**[浅析]特定场景下取代if-else和switch的方案**

世界那么大，景点那么多。有些时候，换个方式，换个角度，换个陪同，都会有不一样感觉与收获。写代码也亦如此。

**1.前言**

相信很多人有这样的经历，在项目比较忙的时候，都是先考虑实现，用当时以为最好的方式先实现方案，在项目不忙的时候，再看下以前代码，想下有什么更好的实现方案，或者优化方案。笔者也不例外，下面就和读者们分享一下自己最近在特定场合下，代替if-else，switch的解决方案。

**2.look-up表代替if-else**

比如大家可能会遇到类似下面的需求：比如某平台的信用分数评级，超过700-950，就是信用极好，650-700信用优秀，600-650信用良好，550-600信用中等，350-550信用较差。

实现很简单

**function** showGrace(grace) {

let \_level='';

**if**(grace>=700){

\_level='信用极好'

}

**else** **if**(grace>=650){

\_level='信用优秀'

}

**else** **if**(grace>=600){

\_level='信用良好'

}

**else** **if**(grace>=550){

\_level='信用中等'

}

**else**{

\_level='信用较差'

}

return \_level;

}

运行也没问题，但是问题也是有

1.万一以后需求，改了比如650-750是信用优秀，750-950是信用极好。这样就整个方法要改。

2.方法存在各种神仙数字：700，650，600，550。日后的维护可能存在问题。

3.if-else太多，看着有点强迫症

所以下面用look-up表，把配数据置和业务逻辑分离的方式实现下

**function** showGrace(grace) {

let graceForLevel=[700,650,600,550];

let levelText=['信用极好','信用优秀','信用良好','信用中等','信用较差'];

**for**(let i=0;i<graceForLevel.length;i++){

**if**(grace>=graceForLevel[i]){

return levelText[i];

}

}

//如果不存在，那么就是分数很低，返回最后一个

return levelText[levelText.length-1];

}

这样的修改，优点就是如果有需求修改，只需要修改graceForLevel，levelText。业务逻辑不需要改。

为什么这里推荐配数据置和业务逻辑分离

1.修改配置数据比业务逻辑修改成本更小，风险更低

2.配置数据来源和修改都可以很灵活

3.荐配置和业务逻辑分离，可以更快的找到需要修改的代码

如果还想灵活一些，可以封装一个稍微通用一点的look-up函数。

**function** showGrace(grace,level,levelForGrace) {

**for**(let i=0;i<level.length;i++){

**if**(grace>=level[i]){

return levelForGrace[i];

}

}

//如果不存在，那么就是分数很低，返回最后一个

return levelForGrace[levelText.length-1];

}

let graceForLevel=[700,650,600,550];

let levelText=['信用极好','信用优秀','信用良好','信用中等','信用较差'];

使用推荐配置数据和业务逻辑分离形式开发，还有一个好处，在上面例子没体现出来，下面简单说下。比如输入一个景点，给出景点所在的城市。

**function** getCityForScenic(scenic) {

let \_city=''

**if**(scenic==='广州塔'){

\_city='广州'

}

**else** **if**(scenic==='西湖'){

\_city='杭州'

}

return \_city;

}

输入广州塔，就返回广州。输入西湖就返回杭州。但是一个城市不止一个景点，那么有人习惯这样写。

**if**(scenic==='广州塔'||scenic==='花城广场'||scenic==='白云山'){

\_city='广州'

}

如果景点很多，数据很长，看着难受，有些人喜欢这样写

let scenicOfHangZhou=['西湖','湘湖','砂之船生活广场','京杭大运河','南宋御街']

**if**(~scenicOfHangZhou.indexOf(scenic)){

\_city='杭州'

}

这样执行没错，但是写出来的代码可能像下面这样，**风格不统一**。

**function** getCityForScenic(scenic) {

let \_city='';

let scenicOfHangZhou=['西湖','湘湖','砂之船生活广场','京杭大运河','南宋御街'];

**if**(scenic==='广州塔'||scenic==='花城广场'||scenic==='白云山'){

\_city='广州'

}

**else** **if**(~scenicOfHangZhou.indexOf(scenic)){

\_city='杭州'

}

return \_city;

}

即使用switch，也有可能出现这样的情况

**function** getCityForScenic(scenic) {

let \_city='';

let scenicOfHangZhou=['西湖','湘湖','砂之船生活广场','京杭大运河','南宋御街'];

switch(true){

**case** (scenic==='广州塔'||scenic==='花城广场'||scenic==='白云山'):\_city='广州';break;

**case** (!!~scenicOfHangZhou.indexOf(scenic)):return '杭州';

}

return \_city;

}

虽然上面的代码出现的概率很小，但毕竟会出现。这样的代码可能会造成日后维看得眼花缭乱。如果使用了配置数据和业务逻辑分离，那就可以避免这个问题。

**function** getCityForScenic(scenic) {

let cityConfig={

'广州塔':'广州',

'花城广场':'广州',

'白云山':'广州',

'西湖':'杭州',

'湘湖':'杭州',

'京杭大运河':'杭州',

'砂之船生活广场':'杭州',

'南宋御街':'杭州',

}

return cityConfig[scenic];

}

有些人不习惯对象的 key 名是中文。也可以灵活处理

**function** getCityForScenic(scenic) {

let cityConfig=[

{

scenic:'广州塔',

city:'广州'

},

{

scenic:'花城广场',

city:'广州'

},

{

scenic:'白云山',

city:'广州'

},

{

scenic:'西湖',

city:'杭州'

},

{

scenic:'湘湖',

city:'杭州'

},

{

scenic:'京杭大运河',

city:'杭州'

},

{

scenic:'砂之船生活广场',

city:'杭州'

}

]

**for**(let i=0;i<cityConfig.length;i++){

**if**(cityConfig[i].scenic===scenic){

return cityConfig[i].city

}

}

}

这样一来，如果以后要加什么景点，对应什么城市，只能修改上面的cityConfig，业务逻辑不需要改，也不能改。代码风格上面就做到了统一。

**这里简单总结下，使用配置数据和业务逻辑分离的形式，好处**

1. 修改配置数据比业务逻辑修改成本更小，风险更低
2. 配置数据来源和修改都可以很灵活
3. 配置和业务逻辑分离，可以更快的找到需要修改的代码
4. 配置数据和业务逻辑可以让代码风格统一

但是并不是所有的if-else都建议这样改造，有些需求不建议使用look-up改造。比如if-else不是很多，if判断的逻辑不统一的使用，还是建议使用if-else方式实现。但是神仙数字，要清除。

比如下面这个根绝传入时间戳，显示评论时间显示的需求，

**发布1小时以内的评论：x分钟前**

**发布1小时~24小时的评论：x小时前**

**发布24小时~30天的评论：x天前**

**发布30天以上的评论：月/日**

**去年发布并且超过30天的评论：年/月/日**

实现不难，几个if-else就行了

**function** formatDate(timeStr){

//获取当前时间戳

let \_now=+new Date();

//求与当前的时间差

let se=\_now-timeStr;

let \_text='';

//去年

**if**(new Date(timeStr).getFullYear()!==new Date().getFullYear()&&se>2592000000){

\_text=new Date(timeStr).getFullYear()+'年'+(new Date(timeStr).getMonth()+1)+'月'+new Date(timeStr).getDate()+'日';

}

//30天以上

**else** **if**(se>2592000000){

\_text=(new Date(timeStr).getMonth()+1)+'月'+new Date(timeStr).getDate()+'日';

}

//一天以上

**else** **if**(se>86400000){

\_text=Math.floor(se/86400000)+'天前';

}

//一个小时以上

**else** **if**(se>3600000){

\_text=Math.floor(se/3600000)+'小时前';

}

//一个小时以内

**else**{

//如果小于1分钟，就显示1分钟前

**if**(se<60000){se=60000}

\_text=Math.floor(se/60000)+'分钟前';

}

return \_text;

}

运行结果没问题，但是也存在一个问题，就是这个需求有神仙数字：2592000000，86400000，3600000，60000。对于后面维护而言，一开始可能并不知道这个数字是什么东西。

所以下面就消灭神仙数字，常量化

**function** formatDate(timeStr){

//获取当前时间戳

let \_now=+new Date();

//求与当前的时间差

let se=\_now-timeStr;

const DATE\_LEVEL={

month:2592000000,

day:86400000,

hour:3600000,

minter:60000,

}

let \_text='';

//去年

**if**(new Date(timeStr).getFullYear()!==new Date().getFullYear()&&se>DATE\_LEVEL.month){

\_text=new Date(timeStr).getFullYear()+'年'+(new Date(timeStr).getMonth()+1)+'月'+new Date(timeStr).getDate()+'日';

}

//一个月以上

**else** **if**(se>DATE\_LEVEL.month){

\_text=(new Date(timeStr).getMonth()+1)+'月'+new Date(timeStr).getDate()+'日';

}

//一天以上

**else** **if**(se>DATE\_LEVEL.day){

\_text=Math.floor(se/DATE\_LEVEL.day)+'天前';

}

//一个小时以上

**else** **if**(se>DATE\_LEVEL.hour){

\_text=Math.floor(se/DATE\_LEVEL.hour)+'小时前';

}

//一个小时以内

**else**{

//如果小于1分钟，就显示1分钟前

**if**(se<DATE\_LEVEL.minter){se=DATE\_LEVEL.minter}

\_text=Math.floor(se/DATE\_LEVEL.minter)+'分钟前';

}

return \_text;

}

运行结果也是正确的，代码多了，但是神仙数字没了。可读性也不差。

这里也顺便提一下，如果硬要把上面的需求改成look-up的方式，代码就是下面这样。这样代码的修改的扩展性会强一些，成本会小一些，但是可读性不如上面。取舍关系，实际情况，实际分析。

**function** formatDate(timeStr){

//获取当前时间戳

let \_now=+new Date();

//求与当前的时间差

let se=\_now-timeStr;

let \_text='';

//求上一年最后一秒的时间戳

let lastYearTime=new Date(new Date().getFullYear()+'-01-01 00:00:00')-1;

//把时间差添加进去（当前时间戳与上一年最后一秒的时间戳的差）添加进去，如果时间差（se）超过这个值，则代表了这个时间是上一年的时间。

//DATE\_LEVEL.unshift(\_now-lastYearTime);

const DATE\_LEVEL={

month:2592000000,

day:86400000,

hour:3600000,

minter:60000,

}

let handleFn=[

{

time:DATE\_LEVEL.month,

fn:**function**(timeStr){

return (new Date(timeStr).getMonth()+1)+'月'+new Date(timeStr).getDate()+'日';

}

},

{

time:DATE\_LEVEL.day,

fn:**function**(timeStr){

return Math.floor(se/DATE\_LEVEL.day)+'天前';

}

},

{

time:DATE\_LEVEL.hour,

fn:**function**(timeStr){

return Math.floor(se/DATE\_LEVEL.hour)+'小时前';

}

},

{

time:DATE\_LEVEL.minter,

fn:**function**(timeStr){

return Math.ceil(se/DATE\_LEVEL.minter)+'分钟前';

}

}

];

//求上一年最后一秒的时间戳

let lastYearTime=new Date(new Date().getFullYear()+'-01-01 00:00:00')-1;

//把时间差（当前时间戳与上一年最后一秒的时间戳的差）和操作函数添加进去，如果时间差（se）超过这个值，则代表了这个时间是上一年的时间。

handleFn.unshift({

time:\_now-lastYearTime,

fn:**function**(timeStr){

**if**(se>DATE\_LEVEL.month){

return new Date(timeStr).getFullYear()+'年'+(new Date(timeStr).getMonth()+1)+'月'+new Date(timeStr).getDate()+'日';

}

},

});

let result='';

**for**(let i=0;i<handleFn.length;i++){

**if**(se>=handleFn[i].time){

result=handleFn[i].fn(timeStr);

**if**(result){

return result;

}

}

}

//如果发布时间小于1分钟，之际返回1分钟

return result='1分钟前'

}

**3.配置对象代替switch**

比如有一个需求：传入cash，check，draft，zfb，wx\_pay，对应输出：现金，支票，汇票，支付宝，微信支付。

需求也很简单，就一个switch就搞定了

**function** getPayChanne(tag){

switch(tag){

**case** 'cash':return '现金';

**case** 'check':return '支票';

**case** 'draft':return '汇票';

**case** 'zfb':return '支付宝';

**case** 'wx\_pay':return '微信支付';

}

}

但是这个的硬伤还是和上面一样，万一下次又要多加一个如：bank\_trans对应输出银行转账呢，代码又要改。类似的问题，同样的解决方案，配置数据和业务逻辑分离。代码如下。

**function** getPayChanne(tag){

let payChanneForChinese = {

'cash': '现金',

'check': '支票',

'draft': '汇票',

'zfb': '支付宝',

'wx\_pay': '微信支付',

};

return payChanneForChinese[tag];

}

同理，如果想封装一个通用的，也可以的

let payChanneForChinese = {

'cash': '现金',

'check': '支票',

'draft': '汇票',

'zfb': '支付宝',

'wx\_pay': '微信支付',

};

**function** getPayChanne(tag,chineseConfig){

return chineseConfig[tag];

}

getPayChanne('cash',payChanneForChinese);

这里使用对象代替 switch 好处就在于

1. 使用对象不需要 switch 逐个 case 遍历判断。
2. 使用对象，编写业务逻辑可能更灵活
3. 使用对象可以使得配置数据和业务逻辑分离。