



Lit

Google

Libreria per la creazione di web components

ELEONORA ROCCHI

- 2022 -

La creazione di un componente Lit implica una serie di concetti:

- ➔ Viene implementato come elemento personalizzato e registrato nel browser.
- ➔ Ha un metodo *render* chiamato per eseguire il rendering del contenuto del componente, nel quale si definisce un template.
- ➔ Le proprietà mantengono lo stato del componente. La modifica di una o più proprietà reattive dei componenti, attiva un ciclo di aggiornamento che si occupa di ri-eseguire il rendering del componente.
- ➔ Un componente può definire stili incapsulati per controllare il proprio aspetto.
- ➔ Lit definisce una serie di callback di cui è possibile eseguire l'override per agganciarsi al ciclo di vita del componente, ad esempio per eseguire codice quando l'elemento viene aggiunto a una pagina o ogni volta che il componente viene aggiornato.

Come si definisce un componente Lit

Va creata una classe che estende LitElement e va registrata nel browser:



```
@customElement('simple-greeting')  
export class SimpleGreeting extends LitElement { /* ... */ }
```


Il decoratore `@customElement` è un'abbreviazione per chiamare `customElements.define`, che registra una classe di elementi personalizzati con il browser e la associa a un nome di elemento (in questo caso, `simple-greeting`).

Usando JavaScript, o senza usare decoratori, si può chiamare `define()` direttamente:



```
export class SimpleGreeting extends LitElement { /* ... */ }  
customElements.define('simple-greeting', SimpleGreeting);
```


**Un componente Lit è un
elemento HTML**

Quando si definisce un componente Lit,
si sta definendo un elemento HTML personalizzato
che si può poi usare come un qualsiasi elemento integrato:



```
<simple-greeting name="Markup"></simple-greeting>
```



```
const greeting = document.createElement('simple-greeting');
```


La classe base `LitElement` è una sottoclasse di `HTMLElement`, quindi un componente Lit eredita tutte le proprietà dei metodi standard di `HTMLElement`.

In particolare, `LitElement` eredita da `ReactiveElement`, che implementa proprietà reattive, e a sua volta eredita da `HTMLElement`.

Sappiamo che TypeScript è in grado di dedurre la classe di un elemento HTML restituito da alcune API DOM in base al nome del tag.

Ad esempio, `document.createElement('img')` restituisce un'istanza `HTMLImageElement` con una proprietà `src: string`.

Si può ottenere lo stesso comportamento sugli elementi personalizzati aggiungendo a `HTMLElementTagNameMap` come segue:

```
● ● ●  
  
@customElement('my-element')  
export class MyElement extends LitElement {  
  @property({type: Number})  
  aNumber: number = 5;  
  /* ... */  
}  
  
declare global {  
  interface HTMLElementTagNameMap {  
    "my-element": MyElement;  
  }  
}
```


In questo modo, il codice seguente esegue correttamente i controlli di tipo:



```
const myElement = document.createElement( 'my-element' );  
myElement.aNumber = 10;
```


Conviene sempre aggiungere una voce `HTMLElementTagNameMap` agli elementi creati in TypeScript e pubblicare i tipi di `.d.ts` nel pacchetto npm.

Cosa fa il Render

È possibile aggiungere un template al componente per definire cosa dovrebbe essere visualizzato.

I template possono includere *espressioni*, che sono placeholder per del contenuto dinamico.

Per definire un template per un componente Lit, va aggiunto un metodo `render()`:

```
import {LitElement, html} from 'lit';
import {customElement} from 'lit/decorators.js';


@customElement('my-element')
class MyElement extends LitElement {

  render(){
    return html`<p>Hello from my template.</p>`;
  }
}
```


Si scrive il template in HTML all'interno di un Javascript tagged template literal usando la funzione tag `html` di Lit.

I template literal sono un tipo speciale di simboli di stringa che possono contenere espressioni Javascript e si estendono su più righe. Usano i caratteri backtick invece delle doppie virgolette. Accetta all'interno anche espressioni arbitrarie.

Questi template possono essere utilizzati anche come funzioni, mettendo una parola chiave davanti alla stringa del template per "taggarla".



```
function useless(strings, ...values) {  
  return 'I render everything useless.';  
}  
  
let name = 'Benedict';  
let occupation = 'being awesome';  
  
let sentence = useless`Hi! I'm ${ name } and I'm busy at ${ occupation  
}`;  
console.log(sentence);  
// I render everything useless.
```


I template Lit possono includere espressioni JavaScript.

È possibile utilizzare le espressioni per impostare contenuto di testo, attributi, proprietà e listener di eventi.

Il metodo `render()` può anche includere qualsiasi JavaScript, ad esempio è possibile creare variabili locali da utilizzare nelle espressioni.

In genere, il metodo `render()` del componente restituisce un singolo oggetto `TemplateResult` (lo stesso tipo restituito dalla funzione `tag html`).

Tuttavia, può restituire tutto ciò che Lit può renderizzare:

- Valori primitivi come stringa, numeri o booleani.
- Oggetti `TemplateResult` creati dalla funzione `html`.
- Nodi DOM.
- Matrici o iterabili di uno qualsiasi dei tipi supportati.

Per sfruttare al meglio il modello di rendering funzionale di Lit, il metodo `render()` dovrebbe:

- ➔ Evitare di modificare lo stato del componente.
- ➔ Evitare di produrre effetti collaterali.
- ➔ Utilizzare solo le proprietà del componente come input.
- ➔ Restituire lo stesso risultato quando vengono assegnati gli stessi valori di proprietà.
- ➔ Evitare di effettuare aggiornamenti DOM al di fuori di `render()`.
- ➔ Esprimere il modello del componente in funzione del suo stato e acquisire il suo stato nelle proprietà.

Si possono comporre template Lit da altri template.

Ad esempio, si può comporre un template per un componente chiamato `<my-page>` da template più piccoli per l'intestazione, il piè di pagina e il contenuto principale della pagina.

```
import {LitElement, html} from 'lit';
import {customElement, property} from 'lit/decorators.js';

@customElement('my-page')
class MyPage extends LitElement {

  @property({attribute: false})
  article = {
    title: 'My Nifty Article',
    text: 'Some witty text.',
  };

  headerTemplate() {
    return html`<header>${this.article.title}</header>`;
  }

  articleTemplate() {
    return html`<article>${this.article.text}</article>`;
  }

  footerTemplate() {
    return html`<footer>Your footer here.</footer>`;
  }

  render() {
    return html`
      ${this.headerTemplate()}
      ${this.articleTemplate()}
      ${this.footerTemplate()}
    `;
  }
}
```


In questo esempio, i singoli template sono definiti come metodi di istanza, quindi una sottoclasse potrebbe estendere questo componente e sovrascrivere uno o più template.

Si può anche comporre un template importando altri elementi e utilizzandoli nel template stesso.

```
import {LitElement, html} from 'lit';
import {customElement} from 'lit/decorators.js';

import './my-header.js';
import './my-article.js';
import './my-footer.js';

@customElement('my-page')
class MyPage extends LitElement {
  render() {
    return html`
      <my-header></my-header>
      <my-article></my-article>
      <my-footer></my-footer>
    `;
  }
}
```




```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <script src="./my-page.js" type="module"></script>
  <title>lit-element code sample</title>
</head>
<body>
  <my-page></my-page>
</body>
</html>
```


Un componente Lit esegue il rendering del suo modello inizialmente quando viene aggiunto al DOM su una pagina.

Dopo il rendering iniziale, qualsiasi modifica alle proprietà reattive del componente attiva un ciclo di aggiornamento, ri-eseguendo il rendering del componente.

Lit aggiorna in batch per massimizzare le prestazioni e l'efficienza.

L'impostazione di più proprietà contemporaneamente attiva un solo aggiornamento, eseguito in modo asincrono al momento del microtask.

Durante un aggiornamento, solo le parti del DOM che cambiano vengono renderizzate nuovamente.

Sebbene i modelli Lit assomiglino all'interpolazione di stringhe, Lit analizza e crea HTML statico una volta, quindi aggiorna solo i valori modificati nelle espressioni, rendendo gli aggiornamenti molto efficienti.

Lit usa il shadow DOM per incapsulare il DOM che un componente esegue il rendering.

Lo shadow DOM consente a un elemento di creare il proprio albero DOM isolato, separato dall'albero del documento principale.

È una caratteristica fondamentale delle specifiche dei web component che consente l'interoperabilità, l'incapsulamento dello stile e altri vantaggi.

Cos'è lo shadow DOM

Lo shadow DOM consente agli sviluppatori Web di creare DOM e CSS compartimentalizzati per i web component.

Lo shadow DOM rimuove alcune fragilità tipiche della creazione di app Web, che derivano dalla natura globale di HTML, CSS e JS.

Ad esempio, quando si utilizza un nuovo ID/classe HTML, non è possibile stabilire se entrerà in conflitto con un nome esistente utilizzato dalla pagina.

Nel CSS si moltiplicano gli !important, i selettori di stile crescono fuori controllo e le prestazioni possono risentirne.

Lo shadow DOM corregge CSS e DOM, introducendo gli *scoped style*.

Senza strumenti o convenzioni di denominazione, si possono raggruppare CSS con markup, nascondere i dettagli di implementazione e creare componenti autonomi in JavaScript vanilla.

Quando il browser carica una pagina web, tra le altre cose, trasforma l'HTML in un documento live.

Per comprendere la struttura della pagina, il browser analizza l'HTML (stringhe di testo statiche) in un modello di dati (oggetti/nodi).

Il browser preserva la gerarchia dell'HTML creando un albero di questi nodi: il DOM, che è una rappresentazione dal vivo di una pagina. A differenza dell'HTML statico, i nodi prodotti dal browser contengono proprietà, metodi e possono essere manipolati da programmi utilizzando JavaScript.

Lo Shadow DOM è semplicemente un normale DOM con due differenze:

1. come viene creato/utilizzato
2. come si comporta rispetto al resto della pagina.

Normalmente, si creano nodi DOM e si aggiungono come figli di un altro elemento.

Con Shadow DOM, si crea un albero DOM con ambito che è collegato all'elemento, ma separato dai suoi figli effettivi. Questo sottoalbero con ambito è chiamato albero ombra. L'elemento a cui è collegato è il suo shadow host. Tutto ciò che si aggiunge nelle shadow diventa locale per l'elemento di hosting, incluso `<style>`.

Questo è il modo in cui lo shadow DOM ottiene il css style scope.