动态规划

动态规划

- 资源
- 开始学习
  - Vue源码调试
  - patch VS diff
    - 最长递增子序列
  - Map
    - ES6里的Map原理（哈希表）
      - 实现Map
    - Object与Map的选择（来自红宝书6.4.3）
      - 内存占用
      - 插入性能
      - 查找速度
      - 删除性能

资源

1. 最长递增子序列
2. LeetCode300

开始学习

Vue源码调试

1. 下载vue3：`git clone https://github.com/vuejs/vue-next.git`
2. 在vue-next下安装依赖：`yarn --ignore-scripts`
3. 生成sourcemap文件，package.json

   `"dev": "node scripts/dev.js --sourcemap"`
4. 编译: `yarn dev`

   生成结果：

   packages\vue\dist\vue.global.js

   packages\vue\dist\vue.global.js.map
5. 调试范例代码： `yarn serve`
6. 在vue/examples下创建patch.html

```
<div id="app">
    <button v-on:click="add">{{count}}</button>

    <ul>
        <li v-for="item in state.arr" :key="item">{{item}}</li>
    </ul>
```

```
</div>

<script src="../dist/vue.global.js"></script>
<!-- <script src="http://unpkg.com/vue@next"></script> -->


<script>
    const {
        createApp,
        ref,
        reactive
    } = Vue
    const app = createApp({
        setup() {
            const count = ref(0)
            const state = reactive({
                arr: [0, 1, 2, 3, 4]
            })
            const add = () => {
                count.value++
                if (count.value % 2) {
                    state.arr = [1, 3, 2, 5]
                } else {
                    state.arr = [0, 1, 2, 3, 4]

                }
            }

            return {
                count,
                add,
                state
            }
        }
    }).mount('#app')
</script>
```

7. 打开地址：http://localhost:5000/packages/vue/examples/patch

patch VS diff

最长递增子序列

场景：Vue中更新阶段，新老虚拟dom的diff的时候，如果有节点移动，那么此时可以计算下dom节点中最长递增子序列，减少move，确保对dom的操作影响到最小。

```
function getSequence(arr: number[]): number[] {
  const p = arr.slice()
  const result = [0]
  let i, j, u, v, c
  const len = arr.length
  for (i = 0; i < len; i++) {
    const arrI = arr[i]
    if (arrI !== 0) {
```

```
      j = result[result.length - 1]
      if (arr[j] < arrI) {
        p[i] = j
        result.push(i)
        continue
      }
      u = 0
      v = result.length - 1
      while (u < v) {
        c = (u + v) >> 1
        if (arr[result[c]] < arrI) {
          u = c + 1
        } else {
          v = c
        }
      }
      if (arrI < arr[result[u]]) {
        if (u > 0) {
          p[i] = result[u - 1]
        }
        result[u] = i
      }
    }
  }
  u = result.length
  v = result[u - 1]
  while (u-- > 0) {
    result[u] = v
    v = p[v]
  }
  return result
}
```

Map

React Vue Diff

old cde [] 单链表 key: value Object Map

new xyde [] []

ES6以前，在js中实现“键/值”式存储可以使用Object来方便高效地完成，也就是使用对象属性作为键，再使用属性来引用值。但这种实现并非没有问题，为此TC39委员会专门为“键/值”存储定义了一个规范。

作为ES6的新增特性，Map是一种新的集合类型，为这门语言带来了真正的“键/值”存储机制。

与Object只能使用数值、字符串或者符号作为键不同，Map可以使用任何JS数据类型作为键。Map内部使用SameValueZero比较操作，基本上相当于使用严格对象相等的标准来检查键的匹配性。与Object类似，映射的值是没有限制的。

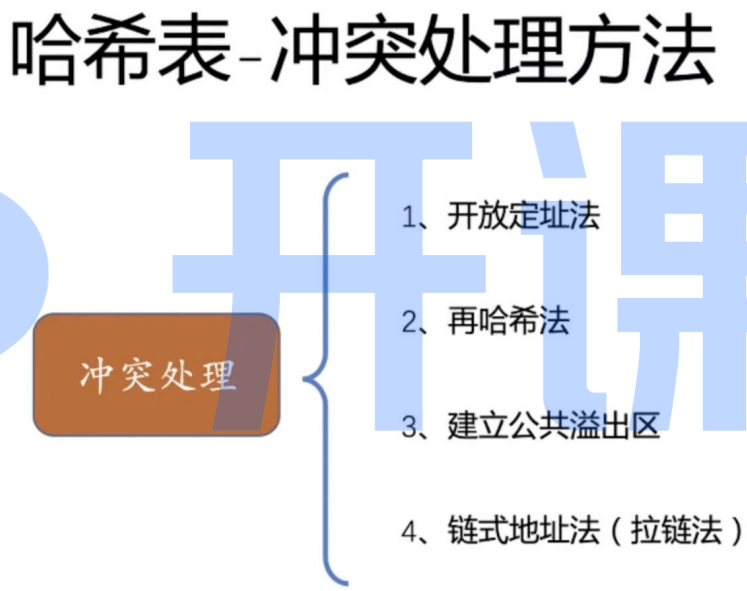
```
const o = {};

function map() {}
```

```
const m = new Map([
  ["key1", "val1"],
  ["key1", "val2"],
  [{}, "val3"],
  [{}, "val3"],
  [o, "ooo"],
  [o, "ooo"],
  [map, "哈哈"],
  [map, "哈哈"],
  [null, "null-null"],
  [null, "null-null"],
  [undefined, "undefined-undefined"],
  [undefined, "undefined-undefined"],
]);

console.log("mm", m.entries()); //sy-log
```

ES6里的Map原理（哈希表）



散列表（Hash table，也叫哈希表），是根据关键码值(Key value)而直接进行访问的数据结构。

Map的底层数据结构就是散列表，即：数组+链表，创建的时候初始化一个数组，每个节点可以为一个链表，即拉链法。

实现Map

`size`属性返回 Map 结构的成员总数。

`set`方法设置键名`key`对应的键值为`value`，然后返回整个 Map 结构。如果`key`已经有值，则键值会被更新，否则就新生成该键。

```
function MyMap() {
  this.init();
}

MyMap.prototype.init = function () {
  // this.bucket = new Array(10).fill({ key: null, value: null, next: null });
```

```
  this.bucket = new Array(8);
  for (let i = 0; i < 8; i++) {
    this.bucket[i] = {};
    this.bucket[i].next = null;
  }

  this.size = 0;
  this.head = null;
  this.tail = null;
};
MyMap.prototype.hash = function (key) {
  // 返回链表下标
  let index = 0;
  if (typeof key == "string") {
    for (let i = 0; i < 10; i++) {
      // 返回字符串第一个字符的 Unicode 编码(H 的 Unicode 值):
      index += isNaN(key.charCodeAt(i)) ? 0 : key.charCodeAt(i);
    }
  } else if (typeof key == "object") {
    index = 0;
  } else if (typeof key == "number") {
    index = key % this.bucket.length;
    console.log("index", index); //sy-log
  } else if (typeof key === "undefined") {
    index = 1;
  } else if (typeof key === "boolean") {
    index = 2;
  }

  return index % this.bucket.length;
};

// 更新
// 添加值
MyMap.prototype.set = function (key, value) {
  const i = this.hash(key);
  let target = this.bucket[i];
  while (target.next) {
    if (target.next.key === key) {
      // 更新
      target.next.value = value;
      return this;
    }
    target = target.next;
  }

  target.next = { key, value, next: null };

  // 记录顺序，set不修改原先的顺序
  if (this.size === 0) {
    // 头结点
    this.head = this.tail = target.next;
  } else {
    this.tail.next = target.next;
    this.tail = this.tail.next;
  }

  this.size++;
```

```
  return this;
};

MyMap.prototype.get = function (key) {
  const i = this.hash(key);
  let target = this.bucket[i];
  while (target.next) {
    if (target.next.key === key) {
      // 更新
      return target.next.value;
    }
    target = target.next;
  }
};

MyMap.prototype.has = function (key) {
  const i = this.hash(key);
  let target = this.bucket[i];
  while (target.next) {
    if (target.next.key === key) {
      // 更新
      return true;
    }
    target = target.next;
  }

  return false;
};

MyMap.prototype.delete = function (key) {
  const i = this.hash(key);
  let target = this.bucket[i];

  while (target.next) {
    if (target.next.key === key) {
      // 删除 变动链表
      target.next = target.next.next;

      return true;
    }
    target = target.next;
  }

  return false;
};

MyMap.prototype.clear = function () {
  this.init();
};

// MyMap.prototype.entries = function* () {
//   let head = this.head;
//   while (head) {
//     if (head.key) {
//       yield [head.key, head.value];
//     }
//     head = head.next;
//   }
```

```
// };

// MyMap.prototype[Symbol.iterator] = MyMap.prototype.entries; //默认遍历器接口，
for of使用

let m = new MyMap(); //使用构造函数的方式实例化map

m.set(1, 1000);

m.set("x", "0");
m.set("y", "1");
// m.set("x", "00");

m.set({}, "对象");
m.set(null, "null");
m.set(undefined, "undefined");

// console.log("get x", m.get("x"));
// console.log("get x", m.get("y"));
// console.log("get z", m.get("z"));

// console.log("has x", m.has("x")); //sy-log
// console.log("delete x", m.delete("x"));
// console.log("has x", m.has("x")); //sy-log

console.log("map", m.size, m); //sy-log
```

Object与Map的选择（来自红宝书6.4.3）

与Object类型的一个主要差异是，Map实例会维护键值对的插入顺序，因此可以根据插入顺序执行迭代操作。

对于多数Web开发任务来说，选择Object还是Map只是个人偏好问题，影响不大。不过，对于在乎key的类型、插入顺序、以及内存和性能的开发者来说，Object和Mao之间确实存在显著的差别。

Map的大多数特性都可以通过Object类型实现，但二者之间还是存在一些细微的差异。具体实践中使用哪一个，还是值得细细甄别。

内存占用

Object和Map的工程级的实现在不同浏览器间存在明显差异，但存储单个键值对所占用的内存数量都会随着键的数量线性增加。批量添加或者删除键值对则取决于各浏览器对该类型内存分配的工程实现。不同浏览器的情况不同，但给定固定大小的内存，Map大约可以比Object多存储50%的键值对。

插入性能

向Object和Map中插入新键值对的消耗大致相当，不过插入Map在所有浏览器中一般会稍快一点。对这两个类型来说，插入速度并不会随着键值对数量而线性增加。如果代码涉及大量插入操作，那么显然Map的性能更佳。

查找速度

如果只包含少量键值对，则Object有时候速度更快。在把Object当成数组使用的情况下（比如使用连续整数作为属性），浏览器引擎可以进行优化，在内存中使用更高效的布局。这对Map来说是不可能的。对这两个类型而言，查找速度不会随着键值对数量增加而线性增加。如果代码涉及大量查找操作，那么在某些情况下可能选择Object更好一些。

删除性能

使用delete删除Object属性的性能一直以来饱受诟病，目前在很多浏览器中仍然如此。为此，出现了一些伪删除对象属性的操作，包括把属性值设置为undefined或者null。但很多时候，这都是一种讨厌的或不合时宜的折中。而对大多数浏览器引擎来说，Map的delete操作都比插入和查找更快。如果代码涉及大量删除操作，那么毫无疑问应该选择Map。

Mini-React（迷你版React18）

1. 实现函数组件、类组件、原生标签、文本等的渲染与更新
2. 实现全部Hooks API
3. 使用ReactDOM.createRoot实现初次渲染
4. 实现Concurrent模式
5. 实现fiber
6. 实现任务调度与最小堆算法