```
        // JS 代码需要继续查找是否有依赖关系
        const child = readCode(absolutePath)
        child.relativePath = relativePath
        dependencies.push(child)
      }
    })
  }
  return dependencies
}
```

- 首先我们读取入口文件，然后创建一个数组，该数组的目的是存储代码中涉及到的所有文件
- 接下来我们遍历这个数组，一开始这个数组中只有入口文件，在遍历的过程中， 如果入口文件有依赖其他的文件，那么就会被 `push` 到这个数组中
- 在遍历的过程中， 我们先获得该文件对应的目录，然后遍历当前文件的依赖关系
- 在遍历当前文件依赖关系的过程中， 首先生成依赖文件的绝对路径，然后判断当前文件是 `CSS` 文件还是 `JS` 文件
- 如果是 `CSS` 文件的话， 我们就不能用 `Babel` 去编译了， 只需要读取 `CSS` 文件中的代码，然后创建一个 `style` 标签，将代码插入进标签并且放入 `head` 中即可
- 如果是 `JS` 文件的话， 我们还需要分析 `JS` 文件是否还有别的依赖关系
- 最后将读取文件后的对象 `push` 进数组中
- 现在我们已经获取到了所有的依赖文件，接下来就是实现打包的功能了

```
function bundle(dependencies, entry) {
  let modules = ''
  // 构建函数参数，生成的结构为
  // { './entry.js': function(module, exports, require) { 代码 } }
  dependencies.forEach(dep => {
    const filePath = dep.relativePath || entry
    modules += `'${filePath}' : (
      function (module, exports, require) { ${dep.code} }
    ),`
  })
  // 构建 require 函数， 目的是为了获取模块暴露出来的内容
  const result = `
    (function(modules) {
      function require(id) {
        const module = { exports : {} }
        modules[id](module, module.exports, require)
        return module.exports
      }
      require('${entry}')
```

```
  })({${modules}})
`
// 当生成的内容写入到文件中
fs.writeFileSync( './bundle.js', result)
}
```

> 这段代码需要结合着 Babel 转换后的代码来看， 这样大家就能理解为什么需要这样写了

```js
// entry.js
var _a = require( './a.js')
var _a2 = _interopRequireDefault(_a)
function _interopRequireDefault(obj) {
    return obj && obj.__esModule ? obj : { default: obj }
}
console.log(_a2.default)
// a.js
Object.defineProperty(exports, '__esModule', {
    value: true
})
var a = 1
exports.default = a
```

> Babel 将我们 ES6 的模块化代码转换为了 CommonJS 的代码，但是浏览器是不支持 CommonJS 的，所以如果这段代码需要在浏览器环境下运行的话，我们需要自己实现 CommonJS 相关的代码， 这就是 bundle 函数做的大部分事情。

接下来我们再来逐行解析 bundle 函数

- 首先遍历所有依赖文件，构建出一个函数参数对象
- 对象的属性就是当前文件的相对路径，属性值是一个函数， 函数体是当前文件下的代码，函数接受三个参数 module 、 exports 、 require
  - module 参数对应 CommonJS 中的 module
  - exports 参数对应 CommonJS 中的 module.export
  - require 参数对应我们自己创建的 require 函数
- 接下来就是构造一个使用参数的函数了， 函数做的事情很简单，就是内部创建一个 require 函数，然后调用 require(entry) ，也就是 require('./entry.js') ， 这样

就会从函数参数中找到 `./entry.js` 对应的函数并执行， 最后将导出的内容通过 `module.export` 的方式让外部获取到

- 最后再将打包出来的内容写入到单独的文件中

> 如果你对于上面的实现还有疑惑的话， 可以阅读下打包后的部分简化代码

```js
;(function(modules) {
  function require(id) {
    // 构造一个 CommonJS 导出代码
    const module = { exports: {} }
    // 去参数中获取文件对应的函数并执行
    modules [id](module, module.exports, require)
    return module.exports
  }
  require( './entry.js')
})({
  './entry.js': function(module, exports, require) {
    // 这里继续通过构造的 require 去找到 a.js 文件对应的函数
    var _a = require( './a.js')
    console.log(_a2.default)
  },
  './a.js': function(module, exports, require) {
    var a = 1
    // 将 require 函数中的变量 module 变成了这样的结构
    // module.exports = 1
    // 这样就能在外部取到导出的内容了
    exports.default = a
  }
  // 省略
})
```

> 虽然实现这个工具只写了不到 `100` 行的代码，但是打包工具的核心原理就是这些了

- 找出入口文件所有的依赖关系
- 然后通过构建 `CommonJS` 代码来获取 `exports` 导出的内容

27 MVVM/虚拟DOM/前端路由

27.1 MVVM

> 涉及面试题：什么是 MVVM ？ 比之 MVC 有什么区别?

首先先来说下 View 和 Model

- View 很简单，就是用户看到的视图
- Model 同样很简单，一 般就是本地数据和数据库中的数据

> 基本上， 我们写的产品就是通过接口从数据库中读取数据，然后将数据经过处理展现到用户看到的视图上 。当然我们还可以从视图上读取用户的输入 ，然后又将用户的输入通过接口写入到数据库中 。但是， 如何将数据展示到视图上，然后又如何将用户的输入写入到数据中，不同的人就产生了不同的看法，从此出现了很多种架构设计。

> 传统的 MVC 架构通常是使用控制器更新模型，视图从模型中获取数据去渲染 。当用户有输入时，会通过控制器去更新模型， 并且通知视图进行更新

- 但是 MVC 有一个巨大的缺陷就是控制器承担的责任太大了， 随着项目愈加复杂，控制器中的代码会越来越臃肿， 导致出现不利于维护的情况。
- 在 MVVM 架构中， 引入了 ViewModel 的概念 。 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 。

- 以 Vue 框架来举例， ViewModel 就是组件的实例 。 View 就是模板， Model 的话在引入 Vuex 的情况下是完全可以和组件分离的。
- 除了以上三个部分， 其实在 MVVM 中还引入了一个隐式的 Binder 层， 实现了 View 和 ViewModel 的绑定

- 同样以 Vue 框架来举例， 这个隐式的 Binder 层就是 Vue 通过解析模板中的插值和指令从而实现 View 与 ViewModel 的绑定。
- 对于 MVVM 来说， 其实最重要的并不是通过双向绑定或者其他的方式将 View 与 ViewModel 绑定起来， 而是通过 ViewModel 将视图中的状态和用户的行为分离出一个

抽象， 这才是 MVVM 的精髓

27.2 Virtual DOM

> 涉及面试题：什么是 Virtual DOM ？为什么 Virtual DOM 比原生 DOM 快?

- 大家都知道操作 DOM 是很慢的， 为什么慢的原因以及在「浏览器渲染原理」章节中说过， 这里就不再赘述了- 那么相较于 DOM 来说， 操作 JS 对象会快很多， 并且我们也可以通过 JS 来模拟 DOM

```js
const ul = {
  tag: 'ul',
  props: {
    class: 'list'
  },
  children: {
    tag: 'li',
    children: '1'
  }
}
```

上述代码对应的 DOM 就是

```html
<ul class= 'list '>
  <li>1</li>
</ul>
```

- 那么既然 DOM 可以通过 JS 对象来模拟， 反之也可以通过 JS 对象来渲染出对应的 DOM 。当然了， 通过 JS 来模拟 DOM 并且渲染对应的 DOM 只是第一步，难点在于如何判断新旧两个 JS 对象的最小差异并且实现局部更新 DOM

> 首先 DOM 是一个多叉树的结构， 如果需要完整的对比两颗树的差异，那么需要的时间复杂度会是 O(n ^ 3) ， 这个复杂度肯定是不能接受的 。于是 React 团队优化了算法， 实现了 O(n) 的复杂度来对比差异 。 实现 O(n) 复杂度的关键就是只对比同层的节点， 而不是跨层对比， 这也是考虑到在实际业务中很少会去跨层的移动 DOM 元素 。 所以判断差异的算法就分为了两步

- 首先从上至下， 从左往右遍历对象，也就是树的深度遍历， 这一步中会给每个节点添加索引，便于最后渲染差异
- 一旦节点有子元素，就去判断子元素是否有不同

> 在第一步算法中我们需要判断新旧节点的 tagName 是否相同， 如果不相同的话就代表节点被替换了 。如果没有更改 tagName 的话，就需要判断是否有子元素，有的话就进行第二步算法。

> 在第二步算法中， 我们需要判断原本的列表中是否有节点被移除，在新的列表中需要判断是否有新的节点加入， 还需要判断节点是否有移动。

举个例子来说，假设页面中只有一个列表， 我们对列表中的元素进行了变更

```js
// 假设这里模拟一个 ul， 其中包含了 5 个 li
[1, 2, 3, 4, 5]
// 这里替换上面的 li
[1, 2, 5, 4]
```

> 从上述例子中， 我们一眼就可以看出先前的 ul 中的第三个 li 被移除了，四五替换了位置。

> 那么在实际的算法中， 我们如何去识别改动的是哪个节点呢？这就引入了 key 这个属性，想必大家在 Vue 或者 React 的列表中都用过这个属性。这个属性是用来给每一个节点打标志的，用于判断是否是同一个节点。

- 当然在判断以上差异的过程中， 我们还需要判断节点的属性是否有变化等等。
- 当我们判断出以上的差异后，就可以把这些差异记录下来 。当对比完两棵树以后，就可以通过差异去局部更新 DOM ， 实现性能的最优化。

> 当然了 Virtual DOM 提高性能是其中一个优势， 其实最大的优势还是在于：

- 将 Virtual DOM 作为一个兼容层，让我们还能对接非 Web 端的系统， 实现跨端开发。
- 同样的， 通过 Virtual DOM 我们可以渲染到其他的平台， 比如实现 SSR 、同构渲染等等。
- 实现组件的高度抽象化

27.3 路由原理

> 涉及面试题：前端路由原理？两种实现方式有什么区别？

> 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， 并且无须刷新页面 。 目前前端使用的路由就只有两种实现方式

- Hash 模式
- History 模式

1. Hash 模式

> www.test.com/#/ 就是 Hash URL ， 当 # 后面的哈希值发生变化时， 可以通过 hashchange 事件来监听到 URL 的变化，从而进行跳转页面， 并且无论哈希值如何变化， 服务端接收到的 URL 请求永远是 www.test.com

```js
window.addEventListener( 'hashchange', () => {
  // ... 具体逻辑
})
```

> Hash 模式相对来说更简单， 并且兼容性也更好

2. History 模式

> History 模式是 HTML5 新推出的功能， 主要使用 history.pushState 和 history.replaceState 改变 URL

- 通过 History 模式改变 URL 同样不会引起页面的刷新， 只会更新浏览器的历史记录。

```js
// 新增历史记录
history.pushState(stateObject, title, URL)
```

```
// 替换当前历史记录
history.replaceState(stateObject, title, URL)
```

> 当用户做出浏览器动作时，比如点击后退按钮时会触发 popState 事件

```js
window.addEventListener('popstate', e => {
  // e.state 就是 pushState(stateObject) 中的 stateObject
  console.log(e.state)
})
```

两种模式对比

- Hash 模式只可以更改 # 后面的内容，History 模式可以通过 API 设置任意的同源 URL
- History 模式可以通过 API 添加任意类型的数据到历史记录中，Hash 模式只能更改哈希值，也就是字符串
- Hash 模式无需后端配置，并且兼容性好。History 模式在用户手动输入地址或者刷新页面的时候会发起 URL 请求，后端需要配置 index.html 页面用于匹配不到静态资源的时候

27.4 Vue 和 React 之间的区别

- Vue 的表单可以使用 v-model 支持双向绑定，相比于 React 来说开发上更加方便，当然了 v-model 其实就是个语法糖，本质上和 React 写表单的方式没什么区别
- 改变数据方式不同，Vue 修改状态相比来说要简单许多，React 需要使用 setState 来改变状态，并且使用这个 API 也有一些坑点。并且 Vue 的底层使用了依赖追踪，页面更新渲染已经是最优的了，但是 React 还是需要用户手动去优化这方面的问题。
- React 16 以后，有些钩子函数会执行多次，这是因为引入 Fiber 的原因
- React 需要使用 JSX，有一定的上手成本，并且需要一整套的工具链支持，但是完全可以通过 JS 来控制页面，更加的灵活。Vue 使用了模板语法，相比于 JSX 来说没有那么灵活，但是完全可以脱离工具链，通过直接编写 render 函数就能在浏览器中运行。
- 在生态上来说，两者其实没多大的差距，当然 React 的用户是远远高于 Vue 的

28 Vue常考知识点

28.1 生命周期钩子函数

在 beforeCreate 钩子函数调用的时候， 是获取不到 props 或者 data 中的数据的，因为这些数据的初始化都在 initState 中。

然后会执行 created 钩子函数，在这一步的时候已经可以访问到之前不能访问到的数据，但是这时候组件还没被挂载，所以是看不到的。

接下来会先执行 beforeMount 钩子函数， 开始创建 VDOM ， 最后执行 mounted 钩子， 并将 VDOM 渲染为真实 DOM 并且渲染数据 。组件中如果有子组件的话，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 beforeUpdate 和 updated ， 这两个钩子函数没什么好说的，就是分别在数据更新前和更新后会调用。
- 另外还有 keep-alive 独有的生命周期，分别为 activated 和 deactivated 。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 actived 钩子函数。
- 最后就是销毁组件的钩子函数 beforeDestroy 和 destroyed 。前者适合移除事件 、定时器等等， 否则可能会引起内存泄露的问题 。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才会执行根组件的 destroyed 钩子函数

28.2 组件通信

> 组件通信一般分为以下几种情况：

- 父子组件通信
- 兄弟组件通信
- 跨多层级组件通信

> 对于以上每种情况都有多种方式去实现，接下来就来学习下如何实现。

1. 父子通信

- 父组件通过 props 传递数据给子组件， 子组件通过 emit 发送事件传递数据给父组件，这两种方式是最常用的父子通信实现办法。
- 这种父子通信方式也就是典型的单向数据流， 父组件通过 props 传递数据， 子组件不能直接修改 props ， 而是必须通过发送事件的方式告知父组件修改数据。
- 另外这两种方式还可以使用语法糖 v-model 来直接实现， 因为 v-model 默认会解析成名为 value 的 prop 和名为 input 的事件 。这种语法糖的方式是典型的双向绑定，常用于 UI 控件上， 但是究其根本， 还是通过事件的方法让父组件修改数据。

- 当然我们还可以通过访问 `$parent` 或者 `$children` 对象来访问组件实例中的方法和数据。
- 另外如果你使用 Vue 2.3 及以上版本的话还可以使用 `$listeners` 和 `.sync` 这两个属性。
- `$listeners` 属性会将父组件中的 (不含 `.native` 修饰器的) `v-on` 事件监听器传递给子组件， 子组件可以通过访问 `$listeners` 来自定义监听器。
- `.sync` 属性是个语法糖， 可以很简单的实现子组件与父组件通信

```html
<!--父组件中-->
<input :value.sync="value" />
<!--以上写法等同于-->
<input :value="value" @update:value="v => value = v"></comp>
<!--子组件中-->
<script>
  this.$emit( 'update:value', 1)
</script>
```

2. 兄弟组件通信

> 对于这种情况可以通过查找父组件中的子组件实现，也就是 `this.$parent.$children` ，在 `$children` 中可以通过组件 `name` 查询到需要的组件实例，然后进行通信。

3. 跨多层次组件通信

> 对于这种情况可以使用 `Vue 2.2` 新增的 `API provide / inject` ， 虽然文档中不推荐直接使用在业务中，但是如果用得好的话还是很有用的。

假设有父组件 `A` ，然后有一个跨多层级的子组件 `B`

```js
// 父组件 A
export default {
  provide: {
    data: 1
  }
}
// 子组件 B
export default {
  inject: [ 'data'],
  mounted() {
```

```
    // 无论跨几层都能获得父组件的 data 属性
    console.log(this.data) // => 1
  }
}
```

终极办法解决一切通信问题

> 只要你不怕麻烦， 可以使用 Vuex 或者 Event Bus 解决上述所有的通信情况。

28.3 extend 能做什么

> 这个 API 很少用到，作用是扩展组件生成一个构造器， 通常会与 $mount 一起使用。

```js
// 创建组件构造器
let Component = Vue.extend({
  template: '<div>test</div>'
})
// 挂载到 #app 上
new Component().$mount( '#app')
// 除了上面的方式， 还可以用来扩展已有的组件
let SuperComponent = Vue.extend(Component)
new SuperComponent({
    created() {
        console.log(1)
    }
})
new SuperComponent().$mount( '#app')
```

28.4 mixin 和 mixins 区别

> mixin 用于全局混入，会影响到每个组件实例， 通常插件都是这样做初始化的

```js
Vue.mixin({
    beforeCreate() {
```

```
        // ...逻辑
        // 这种方式会影响到每个组件的 beforeCreate 钩子函数
    }
})
```

- 虽然文档不建议我们在应用中直接使用 `mixin` ，但是如果不滥用的话也是很有帮助的，比如可以全局混入封装好的 `ajax` 或者一些工具函数等等。
- `mixins` 应该是我们最常使用的扩展组件的方式了。如果多个组件中有相同的业务逻辑，就可以将这些逻辑剥离出来， 通过 `mixins` 混入代码， 比如上拉下拉加载数据这种逻辑等等。
- 另外需要注意的是 `mixins` 混入的钩子函数会先于组件内的钩子函数执行， 并且在遇到同名选项的时候也会有选择性的进行合并， 具体可以阅读 文档。

28.5 computed 和 watch 区别

- `computed` 是计算属性，依赖其他属性计算值， 并且 `computed` 的值有缓存， 只有当计算值变化才会返回内容。
- `watch` 监听到值的变化就会执行回调，在回调中可以进行一些逻辑操作。
- 所以一般来说需要依赖别的属性来动态获得值的时候可以使用 `computed` ，对于监听到值的变化需要做一些复杂业务逻辑的情况可以使用 `watch` 。
- 另外 `computer` 和 `watch` 还都支持对象的写法， 这种方式知道的人并不多。

```
vm.$watch( 'obj', {
    // 深度遍历
    deep: true,
    // 立即触发
    immediate: true,
    // 执行的函数
    handler: function(val, oldVal) {}
})
var vm = new Vue({
  data: { a: 1 },
  computed: {
    aPlus: {
      // this.aPlus 时触发
      get: function () {
        return this.a + 1
      },
      // this.aPlus = 1 时触发
      set: function (v) {
        this.a = v - 1
      }
```

```
    }
  }
})
```

28.6 keep-alive 组件有什么作用

- 如果你需要在组件切换的时候，保存一些组件的状态防止多次渲染，就可以使用 `keep-alive` 组件包裹需要保存的组件。
- 对于 `keep-alive` 组件来说， 它拥有两个独有的生命周期钩子函数，分别为 `activated` 和 `deactivated` 。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 `actived` 钩子函数。

28.7 v-show 与 v-if 区别

- `v-show` 只是在 `display: none` 和 `display: block` 之间切换 。无论初始条件是什么都会被渲染出来，后面只需要切换 `CSS` ， `DOM` 还是一直保留着的 。所以总的来说 `v-show` 在初始渲染时有更高的开销，但是切换开销很小，更适合于频繁切换的场景。
- `v-if` 的话就得说到 `Vue` 底层的编译了 。当属性初始为 `false` 时， 组件就不会被渲染， 直到条件为 `true` ， 并且切换条件时会触发销毁/挂载组件，所以总的来说在切换时开销更高，更适合不经常切换的场景。
- 并且基于 `v-if` 的这种惰性渲染机制， 可以在必要的时候才去渲染组件，减少整个页面的初始渲染开销。

28.8 组件中 data 什么时候可以使用对象

> 这道题目其实更多考的是 JS 功底。

- 组件复用时所有组件实例都会共享 `data` ， 如果 `data` 是对象的话，就会造成一个组件修改 `data` 以后会影响到其他所有组件，所以需要将 `data` 写成函数，每次用到就调用一次函数获得新的数据。
- 当我们使用 `new Vue()` 的方式的时候， 无论我们将 `data` 设置为对象还是函数都是可以的， 因为 `new Vue()` 的方式是生成一个根组件，该组件不会复用，也就不存在共享 `data` 的情况了

> 以下是进阶部分

28.9 响应式原理

> Vue 内部使用了 Object.defineProperty() 来实现数据响应式， 通过这个函数可以监听到 set 和 get 的事件

```js
var data = { name: 'poetries' }
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, {
    // 可枚举
    enumerable: true,
    // 可配置
    configurable: true,
    // 自定义函数
    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， 因为自定义的函数一开始是不会执行的 。只有先执行了依赖收集，从能在属性更新的时候派发更新，所以接下来我们需要先触发依赖收集

```html
<div>
    {{name}}
</div>
```

- 在解析如上模板代码时， 遇到 就会进行依赖收集。
- 接下来我们先来实现一个 `Dep` 类， 用于解耦属性的依赖收集和派发更新操作

```js
// 通过 Dep 解耦属性的依赖和更新操作
class Dep {
  constructor() {
    this.subs = []
  }
  // 添加依赖
  addSub(sub) {
    this.subs.push(sub)
  }
  // 更新
  notify() {
    this.subs.forEach(sub => {
      sub.update()
    })
  }
}
// 全局属性，通过该属性配置 Watcher
Dep.target = null
```

> 以上的代码实现很简单， 当需要依赖收集的时候调用 `addSub` ， 当需要派发更新的时候调用 `notify` 。

> 接下来我们先来简单的了解下 `Vue` 组件挂载时添加响应式的过程 。在组件挂载时，会先对所有需要的属性调用 `Object.defineProperty()` ，然后实例化 `Watcher` ，传入组件更新的回调 。在实例化过程中，会对模板中的属性进行求值，触发依赖收集。

因为这一小节主要目的是学习响应式原理的细节，所以接下来的代码会简略的表达触发依赖收集时的操作。

```js
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)
  }
}
```

> 以上就是 `Watcher` 的简单实现，在执行构造函数的时候将 `Dep.target` 指向自身，从而使得收集到了对应的 `Watcher` ，在派发更新的时候取出对应的 `Watcher` 然后执行 `update` 函数。

接下来， 需要对 `defineReactive` 函数进行改造，在自定义函数中添加依赖收集和派发更新相关的代码

```js
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()
    }
  })
}
```

> 以上所有代码实现了一个简易的数据响应式，核心思路就是手动触发一次属性的 `getter` 来实现依赖收集。

现在我们就来测试下代码的效果， 只需要把所有的代码复制到浏览器中执行，就会发现页面的内容全部被替换了

```js
var data = { name: 'poetries' }
observe(data)
function update(value) {
  document.querySelector( 'div').innerText = value
}
// 模拟解析到 `{{name}}` 触发的操作
new Watcher(data, 'name', update)
// update Dom innerText
data.name = 'yyy'
```

28.9.1 Object.defineProperty 的缺陷

- 以上已经分析完了 `Vue` 的响应式原理，接下来说一点 `Object.defineProperty` 中的缺陷。
- 如果通过下标方式修改数组数据或者给对象新增属性并不会触发组件的重新渲染， 因为 `Object.defineProperty` 不能拦截到这些操作，更精确的来说，对于数组而言，大部分操作都是拦截不到的， 只是 `Vue` 内部通过重写函数的方式解决了这个问题。
- 对于第一个问题， `Vue` 提供了一个 `API` 解决

```js
export function set (target: Array<any> | Object, key: any, val: any): any
  // 判断是否为数组且下标是否有效
  if (Array.isArray(target) && isValidArrayIndex(key)) {
```