ES_05_解构:更方便的数据访问

解构为何有用

在 ES5 及更早版本中,从对象或数组中获取信息、并将特定数据存入本地变量,需要书写许 多并且相似的代码。例如:

```
let options = {
        repeat: true,
        save: false
    };
// 从对象中提取数据
let repeat = options.repeat,
    save = options.save;
```

此代码提取了 options 对象的 repeat 与 save 值,并将其存在同名的本地变量上。虽然 这段代码看起来简单,但想象一下若有大量变量需要处理,你就必须逐个为其赋值;并且若 有一个嵌套的数据结构需要遍历以寻找信息,你可能会为了一点数据而挖掘整个结构。

这就是 ES6 为何要给对象与数组添加解构。当把数据结构分解为更小的部分时,从中提取你 要的数据会变得容易许多。很多语言都能用精简的语法来实现解构,让它更易使用。 ES6 的 解构实际使用的语法其实你早已熟悉,那就是对象与数组的字面量语法。

对象解构

```
let node = {
        type: "Identifier",
        name: "foo"
    };
let { type, name } = node;
console.log(type);      // "Identifier"
console.log(name);      // "foo"
```

不要遗忘初始化器

当使用解构来配合 var 、 let 或 const 来声明变量时,必须提供初始化器(即等号 右边的值)。下面的代码都会因为缺失初始化器而抛出错误:

```
// 语法错误!
var { type, name }; // 语法错误!
let { type, name };
// 语法错误!
const { type, name };
```

const 总是要求有初始化器,即使没有使用解构的变量;而 var 与 let 则仅在使用 解构时才作此要求。

以上对象解构示例都用于变量声明

不过,也可以在赋值的时候使用解构。例如,你可能想在变量声明之后改变它们的值,如下所示:

```
let node = {
        type: "Identifier",
        name: "foo"
    },
    type = "Literal",
    name = 5;
// 使用解构来分配不同的值
({ type, name } = node);
console.log(type);      // "Identifier"
console.log(name);      // "foo"
```

在任何期望有个值的位置都 可以使用解构赋值表达式。例如,传递值给函数:

```
let node = {
        type: "Identifier",
        name: "foo"
    },
    type = "Literal",
    name = 5;
function outputInfo(value) {
    console.log(value === node);
}
outputInfo({ type, name } = node);
console.log(type);      // "Identifier"
console.log(name);      // "foo"
```

当解构赋值表达式的右侧(= 后面的表达式)的计算结果为 null 或 undefined 时, 会抛出错误。因为任何读取 null 或 undefined 的企图都会导致"运行时"错误 (runtime error)。

默认值

当你使用解构赋值语句时,如果所指定的本地变量在对象中没有找到同名属性,那么该变量会被赋值为 undefined 。例如:

```
let node = {
        type: "Identifier",
        name: "foo"
    };
let { type, name, value } = node;
console.log(type);
console.log(name);
console.log(value);
// "Identifier"
// "foo"
// undefined
```

你可以选择性地定义一个默认值,以便在指定属性不存在时使用该值。若要这么做,需要在属性名后面添加一个等号并指定默认值,就像这样:

```
let node = {
        type: "Identifier",
        name: "foo"
    };
let { type, name, value = true } = node;
console.log(type);
console.log(name);
console.log(value);
// "Identifier"
// "foo"
// true
```

在此例中,变量 value 被指定了一个默认值 true ,只有在 node 的对应属性缺失、或对 应的属性值为 undefined 的情况下,该默认值才会被使用。由于此处不存在 node.value 属 性,变量 value 就使用了该默认值。这种工作方式很像函数参数的默认值

赋值给不同的本地变量名

至此的每个解构赋值示例都使用了对象中的属性名作为本地变量的名称,例如,把 node.type 的值存储到 type 变量上。若想使用相同名称,这么做就没问题,但若你不想呢? ES6 有一个扩展语法,允许你在给本地变量赋值时使用一个不同的名称,而且该语法看上去就像是使用对象字面量的非简写的属性初始化。这里有个示例:

```
let node = {
        type: "Identifier",
        name: "foo"
    };
let { type: localType, name: localName } = node;
console.log(localType);     // "Identifier"
console.log(localName);     // "foo"
```

type: localType 这种语法表示要读取名为 type 的属性,并把它的 值存储在变量 localType 上。

该语法实际上与传统对象字面量语法相反,传统语法将名称放在冒号左边、值放在冒号右边;而在本例中,则是名称在右边,需要进行值读取的位置则被放在了左边。

你也可以给变量别名添加默认值

```
let node = {
        type: "Identifier"
    };
let { type: localType, name: localName = "bar" } =
node;
console.log(localType);     // "Identifier"
console.log(localName);     // "bar"
```

而对象解构也可被用于从嵌套的对象结构(即:对象的属性可能还是一个对象)中提取属性值。

嵌套的对象解构

使用类似于对象字面量的语法,可以深入到嵌套的对象结构中去提取你想要的数据。这里有个示例:

```
let node = {
        type: "Identifier",
        name: "foo",
        loc: {
            start: {
                line: 1,
column: 1 },
            end: {
                line: 1,
column: 4 }
} };
let { loc: { start }} = node;
console.log(start.line);
console.log(start.column);
// 1 // 1
```

你还能更进一步,在对象的嵌套解构中同样能为本地变量使用不同的名称:

记住上一节介绍过的,每当有一个冒号在解构模式中出现,就意味着冒号之前的标识符代表需要检查的位置,而冒号右侧则是赋值的目标。当冒号右侧存在花括号时,表示目标被嵌套在对象的更深一层中。

```
let node = {
        type: "Identifier",
        name: "foo",
        loc: {
            start: {
                line: 1,
column: 1 },
            end: {
                line: 1,
column: 4 }
} };
// 提取 node.loc.start
let { loc: { start: localStart }} = node;
console.log(localStart.line);   // 1
console.log(localStart.column); // 1
```

使用嵌套的解构时需要小心,因为你可能无意中就创建了一个没有任何效果的语句。空白花括号在对象解构中是合法的,然而它不会做任何事。例如:

```
// 没有变量被声明!
let { loc: {} } = node;
```

在此语句中并未声明任何变量绑定。由于花括号在右侧, loc 被作为需检查的位置来使 用,而不会创建变量绑定。这种情况仿佛是想用等号来定义一个默认值,但却被语法判 断为想用冒号来定义一个位置。这种语法将来可能是非法的,然而现在它只是需要留意 的一个疑难点。

数组解构

数组解构的语法看起来与对象解构非常相似,只是将对象字面量替换成了数组字面量。数组解构时,解构作用在数组内部的位置上,而不是作用在对象的具名属性上,例如:

```
let colors = [ "red", "green", "blue" ];
let [ firstColor, secondColor ] = colors;
console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

这些值被选择,是由于它们在数组中的位置,实际的变量名称是任意 的(与位置无关)。任何没有在解构模式中明确指定的项都会被忽略。记住,数组本身并没 有以任何方式被改变。

只给感兴趣的项提供变量名

```
let colors = [ "red", "green", "blue" ];
let [ , , thirdColor ] = colors;
console.log(thirdColor);        // "blue"
```

解构赋值

你可以在赋值表达式中使用数组解构,但是与对象解构不同,不必将表达式包含在圆括号内,例如:

```
let colors = [ "red", "green", "blue" ],
    firstColor = "black",
    secondColor = "purple";
[ firstColor, secondColor ] = colors;
console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

大多数情况下,以上可能就是数组解构赋值你需要了解的全部内容,但其实 还有一个很细微却又可能很有用的用法。

数组解构赋值有一个非常独特的用例,能轻易地互换两个变量的值。互换变量值在排序算法 中十分常用,而在 ES5 中需要使用第三个变量作为临时变量,正如下例:

```
// 在 ES5 中互换值
let a = 1, b = 2,
tmp;
tmp = a;
a = b;
b = tmp;
console.log(a);
console.log(b);
// 2 // 1
```

```
// 在 ES6 中互换值
let a = 1, b = 2;
[ a, b ] = [ b, a ];
console.log(a);     // 2
console.log(b);     // 1
```

默认值

数组解构赋值同样允许在数组任意位置指定默认值。当指定位置的项不存在、或其值为 undefined

```
let colors = [ "red" ];
let [ firstColor, secondColor = "green" ] = colors;
console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

嵌套的解构

```
let colors = [ "red", [ "green", "lightgreen" ],
"blue" ]; // 随后
let [ firstColor, [ secondColor ] ] = colors;
console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

剩余项

```
let colors = [ "red", "green", "blue" ];
let [ firstColor, ...restColors ] = colors;
console.log(firstColor);        // "red"
console.log(restColors.length); // 2
console.log(restColors[0]);     // "green"
console.log(restColors[1]);     // "blue"
```

但它还有另一个有用的功能

方便地克隆数组在 JS 中是个明显被遗漏的功能。在 ES5 中开发者往往使用的是一个简单的 方式,也就是用 concat() 方法来克隆数组,例如:

```
// 在 ES5 中克隆数组
var colors = [ "red", "green", "blue" ];
var clonedColors = colors.concat();
console.log(clonedColors);      //"[red,green,blue]"
```

尽管 concat() 方法的本意是合并两个数组,但不使用任何参数来调用此方法,就会获得原 数组的一个克隆品

在 ES6 中,你可以使用剩余项的语法来达到同样效果。实现如下:

```
// 在 ES6 中克隆数组
let colors = [ "red", "green", "blue" ];
let [ ...clonedColors ] = colors;
console.log(clonedColors);      //"[red,green,blue]"
```

混合解构

对象与数组解构能被用在一起,以创建更复杂的解构表达式。在对象与数组混合而成的结构中,这么做便能准确提取其中你想要的信息片段

```
let node = {
        type: "Identifier",
        name: "foo",
        loc: {
            start: {
                line: 1,
column: 1 },
            end: {
                line: 1,
column: 4 }
},
        range: [0, 3]
    };
let {
    loc: { start },
    range: [ startIndex ]
} = node;
console.log(start.line);
console.log(start.column);
console.log(startIndex);
// 1 // 1 // 0
```

参数解构

解构还有一个特别有用的场景,即在传递函数参数时。当 JS 的函数接收大量可选参数时,一 个常用模式是创建一个 options 对象,其中包含了附加的参数,就像这样:

```
// options 上的属性表示附加参数
function setCookie(name, value, options) {
    options = options || {};
    let secure = options.secure,
        path = options.path,
        domain = options.domain,
        expires = options.expires;
// 设置 cookie 的代码
}
// 第三个参数映射到 options
setCookie("type", "js", {
    secure: true,
    expires: 60000
});
```

很多 JS 的库都包含了类似于此例的 setCookie() 函数。在此函数内, name 与 value 参 数是必需的,而 secure 、 path 、 domain 与 expires 则不是。并且因为此处对于其余 数据并没有顺序要求,将它们作为 options 对象的具名属性会更有效率,而无须列出一堆额 外的具名参数。这种方法很有用,但无法仅通过查看函数定义就判断出函数所期望的输入, 你必须阅读函数体的代码。

参数解构提供了更清楚地标明函数期望输入的替代方案。它使用对象或数组解构的模式替代 了具名参数。要看到其实际效果,请查看下例中重写版本的 setCookie() 函数:

```
function setCookie(name, value, { secure, path,
domain, expires }) { // 设置 cookie 的代码
}
setCookie("type", "js", {
    secure: true,
    expires: 60000
});
```

解构的参数是必需的

```
// 出错!
setCookie("type", "js");
```

调用时第三个参数缺失了,因此它不出预料地等于 undefined 。这导致了一个错误,因为参 数解构实际上只是解构声明的简写。当 setCookie() 函数被调用时, JS 引擎实际上是这么 做的:

```
function setCookie(name, value, options) {
    let { secure, path, domain, expires } = options;
// 设置 cookie 的代码
}
```

既然在赋值右侧的值为 null 或 undefined 时,解构会抛出错误,那么未向 setCookie() 函数传递第三个参数就同样会出错。
若你让解构的参数作为必选参数,那么上述行为并不会令人困扰。但若你要求它是可选的,
可以给解构的参数提供默认值来处理这种行为,就像这样:

参数解构的默认值

你可以为参数解构提供可解构的默认值,就像在解构赋值时所做的那样,只需在其中每个参数后面添加等号并指定默认值即可。例如:

```
function setCookie(name, value,
    {
        secure = false,
        path = "/",
        domain = "example.com",
        expires = new Date(Date.now() + 360000000)
} = {} )
```