10 个常考的前端手写题，你全都会吗？

今天分享一下 10 个常见的前端手写功能下集，可以让你在数据处理上得心应手，让你的开发工作事半功倍。开始吧！

1. 实现继承

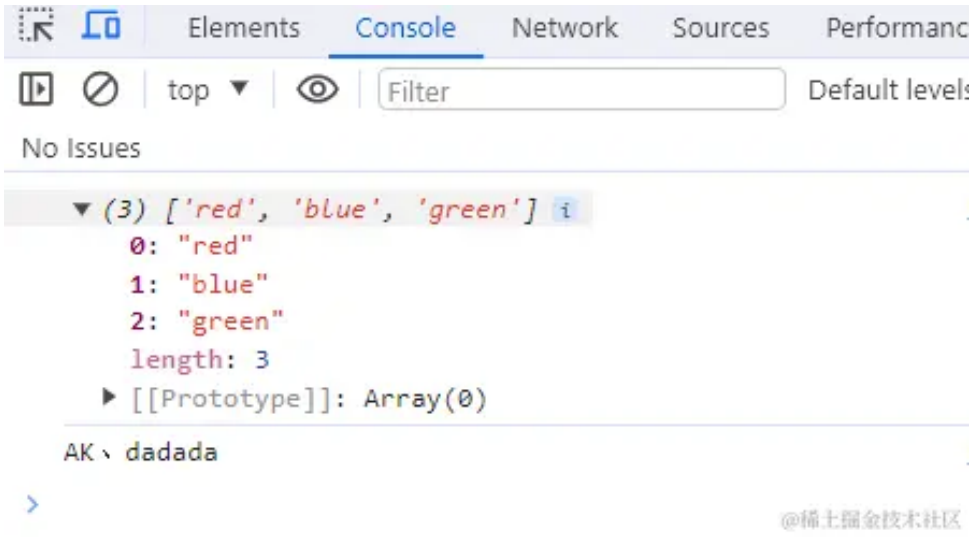
ES5 继承（寄生组合式继承）

寄生组合式继承是对组合式继承（调用了 2 次父构造方法）的改进，使用父类的原型的副本来作为子类的原型，这样就只调用一次父构造函数，避免了创建不必要的属性。

Plain Text

```
function Parent (name) {
    this.name = name;
    this.colors = ['red', 'blue', 'green'];
}
Parent.prototype.getName = function () {
    console.log(this.name)
}
function Child (name, age) {
    Parent.call(this, name);//借用构造函数的方式来实现属性的继承和传参
    this.age = age;
}

//这里不用Child.prototype = new Parent()原型链方式的原因是会调用2次父类的构造方法，导致子类的原型上多了不需要的父类属性
Child.prototype = Object.create(Parent.prototype);//这里就是对组合继承的改进,创建了父类原型的副本
Child.prototype.constructor = Child;//把子类的构造指向子类本身

var child1 = new Child('AK、dadada', '18');
console.log(child1.colors);//[ 'red', 'blue', 'green' ]
child1.getName();//AK、dadada
```

测试结果：



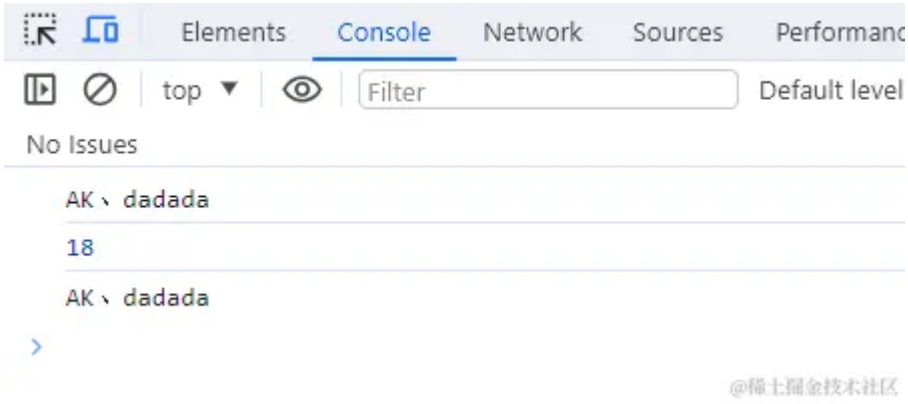
ES6 继承

在 ES6 中，可以使用 class 类去实现继承。使用 extends 表明继承自哪个父类，并且在子类构造函数中必须调用 super。

Plain Text

```
class Parent {
  constructor(name) {
    this.name = name;
  }
  getName() {
    console.log(this.name);
  }
}

class Child extends Parent {
  constructor(name, age) {
    //使用this之前必须先调用super(),它调用父类的构造函数并绑定父类的属性和方法
    super(name);
    //之后子类的构造函数再进一步访问和修改 this
    this.age = age;
  }
}

// 测试
let child = new Child("AK、dadada", 18);
console.log(child.name); // AK、dadada
console.log(child.age); // 18
child.getName(); // AK、dadada
```

测试结果：



ES5 继承和 ES6 继承的区别：

- `ES5` 继承是先创建子类的实例对象，然后再将父类方法添加到 this（`Parent.call(this)` ）上。
- `ES6` 的继承不同，实质是先将父类实例对象的属性和方法，加到 this 上面（所以必须先调用 super 方法），然后再用子类的构造函数修改 this。

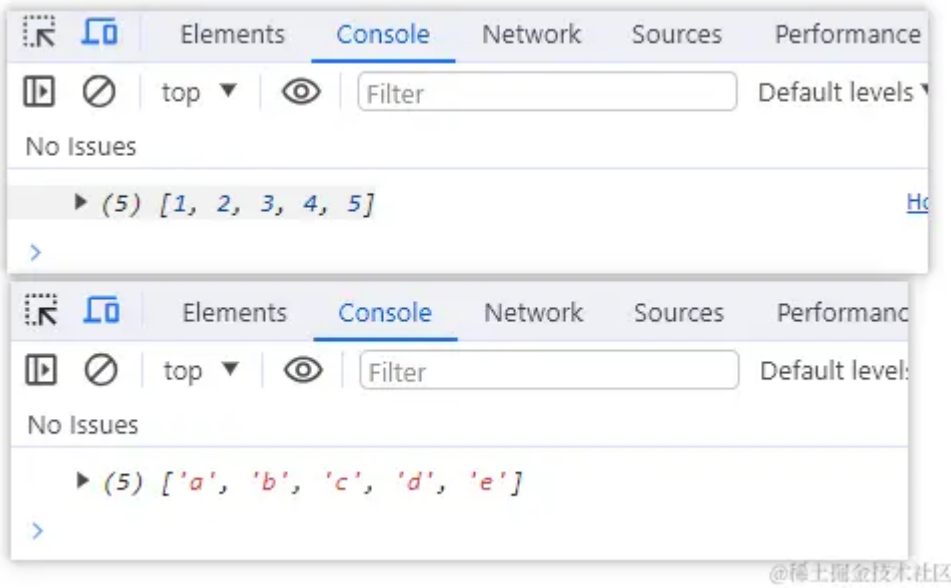
2. 数组排序

sort 排序

Plain Text

```
// 对数字进行排序，简写
let arr = [3, 2, 4, 1, 5];
arr.sort((a, b) => a - b);
console.log(arr); // [1, 2, 3, 4, 5]

// 对字母进行排序
let arr = ["b", "c", "a", "e", "d"];
arr.sort((a, b) => {
  if (a > b) return 1;
  else if (a < b) return -1;
  else return 0;
});
console.log(arr); // ['a', 'b', 'c', 'd', 'e']
```

测试结果：

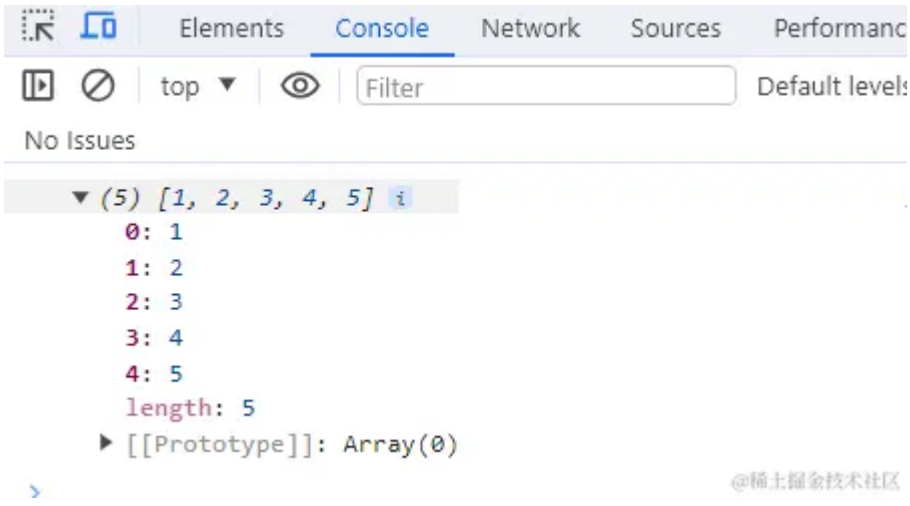


冒泡排序

Plain Text

```
function bubbleSort(arr) {
  let len = arr.length
  for (let i = 0; i < len - 1; i++) {
    // 从第一个元素开始，比较相邻的两个元素，前者大就交换位置
    for (let j = 0; j < len - 1 - i; j++) {
      if (arr[j] > arr[j + 1]) {
        let num = arr[j]
        arr[j] = arr[j + 1]
        arr[j + 1] = num
      }
    }
    // 每次遍历结束，都能找到一个最大值，放在数组最后
  }
  return arr
}

//测试
console.log(bubbleSort([2, 3, 1, 5, 4])) // [ 1, 2, 3, 4, 5 ]
```

测试结果：



3. 手写 reduce

reduce 的使用

Plain Text

```
//普通数组求和
let arr = [1,2,3,4,5,6,7,8,9,10]
arr.reduce((prev, cur) => { return prev + cur }, 0)//55
//多维数组求和
let arr = [1,2,3,[[4,5],6],7,8,9]
arr.flat(Infinity).reduce((prev, cur) => { return prev + cur }, 0)//45
//对象数组求和
let arr = [{a:9, b:3, c:4}, {a:1, b:3}, {a:3}]
arr.reduce((prev, cur) => {
    return prev + cur["a"];//13 求对象数组中所有属性为a的和
}, 0)
```

reduce 的实现

Plain Text

```
Array.prototype.myReduce = function (cb, initialValue) {
  const arr = this; //this就是调用reduce方法的数组
  let total = initialValue ? initialValue : arr[0]; //不传默认取数组第一项
  let startIndex = initialValue ? 0 : 1; // 有初始值的话从0遍历，否则从1遍历
  for (let i = startIndex; i < arr.length; i++) {
    total = cb(total, arr[i], i, arr); //参数为初始值、当前值、索引、当前数组
  }
  return total;
};

//测试
let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let res = arr.myReduce((total, cur) => {
  return total + cur;
}, 0);
console.log(res);//55
```

4. 实现观察者模式

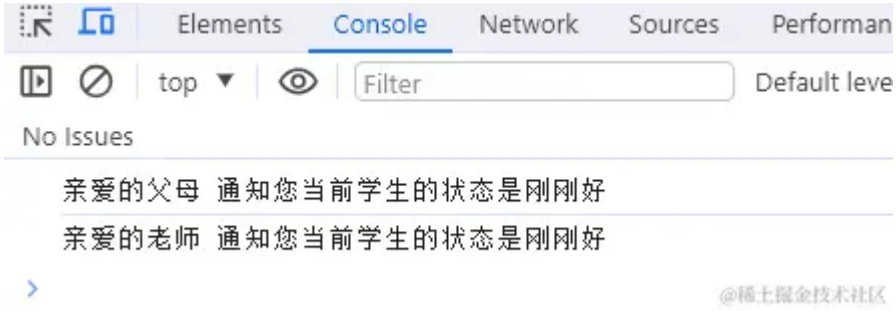
`观察者模式`：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都将得到通知。

Plain Text

```
// 被观察者 学生
class Subject {
  constructor() {
    this.state = "happy";
    this.observers = []; // 存储所有的观察者
  }
  //新增观察者
  add(o) {
    this.observers.push(o);
  }
  //获取状态
  getState() {
    return this.state;
  }
  // 更新状态并通知
  setState(newState) {
    this.state = newState;
    this.notify();
  }
  //通知所有的观察者
  notify() {
    this.observers.forEach((o) => o.update(this));
  }
}

// 观察者 父母和老师
class Observer {
  constructor(name) {
    this.name = name;
  }
  //更新
  update(student) {
    console.log(`亲爱的${this.name} 通知您当前学生的状态是${student.getState()}`);
  }
}

let student = new Subject();
```

```
let parent = new Observer("父母");
let teacher = new Observer("老师");
//添加观察者
student.add(parent);
student.add(teacher);
//设置被观察者的状态
student.setState("刚刚好");
```

测试结果：



5. 实现发布-订阅模式

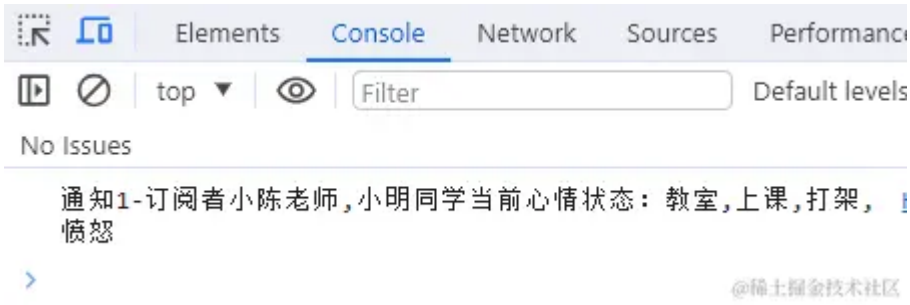
发布订阅模式跟观察者模式很像，但它发布和订阅是不互相依赖的，因为有一个`统一调度中心`

Plain Text

```
class EventBus {
  constructor() {
    // 缓存列表，用来存放注册的事件与回调
    this.cache = {};
  }

  // 订阅事件
  on(name, cb) {
    // 如果当前事件没有订阅过，就给事件创建一个队列
    if (!this.cache[name]) {
      this.cache[name] = []; //由于一个事件可能注册多个回调函数，所以使用数组来存储事件队列
    }
    this.cache[name].push(cb);
  }

  // 触发事件
  emit(name, ...args) {
    // 检查目标事件是否有监听函数队列
    if (this.cache[name]) {
      // 逐个调用队列里的回调函数
      this.cache[name].forEach((callback) => {
        callback(...args);
      });
    }
  }

  // 取消订阅
  off(name, cb) {
    const callbacks = this.cache[name];
    const index = callbacks.indexOf(cb);
    if (index !== -1) {
      callbacks.splice(index, 1);
    }
  }

  // 只订阅一次
  once(name, cb) {
```

```
38      // 执行完第一次回调函数后，自动删除当前订阅事件
39      const fn = (...args) => {
40        cb(...args);
41        this.off(name, fn);
42      };
43      this.on(name, fn);
44    }
45  }
46
47  // 测试
48  let eventBus = new EventBus();
49  let event1 = function (...args) {
50    console.log(`通知1-订阅者小陈老师,小明同学当前心情状态：${args}`)
51  };
52  // 订阅事件，只订阅一次
53  eventBus.once("teacherName1", event1);
54  // 发布事件
55  eventBus.emit("teacherName1", "教室", "上课", "打架", "愤怒");
56  eventBus.emit("teacherName1", "教室", "上课", "打架", "愤怒");
57  eventBus.emit("teacherName1", "教室", "上课", "打架", "愤怒");
```

测试结果：

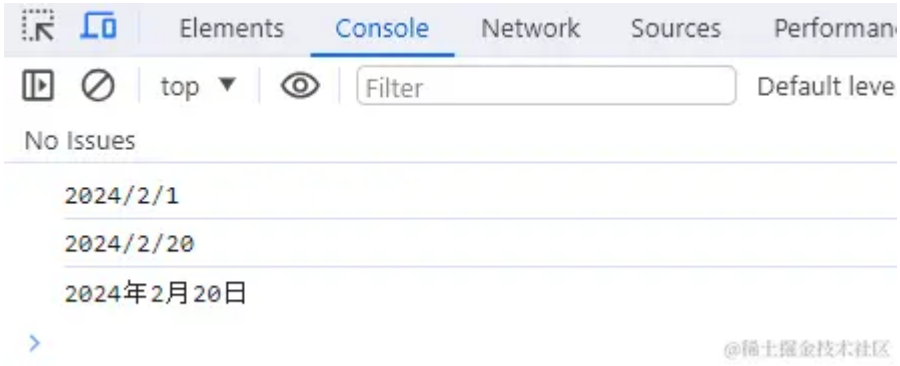


6. 实现日期格式化函数

Plain Text

```
const dateFormat = (dateInput, format) => {
  var day = dateInput.getDate();
  var month = dateInput.getMonth() + 1;
  var year = dateInput.getFullYear();
  format = format.replace(/yyyy/, year);
  format = format.replace(/MM/, month);
  format = format.replace(/dd/, day);
  console.log(format);
  return format;
};

dateFormat(new Date("2024-02-01"), "yyyy/MM/dd"); // 2024/02/01
dateFormat(new Date("2024-02-20"), "yyyy/MM/dd"); // 2024/02/20
dateFormat(new Date("2024-02-20"), "yyyy年MM月dd日"); // 2024年02月20日
```

测试结果：



7. 实现Promise.all

Plain Text

```
function all(promises) {
  return new Promise(function (resolve, reject) {
    //传入参数为一个空的可迭代对象，直接resolve
    if (promises.length === 0) {
      resolve([]);
    } else {
      const res = [];
      let count = 0;
      for (let i = 0; i < promises.length; i++) {
        //为什么不直接promise[i].then, 因为promise[i]可能不是一个promise,也可能是普通值
        Promise.resolve(promises[i])
          .then((data) => {
            res[i] = data;
            count++;
            if (count === promises.length) {
              resolve(res); //如果所有Promise都成功，则返回成功结果数组
            }
          })
          .catch((err) => {
            reject(err); //如果有一个Promise失败，则返回这个失败结果
          });
      }
    }
  });
}

// 测试
const promise1 = Promise.resolve(5);
const promise2 = 4;
const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, "AK、DADADA");
});

all([promise1, promise2, promise3]).then((values) => {
  console.log(values); //[5, 4, "AK、DADADA"]
});
```

测试结果：

8. 使用 setTimeout 实现 setInterval

`setInterval的缺点`：setInterval 的作用是`每隔一段时间执行一个函数`，但是这个执行不是真的到了时间立即执行，它真正的作用是每隔一段时间将事件加入事件队列中去，只有当当前的执行栈为空的时候，才能去从事件队列中取出事件执行。所以可能会出现这样的情况，就是当前执行栈执行的时间很长，导致事件队列里边积累多个定时器加入的事件，当执行栈结束的时候，这些事件会依次执行，因此就不能到间隔一段时间执行的效果。

`针对 setInterval 的这个缺点，我们可以使用 setTimeout 递归调用来模拟 setInterval，这样我们就确保了只有一个事件结束了，我们才会触发下一个定时器事件，这样解决了 setInterval 的问题。`

实现思路是使用递归函数，不断地去执行 setTimeout 从而达到 setInterval 的效果。

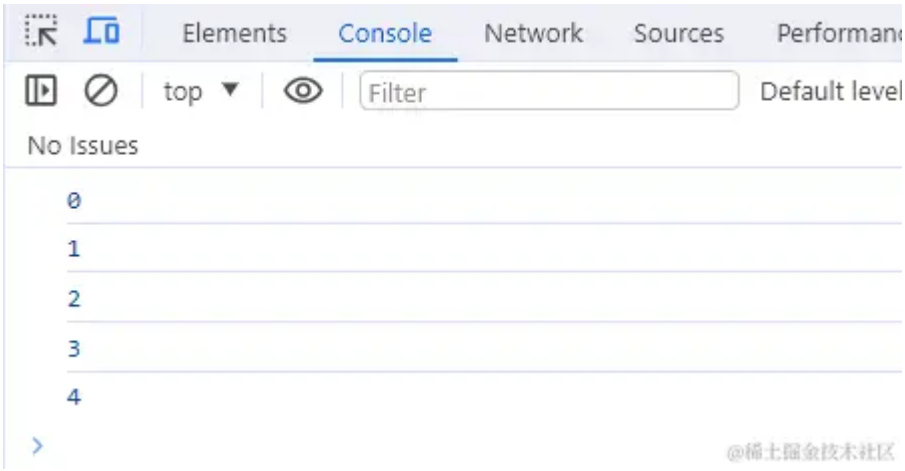
Plain Text

```
function mySetInterval(fn, timeout) {
  // 控制器，控制定时器是否继续执行
  var timer = {
    flag: true,
  };
  // 设置递归函数，模拟定时器执行
  function interval() {
    if (timer.flag) {
      fn();
      setTimeout(interval, timeout);//递归
    }
  }
  // 启动定时器
  setTimeout(interval, timeout);
  // 返回控制器
  return timer;
}

let timer = mySetInterval(() => {
  console.log("1");
}, 1000);
//3秒后停止定时器
setTimeout(() => (timer.flag = false), 3000);
```

9. 实现每隔一秒打印 1,2,3,4

Plain Text

```
// 1.使用 let 块级作用域
for (let i = 0; i < 5; i++) {
  setTimeout(() => {
    console.log(i);
  }, i * 1000);
}

// 2.使用闭包实现
for (var i = 0; i < 5; i++) {
  (function(j) {
    setTimeout(() => {
      console.log(j);
    }, j * 1000);
  })(i);
}
```



10. 循环打印红黄绿

场景：红灯 3s 亮一次，绿灯 1s 亮一次，黄灯 2s 亮一次；如何让三个灯不断**交替重复**亮灯？

红绿灯函数

Plain Text

```
function red() {
  console.log("red");
}
function green() {
  console.log("green");
}
function yellow() {
  console.log("yellow");
}
```

promise 实现

Plain Text

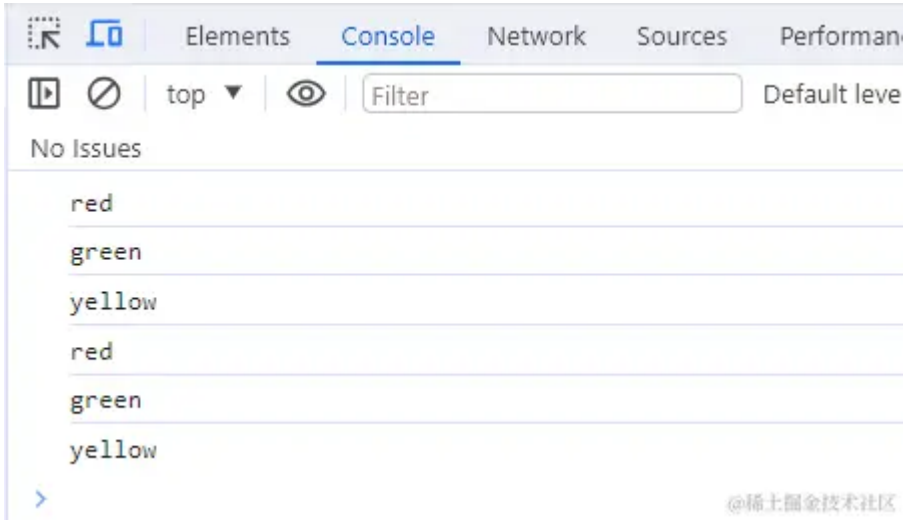
```
const task = (timer, light) =>
    new Promise((resolve, reject) => {
        setTimeout(() => {
            if (light === 'red') {
                red()
            }
            else if (light === 'green') {
                green()
            }
            else if (light === 'yellow') {
                yellow()
            }
            resolve()
        }, timer)
    })
const step = () => {
    task(3000, 'red')
        .then(() => task(2000, 'green'))
        .then(() => task(1000, 'yellow'))
        .then(step)
}
step()
```

async/await 实现

Plain Text

```
const task = (timer, light) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (light === "red") {
        red();
      } else if (light === "green") {
        green();
      } else if (light === "yellow") {
        yellow();
      }
      resolve(); //注意，要resolve让Promise状态变成fulfilled，不然会一直是pending，无法往下执行
    }, timer);
  });
};
const taskRunner = async () => {
  await task(3000, "red");
  await task(2000, "green");
  await task(1000, "yellow");
  taskRunner(); //递归
};
taskRunner();
```

测试结果：


Elements
Console
Network
Sources
Performan
top
Filter
Default leve
No Issues
red
green
yellow
red
green
yellow
@稀土掘金技术社区