对于所有的设计模式，无论思想是什么，都遵守三个基本原则：

单一职责原则，开放封闭原则，依赖倒转原则

对于这三个原则，与其说是设计模式所遵守的原则，不如规定为我们程序设计中应该遵守的原则。在程序设计中，即使我们对设计模式不是很清楚，但是如果我们设计过程中遵守这几个基本原则，那么设计出来的程序还是会很优雅的

首先来看到**单一职责原则**

这个原则在应用场景中其实就是**面向对象**。这个原则注意的本质上就是在类的设计中让其负责的功能尽可能的专一，使得我们在设计过程中可以更加专注于特定功能的实现，能够减轻我们的设计压力和维护压力。

举个简单的例子吧，就比如不久前做的那个堆排序作业。其实现在看来heap类的设计是存在大问题的。这个类负责的功能太多了，它既负责了对于数据的对排序处理，又负责了对于堆排序的渲染。这就导致了我后期在debug时想要去找到对应的错误位置需要去找到整个类中存在的错误。而此时类简单看实现代码行数已经达到了小几百行。这是要去debug是极其不易的。确实有亲身体会。

因此，这个单一职责原则的目的就在于各个功能模块间的实现分离。通过这种分离，我们能够实现更加高效的设计和debug。就比如，当你数据构建层面出错就去找对应数据构建的类。当你渲染出现问题就去找对应渲染的类。同时类功能的分离也能提高代码的复用性。

就比如那个heap类，由于我们将渲染模块和数据模块都耦合到了一起，如果我们想要在这个项目中再去添加与现在的排序类相似的类，我们是无法复用现在已有类的渲染模块的。直接继承将是一个糟糕的注意。而且我们已有的渲染模块是基于heap类去设计的，即使我们想要cv，我们也得考虑新类的设计，这是代码低复用性的体现。相反，我们应该考虑的是一个隐藏实现细节的渲染类来进行各种排序的实现。让heap等排序类专注于数据层面的构建，通过设计出render等渲染类来实现对应的渲染动画。

因此，在我们设计类的时候，我们要尽可能的让类的功能尽可能的单一，通过不同类之间的配合再来实现复杂的功能而不是一股脑塞在一个类中。而且这在类的定义中也可以看出类的设计初衷就是让我们设计一个功能单一的类。类是一组相关属性和行为的集合。注意这里是一组而不是多组。我们在任何时候都不应该有万能的这个概念，毕竟这个概念是不符合我们的设计逻辑的。尽管用户层是希望一个万能，但是用户层的万能与底层的万能不能等效。

接下来看第二个原则 **开放封闭原则**

其实这个原则就是**单一职责原则**的延续。简单来说，如果你的一个类很好的遵守了单一职责原则，换句话说这个类在一个功能上做到了极致，那么无论你添加什么这个类的功能都不会有什么改变。同时，你会发现如果一个类很好的遵守了单一职责原则，那么当出现新的需求时，仅仅依靠现在的类是无法解决的，那么现在就需要进行扩展，那么此时我们就有俩种方式来进行扩展。一种是直接修改我们现在的类去添加对应的功能，但是正如我们前面所说，这样其实是未被单一职责原则的。那么，内求失败后，我们需要外求。

既然外求，由于我们还是需要遵守单一职责原则，那么我们应该考虑实现一个新的类来实现我们新增的功能，而且一般来看这个类是需要和已有的一些类进行联动的。但是这个设计过程并不应该影响到已经设计好的类。在这个新增的类设计完成后，我们应该考虑把这个类和原先的类打包到另外一个大类中去实现。这种设计下，这个大类就对原有的功能进行了扩展，而且俩个小子类又都遵守了单一职责原则。

好，回来看看我们哪里体现了开放封闭原则。首先，我们实现了对已有功能的扩展，这是一种对于原有设计的开放性质的阐述。其次，我们并没有对原有的类进行修改，已完成的类是相对来说封闭的，这是封闭的性质。

这个封闭属性是相当重要的。毕竟如果一个类保留了它的封闭属性，那么它一般是通过了一定的测试，能够稳定运行的，如果我们去对这个类进行添加修改等破坏了它的封闭性，那么对于原有的哪些个用到封闭的类的api地方，又有可能导致一些奇奇怪怪的错误产生。而导致必须对整个类全部进行重新审核和测试。

举个例子吧，在现实中，我们通常可以看到一些男人在外部工作，不理家中事，但是家里还是被老婆打理的不错。这就是一个典型的开放封闭原则的示例。别杠，再杠让你飞起来。家庭是一个大类，这个大类中包含了一个负责家庭资金供给的父亲，一个负责家庭打理的母亲，还有其他一些杂七杂八的东西。种种这些相互独立，但是又组成一个整体。我们可以看做每个对象之间相互独立，但是多个对象构成了一个大类达到了一种和谐。

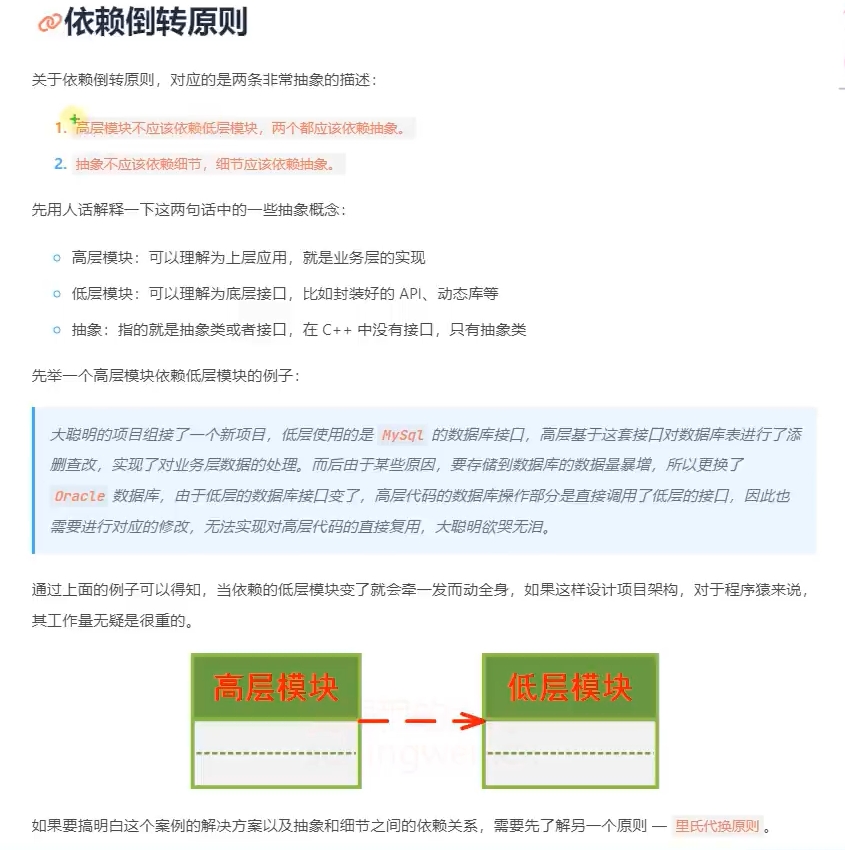
这个其实在QT中可以窥见。在QT中，其实已有的库可以视作一系列的具有封闭属性的库，我们在设计中要做的一般也是设计这些具有封闭性的类。然后，我们会在一些特别的类中，就比如，一个项目初始的窗口类中去进行这个小类的实例化与调用。这也是一种开放封闭原则的设计实例。

接着再来看一些需要注意的地方。就是即使我们实现了一些新的类，我们怎么去实现这些新的类与已有的类之间的联动。这里就需要我们在设计类是预留一些接口。这个接口一般来说是可以预见到的，就比如，商场的购物通常会有”打折”这种事情发生，可能是几折几折，也可能是买一送一等等，但是这些都有一个共性，就是他们都是”打折”这件事。因此，我们在设计商场购物的对应类中，我们应该预留一个专门的接口用于打折以实现以后可能存在的扩展。这个其实身边就能看到，就比如我们的笔记本，笔记本保留了相当多的插口供我们使用，而且一个插口可以使用不同的设备，只要这个设备的插口符合我们笔记本的接口规范。

这种接口的设计一般是通过**多态**的特性来进行实现的。在父类中去预留一个虚函数接口，当我们想要一个具体的打折方案后，我们去继承这个父类并对这个方法进行对应的重写，通过使用这个重写类去对父类进行替换，来实现我们打折方案的改变。而且由于继承这种关系，所以这些个父子类间的差异就在与这个重写的接口类处，不用太去担心冲突问题。

一般来说，这个接口是一个虚类的父类指针以提供该虚类的系列子类的指向。

这种**开放封闭原则**能够保证程序的可扩展性，可维护性，可复用性。



接下来看第三个原则，**依赖倒转原则**，话不多说，直接上图

**高层模块不应该依赖低层模块，二者都应该依赖于抽象；**

**抽象不应该依赖细节，细节应该依赖于抽象**

再解释下高层模块和低层模块。高层模块就是编写的应用程序，也就是应用层，这个你应该可以理解。接下来注意一点，我们要分析的是底层模块，而不是底层模块，所有位于高层模块下的模块都是低层模块。抽象基本就等价于C++中的抽象类以及各类的api。

举例下这里的直接依赖，就比如一个外卖系统中，通过使用一个数据库系统来实现数据的存储与使用，如果直接使用一个具体数据库的引用，那么这个就是直接依赖。而我们**高层模块不应该依赖低层模块，二者都应该依赖于抽象**的目的就是避免这种直接的绑定。你可以想象，你游戏账户直接绑定一个手机号，那么当你绑定的这个手机号出现了问题，你这个游戏账号将会导致一系列的问题，就比如我们无法去登录等。

而为了避免这种情况，我们的想法是通过一个中间层来进行缓冲，而这个的作用就是实现高层模块和底层模块的隔离，避免这俩者之间的直接调用。而且对于这个中间类(抽象类)需要有几个规定。最简单的一点就是这个函数必须在高层和底层的代码发生改动时不必要去修改另外一边的代码，起到所谓的隔离作用。通过这种隔绝，我们能够使得修改代码时只用去关心对应代码区域的修改而不用去关注全局代码的更改。

在使用时，我们一般是需要高层模块来使用这个抽象的api的，在c++中，这个就是所谓的虚函数。但是吧，一般来说，我们的这个抽象的父类是不会提供具体的实现也是不应该提供具体的实现的。需要在对应的子类中去进行具体的不同功能的划分与实现。

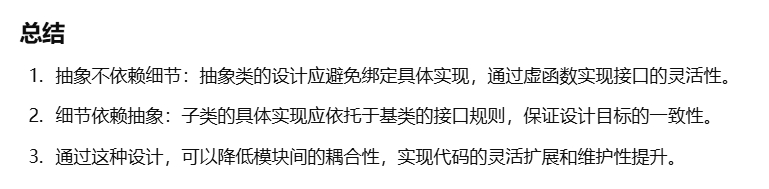
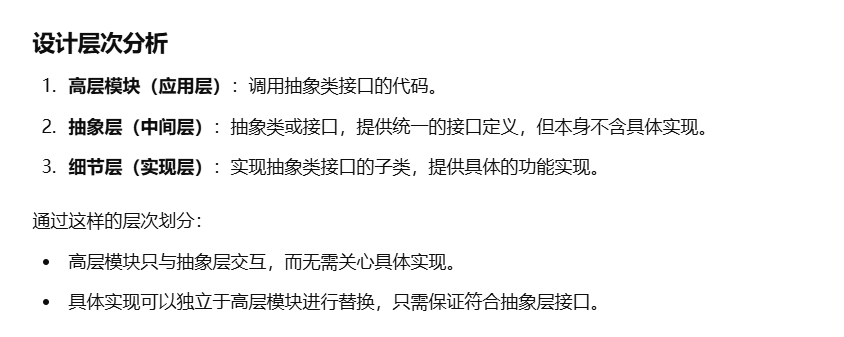
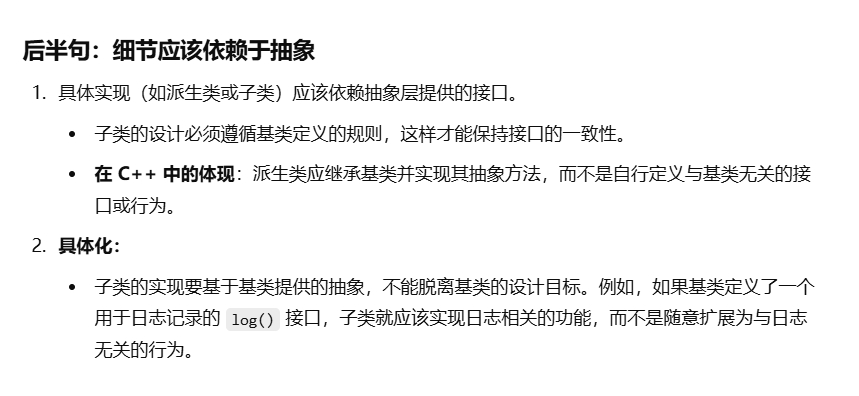
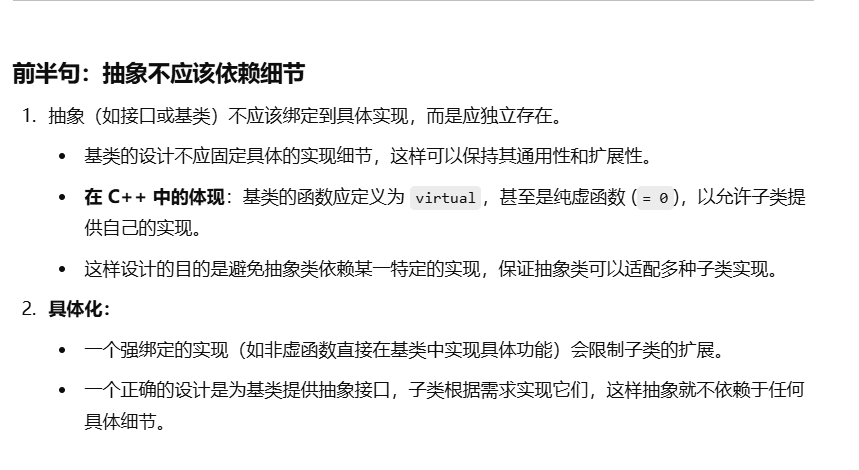
这个就比如吧，我们在进行一个项目的设计中，前期由于一些设计方面的问题，直接使用了mySQL数据库作为使用的数据库。但是在之后的发展中，由于数据的堆积，这个数据库无法满足我们的需求了，我们需要去替换一个更大的数据库，此时如果我们是直接调用的mySQL的数据库的话，那么我们在替换这个数据库的时候，我们需要去把全部用到mySQL的地方都进行更改，这个工作量是巨大且不可接受的，因为这个会导致前面已经通过测试的代码需要全部重新审批。但是如果我们引入了一个中间层来进行设计，提供一个统一的一个接口，我们只需要去注意我们使用接口的对象的替换就行了。如果我们设计的足够合理，我们就只需要去替换这里就行了，这个设计架构就相对于前面的全局替换高效了不止一星半点。

**抽象不应该依赖细节，细节应该依赖于抽象**

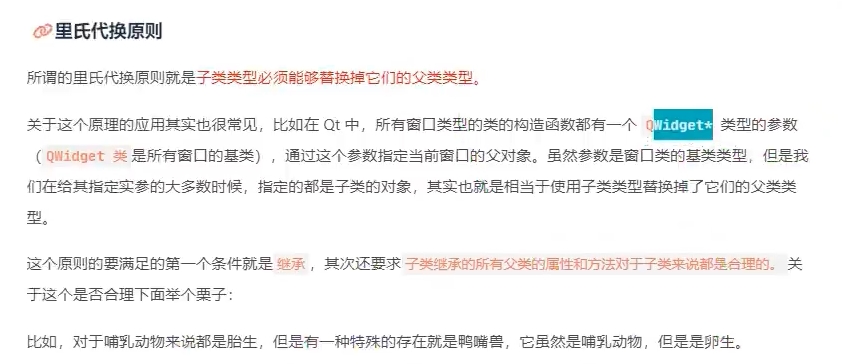
重新来分半简述下这句话

抽象不应该依赖细节：抽象(如接口或基类)不应该直接绑定到具体实现上，而是应该独立存在。简单来说，我们的一个基类函数，不应该导致我们这个抽象的接口被固定了，我们不应该给其提供实现，或者至少说不应该限定它只能有一个实现。简单以实现来看的话，我们不应该限定我们的基类函数为不可改变的。至少是一个virtual的，至于要不要是纯虚函数取决于具体的环境。这种设计下，抽象就不会被绑定到一个具体的实现上，至少来说可以是在子类中重载的。

接下来看后半句话，**细节应该依赖于抽象**。具体的实现(如派生类或子类)应该遵循抽象层定义的规则(如接口或虚函数)。是说，这里的子类的具体实现不应该脱离我们使用的抽象层来实现。还是回到c++中，就是说我们的子类的实现不能够脱离我们基类的根本目的。只有按照基类设计的目的去设计对应的子类，我们才能保证接口能够正确的被使用。

在这种设计中，我们可以进行一点设计层次上的分析。这里调用一个类的类就是一个应用层类似的。一个抽象类的基类其实就相当于一个中间层，提供了一系列的接口，但是这些接口通常是没有意义的，或者说应该是没有意义的。具体的实现应该在这个抽象类的子类中进行实现。通过这样，我们就实现了一种隔离，通过这种隔离我们能够实现本来高耦合的降低。

也就是说，在这种设计要求下，我们如果选择在子类中去进行一些基类中没有的功能的扩展，我们需要在基类中预留一些接口函数去等待这些接口函数的插入。通过这种实现我们能够实现避免大量修改原有的代码。



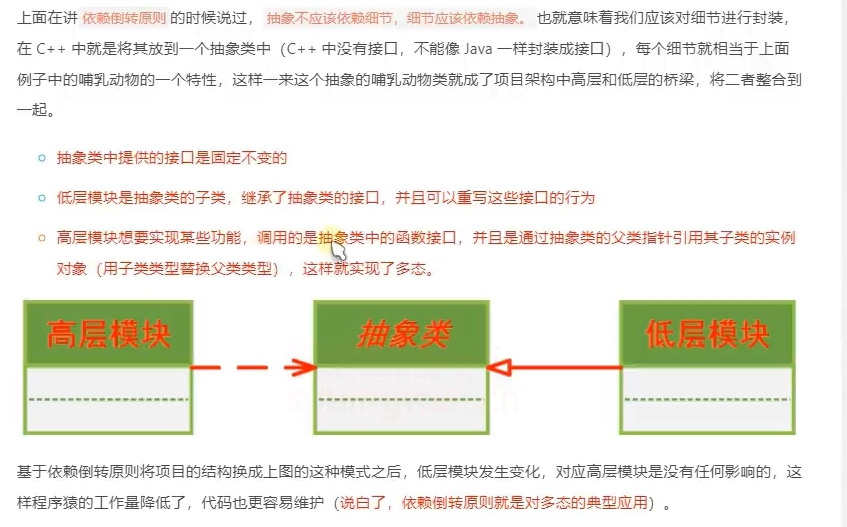
接下来看一个重要的原则，**里氏代换原则**。

这个原则其实很好理解，它本身要求的是子类类型必须能够替换他们的父类类型。我们也应该知道这个规则在QT中时相当常见的。就比如一个窗口类函数参数，我们可以使用对应窗口类的一系列父类和子类，只要他们是QWidget类或者其派生类的就行。而这种课替换性就是里氏代换原则所要求的。

要实现这种原则，就必须要求多态这个面向对象特性。父类指针可以指向子类对象。

对于这种原则，需要简单了解的就是子类必须实现父类中全部要求的接口函数。

额外需要注意我们这里提到的里氏代换原则的目的，只有拥有这个原则的设计，我们才有可能保证依赖倒转原则。



简单来看的话，这个原则要求我们在设计高层模块是我们只需要=也只应该去调用虚类中的父类函数，而不要求我们去了解我们的子类的附加函数，我们应该通过父类的对应接口就能够调用所有的子类方法。