

JSON 对象

1. JSON 格式

JSON 格式 (JavaScript Object Notation 的缩写) 是一种用于数据交换的文本格式, 2001 年提出, 目的是取代繁琐笨重的 XML 格式。相比 XML 格式, JSON 格式有两个显著的优点:

- 书写简单, 一目了然;
- 符合 JavaScript 原生语法, 可以由解释引擎直接处理, 不用另外添加解析代码。所以, JSON 迅速被接受, 已经成为各大网站交换数据的标准格式, 并被写入标准。

每个 JSON 对象就是一个值, 可能是一个数组或对象, 也可能是一个原始类型的值。总之, 只能是一个值, 不能是两个或更多的值。JSON 对值的类型和格式有严格的规定:

- 复合类型的值只能是数组或对象, 不能是函数、正则表达式对象、日期对象。
- 原始类型的值只有四种: 字符串、数值 (必须以十进制表示)、布尔值和 `null` (不能使用 `NaN`, `Infinity`, `-Infinity` 和 `undefined`)。
- 字符串必须使用双引号表示, 不能使用单引号。
- 对象的键名必须放在双引号里面。
- 数组或对象最后一个成员的后面, 不能加逗号。
- JSON文件中不能添加注释。
- JSON文件中只有 5 种符号是合法的: 冒号 `:`, 双引号 `"`, 逗号 `,`, 中括号 `[]`, 大括号 `{}`

以下都是合法的 JSON。

```
[ "one", "two", "three" ]
{ "one": 1, "two": 2, "three": 3 }
{ "names": [ "张三", "李四" ] }
[ { "name": "张三" }, { "name": "李四" } ]``
```

以下都是不合法的 JSON。

```
``json
{ name: "张三", 'age': 32 } // 属性名必须使用双引号
[ 32, 64, 128, 0xFFF ] // 不能使用十六进制值
{ "name": "张三", "age": undefined } // 不能使用 undefined
{ "name": "张三",
  "birthday": new Date('Fri, 26 Aug 2011 07:13:10 GMT'),
  "getName": function () {
    return this.name;
  }
} // 属性值不能使用函数和日期对象
```

`null`、空数组和空对象都是合法的 JSON 值。

2. JSON 对象

JSON 对象是 JavaScript 的原生对象，用来处理 JSON 格式数据。它有两个静态方法：`JSON.stringify()` 和 `JSON.parse()`。

2.1. JSON.stringify()

2.1.1. 基本用法

`JSON.stringify()` 方法用于将一个值转为 JSON 字符串。该字符串符合 JSON 格式，并且可以被 `JSON.parse()` 方法还原。

```
JSON.stringify('abc') // '"abc"'
JSON.stringify(1) // "1"
JSON.stringify(false) // "false"
JSON.stringify([]) // "[]"
JSON.stringify({}) // "{}"
JSON.stringify([1, "false", false]) // '[1,"false",false]'
JSON.stringify({ name: "张三" }) // '{"name":"张三}"'
```

对于原始类型的字符串，转换结果会带双引号。

```
JSON.stringify('foo') === "foo" // false
JSON.stringify('foo') === "\"foo\"" // true
JSON.stringify('foo') === '"foo"' // true
```

上面代码中，字符串 `foo`，被转成了 `"\"foo\""`。这是因为将来还原的时候，内层双引号可以让 JavaScript 引擎知道，这是一个字符串，而不是其他类型的值。

```
JSON.stringify(false) // "false"
JSON.stringify('false') // "\"false\""
```

上面代码中，如果不是内层的双引号，将来还原的时候，引擎就无法知道原始值是布尔值还是字符串。

如果对象的属性是 `undefined`、函数或 XML 对象，该属性会被 `JSON.stringify()` 过滤。

```
let obj = {
  a: undefined, // 属性值是 undefined, 会被 JSON.stringify() 过滤
  b: function () {} // 属性值是 函数, 会被 JSON.stringify() 过滤
};
JSON.stringify(obj) // "{}"
```

如果数组的成员是 `undefined`、函数或 XML 对象，则这些值被转成 `null`。

```
let arr = [undefined, function () {}];
JSON.stringify(arr) // "[null,null]"
```

正则对象会被转成空对象。

```
JSON.stringify(/foo/) // "{}"
```

JSON.stringify()方法会忽略对象的不可遍历的属性。

```
let obj = {};
Object.defineProperty(obj, {
  'foo': {
    value: 1,
    enumerable: true
  },
  'bar': {
    value: 2,
    enumerable: false // bar 是 obj 对象的不可遍历属性，JSON.stringify() 方法会忽略这个属性。
  }
});
JSON.stringify(obj); // '{"foo":1}'
```

2.2. 第二个参数

JSON.stringify() 方法还可以接受一个数组，作为第二个参数，指定参数对象的哪些属性需要转成字符串。

```
let obj = {
  'prop1': 'value1',
  'prop2': 'value2',
  'prop3': 'value3'
};
let selectedProperties = ['prop1', 'prop2'];
JSON.stringify(obj, selectedProperties); // '{"prop1":"value1","prop2":"value2}"'
```

上面代码中，JSON.stringify() 方法的第二个参数指定，只转 prop1 和 prop2 两个属性。

这个类似白名单的数组，只对对象的属性有效，对数组无效。

```
JSON.stringify(['a', 'b'], ['0']); // "["a","b"]"
JSON.stringify({0: 'a', 1: 'b'}, ['0']); // '{"0":"a}"'
```

上面代码中，第二个参数指定 JSON 格式只转 0 号属性，实际上对数组是无效的，只对对象有效。

第二个参数还可以是一个函数，用来更改 `JSON.stringify()` 的返回值。

```
function f(key, value) {
  if (typeof value === "number") {
    value = 2 * value;
  }
  return value;
}
JSON.stringify({ a: 1, b: 2 }, f); // '{"a": 2,"b": 4}'
```

上面代码中的 `f` 函数，接受两个参数，分别是被转换的对象的键名和键值。如果键值是数值，就将它乘以 2，否则就原样返回。

传第二个参数是函数时，这个处理函数是递归处理所有的键。

```
let obj = {a: {b: 1}};
function f(key, value) {
  console.log("[ "+ key + "]: " + value);
  return value;
}
JSON.stringify(obj, f)
// []:[object Object]
// [a]:[object Object]
// [b]:1
// '{"a":{"b":1}}'
```

上面代码中，对象 `obj` 一共会被 `f` 函数处理三次，输出的最后那行是 `JSON.stringify()` 的默认输出。第一次键名为空，键值是整个对象 `obj`；第二次键名为 `a`，键值是 `{b: 1}`；第三次键名为 `b`，键值为 `1`。

递归处理中，每一次处理的对象，都是前一次返回的值。

```
let obj = {a: 1};
function f(key, value) {
  if (typeof value === 'object') {
    return {b: 2};
  }
  return value * 2;
}
JSON.stringify(obj, f); // '{"b": 4}'
```

上面代码中，`f` 函数修改了对象 `obj`，接着 `JSON.stringify()` 方法就递归处理修改后的对象 `obj`。

如果处理函数返回 `undefined` 或没有返回值，则该属性会被忽略。

```
function f(key, value) {
  if (typeof(value) === "string") {
```

```
    return undefined;
  }
  return value;
}
JSON.stringify({ a: "abc", b: 123 }, f); // '{"b": 123}'
```

上面代码中，`a` 属性经过处理后，返回 `undefined`，于是该属性被忽略了。

2.3. 第三个参数

`JSON.stringify()` 还可以接受第三个参数，用于增加返回的 JSON 字符串的可读性。默认返回的是单行字符串，对于大型的 JSON 对象，可读性非常差。第三个参数使得每个属性单独占据一行，并且将每个属性前面添加指定的前缀（不超过10个字符）。

```
JSON.stringify({ p1: 1, p2: 2 }); // 默认输出: '{"p1":1,"p2":2}'
JSON.stringify({ p1: 1, p2: 2 }, null, '\t'); // 分行输出，在每个属性前面添加一个制表符，然后分行显示。
// '{
//   "p1": 1,
//   "p2": 2
// }'
```

第三个属性如果是一个数字，则表示每个属性前面添加的空格（最多不超过10个）。

```
JSON.stringify({ p1: 1, p2: 2 }, null, 2);
/*
"{
  "p1": 1,
  "p2": 2
}"
*/
```

3. 参数对象的 toJSON() 方法

如果参数对象有自定义的 `toJSON()` 方法，那么 `JSON.stringify()` 会使用这个方法的返回值作为参数，而忽略原对象的其他属性。

```
let user = {
  firstName: '三',
  lastName: '张',
  get fullName() {
    return this.lastName + this.firstName;
  }
};
JSON.stringify(user); // '{"firstName":"三","lastName":"张","fullName":"张三}"
```

为这个对象加上 `toJSON()` 方法:

```
let user = {
  firstName: '三',
  lastName: '张',
  get fullName(){
    return this.lastName + this.firstName;
  },
  toJSON: function () {
    return {
      name: this.lastName + this.firstName
    };
  }
};
JSON.stringify(user); // '{"name":"张三}"
```

上面代码中, `JSON.stringify()` 发现参数对象有 `toJSON()` 方法, 就直接使用这个方法的返回值作为参数, 而忽略原对象的其他参数。

`Date` 对象就有一个自己的 `toJSON()` 方法。

```
let date = new Date('2015-01-01');
date.toJSON() // "2015-01-01T00:00:00.000Z"
JSON.stringify(date) // ""2015-01-01T00:00:00.000Z""
```

上面代码中, `JSON.stringify()` 发现处理的是 `Date` 对象实例, 就会调用这个实例对象的 `toJSON()` 方法, 将该方法的返回值作为参数。

`toJSON()` 方法的一个应用是, 将正则对象自动转为字符串。因为 `JSON.stringify()` 默认不能转换正则对象, 但是设置了 `toJSON()` 方法以后, 就可以转换正则对象了。

```
let obj = {
  reg: /foo/
};
JSON.stringify(obj) // 不设置 toJSON 方法时: '{"reg":{}}'

// 设置 toJSON 方法时
RegExp.prototype.toJSON = RegExp.prototype.toString;
JSON.stringify(/foo/) // '"/foo/'
```

上面代码在正则对象的原型上面部署了 `toJSON()` 方法, 将其指向 `toString()` 方法, 因此转换成 JSON 格式时, 正则对象就先调用 `toJSON()` 方法转为字符串, 然后再被 `JSON.stringify()` 方法处理。

4. JSON.parse()

`JSON.parse()` 方法用于将 JSON 字符串转换成对应的值。

```
JSON.parse('{}') // {}
JSON.parse('true') // true
JSON.parse('"foo"') // "foo"
JSON.parse('[1, 5, "false"]') // [1, 5, "false"]
JSON.parse('null') // null

let o = JSON.parse('{"name": "张三"}');
o.name // 张三
```

如果传入的字符串不是有效的 JSON 格式，JSON.parse()方法将报错。

```
JSON.parse("'String'") // illegal single quotes
// SyntaxError: Unexpected token ILLEGAL
```

上面代码中，双引号字符串中是一个单引号字符串，因为单引号字符串不符合 JSON 格式，所以报错。

为了处理解析错误，可以将 JSON.parse() 方法放在 try...catch 代码块中。

```
try {
  JSON.parse("'String'");
} catch(e) {
  console.log('parsing error');
}
```

JSON.parse() 方法可以接受一个处理函数，作为第二个参数，用法与 JSON.stringify() 方法类似。

```
function f(key, value) {
  if (key === 'a') {
    return value + 10;
  }
  return value;
}
JSON.parse('{"a": 1, "b": 2}', f); // {a: 11, b: 2}
```

上面代码中，JSON.parse() 的第二个参数是一个函数，如果键名是 a，该函数会将键值加上 10。