**JavaScript编程规范**

目 录

[1 JavaScript语言编程规范总则 3](#_Toc295468125)

[2 排版规范 4](#_Toc295468126)

[2.1 规则 4](#_Toc295468127)

[2.1.1 缩进程序块 4](#_Toc295468128)

[2.1.2 分界符独占 4](#_Toc295468129)

[2.1.3 限制列宽 5](#_Toc295468130)

[2.1.4 语句独占 5](#_Toc295468131)

[2.1.5 关键字独占 5](#_Toc295468132)

[2.1.6 程序块独立 5](#_Toc295468133)

[2.1.7 明确操作符 6](#_Toc295468134)

[2.2 建议 7](#_Toc295468135)

[2.2.1 声明排序 7](#_Toc295468136)

[3 注释规范 8](#_Toc295468137)

[3.1 规则 8](#_Toc295468138)

[3.2 建议 8](#_Toc295468139)

[4 命名规范 9](#_Toc295468140)

[4.1 规则 9](#_Toc295468141)

[4.2 建议 9](#_Toc295468142)

[5 编码规范 10](#_Toc295468143)

[5.1 规则 10](#_Toc295468144)

[5.2 建议 10](#_Toc295468145)

# JavaScript语言编程规范总则

为了提高源程序的质量和可维护性，最终提高公司软件产品生产力，我们有必要对公司软件产品的源程序的编写风格作出统一的规范约束。

本规范总则是基于JavaScript语言的编程规则的总的原则，针对不同应用环境的编程规范在遵守本规范总则的基础上，还必须遵守相应开发环境的编程规范。本规范适用于使用JavaScript语言编程的部门和产品。

本规范总则的内容包括：排版规范，注释规范，命名规范，编码规范等。

本规范总则的示例采用以下的术语描述：

* **规则**：编程时强制必须遵守的原则。
* **建议**：编程时必须加以考虑的原则。
* **格式**：对此规范格式的说明。
* **说明**：对此规则或建议进行必要的解释。
* **示例**：对此规则或建议从正、反两个方面给出例子。

# 排版规范

## 规则

### 缩进程序块

**说明：**

缩进只使用1个TAB键，不使用空格键。

以免用不同的编辑器阅读程序时，因TAB键所设置的空格数目不同而造成程序布局不整齐，以自己的编程习惯修改按指定空格数显示TAB（一般指定4个空格）。

### 分界符独占

**说明：**

分界符（如大括号‘{’和‘}’）应各独占一行并且位于同一列。

同时与引用它们的语句左对齐。在函数体的开始、类和接口的定义、以及if、for、do、while、switch、case语句中的程序都要采用如上的缩进方式。

**示例：**

|  |  |
| --- | --- |
| **例子不符合规范** | **应如下书写** |
| for (...) {  // program code  } | for (...)  {  // program code  } |
| if (...) {  // program code  } | if (...)  {  // program code  } |
| void example\_fun( void ) {  // program code  } | void example\_fun( void )  {  // program code  } |

### 限制列宽

**说明：**

较长的语句、表达式或参数（>80字符）要分成多行书写，长表达式要在低优先级操作符处划分新行，操作符放在新行之首，划分出的新行要进行适当的缩进，使排版整齐，语句可读。

由于现在的屏幕越来越大，可以根据自己部门的情况，适当调整限制的列宽数。如：100列、120列

### 语句独占

**说明：**

不允许把多个短语句写在一行中，即一行只写一条语句。

### 关键字独占

**说明：**

if, for, do, while, switch, case, default 等语句自占一行，且这些关键字随后的执行语句无论多少都要加括号{}。

### 程序块独立

**说明：**

相对独立的程序块之间、变量说明之后必须加空行。

**示例：**

|  |  |
| --- | --- |
| **例子不符合规范** | **应如下书写** |
| if(log.getLevel() < LogConfig.getRecordLevel())  {  return;  }  LogWriter writer; | if(log.getLevel() < LogConfig.getRecordLevel())  {  return;  }  LogWriter writer; |

### 明确操作符

在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的立即操作符（如.），后不应加空格。

采用这种松散方式编写代码的目的是使代码更加清晰。

**说明：**

由于留空格所产生的清晰性是相对的，所以，在已经非常清晰的语句中没有必要再留空格，如果语句已足够清晰则括号内侧(即左括号后面和右括号前面)不需要加空格，多重括号间不必加空格，因为在Java语言中括号已经是最清晰的标志了。

在长语句中，如果需要加的空格非常多，那么应该保持整体清晰，而在局部不加空格。给操作符留空格时不要连续留两个以上空格。

**示例：**

1. 逗号、分号只在后面加空格。

|  |
| --- |
| int a, b, c; |

1. 比较操作符, 赋值操作符"="、 "+="，算术操作符"+"、"%"，逻辑操作符"&&"、"&"，位域操作符"<<"、"^"等双目操作符的前后加空格

|  |
| --- |
| if (current\_time >= MAX\_TIME\_VALUE)  a = b + c;  a \*= 2;  a = b ^ 2; |

1. "!"、"~"、"++"、"--"、"&"（地址运算符）等单目操作符前后不加空格

|  |
| --- |
| // 非操作"!"与内容之间  flag = !isEmpty;  // "++","--"与内容之间  i++; |

1. "."前后不加空格

|  |
| --- |
| // "."前后不加空格  p.id = pid; |

1. if、for、while、switch等与后面的括号间应加空格，使if等关键字更为突出、明显

|  |
| --- |
| if (a >= b && c > d) |

## 建议

### 声明排序

**说明：**

类属性的声明和类方法的声明不要交叉放置，不同存取范围的属性或者方法也尽量不要交叉放置。

**格式：**

类声明

{

类的公有属性声明

类的保护属性声明

类的私有属性声明

类的公有方法声明

类的保护方法声明

类的私有方法声明

}

# 注释规范

## 规则

### 30％的注释

一般情况下，源程序有效注释量必须在30％以上。

**说明：**

注释的原则是有助于对程序的阅读理解，在该加的地方都加了，注释不宜太多也不能太少，注释语言必须准确、易懂、简洁。可以用注释统计工具来统计。

### 文件注释

文件注释写入文件头部。

**格式：**

|  |
| --- |
| /\*  \* 文件名：[文件名]  \* 版权：〈版权〉  \* 描述：〈描述〉  \* 修改人：〈修改人〉  \* 修改时间：YYYY-MM-DD  \* 跟踪单号：〈跟踪单号〉  \* 修改单号：〈修改单号〉  \* 修改内容：〈修改内容〉  \*/ |

**说明：**

每次修改后在文件头部写明修改信息，CheckIn的时候可以直接把蓝色字体信息粘贴到VSS的注释上。在代码受控之前可以免去。

**示例：**

|  |
| --- |
| /\*  \* 文件名：LogManager.js  \* 版权：Copyright 2000-2001 Huawei Tech. Co. Ltd. All Rights Reserved.  \* 描述： WIN V200R002 WEBSMAP 通用日志系统  \* 修改人： 张三  \* 修改时间：2001-02-16  \* 修改内容：新增  \* 修改人： 李四  \* 修改时间：2001-02-26  \* 跟踪单号：D20103  \* 修改单号：WSS368  \* 修改内容：。。。。。。  \* 修改人： 王五  \* 修改时间：2001-03-25  \* 跟踪单号：D27153  \* 修改单号：WSS498  \* 修改内容：。。。。。。  \*/ |

### 类的注释

该注释放在 function 关键字之前。

**格式：**

|  |
| --- |
| /\*\*  \* 〈一句话功能简述〉  \* 〈功能详细描述〉  \* @author [作者]  \* @version [版本号, YYYY-MM-DD]  \* @see [相关类/方法]  \* @since [产品/模块版本]  \* @deprecated  \*/ |

**说明：**

描述部分说明该类或者接口的功能、作用、使用方法和注意事项，每次修改后增加作者和更新版本号和日期，@since 表示从那个版本开始就有这个类或者接口，@deprecated 表示不建议使用该类或者接口。

**示例：**

|  |
| --- |
| /\*\*  \* LogManager 类集中控制对日志读写的操作。  \* 全部为静态变量和静态方法，对外提供统一接口。分配对应日志类型的读写器，  \* 读取或写入符合条件的日志纪录。  \* @author 张三，李四，王五  \* @version 1.2, 2001-03-25  \* @see LogIteraotor  \* @see BasicLog  \* @since CommonLog1.0  \*/ |

### 类属性的注释

写在类属性上面。

**说明：**

成员变量注释内容：成员变量的意义、目的、功能，可能被用到的地方。

**示例：**

|  |
| --- |
| /\*\*  \* 私有变量的注释内容  \*/  var privateVariable;  /\*\*  \* 公有变量的注释内容  \*/  this.publicVariable = null; |

### 类方法的注释

写在类方法上面。

公有和保护方法注释内容：列出方法的一句话功能简述、功能详细描述、输入参数、输出参数、返回值、异常等。

**格式：**

|  |
| --- |
| /\*\*  \* 〈一句话功能简述〉  \* 〈功能详细描述〉  \* @param [参数1] [参数1说明]  \* @param [参数2] [参数2说明]  \* @return [返回类型说明]  \* @exception/throws [违例类型] [违例说明]  \* @see [类、类#方法、类#成员]  \* @deprecated  \*/ |

**说明：**

* @since 表示从那个版本开始就有这个方法
* @exception或throws 列出可能仍出的异常
* @deprecated 表示不建议使用该方法。

**示例：**

|  |
| --- |
| /\*\*  \* 根据日志类型和时间读取日志。  \*  \* 分配对应日志类型的LogReader， 指定类型、查询时间段和条件，  \* 读取日志记录。查询条件为null或0的表示没有限制。  \* 查询时间为左包含原则，即 [startTime, endTime) 。  \*  \* @param logTypeName 日志类型名（在配置文件中定义的）  \* @param startTime 查询日志的开始时间  \* @param endTime 查询日志的结束时间  \* @param logLevel 查询日志的级别  \* @return 结果集，日志反复器  \*  \* @since CommonLog1.0  \*/  this.publicMethod = function(logType, startTime, endTime, logLevel) |
| function privateMethod(logType, startTime, endTime, logLevel) |

### 上相邻原则

注释应与其描述的代码相近，对代码的注释应放在其上方或右方（对单条语句的注释）相邻位置，不可放在下面，如放于上方则需与其上面的代码用空行隔开。

### 注释的排版

注释与所描述内容进行同样的缩排。

将注释与其上面的代码用空行隔开。

**说明：**

可使程序排版整齐，并方便注释的阅读与理解。

**示例：**

|  |  |
| --- | --- |
| **排版不整齐，阅读稍感不方便** | **应改为如下布局** |
| function example( )  {  //注释  CodeBlock One  // 注释  CodeBlock Two  } | function example( )  {  // 注释  CodeBlock One  // 注释  CodeBlock Two  } |

### 关键字语句的注释

对变量的定义和分支语句（条件分支、循环语句等）必须编写注释。

**说明：**

这些语句往往是程序实现某一特定功能的关键，对于维护人员来说，良好的注释帮助更好的理解程序，有时甚至优于看设计文档。

### no-break-case的注释

对于switch语句下的case语句，如果因为特殊情况需要处理完一个case后进入下一个case处理，必须在该case语句处理完、下一个case语句前加上明确的注释。

**说明：**

这样比较清楚程序编写者的意图，有效防止无故遗漏break语句。

**示例：**

|  |
| --- |
| var days = 0;  switch(month)  {  case 2:  days = ((year % 4 == 0 && year % 100 != 0)  || (year % 400 == 0)) ? 29 : 28;  break;  case 1:  case 3:  case 5:  case 7:  case 8:  case 10:  case 12:  // 这些月份要多1天  days += 1;  // no break here  default:  // 每个月默认有30天  days += 30;  break;  } |

### 注释与代码同步

边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性。不再有用的注释要删除。

### 写清晰的注释

注释的内容要清楚、明了，含义准确，防止注释二义性。

**说明：**

错误的注释不但无益反而有害。

### 甚用缩写

避免在注释中使用缩写，特别是不常用缩写。

**说明：**

在使用缩写时或之前，应对缩写进行必要的说明。

## 建议

### 不要插入注释

避免在一行代码或表达式的中间插入注释。

**说明：**

除非必要，不应在代码或表达中间插入注释，否则容易使代码可理解性变差。

### 使用自注释

通过对函数或过程、变量、结构等正确的命名以及合理地组织代码的结构，使代码成为自注释的。

**说明：**

清晰准确的函数、变量等的命名，可增加代码可读性，并减少不必要的注释。

### 添加辅助信息

在代码的功能、意图层次上进行注释，提供有用、额外的信息。

**说明：**

注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助读者理解代码，防止没必要的重复注释信息。

**示例：**

|  |  |
| --- | --- |
| **如下注释意义不大** | **如下的注释则给出了额外有用的信息** |
| // 如果 receiveFlag 为真  if (receiveFlag) | // 如果从连结收到消息  if (receiveFlag) |

### 对程序块结束的注释

在程序块的结束行右方加注释标记，以表明某程序块的结束。

**说明：**

当代码段较长，特别是多重嵌套时，这样做可以使代码更清晰，更便于阅读。

**示例：**

|  |
| --- |
| if (...)  {  // program code for if  while (index < MAX\_INDEX)  {  // program code for while  } // end of while (index < MAX\_INDEX) // 指明该条while语句结束  } // end of if (...) // 指明是哪条if语句结束 |

### 统一注释语言

注释应考虑程序易读及外观排版的因素，使用的语言若是中、英兼有的，建议多使用中文，除非能用非常流利准确的英文表达。

**说明：**

注释语言不统一，影响程序易读性和外观排版，出于对维护人员的考虑，建议使用中文。

### 方法内使用单行注释

**说明：**

调试程序的时候可以方便的使用 /\* 。。。\*/ 注释掉一长段程序。

具体使用时要看工具的支持，如 eclipse 最好以 // 注释，因 eclipse 会把 /\* 。。。\*/ 中内容做为文体进行格式化，在阅读时不方便。在 eclipse 中注释多行的操作方法：先选中连续的要被注释的行，再按 Ctrl + / 注释已经选中的行。

### 注释要成句成段

注释尽量使用中文注释和中文标点。方法和类描述的第一句话尽量使用简洁明了的话概括一下功能，然后加以句号。接下来的部分可以详细描述。

### 流程的注释

顺序实现流程的说明使用1、2、3、4在每个实现步骤部分的代码前面进行注释。

**示例：**

如下是对设置属性的流程注释

|  |
| --- |
| // 1、判断输入参数是否有效。  。。。。。  // 2、设置本地变量。  。。。。。。 |

### 注释说明复杂代码

一些复杂的代码需要说明。

**示例：**

这里主要是对闰年算法的说明。

|  |
| --- |
| // 1. 如果能被4整除，是闰年；  // 2. 如果能被100整除，不是闰年.；  // 3. 如果能被400整除，是闰年.。 |

# 命名规范

## 规则

### 包

包名采用域后缀倒置的加上自定义的包名，采用小写字母。在部门内部应该规划好包名的范围，防止产生冲突。部门内部产品使用部门的名称加上模块名称。产品线的产品使用产品的名称加上模块的名称。

**格式：**

|  |
| --- |
| com.huawei.产品名.模块名称  com.huawei.部门名称.项目名称 |

**示例：**

|  |
| --- |
| if (!com)  {  var com = new Object();  }  if (!com.huawei)  {  // {} 等价于 new Object()  com.huawei = {};  }  if (!com.huawei.imap)  {  com.huawei.imap = {};  }  if (!com.huawei.imap.iview)  {  com.huawei.imap.iview = {};  } |

### 类

类名使用类意义完整的英文描述，每个英文单词的首字母使用大写、其余字母使用小写的大小写混合法。

**示例：**

|  |
| --- |
| com.huawei.imap.iview.OrderInformation = function()  {  // declare properties and methods  } |
| com.huawei.imap.iview.CustomerList = function()  {  // declare properties and methods  } |
| com.huawei.imap.iview.LogManager = function()  {  // declare properties and methods  } |
| com.huawei.imap.iview.LogConfig = function()  {  // declare properties and methods  } |
| com.huawei.imap.iview.SmpTransaction = function()  {  // declare properties and methods  } |

### 方法

方法名使用类意义完整的英文描述：第一个单词的字母使用小写、剩余单词首字母大写其余字母小写的大小写混合法。

**示例：**

|  |
| --- |
| this.calculateRate = function()  {  // program code  } |
| function addNewOrder()  {  // program code  } |

### 属性

属性名使用意义完整的英文描述：第一个单词的字母使用小写、剩余单词首字母大写其余字母小写的大小写混合法。属性名不能与方法名相同。

**示例：**

|  |
| --- |
| var customerName = undefined; |
| var orderNumber = undefined; |
| this.smpSession = undefined; |

### 常量

常量名使用全大写的英文描述，英文单词之间用下划线分隔开。

**示例：**

|  |
| --- |
| var MAX\_VALUE = 1000; |
| this.DEFAULT\_START\_DATE = "2001-12-08"; |

为常量分组

|  |
| --- |
| var TRUNK = {  IDLE : 0,  BUSY : 1,  UNKNOWN : -1  }; |

### 控制范围

准确地确定成员函数的存取控制符号，使用“this.”或“function”“var”控制方法或属性的使用范围。

**示例：**

|  |  |  |
| --- | --- | --- |
|  | **范围** | **示例** |
| 属性 | private 私有 | var privateMember = undefined; |
| public 公有 | this.publicMember = undefined; |
| external 对外 | this.$$$$$externalMember = undefined; |
| 方法 | private 私有 | function privateMethod()  {  // program code  } |
| public 公有 | this.publicMethod = function()  {  // program code  } |
| external 对外 | this.$$$$$externalMethod = function()  {  // program code  } |

**说明：**

因 JavaScript 语言的特殊性，推荐使用一些特殊标识来标记对外接口（这里使用了$$$$$，当然也可以用不同的$的数量来区分方法使用的范围，如：$$产品内部的模块间可以调用，$$$$$下游产品可以调用）。

## 建议

### 类声明在闭包内

**格式：**

|  |
| --- |
| function package(root, children)  {  var parent = root;  for ( var i = 1; i < arguments.length; i++)  {  if (!parent[arguments[i]])  {  parent[arguments[i]] = {};  }  parent = parent[arguments[i]];  }  } |
| if (!com)  {  var com = new Object();  }  (function(){  // declare package: com.huawei.imap.iview  // 此处也可以使用其它方式声明包  package(com, ‘huawei’, ‘imap’, ‘iview’);  // import  var ImportedClass = com.huawei.imap.iview.ImportedClass;  /\*\*  \* const comment  \*/  var INTERNAL\_CONST = undefined;  var INTERNAL\_CONST\_GROUP = {  CONST\_0 : undefined,  CONST\_1 : undefined  };  /\*\*  \* class comment  \*/  com.huawei.imap.iview.DeclareClass = function()  {  // ===================constants  /\*\*  \* const comment  \*/  this.$$$$$EXTERNAL\_CONST = undefined;  this.PUBLIC\_CONST = undefined;  var PRIVATE\_CONST = undefined;  // ===================constant groups  /\*\*  \* const comment  \*/  this.$$$$$EXTERNAL\_CONST\_GROUP = {  CONST : undefined  };  this.PUBLIC\_CONST\_GROUP = {  CONST : undefined  };  var PRIVATE\_CONST\_GROUP = {  CONST : undefined  };  // ===================properties  /\*\*  \* property comment  \*/  this.$$$$$externalProperty = undefined;  this.publicProperty = undefined;  var privateProperty = undefined;  // ===================methods  /\*\*  \* method comment  \*/  this.$$$$$externalMethod = function()  {  // program code  }  this.publicMethod = function()  {  // program code  }  function privateMethod()  {  // program code  }  }  })(); |

### 类的prototype声明

把类的属性（包括常量）和方法分别声明。这样声明后类的结构比较清晰。

**格式：**

|  |
| --- |
| (function(){  /\*\*  \* class comment  \*/  com.huawei.imap.iview.DeclareClass = function()  {  // ===================constants  /\*\*  \* const comment  \*/  this.$$$$$EXTERNAL\_CONST = undefined;  this.PUBLIC\_CONST = undefined;  var PRIVATE\_CONST = undefined;  // ===================constant groups  /\*\*  \* const comment  \*/  this.$$$$$EXTERNAL\_CONST\_GROUP = {  CONST : undefined  };  this.PUBLIC\_CONST\_GROUP = {  CONST : undefined  };  var PRIVATE\_CONST\_GROUP = {  CONST : undefined  };  // ===================properties  /\*\*  \* property comment  \*/  this.$$$$$externalProperty = undefined;  this.publicProperty = undefined;  var privateProperty = undefined;  // this block is similar to a constructor  {  this.init();  // some other code here  }  }  // ===================methods  com.huawei.imap.iview.DeclareClass.prototype = {  /\*\*  \* method comment  \*/  $$$$$externalMethod : function()  {  // program code  },  publicMethod : function()  {  // program code  },  init : function()  {  // program code  }  };  })(); |

### 变量的后缀

常用组件类的命名以组件名加上组件类型名结尾。

**示例：**

|  |  |  |
| --- | --- | --- |
| **类型** | **后缀** | **示例** |
| Application | App | MainApp |
| Frame | Frame | TopoFrame |
| Panel | Panel | CreateCircuitPanel |
| Bean | Bean | DataAccessBean |
| EJB | EJB | DBProxyEJB |
| Applet | Applet | PictureShowApplet |

### 使用通过的缩写

如果函数名超过15 个字母，可采用以去掉元音字母的方法或者以行业内约定俗成的缩写方式缩写函数名。

**示例：**

|  |  |
| --- | --- |
| **例如** | **缩写** |
| getCustomerInformation() | getCustomerInfo() |

### 有意义的集合命名

含有集合意义的属性命名，尽量包含其复数的意义。

**示例：**

|  |
| --- |
| var customers = new Array(); |
| // [] 等价于 new Array()  var orderItems = []; |

# 编码规范

## 规则

### 明确方法功能

精确（而不是近似）地实现方法设计。一个函数仅完成一件功能，即使简单功能也应该编写方法实现。

**说明：**

虽然为仅用一两行就可完成的功能去编方法好象没有必要，但用方法可使功能明确化，增加程序可读性，亦可方便维护、测试。

### 明确类的功能

精确（而不是近似）地实现类的设计。一个类仅实现一组相近的功能。

**说明：**

划分类的时候，应该尽量把逻辑处理、数据和显示分离，实现类功能的单一性。

数据类不能包含数据处理的逻辑。

通信类不能包含显示处理的逻辑。

### 对外接口参数合法检查

应明确规定对外接口方法参数的合法性检查应由方法的调用者负责还是由接口方法本身负责，缺省是由方法调用者负责。

**说明：**

对于模块间接口方法的参数的合法性检查这一问题，往往有两个极端现象，即：要么是调用者和被调用者对参数均不作合法性检查，结果就遗漏了合法性检查这一必要的处理过程，造成问题隐患；要么就是调用者和被调用者均对参数进行合法性检查，这种情况虽不会造成问题，但产生了冗余代码，降低了效率。

### 用括号代替默认优先级

注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级。

**说明：**

防止阅读程序时产生误解，防止因默认的优先级与设计思想不符而导致程序出错。

**示例：**

|  |  |  |
| --- | --- | --- |
| **有括号** | **无括号** | **结果** |
| (high << 8) | low | high << 8 | low | 虽然不会出错  但语句不易理解 |
| (a | b) && (a & c) | a | b && a & c |
| (a | b) < (c & d) | a | b < c & d | 判断条件出错 |

### 魔鬼数字

避免使用不易理解的数字，用有意义的标识来替代。涉及物理状态或者含有物理意义的常量，不应直接使用数字，必须用有意义的静态变量来代替。

**示例：**

|  |  |
| --- | --- |
| **如下的程序可读性差** | **应改为如下形式** |
| if (state == 0)  {  state = 1;  // program code  } | var TRUNK = {  IDLE : 0,  BUSY : 1,  UNKNOWN : -1  };  if (state == TRUNK.IDLE)  {  state = TRUNK.BUSY;  // program code  } |

### 控制参数数量

方法的参数不要超过5个，如果需要过多的参数，定义结构体做为参数

**示例：**

|  |  |
| --- | --- |
| **如下的程序可读性差** | **应改为如下形式** |
| function queryAlarm(  alarmName,  alarmSeverity,  alarmStatus,  alarmType,  alarmOccurenceTime,  alarmClearanceTime,  alarmAcknowledgedTime,  alarmRemark)  {  // program code  } | /\*\*  \* ... ...  \* @param alarmFilter = {  \* name : ‘...’,  \* severity : ‘...’,  \* status : ‘...’,  \* type : ‘...’,  \* occurenceTime : ‘...’,  \* clearanceTime : ‘...’,  \* acknowledgedTime : ‘...’,  \* remark : ‘...’}  \* ... ...  \*/  function queryAlarm(alarmFilter)  {  // program code  } |

## 建议

### alert与调试器

调试代码的时候，尽量不要使用 alert 进行显示，尽量使用浏览器提供的高度器功能。

**说明：**

有时使用 alert 进行显示，高度完毕后经常忘记删除这些代码。

### 紧密代码相邻

**说明：**

源程序中关系较为紧密的代码应尽可能相邻。便于程序阅读和查找。

**示例：**

矩形的长与宽关系较密切，放在一起。

|  |
| --- |
| rect.width = 10;  rect.height = 5; |

### 使用简单语句

不要使用难懂的技巧性很高的语句，除非很有必要时。

**说明：**

高技巧语句不等于高效率的程序，实际上程序的效率关键在于算法。

### 异常VS错误码

在程序中使用异常处理还是使用错误返回码处理，根据是否有利于程序结构来确定，并且异常和错误码不应该混合使用，推荐使用异常。

**说明：**

一个系统或者模块应该统一规划异常类型和返回码的含义。

但是不能用异常来做一般流程处理的方式，不要过多地使用异常，异常的处理效率比条件分支低，而且异常的跳转流程难以预测。