PivotStudio-internship-2022Spring-XuQiao

PivotStudio-internship-2022Spring-XuQiao

Babel计算器的实现

运行方式

`./test/index.js` 内是测试用例，可以随意写相关的代码，然后打命令`npm run test`执行编译，编译后的文件存放在`./test/compiled/index.js` 中。

主要功能

- 两个数低精度下精准的四则运算操作。（自定义方法，重复计算时只声明一次自定义方法）
- 箭头函数编译为普通函数（考虑了`this`的情形，方案是在箭头函数所在的作用域中声明`var _this = this`，箭头函数内部的`this`用`_this`代替）
- let编译为var。（使用匿名自执行函数来模拟块级作用域，特殊的for循环内的小括号里由`let`声明的变量编译后为了防止重名，会将其及其在内部的引用全部改名）
- 一些数组方法的polyfill（`forEach`、`find`、`filter`）

例如：

编译前：

```
//Before compiling
console.log(1 + 2)
console.log(1 - 2)
console.log(1 * 2)
console.log(1 / 2)

console.log(3 + 4)
console.log(3 - 4)
console.log(3 * 4)
console.log(3 / 4)

let a = 1;

let b = (param1, param2) => {
    console.log(param1);
    return param2;
};

let c = (param1, param2) => param1 + param2;

for (let i = 1; i < 10; i++) {
    setTimeout(function() {
        console.log(i)
    }, 1000)
};
```

```
let arr1 = [1, 3, 5, 7, 9];
arr1.forEach(function(val, index, arr) {
    console.log(val + index)
});
arr1.filter(function(val, index, arr) {
    if (index < 3) return true;
});
arr1.find(function(val, index, arr) {
    if (index = 3) return true;
})

function test() {
    let a = 1;
}
```

编译后：

```
Array.prototype.myFind = function(fn, context = null) {
    var arr = this;
    var len = arr.length;
    var index = 0,
        k = 0;
    var targetValue;

    if (typeof fn !== 'function') {
        throw new TypeError(fn + ' is not a function');
    }

    while (index < len) {
        if (index in arr) {
            var result = fn.call(context, arr[index], index, arr);
            if (result) targetValue = arr[index];
            break;
        }

        index++;
    }

    return targetValue;
};

Array.prototype.myFilter = function(fn, context = null) {
    var arr = this;
    var len = arr.length;
    var index = 0,
        k = 0;
    var newArr = [];

    if (typeof fn !== 'function') {
        throw new TypeError(fn + ' is not a function');
    }
```

```
    while (index < len) {
        if (index in arr) {
            var result = fn.call(context, arr[index], index, arr);
            if (result) newArr[k++] = arr[index];
        }

        index++;
    }

    return newArr;
};

Array.prototype.myForEach = async function(fn, context = null) {
    var index = 0;
    var arr = this;

    if (typeof fn !== 'function') {
        throw new TypeError(fn + ' is not a function');
    }

    while (index < arr.length) {
        if (index in arr) {
            try {
                await fn.call(context, arr[index], index, arr);
            } catch (e) {
                console.log(e);
            }
        }

        index++;
    }
};

function _cstmDiv(arg1, arg2) {
    var multi = _getMulti(arg1, arg2);

    return arg1 * multi / (arg2 * multi);
}

function _cstmMulti(arg1, arg2) {
    var multi = _getMulti(arg1, arg2);

    return arg1 * multi * (arg2 * multi) / (multi * multi);
}

function _cstmMinus(arg1, arg2) {
    var multi = _getMulti(arg1, arg2);

    return (arg1 * multi - arg2 * multi) / multi;
}

function _cstmAdd(arg1, arg2) {
    var multi = _getMulti(arg1, arg2);
```

```
    return (arg1 * multi + arg2 * multi) / multi;
}

function _getMulti(arg1, arg2) {
    var multi1;
    var multi2;

    for (multi1 = 1; multi1 < Infinity; multi1 = multi1 * 10) {
        if (arg1 * multi1 % 1 === 0) {
            break;
        }
    }

    for (multi2 = 1; multi2 < Infinity; multi2 = multi2 * 10) {
        if (arg2 * multi2 % 1 === 0) {
            break;
        }
    }

    return multi1 > multi2 ? multi1 : multi2;
}

console.log(_cstmAdd(1, 2));
console.log(_cstmMinus(1, 2));
console.log(_cstmMulti(1, 2));
console.log(_cstmDiv(1, 2));
console.log(_cstmAdd(3, 4));
console.log(_cstmMinus(3, 4));
console.log(_cstmMulti(3, 4));
console.log(_cstmDiv(3, 4));
var a = 1;

var b = function(param1, param2) {
    console.log(param1);
    return param2;
};

var c = function(param1, param2) {
    return param1 + param2;
};

for (var _i = 1; _i < 10; _i++) {
    (function() {
        setTimeout(function() {
            console.log(_i);
        }, 1000);
    })();
}

;
var arr1 = [1, 3, 5, 7, 9];
arr1.myForEach(function(val, index, arr) {
    console.log(val + index);
```

```
});
arr1.myFilter(function(val, index, arr) {
    if (index < 3) return true;
});
arr1.myFind(function(val, index, arr) {
    if (index = 3) return true;
});
```

文件结构

```
---node_modules
|
|
---src:其中的index.js为插件源码
|
|
---test:放置测试用例和编译后的结果
|
|
---babel.config.json: babel配置文件
|
|
---package.json
|
|
---DailyReport.md: 日志
|
|
...
```

一些细节：

- babel/types并没有直接的在编译中添加注释的方法，目前在https://github.com/babel/babel/issues/5897 中查到有开发者在issue中提到要加相应的方法（类似于`t.comment(type, content)`）来创建注释，目前该issue还是open状态。在babel中注释并不作为一个单独的节点处理，而是作为某个节点的属性（`leadingComments`和`trailingComments`）。但是这个属性并不能在构建节点时加上，只能手动在节点对象添加属性。
- `path.scope.rename(...)`可以将本作用域下的所有声明变量和对它们的引用改名，并保证不重复。
- 为了防止编译时对相同方法重复创建补丁导致重复声明，我添加了一个布尔值进行验证，一旦创建了补丁函数，该布尔值就永久更改。
- 对特定作用域下的某些节点的处理：新创建一个visitor对象，使用path.traverse(visitor)来进行新的深度优先遍历。
- 在let编译为var时，对可能存在的块级作用域的处理：将原本的块级作用域中的所有代码套上一个匿名自执行函数来模拟块级作用域。函数作用域不做处理，只需要改名即可。需要注意的是，for循环中的小括号里由let声明的变量在编译后会称为上级作用域下的变量，为防止重名，我先获取这个变量和子作用域对其所有的引用，再使用`path.scope.rename(...)`改名。
- 手写`Array.prototype.myForEach()`方法考虑了异步的情况。