**帮助文档**

|  |  |
| --- | --- |
| 产品版本 | 下一代云ERP |
| 系统名称 | KDE |
| 模块名称 | 帮助文档 |

修订记录

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ver. No** | **日期** | **编制/修订** | **审核** | **批准** | **修改的章节号** |
| V1.0 | 2018-12-26 | 陈博佳 |  |  |  |
| V1.1 | 2018-12-29 | 陈博佳 |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

目录

[简介 3](#_Toc533607082)

[实现 3](#_Toc533607083)

[引用 3](#_Toc533607084)

[注释 4](#_Toc533607085)

[变量 5](#_Toc533607086)

[数据类型 7](#_Toc533607087)

[对象 9](#_Toc533607088)

[函数 11](#_Toc533607089)

[运算符 14](#_Toc533607090)

[比较和逻辑运算符 15](#_Toc533607091)

[if...else 语句 16](#_Toc533607092)

[switch 语句 18](#_Toc533607093)

[for 循环 19](#_Toc533607094)

[while 循环 21](#_Toc533607095)

[break 和 continue 语句 22](#_Toc533607096)

[错误 23](#_Toc533607097)

# 简介

**Rhino 仅仅是对javascript核心部分的实现，并没有包含操作HTML的对象或方法。**

Rhino是一个开源的脚本引擎框架，可以运行类似javascript语法的脚本，并可以调用java的方法，并可以嵌入Java执行，脚本修改后不需要重启JVM进程，就可以使用解析或编译方式执行，非常方便。由于Rhino脚本中可以写入任何表达式和javacript程序，既可以进行条件规则的判断，也可以进行各类简单或复杂的计算

Rhino 包括

* JavaScript 1.7的全部特性
* 可以用脚本方式调用Java
* 用一个JavaScript Shell来执行JavaScript脚本
* 用一个JavaScript编译器 来将JavaScript脚本文件转换成Java类文件
* 用一个JavaScript调试器 来调试Rhino执行的脚本

# 实现

KDE提供了表单插件、单据插件、列表插件等10种插件类型，根据不同的插件类型构造不同的插件类，并默认提供了一些初始化方法，当然具体逻辑需要自己实现。

**例子**

|  |
| --- |
| **var plugin = new FormPlugin({**  afterBindData : **function(**e**){**  **},**  afterCreateNewData : **function(**e**){**  **},**  click : **function(**e**){**  **},**  closedCallBack : **function(**e**){**  **},**  confirmCallBack : **function(**e**){**  **},**  itemClick : **function(**e**){**  **},**  registerListener : **function(**e**){**  **}**  **});** |

# 引用

KS脚本支持引用Java项目过程的类、方法，对类加上@KSObject注解，方法加上@KSMethod注解即可。可以通过菜单栏代码—增加引入选择所需的Java类，也可以直接添加require(“完整包路径”)。

**例子**

|  |
| --- |
| require("kd.bos.bill.BillShowParameter");  require("kd.bos.bill.BillOperationStatus");  require("kd.bos.bill.KDBillPlugin"); |

# 注释

**注释**

KS脚本不会执行注释。我们可以添加注释来对 KS 进行解释，或者提高代码的可读性。

单行注释以//开头

下面的例子使用单行注释来解释代码：

**例子**

|  |
| --- |
| //界面初始化，新建数据包完毕后，触发此事件  afterCreateNewData : **function(**e**){**  **},**  //注册控件事件的侦听器，申明捕获控件事件  registerListener : **function(**e**){**  **}** |

**多行注释**

多行注释以 /\* 开始，以 \*/ 结尾。

下面的例子使用多行注释来解释代码：

**例子**

|  |
| --- |
| /\*  \* 界面初始化，新建数据包完毕后，触发此事件；  \* 通常在此事件，设置字段默认值  \*  \* 不能在此事件，设置字段、控件的状态（如锁定性等），应该在afterBindData事件  \*/  afterCreateNewData : **function(**e**){**  **},**  /\*  \* 界面上的控件有事件发生时，会触发此事件。  \* 可以在此事件，注册控件事件的侦听器，申明捕获控件事件。  \*/  registerListener : **function(**e**){**  **}** |

**使用注释来阻止执行**

在下面的例子中，注释用于阻止其中一条代码行的执行（可用于调试）：

**例子**

|  |
| --- |
| registerListener : **function(**e**){**  //this.addClickListeners("button1");  **this**.addClickListeners("button2");  **this**.addClickListeners("button3");  **}** |

在下面的例子中，注释用于阻止代码块的执行（可用于调试）：

**例子**

|  |
| --- |
| registerListener : **function(**e**){**  /\*  this.addClickListeners("button1");  this.addClickListeners("button2");  this.addClickListeners("button3");  \*/  **}** |

**在行末使用注释**

在下面的例子中，把注释放到代码行的结尾处进行解释：

**例子**

|  |
| --- |
| **var** x=5; // 声明 x 并把 5 赋值给它  **var** y=x+2; // 声明 y 并把 x+2 赋值给它 |

# 变量

**变量是存储信息的容器。**

**例子**

|  |
| --- |
| **var** x=2;  **var** y=3;  **var** z=x+y; |

**就像代数那样**

|  |
| --- |
| x=2;  y=3;  z=x+y; |

在代数中，我们使用字母（比如 x）来保存值（比如 2）。

通过上面的表达式 z=x+y，我们能够计算出 z 的值为 5。

在 KS 中，这些字母被称为变量。

提示：您可以把变量看做存储数据的容器。

**变量**

与代数一样，KS 变量可用于存放值（比如 x=2）和表达式（比如 z=x+y）。

变量可以使用短名称（比如 x 和 y），也可以使用描述性更好的名称（比如 age, sum, totalvolume）。

变量必须以字母开头

变量也能以 $ 和 \_ 符号开头（不过我们不推荐这么做）

变量名称对大小写敏感（y 和 Y 是不同的变量）

提示：KS 语句和 KS 变量都对大小写敏感。

**数据类型**

KS 变量还能保存其他数据类型，比如文本值 (name="Bill Gates")。

在 KS 中，类似 "Bill Gates" 这样一条文本被称为字符串。

KS 变量有很多种类型，但是现在，我们只关注数字和字符串。

当您向变量分配文本值时，应该用双引号或单引号包围这个值。

当您向变量赋的值是数值时，不要使用引号。如果您用引号包围数值，该值会被作为文本来处理。

**例子**

|  |
| --- |
| **var** pi=3.14;  **var** name="Bill Gates";  **var** answer='Yes I am!'; |

**声明（创建） KS 变量**

在 KS 中创建变量通常称为“声明”变量。

我们使用 var 关键词来声明变量：

|  |
| --- |
| **var** carname; |

变量声明之后，该变量是空的（它没有值）。

如需向变量赋值，请使用等号：

|  |
| --- |
| carname="Volvo"; |

不过，您也可以在声明变量时对其赋值：

|  |
| --- |
| **var** carname="Volvo"; |

在下面的例子中，我们创建了名为 carname 的变量，并向其赋值 "Volvo"，然后把它放入 “textfield” 的 文本框中：

**例子**

|  |
| --- |
| **var** carname="Volvo";  **this**.getView().showMessage (carname); |

**一条语句，多个变量**

您可以在一条语句中声明很多变量。该语句以 var 开头，并使用逗号分隔变量即可：

|  |
| --- |
| **var** name="Gates", age=56, job="CEO"; |

声明也可横跨多行：

|  |
| --- |
| **var** name="Gates",  age=56,  job="CEO"; |

**Value = undefined**

在计算机程序中，经常会声明无值的变量。未使用值来声明的变量，其值实际上是 undefined。

在执行过以下语句后，变量 carname 的值将是 undefined：

|  |
| --- |
| **var** carname; |

**重新声明 KS 变量**

如果重新声明 KS 变量，该变量的值不会丢失：

在以下两条语句执行后，变量 carname 的值依然是 "Volvo"：

|  |
| --- |
| **var** carname="Volvo";  **var** carname; |

**算数**

您可以通过 KS 变量来做算数，使用的是 = 和 + 这类运算符：

**例子**

|  |
| --- |
| y=5;  x=y+2; |

# 数据类型

字符串、数字、布尔、数组、对象、Null、Undefined

**拥有动态类型**

KS脚本拥有动态类型。这意味着相同的变量可用作不同的类型：

**例子**

|  |
| --- |
| **var** x; // x 为 undefined  **var** x = 6; // x 为数字  **var** x = "Bill"; // x 为字符串 |

**字符串**

字符串是存储字符（比如 "Bill Gates"）的变量。

字符串可以是引号中的任意文本。您可以使用单引号或双引号：

**例子**

|  |
| --- |
| **var** carname="Bill Gates";  **var** carname='Bill Gates'; |

您可以在字符串中使用引号，只要不匹配包围字符串的引号即可：

**例子**

|  |
| --- |
| **var** answer="Nice to meet you!";  **var** answer="He is called 'Bill'";  **var** answer='He is called "Bill"'; |

**数字**

KS 脚本只有一种数字类型。数字可以带小数点，也可以不带：

**例子**

|  |
| --- |
| **var** x1=34.00; //使用小数点来写  **var** x2=34; //不使用小数点来写 |

极大或极小的数字可以通过科学（指数）计数法来书写：

**例子**

|  |
| --- |
| **var** y=123e5; // 12300000  **var** z=123e-5; // 0.00123 |

**布尔**

布尔（逻辑）只能有两个值：true 或 false。

|  |
| --- |
| **var** x=**true**;  **var** y=**false**; |

**数组**

下面的代码创建名为 cars 的数组：

|  |
| --- |
| **var** cars=**new** Array();  cars[0]="Audi";  cars[1]="BMW";  cars[2]="Volvo"; |

或者 (condensed array):

|  |
| --- |
| **var** cars=**new** Array("Audi","BMW","Volvo"); |

或者 (literal array):

|  |
| --- |
| **var** cars=["Audi","BMW","Volvo"]; |

数组下标是基于零的，所以第一个项目是 [0]，第二个是 [1]，以此类推。

同样也可以使用动态数组来创建数组：

|  |
| --- |
| **var** cars=**new** ArrayList();  cars.add("Audi");  cars.add("BMW");  cars.add("Volvo"); |

**对象**

对象由花括号分隔。在括号内部，对象的属性以名称和值对的形式 (name : value) 来定义。属性由逗号分隔：

|  |
| --- |
| **var** person={firstname:"Bill", lastname:"Gates", id:5566}; |

上面例子中的对象 (person) 有三个属性：firstname、lastname 以及 id。

空格和折行无关紧要。声明可横跨多行：

|  |
| --- |
| **var** person={  firstname : "Bill",  lastname : "Gates",  id : 5566  }; |

对象属性有两种寻址方式：

**例子**

|  |
| --- |
| name=person.lastname;  name=person["lastname"]; |

**Undefined 和 Null**

Undefined 这个值表示变量不含有值。

可以通过将变量的值设置为 null 来清空变量。

**例子**

|  |
| --- |
| cars=null;  person=null; |

**声明变量类型**

当您声明新变量时，可以使用关键词 "new" 来声明其类型：

|  |
| --- |
| **var** carname=**new** String;  **var** x= **new** Number;  **var** y= **new** Boolean;  **var** cars= **new** Array;  **var** person= **new** Object; |

KS 变量均为对象。当您声明一个变量时，就创建了一个新的对象。

**类型转换**

KS 是弱类型语言，可以通过类型转换转换成强类型

可以将数字转换成整型、长整型等数据类型。

**例子**

|  |
| --- |
| **var** intNum = toInt(3.1415); //intNum = 3  **var** longNum = toLong(3.1415); //longNum = 3  **var** doubleNum = toDouble(3.1415); //doubleNum = 3.1415  **var** floatNumber = toFloat(3.1415); //floatNum = 3.1415 |

转换成强类型后，还可以使用该数据类型定义的方法。下面例子把数字转换成精度类型，并调用setScale()方法进行四舍五入。

**例子**

|  |
| --- |
| **var** num = toBigDecimal(3.1415926);  **var** num1 = num.setScale(4); //num1 = 3.1416  **var** num2 = num.setScale(5); //num2 = 3.14159 |

上述例子中调用了 setScale(int arg) 方法，arg代表了精度，即精确至小数点后第几位，如 setScale(4) 精确到小数点后第四位，setScale(5) 则精确到小数点后第五位。同时，还有 setScale(int arg1, int arg2) 方法，arg1代表精度，arg2 代表取整方式，主要有以下几种方式：

|  |  |
| --- | --- |
| arg2 | 取整方式 |
| 0 | UP：远离零方向舍入。始终对非零舍弃部分前面的数字加 1。 |
| 1 | DOWN：向零方向舍入。从不对舍弃部分前面的数字加 1（即截尾）。 |
| 2 | CEILING：向正无限大方向舍入。 |
| 3 | FLOOR：向负无限大方向舍入。 |
| 4 | HALF\_UP：向最接近的数字方向舍入，如果与两个相邻数字的距离相等，则向上舍入。 |
| 5 | HALF\_DOWN：向最接近的数字方向舍入，如果与两个相邻数字的距离相等，则向下舍入。 |
| 6 | HALF\_EVEN：向最接近数字方向舍入，如果与两个相邻数字的距离相等，则向相邻的偶数舍入。 |
| 7 | UNNECESSARY：用于断言请求的操作具有精确结果，因此不发生舍入。 |

提示：setScale(int arg) 方法其实是采用了UNNECESSARY 取整方式。

将其他类型转化成字符串。

**例子**

|  |
| --- |
| **var** string1 = toString(3.1415); //string1 = “3.1415”  **var** string2 = toString("This is a String"); //string2 = “This is a String”  **var** string3 = toString(**new** Date()); //string3 = “Fri Dec 28 14:37:38 CST 2018” |

转换成日期，调用日期方法。

**例子**

|  |
| --- |
| **var** date1 = toDate("2018-12-28"); //data1 = “Fri Dec 28 00:00:00 CST 2018”  **var** date2 = toString("2018-12-28 18:12:28"); //date2 = “Fri Dec 28 18:12:28 CST 2018”  **var** date = date2.getDate(); //date = 28  **var** day = date2.getDay(); //day = 5  **var** hour = date2.getHours(); //hour = 18 |

# 对象

**KS中的所有事物都是对象：字符串、数字、数组、日期，等等。**

**在 KS 脚本中，对象是拥有属性和方法的数据。**

**属性和方法**

属性是与对象相关的值。

方法是能够在对象上执行的动作。

举例：汽车就是现实生活中的对象。

汽车的属性：

|  |
| --- |
| car.name=Fiat;  car.model=500;  car.weight=850kg;  car.color=white; |

汽车的方法：

|  |
| --- |
| car.start();  car.drive();  car.brake(); |

汽车的属性包括名称、型号、重量、颜色等。

所有汽车都有这些属性，但是每款车的属性都不尽相同。

汽车的方法可以是启动、驾驶、刹车等。

所有汽车都拥有这些方法，但是它们被执行的时间都不尽相同。

**KS 脚本中的对象**

在 KS 脚本中，对象是数据（变量），拥有属性和方法。

当您像这样声明一个 KS 变量时：

|  |
| --- |
| **var** txt = "Hello"; |

您实际上已经创建了一个 KS 字符串对象。字符串对象拥有内建的属性 length。对于上面的字符串来说，length 的值是 5。字符串对象同时拥有若干个内建的方法。

属性：

|  |
| --- |
| txt.length=5 |

方法：

|  |
| --- |
| txt.indexOf();  txt.replace();  txt.search(); |

**创建 KS 对象**

KS 脚本中的几乎所有事务都是对象：字符串、数字、数组、日期、函数，等等。

你也可以创建自己的对象。

本例创建名为 "person" 的对象，并为其添加了四个属性：

**例子**

|  |
| --- |
| person=**new** Object();  person.firstname="Bill";  person.lastname="Gates";  person.age=56;  person.eyecolor="blue"; |

创建新 KS 对象有很多不同的方法，并且您还可以向已存在的对象添加属性和方法。

**访问对象的属性**

访问对象属性的语法是：

|  |
| --- |
| objectName.propertyName |

本例使用 String 对象的 length 属性来查找字符串的长度：

|  |
| --- |
| **var** message="Hello World!";  **var** x=message.length; |

在以上代码执行后，x 的值是：

|  |
| --- |
| 12 |

**访问对象的方法**

您可以通过下面的语法调用方法：

|  |
| --- |
| objectName.methodName() |

这个例子使用 String 对象的 toUpperCase() 方法来把文本转换为大写：

|  |
| --- |
| **var** message="Hello world!";  **var** x=message.toUpperCase(); |

在以上代码执行后，x 的值是：

|  |
| --- |
| HELLO WORLD! |

# 函数

**函数是由事件驱动的或者当它被调用时执行的可重复使用的代码块。**

**例子**

|  |
| --- |
| **var plugin = new FormPlugin({**  registerListener : **function(**e**) {**  // 侦听按钮点击事件  **this**.addClickListeners("button");  **},**  click : **function(**e**) {**  **var** key = e.getSource().getKey();  **if** (key == "button"){  **this**.myFunction();  }  **},**  **myFunction** : **function (){**  **this**.getView().showMessage("This is my function!");  **}**  **});** |

**函数语法**

函数就是包裹在花括号中的代码块，并使用了关键词function：

|  |
| --- |
| functionname :function(){  这里是要执行的代码  } |

当调用该函数时，会执行函数内的代码。

可以在某事件发生时直接调用函数（比如当用户点击按钮时），并且可在任何位置进行调用。

**调用带参数的函数**

在调用函数时，您可以向其传递值，这些值被称为参数。

这些参数可以在函数中使用。

您可以发送任意多的参数，由逗号 (,) 分隔：

|  |
| --- |
| myFunction : function(argument1,argument2) |

当您声明函数时，请把参数作为变量来声明：

|  |
| --- |
| myFunction : function(var1,var2){  这里是要执行的代码  } |

变量和参数必须以一致的顺序出现。第一个变量就是第一个被传递的参数的给定的值，以此类推。

**例子**

|  |
| --- |
| **var plugin = new FormPlugin({**  registerListener : **function(**e**) {**  // 侦听按钮点击事件  **this**.addClickListeners("button");  **},**  click : **function(**e**) {**  **var** key = e.getSource().getKey();  **if** (key == "button"){  **this**.myFunction('Bill Gates','CEO');  }  **},**  **myFunction : function(**name,job**){**  **this**.getView().showMessage("Welcome " + name + ", the " + job);  **}**  **});** |

上面的函数会当按钮被点击时提示 "Welcome Bill Gates, the CEO"。

函数很灵活，您可以使用不同的参数来调用该函数，这样就会给出不同的消息：

**例子**

|  |
| --- |
| **this**.myFunction('Harry Potter','Wizard');  **this**.myFunction('Bob','Builder'); |

根据您点击的不同的按钮，上面的例子会提示 "Welcome Harry Potter, the Wizard" 或 "Welcome Bob, the Builder"。

**带有返回值的函数**

有时，我们会希望函数将值返回调用它的地方。

通过使用 return 语句就可以实现。

在使用 return 语句时，函数会停止执行，并返回指定的值。

**语法**

|  |
| --- |
| myFunction : function(){  var x=5;  return x;  } |

上面的函数会返回值 5。

注释：整个 KS 并不会停止执行，仅仅是函数。KS 将继续执行代码，从调用函数的地方。

函数调用将被返回值取代：

|  |
| --- |
| **var** myVar=myFunction(); |

myVar 变量的值是 5，也就是函数 "myFunction()" 所返回的值。

即使不把它保存为变量，您也可以使用返回值：

|  |
| --- |
| **this**.getModel().setValue("textfield",**this**.myFunction()); |

" textfield "的文本框将赋值为 5，也就是函数 "myFunction()" 所返回的值。

您可以使返回值基于传递到函数中的参数：

计算两个数字的乘积，并返回结果：

**例子**

|  |
| --- |
| **myFunction : function(**a,b**){**  **return** a\*b;  **}**  **this**.getModel().setValue("textfield",**this**.myFunction(4,3)); |

" textfield "的文本框将赋值为

|  |
| --- |
| 12 |

在您仅仅希望退出函数时 ，也可使用 return 语句。返回值是可选的：

|  |
| --- |
| **myFunction : function(**a,b**){**  **if** (a>b){  **return**;  }  x=a+b  **}** |

如果 a 大于 b，则上面的代码将退出函数，并不会计算 a 和 b 的总和。

**局部 KS 变量**

在 KS 函数内部声明的变量（使用 var）是局部变量，所以只能在函数内部访问它。（该变量的作用域是局部的）。

您可以在不同的函数中使用名称相同的局部变量，因为只有声明过该变量的函数才能识别出该变量。

只要函数运行完毕，本地变量就会被删除。

**全局 KS 变量**

在插件类外声明的变量是全局变量，网页上的所有脚本和函数都能访问它。

**KS 变量的生存期**

KS 变量的生命期从它们被声明的时间开始。

局部变量会在函数运行以后被删除。

全局变量会在页面关闭后被删除。

**向未声明的 KS 变量来分配值**

如果您把值赋给尚未声明的变量，该变量将被自动作为全局变量声明。

这条语句：

|  |
| --- |
| carname="Volvo"; |

将声明一个全局变量 carname，即使它在函数内执行。

# 运算符

**运算符 = 用于赋值。**

**运算符 + 用于加值。**

运算符 = 用于给 KS 变量赋值。

算术运算符 + 用于把值加起来。

|  |
| --- |
| y=5;  z=2;  x=y+z; |

在以上语句执行后，x 的值是 7。

**KS算术运算符**

算术运算符用于执行变量与/或值之间的算术运算。

给定 y=5，下面的表格解释了这些算术运算符：

|  |  |  |  |
| --- | --- | --- | --- |
| 运算符 | 描述 | 例子 | 结果 |
| + | 加 | x=y+2 | x=7 |
| - | 减 | x=y-2 | x=3 |
| \* | 乘 | x=y\*2 | x=10 |
| / | 除 | x=y/2 | x=2.5 |
| % | 求余数(保留整数) | x=y%2 | x=1 |
| ++ | 累加 | x=++y | x=6 |
| -- | 递减 | x=--y | x=4 |

**KS赋值运算符**

赋值运算符用于给 KS 变量赋值。

给定 x=10 和 y=5，下面的表格解释了赋值运算符：

|  |  |  |  |
| --- | --- | --- | --- |
| 运算符 | 例子 | 等价于 | 结果 |
| = | x=y |  | x=5 |
| \*= | X+=y | x=x+y | x=15 |
| -= | X-=y | x=x-y | x=5 |
| \*= | X\*=y | x=x\*y | x=50 |
| /= | x/=y | x=x/y | x=2 |
| %= | X%=y | x=x%y | x=0 |

**用于字符串的 + 运算符**

+ 运算符用于把文本值或字符串变量加起来（连接起来）。

如需把两个或多个字符串变量连接起来，请使用 + 运算符。

|  |
| --- |
| txt1="What a very";  txt2="nice day";  txt3=txt1+txt2; |

在以上语句执行后，变量 txt3 包含的值是 "What a verynice day"。

要想在两个字符串之间增加空格，需要把空格插入一个字符串之中：

|  |
| --- |
| txt1="What a very ";  txt2="nice day";  txt3=txt1+txt2; |

或者把空格插入表达式中：

|  |
| --- |
| txt1="What a very";  txt2="nice day";  txt3=txt1+" "+txt2; |

在以上语句执行后，变量 txt3 包含的值是：

|  |
| --- |
| "What a very nice day" |

**对字符串和数字进行加法运算**

请看这些例子：

|  |
| --- |
| x=5+5;  x="5"+"5";  x=5+"5";  x="5"+5; |

**规则是：**

**如果把数字与字符串相加，结果将成为字符串。**

# 比较和逻辑运算符

**比较和逻辑运算符用于测试 true 或 false。**

**比较运算符**

比较运算符在逻辑语句中使用，以测定变量或值是否相等。

给定 x=5，下面的表格解释了比较运算符：

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 例子 |
| == | 等于 | x==8为false |
| === | 全等(值和类型) | x===5为true;x===”5”为false |
| != | 不等于 | x!=8为true |
| > | 大于 | x>8为false |
| < | 小于 | x<8为true |
| >= | 大于或等于 | x>=8为false |
| <= | 小于或等于 | X<=8为true |

**如何使用**

可以在条件语句中使用比较运算符对值进行比较，然后根据结果来采取行动：

|  |
| --- |
| **if** (age<18) **this**.getView().showMessage("Too young"); |

**逻辑运算符**

逻辑运算符用于测定变量或值之间的逻辑。

给定 x=6 以及 y=3，下表解释了逻辑运算符：

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 例子 |
| && | and | (x < 10 && y > 1) 为 true |
| || | or | (x==5 || y==5) 为 false |
| ! | not | !(x==y) 为 true |

**条件运算符**

KS 还包含了基于某些条件对变量进行赋值的条件运算符。

**语法**

|  |
| --- |
| variablename=(condition)?value1:value2 |

**例子**

|  |
| --- |
| greeting=(visitor=="PRES")?"Dear President ":"Dear "; |

如果变量 visitor 中的值是 "PRES"，则向变量 greeting 赋值 "Dear President "，否则赋值 "Dear"。

# if...else 语句

**条件语句用于基于不同的条件来执行不同的动作。**

**条件语句**

通常在写代码时，您总是需要为不同的决定来执行不同的动作。您可以在代码中使用条件语句来完成该任务。

在 KS 中，我们可使用以下条件语句：

* if 语句 - 只有当指定条件为 true 时，使用该语句来执行代码
* if...else 语句 - 当条件为 true 时执行代码，当条件为 false 时执行其他代码
* if...else if....else 语句 - 使用该语句来选择多个代码块之一来执行
* switch 语句 - 使用该语句来选择多个代码块之一来执行

**if 语句**

只有当指定条件为 true 时，该语句才会执行代码。

**语法**

|  |
| --- |
| if (条件){  只有当条件为 true 时执行的代码  } |

注意：请使用小写的 if。使用大写字母（IF）会生成 KS 错误！

当时间小于 20:00 时，生成一个“Good day”问候：

**例子**

|  |
| --- |
| **if** (time<20){  x="Good day";  } |

x 的结果是：

|  |
| --- |
| Good day |

请注意，在这个语法中，没有 ..else..。您已经告诉浏览器只有在指定条件为 true 时才执行代码。

**if...else 语句**

请使用 if....else 语句在条件为 true 时执行代码，在条件为 false 时执行其他代码。

**语法**

|  |
| --- |
| if (条件){  当条件为 true 时执行的代码  }  else{  当条件不为 true 时执行的代码  } |

当时间小于 20:00 时，将得到问候 "Good day"，否则将得到问候 "Good evening"。

**例子**

|  |
| --- |
| **if** (time<20){  x="Good day";  }  **else**{  x="Good evening";  } |

x 的结果是：

|  |
| --- |
| Good day |

**if...else if...else 语句**

使用 if....else if...else 语句来选择多个代码块之一来执行。

**语法**

|  |
| --- |
| if (条件 1){  当条件 1 为 true 时执行的代码  } else if (条件 2){  当条件 2 为 true 时执行的代码  } else{  当条件 1 和 条件 2 都不为 true 时执行的代码  } |

如果时间小于 10:00，则将发送问候 "Good morning"，否则如果时间小于 20:00，则发送问候 "Good day"，否则发送问候 "Good evening"：

**例子**

|  |
| --- |
| **if** (time<10){  x="Good morning";  } **else if** (time<20){  x="Good day";  } **else**{  x="Good evening";  } |

x 的结果是：

|  |
| --- |
| Good day |

# switch 语句

**switch 语句用于基于不同的条件来执行不同的动作。**

**KS Switch 语句**

请使用 switch 语句来选择要执行的多个代码块之一。

**语法**

|  |
| --- |
| switch(n){  case 1:  执行代码块 1  break;  case 2:  执行代码块 2  break;  default:  n 与 case 1 和 case 2 不同时执行的代码  } |

工作原理：首先设置表达式 n（通常是一个变量）。随后表达式的值会与结构中的每个 case 的值做比较。如果存在匹配，则与该 case 关联的代码块会被执行。请使用 break 来阻止代码自动地向下一个 case 运行。

显示今日的周名称。请注意 Sunday=0, Monday=1, Tuesday=2, 等等：

**例子**

|  |
| --- |
| **switch** (day){  **case** 0:  x="Today it's Sunday";  **break**;  **case** 1:  x="Today it's Monday";  **break**;  **case** 2:  x="Today it's Tuesday";  **break**;  **case** 3:  x="Today it's Wednesday";  **break**;  **case** 4:  x="Today it's Thursday";  **break**;  **case** 5:  x="Today it's Friday";  **break**;  **case** 6:  x="Today it's Saturday";  **break**;  } |

x 的结果：

|  |
| --- |
| Today it's Wednesday |

**default 关键词**

请使用 default 关键词来规定匹配不存在时做的事情：

如果今天不是周六或周日，则会输出默认的消息：

**例子**

|  |
| --- |
| **switch** (day){  **case** 6:  x="Today it's Saturday";  **break**;  **case** 0:  x="Today it's Sunday";  **break**;  **default**:  x="Looking forward to the Weekend";  } |

x 的结果：

|  |
| --- |
| Looking forward to the Weekend |

# for 循环

**循环可以将代码块执行指定的次数。**

**KS 循环**

如果您希望一遍又一遍地运行相同的代码，并且每次的值都不同，那么使用循环是很方便的。

我们可以这样遍历数组的值：

|  |
| --- |
| text = text + cars[0];  text = text + cars[1];  text = text + cars[2];  text = text + cars[3];  text = text + cars[4];  text = text + cars[5]; |

不过通常我们这样写：

|  |
| --- |
| **for**(**var** i=0;i<cars.length;i++){  text = text + cars[i];  } |

**不同类型的循环**

KS 支持不同类型的循环：

* for - 循环代码块一定的次数
* for/in - 循环遍历对象的属性
* while - 当指定的条件为 true 时循环指定的代码块
* do/while - 同样当指定的条件为 true 时循环指定的代码块

**for 循环**

for 循环是您在希望创建循环时常会用到的工具。

下面是 for 循环的语法：

|  |
| --- |
| for (语句 1; 语句 2; 语句 3){  被执行的代码块  } |

**语句 1** 在循环（代码块）开始前执行

**语句 2** 定义运行循环（代码块）的条件

**语句 3** 在循环（代码块）已被执行之后执行

**例子**

|  |
| --- |
| **for** (**var** i=0; i<5; i++){  x=x + i;  } |

从上面的例子中，您可以看到：

* Statement 1 在循环开始之前设置变量 (var i=0)。
* Statement 2 定义循环运行的条件（i 必须小于 5）。
* Statement 3 在每次代码块已被执行后增加一个值 (i++)。

**语句 1**

通常我们会使用语句 1 初始化循环中所用的变量 (var i=0)。

语句 1 是可选的，也就是说不使用语句 1 也可以。

您可以在语句 1 中初始化任意（或者多个）值：

**例子**

|  |
| --- |
| **for** (var i=0,len=cars.length; i<len; i++){  text = text + cars[i];  } |

同时您还可以省略语句 1（比如在循环开始前已经设置了值时）：

**例子**

|  |
| --- |
| var i=2,len=cars.length;  **for** (; i<len; i++){  text = text + cars[i];  } |

**语句 2**

通常语句 2 用于评估初始变量的条件。

语句 2 同样是可选的。

如果语句 2 返回 true，则循环再次开始，如果返回 false，则循环将结束。

提示：如果您省略了语句 2，那么必须在循环内提供 break。否则循环就无法停下来。

**语句 3**

通常语句 3 会增加初始变量的值。

语句 3 也是可选的。

语句 3 有多种用法。增量可以是负数 (i--)，或者更大 (i=i+15)。

语句 3 也可以省略（比如当循环内部有相应的代码时）：

**例子**

|  |
| --- |
| **var** i=0,len=cars.length;  **for** (; i<len; ){  text = text + cars[i];  i++;  } |

**for/in 循环**

ks for/in 语句循环遍历对象的属性：

**例子**

|  |
| --- |
| **var** person={fname:"John",lname:"Doe",age:25};  **for** (x **in** person){  txt=txt + person[x];  } |

# while 循环

**只要指定条件为 true，循环就可以一直执行代码。**

**while 循环**

While 循环会在指定条件为真时循环执行代码块。

**语法**

|  |
| --- |
| while (条件){  需要执行的代码  } |

本例中的循环将继续运行，只要变量 i 小于 5：

**例子**

|  |
| --- |
| **while** (i<5){  x=x + i;  i++;  } |

提示：如果您忘记增加条件中所用变量的值，该循环永远不会结束。该可能导致浏览器崩溃。

**do/while 循环**

do/while 循环是 while 循环的变体。该循环会执行一次代码块，在检查条件是否为真之前，然后如果条件为真的话，就会重复这个循环。

**语法**

|  |
| --- |
| do{  需要执行的代码  }  while (条件); |

下面的例子使用 do/while 循环。该循环至少会执行一次，即使条件是 false，隐藏代码块会在条件被测试前执行：

**例子**

|  |
| --- |
| **do**{  x=x i;  i++;  }  **while** (i<5); |

别忘记增加条件中所用变量的值，否则循环永远不会结束！

**比较 for 和 while**

如果您已经阅读了前面那一章关于 for 循环的内容，您会发现 while 循环与 for 循环很像。

**for 语句实例**

本例中的循环使用 for 循环来显示 cars 数组中的所有值：

|  |
| --- |
| cars=["BMW","Volvo","Saab","Ford"];  **var** i=0;  **var** text=””;  **for** (;cars[i];){  text = text + cars[i];  i++;  } |

**while 语句实例**

本例中的循环使用使用 while 循环来显示 cars 数组中的所有值：

|  |
| --- |
| cars=["BMW","Volvo","Saab","Ford"];  **var** i=0;  **var** text=””;  **while** (cars[i]){  text = text + cars[i];  i++;  } |

# break 和 continue 语句

**break 语句用于跳出循环。**

**continue 用于跳过循环中的一个迭代。**

**break 语句**

我们已经在本教程稍早的章节中见到过 break 语句。它用于跳出 switch() 语句。

break 语句可用于跳出循环。

break 语句跳出循环后，会继续执行该循环之后的代码（如果有的话）：

**例子**

|  |
| --- |
| **for** (i=0;i<10;i++){  **if** (i==3){  **break**;  }  x=x + i;  } |

由于这个 if 语句只有一行代码，所以可以省略花括号：

|  |
| --- |
| **for** (i=0;i<10;i++){  if (i==3) break;  x=x + i;  } |

**continue 语句**

continue 语句中断循环中的迭代，如果出现了指定的条件，然后继续循环中的下一个迭代。

该例子跳过了值 3：

**例子**

|  |
| --- |
| **for** (i=0;i<=10;i++){  **if** (i==3) continue;  x=x + i;  } |

**KS 标签**

如需标记 KS 语句，请在语句之前加上冒号：

|  |
| --- |
| label:  语句 |

break 和 continue 语句仅仅是能够跳出代码块的语句。

**语法**

|  |
| --- |
| break labelname;  continue labelname; |

continue 语句（带有或不带标签引用）只能用在循环中。

break 语句（不带标签引用），只能用在循环或 switch 中。

通过标签引用，break 语句可用于跳出任何 KS 代码块：

**例子**

|  |
| --- |
| cars=["BMW","Volvo","Saab","Ford"];  list:{  text = text +cars[0];  text = text +cars[1];  text = text +cars[2];  **break** list;  text = text +cars[3];  text = text +cars[4];  } |

# 错误

**try** 语句测试代码块的错误。

**catch** 语句处理错误。

**throw** 语句创建自定义错误。

**错误一定会发生**

当 KS 引擎执行 KS 代码时，会发生各种错误：

* 可能是语法错误，通常是程序员造成的编码错误或错别字。
* 可能是拼写错误或语言中缺少的功能（可能由于浏览器差异）。
* 可能是由于来自服务器或用户的错误输出而导致的错误。
* 当然，也可能是由于许多其他不可预知的因素。

**KS 抛出错误**

当错误发生时，当事情出问题时，KS 引擎通常会停止，并生成一个错误消息。

描述这种情况的技术术语是：KS 将抛出一个错误。

**KS 测试和捕捉**

**try** 语句允许我们定义在执行时进行错误测试的代码块。

**catch** 语句允许我们定义当 try 代码块发生错误时，所执行的代码块。

ks 语句 **try** 和 **catch** 是成对出现的。

**语法**

|  |
| --- |
| try{  //在这里运行代码  } catch(err)  {  //在这里处理错误  } |

在下面的例子中，我们故意在 try 块的代码中写错了方法名。

catch 块会捕捉到 try 块中的错误，并执行代码来处理它。

**例子**

|  |
| --- |
| **var plugin = new FormPlugin({**  registerListener : **function(**e**) {**  // 侦听按钮点击事件  **this**.addClickListeners("button");  **},**  click : **function(**e**) {**  **var** key = e.getSource().getKey();  **try** {  **this**.getModel().setValue("textfield",this.myFunc(3,4));  } **catch**(err) {  txt="There was an error on this page.\n\n";  txt+="Error description: " + err.message + "\n\n";  txt+="Click OK to continue.\n\n";  **this**.getView().showMessage(txt);  }  **},**  **myFunction(**a,b**){**  **return a\*b;**  **}**  **});** |

**throw 语句**

throw 语句允许我们创建自定义错误。

正确的技术术语是：创建或**抛出异常**（exception）。

如果把 throw 与 try 和 catch 一起使用，那么您能够控制程序流，并生成自定义的错误消息。

**语法**

|  |
| --- |
| throw exception |

异常可以是 KS 字符串、数字、逻辑值或对象。

本例检测输入变量的值。如果值是错误的，会抛出一个异常（错误）。catch 会捕捉到这个错误，并显示一段自定义的错误消息：

**例子**

|  |
| --- |
| **var plugin = new FormPlugin({**  registerListener : **function(**e**) {**  // 侦听按钮点击事件  **this**.addClickListeners("button");  **},**  click : **function(**e**) {**  **var** key = e.getSource().getKey();  **this**.myFunction();  **},**  **myFunction(){**  **try** {  **var** x =**this**.getModel().getValue("textfield");  **if**(x=="") **throw** "empty";  **if**(isNaN(x)) **throw** "not a number";  **if**(x>10) **throw** "too high";  **if**(x<5) **throw** "too low";  } **catch**(err){  **this**.getView().showMessage(err);  }  **}**  **});** |

# 样例清单

## 1、打开页面初始化之前验证权限

## 2、打开页面初始化文本框的值

|  |
| --- |
| **var plugin = new FormPlugin({**  afterBindData : **function(e){**    **},**  afterCreateNewData : **function(e) {**  **this**.getModel().setValue("textfield",”这是一个文本框”);  **},**  click : **function(e){**    **},**  closedCallBack : **function(e){**    **},**  confirmCallBack : **function(e){**    **},**  itemClick : **function(e){**    **},**  registerListener : **function(e){**    **}**  **});** |

## 3、打开页面初始化下拉框的值

|  |
| --- |
| **var plugin = new FormPlugin({**  afterBindData : **function(e){**    **},**  afterCreateNewData : **function(e) {**  **this**.getModel().setValue("combofield",1);  **},**  click : **function(e){**    **},**  closedCallBack : **function(e){**    **},**  confirmCallBack : **function(e){**    **},**  itemClick : **function(e){**    **},**  registerListener : **function(e){**    **}**  **});** |

## 4、打开页面初始化单据体，并默认选择2条记录

|  |
| --- |
| **var plugin = new FormPlugin({**  afterBindData : **function(e){**    **},**  afterCreateNewData : **function(e) {**  //初始化单据体  **this**.getModel().batchCreateNewEntryRow("entryentity", 5);  **for** (**var** i=0; i<5; i++) {  **this**.getModel().setValue("textfield1", "这是一条单据体记录"+toInt(i+1),i);  **this**.getModel().setValue("date", **new** Date(),i);  }  //选中两条记录  **var** grid = **this**.getControl("entryentity");  **var** items = [0,1];  grid.selectRows(items,0);  **},**  click : **function(e){**    **},**  closedCallBack : **function(e){**    **},**  confirmCallBack : **function(e){**    **},**  itemClick : **function(e){**    **},**  registerListener : **function(e){**    **}**  **});** |

## 5、打开页面初始化树

|  |
| --- |
| require("kd.bos.entity.tree.TreeNode");  **var plugin = new FormPlugin({**  afterBindData : **function(e){**    **},**  afterCreateNewData : **function(e) {**  **var** tree = **this**.getView().getControl("treeviewap");  **var** root = **new** TreeNode("","rootnode","这是根节点","root");  **var** childnode1 = **new** TreeNode("","childnode1","这是子节点1",**true**,"node");  **var** leafnode1 = **new** TreeNode("childnode1","leafnode1","这是叶子节点1",**true**);  **var** leafnode2 = **new** TreeNode("childnode1","leafnode2","这是叶子节点2",**true**);  **var** leafnode3 = **new** TreeNode("childnode1","leafnode3","这是叶子节点3",**true**);  childnode1.addChild(leafnode1);  childnode1.addChild(leafnode2);  childnode1.addChild(leafnode3);  **var** childnode2 = **new** TreeNode("","childnode2","这是子节点2",**true**,"node");  **var** leafnode4 = **new** TreeNode("childnode2","leafnode4","这是叶子节点4",**false**);  **var** leafnode5 = **new** TreeNode("childnode2","leafnode5","这是叶子节点5",**false**);  **var** leafnode6 = **new** TreeNode("childnode2","leafnode6","这是叶子节点6",**false**);  childnode2.addChild(leafnode4);  childnode2.addChild(leafnode5);  childnode2.addChild(leafnode6);  **var** childnode3 = **new** TreeNode("","childnode3","这是子节点3",**false**,"node");  root.addChild(childnode1);  root.addChild(childnode2);  root.addChild(childnode3);  tree.addNode(root);  root.setIsOpened(**true**);  **},**  click : **function(e){**    **},**  closedCallBack : **function(e){**    **},**  confirmCallBack : **function(e){**    **},**  itemClick : **function(e){**    **},**  registerListener : **function(e){**    **}**  **});** |

## 6、注册按钮点击事件，并拦截按钮的点击事件

|  |
| --- |
| **var plugin = new FormPlugin({**  afterBindData : **function(e){**    **},**  afterCreateNewData : **function(e) {**    **},**  registerListener : **function(e){**  //注册按钮点击事件  **this**.addClickListeners("button");  **},**  click : **function(e){**  //拦截按钮点击事件  **var** key = e.getSource().getKey();  **if** (key == "button"){  //按钮点击事件  }  **},**  closedCallBack : **function(e){**    **},**  confirmCallBack : **function(e){**    **},**  itemClick : **function(e){**    **}**  **});** |

## 7、注册工具栏的点击事件，并拦截工具栏的点击事件

## 8、注册单据体的点击事件，并拦截单据体的点击事件

## 9、注册树的点击事件，并拦截树的点击事件

## 10、获取文本框的值

## 11、获取下拉框的值

## 12、获取单据体某一行的值

## 13、获取单据体某一行某个单元格的值

## 14、获取当前点击树节点的值

## 15、弹个新页面，关闭页面时，将值带回来