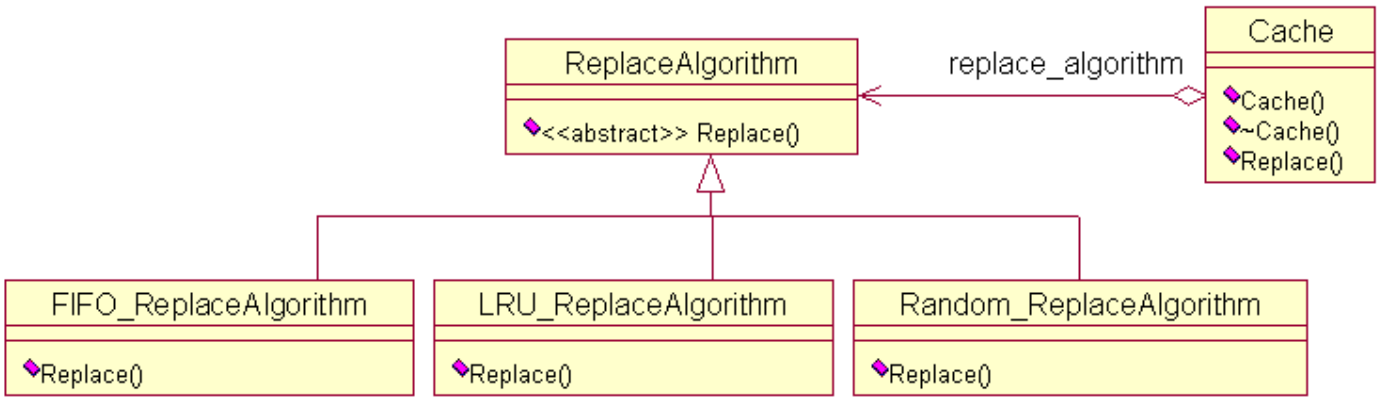
設計模式C++實現（２）——策略模式

星期六, 2013 12月 14, 12:59 上午

軟件領域中的設計模式為開發人員提供了一種使用專家設計經驗的有效途徑。設計模式中運用了面向對象編程語言的重要特性：封裝、繼承、多態，真正領悟設計模式的精髓是可能一個漫長的過程，需要大量實踐經驗的積累。最近看設計模式的書，對於每個模式，用C++寫了個小例子，加深一下理解。主要參考《大話設計模式》和《設計模式:可復用面向對象軟件的基礎》兩本書。本文介紹策略模式的實現。

策略模式是指定義一系列的算法，把它們一個個封裝起來，並且使它們可相互替換。本模式使得算法可獨立於使用它的客戶而變化。也就是說這些算法所完成的功能一樣，對外的接口一樣，只是各自實現上存在差異。用策略模式來封裝算法，效果比較好。下面以高速緩存（Cache）的替換算法為例，實現策略模式。

什麼是Cache的替換算法呢？簡單解釋一下， 當發生Cache缺失時，Cache控制器必須選擇Cache中的一行，並用欲獲得的數據來替換它。改採用的選擇策略就是Cache的替換算法。下面給出相應的UML圖。



ReplaceAlgorithm是一個抽象類，定義了算法的接口，有三個類繼承自這個抽象類，也就是具體的算法實現。Cache類中需要使用替換算法，因此維護了一個 ReplaceAlgorithm的對象。這個UML圖的結構就是策略模式的典型結構。下面根據UML圖，給出相應的實現。

首先給出替換算法的定義。

```
1.  //抽象接口
2.  class ReplaceAlgorithm
3.  {
4.  public:
5.      virtual void Replace() = 0;
6.  };
7.  //三種具體的替換算法
8.  class LRU_ReplaceAlgorithm : public ReplaceAlgorithm
9.  {
10. public:
11.     void Replace() { cout<<"Least Recently Used replace algorithm"<<endl; }
12. };
13.
14. class FIFO_ReplaceAlgorithm : public ReplaceAlgorithm
15. {
16. public:
17.     void Replace() { cout<<"First in First out replace algorithm"<<endl; }
18. };
19. class Random_ReplaceAlgorithm: public ReplaceAlgorithm
20. {
21. public:
22.     void Replace() { cout<<"Random replace algorithm"<<endl; }
```

```
23. };
```

接著給出Cache的定義，這裡很關鍵，Cache的實現方式直接影響了客戶的使用方式，其關鍵在於如何指定替換算法。

方式一：直接通過參數指定，傳入一個特定算法的指針。

```
1. //Cache需要用到替換算法
2. class Cache
3. {
4. private:
5.     ReplaceAlgorithm *m_ra;
6. public:
7.     Cache(ReplaceAlgorithm *ra) { m_ra = ra; }
8.     ~Cache() { delete m_ra; }
9.     void Replace() { m_ra->Replace(); }
10. };
```

如果用這種方式，客戶就需要知道這些算法的具體定義。只能以下面這種方式使用，可以看到暴露了太多的細節。

```
1. int main()
2. {
3.     Cache cache(new LRU_ReplaceAlgorithm()); //暴露了算法的定義
4.     cache.Replace();
5.     return 0;
6. }
```

方式二：也是直接通過參數指定，只不過不是傳入指針，而是一個標籤。這樣客戶只要知道算法的相應標籤即可，而不需要知道算法的具體定義。

```
1. //Cache需要用到替換算法
2. enum RA {LRU, FIFO, RANDOM}; //標籤
3. class Cache
4. {
5. private:
6.     ReplaceAlgorithm *m_ra;
7. public:
8.     Cache(enum RA ra)
9.     {
10.        if(ra == LRU)
11.            m_ra = new LRU_ReplaceAlgorithm();
12.        else if(ra == FIFO)
13.            m_ra = new FIFO_ReplaceAlgorithm();
14.        else if(ra == RANDOM)
15.            m_ra = new Random_ReplaceAlgorithm();
16.        else
17.            m_ra = NULL;
18.    }
19.    ~Cache() { delete m_ra; }
```

```
20.     void Replace() { m_ra->Replace(); }
21. };
```

相比方式一，這種方式用起來方便多了。其實這種方式將簡單工廠模式與策略模式結合在一起，算法的定義使用了策略模式，而Cache的定義其實使用了簡單工廠模式。

```
1. int main()
2. {
3.     Cache cache(LRU); //指定標籤即可
4.     cache.Replace();
5.     return 0;
6. }
```

上面兩種方式，構造函數都需要形參。構造函數是否可以不用參數呢？下面給出第三種實現方式。

方式三：利用模板實現。算法通過模板的實參指定。當然了，還是使用了參數，只不過不是構造函數的參數。在策略模式中，參數的傳遞難以避免，客戶必須指定某種算法。

```
1. //Cache需要用到替換算法
2. template <class RA>
3. class Cache
4. {
5. private:
6.     RA m_ra;
7. public:
8.     Cache() { }
9.     ~Cache() { }
10.    void Replace() { m_ra.Replace(); }
11. };
```

使用方式如下：

```
1. int main()
2. {
3.     Cache<Random_ReplaceAlgorithm> cache; //模板實參
4.     cache.Replace();
5.     return 0;
6. }
```