# 私塾在线《研磨设计模式》
## ——跟着CC学设计系列精品教程

# 本节课程概览

- 学习享元模式

一：初识享元模式

包括：定义、结构、参考实现

二：体会享元模式

包括：场景问题、不用模式的解决方案、使用模式的解决方案

三：理解享元模式

包括：认识享元模式、不需要共享的享元实现 、对享元对象的管理 、享元模式的优缺点

四：思考享元模式

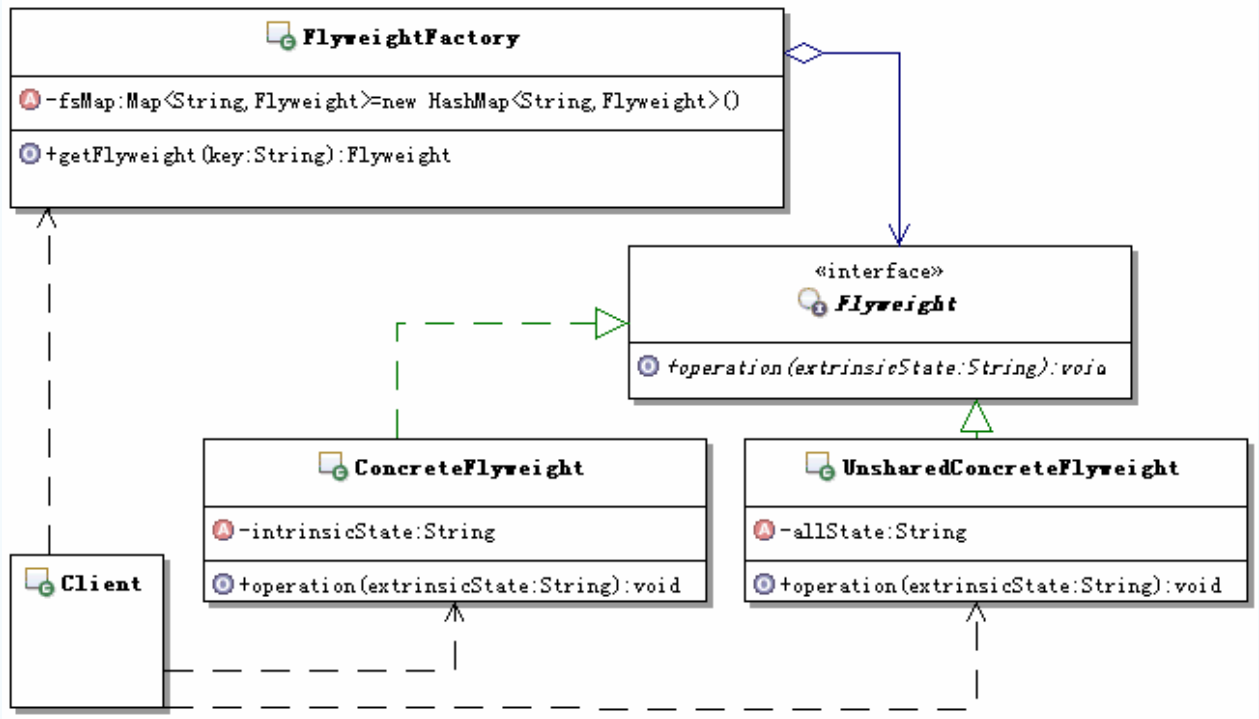
包括：享元模式的本质、何时选用

# 初识享元模式

- 定义

  运用共享技术有效地支持大量细粒度的对象 。

- 结构和说明



# 体会享元模式

Flyweight：

享元接口，通过这个接口flyweight可以接受并作用于外部状态。通过这个接口传入外部的状态，在享元对象的方法处理中可能会使用这些外部的数据。

ConcreteFlyweight：

具体的享元实现对象，必须是可共享的，需要封装flyweight的内部状态。

UnsharedConcreteFlyweight：

非共享的享元实现对象，并不是所有的Flyweight实现对象都需要共享。非共享的享元实现对象通常是对共享享元对象的组合对象。

FlyweightFactory：

享元工厂，主要用来创建并管理共享的享元对象，并对外提供访问共享享元的接口。

Client：

享元客户端，主要的工作是维持一个对flyweight的引用，计算或存储享元对象的外部状态，当然这里可以访问共享和不共享的flyweight对象。

# 体会享元模式

- 加入权限控制

考虑这样一个问题，给系统加入权限控制，这基本上是所有的应用系统都有的功能了。

对于应用系统而言，一般先要登录系统，才可以使用系统的功能，登录过后，用户的每次操作都需要经过权限系统的控制，确保该用户有操作该功能的权限，同时还要控制该用户对数据的访问权限、修改权限等等。总之一句话，一个安全的系统，需要对用户的每一次操作都要做权限检测，包括功能和数据，以确保只有获得相应授权的人，才能执行相应的功能，操作相应的数据。

举个例子来说吧：普通人员都有能查看到本部门人员列表的权限，但是在人员列表中每个人员的薪资数据，普通人员是不可以看到的；而部门经理在查看本部门人员列表的时候，就可以看到每个人员相应的薪资数据。

现在就要来实现为系统加入权限控制的功能，该怎么实现呢？

# 体会享元模式

- 权限管理的基础知识

为了让大家更好的理解后面讲述的知识，先介绍一点权限系统的基础知识。几乎所有的权限系统都分成两个部分，一个是授权部分，一个是验证部分，为了理解它们，首先解释两个基本的名词：安全实体和权限。

1：**安全实体**：就是被权限系统检测的对象，比如工资数据。

2：**权限**：就是需要被校验的权限对象，比如查看、修改等。

安全实体和权限通常要一起描述才有意义，比如有这么个描述：“现在要检测登录人员对工资数据是否有查看的权限”，“工资数据”这个安全实体和“查看”这个权限一定要一起描述。如果只出现安全实体描述，那就变成这样：“现在要检测登录人员对工资数据”，对工资数据干什么呀，没有后半截，一看就知道不完整；当然只有权限描述也不行，那就变成：“现在要检测登录人员是否有查看的权限”，对谁的查看权限啊，也不完整。所以安全实体和权限通常要一起描述。

# 体会享元模式

- 授权和验证

**所谓授权是指：**把对某些安全实体的某些权限分配给某些人员的过程。

**所谓验证是指：**判断某个人员对某个安全实体是否拥有某个或某些权限的过程。

也就是说，**授权过程即是权限的分配过程，而验证过程则是权限的匹配过程**。在目前应用系统的开发中，多数是利用数据库来存放授权过程产生的数据，也就是说：授权是向数据库里面添加数据、或是维护数据的过程，而匹配过程就变成了从数据库中获取相应数据进行匹配的过程了。

为了让问题相对简化一点，就不去考虑权限的另外两个特征，一个是继承性，一个是最近匹配原则，都什么意思呢，还是解释一下：

# 体会享元模式

- 权限的继承性

  指的是：如果多个安全实体存在包含关系，而某个安全实体没有相应的权限限制，那么它会继承包含它的安全实体的相应权限。

- 权限的最近匹配原则

  指的是：如果多个安全实体存在包含关系，而某个安全实体没有相应的权限限制，那么它会向上寻找并匹配相应权限限制，直到找到一个离这个安全实体最近的拥有相应权限限制的安全实体为止。如果把整个层次结构都寻找完了都没有匹配到相应权限限制的话，那就说明所有人对这个安全实体都拥有这个相应的权限限制。

# 体会享元模式

- 不用模式的解决方案

**1：看看现在都已经有什么了**

系统的授权工作已经完成，授权数据记录在数据库里面，具体的数据结构就不去展开了，反正里面记录了人员对安全实体所拥有的权限。假如现在系统中已有如下的授权数据：

| | | | | |
|---|---|---|---|---|
| 张三 | 对 | 人员列表 | 拥有 | 查看的权限 |
| 李四 | 对 | 人员列表 | 拥有 | 查看的权限 |
| 李四 | 对 | 薪资数据 | 拥有 | 查看的权限 |
| 李四 | 对 | 薪资数据 | 拥有 | 修改的权限 |

# 体会享元模式

**2：思路选择**

由于操作人员进行授权操作过后，各人员被授予的权限是记录在数据库中的，刚开始有开发人员提出，每次用户操作系统的时候，都直接到数据库里面去动态查询，以判断该人员是否拥有相应的权限，但很快就被否决掉了，试想一下，用户操作那么频繁，每次都到数据库里面动态查询，这会严重加剧数据库服务器的负担，使系统变慢。

为了加快系统运行的速度，开发小组决定采用一定的缓存，当每个人员登录的时候，就把该人员能操作的权限获取到，存储在内存中，这样每次操作的时候，就直接在内存里面进行权限的校验，速度会大大加快，这是**典型的以空间换时间**的做法。

# 体会享元模式

- 有何问题

看了上面的实现，很简单，而且还考虑了性能的问题，在内存中缓存了每个人相应的权限数据，使得每次判断权限的时候，速度大大加快，实现得挺不错，难道有什么问题吗？

仔细想想，问题就来了，既有缓存这种方式固有的问题，也有我们实现上的问题。先说说缓存固有的问题吧，这个不在本次讨论之列，大家了解一下。

1：**缓存时间长度的问题**
2：**缓存数据和真实数据的同步问题**
3：**缓存的多线程并发控制**

仔细看看上面输出结果，输出的值是不同的，表明这些对象实例肯定不是同一个对象实例，而是多个对象实例。这就引出一个问题了，就是对象实例数目太多，为什么这么说呢？看看就描述这么几条数据，数数看有多少个对象实例呢？

# 体会享元模式

另外，这些对象的粒度都很小，都是简单的描述某一个方面的对象，而且很多数据是重复的，在这些大量重复的数据上耗费掉了很多的内存。比如在前面示例的数据中就会发现有重复的部分，见下面框住的部分：

| | | | | |
|---|---|---|---|---|
| 张三 | 对 | 人员列表 | 拥有 | 查看的权限 |
| 李四 | 对 | 人员列表 | 拥有 | 查看的权限 |
| 李四 | 对 | 薪资数据 | 拥有 | 查看的权限 |
| 李四 | 对 | 薪资数据 | 拥有 | 修改的权限 |

前面讲过，对于安全实体和权限一般要联合描述，因此对于“人员列表 这个安全实体 的 查看权限 限制”，就算是授权给不同的人员，这个描述是一样的。假设在某极端情况下，要把“人员列表 这个安全实体 的 查看权限 限制”授权给一万个人，那么数据库里面会有一万条记录，按照前面的实现方式，会有一万个对象实例，而这些实例里面，有大部分的数据是重复的，而且会重复一万次，你觉得这是不是个很大的问题呢？

把上面的问题描述出来就是：在系统当中，存在大量的细粒度对象，而且存在大量的重复数据，严重耗费内存，如何解决？

# 体会享元模式

- 使用模式来解决的思路

现在造成内存浪费的主要原因：就是细粒度对象太多，而且有大量重复的数据。如果能够有效的减少对象的数量，减少重复的数据，那么就能够节省不少内存。一个基本的思路就是缓存这些包含着重复数据的对象，让这些对象只出现一次，也就只耗费一份内存了。

但是请注意，并不是所有的对象都适合缓存，因为缓存的是对象的实例，实例里面存放的主要是对象属性的值。因此，如果被缓存的对象的属性值经常变动，那就不适合缓存了，因为真实对象的属性值变化了，那么缓存里面的对象也必须要跟着变化，否则缓存中的数据就跟真实对象的数据不同步，可以说是错误的数据了。

因此，需要分离出被缓存对象实例中，哪些数据是不变且重复出现的，哪些数据是经常变化的，真正应该被缓存的数据是那些不变且重复出现的数据，把它们称为对象的内部状态，而那些变化的数据就不缓存了，把它们称为对象的外部状态。

# 体会享元模式

这样在实现的时候，把内部状态分离出来共享，称之为享元，通过共享享元对象来减少对内存的占用。把外部状态分离出来，放到外部，让应用在使用的时候进行维护，并在需要的时候传递给享元对象使用。为了控制对内部状态的共享，并且让外部能简单的使用共享数据，提供一个工厂来管理享元，把它称为享元工厂。

比如：


很明显，可以把安全实体和权限的描述定义成为享元，而和它们结合的人员数据，就可以做为享元的外部数据。

# 理解享元模式

- 认识享元模式

1：变与不变

享元模式设计的重点就在于分离变与不变，把一个对象的状态分成内部状态和外部状态，内部状态是不变的，外部状态是可变的。然后通过共享不变的部分，达到减少对象数量、并节约内存的目的。在享元对象需要的时候，可以从外部传入外部状态给共享的对象，共享对象会在功能处理的时候，使用自己内部的状态和这些外部的状态。

2：共享与不共享

在享元模式中，享元对象又有共享与不共享之分，这种情况通常出现在跟组合模式合用的情况，通常共享的是叶子对象，一般不共享的部分是由共享部分组合而成的，由于所有细粒度的叶子对象都已经缓存了，那么缓存组合对象就没有什么意义了。这个在后面给大家一个示例。

# 理解享元模式

3：内部状态和外部状态

享元模式的内部状态，通常指的是包含在享元对象内部的、对象本身的状态，通常是独立于使用享元的场景的信息，一般创建过后就不再变化的状态，因此可以共享。

外部状态指的是享元对象之外的状态，取决于使用享元的场景，会根据使用场景而变化，因此不可共享。如果享元对象需要这些外部状态的话，可以从外部传递到享元对象里面，比如通过方法的参数来传递。

也就是说享元模式真正缓存和共享的数据是享元的内部状态，而外部状态是不应该被缓存共享的。

另外一点，内部状态和外部状态是独立的，外部状态的变化不应该影响到内部状态。

# 理解享元模式

4：实例池

在享元模式中，为了创建和管理共享的享元部分，引入了享元工厂，享元工厂中一般都包含有享元对象的实例池，享元对象就是缓存在这个实例池中的。

简单介绍一点实例池的知识，**所谓实例池，指的是缓存和管理对象实例的程序，通常实例池会提供对象实例的运行环境，并控制对象实例的生命周期。**

**工业级的实例池实现上有两个最基本的难点，一个就是动态控制实例数量，一个就是动态分配实例来提供给外部使用**。这些都是需要算法来做保证的。

回到享元模式中来，享元工厂中的实例池可没有这么复杂，因为共享的享元对象基本上都是一个实例，一般不会出现同一个享元对象有多个实例的情况，这样就不用去考虑动态创建和销毁享元对象实例的功能；另外只有一个实例，也就不存在动态调度的麻烦，反正就是它了。

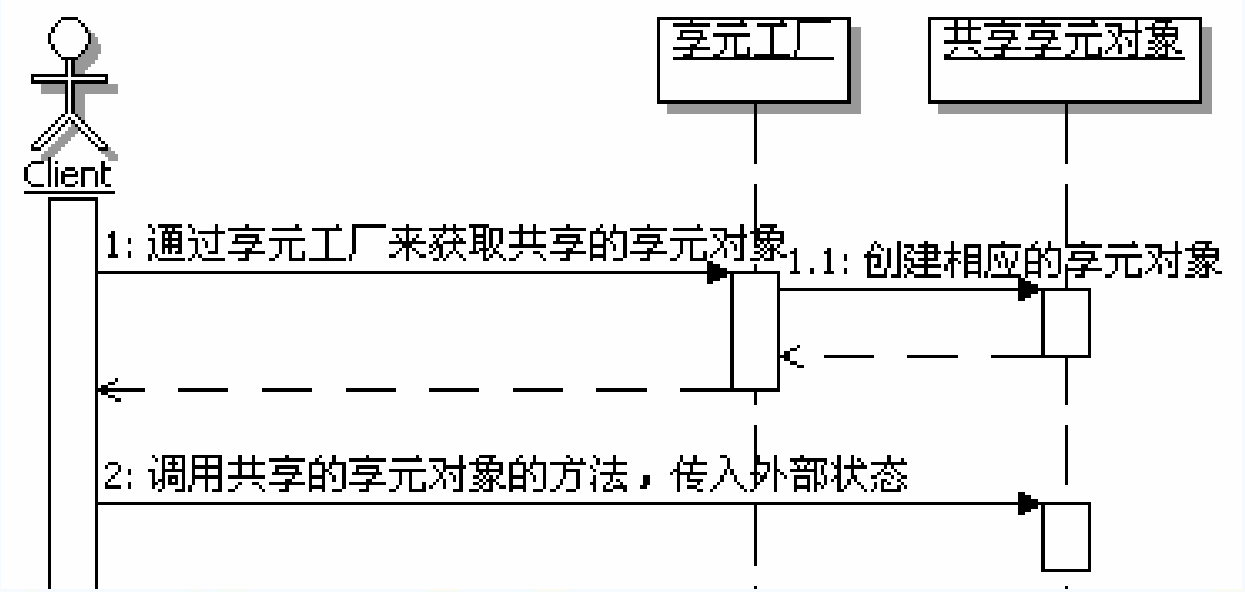
这也主要是因为享元对象封装的多半是对象的内部状态，这些状态通常是不变的，有一个实例就够了，不需要动态控制生命周期，也不需要动态调度，它只需要做一个缓存而已，没有上升到真正的实例池那么个高度。

# 理解享元模式

5：享元模式的调用顺序示意图

享元模式的使用上，有两种情况，一种是没有“不需要共享”的享元对象，就如同前面的示例那样，只有共享享元对象的情况；还有一种是既有共享享元对象，又有不需要共享的享元对象的情况，这种情况后面再示例。

这里看看只有共享享元对象的情况下，享元模式的调用顺序



# 理解享元模式

6：谁来初始化共享对象

在享元模式中，通常是在第一次向享元工厂请求获取共享对象的时候，进行共享对象的初始化，而且多半都是在享元工厂内部实现，不会从外部传入共享对象。当然可以从外部传入一些创建共享对象需要的值，享元工厂可以按照这些值去初始化需要共享的对象，然后就把创建好的共享对象的实例放入享元工厂内部的缓存中，以后再请求这个共享对象的时候就不用再创建了。

# 理解享元模式

- 不需要共享的享元实现

在实际开发中，存在不需要共享的享元实现，这种情况多出现在组合结构中，对于使用已经缓存的享元组合出来的对象，就没有必要再缓存了，也就是把已经缓存的享元当做叶子结点，组合出来的组合对象就不需要再被缓存了。也把这种享元称为复合享元。

比如要给某人分配“薪资数据”这个安全实体的“修改”权限，那么一定会把“薪资数据”的“查看权限”也分配给这个人，如果按照前面的做法，这就需要分配两个对象，为了方便，干脆把这两个描述组合起来，打包成一个对象，命名成为“操作薪资数据”，那么分配权限的时候，可以这么描述：

把 “操作薪资数据” 分配给 张三

这句话的意思就相当于

把 “薪资数据” 的 “查看”权限 分配给 张三

把 “薪资数据” 的 “修改”权限 分配给 张三

这样一来，“操作薪资数据”就相当于是一个不需要共享的享元，它实际由享元“薪资数据 的 查看 权限”，和享元“薪资数据 的 修改 权限”这两个享元组合而成，因此“操作薪资数据”本身也就不需要再共享了。

# 理解享元模式

- 对享元对象的管理

虽然享元模式对于共享的享元对象实例的管理要求，没有实例池对实例管理的要求那么高，但是也还是有很多自身的特点功能，比如：引用计数、垃圾清除等。

所谓引用计数，就是享元工厂能够记录每个享元被使用的次数；所谓垃圾，就是在缓存中存在，但是不再需要被使用的缓存中的对象。而垃圾清除，就是在不需要这些数据的时候，应该把这些数据从缓存中清除，释放相应的内存空间，以节省资源。

在前面的示例中，共享的享元对象是很多人共享的，基本上可以一直存在于系统中，不用清除。但是垃圾清除是享元对象管理的一个很常见功能，还是通过示例给大家讲一下，看看如何实现这些常见的功能。

**1：实现引用计数的基本思路**

要实现引用计数，就在享元工厂里面定义一个Map，它的key值跟缓存享元对象的key是一样的，而value就是被引用的次数，这样当外部每次获取该享元的时候，就把对应的引用计数取出来加上1，然后再记录回去。

# 理解享元模式

## 2：实现垃圾回收的基本思路

要实现垃圾回收就比较麻烦点，首先要能确定哪些是垃圾？其次是何时回收？还有由谁来回收？如何回收？解决了这些问题，也就能实现垃圾回收了。

确定哪些是垃圾的一个简单方案是这样的，定义一个缓存对象的配置对象，在这个对象中描述了缓存的开始时间和最长不被使用的时间，这个时候判断是垃圾的计算公式如下：当前的时间 - 缓存的开始时间 >= 最长不被使用的时间。当然，每次这个对象被使用的时候，就把那个缓存开始的时间更新为使用时的当前时间，也就是说如果一直有人用的话，这个对象是不会被判断为垃圾的。

何时回收的问题，当然是判断出来是垃圾了就可以回收了。

关键是谁来判断垃圾，还有谁来回收垃圾的问题。一个简单的方案是定义一个内部的线程，这个线程在享元工厂被创建的时候就启动运行。由这个线程每隔一定的时间来循环缓存中所有对象的缓存配置，看看是否是垃圾，如果是垃圾，那就可以启动回收了。

怎么回收呢？这个简单，就是直接从缓存的map对象中删除掉相应的对象，让这些对象没有引用的地方，那么这些对象就可以等着被虚拟机的垃圾回收来回收掉了。

# 理解享元模式

- 享元模式的优缺点
  1：减少对象数量，节省内存空间
  2：维护共享对象，需要额外开销

# 思考享元模式

- 享元模式的本质

  享元模式的本质是：**分离与共享**

- 何时选用享元模式

1：如果一个应用程序使用了大量的细粒度对象，可以使用享元模式来减少对象数量

2：如果由于使用大量的对象，造成很大的存储开销，可以使用享元模式来减少对象数量，并节约内存

3：如果对象的大多数状态都可以转变为外部状态，比如通过计算得到，或是从外部传入等，可以使用享元模式来实现内部状态和外部状态的分离

4：如果不考虑对象的外部状态，可以用相对较少的共享对象取代很多组合对象，可以使用享元模式来共享对象，然后组合对象来使用这些共享对象