1、讲述Qt信号槽机制与优势与不足

优点： ①类型安全。需要关联的信号槽的签名必须是等同的。即信号的参数类型和参数个数同接受该信号的槽的参数类型和参数个数相同。若信号和槽签名不一致，编译器会报错。

②松散耦合。信号和槽机制减弱了Qt对象的耦合度。激发信号的Qt对象无需知道是那个对象的那个信号槽接收它发出的信号，它只需在适当的时间发送适当的信号即可，而不需要关心是否被接受和那个对象接受了。Qt就保证了适当的槽得到了调用，即使关联的对象在运行时被删除。程序也不会奔溃。

③灵活性。一个信号可以关联多个槽，或多个信号关联同一个槽。

不足：速度较慢。与回调函数相比，信号和槽机制运行速度比直接调用非虚函数慢10倍。

原因：①需要定位接收信号的对象。②安全地遍历所有关联槽。③编组、解组传递参数。④多线程的时候，信号需要排队等待。（然而，与创建对象的new操作及删除对象的delete操作相比，信号和槽的运行代价只是他们很少的一部分。信号和槽机制导致的这点性能损耗，对实时应用程序是可以忽略的。）

2、Qt信号和槽的本质是什么

回调函数。信号或是传递值，或是传递动作变化；槽函数响应信号或是接收值，或者根据动作变化来做出对应操作。

3、描述Qt中的文件流(QTextStream)和数据流(QDataStream)的区别

文件流(QTextStream)。操作轻量级数据（int,double,QString）数据写入文本件中以后以文本的方式呈现。

数据流(QDataStream)。通过数据流可以操作各种数据类型，包括对象，存储到文件中数据为二进制。

文件流，数据流都可以操作磁盘文件，也可以操作内存数据。通过流对象可以将对象打包到内存，进行数据的传输。

4、描述Qt的TCP通讯流程

服务端：（QTcpServer）

①创建QTcpServer对象

②监听list需要的参数是地址和端口号

③当有新的客户端连接成功回发送newConnect信号

④在newConnection信号槽函数中，调用nextPendingConnection函数获取新连接QTcpSocket对象

⑤连接QTcpSocket对象的readRead信号

⑥在readRead信号的槽函数使用read接收数据

⑦调用write成员函数发送数据

客户端：（QTcpSocket）

①创建QTcpSocket对象

②当对象与Server连接成功时会发送connected 信号

③调用成员函数connectToHost连接服务器，需要的参数是地址和端口号

④connected信号的槽函数开启发送数据

⑤使用write发送数据，read接收数据

5、 描述UDP 之 UdpSocket通讯

UDP（User Datagram Protocol即用户数据报协议）是一个轻量级的，不可靠的，面向数据报的无连接协议。在网络质量令人十分不满意的环境下，UDP协议数据包丢失严重。由于UDP的特性：它不属于连接型协议，因而具有资源消耗小，处理速度快的优点，所以通常音频、视频和普通数据在传送时使用UDP较多，因为它们即使偶尔丢失一两个数据包，也不会对接收结果产生太大影响。所以QQ这种对保密要求并不太高的聊天程序就是使用的UDP协议。

在Qt中提供了QUdpSocket 类来进行UDP数据报（datagrams）的发送和接收。Socket简单地说，就是一个IP地址加一个port端口 。

流程：①创建QUdpSocket套接字对象 ②如果需要接收数据，必须绑定端口 ③发送数据用 writeDatagram，接收数据用 readDatagram 。

6、多线程使用使用方法

方法一：①创建一个类从QThread类派生②在子线程类中重写 run 函数, 将处理操作写入该函数中 ③在主线程中创建子线程对象, 启动子线程, 调用start()函数

方法二：①将业务处理抽象成一个业务类, 在该类中创建一个业务处理函数②在主线程中创建一QThread类对象 ③在主线程中创建一个业务类对象 ④将业务类对象移动到子线程中 ⑤在主线程中启动子线程 ⑥通过信号槽的方式, 执行业务类中的业务处理函数

多线程使用注意事项:

* 1. 业务对象, 构造的时候不能指定父对象

* 2. 子线程中不能处理ui窗口(ui相关的类)

* 3. 子线程中只能处理一些数据相关的操作, 不能涉及窗口

7、多线程下，信号槽分别在什么线程中执行，如何控制

可以通过connect的第五个参数进行控制信号槽执行时所在的线程

connect有几种连接方式，直接连接和队列连接、自动连接

直接连接（Qt::DirectConnection）：信号槽在信号发出者所在的线程中执行

队列连接 (Qt::QueuedConnection)：信号在信号发出者所在的线程中执行，槽函数在信号接收者所在的线程中执行

自动连接 (Qt::AutoConnection)：多线程时为队列连接函数，单线程时为直接连接函数。

8、自定义控件流程

继承需要自定义的控件类，如QPushButton；从外观设计上：QSS、继承绘制函数重绘、继承QStyle相关类重绘、组合拼装等等；从功能行为上：重写事件函数、添加或者修改信号和槽等等。

9、对QObject的理解

QObject 类是Qt 所有类的基类。

QObject是Qt对象模型的核心。这个模型的中心要素就是一种强大的叫做信号与槽无缝对象沟通机制。你可以用 connect() 函数来把一个信号连接到槽，也可以用disconnect() 函数来破坏这个连接。为了避免永无止境的通知循环，你可以用blockSignal() 函数来暂时阻塞信号。保护函数 connectNotify() 和 disconnectNotify() 可以用来跟踪连接。

对象树都是通过QObject 组织起来的，当以一个对象作为父类创建一个新的对象时，这个新对象会被自动加入到父类的 children() 队列中。这个父类有子类的所有权。能够在父类的析构函数中自动删除子类。可以通过findChild()和findChildren() 函数来寻找子类。

每个对象都一个对象名称objectName() ，而且它的类名也可以通过metaObject()函数。你可以通过 inherits() 函数来决定一个类是否继承其他的类。当一个对象被删除时，它会发射destory() 信号，你可以抓住这个信号避免某些事情。

对象可以通过event() 函数来接收事情以及过滤来自其他对象的事件。就好比installEventFiter() 函数和 eventFilter() 函数。childEvent() 函数能够重载实现子对象的事件。

QObject还提供了基本的时间支持，QTimer类 提高了更高层次的时间支持。

任何对象要实现信号与槽机制，Q_OBJECT 宏都是强制的。你也需要在源原件上运行元对象编译器。不管是否真正用到信号与槽机制，最好在所有QObject子类使用Q_OBJECT宏，以避免出现一些不必要的错误。

所有的Qt widgets 都是基础QObject。如果一个对象是widget,那么isWidgetType()函数就能判断出。

10、Qt自定义一个信号槽，触发这个信号，Qt多个信号如何关联一并处理；

第一种方法：

在发送信号时，也发送一个int类型数字，或者说标志，这样在槽函数触发是可以知道是哪个信号发出的；

第二种方法：

在槽函数内有获取发送信号的函数，通过sender（）函数获取发送信号；

11、Qt如果一个信号的处理方法一直未被执行有哪些可能性；

断开了，连接的时候失败了，多线程的时候在排队或者启动锁死了。

12、在Qt5的信号处理中如何使用lambda机制（可以代码示例）；

信号定义了，但是不写对应槽函数，直接将函数写到槽的位置。

connect(musicPlayer,SIGNAL(positionChanged(qint64)),this,SLOT(slotReflushStartTime(qint64)));

connect(musicPlayer,SIGNAL(positionChanged(qint64)),slotReflushStartTime(qint64));

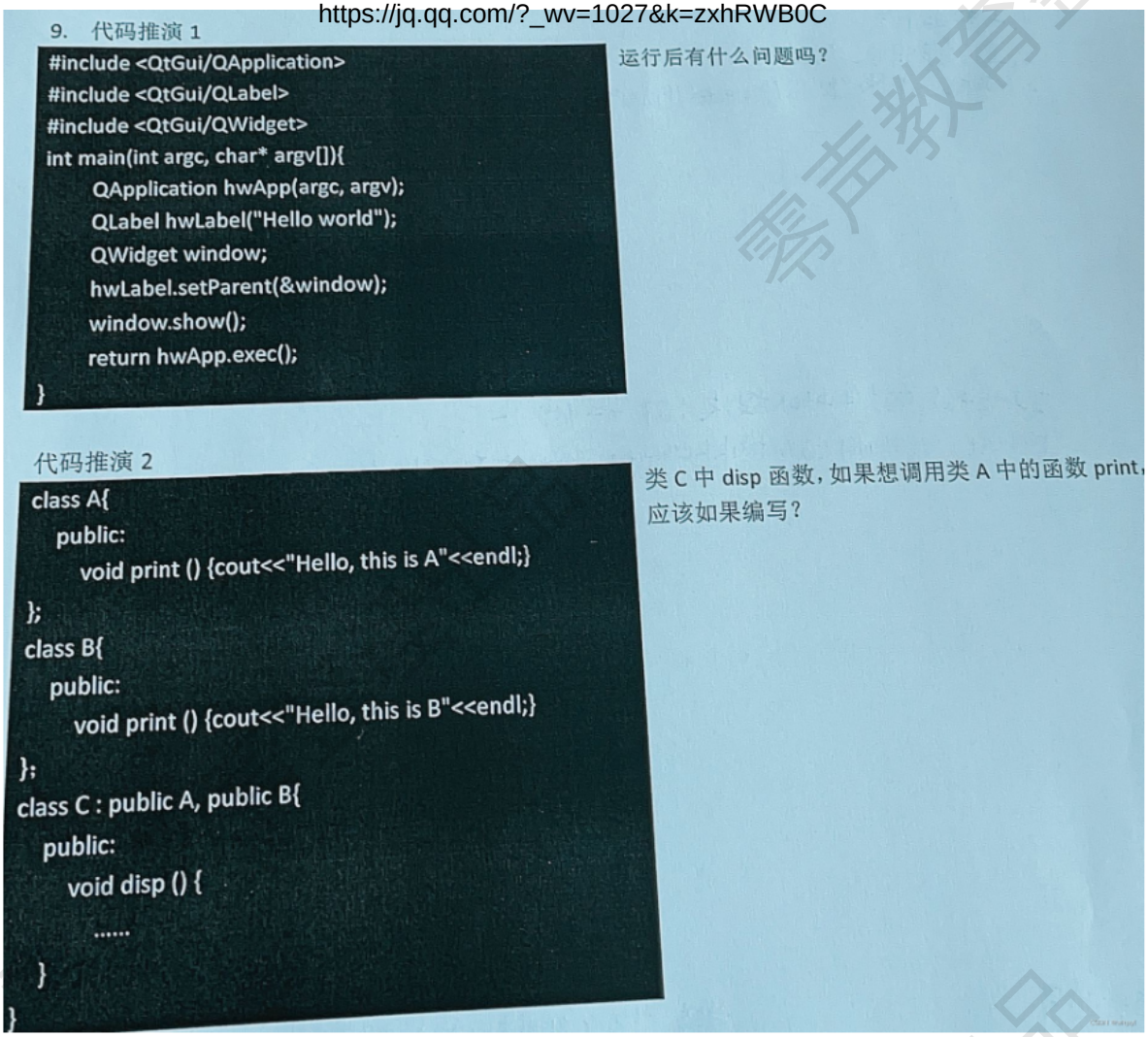
直接就是将对象都不写了，直接写个函数。

13、段错误一般是什么原因造成的，如何快速排查；

一般是指针的问题，出现野指针空指针；点灯或者用Debug去排查问题。

14：如图

题目图：



第一个：在关闭界面时会异常退出，由于是在栈区创建之后会自己释放，所以程序结束时后创建先释放，所以这里是widget后创建先释放，导致label还没释放但是承载他的widget已经连带着自己和label释放了，但是label还做了内存管理，内存管理不会去判断是否已经释放了，就会出现自己再释放一次的问题；所以解决方法就是将widget先创建。

第二个：A：：printf（）；

15、Qt 三大核心机制

信号槽

信号槽的五种连接方式（图略）

connect(信号发出者，信号，信号接收者，槽，连接方式(隐藏默认自动连接))//五个参数

元对象系统

元对象系统分为三大类:QObject类、Q_OBJECT宏和元对象编译器moc

Qt的类包含Q_OBJECT宏 moc编译器会对该类编译成标准的C++代码

事件模型

事件的创建

鼠标事件，键盘事件，窗口调整事件，模拟事件

事件的交付

Qt通过调用虚函数QObject::event()来交付事件。

事件循环模型

主事件循环通过调用QCoreApplication::exec()启动，

随着QCoreApplication::exit()结束，

本地的事件循环可用利用QEventLoop构建。

一般来说，事件是由触发当前的窗口系统产生的，但也可以通过使用 QCoreApplication::sendEvent()和

QCoreApplication::postEvent()来手工产生事件。需要说明的是QCoreApplication::sendEvent()会立即发送事件， QCoreApplication::postEvent()则会将事件放在事件队列中分发。

自定义事件

16、Qt对象树

QT提供了对象树机制，能够自动、有效的组织和管理继承自QObject的对象。

每个继承自QObject类的对象通过它的对象链表(QObjectList)来管理子类对象，当用户创建一个子对象时，其对象链表相应更新子类对象的信息，对象链表可通过children()获取。

当父类对象析构的时候，其对象链表中的所有(子类)对象也会被析构，父对象会自动，将其从父对象列表中删除，QT保证没有对象会被delete两次。开发中手动回收资源时建议使用deleteLater代替delete,因为deleteLater多次是安全的。

17、描述QTextStream(文件流)和QDataStram(数据流)的区别

文本流用来操作轻量级的数据，比如内置的int、QString等，写入文件后以文本的方式呈现

数据流，可以操作各种类型数据

总之，两者都可以进行操作磁盘文件以及内存数据。

18、信号槽的四种写法和五种连接方式？

connect(信号发出者，信号，信号接收者，槽，连接方式(隐藏默认自动连接))//五个参数
四种写法：
1.用宏：
connect(this,SIGNAL(clicked()),this,SLOT(colse())); //连接方式(隐藏默认自动连接))
2.用函数指针： connect(this,&mainwindow::my_signal,this,&mainwindow::my_slot);
3.用重载函数指针Qoverload
connect(this,Qoverload<参数>::of(&mainwindow::my_signal),this,Qoverload<参数>::of(&mainwindow::my_slot));
4.lambda表达式(匿名函数) 匿名函数代替槽
connect(this,&mainwindow::my_signal,this,[=]{qDebug()<<100;});
连接方式：自动连接(默认连接方式)
直接连接(用于单线程,自动匹配)
队列(用于多线程也可用于单线程,自动匹配)

阻塞队列(跨线程,多线程)
唯一连接(跨线程,多线程)

19、Qt模型

答：Qt中的View主要有三种QListView，QTreeView, QTabelView
而对应的Model是：QStringListModel, QAbstractItemModel , QStandardItemModel。
抽象 标准

20、Qt中的MVD了解吧？

Qt的MVD包含三个部分Model（模型），View（视图），代理（Delegate）。Model否则保存数据，View负责展示数据，Delegate负责Item样式绘制或处理输入。这三部分通过信号槽来进行通信，当Model中数据发生变化时将会发送信号到View，在View中编辑数据时，Delegate负责将编辑状态发送给Model层。基类分别为QAbstractItemModel、QAbstractItemView、QAbstractItemDelegate。Qt中提供了默认实现的MVD类，如QTableWidget、QListWidget、QTreeWidget等。

21、Qt如果要进行网络编程首先需要在.pro中添加如下代码 QT network

在头文件中包含相关头文件
include QHostInfo
include QNetworkInterface
因无法显示 略去#与)
2、QT的UdpSocket接收消息使用原则
第一步 new一个UdpSocket
第二步 调用UdpSocket的bind方法 同时指定端口号
第三步 使用connect将接收消息函数和UdpSocket对象做关联
第四步 在接受消息槽函数当中调用readDatagram接收消息

22、static和const的使用

static:静态变量声明，分为局部静态变量，全局静态变量，类静态成员变量。也可修饰类成员函数。有以下几类：

局部静态变量：存储在静态存储区，程序运行期间只被初始化一次，作用域仍然为局部作用域，在变量定义的函数或语句块中有效，程序结束时由操作系统回收资源。

全局静态变量：存储在静态存储区，静态存储区中的资源在程序运行期间会一直存在，直到程序结束由系统回收。未初始化的变量会默认为0，作用域在声明他的文件中有效。

类静态成员变量：被类的所有对象共享，包括子对象。必须在类外初始化，不可以在构造函数内进行初始化。

类静态成员函数：所有对象共享该函数，不含this指针，不可使用类中非静态成员。

const：常量声明，类常成员函数声明。 const和static不可同时修饰类成员函数，const修饰成员函数表示不能修改对象的状态，static修饰成员函数表示该函数属于类，不属于对象，二者相互矛盾。const修饰变量时表示变量不可修改，修饰成员函数表示不可修改任意成员变量。

23、指针和引用的异同

指针：是一个变量，但是这个变量存储的是另一个变量的地址，我们可以通过访问这个地址来修改变量。

引用：是一个别名，还是变量本身。对引用进行的任何操作就是对变量本身进行的操作。

相同点：二者都可以对变量进行修改。

不同点：指针可以不必须初始化，引用必须初始化。指针可以有多级，但是引用只有一级（int&& a不合法， int** p合法）。指针在初始化后可以改变，引用不能进行改变，即无法再对另一个同类型对象进行引用。sizeof指针可以得到指针本身大小，sizeof引用得到的是变量本身大小。指针传参还是值传递，引用传参传的是变量本身。

24、常用数据结构

vector：向量，连续存储，可随机访问。

deque：双向队列，连续存储，随机访问。

list：链表，内存不连续，不支持随机访问。

stack：栈，不可随机访问，只允许再开头增加/删除元素。

queue：单向队列，尾部增加，开头删除。

set：集合，采用红黑树实现，可随机访问。查找、插入、删除时间复杂度为O(logn)。

map：图，采用红黑树实现，可随机访问。查找、插入、删除时间复杂度为O(logn)。

hash_set：哈希表，随机访问。查找、插入、删除时间复杂读为O(1)。

25、谈一谈你对面向对象的理解

C++面向对象编程就是把一切事物都变成一个个对象，用属性和方法来描述对象的信息，比如定义一个猫对象，猫的眼睛、毛发、嘴巴就可以定义为猫对象的属性，猫的叫声和走路就可以定义为猫对象的方法。

用对象的方式编程，不仅方便了程序员，也使得代码的可复用性、可维护性变好。

C++面向对象的三大特性是封装、继承、多态。

26、什么场景下使用继承方式，什么场景下使用组合？

继承：通过扩展已有的类来获得新功能的代码重用方法

组合：新类由现有类的对象合并而成的类的构造方式

使用场景：

1. 如果二者存在一个“是”的关系，并且一个类要对另外一个类公开所有接口，那么继承是更好的选择
2. 如果二者间存在一个“有”的关系，那么首选组合
3. 如果没有找到及其强烈无法辩驳的使用继承的的理由的时候，一律使用组合。组合体现为现实层面，继承主要体现在扩展方面。如果并不需要一个类的所有东西（包括接口和熟悉），那么就不需要使用继承，使用组合更好。如果使用继承，那么必须所有的都继承，如果有的东西你不需要继承但是你继承了，那么这就是滥用继承。

27、如何理解多态

定义：同一操作作用于不同的对象，产生不同的执行结果。C++多态意味着当调用虚成员函数时，会根据调用类型对象的实际类型执行不同的操作。

实现：通过虚函数实现，用virtual声明的成员函数就是虚函数，允许子类重写。声明基类的指针或者引用指向不同的子类对象，调用相应的虚函数，可以根据指针或引用指向的子类的不同从而执行不同的操作。

Overload（重载）：函数名相同，参数类型或顺序不同的函数构成重载。

Override（重写）：派生类覆盖基类用virtual声明的成员函数。

Overwrite（隐藏）：派生类的函数屏蔽了与其同名的基类函数。派生类的函数与基类函数同名，但是参数不同，隐藏基类函数。如果参数相同，但是基类没有virtual关键字，基类函数将被隐藏。

28、虚函数表

多态是由虚函数实现的，而虚函数主要是通过虚函数表实现的。如果一个类中包含虚函数，那么这个类就会包含一张虚函数表，虚函数表存储的每一项是一个虚函数的地址。该类的每个对象都会包含一个虚指针（虚指针存在于对象实例地址的最前面，保证虚函数表有最高的性能），需指针指向虚函数表。注意：对象不包含虚函数表，只有需指针，类才包含虚函数表，派生类会生成一个兼容基类的虚函数表。

C++设计模式面试题**

29、分别写出饿汉和懒汉线程安全的单例模式

单例模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。根据单例对象创建时间，可分为两种模式：懒汉模式和饿汉模式。

懒汉模式：延迟加载，不到万不得已不会去实例化类，也就是说第一次用到类实例的时候才会实例化。

```
#include <iostream>
#include <mutex>

using namespace::std;

// 懒汉模式一：多线程不安全
```

```
template <typename T>
class Singleton
{
public:
    static T* getInstance()
    {
        if (instance_ == nullptr)
        {
            instance_ = new T();
        }

        return instance_;
    }

private:
    Singleton() = delete;
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;

private:
    static T* instance_;
};

template <typename T>
T* Singleton<T>::instance_ = nullptr;

// 懒汉模式二：多线程安全
template <typename T>
class Singleton2
{
public:
    static T* getInstance()
    {
        if (instance_ == nullptr)
        {
            mutex_.lock();

            if (instance_ == nullptr)
            {
               instance_ = new T();
            }

            mutex_.unlock();
        }

        return instance_;
    }

private:
    Singleton2() = delete;
    Singleton2(const Singleton2&) = delete;
    Singleton2& operator=(const Singleton2&) = delete;

private:
    static T* instance_;
```

```
    static mutex mutex_;
};

template <typename T>
T* Singleton2<T>::instance_ = nullptr;

template <typename T>
mutex Singleton2<T>::mutex_;


class Printer
{
    friend class Singleton<Printer>;
    friend class Singleton2<Printer>;

private:
    Printer() = default;
    Printer(const Printer&) = delete;
    Printer& operator=(const Printer&) = delete;

public:
    void print() { cout << "Printer" << endl; }
};


int main(int argc, char* argv[])
{
    Singleton<Printer>::getInstance()->print();
    Singleton2<Printer>::getInstance()->print();
}
```

饿汉模式：在单例类定义的时候（即在main函数之前）就进行实例化。因为main函数执行之前，全局作用域的类成员变量instance_已经初始化，故没有多线程的问题。

```
#include <iostream>
#include <mutex>

using namespace::std;

// 饿汉模式
template <typename T>
class Singleton
{
private:
    Singleton() = delete;
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;

public:
    static T* getInstance()
    {
        return instance_;
    }
```

```
private:
    static T* instance_;
};

template <typename T>
T* Singleton<T>::instance_ = new T();

class Printer
{
    friend class Singleton<Printer>;

private:
    Printer() = default;
    Printer(const Printer&) = delete;
    Printer& operator=(const Printer&) = delete;

public:
    void print() { cout << "Printer" << endl; }
};


int main(int argc, char* argv[])
{
    Singleton<Printer>::getInstance()->print();
}
```

30、说出观察者模式类关系和优点

定义了对象之间一对多的依赖，令多个观察者对象同时监听某一个主题对象，当主题对象发生改变时，所有的观察者都会收到通知并更新。

优点：

抽象耦合：在观察者和被观察者之间，建立了一个抽象的耦合，由于耦合是抽象的，可以很容易扩展观察者和被观察者。
广播通信：观察者模式支持广播通信，类似于消息传播，如果需要接收消息，只需要注册一下即可。

缺点：

依赖过多：观察者之间细节依赖过多，会增加时间消耗和程序的复杂程度。这里的细节依赖指的是触发机制，触发链条，如果观察者设置过多，每次触发都要花很长时间去处理通知。
循环调用：避免循环调用，观察者和被观察者之间绝对不允许循环依赖，否则会触发二者之间的循环调用，导致系统崩溃。

31、说出代理模式类关系和优点

优点：

代理模式能将代理对象与真实被调用的目标对象隔离。
一定程度上降低了系统的耦合度，扩展性好。
可以起到保护目标对象的作用。
可以对目标对象的功能增强。

缺点：

代理模式会造成系统设计中类的数量增加。
在客户端和目标对象增加一个代理对象，会造成请求处理速度变慢。

32、说出工厂模式概念和优点

定义一个创建产品对象的工厂接口，将产品对象的实际创建工作推迟到具体子工厂类当中。这满足创建型模式中所要求的“创建与使用相分离”的特点。简单工厂模式可以决定在什么时候创建哪一个产品类的实例。工厂方法模式有非常良好的扩展性。抽象工厂模式降低了模块间的耦合性，提高了团队开发效率。

33、说出构造者模式概念

构造者模式是较为复杂的创建型模式，它将客户端与包含多个组成部分的复杂对象的创建过程分离。客户端无需知道具体的构造过程，只需要与构造器打交道即可，构建与表示分离。

34、说出适配器模式概念

将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能在一起工作的那些类一起工作。

35、进程和线程的区别？

进程的定义：一个具有一定独立功能的程序在一个数据集合上依次动态执行的过程。进程是一个正在执行程序的实例，包括程序计数器、寄存器和程序变量的当前值。简单来说，进程就是一个程序的执行流程，内部保存程序运行所需的资源。在操作系统中可以有多个进程在运行，可对于CPU来说，同一时刻，一个CPU只能运行一个进程，但在某一时间段内，CPU将这一时间段拆分成更短的时间片，CPU不停的在各个进程间游走，这就给人一种并行的错觉，像CPU可以同时运行多个进程一样，这就是伪并行。

线程的定义：线程是进程当中的一条执行流程，这几乎就是进程的定义，一个进程内可以有多个子执行流程，即线程。从资源组合的角度看，进程把一组相关的资源组合起来，构成一个资源平台环境，包括地址空间（代码段，数据段），打开的文件等各种资源。从运行的角度看：进程是代码在这个资源平台上的执行流程，然而线程貌似也是这样，但是进程比线程多了资源内容列表样式：进程 = 线程 + 共享资源。

进程是操作系统分配资源的单位，线程是调度的基本单位，线程之间共享进程资源。

36、进程之间的通信方式有哪些？

1. 管道
2. 消息队列
3. 共享内存
4. 信号量
5. 套接字
6. 文件

37、信号和信号量的区别是什么？

信号：一种处理异步事件的方式。信号是比较复杂的通信方式，用于通知接收进程有某种事件发生，除了用于进程外，还可以发送信号给进程本身。

信号量：进程间通信处理同步互斥的机制。是在多线程环境下使用的一种设施，它负责协调各个线程，以保证它们能够正确，合理的使用公共资源。

38、你觉得自定义控件的方法主要是哪些？

从外观设计上：QSS、继承绘制函数重绘、继承QStyle相关类重绘、组合拼装等等。

从功能行为上：重写事件函数、添加或者修改信号和槽等等

39、QSS平时使用的多吗？能举几个例子吗？

1.将QSS统一写在一个文件中，通过程序给主窗口加载；

2.写成一个字符串中，通过程序给主窗口加载；

3.需要使用的地方，写一个字符串，加载给对象；

4.QT Designer中填写；

40、Qt程序是事件驱动的，事件到处都可以遇到。能说说平时经常使用到哪些事件吗？

常见的QT事件类型如下:

键盘事件: 按键按下和松开　鼠标事件: 鼠标移动,鼠标按键的按下和松开

拖放事件: 用鼠标进行拖放　滚轮事件: 鼠标滚轮滚动

绘屏事件: 重绘屏幕的某些部分　定时事件: 定时器到时

焦点事件: 键盘焦点移动　进入和离开事件: 鼠标移入widget之内,或是移出

移动事件: widget的位置改变　大小改变事件: widget的大小改变

显示和隐藏事件: widget显示和隐藏　窗口事件: 窗口是否为当前窗口

41、多线程情况下, Qt中的信号槽分别在什么线程中执行, 如何控制?

通过connect函数的第五个参数connectType来控制。

connect用于连接qt的信号和槽，在qt编程过程中不可或缺。它其实有第五个参数，只是一般使用默认值，在满足某些特殊需求的时候可能需要手动设置。

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。

Qt::DirectConnection：**槽函数会在信号发送的时候直接被调用，槽函数运行于信号发送者所在线程。效果看上去就像是直接在信号发送位置调用了槽函数。这个在多线程环境下比较危险，可能会造成奔溃。

Qt::QueuedConnection：**槽函数在控制回到接收者所在线程的事件循环时被调用，槽函数运行于信号接收者所在线程。发送信号之后，槽函数不会立刻被调用，等到接收者的当前函数执行完，进入事件循环之后，槽函数才会被调用。多线程环境下一般用这个。

Qt::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。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。

Qt::UniqueConnection：**这个flag可以通过按位或（|）与以上四个结合在一起使用。当这个flag设置时，当某个信号和槽已经连接时，再进行重复的连接就会失败。也就是避免了重复连接。

42、继承与派生的区别？

1、角度不同

继承是从子类的角度讲的，派生是从基类的角度讲的。

2、定义不同

派生指江河的源头产生出支流。引申为从一个主要事物的发展中分化出来。继承 是面向对象软件技术当中的一个概念，与多态、抽象共为面向对象的三个基本特征。 继承可以使得子类具有父类的属性和方法或者重新定义、追加属性和方法等。

43、单继承和多继承

单继承（派生类只从一个直接基类继承）时派生类的定义：

class 派生类名：继承方式 基类名

{

新增成员声明；

}

多继承时派生类的定义：

class 派生类名：继承方式1 基类名1，继承方式2 基类名2，...

{

成员声明；

}

注意：每一个"继承方式"，只用于限制对紧随其后之基类的继承。

44、知道QT事件机制有几种级别的事件过滤吗？能大致描述下吗？

根据对Qt事件机制的分析, 我们可以得到5种级别的事件过滤,处理办法. 以功能从弱到强, 排列如下:

1）重载特定事件处理函数.

最常见的事件处理办法就是重载象mousePressEvent(), keyPressEvent(), paintEvent() 这样的特定事件处理函数.

2）重载event()函数.

通过重载event()函数,我们可以在事件被特定的事件处理函数处理之前(象keyPressEvent())处理它. 比如, 当我们想改变tab键的默认动作时,一般要重载这个函数. 在处理一些不常见的事件(比如:LayoutDirectionChange)时,evnet()也很有用,因为这些函数没有相应的特定事件处理函数. 当我们重载event()函数时, 需要调用父类的event()函数来处理我们不需要处理或是不清楚如何处理的事件.

3）在Qt对象上安装事件过滤器.

安装事件过滤器有两个步骤: (假设要用A来监视过滤B的事件)

首先调用B的installEventFilter(const QOject *obj), 以A的指针作为参数. 这样所有发往B的事件都将先由A的eventFilter()处理.

然后, A要重载QObject::eventFilter()函数, 在eventFilter() 中书写对事件进行处理的代码.

4）给QAppliction对象安装事件过滤器.

一旦我们给qApp(每个程序中唯一的QApplication对象)装上过滤器,那么所有的事件在发往任何其他的过滤器时,都要先经过当前这个 eventFilter(). 在debug的时候,这个办法就非常有用, 也常常被用来处理失效了的widget的鼠标事件,通常这些事件会被QApplication::notify()丢掉. (在QApplication::notify() 中, 是先调用qApp的过滤器, 再对事件进行分析, 以决定是否合并或丢弃)

5）继承QApplication类,并重载notify()函数.

Qt 是用QApplication::notify()函数来分发事件的.想要在任何事件过滤器查看任何事件之前先得到这些事件,重载这个函数是唯一的办法. 通常来说事件过滤器更好用一些, 因为不需要去继承QApplication类. 而且可以给QApplication对象安装任意个数的事件。

45、有没有使用过Qt4？Qt5的信号槽与Qt4相比有什么改进？

*编译期：检查信号与槽是否存在，参数类型检查，Q_OBJECT是否存在

*信号可以和普通的函数、类的普通成员函数、lambda函数连接（而不再局限于信号函数和槽函数）

*参数可以是 typedef 的或使用不同的namespace specifier

*可以允许一些自动的类型转换（即信号和槽参数类型不必完全匹配）

46、信号槽是同步的还是异步的？分别如何实现？

通常使用的connect，实际上最后一个参数使用的是Qt::AutoConnection类型：Qt支持6种连接方式，其中3中最主要:

1.Qt::DirectConnection（直连方式）（信号与槽函数关系类似于函数调用，同步执行）

当信号发出后，相应的槽函数将立即被调用。emit语句后的代码将在所有槽函数执行完毕后被执行。

2.Qt::QueuedConnection（排队方式）（此时信号被塞到信号队列里了，信号与槽函数关系类似于消息通信，异步执行）

当信号发出后，排队到信号队列中，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，调用相应的槽函数。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。

3.Qt::AutoConnection（自动方式）

Qt的默认连接方式，如果信号的发出和接收这个信号的对象同属一个线程，那个工作方式与直连方式相同；否则工作方式与排队方式相同。

4.Qt::BlockingQueuedConnection(信号和槽必须在不同的线程中，否则就产生死锁)

这个是完全同步队列只有槽线程执行完成才会返回，否则发送线程也会一直等待，相当于是不同的线程可以同步起来执行。

5.Qt::UniqueConnection

与默认工作方式相同，只是不能重复连接相同的信号和槽，因为如果重复连接就会导致一个信号发出，对应槽函数就会执行多次。

6.Qt::AutoCompatConnection

工作方式与Qt::AutoConnection一样。

47、知道死锁吗？死锁是如何产生的？

死锁的产生有如下四个必要条件

1. 资源是互斥的,同一时刻只能有一个进程占有该资源
2. 资源的释放只能有该进程自己完成
3. 线程在获取到需要资源之前,不会释放已有资源

4. 存在这么一条循环等待的队列,线程T1,T2,T3..., Tn

T1持有自己的资源请求T2的资源,....Tn持有自己的资源请求T1的资源

48、Qt线程同步的方法有哪些？

1.互斥量（QMutex）

```
QMutex m_Mutex;        m_Mutex.lock();        m_Mutex.unlock();
```

2.互斥锁（QMutexLocker）

```
QMutexLocker mutexLocker(&m_Mutex);

从声明处开始（在构造函数中加锁），出了作用域自动解锁（在析构函数中解锁）。
```

3.等待条件（QWaitCondition）

QWaitCondtion m_WaitCondition; m_WaitConditon.wait(&m_muxtex, time);

m_WaitCondition.wakeAll();

4. QReadWriteLock类

》一个线程试图对一个加了读锁的互斥量进行上读锁，允许；

》一个线程试图对一个加了读锁的互斥量进行上写锁，阻塞；

》一个线程试图对一个加了写锁的互斥量进行上读锁，阻塞；、

》一个线程试图对一个加了写锁的互斥量进行上写锁，阻塞。

读写锁比较适用的情况是：需要多次对共享的数据进行读操作的阅读线程。

QReadWriterLock 与QMutex相似，除了它对 "read","write"访问进行区别对待。它使得多个读者可以共时访问数据。使用QReadWriteLock而不是QMutex，可以使得多线程程序更具有并发性。

5. 信号量QSemaphore

但是还有些互斥量（资源）的数量并不止一个，比如一个电脑安装了2个打印机，我已经申请了一个，但是我不能霸占这两个，你来访问的时候如果发现还有空闲的仍然可以申请到的。于是这个互斥量可以分为两部分，已使用和未使用。

6.QReadLocker便利类和QWriteLocker便利类对QReadWriteLock进行加解锁

49、工作中有没有使用过动态库和静态库？能不能简单说下两者的区别？

静态库：在链接阶段将汇编生成的目标文件.o与引用库一起链接打包到可执行文件中，可简单看成（.o或者.obj文件的集合）。（1）对函数库的链接是放在编译时期完成的（2）程序在运行时与函数库没有瓜葛，移植方便（3）浪费空间和资源

动态库：（1）将库函数的链接载入推迟到程序运行时期（2）可以实现进程间的资源共享（因此也称为共享库）（3）将一些程序升级变得简单（4）可以真正的做到链接载入完全由程序员在程序代码中控制（显示调用）

动态库一般也会有个lib文件，那么和静态库lib文件有什么区别？

动态库中的.lib文件叫做导入库，对于导入库而言，其实际的执行代码位于动态库中，导入库只包含了地址符号表等，确保程序找到对应函数的一些基本地址信息。

静态库中的.lib叫做静态库，本身就包含了实际执行代码、符号表等等。

50、设计模式平时有使用到吗？能不能说下常见的设计模式有哪些？能不能说说大致的概念？能不能具体说下工作中如何使用的？

总体来说设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

51、HTTP协议有使用过吗？Qt5中使用的相关联的主要的几个类？

QNetworkAccessManager/QNetworkRequest/QNetworkReply。

52、平时使用算法比较多吗？能简单说下快排的思想吗？时间复杂度是多少？

基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

时间复杂度：平均为O(nlogn)，最好为O(nlogn)，最差为O(logn2)

53、如果软件除了问题（Bug），如何快速定位？主要方法有哪些？

打印输出/代码调试/日志记录/分析工具/找同事讨论。

1、二分法定位技巧

无论是有多复杂的代码，利用二分法定位技巧一般都是可以定位到问题所在。

从二分法定位技巧可以延伸出一些具体的处理bug的方法，比如：对输入数据二分、对代码版本二分、注释掉部分代码、在不同位置插入试探性代码、对运行环境二分。

2、IDE调试

IDE的VS debug的功能简直就是立竿见影。它可以加断点，单步调试。

单步调试可以让我们对代码逻辑，执行顺序，以及各种中间结果更加清晰。

至于本身容易出错的BUG，用IDE调试简直是再合适不过了。

3、重新读一遍程序

相对新手程序员来说，如果代码出现bug，可以重新读一遍程序。这种方法是最有效、最快速的 Debug 方式。

4、必杀，重写一遍

如果你发现无论如何也找不到BUG，而且代码只是复杂，本身不是很长，直接重写代码吧!

5、小黄鸭调试法

小黄鸭调试法是程序员们经常使用的调试代码方法之一。

小黄鸭不懂程序，所以我们可以向他解释每一行程序的作用，以此来激发灵感。

54、引用和指针有何区别

1.指针是一个对象，而引用仅是一个对象的别名

2.引用使用时无需解引用，指针需要

3.引用只能在定义时初始化一次，而指针可变

4.引用不能为空，指针可以为空

5.有多级指针没有多级引用

6.不会为引用变量开辟内存空间，它和它引用的变量共用一块内存空间，指针会开辟内存空间

7.引用自加改变引用值的内容，指针自加改变指针的指向

8.sizeof含义不同，引用结果为引用类型的大小，指针始终是地址空间所占大小

9.引用比指针使用起来相对更安全

55、什么情况下使用虚函数？和纯虚函数有什么区别？虚析构函数的作用是什么？（虚函数表）

虚函数的主要作用是“运行时多态”。虚析构函数的作用在于使用delete删除一个对象时，能确保析构函数被正确的执行。

区别：

1. 虚函数和纯虚函数可以定义在同一个类(class)中，含有纯虚函数的类被称为抽象类(abstract class)，而只含有虚函数的类(class)不能被称为抽象类(abstract class)。
2. 虚函数可以被直接使用，也可以被子类(sub class)重载以后以多态的形式调用，而纯虚函数必须在子类(sub class)中实现该函数才可以使用，因为纯虚函数在基类(base class)只有声明而没有定义。
3. 虚函数和纯虚函数都可以在子类(sub class)中被重载，以多态的形式被调用。
4. 虚函数和纯虚函数通常存在于抽象基类(abstract base class -ABC)之中，被继承的子类重载，目的是提供一个统一的接口。
5. 虚函数的定义形式：virtual {method body}

纯虚函数的定义形式：virtual { } = 0;

在虚函数和纯虚函数的定义中不能有static标识符，原因很简单，被static修饰的函数在编译时候要求前期bind,然而虚函数却是动态绑定(run-time bind)，而且被两者修饰的函数生命周期(life recycle)也不一样。

6. 虚函数必须实现，如果不实现，编译器将报错，错误提示为：

   error LNK: unresolved external symbol "public: virtual void __thiscall

   ClassName::virtualFunctionName(void)"

7. 对于虚函数来说，父类和子类都有各自的版本。由多态方式调用的时候动态绑定。
8. 实现了纯虚函数的子类，该纯虚函数在子类中就编程了虚函数，子类的子类即孙子类可以覆盖该虚函数，由多态方式调用的时候动态绑定。
9. 虚函数是C++中用于实现多态(polymorphism)的机制。核心理念就是通过基类访问派生类定义的函数。
10. 多态性指相同对象收到不同消息或不同对象收到相同消息时产生不同的实现动作。C++支持两种多态性：编译时多态性，运行时多态性。

a.编译时多态性：通过重载函数实现

b 运行时多态性：通过虚函数实现。

11. 如果一个类中含有纯虚函数，那么任何试图对该类进行实例化的语句都将导致错误的产生，因为抽象基类(ABC)是不能被直接调用的。必须被子类继承重载以后，根据要求调用其子类的方法。

56、对Qt元对象系统了解吗？

Qt对标准的C++进行了扩展，如信号槽、对象属性等。Qt的元对象编译系统MOC是一个预处理器，当Qt读取源文件时检测到类中包含有Q_OBJECT宏时，则会创建一个新的文件（生成路径下的moc开头的文件），将源码转换为C++编译器可以识别的代码写入moc开头的文件，然后C++编译器对其进行编译。当你的类需要使用Qt的扩展功能时，如信号槽、对象属性等时，则必须使用MOC，反之如果你的类不使用这些功能的时候不要无畏的使用MOC增大源码体积。使用MOC系统的方法：

1. 继承QObject。
2. 类中添加Q_OBJECT宏。

57、Qt中的MVD了解吗？

Qt的MVD包含三个部分Model（模型），View（视图），代理（Delegate）。Model否则保存数据，View负责展示数据，Delegate负责Item样式绘制或处理输入。这三部分通过信号槽来进行通信，当Model中数据发生变化时将会发送信号到View，在View中编辑数据时，Delegate负责将编辑状态发送给Model层。基类分别为QAbstractItemModel、QAbstractItemView、QAbstractItemDelegate。Qt中提供了默认实现的MVD类，如QTableWidget、QListWidget、QTreeWidget等。

58、QObject是否是线程安全的

QObject及其所有子类都不是线程安全的（但都是可重入的）。因此，你不能有两个线程同时访问一个QObject对象，除非这个对象的内部数据都已经很好地序列化（例如为每个数据访问加锁）。

59、QObject的线程依附性是否可以改变

调用QObject::moveToThread()函数。该函数会改变一个对象及其所有子对象的线程依附性。

- 由于QObject本身是线程不安全的，因此moveToThread接口的调用必须在QObject对象所在的线程内调用。

60、如何安全的在另外一个线程中调用QObject对象的接口

QObject被设计成在一个单线程中创建与使用，因此，在一个线程中创建一个对象，而在另外的线程中调用它的函数，这样的行为不能保证工作良好。

使用信号槽的队列连接或者QT的反射系统提供的QMetaObject::invokeMethed的队列连接调用。这要求接口必须是内省的，也就是说这个函数要么是一个槽函数，要么标记有Q_INVOKABLE宏。

将事件提交到接收对象所在线程的事件循环；当事件发出时，响应函数就会被调用。

61、QFrame与QWidget的区别

QFrame 和 QWidget 都是 Qt 中的 GUI 组件，但是它们有一些区别：

继承关系：QFrame 继承自 QWidget，所以 QFrame 具有 QWidget 的所有功能。

功能：QFrame 提供了一个简单的框架，可以作为其他控件的容器。它还可以用来绘制简单的图形，如线条。QWidget 没有这样的功能，但是提供了基础的 GUI 组件功能，如设置尺寸和位置等。

外观：QFrame 可以有边框和背景颜色，因此外观更加丰富。QWidget 只有背景颜色，没有边框。

通常，当需要一个简单的框架时，使用 QFrame，当需要基础的 GUI 组件功能时，使用 QWidget。

62、信号重载了，如何确定连接哪个信号？

采用函数指针确定连接哪个信号。

63、槽函数参数、信号的参数

槽函数的参数可以少于信号的参数。

- 槽函数本身参数比信号的少
- 槽函数参数带有默认参数

64、槽函数的参数是否可以比信号的参数多？

也可以。唯一的情况就是槽函数参数带有默认参数，除去默认参数外，槽函数的参数必须小于等于信号的参数。

65、指针和引用有什么区别？什么情况下用指针，什么情况下用引用？

区别：

1.指针是一个变量，只不过这个变量存储的是一个地址，指向内存的一个存储单元，即指针是一个实体；而引用跟原来的变量实质上是同一个东西，只不过是原变量的一个别名而已。

2.有const指针，但是没有const引用。

3.指针可以有多级，但是引用只能是一级（int** p；合法，而int&& a；不合法）。

4.指针的值可以为空，但是引用的值不可以，并且引用在定义的时候必须初始化。

5.指针的值在初始化后可以改变，即指向其它的存储单元，而引用在初始化后就不会再改变了，从一而终。

6.sizeof引用得到的是所指向的变量（对象）的大小，而sizeof指针得到的是指针本身的大小。

7.指针和引用的自增++运算意义不一样。

相同点：

1.都可以对变量就行修改。

2.都是地址的概念，指针指向一块内存，它的内容是所指内存的地址，引用是某块内存的别名。

何时使用：

1.当考虑到存在不指向任何对象的可能，这时候应该使用指针。

2.当需要在能够在不同的时刻指向不同的对象，这个时候使用指针。如果总是指向一个对象并且一旦指向一个对象后就不会改变指向，那么应该使用引用。

3.当重载某个操作符时，应该使用引用。

66、一般什么情况下会出现内存泄漏？怎么用C++在编码层面尽量避免内存泄漏。

内存泄漏是指程序向系统申请分配内存使用（new），用完以后却没有归还（delete）。结果申请的那块内存程序不再使用，而系统也无法再讲它分配给需要的程序。

造成内存泄漏的几种情况：

1.指针重新赋值
2.错误的内存释放
3.返回值的不正确处理
4.new和delete没有配对使用。

如何避免内存泄漏：

1.确保没有访问空指针。
2.尽量使用智能指针。
3.new和delete配对使用。

67、 对C++11 的智能指针了解多少，可以自己实现一个智能指针吗？

三种智能指针：

std::shared_ptr：使用引用计数，每一个shared_ptr的拷贝都指向相同的内存，每次拷贝都会触发引用计数+1，每次生命周期结束析构的时候引用计数-1，在最后一个shared_ptr析构的时候，内存才会释放。

std::weak_ptr：用来监视shared_ptr的生命周期，它不管理shared_ptr内部的指针，它的拷贝析构都不会影响引用计数，纯粹是作为一个旁观者监视shared_ptr中管理的资源是否存在，可以用来返回this指针和解决循环引用问题。

std::unique_ptr：独占型的智能指针，它不允许其它智能指针共享其内部指针，也不允许unique_ptr的拷贝和赋值。

68、show()和exec()的区别

show显示非模态窗口（不影响用户对其他窗口操作），exec显示模态窗口（阻塞其他窗口，必须在当前窗口操作完成后才能访问其他窗口），open半模态（阻塞其他窗口响应，但不影响后续代码执行）

69、Qt事件循环

Qt的主事件循环能够从事件队列中获取本地窗口系统事件，然后判断事件类型，并将事件分发给特定的接收对象。

主事件循环通过调用QCoreApplication::exec()启动，随着QCoreApplication::exit()结束，本地的事件循环可用利用QEventLoop构建。

70、什么叫自定义控件？

qt本身的控件不能满足需求将原控件功能提升达到要求,此时控件为自定义控件。

71、Qt的D指针（`d_ptr`）与Q指针（`q_ptr`）

题目：讲一下Qt的D指针和Q指针？

D指针

PIMPL模式，指向一个包含所有数据的私有数据结构体。

- 私有的结构体可以随意改变，而不需要重新编译整个工程项目
- 隐藏实现细节
- 头文件中没有任何实现细节，可以作为API使用
- 原本在头文件的实现部分转移到乐源文件，所以编译速度有所提高

Q指针

私有的结构体中储存一个指向公有类的Q指针。

总结

- Qt中的一个类常用一个PrivateXXX类来处理内部逻辑，使得内部逻辑与外部接口分开，这个PrivateXXX对象通过D指针来访问；在PrivateXXX中有需要引用Owner的内容，通过Q指针来访问。
- 由于D和Q指针是从基类继承下来的，子类中由于继承导致类型发生变化，需要通过`static_cast`类型转化，所以`DPTR()`与`QPTR()`宏定义实现了转换。

72、Qt信号槽的调用流程

- MOC查找头文件中的signal与slots，标记出信号槽。将信号槽信息储存到类静态变量staticMetaObject中，并按照声明的顺序进行存放，建立索引。
- connect链接，将信号槽的索引信息放到一个双向链表中，彼此配对。
- emit被调用，调用信号函数，且传递发送信号的对象指针，元对象指针，信号索引，参数列表到active函数。
- active函数在双向链表中找到所有与信号对应的槽索引，根据槽索引找到槽函数，执行槽函数。

73、Qt connect的第五个参数（信号槽链接方式）？

1. `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。
2. `Qt::DirectConnection`：槽函数会在信号发送的时候直接被调用，槽函数运行于信号发送者所在线程。效果看上去就像是直接在信号发送位置调用了槽函数。这个在多线程环境下比较危险，可能

会造成奔溃。

3. `Qt::QueuedConnection`：槽函数在控制回到接收者所在线程的事件循环时被调用，槽函数运行于信号接收者所在线程。发送信号之后，槽函数不会立刻被调用，等到接收者的当前函数执行完，进入事件循环之后，槽函数才会被调用。多线程环境下一般用这个。
4. `Qt::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。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。
5. `Qt::UniqueConnection`：这个flag可以通过按位或（|）与以上四个结合在一起使用。当这个flag设置时，当某个信号和槽已经连接时，再进行重复的连接就会失败。也就是避免了重复连接。

74、了解Qt的QPointer吗？

QPointer

QPointer只能用于指向QObject及派生类的对象。当一个QObject或派生类对象被删除后，QPointer能自动将其内部的指针设置为0，这样在使用QPointer之前就可以判断一下是否有效乐。

QPointer对象超出作用域时，并不会删除它指向的内存对象。

75、了解Qt的QSharedPointer吗？

QSharedPointer

用于实现数据的隐式共享。Qt中大量使用了隐式共享与写时拷贝技术，例如：

```
QString str1 = "abc";
QString str2 = str1;
str2[2] = "X";
```

第二行执行完后，str2和str1指向同一片内存数据。第三句执行时，Qt会为str2的内部数据重新分配内存。这样做的好处是可以有效地减少大片数据拷贝的次数，提高程序的运行效率。

Qt中隐式共享和写时拷贝就是利用QSharedDataPointer和QSharedData这两个类实现的。

76、描述Qt中的文件流(QTextStream)和数据流(QDataStream)的区别，他们都能帮助我们完成一些什么事情

- QTextStream – 文本流, 操作轻量级数据(int, double, QString), 数据写入文件中之后以文本的方式呈现。
- 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。
- QTextStream, QDataStream可以操作磁盘文件, 也可以操作内存数据, 通过流对象可以将数据打包到内存, 进行数据的传输。

77、详解Qt中的内存管理机制

所有继承自QOBJECT类的类，如果在new的时候指定了父亲，那么它的清理时在父亲被delete的时候delete的，所以如果一个程序中，所有的QOBJECT类都指定了父亲，那么他们是会一级级的在最上面的父亲清理时被清理，而不用自己清理；

程序通常最上层会有一个根的QOBJECT，就是放在setCentralWidget（ ）中的那个QOBJECT，这个QOBJECT在 new的时候不必指定它的父亲，因为这个语句将设定它的父亲为总的QAPPLICATION，当整个QAPPLICATION没有时它就自动清理，所以也无需清理。9这里QT4和QT3有不同，QT3中用的是setmainwidget函数，但是这个函数不作为里面QOBJECT的父亲，所以QT3中这个顶层的QOBJECT要自行销毁）。

这是有人可能会问那如果我自行delete掉这些QT接管负责销毁的指针了会出现什么情况呢，如果时这样的话，正常情况下QT的拥有这个对象的那个父亲会知道这件事情，它会直到它的儿子被你直接DELETE了，这样它会将这个儿子移出它的列表，并且重新构建显示内容，但是直接这样做时有风险的！也就是要说的下一条。

当一个QOBJECT正在接受事件队列时如果中途被你DELETE掉了，就是出现问题了，所以QT中建议大家不要直接DELETE掉一个 QOBJECT，如果一定要这样做，要使用QOBJECT的deleteLater()函数，它会让所有事件都发送完一切处理好后马上清除这片内存，而且就算调用多次的deletelater也不会有问题。

QT不建议在一个QOBJECT 的父亲的范围之外持有对这个QOBJECT的指针，因为如果这样外面的指针很可能不会察觉这个QOBJECT被释放，会出现错误，如果一定要这样，就要记住你在哪这样做了，然后抓住那个被你违规使用的QOBJECT的destroyed（ ）信号，当它没有时赶快置零你的外部指针。当然我认为这样做是及其麻烦也不符合高效率编程规范的，所以如果要这样在外部持有QOBJECT的指针，建议使用引用或者用智能指针，如QT就提供了智能指针针对这些情况，见**一条。**

QT中的智能指针封装为QPointer类，所有QOBJECT的子类都可以用这个智能指针来包装，很多用法与普通指针一样，可以详见QT assistant
通过调查这个QT的内存管理功能，发现了很多东西，现在觉得虽然这个QT弄的有点小复杂，但是使用起来还是很方便的，

要说的是某些内存泄露的检测工具会认为QT的程序因为这种方式存在内存泄露，发现时大可不必理会。

78、QSS平时使用的多吗?能举几个例子吗?

1.将QSS统一写在-一个文件中,通过程序给主窗口加载;
2.写成一个字符串Q中,通过程序给主窗口加载;
3.需要使用的地方, 写一个字符串,加载给对象;
4.QT Designer中填写;

79、你觉得自定义控件的方法主要是哪些?

从外观设计上: QSS、继承绘制函数重绘、继承QStyle相关类重绘 、组合拼装等等
从功能行为上:重写事件函数、添加或者修改信号和槽等等

80、知道Qt事件机制有几种级别的事件过滤吗?能大致描述下吗?

根据对Qt事件机制的分析,我们可以得到5种级别的事件过滤，处理办法.以功能从弱到强，排列如下:

1)重载特定事件处理函数.

最常见的事件处理办法就是重载象mousePressEvent), keyPressEvent(), paintEvent()这样的特定事件处理函数.

2)重载event()函数.

通过重载event()函数，我们可以在事件被特定的事件处理函数处理之前(象keyPressEvent()处理它.比如，当我们想改变tab键的默认动作
时，一般要重载这个函数.在处理一些不常 见的事件(比如:L ayoutDirectionChange)时,evnet()也很有用,因为这些函数没有相应的特定事件处
理函数.当我们重载event()函数时,需要调用父类的event()函数来处理我们不需要处理或是不清楚如何处理的事件.

3)在Qt对象上安装事件过滤器.

安装事件过滤器有两个步骤: (假设要用A来监视过滤B的事件)

首先调用B的nstalEventFilter(const QOject *obj),以A的指针作为参数.这样所有发往B的事件都将先由A的eventFilter()处理.

然后，A要重载QObject:eventFilter()函数,在eventilter()中书泻对事件进行处理的代码.

4)给QAppliction对象安装事件过滤器.

一旦我们给qApp(每 个程序中唯一的QApplic ation对象)装上过滤器，那么所有的事件在发往任何其他的过滤器时,都要先经过当前这个
eventFilter().在debug的时候这个办法就非常有用，也常常被用来处理失效了的widget的鼠标事件.通常这些事件会被QApplic aon:notif()
丢掉. (在Qplication:notify()中,是先调用qApp的过滤器，再对事件进行分析,以决定是否合并或丢弃)

5)继承QApplication类 并重载ntify()函数.

Qt是用Qpplcation:notify()函数来分发事件的想要在任何事件过滤器查看任何事件之前先得到这些事件重载这个函数是唯一的办法. 通
常来说事件过滤器更好用一些, 因为不需要去继承QApplication类.而且可以给QApplication对象安装任意个数的事件。

81、什么是Qml

QML是语言的名称（就像C++，那是一些其他的语言.....)

QML代表Qt Meta Language或Qt Modelling Language，是一种人机界面标记语言。

QtQuick是QML的一个工具包，允许用QML语言扩展图形界面（还有其他的QML工具包，有些是图形化的，如Sailfish Silica或BlackBerry Cascade，还有一些是非图形化的，如QBS，它是QMake/CMake/make的替代品...）。

QtQuick 1.X变成了基于Qt4.X，使用QPainter/QGraphicsView API来吸引场景。QtQuick 2.X与Qt5.0一起推出，主要基于Scene Graph，这是一个OpenGLES2抽象层，经过了相当的优化。

随着Qt5.1的推出，Scene Graph变得更适合应用多线程（QtQuick 2.1）和Qt5.2。

82、strcpy/sprintf/memcpy. 它们之间区别?

（1）执行对象
strcpy：字符串
memcpy：可适用于任意数据类型
sprintf：目的对象是字符串，源对象可以是字符串、也可以是任意基本类型的数据
（2）
strcpy：不需要指定长度，它遇到被复制字符的串结束符"\0"才结束
memcpy() 会完整的复制 num 个字节，不会因为遇到"\0"而结束

83、面向对象三大特性以及C++ 成员函数

面向对象的三大特性：

封装、继承、多态

类的六个默认成员函数：

构造函数
拷贝构造函数
析构函数
赋值操作符重载
取地址操作符重载
const修饰的取地址操作符重载

84、使用样式表要注意的点

父控件采用样式表设置属性后，该属性会传递到其子控件上，除非子控件使用同样的方法修改属性。
如：利用样式表设置父控件最小高度为x，则子控件的最小高度也为x，即使用setFixHeight()修改也无法消除，只能通过样式表重新设置子控件的高度才有效。因此一般只对子控件使用样式设置。

85、描述Windows下一个消息从触发到处理的整个路由过程

应用程序启动, 操作系统为程序创建一个对应的消息队列, 用户对创建进行操作, 产生一系列消息, 操作系统首先捕捉到这些消息, 将消息投递到对应的消息队列中, 在应用程序中对应一个消息循环 。

消息循环每次从消息队列中取出消息, 取出的消息如果是虚拟键消息, 会将其转换成标准消息, 将转换的消息再次投递到消息队列, 如果取出的是标准消息, 会将该消息发送给操作系统, 操作系统 会调用对应的窗口过程函数, 下窗口过程函数中对对用的消息进程处理.

86、QApplication的主要作用是什么?

QApplication对象管理QtGui应用程序的控制流程和主要的设置参数

87、请写一个调用消息对话框提示报错的程序

QMessageBox::waring(his,tr("警告"), tr("用户名或密码错误!"), QMessageBox :Yes)

88、Qt都提供哪些标准对话框以供使用，他们实现什么功能

9个QColorDialog颜色对话框，能够允许用户选择颜色、QErrorMessage显示错误信息、QFileDialog文件对话框，能够允许用户选的一个或者多个文件以及目录、QFontDialog字体对话框,允许用户选择/设置字体、QInputDialog输入对话框,允许用户进行简单的输入、QPageSetupDialog叶设置对话框，配置与页相关的打印机选项、QProgressDialog进度对话框指示一个长时间操作的工作进度,以提示用户该操作是否已经停止QPrintDialog打印对话框，配置打印机，可以允许用户选择可用的打印机、QMessageBox。

89、如何将UI界面问件转化成代码的. h文件?(假设ui文件名为gogogo. ui。)

UIC -o gogogo. h　　gogogo. ui

90、Qt5实现一个文件对话框

需要 #include

```
QString file_name=QFileDialog::getOpenFileName(this,"请选择需要打开的文
件：","","*.txt *.png");  //打开文件对话框
        //参数1 父控件
        //参数2 标题
        //参数3  默认路径
        //参数4 过滤文件格式
        //返回值  文件全路径---"D:/ss/注意事项.txt"
    qDebug()<<file_name;
```

91、QMainForm是从哪里派生的？

QMainWindow::QWidget::QObject

92、Qwidget、Qobejct实现了哪些功能

QObject

1、信号和槽的非常强大的机制,使用connect()把信号和槽连接起来并且可以用disconnect()来破坏这种连接。为了避免从不结束的通知循环，你可以调用blockSignals()临时地阻塞信号。保护函数connectNotify()和disconnectNotify()使跟踪连接成为可能。

2、QObject可以通过event()接收事件并且过滤其它对象的事件。详细情况请参考installEventFilter()和eventFilter()。一个方便的处理者，childEvent()，能够被重新实现来捕获子对象事件。

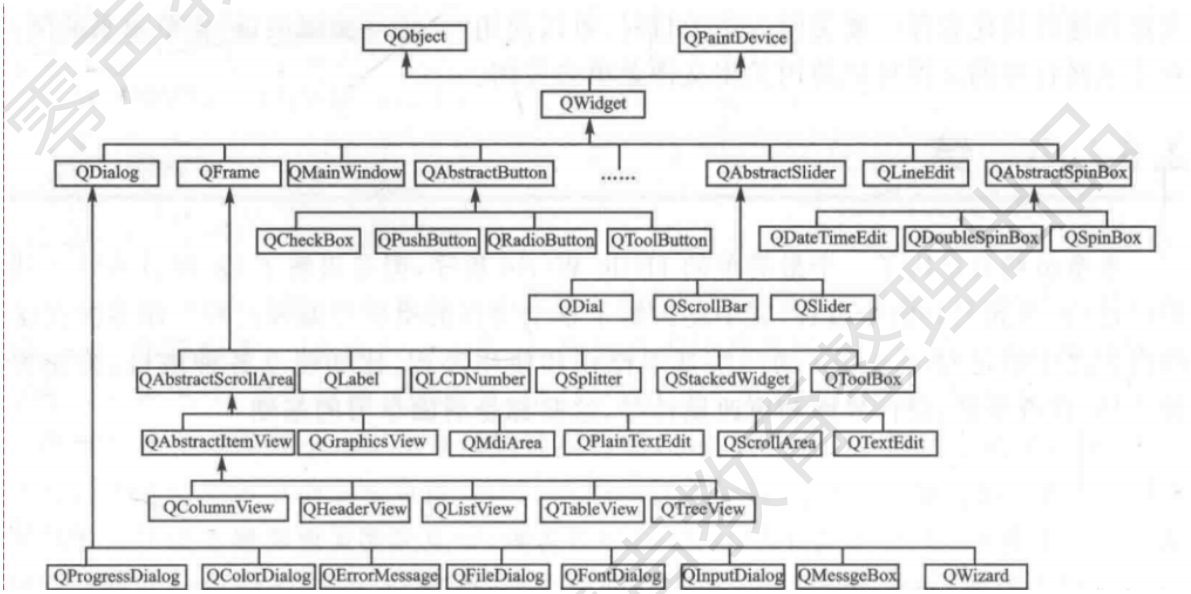
3、最后但不是最不重要的一点，QObject提供了Qt中最基本的定时器，关于定时器的高级支持请参考QTimer。

4、注意Q_OBJECT宏对于任何实现信号、槽和属性的对象都是强制的。

5、所有的Qt窗口部件继承了QObject。方便的函数isWidgetType()返回这个对象实际上是不是一个窗口部件。它比inherits("QWidget")快得多。

QWidget

1、QWidget类是所有用户界面对象的基类。
2、Widget是用户界面的基本单元：它从窗口系统接收鼠标，键盘和其他事件，并在屏幕上绘制自己。每个Widget都是矩形的，它们按照Z-order进行排序。



93、参数传值、指针、引用有什么区别，在什么场景常用哪种传递方式？

传值、传址、传引用的区别，哪个更高效？

1.传值

这种传递方式中，实参和形参是两个不同的地址空间，参数传递的实质是将原函数中变量的值，复制到被调用函数形参所在的存储空间中，这个形参的地址空间在函数执行完毕后，会被回收掉。整个被调用函数对形参的操作，只影响形参对应的地址空间，不影响原来函数中的变量的值，因为这两个不是同一个存储空间。

即使形参的值在函数中发生了变化，实参的值也完全不会受到影响，仍为调用前的值。

2.传址

这种参数传递方式中，实参是变量的地址，形参是指针类型的变量，在函数中对指针变量的操作，就是对实参（变量地址）所对应的变量的操作，函数调用结束后，原函数中的变量的值将会发生改变。

被调用函数中对形参指针所指向的地址中内容的任何改变都会影响到实参。

3.传引用

这种参数传递方式中，形参是引用类型变量，其实就是实参的一个别名，在被调用函数中，对引用变量的所有操作等价于对实参的操作，这样，整个函数执行完毕后，原先的实参的值将会发生改变。

被调函数对形参做的任何操作都影响了主调函数中的实参变量。

4.哪一种更高效？

在内置类型当中三种传递方式的效率上都差不多；

在自定义类型当中，传引用的更高效一些，因为它没有对形参进行一次拷贝

94、const与#define有什么区别

（1）const和#define都可以定义常量，但是const用途更广。

（2）const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误。

（3）有些集成化的调试工具可以对const 常量进行调试，但是不能对宏常量进行调试。

95、struct和class有什么区别？

C++中，class与struct都可以定义一个类。他们有以下两点区别：

1.默认继承权限，如果不指定，来自class的继承按照private继承处理，来自struct的继承按照public继承处理；

2.成员的默认访问权限。class的成员默认是private权限，struct默认是public权限。

以上两点也是struct和class最基本的差别，也是最本质的差别；

但是在C++中，struct进行了扩展，现在它已经不仅仅是一个包含不同数据类型的数据结构了，它包括了更多的功能。

Struct能包含成员函数、有自己的构造函数、可以有析构函数、支持继承、支持多态、支持Private、Protected、Public关键字。

如果是class的父类是struct关键字描述的，那么默认访问属性是什么？

当出现这种情况时，到底默认是public继承还是private继承，取决于子类而不是基类。

class可以继承自struct修饰的类；同时，struct也可以继承自class修饰的类，继承属性如下列描述：

class A{};

class B:A{}; // private 继承

struct B:A{}; // public 继承

最后，那么到底是使用struct，还是使用class呢？

一般来说，两个关键字都是可以的，但是由于编程规范的问题，如果要定义的是一种数据结构，那么用struct，如果是一种对象的话，那么用class。

96、C++内存分配有几种方式？

内存的三种分配方式：

1．从静态存储区分配：此时的内存在程序编译的时候已经分配好，并且在程序的整个运行期间都存在。全局变量，static变量等在此存储。

2．在栈区分配：相关代码执行时创建，执行结束时被自动释放。局部变量在此存储。栈内存分配运算内置于处理器的指令集中，效率高，但容量有限。

3．在堆区分配：动态分配内存。用new/malloc时开辟，delete/free时释放。生存期由用户指定，灵活。但有内存泄露等问题。

97、Qt设计界面有哪些方式？

（1）手工编写创建界面的代码：此方法比较复杂，不够直观；
（2）使用Qt Designer界面编辑器设计：可直接拖放控件、设置控件的属性，简单、直观、易于操作；
（3）动态加载UI文件并生成界面：此方法很灵活，当需要更改界面时只需更改.UI文件即可，无需重新编译程序。

A、手工设计界面

使用手工创建代码时，需要从Qt已有的GUI类库中选择一个类作为基类继承，并且添加必要的其它成员。通常，我们会选择从QDialog、 QWidget、QMainWindow等类中选择一个作为主窗体；然后创建其它的控件，并使用布局管理器布局这些控件；最后将该布局设置为主窗体的布 局。此步骤用图描述如下：例如，对于下图所示的FindDialog对话框，就可以通过从QDialog继承，并添加按钮、布局管理器等到派生类中完成该对话框的设计。相关的代码如下：

```
class FindDialog : public QDialog[喝小酒的网
摘]http://blog.hehehehehe.cn/a/8574.htm
{Q_OBJECTpublic:FindDialog(QWidget *parent = 0);signals:void findNext(const
QString &str, Qt::CaseSensitivity cs);void findPrevious(const QString &str,
Qt::CaseSensitivity cs);private slots:void findClicked();void
enableFindButton(const QString &text);private:
                // 窗体中的控件QLabel *label;QLineEdit *lineEdit;  QCheckBox
*caseCheckBox;QCheckBox *backwardCheckBox;QPushButton *findButton;QPushButton
*closeButton;
};FindDialog::FindDialog(QWidget *parent): QDialog(parent)
{// 下面的代码创建窗体中的控件label = new QLabel(tr("Find &what:"));lineEdit = new
QLineEdit;label->setBuddy(lineEdit);caseCheckBox = new QCheckBox(tr("Match
&case"));backwardCheckBox = new QCheckBox(tr("Search &backward"));findButton =
new QPushButton(tr("&Find"));findButton->setDefault(true);findButton-
>setEnabled(false);closeButton = new QPushButton(tr("Close"));connect(lineEdit,
SIGNAL(textChanged(const QString &)),this, SLOT(enableFindButton(const QString
&)));connect(findButton, SIGNAL(clicked()),this,
SLOT(findClicked()));connect(closeButton, SIGNAL(clicked()),this,
SLOT(close()));// 使用布局管理器布局控件QHBoxLayout *topLeftLayout = new
QHBoxLayout;topLeftLayout->addWidget(label);topLeftLayout-
>addWidget(lineEdit);QVBoxLayout *leftLayout = new QVBoxLayout;leftLayout-
>addLayout(topLeftLayout);leftLayout->addWidget(caseCheckBox);leftLayout-
>addWidget(backwardCheckBox);QVBoxLayout *rightLayout = new
QVBoxLayout;rightLayout->addWidget(findButton);rightLayout-
>addWidget(closeButton);rightLayout->addStretch();QHBoxLayout *mainLayout = new
QHBoxLayout;mainLayout->addLayout(leftLayout);mainLayout-
>addLayout(rightLayout);// 设置窗口的布局管理器setLayout(mainLayout);
setWindowTitle(tr("Find"));setFixedHeight(sizeHint().height());
}
```

B、使用Qt Designer设计界面

采用Qt Designer，使得快速创建对话框成为可能。在Qt Designer环境中，所有的操作都采用可视化的操作，可拖放控件、关联信号与槽、设置特定控件的属性。使用Qt Designer设计界面的方法如下图所示：

C、动态加载UI文件并生成界面

前面的两种方法需要事先创建好相应的文件或代码，然后连同其它文件进行编译，如果后期要修改界面则必须修改代码或UI文件并重新编译。而不需要重新编译整个程序的方法是采用动态加载UI文件的方式。基本的操作方法为先使用Qt Designer设计界面，然后按下图的流程操作：

如下图所示，创建一个mainwindow.ui的UI文件。之后就可以采用QUILoader类动态加载该文件，并生成该窗体。参考的代码如下：

```
#include <QUiLoader>
#include <QFile>int main(int argc, char *argv[])
{QApplication a(argc, argv);QUiLoader loader;QFile
file("mainwindow.ui");loader.load(&file)->show();return a.exec();
}
```

上面的代码中UiLoader::load()使用了QFile对像作为数据源，并且会生成QWidget对像，最后使用了QWidget::show()显示上图中的窗体界面。另外需要注意的是，如果要使能UiLoader动态加载特性，必须在工程文件*.pro中添加如下行：CONFIG += uitools

98、Qt Socket通信的过程

Qt Socket通信的过程主要分为以下几步：

1. 创建Socket：使用QTcpSocket类创建Socket，并初始化连接参数；
2. 连接服务器：使用connectToHost()函数连接服务器；
3. 发送数据：使用write()函数发送数据；
4. 接收数据：使用read()函数接收数据；
5. 断开连接：使用disconnectFromHost()函数断开连接。

99、QWidget和QML的技术本质和使用上，有什么区别?

QWidget是一种基于C++的桌面应用程序开发技术，主要用于开发桌面应用程序，它是一种面向对象的技术，可以使用C++语言来实现用户界面的设计和编程。 QML是一种基于JavaScript的应用程序开发技术，主要用于开发桌面应用程序和移动应用程序，它是一种基于声明式的技术，可以使用JavaScript语言来实现用户界面的设计和编程。 两者的本质有所不同，QWidget是基于C++的，QML是基于JavaScript的；使用上也有所不同，QWidget是面向对象的，QML是基于声明式的。

100、用Qt实现一个三角形的按钮，会如何实现?

首先，我们需要使用Qt的QPushButton类来创建一个按钮，然后设置按钮的样式，使其可以显示出一个三角形的形状。

1. 创建QPushButton类的实例，并设置按钮的样式：

```
QPushButton *triangleButton = new QPushButton();
triangleButton->setStyleSheet("QPushButton{border-image:url(:/images/triangle.png);}");
```

2. 设置按钮的大小：

```
triangleButton->setFixedSize(QSize(30, 30));
```

3. 连接按钮的点击信号和槽函数：

```
connect(triangleButton, SIGNAL(clicked()), this, SLOT(onTriangleButtonClicked()));
```

4. 实现槽函数：

```
void onTriangleButtonClicked()
{
   // 在这里实现点击三角形按钮时要执行的操作
}
```

101、Qt如何实现类似QQ登录窗口的翻转

Qt可以使用QPropertyAnimation类来实现QQ登录窗口的翻转效果。

1、首先，创建一个QPropertyAnimation对象，并设置动画的目标对象、属性和时间曲线：

```
QPropertyAnimation *animation = new QPropertyAnimation(this, "geometry");
animation->setDuration(500);
animation->setEasingCurve(QEasingCurve::OutExpo);
```

2、然后，设置动画的起始值和结束值：

```
//设置起始值
QRect startRect(0, 0, width(), height());
animation->setStartValue(startRect);

//设置结束值
QRect endRect(width(), 0, -width(), height());
animation->setEndValue(endRect);
```

3、最后，启动动画：

```
animation->start();
```

102、Qt窗口圆角如何实现

在Qt中实现窗口圆角，可以使用Qt的样式表实现，如下所示：

```
QWidget {
    border-radius: 10px;
}
```

可以使用如下代码来应用样式表：

```
QFile file("style.qss");
file.open(QFile::ReadOnly);
QString styleSheet = QLatin1String(file.readAll());
qApp->setStyleSheet(styleSheet);
```

103、Qt的智能指针，QSharePoint和shared_ptr有什么区别，weak_ptr呢?

Qt智能指针是一种特殊的指针，它可以指向另一个指针。它可以用来创建复杂的数据结构，如链表或树结构。

QSharePoint是一种智能指针，它可以自动管理指向的对象的内存分配和释放，从而实现自动内存管理。

shared_ptr也是一种智能指针，它可以跟踪指向的对象的引用计数，从而保证在没有任何引用的情况下，可以自动释放指向的对象。

weak_ptr是一种特殊的shared_ptr，它可以指向shared_ptr指向的对象，但不会增加指向对象的引用计数。它可以用来避免循环引用导致的内存泄漏问题。

在Qt中，指针指针（Pointer to Pointer）是一种指向指针的指针，通常用于动态分配内存或者多级指针操作。而QSharedPointer和std::shared_ptr都是C++11中的智能指针，用于管理动态内存，可以避免内存泄漏和空悬指针等问题。它们的主要区别如下：

1. QSharedPointer是Qt框架提供的智能指针，而std::shared_ptr是C++11标准库提供的智能指针。
2. QSharedPointer可以与QObject一起使用，可以自动处理QObject的引用计数，当QObject被删除时，QSharedPointer会自动释放对它的引用。而std::shared_ptr无法处理QObject的引用计数，需要手动管理。
3. QSharedPointer可以使用qSharedPointerCast进行类型转换，可以方便地将一个QSharedPointer转换为另一个QSharedPointer。而std::shared_ptr只能使用std::dynamic_pointer_cast进行类型转换。
4. QSharedPointer的默认删除器是delete，而std::shared_ptr的默认删除器是std::default_delete。

相比之下，weak_ptr则是用来解决shared_ptr循环引用问题的。当两个或多个shared_ptr相互引用时，会形成循环引用，导致内存泄漏。此时，可以使用weak_ptr来打破其中一个shared_ptr的引用，避免循环引用。weak_ptr是一种弱引用，它不会增加内存对象的引用计数，只是用来观察对象是否已经被释放，可以通过lock方法获取其对应的shared_ptr。

104、 Qt的信号与槽，有哪几种连接方式，对应的应用场景是什么？

Qt的信号与槽有三种连接方式：

1. 信号槽的直接连接：使用QObject::connect()函数连接信号和槽，当信号发出时，槽函数自动被调用，适用于信号发出者与槽函数拥有者在同一线程的场景。
2. 信号槽的槽函数链接：使用QObject::connect()函数连接信号和槽函数，当信号发出时，槽函数被调用，适用于信号发出者与槽函数拥有者不在同一线程的场景。
3. 信号槽的信号连接：使用QObject::connect()函数连接信号和信号，当信号发出时，另一个信号也会发出，适用于信号发出者与槽函数拥有者不在同一线程的场景。

105、QShareDataPoint作用

QShareDataPoint是一种用于收集和分享数据的技术。它可以帮助企业收集、分析和共享数据，以便更好地管理业务。它可以帮助企业收集和分析客户行为，改进服务质量，提高运营效率，并帮助企业更好地理解市场动态。

106、死锁怎么解决?

1. 避免死锁：可以采用一些技术避免死锁的发生，比如破坏互斥条件、破坏请求和保持条件、破坏循环等等。
2. 预防死锁：可以采用一些技术来预防死锁的发生，比如限制进程获取资源的数量、安全序列、死锁检测等等。
3. 解除死锁：可以采用一些技术来解除已经发生的死锁，比如银行家算法、延迟算法等等。

107、创建的对象有几种方式，有什么区别

Qt创建对象有两种方式：

1、使用Qt自带的构造函数，如QWidget，QPushButton，QDialog等。

2、使用Qt的meta-object系统，如QMetaObject::newInstance，QMetaObject::invokeMethod等。

这两种方式的区别在于，第一种方式是使用Qt自带的构造函数，它可以直接创建Qt对象，但是不能实现动态创建，也不能调用它们的函数或者访问它们的成员变量。

第二种方式是使用Qt的meta-object系统，它可以实现动态创建Qt对象，可以调用它们的函数或者访问它们的成员变量。

108、你能用几种方法修改QPushButton的大小，文字颜色等属性。

1. 使用Qt Designer设计师：可以使用Qt Designer设计师来调整QPushButton的大小、文字颜色等属性。
2. 使用Qt的API：可以使用Qt的API来调整QPushButton的大小、文字颜色等属性，例如：setMinimumSize()、setMaximumSize()、setStyleSheet()等。
3. 使用CSS：可以使用CSS语法来调整QPushButton的大小、文字颜色等属性，例如：QPushButton {width: 100px; height: 50px; color: red;}。

109、常用的Qt布局有几种，如何自适应缩放？

Qt布局有以下几种：

1. 绝对布局：使用绝对位置和尺寸来定位和调整控件的大小；
2. 盒子布局：使用排列和填充来定位和调整控件的大小；
3. 栅格布局：使用表格排列和调整控件的大小；
4. 流式布局：使用流动排列和调整控件的大小；
5. 堆栈布局：使用堆栈排列和调整控件的大小。

Qt支持自适应缩放，可以使用Qt的布局管理器来实现。例如，可以使用QGridLayout管理器来控制窗口的尺寸和位置，使其能够根据窗口大小的变化而自动调整控件的大小和位置。

110、Qt如何实现QQ两个客户端的私聊功能？

Qt可以通过使用Qt的网络模块来实现QQ两个客户端的私聊功能。Qt的网络模块提供了一系列的网络协议，可以用于实现QQ两个客户端之间的私聊功能。

具体的实现步骤如下：

1. 建立两个客户端的网络连接：首先，使用Qt的网络模块建立两个客户端之间的网络连接，以便进行私聊。
2. 实现私聊功能：使用Qt的网络模块实现两个客户端之间的私聊功能，以便交换文字、图片等信息。
3. 实现断开连接：在两个客户端之间的私聊结束后，可以使用Qt的网络模块实现断开连接，以便释放资源。

111、Qt的多线程，哪些是只有Qthread能实现，QtConcurrent办不到的？

1、QThread可以使用信号和槽机制，而QtConcurrent不支持。
2、QThread可以设置线程优先级，而QtConcurrent不支持。
3、QThread可以实现跨平台多线程，而QtConcurrent只能在支持C++11的平台上实现。
4、QThread可以实现更复杂的多线程任务，而QtConcurrent只能实现简单的多线程任务。

112、什么是UI线程，UI线程阻塞后会怎样？

UI线程是Android应用程序中的主线程，它负责绘制UI界面，处理用户交互，以及调度其他相关任务。如果UI线程被阻塞，用户界面将会停止响应，甚至可能会出现应用程序崩溃的情况。

113、Qt中的兄弟窗口，想刷新重叠部分，请问流程是什么样的，刷新的顺序是什么样的?

1、调用QWidget的update函数，更新当前窗口的所有内容；

2、调用QWidget的updateGeometry函数，更新当前窗口的geometry；

3、调用QWidget的update函数，更新当前窗口的子窗口；

4、调用QWidget的updateGeometry函数，更新当前窗口的子窗口的geometry；

5、调用QWidget的update函数，更新兄弟窗口；

6、调用QWidget的updateGeometry函数，更新兄弟窗口的geometry；

7、调用QWidget的update函数，更新兄弟窗口的子窗口；

8、调用QWidget的updateGeometry函数，更新兄弟窗口的子窗口的geometry；

9、调用QWidget的repaint函数，刷新重叠部分。

刷新的顺序是：更新当前窗口、更新当前窗口的子窗口、更新兄弟窗口、更新兄弟窗口的子窗口、刷新重叠部分。

114、Qt如何操作数据库

Qt操作数据库主要是使用Qt的QSqlDatabase类，它提供了一系列的函数来连接、操作、查询和管理数据库。在Qt中，可以使用QSqlQuery类来查询数据库，QSqlTableModel类来操作数据库表，QSqlRelationalTableModel类来管理关系数据库表，QSqlError类来检测错误，QSqlDriver类来检查数据库驱动程序，QSqlIndex类来创建索引，QSqlRecord类来操作数据库记录，QSqlResult类来执行查询等等。

115、Qt Remote Object的序列化与反序列化

Qt Remote Objects是一个基于Qt的远程对象系统，它可以让你在不同的进程之间共享对象。它使用Qt的序列化技术来序列化和反序列化远程对象，以便在不同的进程之间发送和接收。

Qt的序列化技术使用QDataStream类来序列化和反序列化基本的C++数据类型，QVariant和QObject的子类。它也支持自定义类型的序列化，只要实现QDataStream的operator<<和operator>>运算符重载。

Qt Remote Objects使用QDataStream来序列化和反序列化远程对象。当客户端请求远程对象时，Qt Remote Objects将使用QDataStream将远程对象序列化，然后发送给客户端。当客户端收到序列化的远程对象时，它将使用QDataStream将远程对象反序列化，然后可以访问远程对象的属性和方法。

116、什么情况下，delete需要加一个中括号[]

当要删除的变量是一个数组时，delete需要加一个中括号[ ]，例如：delete array[0];

117、描述过程，如何实现一个自定义按钮，使其在光标进入，按下，离开三种状态下显示不同的图片

1. 在项目中创建三张图片，分别表示光标进入，按下，离开三种状态下的图片。
2. 创建一个自定义按钮类，在该类中重写onMouseEnter，onMouseDown，onMouseLeave三个事件，分别设置按钮的图片为对应的三张图片。
3. 在需要使用自定义按钮的地方，实例化自定义按钮类，并将其添加到页面中，即可实现光标进入，按下，离开三种状态下显示不同的图片。

118、什么是Qt事件循环？

Qt事件循环是一种程序架构，它用于处理窗口系统和其他用户界面事件，以及与用户界面无关的事件，例如定时器和网络事件。 Qt事件循环以循环的方式运行，每次循环都会检查是否有新的事件，如果有，就会调用相应的处理程序来处理它们。

119、Qt打包程序

1. 安装pyinstaller：使用pip安装pyinstaller，在命令行输入pip install pyinstaller即可。
2. 生成可执行文件：在命令行输入pyinstaller -F -w ，其中filename为要打包的文件名。
3. 打包：将生成的可执行文件和其他需要的文件（如图片、音频等）放到一个文件夹中，然后使用pyinstaller -F -w 命令打包。
4. 安装：将生成的可执行文件安装到目标系统中即可。

120、纯虚函数和普通的虚函数有什么区别

1、纯虚函数是抽象类中的虚函数，它只有声明没有实现，它的实现由派生类完成，纯虚函数必须在派生类中实现。
2、普通虚函数是普通类中的虚函数，它有声明也有实现，派生类可以重定义它，也可以不重定义它。

121、虚继承的作用

虚继承的作用是避免了多重继承时出现的二义性问题，即消除了基类的二义性，使得子类中只有一个完整的基类对象，从而避免了多重继承时出现的二义性问题。

122、软件如果出现问题，如何去定位的，如何处理的?

定位：首先，可以通过日志记录和错误报告来定位问题，以及分析程序的运行状态，以便找出问题的根源。其次，可以使用分析工具，如性能分析工具、程序调试器等，以及通过检查程序中的代码，以及分析程序的运行状态，来定位问题。

处理：一旦定位到问题，就可以使用相应的工具或方法来处理问题，比如修改程序的代码，或者增加程序的功能，以及改善程序的性能。

123、为什么要异步刷新，如何异步刷新？

为什么要异步刷新：

异步刷新是指在不阻塞用户界面的情况下，在后台更新数据，以便及时响应用户的操作。异步刷新能够有效地提高网站的性能，提升用户体验，减少网络资源的浪费。

如何异步刷新：

异步刷新可以通过AJAX技术实现，AJAX是一种用于在后台与服务器进行数据交换的技术，它可以在不重新加载整个页面的情况下更新部分页面内容。此外，还可以使用WebSocket技术实现异步刷新，WebSocket是一种双向通信协议，它可以在浏览器和服务器之间建立持久连接，以实现双向通信。

124、windows系统下，是怎么实现窗口刷新(窗口刷新机制);是立即刷新，还是异步刷新;每次我需要一个窗口刷新，他都能立马刷新吗

Windows系统使用异步刷新机制来实现窗口刷新，这意味着窗口刷新可能不会立即发生，而是在需要的时候才发生。比如，当用户移动窗口时，Windows会收集所有窗口的移动操作，然后在一次更新中完成所有窗口的移动。这样做的好处是，它可以减少系统的负担，从而提高系统的性能。

125、如何将键盘和鼠标的相关操作过滤出来并关联到自己想要执行的函数上？

使用jQuery的事件处理函数，可以监听键盘和鼠标的操作，并将它们关联到自己想要执行的函数上。

126、C++多线程加锁，会劣化性能，请问有什么优化的手段？

1. 合理设计程序：尽量减少加锁的次数，把多个操作放到一个加锁的代码段中，减少加锁的次数。
2. 使用原子操作：C++11提供了一些原子操作，比如atomic_compare_exchange_strong()等，可以用来替代锁，在某些场合可以有效提高性能。
3. 使用锁的粒度更小的技术：比如读写锁、条件变量等，可以替代普通的互斥锁，减少锁的粒度，提高性能。
4. 使用锁的粒度更大的技术：比如使用全局锁、细粒度锁等，可以减少锁的粒度，提高性能。
5. 使用非阻塞技术：比如CAS操作，可以避免线程之间的阻塞，从而提高性能。

127、Qt 中的容器类包括

QList：动态数组，支持随机访问和快速插入、删除操作。

QVector：类似 QList，但具有更好的性能。

QLinkedList：双向链表，支持快速插入、删除操作。

QHash：哈希表，支持快速查找、插入、删除操作。

QMap：基于红黑树的映射表，支持快速查找、插入、删除操作。

128、Qt中的模型视图框架是什么？

模型视图框架是Qt中用于显示数据的一种机制。该框架将数据模型和视图分离，使得数据的表示和显示可以独立地进行管理。数据模型提供了数据的接口，视图则负责绘制和交互。Qt中提供了多种类型的模型视图类库，包括QAbstractItemModel、QStandardItemModel、QListView、QTableView等。

129、Qt中的插件是什么？

插件是Qt中用于扩展应用程序功能的一种机制。插件可以是动态链接库或静态链接库，包含了一些特定的功能。Qt提供了QPluginLoader类和QFactoryInterface类用于管理插件。通过插件，可以将应用程序的功能分解为多个独立的部分，方便开发和维护。

130、Qt中的样式表是什么？

样式表是Qt中用于定制界面风格的一种机制。样式表使用CSS语法，可以定义界面元素的属性、颜色、字体等。Qt中的样式表可以应用于整个应用程序或特定的控件，使得应用程序的界面可以与众不同。Qt还提供了QStyle类和QStyleFactory类，用于管理系统默认样式和自定义样式。

131、什么是Qt的MVC架构？

Qt的MVC架构是一种基于模型、视图和控制器的设计模式，它将应用程序的数据、用户界面和业务逻辑分离开来，使得各个部分之间的耦合度更低，易于维护和扩展。

132、什么是Qt的插件机制？

Qt的插件机制是一种将应用程序的功能模块化的方法。通过使用Qt的插件机制，可以将应用程序的一些功能打包成独立的插件，这些插件可以在运行时动态加载和卸载，从而实现应用程序的可扩展性和灵活性。

133、sizeof/strlen区别? C语言中malloc和C+ +语言中new有何区别? C/C++ 程序编译的内存分配情况？

sizeof和strlen的区别：

sizeof是一个操作符，可以用来获取一个变量或类型的字节数，不受变量值的影响。例如，sizeof(int)返回4，sizeof(char)返回1。
strlen是一个函数，用于获取一个字符串的长度，即字符数组中的字符个数，不包括字符串结束符'\0'。例如，strlen("hello")返回5。

malloc和new的区别：

malloc和new都是用于在堆上分配内存的函数。它们的主要区别在于以下几点：

malloc返回一个void*指针，需要强制转换为目标类型指针，而new直接返回目标类型指针。

malloc只负责分配内存，不会自动调用构造函数，而new在分配内存后会自动调用构造函数。

malloc分配的内存可以使用free函数释放，而new分配的内存必须使用delete操作符释放。

C/C++程序编译的内存分配情况：

在C/C++程序编译时，内存分配主要分为以下几种情况：

栈内存分配：用于存储局部变量和函数调用的参数和返回值。栈内存分配由编译器自动完成，不需要手动分配和释放。

堆内存分配：用于存储动态分配的内存，需要使用malloc或new函数手动分配，并在不需要时使用free或delete操作符释放。

全局变量和静态变量分配：在程序运行前就进行内存分配，存储在静态存储区或全局存储区，程序结束后才会释放。

134、strcpy/sprintf/memcpy它们之间区别?

strcpy、sprintf和memcpy都是C语言中的字符串和内存操作函数，它们之间的主要区别如下：

strcpy用于将一个字符串复制到另一个字符串中，比如将src字符串复制到dest字符串中。函数原型为：char *strcpy(char *dest, const char *src)；

sprintf用于将格式化的数据转换成字符串并存储到另一个字符串中，比如将格式化的数字或文本保存到buf字符串中。函数原型为：int sprintf(char *buf, const char *format, ...)；

memcpy用于将一段内存的内容复制到另一个内存中，比如将src内存的内容复制到dest内存中。函数原型为：void *memcpy(void *dest, const void *src, size_t n)；
总的来说，这三个函数的作用不同，strcpy和memcpy主要用于字符串和内存的复制，而sprintf主要用于格式化数据的转换。使用时需要结合具体的需求选择合适的函数来操作。

135、面向对象的三大特征? C+ +语言的空类有哪些成员函数?

面向对象的三大特征是：

封装：将数据和操作数据的函数封装在一个类中，隐藏了具体实现细节，只暴露必要的接口给外部使用，保证了数据的安全性和可靠性。

继承：通过继承已有的类，可以扩展其功能，减少代码的冗余，提高代码的复用性和可维护性。

多态：通过函数重载、虚函数和模板等机制，实现不同对象对同一消息的不同响应，提高了程序的灵活性和可扩展性。

C++语言的空类是指没有任何成员变量和成员函数的类。空类默认会自动生成一些成员函数，包括默认构造函数、析构函数、拷贝构造函数和赋值运算符等。如果需要控制这些默认生成的成员函数的行为，可以通过定义相应的函数来实现。

136、多态实现的原理?链表和数组有何区别?队列和栈区别?

多态实现的原理：

多态是面向对象编程中的一种重要概念，实现多态的关键是虚函数和指针。在C++中，通过将基类中的某个函数声明为虚函数，可以使得派生类中的同名函数成为虚函数，从而实现多态。当通过基类指针或引用调用虚函数时，会根据指针或引用实际指向的对象类型来调用相应的函数，实现了动态绑定。这样就可以在不同的对象之间实现相同的操作，提高了代码的复用性和可维护性。

链表和数组的区别：

链表和数组都是常见的数据结构，它们的主要区别在于：

数组是一组具有相同数据类型的元素的集合，可以通过下标来访问每个元素，但是数组的长度是固定的，不能动态改变。

链表是由一组节点构成的数据结构，每个节点包含数据和指向下一个节点的指针，可以动态添加、删除、修改节点，但是访问链表中的元素需要遍历整个链表，效率较低。

队列和栈的区别：

队列和栈都是常见的数据结构，它们的主要区别在于：

队列是一种先进先出（FIFO）的数据结构，可以在队列的一端插入元素，在队列的另一端删除元素，典型的应用是操作系统的进程调度。

栈是一种后进先出（LIFO）的数据结构，可以在栈顶插入和删除元素，典型的应用是函数调用和表达式求值。

137、多态实现的原理?链表和数组有何区别?队列和栈区别?

多态实现的原理：

多态是面向对象编程中的一种重要概念，实现多态的关键是虚函数和指针。在C++中，通过将基类中的某个函数声明为虚函数，可以使得派生类中的同名函数成为虚函数，从而实现多态。当通过基类指针或引用调用虚函数时，会根据指针或引用实际指向的对象类型来调用相应的函数，实现了动态绑定。这样就可以在不同的对象之间实现相同的操作，提高了代码的复用性和可维护性。

链表和数组的区别：

链表和数组都是常见的数据结构，它们的主要区别在于：

数组是一组具有相同数据类型的元素的集合，可以通过下标来访问每个元素，但是数组的长度是固定的，不能动态改变。

链表是由一组节点构成的数据结构，每个节点包含数据和指向下一个节点的指针，可以动态添加、删除、修改节点，但是访问链表中的元素需要遍历整个链表，效率较低。

队列和栈的区别：

队列和栈都是常见的数据结构，它们的主要区别在于：

队列是一种先进先出（FIFO）的数据结构，可以在队列的一端插入元素，在队列的另一端删除元素，典型的应用是操作系统的进程调度。

栈是一种后进先出（LIFO）的数据结构，可以在栈顶插入和删除元素，典型的应用是函数调用和表达式求值。

138、&&/ & II |有什么区别? Typedef/define/const/static 有什么区别?

&&是逻辑与运算符，表示两个条件都为真时结果为真。

&是位运算符，表示对两个数的位进行与运算。

||是逻辑或运算符，表示两个条件中有一个为真时结果为真。

|是位运算符，表示对两个数的位进行或运算。

Typedef/define/const/static 有什么区别？

typedef：用来给数据类型取一个新的名字，方便在程序中使用。例如：typedef int INT; 表示将int类型取一个新的名字INT。

define：用来定义常量或宏，会在编译时被预处理器替换为定义的内容。例如：#define PI 3.14159 表示将PI定义为3.14159。

const：用来定义常量，在程序中不可修改。例如：const int MAX_NUM = 100; 表示将MAX_NUM定义为100，不可修改。

static：用来定义静态变量，静态变量只会被初始化一次，不会随函数的调用而重复初始化。静态变量的作用域只在定义它的函数内，但是它的值会被保留。静态函数只能在定义它的文件内使用。

139、如何避免“野指针”?

野指针是指指向无效内存地址的指针，它的出现往往会导致程序崩溃或者产生不可预期的结果。为了避免野指针的出现，可以采取以下措施：

将指针初始化为NULL或nullptr，即空指针，可以防止指针误用。

在指针被释放之后，将其赋值为NULL或nullptr，可以避免指针成为野指针。

使用智能指针，智能指针可以自动管理动态内存，避免内存泄漏和野指针问题。

避免指针操作过程中越界访问数组，可以使用STL容器代替数组，STL容器可以自动管理内存。

对于指向栈上的变量的指针，需要注意其生命周期，避免在变量被销毁后仍然使用指针。
总之，在程序中使用指针时，需要时刻保持警惕，避免出现野指针问题。

140、向链表的末尾添加一个元素?从链表尾部到头部打印结点信息?如何合并两个有序链表?

向链表的末尾添加一个元素的步骤如下：

新建一个节点，为其赋值；

遍历链表，找到最后一个节点；

将最后一个节点的next指针指向新节点；

新节点的next指针赋值为NULL，表示链表的末尾。

从链表尾部到头部打印节点信息的步骤如下：

遍历链表，将每个节点的值存放在栈中；

遍历栈，将栈中的值弹出并输出，即为链表的逆序输出。

合并两个有序链表的步骤如下：

新建一个空链表，作为合并后的链表；

遍历两个有序链表，比较当前两个链表的节点值，将较小的节点插入到新链表中；

如果其中一个链表已经遍历完，将另一个链表剩余的节点插入到新链表的末尾；

返回新链表。

示例代码如下：

```
// 向链表的末尾添加一个元素
void addNode(Node* head, int val) {
    Node* newNode = new Node(val);
    Node* cur = head;
    while (cur->next != NULL) {
        cur = cur->next;
    }
    cur->next = newNode;
}
// 从链表尾部到头部打印节点信息
void printListReverse(Node* head) {
    stack<Node*> s;
    Node* cur = head;
    while (cur != NULL) {
        s.push(cur);
        cur = cur->next;
    }
    while (!s.empty()) {
        Node* node = s.top();
        s.pop();
        cout << node->val << " ";
    }
}
// 合并两个有序链表
Node* mergeList(Node* head1, Node* head2) {
    Node* dummy = new Node(-1);
    Node* cur = dummy;
    while (head1 != NULL && head2 != NULL) {
        if (head1->val < head2->val) {
            cur->next = head1;
            head1 = head1->next;
        } else {
            cur->next = head2;
            head2 = head2->next;
        }
        cur = cur->next;
    }
    if (head1 != NULL) {
```

```
        `cur->next = head1;`
    `}`
    `if (head2 != NULL) {`
        `cur->next = head2;`
    `}`
    `return dummy->next;`
`}`
```

141、如何反转链表?判断链表是否是回文链表?如何判断链表相交?

1. 反转链表的步骤如下：

- 新建一个空链表，作为反转后的链表；
- 遍历原链表，将每个节点插入到新链表的头部，即使新节点成为新链表的头节点；
- 返回新链表。 示例代码如下：

```
Node* reverseList(Node* head) {
    Node* newHead = NULL;
    Node* cur = head;
    while (cur != NULL) {
        Node* next = cur->next;
        cur->next = newHead;
        newHead = cur;
        cur = next;
    }
    return newHead;
}
```

2.判断链表是否是回文链表的步骤如下：

- 使用快慢指针法，找到链表的中间节点；
- 将链表的后半部分反转；
- 比较前半部分和后半部分是否相同；
- 将后半部分反转回来；
- 返回比较结果。 示例代码如下：

```
bool isPalindrome(Node* head) {
    if (head == NULL || head->next == NULL) {
        return true;
    }
    Node* slow = head;
    Node* fast = head;
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }
    Node* newHead = reverseList(slow);
    Node* cur1 = head;
```

```
    Node* cur2 = newHead;
    bool res = true;
    while (cur1 != NULL && cur2 != NULL) {
        if (cur1->val != cur2->val) {
            res = false;
            break;
        }
        cur1 = cur1->next;
        cur2 = cur2->next;
    }
    reverseList(newHead);
    return res;
}
```

3.判断链表相交的步骤如下：

- 遍历两个链表，分别得到它们的长度len1和len2；
- 让较长的链表从头开始走len1-len2步，使得两个链表到达相同的位置；
- 同时遍历两个链表，找到第一个相同的节点，即为相交节点；
- 如果两个链表没有相交，则返回NULL；
- 返回相交节点。示例代码如下：

```
Node* getIntersectionNode(Node* headA, Node* headB) {
    int len1 = 0, len2 = 0;
    Node* cur1 = headA;
    Node* cur2 = headB;
    while (cur1 != NULL) {
        len1++;
        cur1 = cur1->next;
    }
    while (cur2 != NULL) {
        len2++;
        cur2 = cur2->next;
    }
    cur1 = headA;
    cur2 = headB;
    int diff = abs(len1 - len2);
    if (len1 > len2) {
        while (diff-- > 0) {
            cur1 = cur1->next;
        }
    } else {
        while (diff-- > 0) {
            cur2 = cur2->next;
        }
    }
    while (cur1 != NULL && cur2 != NULL) {
        if (cur1 == cur2) {
            return cur1;
        }
        cur1 = cur1->next;
        cur2 = cur2->next;
    }
```

```
    return NULL;
}
```

142、假设现有n个有序数组，如何合并成一个有序数组?

假设有n个有序数组A1、A2、...、An，每个数组的长度分别为L1、L2、...、Ln，合并成一个有序数组的步骤如下：

从每个数组中取出第一个元素，将它们放入一个最小堆中；

取出堆顶元素（即当前n个数组中最小的元素），将它放入结果数组中；

如果该元素来自某个数组的末尾，则不再将它的下一个元素加入堆中；

如果堆为空，则说明所有元素都已经被取出，合并结束。
时间复杂度为O(knlogn)，其中k为数组中元素的平均数量。
示例代码如下：

```
vector<int> mergeKSortedArrays(vector<vector<int>>& arrays) {
    vector<int> res;
    int n = arrays.size();
    if (n == 0) {
        return res;
    }
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;
    for (int i = 0; i < n; i++) {
        if (arrays[i].size() > 0) {
            pq.push({arrays[i][0], i});
        }
    }
    while (!pq.empty()) {
        auto p = pq.top();
        pq.pop();
        int val = p.first;
        int i = p.second;
        res.push_back(val);
        if (arrays[i].size() > 1) {
            pq.push({arrays[i][1], i});
        }
        arrays[i].erase(arrays[i].begin());
    }
    return res;
}
```

143、栈和队列、字符串、树、递归、AVL树、红黑树、哈弗曼编码、B+树、map/unordered map、动态规划。

栈和队列、字符串、树、递归、AVL树、红黑树、哈夫曼编码、B+树、map/unordered map、动态规划是常见的数据结构或算法。

栈和队列：栈和队列是线性数据结构，栈是后进先出，队列是先进先出。

字符串：字符串是由字符组成的序列，常用于文本处理、模式匹配等。

树：树是非线性数据结构，由节点和边组成，每个节点可以有多个子节点。

递归：递归是一种算法或编程技巧，通过函数自身调用实现问题的分解和求解。

AVL树：AVL树是自平衡二叉搜索树，保证任意节点的左右子树高度差不超过1。

红黑树：红黑树也是自平衡二叉搜索树，通过变色和旋转操作来保持平衡。

哈夫曼编码：哈夫曼编码是一种将字符编码为二进制的算法，通过构建哈夫曼树来实现。

B+树：B+树是一种多路搜索树，常用于数据库索引等场景。

map/unordered map：map和unordered map是键值对的容器，支持O(1)的查找和O(logn)的插入、删除等操作。

动态规划：动态规划是一种求解最优化问题的算法，通过将大问题分解为小问题求解。

144、什么时候产生默认拷贝构造函数?什么是深拷贝?什么是浅拷贝?

默认拷贝构造函数会在以下情况下自动生成：

如果没有定义自己的拷贝构造函数，且类的成员变量都是可拷贝的，则编译器会自动生成默认的拷贝构造函数；

如果定义了拷贝构造函数，但没有实现任何操作，则编译器也会自动生成默认的拷贝构造函数。

深拷贝和浅拷贝是指在拷贝对象时，是否会将对象的动态内存也复制一份。
深拷贝是指在拷贝对象时，会将对象的动态内存也复制一份，每个对象都有自己的一份动态内存，互不干扰。

浅拷贝是指在拷贝对象时，只是将对象的指针或引用复制一份，两个指针指向同一个动态内存，修改其中一个对象的动态内存会影响到另一个对象。

在使用动态内存分配时，如果不进行深拷贝，可能会导致多个对象共享同一块动态内存，当一个对象释放动态内存时，其他对象也会受到影响，可能会导致程序崩溃或数据错误。因此，在使用动态内存分配时，通常需要进行深拷贝。

142、虚析函数的作用? Vector 底层实现原理?

虚析函数的作用是防止在子类析构时只调用了父类的析构函数，而没有调用子类的析构函数，导致内存泄漏和资源释放不完全的问题。通过将基类析构函数声明为虚析函数，可以确保在子类析构时会先调用子类的析构函数，再调用父类的析构函数，从而避免这种问题。

Vector是一种动态数组，底层实现原理是使用连续的内存块来存储元素，并且在需要扩容时重新分配更大的内存块，并将原有元素拷贝到新的内存块中。具体来说，Vector在创建时会分配一块初始大小的内存，当元素个数超过该内存大小时，会重新分配一块更大的内存，并将原有元素拷贝到新内存中，同时释放原有内存。此过程涉及到内存分配、拷贝和释放，因此Vector的效率受到内存管理的影响。为了避免频繁的内存分配和拷贝，Vector通常会预先分配一定的内存空间，以减少扩容的次数。

143、deque底层实现原理?

deque（双端队列）底层实现原理是使用一段连续的存储空间，被分配为多个内存块，每个内存块独立分配，内部使用指针互相连接。deque中包含一个中控器，中控器中存放着指向第一块内存块和最后一块内存块的迭代器，以及指向每个内存块的迭代器，中控器的作用是管理内存块的分配和释放，并提供访问内存块的接口。

deque的元素在内存中并不是连续存储的，而是分散存储在不同的内存块中，但是中控器提供的接口使得deque的使用者可以像访问连续存储空间一样访问deque的元素，即可以使用迭代器、下标操作符等对deque进行遍历和访问。

对于deque的插入和删除操作，由于元素在内存中是分散存储的，因此需要在中控器中进行内存块的分配和释放，并更新迭代器指向。由于deque的实现比较复杂，因此相比于vector等容器，deque的效率相对较低，但是deque在某些场景下具有优势，比如插入和删除操作比较频繁、需要在deque的两端进行操作等。

144、左值引用与右值引用区别?右值引用意义?

左值引用和右值引用是C++11中引入的新特性，它们的主要区别在于引用的类型和绑定的对象。

左值引用是指对一个左值（可以取地址、有名字的变量或者对象、表达式或函数返回值）进行的引用，它的类型为T&，可以对其进行修改或者取地址等操作。

右值引用是指对一个右值（不能取地址、没有名字的临时对象或者表达式）进行的引用，它的类型为T&&，通常用于移动语义和完美转发等场景。右值引用的一个重要作用是支持移动语义，即通过将右值引用参数移动到目标对象中，避免了昂贵的内存拷贝操作，提高了程序的效率。

右值引用的意义在于提供了一种新的引用类型，使得我们可以更加高效地处理临时对象和表达式，同时也支持了新的语言特性，比如移动语义和完美转发等。右值引用还可以用于实现移动构造函数和移动赋值运算符等操作，从而提高程序的效率和性能。

145、索引为什么要使用B+树而不是:二叉树或者B树?

索引是数据库中常用的数据结构，用于加速数据的查询和检索。B+树相对于二叉树和B树的优势在于其更适合于磁盘存储的特点，主要表现在以下几个方面：

减少磁盘I/O开销：B+树的每个节点可以存储更多的关键字，因此可以减少磁盘I/O操作的次数。相比较而言，B树每个节点的关键字数目较少，因此需要更多的I/O操作，而二叉树的高度也可能很大，导致I/O次数增多。

便于范围查询：B+树的叶子节点形成了一个有序链表，可以方便地进行范围查询。相比较而言，B树和二叉树需要在非叶子节点进行回溯才能找到所有满足条件的记录，这会增加额外的开销。

便于扫描操作：B+树的叶子节点形成了一个有序链表，可以方便地进行顺序扫描，以及分页查询等操作。相比较而言，B树和二叉树需要在非叶子节点进行回溯才能找到下一个节点，这会增加额外的开销。
因此，在需要进行大量磁盘I/O操作的场景下，B+树是更加合适的选择。而在内存中的数据结构，比如红黑树和哈希表等，由于磁盘I/O操作的次数较少，因此使用B+树并不会带来很大的优势。

146、SQL流入原理?如何避免SQL注入?

SQL注入是一种常见的Web攻击方式，攻击者通过在输入框中注入特定的SQL语句，从而获取敏感数据或者控制数据库。为了避免SQL注入，需要采取一些措施，包括以下几个方面：

使用参数化查询：参数化查询可以避免将用户输入的数据直接拼接到SQL语句中，从而避免SQL注入。参数化查询可以将用户输入的数据作为参数传递给SQL语句，而不是直接拼接到SQL语句中。这样，即使用户输入了恶意的SQL语句，也不会对数据库造成影响。

对用户输入进行过滤和验证：对用户输入进行验证和过滤可以避免非法字符和SQL关键字的注入。例如，可以使用正则表达式或者特定的函数对用户输入进行过滤，从而确保输入的数据符合预期的格式。

最小化数据库权限：为了最小化数据库被攻击的风险，应该为数据库分配最小的权限。例如，只授权给应用程序需要的最小权限，避免将管理员权限授予普通用户。

使用防火墙和安全软件：使用防火墙和安全软件可以检测和拦截恶意的SQL语句，从而保护数据库的安全。
SQL流入是指将SQL注入攻击的尝试记录下来，分析攻击的方式和来源，从而采取相应的措施。SQL流入可以通过日志分析软件或者数据库自身的审计功能来实现。通过SQL流入，可以及时发现SQL注入攻击，采取相应的措施保护数据库的安全。

147、MySQL死锁问题产生原因及如何解决?

MySQL死锁是指两个或多个事务相互等待对方释放锁，从而导致事务无法继续执行的情况。产生死锁的原因主要有以下几个方面：

并发访问：多个事务同时访问同一个资源（例如表、行、页等），并且访问方式不同（例如读、写等），从而导致锁的冲突。

锁的顺序：多个事务请求锁的顺序不同，从而导致死锁的产生。

锁的粒度：锁的粒度太细或者太大，都可能导致死锁的产生。
为了解决MySQL死锁问题，可以采取以下几个措施：

优化SQL语句：通过优化SQL语句，减少事务的数量和时间，从而减少死锁的发生概率。

调整锁的粒度：通过调整锁的粒度，使得锁的数量和范围都适当，从而减少死锁的发生概率。

调整事务隔离级别：通过调整事务隔离级别，使得事务的访问方式和范围适当，并且避免两个事务同时对同一个资源进行修改。

监控死锁：通过监控死锁，及时发现死锁的产生，并采取相应的措施解决死锁问题。

加锁顺序：使用相同的加锁顺序，从而避免死锁问题的发生。

限制事务等待时间：通过限制事务等待时间，使得事务在等待一定时间后自动回滚，从而避免死锁的长时间占用资源。
通过以上措施，可以有效地解决MySQL死锁问题，提高数据库的性能和稳定性。

148、TCP三次握手的过程/为什么不可以两次握手?

TCP三次握手是TCP协议建立可靠连接的过程，其过程如下：

客户端发送SYN包给服务器，表示请求建立连接。

服务器收到客户端的SYN包后，发送ACK包给客户端，表示确认请求，并发送自己的SYN包给客户端。

客户端收到服务器的ACK包和SYN包后，发送ACK包给服务器，表示确认连接建立。

因为TCP协议是面向连接的协议，需要在建立连接之后才能进行数据传输。三次握手的过程可以确保建立连接的可靠性，防止因为网络延迟或者丢包导致连接建立失败。如果只进行两次握手，就不能确定对方已经接收到自己的SYN包，也就不能确定是否建立连接。例如，如果客户端发送SYN包后，服务器没有收到，那么客户端会一直等待ACK包，而服务器会认为连接已经建立。这样就会导致数据的不可靠性和安全性问题。因此，TCP协议需要进行三次握手，以确保连接的可靠性。

149、TCP四次挥手的过程? TCP是如何保证可靠性?

CP四次挥手是TCP协议结束连接的过程，其过程如下：

客户端发送FIN包给服务器，表示请求关闭连接。

服务器收到客户端的FIN包后，发送ACK包给客户端，表示确认关闭请求，并且告诉客户端可以关闭连接了。

服务器发送自己的FIN包给客户端，表示请求关闭连接。

客户端收到服务器的FIN包后，发送ACK包给服务器，表示确认关闭请求，并且告诉服务器可以关闭连接了。
TCP协议通过以下几个机制来保证可靠性：

序列号和确认应答：TCP协议在传输数据时，会为每一个数据包设置一个序列号，接收方收到数据包后，需要发送一个确认应答包，告诉发送方收到了哪些数据包。如果发送方没有收到确认应答包，就会重新发送数据包，直到接收方确认收到为止。

超时重传：如果发送方发送了一个数据包，但是没有收到确认应答包，就会重新发送数据包，直到收到确认应答或者达到重传次数上限为止。

滑动窗口：TCP协议中，发送方和接收方都有一个滑动窗口，用来控制数据流的传输。发送方根据接收方的确认应答，动态调整窗口大小，控制数据的发送速度。接收方根据需要，动态调整窗口大小，控制数据的接收速度。

流量控制：TCP协议中，通过滑动窗口来控制数据流的传输，可以防止发送方发送过多的数据，导致接收方无法处理。

拥塞控制：TCP协议中，通过动态调整窗口大小和发送速度，来控制网络拥塞的发生，保证网络的稳定性和可靠性。
通过以上机制，TCP协议可以保证数据的可靠传输，同时也可以保证网络的稳定性和可靠性。

150、什么是连接半打开,头关闭状态?

连接半打开状态（TCP SYN_SENT状态）是指TCP连接建立过程中，客户端发送SYN包给服务器，但是服务器还没有发送ACK包进行确认的状态。在这个状态下，客户端等待服务器的确认，如果服务器没有响应，则客户端会发送多个SYN包，直到建立连接或者达到重试次数上限为止。

头关闭状态（TCP FIN_WAIT_1和FIN_WAIT_2状态）是指TCP连接关闭过程中，发送方（可以是客户端也可以是服务器）发送FIN包给接收方，请求关闭连接之后，等待接收方的ACK包的状态。在FIN_WAIT_1状态下，发送方等待接收方的ACK包，如果接收方没有回复，则发送方会重新发送FIN包；在FIN_WAIT_2状态下，发送方等待接收方的FIN包，如果接收方没有发送FIN包，则发送方会一直等待直到超时。

在连接半打开状态和头关闭状态下，TCP连接可能会出现异常情况，例如网络延迟、丢包等，导致连接无法正常建立或关闭。因此，在使用TCP协议时，需要注意处理连接半打开状态和头关闭状态，以确保连接的可靠性和稳定性。

151、Qt信号槽机制的优点及缺点?

Qt信号槽机制是一种事件驱动的编程模型，它的优点和缺点如下：

优点：

松耦合：信号槽机制可以实现组件之间的松耦合，组件之间不需要直接相互调用，只需要通过信号和槽进行通信即可，这样可以降低组件之间的耦合度，提高代码的复用性和可维护性。

灵活性：信号槽机制可以实现非常灵活的事件处理，可以动态的连接和断开信号和槽，可以在运行时动态修改信号和槽的参数等。

易于扩展：信号槽机制可以非常容易地扩展新的事件和处理逻辑，只需要定义新的信号和槽即可，无需修改原有代码。

跨线程：信号槽机制可以支持跨线程的事件处理，可以将信号和槽连接在不同的线程中，这样可以实现线程之间的通信。

缺点：

性能：信号槽机制的性能相对于直接调用函数来说会有一定的开销，因为它需要进行信号的发射和槽的执行，而且还需要维护信号槽的连接关系。

调试：由于信号槽机制是基于事件驱动的编程模型，因此调试起来可能会比较困难，特别是在信号和槽之间存在多层嵌套的情况下。

安全性：由于信号槽机制可以动态连接和断开信号和槽，因此在使用时需要注意安全性问题，避免出现槽函数被误调用的情况。
总的来说，Qt信号槽机制是一种非常灵活和方便的事件驱动编程模型，它的优点在于松耦合、灵活性、易于扩展和跨线程等方面，但是在性能、调试和安全性等方面需要注意一些问题。

152、Qt如何实现自定义按钮，使其在光标进入、按下、离开三种状态下显示不同的图片?

Qt可以通过派生QPushButton类并重载其鼠标事件函数来实现自定义按钮，具体步骤如下：

1. 定义一个新的类，继承自QPushButton。
2. 在类的构造函数中设置按钮的三种状态下的图片。
3. 重载鼠标进入事件(QEvent::Enter)、鼠标按下事件(QEvent::MouseButtonPress)和鼠标离开事件(QEvent::Leave)函数，分别在这三个事件中设置按钮的状态图片。 示例代码：

```
class MyButton : public QPushButton
{
public:
    MyButton(QWidget *parent = nullptr) : QPushButton(parent)
    {
        // 设置按钮的三种状态图片
        setStyleSheet("QPushButton {border-image: url(normal.png);}"
                      "QPushButton:hover {border-image: url(hover.png);}"
                      "QPushButton:pressed {border-image: url(pressed.png);}");
    }
protected:
    void enterEvent(QEvent *event) override
    {
        // 鼠标进入事件，设置按钮状态为hover
        QPushButton::enterEvent(event);
        setStyleSheet("QPushButton {border-image: url(normal.png);}"
                      "QPushButton:hover {border-image: url(hover.png);}"
                      "QPushButton:pressed {border-image: url(pressed.png);}");
    }
    void mousePressEvent(QMouseEvent *event) override
    {
        // 鼠标按下事件，设置按钮状态为pressed
        QPushButton::mousePressEvent(event);
        setStyleSheet("QPushButton {border-image: url(normal.png);}"
                      "QPushButton:hover {border-image: url(hover.png);}"
                      "QPushButton:pressed {border-image: url(pressed.png);}");
    }
    void leaveEvent(QEvent *event) override
    {
        // 鼠标离开事件，设置按钮状态为normal
        QPushButton::leaveEvent(event);
        setStyleSheet("QPushButton {border-image: url(normal.png);}"
                      "QPushButton:hover {border-image: url(hover.png);}"
                      "QPushButton:pressed {border-image: url(pressed.png);}");
    }
};
```

在这个示例代码中，我们派生了一个新的类MyButton，重载了它的鼠标进入、按下和离开事件函数，根据不同的事件类型，设置按钮的状态图片。当鼠标进入按钮区域时，按钮的样式会变成hover状态，当鼠标按下按钮时，按钮的样式会变成pressed状态，当鼠标离开按钮时，按钮的样式会变成normal状态。

153、Qt信号和槽本质?

Qt中的信号和槽是一种事件驱动的机制，用于在对象之间传递消息和实现对象间的通信。信号和槽的本质可以解释为以下两个方面：

信号和槽本质是函数：

在Qt中，信号和槽本质上是函数，信号是一种特殊的函数，它不包含函数体，只有函数声明和参数列表，用于向外部发出某种事件的通知。槽也是一种函数，它是一个普通的成员函数，用于处理信号发出的事件。在信号和槽连接时，实际上是将信号函数和槽函数通过一个中介对象（连接器）连接起来，使得信号函数能够调用槽函数，从而实现对象间的通信。

信号和槽本质是元对象系统的一部分：

在Qt中，信号和槽是元对象系统的一部分，元对象系统是一个用于支持Qt元编程的框架，它允许在运行时动态地查询和操作对象的元数据，包括类名、属性、方法等。信号和槽的实现依赖元对象系统的元数据，每个QObject对象在创建时都会生成一个元对象，其中包含了该对象的所有信息，包括信号和槽的声明和实现。当信号和槽连接时，会根据元数据进行检查和匹配，保证信号和槽的正确连接和调用。

综上所述，信号和槽本质上是函数，它们通过元对象系统实现对象间的通信，使得对象之间能够相互响应和交互，是Qt中非常重要的一种机制。

154、Qt当中的数据流(QDataStream) 和文件流(QTextStream) 有何区别?

Qt中的数据流(QDataStream)和文件流(QTextStream)都是用于读写数据的流类，但它们有以下区别：

数据类型：数据流(QDataStream)支持Qt中的所有基本数据类型和自定义类型，如QString、QByteArray等，而文件流(QTextStream)只能读写文本数据，不支持二进制数据。

数据格式：数据流(QDataStream)是二进制格式，可以直接读写二进制数据，而文件流(QTextStream)是文本格式，只能读写文本数据，对于二进制数据需要进行编码和解码。

数据编码：文件流(QTextStream)默认使用Unicode编码，可以通过设置不同的编码格式来读写不同的文本数据，而数据流(QDataStream)不需要进行编码，它直接以二进制形式读写数据。

应用场景：数据流(QDataStream)适用于读写二进制数据，例如读写文件、网络传输等场景，而文件流(QTextStream)适用于读写文本数据，例如读写配置文件、日志文件等场景。

综上所述，数据流(QDataStream)和文件流(QTextStream)虽然都是Qt中的流类，但它们的数据类型、格式、编码和应用场景等方面有所不同，需要根据具体的需求选择合适的流类进行读写操作。

155、Qt 网络通信中，TCP./UDP 整体流程(服务器，客户端)？

在Qt中，TCP/UDP网络通信的整体流程主要包括以下步骤：

服务器端的创建和监听

服务器端首先需要创建一个QTcpServer或QUdpSocket对象，用于监听客户端的连接或接收数据。创建时需要指定端口号和IP地址（如果有多个网卡，则需要指定监听的网卡地址）。然后调用listen()函数开始监听客户端的连接或数据包的到来。

客户端的连接或数据发送

客户端需要创建一个QTcpSocket或QUdpSocket对象，用于连接服务器或发送数据包。对于TCP通信，客户端需要调用connectToHost()函数连接服务器；对于UDP通信，客户端可以直接使用writeDatagram()函数发送数据包。

服务器端的响应和数据处理

当客户端连接到服务器或发送数据包时，服务器端会触发相应的信号（如newConnection()、readyRead()等），在相应的槽函数中进行响应和数据处理。对于TCP通信，服务器端需要在newConnection()槽函数中调用nextPendingConnection()函数获取客户端的QTcpSocket对象，然后在客户端的QTcpSocket对象上调用read()函数读取数据；对于UDP通信，服务器端可以直接在readyRead()槽函数中调用readDatagram()函数读取数据包。

客户端的数据接收和响应

当客户端连接到服务器或发送数据包时，客户端会触发相应的信号（如connected()、readyRead()等），在相应的槽函数中进行数据接收和响应。对于TCP通信，客户端需要在connected()槽函数中调用write()函数发送数据，然后在readyRead()槽函数中调用read()函数读取服务器端的响应数据；对于UDP通信，客户端可以直接在readyRead()槽函数中调用readDatagram()函数读取服务器端的响应数据包。

断开连接和清理资源

当通信完成或出现错误时，需要断开连接并清理资源。对于TCP通信，可以在客户端或服务器端的QTcpSocket对象上调用disconnectFromHost()函数或close()函数断开连接，并在socketDisconnected()槽函数中清理资源；对于UDP通信，不需要显式地断开连接，系统会自动管理资源。

综上所述，TCP/UDP网络通信的整体流程包括服务器端的创建和监听、客户端的连接或数据发送、服务器端的响应和数据处理、客户端的数据接收和响应，以及断开连接和清理资源。根据具体的需求，可以选择使用QTcpServer、QTcpSocket、QUdpSocket等类进行开发。

156、Qt编程当中，多线程的两种使用方法?

在Qt编程中，多线程的两种使用方法分别是：

1. 继承QThread类 继承QThread类是一种常见的多线程编程方法。该方法需要定义一个新类，继承自QThread类，并重写run()函数，run()函数中包含了需要在新线程中执行的代码。在主线程中创建该类的实例对象，调用start()函数启动新线程。 示例代码：

```
class MyThread : public QThread
{
    Q_OBJECT
public:
    void run() override
    {
        // 在新线程中执行的代码
    }
};
MyThread thread;
thread.start();
```

2.继承QObject类，使用QThread对象 继承QObject类，使用QThread对象是另一种常见的多线程编程方法。该方法需要定义一个新类，继承自QObject类，并在该类中定义需要在新线程中执行的槽函数。在主线程中创建QThread对象，将新类的实例对象移动到该QThread对象所在的线程中，然后启动该QThread对象。 示例代码：

```
class MyObject : public QObject
{
    Q_OBJECT
public slots:
    void doWork()
    {
        // 在新线程中执行的代码
    }
};
QThread thread;
MyObject *obj = new MyObject();
obj->moveToThread(&thread);
QObject::connect(&thread, &QThread::started, obj, &MyObject::doWork);
thread.start();
```

以上两种方法都可以实现多线程编程，选择哪种方法取决于具体的需求和编程风格。需要注意的是，在多线程编程中，需要注意线程间的同步和互斥问题，避免出现竞态条件和死锁等问题。可以使用Qt提供的线程同步机制（如QMutex、QWaitCondition等）或标准的C++11线程库（如std::mutex、std::condition_variable等）进行处理。

157、创建signal类? QVariant 应用?

在Qt中，创建signal类可以通过继承QObject类并使用signals关键字来声明信号。一个signal类通常包含一个或多个信号，每个信号都由一个信号名称和一个信号参数列表组成，其中信号参数列表可以为空。示例代码如下：

```
class MySignalClass : public QObject
{
    Q_OBJECT
public:
    MySignalClass(QObject *parent = nullptr) : QObject(parent) {}
signals:
    void mySignal(int value);
    void anotherSignal(QString text, int count);
};
```

在上面的代码中，MySignalClass类继承自QObject类，并使用signals关键字声明了两个信号，分别是mySignal和anotherSignal，其中mySignal信号有一个int类型的参数，anotherSignal信号有一个QString类型和一个int类型的参数。 QVariant是Qt中一个用于封装任意类型数据的类，它可以用于在不同类型之间进行数据转换。可以使用QVariant::fromXXX()函数将不同类型的数据转换为QVariant类型，如QVariant::fromInt()、QVariant::fromString()等；可以使用QVariant::toXXX()函数将QVariant类型的数据转换为其他类型，如QVariant::toInt()、QVariant::toString()等。示例代码如下：

```
QVariant var = QVariant::fromInt(123); // 将int类型的数据转换为QVariant类型
int value = var.toInt(); // 将QVariant类型的数据转换为int类型
```

在上面的代码中，QVariant::fromInt()函数将int类型的数据转换为QVariant类型，QVariant::toInt()函数将QVariant类型的数据转换为int类型。需要注意的是，QVariant类型的数据转换可能会存在精度损失和类型错误等问题，需要根据实际情况进行处理。

158、Qt中的指针: QPointer、QScopedPointer、 QSharedPointer、QWeakPointer、std::weak_ ptr、QSharedDataPointer?

在Qt中，指针是一个很重要的概念，Qt提供了多种指针类来帮助我们管理内存和避免内存泄漏。下面是对常见的指针类进行简要介绍：

QPointer
QPointer是Qt中的智能指针，它可以自动管理指针的生命周期，并且可以检测被管理的对象是否已经被销毁。如果被管理的对象已经被销毁，QPointer会自动将指针置为nullptr。QPointer通常用于管理QWidget对象等需要在程序运行期间动态创建和销毁的对象。

QScopedPointer
QScopedPointer是Qt中的局部智能指针，它可以自动管理指针的生命周期，并且可以在指针超出作用域时自动释放指针指向的内存。QScopedPointer通常用于管理局部变量和堆内存对象等需要在作用域结束时自动释放的对象。

QSharedPointer
QSharedPointer是Qt中的共享智能指针，它可以在多个指针之间共享同一个对象，并且可以自动管理指针的生命周期。当最后一个指向对象的QSharedPointer被销毁时，QSharedPointer会自动释放对象的内存。QSharedPointer通常用于管理需要在多个地方使用同一个对象的情况。

QWeakPointer

QWeakPointer是Qt中的弱智能指针，它类似于QSharedPointer，但是不会增加对象的引用计数。QWeakPointer通常用于在多个指针之间共享同一个对象时，避免产生循环引用问题。

std::weak_ptr

std::weak_ptr是C++11标准库中的弱智能指针，它类似于QWeakPointer，但是不是Qt所提供的。std::weak_ptr可以用于在多个指针之间共享同一个对象时，避免产生循环引用问题。

QSharedDataPointer

QSharedDataPointer是Qt中的共享数据指针，它可以在多个对象之间共享同一个数据，而不需要共享整个对象。QSharedDataPointer通常用于实现copy-on-write（COW）技术，避免不必要的复制操作。需要注意的是，不同的指针类适用于不同的场景和需求，需要根据实际情况进行选择和使用。

159、Qt当中的show和exec区别?

在Qt中，show和exec都是用来显示对话框的方法，它们的主要区别在于它们的运行机制和返回值。

show方法是在当前线程中显示对话框并立即返回，该方法不会阻塞当前线程，因此当对话框显示后，程序会继续执行后续代码。如果在对话框显示期间需要执行一些其他操作，可以在对话框关闭事件中处理，或者使用Qt的事件循环机制（如QEventLoop）来阻塞程序执行。

exec方法是在当前线程中显示对话框，并且阻塞当前线程，直到用户关闭对话框为止。在执行exec方法后，程序会进入一个事件循环（QEventLoop），直到对话框关闭事件被触发。因此，如果需要在对话框关闭之前执行一些其他操作，需要在对话框关闭事件之前处理。

需要注意的是，show方法和exec方法都可以用来显示对话框，但是它们的使用场景不同。show方法通常用于显示模态对话框和非模态对话框，而exec方法通常用于显示模态对话框。如果需要在对话框显示期间执行一些其他操作，建议使用show方法，如果需要等待用户关闭对话框后再执行其他操作，建议使用exec方法。

160、 QString与基本数据类型如何转换?

在Qt中，QString与基本数据类型之间可以进行方便的相互转换。下面是常见的转换方法：

1. 将QString转换为基本数据类型 通过QString的各种转换函数，可以将QString转换为int、float、double等基本数据类型。例如：

```
QString str = "123";
int num = str.toInt(); // 将QString转换为int类型
float f = str.toFloat(); // 将QString转换为float类型
double d = str.toDouble(); // 将QString转换为double类型
```

2.将基本数据类型转换为QString 通过QString的静态函数，可以将int、float、double等基本数据类型转换为QString。例如：

```
1. int num = 123;
   QString str = QString::number(num); // 将int类型转换为QString
   float f = 3.14;
   QString str2 = QString::number(f); // 将float类型转换为QString
```

需要注意的是，QString和基本数据类型之间的转换可能会存在精度和数据溢出等问题，需要根据实际情况进行选择和处理。

161、 QMap类/QHash类/QVectoro类作用和区别?

在Qt中，QMap类、QHash类和QVector类都是用来管理数据的容器类，它们的作用和区别如下：

1. QMap类 QMap类是一种有序的映射容器，它可以将一个键值对映射到另一个键值对，且键值对是按照键的顺序进行排序的。QMap类通常用于对一组数据进行映射操作，常见的操作包括查找、插入、删除等。例如：

```
QMap<QString, int> map;
map.insert("Tom", 18);
map.insert("Jack", 20);
map.insert("Lucy", 17);
int age = map.value("Tom"); // 获取键为"Tom"的值
map.remove("Lucy"); // 删除键为"Lucy"的键值对
```

2.QHash类 QHash类是一种无序的哈希容器，它可以将一个键值对映射到另一个键值对，且键值对是按照哈希值进行存储的。QHash类通常用于对一组数据进行哈希操作，常见的操作包括查找、插入、删除等。例如：

```
QHash<QString, int> hash;
hash.insert("Tom", 18);
hash.insert("Jack", 20);
hash.insert("Lucy", 17);
int age = hash.value("Tom"); // 获取键为"Tom"的值
hash.remove("Lucy"); // 删除键为"Lucy"的键值对
```

3.QVector类 QVector类是一种动态数组容器，它可以存储任意类型的数据，并且支持动态增加和删除操作。QVector类通常用于存储一组数据，并且需要对数据进行动态操作，如排序、查找等。例如：

```
QVector<int> vec;
vec.append(1);
vec.append(2);
vec.append(3);
int num = vec.at(1); // 获取第2个元素
vec.removeLast(); // 删除最后一个元素
```

需要注意的是，QMap类、QHash类和QVector类应根据实际情况进行选择和使用。如果需要对数据进行排序和查找操作，建议使用QMap类；如果需要对数据进行哈希操作，建议使用QHash类；如果需要对数据进行动态操作，建议使用QVector类。

162、QList 类/QLinkedList类作用?

在Qt中，QList类和QLinkedList类都是用来管理数据的容器类，它们的作用和区别如下：

1. QList类 QList类是一种动态数组容器，它可以存储任意类型的数据，并且支持动态增加和删除操作。QList类的内部实现是一个可变大小的数组，可以动态调整数组的大小以适应数据的存储需求。QList类通常用于存储一组数据，并且需要对数据进行动态操作，如排序、查找等。例如：

```
QList<int> list;
list.append(1);
list.append(2);
list.append(3);
int num = list.at(1); // 获取第2个元素
list.removeLast(); // 删除最后一个元素
```

2.QLinkedList类 QLinkedList类是一种双向链表容器，它可以存储任意类型的数据，并且支持动态增加和删除操作。QLinkedList类的内部实现是一个双向链表，可以动态调整链表的大小以适应数据的存储需求。QLinkedList类通常用于存储一组数据，并且需要对数据进行动态操作，如排序、查找等。例如：

```
QLinkedList<int> list;
list.append(1);
list.append(2);
list.append(3);
int num = list.at(1); // 获取第2个元素
list.removeLast(); // 删除最后一个元素
```

需要注意的是，QList类和QLinkedList类应根据实际情况进行选择和使用。如果需要对数据进行频繁的插入和删除操作，建议使用QLinkedList类；如果需要对数据进行频繁的访问和遍历操作，建议使用QList类。

163、请说出Qt常用8类个控件?

Qt中常用的8个控件包括：

1. QLabel（标签控件）：用于显示文本或图像。
2. QPushButton（按钮控件）：用于接收用户的点击事件。
3. QLineEdit（单行编辑框控件）：用于接收用户输入的单行文本。
4. QTextEdit（多行编辑框控件）：用于接收用户输入的多行文本。
5. QComboBox（下拉框控件）：用于在预定义的选项中进行选择。
6. QCheckBox（复选框控件）：用于允许用户选择一个或多个选项。
7. QRadioButton（单选按钮控件）：用于允许用户从多个选项中选择一个。
8. QSpinBox/QDoubleSpinBox（数值输入框控件）：用于接收用户输入的数值。 这些控件是Qt中非常常用的控件，常常用于创建用户界面。需要根据实际需求进行选择和使用。

164、QLayout QStackedWidget类/QSplitter类/QDockWidget类?

在Qt中，QLayout类、QStackedWidget类、QSplitter类和QDockWidget类都是用于创建用户界面的类，它们的作用如下：

1. QLayout类 QLayout类是Qt中用于管理控件布局的类。它可以将控件按照一定的布局方式进行排列，如水平布局、垂直布局、网格布局等。QLayout类通常用于创建复杂的用户界面，可以有效地

管理控件的位置和大小。例如：

```
QHBoxLayout *layout = new QHBoxLayout;
layout->addWidget(button1);
layout->addWidget(button2);
layout->addWidget(button3);
setLayout(layout);
```

2.QStackedWidget类 QStackedWidget类是一种堆栈容器控件，它可以将多个子控件按照堆栈的方式进行管理，只显示当前的子控件，其它子控件被隐藏。QStackedWidget类通常用于创建多页应用程序，例如多个窗口之间的切换。例如：

```
QStackedWidget *stack = new QStackedWidget;
stack->addWidget(page1);
stack->addWidget(page2);
stack->addWidget(page3);
stack->setCurrentIndex(0);
```

3.QSplitter类 QSplitter类是一种分割窗口控件，它可以将窗口分割成多个部分，每个部分可以包含一个或多个子控件。QSplitter类通常用于创建可调整大小的用户界面，可以让用户自由调整窗口的大小和位置。例如：

```
QSplitter *splitter = new QSplitter(Qt::Horizontal);
splitter->addWidget(widget1);
splitter->addWidget(widget2);
splitter->addWidget(widget3);
```

4.QDockWidget类 QDockWidget类是一种停靠窗口控件，它可以将窗口停靠在主窗口的边缘或浮动在主窗口上方。QDockWidget类通常用于创建多文档界面（MDI）应用程序，可以让用户自由调整窗口的停靠位置和大小。例如：

```
QDockWidget *dock = new QDockWidget;
dock->setWidget(widget);
dock->setWindowTitle("Dock Window");
addDockWidget(Qt::LeftDockWidgetArea, dock);
```

需要注意的是，QLayout类、QStackedWidget类、QSplitter类和QDockWidget类都是用于创建用户界面的类，需要根据实际需求进行选择和使用。

165、Qt当中文件对话框、字体对话体、输入对话框、消息对话框应用实战?

以下是四个对话框在Qt中的应用实战：

1. 文件对话框（QFileDialog） QFileDialog是Qt中用于打开和保存文件的对话框类。它可以让用户选择文件的路径和名称，并且支持多种文件格式的过滤。以下是一个文件对话框的应用实例：

```
QString fileName = QFileDialog::getOpenFileName(this, "Open File",
QDir::homePath(), "Text Files (*.txt)");
if(!fileName.isEmpty()) {
    QFile file(fileName);
    if(file.open(QIODevice::ReadOnly)) {
        // 读取文件内容
        file.close();
    }
}
```

2.字体对话框（QFontDialog） QFontDialog是Qt中用于选择字体的对话框类。它可以让用户选择字体的名称、大小、颜色等属性。以下是一个字体对话框的应用实例：

```
bool ok;
QFont font = QFontDialog::getFont(&ok, QFont("Helvetica [Cronyx]", 10), this);
if(ok) {
    // 应用选择的字体
    setFont(font);
}
```

3.输入对话框（QInputDialog） QInputDialog是Qt中用于输入文本、数字、列表等的对话框类。它可以让用户输入各种类型的数据，并且支持自定义对话框标题、提示信息等。以下是一个输入对话框的应用实例：

```
QString text = QInputDialog::getText(this, "Input Dialog", "Enter your name:",
QLineEdit::Normal, "", &ok);
if(ok && !text.isEmpty()) {
    // 处理用户输入的文本
}
```

4.消息对话框（QMessageBox） QMessageBox是Qt中用于显示消息、提示、警告等的对话框类。它可以让用户选择不同的按钮选项，并且支持自定义对话框标题、提示信息等。以下是一个消息对话框的应用实例：

```
QMessageBox::StandardButton reply;
reply = QMessageBox::question(this, "Message Box", "Are you sure to quit?",
QMessageBox::Yes | QMessageBox::No);
if(reply == QMessageBox::Yes) {
    // 处理用户选择
}
```

需要注意的是，四个对话框的应用场景不同，需要根据实际需求进行选择和使用。

166、Qt绘制原理双缓冲机制?

Qt绘制原理中的双缓冲机制是指在绘制过程中使用两个缓冲区，一个用于绘制，一个用于显示，从而避免了绘制过程中的闪烁等问题。具体来说，双缓冲机制的实现过程如下：

1. 创建两个缓冲区，一个用于绘制，一个用于显示；
2. 在绘制过程中，将所有的绘制操作都先绘制到绘制缓冲区中；

3. 绘制完成后，将绘制缓冲区中的内容复制到显示缓冲区中；
4. 显示缓冲区中的内容会被显示在屏幕上。 这样，由于绘制操作是在绘制缓冲区中进行的，因此不会直接影响到显示缓冲区中的内容，从而避免了闪烁等问题。 在Qt中，双缓冲机制是由QPainter类和QWidget类共同实现的。QPainter类用于在绘制缓冲区中进行绘制操作，而QWidget类则在显示缓冲区中进行显示操作。具体来说，当QWidget类的paintEvent()函数被触发时，会创建一个QPainter对象，然后在其中调用绘制函数，将所有的绘制操作都绘制到绘制缓冲区中。绘制完成后，QPainter对象会自动将绘制缓冲区中的内容复制到显示缓冲区中，然后显示缓冲区中的内容会被显示在屏幕上。 需要注意的是，双缓冲机制虽然可以避免闪烁等问题，但是也会增加内存的消耗。因此，在实际应用中需要根据具体情况进行选择和使用。

167、Graphics View图形视图框架结构?

Graphics View是Qt中用于显示和处理大量图形元素的框架，其主要包括以下几个重要的类和组件：

1. QGraphicsItem和QGraphicsScene：QGraphicsItem类是Graphics View框架中所有图形项的基类，它定义了所有图形项的共同属性和行为。QGraphicsScene类是Graphics View框架中的场景类，它管理着所有的图形项，并且提供了对它们进行操作的接口。
2. QGraphicsView：QGraphicsView类是Graphics View框架中的视图类，它用于在屏幕上显示QGraphicsScene中的内容，并且提供了对场景进行缩放、平移、旋转等操作的接口。
3. QGraphicsWidget：QGraphicsWidget类是Graphics View框架中的窗口类，它可以作为一个图形项添加到QGraphicsScene中，并且支持鼠标事件、键盘事件等交互操作。
4. QGraphicsProxyWidget：QGraphicsProxyWidget类是Graphics View框架中的窗口代理类，它可以将一个QWidget对象转换为一个图形项，然后添加到QGraphicsScene中。
5. QGraphicsItemGroup：QGraphicsItemGroup类是Graphics View框架中的图形项组类，它可以将多个图形项组合成一个整体，从而方便对它们进行操作。
6. QGraphicsEffect：QGraphicsEffect类是Graphics View框架中的特效类，它可以对QGraphicsItem进行图形效果的处理，例如模糊、阴影、发光等。
7. QGraphicsPixmapItem和QGraphicsTextItem：QGraphicsPixmapItem类是Graphics View框架中的图片项类，它可以将一个QPixmap对象显示在场景中。QGraphicsTextItem类是Graphics View框架中的文本项类，它可以将一个字符串显示在场景中。 以上是Graphics View框架的主要组件和类，它们共同构成了Graphics View框架的结构。通过这些组件和类的使用，我们可以方便地实现各种复杂的图形界面和图形效果。

168、Qt当中如何读写文件?

在Qt中，可以使用QFile类和QTextStream类来读写文件。

1. 读文件：

```
QFile file("test.txt");
if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
    return;
QTextStream in(&file);
while (!in.atEnd()) {
    QString line = in.readLine();
    // 处理每一行数据
}
file.close();
```

上述代码中，首先使用QFile类打开要读取的文件，然后使用QTextStream类对文件进行读取。通过while循环，每次读取一行数据，然后进行处理。最后关闭文件。

2. 写文件：

```
QFile file("test.txt");
if (!file.open(QIODevice::WriteOnly | QIODevice::Text))
    return;
QTextStream out(&file);
out << "Hello world" << endl;
out << "Qt is awesome" << endl;
file.close();
```

上述代码中，首先使用QFile类打开要写入的文件，然后使用QTextStream类对文件进行写入。通过向QTextStream对象中写入数据，可以将数据写入文件中。最后关闭文件。 需要注意的是，在进行文件读写操作时，需要确保文件路径正确，并且文件权限足够。同时，需要根据具体情况使用不同的打开模式，例如只读模式、只写模式、追加模式等。

169、Qt中事件过滤处理方法?

Qt中的事件过滤器可以用于对某个对象的事件进行拦截和处理。事件过滤器是一个QObject对象，它可以安装到任何QObject派生类中，并且可以监听该对象的所有事件。当事件发生时，事件过滤器可以拦截并处理该事件，也可以将该事件转发给原始的事件接收者对象进行处理。 事件过滤器的处理方法如下：

1. 创建一个QObject派生类，实现其eventFilter()函数，该函数会在事件发生时被调用。

```
class MyEventFilter : public QObject
{
    Q_OBJECT
public:
    explicit MyEventFilter(QObject *parent = nullptr);
protected:
    bool eventFilter(QObject *watched, QEvent *event) override;
};
```

2.在需要监听的对象中，调用installEventFilter()函数安装事件过滤器。

```
MyEventFilter *eventFilter = new MyEventFilter;
QLabel *label = new QLabel("Hello, World!");
label->installEventFilter(eventFilter);
```

3.在eventFilter()函数中处理事件，可以通过判断事件类型和事件源来对事件进行不同的处理。

```
bool MyEventFilter::eventFilter(QObject *watched, QEvent *event)
{
    if (watched == label && event->type() == QEvent::MouseButtonPress) {
        QMouseEvent *mouseEvent = static_cast<QMouseEvent *>(event);
        // 处理鼠标事件
        return true;
    }
    return false;
}
```

需要注意的是，在eventFilter()函数中返回true表示该事件已经被处理，不再继续传递给事件接收者对象进行处理；返回false表示该事件仍然需要传递给事件接收者对象进行处理。

170、Qt 操作INI文件、JSON 文件、XML文件?

Qt提供了三种常用的文件格式处理方式，分别是INI文件、JSON文件和XML文件。下面分别介绍如何在Qt中操作这三种文件格式。

1. 操作INI文件 INI文件是一种常用的配置文件格式，在Qt中可以通过QSettings类来读写INI文件。以下是一个例子：

```
QSettings settings("myApp.ini", QSettings::IniFormat);
// 写入配置项
settings.setValue("General/Name", "Tom");
settings.setValue("General/Age", 20);
// 读取配置项
QString name = settings.value("General/Name").toString();
int age = settings.value("General/Age").toInt();
```

2.操作JSON文件 JSON是一种轻量级的数据交换格式，在Qt中可以通过QJsonDocument类和QJsonObject类来读写JSON文件。以下是一个例子：

```
// 读取JSON文件
QFile file("data.json");
if (file.open(QIODevice::ReadOnly)) {
    QJsonParseError error;
    QJsonDocument doc = QJsonDocument::fromJson(file.readAll(), &error);
    if (error.error == QJsonParseError::NoError && doc.isObject()) {
        QJsonObject obj = doc.object();
        QString name = obj.value("name").toString();
        int age = obj.value("age").toInt();
    }
    file.close();
}
// 写入JSON文件
QJsonObject obj;
obj.insert("name", "Tom");
obj.insert("age", 20);
QJsonDocument doc(obj);
QFile file("data.json");
```

```
if (file.open(QIODevice::WriteOnly)) {
    file.write(doc.toJson());
    file.close();
}
```

3.操作XML文件 XML是一种常用的文本格式，用于表示结构化的数据，在Qt中可以通过QXmlStreamReader类和QXmlStreamWriter类来读写XML文件。以下是一个例子：

```
// 读取XML文件
QFile file("data.xml");
if (file.open(QIODevice::ReadOnly)) {
    QXmlStreamReader reader(&file);
    while (!reader.atEnd()) {
        reader.readNext();
        if (reader.isStartElement() && reader.name() == "person") {
            QString name = reader.attributes().value("name").toString();
            int age = reader.attributes().value("age").toInt();
            // 处理读取到的数据
        }
    }
    file.close();
}
// 写入XML文件
QFile file("data.xml");
if (file.open(QIODevice::WriteOnly)) {
    QXmlStreamWriter writer(&file);
    writer.setAutoFormatting(true);
    writer.writeStartDocument();
    writer.writeStartElement("persons");
    writer.writeStartElement("person");
    writer.writeAttribute("name", "Tom");
    writer.writeAttribute("age", "20");
    writer.writeEndElement();
    writer.writeEndElement();
    writer.writeEndDocument();
    file.close();
}
```

以上是Qt中操作INI文件、JSON文件和XML文件的简单示例，通过这些方法，我们可以方便地读写和处理各种数据格式的文件。

171、HTTP 协议、WebSocket 协议?

HTTP协议和WebSocket协议都是常用的网络协议，下面分别介绍它们的特点和使用场景。

1. HTTP协议 HTTP（Hypertext Transfer Protocol）是一种应用层协议，用于在Web浏览器和Web服务器之间传输数据。HTTP协议基于TCP协议，使用请求-响应模式，客户端向服务器发送请求，服务器返回响应。HTTP协议的特点如下：

- 简单易用：HTTP协议的请求和响应格式简单明了，易于理解和实现。
- 无状态协议：HTTP协议是无状态协议，即服务器不会保存客户端的状态信息，每个请求都是独立的。

- 基于请求-响应模式：客户端向服务器发送请求，服务器返回响应，这种模式适合Web页面的请求和响应。 HTTP协议主要用于Web浏览器和Web服务器之间的数据传输，常用于浏览器请求Web页面、获取静态资源、提交表单等场景。

1. WebSocket协议 WebSocket是一种基于TCP协议的全双工通信协议，用于在Web浏览器和Web服务器之间实现双向通信。WebSocket协议支持在同一个TCP连接上进行数据传输，避免了HTTP协议的重复连接和断开操作，从而提高了通信效率。WebSocket协议的特点如下：

- 双向通信：WebSocket协议支持在同一个TCP连接上进行双向通信，客户端和服务器可以同时发送和接收数据。
- 实时性：WebSocket协议支持实时通信，数据传输的延迟和带宽占用都比HTTP协议更低。
- 跨域支持：WebSocket协议支持跨域通信，可以在不同域名和端口之间进行通信。 WebSocket协议主要用于需要实时通信和双向通信的场景，例如在线聊天、实时游戏、远程控制等。需要注意的是，WebSocket协议需要服务器端和客户端都支持该协议才能进行通信。

172、QtChart (图表、曲线图、饼状图、柱形、拆线图等)？

QtChart是Qt自带的图表库，用于绘制各种类型的图表，例如曲线图、饼状图、柱形图、折线图等。使用QtChart可以方便地将数据可视化，并支持用户交互和定制化设置。下面介绍QtChart中常用的几种图表类型及其使用方法。

1. 曲线图 曲线图用于展示一段时间内某个变量的变化趋势，例如温度、湿度、压力等。使用QtChart绘制曲线图的步骤如下：

- 创建QChart对象，并设置标题和坐标轴。
- 创建QLineSeries对象，设置曲线的名称和数据。
- 将QLineSeries对象添加到QChart对象中。
- 创建一个QChartView对象，并将QChart对象设置为其父对象。
- 将QChartView对象添加到布局中。 以下是一个简单的曲线图绘制示例：

```
QChart *chart = new QChart();
chart->setTitle("Temperature");
chart->legend()->hide();
chart->createDefaultAxes();
QLineSeries *series = new QLineSeries();
series->setName("Temperature");
series->append(0, 20);
series->append(1, 22);
series->append(2, 24);
series->append(3, 26);
chart->addSeries(series);
QChartView *chartView = new QChartView(chart);
chartView->setRenderHint(QPainter::Antialiasing);
ui->chartLayout->addWidget(chartView);
```

2.饼状图 饼状图用于展示不同数据之间的比例关系，例如不同国家的GDP占比、不同产品的销售占比等。使用QtChart绘制饼状图的步骤如下：

- 创建QChart对象，并设置标题。
- 创建QPieSeries对象，设置饼状图的名称和数据。

- 将QPieSeries对象添加到QChart对象中。
- 创建一个QChartView对象，并将QChart对象设置为其父对象。
- 将QChartView对象添加到布局中。 以下是一个简单的饼状图绘制示例：

```
QChart *chart = new QChart();
chart->setTitle("Sales");
QPieSeries *series = new QPieSeries();
series->setName("Product");
series->append("Product A", 25);
series->append("Product B", 35);
series->append("Product C", 40);
chart->addSeries(series);
chart->legend()->setAlignment(Qt::AlignRight);
QChartView *chartView = new QChartView(chart);
chartView->setRenderHint(QPainter::Antialiasing);
ui->chartLayout->addWidget(chartView);
```

3.柱形图 柱形图用于展示不同数据之间的数量关系，例如不同月份的销售额、不同省份的人口等。使用QtChart绘制柱形图的步骤如下：

- 创建QChart对象，并设置标题和坐标轴。
- 创建QBarSeries对象，设置柱形图的名称和数据。
- 将QBarSeries对象添加到QChart对象中。
- 创建一个QChartView对象，并将QChart对象设置为其父对象。
- 将QChartView对象添加到布局中。 以下是一个简单的柱形图绘制示例：

```
QChart *chart = new QChart();
chart->setTitle("Sales");
chart->setAnimationOptions(QChart::SeriesAnimations);
QBarSeries *series = new QBarSeries();
series->setName("Monthly Sales");
QBarSet *set1 = new QBarSet("January");
QBarSet *set2 = new QBarSet("February");
QBarSet *set3 = new QBarSet("March");
*set1 << 100 << 200 << 150;
*set2 << 150 << 100 << 250;
*set3 << 200 << 150 << 100;
series->append(set1);
series->append(set2);
series->append(set3);
chart->addSeries(series);
chart->createDefaultAxes();
chart->legend()->setVisible(true);
QChartView *chartView = new QChartView(chart);
chartView->setRenderHint(QPainter::Antialiasing);
ui->chartLayout->addWidget(chartView);
```

4.折线图 折线图用于展示不同数据之间的趋势变化，例如不同时间段内的股票价格、不同季度的GDP增长率等。使用QtChart绘制折线图的步骤与曲线图类似。 以上是QtChart中常用的几种图表类型的绘制示例，通过QtChart，我们可以方便地将数据可视化，让数据更加直观和易于理解。

173、Qt 中音频类和视频类分别是什么?

在Qt中，音频类和视频类分别是QAudio和QMediaPlayer。

1. QAudio QAudio类提供了音频的输入和输出功能，可以用于录音、播放音频等操作。使用QAudio需要以下步骤：

- 创建QAudioDeviceInfo对象，获取可用的音频设备信息。
- 创建QAudioFormat对象，设置音频格式。
- 创建QAudioInput或QAudioOutput对象，设置音频设备和音频格式。
- 开始录音或播放音频数据。 以下是一个简单的录音示例：

```
QAudioFormat format;
format.setSampleRate(44100);
format.setChannelCount(2);
format.setSampleSize(16);
format.setCodec("audio/pcm");
format.setByteOrder(QAudioFormat::LittleEndian);
format.setSampleType(QAudioFormat::SignedInt);
QAudioDeviceInfo info(QAudioDeviceInfo::defaultInputDevice());
if (!info.isFormatSupported(format)) {
    qWarning() << "Default format not supported, trying to use nearest format.";
    format = info.nearestFormat(format);
}
QAudioInput *audioInput = new QAudioInput(info, format);
QBuffer *buffer = new QBuffer(this);
buffer->open(QIODevice::ReadWrite);
audioInput->start(buffer);
```

174、QML鼠标与事件处理? QML布局? Loader 动态加载组件?

在 QML 中，可以通过鼠标与事件处理来响应用户的操作。常见的鼠标事件包括：

- 点击事件（MouseArea）
- 悬停事件（hover）
- 按下事件（pressed）
- 松开事件（released）
- 移动事件（MouseArea） 在 QML 中，可以使用 MouseArea 来处理鼠标事件。例如：

```
Rectangle {
    width: 100
    height: 100
    color: "blue"
    MouseArea {
        anchors.fill: parent
        onClicked: {
            console.log("Clicked!")
        }
    }
}
```

上面的代码创建了一个蓝色的矩形，并在其上添加了一个 MouseArea 来处理鼠标事件。当用户点击矩形时，会在控制台输出一条消息。

QML布局

在 QML 中，可以使用各种布局来管理控件的位置和大小。常见的布局包括：

- ColumnLayout
- RowLayout
- GridLayout
- StackLayout 以 ColumnLayout 为例：

```
ColumnLayout {
    spacing: 10
    Rectangle {
        width: 100
        height: 50
        color: "red"
    }
    Rectangle {
        width: 100
        height: 50
        color: "green"
    }
    Rectangle {
        width: 100
        height: 50
        color: "blue"
    }
}
```

上面的代码创建了一个 ColumnLayout，并在其中添加了三个矩形。这些矩形会依次排列在垂直方向上，并且它们之间的间距为 10。

Loader 动态加载组件

在 QML 中，可以使用 Loader 组件来动态加载其他组件。例如：

```
Loader {
    sourceComponent: Rectangle {
        width: 100
        height: 100
        color: "red"
    }
}
```

上面的代码创建了一个 Loader，并将其 sourceComponent 属性设置为一个红色的矩形。在运行时，Loader 会动态加载这个矩形，并显示在其内部。可以通过改变 sourceComponent 属性来动态加载不同的组件。

175、23 种设计模式应用场景?

设计模式是一种被广泛应用于软件设计中的经验总结和最佳实践。以下是常见的23种设计模式及其应用场景：

1. 单例模式：需要确保一个类只有一个实例时，例如配置类、数据库连接类等。
2. 工厂模式：需要根据不同的参数创建不同的对象时，例如不同类型的数据库连接、日志记录器等。
3. 抽象工厂模式：需要创建一组相关的对象时，例如不同类型的 UI 控件、不同类型的主题等。
4. 建造者模式：需要创建复杂对象时，例如汽车、电脑、房屋等。
5. 原型模式：需要创建大量相似对象时，例如游戏中的敌人、粒子等。
6. 适配器模式：需要将一个类的接口转换成另一个类的接口时，例如兼容不同版本的 API、使用第三方库等。
7. 桥接模式：需要将抽象部分和实现部分分离开来时，例如不同类型的图形界面控件、不同类型的数据存储方式等。
8. 组合模式：需要以树形结构组织对象时，例如目录结构、图形界面中的控件等。
9. 装饰器模式：需要动态地给对象添加额外的功能时，例如添加日志、缓存等。
10. 外观模式：需要简化复杂系统的接口时，例如封装底层库、封装复杂的业务逻辑等。
11. 享元模式：需要共享大量细粒度对象时，例如字符串池、对象池等。
12. 代理模式：需要控制对原始对象的访问时，例如权限控制、远程访问等。
13. 职责链模式：需要将请求发送给多个对象时，并动态确定哪个对象处理该请求时，例如日志记录器、异常处理器等。
14. 命令模式：需要将操作封装成对象，并支持撤销、重做等操作时，例如 GUI 应用中的操作历史记录、文本编辑器中的撤销、重做操作等。
15. 解释器模式：需要解释一种语言或表达式时，例如正则表达式、数学表达式等。
16. 迭代器模式：需要遍历一个对象集合时，例如集合类、文件系统等。
17. 中介者模式：需要将多个对象之间的通信进行解耦时，例如 GUI 应用中的组件之间的交互、多人游戏中的玩家之间的交互等。
18. 备忘录模式：需要保存和恢复对象的状态时，例如文本编辑器中的撤销、重做操作等。
19. 观察者模式：需要实现对象之间的消息通信时，例如事件驱动的 GUI 应用、发布订阅模式等。
20. 状态模式：需要根据对象的状态改变其行为时，例如游戏中的角色状态、多线程中的任务状态等。
21. 策略模式：需要在运行时根据不同的情况选择不同的算法时，例如排序算法、加密算法等。
22. 模板方法模式：需要定义一个算法的框架，并在子类中实现具体的步骤时，例如 GUI 应用中的生命周期、游戏中的角色行为等。
23. 访问者模式：需要对一组对象执行相同的操作时，例如编译器中的 AST、图形界面中的控件等。

176、Qt相机和视频处理技术?

Qt相机和视频处理技术是Qt提供的一组API和库，用于在Qt应用程序中访问摄像头和处理视频。以下是一些常用的Qt相机和视频处理技术：

1. QtMultimedia模块：提供了访问音频、视频、相机等多媒体设备的API。可以用来在Qt应用程序中访问相机，捕捉视频和音频流，以及播放音频和视频文件。
2. QtCamera模块：是QtMultimedia模块的一部分，提供了访问相机的API。可以用来获取相机的设备信息、预览相机的图像、捕捉相机的图像和视频等。
3. QtAV库：是一个基于Qt的多媒体框架，提供了高性能的视频和音频处理能力。可以用来播放各种格式的视频和音频文件，以及进行视频编解码和处理。
4. OpenCV库：是一个开源的计算机视觉库，提供了丰富的图像和视频处理算法。可以用来在Qt应用程序中实现图像和视频的处理，例如人脸识别、目标跟踪、图像滤波等。
5. FFmpeg库：是一个开源的音视频编解码库，支持多种音视频格式和编解码器。可以用来在Qt应用程序中实现音视频的录制、播放和转码等功能。 通过使用这些技术，开发者可以方便地在Qt应用程序中访问相机和处理视频，实现各种有趣的功能，例如人脸识别、视频监控、视频编辑等。

177、OpenCV人脸识别技术方法?

OpenCV是一个开源的计算机视觉库，提供了多种图像处理和分析的算法和工具。其中包括人脸识别技术，以下是一些常用的OpenCV人脸识别技术方法：

1. Haar特征分类器：Haar特征分类器是一种基于Haar小波变换的人脸检测算法。该算法通过对图像中的不同区域进行Haar小波变换，提取出不同的Haar特征，然后使用AdaBoost算法对这些特征进行分类，最终得到一个可以用于检测人脸的分类器。
2. LBP特征分类器：LBP特征分类器是一种基于局部二值模式（LBP）的人脸检测算法。该算法通过对图像中的不同区域进行LBP特征提取，然后使用级联分类器对这些特征进行分类，最终得到一个可以用于检测人脸的分类器。相对于Haar特征分类器，LBP特征分类器具有更快的检测速度和更高的检测精度。
3. 人脸识别：人脸识别是一种基于人脸特征提取和匹配的技术，用于识别和验证人脸身份。常用的人脸识别方法包括基于特征的方法、基于模型的方法、基于深度学习的方法等。其中，基于特征的方法通常使用PCA、LDA等技术对人脸图像进行特征提取和降维，然后使用SVM、KNN等算法进行分类和识别。
4. 人脸跟踪：人脸跟踪是一种用于跟踪人脸运动的技术，通常用于视频监控、人机交互等应用场景。常用的人脸跟踪方法包括基于模板匹配的方法、基于卡尔曼滤波的方法、基于粒子滤波的方法等。通过使用以上的OpenCV人脸识别技术方法，可以实现人脸检测、人脸识别和人脸跟踪等功能，应用于视频监控、人机交互等领域。

178、OpenCV实现图片美化原理机制?

OpenCV实现图片美化可以通过以下几个步骤实现：

1. 图像增强：使用直方图均衡化或CLAHE（对比度受限的自适应直方图均衡化）算法增强图像的对比度和亮度。
2. 锐化：使用高斯滤波器或双边滤波器对图像进行模糊处理，然后使用拉普拉斯滤波器或Sobel算子对图像进行锐化处理，以增强图像的细节和清晰度。
3. 去噪：使用中值滤波器或均值滤波器对图像进行去噪处理，以减少图像中的噪声和杂点。

4. 美白：使用颜色空间转换技术将图像从RGB空间转换到YCbCr空间，然后对亮度分量进行调整，以实现图像的美白效果。
5. 磨皮：使用双边滤波器或高斯双边滤波器对图像进行平滑处理，以减少皮肤细节的噪声和杂点，从而实现磨皮的效果。
6. 美颜：使用图像融合技术将磨皮和美白的效果与原图像进行融合，以实现图像的美颜效果。 上述步骤中，图像增强、锐化和去噪处理可以使用OpenCV中的滤波器和边缘检测算法实现；美白和磨皮处理可以使用OpenCV中的颜色空间转换和双边滤波器实现；美颜处理可以使用OpenCV中的图像融合技术实现。 总的来说，OpenCV实现图片美化的原理机制是通过对图像进行增强、锐化、去噪、美白、磨皮和美颜等多个处理步骤，从而实现对图像的优化和美化。

179、OpenCV多图合成技术原理?

OpenCV多图合成技术可以通过以下几个步骤实现：

1. 读取多张图像：使用OpenCV中的imread函数读取多张图像，存储为Mat对象。
2. 图像融合：使用OpenCV中的图像融合技术将多张图像进行融合，实现多图合成的效果。常用的图像融合方法包括加权平均融合、拉普拉斯金字塔融合、图像拼接等。
3. 图像调整：根据实际需求，对融合后的图像进行调整，如旋转、缩放、裁剪等处理。
4. 图像输出：使用OpenCV中的imwrite函数将处理后的图像输出为文件，以便后续使用。 在以上步骤中，图像融合是多图合成的核心步骤。常用的图像融合方法包括：
5. 加权平均融合：将多张图像按照一定的权重进行加权平均，得到一张融合后的图像。加权平均融合方法简单易懂，但需要手动设置权重，且容易出现过度或不足的情况。
6. 拉普拉斯金字塔融合：将多张图像依次进行高斯金字塔分解和拉普拉斯金字塔重建，然后按照一定的权重进行融合，得到一张融合后的图像。拉普拉斯金字塔融合方法可以自动调整权重，融合效果较好，但计算复杂度较高。
7. 图像拼接：将多张图像按照一定的拼接方式进行拼接，得到一张融合后的大图。常见的拼接方式包括水平拼接、垂直拼接、矩形拼接等。图像拼接方法可以实现较大的图像拼接，但容易出现拼接痕迹和形变。 总的来说，OpenCV多图合成技术原理是通过读取多张图像，使用图像融合技术实现多图合成，然后根据实际需求进行图像调整和输出。常用的图像融合方法包括加权平均融合、拉普拉斯金字塔融合和图像拼接等。

180、OpenCV 的视频中反投影图像技术原理

OpenCV的视频中反投影图像技术（Video-based Image Analysis and PerspectivE Rectification,简称VIPER）是指利用视频序列中的图像信息来对场景进行重建和跟踪的一种技术。其中，反投影图像技术是VIPER技术的核心之一，其原理如下：

1. 构建模板：首先，根据需要跟踪的目标，从视频序列中选择一帧图像，并手动选定目标区域，将其作为模板。
2. 计算直方图：对模板进行颜色直方图计算，得到目标颜色的分布情况。
3. 计算反投影图像：对视频序列中的每一帧图像，进行颜色直方图计算，并将其与模板的直方图进行比较，得到每个像素点的概率值，即反投影图像。反投影图像中，像素点的值越大，表明该像素点属于目标的概率越高。
4. 二值化处理：通过设置阈值，将反投影图像中的像素值大于阈值的像素点标记为目标点，其余像素点标记为背景点。
5. 目标追踪：根据目标点的位置信息，对目标进行跟踪。在后续的视频帧中，重复以上步骤，实现目标的实时追踪。 反投影图像技术利用了目标区域的颜色信息，通过计算直方图的方式将其转换为概率分布，在视频序列中的每一帧图像中进行比较，得到每个像素点的概率值，从而实现对目标的跟

踪。它在目标跟踪方面具有一定的优势，但也存在一些问题，如对光照变化和背景干扰比较敏感，需要进行后续的优化和改进。

181、数据库的常用范式有那些?

数据库的常用范式有以下几种：

1. 第一范式（1NF）：确保每个属性都是原子性的，即不可再分。满足1NF的关系称为平凡关系。
2. 第二范式（2NF）：在满足1NF的基础上，消除非主属性对主键的部分依赖关系。即每个非主属性都完全依赖于主键。
3. 第三范式（3NF）：在满足2NF的基础上，消除非主属性对主键的传递依赖关系。即非主属性不依赖于其他非主属性。
4. 巴斯-科德范式（BCNF）：在满足3NF的基础上，消除主属性对主键的部分依赖关系。即每个主属性都完全依赖于主键。
5. 第四范式（4NF）：在满足BCNF的基础上，消除多值依赖关系。即每个非主属性都不依赖于其他非主属性的多值集合。
6. 第五范式（5NF）：在满足4NF的基础上，消除关联依赖关系。即每个非主属性都不依赖于其他非主属性的多值集合的子集。
7. 高斯范式（GNF）：在满足5NF的基础上，消除无损分解的冗余依赖关系。

范式越高，数据冗余越少，数据更新异常越少，数据的一致性和完整性就越好。但是，高范式的设计也需要考虑查询操作的效率和数据存储的空间占用。因此，在实际应用中，需要根据需求和实际情况进行范式的选择和权衡。

182、MySQL 架构的Server层的执行过程?

MySQL的架构分为Server层和存储引擎层两部分，其中Server层负责处理客户端请求、管理连接、查询解析和优化、权限管理等任务，而存储引擎层则负责数据的存储和管理。下面是MySQL Server层的执行过程：

1. 连接管理：MySQL Server接收客户端的连接请求，根据配置文件中的参数设置和连接请求中携带的信息，对连接进行验证和管理，包括同时连接数、连接超时、最大连接数等。
2. 查询解析和优化：当客户端发送SQL语句请求时，MySQL Server会对SQL语句进行解析和优化，生成执行计划。解析阶段会将SQL语句转换为内部数据结构——解析树，优化阶段会根据解析树进行优化，包括索引选择、关联查询优化等。
3. 查询执行：生成执行计划后，MySQL Server将执行计划交给存储引擎层执行，存储引擎负责具体的数据访问和操作。MySQL Server会将执行结果缓存在内存中，以提高查询效率。
4. 权限管理：MySQL Server会根据用户的权限，对查询和操作进行授权和管理，包括用户的登录授权、资源访问授权等。
5. 错误处理和日志记录：MySQL Server会对执行过程中出现的错误进行处理和记录，并输出错误信息。同时，MySQL Server会将执行日志、慢查询日志、错误日志等信息记录下来，以便后续的优化和分析。 MySQL Server层的执行过程涉及到多个组件和模块，需要协同工作，才能完成客户端请求的处理和查询操作的执行。

183、常用存储引|擎? InnoDB 与MylSAM的区别?

常用的存储引擎包括InnoDB、MyISAM、Memory、Archive、CSV、Blackhole等。其中，InnoDB和MyISAM是MySQL中最常用的两种存储引擎，它们的主要区别如下：

1. 事务支持：InnoDB支持事务，而MyISAM不支持事务。
2. 锁机制：InnoDB采用行级锁，支持多版本并发控制（MVCC），可以实现更高的并发性和更好的读写性能；而MyISAM采用表级锁，只有在读取和更新表时才会对整个表加锁，因此并发性相对较低。
3. 索引结构：InnoDB的主键索引采用B+树结构，数据文件本身就是索引文件，因此查询效率较高；而MyISAM的索引采用B树结构，数据文件与索引文件分开存储，因此查询效率相对较低。
4. 空间占用：InnoDB占用的空间相对较大，因为它需要维护事务日志、回滚日志等；而MyISAM占用的空间相对较小，因为它不需要维护这些额外的信息。
5. 支持全文检索：MyISAM支持全文检索，而InnoDB不支持。 综上所述，InnoDB适合于要求数据完整性和事务支持的应用，如电子商务、金融等；而MyISAM适合于读写比较少、对性能要求较高的应用，如博客、新闻等。当然，选择存储引擎还要根据具体的应用场景和需求进行选择。

184、事务的ACID与实现原理?

事务是指一组逻辑操作，要么全部执行成功，要么全部执行失败，是保证数据完整性和一致性的重要机制。ACID是事务的四个基本特性，分别是原子性、一致性、隔离性和持久性。

1. 原子性：事务是一个原子操作，要么全部执行成功，要么全部执行失败，不存在部分执行的情况。
2. 一致性：事务执行前后，数据的完整性和一致性都得到保证，不会破坏数据的完整性和一致性。
3. 隔离性：事务之间是相互隔离的，一个事务的执行不会影响其他事务的执行。
4. 持久性：事务执行成功后，对数据的修改将被永久保存在数据库中，即使系统崩溃也不会丢失数据。 实现原理： 实现事务的ACID特性需要数据库管理系统提供一些机制和技术来支持，主要包括以下几个方面：
5. 原子性：实现原子性需要使用事务日志（Transaction Log），将所有的事务操作记录到日志中，保证在事务执行过程中，任何一个操作失败或者出现错误，可以通过回滚日志将数据库恢复到事务执行前的状态。
6. 一致性：实现一致性需要使用锁机制，将事务执行过程中涉及的数据进行锁定，防止其他事务对数据进行修改，保证数据的完整性和一致性。
7. 隔离性：实现隔离性需要使用并发控制机制，包括锁机制和多版本并发控制（MVCC）等，在保证数据一致性的前提下，提高并发性能。
8. 持久性：实现持久性需要使用事务日志和缓冲区管理等技术，将事务操作记录到日志中，并在提交事务后将数据写入磁盘，保证即使系统崩溃也不会丢失数据。 总之，事务的ACID特性是数据库管理系统的基本特性之一，它保证了数据的完整性和一致性，是保证数据质量和安全性的重要机制。

185、数据库中的锁机制?

数据库中的锁机制是为了保证多个事务之间的并发执行时，数据的一致性和完整性。常见的锁机制包括行级锁、表级锁、页级锁等，常用的锁有：共享锁（S锁）、排他锁（X锁）、意向锁（IS锁和IX锁）等。

1. 行级锁：是在行级别上对数据进行加锁，只有需要修改该行数据的事务才会加上排他锁，其他事务可以继续读取该行数据，但是不能修改。行级锁可以提高并发性能，降低锁冲突的概率，但是会增加数据库的开销和复杂性。
2. 表级锁：是在表级别上对数据进行加锁，当一个事务对表进行修改时，会对整个表加上排他锁，其他事务不能读取和修改该表的数据。表级锁简单、易于实现，但是会导致并发性能下降，锁冲突的

概率增加。

3. 页级锁：是在页级别上对数据进行加锁，当一个事务对某个数据页进行修改时，会对该数据页加上排他锁，其他事务不能访问该数据页的数据。页级锁可以降低锁冲突的概率，但是会增加数据库的开销和复杂性。
4. 共享锁（S锁）：允许多个事务同时读取同一份数据，但是不能修改数据。共享锁可以提高并发性能，但是会导致读取数据的事务需要等待排他锁释放。
5. 排他锁（X锁）：只允许一个事务对数据进行修改，其他事务不能读取和修改该数据。排他锁可以保证数据的一致性和完整性，但是会导致并发性能下降，因为其他事务需要等待排他锁释放。
6. 意向锁（IS锁和IX锁）：是一种辅助锁，用于协助数据库管理系统判断需要加什么级别的锁。IS锁用于表级别，表示一个事务需要在表中加共享锁；IX锁用于表级别，表示一个事务需要在表中加排他锁。 总之，锁机制是数据库管理系统实现事务隔离性的重要手段，不同的锁级别和锁类型可以提高并发性能，降低锁冲突的概率，但是也会增加数据库的开销和复杂性。

186、MySQL索弓|的实现原理?

MySQL的索引是为了加快查询速度而设计的，它通过将数据存储在特定的数据结构中，以便快速访问和检索数据。MySQL支持多种类型的索引，包括B树索引、哈希索引、全文索引等，其中最常用的是B树索引。 B树索引的实现原理如下：

1. B树是一种平衡二叉树，它的每个节点可以存储多个关键字和对应的指针，通常称为B树节点。
2. B树的节点分为根节点、叶子节点和中间节点，其中叶子节点存储了索引的值和对应的指针，中间节点存储了索引的值和指向子节点的指针。
3. B树节点的大小通常与磁盘页的大小相同，可以减少磁盘I/O的次数，提高访问速度。
4. B树的插入和删除操作需要遵循平衡二叉树的规则，当节点达到一定大小时，需要进行分裂或合并操作，以保持平衡。
5. B树的查找操作可以采用二分查找的方式，从根节点开始逐层查找，直到找到目标节点的位置。
6. B树索引支持范围查找和模糊查找，可以提高查询效率。 总之，B树索引是MySQL最常用的索引类型之一，它通过平衡二叉树的结构和节点的大小优化磁盘I/O，提高访问速度和查询效率。

187、SQL优化和索引优化、表结构优化?

SQL优化、索引优化和表结构优化是提高数据库性能的重要手段，它们可以减少数据库的开销和提高查询效率。下面分别介绍它们的优化方法：

1. SQL优化：

- 避免使用SELECT *，只查询需要的列；
- 避免使用子查询，可以使用JOIN语句代替；
- 避免使用OR，可以使用IN和UNION代替；
- 避免使用LIKE '%!v(MISSING)alue%!'(MISSING)，可以使用LIKE 'value%!'(MISSING)或全文索引代替；
- 避免使用ORDER BY RAND()，可以使用ORDER BY id DESC + LIMIT代替；
- 避免使用大量的UNION，可以使用UNION ALL代替；
- 避免使用GROUP BY，可以使用DISTINCT代替；
- 避免使用HAVING，可以使用WHERE代替。

2.索引优化：

- 选择合适的索引类型，如B树索引、哈希索引、全文索引等；

- 选择合适的索引列，如经常查询的列、联合查询的列、分组和排序的列等；
- 避免使用过多的索引，会增加写入数据的开销；
- 避免使用过长的索引，会增加磁盘I/O的次数；
- 避免使用NULL值，会降低索引的效率；
- 避免使用函数和表达式，会降低索引的效率。

3.表结构优化：

- 采用合适的数据类型，如INT、VARCHAR、DATETIME等；
- 避免使用大量的TEXT和BLOB类型，会增加磁盘I/O的次数；
- 避免使用NULL值，会增加存储空间和查询开销；
- 采用合适的表结构，如垂直拆分、水平拆分等；
- 避免使用过多的表关联，会增加查询开销；
- 避免使用过多的触发器和存储过程，会增加写入数据的开销。 总之，SQL优化、索引优化和表结构优化是提高数据库性能的重要手段，需要根据具体的情况进行优化，以达到最佳的查询效率和性能。

188、数据库参数优先?

数据库参数优化是提高数据库性能的重要手段之一，通过调整数据库的参数设置可以达到提高性能的效果。在优化数据库参数时，需要遵循以下的优先级：

1. 业务需求优先：数据库参数优化的目的是为了提高业务性能，因此需要优先考虑业务需求，包括数据库的读写比例、业务的瓶颈等。
2. 硬件资源优先：数据库参数优化需要结合硬件资源来进行，包括CPU、内存、磁盘等，需要根据硬件资源的实际情况来调整参数设置。
3. 数据库版本优先：不同版本的数据库在参数设置上可能存在差异，因此需要根据具体的数据库版本来进行参数优化。
4. 数据库引擎优先：不同的数据库引擎在参数设置上也存在差异，例如MySQL的InnoDB和MyISAM引擎，需要针对不同的引擎进行参数优化。
5. 优化建议优先：数据库厂商、第三方工具以及其他DBA等都会提供一些优化建议，需要根据实际情况进行参考和调整。 在进行数据库参数优化时，需要结合上述优先级进行综合考虑，以达到最佳的优化效果。此外，还需要进行测试和监测，以确保优化后的数据库性能达到预期的目标。

189、explain 的执行计划?

EXPLAIN是一个用于分析MySQL查询语句的关键字，可以帮助我们了解MySQL是如何执行查询语句的。执行EXPLAIN查询后，MySQL将返回一张包含查询执行计划的结果集，执行计划描述了MySQL是如何处理查询语句的、使用哪些索引以及使用了哪些优化器等信息。 执行EXPLAIN查询时，需要在查询语句前加上EXPLAIN关键字，例如：

```
EXPLAIN SELECT * FROM table WHERE id = 1;
```

执行计划的结果集包含以下列：

1. id：查询中每个SELECT子句和操作表的唯一标识符，可以用来标识哪些操作是相关的。
2. select_type：查询的类型，包括简单查询、联合查询、子查询等。
3. table：操作的表名。
4. partitions：匹配到的分区。

5. type：访问类型，包括全表扫描、索引扫描、范围扫描等。
6. possible_keys：可能使用的索引。
7. key：实际使用的索引。
8. key_len：使用的索引长度。
9. ref：连接匹配条件的列。
10. rows：MySQL估计需要扫描的行数。
11. filtered：过滤行的百分比。
12. Extra：包含MySQL解决查询的详细信息，如使用的优化器、是否使用了临时表等。 通过分析执行计划，可以了解到MySQL是如何处理查询语句的、使用哪些索引以及使用了哪些优化器等信息，从而可以根据执行计划进行SQL性能优化。

190、MySQL的主从复制?

MySQL的主从复制是一种数据复制技术，可以将一个MySQL主服务器上的数据以异步方式复制到一个或多个MySQL从服务器上，从服务器上的数据与主服务器上的数据保持一致。主从复制可以帮助我们实现数据备份、读写分离、负载均衡等功能。 主从复制的基本原理如下：

1. 主服务器将写操作记录到二进制日志（binary log）中，包括对表的增删改操作。
2. 从服务器连接到主服务器，请求复制二进制日志中的数据。
3. 主服务器将二进制日志中的数据发送给从服务器，从服务器接收并应用这些数据。
4. 从服务器定期轮询主服务器，查找更新的二进制日志数据，并将这些数据应用到从服务器上。 在MySQL的主从复制中，主服务器是唯一的写入节点，负责更新数据，而从服务器是只读的，只负责复制数据。主服务器和从服务器之间的复制可以是异步的，因此主服务器上的数据更新可能不会立即反映到从服务器上，但一般情况下这种延迟是可控的，并不会影响系统的稳定性和安全性。 需要注意的是，在主从复制中，从服务器只是主服务器数据的复制品，不能对其进行写入操作，否则可能会导致数据不一致。如果需要对数据进行写入操作，则需要在主服务器上进行操作。

191、读写分离?

读写分离是一种数据库优化方案，通过将读操作和写操作分离到不同的服务器上，从而达到提高数据库性能和可用性的目的。读写分离的基本原理是：将写入操作集中在主服务器上，而将读操作分散到多个从服务器上，从而降低主服务器的读负载，提高数据库的并发处理能力。 读写分离的主要实现方式是：在应用程序中使用多个数据库连接，其中一个连接用于写操作，而其他连接则用于读操作。读写分离的优点包括：

1. 提高性能：通过将读操作分散到多个从服务器上，可以降低主服务器的读负载，提高数据库的并发处理能力。
2. 提高可用性：在主服务器出现故障时，可以通过从服务器提供读服务来保证系统的可用性。
3. 提高扩展性：通过增加从服务器的数量，可以提高系统的扩展性，满足不断增长的访问需求。 读写分离的缺点包括：
4. 数据不一致：由于主服务器和从服务器之间的数据复制是异步的，因此在数据复制过程中可能会出现数据不一致的情况。
5. 配置复杂：读写分离需要配置多个服务器，管理和维护比较复杂。
6. 代码改动：应用程序需要对读写操作进行区分，代码改动比较大。 需要注意的是，在读写分离中，写操作只能在主服务器上执行，而读操作可以在主服务器和从服务器上执行。如果需要保证数据的一致性，可以通过采用主从复制的方式来实现。

192、 分库分表(垂直分表、垂直分库、水平分表、水平分库)？

分库分表是一种常见的数据库水平和垂直扩展方案，用于解决单一数据库性能瓶颈问题。分库分表的实现方式有垂直分表、垂直分库、水平分表和水平分库四种方式。

1. 垂直分表：将一张表按照列拆分成多个表，每个表只存储部分列的数据，不同的表可以存储在不同的数据库中。垂直分表的优点是可以将不同的关注点拆分到不同的表中，减少表的冗余和重复，提高查询效率。缺点是如果需要查询多个表的数据则需要进行关联查询，增加了查询的复杂度。
2. 垂直分库：将不同的业务数据存储在不同的数据库中，每个数据库只处理特定的业务逻辑。垂直分库的优点是可以提高数据库读写的并发性，减少单库数据量，降低单点故障的风险。缺点是增加了系统的复杂性，增加了数据一致性的难度。
3. 水平分表：将一张表按照行进行拆分，将不同的行数据存储在不同的表中，每个表可以存储在不同的数据库中。水平分表的优点是可以将数据分散到多个表中，降低单表数据量，提高查询效率和写入性能。缺点是需要进行数据的路由和管理，增加了系统的复杂性。
4. 水平分库：将一张表按照主键的哈希值进行拆分，将不同的数据存储在不同的数据库中。水平分库的优点是可以将数据分散到多个库中，提高系统的并发能力和可用性。缺点是增加了系统的复杂性，增加了数据一致性的难度。 需要根据实际情况选择合适的分库分表方案，综合考虑性能、可用性、数据一致性和系统复杂度等因素。

193、分区?

数据库分区是一种常见的数据库水平扩展方案，用于解决单一数据库性能瓶颈问题。数据库分区将一张表的数据划分到多个物理存储位置上，每个存储位置可以是不同的磁盘、服务器或者数据中心。数据库分区可以按照不同的数据分区策略进行划分，包括：

1. 范围分区：按照数据的范围进行分区，例如按照时间、地区、部门等进行分区，可以将相似的数据划分到同一个分区中，提高查询效率。
2. 哈希分区：按照数据的哈希值进行分区，可以将数据均匀地分散到多个分区中，避免单个分区数据量过大，提高写入性能和查询效率。
3. 列表分区：按照数据列表进行分区，可以将相似的数据划分到同一个分区中，提高查询效率。
4. 复合分区：按照多个分区策略进行组合，可以更加灵活地进行数据分区。 数据库分区的优点是可以提高数据库的性能、可用性和扩展性，降低单个数据库的压力和风险。缺点是增加了系统的复杂度和运维成本，需要考虑数据一致性、路由和管理等问题。 需要根据实际情况选择合适的数据库分区策略，综合考虑性能、可用性、数据一致性和系统复杂度等因素。

194、 主键一般用自增ID还是UUID?

主键可以使用自增ID或UUID，具体使用哪种方式需要根据实际情况进行选择。 使用自增ID的优点是：

1. 查询效率高：自增ID是单调递增的整数，可以通过B+树等数据结构快速定位到指定的记录。
2. 存储空间小：自增ID通常使用整数类型存储，存储空间较小。
3. 易于管理：自增ID可以自动分配，不需要手动指定主键值，避免因主键冲突导致的错误。 使用UUID的优点是：
4. 全局唯一：UUID是128位的二进制数，可以保证全局唯一性，避免因主键冲突导致的错误。
5. 分布式：UUID可以在分布式系统中使用，避免因主键重复导致的数据同步问题。

6. 不可预测：UUID是随机生成的，不容易被猜测，可以增加数据安全性。 需要根据实际情况选择合适的主键方式，综合考虑查询效率、存储空间、数据安全性和分布式支持等因素。如果主键不需要全局唯一性，且查询效率是优先考虑的因素，可以考虑使用自增ID；如果主键需要全局唯一性，或者需要在分布式系统中使用，可以考虑使用UUID。

195、视图View?

视图（View）是一种虚拟的表，它本身并不存储任何数据，而是由查询语句定义的结果集。视图可以像表一样被查询，但查询的结果实际上是从基本表中检索出来的数据。视图可以将多个表的数据组合在一起，形成一个逻辑上的表，简化了复杂查询的编写和维护。 视图的优点包括：

1. 简化查询：视图可以将多个表的数据组合在一起，形成一个逻辑上的表，简化了复杂查询的编写和维护。
2. 数据安全：视图可以对表进行过滤，只暴露需要访问的数据，增强了数据的安全性和隐私性。
3. 数据独立：视图可以隐藏底层表的结构和数据，使得应用程序和查询不受基础表结构的变化影响。
4. 性能优化：视图可以对常用的复杂查询进行优化和缓存，提高查询性能。 视图的缺点包括：
5. 更新限制：视图只是一个虚拟的表，不能直接进行更新操作，需要通过更新底层表来实现。
6. 查询性能：由于视图需要动态生成结果集，查询性能可能会受到影响。
7. 存储空间：视图本身不存储任何数据，但需要占用一定的存储空间来存储定义和元数据。 需要根据实际情况进行视图的设计和使用，综合考虑数据安全、数据独立、查询性能和更新限制等因素。视图是一个非常常用的数据库对象，可以提高查询和数据管理的效率。

196、存储过程procedure?

存储过程（Procedure）是一种预编译的数据库对象，包含一组SQL语句和程序逻辑，可以像函数一样被调用。存储过程通常被用来完成特定的任务或操作，例如数据校验、数据转换、复杂计算等。存储过程可以接受参数和返回值，也可以包含流程控制语句和异常处理机制，具有很强的灵活性和可重用性。存储过程的优点包括：

1. 提高性能：存储过程可以预编译和缓存，可以减少SQL语句的解析和编译时间，提高查询和操作的性能。
2. 提高安全性：存储过程可以对数据进行访问控制和安全检查，增强了数据的安全性和完整性。
3. 提高可维护性：存储过程可以封装复杂的业务逻辑和数据访问，简化了应用程序的开发和维护。
4. 增强可扩展性：存储过程可以被多个应用程序复用，可以减少重复开发和代码冗余，增强了系统的可扩展性。 存储过程的缺点包括：
5. 存储过程的调试和测试相对困难。
6. 存储过程的部署和维护需要DBA等专业人员进行管理。
7. 存储过程的更新和修改需要谨慎处理，可能会影响到其他应用程序。 需要根据实际情况进行存储过程的设计和使用，综合考虑性能、安全性、可维护性和可扩展性等因素。存储过程是一个非常常用的数据库对象，可以提高应用程序的性能和可维护性。

197、触发器Trigger?

触发器（Trigger）是一种特殊的存储过程，它与数据库表有关联关系，当表中的数据发生变化时，触发器会自动地执行一些定义好的操作。触发器通常被用来实现数据约束、日志记录、数据同步等功能。触发器可以绑定在表的插入、更新、删除操作上，也可以在事务提交前或后执行。 触发器的优点包括：

1. 数据约束：触发器可以实现数据的约束和完整性检查，确保数据的正确性和一致性。
2. 日志记录：触发器可以实现对数据的变更进行日志记录，便于数据的审计和追踪。
3. 数据同步：触发器可以实现数据的同步和复制，确保数据的一致性和可靠性。
4. 灵活性：触发器可以根据业务需求自定义操作和逻辑，具有很强的灵活性。 触发器的缺点包括：
5. 性能影响：触发器的执行会增加数据库的负载和延迟，可能会影响系统的性能。
6. 调试和维护困难：触发器的调试和维护相对困难，需要特定的工具和技能。
7. 复杂性：触发器的逻辑和操作比较复杂，需要仔细考虑和测试。 需要根据实际情况进行触发器的设计和使用，综合考虑性能、数据约束、日志记录和数据同步等因素。触发器是一个非常强大的数据库对象，可以提高数据的安全性和完整性，并实现复杂的业务逻辑和数据操作。