# 一、基本交互

## 1.需要全局上下文对象的定义，需要QQqmlContext：

(1)在cpp端定义，需要包含头文件：#include<QQqmlContext>

利用一个QqqmlContext对象指针来接收engine对象的根上下文对象

|  |
| --- |
| QQmlContext\* **pContext** = engine.rootContext(); |

(2)创建一个全局上下文对象

|  |
| --- |
| pContext->setContextProperty("SCREEN\_WIDTH",640);//创建全局对象供qml使用 |

## 2.一旦创建了全局上下文对象，在qml文件中就可以直接使用

|  |
| --- |
| Window {  width: SCREEN\_WIDTH  height: Screen.desktopAvailableHeight\*2/3  visible: true  title: *qsTr*("Hello World")  } |

## 3.上面的SCREEN\_WIDTH参数的值是写死的，不好，需要动态获取

#### 注意：这一步是在第一步之前做的

(1)需要包含QScreen头文件: #include<QScreen>

(2)利用Qscreend对象指针接收动态获取的屏幕宽度写法1

|  |
| --- |
| #include <QGuiApplication>  #include <QQmlApplicationEngine>  #include<QQmlContext>  #include<QScreen>  int **main**(int **argc**, char \***argv**[])  {  #if QT\_VERSION < QT\_VERSION\_CHECK(6, 0, 0)  QCoreApplication::setAttribute(Qt::AA\_EnableHighDpiScaling);  #endif  QGuiApplication **app**(*argc*, *argv*);  QQmlApplicationEngine **engine**;  //全局对象 上下文对象  QScreen\* **pScreen** = QGuiApplication::primaryScreen();  QQmlContext\* **pContext** = engine.rootContext();  pContext->setContextProperty("SCREEN\_WIDTH",pScreen->geometry().width()/2);//创建全局对象供qml使用 |

(3)利用Qscreend对象指针接收动态获取的屏幕宽度写法2，需要QRect

|  |
| --- |
| #include <QGuiApplication>  #include <QQmlApplicationEngine>  #include<QQmlContext>  #include<QScreen>  #include<QRect>  int **main**(int **argc**, char \***argv**[])  {  #if QT\_VERSION < QT\_VERSION\_CHECK(6, 0, 0)  QCoreApplication::setAttribute(Qt::AA\_EnableHighDpiScaling);  #endif  QGuiApplication **app**(*argc*, *argv*);  QQmlApplicationEngine **engine**;  //全局对象 上下文对象  QScreen\* **pScreen** = QGuiApplication::primaryScreen();  QRect **rect** = pScreen->virtualGeometry();  QQmlContext\* **pContext** = engine.rootContext();  pContext->setContextProperty("SCREEN\_WIDTH",rect.width()/2);//创建全局对象供qml使用 |

## 4.在qml中有一个叫做QtObject的元素

QtObject的作用：当需要将一个元素的某些属性设置为私有的时候可以通过它来实现，私有化后需要利用属性别名来暴露给外部使用

|  |  |  |
| --- | --- | --- |
| //MyRect.qml  import QtQuick 2.0  import QtQuick.Controls 2.15  Rectangle{  width: 200  height:150  color: "deeppink"  property alias attr: *attrib*  QtObject{  id:*attrib*  property int testVal: 200  }  } | //main.qml  import QtQuick 2.15  import QtQuick.Window 2.15  import QtQuick.Controls 2.15  Window {  width: SCREEN\_WIDTH  height: SCREEN\_HEIGHT  visible: true  title: *qsTr*("Hello World")  MyRect{  Component.onCompleted: *console*.log(*attr*.testVal)  }  } | //Sources/main.cpp  #include <QGuiApplication>  #include <QQmlApplicationEngine>  #include<QQmlContext>  #include<QScreen>  #include<QRect>  int **main**(int **argc**, char \***argv**[])  {  #if QT\_VERSION < QT\_VERSION\_CHECK(6, 0, 0)  QCoreApplication::setAttribute(Qt::AA\_EnableHighDpiScaling);  #endif  QGuiApplication **app**(*argc*, *argv*);  QQmlApplicationEngine **engine**;  //全局对象 上下文对象  QScreen\* **pScreen** = QGuiApplication::primaryScreen();  QRect **rect** = pScreen->virtualGeometry();  QQmlContext\* **pContext** = engine.rootContext();  pContext->setContextProperty("SCREEN\_WIDTH",rect.width()/2);//创建全局对象供qml使用  pContext->setContextProperty("SCREEN\_HEIGHT",rect.height()/2);//创建全局对象供qml使用  const QUrl **url**(QStringLiteral("qrc:/main.qml"));  QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,  &app, [url](QObject \***obj**, const QUrl &**objUrl**) {  if (!obj && url == objUrl)  QCoreApplication::exit(-1);  }, Qt::QueuedConnection);  engine.load(url);  return app.exec();  } |

效果：

|  |
| --- |
|  |

# 二、创建自定义对象

## 1.在c++端创建一个类，继承Qobject

注意：一定要继承自QObject，否则无法在qml中使用结构如下：

|  |
| --- |
|  |

## 2.添加属性，并且添加定义使得qml能够使用：将光标移动到需要添加定义的属性，按alt+enter，选择第一项

|  |
| --- |
|  |

会在自定义类的cpp文件中生成类似下面r的东西

|  |
| --- |
| #include "customobject.h"  CustomObject::**CustomObject**(QObject \***parent**)  : QObject{*parent*}  {  }  int CustomObject::**val**() const  {  return m\_val;  }  void CustomObject::**setVal**(int **newVal**)  {  if (m\_val == newVal)  return;  m\_val = newVal;  emit valChanged();  } |

同理，对m\_str也做相同的处理，处理代码如下：

|  |  |
| --- | --- |
| //.h  #ifndef CUSTOMOBJECT\_H  #define **CUSTOMOBJECT\_H**  #include <QObject>  class **CustomObject** : public QObject  {  Q\_OBJECT  public:  explicit **CustomObject**(QObject \***parent** = nullptr);  //signals:  int **val**() const;  void **setVal**(int **newVal**);  const QString &**str**() const;  void **setStr**(const QString &**newStr**);  signals:  void **valChanged**();  void **strChanged**();  private:  int **m\_val**;  QString **m\_str**;  Q\_PROPERTY(int val READ val WRITE setVal NOTIFY valChanged)  Q\_PROPERTY(QString str READ str WRITE setStr NOTIFY strChanged)  }; | //.cpp  #include "customobject.h"  CustomObject::**CustomObject**(QObject \***parent**)  : QObject{*parent*}  {  }  int CustomObject::**val**() const  {  return m\_val;  }  void CustomObject::**setVal**(int **newVal**)  {  if (m\_val == newVal)  return;  m\_val = newVal;  emit valChanged();  }  const QString &CustomObject::**str**() const  {  return m\_str;  }  void CustomObject::**setStr**(const QString &**newStr**)  {  if (m\_str == newStr)  return;  m\_str = newStr;  emit strChanged();  } |

## 3.做一些处理，使得qml可以使用这个cpp类

在CustomObject类在添加一个静态方法，用来获取CustomObject对象的指针

|  |  |
| --- | --- |
| //customobject.h  #ifndef CUSTOMOBJECT\_H  #define **CUSTOMOBJECT\_H**  #include <QObject>  class **CustomObject** : public QObject  {  Q\_OBJECT  public:  explicit **CustomObject**(QObject \***parent** = nullptr);  static CustomObject\* **getInstance**();  int **val**() const;  void **setVal**(int **newVal**);  const QString &**str**() const;  void **setStr**(const QString &**newStr**);  signals:  void **valChanged**();  void **strChanged**();  private:  int **m\_val**;  QString **m\_str**;  Q\_PROPERTY(int val READ val WRITE setVal NOTIFY valChanged)  Q\_PROPERTY(QString str READ str WRITE setStr NOTIFY strChanged)  }; | //customobject.cpp  #include "customobject.h"  CustomObject::**CustomObject**(QObject \***parent**)  : QObject{*parent*}  {  }  CustomObject \*CustomObject::**getInstance**()  {  static CustomObject\* **pObj** = new CustomObject();  return pObj;  }  int CustomObject::**val**() const  {  return m\_val;  }  void CustomObject::**setVal**(int **newVal**)  {  if (m\_val == newVal)  return;  m\_val = newVal;  emit valChanged();  }  const QString &CustomObject::**str**() const  {  return m\_str;  }  void CustomObject::**setStr**(const QString &**newStr**)  {  if (m\_str == newStr)  return;  m\_str = newStr;  emit strChanged();  } |

## 4.将这个对象设置到全局上下文中：

#### 首先，需要在该类的头文件中添加一个QML\_ELEMENT宏将这个类变为一个元素,需要包含QtQml头文件

|  |
| --- |
| #ifndef CUSTOMOBJECT\_H  #define **CUSTOMOBJECT\_H**  #include <QObject>  #include<QtQml>  class **CustomObject** : public QObject  {  Q\_OBJECT  //声明QML元素  QML\_ELEMENT  public:  explicit **CustomObject**(QObject \***parent** = nullptr);  static CustomObject\* **getInstance**();  int **val**() const;  void **setVal**(int **newVal**);  const QString &**str**() const;  void **setStr**(const QString &**newStr**);  signals:  void **valChanged**();  void **strChanged**();  private:  int **m\_val**;  QString **m\_str**;  Q\_PROPERTY(int val READ val WRITE setVal NOTIFY valChanged)  Q\_PROPERTY(QString str READ str WRITE setStr NOTIFY strChanged)  };  #endif // CUSTOMOBJECT\_H |

上面的Q\_PROPERTY宏也可以这么写

|  |
| --- |
|  |

#### 其次，需要在.pro文件中添加模块声明

|  |
| --- |
| CONFIG += qmltypes  QML\_IMPORT\_NAME = CustomObj  QML\_IMPORT\_MAJOR\_VERSION = 1 |

#### 最后将这个类设置到全局上下文值

这种方法不好，因为当将一个复杂对象设置到上下文中，它就是一个全局对象，这样子可能

|  |
| --- |
| //Sources/main.cpp  #include<QScreen>  #include<QRect>  #include"customobject.h"  int **main**(int **argc**, char \***argv**[])  {  #if QT\_VERSION < QT\_VERSION\_CHECK(6, 0, 0)  QCoreApplication::setAttribute(Qt::AA\_EnableHighDpiScaling);  #endif  QGuiApplication **app**(*argc*, *argv*);  QQmlApplicationEngine **engine**;  //全局对象 上下文对象  QScreen\* **pScreen** = QGuiApplication::primaryScreen();  QRect **rect** = pScreen->virtualGeometry();  QQmlContext\* **pContext** = engine.rootContext();  pContext->setContextProperty("SCREEN\_WIDTH",rect.width()/2);//创建全局对象供qml使用  pContext->setContextProperty("SCREEN\_HEIGHT",rect.height()/2);//创建全局对象供qml使用  //获取CustomObject对象指针，并且设置到全局上下文  pContext->setContextProperty("CustomObject",CustomObject::getInstance());  const QUrl **url**(QStringLiteral("qrc:/main.qml"));  QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,  &app, [url](QObject \***obj**, const QUrl &**objUrl**) {  if (!obj && url == objUrl)  QCoreApplication::exit(-1);  }, Qt::QueuedConnection);  engine.load(url);  return app.exec();  } |

#### 还有更好的方法：

|  |
| --- |
| #include <QGuiApplication>  #include <QQmlApplicationEngine>  #include<QQmlContext>  #include<QScreen>  #include<QRect>  #include"customobject.h"  int **main**(int **argc**, char \***argv**[])  {  #if QT\_VERSION < QT\_VERSION\_CHECK(6, 0, 0)  QCoreApplication::setAttribute(Qt::AA\_EnableHighDpiScaling);  #endif  QGuiApplication **app**(*argc*, *argv*);  QQmlApplicationEngine **engine**;  //全局对象 上下文对象  QScreen\* **pScreen** = QGuiApplication::primaryScreen();  QRect **rect** = pScreen->virtualGeometry();  QQmlContext\* **pContext** = engine.rootContext();  pContext->setContextProperty("SCREEN\_WIDTH",rect.width()/2);//创建全局对象供qml使用  pContext->setContextProperty("SCREEN\_HEIGHT",rect.height()/2);//创建全局对象供qml使用  //获取CustomObject对象指针，并且设置到全局上下文，这个方法不太好  //pContext->setContextProperty("CustomObject",CustomObject::getInstance());  //更好的办法是使用qml注册函数  qmlRegisterType<CustomObject>("CustomObj",1,0,"CustomObject");  const QUrl **url**(QStringLiteral("qrc:/main.qml"));  QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,  &app, [url](QObject \***obj**, const QUrl &**objUrl**) {  if (!obj && url == objUrl)  QCoreApplication::exit(-1);  }, Qt::QueuedConnection);  engine.load(url);  return app.exec();  }  //注意：使用这种方法不需要在pro文件中设置任何东西 |

## 5、在qml在使用，需要import

|  |
| --- |
| //main.qml  import QtQuick 2.15  import QtQuick.Window 2.15  import QtQuick.Controls 2.15  import CustomObj 1.0  Window {  width: SCREEN\_WIDTH  height: SCREEN\_HEIGHT  visible: true  // title: qsTr("Hello World")  title: *cmo*.str  // MyRect{  // Component.onCompleted: console.log(attr.testVal)  // }  //创建自定义对象  CustomObject{  id:*cmo*  str:"qml with cpp"  }  } |

## 本案例完整代码

|  |  |
| --- | --- |
| //main.cpp  #include <QGuiApplication>  #include <QQmlApplicationEngine>  #include<QQmlContext>  #include<QScreen>  #include<QRect>  #include"customobject.h"  int **main**(int **argc**, char \***argv**[])  {  #if QT\_VERSION < QT\_VERSION\_CHECK(6, 0, 0)  QCoreApplication::setAttribute(Qt::AA\_EnableHighDpiScaling);  #endif  QGuiApplication **app**(*argc*, *argv*);  QQmlApplicationEngine **engine**;  //全局对象 上下文对象  QScreen\* **pScreen** = QGuiApplication::primaryScreen();  QRect **rect** = pScreen->virtualGeometry();  QQmlContext\* **pContext** = engine.rootContext();  pContext->setContextProperty("SCREEN\_WIDTH",rect.width()/2);//创建全局对象供qml使用  pContext->setContextProperty("SCREEN\_HEIGHT",rect.height()/2);//创建全局对象供qml使用  //获取CustomObject对象指针，并且设置到全局上下文，这个方法不太好  //pContext->setContextProperty("CustomObject",CustomObject::getInstance());  //更好的办法是使用qml注册函数  qmlRegisterType<CustomObject>("CustomObj",1,0,"CustomObject");  const QUrl **url**(QStringLiteral("qrc:/main.qml"));  QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,  &app, [url](QObject \***obj**, const QUrl &**objUrl**) {  if (!obj && url == objUrl)  QCoreApplication::exit(-1);  }, Qt::QueuedConnection);  engine.load(url);  return app.exec();  } | //customobject.h  #ifndef CUSTOMOBJECT\_H  #define **CUSTOMOBJECT\_H**  #include <QObject>  #include<QtQml>  class **CustomObject** : public QObject  {  Q\_OBJECT  //声明QML元素  QML\_ELEMENT  public:  explicit **CustomObject**(QObject \***parent** = nullptr);  static CustomObject\* **getInstance**();  int **val**() const;  void **setVal**(int **newVal**);  const QString &**str**() const;  void **setStr**(const QString &**newStr**);  signals:  void **valChanged**();  void **strChanged**();  private:  int **m\_val**;  QString **m\_str**;  // Q\_PROPERTY(int val READ val WRITE setVal NOTIFY valChanged)  // Q\_PROPERTY(QString str READ str WRITE setStr NOTIFY strChanged)  Q\_PROPERTY(int val MEMBER m\_val NOTIFY valChanged)  Q\_PROPERTY(QString str MEMBER m\_str NOTIFY strChanged)  };  #endif // CUSTOMOBJECT\_H |
| //customobject.cpp  #include "customobject.h"  CustomObject::**CustomObject**(QObject \***parent**)  : QObject{*parent*}  {  }  CustomObject \*CustomObject::**getInstance**()  {  static CustomObject\* **pObj** = new CustomObject();  return pObj;  }  int CustomObject::**val**() const  {  return m\_val;  }  void CustomObject::**setVal**(int **newVal**)  {  if (m\_val == newVal)  return;  m\_val = newVal;  emit valChanged();  }  const QString &CustomObject::**str**() const  {  return m\_str;  }  void CustomObject::**setStr**(const QString &**newStr**)  {  if (m\_str == newStr)  return;  m\_str = newStr;  emit strChanged();  } | //main.qml  import QtQuick 2.15  import QtQuick.Window 2.15  import QtQuick.Controls 2.15  import CustomObj 1.0  Window {  width: *SCREEN\_WIDTH*  height: *SCREEN\_HEIGHT*  property int value: *cmo*.val  visible: true  // title: qsTr("Hello World")  title: *cmo*.str  // MyRect{  // Component.onCompleted: console.log(attr.testVal)  // }  //创建自定义对象  CustomObject{  id:*cmo*  str:"qml with cpp"  val:10  }  Button{  onClicked:{ *cmo*.val++;if(*cmo*.val===100) *cmo*.val=0}  }  onValueChanged: *console*.log("value= ",*value*)  } |

# 三、qml调用c++函数

## 1.需要在类的头文件中将被qml调用的函数前面添加一个叫做Q\_INVOKABLE的宏

实例（main.cpp的内容没有改变，所以没有摘抄）

|  |  |  |
| --- | --- | --- |
| //customobject.h  #ifndef CUSTOMOBJECT\_H  #define **CUSTOMOBJECT\_H**  #include <QObject>  #include<QtQml>  class **CustomObject** : public QObject  {  Q\_OBJECT  //声明QML元素  QML\_ELEMENT  public:  explicit **CustomObject**(QObject \***parent** = nullptr);  static CustomObject\* **getInstance**();  int **val**() const;  void **setVal**(int **newVal**);  const QString &**str**() const;  void **setStr**(const QString &**newStr**);  //普通函数需要添加宏才能被qml用  Q\_INVOKABLE void **func**();  signals:  void **valChanged**();  void **strChanged**();  private:  int **m\_val**;  QString **m\_str**;  // Q\_PROPERTY(int val READ val WRITE setVal NOTIFY valChanged)  // Q\_PROPERTY(QString str READ str WRITE setStr NOTIFY strChanged)  Q\_PROPERTY(int val MEMBER m\_val NOTIFY valChanged)  Q\_PROPERTY(QString str MEMBER m\_str NOTIFY strChanged)  };  #endif // CUSTOMOBJECT\_H | //customobject.cpp  #include "customobject.h"  #include<QDebug>  CustomObject::**CustomObject**(QObject \***parent**)  : QObject{*parent*}  {  }  CustomObject \*CustomObject::**getInstance**()  {  static CustomObject\* **pObj** = new CustomObject();  return pObj;  }  int CustomObject::**val**() const  {  return m\_val;  }  void CustomObject::**setVal**(int **newVal**)  {  if (m\_val == newVal)  return;  m\_val = newVal;  emit valChanged();  }  const QString &CustomObject::**str**() const  {  return m\_str;  }  void CustomObject::**setStr**(const QString &**newStr**)  {  if (m\_str == newStr)  return;  m\_str = newStr;  emit strChanged();  }  void CustomObject::**func**()  {  qDebug()<<"this is a function inside CustomObject";  } | //main.qml  import QtQuick 2.15  import QtQuick.Window 2.15  import QtQuick.Controls 2.15  import CustomObj 1.0  Window {  width: *SCREEN\_WIDTH*  height: *SCREEN\_HEIGHT*  property int value: *cmo*.val  visible: true  // title: qsTr("Hello World")  title: *cmo*.str  // MyRect{  // Component.onCompleted: console.log(attr.testVal)  // }  //创建自定义对象  CustomObject{  id:*cmo*  str:"qml with cpp"  val:10  }  Button{  id:*btn1*  onClicked:{ *cmo*.val++;if(*cmo*.val===100) *cmo*.val=0}  text: "修改值"  }  onValueChanged: *console*.log("value= ",*value*)  Button{  x:150  text: "运行cpp函数"  onClicked: { //在这里运行CustomObject的func函数  *cmo*.func()  }  }  } |

# 四、qml向cpp端发送信号

## 1.在qml端定义信号并且创建一个按钮来发送信号

|  |
| --- |
| import QtQuick 2.15  import QtQuick.Window 2.15  import QtQuick.Controls 2.15  import CustomObj 1.0  Window {  width: *SCREEN\_WIDTH*  height: *SCREEN\_HEIGHT*  property int value: *cmo*.val  visible: true  // title: qsTr("Hello World")  title: *cmo*.str  // MyRect{  // Component.onCompleted: console.log(attr.testVal)  // }  //创建自定义对象  CustomObject{  id:*cmo*  str:"qml with cpp"  val:10  }  Button{  id:*btn1*  onClicked:{ *cmo*.val++;if(*cmo*.val===100) *cmo*.val=0}  text: "修改值"  }  onValueChanged: *console*.log("value= ",*value*)  Button{  x:120  text: "运行cpp函数"  onClicked: { //在这里运行CustomObject的func函数  *cmo*.func()  }  }  //定义信号  signal qmlSig(int i,string s)  Button{  id:*sender*  x:240  text: "发送信号给cpp"  onClicked: {  *qmlSig*(100,"sigal from qml")//发送信号  }  }  } |

## 2、需要在cpp类中定义一个槽函数来处理信号

|  |  |
| --- | --- |
| //customobject.h  ifndef CUSTOMOBJECT\_H  #define **CUSTOMOBJECT\_H**  #include <QObject>  #include<QtQml>  #include<QString>  class **CustomObject** : public QObject  {  Q\_OBJECT  //声明QML元素  QML\_ELEMENT  public:  explicit **CustomObject**(QObject \***parent** = nullptr);  static CustomObject\* **getInstance**();  int **val**() const;  void **setVal**(int **newVal**);  const QString &**str**() const;  void **setStr**(const QString &**newStr**);  //普通函数  Q\_INVOKABLE void **func**();  signals:  void **valChanged**();  void **strChanged**();  public slots:  void **cppSlot**(int **i**,QString **str**);//槽函数，处理qml发过来的信号  private:  int **m\_val**;  QString **m\_str**;  // Q\_PROPERTY(int val READ val WRITE setVal NOTIFY valChanged)  // Q\_PROPERTY(QString str READ str WRITE setStr NOTIFY strChanged)  Q\_PROPERTY(int val MEMBER m\_val NOTIFY valChanged)  Q\_PROPERTY(QString str MEMBER m\_str NOTIFY strChanged)  };  #endif // CUSTOMOBJECT\_H | //customobject.cpp  #include "customobject.h"  #include<QDebug>  CustomObject::**CustomObject**(QObject \***parent**)  : QObject{*parent*}  {  }  CustomObject \*CustomObject::**getInstance**()  {  static CustomObject\* **pObj** = new CustomObject();  return pObj;  }  int CustomObject::**val**() const  {  return m\_val;  }  void CustomObject::**setVal**(int **newVal**)  {  if (m\_val == newVal)  return;  m\_val = newVal;  emit valChanged();  }  const QString &CustomObject::**str**() const  {  return m\_str;  }  void CustomObject::**setStr**(const QString &**newStr**)  {  if (m\_str == newStr)  return;  m\_str = newStr;  emit strChanged();  }  void CustomObject::**func**()  {  qDebug()<<"this is a function inside CustomObject";  }  void CustomObject::**cppSlot**(int **i**, QString **str**)  {  qDebug()<< \_\_FUNCTION\_\_<< " " << i << " " << str;    } |

# 五、连接信号和槽

## 方法一、在qml中使用Connections元素

|  |  |
| --- | --- |
| import QtQuick 2.15  import QtQuick.Window 2.15  import QtQuick.Controls 2.15  import CustomObj 1.0  Window {  id:*window*  width: *SCREEN\_WIDTH*  height: *SCREEN\_HEIGHT*  property int value: *cmo*.val  visible: true  // title: qsTr("Hello World")  title: *cmo*.str  // MyRect{  // Component.onCompleted: console.log(attr.testVal)  // }  //创建自定义对象  CustomObject{  id:*cmo*  str:"qml with cpp"  val:10  }  Button{  id:*btn1*  onClicked:{ *cmo*.val++;if(*cmo*.val===100) *cmo*.val=0}  text: "修改值"  }  onValueChanged: *console*.log("value= ",*value*)  Button{  x:120  text: "运行cpp函数"  onClicked: { //在这里运行CustomObject的func函数  *cmo*.func()  }  }  //定义信号  signal qmlSig(int i,string s)  Button{  id:*sender*  x:240  text: "发送信号给cpp"  onClicked: {  *qmlSig*(100,"sigal from qml")  }  }  Connections{  target: *window*  function *onQmlSig*(i,s){  *cmo*.cppSlot(*i*,*s*)  }  }  } |  |

## 方法二，使用信号的connect方法

|  |  |
| --- | --- |
| import QtQuick 2.15  import QtQuick.Window 2.15  import QtQuick.Controls 2.15  import CustomObj 1.0  Window {  id:*window*  width: *SCREEN\_WIDTH*  height: *SCREEN\_HEIGHT*  property int value: *cmo*.val  visible: true  // title: qsTr("Hello World")  title: *cmo*.str  // MyRect{  // Component.onCompleted: console.log(attr.testVal)  // }  //创建自定义对象  CustomObject{  id:*cmo*  str:"qml with cpp"  val:10  }  Button{  id:*btn1*  onClicked:{ *cmo*.val++;if(*cmo*.val===100) *cmo*.val=0}  text: "修改值"  }  onValueChanged: *console*.log("value= ",*value*)  Button{  x:120  text: "运行cpp函数"  onClicked: { //在这里运行CustomObject的func函数  *cmo*.func()  }  }  //定义信号  signal qmlSig(int i,string s)  Button{  id:*sender*  x:240  text: "发送信号给cpp"  onClicked: {  *qmlSig*(100,"sigal from qml")  }  }  // Connections{  // target: window  // function onQmlSig(i,s){  // cmo.cppSlot(i,s)  // }  // }  Component.onCompleted: {  *qmlSig*.connect(*cmo*.cppSlot);  }  } |  |

## 方法三、在cpp端绑定，比较复杂，需要在engine加载完成后来做

customobject头文件和cpp文件都没有改变，所以不摘抄

|  |  |
| --- | --- |
| //main.cpp  #include <QGuiApplication>  #include <QQmlApplicationEngine>  #include<QQmlContext>  #include<QScreen>  #include<QRect>  #include"customobject.h"  #include<QDebug>  #include<QObject>  int **main**(int **argc**, char \***argv**[])  {  #if QT\_VERSION < QT\_VERSION\_CHECK(6, 0, 0)  QCoreApplication::setAttribute(Qt::AA\_EnableHighDpiScaling);  #endif  QGuiApplication **app**(*argc*, *argv*);  QQmlApplicationEngine **engine**;  //全局对象 上下文对象  QScreen\* **pScreen** = QGuiApplication::primaryScreen();  QRect **rect** = pScreen->virtualGeometry();  QQmlContext\* **pContext** = engine.rootContext();  pContext->setContextProperty("SCREEN\_WIDTH",rect.width()/2);//创建全局对象供qml使用  pContext->setContextProperty("SCREEN\_HEIGHT",rect.height()/2);//创建全局对象供qml使用  //获取CustomObject对象指针，并且设置到全局上下文，这个方法不太好  //pContext->setContextProperty("CustomObject",CustomObject::getInstance());  //更好的办法是使用qml注册函数  qmlRegisterType<CustomObject>("CustomObj",1,0,"CustomObject");  const QUrl **url**(QStringLiteral("qrc:/main.qml"));  QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,  &app, [url](QObject \***obj**, const QUrl &**objUrl**) {  if (!obj && url == objUrl)  QCoreApplication::exit(-1);  }, Qt::QueuedConnection);  engine.load(url);  //方法三（方法一和方法二在qml文件中），注意一定要做load方法执行之后  auto **list** = engine.rootObjects();  //auto btn = list.first()->findChild<QObject\*>("Button3");  auto **win** = list.first();  //QString name = win->objectName();//"Window",是qml界面的根元素  // qDebug()<<name;  QObject::connect(win,SIGNAL(qmlSig(int, QString)),CustomObject::getInstance(),SLOT(cppSlot(int,QString)));  return app.exec();  } | //main.qml  import QtQuick 2.15  import QtQuick.Window 2.15  import QtQuick.Controls 2.15  import CustomObj 1.0  Window {  id:*window*  width: *SCREEN\_WIDTH*  height: *SCREEN\_HEIGHT*  objectName: "Window"  property int value: *cmo*.val  visible: true  // title: qsTr("Hello World")  title: *cmo*.str  // MyRect{  // Component.onCompleted: console.log(attr.testVal)  // }  //创建自定义对象  CustomObject{  id:*cmo*  objectName: "CustomObject"  str:"qml with cpp"  val:10  }  Button{  id:*btn1*  objectName: "Button1"  onClicked:{ *cmo*.val++;if(*cmo*.val===100) *cmo*.val=0}  text: "修改值"  }  onValueChanged: *console*.log("value= ",*value*)  Button{  x:120  text: "运行cpp函数"  objectName: "Button2"  onClicked: { //在这里运行CustomObject的func函数  *cmo*.func()  }  }  //定义信号  signal qmlSig(int i,string s)  Button{  id:*sender*  x:240  objectName: "Button3"  text: "发送信号给cpp"  onClicked: {  *qmlSig*(100,"sigal from qml")  }  }  //方法1 ok  // Connections{  // target: window  // function onQmlSig(i,s){  // cmo.cppSlot(i,s)  // }  // }  // 方法二OK  // Component.onCompleted: {  // qmlSig.connect(cmo.cppSlot);  // }  } |

效果是一样的

# 六、cpp向qml发送信号：参考cppSendToQml项目

## 1.在qml在定义一个函数用来处理cpp发送过来的信号

|  |
| --- |
| import QtQuick 2.15  import QtQuick.Window 2.15  import QtQuick.Controls 2.15  import CustomObj 1.0  Window {  width: 640  height: 480  visible: true  title: *qsTr*("Hello World")  Button{  x:120  text: "处理cpp信号"  objectName: "Button2"  onClicked: { //在这里运行CustomObject的func函数  CustomObject.sendSig()  }  }  //qml处理cpp信号的函数,当在cpp端连接是时候调用  //方法二、在cpp中绑定，需要下面这个槽函数  function *qmlSlot*(i, s){ //定义qml槽函数，处理cpp发送过来的信号  *console*.log("qmlSlot:",*i*,*s*)  }  //方法一在qml连接  // Connections{  // target:CustomObject  // function onCppSig(i,s){  // console.log("cpp sig:",i,s)  // }  // }  } |

## 2.在cpp的CustomObject类中定义一个信号

|  |  |
| --- | --- |
| //customobject.h  #ifndef CUSTOMOBJECT\_H  #define **CUSTOMOBJECT\_H**  #include <QObject>  #include<QtQml>  #include<QString>  class **CustomObject** : public QObject  {  Q\_OBJECT  //声明QML元素  QML\_ELEMENT  public:  explicit **CustomObject**(QObject \***parent** = nullptr);  static CustomObject\* **getInstance**();  int **val**() const;  void **setVal**(int **newVal**);  const QString &**str**() const;  void **setStr**(const QString &**newStr**);  //普通函数  Q\_INVOKABLE void **func**();  Q\_INVOKABLE void **sendSig**();  signals:  void **valChanged**();  void **strChanged**();  void **cppSig**(QVariant **i**,QVariant **s**);  //public slots:  //void cppSlot(int i,QString str);  private:  int **m\_val**;  QString **m\_str**;  // Q\_PROPERTY(int val READ val WRITE setVal NOTIFY valChanged)  // Q\_PROPERTY(QString str READ str WRITE setStr NOTIFY strChanged)  Q\_PROPERTY(int val MEMBER m\_val NOTIFY valChanged)  Q\_PROPERTY(QString str MEMBER m\_str NOTIFY strChanged)  };  #endif // CUSTOMOBJECT\_H | //customobject.cpp  #include "customobject.h"  #include<QDebug>  CustomObject::**CustomObject**(QObject \***parent**)  : QObject{*parent*}  {  }  CustomObject \*CustomObject::**getInstance**()  {  static CustomObject\* **pObj** = new CustomObject();  return pObj;  }  int CustomObject::**val**() const  {  return m\_val;  }  void CustomObject::**setVal**(int **newVal**)  {  if (m\_val == newVal)  return;  m\_val = newVal;  emit valChanged();  }  const QString &CustomObject::**str**() const  {  return m\_str;  }  void CustomObject::**setStr**(const QString &**newStr**)  {  if (m\_str == newStr)  return;  m\_str = newStr;  emit strChanged();  }  void CustomObject::**func**()  {  qDebug()<<"this is a function inside CustomObject";  }  void CustomObject::**sendSig**() //发送信号  {  emit cppSig(1000,"signal from cpp");  }  //void CustomObject::cppSlot(int i, QString str)  //{  // qDebug()<< \_\_FUNCTION\_\_<< " " << i << " " << str;  //} |

## 3.在qml在用Connections元素绑定或者在main.cpp中用cpp代码绑定

下面的是cpp代码绑定，qml绑定参考qml文件注释部分

|  |
| --- |
| #include <QGuiApplication>  #include <QQmlApplicationEngine>  #include<QQmlContext>  #include<QScreen>  #include<QRect>  #include"customobject.h"  #include<QDebug>  #include<QObject>  int **main**(int **argc**, char \***argv**[])  {  #if QT\_VERSION < QT\_VERSION\_CHECK(6, 0, 0)  QCoreApplication::setAttribute(Qt::AA\_EnableHighDpiScaling);  #endif  QGuiApplication **app**(*argc*, *argv*);  QQmlApplicationEngine **engine**;  //全局对象 上下文对象  QScreen\* **pScreen** = QGuiApplication::primaryScreen();  QRect **rect** = pScreen->virtualGeometry();  QQmlContext\* **pContext** = engine.rootContext();  pContext->setContextProperty("SCREEN\_WIDTH",rect.width()/2);//创建全局对象供qml使用  pContext->setContextProperty("SCREEN\_HEIGHT",rect.height()/2);//创建全局对象供qml使用  //获取CustomObject对象指针，并且设置到全局上下文，这个方法不太好  //pContext->setContextProperty("CustomObject",CustomObject::getInstance());  //更好的办法是使用qml注册函数  qmlRegisterType<CustomObject>("CustomObj",1,0,"CustomObject");  //注册一个全局单例对象  qmlRegisterSingletonInstance("CustomObj",1,0,"CustomObject",CustomObject::getInstance());  const QUrl **url**(QStringLiteral("qrc:/main.qml"));  QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,  &app, [url](QObject \***obj**, const QUrl &**objUrl**) {  if (!obj && url == objUrl)  QCoreApplication::exit(-1);  }, Qt::QueuedConnection);  engine.load(url);  //qml-》cpp方法三（方法一和方法二在qml文件中），注意一定要做load方法执行之后  auto **list** = engine.rootObjects();  auto **win** = list.first();  //QString name = win->objectName();//"Window",是qml界面的根元素  // qDebug()<<name;  //QObject::connect(win,SIGNAL(qmlSig(int, QString)),CustomObject::getInstance(),SLOT(cppSlot(int,QString)));  //cpp-》qml方法2，在cpp中连接  QObject::connect(CustomObject::getInstance(),SIGNAL(cppSig(QVariant,QVariant)),win,SLOT(qmlSlot(QVariant,QVariant)));  return app.exec();  } |

|  |
| --- |
|  |

# 七、cpp直接调用qml函数，参考cppSendToQml项目

## 1、在qml在定义一个函数qmlFunc

|  |
| --- |
| import QtQuick 2.15  import QtQuick.Window 2.15  import QtQuick.Controls 2.15  import CustomObj 1.0  Window {  width: 640  height: 480  visible: true  title: *qsTr*("Hello World")  Button{  x:120  text: "处理cpp信号"  objectName: "Button2"  onClicked: { //在这里运行CustomObject的func函数  CustomObject.sendSig()  }  }  //qml处理cpp信号的函数,当在cpp端连接是时候调用  function *qmlSlot*(i, s){ //定义qml槽函数，处理cpp发送过来的信号  *console*.log("qmlSlot:",*i*,*s*)  }  //方法一在qml连接  // Connections{  // target:CustomObject  // function onCppSig(i,s){  // console.log("cpp sig:",i,s)  // }  // }  function *qmlFunc*(i,s){  return "succeess" +*i* + *s*  }  } |

## 2、在main.cpp中通过QmetaObject的invokeMethod方法来调用qml函数，比较复杂

|  |
| --- |
| #include <QGuiApplication>  #include <QQmlApplicationEngine>  #include<QQmlContext>  #include<QScreen>  #include<QRect>  #include"customobject.h"  //#include "qobjectdefs.h"  #include<QDebug>  #include<QObject>  int **main**(int **argc**, char \***argv**[])  {  #if QT\_VERSION < QT\_VERSION\_CHECK(6, 0, 0)  QCoreApplication::setAttribute(Qt::AA\_EnableHighDpiScaling);  #endif  QGuiApplication **app**(*argc*, *argv*);  QQmlApplicationEngine **engine**;  //全局对象 上下文对象  QScreen\* **pScreen** = QGuiApplication::primaryScreen();  QRect **rect** = pScreen->virtualGeometry();  QQmlContext\* **pContext** = engine.rootContext();  pContext->setContextProperty("SCREEN\_WIDTH",rect.width()/2);//创建全局对象供qml使用  pContext->setContextProperty("SCREEN\_HEIGHT",rect.height()/2);//创建全局对象供qml使用  //获取CustomObject对象指针，并且设置到全局上下文，这个方法不太好  //pContext->setContextProperty("CustomObject",CustomObject::getInstance());  //更好的办法是使用qml注册函数  qmlRegisterType<CustomObject>("CustomObj",1,0,"CustomObject");  //注册一个全局单例对象  qmlRegisterSingletonInstance("CustomObj",1,0,"CustomObject",CustomObject::getInstance());  const QUrl **url**(QStringLiteral("qrc:/main.qml"));  QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,  &app, [url](QObject \***obj**, const QUrl &**objUrl**) {  if (!obj && url == objUrl)  QCoreApplication::exit(-1);  }, Qt::QueuedConnection);  engine.load(url);  //qml-》cpp方法三（方法一和方法二在qml文件中），注意一定要做load方法执行之后  auto **list** = engine.rootObjects();  auto **win** = list.first();  //QString name = win->objectName();//"Window",是qml界面的根元素  // qDebug()<<name;  //QObject::connect(win,SIGNAL(qmlSig(int, QString)),CustomObject::getInstance(),SLOT(cppSlot(int,QString)));  //cpp-》qml方法2，在cpp中连接  QObject::connect(CustomObject::getInstance(),SIGNAL(cppSig(QVariant,QVariant)),win,SLOT(qmlSlot(QVariant,QVariant)));  //注意：JavaScript中的数据类型对应于qt cpp的Qvariant类型  QVariant **ret**;  QVariant **arg1**=1000;  QVariant **arg2** = "cpp Call qml function";  QMetaObject::invokeMethod(*win*,"qmlFunc",Q\_RETURN\_ARG(QVariant,*ret*),Q\_ARG(QVariant,arg1),Q\_ARG(QVariant,arg2));  qDebug()<<ret;  return app.exec();  } |