开发红宝书-前端-v1

React-Native开发规范

标签（空格分隔）： React-Native JavaScript

react-native发布最新版本：最新版本为0.64.2。

本次更新：

1-：第三方node_module管理

2-：适配，开发技巧

一、编程规约

(一) 命名规约

1 【强制】 代码中命名均不能以下划线或美元符号开始，也不能以下划线或美元符号结束；

```
反例：_name / $Object / name_ / name$ / Object$
```

2 【强制】 代码中命名严禁使用拼音与英文混合的方式，更不允许直接使用中文的方式； 说明:正确的英文拼写和语法可以让阅读者易于理解，避免歧义。注意，即使纯拼音命名方式 也要避免采用；

```
反例：DaZhePromotion [打折] / getPingfenByName() [评分] / int 某变量 =
```

3 【强制】类名使用 UpperCamelCase 风格，必须遵从驼峰形式，第一个字母必须大写；

```
LoginPage MsgPage
```

4 【强制】方法名、参数名、成员变量、局部变量都统一使用 lowerCamelCase风格，必须遵从驼峰形式，第一个字母必须小写；

```
localValue / getHttpMessage() / inputUserId
```

5 【强制】常量命名全部大写，单词间用下划线隔开，力求语义表达

完整清楚，不要嫌名字长；

`示例：MAX_STOCK_COUNT` `反例：ORGREFRESH org_fresh`

6 【强制】使用抽象单词命名类名或者变量时，需加以修饰；

userMsg 等价于 userMessaage, userPic 等价于 userPicture

7 【强制】包名统一使用小写，点分隔符之间有且仅有一个自然语义的英语单词。包名统一使用单数形式，但是类名如果有复数含义，类名可以使用复数形式；

应用工具类包名为com.yepu.open.util 类名为UrlUtils

8 【强制】文件夹命名统一小写； 组件，或者类名,首字母全部大写，遵守驼峰命名法；

`示例： image 存放图片 app APP一些component art component 一些Art组件`

(二) 常量定义

1 【强制】不允许出现任何魔法值(即未经定义的常量)直接出现在代码中；

如：所有的常量定义在一个文件夹中！！！！

2 【推荐】不要使用一个常量类维护所有常量，应该按常量功能进行归类，分开维护。

如:缓存相关的常量放在类:CacheConsts下;

系统配置相关的常量放在类:ConfigConsts下；

说明:大而全的常量类，非得使用查找功能才能定位到修改的常量，不利于理解和维护；

(三) 格式规约

【强制】大括号的使用约定。如果是大括号内为空，则简洁地写成{}即可，不需要换行;

如果是非空代码块则:

1) 左大括号前不换行；

2) 左大括号后换行；

3) 右大括号前换行；

4) 右大括号后还有else等代码则不换行‘；’表示终止右大括号后必须换行。

```
if(){
 //todo
}else{
 //todo
}
```

【强制】 左括号和后一个字符之间不出现空格;同样，右括号和前一个字符之间也不出现空格；

【强制】if/for/while/switch/do 等保留字与左右括号之间都必须加空格；

【强制】任何运算符左右必须加一个空格；说明:运算符包括赋值运算符=、逻辑运算符&&、加减乘除符号、三目运行符等；

【强制】缩进采用 4 个空格，禁止使用 tab 字符；

【强制】单行字符数限制不超过120个，超出需要换行，换行时遵循如下原则:

5) 第二行相对第一行缩进4个空格，从第三行开始，不再继续缩进，参考示例；

6) 运算符与下文一起换行；

7) 方法调用的点符号与下文一起换行；

8) 在多个参数超长，逗号后进行换行；

```
const path = Path()
        .moveTo(0, -radius/2)
        .arc(0, radius, 1)
        .arc(0, -radius, 1)
        .close();
```

【强制】方法参数在定义和传入时，多个参数逗号后边必须加空格；

```
onMsgByCallAndMsg=(msg, title, type)⇒{
    this.setState({
        callMsgAndMsg:msg
    })
}
```

【推荐】方法体内的执行语句组、变量的定义语句组、不同的业务逻辑之间或者不同的语义之间插入一个空行。相同业务逻辑和语义之间不需要插入空行。

说明:没有必要插入多行空格进行隔开。

【强制】ide使用vscode, 安装eslint插件。

(四) package.json

【强制】在使用npm或者yarn获取资源时，必须在命令末尾添加--save；

说明：使用此命令会把使用的第三方相关信息写入到package.json，这样，其他成员在下载或者更新代码后使用npm i(或npm install)，就可以下载最新的npm，若不加 —save ，执行npm i的时候不会下载，其他成员运行项目后在运行可能会报错，此时需要分析查看报错信息进行重新的npm install XX；

【推荐】使用git或者svn进行代码版本管理时，尽量不上传node_module文件（进行忽略）；

说明：使用package.json进行包管理，下载或更新代码后，只需要执行npm i；当有修改npm包，建议进行版本管理，上传到私有的github仓库。使用私有地址时，使用如下命令：

npm install "私有仓库地址" --save

【强制】使用第三方或拉取新仓库时，第一步使用npm i或者npm install；

说明：

1-：检查版本是否存在冲突

2-：更新node_modules

【推荐】在使用npm或者yarn获取资源时，推荐不在命令后添加 -g；

说明，此命令可以让此资源包在根目录进行获取(全局安装)，不利于资源管理；

【强制】每个项目必须配置一个README.md文件，内容包括测试，正式环境等相关配置文件以及注意事项；

如：

项目：

1-：下载本项目后，请先执行npm i

2-：注意有三个仓库为私有地址

3-：测试账号为：hello 密码：123456

访问：

1-：../../common/Util/Url 为网络配置相关

2-：在打包时，需修改原生安卓目录下：app/common/util/config下的地址，详情请查看此文件

【推荐】安装npm包是，推荐来标记版本号；

说明：和^的作用和区别：会匹配最近的小版本依赖包，比如1.2.3会匹配所有1.2.x版本，但是不包括1.3.0

^会匹配最新的大版本依赖包，比如^1.2.3会匹配所有1.x.x的包，包括1.3.0，但是不包括2.0.0。那么该如何选择呢？当然你可以指定特定的版本号，直接写1.2.3，前面什么前缀都没有，这样固然没问题，但是如果依赖包发布新版本修复了一些小bug，那么需要手动修改package.json文件；和^则可以解决这个问题。但是需要注意^版本更新可能比较大，会造成项目代码错误，旧版本可能和新版本存在部分代码不兼容。所以推荐使用来标记版本号，这样可以保证项目不会出现大的问题，也能保证包中的小bug可以得到修复。

(五) 控制语句

1 【强制】在一个 switch 块内，每个case要么通过 break/return 等来终止，要么注释说明程序将继续执行到哪一个 case 为止;

在一个 switch 块内，都必须包含一个default 语句并且 放在最后，即使它什么代码也没有。

2 【强制】在 if/else/for/while/do 语句中必须使用大括号，即使只有一行代码，避免使用 下面的形式:

`反例: if (condition) statements;`

3 【推荐】推荐尽量少用 else， if-else 的方式可以改写成:

```
if(condition){
      //todo
   return obj;
}
```

// 接着写 else 的业务逻辑代码;

说明:如果非得使用

```
if()
...
else if(

)...else...
```

方式表达逻辑， 【强制】请勿超过3层，

超过请使用状态设计模式。

4 【推荐】使用三目运算，替换if else结构，精简代码

```
正例:
let msg = account > 10 ? "true" : "false";
```

```
反例:
let account=5;
if(account>10){
  console.log("true");
}else {
  console.log("false");
}
```

5 【推荐】除常用方法(如 getXxx/isXxx)等外，不要在条件判断中执行其它复杂的语句，将复 杂逻辑判断的结果赋值给一个有意义的布尔变量名，以提高可读性。

说明:很多 if 语句内的逻辑相当复杂，阅读者需要分析条件表达式的最终结果，才能明确什么 样的条件执行什么样的语句，那么，如果阅读者分析逻辑表达式错误呢?

//伪代码如下

```
boolean existed = (file.open(fileName, "w") ≠
null)&& (...) || (...);
if (existed) {
      ...
}
```

6 【推荐】如果if语句中 && 过多或者 || 过多，尝试去使用数组的方法去操作，提高代码可读性

```
正例:
if ([1,2,3].includes(a))
if (['a','b','c'].some(k⇒obj[key]<1)) {}

反例:
if (a ═══ 1 || a═══ 2 || a═══3 ) {}
let obj = {
    a:1,b:1,c:1
}
if(obj.a<1 || obj.b<1 || obj.c<1){}
```

(六) 注释规约

【强制】类、类属性、类方法的注释必须使用 Javadoc 规范，使用/*内容/格式，不得使用 //xxx 方式；

说明:在 IDE 编辑窗口中，Javadoc 方式会提示相关注释，生成 Javadoc 可以正确输出相应注释;在 IDE 中，工程调用方法时，不进入方法即可悬浮提示方法、参数、返回值的意义，提高阅读效率。

【强制】所有的类都必须添加创建者信息，以及类的说明，可以使用

模版文件进行配置；

【强制】方法内部单行注释，在被注释语句上方另起一行，使用//注释；

方法内部多行注释使用/* */注释，注意与代码对齐。

【强制】所有的常量类型字段必须要有注释，说明每个值的用途；

【推荐】注释掉的代码尽量要配合说明，而不是简单的注释掉。

说明:代码被注释掉有两种可能性:

1)后续会恢复此段代码逻辑。

2)永久不用。前者如果没有备注信息，难以知晓注释动机。后者建议直接删掉(代码仓库保存了历史代码)。

【推荐】对于注释的要求:

第一、能够准确反应设计思想和代码逻辑;

第二、能够描述业务含义，使别的程序员能够迅速了解到代码背后的信息。完全没有注释的大段代码对于阅读者形同 天书，注释是给自己看的，即使隔很长时间，也能清晰理解当时的思路;注释也是给继任者看的，使其能够快速接替自己的工作。

【推荐】好的命名、代码结构是自解释的，注释力求精简准确、表达到位。避免出现注释的一个极端:过多过滥的注释，代码的逻辑一旦修改，修改注释是相当大的负担。

【推荐】特殊注释标记，请注明标记人与标记时间。注意及时处理这些标记，通过标记扫描，经常清理此类标记。

1) 待办事宜(TODO):(标记人，标记时间，[预计处理时间]) 表示需要实现，但目前还未实现的功能。

2) 错误，不能工作:(标记人，标记时间，[预计处理时间])

在注释中用 FIXME标记某代码是错误的，而且不能工作，需要及时纠正的情况。

(七) 日志管理

【强制】 代码中过多使用console.log()会消耗性能，推荐去除不必要的日志输入代码；

```
1  配置babel插件:
2
```

```
module.exports = {
    presets: ['module:metro-react-native-babel-preset'],
    env: {
        production: {
            // For React Native
            plugins: [
                // "babel-plugin-root-import"
                'transform-remove-console', // 打包自动移除所有console
            ],
        },
    },
}
```

(八) 目录结构规范

```
以下目录结构示例中只展示js与静态资源，不包含原生代码：
```

```
├── index.js
├── App.js
└── src
    ├── component                可复用的组件（非完整页面）
    ├── page                     完整页面
    ├── config                       配置项（常量、接口地址、路由、多语言化等预置数据）
    ├── util                   工具类（非UI组件）
    ├── style                   全局样式
    └── image                   图片
在component和page目录中，可能还有一些内聚度较高的模块再建目录
```

二、页面编写规范

(一) state,props

1 【强制】 代码中初始化state因在constructor(props)函数中，而且尽量对每个变量进行注释；

```
正例：

    state={
        msg:'第三种初始化state方式'
      };
反例：

     constructor(props){
     super(props);
     this.state={
       msg:'第一种初始化state方式'
     }
    };

    constructor(props){
        super(props);
        Object.assign(this.state,{
            msg:'第二种初始化state方式'
        })
     };
```

2 【强制】 **代码中使用setState时，因注意异步可能**导致的问题，尽量使用回调函数；

```
    this.setState({
        //todo
     },()⇒{
        //执行setState后执行此函数
     })
```

3 【强制】 代码中用于页面展示处理UI的组件，命名以Page结尾，自定义组件命名中必须包含Component；

例子：

```
LoginPage           登录页
BtuuonComponent     按钮组件
```

4 【强制】代码中创建数组或对象使用以下方式；

```
    const user={
      name:'time',
      sex:'男',
      age:25,
    };

    const itemArray=['0','1','2',3,
{name:'25',age:'男'}];
```

5 【强制】代码中函数绑定this，强制使用箭头函数；

注：除组件原有方法，其他自定义函数命名时，需使用箭头函数；

```
//系统组件生命周期方法
constructor(props){
    super(props);
};
//自定义方法
goMainPage=()⇒{

};
```

6 【强制】代码中一些网络数据初始化，配置信息，推荐在此生命周期进行初始化；

componentDidMount

7 【强制】代码中使用定时器或者DeviceEventEmitter，必须在组件卸载进行销毁或者清除；

```
componentDidMount() {
    //注意addListener的key和emit的key保持一致
    this.msgListener =
DeviceEventEmitter.addListener('Msg',(listenerMsg)
⇒ {
        this.setState({
            listenerMsg:listenerMsg,
        })
```

```
    });
}

goMainPage=()⇒{
    this.timer = setTimeout(
        () ⇒ { console.log('把一个定时器的引用挂在this上'); },
        500
    );
};


componentWillUnmount() {
    //此生命周期内，去掉监听和定时器
    this.msgListener?.remove?.();
    // 如果存在this.timer，则使用clearTimeout清空。
    // 如果你使用多个timer，那么用多个变量，或者用个数组来保存引用，然后逐个clear
    this.timer && clearTimeout(this.timer);
}
```

8 【强制】使用本地图片资源时，需设置宽高并进行适当适配；

imgHeight=screenHeight, imgWidth= screenWidth

9 【强制】在React-Native中使用本地图片资源时，当不指定特殊尺寸图片时，需引入不同尺寸XX.png,XX2@.png,XX3@.png图片，并在代码引用中，使用如下方式：

```
<Image style={{flex: 1, height: screenHeight, width: screenWidth}}
       source={require('../XX.png')}>
```

10 【推荐】 当使用多个state或者props值时，推荐使用以下方式；

```
    const {size, dotRadius, color} = this.props;
    const { maxNumber,minNumber,}=this.state;
```

11 【强制】多使用可选链操作符，避免数据异常时的页面crash，轻质配置eslint规则，不加报错

```
正例:
this.props.route?.key
this.props.route.params?.type

反例: this.props&& this.props.route &&
this.props.route.key   this.props.route.params.type
```

(二) 样式

【强制】 当组件使用样式属性达到三个或者三个以上时，必须使用StyleSheet来创建样式属性并进行引用；

```
const styles = StyleSheet.create({
     container: {
         flex: 1,
         justifyContent: 'center',
         alignItems: 'center',
         backgroundColor: '#F5FCFF',
         marginTop:10,
     },
 });
```

【推荐】 当使用单一属性，或者全局样式属性时，推荐使用公共样式类，可以使用global进行属性样式的全局设定；

```
//StyleCommon.js
 module.exports={
     topColor:{
         backgroundColor: '#3A3D42',
     },
     mainView:{
         backgroundColor: '#12141B',
     },
 }
```

【强制】当使用Text组件时，必须给Text设置color属性，不然某品牌手机会不显示

```
正例: const styles = StyleSheet.create({
    container: {
       fontSize:12,
        color:'#E444444'
    },
}

反例: <Text style={styles.text}></Text>
  const styles = StyleSheet.create({
    container: {
       fontSize:12
    },
});
```

【推荐】谨慎使用overflow:hidden属性，如果布局嵌套过深，在布局渲染成android原生布局时部分手机会出现页面布局错乱，非必要不要使用，寻求替代方案

(三) var,let,const

【强制】对所有变量，对象的引用，使用const,不要使用var;

【强制】如果一定需要引用可变动的变量，对象，建议使用let代替var;

(四) 其他

【强制】对组件引用，变量引用，需遵从以下方式;

```
import React, {Component} from 'react';
import{
          View,
          Text,
          TouchableHighlight,
          Image,
          StyleSheet,
          InteractionManager,
} from 'react-native';
//from react,react-native优先;
//from npm库其次;
import { connect } from 'react-redux';

```

```
14  //from 项目内组件其次;
15  import LoadingAndTime from
    '../component/LoadingAndTime';
16  import { performLoginAction } from
    '../action/LoginAction'
17  import {encode} from '../common/Base64';
18
19  //变量初始化，常量初始化 最后;
20  let screenWidth = Dimensions.get('window').width;
21  let screenHeight =
    Dimensions.get('window').height;
22  let typeCode = Platform.OS == 'android' ?
    'android-phone' : 'ios-phone'
23  let selectColor=Platform.OS=='android'?
    null:'white'
```

【推荐】对于项目中已经封装好的，能直接引用的尽量直接阴影，这样逻辑变动时，仅需要更改一个文件，减少工作量

`正例:import { isAndroid,SCREEN_WIDTH } from react-native-noah-base`

`反例: Platform.OS=='android Dimensions.get('window').width`

【推荐】 对于一个页面，拆分成组件化的同时，如果组件很多的话，建议使用一个入口文件，将所有文件导出，这样引入组件的时候就可以将所有页面相关的组件放入一个目录里。

```
1  正例:
2    import { header, item , content } from
    ./component
3
4   反例:
5    import {} from ./component/header
6    import {} from ./component/item
7    import {} from ./component/content
```

【强制】一个文件一个组件(无状态组件除外)，文件名称和组件名称保持一致，每个文件业务逻辑清晰，避免多个无相关组件放到一个地方，多个组件可复用逻辑尝试使用自定义组件封装 比如useCustom

```
反例: 一个文件放了多个组件，组件逻辑复杂，一个文件放入几千行代码

正例:
   export class DetailPage extends React.PureComponent{}
   function line20(){
     return (
        <View/>
     )
   }
```

【推荐】对组件引用，变量初始化等，在整个页面或组件内未使用，因去除相关代码;

【推荐】某些全局变量请不要使用global，需新建文件进行导出引用；

```
NetUtil.get(global.url + “”)
```

【推荐】render()函数代码过长时，请适当进行拆分，拆分为‘‘页面内组件‘‘，提高可读性。render()函数代码行请勿超过五十行，超过之后，请自行进行拆分；

【推荐】React-Native的组件细粒度越小，越有利于性能和交互

【推荐】单一责任原则, 确保每个功能都完整完成一项功能，比如更细粒度的组件拆分，同时也更利于测试。

【推荐】推荐启用 eslint-plugin-react-hooks 中的 exhaustive-deps 规则。此规则会在添加错误依赖时发出警告并给出修复建议

【推荐】不要把props作为初始值给state，这样会关闭props的更新通知。

```
解决最好的方案是 使用 useMemo 代替 useState
function Button({ text }) {
  const formattedText = useMemo(() => slowlyFormatText(text), [text])

  return <button>{formattedText}</button>
}

```

【推荐】避免使用多个布尔值来表示组件状态。

当编写一个组件并多次迭代后，很容易出现这样一种情况，即内部有多个布尔值来表示 该组件处于哪种状态

```
反例
function Component() {
  const [isLoading, setIsLoading] = useState(false)
  const [isFinished, setIsFinished] =
useState(false)
  const [hasError, setHasError] = useState(false)

  const fetchSomething = () => {
    setIsLoading(true)

    fetch(url)
      .then(() => {
        setIsLoading(false)
        setIsFinished(true)
      })
      .catch(() => {
        setHasError(true)
      })
  }

  if (isLoading) return <Loader />
  if (hasError) return <Error />
  if (isFinished) return <Success />

  return <button onClick={fetchSomething} />
}

正例:

function Component() {
  const [state, setState] = useState('idle')

  const fetchSomething = () => {
    setState('loading')

    fetch(url)
      .then(() => {
```

```
      setState('finished')
    })
    .catch(() => {
      setState('error')
    })
  }

  if (state === 'loading') return <Loader />
  if (state === 'error') return <Error />
  if (state === 'finished') return <Success />

  return <button onClick={fetchSomething} />
}


  //typescript   枚举
  const [state, setState] = useState<'idle' | 'loading' | 'error' | 'finished'>('idle')
```

【推荐】提倡无状态组件，性能是一个原因。我认为 stateless 重点在于阻碍了内部状态的使用，移除了生命周期，所以提高了组件的可控性，也就拓宽了组件的使用场景。

【推荐】兄弟组件传值比较多，请使用mobx，context

【推荐】尽量使用纯函数 (函数式编程,not命令式编程)

【推荐】函数组件中剥离逻辑代码

尽可能的把逻辑从组件中剥离出去，可以把必要的值用参数的形式传给工具类函数。在函数组件外组织你的逻辑让你能够更简单的去追踪 bug 和扩展你的功能。

```
//反例
export default function Component() {
  const [value, setValue] = useState('')

  function isValid() {
    // ...
  }

```

```
  return (
    <>
      <input
        value={value}
        onChange={e => setValue(e.target.value)}
        onBlur={validateInput}
      />
      <button
        onClick={() => {
          if (isValid) {
            // ...
          }
        }}
      >
        Submit
      </button>
    </>
  )
}

// 正例
function isValid(value) {
  // ...
}
export default function Component() {
  const [value, setValue] = useState('')

  return (
    <>
      <input
        value={value}
        onChange={e => setValue(e.target.value)}
        onBlur={validateInput}
      />
      <button
        onClick={() => {
          if (isValid(value)) {
            // ...
          }
        }}
      >
        Submit
      </button>
```

```
52      </>
53    )
54  }
55
```

【推荐】控制props数量、聚合props，当props的数量达到5个以上的时候，这个组件就需要被拆分了。

【推荐】不用数组的下标给跨页面传值，因为数组变化下标容易发生改变

【推荐】hooks 不能放在if判断里面

【推荐】不要过度优化,现代浏览器在运行时进行了大量的优化。很多时候，如果你再优化，那就是在浪费时间。

```
1
2  //反例
3  // On old browsers, each iteration with uncached
   `list.length` would be costly
4  // because of `list.length` recomputation. In
   modern browsers, this is optimized.
5  for (let i = 0, len = list.length; i < len; i++) {
6    // ...
7  }
8
9  //正例
10 for (let i = 0; i < list.length; i++) {
11   // ...
12 }
13
```

三、编码约定

(一)入口文件

1 【强制】开发中，不要使用任何后端的开发模式来构建APP结构，如使用MVC,MVP,MVVM等开发模式，React-Native推荐组件化，颗粒化，以上设计模式严重违背。若使用Redux,Mobx等数据流第三方，可依据第三方结构编写构建App。

2 【推荐】某些输入框的值，放入到state中，并且设置defaultValue，不要使用全局变量进行引用,参照以下方式：

```
 constructor(props) {
    super(props);
    this.state = {
        editSalesPrice:'',  //修改后的商品售价
        editPurchasePrice:'', //修改后的商品进价
    };
}

render(){
  return(
  <View style={styles.viewPadding}>
     <TextInput
          style={styles.rowInput}
          placeholder="请输入调整后的价格"
          onChangeText={(text)=>{
              this.setState({
                  editSalesPrice:text,
              })
          }}
          defaultValue={this.state.editSalesPrice}
          placeholderTextColor='#B0B7C2'
          underlineColorAndroid = 'transparent'
          autoCapitalize={'none'}
          autoCorrect={false}
          clearButtonMode={'while-editing'}
          keyboardType='numeric'
      />
  </View>
  );
}
```

3 【强制】移除定时器，监听请按照如下代码进编写；

```
componentWillUnmount() {
    //此生命周期内，去掉监听和定时器
    this.msgListener?.remove?.();
    // 如果存在this.timer，则使用clearTimeout清空。
    this.timer && clearTimeout(this.timer);
```

```
}
```

(二) 模版文件

1 【**推荐**】根据附件，配置代码编写模版，推荐使用第二种配置方式，可配置多种模版。

(三) ListView,FaltList

【**强制**】使用ListView或者FaltList的renderRow时，需对renderRow里面的组件需进行抽取，使用一个单独组件进行包裹，类似于页面子组件方式引入；

请勿使用如下方式：

```
反例：
renderRow(news) {
   return (
        <TouchableOpacity onPress={() => {
            this.onPress(news)
        }} style={styles.container}>
            <Image source=
{require('../imgs/icon_data.png')}/>
            <View style={styles.row_main}>
                <Text style={styles.row_title}>
{news.belOutlet}</Text>

                <Text style={styles.row_bottom}>所
属支行:{news.belBranch}</Text>
            </View>
            {this._renderNewFlag(news)}
        </TouchableOpacity>
    );
}

正例：
import GoodInCell from './GoodInCell';
renderRow(news) {
    return (
        <GoodInCell  news={news}  />
    );
}
```

说明：使用此方式，可增加代码的可读性和理解性。更符合组件化的开发思路。

四、自定义组件

(一) 自定义组件

【强制】组件命名中必须包含Component;

说明：

ButtonComponent.jsLabelComponent.js

【强制】组件中定义的state和props必须都要有注释，依次说明每个值的含义;

【强制】在每个类的头部注释中，必须使用/**/说明此组件的基础使用方式以及特殊使用方法；

(二) 属性判断

【强制】代码中使用props时，需进行propTypes检测和defaultProps默认值初始化；

```
static propTypes = {
      color: PropTypes.string,
      dotRadius: PropTypes.number,
      size: PropTypes.number
 };
static defaultProps = {
      color: '#1e90ff',
      dotRadius: 10,
      size: 40
};
```

2 【强制】代码中使用props方法时，具体参照以下方式进行调用；

```
     export default class PostCallMsgToPar extends
```

```
Component {
        render() {
            return (
            <View style={styles.container}>
                <TouchableOpacity onPress=
{this.postMsgByCallBack}>
                <Text>使用Callback修改父状态，无返回值
</Text>
                </TouchableOpacity>
            </View>
            );
        }

    postMsgByCallBack=()=>{
            if(this.props.onChangeMsg){
                this.props.onChangeMsg();
            }
        }
    }
```

(三) 性能优化

【强制】无状态组件需使用PureComponent而不是Component；

说明：无状态组件是指内部没有使用state的组件，但是可以使用props来进行某些属性控制;

```
export default class LinkButton extends
PureComponent {
 static  defaultProps= {
     msgName:'请输入此事件函数名！'
 };

 static  propTypes={
     msgName:PropTypes.string.isRequired,
     onPressCall:PropTypes.func,
 };


    render() {
        return (
            <View style={styles.container}>
```

```
                <TouchableOpacity onPress=
{this.onPressCall} >
                    <View>
                        <Text>{this.props.msgName}
</Text>
                    </View>
                </TouchableOpacity>
            </View>
        );
    }

    onPressCall=()=>{
        if(this.props.onPressCall){
            this.props.onPressCall();
        }
    }
}
```

（四） 【强制】组件化开发时，要做好单向数据流，避免父组件和子组件之间的相互传值

1 【推荐】使用InteractionManager.runAfterInteractions，在动画或者某些特定场景中利用InteractionManager来选择性的渲染新场景所需的最小限度的内容；

使用场景类似于：

```
class ExpensiveScene extends React.Component {
  constructor(props, context) {
    super(props, context);
    this.state = {renderPlaceholderOnly: true};
  }

  componentDidMount() {
    InteractionManager.runAfterInteractions(() => {
      this.setState({renderPlaceholderOnly:
false});
    });
  }

  render() {
    if (this.state.renderPlaceholderOnly) {
      return this.renderPlaceholderView();
```

```
    }

    return (
      <View>
        <Text>Your full view goes here</Text>
      </View>
    );
  }


  renderPlaceholderView() {
    return (
      <View>
        <Text>Loading...</Text>
      </View>
    );
  }
};
```

说明：更多使用于Navigator的页面跳转

2 【推荐】使用新版本组件替换旧办法组件；

例如：FlatList替换ListView，React-Navigation替换Navigator等

3 【推荐】组件化时，当需要使用某个库提供的api时，尝试使用库提供的hook，而不是从页面传入，现在无论是官方维护的还是社区维护的库，都会提供相对应的hook便于组件操作

4 【推荐】在使用Touchable系列组件时，进行setState或者大量调帧操作，请使用如下方式：

```
handleOnPress() {
    this.requestAnimationFrame(() => {
       //todo
    });
}
```

5 【推荐】 在使用navigation跳转到下个页面时，在进行耗时操作时，尽量渲染更少的内容来避免整个页面掉帧

正例：进行网络请求时，先加载中间空白页(或者可以做骨架屏)，接口成功后渲染完整的布局

反例：进行网络请求的同时，`render`函数渲染复杂的布局

五、第三方node_module管理

【推荐】包管理工具分为npm,yarn。推荐使用yarn。

【强制】若修改第三方node_module库，请先fokr仓库，使用svn或者git进行版本，禁止保留到本地库。

web编码规范

一、编码规范

HTML 规范

1. 【强制】使用 soft tab 缩进(即2个空格)。
2. 【强制】在属性上，使用双引号包裹属性值。

```
<div class="page">页面内容</div>
```

3. 【强制】属性名全小写，使用中划线做分隔符。

```
<div data-type="page">页面内容</div>
```

4. 【强制】在 head 中声明字符编码使用 `utf-8`

```
<head>
  <meta charset="utf-8">
</head>
```

5. 【强制】在文件开头声明文档类型 `DOCTYPE` 为 `html`

```
<!DOCTYPE html>
<html lang="en-us">
  <head>
    <meta charset="utf-8">
  </head>
```

```
6  </html>
```

6. 【推荐】属性应该按照特定的顺序出现以保证易读性；class是为高可复用组件设计的，所以应处在第一位；id更加具体且应该尽量少使用，所以将它放在第二位。
    1. class
    2. id
    3. name
    4. data-*
    5. src, for, type, href, value , max-length, max, min, pattern
    6. placeholder, title, alt
    7. aria-*, role
    8. required, readonly, disabled
7. 【推荐】添加lang属性。根据HTML5规范：应在html标签上加上lang属性。这会给语音工具和翻译工具帮助，告诉它们应当怎么去发音和翻译。lang 属性对每张页面中的主要语言进行声明，也只是个声明，主要是根据W3C标准，对搜索引擎和浏览器友好。更多关于 lang 属性的说明在这里；在sitepoint上可以查到语言列表；但sitepoint只是给出了语言的大类，例如中文只给出了zh，但是没有区分香港，台湾，大陆。而微软给出了一份更加详细的语言列表，其中细分了zh-cn, zh-hk, zh-tw。

```
1  <html lang="en-us">
2  </html>
```

8. 【推荐】boolean属性不需要声明属性值。boolean属性指不需要声明取值的属性，XHTML需要每个属性声明取值，但是HTML5并不需要；boolean属性的存在表示取值为true，不存在则表示取值为false。

```
1  <input type="text" disabled>
```

9. 【推荐】减少标签数量。在编写HTML代码时，需要尽量避免多余的父节点。

CSS 规范

CSS 规范我们参考 stylelint-config-standard 标准，这个标准集合了行业里主流公司的规范，目前是前端业界主推的规范。我们希望所有的前端团队都能引入这个优秀的工具，规范团队的 CSS 代码。

1. 【强制】前端样式仅允许使用 CSS、SCSS、LESS 开发。
2. 【强制】所有的 css 文件使用 soft tab，即2个空格。
3. 【强制】class 、id 命名采用中划线命名规范（kebab-case）。
4. 【强制】scss 中变量、函数、混合、placeholder 均采用小驼峰命名规范（camelCase）
5. 【强制】不允许有空的规则。
6. 【强制】去掉数字中要的小数点和末尾的0。
7. 【强制】属性值 0 后面不要加单位。
8. 【强制】不要再一个文件中出现两个相同的规则。
9. 【强制】用 `border: 0;` 代替 `border: none;` 。
10. 【强制】发布的代码中不要有 `@import` 。
11. 【强制】提交的代码中不要有 `@debug` 。
12. 【强制】选择器不要超过4层。
13. 【强制】每个属性声明末尾需要加分号。
14. 【强制】在以下场景，不能加空格
    1. 属性名后
    2. 多个规则的分隔符','前
    3. !important '!'后
    4. 属性值中'('后和')'前
    5. 行末不要有多余的空格
15. 【强制】在以下场景中，需要加空格：
    1. 属性值前
    2. 选择器'>', '+', '~'前后
    3. '{'前
    4. !important '!'前
    5. @else 前后
    6. 属性值中的','后
    7. 注释'/后和'*/'前

16. 【强制】以下场景中，需要加空行：
    1. 文件最后保留一个空行
    2. '}'后最好跟一个空行，包括scss中嵌套的规则
    3. 属性之间需要适当的空行，具体见 http://alloyteam.github.io/CodeGuide/#css-declaration-order
17. 【强制】以下场景下需要换行：
    1. '{'后和'}'前
    2. 每个属性独占一行
    3. 多个规则的分隔符','后
18. 【强制】属性值如果需要引号包裹，统一使用双引号。
19. 【强制】url 中的内容使用引号包裹。
20. 【强制】注释的缩进与下一行代码保持一致；可位于一个代码行末尾，与代码间隔一个空格。
21. 【推荐】前端项目采用 stylelint 的 standard 规范进行样式检查，点击查看详情。
22. 【推荐】CSS属性编写顺序。CSS属性编写顺序（一般遵循显示属性-> 自身属性-> 文本属性-> 其他属性的书写格式）
    1. 显示属性：display/list-style/position/float/clear...
    2. 自身属性（盒模型）：width/height/margin/padding/border
    3. 背景：background
    4. 行高：line-height
    5. 文本属性：color/font/text-decoration/text-align/text-indent/vertical-align/white-space/content...
    6. 其他：cursor/z-index/zoom/overflow...
    7. CSS3属性：transform/transition/animation/box-shadow/border-radius 无前缀的标准属性应该写在有前缀的属性后面
23. 【推荐】Sass 相关
    1. 声明顺序：
        1. @extend
        2. 不包含 @content 的 @include
        3. 包含 @content 的 @include
        4. 自身属性
    2. @import 引入的文件不需要开头的'_'和结尾的'.scss'；

3. 嵌套最多不能超过5层；

24. 【推荐】Less 相关
    1. 代码顺序组织：
        1. @import
        2. 变量声明
        3. 样式声明
        4. @import语句引用的文需要写在一对双引号内，.less 后缀不得省略。
        5. @import "mixins/size.less";
    2. 混入（Mixin）
        1. 在定义 mixin 时，如果 mixin 名称不是一个需要使用的 className，必须加上括号，否则即使不被调用也会输出到 CSS 中。
        2. 如果混入的是本身不输出内容的 mixin，需要在mixin 后添加括号（即使不传参数），以区分这是否是一个 className。

Javascript 规范

JS 规范我们采用业界最主流的 Airbnb 的编码规范，此规范目前在 Github 上的 star 数量是前端项目的排名前五，也是业界最推崇的行业规范。对于很多规范的设计细节都体现了严谨的态度，比如我们常见的行末分号、数组最后元素后面的逗号等，从规范层面就避免了一些潜在的质量问题的发生。

我们会统一采用基于eslint的airbnb-base 编码规范（ https://github.com/airbnb/javascript），各个团队可适当自定义配置，覆盖airbnb-base 中的配置，但基本代码风格规则配置遵循以下【编码风格】中已列出规范（ 在下一章的【项目配置】里会提供eslint配置模板）

1. 【强制】标准变量采用驼峰命名（camelCase）。
2. 【强制】常量命名字母全部大写，使用下划线连接。
3. 【强制】构造函数采用大驼峰命名方式（PascalCase）。
4. 【强制】'Android'在变量名中大写第一个字母，'iOS'在变量名中小写第一个，大写后两个字母。
5. 【强制】立即执行函数必须包一层括号。

6. 【强制】对象以缩写的形式书写，不要写在一行。

```
// bad
const obj = {
  lukeSkywalker: lukeSkywalker,
};

// good
const obj = {
  lukeSkywalker,
};
```

6. 【强制】jQuery 变量必须以 $ 开头
7. 【强制】使用 soft tab(2个空格)作为缩进，不能 tab 和 space 混用。
8. 【强制】不能有空的代码块。
9. 【强制】switch的falling through和no default的情况一定要有注释特别说明。
10. 【强制】下列关键字后必须有大括号（即使代码块的内容只有一行）：`if, else, for, while, do, switch, try, catch, finally, with` 。
11. 【强制】公用组件提供 API 文档：模块目录中添加 README.md 文件,在 README文件中说明模块的功能以及使用场景。对于vue组件来说，比较有用的描述是组件的自定义属性即API 的描述介绍。目录结构如下所示：

```
range-slider/
├── range-slider.vue
├── range-slider.less
└── README.md
```

12. 【强制】语句末尾必须加分号。
13. 【强制】以下几种情况不需要加空格：
    1. 对象的属性名后（eslint: key-spacing）
    2. 函数调用括号前（eslint: func-call-spacing）
    3. 数组的'['后和']'前（eslint: array-bracket-spacing）

4. 对象的'{'后和'}'前（eslint: object-curly-spacing）
5. 运算符'('后和')'前（eslint: space-in-parens）
6. function的左括号之前（eslint: space-before-function-paren）

14. 【强制】以下几种情况需要加空格：
    1. 操作符前后（eslint: space-infix-ops）
    2. 代码块'{'前（eslint: space-before-blocks）
    3. 单行注释'//'后（若单行注释和代码同行，则'//'前也需要），多行注释'*'后（eslint: spaced-comment）
    4. 对象的属性值前（eslint: key-spacing）
    5. for循环，分号后留有一个空格，前置条件如果有多个，（eslint: semi-spacing）
    6. 逗号后留一个空格（eslint: comma-spacing）
15. 【强制】以下几种情况，需要增加空行：
    1. 变量声明后（当变量声明在代码块的最后一行时，则无需空行）
    2. 注释前（当注释在代码块的第一行时，则无需空行）（eslint: lines-around-comment）
    3. 代码块后（在函数调用、数组、对象中则无需空行）
    4. 文件最后保留一个空行（eslint: eol-last）
16. 【强制】以下情况，需要换行：
    1. 代码块'{'后和'}'前（eslint: object-curly-newline）
    2. 变量赋值后（eslint:one-var-declaration-per-line）
17. 【强制】以下情况，不需要换行：
    1. 代码块'{'前（eslint：brace-style）
18. 【强制】字符串值用单引号「''」或者「``」包裹。

Vue 规范

Vue 规范我们参考自 vue风格指南，摘取官方制定的必要规范，目的是为了规避错误。

1. 【强制】组件名为多个单词。

```
1  Vue.component('todo-item', {
```

```
  // ...
})

export default {
  name: 'TodoItem',
  // ...
}
```

2. 【强制】组件的 `data` 必须是一个函数。

```
export default {
  data () {
    return {
      foo: 'bar'
    }
  }
}
```

3. 【强制】Prop 定义应该详尽，至少需要指定其类型。

```
// 更好的做法!
props: {
  status: {
    type: String,
    required: true,
    validator: function (value) {
      return [
        'syncing',
        'synced',
        'version-conflict',
        'error'
      ].indexOf(value) !== -1
    }
  }
}

```

4. 【强制】为 `v-for` 设置键值。在组件上必须用 key 配合 v-for 使用。

```
<ul>
  <li
    v-for="todo in todos"
    :key="todo.id"
  >
    {{ todo.text }}
  </li>
</ul>

```

5. 【强制】避免 `v-for` 和 `v-if` 同时用在一个元素上。

```
<ul v-if="shouldShowUsers">
  <li
    v-for="user in users"
    :key="user.id"
  >
    {{ user.name }}
  </li>
</ul>
```

6. 【强制】为组件样式设置作用域。

```
<template>
  <button class="button button-close">X</button>
</template>

<!-- 使用 `scoped` attribute -->
<style scoped>
.button {
  border: none;
  border-radius: 2px;
}

.button-close {
  background-color: red;
}
</style>
```

7. 【推荐】项目目录结构如下设计：

```
src目录：
├── App.vue // APP入口文件
├── components  // 组件文件夹
│   └──common  // 共用组件文件夹
│   └──page  // 我们的页面组件文件夹
│   │   ├── home   // 模块划分文件夹
│   │   ├── user     // 模块划分文件夹
│   └──Main.vue    // 项目主入口文件
├── plugins  // 项目配置文件夹
│   └── ajax // 网络请求配置文件
│   │   ├── http.js     // 网络请求封装文件
│   └── api  // 常用工具文件夹
│   │   ├── index.js     // 自定义工具注册入口
│   │   ├── ValideCard.js     // 身份证验证规则文件
│   │   ├── valideEmail.js     // 邮箱验证规则
│   │   ├── valideTel.js         // 手机号证规则
│   └── url  // 项目配置文件
│   │   ├── index.js     // 接口注册入口
│   │   ├── url.js         // 接口API自定义文件
├── main.js  // 项目配置文件
├── router  // 路由配置文件夹
│   └── index.js  // 路由配置文件
├── style  // scss 样式存放目录（待定，选用less或scss）
│   ├── base   // 基础样式存放目录
│   │   ├── \_base.scss   // 基础样式文件
│   │   ├── \_color.scss  // 项目颜色配置变量文件
│   │   ├── \_mixin.scss  // scss 混入文件
│   │   └── \_reset.scss  // 浏览器初始化文件
│   ├── scss   // 页面样式文件夹
│   │   ├── \_content.scss  // 内容页面样式文件
│   │   └── \_index.scss  // 列表样式文件
        └── style.scss  // 主样式文件

```

8. 【推荐】使用 `plugin:vue/vue3-recommended` 作为项目的
9. 【推荐】使用 `Vue3 + Typescript` 开发项目。
10. 【推荐】组件结构化（代码顺序）。按照一定的结构组织，使得组件便于理解、容易阅读

    1. template
    2. script
        1. name
        2. extends
        3. props
        4. data
        5. computed
        6. components
        7. watch
        8. lifecycle methods
        9. methods
    3. style
11. 【推荐】公共组件代码最好不超过300行。
12. 【推荐】组件名中的单词顺序：应该以高级别的 (通常是一般化描述的) 单词开头，以描述性的修饰词结尾。
13. 【推荐】完整单词的组件名：应该倾向于完整单词而不是缩写。
14. 【推荐】多个 attribute 的元素：分多行撰写，每个 attribute 一行。

React 规范

React 我们主要参考 jsx-eslint。

1. 【强制】在JSX中禁止使用if语句进行判定
2. 【强制】始终"对JSX属性使用双引号（eslint: jsx-quotes）
3. 【强制】当JSX标签跨越多行时，将它们括在括号中（eslint: react/jsx-wrap-multilines）
4. 【强制】总是在Refs里使用回调函数（eslint: react/no-string-refs）
5. 【强制】当标签没有子元素的时候，始终使用自闭合的标签（eslint: react/self-closing-comp）
6. 【强制】如果标签有多行属性，关闭标签要另起一行 （eslint: react/jsx-closing-bracket-location）
7. 【强制】在自闭标签之前留一个空格（eslint: no-multi-spaces, react/jsx-tag-spacing）

8. 【强制】如果属性值为 true, 可以直接省略.（eslint: react/jsx-boolean-value）1.6.2 杂项
9. 【强制】render方法必须有值返回
10. 【强制】State 的更新可能是异步的
11. 【推荐】React 项目的目录结构如下如下设计：

```
├── bin                          # 启动脚本
├── blueprints                   # redux-cli的蓝图
├── build                        # 所有打包配置项
│   └── webpack                  # webpack的指定环境配置文件
├── config                       # 项目配置文件
├── server                       # Express 程序 (使用 webpack 中间件)
│   └── main.js                  # 服务端程序入口文件
├── src                          # 程序源文件
│   ├── main.js                  # 程序启动和渲染
│   ├── components               # 全局可复用的表现组件 (Presentational Components)
│   ├── containers               # 全局可复用的容器组件
│   ├── layouts                  # 主页结构
│   ├── static                   # 静态文件(不要到处 imported源文件)
│   ├── styles                   # 程序样式
│   ├── store                    # Redux指定块
│   │   ├── createStore.js       # 创建和使用redux store
│   │   └── reducers.js          # Reducer注册和注入
│   └── routes                   # 主路由和异步分割点
│       ├── index.js             # 用store启动主程序路由
│       ├── Root.js              # 为上下文providers包住组件
│       └── Home                 # 不规则路由
│           ├── index.js         # 路由定义和代码异步分割
│           ├── assets           # 组件引入的静态资源
│           ├── components       # 直观React组件
│           ├── container        # 连接actions和store
│           ├── modules          # reducers/constants/actions的集合
│           └── routes \*\*      # 不规则子路由(\*\* 可选择的)
└── tests                        # 单元测试
```

```
29
```

10. 【推荐】使用 `Typescript` 开发项目。根据 Sentry 报错统计，引入 `Typescript` 可以避免 90% 生产空指针问题。
11. 【推荐】继承React.Component 的类的方法遵循下面的顺序。组件应该有严格的代码顺序，这样有利于代码维护，我们推荐每个组件中的代码顺序一致性
    1. constructor
    2. optional static methods
    3. getChildContext
    4. componentWillMount
    5. componentDidMount
    6. componentWillReceiveProps
    7. shouldComponentUpdate
    8. componentWillUpdate
    9. componentDidUpdate
    10. componentWillUnmount
    11. clickHandlers or eventHandlers like onClickSubmit() or onChangeDescription()
    12. getter methods for render like getSelectReason() or getFooterContent()
    13. Optional render methods like renderNavigation() or renderProfilePicture()
    14. render
12. 【推荐】使用 React.createClass 时，方法顺序如下：
    1. displayName
    2. propTypes
    3. contextTypes
    4. childContextTypes
    5. mixins
    6. statics
    7. defaultProps
    8. getDefaultProps
    9. getInitialState
    10. getChildContext

    11. componentWillMount
    12. componentDidMount
    13. componentWillReceiveProps
    14. shouldComponentUpdate
    15. componentWillUpdate
    16. componentDidUpdate
    17. componentWillUnmount
    18. clickHandlers or eventHandlers like onClickSubmit() or onChangeDescription()
    19. getter methods for render like getSelectReason() or getFooterContent()
    20. Optional render methods like renderNavigation() or renderProfilePicture()
    21. render
13. 【推荐】使用箭头函数来获取本地变量.
14. 【推荐】推荐使用 Context，如果某个属性在组件树的不同层级的组件之间需要用到，我们应该使用 Context 提供在组件之间共享此属性的方式，而不不是显式地通过组件树的逐层传递 props。

项目规范

1. 【强制】项目名称全部采用中划线命名规则。(kebab-case)

```
1  例: my-project-name
```

2. 【强制】目录命名以采用小驼峰命名规则。（camelCase）

```
1  例: scripts, dataModels, config
```

3. 【强制】文件命名采用小驼峰命名规则。（camelCase）

```
例：accountModel.ts
```

4. 【强制】组件命名采用大驼峰命名规则。（PascalCase）

```
例：UserCenter.vue, EditTable.vue
```

- 组件名：组件名应为多个单词
- 基础组件名：应该全部以一个特定的前缀开头，比如 Base、App 或 V
- 单例组件名：应该以 The 前缀命名，以示其唯一性
- 紧密耦合的组件名：应该以父组件名作为前缀命名

5. 【强制】 URL 路由采用中划线分割(kebab-case)的方式命名

```
例：/user/capital-history
```

6. 【强制】项目中，必须在根目录下有 README.md 文件，阐述项目的背景、目的、技术栈、调试、发布、维护人员。

二、项目配置

- husky、 lint-staged、 **commitizen** 的工作流
- ESLint + styleLint + Prettier规范

(一) husky、lint-staged

- **husky**： Git hooks 工具（https://github.com/typicode/husky/tree/master）
- **lint-staged**：**在git暂存文件上运行linters的工具（https://github.com/okonet/lint-staged）

安装：

npm install husky --save-dev

npm install lint-staged --save-dev

配置：

```
//package.json

{
......
  "husky": {
    "hooks": {
      "pre-commit": "lint-staged"
    }
  },

  "lint-staged": {

  //启用 eslint 代码检查、修复
  "\*.{js,vue}": "eslint --cache --fix",

  //启用 stylelint 代码检查、修复
  "src/\*\*/\*.less": "stylelint --fix",

  //启用 prettier 代码格式化
  "src/\*\*/\*.{js,vue,less}": "prettier-eslint --write"

  }

}
```

(二) ESlint

JavaScript 语法规则和代码风格的检查工具

qwertvue 项目

安装：

npm install -D eslint eslint-config-airbnb-base

npm install -D babel-eslint

npm install -D eslint-plugin-import eslint-import-resolver-webpack//帮助 eslint 解决 webpack里面配置的别名和路径

npm install -D eslint-plugin-vue

npx install-peerdeps --dev eslint-config-airbnb-base

- eslint：JavaScript 语法规则和代码风格的检查插件
- eslint-config-airbnb-base：airbnb 编码规范
- babel-eslint：eslint 与 babel 整合包
- eslint-plugin-import：该插件旨在支持ES2015 +（ES6 +）导入/导出语法的检查，并防止文件路径和导入名称拼写错误的问题
- eslint-import-resolver-webpack：eslint-plugin-import的Webpack-literate模块解析插件，解决 webpack里面配置的别名和路径
- eslint-plugin-vue： 用于Vue的ESLint规则

配置：

//.eslintrc.js

```
module.exports = {
  root: true,
  env: {
    browser: true,//浏览器环境中的全局变量
    node: true,//Node.js 全局变量和 Node.js 作用域
    es6: true, //启用除了modules以外的所有ES6特性（该选项会自动设置 ecmaVersion 解析器选项为 6）
  },
  extends: [
    'plugin:vue/essential',
    'airbnb-base', // 继承airbnb-base中的配置
    'plugin:prettier/recommended', //避免与 prettier 冲突
  ],

// eslint 适配 webpack里面配置的别名和路径

settings: {
  'import/resolver': {
    node: {
      extensions: [
        '.js',
        '.vue',
      ],

      alias: {
```

```
        '@': './src',
      },
    },
    webpack: {
      config: 'build/webpack.base.conf.js',
    },
  },
},
rules: {
  "no-implicit-coercion": 2, // 禁止使用短符号进行类型转换
  'no-unsafe-finally': 2, // 禁止在 finally 语句块中出现控制流语句
  'max-len': 0, // 强制一行的最大长度
  'no-underscore-dangle': 0, // 禁止标识符中有悬空下划线
  'prefer-destructuring': 0, // 优先使用数组和对象解构
  'no-param-reassign': 0, // 禁止对 function 的参数进行重新赋值
  'consistent-return': 0, // 要求 return 语句要么总是指定返回的值，要么不指定
  'no-bitwise': 0, // 禁止使用按位操作符
  'no-unused-expressions': 1, // 禁止未使用过的表达式
  'import/no-extraneous-dependencies': 1, // 禁止使用无关的软件包
  'func-names': 0, // 要求或禁止使用命名的 function 表达式
  'no-plusplus': 0, // 禁用一元操作符 ++ 和 --
  'object-shorthand': 0, // 强制对象字面量缩写语法
  'arrow-parens': 0, // 要求箭头函数的参数使用圆括号
  'max-lines-per-function': 0, // 强制函数最大行数
  'no-irregular-whitespace': 2, // 禁止不规则的空白
  'no-redeclare': 2, // 禁止多次声明同一变量
  'import/extensions': ['error', 'always', {
  js: 'never',
  vue: 'never',
  }], // 确保在导入路径中一致使用文件扩展名
  semi: [2, 'always'], // 句末加分号
  'lines-around-comment': ['error', {
      beforeBlockComment: true,
      allowBlockStart: true,
  }], // 块级注释前空一行
},
```

```
parserOptions: {
  parser: 'babel-eslint',
 },
};
```

qwerttreact 项目

安装：

npm install -D eslint eslint-config-airbnb-base

npm install -D babel-eslint

npm install -D eslint-plugin-import eslint-import-resolver-webpack

npm install -D eslint-plugin-react eslint-plugin-jsx-a11y eslint-plugin-react-hooks

npm install -D @typescript-eslint/eslint-plugin @typescript-eslint/parser

npx install-peerdeps --dev eslint-config-airbnb-base

- eslint：JavaScript 语法规则和代码风格的检查插件
- eslint-config-airbnb-base：airbnb 编码规范
- babel-eslint：eslint 与 babel 整合包
- eslint-plugin-import：该插件旨在支持ES2015 +（ES6 +）导入/导出语法的检查，并防止文件路径和导入名称拼写错误的问题
- eslint-import-resolver-webpack：eslint-plugin-import的Webpack-literate模块解析插件，解决 webpack里面配置的别名和路径
- eslint-plugin-react： 用于React的ESLint规则
- eslint-plugin-jsx-a11y：提供 jsx 元素可访问性校验
- eslint-plugin-react-hooks：根据 Hooks API 校验 Hooks 的使用
- @typescript-eslint/eslint-plugin： 一个ESLint插件，为TypeScript代码库提供lint规则
- ~~@typescript-eslint/eslint-plugin-tslint 允许在ESLint中运行TSLint的完整实例~~
- @typescript-eslint/parser：将 TypeScript 转换为 ESTree，使 eslint 可以识别

配置：

// .eslintrc.js

```
module.exports = {
  root: true,
  extends: [
    'airbnb-base',
    'plugin:@typescript-eslint/recommended', //使用@typescript-eslint/eslint-plugin中的推荐规则
    'plugin:react/recommended', //使用@eslint-plugin-react中的推荐规则
    'prettier/@typescript-eslint', //使用eslint-config-prettier禁用@typescript-eslint/eslint-plugin中的ESLint规则，避免与 prettier 冲突
    'plugin:prettier/recommended', //启用eslint-plugin-prettier并将prettier的错误显示为ESLint错误。确保这始终是扩展数组中的最后一个配置。
  ],

  env: {
    'browser': true
    'es6': true
  },

  parser: '@typescript-eslint/parser',
  parserOptions: {
    project: 'tsconfig.json',
    sourceType: 'module',
    ecmaFeatures: {
      'jsx': true
    }
  },

  plugins: [
    'react',
    '@typescript-eslint',
    //'@typescript-eslint/tslint',
    'react-hooks',
    'jest',
    'prettier',
  ],

  overrides: [
    {
```

```
      files: ['\*.ts', '\*.tsx'],
      rules: {
        '@typescript-eslint/no-unused-vars': [2, { args: 'none' }],
        'no-unused-expressions': 'off',
        '@typescript-eslint/no-unused-expressions': 2,
      },

    },
  ],

  // eslint 适配 webpack里面配置的别名和路径

  settings: {
    'import/resolver': {
      node: {
        extensions: [
          '.ts',
          '.tsx',
        ],
        alias: {
          '@': './src',
        },
      },
      webpack: {
          //config: 'build/webpack.base.conf.js',
      },
    },
  },

    rules: {
    "no-implicit-coercion": 2, // 禁止使用短符号进行类型转换
    'no-unsafe-finally': 2, // 禁止在 finally 语句块中出现控制流语句
    'max-len': 0, // 强制一行的最大长度
    'no-underscore-dangle': 0, // 禁止标识符中有悬空下划线
    'prefer-destructuring': 0, // 优先使用数组和对象解构
    'no-param-reassign': 0, // 禁止对 function 的参数进行重新赋值
```

```
    'consistent-return': 0, // 要求 return 语句要么总是指定返回的值，要么不指定
    'no-bitwise': 0, // 禁止使用按位操作符
    'no-unused-expressions': 1, // 禁止未使用过的表达式
    'import/no-extraneous-dependencies': 1, // 禁止使用无关的软件包
    'func-names': 0, // 要求或禁止使用命名的 function 表达式
    'no-plusplus': 0, // 禁用一元操作符 ++ 和 --
    'object-shorthand': 0, // 强制对象字面量缩写语法
    'arrow-parens': 0, // 要求箭头函数的参数使用圆括号
    'max-lines-per-function': 0, // 强制函数最大行数
    'no-irregular-whitespace': 2, // 禁止不规则的空白
    'no-redeclare': 2, // 禁止多次声明同一变量
    'import/extensions': ['error', 'always', {
      js: 'never',
      vue: 'never',
    }], // 确保在导入路径中一致使用文件扩展名
    semi: [2, 'always'], // 句末加分号
    'lines-around-comment': ['error', {
      beforeBlockComment: true,
      allowBlockStart: true,
    }], // 块级注释前空一行

    'jsx-a11y/no-static-element-interactions': 0, // 强制<div>具有单击处理程序的非交互式可见元素（例如）使用role属性
    'jsx-a11y/click-events-have-key-events': 0,// 强制一个可单击的非交互式元素具有至少一个键盘事件侦听器
    'jsx-a11y/anchor-is-valid': 0, // 强制所有锚点都是有效的可导航元素
    'react/jsx-filename-extension': 0,// 限制可能包含JSX的文件扩展名
    'react/destructuring-assignment': 0,// 强制使用一致的props, state, context解构分配
    'react-hooks/rules-of-hooks': 'error', // 检查钩子规则 https://reactjs.org/docs/hooks-rules.html
    'react-hooks/exhaustive-deps': 'warn', // Checks effect dependencies
  },
};

```

```
//tsconfig.json

{
  "compilerOptions": {
    "target": "es5", // 指定 ECMA 版本
    "lib": [
      "dom",
      "dom.iterable",
      "esnext"
    ], // 要包含在编译中的依赖库文件列表

    "allowJs": true, // 允许编译 Java 文件
    "skipLibCheck": true, // 跳过所有声明文件的类型检查
    "esModuleInterop": true, // 禁用命名空间引用 (import \* as fs from "fs") 启用 CJS/AMD/UMD 风格引用 (import fs from "fs")
    "allowSyntheticDefaultImports": true, // 允许从没有默认导出的模块进行默认导入
    "strict": true, // 启用所有严格类型检查选项
    "forceConsistentCasingInFileNames": true, // 不允许对同一个文件使用不一致格式的引用
    "noFallthroughCasesInSwitch": true, // 报告switch语句的fallthrough错误（不允许switch的case语句贯穿）
    "module": "esnext", // 指定模块代码生成
    "moduleResolution": "node", // 使用 Node.js 风格解析模块
    "resolveJsonModule": true, // 允许使用 .json 扩展名导入的模块
    "isolatedModules": true, // 无条件地给没有解析的文件生成imports
    "noEmit": true, // 不输出(意思是不编译代码，只执行类型检查)
    "jsx": "react-jsx", // 在.tsx文件中支持JSX
    "sourceMap": true, // 生成相应的.map文件
    "declaration": true, // 生成相应的.d.ts文件
    "noUnusedLocals": true, // 报告未使用的本地变量的错误
    "noUnusedParameters": true, // 报告未使用参数的错误
    "experimentalDecorators": true, // 启用对ES装饰器的实验性支持
    "incremental": true, // 通过从以前的编译中读取/写入
```

```
    信息到磁盘上的文件来启用增量编译
  },

  "include": [
    "src"
  ],

  "exclude": [
    "node\_modules",
    "build"
  ], // \*\*\* 不进行类型检查的文件 \*\*\*
}
```

webpack配置

新项目可增加webpack loader配置，通过在webpack中引入eslint-loader来启动eslint，启动webpack-dev-server的时候，每次保存代码同时自动对代码进行格式化。*（注意关闭编辑器的保存自动修复功能）*

安装：**

//webpack插件

npm install --save-dev eslint-loader

配置：

// package.json

```
rules: [
  {
    test: /\.(vue|ts|js)$/,
    exclude: /node\_modules/,
    enforce: 'pre',
    loader: 'eslint-loader',
    options: {
      fix: true,
      emitWarning: false,
    },
  },
],
```

eslintignore

/build/**

/config/**

/dist/**

/node_modules/**

/*.js

!.eslintrc.js

老项目处理方案

1 如老项目已经配置eslint

继续沿用之前的规则，但是编码规范里明确的缩进、分号、引号、空格等等保持一致

2 检查之前的规则，加上注释，如果一些很好的规则被放开了，改为提醒

3 如老项目无eslint

引入eslint检查，配置参考如下（为避免需要改动之前的代码逻辑，一些规则设置为提醒）

//.eslintrc.js

```
module.exports = {
    root: true,
    env: {
        browser: true,//浏览器环境中的全局变量
        node: true,//Node.js 全局变量和 Node.js 作用域
        es6: true, //启用除了modules以外的所有ES6特性（该选项会自动设置 ecmaVersion 解析器选项为 6）
    },

```

```
  extends: [
     'plugin:vue/essential',
     'airbnb-base', // 继承airbnb-base中的配置
     'plugin:prettier/recommended', //避免与 prettier 冲突
  ],

// eslint 适配 webpack里面配置的别名和路径

settings: {
  'import/resolver': {
    node: {
     extensions: [
        '.js',
        '.vue',
     ],
     alias: {
        '@': './src',
     },
   },
   webpack: {
     config: 'build/webpack.base.conf.js',
   },
  },
},

rules: {
  eqeqeq: 1, // 要求使用 === 和 !==
  'no-console': 1,
  'max-len': 0, // 强制一行的最大长度
  'no-underscore-dangle': 0, // 禁止标识符中有悬空下划线
  'prefer-destructuring': 0, // 优先使用数组和对象解构
  'no-param-reassign': 0, // 禁止对 function 的参数进行重新赋值
  'consistent-return': 0, // 要求 return 语句要么总是指定返回的值，要么不指定
  'no-bitwise': 0, // 禁止使用按位操作符
  'linebreak-style': 0, // 强制使用一致的换行风格
  'no-unused-expressions': 1, // 禁止未使用过的表达式
  'no-restricted-globals': 1, // 禁用特定的全局变量
  'no-restricted-syntax': 1, // 禁止使用特定的语法
```

```
    'import/no-extraneous-dependencies': 1, // 禁止使用无关的软件包
    'func-names': 0, // 要求或禁止使用命名的 function 表达式
    'no-plusplus': 0, // 禁用一元操作符 ++ 和 --
    'object-shorthand': 0, // 强制对象字面量缩写语法
    'arrow-parens': 0, // 要求箭头函数的参数使用圆括号
    'array-callback-return': 1, // 强制数组方法的回调函数中有 return 语句
    'no-nested-ternary': 1, // 禁用嵌套的三元表达式
    'no-shadow': 1, // 禁止变量声明覆盖外层作用域的变量
    'max-lines-per-function': 0, // 强制函数最大行数
    'import/extensions': ['error', 'always', {
       js: 'never',
       vue: 'never',
     }], // 确保在导入路径中一致使用文件扩展名
    semi: [2, 'always'], // 句末加分号
    'lines-around-comment': ['error', {
       beforeBlockComment: true,
       allowBlockStart: true,
     }], // 块级注释前空一行
  },
  parserOptions: {
   parser: 'babel-eslint',
  },
};
```

yu'iop (三) prettier

前端代码格式化工具

安装：

npm i -D prettier eslint-plugin-prettier eslint-config-prettier prettier-eslint-cli

- prettier：prettier插件的核心代码
- eslint-plugin-prettier：将prettier作为ESLint规范来使用
- eslint-config-prettier：解决ESLint中的样式规范和prettier中样式规范的冲突，以prettier的样式规范为准，使ESLint中的样式规范自动失效
- prettier-eslint-cli:prettier-eslint-cli 允许你对多个文件用prettier-eslint进行格式化。

配置：

// .prettierrc.js

```
module.exports = {
  singleQuote: true,
  trailingComma: 'all',
  printWidth: 100,
  proseWrap: 'never',
  endOfLine: 'lf',
  tabWidth: 2
};
```

// .prettierignore

```
\*\*/\*.svg
\*\*/\*.woff
\*\*/\*.snap
package.json
package-lock.json
/dist
/static
.gitignore
.stylelintignore
.eslintignore
.prettierignore
\*.png
\*.jpg
\*.toml
.editorconfig
.eslintcache
.history
favicon.icon
test.jsontest.http
yarn.lock
/node\_modules
eslint-result.xml
/testReport
```

(四) stylelint

CSS linter 工具，用于校对风格的规则、用于判别代码可维护性的规则、以及用于判断代码错误的规则

安装：

npm install --save-dev stylelint stylelint-config-standard stylelint-config-prettier

配置：

// .stylelintrc.js

```
module.exports = {
  "extends": "stylelint-config-standard",
  "rules": {
    "indentation": 4,
    "string-quotes": "single",
    "property-no-unknown": [
      true,
      {
        "ignoreProperties": ["composes"]
      }
    ],
    "selector-pseudo-class-no-unknown": [
      true,
      {
        "ignorePseudoClasses": ["global"]
      }
    ],
    "no-descending-specificity": [
      true,
      {
        "ignore": ["selectors-within-list"]
      }
    ]
  }
}
```

//.stylelintignore

```
/dist/\*\*
/node\_modules/\*\*
```

(五) commitizen

commit 提交规范

安装：

// NPM版本高于5.2

npm install --save-dev commitizen

npx commitizen init cz-customizable --save-dev --save-exact

npx git-cz

配置：

// .cz-config.js

```
module.exports = {
types: [
    { value: '特性', name: '特性: 一个新的特性' },
    { value: '修复', name: '修复: 修复了一个bug' },
    { value: '文档', name: '文档: 变更了文档' },
    { value: 'UI', name: 'UI: 更新UI',},
    { value: '性能', name: '性能: 提升性能', },
    { value: '测试', name: '测试: 添加一个测试' },
    { value: '配置', name: '配置: 配置文件改动'},
    { value: '依赖', name: '依赖: 添加依赖包'},
    { value: '重构', name: '重构: 代码重构，注意和特性、修复区分开',},
    { value: '删除', name: '删除: 删除代码/文件' },
    { value: '回滚', name: '回滚: 代码回退' },
  ],
scopes: [{
    name: '首页'
  },{
    name: '项目管理'
  },{
    name: '产品管理'
  },{
    name: '基金评价'
  },{
    name: '存续管理'
  }, {
```

```
    name: '交易管理'
  }, {
    name: '系统管理'
  },{
    name: '其他'
  }],
allowTicketNumber: false,
isTicketNumberRequired: false,
ticketNumberPrefix: 'TICKET-',
ticketNumberRegExp: '\\d{1,5}',
// it needs to match the value for field type. Eg.: 'fix'
/\*
  scopeOverrides: {
    fix: [
      {name: 'merge'},
      {name: 'style'},
      {name: 'e2eTest'},
      {name: 'unitTest'}
    ]
  },
\*/

  // override the messages, defaults are as follows
  messages: {
    type: '选择更改类型:\n',
    scope: '更改的范围:\n',
    // 如果allowcustomscopes为true，则使用
    // customScope: 'Denote the SCOPE of this change:',
    subject: '简短描述:\n',
    body: '详细描述. 使用"|"换行:\n',
    //breaking: 'Breaking Changes列表:\n',
    footer: '关闭的issues/bug列表. E.g.: #issue-31, #bug-34:\n',
    confirmCommit: '确认提交?'
  },

  allowCustomScopes: true,
  allowBreakingChanges: ['feat', 'fix'],
  // skip any questions you want
  skipQuestions: ['body'],
  // limit subject length
```

```
  subjectLimit: 100,
  // breaklineChar: '|', // It is supported for fields body and footer.
  // footerPrefix : 'ISSUES CLOSED:'
  // askForBreakingChangeFirst : true, // default is false
};
```

(六) editorconfig

EditorConfig有助于维护跨多个编辑器和IDE从事同一项目的多个开发人员的一致编码风格。(https://editorconfig.org/)

配置：

```
// .editorconfig

root = true
[\*]
charset = utf-8
indent\_style = space
indent\_size = 2
end\_of\_line = lf
insert\_final\_newline = true
trim\_trailing\_whitespace = true
insert\_final\_newline = true
```

(七) 示例

eslint检查不通过：

提交规范（使用 git cz 代替 gitcommit）：

(八) 项目参与人员准备工作

拉取代码，更新配置文件

2 重新 npm install，安装相关依赖

3 确认vscode是否可以编辑保存的时候，是否会自动格式化代码，包括：

- eslint 自动修复
- stylelint 自动修复

如不，检查是否安装相关插件 ,参考下面的 vscode 配置

使用 git cz 代替 git commit 提交代码

三、vscode配置

(一) 插件安装

Git Blame：**会告诉你这行代码是在哪个提交和哪个分支上修改的。并且能够跳转到 web。

GitLens：每行代码最新的修改一目了然

vetur：** 语法高亮，智能提示，emmet，错误提示，格式化，自动补全，debugger （command+shift+F，window 下为 alt+shift+F）

ESlint：自动格式化以符合.eslintrc.js文件中的规则

stylelint： 自动格式化以符合 .stylelint 文件中的规则

EditorConfig：**该插件尝试使用.editorconfig文件中的设置覆盖用户/工作区设置（跨编辑器）

koroFileHeader：添加文件头、函数注释

文件头部注释：window：ctrl+alt+i,mac：ctrl+cmd+i, linux: ctrl+meta+i

```
1  光标处函数注释：window：ctrl+alt+t,mac，ctrl+cmd+t,linux: ctrl+meta+t
```

如下图

(二) vscode 配置文件

文件->首选项->设置->工作区，输入@modified，并搜索

配置.vscode/settings.json

点击 **在** settings.json ****中编辑** 的选项后，你会发现当前的项目多了一个 **.vscode** 文件夹。只配置当前项目的设置的好处就在于，迁移项目或者在其他电脑下载项目后，只要新环境安装了相应的插件，你配置的内容就能生效（所以不选择全局配置，不然每一个新环境都要重新配置，并且不利于不同项目不同设置的管理）

示例（根据需要增删）：

```
{
  "editor.minimap.enabled": true,
  "editor.renderControlCharacters": false,
  "breadcrumbs.enabled": true,
  "editor.tabSize": 2,
  "fileheader.customMade": {
    "Description": "",
    "Date": "Do not edit",
    "LastEditTime": "Do not edit"
  },

  "vetur.format.defaultFormatter.html": "js-beautify-html", //格式化.vue中html
  "vetur.format.defaultFormatterOptions": {
  "js-beautify-html": {
    "wrap\_attributes": "force-aligned" //属性强制折行对齐
  }
},

  "search.exclude": {
    "\*\*/node\_modules": true,
    "\*\*/dist": true
  },

```

```
  "window.title":
"${dirty}${activeEditorMedium}${separator}${rootNam
e}",
  "editor.codeActionsOnSave": {
    // 文件保存时开启eslint自动修复程序
    "source.fixAll.eslint": true,
    // 文件保存时开启stylelint自动修复程序
    "source.fixAll.stylelint": true
  },
  "editor.wordWrap":  "on", // 自动换行
}
```

(三) 最终效果

- 项目参与人员使用相同的vscode工作区初始配置文件，可根据自身情况做相关调整（前提是使用vscode编辑器）
- 按下Ctrl+s，你会发现代码格式会自动根据规则格式化

小程序开发规范

一、数据部分

1.1、慎用 setData() 方法（C类问题）

1、每次 setData 都传递大量新数据

1 首屏 setData 太多屏数据，导致初始化渲染很慢

【推荐】：拆分首屏数据接口，将首屏内容接口和其他内容接口分开，非首屏接口延迟请求渲染

临时方案：在后端接口无法快速支持的情况下，可以暂时手动将第二次 setData 延迟 300ms，首次渲染控制在两屏以内

2 分页数据，整个大数组传入

【推荐】：使用二维数组，第一维对应每一页的数据，第二维对应当前页中的每一条数据，当分页数据需要插入时，只传递变化部分数据

```
text
wx.setData({[`list[${index}]`]: array })
```

2、频繁的去调用 setData

1 未变化的 data 仍然调用 setData 处理

【推荐】：setData 前先判断值是否变化，相当于业务逻辑中手动做了 diff

2 input、scroll 事件中频繁调用

【推荐】：在频繁触发的用户事件回调用进行节流、防抖处理，避免频繁调用 setData

3、后台态页面进行 setData（例如后台态页面定时器未清理）

【推荐】：在 onHide、onUnload 等生命周期内手动清理定时器

> 微信官方文档：
>
> 小程序的视图层目前使用 WebView 作为渲染载体，而逻辑层是由独立的 JavascriptCore 作为运行环境。在架构上，WebView 和 JavascriptCore 都是独立的模块，并不具备数据直接共享的通道。
>
> 当前，视图层和逻辑层的数据传输，实际上通过两边提供的 evaluateJavascript 所实现。即用户传输的数据，需要将其转换为字符串形式传递，同时把转换后的数据内容拼接成一份 JS 脚本，再通过执行 JS 脚本的形式传递到两边独立环境。而 evaluateJavascript 的执行会受很多方面的影响，数据到达视图层并不是实时的。

1.2、减少事件传递参数大小（C类问题）

原理同1.1，WXML元素上绑定的dataset数据（data-xxx），在事件响应中同样会涉及到一次从视图层到逻辑层的数据传输

【推荐】：dataset 数据需要严格控制大小，仅保留必要数据，避免将整个大对象直接传入

1.3、与页面渲染无关，在 wxml 不会用到的数据，放在自定义的 extraData 中维护（C类问题）

1、小程序启动时，会把所有调用 Page() 方法的 object 存在一个队列里。每次页面访问的时候，微信会重新创建一个新的对象实例，即深拷贝一个 page 对象。

【推荐】：应该尽量减少默认 data 的大小，以及减少 page 对象内的自定义属性

2、非视图数据变更，setData 引起不必要的重新生成 vdom、diff 和

通信等

【推荐】：将与视图渲染无关的数据置于自定义的 extraData 中

1.4、页面展示数据处理方法放在 wxs 里操作（B、C类问题）

> 微信官方文档：
>
> 分析：由于运行环境的差异，在 iOS 设备上小程序内的 WXS 会比 JavaScript 代码快 2 ~ 20 倍。在 android 设备上二者运行效率无差异。

【推荐】：一些与页面展示的不涉及复杂逻辑数据放在 wxs 里处理，例如时间格式化，数字格式化，商品名过滤等

1.5、交易金额等关键数据展示（A、B类问题）

前后端需要对金额类等重要数据接口的字段制定规范（例如：金额数据统一使用分为单位），避免维护者不知情直接使用或重复转换，造成展示错误

【推荐】：视图层可以统一使用规范展示模版，例如：`<template is="money" />`

1.6、页面 data 默认值设置为与渲染相关的初始数据（B、C类问题）

防止页面因默认数据不存在显示异常，页面中展示 undefined、NaN 等，用户体验较差

> 微信官方资料：

二、资源部分

2.1、小型 icon 图片使用雪碧图（C类问题）

使用雪碧图合并小图，减少图片资源发起过多 HTTP 请求

【推荐】：webpack 中可以使用 spritesmith 这样的插件去自动化生成雪碧图

2.2、图片优化（C类问题）

1 图片尺寸：与设计稿宽高保持 1:1，避免加载不必要的大资源

2 图片格式：透明底用 Png，否则用 Jpg，节约资源，或者选择性地使用 Webp 格式的图片

3 使用渐进式加载：首屏首次加载低质量图片，随后加载正常规格图片（但存在重复渲染的性能压力）

2.3、cdn 图片资源（B、C类问题）

1 图片上传前可通过 tinypng 这类工具进行压缩

2 图片资源上传到 cdn 上后可以设置所需的缓存策略

【推荐】：图片上传 cdn 可以有效地减小小程序包体积，并且方便及时替换活动图片。特例：主流程中稳定存在的 icon 图片无需上传

2.4、视频资源页面中只默认加载首屏图片（C类问题）

【推荐】：点击观看视频时才去加载真实视频资源，减少资源请求以提升页面初始渲染速度

三、容错部分

3.1、注意数组、对象取值空指针问题（A类问题）

【推荐】：使用 lodash.get、可选链操作符（@babel/plugin-proposal-optional-chaining）等方式，避免空指针异常

3.2、setStorage 失败容错（A类问题）

【推荐】：使用异步方法，在错误回调中进行容错处理

3.3、判断 API 可用性（A类问题）

【推荐】对有兼容性问题的API作检测（canIUse / 真值检测）、ES6方法兼容

四、样式部分

4.1、对原生组件样式、行为的特殊处理（A、B、C类问题）

诸如 video、canvas、textarea 等组件层级会比较高，会出现一些遮盖问题

【推荐】：需要用真机观察，可能针对场景需要做显示/隐藏的操作等

4.2、根结点设置样式（B类问题）

【推荐】：规范样式命名空间，避免样式渗透，避免污染其他节点样式

4.3、页面使用通用样式（B类问题）

【推荐】：抽离通用样式可以减少代码量，加快开发效率

4.4、宽高避免使用含小数点单位的样式（A类问题）

部分机型会丢失属性或者去掉小数点，导致与预期样式不一致

4.5、css样式（A类问题）

小程序兼容要求：Android >= 5.0.0, iOS >= 8

【推荐】：对于低版本系统手机（安卓5，IOS 8、9），需要做兼容处理，并针对该机型进行兼容性测试

五、组件部分

5.1、业务组件（B类问题）

业务组件主要指包含业务逻辑，有些也包括一些与后端接口通信的组件。当业务组件被主包业务依赖时，应将其归入主包中，便于复用，否则应该放在该业务所在的包中

5.2、UI组件（B类问题）

UI 组件主要指不包含任何业务相关逻辑，可以被轻松复用的组件。当一个 UI 模块有频率较高的复用场景，应该将其抽离出来作为一个 UI 组件放在主包中

六、性能部分

6.1、使用懒加载（C类问题）

频繁 setData，接口请求频繁、大图片资源是目前已知造成卡顿、白屏主要原因。

【推荐】：长页面的接口、图片进行懒加载，视图滚动到对应模块再请求对应接口和 setData 调用

6.2、分包和分包预加载（B、C类问题）

1 小程序除了一个主包，还可以包含任意数量分包。需要合理把页面放到对应包里面，做到分类管理

2 将业务主流程外的页面都放分包，减少主包大小，加快首屏加载

3 分包预加载可以在主包页面提前加载分包代码，用户进入分包会更快

6.3、业务迭代产生的无用代码清理（B、C类问题）

1）清理无用代码，减少包大小

2）无用代码会让项目越来越臃肿，无法维护。

3）好的代码清理方式，会重新组织代码结构，添加必要注视，让代码变得更加易于维护

6.4、减少get/setStorageSync使用，公共流程尽量不使用（C类问题）

get/setStorageSync 会与 bridge 通信，延迟较长，积少成多

【推荐】：若非必要，都使用get/setStorage异步加载。

七、请求部分

7.1、网络异常兜底处理（C类问题）

【推荐】：使用公共的页面状态组件来承接异常状态下的页面展示

7.2、所有服务接口按业务场景进行容错兜底处理（A、C类问题）

1 影响主流程的接口请求失败，触发页面状态组件错误状态展示

2 可隐藏模块的接口请求失败进行隐藏处理，不提示

3 不可隐藏模块的接口请求失败显示局部错误状态，或进行 toast 提示

4 登陆请求失败跳错误页面

7.3、减少http请求数量（C类问题）

1 要求服务端接口去除无用的数据返回，减少报文大小

2 减少 cdn 资源的请求，例如合并雪碧图

3 需要调用接口的组件要根据当前调用页是否有所需数据来判断是否需要组件内去调用接口

4 对于一些监听事件（如滚动、拖拽、输入等）引起的请求做函数节流

7.4、前后端边界（A、B、C类问题）

【推荐】遵循 “重渲染、轻逻辑” 的原则，复杂业务逻辑和计算尽量放在服务端，前端重点关注渲染以及性能体验

flutter开发规范

一、说明

(一) 使用背景

- 鉴于使用flutter框架开发APP开发统一化
- 开发需要统一的架构及规范
- 故在此进行架构及开发规范总结，后续继续补充

(二) 分支命名及使用规范

- 分支命名规范
  - 自己开发分支命名统一为 username ，如：zhangsan
  - 分支两条主线为 Master分支和feature分支
  - feature分支对应年月日（例如： feature/20220308）
  - Master作为发布分支，feature作为开发测试分支、自己开发分支从feature checkout出去，发布即 merge to master
- 分支合并规范
  - 从最新的feature分支checkout出自己的开发分支
  - 在自己开发开发分支开发完成后，先去feature分支pull最新代码，
  - 将feature 分支最新代码 merge 到自己分支，确保无冲突
  - 再切回feature分支merge自己开发分支代码，确保无冲突，且能正常运行
- Tag命名

  上线版本之后版本增加1 + 日期，

例如：1.0.1（20220101）

(三) commit 提交规范

- $git cz
- 用于说明 commit 的类别，只允许使用下面7个标识。
  - feat：新功能（feature）
  - fix：修补bug
  - docs：文档（documentation）
  - style： 格式（不影响代码运行的变动）
  - refactor：重构（即不是新增功能，也不是修改bug的代码变动）
  - test：增加测试
  - chore：构建过程或辅助工具的变动

(四) 代码规范

- 文件命名规范
  - 文件命名使用下划线命名法，如：hello_world
  - 请使用英文进行命名，不允许使用拼音。命名要求具有可读性，尽量避免使用缩写与数字
  - 未完待续
- 代码编码规范
  - 文件编码统一使用 UTF-8 编码；
  - 前端编码采用首字母小写驼峰法. Widget Class 必须采用首字母大写驼峰法.

(五) 文件目录结构(以LIb文件说明)

- lib
  - main.dart 入口文件
  - blocs 如果采用BLoC开发模式的话存放在此目录
  - config 基础配置目录
  - enums 枚举类存放目录
  - generated 数据模型解析中间层
  - models 存放模型, 不应该加入逻辑层

- net 网络基础层
- observer 生命周期监测层
- page 展示界面
- plugs 本地plugin 非远程加载
- routers 路由处理
- style 公共样式处理层
- utils 工具类
- viewmodel viewModel层
- widgets 页面封装层

```
├── main.dart //入口文件
├── blocs //采用BLoC或provider开发模式存放的目录
│   ├── bloc_provider.dart
│   └── xxx.dart
├── config //基础配置目录
│   └── xxx.dart
├── enums //枚举类存放目录
│   └── xxx.dart
├── generated //数据模型解析中间层
│   └── xxx.dart
├── models //本地存放模型, 不应该加入逻辑层
│   └── xx.dart
├── net //网络基础层
│   └── xx.dart
├── observer //生命周期监测层
│   └── xx.dart
├── plugs //本地plugin 非远程加载
│   └── xx.dart
├── style //公共样式处理层
│   └── xx.dart
├── routers //路由处理
│   └── routers.dart
├── utils //工具类
│   └── xxx.dart
├── viewmodel //viewModel层
│   └── xxx.dart
├── page //app展示界面
│   ├── home
│   │   ├── home.dart
│   │   └── xx.dart
```

```
31 │   └── xx.dart
32 └── widgets
33     └── ... //下面详细说明
```

二、代码风格

(一) 标识符三种类型

【强制】 1 大驼峰

类、枚举、typedef和类型参数

```
1  class SliderMenu { ... }
2
3  class HttpRequest { ... }
4
5  typedef Predicate = bool Function<T>(T value);
```

包括用于元数据注释的类

```
1  class Foo {
2    const Foo([arg]);
3  }
4
5  @Foo(anArg)
6  class A { ... }
7
8  @Foo()
9  class B { ... }
```

【强制】 2 使用小写加下划线来命名库和源文件

正例:

```
1  library peg_parser.source_scanner;
2
3  import 'file_system.dart';
4  import 'slider_menu.dart';
```

反例：

```
library pegparser.SourceScanner;

import 'file-system.dart';
import 'SliderMenu.dart';
```

【强制】3 使用小写加下划线来命名导入前缀

正例：

```
import 'dart:math' as math;
import
'package:angular_components/angular_components'
    as angular_components;
import 'package:js/js.dart' as js;
```

反例：

```
import 'dart:math' as Math;
import
'package:angular_components/angular_components'
    as angularComponents;
import 'package:js/js.dart' as JS;
```

【强制】4 使用小驼峰法命名其他标识符

正例：

```
var item;

HttpRequest httpRequest;

void align(bool clearItems) {
  // ...
}
```

5 【强制】优先使用小驼峰法作为常量命名

正例：

```
const pi = 3.14;
const defaultTimeout = 1000;
final urlScheme = RegExp('^([a-z]+):');

class Dice {
  static final numberGenerator = Random();
}
```

反例：

```
const PI = 3.14;
const DefaultTimeout = 1000;
final URL_SCHEME = RegExp('^([a-z]+):');

class Dice {
  static final NUMBER_GENERATOR = Random();
}
```

【推荐】6 不使用前缀字母

因为Dart可以告诉您声明的类型、范围、可变性和其他属性，所以没有理由将这些属性编码为标识符名称。

正例：

```
defaultTimeout
```

反例：

```
kDefaultTimeout
```

(二) 排序

为了使你的文件前言保持整洁，我们有规定的命令，指示应该出现在

其中。每个"部分"应该用空行分隔。

【推荐】1 在其他引入之前引入所需的dart库

```
import 'dart:async';
import 'dart:html';

import 'package:bar/bar.dart';
import 'package:foo/foo.dart';
```

【推荐】2 在相对引入之前先引入在包中的库

```
import 'package:bar/bar.dart';
import 'package:foo/foo.dart';

import 'util.dart';
```

【推荐】3 第三方包的导入先于其他包

```
import 'package:bar/bar.dart';
import 'package:foo/foo.dart';

import 'package:my_package/util.dart';
```

【推荐】4 在所有导入之后，在单独的部分中指定导出

正例：

```
import 'src/error.dart';
import 'src/foo_bar.dart';

export 'src/error.dart';
```

反例：

```
import 'src/error.dart';
export 'src/error.dart';
```

```
3    import 'src/foo_bar.dart';
```

(三) 所有流控制结构，请使用大括号

【推荐】这样做可以避免悬浮的else问题

正例：

```
1    if (isWeekDay) {
2      print('Bike to work!');
3    } else {
4      print('Go dancing or read a book!');
5    }
```

例外

一个if语句没有else子句，其中整个if语句和then主体都适合一行。在这种情况下，如果你喜欢的话，你可以去掉大括号

```
1    if (arg == null) return defaultValue;
```

如果流程体超出了一行需要分划请使用大括号：

```
1    if (overflowChars != other.overflowChars) {
2      return overflowChars < other.overflowChars;
3    }
```

反例：

```
1    if (overflowChars != other.overflowChars)
2      return overflowChars < other.overflowChars;
```

三、注释

(一) 要像句子一样格式化

【参考】除非是区分大小写的标识符，否则第一个单词要大写。以句号结尾(或“!”或“?”)。对于所有的注释都是如此：doc注释、内联内容，甚至TODOs。即使是一个句子片段。

正例：

```
greet(name) {
  // Assume we have a valid name.
  print('Hi, $name!');
}
```

反例：

```
greet(name) {
  /* Assume we have a valid name. */
  print('Hi, $name!');
}
```

可以使用块注释(/.../)临时注释掉一段代码，但是所有其他注释都应该使用//

(二) Doc注释

【参考】使用///文档注释来记录成员和类型。

使用doc注释而不是常规注释，可以让dartdoc找到并生成文档。

```
/// The number of characters in this chunk when unsplit.
int get length => ...
```

> 由于历史原因，达特茅斯学院支持道格评论的两种语法:///(“C#风格”)和/**...* /(“JavaDoc风格”)。我们更喜欢/// 因为它更紧凑。/和/*在多行文档注释中添加两个无内容的行。在某些情况下，///语法也更容易阅读，例如文档注释包含使用*标记列表项的项目符号列表。

(三) 考虑为私有api编写文档注释

【参考】Doc注释并不仅仅针对库的公共API的外部使用者。它们还有

助于理解从库的其他部分调用的私有成员
1 用一句话总结开始doc注释

以简短的、以用户为中心的描述开始你的文档注释，以句号结尾。

正例：

```
/// Deletes the file at [path] from the file system.
void delete(String path) {
  ...
}
```

反例：

```
  /// Depending on the state of the file system and the user's permissions,
  /// certain operations may or may not be possible. If there is no file at
  /// [path] or it can't be accessed, this function throws either [IOError]
  /// or [PermissionError], respectively. Otherwise, this deletes the file.
  void delete(String path) {
    ...
  }
```

2 “doc注释”的第一句话分隔成自己的段落

在第一个句子之后添加一个空行，把它分成自己的段落

```
  /// Deletes the file at [path].
  ///
  /// Throws an [IOError] if the file could not be found. Throws a
  /// [PermissionError] if the file is present but could not be deleted.
  void delete(String path) {
    ...
  }
```

四、使用参考

(一) 库的引用

【推荐】导入lib下文件库，统一指定包名，避免过多的 `../../`

```
package:flutter_packages/
```

(二) 字符串的使用

【推荐】1 使用相邻字符串连接字符串文字

如果有两个字符串字面值(不是值，而是实际引用的字面值)，则不需要使用+连接它们。就像在C和c++中，简单地把它们放在一起就能做到。这是创建一个长字符串很好的方法但是不适用于单独一行。

正例：

```
raiseAlarm(
    'ERROR: Parts of the spaceship are on fire. Other '
    'parts are overrun by martians. Unclear which are which.');
```

反例:

```
raiseAlarm('ERROR: Parts of the spaceship are on fire. Other ' +
    'parts are overrun by martians. Unclear which are which.');
```

【推荐】2 优先使用模板字符串

```
'Hello, $name! You are ${year - birth} years old.';
```

【推荐】3 在不需要的时候，避免使用花括号

```
正例:
  'Hi, $name!'
  "Wear your wildest $decade's outfit."

反例1:
  'Hello, ' + name + '! You are ' + (year - birth).toString() + ' y...';

反例2:
  'Hi, ${name}!'
  "Wear your wildest ${decade}'s outfit."
```

(三) 集合

【推荐】1 尽可能使用集合字面量

如果要创建一个不可增长的列表，或者其他一些自定义集合类型，那么无论如何，都要使用构造函数。

```
正例:
  var points = [];
  var addresses = {};
  var lines = <Lines>[];

反例:
  var points = List();
  var addresses = Map();
```

【推荐】2 不要使用.length查看集合是否为空

```
正例:
if (lunchBox.isEmpty) return 'so hungry...';
if (words.isNotEmpty) return words.join(' ');

反例:
  if (lunchBox.length == 0) return 'so hungry...';
  if (!words.isEmpty) return words.join(' ');
```

【推荐】3 考虑使用高阶方法转换序列

如果有一个集合，并且希望从中生成一个新的修改后的集合，那么使用.map()、.where()和Iterable上的其他方便的方法通常更短，也更具有声明性

```
  var aquaticNames = animals
      .where((animal) => animal.isAquatic)
      .map((animal) => animal.name);
```

【推荐】4 避免使用带有函数字面量的Iterable.forEach()

在Dart中，如果你想遍历一个序列，惯用的方法是使用循环。

```
正例:
for (var person in people) {
  ...
}

反例:
  people.forEach((person) {
    ...
  });
```

【推荐】5 不要使用List.from()，除非打算更改结果的类型

给定一个迭代，有两种明显的方法可以生成包含相同元素的新列表

```
var copy1 = iterable.toList();
var copy2 = List.from(iterable);
```

明显的区别是第一个比较短。重要的区别是第一个保留了原始对象的类型参数

```
// Creates a List<int>:
var iterable = [1, 2, 3];

// Prints "List<int>":
print(iterable.toList().runtimeType);
```

```
// Creates a List<int>:
var iterable = [1, 2, 3];

// Prints "List<dynamic>":
print(List.from(iterable).runtimeType);
```

(四) 参数的使用

【推荐】1 使用=将命名参数与其默认值分割开

由于遗留原因，Dart均允许":"和"="作为指定参数的默认值分隔符。为了与可选的位置参数保持一致，使用"="。

```
正例:

  void insert(Object item, {int at = 0}) { ... }

反例:
  void insert(Object item, {int at: 0}) { ... }
```

【推荐】2 不要使用显式默认值null

如果参数是可选的，但没有给它一个默认值，则语言隐式地使用null作为默认值，因此不需要编写它

```
正例:

void error([String message]) {
  stderr.write(message ?? '\n');
}

反例:

void error([String message = null]) {
  stderr.write(message ?? '\n');
}
```

(五) 变量

【强制】1 不要显式地将变量初始化为空

在Dart中，未显式初始化的变量或字段自动被初始化为null。不要多余赋值null

```
正例:
  int _nextId;

  class LazyId {
    int _id;

    int get id {
      if (_nextId == null) _nextId = 0;
      if (_id == null) _id = _nextId++;

      return _id;
    }
  }
```

反例:

```
  int _nextId = null;

  class LazyId {
    int _id = null;

    int get id {
      if (_nextId == null) _nextId = 0;
      if (_id == null) _id = _nextId++;

      return _id;
    }
  }
```

【推荐】2 避免储存你能计算的东西

在设计类时，您通常希望将多个视图公开到相同的底层状态。通常你会看到在构造函数中计算所有视图的代码，然后存储它们:

反例:

```
class Circle {
  num radius;
  num area;
  num circumference;

  Circle(num radius)
      : radius = radius,
        area = pi * radius * radius,
        circumference = pi * 2.0 * radius;
}
```

如上代码问题：

- 浪费内存
- 缓存的问题是无效——如何知道何时缓存过期需要重新计算？

正例：

```
class Circle {
  num radius;

  Circle(this.radius);

  num get area ⇒ pi * radius * radius;
  num get circumference ⇒ pi * 2.0 * radius;
}
```

(六) 类成员

【推荐】1 不要把不必要地将字段包装在getter和setter中

```
反例：
class Box {
  var _contents;
  get contents ⇒ _contents;
  set contents(value) {
    _contents = value;
  }
}
```

【推荐】2 优先使用final字段来创建只读属性

尤其对于 `StatelessWidget`

【推荐】3 在不需要的时候不要用this

反例：

```
class Box {
  var value;

  void clear() {
    this.update(null);
  }

  void update(value) {
    this.value = value;
  }
}
```

正例：

```
class Box {
  var value;

  void clear() {
    update(null);
  }

  void update(value) {
    this.value = value;
  }
}
```

(七) 构造函数

【推荐】1 尽可能使用初始化的形式

反例：

```
class Point {
  num x, y;
  Point(num x, num y) {
    this.x = x;
    this.y = y;
  }
}
```

正例：

```
class Point {
  num x, y;
  Point(this.x, this.y);
}
```

【推荐】2 不要使用new

Dart2使new 关键字可选

正例：

```
Widget build(BuildContext context) {
  return Row(
    children: [
      RaisedButton(
        child: Text('Increment'),
      ),
      Text('Click!'),
    ],
  );
}
```

反例：

```
Widget build(BuildContext context) {
  return new Row(
    children: [
      new RaisedButton(
```

```
          child: new Text('Increment'),
        ),
        new Text('Click!'),
      ],
    );
  }
```

(八) 异步

【推荐】1 优先使用async/await代替原始的futures

async/await语法提高了可读性，允许你在异步代码中使用所有Dart控制流结构。

```
  Future<int> countActivePlayers(String teamName) async {
    try {
      var team = await downloadTeam(teamName);
      if (team == null) return 0;

      var players = await team.roster;
      return players.where((player) => player.isActive).length;
    } catch (e) {
      log.error(e);
      return 0;
    }
  }
```

【推荐】2 当异步没有任何用处时，不要使用它

如果可以在不改变函数行为的情况下省略异步，那么就这样做。、

正例：

```
  Future afterTwoThings(Future first, Future second) {
    return Future.wait([first, second]);
  }
```

反例：

```
  Future afterTwoThings(Future first, Future second) async {
    return Future.wait([first, second]);
  }
```

五、空安全

【强制】(一) 如果一个对象不确定是否初始化尽量不要用late 用?

例如

```
User? user;

//接口获取处理：
user = User.jsonFrom(json);

//调用
String? name = user?.name;
Text(user.name!);
```

【强制】(二) 对于json数据解析，尽量不要直接强转

```
反例：
userFromJson( Map<String, dynamic> json){
    User user = User();
    user.list = []..addAll((map['list'] as List ?? []).map((o) => ListModel.fromMap(o)));
    user.name = json['name'];
}

正例：
userFromJson( Map<String, dynamic> json){
    User user = User();
    user.list = json['list'] == null ? [] []..addAll((json['list'] as List).map((o) => ListModel.fromMap(o)));
```

```
    user.name = json['name'];
}
```

【推荐】(三) late 用法 在调用的时候才加载，不调用不加载

```
late User user = User();
```

【推荐】(四) late 尽量少用 如果没有确定初始化就调用的地方尽量别用

```
late ExampleModel model = ExampleModel();

ExampleModel? model;

```

【强制】(五)网络请求json数据解析 尽量用 ?标识符

```
setMyName(name = json["name"]);

setMyName(String? name){
}

```

【推荐】(六)bool类型的属性 在做判断的时间 尽量 做空校验

```
if(item.sysGroupLabelFlag ?? false) {}

```

【强制】(七)var尽量少用 如果用var 下边一定要有类型判断

```
var list = json["list"];
if(list is List) return;


```