设计模式--Note

Author：*wangml*

Date：*20210904*

参考课程：C++设计模式哔哩哔哩bilibili

在面对重复出现的问题，使用一套可复用的解决方案，避免重复劳动。

目标

> 1. 理解松耦合设计思想
> 2. 掌握面向对象设计原则
> 3. 掌握重构技法改善设计
> 4. 掌握GOF核心设计模式

分解与抽象

以实现一个画图窗口为例

```
// 分解 分而治之的思想
class Window {
public:
    void drawWindow() {
        ...
        for (auto line : lines) {
            line.drawLine();
        }

        for (auto circle : circles) {
            circle.drawCircle();
        }

        // 增加
        ...
    }
private:
    vector<Line> lines;
    vector<Circle> circles;
    // 增加
};

class Line {
public:
    Line();
    void drawLine();
private:
    Point a;
    Point b;
};

class Circle {
public:
    Circle();
```

```
    void drawCircle();
private:
    Point o;
    double len;
};

...
```

```
// 抽象
class Window {
public:
    void drawWindow() {
        ...
        for (auto shape : shapes) {
            shape->draw();
        }
        ...
    }
private:
    vector<Shape*> shapes;// 析构时 注意销毁
};

class Shape {
public:
    Shape();
    virtual void draw() {}
}

class Line : public Shape {
public:
    Line();
    virtual ~Line();// 使得通过Shape指针可以调用正确的析构函数
    void draw();
private:
    Point a;
    Point b;
};

class Circle : public Shape {
public:
    Circle();
    virtual ~Circle();
    void draw();
private:
    Point o;
    double len;
};
```

考虑，当系统需要增加一个三角形的类

> 此时除了实现一个新的类之外，抽象相对于分解的方法，对原代码的的改动要少得多

```
// 分解
class Triangle {
public:
    ...
    void drawTriangle();
```

```
};


// 抽象
class Triangle : public Shape {
public:
    void draw();
    ...
};
```

面向对象的设计

变化是复用的天敌

面向对象与软件设计

1. 隔离变化：将变化带来的影响降为最小
2. 微观层面：各司其职，新产生的类不影响已存在的类

> **类的责任**

对象

1. 语言层面：封装了数据和代码
2. 规格层面：一系列可被使用的公共接口
3. 概念层面：拥有某种责任的抽象

面向对象设计原则

依赖倒置原则DIP

1. 高层模块（稳定）不应该依赖于低层模块（变化），二者都应该依赖于抽象（稳定）。
2. 抽象（稳定）不应该依赖于实现细节（变化），实现细节应该依赖于抽象（稳定）。

> 这两换句话在上述Window的例子中的体现：
>
> 1. Window不应该因为各个形状(Line等)的变化，而需要调整自身，Window和Line等都转而依赖于抽象的Shape
> 2. 在Shape中不应该使用具体的（Line等）的操作，而是让Line等自己负责自身的实现细节（draw）

开放封闭原则OCP

1. 对扩展开放，对更改封闭
2. 类模块应该是可扩展的，但不可修改

> 在Window中，可以扩展更多的形状（Triangle），而不比对Window做出更改

单一职责原则SRP

1. 一个类应该仅有一个引起它变化的原因
2. 变化的方向隐含着类的责任

Liskov替换原则LSP

1. 子类必须能够替换他们的基类（IS-A）
2. 继承表达类型抽象

接口隔离原则ISP

1. 不应该强迫客户程序依赖它们不用的方法
2. 接口应该小而完备

> 只public必要的，仅子类用的就protected，仅自身使用就private

优先使用对象组合而不是类继承

1. 类继承通常为白箱复用，对象组合通常为黑箱复用
2. 继承在某种程度上破坏了封装性，子类父类耦合度高

> 继承有时候导致父类对子类暴露的东西过多

3. 而对象组合只要求被组合的对象具有良好的接口，耦合程度低

封装变化点

1. 使用封装来创建对象之间的分界层，让设计者可以在分界层的一侧进行修改，而不会对另一侧产生不良影响，从而实现层次之间的松耦合

针对接口编程而不是针对实现编程

1. 不将变量类声明为某个特定的具体类，而是声明为某个接口
2. 客户程序无需获知对象的具体类型，只需知道对象所具有的接口
3. 减少系统中各部分的依赖关系，从而实现 高内聚，松耦合 的类型设计方案

> Window的实现中，分解的实现包含了具体类的vector，而抽象实现使用Shape*

GOF-23

模式分类

1. 从目的来看
    1. 创建型模式
    2. i结构型模式
    3. 行为型模式
2. 从范围看
    1. 类模式处理类与子类的静态关系
    2. 对象模式处理对象间的动态关系

从封装变化角度对模式分类

1. 组件协作

> 框架与应用软件之间的划分
>
> 组件协作模式通过晚期绑定来实现框架和应用程序之间的松耦合

    1. Template Method
    2. Strategy

    3. Observer / Event
2. 单一职责

> 在软件组件的设计中，如果责任划分的不是很清晰，使用继承得到的结果往往时随着需求变化的，子类急剧膨胀，同时充斥着重复代码，这时候的关键是划清责任。

    1. Decorator
    2. Bridge
3. 对象创建

> 通过"对象创建"模式绕开new，来避免对象创建（new）过程中所导致的紧耦合（依赖具体类），从而支持对象创建的稳定。
>
> 它是接口抽象之后的第一步工作。

    1. Factory Method
    2. Abstract Factory
    3. Prototype
    4. Builder
4. 对象性能

> 面向对象很好地解决了"抽象"问题，但是不可避免地要付出一定的代价。对于通常情况来讲，面向对象的成本大都可以忽略不计。但是某些情况下，面向对象所带来的成本必须谨慎处理。

    1. Singleton
    2. Flyweight
5. 接口隔离

> 在组件构建过程中，某些接口之间直接的依赖常常会带来很多问题，甚至根本无法实现。采用添加一层间接（稳定）接口，来隔离本来互相紧密关联的接口是一种常见的解决方案。

    1. Facade
    2. Proxy
    3. Mediator
    4. Adapter
6. 状态变化

> 在组建构建过程中，某些对象的状态经常面临变化，如何对这些变化进行有效的管理？同时又维护高层模块的稳定？"状态变化"模式为这一问题提供了一种解决方案。

    1. Memento
    2. State
7. 数据结构

> 常常有一些组件在内部具有特定的数据结构，如果让客户程序依赖这些特定的数据结构，将极大地破坏组件的复用。这时候，讲这些特定的数据结构封装在内部，在外部提供统一的接口，来实现与特定数据结构无关的访问，是一种行之有效的解决方案。

    1. Composite
    2. Iterator
    3. Chain of Resposibility
8. 行为变化

> 在组件的构建过程中，组件行为的变化经常导致组件本身剧烈的变化。"行为变化"模式将组件的行为和组件本身进行结构，从而支持组件行为的变化，实现两者之间的松耦合。

    1. Command
    2. Visitor
9. 领域问题

> 在特定领域中，某些变化虽然频繁，但可以抽象为某种规则。这时候，结合特定领域，将问题抽象为语法规则，从而给出在该领域下的一般性解决方案。

    1. Interpreter

重构关键技法

1. 静态到动态
2. 早绑定到晚绑定
3. 继承到组合
4. 编译时依赖到运行时依赖
5. 紧耦合到松耦合

> 从不同角度阐述同一个问题?

Template Method

> 定义一个操作中的算法的骨架（稳定），而将一些步骤的实现延迟（变化）到子类中。Template Method使得子类可以不改变（复用）一个算法的结构即可重定义（override重写）该算法的某些特定步骤。

早绑定与晚绑定

区分稳定与变化

要点总结

1. 非常基础的设计模式
2. 最简洁的机制（虚函数的多态性）
3. 提供扩展点（继承+多态）
4. 反向控制结构（Lib控制App），Lib调用App

> 上述实例：
>
> 1. run()是固定的
> 2. 运行时，Lib的run()调用App的Step2、Step4

5. 具体实现中，被Template Method调用的方法可以有也可以没有具体实现，一般推荐设置为protected方法

示例

```
// 结构化
class Library {
public:
    Step1();
    Step3();
    Step5();
    ...
};
```

```
class Application {
public:
    Step2();
    Step4();
};

int main() {
    Library lib;
    Application app;

    lib.Step1();

    if (app.Step2()) {
        lib.Step3();
    }

    for (...) {
        app.Step4();
    }

    lib.Step5();
    ...
}
```

```
// 面向对象
class Library {
public:
    // 稳定中包含变化
    void Run() {
        Step1();

        if (Step2()) {  // 支持变化 虚函数多态调用
            Step3();
        }

        for (...) {
            Step4();// 支持变化 虚函数多态调用
        }

        Step5();
    }

    virtual ~Library();
private:
    Step1();
    Step3();
    Step5();

    virtual Step2();
    virtual Step4();
};

class Application : public Library {
public:
    Step4();
    Step5();
};
```

```
int main() {
    Library* pLib = new Application;
    pLib->run();// run()并不是虚函数，此处调用基类的run()，但是在run()内部的Step2()、Step4()又是虚函数，调用的时Application的Step2()、Step4()
    delete pLib;
    ...
}
```

Strategy

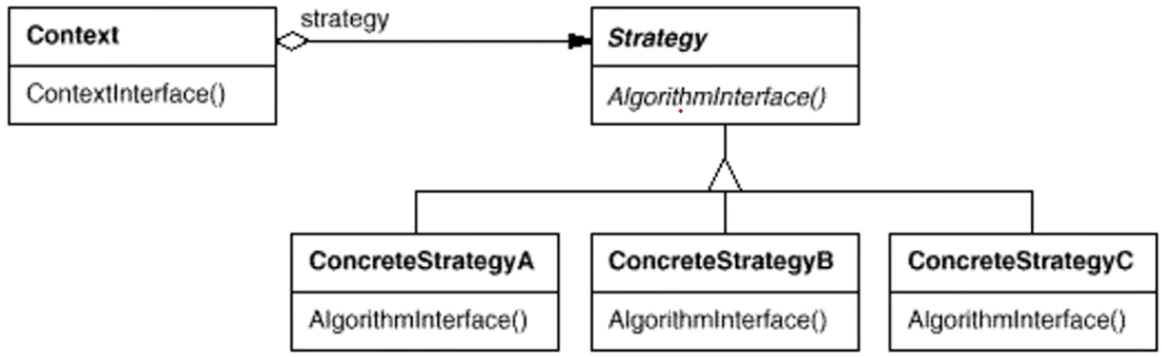
策略

> 定义一系列算法，把它们一个个封装起来，并且使它们可以相互替换（变化）。该模式使得算法可以独立于使用它的客户程序（稳定）而变化（扩展、子类化）。

解决什么问题

> 在软件构建过程中，某些对象使用的算法可能多种多样，经常改变，如果将这些算法都编码到对象中，会使得对象变得异常复杂，而且有时候支持不使用的算法也是一个性能负担。
>
> 如何在运行时根据需求更透明的更改对象的算法?
>
> 将算法与对象本身解耦，从而避免上述问题?

结构



要点总结

1. Strategy及其子类为组件提供了一系列可重用的算法，从而使得类型在运行时方便的根据需要在各个算法之间切换
2. Strategy提供了判断语句外的另一种选择
3. 如果Strategy对象没有实例变量，那么各个上下文可以共享同一个Strategy对象，从而节省开销

示例：税率问题

```
// 结构化
enum TaxBase { // 变化
    CN_Tax,
    US_Tax,
    DE_Tax,
    FR_Tax  // 增加需求
};

class SalesOrder {
public:
```

```
    // 稳定
    double CalculateTax() {
        // ...

        if (tax == CN_Tax) {
            // ...
        }
        else if (tax == US_Tax) {

        }
        else if (tax == DE_Tax) {

        }
        else if (tax == FR_Tax) { // 增加 应该对扩展开放，对修改封闭

        }
        //...
    }

private:
    TaxBase tax;
};
```

```
// Strategy
class TaxStrategy {
public:
    virtual double Calculate(const Context& context)=0;
    virtual ~TaxStrategy(){}// 基类最好都实现一个虚的析构函数
};

// 变化
class CNTax : public TaxStratygy {
public:
    virtual double Calculate(const Context& context) {
        // ...
    }
};

class USTax : public TaxStratygy {
public:
    virtual double Calculate(const Context& context) {
        // ...
    }
};

class DETax : public TaxStratygy {
public:
    virtual double Calculate(const Context& context) {
        // ...
    }
};

// 增加
class FRTax : public TaxStratygy {
public:
    virtual double Calculate(const Context& context) {
        // ...
```

```
    }
};

// 稳定
class SalesOrder {
public:
    SalesOrder(StrategyFactory* strategyFactory) {
        this->strategy = strategyFactory->NewStrategy();
    }

    ~SalesOrder() {
        delete this->strategy;
    }

    double Calculate() {
        // ...
        Context context();

        double val = strategy->Calculate(contex);

        // ...
    }

private:
    TaxStrategy* strategy;
};
```

Observer

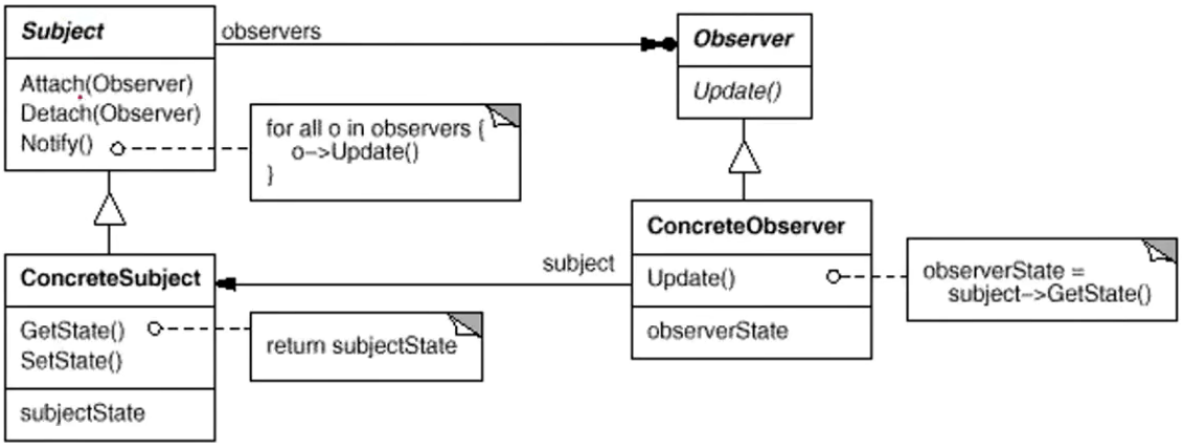
观察者模式

> 定义对象间的一种一对多（变化）的依赖关系，以便当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动更新。

解决什么问题

> 在软件构件过程中，我们需要为某些对象建立一种"通知依赖关系"，一个对象（目标对象）的状态发送改变，所有的依赖对象（观察者对象）都将得到通知。如果这样的依赖关系过于亲密，将使软件不能很好地抵御变化。
>
> 使用面向对象技术，可以将这种依赖关系弱化，并形成一种稳定的依赖关系。从而实现软件体系结构的松耦合。

结构



要点总结

1. 使用面向对象的抽象，Observer模式使得我们可以独立地改变目标与观察者，从而使二者之间的依赖关系达致松耦合。
2. 目标发送通知时，无需指定观察者，通知（可以携带通知信息作为参数）会自动传播。
3. 观察者自己决定是否需要订阅通知，目标对象对此一无所知。
4. Observer模式是基于事件的UI框架中非常常用的设计模式，也是MVC模式的一个重要组成部分。

示例：窗口消息通知

```
// 最初实现
class FileSplitter {
public:
    FileSplitter(const string& filePath, int fileNumber) :
        m_filePath(filePath),
        m_fileNumber(fileNumber) {

    }

    void split() {
        // 1.读取文件

        // 2.分批向小文件写入
        for (int i = 0; i < m_fileNumber; ++i) {

        }
    }

private:
    string m_filePath;
    int m_fileNumber;
};

class MainForm : public Form {
public:
    void Button1_Click() {
        string filePath = txtFilePath->getText();
        int number = atoi(txtFileNumber->getText().c_str());

        FileSplitter splitter(filePath, number);

        splitter.split();
    }

private:
    TextBox* txtFilePath;
    TextBox* txtFileNumber;
};
```

```
// 增加一个通知功能 显示文件切分进度
class FileSplitter {
public:
    FileSplitter(const string& filePath, int fileNumber, ProgressBar*
progressBar) :
        m_filePath(filePath),
        m_fileNumber(fileNumber),
```

```
        m_progressBar(progressBar) {

    }

    void split() {
        // 1.读取文件

        // 2.分批向小文件写入
        for (int i = 0; i < m_fileNumber; ++i) {
            // ...

            if (m_progressBar != nullptr) {
                float progressValue = m_fileNumber;
                m_progressBar->setValur((i + 1) / progressValue);
            }

        }
    }

private:
    string m_filePath;
    int m_fileNumber;

    ProgressBar* m_progressBar;// 通知的方式是变化的 当通知是稳定的 稳定不应该依赖变化，而应该依赖抽象
};

class MainForm : public Form {
public:
    void Button1_Click() {
        string filePath = txtFilePath->getText();
        int number = atoi(txtFileNumber->getText().c_str());

        FileSplitter splitter(filePath, number, progressBar);

        splitter.split();
    }

private:
    TextBox* txtFilePath;
    TextBox* txtFileNumber;

    ProgressBar* progressBar;
};
```

```
// 通知时，除了ProgressBar外，可能会使用其他方式
// 间ProgressBar与FileSplitter解耦合
class IProgress {
public:
    virtual void DoProgress(float value) = 0;
    virtual ~IProgress() {}
};

class ProgressBar {
public:
    void setValue(float value) {
        // ...
```

```
    }
}

class FileSplitter {
public:
    FileSplitter(const string& filePath, int fileNumber, IProgress* iprogress) :
        m_filePath(filePath),
        m_fileNumber(fileNumber),
        m_progressBar(iprogress) {

    }

    void split() {
        // 1.读取文件

        // 2.分批向小文件写入
        for (int i = 0; i < m_fileNumber; ++i) {
            // ...
            float progressValue = m_fileNumber;
            progressValue = (i + 1) / progressValue;
            onProgress(progressValue);

        }
    }
protected:
    void onProgress(float value) {
        if (m_iprogress != nullptr) {
            m_iprogress->DoProgress(value);
        }
    }


private:
    string m_filePath;
    int m_fileNumber;

    //ProgressBar* m_progressBar;// 具体通知控件
    IProgress* m_iprogress;   // 抽象通知
};

class MainForm : public Form, public IProgress {
public:
    void Button1_Click() {
        string filePath = txtFilePath->getText();
        int number = atoi(txtFileNumber->getText().c_str());

        FileSplitter splitter(filePath, number, this);

        splitter.split();
    }

    virtual void DoProgress(float value) {
        progressBar->setValue(value);
    }

private:
    TextBox* txtFilePath;
    TextBox* txtFileNumber;
```

```
    ProgressBar* progressBar;
};
```

```
// 支持出现变化时通知多个对象
class IProgress {
public:
    virtual void DoProgress(float value) = 0;
    virtual ~IProgress() {}
};

class ProgressBar {
public:
    void setValue(float value) {
        // ...
    }
}

class ConsoleNotifier : public IProgress {
public:
    virtual void DoProgress(float value) {
        // ...
    }
};

class FileSplitter {
public:
    FileSplitter(const string& filePath, int fileNumber) :
        m_filePath(filePath),
        m_fileNumber(fileNumber) {

    }

    void split() {
        // 1.读取文件

        // 2.分批向小文件写入
        for (int i = 0; i < m_fileNumber; ++i) {
            // ...
            float progressValue = m_fileNumber;
            progressValue = (i + 1) / progressValue;
            onProgress(progressValue);

        }
    }

    // 稳定
    void addIProgress(IProgress* iprogress) {
        m_iprogressList.push_back(iprogress);
    }

    void removeIProgress(IProgress* iprogress)
        m_iprogressList.remove(iprogress);
    }
protected:
    void onProgress(float value) {
        for (auto iprogress : m_iprogressList) {
```

```
            m_iprogress->DoProgress(value);
        }
    }


private:
    string m_filePath;
    int m_fileNumber;

    //ProgressBar* m_progressBar;// 具体通知控件
    //IProgress* m_iprogress;   // 抽象通知
    vector<IProgress*> m_iprogressList;  // 支持多个观察者
};

class MainForm : public Form, public IProgress {
public:
    void Button1_Click() {
        string filePath = txtFilePath->getText();
        int number = atoi(txtFileNumber->getText().c_str());

        ConsoleNotifier cn;
        FileSplitter splitter(filePath, number);

        splitter.addIProgress(this);
        splitter.addIProgress(&cn);

        splitter.split();
    }

    virtual void DoProgress(float value) {
        progressBar->setValue(value);
    }

private:
    TextBox* txtFilePath;
    TextBox* txtFileNumber;

    ProgressBar* progressBar;
};
```

Decorator

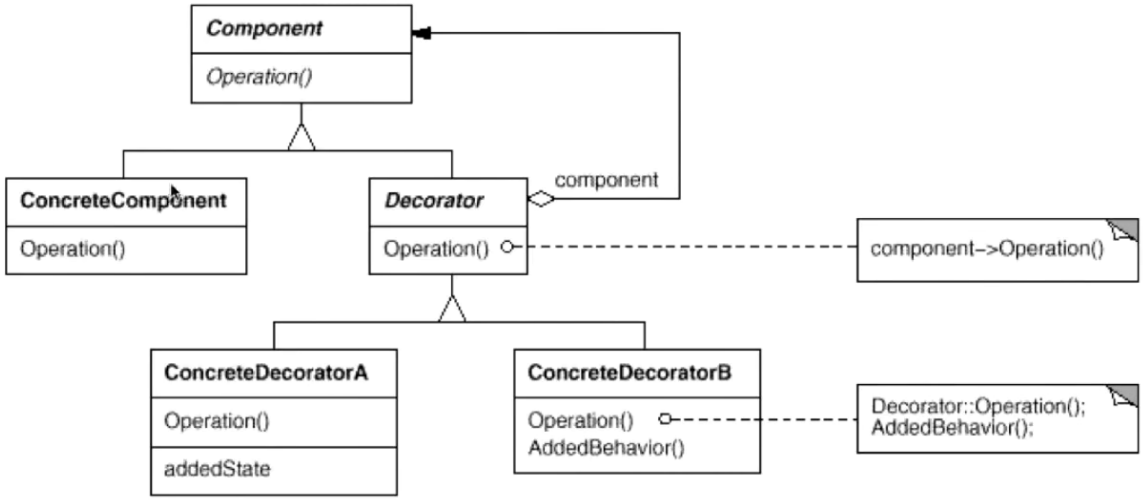
装饰

> 动态（组合）地给一个对象增加一些额外的职责。就增加功能而言，Decorator模式比生成子类（继承）更为灵活（消除重复代码&减少子类个数）。

解决什么问题

> 在某些情况下我们可能会"过度地使用继承来扩展对象的功能"，由于继承为类型引入的静态特质（继承自基类的东西无法改变，被绑死在这个基类上），使得这种扩展方式缺乏灵活性；并且随着子类的增多（扩展功能的增多），各种子类的组合（扩展功能的组合）会导致更多子类的膨胀。
>
> 如何使"对象功能的扩展"能够根据需要来动态地实现？同时避免"扩展功能的增多"带来的子类膨胀问题？从而使得任何"功能扩展变化"所导致的影响降为最低。

结构



要点总结

1. 通过采用组合而非继承的手法，Decorator模式实现了在运动时动态扩展对象功能的能力，而且可以根据需要扩展多个功能。避免了使用继承带来的"灵活性差"和"多子类衍生问题"。
2. Decorator类在接口上表现为is-a Component的继承关系，即Decorator类继承了Component类所具有的接口。但在实现上又表现为has-a Component的组合关系，即Decorator类又使用了另外一个Component类。
3. Decorator模式的目的并非解决"多子类衍生的多继承"问题，Decorator模式应用的要点在于解决"主体类在多个方向上的扩展功能"--是为"装饰"的含义。

注意：当某个类继承自另一个类，且拥有该类指针，则很有可能采用的是Decorator设计模式

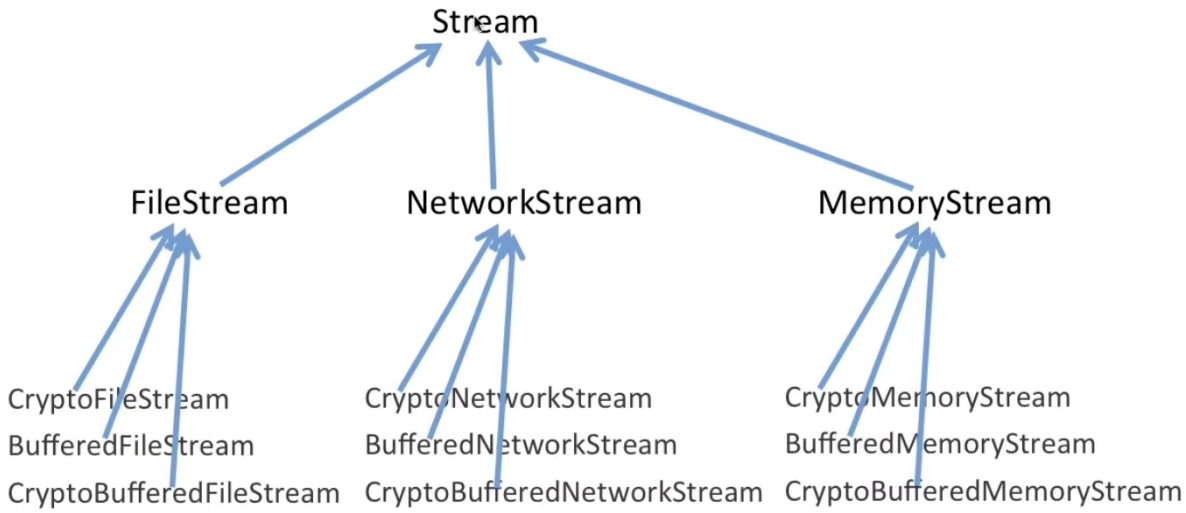
示例：流操作

```
// 多子类继承
// 业务操作
class Stream {
public:
    virtual char Read(int number)=0;
    virtual void Seek(int position)=0;
    virtual void Write(char data)=0;

    virtual ~Stream() {}
};

// 主体类
class FileStream : public Stream {
public:
    virtual char Read(int number) {
        // 文件流读
    }
    virtual void Seek(int position) {
        // 文件流定位
    }
    virtual void Write(char data) {
        // 文件流写
    }
};

class NetworkStream : public Stream {
```

```
public:
    virtual char Read(int number) {
        // 网络流读
    }
    virtual void Seek(int position) {
        // 网络流定位
    }
    virtual void Write(char data) {
        // 网络流写
    }
};

class MemoryStream : public Stream {
public:
    virtual char Read(int number) {
        // 内存流读
    }
    virtual void Seek(int position) {
        // 内存流定位
    }
    virtual void Write(char data) {
        // 内存流写
    }
};

// 扩展操作
// 加密功能
class CryptoFileStream : public FileStream {
public:
    virtual char Read(int number) {
        // 额外的加密操作

        FileStream::Read(number); // 读文件流
        // ...
    }

    virtual void Seek(int position) {
        // 额外的加密操作

        FileStream::Seek(position); // 读文件流
        // ...
    }
    virtual void Write(char data) {
        // 额外的加密操作

        FileStream::Write(data); // 读文件流
        // ...
    }
};

class CryptoNetworkStream : public NetworkStream {
public:
    virtual char Read(int number) {
        // 额外的加密操作

        NetworkStream::Read(number); // 读文件流
        // ...
    }
```

```
    virtual void Seek(int position) {
        // 额外的加密操作

        NetworkStream::Seek(position); // 读文件流
        // ...
    }
    virtual void Write(char data) {
        // 额外的加密操作

        NetworkStream::Write(data); // 读文件流
        // ...
    }
};

class CryptoMemoryStream : public MemoryStream {
    // ...
};

// 缓冲功能
class BufferedFileStream : public FileStream {
    // ...
};

class BufferedNetworkStream : public NetworkStream {
    // ...
};

class BufferedMemoryStream : public MemoryStream {
    // ...
};

// 缓冲+加密
class CryptoBufferedFileStream : public FileStream {
public:
    virtual char Read(int number) {
        // 加密操作
        // 缓冲操作
        FileStream::Read(number);
    }

    // ...
};

// ...

void Process() {
    // 编译时装配
    CryptoFileStream *fs1 = new CryptoFileStream();

    BufferedFileStream *fs2 = new BufferedFileStream();

    CryptoBufferedFileStream *fs3 = new CryptoBufferedFileStream();
}
```

类图关系




```cpp
// 修改后
// 业务操作
class Stream {
public:
    virtual char Read(int number)=0;
    virtual void Seek(int position)=0;
    virtual void Write(char data)=0;

    virtual ~Stream() {}
};

// 主体类
class FileStream : public Stream {
public:
    virtual char Read(int number) {
        // 文件流读
    }
    virtual void Seek(int position) {
        // 文件流定位
    }
    virtual void Write(char data) {
        // 文件流写
    }
};

class NetworkStream : public Stream {
public:
    virtual char Read(int number) {
        // 网络流读
    }
    virtual void Seek(int position) {
        // 网络流定位
    }
    virtual void Write(char data) {
        // 网络流写
    }
};

class MemoryStream : public Stream {
public:
    virtual char Read(int number) {
        // 内存流读
```

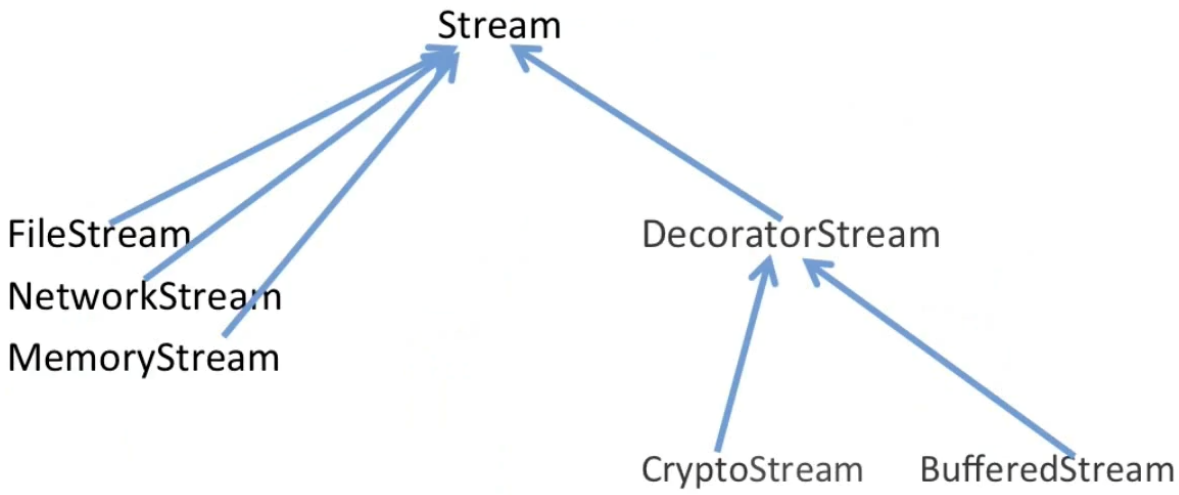
```
    }
    virtual void Seek(int position) {
        // 内存流定位
    }
    virtual void Write(char data) {
        // 内存流写
    }
};

// 扩展操作
// 加密功能
class CryptoStream : public Stream {
    Stream* stream;  // 将由继承而被固定的 改为运行时动态的绑定
public:
    CryptoStream(Stream* stm) : stream(stm) { }

    virtual char Read(int number) {
        // 额外的加密操作

        stream->Read(number); // 读文件流
        // ...
    }

    virtual void Seek(int position) {
        // 额外的加密操作

        stream->Seek(position); // 读文件流
        // ...
    }
    virtual void Write(char data) {
        // 额外的加密操作

        stream->Write(data); // 读文件流
        // ...
    }
};

// 缓冲功能
class BufferedStream : public Stream {
    // ...
};

void Process() {
    // 运行时装配
    FileStream *s1 = new FileStream();
    CryptoStream* s2 = new CryptoStream(s1);
    BufferedStream *s3 = new BufferedStream(s1);
    BufferedStream* s4 = new BufferedStream(s2);
}
```

```
// Decorator
// 业务操作
class Stream {
public:
    virtual char Read(int number)=0;
    virtual void Seek(int position)=0;
    virtual void Write(char data)=0;
```

```
    virtual ~Stream() {}
};

// 主体类
class FileStream : public Stream {
public:
    virtual char Read(int number) {
        // 文件流读
    }
    virtual void Seek(int position) {
        // 文件流定位
    }
    virtual void Write(char data) {
        // 文件流写
    }
};

class NetworkStream : public Stream {
public:
    virtual char Read(int number) {
        // 网络流读
    }
    virtual void Seek(int position) {
        // 网络流定位
    }
    virtual void Write(char data) {
        // 网络流写
    }
};

class MemoryStream : public Stream {
public:
    virtual char Read(int number) {
        // 内存流读
    }
    virtual void Seek(int position) {
        // 内存流定位
    }
    virtual void Write(char data) {
        // 内存流写
    }
};

// 扩展操作
class DecoratorStream : public Stream {
protected:
    Stream* stream;

    DecoratorStream(Stream* stm) : stream(stm) {

    }
};

// 加密功能
class CryptoStream : public DecoratorStream {
public:
    CryptoStream(Stream* stm) : DecoratorStream(stm) { }
```

```
    virtual char Read(int number) {
        // 额外的加密操作

        stream->Read(number); // 读文件流
        // ...
    }

    virtual void Seek(int position) {
        // 额外的加密操作

        stream->Seek(position); // 读文件流
        // ...
    }
    virtual void Write(char data) {
        // 额外的加密操作

        stream->Write(data); // 读文件流
        // ...
    }
};

// 缓冲功能
class BufferedStream : public DecoratorStream {
    // ...
};

void Process() {
    // 运行时装配
    FileStream *s1 = new FileStream();
    CryptoStream* s2 = new CryptoStream(s1);
    BufferedStream *s3 = new BufferedStream(s1);
    BufferedStream* s4 = new BufferedStream(s2);
}
```

此时的类图关系



Bridge

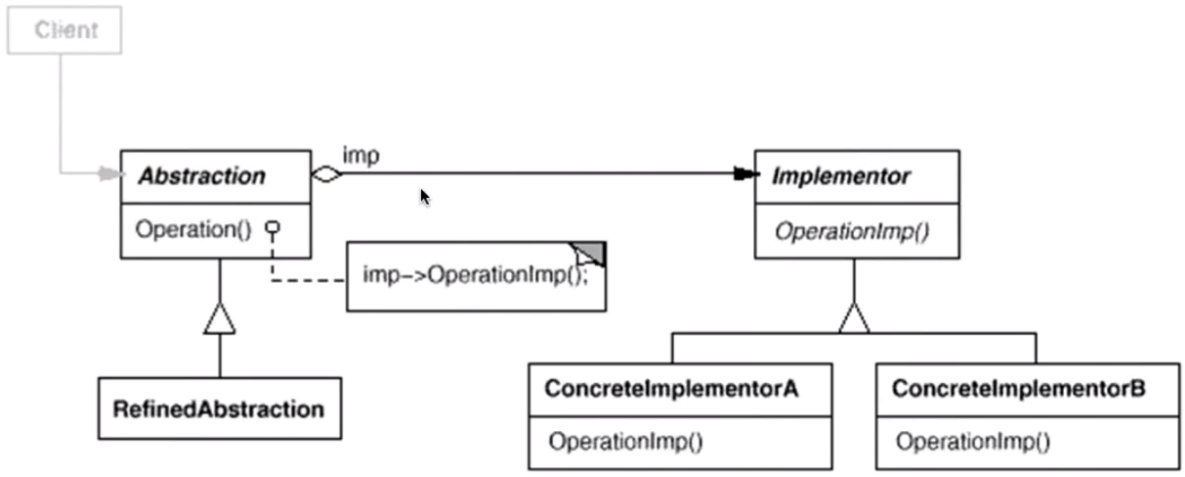
桥

> 将抽象部分（业务功能）与实现部分（平台实现）分离，使得它们都可以独立地变化。

解决什么问题

> 由于某些类型的固有的实现逻辑，使得它们具有两个变化的维度，乃至多个变化的维度。
>
> 如何应对这种"多维度的变化"？如何利用面向对象技术来使得类型可以轻松地沿着两个乃至多个方向变化，而不引入额外的复杂度？

结构



要点总结

1. Bridge模式使用"对象间的组合关系"解耦了**抽象和实现**之间固有的绑定关系，使得抽象和实现可以沿着各自的维度来变化。所谓抽象和实现沿着各自的维度的变化，即"子类化"它们。
2. Bridge模式有时候勒斯与多继承方案，但是多继承方案往往违背单一职责原则（即一个类只有一个变化的原因），复用性比较差。Bridge模式是比多继承方案更好的解决办法。
3. Bridge模式的应用一般在"两个非常强的变化维度"，有时一个类也有多于两个的变化维度，这时可以使用Bridge的扩展模式。

示例：Message轻量版和完美版，且对应不同平台

```
// 最初实现版本
class Messager {
public:
    virtual ~Messager() {}

    virtual void Login(const std::string& username, const std::string& password)
= 0;
    virtual void SendMessage(const std::string& message) = 0;
    virtual void SendPicture(const Image& picture) = 0;

    virtual void PlaySound() = 0;
    virtual void DrawShape() = 0;
    virtual void WriteText() = 0;
    virtual void Connect() = 0;
};

// 平台实现
```

```
class PCMessagerBase : public Message {
public:
    virtual void PlaySound() {
        // ...
    }

    virtual void DrawShape() {
        // ...
    }

    virtual void WriteText() {
        // ...
    }

    virtual void Connect() {
        // ...
    }
}

class MobileMessagerBase : public Message {
public:
    virtual void PlaySound() {
        // ...
    }

    virtual void DrawShape() {
        // ...
    }

    virtual void WriteText() {
        // ...
    }

    virtual void Connect() {
        // ...
    }
}

// 业务抽象
class PCMessagerLite : public PCMessagerBase {
public:
    virtual void Login(const std::string& username, const std::string& password)
{
        PCMessagerBase::Connect();
        // ...
    }

    virtual void SendMessage(const std::string& message) {
        PCMessagerBase::WriteText();
        // ...
    }

    virtual void SendPicture(const Image& picture) {
        PCMessagerBase::DrawShape();
        // ...
    }
};
```

```
class PCMessagerPerfect : public PCMessagerBase {
public:
    virtual void Login(const std::string& username, const std::string& password)
{
        PCMessagerBase::PlaySound();
        // ...
        PCMessagerBase::Connect();
        // ...
    }

    virtual void SendMessage(const std::string& message) {
        PCMessagerBase::PlaySound();
        // ...
        PCMessagerBase::WriteText();
        // ...
    }

    virtual void SendPicture(const Image& picture) {
        PCMessagerBase::PlaySound();
        // ...
        PCMessagerBase::DrawShape();
        // ...
    }
};

class MobileMessagerLite : public MobileMessagerBase {
public:
    virtual void Login(const std::string& username, const std::string& password)
{
        MobileMessagerBase::Connect();
        // ...
    }

    virtual void SendMessage(const std::string& message) {
        MobileMessagerBase::WriteText();
        // ...
    }

    virtual void SendPicture(const Image& picture) {
        MobileMessagerBase::DrawShape();
        // ...
    }
};

class MobileMessagerPerfect : public MobileMessagerBase {
public:
    virtual void Login(const std::string& username, const std::string& password)
{
        MobileMessagerBase::PlaySound();
        // ...
        MobileMessagerBase::Connect();
        // ...
    }

    virtual void SendMessage(const std::string& message) {
        MobileMessagerBase::PlaySound();
        // ...
        MobileMessagerBase::WriteText();
```

```
        // ...
    }

    virtual void SendPicture(const Image& picture) {
        MobileMessagerBase::PlaySound();
        // ...
        MobileMessagerBase::DrawShape();
        // ...
    }
};

// ...

void Process() {
    // 编译时装配
    Message* m = new MobileMessagerPerfect();
}
```

```
// Bridge
class Messager {
protected:
    MessagerImp* messagerImp;
public:
    Messager(MessagerImp* imp) : messagerImp(imp) {

    }

    virtual ~Messager() {}

    virtual void Login(const std::string& username, const std::string& password)
= 0;
    virtual void SendMessage(const std::string& message) = 0;
    virtual void SendPicture(const Image& picture) = 0;
};

class MessagerImp {
public:
    virtual ~MessagerImp() {}

    virtual void PlaySound() = 0;
    virtual void DrawShape() = 0;
    virtual void WriteText() = 0;
    virtual void Connect() = 0;
};

// 平台实现
class PCMessagerImp : public MessagerImp {
public:
    virtual void PlaySound() {
        // ...
    }

    virtual void DrawShape() {
        // ...
    }

    virtual void WriteText() {
```

```
        // ...
    }

    virtual void Connect() {
        // ...
    }
}

class MobileMessagerImp : public MessagerImp {
public:
    virtual void PlaySound() {
        // ...
    }

    virtual void DrawShape() {
        // ...
    }

    virtual void WriteText() {
        // ...
    }

    virtual void Connect() {
        // ...
    }
}

// 业务抽象
class MessagerLite : public Messager {
public:
    MessagerLite(MessagerImp* imp) : Messager(imp) {

    }

    virtual void Login(const std::string& username, const std::string& password)
{
        messagerImp->Connect();
        // ...
    }

    virtual void SendMessage(const std::string& message) {
        messagerImp->WriteText();
        // ...
    }

    virtual void SendPicture(const Image& picture) {
        messagerImp->DrawShape();
        // ...
    }
};

class MessagerPerfect : public Messager {
public:
    MessagerPerfect(MessagerImp* imp) : Messager(imp) {

    }
    virtual void Login(const std::string& username, const std::string& password)
{
```

```
        messagerImp->PlaySound();
        // ...
        messagerImp->Connect();
        // ...
    }

    virtual void SendMessage(const std::string& message) {
        messagerImp->PlaySound();
        // ...
        messagerImp->WriteText();
        // ...
    }

    virtual void SendPicture(const Image& picture) {
        messagerImp->PlaySound();
        // ...
        messagerImp->DrawShape();
        // ...
    }
};

// ...

void Process() {
    // 运行时装配
    MessagerImp* mImp = new PCMessagerImp();
    Message* m = new MessagerPerfect(mImp);
}
```

Factory Method

工厂方法

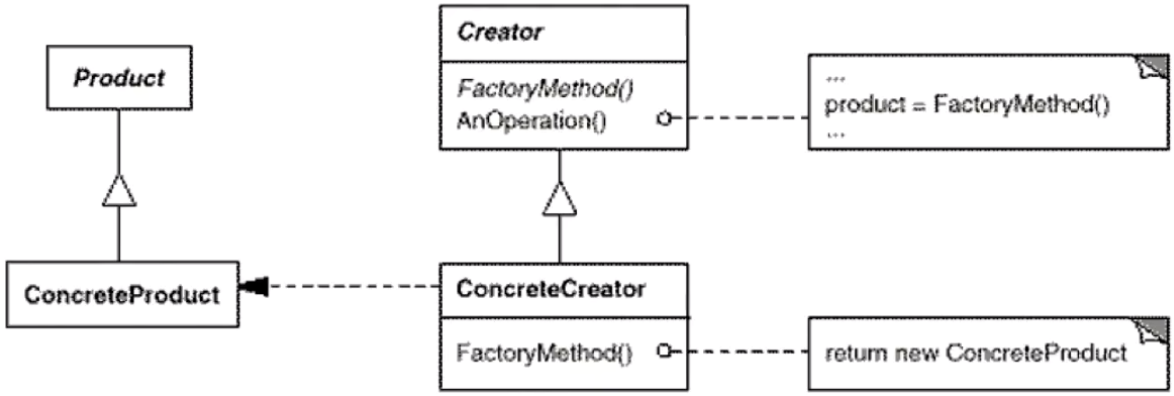
> 定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method使得一个类的实例化延迟（目的：解耦，手段：虚函数）到子类。

解决什么问题

> 在软件系统中，经常面临着创建对象的工作；由于需求的变化，需要创建的对象的具体类型经常变化。
>
> 如何应对这种变化？如何绕过常规的对象创建方法（new），提供一种“封装机制”来避免客户程序和这种“具体对象创建工作”的紧耦合？（new导致的紧耦合）

并非消除变化，而是将变化关进笼子

结构



要点总结

1. Factory Method模式用于隔离类对象的使用者和具体类型之间的耦合关系。面对一个经常变化的具体类型，紧耦合关系（new）会导致软件的脆弱。
2. Factory Method模式通过面向对象的手法，将所要创建的具体对象工作延迟到子类，从而实现一种扩展（而非更改）的策略，较好地解决了这种紧耦合关系。
3. Factory Method模式解决"单个对象"的需求变化。缺点在于要求创建方法/参数相同。

示例：针对不同文件的分割器

```
// 最初实现
class MainForm : public Form {
public:
    void Button1_Click() {
        // ...

        FileSplitter* splitter = new FileSplitter();  // 此处MainForm与
FileSplitter紧耦合 当需要使用其它分割器 就需要更改代码
        splitter->split();
    }
};

class FileSplitter {
public:
    void split() {
        // ...
    }
}

class VideoSplitter {
public:
    void split() {
        // ...
    }
}

class AudioSplitter {
public:
    void split() {
        // ...
    }
}

// ...
```

```
// 进一步 当我们考虑将不同分割器设计成一个抽象基类的子类时
class MainForm : public Form {
public:
    void Button1_Click() {
        // ...

        // Splitter* = new Splitter(); // 抽象基类不能实例化
        Splitter* splitter = new FileSplitter();  // 此时并没有解决MainForm与
FileSplitter紧耦合问题
        splitter->split();
    }
};

class Splitter {
public:
    virtual void split() = 0;
};

class FileSplitter : public Splitter {
public:
    void split() {
        // ...
    }
};

class VideoSplitter : public Splitter {
public:
    void split() {
        // ...
    }
};

class AudioSplitter : public Splitter {
public:
    void split() {
        // ...
    }
};

// ...
```

```
// 使用Factory Method
class MainForm : public Form {
private:
    SplitterFactory* splitterFactory;
public:
    MainForm(SplitterFactory* splitterFactory) {
        this->splitterFactory = splitterFactory;
    }

    void Button1_Click() {
        // ...

        Splitter* splitter = splitterFactory->CreateSplitter();  //
```

```
        splitter->split();
    }
};

class Splitter {
public:
    virtual void split() = 0;
    virtual ~Splitter() {}
};

class FileSplitter : public Splitter {
public:
    void split() {
        // ...
    }
};

class VideoSplitter : public Splitter {
public:
    void split() {
        // ...
    }
};

class AudioSplitter : public Splitter {
public:
    void split() {
        // ...
    }
};

class SplitterFactory {
public:
    virtual Splitter* CreateSplitter() = 0;
    virtual ~SplitterFactory() {}
};

class FileSplitterFactory : public SplitterFactory {
public:
    Splitter* CreateSplitter() {
        return new FileSplitter();
    }
}

class VideoSplitterFactory : public SplitterFactory {
public:
    Splitter* CreateSplitter() {
        return new VideoSplitter();
    }
}

class VideoSplitterFactory : public SplitterFactory {
public:
    Splitter* CreateSplitter() {
        return new VideoSplitter();
    }
}
```

Abstract Factory

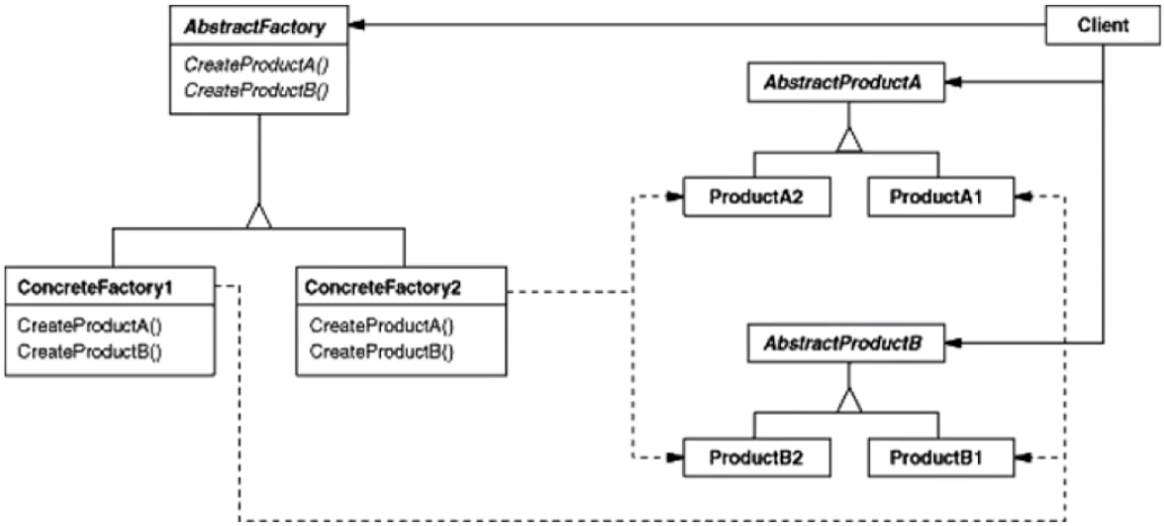
抽象工厂

> 提供一个接口，让该接口负责创建一些列“相关或者相互依赖的对象”，无需指定他们具体的类。

解决什么问题

> 在软件系统中，经常面临着“一系列**相互依赖**的对象”的创建工作；同时，由于需求的变化，往往存在更多系列对象的创建工作。
>
> 如何应对这种变化？如何绕开常规的对象创建方法（new），提供一种“封装机制”来避免客户程序和这种“多系列具体对象创建工作”的紧耦合？

结构



要点总结

1. 如果没有应对”多系列对象构建“的需求变化，则没有必要使用Abstract Factory模式，这时候使用简单的工厂完全可以。
2. ”系列对象“指的是在某一特定系列下的对象之间有**相互依赖、或作用的关系。**不同系列的对象之间不能相互依赖。
3. Abstract Factory模式主要在于应对”新系列“的需求变动。其缺点在于难以应对”新对象“的需求变动。

示例：数据库访问接口

```
// 最初实现
class EmployeeDAO {
public:
    vector<EmployeeDAO> GetEmployees() {
        // 考虑到 当需求变动时 我们需要支持不同的数据库 那么应该怎么做
        SqlConnection* connection = new SqlConnection();
        connection->ConnectionString("...");

        SqlCommand* command = new SqlCommand();
        command->SetConnection(connection);
        command->CommandText("...");

        SqlDataReader* reader = command->ExecuteReader();
        while (reader->Read()) {
```

```
        }
    }
};
```

```
// 首先想到使用Factory Method
class IDBConnection {
public:
    virtual ~IDBConnection() {}
    // 其他接口...
};
class IDBCommand {
public:
    virtual ~IDBCommand() {}
    // 其他接口...
};
class IDBDataReader {
public:
    virtual ~IDBDataReader() {}
    // 其他接口...
};

// 不同数据库支持
class SQLServerConnection : public IDBConnection {

};
class SQLServerCommand : public IDBCommand {

};
class SQLServerDataReader : public IDBDataReader {

};

class MySQLConnection : public IDBConnection {

};
class MySQLCommand : public IDBCommand {

};
class MySQLDataReader : public IDBDataReader {

};
// ...等其他数据库

// 此时还需要实现各个工厂
class IDBConnectionFactory {
public:
    virtual IDBConnection* createDBConnection() = 0;
    // ...
};
class IDBCommandFactory {
public:
    virtual IDBCommand* createDBCommand() = 0;
    // ...
};
class IDBDataReaderFactory {
public:
```

```
    virtual IDBDataReader* createDBDataReader() = 0;
    // ...
};

// 实现各个具体工厂
class SQLServerConnectionFactory : public IDBConnectionFactory {

};
class SQLServerCommandFactory : public IDBCommandFactory {

};
class SQLServerDataReaderFactory : public IDBDataReaderFactory {

};

class MySQLConnectionFactory : public IDBConnectionFactory {

};
class MySQLCommandFactory : public IDBCommandFactory {

};
class MySQLDataReaderFactory : public IDBDataReaderFactory {

};
// ...

class EmployeeDAO {  // 此时好像解决了具体数据库和EmployeeDAO耦合的问题，但是也引入新问题
private:
    // 如果此处 我们传入的Connecion、Command不是同一个数据库
    // 可以很明显看到 下面三个对象必须是同一个数据库 但是在这里的代码并没有保证这一点
    IDBConnectionFactory* dbConnectionFactory;
    IDBCommandFactory* dbCommandFactory;
    IDBDataReaderFactory* dbDataReaderFactory;

public:
    // ...

    vector<EmployeeDAO> GetEmployees() {
        IDBConnection* connection = dbConnectionFactory->createDBConnection();
        connection->ConnectionString("...");

        IDBCommand* command = dbCommandFactory->createDBCommand();
        command->SetConnection(connection);
        command->CommandText("...");

        IDBDataReader* reader = command->ExecuteReader();
        while (reader->Read()) {

        }
    }
};
```

```
// 使用Abstract Factory
class IDBConnection {
public:
    virtual ~IDBConnection() {}
    // 其他接口...
```

```
};
class IDBCommand {
public:
    virtual ~IDBCommand() {}
    // 其他接口...
};
class IDBDataReader {
public:
    virtual ~IDBDataReader() {}
    // 其他接口...
};

// 不同数据库支持
class SQLServerConnection : public IDBConnection {

};
class SQLServerCommand : public IDBCommand {

};
class SQLServerDataReader : public IDBDataReader {

};

class MySQLConnection : public IDBConnection {

};
class MySQLCommand : public IDBCommand {

};
class MySQLDataReader : public IDBDataReader {

};
// ...等其他数据库

// 此时还需要实现各个工厂
class IDBFactory {
public:
    // 高内聚 低耦合
    // 将相关性高的放在一起
    virtual IDBConnection* createDBConnection() = 0;
    virtual IDBCommand* createDBCommand() = 0;
    virtual IDBDataReader* createDBDataReader() = 0;
    // ...
};

// 实现各个具体工厂
class SQLServerFactory : public IDBFactory {

};

class MySQLFactory : public IDBFactory {

};

// ...

class EmployeeDAO {
private:
```

```
    IDBFactory* dbFactory;

public:
    // ...

    vector<EmployeeDAO> GetEmployees() {
        IDBConnection* connection = dbFactory->createDBConnection();
        connection->ConnectionString("...");

        IDBCommand* command = dbFactory->createDBCommand();
        command->SetConnection(connection);
        command->CommandText("...");

        IDBDataReader* reader = command->ExecuteReader();
        while (reader->Read()) {

        }
    }
};
```

Prototype

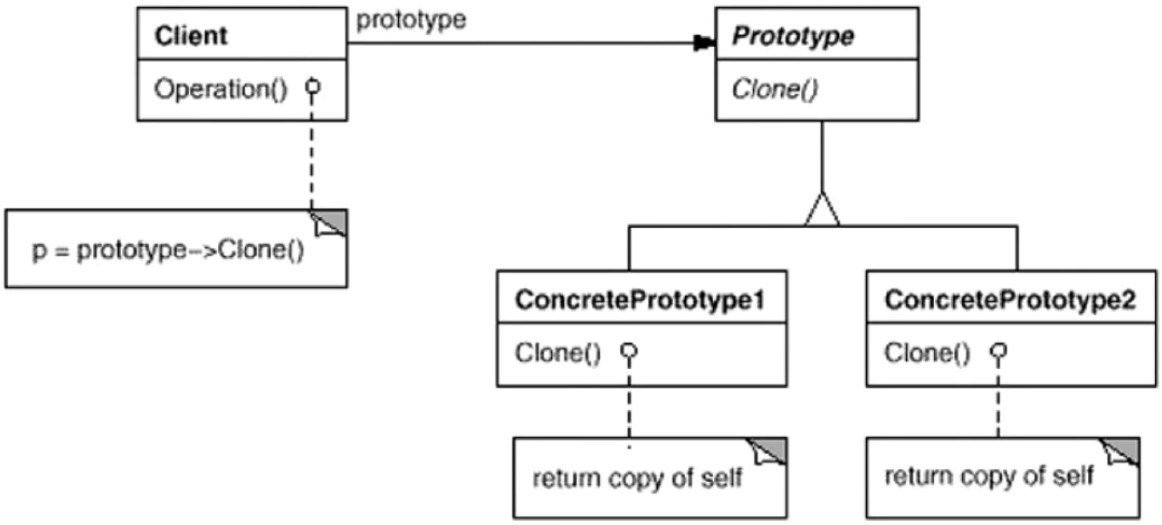
原型模式

> 使用原型实例指定创建对象的种类，然后通过拷贝这些原型来创建新的对象。

解决什么问题

> 在软件系统中，经常面临着**"某些结构复杂的对象"**的创建工作；由于需求的变化，这些对象经常面临着剧烈的变化，但是它们却拥有比较稳定一致的接口。
>
> 如何应对这种变化？如何向"客户程序（使用这些对象的程序）"隔离出"这些易变特性"，从而使得这些易变对象的客户程序"不随着需求的改变而改变"？

结构



要点总结

1. Prototype模式同样用于隔离类对象的使用者和具体类型（易变类）之间的耦合关系，它同样要求这些"易变类"拥有"稳定的接口"。
2. Prototype模式对于"如何创建易变类的实体对象"采用"原型克隆"的方法来做，它使得我们可以非常灵活地动态创建"拥有某些稳定接口"的新对象——所需工作仅仅是注册一个新类的对象（即原型），然后在任何需要的地方Clone。
3. Prototype模式中的Clone方法可以利用某些框架中的序列化来实现深拷贝。

示例：针对不同文件的分割器，对比Factory Method

> 对于选择Factory Method还是Prototype，如果对象可以在Factory中很简单的创建出来，就使用Factory Method，但是对于复杂对象，在创建对象时需要保留各种状态，不方便甚至难以在Factory中做到，这时选用Prototype

```
// 使用Prototype
class MainForm : public Form {
private:
    Splitter* prototype;
public:
    MainForm(Splitter* prototype) {
        this->prototype = prototype;
    }

    void Button1_Click() {
        // ...

        Splitter* splitter = prototype->clone();  // 克隆原型
        splitter->split();
    }
};

class Splitter {
public:
    virtual void split() = 0;
    virtual ~Splitter() {}

    virtual Splitter* clone() = 0;
};

class FileSplitter : public Splitter {
public:
    void split() {
        // ...
    }

    virtual Splitter* clone() {
        return new FileSplitter(*this);
    }
};

class VideoSplitter : public Splitter {
public:
    void split() {
        // ...
    }
```

```
    virtual Splitter* clone() {
        return new VideoSplitter(*this);
    }
};

class AudioSplitter : public Splitter {
public:
    void split() {
        // ...
    }

    virtual Splitter* clone() {
        return new AudioSplitter(*this);
    }
};
```

Builder

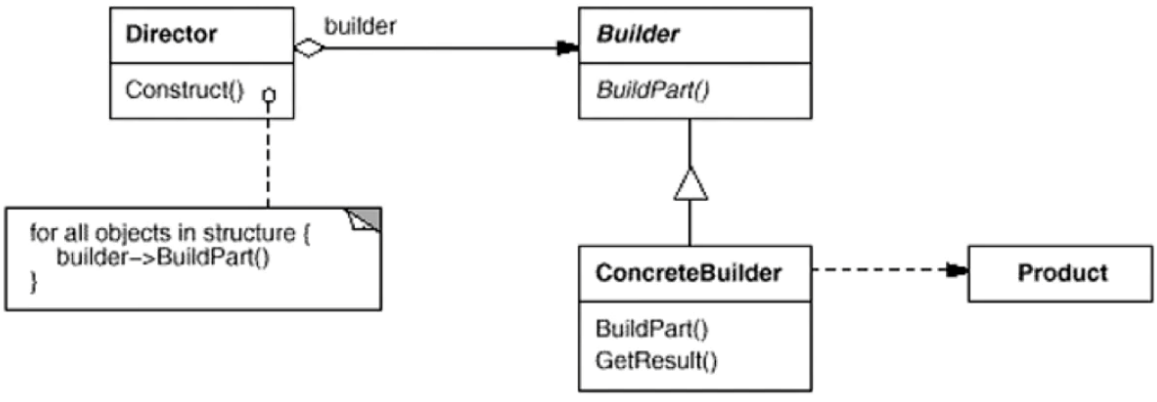
构建器

> 将一个复杂对象的构建与其表示相分离，使得同样的构建过程（稳定）可以创建不同的表示（变化）。

解决什么问题

> 在软件系统中，有时候面临着”一个复杂对象“的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。
>
> 如何应对这种变化？如何提供一种”封装机制“来隔离出”复杂对象的各个部分“的变化，从而保持系统中的”稳定构建算法“不随着需求的改变而改变？

结构



要点总结

1. Builder模式主要用于”分步骤构建一个复杂的对象“。在这其中”分步骤“是一个稳定的算法，而复杂对象的各个部分则经常变化。
2. 变化点在哪里，封装哪里——Builder模式主要在于应对”复杂对象各个部分“的频繁需求变动。其缺点在于难以应对”分步骤构建算法“的需求变动。
3. 在Builder模式中，要注意不同语言中构造器内调用虚函数的差别（C++、C#）

示例：用一个类表示不同的房子，建房子的材料不同，但是步骤一致

```
// 初始实现
class House {
public:
    void Init() {// 稳定
        this->BuildPart1();

        for (int i = 0; i < 4; ++i) {
            this->BuildPart2();
        }

        bool flag = this->BuildPart3();

        if (flag) {
            this->BuildPart4();
        }

        this->BuildPart5();
    }

    virtual ~House() {}

protected:
    // 变化
    virtual void BuildPart1() = 0;
    virtual void BuildPart2() = 0;
    virtual void BuildPart3() = 0;
    virtual void BuildPart4() = 0;
    virtual void BuildPart5() = 0;
};

class StoneHouse : public House {
protected:
    virtual void BuildPart1() {

    }
    virtual void BuildPart2() {

    }
    virtual void BuildPart3() {

    }
    virtual void BuildPart4() {

    }
    virtual void BuildPart5() {

    }
};

void Process {
    House* pHouse = new StoneHouse();
    pHouse->Init();
}
```

```
class House {
    // ...
};

class HouseBuilder {
public:
    House* GetResult() {
        return pHouse;
    }
    virtual ~HouseBuilder() {}

protected:
    House* pHouse;
    // 变化
    virtual void BuildPart1() = 0;
    virtual void BuildPart2() = 0;
    virtual void BuildPart3() = 0;
    virtual void BuildPart4() = 0;
    virtual void BuildPart5() = 0;
};

// 变化
class StoneHouse : public House {

};

class StoneHouseBuilder : public HouseBuilder {
protected:
    virtual void BuildPart1() {
        // pHouse->Part1 = ...
    }
    virtual void BuildPart2() {

    }
    virtual void BuildPart3() {

    }
    virtual void BuildPart4() {

    }
    virtual void BuildPart5() {

    }
};

class HouseDirector {// 稳定
public:
    HouseDirector(HouseBuilder* pHouseBuilder) {
        this->pHouseBuilder = pHouseBuilder;
    }

    House* Construct() {// 稳定
        pHouseBuilder->BuildPart1();

        for (int i = 0; i < 4; ++i) {
            pHouseBuilder->BuildPart2();
        }
```

```
        bool flag = pHouseBuilder->BuildPart3();

        if (flag) {
            pHouseBuilder->BuildPart4();
        }

        pHouseBuilder->BuildPart5();

        return pHouseBuilder->GetResult();
    }
private:
    HouseBuilder* pHouseBuilder;
};

void Process {
    HouseBuilder* pHouseBuilder = new StoneHouseBuilder();
    HouseDirector* pHouseDirector = new HouseDirector(pHouseBuilder);
    House* pHouse = pHouseDirector->Construct();
}
```

Singleton

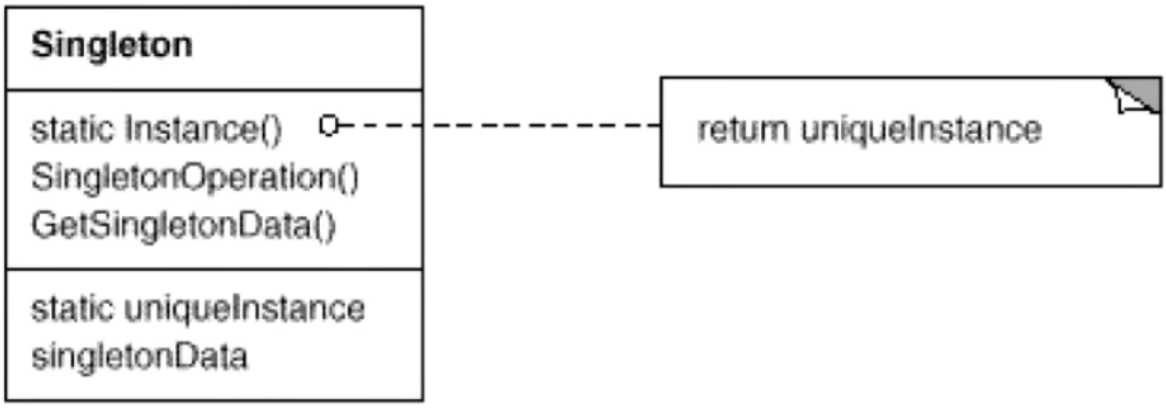
单件模式

> 保证一个类仅有一个实例，并提供一个该实例的全局访问点。

解决什么问题

> 在软件系统中，经常有这样一些特殊的类，必须保证他们在系统中只存在一个实例，才能保证它们逻辑的正确性、以及良好的效率。
>
> 如何绕过常规的构造器，提供一种机制来保证一个类只有一个实例?
>
> 这应该是类设计者的责任，而不是使用者的责任。

结构



要点总结

1. Singleton模式中的构造器可以设置为protected以允许子类派生。
2. Singleton模式一般不要支持拷贝构造函数和Clone接口，因为这有可能导致多个对象实例，与Singleton模式的初衷违背。
3. 如何实现多线程环境下安全的Singleton？注意对双检查锁的正确使用。

示例

> (2 封私信 / 14 条消息) 如何理解 C++11 的六种 memory order？ - 知乎 (zhihu.com)

```
class Singleton {
private:
    Singleton();
    Singleton(const Singleton&);
public:
    static Singleton* getInstance();
    static Singleton* m_instance;
};

Singleton* Singleton::m_instance = nullptr;

// 线程非安全版本
Singleton* Singleton::getInstance() {
    if (m_instance == nullptr) {
        m_instance = new Singleton();
    }
    return m_instance;
}

// 线程安全版本，但是锁代价过高
Singleton* Singleton::getInstance() {
    Lock lock;
    if (m_instance == nullptr) {
        m_instance = new Singleton();
    }
    return m_instance;
}

// 双检查锁，但由于内存读写reorder不安全
Singleton* Singleton::getInstance() {
    if (m_instance == nullptr) {
        Lock lock;
        if (m_instance == nullptr) {
            m_instance = new Singleton();
        }
    }
    return m_instance;
}

// C++11 之后的跨平台实现 （volatile）
std::atomic<Singleton*> Singleton::m_instance;
std::mutex Singleton::m_mutex;

Singleton* Singleton::getInstance() {
    Singleton* tmp = m_instance.load(std::memory_order_relaxed);
    std::atomic_thread_fence(std::memory_order_acquire);// 获取内存fence
    if (tmp == nullptr) {
        std::lock_gard<std::mutex> lock(m_mutex);
        tmp = m_instance.load(std::memory_order_release);
        if (tmp == nullptr) {
            tmp = new Singleton;
            std::atomic_thread_fence(std::memory_order_release);// 释放内存fence
            m_instance.store(tmp, std::memory_order_relaxed);
```

```
        }
    }
    return tmp;
}
```

Flyweight

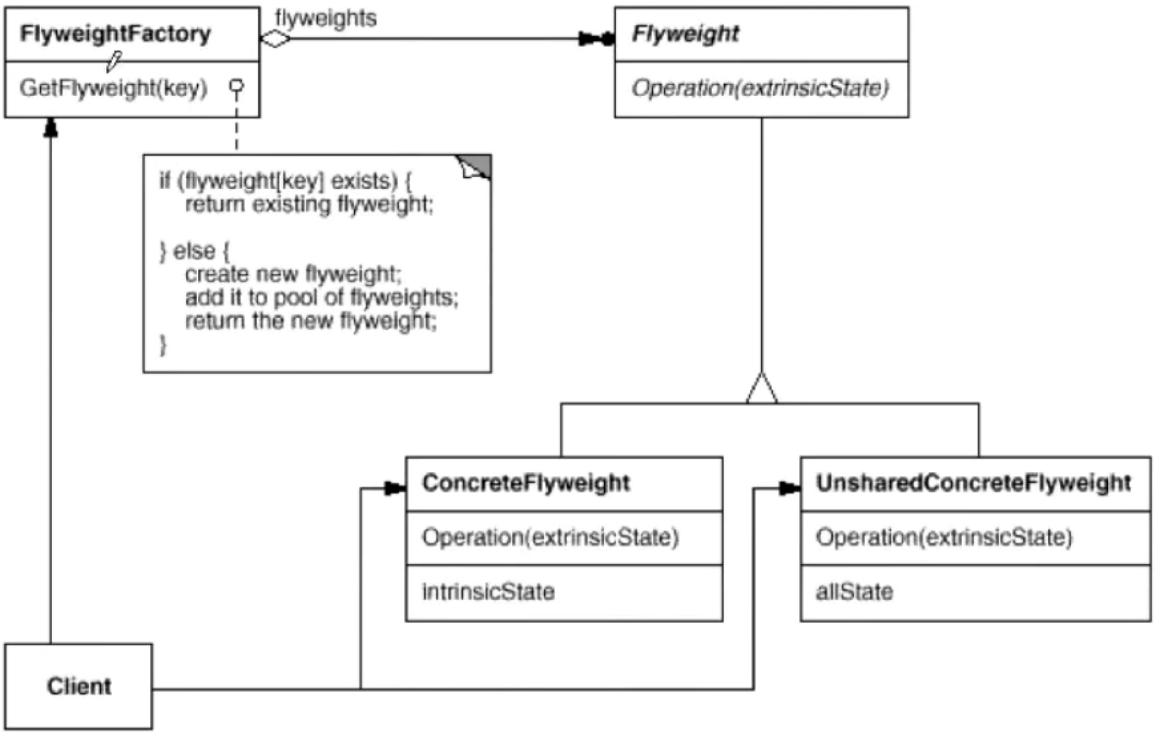
享元模式

> 运用共享技术有效地支持**大量**细粒度的对象。

解决什么问题

> 在软件系统中采用纯粹对象方案的问题在于大量细粒度的对象会很快充斥系统中，从而带来很高的运行时代价——主要指内存需求方面的代价。
>
> 如何在避免大量细粒度对象问题的同时，让外部客户程序仍能够透明地使用面向对象的方式来进行操作?

结构



要点总结

1. 面向对象很好地解决了抽象性的问题，但是作为一个运行在机器中的程序实体，我们需要考虑一个对象的代价问题。Flyweight主要解决面向对象的代价问题，一般不触及面向对象的抽象性问题。
2. Flyweight采用对象共享的做法来降低系统中对象的个数，从而降低细粒度对象给系统带来的内存压力。在具体实现方面，要注意对象状态的处理。
3. 对象的数量太大从而导致对象内存开销加大——什么样的数量才算大？这需要我们仔细地根据具体应用情况进行评估，而不能凭空臆断。

示例：字体对象

```
class Font {
private:
    // unique object key
    string key;
```

```
    // object state
    // ........

public:
    Font(const string& key) {
        // ...
    }
};

class FontFactory {
private:
    map<string, Font*> fontPool;

public:
    Font* GetFont(const string& key) {
        map<string, Font*>::iterator it = fontPool.find(key);

        if (it != fontPool.end()) {
            return fontPool[key];
        }
        else {
            Font* font = new Font(key);
            fontPool[key] = font;
            return font;
        }
    }

    void clear() {
        // ...
    }
};
```

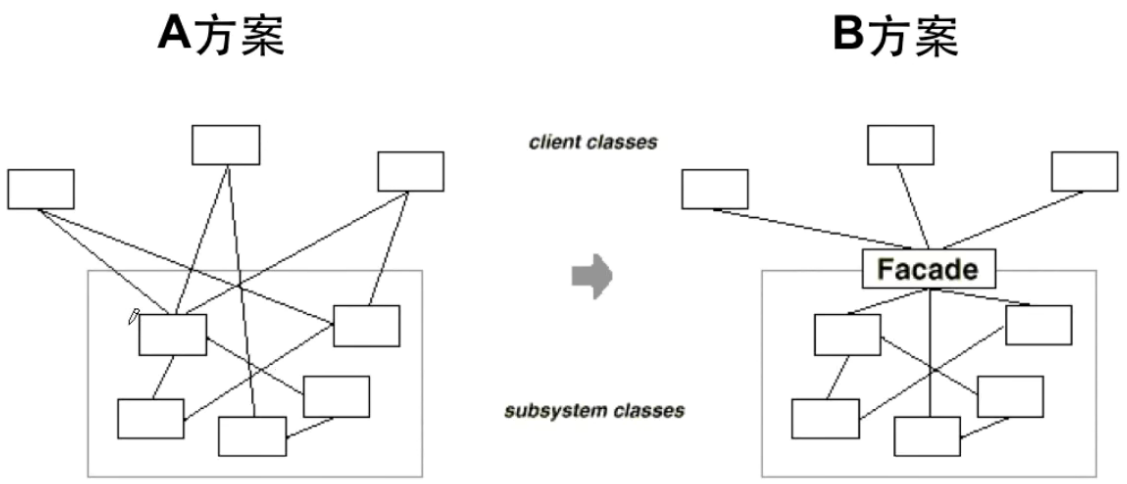
Facade

门面模式

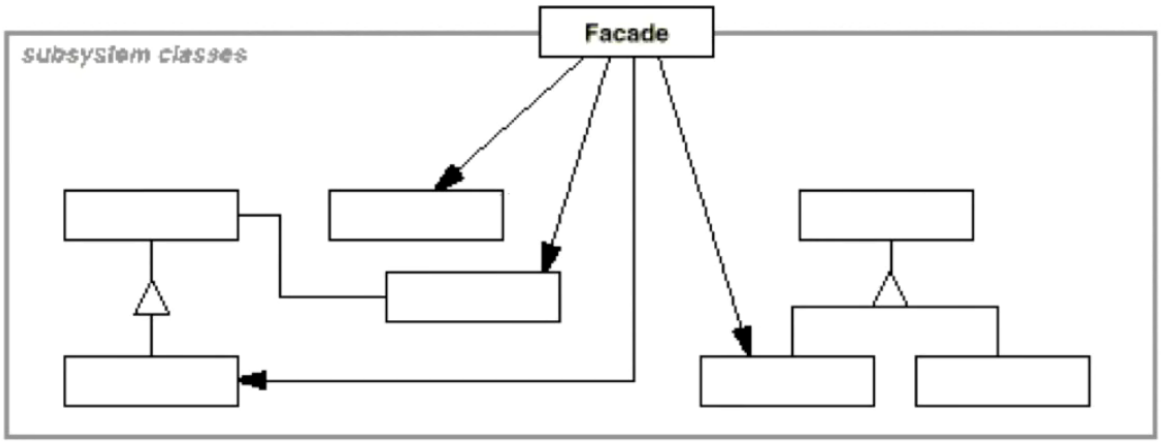
> 为子系统中的一组接口提供一个一致（稳定）的界面，Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用（复用）。

解决什么问题

> **解决系统间耦合的问题**
>
> 下图中方案A的问题在于组件的客户和组件中各种复杂的子系统有了过多的耦合，随着外部客户程序和各子系统的演化，这种过多的耦合面临很多变化的挑战
>
> 如何简化外部客户程序和系统将的交互接口？如何将外部客户程序的演化和内部子系统的变化之间的依赖相互解耦



结构



要点总结

1. 从客户程序的角度看，Facade模式简化了整个组件系统的接口，对于组件内部与组件外部客户程序来说，达到一种"解耦"的效果——内部子系统的任何变化不会影响到Facade接口的变化。
2. Facade设计模式更注重从架构的层次去看整个系统，而不是单个类的层次。Facade很多时候更是一种架构设计模式。
3. Facade设计模式并非一个集装箱，可以任意地放进任何多个对象。Facade模式中组件的内部应该是"相互耦合关系比较大的一系列组件"，而不是一个简单的功能集合。

Proxy

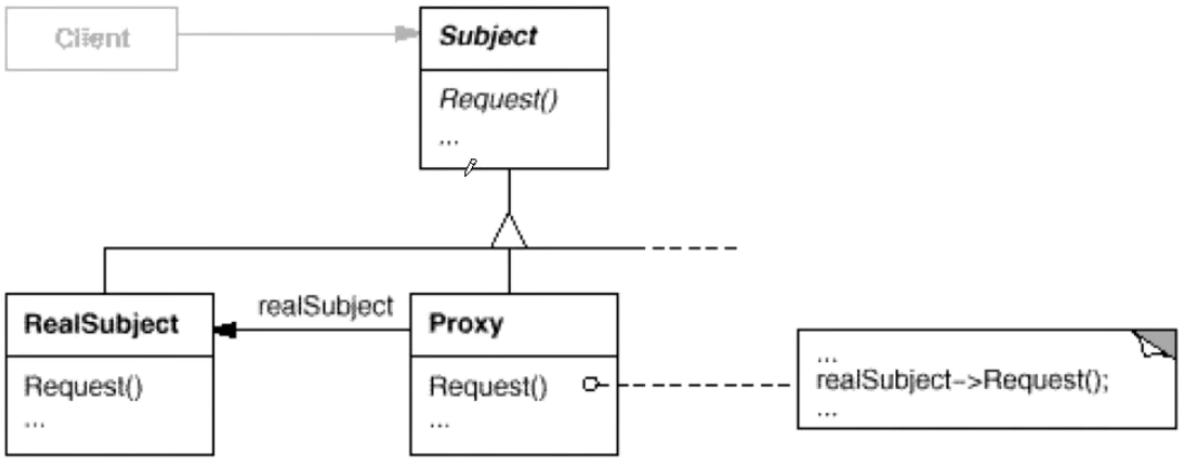
代理模式

> 为其他对象提供一种代理以控制（隔离、使用接口）对这个对象的访问。

解决什么问题

> 在面向对象系统中，有些对象由于某种原因（比如对象创建的开销很大，或者某些操作需要的安全控制，或者需要进程外的访问等），直接访问会给使用者或者系统结构带来很多麻烦。
>
> 如何在不失去透明操作对象的同时来控制/管理这些对象特有的复杂性？增加一层间接层是软件开发中常见的解决方式。

结构



要点总结

1. “增加一层间接层”是软件系统中对许多复杂问题的一种常见解决方法。在面向对象系统中，直接使用某些对象会带来很多问题，作为间接层的proxy对象便是解决这一问题的常用手段。
2. 具体proxy设计模式的实现方法、实现粒度都相差很大，有些可能对单个对象做细粒度的控制，如copy-on-write技术，有些可能对组件模块提供抽象代理层，在架构层次对对象做proxy。
3. Proxy并不一定要求保持接口完整的一致性，只要能够实现间接控制，有时候损及一些透明性是可以接受的。

示例

```
class ISubject {
public:
    virtual void process();
    // ...
};

class RealSubject : public ISubject {
public:
    virtual void process() {
        // ...
    }
};

class ClientApp {
private:
    ISubject* subject;
public:
    ClientApp() {
        // ...
        subject = new RealSubject();// 由于某些原因 我们做不到直接生成RealSubject
        // ...
    }
    void DoWork() {
        // ...
        subject->process();

    }
};
```

```
class ISubject {
public:
    virtual void process();
    // ...
};


class SubjectProxy : public ISubject {
private:
    // RealSubject* ...
public:
    virtual void process() {
        // 对RealSubject的间接访问
        // ...
    }
};

class ClientApp {
private:
    ISubject* subject;
public:
    ClientApp() {
        // ...
        subject = new SubjectProxy();
        // ...
    }
    void DoWork() {
        // ...
        subject->process();

    }
};
```

Adapter

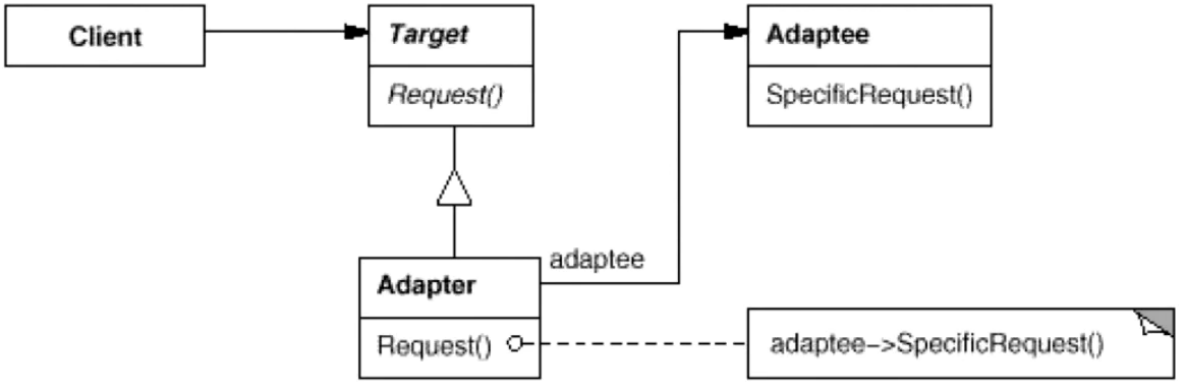
适配器

> 将一个类的接口转化成客户希望的另一个接口。Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

解决什么问题

> 在软件系统中，由于应用环境的变化，常常需要将“一些现存的对象”放在新的环境中应用，但是新环境要求的接口是这些现存对象所不满足的。
>
> 如何应对这种“迁移的变化”？如何既能利用现有对象的良好表现，同时又能满足新的应用环境所要求的的接口？

结构



示例

```
// 目标接口（新接口）
clas ITarget {
public:
    virtual void Process() = 0;
};

// 遗留接口（老接口）
class IAdaptee {
public:
    virtual void foo(int data) = 0;
    virtual int bar() = 0;
};

class OldClass : public IAdaptee {
    // ...
}

class Adapter : public ITarget {
protected:
    IAdaptee* pAdaptee;

public:
    Adapter(IAdaptee* pAdaptee) {
        this->pAdaptee = pAdaptee;
    }

    virtual void Process() {
        // 使用遗留接口
        // ...
        pAdaptee->foo(data);
        // ...
    }
};

int main() {
    IAdaptee* pAdaptee = new OldClass();
    ITarget* pTarget = new Adapter(pAdaptee);
    pTarget->Process();
}
```

Mediator

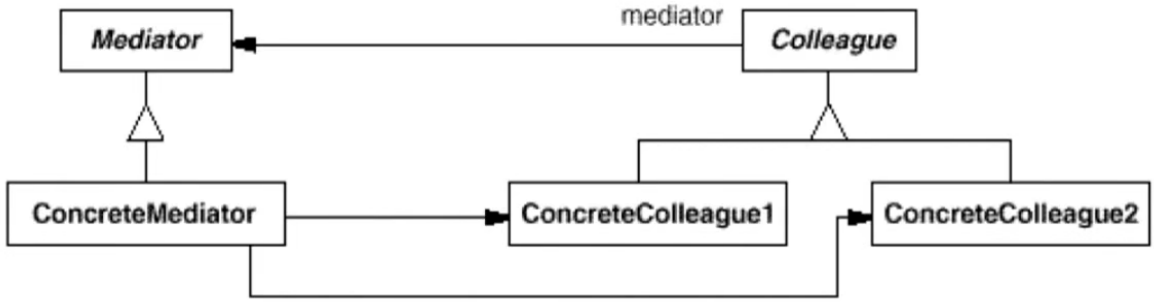
中介者

> 用一个中介对象来封装（封装变化）一系列的对象交互。中介者使各对象不需要显示的相互引用（编译时依赖--->运行时依赖），从而使其耦合松散（管理变化），而且可以独立地改变它们之间的交互。

解决什么问题

> 在软件构建过程中，经常会出现多个对象互相关联交互的情况，对象之间常常会维持一种复杂的引用关系，如果遇到一些需求的更改，这种直接的引用关系将会面临不断地变化。
>
> 在这种情况下，我们可以使用一个“中介对象”来管理对象间的关联关系，避免相互交互的对象之间的紧耦合引用关系，从而更好地抵御变化。

结构



要点总结

1. 将多个对象间复杂的关联关系解耦，Mediator模式将多个对象间的控制逻辑进行集中管理，变“多个对象互相关联”为“多个对象和一个中介者关联”，简化了系统的维护，抵御了可能的变化。
2. 随着控制逻辑的复杂化，Mediator具体对象的实现可能相当复杂。这时候可以对Mediator对象进行分解处理。
3. Facade模式是解耦系统间（单向）的对象关联关系。Mediator模式是结构系统内各个对象之间（双向）的关联关系。

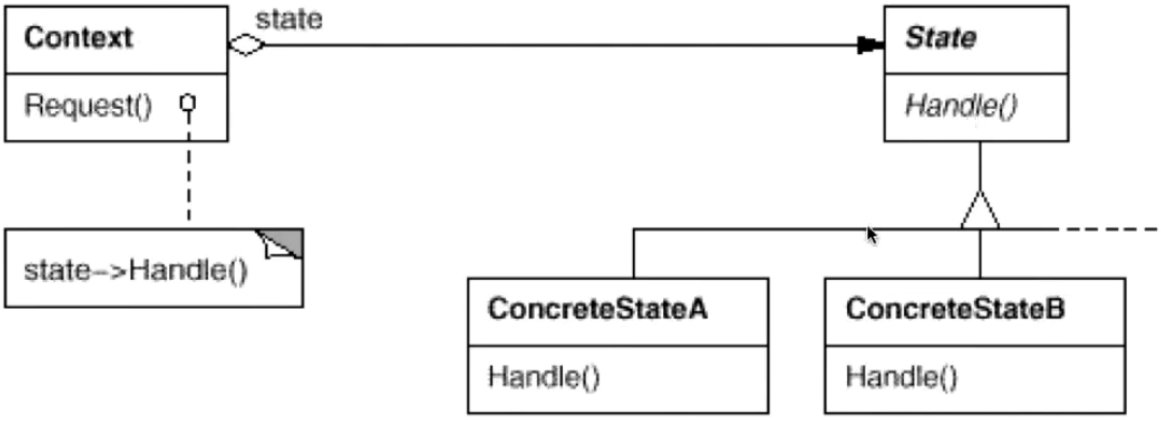
State

状态模式

解决什么问题

> 在软件构建过程中，某些对象的状态如果改变，其行为也会随之而发生改变，比如文档处于只读状态，其支持的行为和读写状态支持的行为就可能完全不同。
>
> 如何在运行时根据对象的状态来透明地更改对象的行为？而不会为对象操作和状态转化之间引入紧耦合。

结构



要点总结

1. State模式将所有与一个特定状态相关的行为都放入一个State的子类对象中，在对象状态切换时，切换相应的对象；但同时维持State接口，这样实现了具体操作与状态转换之间的解耦。
2. 为不同状态引入不同的对象使得状态装换变得更加明确，而且可以保证不会出现状态不一致的情况，因为转换是原子性的——即要么彻底转换过来，要么不转换。
3. 如果State对象没有实例变量，那么各个上下文可以共享同一个State对象，从而节省对象开销。

示例：一个网络程序，根据不同的网络状态产生不同的行为

```
// 最初实现
enum NetworkState {
    Network_Open,
    Network_Close,
    Network_Connect,
    Network_Wait,    // 新添加
};

class NetworkProcessor {
private:
    NetworkState state;
public:
    void Operation1 {
        if (state == Network_Open) {

            // ...
            state = Network_Close;
        }
        else if (state == Network_Close) {

            // ...
            state = Network_Connect;
        }
        else if (state == Network_Connect) {

        }
        // ...
    }

    void Operation2 {
        if (state == Network_Open) {

            // ...
```

```
        }
        else if (state == Network_Close) {

            // ...
        }
        else if (state == Network_Connect) {

        }
        // ...
    }

    void Operation3 {
        // ...
    }
    // ...
};
```

```
// State模式
class NetworkState {
public:
    virtual ~NetworkState() {}

    NetworkState* pNext;
    virtual void Operation1() = 0;
    virtual void Operation2() = 0;
    virtual void Operation3() = 0;
    // ...
};

class OpenState : public NetworkState {
private:
    static NetworkState* m_instance;
public:
    static NetworkState* getInstance() {
        if (m_instance == nullptr) {
            m_instance = new OpenState();
        }
        return m_instance;
    }

    void Operation1 {

        // ...
        pNext = CloseState::getInstance();
    }

    void Operation2 {

        // ...
        pNext = ConnectState::getInstance();
    }

    void Operation3 {
        // ...
        pNext = OpenState::getInstance();
    }
    // ...
```

```
}

class CloseState : public NetworkState {

}

class ConnectState : public NetworkState {

}

class WaitState : public NetworkState {

}

class NetworkProcessor {
private:
    NetworkState* pState;
public:
    NetworkProcessor(NetworkState* pState) : pState(pState) { }

    void Operation1 {

        // ...
        pState->Operation1();
        pState = pState->pNext;
    }

    void Operation2 {

        // ...
        pState->Operation2();
        pState = pState->pNext;
    }

    void Operation3 {
        // ...
        pState->Operation3();
        pState = pState->pNext;
    }
    // ...
};
```

Memento

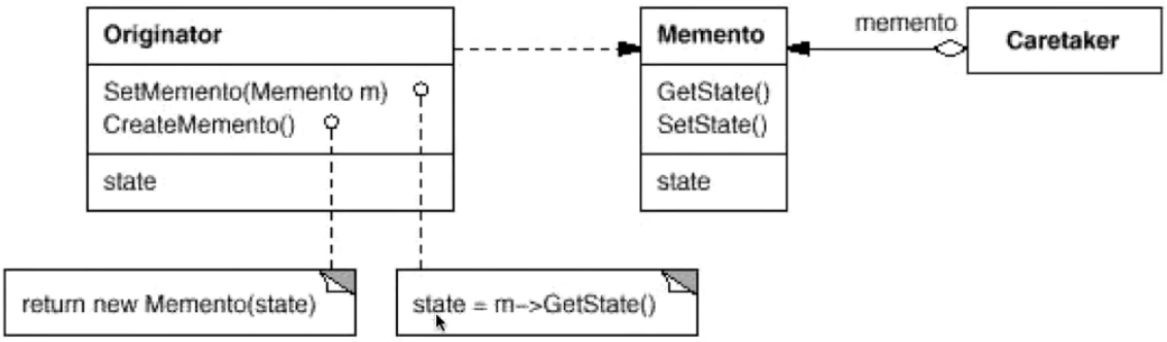
备忘录

> 在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可以将该对象恢复到原先保存的状态。

解决什么问题

> 在软件构建过程中，其某些对象的状态在转换过程中，可能由于某种需要，要求程序能够回溯到对象之前处于某个点时的状态。如果使用一些公有接口来让其他对象得到对象的状态，便会暴露对象的细节实现。
>
> 如何实现对象状态的良好保存与恢复？但同时又不会因此而破坏对象本身的封装性。

结构



要点总结

1. 备忘录（Memento）存储原发器（Originator）对象的内部状态，在需要时恢复原发器状态。
2. Memento模式的核心是信息隐藏，即Originator需要向外界隐藏信息，保持其封装性。但同时又需要将状态保持到外界（Memento）。
3. 由于现代语言运行时（如C#、Java等）都具有相当的对象序列化支持，因此往往采用效率较高、又较容易正确实现的序列化方案来实现Memento模式。

示例

```
class Memento {
private:
    string state;
    // ...
public:
    Memento(const string& s) : state(s) {}
    string getState() const { return state; }
    void setState(const string& s) { state = s; }
};

class Originator {
private:
    string state;
    // ... and
public:
    Originator() {}
    Memento createMemento() {
        Memento m(state);
        return m;
    }
    void setMemento(Memento& m) {
        state = m.getState();
    }
};

int main()
{
    Originator originator;

    // 存储到备忘录
    Memento mem = originator.createMemento();

    // ... 改变originator状态
```

```
    // 从备忘录中恢复
    originator.setMemento(mem);

    // ................
}
```

Composite

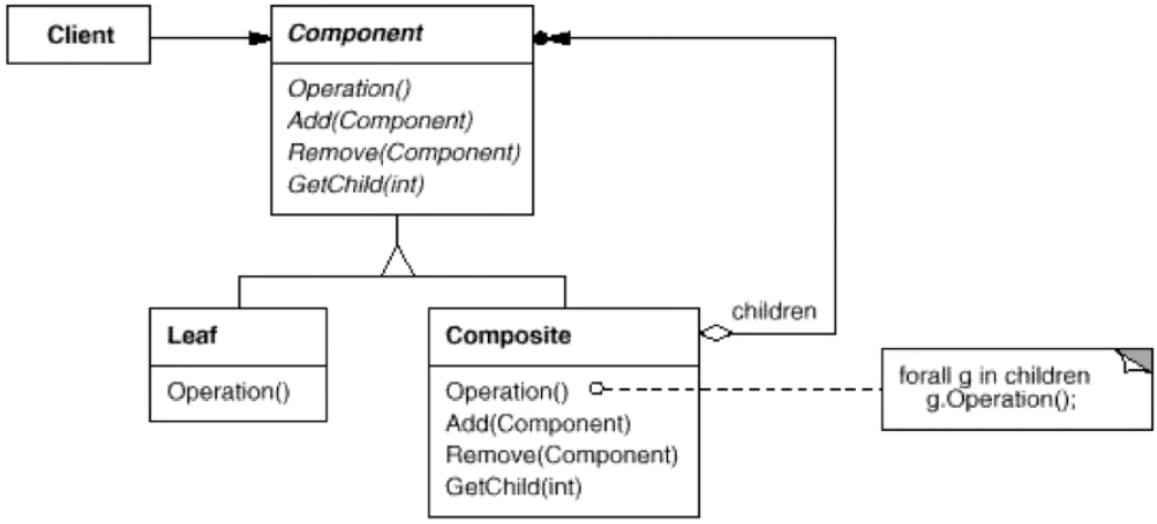
组合模式

> 将对象组成树形结构以表示“部分-整体”的层次结构。Composite使得用户对单个对象和组和对象的使用具有一致性（稳定）。

解决什么问题

> 软件在某些情况下，客户代码过多地依赖于对象容器负载的内部实现结构，对象容器内部实现结构（而非抽象接口）的变化将引起客户代码（invoke函数）的频繁变化，带来了代码的维护性、扩展性等弊端。
>
> 如何将“客户代码与复杂对象容器结构”解耦？让对象容器自己来实现自身的复杂结构，从而使得客户代码就像处理简单对象一样来处理复杂的对象容器。

结构



要点总结

1. Composite模式采用树形结构来实现普遍存在的对象容器，从而将“一对多”的关系转化为“一对一”的关系，使得客户代码可以一致地（复用）处理对象和对象容器，无需关心处理的是单个的对象，还是组合的对象容器。
2. 将“客户代码与复杂的对象容器结构”解耦是Composite的核心思想，解耦之后，客户代码将与纯粹的抽象接口——而非对象容器的内部实现结构——发生依赖，从而更能“应对变化”。
3. Composite模式在具体实现中，可以让父对象中的子对象反向追溯；如果父对象有频繁的遍历需求，可以使用缓存技巧来改善效率。

示例

```
class Component {
public:
    virtual void process() = 0;
    virtual ~Component() {}
};
```

```
class Composite : public Component {
private:
    string name;
    list<Component*> elements;
public:
    Composite(const string& s) : name(s) {}
    void add(Component* element) {
        elements.push_back(element);
    }
    void remove(Component* element) {
        elements.remove(element);
    }

    void process() {
        // process current node

        // process leaf nodes
        for (auto &e : elements) {
            e->process();
        }
    }
};

class Leaf : public Component {
private:
    string name;
public:
    Leaf(string s) : name(s) {}

    void process() {
        // process current node
    }
};

// 稳定
void Invoke(Component& element) {
    // ...
    element.process();
}

int main() {
    Composite root("root");
    Composite treeNode1("treeNode1");
    Composite treeNode2("treeNode2");
    Composite treeNode3("treeNode3");
    Composite treeNode4("treeNode4");
    Leaf left1("left1");
    Leaf left2("left2");

    root.add(&treeNode1);
    treeNode1.add(&treeNode2);
    treeNode2.add(&left1);

    root.add(&treeNode3);
    treeNode3.add(&treeNode4);
    treeNode4.add(&left2);
```

```
    process(root);
}
```

Iterator

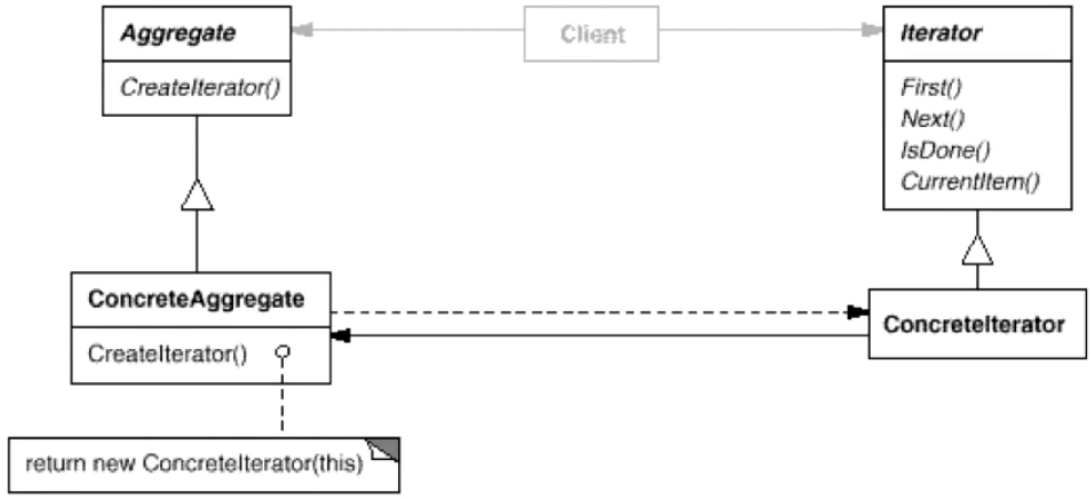
迭代器

> 提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露（稳定）该对象的内部表示。

解决什么问题

> 在软件构建过程中，集合对象内部结构常常变化各异。但对于这些集合对象，我们希望不暴露其内部结构的同时，可以让外部客户嗲吗透明地访问其中包含的元素；同时这种“透明遍历”也为“同一种算法在多种集合对象上进行操作”提供了可能。
>
> 使用面向对象的技术将这种遍历机制抽象为“迭代器对象”为“应对变化中的集合对象”提供了一种优雅的方式。

结构



要点总结

1. 迭代抽象：访问一个聚合对象的内容而无需暴露它的内部表示。
2. 迭代多态：为遍历不同的集合结构提供一个统一的接口，从而支持同样的算法在不同集合结构上进行操作。
3. 迭代器的健壮性考虑：遍历的同时更改迭代器所在的集合结构，会导致问题。

示例

```
template<typename T>
class Iterator {
public:
    virtual void first() = 0;
    virtual void next() = 0;
    virtual bool isDone() const = 0;
    virtual T& current() = 0;
};

template<typename T>
class MyCollection {
public:
```

```
    Iterator<T> GetIterator() {
        // ...
    }

};

template<typename T>
class CollectionIterator : public Iterator<T> {
private:
    MyCollection<T> mc;
public:
    CollectionIterator(const MyCollection<T>& c) : mc(c) { }

    void first() override {

    }
    void next() override {

    }
    bool isDone() const override {

    }
    T& current() override {

    }
}

void MyAlgorithm() {
    MyCollection<int> mc;

    Iterator<int> iter = mc.GetIterator();

    for (iter.first(); !iter.isDone(); iter.next()) {
        // ...
    }
}
```

Chain of Resposibility

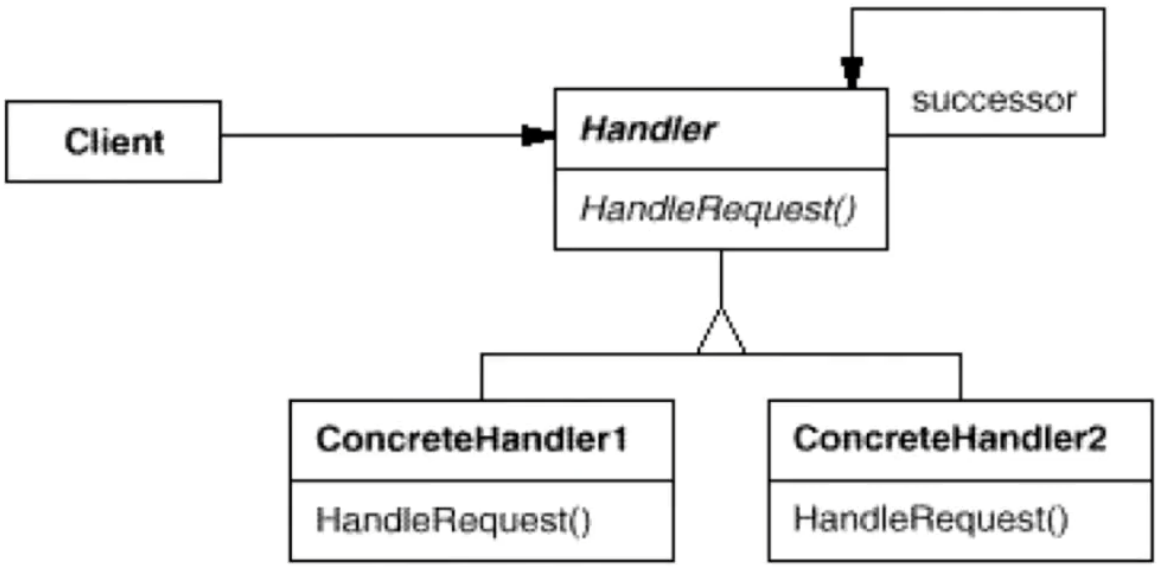
职责链

> 使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递请求，直到有一个对象处理它为止。

解决什么问题

> 在软件构建过程中，一个请求可能被多个对象处理，但是每个请求在运行时只能有一个接受者，如果显示指定，将必不可少地带来请求发送者与接受着的紧耦合。
>
> 如何使请求的发送者不需要指定具体的接受者？让请求的接受者自己在运行时决定来处理请求，从而使两者解耦。

结构



要点总结

1. Chain of Responsibility 模式的应用场合在于“一个请求可能有多个接受者，但最后真正的接受者只有一个”，这时候请求发送者与接受者的耦合有可能出现“变化脆弱”的症状，职责链的目的就是将二者解耦，从而更好地应对变化。
2. 应用了Chain of Responsibility模式后，对象的职责分派将更具灵活性。我们可以在运行时动态添加/修改请求的处理职责。
3. 如果请求传递到职责链的末尾仍得不到处理，应该有一个合理的缺省机制。这也是每一个接受对象的责任，而不是发出请求的对象的责任。

示例

```
class Request {
private:
    string description;
    RequestType reqType;
public:
    Request(const string& desc, RequestType type) : description(desc),
reqType(type) {}
    RequestType getReqType() const { return reqType; }
    const string& getDescription() const { return description; }
};

class ChainHandler {
private:
    ChainHandler* nextChain;
    void sendRequestToNextHandler(const Request& req) {
        if (nextChain != nullptr) {
            nextChain->handle(req);
        }
    }
protected:
    virtual bool canHandleRequest(const Request& req) = 0;
    virtual void processRequest(const Request& req) = 0;
public:
    virtual ~ChainHandler() {}
```

```
    ChainHandler() {
        nextChain = nullptr;
    }
    void setNextChain(ChainHandler* next) {
        nextChain = next;
    }
    void handle(const Request& req) {
        if (canHandleRequest(req)) {
            processRequest(req);
        }
        else {
            sendRequestToNextHandler(req);
        }
    }
};

class Handler1 : public ChainHandler {
protected:
    bool canHandleRequest(const Request& req) override {
        return req.getReqType() == RequestType::REQ_HANDLER1;
    }
    void processRequest(const Request& req) override {
        // ...
    }
};

class Handler2 : public ChainHandler {
protected:
    bool canHandleRequest(const Request& req) override {
        return req.getReqType() == RequestType::REQ_HANDLER2;
    }
    void processRequest(const Request& req) override {
        // ...
    }
};

class Handler3 : public ChainHandler {
protected:
    bool canHandleRequest(const Request& req) override {
        return req.getReqType() == RequestType::REQ_HANDLER3;
    }
    void processRequest(const Request& req) override {
        // ...
    }
};

int main() {
    Handler1 h1;
    Handler2 h2;
    Handler3 h3;

    h1.setNextChain(&h2);
    h2.setNextChain(&h3);

    Request request("process task ...", RequestType::REQ_HANDLER3);
    h1.handle(request);
    return 0;
}
```

Command

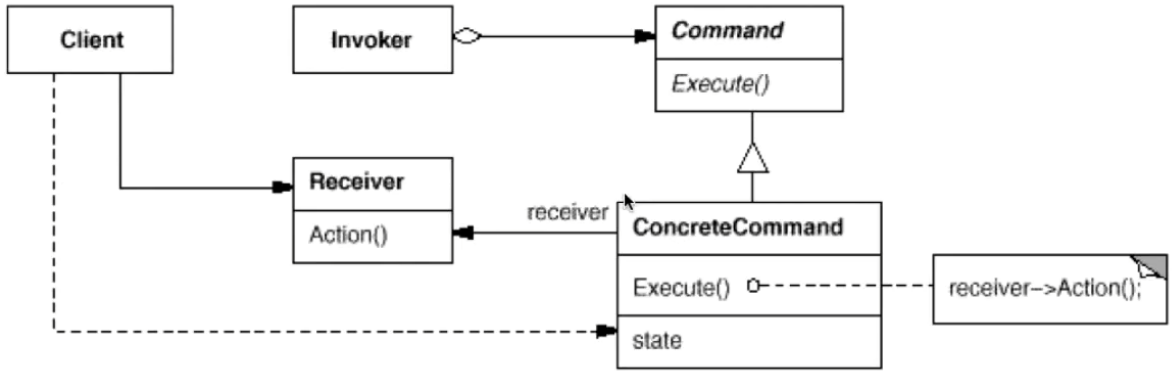
命令模式

> 将一个请求（行为）封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作。

解决什么问题

> 在软件构建过程中，“行为请求者”与“行为实现者”通常呈现一种“紧耦合”。但在某些场合——比如需要对行为进行”记录、撤销/重做（undo/redo）、事务“等处理，这种无法抵御变化的紧耦合是不合适的。
>
> 在这种情况下，如何将“行为请求者”与"行为实现者"解耦？将一组行为抽象为对象，可以实现二者之间的松耦合。

结构



要点总结

1. Command模式的根本目的在于将“行为请求者”与“行为实现者”解耦，在面向对象语言中，常见的实现手段是“将行为抽象为对象”。
2. 实现Command接口的具体命令对象ConcreteCommand有时候根据需要可能会保存一些额外的状态信息。通过使用Composite模式，可以将多个“命令”封装为一个“复合命令”MacroCommand。
3. Command模式与C++中的函数对象有些类似。但两者定义行为接口的规范有所区别：Command以面向对象中的“接口-实现”来定义行为接口规范，更严格，但有性能损失；C++函数对象以函数签名来定义行为接口规范，更灵活，性能更高。

示例

```
class Command {
public:
    virtual void execute() = 0;
};

class ConcreteCommand1 : public Command {
private:
    string arg;
public:
    ConcreteCommand1(const string& a) : arg(a) {}
    void execute() override {
        // ...
    }
};
```

```
class ConcreteCommand2 : public Command {
private:
    string arg;
public:
    ConcreteCommand2(const string& a) : arg(a) {}
    void execute() override {
        // ...
    }
};

class MacroCommand : public Command {
private:
    vector<Command*> commands;
public:
    void addCommand(Command* c) { commands.push_back(c); }
    void execute() override {
        for (auto& c : commands) {
            c->execute();
        }
    }
};

void process() {
    ConcreteCommand1 command1(receiver, "...");
    ConcreteCommand2 command2(receiver, "...");

    MacroCommand macro;
    macro.addCommand(&command1);
    macro.addCommand(&command2);

    macro.execute();
}
```

Visitor

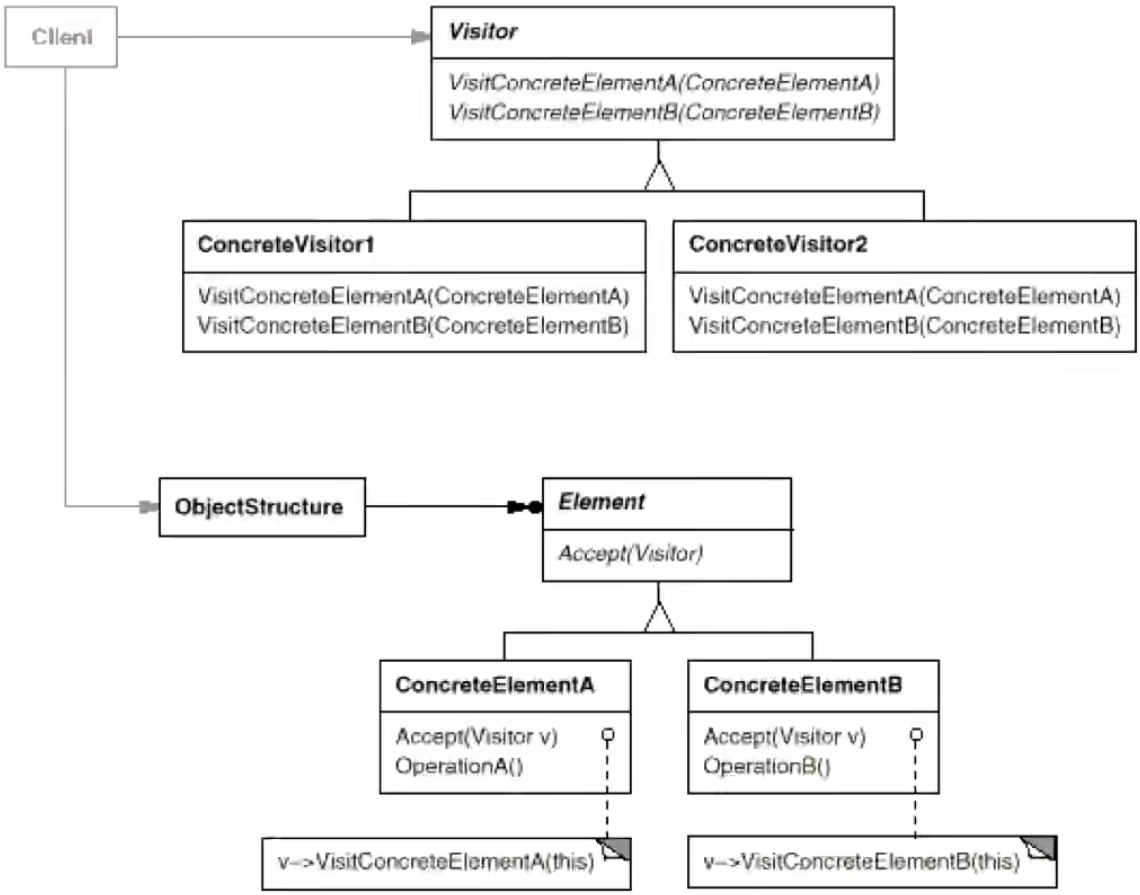
访问器

> 表示一个作用于某对象结构中的各元素的操作。使得可以在不改变（稳定）各元素的类的前提下定义（扩展）作用于这些元素的新操作（变化）。

解决什么问题

> 在软件构建过程中，由于需求的改变，某些类层次结构中常常需要增加新的行为（方法），如果直接在基类中做这样的更改，将会给子类带来很繁重的变更负担，甚至破坏原有的设计。
>
> 如何在不改变类层次结构的前提下，在运行时根据需要透明地为类层次结构上的各个类动态添加新的操作，从而避免上述问题。

结构



要点总结

1. Visitor模式通过所谓的双重分发（double dispatch）来实现在不更改（不添加新的操作-编译时）Element类层次结构的前提下，在运行时透明地为类层次结构上的各个类动态添加新的操作（支持变化）。
2. 所谓双重分发即Visitor模式中间包括了两个多态分发（注意其中的多态机制）：第一个为accepte方法的多态辨析；第二个为visitElementX方法的多态辨析。
3. Visitor模式的最大缺点在于扩展类层次结构（添加新的Element子类），会导致Visitor类的改变。因此Visitor模式适用于“**Element类层次结构稳定**”，而其中的操作却经常面临频繁改动。

示例

```
class Element {
public:
    virtual ~Element() {}

    virtual void Func1() = 0;
    virtual void Func2(int data) = 0;
};

class ElementA : public Element {
public:
    void Func1() override {
        // ...
    }

    void Func2(int data) override {
        // ...
    }
};
```

```
class ElementB : public Element {
public:
    void Func1() override {
        // ...
    }

    void Func2(int data) override {
        // ...
    }
};

// 需要添加一个新功能 Func2
// 在基类添加接口 并在各个子类添加对应接口的实现
```

```
// 采用Visitor模式
// 稳定
class Visitor {
public:
    virtual ~Visitor() {}

    virtual void visitElementA(ElementA& element) = 0;
    virtual void visitElementB(ElementB& element) = 0;
};

class Element {
public:
    virtual ~Element() {}

    virtual void accept(Visitor& visitor) = 0;
};

class ElementA : public Element {
public:
    void accept(Visitor& visitor) override {
        visitor.visitElementA(*this);
    }
};

class ElementB : public Element {
public:
    void accept(Visitor& visitor) override {
        visitor.visitElementB(*this);
    }
};

// ================================================================
// 扩展 变化
class Visitor1 : public Visitor {
public:
    void visitElementA(ElementA& element) override {
        // ...
    }

    void visitElementB(ElementB& element) override {
        // ...
    }
```

```
}

// 需要对Element添加新功能 Func2时
class Visitor2 : public Visitor {
public:
    void visitElementA(ElementA& element) override {
        // ...
    }

    void visitElementB(ElementB& element) override {
        // ...
    }
}

void Process() {
    Visitor2 visitor;
    ElementB element;
    element.accept(visitor);// 相当于elementB.Func2()

    return 0;
}
```

Interpreter

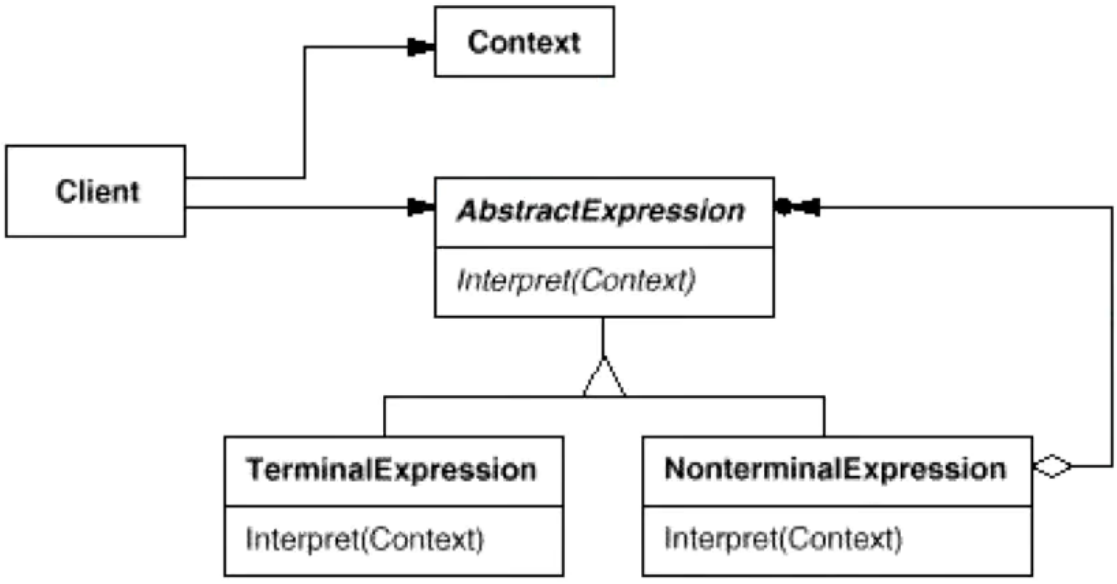
解析器

> 给定一个语言，定义它的文法的一种表示，并定义一种解释器，这个解释器使用该表示来解释语言中的句子。

解决什么问题

> 在软件构建过程中，如果某一特定领域的问题比较复杂，类似的结构不断重复出现，如果使用普通的编程方式来实现将面临非常频繁的变化。
>
> 在这种情况下，将特定领域的问题表达为某种语法规则下的句子，然后构建一个解释器来解释这样的句子，从而达到解决问题的目的。

结构



要点总结

1. Interpreter模式的应用场合是Interpreter模式应用中的难点，只有满足"业务规则频繁变化，且类似的结构不断重复出现，并且容易抽象为语法规则的问题"才适合使用Interpreter模式。
2. 使用Interpreter模式来表示文法规则，从而可以使用面向对象技巧来方便地"扩展"文法。
3. Interpreter模式比较适合简单的文法表示，对于复杂的文法表示，Interpreter模式会产生比较大的类层次结构，需要求求助于语法分析生成器这样的标准工具。

示例

```
class Expression {
public:
    virtual int interpreter(map<char, int> var) = 0;
    virtual ~Expression() {}
};

// 变量表达式
class VarExpression : public Expression {
private:
    char key;
public:
    VarExpression(const char& key) {
        this->key = key;
    }

    int interpreter(map<char, int> var) override {
        return var[key];
    }
};

// 符号表达式
class SymbolExpression : public Expression {
protected:
    Expression* left;
    Expression* right;
public:
    SymbolExpression(Expression* left, Expression* right) : left(left),
right(right) {

    }
};

// 加法运算
class AddExpression : public SymbolExpression {
public:
    AddExpression(Expression* left, Expression* right) : SymbolExpression(left,
right) {

    }

    int interpreter(map<char, int> var) override {
        return left->interpreter(var) + right->interpreter(var);
    }
};

// 减法运算
class SubExpression : public SymbolExpression {
```

```
public:
    SubExpression(Expression* left, Expression* right) : SymbolExpression(left,
right) {

    }

    int interpreter(map<char, int> var) override {
        return left->interpreter(var) - right->interpreter(var);
    }
};

Expression* analyse(string expStr) {
    stack<Expression*> expStack;
    Expression* left = nullptr;
    Expression* right = nullptr;

    for (int i = 0; i < expStr.length(); i++) {
        switch (expStr[i]) {
            case '+':
                // 加法
                left = expStack.top();
                right = new VarExpression(expStr[++i]);
                expStack.push(new AddExpression(left, right));
                break;
            case '-':
                // 减法
                left = expStack.top();
                right = new VarExpression(expStr[++i]);
                expStack.push(new SubExpression(left, right));
                break;
            default:
                expStack.push(new VarExpression(expStr[i]));
        }
    }

    Expression* expression = expStack.top();
    return expression;
}

void release(Expression* expression) {
    // 释放树节点内存

}

void process() {
    string expStr = "a+b-c+d";
    map<char, int> var;
    var['a'] = 5;
    var['b'] = 2;
    var['c'] = 1;
    var['d'] = 6;

    Expression* expression = analyse(expStr);

    int result = expression->interpreter(var);

    cout << result << endl;
}
```