专题技术报告

Qt简介

Qt是一个跨平台的C++图形用户界面应用程序开发框架，由Qt Company开发。它不仅可以用来开发GUI程序，也适用于非GUI程序，如控制台工具和服务器。Qt支持多种操作系统，包括Windows、Linux、Unix以及各种嵌入式系统。

专用IDE

Qt框架专用IDE为QT Creator，支持可视化界面设计，有图形化界面用于调整界面内容，可以大量节省代码工作量。

信号与槽函数机制

Qt的对象可以发出信号，并且可以连接到槽，这是一种特殊的成员函数，用于响应特定的信号。信号与槽（Signals and Slots）机制是一种高级的回调机制，用于对象之间的通信。它允许一个对象在发生特定事件时通知其他对象。

信号（Signal）的特征：

- 由 `signals` 关键字声明，不需要实现，仅需声明。
- 可以有参数，参数类型可以是任意的。
- 一个信号可以连接到多个槽函数。

槽（Slot）的特征：

- 由 `slots` 关键字声明的成员函数。
- 槽函数也可以有参数，并且参数类型可以是任意的。
- 槽函数用于响应信号，执行相应的动作。

Qt的`connect`函数用于连接信号和槽：

- Qt允许使用函数指针直接连接信号和槽，提高了类型安全性。

以下是一个使用Qt5信号和槽的示例代码：

```
#include <QObject>
#include <QDebug>

class MyObject : public QObject {
    Q_OBJECT

signals:
    void mySignal(int value);

public slots:
    void mySlot(int value) {
        qDebug() << "Slot function called with value:" << value;
    }
};

int main(int argc, char *argv[]) {
```

```
    QCoreApplication a(argc, argv);
    MyObject obj;
    QObject::connect(&obj, &MyObject::mySignal, &obj, &MyObject::mySlot);
    // 发射信号
    emit obj.mySignal(42);
    return a.exec();
}
```

在这个例子中，`MyObject` 类有一个信号 `mySignal` 和一个槽 `mySlot`。当 `mySignal` 被发射时，它会调用连接到这个信号的所有槽函数。在这个例子中，`mySlot` 将被调用，并打印出传递给它的值。

接下来看一个更详细的例子，是本次我的大作业内容。

```
class Checkerboard : public QWidget {
public:
    Checkerboard(QWidget *parent = nullptr) : QWidget(parent) {
        // 设置窗口标题
        setWindowTitle(tr("Chinese Checkers"));

        QPushButton *refleshButton = new QPushButton("New Game", this);
        QPushButton *ruleButton = new QPushButton("Game Rule", this);
        QPushButton *nextButton = new QPushButton("End Round", this);

        connect(refleshButton, &QPushButton::clicked, this,
&Checkerboard::reflesh);
        connect(nextButton, &QPushButton::clicked, this,
&Checkerboard::nextround);
        connect(ruleButton, &QPushButton::clicked,this,&Checkerboard::showrule);


    }
}
```

- 以上是我的主界面的类的构造函数，只保留了构造函数，删除了信号和槽函数之外的无关元素。
- 首先定义了三个按钮，之后利用connect槽函数设置了点击之后的响应事件。
- 槽函数的四个主要参数为：信号发送者，信号内容，响应执行者，相应内容。
- 从而实例中第一个connect的意义为如果名为refleshButton按钮发出了clicked信号（即此按钮被点击），界面的主类调用showrule函数进行响应。

事件处理

Qt应用程序是事件驱动的。事件如鼠标点击或按键等，都可以通过重写事件处理函数来自定义响应。

Qt的事件驱动机制是其核心特性之一，它允许应用程序以响应式的方式处理用户界面事件和其他类型的事件。以下是Qt事件驱动机制的详细解析和示例代码：

事件循环（Event Loop）：

- 事件循环是Qt应用程序的核心，负责接收和分发事件。
- 它运行在一个无限循环中，等待事件的发生，并将它们分发到相应的对象进行处理。

事件（Event）：

- 事件是由用户操作（如鼠标点击、按键输入）或系统（如定时器超时）产生的。
- Qt中的事件通常是`QEvent`类的实例，包含事件的所有信息，如类型和时间戳。

事件处理:

- 事件被发送到应用程序的不同部分，通常是某个窗口部件（widget），以触发相应的响应动作。
- 开发者可以通过重写事件处理函数（如`mousePressEvent`、`keyPressEvent`等）来自定义组件的行为。

以下是一个简单的Qt事件处理示例代码及其解析：

```
#include <QApplication>
#include <QWidget>
#include <QMouseEvent>

class MyWidget : public QWidget {
public:
    // 构造函数
    MyWidget() {}

protected:
    // 重写鼠标点击事件处理函数
    void mousePressEvent(QMouseEvent *event) override {
        // 检查是否是左键点击
        if (event->button() == Qt::LeftButton) {
            // 执行相关操作
            qDebug() << "Left mouse button pressed.";
        }
    }
};

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    MyWidget widget;
    widget.show();
    return app.exec(); // 启动事件循环
}
```

在这个示例中，我们创建了一个`MyWidget`类，它继承自`QWidget`。我们重写了`mousePressEvent`函数，以便在用户点击窗口时进行自定义操作。当左键点击时，程序会输出一条消息到控制台。

下面再看一个例子，是我的大作业中的事件重写内容。

```
protected:
    void paintEvent(QPaintEvent *event) override {
        Q_UNUSED(event);
        QPainter painter(this);
        // 绘制棋盘
        drawCheckerboard(painter);

    }

    void drawCheckerboard(QPainter &painter) {
        //具体实现...
    }
```

```
void mousePressEvent(QMouseEvent* event) override
{
    if(present==coord(6,6,6)){
        if(fetch(event->pos())==coord(-1,-1,-1)){
            if(centrestate==turn){
                present=coord(-1,-1,-1);
            }
        }else if(fetch(event->pos())==coord(6,6,6)){
            return;
        }else{
            coord now=fetch(event->pos());
            if(state[now.r][now.s][now.t]==turn){
                present=fetch(event->pos());
            }
        }
    }else if(present==coord(-1,-1,-1)){
        //...
            }else if(state[now.r][now.s][now.t]==-1){
                //...
            }else{
                return;
            }
        }
    }else{
        //...
    }

    update();
    return;
}
```

这段代码是Qt中一个自定义窗口部件的绘图事件处理函数以及鼠标点击事件处理函数的实现，我们将其重写为想要的形式，这样就可以在重新刷新绘制界面和鼠标在界面上点击的时候达到指定的图形界面效果。

1. `paintEvent(QPaintEvent *event)`函数是QWidget的一个事件处理函数，它在窗口部件需要重绘时被调用。
2. `QPainter painter(this);`创建了一个`QPainter`对象，用于在当前窗口部件上进行绘制。
3. `drawCheckerboard(painter);`调用自定义的`drawCheckerboard`函数，传入`painter`对象作为参数，用于绘制棋盘。
4. 我们可以使用参数event的方法获取某一次点击的信息（例如event->pos()获得其点击位置，event->button()获得其点击的按钮信息），用来重写点击处理函数。
5. `update()`和`repaint()`函数都用于请求重绘窗口部件，但它们之间有一些关键的区别：
    - **update()**:
        - **合并重绘事件**：如果在短时间内多次调用`update()`，Qt会将这些调用合并成一个重绘事件，从而减少实际的重绘次数，提高效率。这意味着多次调用`update()`通常只会导致一次`paintEvent()`调用
        - **事件队列**：`update()`将重绘事件放入事件队列中，Qt会在适当的时候处理这个事件，通常是在当前事件处理循环结束后。

    - **性能优化**：由于`update()`不会立即重绘，它允许Qt进行优化，比如合并多个重绘区域，减少闪烁和提高性能。
- **repaint()**:
    - **立即重绘**：调用`repaint()`会立即重绘指定的窗口部件，不等待事件队列的处理。
    - **不合并事件**：`repaint()`不会合并多个重绘调用，因此如果频繁使用可能会导致性能问题。
    - **可能导致闪烁**：由于`repaint()`立即触发重绘，如果过于频繁使用，可能会导致界面闪烁。

界面布局

在Qt中，界面布局是通过布局管理器（layout managers）来设置的，它们负责控件的位置和大小。布局管理器自动适应窗口的大小变化，使得界面元素能够灵活地进行调整。以下是几种常用的布局管理器及其用法：

- **水平布局（QHBoxLayout）**：将控件水平排列。
- **垂直布局（QVBoxLayout）**：将控件垂直排列。
- **网格布局（QGridLayout）**：将控件放置在网格中。
- **表单布局（QFormLayout）**：专门用于表单类型的界面，自动创建标签和输入字段。

以下是一个简单的示例，展示了如何使用垂直布局来排列几个控件：

```
#include <QtWidgets>

class MyWidget : public QWidget {
public:
    MyWidget(QWidget *parent = nullptr) : QWidget(parent) {
        // 创建一个垂直布局
        QVBoxLayout *layout = new QVBoxLayout(this);

        // 创建几个控件
        QLabel *label1 = new QLabel("Label 1", this);
        QLabel *label2 = new QLabel("Label 2", this);
        QPushButton *button = new QPushButton("Button", this);

        // 将控件添加到布局中
        layout->addWidget(label1);
        layout->addWidget(label2);
        layout->addWidget(button);
    }
};

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    MyWidget widget;
    widget.show();
    return app.exec();
}
```

在这个例子中，我们首先创建了一个`QVBoxLayout`对象，然后创建了两个`QLabel`和一个`QPushButton`。这些控件被添加到布局中，布局管理器会自动处理它们的位置和大小。当`MyWidget`窗口的大小改变时，布局管理器会重新调整控件的大小和位置，以适应新的窗口尺寸。

- `QVBoxLayout *layout = new QVBoxLayout(this);` 创建了一个垂直布局，并将其父窗口设置为当前的`MyWidget`。
- `layout->addWidget(label1);` 将标签`label1`添加到布局中。布局管理器会自动分配位置。
- `widget.show();` 显示`MyWidget`窗口。
- `return app.exec();` 进入Qt的事件循环，等待用户操作。

下面我们再看一个本次大作业中的例子。

```
class Checkerboard : public QWidget {
public:
    Checkerboard(QWidget *parent = nullptr) : QWidget(parent) {
        // 设置窗口标题为"Chinese Checkers"
        setWindowTitle(tr("Chinese Checkers"));

        // 定义棋盘大小并设置窗口的固定大小为棋盘的两倍宽度和相同的高度
        boardSize = 750;
        setFixedSize(2*boardSize, boardSize);

        // 创建四个按钮：新游戏、游戏规则、结束回合和当前回合
        QPushButton *refleshButton = new QPushButton("New Game", this);
        QPushButton *ruleButton = new QPushButton("Game Rule", this);
        QPushButton *nextButton = new QPushButton("End Round", this);
        QPushButton *turnButton = new QPushButton("Current Turn", this);

        // 设置按钮的大小策略为固定
        refleshButton->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed);
        ruleButton->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed);
        nextButton->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed);
        turnButton->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed);

        // 设置按钮的固定大小为200x30像素
        refleshButton->setFixedSize(200, 30);
        ruleButton->setFixedSize(200, 30);
        nextButton->setFixedSize(200, 30);
        turnButton->setFixedSize(200, 30);

        // 创建一个垂直布局管理器
        QVBoxLayout *layout = new QVBoxLayout(this);
        // 设置布局中控件的间距为1像素
        layout->setSpacing(1);
        // 设置布局的边距为10像素
        layout->setContentsMargins(10, 10, 10, 10);

        // 将按钮添加到布局中
        layout->addWidget(refleshButton);
        layout->addWidget(ruleButton);
        layout->addWidget(nextButton);
        // 在按钮之间添加一个可伸缩的空间，使得按钮可以在布局的顶部
        layout->addStretch();
        layout->addWidget(turnButton);
        // 再次添加一个可伸缩的空间，使得最后一个按钮可以在布局的底部
        layout->addStretch();
    }
```

```
};
```

- `setWindowTitle(tr("Chinese Checkers"));` 设置窗口的标题。`tr()` 函数用于国际化，它会在运行时翻译文本。
- `setFixedSize(2*boardSize, boardSize);` 设置窗口的大小为固定值，这里是棋盘大小的两倍宽度和相同的高度。
- `refleshButton->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed);` 设置按钮的大小策略为固定，这意味着按钮的大小不会随着窗口的改变而改变。
- `refleshButton->setFixedSize(200, 30);` 设置按钮的固定大小。
- `QVBoxLayout *layout = new QVBoxLayout(this);` 创建一个垂直布局管理器，并将其设置为当前窗口的布局。
- `layout->addWidget(refleshButton);` 将新游戏按钮添加到布局中。
- `layout->addStretch();` 在布局中添加一个可伸缩的空间，这样可以将按钮推向布局的顶部或底部。
- 以上只是本次用到的一些布局功能和设置，实际上可以根据界面设计来选择适合的布局设置。

资源和社区

以上介绍的只是Qt的一些基础功能，如果想要进一步深入学习，Qt拥有一个活跃的开发者社区，可以在Qt的官方论坛、Stack Overflow以及GitHub上的社区上进行学习。

可以参考以下资源：

- Qt5编程入门教程：非常详细的Qt5编程入门教程，包括环境搭建、基本概念、常用组件等。
- Qt全部基础知识架构：60分钟内让你掌握Qt的全部基础知识。
- Qt-Tutorial GitHub：包含完整案例与源代码的Qt图形化开发入门系列教程。
- Qt软件开发入门到项目实战PDF：配有完整示例代码的QT5软件开发入门到项目实战教程。
- Qt5编程入门教程：基于Qt5版本的详细入门教程，涵盖多个开发领域。