1. **UI与代码的区别**

.ui优点：

1、ui更方便，更快捷，在界面复杂度不高且不需要多平台支持的工程中，使用.ui优势更大。

2、.ui文件实现的界面，在实现国际化多语言的时候，代码量更少。

.ui缺点：

1、难以跨编码不同的平台。

2、在复杂度高的界面项目中，.ui难以实现，例如多层布局嵌套且还存在大量嵌套外控件的项目，例如多尺寸界面。

3、.ui性能相对低于代码实现，因为在ui\_xxx.h中存在国际化相关代码。

4、存在多项目存在相同QWidget提升时，会出现编译问题。

5、.ui文件不支持二进制兼容。

1. **主窗口布局的区别**

**1、QMainWindow 类**  
QMainWindow 类提供一个菜单条、锚接窗口（如工具栏）和一个状态条的主应用程序窗口。主窗口通常用在提供一个大的中央窗口部件以及周围菜单、工具条和一个状态条。QMainWindow常常被继承，因为这使的封装中央部件、菜单和工具以及窗口状态条变得容易，当用户点击菜单项或工具条按钮时，槽会被调用。基于主窗口的应用程序，默认已经有了自己的布局管理器。

**2、QDialog类是对话框窗口的基类。**  
对话框窗口是主要用于短时期任务以及用户进行简要通讯的顶级窗口。QDialog可以是模态对话框也可以是非模态对话框。QDialog支持扩展性并且可以提供返回值。他们可以有默认按钮。QDialog也可以有一个QSizeGrip在它的右下方，使用setSizeGripEnable（）。  
注意：QDialog使用父窗口部件的方法和Qt中其他类不同。对话框总是顶级窗口部件，但是如果它有一个父对象，它的默认位置就是父对象的中间。他也将和父对象共享工具条条目。  
QDialog 是最普通的顶级窗口（一个不会被嵌入到父窗口部件的窗口部件叫顶级窗口部件）。通常情况下，顶级窗口部件是有框架和标题栏的窗口（尽管使用了一定的窗口部件标记，创建顶级窗口部件时也可以没有这个修饰）在Qt中。QMainWindow和不同的QDialog的子类是最普通的顶级窗口。  
非顶级窗口部件就是子窗口部件。他们是他们的父窗口部件中的子窗口。你通常不能在视觉角度从它们的父窗口部件辨别一个子窗口部件。在Qt中的绝大多数其他窗口部件仅仅作为子窗口部件才是有用的。（当然把一个按钮作为或者叫做顶级窗口部件也是有可能的，但是绝大多数人喜欢把它们的按钮放到其他部件当中）  
如果是顶级对话框，那就是基于QDialog创建，如果是主窗体，就基于QMainWindow，如果不确定，或有可能作为顶级窗体，或有可能嵌入到其他窗体中，则基于QWidget创建。

**3、QFrame类是有框架的窗口部件的基类。**  
它绘制部件并且调用一个虚函数drawContents（）函数来填充这个框架。这个函数是被子类重新实现的。QFrame类也可以之间创建没有任何内容的简单框架，尽管通常情况下，要用到QHBox 或QVBox，因为它们可以自动布置你放到框架的窗口部件。

**三、lambda表达式：**

Lambda表达式, 匿名函数对象，其可以理解为没有函数名的函数，是C++11增加的新特性。

在Qt中，Lambda表达式常和信号一起使用，非常方便，需要在项目文件中添加： CONFIG += C++11

首先看一下Lambda表达式的基本构成：

[capture](parameters) mutable ->return-type

{

statement

}

[函数对象参数] (操作符重载函数参数)mutable ->返回值{函数体}

**函数对象参数**

[]，标识一个Lambda表达式的开始，不能省略。函数对象参数是传递给编译器自动生成的函数对象类的构造函数的。函数对象参数只能使用Lambda之前的Lambda所在范围内可见的局部变量（包括Lambda所在类的this）。

函数对象参数有以下形式：

函数对象参数 说明

[] 不使用任何变量

[=] 以传值的方式使用所有外部变量

[&] 以传引用的方式使用所有外部变量

[x] 将x以传值的方式使用。使用时，函数体内不能修改传递进来的x的拷贝(默认情况下函数是const)，添加mutable修饰符可对传递进来的x的拷贝进行修改

[x,&y] x以传值的方式使用，y以传引用的方式使用

[=,&x] x以传引用的方式使用，其余变量以传值的方式使用

[&,x] x以传值的方式使用，其余变量以传引用的方式使用

2、**操作符重载函数参数**

标识重载的()操作符的参数，没有参数时，这部分可以省略。

参数可以通过按值（如：(a,b)）和按引用（如：(&a,&b)）两种方式进行传递。

3、**mutable**

mutable声明，这部分可以省略。

4、**返回值**

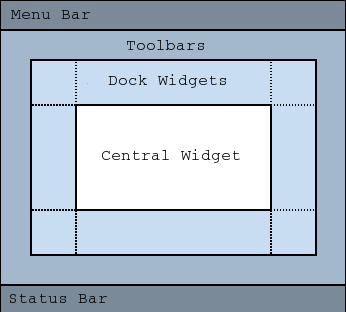
->返回值类型，标识函数返回值的类型，当返回值为void，或者函数体中只有一处return的地方（此时编译器可以自动推断出返回值类型）时，这部分可以省略。

5、**{函数体}**

{}，标识函数的实现，这部分不能省略，但函数体可以为空。

**四、QMainWindow**

*https://blog.csdn.net/weixin\_58176527/article/details/127492141*



1. 菜单栏

例：

QMenuBar \*bar = menuBar();//创建菜单栏,菜单栏只能有一个

setMenuBar(bar);//将菜单栏放入到窗口;

QMenu \* fileMenu=bar->addMenu("文件"); //创建菜单

QMenu \* editMenu=bar->addMenu("编辑"); //返回的是QMenu \*

fileMenu->addAction("新建");//创建菜单项目，返回的是QAction \*

fileMenu->addSeparator();//添加分隔线

fileMenu->addAction("打开");

1. 工具栏

例：

QToolBar \*toolBar = new QToolBar(this); //工具栏 ,可以有多个

//将工具栏放在窗口中addToolBar

addToolBar(Qt::LeftToolBarArea,toolBar);//参数一：自定义工具栏默认位置

//后期设置 只允许 左右停靠

toolBar->setAllowedAreas(Qt::LeftToolBarArea|Qt::RightToolBarArea);

//设置浮动操作

toolBar->setFloatable(0);//bool类型，0代表不允许浮动

//设置移动（总开关）

toolBar->setMovable(0);//bool型，0-不允许移动，1-允许

toolBar->addAction("新建"); //工具栏中可以 设置内容

toolBar->addSeparator();//分割线

toolBar->addAction("打开");

QPushButton \*btn = new QPushButton("open",this);

toolBar->addWidget(btn);//工具栏中添加控件

1. 状态栏：

例：

QStatusBar \*stBar= statusBar(); //状态栏 最多只能有一个

setStatusBar(stBar); //设置在窗口中

QLabel \*label = new QLabel("提示信息",this); //放标签控件

stBar->addWidget(label);//左侧

QLabel \*label2 = new QLabel("右侧提示信息",this);

stBar->addPermanentWidget(label2);//右侧

1. 铆接部件（浮动窗口）

例：

QDockWidget \*dockwidget = new QDockWidget("浮动",this);//铆接部件（浮动窗口） 可以有多个

addDockWidget(Qt::BottomDockWidgetArea,dockwidget);

//设置后期停靠区域，只允许上下

dockwidget->setAllowedAreas(Qt::TopDockWidgetArea|Qt::BottomDockWidgetArea);

1. 中心部件

//设置中心部件 只能有一个

QTextEdit \* edit = new QTextEdit(this);

setCentralWidget(edit);

注：QMainWindow拥有默认布局，使用布局对象嵌套于中心部件中，可以将其他布局实现于QMainWindow中。

1. **信号与槽：**

**优点：**

1. 参数安全
2. 松散耦合

**缺点**：速度慢，约等于回调函数的10倍

**写法：**

1. 标准：

connect(this,SIGNAL(scoreChanged(int)),this,SLOT(onScoreChanged(int)));2、2、指针：

connect(this,&BateObject::scoreChanged,this,&BateObject::onScoreChanged);

3、lambda表达式

见上面（三）

**常见问题分析：**

1、QObject::connect: No such slot xxx错误提示：

这是因为在自己的派生类中没有加入Q\_OBJECT.  
只有加入Q\_OBJECT才能允许自己的派生类中添加信号与槽机制。

1. undefined reference to `vtable for’”问题的原因及解决方法

删除已创建的编译/构建的所有文件，重新编译即可

**多线程环境下， 如何控制信号槽在什么样的线程中执行？**

通过connect(...)第5个参数控制信号槽执行所在线程

connect(...)它的连接方式：队列连接、自动连接和直接连接

队列连接 Qt::QueuedConnection

信号发出后，信号会暂时被放到一个消息队列中，需等到接收对象所属线程的事件循环取得控制权时才取得该信号，然后执行和信号关联的槽函数，这种方式既可以在同一线程内传递消息也可以跨线程操作。

emit语句后的代码将在发出信号后立即被执行，**无需等待槽函数执行完毕**

槽函数在接收者所依附线程执行。

自动连接 Qt::AutoConnection

默认值，使用这个值则连接类型会在信号发送时决定。如果接收者和发送者在同一个线程，则自动使用Qt::DirectConnection类型。如果接收者和发送者不在一个线程，则自动使用Qt::QueuedConnection类型

直接连接 Qt::DirectConnection

槽函数会在信号发送的时候直接被调用，槽函数和信号发送者在同一线程。效果看上去就像是直接在信号发送位置调用了槽函数，效果上看起来像函数调用，同步执行。

emit语句后面的代码将在与信号关联的所有槽函数执行完毕后才被执行。

无论槽函数所属对象在哪个线程，槽函数都在发射信号的线程内执行。

BlockingQueuedConnection

槽函数的调用时机与Qt::QueuedConnection一致，不过发送完信号后发送者所在线程会**阻塞**，直到槽函数运行完。而且接收者和发送者绝对不能在一个线程，否则程序会死锁。在多线程间需要同步的场合可能需要这个。

UniqueConnection 配合前四种使用，确保相同的信号，相同的槽保持唯一连接。为了避免重复连接。

例如：connect(....., Qt::ConnectionType(Qt::AutoConnection | Qt::UniqueConnection))

1. **互斥量问题：**
2. QMutex

互斥量在使用过程中，对代码进行封锁，使同一时间只有一个线程能过执行这段代码

例如：

QMutex mutex;

int creatKey() { mutex.lock(); ++key; return key; mutex. unlock();}

int value() { mutex.lock(); return key; mutex.unlock() ; }

问题分析：首先C++中的++操作非原子操作，会出现线程安全问题，故使用互斥量封锁这部分代码，但是案例中的操作不得不将unlock()置于return之后，导致unlock永远无法执行，导致死锁。

1. QMutexlocker类

Qt 提供的 QMutexLocker 类可以简化互斥 的处理，它在构造函数中接收 QMutex 对象作为参数并将其锁定，在析构函数中解锁这个互斥 ，这样就解决了以上问题。

例如：

int creatKey() (QMutexLocker locker(&mutex); ++key; return key; }

int value() const {QMutexLocker locker(&mutex); return key; }

locker() 函数作为局部变量会在函数退出时结束其作用域，从而自动对互斥量 mutex 解锁

1. **信号量 QSemaphore**

信号 可以理解为对互斥功能的扩展，互斥量只能锁定1次而信号量可以获取多次，它可以用来保护一定数的同种资源。信号量的典型用例是控制生产者／消费者之间共享的环形缓冲区

生产者／消费者实例中对同步的需求有两处：

(1) 如果生产者过快地生产数据，将会覆盖消费者还没有读取的数据。

(2) 如果消费者过快地读取数据，将越过生产者并且读取到些过期数据。

针对以上问题，有两种解决方法：

(1) 首先使生产者填满整个缓冲区，然后等待消费者读取整个缓冲区，这是 种比较笨拙

的方法

(2) 使生产者和消费者线程同时分别操作缓冲区的不同部分，这是种比较高

效的方法

例如：

QSemaphore freeBytes(BufferSize);

QSemaphore useBytes(0);

void Producer::run()

{

for(int i=0;i<DataSize;i++){

freeBytes.acquire(); //申请空闲单元

buffer[i%BufferSize]=(i%BufferSize);

useBytes.release(); //可用资源扩充

}

}

void Consumer::run()

{

fprintf(stderr,"开始了");

for(int i=0;i<DataSize;i++){

useBytes.acquire(); //可用资源申请使用

qDebug()<<buffer[i%BufferSize];

if(i%16==0&&i!=0){

qDebug()<<"\n";

}

freeBytes.release(); //空闲单元扩充

}

qDebug()<<"\n";

}

1. **线程等待与唤醒**

QWaitCondition类

例如：

QWaitCondition bufferEmpty;

QWaitCondition bufferFull;

void Producer::***run***()

{

for(int i=0;i<DataSize;i++){

mutex.lock();//互斥锁

if(numUsedBytes==BufferSize){

//如果生产队列满了，释放互斥锁并等待

bufferEmpty.wait(&mutex);

}

buffer[i%BufferSize]=numUsedBytes;

++numUsedBytes;

//唤醒所有bufferFull锁

bufferFull.wakeAll();

mutex.unlock();

}

}

void Consumer::***run***()

{

for(int i=0;i<DataSize;i++){

mutex.lock();

if(numUsedBytes==0){

bufferFull.wait(&mutex);

}

printf("%u1::[%d]=%d\n",currentThreadId(),rIndex,buffer[rIndex]);

rIndex=(++rIndex)%BufferSize;

--numUsedBytes;

bufferEmpty.wakeAll();

mutex.unlock();

}

qDebug()<<"\n";

}

1. **emit**

emit是用来发射信号的，这个信号如果要作为action或者其他用途就需要用到connect，connect的作用是把信号与信号or信号和槽关联起来。

Qt中信号（SIGNAL）的本质:

* 信号指示一个特殊的成员函数声明
* 函数的返回值是void类型
* 函数只能声明，不能定义
* 信号必须使用signals关键字进行声明
* 函数的访问属性自动被设置为protected
* 只能通过emit关键字调用函数（发射信号)

例如:

Class{

signals://类中signals关键字定义信号

void error(QTcpSocket::SocketError socketError);

}

Function{

//调用信号时通过emit直接调用并传入参数

emit error(tcpSocket.error());

}

1. **数据库**

1、**QT+= sql**

使用qt的sql模块

2、创建连接

**QSqlDatabase db =QSqlDatabase::addDatabase("QSQLITE");**

**db.setHostName("easybook 3313b0"); // 设置数据库主机名**

**db.setDatabaseName ("qtDB. db") ;//设置数据库名**

**db.setUserName("zhouh jun"); // 设置数据库用户名**

**db.setPassword("123456"); //设置数据库密码**

**db.open(); //打开连接**

其中，静态函数 QSqlDatabase: :addDatabase返回一条新建立的数据库连接，其原型为：

**QSqlDatabase::addDatabase**

**{**

**const OString &type,**

**const QString &connectionName=QLatinlString(defaultConnection)**

**}**

* 参数 type 为驱动名，本例使用的是 QSQLITE 驱动。
* 参数 connectionName 为连接名，默认值为默认连接，本例的连接名为 connect 。如果没有指定此参数，则新建立的数据库连接将成为本程序的默认连接，并且可以被后续不带参数的函数 databaseO 引用。如果指定了此参数（连接名），则函数 database (connectionName) 将获取这个指定的数据库连接。

3、**QSqlQuery:**

创建 QSqlQuery 对象。 QtSql 模块中的 QSqlQuery 类提供了个执行SQL 语句的接口，并且可以遍历执行的返回结果集。除 QSqlQuery 类外， Qt还提供了三种用于访问数据库的高层类，即 QSqlQueryModel QSqlTableModel QSqlRelationTableModel 。它们无须使用 SQL 语句就可以进行数据库操作，而且可以很容易地将结果在表格中表示出来。

类名

**QSqlQueryModel 基于任意 SQL 语句的只读模型**

**QsqlTableModel 基于于单个表的读写模型**

**QSqlRelationalTableModel QSqlTableModel 的子类，增加了外键支待**

**执行sql**：

QSqlQuery query

1. query.exec(“需要执行的sql语句”)

返回bool类型

1. 预处理

**void** QSqlQuery::**bindValue**(**int** *pos*, const **[QVariant](../qtcore/qvariant.html)** &*val*, **[QSql::ParamType](qsql.html" \l "ParamTypeFlag-enum)** *paramType* = QSql::In)

query.prepare("insert into automobilvalues(?,?,?,?,?,?,?,?,?,?)");

query.bindValue(0,i);

query.bindValue(1," 四轮") ;

query.bindValue(2," 轿车") ;

query.bindValue(3," 富康") ;

query.bindValue(4,rand()%100);

query. bindValue (5, rand() %10000);

query .bindValue (6, rand() %300) ;

query .bindValue (7, rand() %200000);

query .bindValue (8, rand() %52);

query .bindValue (9, rand() %100);

或者：

**void** QSqlQuery::**bindValue**(const **[QString](../qtcore/qstring.html)** &*placeholder*, const **[QVariant](../qtcore/qvariant.html)** &*val*, **[QSql::ParamType](qsql.html" \l "ParamTypeFlag-enum)** *paramType* = QSql::In)

query.prepare("insert into automobile(id,attribute,type,kind,nation,

carnumber,elevaltor,dis ance,oil,temperature)

values(:id, :attribute, :type, :kind, :nation,

: carnumber, :elevaltor,:distance,:oil,:temperature)");

long records=lOO;

query .bindValue (: id, i);

query.bindValue(:attribute," 四轮") ;

query.bindValue (:type," 轿车") ;

query.bindValue(:kind," 富康") ;

query .bindValue (: nation, rand() %100);

query .bindValue (: carnumber, rand() %10000);

query. bindValue (: eleval tor, rand() %300) ;

query. bindValue (: distance, rand() %200000);

query. bindValue (: oil, rand() %52);

query .bindValue (: temperature, rand() %100);

1. **双缓冲机制**

所谓双缓冲机制，是指在绘制控件时，首先将要绘制的内容绘制在一个图片中，再将图片一次性地绘制到控件上。在早期的 Qt 版本中，若直接在控件上进行绘制工作，则在控件重绘时会产生闪烁的现象，控件重绘频繁时，闪烁尤为明显。双缓冲机制可以有效地消除这种闪烁现象。自Qt 版本之后， QWidget 控件已经能够自动处理闪烁的问题。因此，在控件上直接绘图时，不用再操心显示的闪烁问题，但双缓冲机制在很多场合仍然有其用武之地。当所需绘制的内容较复杂并需要频繁刷新，或者每次只需要刷新整个控件的一小部分时，仍应尽量采用双缓冲机制。

1. **udp与tcp通讯流程**
2. tcp:

创建tcpserver对象->

执行listen(const QHostAddress &address = QHostAddress::Any, quint16 port = 0)，绑定ip和端口进行监听连接请求->

接受客户端连接请求->

创建tcpSocket接受客户端请求数据->

服务端数据处理->

通过上述tcpSocket返回结果给客户端->

接收到客户端断开连接请求->

连接断开

1. udp:

数据发送方:

创建udpSocket->

封装数据包->

执行writeDatagram(const char \*data, qint64 size, const QHostAddress &address, quint16 port)，发送数据包->结束

数据接收方:

创建udpSocket->

绑定端口->

readyRead()->

接受数据包并处理->结束

1. **C++与Qt的智能指针**

智能指针：

智能指针是C++语言中的一种工具，旨在提供更加安全、简单和方便的内存管理方式。智能指针实际上是一种对象，其在内部管理着一个指针，该指针指向某个动态分配的内存块。

智能指针的实现依赖于C++语言的RAII（资源获取即初始化）技术，即资源的获取和释放应该与对象的构造和析构分别相关联。这意味着，在对象构造时分配资源，而在对象析构时释放资源，可以确保资源的正确使用。

智能指针的原理就是将动态分配的内存块与一个或多个智能指针对象相关联，以确保内存块在不再需要时能够自动释放。通常情况下，智能指针会将内存的所有权转移到其自己的对象中，并在其析构函数中释放内存。这样就可以避免常见的内存泄漏和释放非法内存等问题。

智能指针在实现时通常使用引用计数技术，即每个智能指针对象都包含一个计数器，用于记录有多少个智能指针对象引用了同一个内存块。当引用计数器为0时，说明没有任何智能指针对象引用该内存块，此时内存块就可以被释放。

需要注意的是，由于引用计数技术可能存在循环引用的问题，即两个或多个智能指针对象相互引用，导致其引用计数器无法归零，从而导致内存泄漏。为了解决这个问题，C++11标准引入了weak\_ptr，可以解决循环引用的问题。

**C++**: #include<memory>

unique\_ptr:独占式智能指针

确保了只有一个指针可以指向资源，可以通过std::move()函数将资源的所有权转移给其他unique\_ptr对象

例如： std::unique\_ptr<int> p(new int(32)) //创建一个int类型资源

p.release() //释放资产所有权

p.reset(new int(23)) //重置资源所有权

shared\_ptr:共享式智能指针（强引用）

可以被多个指针共享同一个资源，资源的引用计数会被自动管理，当所有的shared\_ptr对象都不再需要该资源时，资源会自动被销毁。

例如： std::shared\_ptr<int> p(new int(32)) //创建一个int类型资源

p.reset(new int(23)) //重置资源所有权

使用场景：

1. 有多个使用者共同使用同一个对象，而这个对象没有一个明确的拥有者；

这个在多线程操作中使用shared\_ptr很方便，可以很自然的用这个指针而不能担心内存泄漏的问题

1. 某一个对象的复制操作很费时
2. 要把指针存入标准库容器时

weak\_ptr：弱引用智能指针

这个指针一般不单独使用，是和shared\_ptr搭配使用的，甚至于，我们可以将 weak\_ptr 类型指针视为 shared\_ptr 指针的一种辅助工具，借助 weak\_ptr 类型指针， 我们可以获取 shared\_ptr 指针的一些状态信息，比如有多少指向相同的 shared\_ptr 指针、shared\_ptr 指针指向的堆内存是否已经被释放等等。

当 weak\_ptr 类型指针的指向和某一shared\_ptr 指针相同时，weak\_ptr 指针并不会使所指堆内存的引用计数加1；同样，当 weak\_ptr 指针被释放时，之前所指堆内存的引用计数也不会因此而减 1。也就是说，weak\_ptr 类型指针并不会影响所指堆内存空间的引用计数。

使用场景：强指针项目引用，资源泄露

解决方法：强指针之间使用weak\_ptr引用

**Qt**:

QPointer:一个被保护的指针，行为类似于普通的c++指针T \*，会在被引用的对象被销毁时自动清除。当需要存储一个指向其他地方（可能是类或者他人需要修改的指针）拥有的QObject的指针时，这时使用保护指针非常合适，如果此时程序中仍然持有该内存区域的引用时，该指针区域可能会在其他的地方被销毁。那么使用保护指针可以安全地测试指针的有效性，从而避免使用无效的、非法的或者悬空指针。

场景：

在多个不同地方的指针指向同一个对象, 当一个地方delete了这个对象后, 其他地方依然会使用指向这个对象的指针, 此时如果没有使用QPointer封装, if (pointer)返回的是true, 而如果使用QPointer封装, QPointer检测到对象被销毁那么if (pointer)返回的是false。

QSharedPointer：

类似于shared\_ptr,引用计数器

SharedPointer 是线程安全的，因此即使有多个线程同时修改 QSharedPointer 对象也不需要加锁。这里要特别说明一下，虽然 QSharedPointer 是线程安全的，但是 QSharedPointer 指向的内存区域可不一定是线程安全的。所以多个线程同时修改 QSharedPointer 指向的数据时还要应该考虑加锁的

QScopedPointer:

类似 unique\_ptr，当我们的内存数据只在一处被使用，用完就可以安全的释放时就可以使用

QScopedArrayPointer：

如果我们指向的内存数据是一个数组，这时可以用 QScopedArrayPointer。QScopedArrayPointer 与 QScopedPointer 类似，用于简单的场景

QSharedDataPointer：

QSharedDataPointer 这个类是帮我们实现数据的隐式共享的。我们知道 Qt 中大量的采用了隐式共享和写时拷贝技术。Qt 中隐式共享和写时拷贝就是利用 QSharedDataPointer 和 QSharedData 这两个类来实现的。

1. **Qt强制类型转换**

|  |  |
| --- | --- |
| const\_cast | 去const属性 |
| static\_cast | 静态类型转换，如int转换成char |
| dynamic\_cast | 动态类型转换，如子类和父类之间的多态类型转换 |
| reinterpret\_cast | 仅仅重新解释类型，但没有进行二进制的转换 |
| qobject\_cast |  |

const\_cast：用于const和volatile的转换。

const\_cast<类型>（要转换的对象）

·常量指针被转化成非常量的指针，并且仍然指向原来的对象。

·常量引用被转换成非常量的引用，并且仍然指向原来的对象。

·const\_cast一般用于修改底指针。如const char \*p形式。

static\_cast：

·用于类层次结构中基类（父类）和派生类（子类）之间指针或引用的转换；进行上行转换（把派生类的指针或引用转换成基类表示）是安全的；进行下行转换（把基类指针或引用转换成派生类表示）时，由于没有动态类型检查，所以是不安全的。

·数据类型之间的转换，如把int转换成char，把int转换成enum。这种转换的安全性也要开发人员来保证。

·把空指针转换成目标类型的空指针。

·把任何类型的表达式转换成void类型。

·static\_cast不能转换掉expression的const、volatile、或者\_\_unaligned属性

reinterpret\_cast：

用于两个对象之间没有任何关系时进行转换。

dynamic\_cast：

用于转换基类和派生类

1. **文件流与数据流**

QTextStream: 文件流

一个用于读写文本的 Qt 类，可以从文件、字节数组、标准输入输出设备等读取和写入文本

1. 构造函数：

QFile file("example.txt");

if (!file.open(QIODevice::ReadOnly | QIODevice::Text))

return;

QTextStream in(&file); // 从文件读取文本

1. 文本读取

QChar c = in.read(); // 读取单个字符

QString word = in.readWord(); // 读取一个单词

QString line = in.readLine(); // 读取一行文本

1. 写入

out << "Hello, world!" << endl; // 写入字符串和换行符

out << 42 << " is the answer." << endl; // 写入数值和字符串

1. 格式化输出

out.setFieldWidth(10);

out.setPadChar('-');

out.setRealNumberPrecision(4);

out << 3.14159265358979323846 << endl; // 输出：--3.1416

1. 字符编码设置

in.setCodec("UTF-8"); // 设置输入流的字符编码为 UTF-8

out.setCodec("GBK"); // 设置输出流的字符编码为 GBK

QDataStream: 数据流

通过数据流可以操作各种数据类型，包括类对象，存储到文件中数据可以还原到内存

QDataStream 实现了基本的 C++数据类型的序列化，比如 char，short，int，char \*等。更复杂的数据类型的序列化是通过分解原始单元来完成的

QDataStream 支持的 Qt 类型有 QBrush、QColor、QDateTime、QFont、QPixmap、QString、QVariant 等类型，还包括容器类型，比如 QList、QVector、QSet、QMap 等

对于整数，建议始终转换为 Qt 整数类型(比如 qint32 等)进入写入，并将其读入为相同的Qt整数类型，这样可以确保获取确定的大小的整数，以避免编译器和平台差异的影响(注：C++语法只规定了 int，short 等类型的最小长度，未规定最大长度)

1. **互斥锁、读写锁、自旋锁**

1、读写锁：

QReadWriteLock（读写锁）：用于实现读写分离的锁机制，允许多个线程同时读取共享资源，但只有一个线程可以进行写入操作。可以通过lockForRead()和lockForWrite()手动控制读取锁和写入锁。

QReadLocker和QWriteLocker：是QReadWriteLock的RAII类，用于简化对QReadWriteLock的加读锁和加写锁过程。在创建QReadLocker或QWriteLocker对象时，会自动加读锁或加写锁，离开作用域时会自动解锁，确保资源在合适的时候被解锁。

int main(int argc, char \*argv[])

{

QCoreApplication a(argc, argv);

QReadWriteLock rwLock;

int data = 0;

// 读取操作

for (int i = 0; i < 100; ++i) {

QReadLocker locker(&rwLock);

qDebug() << "Read data:" << data;

}

// 写入操作

for (int i = 0; i < 10; ++i) {

QWriteLocker locker(&rwLock);

data += 10;

qDebug() << "Write data:" << data;

}

return a.exec();

}

其余锁类型见上文

1. **多线程实现方法**

1、继承QThread，重载run函数。

2、继承QObject，调用void QObject::moveToThread(QThread \*targetThread)。

3、QThreadPool and QRunnabl。

4、Qt Concurrent。

1. **事件驱动。事件类型**
2. **事件过滤，5种**
3. **QTextEdit**

富文本控件