命令模式 - 海上餐厅巴拉蒂

发表于 2022-09-14 | 更新于 2023-04-06 | 设计模式

| 字数总计: 2.5k | 阅读时长: 8 分钟 | 阅读量: 402 | 评论数: 0

配套视频课程已更新完毕，大家可通过以下两种方式观看视频讲解：

关注公众号： 爱编程的大丙 ，或者进入 大丙课堂 学习。

苏丙榅

合抱之木，生于毫末；九层之台，起于垒土；千里之行，始于足下。

1. 海上餐厅

在海贼世界中，巴拉蒂是位于东海桑巴斯海域的一个海上餐厅。外形是条巨型的船，船两头有鱼形状的船首，整艘船能转换成战斗状态。 巴拉蒂是东海最有名的餐厅 ，有不少人特地为了品尝老板兼主厨哲普所做的美味料理而来到这里，甚至连海军的重要角色都会来这里吃饭。

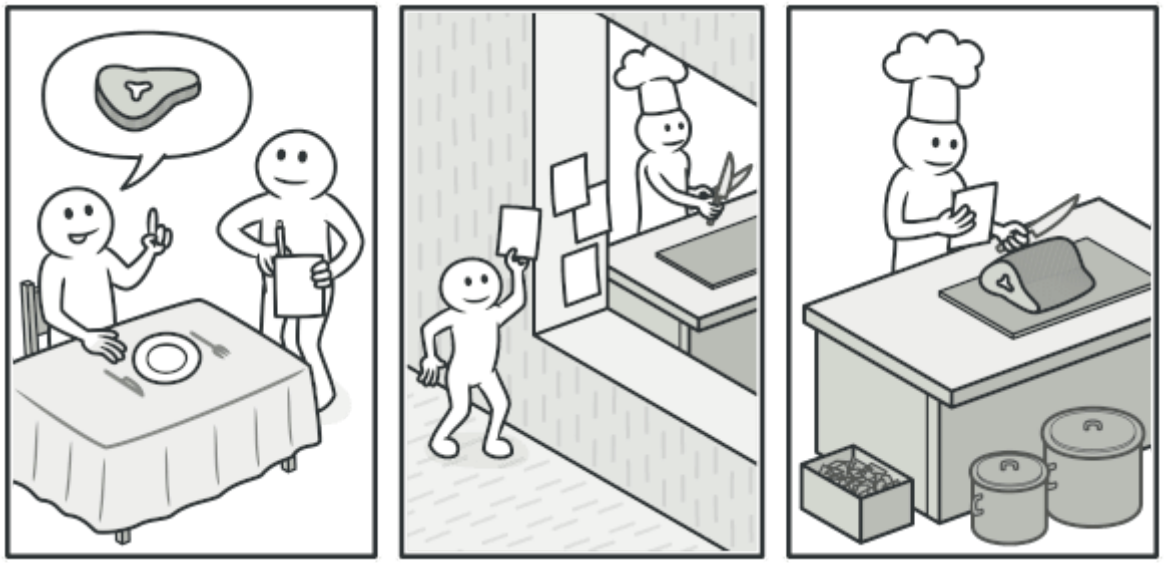
当出身东海的路飞路过巴拉蒂餐厅的时候，由于损坏了餐厅的财物，被强行扣留下来打工以此偿还他造成的损失，替自己赎身。这天来了几个人吃饭，路飞作为服务员接待他们点餐。意料之中，他又把事儿搞砸了，下面作为程序猿的我打算替路飞写一个点餐的小程序，先来分析需求：

1. 允许顾客点多个菜，点餐完毕后，厨师才开始制作

目录

最新文章

2. 点餐过程中需要及时提醒顾客，这个菜现在是不是可以制作（可能原材料用完了）
3. 需要有点餐记录，结账的时候用
4. 顾客可以取消已下单但是还没有制作的菜



如果想要实现上述的需求，需要在程序中准备如下几个对象：

1. 替顾客下单的 服务员路飞
2. 给顾客炒菜的 厨师哲普
3. 由路飞写好的 顾客点餐列表

我们可以将 顾客的点餐列表看作是一个待执行的命令的列表，这样就可以总结出三者之间的关系了：厨师哲普是这些命令的接收者和执行者，路飞是这些命令的调用者。 如果没有这张点餐列表，路飞需要

CMake 保姆级教程（下）
2023-03-15

CMake 保姆级教程（上）
2023-03-06

访问者模式 - 再见，香波地群岛
2022-09-22

模板方法模式 - 和平主义者
2022-09-21

状态模式 - 文斯莫克・山治
2022-09-20

非常频繁地穿梭在餐厅与厨房之间，而且哪个顾客点了什么菜也容易弄混，从某种程度上讲这个点餐列表就相当于一个任务队列。

上面的这种解决问题的思路用到的就是设计模式中的命令模式。命令模式就是将请求转换为一个包含与请求相关的所有信息的独立对象，通过这个转换能够让使用者根据不同的请求将客户参数化、延迟请求执行或将请求放入队列中或记录请求日志， 且能实现可撤销操作。

2. 慕名而来

2.1 厨师哲普

哲普作为一个厨师可能会制作很多的美食，在定义哲普对应的类的时候，每道菜都应该对应一个处理函数，所以这个类应该是这样的：

C++

```
// 厨师哲普
class CookerZeff
{
public:
    void makeDSX()
    {
        cout << "开始烹饪地三鲜...";
    }
    void makeGBJD()
    {
        cout << "开始烹饪宫保鸡丁...";
```

```
    }
    void makeYXRS()
    {
        cout << "开始烹饪鱼香肉丝...";
    }
    void makeHSPG()
    {
        cout << "开始烹饪红烧排骨...";
    }
};
```

这个厨师类是命令模式中命令的接收者，收不到命令厨师是不能工作的。

2.2 下单

在哲普制作美食之前，需要顾客先下单，在命令模式中顾客每点一道美食对应的就是一个命令，虽然每次点的食物不同，但点餐这个动作是不变的。因此我们可以先定义一个关于点餐命令的基类：

C++

```
// 点餐的命令 - 抽象类
class AbstractCommand
{
public:
    AbstractCommand(CookerZeff* receiver) : m_cooker(receiver) {}
    virtual void excute() = 0;
    virtual string name() = 0;
    ~AbstractCommand() {}
protected:
```

```
    CookerZeff* m_cooker = nullptr;
};
```

在这个抽象类中关联了一个厨师对象 `CookerZeff* m_cooker`，有了这个厨师对象就可以去执行对应的炒菜的动作了 `excute()`。基于这个抽象的基类就可以派生出若干子类，在子类中让厨师去炒菜，也就是重写 `excute()`。

C++

```cpp
// 地三鲜的命令
class DSXCommand : public AbstractCommand
{
public:
    using AbstractCommand::AbstractCommand;
    void excute() override
    {
        m_cooker->makeDSX();
    }
    string name() override
    {
        return "地三鲜";
    }
};

// 宫保鸡丁的命令
class GBJDCommand : public AbstractCommand
{
public:
    using AbstractCommand::AbstractCommand;
    void excute() override
    {
```

```
        m_cooker->makeGBJD();
    }
    string name() override
    {
        return "宫保鸡丁";
    }
};

// 鱼香肉丝的命令
class YXRSCommand : public AbstractCommand
{
public:
    using AbstractCommand::AbstractCommand;
    void excute() override
    {
        m_cooker->makeYXRS();
    }
    string name() override
    {
        return "鱼香肉丝";
    }
```

可以看到在这四个子类中，分别重写父类的纯虚函数 `excute()`，在该函数内部通过关联的厨师对象分别制作出了 地三鲜、宫保鸡丁、鱼香肉丝、红烧排骨 。

顾客下单就是命令模式中的命令，这些命令的接收者是厨师，命令被分离出来实现了和厨师类的解耦合。通过这种方式可以控制命令执行的时机，毕竟厨师都是在顾客点餐完毕之后才开始炒菜的。

2.3 服务员路飞

顾客点餐并不直接和厨师产生交集，而是通过服务员完成的，所以通过服务员类需要实现 点餐、沟通、取消订单、结账 等功能，下面是关于路飞这个服务员类的定义：

C++

```cpp
// 服务器路飞 - 命令的调用者
class WaiterLuffy
{
public:
    // 下单
    void setOrder(int index, AbstractCommand* cmd)
    {
        cout << index << "号桌点了" << cmd->name() << endl;
        if (cmd->name() == "鱼香肉丝")
        {
            cout << "没有鱼了, 做不了鱼香肉丝, 点个别的菜吧..." << endl;
            return;
        }
        // 没找到该顾客
        if (m_cmdList.find(index) == m_cmdList.end())
        {
            list<AbstractCommand*> mylist{ cmd };
            m_cmdList.insert(make_pair(index, mylist));
        }
        else
        {
            m_cmdList[index].push_back(cmd);
        }
    }
```

```
25      // 取消订单
26      void cancelOrder(int index, AbstractCommand* cmd)
27      {
28          if (m_cmdList.find(index) ≠ m_cmdList.end())
29          {
30              m_cmdList[index].remove(cmd);
31              cout << index << "号桌, 撤销了" << cmd→name() << endl;
32          }
33      }
34      // 结账
35      void checkOut(int index)
36      {
37          cout << "第[" << index << "]号桌的顾客点的菜是: 【";
38          for (const auto& item : m_cmdList[index])
39          {
40              cout << item→name();
41              if (item ≠ m_cmdList[index].back())
42              {
43                  cout << ", ";
```

在路飞对应的服务员类中，通过一个 `map` 容器保存了所有顾客的下单信息， `key` 值是顾客就餐的餐桌编号， `value` 值存储的是顾客所有的点餐信息。并且这个 `value` 是一个 `list` 容器，用于存储某个顾客的所有的点餐信息。

顾客点餐的时候，每点一个菜都会对应一个 `AbstractCommand*` 类型的命令对象，这个类有很多子类，在容器中实际存储的是这个类的子类对象，此处用到了多态。

在命令模式中，服务员类是命令的调用者，顾客点餐完成之后服务员调用这些命令，命令的接收者也是执行者 – 厨师就开始给顾客做菜了。

2.4 大快朵颐

万事俱备只欠东风，点餐结束经过短暂的等待，就可以享用美食了：

C++

```cpp
int main()
{
    CookerZeff* cooker = new CookerZeff;
    WaiterLuffy* luffy = new WaiterLuffy;

    YXRSCommand* yxrs = new YXRSCommand(cooker);
    GBJDCommand* gbjd = new GBJDCommand(cooker);
    DSXCommand* dsx = new DSXCommand(cooker);
    HSPGCommand* hspg = new HSPGCommand(cooker);

    cout << "================== 开始点餐 ==================" << endl;
    luffy->setOrder(1, yxrs);
    luffy->setOrder(1, dsx);
    luffy->setOrder(1, gbjd);
    luffy->setOrder(1, hspg);
    luffy->setOrder(2, dsx);
    luffy->setOrder(2, gbjd);
    luffy->setOrder(2, hspg);
    cout << "================== 撤销订单 ==================" << endl;
    luffy->cancelOrder(1, dsx);
    cout << "================== 开始烹饪 ==================" << endl;
```

```
    luffy->notify(1);
    luffy->notify(2);
    cout << "==================== 结账 ====================" << endl;
    luffy->checkOut(1);
    luffy->checkOut(2);

    return 0;
}
```

程序输出的结果:

C++

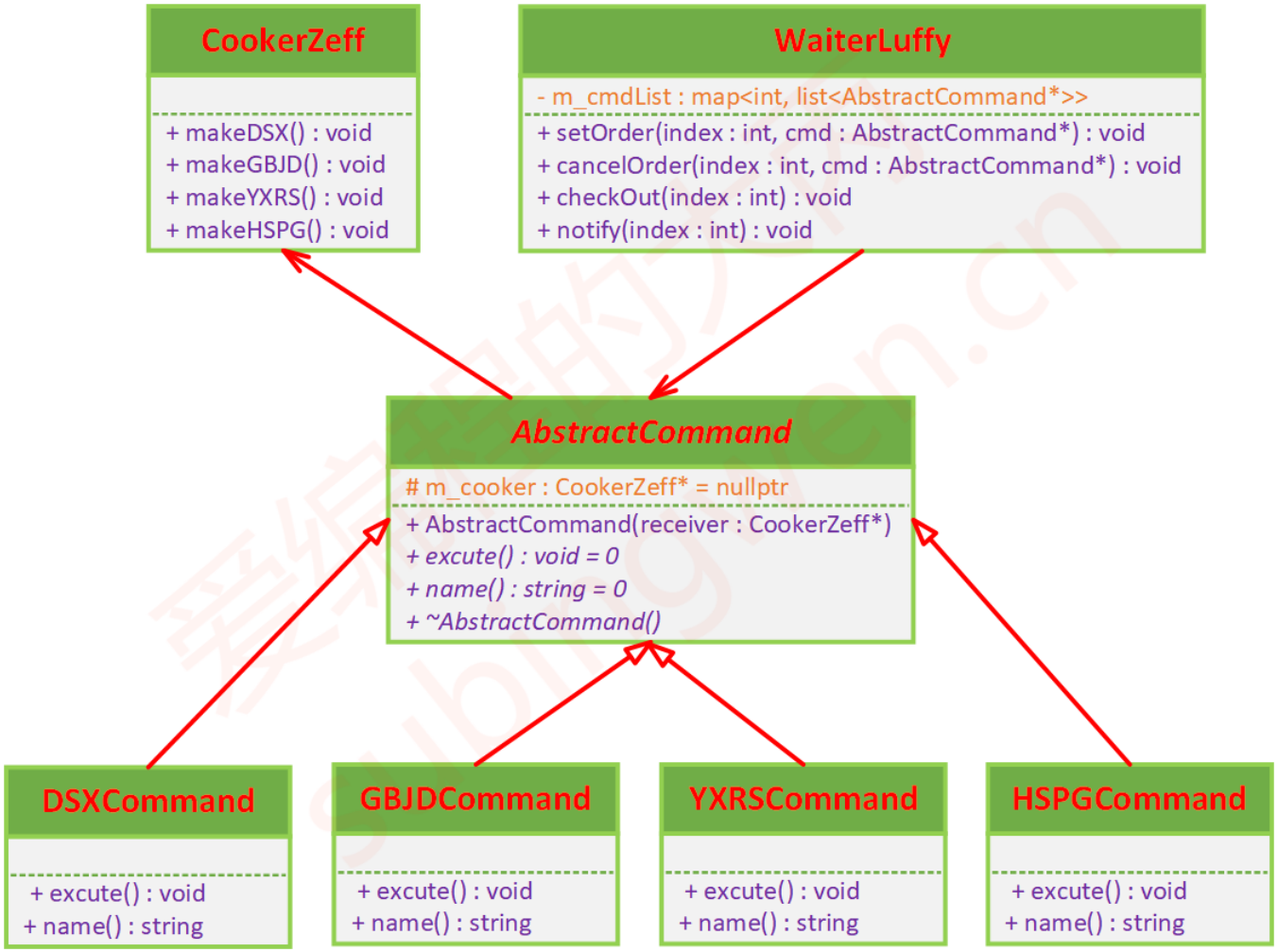
```
==================== 开始点餐 ====================
1号桌点了鱼香肉丝
没有鱼了，做不了鱼香肉丝，点个别的菜吧...
1号桌点了地三鲜
1号桌点了宫保鸡丁
1号桌点了红烧排骨
2号桌点了地三鲜
2号桌点了宫保鸡丁
2号桌点了红烧排骨
==================== 撤销订单 ====================
1号桌，撤销了地三鲜
==================== 开始烹饪 ====================
开始烹饪宫保鸡丁...1号桌
开始烹饪红烧排骨...1号桌
开始烹饪地三鲜...2号桌
开始烹饪宫保鸡丁...2号桌
开始烹饪红烧排骨...2号桌
==================== 结账 ====================
```

```
第[1]号桌的顾客点的菜是：【宫保鸡丁，红烧排骨】
第[2]号桌的顾客点的菜是：【地三鲜，宫保鸡丁，红烧排骨】
```

有了这款点餐软件，路飞表示再也没有因为点餐出错而被扣工资了。

3. 结构图

最后根据上面的例子把对应的 UML 类图画一下（学会了命令模式之后，应该先画 UML 类图，再写程序。）



命令模式最大的特点就是松耦合设计，它有以下几个优势：

1. 使用这种模式可以很容易地设计出一个命令队列 （对应路飞类中的点餐列表）
2. 可以很容易的将命令记录到日志中 （对应例子中的账单信息）
3. 允许接收请求的一方决定是否要否决请求 （对应例子中的鱼香肉丝）
4. 可以很容易的实现对请求的撤销和重做 （对应例子中的撤单函数）

设计模式

打赏

相关推荐

评论

昵称 邮箱 网址(http://)

来都来了, 说点什么吧...

提交

来发评论吧 ~

Powered By Valine
v1.5.1