设计模式

- 行为型模式
  - 模板方法模式
  - 策略模式
  - 迭代器模式
- 结构性模式
- 创建型模式

模板方法模式

- 抽象类（父类）定义算法骨架
- 细节由实现类（子类）负责实现
- 定义一个新类，需要重新弄一个实现所有函数的子类
- Q：如果函数之间可自由组合？使用模板方法需要定义所有组合数量个子类。

策略模式

- monitor对函数进行组合

代码实现

```
int main(int argc, char *argv[]){
    //为每个策略的选择具体的实现算法，并创建监控器类
    GangliaLoadStrategy loadStrategy;
    WinMemoryStrategy   memoryStrategy;
    PingLatencyStrategy latencyStrategy;
    WindowsDisplay      display;
    //具体构建过程是将每个策略的具体算法类传入构造函数
    Monitor monitor(&loadStrategy,
                    &memoryStrategy,
                    &latencyStrategy,
                    &display);
    while (running()) {
        //统一的接口获取系统信息
        monitor.getLoad();
        monitor.getTotalMemory();
        monitor.getUsedMemory();
        monitor.getNetworkLatency();
        //统一的接口输出系统信息
        monitor.show();
        sleep(1000);
    }
}
```

比较

- 模板方法：

    - 定义算法的骨架，而将具体实现步骤延迟到子类中。
    - 子类可以不改变算法结构即可重定义该算法的某些特定步骤
    - 优先继承行为，重视功能的抽象与归纳
    - 优点：
        - 基类高度抽象统一，逻辑简洁明了
        - 子类之间关联不紧密时易于简单快速实现
        - 封装性好，实现类内部不会对外暴露
    - 弊端：
        - 接口同时负责所有的功能（算法）
        - 任何算法的修改都导致整个实现类的变化
  - 逻辑复杂但结构稳定
- 策略模式：
    - 定义一系列的算法，把它们一个个封装起来，使它们可相互替换。本模式使得算法可独立于使用它的客户而变化
    - 优先组合行为，重视功能的划分与组合
    - 优点：
        - 每个策略只负责一个功能，易于拓展
        - 算法的修改被限制在单个策略类的变化中，任何算法的修改对整体不造成影响
    - 弊端：
        - 在功能较多的情况下结构复杂
        - 策略组合时对外暴露，封装性相对较差
  - 功能灵活多变，多种算法之间协同工作

迭代器

实现了算法和数据存储的隔离