创建型模式

工厂方法模式

- 一种创建型设计模式，其在父类中提供一个创建对象的方法，允许子类决定实例化对象的类型。
- 工厂方法模式结构

    1. **产品** （Product） 将会对**接口**进行声明。 对于所有由创建者及其子类构建的对象， 这些接口都是通用的。
    2. **具体产品** （Concrete Products） 是产品接口的不同实现。
    3. **创建者** （Creator） 类声明返回产品对象的工厂方法。 该方法的返回对象类型必须与产品接口相匹配。

        可以将工厂方法声明为**抽象方法，** 强制要求每个子类以不同方式实现该方法。 或者， 你也可以在基础工厂方法中返回默认产品类型。
    4. **具体创建者** （Concrete Creators） 将会重写基础工厂方法， 使其返回不同类型的产品。并不一定每次调用工厂方法都会**创建**新的实例。 工厂方法也可以返回缓存、 对象池或其他来源的已有对象。
- 适用场景
    - 当在编写代码的过程中 ，如果无法预知对象确切类别及其依赖关系时，可使用工厂方法
    - 如果你希望复用现有对象来节省系统资源，而不是每次都重新创建对象，可使用工厂方法
    - 如果你希望用户能扩展你软件库或框架的内部组件，可使用工厂方法。
- 实现方式
    1. 让所有产品都遵循同一接口。 该接口必须声明对所有产品都有意义的方法。
    2. 在创建类中添加一个空的工厂方法。 该方法的返回类型必须遵循通用的产品接口。
    3. 在创建者代码中找到对于产品构造函数的所有引用。 将它们依次替换为对于工厂方法的调用， 同时将创建产品的代码移入工厂方法。 你可能需要在工厂方法中添加临时参数来控制返回的产品类型。
    4. 为工厂方法中的每种产品编写一个创建者子类， 然后在子类中重写工厂方法， 并将基本方法中的相关创建代码移动到工厂方法中。
    5. 如果应用中的产品类型太多， 那么为每个产品创建子类并无太大必要， 这时你也可以在子类中复用基类中的控制参数。
    6. 如果代码经过上述移动后， 基础工厂方法中已经没有任何代码， 你可以将其转变为抽象类。 如果基础工厂方法中还有其他语句， 你可以将其设置为该方法的默认行为。
- 工厂方法模式优缺点
    - 优点
        - 可以避免创建者和具体产品之间的紧密耦合。
        - *单一职责原则*。 你可以将产品创建代码放在程序的单一位置， 从而使得代码更容易维护。
        - *开闭原则*。 无需更改现有客户端代码， 你就可以在程序中引入新的产品类型。
    - 缺点
        - 应用工厂方法模式需要引入许多新的子类， 代码可能会因此变得更复杂。 最好的情况是将该模式引入创建者类的现有层次结构中。
- 与其他设计模式的关系

    - 在许多设计工作的初期都会使用工厂方法模式 （较为简单， 而且可以更方便地通过子类进行定制）， 随后演化为使用抽象工厂模式、 原型模式或生成器模式 （更灵活但更加复杂）。
    - 抽象工厂模式通常基于一组工厂方法， 但你也可以使用原型模式来生成这些类的方法。
    - 可以同时使用工厂方法和迭代器模式来让子类集合返回不同类型的迭代器， 并使得迭代器与集合相匹配。
    - 原型并不基于继承， 因此没有继承的缺点。 另一方面， *原型*需要对被复制对象进行复杂的初始化。 工厂方法基于继承， 但是它不需要初始化步骤。
    - 工厂方法是模板方法模式的一种特殊形式。 同时， *工厂方法*可以作为一个大型*模板方法*中的一个步骤。
- 代码
    - 识别方法：工厂方法可通过构建方法来识别， 它会创建具体类的对象， 但以抽象类型或接口的形式返回这些对象
    - Creator类中可以提供一些共有的方法用于调用具体产品类中特有的操作（非必须）
- 总结：
    - 一个工厂基类、一个产品基类
    - 具体的工厂子类生成相应的具体产品类
    - 工厂基类中有一个工厂方法用于生成产品，该方法为虚函数，在子类中实现

抽象工厂模式

- **抽象工厂模式**是一种创建型设计模式， 它能创建**一系列相关**的对象， 而无需指定其具体类。
- 抽象工厂模式

    1. **抽象产品** （Abstract Product） 为构成系列产品的一组不同但相关的产品声明接口。
    2. **具体产品** （Concrete Product） 是抽象产品的多种不同类型实现。
    3. **抽象工厂** （Abstract Factory） 接口声明了一组创建各种抽象产品的方法。
    4. **具体工厂** （Concrete Factory） 实现抽象工厂的构建方法。 每个具体工厂都对应特定产品变体， 且仅创建此种产品变体。
    5. 尽管具体工厂会对具体产品进行初始化， 其构建方法签名必须返回相应的*抽象*产品。 这样， 使用工厂类的客户端代码就不会与工厂创建的特定产品变体耦合。 **客户端** （Client） 只需通过抽象接口调用工厂和产品对象， 就能与任何具体工厂/产品变体交互。
- 适用场景
    - 如果代码需要与多个不同系列的相关产品交互，但是由于无法提前获取相关信息， 或者出于对未来扩展性的考虑， 你不希望代码基于产品的具体类进行构建， 在这种情况下，你可以使用抽象工厂。
        - 抽象工厂为你提供了一个接口， 可用于创建每个系列产品的对象。 只要代码通过该接口创建对象， 那么你就不会生成与应用程序已生成的产品类型不一致的产品。
    - 在设计良好的程序中， *每个类仅负责一件事*。 如果一个类与多种类型产品交互， 就可以考虑将工厂方法抽取到独立的工厂类或具备完整功能的抽象工厂类中。
- 优缺点
    - 优点
        - 可以确保同一工厂生成的产品相互匹配
        - 可以避免客户端和具体产品代码的耦合
        - *单一职责原则*。 你可以将产品生成代码抽取到同一位置， 使得代码易于维护。
        - *开闭原则*。 向应用程序中引入新产品变体时， 你无需修改客户端代码。
    - 缺点
        - 由于采用该模式需要向应用中引入众多接口和类， 代码可能会比之前更加复杂。

- 与其他设计模式的关系
    - 在许多设计工作的初期都会使用工厂方法模式 （较为简单， 而且可以更方便地通过子类进行定制）， 随后演化为使用抽象工厂模式、 原型模式或生成器模式 （更灵活但更加复杂）。
    - 生成器重点关注如何分步生成复杂对象。 抽象工厂专门用于生产一系列相关对象。 *抽象工厂*会马上返回产品， *生成器*则允许你在获取产品前执行一些额外构造步骤。
    - 抽象工厂模式通常基于一组工厂方法， 但你也可以使用原型模式来生成这些类的方法。
    - 当只需对客户端代码隐藏子系统创建对象的方式时， 你可以使用抽象工厂来代替外观模式。
    - 你可以将抽象工厂和桥接模式搭配使用。 如果由*桥接*定义的抽象只能与特定实现合作， 这一模式搭配就非常有用。 在这种情况下， *抽象工厂*可以对这些关系进行封装， 并且对客户端代码隐藏其复杂性。
    - 抽象工厂、 生成器和原型都可以用单例模式来实现。
- 代码
    - 识别方法：会返回一个工厂对象，该工厂被用于创建特定的子组件
- 总结：
    - 一个抽象工厂类（用于客户端操作），多个具体工厂类、多个抽象产品类，多个具体产品类
    - 抽象工厂类包含生成一系列产品的接口，一个具体工厂类可以生成一系列产品对象

生成器模式

- **生成器模式**是一种创建型设计模式， 使你能够**分步骤创建复杂对象。** 该模式允许你使用相同的创建代码生成不同类型和形式的对象。
- 生成器模式

    1. **生成器 （Builder）** 接口声明在所有类型生成器中通用的产品构造步骤。
    2. **具体生成器** （Concrete Builders） 提供构造过程的不同实现。 具体生成器也可以构造不遵循通用接口的产品。
    3. **产品** （Products） 是最终生成的对象。 由不同生成器构造的产品无需属于同一类层次结构或接口。
    4. **主管** （Director） 类定义调用构造步骤的顺序， 这样你就可以创建和复用特定的产品配置。
    5. **客户端** （Client） 必须将某个生成器对象与主管类关联。 一般情况下， 你只需通过主管类构造函数的参数进行一次性关联即可。 此后主管类就能使用生成器对象完成后续所有的构造任务。 但在客户端将生成器对象传递给主管类制造方法时还有另一种方式。 在这种情况下， 你在使用主管类生产产品时每次都可以使用不同的生成器。
- 适用场景
    - 使用生成器模式可避免 “重叠构造函数 （telescopic constructor）” 的出现。
    - 当希望使用代码创建不同形式的产品 （例如石头或木头房屋） 时， 可使用生成器模式。
    - 使用生成器构造**组合树**或其他复杂对象。
        - 生成器模式让你能分步骤构造产品。 你可以延迟执行某些步骤而不会影响最终产品。 你甚至可以递归调用这些步骤， 这在创建对象树时非常方便。生成器在执行制造步骤时， 不能对外发布未完成的产品。 这可以避免客户端代码获取到不完整结果对象的情况。
- 实现方式
    - 清晰的定义通用步骤，确保他们可以制造所有形式的产品。否则无法进一步实施该模式。
    - 在基本生成器接口中声明这些步骤。
    - 为每个形式的产品创建具体生成器类， 并实现其构造步骤。
        - 需要实现获取生成产品对象的方法
    - 考虑创建主管类。 它可以使用同一生成器对象来封装多种构造产品的方式。

    - 客户端代码会同时创建生成器和主管对象。
    - 只有在所有产品都遵循相同接口的情况下， 构造结果可以直接通过主管类获取。 否则， 客户端应当通过生成器获取构造结果。
- 优缺点
    - 优点
        - 可以分步创建对象， 暂缓创建步骤或递归运行创建步骤。
        - 生成不同形式的产品时， 你可以复用相同的制造代码。
        - *单一职责原则*。 你可以将复杂构造代码从产品的业务逻辑中分离出来。
    - 缺点
        - 由于该模式需要新增多个类， 因此代码整体复杂程度会有所增加。
- 与其他模式的关系
    - 在许多设计工作的初期都会使用工厂方法模式 （较为简单， 而且可以更方便地通过子类进行定制）， 随后演化为使用抽象工厂模式、 原型模式或生成器模式 （更灵活但更加复杂）。
    - 生成器重点关注如何分步生成复杂对象。 抽象工厂专门用于生产一系列相关对象。 *抽象工厂*会马上返回产品， *生成器*则允许你在获取产品前执行一些额外构造步骤。
    - 你可以在创建复杂组合模式树时使用生成器， 因为这可使其构造步骤以递归的方式运行。
    - 你可以结合使用生成器和桥接模式： *主管*类负责抽象工作， 各种不同的*生成器*负责*实现*工作。
    - 抽象工厂、 生成器和原型都可以用单例模式来实现。
- 代码
    - 生成器模式可以通过类来识别， 它拥有一个构建方法和多个配置结果对象的方法。 生成器方法通常支持方法链 （例如 `someBuilder->setValueA(1)->setValueB(2)->create()` ）。
- 总结
    - 基类生成器包含多个对产品修饰的函数接口，这些结构都是通用的
    - 具体生成器类需要一个返回产品的方法
    - 主管类可使用具体生成器类生成特定产品
    - 可自由使用生成器类创建产品

原型模式

- **原型模式**是一种创建型设计模式， 能够**复制已有对象**， 而又无需使代码依赖它们所属的类。
- 原型模式将克隆过程委派给被克隆的实际对象。模式为所有支持克隆的对象声明了一个通用接口，该接口让你能够克隆对象， 同时又无需将代码和对象所属类耦合。 通常情况下， 这样的接口中仅包含一个 `克隆` 方法。
- 原型模式结构

    1. **原型** （Prototype） 接口将对克隆方法进行声明。 在绝大多数情况下， 其中只会有一个名为 `clone` 克隆的方法。
    2. **具体原型** （Concrete Prototype） 类将实现克隆方法。 除了将原始对象的数据复制到克隆体中之外， 该方法有时还需处理克隆过程中的极端情况， 例如克隆关联对象和梳理递归依赖等等。
    3. **客户端** （Client） 可以复制实现了原型接口的任何对象。
- 原型注册表实现

    - **原型注册表** （Prototype Registry） 提供了一种访问常用原型的简单方法， 其中存储了一系列可供随时复制的预生成对象。 最简单的注册表原型是一个 `名称 → 原型` 的哈希表。 但如果

需要使用名称以外的条件进行搜索， 你可以创建更加完善的注册表版本。

- 适用场景
  - 如果你需要复制一些对象， 同时又希望代码独立于这些对象所属的具体类， 可以使用原型模式。
    - 这一点考量通常出现在代码需要处理第三方代码通过接口传递过来的对象时。 即使不考虑代码耦合的情况， 你的**代码也不能依赖这些对象所属的具体类**， 因为你不知道它们的具体信息。
  - 如果子类的区别仅在于其对象的初始化方式， 那么你可以使用该模式来减少子类的数量。 别人创建这些子类的目的可能是为了创建特定类型的对象。
    - 在原型模式中， 你可以使用一系列预生成的、 各种类型的对象作为原型。
    - 客户端不必根据需求对子类进行实例化， 只需找到合适的原型并对其进行克隆即可。
- 实现方式
  - 创建原型接口， 并在其中声明 `克隆` 方法。 如果你已有类层次结构， 则只需在其所有类中添加该方法即可。
  - 原型类必须另行定义一个以该类对象为参数的构造函数。 （拷贝构造函数）
  - 克隆方法通常只有一行代码： 使用 `new` 运算符调用原型版本的构造函数。
  - 你还可以创建一个中心化原型注册表， 用于存储常用原型。
    - 可以新建一个**工厂类**来实现注册表， 或者在原型基类中添加一个获取原型的静态方法。 该方法必须能够根据客户端代码设定的条件进行搜索。 搜索条件可以是简单的字符串， 或者是一组复杂的搜索参数。 找到合适的原型后， 注册表应对原型进行克隆， 并将复制生成的对象返回给客户端。
    - 最后还要将对子类构造函数的直接调用替换为对原型注册表工厂方法的调用。
- 优缺点
  - 优点
    - 克隆对象， 而无需与它们所属的具体类相耦合。
    - 克隆预生成原型， 避免反复运行初始化代码。
    - 更方便地生成复杂对象。
    - 可以用继承以外的方式来处理复杂对象的不同配置。
  - 缺点
    - 克隆包含循环引用的复杂对象可能会非常麻烦。
- 与其他设计模式的比较
  - 在许多设计工作的初期都会使用工厂方法模式 （较为简单， 而且可以更方便地通过子类进行定制）， 随后演化为使用抽象工厂模式、 原型模式或生成器模式 （更灵活但更加复杂）。
  - 抽象工厂模式通常基于一组工厂方法， 但你也可以使用原型模式来生成这些类的方法。
  - 原型可用于保存命令模式的历史记录。
  - 大量使用组合模式和装饰模式的设计通常可从对于原型的使用中获益。 你可以通过该模式来复制复杂结构， 而非从零开始重新构造。
  - 原型并不基于继承， 因此没有继承的缺点。 另一方面， *原型*需要对被复制对象进行复杂的初始化。 工厂方法基于继承， 但是它不需要初始化步骤。
  - 有时候原型可以作为备忘录模式的一个简化版本， 其条件是你需要在历史记录中存储的对象的状态比较简单， 不需要链接其他外部资源， 或者链接可以方便地重建。
  - 抽象工厂、 生成器和原型都可以用单例模式来实现。
- 代码
  - 原型可以简单地通过 `clone` 或 `copy` 等方法来识别。
- 总结
  - 原型模式主要有一个原型基类（包含一个clone方法），多个具体原型子类

- 可使用工厂方法模式管理原型子类的生成，在创建对象的方法中适用clone方法返回一个拷贝对象

单例模式

- 单例模式是一种创建型设计模式， 让你能够保证一个类只有一个实例， 并提供一个访问该实例的全局节点。
- 单例模式同时解决了两个问题， 所以违反了*单一职责原则*:
  - **保证一个类只有一个实例。**
  - **为该实例提供一个全局访问节点。**
- 所有单例的实现都包含以下两个相同的步骤:
  - **将默认构造函数设为私有**， 防止其他对象使用单例类的 `new` 运算符。
  - **新建一个静态构建方法作为构造函数。** 该函数会 “偷偷” 调用私有构造函数来创建对象， 并将其保存在一个静态成员变量中。 此后所有对于该函数的调用都将返回这一缓存对象。
- 单例模式结构

  - **单例** （Singleton） 类声明了一个名为 `getInstance` 获取实例的静态方法来返回其所属类的一个相同实例。单例的构造函数必须对客户端 （Client） 代码隐藏。 调用 `获取实例` 方法必须是获取单例对象的唯一方式。
- 基础要点
  - 全局只有一个实例：static 特性，同时禁止用户自己声明并定义实例（把构造函数设为 private）
  - 线程安全
  - 禁止赋值和拷贝
  - 用户通过接口获取实例：使用 static 类成员函数
- **有缺陷的懒汉式**：懒汉式(Lazy-Initialization)的方法是直到使用时才实例化对象，也就说直到调用 get_instance() 方法的时候才 new 一个单例的对象， 如果不被调用就不会占用内存:

```
class Singleton{
private:
    Singleton(){
        std::cout<<"constructor called!"<<std::endl;
    }
    Singleton(Singleton&)=delete;
    Singleton& operator=(const Singleton&)=delete;
    static Singleton* m_instance_ptr;
public:
    ~Singleton(){
        std::cout<<"destructor called!"<<std::endl;
    }
    static Singleton* get_instance(){
        if(m_instance_ptr==nullptr){
              m_instance_ptr = new Singleton;
        }
        return m_instance_ptr;
    }
    void use() const { std::cout << "in use" << std::endl; }
};
```

  - 懒汉式单例模式存在问题:

- **线程安全问题**：当多线程获取单例时有可能引发竞态条件：第一个线程在if中判断`m_instance_ptr`是空的，于是开始实例化单例;同时第2个线程也尝试获取单例，这个时候判断`m_instance_ptr`还是空的，于是也开始实例化单例;这样就会实例化出两个对象,这就是线程安全问题的由来;
  - **解决办法**：加锁
- **内存泄漏**. 注意到类中只负责new出对象，却没有负责delete对象，因此只有构造函数被调用，析构函数却没有被调用;因此会导致内存泄漏。
  - **解决办法**：使用共享指针；或者构造一个嵌套类Deletor专门用于析构对象

- 懒汉式（线程安全、内存安全）

```
class Singleton{
public:
    typedef std::shared_ptr<Singleton> Ptr;
    ~Singleton(){
        std::cout<<"destructor called!"<<std::endl;
    }
    Singleton(Singleton&)=delete;
    Singleton& operator=(const Singleton&)=delete;
    static Ptr get_instance(){

        // "double checked lock"
        if(m_instance_ptr==nullptr){
            std::lock_guard<std::mutex> lk(m_mutex);
            if(m_instance_ptr == nullptr){
              m_instance_ptr = std::shared_ptr<Singleton>(new
Singleton);
            }
        }
        return m_instance_ptr;
    }


private:
    Singleton(){
        std::cout<<"constructor called!"<<std::endl;
    }
    static Ptr m_instance_ptr;
    static std::mutex m_mutex;
};

Singleton::Ptr Singleton::m_instance_ptr = nullptr;
std::mutex Singleton::m_mutex;
```

  - shared_ptr和mutex都是C++11的标准，以上这种方法的优点是
    - 基于 shared_ptr, 用了C++比较倡导的 RAII思想，用对象管理资源,当 shared_ptr 析构的时候，new 出来的对象也会被 delete掉。以此避免内存泄漏。
    - 加了锁，使用互斥量来达到线程安全。这里使用了两个 `if` 判断语句的技术称为**双检锁**；好处是，只有判断指针为空的时候才加锁，避免每次调用 `get_instance()` 的方法都加锁，锁的开销毕竟还是有点大的。
  - 不足之处在于：
    - 使用智能指针会要求用户也得使用智能指针，非必要不应该提出这种约束; 使用锁也有开销; 同时代码量也增多了，实现上我们希望越简单越好。
    - 可能出现*双重检锁*的问题

- 局部静态变量的懒汉式（Meyers Singleton）
    - 原理：如果当变量在初始化的时候，并发同时进入声明语句，并发线程将会阻塞等待初始化结束。这样保证了并发线程在获取静态局部变量的时候一定是初始化过的，所以具有线程安全性。
    - **这是最推荐的一种单例实现方式：**
        1. 通过**局部静态变量的特性保证了线程安全** (C++11, GCC > 4.3, VS2015支持该特性);
        2. 不需要使用共享指针，代码简洁；
        3. 注意在使用的时候需要声明单例的引用 `Single&` 才能获取对象。

```
#include <iostream>

class Singleton
{
public:
    ~Singleton(){
        std::cout<<"destructor called!"<<std::endl;
    }
    Singleton(const Singleton&)=delete;
    Singleton& operator=(const Singleton&)=delete;
    static Singleton& get_instance(){
        static Singleton instance;
        return instance;

    }
private:
    Singleton(){
        std::cout<<"constructor called!"<<std::endl;
    }
};
```

    - 不能返回指针，因为无法避免用户使用 `delete instance` 导致对象被提前销毁

```
static Singleton& get_instance(){
    static Singleton instance;
    return instance;
}
```

- 饿汉式
    - 单例实例在程序运行时被立即执行初始化
    - 由于在main函数之前初始化，所以没有线程安全的问题。
    - 潜在问题在于no-local static对象（函数外的static对象）在不同编译单元中的初始化顺序是未定义的。也即，static Singleton instance;和static Singleton& getInstance()二者的初始化顺序不确定，如果在初始化完成之前调用 getInstance() 方法会返回一个未定义的实例。
- 适用场景
    - 如果程序中的某个类对于所有客户端只有一个可用的实例， 可以使用单例模式。
    - 如果你需要更加严格地控制全局变量， 可以使用单例模式。
- 实现方式
    - 在类中添加一个私有静态成员变量用于保存单例实例。
    - 声明一个公有静态构建方法用于获取单例实例。
    - 禁用拷贝构造函数和赋值构造函数
    - 构造函数设为私有
- 优缺点

    - 优点：
        - 保证一个类只有一个实例。
        - 获得了一个指向该实例的全局访问节点。
        - 仅在首次请求单例对象时对其进行初始化。
    - 缺点：
        - 违反了*单一职责原则*。 该模式同时解决了两个问题。
        - 注意内存安全、线程安全
        - 单元测试困难
- 与其他设计模式的关系
    - 外观模式类通常可以转换为单例模式类， 因为在大部分情况下一个外观对象就足够了。
    - 如果你能将对象的所有共享状态简化为一个享元对象， 那么享元模式就和单例类似了。 但这两个模式有两个根本性的不同。
        1. 只会有一个单例实体， 但是*享元*类可以有多个实体， 各实体的内在状态也可以不同。
        2. *单例*对象可以是可变的。 享元对象是不可变的。
    - 抽象工厂模式、 生成器模式和原型模式都可以用单例来实现。
- 总结
    - 懒汉式
        - 注意线程安全、内存安全
        - Meyers模式
    - 饿汉式
        - 线程是安全的，但是具有潜在问题

结构型模式

适配器模式

- **适配器模式**是一种结构型设计模式， 它能使接口不兼容的对象能够相互合作
    - 适配器实现与其中一个现有对象兼容的接口
    - 现有对象可以使用该接口安全地调用适配器方法
    - 适配器方法被调用后将以另一个对象兼容的格式和顺序将请求传递给该对象
- 适配器模式结构（对象适配器模式）

    1. **客户端** **（Client）** 是包含当前程序业务逻辑的类。
    2. **客户端接口** （Client Interface） 描述了其他类与客户端代码合作时必须遵循的协议。
    3. **服务** （Service） 中有一些功能类 （通常来自第三方或遗留系统）。 客户端与其接口不兼容， 因此无法直接调用其功能。
    4. **适配器** （Adapter） 是一个可以同时与客户端和服务交互的类： 它在实现客户端接口的同时封装了服务对象。 适配器接受客户端通过适配器接口发起的调用， 并将其转换为适用于被封装服务对象的调用。
    5. **客户端代码只需通过接口与适配器交互即可**， 无需与具体的适配器类耦合。 因此， 你可以向程序中添加新类型的适配器而无需修改已有代码。 这在服务类的接口被更改或替换时很有用： 你无需修改客户端代码就可以创建新的适配器类。如下使用继承机制（**类适配器模式**）：

> 1．类适配器不需要封装任何对象， 因为它同时继承了客户端和服务的行为。 适配功能在重写的方法中完成。 最后生成的适配器可替代已有的客户端类进行使用。

- 适用场景
  - 当希望使用某个类， 但是其接口与其他代码不兼容时， 可以使用适配器类。
  - 如果您需要复用这样一些类， 他们处于同一个继承体系， 并且他们又有了额外的一些共同的方法， 但是这些共同的方法不是所有在这一继承体系中的子类所具有的共性。
- 实现方式
  - 确保至少有两个类的接口不兼容：
    - 一个无法修改 （通常是第三方、 遗留系统或者存在众多已有依赖的类） 的功能性*服务*类。
    - 一个或多个将受益于使用服务类的*客户端*类。
  - 声明客户端接口， 描述客户端如何与服务交互。
  - 创建遵循客户端接口的适配器类。 所有方法暂时都为空。
  - 在适配器类中添加一个成员变量用于保存对于服务对象的引用。 通常情况下会通过构造函数对该成员变量进行初始化， 但有时在调用其方法时将该变量传递给适配器会更方便。
  - 依次实现适配器类客户端接口的所有方法。 适配器会将实际工作委派给服务对象， 自身只负责接口或数据格式的转换。
  - 客户端必须通过客户端接口使用适配器。 这样一来， 你就可以在不影响客户端代码的情况下修改或扩展适配器。
- 优缺点
  - 优点：
    - 桥接模式通常会于开发前期进行设计， 使你能够将程序的各个部分独立开来以便开发。 另一方面， 适配器模式通常在已有程序中使用， 让相互不兼容的类能很好地合作。
    - 适配器可以对已有对象的接口进行修改， 装饰模式则能在不改变对象接口的前提下强化对象功能。 此外， *装饰*还支持递归组合， *适配器*则无法实现。
    - 适配器能为被封装对象提供不同的接口， 代理模式能为对象提供相同的接口， 装饰则能为对象提供加强的接口。
    - 外观模式为现有对象定义了一个新接口， 适配器则会试图运用已有的接口。 *适配器*通常只封装一个对象， *外观*通常会作用于整个对象子系统上。
    - 桥接、 状态模式和策略模式 （在某种程度上包括适配器） 模式的接口非常相似。 实际上， 它们都基于组合模式——即将工作委派给其他对象， 不过也各自解决了不同的问题。 模式并不只是以特定方式组织代码的配方， 你还可以使用它们来和其他开发者讨论模式所解决的问题。
- 总结：
  - 分为类模式适配器和对象模式适配器
    - 类模式：双重继承，继承目标类和适配类
    - 对象模式：继承目标类，适配器类依赖适配类

外观模式

- **外观模式**是一种结构型设计模式， 能为程序库、 框架或其他复杂类提供一个简单的接口。
- 外观类为包含许多活动部件的复杂子系统提供一个简单的接口。
- 外观模式

1. **外观** （Facade） 提供了一种访问特定子系统功能的便捷方式， 其了解如何**重定向**客户端请求， 知晓如何操作一切活动部件。
2. 创建**附加外观** （Additional Facade） 类可以避免多种不相关的功能污染单一外观， 使其变成又一个复杂结构。 客户端和其他外观都可使用附加外观。
3. **复杂子系统** （Complex Subsystem） 由数十个不同对象构成。 如果要用这些对象完成有意义的工作， 你必须深入了解子系统的实现细节， 比如按照正确顺序初始化对象和为其提供正确格式的数据。

   子系统类不会意识到外观的存在， 它们在系统内运作并且相互之间可直接进行交互。
4. **客户端** （Client） 使用外观代替对子系统对象的直接调用。

- 适用场景
  - 如果你需要一个指向复杂子系统的直接接口， 且该接口的功能有限， 则可以使用外观模式。
  - 如果需要将子系统组织为多层结构， 可以使用外观。
- 优缺点
  - 优点：可以让代码独立于复杂子系统。
  - 缺点：外观可能成为与程序中所有类都耦合的上帝对象。
- 与其他模式的关系
  - 外观模式为现有对象定义了一个新接口， 适配器模式则会试图运用已有的接口。 *适配器*通常只封装一个对象， *外观*通常会作用于整个对象子系统上。
  - 当只需对客户端代码隐藏子系统创建对象的方式时， 你可以使用抽象工厂模式来代替外观。
  - 享元模式展示了如何生成大量的小型对象， 外观则展示了如何用一个对象来代表整个子系统。
  - 外观和中介者模式的职责类似： 它们都尝试在大量紧密耦合的类中组织起合作。
    - *外观*为子系统中的所有对象定义了一个简单接口， 但是它不提供任何新功能。 子系统本身不会意识到外观的存在。 子系统中的对象可以直接进行交流。
    - *中介者*将系统中组件的沟通行为中心化。 各组件只知道中介者对象， 无法直接相互交流。
  - 外观类通常可以转换为单例模式类， 因为在大部分情况下一个外观对象就足够了。
  - 外观与代理模式的相似之处在于它们都缓存了一个复杂实体并自行对其进行初始化。 *代理*与其服务对象遵循同一接口， 使得自己和服务对象可以互换， 在这一点上它与*外观*不同。
- 总结：
  - Facade类中依赖了多个子系统的类

行为模式

观察者模式

- **观察者模式**是一种行为设计模式， 允许你定义一种订阅机制， 可在对象事件发生时通知多个 “观察” 该对象的其他对象。
- 发布者和订阅者
  - 一个用于存储订阅者对象引用的列表成员变量；
  - 几个用于添加或删除该列表中订阅者的公有方法
- 所有订阅者都必须实现同样的接口， 发布者仅通过该接口与订阅者交互
- 观察者模式

1. **发布者** （Publisher） 会向其他对象发送值得关注的事件。 事件会在发布者自身状态改变或执行特定行为后发生。 发布者中包含一个允许新订阅者加入和当前订阅者离开列表的订阅构架。
2. 当新事件发生时， 发送者会遍历订阅列表并调用每个订阅者对象的通知方法。 该方法是在订阅者接口中声明的。
3. **订阅者** （Subscriber） 接口声明了通知接口。 在绝大多数情况下， 该接口仅包含一个 `update` 更新方法。 该方法可以拥有多个参数， 使发布者能在更新时传递事件的详细信息。
4. **具体订阅者** （Concrete Subscribers） 可以执行一些操作来回应发布者的通知。 所有具体订阅者类都实现了同样的接口， 因此发布者不需要与具体类相耦合。
5. 订阅者通常需要一些上下文信息来正确地处理更新。 因此， 发布者通常会将一些上下文数据作为通知方法的参数进行传递。 发布者也可将自身作为参数进行传递， 使订阅者直接获取所需的数据。
6. **客户端** （Client） 会分别创建发布者和订阅者对象， 然后为订阅者注册发布者更新。

- 对象可在运行时根据程序需要开始或停止监听通知。
- 适用场景
  - 当一个对象状态的改变需要改变其他对象， 或实际对象是事先未知的或动态变化的时， 可使用观察者模式。
  - 当应用中的一些对象必须观察其他对象时， 可使用该模式。 但仅能在有限时间内或特定情况下使用。
- 优缺点
  - 优点：
    - 开闭原则，你无需修改发布者代码就能引入新的订阅者类 （如果是发布者接口则可轻松引入发布者类）。
    - 可以在运行时建立对象之间的联系。
  - 缺点：
    - 订阅者的通知顺序是随机的。
- 与其他模式的关系
  - 责任链模式、 命令模式、 中介者模式和观察者模式用于处理请求发送者和接收者之间的不同连接方式：
    - *责任链*按照顺序将请求动态传递给一系列的潜在接收者， 直至其中一名接收者对请求进行处理。
    - *命令*在发送者和请求者之间建立单向连接。
    - *中介者*清除了发送者和请求者之间的直接连接， 强制它们通过一个中介对象进行间接沟通。
    - *观察者*允许接收者动态地订阅或取消接收请求。
- 总结：
  - 一个目标（subject）基类和一个观察者基类（observer）
  - 目标者基类中包含addOberver、removeObserver和notify的接口，一个观察者列表（链表）
  - 观察者基类中包含一个update接口用于目标对象更新