data
computed
methods
watch

<script> </script> 脚本
<style> </style> 样式
<template> </template> 模板

} 3) *.vue .

Vue 文档 Learn.

Hello world

```
<div id="app">
  {{ message }}
</div>
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

el：对应的container

data：model(对应的数据)

v-bind: bind dom的 prop.

```
<div id="app-2">
  <span v-bind:title="message">
    鼠标悬停几秒钟查看此处动态绑定的提示信息！
  </span>
</div>
var app2 = new Vue({
  el: '#app-2',
  data: {
    message: '页面加载于 ' + new Date().toLocaleString()
  }
})
```

鼠标悬停几秒钟查看此处动态绑定的提示信息！

v-bind:title ="message"

v-bind:title => :title 缩写.

将 span 的 title prop 和 message 绑定.

v-if.

```
<div id="app-3">
  <p v-if="seen">现在你看到我了</p>
</div>
var app3 = new Vue({
  el: '#app-3',
  data: {
    seen: true
  }
})
```

v-for

```
<div id="app-4">
  <ol>
    <li v-for="todo in todos">
      {{ todo.text }}
    </li>
  </ol>
</div>
var app4 = new Vue({
  el: '#app-4',
  data: {
    todos: [
      { text: '学习 JavaScript' },
      { text: '学习 Vue' },
      { text: '整个牛项目' }
    ]
  }
})
```

列表.

v-on 绑定一个事件 Listener

```
<div id="app-5">
  <p>{{ message }}</p>
  <button v-on:click="reverseMessage">逆转消息</button>
</div>
var app5 = new Vue({
  el: '#app-5',
  data: {
    message: 'Hello Vue.js!'
  },
  methods: {
    reverseMessage: function () {
      this.message = this.message.split('').reverse().join('')
    }
  }
})
```

@click=xx 事件.

container.

数据

方法.

v-model: 双向 bind.

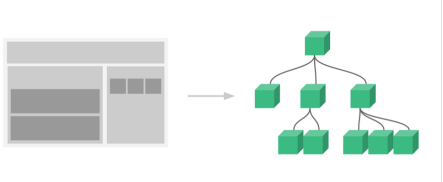
```
<div id="app-6">
  <p>{{ message }}</p>
  <input v-model="message">
</div>
var app6 = new Vue({
  el: '#app-6',
  data: {
    message: 'Hello Vue!'
  }
})
```

message <=> 用户的input.

双向 bind.

v-model.

组件



```
Vue.component('todo-item', {
  // todo-item 组件现在接受一个
  // "prop"，类似于一个自定义属性
  // 这个属性名为 todo。
  props: ['todo'],
  template: '<li>{{ todo.text }}</li>'
})
现在，我们可以使用 v-bind 指令将 todo 传到每一个重复的组件中：
<div id="app-7">
  <ol>
    <!--
      现在我们为每个 todo-item 提供 todo 对象
      todo 对象是变量，即其内容可以是动态的。
      我们也需要为每个组件提供一个“key”，晚些时候我们会做个解释。
    -->
    <todo-item
      v-for="item in groceryList"
      v-bind:todo="item"
      v-bind:key="item.id">
    </todo-item>
  </ol>
</div>
```

通过 props 实现与父组件的解耦.

Vue 根实例

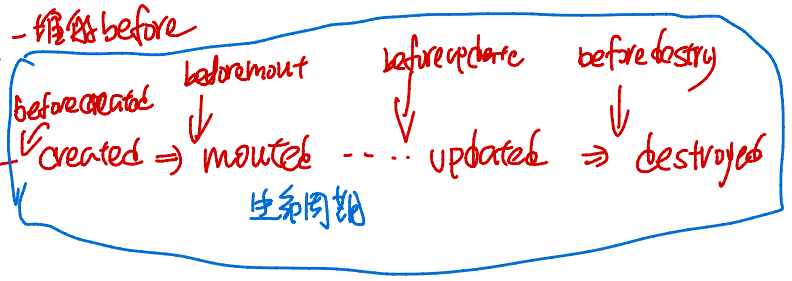
```
var vm = new Vue({
  // 选项
})
```

在实例化 Vue 时，需要传入一个选项对象，它可以包含数据、模板、挂载元素、方法、生命周期钩子等选项。全部的选项可以在 API 文档中查看。

data

template

要mounted的地方



Vue 会代理 data 的所有属性

```
var data = { a: 1 }
var vm = new Vue({
  data: data
})
vm.a === data.a // -> true
// 设置属性也会影响到原始数据
vm.a = 2
data.a // -> 2
// ... 反之亦然
data.a = 3
vm.a // -> 3
```

只有这些被代理的属性是响应的，也就是说值的任何改变都是触发视图的重新渲染

会触发 render

代理？ Vue 中怎么实现这样的代理？ ES6 中的 Property 怎样定义？ setter？ getter？

Vue 有用的属性和方法：以 $ 开始

```
var data = { a: 1 }
var vm = new Vue({
  el: '#example',
  data: data
})
vm.$data === data // -> true
vm.$el === document.getElementById('example') // -> true
// $watch 是一个实例方法
vm.$watch('a', function (newVal, oldVal) {
  // 这个回调将在 `vm.a` 改变后调用
})
```

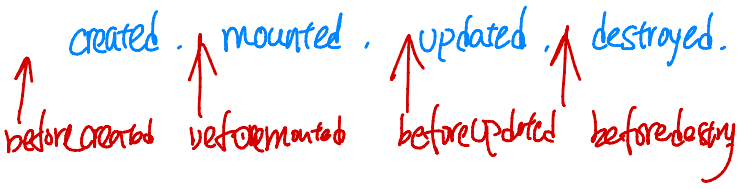
$data , $el , $watch.

$watch 很像 ng 中的 scope watch

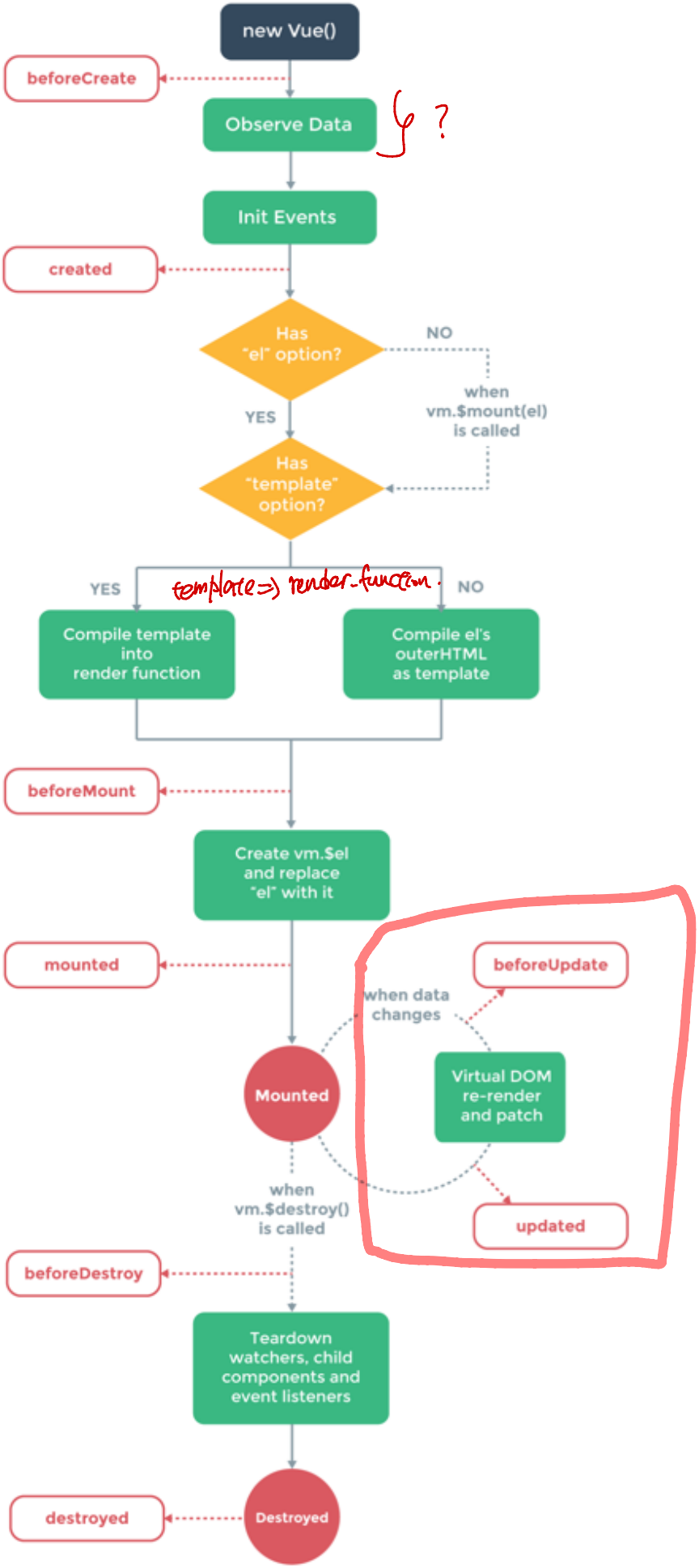
箭头函数：arrow function：this 绑定

注意，不要在实例属性或者回调函数中（如 vm.$watch('a', newVal => this.myMethod())）使用箭头函数。因为箭头函数绑定父级上下文，所以 this 不会像预想的一样是 Vue 实例，而且 this.myMethod 是未被定义的

Vue instance 的生命周期：life cycle



Vue 实例的生命周期



核心：更新. ⇆ Render cycle

核心更新函数

Vue template.

在底层的实现上， Vue 将模板编译成虚拟 DOM 渲染函数，结合响应系统，在应用状态改变时， Vue 能够智能地计算出重新渲染组件的最小代价并应用到 DOM 操作上。

virtual DOM.

template ⇒ render function.

如果你熟悉虚拟 DOM 并且偏爱 JavaScript 的原始力量，你也可以不用模板，直接写渲染（render）函数，使用可选的 JSX 语法

JSX 语法的 render function.

```
<span>Message: {{ msg }}</span>
```

{{ msg }} 语法.改.

v-bind bind到Component的属性.

```
<div v-bind:id="dynamicId"></div>

<button v-bind:disabled="isButtonDisabled">Button</button>
```

表达式

```
{{ number + 1 }}
{{ ok ? 'YES' : 'NO' }}
{{ message.split('').reverse().join('') }}
<div v-bind:id="'list-' + id"></div>
```

单个表达式.

模板表达式都被放在沙盒中，只能访问全局变量的一个白名单，如 Math 和 Date 。你不应该在模板表达式中试图访问用户定义的全局变量。

```
<form v-on:submit.prevent="onSubmit"></form>
```

.prevent 修饰符告诉 v-on 指令对于触发的事件调用 event.preventDefault()

修饰符

过滤，字符格式转换

```
<!-- in mustaches -->
{{ message | capitalize }}
<!-- in v-bind -->
<div v-bind:id="rawId | formatId"></div>
```

格式转换

由于最初计划过滤器的使用场景，是用于文本转换，所以 Vue 2.x 过滤器只能用于双花括号插值(mustache interpolation)和 v-bind 表达式中（后者在 2.1.0+ 版本支持）。对于复杂数据的转换，应该使用计算属性。

v-bind 的缩写 : 省略

```
<!-- 完整语法 -->
<a v-bind:href="url"></a>
<!-- 缩写 -->
<a :href="url"></a>
```

计算属性

```
<div id="example">
  <p>Original message: "{{ message }}"</p>
  <p>Computed reversed message: "{{ reversedMessage }}"</p>
</div>
var vm = new Vue({
  el: '#example',
  data: {
    message: 'Hello'
  },
  computed: {
    // a computed getter
    reversedMessage: function () {
      // `this` points to the vm instance
      return this.message.split('').reverse().join('')
    }
  }
})
```

computed 计算属性，对data依赖，双向

getter 在这没有副作用.

computed：对依赖的属性值缓存

Cache

v-on 的缩写 @

```
<!-- 完整语法 -->
<a v-bind:href="url"></a>
<!-- 缩写 -->
<a :href="url"></a>
```

```
computed: {
  fullName: {
    // getter
    get: function () {
      return this.firstName + ' ' + this.lastName
    },
    // setter
    set: function (newValue) {
      var names = newValue.split(' ')
      this.firstName = names[0]
      this.lastName = names[names.length - 1]
    }
  }
}
```

计算属性的 setter

```
watch: {
  // 如果 question 发生改变，这个函数就会运行
  question: function (newQuestion) {
    this.answer = 'Waiting for you to stop typing...'
    this.getAnswer()
  }
},
```

Watch

Class 与 Style 绑定.

:class

```
<div v-bind:class="{ active: isActive }"></div>
```

有点像 React 中的 className 插件.

```
<div class="static"
     v-bind:class="{ active: isActive, 'text-danger': hasError }">
</div>
```

```
<div v-bind:class="classObject"></div>
data: {
  classObject: {
    active: true,
    'text-danger': false
  }
}
```

bind到数据属性上.

```
<div v-bind:class="classObject"></div>
data: {
  isActive: true,
  error: null
},
computed: {
  classObject: function () {
    return {
      active: this.isActive && !this.error,
      'text-danger': this.error && this.error.type === 'fatal',
    }
  }
}
```

bind到计算属性上.

```
<div v-bind:class="[activeClass, errorClass]">
data: {
  activeClass: 'active',
  errorClass: 'text-danger'
}
```

数组语法

```
Vue.component('my-component', {
  template: '<p class="foo bar">Hi</p>'
})
然后在使用它的时候添加一些 class：
<my-component class="baz boo"></my-component>
HTML 最终将被渲染成为：
<p class="foo bar baz boo">Hi</p>
同样的适用于绑定 HTML class:
<my-component v-bind:class="{ active: isActive }"></my-component>
```

bind到 自定义的组件.

bind css style. 使用驼峰命名法代替短横线.

```
<div v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>
data: {
  activeColor: 'red',
  fontSize: 30
}
```

```
<div v-bind:style="styleObject"></div>
data: {
  styleObject: {
    color: 'red',
    fontSize: '13px'
  }
}
```

fontSize ⇒ font-size

条件渲染：

v-if、v-else、v-else-if

```
<div v-if="Math.random() > 0.5">
  Now you see me
</div>
<div v-else>
  Now you don't
</div>
```

保留在计数器。

```
  A
</div>
<div v-else-if="type === 'B'">
  B
</div>
<div v-else-if="type === 'C'">
  C
</div>
<div v-else>
  Not A/B/C
</div>
```

v-else-if .

key 管理可复用的元素是完全独立的。

```
<template v-if="loginType === 'username'">
  <label>Username</label>
  <input placeholder="Enter your username" key="username-input">
</template>
<template v-else>
  <label>Email</label>
  <input placeholder="Enter your email address" key="email-input">
</template>
```

v-show

v-for：列表渲染。

```
<ul id="example-1">
  <li v-for="item in items">
    {{ item.message }}
  </li>
</ul>
var example1 = new Vue({
  el: '#example-1',
  data: {
    items: [
      {message: 'Foo' },
      {message: 'Bar' }
    ]
  }
})
```

```
<ul>
  <template v-for="item in items">
    <li>{{ item.msg }}</li>
    <li class="divider"></li>
  </template>
</ul>
```

数组检测更新的限制。

由于 JavaScript 的限制， Vue 不能检测以下变动的数组：

当你利用索引直接设置一个项时，例如：vm.items[indexOfItem] = newValue

当你修改数组的长度时，例如：vm.items.length = newLength

使用 splice 代替。

```
// Vue.set
Vue.set(example1.items, indexOfItem, newValue)
// Array.prototype.splice
example1.items.splice(indexOfItem, 1, newValue)
```

为了解决第二类问题，你可以使用 splice：

```
example1.items.splice(newLength)
```

value key

```
<ul id="repeat-object" class="demo">
  <li v-for="value in object">
    {{ value }}
  </li>
</ul>
new Vue({
  el: '#repeat-object',
  data: {
    object: {
      firstName: 'John',
      lastName: 'Doe',
      age: 30
    }
  }
})
```

```
<div v-for="(value, key) in object">
  {{ key }} : {{ value }}
</div>
```

value, key, index

```
<div v-for="(value, key, index) in object">
  {{ index }}. {{ key }} : {{ value }}
</div>
```

range(10)

```
<div>
  <span v-for="n in 10">{{ n }} </span>
</div>
```

```
<div v-for="item in items" :key="item.id">
  <!-- 内容 -->
</div>
```

指定子元素的 key .

事件处理：

v-on 缩写 @ . @click

v-on:click

```
<div id="example-1">
  <button v-on:click="counter += 1">增加 1</button>
  <p>这个按钮被点击了 {{ counter }} 次。</p>
</div>
var example1 = new Vue({
  el: '#example-1',
  data: {
    counter: 0
  }
})
```

执行简单的 JS function

```
<div id="example-2">
  <!-- `greet` 是在下面定义的方法名 -->
  <button v-on:click="greet">Greet</button>
</div>
var example2 = new Vue({
  el: '#example-2',
  data: {
    name: 'Vue.js'
  },
  // 在 `methods` 对象中定义方法
  methods: {
    greet: function (event) {
      // `this` 在方法里指当前 Vue 实例
      alert('Hello ' + this.name + '!')
      // `event` 是原生 DOM 事件
      if (event) {
        alert(event.target.tagName)
      }
    }
  }
})
```

指定 function 的名字，

指定 function 的参数

```
<div id="example-3">
  <button v-on:click="say('hi')">Say hi</button>
  <button v-on:click="say('what')">Say what</button>
</div>
new Vue({
  el: '#example-3',
  methods: {
    say: function (message) {
      alert(message)
    }
  }
})
```

```
<button v-on:click="warn('Form cannot be submitted yet.', $event)">
  Submit
</button>
```

把 event 传过去

事件的修饰符，这语法挺不错

① 事件的修饰符

- .stop
- .prevent
- .capture
- .self
- .once

```
<!-- 阻止单击事件冒泡 -->
<a v-on:click.stop="doThis"></a>
<!-- 提交事件不再重载页面 -->
<form v-on:submit.prevent="onSubmit"></form>
<!-- 修饰符可以串联  -->
<a v-on:click.stop.prevent="doThat"></a>
<!-- 只有修饰符 -->
<form v-on:submit.prevent></form>
<!-- 添加事件侦听器时使用事件捕获模式 -->
<div v-on:click.capture="doThis">...</div>
<!-- 只当事件在该元素本身（比如不是子元素）触发时触发回调 -->
<div v-on:click.self="doThat">...</div>
```

② key 的 modifier

```
<!-- 同上 -->
<input v-on:keyup.enter="submit">
<!-- 缩写语法 -->
<input @keyup.enter="submit">
```

- .enter
- .tab
- .delete (捕获 “删除” 和 “退格” 键)
- .esc
- .space
- .up
- .down
- .left
- .right

- .ctrl
- .alt
- .shift
- .meta

```
<!-- Alt + C -->
<input @keyup.alt.67="clear">
<!-- Ctrl + Click -->
<div @click.ctrl="doSomething">Do something</div>
```

③ Mouse 的修饰符.

- .left
- .right
- .middle

表单控件绑定：v-model

input

```
<input v-model="message" placeholder="edit me">
<p>Message is: {{ message }}</p>
```

textarea

```
<span>Multiline message is:</span>
<p style="white-space: pre-line">{{ message }}</p>
<br>
<textarea v-model="message" placeholder="add multiple lines"></textarea>
```

checkbox

多选，bind 到一个数组上。

```
<input type="checkbox" id="jack" value="Jack" v-model="checkedNames">
<label for="jack">Jack</label>
<input type="checkbox" id="john" value="John" v-model="checkedNames">
<label for="john">John</label>
<input type="checkbox" id="mike" value="Mike" v-model="checkedNames">
<label for="mike">Mike</label>
<br>
<span>Checked names: {{ checkedNames }}</span>
```

radiobox

单选 => 一个值

```
<div id="example-4">
  <input type="radio" id="one" value="One" v-model="picked">
  <label for="one">One</label>
  <br>
  <input type="radio" id="two" value="Two" v-model="picked">
  <label for="two">Two</label>
  <br>
  <span>Picked: {{ picked }}</span>
</div>
```

select single select

单选，bind 到一个值上

```
<div id="example-5">
  <select v-model="selected">
    <option disabled value="">请选择</option>
    <option>A</option>
    <option>B</option>
    <option>C</option>
  </select>
  <span>Selected: {{ selected }}</span>
</div>
```

select: multi select

多选，bind，selected 到一个数组上

```
<div id="example-6">
  <select v-model="selected" multiple style="width: 50px">
    <option>A</option>
    <option>B</option>
    <option>C</option>
  </select>
  <br>
  <span>Selected: {{ selected }}</span>
</div>
```

bind 绑定动态 value

```
<!-- 当选中时，`picked` 为字符串 "a" -->
<input type="radio" v-model="picked" value="a">
<!-- `toggle` 为 true 或 false -->
<input type="checkbox" v-model="toggle">
<!-- 当选中时，`selected` 为字符串 "abc" -->
<select v-model="selected">
  <option value="abc">ABC</option>
</select>
```

```
<input
  type="checkbox"
  v-model="toggle"
  v-bind:true-value="a"
  v-bind:false-value="b"
>
```

true-value
false-value

```
<select v-model="selected">
    <!-- 内联对象字面量 -->
  <option v-bind:value="{ number: 123 }">123</option>
</select>
```

修饰符

.lazy

```
<!-- 在 "change" 而不是 "input" 事件中更新 -->
<input v-model.lazy="msg" >
```

.number

自动转为 Number 类型。

```
<input v-model.number="age" type="number">
```

.trim

```
<input v-model.trim="msg">
```

组件 Component.

要注册一个全局组件，你可以使用 Vue.component(tagName, options)。例如：

```
Vue.component('my-component', {
  // 选项
})
```

全局组件.

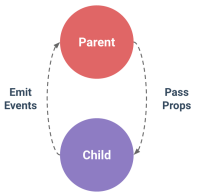
```
var data = { counter: 0 }
Vue.component('simple-counter', {
  template: '<button v-on:click="counter += 1">{{ counter }}</button>',
  // 技术上 data 的确是一个函数了，因此 Vue 不会警告，
  // 但是我们返回给每个组件的实例的却引用了同一个data对象
  data: function () {
    return data
  }
})
new Vue({
  el: '#example-2'
})
```

data 必须是个 function

```
var Child = {
  template: '<div>A custom component!</div>'
}
new Vue({
  // ...
  components: {
    // <my-component> 将只在父模板可用
    'my-component': Child
  }
})
```

局部组件.

父子组件中的通信



父组件向下传递 Props.

子组件向上发送事件. $emit(event-name)

```
Vue.component('child', {
  // 声明 props
  props: ['message'],
  // 就像 data 一样，prop 可以用在模板内
  // 同样也可以在 vm 实例中像“this.message”这样使用
  template: '<span>{{ message }}</span>'
})
然后我们可以这样向它传入一个普通字符串：
<child message="hello!"></child>
```

父组件中传递 Props 给子组件.

HTML 特性是不区分大小写的。所以，当使用的不是字符串模版，camelCased (驼峰式) 命名的 prop 需要转换为相对应的 kebab-case (短横线隔开式) 命名：

```
Vue.component('child', {
  // camelCase in JavaScript
  props: ['myMessage'],
  template: '<span>{{ myMessage }}</span>'
})
<!-- kebab-case in HTML -->
<child my-message="hello!"></child>
```

命名规则

myMessage ⇒ my-message

在模板中，要动态地绑定父组件的数据到子模板的 props，与绑定到任何普通的HTML特性相类似，就是用 v-bind。每当父组件的数据变化时，该变化也会传导给子组件：

```
<div>
  <input v-model="parentMsg">
  <br>
  <child v-bind:my-message="parentMsg"></child>
</div>
```

动态 Props

```
<!-- 传递了一个字符串 "1" -->
<comp some-prop="1"></comp>
```

传递是字符串

因为它是一个字面 prop，它的值是字符串 "1" 而不是 number。如果想传递一个实际的 number，需要使用 v-bind，从而让它的值被当作 JavaScript 表达式计算：

```
<!-- 传递实际的 number -->
<comp v-bind:some-prop="1"></comp>
```

被当作 JS 表达式.

单向数据流

定义一个局部变量，并用 prop 的值初始化它：

```
props: ['initialCounter'],
data: function () {
  return { counter: this.initialCounter }
}
```

定义一个计算属性，处理 prop 的值并返回。

```
props: ['size'],
computed: {
  normalizedSize: function () {
    return this.size.trim().toLowerCase()
  }
}
```

注意在 JavaScript 中对象和数组是引用类型，指向同一个内存空间，如果 prop 是一个对象或数组，在子组件内部改变它会影响父组件的状态。

对象和数组是引用类型.

Props 验证

传入数据不合格时，会报Warning.

```
Vue.component('example', {
  props: {
    // 基础类型检测 (`null` 意思是任何类型都可以)
    propA: Number,
    // 多种类型
    propB: [String, Number],
    // 必传且是字符串
    propC: {
      type: String,
      required: true
    },
    // 数字，有默认值
    propD: {
      type: Number,
      default: 100
    },
    // 数组/对象的默认值应当由一个工厂函数返回
    propE: {
      type: Object,
      default: function () {
        return { message: 'hello' }
      }
    },
    // 自定义验证函数
    propF: {
      validator: function (value) {
        return value > 10
      }
    }
  }
})
```

type 的类型.

- String
- Number
- Boolean
- Function
- Object
- Array
- Symbol

type 也可以是一个自定义构造器函数，使用 instanceof 检测。

<bs-date-input data-3d-date-picker="true"></bs-date-input>
添加属性 data-3d-date-picker="true" 之后，它会被自动添加到 bs-date-input 的根元素上

设置预定义的 props，实现以下功能

通常是这些.

对于多数特性来说，传递给组件的值会覆盖组件本身设定的值。即例如传递 type="large" 将会覆盖 type="date" 且有可能破坏该组件！索性我们对待 class 和 style 特性会更聪明一些，这两个特性的值都会做合并 (merge) 操作，让最终生成的值为：form-control date-picker-theme-dark。

class 和 style 的合并 merge

自定义事件：

每个 Vue 实例都实现了事件接口 (Events interface)，即：
使用 $on(eventName) 监听事件
使用 $emit(eventName) 触发事件

父组件监听

```
<div id="counter-event-example">
  <p>{{ total }}</p>
  <button-counter v-on:increment="incrementTotal"></button-counter>
  <button-counter v-on:increment="incrementTotal"></button-counter>
</div>
Vue.component('button-counter', {
  template: '<button v-on:click="incrementCounter">{{ counter }}</button>',
  data: function () {
    return {
      counter: 0
    }
  },
  methods: {
    incrementCounter: function () {
      this.counter += 1
      this.$emit('increment')
    }
  },
})
new Vue({
  el: '#counter-event-example',
  data: {
    total: 0
  },
  methods: {
    incrementTotal: function () {
      this.total += 1
    }
  }
})
```

$emit(...). 子组件触发事件.

Vue 的事件系统分离自浏览器的 EventTarget API。尽管它们的运行类似，但是 $on 和 $emit 不是addEventListener 和 dispatchEvent 的别名。

不能用 $on 侦听子组件释放的事件，而必须在模板里直接用 v-on 绑定，就像以下的例子：

.sync 修饰符

从 2.3.0 起我们重新引入了 .sync 修饰符，但是这次它只是作为一个编译时的语法糖存在。它会被扩展为一个自动更新父组件属性的 v-on 侦听器。

```
<comp :foo.sync="bar"></comp>
会被扩展为：
<comp :foo="bar" @update:foo="val => bar = val"></
comp>
当子组件需要更新 foo 的值时，它需要显式地触发一个更新事件：
this.$emit('update:foo', newValue)
```

定制组件的 v-model

默认情况下，一个组件的 v-model 会使用 value 属性和 input 事件，但是诸如单选框、复选框之类的输入类型可能把 value 属性用作了别的目的。model 选项可以回避这样的冲突：

```
Vue.component('my-checkbox', {
  model: {
    prop: 'checked',
    event: 'change'
  },
  props: {
    checked: Boolean,
    // this allows using the `value` prop for a different purpose
    value: String
  },
  // ...
})
<my-checkbox v-model="foo" value="some value"></my-checkbox>
```

上述代码等价于：

```
<my-checkbox
  :checked="foo"
  @change="val => { foo = val }"
  value="some value">
</my-checkbox>
```

没怎么看明白这块的意思。

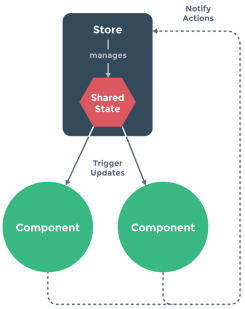
非子组件的通信

有时候两个组件也需要通信 (非父子关系)。在简单的场景下，可以使用一个空的 Vue 实例作为中央事件总线：

```
var bus = new Vue()
// 触发组件 A 中的事件
bus.$emit('id-selected', 1)
// 在组件 B 创建的钩子中监听事件
bus.$on('id-selected', function (id) {
  // ...
})
```

bus.

=> 状态管理模式 Vuex



Slot:

```
<div>
  <h2>我是子组件的标题</h2>
  <slot>
    只有在没有要分发的内容时才会显示。
  </slot>
</div>
```

<slot> </slot>

父组件模版：

```
<div>
  <h1>我是父组件的标题</h1>
  <my-component>
    <p>这是一些初始内容</p>
    <p>这是更多的初始内容</p>
  </my-component>
</div>
```

渲染结果：

```
<div>
  <h1>我是父组件的标题</h1>
  <div>
    <h2>我是子组件的标题</h2>
    <p>这是一些初始内容</p>
    <p>这是更多的初始内容</p>
  </div>
</div>
```

带有 name 的 slot

```
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

父组件模版：

```
<app-layout>
  <h1 slot="header">这里可能是一个页面标题</h1>
  <p>主要内容的一个段落。</p>
  <p>另一个主要段落。</p>
  <p slot="footer">这里有一些联系信息</p>
</app-layout>
```

渲染结果为：

```
<div class="container">
  <header>
    <h1>这里可能是一个页面标题</h1>
  </header>
  <main>
    <p>主要内容的一个段落。</p>
    <p>另一个主要段落。</p>
  </main>
  <footer>
    <p>这里有一些联系信息</p>
  </footer>
</div>
```

作用域 Slot.

作用域插槽是一种特殊类型的插槽，用作使用一个 (能够传递数据到) 可重用模板替换已渲染元素。

```
div class="child">
  <slot text="hello from child"></slot>
</div>
```

在父级中，具有特殊属性 scope 的 <template> 元素必须存在，表示它是作用域插槽的模板。scope 的值对应一个临时变量名，此变量接收从子组件中传递的 props 对象：

```
<div class="parent">
  <child>
    <template scope="props">
      <span>hello from parent</span>
      <span>{{ props.text }}</span>
    </template>
  </child>
</div>
```

⇒

```
<div class="parent">
  <div class="child">
    <span>hello from parent</span>
    <span>hello from child</span>
  </div>
</div>
```

这什么有点设计服务.

```
<my-awesome-list :items="items">
  <!-- 作用域插槽也可以是具名的 -->
  <template slot="item" scope="props">
    <li class="my-fancy-item">{{ props.text }}</li>
  </template>
</my-awesome-list>
```

列表组件的模板：

```
<ul>
  <slot name="item"
    v-for="item in items"
    :text="item.text">
    <!-- 这里写入备用内容 -->
  </slot>
</ul>
```

动态组件.

```
var vm = new Vue({
  el: '#example',
  data: {
    currentView: 'home'
  },
  components: {
    home: { /* ... */ },
    posts: { /* ... */ },
    archive: { /* ... */ }
  }
})
<component v-bind:is="currentView">
  <!-- 组件在 vm.currentview 变化时改变！ -->
</component>
```

v-bind-is

keep-alive.

如果把切换出去的组件保留在内存中，可以保留它的状态或避免重新渲染。为此可以添加一个 keep-alive 指令参数：

```
<keep-alive>
  <component :is="currentView">
    <!-- 非活动组件将被缓存！ -->
  </component>
</keep-alive>
```

Vue 组件的 API 来自三部分 - props, events 和 slots：

Props 允许外部环境传递数据给组件

Events 允许从外部环境在组件内触发副作用

Slots 允许外部环境将额外的内容组合在组件中。

```
<my-component
  :foo="baz"
  :bar="qux"
  @event-a="doThis"
  @event-b="doThat"
>
  <img slot="icon" src="...">
  <p slot="main-text">Hello!</p>
</my-component>
```

子组件索引

```
<div id="parent">
  <user-profile ref="profile"></user-profile>
</div>
var parent = new Vue({ el: '#parent' })
// 访问子组件
var child = parent.$refs.profile
```

异步组件：

在大型应用中，我们可能需要将应用拆分为多个小模块，按需从服务器下载。为了让事情更简单，Vue.js 允许将组件定义为一个工厂函数，动态地解析组件的定义。Vue.js 只在组件需要渲染时触发工厂函数，并且把结果缓存起来，用于后面的再次渲染。例如：

工厂函数：function (resolve, reject)

```
Vue.component('async-example', function (resolve, reject) {
  setTimeout(function () {
    // Pass the component definition to the resolve callback
    resolve({
      template: '<div>I am async!</div>'
    })
  }, 1000)
})
```

结合 Webpack 的 code splitting 功能一起使用：

```
Webpack 的代码分割功能：
Vue.component('async-webpack-example', function (resolve) {
  // 这个特殊的 require 语法告诉 webpack
  // 自动将编译后的代码分割成不同的块，
  // 这些块将通过 Ajax 请求自动下载。
  require(['./my-async-component'], resolve)
})
```

Webpack2 + ES2015：promise.

```
Vue.component(
  'async-webpack-example',
  () => import('./my-async-component')
)
当使用局部注册时，你也可以直接提供一个返回 Promise 的函数：
new Vue({
  // ...
  components: {
    'my-component': () => import('./my-async-component')
  }
})
```

高级异步组件：

```
const AsyncComp = () => ({
  // 需要加载的组件. 应当是一个 Promise
  component: import('./MyComp.vue'),
  // loading 时应当渲染的组件
  loading: LoadingComp,
  // 出错时渲染的组件
  error: ErrorComp,
  // 渲染 loading 组件前的等待时间。默认：200ms.
  delay: 200,
  // 最长等待时间。超出此时间则渲染 error 组件。默认：Infinity
  timeout: 3000
})
```

注意，当一个异步组件被作为 vue-router 的路由组件使用时，这些高级选项都是无效的，因为在路由切换前就会提前加载所需要的异步组件。另外，如果你要在路由组件中使用上述写法，需要使用 vue-router 2.4.0+。

组件间循环引用

在我们的例子中，我们选择在tree-folder 组件中来告诉模块化管理系统循环引用的组件间的处理优先级，
我们知道引起矛盾的子组件是tree-folder-contents，所以我们在beforeCreate 生命周期钩子中去注册它：

```
beforeCreate: function () {
  this.$options.components.TreeFolderContents = require('./tree-folder-contents.vue').default
}
```

单文件组件.

<template> ... </template> 模板

<script> ... </script> 脚本

<style> ... </style> CSS./Less

生产环境部署: Vue-cli Vue的脚手架工具

Process.env.NODE_ENV. 环境变量. Production.

```
var webpack = require('webpack')
module.exports = {
  // ...
  plugins: [
    // ...
    new webpack.DefinePlugin({
      'process.env': {
        NODE_ENV: '"production"'
      }
    }),
    new webpack.optimize.UglifyJsPlugin({
      compress: {
        warnings: false
      }
    })
  ]
}
```

DefinePlugin 定义全局的变量,在webpack中可见.

JS代码压缩

CSS提取到同一个文件内

Vue路由: Vue-Router

```
<div id="app">
  <h1>Hello App!</h1>
  <p>
    <!-- 使用 router-link 组件来导航. -->
    <!-- 通过传入 `to` 属性指定链接. -->
    <!-- <router-link> 默认会被渲染成一个 `<a>` 标签 -->
    <router-link to="/foo">Go to Foo</router-link>
    <router-link to="/bar">Go to Bar</router-link>
  </p>
  <!-- 路由出口 -->
  <!-- 路由匹配到的组件将渲染在这里 -->
  <router-view></router-view>
</div>
```

1. 编写路由 map 映射. Path ⇒ Component.

```
// 0. 如果使用模块化机制编程，导入Vue和VueRouter，要调用
Vue.use(VueRouter)

// 1. 定义（路由）组件。
// 可以从其他文件 import 进来
const Foo = { template: '<div>foo</div>' }
const Bar = { template: '<div>bar</div>' }

// 2. 定义路由
// 每个路由应该映射一个组件。 其中"component" 可以是
// 通过 Vue.extend() 创建的组件构造器，
// 或者，只是一个组件配置对象。
// 我们晚点再讨论嵌套路由。
const routes = [
  { path: '/foo', component: Foo },
  { path: '/bar', component: Bar }
]

// 3. 创建 router 实例，然后传 `routes` 配置
// 你还可以传别的配置参数, 不过先这么简单着吧。
const router = new VueRouter({
  routes // （缩写）相当于 routes: routes
})

// 4. 创建和挂载根实例。
// 记得要通过 router 配置参数注入路由，
// 从而让整个应用都有路由功能
const app = new Vue({
  router
}).$mount('#app')
```

组件定义

主要部分. Path ⇒ Component 的映射.

将 Router mount 到 App 根实例上.

Vue路由：动态路由.

```
const User = {
  template: '<div>User</div>'
}

const router = new VueRouter({
  routes: [
    // 动态路径参数 以冒号开头
    { path: '/user/:id', component: User }
  ]
})
```

⇒

```
const User = {
  template: '<div>User {{ $route.params.id }}</div>'
}
```

/user/:id → $route.params.id

```
/user/:username   /user/evan     { username: 'evan' }
/user/:username/post/:post_id  /user/evan/post/123   { username: 'evan', post_id: 123 }
```

```
const User = {
  template: '...',
  watch: {
    '$route' (to, from) {
      // 对路由变化作出响应...
    }
  }
}
```

watch '$Route'的变化.

ng里面的Watch.

路由的正则匹配.

<router-link ... >

<route-view>... </router-view>

router.push ,

router.replace

```
import Vue from 'vue'
import VueRouter from 'vue-router'

Vue.use(VueRouter)

// The matching uses path-to-regexp, which is the matching engine used
// by express as well, so the same matching rules apply.
// For detailed rules, see https://github.com/pillarjs/path-to-regexp
const router = new VueRouter({
  mode: 'history',
  base: __dirname,
  routes: [
    { path: '/' },
    // params are denoted with a colon ":"
    { path: '/params/:foo/:bar' },
    // a param can be made optional by adding "?"
    { path: '/optional-params/:foo?' },
    // a param can be followed by a regex pattern in parens
    // this route will only be matched if :id is all numbers
    { path: '/params-with-regex/:id(\\d+)' },
    // asterisk can match anything
    { path: '/asterisk/*' },
    // make part of th path optional by wrapping with parens and add "?"
    { path: '/optional-group/(foo/)?bar' }
  ]
})
```

支持浏览器的前进后退模式.

base是做什么用的?

可选的

正则匹配.

* 匹配所有的

(foo/)这部分是可选的

```
new Vue({
  router,
  template: `
    <div id="app">
      <h1>Route Matching</h1>
      <ul>
        <li><router-link to="/">/</router-link></li>
        <li><router-link to="/params/foo/bar">/params/foo/bar</router-link></li>
        <li><router-link to="/optional-params">/optional-params</router-link></li>
        <li><router-link to="/optional-params/foo">/optional-params/foo</router-link></li>
        <li><router-link to="/params-with-regex/123">/params-with-regex/123</router-link></li>
        <li><router-link to="/params-with-regex/abc">/params-with-regex/abc</router-link></li>
        <li><router-link to="/asterisk/foo">/asterisk/foo</router-link></li>
        <li><router-link to="/asterisk/foo/bar">/asterisk/foo/bar</router-link></li>
        <li><router-link to="/optional-group/bar">/optional-group/bar</router-link></li>
        <li><router-link to="/optional-group/foo/bar">/optional-group/foo/bar</router-link></li>
      </ul>
      <p>Route context</p>
      <pre>{{ JSON.stringify($route, null, 2) }}</pre>
    </div>
  `
}).$mount('#app')
```

Vue Route：nested Router 嵌套路由。

{path：xxx，component：xxx，children：...}
嵌套的子路由。

```
const User = {
  template: `
    <div class="user">
      <h2>User {{ $route.params.id }}</h2>
      <router-view></router-view>
    </div>
  `
}

const router = new VueRouter({
  routes: [
    { path: '/user/:id', component: User,
      children: [
        {
          // 当 /user/:id/profile 匹配成功，
          // UserProfile 会被渲染在 User 的 <router-view> 中
          path: 'profile',
          component: UserProfile
        },
        {
          // 当 /user/:id/posts 匹配成功
          // UserPosts 会被渲染在 User 的 <router-view> 中
          path: 'posts',
          component: UserPosts
        }
      ]
    }
  ]
})
```

```
const router = new VueRouter({
  routes: [
    {
      path: '/user/:id', component: User,
      children: [
        // 当 /user/:id 匹配成功，
        // UserHome 会被渲染在 User 的 <router-view> 中
        { path: '', component: UserHome },

        // ...其他子路由
      ]
    }
  ]
})
```

默认的子路由。

声明式

<route-link :to="..."/>

window.history

RouterPush

Router.replace

Router.go

Router.push

```
router.push('home')

// 对象
router.push({ path: 'home' })

// 命名的路由
router.push({ name: 'user', params: { userId: 123 }})

// 带查询参数，变成 /register?plan=private
router.push({ path: 'register', query: { plan: 'private' }})
```

命名Route

有时候，通过一个名称来标识一个路由显得更方便一些，特别是在链接一个路由，或者是执行一些跳转的时候。你可以在创建 Router 实例的时候，在 routes 配置中给某个路由设置名称。

```
const router = new VueRouter({
  routes: [
    {
      path: '/user/:userId',
      name: 'user',
      component: User
    }
  ]
})
```

name：user

要链接到一个命名路由，可以给 router-link 的 to 属性传一个对象：

```
<router-link :to="{ name: 'user', params: { userId: 123 }}">User</router-link>
```

这样更清晰。

这跟代码调用 router.push() 是一回事：

```
router.push({ name: 'user', params: { userId: 123 }})
```

这两种方式都会把路由导航到 /user/123 路径。

带命名的 view。

有时候想同时（同级）展示多个视图，而不是嵌套展示，例如创建一个布局，有 sidebar（侧导航） 和 main（主内容） 两个视图，这个时候命名视图就派上用场了。你可以在界面中拥有多个单独命名的视图，而不是只有一个单独的出口。如果 router-view 没有设置名字，那么默认为 default。

```
<router-view class="view one"></router-view>
<router-view class="view two" name="a"></router-view>
<router-view class="view three" name="b"></router-view>
```

一个视图使用一个组件渲染，因此对于同个路由，多个视图就需要多个组件。确保正确使用 components 配置（带上 s）：

```
const router = new VueRouter({
  routes: [
    {
      path: '/',
      components: {
        default: Foo,
        a: Bar,
        b: Baz
      }
    }
  ]
})
```

多个组件。

router-view 会根据这个 Path 来匹配对应的 Component。

Route 的重定向(redirect)和别名 alias.

```
const router = new VueRouter({
  routes: [
    { path: '/a', redirect: '/b' }
  ]
})
```

redirect.

重定向的目标也可以是一个命名的路由：

```
const router = new VueRouter({
  routes: [
    { path: '/a', redirect: { name: 'foo' }}
  ]
})
```

name

甚至是一个方法，动态返回重定向目标：

```
const router = new VueRouter({
  routes: [
    { path: '/a', redirect: to => {
      // 方法接收 目标路由 作为参数
      // return 重定向的 字符串路径/路径对
象
    }}
  ]
})
```

函数.

重定向』的意思是，当用户访问 /a时，URL 将会被替换成 /b，然后匹配路由为 /b，那么『别名』又是什么呢?

/a 的别名是 /b，意味着，当用户访问 /b 时，URL 会保持为 /b，但是路由匹配则为 /a，就像用户访问 /a 一样。

上面对应的路由配置为：

```
const router = new VueRouter({
  routes: [
    { path: '/a', component: A, alias: '/b' }
  ]
})
```

类似于 redirect，只是保持 url 不变.

alias.

『别名』的功能让你可以自由地将 UI 结构映射到任意的 URL，而不是受限于配置的嵌套路由结构。

history mode：让 api 变得干净

```
const router = new VueRouter({
  mode: 'history',
  routes: [...]
})
```

当你使用 history 模式时，URL 就像正常的 url，例如 http://yoursite.com/user/id，也好看！

不过这种模式要玩好，还需要后台配置支持。因为我们的应用是个单页客户端应用，如果后台没有正确的配置，当用户在浏览器直接访问 http://oursite.com/user/id 就会返回 404，这就不好看了。

所以呢，你要在服务端增加一个覆盖所有情况的候选资源：如果 URL 匹配不到任何静态资源，则应该返回同一个 index.html 页面，这个页面就是你 app 依赖的页面。

```
location / {
  try_files $uri $uri/ /index.html;
}
```

返回同一个 index.html 页面

前端的404.

```
const router = new VueRouter({
  mode: 'history',
  routes: [
    { path: '*', component: NotFoundComponent }
  ]
})
```

路由元信息

```
const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      component: Foo,
      children: [
        {
          path: 'bar',
          component: Bar,
          // a meta field
          meta: { requiresAuth: true }
        }
      ]
    }
  ]
})
```

这些元信息，可以在 hook 中使用。

hook（钩子）

遍历 matched。

```
router.beforeEach((to, from, next) => {
  if (to.matched.some(record => record.meta.requiresAuth)) {
    // this route requires auth, check if logged in
    // if not, redirect to login page.
    if (!auth.loggedIn()) {
      next({
        path: '/login',
        query: { redirect: to.fullPath }
      })
    } else {
      next()
    }
  } else {
    next() // 确保一定要调用 next()
  }
})
```

我们需要遍历 $route.matched 来检查路由记录中的 meta 字段。

首先，我们称呼 routes 配置中的每个路由对象为 路由记录。路由记录可以是嵌套的，因此，当一个路由匹配成功后，他可能匹配多个路由记录

```
<template>
  <div class="post">
    <div class="loading" v-if="loading">
      Loading...
    </div>

    <div v-if="error" class="error">
      {{ error }}
    </div>

    <div v-if="post" class="content">
      <h2>{{ post.title }}</h2>
      <p>{{ post.body }}</p>
    </div>
  </div>
</template>
export default {
  data () {
    return {
      loading: false,
      post: null,
      error: null
    }
  },
  created () {
    // 组件创建完后获取数据，
    // 此时 data 已经被 observed 了
    this.fetchData()
  },
  watch: {
    // 如果路由有变化，会再次执行该方法
    '$route': 'fetchData'
  },
  methods: {
    fetchData () {
      this.error = this.post = null
      this.loading = true
      // replace getPost with your data fetching util / API wrapper
      getPost(this.$route.params.id, (err, post) => {
        this.loading = false
        if (err) {
          this.error = err.toString()
        } else {
          this.post = post
        }
      })
    }
  }
}
```

template scope =?

这个scope指的是什么？

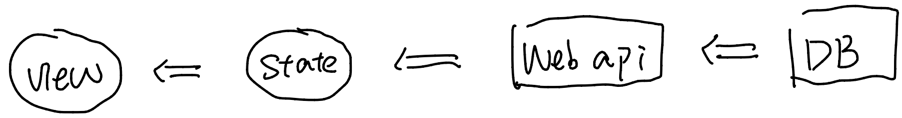
data 状态

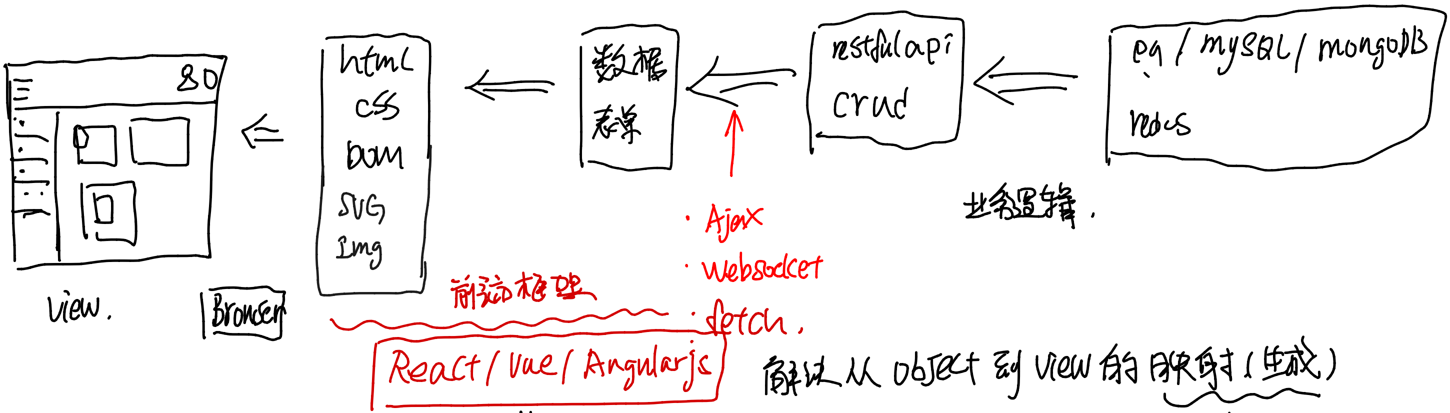
"fetchData" 在methods中对应的函数。

在fetchData.

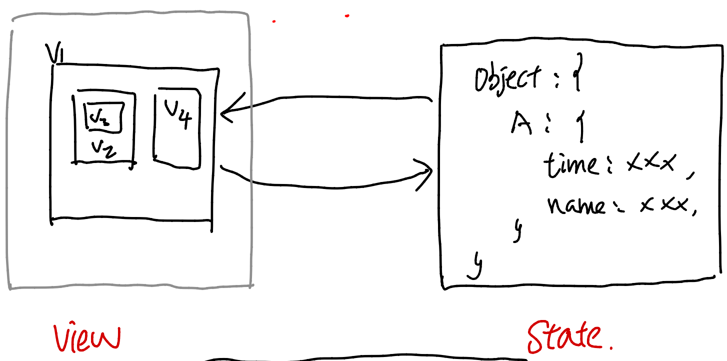
前端务系统：React / Vue / Redux / vuex.

1. Web 数据流





解决从 object 到 view 的映射(生成)
Render.



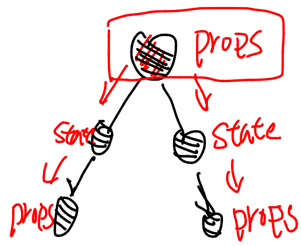
Redux / vuex

React：单向数据流

Agularjs / vue：双向数据流

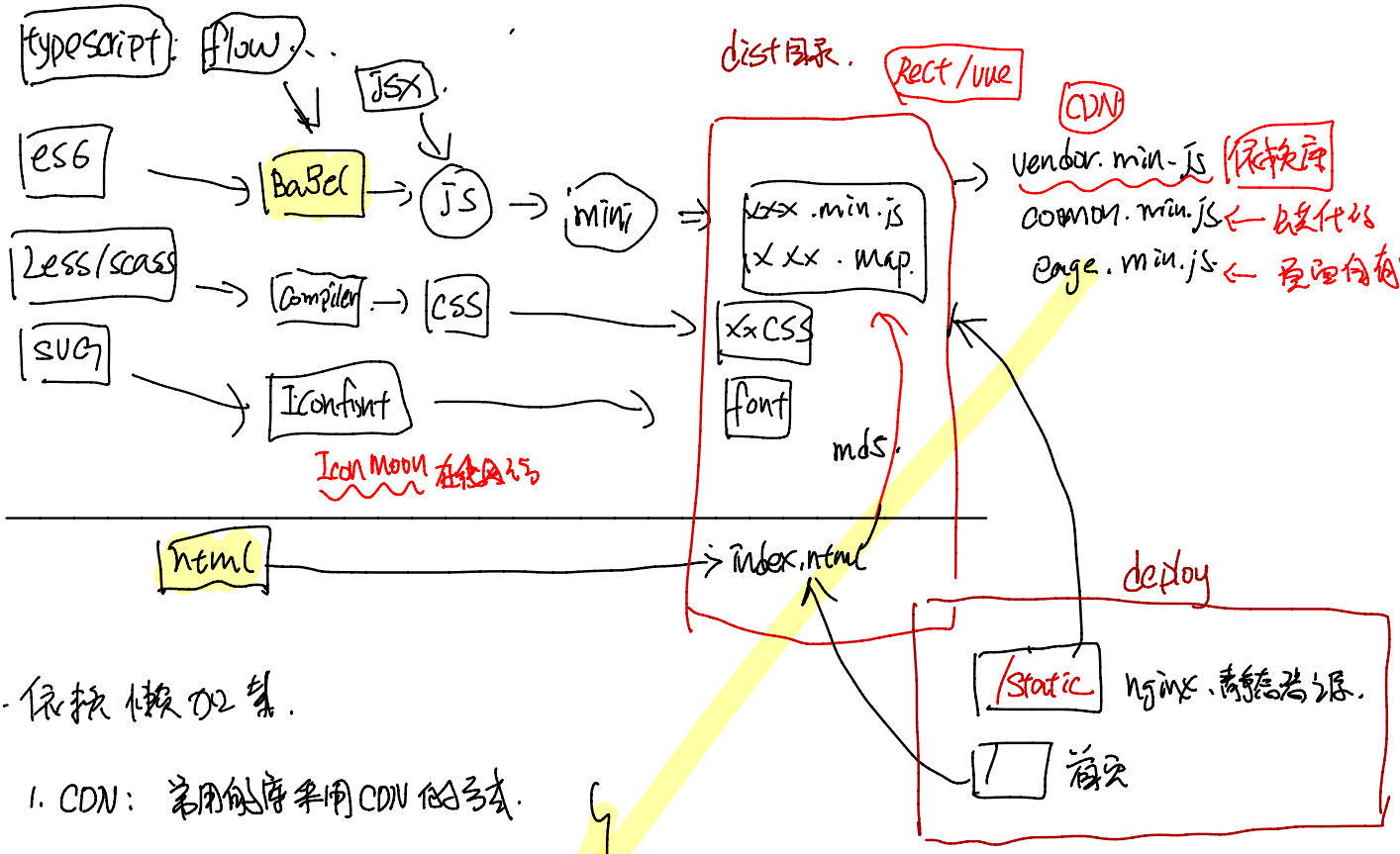
<my component
props = {... } >
<childComponent
props = {... } />
</my Component>

1. template JSX. Render : Data => View => html.
2. 父子组件的通信 props => state. 组件化
3. 组件之间的通信 Redux / vuex
4. 事件绑定 emit onclick...
5. 路由. #? hash params= - - -
   hash 锚. 跳转. view 切换.



props 和 state 之间的 bind. ...

2. 打包. make 工具:



3. 依赖模块加载.

1. CDN: 常用的库采用CDN的方式.
2. common.: site的公共的代码块
3. page.: 每个独有的逻辑
4. chunk加载. 懒加载.
5. 缓存删除问题: md5

3. 模块化.

1. es6的 module import.

2. node的 require方式 require / export

require(xxx.css)

4. 模板：

Object ⇒ template ⇒ html

<elm attr=xxx class=xx />

1. basic：

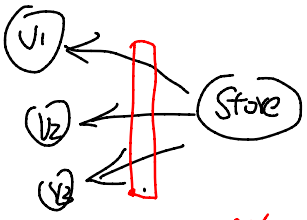
{{ message }}

{{ a.b }}

2. for/if else　　v-for　key　item. index.

3. 数据复用.

Redux . Vuex: 订阅者模式，eventBus . . .



User input

click
onChange

Async

server response

start ajax request

Action create

action ⇒ type params

store dispatch ⇐ OldState

middleware

常用的中间件

redux-trunk

redux-promise

redux-saga

```
let store = createStore(
  todoApp,
  // applyMiddleware() tells createStore() how to handle
middleware
  applyMiddleware(logger, crashReporter)
)
```

middleware

reducer根据action的type
做相应的处理

combiledReducer ⇒

newState[key1] = Reducer1(oldState[key1], action)

newState[key2] = Reducer2(oldState[key2], action)

这个地方场和immutable.js配合，immutable.js实现了copy on write所以效率很高。

new state

更新UI

React-Redux

connect

ReactView

provider