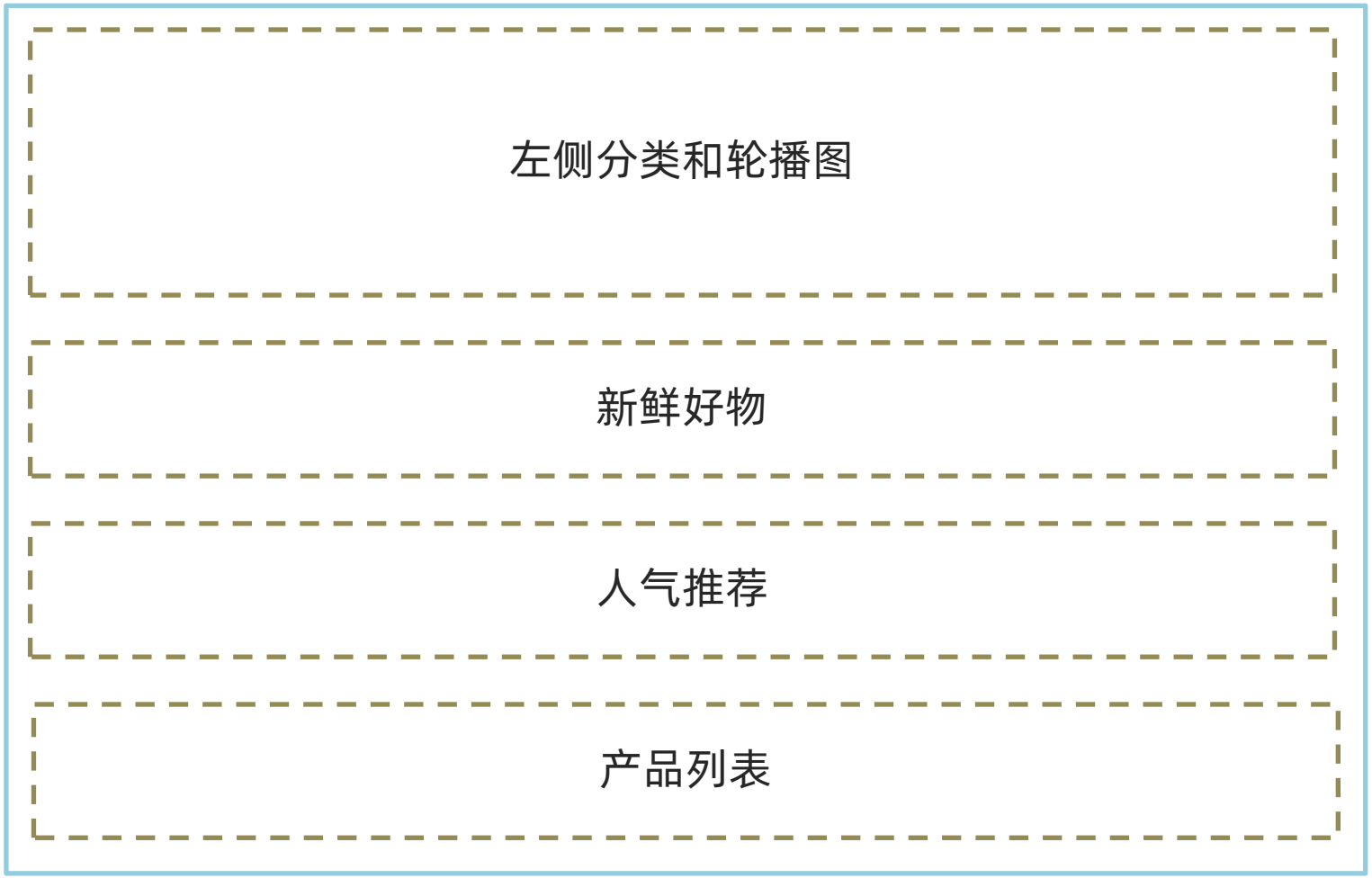
Home–整体结构搭建和分类实现

页面结构



分类实现

准备模版 → 使用Pinia中的数据渲染

Home-banner轮播图功能实现

轮播图实现

准备模版 → 熟悉elementPlus相关组件 → 获取接口数据 → 渲染组件

Home–面板组件封装

场景说明

问：组件封装解决了什么问题？

答：1. 复用问题　2. 业务维护问题

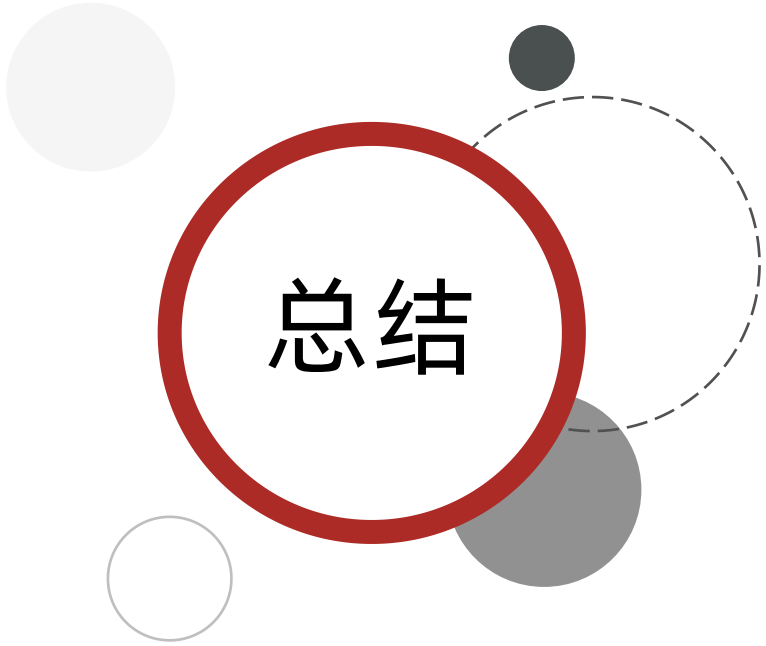
新鲜好物和人气推荐模块，在结构上非常相似，只是内容不同，通过组件封装可以实现复用结构的效果

组件封装

核心思路：把可复用的结构只写一次，把可能发生变化的部分抽象成组件参数（props / 插槽）

实现步骤

1. 不做任何抽象，准备静态模版
2. 抽象可变的部分
   - 主标题和副标题是纯文本，可以抽象成prop传入
   - 主体内容是复杂的模版，抽象成插槽传入



纯展示类组件通用封装思路总结：

1. 搭建纯静态的部分，不管可变的部分

2. 抽象可变的部分为组件参数

非复杂的模版抽象成props，复杂的结构模版抽象为插槽

04 Home–新鲜好物和人气推荐实现

新鲜好物实现

准备模版（HomePanel组件） → 定制Props → 定制插槽内容（接口 + 渲染模版）

05 Home–图片懒加载指令实现

场景和指令用法

场景：电商网站的首页通常会很长，用户不一定能访问到页面靠下面的图片，这类图片通过懒加载优化手段可以做到只有进入视口区域才发送图片请求

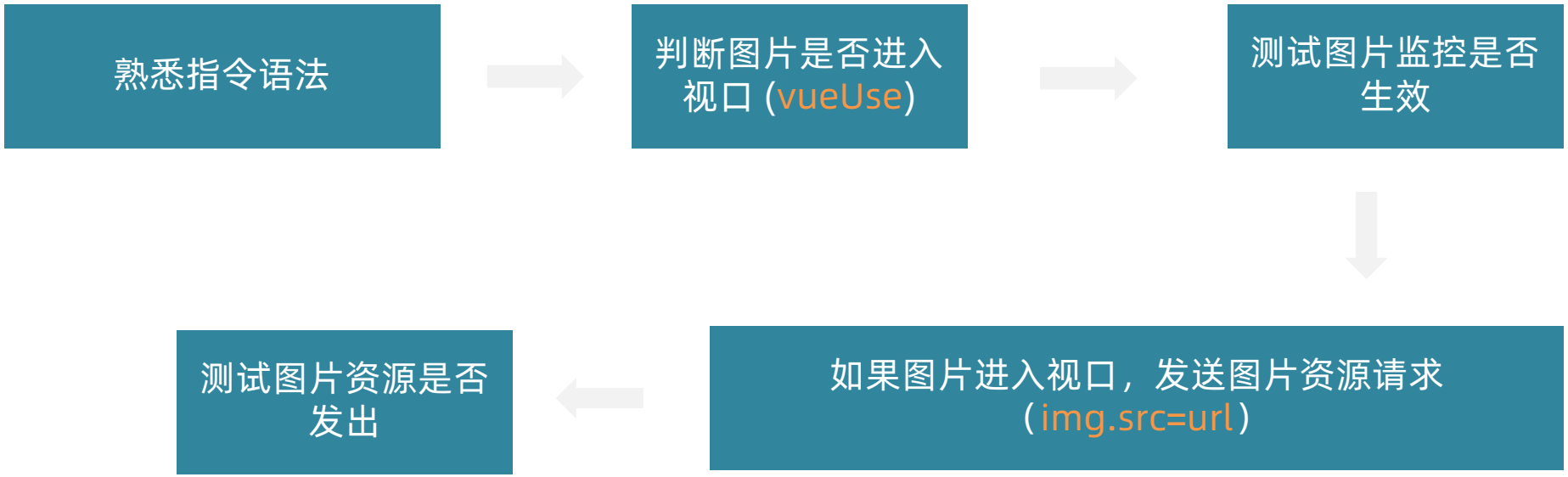
指令用法：

```
<img v-img-lazy="item.picture" />
```

在图片img身上绑定指令，该图片只有在正式进入到视口区域时才会发送图片网络请求

实现思路和步骤

核心原理：图片进入视口才发送资源请求



回顾核心步骤代码

1. 空指令实现

```
app.directive('img-lazy', {
  mounted (el, binding) {
    // el: 指令绑定的那个元素 img
    // binding: binding.value  指令等于号后面绑定的表达式的值  图片url
  }
})
```

```
<img v-img-lazy="item.picture" />
```

2. 指令逻辑实现

```
useIntersectionObserver(
      el,
      ([{ isIntersecting }]) => {
        console.log(isIntersecting)
        if (isIntersecting) {
          // 进入视口区域
          el.src = binding.value
        }
      },
    )
```

Home–懒加载指令优化

问题1：逻辑书写位置不合理

问：懒加载指令的逻辑直接写到入口文件中，合理吗？

答：不合理，入口文件通常只做一些初始化的事情，不应该包含太多的逻辑代码，可以通过插件的方法把懒加载指令封装为插件，main.js入口文件只需要负责注册插件即可

```
const directivePlugin = {
  install (app) {
    // 使用app实现懒加载指令逻辑
  }
}
```



```
import directivePlugin from '@/directives'
app.use(directivePlugin)
```

问题2：重复监听问题

useIntersectionObserver对于元素的监听是一直存在的，除非手动停止监听，存在内存浪费

解决思路：在监听的图片第一次完成加载之后就停止监听

```
const { stop } = useIntersectionObserver(
        el,
        ([{ isIntersecting }]) => {
          if (isIntersecting) {
            el.src = binding.value
            stop()
          }
        },
      )
```

Home–Product产品列表实现

Product产品列表

Product产品列表是一个常规的列表渲染，实现步骤如下：

08 Home–GoodsItem组件封装

为什么要封装GoodsItem组件

朱炳仁·铜忠义千秋...

众今礼敬关二爷，祈愿...

¥4280.00

换季好帮手，大容量...

棉被收纳，干净防尘

¥69.00

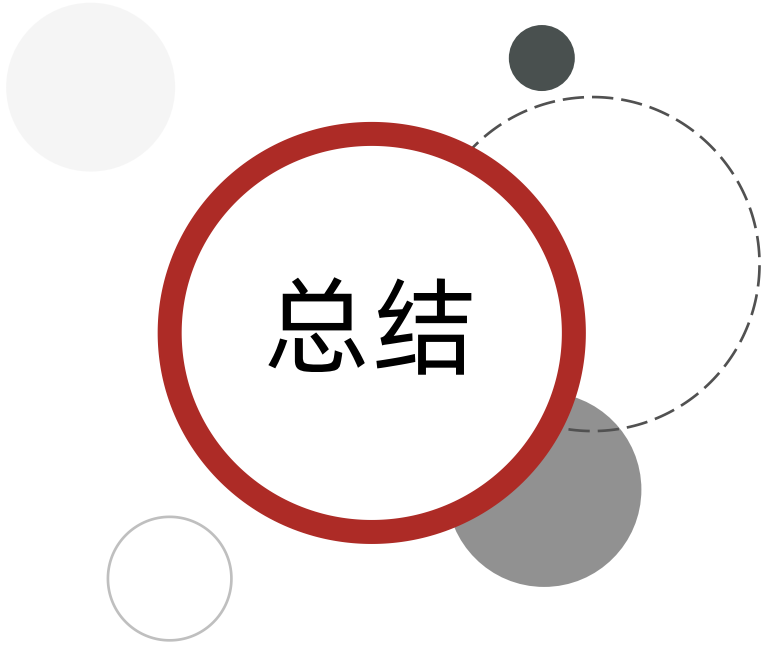
在小兔鲜项目的很多个业务模块中都需要用到同样的商品展示模块，没必要重复定义，封装起来，方便复用

如何封装

核心思想：把要显示的数据对象设计为props参数，传入什么数据对象就显示什么数据

```vue
<ul class="goods-list">
  <li v-for="goods in cate.goods" :key="goods.id">
    <RouterLink to="/" class="goods-item">
      <img v-img-lazy="goods.picture" alt="" />
      <p class="name ellipsis">{{ goods.name }}</p>
      <p class="desc ellipsis">{{ goods.desc }}</p>
      <p class="price">&yen;{{ goods.price }}</p>
    </RouterLink>
  </li>
</ul>
```

```vue
<ul class="goods-list">
  <li v-for="goods in cate.goods" :key="goods.id">
    <GoodsItem :goods="goods" />
  </li>
</ul>
```



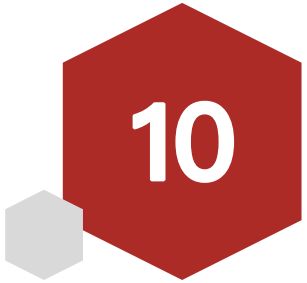
GoodsItem属于纯展示类组件，这类组件的封装思路是什么？

抽象Props参数，传入什么就显示什么

一级分类–整体认识和路由配置

路由配置

准备路由组件 → 配置路由（参数） → 修改导航组件 → 测试路由跳转



一级分类–面包屑导航渲染

面板屑导航渲染

准备组件模版 → 封装接口函数 → 调用接口获取数据（使用路由参数） → 渲染模版

一级分类–banner轮播图实现

分类轮播图实现

分类轮播图和首页轮播图的区别只有一个，接口参数不同，其余逻辑完成一致

请求参数

参数名	位置	类型	必填	说明
distributionSite	query	string	否	示例值： 1 说明： 广告区域展示位置（投放位置 投放位置，1为首页，2为分类商品页） 默认是1

改造先前的接口（适配参数） → 迁移首页轮播图逻辑

一级分类–激活状态显示和分类列表渲染

激活状态显示

首页　居家　美食　服饰　母婴　个护　严选　数码

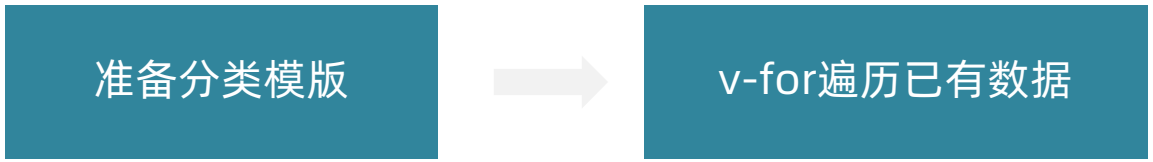
RouterLink组件默认支持激活样式显示的类名，只需要给active-class属性设置对应的类名即可

```
<RouterLink active-class="active" :to="`/category/${item.id}`">
  {{ item.name }}
</RouterLink>
```

```
.active {
  color: $xtxColor;
  border-bottom: 1px solid $xtxColor;
}
```

分类列表渲染

分类的数据已经在面包屑导航实现的时候获取到了，只需要通过v-for遍历出来即可



13

一级分类–解决路由缓存问题

什么是路由缓存问题

响应路由参数的变化

在 Vue School 上观看免费视频课程

使用带有参数的路由时需要注意的是，当用户从 `/users/johnny` 导航到 `/users/jolyne` 时，**相同的组件实例将被重复使用**。因为两个路由都渲染同个组件，比起销毁再创建，复用则显得更加高效。**不过，这也意味着组件的生命周期钩子不会被调用**。

问题：一级分类的切换正好满足上面的条件，组件实例复用，导致分类数据无法更新

解决问题的思路：1. 让组件实例不复用，强制销毁重建　2. 监听路由变化，变化之后执行数据更新操作

方案一：给router-view添加key

以当前路由完整路径为key的值，给router-view组件绑定

```
1 <RouterView :key="$route.fullPath" />
```

最常见的用例是与 `v-for` 结合：

```
template
<ul>
  <li v-for="item in items" :key="item.id">...</li>
</ul>
```

也可以用于强制替换一个元素/组件而不是复用它。当你想这么做时它可能会很有用：

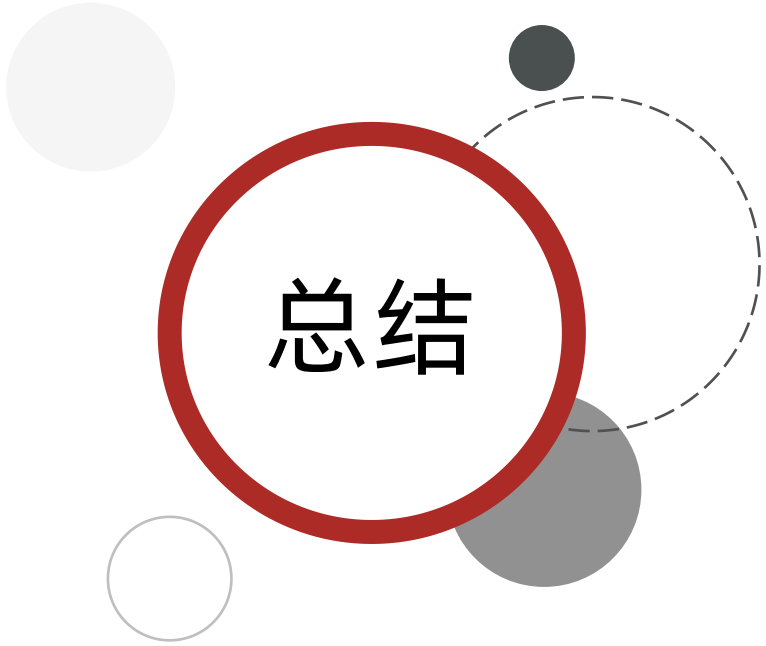
- 在适当的时候触发组件的生命周期钩子
- 触发过渡

方案二：使用beforeRouteUpdate导航钩子

beforeRouteUpdate钩子函数可以在每次路由更新之前执行，在回调中执行需要数据更新的业务逻辑即可

或者，使用 `beforeRouteUpdate` 导航守卫，它也可以取消导航：

```js
const User = {
  template: '...',
  async beforeRouteUpdate(to, from) {
    // 对路由变化做出响应...
    this.userData = await fetchUser(to.params.id)
  },
}
```



1. 路由缓存问题产生的原因是什么？

   路由只有参数变化时，会复用组件实例

2. 俩种方案都可以解决路由缓存问题，如何选择呢？

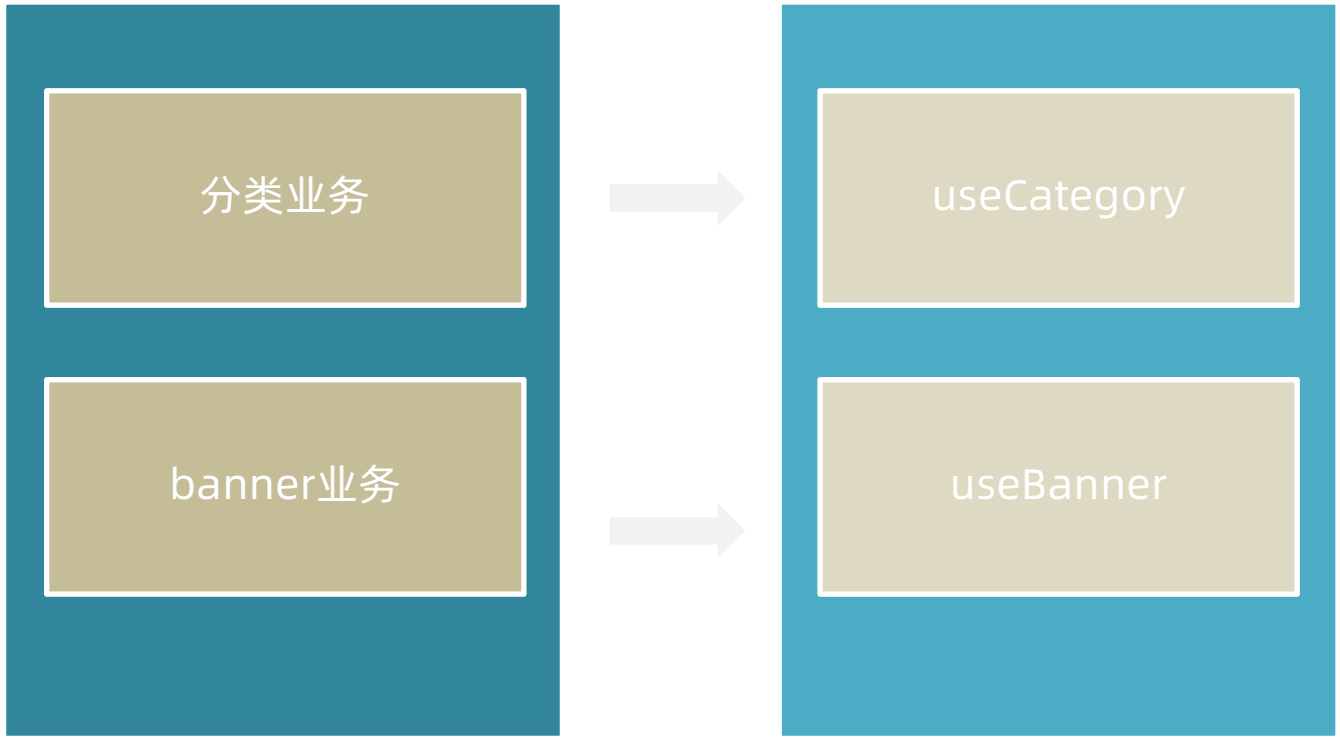
   在意性能问题，选择onBeforeUpdate, 精细化控制

   不在意性能问题，选择key，简单粗暴

一级分类–使用逻辑函数拆分业务

概念理解

基于逻辑函数拆分业务是指把同一个组件中独立的业务代码通过函数做封装处理，提升代码的可维护性



具体怎么做

实现步骤：

1. 按照业务声明以 `use` 打头的逻辑函数

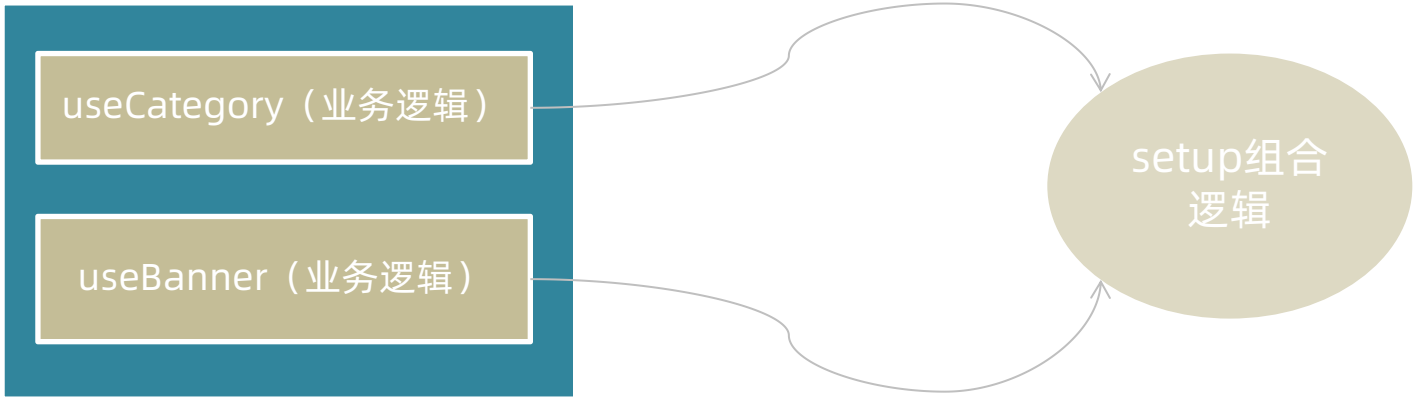
2. 把独立的业务逻辑封装到各个函数内部

3. 函数内部把组件中需要用到的数据或者方法return出去

4. 在组件中调用函数把数据或者方法组合回来使用

核心思想总结

1. 逻辑拆分的过程是一个拆分再组合的过程



2. 函数use打头，内部封装逻辑，return组件需要用到的数据和方法给组件消费

```
export function useBanner () {
  // 业务逻辑
  return {
    // 数据 + 方法
  }
}
```

```
const { bannerList } = useBanner()
const { categoryData } = useCategory()
```

传智教育旗下高端IT教育品牌