

享元模式 - 铁拳卡普

📅 发表于 2022-09-10 | ⌚ 更新于 2023-04-06 | 📄 设计模式

| 📄 字数总计: 3.8k | ⌚ 阅读时长: 13 分钟 | 👁 阅读量: 350 | 💬 评论数: 1



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。



苏丙楦

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. 拳骨陨石

蒙奇·D·卡普，也被称为“铁拳卡普”，是海军中的传奇人物，相传卡普数次将海贼王罗杰逼入绝境，被誉为“海军英雄”。卡普是路飞的爷爷，经常将霸气缠绕在拳头上来打艾斯与路飞的头，因为攻击中充满了爱，所以名为“爱之铁拳”，每次都会在路飞头上留下包。



文章	标签	分类
134	37	12

大丙课堂



公告

微信公众号 爱编程的大丙 和
大丙课堂 上线了，可
点击上方 图标关注 ~ ~ ~

三 目录

1. 拳骨陨石
2. 设计炮弹
3. 结构图

🕒 最新文章

虽然自己是海军，但是儿子却是革命军，孙子是海贼。**拳骨陨石·流星群**是卡普的战斗招式之一，即连续不断地向敌人投掷炮弹，其效果就如流星砸向敌人一般。卡普曾以此招在水之七都向路飞告别，我们来回顾一下当时的场景：



关于卡普老爷子的实力是毋庸置疑的，假设我们现在是负责游戏开发的程序猿，要复刻这段场景，其中出现频率最高的就是炮弹。这里有一个很现实的亟待解决的问题：**内存的消耗问题**。

- 每个炮弹都是一个对象，每个对象都会占用一块内存
- 炮弹越多，占用的内存就越大，如果炮弹足够多可能会出现内存枯竭问题
- 假设内存足够大，频繁地创建炮弹对象，会影响游戏的流畅度，性能低

关于游戏中的炮弹，应该有以下一些需要处理的属性：



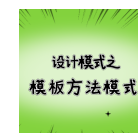
CMake 保姆级教程
(下)
2023-03-15



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

1. 炮弹的坐标
2. 炮弹的速度
3. 炮弹的颜色渲染
4. 炮弹的精灵图 (就是一张大图上有很多小的图片, 通过进行位置的控制, 从大图中取出想要的某一张小的图片)



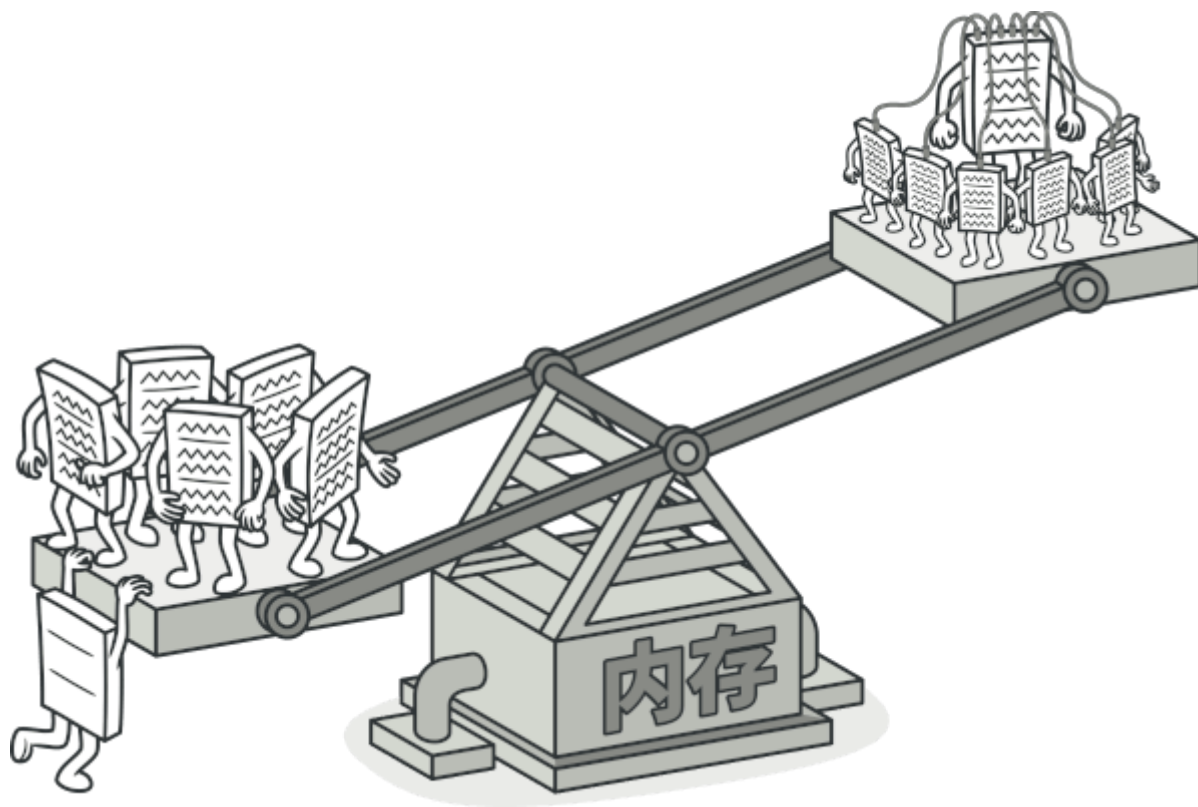
上面就是一张关于炮弹爆炸过程的精灵图片, 假设一张炮弹的精灵图片有 200k, 那么 1000 个这样的炮弹占用的内存就是 200M, 这对于内存的消耗是非常惊人的。

游戏中制作一个炮弹需要的数据在上面已经分析出来了, 在这四部分数据中有些属性是动态的, 有些属性是静态的:

- 静态资源: 精灵图 和 渲染的颜色
- 动态属性: 坐标 和 速度

对应的动态资源肯定是不能被复用，所有炮弹可共享的就是这些静态资源，不论有多少炮弹，它们对应的 **精灵图** 和 **渲染颜色** 数据可以只有一份，这样对于内存的开销就大大降低了。

上面的这种设计思路和设计模式中的享元模式很类似，**享元模式就是摒弃了在每个对象中都保存所有的数据的这种方式，通过数据共享（缓存）让有限的内存可以加载更多的对象。**



在上图跷跷板的左侧每个对象使用的都是独立内存，而右侧所有对象都共同使用了某一部分内存，所以左侧重右侧轻，左侧占用内存多，右侧占用内存少。

享元模式和线程池比较类似，线程池可以 **复用线程**，有效避免了线程的频繁创建和销毁，减少了性能消耗并提高了工作效率。享元模式中的共享内存也可以将其称之为缓存，这种模式中共享的是

对象。

对象的常量数据通常被称为内在状态，其位于对象中，其他对象只能读取但不能修改其数值。而对象的其他状态常常能被其他对象“从外部”改变，因此被称为外在状态。使用享元模式一般建议将内在状态和外在状态分离，将内在状态单独放到一个类中，这种类我们可以将其称之为享元类。

2. 设计炮弹

2.1 炸弹弹体

炮弹的共享数据其实就是享元模式中的享元数据，先定义一个享元数据类：

```
✓ C++  
1 // 共享数据类  
2 class SharedBombBody  
3 {  
4 public:  
5     SharedBombBody(string sprite) : m_sprite(sprite)  
6     {  
7         cout << "正在创建 <" << m_sprite << ">..." << endl;  
8     }  
9     void move(int x, int y, int speed)  
10    {  
11        cout << "炸弹以每小时" << speed << "速度飞到了("  
12            << x << ", " << y << ") 的位置..." << endl;  
13    }  
14    void draw(int x, int y)
```

```
15     {
16         cout << "在 (" << x << ", " << y << ") 的位置重绘炸弹弹体..." << endl;
17     }
18
19 private:
20     string m_sprite;    // 精灵图片
21     string m_color = string("black");    // 渲染颜色
22 };
```

通过构造函数得到精灵图片之后，该类对象中的数据就不会再发生任何变化了。

🔗2.2 炸弹

有了炸弹的弹体，卡普就可以基于这部分静态资源扔一枚炮弹出去了，先定义一个发射炸弹的类：

▼ C++

```
1 // 发射炮弹
2 class LaunchBomb
3 {
4 public:
5     LaunchBomb(SharedBombBody* body) : m_bomb(body) {}
6     int getX()
7     {
8         return m_x;
9     }
10    int getY()
11    {
12        return m_y;
13    }
```

```
14     void setSpeed(int speed)
15     {
16         m_speed = speed;
17     }
18     int getSpeed()
19     {
20         return m_speed;
21     }
22     void move(int x, int y)
23     {
24         m_x = x;
25         m_y = y;
26         m_bomb→move(m_x, m_y, m_speed);
27         draw();
28     }
29     void draw()
30     {
31         m_bomb→draw(m_x, m_y);
32     }
33
34 private:
35     int m_x = 0;
36     int m_y = 0;
37     int m_speed = 100;
38     SharedBombBody* m_bomb = nullptr;
39 };
```

由于发射出的每一枚型号相同的炮弹它们的外形都是相同的，所以这些炸弹可以共享同一个弹体对象（在类内部没有创建 **SharedBombBody** 类对象）。对于炸弹被发射出去之后它的坐标以及速度肯定是会变化的，所以在上面的 **LaunchBomb** 类中添加了对应的 **get** 和 **set** 方法。

🔗 2.3 彩蛋

在很多游戏中，由于玩家触发了某个条件，此时系统会赠送给玩家一个彩蛋，这个彩蛋一般都是独一无二的。假设卡普在投掷炸弹的过程中，路飞通过自己的橡胶果实能力连续接住了 10 个炸弹，并将其反弹出去，这个时候卡普投出的某一个炸弹就会变成一个彩蛋（卡普最后扔出的超巨型炸弹也可以视为是一个彩蛋），对于彩蛋的处理我们的分析如下：

- 这个彩蛋拥有和炸弹不一样的外观（使用的精灵图不同）
- 不论是炸弹还是彩蛋，对于卡普来说对它们的处理动作是一样的
- 炸弹爆炸会造成伤害，彩蛋爆炸会给玩家提供奖励或者造成非常严重的伤害或开启一段支线剧情等

通过上述的三点分析，我们可以得出结论，彩蛋和炸弹有相同的处理动作，只不过在细节处理上略有不同，对于这种情况，我们一般会提供一个抽象的基类并在这个类中提供一套虚操作函数，这样在子类中就可以重写父类提供的虚函数，提供不同的处理动作了。

▼ C++

```
1 // 享元基类
2 class FlyweightBody
3 {
4 public:
5     FlyweightBody(string sprite) : m_sprite(sprite) {}
6     virtual void move(int x, int y, int speed) = 0;
7     virtual void draw(int x, int y) = 0;
8     virtual ~FlyweightBody() {}
9 protected:
10     string m_sprite;    // 精灵图片
11     string m_color = "black";    // 渲染颜色
```

```
12 };
13
14 // 炸弹弹体
15 class SharedBombBody : public FlyweightBody
16 {
17 public:
18     using FlyweightBody::FlyweightBody;
19     void move(int x, int y, int speed) override
20     {
21         cout << "炸弹以每小时" << speed << "速度飞到了("
22             << x << ", " << y << ") 的位置..." << endl;
23     }
24     void draw(int x, int y) override
25     {
26         cout << "在 (" << x << ", " << y << ") 的位置重绘炸弹弹体..." << endl;
27     }
28 };
29
30 // 唯一的炸弹彩蛋
31 class UniqueBomb : public FlyweightBody
32 {
33 public:
34     using FlyweightBody::FlyweightBody;
35     void move(int x, int y, int speed) override
36     {
37         // 此处省略对参数 x, y, speed 的处理
38         cout << "彩蛋在往指定位置移动, 准备爆炸发放奖励..." << endl;
39     }
40     void draw(int x, int y) override
41     {
42         cout << "在 (" << x << ", " << y << ") 的位置重绘彩蛋运动轨迹..." << er
```

```
43     }  
44 };
```

一般享元数据都是共享的，但是这里的 `UniqueBomb` 类，它虽然是享元类的子类，但这个类的实例对象却是不共享数据（假设每个彩蛋的外观和用途都是不同的），表面看起来矛盾，但是也合乎常理。尽管我们大部分时间都需要共享对象来降低内存的损耗，但在个别时候也有可能不需要共享的数据，此时 `UniqueBomb` 子类就有存在的必要了，它可以帮助我们解决那些不需要共享对象场景下的问题，使用这种处理方式对应的操作流程是无需做出任何改变的。如果有上述的需求，就可以和示例代码中一样给享元类提供一个基类。

🔗 2.4 享元工厂

假设炮弹有很多种型号，此时就需要有很多张精灵图，也就是说 `SharedBombBody` 类型的对象对应也应该有很多个，此时我们就可以再添加一个享元工厂类，专门用来生产这些共享的享元类对象。

```
▼                                CPP                                📄  
  
1  // 享元工厂类  
2  class BombBodyFactory  
3  {  
4  public:  
5      SharedBombBody* getSharedData(string name)  
6      {  
7          SharedBombBody* data = nullptr;  
8          // 遍历容器  
9          for (auto item : m_bodyMap)  
10         {  
11             if (item.first == name)
```

```
12         {
13             // 找到了
14             data = item.second;
15             cout << "正在复用 <" << name << ">..." << endl;
16             break;
17         }
18     }
19     if (data == nullptr)
20     {
21         data = new SharedBombBody(name);
22         cout << "正在创建 <" << name << ">..." << endl;
23         m_bodyMap.insert(make_pair(name, data));
24     }
25     return data;
26 }
27 ~BombBodyFactory()
28 {
29     for (auto item : m_bodyMap)
30     {
31         delete item.second;
32     }
33 }
34 private:
35     map<string, SharedBombBody*> m_bodyMap;
36 };
```

在享元工厂内部有一个 **map 容器**，用于存储各种型号的炮弹的享元数据，这个享元工厂就相当于一个 **对象池**，当调用了 **getSharedData(string name)** 函数之后，如果能够从 **map 容器** 找到 **name** 对应的享元对象就返回该对象，如果找不到就创建一个新的享元对象并储存起来，这样就可以实现对象的复用了。

🔗2.5 发射炮弹


最后就可以把炮弹制作出来并让其在游戏中按照指定的轨迹运动了:

```
▼ C++  
  
1  int main()  
2  {  
3      // 发射炮弹  
4      BombBodyFactory* factory = new BombBodyFactory;  
5      vector<LaunchBomb*> ballList;  
6      vector<string> namelist = { "撒旦-1", "撒旦-1", "撒旦-2", "撒旦-2", "撒旦-  
7      for (auto name : namelist)  
8      {  
9          int x = 0, y = 0;  
10         LaunchBomb* ball = new LaunchBomb(factory->getSharedData(name));  
11         for (int i = 0; i < 3; ++i)  
12         {  
13             x += rand() % 100;  
14             y += rand() % 50;  
15             ball->move(x, y);  
16         }  
17         cout << "=====" << endl;  
18         ballList.push_back(ball);  
19     }  
20     // 彩蛋  
21     UniqueBomb* unique = new UniqueBomb("大彩蛋");  
22     LaunchBomb* bomb = new LaunchBomb(unique);  
23     int x = 0, y = 0;  
24     for (int i = 0; i < 3; ++i)  
25     {
```

```
26         x += rand() % 100;
27         y += rand() % 50;
28         bomb→move(x, y);
29     }
30
31     for (auto ball : ballList)
32     {
33         delete ball;
34     }
35     delete factory;
36     delete unique;
37     delete bomb;
38     return 0;
39 }
```

上面的测试程序就相当于在游戏中，卡普扔出了 6 个炸弹和一个彩蛋，不论是炸弹还是彩蛋都可以通过 **LaunchBomb** 类 进行处理，这个类的构造函数在接收实参的时候实际上就是一个多态的应用。

程序的输出的结果：

▼ C++ 

```
1 正在创建 <撒旦-1> ...
2 炸弹以每小时100速度飞到了(41, 17) 的位置 ...
3 在 (41, 17) 的位置重绘炸弹弹体 ...
4 炸弹以每小时100速度飞到了(75, 17) 的位置 ...
5 在 (75, 17) 的位置重绘炸弹弹体 ...
6 炸弹以每小时100速度飞到了(144, 41) 的位置 ...
7 在 (144, 41) 的位置重绘炸弹弹体 ...
8 =====
9 正在复用 <撒旦-1> ...
```



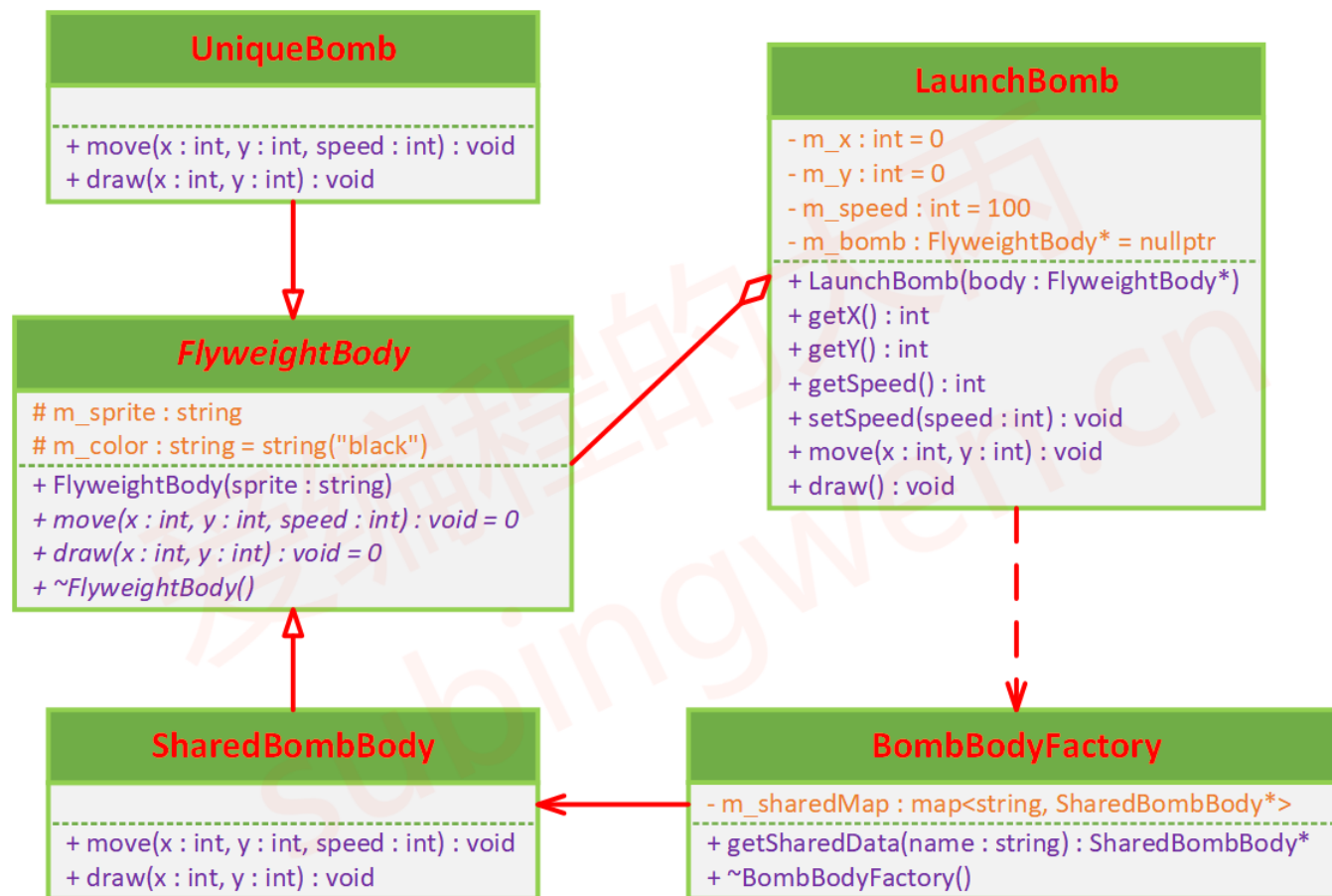
```
10 炸弹以每小时100速度飞到了(78, 8) 的位置 ...
11 在 (78, 8) 的位置重绘炸弹弹体 ...
12 炸弹以每小时100速度飞到了(140, 22) 的位置 ...
13 在 (140, 22) 的位置重绘炸弹弹体 ...
14 炸弹以每小时100速度飞到了(145, 67) 的位置 ...
15 在 (145, 67) 的位置重绘炸弹弹体 ...
16 =====
17 正在创建 <撒旦-2> ...
18 炸弹以每小时100速度飞到了(81, 27) 的位置 ...
19 在 (81, 27) 的位置重绘炸弹弹体 ...
20 炸弹以每小时100速度飞到了(142, 68) 的位置 ...
21 在 (142, 68) 的位置重绘炸弹弹体 ...
22 炸弹以每小时100速度飞到了(237, 110) 的位置 ...
23 在 (237, 110) 的位置重绘炸弹弹体 ...
24 =====
25 正在复用 <撒旦-2> ...
26 炸弹以每小时100速度飞到了(27, 36) 的位置 ...
27 在 (27, 36) 的位置重绘炸弹弹体 ...
28 炸弹以每小时100速度飞到了(118, 40) 的位置 ...
29 在 (118, 40) 的位置重绘炸弹弹体 ...
30 炸弹以每小时100速度飞到了(120, 43) 的位置 ...
31 在 (120, 43) 的位置重绘炸弹弹体 ...
32 =====
33 正在复用 <撒旦-2> ...
34 炸弹以每小时100速度飞到了(92, 32) 的位置 ...
35 在 (92, 32) 的位置重绘炸弹弹体 ...
36 炸弹以每小时100速度飞到了(113, 48) 的位置 ...
37 在 (113, 48) 的位置重绘炸弹弹体 ...
38 炸弹以每小时100速度飞到了(131, 93) 的位置 ...
39 在 (131, 93) 的位置重绘炸弹弹体 ...
40 =====
```

- 41 正在创建 <撒旦-3> ...
- 42 炸弹以每小时100速度飞到了(47, 26) 的位置 ...



3. 结构图

最后根据上面的代码就可以画出享元模式的 UML 类图了 (学会了享元模式之后, 应该先画 UML 类图, 然后根据类图写代码。)



关于投掷炸弹可能也对应一个类，在上面的测试程序中对应的就是 `main()` 函数，在这个 UML 类图中并没有画出投掷炸弹的这个类。除此之外还有几个知识点需要我们做到心中有数：

1. 享元模式中的享元类可以有子类也可以没有
2. 享元模式中可以增加享元工厂也可以不添加
3. 享元工厂的作用和单例模式类似，但二者的关注点略有不同
 - 单例模式关注的是类的对象有且只有一个
 - 享元工厂关注的是某个实例对象是否可以共享

文章作者: 苏丙楹



文章链接: <https://subingwen.cn/design-patterns/flyweight/>

版权声明: 本博客所有文章除特别声明外，均采用 [CC BY-NC-SA 4.0](#) 许可协议。转载请注明来自 [爱编程的大丙](#)！

设计模式

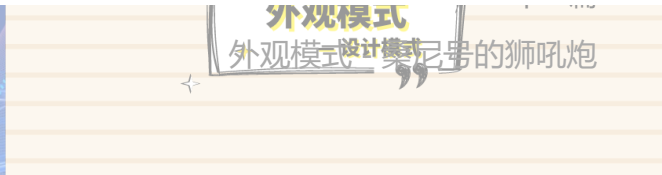


打赏

上一篇

设计模式之

下一篇



👍相关推荐



💬 评论

昵称

邮箱

网址(http://)

来都来了, 说点什么吧...



提交

1 评论



Anonymous

Safari 13.0 macOS 10.12.6

2023-02-28

回复

LaunchBomb (SharedBombBody* body) : m_bomb (body) 中的形参类型应该是 FlyweightBody * 吧, 大佬

Powered By [Valine](#)

v1.5.1

冀 ICP 备 2021000342 号 - 1



冀公网安备 13019902000353 号