关闭暂停而创造的页面
 - `onGameEndButtonClicked()` : 按下结束游戏按钮之后的槽函数. 发出信号. 并且对所有对象做析构.
 - `onTowerSelectButtonClicked(int)`
 - `onTowerUpdated(int)` : 扣除 int 对应的钱量, 在防御塔列表中升级此防御塔.
 - `onDeleteTowerButtonClicked(int)` : 返还对应的钱币量. 并且从列表中去除此防御塔.
 - `updatePlayerLives(int lives)` : 更新玩家血量的标签 (它与 *player* 的血量改变信号相连接, 连接要在 GameController 中进行)
 - `onEnemyDead(int reward)` : 获得钱币. 从敌人列表删除敌人
 - `onEnemyArrived(int damage)` : 敌人到达, 给玩家带来伤害. 并且从敌人列表删除敌人.
 - `onObstacleDestoried(int)` : 障碍物被摧毁, 给玩家金币.

5. SettingsMenu

- **继承自:** `QWidget`
- **功能:** 显示设置界面, 允许玩家调整游戏参数.
- **主要成员变量:**
 - `volumeSlider` : 音量调节滑块
 - `difficultyComboBox` : 难度选择下拉框

- **主要方法:**
 - `show()` : 显示设置菜单
 - `hide()` : 隐藏设置菜单
 - `applySettings()` : 应用设置
- **信号:**
 - `volumeChanged(int volume)` : 发出调整音量的信号
 - `DifficultyComboBoxChanged(int index)` : 发出挑战游戏难度的信号
 - `gameBgmChanged(int index)` : 发出修改 `gameBgm` 的信号
- **槽:**
 - `onVolumeSliderChanged(int value)` : 检测用户改变音量的操作
 - `onDifficultyComboBoxChanged(int index)` : 检测用户改变难度的操作
 - `onGameBgmChanged(int index)` : 检测用户改变 bgm 的操作

6. Player

- **功能:** 定义玩家的状态. 改变玩家的状态. 判断游戏是否结束. **注意这里面的东西只能直接在 GameScene 中访问.**
- **主要成员变量:**
 - `shilling` : 玩家拥有的金币数, 名称叫先令 (中世纪英国和神圣罗马等国家使用的货币)
 - `health` : 玩家剩余的生命数.
- **主要方法:**
 - **构造函数** : 传递生命值和金币两个数值. 作为初始生命值与金币
 - `spendMoney(int amount)` : 消耗金币
 - `earnMoney(int amount)` : 获得金币
 - `loseLife()` : 玩家失去生命 (顺便判断玩家是否失去所有生命)
- **信号:**
 - `gameOver()` : 失去所有生命值, 本局游戏失败, 自动重启.

7. Tower (防御塔)

- **功能:** 定义不同的防御塔类型. 以它作为父类可以生成至少三种子类防御塔. 所以有些函数可以是虚函数, 是 `protected`. 不需要调用基类, 调用它的子类就行. 希望全局控制者能维护一个敌人队列, 每次从队列中取出一个敌人给防御塔攻击. 防御塔会完成范围内敌人检测与发射子弹的功能, 不需要别的类来完成. 防御塔类也重写了自己的鼠标事件, 包括了升级和删除的界面弹出.
- **主要成员变量:**
 - `towerSize` : 防御塔本体的大小(静态)

- `int projectType` //投掷物的种类
- `int towertype` //防御塔的种类
- `int level` //防御塔现在的等级
- `int attackRange` :攻击范围
- `int attackSpeed` :攻击速度
- `int buyCost` :购买花费
- `int sellPrice` :出售价格
- `QGraphicsItem* target` :攻击的敌人
- `QString picDir` :图片位置
- `QVector<int> upgradeFee` :升级费用列表 (包括第一级, 第二级...)
- `QVector<Projectile*> projectileList` :投掷物列表
- `QPointF TowerCentral` ; //相对于场景的坐标

• 主要方法:

- `explicit TowerFrame(QPoint pos_=QPoint(0,0),int type=0)` ;防御塔基类构造函数
- `virtual void attack()=0` ; 防御塔攻击被添加至敌人列表中的敌人(不需要手动调用)
- `void findEnemy()` ; 防御塔自动瞄准敌人(不需要手动调用).
- `int getBuyCost(){return buy_cost;}` ;获得购买的价格
- `int getSellPrice(){return sellPrice;}` ;获得出售的价格
- `int getUpdateCost()` :获得升级需要的价格
- `void setTarget(QGraphicsItem* target_out=nullptr)` ;给防御塔设置敌人
- `void resetTarget()` ;(不需要手动调用,已经自动实现)如果敌人死了,调用这个方法把这个防御塔和投掷物的敌人置空
- `void paint(QPainter * painterconst,const QStyleOptionGraphicsItem *option, QWidget *widget)override` ;//画出防御塔
- `QRectF boundingRect() const override` ;
- `void mousePressEvent(QGraphicsSceneMouseEvent *event) override` ;
- `void contextMenuEvent(QGraphicsSceneContextMenuEvent *event) override` ;右键防御塔会出现升级和出售

• public slots:

- `virtual void upgrade()=0` ; //升级植物,界面设计者要根据现有的钱和updatecost比较获得是否可行,别忘了扣钱; 升级是一级一级来的, 升完范围和伤害, 才能升特殊能力.
- `void sell()` ; //出售植物并将植物删除`

8. Enemy (敌人)

- **功能:** 定义敌人的属性和行为。有不同的子类, 有不同的血量, 速度和伤害与外形. 敌人正常就沿着路径移动. 敌人可能会受到 debuff, 比如燃烧与减速. 敌人受到的伤害会受其类型的不同而不同 (具体敌人有哪几类请看项目文档里游戏内容设计部分). 敌人要检测自己碰撞范围内是否有子弹, 然后进行伤害判定. 敌人走到目标位置以后会对目标造成伤害, 然后将自己消亡.
- **主要成员变量:**
 - `health` : 生命值
 - `speed` : 移动速度
 - `damage` : 对萝卜的伤害
 - `reward` : 击败后给玩家的奖励 (金币)
 - `routine` : 敌人的移动路径 (可以使用一个由 tuple 组成的数组来存放)
 - `enemyType` : 敌人的类型, 人类单位还是异鬼单位.
- **主要方法:**
 - `Enemy(vector<<int>, <int>>>)` 构造函数要传入初位置与路径
 - `move()` : 移动到下一个位置
 - `takeDamage(int damage)` : 接受伤害, 检查是否死亡, 死亡则触发信号. 实现上, 在自己的 `rect` 碰撞范围内检测 Item, 如果是子弹, 用 `getType()` 判断是否是龙焰类型 (具体龙焰采用什么数字, 后续待定), 如果是龙焰, 敌人受到伤害. 如果不是, 除了敌人受到伤害以外, 还会将这个子弹直接消亡.
- **信号:**
 - `isDead(int reward)` : 在被打死以后触发; 如果是到达终点, 不触发
 - `isArrived(int damage)` : 检查敌人是否到终点

示例代码:

检测自己范围内的敌人的示例代码

```
void Tower_frame::CheckForItemsInBoundingRect() {
    // 获取当前项的 boundingRect, 并将其转换为场景坐标
    QRectF sceneBoundingRect = mapRectToScene(boundingRect());

    // 获取在该区域内的所有项
    QList<QGraphicsItem*> itemsInBoundingRect = scene()-
>items(sceneBoundingRect);

    // 移除自身 (当前塔) 避免自检测
    itemsInBoundingRect.removeOne(this);

    if (!itemsInBoundingRect.isEmpty()) {
        qDebug() << "Found items within boundingRect!";
        for (auto* item : itemsInBoundingRect) {
            qDebug() << "Item at:" << item->pos();
        }
    }
}
```



```

    }
} else {
    qDebug() << "No items found within boundingRect.";
}
}

```

9. Projectile (投掷物)

- **功能:** 定义防御塔发射的子弹或魔法攻击。有不同的子类. 发射向敌人, 除了龙焰以外的子弹, 碰到敌人会被析构, 同时造成伤害. 龙焰具有穿透性.
- **主要成员变量:**
 - `int speed;` :子弹的速度
 - `int damage;` :子弹的伤害
 - `QString src;` :投掷物的图片
 - `QGraphicsItem* enemys;` :子弹的目标
 - `QPointF delta;` :子弹位移的偏移量
 - `QTimer* moveTimer ;` :子弹移动的计时器
 - `QPointF towerCor;` :子弹所属防御塔中心坐标
 - `qreal tattackRange;` :塔的攻击范围
- **主要方法:**
 - `explicit Projectile(QPointF pos,QPointF Tower_c,qreal attack_range);`
 - `void setTarget(QGraphicsItem* Enemy=nullptr);` :设置子弹的攻击敌人
 - `void moveToEneny();` :自动向敌人移动
 - `void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget)override;`
 - `QRectF boundingRect() const override;`
 - `static qreal pix_size;` :子弹的大小
 - `void outOfRange();` :删除超过攻击范围的子弹
 - `void destroy();` :子弹销毁的信号
 - `int getDamage();` :获得子弹的伤害.
 - `int getType();` :获得子弹的类型

投掷物需要大量生成与析构, 所以建议使用对象池实现
示例代码:

```

#ifndef MYOBJECTPOOL_H
#define MYOBJECTPOOL_H

```



```

#include <QSharedPointer>
#include <QList>

template <typename T>
class MyObjectPool
{
public:
    MyObjectPool();
    explicit MyObjectPool(int initNum); // 注意，这个初始化函数也会增加对象池的大
    小!
    ~MyObjectPool();

    QSharedPointer<T> acquire();
    void release(QSharedPointer<T>& object);

    int get_pool_size(){ return pool_size;}

private:
    QList<QSharedPointer<T>> pool;
    int pool_size; // 这个是对象池的最大大小

    void expandPool(int expSize);
};

template<typename T>
MyObjectPool<T>::MyObjectPool()
{
    pool_size = 0;
}

template<typename T>
MyObjectPool<T>::MyObjectPool(int initNum)
{
    pool_size = 0;
    expandPool(initNum);
}

template<typename T>
MyObjectPool<T>::~~MyObjectPool()
{

```

```

        pool.clear();
    }

template<typename T>
QSharedPointer<T> MyObjectPool<T>::acquire()
{
    if (pool.isEmpty()) {
        expandPool(10);
    }

    return pool.takeFirst();
}

template<typename T>
void MyObjectPool<T>::release(QSharedPointer<T> &object)
{
    pool.append(object);
}

template<typename T>
void MyObjectPool<T>::expandPool(int expSize)
{
    for (int i = 0; i < expSize; ++i) {
        pool.append(QSharedPointer<T>(new T));
    }
    pool_size += expSize;
}

#endif // MYOBJECTPOOL_H

```

10. Map (地图)

- **功能:** 定义关卡与地图布局. 从外面的文件读取: 地图样貌; 敌人生成点位置, 敌人移动路径, 敌人生成间隔与类型; 障碍物图片, 类型与位置; 可放置位置; 目标点位置与图片. 玩家初始血量与钱币数.
- **主要成员变量:**
 - `grid`: 二维数组表示地图的网格 (包含障碍物, 道路和可放置位置)
 - `playerHealth`: 玩家初始血量
 - `playerMoney`: 玩家初始钱币数

- `QVector<<int>,<int>> enemyPath` : 敌人道路
- `QVector<int> enemyTypes` : 敌人类型
- `QVector<<int>,<int>,<int>> obsPos` : 障碍物位置与类型
- **主要方法:**
 - `loadMap(int level)` : 根据关卡载入地图. 在此函数内要读取文件信息.
 - `isPlaceAble(QPoint pos_)` : 检查某个位置是否可放置方块
 - `getPath()` : 返回敌人可以走的道路
 - `getSpawnPoints()` : 返回敌人生成的起始位置
 - `getEnemyType()` : 返回敌人的类型. 0 表示不生成敌人, -1 表示游戏胜利. 其他数字一个表示一类敌人. 每调用一次这个函数就将迭代器加一, 获取下一个敌人类型.
 - `getObsPosType()` : 返回障碍物的网格地图位置与类型.

示例代码:

```
class Map : public QObject {
    Q_OBJECT

public:
    Map(QObject *parent = nullptr);

    void loadMap(int level);
    QVector<QPoint> getSpawnPoints() const;
    QVector<QPoint> getPath() const;
    int getEnemySpawnInterval() const;
    QVector<int> getEnemyTypes() const;

private:
    QVector<QPoint> spawnPoints;
    QVector<QPoint> enemyPath;
    int enemySpawnInterval;
    QVector<int> enemyTypes; // 用于存储敌人的类型
};

Map::Map(QObject *parent)
    : QObject(parent), enemySpawnInterval(1000) {
}

void Map::loadMap(int level) {
    // 根据关卡载入地图和敌人生成信息
    // 示例代码, 实际实现需要根据具体关卡信息进行调整
}
```

```

    if (level == 1) {
        spawnPoints = {QPoint(0, 0)};
        enemyPath = {QPoint(0, 0), QPoint(1, 0), QPoint(1, 1)};
        enemySpawnInterval = 1000; // 每秒生成一个敌人
        enemyTypes = {1, 2, 1}; // 敌人的类型
    }
}

 QVector<QPoint> Map::getSpawnPoints() const {
    return spawnPoints;
}

 QVector<QPoint> Map::getPath() const {
    return enemyPath;
}

 int Map::getEnemySpawnInterval() const {
    return enemySpawnInterval;
}

 QVector<int> Map::getEnemyTypes() const {
    return enemyTypes;
}

```

11.Obstacle(障碍物)

- **功能:** 固定不动, 被击碎可以给予玩家金币. 不会复活. 实现上可以使用 type 变量表示不同的障碍物, 不需要定义各种子类. (被塔攻击的优先级低于敌人)
- **主要成员变量:**
 - price : 被击碎后可以给予玩家的金币
 - health : 生命值
- **主要方法:**
 - checkIfDead()
- **信号:**
 - isDamaged(int price) : 障碍物被破坏时发出此信号, 给予玩家金币