(19)国家知识产权局

(12)发明专利申请

(10)申请公布号 CN 115048092 A
(43)申请公布日 2022.09.13

(21)申请号 202210674015.0
(22)申请日 2022.06.14
(71)申请人 北京知道创宇信息技术股份有限公司
地址 100000 北京市朝阳区阜通东大街1号院5号楼1单元311501室
(72)发明人 张苗
(74)专利代理机构 北京超凡宏宇专利代理事务所(特殊普通合伙) 11463
专利代理师 杜杨
(51)Int.Cl.
G06F 8/34(2018.01)
G06F 8/36(2018.01)
G06F 8/38(2018.01)
G06F 8/10(2018.01)
G06F 8/30(2018.01)

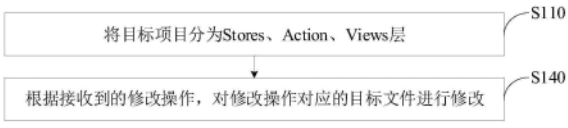
权利要求书2页 说明书10页 附图7页

(54)发明名称

状态管理模型优化方法、装置、电子设备及可读存储介质

(57)摘要

本申请提供了一种状态管理模型优化方法、装置、电子设备及可读存储介质，涉及计算机技术领域。该方法包括：将目标项目分为包括Store类的Stores层、包括用于指示交互逻辑的Action类的Action层、包括组件的Views层，不同层对应的存储范围不同，Store类中包括属性文件，Action类中包括至少一个Action函数；根据接收到的修改操作，对修改操作对应的目标文件进行修改，其中，目标文件为修改操作指定的属性文件或Action函数。如此，通过将项目分层，不同层对应不同的存储范围，便于管理，并且在需要进行修改时，可缩小查找范围，从相应的存储范围查找出要修改的文件。



CN 115048092 A

1.一种状态管理模型优化方法,其特征在于,所述方法包括:

将目标项目分为Stores、Action、Views层,其中,不同层对应的存储范围不同,Stores层中包括Store类,所述Store类中包括属性文件,所述Action层包括用于指示交互逻辑的Action类,所述Action类中包括至少一个Action函数,所述Views层中包括组件;

根据接收到的修改操作,对所述修改操作对应的目标文件进行修改,其中,所述目标文件为所述修改操作指定的属性文件或Action函数。

2.根据权利要求1所述的方法,其特征在于,在所述根据接收到的修改操作,对所述修改操作对应的目标文件进行修改之前,所述方法还包括:

获得各Store类的位置以及各Action类的位置;

根据各Store类的位置以及各Action类的位置,生成单一数据源。

3.根据权利要求2所述的方法,其特征在于,所述获得各Store类的位置以及各Action类的位置,包括:

在执行所述Store类及Action类之前,执行目标装饰器;

利用所述目标装饰器获得各Store类的位置以及各Action类的位置。

4.根据权利要求2所述的方法,其特征在于,所述根据各Store类的位置以及各Action类的位置,生成单一数据源,包括:

针对各Store类,根据该Store类的位置生成该各Store类的属性标识;

针对各Action类,根据该Action类的位置生成该各Action类的属性标识;

将各Store类的属性标识及各Action类的属性标识存入数据源中,以得到所述单一数据源。

5.根据权利要求2所述的方法,其特征在于,所述方法还包括:

在所述目标项目初始时,对所述单一数据源中的至少一部分类进行实例化。

6.根据权利要求5所述的方法,其特征在于,所述对所述单一数据源中的至少一部分类进行实例化,包括:

获得所述单一数据源中各类的使用频率;

根据各类的使用频率,从所述单一数据源中选出至少一个第一目标类,其中,所述第一目标类的使用频率不小于未被选择的类的使用频率;

对所述第一目标类进行实例化。

7.根据权利要求5所述的方法,其特征在于,所述方法还包括:

在需要使用未实例化的第二目标类时,对所述第二目标类进行实例化。

8.一种状态管理模型优化装置,其特征在于,所述装置包括:

分层模块,用于将目标项目分为Stores、Action、Views层,其中,不同层对应的存储范围不同,Stores层中包括Store类,所述Store类中包括属性文件,所述Action层包括用于指示交互逻辑的Action类,所述Action类中包括至少一个Action函数,所述Views层中包括组件;

修改模块,用于根据接收到的修改操作,对所述修改操作对应的目标文件进行修改,其中,所述目标文件为所述修改操作指定的属性文件或Action函数。

9.一种电子设备,其特征在于,包括处理器和存储器,所述存储器存储有能够被所述处理器执行的机器可执行指令,所述处理器可执行所述机器可执行指令以实现权利要求1-7

中任意一项所述的状态管理模型优化方法。

10.一种可读存储介质，其上存储有计算机程序，其特征在于，所述计算机程序被处理器执行时实现如权利要求1-7中任意一项所述的状态管理模型优化方法。

状态管理模型优化方法、装置、电子设备及可读存储介质

技术领域

[0001] 本申请涉及计算机技术领域，具体而言，涉及一种状态管理模型优化方法、装置、电子设备及可读存储介质。

背景技术

[0002] React是由Facebook研发的用于构建用户界面的JavaScript库，可以构建管理自身状态的封装组件，然后对组件进行组合以构成复杂的UI(User Interface，用户界面)。组件中的状态是指能驱动应用的数据，组件除了可以维护组件内部的状态数据(通过this.state访问)以外，还可以使用组件外部的数据(通过this.props访问)。当组件的状态数据改变的时候，React能够高效更新并渲染合适的组件，从而达到页面快速响应的效果。

[0003] 但是单靠React自身的能力是无法满足现在复杂应用的需求的。虽然React能够管理一部分状态数据，但是随着应用开发日趋复杂，需要管理的状态数据也越来越多。这些状态数据可能包括服务器响应结果、缓存数据、UI状态、页面标记等。而且同一个状态数据，可能在多个组件中使用，这又涉及父子组件，兄弟组件，以及不相邻组件间的数据传递，对于以树状结构构成的React应用来说，兄弟组件，以及不相邻组件间的数据传递是困难且十分影响性能的。

[0004] 由上可知，React需要引入专门的状态管理工具。该工具需要能够管理全局使用的状态，使数据的传递变得更加便利；同时也能够像React组件自身状态那样，当某个全局状态改变，对应使用该状态的组件也会触发更新。MobX就是现存的状态管理工具之一。

[0005] MobX的一大特点是非常的灵活，状态数据及交互逻辑可随处定义。但是，灵活也意味着随意，MobX的各种代码散落在项目各处，无法实现统一管理，不便于对状态数据文件或者交互逻辑进行修改。

发明内容

[0006] 本申请实施例提供了一种状态管理模型优化方法、装置、电子设备及可读存储介质，其通过将项目分为Stores、Action、Views层，不同层对应不同的存储范围，便于管理，并且在需要进行修改时，可从相应的存储范围查找出要修改的文件，该方式可缩小查找范围。

[0007] 本申请的实施例可以这样实现：

[0008] 第一方面，本申请实施例提供一种状态管理模型优化方法，所述方法包括：

[0009] 将目标项目分为Stores、Action、Views层，其中，不同层对应的存储范围不同，Stores层中包括Store类，所述Store类中包括属性文件，所述Action层包括用于指示交互逻辑的Action类，所述Action类中包括至少一个Action函数，所述Views层中包括组件；

[0010] 根据接收到的修改操作，对所述修改操作对应的目标文件进行修改，其中，所述目标文件为所述修改操作指定的属性文件或Action函数。

[0011] 第二方面，本申请实施例提供一种状态管理模型优化装置，所述装置包括：

[0012] 分层模块，用于将目标项目分为Stores、Action、Views层，其中，不同层对应的存

储范围不同，Stores层中包括Store类，所述Store类中包括属性文件，所述Action层包括用于指示交互逻辑的Action类，所述Action类中包括至少一个Action函数，所述Views层中包括组件；

[0013] 修改模块，用于根据接收到的修改操作，对所述修改操作对应的目标文件进行修改，其中，所述目标文件为所述修改操作指定的属性文件或Action函数。

[0014] 第三方面，本申请实施例提供一种电子设备，包括处理器和存储器，所述存储器存储有能够被所述处理器执行的机器可执行指令，所述处理器可执行所述机器可执行指令以实现前述实施方式所述的状态管理模型优化方法。

[0015] 第四方面，本申请实施例提供一种可读存储介质，其上存储有计算机程序，所述计算机程序被处理器执行时实现如前述实施方式所述的状态管理模型优化方法。

[0016] 本申请实施例提供的状态管理模型优化方法、装置、电子设备及可读存储介质，将目标项目分为Stores、Action、Views层，其中，不同层对应的存储范围不同，Stores层中包括Store类，所述Store类中包括属性文件，所述Action层包括用于指示交互逻辑的Action类，所述Action类中包括至少一个Action函数，所述Views层中包括组件；根据接收到的修改操作，对所述修改操作对应的目标文件进行修改，其中，所述目标文件为所述修改操作指定的属性文件或Action函数。如此，通过分层，便于对目标项目进行管理；当需要修改文件时，可从文件所属层对应的存储范围中查找相应的文件，进而进行修改，该方式可缩小查询范围，便于快速完成对状态数据文件或者交互逻辑的修改。

附图说明

[0017] 为了更清楚地说明本申请实施例的技术方案，下面将对实施例中所需要使用的附图作简单地介绍，应当理解，以下附图仅示出了本申请的某些实施例，因此不应被看作是对范围的限定，对于本领域普通技术人员来讲，在不付出创造性劳动的前提下，还可以根据这些附图获得其他相关的附图。

[0018] 图1为Mobx的工作流程图；

[0019] 图2为用户交互触发Action事件的流程图；

[0020] 图3为常见的MobX结构示意图；

[0021] 图4为本申请实施例提供的电子设备的方框示意图；

[0022] 图5为本申请实施例提供的状态管理模型优化方法的流示意图之一；

[0023] 图6为本申请实施例提供的状态管理模型优化方法的流示意图之二；

[0024] 图7为本申请实施例提供内的单一数据源结构示意图；

[0025] 图8为本申请实施例提供的状态管理模型优化方法的流示意图之三；

[0026] 图9为图8中步骤S150包括的子步骤的流程示意图；

[0027] 图10为本申请实施例提供的状态管理模型优化方法的流示意图之四；

[0028] 图11为本申请实施例提供的状态管理模型优化装置的方框示意图之一；

[0029] 图12为本申请实施例提供的状态管理模型优化装置的方框示意图之二；

[0030] 图13为本申请实施例提供的状态管理模型优化装置的方框示意图之三。

[0031] 图标：100-电子设备；110-存储器；120-处理器；130-通信单元；200-状态管理模型优化装置；210-分层模块；220-数据源生成模块；230-修改模块；240-实例化模块。

具体实施方式

[0032] 为使本申请实施例的目的、技术方案和优点更加清楚，下面将结合本申请实施例中的附图，对本申请实施例中的技术方案进行清楚、完整地描述，显然，所描述的实施例是本申请一部分实施例，而不是全部的实施例。通常在此处附图中描述和示出的本申请实施例的组件可以以各种不同的配置来布置和设计。

[0033] 因此，以下对在附图中提供的本申请的实施例的详细描述并非旨在限制要求保护的本申请的范围，而是仅仅表示本申请的选定实施例。基于本申请的实施例，本领域技术人员在没有做出创造性劳动的前提下所获得的所有其他实施例，都属于本申请保护的范围。

[0034] 需要说明的是，术语“第一”和“第二”等之类的关系术语仅仅用来将一个实体或者操作与另一个实体或操作区分开来，而不一定要求或者暗示这些实体或操作之间存在任何这种实际的关系或者顺序。而且，术语“包括”、“包含”或者其任何其他变体意在涵盖非排他性的包含，从而使得包括一系列要素的过程、方法、物品或者设备不仅包括那些要素，而且还包括没有明确列出的其他要素，或者是还包括为这种过程、方法、物品或者设备所固有的要素。在没有更多限制的情况下，由语句“包括一个……”限定的要素，并不排除在包括所述要素的过程、方法、物品或者设备中还存在另外的相同要素。

[0035] MobX是简单、可扩展的JavaScript状态容器，提供全局的、可预测化的状态管理。MobX提供了优化应用状态与React组件同步的机制，这种机制就是使用响应式虚拟依赖状态图表，它只有在真正需要的时候才更新并且永远保持是最新的状态。MobX的工作流程如图1所示。

[0036] Action表示任一一段可以改变状态的代码，比如用户事件、后端数据推送、预定事件等等。MobX支持单向数据流，通过动作(Action)改变状态(state)，当状态改变时，所有衍生都会进行原子级的自动更新，而状态的改变会更新所有受影响的视图。因为进行的变更是原子级的，所以MobX更加的灵活。

[0037] 在现有的React项目中，MobX可以通过npm、yarn等的包管理工具下载，在文件中引入该包即可使用。项目通常会根据系统模块或页面将MobX管理的状态、相应逻辑函数及衍生分割成一系列相互协作的类，这种类被称为Store。它们随着系统模块或页面分布在整个项目中。在Store中有任意数目与之相依赖的观察者，通过给Store属性添加@observable装饰器，可以使该属性成为一个可观察的状态。Store中也有多个用于修改被观察状态的Action函数，Action是唯一可以修改状态的代码，通常会在Action中向服务器发送数据的请求。使用@observer可以将组件变成可观察组件，使用@inject将被观察的状态及Action注入组件之中，这样就将组件与Store关联了起来。

[0038] 如图2所示，当用户操作页面进行交互，触发了changeKey1事件。由于该组件是可观察组件，且changeKey1是Action事件，所以会通知Store进行相关逻辑处理，然后更新对应状态，进而引发页面的重新渲染。其中，图2中的Component表示组件。

[0039] MobX的一大特点是非常的灵活，常见的MobX结构如图3所示，状态数据可随处定义，比如可以直接定义在组件内来替代state的作用，也可以定义在单独的Store文件内；交互逻辑也是可以选择写在组件内或者是写在Store文件中。但是，灵活也意味着随意，MobX的各种代码散落在项目各处，无法实现统一管理。而且用户交互往往涉及多个Store数据的处理，Store间可能形成交叉引用的网状结构，这在页面数量庞大、功能复杂的大型项目中，

是一个致命的弱点。因为这种网状结构会导致各种Store互相依赖，耦合度变高。当要重构某个功能模块时，可能会涉及该功能模块对应的Store内状态和方法的修改，此时就需要考虑其他功能模块是否使用了这个Store的状态或方法，就需要一个个的去排查。这就造成系统功能重构困难，组件复用性变差，模块间的组合也会花费更多的精力了。

[0040] 针对以上情况，本申请实施例提供了一种状态管理模型优化方法、装置、电子设备及可读存储介质，通过将项目分为Stores、Action、Views层，不同层对应不同的存储范围，便于修改等管理，可保证大型项目的可维护性，避免项目因为状态管理而导致的结构的凌乱。

[0041] 下面结合附图，对本申请的一些实施方式作详细说明。在不冲突的情况下，下述的实施例及实施例中的特征可以相互组合。

[0042] 请参照图4，图4为本申请实施例提供的电子设备100的方框示意图。所述电子设备100可以是，但不限于，电脑、服务器等。所述电子设备100可以包括存储器110、处理器120及通信单元130。所述存储器110、处理器120以及通信单元130各元件相互之间直接或间接地电性连接，以实现数据的传输或交互。例如，这些元件相互之间可通过一条或多条通讯总线或信号线实现电性连接。

[0043] 其中，存储器110用于存储程序或者数据。所述存储器110可以是，但不限于，随机存取存储器(Random Access Memory，RAM)，只读存储器(Read Only Memory，ROM)，可编程只读存储器(Programmable Read-Only Memory，PROM)，可擦除只读存储器(Erasable Programmable Read-Only Memory，EPROM)，电可擦除只读存储器(Electric Erasable Programmable Read-Only Memory，EEPROM)等。

[0044] 处理器120用于读/写存储器110中存储的数据或程序，并执行相应地功能。比如，存储器110中存储有状态管理模型优化装置200，所述状态管理模型优化装置200包括至少一个可以软件或固件(firmware)的形式存储于所述存储器110中的软件功能模块。所述处理器120通过运行存储在存储器110内的软件程序以及模块，如本申请实施例中的状态管理模型优化装置200，从而执行各种功能应用以及数据处理，即实现本申请实施例中的状态管理模型优化方法。

[0045] 通信单元130用于通过网络建立所述电子设备100与其它通信终端之间的通信连接，并用于通过所述网络收发数据。

[0046] 应当理解的是，图4所示的结构仅为电子设备100的结构示意图，所述电子设备100还可包括比图4中所示更多或者更少的组件，或者具有与图4所示不同的配置。图4中所示的各组件可以采用硬件、软件或其组合实现。

[0047] 请参照图5，图5为本申请实施例提供的状态管理模型优化方法的流示意图之一。所述方法可以应用于上述电子设备100。下面对状态管理模型优化方法的具体流程进行详细阐述。在本实施例中，所述方法可以包括步骤S110及步骤S140。

[0048] 步骤S110，将目标项目分为Stores、Action、Views层。

[0049] 步骤S140，根据接收到的修改操作，对所述修改操作对应的目标文件进行修改。

[0050] 在本实施例中，所述目标项目可以为用户指定的React项目。开发人员在编写该目标项目时，可以将相应的文件放到文件所属层对应的存储范围中，如此，可以将目标项目分为多层。不同层对应的存储范围不同。当然可以理解的是，上述仅为举例说明，也可以通过

其他方式实现分层。所述多层包括Stores层、Action层、Views层。

[0051] 其中，Stores层中包括Store类，所述Store类中包括属性文件；所述Action层包括用于指示交互逻辑的Action类，所述Action类中包括至少一个Action函数；所述Views层中包括组件。

[0052] 比如，当开发人员在编写一个属性文件时，由于该属性文件属于Stores层，因此可以将该属性文件放在Stores层对应的存储范围1中。后续若需要查找到该属性文件，则可以从存储范围1中进行查找，以获得该属性文件。

[0053] 当开发人员根据需求确定要修改属性文件或Action函数时，则可以从文件所属的层对应的范围中进行查找，并将查找到的属性文件或Action函数作为目标文件，然后针对该目标文件输入修改操作。在接收到该修改操作的情况下，则可以基于修改操作对所述目标文件进行修改。如此，无需工作人员进行大范围查找，即可确定要修改的文件，并且便于管理。

[0054] 请参照图6，图6为本申请实施例提供的状态管理模型优化方法的流示意图之二。为进一步便于管理，在本实施例中，在步骤S140之前，所述方法还可以包括步骤S120及步骤S130。

[0055] 步骤S120，获得各Store类的位置以及各Action类的位置。

[0056] 步骤S130，根据各Store类的位置以及各Action类的位置，生成单一数据源。

[0057] 可选地，如图7所示，可以通过任意方式获得各Store类的位置以及各Action类的位置，以生成单一数据源。其中，图7中的Store1表示一个Store类，Store2表示另一个Store类；action1表示一个action类，action2表示另一个action类。作为一种可能的实现方式，因为MobX的@observable、@inject等很多功能都是使用装饰器的模式，所以在本方式中也可以同样使用装饰器模式来进行位置的查找，这样可以使MobX相关功能风格保持一致。可以预先设一目标装饰器，装饰器其实本身是一个函数，它在被装饰的类初始化之前先被执行，能获取到被装饰类的本身。这样就能在Store类和Action类实例化前，获取被装饰的Store和Action类的位置，进而通过统一处理形成单一数据源。该单一数据源可被组件访问。

[0058] 也即，可预先设置一目标装饰器，在执行各Store类及各Action类之前，执行目标装饰器；利用所述目标装饰器获得各Store类的位置以及各Action类的位置。

[0059] 该过程的伪代码示例如下：

```
// store 装饰器的实现，用于记录 Store 的位置，便于查找
let storeDecorator = (config) => target => {
   //rootStore 为单一数据源状态管理类
   rootStore['store'] = rootStore['store'] || []
   // 存入数据源
   rootStore['store'].push({ target, path: config.path})
   return target;
}

//装饰器
@storeDecorator({
      path:'pageA.storeA', //在全局 store 中的访问路径
})
class StoreA{

}

//在视图组件中访问单一数据源 storeA 中内容
This.props.rootStore.storeA
```

Action 类和其装饰器实现思路与 Store 类是相同的。

[0060] 可选地，在生成单一数据源时，可以针对各Store类，根据该Store类的位置生成该各Store类的属性标识；针对各Action类，根据该Action类的位置生成该各Action类的属性标识。其中，所述属性标识可以是属性名。根据地址确定属性标识的具体方式可以结合实际

需求设置。为便于后续可以基于属性标识确定出所对应的位置，同时减小对存储空间的占用，可以预先设置好属性标注与位置的映射关系，基于一条映射关系，既可基于地址确定出属性标识，也可以基于属性标识确定出地址。在获得各类的属性标识之后，将该属性标识存入一个数据源中，以作为所述单一数据源。

[0061] 在生成了单一数据源的情况下，当开发人员需要修改文件时，开发人员还可以基于该单一数据源中找到目标文件的具体位置，进而针对该目标文件输入修改操作。如此，可以进一步加快查找到目标文件的速度。

[0062] 目前，Store实例化的时机和方式不可控。因为程序会在第一次使用到某个Store中的方法或状态时，才会进行对它进行实例化。若是需要根据其中某个状态来动态展示页面时，实例化还没完成，就会造成页面更新不及时；同时，也会由于实例化未完成，导致在出错时得到的报错信息中不会包括较多可用于排除问题的信息，从而开发者造成调试的难度，无法精确快速的定位问题所在的位置。

[0063] 大型项目很可能会涉及多个Store交互，为了避免Store实例化的时机和方式不可控，可找到所有Store，并形成全局唯一数据源，然后基于该数据源对Store实例化的时机和方式进行控制。请参照图8，图8为本申请实施例提供的状态管理模型优化方法的流示意图之三。在本实施例中，在步骤S130之后，所述方法还可以包括步骤S150。

[0064] 步骤S150，在所述目标项目初始时，对所述单一数据源中的至少一部分类进行实例化。

[0065] 可选地，在所述目标项目初始化时，可以基于该单一数据源，一次性的寻找到所有的Store类和Action类进行实例化以达到控制实例化时机的效果。也可以在所述目标项目初始化时，仅对一部类进行实例化，对于另外一些类则按需加载。

[0066] 作为一种可能的实现方式，为了追求性能，可以将使用频率高的类在项目初始时进行实例化，将使用频率低的类按需实例化，这样既可以降低开发过程中状态更新滞后，又有一定的性能提升。请参照图9，图9为图8中步骤S150包括的子步骤的流程示意图。在本实施例中，步骤S150可以包括子步骤S151～子步骤S153。

[0067] 子步骤S151，获得所述单一数据源中各类的使用频率。

[0068] 各类的使用频率，可以是工作人员手动设置的，也可以是通过其他方式获得的。

[0069] 子步骤S153，根据各类的使用频率，从所述单一数据源中选出至少一个第一目标类。

[0070] 可选地，可以将使用频率大于预设频率的类均作为第一目标类；也可以按照使用频率大小，对所述单一数据源中的类(包括Store类及Action类)进行排序，然后按照频率大小从大到小的顺序选取一定数量的类作为第一目标类。所述第一目标类的使用频率不小于未被选择的类(即未被选作第一目标类的类)的使用频率。

[0071] 子步骤S153，对所述第一目标类进行实例化。

[0072] 请参照图10，图10为本申请实施例提供的状态管理模型优化方法的流示意图之四。在本实施例中，在步骤S150之后，所述方法还可以包括步骤S160。

[0073] 步骤S160，在需要使用未实例化的第二目标类时，对所述第二目标类进行实例化。

[0074] 在为了提升性能采用按需实例化时，可以在项目开始时，将所有Store类和Action类的位置查找到，并将不需要按需加载的Store类和Action类实例化；对于需要按需加载的

类，在使用到的时候再实例化。比如，在访问到rootStore的storeA属性时，如果已经存在storeA属性，则直接返回结果。如果发现不存在storeA属性，再对rootStore进行动态的实例化。可以使用JavaScript的对象访问器属性来实现这个功能。伪代码示例如下：

```
Object.defineProperty(rootStore, 'storeA', {
  configurable: true,
  enumerable: true,
  get() {
    // 检查是否已经实例化， __instance 是按需实例化的标志
    StoreA['__instance'] = StoreA['__instance'] || new StoreA();
    return StoreA['__instance'];
  },
  set() {
    throw Error('can not set store again');
  },
});
```

[0075]

[0076] 通过本申请实施例可以实现按需加载的单一数据源的状态管理，这种技术能够对上文提到的MobX在大型项目中的几个缺陷起到非常明显的改善效果，可以保证大型项目的可维护性，避免项目因为状态管理而导致的结构的凌乱；可统一管理MobX状态文件，通过使用自动查找机制，可减轻开发人员负担；通过实现Store的按需实例化，可提升性能。

[0077] 为了执行上述实施例及各个可能的方式中的相应步骤，下面给出一种状态管理模型优化装置200的实现方式，可选地，该状态管理模型优化装置200可以采用上述图4所示的电子设备100的器件结构。进一步地，请参照图11，图11为本申请实施例提供的状态管理模型优化装置200的方框示意图之一。需要说明的是，本实施例所提供的状态管理模型优化装置200，其基本原理及产生的技术效果和上述实施例相同，为简要描述，本实施例部分未提及之处，可参考上述的实施例中相应内容。所述状态管理模型优化装置200可以包括：分层模块210及修改模块230。

[0078] 所述分层模块210，用于将目标项目分为Stores、Action、Views层。其中，不同层对应的存储范围不同，Stores层中包括Store类，所述Store类中包括属性文件，所述Action层包括用于指示交互逻辑的Action类，所述Action类中包括至少一个Action函数，所述Views层中包括组件；

[0079] 所述修改模块230，用于根据接收到的修改操作，对所述修改操作对应的目标文件

进行修改。其中,所述目标文件为所述修改操作指定的属性文件或Action函数。

[0080] 请参照图12,图12为本申请实施例提供的状态管理模型优化装置200的方框示意图之二。在本实施例中,所述状态管理模型优化装置200可以包括数据源生成模块220。

[0081] 所述数据源生成模块220,用于:获得各Store类的位置以及各Action类的位置;根据各Store类的位置以及各Action类的位置,生成单一数据源。

[0082] 请参照图13,图13为本申请实施例提供的状态管理模型优化装置200的方框示意图之三。在本实施例中,所述状态管理模型优化装置200还可以包括实例化模块240。

[0083] 所述实例化模块240,用于在所述目标项目初始时,对所述单一数据源中的至少一部分类进行实例化。

[0084] 所述实例化模块240,还用于在需要使用未实例化的第二目标类时,对所述第二目标类进行实例化。

[0085] 可选地,上述模块可以软件或固件(Firmware)的形式存储于图4所示的存储器110中或固化于电子设备100的操作系统(Operating System,OS)中,并可由图4中的处理器120执行。同时,执行上述模块所需的数据、程序的代码等可以存储在存储器110中。

[0086] 本申请实施例还提供一种可读存储介质,其上存储有计算机程序,所述计算机程序被处理器执行时实现所述的状态管理模型优化方法。

[0087] 综上所述,本申请实施例提供一种状态管理模型优化方法、装置、电子设备及可读存储介质,将目标项目分为Stores、Action、Views层,其中,不同层对应的存储范围不同,Stores层中包括Store类,所述Store类中包括属性文件,所述Action层包括用于指示交互逻辑的Action类,所述Action类中包括至少一个Action函数,所述Views层中包括组件;根据接收到的修改操作,对所述修改操作对应的目标文件进行修改,其中,所述目标文件为所述修改操作指定的属性文件或Action函数。如此,通过分层,便于对目标项目进行管理;当需要修改文件时,可从文件所属层对应的存储范围中查找相应的文件,进而进行修改,该方式可缩小查询范围,便于快速完成对状态数据文件或者交互逻辑的修改。。

[0088] 在本申请所提供的几个实施例中,应该理解到,所揭露的装置和方法,也可以通过其它的方式实现。以上所描述的装置实施例仅仅是示意性的,例如,附图中的流程图和框图显示了根据本申请的多个实施例的装置、方法和计算机程序产品的可能实现的体系架构、功能和操作。在这点上,流程图或框图中的每个方框可以代表一个模块、程序段或代码的一部分,所述模块、程序段或代码的一部分包含一个或多个用于实现规定的逻辑功能的可执行指令。也应当注意,在有些作为替换的实现方式中,方框中所标注的功能也可以以不同于附图中所标注的顺序发生。例如,两个连续的方框实际上可以基本并行地执行,它们有时也可以按相反的顺序执行,这依所涉及的功能而定。也要注意的是,框图和/或流程图中的每个方框、以及框图和/或流程图中的方框的组合,可以用执行规定的功能或动作的专用的基于硬件的系统来实现,或者可以用专用硬件与计算机指令的组合来实现。

[0089] 另外,在本申请各个实施例中的各功能模块可以集成在一起形成一个独立的部分,也可以是各个模块单独存在,也可以两个或两个以上模块集成形成一个独立的部分。

[0090] 所述功能如果以软件功能模块的形式实现并作为独立的产品销售或使用时,可以存储在一个计算机可读取存储介质中。基于这样的理解,本申请的技术方案本质上或者说对现有技术做出贡献的部分或者该技术方案的部分可以以软件产品的形式体现出来,该计

算机软件产品存储在一个存储介质中,包括若干指令用以使得一台计算机设备(可以是个人计算机,服务器,或者网络设备等)执行本申请各个实施例所述方法的全部或部分步骤。而前述的存储介质包括:U盘、移动硬盘、只读存储器(ROM,Read-Only Memory)、随机存取存储器(RAM,Random Access Memory)、磁碟或者光盘等各种可以存储程序代码的介质。

[0091] 以上所述仅为本申请的可选实施例而已,并不用于限制本申请,对于本领域的技术人员来说,本申请可以有各种更改和变化。凡在本申请的精神和原则之内,所作的任何修改、等同替换、改进等,均应包含在本申请的保护范围之内。

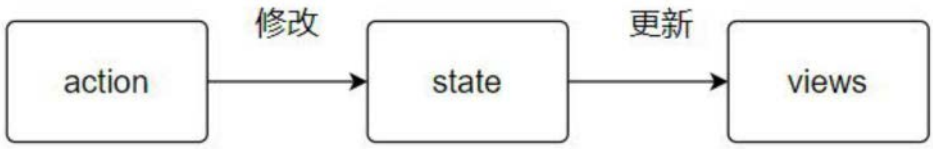


图1

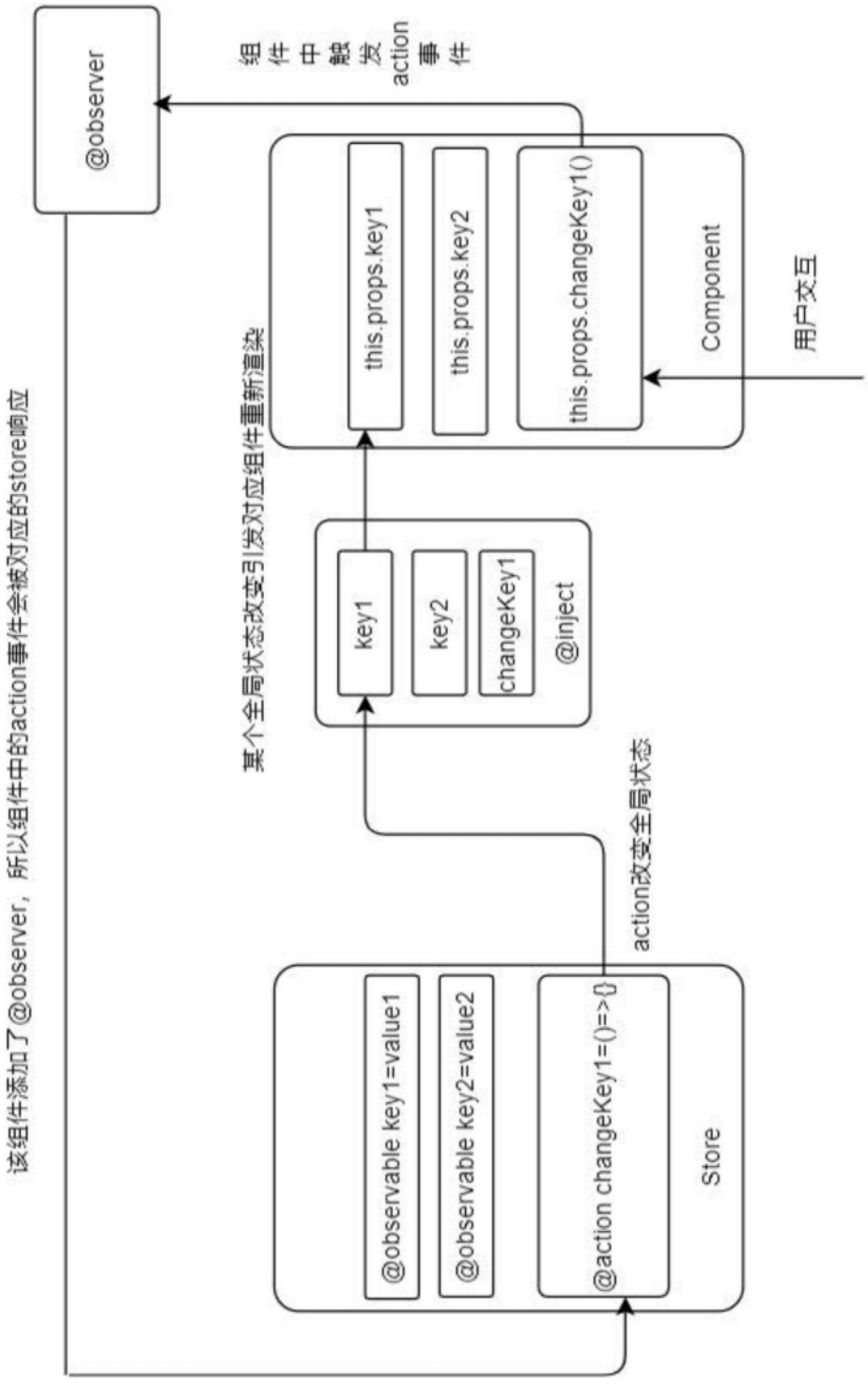

@observer
组件中触发action事件
this.props.key1
this.props.key2
this.props.changeKey1()
Component
用户交互
某个全局状态改变引发对应组件重新渲染
key1
key2
changeKey1
@inject
action改变全局状态
@observable key1=value1
@observable key2=value2
@action changeKey1=()=>{}
Store
该组件添加了@observer，所以组件中的action事件会被对应的store响应

图2

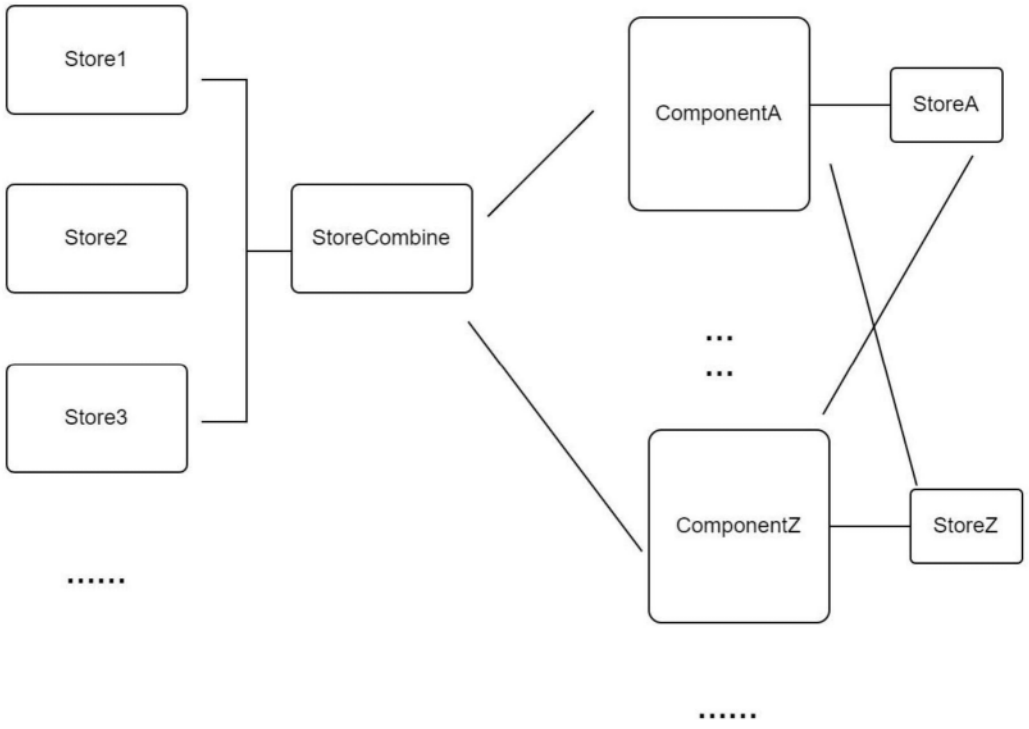

Store1
Store2
Store3
.....
StoreCombine
ComponentA
StoreA
...
...
ComponentZ
StoreZ
.....

图3

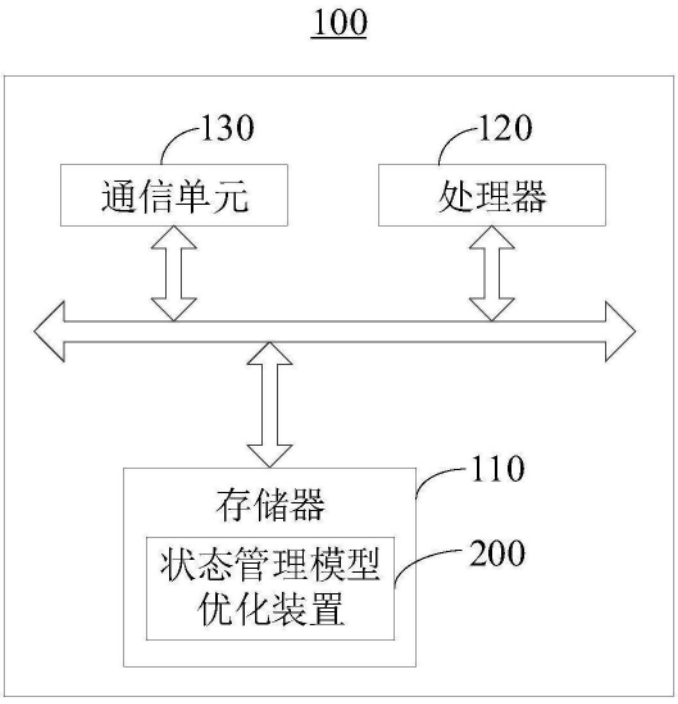

100
130
通信单元
120
处理器
存储器
110
状态管理模型
优化装置
200

图4


S110
将目标项目分为Stores、Action、Views层
S140
根据接收到的修改操作，对修改操作对应的目标文件进行修改

图5

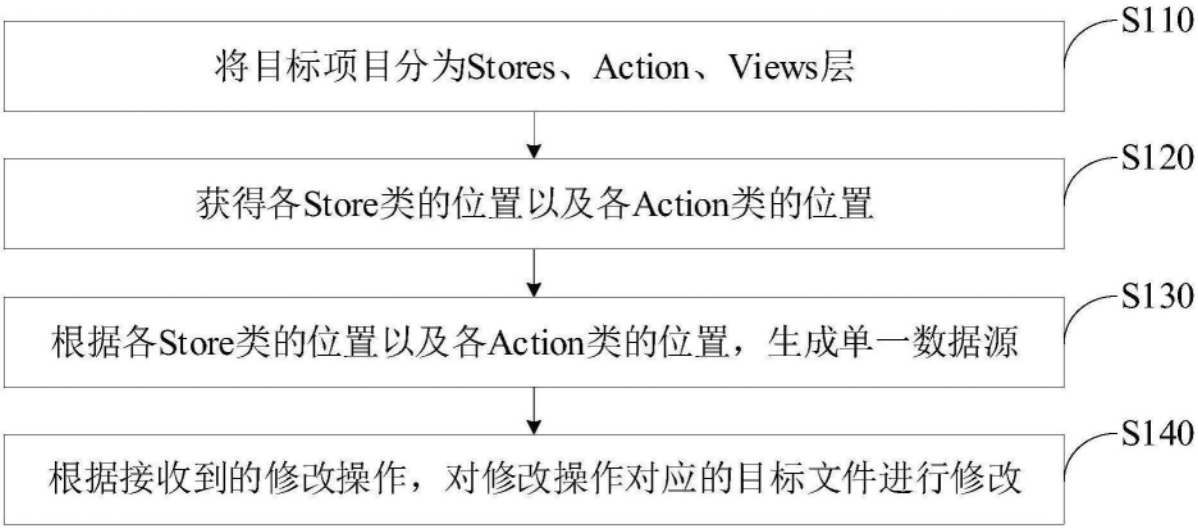

S110
将目标项目分为Stores、Action、Views层
S120
获得各Store类的位置以及各Action类的位置
S130
根据各Store类的位置以及各Action类的位置，生成单一数据源
S140
根据接收到的修改操作，对修改操作对应的目标文件进行修改

图6

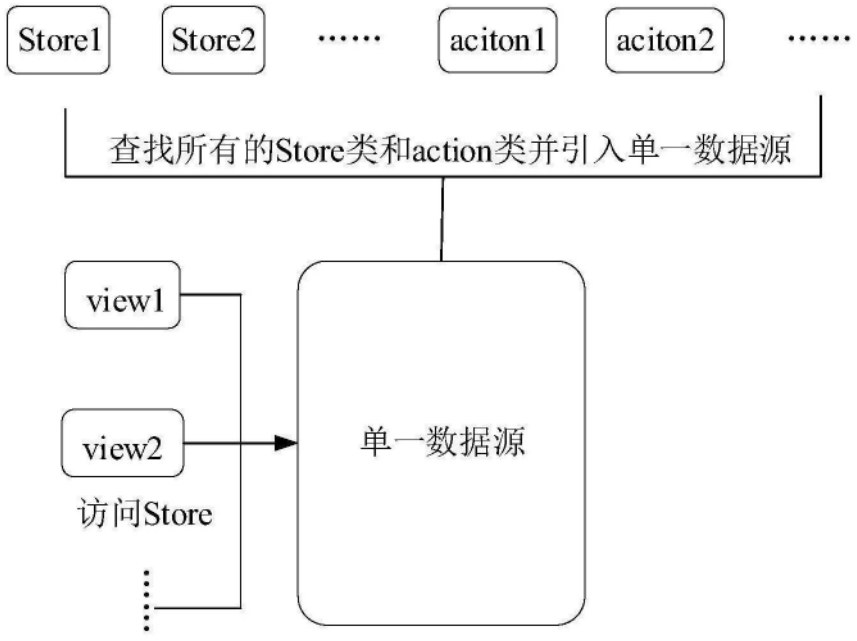

Store1
Store2
……
aciton1
aciton2
……
查找所有的Store类和action类并引入单一数据源
view1
view2
访问Store
单一数据源

图7

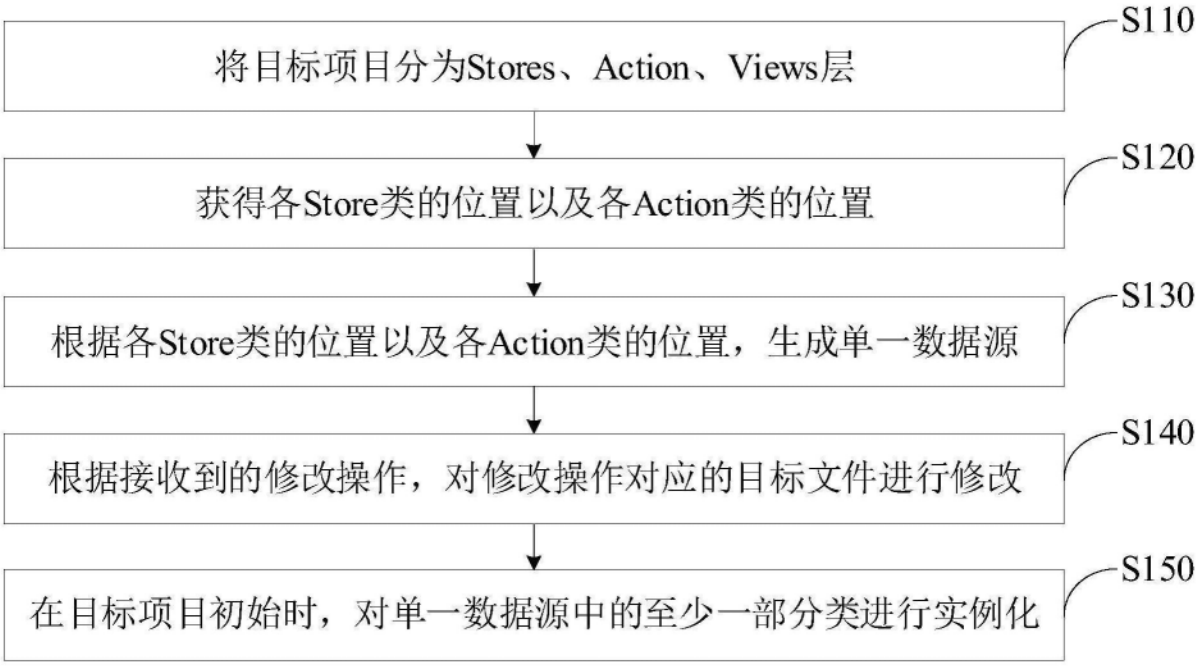

S110
将目标项目分为Stores、Action、Views层
S120
获得各Store类的位置以及各Action类的位置
S130
根据各Store类的位置以及各Action类的位置，生成单一数据源
S140
根据接收到的修改操作，对修改操作对应的目标文件进行修改
S150
在目标项目初始时，对单一数据源中的至少一部分类进行实例化

图8

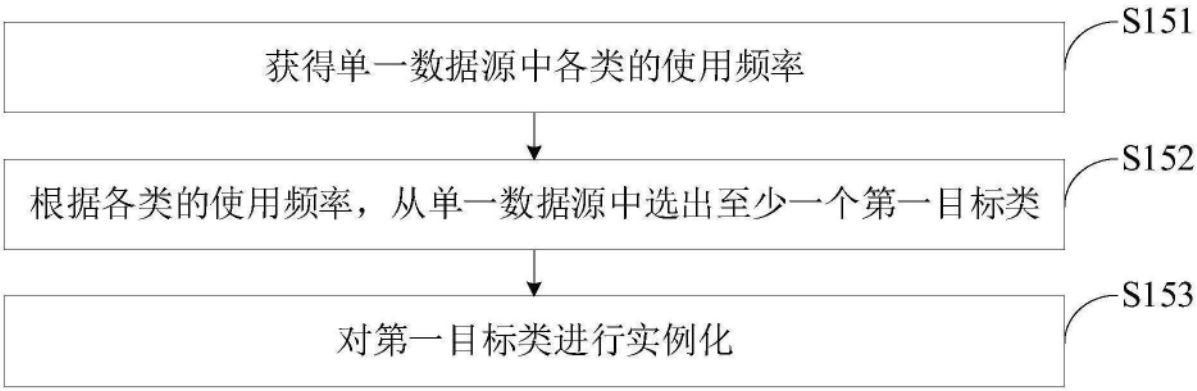

S151
获得单一数据源中各类的使用频率
S152
根据各类的使用频率，从单一数据源中选出至少一个第一目标类
S153
对第一目标类进行实例化

图9

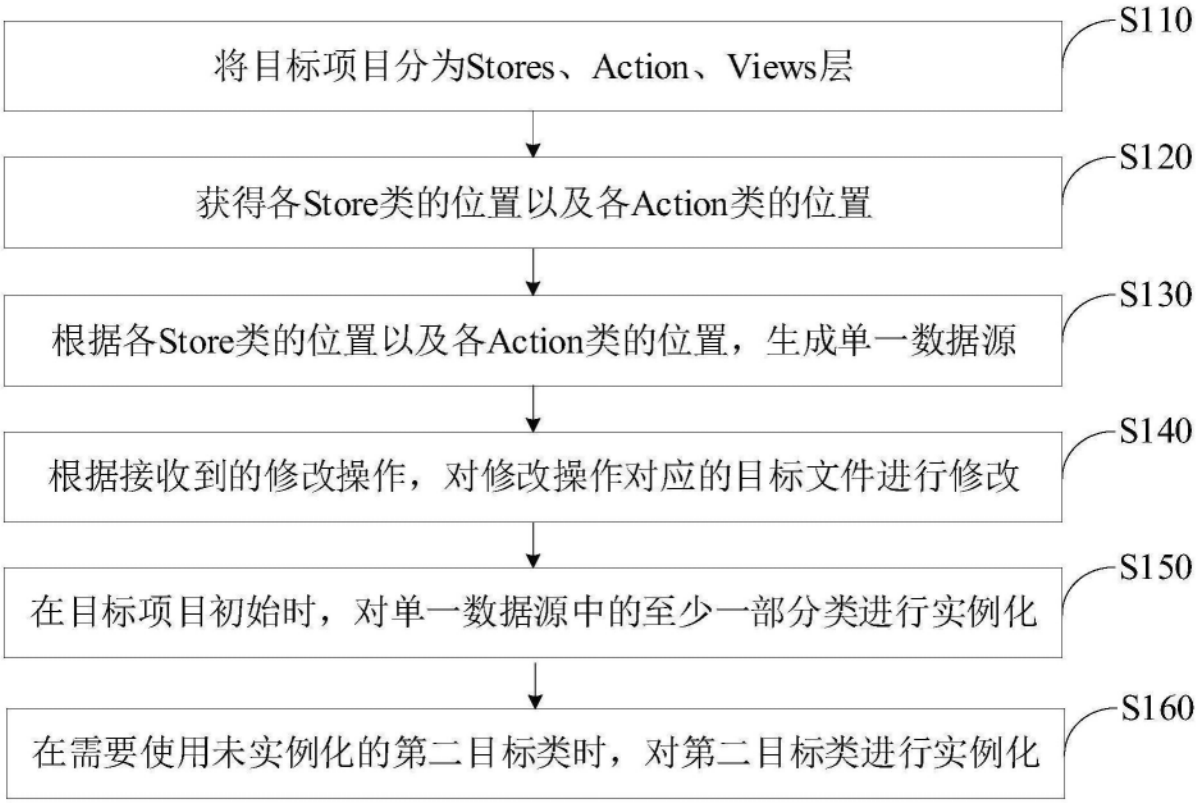

将目标项目分为Stores、Action、Views层
S110
获得各Store类的位置以及各Action类的位置
S120
根据各Store类的位置以及各Action类的位置，生成单一数据源
S130
根据接收到的修改操作，对修改操作对应的目标文件进行修改
S140
在目标项目初始时，对单一数据源中的至少一部分类进行实例化
S150
在需要使用未实例化的第二目标类时，对第二目标类进行实例化
S160

图10

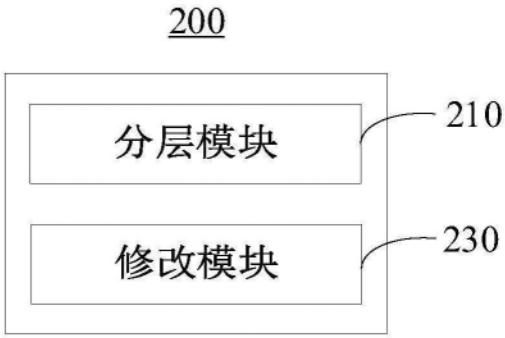

200
分层模块
210
修改模块
230

图11

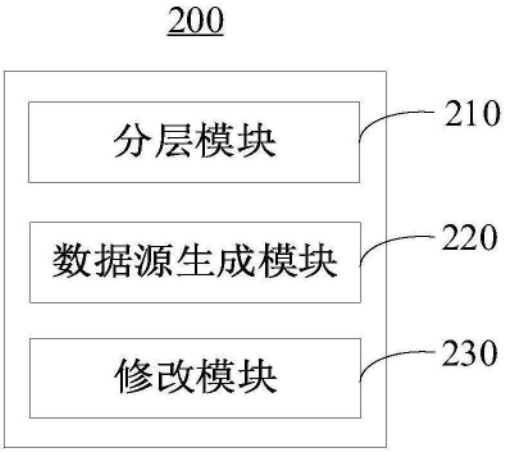

200
分层模块
210
数据源生成模块
220
修改模块
230

图12

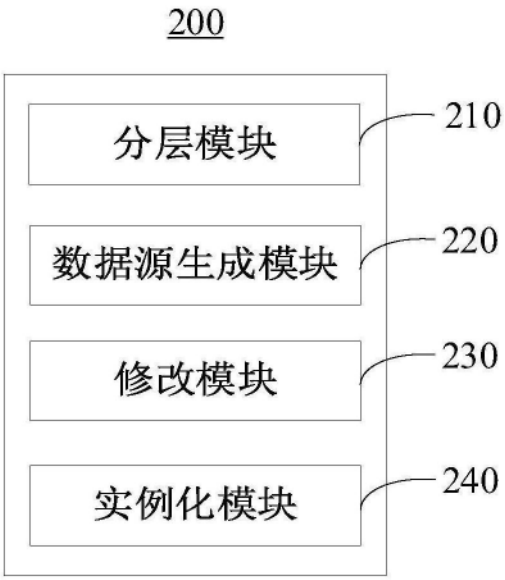

200
分层模块
210
数据源生成模块
220
修改模块
230
实例化模块
240

图13