

行为型模式

行为型模式(Behavioral Pattern)是对在不同的对象之间划分责任和算法的抽象化。

行为型模式不仅仅关注类和对象的结构，而且重点关注它们之间的相互作用。

通过行为型模式，可以更加清晰地划分类与对象的职责，并研究系统在运行时实例对象 之间的交互。在系统运行时，对象并不是孤立的，它们可以通过相互通信与协作完成某些复杂功能，一个对象在运行时也将影响到其他对象的运行。

行为型模式分为类行为型模式和对象行为型模式两种：

- 类行为型模式：类的行为型模式使用继承关系在几个类之间分配行为，类行为型模式主要通过多态等方式来分配父类与子类的职责。
- 对象行为型模式：对象的行为型模式则使用对象的聚合关联关系来分配行为，对象行为型模式主要是通过对象关联等方式来分配两个或多个类的职责。根据“合成复用原则”，系统中要尽量使用关联关系来取代继承关系，因此大部分行为型设计模式都属于对象行为型设计模式。

包含模式

- 职责链模式(Chain of Responsibility)
重要程度：3
- 命令模式(Command)
重要程度：4
- 解释器模式(Interpreter)
重要程度：1
- 迭代器模式(Iterator)
重要程度：5
- 中介者模式(Mediator)
重要程度：2
- 备忘录模式(Memento)
重要程度：2
- 观察者模式(Observer)

重要程度：5

- 状态模式(State)

重要程度：3

- 策略模式(Strategy)

重要程度：4

- 模板方法模式(Template Method)

重要程度：3

- 访问者模式(Visitor)

重要程度：1

目录

- 1. 命令模式

- 1.1. 模式动机
- 1.2. 模式定义
- 1.3. 模式结构
- 1.4. 时序图
- 1.5. 代码分析
- 1.6. 模式分析
- 1.7. 实例
- 1.8. 优点
- 1.9. 缺点
- 1.10. 适用环境
- 1.11. 模式应用
- 1.12. 模式扩展
- 1.13. 总结

- 2. 中介者模式

- 2.1. 模式动机
- 2.2. 模式定义
- 2.3. 模式结构
- 2.4. 时序图
- 2.5. 代码分析
- 2.6. 模式分析
- 2.7. 实例
- 2.8. 优点
- 2.9. 缺点
- 2.10. 适用环境
- 2.11. 模式应用
- 2.12. 模式扩展
- 2.13. 总结

- 3. 观察者模式

- 3.1. 模式动机
- 3.2. 模式定义
- 3.3. 模式结构
- 3.4. 时序图
- 3.5. 代码分析
- 3.6. 模式分析
- 3.7. 实例
- 3.8. 优点
- 3.9. 缺点
- 3.10. 适用环境
- 3.11. 模式应用
- 3.12. 模式扩展
- 3.13. 总结
- 4. 状态模式
 - 4.1. 模式动机
 - 4.2. 模式定义
 - 4.3. 模式结构
 - 4.4. 时序图
 - 4.5. 代码分析
 - 4.6. 模式分析
 - 4.7. 实例
 - 4.8. 优点
 - 4.9. 缺点
 - 4.10. 适用环境
 - 4.11. 模式应用
 - 4.12. 模式扩展
 - 4.13. 总结
- 5. 策略模式
 - 5.1. 模式动机
 - 5.2. 模式定义
 - 5.3. 模式结构
 - 5.4. 时序图
 - 5.5. 代码分析
 - 5.6. 模式分析
 - 5.7. 实例
 - 5.8. 优点
 - 5.9. 缺点
 - 5.10. 适用环境
 - 5.11. 模式应用
 - 5.12. 模式扩展
 - 5.13. 总结

1. 命令模式

目录

- 命令模式
 - 模式动机
 - 模式定义
 - 模式结构
 - 时序图
 - 代码分析
 - 模式分析
 - 实例
 - 优点
 - 缺点
 - 适用环境
 - 模式应用
 - 模式扩展
 - 总结

1.1. 模式动机

在软件设计中，我们经常需要向某些对象发送请求，但是并不知道请求的接收者是谁，也不知道被请求的操作是哪个，我们只需在程序运行时指定具体的请求接收者即可，此时，可以使用命令模式来进行设计，使得请求发送者与请求接收者消除彼此之间的耦合，让对象之间的调用关系更加灵活。

命令模式可以对发送者和接收者完全解耦，发送者与接收者之间没有直接引用关系，发送请求的对象只需要知道如何发送请求，而不必知道如何完成请求。这就是命令模式的模式动机。

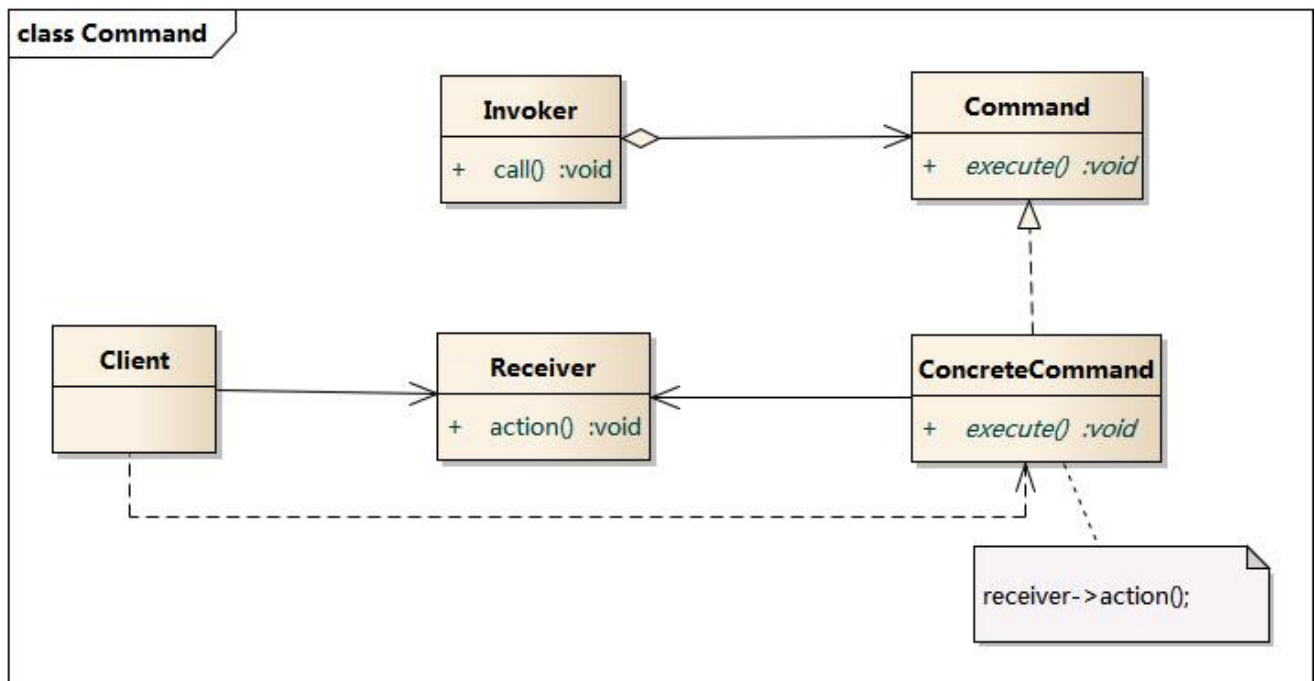
1.2. 模式定义

命令模式(Command Pattern)：将一个请求封装为一个对象，从而使我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。命令模式是一种对象行为型模式，其别名为动作(Action)模式或事务(Transaction)模式。

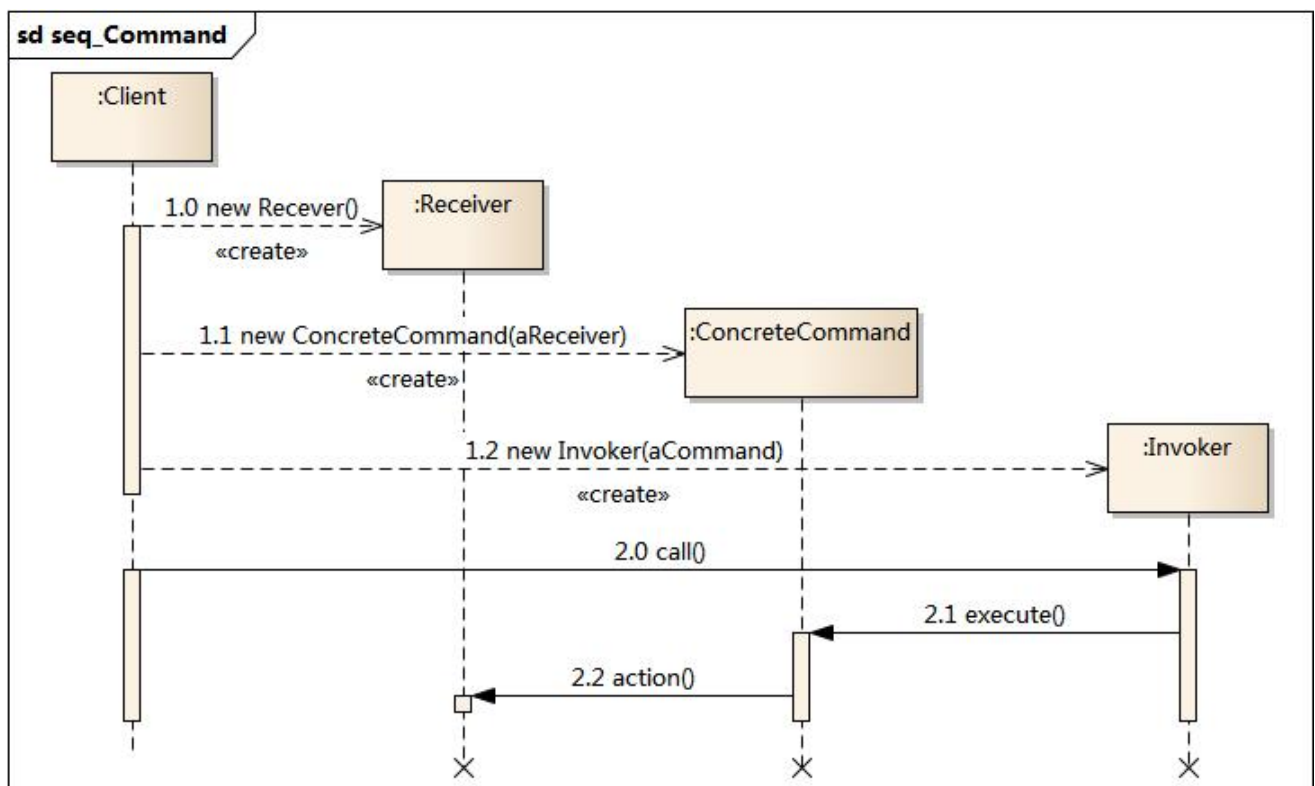
1.3. 模式结构

命令模式包含如下角色：

- Command: 抽象命令类
- ConcreteCommand: 具体命令类
- Invoker: 调用者
- Receiver: 接收者
- Client: 客户类



1.4. 时序图



1.5. 代码分析

```

1  #include <iostream>
2  #include "ConcreteCommand.h"
3  #include "Invoker.h"
4  #include "Receiver.h"
5
6  using namespace std;
7
8  int main(int argc, char *argv[])
9  {
10     Receiver * pReceiver = new Receiver();
11     ConcreteCommand * pCommand = new ConcreteCommand(pReceiver);
12     Invoker * pInvoker = new Invoker(pCommand);
13     pInvoker->call();
14
15     delete pReceiver;
16     delete pCommand;
17     delete pInvoker;
18     return 0;
19 }

```

```

1  //////////////////////////////////////
2  // Receiver.h
3  // Implementation of the Class Receiver
4  // Created on:      07-十月-2014 17:44:02
5  // Original author: colin
6  //////////////////////////////////////
7
8  #if !defined(EA_8E5430BB_0904_4a7d_9A3B_7169586237C8__INCLUDED_)
9  #define EA_8E5430BB_0904_4a7d_9A3B_7169586237C8__INCLUDED_
10
11  class Receiver
12  {
13  public:
14     Receiver();
15     virtual ~Receiver();
16
17     void action();
18
19 };
20 #endif // !defined(EA_8E5430BB_0904_4a7d_9A3B_7169586237C8__INCLUDED_)
21

```

```

1  //////////////////////////////////////
2  // Receiver.cpp
3  // Implementation of the Class Receiver
4  // Created on:      07-十月-2014 17:44:02
5  // Original author: colin
6  //////////////////////////////////////
7  #include "Receiver.h"
8  #include <iostream>
9  using namespace std;
10 Receiver::Receiver(){
11 }
12
13 Receiver::~Receiver(){
14 }
15
16 void Receiver::action(){
17     cout << "receiver action." << endl;
18 }
19
20
21
22

```

```

1  //////////////////////////////////////
2  // ConcreteCommand.h
3  // Implementation of the Class ConcreteCommand
4  // Created on:      07-十月-2014 17:44:01
5  // Original author: colin
6  //////////////////////////////////////
7  #if !defined(EA_1AE70D53_4868_4e81_A1B8_1088DA355C23__INCLUDED_)
8  #define EA_1AE70D53_4868_4e81_A1B8_1088DA355C23__INCLUDED_
9
10 #include "Command.h"
11 #include "Receiver.h"
12
13 class ConcreteCommand : public Command
14 {
15 public:
16     ConcreteCommand(Receiver * pReceiver);
17     virtual ~ConcreteCommand();
18     virtual void execute();
19 private:
20     Receiver *m_pReceiver;
21
22 };
23 #endif // !defined(EA_1AE70D53_4868_4e81_A1B8_1088DA355C23__INCLUDED_)
24
25
26
27

```

```

1  //////////////////////////////////////
2  //  ConcreteCommand.cpp
3  //  Implementation of the Class ConcreteCommand
4  //  Created on:      07-十月-2014 17:44:02
5  //  Original author: colin
6  //////////////////////////////////////
7  #include "ConcreteCommand.h"
8  #include <iostream>
9  using namespace std;
10
11 ConcreteCommand::ConcreteCommand(Receiver *pReceiver){
12     m_pReceiver = pReceiver;
13 }
14
15 ConcreteCommand::~ConcreteCommand(){
16 }
17
18 void ConcreteCommand::execute(){
19     cout << "ConcreteCommand::execute" << endl;
20     m_pReceiver->action();
21 }
22
23
24
25
26

```

```

1  //////////////////////////////////////
2  //  Invoker.h
3  //  Implementation of the Class Invoker
4  //  Created on:      07-十月-2014 17:44:02
5  //  Original author: colin
6  //////////////////////////////////////
7  #if !defined(EA_3DACB62A_0813_4d11_8A82_10BF1FB00D9A__INCLUDED_)
8  #define EA_3DACB62A_0813_4d11_8A82_10BF1FB00D9A__INCLUDED_
9
10 #include "Command.h"
11
12 class Invoker
13 {
14 public:
15     Invoker(Command * pCommand);
16     virtual ~Invoker();
17     void call();
18
19 private:
20     Command *m_pCommand;
21 };
22 #endif // !defined(EA_3DACB62A_0813_4d11_8A82_10BF1FB00D9A__INCLUDED_)
23
24
25
26

```

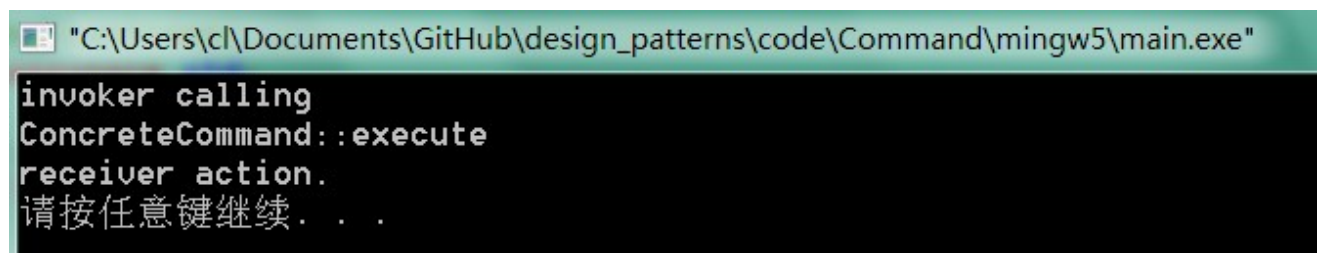


```

1  //////////////////////////////////////
2  //  Invoker.cpp
3  //  Implementation of the Class Invoker
4  //  Created on:      07-十月-2014 17:44:02
5  //  Original author: colin
6  //////////////////////////////////////
7  #include "Invoker.h"
8  #include <iostream>
9  using namespace std;
10
11  Invoker::Invoker(Command * pCommand){
12      m_pCommand = pCommand;
13  }
14
15  Invoker::~Invoker(){
16  }
17
18  void Invoker::call(){
19      cout << "invoker calling" << endl;
20      m_pCommand->execute();
21  }
22
23

```

运行结果：



```

C:\Users\c\Documents\GitHub\design_patterns\code\Command\mingw5\main.exe
invoker calling
ConcreteCommand::execute
receiver action.
请按任意键继续...

```

1.6. 模式分析

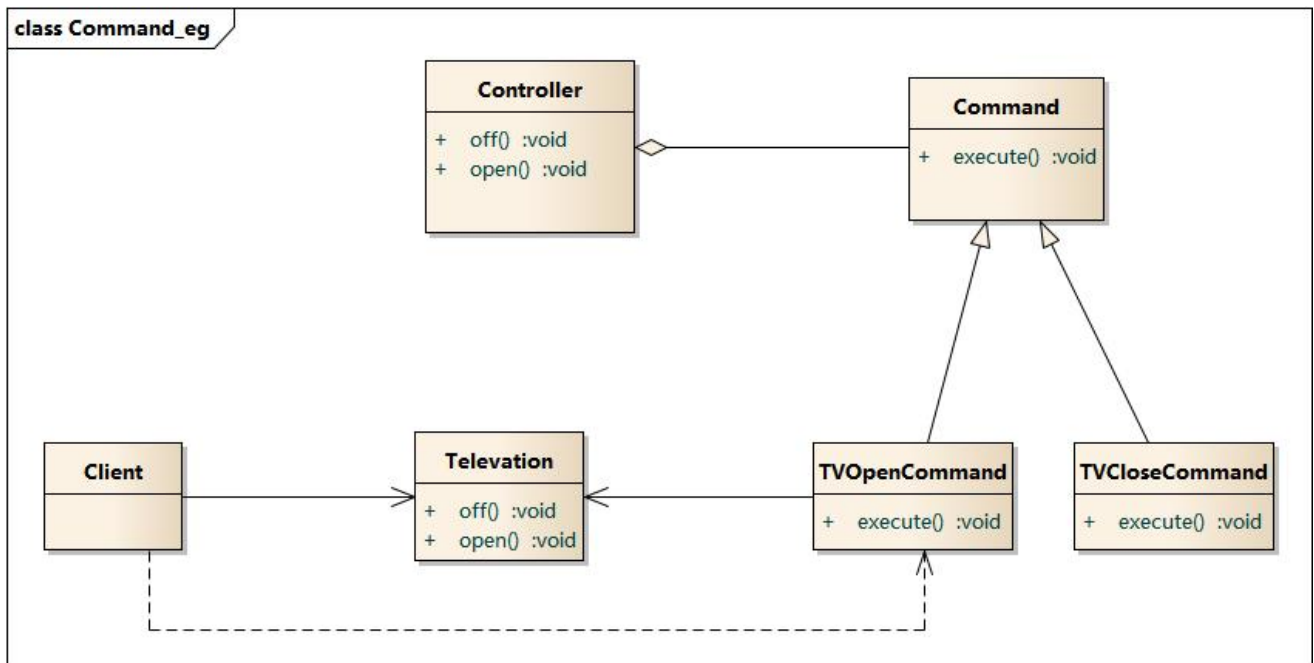
命令模式的本质是对命令进行封装，将发出命令的责任和执行命令的责任分割开。

- 每一个命令都是一个操作：请求的一方发出请求，要求执行一个操作；接收的一方收到请求，并执行操作。
- 命令模式允许请求的一方和接收的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否被执行、何时被执行，以及是怎么被执行的。
- 命令模式使请求本身成为一个对象，这个对象和其他对象一样可以被存储和传递。
- 命令模式的关键在于引入了抽象命令接口，且发送者针对抽象命令接口编程，只有实现了抽象命令接口的具体命令才能与接收者相关联。

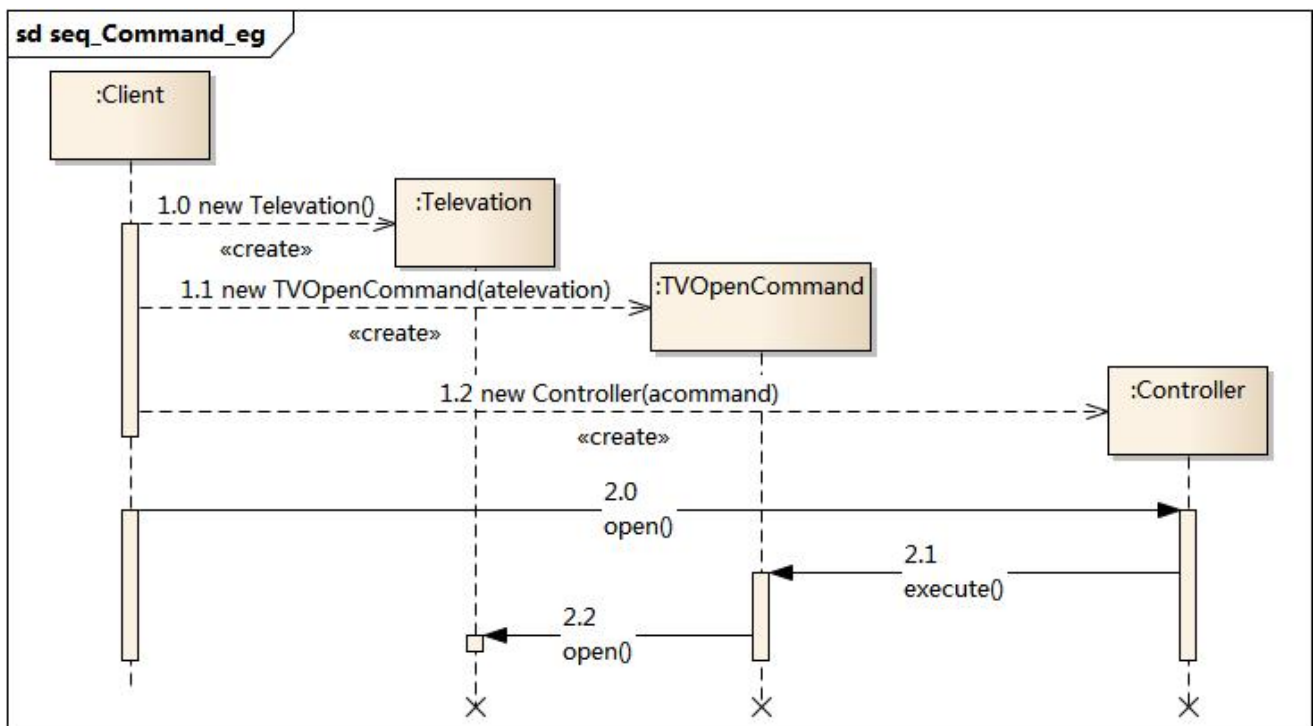
1.7. 实例

实例一：电视机遥控器

- 电视机是请求的接收者，遥控器是请求的发送者，遥控器上有一些按钮，不同的按钮对应电视机的不同操作。抽象命令角色由一个命令接口来扮演，有三个具体的命令类实现了抽象命令接口，这三个具体命令类分别代表三种操作：打开电视机、关闭电视机和切换频道。显然，电视机遥控器就是一个典型的命令模式应用实例。



时序图:



1.8. 优点

命令模式的优点

- 降低系统的耦合度。
- 新的命令可以很容易地加入到系统中。
- 可以比较容易地设计一个命令队列和宏命令（组合命令）。

- 可以方便地实现对请求的Undo和Redo。

1.9. 缺点

命令模式的缺点

- 使用命令模式可能会导致某些系统有过多的具体命令类。因为针对每一个命令都需要设计一个具体命令类，因此某些系统可能需要大量具体命令类，这将影响命令模式的使用。

1.10. 适用环境

在以下情况下可以使用命令模式：

- 系统需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互。
- 系统需要在不同的时间指定请求、将请求排队和执行请求。
- 系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作。
- 系统需要将一组操作组合在一起，即支持宏命令

1.11. 模式应用

很多系统都提供了宏命令功能，如UNIX平台下的Shell编程，可以将多条命令封装在一个命令对象中，只需要一条简单的命令即可执行一个命令序列，这也是命令模式的应用实例之一。

1.12. 模式扩展

宏命令又称为组合命令，它是命令模式和组合模式联用的产物。

-宏命令也是一个具体命令，不过它包含了对其他命令对象的引用，在调用宏命令的execute()方法时，将递归调用它所包含的每个成员命令的execute()方法，一个宏命令的成员对象可以是简单命令，还可以继续是宏命令。执行一个宏命令将执行多个具体命令，从而实现对命令的批处理。

1.13. 总结

- 在命令模式中，将一个请求封装为一个对象，从而使我们可以用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。命令模式是一种对象行为型模式，其别名为动作模式或事务模式。
- 命令模式包含四个角色：抽象命令类中声明了用于执行请求的execute()等方法，通过这些方法可以调用请求接收者的相关操作；具体命令类是抽象命令类的子类，实现了在抽象命令类中声明的方法，它对应具体的接收者对象，将接收者对象的动作绑定其中；调用者即请求的发送者，又称为请求者，它通过命令对象来执行请求；接收者执行与请求相关的操作，它具体实现对请求的业务处理。

- 命令模式的本质是对命令进行封装，将发出命令的责任和执行命令的责任分割开。命令模式使请求本身成为一个对象，这个对象和其他对象一样可以被存储和传递。
- 命令模式的主要优点在于降低系统的耦合度，增加新的命令很方便，而且可以比较容易地设计一个命令队列和宏命令，并方便地实现对请求的撤销和恢复；其主要缺点在于可能会导致某些系统有过多的具体命令类。
- 命令模式适用情况包括：需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互；需要在不同的时间指定请求、将请求排队和执行请求；需要支持命令的撤销操作和恢复操作，需要将一组操作组合在一起，即支持宏命令。

2. 中介者模式

目录

- 中介者模式
 - 模式动机
 - 模式定义
 - 模式结构
 - 时序图
 - 代码分析
 - 模式分析
 - 实例
 - 优点
 - 缺点
 - 适用环境
 - 模式应用
 - 模式扩展
 - 总结

2.1. 模式动机

- 在用户与用户直接聊天的设计方案中，用户对象之间存在很强的关联性，将导致系统出现如下问题：
- 系统结构复杂：对象之间存在大量的相互关联和调用，若有一个对象发生变化，则需要跟踪和该对象关联的其他所有对象，并进行适当处理。
- 对象可重用性差：由于一个对象和其他对象具有很强的关联，若没有其他对象的支持，一个对象很难被另一个系统或模块重用，这些对象表现出来更像一个不可分割的整体，职责较为混乱。
- 系统扩展性低：增加一个新的对象需要在原有相关对象上增加引用，增加新的引用关系也需要调整原有对象，系统耦合度很高，对象操作很不灵活，扩展性差。
- 在面向对象的软件设计与开发过程中，根据“单一职责原则”，我们应该尽量将对象细化，使其只负责或呈现单一的职责。
- 对于一个模块，可能由很多对象构成，而且这些对象之间可能存在相互的引用，为了减少对象两两之间复杂的引用关系，使之成为一个松耦合的系统，我们需要使用中介者模式，这就是中介者模式的模式动机。

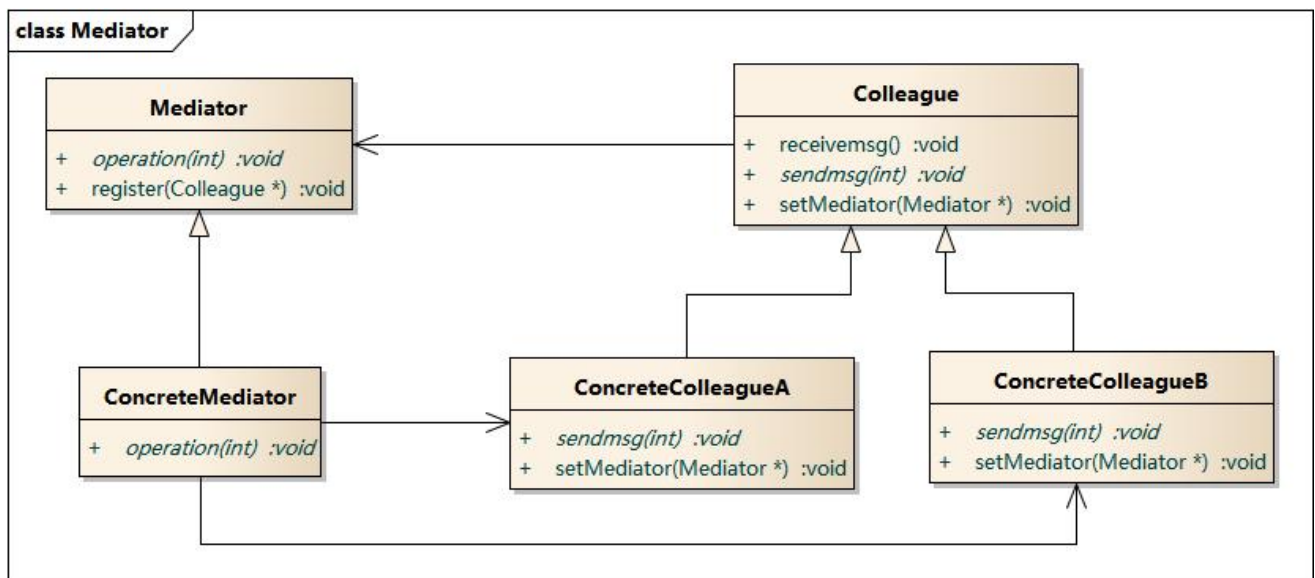
2.2. 模式定义

中介者模式(Mediator Pattern)定义：用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。中介者模式又称为调停者模式，它是一种对象行为型模式。

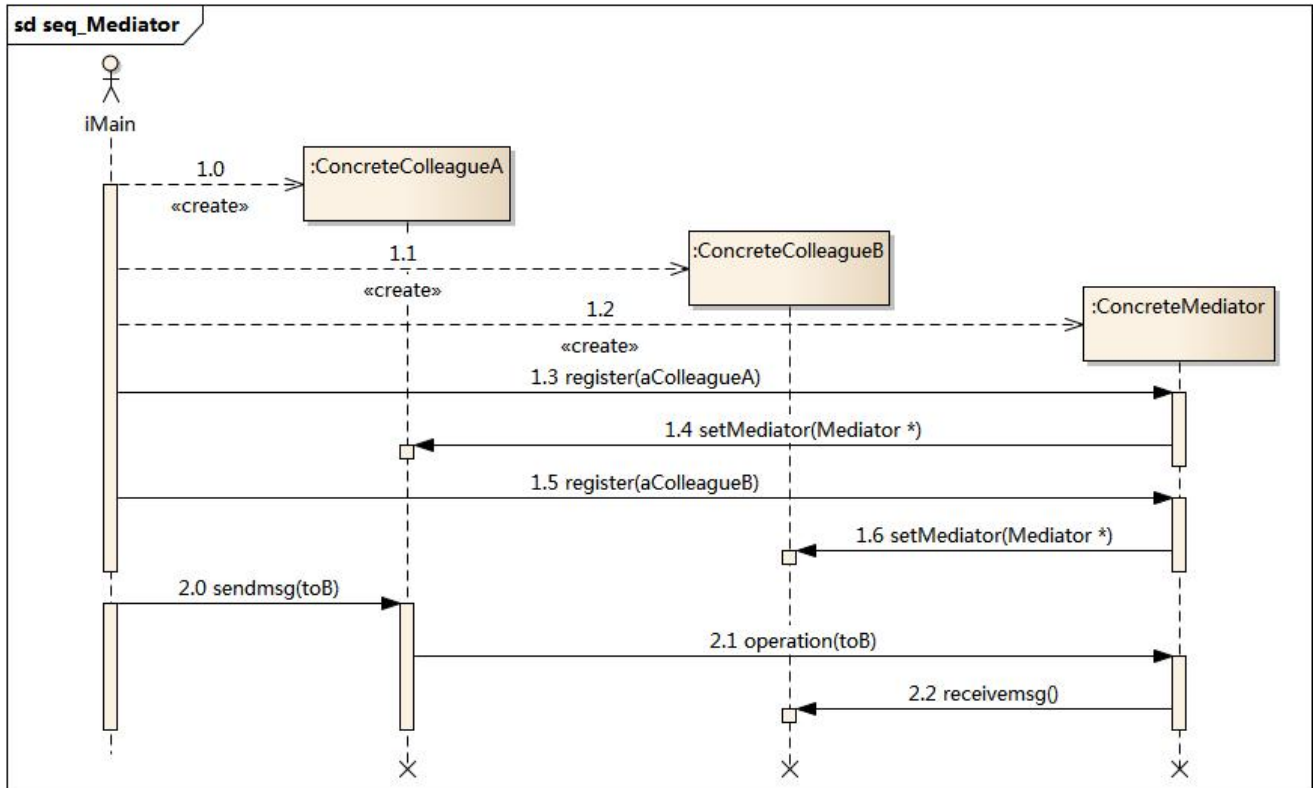
2.3. 模式结构

中介者模式包含如下角色：

- Mediator: 抽象中介者
- ConcreteMediator: 具体中介者
- Colleague: 抽象同事类
- ConcreteColleague: 具体同事类



2.4. 时序图



2.5. 代码分析

```

1  #include <iostream>
2  #include "ConcreteColleagueA.h"
3  #include "ConcreteMediator.h"
4  #include "ConcreteColleagueB.h"
5
6  using namespace std;
7
8  int main(int argc, char *argv[])
9  {
10     ConcreteColleagueA * pa = new ConcreteColleagueA();
11     ConcreteColleagueB * pb = new ConcreteColleagueB();
12     ConcreteMediator * pm = new ConcreteMediator();
13     pm->registered(1,pa);
14     pm->registered(2,pb);
15
16     // sendmsg from a to b
17     pa->sendmsg(2,"hello,i am a");
18     // sendmsg from b to a
19     pb->sendmsg(1,"hello,i am b");
20
21     delete pa,pb,pm;
22     return 0;
23 }
  
```

```
1 //////////////////////////////////////////////////
2 // ConcreteMediator.h
3 // Implementation of the Class ConcreteMediator
4 // Created on:      07-十月-2014 21:30:47
5 // Original author: colin
6 //////////////////////////////////////////////////
7 #if !defined(EA_8CECE546_61DD_456f_A3E7_D98BC078D8E8__INCLUDED_)
8 #define EA_8CECE546_61DD_456f_A3E7_D98BC078D8E8__INCLUDED_
9
10 #include "ConcreteColleagueB.h"
11 #include "Mediator.h"
12 #include "ConcreteColleagueA.h"
13 #include <map>
14 using namespace std;
15 class ConcreteMediator : public Mediator
16 {
17 public:
18     ConcreteMediator();
19     virtual ~ConcreteMediator();
20
21     virtual void operation(int nWho,string str);
22     virtual void registered(int nWho, Colleague * aColleague);
23 private:
24     map<int,Colleague*> m_mpColleague;
25 };
26 #endif // !defined(EA_8CECE546_61DD_456f_A3E7_D98BC078D8E8__INCLUDED_)
27
28
```



```
1  //////////////////////////////////////
2  //  ConcreteMediator.cpp
3  //  Implementation of the Class ConcreteMediator
4  //  Created on:      07-十月-2014 21:30:48
5  //  Original author: colin
6  //////////////////////////////////////
7  #include "ConcreteMediator.h"
8  #include <map>
9  #include <iostream>
10 using namespace std;
11
12 ConcreteMediator::ConcreteMediator(){
13 }
14 ConcreteMediator::~ConcreteMediator(){
15 }
16
17 void ConcreteMediator::operation(int nWho,string str){
18     map<int,Colleague*>::const_iterator itr = m_mpColleague.find(nWho);
19     if(itr == m_mpColleague.end())
20     {
21         cout << "not found this colleague!" << endl;
22         return;
23     }
24     Colleague* pc = itr->second;
25     pc->receivemsg(str);
26 }
27
28 void ConcreteMediator::registered(int nWho,Colleague * aColleague){
29     map<int,Colleague*>::const_iterator itr = m_mpColleague.find(nWho);
30     if(itr == m_mpColleague.end())
31     {
32         m_mpColleague.insert(make_pair(nWho,aColleague));
33         //同时将中介类暴露给colleague
34         aColleague->setMediator(this);
35     }
36 }
37
38
39
40
41
42
```

```

1  //////////////////////////////////////
2  //  ConcreteColleagueA.h
3  //  Implementation of the Class ConcreteColleagueA
4  //  Created on:      07-十月-2014 21:30:47
5  //  Original author: colin
6  //////////////////////////////////////
7  #if !defined(EA_79979DD4_1E73_46db_A635_E3F516ACCE0A__INCLUDED_)
8  #define EA_79979DD4_1E73_46db_A635_E3F516ACCE0A__INCLUDED_
9
10 #include "Colleague.h"
11
12 class ConcreteColleagueA : public Colleague
13 {
14 public:
15     ConcreteColleagueA();
16     virtual ~ConcreteColleagueA();
17
18     virtual void sendmsg(int toWho,string str);
19     virtual void receivemsg(string str);
20
21 };
22 #endif // !defined(EA_79979DD4_1E73_46db_A635_E3F516ACCE0A__INCLUDED_)
23
24

```

```

1  //////////////////////////////////////
2  //  ConcreteColleagueA.cpp
3  //  Implementation of the Class ConcreteColleagueA
4  //  Created on:      07-十月-2014 21:30:47
5  //  Original author: colin
6  //////////////////////////////////////
7  #include "ConcreteColleagueA.h"
8  #include <iostream>
9  using namespace std;
10
11 ConcreteColleagueA::ConcreteColleagueA(){
12 }
13
14 ConcreteColleagueA::~~ConcreteColleagueA(){
15 }
16
17 void ConcreteColleagueA::sendmsg(int toWho,string str){
18     cout << "send msg from colleagueA,to:" << toWho << endl;
19     m_pMediator->operation(toWho,str);
20 }
21
22 void ConcreteColleagueA::receivemsg(string str){
23     cout << "ConcreteColleagueA reveivemsg:" << str <<endl;
24 }
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

运行结果：

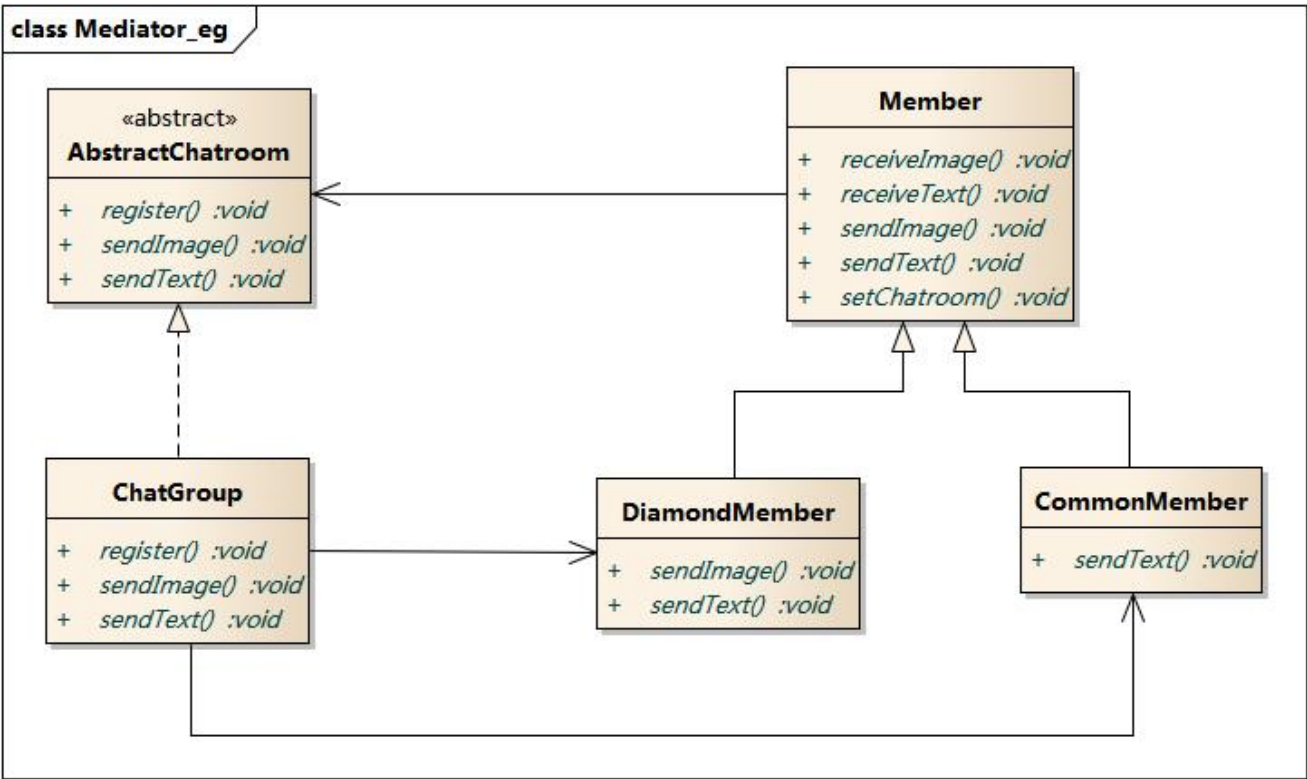
```
"C:\Users\c\Documents\GitHub\design_patterns\code\Mediator\mingw5\main.exe"
send msg from colleagueA,to:2
ConcreteColleagueB reveivemsg:hello,i am a
send msg from colleagueB,to:1
ConcreteColleagueA reveivemsg:hello,i am b
请按任意键继续. . .
```

2.6. 模式分析

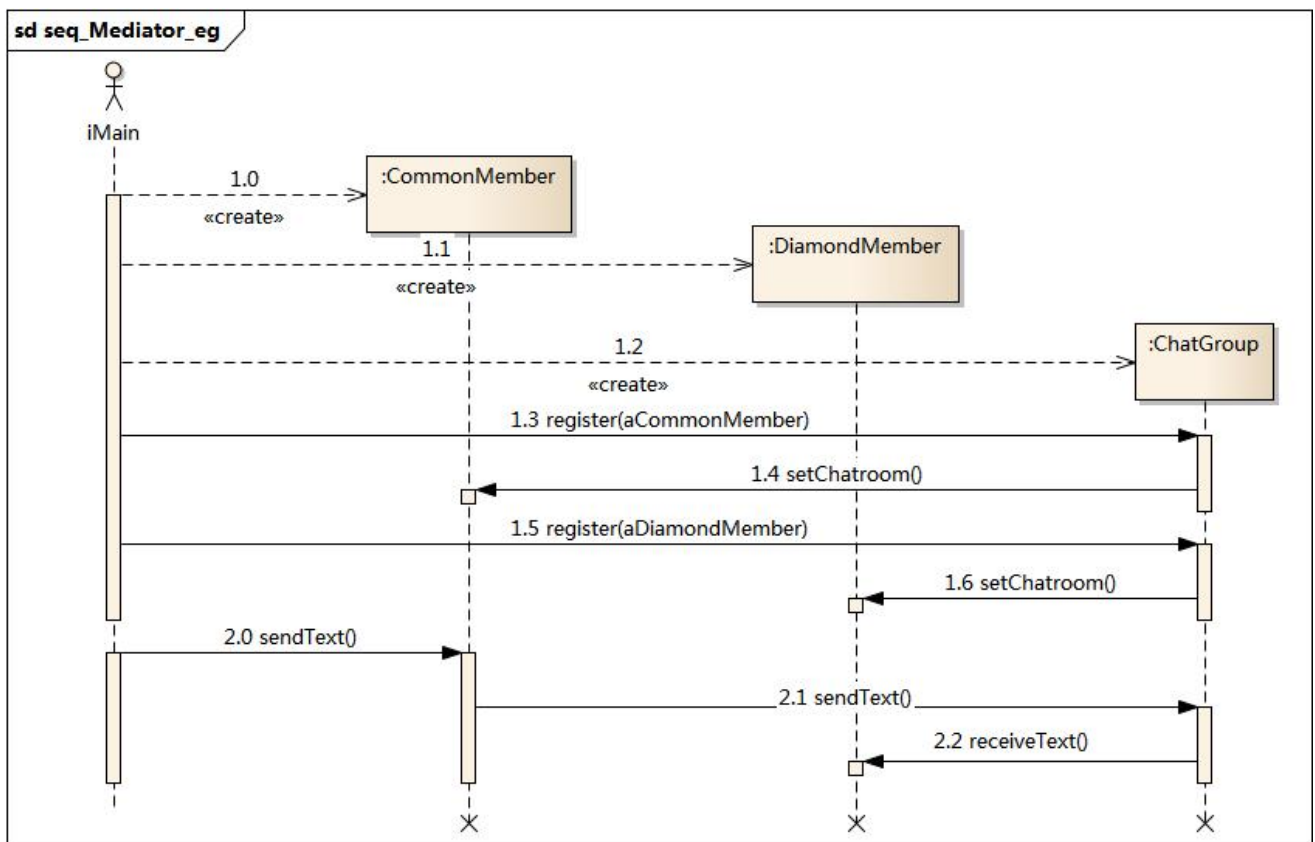
中介者模式可以使对象之间的关系数量急剧减少。

中介者承担两方面的职责：

- 中转作用（结构性）：通过中介者提供的中转作用，各个同事对象就不再需要显式引用其他同事，当需要和其他同事进行通信时，通过中介者即可。该中转作用属于中介者在结构上的支持。
- 协调作用（行为性）：中介者可以更进一步的对同事之间的关系进行封装，同事可以一致地和中介者进行交互，而不需要指明中介者需要具体怎么做，中介者根据封装在自身内部的协调逻辑，对同事的请求进行进一步处理，将同事成员之间的关系行为进行分离和封装。该协调作用属于中介者在行为上的支持。



时序图



2.7. 实例

实例：虚拟聊天室

某论坛系统欲增加一个虚拟聊天室，允许论坛会员通过该聊天室进行信息交流，普通会员(CommonMember)可以给其他会员发送文本信息，钻石会员(DiamondMember)既可以给其他会员发送文本信息，还可以发送图片信息。该聊天室可以对不雅字符进行过滤，如“日”等字符；还可以对发送的图片大小进行控制。用中介者模式设计该虚拟聊天室。

2.8. 优点

中介者模式的优点

- 简化了对象之间的交互。
- 将各同事解耦。
- 减少子类生成。
- 可以简化各同事类的设计和实现。

2.9. 缺点

中介者模式的缺点

- 在具体中介者类中包含了同事之间的交互细节，可能会导致具体中介者类非常复杂，使得系统难以维护。

2.10. 适用环境

在以下情况下可以使用中介者模式：

- 系统中对象之间存在复杂的引用关系，产生的相互依赖关系结构混乱且难以理解。
- 一个对象由于引用了其他很多对象并且直接和这些对象通信，导致难以复用该对象。
- 想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。可以通过引入中介者类来实现，在中介者中定义对象。
- 交互的公共行为，如果需要改变行为则可以增加新的中介者类。

2.11. 模式应用

MVC架构中控制器

Controller 作为一种中介者，它负责控制视图对象View和模型对象Model之间的交互。如在Struts中，Action就可以作为JSP页面与业务对象之间的中介者。

2.12. 模式扩展

中介者模式与迪米特法则

- 在中介者模式中，通过创造出一个中介者对象，将系统中有关对象所引用的其他对象数目减少到最少，使得一个对象与其同事之间的相互作用被这个对象与中介者对象之间的相互作用所取代。因此，中介者模式就是迪米特法则的一个典型应用。

中介者模式与GUI开发

- 中介者模式可以方便地应用于图形界面(GUI)开发中，在比较复杂的界面中可能存在多个界面组件之间的交互关系。
- 对于这些复杂的交互关系，有时候我们可以引入一个中介者类，将这些交互的组件作为具体的同事类，将它们之间的引用和控制关系交由中介者负责，在一定程度上简化系统的交互，这也是中介者模式的常见应用之一。

2.13. 总结

- 中介者模式用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。中介者模式又称为调停者模式，它是一种对象行为型模式。
- 中介者模式包含四个角色：抽象中介者用于定义一个接口，该接口用于与各同事对象之间的通信；具体中介者是抽象中介者的子类，通过协调各个同事对象来实现协作行为，了解并维护它的各个同事对象的引用；抽象同事类定义各同事的公有方法；具体同事类是抽象同事类的子类，每一个同事对象都引用一个中介者对象；每一个同事对象在需要和其他同

事对象通信时，先与中介者通信，通过中介者来间接完成与其他同事类的通信；在具体同事类中实现了在抽象同事类中定义的方法。

- 通过引入中介者对象，可以将系统的网状结构变成以中介者为中心的星形结构，中介者承担了中转作用和协调作用。中介者类是中介者模式的核心，它对整个系统进行控制和协调，简化了对象之间的交互，还可以对对象间的交互进行进一步的控制。
- 中介者模式的主要优点在于简化了对象之间的交互，将各同事解耦，还可以减少子类生成，对于复杂的对象之间的交互，通过引入中介者，可以简化各同事类的设计和实现；中介者模式主要缺点在于具体中介者类中包含了同事之间的交互细节，可能会导致具体中介者类非常复杂，使得系统难以维护。
- 中介者模式适用情况包括：系统中对象之间存在复杂的引用关系，产生的相互依赖关系结构混乱且难以理解；一个对象由于引用了其他很多对象并且直接和这些对象通信，导致难以复用该对象；想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。

3. 观察者模式

目录

- 观察者模式
 - 模式动机
 - 模式定义
 - 模式结构
 - 时序图
 - 代码分析
 - 模式分析
 - 实例
 - 优点
 - 缺点
 - 适用环境
 - 模式应用
 - 模式扩展
 - 总结

3.1. 模式动机

建立一种对象与对象之间的依赖关系，一个对象发生改变时将自动通知其他对象，其他对象将相应做出反应。在此，发生改变的对象称为观察目标，而被通知的对象称为观察者，一个观察目标可以对应多个观察者，而且这些观察者之间没有相互联系，可以根据需要增加和删除观察者，使得系统更易于扩展，这就是观察者模式的模式动机。

3.2. 模式定义

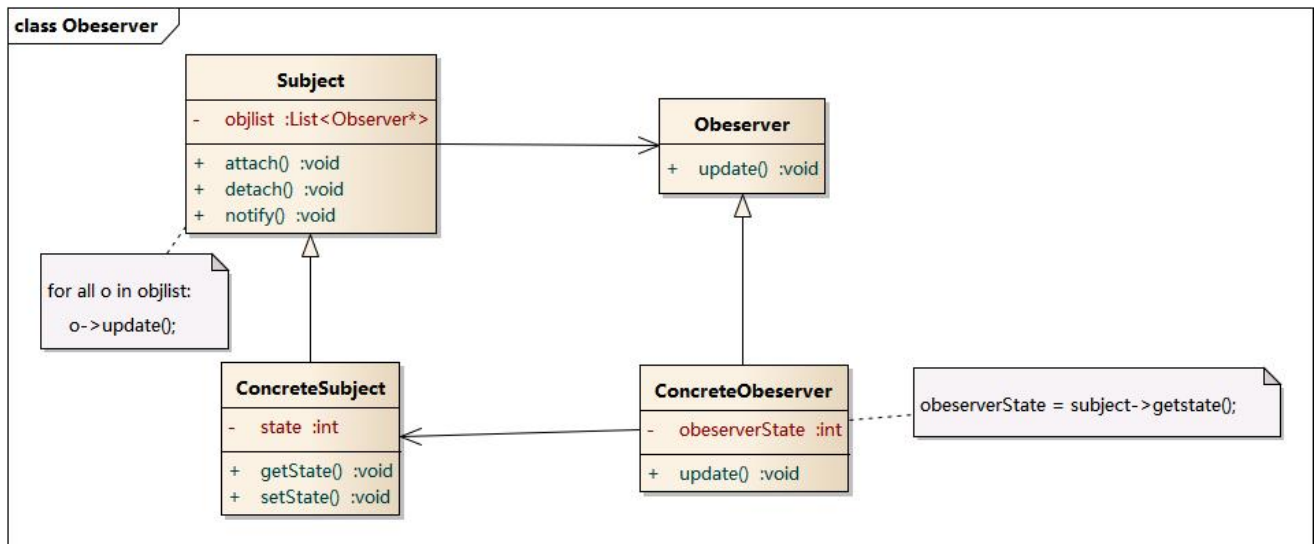
观察者模式(Observer Pattern)：定义对象间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。观察者模式又叫做发布-订阅(Publish/Subscribe)模式、模型-视图(Model/View)模式、源-监听器(Source/Listener)模式或从属者(Dependents)模式。

观察者模式是一种对象行为型模式。

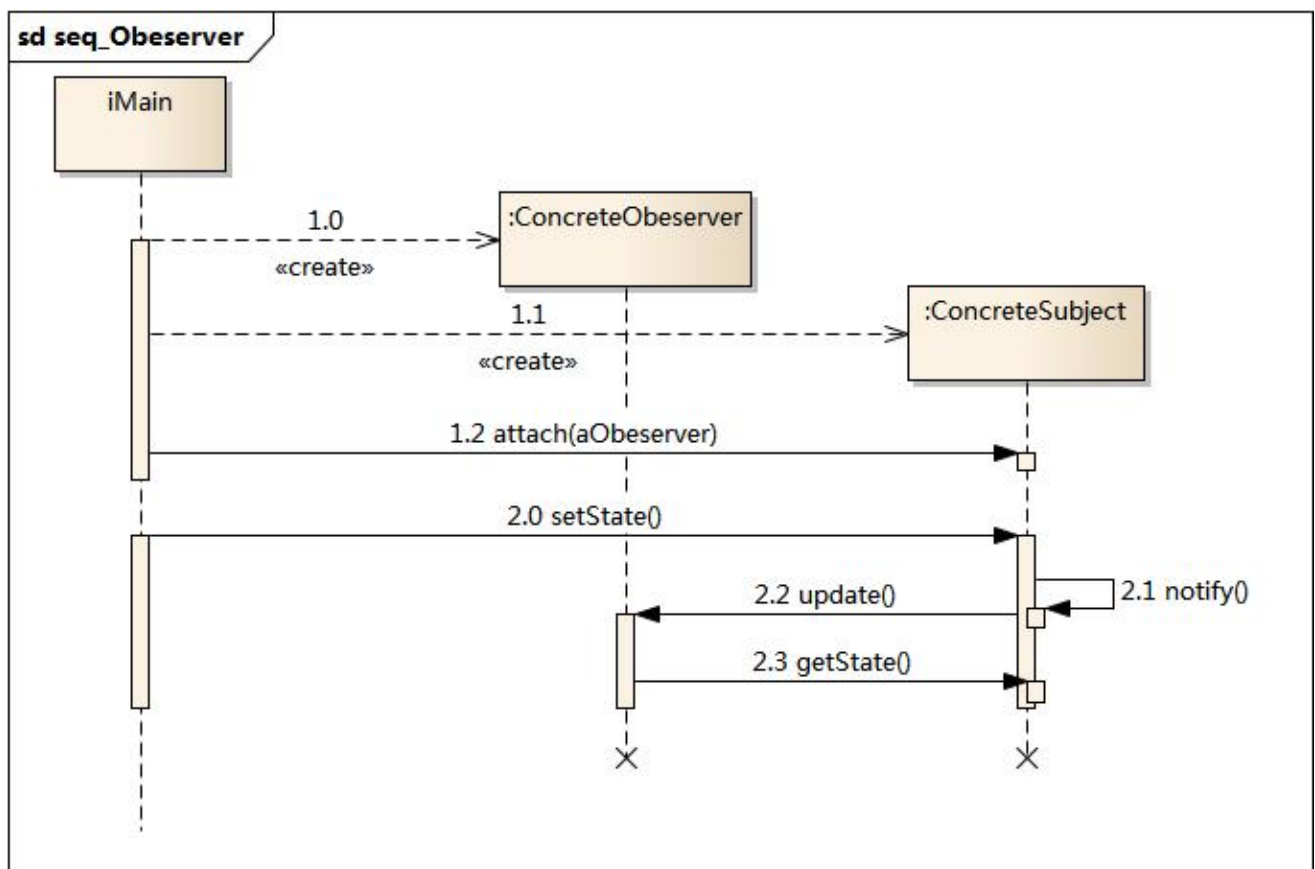
3.3. 模式结构

观察者模式包含如下角色：

- Subject: 目标
- ConcreteSubject: 具体目标
- Observer: 观察者
- ConcreteObserver: 具体观察者



3.4. 时序图



3.5. 代码分析




```
1  #include <iostream>
2  #include "Subject.h"
3  #include "Obeserver.h"
4  #include "ConcreteObeserver.h"
5  #include "ConcreteSubject.h"
6
7  using namespace std;
8
9  int main(int argc, char *argv[])
10 {
11     Subject * subject = new ConcreteSubject();
12     Obeserver * objA = new ConcreteObeserver("A");
13     Obeserver * objB = new ConcreteObeserver("B");
14     subject->attach(objA);
15     subject->attach(objB);
16
17     subject->setState(1);
18     subject->notify();
19
20     cout << "-----" << endl;
21     subject->detach(objB);
22     subject->setState(2);
23     subject->notify();
24
25     delete subject;
26     delete objA;
27     delete objB;
28
29     return 0;
30 }
```

```
1  //////////////////////////////////////
2  // Subject.h
3  // Implementation of the Class Subject
4  // Created on:      07-十月-2014 23:00:10
5  // Original author: cl
6  //////////////////////////////////////
7  #if !defined(EA_61998456_1B61_49f4_B3EA_9D28EEBC9649__INCLUDED_)
8  #define EA_61998456_1B61_49f4_B3EA_9D28EEBC9649__INCLUDED_
9
10 #include "Obeserver.h"
11 #include <vector>
12 using namespace std;
13
14 class Subject
15 {
16 public:
17     Subject();
18     virtual ~Subject();
19     Obeserver *m_Obeserver;
20
21     void attach(Obeserver * pObeserver);
22     void detach(Obeserver * pObeserver);
23     void notify();
24
25     virtual int getState() = 0;
26     virtual void setState(int i)= 0;
27
28 private:
29     vector<Obeserver*> m_vtObj;
30
31 };
32 #endif // !defined(EA_61998456_1B61_49f4_B3EA_9D28EEBC9649__INCLUDED_)
33
34
```

```
1  //////////////////////////////////////
2  // Subject.cpp
3  // Implementation of the Class Subject
4  // Created on:      07-十月-2014 23:00:10
5  // Original author: cl
6  //////////////////////////////////////
7  #include "Subject.h"
8  Subject::Subject(){
9
10 }
11 Subject::~Subject(){
12
13 }
14 void Subject::attach(Obeserver * pObeserver){
15     m_vtObj.push_back(pObeserver);
16 }
17 void Subject::detach(Obeserver * pObeserver){
18     for(vector<Obeserver*>::iterator itr = m_vtObj.begin();
19         itr != m_vtObj.end(); itr++)
20     {
21         if(*itr == pObeserver)
22         {
23             m_vtObj.erase(itr);
24             return;
25         }
26     }
27 void Subject::notify(){
28     for(vector<Obeserver*>::iterator itr = m_vtObj.begin();
29         itr != m_vtObj.end();
30         itr++)
31     {
32         (*itr)->update(this);
33     }
34
35
36
37
38
39
40
41
```

```

1  //////////////////////////////////////
2  //  Obeserver.h
3  //  Implementation of the Class Obeserver
4  //  Created on:      07-十月-2014 23:00:10
5  //  Original author: cl
6  //////////////////////////////////////
7  #if !defined(EA_2C7362B2_0B22_4168_8690_F9C7B76C343F__INCLUDED_)
8  #define EA_2C7362B2_0B22_4168_8690_F9C7B76C343F__INCLUDED_
9
10 class Subject;
11
12 class Obeserver
13 {
14 public:
15     Obeserver();
16     virtual ~Obeserver();
17     virtual void update(Subject * sub) = 0;
18 };
19 #endif // !defined(EA_2C7362B2_0B22_4168_8690_F9C7B76C343F__INCLUDED_)
20
21

```

```

1  //////////////////////////////////////
2  //  ConcreteObeserver.h
3  //  Implementation of the Class ConcreteObeserver
4  //  Created on:      07-十月-2014 23:00:09
5  //  Original author: cl
6  //////////////////////////////////////
7  #if !defined(EA_7B020534_BFEA_4c9e_8E4C_34DCE001E9B1__INCLUDED_)
8  #define EA_7B020534_BFEA_4c9e_8E4C_34DCE001E9B1__INCLUDED_
9  #include "Obeserver.h"
10 #include <string>
11 using namespace std;
12
13 class ConcreteObeserver : public Obeserver
14 {
15 public:
16     ConcreteObeserver(string name);
17     virtual ~ConcreteObeserver();
18     virtual void update(Subject * sub);
19
20 private:
21     string m_objName;
22     int m_obeserverState;
23 };
24 #endif // !defined(EA_7B020534_BFEA_4c9e_8E4C_34DCE001E9B1__INCLUDED_)
25
26

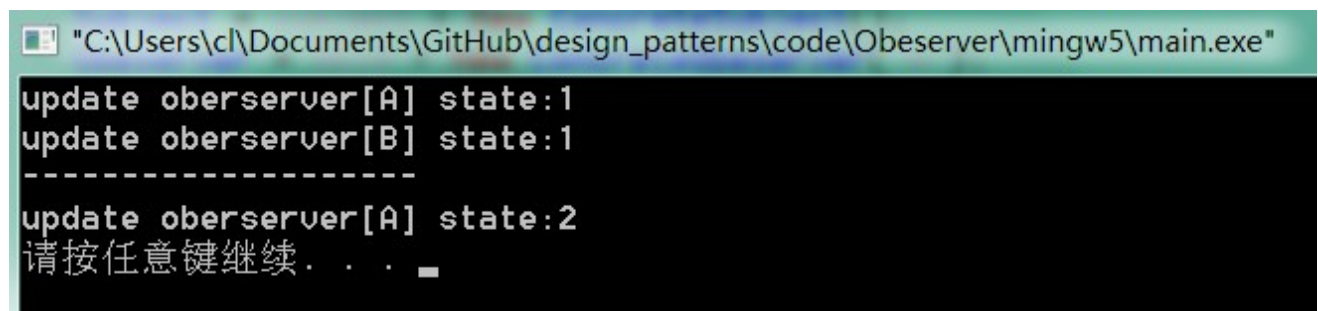
```

```

1  //////////////////////////////////////
2  //  ConcreteObeserver.cpp
3  //  Implementation of the Class ConcreteObeserver
4  //  Created on:      07-十月-2014 23:00:09
5  //  Original author: cl
6  //////////////////////////////////////
7
8  #include "ConcreteObeserver.h"
9  #include <iostream>
10 #include <vector>
11 #include "Subject.h"
12 using namespace std;
13
14 ConcreteObeserver::ConcreteObeserver(string name){
15     m_objName = name;
16 }
17
18 ConcreteObeserver::~ConcreteObeserver(){
19 }
20
21 void ConcreteObeserver::update(Subject * sub){
22     m_obeserverState = sub->getState();
23     cout << "update oberserver[" << m_objName << "] state:" <<
24     m_obeserverState << endl;
25 }

```

运行结果：



```

"C:\Users\cl\Documents\GitHub\design_patterns\code\Obeserver\mingw5\main.exe"
update oberserver[A] state:1
update oberserver[B] state:1
-----
update oberserver[A] state:2
请按任意键继续. . .

```

3.6. 模式分析

- 观察者模式描述了如何建立对象与对象之间的依赖关系，如何构造满足这种需求的系统。
- 这一模式中的关键对象是观察目标和观察者，一个目标可以有任意数目的与之相依赖的观察者，一旦目标的状态发生改变，所有的观察者都将得到通知。
- 作为对这个通知的响应，每个观察者都将即时更新自己的状态，以与目标状态同步，这种交互也称为发布-订阅(publishsubscribe)。目标是通知的发布者，它发出通知时并不需要知道谁是它的观察者，可以有任意数目的观察者订阅它并接收通

知。

3.7. 实例

3.8. 优点

观察者模式的优点

- 观察者模式可以实现表示层和数据逻辑层的分离，并定义了稳定的消息更新传递机制，抽象了更新接口，使得可以有各种各样不同的表示层作为具体观察者角色。
- 观察者模式在观察目标和观察者之间建立一个抽象的耦合。
- 观察者模式支持广播通信。
- 观察者模式符合“开闭原则”的要求。

3.9. 缺点

观察者模式的缺点

- 如果一个观察目标对象有很多直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。
- 如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。
- 观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

3.10. 适用环境

在以下情况下可以使用观察者模式：

- 一个抽象模型有两个方面，其中一个方面依赖于另一个方面。将这些方面封装在独立的对象中使它们可以各自独立地改变和复用。
- 一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体有多少对象将发生改变，可以降低对象之间的耦合度。
- 一个对象必须通知其他对象，而并不知道这些对象是谁。
- 需要在系统中创建一个触发链，A对象的行为将影响B对象，B对象的行为将影响C对象.....，可以使用观察者模式创建一种链式触发机制。

3.11. 模式应用

观察者模式在软件开发中应用非常广泛，如某电子商务网站可以在执行发送操作后给用户多个发送商品打折信息，某团队战斗游戏中某队友牺牲将给所有成员提示等等，凡是涉及到一对一或者一对多的对象交互场景都可以使用观察者模式。

3.12. 模式扩展

MVC模式

- MVC模式是一种架构模式，它包含三个角色：模型(Model)，视图(View)和控制器(Controller)。观察者模式可以用来实现MVC模式，观察者模式中的观察目标就是MVC模式中的模型(Model)，而观察者就是MVC中的视图(View)，控制器(Controller)充当两者之间的中介者(Mediator)。当模型层的数据发生改变时，视图层将自动改变其显示内容。

3.13. 总结

- 观察者模式定义对象间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。观察者模式又叫做发布-订阅模式、模型-视图模式、源-监听器模式或从属者模式。观察者模式是一种对象行为型模式。
- 观察者模式包含四个角色：目标又称为主题，它是指被观察的对象；具体目标是目标类的子类，通常它包含有经常发生改变的数据，当它的状态发生改变时，向它的各个观察者发出通知；观察者将对观察目标的改变做出反应；在具体观察者中维护一个指向具体目标对象的引用，它存储具体观察者的有关状态，这些状态需要和具体目标的状态保持一致。
- 观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个目标对象，当这个目标对象的状态发生变化时，会通知所有观察者对象，使它们能够自动更新。
- 观察者模式的主要优点在于可以实现表示层和数据逻辑层的分离，并在观察目标和观察者之间建立一个抽象的耦合，支持广播通信；其主要缺点在于如果一个观察目标对象有很多直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间，而且如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。
- 观察者模式适用情况包括：一个抽象模型有两个方面，其中一个方面依赖于另一个方面；一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体有多少对象将发生改变；一个对象必须通知其他对象，而并不知道这些对象是谁；需要在系统中创建一个触发链。
- 在JDK的java.util包中，提供了Observable类以及Observer接口，它们构成了Java语言对观察者模式的支持。

4. 状态模式

目录

- 状态模式
 - 模式动机
 - 模式定义
 - 模式结构
 - 时序图
 - 代码分析
 - 模式分析
 - 实例
 - 优点
 - 缺点
 - 适用环境
 - 模式应用
 - 模式扩展
 - 总结

4.1. 模式动机

- 在很多情况下，一个对象的行为取决于一个或多个动态变化的属性，这样的属性叫做状态，这样的对象叫做有状态的(stateful)对象，这样的对象状态是从事先定义好的一系列值中取出的。当一个这样的对象与外部事件产生互动时，其内部状态就会改变，从而使得系统的行为也随之发生变化。
- 在UML中可以使用状态图来描述对象状态的变化。

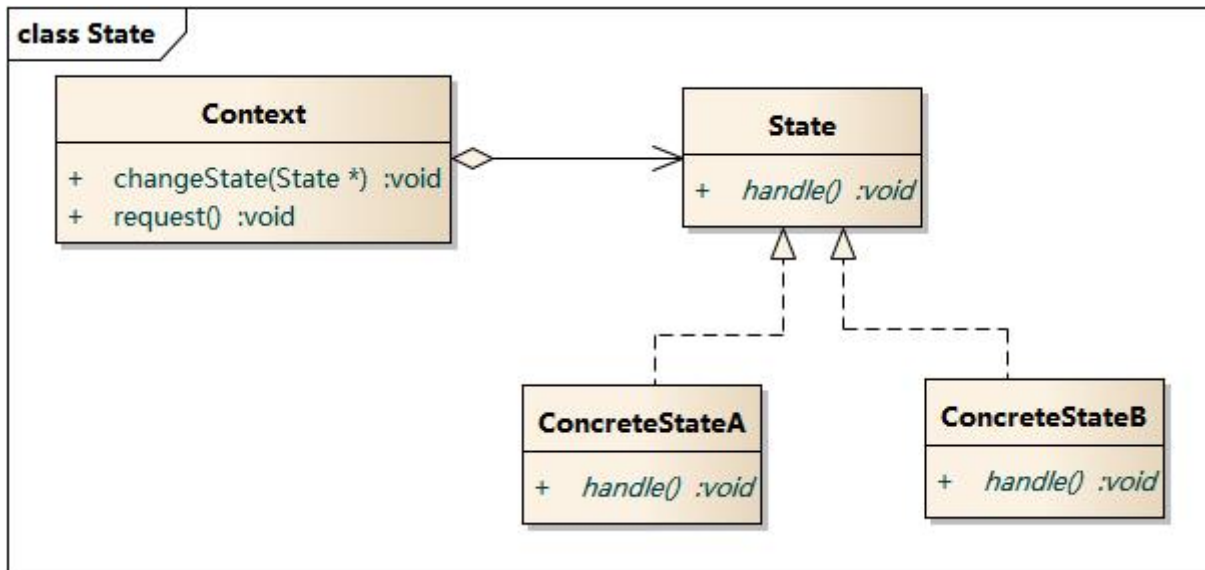
4.2. 模式定义

状态模式(State Pattern)：允许一个对象在其内部状态改变时改变它的行为，对象看起来似乎修改了它的类。其别名为状态对象(Objects for States)，状态模式是一种对象行为型模式。

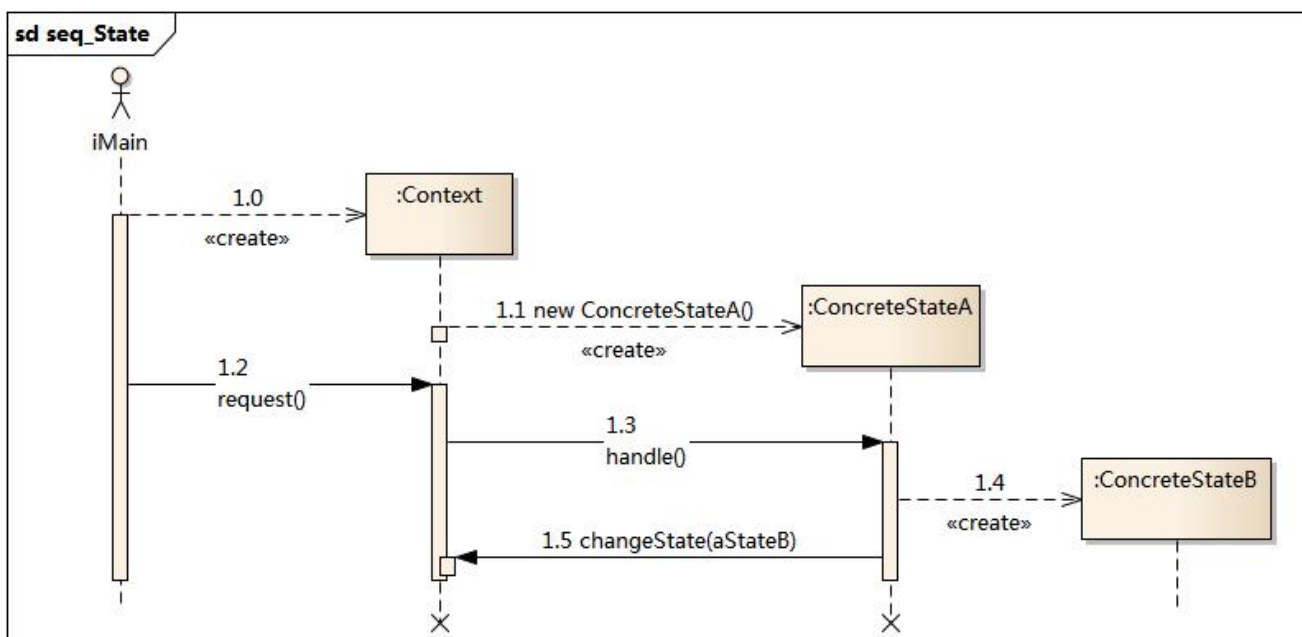
4.3. 模式结构

状态模式包含如下角色：

- Context: 环境类
- State: 抽象状态类
- ConcreteState: 具体状态类



4.4. 时序图



既然是状态模式，加上状态图，对状态转换间的理解会清晰很多：

_static/State1.png

4.5. 代码分析

```

1  #include <iostream>
2  #include "Context.h"
3  #include "ConcreteStateA.h"
4  #include "ConcreteStateB.h"
5
6  using namespace std;
7
8  int main(int argc, char *argv[])
9  {
10     char a = '0';
11     if('0' == a)
12         cout << "yes" << endl;
13     else
14         cout << "no" << endl;
15
16     Context * c = new Context();
17     c->request();
18     c->request();
19     c->request();
20
21     delete c;
22     return 0;
23 }

```

```

1  //////////////////////////////////////
2  // Context.h
3  // Implementation of the Class Context
4  // Created on:      09-十月-2014 17:20:59
5  // Original author: colin
6  //////////////////////////////////////
7
8  #if !defined(EA_F245CF81_2A68_4461_B039_2B901BD5A126__INCLUDED_)
9  #define EA_F245CF81_2A68_4461_B039_2B901BD5A126__INCLUDED_
10
11 #include "State.h"
12
13 class Context
14 {
15 public:
16     Context();
17     virtual ~Context();
18
19     void changeState(State * st);
20     void request();
21
22 private:
23     State *m_pState;
24 };
25 #endif // !defined(EA_F245CF81_2A68_4461_B039_2B901BD5A126__INCLUDED_)
26

```

```

1  //////////////////////////////////////
2  // Context.cpp
3  // Implementation of the Class Context
4  // Created on:      09-十月-2014 17:20:59
5  // Original author: colin
6  //////////////////////////////////////
7  #include "Context.h"
8  #include "ConcreteStateA.h"
9
10 Context::Context(){
11     //default is a
12     m_pState = ConcreteStateA::Instance();
13 }
14
15 Context::~Context(){
16 }
17
18 void Context::changeState(State * st){
19     m_pState = st;
20 }
21
22 void Context::request(){
23     m_pState->handle(this);
24 }
25

```

```

1  //////////////////////////////////////
2  // ConcreteStateA.h
3  // Implementation of the Class ConcreteStateA
4  // Created on:      09-十月-2014 17:20:58
5  // Original author: colin
6  //////////////////////////////////////
7  #if !defined(EA_84158F08_E96A_4bdb_89A1_4BE2E633C3EE__INCLUDED_)
8  #define EA_84158F08_E96A_4bdb_89A1_4BE2E633C3EE__INCLUDED_
9
10 #include "State.h"
11
12 class ConcreteStateA : public State
13 {
14 public:
15     virtual ~ConcreteStateA();
16
17     static State * Instance();
18
19     virtual void handle(Context * c);
20
21 private:
22     ConcreteStateA();
23     static State * m_pState;
24 };
25 #endif // !defined(EA_84158F08_E96A_4bdb_89A1_4BE2E633C3EE__INCLUDED_)
26
27

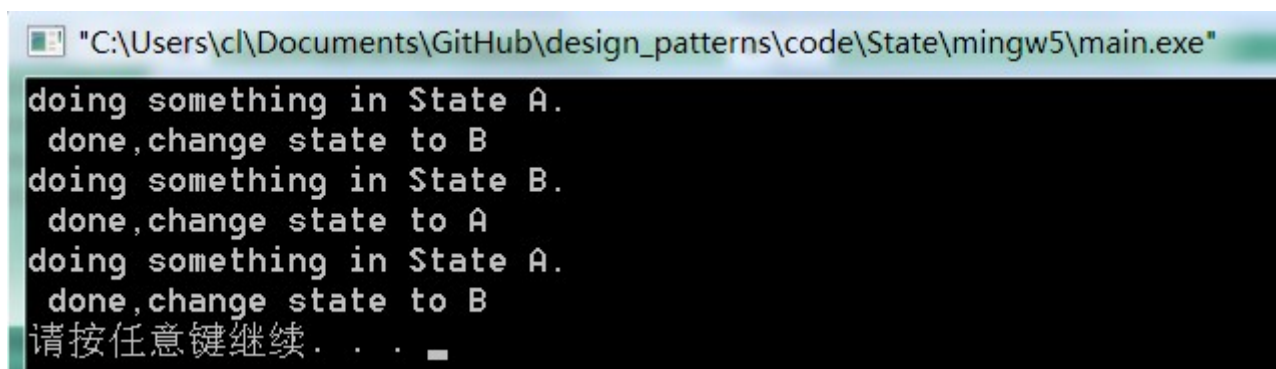
```

```

1  //////////////////////////////////////
2  //  ConcreteStateA.cpp
3  //  Implementation of the Class ConcreteStateA
4  //  Created on:      09-十月-2014 17:20:58
5  //  Original author: colin
6  //////////////////////////////////////
7
8  #include "ConcreteStateA.h"
9  #include "ConcreteStateB.h"
10 #include "Context.h"
11 #include <iostream>
12 using namespace std;
13
14 State * ConcreteStateA::m_pState = NULL;
15 ConcreteStateA::ConcreteStateA(){
16 }
17
18 ConcreteStateA::~~ConcreteStateA(){
19 }
20
21 State * ConcreteStateA::Instance()
22 {
23     if ( NULL == m_pState)
24     {
25         m_pState = new ConcreteStateA();
26     }
27     return m_pState;
28 }
29
30 void ConcreteStateA::handle(Context * c){
31     cout << "doing something in State A.\n done,change state to B" << endl;
32     c->changeState(ConcreteStateB::Instance());
33 }

```

运行结果：



```

"C:\Users\cl\Documents\GitHub\design_patterns\code\State\mingw5\main.exe"
doing something in State A.
done,change state to B
doing something in State B.
done,change state to A
doing something in State A.
done,change state to B
请按任意键继续. . .

```

4.6. 模式分析

- 状态模式描述了对对象状态的变化以及对象如何在每一种状态下表现出不同的行为。
- 状态模式的关键是引入了一个抽象类来专门表示对象的状态，这个类我们叫做抽象状态类，而对象的每一种具体状态类都继承了该类，并在不同具体状态类中实现了不同状态的行为，包括各种状态之间的转换。

在状态模式结构中需要理解环境类与抽象状态类的作用：

- 环境类实际上就是拥有状态的对象，环境类有时候可以充当状态管理器(State Manager)的角色，可以在环境类中对状态进行切换操作。
- 抽象状态类可以是抽象类，也可以是接口，不同状态类就是继承这个父类的不同子类，状态类的产生是由于环境类存在多个状态，同时还满足两个条件： 这些状态经常需要切换，在不同的状态下对象的行为不同。因此可以将不同对象下的行为单独提取出来封装在具体的状态类中，使得环境类对象在其内部状态改变时可以改变它的行为，对象看起来似乎修改了它的类，而实际上是由于切换到不同的具体状态类实现的。由于环境类可以设置为任一具体状态类，因此它针对抽象状态类进行编程，在程序运行时可以将任一具体状态类的对象设置到环境类中，从而使得环境类可以改变内部状态，并且改变行为。

4.7. 实例

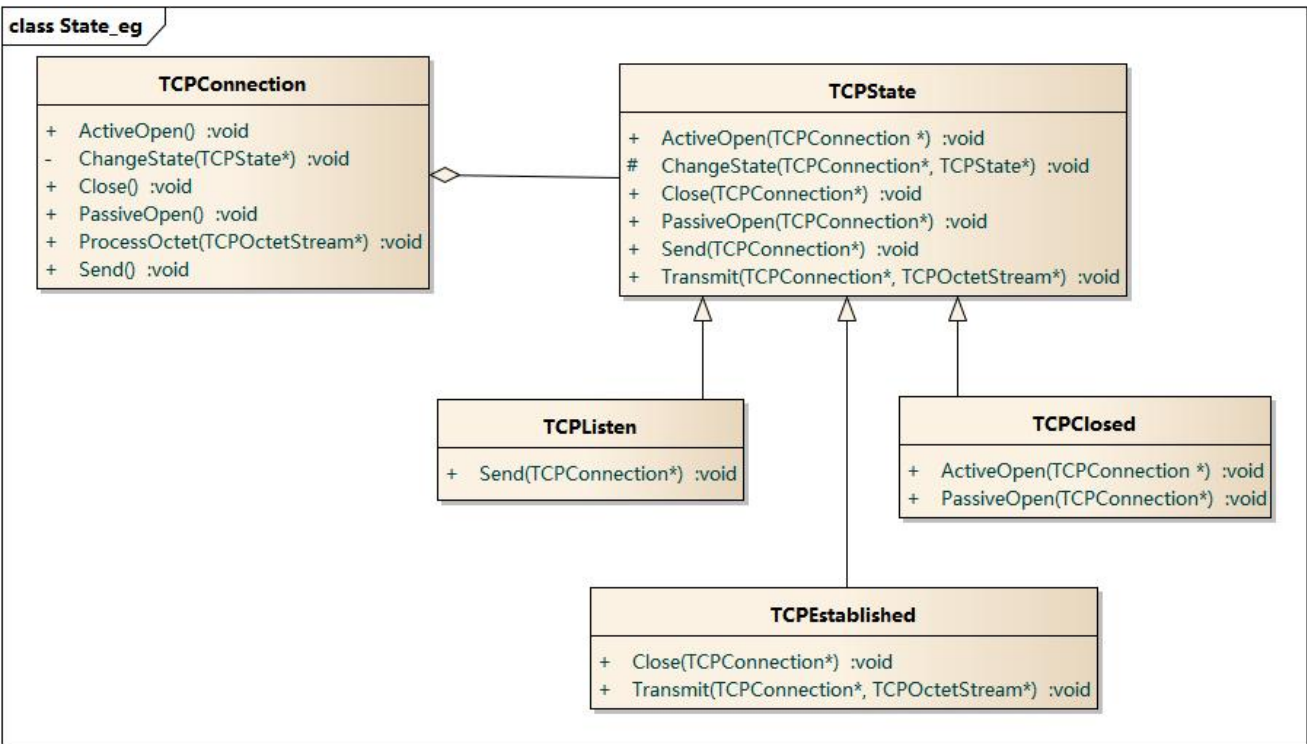
TCPConnection

这个示例来自《设计模式》,展示了一个简化版的TCP协议实现; TCP连接的状态有多种可能，状态之间的转换有相应的逻辑前提； 这是使用状态模式的场合；

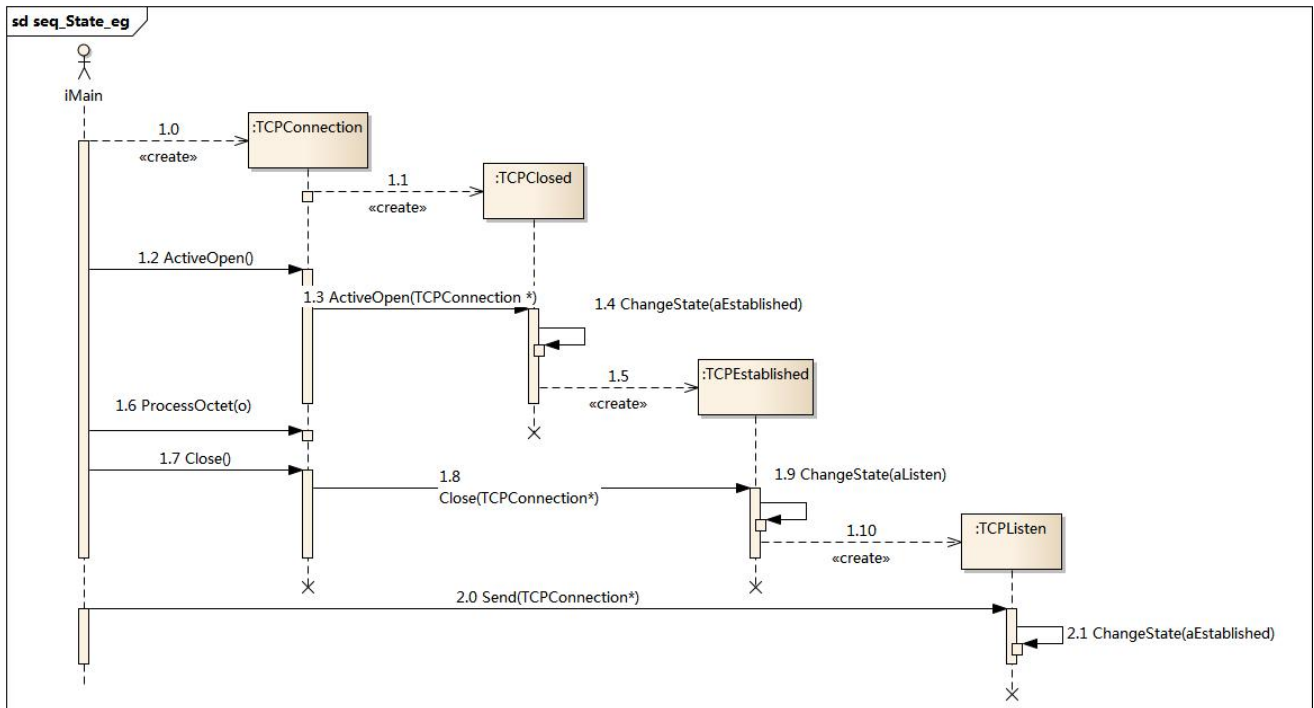
状态图：



结构图：



时序图：



4.8. 优点

状态模式的优点

- 封装了转换规则。
- 枚举可能的状态，在枚举状态之前需要确定状态种类。
- 将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。
- 允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。
- 可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数。

4.9. 缺点

状态模式的缺点

- 状态模式的使用必然会增加系统类和对象的个数。
- 状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。
- 状态模式对“开闭原则”的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码，否则无法切换到新增状态；而且修改某个状态类的行为也需修改对应类的源代码。

4.10. 适用环境

在以下情况下可以使用状态模式：

- 对象的行为依赖于它的状态（属性）并且可以根据它的状态改变而改变它的相关行为。
- 代码中包含大量与对象状态有关的条件语句，这些条件语句的出现，会导致代码的可维护性和灵活性变差，不能方便地增加和删除状态，使客户类与类库之间的耦合增强。在这些

条件语句中包含了对象的行为，而且这些条件对应于对象的各种状态。

4.11. 模式应用

状态模式在工作流或游戏等类型的软件中得以广泛使用，甚至可以用于这些系统的核心功能设计，如在政府OA办公系统中，一个批文的状态有多种：尚未办理；正在办理；正在批示；正在审核；已经完成等各种状态，而且批文状态不同时对批文的操作也有所差异。使用状态模式可以描述工作流对象（如批文）的状态转换以及不同状态下它所具有的行为。

4.12. 模式扩展

共享状态

- 在有些情况下多个环境对象需要共享同一个状态，如果希望在系统中实现多个环境对象实例共享一个或多个状态对象，那么需要将这些状态对象定义为环境的静态成员对象。

简单状态模式与可切换状态的状态模式

1. 简单状态模式：简单状态模式是指状态都相互独立，状态之间无须进行转换的状态模式，这是最简单的一种状态模式。对于这种状态模式，每个状态类都封装与状态相关的操作，而无须关心状态的切换，可以在客户端直接实例化状态类，然后将状态对象设置到环境类中。如果是这种简单的状态模式，它遵循“开闭原则”，在客户端可以针对抽象状态类进行编程，而将具体状态类写到配置文件中，同时增加新的状态类对原有系统也不造成任何影响。
2. 可切换状态的状态模式：大多数的状态模式都是可以切换状态的状态模式，在实现状态切换时，在具体状态类内部需要调用环境类Context的setState()方法进行状态的转换操作，在具体状态类中可以调用到环境类的方法，因此状态类与环境类之间通常还存在关联关系或者依赖关系。通过在状态类中引用环境类的对象来回调环境类的setState()方法实现状态的切换。在这种可以切换状态的状态模式中，增加新的状态类可能需要修改其他某些状态类甚至环境类的源代码，否则系统无法切换到新增状态。

4.13. 总结

- 状态模式允许一个对象在其内部状态改变时改变它的行为，对象看起来似乎修改了它的类。其别名为状态对象，状态模式是一种对象行为型模式。
- 状态模式包含三个角色：环境类又称为上下文类，它是拥有状态的对象，在环境类中维护一个抽象状态类State的实例，这个实例定义当前状态，在具体实现时，它是一个State子类的对象，可以定义初始状态；抽象状态类用于定义一个接口以封装与环境类的一个特定状态相关的行为；具体状态类是抽象状态类的子类，每一个子类实现一个与环境类的一个状态相关的行为，每一个具体状态类对应环境的一个具体状态，不同的具体状态类其行为有所不同。
- 状态模式描述了对对象状态的变化以及对象如何在每一种状态下表现出不同的行为。
- 状态模式的主要优点在于封装了转换规则，并枚举可能的状态，它将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为，还可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数；其缺点

在于使用状态模式会增加系统类和对象的个数，且状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱，对于可以切换状态的状态模式不满足“开闭原则”的要求。

- 状态模式适用情况包括：对象的行为依赖于它的状态（属性）并且可以根据它的状态改变而改变它的相关行为；代码中包含大量与对象状态有关的条件语句，这些条件语句的出现，会导致代码的可维护性和灵活性变差，不能方便地增加和删除状态，使客户类与类库之间的耦合增强。

5. 策略模式

目录

- 策略模式
 - 模式动机
 - 模式定义
 - 模式结构
 - 时序图
 - 代码分析
 - 模式分析
 - 实例
 - 优点
 - 缺点
 - 适用环境
 - 模式应用
 - 模式扩展
 - 总结

5.1. 模式动机

- 完成一项任务，往往可以有多种不同的方式，每一种方式称为一个策略，我们可以根据环境或者条件的不同选择不同的策略来完成该项任务。
- 在软件开发中也常常遇到类似的情况，实现某一个功能有多个途径，此时可以使用一种设计模式来使得系统可以灵活地选择解决途径，也能够方便地增加新的解决途径。
- 在软件系统中，有许多算法可以实现某一功能，如查找、排序等，一种常用的方法是硬编码(Hard Coding)在一个类中，如需要提供多种查找算法，可以将这些算法写到一个类中，在该类中提供多个方法，每一个方法对应一个具体的查找算法；当然也可以将这些查找算法封装在一个统一的方法中，通过if...else...等条件判断语句来进行选择。这两种实现方法我们都可以称之为硬编码，如果需要增加一种新的查找算法，需要修改封装算法类的源代码；更换查找算法，也需要修改客户端调用代码。在这个算法类中封装了大量查找算法，该类代码将较复杂，维护较为困难。
- 除了提供专门的查找算法类之外，还可以在客户端程序中直接包含算法代码，这种做法更不可取，将导致客户端程序庞大而且难以维护，如果存在大量可供选择的算法时问题将变得更加严重。
- 为了解决这些问题，可以定义一些独立的类来封装不同的算法，每一个类封装一个具体的算法，在这里，每一个封装算法的类我们都可以称之为策略(Strategy)，为了保证这些策略的一致性，一般会用一个抽象的策略类来做算法的定义，而具体每种算法则对应于一个具体策略类。

5.2. 模式定义

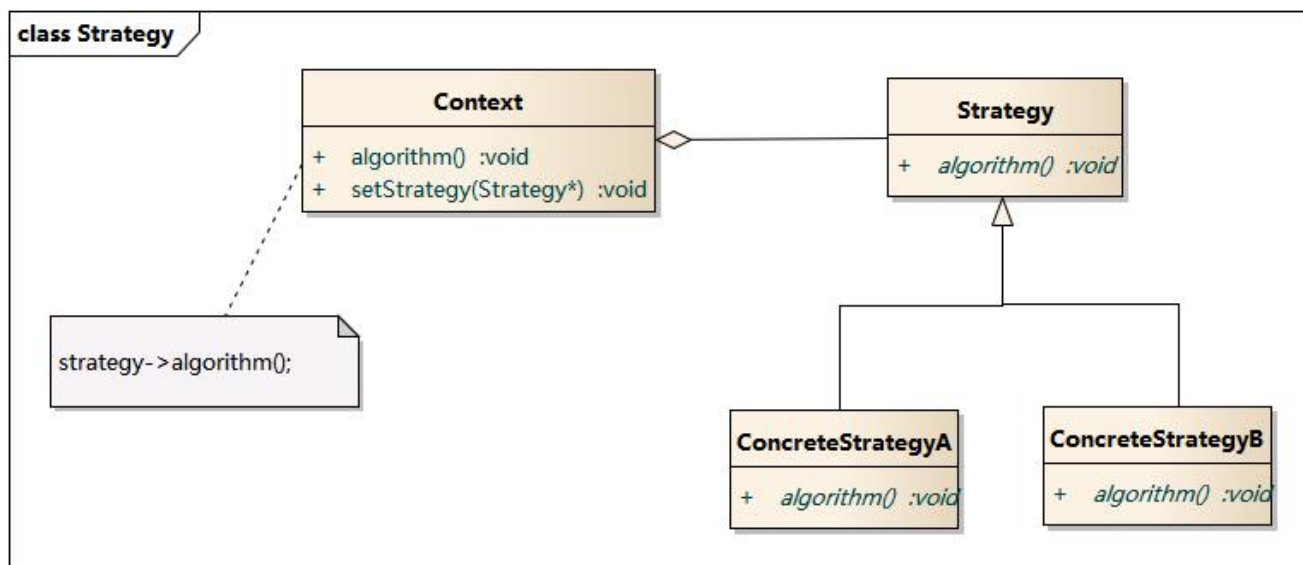
策略模式(Strategy Pattern): 定义一系列算法, 将每一个算法封装起来, 并让它们可以相互替换。策略模式让算法独立于使用它的客户而变化, 也称为政策模式(Policy)。

策略模式是一种对象行为型模式。

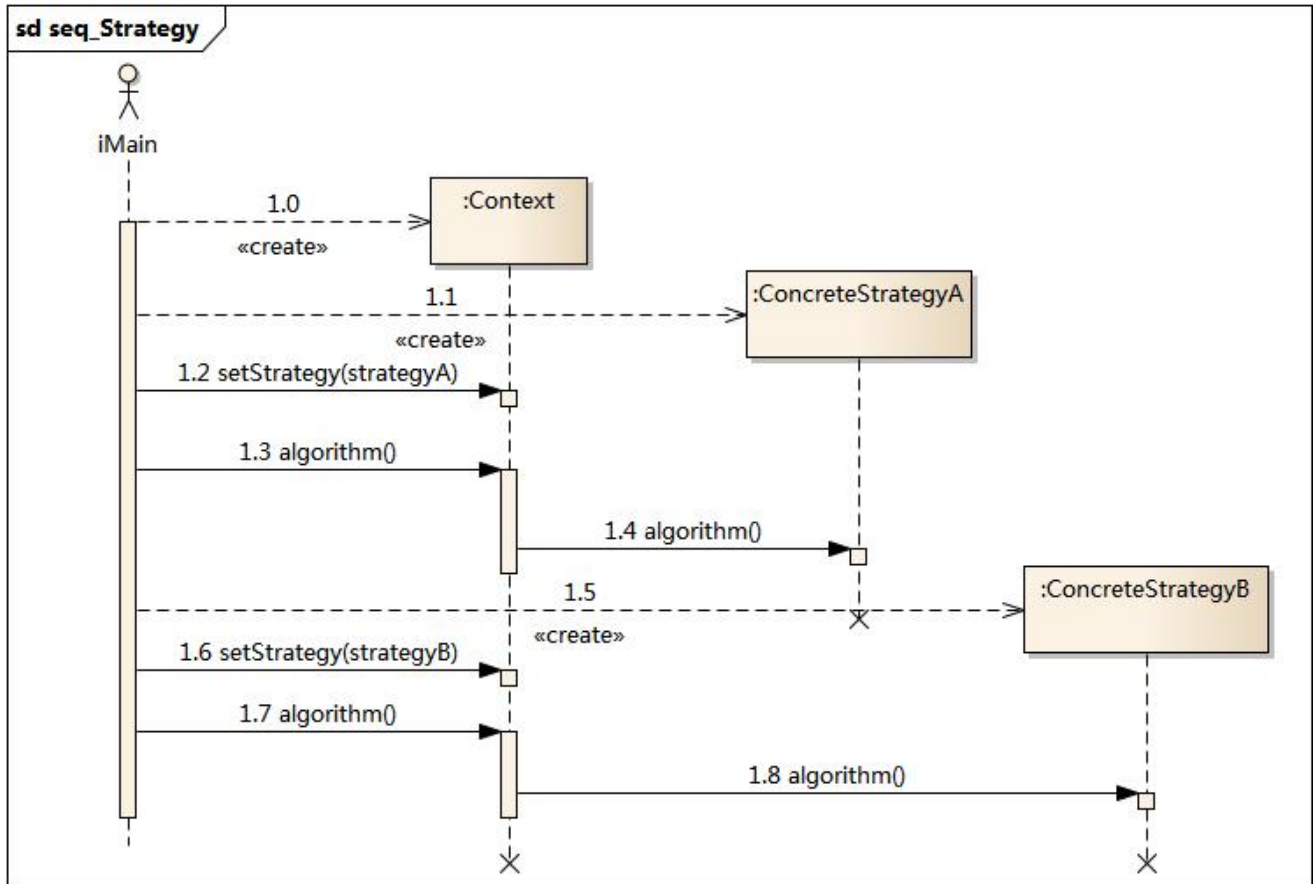
5.3. 模式结构

策略模式包含如下角色:

- Context: 环境类
- Strategy: 抽象策略类
- ConcreteStrategy: 具体策略类



5.4. 时序图



5.5. 代码分析

```
1  #include <iostream>
2  #include "Context.h"
3  #include "ConcreteStrategyA.h"
4  #include "ConcreteStrategyB.h"
5  #include "Strategy.h"
6  #include <vector>
7  using namespace std;
8
9  int main(int argc, char *argv[])
10 {
11     Strategy * s1 = new ConcreteStrategyA();
12     Context * cxt = new Context();
13     cxt->setStrategy(s1);
14     cxt->algorithm();
15
16     Strategy *s2 = new ConcreteStrategyB();
17     cxt->setStrategy(s2);
18     cxt->algorithm();
19
20     delete s1;
21     delete s2;
22
23     int rac1 = 0x1;
24     int rac2 = 0x2;
25     int rac3 = 0x4;
26     int rac4 = 0x8;
27
28     int i = 0xe;
29     int j = 0x5;
30
31     int r1 = i & rac1;
32     int r2 = i & rac2;
33     int r3 = i & rac3;
34     int r4 = i & rac4;
35
36     cout <<"res:" << r1 << "/" << r2 << "/" << r3 << "/" << r4 << endl;
37
38     return 0;
39 }
```

```

1  //////////////////////////////////////
2  // Context.h
3  // Implementation of the Class Context
4  // Created on:      09-十月-2014 22:21:07
5  // Original author: colin
6  //////////////////////////////////////
7  #if !defined(EA_0DA87730_4DEE_4392_9BAF_4AC64A8A07A4__INCLUDED_)
8  #define EA_0DA87730_4DEE_4392_9BAF_4AC64A8A07A4__INCLUDED_
9  #include "Strategy.h"
10
11 class Context
12 {
13 public:
14     Context();
15     virtual ~Context();
16
17     void algorithm();
18     void setStrategy(Strategy* st);
19 private:
20     Strategy *m_pStrategy;
21 };
22 #endif // !defined(EA_0DA87730_4DEE_4392_9BAF_4AC64A8A07A4__INCLUDED_)
23
24
25
26
27
28

```

```

1  //////////////////////////////////////
2  // Context.cpp
3  // Implementation of the Class Context
4  // Created on:      09-十月-2014 22:21:07
5  // Original author: colin
6  //////////////////////////////////////
7  #include "Context.h"
8
9  Context::Context(){
10 }
11
12 Context::~~Context(){
13 }
14
15 void Context::algorithm(){
16     m_pStrategy->algorithm();
17 }
18
19 void Context::setStrategy(Strategy* st){
20     m_pStrategy = st;
21 }
22

```

```

1  //////////////////////////////////////
2  //  ConcreteStrategyA.h
3  //  Implementation of the Class ConcreteStrategyA
4  //  Created on:      09-十月-2014 22:21:06
5  //  Original author: colin
6  //////////////////////////////////////
7  #if !defined(EA_9B180F12_677B_4e9b_A243_1F5DAD93FE1D__INCLUDED_)
8  #define EA_9B180F12_677B_4e9b_A243_1F5DAD93FE1D__INCLUDED_
9
10 #include "Strategy.h"
11
12 class ConcreteStrategyA : public Strategy
13 {
14 public:
15     ConcreteStrategyA();
16     virtual ~ConcreteStrategyA();
17
18     virtual void algorithm();
19 };
20 #endif // !defined(EA_9B180F12_677B_4e9b_A243_1F5DAD93FE1D__INCLUDED_)
21
22
23

```

```

1  //////////////////////////////////////
2  //  ConcreteStrategyA.cpp
3  //  Implementation of the Class ConcreteStrategyA
4  //  Created on:      09-十月-2014 22:21:07
5  //  Original author: colin
6  //////////////////////////////////////
7  #include "ConcreteStrategyA.h"
8  #include <iostream>
9  using namespace std;
10
11 ConcreteStrategyA::ConcreteStrategyA(){
12 }
13
14 ConcreteStrategyA::~~ConcreteStrategyA(){
15 }
16
17 void ConcreteStrategyA::algorithm(){
18     cout << "use algorithm A" << endl;
19 }
20
21
22

```

运行结果：

```
"C:\Users\cl\Documents\GitHub\design_patterns\code\Strategy\mingw5\main.exe"
use algorithm A
use algorithm B
请按任意键继续. . .
```

5.6. 模式分析

- 策略模式是一个比较容易理解和使用的设计模式，策略模式是对算法的封装，它把算法的责任和算法本身分割开，委派给不同的对象管理。策略模式通常把一个系列的算法封装到一系列的策略类里面，作为一个抽象策略类的子类。用一句话来说，就是“准备一组算法，并将每一个算法封装起来，使得它们可以互换”。
- 在策略模式中，应当由客户端自己决定在什么情况下使用什么具体策略角色。
- 策略模式仅仅封装算法，提供新算法插入到已有系统中，以及老算法从系统中“退休”的方便，策略模式并不决定在何时使用何种算法，算法的选择由客户端来决定。这在一定程度上提高了系统的灵活性，但是客户端需要理解所有具体策略类之间的区别，以便选择合适的算法，这也是策略模式的缺点之一，在一定程度上增加了客户端的使用难度。

5.7. 实例

5.8. 优点

策略模式的优点

- 策略模式提供了对“开闭原则”的完美支持，用户可以在不修改原有系统的基础上选择算法或行为，也可以灵活地增加新的算法或行为。
- 策略模式提供了管理相关的算法族的办法。
- 策略模式提供了可以替换继承关系的办法。
- 使用策略模式可以避免使用多重条件转移语句。

5.9. 缺点

策略模式的缺点

- 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。
- 策略模式将造成产生很多策略类，可以通过使用享元模式在一定程度上减少对象的数量。

5.10. 适用环境

在以下情况下可以使用策略模式：

- 如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。

- 一个系统需要动态地在几种算法中选择一种。
- 如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。
- 不希望客户端知道复杂的、与算法相关的数据结构，在具体策略类中封装算法和相关的数据结构，提高算法的保密性与安全性。

5.11. 模式应用

5.12. 模式扩展

策略模式与状态模式

- 可以通过环境类状态的个数来决定是使用策略模式还是状态模式。
- 策略模式的环境类自己选择一个具体策略类，具体策略类无须关心环境类；而状态模式的环境类由于外在因素需要放进一个具体状态中，以便通过其方法实现状态的切换，因此环境类和状态类之间存在一种双向的关联关系。
- 使用策略模式时，客户端需要知道所选的具体策略是哪一个，而使用状态模式时，客户端无须关心具体状态，环境类的状态会根据用户的操作自动转换。
- 如果系统中某个类的对象存在多种状态，不同状态下行为有差异，而且这些状态之间可以发生转换时使用状态模式；如果系统中某个类的某一行为存在多种实现方式，而且这些实现方式可以互换时使用策略模式。

5.13. 总结

- 在策略模式中定义了一系列算法，将每一个算法封装起来，并让它们可以相互替换。策略模式让算法独立于使用它的客户而变化，也称为政策模式。策略模式是一种对象行为型模式。
- 策略模式包含三个角色：环境类在解决某个问题时可以采用多种策略，在环境类中维护一个对抽象策略类的引用实例；抽象策略类为所支持的算法声明了抽象方法，是所有策略类的父类；具体策略类实现了在抽象策略类中定义的算法。
- 策略模式是对算法的封装，它把算法的责任和算法本身分割开，委派给不同的对象管理。策略模式通常把一个系列的算法封装到一系列的策略类里面，作为一个抽象策略类的子类。
- 策略模式主要优点在于对“开闭原则”的完美支持，在不修改原有系统的基础上可以更换算法或者增加新的算法，它很好地管理算法族，提高了代码的复用性，是一种替换继承，避免多重条件转移语句的实现方式；其缺点在于客户端必须知道所有的策略类，并理解其区别，同时在一定程度上增加了系统中类的个数，可能会存在很多策略类。
- 策略模式适用情况包括：在一个系统里面有许多类，它们之间的区别仅在于它们的行为，使用策略模式可以动态地让一个对象在许多行为中选择一种行为；一个系统需要动态地在几种算法中选择一种；避免使用难以维护的多重条件选择语句；希望在具体策略类中封装算法和与相关的数据结构。