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  }
```

享元模式重构文件上传

上一节的代码最大的问题就是Upload对象过多，这里我们用享元模式进行重构。

首先我们需要区分哪些是内部状态，哪些是外部状态，这里根据之前的原则：

- 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

我们认为uploadType是内部状态，因为upload对象必须依赖此属性工作，因为不同Type的上传原理可能不一样，接口也不一样，需要走各自的流程与逻辑

而fileName和fileSize是外部状态，根据场景而变化，每一个上传对象也都不一样

剥离外部状态

这里我们需要重构Upload构造函数：

```
1  /**
2   * 享元模式重构
3   */
4   class Upload {
5      constructor(uploadType){
6          // 只保留内部状态
7          this.uploadType = uploadType;
8      }
9
```

```
    //原先init函数不再需要，初始化转移到uploadManager的add函数中

    delFile = (id) => {
        // 把id对应对象的外部状态组装到共享对象中，因为下面需要获取fileSize了
        uploadManager.setExternalState(id, this);

        if(this.fileSize < 3000){
            return this.dom.parentNode.removeChild(this.dom)
        }

        if(window.confirm('确定要删除该文件吗? ')){
            return this.dom.parentNode.removeChild(this.dom)
        }
    }
}
```

工厂对象进行实例化

```
// 工厂负责创建Upload对象，判断是否需要返回共享对象
const UploadFactory = {
    createdFlyWeightObjs: {},
    create: function(uploadType){
        if(this.createdFlyWeightObjs[uploadType]){
            return this.createdFlyWeightObjs[uploadType]
        }
        return this.createdFlyWeightObjs[uploadType] = new Upload(uploadType)
    }
}
```

管理器封装外部状态

现在我们来完善前面提到的uploadManager对象，它负责向工厂提交创建对象的请求，并通过uploadDatabase对象保存所有upload对象的外部状态，并在程序运行过程中给upload实例进行外部状态组装和设置：

```
const uploadManager = {
    uplodaDatabase: {},
    add: function(id, uploadType, fileName, fileSize){
        const flyweightObj = UploadFactory.create(uploadType);
        const dom = document.createElement('div');
        dom.innerHTML = `<span>文件名称${this.fileName}, 文件大小${this.fileSize}<
        dom.querySelector('.delFile').onclikc = () => {
```

```
            flyweightObj.delFile(id);
        }
        document.body.appendChild(this.dom);
        this.uplodaDatabase[id] = {
            fileName,
            fileSize,
            dom
        };
        return flyweightObj;
    },
    setExternalState: function(id, flyweightObj){
        const uploadData = this.uplodaDatabase[id];
        for(let i in uploadData){
            flyweightObj[i] = uploadData[i];
        }
    }
}
```

最后我们启动：

```
let id2 = 0;
window.startUpload = function(uploadType, files){
    for(let i=0,file;file = files[i++];){
        uploadManager.add(++id2, uploadType, file.fileName, file.fileSize);
    }
}
```

这样即便同时上传2000个文件，需要创建的upload对象数量依然为2，只有Type挂钩

享元模式的适用性

虽然他是一种很好的优化方案，但也会有一些问题，我们需要多维护factory和manager对象，所以我们需要根据具体情况判断需不需要使用该模式：

- 一个程序中有大量相似对象
- 由于使用大量对象造成很大的内存开销
- 对象大多数状态都可以变为外部状态
- 剥离之后，可以使用较少的共享对象取代大量对象

再谈内部状态与外部状态

现在我们考虑如下的两种极端情况

没有内部状态的享元

还是文件上传的例子，如果浏览器只支持插件上传，还不支持或者开发者不想要支持Flash上传，那其实作为内部状态的uploadType是可以删除的。

这样Upload构造函数就是无参数的形式。这意味着也只需要唯一的一个共享对象，这个时候其实共享对象工厂变成了一个单例工厂，虽然没有内部状态，但还是可以解决外部状态的问题

没有外部状态的享元

网上许多资料，经常把Java和C#的字符串看成享元，这种说法是否正确呢？我们看看下面这段Java代码，来分析一下：

```
1  public class StringInternExample {
2      public static void main(String[] args) {
3          String str1 = "Hello";  // 字符串常量
4          String str2 = new String("Hello");  // 新建字符串对象
5          String str3 = str2.intern();  // 调用 intern 方法
6
7          System.out.println(str1 == str2);  // false，比较引用地址
8          System.out.println(str1 == str3);  // true，比较引用地址
9      }
10 }
```

当使用字符串常量时，Java 中的字符串常量池（String Pool）会自动进行字符串的 intern 操作。在字符串常量池中，相同内容的字符串只会保存一份，以节省内存空间，并提高字符串比较的效率

但是这里其实并不是使用了享元模式的结果，他的关键在于区别内外状态。这里并没有剥离外部状态的过程，str1和str3完全指向的就是同一个对象，这里其实使用了共享的技术，并并不是一个纯粹的享元

对象池

前面我们提到了Java中String的对象池，这里我们学习一下这种共享技术。对象池的原理也很好理解，例如我们小组学习《JS权威指南这本书》，从节约角度看，其实没有必要没人买一本，只需要买一本或者建立一个内部小型图书馆，然后大家需要时候借阅即可。如果同一时间要看的人多，我们可以再补充

对象池技术应用非常广泛，例如HTTP连接池和数据库连接池都是代表。Web前端最大应用场景大概是跟DOM相关的操作，以避免频繁创建和删除DOM

对象池的实现

假设我们在开发地图应用，地图上经常会出现一些标志地名的小气泡，我们叫它toolTip。

搜索我家附近时候，有2个气泡。接着搜索附近兰州拉面时，有6个气泡，按照对象池思想，其实第二次我们应该复用两个并新建四个。

```
/**
 * 地图气泡对象池
 */
const toolTipFactory = {
    toolTipPool: [],
    create: function(){
        if(this.toolTipPool.length === 0){
            const div = document.createElement('div');
            document.body.appendChild(div);
            return div;
        } else {
            // 对象池不为空则取出一个dom
            return this.toolTipPool.shift();
        }
    },
    recover: function(toolTipDom){
        // 回收
        return this.toolTipPool.push(toolTipDom);
    }
}

const ary = [];
for(let i = 0, str;str=['A','B'][i++];){
    const toolTip = toolTipFactory.create();
    toolTip.innerHTML = str;
    ary.push(toolTip);
}
// 此时页面会出现两个A和B的div节点
// 现在地图需要重绘制，我们先回收，然后再利用
for(let i = 0,toolTip;toolTip=ary[i++];){
    toolTipFactory.recover(toolTip);
}
for(let i = 0, str;str=['A','B','C','D','E','F'][i++];){
    const toolTip = toolTipFactory.create();
    toolTip.innerHTML = str;
}

```

通用对象池实现

这里我么可以把创建对象的具体过程抽象出来，实现一个更通用的对象池：

```
/**
 * 通用对象池
 */
const objectPoolFactory = function (createObjFn) {
    const objectPool = [];

    return {
        create: function () {
            const obj = objectPool.length === 0 ? createObjFn.apply(this, argume
            return obj;
        },
        recover: function (obj) {
            return objectPool.push(obj);
        }
    }
}

// 利用上面对象池创建一个iframe对象池
const iframeFactory = objectPoolFactory(function () {
    const iframe = document.createElement('iframe');
    document.body.appendChild(iframe);

    iframe.onload = function () {
        iframe.onload = null;
        // iframe加载完成之后回收节点
        iframeFactory.recover(iframe);
    }
    return iframe;
})

const iframe1 = iframeFactory.create();
iframe1.src = 'http://www.baidu.com';
const ifrmae2 = iframeFactory.create();
ifrmae2.src = 'http://www.QQ.com';

setTimeout(() => {
    // 复用节点
    const ifrmae3 = iframeFactory.create();
    ifrmae2.src = 'http://www.123.com';
}, 3000)
```

对象池是另一种性能优化方案，虽然和享元模式有几分相似，但是没有分离内外部状态这个过程。之前的文件上传也可以通过对象池+事件委托实现

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。