致　谢

- 本次学习汇报关于C++移动语义部分参考了《C++ primer plus》一书。
- 特别的，Swap&Copy（P9）部分和以及完美转发（P11）两部分参考自@越行勤（https://space.bilibili.com/292678409）在博客（https://blog.yxqin.top/2022/03/13/cppmoveforward）中的相关表述，十分感谢他的无私分享，有关移动语义的详细内容欢迎大家到他的视频BV19S4y137yS进行学习交流。
- QT部分的案例根据@传智教育（https://space.bilibili.com/396491181）的教学课程（BV1g4411H78N）进行学习，所有素材均来自于课程资源的分享。十分感谢@传智教育提供的高质量公开课以及学习素材，也欢迎大家跳转视频链接学习交流!

学习进度汇报

——王壹

- C++11移动语义和类的新功能
- QT学习汇报

移动语义

左值lvalue与右值rvalue

- 左值：有地址的值 located value，可以放在等号左边。
- 右值：临时值，不能取地址，不能放在等号左边。
- i, a：在内存中有位置的实际变量。
- 10和GetValue：字面量，没有存储空间和位置。

```
int i = 10;
10 = i;
```

```
int a = i;
```

```
int GetValue()
{
    return 10;
}
```

```
int i = GetValue();
GetValue() = i;
```

注意：

- 字符串的字面值也属于左值，其他字面值都是右值。
- 返回非引用类型的函数是右值，返回左值引用的是左值。
- 解引用表达式*p是左值，取地址表达式&a是右值。
- ++i是左值，而i++是右值。

```
int& GetValue()
{
    static int value = 10;
    return value;
}
```

```
int i = GetValue();
GetValue() = i;
```

移动语义

左值引用与右值引用

- 左值引用：对左值进行引用的类型。
- 右值引用：对右值进行引用的类型。

```
int & name = i; // 左值引用
int &&name = 10;   // 右值引用
```

```
1  int a = 5;
2  int &b = a; // b是左值引用
3  b = 4;
4  int &c = 10; // error，10无法取地址，无法进行引用
5  const int &d = 10; // ok，因为是常引用，引用常量数字，这个常量数字会存储在内存中，
```

注意：

- 使用const的左值引用可以引用右值，因为const的左值引用遇到 一个将亡值，会将将亡值的生命周期延长为这个引用的生命周期的长度。

移动语义

为什么需要移动语义

```
vector<string> vstr;
//创建一个有20000个string的vector，每个string有1000个字母
vetor<string> allcaps(const vector<string> & vs)
{
    vector<string> temp;
    … //处理vector
    return temp;
}
vector<string> vstr_copy1(vstr); //#1
vector<string> vstr_copy2(allcaps (vstr)); //#2
```

解释：

- 连续的临时数据复制做了无用功。
- 直接把temp的所有权转让给vstr_copy2？
- 移动语义保留实际文件的位置，只修改记录。

移动语义

一个示例

```
Useless();
explicit Useless(int k);
Useless(int k, char ch);
Useless(const Useless & f); // regular copy constructor
Useless(Useless && f);      // move constructor
~Useless();
Useless operator+(const Useless & f)const;
```

```
private:
    int n;
    char * pc;
    static int ct;
```

```
Useless::Useless(Useless && f): n(f.n)
{
    ++ct;
    cout << "move constructor called; number of objects: " <<
    ct << endl;
    pc = f.pc;        // steal address
    f.pc = nullptr;  // give old object nothing in return
    f.n = 0;
    ShowObject();
}
```

- 不能对同一个地址delete[]两次，因此要指向空指针。
- 同样的，窃取应将原始对象的元素值设置为0。
- 由于修改了f对象，因此不能在参数声明中使用const。

移动语义

一个示例

```
Useless one(10, 'x');
Useless two = one;          // 复制构造函数
Useless three(20, 'o');
Useless four(one + three);  // +运算符和移动构造函数
```

```
int, char constructor called; number of objects: 1
Number of elements: 10 Data address: 006F4B68
copy const called; number of objects: 2
Number of elements: 10 Data address: 006F4BB0
int, char constructor called; number of objects: 3
Number of elements: 20 Data address: 006F4BF8
Entering operator+()
int constructor called; number of objects: 4
Number of elements: 30 Data address: 006F4C48
temp object:
Leaving operator+()
move constructor called; number of objects: 5
Number of elements: 30 Data address: 006F4C48
```

移动语义

一个示例

```
Useless one(10, 'x');
Useless two = one;          // 复制构造函数
Useless three(20, 'o');
Useless four(one + three);  // +运算符和移动构造函数
```

int, char constructor called; number of objects: 1
Number of elements: 10 Data address: 0x1f1920
copy const called; number of objects: 2
Number of elements: 10 Data address: 0x1f1960
int, char constructor called; number of objects: 3
Number of elements: 20 Data address: 0x1f3b70
Entering operator+()
int constructor called; number of objects: 4
Number of elements: 30 Data address: 0x1f3bc0

- 创建four的时候只调用了一次构造函数，这里没有显示的调用移动构造函数是因为编译器推断four是operator+()的受益人，因此自动将operator+()创建的对象转移到four名下。

移动语义

Swap & copy

- 拷贝构造函数和拷贝赋值函数、移动构造函数和移动赋值函数的代码重复？

```
static void Swap(buffer& lhs, buffer& rhs)noexcept
{
    std::swap(lhs.buf, rhs.buf);
    std::swap(lhs.capacity, rhs.capacity);
    std::swap(lhs.len, rhs.len);
}
```

```
buffer& operator=(buffer other)noexcept
{
    Swap(*this, other);
    return *this;
}
```

```
buffer(buffer&& buffer):noexcept :capacity(0), len(0), buf
(nullptr)
{
    Swap(*this, buffer);
}
```

```
if(capacity)
    std::copy(buffer.buf, buffer.buf + buffer.capacity,
    this->buf);
```

- 对于赋值构造函数，使用值传递的形式。
- 传入左值时调用一次拷贝构造，因此代替了手动copy。
- 传入右值时调用一次移动构造，再交换到this中。

移动语义

另一个示例

```
Useless choices[10];
Useless best;
int pick;
... // 选择一个对象
best = choices[pick];
```

- 问题： choices[pick]是左值，因此调用的是复制赋值运算符而非移动赋值运算符，但是使用时希望以右值引用的方式进行调用。
- 解决1：使用static_cast<>将对象类型强制转化为Useless &&。
- 解决2：使用头文件utility中声明的函数std::move()。

```
four = std::move(one);     // f
cout << "now object four = ";
```

注意：

- 对std::move()，如果没有定义移动赋值运算符，则编译器会调用复制赋值运算符。

移动语义

另一个示例

```
Useless one(10, 'x');
Useless two = one + one; // calls move constructor
Useless three, four;
three = one; // automatic copy assignment
four = one + two; // automatic move assignment
four = std::move(one); // forced move assignment
```

```
copy assignment operator called:
now object three = xxxxxxxxxx
and object one = xxxxxxxxxx
four = one + two
move assignment operator called:
now object four = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
four = move(one)
move assignment operator called:
now object four = xxxxxxxxxx
and object one = (object empty)
```

- 将one赋值给three调用了复制赋值运算符。
- 将move(one)赋值给four调用的是移动赋值运算符。

移动语义

完美转发：返回值的左/右属性和参数的一致。

- 使用万能引用可以保证完美转发，因为左值会被展开为T&，而右值会被展开为T&&。
- 一般转发：arg是左值

```cpp
template<typename T,typename Arg>
std::shared_ptr<T> make_shared_prt(Arg&& arg) {
        return std::shared_ptr<T>(new T(arg))
}
```

- 使用std::forward，这样我们就可以让 arg 被转发参数的左、右值属性不变。

```cpp
template<typename T,typename Arg>
std::shared_ptr<T> make_shared_prt(Arg&& arg) {
        return std::shared_ptr<T>(new T(std::forward<Arg>(arg)));
}
```

```cpp
template<typename T>
void fun(T&& t)
{
        std::cout<<"万能引用\n";
}


        int i=0;
        fun(1);
        fun(i);
```

类的新功能

特殊成员函数：

- C++11在原有的4个特殊成员函数（默认构造函数、复制构造函数、复制赋值运算符和析构函数）的基础上增加了两个：移动构造函数和移动赋值运算符。

  Someclass::Someclass(Slomeclass &&);

  Someclass::Someclass operator=(Someclass &&);

- 特别的，如果个人提供了析构、复制构造和复制赋值，则编译器不会自动提供移动构造函数和移动赋值运算符。
- 如果个人提供了移动构造和移动赋值，则编译器不会自动提供复制构造函数和复制赋值运算符。

类的新功能

默认方法的禁用：

- 由于个人提供了一些函数导致编译器不会自动创建默认的其他函数时，可以使用**default**显式的进行声明。
- 如果不想使用编译器的特定方法可以使用关键词**delete**。

```
class Someclass
{
public:
        Someclass(Someclass &&);
        Someclass() = default;
        Someclass(const Someclass &) = default;
        Someclass & operator=(Someclass &&) = delete;
…
}
```

- 关键字default只能用于6个特殊成员函数，但delete可用于任何成员函数。

类的新功能

委托构造函数：

- C++11允许在一个构造函数的定义中使用另一个构造函数，这被称为委托。委托使用成员初始化列表语法的变种：

```
1  class Notes
2  {
3      int k;
4      double x;
5      std::string st;
6  public:
7      Notes();
8      Notes(int);
9      Notes(int, double);
10     Notes(int, double , std::string);
11 };
12 Notes::Notes(int kk, double xx, std::string stt) : k(kk),
13 x(xx), st(stt) {}
14 Notes::Notes() : Notes(0, 0,01, “Oh”) {}
15 Notes::Notes(int kk) : Notes(kk, 0,01, “Ah”) {}
16 Notes::Notes(int kk, double xx) : Notes(kk, xx, “Uh”) {}
```

类的新功能

继承构造函数：

- 在派生类中使用using声明可以让派生类继承基类的所有构造函数（默认构造函数、复制构造函数和移动构造函数除外）。
- 但不会使用与派生类构造函数的特征标匹配的构造函数。
- 继承基类的构造函数只初始化基类成员，如果还要初始化派生类成员，则应使用成员列表初始化语法。

```
1  class BS
2  {
3      int q;
4      double w;
5  public:
6      BS() : q(0), w(0) {}
7      BS(int k) : q(k), w(100) {}
8      BS(double x) : q(-1), w(x) {}
9      BS(int k, double x) : q(x), w(x) {}
10 };
11
12 class DR : public BS
13 {
14     short j;
15 public:
16     using BS::BS;
17     DR() : j(-100) {}
18     DR(double x) : BS(2 * x), j(int(x)) {}
19     DR(int i) : j(-2), BS(i, 0.5 * i) {}
20 };
21
22 int main()
23 {
24     DR o1;              //使用DR()
25     DR o2(18.81);   //使用DR(double)而不是BS(double)
26     DR o3(10, 1.8); //使用BS(int, double)
27     //...
28 }
```

类的新功能

管理虚方法：

- 假设基类声明了一个虚方法，而在派生类中提供了不同的版本，这将覆盖旧版本。如果特征标不匹配，将隐藏而不是覆盖旧版本：

```
class Action
{
    int a;
public:
    Action(int i = 0) : a(i) {}
    int val() const { return a; }
    virtual void f(char ch) const { std::cout << val() << ch << "\n";}
};

class Bingo : public Action
{
public:
    Bingo(int i = 0) : Action(i) {}
    virtual void f(char * ch) const {std::cout < val() <<ch <<"!\n";}
};
```

- 无法使用char的方法

```
Bingo b(10);
b.f('@');
```

- 可使用虚说明符override指出要覆盖一个虚函数，编译器自动报警

```
virtual void f(char * ch) const  override
```

- 如果想禁止派生类覆盖特定的虚方法，可在参数列表后面加上final。

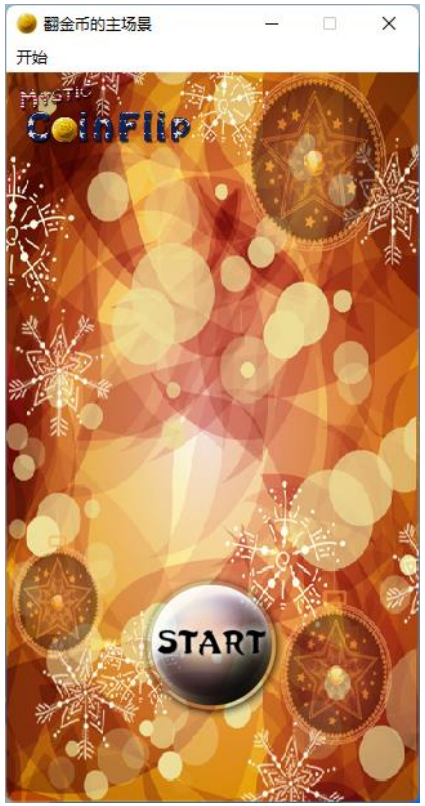
```
virtual void f(char * ch) const final
```

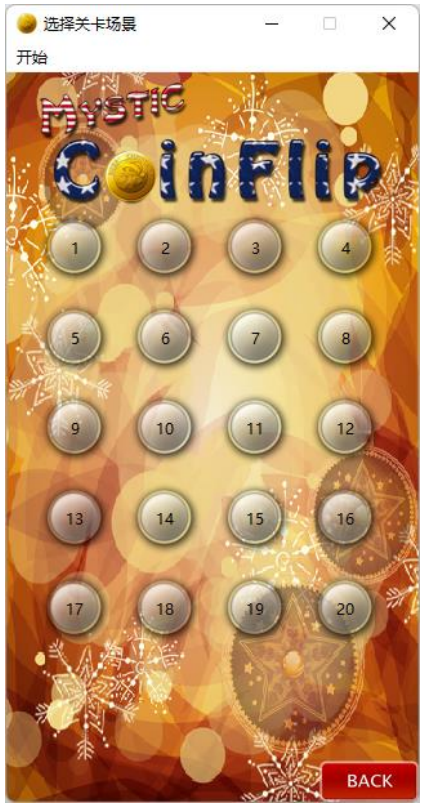
- QT学习汇报（传智教育课程学习案例）

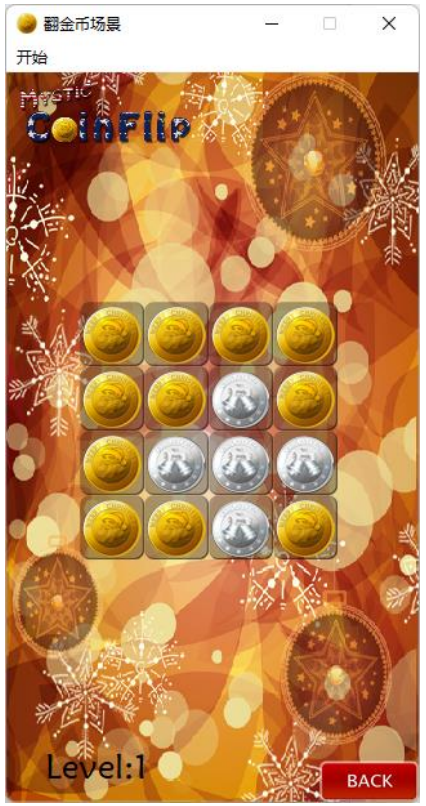
游戏案例

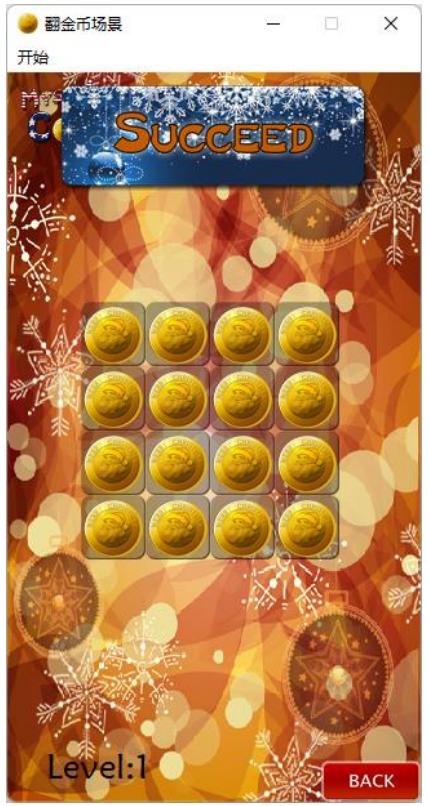
功能要求与目标：

- 让所有的金币翻为金色则实现目标。









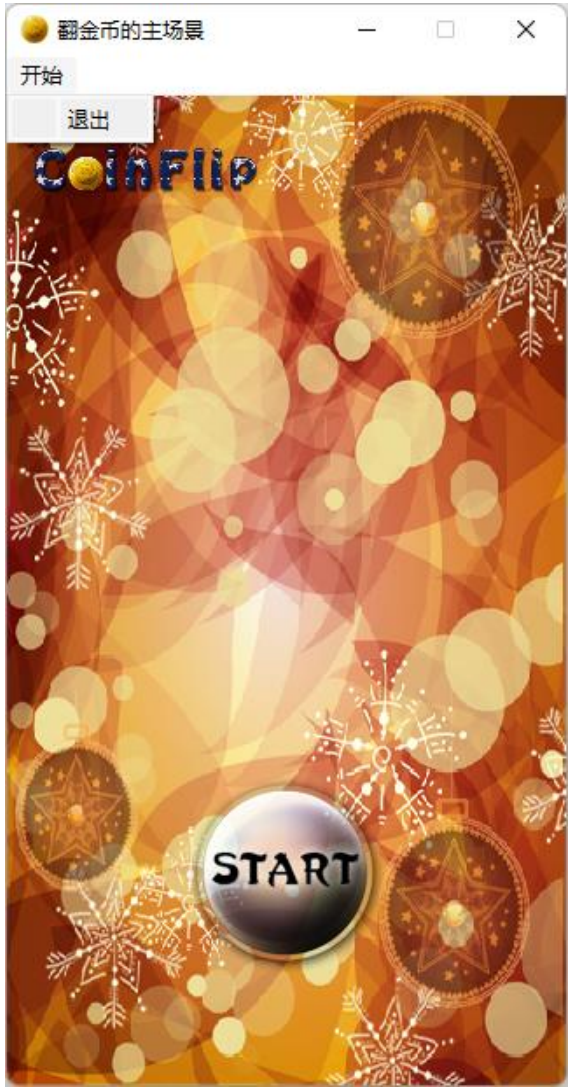
游戏案例

主界面MainScene实现：

- 场景布置：

1. 使用ui界面布置MenuBar，设置了开始下面的退出选项。
2. 在构造函数中配置标题，图标等场景。
3. 重写paintEvent以绘制背景图片。

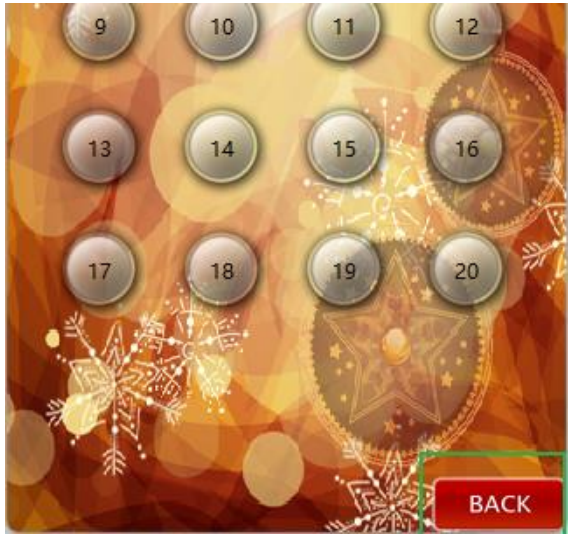
```
void MainScene::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix;
    pix.load(":/res/PlayLevelSceneBg.png");
    painter.drawPixmap(0, 0, this->width(), this->height(), pix);

    //画背景上的图标
    pix.load(":/res/Title.png");

    pix = pix.scaled(pix.width() * 0.5, pix.height()*0.5);
    painter.drawPixmap(10, 30, pix);
}
```



游戏案例

主界面MainScene实现：

- 重写按钮：继承QPushButton
  1. 应能满足START按钮的需求：
     - 应该能实现点击后弹跳的动作。
  2. 应能满足back按钮的需求：
     - 点击后按钮的图片进行变化。



- 按钮的构造函数参数应包括两个图片：

```
//构造函数 参数1正常显示 参数2按下显示
MyPushButton(QString normalImg, QString pressImg = "");
//设置按钮大小
this->setFixedSize(pix.width(), pix.height());
//设置不规则图片样式
this->setStyleSheet("QPushButton{border:0px}");
//设置图标
this->setIcon(pix);
//设置图标大小
this->setIconSize(QSize(pix.width(),pix.height()));
```

游戏案例

主界面MainScene实现：

- 重写按钮——START按钮的功能：
  - 应该能实现点击后弹跳的动作。

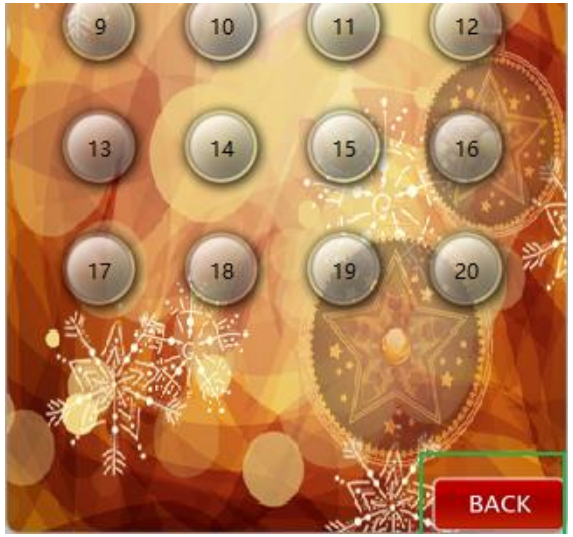

```
void MyPushButton::zoom1()
{
    //创建动画对象
    QPropertyAnimation * animation = new QPropertyAnimation(this, "geometry");
    //动画时间间隔
    animation->setDuration(200);

    //起始位置
    animation->setStartValue(QRect(this->x(), this->y(), this->width(), this->height()));

    //结束为止
    animation->setEndValue(QRect(this->x(), this->y()+10, this->width(), this->height()));

    //设置弹跳曲线
    animation->setEasingCurve(QEasingCurve::OutBounce);

    //执行动画
    animation->start();
}
void MyPushButton::zoom2()
```

游戏案例

主界面MainScene实现：

- 重写按钮——back按钮的功能：
  - 点击后按钮的图片进行变化。

```
void MyPushButton::mousePressEvent(QMouseEvent * e)
{
    if(this->pressImgPath != "")
    {
        QPixmap pix;
        pix.load(this->pressImgPath);
        //设置图片大小
        this->setFixedSize(pix.width(), pix.height());
        //设置不规则图片样式
        this->setStyleSheet("QPushButton{border:0px}");
        //设置图标
        this->setIcon(pix);
        //设置图标大小
        this->setIconSize(QSize(pix.width(),pix.height()));
    }

    //让父类执行其他内容
    QPushButton::mousePressEvent(e);
}

void MyPushButton::mouseReleaseEvent(QMouseEvent * e)
```



游戏案例

主界面MainScene实现：

- 信号与槽：实现点击的功能。

1. menuBar下的退出功能实现：

```
//退出按钮的实现
connect(ui->actionquit,&QAction::triggered, this, [=](){
    this->close();
});
```

2. 开始按钮的功能实现：

```
connect(startBtn, &MyPushButton::clicked, this, [=](){

    //做一个弹起的特效
    startBtn->zoom1();
    startBtn->zoom2();

    //延时进入
    QTimer::singleShot(300,this,[=](){
        //设置场景位置
        chooseScene->setGeometry(this->geometry());
        //进入到选择关卡
        //自身隐藏
        this->hide();
        //显示选择场景
        chooseScene->show();
    });
});
```

3. 实例化关卡选择界面与监听返回信号

```
//实例化选择关卡场景
chooseScene = new ChooseLevelScene;

//监听选择关卡的返回按钮的信号
connect(chooseScene,&ChooseLevelScene::chooseSceneBack, this, [=](){
    this->setGeometry(chooseScene->geometry());
    chooseScene->hide();
    this->show();
});
```

注：

- 延迟进入选择关卡避免了动画没有执行完就进入关卡的bug。
- 返回按钮虽然在chooseScene的场景中，但是信号监听和执行却是在MainScene中实现。
- setGeometry避免了下一场景和当前场景界面位置不同的bug。

游戏案例

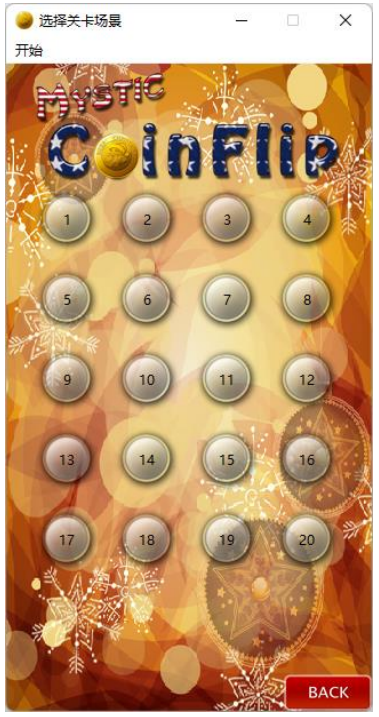
关卡界面ChooseScene实现：

- 菜单栏，背景图片设置等基础功能。
- 能执行点击时图片更改的返回按钮（通过重写mousePressEvent实现的）。

```
//返回按钮
MyPushButton * backBtn = new MyPushButton(":/res/BackButton.png", ":/res/BackButtonSelected.png");
backBtn->setParent(this);
backBtn->move(this->width() - backBtn->width(), this->height() - backBtn->height());

//点击返回连接
connect(backBtn, &MyPushButton::clicked, [=](){
    //告诉主场景返回 主场景要监听返回
        QTimer::singleShot(200,this,[=](){
        emit this->chooseSceneBack();
    });
});
```

- 点击返回按钮时发出返回信号。

```
signals:
    //自定义信号
    void chooseSceneBack();
```



游戏案例

关卡界面ChooseScene实现：

- 关卡按钮的实现。

```
for (int i = 0; i < 20; i++)
{
    MyPushButton * menuBtn = new MyPushButton(":/res/LevelIcon.png");
    menuBtn->setParent(this);
    menuBtn->move(25 + i%4 * 70, 130 + i/4 * 70);

    //设置label
    QLabel * label = new QLabel;
    label->setParent(this);
    label->setFixedSize(menuBtn->width(),menuBtn->height());
    label->setText(QString::number(i+1));
    label->setAlignment(Qt::AlignHCenter | Qt::AlignVCenter);
    label->move(25 + i%4 * 70, 130 + i/4 * 70);
    //设置鼠标穿透
    label->setAttribute(Qt::WA_TransparentForMouseEvents);
}
```

- Label用于显示数字，鼠标穿透避免了鼠标点在label上按钮接受不到响应的问题。

游戏案例

关卡界面ChooseScene实现：

- 关卡按钮的监听。

```
//监听
connect(menuBtn,&MyPushButton::clicked,this,[=](){
    QString str = QString("您选择的是第 %1 关").arg(i+1);
    qDebug() << str;

    //进入游戏场景
    play = new PlayScene(i+1);
    //初始位置
    play->setGeometry(this->geometry());
    this->hide();
    play->show();
    //监听下一个界面的返回
    connect(play, &PlayScene::chooseSceneBack, this, [=](){
        this->setGeometry(play->geometry());
        this->show();
        delete play;
        play = nullptr;
    });
});
```

注：

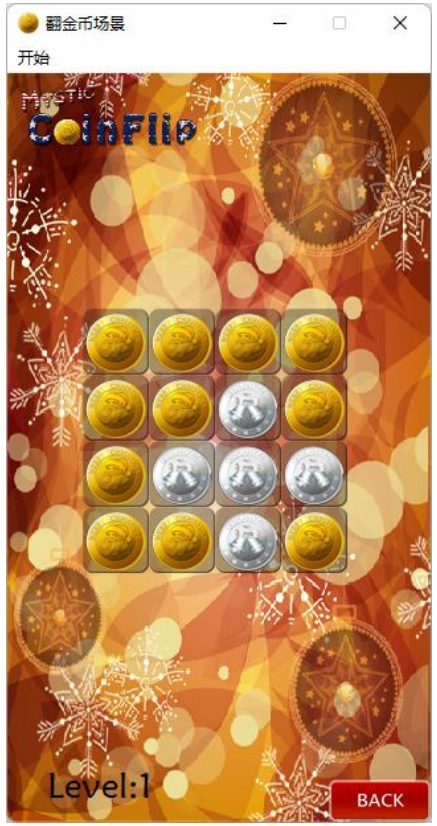
- 监听本身处于for的循环内，以确保每个按钮都实现监听。
- 监听返回处于创建关卡的监听是因为离开lambda表达式后创建的play场景就被自动销毁了。

游戏案例

游戏界面PlayScene实现：

- 菜单栏，背景图片设置，关卡数字显示等基础功能。
- 返回按钮的信号监听chooseScene一致。
- 关卡数据由dataConfig类中的map容器进行输入。 `QMap<int, QVector<QVector<int>> > mData;`

```
dataConfig::dataConfig(QObject *parent) : QObject(parent)
{

    int array1[4][4] = {{1, 1, 1, 1},
                        {1, 1, 0, 1},
                        {1, 0, 0, 0},
                        {1, 1, 0, 1} } ;

    QVector< QVector<int>> v;
    for(int i = 0 ; i < 4;i++)
    {
        QVector<int>v1;
        for(int j = 0 ; j < 4;j++)
        {

            v1.push_back(array1[i][j]);
        }
        v.push_back(v1);
    }

    mData.insert(1,v);
```



游戏案例

游戏界面PlayScene实现——数据的维护：

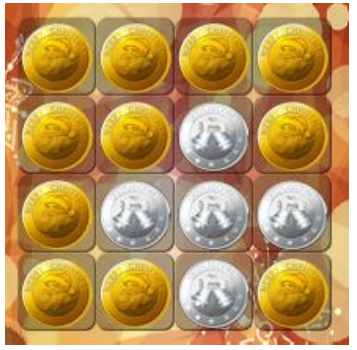
```
PlayScene(int levelNum);
```

- 关卡数由一个int类型管理，游戏场景的构造函数参数中指明了关卡数。 `int levelIndex;`
- 关卡数据在游戏场景中由一个二维int数组进行管理。 `int gameArray[4][4];`

```
dataConfig config;
//初始化每个关卡的二维数组
for(int i = 0; i < 4; i++)
{
    for(int j = 0; j < 4; j++)
    {
        this->gameArray[i][j] = config.mData[this->levelIndex][i][j];
    }
}
```

- 关卡的金币是由一个金币类型的二维数组进行维护的。 `MyCoin * coinBtn[4][4];`
- 金币的背景图案是使用label来实现的。
- 判断胜利的条件是由一个bool值维护的。

```
//是否胜利
bool isWin;
```



游戏案例

游戏界面PlayScene实现——金币按钮：

- 构造函数的参数指明了是金币或者是银币路径。 `MyCoin(QString btnImg);`
- 包括的成员应有：金币的位置、正反标示、启动翻面动作的2个定时器、记录翻面时的图片顺序值、动画是否完成的标志以及判断是否胜利的标志。

```
//金币属性
int posX; //x坐标位置
int posY; //y坐标位置
bool flag; //正反标示
```

```
QTimer * timer1; //正面翻反面的定时器
QTimer * timer2; //反面翻正面的定时器
int min = 1;
int max = 8;
```

```
//执行动画的标志
bool isAnimation = false;
//是否胜利
bool isWin = false;
```

- 方法应包括改变标志的方法，以及重写的鼠标按下时的事件。

```
//改变标志的方法
void changeFlag();
```

```
//重写 按下
void mousePressEvent(QMouseEvent *e);
```

游戏案例

游戏界面PlayScene实现——金币按钮：

- 金币按钮的图片大小设置等初始化信息。
- 构造函数中利用timer定时器监听翻面工作的开始。

```
connect(timer1, &QTimer::timeout, this, [=](){
    QPixmap pix;
    QString str = QString(":/res/Coin000%1.png").arg(this->min++);
    pix.load(str);

    this->setFixedSize(pix.width(),pix.height());
    this->setStyleSheet("QPushButton{border:0px;}");
    this->setIcon(pix);
    this->setIconSize(QSize(pix.width(),pix.height()));
    //判断 如果翻完了则重置
    if(this->min > this->max)
    {
        this->min = 1;
        timer1->stop();
        isAnimation = false;
    }
});
```

游戏案例

游戏界面PlayScene实现——金币按钮：

- 改变正反面的方法。

```
void MyCoin::changeFlag()
{
    //如果是正面 翻成反面
    if(this->flag)
    {
        timer1->start(30);
        isAnimation = true; //开始做动画
        this->flag = false;
    }
    else
    {
        timer2->start(30);
        isAnimation = true; //开始做动画
        this->flag = true;
    }
}
```

- 重写的鼠标按下时的事件：金币翻转和胜利时禁止金币的点击。

```
void MyCoin::mousePressEvent(QMouseEvent *e)
{
    if (this->isAnimation || this->isWin)
    {
        return;
    }
    else
    {
        QPushButton::mousePressEvent(e);
    }
}
```

这个事件避免了一个bug：同一个金币在翻转的过程中再次点击的情况，isAnimation为真时鼠标点击的行为被拦截。

游戏案例

游戏界面PlayScene实现：

- 翻面动作的实现：在一个嵌套的for循环中创建金币并进行赋值，对点击行为进行监听。

```
connect(coin, &MyCoin::clicked, this, [=](){
    coin->changeFlag();
    this->gameArray[i][j] = !(this->gameArray[i][j]);
    QTimer::singleShot(300, this, [=](){
        if(coin->posX + 1 <= 3) //右侧金币翻转条件
        {
            coinBtn[coin->posX+1][coin->posY]->changeFlag();
            this->gameArray[coin->posX+1][coin->posY] =
                !(this->gameArray[coin->posX+1][coin->posY]);
        }
```

- 判断胜利的条件：

```
this->isWin = true;
for(int i = 0; i < 4; i++)
{
    for (int j = 0; j < 4; j++)
    {
        if(coinBtn[i][j]->flag == false)
        {
            this->isWin = false;
            break;
        }
    }
}
```

游戏案例

游戏界面PlayScene实现：

- 一个bug的修复：金币中重写的鼠标事件避免了金币翻转时继续被鼠标点击，但是周围正在翻转的金币可能是受到鼠标点击。
- 在金币翻转前将金币的设置为胜利条件以避免继续点击，点击活动结束后再设置为false。

```
for (int i = 0; i < 4 ; i++ ) {
    for(int j = 0; j < 4; j++)
    {
        this->coinBtn[i][j]->isWin = true;
    }
}
```

- 利用label显示胜利图片，利用animation实现图片的运动。

```
QLabel* winLabel = new QLabel;
QPixmap tmpPix;
QPropertyAnimation * animation = new QPropertyAnimation(winLabel, "geometry");
animation->start();
```