

# 为什么要有Code Review?

1. 自己写代码不太能够注意到问题
2. 团队里有人踩过的坑就不要再踩了
3. 团队成员间相互交流学习的机会，学习好的写法，警惕不好的写法
4. 督促团队成员有意识地提高代码质量，总结出checklist，自我对代码检查

# 欢迎大家来找茬，看看代码哪里有问题？

```
// 页面传递参数
let params = {
  junction_ids: junc_id,
  task_id: this.current_task.task_id,
  map_version: this.adj_result.map_version,
  time_point: this.timeRange.time_point,
  dates: this.current_task.dates,
};
params = encodeURIComponent(JSON.stringify(params));
href = `#/coordinate-detail?params=${params}`;
```

```
<router-link v-else target="_blank" :to="{name: 'assessment-detail', query: detailQuery}">
  <t-button type="primary">详情对比</t-button>
</router-link>
</div>
```

屏蔽了不同路由模式的差别，任意切换路由模式  
更高层次的抽象意味着可以应付更容易变化的需求

面向接口编程，而不是面向实现编程  
不同的实现有着相同的接口，那么可以方便的进行  
替换，即为里氏替换原则

<https://blog.csdn.net/zhengzhb/article/details/7281833>

```
// 更新SOURCE
updateSource(logic_junction_id, properties) {
  const source = this.map.getSource('marker_source');
  if (!source) return;
  const source_data = source._data;
  const feature = source_data.features.find(item => item.properties.junction.logic_junction_id === logic_junction_id);
  if (!feature) return;
  feature.properties = { ...feature.properties, ...properties };
  source.setData(source_data);
},
```

```
// 更新source
updateSource(logic_junction_id, properties) {
  if (!this.markers || !this.markers.features) return;
  const features = this.markers.features.map(feature => {
    if (feature.properties.junction.logic_junction_id === logic_junction_id) {
      return {
        ...feature,
        properties: { ...feature.properties, ...properties },
      };
    } else {
      return {
        ...feature,
        properties: { ...feature.properties, selected: false },
      };
    }
  });
  this.markers = { ...this.markers, features };
},
```

同理didi-map组件已经屏蔽了 mapbox一些细节，直接用就好，同时也更符合数据驱动的思想。

```

},
async queryTrendOption() {
  const result = await this.querySevenDaysAlarmChange(this.panelParams);
  this.trend_options = this.getTrendOption(result, 'alarm', this.theme);
  this.trend_loading = false;
},
async queryAlarmInfo() {

```

```

const result = await api.getQuotaTrend(payload, {
  loading: true,
  name: 'getQuotaTrend',
});

```

```

export function startLoading(options) {
  options.loading && store.commit(types.LOADING, { show: options.loading, name: options.name });
}

//终止loading动画
export function endLoading(options) {
  options.loading && setTimeout(() => store.commit && store.commit(types.LOADING, { show: false, name: options.name }), 500);
}

```

重复三次以上的代码应该思考下抽象处理  
拦截所有的请求，统一做loading的处理  
一些业务无关的操作可以统一抽离出来  
拦截器和过滤器是AOP(面向切面编程)常见的应用场景

比如vue中一些通用的逻辑，同样可以抽到mixin中统一处理

## dva-loading

npm v2.0.5 build failing coverage 100% downloads 20k/m

Auto loading plugin for dva. 🙌 You don't need to write `showLoading` and `hideLoading` any more.

什么时候要用到面向切面AOP呢？

举个例子，你想给你的网站加上鉴权。

对某些url，你认为不需要鉴权就可以访问，

对于某些url，你认为需要有特定权限的用户才能访问

如果你依然使用OOP，面向对象，

那你只能在那里url对应的Controller代码里面，一个一个写上鉴权的代码

而如果你使用了AOP呢？

那就像使用Spring Security进行安全管理一样简单（更新：Spring Security的拦截是基于Servlet的Filter的，不是aop，不过两者在使用方式上类似）：

```
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .authorizeRequests()  
            .antMatchers("/static", "/register").permitAll()  
            .antMatchers("/user/**").hasRoles("USER", "ADMIN")
```

这样的做法，对原有代码毫无入侵性，这就是AOP的好处了，把和主业务无关的事情，放到代码外面去做。

所以当你下次发现某一行代码经常在你的Controller里出现，比如方法入口日志打印，那就要考虑使用AOP来精简你的代码了。

聊完了AOP是啥，现在再来聊聊实现原理。

AOP像OOP一样，只是一种编程范式，AOP并没有规定说，实现AOP协议的代码，要用什么方式去实现。

<https://www.zhihu.com/question/24863332>



```
if (val) {
  const junctionsList = this.config_type === 'road' ? val.junctions_info : val.junction_list;
  if (!junctionsList) this.selected_path = [];
  const centerMap = this.config_type === 'road' ? [+val.center.lng, +val.center.lat] : [+val.center_lng, +val.center_lat];
  junctionsList.forEach((item, index) => {
    const center = [item.lng, item.lat];
    let flag = '';
    if (this.config_type === 'road') {
      if (index === 0) {
        flag = 'start';
      } else if (index === junctionsList.length - 1 && junctionsList.length !== 1) {
        flag = 'end';
      }
    }
    const markerTemp = {
      point: center,
      property: { junction: item, selected: true, flag: flag },
    };
    this.markers.showMarker.push({ ...markerTemp });
    this.markers.clickMarker.push({ ...markerTemp });
  });
}
```



尽量用filter map reduce，减少过程式的编码。

函数式通常描述做了什么，过程式你得一步一步看干了什么（函数式编程关心数据的映射，命令式编程关心解决问题的步骤）

就算没有经过任何优化， $O(xn)$  的复杂度等价于  $O(n)$ ，但可读性大大提高



```
var collection = [
  { name: 'Janice', age: 38, gender: 'f' },
  { name: 'Joey', age: 20, gender: 'm' },
  { name: 'Lauren', gender: 'f' },
  { name: 'Drew', gender: 'm' }
];

_(collection)
  .where({ gender: 'f' })
  .filter(_._flow(_._property('age'), _._isFinite))
  .value();

// → [ { name: "Janice", age: 38, gender: "f" } ]
```



```
var collection = [
  { gender: 'f', dob: new Date(1984, 3, 8) },
  { gender: 'm', dob: new Date(1983, 7, 16) },
  { gender: 'f', dob: new Date(1987, 2, 4) },
  { gender: 'm', dob: new Date(1988, 5, 2) }
];

var result = _(collection)
  .where({gender: 'm'})
  .pluck('dob')
  .map(function(item){
    return item.toLocaleString();
  })
  .value();

//[ '1983-08-16 00:00:00', '1988-06-02 01:00:00' ]
console.log(result);
```



# 再延伸一下？

transduce  
compose  
惰性计算

```
var result = _(source).map(func1).map(func2).map(func3).value();
```

```
var result = [];  
for(var i = 0; i < source.length; i++) {  
    result[i] = func3(func2(func1(source[i])));  
}
```

<https://github.com/cognitect-labs/transducers-js>

<https://juejin.im/post/5b5a9451f265da0f6a036346>

<https://segmentfault.com/a/11900000006998998>

```
,  
  computed: {  
    mapHeight() {  
      return `${document.body.clientHeight - 220}px`;  
    },  
  },  
  methods: {}  
}
```

1. 不在 vue 掌控范围内的用 computed, 是不会有依赖推导的效果, 容易起到误导作用
2. 能用html和css解决尽量用html和css解决
  - 2.1 浏览器会有性能优化
  - 2.2 html和css解决起来通常会更加简单, 简单就意味着更好维护

```
return {  
    //当前页面是指标页还是诊断页, 1: 指标详情页 2: 诊断详情页、概览页  
    type: 1,  
    queryModel: {  
        //指标计算类型 1: 综合 0: 时间点  
        type: 0,  
        //...  
    }  
}
```

更偏向于使用有实际意义的映射或字符串来表示状态，而不是数字。

0xx 1xx 2xx -> FINISHED PENDING FAILED 反正最终都会被编译压缩，体积和性能上两者没有区别，但后者明显可读性更好

C编程风格上留下的糟粕

```
<template>
<div class="report">
  <report-list :type="type" class="report-item"></report-list>
  <report-config :type="type" class="report-item"></report-config>
</div>
</template>
```

```
export default {
  name: 'ReportList',
  mixins: [mxin],
  props: ['type'],
  data() {
    return {
      reports: [],
    };
  },
  methods: {
    ...mapActions({
      queryReports: types.QUERY_REPORTS,
    }),
    async query() {
      const { list } = await this.queryReports({
        city_id: this.current_city.code,
        type: this.type,
        page_size: 5,
        page_no: 1,
      });
      this.reports = list;
    },
  },
  created() {
    this.query();
  },
}
```

不要滥用mixin，组合优于继承

1、mixins 带来了隐式依赖

2、mixins 与 mixins 之间，mixins 与组件之间容易导致命名冲突，比如说定义了同名的method或者data属性

3、由于 mixins 是侵入式的，它改变了原组件，所以修改 mixins 等于修改原组件，随着需求的增长 mixins 将变得复杂，导致滚雪球的复杂性。



# vue如何实现类似react 的hoc和 render props功能

hoc: <http://hcysun.me/vue-design/more/vue-hoc.html>

render props:slot-scope

为什么要有这些东西呢?

有一些逻辑是通用，但UI是容易变化的

```
<section>
  <section class="clock-1">
    <h2>时钟1</h2>
    <clock></clock>
  </section>

  <section class="clock-2">
    <h2>时钟2</h2>
    <clock>
      <div slot-scope="{date, weekDay}">
        <p>{{weekDay}} </p>
        <p>{{+date}}</p>
      </div>
    </clock>
  </section>
</section>
</template>

<script>
/** @format */

export default {
  name: 'ClockDemo',
  description: '前端的时钟，能够根据初始传入的时间进行时间修正',
};
</script>
```

```
props: {
  startTimestamp: {
    type: Number,
  },
  tick: {
    type: Number,
    default: 1000,
    validator(value) {
      return value;
    },
  },
  showWeekDay: {
    type: Boolean,
    default: true,
  },
  showTime: {
    type: Boolean,
    default: true,
  },
  showDate: {
    type: Boolean,
    default: true,
  },
},
```

```
    },  
    item.quota_key.forEach(quota_key => {  
      params.push({  
        ...this.baseParams,  
        ...quota_param,  
        quota_key,  
      });  
  
      const time_type = TIME_TYPE.find(type => type.key === item.time_type);  
      this.tops.push({  
        quota_title: '',  
        quota_name: '',  
        quota_desc: '',  
        summary: '',  
        loading: true,  
        columns: [],  
        data: [],  
      });  
    });  
  });  
});
```

上面代码的复杂度是多少？

完蛋了  $O(n^2)$ , 万一后端给我返回了1000条数据, 循环次数最坏可能是  $1000 * 1000 = 1000000$  次 = 100W 次

```
const list = [{id: 1, name: 'xx'}, {id: 2, name: 'yy'}],
const listMap = list.reduce((pre, curr) => {
  return {...pre, [curr.id]: curr}
}, {})

item.quota_key.forEach(quota_key => {
  params.push({
    ...this.baseParams,
    ...quota_param,
    quota_key,
  });
  const time_type = listMap[key]
  this.tops.push({
    quota_title: '',
    quota_name: '',
    quota_desc: '',
    summary: '',
    loading: true,
    columns: [],
    data: [],
  });
});
```

$O(n)$

$O(n)$

复杂度变为 $O(n)$ 了，完美

<https://github.com/paularmstrong/normalizr>

```
{
  "id": "123",
  "author": {
    "id": "1",
    "name": "Paul"
  },
  "title": "My awesome blog post",
  "comments": [
    {
      "id": "324",
      "commenter": {
        "id": "2",
        "name": "Nicole"
      }
    }
  ]
}
```



Now, `normalizedData` will be:

```
{
  result: "123",
  entities: {
    "articles": {
      "123": {
        id: "123",
        author: "1",
        title: "My awesome blog post",
        comments: [ "324" ]
      }
    },
    "users": {
      "1": { "id": "1", "name": "Paul" },
      "2": { "id": "2", "name": "Nicole" }
    },
    "comments": {
      "324": { id: "324", "commenter": "2" }
    }
  }
}
```

有点类似后端数据库的感觉了

经常遇到的一些隐含交互逻辑的清单，测试不一定能覆盖，主要供开发和codereview时参考

1. 对object和array的取值，是否安全;和外部打交道的部分默认值使用 `const x = data || []` ,js 业务中可使用解构 `var {x = []} = result`
2. 接口通信时，是否有loading、错误处理、超时处理
3. 通用字体、颜色、间距等是否复用
4. 独立业务之外的交互组件，是否存在复用，或是否封装，并提供注释
5. 需要引入工具函数时，检查是否存在复用
6. 业务组件存在复用的
7. 图标是否导出为svg并上传到iconfont
8. 页面引入的图片大小是否超过100KB
9. 点击区域是否过小，能否满足至少32px矩形
10. 元素是否需要touch后的active反馈
11. 标题等定高元素是否存在折行可能，切换语言后布局有无影响
12. html 中书写的 script 标签绝对不能自闭合！！
13. 分享出去的链接必须统一做 encodeURIComponent处理，甚至可以做短连接优化！否则分享到聊天软件中可能被截断
14. IE10以上无法使用 `lte ie 11` 这样的形式探测！pollyfil 使用属性探测添加即可
15. 长时间轮询定时器可能会受电脑休眠等因素影响，不建议使用 参照
16. 所有的数组和对象返回值为空的时候不要改变原有类型，返回空数组和空对象。
17. 状态是怎么放？放在 vuex 里还是 vue 里？业务状态通常要被各个组件消费，所以推荐放在 store 中；UI状态通常只是临时状态，不需要被共享，所以放在组件内部消费即可。还有些状态只会被组件自身消费，放在组件内部即可，比如一个搜索组件，search-suggest。如果组件 destroy 之后状态需要保留，最好还是放在 store 中。
18. 通用组件不开启scoped，业务组件全部开启scoped
19. 开闭原则、依赖注入、显式优于隐式、统一配置，一处修改，用到的地方都生效，业务的常量都定义在 constant 中
20. 状态是怎么放？放在 vuex 里还是 vue 里？业务状态通常要被各个组件消费，所以推荐放在 store 中；UI状态通常只是临时状态，不需要被共享，所以放在组件内部消费即可。还有些状态只会被组件自身消费，放在组件内部即可，比如一个搜索组件，search-suggest。如果组件 destroy 之后状态需要保留，最好还是放在 store 中。
21. 慎用 watch，watch有可能会死循环
22. 所有的异常应该继续往下抛

有时候会困惑，好像做的东西没  
什么技术难度

能用简单的代码实现复杂的业务，  
已经是很大的技术难度了