

## 2万5千字大厂面经 | 掘金技术征文

以下面试题来自腾讯、阿里、网易、饿了么、美团、拼多多、百度等等大厂综合起来常考的题目。

### 如何写一个漂亮的简历

---

简历不是一份记流水账的东西，而是让用人方了解你的亮点的。

平时有在做一些修改简历的收费服务，也算看过蛮多简历了。很多简历都有如下特征

- 喜欢说自己的特长、优点，用人方真的不关注你的性格是否阳光等等
- 个人技能能够占半页的篇幅，而且长得也都差不多
- 项目经验流水账，比如我会用这个 API 实现了某某功能
- 简历页数过多，真心看不下去

以上类似简历可以说用人方也看了无数份，完全抓不到你的亮点。除非你呆过大厂或者教育背景不错或者技术栈符合人家要求了，否则基本就是看运气约面试了。

以下是我经常给别人修改简历的意见：

简历页数控制在 2 页以下

- 技术名词注意大小写
- 突出个人亮点，扩充内容。比如在项目中如何找到 Bug，解决 Bug 的过程；比如如何发现的性能问题，如何解决性能问题，最终提升了多少性能；比如为何如此选型，目的是什么，较其他有什么优点等等。总体思路就是不写流水账，突出你在项目中具有不错的解决问题的能力 and 独立思考的能力。
- 斟酌熟悉、精通等字眼，不要给自己挖坑
- 确保每一个写上去的技术点自己都能说出点什么，杜绝面试官问你一个技术点，你只能答出会用 API 这种减分的情况

做到以上内容，然后在投递简历的过程中加上一份求职信，对你的求职之路相信能帮上很多忙。

### 谈谈变量提升?

当执行 JS 代码时，会生成执行环境，只要代码不是写在函数中的，就是在全局执行环境中，函数中的代码会产生函数执行环境，只此两种执行环境。

接下来让我们看一个老生常谈的例子，`var`

javascript 复制代码

```
b() // call b
console.log(a) // undefined

var a = 'Hello world'

function b() {
  console.log('call b')
}
```

想必以上的输出大家肯定都已经明白了，这是因为函数和变量提升的原因。通常提升的解释是说将声明的代码移动到了顶部，这其实没有什么错误，便于大家理解。但是更准确的解释应该是：在生成执行环境时，会有两个阶段。第一个阶段是创建的阶段，JS 解释器会找出需要提升的变量和函数，并且给他们提前在内存中开辟好空间，函数的话会将整个函数存入内存中，变量只声明并且赋值为 `undefined`，所以在第二个阶段，也就是代码执行阶段，我们可以直接提前使用。

在提升的过程中，相同的函数会覆盖上一个函数，并且函数优先于变量提升

javascript 复制代码

```
b() // call b second

function b() {
  console.log('call b fist')
}
function b() {
  console.log('call b second')
}
var b = 'Hello world'
```

`var` 会产生很多错误，所以在 ES6 中引入了 `let`。`let` 不能在声明前使用，但是这并不是常说的 `let` 不会提升，`let` 提升了，在第一阶段内存也已经为他开辟好了空间，但是因为这个声明的特性导致了并不能在声明前使用。

## bind、call、apply 区别

首先说下前两者的区别。

`call` 和 `apply` 都是为了解决改变 `this` 的指向。作用都是相同的，只是传参的方式不同。

除了第一个参数外，`call` 可以接收一个参数列表，`apply` 只接受一个参数数组。

javascript 复制代码

```
let a = {
  value: 1
}
function getValue(name, age) {
  console.log(name)
  console.log(age)
  console.log(this.value)
}
getValue.call(a, 'yck', '24')
getValue.apply(a, ['yck', '24'])
```

`bind` 和其他两个方法作用也是一致的，只是该方法会返回一个函数。并且我们可以通过 `bind` 实现柯里化。

## 如何实现一个 bind 函数

对于实现以下几个函数，可以从几个方面思考

- 不传入第一个参数，那么默认为 `window`
- 改变了 `this` 指向，让新的对象可以执行该函数。那么思路是否可以变成给新的对象添加一个函数，然后在执行完以后删除？

javascript 复制代码

```
Function.prototype.myBind = function (context) {
  if (typeof this !== 'function') {
    throw new TypeError('Error')
  }
  var _this = this
  var args = [...arguments].slice(1)
  // 返回一个函数
  return function F() {
    // 因为返回了一个函数，我们可以 new F(), 所以需要判断
    if (this instanceof F) {
      return new _this(...args, ...arguments)
    }
    return _this.apply(context, args.concat(...arguments))
  }
}
```

```
}  
}
```

## 如何实现一个 call 函数

javascript 复制代码

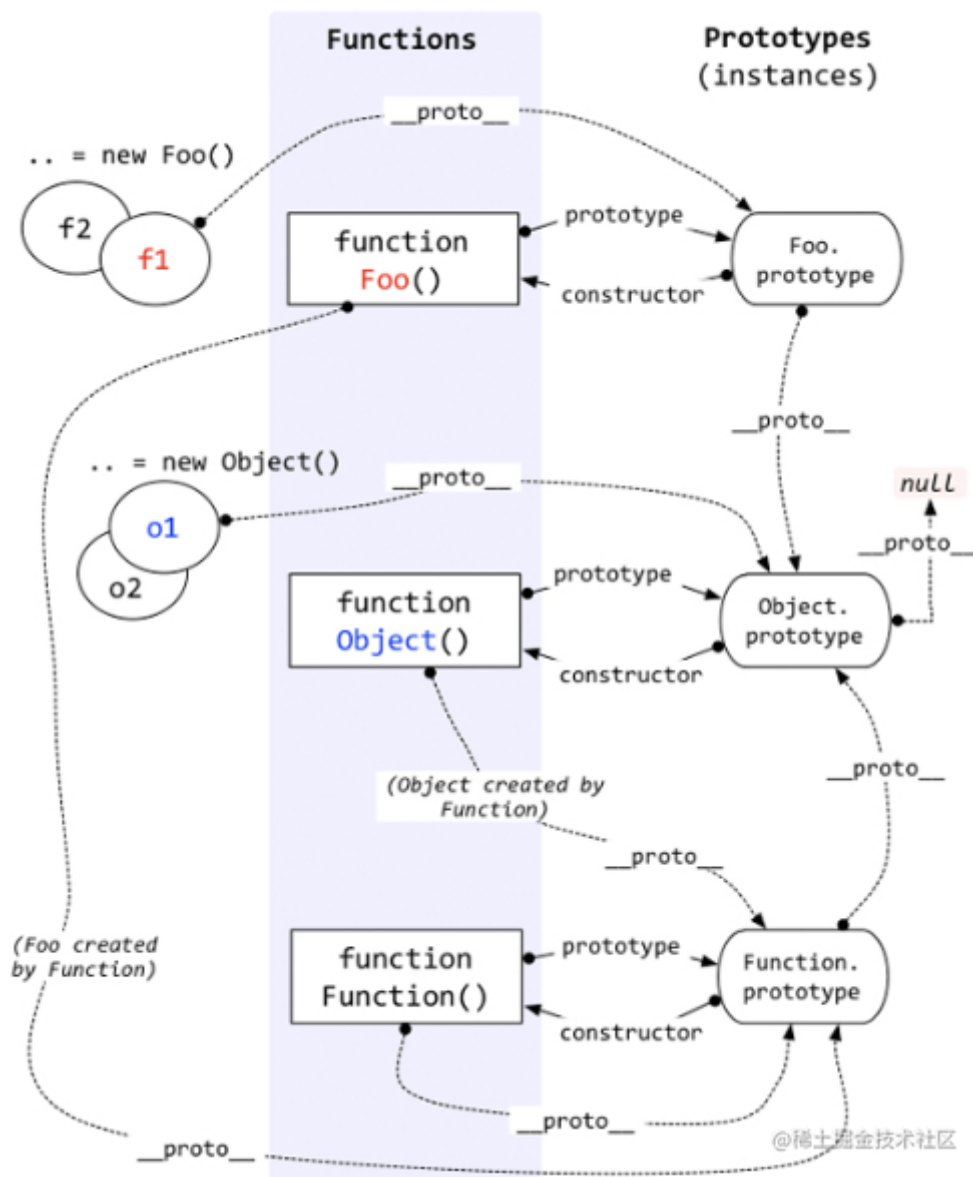
```
Function.prototype.myCall = function (context) {  
  var context = context || window  
  // 给 context 添加一个属性  
  // getValue.call(a, 'yck', '24') => a.fn = getValue  
  context.fn = this  
  // 将 context 后面的参数取出来  
  var args = [...arguments].slice(1)  
  // getValue.call(a, 'yck', '24') => a.fn('yck', '24')  
  var result = context.fn(...args)  
  // 删除 fn  
  delete context.fn  
  return result  
}
```

## 如何实现一个 apply 函数

javascript 复制代码

```
Function.prototype.myApply = function (context) {  
  var context = context || window  
  context.fn = this  
  
  var result  
  // 需要判断是否存储第二个参数  
  // 如果存在，就将第二个参数展开  
  if (arguments[1]) {  
    result = context.fn(...arguments[1])  
  } else {  
    result = context.fn()  
  }  
  
  delete context.fn  
  return result  
}
```

## 简单说下原型链？



每个函数都有 `prototype` 属性，除了 `Function.prototype.bind()`，该属性指向原型。

每个对象都有 `__proto__` 属性，指向了创建该对象的构造函数的原型。其实这个属性指向了 `[[prototype]]`，但是 `[[prototype]]` 是内部属性，我们并不能访问到，所以使用 `__proto__` 来访问。

对象可以通过 `__proto__` 来寻找不属于该对象的属性，`__proto__` 将对象连接起来组成了原型链。

如果你想更进一步的了解原型，可以仔细阅读 [深度解析原型中的各个难点](#)。

## 怎么判断对象类型？

- 可以通过 `Object.prototype.toString.call(xx)`。这样我们就可以获得类似 `[object Type]` 的字符串。

- `instanceof` 可以正确的判断对象的类型，因为内部机制是通过判断对象的原型链中是不是能找到类型的 `prototype`。

## 箭头函数的特点

javascript 复制代码

```
function a() {  
    return () => {  
        return () => {  
            console.log(this)  
        }  
    }  
}  
  
console.log(a()()())
```

箭头函数其实是没有 `this` 的，这个函数中的 `this` 只取决于他外面的第一个不是箭头函数的函数的 `this`。在这个例子中，因为调用 `a` 符合前面代码中的第一个情况，所以 `this` 是 `window`。并且 `this` 一旦绑定了上下文，就不会被任何代码改变。

## This

`this` 是很多人会混淆的概念，但是其实他一点都不难，你只需要记住几个规则就可以了。

scss 复制代码

```
function foo() {  
    console.log(this.a)  
}  
  
var a = 1  
foo()  
  
var obj = {  
    a: 2,  
    foo: foo  
}  
  
obj.foo()
```

// 以上两者情况 `this` 只依赖于调用函数前的对象，优先级是第二个情况大于第一个情况

// 以下情况是优先级最高的，`this` 只会绑定在 `c` 上，不会被任何方式修改 `this` 指向

```
var c = new foo()  
c.a = 3  
console.log(c.a)
```

// 还有种就是利用 `call`, `apply`, `bind` 改变 `this`，这个优先级仅次于 `new`

## async、await 优缺点

`async` 和 `await` 相比直接使用 `Promise` 来说，优势在于处理 `then` 的调用链，能够更清晰准确的写出代码。缺点在于滥用 `await` 可能会导致性能问题，因为 `await` 会阻塞代码，也许之后的异步代码并不依赖于前者，但仍然需要等待前者完成，导致代码失去了并发性。

下面来看一个使用 `await` 的代码。

less 复制代码

```
var a = 0
var b = async () => {
  a = a + await 10
  console.log('2', a) // -> '2' 10
  a = (await 10) + a
  console.log('3', a) // -> '3' 20
}
b()
a++
console.log('1', a) // -> '1' 1
```

对于以上代码你可能会有疑惑，这里说明下原理

- 首先函数 `b` 先执行，在执行到 `await 10` 之前变量 `a` 还是 0，因为在 `await` 内部实现了 `generators`，`generators` 会保留堆栈中东西，所以这时候 `a = 0` 被保存了下来
- 因为 `await` 是异步操作，遇到 `await` 就会立即返回一个 `pending` 状态的 `Promise` 对象，暂时返回执行代码的控制权，使得函数外的代码得以继续执行，所以会先执行 `console.log('1', a)`
- 这时候同步代码执行完毕，开始执行异步代码，将保存下来的值拿出来使用，这时候 `a = 10`
- 然后后面就是常规执行代码了

## generator 原理

Generator 是 ES6 中新增的语法，和 `Promise` 一样，都可以用来异步编程

javascript 复制代码

```
// 使用 * 表示这是一个 Generator 函数
// 内部可以通过 yield 暂停代码
// 通过调用 next 恢复执行
function* test() {
  let a = 1 + 2;
  yield 2;
  yield 3;
```

```

}
let b = test();
console.log(b.next()); // > { value: 2, done: false }
console.log(b.next()); // > { value: 3, done: false }
console.log(b.next()); // > { value: undefined, done: true }

```

从以上代码可以发现，加上 `*` 的函数执行后拥有了 `next` 函数，也就是说函数执行后返回了一个对象。每次调用 `next` 函数可以继续执行被暂停的代码。以下是 Generator 函数的简单实现

javascript 复制代码

```

// cb 也就是编译过的 test 函数
function generator(cb) {
  return (function() {
    var object = {
      next: 0,
      stop: function() {}
    };

    return {
      next: function() {
        var ret = cb(object);
        if (ret === undefined) return { value: undefined, done: true };
        return {
          value: ret,
          done: false
        };
      }
    };
  })();
}

// 如果你使用 babel 编译后可以发现 test 函数变成了这样
function test() {
  var a;
  return generator(function(_context) {
    while (1) {
      switch ((_context.prev = _context.next)) {
        // 可以发现通过 yield 将代码分割成几块
        // 每次执行 next 函数就执行一块代码
        // 并且表明下次需要执行哪块代码
        case 0:
          a = 1 + 2;
          _context.next = 4;
          return 2;
        case 4:
          _context.next = 6;
          return 3;
        // 执行完毕

```



```

        case 6:
        case "end":
            return _context.stop();
    }
}
});
}

```

## Promise

Promise 是 ES6 新增的语法，解决了回调地狱的问题。

可以把 Promise 看成一个状态机。初始是 `pending` 状态，可以通过函数 `resolve` 和 `reject`，将状态转变为 `resolved` 或者 `rejected` 状态，状态一旦改变就不能再次变化。

`then` 函数会返回一个 Promise 实例，并且该返回值是一个新的实例而不是之前的实例。因为 Promise 规范规定除了 `pending` 状态，其他状态是不可以改变的，如果返回的是一个相同实例的话，多个 `then` 调用就失去意义了。

对于 `then` 来说，本质上可以把它看成是 `flatMap`

## 如何实现一个 Promise

ini 复制代码

```

// 三种状态
const PENDING = "pending";
const RESOLVED = "resolved";
const REJECTED = "rejected";
// promise 接收一个函数参数，该函数会立即执行
function MyPromise(fn) {
    let _this = this;
    _this.currentState = PENDING;
    _this.value = undefined;
    // 用于保存 then 中的回调，只有当 promise
    // 状态为 pending 时才会缓存，并且每个实例至多缓存一个
    _this.resolvedCallbacks = [];
    _this.rejectedCallbacks = [];

    _this.resolve = function (value) {
        if (value instanceof MyPromise) {
            // 如果 value 是个 Promise，递归执行
            return value.then(_this.resolve, _this.reject)
        }
        setTimeout(() => { // 异步执行，保证执行顺序

```

```

        if (_this.currentState === PENDING) {
            _this.currentState = RESOLVED;
            _this.value = value;
            _this.resolvedCallbacks.forEach(cb => cb());
        }
    })
};

_this.reject = function (reason) {
    setTimeout(() => { // 异步执行, 保证执行顺序
        if (_this.currentState === PENDING) {
            _this.currentState = REJECTED;
            _this.value = reason;
            _this.rejectedCallbacks.forEach(cb => cb());
        }
    })
}

// 用于解决以下问题
// new Promise(() => throw Error('error'))
try {
    fn(_this.resolve, _this.reject);
} catch (e) {
    _this.reject(e);
}
}

MyPromise.prototype.then = function (onResolved, onRejected) {
    var self = this;
    // 规范 2.2.7, then 必须返回一个新的 promise
    var promise2;
    // 规范 2.2.onResolved 和 onRejected 都为可选参数
    // 如果类型不是函数需要忽略, 同时也实现了透传
    // Promise.resolve(4).then().then((value) => console.log(value))
    onResolved = typeof onResolved === 'function' ? onResolved : v => v;
    onRejected = typeof onRejected === 'function' ? onRejected : r => throw r;

    if (self.currentState === RESOLVED) {
        return (promise2 = new MyPromise(function (resolve, reject) {
            // 规范 2.2.4, 保证 onFulfilled, onRejected 异步执行
            // 所以用了 setTimeout 包裹下
            setTimeout(function () {
                try {
                    var x = onResolved(self.value);
                    resolutionProcedure(promise2, x, resolve, reject);
                } catch (reason) {
                    reject(reason);
                }
            });
        }));
    }
});

```

```

}

if (self.currentState === REJECTED) {
  return (promise2 = new MyPromise(function (resolve, reject) {
    setTimeout(function () {
      // 异步执行onRejected
      try {
        var x = onRejected(self.value);
        resolutionProcedure(promise2, x, resolve, reject);
      } catch (reason) {
        reject(reason);
      }
    });
  }));
}

if (self.currentState === PENDING) {
  return (promise2 = new MyPromise(function (resolve, reject) {
    self.resolvedCallbacks.push(function () {
      // 考虑到可能会有报错，所以使用 try/catch 包裹
      try {
        var x = onResolved(self.value);
        resolutionProcedure(promise2, x, resolve, reject);
      } catch (r) {
        reject(r);
      }
    });

    self.rejectedCallbacks.push(function () {
      try {
        var x = onRejected(self.value);
        resolutionProcedure(promise2, x, resolve, reject);
      } catch (r) {
        reject(r);
      }
    });
  }));
}

};

// 规范 2.3
function resolutionProcedure(promise2, x, resolve, reject) {
  // 规范 2.3.1, x 不能和 promise2 相同，避免循环引用
  if (promise2 === x) {
    return reject(new TypeError("Error"));
  }

  // 规范 2.3.2
  // 如果 x 为 Promise，状态为 pending 需要继续等待否则执行
  if (x instanceof MyPromise) {
    if (x.currentState === PENDING) {

```

```

    x.then(function (value) {
        // 再次调用该函数是为了确认 x resolve 的
        // 参数是什么类型，如果是基本类型就再次 resolve
        // 把值传给下个 then
        resolutionProcedure(promise2, value, resolve, reject);
    }, reject);
} else {
    x.then(resolve, reject);
}
return;
}
// 规范 2.3.3.3.3
// reject 或者 resolve 其中一个执行过得话，忽略其他的
let called = false;
// 规范 2.3.3，判断 x 是否为对象或者函数
if (x !== null && (typeof x === "object" || typeof x === "function")) {
    // 规范 2.3.3.2，如果不能取出 then，就 reject
    try {
        // 规范 2.3.3.1
        let then = x.then;
        // 如果 then 是函数，调用 x.then
        if (typeof then === "function") {
            // 规范 2.3.3.3
            then.call(
                x,
                y => {
                    if (called) return;
                    called = true;
                    // 规范 2.3.3.3.1
                    resolutionProcedure(promise2, y, resolve, reject);
                },
                e => {
                    if (called) return;
                    called = true;
                    reject(e);
                }
            );
        } else {
            // 规范 2.3.3.4
            resolve(x);
        }
    } catch (e) {
        if (called) return;
        called = true;
        reject(e);
    }
} else {
    // 规范 2.3.4，x 为基本类型
    resolve(x);
}

```

```
}  
}
```

## == 和 === 区别，什么情况用 ==



上图中的 `toPrimitive` 就是对象转基本类型。

这里来解析一道题目 `[] == ![] // -> true`，下面是这个表达式为何为 `true` 的步骤

scss 复制代码

```
// [] 转成 true，然后取反变成 false  
[] == false  
// 根据第 8 条得出  
[] == ToNumber(false)  
[] == 0  
// 根据第 10 条得出  
ToPrimitive([]) == 0  
// [].toString() -> ''  
'' == 0  
// 根据第 6 条得出  
0 == 0 // -> true
```

`===` 用于判断两者类型和值是否相同。在开发中，对于后端返回的 `code`，可以通过 `==` 去判断。

## 垃圾回收

V8 实现了准确式 GC，GC 算法采用了分代式垃圾回收机制。因此，V8 将内存（堆）分为新生代和老生代两部分。

## 新生代算法

新生代中的对象一般存活时间较短，使用 Scavenge GC 算法。

在新生代空间中，内存空间分为两部分，分别为 From 空间和 To 空间。在这两个空间中，必定有一个空间是使用的，另一个空间是空闲的。新分配的对象会被放入 From 空间中，当 From 空间被占满时，新生代 GC 就会启动了。算法会检查 From 空间中存活的对象并复制到 To 空间

中，如果有失活的对象就会销毁。当复制完成后将 From 空间和 To 空间互换，这样 GC 就结束了。

## 老年代算法

老年代中的对象一般存活时间较长且数量也多，使用了两个算法，分别是标记清除算法和标记压缩算法。

在讲算法前，先来说下什么情况下对象会出现在老年代空间中：

- 新生代中的对象是否已经经历过一次 Scavenge 算法，如果经历过的话，会将对象从新生代空间移到老年代空间中。
- To 空间的对象占比大小超过 25 %。在这种情况下，为了不影响到内存分配，会将对象从新生代空间移到老年代空间中。

老年代中的空间很复杂，有如下几个空间

arduino 复制代码

```
enum AllocationSpace {
    // TODO(v8:7464): Actually map this space's memory as read-only.
    RO_SPACE,      // 不变的对象空间
    NEW_SPACE,     // 新生代用于 GC 复制算法的空间
    OLD_SPACE,     // 老年代常驻对象空间
    CODE_SPACE,    // 老年代代码对象空间
    MAP_SPACE,     // 老年代 map 对象
    LO_SPACE,      // 老年代大空间对象
    NEW_LO_SPACE,  // 新生代大空间对象

    FIRST_SPACE = RO_SPACE,
    LAST_SPACE = NEW_LO_SPACE,
    FIRST_GROWABLE_PAGED_SPACE = OLD_SPACE,
    LAST_GROWABLE_PAGED_SPACE = MAP_SPACE
};
```

在老年代中，以下情况会先启动标记清除算法：

- 某一个空间没有分块的时候
- 空间中被对象超过一定限制
- 空间不能保证新生代中的对象移动到老年代中

在这个阶段中，会遍历堆中所有的对象，然后标记活的对象，在标记完成后，销毁所有没有被标记的对象。在标记大型对内存时，可能需要几百毫秒才能完成一次标记。这就会导致一些性

能上的问题。为了解决这个问题，2011 年，V8 从 stop-the-world 标记切换到增量标志。在增量标记期间，GC 将标记工作分解为更小的模块，可以让 JS 应用逻辑在模块间隙执行一会，从而不至于让应用出现停顿情况。但在 2018 年，GC 技术又有了一个重大突破，这项技术名为并发标记。该技术可以让 GC 扫描和标记对象时，同时允许 JS 运行，你可以点击 [该博客](#) 详细阅读。

清除对象后会造成堆内存出现碎片的情况，当碎片超过一定限制后会启动压缩算法。在压缩过程中，将活的对象像一端移动，直到所有对象都移动完成然后清理掉不需要的内存。

## 闭包

闭包的定义很简单：函数 A 返回了一个函数 B，并且函数 B 中使用了函数 A 的变量，函数 B 就被称为闭包。

javascript 复制代码

```
function A() {  
  let a = 1  
  function B() {  
    console.log(a)  
  }  
  return B  
}
```

你是否会疑惑，为什么函数 A 已经弹出调用栈了，为什么函数 B 还能引用到函数 A 中的变量。因为函数 A 中的变量这时候是存储在堆上的。现在的 JS 引擎可以通过逃逸分析辨别出哪些变量需要存储在堆上，哪些需要存储在栈上。

经典面试题，循环中使用闭包解决 `var` 定义函数的问题

css 复制代码

```
for ( var i=1; i<=5; i++) {  
  setTimeout( function timer() {  
    console.log( i );  
  }, i*1000 );  
}
```

首先因为 `setTimeout` 是个异步函数，所有会先把循环全部执行完毕，这时候 `i` 就是 6 了，所以会输出一堆 6。

解决办法两种，第一种使用闭包

```
for (var i = 1; i <= 5; i++) {
  (function(j) {
    setTimeout(function timer() {
      console.log(j);
    }, j * 1000);
  })(i);
}
```

第二种就是使用 `setTimeout` 的第三个参数

```
for ( var i=1; i<=5; i++) {
  setTimeout( function timer(j) {
    console.log( j );
  }, i*1000, i);
}
```

第三种就是使用 `let` 定义 `i` 了

```
for ( let i=1; i<=5; i++) {
  setTimeout( function timer() {
    console.log( i );
  }, i*1000 );
}
```

因为对于 `let` 来说，他会创建一个块级作用域，相当于

```
{ // 形成块级作用域
  let i = 0
  {
    let ii = i
    setTimeout( function timer() {
      console.log( ii );
    }, i*1000 );
  }
  i++
  {
    let ii = i
  }
  i++
  {
    let ii = i
  }
  ...
}
```



```
}
```

## 基本数据类型和引用类型在存储上的差别

前者存储在栈上，后者存储在堆上

## 浏览器 Eventloop 和 Node 中的有什么区别

众所周知 JS 是一门非阻塞单线程语言，因为在最初 JS 就是为了和浏览器交互而诞生的。如果 JS 是一门多线程的语言，我们在多个线程中处理 DOM 就可能会发生问题（一个线程中新加节点，另一个线程中删除节点），当然可以引入读写锁解决这个问题。

JS 在运行的过程中会产生执行环境，这些执行环境会被顺序的加入到执行栈中。如果遇到异步的代码，会被挂起并加入到 Task（有多种 task）队列中。一旦执行栈为空，Event Loop 就会从 Task 队列中拿出需要执行的代码并放入执行栈中执行，所以本质上来说 JS 中的异步还是同步行为。

javascript 复制代码

```
console.log('script start');

setTimeout(function() {
  console.log('setTimeout');
}, 0);

console.log('script end');
```

以上代码虽然 `setTimeout` 延时为 0，其实还是异步。这是因为 HTML5 标准规定这个函数第二个参数不得小于 4 毫秒，不足会自动增加。所以 `setTimeout` 还是会在 `script end` 之后打印。

不同的任务源会被分配到不同的 Task 队列中，任务源可以分为微任务（microtask）和宏任务（macrotask）。在 ES6 规范中，microtask 称为 `jobs`，macrotask 称为 `task`。

javascript 复制代码

```
console.log('script start');

setTimeout(function() {
  console.log('setTimeout');
}, 0);

new Promise((resolve) => {
  console.log('Promise')
})
```

```

    resolve()
  }).then(function() {
    console.log('promise1');
  }).then(function() {
    console.log('promise2');
  });

console.log('script end');
// script start => Promise => script end => promise1 => promise2 => setTimeout

```

以上代码虽然 `setTimeout` 写在 `Promise` 之前，但是因为 `Promise` 属于微任务而 `setTimeout` 属于宏任务，所以会有以上的打印。

微任务包括 `process.nextTick` , `promise` , `Object.observe` , `MutationObserver`

宏任务包括 `script` , `setTimeout` , `setInterval` , `setImmediate` , `I/O` , `UI rendering`

很多人有个误区，认为微任务快于宏任务，其实是错误的。因为宏任务中包括了 `script` ，浏览器会先执行一个宏任务，接下来有异步代码的话就先执行微任务。

所以正确的一次 Event loop 顺序是这样的

1. 执行同步代码，这属于宏任务
2. 执行栈为空，查询是否有微任务需要执行
3. 执行所有微任务
4. 必要的话渲染 UI
5. 然后开始下一轮 Event loop，执行宏任务中的异步代码

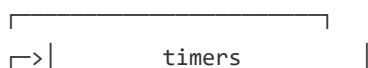
通过上述的 Event loop 顺序可知，如果宏任务中的异步代码有大量的计算并且需要操作 DOM 的话，为了更快的 界面响应，我们可以把操作 DOM 放入微任务中。

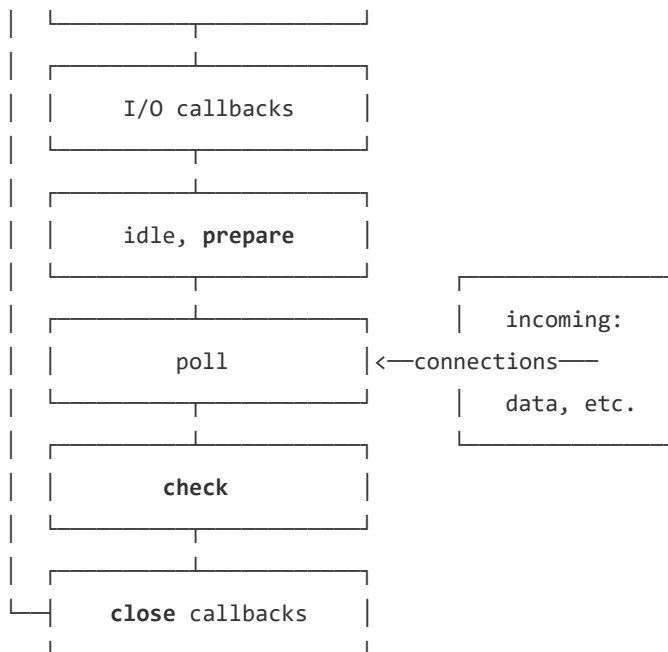
## Node 中的 Event loop

Node 中的 Event loop 和浏览器中的不相同。

Node 的 Event loop 分为6个阶段，它们会按照顺序反复运行

sql 复制代码





## timer

timers 阶段会执行 `setTimeout` 和 `setInterval`

一个 `timer` 指定的时间并不是准确时间，而是在达到这个时间后尽快执行回调，可能会因为系统正在执行别的事务而延迟。

下限的时间有一个范围： `[1, 2147483647]`，如果设定的时间不在这个范围，将被设置为1。

**\*\*I/O \*\***

I/O 阶段会执行除了 `close` 事件，定时器和 `setImmediate` 的回调

## idle, prepare

idle, prepare 阶段内部实现

## poll

poll 阶段很重要，这一阶段中，系统会做两件事情

1. 执行到点的定时器
2. 执行 poll 队列中的事件

并且当 poll 中没有定时器的情况下，会发现以下两件事情

- 如果 poll 队列不为空，会遍历回调队列并同步执行，直到队列为空或者系统限制
- 如果 poll 队列为空，会有两件事发生
  - 如果有 `setImmediate` 需要执行，poll 阶段会停止并且进入到 check 阶段执行 `setImmediate`
  - 如果没有 `setImmediate` 需要执行，会等待回调被加入到队列中并立即执行回调

如果有别的定时器需要被执行，会回到 timer 阶段执行回调。

## check

check 阶段执行 `setImmediate`

## close callbacks

close callbacks 阶段执行 close 事件

并且在 Node 中，有些情况下的定时器执行顺序是随机的

javascript 复制代码

```
setTimeout(() => {
  console.log('setTimeout');
}, 0);
setImmediate(() => {
  console.log('setImmediate');
})
// 这里可能会输出 setTimeout, setImmediate
// 可能也会相反的输出，这取决于性能
// 因为可能进入 event loop 用了不到 1 毫秒，这时候会执行 setImmediate
// 否则会执行 setTimeout
```

当然在这种情况下，执行顺序是相同的

javascript 复制代码

```
var fs = require('fs')

fs.readFile(__filename, () => {
  setTimeout(() => {
    console.log('timeout');
  }, 0);
  setImmediate(() => {
    console.log('immediate');
  });
});
// 因为 readFile 的回调在 poll 中执行
// 发现有 setImmediate，所以会立即跳到 check 阶段执行回调
```

```
// 再去 timer 阶段执行 setTimeout
// 所以以上输出一定是 setImmediate, setTimeout
```

上面介绍的都是 macrotask 的执行情况，microtask 会在以上每个阶段完成后立即执行。

javascript 复制代码

```
setTimeout(()=>{
  console.log('timer1')

  Promise.resolve().then(function() {
    console.log('promise1')
  })
}, 0)

setTimeout(()=>{
  console.log('timer2')

  Promise.resolve().then(function() {
    console.log('promise2')
  })
}, 0)

// 以上代码在浏览器和 node 中打印情况是不同的
// 浏览器中一定打印 timer1, promise1, timer2, promise2
// node 中可能打印 timer1, timer2, promise1, promise2
// 也可能打印 timer1, promise1, timer2, promise2
```

Node 中的 `process.nextTick` 会先于其他 microtask 执行。

javascript 复制代码

```
setTimeout(() => {
  console.log("timer1");

  Promise.resolve().then(function() {
    console.log("promise1");
  });
}, 0);

process.nextTick(() => {
  console.log("nextTick");
});
// nextTick, timer1, promise1
```

## setTimeout 倒计时误差

JS 是单线程的，所以 `setTimeout` 的误差其实是无法被完全解决的，原因有很多，可能是回调中的，有可能是浏览器中的各种事件导致。这也是为什么页面开久了，定时器会不准的原因，当然我们可以通过一定的办法去减少这个误差。

以下是一个相对准备的倒计时实现

ini 复制代码

```
var period = 60 * 1000 * 60 * 2
var startTime = new Date().getTime();
var count = 0
var end = new Date().getTime() + period
var interval = 1000
var currentInterval = interval

function loop() {
  count++
  var offset = new Date().getTime() - (startTime + count * interval); // 代码执行所消耗的时间
  var diff = end - new Date().getTime()
  var h = Math.floor(diff / (60 * 1000 * 60))
  var hdiff = diff % (60 * 1000 * 60)
  var m = Math.floor(hdiff / (60 * 1000))
  var mdiff = hdiff % (60 * 1000)
  var s = mdiff / (1000)
  var sCeil = Math.ceil(s)
  var sFloor = Math.floor(s)
  currentInterval = interval - offset // 得到下一次循环所消耗的时间
  console.log('时: '+h, '分: '+m, '毫秒: '+s, '秒向上取整: '+sCeil, '代码执行时间: '+offset, '下次循环间

  setTimeout(loop, currentInterval)
}

setTimeout(loop, currentInterval)
```

## 防抖

你是否在日常开发中遇到一个问题，在滚动事件中需要做个复杂计算或者实现一个按钮的防二次点击操作。

这些需求都可以通过函数防抖动来实现。尤其是第一个需求，如果在频繁的事件回调中做复杂计算，很有可能导致页面卡顿，不如将多次计算合并为一次计算，只在一个精确点做操作。

PS：防抖和节流的作用都是防止函数多次调用。区别在于，假设一个用户一直触发这个函数，且每次触发函数的间隔小于wait，防抖的情况下只会调用一次，而节流的情况会每隔一定时间

(参数wait) 调用函数。

我们先来看一个袖珍版的防抖理解一下防抖的实现：

javascript 复制代码

```
// func是用户传入需要防抖的函数
// wait是等待时间
const debounce = (func, wait = 50) => {
  // 缓存一个定时器id
  let timer = 0
  // 这里返回的函数是每次用户实际调用的防抖函数
  // 如果已经设定过定时器了就清空上一次的定时器
  // 开始一个新的定时器，延迟执行用户传入的方法
  return function(...args) {
    if (timer) clearTimeout(timer)
    timer = setTimeout(() => {
      func.apply(this, args)
    }, wait)
  }
}
// 不难看出如果用户调用该函数的间隔小于wait的情况下，上一次的时间还未到就被清除了，并不会执行函数
```

这是一个简单版的防抖，但是有缺陷，这个防抖只能在最后调用。一般的防抖会有immediate选项，表示是否立即调用。这两者的区别，举个栗子来说：

- 例如在搜索引擎搜索问题的时候，我们当然是希望用户输入完最后一个字才调用查询接口，这个时候适用 **延迟执行** 的防抖函数，它总是在一连串（间隔小于wait的）函数触发之后调用。
- 例如用户给interviewMap点star的时候，我们希望用户点第一下的时候就去调用接口，并且成功之后改变star按钮的样子，用户就可以立马得到反馈是否star成功了，这个情况适用 **立即执行** 的防抖函数，它总是在第一次调用，并且下一次调用必须与前一次调用的时间间隔大于wait才会触发。

下面我们来实现一个带有立即执行选项的防抖函数

javascript 复制代码

```
// 这个是用来获取当前时间戳的
function now() {
  return +new Date()
}
/**
 * 防抖函数，返回函数连续调用时，空闲时间必须大于或等于 wait，func 才会执行
 *
 * @param {function} func      回调函数
 * @param {number} wait        表示时间窗口的间隔
```

```

* @param {boolean} immediate  设置为ture时，是否立即调用函数
* @return {function}          返回客户调用函数
*/
function debounce (func, wait = 50, immediate = true) {
  let timer, context, args

  // 延迟执行函数
  const later = () => setTimeout(() => {
    // 延迟函数执行完毕，清空缓存的定时器序号
    timer = null
    // 延迟执行的情况下，函数会在延迟函数中执行
    // 使用到之前缓存的参数和上下文
    if (!immediate) {
      func.apply(context, args)
      context = args = null
    }
  }, wait)

  // 这里返回的函数是每次实际调用的函数
  return function(...params) {
    // 如果没有创建延迟执行函数（later），就创建一个
    if (!timer) {
      timer = later()
      // 如果是立即执行，调用函数
      // 否则缓存参数和调用上下文
      if (immediate) {
        func.apply(this, params)
      } else {
        context = this
        args = params
      }
    }
    // 如果已有延迟执行函数（later），调用的时候清除原来的并重新设定一个
    // 这样做延迟函数会重新计时
    } else {
      clearTimeout(timer)
      timer = later()
    }
  }
}

```

整体函数实现的不难，总结一下。

- 对于按钮防点击来说的实现：如果函数是立即执行的，就立即调用，如果函数是延迟执行的，就缓存上下文和参数，放到延迟函数中去执行。一旦我开始一个定时器，只要我定时器还在，你每次点击我都重新计时。一旦你点累了，定时器时间到，定时器重置为 `null`，就可以再次点击了。



- 对于延时执行函数来说的实现：清除定时器ID，如果是延迟调用就调用函数

## 数组降维

javascript 复制代码

```
[1, [2], 3].flatMap((v) => v + 1)
// -> [2, 3, 4]
```

如果想将一个多维数组彻底的降维，可以这样实现

ini 复制代码

```
const flattenDeep = (arr) => Array.isArray(arr)
  ? arr.reduce( (a, b) => [...a, ...flattenDeep(b)] , [])
  : [arr]

flattenDeep([1, [[2], [3, [4]], 5]])
```

## 深拷贝

这个问题通常可以通过 `JSON.parse(JSON.stringify(object))` 来解决。

css 复制代码

```
let a = {
  age: 1,
  jobs: {
    first: 'FE'
  }
}

let b = JSON.parse(JSON.stringify(a))
a.jobs.first = 'native'
console.log(b.jobs.first) // FE
```

但是该方法也是有局限性的：

- 会忽略 `undefined`
- 会忽略 `symbol`
- 不能序列化函数
- 不能解决循环引用的对象

ini 复制代码

```
let obj = {
  a: 1,
  b: {
    c: 2,
```

```

    d: 3,
  },
}
obj.c = obj.b
obj.e = obj.a
obj.b.c = obj.c
obj.b.d = obj.b
obj.b.e = obj.b.c
let newObj = JSON.parse(JSON.stringify(obj))
console.log(newObj)

```

如果你有这么一个循环引用对象，你会发现你不能通过该方法深拷贝

✖ ▶ Uncaught TypeError: Converting circular structure to JSON  
 at JSON.stringify (<anonymous>)  
 at <anonymous>:13:30 @稀土掘金技术社区

在遇到函数、`undefined` 或者 `symbol` 的时候，该对象也不能正常的序列化

css 复制代码

```

let a = {
  age: undefined,
  sex: Symbol('male'),
  jobs: function() {},
  name: 'yck'
}
let b = JSON.parse(JSON.stringify(a))
console.log(b) // {name: "yck"}

```

你会发现在上述情况中，该方法会忽略掉函数和 `undefined`。

但是在通常情况下，复杂数据都是可以序列化的，所以这个函数可以解决大部分问题，并且该函数是内置函数中处理深拷贝性能最快的。当然如果你的数据中含有以上三种情况下，可以使用 [lodash 的深拷贝函数](#)。

如果你所需拷贝的对象含有内置类型并且不包含函数，可以使用 `MessageChannel`

javascript 复制代码

```

function structuralClone(obj) {
  return new Promise(resolve => {
    const {port1, port2} = new MessageChannel();
    port2.onmessage = ev => resolve(ev.data);
    port1.postMessage(obj);
  });
}

```

```
var obj = {a: 1, b: {
  c: b
}}
// 注意该方法是异步的
// 可以处理 undefined 和循环引用对象
(async () => {
  const clone = await structuralClone(obj)
})();
```

## typeof 于 instanceof 区别

**typeof** 对于基本类型，除了 **null** 都可以显示正确的类型

```
typeof 1 // 'number'
typeof '1' // 'string'
typeof undefined // 'undefined'
typeof true // 'boolean'
typeof Symbol() // 'symbol'
typeof b // b 没有声明，但是还会显示 undefined
```

javascript 复制代码

**typeof** 对于对象，除了函数都会显示 **object**

```
typeof [] // 'object'
typeof {} // 'object'
typeof console.log // 'function'
```

javascript 复制代码

对于 **null** 来说，虽然它是基本类型，但是会显示 **object**，这是一个存在很久的 Bug

```
typeof null // 'object'
```

csharp 复制代码

PS：为什么会出现这种情况呢？因为在 JS 的最初版本中，使用的是 32 位系统，为了性能考虑使用低位存储了变量的类型信息，**000** 开头代表是对象，然而 **null** 表示为全零，所以将它错误的判断为 **object**。虽然现在的内部类型判断代码已经改变了，但是对于这个 Bug 却一直流传下来。

**instanceof** 可以正确的判断对象的类型，因为内部机制是通过判断对象的原型链中是不是能找到类型的 **prototype**。

我们也可以试着实现一下 **instanceof**

```
function instanceof(left, right) {  
  // 获得类型的原型  
  let prototype = right.prototype  
  // 获得对象的原型  
  left = left.__proto__  
  // 判断对象的类型是否等于类型的原型  
  while (true) {  
    if (left === null)  
      return false  
    if (prototype === left)  
      return true  
    left = left.__proto__  
  }  
}
```

## 浏览器相关

### cookie和localSorage、session、indexDB 的区别

特性	cookie	localStorage	sessionStorage	indexDB
数据生命周期	一般由服务器生成，可以设置过期时间	除非被清理，否则一直存在	页面关闭就清理	除非被清理，否则一直存在
数据存储大小	4K	5M	5M	无限
与服务端通信	每次都会携带在header 中，对于请求性能影响	不参与	不参与	不参与

从上表可以看到，`cookie` 已经不建议用于存储。如果没有大量数据存储需求的话，可以使用 `localStorage` 和 `sessionStorage` 。对于不怎么改变的数据尽量使用 `localStorage` 存储，否则可以用 `sessionStorage` 存储。

对于 `cookie` ，我们还需要注意安全性。

属性	作用
value	如果用于保存用户登录态，应该将该值加密，不能使用明文的用户标识
http-only	不能通过 JS 访问 Cookie，减少 XSS 攻击
secure	只能在协议为 HTTPS 的请求中携带
same-site	规定浏览器不能在跨域请求中携带 Cookie，减少 CSRF 攻击

## 怎么判断页面是否加载完成？

Load 事件触发代表页面中的 DOM，CSS，JS，图片已经全部加载完毕。

DOMContentLoaded 事件触发代表初始的 HTML 被完全加载和解析，不需要等待 CSS，JS，图片加载。

## 如何解决跨域

因为浏览器出于安全考虑，有同源策略。也就是说，如果协议、域名或者端口有一个不同就是跨域，Ajax 请求会失败。

我们可以通过以下几种常用方法解决跨域的问题

## JSONP

JSONP 的原理很简单，就是利用 `<script>` 标签没有跨域限制的漏洞。通过 `<script>` 标签指向一个需要访问的地址并提供一个回调函数来接收数据当需要通讯时。

xml 复制代码

```
<script src="http://domain/api?param1=a&param2=b&callback=jsonp"></script>
<script>
    function jsonp(data) {
        console.log(data)
    }
</script>
```

JSONP 使用简单且兼容性不错，但是只限于 `get` 请求。

在开发中可能会遇到多个 JSONP 请求的回调函数名是相同的，这时候就需要自己封装一个 JSONP，以下是简单实现

ini 复制代码

```
function jsonp(url, jsonpCallback, success) {
  let script = document.createElement("script");
  script.src = url;
  script.async = true;
  script.type = "text/javascript";
  window[jsonpCallback] = function(data) {
    success && success(data);
  };
  document.body.appendChild(script);
}
jsonp(
  "http://xxx",
  "callback",
  function(value) {
    console.log(value);
  }
);
```

## CORS

CORS需要浏览器和后端同时支持。IE 8 和 9 需要通过 `XDomainRequest` 来实现。

浏览器会自动进行 CORS 通信，实现CORS通信的关键是后端。只要后端实现了 CORS，就实现了跨域。

服务端设置 `Access-Control-Allow-Origin` 就可以开启 CORS。该属性表示哪些域名可以访问资源，如果设置通配符则表示所有网站都可以访问资源。

## document.domain

该方式只能用于二级域名相同的情况下，比如 `a.test.com` 和 `b.test.com` 适用于该方式。

只需要给页面添加 `document.domain = 'test.com'` 表示二级域名都相同就可以实现跨域

## postMessage

这种方式通常用于获取嵌入页面中的第三方页面数据。一个页面发送消息，另一个页面判断来源并接收消息

javascript 复制代码

```
// 发送消息端
window.parent.postMessage('message', 'http://test.com');
// 接收消息端
var mc = new MessageChannel();
mc.addEventListener('message', (event) => {
  var origin = event.origin || event.originalEvent.origin;
  if (origin === 'http://test.com') {
    console.log('验证通过')
  }
});
```

## 什么是事件代理

如果一个节点中的子节点是动态生成的，那么子节点需要注册事件的话应该注册在父节点上

xml 复制代码

```
<ul id="ul">
  <li>1</li>
  <li>2</li>
  <li>3</li>
  <li>4</li>
  <li>5</li>
</ul>
<script>
  let ul = document.querySelector('#ul')
  ul.addEventListener('click', (event) => {
    console.log(event.target);
  })
</script>
```

事件代理的方式相对于直接给目标注册事件来说，有以下优点

- 节省内存
- 不需要给子节点注销事件

## Service worker

Service workers 本质上充当Web应用程序与浏览器之间的代理服务器，也可以在网络可用时作为浏览器和网络间的代理。它们旨在（除其他之外）使得能够创建有效的离线

体验，拦截网络请求并基于网络是否可用以及更新的资源是否驻留在服务器上来采取适当的动作。他们还允许访问推送通知和后台同步API。

目前该技术通常用来做缓存文件，提高首屏速度，可以试着来实现这个功能。

javascript 复制代码

```
// index.js
if (navigator.serviceWorker) {
  navigator.serviceWorker
    .register("sw.js")
    .then(function(registration) {
      console.log("service worker 注册成功");
    })
    .catch(function(err) {
      console.log("servcie worker 注册失败");
    });
}

// sw.js
// 监听 `install` 事件，回调中缓存所需文件
self.addEventListener("install", e => {
  e.waitUntil(
    caches.open("my-cache").then(function(cache) {
      return cache.addAll(["./index.html", "./index.js"]);
    })
  );
});

// 拦截所有请求事件
// 如果缓存中已经有请求的数据就直接用缓存，否则去请求数据
self.addEventListener("fetch", e => {
  e.respondWith(
    caches.match(e.request).then(function(response) {
      if (response) {
        return response;
      }
      console.log("fetch source");
    })
  );
});
```

打开页面，可以在开发者工具中的 **Application** 看到 Service Worker 已经启动了



## 浏览器缓存



缓存对于前端性能优化来说是个很重要的点，良好的缓存策略可以降低资源的重复加载提高网页的整体加载速度。

通常浏览器缓存策略分为两种：强缓存和协商缓存。

## 强缓存

实现强缓存可以通过两种响应头实现：`Expires` 和 `Cache-Control`。强缓存表示在缓存期间不需要请求，`state code` 为 200

```
Expires: Wed, 22 Oct 2018 08:41:00 GMT
```

yaml 复制代码

`Expires` 是 HTTP / 1.0 的产物，表示资源会在 `Wed, 22 Oct 2018 08:41:00 GMT` 后过期，需要再次请求。并且 `Expires` 受限于本地时间，如果修改了本地时间，可能会造成缓存失效。

```
Cache-control: max-age=30
```

ini 复制代码

`Cache-Control` 出现于 HTTP / 1.1，优先级高于 `Expires`。该属性表示资源会在 30 秒后过期，需要再次请求。

## 协商缓存

如果缓存过期了，我们就可以使用协商缓存来解决问题。协商缓存需要请求，如果缓存有效会返回 304。

协商缓存需要客户端和服务端共同实现，和强缓存一样，也有两种实现方式。

## Last-Modified 和 If-Modified-Since

`Last-Modified` 表示本地文件最后修改日期，`If-Modified-Since` 会将 `Last-Modified` 的值发送给服务器，询问服务器在该日期后资源是否有更新，有更新的话就会将新的资源发送回来。

但是如果在本地打开缓存文件，就会造成 `Last-Modified` 被修改，所以在 HTTP / 1.1 出现了 `ETag`。

## ETag 和 If-None-Match

ETag 类似于文件指纹，If-None-Match 会将当前 ETag 发送给服务器，询问该资源 ETag 是否变动，有变动的话就将新的资源发送回来。并且 ETag 优先级比 Last-Modified 高。

## 选择合适的缓存策略

对于大部分的场景都可以使用强缓存配合协商缓存解决，但是在一些特殊的地方可能需要选择特殊的缓存策略

- 对于某些不需要缓存的资源，可以使用 `Cache-control: no-store`，表示该资源不需要缓存
- 对于频繁变动的资源，可以使用 `Cache-Control: no-cache` 并配合 ETag 使用，表示该资源已被缓存，但是每次都会发送请求询问资源是否更新。
- 对于代码文件来说，通常使用 `Cache-Control: max-age=31536000` 并配合策略缓存使用，然后对文件进行指纹处理，一旦文件名变动就会立刻下载新的文件。

## 浏览器性能问题

### 重绘 (Repaint) 和回流 (Reflow)

重绘和回流是渲染步骤中的一小节，但是这两个步骤对于性能影响很大。

- 重绘是当节点需要更改外观而不会影响布局的，比如改变 `color` 就叫称为重绘
- 回流是布局或者几何属性需要改变就称为回流。

回流必定会发生重绘，重绘不一定会引发回流。回流所需的成本比重绘高的多，改变深层次的节点很可能导致父节点的一系列回流。

所以以下几个动作可能会导致性能问题：

- 改变 window 大小
- 改变字体
- 添加或删除样式
- 文字改变
- 定位或者浮动
- 盒模型

很多人不知道的是，重绘和回流其实和 Event loop 有关。

1. 当 Event loop 执行完 Microtasks 后，会判断 document 是否需要更新。因为浏览器是 60Hz 的刷新率，每 16ms 才会更新一次。
2. 然后判断是否有 `resize` 或者 `scroll`，有的话会去触发事件，所以 `resize` 和 `scroll` 事件也是至少 16ms 才会触发一次，并且自带节流功能。
3. 判断是否触发了 media query
4. 更新动画并且发送事件
5. 判断是否有全屏操作事件
6. 执行 `requestAnimationFrame` 回调
7. 执行 `IntersectionObserver` 回调，该方法用于判断元素是否可见，可以用于懒加载上，但是兼容性不好
8. 更新界面
9. 以上就是一帧中可能会做的事情。如果在一帧中有空闲时间，就会去执行 `requestIdleCallback` 回调。

以上内容来自于 [HTML 文档](#)

## 减少重绘和回流

- 使用 `translate` 替代 `top`

xml 复制代码

```
<div class="test"></div>
<style>
    .test {
        position: absolute;
        top: 10px;
        width: 100px;
        height: 100px;
        background: red;
    }
</style>
<script>
    setTimeout(() => {
        // 引起回流
        document.querySelector('.test').style.top = '100px'
    }, 1000)
</script>
```

- 使用 `visibility` 替换 `display: none`，因为前者只会引起重绘，后者会引发回流（改变了布局）
- 把 DOM 离线后修改，比如：先把 DOM 给 `display:none`（有一次 Reflow），然后你修改 100 次，然后再把它显示出来
- 不要把 DOM 结点的属性值放在一个循环里当成循环里的变量

javascript 复制代码

```
for(let i = 0; i < 1000; i++) {  
    // 获取 offsetTop 会导致回流，因为需要去获取正确的值  
    console.log(document.querySelector('.test').style.offsetTop)  
}
```

- 不要使用 table 布局，可能很小的一个小改动会造成整个 table 的重新布局
- 动画实现的速度的选择，动画速度越快，回流次数越多，也可以选择使用 `requestAnimationFrame`
- CSS 选择符从右往左匹配查找，避免 DOM 深度过深
- 将频繁运行的动画变为图层，图层能够阻止该节点回流影响别的元素。比如对于 `video` 标签，浏览器会自动将该节点变为图层。



## 图片优化

### 计算图片大小

对于一张 100 \* 100 像素的图片来说，图像上有 10000 个像素点，如果每个像素的值是 RGBA 存储的话，那么也就是说每个像素有 4 个通道，每个通道 1 个字节（8 位 = 1 个字节），所以该图片大小大概为 39KB（ $10000 * 1 * 4 / 1024$ ）。

但是在实际项目中，一张图片可能并不需要使用那么多颜色去显示，我们可以通过减少每个像素的调色板来相应缩小图片的大小。

了解了如何计算图片大小的知识，那么对于如何优化图片，想必大家已经有 2 个思路了：

- 减少像素点
- 减少每个像素点能够显示的颜色

## 图片加载优化

1. 不用图片。很多时候会使用到很多修饰类图片，其实这类修饰图片完全可以用 CSS 去代替。
2. 对于移动端来说，屏幕宽度就那么点，完全没有必要去加载原图浪费带宽。一般图片都用 CDN 加载，可以计算出适配屏幕的宽度，然后去请求相应裁剪好的图片。
3. 小图使用 base64 格式
4. 将多个图标文件整合到一张图片中（雪碧图）
5. 选择正确的图片格式：
  - 对于能够显示 WebP 格式的浏览器尽量使用 WebP 格式。因为 WebP 格式具有更好的图像数据压缩算法，能带来更小的图片体积，而且拥有肉眼识别无差异的图像质量，缺点就是兼容性并不好
  - 小图使用 PNG，其实对于大部分图标这类图片，完全可以使用 SVG 代替
  - 照片使用 JPEG

## 其他文件优化

- CSS 文件放在 `head` 中
- 服务端开启文件压缩功能
- 将 `script` 标签放在 `body` 底部，因为 JS 文件执行会阻塞渲染。当然也可以把 `script` 标签放在任意位置然后加上 `defer`，表示该文件会并行下载，但是会放到 HTML 解析完成后顺序执行。对于没有任何依赖的 JS 文件可以加上 `async`，表示加载和渲染后续文档元素的过程将和 JS 文件的加载与执行并行无序进行。
- 执行 JS 代码过长会卡住渲染，对于需要很多时间计算的代码可以考虑使用 `Webworker`。  
`Webworker` 可以让我们另开一个线程执行脚本而不影响渲染。

## CDN

静态资源尽量使用 CDN 加载，由于浏览器对于单个域名有并发请求上限，可以考虑使用多个 CDN 域名。对于 CDN 加载静态资源需要注意 CDN 域名要与主站不同，否则每次请求都会带上主站的 Cookie。

## 使用 Webpack 优化项目

- 对于 Webpack4，打包项目使用 production 模式，这样会自动开启代码压缩
- 使用 ES6 模块来开启 tree shaking，这个技术可以移除没有使用的代码
- 优化图片，对于小图可以使用 base64 的方式写入文件中
- 按照路由拆分代码，实现按需加载
- 给打包出来的文件名添加哈希，实现浏览器缓存文件

## Webpack

---

### 优化打包速度

- 减少文件搜索范围
  - 比如通过别名
  - loader 的 test, include & exclude
- Webpack4 默认压缩并行
- Happypack 并发调用
- babel 也可以缓存编译

### Babel 原理

本质就是编译器，当代码转为字符串生成 AST，对 AST 进行转变最后再生成新的代码

- 分为三步：词法分析生成 Token，语法分析生成 AST，遍历 AST，根据插件变换相应的节点，最后把 AST 转换为代码

### 如何实现一个插件

- 调用插件 apply 函数传入 compiler 对象
- 通过 compiler 对象监听事件

比如你想实现一个编译结束退出命令的插件

javascript 复制代码

```
class BuildEndPlugin {
  apply (compiler) {
    const afterEmit = (compilation, cb) => {
      cb()
      setTimeout(function () {
        process.exit(0)
      }, 1000)
    }
  }
}
```

```
    compiler.plugin('after-emit', afterEmit)
  }
}

module.exports = BuildEndPlugin
```

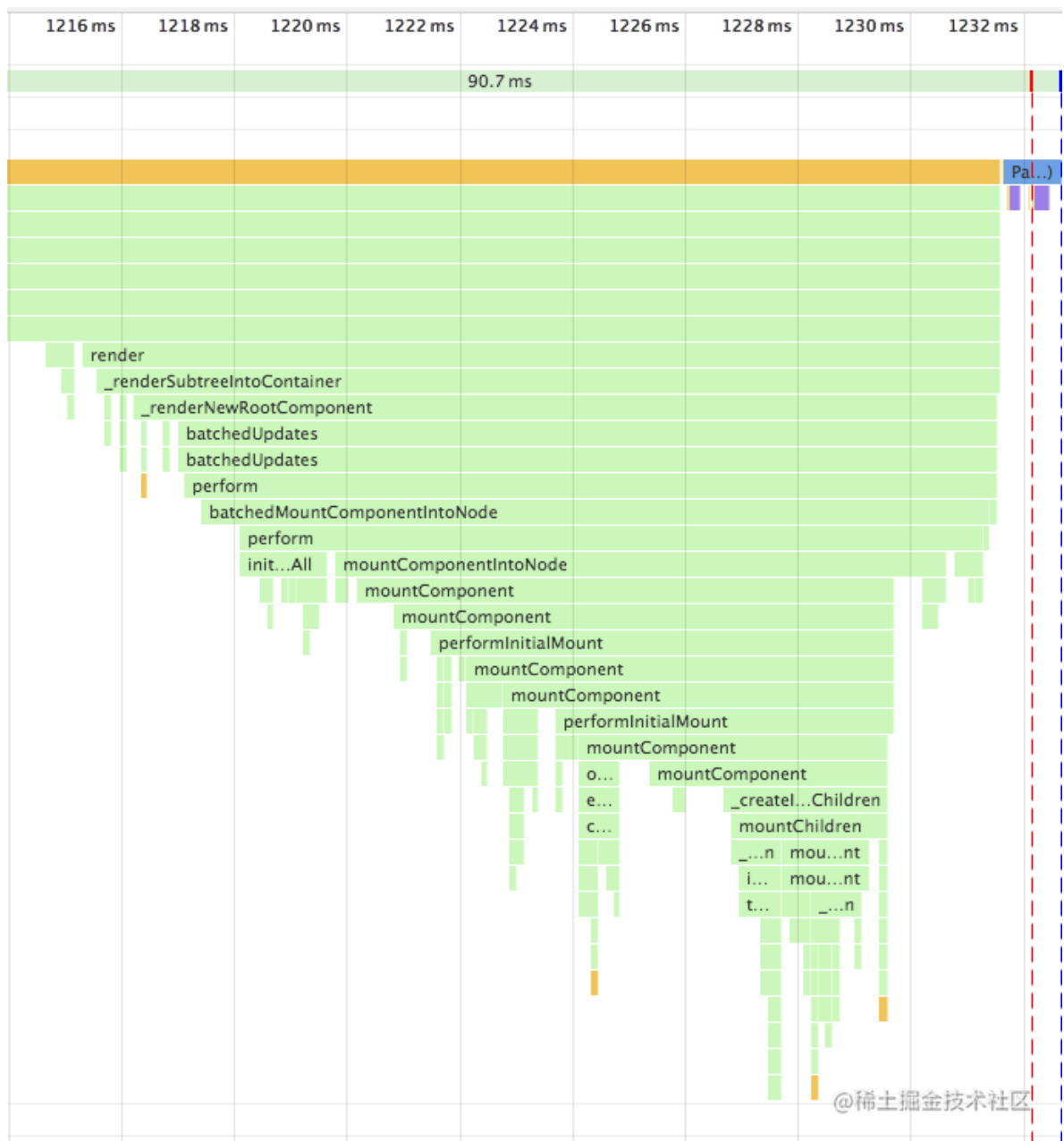
## 框架

---

### React 生命周期

在 V16 版本中引入了 Fiber 机制。这个机制一定程度上的影响了部分生命周期的调用，并且也引入了新的 2 个 API 来解决问题。

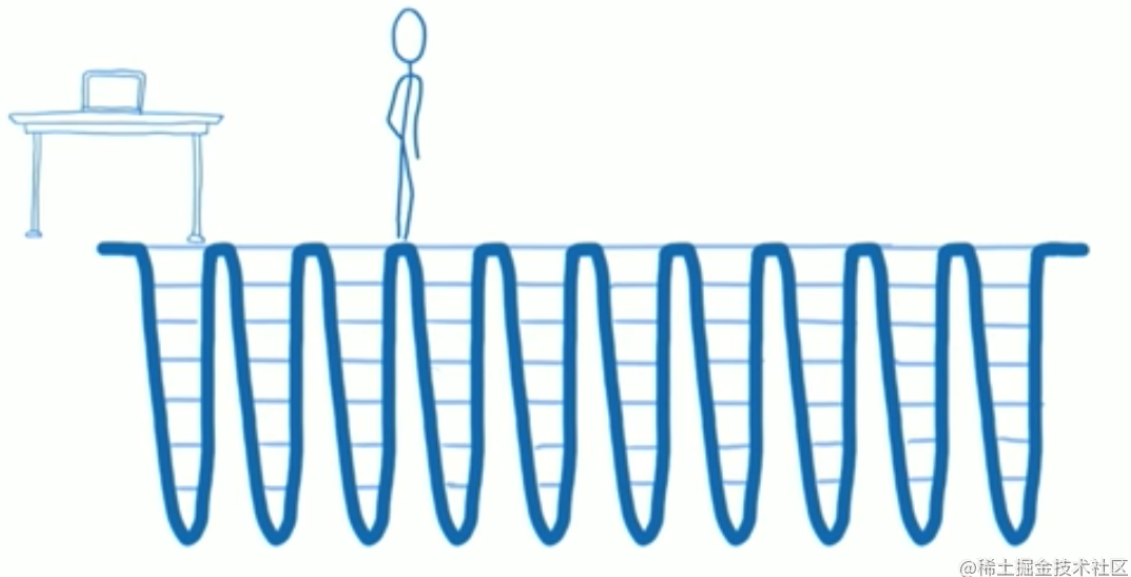
在之前的版本中，如果你拥有一个很复杂的复合组件，然后改动了最上层组件的 `state`，那么调用栈可能会很长



调用栈过长，再加上中间进行了复杂的操作，就可能导致长时间阻塞主线程，带来不好的用户体验。Fiber 就是为了解决该问题而生。

Fiber 本质上是一个虚拟的堆栈帧，新的调度器会按照优先级自由调度这些帧，从而将之前的同步渲染改成了异步渲染，在不影响体验的情况下去分段计算更新。





对于如何区别优先级，React 有自己的一套逻辑。对于动画这种实时性很高的东西，也就是 16 ms 必须渲染一次保证不卡顿的情况下，React 会每 16 ms（以内） 暂停一下更新，返回来继续渲染动画。

对于异步渲染，现在渲染有两个阶段：`reconciliation` 和 `commit`。前者过程是可以打断的，后者不能暂停，会一直更新界面直到完成。

### Reconciliation 阶段

- `componentWillMount`
- `componentWillReceiveProps`
- `shouldComponentUpdate`
- `componentWillUpdate`

### Commit 阶段

- `componentDidMount`
- `componentDidUpdate`
- `componentWillUnmount`

因为 `reconciliation` 阶段是可以被打断的，所以 `reconciliation` 阶段会执行的生命周期函数就可能会出现调用多次的情况，从而引起 Bug。所以对于 `reconciliation` 阶段调用的几个函数，除了 `shouldComponentUpdate` 以外，其他都应该避免去使用，并且 V16 中也引入了新的 API 来解决这个问题。

`getDerivedStateFromProps` 用于替换 `componentWillReceiveProps`，该函数会在初始化和 `update` 时被调用

scala 复制代码

```
class ExampleComponent extends React.Component {  
  // Initialize state in constructor,  
  // Or with a property initializer.  
  state = {};  
  
  static getDerivedStateFromProps(nextProps, prevState) {  
    if (prevState.someMirroredValue !== nextProps.someValue) {  
      return {  
        derivedData: computeDerivedState(nextProps),  
        someMirroredValue: nextProps.someValue  
      };  
    }  
  
    // Return null to indicate no change to state.  
    return null;  
  }  
}
```

`getSnapshotBeforeUpdate` 用于替换 `componentWillUpdate`，该函数会在 `update` 后 DOM 更新前被调用，用于读取最新的 DOM 数据。

## V16 生命周期函数用法建议

javascript 复制代码

```
class ExampleComponent extends React.Component {  
  // 用于初始化 state  
  constructor() {}  
  // 用于替换 `componentWillReceiveProps`，该函数会在初始化和 `update` 时被调用  
  // 因为该函数是静态函数，所以取不到 `this`  
  // 如果需要对比 `prevProps` 需要单独在 `state` 中维护  
  static getDerivedStateFromProps(nextProps, prevState) {}  
  // 判断是否需要更新组件，多用于组件性能优化  
  shouldComponentUpdate(nextProps, nextState) {}  
  // 组件挂载后调用  
  // 可以在该函数中进行请求或者订阅  
  componentDidMount() {}  
  // 用于获得最新的 DOM 数据  
  getSnapshotBeforeUpdate() {}  
  // 组件即将销毁  
  // 可以在此处移除订阅，定时器等  
  componentWillUnmount() {}  
  // 组件销毁后调用  
  componentDidUnmount() {}  
}
```

```

// 组件更新后调用
componentDidUpdate() {}
// 渲染组件函数
render() {}
// 以下函数不建议使用
UNSAFE_componentWillMount() {}
UNSAFE_componentWillUpdate(nextProps, nextState) {}
UNSAFE_componentWillReceiveProps(nextProps) {}
}

```

## setState

`setState` 在 React 中是经常使用的一个 API，但是它存在一些问题，可能会导致犯错，核心原因就是因为这个 API 是异步的。

首先 `setState` 的调用并不会马上引起 `state` 的改变，并且如果你一次调用了多个 `setState`，那么结果可能并不如你期待的一样。

kotlin 复制代码

```

handle() {
  // 初始化 `count` 为 0
  console.log(this.state.count) // -> 0
  this.setState({ count: this.state.count + 1 })
  this.setState({ count: this.state.count + 1 })
  this.setState({ count: this.state.count + 1 })
  console.log(this.state.count) // -> 0
}

```

第一，两次的打印都为 0，因为 `setState` 是个异步 API，只有同步代码运行完毕才会执行。

`setState` 异步的原因我认为在于，`setState` 可能会导致 DOM 的重绘，如果调用一次就马上去进行重绘，那么调用多次就会造成不必要的性能损失。设计成异步的话，就可以将多次调用放入一个队列中，在恰当的时候统一进行更新过程。

第二，虽然调用了三次 `setState`，但是 `count` 的值还是为 1。因为多次调用会合并为一次，只有当更新结束后 `state` 才会改变，三次调用等同于如下代码

kotlin 复制代码

```

Object.assign(
  {},
  { count: this.state.count + 1 },
  { count: this.state.count + 1 },
  { count: this.state.count + 1 },

```

)

当然你也可以通过以下方式来实现调用三次 `setState` 使得 `count` 为 3

javascript 复制代码

```
handle() {
  this.setState((prevState) => ({ count: prevState.count + 1 }));
  this.setState((prevState) => ({ count: prevState.count + 1 }));
  this.setState((prevState) => ({ count: prevState.count + 1 }));
}
```

如果你想在每次调用 `setState` 后获得正确的 `state` , 可以通过如下代码实现

javascript 复制代码

```
handle() {
  this.setState((prevState) => ({ count: prevState.count + 1 })), () => {
    console.log(this.state)
  })
}
```

## Vue的 nextTick 原理

`nextTick` 可以让我们在下次 DOM 更新循环结束之后执行延迟回调, 用于获得更新后的 DOM。

在 Vue 2.4 之前都是使用的 `microtasks`, 但是 `microtasks` 的优先级过高, 在某些情况下可能会出现比事件冒泡更快的情况, 但如果都使用 `macrotasks` 又可能会出现渲染的性能问题。所以在新版本中, 会默认使用 `microtasks`, 但在特殊情况下会使用 `macrotasks`, 比如 `v-on`。

对于实现 `macrotasks` , 会先判断是否能使用 `setImmediate` , 不能的话降级为 `MessageChannel` , 以上都不行的话就使用 `setTimeout`

javascript 复制代码

```
if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
  macroTimerFunc = () => {
    setImmediate(flushCallbacks)
  }
} else if (
  typeof MessageChannel !== 'undefined' &&
  (isNative(MessageChannel) ||
    // PhantomJS
    MessageChannel.toString() === '[object MessageChannelConstructor]')
) {
  const channel = new MessageChannel()
```

```

const port = channel.port2
channel.port1.onmessage = flushCallbacks
macroTimerFunc = () => {
  port.postMessage(1)
}
} else {
  /* istanbul ignore next */
  macroTimerFunc = () => {
    setTimeout(flushCallbacks, 0)
  }
}
}

```

**nextTick** 同时也支持 Promise 的使用，会判断是否实现了 Promise

javascript 复制代码

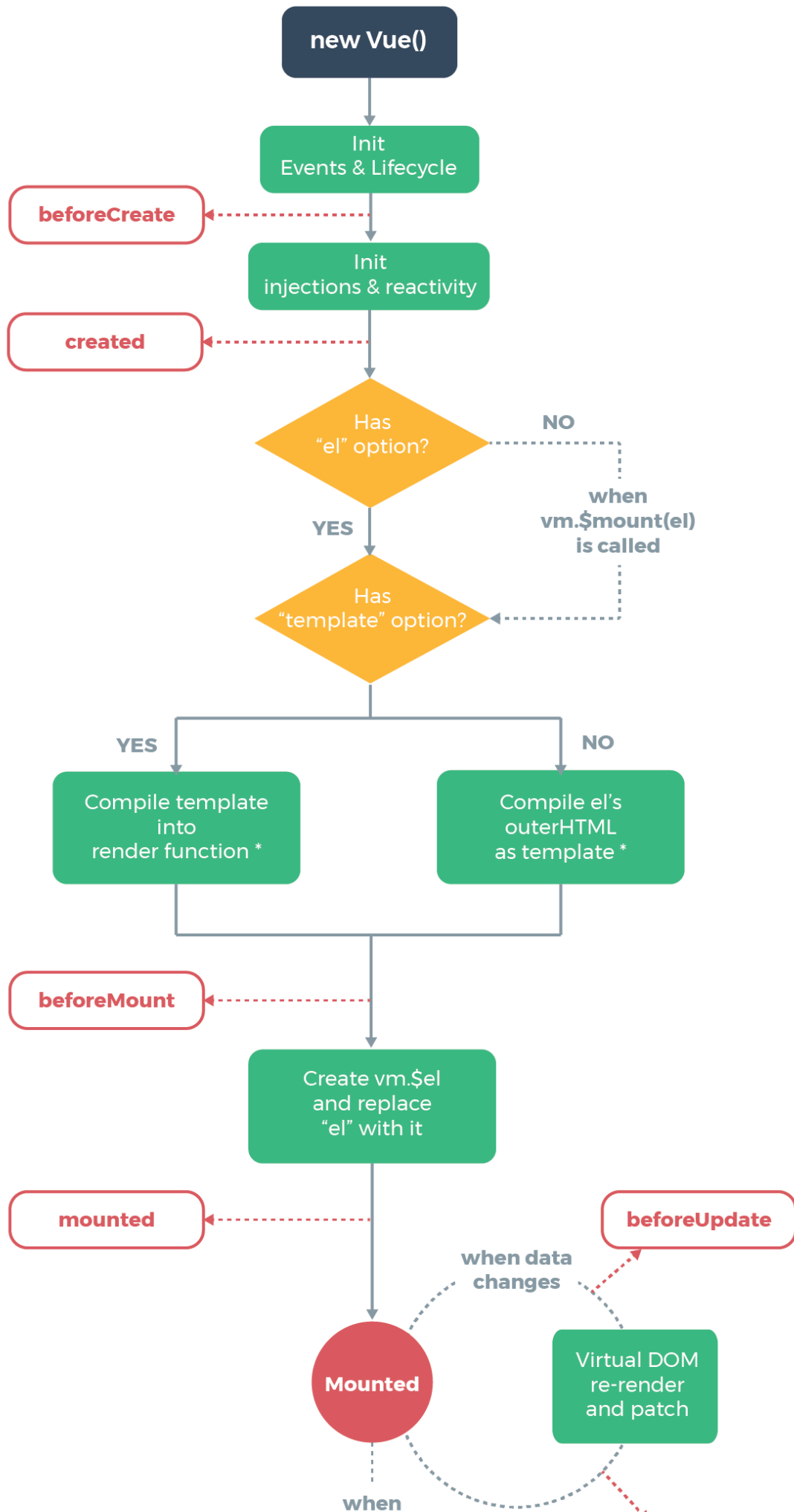
```

export function nextTick(cb?: Function, ctx?: Object) {
  let _resolve
  // 将回调函数整合进一个数组中
  callbacks.push(() => {
    if (cb) {
      try {
        cb.call(ctx)
      } catch (e) {
        handleError(e, ctx, 'nextTick')
      }
    } else if (_resolve) {
      _resolve(ctx)
    }
  })
  if (!pending) {
    pending = true
    if (useMacroTask) {
      macroTimerFunc()
    } else {
      microTimerFunc()
    }
  }
  // 判断是否可以使用 Promise
  // 可以的话给 _resolve 赋值
  // 这样回调函数就能以 promise 的方式调用
  if (!cb && typeof Promise !== 'undefined') {
    return new Promise(resolve => {
      _resolve = resolve
    })
  }
}

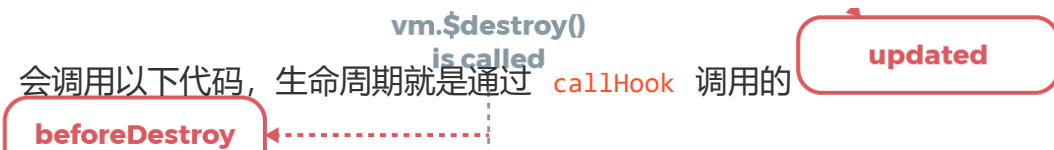
```

## Vue 生命周期

生命周期函数就是组件在初始化或者数据更新时会触发的钩子函数。



在初始化时，会调用以下代码，生命周期就是通过 `callHook` 调用的



scss 复制代码

```
Vue.prototype._init = function(options) {  
  initLifecycle(vm)  
  initEvents(vm)  
  initRender(vm)  
  callHook(vm, 'beforeCreate') // 拿不到 props data  
  initInjections(vm)  
  initState(vm)  
  initProvide(vm)  
  callHook(vm, 'created')  
}
```

可以发现在以上代码中，`beforeCreate` 调用的时候，是获取不到 `props` 或者 `data` 中的数据，因为这些数据的初始化都在 `initState` 中。

\* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

接下来会执行挂载函数

@稀土掘金技术社区

javascript 复制代码

```
export function mountComponent {  
  callHook(vm, 'beforeMount')  
  // ...  
  if (vm.$vnode == null) {  
    vm._isMounted = true  
    callHook(vm, 'mounted')  
  }  
}
```

`beforeMount` 就是在挂载前执行的，然后开始创建 VDOM 并替换成真实 DOM，最后执行 `mounted` 钩子。这里会有个判断逻辑，如果是外部 `new Vue({})` 的话，不会存在 `$vnode`，所以直接执行 `mounted` 钩子了。如果有子组件的话，会递归挂载子组件，只有当所有子组件全部挂载完毕，才会执行根组件的挂载钩子。

接下来是数据更新时会调用的钩子函数

scss 复制代码

```
function flushSchedulerQueue() {  
  // ...  
  for (index = 0; index < queue.length; index++) {  
    watcher = queue[index]  
    if (watcher.before) {  
      watcher.before() // 调用 beforeUpdate  
    }  
  }  
}
```



```

    id = watcher.id
    has[id] = null
    watcher.run()
    // in dev build, check and stop circular updates.
    if (process.env.NODE_ENV !== 'production' && has[id] !== null) {
      circular[id] = (circular[id] || 0) + 1
      if (circular[id] > MAX_UPDATE_COUNT) {
        warn(
          'You may have an infinite update loop ' +
            (watcher.user
              ? `in watcher with expression "${watcher.expression}"`
              : `in a component render function.`),
          watcher.vm
        )
        break
      }
    }
  }
}
callUpdatedHooks(updatedQueue)
}

function callUpdatedHooks(queue) {
  let i = queue.length
  while (i--) {
    const watcher = queue[i]
    const vm = watcher.vm
    if (vm._watcher === watcher && vm._isMounted) {
      callHook(vm, 'updated')
    }
  }
}

```

上图还有两个生命周期没有说，分别为 **activated** 和 **deactivated**，这两个钩子函数是 **keep-alive** 组件独有的。用 **keep-alive** 包裹的组件在切换时不会进行销毁，而是缓存到内存中并执行 **deactivated** 钩子函数，命中缓存渲染后会执行 **activated** 钩子函数。

最后就是销毁组件的钩子函数了

php 复制代码

```

Vue.prototype.$destroy = function() {
  // ...
  callHook(vm, 'beforeDestroy')
  vm._isBeingDestroyed = true
  // remove self from parent
  const parent = vm.$parent
  if (parent && !parent._isBeingDestroyed && !vm.$options.abstract) {
    remove(parent.$children, vm)
  }
}

```

```

}
// teardown watchers
if (vm._watcher) {
  vm._watcher.teardown()
}
let i = vm._watchers.length
while (i--) {
  vm._watchers[i].teardown()
}
// remove reference from data ob
// frozen object may not have observer.
if (vm._data.__ob__) {
  vm._data.__ob__.vmCount--
}
// call the last hook...
vm._isDestroyed = true
// invoke destroy hooks on current rendered tree
vm.__patch__(vm._vnode, null)
// fire destroyed hook
callHook(vm, 'destroyed')
// turn off all instance listeners.
vm.$off()
// remove __vue__ reference
if (vm.$el) {
  vm.$el.__vue__ = null
}
// release circular reference (#6759)
if (vm.$vnode) {
  vm.$vnode.parent = null
}
}
}

```

在执行销毁操作前会调用 `beforeDestroy` 钩子函数，然后进行一系列的销毁操作，如果有子组件的话，也会递归销毁子组件，所有子组件都销毁完毕后会执行根组件的 `destroyed` 钩子函数。

## Vue 双向绑定

- 在初始化 data props 时，递归对象，给每一个属性双向绑定，对于数组而言，会拿到原型重写函数，实现手动派发更新。因为函数不能监听到数据的变动，和 proxy 比较一下。
- 除了以上数组函数，通过索引改变数组数据或者给对象添加新属性也不能触发，需要使用自带的 set 函数，这个函数内部也是手动派发更新
- 在组件挂载时，会实例化渲染观察者，传入组件更新的回调。在实例化过程中，会对模板中的值对象进行求值，触发依赖收集。在触发依赖之前，会保存当前的渲染观察者，用于组件

含有子组件的时候，恢复父组件的观察者。触发依赖收集后，会清理掉不需要的依赖，性能优化，防止不需要的地方去重复渲染。

- 改变值会触发依赖更新，会将收集到的所有依赖全部拿出来，放入 nextTick 中统一执行。执行过程中，会先对观察者进行排序，渲染的最后执行。先执行 beforeupdate 钩子函数，然后执行观察者的回调。在执行回调的过程中，可能 watch 会再次 push 进来，因为存在在回调中再次赋值，判断无限循环。

## v-model原理

- v-model 在模板编译的时候转换代码
- v-model 本质是 :value 和 v-on，但是略微有点区别。在输入控件下，有两个事件监听，输入中文时只有当输出中文才触发数据赋值
- v-model 和:bind 同时使用，前者优先级更高，如果 :value 会出现冲突
- v-model 因为语法糖的原因，还可以用于父子通信

## watch 和 computed 的区别和运用的场景

- 前者是计算属性，依赖其他属性计算值。并且 computer 的值有缓存，只有当计算值变化才变化触发渲染。后者监听到值得变化就会执行回调
- computer 就是简单计算一下，适用于渲染页面。watch 适合做一些复杂业务逻辑
- 前者有依赖两个 watcher，computer watcher 和渲染 watcher。判断计算出的值变化后渲染 watcher 派发更新触发渲染

## Vue 的父子通信

- 使用 `v-model` 实现父传子，子传父。因为 v-model 默认解析成 :value 和 :input
- 父传子
  - 通过 `props`
  - 通过 `$children` 访问子组件数组，注意该数组乱序
  - 对于多级父传子，可以使用 `v-bind={$attrs}`，通过对象的方式筛选出父组件中传入但子组件不需要的 props
  - `$listeners` 包含了父作用域中的 (不含 `.native` 修饰器的) `v-on` 事件监听器。
- 子传父
  - 父组件传递函数给子组件，子组件通过 `$emit` 触发
  - 修改父组件的 `props`
  - 通过 `$parent` 访问父组件
  - `.sync`
- 平行组件

- EventBus
- Vuex 解决一切

## 路由原理

前端路由实现起来其实很简单，本质就是监听 URL 的变化，然后匹配路由规则，显示相应的页面，并且无须刷新。目前单页面使用的路由就只有两种实现方式

- hash 模式
- history 模式

`www.test.com/#/` 就是 Hash URL，当 # 后面的哈希值发生变化时，不会向服务器请求数据，可以通过 `hashchange` 事件来监听到 URL 的变化，从而进行跳转页面。



History 模式是 HTML5 新推出的功能，比之 Hash URL 更加美观



## MVVM

MVVM 由以下三个内容组成

- View: 界面
- Model: 数据模型
- ViewModel: 作为桥梁负责沟通 View 和 Model

在 JQuery 时期，如果需要刷新 UI 时，需要先取到对应的 DOM 再更新 UI，这样数据和业务的逻辑就和页面有强耦合。

在 MVVM 中，UI 是通过数据驱动的，数据一旦改变就会相应的刷新对应的 UI，UI 如果改变，也会改变对应的数据。这种方式就可以在业务处理中只关心数据的流转，而无需直接和页面打交道。ViewModel 只关心数据和业务的处理，不关心 View 如何处理数据，在这种情况下，View 和 Model 都可以独立出来，任何一方改变了也不一定需要改变另一方，并且可以将一些可复用的逻辑放在一个 ViewModel 中，让多个 View 复用这个 ViewModel。

在 MVVM 中，最核心的也就是数据双向绑定，例如 Angular 的脏数据检测，Vue 中的数据劫持。

## 脏数据检测

当触发了指定事件后会进入脏数据检测，这时会调用 `$digest` 循环遍历所有的数据观察者，判断当前值是否和先前的值有区别，如果检测到变化的话，会调用 `$watch` 函数，然后再次调用 `$digest` 循环直到发现没有变化。循环至少为二次，至多为十次。

脏数据检测虽然存在低效的问题，但是不关心数据是通过什么方式改变的，都可以完成任务，但是这在 Vue 中的双向绑定是存在问题的。并且脏数据检测可以实现批量检测出更新的值，再去统一更新 UI，大大减少了操作 DOM 的次数。所以低效也是相对的，这就仁者见仁智者见智了。

## 数据劫持

Vue 内部使用了 `Object.defineProperty()` 来实现双向绑定，通过这个函数可以监听到 `set` 和 `get` 的事件。

javascript 复制代码

```
var data = { name: 'yck' }  
  
observe(data)  
  
let name = data.name // -> get value  
data.name = 'yyy' // -> change value  
  
function observe(obj) {  
  // 判断类型  
  if (!obj || typeof obj !== 'object') {  
    return  
  }  
  Object.keys(obj).forEach(key => {  
    defineReactive(obj, key, obj[key])  
  })  
}
```

```

    })
  }

function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter() {
      console.log('get value')
      return val
    },
    set: function reactiveSetter(newVal) {
      console.log('change value')
      val = newVal
    }
  })
}

```

以上代码简单的实现了如何监听数据的 `set` 和 `get` 的事件，但是仅仅如此是不够的，还需要在适当的时候给属性添加发布订阅

```

<div>
  {{name}}
</div>

```

css 复制代码

在解析如上模板代码时，遇到 `{{name}}` 就会给属性 `name` 添加发布订阅。

```

// 通过 Dep 解耦
class Dep {
  constructor() {
    this.subs = []
  }
  addSub(sub) {
    // sub 是 Watcher 实例
    this.subs.push(sub)
  }
  notify() {
    this.subs.forEach(sub => {
      sub.update()
    })
  }
}

// 全局属性，通过该属性配置 Watcher

```

kotlin 复制代码

```

Dep.target = null

function update(value) {
  document.querySelector('div').innerText = value
}

class Watcher {
  constructor(obj, key, cb) {
    // 将 Dep.target 指向自己
    // 然后触发属性的 getter 添加监听
    // 最后将 Dep.target 置空
    Dep.target = this
    this.cb = cb
    this.obj = obj
    this.key = key
    this.value = obj[key]
    Dep.target = null
  }
  update() {
    // 获得新值
    this.value = this.obj[this.key]
    // 调用 update 方法更新 Dom
    this.cb(this.value)
  }
}

var data = { name: 'yck' }
observe(data)
// 模拟解析到 `{name}` 触发的操作
new Watcher(data, 'name', update)
// update Dom innerText
data.name = 'yyy'

```

接下来,对 `defineReactive` 函数进行改造

javascript 复制代码

```

function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  let dp = new Dep()
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter() {
      console.log('get value')
      // 将 Watcher 添加到订阅
      if (Dep.target) {
        dp.addSub(Dep.target)
      }
    }
  })
}

```

```

    return val
  },
  set: function reactiveSetter(newVal) {
    console.log('change value')
    val = newVal
    // 执行 watcher 的 update 方法
    dp.notify()
  }
})
}

```

以上实现了一个简易的双向绑定，核心思路就是手动触发一次属性的 getter 来实现发布订阅的添加。

## Proxy 与 Object.defineProperty 对比

**Object.defineProperty** 虽然已经能够实现双向绑定了，但是他还是有缺陷的。

1. 只能对属性进行数据劫持，所以需要深度遍历整个对象
2. 对于数组不能监听到数据的变化

虽然 Vue 中确实能检测到数组数据的变化，但是其实是使用了 hack 的办法，并且也是有缺陷的。

javascript 复制代码

```

const arrayProto = Array.prototype
export const arrayMethods = Object.create(arrayProto)
// hack 以下几个函数
const methodsToPatch = [
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]
methodsToPatch.forEach(function (method) {
  // 获得原生函数
  const original = arrayProto[method]
  def(arrayMethods, method, function mutator (...args) {
    // 调用原生函数
    const result = original.apply(this, args)
    const ob = this.__ob__
    let inserted

```



```

switch (method) {
  case 'push':
  case 'unshift':
    inserted = args
    break
  case 'splice':
    inserted = args.slice(2)
    break
}
if (inserted) ob.observeArray(inserted)
// 触发更新
ob.dep.notify()
return result
})
})

```

反观 Proxy 就没以上的问题，原生支持监听数组变化，并且可以直接对整个对象进行拦截，所以 Vue 也将在下一个大版本中使用 Proxy 替换 Object.defineProperty

javascript 复制代码

```

let onWatch = (obj, setBind, getLogger) => {
  let handler = {
    get(target, property, receiver) {
      getLogger(target, property)
      return Reflect.get(target, property, receiver);
    },
    set(target, property, value, receiver) {
      setBind(value);
      return Reflect.set(target, property, value);
    }
  };
  return new Proxy(obj, handler);
};

let obj = { a: 1 }
let value
let p = onWatch(obj, (v) => {
  value = v
}, (target, property) => {
  console.log(`Get '${property}' = ${target[property]}`);
})
p.a = 2 // bind `value` to `2`
p.a // -> Get 'a' = 2

```

## 虚拟 DOM

虚拟 DOM 涉及的内容很多，具体可以参考我之前 [写的文章](#)

## 路由鉴权

- 登录页和其他页面分开，登录以后实例化 Vue 并且初始化需要的路由
- 动态路由，通过 addRoute 实现

## Vue 和 React 区别

- Vue 表单支持双向绑定开发更方便
- 改变数据方式不同，setState 有使用坑
- props Vue 可变，React 不可变
- 判断是否需要更新 React 可以通过钩子函数判断，Vue 使用依赖追踪，修改了什么才渲染什么
- React 16以后 有些钩子函数会执行多次
- React 需要使用 JSX，需要 Babel 编译。Vue 虽然可以使用模板，但是也可以通过直接编写 render 函数不需要编译就能运行。
- 生态 React 相对较好

## 网络

---

### TCP 3次握手



在 TCP 协议中，主动发起请求的一端为客户端，被动连接的一端称为服务端。不管是客户端还是服务端，TCP 连接建立完后都能发送和接收数据，所以 TCP 也是一个全双工的协议。

起初，两端都为 CLOSED 状态。在通信开始前，双方都会创建 TCB。服务器创建完 TCB 后进入 LISTEN 状态，此时开始等待客户端发送数据。

#### 第一次握手

客户端向服务端发送连接请求报文段。该报文段中包含自身的数据通讯初始序号。请求发送后，客户端便进入 SYN-SENT 状态，x 表示客户端的数据通信初始序号。

#### 第二次握手

服务端收到连接请求报文段后，如果同意连接，则会发送一个应答，该应答中也会包含自身的数据通讯初始序号，发送完成后便进入 SYN-RECEIVED 状态。

### 第三次握手

当客户端收到连接同意的应答后，还要向服务端发送一个确认报文。客户端发完这个报文段后便进入 ESTABLISHED 状态，服务端收到这个应答后也进入 ESTABLISHED 状态，此时连接建立成功。

PS：第三次握手可以包含数据，通过 TCP 快速打开（TFO）技术。其实只要涉及到握手的协议，都可以使用类似 TFO 的方式，客户端和服务端存储相同 cookie，下次握手时发出 cookie 达到减少 RTT 的目的。

### 你是否有疑惑明明两次握手就可以建立起连接，为什么还需要第三次应答？

因为这是为了防止失效的连接请求报文段被服务端接收，从而产生错误。

可以想象如下场景。客户端发送了一个连接请求 A，但是因为网络原因造成了超时，这时 TCP 会启动超时重传的机制再次发送一个连接请求 B。此时请求顺利到达服务端，服务端应答完就建立了请求。如果连接请求 A 在两端关闭后终于抵达了服务端，那么这时服务端会认为客户端又需要建立 TCP 连接，从而应答了该请求并进入 ESTABLISHED 状态。此时客户端其实是 CLOSED 状态，那么就会导致服务端一直等待，造成资源的浪费。

PS：在建立连接中，任意一端掉线，TCP 都会重发 SYN 包，一般会重试五次，在建立连接中可能会遇到 SYN FLOOD 攻击。遇到这种情况你可以选择调低重试次数或者干脆在不能处理的情况下拒绝请求。

### TCP 拥塞控制

拥塞处理和流量控制不同，后者是作用于接收方，保证接收方来得及接受数据。而前者是作用于网络，防止过多的数据拥塞网络，避免出现网络负载过大的情况。

拥塞处理包括了四个算法，分别为：慢开始，拥塞避免，快速重传，快速恢复。

### 慢开始算法

慢开始算法，顾名思义，就是在传输开始时将发送窗口慢慢指数级扩大，从而避免一开始就传输大量数据导致网络拥塞。

慢开始算法步骤具体如下

1. 连接初始设置拥塞窗口 (Congestion Window) 为 1 MSS (一个分段的最大数据量)
2. 每过一个 RTT 就将窗口大小乘二
3. 指数级增长肯定不能没有限制的, 所以有一个阈值限制, 当窗口大小大于阈值时就会启动拥塞避免算法。

## 拥塞避免算法

拥塞避免算法相比简单点, 每过一个 RTT 窗口大小只加一, 这样能够避免指数级增长导致网络拥塞, 慢慢将大小调整到最佳值。

在传输过程中可能定时器超时的情况, 这时候 TCP 会认为网络拥塞了, 会马上进行以下步骤:

- 将阈值设为当前拥塞窗口的一半
- 将拥塞窗口设为 1 MSS
- 启动拥塞避免算法

## 快速重传

快速重传一般和快恢复一起出现。一旦接收端收到的报文出现失序的情况, 接收端只会回复最后一个顺序正确的报文序号 (没有 Sack 的情况下)。如果收到三个重复的 ACK, 无需等待定时器超时再重发而是启动快速重传。具体算法分为两种:

### TCP Tahoe 实现如下

- 将阈值设为当前拥塞窗口的一半
- 将拥塞窗口设为 1 MSS
- 重新开始慢开始算法

### TCP Reno 实现如下

- 拥塞窗口减半
- 将阈值设为当前拥塞窗口
- 进入快恢复阶段 (重发对端需要的包, 一旦收到一个新的 ACK 答复就退出该阶段)
- 使用拥塞避免算法

### TCP New Ren 改进后的快恢复

**TCP New Reno** 算法改进了之前 **TCP Reno** 算法的缺陷。在之前，快恢复中只要收到一个新的 ACK 包，就会退出快恢复。

在 **TCP New Reno** 中，TCP 发送方先记下三个重复 ACK 的分段的最大序号。

假如我有一个分段数据是 1 ~ 10 这十个序号的报文，其中丢失了序号为 3 和 7 的报文，那么该分段的最大序号就是 10。发送端只会收到 ACK 序号为 3 的应答。这时候重发序号为 3 的报文，接收方顺利接收并会发送 ACK 序号为 7 的应答。这时候 TCP 知道对端是有多个包未收到，会继续发送序号为 7 的报文，接收方顺利接收并会发送 ACK 序号为 11 的应答，这时发送端认为这个分段接收端已经顺利接收，接下来会退出快恢复阶段。

## HTTPS 握手

HTTPS 还是通过了 HTTP 来传输信息，但是信息通过 TLS 协议进行了加密。

## TLS

---

TLS 协议位于传输层之上，应用层之下。首次进行 TLS 协议传输需要两个 RTT，接下来可以通过 Session Resumption 减少到一个 RTT。

在 TLS 中使用了两种加密技术，分别为：对称加密和非对称加密。

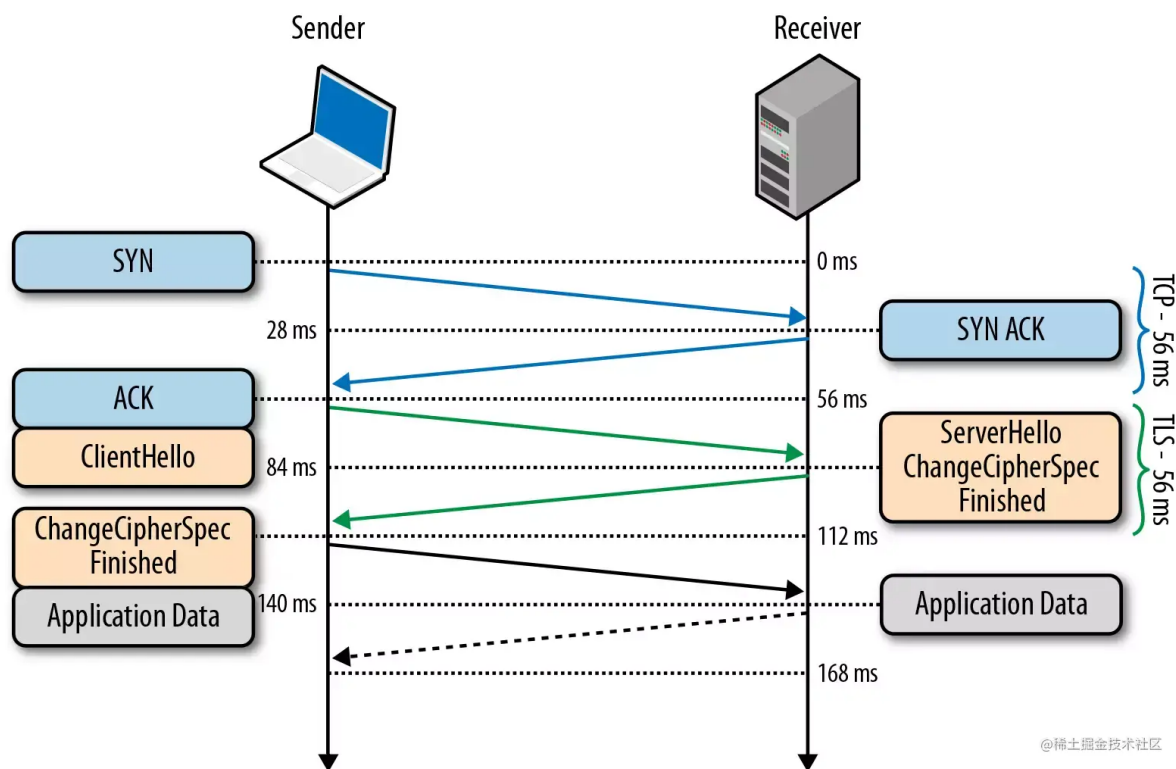
### 对称加密：

对称加密就是两边拥有相同的密钥，两边都知道如何将密文加密解密。

### 非对称加密：

有公钥私钥之分，公钥所有人都是可以知道，可以将数据用公钥加密，但是将数据解密必须使用私钥解密，私钥只有分发公钥的一方才知道。

### TLS 握手过程如下图：



1. 客户端发送一个随机值，需要的协议和加密方式
2. 服务端收到客户端的随机值，自己也产生一个随机值，并根据客户端需求的协议和加密方式来使用对应的方式，发送自己的证书（如果需要验证客户端证书需要说明）
3. 客户端收到服务端的证书并验证是否有效，验证通过会再生成一个随机值，通过服务端证书的公钥去加密这个随机值并发送给服务端，如果服务端需要验证客户端证书的话会附带证书
4. 服务端收到加密过的随机值并使用私钥解密获得第三个随机值，这时候两端都拥有了三个随机值，可以通过这三个随机值按照之前约定的加密方式生成密钥，接下来的通信就可以通过该密钥来加密解密

通过以上步骤可知，在 TLS 握手阶段，两端使用非对称加密的方式来通信，但是因为非对称加密损耗的性能比对称加密大，所以在正式传输数据时，两端使用对称加密的方式通信。

PS：以上说明的都是 TLS 1.2 协议的握手情况，在 1.3 协议中，首次建立连接只需要一个 RTT，后面恢复连接不需要 RTT 了。

## 从输入 URL 到页面加载全过程

1. 首先做 DNS 查询，如果这一步做了智能 DNS 解析的话，会提供访问速度最快的 IP 地址回来
2. 接下来是 TCP 握手，应用层会下发数据给传输层，这里 TCP 协议会指明两端的端口号，然后下发给网络层。网络层中的 IP 协议会确定 IP 地址，并且指示了数据传输中如何跳转路由器。然后包会再被封装到数据链路层的数据帧结构中，最后就是物理层面的传输了

3. TCP 握手结束后会进行 TLS 握手，然后就开始正式的传输数据
4. 数据在进入服务端之前，可能还会先经过负责负载均衡的服务器，它的作用就是将请求合理的分发到多台服务器上，这时假设服务端会响应一个 HTML 文件
5. 首先浏览器会判断状态码是什么，如果是 200 那就继续解析，如果 400 或 500 的话就会报错，如果 300 的话会进行重定向，这里会有个重定向计数器，避免过多次的重定向，超过次数也会报错
6. 浏览器开始解析文件，如果是 gzip 格式的话会先解压一下，然后通过文件的编码格式知道该如何去解码文件
7. 文件解码成功后会正式开始渲染流程，先会根据 HTML 构建 DOM 树，有 CSS 的话会去构建 CSSOM 树。如果遇到 `script` 标签的话，会判断是否存在 `async` 或者 `defer`，前者会并行进行下载并执行 JS，后者会先下载文件，然后等待 HTML 解析完成后顺序执行，如果以上都没有，就会阻塞住渲染流程直到 JS 执行完毕。遇到文件下载的会去下载文件，这里如果使用 HTTP 2.0 协议的话会极大的提高多图的下载效率。
8. 初始的 HTML 被完全加载和解析后会触发 `DOMContentLoaded` 事件
9. CSSOM 树和 DOM 树构建完成后会开始生成 Render 树，这一步就是确定页面元素的布局、样式等等诸多方面的东西
10. 在生成 Render 树的过程中，浏览器就开始调用 GPU 绘制，合成图层，将内容显示在屏幕上了

## HTTP 常用返回码

### 2XX 成功

- 200 OK，表示从客户端发来的请求在服务器端被正确处理
- 204 No content，表示请求成功，但响应报文不含实体的主体部分
- 205 Reset Content，表示请求成功，但响应报文不含实体的主体部分，但是与 204 响应不同在于要求请求方重置内容
- 206 Partial Content，进行范围请求

### 3XX 重定向

- 301 moved permanently，永久性重定向，表示资源已被分配了新的 URL
- 302 found，临时性重定向，表示资源临时被分配了新的 URL
- 303 see other，表示资源存在着另一个 URL，应使用 GET 方法获取资源
- 304 not modified，表示服务器允许访问资源，但因发生请求未满足条件的情况
- 307 temporary redirect，临时重定向，和 302 含义类似，但是期望客户端保持请求方法不变向新的地址发出请求

## 4XX 客户端错误

- 400 bad request, 请求报文存在语法错误
- 401 unauthorized, 表示发送的请求需要有通过 HTTP 认证的认证信息
- 403 forbidden, 表示对请求资源的访问被服务器拒绝
- 404 not found, 表示在服务器上没有找到请求的资源

## 5XX 服务器错误

- 500 internal sever error, 表示服务器端在执行请求时发生了错误
- 501 Not Implemented, 表示服务器不支持当前请求所需要的某个功能
- 503 service unavailable, 表明服务器暂时处于超负载或正在停机维护, 无法处理请求

## 数据结构算法

---

### 常见排序

以下两个函数是排序中会用到的通用函数, 就不一一写了

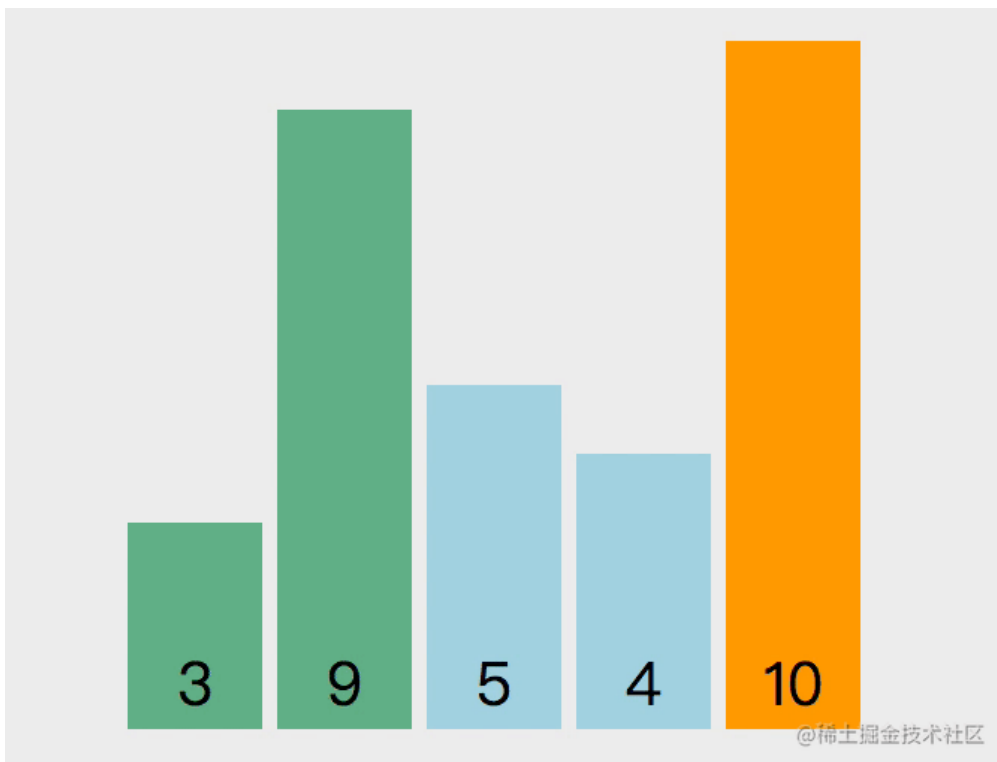
sql 复制代码

```
function checkArray(array) {  
  if (!array || array.length <= 2) return  
}  
function swap(array, left, right) {  
  let rightValue = array[right]  
  array[right] = array[left]  
  array[left] = rightValue  
}
```

### 冒泡排序

冒泡排序的原理如下, 从第一个元素开始, 把当前元素和下一个索引元素进行比较。如果当前元素大, 那么就交换位置, 重复操作直到比较到最后一个元素, 那么此时最后一个元素就是该数组中最大的数。下一轮重复以上操作, 但是此时最后一个元素已经是最大数了, 所以不需要再比较最后一个元素, 只需要比较到 `length - 1` 的位置。





以下是实现该算法的代码

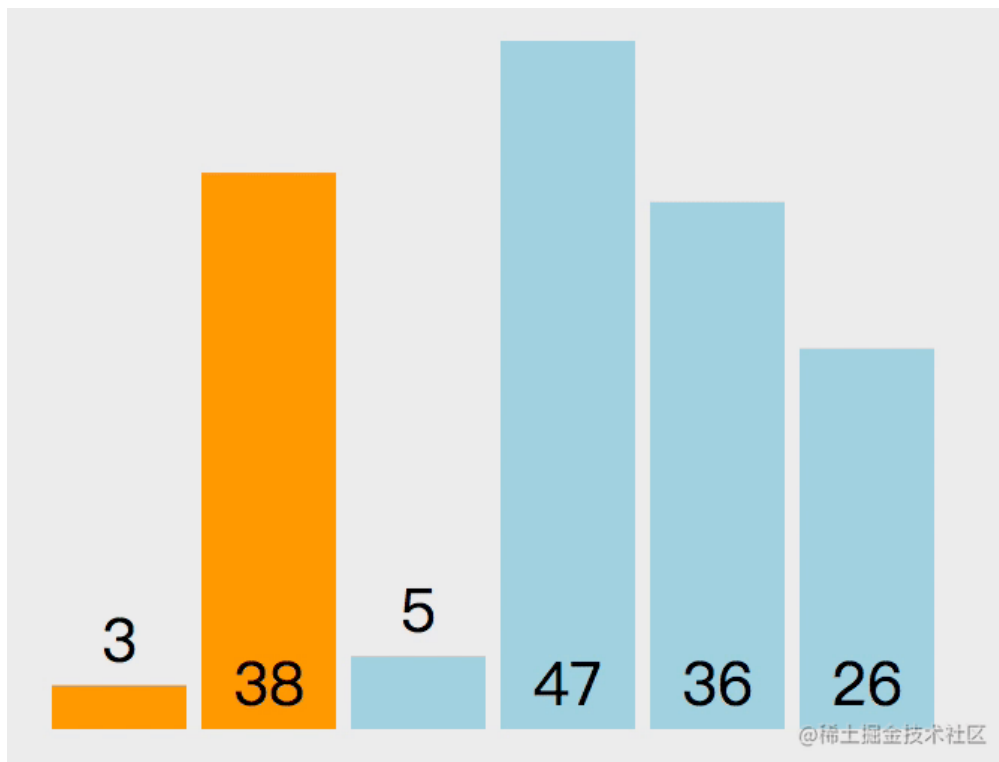
php 复制代码

```
function bubble(array) {  
  checkArray(array);  
  for (let i = array.length - 1; i > 0; i--) {  
    // 从 0 到 `length - 1` 遍历  
    for (let j = 0; j < i; j++) {  
      if (array[j] > array[j + 1]) swap(array, j, j + 1)  
    }  
  }  
  return array;  
}
```

该算法的操作次数是一个等差数列  $n + (n - 1) + (n - 2) + 1$ ，去掉常数项以后得出时间复杂度是  $O(n * n)$

## 插入排序

插入排序的原理如下。第一个元素默认是已排序元素，取出下一个元素和当前元素比较，如果当前元素大就交换位置。那么此时第一个元素就是当前的最小数，所以下次取出操作从第三个元素开始，向前对比，重复之前的操作。



以下是实现该算法的代码

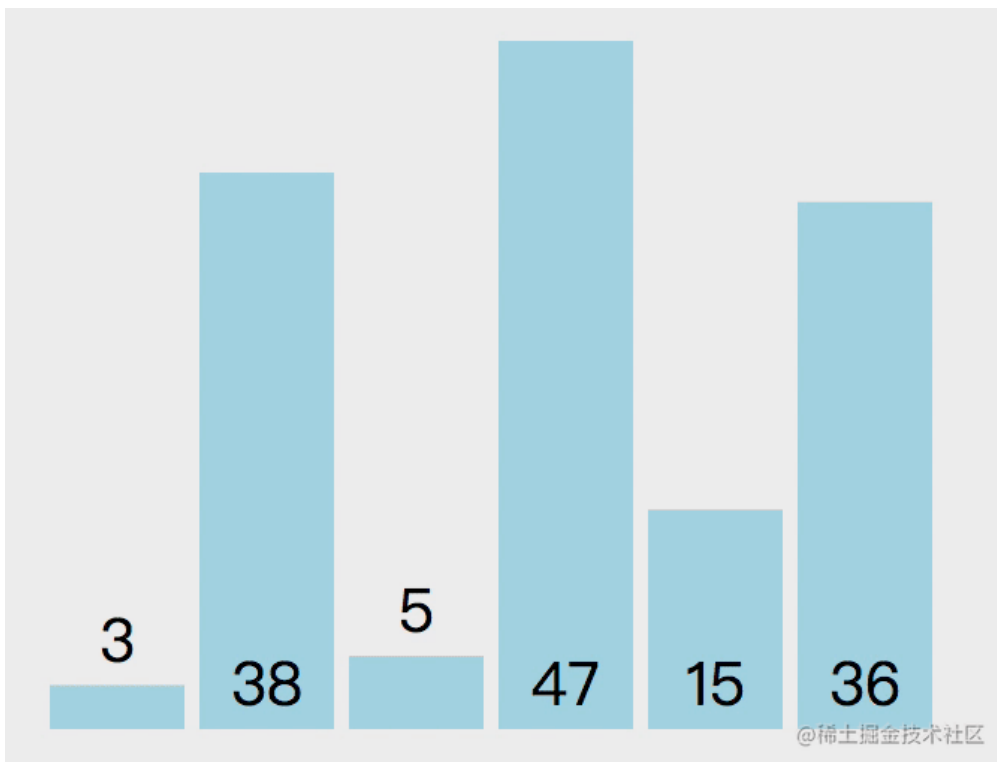
php 复制代码

```
function insertion(array) {  
    checkArray(array);  
    for (let i = 1; i < array.length; i++) {  
        for (let j = i - 1; j >= 0 && array[j] > array[j + 1]; j--)  
            swap(array, j, j + 1);  
    }  
    return array;  
}
```

该算法的操作次数是一个等差数列  $n + (n - 1) + (n - 2) + 1$ ，去掉常数项以后得出时间复杂度是  $O(n * n)$

## 选择排序

选择排序的原理如下。遍历数组，设置最小值的索引为 0，如果取出的值比当前最小值小，就替换最小值索引，遍历完成后，将第一个元素和最小值索引上的值交换。如上操作后，第一个元素就是数组中的最小值，下次遍历就可以从索引 1 开始重复上述操作。



以下是实现该算法的代码

ini 复制代码

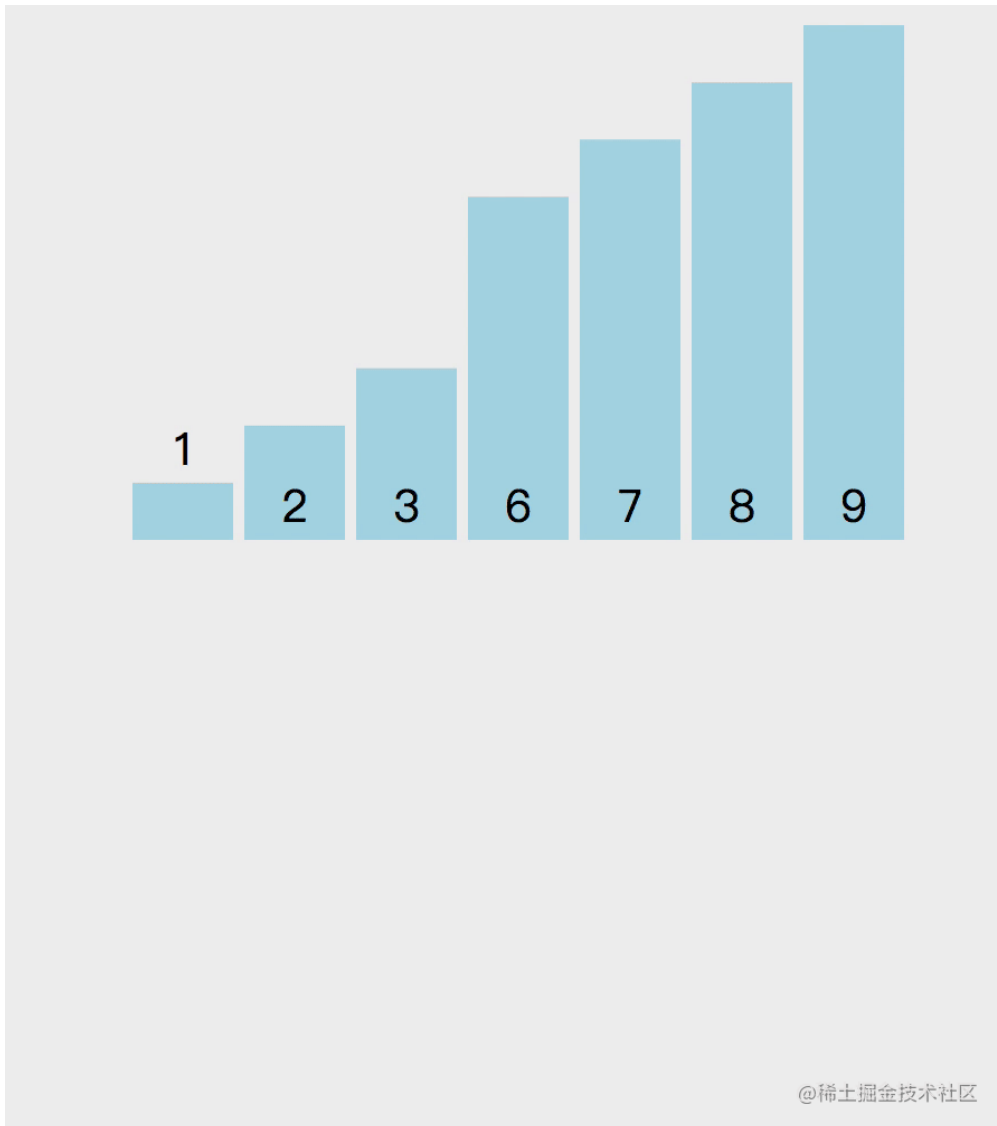
```
function selection(array) {  
  checkArray(array);  
  for (let i = 0; i < array.Length - 1; i++) {  
    let minIndex = i;  
    for (let j = i + 1; j < array.Length; j++) {  
      minIndex = array[j] < array[minIndex] ? j : minIndex;  
    }  
    swap(array, i, minIndex);  
  }  
  return array;  
}
```

该算法的操作次数是一个等差数列  $n + (n - 1) + (n - 2) + 1$ ，去掉常数项以后得出时间复杂度是  $O(n * n)$

## 归并排序

归并排序的原理如下。递归的将数组两两分开直到最多包含两个元素，然后将数组排序合并，最终合并为排序好的数组。假设我有一组数组  $[3, 1, 2, 8, 9, 7, 6]$ ，中间数索引是 3，先排序数组  $[3, 1, 2, 8]$ 。在这个左边数组上，继续拆分直到变成数组包含两个元素（如果数组长度是奇数的话，会有一个拆分数组只包含一个元素）。然后排序数组  $[3, 1]$  和  $[2, 8]$

，然后再排序数组 `[1, 3, 2, 8]`，这样左边数组就排序完成，然后按照以上思路排序右边数组，最后将数组 `[1, 2, 3, 8]` 和 `[6, 7, 9]` 排序。



以下是实现该算法的代码

ini 复制代码

```
function sort(array) {
  checkArray(array);
  mergeSort(array, 0, array.length - 1);
  return array;
}

function mergeSort(array, left, right) {
  // 左右索引相同说明已经只有一个数
  if (left === right) return;
  // 等同于 `left + (right - left) / 2`
  // 相比 `(left + right) / 2` 来说更加安全，不会溢出
  // 使用位运算是因为位运算比四则运算快
  let mid = parseInt(left + ((right - left) >> 1));
  mergeSort(array, left, mid);
```

```

mergeSort(array, mid + 1, right);

let help = [];
let i = 0;
let p1 = left;
let p2 = mid + 1;
while (p1 <= mid && p2 <= right) {
    help[i++] = array[p1] < array[p2] ? array[p1++] : array[p2++];
}
while (p1 <= mid) {
    help[i++] = array[p1++];
}
while (p2 <= right) {
    help[i++] = array[p2++];
}
for (let i = 0; i < help.length; i++) {
    array[left + i] = help[i];
}
return array;
}

```

以上算法使用了递归的思想。递归的本质就是压栈，每递归执行一次函数，就将该函数的信息（比如参数，内部的变量，执行到的行数）压栈，直到遇到终止条件，然后出栈并继续执行函数。对于以上递归函数的调用轨迹如下

scss 复制代码

```

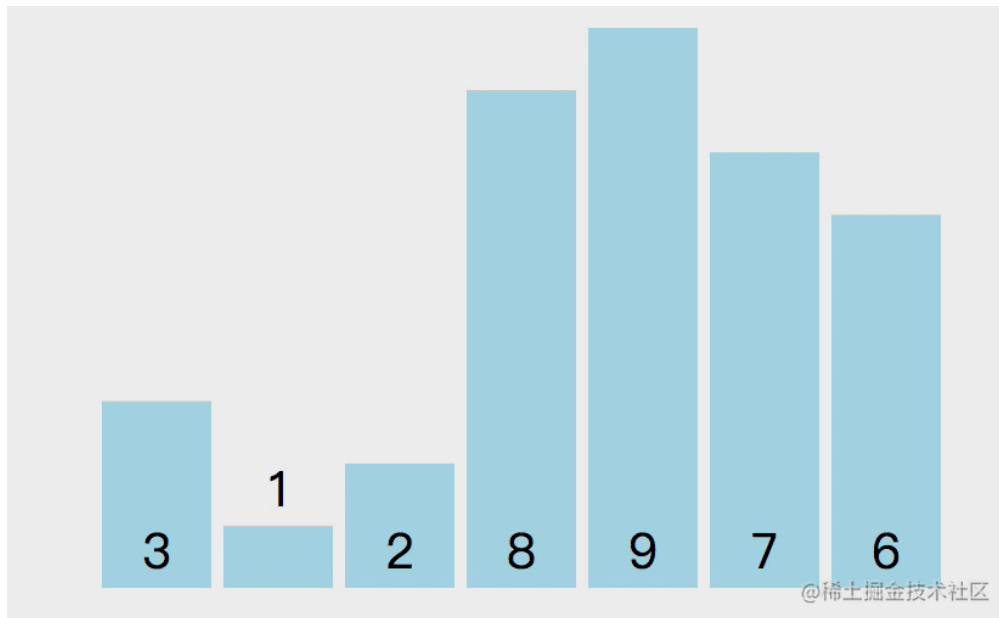
mergeSort(data, 0, 6) // mid = 3
  mergeSort(data, 0, 3) // mid = 1
    mergeSort(data, 0, 1) // mid = 0
      mergeSort(data, 0, 0) // 遇到终止，回退到上一步
      mergeSort(data, 1, 1) // 遇到终止，回退到上一步
      // 排序 p1 = 0, p2 = mid + 1 = 1
      // 回退到 `mergeSort(data, 0, 3)` 执行下一个递归
    mergeSort(2, 3) // mid = 2
      mergeSort(3, 3) // 遇到终止，回退到上一步
      // 排序 p1 = 2, p2 = mid + 1 = 3
      // 回退到 `mergeSort(data, 0, 3)` 执行合并逻辑
      // 排序 p1 = 0, p2 = mid + 1 = 2
      // 执行完毕回退
      // 左边数组排序完毕，右边也是如上轨迹

```

该算法的操作次数是可以这样计算：递归了两次，每次数据量是数组的一半，并且最后把整个数组迭代了一次，所以得出表达式  $2T(N/2) + T(N)$ （T 代表时间，N 代表数据量）。根据该表达式可以套用 [该公式](#) 得出时间复杂度为  $O(N * \log N)$

# 快排

快排的原理如下。随机选取一个数组中的值作为基准值，从左至右取值与基准值对比大小。比基准值小的放数组左边，大的放右边，对比完成后将基准值和第一个比基准值大的值交换位置。然后将数组以基准值的位置分为两部分，继续递归以上操作。



以下是实现该算法的代码

scss 复制代码

```
function sort(array) {
  checkArray(array);
  quickSort(array, 0, array.length - 1);
  return array;
}

function quickSort(array, left, right) {
  if (left < right) {
    swap(array, , right)
    // 随机取值，然后和末尾交换，这样做比固定取一个位置的复杂度略低
    let indexs = part(array, parseInt(Math.random() * (right - left + 1)) + left, right);
    quickSort(array, left, indexs[0]);
    quickSort(array, indexs[1] + 1, right);
  }
}

function part(array, left, right) {
  let less = left - 1;
  let more = right;
  while (left < more) {
    if (array[left] < array[right]) {
      // 当前值比基准值小，`less` 和 `left` 都加一
      ++less;
    }
  }
}
```

```

    ++left;
} else if (array[left] > array[right]) {
    // 当前值比基准值大，将当前值和右边的值交换
    // 并且不改变 `left`，因为当前换过来的值还没有判断过大小
    swap(array, --more, left);
} else {
    // 和基准值相同，只移动下标
    left++;
}
}
// 将基准值和比基准值大的第一个值交换位置
// 这样数组就变成 `[比基准值小, 基准值, 比基准值大]`
swap(array, right, more);
return [less, more];
}

```

该算法的复杂度和归并排序是相同的，但是额外空间复杂度比归并排序少，只需  $O(\log N)$ ，并且相比归并排序来说，所需的常数时间也更少。

## 面试题

**Sort Colors**：该题目来自 [LeetCode](#)，题目需要我们将 `[2,0,2,1,1,0]` 排序成 `[0,0,1,1,2,2]`，这个问题就可以使用三路快排的思想。

以下是代码实现

ini 复制代码

```

var sortColors = function(nums) {
    let left = -1;
    let right = nums.length;
    let i = 0;
    // 下标如果遇到 right，说明已经排序完成
    while (i < right) {
        if (nums[i] == 0) {
            swap(nums, i++, ++left);
        } else if (nums[i] == 1) {
            i++;
        } else {
            swap(nums, i, --right);
        }
    }
}
};

```

**Kth Largest Element in an Array**: 该题目来自 [LeetCode](#), 题目需要找出数组中第 K 大的元素, 这问题也可以使用快排的思路。并且因为是找出第 K 大元素, 所以在分离数组的过程中, 可以找出需要的元素在哪边, 然后只需要排序相应的一边数组就好。

以下是代码实现

ini 复制代码

```
var findKthLargest = function(nums, k) {  
    let l = 0  
    let r = nums.length - 1  
    // 得出第 K 大元素的索引位置  
    k = nums.length - k  
    while (l < r) {  
        // 分离数组后获得比基准数大的第一个元素索引  
        let index = part(nums, l, r)  
        // 判断该索引和 k 的大小  
        if (index < k) {  
            l = index + 1  
        } else if (index > k) {  
            r = index - 1  
        } else {  
            break  
        }  
    }  
    return nums[k]  
};  
  
function part(array, left, right) {  
    let less = left - 1;  
    let more = right;  
    while (left < more) {  
        if (array[left] < array[right]) {  
            ++less;  
            ++left;  
        } else if (array[left] > array[right]) {  
            swap(array, --more, left);  
        } else {  
            left++;  
        }  
    }  
    swap(array, right, more);  
    return more;  
}
```

## 堆排序

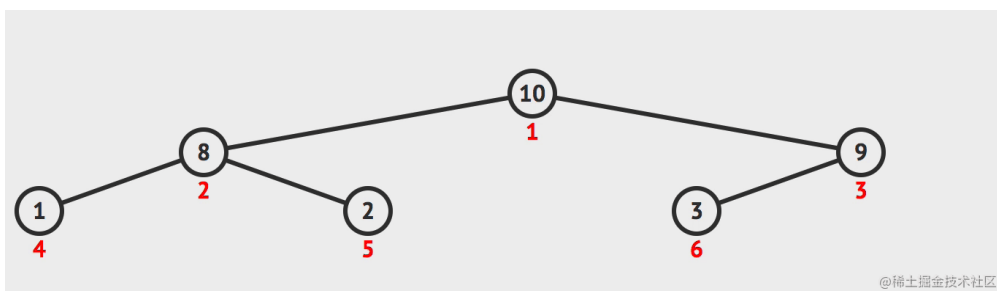


堆排序利用了二叉堆的特性来做，二叉堆通常用数组表示，并且二叉堆是一颗完全二叉树（所有叶节点（最底层的节点）都是从左往右顺序排序，并且其他层的节点都是满的）。二叉堆又分为大根堆与小根堆。

- 大根堆是某个节点的所有子节点的值都比他小
- 小根堆是某个节点的所有子节点的值都比他大

堆排序的原理就是组成一个大根堆或者小根堆。以小根堆为例，某个节点的左边子节点索引是  $i * 2 + 1$ ，右边是  $i * 2 + 2$ ，父节点是  $(i - 1) / 2$ 。

1. 首先遍历数组，判断该节点的父节点是否比他小，如果小就交换位置并继续判断，直到他的父节点比他大
2. 重新以上操作 1，直到数组首位是最大值
3. 然后将首位和末尾交换位置并将数组长度减一，表示数组末尾已是最大值，不需要再比较大小
4. 对比左右节点哪个大，然后记住大的节点的索引并且和父节点对比大小，如果子节点大就交换位置
5. 重复以上操作 3 - 4 直到整个数组都是大根堆。



@稀土掘金技术社区

以下是实现该算法的代码

scss 复制代码

```
function heap(array) {  
  checkArray(array);  
  // 将最大值交换到首位  
  for (let i = 0; i < array.length; i++) {  
    heapInsert(array, i);  
  }  
  let size = array.length;  
  // 交换首位和末尾  
  swap(array, 0, --size);  
  while (size > 0) {  
    heapify(array, 0, size);  
    swap(array, 0, --size);  
  }  
  return array;  
}
```

```

}

function heapInsert(array, index) {
  // 如果当前节点比父节点大，就交换
  while (array[index] > array[parseInt((index - 1) / 2)]) {
    swap(array, index, parseInt((index - 1) / 2));
    // 将索引变成父节点
    index = parseInt((index - 1) / 2);
  }
}

function heapify(array, index, size) {
  let left = index * 2 + 1;
  while (left < size) {
    // 判断左右节点大小
    let largest =
      left + 1 < size && array[left] < array[left + 1] ? left + 1 : left;
    // 判断子节点和父节点大小
    largest = array[index] < array[largest] ? largest : index;
    if (largest === index) break;
    swap(array, index, largest);
    index = largest;
    left = index * 2 + 1;
  }
}

```

以上代码实现了小根堆，如果需要实现大根堆，只需要把节点对比反一下就好。

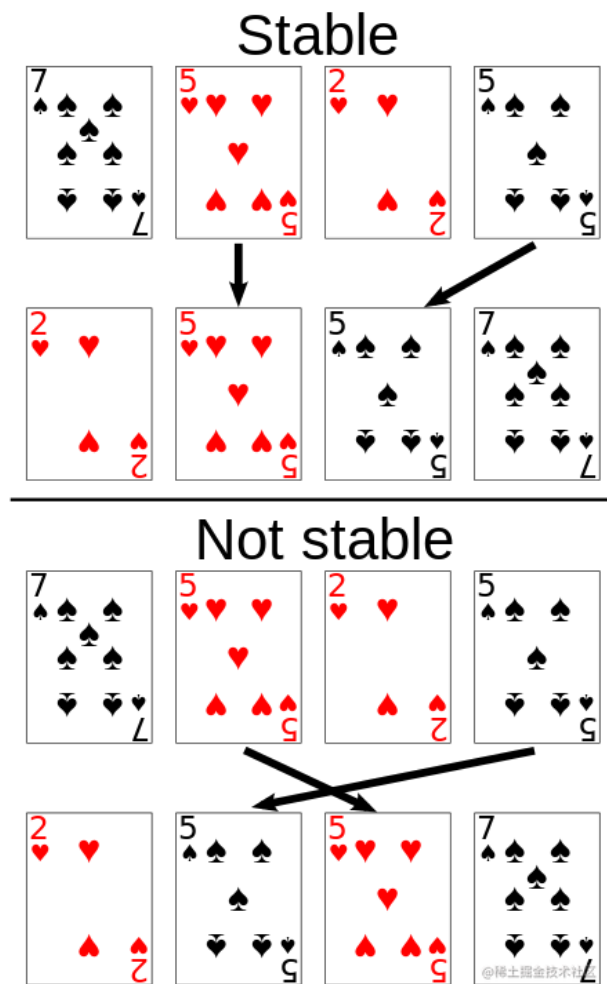
该算法的复杂度是  $O(\log N)$

## 系统自带排序实现

每个语言的排序内部实现都是不同的。

对于 JS 来说，数组长度大于 10 会采用快排，否则使用插入排序 [源码实现](#)。选择插入排序是因为虽然时间复杂度很差，但是在数据量很小的情况下和  $O(N * \log N)$  相差无几，然而插入排序需要的常数时间很小，所以相对别的排序来说更快。

对于 Java 来说，还会考虑内部的元素的类型。对于存储对象的数组来说，会采用稳定性好的算法。稳定性的意思就是对于相同值来说，相对顺序不能改变。



搜索二叉树

其他

---

介绍下设计模式

工厂模式

工厂模式分为好几种，这里就不一一讲解了，以下是一个简单工厂模式的例子

javascript 复制代码

```
class Man {  
  constructor(name) {  
    this.name = name  
  }  
  alertName() {  
    alert(this.name)  
  }  
}
```

```
class Factory {
  static create(name) {
    return new Man(name)
  }
}

Factory.create('yck').alertName()
```

当然工厂模式并不仅仅是用来 new 出**实例**。

可以想象一个场景。假设有一份很复杂的代码需要用户去调用，但是用户并不关心这些复杂的代码，只需要你提供给我一个接口去调用，用户只负责传递需要的参数，至于这些参数怎么使用，内部有什么逻辑是不关心的，只需要你最后返回我一个实例。这个构造过程就是工厂。

工厂起到的作用就是隐藏了创建实例的复杂度，只需要提供一个接口，简单清晰。

在 Vue 源码中，你也可以看到工厂模式的使用，比如创建异步组件

javascript 复制代码

```
export function createComponent (
  Ctor: Class<Component> | Function | Object | void,
  data: ?VNodeData,
  context: Component,
  children: ?Array<VNode>,
  tag?: string
): VNode | Array<VNode> | void {

  // 逻辑处理...

  const vnode = new VNode(
    `vue-component-${Ctor.cid}${name ? `-${name}` : ''}`,
    data, undefined, undefined, undefined, context,
    { Ctor, propsData, listeners, tag, children },
    asyncFactory
  )

  return vnode
}
```

在上述代码中，我们可以看到我们只需要调用 `createComponent` 传入参数就能创建一个组件实例，但是创建这个实例是很复杂的一个过程，工厂帮助我们隐藏了这个复杂的过程，只需要一句代码调用就能实现功能。

## 单例模式

单例模式很常用，比如全局缓存、全局状态管理等等这些只需要一个对象，就可以使用单例模式。

单例模式的核心就是保证全局只有一个对象可以访问。因为 JS 是门无类的语言，所以别的语言实现单例的方式并不能套入 JS 中，我们只需要用一个变量确保实例只创建一次就行，以下是如何实现单例模式的例子

javascript 复制代码

```
class Singleton {
  constructor() {}
}

Singleton.getInstance = (function() {
  let instance
  return function() {
    if (!instance) {
      instance = new Singleton()
    }
    return instance
  }
})()

let s1 = Singleton.getInstance()
let s2 = Singleton.getInstance()
console.log(s1 === s2) // true
```

在 Vuex 源码中，你也可以看到单例模式的使用，虽然它的实现方式不大一样，通过一个外部变量来控制只安装一次 Vuex

javascript 复制代码

```
let Vue // bind on install

export function install (_Vue) {
  if (Vue && _Vue === Vue) {
    // ...
    return
  }
  Vue = _Vue
  applyMixin(Vue)
}
```

## 适配器模式

适配器用来解决两个接口不兼容的情况，不需要改变已有的接口，通过包装一层的方式实现两个接口的正常协作。

以下是如何实现适配器模式的例子

javascript 复制代码

```
class Plug {
  getName() {
    return '港版插头'
  }
}

class Target {
  constructor() {
    this.plug = new Plug()
  }
  getName() {
    return this.plug.getName() + ' 适配器转二脚插头'
  }
}

let target = new Target()
target.getName() // 港版插头 适配器转二脚插头
```

在 Vue 中，我们其实经常使用到适配器模式。比如父组件传递给子组件一个时间戳属性，组件内部需要将时间戳转为正常的日期显示，一般会使用 `computed` 来做转换这件事情，这个过程就使用到了适配器模式。

## 装饰模式

装饰模式不需要改变已有的接口，作用是给对象添加功能。就像我们经常需要给手机戴个保护套防摔一样，不改变手机自身，给手机添加了保护套提供防摔功能。

以下是如何实现装饰模式的例子，使用了 ES7 中的装饰器语法

ini 复制代码

```
function readonly(target, key, descriptor) {
  descriptor.writable = false
  return descriptor
}

class Test {
  @readonly
  name = 'yck'
}
```

```
let t = new Test()

t.yck = '111' // 不可修改
```

在 React 中，装饰模式其实随处可见

```
import { connect } from 'react-redux'
class MyComponent extends React.Component {
  // ...
}
export default connect(mapStateToProps)(MyComponent)
```

scala 复制代码

## 代理模式

代理是为了控制对对象的访问，不让外部直接访问到对象。在现实生活中，也有很多代理的场景。比如你需要买一件国外的产品，这时候你可以通过代购来购买产品。

在实际代码中其实代理的场景很多，也就不举框架中的例子了，比如事件代理就用到了代理模式。

```
<ul id="ul">
  <li>1</li>
  <li>2</li>
  <li>3</li>
  <li>4</li>
  <li>5</li>
</ul>
<script>
  let ul = document.querySelector('#ul')
  ul.addEventListener('click', (event) => {
    console.log(event.target);
  })
</script>
```

xml 复制代码

因为存在太多的 `li`，不可能每个都去绑定事件。这时候可以通过给父节点绑定一个事件，让父节点作为代理去拿到真实点击的节点。

## 发布-订阅模式

发布-订阅模式也叫做观察者模式。通过一对一或者一对多的依赖关系，当对象发生改变时，订阅方都会收到通知。在现实生活中，也有很多类似场景，比如我需要在购物网站上购买一个产品，但是发现该产品目前处于缺货状态，这时候我可以点击有货通知的按钮，让网站在产品有货的时候通过短信通知我。

在实际代码中其实发布-订阅模式也很常见，比如我们点击一个按钮触发了点击事件就是使用了该模式

xml 复制代码

```
<ul id="ul"></ul>
<script>
  let ul = document.querySelector('#ul')
  ul.addEventListener('click', (event) => {
    console.log(event.target);
  })
</script>
```