WebComponents

Stencil开发实践

主题

Components API

@Prop

@State

JSX

@Event & @Listen

@Method

Styling

Functional Components

Lifecycle

Components API

My first Comoponent


```
import { Component, Prop, h } from '@stencil/core';

@Component({
  tag: 'hello-world',
})
export class HelloWorld {

  @Prop() name: string;

  render() {
    return (
      <p>
        Hello {this.name}
      </p>
    );
  }
}
```

```
<hello-world name="CitiBank"></hello-world>
```

Decorators

Decorators are a pure compiler-time construction used by stencil to collect all the metadata about a component, the properties, attributes and methods it might expose, the events it might emit or even the associated stylesheets. Once all the metadata has been collected, all the decorators are removed from the output, so they don't incur any runtime overhead.

- **@Component()** declares a new web component
- **@Prop()** declares an exposed property/attribute
- **@State()** declares an internal state of the component
- **@Watch()** declares a hook that runs when a property or state changes
- **@Element()** declares a reference to the host element
- **@Method()** declares an exposed public method
- **@Event()** declares a DOM event the component might emit
- **@Listen()** listens for DOM events

Lifecycle hooks

- connectedCallback()
- disconnectedCallback()
- componentWillLoad()
- componentDidLoad()
- componentShouldUpdate(newValue, oldValue, propName): boolean
- componentWillRender()
- componentDidRender()
- componentWillUpdate()
- componentDidUpdate()
- render()

The appload event

In addition to component-specific lifecycle hooks, a special event called `appload` will be emitted when the app and all of its child components have finished loading. You can listen for it on the `window` object.

If you have multiple apps on the same page, you can determine which app emitted the event by checking `event.detail.namespace`. This will be the value of the namespace config option you've set in your Stencil config.

```
window.addEventListener('appload', (event) => {
  console.log(event.detail.namespace);
});
```

@Prop

@Prop


```
import { Component, Prop, h } from '@stencil/core';

@Component({
    tag: 'todo-list-item',
})
export class ToDoListItem {
    // thingToDo is 'camelCased'
    @Prop() thingToDo: string;

    render() {
        return <div>{this.thingToDo}</div>;
    }
}
```

When we use our component in a TSX file, an attribute uses camelCase:

```
<todo-list-item thingToDo={"Learn about Stencil Props"}></todo-l
```

In HTML, the attribute must use 'dash-case' like so:

```
<todo-list-item thing-to-do="Learn about Stencil Props"></todo-l
```

Types

Props can be a `boolean`, `number`, `string`, or even an `Object` or `Array`. The example below expands the `todo-list-item` to add a few more props with different types.

```
import { Component, Prop, h } from '@stencil/core';
// `MyHttpService` is an `Object` in this example
import { MyHttpService } from '../some/local/directory/MyHttpSer

@Component({
    tag: 'todo-list-item',
})
export class ToDoListItem {
    @Prop() isComplete: boolean;
    @Prop() timesCompletedInPast: number;
    @Prop() thingToDo: string;
    @Prop() myHttpService: MyHttpService;
}
```

Options

The `@Prop()` decorator accepts an optional argument to specify certain options to modify how a prop on a component behaves. `@Prop()` 's optional argument is an object literal containing one or more of the following fields:

```
export interface PropOptions {
  attribute?: string;
  mutable?: boolean;
  reflect?: boolean;
}
```

Prop Mutability (`mutable`)

A Prop is by default immutable from inside the component logic. However, it's possible to explicitly allow a Prop to be mutated from inside the component, by declaring it as mutable, as in the example below:

```
import { Component, Prop, h } from '@stencil/core';

@Component({
   tag: 'todo-list-item',
})
export class ToDoListItem {
   @Prop({ mutable: true }) thingToDo: string;

   componentDidLoad() {
      this.thingToDo = 'Ah! A new value!';
   }
}
```

Validatioin

To do validation of a Prop, you can use the **@Watch()** decorator:

```
import { Component, Prop, Watch, h } from '@stencil/core';

@Component({
   tag: 'todo-list-item',
})
export class TodoList {
  // Mark the prop as required, to make sure it is provided when
  // We want stricter guarantees around the contents of the stri
  @Prop() thingToDo!: string;

  @Watch('thingToDo')
  validateName(newValue: string, _oldValue: string) {
    // don't allow `thingToDo` to be the empty string
    const isBlank = typeof newValue !== 'string' || newValue ===
    if (isBlank) {
        throw new Error('thingToDo is a required property and ca
    };
    // don't allow `thingToDo` to be a string with a length of 1
    const has2chars = typeof newValue === 'string' && newValue.l
    if (!has2chars) {
       throw new Error('thingToDo must have a length of more tha
    };
  }
}
```

Demos

@Prop

```
import { Component, Prop, h } from '@stencil/core';

@Component({
  tag: 'todo-list',
})
export class TodoList {
  render() {
    return (
      <div>
        <h1>To-Do List Name: Stencil To Do List</h1>
        <ul>
           {/* Below are three Stencil components that are children of `todo-list`, each
representing an item on our list */}
          <todo-list-item thingToDo={"Learn about Stencil Props"}></todo-list-item>
          <todo-list-item thingToDo={"Write some Stencil Code with Props"}></todo-list-item>
          <todo-list-item thingToDo={"Dance Party"}></todo-list-item>
        </ul>
      </div>
    )
  }
}
```

```
import { Component, Prop, h } from '@stencil/core';

@Component({
    tag: 'todo-list-item',
})
export class ToDoListItem {
  @Prop() thingToDo: string;

  render() {
    return <li>{this.thingToDo}</li>;
  }
}
```

@State

[演示]


```
import { Component, State, h } from '@stencil/core';

@Component({
    tag: 'current-time',
})
export class CurrentTime {
    timer: number;

    // `currentTime` is decorated with `@State()`,
    // as we need to trigger a rerender when its
    // value changes to show the latest time
    @State() currentTime: number = Date.now();

    connectedCallback() {
        this.timer = window.setInterval(() => {
            // the assignment to `this.currentTime`
            // will trigger a re-render
            this.currentTime = Date.now();
        }, 1000);
    }

    disconnectedCallback() {
        window.clearInterval(this.timer);
    }

    render() {
        const time = new Date(this.currentTime).toLocaleTimeString();

        return (
            <span>{time}</span>
        );
    }
}
```

JSX

JSX语法

JSX JavaScript XML，一种在JavaScript中构建标签的类 XML的语法。

```
render() {
    return (
      <div>Hello World</div>
    )
  }
```

```
export class MyComponent {
    inputChanged(event) {
      console.log('input changed: ', event.target.value);
    }

    render() {
      return (
        <input onChange={(event: UIEvent) => this.inputChanged(event)}>
      )
    }
}
```

JSX语法

数据绑定

条件渲染

Slots

循环

事件句柄

多root

https://stenciljs.com/docs/templating-jsx

@Event & @Listen

Event

@Event & @Listen

There is **NOT** such a thing as *stencil events*, instead, Stencil encourages the use of DOM events. However, Stencil does provide an API to specify the events a component can emit, and the events a component listens to. It does so with the `Event()` and `Listen()` decorators.

Event Decorator

Components can emit data and events using the Event Emitter decorator.

To dispatch Custom DOM events for other components to handle, use the `@Event()` decorator.

```
import { Event, EventEmitter } from '@stencil/core';

...
export class TodoList {

  @Event() todoCompleted: EventEmitter<Todo>;

  todoCompletedHandler(todo: Todo) {
    this.todoCompleted.emit(todo);
  }
}
```

```
<todo-list onTodoCompleted={ev => this.someMethod(ev)} />
```

Listen Decorator

The `Listen()` decorator is for listening to DOM events, including the ones dispatched from `@Events`.

In the example below, assume that a child component, `TodoList`, emits a `todoCompleted` event using the `EventEmitter`.

```
import { Listen } from '@stencil/core';

...
export class TodoApp {

  @Listen('todoCompleted')
  todoCompletedHandler(event: CustomEvent<Todo>) {
    console.log('Received the custom todoCompleted event: ', eve
  }
}
```

[演示]


```
import { Component, Event, EventEmitter,Listen, h } from '@stencil/core';

@Component({
    tag: 'demo-event',
})
export class DemoEvent {
    @Event({
        eventName: 'loading',
        composed: true,
        cancelable: true,
        bubbles: true,
    }) loadingEvent: EventEmitter;

    emitLoading() {
        debugger

        const event = this.loadingEvent.emit({});
        if (!event.defaultPrevented) {
            // if not prevented, do some default handling code
        }
    }

    // @Listen('loading')
    // todoCompletedHandler(event) {
    //  debugger
    // }

    render() {
        return (
            <div>
                <button onClick={e=>this.emitLoading()}>Emit Loading</button>
            </div>
        );
    }
}
```

@Method

@Method

Method Decorator

The `@Method()` decorator is used to expose methods on the public API. Functions decorated with the `@Method()` decorator can be called directly from the element, i.e. they are intended to be callable from the outside!

> Developers should try to rely on publicly exposed methods as little as possible, and instead default to using properties and events as much as possible. As an app scales, we've found it's easier to manage and pass data through @Prop rather than public methods.

```
import { Method } from '@stencil/core';

export class TodoList {

  @Method()
  async showPrompt() {
    // show a prompt
  }
}
```

```
const todoListElement = document.querySelector('todo-list');
await todoListElement.showPrompt();
```

Styling

Styling

Shadow DOM in Stencil

The shadow DOM hides and separates the DOM of a component in order to prevent clashing styles or unwanted side effects. We can use the shadow DOM in our Stencil components to ensure our components won't be affected by the applications in which they are used.

To use the Shadow DOM in a Stencil component, you can set the `shadow` option to `true` in the component decorator.

```
@Component({
  tag: 'shadow-component',
  styleUrl: 'shadow-component.css',
  shadow: true,
})
export class ShadowComponent {}
```

```
:host {
  color: black;
}

div {
  background: blue;
}
```

Styling

Scoped CSS

An alternative to using the shadow DOM is using scoped components. You can use scoped components by setting the `scoped` option to `true` in the component decorator.

```
@Component({
  tag: 'scoped-component',
  styleUrl: 'scoped-component.css',
  scoped: true,
})
export class ScopedComponent {}
```

CSS Custom Properties

CSS custom properties, also often referred to as CSS variables, are used to contain values that can then be used in multiple CSS declarations. For example, we can create a custom property called `--color-primary` and assign it a value of `blue`.

```
:host {
  --color-primary: blue;
}
```

And then we can use that custom property to style different parts of our component

```
h1 {
  color: var(--color-primary);
}
```

Functional Components

Functional Components

Functional components are quite different to normal Stencil web components because they are a part of Stencil's JSX compiler. A functional component is basically a function that takes an object of props and turns it into JSX.

```
const Hello = props => <h1>Hello, {props.name}!</h1>;
```

When the JSX transpiler encounters such a component, it will take its attributes, pass them into the function as the `props` object, and replace the component with the JSX that is returned by the function.

```
<Hello name="World" />
```

Functional components also accept a second argument `children`.

```
const Hello = (props, children) => [
  <h1>Hello, {props.name}</h1>,
  children
];
```

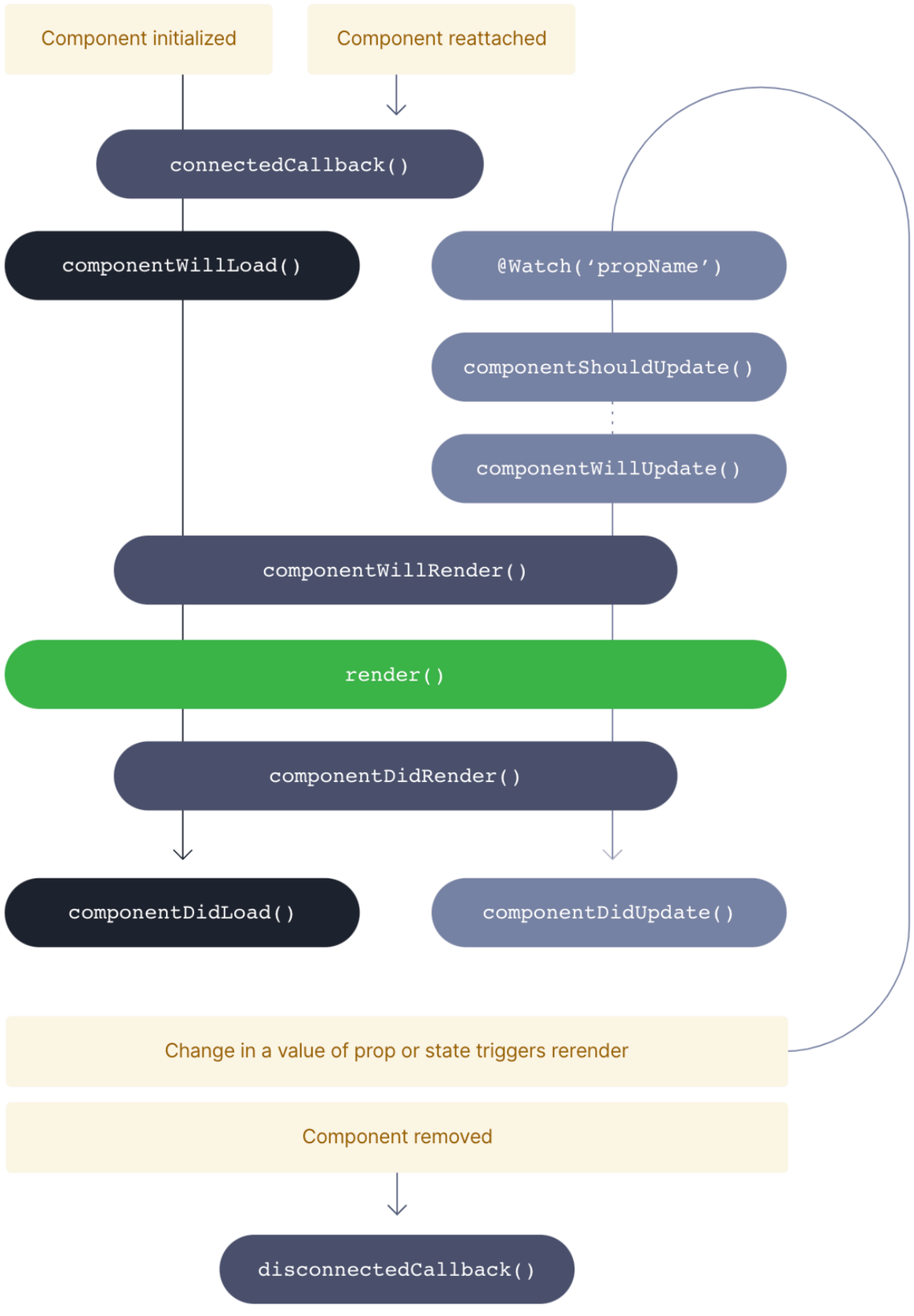
The JSX transpiler passes all child elements of the component as an array into the function's `children` argument.

```
<Hello name="World">
  <p>I'm a child element.</p>
</Hello>
```

Lifecycle

Lifecycle

https://stenciljs.com/docs/component-lifecycle



数据绑定

事件处理


```
@Component({
    tag: 'my-name',
    styleUrl: 'my-name.scss'
  })
  export class MyName {

    @State() value: string;

    handleSubmit(e) {
      e.preventDefault()
      console.log(this.value);
    }

    handleChange(event) {
      this.value = event.target.value;
    }

    render() {
      return (
        <form onSubmit={(e) => this.handleSubmit(e)}>
          <label>
            Name:
            <input type="text" value={this.value} onInput={() =>
this.handleChange(event)} />
          </label>
          <input type="submit" value="Submit" />
        </form>
      );
    }
  }
```

组件生命周期

代码分析


```
import { Component, Prop,ComponentDidLoad,ComponentDidUnload,ComponentWillLoad,ComponentWillUpdate } from '@stencil/
core';

@Component({
  tag: 'my-component',
  styleUrl: 'my-component.css',
  shadow: false
})
export class MyComponent implements ComponentDidLoad ,ComponentDidUnload,ComponentWillLoad,ComponentWillUpdate{

  @Prop() first: string;
  @Prop() last: string;
   componentWillLoad(){
     console.log("Component will to be rendered")
   }
  componentDidLoad() {
    console.log('Component has been rendered');
  }
  componentWillUpdate() {
    console.log('Component will update and re-render');
  }
  componentDidUpdate() {
    console.log('Component did update');
  }
  componentDidUnload() {
    console.log('Component removed from the DOM');
  }

  render() {
    return (
      <div >
        Hello, World! I'm {this.first} {this.last}
      </div>
    );
  }
}
```

Stencil单元测试

Stencil单元测试

配置

组件渲染测试

断言检查

检测元素

【练习】Demos

各场景练习

自定义布局组件

对话框组件

用户列表（服务请求）组件

内容回顾