12 享元模式

享元（flyweight）模式是一种用于性能优化的模式，“fly”是指苍蝇，意思为蝇量级。

核心是利用共享技术来有效支持大量细粒度的对象。

系统中如果创建了大量比较类似的对象而导致内存占用过高，就非常有用了。尤其是在内存不多的移动端浏览器中更有意义

初始享元模式

假设有个内衣工厂，目前有50款男士与女士内衣，此时需要定制一些塑料模特来穿内衣拍广告，正常情况也许会这样实现：

```
/**
 * 不使用享元模式
 */
class Model {
    constructor(sex,underwear){
        this.sex = sex;
        this.underwear = underwear;
    }

    takePhoto = () => {
        console.log(this.sex + this.underwear);
    }
}

for(let i = 1; i < 50; i++){
    const maleModel = new Model('male','underwear'+i);
    maleModel.takePhoto();
}

for(let i = 1; i < 50; i++){
    const femaleModel = new Model('female','underwear'+i);
    femaleModel.takePhoto();
}
```

假如内衣款式更多，那么程序可能因为对象过多而崩溃。

这个时候我们分析发现其实模特本身有一个就够了，只需要穿不同的衣服拍照而已是特殊的，我们可以这样改进使得最多两个对象就可以实现功能：

```
/**
 * 改进之后
 */
 class Model {
    constructor(sex){
        this.sex = sex;
    }

    takePhoto = () => {
        console.log(this.sex + this.underwear);
    }
}

const maleModel = new Model('male');
const femaleModel = new Model('female');

for(let i = 1; i < 50; i++){
    maleModel.underwear = 'underwear'+i;
    maleModel.takePhoto();
}

for(let i = 1; i < 50; i++){
    femaleModel.underwear = 'underwear'+i;
    femaleModel.takePhoto();
}
```

内部状态与外部状态

上面的例子便是享元模式的雏形，他要求将对象的属性划分为内部状态与外部状态，目标是尽量减少共享对象的数量，具体划分规则参考：

- 内部状态存储与对象内部
- 内部状态可以被一些对象共享
- 内部状态独立于具体业务场景，通常不变
- 外部状态取决于具体业务场景，通常会变，不能共享

这样我们可以尝试剥离内外部状态，外部状态只在必要时传入共享对象来组装为完整对象，大大减少了系统对象的数量，但是组装过程是会增加一些耗时，本质是一种以时间换空间的优化模式

上面性别是内部状态，内衣是外部状态，因为每件衣服都是不同的，不能被共享

享元模式的通用结构

上面的例子还存在一些问题:

- 也许并不是一开始就需要所有的共享对象，我们的例子则是提前初始化完成好的
  - 尝试通过对象工厂来解决
- 给model对象手动设置underwear外部状态可能不是一个好的方式，尤其是复杂系统可能会相当复杂
  - 需要一个管理器来记录对象相关的外部状态，使得外部状态通过某个钩子和共享对象联系起来

文件上传的例子

在我们上传模块的开发中，我们曾经借助了享元模式提高性能

对象爆炸

文件上传功能允许同时选择2000个文件，第一版中我们直接new了2000个upload对象，IE浏览器下直接进入了假死状态。

上传支持好几种模式，包括浏览器插件，Flash，表单上传等。这里我们假设只有插件和Flash上传，原理其实都是一样的，当用户选择了文件之后，插件或Flash都会通知调用Window下的一个全局JS函数startUpload，组合用户选择的文件并合成数组塞进函数参数列表

此外为了简化示例，我们暂且去掉Upload对象其他功能，只保留删除文件功能，看一下第一版核心代码：

```
/**
 * 初版文件上传
 */
class Upload {
    constructor(uploadType, fileName, fileSize){
        this.uploadType = uploadType;
        this.fileName = fileName;
        this.fileSize = fileSize;
        this.dom = null;
    }

    init = (id) => {
        let that = this;
        this.id = id;
        this.dom = document.createElement('div');
        this.dom.innerHTML = `<span>文件名称${this.fileName}, 文件大小${this.fileS
        this.dom.querySelector('delFile').onclikc = () => {
            this.delFile();
        }
        document.body.appendChild(this.dom);
    }
```

```
22
23      delFile = () => {
24          if(window.confirm('确定要删除该文件吗? ')){
25              return this.dom.parentNode.removeChild(this.dom)
26          }
27      }
28  }
29
30  let id = 0;
31  window.startUpload = (uploadType, files){
32      for(let i = 0,file;file = files[i++];){
33          const uploadObj = new Upload(uploadType, file.fileName, file.fileSize);
34          uploadObj.init(id++);
35      }
36  }
```

总结

享元模式（Flyweight Pattern）是一种结构型设计模式，旨在通过共享对象来最大限度地减少内存使用和提高性能。该模式适用于存在大量相似对象的情况，通过共享相同的状态，可以有效地减少内存消耗。

在享元模式中，对象分为两种类型：内部状态（Intrinsic State）和外部状态（Extrinsic State）。内部状态是可以被多个对象共享的状态，它们独立于具体的场景，因此可以在不同的对象之间共享。外部状态是对象的一些特定属性，它们会随着场景的改变而改变，因此不能被共享。

享元模式的核心思想是将对象的创建和管理分离，通过工厂类来管理共享对象的创建和获取。当需要使用对象时，先从工厂类获取对象，如果对象已存在，则直接返回共享的对象；如果对象不存在，则创建一个新的对象并添加到共享池中，以便下次复用。

使用享元模式的好处包括：

1. 减少内存消耗：通过共享相同的对象实例，可以大大减少系统的内存消耗。
2. 提高性能：共享对象的复用可以减少对象的创建和销毁的开销，从而提高系统的性能。
3. 简化系统：享元模式可以将对象的状态分为内部状态和外部状态，从而简化对象的设计和管理。

然而，享元模式也存在一些注意事项：

1. 对象的内部状态和外部状态需要区分清楚，否则可能会导致混乱。
2. 共享对象需要是可共享且线程安全的，确保在多线程环境下使用时不会出现问题。
3. 对象的共享可能会导致对象的修改影响其他对象，需要谨慎处理对象状态的改变。

总而言之，享元模式通过共享对象的方式来减少内存消耗和提高性能，特别适用于存在大量相似对象的场景，可以提高系统的效率和简化对象的管理。

在Web开发中，享元模式可以应用于以下场景：

1. 字符串缓存：在Web应用中，经常会使用一些固定的字符串，比如错误提示信息、日志信息等。这些字符串可以被看作是享元对象的内部状态，可以被多个对象共享，通过使用享元模式，可以将这些字符串缓存起来，减少内存消耗。
2. 数据库连接池：Web应用通常需要与数据库进行交互，每个数据库连接都是一种昂贵的资源。通过使用享元模式，可以将数据库连接作为共享的对象实例，避免频繁地创建和销毁连接，提高系统的性能和资源利用率。
3. 缓存管理：在Web开发中，常常需要对一些计算结果或数据进行缓存，以提高系统的响应速度。通过使用享元模式，可以将缓存对象作为享元对象，共享缓存数据，避免重复计算或访问数据库，提高系统的性能。
4. UI组件复用：在前端开发中，UI组件的复用是一个常见的需求。通过使用享元模式，可以将通用的UI组件作为享元对象，共享组件的状态和行为，减少重复的代码编写和组件的创建开销。

总的来说，享元模式在Web开发中的应用可以帮助减少内存消耗、提高系统性能、简化代码编写和资源管理，特别适用于需要大量创建和管理相似对象的场景。