我的 React 最佳实践

免责声明：以下充满个人观点，辩证学习

React 目前开发以**函数组件**为主，辅以 `hooks` 实现大部分的页面逻辑。目前数栈的 react 版本是 16.13.1，该版本是支持 `hooks` 的，故以下实践是 `hooks` 相关的最佳实践。

前置理解

首先，应当明确 React 所推崇的函数式编程以及 `f(data) = UI` 是什么？

函数式编程

函数式编程是什么？这里的函数并非 `JavaScript` 中的函数，或任何语言中的函数。此处的函数是数学概念中的函数。让我们来回忆一下数学中的函数是什么。

$$y = x^2 = f(x)$$

在数学概念中，函数即一种特殊的**映射**，如何理解**映射**？

以一元二次方程为例， *f(x)* 是一种映射关系，给定一个的 *x* ，则存在 *y* 与之对应。

我们将一元二次方程理解为一个黑盒，该黑盒存在一个输入，一个输出，在输入端我们给入一个值 *x = 2* 则输出端必然会给出一个 *y = 4* 。

`f(data)=UI`

在了解了数学中函数的概念后，将其概念套用到 React 中，我们就可以明白 *f(data)=UI* 到底指什么意思？

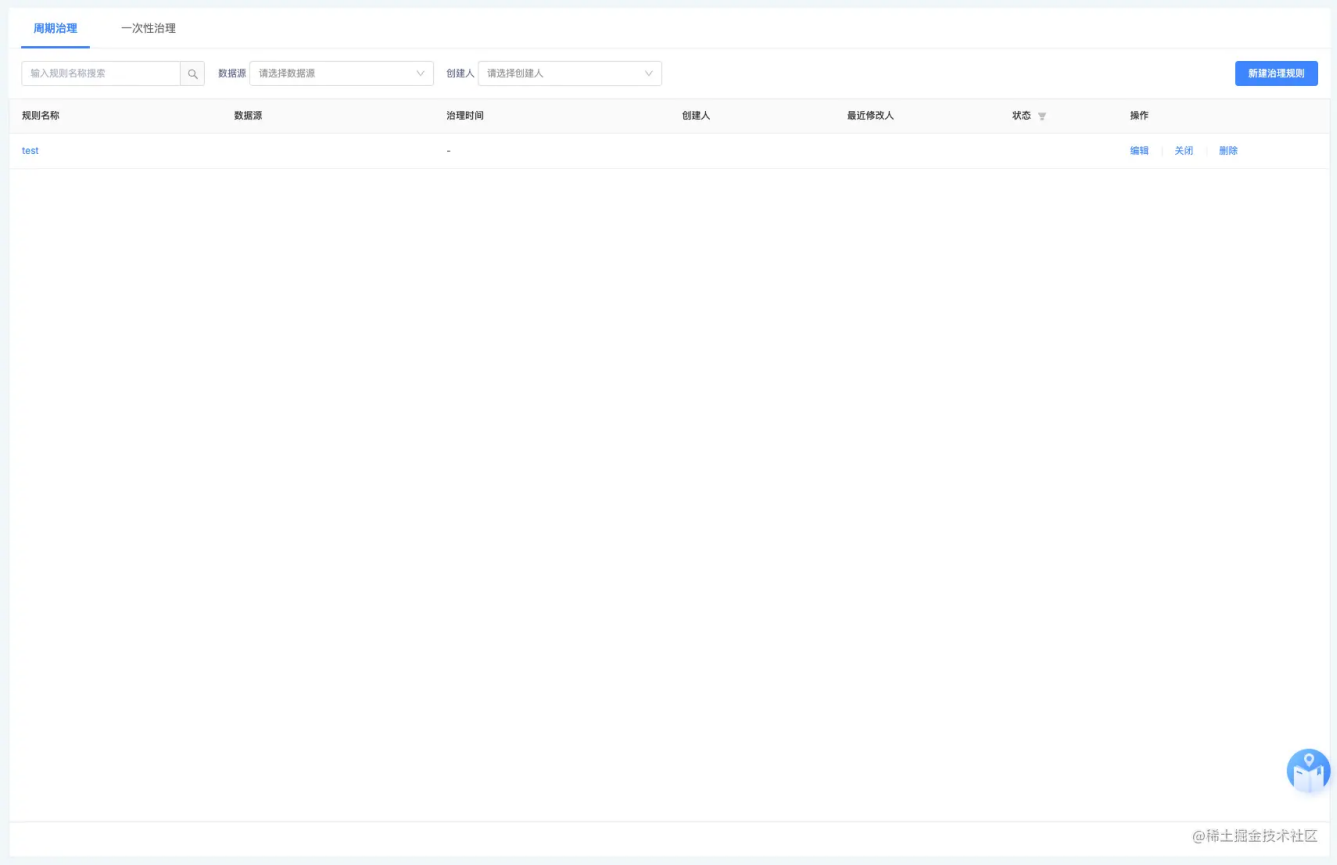
结论：个人理解，将当前组件内部的所有逻辑视为一个黑盒，该黑盒有且仅有一个输入，有且仅有一个输出，输入端为 `props` ，输出端则是当前组件的 `UI` ，不同的输入会决定不同的输出，就像把 *x = 1* 和 *x = 2* 给到所得到的结果是不一样的一样。而将这样的组件组合起来就是每一个页面，即 React 应用。

业务开发

目前绝大部分的后台管理系统，不仅仅是数栈，包括所有 ERP 系统，图书管理系统等等。其主体页面大致可以包括一下三类：

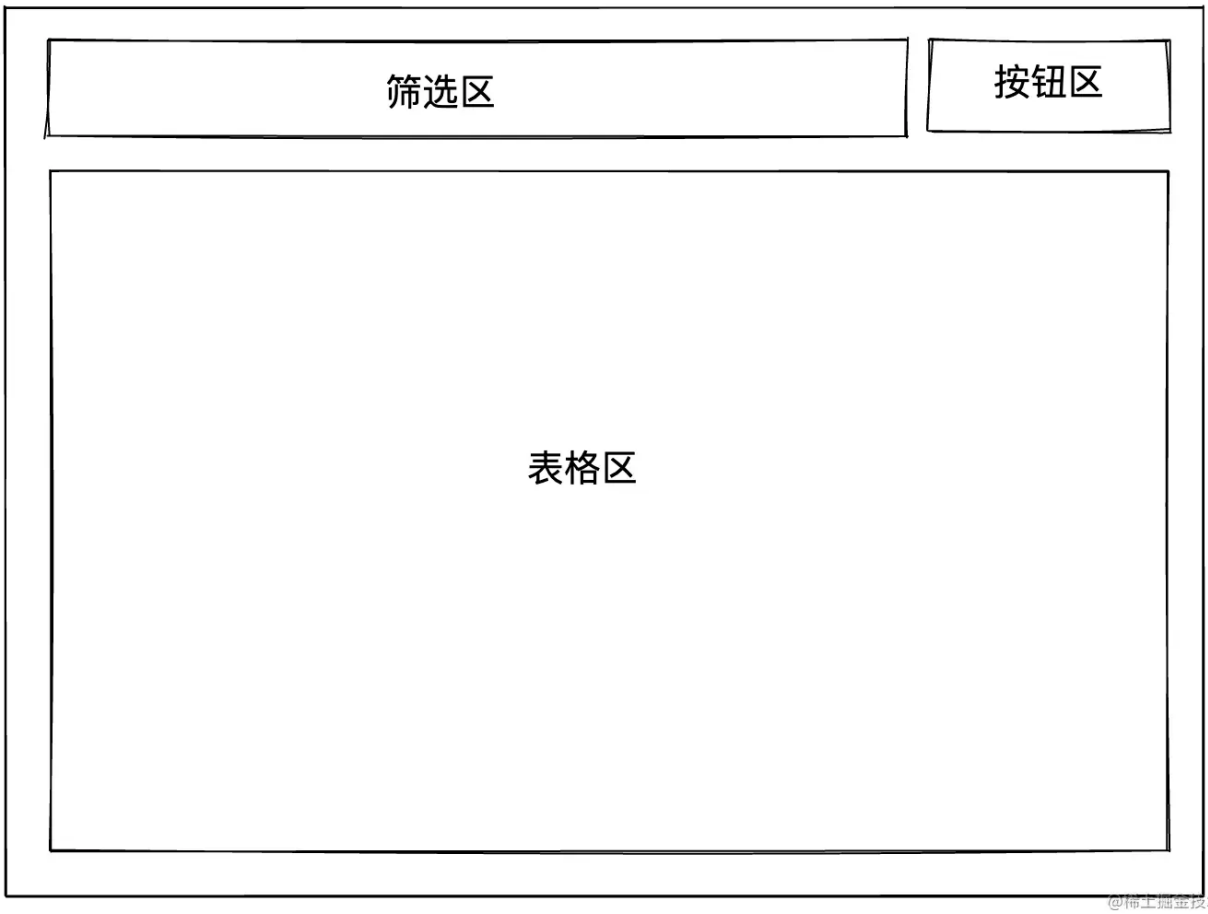
- 筛选条件（Filter）+ 表格（Table）
- 表单（Form）相关的新增，编辑功能
- 概览页面（Overview），包括一些图表和表格

第一类



以上是这次在资产中负责开发的文件治理规则页面，属于是第一类的典型页面，那这一类的页面应该如何开发才是最佳实践？

首先，我们将这一类页面抽象成如下结构



如此一来，我们不难实现如下的 dom 结构

```html 复制代码
<div className="container">
  <div className="header">
    <div className="filter">筛选区</div>
    <div className="buttonGroup">按钮区</div>
  </div>
  <div className="content">表格区</div>
</div>
```

接下来，我们需要补充各个区域的内容。

筛选区通常会有一些筛选项，这里我们有两个选择，如果比较复杂的筛选项，如存在 5 个以上或存在联动的交互，则选择通过 Form 来实现，而上图中这种比较简单的则可以直接实现。 这里我们不难观察到我们需要 3 个 value 值来实现这 3 个输入框或下拉框的受控模式，这里我们通过声明一个 filter 的变量，将这 3 个值做一个整合。

即

```jsx 复制代码
function (){
  const [filter, setFilter] = useState({
    a: undefined,
    b: undefined,
    c: undefined
  });

  return <><>
}
```

提问：为什么要 3 个值都放入到一个变量里?

答：1. 减少冗长的定义。 2. 为后续铺垫。

声明完 3 个值后，我们把组件填入

```jsx 复制代码
function (){
  const [filter, setFilter] = useState({
    a: undefined,
    b: undefined,
    c: undefined
  });

  return (
    <div className="container">
      <div className="header">
        <div className="filter">
          <Input value={filter.a} onChange={(e) => setFilter(f => ({...f, a: e.target.value}))} />
          <Input value={filter.b} onChange={(e) => setFilter(f => ({...f, b: e.target.value}))} />
          <Input value={filter.c} onChange={(e) => setFilter(f => ({...f, c: e.target.value}))} />
        </div>
        <div className="buttonGroup">按钮区</div>
      </div>
      <div className="content">表格区</div>
    </div>
  )
}
```

接着，我们实现表格区，有经验的同学可以知道，一个 Table 需要 `dataSource` 数据，`columns`， `pagination`， `total`，有时候还需要 `selectedRow` 和 `sorter` 和 `filter`。

我们先考虑 dataSource 和 columns。首先，我们先考虑数据获取，实现 `getDataSourceList`

tsx 复制代码

```tsx
interface ITableProps {}

function (){
  const [filter, setFilter] = useState({
    a: undefined,
    b: undefined,
    c: undefined
  });
  const [loading, setLoading] = useState(false);
  const [dataSource, setDataSource] = useState<ITableProps>([]);

  const getDataSourceList = () => {
    setLoading(true);
    Promise.then(() => {
      /// xxxx
    }).finally(() => {
      setLoading(false);
    })
  };

  return (
    <div className="container">
      <div className="header">
        <div className="filter">
          <Input value={filter.a} onChange={(e) => setFilter(f => ({...f, a: e.target.value}))} />
          <Input value={filter.b} onChange={(e) => setFilter(f => ({...f, b: e.target.value}))} />
          <Input value={filter.c} onChange={(e) => setFilter(f => ({...f, c: e.target.value}))} />
        </div>
        <div className="buttonGroup">按钮区</div>
      </div>
      <div className="content">表格区</div>
    </div>
  )
}
```

可以看到，我们新增了 loading 变量，用来优化交互的过程，新增了 `ITableProps` 类型，用来声明表格数据的类型，此时**应当和服务端的同学沟通，从接口文档中获取到相关的数据结构，并补全这一块的类型。**

假设我们此时已经完成了类型的补全，那接下来我们需要完善 `columns`

tsx 复制代码

```tsx
const columns: ColumnType<ITableProps>[] = [];
```

这里存在部分分歧，有部分人认为需要加上 `useMemo` 。 为什么不加 useMemo?

1. 因为 useMemo 主要是为了缓存计算量比较大的值，而此处并没有计算量，只是一个变量的声明而已
2. 如果重复声明 useMemo 的话，是否会导致 Table 的重复渲染。我认为**过早的性能优化不如不优化**，如果真的存在这样子的问题再进行优化也不急。
3. 如果这里加上 useMemo 在某些情况下会有比较多的 depths 需要加到依赖项里，个人感觉这不优雅

在完善 columns 后，我们继续刚才提到的 `Pagination` 和 `total` 字段。 这里我们需要声明两个变量

tsx 复制代码

```tsx
const [pagination, setPagination] = useState({current: 1, pageSize: 20});
const [total, setTotal] = useState(0);
```

思考：这里的 total 和 pagination 是否可以合并成一个变量?

答：可以，但是我不愿意，因为我们后面会有存在如下代码，会导致无限循环

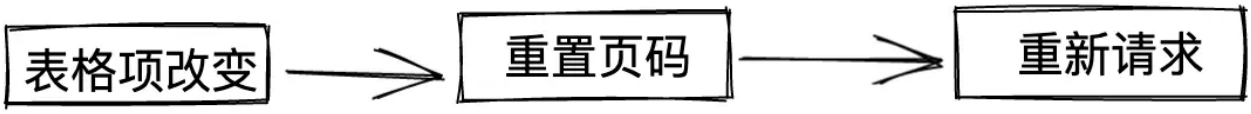
tsx 复制代码

```tsx
useEffect(() => {
  // getDataSourceList 中存在改变 total 的值，会导致无限循环
  // 如果要解决这个问题，则需要在 depths 中分别写 current 和 pageSize
  // 我不愿意
  getDataSourceList();
},[pagination]);
```

然后接下来我们要实现请求功能，不难总结出我们需要在以下几种情况下做请求：

- 页面初始化
- Pagination 改变进行请求
- 筛选条件改变进行请求
- Table 的 filter 或 sorter 发生改变进行请求
- 交互修改数据后进行请求（如删除，新增等）

除了最后这个暂时不做考虑，**第三**和**第四**项的请求我们可以再细分为如图所示



那么，就可以和前两项进行合并，总结如下：

- 页面初始化
- Pagination 改变

将上述思路转化为代码可得

```tsx 复制代码
function (){
  const [filter, setFilter] = useState<Record<string, any>>({
    a: undefined,
    b: undefined,
    c: undefined,
    d: undefined,
    sorter: undefined
  });
  const [pagination, setPagination] = useState({ current: 1, pageSize: 20 });

  const getDataSourceList = () => {};

  useEffect(() => {
    setPagination(p => ({ ...p, current: 1 }));
  }, [filter]);

  useEffect(() => {
    getDataSourceList();
  }, [pagination]);
}
```

到这里，我们已经实现了绝大部分主要功能，接下来我们简单实现一个 `selectedRows` 即可。

```tsx 复制代码
function (){
  const [selectedRows, setSelectedRows] = useState([]);

  return (
    <Table
      rowSelection={{
        selectedRowKeys,
        onChange: (selected) => setSelectedRows(selected)
      }}
    />
  )
}
```

问：为什么这里的 rowSelection 不提出来，放到 useMemo 里去?

至此，一个满足业务的一类业务相关的框架代码已经编写完成。 到这里之后，一类业务页面还有剩下什么东西需要开发?

第二类

二类页面的特点通常是新增和编辑复用同一个页面，需要 Form 表单。

除此之外，通常二类页面有通过 Drawer 或者 Modal 或者跳转路由的方式，但这不影响代码的书写。 不论是 Drawer 还是 Modal 还是路由的方式，我们都需要将表单内容抽离出一个新的组件。

首先，我们看一个比较普遍且较为简单的新增或编辑页面

可以观察到，这个表单通过 Steps 组件分割成了 3 个步骤

这里我们需要明确一个思想，即上面强调的 ，那么这里不论是新增还是编辑，其差异只存在于 data 中是否存在 id 值。 这里有两种做法，第一种做法是 3 个步骤只用一个 Form，第二种做法是 3 个步骤 3 个 Form。我个人比较喜欢第一种做法。理由如下：

- 因为 data 是一份的，如果用第二种做法，需要将一份 data 拆分成三份
- 倘若出现在第一步中填写某个值会影响第二步中的下拉项，如果是第二种做法，还需要将 Form 的值传给第二步的组件。第一步的做法可以直接通过 `form.getFieldValue('xxx')`
- 无它，就是图个简洁

比较复杂的表单通常都有联动情况，比如数据同步任务的表单，或者这里的数据源和表选择的交互。 这一类交互在 `antd@3` 中通常实现起来会比较的繁琐，在 `antd@4` 中善用 `dependencies` 和 `onValuesChanged` 可以很好地解决这一类问题。

tsx 复制代码

```tsx
{[TRINO, KINGBASEES8, SAPHANA1X].includes(dataSourceType) && (
  <FormItem
    name="schemaName"
    label="选择Schema"
    initialValue={schemaName}
    rules={[
      {
        required: true,
        message: '请选择Schema',
      },
    ]}
    >
    <Select
      showSearch
      style={{
        width: '100%',
      }}
      onSelect={this.onSchemaChange}
      onPopupScroll={this.handleSchemaScroll}
      onSearch={this.handleSchemaSearch}
      >
      {this.renderSchemaListOption(currentSchema)}
    </Select>
  </FormItem>
)}
```

如上代码所示， `schema` 的字段只有在所选择的数据源是 xxx 这几种情况下才会展示，如果我们按照上述代码的写法的话，需要在 `state` 中新增 `dataSourceType` 字段 那如果用 `dependencies` 的话，可以改成如下写法：

tsx 复制代码

```tsx
<FormItem noStyle dependencies={['sourceId']}>
  {({ getFieldValue }) => (
  isXXXXX(options.find(o => o.sourceId === getFieldValue('sourceId'))?.type) && (
    <FormItem
      name="schemaName"
      label="选择Schema"
      initialValue={schemaName}
      rules={[
        {
          required: true,
          message: '请选择Schema',
        },
      ]}
      >
      <Select
        showSearch
```

```
        style={{
          width: '100%',
        }}
        onSelect={this.onSchemaChange}
        onPopupScroll={this.handleSchemaScroll}
        onSearch={this.handleSchemaSearch}
        >
        {this.renderSchemaListOption(currentSchema)}
      </Select>
    </FormItem>
  )
)}
</FormItem>
```

更进一步，可以把 `find` 抽象一个 `getTypeBySourceId` 函数出来，即优化了可维护性，又减少了变量声明。

除此之外，还有下拉菜单的联动，如 A 的选择会引起 B 的下拉菜单获取，B 的下拉菜单可能又会引起 C 的下拉菜单改变，如此链路下去，会导致声明的 `handleChange` 函数又多又长

tsx 复制代码

```
function(){
  const handleAChanged = () => {};

  const handleBChanged = () => {};

  const handleCChanged = () => {};

  return (
    <Form>
      <FormItem name ="a">
        <Select onChange={handleAChanged} />
      </FormItem>
      <FormItem name ="b">
        <Select onChange={handleBChanged} />
      </FormItem>
      <FormItem name ="c">
        <Select onChange={handleCChanged} />
      </FormItem>
    </Form>
  )
}
```

那我们可以借助 `antd@4` 的 `onValuesChanged` 函数，来把所有相关组件的 onChange 做合并，即如下：

```tsx 复制代码
function(){
   const handleFormFieldChanged = (changed: Partial<IFormFieldProps>) => {
    if('a' in changed){
      // do something about a
      getOptionsForB();
      form.resetFields(['b', 'c']);
    }

    if('b' in changed){
      // do something about b
      getOptionsForC();
      form.resetFields(['c']);
    }

    if('c' in changed){
      // do something about c
      getOptionsForD();
    }
  }

  return (
    <Form onValuesChanged={handleFormFieldChanged}>
      <FormItem name ="a">
        <Select />
      </FormItem>
      <FormItem name ="b">
        <Select />
      </FormItem>
      <FormItem name ="c">
        <Select />
      </FormItem>
    </Form>
  )
}
```

同时，在这个函数里我们也可以顺便把 `reset` 的操作做了

到这里，我们大致完成了 Form 表单的架构思路。接下来，我们需要处理新增和编辑的区分。通常来说，新增是不需要赋**初始值**的，而编辑是需要赋**初始值**的。 这里需要注意的点在于，我们所理解的**初始值**并不是 Form 组件中 **initialValue** 的含义。（至少我认为不是） 我认为 Form 组件的生命周期应当分为如下部分：

- 初始化阶段（该阶段 Form 表单用 **initialValue**把表单 UI 渲染出来）
- 赋值阶段（该阶段用户通过 set 操作把**初始值**赋给表单的 state）
- 交互阶段

- 提交阶段

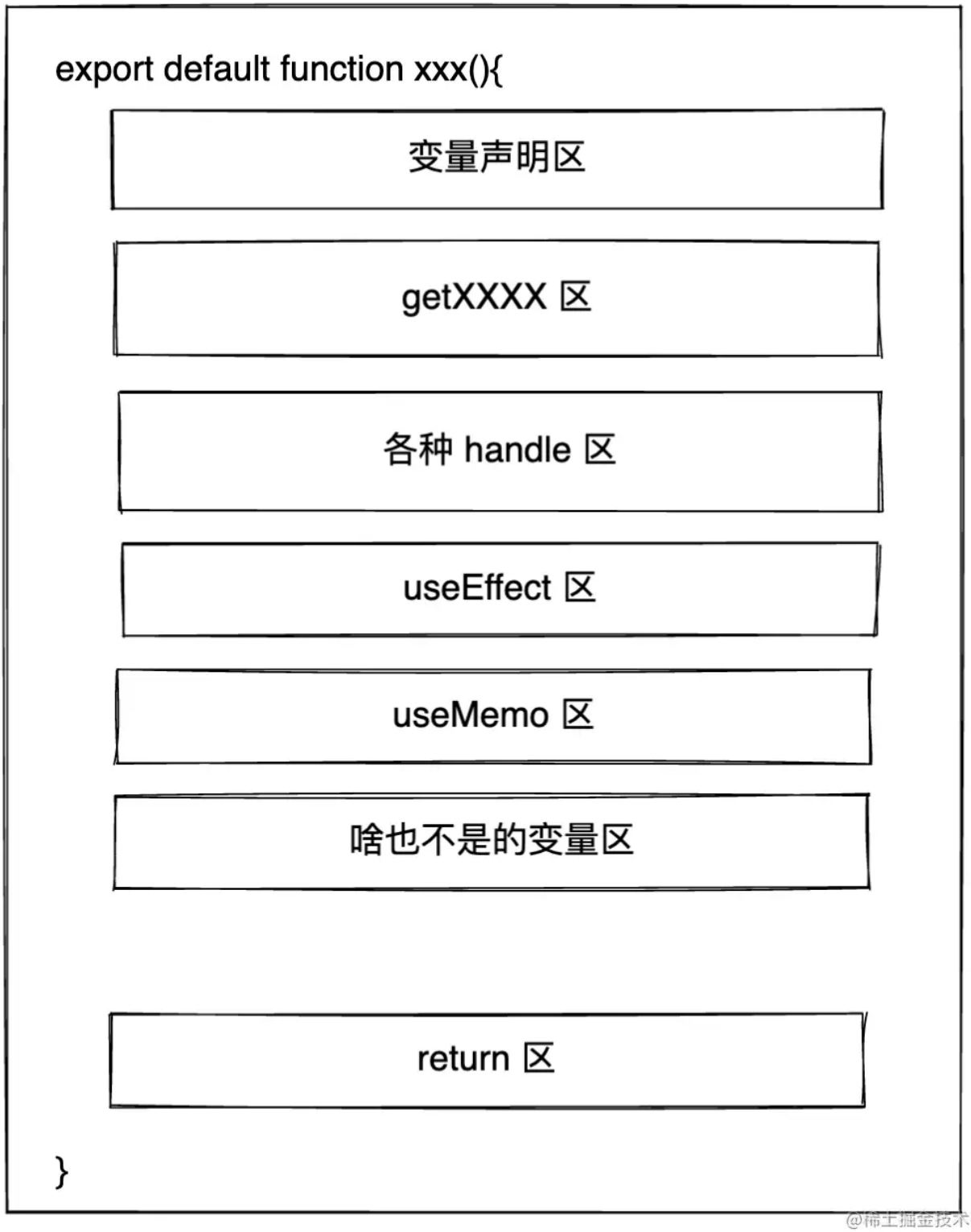
如上阶段中说明所示，我认为初始值的操作应当通过 set 操作完成，那代码实现起来应当如下所示：

tsx 复制代码

```tsx
function(){
  useEffect(() => {
    if(router.record.id){
      setLoading(true);
      api.getxxx({recordId: record.id}).then(res => {
        if(res.success){
          form.setFieldsValue({
            a: res.data.a,
            b: res.data.b
          })
        }
      }).finally(() => {
        setLoading(false);
      });
    }
  }, []);
}
```

把编辑的赋值操作全都放到 `useEffect` 中执行，统一了书写的地方，有利于后期的维护。

总结后，可以得出整个 Form 表单页面的概览大致如下图所示



相信各位同学到这里之后，面对一个表单的原型图，心里已经有一个大致的伪代码的实现了。

第三类

通常概览页面的要素是，统计数据、时间选择、图表。这一类页面由于通常是作为用户第一个打开的页面，所以需要额外注意的是 `loading` 状态的展示。

需要注意的是这里的 loading 会存在以下几种情况：

- 整个页面的 loading，这种情况通常是全局的一个时间选择器，然后同时获取统计数据和图表，需要借助 `Promise.all` 或 `Promise` .allSettled 实现。
- 分区域的 loading，如图表区有图表区的 loading，表格区有表格的 loading，统计数据区有统计数据区的 loading，这种时候需要把 loading 更加细分，为每一个区域增加 loading 变量，确保各个请求都有 loading 状态展示

其他没什么好说的，主要是 CSS 的要求会更高。大致的伪代码会如下：

tsx 复制代码

```tsx
function(){
  const [options, setOptions] = useState({...defaultOptions});
  const [loading, setLoading] = useState(false);
  const [timeRange, setTimeRange] = useState([moment(), moment()]);
  const [statistic, setStatistic] = useState({
    a: 0,
    b: 0,
  });

  const getStatistic = () => {
    return new Promise((resolve) => {
      setStatistic({a: 1000, b: 30});
      resolve();
    });
  };

  const getCharts = () => {
    return new Promise((resolve) => {
      options.xxx = xxxx;
      setOptions({...options});
    });
  };

  useEffect(() => {
    setLoading(true);
    Promise.all([getStatistic(), getCharts()]).finally(() => {
      setLoading(false);
    });
  }, [timeRange]);


  return (
    <>
      <DateRange value={timeRange} onChange={(val) => setTimeRange(val)} />
      <Statistic />
      <LineCharts />
```

```
    </>
  )
}
```

代码开发

通常来说，我个人的习惯是先实现需求，再进行代码优化和分割，模块的提取等。所以以下优化都是基于所有业务逻辑已经完成的情况下。

hooks

通常，在完成业务后，一个组件内部会包含大量的 hooks 相关的东西。通常优化手段如下：

- 减少 `useState` 的使用，将不影响渲染的数据放到 `useRef` 里去，甚至说常量可以放到函数外部。同时，如果是同一个种类的可以进行合并
- 减少 `useMemo` 的时候，普通的赋值或声明或简单的计算完全不需要引入 `useMemo` ，可以在复杂的计算时加上 `useMemo`
- 避免 `useCallback` 的使用，目前想到的 `useCallback` 的场景，只有 `addEventListener` 的时候，其余情况下大部分都用不到。
- `useEffect` 可以进行写多个的，所以有些时候不同的逻辑可以放到不同的 `useEffect` 里去
- `useRef` 可以大量持有， `useRef` 能 `cover` 的场景远大于 `ref`
- `useContext` 在简单场景下完全可以替代 `redux` ，但是有性能问题。所以复杂场景下，建议是配合 `use-context-selector` 使用，或者选择其他状态管理工具，如： `recoil`
- `useLayoutEffect` 和 `useEffect` 在绝大部分应用情况下没有差异，只需要直接使用 `useEffect` 即可。目前考虑前者的唯一不可替代性仅存在于「闪烁」场景。
- `useReducer` 一般来说用不到，其使用场景应该是当存在多个数据，而某一个参数的改变会引起其他数据同时改变，从而引起页面重渲染。

tsx 复制代码

```tsx
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}
```

```
function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

- `useImperativeHandler` 在通常情况下避免使用，使用场景应该只有两种。第一种是要做 ref 的转发，比如封装了一个组件，需要把组件内部的某一个 input 的 ref 转发给父组件。第二种是封装了一个组件，该组件内部实现逻辑是非 JSX 的，存在实例，则需要通过该 hooks 把相关实例转发给父组件。
- `useDebugValue` 不熟
- `useDeferredValue` 不熟
- `useTransition` 不熟
- `useId` 不熟
- `useSyncExternalStore` 不熟
- `useInsertionEffect` 不熟

Ant Design 相关

众所周知，React 库搭配 Ant Design 食用是后台管理系统的高效的原因之一。 Ant Design 相关实践如下：

- 巧用 `Space` 组件，个人感觉有点类似于简单场景的 `Grid` 组件
- 复杂表单经常使用 Steps 做步骤分割，个人更加推荐 `Steps+Form` 而不是 `Steps.item+Form`
- `AutoComplete` 个人感觉有很多问题，譬如数据回填高亮等问题，个人感觉不如直接 `Input`
- `Form` 组件比较复杂，且配合其他组件的用法比较多，如 `Form+Steps` `Form+Tabs` `Form+Modal` 等，需要注意 `inactiveDestroy + preserve={false}` 。另外需要注意 `requiredMark` 和 `required` 的区别，validator 和其他 rules 的区别， `setFieldsValue` 和 `setFields` 的区别
- `Input` 等输入控件需要注意 `placeholder` 的值，Select 需要额外主要 `allowSearch`
- `Select` 个人习惯直接通过 `options` 赋值，而不是 `children` （原因：相比 jsx 会有更加好的渲染性能）
- `select` 可以通过 `dropdownRender` 自定义下拉内容

- `Tree` 组件面向的场景比较复杂的情况下，个人觉得 antd 的 tree 组件不好用，偏向于自己手写
- 所有的 `popup` 相关组件都可以设置 `getPopupContainer`，该属性用来修改组件渲染位置，如果遇到弹出层没有随滚动条滚动可以设置，但是设置完可能需要考虑 `overflow:hidden` 的问题
- `Table` 组件的 rowKey 属性很重要，必加。
- 需要注意 `Table + Modal` 的写法，不要把 `Modal` 写到操作列里面去
- `Modal` 和 `Drawer` 组件不推荐用 `visible && <Modal />` 的写法，会导致动画丢失。建议在写 `Modal` 或者 `Drawer` 的时候，把「空状态」考虑进去。同时可以配合 `Modal` 的 `destroyOnClose` 属性可以让每次 Modal 打开内容都是新内容。（PS：Form 除外）
- `Spin` 可以多，但不能少

其他

- 爱惜你的 `div`，不要动不动就搞个 `div`，过多的层级结构会影响加载速度的。量变引起质变
- 尽量避免 `ref` 的使用，在通常情况下如果考虑用 `ref` 解决问题的话，那可能代表了你的思路不对或代码设计不对。
- 有时候，可以用 `IIFE`。做到不脱离当前的上下文，又实现了相关逻辑的提取。比函数中定义函数更好一些。

总结

以上的相关实践，是本人在日积月累中总结和摸索出来的。 如有雷同，说明你和我有一样的感受。