百度San框架教程

书栈(BookStack.CN)

目　录

- 致谢
- README
- 指南
  - 数据
    - 什么东西不要保存在 data 里？
    - 什么东西可以保存在data里？
    - data bind时的auto camel
  - 试图模板
    - 如何遍历一个对象？
    - 数组深层更新如何触发视图更新？
    - 如何实现元素的显示/隐藏？
  - 组件间通信
    - 父组件如何更新子组件？
    - 子组件如何通知父组件？
    - 子组件与更高层组件如何通信？
    - 动态子组件如何传递消息给父组件?
  - 组件管理
    - 我们可以操作 DOM 吗？
  - 路由管理
    - 如何使用 san-router 建立一个单页应用的后台系统？
  - 应用状态管理
    - 如何使用 san-store 实现后台系统的状态管理？
  - FAQ
    - Q&A集锦
  - 如何处理绝对定位组件的 DOM？
- 教程
  - 安装
  - 背景
  - 开始
  - 模板
  - 数据操作
  - 数据校验
  - 样式
  - 条件
  - 循环
  - 事件处理

- 表单
- 插槽
- 过渡
- 组件
- 组件反解（old）
- 组件反解
- 服务端渲染

API

- 主模块API
- 组件API

致谢

当前文档 《百度San框架教程》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-07-05。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址: http://www.bookstack.cn/books/san-zh

书栈官网: http://www.bookstack.cn

书栈开源: https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

README

San Website

prepare

```
$ npm i
```

preview

```
# npm start
hexo s
```

deploy

```
# npm run deploy
$ hexo deploy
```

给 San 文档做贡献

San 是一个传统的 MVVM 组件框架，官网地址 https://baidu.github.io/san/

如果你在使用 San 过程中遇到任何问题，请通过 github: https://github.com/baidu/san 给我们提 issue;
如果你在看 San 文档的过程中发现任何问题，可以通过文档 github: https://github.com/baidu/san-website 给我们提 issue，或者在相应位置修改后发起 PR;
当然我们非常欢迎您在实践 San 框架过程中，有任何实践经验或总结文档，可以通过 PR 的方式提交到文档 github: https://github.com/baidu/san-website， 经过我们 review 后的文档会合入 San 的官方文档中。

San 文档 PR 规范

1. 提交前的 **fork** 同步更新操作：每次 PR 前请进行 fork 同步更新操作，避免产生冲突。

2. 文档内容： 必须包含对实践问题的原理分析总结，包含实际的 demo，demo 的编写请使用 codepen，最后将其嵌入到文档中，具体详情及嵌入方式请见例子：https://baidu.github.io/san/practice/traverse-object/.

3. 实践类文档项目路径：

    - [ ] 添加文档可以往这里发 PR

    ```
    1.  https://github.com/baidu/san-website/tree/master/source/_posts/practice
    ```

    - [ ] 链接是手工加

    ```
    1.  https://github.com/baidu/san-website/blob/master/themes/san/layout/practice.ejs
    ```

4. **PR** 标题与内容：PR 标题和内容，请对文档进行详细说明，并提供文档的最终截图。

有任何文档问题可以给我们提 issue

来源(书栈小编注)

指南

- 数据
- 试图模板
- 组件间通信
- 组件管理
- 路由管理
- 应用状态管理
- FAQ
- 如何处理绝对定位组件的 DOM？

数据

- 什么东西不要保存在 data 里？
- 什么东西可以保存在data里？
- data bind时的auto camel

什么东西不要保存在 data 里？

title: 什么东西不要保存在 data 里？
categories:

- practice

我们知道，data 里应该存与视图相关的数据状态。我们在下面列举了一些不当的使用场景，这些场景是我们不止发现过一次的。

函数

不要把函数作为数据存放。函数应该是独立的，或者作为组件方法存在。

```
// bad
this.data.set('camel2kebab', function (source) {
    return source.replace(/[A-Z]/g, function (match) {
        return '-' + match.toLowerCase();
    });
});
```

DOM 对象

这个应该不用解释吧。

```
// bad
this.data.set('sideEl', document.querySelector('sidebar'));
```

组件等复杂对象

不要使用数据来做动态子组件的管理。动态子组件对象可以直接存在组件的成员中。

```
// bad
var layer = new Layer();
layer.attach(document.body);
this.data.set('layer', layer);

// good
var layer = new Layer();
layer.attach(document.body);
```

```
9.  this.layer = layer;
```

什么东西可以保存在data里？

在 San 的文档中写道：” San 是一个 MVVM 的组件框架，通过 San 的视图引擎能够让用户只用操作数据，视图自动更新”。这里要说的 data 指的就是用户操作的”数据”。data 里应该存与视图相关的数据状态。

data 中保存的数据

对于一个组件，data 数据的来源可分为如下两种：
1、组件自身定义的数据；
2、从父组件中传入的数据；
3、computed 中定义的数据；
以上三种，我们都可以通过 `this.data.get()` 方法获取。

组件自身定义的数据（状态）保存在该组件的 data 中，可以对其进行修改从而影响当前组件以及子组件的视图。从父组件传入的数据，可以从 data 中获取，但通常我们只是使用这个数据，如果要更改从父组件传入的数据，虽然可以直接在组件内更改，但通常的做法是到该数据初始化的地方去更改。

data 数据应该是纯数据

data 数据可以是字符串、数值、 数组、原生对象这样的纯数据。对于正则表达式，纯函数这样的，如果是在组件自身使用，则需要外部引入，或者作为组件的方法。 但如果要传递给子组件使用，则可以存放在 data 中。 例如，在表单组件中，我们可能会在业务层自定义验证方法，传入子组件使用。

data bind时的auto camel

在 san 组件中，data 的键值必须遵守 camelCase (驼峰式)的命名规范，不得使用 kebab-case (短横线隔开式)规范。

场景一

当一个父组件调用子组件并进行 data 绑定时，如果某一项属性写法使用了 kebab-case，san 会自动将其转换为 camelCase，然后传入子组件。下面的一个例子说明了这一点：

示例一

```
class Child extends san.Component {
  static template = `
    <ol>
      <li>{{dataParent}}</li>
      <li>{{data-parent}}</li>
    </ol>
  `;
}

class Parent extends san.Component {
  static template = `
    <div>
      <san-child data-parent="data from parent!"/>
    </div>
  `;

  static components = {
    'san-child': Child
  };
}

new Parent().attach(document.body);
```

See the Pen vJQgWm by Ma Lingyang (@mly-zju) on CodePen.

分析

上面例子中，父组件调用子组件，为 `data-parent` 属性传入了”data from parent!”字符串。在

子组件中，同时在li标签中输出 `dataParent` 和 `data-parent` 属性的值，可以看到， `dataParent` 打印出的正是父组件绑定的值，作为对比， `data-parent` 并没有输出我们期望的绑定值。从这个例子中可以很明显看出，对于传入的属性键值，san会自动将 kebab-case 写法转换为 camelCase。而作为对比，在原生 html 标签中，并不会有 auto-camel 的特性，我们如果传入一个自定义的 kebab-case 写法的属性，依然可以通过 `dom.getAttribute('kebab-case')` 来进行读取。san 的 template 与原生 html 的这一点不同值得我们注意。

在这个场景中的 auto camel 是很有迷惑性的，这个特性很容易让我们误以为在开发中，定义组件的属性键值时候我们可以随心所欲的混用 camelCase 和 kebab-case，因为反正 san 会自动帮我们转换为 camelCase 形式。那么，实际上是不是如此呢？来看场景二。

场景二

在场景一中，父组件为子组件绑定了一个 kebab-case 写法的属性，被自动转换为 camelCase。那么在子组件中，如果自身返回的初始 data 属性本身就是 kebab-case 类型，又会出现怎样的情况呢？我们看第二个例子：

示例二

```
class Child extends san.Component {
  static template = `
    <ol>
      <li>{{dataSelf}}</li>
      <li>{{data-self}}</li>
    </ol>
  `;

  initData() {
    return {
      'data-self': 'data from myself!'
    }
  }
}

new Child().attach(document.body);
```

See the Pen QMJpvL by Ma Lingyang (@mly-zju) on CodePen.

分析

在上面例子中，Child 组件初始 data 中包含一项键值为 `data-self` 的数据。我们将其分别

以 `dataSelf` 和 `data-self` 打印到 li 标签中，可以看到，两种都没有正确打印出我们初始化的值。说明对于自身 data 属性而言，如果属性的键值不是 camelCase 的形式，san 并不会对其进行 auto camel 转换，所以我们无论以哪种方式，都无法拿到这个数据。

原理分析

在 san 的 compile 过程中，对 template 的解析会返回一个 ANODE 类的实例。其中 template 中绑定属性的时候，属性对象的信息会解析为 ANODE 实例中的 props 属性。对于子组件来说，会根据父组件的 aNode.props 来生成自身的 data binds。

在 san 中，非根组件做 data binds 过程中，接受父组件的 aNode.props 这一步时，会做 auto camel 处理。这就解释了上述两个例子为什么父组件 kebab 属性传入后，子组件 camel 属性表现正常，其余情况都是异常的。事实上在 san 的源码中，我们可以找到相关的处理函数：

```
1.  function kebab2camel(source) {
2.      return source.replace(/-([a-z])/g, function (match, alpha) {
3.          return alpha.toUpperCase();
4.      });
5.  }
6.
7.  function camelComponentBinds(binds) {
8.      var result = new IndexedList();
9.      binds.each(function (bind) {
10.         result.push({
11.             name: kebab2camel(bind.name),
12.             expr: bind.expr,
13.             x: bind.x,
14.             raw: bind.raw
15.         });
16.     });
17.
18.     return result;
19. }
```

在生成子组件的绑定过程中，正是由于调用了 camelComponentBinds 这个函数，所以才有 auto camel 的特性。

结论

san 的 auto camel 只适用于父组件调用子组件时候的数据绑定。对于一个组件自身的初始数据，如果属性为 kebab-case，我们将无法正确拿到数据。因此，在写 san 组件的过程中，无论何时，

对于 data 中的属性键值，我们都应该自觉地严格遵循 camelCase 规范。

试图模板

- 如何遍历一个对象？
- 数组深层更新如何触发视图更新？
- 如何实现元素的显示/隐藏？

如何遍历一个对象？

在San中已经提供了 san-for 指令（可以简写为 s-for ）将 Array 渲染为页面中的列表，那么对于 Object 想要进行遍历并渲染应当怎么做呢？由于 San 的指令并不直接支持 Object 的遍历，因此可以使用计算属性进行对象的遍历

使用

```
class MyComponent extends San.component {
    static computed = {
        list() {
            let myObject = this.data.get('myObject');
            return Object.keys(myObject).map(item => {
                return {
                    key: item,
                    value: myObject[item]
                }
            });
        }
    };
}
```

示例

See the Pen san-traverse-object by liuchaofan (@asd123freedom) on CodePen.

数组深层更新如何触发视图更新？

在 San 组件中，对数据的变更需要通过 `set` 或 `splice` 等方法，实现用最简单的方式，解决兼容性的问题，同时为了保证数据操作的过程可控，San 的数据变更在内部是 Immutable 的，因此遇到数组深层做数据交换时直接 set 数据会发现没有触发视图的更新

场景描述

```
class MyApp extends san.Component {
    static template = `
        <div>
            <div
                style="cursor: pointer"
                on-click="handlerClick($event)">点我交换数据</div>
            <ul>
                <li s-for="item in list">{{item.title}}</li>
            </ul>
        </div>
    `;
    initData() {
        return {
            list: [
                {
                    title: 'test1'
                },
                {
                    title: 'test2'
                }

            ]
        };
    }
    handlerClick() {

        // 想交换两个值
        let firstNews = this.data.get('list');
        let firstData = firstNews[0];
        let secondData = firstNews[1];
        firstNews[1] = firstData;
        firstNews[0] = secondData;

```

```
        // 在这里直接set数据发现并没有触发视图的更新
        this.data.set('list', firstNews);
    }
}

let myApp = new MyApp();
myApp.attach(document.body);
```

原因分析

San 的 data 的数据是 Immutable 的，因此 set firstNews 时变量的引用没变， diff 的时候还是相等的，不会触发更新。

解决方式如下

See the Pen
数组深层更新触发视图更新
by solvan(@sw811) on
CodePen.

如何实现元素的显示/隐藏？

通过 `s-if` 指令，我们可以为元素指定条件。只有当条件成立时元素才会渲染，否则元素不会被加载。

但 `s-if` 无法实现这样的需求：我们需要在符合条件的情况下显示某元素，条件不满足时，元素在页面中隐藏，但依然被挂载到 DOM 。这个时候，元素的展现用 CSS 控制更为合适。

这一需求的本质可以归纳为：如何根据条件实现元素的显示/隐藏。

如何处理

San 提供在视图模板中进行样式处理的方案，详见教程。你可以用不同的 class 控制样式，也可以用 inline 样式实现。

1. 用 class 控制元素的显示与隐藏

```
1.  <!-- template -->
2.  <div>
3.      <ul class="list{{isHidden ? ' list-hidden' : ' list-visible'}}"></ul>
4.  </div>
```

注意， class 属性有多个类名时，需要为第一个以后的类名加上空格。

codepen 演示如下：

See the Pen
根据条件添加不同样式－用class控制
by MinZhou (@Mona_) on
CodePen.

CSS 控制着样式的展现，所以 DOM 始终都存在页面节点树中。你可以打开控制台看看。

2. 用内联样式控制元素的隐藏与显示

```
1.  <!-- template -->
2.  <div>
3.      <ul style="display: {{isHidden ? 'none' : 'block'}}">visible</ul>
4.  </div>
```

See the Pen
根据条件添加不同样式－用内联样式控制

by MinZhou (@Mona_) on CodePen.

有时候数据可能并不存在，所以把样式名包含在插值中更为可靠。

```
<!-- template -->
<div>
    <ul style="{{isHidden === false ? 'display: none' : 'display: block'}}">visible</ul>
</div>
```

3. 使用计算属性

前面的两种方案都可以通过使用计算属性，将判断逻辑从模板中解耦出来，以便更好的应对可能变得更为复杂的需求。下面是基于 class 的例子：

```
san.defineComponent({
    template: `
        <div>
            <ul class="{{ulClass}}"></ul>
        </div>
    `,
    computed: {
        ulClass() {
            const isHidden = this.data.get('isHidden');
            if (isHidden) {
                return 'list list-hidden';
            }
            return 'list list-visible';
        }
    }
})
```

codepen 演示如下：

See the Pen 基于 computed 的元素显示隐藏 by LeuisKen (@LeuisKen) on CodePen.

4. 使用 filter

filter 也可以用于对 class 和 style 进行处理，解耦的效果和 computed 类似，其特点是能够显式地声明属性值与数据的依赖关系。下面是基于 class 的例子：

```
san.defineComponent({
    template: `
        <div>
            <ul class="{{isHidden | handleHidden}}"></ul>
        </div>
    `,
    filters: {
        handleHidden(isHidden) {
            if (isHidden) {
                return 'list list-hidden';
            }
            return 'list list-visible';
        }
    }
})
```

codepen 演示如下：

See the Pen 基于 filter 的元素显示隐藏 by LeuisKen (@LeuisKen) on CodePen.

我们可以很明显地看出，class 是由 isHidden 控制的。

这里要额外注意的是，如果和 class 关联的有多个 data ，用 filter 的方法可能会有一些问题，比如我在下面的例子中实现了一个tab组件：

```
san.defineComponent({
    template: `
        <div class="tab">
            <div
                s-for="tab in tabs"
                class="{{tab.value | mapActive}}"
                on-click="tabChange(tab.value)"
                >
                {{tab.name}}
            </div>
        </div>
    `,
    initData() {
        return {
            active: '',
            tabs: [
                {
```

```
                    name: '第一项',
                    value: 'one'
                },
                {
                    name: '第二项',
                    value: 'two'
                }
            ]
        };
    },
    tabChange(value) {
        this.data.set('active', value);
    },
    filters: {
        mapActive(value) {
            const active = this.data.get('active');
            const classStr = 'sm-tab-item';
            if (value === active) {
                return classStr + ' active';
            }
            return classStr;
        }
    }
});
```

codepen 演示如下：

See the Pen 没有显式声明依赖的tab bug演示 by LeuisKen (@LeuisKen) on CodePen.

此处当我在点击 tab 的时候，虽然 active 能够正常更新，但是视图不会引起变化，因为 San 的依赖收集机制不认为 active 的修改会影响到视图，因此需要我们在模板中显式声明对 active 的依赖，参考如下代码：

```
san.defineComponent({
    // 将下面的 mapActive 改成 mapActive(active)，显示声明视图对 active 的依赖
    template: `
        <div class="tab">
            <div
                s-for="tab in tabs"
                class="{{tab.value | mapActive(active)}}"
                on-click="tabChange(tab.value)"
                >
                {{tab.name}}
```

```
            </div>
        </div>
    `,
    initData() {
        return {
            active: '',
            tabs: [
                {
                    name: '第一项',
                    value: 'one'
                },
                {
                    name: '第二项',
                    value: 'two'
                }
            ]
        };
    },
    tabChange(value) {
        this.data.set('active', value);
        this.fire('change', value);
    },
    filters: {
        // 这里就不需要通过 this.data.get('active') 拿到 active 了
        mapActive(value, active) {
            const classStr = 'sm-tab-item';
            if (value === active) {
                return classStr + ' active';
            }
            return classStr;
        }
    }
});
```

codepen 演示如下：

See the Pen 显式声明依赖的tab演示 by LeuisKen (@LeuisKen) on CodePen.

通过在模板中显示声明视图对 active 的依赖， San 就能正常更新视图了。这也是为什么我会在一开始说 filter 的特点是能够显式地声明属性值与数据的依赖关系。

结语

隐藏和显示是开发中较为常见的需求，还有一些其他的样式切换需求，使用以上两种方法都可以轻松实现。

总结一下，如果你要控制元素的渲染与否（是否添加到节点树），你需要使用 `s-if` 指令；如果你仅仅只想控制 DOM 节点的样式，比如元素的显示/隐藏样式，请使用数据控制 `class` 或内联样式。

组件间通信

- 父组件如何更新子组件？
- 子组件如何通知父组件？
- 子组件与更高层组件如何通信？
- 动态子组件如何传递消息给父组件?

父组件如何更新子组件？

props

最简单的也是最常用的父组件更新子组件的方式就是父组件将数据通过**props**传给子组件，当相关的变量被更新的时候，MVVM框架会自动将数据的更新映射到视图上。

```
class Son extends san.Component {
    static template = `
        <div>
            <p>Son's name: {{firstName}}</p>
        </div>
    `;
};

class Parent extends san.Component {
    static template = `
        <div>
             <input value="{= firstName =}" placeholder="please input">
             <ui-son firstName="{{firstName}}"/>
        </div>
    `;

    static components = {
        'ui-son': Son
    };

    initData() {
        return {
            firstName: 'trump'
        }
    }
};
```

See the Pen san-parent-to-child-prop by liuchaofan (@asd123freedom) on CodePen.

ref

更灵活的方式是通过**ref**拿到子组件的实例，通过这个子组件的实例可以手动调用 `this.data.set` 来更新子组件的数据，或者直接调用子组件声明时定义的成员方法。

```
class Son extends san.Component {
    static template = `
        <div>
            <p>Son's: {{firstName}}</p>
        </div>
    `;
};

class Parent extends san.Component {
    static template = `
        <div>
            <input value="{= firstName =}" placeholder="please input">
            <button on-click='onClick'>传给子组件</button>
            <ui-son san-ref="son"/>
        </div>
    `;
    static components = {
      'ui-son': Son
    };
    onClick() {
      this.ref('son').data.set('firstName', this.data.get('firstName'));
    }
}
```

See the Pen san-parent-to-child-ref by liuchaofan (@asd123freedom) on CodePen.

message

除了**ref**外，父组件在接收子组件向上传递的消息的时候，也可以拿到子组件的实例，之后的操作方式就和上面所说的一样了。

```
class Son extends san.Component {
    static template = `
        <div>
            <p>Son's name: {{firstName}}</p>
            <button on-click='onClick'>I want a name</button>
        </div>
    `;

    onClick() {
        this.dispatch('son-clicked');
```

```
    }
};

class Parent extends san.Component {
    static template = `
        <div>
            <input value="{= firstName =}" placeholder="please input">
            <ui-son/>
        </div>
    `;

    // 声明组件要处理的消息
    static messages = {
        'son-clicked': function (arg) {
            let son = arg.target;
            let firstName = this.data.get('firstName');
            son.data.set('firstName', firstName);
        }
    };

    static components = {
        'ui-son': Son
    };

    initData() {
        return {
            firstName: 'trump'
        }
    }
};
```

See the Pen san-parent-to-child-prop by liuchaofan (@asd123freedom) on CodePen.

子组件如何通知父组件？

San的组件体系提供了事件功能，子组件可以通过调用组件的fire方法派发一个自定义事件，父组件在视图模板中通过on-事件名的方式或通过子组件实例的on方法就可以监听子组件派发的自定义事件，实现子组件到父组件的通信。

使用

```
var childComponent = san.defineComponent({
    template: `
        <div>
            <button on-click="onClick">change</button>
        </div>
    `,

    onClick: function () {
        // 向父组件派发一个child-change事件
        this.fire('child-change', 'from child');
    }
});

var parentComponent = san.defineComponent({
    components: {
      'my-child': 'childComponent'
    },

    template: `
        <div>
            <my-child on-child-change="changeHandler($event)"/>
        </div>
    `,

    changeHandler: function (val) {
        // 事件处理
    }

});
```

说明：我们知道使用「双向绑定」可以将子组件内部的数据变化同步给父组件，但除了类表单组件外，其它情况不建议使用「双向绑定」的方式来达到通知父组件的目的。

示例

See the Pen child-to-parent by funa (@naatgit) on CodePen.

子组件与更高层组件如何通信？

使用

子组件通过**dispatch**方法向组件树上层派发消息。

```
class Son extends san.Component {
    static template = `
        <div>
            <button on-click='onClick'>向上传递</button>
        </div>
    `;

    onClick() {
        const value = this.data.get('value');
        // 向组件树的上层派发消息
        this.dispatch('son-clicked', value);
    }
};
```

消息将沿着组件树向上传递，直到遇到第一个处理该消息的组件，则停止。通过 **messages** 可以声明组件要处理的消息。**messages** 是一个对象，key 是消息名称，value 是消息处理的函数，接收一个包含 target(派发消息的组件) 和 value(消息的值) 的参数对象。

```
class GrandParent extends san.Component {
    static template = '<div><slot></slot></div>';

    // 声明组件要处理的消息
    static messages = {
        'son-clicked': function (arg) {
            // arg.target 可以拿到派发消息的组件
            // arg.value 可以拿到派发消息的值
            this.data.set('value', arg.value);

        }
    }
};
```

示例

See the Pen higher-communication by Swan (@jiangjiu8357) on CodePen.

动态子组件如何传递消息给父组件?

title:
categories:

- practice

组件的创建中，可能需要在运行时，通过状态树渲染出一个动态组件树。通常的方法，我们通过 **dispatch/message** 但是由于父组件及子组件都是单独动态创建的，因此父子组件之间实际上是没有父子关系的，因此需要将子组件的parentComponent指向父组件，以实现动态父子组件之间的消息传递。

example

此处给一个简单的例子，我们需要根据一个简单的状态树实现一个相应的组件样式，并实现父子组件的通信:

```
const Child = san.defineComponent({
    template: `
        <div class="child">
            {{name}}<button on-click="sendMsg">send msg</button>
        </div>
    `,
    sendMsg() {
        this.dispatch('child-msg', this.data.get('msg'));
    }
});

const Parent = san.defineComponent({
    template: `
        <div class="parent" style="border: 1px solid red">
            I am parent
            <button on-click="addChild">
                add child
            </button>
            {{childMsg}}
        </div>`,

    addChild() {

```

```
        const childs = this.data.get('childs');
        const parentEl = this.el;

        childs.forEach(child => {

            let childIns = new Child({
                parent: this,
                data: child
            });

            childIns.attach(parentEl);
            this.childs.push(childIns);

        });
    },

    messages: {
        'child-msg': function(arg) {
            this.data.set('childMsg', arg.value);
        }
    }
});

const parent = new Parent({
    data: {
        childs: [{
            name: 'I am child1',
            msg: 'child1 send msg'
        }, {
            name: 'I am child2',
            msg: 'child2 send msg'
        }]
    }
});

parent.attach(document.body);
```

实例

See the Pen QMMZPV by zhanfang (@zhanfang) on CodePen.

组件管理

- 我们可以操作 DOM 吗？

我们可以操作 DOM 吗？

我们在使用 San 的时候，特别是刚刚使用不久的新人，且 MVVM 框架的经验不是那么丰富，我们还是更习惯于使用 jQuery 作为类库来操作页面的交互，于是很自然的写出了这样的代码。

```
var MyApp = san.defineComponent({
    template: '<input value="没点" class="ipt"/><button class="btn"></button>',
    attached: function () {
        this.bindSomeEvents();
    },
    bindSomeEvents: function () {
        $('.btn').click(()=>{
            $('.ipt').val('点了');
        });
    }
});
var myApp = new MyApp();
myApp.attach(document.querySelector('#app'));
```

然后用浏览器运行了这段程序，结果完全符合预期，完美~

然而当我们进一步熟悉了 San 的使用方式后，对于上面的功能我们会写出这样的代码。

```
var MyApp = san.defineComponent({
    template: '<div><input value="{{value}}"/><button on-click="clickHandler">点我</button></div>',
    initData: function () {
        return {
            value: '没点'
        };
    },
    clickHandler: function () {
        this.data.set('value', '点了')
    }
});
var myApp = new MyApp();
myApp.attach(document.querySelector('#app'));
```

仔细推敲了下这两段代码，不禁产生了一个疑问。

直观的来看，San 的代码中我们直接调用 this.data.set 来修改某个属性的值，它自动将修改后的内容渲染到了 DOM 上，似乎看起来非常的神奇，但是它的根本上还是对 DOM 进行的操作，只不过这

个操作是San框架帮你完成的，既然是这样，那我们为什么不能直接像第一段代码一样，直接修改，而要把这些操作交给 San 来完成呢？如果从性能上考虑交给 San 来做，它要完成从 Model 到视图上的关系绑定，还需要有一部分性能的损失，这样看起来代价还挺大的，那我们为什么还要这么做呢？

带着这个问题，我们可以从这几方面进行考虑。

使用 San 的初衷？

San 是一个 MVVM（Model-View-ViewModel） 的组件框架，借助 MVVM 框架，我们只需完成包含 声明绑定 的视图模板，编写 ViewModel 中业务数据变更逻辑，View 层则完全实现了自动化。这将极大的降低前端应用的操作复杂度、极大提升应用的开发效率。MVVM 最标志性的特性就是 数据绑定 ，MVVM 的核心理念就是通过 声明式的数据绑定 来实现 View 层和其他层的分离，完全解耦 View 层这种理念，也使得 Web 前端的单元测试用例编写变得更容易。

简单来说就是：操作数据，就是操作视图，也就是操作 DOM。

此 DOM 非彼 DOM

在我们写的代码中的 template 属性，在 San 中被称作 内容模板，它是一个符合 HTML 语法规则的字符串，它会被 San 解析，返回一个 ANode 对象。

也就是说我们在 template 中写的东西实际上并不是要放到 DOM 上的，它是给 San 使用的，真正生成的 DOM 实际上是 San 根据你的 template 的解析结果也就是 ANode 生成的，你的代码与DOM之间其实还隔了一层 San。

我们如果直接使用原生的 api 或者 jQuery 来直接操作 San 生成的DOM，这是不合理的，因为那些DOM根本不是我们写的，而我们却要去试图修改它，显然我们不应该这样做。

不直接操做 DOM 这其实也是符合计算机领域中分层架构设计的基本原则的，每一层完成独立的功能，然后上层通过调用底层的 api 来使用底层暴露出来的功能，但禁止跨层的调用。

有时候我们过度的考虑了性能这个问题

San 框架极大的提升了应用的开发效率，它帮我们屏蔽繁琐的 DOM 操作，帮我们处理了 Model 与 View的关系，这看起来真的很美好，但一切美好的事情总是要付出代价的，San要做这些，就会带来性能上的开销，所以它用起来比直接操做 DOM 性能要差，这是毋庸置疑的，世界上也不可能存在这种框架性能比直接操作 DOM 还要好，如果你要改变一个页面的显示状态，DOM 是它的唯一 API，任何框架都不可能绕过。

但这种性能上的消耗真的给我的应用带来的不可维护的问题了吗，反而是大部分原因是因为我们在开发中代码结构的不合理，代码不够规范，功能划分不够清晰，等一系列主观上的问题导致的项目无法维护下去。

总之

在我们的项目中选择 San 做为框架，它不仅可以让你从繁琐的 DOM 操作中解脱出来，通过 MVVM 的模式极大的降低前端应用的操作复杂度、极大提升应用的开发效率，它的组件系统作为一个独立的数据、逻辑、视图的封装单元更是能够帮你很好的在开发中梳理好应用的代码结构，保证系统能够更加易于维护。

路由管理

- 如何使用 san-router 建立一个单页应用的后台系统？

如何使用 san-router 建立一个单页应用的后台系统？

引言

众所周知，Web 系统的早期路由是由后端来实现的，服务器根据 URL 来重新加载整个页面。这种做法用户体验不但不好，而且页面若是变得复杂，服务器端的压力也会随之变大。随着 Ajax 的广泛应用，页面能够做到无需刷新浏览器也可更新数据，这也给单页应用和前端路由的出现奠定了基础。因此，在单页应用系统中使用前端路由也十分常见，很多前端框架也提供或者推荐配套使用的路由系统。san-router 是 San 框架的官方 router，以方便用户基于 san 构建单页或同构应用为目标。本文也主要来说明实践过程中如何使用 san-router 来构建一个单页面后台管理系统。

路由配置

使用 san-router 和 San 构建单页应用的系统主要基于路由和组件。路由处理放在浏览器端来直接响应浏览器地址的变换，分发到对应的路由。在路由发生变化时，通过加载相应的组件，替换需要改变的部分，来向用户呈现对应的界面。所以路由配置是比较关键的一步。

单页应用系统中应该创建一个入口 js 文件(如 main.js)，在其中可以配置相关路由，attach 一个根组件，并将路由的 target 设置为根组件中的标签：

```
1.  // main.js
2.
3.  import san from 'san';
4.  import {router} from 'san-router';
5.  import App from './App.san';
6.
7.  // attach 根组件 App
8.  new App().attach(document.getElementById('app'));
9.  // 路由规则
10. const routes = [
11.     {
12.         rule: '/',
13.         Component: Home
14.     },
15.     {
16.       rule: '/list',
17.       Component: List
18.     },
19.     {
20.       rule: '/about',
21.       Component: About
```

```
    }
];
// 将路由规则的 target 属性设置为根组件中的标签
routes.forEach(item => {
    router.add({
        ...item,
        target: '#main'
    });
});
// 设置路由模式 'html5 | hash'
router.setMode('html5');
// 设置路由监听
router.listen((e, config) => {
    // 在路由发生变化时触发
    console.log(e);
    console.log(config);
});
// 启动路由
router.start();
```

在路由规则配置过程中，通过调用 router.add({Object}options) 来添加路由规则，在 options 对象中指定 Component 和 target 参数。将特定的 URL 规则映射到相应的组件类上，在 URL 变化并匹配路由规则时，将对应逻辑子组件初始化并渲染到页面中。

> *san-router 有两种路由规则配置:*
> *rule 为 string 时，URL 的 path 部分与字符串完全匹配才可*
> *rule 为 RegExp (正则)时，URL 的 path 部分与该正则部分匹配即可*

路由规则配置完成后，可以通过调用 setMode 方法来设置路由模式，通过调用 listen 方法来添加路由监听器，当发生路由行为时被触发。

最后，可通过调用 start 方法来启动路由，根据 URL 的变化来匹配规则，渲染相应的组件到界面上。

App根组件

App 作为根组件，布局了整个系统界面不需要更新的部分，搭建出了系统界面基本的骨架。那些需要更新的部分则是在 App 组件被附加到页面后，通过启动路由，来加载不同的逻辑组件，渲染到路由规则 target 属性对应的标签里：

```
// App.san

// Link 组件
```

```
import {Link} from 'san-router';

// App Component
class App extends san.Component {
    static components = {
      'router-link': Link
    };
    static template = `
          <div class="app-container">
            <div class="app-drawer">
                <div class="drawer-title">
                  <h3>XXX管理系统</h3>
                </div>
                <div class="menu">
                    <ul>
                        <li><router-link to="/">Home</router-link></li>
                        <li><router-link to="/list">List</router-link></li>
                        <li><router-link to="/about">about</router-link></li>
                    </ul>
                </div>
            </div>
            <div class="app-bar">
                <div class="user-info">
                    <span>userName</span>
                </div>
            </div>
            <div class="app-content">
              <!-- 逻辑组件渲染处 -->
              <div id="main"></div>
            </div>
          </div>
    `;
}
```

逻辑子组件

逻辑子组件是指根据路由匹配规则渲染到页面中的业务逻辑组件。这些组件按照业务逻辑，由基础组件库中的组件组装而成，在匹配到对应路由时，进行初始化和渲染。

逻辑子组件是正规的 san 组件，每一个逻辑子组件可以放在一个单独的文件里，调用基本组件库来组装而成，设置在不同生命周期阶段想要处理的业务：

```
// About.san

class About extends san.Component {
    static template = `
        <p>关于关于</p>
    `;
    initData() {
        return {};
    }
    route() {}
    attached() {}
}

// List.san

class List extends san.Component {
    static template =  `
        <p>list,list</p>
    `;
    initData() {
        return {};
    }
    route() {}
    attached() {}
}

// Home.san

class Home extends san.Component {
    static template = `
        <p>Home</p>
    `;
    initData() {
        return {};
    }
    route() {}
    attached() {}
}
```

总结

使用 san-router 构建一个单页应用后台系统的关键点在路由配置、根组件和逻辑子组件三个方面，

如果能优雅地做好以上三个方面，就能在后期开发与扩展过程中复用组件和模块，提高开发效率。另外，单页应用基于前端路由、组件化思想和前端数据流方案。因此，在构建一个单页应用系统时，还需关注前端数据流管理，对于业务比较复杂多变的后台管理系统，复用组件、有效管理 Ajax 请求和前端数据流有利于提高开发和维护效率。所以单页应用在实践中也被广泛应用，但是每种技术方案有其局限性，单页应用要在一个页面上提供所有功能，首次需要加载大量资源，资源加载时间也相对较长，在选择技术方案时还需要兼顾具体应用场景。

示例

See the Pen san-router-spa by sqliang (@sqliang) on CodePen.

应用状态管理

- 如何使用 `san-store` 实现后台系统的状态管理？

如何使用 san-store 实现后台系统的状态管理？

引言

首先确保已经理解了san-store 中是否需要状态管理的内容以及相关概念，下面开始。

本项目代码在 https://github.com/jiangjiu/san-store-spa 可以查看。

搭建环境

上一篇文档 如何使用 san-router 建立一个单页应用的后台系统? 已经搭建了一个san+san-router的单页后台应用，我们在它的基础上加入san-store来管理应用状态。

```
1.      // 只需安装san-store和san-update
2.      npm i san-update san-store --save
```

状态设计

目前系统有三个频道，home、about、list。

假设这是一个类似电商后台管理订单的系统:

1. 不同的频道都需要同步当前订单的状态（待付款、待发货、交易完成 => orderState:1、2、3）
2. 不同频道有权利修改当前订单状态
3. 每次修改都需要异步请求到服务端进行确认

这里的状态管理混合了异步请求，为了简单起见，暂不考虑安全性及异常处理。

思考

如果不使用san-store，每一个频道都需要自行发起异步请求，同时要和其他频道通信当前的订单状态，在实际业务中会是件很头疼的事儿。

使用san-store后，异步请求在action中发起而无需在不同组件中分别处理，同时store作为唯一应用状态源，无需考虑信息同步问题，系统流程清晰很多，简单可靠。

创建store

首先新建一个文件来初始化和管理store。

```
1.  // store.js
2.  import {updateBuilder} from 'san-update/src/index';
```

```
import {store} from 'san-store';

// 第一个action，处理边界条件和异步请求
store.addAction('changeOrderState', (state, {getState, dispatch}) => {
    // 取出当前订单状态值，如果为空就初始化为1
    const orderState = getState('orderState');
    if (!state) {
        return dispatch('fillOrderState', 1);
    }
    // 如果改变的订单值和原来状态相同或异常值就不更新了
    else if (state === orderState || state < 1 || state > 3) {
        return;
    }
    // 符合修改条件后，发起异步请求
    axios.post('/api/orderState', {state})
        .then(res => {
            // 状态码正确，修改store中的订单值
            if (res.status === 200) {
                dispatch('fillOrderState', state);
            }

        })
        .catch(error => {
            console.log(error);
        });
});
// 同步orderState值
store.addAction('fillOrderState', state => updateBuilder().set('orderState', state));

// 给订单状态一个初始值
store.dispatch('fillOrderState', 1);
```

初始值

看到上面的 `store.dispatch('fillOrderState', 1)` 了吗？
这是为了给订单状态一个初始值。
为什么会这样做？

也许你会想到san-store手动实例化store时中提供了initData属性：

```
let myStore = new Store({
```

```
    initData: {
        user: {
            name: 'your name'
        }
    },

    actions: {
        changeUserName(name) {
            return builder().set('user.name', name);
        }
    }
})
```

这确实是个很不错的初始办法。
可惜的是，`connect.san`方法只能连接san-store默认提供的store，手动实例化的store无法使用`connect.san`方法。

而且erik和灰大在设计之初认为：

1. store应该只存在一个（按常理出牌），如果提供连接其他store的方法，可能会在业务中被玩坏
2. 大部分初始值都是异步获取的，仍然需要dispatch action获得

所以当初并没有提供手动指定store进行连接的能力。
好消息是，我们会在近期开放这个功能，敬请期待。

入口文件引入store.js

别忘了在main.js中添加store.js。

```
// 入口文件 main.js
import './store';
```

修改频道

为不同频道增加显示以及修改订单状态。

首先修改Home频道。

```
// 修改Home.js
import {connect} from 'san-store';
import san from 'san';

```

```
const Home = san.defineComponent({
    template: `
        <div>
            <p>目前状态：{{orderState}}</p>
            <button on-click="onClick">订单更改为状态2：待发货</button>
        </div>
    `,
    onClick() {
        // 改变订单状态至待发货，简单起见就不做成下拉框可选形式了
        this.actions.changeOrderState(2);
    }
});

// 连接这个组件至store
export default connect.san(
    {orderState: 'orderState'},
    {changeOrderState: 'changeOrderState'}
)(Home);
```

然后修改About频道。

```
// 修改 About.js
import {connect} from 'san-store';
import san from 'san';

const About = san.defineComponent({
    template: `
        <div>
            <span>目前状态：{{orderState}}</span>
            <button on-click="onClick">订单更改为状态3：交易完成</button>
        </div>
    `,

    onClick() {
        // 改变订单状态至交易完成，简单起见就不做成下拉框可选形式了
        this.actions.changeOrderState(3);
    }
});

export default connect.san(
    {orderState: 'orderState'},
    {changeOrderState: 'changeOrderState'}
```

```
22.  )(About);
```

就改两个频道好了。
可以看到，不同路由下（Home、About）都正确显示了订单状态orderState，同时不同频道修改成不同的订单状态也无需手动监听通信，san-store自动完成了orderState的更新。

总结

以上只是一个简单的例子，演示了后台系统如何添加store来管理应用状态。

> 我们并不认为 *san-store* 适合所有场景。统一的进行应用状态管理，只有当你的应用足够大时，它带来维护上的便利才会逐渐显现出来。如果你只是开发一个小系统，并且预期不会有陆续的新需求，那我们并不推荐你使用它。大多数增加可维护性的手段意味着拆分代码到多处，意味着你没有办法在实现一个功能的时候一路到尾畅快淋漓，意味着开发成本可能会上升。

所以，你应该根据你要做的是一个什么样的应用，决定要不要使用 san-store。

FAQ

- Q&A集锦

Q&A集锦

这里是一些常见问题的解答，有任何问题随时给我们提 issue，我们会持续更新这里，方便您尽快发现问题的答案

实践问题

1. `san-for` 和 `san-ref` 可以一起使用吗？
A: 可以的，使用方式
为https://github.com/baidu/san/blob/master/test/component.spec.js#L1547

其它问题

如何处理绝对定位组件的 DOM？

在我们使用 San 开发的时候，我们常常会写各种的组件，当一个父组件的子组件是绝对定位组件(比如：Select、Tip 等)的时候，我们会遇到两种场景：

- 场景一：父(祖)组件足够大或不存在 `overflow: hidden;`
- 场景二：父(祖)组件不够大且存在 `overflow: hidden;`

而这两种情形下，我们需要对绝对定位组件的 DOM 做一些处理。

那针对这两种场景我们分别可以如何处理呢？

如何处理

场景一

父(祖)组件足够大或不存在 `overflow: hidden;` 时的绝对定位。

这种情况比较常规，我们可以直接引入组件，然后可选择在外部或组件内部包含一个 (not static) 元素，来控制显示即可。

使用

```
class AbsComponent extends san.Component {
    static template = `
        <div>
            <p class="absolute">子绝对定位组件</p>
        </div>
    `;
}

class Parent extends san.Component {
    static template = `
        <div>
            <div class="parent-rel">
                <h3>父是static</h3>
                <abs-comp></abs-comp>
            </div>
        </div>
    `;

    static components = {
        'abs-comp': AbsComponent
```

```
    };
}

new Parent().attach(document.querySelector('#paIsRel'));
```

示例

See the Pen
position-absolute-dom
by dengxiaohong (@The-only)
on CodePen.

场景二

父(祖)组件不够大且存在 `overflow: hidden;` 时的绝对定位。

这种情况会很常见，如果直接包含的话绝对定位元素会因为父(祖)组件有 `overflow: hidden;` 且不够大而导致组件中超出部分被遮住。

若不想被遮住的话，我们可以在组件中做一层处理：

- 将组件元素挂到 body 上
- 需要显示的时候进行位置控制
- 父组件调用

使用

```
class AbsComponent extends san.Component {
    static template = `
        <div class="abs-wrap">
            <div class="abs" style="{{mainStyle}}">
                show content
            </div>
        </div>
    `;

    initData() {
        return {
            targetElem: document.body,
            mainStyle: ''
        };
    }

    attached() {
```

```
18.         // 将绝对定位元素放在body上
19.         if (this.el.parentNode !== document.body) {
20.             document.body.appendChild(this.el);
21.         }
22.
23.         // 显示的时候进行位置控制
24.         this.changePosition();
25.     }
26.
27.     /**
28.      * 调整位置信息
29.      */
30.     changePosition() {
31.         // 这里可以替换成封装的组件来进行位置控制
32.         let targetElem = this.data.get('targetElem');
33.         targetElem = typeof targetElem === 'function' ? targetElem() :
    targetElem;
34.
35.         let rect = targetElem.getBoundingClientRect();
36.         let left = rect.left;
37.         let top = document.body.scrollTop + rect.top + rect.height;
38.
39.         let str = 'left:' + left + 'px;top:' + top + 'px;';
40.         this.data.set('mainStyle', str);
41.     }
42. }
43.
44. class Parent extends san.Component {
45.     static template = `
46.         <div>
47.             <p class="info">父元素100px overflow:hidden;</p>
48.             <div class="parent-wrap">
49.                 <span class="btn">click toggle</span>
50.                 <abs-comp targetElem="{{getTarget}}"></abs-comp>
51.             </div>
52.         </div>
53.     `;
54.
55.     static components = {
56.         'abs-comp': AbsComponent
57.     };
58.
```

```
59.     initData() {
60.         return {
61.             getTarget: this.getTarget.bind(this)
62.         };
63.     }
64.
65.     /**
66.      * 获取相对定位的元素
67.      *
68.      * @return {HTMLElemnt} 相对定位的元素
69.      */
70.     getTarget() {
71.         return this.el.querySelector('.btn');
72.     }
73. }
74.
75. new Parent().attach(document.querySelector('#instance'));
```

示例

See the Pen
position-absolute-dom
by dengxiaohong (@The-only)
on CodePen.

教程

- 安装
- 背景
- 开始
- 模板
- 数据操作
- 数据校验
- 样式
- 条件
- 循环
- 事件处理
- 表单
- 插槽
- 过渡
- 组件
- 组件反解（old）
- 组件反解
- 服务端渲染

安装

你可以通过任何喜欢或者习惯的方式安装和使用 San。

下载

直接下载

从 下载页面 可以获得最新以及过往版本的下载地址。

CDN

通过 unpkg，你可以无需下载，直接引用。

开发版本：

```
<script src="https://unpkg.com/san@latest/dist/san.dev.js"></script>
```

生产版本：

```
<script src="https://unpkg.com/san@latest"></script>
```

> 建议在开发环境不要用生产版本，开发版本提供了有助于开发的错误提示和警告！

NPM

在使用 san 来构建大型应用时我们推荐使用 NPM 来安装。通过它能够方便的管理依赖包，以及和社区的各种开发构建工具良好配合，构建你的应用程序。

```
# 安装最新版本
$ npm install san
```

使用

script

在页面上通过 script 标签引用需要的文件是常用的方式。可以引用下载下来的 San，也可以通过 CDN 引用。

```
<!-- 引用直接下载下来的San -->
<script src="san的目录/dist/san.js"></script>

<!-- 引用通过NPM下载下来的San -->
<script src="node_modules/san/dist/san.js"></script>
```

注意：在引用时，

- 如果页面上没有 AMD 环境，将会在页面上注册全局变量 `san`
- 如果页面上有 AMD 环境，将会注册为模块 `san`

AMD

将 San 下载下来后，通过 AMD 的方式引用 src 目录下的 main.js，可以获得灵活的模块名和整体编译的好处。但是你可能需要先配置好 packages 或 paths 项。

```
require.config({
    packages: [
        {
            name: 'san',
            location: 'san-path/dist/san'
        }
    ]
});
```

在这个例子里，我们可以看到一个通过 AMD 管理模块的项目是怎么引用 San 的。

ESNext

在支持 ESNext 的环境中，可以直接引用

```
import san from 'san';
```

San component

一个语法如下的 `.san` 文件，就是一个 `San component`

```
<template>
    <div class="hello">hello {{msg}}</div>
</template>

```

```
<script>
    export default {
        initData () {
            return {
                msg: 'world'
            };
        }
    }
</script>

<style>
    .hello {
        color: blue;
    }
</style>
```

在 `webpack` 中可以使用 san-loader 来加载 `.san` 文件

在 这个例子 里，
我们可以看到如何使用 `San component` 构建一个应用

开发版本 vs 生产版本

在开发中，我们推荐使用 `san.dev.js`（位于 `san/dist/san.dev.js`）。`san.dev.js` 提供了包括 数据校验 等辅助开发功能。这些辅助开发功能可以帮助你在更轻松、快速地定位和解决问题。

但出于性能考虑，正式的生产环境上需要移除了这些辅助开发功能。在 san 的发布包中提供了构建好的生产版本给大家使用，即 `san.js`（位于 `san/dist/san.js`）。你应当在构建应用的生产版本时使用它。

如果你使用 webpack 进行开发和构建 ，那么你可以通过在 webpack 配置添加 `resolve.alias` 再配合指定 `NODE_ENV` 来解决：

```
{
    module: {
        loaders: [
            {
                test: /\.san$/,
                loader: 'san-loader'
            }
        ]
    },
```

```
    resolve: {
        alias: {
            san: process.env.NODE_ENV === 'production'
                ? 'san/dist/san.js'
                : 'san/dist/san.dev.js'
        }
    }
}
```

最后，你可以通过添加两个 npm scripts 来使用不同的 webpack 配置：

```
{
    "name": "my-san-app",
    "scripts": {
        "dev": "NODE_ENV=development webpack-dev-server --config webpack.config.js",
        "build": "NODE_ENV=production webpack --config webpack.config.js"
    }
}
```

开始开发：

```
npm run dev
```

开始构建：

```
npm run build
```

背景

引言

2008年，V8 引擎随 Chrome 浏览器横空出世，JavaScript 这门通用的 Web 脚本语言的执行效率得到质的提升。 V8 引擎的出现，注定是 JavaScript 发展史上一个光辉的里程碑。它的出现，让当时研究高性能服务器开发、长时间一筹莫展的 Ryan Dahl 有了新的、合适的选择，不久，在2009年的柏林的 JSConf 大会上，基于 JavaScript 的服务端项目 Node.js 正式对外发布。Node.js 的发布，不仅为开发者带来了一个高性能的服务器，还很大程度上推动了前端的工程化，带来了前端的大繁荣。与此同时，因为 JavaScript 执行效率的巨大提升，越来越多的业务逻辑开始在浏览器端实现，前端逻辑越来越重，前端架构随之提上日程。于是，我们谈论的主角，MVVM 模式，走进了 Web 前端的架构设计中。

概念

MVVM 模式，顾名思义即 Model-View-ViewModel 模式。它萌芽于2005年微软推出的基于 Windows 的用户界面框架 WPF ，前端最早的 MVVM 框架 knockout 在2010年发布。

一句话总结 Web 前端 MVVM：操作数据，就是操作视图，就是操作 DOM（所以无须操作 DOM ）。

无须操作 DOM ！借助 MVVM 框架，开发者只需完成包含 声明绑定 的视图模板，编写 ViewModel 中业务数据变更逻辑，View 层则完全实现了自动化。这将极大的降低前端应用的操作复杂度、极大提升应用的开发效率。MVVM 最标志性的特性就是 数据绑定 ，MVVM 的核心理念就是通过 声明式的数据绑定 来实现 View 层和其他层的分离。完全解耦 View 层这种理念，也使得 Web 前端的单元测试用例编写变得更容易。

MVVM，说到底还是一种分层架构。它的分层如下：

- Model：域模型，用于持久化
- View：作为视图模板存在
- ViewModel：作为视图的模型，为视图服务

Model 层

Model 层，对应数据层的域模型，它主要做 域模型的同步 。通过 Ajax/fetch 等 API 完成客户端和服务端业务 Model 的同步。在层间关系里，它主要用于抽象出 ViewModel 中视图的 Model。

View 层

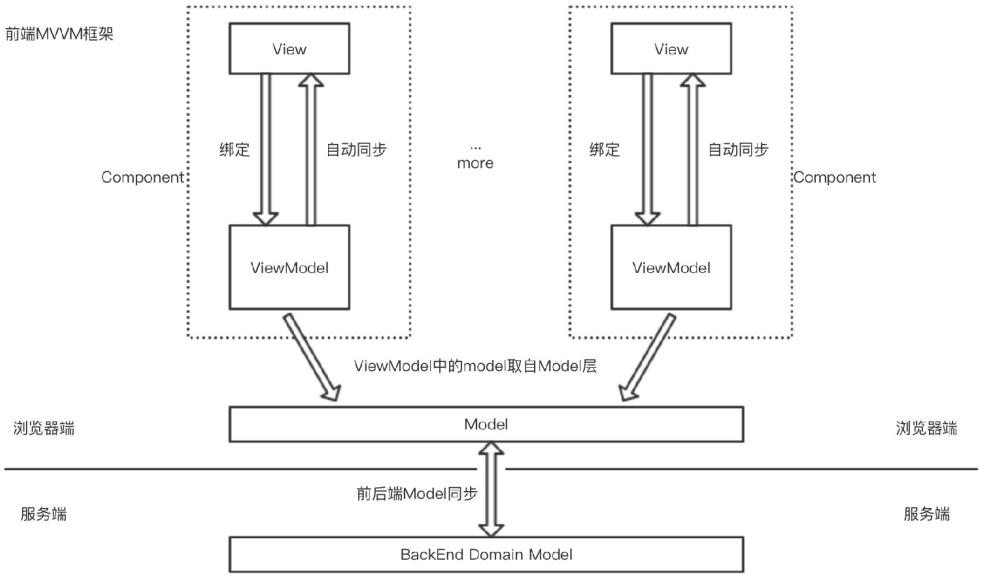
View 层，作为视图模板存在，在 MVVM 里，整个 View 是一个动态模板。除了定义结构、布局外，

它展示的是 ViewModel 层的数据和状态。View 层不负责处理状态，View 层做的是 数据绑定的声明、 指令的声明、 事件绑定的声明。

ViewModel 层

ViewModel 层把 View 需要的层数据暴露，并对 View 层的 数据绑定声明、 指令声明、 事件绑定声明 负责，也就是处理 View 层的具体业务逻辑。ViewModel 底层会做好绑定属性的监听。当 ViewModel 中数据变化，View 层会得到更新；而当 View 中声明了数据的双向绑定（通常是表单元素），框架也会监听 View 层（表单）值的变化。一旦值变化，View 层绑定的 ViewModel 中的数据也会得到自动更新。

前端 MVVM 图示



如图所示，在前端 MVVM 框架中，往往没有清晰、独立的 Model 层。在实际业务开发中，我们通常按 **Web Component** 规范来组件化的开发应用，Model 层的域模型往往分散在在一个或几个 Component 的 ViewModel 层，而 ViewModel 层也会引入一些 View 层相关的中间状态，目的就是为了更好的为 View 层服务。

开发者在 View 层的视图模板中声明 数据绑定、 事件绑定 后，在 ViewModel 中进行业务逻辑的数据 处理。事件触发后，ViewModel 中 数据 变更， View 层自动更新。因为 MVVM 框架的引入，开发者只需关注业务逻辑、完成数据抽象、聚焦数据，MVVM 的视图引擎会帮你搞定 View。因为数据驱动，一切变得更加简单。

MVVM框架的工作

不可置否，MVVM 框架极大的提升了应用的开发效率。It’s amazing！But，MVVM 框架到底做了什么？

- 视图引擎

视图引擎：我是视图引擎，我为 View 层作为视图模板提供强力支持，开发者，你们不需要操作 DOM ，丢给我来做！

- 数据存取器

数据存取器：我是数据存取器，我可以通过 `Object.defineProperty()` API 轻松定义，或通过自行封装存取函数的方式曲线完成。我的内部往往封装了 发布/订阅模式，以此来完成对数据的监听、数据变更时通知更新。我是 数据绑定 实现的基础。

- 组件机制

组件机制：我是组件机制。有追求的开发者往往希望按照面向未来的组件标准 — **Web Components** 的方式开发，我是为了满足你的追求而生。MVVM 框架提供组件的定义、继承、生命周期、组件间通信机制，为开发者面向未来开发点亮明灯。

- more…

结语

> 有了前端 *MVVM* 框架，应用开发如此简单*!*

前端 MVVM 已是趋势，是大型 Web 应用开发效率提升的利器。由百度 EFE 出品的 MVVM 框架 — san，在保持功能强大、特性支持完整的前提下，还兼顾到IE8的市场份额，对老版本浏览器提供了良好的兼容性，更难能可贵的是 GZip 后体积仅 **11k**， 现已为百度内多个产品提供了强劲驱动，可谓百度 EFE 又一精工之作！开源的 san 欢迎广大开发者体验、使用，更欢迎广大开发者加入到 san 生态 的建设中来，让 san 变得更好！

开始

San，是一个 MVVM 的组件框架。它体积小巧（13K），兼容性好（IE6），性能卓越，是一个可靠、可依赖的实现响应式用户界面的解决方案。

San 通过声明式的类 HTML 视图模板，在支持所有原生 HTML 的语法特性外，还支持了数据到视图的绑定指令、业务开发中最常使用的分支、循环指令等，在保持良好的易用性基础上，由框架完成基于字符串的模板解析，并构建出视图层的 节点关系树，通过高性能的视图引擎快速生成 UI 视图。San 中定义的数据会被封装，使得当数据发生有效变更时通知 San 组件，San 组件依赖模板编译阶段生成的节点关系树，确定需要变更的最小视图，进而完成视图的异步更新，保证了视图更新的高效性。

组件是 San 的基本单位，是独立的数据、逻辑、视图的封装单元。从页面角度看，组件是 HTML 元素的扩展；从功能模式角度看，组件是一个 ViewModel。San 组件提供了完整的生命周期，与 WebComponent 的生命周期相符合，组件间是可嵌套的树形关系，完整的支持了组件层级、组件间的通信，方便组件间的数据流转。San 的组件机制，可以有效支撑业务开发上的组件化需求。

San 支持组件反解，以此提供服务端渲染能力，可以解决纯前端渲染导致的响应用户交互时延长、SEO 问题。除此之外，San 还提供了一些周边开源产品，与 San 配合使用，可以帮助开发者快速搭建可维护的大型 SPA 应用。

现在，我们从一些简单的例子，开始了解 San。这些例子可以从这里找到。

Hello

```
var MyApp = san.defineComponent({
    template: '<p>Hello {{name}}!</p>',

    initData: function () {
        return {
            name: 'San'
        };
    }
});


var myApp = new MyApp();
myApp.attach(document.body);
```

可以看到，通常情况使用 San 会经过这么几步：

1. 我们先定义了一个 San 的组件，在定义时指定了组件的 内容模板 与 初始数据 。

2. 初始化组件对象
3. 让组件在相应的地方渲染

额外提示：在 JavaScript 中书写 HTML 片段对维护来说是不友好的，我们可以通过 WebPack、AMD plugin、异步请求等方式管理。这里为了例子的简单就写在一起了。

列表渲染

```
var MyApp = san.defineComponent({
    template: '<ul><li s-for="item in list">{{item}}</li></ul>',

    attached: function () {
        this.data.set('list', ['san', 'er', 'esui', 'etpl', 'esl']);
    }
});

var myApp = new MyApp();
myApp.attach(document.body);
```

上面的例子使用 for 指令对列表进行遍历并输出。这里有个很常用的实践方法：在生命周期 **attached** 中重新灌入数据，使视图刷新。在这里，我们可以发起获取数据的请求，在请求返回后更新数据。

双向绑定

```
var MyApp = san.defineComponent({
    template: ''
        + '<div>'
        +   '<input value="{= name =}" placeholder="please input">'
        +   'Hello {{name}}!'
        + '</div>'
});

var myApp = new MyApp();
myApp.attach(document.body);
```

双向绑定通常出现在用户输入的场景，将用户输入结果自动更新到组件数据。在这个例子中，通过 **{= expression =}** 声明双向绑定，把输入框的 value 与数据项 name 绑定起来。

模板

在模板 HTML 中，我们通过一定的数据绑定语法编写，相应的数据就能在视图上呈现。

场景

插值

和许多模板引擎一样，插值的语法形式是表达式位于双大括号中，表达式后可以接任意多个过滤器。

```
{{ expr [[| filter-call1] | filter-call2...] }}
```

在文本内容区域我们可以使用插值替换。

```
<p>Hello {{name}}!</p>
```

在 HTML 标签的 属性(attribute) 内，我们也可以使用插值替换。

```
<span title="This is {{name}}">{{name}}</span>
```

使用过滤器可以对插值数据进行处理。

```
<p>Hello {{name | upper}}!</p>
```

表达式 ：在插值替换的双大括号中，San 提供了丰富的表达式支持，具体请参考本篇后续的表达式章节。

过滤器 ：过滤器相关描述请参考本篇后续的过滤器章节。

提示 ：插值将默认进行 HTML 转义。如果不需要 HTML 转义请参考本篇后续的输出HTML章节。

属性绑定

顾名思义，属性绑定的意思是，将数据绑定到子组件的 属性(property) 上。属性绑定的形式和插值绑定相同，通过 HTML 标签的 属性(attribute)声明，通常情况只声明一个 **{{ expression }}**。

下面的例子中，当 jokeName 数据变化时，会自动将新的值设置到 label 组件的 text 属性中。

```
<ui-label text="{{jokeName}}"></ui-label>
```

表达式 ：属性绑定支持任意类型的表达式，具体请参考本篇后续的表达式章节。

属性整体绑定

版本 ：>= 3.5.8

通过 **s-bind** ，可以为组件绑定整个数据。当**s-bind** 和单个的属性绑定并存时，单个的属性绑定将覆盖整体绑定中相应的数据项。

```
<ui-label s-bind="{{ {text: email, title: name} }}"></ui-label>

<!-- text 单个属性声明将覆盖 s-bind 中的 text 项 -->
<ui-label s-bind="{{ {text: email, title: name} }}" text="{{name}}"></ui-label>
```

提示 ：对象字面量常用于属性整体绑定。

双向绑定

通过 HTML 标签的 属性(attribute)声明 **{= expression =}** 的形式，可以声明双向绑定。

双向绑定通常出现在 用户输入 的场景，将用户输入结果自动更新到组件数据。所以我们通常在 输入表单元素 或 具有输入反馈的自定义组件 上应用双向绑定。

下面的例子将 input、select、自定义组件的 value 属性与声明的数据项（name、online、color）建立了双向绑定关系。当用户输入时，相应数据项的值会发生变化。

```
<input type="text" value="{= name =}">

<select value="{= online =}">
    <option value="errorrik">errorrik</option>
    <option value="otakustay">otakustay</option>
    <option value="firede">firede</option>
</select>

<ui-colorpicker value="{= color =}"></ui-colorpicker>
```

表达式 ：双向绑定仅支持普通变量和属性访问表达式，否则可能导致不可预测的问题。

输出HTML

有时候我们希望输出原封不动的 HTML，不希望经过 HTML 转义。在 San 里有两种方式可以做到。

1. 指令 s-html
2. 过滤器 raw。过滤器 raw 是一个虚拟过滤器

```
<p s-html="rawHTML"></p>
<p>{{rawHTML | raw}}</p>
```

表达式

San 提供了丰富的表达式类型支持，让使用者在编写视图模板时更方便。

- 数据访问(普通变量、属性访问)
- 一元否定
- 二元运算
- 二元关系
- 三元条件
- 括号
- 字符串
- 数值
- 布尔
- 数组字面量 `>= 3.5.9`
- 对象字面量 `>= 3.5.9`

下面通过插值替换的例子列举支持的表达式类型。

```
<!-- 普通变量 -->
<p>{{name}}</p>

<!-- 属性访问 -->
<p>{{person.name}}</p>
<p>{{persons[1]}}</p>

<!-- 一元否定 -->
<p>{{!isOK}}</p>
<p>{{!!isOK}}</p>

<!-- 二元运算 -->
<p>{{num1 + num2}}</p>
<p>{{num1 - num2}}</p>
<p>{{num1 * num2}}</p>
<p>{{num1 / num2}}</p>
<p>{{num1 + num2 * num3}}</p>
```

```
<!-- 二元关系 -->
<p>{{num1 > num2}}</p>
<p>{{num1 !== num2}}</p>

<!-- 三元条件 -->
<p>{{num1 > num2 ? num1 : num2}}</p>

<!-- 括号 -->
<p>{{a * (b + c)}}</p>

<!-- 数值 -->
<p>{{num1 + 200}}</p>

<!-- 字符串 + 三元条件 -->
<p>{{item ? ',' + item : ''}}</p>

<!-- 数组字面量 -->
<x-list data="{{ persons || [] }}" />

<!-- 对象字面量 -->
<x-article data="{{ {title: articleTitle, author: user} }}" />
```

注意 ：双向绑定仅支持普通变量和属性访问表达式。

过滤器

在插值替换时，过滤器可以实现对插值数据的处理和变换，使其转换成更适合视图呈现的数据形式。 注意 ：过滤器仅在插值替换时支持。

插值替换支持多过滤器处理。过滤器之间以类似管道的方式，前一个过滤器的输出做为后一个过滤器的输入，向后传递。

```
<!-- 普通变量 -->
<p>{{myVariable | html | url}}</p>
```

filter支持参数传递，参数可以是任何支持的表达式。

```
<!-- // 假设存在过滤器: comma -->
<p>{{myVariable | comma(3)}}</p>
<p>{{myVariable | comma(commaLength)}}</p>
```

```
4.  <p>{{myVariable | comma(commaLength + 1)}}</p>
```

过滤器在组件声明时注册，具体请参考组件文档。

字符实体

在我们编写 HTML 时，如果内容中包含 HTML 预留字符或特殊字符，我们需要写字符实体。San 的模板 HTML 是自解析的，由于体积的关系，只支持有限的具名字符实体：

- <
- >
-
- "
-
-
-
- ©
- ®
- ‌
- ‍
- &

```
1.  <p>LiLei &amp; HanMeiMei are a couple.</p>
2.  <p>1 + 1 &lt; 3</p>
```

除此之外，我们可以使用 `&#[entity_number];` 或 `&#x[entity_number];` 形式的编号字符实体。

```
1.  <p>LiLei &#38; HanMeiMei are a couple.</p>
2.  <p>1 + 1 &#60; 3</p>
```

数据操作

San 在组件的 data 上提供了一些数据操作的方法。通过 get 方法可以获取数据；通过 set、splice 等方法修改数据，相应的视图会被自动刷新。

说明 ：为什么是通过 San 提供的方法操作数据，而不是直接操作数据？因为defineProperty并未被国内常用的浏览器广泛支持，并且我们也并不喜欢这种侵入式的风格，所以我们选择了折中的方式。因此，只有通过 San 提供的方法修改数据，视图才会自动刷新。

初始化

通过定义 initData 方法，可以在定义组件时指定组件初始化时的数据。initData 方法返回组件实例的初始化数据。

```
san.defineComponent({
    initData: function () {
        return {
            width: 200,
            top: 100,
            left: -1000
        };
    }
});
```

get

```
{*} get({string|Object?}expr)
```

解释 ：

get 方法能够让我们从 data 中获取数据。

```
san.defineComponent({
    initData: function () {
        return {
            width: 200,
            top: 100,
            left: -1000
        };
    },
```

```
9.
10.     attached: function () {
11.         this.data.get('width'); // 200
12.     }
13. });
```

当 expr 参数为空时，将返回整个数据项的对象。提供无参的 **get** 方法，主要是为了 ESNext 的解构，能够在一个表达式中获取多个数据项。

```
1.  san.defineComponent({
2.      initData: function () {
3.          return {
4.              width: 200,
5.              top: 100,
6.              left: -1000
7.          };
8.      },
9.
10.     attached: function () {
11.         this.data.get().width; // 200
12.
13.         // top: 100
14.         // left: -1000
15.         let {top, left} = this.data.get();
16.     }
17. });
```

注意 ： **get** 方法获取的数据不能直接修改，否则可能导致不一致的问题。数据修改请使用本文下面的 **set** 、 **splice** 等方法

set

```
1.  set({string|Object}expr, {*}value, {Object?}option)
```

解释 ：

set 方法是最常用的数据修改方法，作用相当于 JavaScript 中的赋值 (=)。

用法 ：

```
1.  san.defineComponent({
```

```
    attached: function () {
        requestUser().then(this.userReceived.bind(this));
    },

    userReceived: function (data) {
        this.data.set('user', data);
    },

    changeEmail: function (email) {
        this.data.set('user.email', email);
    }
});
```

merge

```
merge({string|Object}expr, {*}value, {Object?}option)
```

`版本` : >= 3.4.0

`解释` :

merge 方法用于将目标数据对象（target）和传入数据对象（source）的键进行合并，作用类似于 JavaScript 中的 `Object.assign(target, source)` 。

`用法` :

```
san.defineComponent({
    attached: function () {
        requestUser().then(this.updateUserInfo.bind(this));
    },

    updateUserInfo: function (data) {
        this.data.merge('user', data);
    }
});
```

apply

```
apply({string|Object}expr, {function({*}):{*}}value, {Object?}option)
```

`版本` : >= 3.4.0

`解释` :

apply 方法接受一个函数作为参数，传入当前的值到函数，然后用新返回的值更新它。其行为类似 `Array.prototype.map` 方法。

`用法` :

```
san.defineComponent({
    attached: function () {
        this.data.set('number', {
            value: 1
        });
        this.updateNumber();
    },

    updateNumber: function (data) {
        this.data.apply('number', function (number) {
            return {
                value: number.value * 2
            };
        })
    }
});
```

数组方法

我们提供了一些数组操作的方法，这些方法与 JavaScript 的数组操作方法基本同名，以减少使用者的学习与记忆成本。除了 删除 操作。

`提示` : 修改数组项还是直接使用 set 方法。我们可以认为，基本上所有写 JavaScript 时使用 = 赋值的场景，都用 set 方法。

```
san.defineComponent({
    flag: function () {
        this.data.set('users[0].flag', true);
    }
});
```

push

```
{number} push({string|Object}expr, {*}value, {Object?}option)
```

解释：

在数组末尾插入一条数据。

用法：

```
san.defineComponent({
    addUser: function (name) {
        this.data.push('users', {name: name});
    }
});
```

pop

```
{*} pop({string|Object}expr, {Object?}option)
```

解释：

在数组末尾弹出一条数据。

用法：

```
san.defineComponent({
    rmLast: function () {
        this.data.pop('users');
    }
});
```

unshift

```
{number} unshift({string|Object}expr, {*}value, {Object?}option)
```

解释：

在数组开始插入一条数据。

用法：

```
san.defineComponent({
    addUser: function (name) {
```

```
        this.data.unshift('users', {name: name});
    }
});
```

shift

```
{*} shift({string|Object}expr, {Object?}option)
```

解释：

在数组开始弹出一条数据。

用法：

```
san.defineComponent({
    rmFirst: function () {
        this.data.shift('users');
    }
});
```

remove

```
remove({string|Object}expr, {*}item, {Object?}option)
```

解释：

移除一条数据。只有当数组项与传入项完全相等(===)时，数组项才会被移除。

用法：

```
san.defineComponent({
    rm: function (user) {
        this.data.remove('users', user);
    }
});
```

removeAt

```
removeAt({string|Object}expr, {number}index, {Object?}option)
```

解释：

通过数据项的索引移除一条数据。

用法：

```
san.defineComponent({
    rmAt: function (index) {
        this.data.removeAt('users', index);
    }
});
```

splice

```
{Array} splice({string|Object}expr, {Array}spliceArgs, {Object?}option)
```

解释：

向数组中添加或删除项目。

用法：

```
san.defineComponent({
    rm: function (index, deleteCount) {
        this.data.splice('users', [index, deleteCount]);
    }
});
```

option

每个数据操作方法，最后都可以包含一个类型为 Object 的数据操作选项参数对象，该对象中的参数可控制视图更新行为。

silent

解释：

静默更新数据，不触发视图变更。

force

版本：>= 3.5.5

解释：

设置相同的数据时，强制触发视图变更。该参数仅对 `set` 方法有效

数据校验

我们可以给组件的 data 指定校验规则。如果传入的数据不符合规则，那么 san 会抛出异常。当组件给其他人使用时，这很有用。

指定校验规则，需要使用 `DataTypes` 进行声明：

```
import san, {DataTypes} from 'san';

let MyComponent = san.defineComponent({

    dataTypes: {
        name: DataTypes.string
    }

});
```

`DataTypes` 提供了一系列校验器，可以用来保证组件得到的数据是合法的。在上边的示例中，我们使用了 `DataTypes.string` 。当一个 `name` 得到了一个不合法的数据值时，san 会抛出异常。

考虑到性能原因， **`dataTypes`** 只会在 **`development`** 模式下进行数据校验。

请参考这里来确认在不同的 san 发布版本中数据校验功能的支持情况。

DataTypes

下边是 `DataTypes` 提供的各种校验的一个示例代码：

```
import san, {DataTypes} from 'san';

san.defineComponent({

    // 你可以声明数据为 JS 原生类型。
    // 默认的以下这些数据是可选的。
    optionalArray: DataTypes.array,
    optionalBool: DataTypes.bool,
    optionalFunc: DataTypes.func,
    optionalNumber: DataTypes.number,
    optionalObject: DataTypes.object,
    optionalString: DataTypes.string,
    optionalSymbol: DataTypes.symbol,
```

```
14.
15.     // 你也可以声明数据为指定类的实例。
16.     // 这里会使用 instanceof 进行判断。
17.     optionalMessage: DataTypes.instanceOf(Message),
18.
19.     // 如果你可以确定你的数据是有限的几个值之一，那么你可以将它声明为枚举类型。
20.     optionalEnum: DataTypes.oneOf(['News', 'Photos']),
21.
22.     // 可以是指定的几个类型之一
23.     optionalUnion: DataTypes.oneOfType([
24.         DataTypes.string,
25.         DataTypes.number,
26.         DataTypes.instanceOf(Message)
27.     ]),
28.
29.     // 数组中每个元素都必须是指定的类型
30.     optionalArrayOf: DataTypes.arrayOf(DataTypes.number),
31.
32.     // 对象的所有属性值都必须是指定的类型
33.     optionalObjectOf: DataTypes.objectOf(DataTypes.number),
34.
35.     // 具有特定形状的对象
36.     optionalObjectWithShape: DataTypes.shape({
37.         color: DataTypes.string,
38.         fontSize: DataTypes.number
39.     }),
40.
41.     // 以上所有校验器都拥有 `isRequired` 方法，来确保此数据必须被提供
42.     requiredFunc: DataTypes.func.isRequired,
43.     requiredObject: DataTypes.shape({
44.         color: DataTypes.string
45.     }).isRequired,
46.
47.     // 一个必须的但可以是任何类型的数据
48.     requiredAny: DataTypes.any.isRequired,
49.
50.     // 你也可指定一个自定义的校验器。
51.     // 如果校验失败，它应该丢出一个异常。
52.     customProp: function (props, propName, componentName) {
53.         if (!/matchme/.test(props[propName])) {
54.             throw new Error(
55.                 'Invalid prop `' + propName + '` supplied to' +
```

```
                ' `' + componentName + '`. Validation failed.'
            );
        }
    },

    // 你也可以给 `arrayOf` 和 `objectOf` 提供一个自定义校验器。
    // 如果校验失败，那么应该当抛出一个异常。
    // 对于数组或者对象中的每个元素都会调用校验器进行校验。
    // 第一个参数是这个数组或者对象，第二个参数是元素的 key。
    customArrayProp: DataTypes.arrayOf(function (dataValue, key, componentName, dataFullName) {
        if (!/matchme/.test(dataValue[key])) {
            throw new Error(
                'Invalid prop `' + dataFullName + '` supplied to' +
                ' `' + componentName + '`. Validation failed.'
            );
        }
    })

});
```

样式

样式处理是编写视图模板时常见的场景，涉及到的 attribute 有 class 和 style，它们的处理方式和其他元素 attribute 有一些区别。本文专门描述样式处理上常见的场景。

在开始前，先强调一下：San 并没有为 class 和 style 处理提供特殊的绑定语法，他们的处理与其它 attribute 方式一样。

class

我们可能会设计一些用于表示状态的 class，这些 class 是否应该被添加到元素上，取决于某些数据的值。一个简单的场景是下拉列表的收起和展开状态切换。

```
<!-- template -->
<div>
    <button on-click="toggle"></button>
    <ul class="list{{isHidden ? ' list-hidden' : ''}}">...</ul>
</div>
```

```
// Component
san.defineComponent({
    toggle: function () {
        var isHidden = this.data.get('isHidden');
        this.data.set('isHidden', !isHidden);
    }
});
```

上面的例子中，isHidden 数据为真时，ul 具有 list-hidden 的 class，为假时不具有。

San 在设计时，希望视图模板开发者像写正常的 attribute 一样编写 class 与 style，所以没有提供特殊的绑定语法。通过三元运算符的支持可以处理这样的场景。

下面例子是一个根据状态不同，切换不同 class 的场景。

```
<ul class="list {{isHidden ? 'list-hidden' : 'list-visible'}}">...</ul>
```

style

对 style 的处理通常没有 class 那么复杂。我们很少会把样式信息写在数据中，但有时我们期望用户能定制一些界面样式，这个时候样式可能来源于数据。

```
<ul>
    <li
        s-for="item, index in datasource"
        style="background: {{item.color}}"
        class="{{item.id == value ? 'selected' : ''}}"
        on-click="itemClick(index)"
    >{{ item.title }}</li>
</ul>
```

此时需要警惕的是，数据可能并不存在，导致你设置的 style 并不是一个合法的样式。如果你不能保证数据一定有值，需要把样式名包含在插值中。

```
<ul>
    <li
        s-for="item, index in datasource"
        style="{{item.color ? 'background:' + item.color : ''}}"
        class="{{item.id == value ? 'selected' : ''}}"
        on-click="itemClick(index)"
    >{{ item.title }}</li>
</ul>
```

条件

s-if

通过 **s-if** 指令，我们可以为元素指定条件。当条件成立时元素可见，当条件不成立时元素不存在。

提示：当不满足条件时，San 会将元素从当前页面中移除，而不是隐藏。

```
<span s-if="isOK">Hello San!</span>
```

s-if 指令的值可以是任何类型的表达式。

```
<span s-if="isReady && isActive">Hello San!</span>
```

提示：San 的条件判断不是严格的 `=== false`。所以，一切 JavaScript 的假值都会认为条件不成立：`0`、空字符串、`null`、`undefined`、`NaN`等。

s-elif

`> 3.2.3`

s-elif 指令可以给 **s-if** 增加一个额外条件分支块。**s-elif** 指令的值可以是任何类型的表达式。

```
<span s-if="isActive">Active</span>
<span s-elif="isOnline">Pending</span>
```

提示：**s-elif** 指令元素必须跟在 **s-if** 或 **s-elif** 指令元素后，否则将抛出 **elif not match if** 的异常。

s-else-if

`> 3.5.6`

s-else-if 指令是 **s-elif** 指令的别名，效果相同。

```
<span s-if="isActive">Active</span>
<span s-else-if="isOnline">Pending</span>
```

s-else

s-else 指令可以给 s-if 增加一个不满足条件的分支块。s-else 指令没有值。

```
<span s-if="isOnline">Hello!</span>
<span s-else>Offline</span>
```

提示 **：s-else 指令元素必须跟在 s-if 或 s-elif 指令元素后，否则将抛出 else not match if 的异常。**

虚拟元素

在 san 中，template 元素在渲染时不会包含自身，只会渲染其内容。对 template 元素应用 if 指令能够让多个元素同时根据条件渲染视图，可以省掉一层不必要的父元素。

```
<div>
    <template s-if="num > 10000">
        <h2>biiig</h2>
        <p>{{num}}</p>
    </template>
    <template s-elif="num > 1000">
        <h3>biig</h3>
        <p>{{num}}</p>
    </template>
    <template s-elif="num > 100">
        <h4>big</h4>
        <p>{{num}}</p>
    </template>
    <template s-else>
        <h5>small</h5>
        <p>{{num}}</p>
    </template>
</div>
```

循环

通过循环渲染列表是常见的场景。通过在元素上作用 **s-for** 指令，我们可以渲染一个列表。

语法

s-for 指令的语法形式如下:

```
item-identifier[, index-identifier] in expression[ trackBy accessor-expression]
```

列表渲染

下面的代码描述了在元素上作用 **s-for** 指令，渲染一个列表。在列表渲染的元素内部，可以正常访问到 owner 组件上的其他数据（下面例子中的dept）。

```
<!-- Template -->
<dl>
    <dt>name - email</dt>
    <dd s-for="p in persons" title="{{p.name}}">{{p.name}}({{dept}}) - {{p.email}}</dd>
</dl>
```

```
// Component
san.defineComponent({
    // template

    initData: function () {
        return {
            dept: 'ssg',
            persons: [
                {name: 'errorrik', email: 'errorrik@gmail.com'},
                {name: 'otakustay', email: 'otakustay@gmail.com'}
            ]
        };
    }
});
```

索引

s-for 指令中可以指定索引变量名（下面例子中的index），从而在列表渲染过程获得列表元素的索引。

```
<!-- Template -->
<dl>
    <dt>name - email</dt>
    <dd s-for="p, index in persons" title="{{p.name}}">{{index + 1}}.
{{p.name}}({{dept}}) - {{p.email}}</dd>
</dl>
```

```
// Component
san.defineComponent({
    // template

    initData: function () {
        return {
            dept: 'ssg',
            persons: [
                {name: 'errorrik', email: 'errorrik@gmail.com'},
                {name: 'otakustay', email: 'otakustay@gmail.com'}
            ]
        };
    }
});
```

列表数据操作

列表数据的增加、删除等操作请使用组件 data 提供的数组方法。详细请参考数组方法文档。

虚拟元素

和 if 指令一样，对 template 元素应用 for 指令，能够让多个元素同时根据遍历渲染，可以省掉一层不必要的父元素。

```
<!-- Template -->
<dl>
    <template s-for="p in persons">
        <dt>{{p.name}}</dt>
        <dd>{{p.email}}</dd>
    </template>
```

```
</dl>
```

trackBy

`>= 3.6.1`

在 **s-for** 指令声明中指定 **trackBy**，San 在数组更新时，将自动跟踪项的变更，进行相应的 insert/remove 等操作。 **trackBy** 只能声明 item-identifier 的属性访问。

```
<!-- Template -->
<dl>
    <dt>name - email</dt>
    <dd s-for="p in persons trackBy p.name" title="{{p.name}}">{{p.name}} ({{dept}}) - {{p.email}}</dd>
</dl>
```

```
// Component
san.defineComponent({
    // template

    initData: function () {
        return {
            dept: 'ssg',
            persons: [
                {name: 'errorrik', email: 'errorrik@gmail.com'},
                {name: 'otakustay', email: 'otakustay@gmail.com'}
            ]
        };
    }
});
```

trackBy 通常用在对后端返回的 JSON 数据渲染时。因为前后两次 JSON parse 无法对列表元素进行 === 比对，通过 trackBy， 框架将在内部跟踪变更。结合 transition 时，变更过程的动画效果将更符合常理。

在下面的场景下，使用 trackBy 的性能上反而会变差：

- 所有数据项都发生变化
- 数据项变化前后的顺序不同

事件处理

事件是开发中最常用的行为管理方式。通过 **on-** 前缀，可以将事件的处理绑定到组件的方法上。

提示：在 San 中，无论是 DOM 事件还是组件的自定义事件，都通过 **on-** 前缀绑定，没有语法区分。

DOM 事件

on- + 事件名 将 DOM 元素的事件绑定到组件方法上。当 DOM 事件触发时，组件方法将被调用，this 指向组件实例。下面的例子中，当按钮被点击时，组件的 submit 方法被调用。

```
san.defineComponent({
    template: '...<button type="button" on-click="submit">submit</button>',

    submit: function () {
        var title = this.data.get('title');
        if (!title) {
            return;
        }

        sendData({title: title});
    }
});
```

绑定事件时，可以指定参数，引用当前渲染环境中的数据。参数可以是任何类型的表达式。

```
<!-- Template -->
<ul>
    <li s-for="item, index in todos">
        <h3>{{ item.title }}</h3>
        <p>{{ item.desc }}</p>
        <i class="fa fa-trash-o" on-click="rmTodo(item)"></i>
    </li>
</ul>
```

```
// Component
san.defineComponent({
    rmTodo: function (todo) {
        service.rmTodo(todo.id);
```

```
        this.data.remove('todos', todo);
    }
});
```

指定参数时，**$event** 是 San 保留的一个特殊变量，指定 $event 将引用到 DOM Event 对象。从而你可以拿到事件触发的 DOM 对象、鼠标事件的鼠标位置等事件信息。

```
san.defineComponent({
    template: '<button type="button" on-click="clicker($event)">click here</button>',

    clicker: function (e) {
        alert(e.target.tagName); // BUTTON
    }
});
```

自定义事件

在组件上通过 **on-** 前缀，可以绑定组件的自定义事件。

下面的例子中，MyComponent 为 Label 组件绑定了 done 事件的处理方法。

```
var MyComponent = san.defineComponent({
    components: {
        'ui-label': Label
    },

    template: '<div><ui-label bind-text="name" on-done="labelDone($event)"></ui-label></div>',

    labelDone: function (doneMsg) {
        alert(doneMsg);
    }
});
```

San 的组件体系提供了事件功能，Label 直接通过调用 fire 方法就能方便地派发一个事件。

```
var Label = san.defineComponent({
    template: '<template class="ui-label" title="{{text}}">{{text}}</template>',

```

```
    attached: function () {
        this.fire('done', this.data.get('text') + ' done');
    }
});
```

修饰符

capture

版本 : >= 3.3.0

在元素的事件声明中使用 capture 修饰符，事件将被绑定到捕获阶段。

```
var MyComponent = san.defineComponent({
    template: ''
        + '<div on-click="capture:mainClick">'
            + '<button on-click="capture:btnClick">click</button>'
        + '</div>',

    mainClick: function (title) {
        alert('Main');
    },

    btnClick: function (title) {
        alert('Button');
    }
});
```

注意 : 只有在支持 **addEventListener** 的浏览器环境支持此功能，老旧 IE 上使用 capture 修饰符将没有效果。

native

版本 : >= 3.3.0

在组件的事件声明中使用 native 修饰符，事件将被绑定到组件根元素的 DOM 事件。

```
var Button = san.defineComponent({
    template: '<a class="my-button"><slot/></a>'
});

```

```
var MyComponent = san.defineComponent({
    components: {
        'ui-button': Button
    },

    template: '<div><ui-button on-click="native:clicker(title)">{{title}}</ui-button></div>',

    clicker: function (title) {
        alert(title);
    }
});
```

有时候组件封装了一些基础结构和样式，同时希望点击、触摸等 DOM 事件由外部使用方处理。如果组件需要 fire 每个根元素 DOM 事件是很麻烦并且难以维护的。native 修饰符解决了这个问题。

表单

表单是常见的用户输入承载元素，本篇介绍一些常用表单元素的用法。在 MVVM 中，我们一般在用户输入的表单元素或组件上应用 双向绑定。

输入框

输入框的绑定方法比较简单，直接对输入框的 value 属性应用双向绑定就行了。

```
<input type="text" value="{= name =}">
```

checkbox

checkbox 常见的使用场景是分组，在组件模板中，我们把需要分组的 checkbox 将 checked 属性双向绑定到同名的组件数据中。

提示 ：除非你需要进行传统的表单提交，否则无需指定 checkbox 的 name 属性。San 仅以 checked 作为分组的依据。

```
<!-- Template -->
<div>
    <label><input type="checkbox" value="errorrik" checked="{= online =}">errorrik</label>
    <label><input type="checkbox" value="otakustay" checked="{= online =}">otakustay</label>
    <label><input type="checkbox" value="firede" checked="{= online =}">firede</label>
</div>
```

我们期望 checkbox 绑定到的数据项是一个 **Array<string>** 。当 checkbox 被选中时，其 value 会被添加到绑定的数据项中；当 checkbox 被取消选中时，其 value 会从绑定数据项中移除。

```
// Component
san.defineComponent({
    // ...

    initData: function () {
        return {
            online: []
```

```
        };
    },

    attached: function () {
        this.data.set('online', ['errorrik', 'otakustay']);
    }
});
```

radio

与 checkbox 类似，我们在组件模板中，把需要分组的 radio 将 checked 属性绑定到同名的组件数据中。

提示：你需要手工指定分组 radio 的 name 属性，使浏览器能处理 radio 选择的互斥。可以把它设置成与绑定数据的名称相同。

```
<!-- Template -->
<div>
    <label><input type="radio" value="errorrik" checked="{= online =}" name="online">errorrik</label>
    <label><input type="radio" value="otakustay" checked="{= online =}" name="online">otakustay</label>
    <label><input type="radio" value="firede" checked="{= online =}" name="online">firede</label>
</div>
```

我们期望 radio 绑定到的数据项是一个 **string** 。当 radio 被选中时，绑定的数据项值被设置成选中的 radio 的 value 属性值。

```
// Component
san.defineComponent({
    // ...

    initData: function () {
        return {
            online: 'errorrik'
        };
    }
});
```

select

select 的使用方式和输入框类似，直接对 value 属性应用双向绑定。

```
<!-- Template -->
<select value="{= online =}">
    <option value="errorrik">errorrik</option>
    <option value="otakustay">otakustay</option>
    <option value="firede">firede</option>
</select>
```

提示：在浏览器中，select 的 value 属性并不控制其选中项，select 的选中项是由 option 的 selected 属性控制的。考虑到开发的方便，开发者不需要编写 option 的 selected 属性，San 会在下一个视图更新时间片中刷新 select 的选中状态。

```
// Component
san.defineComponent({
    // ...

    initData: function () {
        return {
            online: 'errorrik'
        };
    }
});
```

插槽

在视图模板中可以通过 slot 声明一个插槽的位置，其位置的内容可以由外层组件定义。

```
var Panel = san.defineComponent({
    template: '<div>'
        + '  <div class="head" on-click="toggle">title</div>'
        + '  <p style="{{fold | yesToBe(\'display:none\')}}"><slot></slot></p>'
        + '</div>',

    toggle: function () {
        this.data.set('fold', !this.data.get('fold'));
    }
});

var MyComponent = san.defineComponent({
    components: {
        'ui-panel': Panel
    },

    template: '<div><ui-panel>Hello San</ui-panel></div>'
});

/* MyComponent渲染结果
<div>
  <div class="head">title</div>
  <p style="display:none">Hello San</p>
</div>
*/
```

HTML 标准关于 slot 的描述可以参考 这里

数据环境

插入 slot 部分的内容，其数据环境为 声明时的环境。

```
var Panel = san.defineComponent({
    template: '<div>'
        + '  <div class="head" on-click="toggle">title</div>'
        + '  <p><slot></slot></p>'
        + '</div>',
```

```
    initData: function () {
        return {name: 'Panel'};
    }
});

var MyComponent = san.defineComponent({
    components: {
        'ui-panel': Panel
    },

    template: '<div><ui-panel>I am {{name}}</ui-panel></div>',

    initData: function () {
        return {name: 'MyComponent'};
    }
});

/* MyComponent渲染结果
<div>
  <div class="head">title</div>
  <p>I am MyComponent</p>
</div>
*/
```

命名

通过 name 属性可以给 slot 命名。一个视图模板的声明可以包含一个默认 slot 和多个命名 slot。外层组件的元素通过 `slot="name"` 的属性声明，可以指定自身的插入点。

```
var Tab = san.defineComponent({
    template: '<div>'
        + '  <header><slot name="title"></slot></header>'
        + '  <main><slot></slot></main>'
        + '</div>'
});

var MyComponent = san.defineComponent({
    components: {
        'ui-tab': Tab
    },
```

```
12.
13.     template: '<div><ui-tab>'
14.         + '<h3 slot="title">1</h3><p>one</p>'
15.         + '<h3 slot="title">2</h3><p>two</p>'
16.         + '</ui-tab></div>'
17. });
18.
19. /* MyComponent渲染结果
20. <div>
21.   <header><h3>1</h3><h3>2</h3></header>
22.   <main><p>one</p><p>two</p></main>
23. </div>
24. */
```

`注意`：外层组件的替换元素，只有在直接子元素上才能声明 `slot="name"` 指定自身的插入点。下面例子中的 a 元素无法被插入 title slot。

```
1.  var Tab = san.defineComponent({
2.      template: '<div>'
3.          + '  <header><slot name="title"></slot></header>'
4.          + '  <main><slot></slot></main>'
5.          + '</div>'
6.  });
7.
8.  var MyComponent = san.defineComponent({
9.      components: {
10.         'ui-tab': Tab
11.     },
12.
13.     template: '<div><ui-tab>'
14.         + '<h3 slot="title">1</h3><p>one</p>'
15.         + '<h3 slot="title">2</h3><p>two<a slot="title">slot fail</a></p>'
16.         + '</ui-tab></div>'
17. });
18.
19. /* MyComponent渲染结果, a 元素无法被插入 title slot
20. <div>
21.   <header><h3>1</h3><h3>2</h3></header>
22.   <main><p>one</p><p>two<a>slot fail</a></p></main>
23. </div>
24. */
```

插槽指令应用

版本 : >= 3.3.0

在 slot 声明时应用 if 或 for 指令，可以让插槽根据组件数据动态化。

if指令

下面的例子中，panel 的 hidden 属性为 true 时，panel 中默认插槽将不会渲染，仅包含 title 插槽，通过 slot 方法获取的数组长度为 0。

```
var Panel = san.defineComponent({
    template: '<div><slot name="title"/><slot s-if="!hidden"/></div>',
});

var MyComponent = san.defineComponent({
    components: {
      'x-panel': Panel
    },

    template: ''
        + '<div>'
          + '<x-panel hidden="{{folderHidden}}" s-ref="panel">'
              + '<b slot="title">{{name}}</b>'
              + '<p>{{desc}}</p>'
          + '</x-panel>'
        + '</div>',

    attached: function () {
        // 0
        this.ref('panel').slot().length
    }
});


var myComponent = new MyComponent({
    data: {
        folderHidden: true,
        desc: 'MVVM component framework',
        name: 'San'
    }
});
```

```
32.
33. /* MyComponent渲染结果，hidden为true所以不包含default slot
34. <div>
35.     <b>San</b>
36. </div>
37. */
```

for指令

下面的例子没什么用，纯粹为了演示 slot 上应用 for 指令。在后续 **scoped** 插槽 章节中可以看到有意义的场景。

```
1. var Panel = san.defineComponent({
2.     template: '<div><slot s-for="item in data"/></div>',
3. });
4.
5. var MyComponent = san.defineComponent({
6.     components: {
7.       'x-panel': Panel
8.     },
9.
10.     template: ''
11.         + '<div>'
12.           + '<x-panel data="{{panelData}}" s-ref="panel">'
13.               + '<p>{{name}}</p>'
14.           + '</x-panel>'
15.         + '</div>',
16.
17.     attached: function () {
18.         // 0
19.         this.ref('panel').slot().length
20.     }
21. });
22.
23.
24. var myComponent = new MyComponent({
25.     data: {
26.         panelData: [1, 2, 3],
27.         name: 'San'
28.     }
29. });
30.
```

```
/* MyComponent渲染结果, <p>{{name}}</p>输出 3 遍
<div>
    <p>San</p>
    <p>San</p>
    <p>San</p>
</div>
*/
```

scoped 插槽

如果 slot 声明中包含 **s-bind** 或 1 个以上 **var-** 数据前缀声明，该 slot 为 scoped slot。scoped slot 具有独立的 数据环境。

scoped slot 通常用于组件的视图部分期望由 外部传入视图结构，渲染过程使用组件内部数据。

注意 : scoped slot 中不支持双向绑定。

var

版本 : >= 3.3.0

var- 的 scoped 数据声明的形式为 **var-name=”expression”**。

```
var Men = san.defineComponent({
    template: '<div>'
      + '<slot s-for="item in data" var-n="item.name" var-email="item.email" var-sex="item.sex ? \'male\' : \'female\'">'
        + '<p>{{n}},{{sex}},{{email}}</p>'
      + '</slot>'
      + '</div>'
});

var MyComponent = san.defineComponent({
    components: {
        'x-men': Men
    },

    template: '<div><x-men data="{{men}}" s-ref="men">'
          + '<h3>{{n}}</h3>'
          + '<p><b>{{sex}}</b><u>{{email}}</u></p>'
        + '</x-men></div>',

```

```
    attached: function () {
        var slots = this.ref('men').slot();

        // 3
        slots.length

        // truthy
        slots[0].isInserted

        // truthy
        contentSlot.isScoped
    }
});

var myComponent = new MyComponent({
    data: {
        men: [
            {name: 'errorrik', sex: 1, email: 'errorrik@gmail.com'},
            {name: 'leeight', sex: 0, email: 'leeight@gmail.com'},
            {name: 'otakustay', email: 'otakustay@gmail.com', sex: 1}
        ]
    }
});

/* MyComponent渲染结果
<div>
    <h3>errorrik</h3>
    <p><b>male</b><u>errorrik@gmail.com</u></p>
    <h3>leeight</h3>
    <p><b>female</b><u>leeight@gmail.com</u></p>
    <h3>otakustay</h3>
    <p><b>male</b><u>otakustay@gmail.com</u></p>
</div>
*/
```

s-bind

版本 : >= 3.6.0

s-bind 的 scoped 数据声明的形式为 **s-bind="expression"**。

当 **s-bind** 和 **var-** 并存时，**var-** 将覆盖整体绑定中相应的数据项。

```
var Men = san.defineComponent({
    template: '<div>'
      + '<slot s-for="item in data" s-bind="{n: item.name, email: item.email, sex: item.sex ? \'male\' : \'female\'}">'
        + '<p>{{n}},{{sex}},{{email}}</p>'
      + '</slot>'
      + '</div>'
});

var MyComponent = san.defineComponent({
    components: {
        'x-men': Men
    },

    template: '<div><x-men data="{{men}}" s-ref="men">'
          + '<h3>{{n}}</h3>'
          + '<p><b>{{sex}}</b><u>{{email}}</u></p>'
        + '</x-men></div>',

    attached: function () {
        var slots = this.ref('men').slot();

        // 3
        slots.length

        // truthy
        slots[0].isInserted

        // truthy
        contentSlot.isScoped
    }
});

var myComponent = new MyComponent({
    data: {
        men: [
            {name: 'errorrik', sex: 1, email: 'errorrik@gmail.com'},
            {name: 'leeight', sex: 0, email: 'leeight@gmail.com'},
            {name: 'otakustay', email: 'otakustay@gmail.com', sex: 1}
        ]
    }
});
```

```
42.
43. /* MyComponent渲染结果
44. <div>
45.     <h3>errorrik</h3>
46.     <p><b>male</b><u>errorrik@gmail.com</u></p>
47.     <h3>leeight</h3>
48.     <p><b>female</b><u>leeight@gmail.com</u></p>
49.     <h3>otakustay</h3>
50.     <p><b>male</b><u>otakustay@gmail.com</u></p>
51. </div>
52. */
```

访问环境数据

版本 : >= 3.3.1

scoped slot 中，除了可以访问 **var-** 声明的数据外，还可以访问当前环境的数据。

- 使用 slot 默认内容时，可以访问组件内部环境数据
- 外层组件定义的 slot 内容，可以访问外层组件环境的数据

```
1. var Man = san.defineComponent({
2.     template: '<p>'
3.       +   '<slot var-n="who.name" var-email="who.email">'
4.       +     '{{n}},{{email}},{{country}}'
5.       +   '</slot>'
6.       + '</p>'
7. });
8.
9. var MyComponent = san.defineComponent({
10.     components: {
11.         'x-man': Man
12.     },
13.
14.     template: ''
15.         + '<div><x-man who="{{man}}" country="{{country}}">'
16.         +   '<b>{{n}} - {{province}}</b>'
17.         +   '<u>{{email}}</u>'
18.         + '</x-men></div>'
19. });
20.
21. var myComponent = new MyComponent({
```

```
    data: {
        man: {
            name: 'errorrik',
            email: 'errorrik@gmail.com'
        },
        country: 'China',
        province: 'HN'
    }
});

/* MyComponent渲染结果
<div>
    <p>
        <b>errorrik - HN</b>
        <u>errorrik@gmail.com</u>
    </p>
</div>
*/
```

动态命名

版本 : >= 3.3.1

slot 声明中，组件可以使用当前的数据环境进行命名，从而提供动态的插槽。插槽的动态命名常用于组件结构根据数据生成 的场景下，比如表格组件。

```
var Table = san.defineComponent({
    template: ''
        + '<table>'
        +   '<thead><tr><th s-for="col in columns">{{col.label}}</th></tr></thead>'
        +   '<tbody>'
        +     '<tr s-for="row in datasource">'
        +       '<td s-for="col in columns">'
        +         '<slot name="col-{{col.name}}" var-row="row" var-col="col">{{row[col.name]}}</slot>'
        +       '</td>'
        + '    </tr>'
        +   '</tbody>'
        + '</table>'
});
```

```
14.
15. var MyComponent = san.defineComponent({
16.     components: {
17.         'x-table': Table
18.     },
19.
20.     template: ''
21.         + '<div>'
22.         +   '<x-table columns="{{columns}}" datasource="{{list}}">'
23.         +     '<b slot="col-name">{{row.name}}</b>'
24.         +   '</x-table>'
25.         + '</div>'
26.
27. });
28.
29. var myComponent = new MyComponent({
30.     data: {
31.         columns: [
32.             {name: 'name', label: '名'},
33.             {name: 'email', label: '邮'}
34.         ],
35.         list: [
36.             {name: 'errorrik', email: 'errorrik@gmail.com'},
37.             {name: 'leeight', email: 'leeight@gmail.com'}
38.         ]
39.     }
40. });
41.
42. /* MyComponent渲染结果
43. <div>
44.     <table>
45.         <thead>
46.             <tr>
47.                 <th>名</th>
48.                 <th>邮</th>
49.             </tr>
50.         </thead>
51.         <tbody>
52.             <tr>
53.                 <td><b>errorrik</b></td>
54.                 <td>errorrik@gmail.com</td>
55.             </tr>
```

```
            <tr>
                <td><b>leeight</b></td>
                <td>leeight@gmail.com</td>
            </tr>
        </tbody>
    </table>
</div>
*/
```

注意：表格的视图更新在 IE 下可能存在兼容性问题。

过渡

在视图中，过渡效果是常见的场景。平滑的过渡动画能够给用户更好的感官体验。san 提供了基础的过渡机制，你可以基于此开发丰富的过渡效果。

版本 : >= 3.3.0

s-transition

在元素上通过 **s-transition** 指令，可以声明过渡动画控制器。

```
<button s-transition="opacityTransition">click</button>
```

这个对象是元素 owner 的成员。

```
san.defineComponent({
    template: '<div><button s-transition="opacityTransition">click</button></div>',

    opacityTransition: {
        // 过渡动画控制器的结构在下文中描述
        // ...
    }
});
```

我们通常把 **s-transition** 和条件或循环指令一起使用。

```
<button s-transition="opacityTransition" s-if="allowEdit">Edit</button>
<b s-transition="opacityTransition" s-else>Edit not allow</b>
```

s-transition 声明的过渡动画控制器可以是 owner 组件的深层成员。

```
san.defineComponent({
    template: '<div><button s-transition="trans.opacity">click</button></div>',

    trans: {
        opacity: {
            // 过渡动画控制器的结构在下文中描述
            // ...
        }
    }
```

```
});
```

注意 **：s-transition** 只能应用在具体的元素中。template 这种没有具体元素的标签上应用 **s-transition** 将没有效果。

动画控制器

过渡动画控制器是一个包含 **enter** 和 **leave** 方法的对象。

enter 和 **leave** 方法的签名为 **function({HTMLElement}el, {Function}done)**。san 会把要过渡的元素传给过渡动画控制器，控制器在完成动画后调用 **done** 回调函数。

```
san.defineComponent({
    template: `
        <div>
            <button on-click="toggle">toggle</button>
            <button s-if="isShow" s-transition="opacityTrans">Hello San!
</button>
            <button s-else s-transition="opacityTrans">Hello ER!</button>
        </div>
    `,

    toggle: function () {
        this.data.set('isShow', !this.data.get('isShow'));
    },

    opacityTrans: {
        enter: function (el, done) {
            var steps = 20;
            var currentStep = 0;

            function goStep() {
                if (currentStep >= steps) {
                    el.style.opacity = 1;
                    done();
                    return;
                }

                el.style.opacity = 1 / steps * currentStep++;
                requestAnimationFrame(goStep);
            }

```

```
            goStep();
        },

        leave: function (el, done) {
            var steps = 20;
            var currentStep = 0;

            function goStep() {
                if (currentStep >= steps) {
                    el.style.opacity = 0;
                    done();
                    return;
                }

                el.style.opacity = 1 - 1 / steps * currentStep++;
                requestAnimationFrame(goStep);
            }

            goStep();
        }
    }
});
```

提示：

san 把动画控制器留给应用方实现，框架本身不内置动画控制效果。应用方可以：

- 使用 css 动画，在 transitionend 或 animationend 事件监听中回调 done
- 使用 requestAnimationFrame 控制动画，完成后回调 done
- 在老旧浏览器使用 setTimeout / setInterval 控制动画，完成后回调 done
- 发挥想象力

动画控制器 Creator

s-transition 指令声明对应的对象如果是一个 function，san 将把它当成 过渡动画控制器 **Creator**。

每次触发过渡动画前，san 会调用过渡动画控制器 **Creator**，用其返回的对象作为过渡动画控制器。

```
san.defineComponent({
    template: `
        <div>
```

```
            <button on-click="toggle">toggle</button>
            <button s-if="isShow" s-transition="opacityTrans">Hello San!
</button>
            <button s-else s-transition="opacityTrans">Hello ER!</button>
        </div>
    `,

    toggle: function () {
        this.data.set('isShow', !this.data.get('isShow'));
    },

    opacityTrans: function () {
        return {
            enter: function (el, done) {
                var steps = 20;
                var currentStep = 0;

                function goStep() {
                    if (currentStep >= steps) {
                        el.style.opacity = 1;
                        done();
                        return;
                    }

                    el.style.opacity = 1 / steps * currentStep++;
                    requestAnimationFrame(goStep);
                }

                goStep();
            },

            leave: function (el, done) {
                var steps = 20;
                var currentStep = 0;

                function goStep() {
                    if (currentStep >= steps) {
                        el.style.opacity = 0;
                        done();
                        return;
                    }

```

```
                    el.style.opacity = 1 - 1 / steps * currentStep++;
                    requestAnimationFrame(goStep);
                }

                goStep();
            }
        }
    }
});
```

和事件声明类似，过渡动画控制器 **Creator**调用支持传入参数。

```
san.defineComponent({
    template: `
        <div>
            <button on-click="toggle">toggle</button>
            <button on-click="toggleTrans">toggle transition</button>
            <button s-if="isShow" s-transition="opacityTrans(noTransition)">Hello San!</button>
            <button s-else s-transition="opacityTrans(noTransition)">Hello ER!</button>
        </div>
    `,

    toggle: function () {
        this.data.set('isShow', !this.data.get('isShow'));
    },

    toggleTrans: function () {
        this.data.set('noTransition', !this.data.get('noTransition'));
    },

    initData: function () {
        return {
            noTransition: false
        };
    },

    opacityTrans: function (disabled) {
        return {
            enter: function (el, done) {
                if (disabled) {
```

```
                    done();
                    return;
                }

                var steps = 20;
                var currentStep = 0;

                function goStep() {
                    if (currentStep >= steps) {
                        el.style.opacity = 1;
                        done();
                        return;
                    }

                    el.style.opacity = 1 / steps * currentStep++;
                    requestAnimationFrame(goStep);
                }

                goStep();
            },

            leave: function (el, done) {
                if (disabled) {
                    done();
                    return;
                }

                var steps = 20;
                var currentStep = 0;

                function goStep() {
                    if (currentStep >= steps) {
                        el.style.opacity = 0;
                        done();
                        return;
                    }

                    el.style.opacity = 1 - 1 / steps * currentStep++;
                    requestAnimationFrame(goStep);
                }

                goStep();
```

```
71.             }
72.         }
73.     }
74. });
```

组件

组件是 San 的基本单位，是独立的数据、逻辑、视图的封装单元。从页面的角度看，组件是 HTML 元素的扩展。从功能模式的角度看，组件是一个 ViewModel。

组件定义

定义组件最基本的方法是，从 **san.Component** 继承。San 提供了 **san.inherits** 方法，用于继承。

```
function MyApp(options) {
    san.Component.call(this, options);
}
san.inherits(MyApp, san.Component);

MyApp.prototype.template = '<ul><li s-for="item in list">{{item}}</li></ul>';

MyApp.prototype.attached = function () {
    this.data.set('list', ['san', 'er', 'esui', 'etpl', 'esl']);
};
```

然后，通过 new 的方式就可以使用这个组件了。当然，通常你可能希望让组件出现在页面上，所以需要调用 attach 方法，将组件添加到页面的相应位置。

```
var myApp = new MyApp();
myApp.attach(document.body);
```

通过继承的方式定义组件的好处是，当你使用 ESNext 时，你可以很自然的 extends。

注意 ：由于 ESNext 没有能够编写 prototype 属性的语法，所以 San 对组件定义时的属性支持 static property，通过 ESNext 的 extends 继承时，template / filters / components 属性请使用 static property 的方式定义。

```
import {Component} from 'san';

class HelloComponent extends Component {

    constructor(options) {
        super(options);
        // .....
```

```
    }

    static template = '<p>Hello {{name}}!</p>';

    initData() {
        return {name: 'San'}
    }
}

new HelloComponent().attach(document.body);
```

对于不使用 ESNext 时，写一个 function 然后调用 **san.inherits** 再写各种 prototype 实在是有点麻烦，San 提供了快捷方法 **san.defineComponent** 用于方便地定义组件。

```
var MyApp = san.defineComponent({
    template: '<ul><li s-for="item in list">{{item}}</li></ul>',

    attached: function () {
        this.data.set('list', ['san', 'er', 'esui', 'etpl', 'esl']);
    }
});
```

生命周期

San 的组件是 HTML 元素扩展的风格，所以其生命周期与 WebComponents 相符合。

- `compiled` - 组件视图模板编译完成
- `inited` - 组件实例初始化完成
- `created` - 组件元素创建完成
- `attached` - 组件已被附加到页面中
- `detached` - 组件从页面中移除
- `disposed` - 组件卸载完成

组件的生命周期有这样的一些特点：

- 生命周期代表组件的状态，生命周期本质就是状态管理。
- 在生命周期到达时，对应的钩子函数会被触发运行。
- 并存。比如 attached 和 created 等状态是同时并存的。
- 互斥。attached 和 detached 是互斥的，disposed 会互斥掉其它所有的状态。
- 有的时间点并不代表组件状态，只代表某个行为。当行为完成时，钩子函数也会触发。如 **updated** 代表每次数据变化导致的视图变更完成。

通过声明周期的钩子函数，我们可以在生命周期到达时做一些事情。比如在生命周期 **attached** 中发起获取数据的请求，在请求返回后更新数据，使视图刷新。

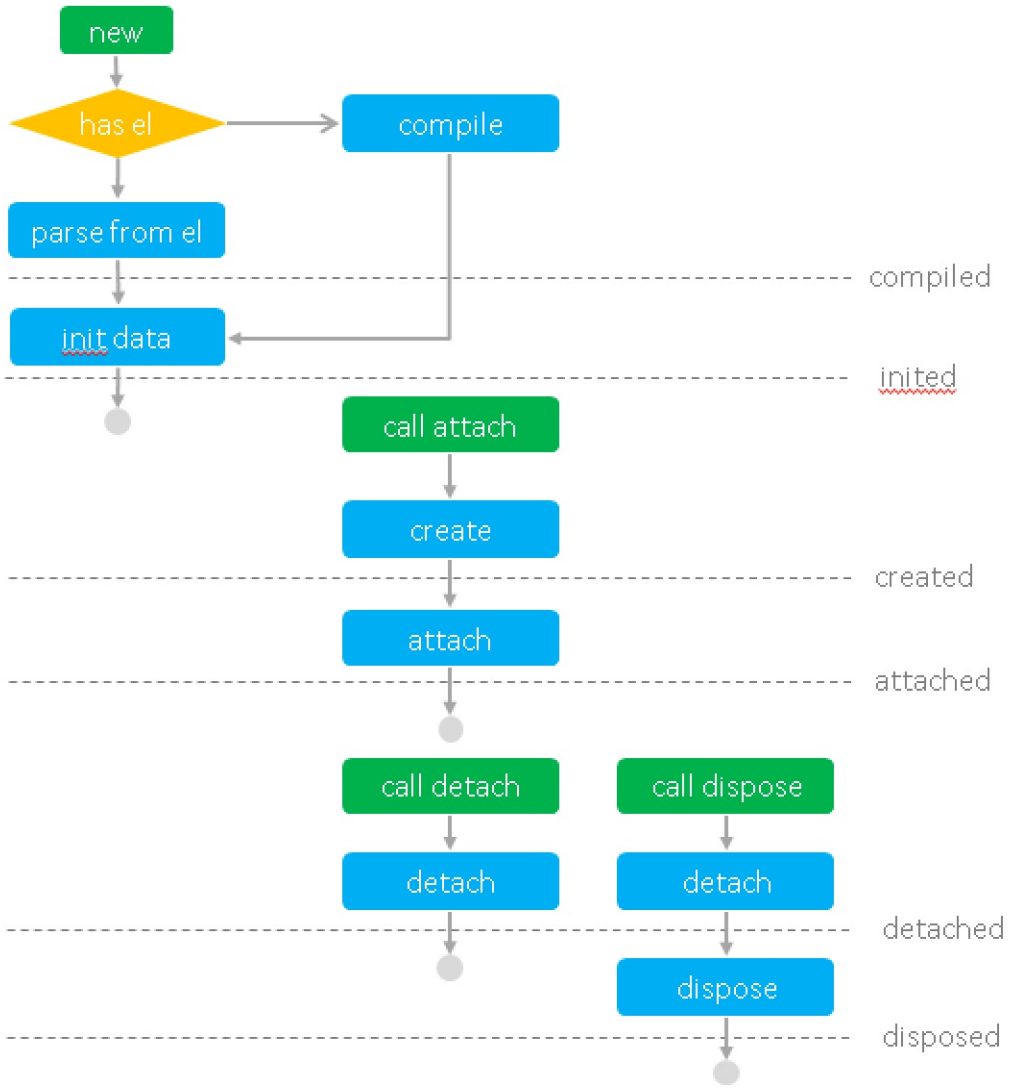
```
var ListComponent = san.defineComponent({
    template: '<ul><li s-for="item in list">{{item}}</li></ul>',

    initData: function () {
        return {
            list: []
        };
    },

    attached: function () {
        requestList().then(this.updateList.bind(this));
    },

    updateList: function (list) {
        this.data.set('list', list);
    }
});
```

下图详细描述了组件的生存过程：



视图

视图模板

定义组件时，通过 `template` 可以定义组件的视图模板。

```
1.  san.defineComponent({
2.      template: '<div>'
3.          + '<label><input type="checkbox" value="errorrik" checked="{= online
    =}">errorrik</label>'
4.          + '<label><input type="checkbox" value="otakustay" checked="{= online
    =}">otakustay</label>'
5.          + '<label><input type="checkbox" value="firede" checked="{= online
    =}">firede</label>'
6.          + '</div>',
7.
```

```
    initData: function () {
        return {
            online: ['errorrik', 'otakustay']
        };
    }
});
```

通常，将 HTML 片段写在 JavaScript 中是不友好的，我们可以把模板写在单独的文件里，通过工具或装载器去管理。

在 webpack + ESNext 环境下引用模板：

```
待补充
```

在 AMD 环境下通过 text plugin 引用模板：

```
san.defineComponent({
    template: require('text!./template.html'),

    initData: function () {
        return {
            online: ['errorrik', 'otakustay']
        };
    }
});
```

强调：San 要求组件对应 一个 HTML 元素，所以视图模板定义时，只能包含一个 HTML 元素，其它所有内容需要放在这个元素下。

```
<dl>
    <dt>name - email</dt>
    <dd s-for="p in persons" title="{{p.name}}">{{p.name}}({{dept}}) - {{p.email}}</dd>
</dl>
```

组件对应的 HTML 元素可能是由其 **owner** 组件通过视图模板指定的，视图模板不好直接定死对应 HTML 元素的标签。此时可以将视图模板对应的 HTML 元素指定为 **template**。

```
<template class="ui-timepicker">{{ value | valueText }}</template>
```

插槽

在视图模板中可以通过 slot 声明一个插槽的位置，其位置的内容可以由外层组件定义。具体请参考插槽文档。

```
var Panel = san.defineComponent({
    template: '<div>'
        + '  <div class="head" on-click="toggle">title</div>'
        + '  <p style="{{fold | yesToBe(\'display:none\')}}"><slot></slot></p>'
        + '</div>',

    toggle: function () {
        this.data.set('fold', !this.data.get('fold'));
    }
});

var MyComponent = san.defineComponent({
    components: {
        'ui-panel': Panel
    },

    template: '<div><ui-panel>Hello San</ui-panel></div>'
});

/* MyComponent渲染结果
<div>
  <div class="head">title</div>
  <p style="display:none">Hello San</p>
</div>
*/
```

el

组件实例的属性 **el** 表示组件对应的 HTML 元素，组件初始化时可以通过 option 传入。

基本上在编写组件时不需要关心它，但是在初始化组件时如果传入 **el**，意味着让组件以此元素作为组件根元素。元素将：

- 不会使用预设的 template 渲染视图
- 不会创建根元素
- 直接到达 compiled、created、attached 生命周期

有时我们为了首屏时间，期望初始的视图是直接的 HTML，不希望初始视图是由组件渲染的。但是我们希望组件为我们管理数据、逻辑与视图，后续的用户交互行为与视图变换通过组件管理，此时就可以通

过 **el** 传入一个现有元素。

组件将以传入的 **el** 元素作为组件根元素并反解析出视图结构。这个过程我们称作 组件反解。详细请参考组件反解文档。

数据

所有组件数据相关的操作，都由组件实例的 **data** 成员提供。

获取数据

通过 **data.get** 方法可以获取数据。

```
1.  san.defineComponent({
2.      attached: function () {
3.          var params = this.data.get('params');
4.          this.data.set('list', getList(params[1]));
5.      }
6.  });
```

data.get 方法接受一个表示 property accessor 的字符串，所以上面的例子也可以写成这样：

```
1.  san.defineComponent({
2.      attached: function () {
3.          var param = this.data.get('params[1]');
4.          this.data.set('list', getList(param));
5.      }
6.  });
```

操作数据

data 上提供了一些数据操作的方法，具体请参考数据操作文档。

初始数据

组件在实例化时可以通过 option 传入 **data**，指定组件初始化时的数据。

```
1.  var MyApp = san.defineComponent({
2.      template: '<ul><li s-for="item in list">{{item}}</li></ul>'
3.  });
4.
```

```
var myApp = new MyApp({
    data: {
        list: ['san', 'er', 'esui', 'etpl', 'esl']
    }
});
myApp.attach(document.body);
```

new 时传入初始数据是针对实例的特例需求。有时我们在定义组件时希望每个实例都具有初始的一些数据，此时可以定义 **initData** 方法，可以在定义组件时指定组件初始化时的数据。**initData** 方法返回组件实例的初始化数据。

```
var MyApp = san.defineComponent({
    template: '<ul><li s-for="item in list">{{item}}</li></ul>',

    initData: function () {
        return {
            list: ['san', 'er', 'esui', 'etpl', 'esl']
        };
    }
});

var myApp = new MyApp();
myApp.attach(document.body);
```

计算数据

有时候，一个数据项的值可能由其他数据项计算得来，这时我们可以通过 **computed** 定义计算数据。**computed** 是一个对象，key 为计算数据项的名称，value 是返回数据项值的函数。

```
san.defineComponent({
    template: '<a>{{name}}</a>',

    // name 数据项由 firstName 和 lastName 计算得来
    computed: {
        name: function () {
            return this.data.get('firstName') + ' ' +
this.data.get('lastName');
        }
    }
});
```

上面的例子中，name 数据项是计算数据，依赖 firstName 和 lastName 数据项，其值由 firstName 和 lastName 计算得来。

注意：计算数据的函数中只能使用 *this.data.get* 方法获取数据项的值，不能通过 this.method 调用组件方法，也不能通过 this.data.set 设置组件数据。

```
san.defineComponent({
    template: '<a>{{info}}</a>',

    // name 数据项由 firstName 和 lastName 计算得来
    computed: {
        name: function () {
            return this.data.get('firstName') + ' ' + this.data.get('lastName');
        },

        info: function () {
            return this.data.get('name') + ' - ' + this.data.get('email');
        }
    }
});
```

计算数据项可以依赖另外一个计算数据项，上面的例子中，info 项依赖的 name 项就是一个计算数据项。但是使用时一定要注意，不要形成计算数据项之间的循环依赖。

过滤器

在定义视图模板时，插值是常用的展现数据的方式。在编写插值时，我们常使用 过滤器 将数据转换成适合视图展现的形式。

```
{{createTime | dateFormat('yyyy-MM-dd')}}
```

内置过滤器

San 针对常用场景，内置了几个过滤器：

- `html` - HTML 转义。当不指定过滤器时，默认使用此过滤器
- `url` - URL 转义
- `raw` - 不进行转义。当不想使用 HTML 转义时，使用此过滤器

定制过滤器

通过定义组件的 **filters** 成员，可以指定组件的视图模板可以使用哪些过滤器。

```
san.defineComponent({
    template: '<a>{{createTime | dateFormat("yyyy-MM-dd")}}</a>',

    filters: {
        dateFormat: function (value, format) {
            return moment(value).format(format);
        }
    }
});
```

过滤器函数的第一个参数是表达式对应的数据值，过滤器调用时传入的参数从第二个参数开始接在后面。

注意：考虑到组件的独立性，San 没有提供全局过滤器注册的方法，组件要使用的过滤器必须在自身的 **filters** 中定义。

组件层级

我们知道组件体系下，组件必须是可嵌套的树形关系。下面从一段代码，做一些说明。在下面的代码中，AddForm 内部使用了两个自定义组件：ui-calendar 和 ui-timepicker。

```
<!-- Template -->
<div class="form">
    <input type="text" class="form-title" placeholder="标题" value="{= title =}">
    <textarea class="form-desc" placeholder="备注" value="{= desc =}"></textarea>

    <div>预期完成时间：
        <ui-calendar value="{= endTimeDate =}" s-ref="endDate"></ui-calendar>
        <ui-timepicker value="{= endTimeHour =}" s-ref="endHour"></ui-timepicker>
    </div>

    <div class="form-op">
        <button type="button" on-click="submit">ok</button>
    </div>
</div>
```

```
var AddForm = san.defineComponent({
    // template

    components: {
        'ui-timepicker': require('../ui/TimePicker'),
        'ui-calendar': require('../ui/Calendar')
    },

    submit: function () {
        this.ref('endDate')
        this.ref('endHour')
    }
});
```

components

组件中通常通过声明自定义元素，使用其它组件。

组件视图可以使用哪些子组件类型，必须通过定义组件的 **components** 成员指定。key 是自定义元素的标签名，value 是组件的类。

注意：考虑到组件的独立性，San 没有提供全局组件注册的方法，组件必须在自身的 **components** 中声明自己内部会用到哪些组件。

有些组件可能在内容中会使用自己，比如树的节点。我们可以将 **components** 中这一项的值设置为字符串 **self**。

```
var Node = san.defineComponent({
    // template

    components: {
        'ui-node': 'self'
    }
});
```

owner 与 parent

owner 与 **parent** 的概念已经被 react 明确过了，但这里还是要专门明确下。

owner 指的是目标在声明时位于哪个组件的组件视图中，其生存时间、交互的通信等行为都由 **owner** 管理。**owner** 必须是一个组件。ui-calendar 的 **owner** 是 AddForm 组件。

parent 指的是目标在视图中对应的直接父级元素。ui-calendar 的 **parent** 是其上层的 div。**parent** 对组件管理并没有直接的意义。

ref

子组件声明时如果通过 **s-ref** 指定了名称，则可以在 JavaScript 中通过组件实例的 **ref** 方法调用到。

提示：有了声明式的初始化、数据绑定与事件绑定，我们很少需要在 JavaScript 中拿到子组件的实例。San 提供了这个途径，但当你用到它的时候，请先思考是不是非要这么干。

消息

通过 **dispatch** 方法，组件可以向组件树的上层派发消息。

```
var SelectItem = san.defineComponent({
    template: '<li on-click="select"><slot></slot></li>',

    select: function () {
        var value = this.data.get('value');

        // 向组件树的上层派发消息
        this.dispatch('UI:select-item-selected', value);
    }
});
```

消息将沿着组件树向上传递，直到遇到第一个处理该消息的组件，则停止。通过 **messages** 可以声明组件要处理的消息。**messages** 是一个对象，key 是消息名称，value 是消息处理的函数，接收一个包含 target(派发消息的组件) 和 value(消息的值) 的参数对象。

```
var Select = san.defineComponent({
    template: '<ul><slot></slot></ul>',

    // 声明组件要处理的消息
    messages: {
        'UI:select-item-selected': function (arg) {
            var value = arg.value;
            this.data.set('value', value);

            // arg.target 可以拿到派发消息的组件
        }
    }
```

```
});
```

消息主要用于组件与非 **owner** 的上层组件进行通信。比如，slot 内组件 SelectItem 的 **owner** 是更上层的组件，但它需要和 Select 进行通信。

```
san.defineComponent({
    components: {
        'ui-select': Select,
        'ui-selectitem': SelectItem
    },

    template: ''
        + '<div>'
        + '  <ui-select value="{=value=}">'
        + '    <ui-selectitem value="1">one</ui-selectitem>'
        + '    <ui-selectitem value="2">two</ui-selectitem>'
        + '    <ui-selectitem value="3">three</ui-selectitem>'
        + '  </ui-select>'
        + '</div>'
});
```

动态子组件

在一些场景下，我们希望组件不在自身视图渲染时创建子组件，而是通过 JavaScript 灵活控制在未来的某些时间点创建子组件。比如:

- 浮动层子组件的 **parent** 不在其根元素 **el** 内，声明式用着不方便
- 列表只有在用户点击时才需要创建并展示

动态子组件对开发者要求更高，我们在这里给出一些需要注意的地方，下面节选的代码也做了一些简单的说明:

- 动态创建的子组件无需在 **components** 中声明类型
- 保证动态子组件不要被重复创建。常见的做法是在实例的属性上持有对创建组件的引用，并以此作判断
- 保证动态子组件能够被销毁。你可以在创建时 push 到 **children** 中，或者在 **disposed** 中销毁它

```
san.defineComponent({
    mainClick: function () {
        if (!this.layer) {
            this.layer = new Layer();
```

```
            this.layer.attach(document.body);

            // 如果有下面一句，则可以不用手动在 disposed 中释放
            // this.children.push(this.layer);
        }

        this.layer.show();
    },

    disposed: function () {
        if (this.layer) {
            this.layer.dispose();
        }

        this.layer = null;
    }
});
```

组件反解（old）

该文档已经过期。新的组件反解基于数据和模板匹配的机制，代替原来的标记机制。

版本 : < 3.4.0

提示 : 通过 San 进行服务端渲染，一定能通过相同版本的 San 在浏览器端进行反解。

概述

组件初始化时传入 **el**，其将作为组件根元素，并以此反解析出视图结构。

该特性的意图是：有时我们为了首屏时间，期望初始的视图是直接的 HTML，不由组件渲染。但是我们希望组件为我们管理数据、逻辑与视图，后续的用户交互行为与视图变换通过组件管理。

```
var myComponent = new MyComponent({
    el: document.getElementById('wrap').firstChild
});
```

以 **el** 初始化组件时，San 会尊重 **el** 的现时 HTML 形态，不会执行任何额外影响视觉的渲染动作。

- 不会使用预设的 template 渲染视图
- 不会创建根元素
- 直接到达 compiled、created、attached 生命周期

但是，**el** 可能需要一些额外的标记，来帮助 San 认识数据与视图的结构（插值、绑定、循环、分支、组件等），以便于后续的行为管理与视图刷新。

数据和视图是组件重要的组成部分，我们将从这两个方面说明组件反解的功能。

提示 : 如果使用 NodeJS 做服务端，San 提供了 服务端渲染 的支持，能够天然输出标记好可被组件反解的 HTML，你无需了解组件反解的标记形式。如果你服务端使用其他的语言（比如PHP），请继续往下阅读。

视图

该章节介绍如何对视图的结构（插值、绑定、循环、分支、组件等）进行标记。

插值文本

插值文本的标记方式是：在文本的前后各添加一个注释。

- 文本前的注释以 **s-text:** 开头，紧跟着插值文本的声明。
- 文本后的注释内容为 **/s-text**，代表插值文本片段结束。

```
1. <span><!--s-text:{{name}} - {{email}}-->errorrik - errorrik@gmail.com<!--/s-
   text--></span>
```

提示 **：s-** 开头的 **HTML Comment** 是重要的标记手段，在循环与分支标记中也会用到它。

插值属性

插值属性的标记方式是：在 **prop-** 为前缀的属性上声明属性的内容。

```
1. <span title="errorrik - errorrik@gmail.com" prop-title="{{name}} - {{email}}">
   </span>
```

绑定

绑定属性的标记方式与插值属性完全一样：在 **prop-** 为前缀的属性上声明属性的内容。

```
1. <ui-label prop-title="{{name}}" prop-text="{{jokeName}}"></ui-label>
```

双向绑定用 **{= expression =}** 的形式。

```
1. <input prop-value="{=name=}" value="errorrik">
```

循环

对于循环，我们需要以桩元素，分别标记循环的 起始 和 结束。

```
1. <ul id="list">
2.     <!--s-for:<li s-for="p,i in persons" title="{{p.name}}">{{p.name}} -
   {{p.email}}</li>-->
3.     <li prop-title="{{p.name}}" title="errorrik"><!--s-text:{{p.name}} -
   {{p.email}}-->errorrik - errorrik@gmail.com<!--/s-text--></li>
4.     <li prop-title="{{p.name}}" title="otakustay"><!--s-text:{{p.name}} -
   {{p.email}}-->otakustay - otakustay@gmail.com<!--/s-text--></li>
5.     <!--/s-for-->
6. </ul>
```

起始 的桩元素标记是一个以 **s-for:** 开头的 HTML Comment，接着是声明循环的标签内容。

```
<!--s-for:<li s-for="p,i in persons" title="{{p.name}}">{{p.name}} -
{{p.email}}</li>-->
```

结束 的桩元素标记是一个内容为 **/s-for** 的 HTML Comment。

```
<!--/s-for-->
```

对于循环的每个元素，按照普通元素标记，无需标记 **for** directive。通常它们在 HTML 输出端也是以循环的形式存在，不会带来重复编写的工作量。

```
<li prop-title="{{p.name}}" title="otakustay"><!--s-text:{{p.name}} -
{{p.email}}-->otakustay - otakustay@gmail.com<!--/s-text--></li>
```

提示：当初始没有数据时，标记循环只需要声明 起始 和 结束 桩即可。

分支

分支的标记比较简单，以 **s-if** 标记分支元素即可。

```
<span s-if="condition" title="errorrik" prop-title="{{name}}"></span>
```

当初始条件为假时，分支元素不会出现，此时以 **HTML Comment** 为桩，标记分支。在桩的内部声明分支的语句。

```
<!--s-if:<span s-if="cond" title="{{name}}">{{name}}</span>--><!--/s-if-->
```

一个包含完整 if-else 的分支，总有一个元素是具体元素，有一个元素是桩。

```
<!--s-if:<span s-if="isErik" title="{{name}}">{{name}}</span>--><!--/s-if-->
<span s-else title="otakustay" prop-title="{{name2}}"><!--s-text:{{name2}}--
>otakustay<!--/s-text--></span>
```

```
<span s-if="isErik" title="errorrik" prop-title="{{name}}"><!--s-text:{{name}}-
->errorrik<!--/s-text--></span>
<!--s-else:<span s-else title="{{name2}}">{{name2}}</span>--><!--/s-else-->
```

组件

```
san.defineComponent({
```

```
    components: {
        'ui-label': Label
    }
});
```

组件的标记与视图模板中声明是一样的，在相应的自定义元素上标记绑定。San 将根据自定义元素的标签自动识别组件。

```
<ui-label prop-title="{{name}}" prop-text="{{email}}">
    <b prop-title="{{title}}" title="errorrik"><!--s-text:{{text}}-->errorrik@gmail.com<!--/s-text--></b>
</ui-label>
```

我们可能因为样式、兼容性等原因不想使用自定义元素。当组件未使用自定义元素时，可以在元素上通过 **s-component** 标记组件类型。

```
<label s-component="ui-label" prop-title="{{name}}" prop-text="{{email}}">
    <b prop-title="{{title}}" title="errorrik"><!--s-text:{{text}}-->errorrik@gmail.com<!--/s-text--></b>
</label>
```

slot

slot 的标记与循环类似，我们需要以桩元素，分别标记循环的 起始 和 结束。

```
<div id="main">
    <!--s-data:{"tabText":"tab","text":"one","title":"1"}-->
    <div s-component="ui-tab" prop-text="{{tabText}}">
        <div prop-class="head" class="head">
            <!--s-slot:title-->
            <h3 prop-title="{{title}}" title="1"><!--s-text:{{title}}-->1<!--/s-text--></h3>
            <!--/s-slot-->
        </div>
        <div>
            <!--s-slot-->
            <p prop-title="{{text}}" title="one"><!--s-text:{{text}}-->one<!--/s-text--></p>
            <!--/s-slot-->
        </div>
    </div>
```

```
</div>
```

```
var Tab = san.defineComponent({
    template: [
        '<div>',
        '    <div class="head"><slot name="title"></slot></div>',
        '    <div><slot></slot></div>',
        '</div>'
    ].join('\n')
});

var MyComponent = san.defineComponent({
    components: {
        'ui-tab': Tab
    },

    template: [
        '<div><ui-tab text="{{tabText}}">',
        '    <h3 slot="title" title="{{title}}">{{title}}</h3>',
        '    <p title="{{text}}">{{text}}</p>',
        '</ui-tab></div>'
    ].join('\n')
});

var myComponent = new MyComponent({
    el: document.getElementById('main')
});
```

起始 的桩元素标记是一个以 **s-slot:** 开头的 HTML Comment，接着是 slot 名称。

```
<!--s-slot:title-->
```

结束 的桩元素标记是一个内容为 **/s-slot** 的 HTML Comment。

```
<!--/s-slot-->
```

当 owner 未给予相应内容时，slot 的内容为组件内声明的默认内容，这时 slot 内环境为组件内环境，而不是组件外环境。对默认内容，需要在 起始 的桩元素 name 之前加上 ! 声明。

```
<!--s-slot:!title-->
```

数据

组件的视图是数据的呈现。我们需要通过在组件起始时标记 **data**，以指定正确的初始数据。初始数据标记是一个 **s-data:** 开头的 HTML Comment，在其中声明数据。

```
<div id="wrap">
    <!--s-data:{
        email: 'error@gmail.com',
        name: 'errorrik'
    }-->
    <span title="errorrik@gmail.com" prop-title="{{email}}"><!--s-text:{{name}}-->errorrik<!--/s-text--></span>
</div>
```

```
var myComponent = new MyComponent({
    el: document.getElementById('wrap')
});
```

警告：如果 HTML 中只包含视图结果，不包含数据，组件无法从视图的 DOM 结构中解析出其代表数据，在后续的操作中可能会导致不期望的后果。

比如，对于列表数据应该在初始化时保证数据与视图的一致，因为列表的添加删除等复杂操作与视图更新上关系密切，如果一开始对应不上，视图更新可能产生难以预测的结果。

```
<ul id="list">
    <li>name - email</li>
    <!--s-for:<li s-for="p,i in persons" title="{{p.name}}">{{p.name}} - {{p.email}}</li>-->
    <li prop-title="{{p.name}}" title="errorrik"><!--s-text:{{p.name}} - {{p.email}}-->errorrik - errorrik@gmail.com<!--/s-text--></li>
    <li prop-title="{{p.name}}" title="otakustay"><!--s-text:{{p.name}} - {{p.email}}-->otakustay - otakustay@gmail.com<!--/s-text--></li>
    <!--/s-for-->
</ul>
```

```
var myComponent = new MyComponent({
    el: document.getElementById('list')
});

// 组件不包含初始数据标记
// 下面的语句将导致错误
```

```
myComponent.data.removeAt('persons', 1);
```

提示：如果一个组件拥有 owner，可以不用标记初始数据。其初始数据由 owner 根据绑定关系灌入。

```
<!-- ui-label 组件拥有 owner，无需进行初始数据标记 -->
<div id="main">
    <!--s-data:{"name":"errorrik"}-->
    <span s-component="ui-label" prop-text="{{name}}"><!--s-text:{{text}}-->errorrik<!--/s-text--></span>
</div>
```

```
var Label = san.defineComponent({
    template: '<span>{{text}}</span>'
});

var MyComponent = san.defineComponent({
    components: {
        'ui-label': Label
    },

    template: '<div><ui-label text="{{name}}"></ui-label></div>'
});

var myComponent = new MyComponent({
    el: document.getElementById('main')
});
```

组件反解

3.4.0 对组件反解机制做了全面的升级。新的组件反解基于数据和模板匹配的机制，代替原来的标记机制。

旧的基于标记的组件反解请见老文档

版本 : >= 3.4.0

提示 : 通过 San 进行服务端渲染，一定能通过相同版本的 San 在浏览器端进行反解。

概述

组件初始化时传入 **el**，其将作为组件根元素，并以此反解析出视图结构。

该特性的意图是：有时我们为了首屏时间，期望初始的视图是直接的 HTML，不由组件渲染。但是我们希望组件为我们管理数据、逻辑与视图，后续的用户交互行为与视图变换通过组件管理。

```
var myComponent = new MyComponent({
    el: document.getElementById('wrap').firstChild
});
```

以 **el** 初始化组件时，San 会尊重 **el** 的现时 HTML 形态，不会执行任何额外影响视觉的渲染动作。

- 不会使用预设的 template 渲染视图
- 不会创建根元素
- 直接到达 compiled、created、attached 生命周期

数据

组件的视图是数据的呈现。我们需要通过在组件起始时标记 **data**，以指定正确的初始数据。初始数据标记是一个 **s-data:** 开头的 HTML Comment，在其中声明数据。

警告 : San 的组件反解过程基于数据和组件模板进行视图结构反推与匹配。反解的组件必须拥有正确的标记数据，否则反解过程会发生错误。比如对于模板中的 s-if 条件进行视图反推，如果没有正确的标记数据，反推就会因为元素对应不上，得到不期望的结果。

```
<a id="wrap"><!--s-data:{
        email: 'error@gmail.com',
        name: 'errorrik'}-->
```

```
    <span title="errorrik@gmail.com">errorrik</span>
</a>
```

```
var MyComponent = san.defineComponent({
    template: ''
        + '<a>\n'
        + '    <span title="{{email}}">{{name}}</span>\n'
        + '</a>'
});

var myComponent = new MyComponent({
    el: document.getElementById('wrap')
});
```

如果一个组件拥有 owner，不用标记初始数据。其初始数据由 owner 根据绑定关系灌入。

```
<!-- ui-label 组件拥有 owner, 无需进行初始数据标记 -->
<div id="main"><!--s-data:{"name":"errorrik"}-->
    <span>errorrik</span>
</div>
```

```
var Label = san.defineComponent({
    template: '<span>{{text}}</span>'
});

var MyComponent = san.defineComponent({
    components: {
        'ui-label': Label
    },

    template: ''
        + '<div>\n'
        + '    <ui-label text="{{name}}"></ui-label>\n'
        + '</div>'
});

var myComponent = new MyComponent({
    el: document.getElementById('main')
});
```

复合插值文本

San 支持在插值文本中直接输出 HTML，此时插值文本对应的不是一个 TextNode，而可能是多个不同类型的元素。对于这种复合插值文本，需要在内容的前后各添加一个注释做标记。

- 文本前的注释内容为 **s-text**，代表插值文本片段开始。
- 文本后的注释内容为 **/s-text**，代表插值文本片段结束。

```
<a id="wrap"><!--s-data:{
        name: 'new <b>San</b>'}-->
    <span>Hello <!--s-text-->new <b>San</b><!--/s-text-->!</span>
</a>
```

```
var MyComponent = san.defineComponent({
    template: ''
        + '<a>\n'
        + '    <span>Hello {{name|raw}}!</span>\n'
        + '</a>'
});

var myComponent = new MyComponent({
    el: document.getElementById('wrap')
});
```

服务端渲染

San 的服务端渲染支持是基于 组件反解 的：

- 服务端输出的 HTML 中带有对视图无影响，能帮助组件了解数据与视图结构的标记片段
- 浏览器端，组件初始化时从标记片段理解组件结构，在后续用户操作时组件能正确响应，发挥作用

提示：由于组件运行环境需要考虑浏览器各版本和NodeJS，示例代码为保证简单无需transform，全部采用ES5编写。

是否需要SSR

服务端渲染，视图 HTML 直出有一些显而易见的好处：

- SEO 友好，HTML 直接输出对搜索引擎的抓取和理解更有利
- 用户能够第一时间看到内容。从开发者角度来说，首屏时间更快到来

但是，如果使用服务端渲染，我们将面对：

- 更高的成本。虽然我们开发的是一份组件，但我们需要考虑其运行时是 NodeJS 和 浏览器 双端的，我们需要考虑在服务端渲染要提前编译，我们需要考虑组件的源码如何输出到浏览器端，我们需要考虑开发组件的浏览器兼容性，到底要写老旧的浏览器兼容性好的代码还是按ESNext写然后通过打包编译时 transform。这依然带来了维护成本的增加，即使不多
- 用户可交互时间不一定更早到来。交互行为是由组件管理的，组件从当前视图反解出数据和结构需要遍历 DOM 树，反解的时间不一定比在前端直接渲染要快

所以，我们建议，使用服务端渲染时全面评估，只在必须使用的场景下使用。下面是一些场景建议：

- 后台系统（CMS、MIS、DashBoard之类）大多使用 Single Page Application 的模式。显而易见，这类系统不需要使用 SSR
- 功能型页面，不需要使用 SSR。比如个人中心、我的收藏之类
- 仅在 App 的 WebView 中展现，不作为开放 Web 存在的页面，不需要使用 SSR
- 偏重内容型页面，可以使用 SSR。但是组件是管理行为交互的，对内容部分无需进行组件渲染，只需要在有交互的部分进行组件反解渲染

输出HTML

```
1.  var MyComponent = san.defineComponent({
2.      template: '<a><span title="{{email}}">{{name}}</span></a>'
3.  });
```

```
4.
5.  var render = san.compileToRenderer(MyComponent);
6.  render({
7.      email: 'errorrik@gmail.com',
8.      name: 'errorrik'
9.  });
10. // render html result:
11. // <a>....</a>
```

San 在主包下提供了 **compileToRenderer** 方法。该方法接收组件的类作为参数，编译返回一个 **{string}render({Object} data)** 方法。 **render** 方法接收数据，返回组件渲染后的 HTML 字符串。

编译NodeJS模块

有时候，我们希望组件编译的 render 方法是一个单独的 NodeJS Module，以便于其他模块引用它。通过 San 主包下提供的 **compileToSource** 方法我们可以编译 NodeJS Module。

```
1.  var san = require('san');
2.  var fs = require('fs');
3.
4.  var MyComponent = san.defineComponent({
5.      template: '<a><span title="{{email}}">{{name}}</span></a>'
6.  });
7.
8.  var renderSource = san.compileToSource(MyComponent);
9.  fs.writeFileSync('your-module.js', 'exports = module.exports = ' +
    renderSource, 'UTF-8');
```

compileToSource 方法接收组件的类作为参数，编译返回组件 render 的 source code，具体为 `function (data) {...}` 形式的字符串。我们只需要在前面增加 `exports = module.exports =` ，并写入 **.js** 文件中，就能得到一个符合 CommonJS 标准的 NodeJS Module。

API

- 主模块API
- 组件API

主模块API

组件初始化

defineComponent

描述 : defineComponent({Object}propertiesAndMethods)

解释 :

方法 。定义组件的快捷方法。详细请参考组件定义文档。

用法 :

```
var MyApp = san.defineComponent({
    template: '<ul><li s-for="item in list">{{item}}</li></ul>',

    initData: function () {
        return {
            list: ['san', 'er', 'esui', 'etpl', 'esl']
        };
    }
});
```

compileComponent

版本 : >= 3.3.0

描述 : {void} compileComponent({Function}ComponentClass)

解释 :

方法 。编译组件，组件的编译过程主要是解析 template 成 ANode，并对 components 中的 plain object 执行 defineComponent。

组件会在其第一个实例初始化时自动编译。我们通常不会使用此方法编译组件，除非你有特殊的需求希望组件的编译过程提前。

用法 :

```
var MyApp = san.defineComponent({
    template: '<ul><li s-for="item in list">{{item}}</li></ul>',
```

```
3.
4.      initData: function () {
5.          return {
6.              list: ['san', 'er', 'esui', 'etpl', 'esl']
7.          };
8.      }
9.  });
10.
11. typeof MyApp.prototype.aNode // undefined
12. san.compileComponent(MyApp);
13. typeof MyApp.prototype.aNode // object
```

Component

类型 : Function

解释 :

属性 。组件类，定义组件时可以从此继承。通常通过 **san.defineComponent** 定义组件，不使用此方法。详细请参考组件定义文档。

用法 :

```
1.  import {Component} from 'san';
2.
3.  class HelloComponent extends Component {
4.
5.      static template = '<p>Hello {{name}}!</p>';
6.
7.      initData() {
8.          return {name: 'San'}
9.      }
10. }
```

inherits

描述 : inherits({Function}SubClass, {Function}SuperClass)

解释 :

方法 。一个通用的实现继承的方法，定义组件时可以使用此方法从 **san.Component** 继承。通常在 ES5 下通过 **san.defineComponent** 定义组件，在 ESNext 下使用 **extends** 定义组件。

绝大多数情况不推荐使用此方法。详细请参考组件定义文档。

服务端渲染

compileToRenderer

版本 : >= 3.1.0

描述 : {function(Object):string} compileToRenderer({Function}ComponentClass)

解释 :

方法 。将组件类编译成 renderer 方法。详细请参考服务端渲染文档。

用法 :

```
var MyApp = san.defineComponent({
    template: '<ul><li s-for="item in list">{{item}}</li></ul>',

    initData: function () {
        return {
            list: ['san', 'er', 'esui', 'etpl', 'esl']
        };
    }
});

var render = san.compileToRenderer(MyApp);
```

compileToSource

版本 : >= 3.1.0

描述 : {string} compileToRenderer({Function}ComponentClass)

解释 :

方法 。将组件类编译成 renderer 方法的源文件。详细请参考服务端渲染文档。

用法 :

```
var MyApp = san.defineComponent({
    template: '<ul><li s-for="item in list">{{item}}</li></ul>',
```

```
3.
4.      initData: function () {
5.          return {
6.              list: ['san', 'er', 'esui', 'etpl', 'esl']
7.          };
8.      }
9.  });
10.
11. var renderSource = san.compileToSource(MyApp);
12. fs.writeFileSync('your-module.js', 'exports = module.exports = ' +
    renderSource, 'UTF-8');
```

模板编译

ExprType

版本 : >= 3.0.3

类型 : Object

解释 :

属性 。表达式类型枚举，有助于帮你理解和使用 San 的模板编译结果。详细请参考ANode文档。

parseExpr

版本 : >= 3.0.3

描述 : {Object} parseExpr({string}source)

解释 :

方法 。将源字符串解析成表达式对象。详细请参考ANode文档。

用法 :

```
1.  var expr = san.parseExpr('!user.isLogin');
2.  /*
3.  expr = {
4.      type: ExprType.UNARY,
5.      expr: {
6.          type: ExprType.ACCESSOR,
7.          paths: [
```

```
            {type: ExprType.STRING, value: 'user'},
            {type: ExprType.STRING, value: 'isLogin'}
        ]
    }
}
*/
```

parseTemplate

版本 : >= 3.0.3

描述 : {ANode} parseTemplate({string}source)

解释 :

方法 。将源字符串解析成 ANode 对象。如果你想使用 San 的模板形式，但是自己开发视图渲染机制，可以使用该方法解析模板。详细请参考ANode文档。

用法 :

```
var aNode = san.parseTemplate('<p>Hello {{name}}!</p>');
/*
aNode = {
    "directives": [],
    "props": [],
    "events": [],
    "children": [
        {
            "isText": true,
            "text": "Hello {{name}}!",
            "textExpr": {
                "type": ExprType.TEXT,
                "segs": [
                    {
                        "type": ExprType.STRING,
                        "value": "Hello "
                    },
                    {
                        "type": ExprType.INTERP,
                        "expr": {
                            "type": ExprType.ACCESSOR,
                            "paths": [
                                {
```

```
                                    "type": ExprType.STRING,
                                    "value": "name"
                                }
                            ]
                        },
                        "filters": []
                    }
                ]
            }
        }
    ],
    "tagName": "p"
}
*/
```

数据

San 开放了组件中使用的数据容器类与表达式计算函数，开发者可以用来管理一些与组件无关的数据，比如应用状态等。

Data

版本 : >= 3.5.6

类型 : Class Function

解释 :

数据容器类 ，包含 get、set、splice、push、pop、unshift、shift、merge、apply 数据方法，详细请参考数据操作文档。

通过方法变更数据时，data 对象将 fire change 事件。通过 listen 和 unlisten 方法可以监听或取消监听 change 事件。

```
var data = new san.Data({
    num1: 1,
    num2: 2
});

data.listen(function (change) {
    console.log(change.value);
});
```

```
9.
10. data.set('num2', 10);
11. // console 10
```

evalExpr

`版本` : >= 3.5.6

`描述` : {*} evalExpr({Object}expr, {Data}data, {Component=}owner)

`解释` :

方法 ，计算表达式的值。

- `expr` 可以通过 parseExpr 方法得到。支持的表达式类型可参考表达式文档
- `data` 可以是组件的数据对象，也可以是自己通过 new Data 得到的数据对象
- `owner` 仅用于表达式中 filter 的执行，表达式中无自定义 filter 时无需此参数

```
1. var data = new san.Data({
2.     num1: 1,
3.     num2: 2
4. });
5.
6. san.evalExpr(san.parseExpr('num1 + num2'), data)
7. // console 3
```

其他

debug

`类型` : boolean

`解释` :

属性 。是否开启调试功能。当同时满足以下两个条件时，可以在 chrome 中使用 **devtool** 进行调试。

- 主模块 **debug** 属性设为 **true**
- 当前页面环境中的 San 是带有 **devtool** 功能的版本。查看San的打包发布版本

version

类型 : string

解释 :

属性 。当前的 San 版本号。

LifeCycle

版本 : < 3.3.0 (已废弃)

类型 : Function

解释 :

属性 。生命周期类。如果你想自己开发管理组件的渲染和交互更新过程，LifeCycle 可能对你有所帮助。

LifeCycle 定义了以下生命周期，并且生命周期之间包含互斥关系，描述如下：

```
{
    compiled: {
        value: 1
    },

    inited: {
        value: 2
    },

    created: {
        value: 3
    },

    attached: {
        value: 4,
        mutex: 'detached'
    },

    detached: {
        value: 5,
        mutex: 'attached'
    },

    disposed: {
        value: 6,
```

```
        mutex: '*'
    }
}
```

通过 LifeCycle 的 set 方法，可以指定生命周期； 通过 LifeCycle 的 is 方法，可以判断是否处于生命周期。

用法：

```
var lifeCycle = new san.LifeCycle();

lifeCycle.set('attached');
lifeCycle.is('attached'); // true

lifeCycle.set('detached');
lifeCycle.is('attached'); // false
```

组件API

该文档描述了组件的 API，在 San 主模块上暴露的 API 请参考文档 主模块API。

初始化参数

data

解释：

组件初始化数据。通常在组件反解的场景下使用。

类型： Object

用法：

```
var MyComponent = san.defineComponent({});

var myComponent = new MyComponent({
    el: document.getElementById('my-label'),
    data: {
        email: 'errorrik@gmail.com',
        name: 'errorrik'
    }
});

/* html:
<label id="my-label">
    <span title="errorrik@gmail.com" prop-title="{{email}}">errorrik</span>
</label>
```

el

解释：

组件根元素。传入此参数意味着不使用组件的 **template** 作为视图模板，组件视图由 San 自动反解。详情可参考组件反解文档。

类型： HTMLElement

用法：

```
var MyComponent = san.defineComponent({});

var myComponent = new MyComponent({
    el: document.getElementById('my-label'),
    data: {
        email: 'errorrik@gmail.com',
        name: 'errorrik'
    }
});

/* html:
<label id="my-label">
    <span title="errorrik@gmail.com" prop-title="{{email}}">errorrik</span>
</label>
*/
```

transition

解释 :

组件的过渡动画控制器。可参考 动画控制器 和 动画控制器 Creator 文档。

版本 : >= 3.6.0

类型 : Object

用法 :

```
var MyComponent = san.defineComponent({
    template: '<span>transition</span>'
});

var myComponent = new MyComponent({
    transition: {
        enter: function (el, done) { /* 进入时的过渡动画 */ },
        leave: function (el, done) { /* 离开时的过渡动画 */ },
    }
});
```

生命周期钩子

生命周期代表组件的生存过程，在每个过程到达时将触发钩子函数。具体请参考生命周期文档。

compiled

解释：

组件视图模板编译完成。组件上的 **compiled** 方法将会被调用。

inited

解释：

组件实例初始化完成。组件上的 **inited** 方法将会被调用。

created

解释：

组件元素创建完成。组件上的 **created** 方法将会被调用。

attached

解释：

组件已被附加到页面中。组件上的 **attached** 方法将会被调用。

detached

解释：

组件从页面中移除。组件上的 **detached** 方法将会被调用。

disposed

解释：

组件卸载完成。组件上的 **disposed** 方法将会被调用。

updated

解释：

组件由于数据变化，视图完成一次刷新。组件上的 **updated** 方法将会被调用。

定义组件成员

template

解释 :

组件的视图模板。详细描述请参考视图模板文档。

类型 : string

用法 :

```
san.defineComponent({
    template: '<span title="{{text}}">{{text}}</span>'
});
```

filters

解释 :

声明组件视图模板中可以使用哪些过滤器。详细描述请参考过滤器文档。

类型 : Object

用法 :

```
san.defineComponent({
    template: '<a>{{createTime | dateFormat('yyyy-MM-dd')}}</a>',

    filters: {
        dateFormat: function (value, format) {
            return moment(value).format(format);
        }
    }
});
```

components

解释 :

声明组件中可以使用哪些类型的子组件。详细描述请参考components文档。

类型 : Object

用法 :

```
var AddForm = san.defineComponent({
    components: {
        'ui-timepicker': TimePicker,
        'ui-calendar': Calendar
    }
});
```

computed

解释 :

声明组件中的计算数据。详细描述请参考计算数据文档。

类型 : Object

用法 :

```
san.defineComponent({
    template: '<a>{{name}}</a>',

    // name 数据项由 firstName 和 lastName 计算得来
    computed: {
        name: function () {
            return this.data.get('firstName') + ' ' +
this.data.get('lastName');
        }
    }
});
```

messages

解释 :

声明处理子组件派发消息的方法。详细描述请参考消息文档。

类型 : Object

用法 :

```
var Select = san.defineComponent({
    template: '<ul><slot></slot></ul>',

    messages: {
```

```
        'UI:select-item-selected': function (arg) {
            // arg.target 可以拿到派发消息的组件
            var value = arg.value;
            this.data.set('value', value);
        }
    }
});
```

initData

解释：

返回组件实例的初始数据。详细描述请参考初始数据文档。

类型： function():Object

用法：

```
var MyApp = san.defineComponent({
    template: '<ul><li s-for="item in list">{{item}}</li></ul>',

    initData: function () {
        return {
            list: ['san', 'er', 'esui', 'etpl', 'esl']
        };
    }
});
```

trimWhitespace

定义组件模板解析时对空白字符的 trim 模式。

- 默认为 **none**，不做任何事情
- **blank** 时将清除空白文本节点
- **all** 时将清除所有文本节点的前后空白字符

版本：>= 3.2.5

类型： string

用法：

```
var MyApp = san.defineComponent({
```

```
    trimWhitespace: 'blank'

    // ,
    // ......
});
```

delimiters

解释 :

定义组件模板解析时插值的分隔符。值为2个项的数组，分别为起始分隔符和结束分隔符。默认为 `['{{', '}}']` 。

版本 : >= 3.5.0

类型 : Array

用法 :

```
var MyComponent = san.defineComponent({
    delimiters: ['{%', '%}'],
    template: '<a><span title="Good {%name%}">Hello {%name%}</span></a>'
});
```

transition

解释 :

定义组件根节点的过渡动画控制器。已废弃。

版本 : >= 3.3.0, < 3.6.0

类型 : Object

用法 :

```
var MyComponent = san.defineComponent({
    template: '<span>transition</span>',
    transition: {
        enter: function (el) { /* 根节点进入时的过渡动画 */ },
        leave: function (el, done) { /* 根节点离开时的过渡动画 */ },
    }
});
```

组件方法

fire

描述： fire({string}eventName, {*}eventArgument)

解释：

派发一个自定义事件。San 为组件提供了自定义事件功能，组件开发者可以通过该方法派发事件。事件可以在视图模板中通过 **on-** 的方式绑定监听，也可以通过组件实例的 **on** 方法监听。可参考Event文档。

用法：

```
var Label = san.defineComponent({
    template: '<template class="ui-label"><a on-click="clicker" title="{{text}}">{{text}}</a></template>',

    clicker: function () {
        this.fire('customclick', this.data.get('text') + ' clicked');
    }
});

var MyComponent = san.defineComponent({
    initData: function () {
        return {name: 'San'};
    },

    components: {
        'ui-label': Label
    },

    template: '<div><ui-label text="{{name}}" on-customclick="labelClicker($event)"></ui-label></div>',

    labelClicker: function (doneMsg) {
        alert(doneMsg);
    }
});
```

on

描述： on({string}eventName, {Function}eventListener)

解释：

添加自定义事件监听器。 **on** 一般仅用在使用 JavaScript 动态创建的组件中，通过视图模板创建的子组件应通过 **on-** 的方式绑定监听。可参考动态子组件文档

un

描述： un({string}eventName, {Function=}eventListener)

解释：

移除事件监听器。 当 eventListener 参数为空时，移除所有 eventName 事件的监听器。

dispatch

描述： dispatch({string}name, {*}value)

解释：

派发一个消息。消息将沿着组件树向上传递，直到遇到第一个处理该消息的组件。上层组件通过 **messages** 声明组件要处理的消息。消息主要用于组件与非 **owner** 的上层组件进行通信。可参考消息文档。

用法：

```
var SelectItem = san.defineComponent({
    template:
        '<li on-click="select" class="{{value === selectValue ? \'selected\' : \'\'}}">'
        + '<slot></slot>'
        + '</li>',

    // 子组件在各种时机派发消息
    select: function () {
        var value = this.data.get('value');
        this.dispatch('UI:select-item-selected', value);
    },

    attached: function () {
        this.dispatch('UI:select-item-attached');
    },

```

```
    detached: function () {
        this.dispatch('UI:select-item-detached');
    }
});

var Select = san.defineComponent({
    template: '<ul><slot></slot></ul>',

    // 上层组件处理自己想要的消息
    messages: {
        'UI:select-item-selected': function (arg) {
            var value = arg.value;
            this.data.set('value', value);

            // 原则上上层组件允许更改下层组件的数据，因为更新流是至上而下的
            var len = this.items.length;
            while (len--) {
                this.items[len].data.set('selectValue', value);
            }
        },

        'UI:select-item-attached': function (arg) {
            this.items.push(arg.target);
            arg.target.data.set('selectValue', this.data.get('value'));
        },

        'UI:select-item-detached': function (arg) {
            var len = this.items.length;
            while (len--) {
                if (this.items[len] === arg.target) {
                    this.items.splice(len, 1);
                }
            }
        }
    },

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

var MyComponent = san.defineComponent({
```

```
    components: {
        'ui-select': Select,
        'ui-selectitem': SelectItem
    },

    template: ''
        + '<div>'
        + '  <ui-select value="{=value=}">'
        + '    <ui-selectitem value="1">one</ui-selectitem>'
        + '    <ui-selectitem value="2">two</ui-selectitem>'
        + '    <ui-selectitem value="3">three</ui-selectitem>'
        + '  </ui-select>'
        + '</div>'
});
```

watch

描述 : watch({string}dataName, {function({*}value)}listener)

解释 :

监听组件的数据变化。通常我们使用绑定，在子组件数据变化时自动更新父组件的对应数据。 **watch** 一般仅用在使用 JavaScript 动态创建的组件中。可参考动态子组件文档

```
san.defineComponent({
    // ...

    initLayer: function () {
        if (!this.monthView) {
            var monthView = new MonthView();
            this.monthView = monthView;

            this.monthView.watch('value', (function (value) {
                this.data.set('value', value);
            }).bind(this));

            this.watch('value', function (value) {
                monthView.data.set('value', value);
            });

            this.monthView.attach(document.body);
        }
```

```
    }
});
```

ref

描述： ref({string}name)

解释：

获取定义了 **s-ref** 的子组件。详细请参考组件层级文档。

用法：

```
var AddForm = san.defineComponent({
    // template

    components: {
        'ui-timepicker': require('../ui/TimePicker'),
        'ui-calendar': require('../ui/Calendar')
    },

    submit: function () {
        this.ref('endDate')
        this.ref('endHour')
    }
});

/* template:
<div class="form">
    <div>预期完成时间：
        <ui-calendar bindx-value="endTimeDate" s-ref="endDate"></ui-calendar>
        <ui-timepicker bindx-value="endTimeHour" s-ref="endHour"></ui-
timepicker>
    </div>

    <div class="form-op">
        <button type="button" on-click="submit">ok</button>
    </div>
</div>
*/
```

slot

版本 : >= 3.3.0

描述 : {Array} slot({string=}name)

解释 :

获取组件插槽的节点信息。返回值是一个数组，数组中的项是节点对象。通常只有一项，当 slot 声明中应用了 if 或 for 时可能为 0 项或多项。节点对象包含 isScoped 、 isInserted 和 children。

插槽详细用法请参考 slot 文档。

注意 : 不要对返回的 slot 对象进行任何修改。如果希望操作视图变更，请操作数据。

用法 :

```
var Panel = san.defineComponent({
    template: '<div><slot s-if="!hidden"/></div>',
});

var MyComponent = san.defineComponent({
    components: {
      'x-panel': Panel
    },

    template: ''
        + '<div>'
          + '<x-panel hidden="{{folderHidden}}" s-ref="panel"><p>{{desc}}</p></x-panel>'
        + '</div>',

    attached: function () {
        // 1
        this.ref('panel').slot().length

        var contentSlot = this.ref('panel').slot()[0];

        // truthy
        contentSlot.isInserted

        // falsy
        contentSlot.isScoped
    }
});
```

```
28.
29.
30. var myComponent = new MyComponent({
31.     data: {
32.         desc: 'MVVM component framework',
33.     }
34. });
```

nextTick

解释：

San 的视图更新是异步的。组件数据变更后，将在下一个时钟周期更新视图。如果你修改了某些数据，想要在 DOM 更新后做某些事情，则需要使用 `nextTick` 方法。

用法：

```
1.  const Component = san.defineComponent({
2.      template: `
3.          <div>
4.              <div s-ref="name">{{name}}</div>
5.              <button on-click="clicker">change name</button>
6.          </div>
7.      `,
8.
9.      initData() {
10.         return {name: 'erik'};
11.     },
12.
13.     clicker() {
14.         this.data.set('name', 'leeight');
15.         console.log(this.ref('name').innerHTML); // erik
16.         this.nextTick(() => {
17.             console.log(this.ref('name').innerHTML); // leeight
18.         });
19.     }
20. });
```