设计原则

1、类之间的关系

1）依赖关系：对于两个相对独立的对象，当一个对象负责构造另一个对象的实例，或者依赖另一个对象的服务时，这两个对象之间主要体现为依赖关系；
在代码层面：一个类作为参数被另一个类在某个方法中使用，例如：局部变量，方法中的参数，对静态方法的调用等；
UML 表示方法：虚线箭头。

```
class B {...}
class A {...}
A::Function1(B &b)
//或 A::Function1(B *b)  或 A::Function1(B b)
//或 B* A::Function1()  或 B& A::Function1()
//或 int A::Function1() { B* pb = new B; /* ... */ ;delete pb; }
```

2）关联关系：对于两个相对独立的对象，当一个对象的实例与另一个对象的一些特定实例存在固定的对应关系时，这两个对象之间为关联关系。这种关系比依赖更强、不存在依赖关系的偶然性、关系也不是临时性的，一般是长期性的，而且双方的关系一般是平等的；
在代码层面：一个类拥有另一个类的指针或引用作为成员变量；
UML 表示方法：实线箭头，如果使用双箭头或不使用箭头表示双向关联。

```
class B {...}
class A { B* b; ... }
```

3）集合/聚集关系：聚合体现的是整体与部分拥有的关系，即 has-a 的关系，此时整体与部分之间是可分离的，可以具有各自的生命周期，部分可以属于多个整体对象，也可以为多个整体对象共享；
在代码层面：聚合和关联关系是一致的，只能从语义级别来区分：关联关系中两个类是处于相同的层次，而聚合关系中两个类是处于不平等的层次，一个表示整体，一个表示部分；
UML 表示方法：尾部为空心菱形的实线箭头（也可以没箭头）。

```
class B {...}
class A { B* b; ... }
```

4）组合/合成关系：组合也是关联关系的一种特例，体现的是一种 contains-a 的关系，这种关系比聚合更强，也称为强聚合；同样体现整体与部分间的关系，但此时整体与部分是不可分的，整体的生命周期结束也就意味着部分的生命周期结束。整体类负责部分类对象的生存与消亡；
在代码层面：组合和关联关系是一致的，只能从语义级别来区分。
UML 表示方法：尾部为实心菱形的实现箭头（也可以没箭头）。

```
class B {...}
class A { B b; ...}
或
class A {
private: B *pb;
```

```
public: A():pb(new B){}
        ~A(){ delete pb; }
}
```

5）泛化关系：泛化是一种一般与特殊、一般与具体之间关系的描述，具体描述建立在一般描述的基础之上，并对其进行了扩展；
在代码层面：泛化是通过继承实现的；
UML 表示方法：空心三角形箭头的实线。

```
class B { }
class A : public B{ }
```

6）实现关系：实现是一种类与接口的关系，表示类是接口所有特征和行为的实现；
在代码层面：实现一般通过类实现接口来描述的；
UML 表示方法：空心三角形箭头的虚线。

```
interface A {…}
class B implements A {…}
```

7）几种关系所表现的强弱程度依次为：**泛化/实现>组合>聚合>关联>依赖**。

2、单一职责原则（Single Responsibility Principle，SRP）

1）定义：应该有且仅有一个原因引起类的变更。
2）为什么一个类不能有多于一个以上的职责呢？
如果一个类具有一个以上的职责，那么就会有多个不同的原因引起该类变化，而这种变化将影响到该类不同职责的使用者（不同用户）：
（1）一方面，如果一个职责使用了外部类库，则使用另外一个职责的用户却也不得不包含这个未被使用的外部类库。
（2）另一方面，某个用户由于某个原因需要修改其中一个职责，另外一个职责的用户也将受到影响，他将不得不重新编译和配置。
3）我们是面向接口编程，对外公布的是接口而不是实现类。
4）SRP 适用于接口、类以及方法。
5）建议：接口一定要做到 SRP，类的设计尽量做到只有一个原因引起变化（但很难实现）。

3、里氏替换原则（Liskov Substitution Principle，LSP）

1）定义：只要父类能出现的地方子类就可以出现。
2）如果子类不能完整的实现父类的方法，或者父类的某些方法在子类中已经发生“畸变”，则建议断开父子继承关系，采用组合、聚集、依赖等关系代替继承。
3）重写或实现父类的方法时输入参数可以被放大，输出结果可以被缩小。
4）在进行设计的时候，我们尽量从抽象类继承，而不是从具体类继承。如果从继承等级树来看，所有叶子节点应当是具体类，而所有的树枝节点应当是抽象类或者接口。

4、依赖倒置原则（Dependence Inversion Principle，DIP）

1）定义：模块间的依赖通过抽象发生，实现类之间不发生直接的依赖关系，其依赖关系是

通过接口或抽象产生的；接口或抽象类不依赖于实现类，实现类依赖接口或抽象类。
2）任何类都不应该从具体类派生，每个类都尽量有接口或抽象类，尽量不重写基类的方法。
3）变量的表面类型尽量是接口或抽象类。
4）结合 LSP 使用：接口负责定义公共属性和方法，并且声明与其他对象的依赖关系，抽象类负责公共构造部分的实现，实现类准确的实现业务逻辑，同时在适当的时候对其父类进行细化。
5）依赖正置就是实现类间的依赖关系，比如一个实现类的方法的参数类型是另一个实现类的类型，也就是面向实现编程，这也是人的正常思维方式，我要开奔驰就依赖奔驰车，我要用 iPhone 就依赖 iPhone；而编程需要对现实世界的事物进行抽象，抽象的结果就有了抽象类和接口，从而产生了抽象间的依赖，代替了人们传统思维中事物间的依赖，“倒置”由此产生。
6）依赖倒置原则的优点在小型项目中很难体现，在大型项目中，依赖倒置很好的使各个类和模块彼此独立，不互相影响，实现了代码间的松耦合，实现了并行开发；并且是项目更容易维护和扩展。

5、接口隔离原则（Interface Segregation Principle，ISP）

1）定义：要求对接口尽量细化，保证其纯洁性。不能强迫用户去依赖那些他们不使用的接口。换句话说，使用多个专门的接口比使用单一的总接口总要好。
2）与 SRP 的区别：审视角度不同，单一职责要求的是类和接口职责单一，注重的是职责，这是业务逻辑上的划分，而接口隔离原则要求接口的方法尽量少。
3）根据接口隔离原则拆分接口时，首先必须满足单一职责原则。

6、迪米特原则（Law of Demeter，LoD）

1）定义：也称为最少知道原则（Least Knowledge Principle，LKP），一个类应该对自己需要耦合或调用的类知道的最少，只需要知道对方提供的 public 方法，对内部实现概不关心。
2）一个类只和朋友类交流。朋友类：出现在成员变量、方法的输入输出参数中的类，而出现在方法内部的类不属于朋友类，方法是类的一个行为，类竟然不知道自己的行为与其他类产生了依赖关系，这是不允许的。
3）朋友间也是有距离的。强耦合会造成程序的可维护性、重用性大大降低，一个类公开的 public 属性或方法越多，修改时涉及的面也就越大，变更引起的风险扩散也就越大。因此，为了保持朋友类间的距离，在设计时需要反复衡量：是否还可以再减少 public 方法和属性，是否可以修改为 private 等。
4）迪米特原则的初衷在于降低类之间的耦合。由于每个类尽量减少对其他类的依赖，因此，很容易使得系统的功能模块功能独立，相互之间不存在（或很少有）依赖关系。

7、开闭原则（Open Closed Principle，OCP）

1）定义：一个软件实体（模块，类，方法等）应该对扩展开放，对修改关闭。
2）开闭原则是面向对象的可复用设计的基石，其他设计原则是实现开闭原则的手段和工具。根据开闭原则，在设计一个软件系统模块（类，方法）的时候，应该可以在不修改原有的模

块（修改关闭）的基础上，能扩展其功能（扩展开放）。
3）扩展开放：某模块的功能是可扩展的，则该模块是扩展开放的。软件系统的功能上的可扩展性要求模块是扩展开放的。
修改关闭：某模块被其他模块调用，如果该模块的源代码不允许修改，则该模块修改关闭的。软件系统的功能上的稳定性，持续性要求是修改关闭的。
4）接口可以被复用，但接口的实现却不一定能被复用。接口是稳定的，关闭的，但接口的实现是可变的，开放的。可以通过对接口的不同实现以及类的继承行为等为系统增加新的或改变系统原来的功能，实现软件系统的柔性扩展。
简单地说，软件系统是否有良好的接口（抽象）设计是判断软件系统是否满足开闭原则的一种重要的判断基准。现在多把开闭原则等同于面向接口的软件设计。

8、理解感悟

所有的设计原则都是为了降低模块间的耦合性，对数据进行更好的封装，使系统更加稳定，提高了系统的重用性和可维护性，实现大型项目的并行开发。而且设计原则大多要求面向接口编程，而不是面向实现编程；多用组合，少用继承，降低耦合性。