

用JS实现基础数据结构(栈、队列、单双链表、二叉搜索树)

Stack

```
function Stack() {
    var items = [];

    this.push = function (element) {
        items.push(element);
    };

    this.pop = function () {
        return items.pop();
    };

    this.peek = function () {
        return items[items.length - 1];
    };

    this.isEmpty = function () {
        return items.length === 0;
    };

    this.size = function () {
        return items.length;
    };

    this.clear = function () {
        items = [];
    };

    this.print = function () {
        console.log(items.toString());
    };
}
```

Queue

```
function Queue() {
    var items = [];

    this.enqueue = function (element) {
        items.push(element);
    };

    /**
     * 移除队列第一项
     */
    this.dequeue = function () {
```

```

        return items.shift();
    };

    /**
     * 返回队列中第一个元素
     */
    this.front = function () {
        return items[0];
    };

    this.isEmpty = function () {
        return items.length === 0;
    };

    this.clear = function () {
        items = [];
    };

    this.size = function () {
        return items.length;
    };

    this.print = function () {
        console.log(items.toString());
    };
}

```

优先队列

```

function PriorityQueue() {
    var items = [];

    /**
     * 优先队列元素带有priority
     * @param element any
     * @param priority number
     */
    function QueueElement(element, priority) {
        this.element = element;
        this.priority = priority;
    }

    this.enqueue = function (element, priority) {
        var queueElement = new QueueElement(element, priority);

        if (this.isEmpty()) {
            items.push(queueElement);
        } else {
            var added = false;
            // 如果找到优先级更高的元素，在当前位置插入即可
            for (var i = 0; i < items.length; i++) {
                if (priority < items[i].priority) {
                    items.splice(i, 0, queueElement);
                    added = true;
                }
            }
            if (!added) {
                items.push(queueElement);
            }
        }
    };

    this.dequeue = function () {
        return items.shift();
    };

    this.front = function () {
        return items[0];
    };

    this.isEmpty = function () {
        return items.length === 0;
    };

    this.clear = function () {
        items = [];
    };

    this.size = function () {
        return items.length;
    };

    this.print = function () {
        console.log(items.toString());
    };
}

```

```

        // 找到一个就可以终止循环了
        break;
    }

    // 没找到优先级更高的，直接往队列最后推
    if (!added) {
        items.push(queueElement);
    }
}

};

this.dequeue = function () {
    return items.shift();
};

this.front = function () {
    return items[0];
};

this.isEmpty = function () {
    return items.length === 0;
};

this.clear = function () {
    items = [];
};

this.size = function () {
    return items.length;
};

this.print = function () {
    console.log(items.toString());
};
}

```

单链表

```

function SingleLinkedList() {

    var Node = function (element) {
        this.element = element;
        this.next = null;
    };

    var length = 0;
    var head = null;

    /**
     * @param element any
     */
    this.append = function (element) {
        var node = new Node(element);

```

```
    var current;

    if (head === null) {
        head = node;
    } else {
        current = head;

        // 从头部元素开始往后查询
        while (current.next) {
            current = current.next;
        }

        // 在最后一个阶段后插入
        current.next = node;
    }

    length++;
};

/** 
 * @param position number
 * @param element any
 * @returns {boolean}
 */
this.insert = function (position, element) {
    if (position < 0 || position > length) {
        return false;
    }

    var node = new Node(element);
    var current = head;
    var previous;
    var index = 0;

    if (position === 0) {
        node.next = current;
        head = node;
    } else {
        while (index++ < position) {
            previous = current;
            current = current.next;
        }
        node.next = current;
        previous.next = node;
    }

    length++;
    return true;
};

/** 
 * @param position number
 * @returns {null|*}
 */
this.removeAt = function (position) {
```

```
    if (position < 0 || position > length) {
        return null;
    }

    var current = head;
    var previous;
    var index = 0;

    if (position === 0) {
        head = current.next;
    } else {
        while (index++ < position) {
            previous = current;
            current = current.next;
        }
    }

    // 跳过current, 连接前后两个节点
    previous.next = current.next;
}

length--;
return current.element;
};

this.remove = function (element) {
    var index = this.indexOf(element);
    return this.removeAt(index);
};

this.indexOf = function (element) {
    var current = head;
    var index = 0;
    while (current) {
        if (element === current.element) {
            return index;
        }
        index++;
        current = current.next;
    }
    return -1;
};

this.isEmpty = function () {
    return length === 0;
};

this.size = function () {
    return length;
};

this.getHead = function () {
    return head;
};

this.toString = function () {
    var current = head;
    var string = '';

    while (current) {
        string += ',' + current.element;
        current = current.next;
    }
    return string;
};
```

```

    }

    return string.slice(1);
};

this.print = function () {
    console.log(this.toString());
};

}

```

双链表

```

function DoubleLinkedList() {
    var Node = function (element) {
        this.element = element;
        this.next = null;
        this.prev = null;
    };

    var length = 0;
    var head = null;
    var tail = null;

    /**
     * @param element any
     */
    this.append = function (element) {
        var node = new Node(element);
        var current;

        if (head === null) {
            head = node;
        } else {
            current = head;

            // 从头部元素开始往后查询
            while (current.next) {
                current = current.next;
            }

            // 在最后一个阶段后插入
            current.next = node;
        }

        length++;
    };

    /**
     *
     * @param position number
     * @param element any
     * @returns {boolean}
     */
    this.insert = function (position, element) {
        if (position < 0 || position > length) {

```

```

        return false;
    }

    var node = new Node(element);
    var current = head;
    var previous;
    var index = 0;

    if (position === 0) {
        if (!head) {
            head = node;
            tail = node;
        } else {
            node.next = current;
            current.prev = node;
            head = node;
        }
    } else if (position === length) {
        current = tail;
        current.next = node;
        node.prev = current;
    } else {
        while (index++ < position) {
            previous = current;
            current = current.next;
        }
        node.next = current;
        previous.next = node;

        current.prev = node;
        node.prev = previous;
    }

    length++;
    return true;
};

/**
 * @param {number} position
 * @returns {null|*}
 */
this.removeAt = function (position) {
    if (position < 0 || position > length) {
        return null;
    }

    var current = head;
    var previous;
    var index = 0;

    if (position === 0) {
        head = current.next;
        if (length === 1) {
            tail = null;
        } else {
            head.prev = null;
        }
    } else if (position === length - 1) {
        previous = current.prev;
        previous.next = null;
        tail = previous;
    } else {
        previous = current.prev;
        current = current.next;
        previous.next = current;
        current.prev = previous;
    }
}
```

```
        }
    } else if (position === length - 1) {
        current = tail;
        tail = current.prev;
        tail.next = null;
    } else {
        while (index++ < position) {
            previous = current;
            current = current.next;
        }

        // 跳过current, 连接前后两个节点
        previous.next = current.next;
        current.next.prev = previous;
    }

    length--;
    return current.element;
};

this.remove = function (element) {
    var index = this.indexOf(element);
    return this.removeAt(index);
};

this.indexOf = function (element) {
    var current = head;
    var index = 0;
    while (current) {
        if (element === current.element) {
            return index;
        }
        index++;
        current = current.next;
    }
    return -1;
};

this.isEmpty = function () {
    return length === 0;
};

this.size = function () {
    return length;
};

this.getHead = function () {
    return head;
};

this.toString = function () {
    var current = head;
    var string = '';

    while (current) {
        string += ',' + current.element;
        current = current.next;
    }

    return string.slice(1);
};
```

```
        this.print = function () {
            console.log(this.toString());
        };
    }
}
```

二叉搜索树

```
function BinarySearchTree() {
    var Node = function (key) {
        this.key = key;
        this.left = null;
        this.right = null;
    };

    var root = null;

    this.insert = function (key) {
        var newNode = new Node(key);

        if (root === null) {
            root = newNode;
        } else {
            insertNode(root, newNode);
        }
    };

    var insertNode = function (node, newNode) {
        if (newNode.key < node.key) {
            if (node.left === null) {
                node.left = newNode;
            } else {
                insertNode(node.left, newNode);
            }
        } else {
            if (node.right === null) {
                node.right = newNode;
            } else {
                insertNode(node.right, newNode);
            }
        }
    };
}
```

用JS实现基础排序和搜索算法



一、初始化

```

function ArrayList() {
    let array = [];

    this.insert = (item) => {
        array.push(item);
    };

    this.toString = () => {
        return array.join();
    };

    // 先定义一个内部swap方法 用于交换数组中的两个值
    // 注意，只能用在ArrayList内部
    const swap = function (index1, index2) {
        const aux = array[index1];
        array[index1] = array[index2];
        array[index2] = aux;
    };
}

```

二、排序

1、冒泡算法

比较任意两个相邻的项，如果第一个比第二个大，则交换顺序

```

var bubbleSort = function () {
    const len = array.length;
    for (let i = 0; i < len; i++) {
        for (let j = 0; j < len - 1; j++) {
            if (array[j] > array[j + 1]) {
                swap(j, j + 1);
            }
        }
    }
};

```

改进一下，从内循环中减去外循环中已跑过的轮数

```
var modifiedBubbleSort = function () {
    const len = array.length;
    for (let i = 0; i < len; i++) {
        // 看这里 j < len - 1 - i
        for (let j = 0; j < len - 1 - i; j++) {
            if (array[j] > array[j + 1]) {
                swap(j, j + 1);
            }
        }
    }
};
```

2、选择排序

找到数组中最小的项并将其放到第一位，找到第二小的值，并将其放到第二位，依次.....

```
var selectionSort = function () {
    const len = array.length;
    let indexMin; // 最小值下标
    for (let i = 0; i < len - 1; i++) {
        // 假设当前遍历的是最小值
        indexMin = i;
        // 前面已经拍过序的不用再循环了
        for (let j = i; j < len; j++) {
            // 依次比较，交换最小值下标
            if (array[indexMin] > array[j]) {
                indexMin = j;
            }
        }
        // 如果找到的最小值跟原来设定的最小值不一样，交换其值
        if (i !== indexMin) {
            swap(i, indexMin);
        }
    }
};
```

3、插入排序

每次只排序一个数组项，确定它应该插入到哪个位置

```
var insertionSort = function () {
    let j, temp;
    // 默认第一项已经排序，所以从第二项开始
    for (let i = 1; i < a.length; i++) {
        // 辅助变量和值，存储当前下标和值
        j = i;
        temp = a[i];
        // 一直跟前一项比较，直到找到正确的位置插入
        while (j > 0 && a[j - 1] > temp) {
            // 移到当前位置
            a[j] = a[j - 1];
            j--;
        }
    }
};
```

```

        }
        a[j] = temp;
    }
};
```

4、归并排序（分治）

- 将数组拆分成较小的数组，直到每个数组的长度为1；
- 合并和排序小数组，直到回到原始数组的长度；

```

var mergeSort = function () {
    // 递归的停止条件
    if (array.length == 1) {
        return array;
    }
    // 中间值取整，分成两个小组
    var middle = Math.floor(array.length / 2),
        left = array.slice(0, middle),
        right = array.slice(middle);
    // 递归，对左右两部分数据进行合并排序
    return merge(mergeSort(left), mergeSort(right));
};

function merge(left, right) {
    var result = [];
    while (left.length > 0 && right.length > 0) {
        // 比较左边的数组的值是否被右边的小
        if (left[0] < right[0]) {
            result.push(left.shift());
        } else {
            result.push(right.shift());
        }
    }
    return result.concat(left).concat(right);
}
```

5、快速排序（分治）

- 从数组中选择中间项目作为主元
- 建立左右两个数组，分别存储左边和右边的数组
- 利用递归进行下次比较

```

var quickSort = function () {
    if (array.length <= 1) {
        return array;
    }
    // 取中间数作为基准索引，浮点数向下取整
    var index = Math.floor(array.length / 2);
    // 取得该值
    var pivot = array.splice(index, 1);
    // 分别建立左右空数组，作为push所用
    var left = [];
    var right = [];
    for (var i = 0; i < array.length; i++) {
```

```

    // 基准左边的传到左数组,右边的传到右数组
    if (array[i] < pivot) {
        left.push(array[i]);
    } else {
        right.push(array[i]);
    }
}
// 不断递归重复比较
return quickSort(left).concat(pivot, quickSort(right));
};

```

三、搜索

1、顺序搜索

将每一个数据结构中的元素和我们要找的元素做比较
最低效的一种搜索算法

```

var sequentialSearch = function (item) {
    for (let i = 0; i < array.length; i++) {
        if (item === array[i]) {
            return i;
        }
    }
    return -1;
};

```

2、二分搜索

- 选择数组的中间值
- 如果选中值是待搜索值，那么算法执行完毕（值找到了）。
- 如果待搜索值比选中值要小，则返回步骤1并在选中值左边的子数组中寻找
- 如果待搜索值比选中值要大，则返回步骤1并在选中值右边的子数组中寻找

```

var binarySearch = function (item) {
    // 先将数组进行排序
    this.quickSort();

    let low = 0, hight = array.length - 1, mid, element;

    while (low <= hight) {
        // 取中间值
        mid = Math.floor((low + hight) / 2);
        element = array[mid];
        if (element < item) {
            low = mid + 1;
        } else if (element > item) {
            hight = mid - 1;
        } else {
            return mid;
        }
    }
};

```

```
    return -1;
};
```

转Vue 3前再读一次Vue2源码

Vue3如火如荼，再不读估计以后就再也不会读了

找到vue代码入口

首先，从 [GitHub - vuejs/vue](#) 下载源码

直接看 package.json 文件， main 和 module 字段代表都是已经构建好的最终代码

那就从npm scripts看起

```
dev -> rollup -w -c scripts/config.js --environment TARGET:web-full-dev ->
scripts/config.js -> web-full-dev 命令对应的文件 src/platforms/web/entry-runtime-with-
compiler.js -> 一路向下找到入口文件 src/core/instance/index.js
```

```
function Vue (options) {
  if (process.env.NODE_ENV !== 'production' &&
  !(this instanceof Vue)
) {
  warn('Vue is a constructor and should be called with the `new` keyword')
}
this._init(options)
}

initMixin(Vue)
stateMixin(Vue)
eventsMixin(Vue)
lifecycleMixin(Vue)
renderMixin(Vue)

export default Vue
```

调试vue源码

在开始读源码之前，先把调试源码的工作准备好

首先，在 dev 命令中加一个配置项 --sourcemap，完整命令如下

```
"dev": "rollup -w -c scripts/config.js --sourcemap --environment TARGET:web-
full-dev",
```

然后 npm run dev 启动， rollup会在dist目录下生成一份带有映射关系的vue.js文件，并且会监听更改



利用这份 dist/vue.js 进行调试就可以了

在 examples 目录下建一份html文件，然后引入上面的 dist/vue.js 文件

The screenshot shows the WebStorm IDE interface. On the left, the Project tool window displays a file tree for a 'vue' project, including 'core/index.js', 'global-api/index.js', 'instance/index.js', 'index.html', 'app.js', and 'demo' (which contains 'app.js' and 'index.html'). A red box highlights the 'index.html' file. In the center, the Editor pane shows the content of 'index.html':

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Vue Demo</title>
    <script src="../../dist/vue.js"></script>
</head>
<body>
    <div id="demo">
        <template>
            <span>{{text}}</span>
        </template>
    </div>
    <script src="./app.js"></script>
</body>
</html>
```

A red box highlights the line '`<script src="../../dist/vue.js"></script>`' with the annotation '这里引入了源码' (Here the source code is introduced). In the bottom right corner of the code editor, there is a browser icon.

On the right, the 'app.js' file is open in another editor pane:

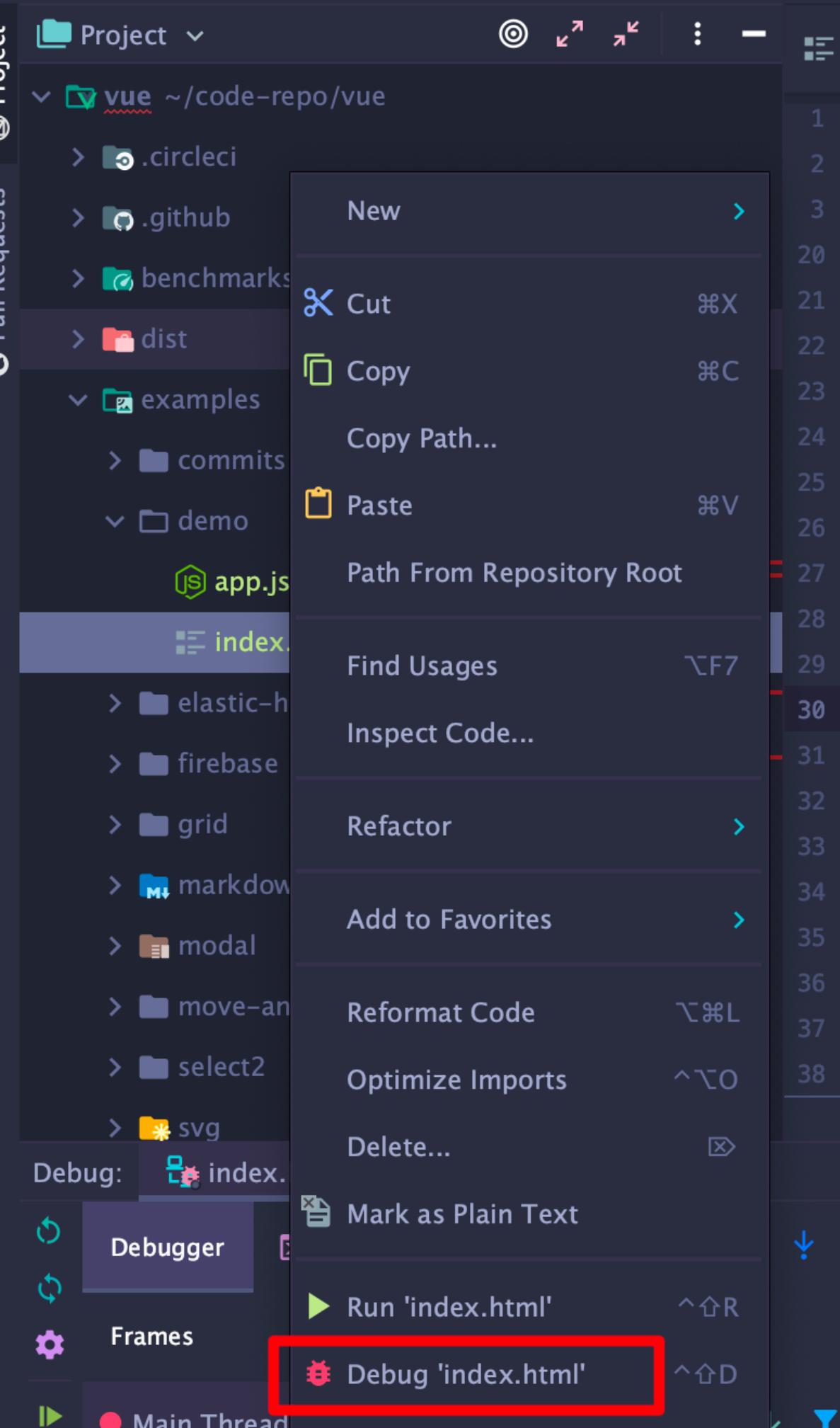
```
var demo = new Vue({
    el: '#demo',
    data() {
        return {
            text: 'hello world!'
        };
    },
    console.dir(Vue);
});
```

A red box highlights the line '`var demo = new Vue({`' with the annotation '自己再写一份js文件，方便写其他逻辑' (Write your own JS file to facilitate writing other logic).

WebStorm 提供了一站式 debug，非常方便

如下直接右键 debug html文件，在源码需要的地方打断点即可开始

vscode也提供了类似的功能，自行摸索一下



Main Thread More Run/Debug

Project: vue ~/code-repo/vue

File: global-api/index.js

```
/* @flow */
import ...
export function initGlobalAPI (Vue: GlobalAPI) {
  // config
  const configDef = {}
  configDef.get = () => config
  if (process.env.NODE_ENV !== 'production') {
    configDef.set = () => {
      warn(
        'Do not replace the Vue.config object, set individual fields instead.'
      )
    }
  }
  Object.defineProperty(Vue, p: 'config', configDef)
  configDef: undefined
  Vue: function Vue (options) {
    // exposed util methods.
    initGlobalAPI() > set()
  }
}
```

Debug: index.html

Debugger

Frames

Main Thread

initGlobalAPI(), index.js:23

Vue.set = undefined

Vue = function Vue (options) {

- length = 1
- name = "Vue"

Object.prototype = Object { _init: Function, \$set: Function, \$delete: Function, \$watch: Function, \$on: Function, ... }

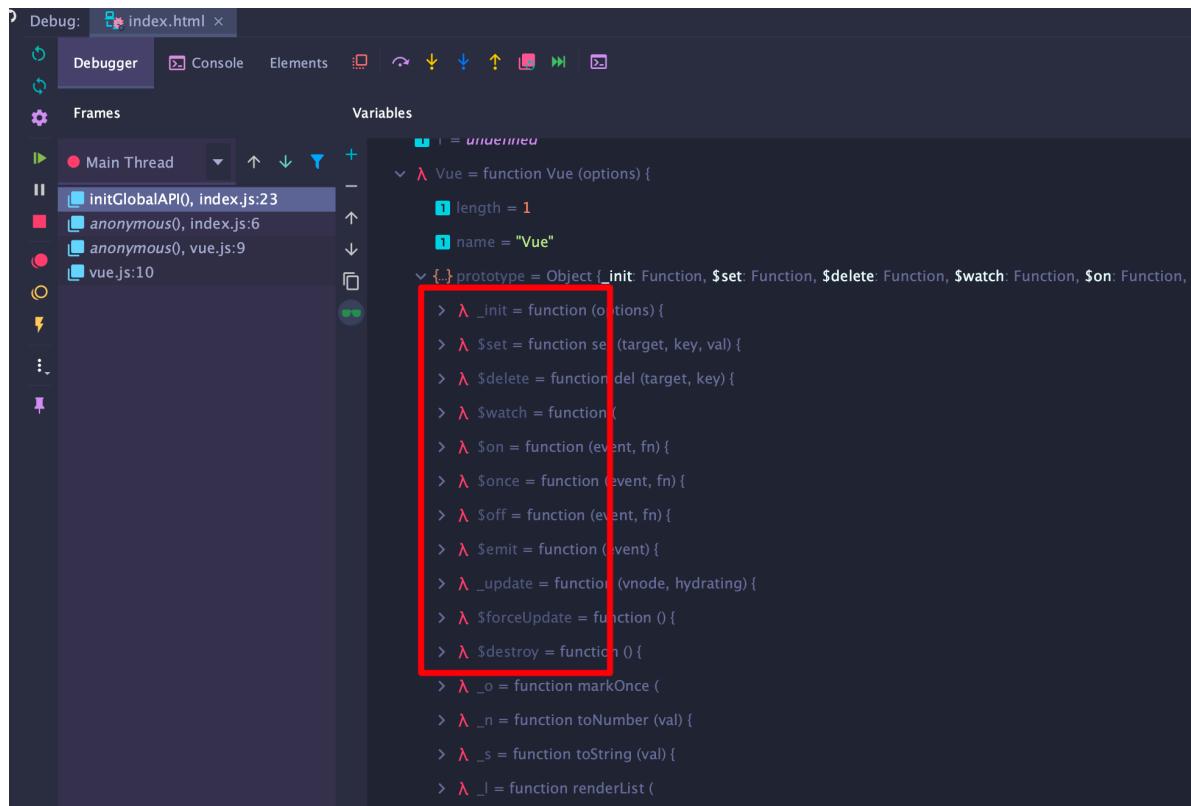
- _init = function (options) {
- \$set = function set (target, key, val) {
- \$delete = function del (target, key) {
- \$watch = function (
- \$on = function (event, fn) {
- \$once = function (event, fn) {
- \$off = function (event, fn) {
- \$emit = function (event) {
- _update = function (vnode, hydrating) {
- \$forceUpdate = function () {

Vue对象初始化

Vue原型挂载

通过这五个方法，可以看出 `instance/index.js` 在 `Vue.prototype` 上挂载了一些方法

- `initMixin(Vue)`
- `stateMixin(Vue)`
- `eventsMixin(Vue)`
- `lifecycleMixin(Vue)`
- `renderMixin(Vue)`



Vue属性挂载

回到 core/index.js

```
import Vue from './instance/index'
import { initGlobalAPI } from './global-api/index'
import { isServerRendering } from 'core/util/env'
import { FunctionalRenderContext } from 'core/vdom/create-functional-component'

initGlobalAPI(Vue)

Object.defineProperty(Vue.prototype, '$isServer', {
  get: isServerRendering
})

Object.defineProperty(Vue.prototype, '$ssrContext', {
  get () {
    /* istanbul ignore next */
    return this.$vnode && this.$vnode.ssrContext
  }
})

// expose FunctionalRenderContext for ssr runtime helper installation
Object.defineProperty(Vue, 'FunctionalRenderContext', {
  value: FunctionalRenderContext
})

Vue.version = '__VERSION__'

export default Vue
```

主要通过 `initGlobalAPI(Vue)` 在Vue对象上添加了一波属性，主要是一些静态属性和方法

```

9   import {
10     query,
11     mustUseProp,
12     isReservedTag,
13     isReservedAttr,
14     getTagNamespace,
15     isUnknownElement
16   } from 'web/util/index'
17
18   import { patch } from './patch'
19   import platformDirectives from './directives/index'
20   import platformComponents from './components/index'
21
22   // install platform specific utils
23   Vue.config.mustUseProp = mustUseProp
24
25   Vue
26   Vue

```

The screenshot shows the `runtime/index.js` file in a code editor. The `Vue` class definition is highlighted. A red box surrounds the constructor function `Vue = function Vue (options) {` and its initial properties and methods.

```

Vue = function Vue (options) {
  > ... util = Object {warn: Function, extend: Function, mergeOptions: Function}
  > ... set = function set (target, key, val) {
  > ... delete = function delete (target, key) {
  > ... nextTick = function nextTick (cb, ctx) {
  > ... observable = function (obj) {
  > ... options = Object {components: Object, directives: Object, filters: Object}
  > ... use = function (plugin) {
  > ... mixin = function (mixin) {
    > cid = 0
  > ... extend = function (extendOptions) {
  > ... component = function (
  > ... directive = function (
  > ... filter = function
    > version = "2.6.14"
    > length = 1
    > name = "Vue"
  > ... prototype = Object {init: Function, $set: Function, $delete: Function,
    config = ... (invoke getter)
  > ... getConfig = function () { return config; }
  > ... setConfig = function () {

```

Vue完整初始化

再往下走，回到 `src/platforms/web/runtime/index.js`，主要是对web平台添加一下配置、组件、指令（只看大方向，不看具体内部实现和逻辑）

再往外走，到 `src/platforms/web/entry-runtime-with-compiler.js`

- 覆盖 `$mount` 方法
- 挂载 `compile` 方法，提供了编译 `template` 的能力（完整版和运行时版本的区别）

做个小结

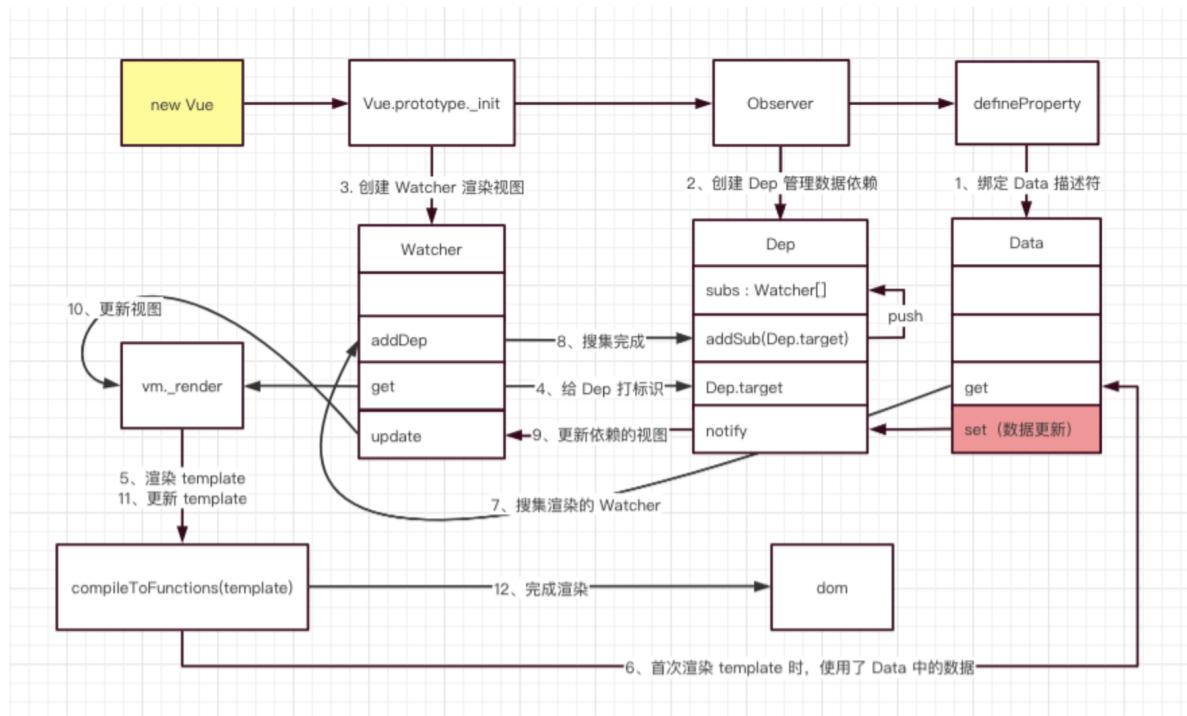
- `instance/index.js` 对 `vue.prototype` 进行属性和方法挂载
- `core/index.js` 对 `vue` 进行属性和方法挂载
- `runtime/index.js` 对不同platform，进行配置、组件、指令的差异化挂载
- `entry-runtime-with-compiler.js` 为 `$mount` 方法增加 `compile` 能力

再回 `src/core/runtime/index.js` 的 `this._init(options)`，一切从这里开始 ->
`Vue.prototype._init` -> 经过一系列的初始化以及合并配置工作（此处省略一大堆）-> 从
`src/core/instance/lifecycle.js` 的 `Vue.prototype.$mount` -> `mountComponent`

响应式系统

之前写的 [Vue 响应式原理核心 · Issue #63 · amandakelake/blog · GitHub](#)

本质上就是对 [Object.defineProperty\(\) - JavaScript | MDN](#) 的理解和运用，再加上 `watcher`、`Dep` 组合的发布订阅模式，组成 vue 的核心原理



图片转载于 [图解 Vue 响应式原理 - 掘金](#)

核心流程

- `new vue` 开始初始化，通过 `Object.defineProperty` 监听 `data` 里面的数据变化，创建 `Observer` 并遍历 `data` 创建 `Dep` 来收集使用当前 `data` 的 `watcher`，每个 `key` 都会 `new` 一个 `Dep`
- 编译模板时会每个组件都会创建一个 `watcher`，同时会将 `Dep.target` 标识为当前 `watcher`
- 编译模板/`mountComponent` 时，如果使用到了数据，会触发 `data.get` -> `Dep.addSub` 将相关的 `watcher` 收集到 `Dep.subs` 中
- 数据更新触发 `data.set` -> `Dep.notify` -> 通知相关的 `watcher` 调用 `vm._render` 更新DOM
- 触发 `watcher` 的 `update` 过程，利用队列做了优化，在 `nextTick` 后执行所有 `watcher` 的 `run` 方法，最后再执行它们的回调函数

Dep.target = watcher 的概念有点绕，可以多看几遍这里

Vue源码详细解析(一)——数据的响应化 · Issue #1 · Ma63d/vue-analysis · GitHub

我们来说说这个Dep，Dep类的定义极其简单，一个id，一个数组，他就是一个很基本的发布者-观察者模式的实现，作为一个发布者，他的subs属性用来存放了订阅他的观察者，也就是后面我们会说到的watcher。

defineProperty是用来将对象的属性转化为响应式的getter/setter的，defineProperty函数执行过程中新建了一个Dep，闭包在了属性的getter和setter中，因此每个属性都有一个唯一的Dep与其对应，我们暂且可以把属性和他对应的Dep理解为一体的。

Dep其实是dependence依赖的缩写，我之前一直没能理解依赖、依赖收集是什么，其实对于我们的一个模板{{a+b}}，我们会说他的依赖有a和b，其实就是依赖了data的a和b属性，更精确的说是依赖了a属性中闭包的dep实例和b属性中闭包的那个dep实例。

详细来说：我们的这个{{a+b}}在dom里最终会被“a+b”表达式的真实值所取代，所以存在一个求出这个“a+b”的表达式的过程，求值的过程就会自然的分别触发a和b的getter，而在getter中，我们看到执行了dep.depend()，这个函数实际上回做 dep.addSub(Dep.target)，即在dep的订阅者数组中存放了Dep.target，让Dep.target订阅dep。

那Dep.target是什么？他就是我们后面介绍的Watcher实例，为什么要放在Dep.target里呢？是因为getter函数并不能传参，dep可以通过闭包的形式放进去，那watcher可就不行了，watcher内部存放了a+b这个表达式，也是由watcher计算a+b的值，在计算前他会把自己放在一个公开的地方（Dep.target），然后计算a+b，从而触发表达式中所有遇到的依赖的getter，这些getter执行过程中会把Dep.target加到自己的订阅列表中。等整个表达式计算成功，Dep.target又恢复为null。这样就成功的让watcher分发到了对应的依赖的订阅者列表中，订阅到了自己的所有依赖。

我们可以看到这是极其精妙的一笔！在一个表达式的求值过程中隐式的完成依赖订阅。

上面完成的是订阅的过程，而上面setter代码里的 dep.notify 就负责完成数据变动时通知订阅者的功能。而且数据变化时，后文会说明只有依赖他的那些dom会精确更新，不会出现一些介绍mvvm的文章里虽然实现了订阅更新但是重新计算整个视图的情况。

于是一整个对象订阅、notify的过程就结束了。

下面是一份简单的响应式系统的实现

```
class Observer {
    constructor(data) {
        this.walk(data);
    }

    walk(data) {
        // 此处简化，只处理对象
        if (data instanceof Object) {
            for (let key in data) {
                if (data.hasOwnProperty(key)) {
                    this.defineReactive(data, key, data[key]);
                }
            }
        }
    }

    /**
     * getter中收集依赖，setter中触发依赖
     */
    defineReactive(data, key, val) {
        const _this = this;
        // 每个key实例一个dep
        const dep = new Dep();
        Object.defineProperty(data, key, {
            enumerable: true,
            configurable: true,
            get: function () {
                // 将当前的watcher实例收集到依赖中
                Dep.target && dep.addSub(Dep.target);
                return val;
            },
        });
    }
}
```

```

        set: function (newVal) {
            if (val === newVal) {
                return;
            }
            val = newVal;
            // 递归遍历新值
            _this.walk(newVal);
            // 触发依赖
            dep.notify();
        },
    );
}
}

class Dep {
    static target;

    constructor() {
        this.subs = [];
    }

    addSub(sub) {
        if (sub && sub.update) {
            this.subs.push(sub);
        }
    }

    notify() {
        this.subs.forEach(sub => sub.update());
    }
}

class Watcher {
    constructor(data, key, callback) {
        // getter收集依赖, getter不能传参, 所以通过闭包传进去
        Dep.target = this;
        this.data = data;
        this.key = key;
        this.callback = callback;
        this.value = data[key];
        // 完成某个属性的依赖收集后,清空Dep.target
        // notify方法会重新调用getter(重新获取值, 重新收集依赖)
        // 清空Dep.target, 防止notify中不停绑定Watcher与Dep -> 代码死循环
        Dep.target = null;
    }

    /**
     * 更新视图 vm._render / 执行user逻辑
     */
    update() {
        this.value = this.data[this.key];
        this.callback(this.value);
    }
}

```

```
const data = {
  a: 1,
  b: 1,
};

new Observer(data);

new Watcher(data, 'a', (value) => {
  console.log('watcher update 新值 -> ' + value);
});

data.a = 2;
data.a = {
  c: 'c',
};

new Watcher(data.a, 'c', (value) => {
  console.log('watcher update 新值 -> ' + value);
});

data.a.c = 'hello new world';
```

上面代码直接跑的结果如图

```
watcher update 新值 -> 2
watcher update 新值 -> [object Object]
watcher update 新值 -> hello new world
```

[依赖收集 | Vue.js 技术揭秘](#) 黄老的课永远可以信赖

- computed的本质是 `computed watcher`
- watch的本质是 `user watcher`

Patch和Diff原理

DOM操作是昂贵的，尽量减少DOM操作

找出必须更新的节点，非必要不更新

数据发生变化时，会触发渲染 `watcher` 的回调函数，执行组件更新

```
// src/core/instance/lifecycle.js

updateComponent = () => {
  vm._update(vm._render(), hydrating)
}

new watcher(vm, updateComponent, noop, {
  before () {
    if (vm._isMounted && !vm._isDestroyed) {
      callHook(vm, 'beforeUpdate')
    }
  }
}, true /* isRenderwatcher */)
```

组件更新调用了 `vm._update`

```
// src/core/instance/lifecycle.js
Vue.prototype._update = function (vnode: VNode, hydrating?: boolean) {
  const vm: Component = this
  const prevEl = vm.$el
  const prevVnode = vm._vnode
  vm._vnode = vnode
  if (!prevVnode) {
    // initial render
    vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly */)
  } else {
    // updates
    vm.$el = vm.__patch__(prevVnode, vnode)
  }
  // ...
}
```

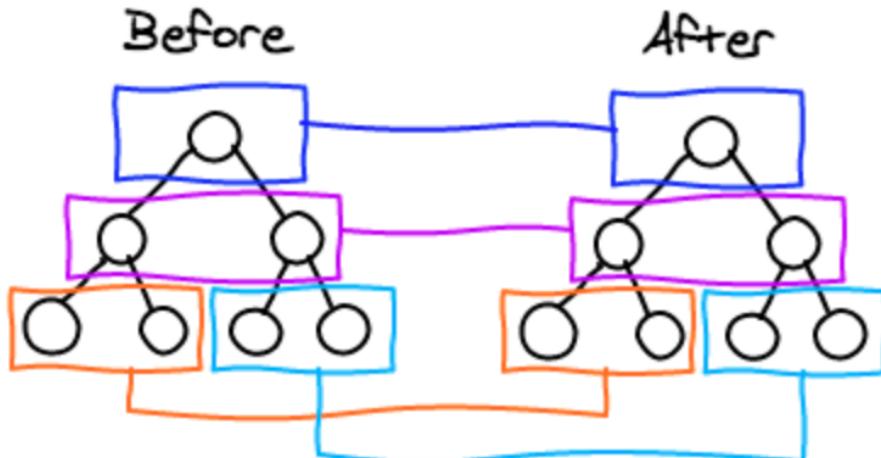
深入 patch 代码细节前，先记住vue的diff算法特点： 同层比较、不会跨层级、不同节点直接删除

vue和react的diff算法大同小异，网上大部分文章都来源于这篇2013年的

[React's diff algorithm](#)

、[中文版本](#)

包括这张传遍大街小巷的图



执行 `vm.__patch__` 的逻辑在 `src/core/vdom/patch.js` 中，`patch` 主要处理新旧节点是否相同

- 新旧节点不同：创建新节点 -> 更新父占位符节点 -> 删除旧节点
- 新旧节点相同：获取 `children`, `diff child` 做不同更新逻辑 -> 比较核心的 `updateChildren` 方法

先看 `patch` 的核心逻辑，以下代码对源码做了精简和注释，方便理解

```
function patch(oldVnode, vnode) {  
  if (!oldVnode) {  
    // 空 mount (likely as component), create new root element  
    createElm(vnode);  
  } else {  
    if (sameVnode(oldVnode, vnode)) {  
      // 二、新旧节点相同, diff children  
      // patch existing root node  
      patchVnode(oldVnode, vnode);  
    } else {  
      // 一、新旧节点不同的情况  
  
      const oldElm = oldVnode.elm;  
      const parentElm = nodeOps.parentNode(oldElm);  
  
      // 1、创建新节点  
      createElm(vnode);  
  
      // 2、更新父的占位符节点  
      if (parentElm) {  
        // 更新逻辑  
      }  
  
      // 3、删除旧节点  
      removeVnodes(oldVnode);  
    }  
  }  
  
  // 返回vnode, 此时vnode.el已经对应了真实的dom  
  return vnode;  
}
```

以上可以看出，打补丁做的事情就算给 `vnode.el` 对应到真实的dom上面

看下辅助方法 `sameVnode`，它的目的就是判断两个节点是否值得比较

比较逻辑很简单，就是先看 `key`，`key` 不同则是不同组件，再开始判断 `tag`、`isComment`、`data`、`input` 等类型

```
function sameVnode (a, b) {  
  return (  
    a.key === b.key &&  
    a.asyncFactory === b.asyncFactory && (  
      (  
        a.tag === b.tag &&  
        a.isComment === b.isComment &&  
        isDef(a.data) === isDef(b.data) &&
```

```

        sameInputType(a, b)
    ) || (
        isTrue(a.isAsyncPlaceholder) &&
        isUndef(b.asyncFactory.error)
    )
)
}

```

只有当两个节点值得比较时，才会进入 `patchVnode(oldVnode, vnode)` 的流程

```

function patchVnode (oldVnode, vnode) {

    const elm = vnode.elm = oldVnode.elm

    // 如果新旧节点相同, return
    if (oldVnode === vnode) {
        return
    }

    // 如果新旧节点都是文本节点, return
    if (vnode.isStatic && oldVnode.isStatic && vnode.key === oldVnode.key) {
        vnode.componentInstance = oldVnode.componentInstance
        return
    }

    // 1、prepatch -> updateChildComponent, 更新vnode对应实例的属性, 如$ vnode、slot、
    // listeners、props等
    prepatch(oldVnode, vnode)

    // 2、执行update钩子以及用户自定义的update方法
    callUpdateHook(vnode)

    // 3、patch
    const oldCh = oldVnode.children
    const ch = vnode.children

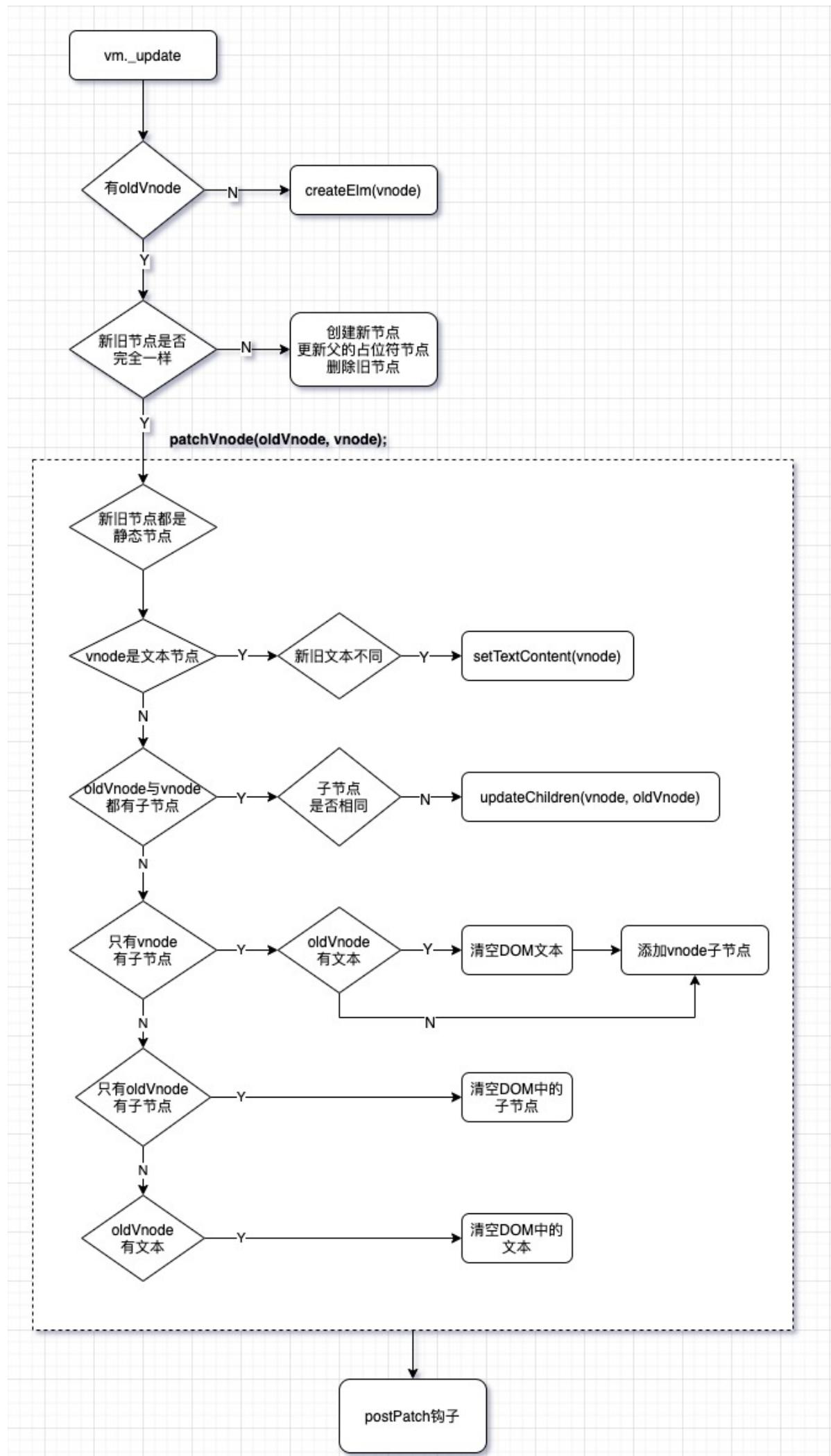
    // 不是文本节点
    if (!vnode.text) {
        if (oldCh && ch) {
            if (oldCh !== ch) {
                updateChildren(elm, oldCh, ch)
            }
        } else if (ch) {
            if (oldVnode.text) {
                nodeOps.setTextContent(elm, '')
            }
            addVnodes(elm, ch)
        } else if (oldCh) {
            removeVnodes(oldCh)
        } else if (oldVnode.text) {
            nodeOps.setTextContent(elm, '')
        }
    }
    // 是文本节点, 且新旧文本不同

```

```
    } else if (oldVnode.text !== vnode.text) {
        nodeOps.setTextContent(elm, vnode.text)
    }

    postpatch(oldVnode, vnode)
}
```

以上伪代码都不难读，配合下面流程图会更清晰



还剩下一个 `updateChildren(vnode, oldVnode)` 流程，既是diff里的重点也是难点，该方法的核心规律是通过while进行遍历收缩循环

- 从头尾开始移动，当交叉则停止
 - `oldStartIdx -> newStartIdx`
 - `oldEndIdx -> newEndIdx`
 - `oldStartIdx -> newEndIdx`
 - `oldEndIdx -> newStartIdx`
- 如果以上情况都不符合，则通过key进行判断

文字读起来比较繁杂，最快的方式是看一下前人的视频或者动画，再来理解会事半功倍，推荐[diff算法之理解updateChildren函数](#)

其他

NextTick

```
for (macroTask of macroTaskQueue) {  
    // 1. Handle current MACRO-TASK  
    handleMacroTask();  
  
    // 2. Handle all MICRO-TASK  
    for (microTask of microTaskQueue) {  
        handleMicroTask(microTask);  
    }  
}
```

在浏览器环境中

- 常见的 macro task 有 `setTimeout`、`MessageChannel`、`postMessage`、`setImmediate`；
- 常见的 micro task 有 `MutationObserver` 和 `Promise.then`。

nextTick的降级策略，先微任务，再降级到宏任务

- promise
- MutationObserver
- setImmediate
- setTimeout(0)

*

[Vue源码详解之nextTick：MutationObserver只是浮云，microtask才是核心！ · Issue #6 · Ma63d/vue-analysis · GitHub](#)

@jin5354

第二点有问题，用户修改 reactive 数据不会立即触发 DOM 更新，而只是被所有订阅这个 reactive 数据的 watcher 监听到，然后稍后进行批处理，把这些监听到数据变动的 watcher 的真实数据写进 DOM。

也就是你改数据后，先不改 DOM，过一会，等待当前的 microtask 完成之后再去批处理执行所有改 DOM 的操作。

Vue 提供这个操作并不是为了你说的去『观测 DOM 更新』。DOM modification 是实时的，DOM modification 是实时的，DOM modification 是实时的，重要的话说三遍。虽然我其实在文章里多次强调这句话。DOM modification 是实时的、同步的，是你在上一行代码修改 DOM，DOM tree 就立即、实时、直接、同步的修改完成了的。不需要任何回调去进行观测。

Vue 采用先不修改 DOM，累计一段时间后再去修改 DOM 的原因我在文章中其实详细说了：

这样做了之后即使你在 task 里改了一个 watcher 的依赖100次，我最终只会计算一次 value、改 DOM 一次。一方面省去了不必要的 DOM 修改，另一方面将 DOM 操作聚集，可以提升 DOM Render 效率。

另外，我在博客上写过一篇长文深入说明过浏览器渲染和 eventloop 的关系，也说过 rAF 的东西，可是目前我的博客 chuckliu.me 正在备案无法访问，等可以访问了之后我@你吧。

Vue响应式对数组的处理

为什么vue2中监听不到数组的变化？

并不是因为 `Object.defineProperty` 的问题，它本身对数组的表现跟对象是一致的，数组的索引就可以看做 key 来使用，它本身有监控数组下标变化的能力

其实是 vue2 中放弃了这个特性，在 `observer` 中不会对数组进行 `walk` 处理去遍历所有属性，而是进行了特殊处理

```
export class Observer {
  value: any;
  dep: Dep;
  vmCount: number; // number of vms that have this object as root $data

  constructor (value: any) {
    this.value = value
    this.dep = new Dep()
    this.vmCount = 0
    def(value, key: 'ob', this)
    if (Array.isArray(value)) {
      if (hasProto) {
        protoAugment(value, arrayMethods)
      } else {
        copyAugment(value, arrayMethods, arrayKeys)
      }
      this.observeArray(value)
    } else {
      this.walk(value)
    }
  }
}
```

按照祖师爷的说法是：性能问题，性能代价和获得的用户体验收益不成正比

yyx990803 commented on 27 Jul 2018 · edited

要求写得很清楚，不要用 issue 问问题。

看你的文章研究得挺有热情，破例回答一下 就是因为性能问题。

yyx990803 commented on 27 Jul 2018

性能代价和获得的用户体验收益不成正比。

具体可参考[为什么vue没有提供对数组属性的监听](#)

实际处理

改写数组的push、pop等8个方法，让他们在执行之后通知数组更新了，缺点：[参见官网](#))

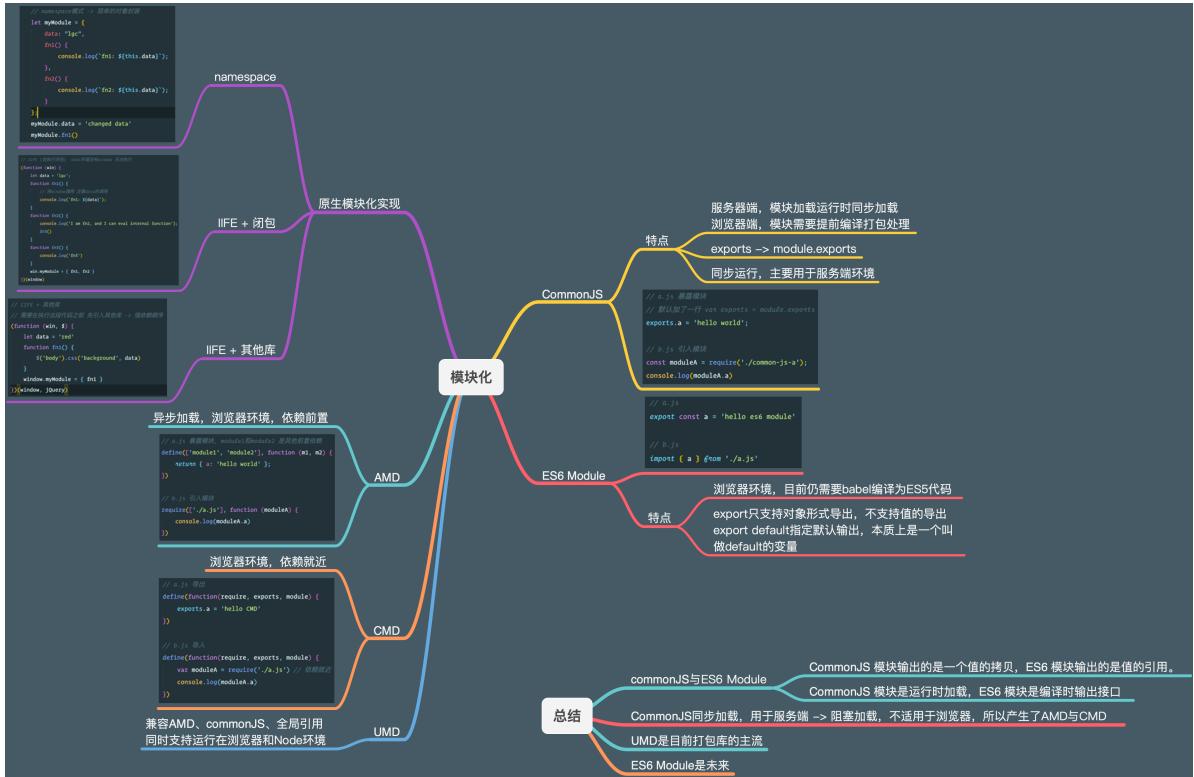
- 不能直接修改数组的长度 `this.list.length = 0`
- 通过下标去修改数组 `this.list[1] = 'a'`

改写步骤

- 继承原生 `Array` 的原型方法
- 对继承后的对象使用 `Object.defineProperty` 进行拦截
- 把被拦截后的响应式原型，赋值到数组数据的原型上

JS 模块化

#Front-End/JS



Common JS

- 模块可以多次加载, 只会在第一次加载时运行一次, 运行结果就会缓存下来, 要再次运行模块, 必须清除缓存
- 同步加载, 模块加载会阻塞后面代码的执行
- 用于服务器环境 (nodejs)

`exports` 只是对 `module.exports` 的引用, 相当于Node为每个模块提供了一个 `exports` 变量, 指向 `module.exports`, 相当于每个模块头部都有这么一行代码

```
var exports = module.exports
```

模块导出与引用

```
// common-js-a.js 暴露模块
// 默认加了一行 var exports = module.exports
exports.a = 'hello world';

// common-js-b.js 引入模块
const moduleA = require('./common-js-a');
console.log(moduleA.a)
```

AMD

- 异步加载
- 浏览器环境, 依赖前置

```
// a.js 暴露模块，module1和module2 是其他前置依赖
define(['module1', 'module2'], function (m1, m2) {
    return { a: 'hello world' };
})

// b.js 引入模块
require('./a.js'), function (moduleA) {
    console.log(moduleA.a)
})
```

CMD

- 浏览器环境
- 异步加载，就近依赖

```
// 异步加载 就近依赖

// a.js 导出
define(function(require, exports, module) {
    exports.a = 'hello CMD'
})

// b.js 导入
define(function(require, exports, module) {
    var moduleA = require('./a.js') // 依赖就近
    console.log(moduleA.a)
})
```

UMD

兼容AMD、commonJS、全局引用
同时支持运行在浏览器和Node环境

```
(function webpackUniversalModuleDefinition(root, factory) {
    // Test Comment
    if(typeof exports === 'object' && typeof module === 'object')
        module.exports = factory(require('lodash'));
    // Test Comment
    else if(typeof define === 'function' && define.amd)
        define(['lodash'], factory);
    // Test Comment
    else if(typeof exports === 'object')
        exports['someLibName'] = factory(require('lodash'));
    // Test Comment
    else
        root['someLibName'] = factory(root['_']);
})(this, function(__WEBPACK_EXTERNAL_MODULE_1__) {
    // ...
});
```

ES6 Module

```
// a.js
export const a = 'hello es6 module'

// b.js
import { a } from './a.js'
```

- 浏览器环境，目前仍需要babel编译为ES5代码
- export只支持对象形式导出，不支持值的导出，export default指定默认输出，本质上是一个叫做default的变量

总结

- CommonJS的同步加载机制主要用于服务器端，也就是Node，但与之相伴的阻塞加载特点并不适用于浏览器资源的加载，所以诞生了AMD、CMD规范
- AMD与CMD都可以在浏览器中异步加载模块，但实际上这两种规范的开发成本都比较高
- UMD同时兼容AMD、commonJS、全局引用等规范，算是目前打包JS库的主流吧
- ES6在语言标准的层面上实现了模块化，使用起来非常舒服，目前算是浏览器端的标准方案，再加上现代打包工具的加持，称霸Node服务端也指日可待

Node connect

#Front-End/js/Node/NodeInAction

[深入浅出Node.js \(七\) : Connect模块解析 \(之一\)](#)

一、简单的connect程序

Connect是第三方模块，不存在Node的默认安装之列，需要自行安装

```
npm install connect
```

最简单的connect程序

```
const connect = require('connect');
const app = connect();
app.listen(3000);
```

加上两个中间件

```
const connect = require('connect');

function logger(req, res, next) {
  console.log('%s %s', req.method, req.url);
  next()
}

// hello中没有next回调，因为这个组件结束了HTTP响应，所以不需要把控制器交回给分派器
function hello(req, res) {
  res.setHeader('Content-Type', 'text/plain');
  res.end('hello world');
}
```

```
connect().use(logger).use(hello).listen(3000);
```

中间件的顺序

当一个组件不调用 next() 时，命令链中的后续中间件用于不会被调用
试想下做用户认证的时候

```
connect()  
.use(logger)  
.use(authConfirm)  
.use(serviceStatic)  
.use(hello);
```

当用户通过 authConfirm 认证的时候，用于不会调用 next()，那么自然就进不去下一步

挂载

connect中有一个挂载的概念，可以给中间件或者整个程序定义一个路径前缀，除了为根路径重写请求，更厉害的是只对路径前缀（挂载点）内的请求调用中间件或者程序，为认证、管理、路由分割、错误处理打下了基础

错误处理

connect按照常规中间件的规则实现了一种用来处理错误的中间件变体，除了请求和响应对象，还接受一个错误对象作为参数

1、默认错误处理器

connect给出的默认错误响应是500，包含文本'Internal Server Error'和错误自身相信信息的响应主体

2、自行处理程序错误

错误处理中间件函数必须接受四个参数

err、req、res、next

```
function errorHandler() {  
  var env = process.env.NODE_ENV || 'development';  
  return function(err, req, res, next) {  
    res.statusCode = 500;  
    switch (env) {  
      case 'development':  
        res.setHeader('Content-Type', 'application/json');  
        res.end(JSON.stringify(err));  
        break;  
      default:  
        res.end('Server error');  
    }  
  }  
}
```

mongoDB基础

```
#develop/mongoDB
```

1、安装

mac安装很简单

```
brew install mongodb
```

其他系统请自行下载[MongoDB for GIANT Ideas | MongoDB](#)

2、找到安装目录

安装完后，目前homebrew会把MongoDB安装在这个目录下面

```
usr/local/cellar/mongodb
```

如果找不到 `usr` 目录，就一直 `cd ..` 往后退到mac的根目录

3、建立存储数据的文件夹

创建一个你喜欢的文件夹来存储你的数据，比如我就放在了 `/users/macbookpro-luoguangcong/Data-DB` 下面

4、启动MongoDB

进入mongoDB执行目录，现在安装的monoDB下面还有一个版本目录的，比如我的是3.6.4，并进入到 bin目录 `mongodb/3.6.4/bin`

执行命令 `mongod --dbpath <pathname>`, `pathname` 就是上面建立的存储数据的文件夹

```
mongodb/3.6.4/bin
▶ mongod --dbpath /users/macbookpro-luoguangcong/Data-DB
```

看到 `waiting for connections on port 27017` 这样的字眼，那就是OK了

打开浏览器输入 `localhost:27017`, 会看到 `It looks like you are trying to access MongoDB over HTTP on the native driver port.` 的字眼

如果不想要每次启动mongodb都要切换到 `/usr/local/cellar/mongoDB/3.6.4/bin` 目录下，我们可以将它添加到环境变量，操作如下

```
echo 'export PATH=/usr/local/cellar/mongoDB/3.6.4/bin:$PATH'>>~/.bash_profile
```

如果用的是iterm2，用的shell是zsh的话，把 `.bash_profile` 换成 `.zshrc`

重启命令终端，或者 `source ~/.zshrc` 即可生效

接下来我在任何目录下执行 `mongod --dbpath ~/Data-DB` 都可以直接开启了。

5、基础操作

在另一个命令行窗口进入如下操作

```
mongo
```

终端会一直出现 > 的符号，就可以输入各种命令了

[Tutorials-for-Web-Developers/MongoDB 极简实践入门.md at master · StevenSLXie/Tutorials-for-Web-Developers · GitHub](https://github.com/StevenSLXie/Tutorials-for-Web-Developers/blob/master/MongoDB%20极简实践入门.md)

显示数据库

```
> show dbs
admin    0.000GB
config   0.000GB
local    0.000GB
```

没有test这个库的话会直接新建一个

```
> use test
switched to db test
```

但 show dbs 的时候 test 不会出现，因为这时候这个数据库是空的

往 test 数据库里添加一个集合(collection)，集合类似于SQL中的表格

```
> db.createCollection('author')
{ "ok" : 1 }
```

这时候再 show dbs 和 show collections 就能看见test数据库相关的内容了

```
> show databases
admin    0.000GB
config   0.000GB
local    0.000GB
test     0.000GB
> show collections
author
```

如果不需要author这个集合，可以这样删除

```
db.author.drop()
```

集合(collection)类似于SQL的表格(table)，类似于Excel的一个个表格

6、插入

创建一个叫电影的集合

```
db.createCollection('movie')
```

插入数据

```
db.movie.insert(
{
  title: 'Forrest Gump',
  directed_by: 'Robert Zemeckis',
  stars: ['Tom Hanks', 'Robin Wright', 'Gary Sinise'],
```

```
tags: ['drama', 'romance'],
debut: new Date(1994, 7, 6, 0, 0),
likes: 864367,
dislikes: 30127,
comments: [
  {
    user: 'user1',
    message: 'My first comment',
    dateCreated: new Date(2013, 11, 10, 2, 35),
    like: 0
  },
  {
    user: 'user2',
    message: 'My first comment too!',
    dateCreated: new Date(2013, 11, 11, 6, 20),
    like: 0
  }
]
```

能看到如下输出 `writeResult({ "nInserted" : 1 })`

如下命令查找内容，但 `find()` 里面什么都没写，也就是说不做筛选，所以会全部返回，`pretty()` 是格式化，试一下就知道了

```
db.movie.find()
db.movie.find().pretty()
```

但是你会看到，结果中多了一个

```
"_id" : ObjectId("5b00cba4640212f1efd91c3c")
```

这是数据库自动创建的一个ID号，在同一个数据库里，每个文档的ID号都是不同的

7. 正确退出数据库

```
use admin;
db.shutdownServer();
```

到这里，已经有了最基础的基础认知，要学之后的各种查询、更新、搜索等等也就是看看API文档的事情。`# Node 数据储存`

`#Front-End/js/Node/NodeInAction`

DBMS：数据库管理系统

储存机制选择

- 1、无服务器的数据储存
- 2、关系型数据库管理系统：MySQL、PostgreSQL
- 3、NoSQL数据库：Redis、MongoDB、Mongoose

一、无服务器的数据储存

- 1、内存储存
- 2、基于文件的储存

1、内存储存

使用变量储存数据

但是，一旦服务器和程序重启后，数据就丢失了

```
const http = require('http');
const counter = 0;

http.createServer((req, res) => {
  counter++;
  res.write(`I have been assessed ${counter} times.`)
}).listen(3000);
```

2、基于文件的储存

可以做数据的持久化保存，经常用来储存程序的配置信息，服务器和程序重启后依然有效

并发问题：如果程序有多个用户同时读写该文件，可能会出现并发问题
所以数据库管理系统是更合理的选择

二、关系型数据库管理系统 (RDBMS)

这里只记录了MySQL的基本操作

1、安装 MySQL Node模块

```
npm install mysql
```

2、创建服务器程序逻辑 (服务器+基本的路由逻辑+连接上数据库)

```
const http = require('http');
const work = require('./timetrack');
const mysql = require('mysql');

// 连接MySQL
const db = mysql.createConnection({
  host: '127.0.0.1',
  user: 'root',
  password: '123456',
  database: 'timetrack'
});

// HTTP 请求路由
const server = http.createServer((req, res) => {
  switch (req.method) {
    case 'POST':
      switch (req.url) {
        case '/':
          break;
        case '/start':
          work.start();
          res.end('started');
          break;
        case '/stop':
          work.stop();
          res.end('stopped');
          break;
        case '/status':
          res.end(`Time tracked: ${work.get()} ms`);
          break;
      }
    case 'GET':
      if (req.url === '/') {
        res.end(`Time tracked: ${work.get()} ms`);
      } else {
        res.end('Not found');
      }
  }
}).listen(3001, () => {
  console.log('listening on port 3001');
});
```

```

        work.add(db, req, res);
        break;
    case '/archive':
        work.archive(db, req, res);
        break;
    case '/delete':
        work.delete(db, req, res);
        break;
    }
    break;
case 'GET':
    switch (req.url) {
        case '/':
            work.show(db, res);
            break;
        case '/archive':
            work.showArchived(db, res);
            break;
    }
    break;
}
});

// 注意，不直接启动服务器，由数据库表来启动
// server.listen(3000);

db.query(
    "CREATE TABLE IF NOT EXISTS work (
        + "id INT(10) NOT NULL AUTO_INCREMENT, "
        + "hours DECIMAL(5,2) DEFAULT 0, "
        + "date DATE, "
        + "archived INT(1) DEFAULT 0, "
        + "description LONGTEXT, "
        + "PRIMARY KEY(id))",
    err => {
        if (err) {
            throw err;
        }
        console.log('Server started...');
        server.listen(3000, '127.0.0.1');
    }
)

```

3、创建数据库基本操作辅助函数

详见 ./Node/Node Store Data/MySQL/timettrack.js 文件

三、NoSQL数据库

关系型DBMS为可靠性牺牲了性能

但NoSQL数据库却把性能放在了第一位

所以，对于**实时分析**或**消息传递**而言，NoSQL可能是更好的选择

它不需要预先定义数据schema

1、Redis

非常适合处理那些不需要长期访问的简单数据储存，比如短信和游戏中的数据

Redis把数据存在RAM中，并在磁盘中记录数据的变化

好处：数据操作非常快

缺点：存储空间有限

如果Redis服务器崩溃，RAM的内存丢失，可以用磁盘中的日志回复数据

2、MongoDB

通用的非关系型数据库，使用RDBMS的那类程序都可以使用MongoDB

MongoDB把文档（document）存在集合（collection）中，数据（更多是json格式）存在document中

3、Mongoose

Mongoose是一个Node模块，并不是数据库，它可以让你更顺畅的使用MongoDB

四、总结

内存储存：极度关心速度和性能，不关心程序重启的数据持久化

文件储存：不关心性能，数据不复杂，类似命令行程序

SQL可靠严谨，性能和灵活性上欠佳

MongoDB是极佳的通用DBMS，Redis擅长处理变化频繁、相对简单的数据

例如，要构建内容管理系统

文件储存Web程序的配置选项，MongoDB储存文章，Redis储存用户的评论和文章评级

快速上手node+express+MongoDB

一、Node+Express

[Express - Node.js web application framework](#)

能快速搭建hello world即可

- app.get、app.post分别开发get和post接口
- app.use使用模块
- res.send、res.json、res.sendFile响应不同的内容

二、nodemon (自动重启，类似热更新)

```
npm install -g nodemon
```

三、MongoDB

具体操作可以看前面写的这篇

[mongoDB基础](#)

四、mongoose：操作MongoDB

安装mongoose，操作JSON

```
npm install mongoose --save
```

然后在前面的Express生成器生成的项目中的app.js文件加入如下代码

```
var mongoose = require('mongoose');

// 这里会自动连接test数据库，如果没有则会自动创建
mongoose.connect('mongodb://localhost/test');

var Cat = mongoose.model('Cat', {
  name: String,
  friends: [String],
  age: Number,
});

var kitty = new Cat({ name: 'zildjian', friends: ['tom', 'jerry'] });
kitty.age = 3;
kitty.save(function (err) {
  if (err) // ...
  console.log('meow');
});
```

在上面关于[MongoDB基础](#)的教程中启动MongoDB

然后在express项目中启动app set DEBUG=myapp:* & npm start

mongodb启动的那个命令终端会显示连接

```
2018-05-20T16:34:00.024+0800 I NETWORK [listener] connection accepted from
127.0.0.1:64736 #3 (2 connections now open)
```

再回到mongo的操作终端，`show collections`会看到cats

再执行`db.cats.find().pretty()`就能看到对应的数据了# Node web服务器

#Front-End/js/Node

最简单的HTTP服务器

```
const http = require('http');
const server = http.createServer((req, res) => {
  res.end('Hello World');
})
server.listen(3000);
```

静态文件服务器

```
const http = require('http');
const parse = require('url').parse;
const join = require('path').join;
const fs = require('fs');

const root = __dirname;
```

```

const server = http.createServer((req, res) => {
  const url = parse(req.url);
  // 构造绝对路径
  const path = join(root, url.pathname);
  // 判断文件是否存在
  fs.stat(path, (err, stat) => {
    if (err) {
      // 如果文件不存在, fs.stat()会在err.code中放入ENOENT作为响应
      if ('ENOENT' === err.code) {
        res.statusCode = 404;
        res.end('Not Found');
      } else {
        // 其他错误, 返回通用错误码500
        res.statusCode = 500;
        res.end('Internal Server Error');
      }
    } else {
      //
      res.setHeader('Content-Length', stat.size);
      let stream = fs.createReadStream(path);
      stream.pipe(res);
      stream.on('error', err => {
        res.statusCode = 500;
        res.end('Internal Server Error');
      })
    }
  })
}

server.listen(3000);

```

HTTPS服务器

HTTPS把HTTP和TLS/SSL传输层结合到一起，数据经过加密，难以窃听

在Node里面使用HTTPS，需要一个私钥+一份证书

一、私钥

私钥：本质上是个密钥，可以用来解密客户端发给服务器的数据，存放在服务器的一个文件里，不可信用户无法访问

如何生成私钥？

输入下面的命令行，openssl在安装node的时候已经装好了

```
openssl genrsa 1024 > key.pem
```

```

▶ openssl genrsa 1024 > key.pem
Generating RSA private key, 1024 bit long modulus
.....+++++
.....+++++
e is 65537 (0x10001)

```

二、证书

证书可以分享，包含了公钥和证书持有者的信息

公钥用来加密从客户端发往服务器的数据

如何创建证书？

创建证书需要私钥，已经有了

输入如下命令行

```
openssl req -x509 -new -key key.pem > key-cert.pem
```

会有一些问题，关于证书信息的，直接输入即可，如下所示

```
► openssl req -x509 -new -key key.pem > key-cert.pem
You are about to be asked to enter information that will be incorporated
into your certificate request.
what you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) []:ZH
State or Province Name (full name) []:GD
Locality Name (eg, city) []:GZ
Organization Name (eg, company) []:personal
Organizational Unit Name (eg, section) []:personal
Common Name (eg, fully qualified host name) []:personal
Email Address []: *****@qq.com
```

到目前为止，当前目录下已经有了 `key.pem` 和 `key-cert.pem` 两份文件，私钥最好放到 `~/.ssh` 目录

不过这里是本地开发和测试，就留在本地吧，跑程序会显示警告信息

如果要把网址部署到公网上，就应该找个证书颁发机构（CA）进行注册，获取真实、受信的证书。

三、配置HTTPS服务器

```
const https = require('https');
const fs = require('fs');

const options = {
  key: fs.readFileSync('./key.pem'),
  cert: fs.readFileSync('./key-cert.pem')
};

https.createServer(options, (req, res) => {
  res.writeHead(200);
  res.end('Hello https');
}).listen(3000);
```

Node模块机制之require源码

CommonJS规范为Javascript制定了一个美好的愿景——希望Javascript能够在任何地方运行

CommonJS规范

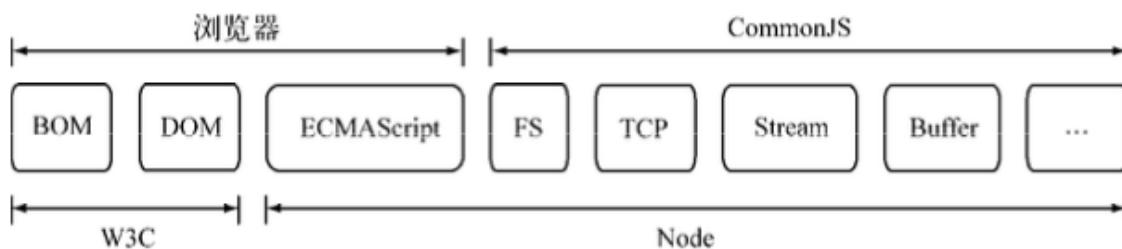


图2-2 Node与浏览器以及W3C组织、CommonJS组织、ECMAScript之间的关系

在Node中，一个文件就是一个模块，每个模块拥有独立的作用域、变量、方法
在模块上下文中

- `module` 变量代表当前模块
- 通过 `require` 方法来引入模块
- 提供了 `exports` 对象来导出当前模块的变量或者方法

模块的分类

在Node中，模块分为两大类

- 核心(原生)模块：Node提供的
- 内建模块：由纯 C/C++ 编写提供的
- 全局模块：Node启动时，生成的全局变量，比如 `process`
- 文件模块：用户编写的模块
- 普通模块：`node_modules` 下的模块，或者用户自己编写的文件
- 外部编写的 C++ 模块

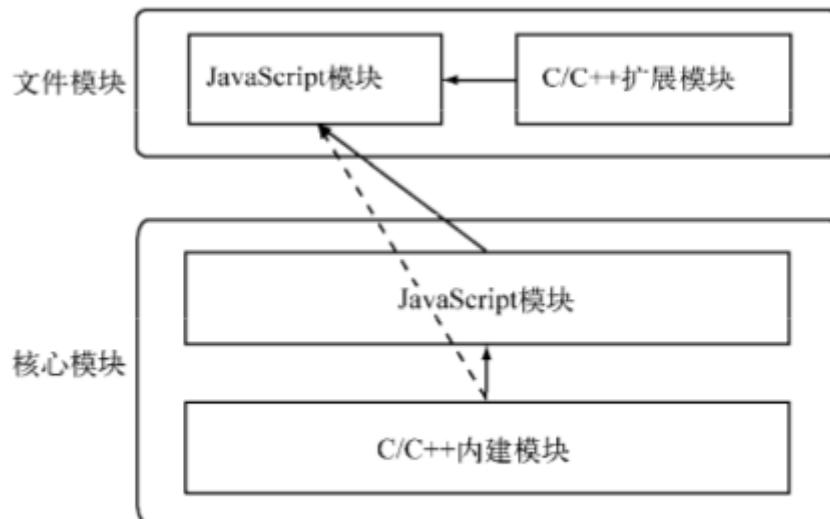


图2-8 模块之间的调用关系

require 伪代码算法

在读源码之前，先看看官方文档关于 `require` 的内部实现，写的非常详细了
[Modules: CommonJS modules | Node.js v15.8.0 Documentation](#)

```

require(x) from module at path Y
1. If x is a core module,
   a. return the core module
   b. STOP
2. If x begins with '/'
   a. set Y to be the filesystem root
3. If x begins with './' or '/' or '../'
   a. LOAD_AS_FILE(Y + X)
   b. LOAD_AS_DIRECTORY(Y + X)
   c. THROW "not found"
4. If x begins with '#'
   a. LOAD_PACKAGE_IMPORTS(X, dirname(Y))
5. LOAD_PACKAGE_SELF(X, dirname(Y))
6. LOAD_NODE_MODULES(X, dirname(Y))
7. THROW "not found"

```

简单翻译一下 (建议还是直接读最新的英文文档)

在路径Y下require(x)模块

1. 如果 x 是核心模块
 - a. 返回该核心模块 # 在node进程启动时，部分核心模块会被编译成二进制，被加载进内存了
 - b. 返回，不再继续往下执行
2. 如果 x 以 '/' 开头
 - a. 将 Y 设置为文件系统根目录
3. 如果 x 以 './' 或者 '/' 或者 '../' 开头
 - a. LOAD_AS_FILE(Y + X) # LOAD_AS_FILE(X) 依次寻找 x、x.js、x.json、x.node
 - b. LOAD_AS_DIRECTORY(Y + X) # LOAD_AS_DIRECTORY 依次寻找X/package.json里的main字段、X/index.js、X/index.json、X/index.node
 - c. 抛出异常 "not found"
- # 4和5可以先忽略，不影响理解
4. 如果以 '#' 开头
 - a. LOAD_PACKAGE_IMPORTS(X, dirname(Y))
5. LOAD_PACKAGE_SELF(X, dirname(Y))
- # 加载node模块，依次向上寻找可能的目录，依次当做文件、目录名加载，这里可能有点难理解，参考下面例子
6. LOAD_NODE_MODULES(X, dirname(Y))
7. THROW "not found"

举个例子，在文件 /home/dir1/a.js 内执行 require('b')，直接跑到典型的第6步，会依次先搜索以下目录

- /home/dir1/node_modules/b
- /home/node_modules/b
- /node_modules/b

搜索每个目录时，先把bar当成文件来查找，依次查找以下文件

- b
- b.js
- b.json
- b.node

如果都找不到，就把b当做目录来查找，依次加载以下文件

- b/package.json 里面的 main 字段代表的文件
- b/index.js
- b/index.json

- `b/index.node`

module是什么

建个`index.js`文件，然后打印`console.log(module)`，运行输出如下

```
# console.log(module)
$ node src/index.js
Module {
  id: '.',
  path: '/Users/lgc/code-repo/node-repo/fake-module/src',
  exports: {},
  parent: null,
  filename: '/Users/lgc/code-repo/node-repo/fake-module/src/index.js',
  loaded: false,
  children: [],
  paths: [
    '/Users/lgc/code-repo/node-repo/fake-module/src/node_modules',
    '/Users/lgc/code-repo/node-repo/fake-module/node_modules',
    '/Users/lgc/code-repo/node-repo/node_modules',
    '/Users/lgc/code-repo/node_modules',
    '/Users/lgc/node_modules',
    '/Users/node_modules',
    '/node_modules'
  ]
}
```

Module源码

先把node源码clone一份到本地 [GitHub - nodejs/node](#)

用IDE可以直接找到`Module`定义的地方 `lib/internal/modules/cjs/loader.js`

为方便理解，下面基本是伪代码，在源码的基础上尽量只挑重要的、核心的代码，减少干扰

```
function Module(id = '', parent) {
  this.id = id; // require的路径
  this.path = path.dirname(id); // 获取id对应的文件路径
  this.exports = {}; // 要导出的内容，先初始化为空对象
  moduleParentCache.set(this, parent);
  updateChildren(parent, this, false);
  this.filename = null; // 模块文件名
  this.loaded = false; // 标志当前模块是否已加载
  this.children = [];
}

Module._cache = ObjectCreate(null); // 创建一个空的缓存对象
Module._extensions = ObjectCreate(null); // 创建一个扩展名对象（跟上面说的js、json、node这些扩展名相关，这里先不管，后面就清楚了）
```

`Module`的初始化并不复杂，根据打印出来的内容，对应一下就了解了

require源码

每个模块实例都有一个 `require` 方法，挂在 `Module.prototype` 上，还是在

`lib/internal/modules/cjs/loader.js`

```
Module.prototype.require = function(id) {
    return Module._load(id, this, /* isMain */ false);
};
```

这里可以知道，`require` 并不是全局的命令，而是每个模块提供的内部方法，只有在模块内部才能使用，它调用的是 `Module._load` 方法

```
Module._load = function(request, parent, isMain) {
    // 1、计算绝对路径
    const filename = Module._resolveFilename(request, parent, isMain);

    // 2、取出缓存，直接返回
    const cachedModule = Module._cache[filename];
    if (cachedModule !== undefined) {
        return cachedModule.exports;
    }

    // 3、如果是内置模块，直接返回
    const mod = loadNativeModule(filename, request);
    if (mod && mod.canBeRequiredByUsers) return mod.exports;

    // 4、实例化Module，并存入缓存(此时缓存也是空的)
    const module = cachedModule || new Module(filename, parent);
    Module._cache[filename] = module;

    // 5、加载模块
    try {
        module.load(filename);
    } finally {
        // 如果有异常，删除缓存
        if (threw) {
            delete Module._cache[filename];
        }
    }

    // 6、返回模块的exports属性，并不是返回module
    return module.exports;
};
```

这里比较核心的两个地方

```
Module._resolveFilename() # 计算模块路径
module.load(filename); # 加载模块
```

计算模块路径 `Module._resolveFilename`

```
Module._resolveFilename = function(request, parent, isMain, options) {
    // 如果是内置模块，直接返回request，也就是从最开始传入的 id
    if (NativeModule.canBeRequiredByUsers(request)) {
```

```

        return request;
    }

    // 确定路径，也就是层层往上寻找node_modules
    let paths;
    paths = Module._resolveLookupPaths(request, parent);

    // 确认最终的filename
    const filename = Module._findPath(request, paths, isMain, false);
    if (filename) return filename;
}

```

`Module.resolveLookupPaths()`方法是列出所有可能的路径，层层往上找，这里不继续看它的源码了，将所有可能的path传入`Module._findPath`找到模块的最终绝对路径

```

Module._findPath = function(request, paths, isMain) {
    // 是否绝对路径
    const absoluteRequest = path.isAbsolute(request);
    if (absoluteRequest) {
        paths = [''];
    } else if (!paths || paths.length === 0) {
        return false;
    }

    // 如果缓存中有该路径，直接返回
    const entry = Module._pathCache[cacheKey];
    if (entry) return entry;

    // 遍历所有可能的路径，前面Module.resolveLookupPaths返回的
    for (let i = 0; i < paths.length; i++) {
        let filename;
        const rc = stat(basePath);
        if (!trailingSlash) {
            // 当做文件寻找
            if (rc === 0) {
                filename = toRealPath(basePath);
            }
        }

        // 依次加上后缀名来找，js、json、node
        if (!filename) {
            filename = tryExtensions(basePath, exts, isMain);
        }
    }

    // 如果是目录，按照package.json['main']、index.js、index.json、inde.node往下
    找
    if (!filename && rc === 1) {
        filename = tryPackage(basePath, exts, isMain, request);
    }

    // 将找到的路径存入缓存
    if (filename) {
        Module._pathCache[cacheKey] = filename;
        return filename;
    }
}

```

```
    }
    // 没有找到路径，返回false
    return false;
};
```

加载模块 module.load()

找到模块的绝对路径，就可以开始加载模块了，再回到 module.load() 方法

```
Module.prototype.load = function(filename) {
    const extension = findLongestRegisteredExtension(filename);
    Module._extensions[extension](this, filename);
    this.loaded = true;
};
```

先确认模块的后缀名，不同的后缀名对应不同的处理方法，这里只看 js 和 json 的处理
注意一下 fs.readFileSync，模块的加载都是同步的

```
Module._extensions['.js'] = function(module, filename) {
    content = fs.readFileSync(filename, 'utf8');
    module._compile(content, filename);
};

// Native extension for .json
Module._extensions['.json'] = function(module, filename) {
    const content = fs.readFileSync(filename, 'utf8');
    try {
        module.exports = JSONParse(stripBOM(content));
    } catch (err) {
        err.message = filename + ': ' + err.message;
        throw err;
    }
};
```

.json 文件的处理比较简单，直接 JSONParse 就完事了，有异常就抛出
重点看一下 .js 文件的处理

加载 .js 模块

首先通过 fs.readFileSync 同步读取文件内容，然后执行 module._compile

```

Module.prototype._compile = function(content, filename) {
    //
    const compiledwrapper = wrapSafe(filename, content, this);
    const require = makeRequireFunction(this, redirects);
    let result;
    const exports = this.exports;
    const thisvalue = exports;
    const module = this;
    // 这里涉及到上下文的传递
    result = ReflectApply(compiledwrapper, thisvalue, [exports, require, module,
filename, dirname]);
    return result;
};

```

在编译的过程中，Node对获取的JS文件内容进行了头尾包装，每个模块文件直接都进行了作用域隔离，按照以下的格式导出

```

(function (exports, require, module, __filename, __dirname) {
    // 模块源码
    exports.[xxx] = fn
});

```

这一段本质上是利用node的虚拟机模块 `vm` 的 `runInThisContext` 方法将字符串（文件内容）编译成一个函数，将 `exports` 属性、`require` 方法、`module`（模块对象本身）、`__filename`、`__dirname` 等变量作为参数传递给这个函数执行，注入到模块上下文

返回结果

再回到最上面的 `Module._load` 中，`return module.exports;`

模块可以任意修改 `module.exports` 的值作为最终输出结果

```

exports = {
    a: '1'
}

module.exports = {
    a: '1'
}

```

像上面这类代码，并不会改变模块的导出结果，只是改变了 `exports` 这个变量而已，看个例子

```

let a = 'a';

console.log('module.exports', module.exports); // 空对象 {}
console.log('exports', exports); // 空对象 {}

exports.a = 1; // 修改本模块的值为 { a: 1 }

exports = '88咯'; // 修改exports的引用，但并没有修改本模块

console.log('module', module);

```

执行结果如下

```
module.exports {}

exports {}

module Module {
  id: '..',
  path: '...',
  exports: { a: 1 },
  parent: null,
  filename: '...',
  loaded: false,
  children: [],
  paths: [
    # ...
  ]
}
```

可以看到模块的值依然是 `{ a: 1 }`，`exports`的引用更改并不会修改模块本身值

总结

以上的流程，简单总结起来

```
Resolution (解析) -> Loading (加载) -> Wrapping (私有化) -> Evaluation (执行) ->
Caching (缓存)
```

题外：CommonJS和ES6 Modules的循环引用问题

CommonJS循环引用原则：一旦出现某个模块被 `循环引用`，就只输出已经执行的部分，还未执行的部分不会输出

ES6 Modules加载原理：遇到模块加载命令 `import` 时，不会去执行模块，而是生成一个引用，等到真正需要用时，再到模块内去取值，因此ES6模块是动态引用，不存在缓存值的问题，而且模块里面的变量，绑定其所在的模块

Reference

- [Modules: CommonJS modules | Node.js v15.8.0 Documentation](#) Node官方文档
- 《深入浅出NodeJS》 — 朴灵
- [彻底搞懂 Node.js 中的 Require 机制\(源码分析到手写实践\)](#)
- [require\(\) 源码解读 - 阮一峰的网络日志](#)
- [JavaScript 模块的循环加载 - 阮一峰的网络日志](#)
- [Node.js 如何处理 ES6 模块 - 阮一峰的网络日志](#)

Node 多进程与多线程

目录

- 服务模型的变化
 - 同步

- 复制进程
- 多线程
- 事件驱动
- Node多进程
- Nodejs进程创建 [child_process | Node.js API 文档](#)
- Nodejs多进程架构模型
- cluster应用与原理
 - 如何多个进程监听一个端口的
 - 多个进程之间如何通信 (IPC)
 - Node.js 是如何进行负载均衡请求分发的 (RoundRobin)
- 如何守护进程 (PM2)

服务模型

- 同步
 - 假设每次响应服务耗时N秒, QPS为 $1/N$
- 复制进程
 - N 个连接启动N个进程来服务
 - 缺点: 复制进程内部状态带来浪费, 并发过高时内存随着进程数增长会耗尽
 - 如果进程上限为M个, QPS为 M/N
- 多线程
 - 线程之间共享数据, 内存浪费的问题得到一定解决, 还可以利用线程池减少创建和销毁线程的开销
 - CPU核心一个时刻只能做一件事, 操作系统需要通过**时间切片**的方法来不断切换线程的上下文, 线程数量过多时, 时间耗在上下文切换
- 事件驱动
 - 基于事件驱动的单线程, 避免了不必要的内存开销和上下文切换开销, 可伸缩性比较高, 影响性能的点在于CPU的计算能力
 - 核心: **多核CPU利用率不足**

Node多进程

demo

```
// master.ts
const path = require('path');
const { fork } = require('child_process');
const numCPUs = require('os').cpus().length;

for (let i = 0; i < numCPUs; i++) {
  fork(path.resolve(__dirname, './worker.ts'));
}
```

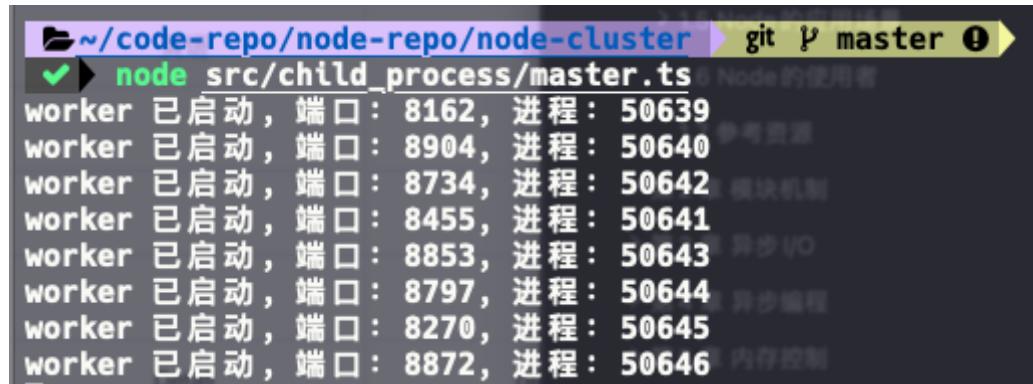
```
// worker.ts
const http = require('http');

const PORT = Math.round(8001 + Math.random() * 1000);

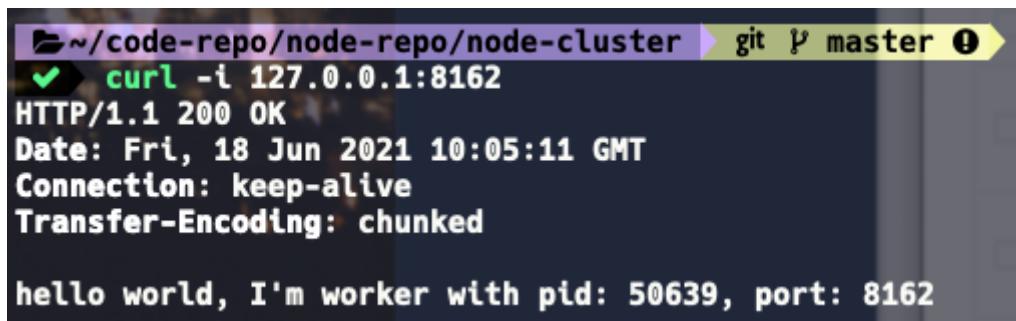
http.createServer((req, res) => {
  res.writeHead(200);
  res.end(`hello world, I'm worker with pid: ${process.pid}, port: ${PORT}
\n`);
}).listen(PORT);

console.log(`worker 已启动, 端口: ${PORT}, 进程: ${process.pid}`);
```

启动服务，然后访问其中一个子进程监听的端口



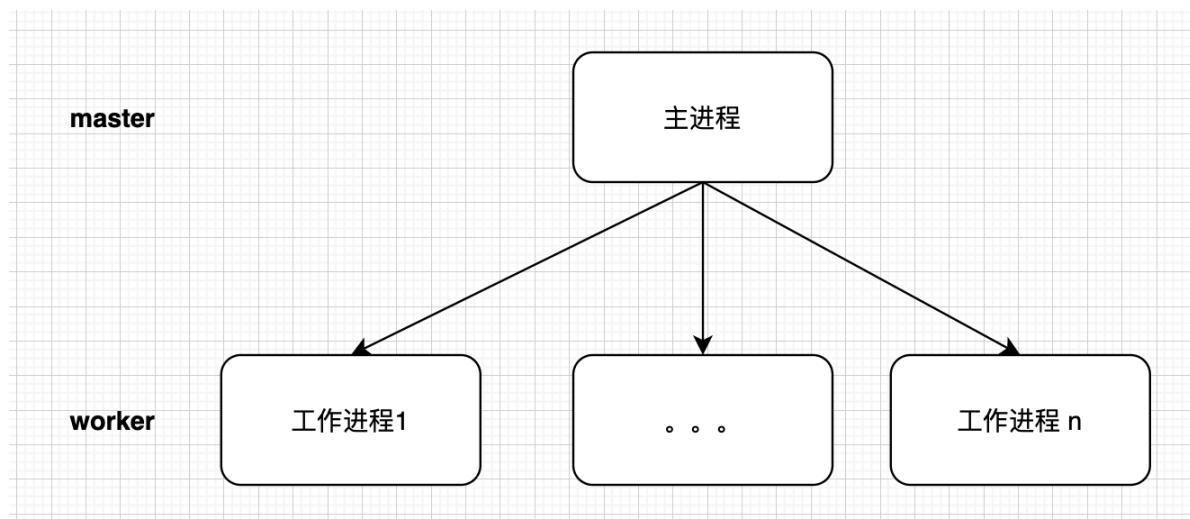
```
~/code-repo/node-repo/node-cluster git [master] ✘
✓ node src/child_process/master.ts
worker 已启动, 端口: 8162, 进程: 50639
worker 已启动, 端口: 8904, 进程: 50640
worker 已启动, 端口: 8734, 进程: 50642
worker 已启动, 端口: 8455, 进程: 50641
worker 已启动, 端口: 8853, 进程: 50643
worker 已启动, 端口: 8797, 进程: 50644
worker 已启动, 端口: 8270, 进程: 50645
worker 已启动, 端口: 8872, 进程: 50646
```



```
~/code-repo/node-repo/node-cluster git [master] ✘
✓ curl -i 127.0.0.1:8162
HTTP/1.1 200 OK
Date: Fri, 18 Jun 2021 10:05:11 GMT
Connection: keep-alive
Transfer-Encoding: chunked

hello world, I'm worker with pid: 50639, port: 8162
```

典型的【Master-Worker】模式，也叫主从模式



创建子进程 child_process

[Child process | Node.js v16.3.0 Documentation](#)

[child_process 子进程 | Node.js API 文档](#)

进程间通信

master和worker之间可以利用IPC进行通信，通过 `process.on('message')` 和 `process.send()`

```
// master.ts
const path = require('path');
const { fork } = require('child_process');
const numCPUs = require('os').cpus().length;

for (let i = 0; i < numCPUs; i++) {
  const worker = fork(path.resolve(__dirname, './worker.ts'));

  worker.on('message', msg => {
    console.log(`master get message from worker: ${msg}`);
  });
  worker.send('我是爸爸');
}
```

```
// wrker.ts
const http = require('http');

const PORT = Math.round(8001 + Math.random() * 1000);

http.createServer((req, res) => {
  res.writeHead(200);
  res.end(`hello world, I'm worker with pid: ${process.pid}, port: ${PORT}
\n`);
}).listen(PORT);

console.log(`worker 已启动， 端口: ${PORT}, 进程: ${process.pid}`);

process.on('message', msg => {
  console.log(`worker get message from master: ${msg}`);
});
process.send(`${PORT} ready`);
```

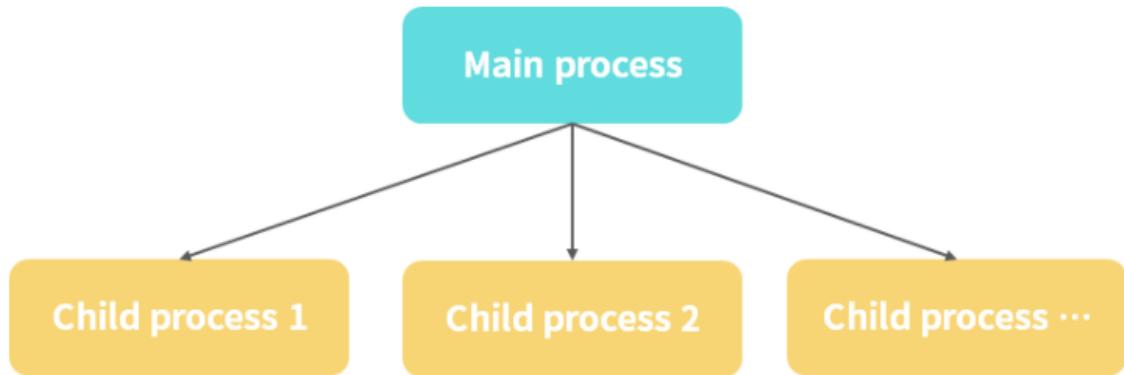
上面是主进程与子进程之间的通信，那么子进程与子进程之间呢？

[Node Cluster Workers IPC](#)

Cluster应用与原理

[cluster | Node.js API 文档](#)

为了充分利用主机的多核CPU能力，Node 提供了cluster模式用于实现**多进程分发策略**
其实就是上面的官方封装版



有两种实现模式

- 主进程监听一个端口，子进程不监听端口，通过主进程分发请求到子进程 (cluster模式的实现)
- ~~主进程与子进程分别监听不同端口，通过主进程分发请求到子进程~~

cluster demo

```

const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
    console.log(`主进程 ${process.pid} 正在运行`);

    for (let i = 0; i < numCPUs; i++) {
        cluster.fork();
    }

    cluster.on('exit', (worker, code, signal) => {
        console.log(`工作进程 ${worker.process.pid} 已退出`);
    });
} else {
    http.createServer((req, res) => {
        res.writeHead(200);
        res.end(`hello world, start with cluster ${process.pid}\n`);
    }).listen(8000);

    console.log(`工作进程 ${process.pid} 已启动`);
}

```

- 先判断是否主进程，是则创建具体的http服务
- 如果是子进程，则使用 `cluster.fork()` 创建子进程

启动成功后，再通过 curl -i http://127.0.0.1:8000 多次访问，后面的进程ID比较有规律，都是我们fork出来的子进程的ID

```
~/code-repo/node-repo/node-cluster
curl -i http://127.0.0.1:8000
HTTP/1.1 200 OK
Date: Mon, 07 Jun 2021 02:13:02 GMT
Connection: keep-alive
Transfer-Encoding: chunked

hello world, start with cluster 28898

~/code-repo/node-repo/node-cluster
curl -i http://127.0.0.1:8000
HTTP/1.1 200 OK
Date: Mon, 07 Jun 2021 02:13:05 GMT
Connection: keep-alive
Transfer-Encoding: chunked

hello world, start with cluster 28899

~/code-repo/node-repo/node-cluster
curl -i http://127.0.0.1:8000
HTTP/1.1 200 OK
Date: Mon, 07 Jun 2021 02:13:07 GMT
Connection: keep-alive
Transfer-Encoding: chunked

hello world, start with cluster 28901

~/code-repo/node-repo/node-cluster
curl -i http://127.0.0.1:8000
HTTP/1.1 200 OK
Date: Mon, 07 Jun 2021 02:13:08 GMT
Connection: keep-alive
Transfer-Encoding: chunked

hello world, start with cluster 28902
```

原理

对比 child_process 和 cluster 的实现，有一个很大的区别

- child_process 的实现，每个 worker 监听单独的接口
- cluster 模式只由 master 监听一个端口，再转发给所有 worker

那么，围绕下面3个问题出发，从源码找一下答案

- Node.js 的 cluster 是如何做到多个进程监听一个端口的
- 多个进程之间如何通信
- Node.js 是如何进行负载均衡请求分发的 (RoundRobin)

看 cluster 的入口

```
// lib/cluster.js

'use strict';

const childOrPrimary = 'NODE_UNIQUE_ID' in process.env ? 'child' : 'primary';
module.exports = require(`internal/cluster/${childOrPrimary}`);
```

- 如果没有设置过进程环境变量 `NODE_UNIQUE_ID` (cluster实现里的一个自增id), 则为master进程 (第一次调用)
- 否则为子进程

分部require不同的文件

- `lib/internal/cluster/primary.js`
- `lib/internal/cluster/child.js`

先看master的创建过程, 调用 `cluster.fork` 时首先调用了 `cluster.setupPrimary` 方法, 通过全局变量 `initialized` 来区分是否首次

```
cluster.setupPrimary = function(options) {
    // ...
    // 通过全局变量来区分是否首次创建
    if (initialized === true)
        return process.nextTick(setupSettingsNT, settings);

    initialized = true;
    //...
```

看 `cluster.fork` 具体实现

```
cluster.fork = function(env) {
    // 1、创建主进程,
    cluster.setupPrimary();
    const id = ++ids;
    // 2、创建 worker 子进程, 最终是通过 `child_process` 来创建子进程
    const workerProcess = createWorkerProcess(id, env);
    const worker = new Worker({
        id: id,
        process: workerProcess
    });

    worker.on('message', function(message, handle) {
        cluster.emit('message', this, message, handle);
    });
    // ...
    process.nextTick(emitForkNT, worker);
    cluster.workers[worker.id] = worker;
    return worker;
};
```

- 在最初我们写的初始化代码中, 调用了 `numCPUs` 次 `cluster.fork` 方法, 通过 `createWorkerProcess` 创建了 `numCPUs` 个子进程, 本质上是调用 `child_process.fork`

- `child_process.fork()` 的时候会重新运行 node 的启动命令，这时候就会重新调用根目录下的 `lib/cluster.js` 来启动新实例
- 由于 `cluster.isMaster` 为 `false`，因此调用 `lib/internal/cluster/child.js` 模块
- 由于是 worker 进程，最顶部的代码开始执行创建 http 服务的逻辑
- （这段看后面）在 http 服务中虽然是启动了监听端口，由于监听端口的方法被重写了，因此只是向主进程发送了一个消息，告诉父进程可以向我发送消息了，因此可以一个端口多个进程来服务

```

if (cluster.isMaster) {
    // ...
} else {
    http.createServer((req, res) => {
        res.writeHead(200);
        res.end(`hello world, start with cluster ${process.pid}\n`);
    }).listen(8000);

    console.log(`工作进程 ${process.pid} 已启动`);
}

```

在 master 的 server 监听了具体的端口，重点在于 `server.listen` 方法，代码在 `lib/net.js`。
`Server.prototype.listen` 重点调用了 `listenInCluster` 方法

```

function listenInCluster(server, address, port, addressType,
                         backlog, fd, exclusive, flags) {
    exclusive = !!exclusive;

    if (cluster === undefined) cluster = require('cluster');

    // 1、如果是主进程，开始真实监听端口启动服务
    if (cluster.isPrimary || exclusive) {
        // will create a new handle
        // _listen2 sets up the listened handle, it is still named like this
        // to avoid breaking code that wraps this method
        server._listen2(address, port, addressType, backlog, fd, flags);
        return;
    }

    const serverQuery = {
        address: address,
        port: port,
        addressType: addressType,
        fd: fd,
        flags,
    };
    // 2、非主进程，调用 cluster._getServer
    // Get the primary's server handle, and listen on it
    cluster._getServer(server, serverQuery, listenOnPrimaryHandle);

    function listenOnPrimaryHandle(err, handle) {
        // ...
    }
}

```

重点看看非主进程，调用的 `cluster._getServer` 方法，定义在 `lib/internal/cluster/child.js` 模块

```
obj.once('listening', () => {
  cluster.worker.state = 'listening';
  const address = obj.address();
  message.act = 'listening';
  message.port = (address && address.port) || options.port;
  send(message);
});
```

`cluster._getServer` 主要做了两件事

- 向master进程注册该worker，若master进程是第一次接收到监听此端口/描述符下的worker，则起一个内部TCP服务器，来承担监听该端口/描述符的职责，随后在master中记录下该worker
- Hack掉worker进程中的net.Server实例的listen方法里监听端口/描述符的部分，使其不再承担该职责

第一个问题：为什么多个进程server可以监听同一个port？

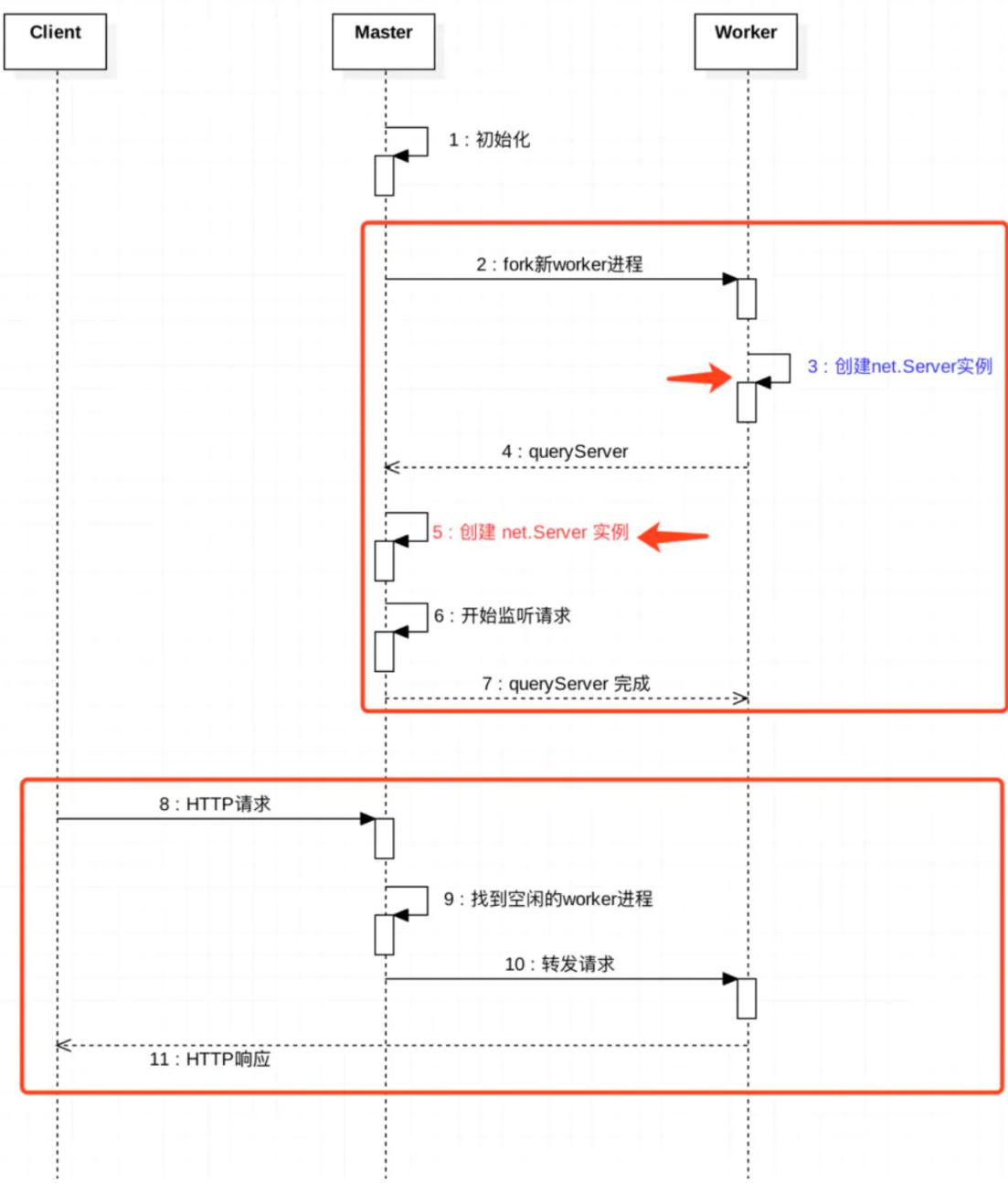
The first one (and the default one on all platforms except Windows), is the round-robin approach, where the master process listens on a port, accepts new connections and distributes them across the workers in a round-robin fashion, with some built-in smarts to avoid overloading a worker process. 第一种方法（也是除 Windows 外所有平台的默认方法）是循环法，由主进程负责监听端口，接收新连接后再将连接循环分发给工作进程，在分发中使用了一些内置技巧防止工作进程任务过载。

The second approach is where the master process creates the listen socket and sends it to interested workers. The workers then accept incoming connections directly. 第二种方法是，主进程创建监听 socket 后发送给感兴趣的工作进程，由工作进程负责直接接收连接。

The second approach should, in theory, give the best performance. In practice however, distribution tends to be very unbalanced due to operating system scheduler vagaries. Loads have been observed where over 70% of all connections ended up in just two processes, out of a total of eight. 理论上第二种方法应该是效率最佳的。但在实际情况下，由于操作系统调度机制的难以捉摸，会使分发变得不稳定。可能会出现八个进程中只有两个分担了 70% 的负载。

官方支持2种方法，其实都是主进程负责监听端口，子进程会fork一个handle句柄给主线，通过循环分发或监听发送与worker进程通信，交替处理任务。

[Node.js进阶：cluster模块深入剖析](#) 端口转发这块讲的也比较细



[nodejs cluster模块分析 包磊磊的博客-CSDN博客](#) 重点端口监听相关

以上就是 cluster 的原理，总结一下就是： **cluster 模块应用 child_process 来创建子进程，子进程通过复写掉 cluster._getServer 方法，从而在 server.listen 来保证只有主进程监听端口，主子进程通过 IPC 进行通信**，其次主进程根据平台或者协议不同，应用两种不同模块（round_robin_handle.js 和 shared_handle.js）进行请求分发给子进程处理

负载均衡

[Node.js V0.12新特性之Cluster轮转法负载均衡-InfoQ](#) Node负载均衡，Node核心开发者的文章

[nodejs负载均衡（一）：服务负载均衡 - 知乎](#) 服务器集群和Node本身集群的架构

[nodejs负载均衡（二）：RPC负载均衡 - 知乎](#)

如何守护进程（PM2）

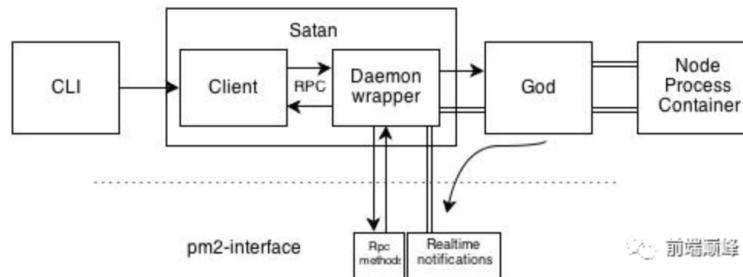
能读到这里，那么对守护进程应该有些感觉了

- 对于子进程：父进程监听子进程退出事件，适当fork新的子进程

那如果父进程崩了呢？谁来重启父进程

pm2、docker走起~

pm2 : process manager for Node apps



pm2包括 **Satan**进程、**God Deamon**守护进程、进程间的远程调用**rpc**、**cluster**等几个概念

如果不知道点西方文化，还真搞不清他的文件名为啥是 **Satan** 和 **God**：

撒旦 (Satan)，主要指《圣经》中的堕天使（也称堕天使撒旦），被看作与上帝的力量相对的邪恶、黑暗之源，是God的对立面。

1.Satan.js提供了程序的退出、杀死等方法，因此它是魔鬼；God.js 负责维护进程的正常运行，当有异常退出时能保证重启，所以它是上帝。作者这么命名，我只能说一句：oh my god。

God进程启动后一直运行，它相当于cluster中的Master进程，守护者worker进程的正常运行。

2.**rpc (Remote Procedure Call Protocol)** 是指远程过程调用，也就是说两台服务器A，B，一个应用部署在A服务器上，想要调用B服务器上应用提供的函数/方法，由于不在一个内存空间，不能直接调用，需要通过网络来表达调用的语义和传达调用的数据。同一机器不同进程间的方法调用也属于rpc的作用范畴。

3.代码中采用了**axon-rpc** 和 **axon** 两个库，基本原理是提供服务的server绑定到一个域名和端口下，调用服务的client连接端口实现rpc连接。后续新版本采用了**pm2-axon-rpc** 和 **pm2-axon**两个库，绑定的方法也由端口变成.sock文件，因为采用port可能会和现有进程的端口产生冲突。

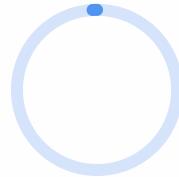
参考资料

- [Node.js的线程和进程详解 - 知乎 精彩](#)
- [进程与线程的一个简单解释 - 阮一峰的网络日志](#) 文章以工厂/车间/工人来类比进程/线程之间的关系，可作为入门理解，但真正精彩的内容在评论区
- [Node.js 进阶之进程与线程-五月君](#) 五月君的
- [浅析 Node 进程与线程 包磊磊的博客](#) 可一读
- [cluster是怎样开启多进程的，并且一个端口可以被多个 进程监听吗？ 包磊磊的博客-CSDN博客](#) 比较深入代码细节，有比较多实现demo
- [Node.js进阶：cluster模块深入剖析](#) 端口转发那一节流程图画的好
- [深入浅出 Node.js Cluster 宏观层面解释，不读代码](#)
- [淘系前端团队- 当我们谈论 cluster 时我们在谈论什么（下）](#) 比较久的文章，但依然是精华
- [Nodejs 进阶：解答 Cluster 模块的几个疑问 - 知乎- 五月君](#)
- [分享 10 道 Nodejs 进程相关 五月君](#)
- 以下偏源码
- [【nodejs原理&源码赏析（4）】深度剖析cluster模块源码与node.js多进程（上） - 大史不说话 - 博客园](#)
- [【nodejs原理&源码赏析（5）】net模块与通讯的实现 - 大史不说话 - 博客园](#)
- [Nodejs cluster模块深入探究 - SegmentFault 思否](#)
- [源码分析Node的Cluster模块-阿里云开发者社区](#)

- [通过Node.js的Cluster模块源码，深入PM2原理 - SegmentFault 思否](#)
- [句柄是什么？ - 知乎 读端口转发时辅助理解](#)
- [子进程 | Node.js 中文文档 | Node.js 中文网](#)
- [浅析 NodeJS 多进程和集群](#)
- [deep-into-node/chapter4-1.md at master · yjhjstz/deep-into-node · GitHub](#)

personal blog

合抱之木，生于毫末；九层之台，起于累土；千里之行，始于足下 ———《老子》



Browser

- [【浏览器原理-01】宏观下的浏览器](#)
- [【浏览器原理-02】浏览器中的JS执行机制](#)
- [【浏览器原理-03】V8 工作原理](#)
- [【浏览器原理-04】页面循环系统](#)

Javascript

- [ECMAScript 2016 2017 2018 新特性](#)
- [根据Promise/A+规范实现 Promise](#)
- [异步（一）：Promise深入理解与实例分析](#)
- [异步（二）：Generator（生成器）深入理解](#)
- [异步（三）：Async/await](#)
- [JS文件：读取与拖拽、转换bsae64、预览、FormData上传、七牛上传、分割文件](#)
- [JS事件：捕获与冒泡、事件处理程序、事件对象、跨浏览器、事件委托](#)
- [深入原型和原型链：彻底捋清prototype和proto](#)
- [深入继承：一步步捋清五种继承方式](#)
- [彻底捋清楚 instanceof](#)
- [彻底捋清楚 new 的实现](#)
- [模拟实现call、apply、bind](#)
- [Object.defineProperty\(\) 和简易双向绑定原理](#)
- [变量对象（真正理解何为提升）](#)
- [this](#)
- [You don't konw JavaScript => 闭包](#)
- [深入理解JS的类型、值、类型转换](#)
- [同一个函数形成的多个闭包的值都是相互独立的](#)
- [函数的内部属性和方法（arguments、callee）](#)

- [执行上下文](#)
- [JavaScript的参数传递\(引用类型\)](#)
- [map、forEach、filter、reduce](#)
- [一道综合面试题 \(原型、this、作用域、构造函数、运算符优先级\)](#)
- [JS模块化: CommonJS、AMD、CMD、UMD、ES6 Module](#)
- [ES6中export及export default、Node中exports和module.exports的区别](#)

Network、Web performance、Business Scenario

- 浏览器工作原理: 从输入URL到页面加载完成
- 事件循环机制 Event-Loop及其延伸
- [Web Performance Optimizations](#)
- [【性能优化】DNS预解析、域名发散、域名收敛](#)
- [【性能优化】优化关键渲染路径，加速浏览器首次渲染](#)
- [【性能优化】手把手实现图片懒加载](#)
- [【性能优化】图片优化——总览](#)
- [缓存（一）——缓存总览: 从性能优化的角度看缓存](#)
- [缓存（二）——浏览器缓存机制: 强缓存、协商缓存](#)
- [缓存（三）——数据存储: cookie、Storage、indexedDB](#)
- [缓存（四）——离线应用缓存: App Cache => Manifest](#)
- [深入了解HTTP/2的前世今生以及Web性能优化总结](#)
- [GET和POST: 辩证看100 continue, 以及最根本区别](#)
- [cookie实践 \(从搭建服务器到cookie操作全流程\)](#)
- [TCP概览](#)
- [TCP与UDP对比](#)
- [CORS 简单请求+预检请求 \(理解HTTP跨域原理\)](#)
- [前端跨域常用方法](#)
- [canvas+vue实现60帧每秒的抢金币动画 \(类天猫红包雨\)](#)
- [canvas合成图片海报、分享, 小坑记录](#)
- [前端曝光数据埋点——IntersectionObserver+vue指令](#)
- [Intersection Observer + Vue指令 优雅实现图片懒加载](#)

Framework

- vue
 - [转Vue 3前再读一次Vue2源码](#)
 - [Vue 响应式原理](#)
 - [Vue 源码阅读【1】—— Vue本质](#)
 - [Vue 源码阅读【2】—— 数据驱动](#)
 - [【译】VueJS 最佳实践](#)
- react
 - [理解后端渲染、CSR\(客户端渲染\)、SSR \(服务端渲染\) 的核心原理及区别](#)
 - [Next.js 生命周期理解 -> 流程图](#)
 - [Build a react+react-router ssr App from the ground up](#)
 - [SPA路由原理+build your own react router v4](#)
 - [React Native 性能优化总结 \(持续更新。。。\)](#)
 - [React+RN开发过程中的一些问题总结 \(持续更新。。。\)](#)
 - [React-Native 从零搭建App\(长文\)](#)
 - [React源码分析：组件实现 \(基于版本16\)](#)

- [React源码分析: setState](#)
- [Redux源码分析](#)

Webpack、Engineering

- 应用系列
 - [【webpack应用系列\(一\)】从零搭建vue开发环境](#)
 - [【webpack应用系列\(二\)】从零搭建vue生产环境](#)
 - [【webpack应用系列\(三\)】webpack性能优化](#)
 - [【规范】Eslint+Prettier+git hook -> make your life easier](#)
- 源码进阶系列
 - [【webpack进阶系列\(一\)】手撸一个mini-webpack\(1\): 分析收集依赖](#)
 - [【webpack进阶系列\(二\)】手撸一个mini-webpack\(2\): 打包依赖代码](#)
 - [【webpack进阶系列\(三\)】webpack插件骨架-Tapable](#)
 - [【webpack进阶系列\(四\)】webpack核心构建原理\(vscode断点调试源码\)](#)
 - [【webpack进阶系列\(五\)】构建module流程](#)
 - [【webpack进阶系列\(六\)】seal封装生成文件核心流程](#)
 - [【webpack进阶系列\(七\)】写个plugin](#)
 - [【webpack进阶系列\(八\)】写个loader](#)

Node、DevOps、Linux

- Node
 - [Node 模块机制之require源码](#)
 - [Node 多进程与多线程、cluster](#)
- DevOps
 - [Vue SSR 持续集成部署 \(CI/CD\) : Travis + PM2 自动化部署 Node 应用](#)
 - [买了云服务器后，第一步要干嘛。。。？](#)
 - [Linux基础之文件权限](#)
 - [Vim 基础操作](#)
 - [Docker基础认识](#)
 - [Docker 运行Node.js应用](#)
- other
 - [mongoDB基础](#)
 - [快速上手node+express+MongoDB](#)
 - [Express + create-react-app 快速构建前后端开发环境](#)

Javascript Design Pattern

- [随手实现一个订阅-发布模式](#)
- [代理模式+ES6 Proxy](#)
- [装饰者模式+ES7 Decorator](#)
- [单例模式](#)

Algorithm

- [《学习JavaScript数据结构与算法》读书笔记](#)
- [JS实现基础数据结构\(栈、队列、单双链表、二叉搜索树\)](#)

- [JS实现基础排序和搜索算法](#)

HTML、CSS

- [BFC \(块级格式上下文\)](#)
- [三栏布局-七种实现](#)

【浏览器原理 - 01】宏观下的浏览器

#develop/Front-End/浏览器原理

本系列文章，是学习[浏览器工作原理与实践](#)[浏览器V8原理-极客时间](#)的总结和扩展，部分内容和图片都来自于这门课程，先做个说明，算是部分转载

1、Chrome的多进程架构

[01 | Chrome架构：仅仅打开了1个页面，为什么有4个进程？-极客时间](#)

先从一个问题开始，开一个chrome页面，会启动多少个进程？

打开一个最简单的无痕浏览器，关闭所有插件，只开一个tab访问google首页，然后从【右上角选项 -> 更多工具 -> 任务管理器】，打开chrome的任务管理器窗口，可以看到有4个进程



The screenshot shows the Windows Task Manager window titled "任务管理器 - Google Chrome". It displays a list of processes with columns for Task, Memory Usage, CPU Usage, Network, and Process ID. There are four entries listed:

任务	内存占用空间	CPU	网络	进程 ID
浏览器	141 MB	4.7	0	2928
GPU 进程	67.7 MB	0.7	0	2939
实用程序: Network Service	25.3 MB	0.6	0	2942
隐身标签页: Google	35.6 MB	0.7	0	3026

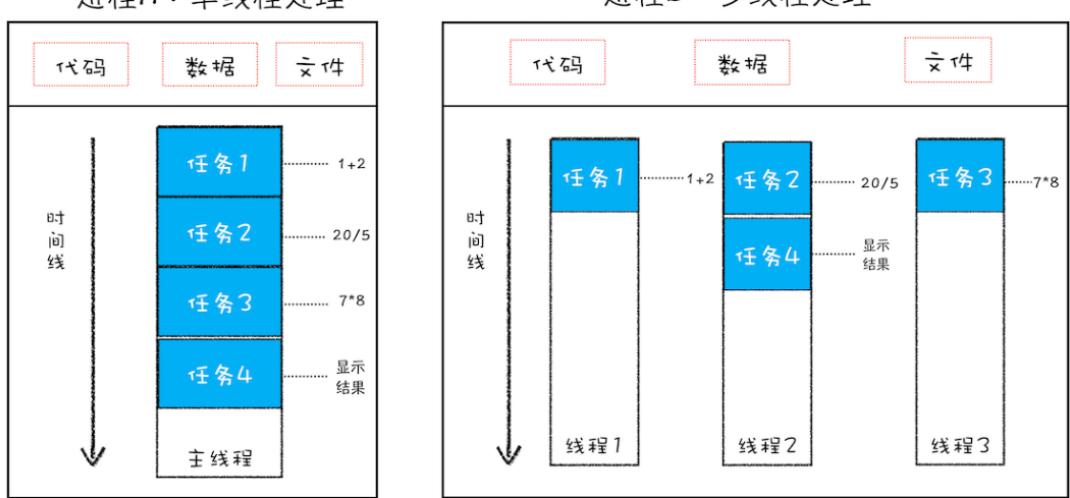
在聊4个进程之前，先来了解一下进程和线程的概念

1.1 进程和线程

一个进程就是一个程序的运行实例。详细解释就是，启动一个程序的时候，操作系统会为该程序创建一块内存，用来存放代码、运行中的数据和一个执行任务的主线程，我们把这样的一个运行环境叫进程。

线程是依附于进程的，并不能单独存在，由进程来启动和管理

进程分为单线程和多线程，如下图理解，进程中使用多线程并行处理能提升运算效率



单线程与多线程的进程对比图

线程和进程之间的关系有4个特点

- 1、进程中的任意一线程执行出错，都会导致整个进程的崩溃
- 2、线程之间共享进程中的数据
- 3、当一个进程关闭之后，操作系统会回收进程所占用的内存
- 4、进程之间的内容相互隔离

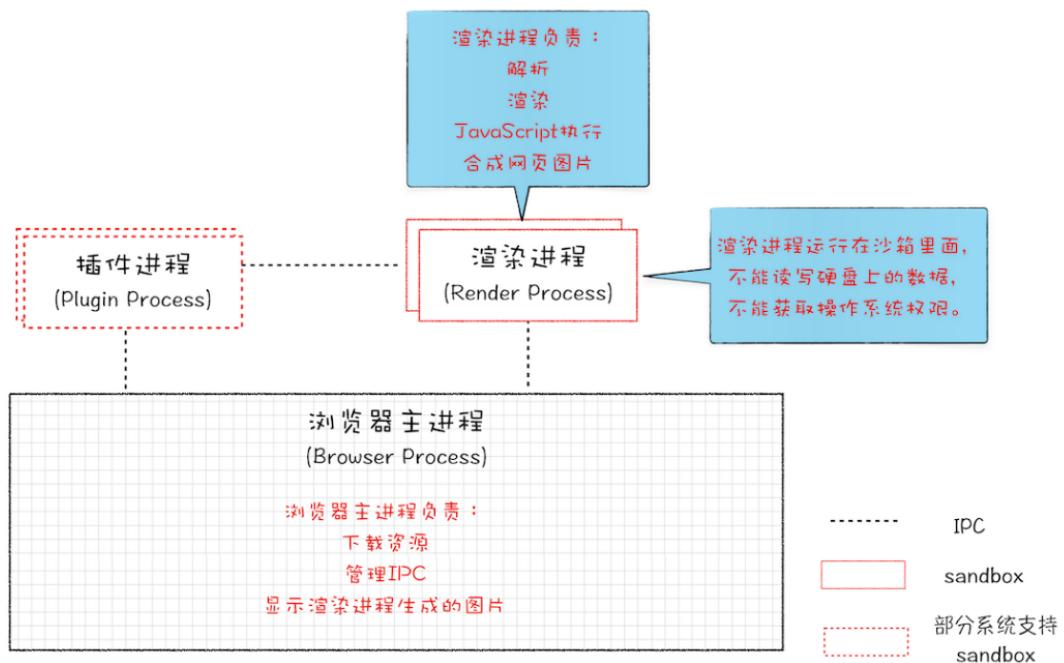
1.2 单进程浏览器 -> 多进程浏览器

单进程浏览器是指浏览器的所有功能模块都是运行在同一个进程中，这些模块包含了网络、插件、JavaScript 运行环境、渲染引擎和页面等。

2008年chrome浏览器面世之前，市面上的浏览器都是单进程的，单进程浏览器有几个缺点

- 1、**不稳定**，插件或者渲染引擎（这两货极其不稳定）只要有一个崩溃，整个浏览器就崩溃了
- 2、**不流畅**，如果某个插件或者脚本执行非常慢，或者内存泄露，那么整个浏览器都会卡顿
- 3、**不安全**，还是插件和脚本，无论是C/C++插件获取操作系统资源还是JS脚本通过浏览器漏洞获取系统权限，都能引发安全问题

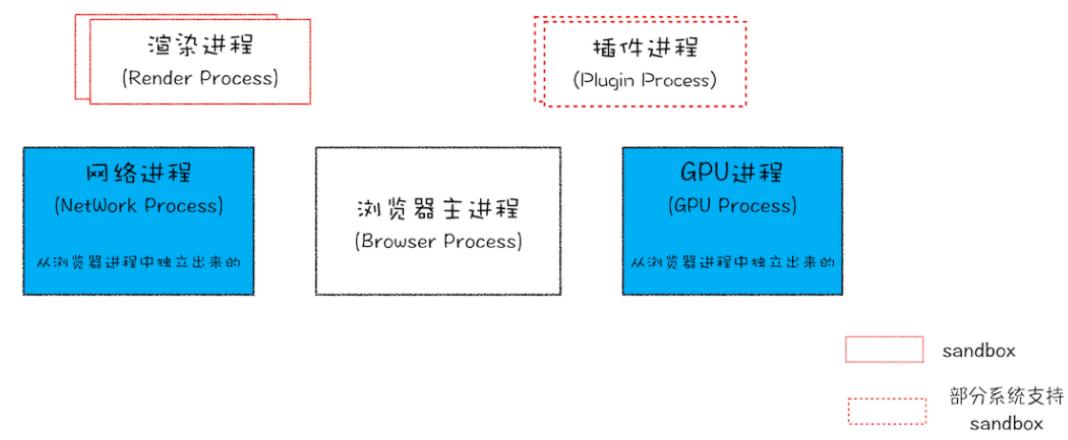
08年 chrome带来了多进程浏览器架构



早期 Chrome 进程架构图

插件和渲染使用独立的进程，进程之间是互相隔离的安全沙箱环境，上面提到的3个问题都迎刃而解

现代浏览器基于以上多进程架构有了一定的扩展和变化



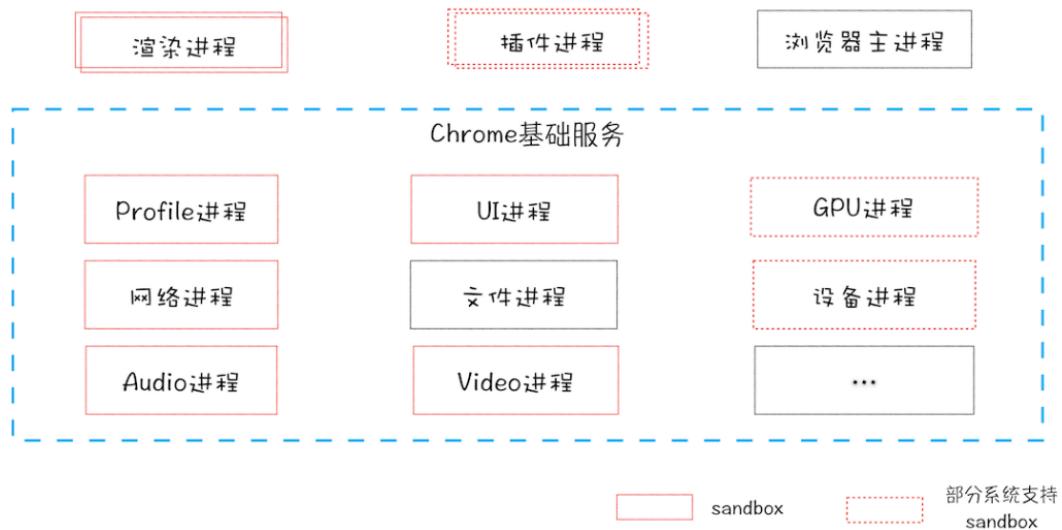
最新的 Chrome 进程架构图

核心的进程有

- **浏览器进程**, 主要负责界面显示、用户交互、子进程管理, 同时提供存储等功能
- **渲染进程** (多个, 暂时理解为一个tab页面一个渲染进程), 排版引擎 Blink 和 JavaScript 引擎 V8 都是运行在该进程中
- **GPU进程**, 网页、Chrome 的 UI 界面都选择采用 GPU 来绘制
- **网络进程**, 主要负责页面的网络资源加载
- **插件进程** (多个), 插件容易崩溃, 最是需要独立隔离

1.3 未来面向服务的架构

Chrome 整体架构会朝向现代操作系统所采用的“面向服务的架构”方向发展，原来的各种模块会被重构为独立的服务（Service），每个服务（Service）都可以在独立的进程中运行，访问服务（Service）必须使用定义好的接口，通过 IPC 来通信，从而构建一个更内聚、松耦合、易于维护和扩展的系统，更好实现 Chrome 简单、稳定、高速、安全的目标。



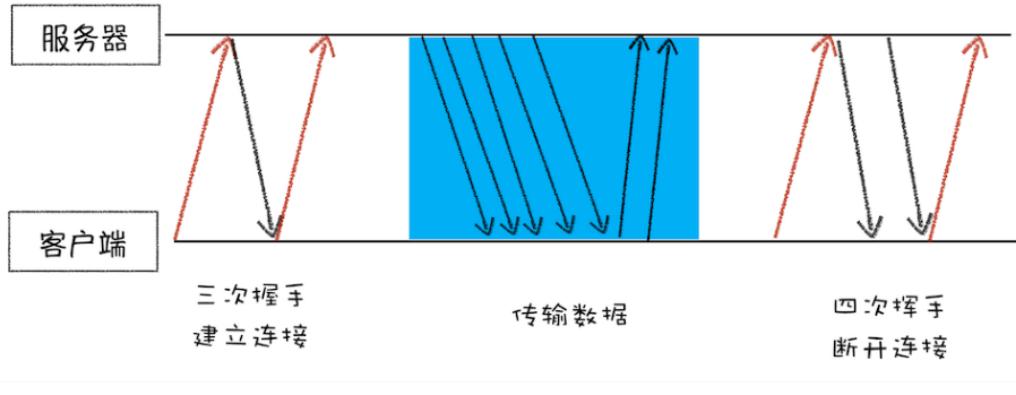
Chrome“面向服务的架构”进程模型图

2、TCP协议：如何保证页面文件能被完整送达浏览器？

[02 | TCP协议：如何保证页面文件能被完整送达浏览器？-极客时间](#)

TCP (Transmission Control Protocol, 传输控制协议) 是一种面向连接的、可靠的、基于字节流的传输层通信协议。

- 互联网中的数据是通过数据包来传输的，数据包在传输过程中容易丢失或出错。
- IP用来寻址，寻找到对应计算机的地址，负责把数据包送达目的主机
- UDP通过端口号来寻找对应的处理程序，负责把数据包送达具体应用（通过端口号），但会丢包
- TCP 引入了重传机制和数据排序机制来保证数据完整地传输，TCP连接分为三个阶段：建立连接（三次握手）、传输数据和断开连接（四次挥手），但为了保证数据传输的可靠性，牺牲了数据包的传输速度



一个 TCP 连接的生命周期

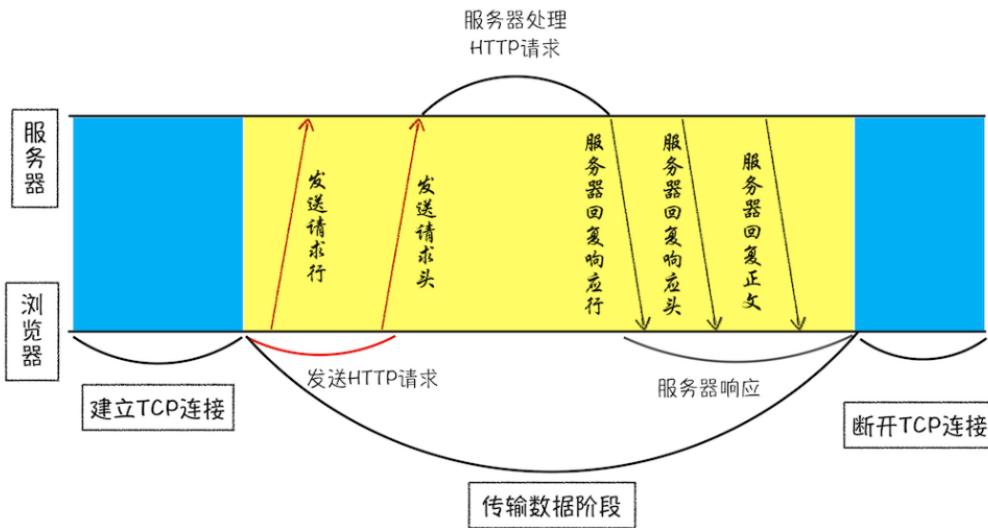
几个小问题

- HTTP和TCP的区别
- HTTP协议和TCP协议都是TCP/IP协议簇的子集。
- HTTP协议属于应用层，TCP协议属于传输层，HTTP协议位于TCP协议的上层
- http 和 websocket的区别
- 都是应用层协议，而且websocket名字取的比较有迷惑性，其实和socket完全不一样，可以把websocket看出是http的改造版本，增加了服务器向客户端主动发送消息的能力

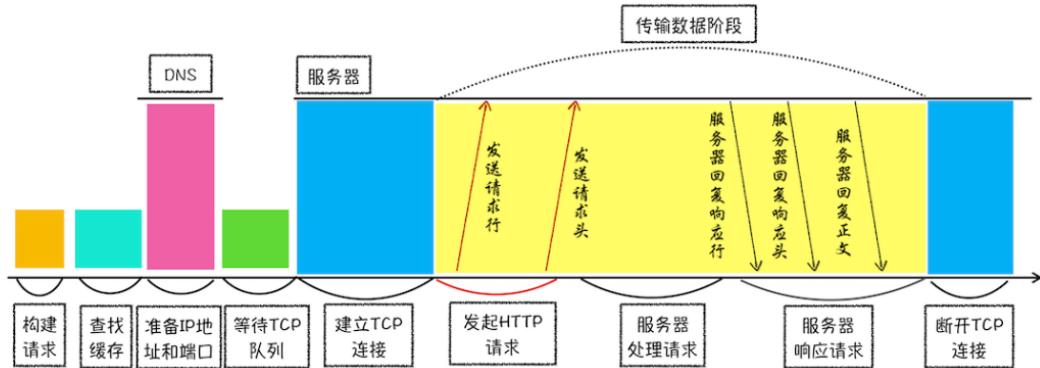
3 HTTP请求流程

[03 | HTTP请求流程：为什么很多站点第二次打开速度会很快？-极客时间](#)

HTTP 的内容是通过 TCP 的传输数据阶段来实现的



TCP 和 HTTP 的关系示意图



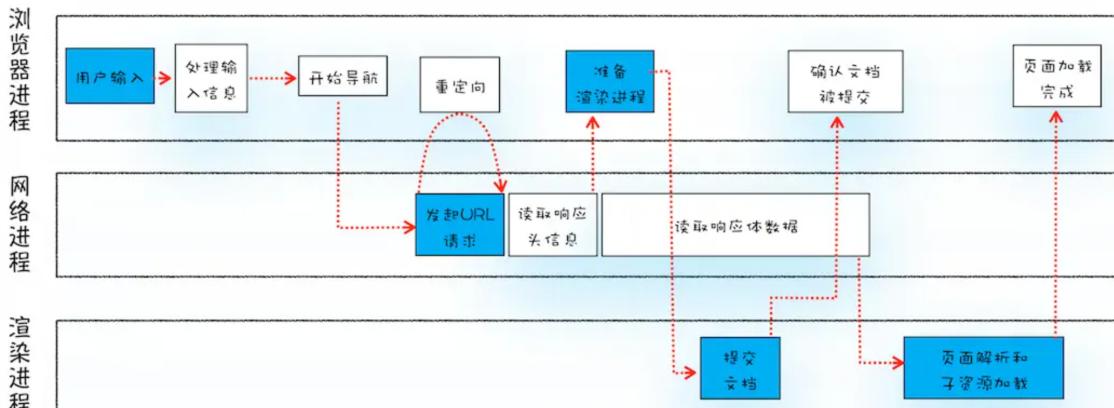
HTTP 请求流程示意图

同一个域名同时最多只能建立 6 个 TCP 连接，不光是指Ajax，还包括页面中的资源加载，只要是一个域名下的资源，浏览器同一时刻最多只支持6个并行请求

但如果换成http2，可以并行请求资源，但浏览器只会为每个域名维护一个TCP连接

4 导航流程：从输入URL到页面展示，这中间发生了什么？

[GitHub - Alex/what-happens-when: An attempt to answer the age old interview question "What happens when you type google.com into your browser and press enter?"](https://github.com/Alex/what-happens-when) 这个太详尽了 不太建议看
哈哈



4.1 用户输入

用户输入URL并回车，地址栏判断是搜索内容还是请求URL

- 搜索内容，使用默认搜索引擎，拼接成URL再往下走
- 请求URL合法，（如果没有监听beforeUnload事件或者同意往下走），加上协议头，浏览器主进程通过 IPC (进程间通信) 将 URL 请求传输给网络进程，如 `GET /index.html HTTP1.1`

这时候，标签页上的图标以及进入了加载状态，但当前页面显示的依然是之前打开的页面，当前属于**等待提交文档**的阶段

在浏览器中输入地址后页面没有马上消失，这一步是触发当前页的卸载事件和收集需要释放的内存，占用一些时间，大头是请求新的url的返回

4.2 URL请求过程

- 网络进程检查本地缓存，如果有缓存资源，直接 200 返回资源（from memory 或者 from disk）给浏览器进程；如果没有缓存资源，进入网络请求流程
- 进行DNS解析（本地hosts -> DNS域名解析服务器），获取 IP 地址，如果是 HTTPS 请求，先建立 TLS 连接
- 根据 IP 地址与服务器建立 TCP 连接，浏览器端构建请求行、请求头、附加 Cookie 信息，向服务器发送构建的请求信息
- 服务器收到请求信息后，根据请求信息生成响应数据，发给网络进程；网络进程接收到响应行和响应头后，开始解析响应头的内容
- 网络进程解析响应头，如果是301(永久)/302(临时)，网络进程从 Location 字段读取重定向地址，重新发起 HTTP(S) 请求，一切重头开始
- 分析下载类型 `Content-Type`，判断响应体的数据类型，如果碰到 `application/octet-stream` 这种字节流类型，则按照下载类型来处理，将请求提交给浏览器的下载管理器，当前URL的导航流程结束；`text/html` 则是 HTML格式，继续往下走，准备渲染流程

这里补充 Chrome 的一个机制，同一个域名同时最多只能建立 6 个TCP连接(如果只有一个TCP连接 那就串行了 100张图片会炸，但TCP连接太多，服务器又吃不消)，如果在同一个域名下同时有 10 个请求发生，那么其中 4 个请求会进入排队等待状态，直至进行中的请求完成。如果当前请求数量少于6个，会直接建立TCP连接。

4.3 分配渲染进程

- 默认情况下，Chrome会为每个页面分配一个新的渲染进程，但也有例外，如果打开的新页面和当前页面属于同一站点，那么新页面会复用父页面的渲染进程，官方策略叫 `process-per-site-instance`

4.4 提交文档：将网络进程接收到的HTML数据提交给渲染进程

- 当浏览器进程收到网络进程的响应头数据后，向渲染进程发起CommitNavigation 【提交文档】的消息，会携带响应头等消息
- 渲染进程收到消息后，开始准备接收HTML数据，和网络进程建立传输数据的管道
- 文档数据（响应体数据，第一批html数据，无须等里面的JS和CSS）传输完成后，渲染进程会返回 【确认提交】的消息给浏览器进程 [Inside look at modern web browser \(part 2\) | Google Developers](#)
- 浏览器进程收到确认的消息后，更新浏览器界面状态，包括安全状态、地址栏URL、前进后退历史，更新当前页面，此时页面是空白的

4.5 渲染阶段

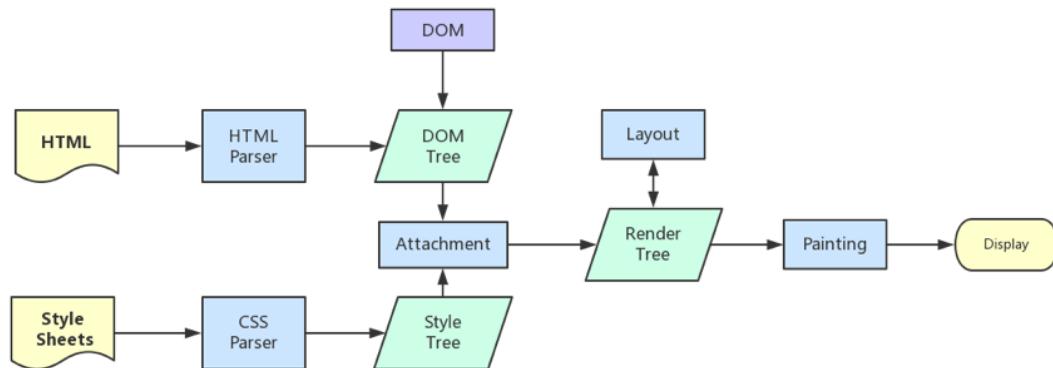
- 文档提交后，渲染进程开始页面解析和子资源加载（下一节），一旦页面生成完成，渲染进程发送消息给浏览器进程，停止标签页上的loading加载动画

5 HTML、CSS、JS的渲染流程

[浏览器渲染详细过程：重绘、重排和 composite 只是冰山一角 - 前端 - 掘金](#) 辅助理解
[浏览器重绘\(repaint\)重排\(reflow\)与优化浏览器机制 - 掘金](#)

5.1 渲染前

- 1、构建DOM树：渲染引擎将HTML解析为浏览器可以理解的DOM
- 2、样式计算（Recalculate Style）：根据CSS样式表，计算出DOM树所有节点的样式
- 3、构建布局树（Render Tree）：计算每个元素的几何坐标位置，并将信息保存在布局树中



一个小问题，加深理解

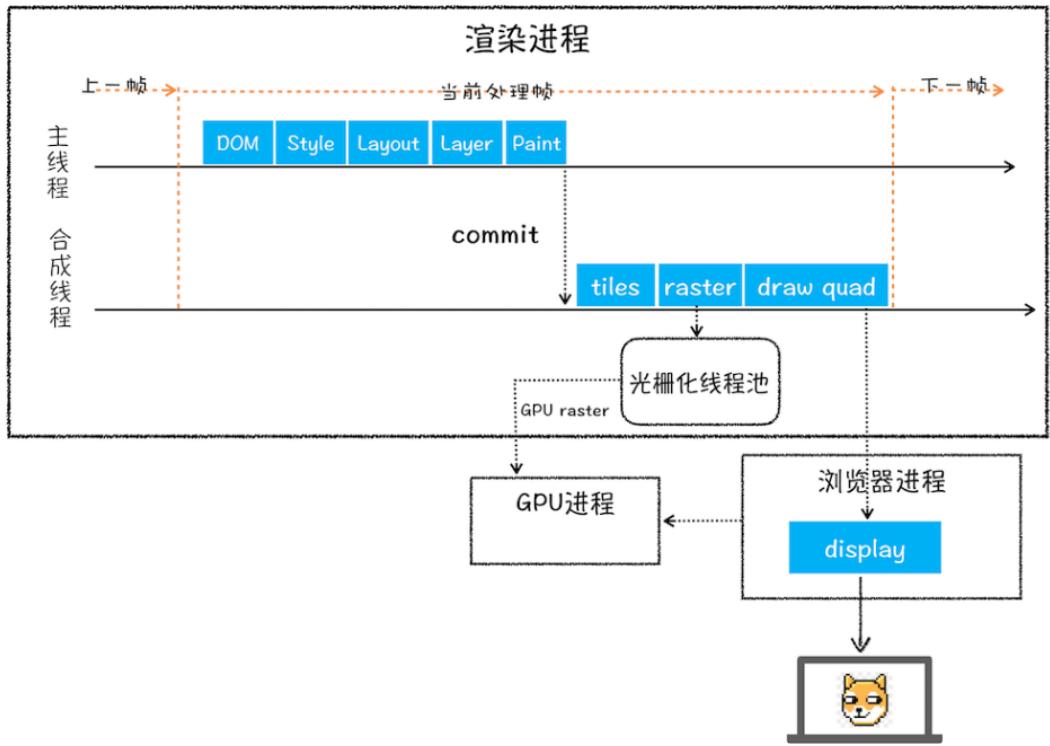
1、如果下载CSS文件阻塞了，会阻塞DOM树的合成吗？会阻塞页面的显示吗？

个人理解：外链CSS文件加载（单独的网络进程下载）不会阻塞DOM树解析，但会阻塞Render树渲染，但现代浏览器会有超时限制，可以直接用user-agent的默认样式，因为HTML本身也是有语义化的

5.2 渲染

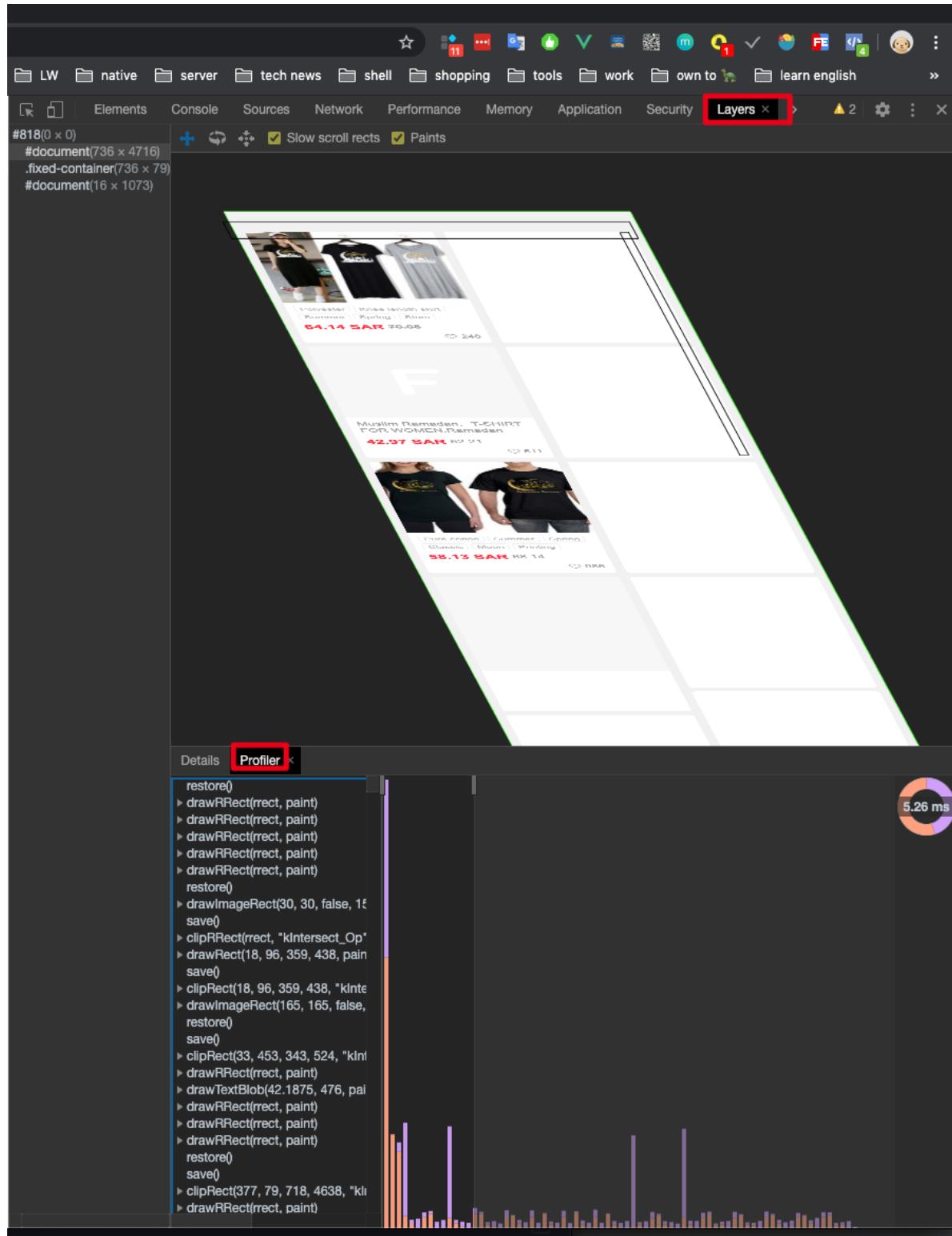
接下来是真正的渲染过程

- 4、分层：对布局树进行分层生成分层树
- 5、图层绘制：为每个图层生成绘制列表，并提交到合成线程
- 6、栅格化（raster）：绘制列表只是用来记录绘制顺序和绘制指令的列表，而实际上绘制操作是由渲染引擎中的合成线程来完成的，这一步由合成线程将图层分成图块，在光栅化线程池中转为位图
- 7、合成与显示：合成线程发送绘制图块命令 DrawQuad 给浏览器进程，合成和显示，浏览器进程根据 DrawQuad 消息生成页面，并显示到显示器上



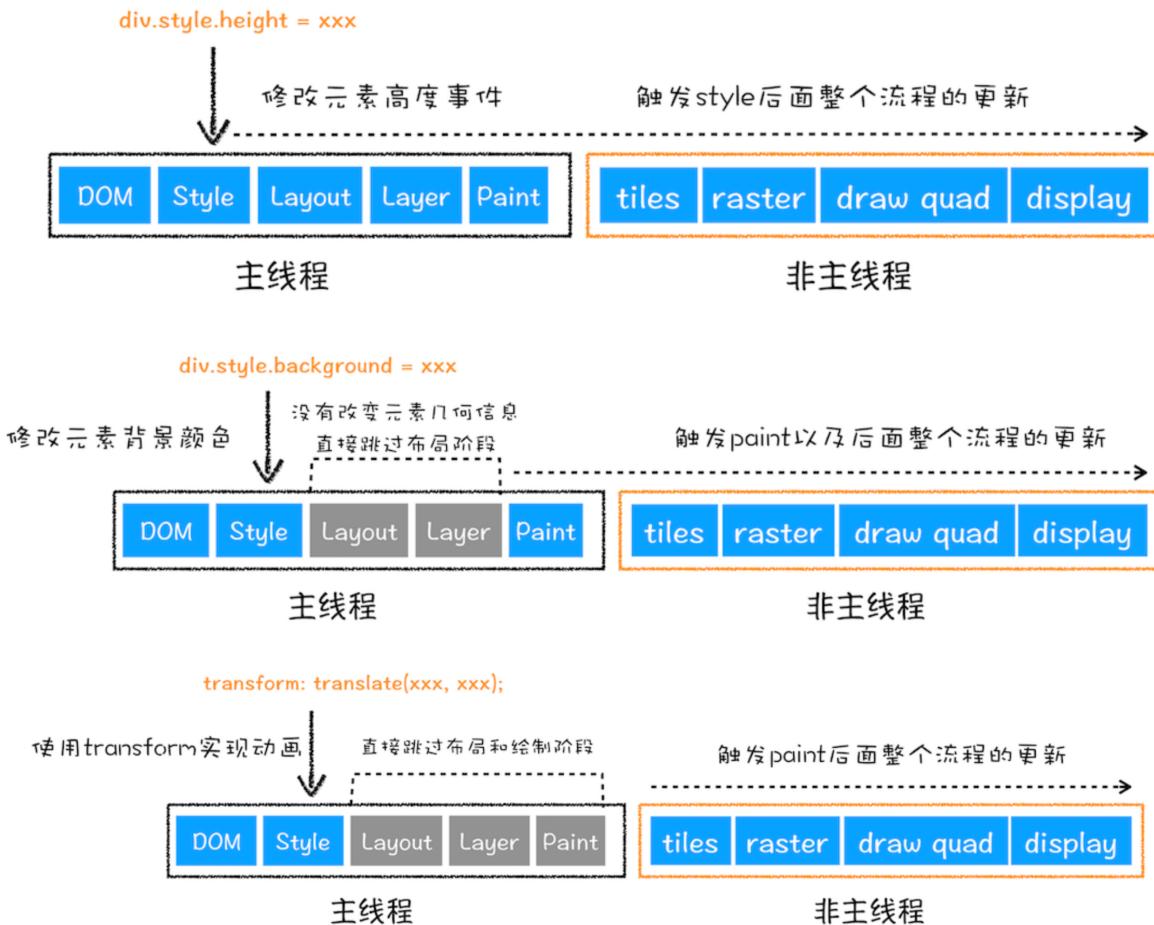
完整的渲染流水线示意图

浏览器Layers标签中可以看到详细的绘制过程



5.3 重排、重绘、合成优化

- 重绘跳过了布局和分层阶段，执行效率会比重排稍高一些
- 直接合成效率最高，直接跳过布局和绘制，只执行后续的合成操作，比如CSS的transform动画不占用主线程上资源



一些优化手段

- DOM样式读写分离

```
// 这样会触发2次重排+重绘
div.style.left = '10px';
div.style.top = '10px';
console.log(div.offsetLeft);
console.log(div.offsetTop);

// 前面两个写操作的渲染队列已被清空, console操作不会触发重排
div.style.left = '10px';
console.log(div.offsetLeft);
div.style.top = '10px';
console.log(div.offsetTop);
```

- 样式集中改变, 用class集中一次改变样式
- 避免使用table布局
- 缓存布局信息
- 离线改变dom -> 进阶版使用虚拟dom -> vue/react
 - 先通过 `display: none` 隐藏元素, 修改完后再恢复
 - 使用[DocumentFragment - Web API 接口参考 | MDN](#)先创建文档水平, 操作完后再添加到文档, 只会触发一次重排
 - 复制节点, 在副本上操作, 再替换
- 固定或者绝对定位, `position: fixed/absolute`
- Debounce window resize事件

- 动画，启用GPU加速
 - Canvas2D、CSS3转换 transition、3D变换 transform、WebGL、视频
- will-change 提前告诉浏览器需要做优化的元素

【浏览器原理-02】浏览器中的JS执行机制

1、变量提升

[变量对象 \(真正理解何为提升\) · Issue #7 · amandakelake/blog · GitHub](#)

JS代码执行

- 编译阶段
 - 变量和函数被存放到**变量环境中**，变量默认值为 `undefined`，函数名只是一个引用，函数实体是保存到堆中
- 代码执行阶段
 - 从变量环境中查找变量和函数，并执行赋值操作

2、调用栈

- 每调用一个函数，JavaScript 引擎会为其创建执行上下文，并把该执行上下文压入调用栈，然后 JavaScript 引擎开始执行函数代码
- 如果在一个函数 A 中调用了另外一个函数 B，那么 JavaScript 引擎会为 B 函数创建执行上下文，并将 B 函数的执行上下文压入栈顶
- 当前函数执行完毕后，JavaScript 引擎会将该函数的执行上下文弹出栈
- 当分配的调用栈空间被占满时，会引发“堆栈溢出”问题

调用栈有两个指标，每个平台不一样

- 最大栈容量
- 最大调用深度

对于闭包来说，执行上下文已经没了，不过内部函数引用的变量还保存在堆上，所以不影响操作

```
function runStack (n) {
  if (n === 0) return 100;
  return runStack(n - 2);
}
runStack(50000)
```

对于栈溢出的问题，有以下优化方法

- 循环，不使用递归函数就不存在堆栈溢出

```
function runStack(n) {
  while (true) {
    if (n === 0) {
      return 100;
    }

    if (n === 1) { // 防止陷入死循环
      return 200;
    }
  }
}
```

```
n = n - 2;
}
}
console.log(runStack(50000));
```

- 通过异步执行，不进栈

```
function runStack (n) {
  if (n === 0) return 100;
  return setTimeout(function(){runStack( n- 2)},0);
}
runStack(50000)
```

- 考虑尾递归 [尾调用优化 - 阮一峰的网络日志](#)

- 尾调用由于是函数的最后一步操作，所以不需要保留外层函数的调用记录，因为调用位置、内部变量等信息都不会再用到了，只要直接用内层函数的调用记录，取代外层函数的调用记录就可以了
- 下方代码 -> “尾调用优化” (Tail call optimization)，即只保留内层函数的调用记录。如果有所有函数都是尾调用，那么完全可以做到每次执行时，调用记录只有一项，这将大大节省内存。这就是“尾调用优化”的意义。

```
function g() {
}

function f() {
  let m = 1;
  let n = 2;
  return g(m + n);
}
f();

// 等同于
function f() {
  return g(3);
}
f();

// 等同于
g(3);
```

3、var缺陷，为何引入const/let -> 变量提升带来的变量覆盖、污染

作用域就是变量与函数的可访问范围，即作用域控制着变量和函数的可见性和生命周期

ES6之前，ES只有两种作用域

- 全局作用域：这里的对象在代码的任何地方都能访问，生命周期伴随着页面的生命周期
- 函数作用域：函数内部定义的变量或者函数，只能在函数内部被访问，函数执行之后，内部定义的变量会被销毁（闭包暂不讨论）

ES6引入了块级作用域

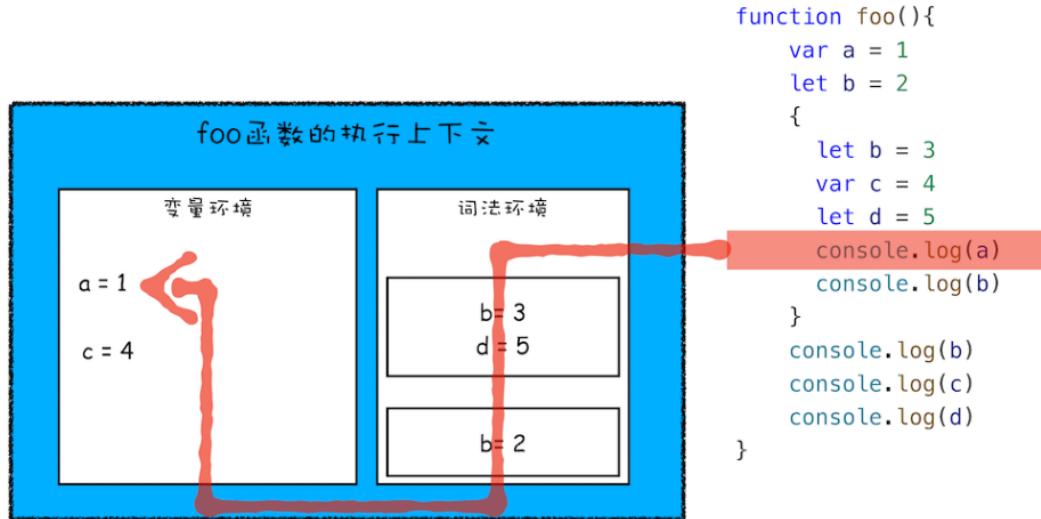
- 函数内部通过 var 声明的变量，在编译阶段全都被存放到变量环境里面
- 通过 let 声明的变量，在编译阶段会被存放到词法环境（Lexical Environment）中

块级作用域就是通过词法环境的栈结构来实现的

而变量提升是通过变量环境来实现，

通过这两者的结合，JavaScript 引擎也就同时支持了变量提升和块级作用域了

一个变量的查找过程：沿着词法环境的栈顶向下查询，如果在词法环境中的某个块中查找到了，就直接返回给 JavaScript 引擎，如果没有查找到，那么继续在变量环境中查找



- Var的创建和初始化被提升，赋值不会被提升。
- Let的创建被提升，初始化(uninitialized)和赋值不会被提升。
- Function的创建、初始化和赋值均会被提升。

Let的暂时性死区是因为V8虚拟机做了限制，虽然a在内存中，但是当你在let a 之前访问a时，根据 ECMAScript定义，虚拟机会阻止的访问

```

function test() {
  console.log(a)
  let a = 7;
}
test();

```

```

> function test(){
  console.log(a)
  let a = 7;
}
test()
✖ > Uncaught ReferenceError: Cannot access 'a' before initialization
  at test (<anonymous>:2:17)
  at <anonymous>:5:1
>

```

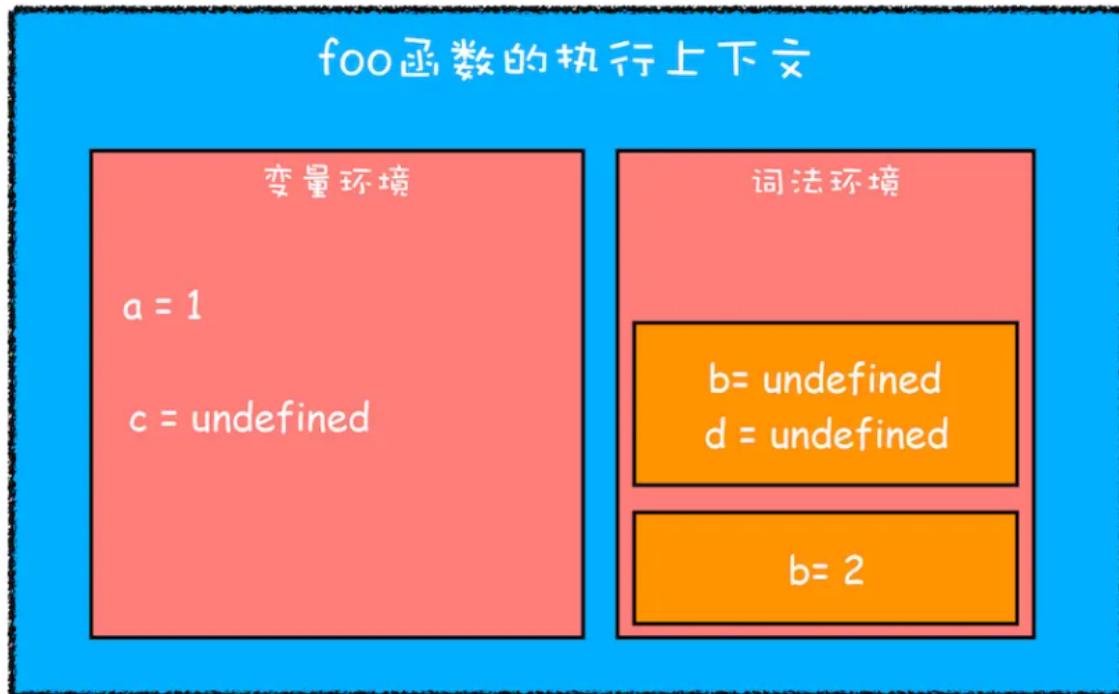
既然聊到了作用域，那最后我们再简单聊下编程语言吧。经常有人争论什么编程语言是世界上最好的语言，但如果站在语言本身来说，我觉得这种争论没有意义，因为语言是工具，而工具是用来创造价值的，至于能否创造价值或创造多大价值不完全由语言本身的特性决定。这么说吧，即便一门设计不那么好的语言，它也可能拥有非常好的生态，比如有完善的框架、非常多的落地应用，又或者能够给开发者带来更多的回报，这些都是评判因素。

如果站在语言层面来谈，每种语言其实都是在相互借鉴对方的优势，协同进化，比如 JavaScript 引进了块级作用域、迭代器和协程，其底层虚拟机的实现和 Java、Python 又是非常相似，也就是说如果你理解了 JavaScript 协程和 JavaScript 中的虚拟机，其实你也就理解了 Java、Python 中的协程和虚拟机的实现机制。

所以说，语言本身好坏不重要，重要的是能为开发者创造价值。

4、作用域链和闭包

变量提升是通过变量环境来实现，而块级作用域就是通过词法环境的栈结构来实现的，通过这两者的结合，JavaScript 引擎也就同时支持了变量提升和块级作用域了



5、this

[this · Issue #5 · amandakelake/blog · GitHub](#)

作用域链和 `this` 是两套不同的系统，它们之间没太多联系

【浏览器原理-03】V8 工作原理

1、内存机制

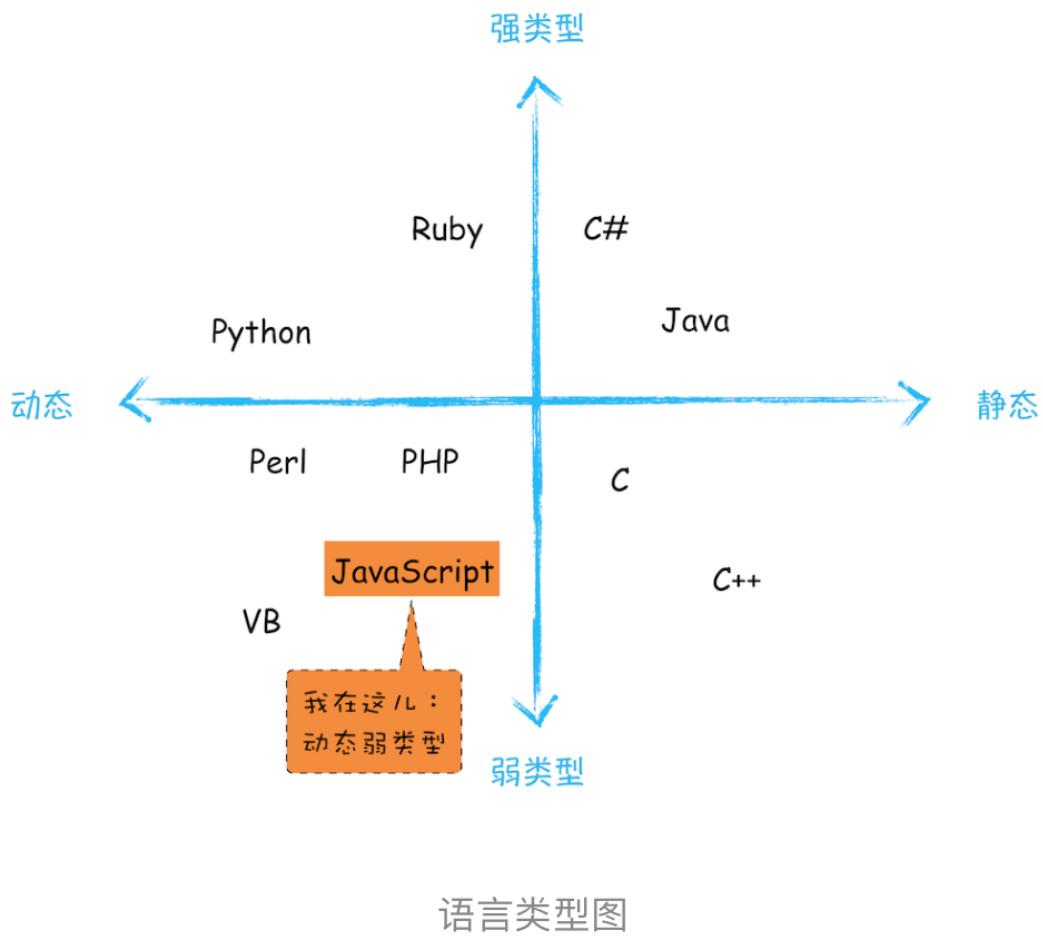
1.1 语言类型

静态语言：使用前就需要确认变量数据类型

动态语言：运行过程中需要检查数据类型

弱类型：支持隐式类型替换

强类型：不支持隐式类型替换

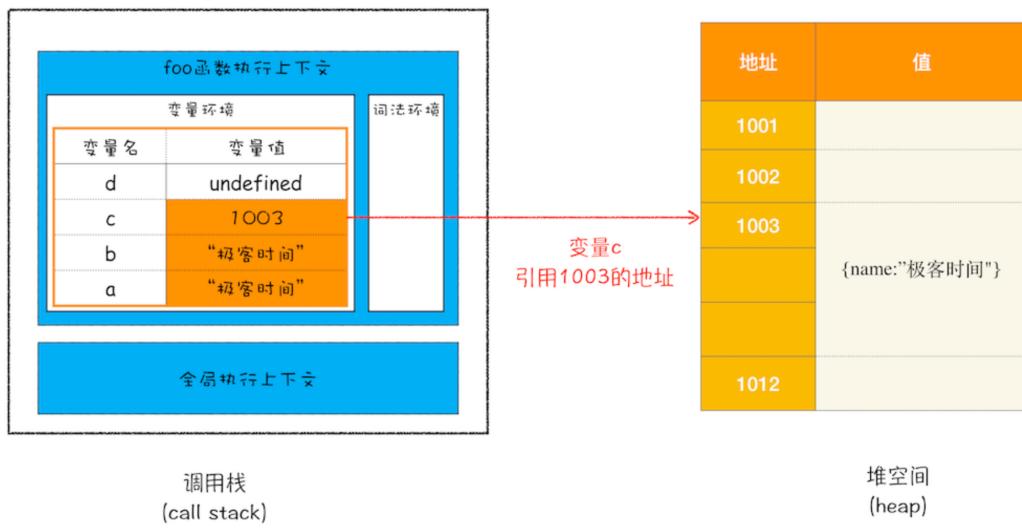


Javascript是动态、弱类型语言

1.2 内存空间

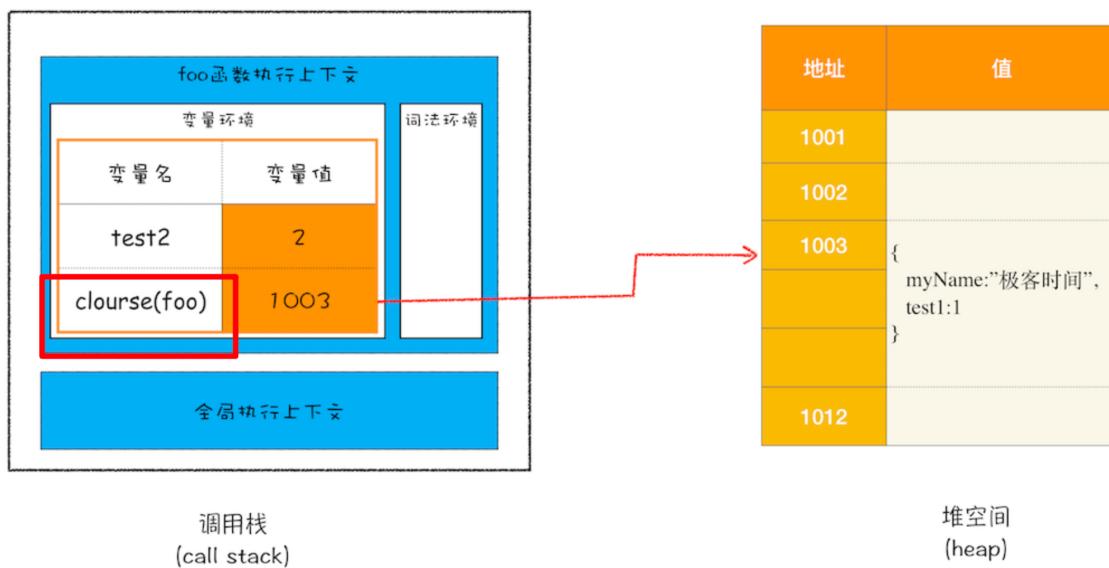
JS执行过程中，主要有3种类型的内存空间

- 代码空间
- 栈空间（一般不大）
- 调用栈，存储执行上下文
- 原始类型数据
- 堆空间（很大）
- 引用类型



闭包的产生

JS引擎预扫描内部函数，判断为闭包后，在堆空间创建一个closure对象，用来保存外部变量，栈空间中保存着对闭包的引用

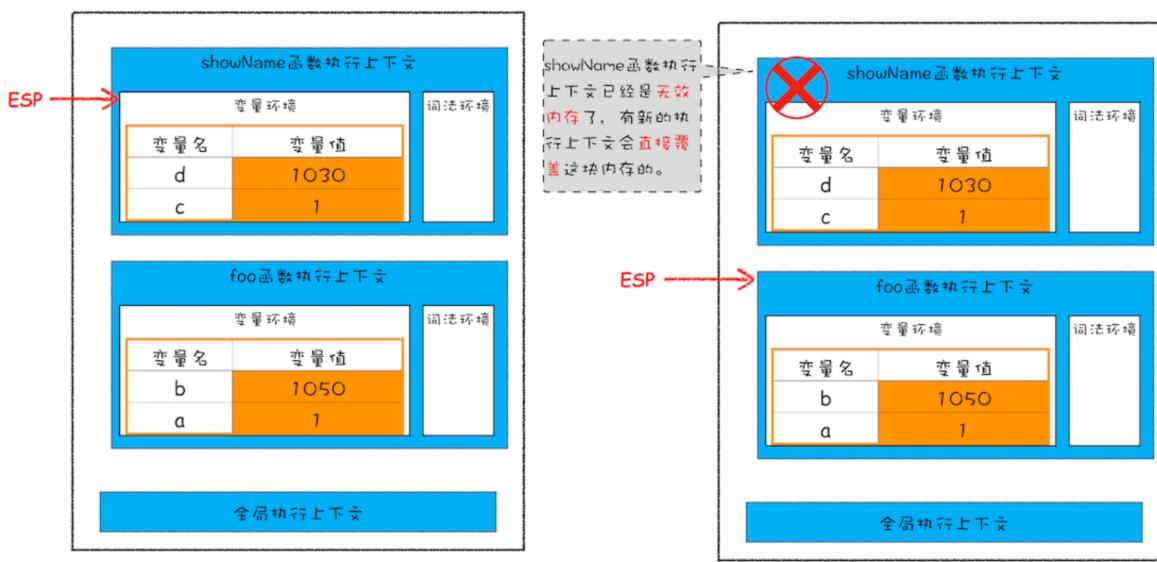


2、垃圾回收

垃圾回收分为手动回收和自动回收两种策略

对于JS，分别看栈和堆中的垃圾回收

- 栈：当一个函数执行结束之后，JavaScript引擎会通过向下移动ESP（记录当前执行状态的指针）来销毁该函数保存在栈中的执行上下文

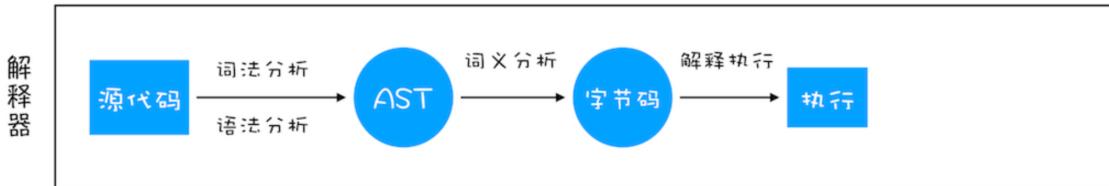
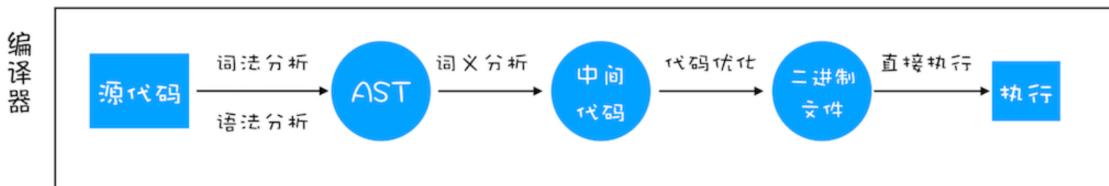


- 堆：主要通过副垃圾回收器（新生代）和主垃圾回收器（老生代），副垃圾回收器采用scavenge算法将区域分为对象区域和空闲区域，通过两个区域的反转让新生代区域无限使用下去。主垃圾回收器采用Mark-Sweep（Mark-Compact Incremental Marking解决不同场景下问题的算法改进）算法进行空间回收的。无论是主副垃圾回收器的策略都是标记-清除-整理三个大的步骤。另外还有新世代的晋升策略（两次未清除的），大对象直接分配在老生代。

3、编译器和解释器：V8是如何执行一段JavaScript代码的

前置概念和原理

- 编译器 (Compiler)
- 解释器 (Interpreter)
- 抽象语法树 (AST)
- 字节码 (Bytecode)
- 即时编译器 (JIT)
- 编译型语言：程序执行之前，需要经过编译器的编译过程，并且编译之后会直接保留机器能读懂的二进制文件，这样每次运行程序时，都可以直接运行该二进制文件，而不需要再次重新编译了。比如 C/C++、GO 等都是编译型语言。
- 解释型语言：每次运行时都需要通过解释器对程序进行动态解释和执行。比如 Python、JavaScript 等都属于解释型语言

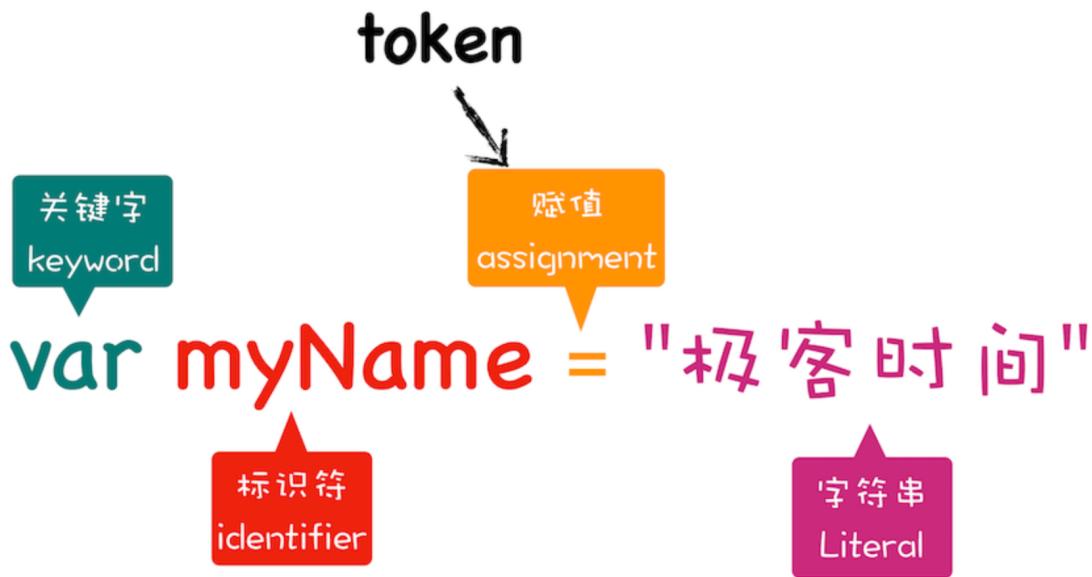


3、V8是如何执行一段JS代码的

- 生成抽象语法树(AST)和执行上下文
 - 分词 (tokenize) , 词法分析
 - 解析 (parse) , 语法分析
- 生成字节码
 - 字节码、机器码
- 执行代码

3.1 生成AST

分词 (tokenize) , 词法分析：将一行行的源码拆解成一个个 token, 即语法上不可能再分的、最小的单个字符或字符串



解析 (parse) , 语法分析: 将上一步生成的 token 数据, 根据语法规则转为 AST

AST的应用非常广泛, 比如Babel, 工作原理就是先将 ES6 源码转换为 AST, 然后再将 ES6 语法的 AST 转换为 ES5 语法的 AST, 最后利用 ES5 的 AST 生成 JavaScript 源代码。比如Eslint也是利用了AST来检查JS语法

3.2 生成字节码

生成字节码: V8的解释器 Ignition会根据 AST 生成字节码, 并解释执行字节码

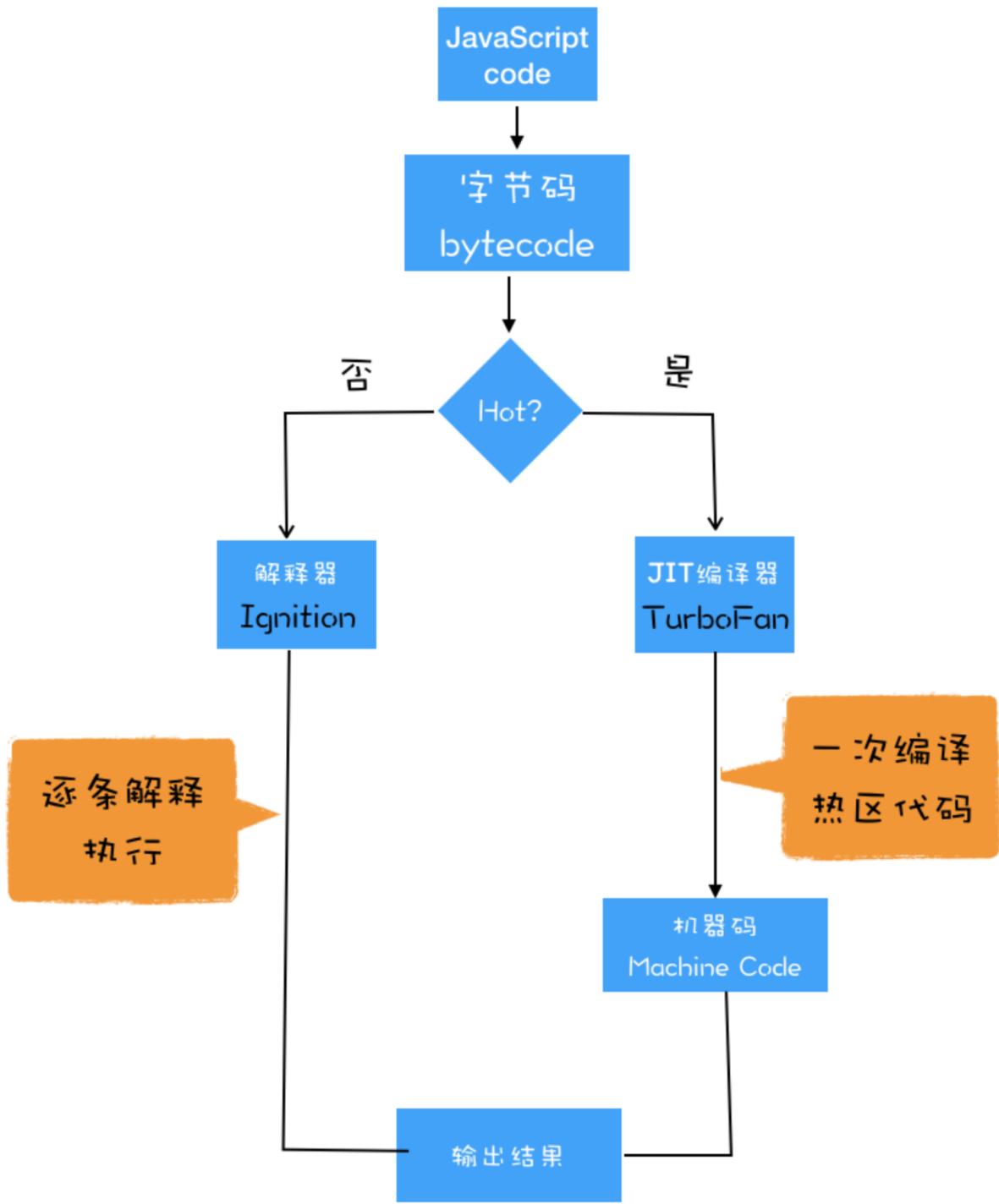
其实一开始 V8 并没有字节码, 而是直接将 AST 转换为机器码, 由于执行机器码的效率是非常高效的, 但机器码的问题在于占用内存比较大, V8团队就引入了字节码

字节码就是介于 AST 和机器码之间的一种代码。但是与特定类型的机器码无关, 字节码需要通过解释器将其转换为机器码后才能执行



3.3 执行代码

- 第一次碰到字节码, 解释器Ignition会逐条解释执行
- 如果发现了热点代码 (多次使用), 交给编译器TurboFan 编译为机器码并保存下来, 下次执行就会更加高效



字节码配合解释器和编译器的技术被称之为**即时编译JIT**，Java 和 Python 的虚拟机也都是基于这种技术实现的

V8执行越久，被编译成机器码的热点代码就越多，所以整体执行效率就越高
由于引入了字节码，有了弹性空间，也可以在内存和执行速度之间做调节了
相比之前的V8，将JS代码全部编译成字节码，这种模式就没有协商的空间了

4、JS性能

其实上面的策略，V8已经做得很牛逼了，对于开发者来说，基本不太需要关注底层的实现，而是将更多的精力花在单次脚本执行时间和网络下载资源上

【浏览器原理-04】 页面循环系统

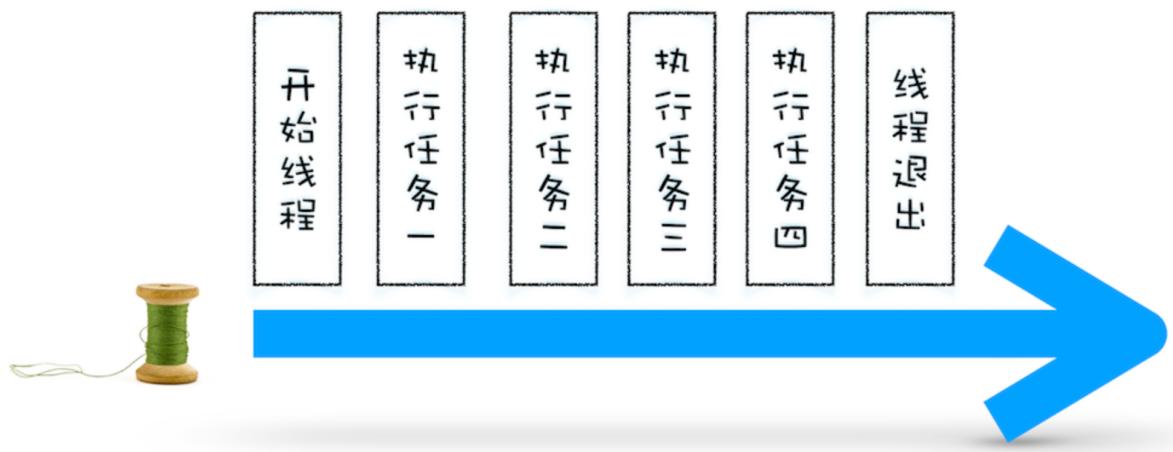
1、消息队列和事件循环

每个渲染进程都有一个主线程，但主线程非常繁忙，要同时处理DOM、计算样式、处理布局、JS任务以及各种输入事件，所以需要引入一个系统来统筹调度这些任务，这个系统就是接下来要讨论的**消息队列和事件循环系统**

我们先从最简单的入手

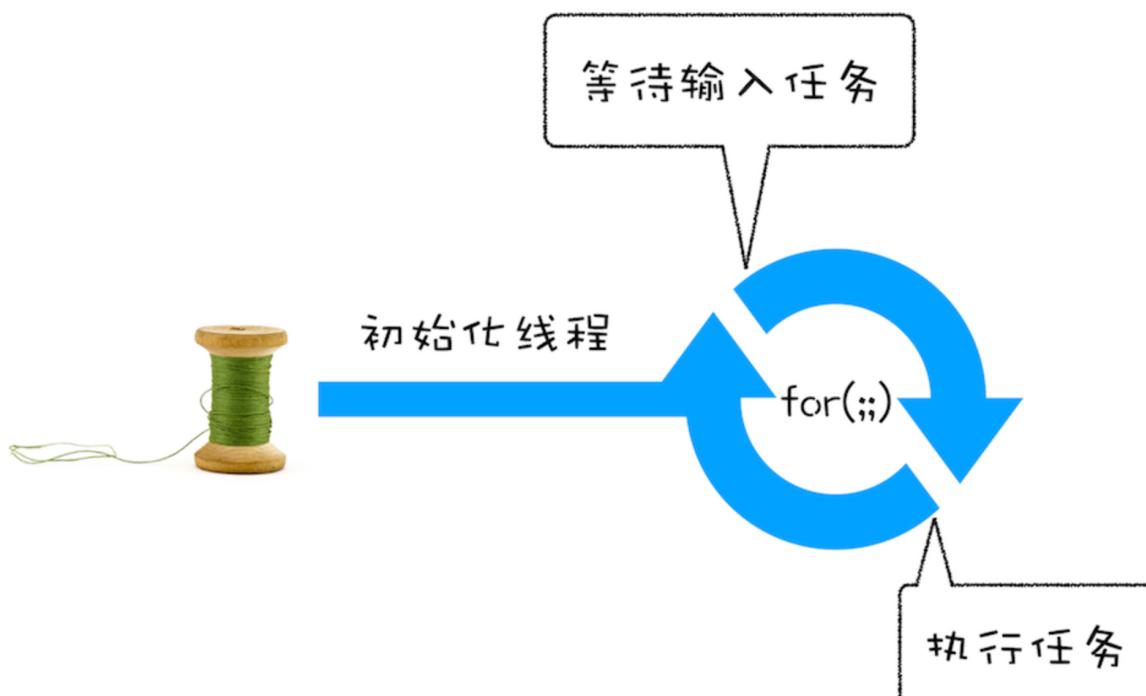
1.1 单线程：顺序处理任务

- 任务1：1+1
- 任务2：2+2
- 任务3：3+3
- . . . 一系列同步任务
- 任务结束



1.2 在线程运行过程中处理新任务

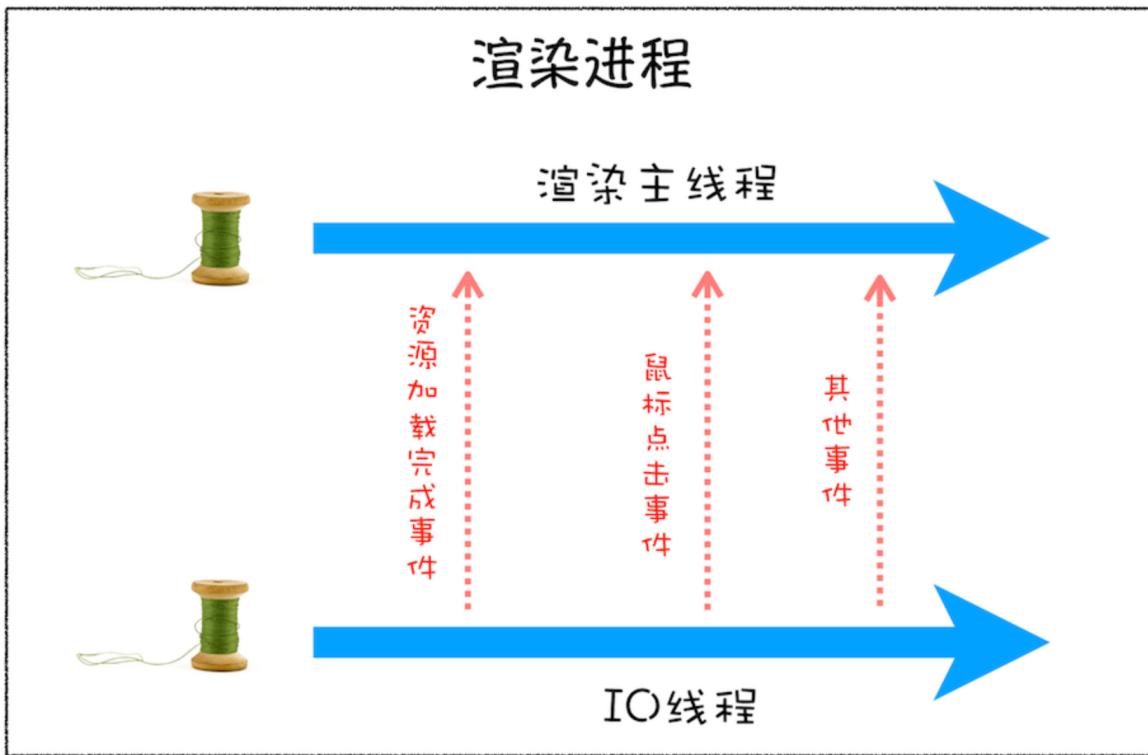
上面的任务都是事先安排好的，但是如果在线程运行中，需要接收并执行新的任务，那么就需要引入事件循环机制了



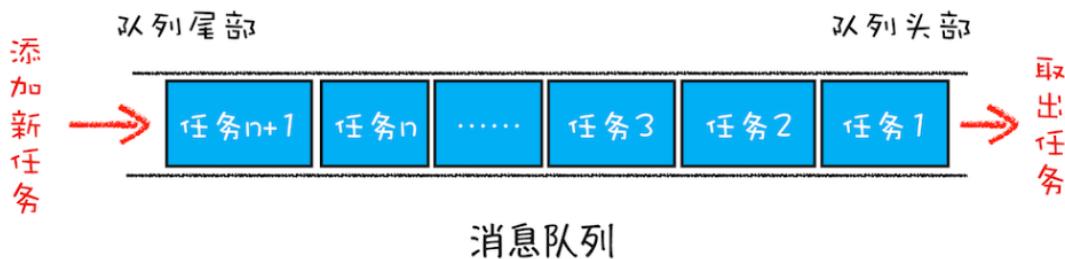
- 引入循环机制，在线程后添加一个for循环语句，让线程一直循环执行
- 引入事件，把一个任务理解为一个事件，一次for循环可以等待一个事件的执行

1.3 处理其他线程发送过来的任务

如果另外一个线程想要让主线程执行一个任务，以上的模型是做不到的，比如下图

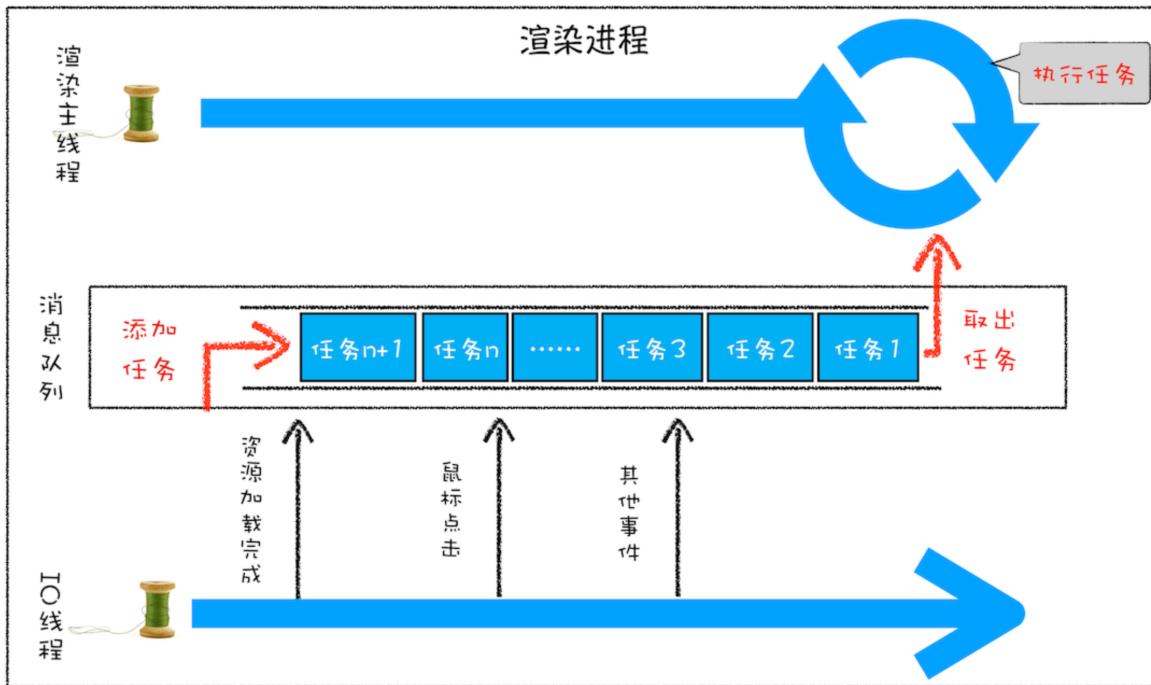


那么我们需要引入消息队列的概念



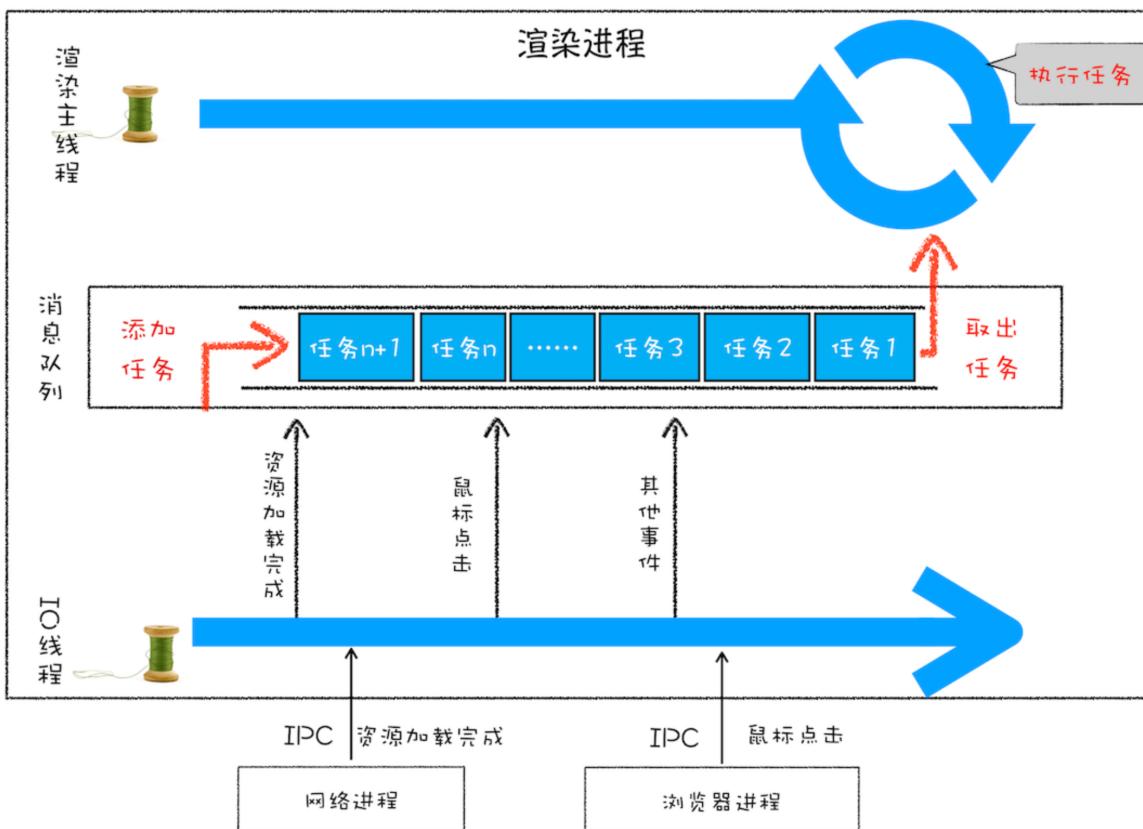
消息队列是一种数据结构，其实就是栈，先进先出

然后改造后的第三版模型



1.4 跨进程执行任务

有了第三版的消息队列模型，实现了主进程下线程之间的消息通信，那再扩大一点，跨进程之间的任务处理也是类似的，在渲染进程中有一个专门的IO线程用来接收其他进程传来的消息，然后组装成任务推进消息队列



1.5 单线程的缺点

- 如何处理高优先级的任务
- 如何解决单个任务执行时间过长的问题

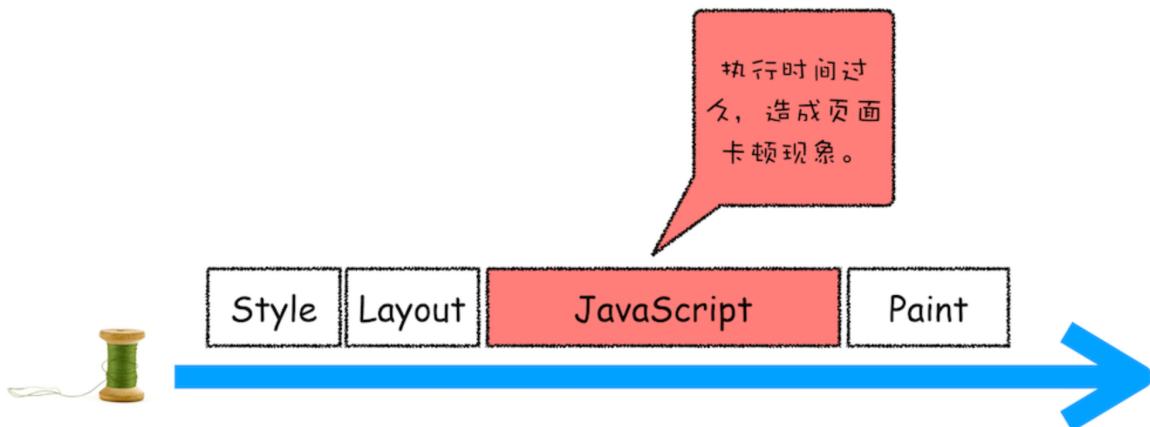
第一个问题，比如监控DOM节点变化，执行业务逻辑

通用方案是JS设计监听接口，渲染引擎同步调用接口，但带来的问题是每次变化时都执行JS接口，当前任务会被无限拉长，执行效率过低；但如果把DOM变化做成异步消息事件，推到消息队列尾部，那么监控的实时性又会降低

如果DOM发生变化，采用同步通知的方式，会影响当前任务的执行效率；如果采用异步方式，又会影响到监控的实时性。

这时候**微任务**就诞生了，把DOM变化的任务推进微任务队列，等当前宏任务执行完后，不是立马去执行下一个宏任务，而是先把当前宏任务下的微任务队列给执行清空，这样就比较合理的兼顾了实时性和效率的问题

但是，如果这个宏任务中存在一个耗时比较长的任务呢，也就是第二个问题，单任务执行过长阻塞，如图所示



比如用JS执行动画，如果某个任务执行时间过长，占用了动画单帧的时间，页面就会卡顿，也就是掉帧，在JS中我们通过滞后该任务，也就是回调的形式，后面再讨论

1.6 小结

- 简单同步线程模型，执行事先确认好的同步任务
- 引入循环语句和事件系统，在线程执行中接收并处理新的任务
- 引入消息队列，接收其他线程发送过来的任务
- 由渲染进程的IO线程来接收其他进程通过IPC发送过来的任务，继续沿用消息队列模型
- 为了优化消息队列的效率和实时性问题，引入了微任务

Docker基础

#develop/Docker

一、核心

docker通过内核虚拟化技术（namespace及cgroups等）来提供容器的资源隔离与安全保障等，由于docker通过操作系统层的虚拟化实现隔离，所以docker容器在运行时，不需要类似虚拟机额外的操作系统开销，提供资源利用率



Docker 容器本质上是宿主机上的一个进程。Docker 通过 namespace 实现了资源隔离，通过 cgroups 实现了资源的限制，通过写时复制机制（copy-on-write）实现了高效的文件操作。

Docker 有五个命名空间：进程、网络、挂载、宿主和共享内存，为了隔离有问题的应用，Docker 运用 Namespace 将进程隔离，为进程或进程组创建已隔离的运行空间，为进程提供不同的命名空间视图。这样，每一个隔离出来的进程组，对外就表现为一个 container(容器)。需要注意的是，Docker 让用户误以为自己占据了全部资源，但这并不是“虚拟机”

Docker 组件：镜像、容器、仓库

镜像

一个只读层被称为镜像，一个镜像是永久不会变的，也是无状态的

Docker 使用一个统一文件系统，Docker 进程认为整个文件系统是以读写方式挂载的



每一个镜像都可能依赖于由一个或多个下层的组成的另一个镜像。我们有时说，下层那个 镜像是上层镜像的父镜像，一个镜像不能超过 127 层

容器

容器是从镜像创建的运行实例。

它可以被启动、开始、停止、删除。

每个容器都是相互隔离的、保证安全的平台。可以把容器看做是一个简易版的 Linux 环境，Docker 利用容器来运行应用。

镜像是只读的，容器在启动的时候创建一层可写层作为最上层。

仓库

仓库是集中存放镜像文件的场所，仓库注册服务器（Registry）上往往存放着多个仓库，每个仓库中又包含了多个镜像，每个镜像有不同的标签（tag）

试运行一个 web 应用（容器使用）

载入一个镜像（一个 Python Flask 应用）



运行应用 -d 让容器在后台运行 -P：将容器内部的网络端口映射到我们使用的主机上

```
docker run -d -P training/webapp python app.py
```

然后用 `docker ps` 来查看



`docker container ls` 也可查看正在运行的容器

Docker 开放了 5000 端口（默认 Python Flask 端口）映射到主机端口 32769 上。

这时我们可以通过浏览器 `http://localhost:32768/` 访问



也可以指定 -p 标识来绑定指定端口

```
docker run -d -p 5000:5000 training/webapp python app.py
```

```
docker stop ID 停止web应用容器
```

```
docker start ID 重新开启
```

```
docker rm ID 删除容器
```

镜像使用

当运行容器时，使用的镜像如果在本地中不存在，docker 就会自动从 docker 镜像仓库中下载，默认是从 [Docker Hub](#) 公共镜像源下载。

```
docker images 列出本地主机上的镜像
```



- REPOSITORY: 表示镜像的仓库源
- TAG: 镜像的标签(表示不同版本)
- IMAGE ID: 镜像ID
- CREATED: 镜像创建时间
- SIZE: 镜像大小

```
docker search [Name] 搜索镜像
```

```
docker pull [Name] 下载镜像
```

```
docker run [Name] 运行镜像
```

```
docker build 构建镜像 (需要创建文件)
```

容器链接

1、网络端口映射

- -P :是容器内部端口随机映射到主机的高端口
- -p :是容器内部端口绑定到指定的主机端口

```
docker run -d -p 5000:5000 training/webapp python app.py
```

默认都是绑定 tcp 端口，如果要绑定 UDP 端口，可以在端口后面加上 /udp

```
docker run -d -p 127.0.0.1:5000:5000/udp training/webapp python app.py
```

```
docker port 快速查看端口绑定情况
```

docker连接系统-创建父子关系

当我们创建一个容器的时候，docker会自动对它进行命名

我们也可以使用--name标识来命名容器

```
docker run -d -P --name runoob training/webapp python app.py
```

Docker 运行Node.js应用

#Front-End/js/Node

[Dockerizing a Node.js web app | Node.js](#)

一、创建node-app

```
mkdir docker-node-app  
cd docker-node-app  
touch package.json
```

package.json文件内容

```
{  
  "name": "docker_web_app",  
  "version": "1.0.0",  
  "description": "Node.js on Docker",  
  "author": "First Last <first.last@example.com>",  
  "main": "server.js",  
  "scripts": {  
    "start": "node server.js"  
  },  
  "dependencies": {  
    "express": "^4.16.1"  
  }  
}
```

安装依赖

```
npm install
```

新建 server.js 文件

```
touch server.js
```

server.js 文件内容

```
'use strict';  
  
const express = require('express');  
  
// 官方用的是0.0.0.0地址，指通配地址，我改了下  
// 127.0.0.1地址代表本地地址，也就是localhost，一般是测试所用  
const PORT = 8080;  
const HOST = '127.0.0.1';  
  
const app = express();  
app.get('/', (req, res) => {  
  res.send('I am Docker-node-app');  
});  
  
app.listen(PORT, HOST);  
console.log(`Running on http://${HOST}:${PORT}`);
```

试跑一下

```
▶ node server.js  
Running on http://127.0.0.1:3000
```

浏览器打开 <http://127.0.0.1:3000> 应该能看到 I am Docker-node-app

二、Creating a Dockerfile

```
touch Dockerfile
```

编辑 Dockerfile 配置文件

```
# 使用最新的LTS node版本，将会从Docker Hub上面拉取这个版本  
FROM node:carbon  
  
# 定义项目要上传的容器位置，也就是我们这个项目要放到那个容器中  
WORKDIR /ndoe/app  
  
# 需要使用npm 所以复制一发  
# npm 从版本4以上会生成package-lock.json文件  
COPY package*.json ./  
  
# 运行安装指令  
RUN npm install  
  
# 复制当前app目录文件到上面定义的目录WORKDIR中  
# Bundle app source  
COPY . .  
  
# 对外开放端口，让外部可以访问容器内的app  
EXPOSE 8080  
  
# 启动App，两条命令其实一样的  
# CMD ["node", "server.js"]  
CMD ["npm", "start"]
```

三、.dockerignore file

类似 .gitignore 文件，排除某些文件
这些文件就不会被添加到镜像中去

```
node_modules  
npm-debug.log
```

四、构建镜像

在有 Dockerfile 的目录，也就是当前目录下，使用如下命令构建镜像
注意最后有个小点点，不要漏了

```
docker build -t <your username>/<appname> .
```

比如我的是这样的（注意这个/ 下面是会用到的，不然最后访问端口的时候会失败）

```
docker build -t lgc/docker-node-app-image .
```

等待docker拉取和执行，看到 `successful` 字眼，就是成功了

五、运行镜像

`-d`：后台运行

`-P 3000:3000`：将容器中的3000端口映射到外部的3000端口

`--name <appname>` 给该容器命名

`lgc/docker-node-app-image` 前面构建镜像时定义的名字

我的是这样的

```
docker run -d --name docker-node-app -p 3000:3000 lgc/docker-node-app-image
```

然后浏览器输入 `http://127.0.0.1:3000/` 即可访问了

vim 是 **vi** 的升级版本，它不仅兼容 **vi** 的所有指令，还有一些新特性

```
// 使用vim模式打开文件  
► vim [path/filename]
```

概述

- 命令模式 (command mode) - 默认模式
- 插入模式 (Insert mode) 通过按下字母 `i` 进入该模式，用于输入文字
- 可视模式 按下 `v` 进入，常用于复制粘贴
- 底行模式 (last line mode) 通过 `:` 进入，用于保存以及退出

一、插入模式

- `i` 进入插入模式后从当前位置开始输入文字
- `o` 插入新的一行，从行首开始输入文字
- `a` 从当前光标的下一个位置开始输入文字

`esc` 退出插入模式

二、命令模式

1、移动

正规的 **vi** 操作是 `h`(左)、`j`(下)、`k`(上)、`l`(右) 分别控制光标移动

- `ctrl+u` 屏幕向上半页，不是光标移动
- `ctrl+d` 屏幕向下半页，不是光标移动
- `0` 到行首
- `$` 到行末
- `w` 到下个单词的开头
- `e` 到下个单词的字尾
- `b` 回到上个单词的开头

2、删除

- 小写 `x` 每按一次，删除当前光标一个字符，`#x` 删除 # 个
- 大写 `X` 每按一次，删除当前光标前面一个字符
- `dd` 删除一行，`#dd` 从当前行开始删除 # 行

3、复制粘贴

- `yy` 复制当前行，`#yy` 复制 # 行
- `p` 粘贴 粘贴到当前行的下方

4、撤销

`u` 撤销上一步操作

三、可视模式

- 从当前光标开始，移动光标，会选中内容
- `y` 复制
- 移动光标到对应位置，`p` 粘贴
- `esc` 退出

四、底行模式

- `w [filename]` 将文章以指定的文件名 [filename] 保存
- `w` 保存 `wq` 保存并退出
- `q!` 不保存强制退出
- `set nu` 列出行数
- `#` 再回车，光标就会自动跳转到第 # 行
- `[#1],[#2]d` 回车，删除 [#1] -> [#2] 行
- `/[搜索词]`，一直按 `n` 可持续向下查找
- `?[搜索词]`，一直按 `n` 可持续向上查找

五、其他操作

1、多行注释/插入

- `ctrl + v` 进入区块模式 (VISUAL BLOCK)
- 移动光标选择多行 (可视模式下移动光标即为选择该行)
- 大写 `I`，进入插入模式
- 输入注释符 `#` 或者 `//`，这里只会看见注释了一行，不要慌
- `esc` 退出，稍等一会选中的全部行都会被注释

如果要同时编辑多行，道理是一样的，可以自己试一下，把注释符号理解为插入字符即可

2、多行删除

- `ctrl + v` 进入区块模式 (VISUAL BLOCK)
- 移动光标选择多行 (可视模式下移动光标即为选择该行)
- `x` 或者 `d` 删除对应字符

注意，按一次只能删除一个字符，如果是 `//` 这种，需要执行两次操作

Express + create-react-app 快速构建前后端开发环境

一、快速构建react app

上官网全局安装create-react-app

[GitHub - facebook/create-react-app: Create React apps with no build configuration.](https://github.com/facebook/create-react-app)

```
npx create-react-app react-express-fullstack  
cd react-express-fullstack  
yarn start
```

这时候访问 `http://localhost:3000/` 可以正常访问

二、配合 Express 构建 server 端应用

创建一个叫server的文件夹，并初始化 `package.json` 文件

```
mkdir server && cd server  
yarn init
```

安装几个必备依赖

```
yarn add express body-parser nodemon babel-cli babel-preset-es2015
```

body-parser用于解析post请求

nodemon检测node.js 改动并自动重启，适用于开发阶段

babel相关的都是为了用ES6进行开发

在 `package.json` 文件增加如下命令

```
"scripts": {  
  "start": "nodemon --exec babel-node -- ./server.js",  
  "build": "babel ./server.js --out-file server-compiled.js",  
  "serve": "node server-compiled.js"  
}
```

在上面的react app中会启动一个静态资源服务器

那么server这边的服务器要怎么启动呢？

同时开两个服务器也是可以的，但我想一条命令跑两个本地服务器的时候怎么做呢？

回去修改react app的 `package.json` 文件，它原来有这样的一段代码

```
"scripts": {  
  "start": "react-scripts start",  
  "build": "react-scripts build",  
  "test": "react-scripts test --env=jsdom",  
  "eject": "react-scripts eject"  
}
```

我们主要修改一下 `start` 和 `build`，也就是把server端的命令合并进来即可

先安装一个依赖包 `concurrently`，作用是同时执行多条命令

```
"start": "concurrently 'react-scripts start' 'cd server && yarn start'"
```

```
"build": "concurrently 'react-scripts build' 'cd server && yarn build'"
```

这时，我们只要执行 `yarn start` 会同步启动 `webpack` 以及 `server` 文件夹下的 `nodeman`

在 `server` 目录下新建 `server.js` 文件，并写入如下代码

```
var express = require('express');
var app = express();

app.get('/users', function (req, res) {
  res.json([
    {
      "id": 1,
      "name": 'one'
    },
    {
      "id": 2,
      "name": 'two'
    }
  ]);
}

app.listen(5000, function () {
  console.log('Example app listening on port 3000!');
});
```

注意这里的5000，是本地服务器的端口

然后，在react app项目的 `package.json` 文件中写入如下代码，设置项目的请求代理 => server的5000端口

```
"proxy": "http://127.0.0.1:5000"
```

改下 `src/app.js` 文件，请求 `users` 接口

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {

  state = {
    users: []
  }

  componentDidMount() {
    fetch('/users')
      .then(res => res.json())
      .then(users => this.setState({ users }));
  }
}
```

```
render() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <h1 className="App-title">Welcome to React</h1>
      </header>
      {this.state.users.map(user => {
        return <div key={user.id}>{user.name}</div>
      })}
    </div>
  );
}

export default App;
```

如果请求到users数据，说明请求成功了