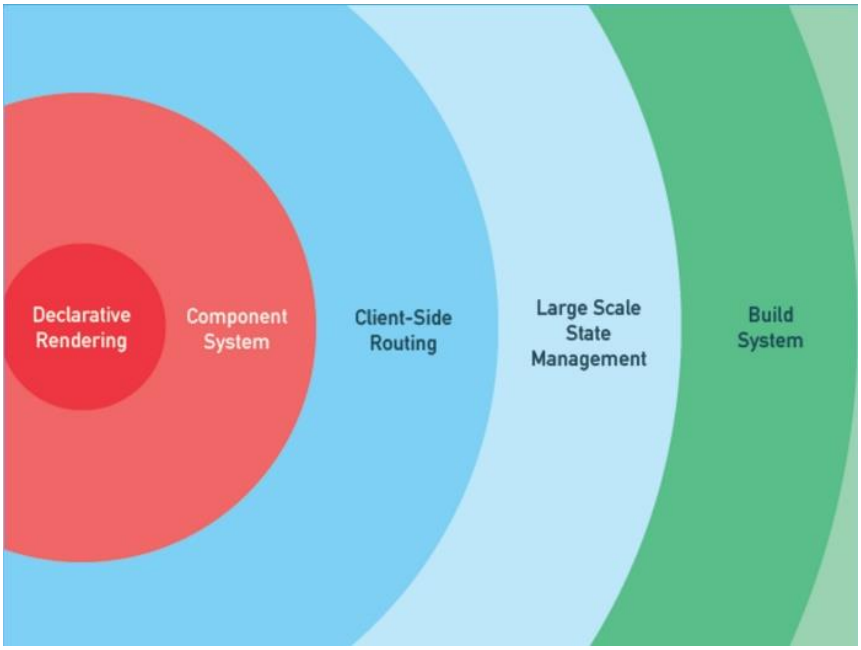
vue

1、开发需装的插件

Vetur

Vue VSCode Snippets

2、渐进式：需要用到哪部分则引入哪部分



3、安装

CDN 引入、NPM 安装、CLI 安装

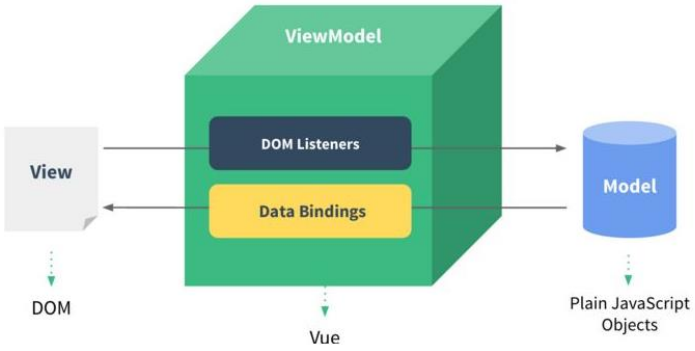
4、系统：

路由：vue-router

大规模状态管理：vuex

构建系统：vue-cli、vue-test-utils

5、mvvm



6、基础 api

- 文本插值：{{ }} `{{msg + "123" + "234"}}`
- 通过 el 确定根容器 `el: "#app",`
- v- 指令 简写是冒号，即：`:test=`
- 可以写表达式 `"myTest + 'vue---'"` `{{msg + "123" + "234"}}`

```
<div id="app" :test="myTest + 'vue---'">{{msg + "123" + "234"}}</div>
<script>
  // 文本插值 → {{}}
  // v-bind 指令 → 简写 → :
  //

  // console.log(Vue);
  // hello world
  // vue
  // mvvm
  // m → model
  // v → view
  // vm → viewModel
  // 1. 视图 view
  // 2. model → js 逻辑
  // 3. 视图和model 绑定
  //    视图和 model 层 之间的接口
  // 开发思想
  // 改变我们的数据逻辑
  app = new Vue({
    el: "#app",
    data: {
      msg: "hello world",
      myTest: "myTest111",
    },
  });
```

- *v-on* 添加事件 简写是@

```
  <button @click="handleClick">click</button>
</div>
<script>
  // v-on 添加事件
  // 简写 @
  app = new Vue({
    el: "#app",
    data: {},
    methods: {
      handleClick(e) {
        console.log("click");
        console.log(e);
      },
    },
  });
```

- *Inline Statement* `count: 0,` `count++`

```
  <button @click="count++">click</button>
</div>
<script>
  // v-on 添加事件
  // 简写 @
  app = new Vue({
    el: "#app",
    data: {
      count: 0,
    },
    methods: {
      handleClick(type, e) {
        console.log("click");
        console.log(e);
      },
    },
  });
```

- 修饰符

.once：只触发一次回调

```
<button @click.once="count++">click</button>
```

.prevent：可用于防止提交表单时页面的自动刷新

```
  <form action="">
    <button @click.prevent="handleSubmit">submit</button>
  </form>
</div>
<script>
  // v-on 添加事件
  // 简写 @
  app = new Vue({
    el: "#app",
    data: {
      count: 0,
    },
    methods: {
      handleSubmit(e) {

        console.log("submit");
      },
      handleClick(type, e) {
        console.log("click");
        console.log(e);
      },
    },
  });
```

- *computed*：计算属性

```
<!-- <div id="app">{{msg.split("").reverse().join("")}}</div> -->
<div id="app">{{reverseMsg}}</div>
<script>
  // 1. 代码的可读性
  //    - 维护人?
  //    - 可扩展性 也会很好
  //    - 加班 效率
  //       5  2 小时
  // 计算属性
  app = new Vue({
    el: "#app",
    data: {
      msg: "hello world",
    },
    computed: {
      reverseMsg() {
        return this.msg.split("").reverse().join("");
      },
    },
  });
```

- *methods 和 computed 的区别：computed* 会有缓存的效果，即其依赖的属性（图中为 *hello world*）如果没有变更，则其只会执行一次，因此若 *msg* 不变则多次执行也不会打印出多个 *reverseMsg*`console.log("reverseMsg")`；而 *methods* 则无此效果。

  总结：无论 *Msg* 是否变化，*methods* 每次都执行；若 *Msg* 不变，*computed* 则只执行一次，后续再调用并不执行，若 *Msg* 变化，*computed* 则会执行。

```
//    计算属性是依赖别的属性的属性 如果依赖的属性没有更新的话，那么当前属性就使用之前的值
app = new Vue({
  el: "#app",
  data: {
    msg: "hello world",
  },
  methods: {
    reverseMessage() {
      console.log("reverseMessage");
      return this.msg.split("").reverse().join("");
    },
  },
  computed: {
    reverseMsg() {
      // 怎么实现的
      // 低层次代码
      console.log("reverseMsg");
      return this.msg.split("").reverse().join("");
    },
  },
});
```

```
<div id="app">{{reverseMsg}}


  <br>

  {{reverseMessage()}}

</div>
```

批注 [M帐1]: 为何调用 *methods* 时需加()，调用 *computed* 则不用？

- 多对一：computed 比较适合对多个变量或者对象进行处理后返回一个结果值，也就是多个变量中的某一个值发生了变化则我们监控的这个值也就会发生变化，举例：购物车里面的商品列表和总金额之间的关系，只要商品列表里面的商品数量发生变化，或减少或增多或删除商品，总金额都应该发生变化。这里的这个总金额使用 computed 属性来进行计算是最好的选择。即一依赖于多。

- watch：监测变化，执行异步请求或开销较大的操作时使用。
  - 一对多：即多依赖于一。如下图，msg1、msg2、msg3 都依赖于 count，当 count 变化时，watch 会触发，使得 msg1、msg2、msg3 也都变化。

```
app = new Vue({
  el: "#app",
  data: {
    count: 0,
    msg: "hello",
    msg1: "",
    msg2: "",
    msg3: "",
  },
  watch: {
    count(newValue, oldValue) {
      console.log(newValue, oldValue);
      // 异步请求
      // 副作用
      // fetch(`http://baidu.com/q=${this.count}`)
      this.msg1 = newValue + "1";
      this.msg2 = newValue + "2";
      this.msg3 = newValue + "3";
    },
```

watch 观察 count 变量，
newValue 为其新值，
oldValue 为其旧值。

  - 若 count 值不变（赋给 count 的 newValue=oldValue 的情况下），则 watch 中的 count 也不会执行，因此 msg1、msg2、 msg3 也不会显示 or 改变。
  - immediate：默认会先执行一次 watch（watch 默认不会先执行）

```
watch: {
  count: {
    handler(newValue,oldValue) {
      console.log(newValue, oldValue);
      // 异步请求
      // 副作用
      // fetch(`http://baidu.com/q=${this.count}`)
      this.msg1 = newValue + "1";
      this.msg2 = newValue + "2";
      this.msg3 = newValue + "3";
    },
    immediat: true,
  },
```

- *deep*：深度 *watch*，可以监测对象内部的属性等的变化（没用 *deep* 时不可以取到对象内部的值）

```
app = new Vue({
  el: "#app",
  data: {
    count: 0,
    msg: "hello",
    msg1: "",
    msg2: "",
    msg3: "",
    user: {
      name: "海哥",
    },
  },
  watch: {
    user: {
      deep: true,
      handler(newValue) {
        console.log(newValue);
      },
    },
```

- 条件渲染
  - *v-if 和 v-else* 隐藏模式下并不渲染（以下图代码为例，由于 *age=1*，因此“花一样的年龄”和“成年人”所在的两个 *div* 均不渲染，在浏览器查看代码会看到这两个 *div* 应在的地方是两行注释，即<!---->），懒加载

```
<div id="app">
  <div v-if="age === 18">花一样的年龄</div>
  <div v-if="age > 18">成年人</div>
  <div v-else>未成年人</div>
</div>
<script>
  // js
  // if else if else
  app = new Vue({
    el: "#app",
    data: {
      age: 1,
    },
  });
```

  - *v-show* 显示与隐藏，隐藏模式下仍然渲染（即在浏览器查看代码会看到有对应 * *），只是设置 *display* 为 *none*

条件渲染

- v-if
  - v-else-if
  - v-else
- v-show
  - 控制 display
- v-if vs v-show
  - v-if 是"真正"的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。
  - v-if也是**惰性的**：如果在初始渲染时条件为假，则什么也不做—直到条件第一次变为真时，才会开始渲染条件块。
  - v-show就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 进行切换。
  - 一般来说，v-if 有更高的切换开销，而 v-show 有更高的初始渲染开销。因此，如果需要非常频繁地切换，则使用 v-show 较好；如果在运行时条件很少改变，则使用 v-if 较好。

- 列表渲染
  - *v-for* 可以循环列表

```
  <ul>
    <li v-for="(item,index) in list">
      {{index}} -- {{item.name}} -- {{item.age}}
    </li>
  </ul>
</div>
<script>
  app = new Vue({
    el: "#app",
    data: {
      list: [
        {
          name: "海哥",
          age: 30,
        },

        {
          name: "钟老师",
          age: 35,
        },
      ],
    },
  });
```

循环数组和循环对象的不同点：

```
<ul>
  <li v-for="(item,index) in list">
    {{index}} -- {{item.name}} -- {{item.age}}
  </li>
</ul>

<ul>
  <li v-for="(val,key,index) in users">
    {{index}} - {{val.name}} - {{val.age}} - {{key}}
  </li>
</ul>
```

数组

对象

```
app = new Vue({
  el: "#app",
  data: {
    list: [                    数组
      {
        name: "海哥",
        age: 30,
      },

      {
        name: "钟老师",
        age: 35
      },
    ],
    users: {                   对象
      1: {
        name: "海哥",
        age: 30,
      },

      2: {
        name: "钟老师哥",
        age: 34,
      },
    },
```

- 0 -- 海哥 -- 30
- 1 -- 钟老师 -- 35

- 0 – 海哥 – 30 – 1
- 1 – 钟老师哥 – 34 – 2

结果：

- *key*：优化 *diff* 查找算法

批注 [M帐2]:

```
<li v-for="(item,index) in list" :key="index">
  {{index}} -- {{item.name}} -- {{item.age}}
```

- *v-for* 和 *v-if* 可联合起来使用：用 *v-for* 遍历数组或对象，并用 *v-if* 筛选出符合条件的 *item*

- *class*、*style*：支持对象形式和数组形式（数组里嵌套对象、使用三元表达式也行）

```
<style>
  .red {
    color: red;
  }

  .size {
    font-size: 50px;
  }
</style>
</head>

<body>
<div id="app">
  <div :class="classes">color</div>
</div>
<script>
  app = new Vue({
    el: "#app",
    data: {
      classes: {
        red: true,
        size: true,
      },
    },
  });
```

对象

```
classes:[
  "red",
  "size"
]
```

数组

三元表达式

```
classes: [false ? "red" : "", "size"],
```

- 表单输入绑定 v-model：语法糖，与添加监听事件类似

  - 修饰符：.number：将其他类型（如 string 类型）转换为 number 类型

```
<input type="text" v-model.number="value" />
```

    如 input 中输入“123”，则将 value 打印出来后可看到，不加修饰符时其类型为 string，加了修饰符.number 后则其类型为 number。

7、组件：提高复用性

- 注册组件：

  全局 Vue.component

  局部 let app = new Vue(components:{组件名称:{组件配置}}); 此处注册的组件只能在这个 app 内部使用

  - 每一个组件的 template 有且仅有一个根节点：

```
Vue.component('kkb-user', {
    // 每一个组件的 template 有且仅有一个根节点
    template: `
        <div></div>
        <dl>
            <dt>zMouse</dt>
            <dd>性别：男</dd>
            <dd>年龄：24</dd>
        </dl>
    `
});
```

  - component 里的 data 必须为函数，且 data 函数必须返回对象。因为若不如此，那么多个组件注册使用时共用一个地址，当改变组件 1 里的 data 时，组件 2 里的 data 也会随之改变。而当 data 为函数且返回对象时，每 new 一次组件，都会执行一次 data 函数，创建一个新的对象，因此组件 1 和组件 2 不会再互相影响。

批注 [M帐3]:

data 必须是一个函数

当我们定义这个组件时，你可能会发现它的 data 并不是像这样直接提供一个对象：

```
data: {
  count: 0
}
```

取而代之的是，一个组件的 data 选项必须是一个函数，因此每个实例可以维护一份被返回对象的独立的拷贝：

```
data: function () {
  return {
    count: 0
  }
}
```

如果 Vue 没有这条规则，点击一个按钮就可能会影响到其它所有组件实例。可以看出，注册组件时传入的配置和创建Vue实例差不多，但也有不同，其中一个就是data属性必须是一个函数。这是因为如果像Vue实例那样，传入一个对象，由于JS中对象类型的变量实际上保存的是对象的引用，所以当存在多个这样的组件时，会共享数据，导致一个组件中数据的改变会引起其他组件数据的改变。而使用一个返回对象的函数，每次使用组件都会创建一个新的对象，这样就不会出现共享数据的问题来了。

```
Vue.component('kkb-user', {
    data() {
        return {
            id: 1,
            name: 'zMouse',
            gender: '男',
            age: 24
        };
    },
    template: `
        <dl>
            <dt>{{name}}</dt>
            <dd>性别：男</dd>
            <dd>年龄：24</dd>
        </dl>
    `
});
```

- *component* 在注册时不是组件对象，而是组件类，类似于 *class*，在使用时才会被 *Vue* 解析并实例化成为对象。

- *props*
  - 当没有 *template* 时，*vue* 会自动将当前组件中的 *el* 对应的元素的 *outerHTML*（包含 *el* 对应的元素本身）作为 *template*

    ```
    template: document.querySelector('#app').outerHTML
    ```

  - *props* 用来定义该组件可以接受外部传入的属性/参数
  - 只能在内容中使用*{{}}*，而不能在 *html* 元素的<></>中使用，即：

    <a>{{data}} 可以

    ** 不行
  - 传入 *users[0]*并不传入 *users* 数组 *index* 为 *0* 的对象，而是字符串“*users[0]*”

```
// 非指令的 attribute 是 html 属性，其值不会做任何特殊处理
template: `
  <div id="app">
      <kkb-user data="users[0]"></kkb-user>
      <hr />
      <kkb-user data="users[1]"></kkb-user>
  </div>
`
```

加上指令 v-bind 才会对其进行解析，传入 users 数组 index 为 0 的对象

```
// 非指令的 attribute 是 html 属性，其值不会做任何特殊处理
// v-bind 指令的值 会首先被 Vue 进行解析
template: `
  <div id="app">
      <kkb-user v-bind:data="users[0]"></kkb-user>
      <hr />
      <kkb-user data="users[1]"></kkb-user>
  </div>
`
```

- 单向数据流：不建议直接在子组件中修改 *props* 传入的数据，因为数据会被破坏，子组件作为外部数据的消费者，只能拥有使用权，而不能有修改权，即需要保证数据的安全性，因为该数据有可能被其它的组件进行使用。数据的修改权限应该由数据的原始持有者决定。若想修改则最好利用自定义事件方式通知父级。下图中使用 *+1* 而不是++的原因是，*+1* 不改变 *data.age* 本身，++则会改变 *data.age* 本身，相当于子组件还是修改了父级中的 *data*，这是前述内容中已阐明不推荐的。

子组件：

```
Vue.component('kkb-user', {
    props: ['data'],
    template: `
        <dl>
            <dt>{{data.name}}</dt>
            <dd>性别：{{data.gender}}</dd>
            <dd>年龄：{{data.age}} <button @click="inc">+</button></dd>
        </dl>
    `,
    methods: {
        inc() {
            // 不建议大家直接修改 props 传入的数据
            // 因为数据会被破坏，组件作为外部数据的消费者，只能拥有使用权，而不
            // 数据的修改权限应该由数据的原始持有人去决定
            // this.data.age++;

            // 利用自定义事件的方式去通知父级
            this.$emit('change', {
                id: this.data.id,
                newAge: this.data.age + 1
            });
        }
    }
});
```

父级：

```
let app = new Vue({
    el: '#app',
    data: {
        users: [
            {
                id: 1,
                name: 'zMouse',
                gender: '男',
                age: 24
            },
            {
                id: 2,
                name: 'xiaorui',
                gender: '男',
                age: 20
            }
        ]
    },
    template: `
      <div id="app">
          <h2>统计：</h2>
          <p> {{users[0].age + users[1].age}} </p>
          <hr />
          <kkb-user :data="users[0]" @change="changeAge"></kkb-user>
          <hr />
          <kkb-user :data="users[1]" @change="changeAge"></kkb-user>
      </div>
    `,
    methods: {
        changeAge(changeData) {
            // console.log(newAge);
            this.users = this.users.map( user => {
                if (user.id === changeData.id) {
                    user.age = changeData.newAge;
                }
                return user;
            } );
        }
    }
});
```

- 双向绑定：v-model，如下两图，但并不推荐，因为并不知道父级的数据在内部改变之后会同步到外部的子组件的哪里。

批注 [M帐4]:

子组件：

```
template: `
      <dl>
          <dt>{{data.name}}</dt>
          <dd>性别：{{data.gender}}</dd>
          <dd>年龄：{{data.age}} <button @click="inc">+</button></dd>
      </dl>
`,
methods: {
    inc() {
        this.$emit('change', {
            ...this.data,
            age: this.data.age + 1
        });
    }
}
```

父组件：

```
template: `
  <div id="app">
      <h2>统计: </h2>
      <p> {{users[0].age + users[1].age}} </p>
      <hr />
      <kkb-user v-model="users[0]"></kkb-user>
      <hr />
      <kkb-user v-model="users[1]"></kkb-user>
  </div>
`
```

.sync 同步，自动更新父组件属性的 v-on 监听器 update:要更新的数据的名称 sync 支持多个 prop 同步，而 v-model 只支持一个

子组件：

```
methods: {
    inc() {
        this.$emit('update:data', {
            ...this.data,
            age: this.data.age + 1
        });
    }
}
```

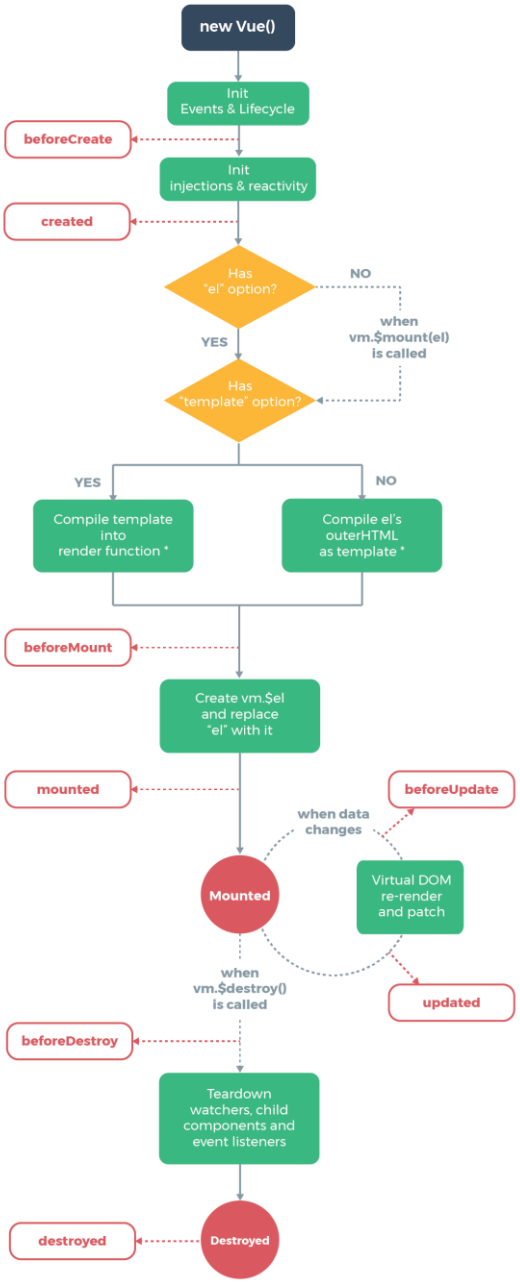
父组件：

```
template: `
  <div id="app">
      <h2>统计: </h2>
      <p> {{users[0].age + users[1].age}} </p>
      <hr />
      <kkb-user :data.sync="users[0]"></kkb-user>
      <hr />
      <kkb-user :data.sync="users[1]"></kkb-user>
  </div>
`
```

- 验证：即类型检查，如限定传入的参数的值为 Number 类型或任何其他类型等

```
Vue.component('my-component', {
  props: {
    // 基础的类型检查 (`null` 和 `undefined` 会通过任何类型验证)
    propA: Number,
    // 多个可能的类型
    propB: [String, Number],
    // 必填的字符串
    propC: {
      type: String,
      required: true
    },
    // 带有默认值的数字
    propD: {
      type: Number,
      default: 100
    },
    // 带有默认值的对象
    propE: {
      type: Object,
      // 对象或数组默认值必须从一个工厂函数获取
      default: function () {
        return { message: 'hello' }
      }
    },
    // 自定义验证函数
    propF: {
      validator: function (value) {
        // 这个值必须匹配下列字符串中的一个
        return ['success', 'warning', 'danger'].indexOf(value) !== -1
      }
    }
  }
})
```

- 组件生命周期
  - 生命周期钩子：组件各个生命周期中对外暴露的接口



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

组件被销毁，但垃圾尚未被回收，因此若在组件被销毁之前操作了 *data*，如 *setInterval()*计时器，那么组件被销毁之后，该计时器仍起作用，*data* 还是会随操作而改变。

- 插槽
  - *slot*：*vue* 内置组件，用来获取组件外部子节点内容