

用 TaskBuilder 快速学习 JavaScript 基础语法

作者：任讯熹哥



技术支持群

www.taskbuilder.org

TaskBuilder 定位是“低代码”开发工具，不是“零代码”开发，所以在使用 TaskBuilder 开发的过程中，多多少少会与一些代码打交道，为了让没有任何编程基础的用户也能用 TaskBuilder 进行开发，本章把一些 JavaScript 编程的基础知识简单介绍一下，但不会介绍得特别详尽，目的只是为了让大家了解一些基本的概念，如果要系统性的学习这些知识，请自行查看相关资料。

JavaScript 语言简介

JavaScript 是一种脚本语言，简称 js，原来主要运行在浏览器环境中(前端)，用来控制网页元素进行一些互动，后来 Node.js 横空出世，让 JavaScript 在服务器端(后端或后台)也能运行了，这样就能做到只要掌握一种语言，就可以进行 Web 系统的前后端开发，不必再像以前那样，前端用 JavaScript，后端还得用 PHP、Java、C#等其他语言，大大降低了开发 Web 系统的难度，也正是看中这一点，我们整个低代码开发平台都是采用的 JavaScript 进行开发。

下面给大家介绍一下 JavaScript 语言的基本概念：

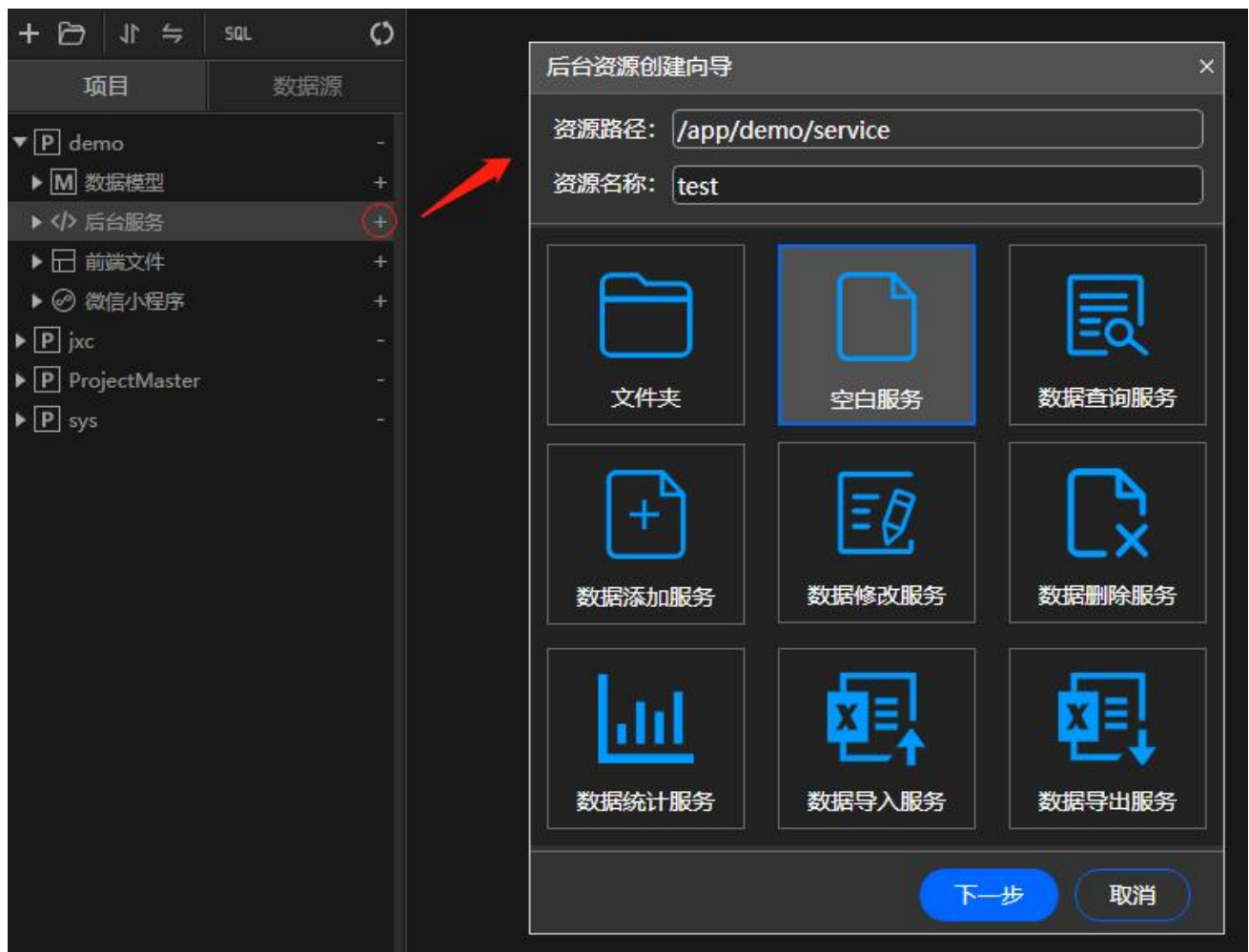
准备工作

在介绍本章的编程基础知识之前，先教大家怎么新建一个空白的后台服务，然后编辑、保存并测试，本章后续介绍的各种理论知识会在这个后台服务中进行实践操作，以便大家能更好地掌握这些知识点。

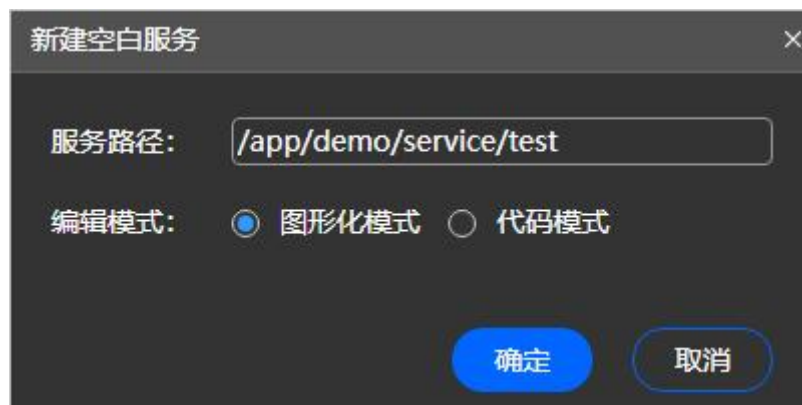
1、新建空白服务

新建空白服务的操作步骤如下：

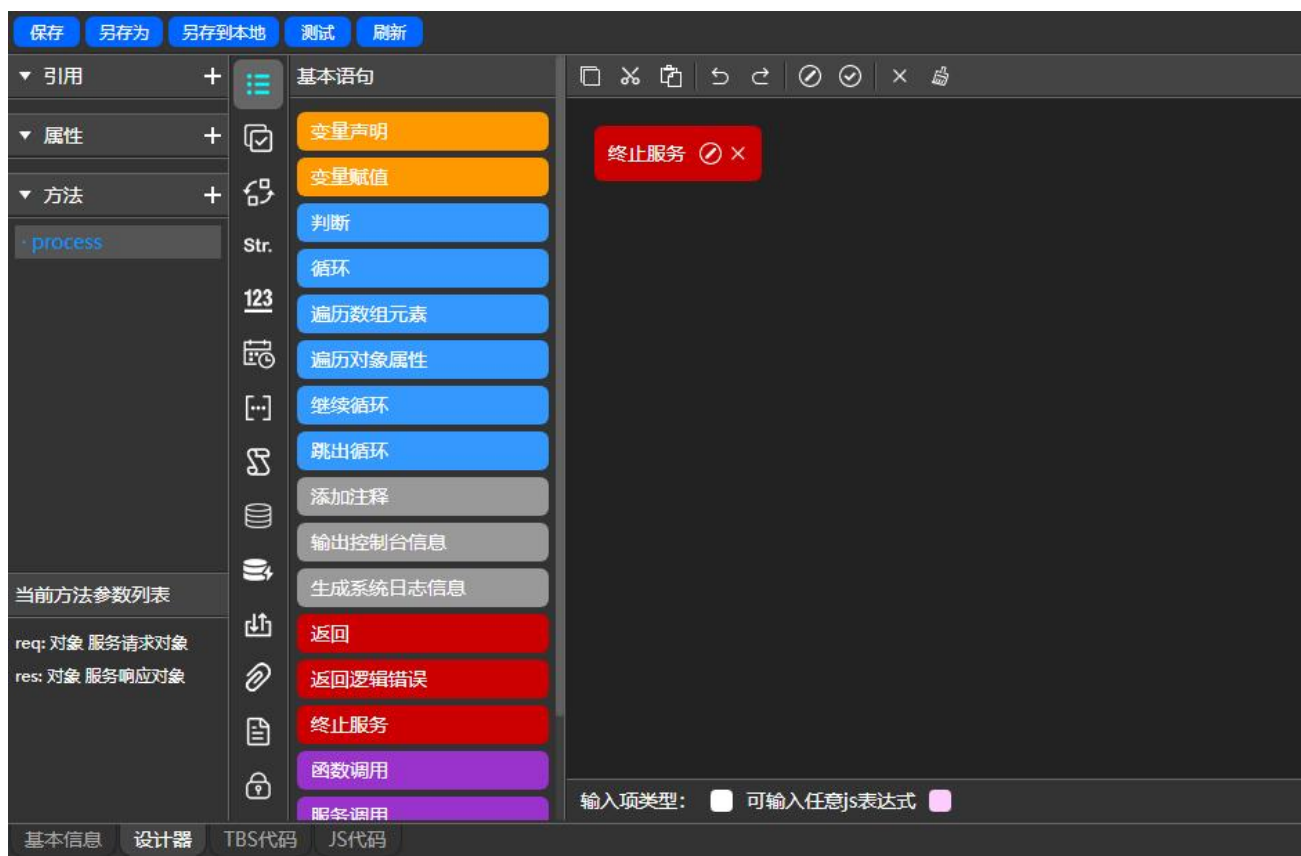
在系统资源管理器内，点击项目或产品里后台服务节点右侧的加号，可以打开 **后台资源创建向导**，如下图所示：



在该对话框中，输入资源名称（即后台服务文件名），在下面的资源类型列表中
表中选择 **空白服务**，点击 **下一步** 按钮，弹出如下对话框：



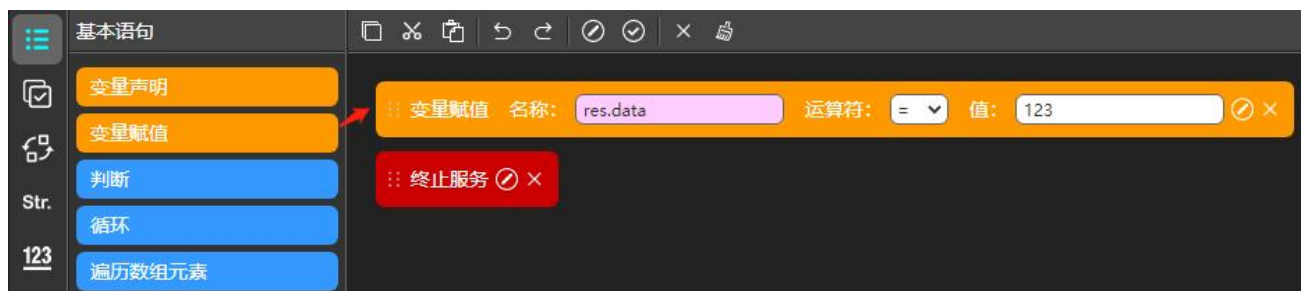
在该对话框中，编辑模式选 **图形化模式**，然后点击 **确定** 按钮，即可创建
可以图形化编辑的空白服务，并打开 **后台服务编辑器** 显示该服务的内容，如下
图所示：



在该编辑器中，默认仅放置了一个 **终止服务** 的操作，开发者可以根据需要拖拽设置该后台服务的详细业务逻辑。

2、编辑服务的业务逻辑

从 **后台服务设计器** 语句库中找到要使用的语句类型，将鼠标放在该语句上面，然后按住鼠标左键，并拖拽到右侧的业务逻辑设计区适当的位置，然后松开鼠标，即可新建一条该类型的语句，每条语句会显示为一个矩形条，在该矩形条上提供了相关参数可供设置，不同的语句，参数不一样，开发者可以根据需要进行设置。



3、保存服务

设置好各个语句的参数后，点击工具上的 **保存** 按钮，即可将设置的内容保存起来。



4、测试服务

点击工具栏上的 **测试** 按钮，可以打开 **后台服务测试工具**，点击该工具顶部的 **发送请求** 按钮，即可模拟客户端请求该后台服务，并在底部查看服务响应结果，如下图所示：



关键字和保留字

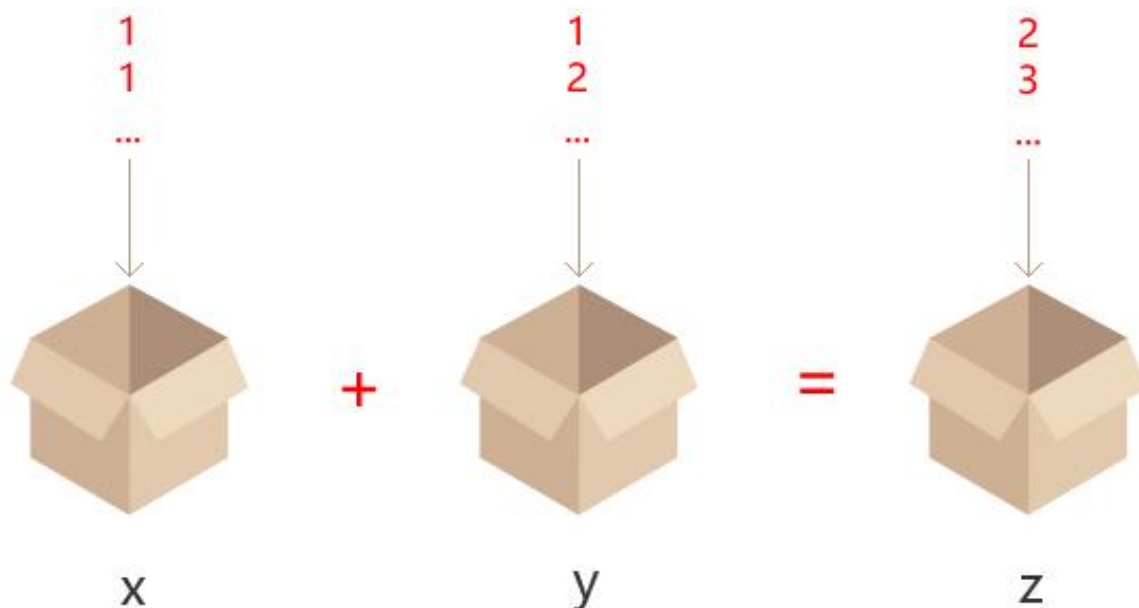
指 JavaScript 语言本身需要用到的一些词汇, 例如: break、case、catch、continue、default、delete、do、else、finally、for、function、if、in、instanceof、new、return、switch、this、throw、try、typeof、var、let、void、while、with、Date、Array、Object、null、undefined、NaN、boolean、byte、char、class、const、debugger、double、enum、export、extends、final、float、goto、implements、import、int、interface、long、native、package、private、protected、public、short、static、super、synchronized、throws、transient、volatile 等。

开发者自己定义的变量和函数名不能使用这些关键字。

变量

在 JavaScript 中, 0、true、"TaskBuilder"这种确定的数据称为 **值**, 如果值不确定, 则可以使用 **变量**, **变量** 就是指值可以变的量, 可以简单地将变量比作装东西的盒子, 用来在程序执行的过程中临时存储某个值, 这样就可以在程序代码里, 先用变量名把计算公式写好, 然后在实际执行程序时, 给变量赋不同的值, 就能得出不同的结果。

举个最简单的例子, 如果我们要编写一个实现计算任意两个数相加之和的程序, 就需要定义三个变量, 例如 x、y 和 z, x 和 y 分别存储两个加数, z 存储计算结果, 实现该运算的代码就是: $z=x+y$, 这样只要给 x 和 y 赋一个具体的数值, 该程序就能自动计算出他们的和, 这种可以根据变量不同的值计算出不同结果的能力是计算机程序最大的功能特点。如果 x 和 y 都是固定的值, 例如都是 1, 那就没有必要用程序去计算了, 幼儿园小朋友都能回答。



1、变量声明

在使用变量前，需要先声明变量，给变量设置一个名称，否则，后续代码中无法使用该变量，JavaScript 的变量声明语句格式为：

var 变量名;

其中，var (variable 的缩写) 是 JavaScript 的声明变量的关键字（也可以使用 let，具体区别可以自行查看相关资料）。

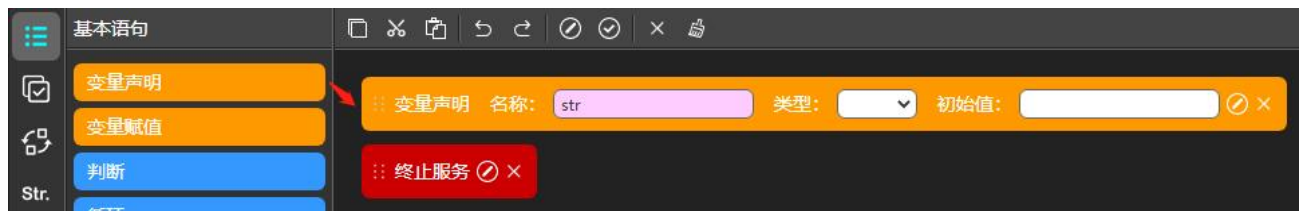
注意：

- 1、变量名只能是字母、数字或下划线，且不能以数字开头。
- 2、变量名不能使用 JavaScript 的关键字，例如：var、let、for、while、null 等。
- 3、JavaScript 的变量名是区分大小写的，变量 a 和变量 A 不是同一个变量。
- 4、变量命名建议采用驼峰命名法，即首字母小写，后面的每个单词首字母大写，例如：myFirstName。

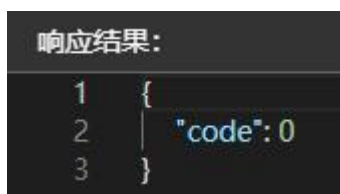
另外，对象的属性其实也是一种变量，后面介绍对象这个概念时会讲到。

下面，我们在 TaskBuilder 里实际操作一下，打开前面创建的那个空白的后台服务，从语句库的基本语句中，用鼠标左键按住 **变量声明** 语句并拖拽到业务逻辑设计区，然后松开鼠标，就会在业务逻辑设计区创建一个变量声明语句的设

置面板, 在该设置面板的 **名称** 输入框中输入变量名, 即可完成变量的声明, 如下图所示:



点击工具栏上的 **保存** 按钮保存修改, 然后点 **测试** 按钮, 打开 后台服务测试工具, 再点该工具里的 **发送请求** 按钮, 这时可以在底部的响应结果里看到如下所示的内容:



该响应内容是一个 JSON 格式的对象, 这是由引擎服务器在加载并运行后台服务后返回的结果, 里面默认都会有一个名为 code 的属性, 默认值为 0, 表示请求成功, 如果不等于 0, 则可能是发生了错误, 会有一个名为 message 的对象描述具体的错误原因。

由于在上面的后台服务里, 我们没有在服务响应对象里返回任何数据, 所以, 响应结果里除了这个默认的 code 属性, 没有其他任何信息, 属于正常情况。

2、变量赋值

光定义变量没有用, 还得给它赋值, 才能用它进行实际的运算, 变量赋值的语句格式如下:

变量名 = 变量值;

也可以在声明变量时直接赋一个初始值, 格式如下:

var 变量名 = 变量值;

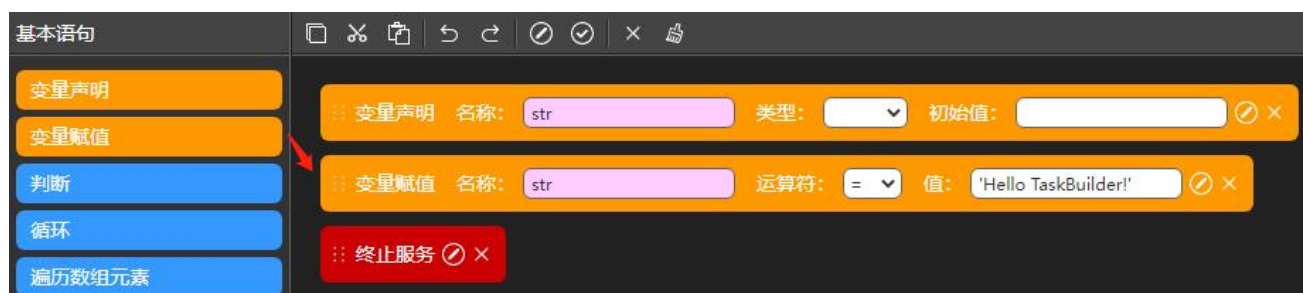
赋值时, 等于号右边的变量值不是非得是具体的值, 也可以是其他变量, 或者任意有计算结果的表达式或函数调用语句。

JavaScript 的变量是弱类型的, 就是没有固定的数据类型限制, 可以保存任

意类型的数据，也可以随时修改为其他类型的数据。

如果变量声明后没有赋值，则该变量的值为 undefined。

下面，我们在 TaskBuilder 里实际操作一下，打开前面创建的那个空白的后台服务，从语句库的基本语句中，用鼠标左键按住 **变量赋值** 语句并拖拽到业务逻辑设计区，然后松开鼠标，就会在业务逻辑设计区创建一个变量赋值语句的设置面板，在该设置面板的 **名称** 输入框中输入上面声明的变量名，在 **值** 输入框中输入变量值，即可完成变量的赋值，如下图所示：



上面的这两条语句实际上也可以合并为如下图所示的一条语句：



这样就相当于在变量声明时直接为变量设置了一个初始值。但如果变量的值在声明时并没有确定，而是需要经过后续一些其他语句的运算才能确定，则后面还是需要使用赋值语句设置变量的值。

为了测试上面的变量声明和赋值语句是否设置正确，我们可以再拖拽一个赋值语句，把上面这个变量的值保存到服务响应对象 `res` 里（后面会介绍对象的概念），返回给客户端，如下图所示：



然后保存该服务，并点击工具栏上的 **测试** 按钮，打开 **后台服务测试工具**，

再点击 **发送请求** 按钮, 就可以看到如下图所示的响应结果:

响应结果:

```
1 {
2   "code": 0,
3   "data": "Hello TaskBuilder!"
4 }
```

可以看到, str 变量声明和赋值成功, 并通过 res 对象将变量值返回给了客户端。

3、变量作用域

变量的作用域就是指能使用该变量的范围, 可以分为以下三种类型:

3.1 全局作用域

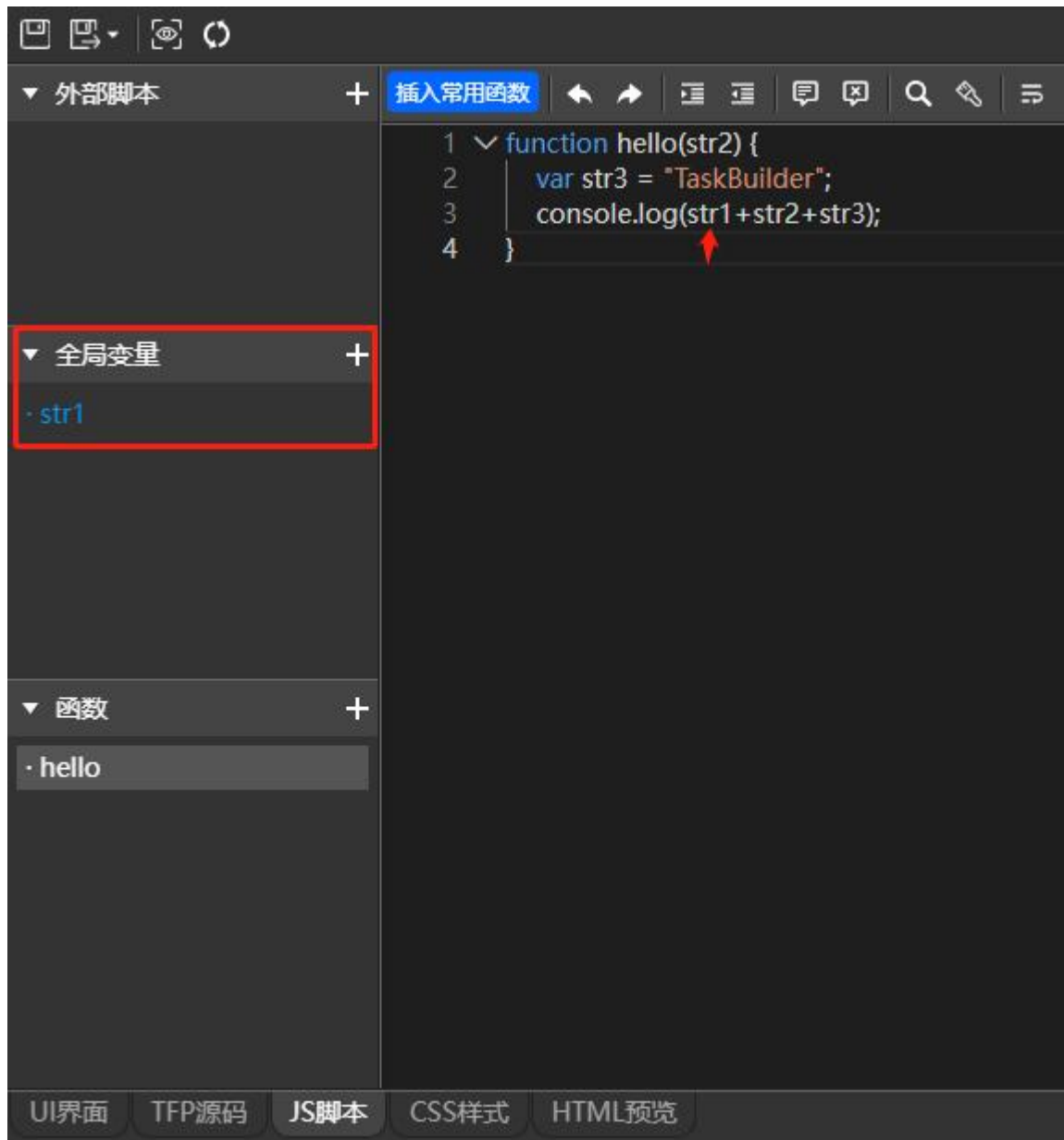
不属于任何函数或代码块的变量, 称为全局变量, 可以在当前代码文件中的任意位置访问, 例如下面代码中的变量 str1 就属于全局变量, 在以下代码的任何位置都可以使用, 而 str2 和 str3 就不是全局变量, 只能在 hello 函数中使用:

```
var str1 = "Hello";

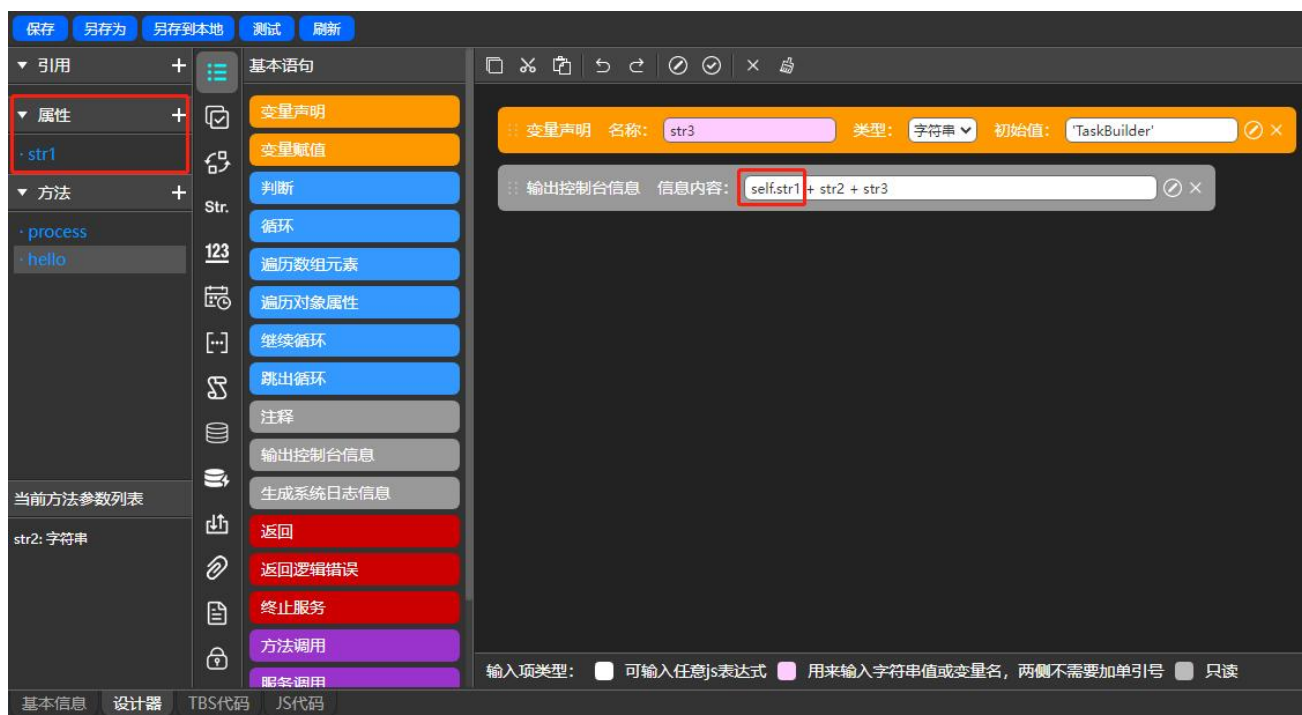
function hello(str2) {
  var str3 = "TaskBuilder!";
  console.log(str1+str2+str3);
}

hello(" "); //输出结果: Hello TaskBuilder!
```

在前端页面的 JS 脚本编辑器中, 可以定义全局变量, 然后在整个页面的所有函数中都可以使用这个变量, 如下图所示:



在后台服务中，后台服务对象的属性也可以当着全局变量使用，只不过在使用时，需要在属性名前面加上 `self.`，如下图所示：



这里的 self 表示当前后台服务对象。

3.2 局部作用域

局部作用域也叫函数作用域, 对应的范围就是各个函数的函数体内的所有代码 (位于函数定义语句的 `{ }` 之间的代码)。函数的参数 (在函数调用语句的括号内传入的参数), 或者在函数内定义的变量, 只能在该函数的函数体内访问, 例如上面介绍全局作用域时提供的示例代码中的 `str2` 和 `str3` 就属于局部变量, 只能在 `hello` 这个函数内部使用。同样, 在上面介绍变量声明和变量赋值语句时, 在后台服务默认提供的 `process` 方法里, 该方法的 `req` 和 `res` 参数, 包括在这个方法里声明的那个 `str` 变量, 都属于局部变量, 只能这个 `process` 方法里使用, 因为方法也是函数, 后面讲解对象这个概念时会进行介绍。

3.3 块作用域

在某个代码块里定义的变量, 只能在这个代码块内使用, 其作用域就属于块作用域, 例如下面 `for` 循环中的变量 `i` 和 `j` 的作用域就属于块作用域, 只能在这个 `for` 循环中使用:

```
for(var i=1; i<10; i++) {  
    var j = i % 2; //求余数  
    if(j==0) {  
        console.log(i+"是偶数");  
    } else {  
        console.log(i+"是奇数");  
    }  
}
```

以上代码在介绍 for 循环时，再告诉大家怎么在 TaskBuilder 的后台服务设计器里实现。

数据类型

为了更好地处理数据，JavaScript 将数据分成了几种类型，不同的类型提供了不同的方法或属性，例如字符串类型的数据可以进行查找、替换和截取等操作，可以获得字符串长度；数值类型的数据可以进行加减乘除运算；布尔值可以进行判断；日期值可以获得当前年份、月份和日期等。这些数据类型又分为简单数据类型和复杂数据类型两大类，下面分别介绍一下：

1、简单数据类型

简单数据类型的数据在赋值给变量时，可以直接赋值，例如：

```
var str = 'abc';  
var i = 123;  
var isExists = false;
```

简单数据类型包括以下几种：

1.1 字符串 (String)

表示一个或多个字符,字符串值两边要用英文双引号或单引号引起来,例如:
"abc" 或 'abc' 都可以,但建议在 TaskBuilder 的可视化设置界面中,在需要设置字符串值的地方,尽量使用单引号,因为不管是前端页面,还是后台服务,都会保存为 JSON 格式的文件,而 JSON 格式规定,属性名称和字符串值两边都必须用双引号,如果在前端页面或后台服务的各项设置中也使用双引号,则可能会产生冲突。

多个字符串值或变量可以用 + 号进行拼接,例如:

```
var str1 = "Hello";  
var str2 = "TaskBuilder";  
var str3 = str1 + " " + str2 + "!";
```

JavaScript 语言提供了丰富的字符串处理函数,可以对字符串进行截取、查找、替换等操作,示例代码如下:

```
var str = "Hello TaskBuilder!";  
var index = str.indexOf("TaskBuilder"); //查找字符串 "TaskBuilder"  
在 str 变量中的位置  
var str2 = str.substring(0, index); //截取 str 变量的值,只保留 "Hello "
```

为了方便大家对字符串进行操作,在 TaskBuilder 的前端页面设计器和后台服务设计器的语句库里都提供了丰富的字符串处理函数,支持用图形化的形式对字符串值或变量进行相关处理,如下图所示:



上面字符串查找和替换的代码, 在 TaskBuilder 的后台服务设计器里用图形化的语句设置的界面如下图所示:



测试该服务的结果如下:

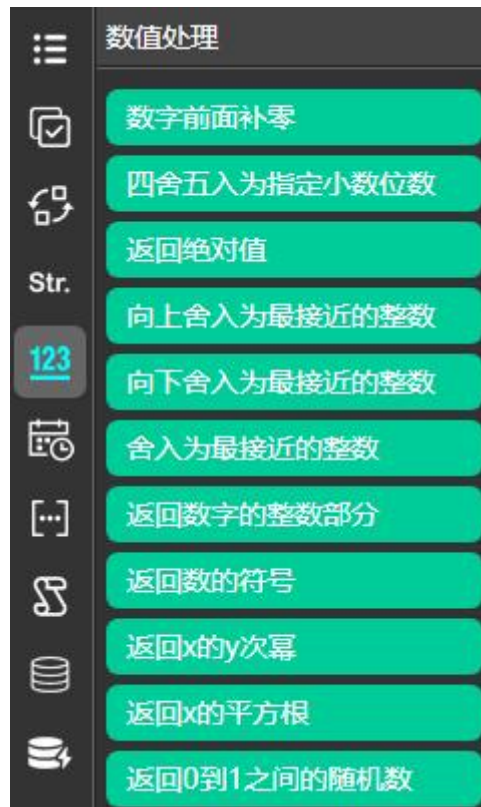


1.2 数值 (Number)

用来处理数值类型的数据，包括整数和小数，例如：0、3.14159，数值类型的值或变量可以用计算操作符进行加减乘除和求模（计算余数）运算，例如：

```
var i = 100;  
var j = 3;  
var k = i + j; //加法运算  
var h = i - j; //减法运算  
var x = i * j; //乘法运算  
var y = i / j; //除法运算  
var z = i % j; //求模运算
```

为了方便大家对数值进行操作，在 TaskBuilder 的前端页面设计器和后台服务设计器的语句库里都提供了丰富的数值处理函数，支持用图形化的形式对数值或变量进行相关处理，如下图所示：



下图所示的语句实现了对除法运算的结果再进行四舍五入计算：

```

graph TD
    A[:: 变量声明 名称: num 类型: 浮点数 初始值: 100/3] --> B[:: 四舍五入为指定小数位数 数值: num 小数点位数: 2 返回变量: ret]
    B --> C[:: 变量赋值 名称: res.data 运算符: = 值: ret]
    C --> D[:: 终止服务]
    
```

测试该后台服务的执行结果如下:

```

响应结果:
1  {
2   "code": 0,
3   "data": "33.33"
4  }
    
```

1.3 布尔值 (Boolean)

这种类型只有两个值 true 和 false , 可以理解为汉字的是和否。有些表达式或语句运算后的结果也是布尔值, 例如: $i > j$ 、 $!obj$ 。布尔值主要用来进行判断。

下图所示的语句可以判断今天是否是星期天:

```

graph TD
    A[:: 获得当前时间 返回变量: curDate] --> B[:: 获得星期值(数字) 日期时间变量名: curDate 返回变量: week]
    B --> C{:: 判断 判断条件: week == 0}
    C -- 是 --> D[:: 变量赋值 名称: res.data 运算符: = 值: '今天是星期天']
    C -- 否 --> E[+如果]
    E --> F[:: 变量赋值 名称: res.data 运算符: = 值: '今天不是星期天']
    F --> G[:: 终止服务]
    
```

测试该服务可以得到如下图所示结果:

```
响应结果:
1  {
2    "code": 0,
3    "data": "今天不是星期天"
4  }
```

今天是 2023-07-18, 所以得到的结果是: 今天不是星期天。

1.4 null

null 是 JavaScript 的关键字, 变量的值如果设置为 null, 表示该变量的值目前是空的, 就是什么也没有。

1.5 undefined

undefined 表示 “缺少值”, 即此处应该有一个值, 但是还没有定义。

以下几种情况会出现 undefined 的值:

(1) 变量被声明了, 但没有赋值时, 就等于 undefined。

```
var a;
a // undefined
```

(2) 调用函数时, 应该提供的参数没有提供, 该参数等于 undefined。

```
function f(x){console.log(x)}
f() // undefined
```

(3) 对象没有赋值的属性, 该属性的值为 undefined。

```
var o = {};
```

```
o.p // undefined
```

(4) 函数没有返回值时，默认返回 undefined。

```
function f() {console.log(1)}  
var a = f();  
a // undefined
```

2、复杂数据类型

复杂数据类型都属于对象类型，需要使用 **new** 关键字创建，包括以下几种类型：

2.1 对象 (Object)

对象是指按照某个类的定义创建的具体一个实例，基于同一个类创建的多个对象，都拥有相同名称的属性和方法，但每一个对象都是独立的变量，修改一个对象的属性不会影响到另外一个对象。

对象可以将多种类型的数据整合到一起形成复杂的数据结构，更加适合描述自然界的各种事物。在现代编程语言里，都有类 (Class) 和对象 (Object) 的概念，类表示某一类事物，例如：人类、鱼类、蕨类等等，而对象则是指具体的某个事物，例如某个人，某只猫，某片树叶。

在 JavaScript 语言里，可以用类来定义某一类对象都具备的属性和方法，属性可以理解为事物的某个特征，例如汽车的品牌、型号、长度、颜色等，实际就是依附于某个对象的变量，可以是任意数据类型，可以通过 **对象名.属性名** 的形式访问；而方法用来表示事物的某个行为或能力，例如汽车的启动、停止、前进、后退、转弯等，实际就是依附于某个对象的函数，可以通过 **对象名.方法名([参数,...])** 的形式访问。

在 JavaScript 语言里，所有类的根类是 Object 这个类，即所有类都是从 Object 这个类继承而来的，包括后面要介绍的日期、数组、键值对等，这些是 JavaScript 语言本身已经预先定义好的类，开发者可以直接创建这些类的实例对象，并使用它们提供的属性和方法。

如果要定义自己的类，可以用以下格式进行定义：

```
class 类名{  
    constructor([初始值...]) { //构造函数  
        this.属性 1 = 初始值 1;  
        ...  
    }  
  
    方法 1() {  
        //  
    }  
  
    ...  
}
```

其中，**class** 是定义类的关键字，**constructor()**函数是类的构造函数，在创建该类的实例对象时，会调用该构造函数进行创建和初始化，可以在该函数内定义类的属性，并传入初始参数设置这些属性的初始值，然后在构造函数后面，定义该类的各个方法。

下面是定义一个汽车类的 JavaScript 代码：

```
class Car{  
    //构造函数  
    constructor(brand, model, length, color) {  
        this.brand = brand;    //品牌  
        this.model = model;    //型号  
        this.length = length;  //长度  
    }  
}
```

```
        this.color = color;      //颜色
        this.status = "未启动"; //状态
    }

    start() { //启动
        this.status = "已启动";
    }

    stop() { //停止
        this.status = "未启动";
    }
}
```

在上面的代码中，**this** 关键字表示当前对象，调用哪个对象的方法，这个方法里的 **this** 对应的就是哪个对象。

类定义好后，不能直接使用该类的属性和方法，因为类只是一个定义，并没有具体的值，需要创建该类的实例对象，通过对象访问属性或调用方法，对象创建语句的格式如下：

```
new 类名([初始化参数,...])
```

其中，**new** 是 JavaScript 创建新对象的关键字，后面用空格隔开类名，然后是括号，如果构造函数可以传入初始化参数，则在括号内传入这些参数，如果没有，则可以为空。

下面是创建上面的车辆类的实例对象的示例代码：

```
var car = new Car("比亚迪", "汉 EV", 4995, "玄空黑");
```

这样，就创建了一个独立的汽车对象，然后就可以使用该对象的属性和方法了，示例代码如下：

```
console.log(car.status); //在控制台输出车辆的状态：未启动
car.start();             //调用车辆对象的启动方法，修改状态属性
console.log(car.status); //在控制台输出车辆的状态：已启动
```

通过类和对象的概念,可以实现代码的模块化管理,即可以把具有某些特定功能的代码封装成类,并放到单独的代码文件里,其他代码需要使用这些功能时,先引入这个文件,然后创建这个类的实例对象,调用其相关方法即可使用这些功能。这样就不用把所有代码都放在一个文件里,既方便了代码的查看和维护,也可以实现重复利用。

在前端页面里,所有的 tfp 组件都有专门的类对该组件的属性、方法和事件等进行定义,开发者创建的 tfp 页面里,每个组件实际上都是这个组件类的一个实例对象,对象名就是这个组件的 id,在前端脚本里就可以使用 **组件 id.属性** 和 **组件 id.方法()** 的形式访问这个组件对象的属性和方法。

在服务器端,所有后台服务文件里也都定义了一个继承自 **Servcie** 的类,在客户端请求这个服务时,引擎服务器就会加载这个后台服务文件,并创建一个这个服务类的实例对象,然后调用它的 `process(req, res)` 方法,在该方法内,可以从 `req` 参数中获得客户端发送的请求参数,然后进行相关处理,并把处理结果保存到 `res` 参数中,引擎服务器再将该参数返回给客户端。

在后台服务中,数据库操作、文件操作、服务调用、HTTP 请求等都是异步操作,这些操作执行完,会通过回调函数继续执行其他操作。在回调函数里, **this** 关键字对应的是当前操作所使用的对象,例如数据库访问对象 (`dao`)、文件访问对象 (`fs`) 等,并不是对应的当前后台服务对象,为了方便开发者在这些回调函数中访问当前后台服务对象, `.tbs` 格式的后台服务,在所有方法里都会自动定义一个名为 **self** 的变量,通过该变量,可以访问当前后台服务对象,例如下面的代码就是调用当前服务对象的终止服务方法:

```
self.end(res);
```

大多数情况下,Taskbuilder 用户不需要自己定义单独的类,但如果了解了以上基础概念,对后续开发和理解 Taskbuilder 提供的相关功能会有较大帮助。

如果创建对象只是为了临时存一些数据,不需要为该对象定义任何方法,则可以用下面这种更简洁的方式创建对象:

```
var car = {};
```

这种方式等同于用 **new Object()** 的形式创建了一个 `Object` 类的对象,但

上面的代码更加简洁，推荐采用这种方式创建。

创建好对象后，可以用下面的方式设置对象的属性：

```
car.brand = "比亚迪";  
car.model = "汉 EV";  
car.length = 4995, //mm  
car.color = "玄空黑";  
car.status = "未启动";
```

从上面的代码可以看到，即使这些属性没有实现定义，也可以直接设置，设置后，这个对象就有了这个属性。

另外，也可以在创建对象时直接设置好相关属性：

```
var car = {  
  brand: "比亚迪",  
  model: "汉 EV",  
  length: 4995, //mm  
  color: "玄空黑",  
  status: "未启动"  
};
```

后续就可以使用 **对象名.属性名** 的形式访问或设置对象的属性值。

2.2 日期 (Date)

日期类型是 JavaScript 语言内置的一种类，类的名称为 **Date**，用来处理日期和时间信息，创建日期对象的语句如下：

```
var d = new Date();
```

新创建的日期对象表示的是当前日期和时间。

然后就可以用该对象提供的方法对日期和时间进行相关处理：

```
var date = d.getDate(); //返回一个月中的第几天
var year = d.getFullYear(); //返回 4 位年份
```

为了方便大家对日期对象进行操作, 在 TaskBuilder 的前端页面设计器和后台服务设计器的语句库里都提供了丰富的日期处理函数, 支持用图形化的形式对日期对象或变量进行相关处理, 如下图所示:



上面的示例代码对应的图形化语句如下图所示:



测试该后台服务得到的响应数据如下图所示:



2.3 数组 (Array)

数组类型是 JavaScript 语言内置的一种类型，类的名称为 **Array**，数组是值的有序集合，数组中的每个值称为一个元素，每个元素在数组中都有一个唯一的数字位置，称为索引，索引从 0 开始，依次递增。



可以将数组理解为一个正在排队等车的队伍，每个人为数组里的一个元素，每个人都有一个数字编号，第一个人的编号为 0，往后依次递增。

数组里的每个元素可以是任意数据类型，不要求所有元素的类型必须一致，例如数组的第一个元素是数值，第二个元素可以是日期，第三个元素可以是字符

串，是完全可以的，甚至数组里还可以存数组，构成多维数组。

创建数组对象的语句如下：

```
var arr = new Array();
```

也可以用以下简化方式创建数组（推荐采用这种方式）：

```
var arr = ["张三", "李四", "王五"];
```

这样相当于不仅创建了一个数组对象，而且还为其设置了 3 个元素，多个元素之间用英文逗号分隔，如果要用上面的形式创建一个空的数组，则把等于号后面的内容改成 `[]` 即可，也就是说，在 JavaScript 里，一对中括号表示一个数组。

数组常见的操作代码示例如下：

```
var len = arr.length; //获得数组的长度
var name = arr[1]; //获得指定索引位置的元素
arr[1] = "Tom"; //设置指定索引位置的元素
arr.push("二狗"); //将元素添加到数组末尾
```

在后续的开发过程中，会经常用到数据，例如后台数据查询操作返回的结果就是数组，下拉列表、数据表格组件和可编辑表格等前端组件需要绑定的数据都是数组类型。

为了方便大家对数组对象进行操作，在 TaskBuilder 的前端页面设计器和后台服务设计器的语句库里都提供了丰富的数组处理函数，支持用图形化的形式对数组对象或变量进行相关处理，如下图所示：



下图所示的语句演示了几个常用的数组操作：



测试该后台服务得到的响应数据如下图所示:

```
响应结果:
1  {
2    "code": 0,
3    "data": [
4      "b",
5      123,
6      "d"
7    ]
8  }
```

另外, 还有函数变量, 在后续介绍函数这个概念时会详细介绍。

2.4 JSON 对象

内容暂时还未提供。

运算符

运算符就是用来进行各种运算的符号, JavaScript 提供了以下几种类型的运算符:

1、算术运算符

算术运算符用来进行加减乘除等算术运算, 具体功能和示例见下表:

运算符	功能描述	例子	结果
+	加法	1 + 1	2
-	减法	2 - 1	1
*	乘法	2 * 3	6
/	除法	6 / 2	3

%	取模 (求余数)	5 % 2	1
++	自增	i=2;i++;	3
--	自减	i=3;i--;	2

2、赋值运算符

赋值运算符用来给变量赋值, 如果给定 $x=10$, $y=5$, 下表为使用各种赋值运算符的示例:

运算符	例子	等同于	运算结果
=	$x=y$		$x=5$
+=	$x+=y$	$x=x+y$	$x=15$
-=	$x-=y$	$x=x-y$	$x=5$
=	$x=y$	$x=x*y$	$x=50$
/=	$x/=y$	$x=x/y$	$x=2$
%=	$x\%=y$	$x=x\%y$	$x=0$

3、比较运算符

比较运算符用来进行比较运算, 运算结果为布尔值, 具体功能和示例见下表:

运算符	功能描述	例子	结果
==	等于	$1 == 2$	false
!=	不等于	$1 != 2$	true
>	大于	$1 > 2$	false
<	小于	$1 < 2$	true
>=	大于或等于	$1 >= 2$	false
<=	小于或等于	$1 <= 2$	true

4、逻辑运算符

逻辑运算符用来进行与或非这三种逻辑运算, 运算结果为布尔值, 如果给定 $x=6$ 且 $y=3$, 下表展示各种逻辑运算的示例:

运算符	功能描述	例子	结果
&&	并且	(x < 10 && y > 1)	true
	或者	(x == 5 y == 5)	false
!	非	!(x == y)	true

5、字符串连接字符

两个字符串如果要拼接为一个字符串，可以使用 + 号，示例代码如下：

```
var str = "abc" + "def"; //abcdef
```

数值也可以和字符串直接用 + 号连接，不管字符串在前面还是后面，返回结果都是字符串，示例代码如下：

```
var str = "abc" + 123; //abc123
```

注意：赋值运算符的 = 和判断运算符的 == 容易搞混，= 号是将右边的值赋给左边的变量，== 是判断左右两边的值是否相等，请注意区分。

表达式

一个表达式 (expressions) 会产生一个值，它可以放在任何需要一个值的地方，比如作为一个函数调用的参数，作为判断条件，作为赋值语句的值等等，表达式是构成语句的基本元素。

JavaScript 的表达式包括以下几类：

1、原始表达式

包括具体的值（字面量）和变量，例如：123, "abc", true, null, i, str , 其中 i 和 str 是事先声明的变量。

2、数组或对象值表达式

用来表示数组或对象值的表达式，例如：

```
[1,2,3,4] //定义一个数组  
{x:1, y:2} //定义一个对象
```

3、运算表达式

值或变量与各种运算符构成的表达式，例如：

```
x + y //算术表达式  
x > y //比较表达式  
(x >= 1) && (y <= 5) //逻辑表达式  
str + "abc" //字符连接表达式
```

4、属性或元素访问表达式

访问对象属性或数组元素的表达式，例如：

```
car.status //访问对象指定名称的属性  
arr[0] //访问数组指定索引位置的元素
```

另外，还有函数定义表达式和函数调用表达式，这部分内容会在后面介绍函数的概念时再详细介绍。

语句

语句是指可以独立运行的一行 JavaScript 代码或多行代码块，语句由值、变量、运算符、表达式、关键字和注释等元素构成，一个完整的 JavaScript 程序由多条语句构成，并按照从上到下的顺序执行（不包括异步函数）。

语句用分号 `;` 结尾（不是必须，但建议加分号），例如：

```
a = 5;  
b = 6;  
c = a + b;
```

一行可以有多条语句（一般不建议这样，可读性差），例如：

```
a = 5; b = 6; c = a + b;
```

语句也可能有多行，例如判断和循环语句，例如：

```
if(x % 2 == 1) {  
    console.log(x + "是奇数!");  
}
```

下面介绍一下 JavaScript 几种常用的语句：

1、表达式语句

表达式语句是 JavaScript 中最简单的语句，前面介绍的几种表达式在后面加上分号 `;`，就可以构成一条语句，例如：

```
x = y * 10; //赋值语句
```

```
i++; //数值变量自加自减运算语句  
arr[1] = 123; //数组元素设置语句  
car.status = "已启动"; //对象属性设置语句  
car.start() //函数或对象方法调用语句
```

2、变量声明语句

这种类型的语句，在前面介绍变量时已经讲过了，示例代码如下：

```
var x; //声明变量  
var i = 0; //声明变量，同时设置初始值
```

3、复合语句

用一对大括号 `{ }` 将多条语句括起来便是一个复合语句，看起来就像一条语句一样，也可以称为一个代码块，在代码块最后边不需要加分号。同一个代码块中的语句，要么都执行，要么都不执行。示例代码如下：

```
{  
    a = 5;  
    b = 6;  
    c = a + b;  
}
```

4、if 判断语句

使用 if 条件判断语句可以在执行某个语句之前先进行判断，如果条件成立

才会执行，不成立则不执行。

if 判断语句有三种语法：

4.1 语法一

if (条件表达式) 语句;

如果条件表达式的值为 true, 则执行 if 关键字后面的语句; 如果值为 false, 则不执行。

if 语句只能控制紧随其后的那个语句，如果希望 if 语句可以控制多条语句，可以将这些语句统一放到代码块中。if 语句后的代码块不是必须的，但是在开发中建议尽量写，使用代码块被认为是一种最佳的编程实践，即使要执行的代码只有一行。这样做可以使每个条件要执行什么一目了然。

```
var a = 15;
if(a > 10 && a <= 20) {
    console.log("a 大于 19, 小于等于 20!");
}
```

为了方便大家实现判断操作，在 TaskBuilder 的前端页面设计器和后台服务设计器的语句库中的 **基本语句** 里都提供了 **判断** 语句，该语句就是这里介绍的 if 判断语句，支持用图形化的形式实现各种业务判断，如下图所示：



测试该后台服务，会得到如下图所示的响应结果：

```

响应结果:
1  {
2    "code": 0,
3    "data": "a大于19, 小于等于20!"
4  }
    
```

4.2 语法二

```

if (条件表达式) {
    语句...
} else {
    语句...
}
    
```

这种语法表示当条件表达式的值为 true 时,则执行 if 关键字后面代码块里的语句, 如果为 false, 则执行 else 关键字后面的代码块里的语句。

点击图形化判断语句设置面板底部矩形条最右侧的 **否则** 按钮:



可以显示否则代码块设置区域, 如下图所示:



前面在介绍数据类型里的布尔值时, 提供了这种语法的图形化语句示例, 这里就不再提供示例了。

4.3 语法三

```
if (条件表达式 1) {  
    语句...  
} else if (条件表达式 2) {  
    语句...  
} else if (条件表达式 3) {  
    语句...  
} else {  
    语句...  
}
```

这种语法表示当有多个条件表达式时, 从上至下依次判断, 如果其中某个表达式的值为 true, 则执行这个 **if** 或 **else if** 关键字后面代码块里的语句, 然后终止整个判断语句, 不再进行后续的判断; 如果上面的所有条件表达式的值都为 false, 则执行 **else** 关键字后面的代码块里的语句。

这种语法里的第一个 if 判断是必须的, 中间的 **else if** 判断可以有任意多个, 最后面的 **else** 及其后面的代码块最多只能有一个, 且不是必须的, 如果没有这部分, 则表示如果上面的所有条件表达式的值都为 false 时, 不会执行任何操作。

在可视化业务逻辑设计器中, 点击判断语句底部矩形条右侧的 **如果** 按钮, 可以添加 **else if** 判断及其语句块, 如下图所示:



完整的图形化语句实例如下:



4.4 几种特殊值的判断

可以直接用 if 条件判断语句对变量或表达式的值进行判断, 如果值为 0、null、undefined 或空字符串这四种特殊的值时, 条件不成立, 反之成立, 例如下属

代码:

```
var str; //此时 str 的值为 undefined
if(!str) {
    console.log("str 变量未赋值! ");
}
str = "Hello TaskBuilder!";
if(str) {
    console.log(str);
}
```

5、for 循环语句

5.1 for 循环基本语法

for 循环是 JavaScript 中最常用的一种循环方式, 在 for 循环中, 为我们提供了专门的位置用来放三个表达式:

1. 初始化表达式
2. 条件表达式
3. 更新表达式

语法:

```
for (初始化表达式; 条件表达式; 更新表达式) {
    语句...
}
```

示例代码如下:

```
for (var i = 0; i < 10; i++) {
    console.log(i)
```

```
}
```

以上代码会循环输出 10 个数字，从 0 开始，依次递增 1。

for 循环的执行流程：

1. 初始化表达式，初始化变量；
2. 执行条件表达式判断是否执行循环：如果为 true，执行循环代码块里的语句，false 则终止循环；
3. 执行更新表达式，更新表达式执行完毕继续重复步骤 2；

for 循环中的三个部分都可以省略，也可以写在外部。

for 循环语句可以层层嵌套，下面的代码实现九九乘法表：

```
for (i = 0; i <= 9; i++) {
    for (let j = 0; j <= i; j++) {
        console.log( i+"*"+j+"="+i*j );
    }
}
```

以上代码用图形化语句设置的界面如下图所示：



测试该后台服务，在任擎服务管理器内可以看到如下图所示的输出：

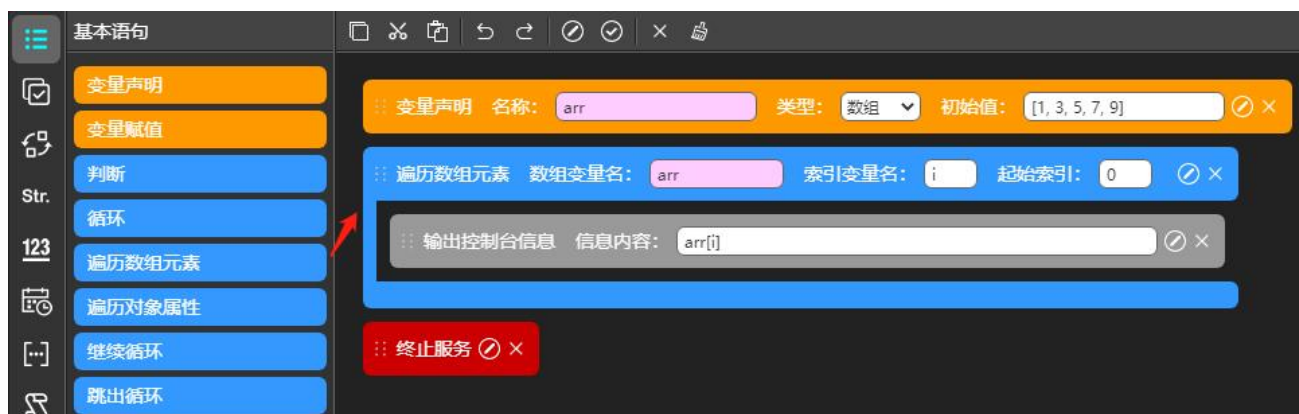
```
22:17:14.941 rid:2307202217140109 请求路径: /Service
22:17:14.941 rid:2307202217140109 接收URL请求数据: {}
22:17:14.942 rid:2307202217140109 接收POST请求数据:
{"_requestId":182,"_auth_org":"xigetest","_auth_ts":1689862634926,"_auth_data":"DTRnoq
NqjAMT9vsJTqoCtE1XPN1HXADEzWRiVNVdRrsNELNFKK/2Q==","service":"demo/service/test.tbs"}
22:17:14.942 请求服务: /app/demo/service/test.tbs
0*0=0
1*0=0
1*1=1
2*0=0
2*1=2
2*2=4
3*0=0
3*1=3
3*2=6
3*3=9
4*0=0
4*1=4
4*2=8
4*3=12
4*4=16
```

5.2 遍历数组元素

可以使用 for 循环，循环访问或设置数组元素，示例代码：

```
var arr = [1, 3, 5, 7, 9];
for(let i = 0;i<arr.length;i++) {
    console.log(arr[i]);
}
```

为了方便大家循环访问数组的各个元素，TaskBuilder 的可视化语句库的基本语句中提供了 **遍历数组元素** 的语句，开发者可以直接拖拽使用，示例如下图所示：



测试该语句，在服务器端可以看到如下输出内容：

```
08:07:03.640 rid:2307210807030002 请求路径: /Service
08:07:03.641 rid:2307210807030002 接收URL请求数据: {}
08:07:03.641 rid:2307210807030002 接收POST请求数据:
{"_requestId":184,"_auth_org":"xigetest","_auth_ts":1689898023632,"_auth_data":"g7rj/
jqIVJiLH/3vs9dFSa10uFKygFhS7X/zLSFEu5yQ6h7z0hAw=", "service":"demo/service/test.tbs"}
08:07:03.641 请求服务: /app/demo/service/test.tbs
1
3
5
7
9
08:07:03.667 rid:2307210807030002 发送响应数据: {"code":0}
```

5.3 遍历对象属性

for 循环有一个变体，叫 for in 循环，主要用来遍历对象属性，使用它可以将对象中的属性依次循环出来，其语法格式如下：

```
for (variable in object) {
    // 要执行的代码
}
```

其中，variable 为一个变量，每次循环时这个变量都会被赋予不同的值，我们可以在{ }的代码块中使用这个变量来进行一系列操作；object 为要遍历的对象，在每次循环中，会将 object 对象中的一个属性的名称赋值给变量 variable，直到对象中的所有属性都遍历完。

for in 循环示例代码：

```
// 定义一个对象
var person = {"name": "Clark", "surname": "Kent", "age": "36"};

// 遍历对象中的所有属性
for(var prop in person) {
    console.log( prop + " = " + person[prop] );
}
```

为了方便大家循环访问对象的各个属性，TaskBuilder 的可视化语句库的基本语句中提供了 **遍历对象属性** 的语句，开发者可以直接拖拽使用，示例如下图所示：



测试该服务，可以在服务管理器中看到如下输出：

```
08:52:56.497 rid:2307210852560005 请求路径: /Service
08:52:56.497 rid:2307210852560005 接收URL请求数据: {}
08:52:56.497 rid:2307210852560005 接收POST请求数据:
{"_requestId":187,"_auth_org":"xigetest","_auth_ts":1689900776494,"_auth_data":"i+ISQ8iL8dNW3iqRFYrqjnhQrWfB3GD8FxElpk1DVrhlDMg+tes+ng=", "service": "demo/service/test.tbs"}
08:52:56.497 请求服务: /app/demo/service/test.tbs
name=Clark
surname=Kent
age=36
```

6、continue 语句

continue 语句用于跳过循环的当前迭代，程序的控制流转到下一个迭代。

continue 语句只是退出当前迭代，根据控制表达式还允许继续进行下一次循环。

示例代码如下：

```
var iNum = 0;

for (var i=1; i<10; i++) {
    if (i % 5 == 0) {
        continue;
    }
    iNum++;
}

console.log(iNum); //输出：8
```

上面这段代码最后的输出为：8，即执行循环的次数。应执行的循环总数为 9，不过当 i 的值为 5 时，因为里面的 if 判断条件成立，将执行 continue 语句，会使循环跳过后面的语句，返回循环开头继续执行下一轮循环。

7、break 语句

break 语句用于立即终止当前循环语句，表示不再执行该循环内的任何代码。

示例代码如下：

```
var iNum = 0;

for (var i=1; i<10; i++) {
    if (i % 5 == 0) {
        break;
    }
    iNum++;
}

console.log(iNum); //输出：4
```

在以上代码中，for 循环从 1 到 10 迭代变量 i。在循环主体中，if 语句将（使用取模运算符）检查 i 的值是否能被 5 整除。如果能被 5 整除，将执行 break 语句。最后输出的结果为：4，即退出循环前执行循环的次数。

如果有多个循环进行了嵌套，则 break 语句只会退出离它最近的那个循环，示例代码如下：

```
for (let i = 1; i <= 3; i++) {  
  for (let j = 1; j <= 3; j++) {  
    if (i == 2) {  
      break;  
    }  
    console.log(`i = ${i}, j = ${j}`);  
  }  
}
```

在上面的代码中，当 i == 2，break 语句执行。它终止内部循环，程序的控制流移到外部循环。因此，i=2 的值永远不会显示在输出中。输出结果如下：

```
i = 1, j = 1  
i = 1, j = 2  
i = 1, j = 3  
i = 3, j = 1  
i = 3, j = 2  
i = 3, j = 3
```

函数

JavaScript 函数是被设计为可以执行特定任务的代码块，函数会在被其他代码调用时执行。通过函数能够对代码进行复用，只要定义一次代码，就可以多

次使用它，每次调用函数时传递不同的参数，可以产生不同的结果。

1、函数定义

在使用函数之前，需要先定义好函数，JavaScript 函数通过 `function` 关键词进行定义，其后是函数名和一对小括号 `()`，函数名可包含字母、数字或下划线（规则与变量名相同），小括号里可包括由逗号分隔的一个或多个参数名，由函数执行的代码被放置在大括号 `{}` 中，函数定义语句的具体格式如下：

```
function 函数名(参数 1, 参数 2, ...){  
    函数体;  
    返回值; //非必须  
}
```

示例代码如下：

```
//无参数，无返回值  
function hello() {  
    console.log("Hello TaskBuilder!");  
}
```

```
//有参数，有返回值  
function add(x, y) {  
    return x + y;  
}
```

另外，在前面介绍对象这种数据类型时讲过了，对象的方法其实也是函数，只不过这个函数是属于具体某个对象的，怎么定义对象的方法，见前面数据类型里对象这种类型的介绍。

2、函数调用

函数调用表达式用来执行指定的函数，格式如下：

函数名(参数 1, 参数 2, ...)

通过以上格式的代码可以激活并执行函数，在小括号中可以包含零个或多个参数，参数之间通过逗号进行分隔。

在函数调用表达式后面加上分号 `;`，就变成了函数调用语句，可以独立运行，例如：

```
hello();
```

函数调用表达式可以放到别的表达式里面或者作为参数传给其他函数，例如：

```
var n = add(1, 2) * 100;
```

在下面示例中，使用小括号调用函数，然后直接把返回值传入函数，进行第二次运算，这样可以节省两个临时变量。

```
var n = add( add(1, 2) , 3) * 100;
```

如果是要调用对象的方法，则调用语句格式如下：

对象名.方法名(参数 1, 参数 2, ...);

示例代码如下：

```
car.start();
```

```
car.stop();
```

3、函数参数

函数参数 (Function parameters) 在函数定义语句的小括号 () 中列出参数名称, 然后在函数调用表达式里传入具体的值。

函数参数是局部变量, 只能在该函数大括号 {} 中的代码块中使用。

在调用 JavaScript 函数时, 如果该函数定义了多个参数, 如果位于后面的一个或多个参数没有值, 在调用语句里可以省略不写, 如果所有参数都没有值, 则可以全部省略。例如下面的函数, 定义了三个参数, 调用时如果只有两个参数有值, 则第三个参数可以不传, 这时, 第三个参数的值为 undefined, 用 `if(str3)` 判断语句判断时, 条件不成立:

```
//函数定义
```

```
function str2arr(str1, str2, str3) {  
    var arr = [];  
    if(str1) arr.push(str1);  
    if(str2) arr.push(str2);  
    if(str3) arr.push(str3);  
    return arr;  
}
```

```
//函数调用
```

```
var arr = str2arr("cat", "dog");
```

如果第一个参数没有值, 但第二个参数有值, 这时, 第一个参数需要传 `null`, 不能空着, 否则会报错, 例如:

```
var arr = str2arr(null, "dog");
```

```
var arr = str2arr(, "dog"); //这样写是不对的, 运行时会出现错误
```

4、函数返回值

在函数体的代码块内, 可以通过 **return** 语句给函数调用者返回值, 格式如下:

```
return 返回值;
```

如果 **return** 语句没有提供返回值, 或者该函数内没有 **return** 语句, 则表示该函数没有返回值, 如果在其他表达式中调用该函数, 则该函数此时的返回值为 `undefined`。

当函数体内的代码运行到 **return** 语句时, 整个函数会立即退出, 不再运行函数体内该语句后面的代码。

一个函数内可以有多个 **return** 语句, 然后通过条件判断, 在满足不同的条件时, 返回不同的值, 或终止函数运行, 示例代码如下:

```
function str2arr(str1, str2, str3) {  
    if(str1) return str1;  
    if(str2) return str2;  
    if(str3) return str3;  
    return null;  
}
```

5、匿名函数

匿名函数是指没有设置名称的函数, 也称为函数表达式, 可以将匿名函数赋值给一个变量, 然后在后面需要使用时, 再用该变量名调用这个函数。

定义匿名函数并赋值给变量的语句格式如下:

```
var 函数变量 = function(参数 1,参数 2,...) {  
    函数体;  
    返回值; //非必须  
}
```

示例代码如下:

```
//函数定义  
var hello = function() {  
    console.log("Hello TaskBuilder!");  
}  
//函数调用  
hello();
```

另外, 匿名函数也可以作为参数传给其他函数, 例如下面介绍的回调函数。

6、回调函数

回调函数是指作为一个参数传递给另外一个函数的函数, 示例代码如下:

```
function add(num1, num2, callback) {  
    var sum = num1 + num2;  
    callback(sum);  
}  
  
function print(num) {  
    console.log(num);  
}  
  
add(1, 2, print); //3
```

以上示例代码中的 `callback` 参数就是回调函数，在定义它的函数内，可以把它当做一个函数名执行正常的函数调用语句，例如：`callback(sum)`，`callback` 具体对应的是哪个函数，取决于在调用 `add` 函数时，给 `callback` 参数传入的函数名，这里是 `print`。

回调函数作为参数传递给一个函数的时候，传递的只是函数的定义并不会立即执行。和普通的函数一样，回调函数在调用函数中也要通过 `()` 运算符调用才会执行。

也可以用匿名函数直接作为参数传入，上面调用 `add` 函数的代码可以修改如下：

```
add(1, 2, function(num) {  
    console.log(num);  
});
```

上面的代码中，`callback` 参数对应的是一个匿名函数，这个函数的整个定义语句作为参数传给了 `add` 函数，这样就不需要再单独定义一个 `print` 函数了，这种方式更加简洁。

回调函数使用得最多的场景就是在异步操作中，具体见下面的介绍。

7、异步函数

JavaScript 作为一种脚本语言，最初设计的目的是为了用户在使用浏览器浏览网页时，可以在页面内实现一些互动操作，或者用 JavaScript 脚本控制页面元素实现一些界面效果，所以，JavaScript 语言设计得比较简单，只支持单线程，程序默认是同步执行的，即同一时间只能做一件事。但对于一些特定的场景来说，这种只能按时间先后一步一步执行的模式，用户体验会比较差，例如打开一个需要从数据库中查询数据的动态页面时，如果必须等数据都查询完并返

回给浏览器后再显示整个页面，遇到网速慢，或者数据较多时，就会出现整个浏览器一片空白，需要让用户等半天才能看到具体的内容，让用户觉得系统卡顿，这样的体验肯定不好，为此，JavaScript 增加了异步执行的模式，即在启动某项工作后，不用等它执行完，可以先去做别的事，等它执行完了，再回过头来执行与这项工作相关的操作。例如上面这个页面，可以在浏览器里先显示页面固定的模版内容，同时去加载数据，等数据加载完，再把数据显示在页面指定的区域。

为了更好地理解异步操作，再举个更加浅显的例子：

小明妈妈刚准备做晚饭，但这时小红来找小明玩，小明怎么办呢？

如果小明等妈妈做完饭，并吃完饭后再去找小红玩，天都黑了，小红肯定不会出来了。

那就跟妈妈说一声吧，告诉妈妈去和小红玩了，等做完饭再喊他。

这时，妈妈做饭和小明出去玩这两件事就可以同时进行，而不用非得等一件事做完再做另一件事，等妈妈做完饭，再喊小明回家吃饭，两不耽误。

JavaScript 里的定时器方法 `setTimeout()`和 `setInterval()`、事件监听器、前端页面里请求后台服务、后台服务里进行数据库操作、读写文件、连接其他服务器等这些需要较长时间才能完成或定时执行的任务，都是异步执行的，那么，怎么才能实现在异步任务执行完后，再去执行与之相关的其他任务呢，钥匙就是上面介绍的回调函数。

示例代码如下：

```
console.log("1");
setTimeout(function() {
  console.log("2");
}, 1000);
console.log("3");
```

输出结果：

1
3

2

上面代码中的 `setTimeout()` 函数是 JavaScript 语言内置的一个定时器函数, 调用它时, 需要传入两个参数, 第一个参数是回调函数, 即到达设定的时间间隔时, 就执行该回调函数, 第二个参数是定时的时间间隔值, 单位是毫秒。

从代码的执行顺序看, 应先输出 2, 但实际上最后才输出 2, 就是因为 `setTimeout()` 函数是异步执行的, 它的第 2 个参数设置了需要等待 1000 毫秒后, 再执行第一个参数里的回调函数, 而不是运行 `setTimeout()` 函数时, 马上就执行回调函数。

上面举的例子是定时执行某项任务, 下面再举个真正异步执行的例子, 即在前端页面里, 通过服务组件 (Service) 请求后台服务, 并输出服务器端的响应结果:

```
console.log("start");
service1.path = "sys/service/dep/getByCode";
service1.request({ code: "1001" }, function(req, res){
    console.log(res);
});
console.log("end");
```

上面的代码中, `service1` 是前端页面里的一个服务组件 (Service), 它的 `request()` 方法属于异步函数, 用来向服务器端发送请求, 这个函数的第一个参数用来设置需要传递给服务器端的请求参数, 第二个参数用来设置在服务器端返回响应结果后需要执行的回调函数。

代码执行到 `service1.request()` 这一段时, 并不会一直在这里等待, 而是会继续执行下面的代码: `console.log("end")`, 当服务器端返回结果后, 再执行回调函数里的: `console.log(res)`, 这样就可以很好地解决上面说的页面打开时必须等所有数据都加载完才能显示页面的问题, 通过异步和回调函数, 可以先显示页面的静态内容, 同时请求后台服务查询动态数据, 服务器端返回结果后, 再把数据显示到页面里, 这样的用户体验会更好。

在对象方法的回调函数里, 需要注意 **this** 关键字的指向问题, 在回调函数里, **this** 关键字指向的是当前运行时的全局对象, 并不是指向该回调函数所在的方法所属的对象, 如果要在回调函数里访问当前方法所属的对象, 可以在调用该方法之前, 先将 **this** 存到一个变量内, 然后在该方法内, 就可以用该变量访问当前对象, 示例代码如下:

```
var name = 'my name is window';
var obj = {
  name: 'my name is obj',
  fn: function () {
    var that = this;
    setTimeout(function () {
      console.log(that.name);    //my name is obj
    }, 1000)
  }
}
```

上面的代码中, 定义了一个名为 **obj** 的对象, 在该对象里, 定义了一个名为 **fn** 的方法, 在该方法里, 调用了 **setTimeout()** 这个异步函数, 在该函数内, 因为不能直接用 **this** 关键字访问 **obj** 这个对象, 于是在调用 **setTimeout()** 函数前, 将 **this** 保存到 **that** 这个变量里, 然后在 **setTimeout()** 函数里就可以使用 **that** 变量访问 **obj** 这个对象了, 因为 **that** 变量是在 **fn** 这个函数里定义的, 要是这个函数中 **that** 变量定义语句后的代码, 都可以访问到这个变量。

在使用 TaskBuilder 开发时, 有个“潜规则”, 即在前端页面里, 用 **that** 指代 **this**, 在后台服务里, 用 **self** 指代 **this**, 在前端页面里不能定义名为 **self** 的变量, 因为该变量对应的是当前页面的 window 对象。

代码引入

可以将 JavaScript 代码放到单独的 js 文件中，在前端页面或后台服务里需要使用这些代码时，再通过代码引入将这个文件中的代码添加进来。

1、代码引入的好处

js 代码引入有以下好处：

- **提高代码的可维护性**

将 js 代码分离成独立的文件，可以使代码更加结构化和易于维护。如果你要修改代码，只需要修改一个独立的文件就可以了，而不用去修改每个前端页面或后台服务文件中的代码。

- **方便代码的重用**

如果你有多个文件需要使用相同的 js 代码，那么使用外部 JavaScript 文件可以方便地实现代码的重用。只需要在不同的前端页面或后台服务文件中引入同一个 js 文件即可。

- **减少代码的加载时间**

使用外部 js 文件可以减小前端页面文件的体积，进而减少页面的加载时间。因为 js 文件可以缓存，如果多个前端页面引用同一个 js 文件，那么在第一个页面加载 js 文件时，它就已经被缓存了，从而在之后的页面加载中不会再次下载该文件。

下面分别介绍一下如何在前端页面和后台服务中引入外部 js 文件。

2、在前端页面中引入 js 脚本

在 TaskBuilder 的前端页面设计器中，引入外部 JavaScript 脚本的方法如下：

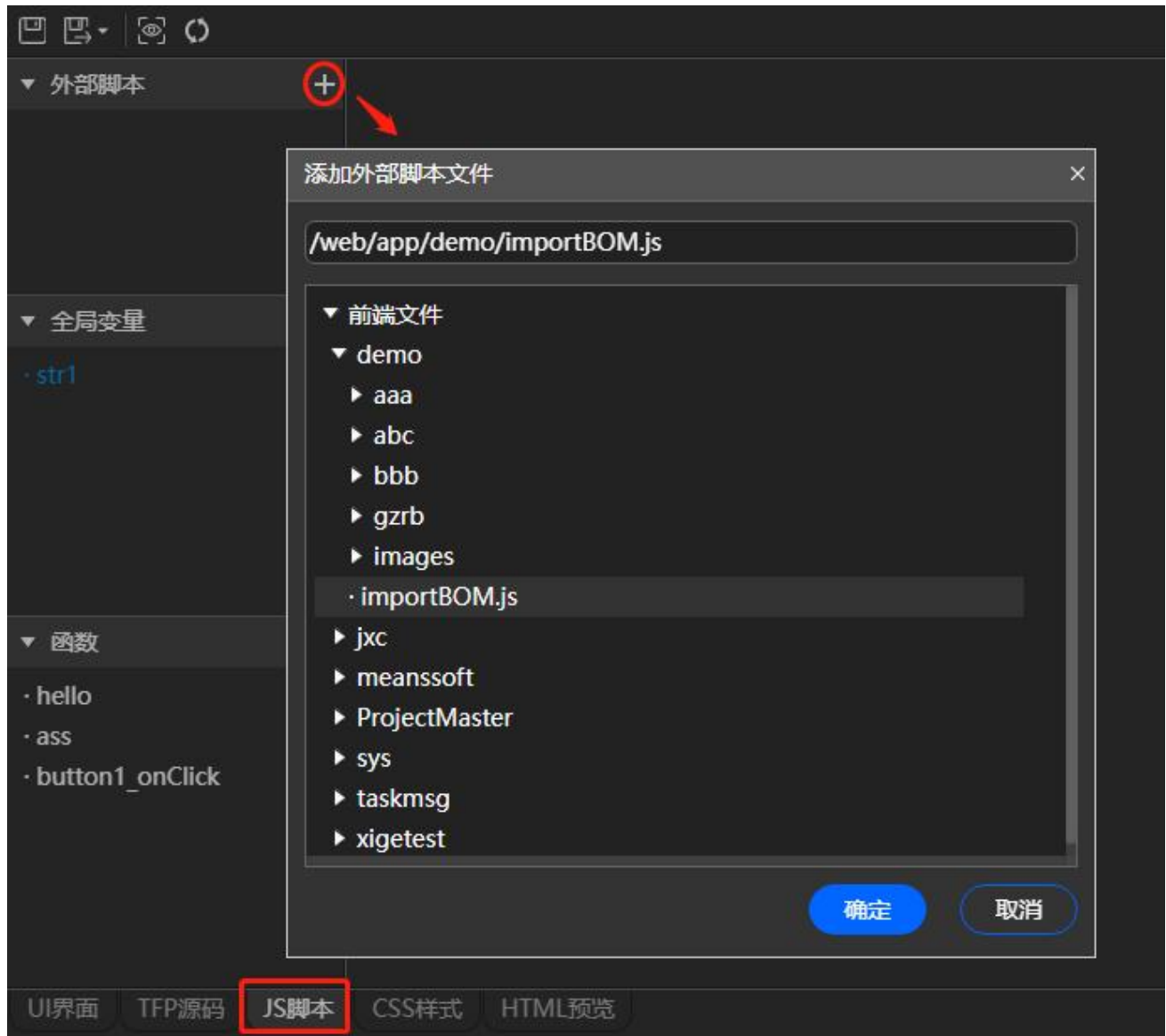
- 1、点击前端页面设计器底部的 **JS 脚本** 选项卡，显示 JavaScript 脚本编

编辑器;

2、再点击 JavaScript 脚本编辑器左上角的 **外部脚本** 栏右侧的添加按钮, 弹出 **前端文件选择器**;

3、找到要引用的 js 文件, 选中后, 点击确定按钮, 即可完成添加外部脚本的引用。

具体界面如下图所示:



3、后台服务添加外部模块引用

为了实现 JavaScript 代码的模块化管理, 业界先后提出了多种 JavaScript 模块化规范, 目前最常见的就是 CommonJS 和 ES Modules 这两种模块化规范,

其中, CommonJS 是 Node.js 早期采用的一种规范, 主要应用在服务器端程序中, ES Modules 是由 ECMA (欧洲计算机制造商协会) 制定的一种 JavaScript 模块化规范, 目前主流的 Web 浏览器和 Node.js 都支持该规范, 前后端 js 模块都可以采用该规范。这两种模块化规范的具体介绍和对比请自行查阅相关资料。

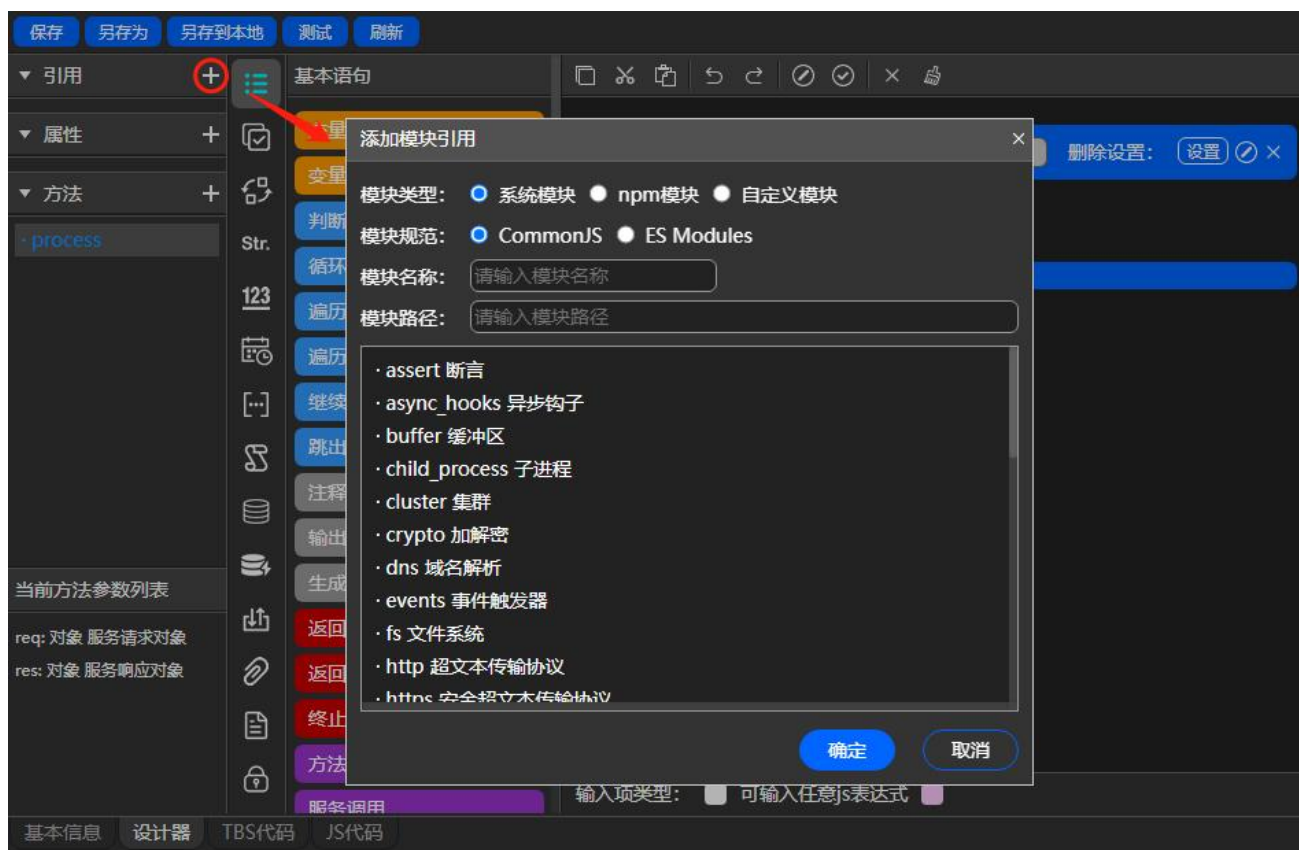
为了兼容以前的一些代码, 用 TaskBuilder 开发的后台服务最终都会按照 CommonJS 规范编译成相应的 js 代码。

在后台服务里引入其他模块的主要目的是为了使用一些已经封装好的功能模块, 不用在该服务里再重新编码实现, 例如文件压缩解压、邮件收发、短信发送、设备通信等等, 这些功能模块有以下三种类型:

- **系统模块:** 指 node.js 内置的系统模块, 例如: fs、http、net、zlib 等, 如果后台服务需要使用这些功能, 就需要先引入。TaskBuilder 后台服务设计器的语句库中有些语句会自动引入关联的系统模块, 例如文件操作会自动引入 fs 模块, HTTP 请求语句会自定引入 http 模块, 这时就不需要再手动引入这些系统模块。
- **npm 模块:** 指通过 node.js 的 npm 命令安装到 tasgine\node_modules 目录里的模块, npm 是 Node.js 默认的包管理器, 提供了 100 多万个 JavaScript 功能模块包, 可以满足很多开发需求, npm 的具体介绍和使用可以上 npm 官网 <https://www.npmjs.com> 或者搜索 npm 相关的资料进行了解。
- **自定义模块:** 指用户自己开发的、部署在 tasgine\app 目录或其子目录里的后台代码文件, 支持.js 和.tbs 两种格式。

在 TaskBuilder 的后台服务设计器中, 引入 JavaScript 模块的方法如下:

点击后台服务设计器左上角 **引用** 栏右侧的加号, 打开 **添加模块引用** 对话框, 如下图所示:



在上图所示的对话框中，需要设置以下参数：

- **模块类型**：可以选择系统模块、npm 模块或自定义模块，具体区别见前面的介绍，默认为系统模块。如果选择的系统模块，则在下面的模块列表里会列出 node.js 内置的系统模块供选择；如果选择的 npm 模块，则会列出 tasgine\node_modules 目录里安装的 npm 模块供选择；如果选择的自定义模块，则会列出 tasgine\app 目录里的子目录和后台服务文件供选择。
- **模块规范**：可以选择 CommonJS 和 ES Modules，默认为 CommonJS。
- **模块名称**：用来设置引入的模块在当前后台服务中的变量名，模块名称区分大小写，只能是字母、数字或下划线，且必须以字母或下划线开头。
- **模块路径**：用来设置模块的路径，可以从下面的列表中选择，也可以手动输入。

JavaScript 的几种运行时环境

在用 TaskBuilder 开发的应用中，不同文件中的 JavaScript 代码会运行在不同的运行时环境中，在这些运行时环境中，js 的基本语法都是一样的，包括前面介绍的变量、数据类型、运算符、表达式等，但每种环境各自都有不同的适用场景和一些特殊功能，具体见下表：

运行时环境	适用场景	扩展功能
引擎服务器	用来运行后台服务的 JavaScript 代码。	可以进行数据库增删改查、读写服务器端文件等操作。
Web 浏览器	用来运行前端页面中的 JavaScript 代码。 电脑端页面由电脑端 Web 浏览器运行，手机端 H5 页面由手机端浏览器或手机 APP 内置的浏览器运行。	可以访问和控制页面中的组件或 HTML 元素。出于安全考虑，浏览器不允许网页内的 js 访问电脑或手机上的文件、硬件等资源。
任讯 APP、阿里钉钉、企业微信	用来运行专门为这几款 APP 开发的手机端页面中的 JavaScript 代码，本质上也是通过嵌入的浏览器来运行，只不过各自都提供了一些特殊的扩展功能。	除了可以像浏览器中的 js 一样访问和控制页面元素外，还提供了拍照、扫码、定位等扩展功能。
微信	用来运行微信小程序中的 JavaScript 代码，页面元素与 H5 页面不完全一样，有自己的一套组件和访问控制机制。	可以访问和控制小程序页面中的组件，并提供了读取微信用户信息、拍照、扫码等扩展功能。