

# React DDD

领域驱动，各自只管各自的模块，顶层再来进行组装和分配

- 坚持根据特性区命名目录。
- 坚持为每个特性区创建一个 NgModule。

能提供限界上下文，将某些功能牢牢地锁在一个地方，开发某个功能时，只需要关心这个模块就够了。

视图的归试图，逻辑的归逻辑

```
function SomeComponent() {  
  const someService = useService();  
  return <div>{someService.state}</div>;  
}
```

跨组件数据传递？

```
function useGlobalService() {  
  return { state: "" };  
}  
  
const GlobalService = createContext(null);  
  
function SomeComponent() {  
  return (  
    <GlobalService.Provider value={useGlobalService()}>  
    </GlobalService.Provider>  
  );  
}  
  
function useSomeService() {  
  const globalService = useContext(GlobalService);  
  return <div>{globalService.state}</div>;  
}
```

上下文注入节点，本身就是按照试图来的

# 函数 DDD

只用函数实现 DDD，它有多优美

我们先比较一下这两种写法，对于一个类：

```
class SomeClass {
  name:string,
  password:string,
  constructor(name,password){
    this.name = name
    this.password = password
  }
}

const initValue = { name: "", password: "" };
function useClass() {
  const [state, setState] = useState(initValue);
  return { state, setState };
}
```

下面的自带响应式，getter，setter 也自动给出了，同时使用了工厂模式，不需要了解函数内部的逻辑。

## 生命周期复用

每个 useFunc 都是 拆掉 的管道函数，框架帮你组装，简直就是一步到位！

## 效率

```
function useSomeService() {
  const [form] = useForm();
  const request = useRequest();
  const model = useModel();
  useEffect(() => {
    form.onFieldsChange = () => {
      request.run(form.getFieldsValue);
    };
  }, [form]);
  return {
    model,
    form,
  };
};
```

```
}  
<Form form={someService.form}>  
  <Form.Item name="xxx" label="xxx">  
    <!-- 没你service啥事了! 别看! 这里是纯视图 -->  
  </Form.Item>  
</Form>
```

这个表单服务你想要在哪控制?

想要多个组件同时控制?

加个 token, 也就是 createContext, 把依赖提上去!

他特么自然了!

# Hooks 版本架构

## 执行 LIFT 原则

- 顶层文件夹最多包含: assets, pages, layouts, app 四个 (其中 pages, layouts 是为了照顾某些 ssr 开发栈), 名字可以变更, 但是不可以有多余文件夹, 激进的话可以只有一个 app 文件夹
- 按功能划分文件夹, 每个功能只能包含以下四种文件: Xxx.less, Xxx.tsx, useXxx.ts, useXxx.spec.ts, 采用嵌套结构组织
- 一个文件夹包含该领域内所有逻辑 (视图, 样式, 测试, 状态, 接口), 禁止将逻辑放置于文件夹以外
- 如果需要由其他功能调用, 利用 SOA 反转  
为何如此?
- 功能结构即文件结构, 开发人员可以快速定位代码, 扫一眼就能知道每个文件代表什么, 目录尽可能保持扁平, 既没有重复也没有多余的名字
- 当有很多文件时 (例如 10 个以上), 在专用目录型结构中定位它们会比在扁平结构中更容易
- 惰性加载可路由的功能变得更容易
- 隔离、测试和复用特性更容易
- 管理上, 相关领域文件夹可以分配给专人, 开发效率高, 可追责和计量工作量, 很明显应该禁止多人同时操作同一层级文件

- 只需要对 useXxx 进行测试，测试复杂度，工作量都很小，视图测试交给 e2e

## 利用 SOA 实现跨组件逻辑复用

利用 注入令牌 + 服务函数 + 注入点，实现灵活的 SOA

命名格式为

```
XxxService = useToken(useXxxService)
```

XxxService 为注入令牌 和文件名

useXxxService 为服务函数

```
<XxxService.Provider value={useXxxService()} />
```

XxxService.Provider 为注入点

注入令牌与服务函数紧挨

与注入节点处于同一文件结构层级

禁止除 SOA 以外的所有数据源

---

为何如此？

- 符合单一数据，单一职责，接口隔离原则
- 通过泛型约束，可以有更加自然的 Typescript 体验，不需要手动声明注入数据类型，所有类型将自动获得
- 层次化注入，可以实现 DDD，将逻辑全部约束与一处，方便团队协作
- 当你在根注入器上提供该服务时，该服务实例在每个需要该服务的组件和服务中都是共享的。当服务要共享方法或状态时，这是数学意义上的最理想的选择。
- 配合组件和功能划分，可以方便处理嵌套结构，防止对象复制被滥用，类似深复制之类的操作应该禁止

实现一个 IOC 注入令牌的方法为

```
import { createContext } from 'react';

/**
 * 泛型约束获取注入令牌
 *
 * @export
 * @template T
```

```

* @param {...args: any[]} => T} func
* @param {(T | undefined)} [initialValue=undefined]
* @returns
*/
export default function useToken<T>(
  func: (...args: any[]) => T,
  initialValue: T | undefined = undefined,
) {
  return createContext(initialValue as T);
}

```

一个典型可注册服务为：

```

import { useState } from "react";
import useToken from "../useToken";


export const AppService = useToken(useAppService);

export default function useAppService() {
  // 可注入其他服务，以嵌套
  // eq:
  // const someOtherService = useSomeOtherService(SomeOtherService)
  const [appName, setAppName] = useState("appName");
  return {
    appName,
    setAppName,
    // ...
  };
}

```

## 最小权限

人为保证代码结构，各个组成之间的最小权限，是一个好习惯

- 所有大写字母开头的 `tsx` 文件都是组件
- 所有 `use` 开头的文件，都是服务，其中，`useXxxService` 是可注入服务，默认将所有组件配套的服务设置为可注入服务，可以方便进行依赖管理
- 禁止在组件函数种出现任何非服务注入代码，禁止在组件中写入与视图不想关的
- 为复杂结构数据定义 `class`
- 如果可以的话，将单例服务由全局 `service` 组织，嵌套结构，共享实例，页面初始化 除外
-  禁止深复制

为何如此？

- 当逻辑被放置到服务里，并以函数的形式暴露时，可以被多个组件重复使用
- 在单元测试时，服务里的逻辑更容易被隔离。当组件中调用逻辑时，也很容易被模拟
- 从组件移除依赖并隐藏实现细节
- 保持组件苗条、精简和聚焦
- 利用 `class` 可以减少初始化复杂度，以及因此产生的类型问题
- 局管理单例服务，可以一步消灭循环依赖问题（道理同 `Redux` 替代 `Flux`）
- 深复制有非常严重的性能问题，且容易产生意外变更，尤其是 `useEffect` 语境下

## JUST USE REACT HOOKS

抛弃 `class` 这样的，`this` 挂载变更的历史方案，不可复用组件会污染整个项目，导致逻辑无法集中于一处，甚至出现耦合，`LIFT`，`SOA`，`DDD` 等架构无从谈起

### 项目只存在

- 大写并与文件同名的组件，且其中除了注入服务操作外，`return` 之前，无任何代码
- `use` 开头并与文件夹同名的服务
- `use` 开头，`Service` 结尾，并与文件夹同名的可注入服务
- 服务中只存在 基础 `hooks`，自定义 `hooks`，第三方 `hooks`，静态数据，工具函数，工具类

### 以下为细化阐述为何如此设计的出发点

- 快速定位 `Locate`
- 一眼识别 `Identify`
- 尽量保持扁平结构 (`Flattest`)
- 尝试 `Try` 遵循 `DRY (Do Not Repeat Yourself)`，不重复自己)

### 此为 `LIFT` 原则

- 优先将组件视为元素，而并非功能逻辑单位（视图的归视图，业务的归业务）
- 隔离原则（属于一个成员的工作，必定属于该成员负责的文件夹，也只能属于该成员负责的文件夹）
- 最小依赖（禁止不必要的工具使用，比如当前需求下，引入 `Redux/Flux/Dva/Mobx` 等工具，并没有解决什么问题，却导致功能更加受限，影响隔离原则比如当两个组件需要服务的不同实例的情况，以上工具属于上个版本或某种特殊需求，比如前后端同构，不能影响这个版本当前需求的架构）
- 优先响应式（普及管道风格的函数式方案，大胆使用 `useEffect` 等 `api`，不提倡松散的函数组合，只要是视图所用的数据，必须全部都为响应式数据，并响应变更）
- 测试友好（边界清晰，风格简洁，隔离完整，即为测试友好）

- 设计友好（支持模块化设计）

## 建议的技术栈搭配

- create-react-app + react-router-dom + antd + ahooks + styled-components（大多数场景下，强烈推荐！可以上 ProComponent，但是要注意提取功能逻辑，不可将逻辑写于组件）
- umi + ahooks（请删除 models, services, components, utils 等非必要顶层文件夹，禁止使用 dva）
- umi(ssr) + dva + ahooks（同上，但可仅基于 dva 沟通前后端和首屏数据，非 ssr 同样禁用 dva）
- next.js + react suite/material ui + swr（利用不到 useAntdTable 之类的功能，ahooks 就鸡肋了）

## Hook 使你在无需修改组件结构的情况下复用状态逻辑

当你思维聚焦于组件时，在这种情况下，你是必须逼迫自己，在组件里写业务逻辑，或者重新组织业务逻辑！

并且，因为 state 是不反应业务逻辑的，它也天然不可以对业务逻辑进行组合

```
function useSomeTable() {
  // 这个是个表单，抽象的
  const [form] = Form.useForm();
  // 这个是个表单联动的表格
  const { tableProps, loading } = useAntdTable(
    // 自动处理分页相关问题
    ({ current, pageSize }, formData) => fetch("http://sdfdsfsdf"), // 抽象的状态请求
    {
      form, // 表单在这里与表格组合，实现联动
      defaultParams: {
        // ...
      },
      // 很多功能都能集成，只需要一个配置项
      debounceInterval: 300, // 节流
    }
  );
  return {
    form,
    loading,
    tableProps,
  };
}
```

```
}  
<Form form={SomeTable.form}><!--里面全部状态无关，不用看了--></Form>  
<Table {...tableProps} columns={} rowKey="id"/>
```

你能将视图和逻辑完全组织为一个结构，交给一个特定的人，完全不用关心他到底是怎么开发的

这便是 —— 逻辑视图分离👍

# React SOA

## 基本的服务

```
function useSimpleService() {  
  const [val1, setVal1] = useState(0);  
  const [val2, setVal2] = useState(0);  
  
  useEffect(() => {  
    setVal2(val1);  
  }, [val1]);  
  
  return {  
    val1,  
    setVal1,  
    val2,  
  };  
}
```

- 叫它 **service**，是 SOA 模型下的管用叫法，意思是 —— 我只会在这样的结构种写逻辑，组件中的逻辑全部消失（优先将组件视为元素）
- 只暴露你需要暴露的状态逻辑（状态逻辑必须一起说，只做状态复用很扯淡，毕竟 2021 年了）
- **useRef**，同样也可以封装在 **Service** 中，而且建议如此做，**ref** 的获取不是视图，是逻辑



# 组合服务

有另外一个服务，useAnotherService

```
function useAnotherService() {  
  const [val, setVal] = useLocalStorage(0);  
  return { val, setVal };  
}
```

然后与基本服务进行组合

```
function useSimpleService() {  
  const [val1, setVal1] = useState(0);  
  const [val2, setVal2] = useState(0);  
  const { setVal } = useAnotherService();  
  
  useEffect(() => {  
    setVal2(val1);  
  }, [val1]);  
  
  useEffect(() => {  
    setVal(val2);  
  }, [val2]);  
  
  return {  
    val1,  
    setVal1,  
    val2,  
  };  
}
```

就能为基本服务动态添加功能

- 为什么不直接 import? 因为需要框架内的响应式能力，这个叫控制反转，框架将响应式的控制权转交给了开发者
- 如果有另外一个服务，单单只要 AnotherService 的功能，你只需要调用 useAnotherService 就好了
- 最好是调用者修改被调用者，可以对比 ahooks 对代 useRef 的改动，就是本着这个次序，因为被调用者可能被多次调用，保证复用性
- useEffect 是一种管道模型，如同 rxjs 一般，只是框架帮你按顺序组装而已（你以为为啥非要你按顺序来？），是极限的函数式方案，不存在纯度问题，函数式得不要不要的。但是有个要求，依赖必须写清楚，这个依赖是管道操作中的参数，React 将你的 hook 重新组合成了管道，但是参数必须提供，在它自动分析依赖之前
- 使用了 useAnotherService 的细节被隐藏，形成了一个树形调用结构，这种结构被

称作“依赖树”或者“注入树”，别盯着我，名字不是我定的

## 注入单例服务

当前服务如果需要被多个组件使用，服务会被初始化很多次，如何让它只注入一次？

利用 createContext

```
export const SimpleService = createContext(null);
export default function useSimpleService() {
  // ...
}
```

但是，单例需要注入到唯一节点，因此，你需要在所有需要用到这个服务的组件的最顶层：

```
<SimpleService.Provider value={useSimpleService()}>
  {props.children}
</SimpleService.Provider>
```

这样，这个服务的单例就对所有子孙组件敞开了怀抱，同时，所有子孙组件对其的修改都将生效

```
function SomeComponent(){
  const {val1, setVal1} = useContext(SomeService)
  return <div onClick={()=>{setVal1('fuck')}}>val1</div>
}
```

- 直接在 jsx 的 provider 种 value = {useSomeService()} 在本组件没有任何其它响应式变量的情况下是可行的，因为不会重新初始化，在良好的架构下 —— 组件除注入，无任何逻辑，return 之前没有东西，同时，上下文单独封装组件，可以作为“模块标识”
- 这个有共同单例 Service 的一系列组件，被称为模块，它们有自己的“限界上下文”，并且，视图，逻辑，样式都在其中，如果这个模块是按照功能划分的，那么这种 SOA 实现被称为 领域驱动设计 (DDD)，某些架构强推的所谓‘微前端’，目的就是得到这个东西
- 一定要注意，这个模块的上层数据变更，模块的限界上下文会刷新，这个是默认操作，这也是为何 jsx 直接赋值的原因，如果你不需要这个东西，可以采用 const value = useService() 包裹，或者直接 memo 这个模块标识组件

# 单例服务

## 解决深层嵌套对象问题

深层嵌套对象怎么处理？useReducer？immutable？还是直接深复制？

你首先明白你要实现什么逻辑，深层嵌套对象之所以难处理，是因为你想在子组件实现 **对深层目标的部分变更逻辑**

之前你之所以有这些奇奇怪怪工具甚至深复制的需求，是因为你没有办法将逻辑也拆分给子组件，明白为什么如此

现在，逻辑可以拆分复用：

```
function useSomeService() {
  const [value, setValue] = useState({
    username: "",
    password: "",
    info: {
      nickname: "",
      others: [],
    },
  });
  return { value, setValue };
}
// 注入部分省略...
```

修改 info:

```
setValue((res) => {
  res.info.nickname === "fuck";
  return res;
});
```

配合 map 修改数组：

```
// 分形部分：
new Array(5).map((_, key) => <SomeCompo index={key} key={key} />);
// 组件
function SomeComponent(props) {
  const { setValue } = useContext(SomeService);
  return (
    <div
      onClick={() => {
```

```

        setValue((res) => {
            res.info.others[props.index] = "fuck";
            return res;
        });
    }}
</div>
);
}

```

如果需要划分模块，通过 `getter`, `setter` 传递这个嵌套结构：

```

function subInjectedService() {
    const { value, setValue } = useContext(SomeService);
    const info = useMemo(() => value.info.others, [value]);
    const setInfo = useCallback((val) => {
        setValue((res) => {
            res.info.others[props.index] = val;
            return res;
        });
    }, []);
    return {
        info,
        setInfo,
    };
}
// 忽略注入部分...

```

这样的话，这个重新划分的模块内部，想要修改上层的数据，只需要通过 `info`, `setInfo` 即可

- 不用担心纯度和不变性的问题，因为 `hooks` 都是纯的，没有不纯的情况
- 全局副作用是状态 + 函数全局逻辑封装（分层）考虑的问题，将函数和组件，视图功能逻辑样式全部作为模块，副作用是以模块为单位的，而 `info` 和 `setInfo` 的 `getter`, `setter` 封装，叫做 —— 模块间通讯
- `useReducer` 只涉及调试，也就是有个 `action` 名字方便你定位问题，模块划分如果足够细，你根本不需要这个 `action` 来记录你的变更，采用 `useReducer` 与 DDD 原则背离，但是也不会禁止。不过，全局 `useReducer` 必须明令禁止，这种方式是个灾难，`useReducer` 必须是以模块为单位，不能更小，也不能更大
- 组件和服务一起，处理一部分数据，保证了单例修改，不变性也不用担心，`hooks` 来保证这个
- 在这里，你会发现 `props` 的功能好像只有‘分形’，也就是 `map` 种将数据的标识传递给子组件，是的 —— 优先使用服务共享状态逻辑
- `getter`, `setter` 叫做响应式，如果你不需要响应式修改，`setter` 可以删除，但是 `getter` 同时还有防止重新渲染的作用，保留即可，除非纯组件

# 服务获取时的类型问题

如果你使用的是 Typescript，那么，用泛型约束获得自动类型推断，会让你如虎添翼

```
import { createContext } from 'react';  
/**  
 * 泛型约束获取注入令牌  
 *  
 * @export  
 * @template T  
 * @param {...args: any[]} => T} func  
 * @param {(T | undefined)} [initialValue=undefined]  
 * @returns  
 */  
export default function useToken<T>(  
  func: (...args: any[]) => T,  
  initialValue: T | undefined = undefined,  
) {  
  return createContext(initialValue as T);  
}
```

然后将 `createContext()` 改为 `useToken(SomeService)` 即可，这样你就拥有了指哪打哪的类型支持，无需单独的类型声明，代码更加清爽

- 如果是 Javascript 环境，建议老老实实写 `createContext` 的 `defaultValue`，虽然注入之后，子孙组件都不会出现 `defaultValue`，但是 javascript 语境下有代码提示
- 不建议 typescript 下声明 `defaultValue`，因为模块外的服务调用，应该被禁止，这是 DDD 架构的基础，如果你想要在外使用单例服务 —— 请将其提升至外部

## 顶层注入服务

平凡提升模块服务层级，可能会产生循环依赖，而且会影响模块的封装度，因此：

**⚠️ 优先思考清楚自己应用的模块关系！**

循环依赖产生根源是功能领域，功能模块划分有问题，优先解决根本问题，而不是转移矛盾。如果你实在思考不清楚，又想要立刻开始开发，那么可以尝试顶层注入服务：

```
function useAppService(){
  return {
    someService:useSomeService()
    anotherService:useAnotherService()
  }
}
```

- 模块间进行嵌套组合将变得无比困难，不再是一个 getter，setter 能够搞定的，如果不是绝对的必要，尽量不要采用此种方式！它有悖于 DDD 原则 —— 分治
- 多组件共享不同实例将彻底失败，这不是你愿意看到的

## 可选服务

模块服务划分的另一个巨大优势，就是将逻辑变为可选项，这在重型应用中，几乎就是采用 DDD 的关键

```
function useServiceByOneLogic() {
  return {
    activated,
    // ...
  };
}

function useServiceByAnotherLogic() {
  return {
    activated,
    // ...
  };
}

function useSomeService() {
  const [...serviceList] = [useServiceByOneLogic(),
useServiceByAnotherLogic()];
  // 选择激活的服务
  const usedService = useMemo(() => {
    for (let service of serviceList) {
      if (service.activated === true) {
        return service;
      }
    }
  }, [serviceList]);
  return service;
}

// 注入过程省略...
```

- 你也可以通过各种条件筛选服务，这种方式是在前端实现的高可用
- △ 注意，服务最好只是内部实现不同，接口应该尽可能相同，否则会出现可选类型
- 最典型的应用，就是多家云服务厂商的短信验证（验证码，人机校验等），通过可选服务根据用户网络情况进行筛选，用最适合当前用户的那一个
- 还有一个非常有意思的方案，通过服务来做数据 **mock**，因为服务直接对接视图，你只需要模拟视图数据即可，提供两个服务，一个真实服务，一个 **mock** 服务，这样是用真实数据还是 **mock** 数据，都是服务自动判断的，对你来说没有流程差别

## 样式封装

注意，模块是包含了样式的，上文在讲述逻辑和视图的封装，接下来说说样式

- 典型的 `cssModule`，`styled-components` 之类的方案
- `shadowDom`，仿真样式（`Angular` 原生支持，`React` 可以用 `cssModule` 之类工具间接实现），可以实现跨技术栈样式封装（没错，所谓‘微前端’的样式封装）
- 样式最好只包含排版，企业 **vis** 统一性是标准，没有必要违背这个

## 继续分析 SOA

从上一篇文章的例子可以看出什么呢？

首先，按照功能领域划分文件，可以很快分析出应用的逻辑结构

也就是逻辑可读性更强，这个可读性不只是针对用户的，还有针对软件的

比如，`TodoService` 和 `TableHandlerService` 有什么关系？

## `useTableHandlerService`

`useTableHandlerService`

`useTodoService`

这些逻辑关系，仅仅依靠相关工具就能定位，并生成图形，辅助你分析领域间的关系

谁依赖谁，一目了然 —— 比如 有个 `useState` 的值 依赖 `useLocalStorageState`，肉眼看起来比较困难，但是在图中一目了然

只是，不具名这一点有点神烦！

还有，`React` 内部因为没有管理好这个部分传递，没办法像 `Angular` 一样，瞬间生成一大堆密密麻麻的依赖树，这就给 `React` 在大项目工程化上带来了阻碍

不过一般项目做不到那么大，领域驱动可以帮助你做到 **Angular** 项目极限的 95%，剩下那 5%，也只是稍稍痛苦些而已，并且，没有办法给管理者看到完整蓝图

不过就国内目前前端技术管理者和产品经理的水品，你给他们看 **uml** 蓝图，我担心他们也看不懂，所以这部分不用太在意，感觉有地方依赖拿不准，只显示这个领域的蓝图就好

其次，测试边界清晰，且易于模拟

视图你不用测试，因为没有视图逻辑，什么时候需要视图测试？比如 **Form** 和 **FormItem** 等出现嵌套注入的地方，需要进行视图测试，这部分耦合出现的概率非常小，大部分都是第三方框架的工作

你只需要测试这些 **useFunction** 就好，并且提供两个个框，比如空组件直接 **use**，嵌套组件先 **provide** 再 **useContext**，然后直接只模拟 **useFunction** 边界，并提供测试，大家可以尝试一下，以前觉得测试神烦，现在可以试试在清晰领域边界下，测试可以有多愉悦

最后

谁再提状态管理我和谁急！

你看看这个应用，哪里有状态管理插手的地方？任何状态管理库都不行，它是上个时代的遮羞布

---

## 服务间通讯结构

### 全局单一服务（类 *Redux* 方案）

但是，单一服务是不得已而为之，老版本没有逻辑复用导致的

在这种方式下，你的调试将变得无比复杂，任何一处变更将牵扯所有本该封装为模块的组件

所以必须配合相应的调试工具

所有多人协作项目，采用此种方式，最后的结果只有项目不可维护一条路！



## 中台+ 其他服务（双层结构

由一个，appService 提供基础服务，并管理服务间的调度，此种方式比第一种要好很多，但是还是有个问题，顶层处理服务关系，永远比服务间处理服务关系来的复杂，具体问题详见上文“顶层注入”

## 树形结构模块

这是理论最优的结构，它的优势不再赘述，上文有提到

劣势有一个：

跨模块层级的变更，容易形成循环依赖（也不叫劣势，因为此种变更对于其他方式来说，是灾难）

理清自己的业务逻辑，有必要划出功能结构图，再开始开发，是个好习惯，同时，功能层级发生改变，应该敏锐意识到，及时提升服务的模块层级即可

---

## 编程范式

首先，编程范式除了实现方式不同以外，其区别的根源在于 — 关注点不同

- 函数的关注点在于 — 变化
- 面向对象的关注点在于 — 结构


对于函数，因为结构方便于处理变化，即输入输出是天然关注点，所以 —

管理状态和副作用很重要

js

```
var a = 1;
function test(c) {
  var b = 2;
  a = 2;
  var a = 3;
  c = 1;
  return { a, b, c };
}
```

这里故意用 `var` 来声明变量，让大家又更深的体会

在函数中变更函数外的变量 —— **破坏了函数的封装性** 

这种破坏极其危险，比如上例，如果其他函数修改了 `a`，在重新赋值之前，你知道 `a` 是多少么？如果函数很长，你如何确定本地变量 `a` 是否覆盖外部变量？

无封装的函数，不可能有可装配性和可调试性

所以，使用函数封装逻辑，不能引入任何副作用！注意，这个是强制的，在任何多人协作，多模块多资源的项目中 ——

**封装是第一要务，更是基本要求！**

所以，你必须将数据（或者说状态）全部包裹在函数内部，不可以在函数内修改任何函数以外的数据！

所以，函数天然存在一个缺点 —— 封装性需要人为保证（即你需要自己要求自己，写出无副作用函数）

当然，还存在很多优点 —— 只需要针对输入输出测试，更加符合物体实际运行情况（变化是哲学基础）

这部分没有加重点符号，是因为它不重要 —— 对一个思想方法提优缺点，只有指导意义，因为思想方法可以综合运用，不受限制

再来看看面相对象，来看看类结构：

```
class Test {  
  a = 1;  
  b = 2;  
  c = 3;  
  constructor() {  
    this.changeA();  
  }  
  changeA() {  
    this.a = 2;  
  }  
}
```

这个结构一眼看去就具有 —— 自解释性，自封装性

还有一个涉及应用领域的优势 —— 对观念系统的模拟 —— 这个词不打着重符，不需要太关心，翻译过来就是，可以直译人脑中的观念（动物，人，车等等）

但它也有非常严重的问题 —— 初始化，自解耦麻烦，组合麻烦

需要运用到大量的‘构建’，‘运行’设计模式！

对的，设计模式的那些名字就是怎么来的

其实，你仔细一想，就能明白为什么会这样 ——

如果你关注变化，想要对真实世界直接模拟，你就需要处理静态数据，需要自己对一个领域进行人为解释如果你关注结构，想要对人的观念进行模拟，你就需要处理运行时问题，需要自己处理一个运行时对象的生成问题

鱼与熊掌，不可兼得，按住了这头，那头就会翘起来，你按住了那头，这头就会翘起来

想要只通过一个编程范式解决所有问题，就像用手去抓沙子，最后你什么都得不到

## 函数式面向对象

通过函数和对象（注意是对象，类是抽象的，观念中的对象）的分析，很容易发现他们的优势

函数 —— 测试简单，模拟真实（效率高）

对象 —— 自封装性，模拟观念（继承多态）

将两者发扬光大，更加极限地使用，你会得到以下衍生范式：

管道 / 流

既然函数只需要对输入输出进行测试，那么，我将无数函数用函数串联起来，就形成了只有统一输入输出的结构

听不懂？换个说法 ——

只需要 e2e 测试，不需要单元测试！

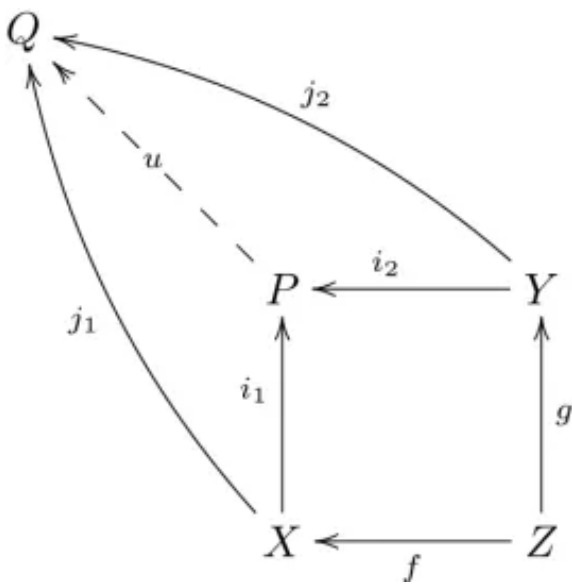
如果我加上类型校验，就可以构造出 —— **理想无 bug 系统**

这样的话，你就只剩调试，没有测试（如果顶层加个校验取代 e2e 的话）

而且，还有模式识别，异步亲和性等很多好处，甚至可以自建设计语言（比如麻省老教材《如何设计计算机语言》就是以 **lisp** 作为标准）

在 js 中，**Cycle.js** 和 **Rxjs** 就是极限的管道风格函数式，还有大家熟悉并且讨厌的 **Node.js** 的 **Stream** 也是如此，即便他是用 类 实现的，骨子里也是浓浓的函数式

分析一下这样的系统，你会发现 ——  
它首先关注底层逻辑 —— 也就是 `or/c` , `is-a` , `and/c` , `not/c` 这样的函数，最后再组装  
按照范畴学的语言：



范畴学

`u of i2` , `i2 of g` 的讲法，与它的真实运行方向，是相反的！

函数的组合方式，与开发目标的构建方式，也是相反的！

它的构建方法叫做 —— 自底向上

这也是为啥你在很多 JS 的库中发现了好多零零碎碎的东西，还有为何会有 `lodash` , `ramda` 等粒度非常小的库了

在极限函数式编程下 ——

我先做出来，再看能干什么，比先确定干什么，再做，更重要！

因为这部分，可以第三方甚至官方自己提供！

所以，**函数式是库的第一优先级构建范式**！因为作为库的提供者，你根本不可能预测用户会用这个库来干什么

领域模块

函数式可以将其优势通过管道发挥到极致，面向对象一样可以将其优势发挥到极致，这便是领域模块

领域，就是一系列相同目的，相同功能的资源的集合

比如，学校，公司，这两个类，如果分别封装了大量的其他类以及相关资源，共同构成一个整体，自行管理，自行测试，甚至自行构建发布，对外提供统一的接口，那这就是领域

这么说，如果实现了一个类和其相关资源的自行管理，自行测试，这就是 —— **DDD**

如果实现了对其的自行构建发布，这就是 —— **微服务**

这种模型给了应用规模化的能力 —— 横向，纵向扩展能力

还有高可用，即类的组合间的松散耦合范式

对于这样的范式，你首先思考的是 —— 你要做什么！

这就是 ——

**\*\*** 这种模型给了应用规模化的能力 —— 横向，纵向扩展能力

还有高可用，即类的组合间的松散耦合范式

对于这样的范式，你首先思考的是 —— 你要做什么！

这就是 —— **自顶向下**

- 我要做什么应用？
- 这个应用有哪些功能？
- 我该怎么组织我的资源和代码？
- 该怎么和其他职能合作？
- 工期需要多久？

现实告诉你，单用任何一种都不行开发过程中，不止有自底向上封装的工具，还有自顶向下设计的结构

产品经理不会把要用多少个 `isObject` 判断告诉你，他只会告诉你应用有哪些功能

同理，再丰富细致的功能划分，没有底层一个个工具函数，也完成不了任何工作

这个世界的确处在变化之中，世界的本质就是变化，就是函数，但是软件是交给人用，交给人开发的

**观念系统和实际运行，缺一不可！**

凡是动不动就跟你说某某框架函数式，某某应用要用函数式开发的人，大多都学艺不精，根本没有理解这些概念的本质

人类编程历史如此久远，真正的面向用户的纯粹函数式无 **bug** 系统，还没有出现过.....

当然，其在人工智能，科研等领域有无可替代的作用。不过，是人，就有组织，有公司，进而有职能划分，大家只会通过观念系统进行交流——你所说的每一个词汇，都是观念，都是类！

## React 提倡函数式

```
class OOStyle {
  name: string;
  password: string;
  constructor() {}
  get nameStr() {}
  changePassword() {}
}
function OOStyleFactory() {
  return new OOStyle(/* ... */);
}
```

这是面向对象风格的写法（注意，只是风格，不是指只有这个是面向对象）

```
function funcStyle(name, password) {
  return {
    name,
    password,
    getName() {},
    changePassword() {},
  };
}
```

这个是函数风格的写法（注意，这只是风格，这同时也是面向对象）

这两种风格的逻辑是一样的，唯一的区别，只在于可读性

不要理解错，这里的可读性，还包括对于程序而言的可读性，即：自动生成文档，自动生成代码结构或者由产品设计直接导出代码框架等功能

但是函数风格牺牲了可读性，得到了灵活性这一点，也是值得考虑的

编程其实是个权衡过程，对于我来说，我愿意

- 在处理复杂结构时使用 面向对象 风格
- 在处理复杂逻辑时，使用 函数 风格各取所长，才是最佳方案！

## Redux

```
// redux reducer
function todoApp(state, action) {
  if (typeof state === "undefined") {
    return initialState;
  }

  // 这里暂不处理任何 action,
  // 仅返回传入的 state。
  return state;
}
```

这其实就是用函数风格实现的 面向对象 封装，没有这一步，你无法进行顶层设计！

用类的写法来转换一下 redux 的写法：

```
class MonistRedux {
  // initial, 想要不变性可以将 name,password 组合为 state
  name = "";
  password = "";
  // 惰性初始化（配合工厂）
  constructor() {
    this.name = "";
    this.password = "";
  }
  // action
  changeName() {}
}
```

### 只有函数的封装性才受副作用限制

注意这一点，React 程序员非常容易犯的错误，就是到了 class 里面还在想纯度的问题，恨不得将每个成员函数都变成纯函数

没必要以词害意，需要融汇贯通

同样，以上例子也说明，如果你的技术栈提供直接生成对象的方案 —— **你可以只用函数直接完成面向对象和函数式的设计**

```
function ImAClass() {  
  return {  
    // ...  
  };  
}
```

我就说这个是类！为什么不行？

他要成员变量有成员变量，要成员函数有成员函数，封装，多态，哪个特性没有？

什么？继承？这年头还有搞面向对象的提继承？组合优于继承是常识！

抛弃了继承，你需要 `this` 么？你不需要，本来你就不需要 `this`（除了装饰器等附加逻辑，但是函数本身就能够实现附加逻辑 —— 高阶函数）

同样，你也可以综合面向对象和函数式的特点，各取所长，对你的项目进行顶层构建和底层实现

这也是很方便的

## Hooks、Composition、ngModule

我们来看看上面的那个函数风格的类

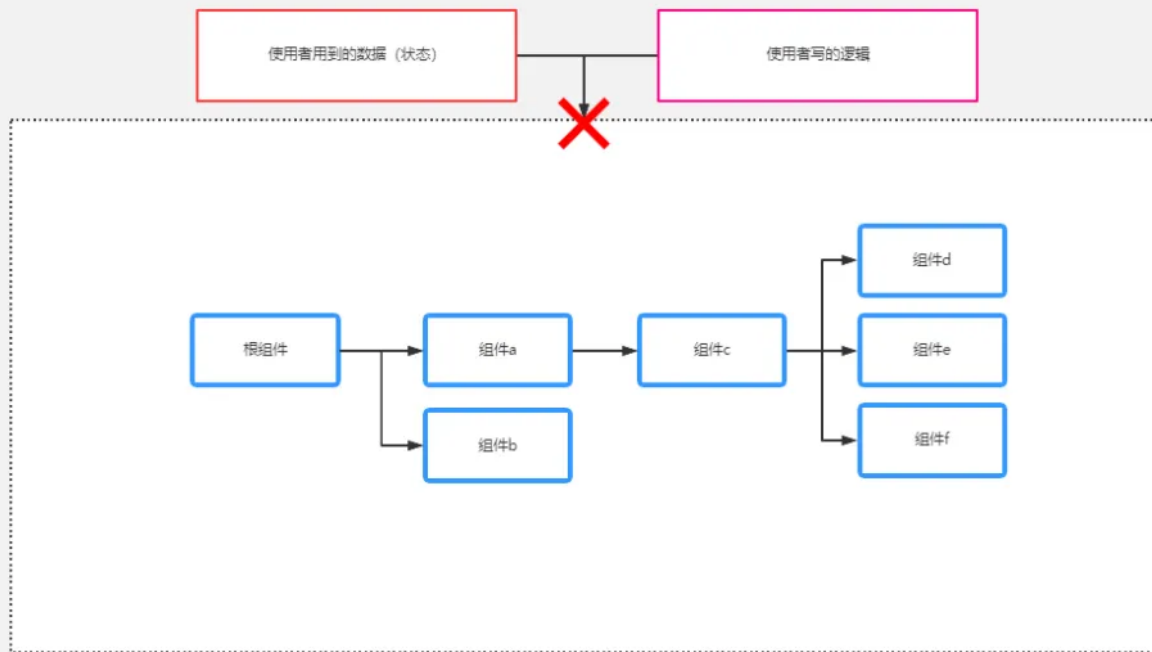
像不像什么东西？

```
function useThisClass() {  
  const [val1, setVal1] = useState(0);  
  const [val2, setVal2] = useState(0);  
  useEffect(() => {}, []);  
  const otherObject = useOtherClass();  
  return { val1, setVal1, val2, setVal2, otherObject };  
}
```

Hooks 恭喜各位，用得开心！

以 React 为例，老一代的 React 在组件结构上是管道，也就是单向数据流，但是对于我们这些使用者来说，我们写的逻辑，基本上是放养状态，根本没有接入 React 的体系，完全游离在函数式风格以外：



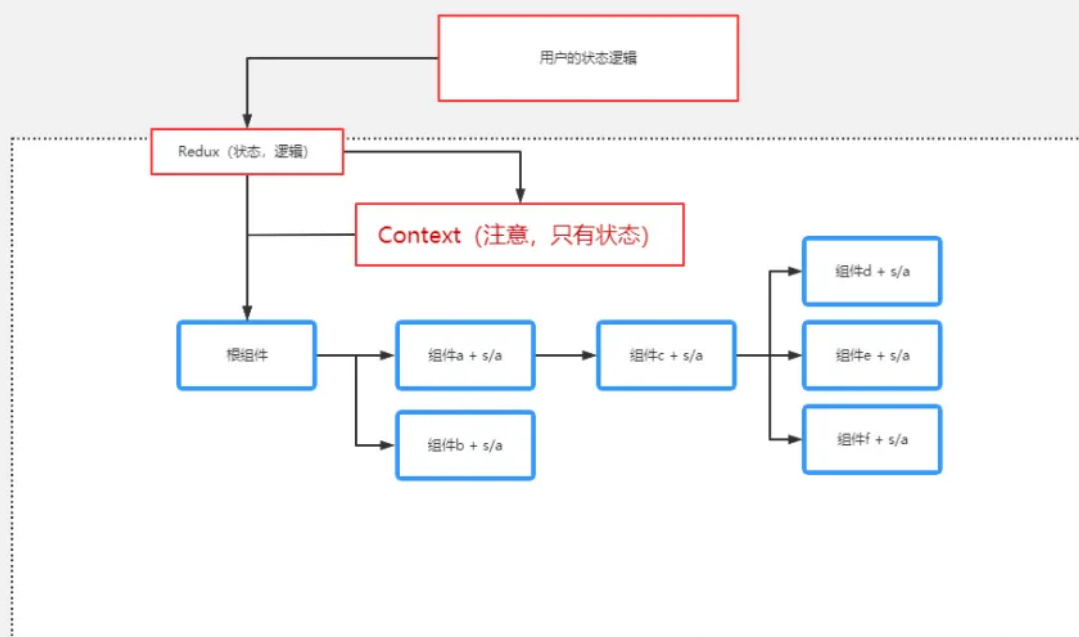


## 老一代的 React

换句话说，只有 React 写的代码才叫函数式风格，你写的顶多叫函数！

你没有办法把逻辑写在 React 的组件之外，注意，是完全没有办法！

好办，我逻辑全部写在顶层组件，那不就行了？



新一代的 React

(其中 s/a 指的是 state, action)

为什么要有 state, action?

为什么要在每个组件里用 s/a?

action 其实是用命令模式，将逻辑复写为状态，以便 Context 传递，为何?

因为生命周期在组件里，setState 在组件的 this 上

换句话说，框架没有提供给你，将用户代码附加于框架之上的能力!

这个能力，叫做 IOC 控制反转，即 框架将功能控制权移交到你的手上

不要把这个类 Redux 开发模式作为最自然的开发方式，否则你会非常痛苦!

## 只有集成度不高的系统，才需要中介模式，才需要MVC

之前的 React/Vue 集成度不高，没有 Redux 作为中介者 Controller，你无法将用户态代码在架构层级和 React/Vue 产生联系，并且这个层级天然应该用领域模块的思想方法来处理问题

因为**框架没有这个能力**，所以你才需要这些工具

所谓的状态管理，所谓的单一 Store，都是没有 IOC 的妥协之举，并且是在完全抛弃面向对象思想的基础上强行用函数式语言解释的后果，是一种畸形的技术产物，是框架未达到最终形态之前的临时方案，不要作为核心技术去学习，这些东西不是核心技术!

回头看看 React 那些暧昧的话语，有些值得玩味：

- Hook 使你在无需修改组件结构的情况下复用状态逻辑（注意！是状态逻辑，不是状态，是状态逻辑一起复用，不是状态复用）
- 我们推荐用 自定义 hooks 探索更多可能
- 提供渐进式策略，提供 useReducer 实现大对象操作（好的领域封装哪来的操作大对象？）

他决口不提 面向对象，领域驱动，和之前的设计失误，是因为他需要顾及影响和社区生态，但是使用者不要被这些欺骗!

当你有了 hooks, Composition, Service/Module 的时候, 你应该主动抛弃所有类似

- 状态管理
- 一定要写纯函数
- 副作用要一起处理
- 一定要保证不变形

这之类的所有言论, 因为在你手上, 不仅仅只有函数式一件武器

你还有面向对象和领域驱动

用领域驱动解决高层级问题, 用函数式解决低层级问题, 才是最佳开发范式

也就是说, 函数式和面向对象, 没有好坏, 他们只是两个关注点不同的思想方法而已

你要是被这种思想方法影响, 眼里只有对错 —— 实际上是被忽悠了

---

## 管道风格的函数式

unidirectional network 单项数据流

这是函数式语言基本特性, 将一个个符合封装要求的函数串联起来, 你就能得到统一输入输出

函数将沦为算式, 单测作用将消失, 理想无 bug 系统呼之欲出

它会形如以下结构:

```
func1(func2(), func3(func4(startParams)));
```

或者:

```
func1().func2().func3().func4();
```

看到这些形态, 大家会首先想到什么?

没错 `jsx` 就是第一种结构 (是的, **jsx 是正宗函数式**, 纯粹无副作用, 无需测试, 仅需输入输出校验调试)

也没错, `promise.then().then()` 就是第二种, 将**函数式**处理并发, 异步的优势发挥了出来

那第二种和第一种, 有什么区别呢?

区别就是 ——

## 调度

函数是运行时的结构，如果**没有利用模式匹配**，每次函数执行只有一个结果，那么整个串行函数管道的返回也只会会有一个结果

如果利用了呢？它将会向路牌一样，指示着逻辑倒流到特定的 `lambda` 函数中，形成分形结构

请注意，这个分形结构不只是空间上的，还有时间上的

这，就是调度！！

说的很悬奥，大家领会一下意思就可以了，如果想要了解更多，直接去成熟工具处了解，他们有更多详细的说明：

微软出品的 `ReactiveX[2]`（脱胎于 `linq`），就是这方面的集大成者，网络上有非常多的讲解，还有视频演示

## Hooks api

Hooks api 就是 React 的调度控制权

### **`useState` 整个单项数据流的调度发起者**

React 将它的调度控制权，拱手交到了你的手上，这就是 `hooks`，它的意义绝对不仅仅是“在函数式组件中修改状态”那么简单

`useState`，加上 `useReducer`（个人不推荐，但是与 `useState` 返回类型一致）

属于：响应式对象（注意，这里的对象是 `subject`，不是 `object`，有人能帮忙翻译一下么？）

**`dispatch` 是整个应用开始渲染的根源**

没错，`this.setState` 也有同样功能（但是仅仅是组件内）

# useEffect 整个单项数据流调度的指挥者

useEffect 是分形指示器

在 Rxjs 中被称作 操作函数 (operational function)，将你的某部分变更，衍射到另一处变更

在这个 api 中，大量模式匹配得以正常工作

# useMemo 整个单项数据流调度的控制者

最后，useMemo

它能够通过只判断响应值是否改变，而输出控制

当然，你可以用 if 语句在 useEffect 中判断是否改变来实现，但是 —— 模式匹配就是为了不写 if 啊～

单独提出 useMemo，是为了将 设计部分 和 运行时调度控制 部分分离，即静态和动态分离

调度永远是你真正开始写函数式代码，最应该考虑的东西，它是精华

纠结什么是什么并不重要，即便 `useMemo = useEffect + if + useState`，这些也不是你用这部分 api 的时候应该考虑的问题

最后

你明白这些，再加上 hooks 书写时的要求：

不要在循环，条件或嵌套函数中调用 Hook，确保总是在你的 React 函数的最顶层调用他们。遵守这条规则，你就能确保 Hook 在每一次渲染中都按照同样的顺序被调用。这让 React 能够在多次的useState和useEffect调用之间保持 hook 状态的正确。

你就能明白，为什么很多人说 React hooks 和 Rx 是同一个东西了

但是请注意，React 只是将它自己的调度控制权交给你，你若是自己再用 rx 等调度框架，且不说 requestAnimationFrame 的调度频道React占了，两个调度机制，弥合起来会非常麻烦，小心出现不可调式的大bug

函数式在处理业务逻辑上，有着非常恐怖的锋利程度，学好了百利而无一害

但是请注意，函数式作为对计算过程的一般抽象，只在组件服务层级以下发挥作用，用它处理通讯，算法，异步，并发都是上上之选

但是在软件架构层面，函数式没有任何优势，组件层级以上，永远用面向对象的思维审视，是个非常好的习惯～

## 组件明明就只是逻辑和状态（服务）调配的地方，它压根就不应该有逻辑

把逻辑放到服务里

- 坚持在组件中只包含与视图相关的逻辑。所有其它逻辑都应该放到服务中。
- 坚持把可复用的逻辑放到服务中，保持组件简单，聚焦于它们预期目的。
- 为何？当逻辑被放置到服务里，并以函数的形式暴露时，可以被多个组件重复使用。
- 为何？在单元测试时，服务里的逻辑更容易被隔离。当组件中调用逻辑时，也很容易被模拟。
- 为何？从组件移除依赖并隐藏实现细节。
- 为何？保持组件苗条、精简和聚焦。

## 类型问题

React DDD 下如何处理类型问题？

### 泛型约束 InjectionToken

```
/**
 * 泛型约束，对注入数据的类型推断支持
 *
 * @export
 * @template T
 * @param {...args: any} => T useFunc
 * @param {(T | undefined)} [initialData=undefined]
 * @returns
 */
export default function getServiceToken<T>(  
  useFunc: (...args: any) => T,  
  initialData: T | undefined = undefined  
) {  
  return createContext(initialData as T);  
}
```

这样，你的 `useContext` 就能有完整类型支持了

# 虚拟数据泛型约束

```
/**
 * 获得虚拟的服务数据
 *
 * @export
 * @template T
 * @param {...args: any} => T useFunc
 * @param {(T | undefined)} [initialData=undefined]
 * @returns
 */
export function getMockService<T>(
  useFunc: (...args: any) => T,
  initialData: T | undefined = undefined
) {
  return initialData as T;
}
```

类服务在功能上，其实和函数服务是一样的

函数返回对象，本身也是构造方式之一，属于 js 特色（返回数组的话，本质也是对象）

所以功能上来说，函数服务完全可以覆盖类服务，实现面向对象

但是，需求可不止有功能一说：

- 需要有更好自解释性
- 需要更好自封装性
- 需要更好的可读性（自动生成文档，自动分析）

类型自动化：

```
function getClassContext<T>(constructor: new (...args: any) => T) {
  return createContext((undefined as unknown) as T);
}
```

## 实现一个类

```
/** * some service * * @export * @class SomeService */export class
SomeService { static Context = getClassContext(SomeService); name:
string; setName: Dispatch<SetStateAction<string>>; constructor() {
// eslint-disable-next-line react-hooks/rules-of-hooks const [name,
setName] = useState(""); this.name = name; this.setName = setName;
}}
```

## 使用这个类

最后，自动分析类的结构：

可自动文档化，在配合其他工具实现框架外需求时，能够带给你方便的使用体验

但是注意，目前官方是禁止在 class 中使用 hooks 的需要禁用提示

同时，需要保证在 **constructor** 中使用 **hooks**

---

## FAQ

### **React DDD 下，class 风格组件能用么？**

最好不要用，因为 class 风格组件的逻辑无法提取，无法连接到统一的服务注入树中，因此会破坏应用的单一数据原则，破坏封装复用性

所以尽量不要使用，至于目前

getSnapshotBeforeUpdate, getDerivedStateFromError 和

componentDidCatch 的等价写法 Hooks 尚未加入，这其实并不是大问题，因为在管道风格中，错误优先表征为状态，比如 useRequest 中的 error

### **React DDD 是相比之前的类redux 分层是更简单还是更难？**

简单太多了，整个 React DDD 的体系只需要 useXxx 表示服务 Xxx 表示组件，只需要用几个 hooks api，通过注入提供上下文，不需要高阶组件，好的模式也不需要 render props，更不需要太过重视性能调优（别担心性能问题，除了高消耗运算惰性加载以外），你只是无法在组件层级处理错误而已，个人认为，错误还是在用户端处理吧



尤其是在 Typescript 下，你代码的几乎任何一处都有完整的类型提示，不需要你去附加类型声明或者指定 interface

但是，它会将业务问题提前暴露，在没有想好业务逻辑关系的情况下，请不要下手写代码，但是这也不是它的缺点，因为在逻辑错乱的情况下，分层直接会变得不可维护

## **React DDD 需要用到什么工具么？**

不需要，直接使用 React hooks 就好，没有其他工具需求

## **React DDD 性能会有问题么？**

不会，React DDD 的方案性能比 class 风格组件 + 类 redux 分层要强得多，而且你可以精细化控制组件的调度和响应式，下限比 redux 上限还要高，上限几乎可以摸到框架极限

## **React DDD 适合大体量项目么？**

是的，从最小体量项目，到超大体量项目，React DDD 都很适合，原因在于回归面向对象，承认面向对象对顶层设计有优势的同时，业务逻辑采用极限函数式开发

无论在架构上，还是业务逻辑实现上，都会将效率，可复用性，封装度，集成度，可调式，可测试性直接拉满，所以不用担心

## **React DDD 效率高么？**

它不是高不高的问题，它是直接可以将大部分业务效率直接提高十倍的大杀器，而且还有很多第三方库可以被直接使用，让第三方帮你处理逻辑，比如 ahooks, swr 等等

于此同时，它直接跟业界主流工程化模式对接，有领域模块的加持，多人协作将变得更加有效率，也能形成特别多的技术资产

## **Redux 之类的工具还有意义么？**

没有意义了，它只是解决框架没有 IOC 情况下，保持和框架相同的单向数据流，保持用户态代码的脱耦而已，由于状态分散不易测试，提供一个切面给你调试而已

这种方案相当于强制在前端封层，相当不合理，同时 typescript 支持还很差

在框架有 IOC 的情况下，用户代码的状态逻辑实际上形成了一个和组件结构统一的树，称之为逻辑树或者注入树，依赖树，很自然地与组件相统一，很自然地保证单向数据流和一致性

所以，Redux 之类的工具最好不要用，妄图在应用顶层一个服务解决问题的方法，都很傻

## 现有项目能直接改成 React DDD 么？

这是它最大的缺点，不能！

因为问题的根源出在框架上，IOC 应该是框架的大变更，个人认为 React 应该直接暴力更新，摒弃所有老旧写法，如同 15 年的 Angular 一样，虽然有阵痛，但是对提升社区的好处大于坏处，当然，这是没有考虑市场的想法

如果你想更纯粹使用 React DDD，最好还是采取重构的方案

## React DDD 下，应该怎么划分文件结构？

按照功能划分，你的功能有哪些包含关系，你的文件结构就是如何

你的功能在哪个范围需要提供限界上下文，哪里就进行服务注入

所以类似拆分 store，action，models 之类的文件夹，就不要有了，前端没有数据库，即便有，也没有到需要抽象 dao 的程度，即便抽象 dao，dao 本身也是不符合工程化和 DDD 的

## 所以微前端可以由 React DDD 实现么？

是的，有了限界上下文，分开开发，分开测试，分开部署都可以实现

但是一定要在相同框架内，个人认为前端采用不同框架开发是个伪需求

现如今 Angular，React，Vue，都有 IOC，写法都可以互通

你要讲模块剥离，直接构建的时候把模块文件夹 cp 到指定目录，覆盖掉占位的文件夹即可

不过注意，‘微应用’需要有模拟顶层 context 的可选服务，当然，这些东西不管你怎么实现都是需要的

至于说重构兼容老代码而采用 shadowDom 和 iframe 的，我只能说，作为终端开发，重构兼容老代码只是临时状态，他不能作为架构，更不能作为一个技术来推广

## React DDD 和 Angular 的架构好像，为什么？

因为 Angular 15 年首先实现了 IOC，组件中推荐不要写逻辑，也是 Angular 最早提出的方案

service（状态逻辑单元），module

（service+component+template+css+staticAssets...），领域模块封装也是 Angular 最早提出的，但是因为当时它走太快，社区没跟上，生命周期复用也麻烦，因为采用装饰器，组件还保留了一个壳，推进最佳实践的难度比较大

而 React 的 hooks 可以更加抽象，也更简单直接，直接就是两个函数，服务注入也是通过组件，也就是强制与组件保持一致

这时候再推动 DDD 就非常容易且水到渠成了

但是 Angular 的很多特性 React 还不支持，比如组件样式封装，多语言依赖到视图，服务摇树，动态组件摇树，异步服务（suspense，concurrent 还在试验阶段），还有真正解决性能问题的大杀器 platform-webworker（能够在应用层级支持浏览器高刷）

但是这些需求，在没有超大体量（世界级应用）下，用到的可能性很小，不妨碍 React 的普适性

而且 React 社区更活跃，管道风格函数是对社区的依赖是很强的，这方面 Angular 干写 rxjs 管道有些磨人

## React DDD 会是未来趋势么？

这点我觉得毫无疑问，因为 DDD 是整个软甲开发架构设计的趋势，而且这个趋势伴随着 微服务的普及已经不可逆转，只要前端承认自己是编程，这个趋势同样也逃不过去

前端还是单节点，但是未来会有端到端

这个东西现在的可用性易用性在 React 语境下已经相当高了，未来也只会更有用

其他的不用说，光效率的提升，就可以让开发者大呼过瘾了

只是 React/Vue 还有很多历史问题需要解决，等这些问题解决了，DDD 肯定会大跨步向前的

# 管道流难度会不会很高？

是的，作为极限函数式开发，在给你提供更好的类型支持，容易调试测试的支持后，首当其冲的，就是纯粹函数式的爆炸难度

正常模式下，你是需要先化简范畴运算式再写代码的，不过这明显老学究了，哈哈哈

但是，React hooks 有非常活跃的社区，你不需要自己实现封装很多逻辑，这部分可以直接求助于社区实现

需要你实现的管道功能很少

不像 Angular 写 rxjs，管道需要自己根据一百多个操作函数配置，脑力负担太大，并且操作函数都是抽象的，调度权限给到你之后，复杂度又加了个 3 次方

React 的管道复用第三方，大多都是直接面向业务的，比如 swr 和 ahooks，要直接很多所以在，真正需要你写的管道逻辑并不多，这一点值得庆幸

但是，管道风格也是未来趋势，可以说管道和领域，分别是函数式和面向对象推演到极致的结果，两者都是最佳范式，两者都得学习

你只需要学思想方法，而且这样的思想方法，放诸四海皆可，任何编程平台，除了特别纯粹的那种：无类型 lisp 和无 lambda java，基本上这些概念都是想通且能够交流的

管道也是存在于编程的方方面面，elasticSearch, mongo aggregation, node stream, graphQL, 等等等等...

## 参考资料

[1]前端备忘录 - 江湖术士: <https://www.zhihu.com/column/plightfield>

[2]ReactiveX: <https://mcxiaoke.gitbooks.io/rxdocs/content/Intro.html>

转自：前端备忘录 - 江湖术士

<https://hqwuzhaoyi.github.io/2021/01/14/74.HookDDD/>