SUPSI

Human-computer interaction

HCI 2018-2019

anno accademico 2018-2019

Nicola Rizzo - nicola.rizzo@supsi.ch

SUPSI

Gestione delle dipendenze

Gestione delle dipendenze - il problema

- Prima dell'utilizzo di applicazioni RIA, le dipendenze venivano caricate nel browser come lista di script (non sempre) in fondo alla pagina
- Questo provocava performance subottimali e una estrema difficoltà di far convivere diversi software nella stessa applicazione
    - Alcune librerie per esempio erano note per non essere compatibili fra loro, poiché dichiaravano variabili globali omonime o sovrascrivevano gli stessi oggetti nativi!

Dipendenze e conflitti

- Oggi è comunemente ritenuta cattiva programmazione sovrascrivere i Native Objects (Number, String, Image, Date…).

- L’unica eccezione tollerata sono i **polyfill**, script in grado di aggiungere funzionalità specifiche a un runtime che non le supporta o di correggere bug

- Si sono trovati dei metodi alternativi all’(ab)uso di variabili globali per la gestione delle dipendenze; questi hanno infine portato all’emergere di vari standard

revealing module pattern

```
let myModule = (function(){
  // private
  let sheeps = 0;
  let increment = function(){
    ++sheeps
  };
  let decrement = function(){
    sheeps = Math.min(sheeps, sheeps - 1);
  };
  let getSheeps = function(){
    return sheeps;
  };
  return {
    getSheeps,
    increment,
    decrement
  }
})();
```

Parte privata

Parte pubblica

revealing module pattern, spiegazione

SUPSI

- Il pattern è reso possibile dal fatto che in JavaScript possano essere formate delle **closure**: una closure è la combinazione di una funzione e degli oggetti a cui può accedere (l'ambito lessicale in cui la funzione è dichiarata)

- Sembra controintuitivo, ma gli oggetti restano disponibili anche dopo l'esecuzione della funzione stessa

- Questo pattern garantisce di avere privacy e isolamento in un linguaggio che non ha il concetto di variabili private

Moduli e gestione delle dipendenze - divide et impera

SUPSI

- In qualsiasi software di una certa complessità è necessario suddividere i sorgenti in moduli che svolgano funzioni differenti e abbiano complessità ridotta

- In questo modo è più facile testare le singole unità del software

- Molti linguaggi offrono nativamente supporto per questa operazione

- Con l'aumentare della complessità delle interfacce utente è stato necessario inventare un meccanismo analogo anche per le applicazioni scritte con tecnologie web, anche per poter gestire in maniera automatica eventuali dipendenze circolari e l'ordine di caricamento

Moduli - peculiarità

- Essendo l'applicazione scaricata nel client dell'utente per essere interpretata, si sono sviluppate alcune modalità alternative
  - AMD (Asynchronous Module Definition)
  - CommonJS
  - ES6 (ES2015) modules
  - UMD (Universal Module Definition)

AMD

- Con l'approccio AMD, il client è in grado di richiedere **a runtime** le dipendenze di cui ha bisogno in modo asincrono per offrire determinate funzionalità

  - **PRO:** il caricamento iniziale dell'applicazione è più veloce, quando le feature vengono aggiunte incrementalmente

  - **CONTRO:** quando viene richiesto il caricamento del nuovo modulo, l'utente deve aspettare (a meno che non si riesca a fare un caricamento nei tempi morti, ma il meccanismo diventa complesso). La sintassi di AMD è più verbosa di quella usata da CommonJS e ES2015.

  - **CASI D'USO:** funzionalità o pezzi di interfaccia non usati nella totalità dei casi. Può far parte del modulo anche un frammento di interfaccia, non solo il codice che la utilizza (esempio la chat in Gmail). Non indicato per caricare le varie parti dell'applicazione a startup

CommonJS

- Le dipendenze vengono caricate in maniera sincrona, tramite un transpiler (webpack, browserify…); di fatto c’è un passo di compilazione preventiva, quindi non esiste il problema di blocco del main thread

  - **PRO:** le dipendenze richieste sono tutte presenti al caricamento dell’applicazione

  - **CONTRO:** nel caso di risorse da scaricare dalla rete, viene potenzialmente importato nel client anche codice che non verrà utilizzato, comprese le risorse a esso collegate (template, immagini…)

  - **CASI D’USO:** il core con le parti fondamentali di una applicazione. Con una unica chiamata HTTP(S) si hanno prestazioni migliori che con N chiamate

ES2015

- Molto simile a CommonJS, ha una sintassi differente. Supportata dai transpiler e da Node.js dalla versione 10.11 (sperimentalmente, dietro flag)

UMD

- Permette il caricamento in entrambe le modalità, sincrona e on-demand (asincrona)
- **PRO:** unisce i pro dei metodi precedenti
- **CONTRO:** la sintassi è molto verbosa
- **CASI D'USO:** applicazioni molto complesse in cui la codebase sia molto elevata

transpiler, package manager e module bundlers

SUPSI

- Durante il corso utilizzeremo webpack come strumento per impacchettare il software che scriveremo

- Il bundler si occuperà, anche tramite plugins, di

  - fornire meccanismi AMD / CommonJS
  - tradurre ES2015 (o altri linguaggi) in una versione di JavaScript adatta ai vecchi browser
  - compilare da SCSS a CSS (!)
  - caricare templates, immagini e altre risorse
  - importare le dipendenze esterne acquisite mediante npm, impedendo caricamenti multipli delle medesime dipendenze

SUPSI

Sintassi - es2015

```
const HelloButton = {
…
};

export { HelloButton };


// in un altro modulo

import { HelloButton } from ‘./button';
```

Sintassi - CommonJS

```
const HelloButton = {
…
};

module.exports = HelloButton;


// in un altro modulo

const HelloButton = require('./button');
```

Sintassi - differenze

- La differenza più evidente è che la sintassi con *require* permette, qualora si stia utilizzando NodeJS (quindi solo sul server), di caricare le dipendenze on-demand

- In alcuni casi può essere meglio usare la sintassi con require per motivi di praticità (per esempio nell'esecuzione dei test)

import on-demand

- Utile quando parte dell'applicazione / interfaccia può essere caricata successivamente all'avvio (lazy loading) per garantire un caricamento iniziale più veloce

```
import('../anothermodule').then((m) => {
  console.log('the Answer is =', m.AnotherModule.answer);
  div.textContent = m.AnotherModule.answer;
  document.body.appendChild(div);
});
```

Promises 1 / 2

- I caricamenti lazy devono essere eseguiti in maniera **asincrona** per non bloccare il thread principale (l'unico!)

- Sarebbe sufficiente una funzione di callback, che però porterebbe con sé altri problemi:

  - continua indentazione del codice e scarsa leggibilità (Pyramid of Doom)

  - difficoltà di gestione

  - impossibilità di eseguire try...catch

Promises 2 / 2

- Le promises consentono di superare tutto questo
- L’esecuzione di then avviene quando il dato è pronto, come una callback, con la differenza che promises asincrone **possono essere concatenate o eseguite contemporaneamente** con estrema facilità

Promises 3 / 2

```
let myFn = function () {
  let p = new Promise((resolve, reject) => {
    // something wrong
    if (err) {
      return reject(err);
    }

    // ...
    // something asynchronous
    let value = 1;
    resolve(value);

  });
  return p;
}
```

Resolve e reject

- resolve deve essere invocata con il valore da restituire nella then()

- reject() deve essere restituita in caso di errore, possibilmente con una spiegazione relativa all'eccezione

then e catch

```
myFn().then((v) => {
  console.log(v); // 1, se tutto è andato bene
}).catch((e) => {
  console.error('Si è verificato un errore');
});
```