代码优化

代码的的整体结构是影响运行速度的主要元素之一。代码数量少并不意味着运行速度就快，代码数量多也不意味着运行速度一定慢。代码的组织结构和解决具体问题的思路是影响代码性能的主要因素。

JS 与其他语言不同在于它的执行效率很大程度是取决于 JS engine 的效率。除了引擎实现的优劣外，引擎自己也会为一些特殊的代码模式采取一些优化的策略。例如 FF、Opera 和 Safari 的 JAVASCRIPT 引擎，都对字符串的拼接运算（+）做了特别优化。所以应该根据不同引擎进行不同优化。

而如果做跨浏览器的 web 编程，则最大的问题是在于 IE6（JScript 5.6），因为在不打 hotfix 的情况下，JScript 引擎的垃圾回收的 bug，会导致其在真实应用中的 performance 跟其他浏览器根本不在一个数量级上。因此在这种场合做优化，实际上就是为 JScript 做优化，所以第一原则就是只需要为 IE6（未打补丁的 JScript 5.6 或更早版本）做优化。接下来从若干的方向来告诉大家如何做好前端代码优化。

全局变量

创建全局变量被认为是糟糕的实践，尤其在团队开发的大背景下更是问题多多。随着代码量的增长，全局变量会导致一些非常重要的可维护性难题，全局变量越多，引入错误的概率会变得越高。
一般而言，有如下三种解决办法：

创建全局变量被认为是糟糕的实践，尤其在团队开发的大背景下更是问题多多。随着代码量的增长，全局变量会导致一些非常重要的可维护性难题，全局变量越多，引入错误的概率会变得越高
一般而言，有如下三种解决办法
1、零全局变量
实现方法是使用一个立即调用函数 IIFE 并将所有脚本放置其中

```
(function(){
 var doc = win.document;
})(window);
```

这种模式的使用场景有限，只要代码需要被其他的代码所依赖，或者需要在运行中被不断扩展或修改，就不能使用这种方式。
2、单全局变量和命名空间
依赖尽可能少的全局变量，即只创建一个全局变量，使用单变量模式，如 YUI 或 jQuery。

单全局变量，即所创建的这个唯一全局对象名是独一无二的，并将所有的功能代码都挂载到这个全局对象上。因此，每个可能的全局变量，都成为唯一全局变量的属性，从而不会创建多个全局变量。
命名空间是简单的通过全局对象的单一属性表示的功能性分组。比如 Y.DOM 下的所有方法都是和 DOM 操作相关的，Y.Event 下的所有方法都是和事件相关的。常见的约定是每个文件中都通过新的全局对象来声明自己的命名空间。
3、使用模块
模块是一种通用的功能片段，它并没有创建新的全局变量或命名空间。相反，所有的这些代码都存放于一个表示执行一个任务或发布一个接口的单函数中。可以用一个名称来表示这个模块，同样这个模块可以依赖其他模块。

事件处理

将事件处理相关的代码和事件环境耦合在一起，导致可维护性很糟糕。
1、隔离应用逻辑
将应用逻辑从所有事件处理程序中抽离出来是一种最佳实践，将应用逻辑和事件处理的代码拆分开来。

```
//不好的做法
function handleClick(event){
 var popup = document.getElementById('popup');
 popup.style.left = event.clientX + 'px';
 popup.style.top = event.clientY + 'px';
 popup.className = 'reveal';
}
addListener(element,'click',handleClick);
//好的做法
var MyApplication = {
 handleClick: function(event){
  this.showPopup(event);
 },
 showPopup: function(event){
  var popup = document.getElementById('popup');
  popup.style.left = event.clientX + 'px';
  popup.style.top = event.clientY + 'px';
  popup.className = 'reveal';
 }
};
addListener(element,'click',function(event){
 MyApplication.handleClick(event);
});
```

2、不要分发事件对象
应用逻辑不应当依赖于 event 对象来正确完成功能，方法接口应该表明哪些数据是必要的。代码不清晰就会导致 bug。最好的办法是让事件处理程序使用 event 对象来处理事件，然后拿到所有需要的数据传给应用逻辑

```
//改进的做法
var MyApplication = {
 handleClick: function(event){
  this.showPopup(event.clientX,event.clientY);
 },
 showPopup: function(x,y){
  var popup = document.getElementById('popup');
  popup.style.left = x + 'px';
  popup.style.top = y + 'px';
  popup.className = 'reveal';
 }
};
addListener(element,'click',function(event){
 MyApplication.handleClick(event);
});
```

当处理事件时，最好让事件程序成为接触到 event 对象的唯一的函数。事件处理程序应当在进入应用逻辑之前针对 event 对象执行任何必要的操作，包括阻止事件冒泡，都应当直接包含在事件处理程序中

```
var MyApplication = {
 handleClick: function(event){
  event.preventDefault();
  event.stopPropagation();
  this.showPopup(event.clientX,event.clientY);
 },
 showPopup: function(x,y){
  var popup = document.getElementById('popup');
  popup.style.left = x + 'px';
  popup.style.top = y + 'px';
  popup.className = 'reveal';
 }
};
addListener(element,'click',function(event){
 MyApplication.handleClick(event);
```

```
});
```

使用事件代理

- 当存在多个元素需要注册事件时，在每个元素上绑定事件本身就会对性能有一定损耗。
- 由于 DOM Level2 事件模 型中所有事件默认会传播到上层文档对象，可以借助这个机制在上层元素注册一个统一事件对不同子元素进行相应处理。

捕获型事件先发生。两种事件流会触发 DOM 中的所有对象，从 document 对象开始，也在 document 对象结束。

```
/*<ul id="parent-list">
    <li id="post-1">Item 1
    <li id="post-2">Item 2
    <li id="post-3">Item 3
    <li id="post-4">Item 4
    <li id="post-5">Item 5
    <li id="post-6">Item 6
</li></ul> */
// Get the element, add a click listener...
document.getElementById("parent-list").addEventListener("click",function(e) {
    // e.target is the clicked element!
    // If it was a list item
    if(e.target && e.target.nodeName == "LI") {
        // List item found! Output the ID!
        console.log("List item ",e.target.id.replace("post-")," was
clicked!");
    }
});
```

配置数据

代码无非是定义一些指令的集合让计算机来执行。我们常常将数据传入计算机，由指令对数据进行操作，并最终产生一个结果。当不得不修改数据时，可能会带来一些不必要的风险。应当将关键数据从代码中抽离出来
配置数据是指导在应用中写死的值，且将来可能会被修改，包括如下内容

1、URL

2、需要展现给用户的字符串
3、重复的值
4、配置项
5、任何可能发生变更的值

下面是未处理配置数据的做法

```
//不好的做法
function validate(value){
 if(!value){
  alert('Invalid value');
  location.href="/errors/invalid.php" rel="external nofollow" ;
 }
}
function toggleSelected(element){
 if(hasClass(element,'selected')){
  removeClass(element,'selected');
 }else{
  addClass(element,'selected');
 }
}
```


下面代码中将配置数据保存在了 config 对象中，config 对象的每个属性都保存了一个数据片段，每个属性名都有前缀，用以表明数据的类型(MSG 表示展现给用户的信息，URL 表示网络地址，CSS 表示这是一个 className)。当然，也可以将整个 config 对象放到单独的文件中，这样对配置数据的修改可以完全和使用这个数据的代码隔离开来

```
//好的做法
var config = {
 MSG_INVALID_VALUE: 'Invalid value',
 URL_INVALID:'/errors/invalid.php',
 CSS_SELECTED:'selected'
}
function validate(value){
 if(!value){
  alert(config.MSG_INVALID_VALUE);
  location.href=config.URL_INVALID;
 }
}
function toggleSelected(element){
```

```
 if(hasClass(element,config.CSS_SELECTED)){
  removeClass(element,config.CSS_SELECTED);
 }else{
  addClass(element,config.CSS_SELECTED);
 }
}
```

选择器优化

将选择器选择到的元素作为对象的静态属性集中到一个地方统一管理。

```
initializeElements: function() {
  var eles = app.Eles;
  for (var name in eles) {
    if (eles.hasOwnProperty(name)) {
      this[name] = $(eles[name]);
    }
  }
}
```


下面是一个例子

```
//好的做法
app.Eles = {
  widgetDiv: ".left-widget div",
  inputResize: '.input-resize',
  hr: '.hr',
  txt: '.input-group-btn button',
  cus: '#paper-type-cus',
  hid: '#hidden',
  mainCon: '#mainCon',
  rulerX: '.ruler-x',
  rulerY: '.ruler-y',
};
```

函数优化

一、提炼函数

在 javascript 开发中，大部分时间都在与函数打交道，所以希望这些函数有着良好的命名，函数体内包含的逻辑清晰明了。如果一个函数过长，不得不加上若干注释才能让这个函数显得易读一些，那这些函数就很有必要进行重构。

如果在函数中有一段代码可以被独立出来，那最好把这些代码放进另外一个独立的函数中。这是一种很常见的优化工作，这样做的好处主要有以下几点。

1、避免出现超大函数,还有使用纯函数最佳。

2、独立出来的函数有助于代码复用。

3、独立出来的函数更容易被覆写。

4、独立出来的函数如果拥有一个良好的命名，它本身就起到了注释的作用。

比如在一个负责取得用户信息的函数里面，还需要打印跟用户信息有关的 log，那么打印 log 的语句就可以被封装在一个独立的函数里：

```
var getUserInfo = function(){
  ajax( 'http:// xxx.com/userInfo', function( data ){
    console.log( 'userId: ' + data.userId );
    console.log( 'userName: ' + data.userName );
    console.log( 'nickName: ' + data.nickName );
  });
};
//改成：
var getUserInfo = function(){
  ajax( 'http:// xxx.com/userInfo', function( data ){
    printDetails( data );
  });
};
var printDetails = function( data ){
  console.log( 'userId: ' + data.userId );
  console.log( 'userName: ' + data.userName );
  console.log( 'nickName: ' + data.nickName );
};
```

二、尽量减少参数数量

如果调用一个函数时需要传入多个参数，那这个函数是让人望而生畏的，必须搞清楚这些参数代表的含义，必须小心翼翼地把它们按照顺序传入该函数。在实际开发中，向函数传递参数不可避免，但应该尽量减少函数接收的参数数量。下面举个非常简单的示例。有一个画图函数 draw，它现在只能绘制正方形，接收了 3 个参数，分别是图形的 width、heigth 以及 square：

```
var draw = function(width,height,square){};
```

但实际上正方形的面积是可以通过 width 和 height 计算出来的，于是我们可以把参数 square 从 draw 函数中去掉：

```
var draw = function( width, height ){
  var square = width * height;
};
```

假设以后这个 draw 函数开始支持绘制圆形，需要把参数 width 和 height 换成半径 radius，但图形的面积 square 始终不应该由客户传入，而是应该在 draw 函数内部，由传入的参数加上一定的规则计算得来。此时，可以使用策略模式，让 draw 函数成为一个支持绘制多种图形的函数

三、传递对象参数代替过长的参数列表

有时候一个函数有可能接收多个参数，而参数的数量越多，函数就越难理解和使用。使用该函数的人首先得搞明白全部参数的含义，在使用的时候，还要小心翼翼，以免少传了某个参数或者把两个参数搞反了位置。如果想在第 3 个参数和第 4 个参数之中增加一个新的参数，就会涉及许多代码的修改，代码如下：

```
var setUserInfo = function( id, name, address, sex, mobile, qq ){
  console.log( 'id= ' + id );
  console.log( 'name= ' +name );
  console.log( 'address= ' + address );
  console.log( 'sex= ' + sex );
  console.log( 'mobile= ' + mobile );
  console.log( 'qq= ' + qq );
};
setUserInfo( 1314, 'xiaohuochai', 'beijing', 'male', '150********',
121631835 );
```

这时可以把参数都放入一个对象内，然后把该对象传入 setUserInfo 函数，setUserInfo 函数需要的数据可以自行从该对象里获取。现在不用再关心参数的数量和顺序，只要保证参数对应的 key 值不变就可以了：

```
var setUserInfo = function( obj ){
    console.log( 'id= ' + obj.id );
    console.log( 'name= ' + obj.name );
    console.log( 'address= ' + obj.address );
    console.log( 'sex= ' + obj.sex );
    console.log( 'mobile= ' + obj.mobile );
    console.log( 'qq= ' + obj.qq );
  };
```

```
  setUserInfo({
    id: 1314,
    name: 'xiaohuochai',
    address: 'beijing',
    sex: 'male',
    mobile: '150********',
    qq: 121631835
  });
```

条件优化

JS 优化总是出现在大规模循环的地方：

这倒不是说循环本身有性能问题，而是循环会迅速放大可能存在的性能问题，所以第二原则就是以大规模循环体为最主要优化对象。

以下的优化原则，只在大规模循环中才有意义，在循环体之外做此类优化基本上是没有意义的。

目前绝大多数 JS 引擎都是解释执行的，而解释执行的情况下，在所有操作中，函数调用的效率是较低的。此外，过深的 prototype 继承链或者多级引用也会降低效率。JScript 中，10 级引用的开销大体是一次空函数调用开销的 1/2。这两者的开销都远远大于简单操作（如四则运算）。

一、合并条件片段

如果一个函数体内有一些条件分支语句，而这些条件分支语句内部散布了一些重复的代码，那么就有必要进行合并去重工作。假如有一个分页函数 paging，该函数接收一个参数 currPage，currPage 表示即将跳转的页码。在跳转之前，为防止 currPage 传入过小或者过大的数字，要手动对它的值进行修正，详见如下伪代码：

```
var paging = function( currPage ){
  if ( currPage <= 0 ){
    currPage = 0;
    jump( currPage ); // 跳转
  }else if ( currPage >= totalPage ){
    currPage = totalPage;
    jump( currPage ); // 跳转
  }else{
    jump( currPage ); // 跳转
  }
```

```
};
```

可以看到，负责跳转的代码 jump(currPage)在每个条件分支内都出现了，所以完全可以把这句代码独立出来：

```
var paging = function( currPage ){
  if ( currPage <= 0 ){
    currPage = 0;
  }else if ( currPage >= totalPage ){
    currPage = totalPage;
  }
  jump( currPage ); // 把 jump 函数独立出来
};
```

二、把条件分支语句提炼成函数

在程序设计中，复杂的条件分支语句是导致程序难以阅读和理解的重要原因，而且容易导致一个庞大的函数。假设现在有一个需求是编写一个计算商品价格的 getPrice 函数，商品的计算只有一个规则：如果当前正处于夏季，那么全部商品将以 8 折出售。代码如下：

```
var getPrice = function( price ){
  var date = new Date();
  if ( date.getMonth() >= 6 && date.getMonth() <= 9 ){ // 夏天
    return price * 0.8;
  }
  return price;
};
```

观察这句代码：

```
date.getMonth()>=6&&date.getMonth()<=9
```

这句代码要表达的意思很简单，就是判断当前是否正处于夏天（7~10 月）。尽管这句代码很短小，但代码表达的意图和代码自身还存在一些距离，阅读代码的人必须要多花一些精力才能明白它传达的意图。其实可以把这句代码提炼成一个单独的函数，既能更准确地表达代码的意思，函数名本身又能起到注释的作用。代码如下：

```
var isSummer = function(){
  var date = new Date();
  return date.getMonth() >= 6 && date.getMonth() <= 9;
};
var getPrice = function( price ){
  if ( isSummer() ){ // 夏天
```

```
    return price * 0.8;
  }
  return price;
};
```

三、提前让函数退出代替嵌套条件分支

许多程序员都有这样一种观念：“每个函数只能有一个入口和一个出口。”现代编程语言都会限制函数只有一个入口。但关于“函数只有一个出口”，往往会有一些不同的看法。下面这段伪代码是遵守“函数只有一个出口的”的典型代码：

```
var del = function( obj ){
  var ret;
  if ( !obj.isReadOnly ){ // 不为只读的才能被删除
    if ( obj.isFolder ){ // 如果是文件夹
      ret = deleteFolder( obj );
    }else if ( obj.isFile ){ // 如果是文件
      ret = deleteFile( obj );
    }
  }
  return ret;
};
```


嵌套的条件分支语句绝对是代码维护者的噩梦，对于阅读代码的人来说，嵌套的 if、else 语句相比平铺的 if、else，在阅读和理解上更加困难。嵌套的条件分支往往是由一些深信“每个函数只能有一个出口的”程序员写出的。但实际上，如果对函数的剩余部分不感兴趣，那就应该立即退出。引导阅读者去看一些没有用的 else 片段，只会妨碍他们对程序的理解

于是可以挑选一些条件分支，在进入这些条件分支之后，就立即让这个函数退出。要做到这一点，有一个常见的技巧，即在面对一个嵌套的 if 分支时，可以把外层 if 表达式进行反转。重构后的 del 函数如下：

```
var del = function( obj ){
  if ( obj.isReadOnly ){ // 反转 if 表达式
    return;
  }
  if ( obj.isFolder ){
    return deleteFolder( obj );
  }
  if ( obj.isFile ){
```

```
    return deleteFile( obj );
  }
};
```

循环优化

一、合理使用循环
在函数体内，如果有些代码实际上负责的是一些重复性的工作，那么合理利用循环不仅可以完成同样的功能，还可以使代码量更少。下面有一段创建 XHR 对象的代码，为了简化示例，只考虑版本 9 以下的 IE 浏览器，代码如下：

```
var createXHR = function(){
  var xhr;
  try{
    xhr = new ActiveXObject( 'MSXML2.XMLHttp.6.0' );
  }catch(e){
    try{
      xhr = new ActiveXObject( 'MSXML2.XMLHttp.3.0' );
    }catch(e){
      xhr = new ActiveXObject( 'MSXML2.XMLHttp' );
    }
  }
  return xhr;
};
var xhr = createXHR();
```

下面灵活地运用循环，可以得到跟上面代码一样的效果：

```
//下面我们灵活地运用循环，可以得到跟上面代码一样的效果：
var createXHR = function(){
  var versions= [ 'MSXML2.XMLHttp.6.0ddd', 'MSXML2.XMLHttp.3.0',
'MSXML2.XMLHttp' ];
  for ( var i = 0, version; version = versions[ i++ ]; ){
    try{
      return new ActiveXObject( version );
```

```
    }catch(e){
    }
  }
};
var xhr = createXHR();
```

二、用 return 退出多重循环
假设在函数体内有一个两重循环语句，需要在内层循环中判断，当达到某个临界条件时退出外层的循环。大多数时候会引入一个控制标记变量：

```
var func = function(){
  var flag = false;
  for ( var i = 0; i < 10; i++ ){
    for ( var j = 0; j < 10; j++ ){
      if ( i * j >30 ){
        flag = true;
        break;
      }
    }
    if ( flag === true ){
      break;
    }
  }
};
```

第二种做法是设置循环标记：

```
var func = function(){
  outerloop:
  for ( var i = 0; i < 10; i++ ){
    innerloop:
    for ( var j = 0; j < 10; j++ ){
      if ( i * j >30 ){
        break outerloop;
      }
    }
  }
};
```

这两种做法无疑都让人头晕目眩，更简单的做法是在需要中止循环的时候直接退出整个方法：

```
var func = function(){
  for ( var i = 0; i < 10; i++ ){
    for ( var j = 0; j < 10; j++ ){
      if ( i * j >30 ){
        return;
      }
    }
  }
};
```

当然用 return 直接退出方法会带来一个问题，如果在循环之后还有一些将被执行的代码呢？如果提前退出了整个方法，这些代码就得不到被执行的机会：

```
var func = function(){
  for ( var i = 0; i < 10; i++ ){
    for ( var j = 0; j < 10; j++ ){
      if ( i * j >30 ){
        return;
      }
    }
  }
  console.log( i ); // 这句代码没有机会被执行
};
```

为了解决这个问题，可以把循环后面的代码放到 return 后面，如果代码比较多，就应该把它们提炼成一个单独的函数：

```
var print = function( i ){
  console.log( i );
};
var func = function(){
  for ( var i = 0; i < 10; i++ ){
    for ( var j = 0; j < 10; j++ ){
      if ( i * j >30 ){
```

```
        return print( i );
      }
    }
  }
};func();
```

尽量避免过多的引用层级和不必要的多次方法调用：

特别要注意的是，有些情况下看似是属性访问，实际上是方法调用。例如所有 DOM 的属性，实际上都是方法。在遍历一个 NodeList 的时候，循环 条件对于 nodes.length 的访问，看似属性读取，实际上是等价于函数调用的。而且 IE DOM 的实现上，childNodes.length 每次是要通过内部遍历重新计数的。（My god，但是这是真的！因为我测过，childNodes.length 的访问时间与 childNodes.length 的值成正比！）这非常耗费。所以 预先把 nodes.length 保存到 js 变量，当然可以提高遍历的性能。

同样是函数调用，用户自定义函数的效率又远远低于语言内建函数，因为后者是对引擎本地方法的包装，而引擎通常是 c,c++,java 写的。进一步，同样的功能，语言内建构造的开销通常又比内建函数调用要效率高，因为前者在 JS 代码的 parse 阶段就可以确定和优化。

尽量使用语言本身的构造和内建函数：

这里有一个例子是高性能的 String.format 方法。 String.format 传统的实现方式是用 String.replace(regex, func)，在 pattern 包含 n 个占位符（包括重复的）时，自定义函数 func 就被调用 n 次。而这个高性能实现中，每次 format 调用所作的只是一次 Array.join 然后一次 String.replace(regex, string)的操作，两者都是引擎内建方法，而不会有任何自定义函数调用。两次内建方法调用和 n 次的自定义方法调用，这就是性能上的差别。

同样是内建特性，性能上也还是有差别的。例如在 JScript 中对于 arguments 的访问性能就很差，几乎赶上一次函数调用了。因此如果一个 可变参数的简单函数成为性能瓶颈的时候，可以将其内部做一些改变，不要访问 arguments，而是通过对参数的显式判断来处理，比如：

```
function sum() {
    var r = 0;
    for (var i = 0; i < arguments.length; i++) {
        r += arguments[i];
    }
```

```
    return r;
}

//这个 sum 通常调用的时候个数是较少的，我们希望改进它在参数较少时的性能。如果改成：
function sum() {
    switch (arguments.length) {
        case 1: return arguments[0];
        case 2: return arguments[0] + arguments[1];
        case 3: return arguments[0] + arguments[1] + arguments[2];
        case 4: return arguments[0] + arguments[1] + arguments[2] +
arguments[3];
        default:
            var r = 0;
            for (var i = 0; i < arguments.length; i++) {
                r += arguments[i];
            }
            return r;
    }
}

//其实并不会有多少提高，但是如果改成：
function sum(a, b, c, d, e, f, g) {
    var r = a ? b ? c ? d ? e ? f ? a + b + c + d + e + f : a + b + c + d +
e : a + b + c + d : a + b + c : a + b : a : 0;
    if (g === undefined) return r;
    for (var i = 6; i < arguments.length; i++) {
        r += arguments[i];
    }
    return r;
}

//就会提高很多（至少快 1 倍）。
```

总结

以上方法是根据个人经验和想法进行的一些可优化的思维拓展，代码的优化道路上，从来都是要特定场景下解决特定需求的，为的还是要让使用更简单，让使用者更习惯、高效的开发，提前或者滞后的将代码进行优化重构固然都是错的，但如果一点点优化的思考和什么程度应该去做重构了不去探索就进步太慢了。