ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea in Ingegneria Elettronica, Informatica e Telecomunicazioni

PROGRAMMAZIONE ASINCRONA IN JAVASCRIPT

Elaborata in: Sistemi Operativi

Relatore:
Prof. ALESSANDRO RICCI

Co-relatore:
Prof. ANDREA SANTI

Presentata da:
RICCARDO DRUDI

SESSIONE II
ANNO ACCADEMICO 2012-2013

PAROLE CHIAVE

Programmazione Asincrona

Javascript

Eventi

jQuery

Node.js

alla mia famiglia

Indice

Introduzione

L'obiettivo della tesi è esplorare gli aspetti fondamentali che riguardano la programmazione asincrona, in particolare in JavaScript, analizzando le librerie e i metodi più diffusi per risolvere problemi tipici di asincronicità. Il motivo dello studio di JavaScript è il grande utilizzo di tale linguaggio per la costruzione delle moderne web-app. Per la realizzazione di questo elaborato si è partiti dallo studio dell'evoluzione del web, dal Web 1.0 al web 2.0 e dall'evoluzione delle web app in questo contesto, successivamente si è raffinata la ricerca allo studio più specifico di Javascript e, di conseguenza, ai suoi costrutti ed ai suoi stili di programmazione, come la programmazione asincrona e la programmazione ad eventi.

Capitolo 1

Evoluzione del Web

1.1 Dalla nascita al Web 1.0

1.1.1 I primi passi di Internet

L'origine di Internet risale agli anni sessanta per merito degli Stati Uniti che misero a punto, durante la guerra fredda, un nuovo sistema di controspionaggio. Già in quegli anni gli scienziati teorizzavano una rete di computer mondiale ad accesso pubblico, ma per renderla realizzabile bisognerà attendere il 1991 quando il governo degli Stati Uniti emana la *High performance computing act*, una legge che permetterà di ampliare questa rete a livello globale per merito di iniziative private e di destinarla al mondo scientifico.

ARPANET

ARPANET è considerato il progenitore di Internet. La rete venne fisicamente realizzata nel 1969 collegando 4 nodi: l'Università della California di Los Angeles, l'SRI di Stanford, l'Università della California di Santa Barbara, e l'Università dello Utah. Per definire le caratteristiche della rete vennero introdotti dei documenti, che sono i precursori dei protocolli di rete. La rete dei giorni nostri non è altro che un'estensione di questa prima rete, creata sotto il nome di ARPANET.

Da ARPANET a Internet

In pochi anni ARPANET si espanse notevolemte anche oltreoceano in Francia e in Inghilterra. Ci si stava preparando all'avvento di Internet. Vennero infatti definiti il *Transmission Control Protocol* (TCP) e l'*Internet Protocol* (IP) che diedero definitivamente il via all'Internet che conosciamo oggi.

Nascita del Web

Il dispiegamento delle potenzialità di Internet e la sua progressiva diffusione popolare sono però frutto dello sviluppo del WWW, il World Wide Web, un sistema per la condivisione di informazioni in ipertesto sviluppato da Tim Berners-Lee nel 1990 presso il CERN (Centro Europeo per la Ricerca Nucleare).

Tim Berners-Lee e Robert Cailliau nel 1990 misero a punto il protocollo HTTP e una prima specifica del linguaggio HTML, sulla base dei quali sono stati realizzati un primo programma browser/editor ipertestuale per il WWW, utilizzato all'interno del CERN nel 1991. Nel 1993 uscì la release Mosaic.

Con Mosaic nacque la Mosaic Communications, che poi prese il nome di Netscape Communication e che creò nel 1994 il primo browser commerciale, Netscape Navigator, che successivamente è stato reso disponibile online. Nel 1995, la Sun Microsystem progettò il linguaggio di programmazione Java, che permise di eseguire programmi scaricati da Internet in sicurezza grazie alla tecnologia degli Applet.

1.1.2 Web 1.0

Internet è nato nella mente e nelle utopie di tante persone nei primi anni sessanta, ma per la massa, per le imprese e per il mondo nel complesso, Internet nasce effettivamente nel 1995. È dall'ampia diffusione delle tecnologie del World Wide Web che si comincia a parlare sempre di più del commercio elettronico anche per gli utenti finali e non solo per le transazioni fra grandi imprese.

Il Web 1.0 è una fase iniziale dell'evoluzione concettuale del Word Wide Web, centrata intorno ad un approccio top-down per l'uso del web e per l'interfaccia utente.
Socialmente gli utenti possono soltanto visualizzare le pagine web, ma non possono contribuire in alcun modo al suo contenuto. Tecnicamente le informazioni contenute nei siti Web 1.0 non permettono una modifica esterna. Pertanto l'informazione non è dinamica e può essere aggiornata solo dal Webmaster.

Le caratteristiche principali del Web 1.0 sono le seguenti:

- **Struttura informativa gerarchica**: esistevano due utilizzatori della Rete, quelli che fornivano notizie e quelli che ne fruivano.
- **Pagine Web statiche**: le pagine che si adattavano al tipo di schermo o al tipo di device erano ancora inesistenti. I siti si programmavano in HTML puro e non si conoscevano i CMS.
- **Basso Rischio**: il rischio era quasi inesistente in quanto mancava un canale di ritorno che poteva scombinare ciò che era stato pianificato.
- **Comunicazione monodirezionale**: essendo le community limitatissime e non esistendo i social network, la comunicazione andava dall'alto verso il basso.

1.2 HTML

Tra le tecnologie di costruzione di siti Web 1.0 è di fondamentale importanza il linguaggio HTML (HyperText Markup Language). HTML non è un linguaggio di programmazione, infatti non ha meccanismi che consentono di prendere delle decisioni, non è in grado di compiere delle iterazioni e non ha altri costrutti propri della programmazione. HTML è invece un linguaggio di contrassegno o di markup cioè, attraverso degli appositi marcatori detti tag, permette di indicare come disporre gli elementi all'interno di una pagina web.
Il contenuto delle pagine web solitamente consiste dunque di un documento HTML e dei file ad esso correlati che un web browser scarica da uno o più web server per elaborarli, interpretando il codice sorgente, al fine di

generare la visualizzazione, sullo schermo del computer-client, della pagina desiderata, grazie al motore di rendering del browser stesso.

Il linguaggio HTML, la cui sintassi è stabilita dal World Wide Web Consortium (W3C), è di pubblico dominio ed è basato su un altro linguaggio avente scopi più generici: SGML.
È stato sviluppato verso la fine degli anni ottanta da Tim Berners-Lee al CERN di Ginevra assieme al noto protocollo HTTP che supporta invece il trasferimento di documenti in tale formato. Verso il 1994 ha avuto una forte diffusione in seguito ai primi utilizzi commerciali del web.

Attualmente i documenti HTML sono in grado di incorporare molte tecnologie, che offrono la possibilità di aggiungere al documento ipertestuale controlli più sofisticati sulla resa grafica, interazioni dinamiche con l'utente, animazioni interattive e contenuti multimediali. Si tratta di linguaggi come CSS, JavaScript e jQuery, XML, JSON, o di altre applicazioni multimediali di animazione vettoriale o di streaming audio o video.

1.2.1 Descrizione Generale

Ogni documento ipertestuale scritto in HTML deve essere contenuto in un file la cui estensione è tipicamente .htm o .html. Il componente principale della sintassi di questo linguaggio è l'elemento, inteso come struttura di base a cui è delegata la funzione di formattare i dati o indicare al browser delle informazioni, ogni elemento è racchiuso all'interno di marcature dette tag.

Un documento HTML comincia con l'indicazione della definizione del tipo di documento (Document Type Definition o DTD), la quale segnala al browser l'indirizzo URL delle specifiche HTML utilizzate per il documento, indicando quindi, implicitamente, quali elementi, attributi ed entità si possono utilizzare e a quale versione di HTML si fa riferimento. Di fatto questa informazione serve al browser per identificare le regole di interpretazione e visualizzazione appropriate per lo specifico documento. Questa definizione deve pertanto precedere tutti i tag relativi al documento stesso.

Dopo il DTD, il documento HTML presenta una struttura ad albero annidato, composta da sezioni delimitate da tag opportuni che al loro interno contengono a loro volta sottosezioni più piccole, sempre delimitate da tag.

La struttura più esterna è quella che delimita l'intero documento, eccetto la DTD, ed è all'interno del tag html. All'interno dei tag html lo standard prevede sempre la definizione di due sezioni ben distinte e disposte in sequenza ordinata:

- la sezione di intestazione o header, delimitata dal tag head, che contiene informazioni di controllo normalmente non visualizzate dal browser, con l'eccezione di alcuni elementi

- la sezione del corpo o body, delimitata dal tag body, che contiene la parte informativa vera e propria, ossia il testo, le immagini e i collegamenti che costituiscono la parte visualizzata dal browser.

Al di sotto di questa suddivisione generale, lo standard non prevede particolari obblighi per quanto riguarda l'ordine e il posizionamento delle ulteriori sottosezioni all'interno dell'header o del body, a parte l'indicazione del rispetto dei corretti annidamenti (le sottosezioni non si devono sovrapporre, ossia ogni sottosezione deve essere chiusa prima di iniziare la sottosezione successiva), lasciando così completa libertà allo sviluppatore o al progettista per quanto riguarda la strutturazione e l'organizzazione successive.

1.3 Il Web 2.0

Si indica come Web 2.0 l'insieme di tutte quelle applicazioni online che permettono un elevato livello di interazione tra il sito web e l'utente come i blog, i forum, le chat, i wiki, le piattaforme di condivisione di media come Flickr, YouTube, Vimeo, i social network come Facebook, Myspace, Twitter, Google+, Linkedin, Foursquare, ecc. ottenute tipicamente attraverso opportune tecniche di programmazione Web e relative applicazioni Web afferenti al paradigma del Web dinamico in contrapposizione al cosiddetto Web statico o Web 1.0.

1.3.1 Concetti

Il Web 2.0 può essere descritto in 3 parti:

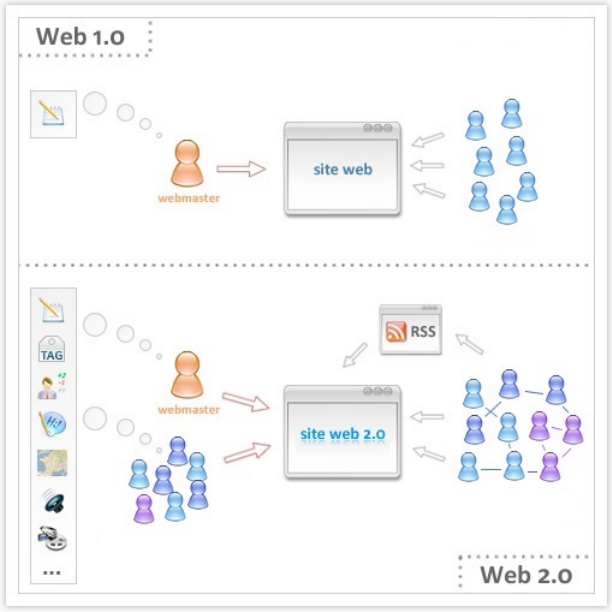
- **Rich Internet application (RIA)**: ha portato le applicazioni dal desktop al browser da un punto di vista grafico e da un punto di vista dell'usabilità. Alcune parole d'ordine relativi alla RIA sono Ajax e Flash.

- **Web-oriented architecture (WOA)**: è fondamentale nel Web 2.0 e definisce come le applicazioni Web 2.0 espongono le loro funzionalità in modo che altre applicazioni le possano sfruttare. In questo modo un'applicazione integra le funzionalità di altre applicazioni, rendendo l'esperienza dell'utente molto più ricca. Esempi sono i feed RSS, Web Services, mash-up.

- **Social Web**: definisce la tendenza del Web 2.0 ad interagire molto di più con l'utente finale.

1.3.2 Generalità

Da un punto di vista strettamente di tecnologia di rete, il Web 2.0 è del tutto equivalente al Web 1.0, in quanto l'infrastruttura a livello di rete continua ad essere costituita da TCP/IP e HTTP e l'ipertesto è ancora il concetto base delle relazioni tra i contenuti. La differenza, più che altro, sta nell'approccio con il quale gli utenti si rivolgono al Web, che passa fondamentalmente dalla semplice consultazione (seppure supportata da efficienti strumenti di ricerca, selezione e aggregazione) alla possibilità di contribuire popolando e alimentando il Web con propri contenuti.



Il Web 2.0 costituisce anzitutto un approccio filosofico alla rete che ne connota la dimensione sociale, della condivisione, dell'autorialità rispetto alla mera fruizione: sebbene dal punto di vista tecnologico molti strumenti della rete possano apparire invariati (come forum, chat e blog, che preesistevano già nel web 1.0) è proprio la modalità di utilizzo della rete ad aprire nuovi scenari fondati sulla compresenza nell'utente della possibilità di fruire e di creare/modificare i contenuti multimediali.

Gli utenti possono fornire i dati che sono su un sito Web 2.0 e esercitare un certo controllo su tali dati. Si dice che il sito ha una "architettura della partecipazione" che incoraggia gli utenti ad aggiungere valore alla domanda.

1.3.3 Tecnologie

Le tecnologie utilizzate lato client dai siti web 2.0 includono diversi framework di Ajax e JavaScript come YUI Library, Dojo Toolkit, MooTools, jQuery, Extjs e Prototype.

I dati recuperati da una richiesta Ajax sono tipicamente formattati in formato XML o JSON (JavaScript Object Notation). Un programmatore

può facilmente utilizzare questi formati per trasmettere i dati strutturati all'applicazione web. Quando questi dati vengono ricevuti, il programma JavaScript allora utilizza il Document Object Model (DOM) per aggiornare dinamicamente la pagina web sulla base dei nuovi dati, consentendo rapidità e interattività agli occhi dell'utente. In breve, utilizzando queste tecniche, i progettisti Web possono costruire nelle pagine delle vere e proprie applicazioni desktop. Per esempio, Google Docs utilizza questa tecnica per creare un word processor web based.

Lato server invece, le tecnologie più utilizzate sono PHP, Ruby, Perl, Python, così come JSP e ASP.NET. Questi linguaggi sono utilizzati dagli sviluppatori per mostrare dei dati in modo dinamico utilizzando le informazioni dai file e dai database. Per condividere i propri dati con altri siti, un sito web deve essere in grado di generare l'output in formati leggibili come XML (Atom, RSS, ecc) e JSON. Quando i dati di un sito sono disponibili in uno di questi formati, un altro sito web può utilizzare tali dati per integrare una parte della funzionalità del sito in sé, che collega le due cose insieme. Questo modello di progettazione rappresenta un segno distintivo della filosofia del movimento Web 2.0.

1.3.4 Applicazioni Web-Based

Attraverso l'utilizzo di linguaggi di scripting come Javascript, degli elementi dinamici e dei fogli di stile (CSS) per gli aspetti grafici, si possono creare delle vere e proprie applicazioni web che si discostano dal vecchio concetto di semplice ipertesto e che puntano a somigliare ad applicazioni tradizionali per computer.

Questo modello applicativo è divenuto piuttosto popolare alla fine degli anni novanta, in considerazione della possibilità per un client generico di accedere a funzioni applicative residenti su un application server, utilizzando come terminali utente normali web browser (i client finalizzati unicamente a collegarsi come terminali di web-application vengono chiamati sovente thin client).

Infatti l'opportunità di aggiornare ed evolvere a costo ridotto il proprio

applicativo, senza essere costretti a distribuire numerosi aggiornamenti ai propri clienti attraverso supporti fisici, ha reso la soluzione piuttosto popolare per molti produttori software.

Più di recente colossi come Google e Microsoft hanno implementato interi pacchetti applicativi per office automation, tradizionalmente venduti in modo distribuito su supporti CD-ROM, e che ora si stanno velocemente trasformando a tutti gli effetti in webapps (es. Google Docs). Il filone delle Web Applications comprende molte delle applicazioni ad uso business o enterprise ovvero aziendali.

Caratteristiche

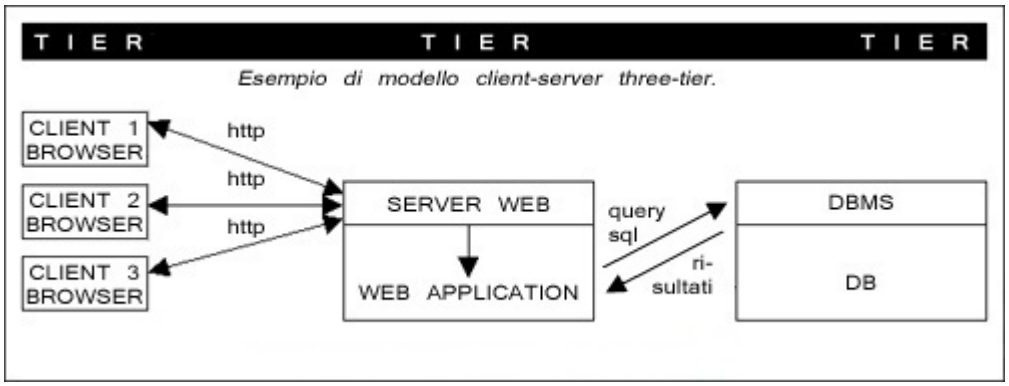
Per sua natura una webapp può presentarsi con diverse strutture ed organizzazioni logiche, poiché di fatto racchiude in sé, allo stesso tempo, un modello tecnico ed una filosofia di sviluppo. Tuttavia, sul piano dell'informatica teorica è possibile riconoscere una strutturazione tipica su tre livelli (architettura three-tier) che va a mappare l'architettura a livello fisico-infrastrutturale di un sistema informatico sul quale l'applicazione web è presente e viene eseguita.

Nella maggioranza dei casi è infatti possibile identificare:

- un primo livello associabile al terminale di fruizione, visualizzazione o presentazione a favore dell'utente (front-end) attraverso il web browser del client (tìpicamente tramite pagine HTML e CSS o utilizzando framework).

- un secondo livello costituito dal motore applicativo, ovvero un core applicativo (logica di business o back-end), presente tipicamente su un application server e costituito da codice sorgente in un qualche linguaggio di sviluppo dinamico lato-server (per es. PHP, ASP, ASP.NET, un qualche CGI, servlet/JSP/Java, ecc.).

- un terzo eventuale livello riconducibile al motore database associato (per es. MySQL, MSSql, Oracle, DB2 ecc.) per la gestione della persistenza dei dati e la loro interrogazione.



Ricapitolando, il web browser del client invia le proprie richieste al livello intermedio, ovvero al motore applicativo dinamico del web server, che da una parte interpreta e gestisce le interrogazioni al motore DBMS e dall'altra genera il risultato in un output diretto allo stesso browser, che lo interpreta e lo restituisce all'utente sotto forma di pagine Web.

Le applicazioni web-based oggi concorrono ad implementare in tutto o in parte soluzioni software quali Social Network, CMS, Webmail, e-commerce, web forum, blog, MMORPG, gestionali di vario tipo e molto altro ancora.

Un esempio tipico e didattico è un form di login, che rappresenta la parte di presentazione all'utente dell'applicazione, una parte di elaborazione che fa il controllo dei dati acquisiti dal sistema con quelli presenti nel database per la loro validazione o meno ed infine una pagina di conferma o errore per l'operazione effettuata dall'utente.

1.3.5 L'arrivo di HTML5

Come precedentemente riportato le tecnologie dei siti Web 2.0 sono le stesse del Web 1.0 ed anche il linguaggio HTML è rimasto lo stesso, anche se è stato notevolmente sviluppato negli anni. L'ultima versione disponibile è appunto HTML5.
Lo sviluppo venne avviato dal gruppo di lavoro Web Hypertext Application Technology Working Group (WHATWG) fondato nel 2004 da sviluppatori appartenenti ad Apple, Mozilla Foundation e Opera Software. Questo gruppo di lavoro si pose come obiettivo quello di progettare specifiche per lo sviluppo di applicazioni web, focalizzandosi su miglioramenti e aggiunte

ad HTML e alle tecnologie correlate.
Il World Wide Web Consortium ha annunciato che la prima versione dello standard sarà pronta per fine 2014 e HTML 5.1 per il 2016; la prima Candidate Recommendation è stata pubblicata dal W3C il 17 dicembre 2012.

Le novità introdotte da HTML5 rispetto a HTML 4 sono finalizzate soprattutto a migliorare il disaccoppiamento tra struttura, definita dal markup, caratteristiche di resa (tipo di carattere, colori, eccetera), definite dalle direttive di stile, e contenuti di una pagina web, definiti dal testo vero e proprio. Inoltre HTML5 prevede il supporto per la memorizzazione locale di grosse quantità di dati scaricati dal web browser, per consentire l'utilizzo di applicazioni basate su web (come per esempio le caselle di posta di Google o altri servizi analoghi) anche in assenza di collegamento a Internet.

Tra le tante novità introdotte da HTML 5 è opportuno ricordarne qualcuna: l'introduzione di elementi specifici per il controllo di contenuti multimediali (tag video e tag audio), il supporto di Canvas che permette di utilizzare JavaScript per creare animazioni e grafica bitmap, l'introduzione della geolocalizzazione dovuta ad una forte espansione di sistemi operativi mobili (quali Android e iOS, tra i più diffusi), la standardizzazione di programmi JavaScript chiamati Web Workers ed infine la possibilità di utilizzare alcuni siti offline.

Capitolo 2

Javascript

2.1 Descrizione Generale

Javascript è un linguaggio single-thread, ed è utilizzato sia da Apple sia da Microsoft sia da Adobe. Quello che era iniziato come una semplice funzione del browser è diventato uno dei linguaggi più utilizzati al mondo, soprattutto per la costruzione di web-app. Grazie alla capillarità dei browser web, JavaScript si è avvicinato molto di più di qualsiasi altro linguaggio a soddisfare quella vecchia promessa di java:

> write once, run everywhere

Storia

Fu originariamente sviluppato da Brendan Eich della Netscape Communications con il nome di Mocha e successivamente di LiveScript, ma in seguito è stato rinominato JavaScript ed è stato formalizzato con una sintassi più vicina a quella del linguaggio Java di Oracle (ai tempi ancora di Sun Microsystems).
Netscape ha introdotto un implementazione del linguaggio di scripting server-side con Netscape Enterprise Server, la prima volta nel dicembre 1994 (subito dopo il rilascio di JavaScript per i browser). Dalla metà degli anni 2000, vi è stata una proliferazione di implementazioni JavaScript lato server; Node.js ne è un esempio recente ed è utilizzato in applicazioni reali.

JavaScript molto rapidamente ha guadagnato successo, e venne sempre più diffuso come un linguaggio di scripting lato client per le pagine web. Microsoft ha introdotto il supporto JavaScript nel proprio browser web, Internet Explorer, nella versione 3.0, rilasciata nel mese di agosto 1996. Il server web di Microsoft, Internet Information Server, ha introdotto il supporto per lo scripting server-side in JavaScript con release 3.0 (1996). Microsoft ha iniziato a promuovere la creazione di script nelle pagine web utilizzando il termine Dynamic HTML. L'implementazione JavaScript di Microsoft è stata successivamente rinominata JScript per evitare problemi legati al marchio.

Nel novembre 1996, Netscape ha annunciato di aver presentato JavaScript per Ecma International per l'esame come standard del settore, e il successivo lavoro ha portato alla versione standardizzata dal nome ECMAScript. Nel giugno 1997, Ecma International ha pubblicato la prima edizione della specifica ECMA-262. Un anno dopo, nel giugno del 1998, alcune modifiche sono state fatte per adattarlo allo standard ISO/IEC-16262, e la seconda edizione è stata rilasciata. La terza edizione di ECMA-262 (pubblicato nel dicembre 1999) è la versione maggior parte dei browser attualmente in uso. L'attuale edizione dello standard ECMAScript è 5.1, rilasciato nel giugno 2011.

JavaScript è diventato uno dei linguaggi di programmazione più popolari sul web. Inizialmente, tuttavia, molti programmatori professionisti lo denigrarono perché il suo target di riferimento era costituito da autori web e altri dilettanti. Con l'avvento di Ajax JavaScript è tornato alla ribalta e ha portato più attenzione alla programmazione professionale. Il risultato è stato un proliferare di framework e librerie complete, migliori pratiche di programmazione e maggiore utilizzo di JavaScript al di fuori del browser web, come si vede dalla proliferazione di piattaforme JavaScript lato server. Nel gennaio 2009, il progetto CommonJS stato fondato con l'obiettivo di specificare una libreria standard comune, principalmente per lo sviluppo JavaScript di fuori del browser.

JavaScript è stato standardizzato per la prima volta tra il 1997 e il 1999

dalla ECMA con il nome ECMAScript. Oggi, JavaScript è un marchio registrato di Oracle Corporation. È utilizzato su licenza per tecnologia inventata e implementata da Netscape Communications e le entità attuali, come la Mozilla Foundation.

Aspetti Strutturali

Imperativo e Strutturato

JavaScript supporta gran parte della sintassi di programmazione strutturata da C (ad esempio, if, while, switch, ecc.). Come C, JavaScript fa una distinzione tra le espressioni e le dichiarazioni. Una differenza sintattica da C è l'inserimento automatico dei punti e virgola, in cui i punti e virgola dove terminano le dichiarazioni possono essere omessi.

Object Base

JavaScript è quasi interamente basato su oggetti. Oggetti JavaScript sono gli array associativi, potenziati con i prototipi. Supportano due sintassi equivalenti: dot notation (obj.x = 10) e la notazione classica (obj ['x'] = 10). Le proprietà dei relativi valori possono essere aggiunte, modificate o eliminate in fase di esecuzione. La maggior parte delle proprietà di un oggetto (e quelli sulla sua catena di ereditarietà di prototipo) possono essere enumerate utilizzando cicli for-in.

Tipizzazione Dinamica

Come nella maggior parte dei linguaggi di scripting, i tipi sono associati a valori, non con le variabili. Ad esempio, una variabile 'x' potrebbe essere un numero, poi convertita a una stringa. JavaScript supporta diversi modi per verificare il tipo di un oggetto.

Run-time eval

JavaScript include una funzione eval che può eseguire istruzioni forniti come stringhe a run-time.

Funzioni
Le funzioni sono anch'esse oggetti. Come tali, essi hanno proprietà e metodi, come ad esempio. Call () e. Bind (). Una funzione annidata è una funzione definita all'interno di un'altra funzione. Essa viene creata ogni volta che la funzione esterna viene chiamata. Inoltre, ogni funzione creata forma una chiusura lessicale: l'ambito lessicale della funzione esterna, compresi eventuali costanti, variabili locali e valori degli argomenti, entra a far parte dello stato interno di ogni oggetto della funzione interna, anche dopo la fine dell'esecuzione della funzione esterna.

Basato sui Prototipi
JavaScript utilizza prototipi dove molti altri linguaggi orientati agli oggetti utilizzano le classi per l'ereditarietà. È possibile simulare molte caratteristiche di classe a base di prototipi in JavaScript.

Altri aspetti di interesse: in JavaScript lato client, il codice viene eseguito direttamente sul client e non sul server. Il vantaggio di questo approccio è che, anche con la presenza di script particolarmente complessi, il web server non viene sovraccaricato a causa delle richieste dei clients. Di contro, nel caso di script che presentino un codice sorgente particolarmente grande, il tempo per lo scaricamento può diventare abbastanza lungo. Un altro svantaggio è il seguente: ogni informazione che presuppone un accesso a dati memorizzati in un database remoto deve essere rimandata ad un linguaggio che effettui esplicitamente la transazione, per poi restituire i risultati ad una o più variabili JavaScript; operazioni del genere richiedono il caricamento della pagina stessa. Con l'avvento di AJAX tutti questi limiti sono stati superati.

2.2 Uso in HTML

L'uso principale di Javascript è di scrivere funzioni che vengono poi incorporate e incluse nelle pagine HTML e che successivamente interagiscono con il Document Object Model (DOM: una forma di rappresentazione dei documenti strutturati come modello orientato agli oggetti) della pagina.
Qualche semplice esempio di utilizzo di JavaScript:

- Caricamento di un nuovo contenuto o l'invio di dati al server tramite la tecnologia AJAX senza ricaricare la pagina (ad esempio, nei social network potrebbe permettere all'utente di inviare aggiornamenti di stato senza lasciare la pagina)
- Animazione di elementi della pagina, dissolvenza dentro e fuori, ridimensionare, spostare ecc. (jQuery)
- Contenuti interattivi, ad esempio i giochi, e la riproduzione di audio e video
- Convalida valori di ingresso di web form per assicurarsi che essi siano accettabili prima di essere inviati al server.
- La trasmissione di informazioni sulle abitudini di lettura degli utenti e le attività di navigazione a vari siti web. Le pagine Web spesso fanno analisi, monitorano gli annunci ecc.

Poiché il codice JavaScript può essere eseguito localmente nel browser di un utente (piuttosto che su un server remoto), il browser è in grado di rispondere alle azioni degli utenti in modo rapido, e in modo più reattivo. Inoltre, il codice JavaScript è in grado di rilevare le azioni dell'utente, mentre con l'utilizzo del solo HTML non è possibile, ad esempio, singoli tasti. Le applicazioni come Gmail approfittano di questo: gran parte della logica di interfaccia utente è scritta in JavaScript, e JavaScript invia richieste (come ad esempio il contenuto di un messaggio e-mail) al server. La tendenza più ampia di programmazione Ajax sfrutta allo stesso modo questa forza.

Un motore JavaScript (noto anche come interprete JavaScript) interpreta il codice sorgente JavaScript e esegue lo script di conseguenza. Il primo motore JavaScript è stato creato da Brendan Eich a Netscape Communications Corporation, per il browser Netscape Navigator. Il motore, nome in

codice SpiderMonkey, è implementato in C. Da allora è stato aggiornato (in JavaScript 1.5) per conformarsi alle ECMA-262 Edition 3.

Un browser web è di gran lunga l'ambiente host più comune per JavaScript. I browser Web di solito creano host objects per rappresentare il Document Object Model (DOM) in JavaScript. Il web server è un altro ambiente di gestione. Un webserver JavaScript tipicamente risponde alle richieste HTTP, un programma JavaScript (client) può quindi interrogare il webserver generare dinamicamente le pagine web.

Poiché JavaScript viene eseguito in ambienti molto diversi, una parte importante di test e debug è quello verificare che il JavaScript funzioni su più browser, soprattutto i più utilizzati.
Per far fronte alle differenze tra browser gli autori JavaScript possono tentare di scrivere codice conforme agli standard che verrà eseguito correttamente nella maggior parte dei browser, altrimenti, se questo non è possibile, si può scrivere codice che controlla la presenza di alcune caratteristiche del browser e si comporta in modo diverso se alcune funzionalità non sono disponibili.
In alcuni casi, due browser possono entrambi implementare una funzionalità, ma con un comportamento diverso, e per i programmatori potrebbe essere utile rilevare quale browser sta eseguendo il codice per poter poi cambiare lo script da abbinare.

Inoltre, gli script possono non funzionare comunque, ad esempio un utente può:

- Utilizzare un browser vecchio o particolarmente raro che non supporta, o supporta limitatamente gli elementi DOM
- Utilizzare un PDA (palmare) o un browser per smartphone che non supportano JavaScript
- Aver bloccato l'esecuzione di codice JavaScript per una questione di sicurezza

Per supportare questi utenti, i programmatori possono cercare di creare pagine con un alta tolleranza ai guasti, soprattutto per i browser che non

supportano JavaScript. In particolare la pagina deve cercare di rimanere agibile e usabile, ovviamente senza le funzionalità aggiunte dal codice JavaScript.

2.3 Librerie JavaScript

JavaScript vanta un ampio parco di framework utili per aggiungere dinamicità, effetti, interazioni asincrone ed altro ancora, spesso in pochissime linee di codice. Verranno elencate successivamente le librerie che hanno avuto più successo negli ultimi anni

2.3.1 jQuery

jQuery è una libreria multi-browser di JavaScript progettata per semplificare la creazione di script HTML lato client. è stata rilasciata nel 2006 nel BarCamp NYC da John Resing. Dave Methvin e un team di sviluppatori stanno contribuendo ancora oggi al suo sviluppo. Oltre il 65% dei 10.000 siti più visitati utilizza questa libreria, questo dato rende di fatto jQuery la libreria JavaScript più utilizzata al mondo.

jQuery è gratis, open source e sotto la licenza del MIT. La sua sintassi è progettata per rendere più facile la navigazione di un documento, selezionare elementi DOM, creare animazioni, gestire gli eventi e sviluppare applicazioni Ajax. jQuery fornisce anche funzionalità per gli sviluppatori per creare plug-in. Questo permette agli sviluppatori di creare astrazioni per l'interazione a basso livello e di animazione, effetti avanzati e di alto livello. L'approccio modulare alla libreria jQuery permette la creazione di potenti pagine web dinamiche e applicazioni web.

Negli ultimi anni sono state inoltre sviluppate da jQuery altre librerie specifiche per determinati scopi come **jQuery UI** dove il termine UI sta per User Interface e serve a rendere il più personalizzabile possibile la creazione di widget, le interazioni, le animazioni ecc ecc.

Un'altra libreria derivata da jQuery è **jQuery Mobile**: un framework appositamente creato per lo sviluppo di pagine web compatibili con dispositivi mobile come tablet o smartphone di ultima generazione.

2.3.2 Node.js

Node.js è una piattaforma software che viene utilizzato per costruire applicazioni web scalabili, soprattutto lato server. Raggiunge un elevato throughput attraverso funzioni di I/O non bloccanti e attraverso un ciclo di eventi single-threaded.
Node.js contiene una libreria incorporata per un server HTTP, che permette di eseguire un server web senza l'utilizzo di software esterni, come ad esempio Apache o Lighttpd, e consentendo un maggiore controllo di come funziona il web server.

Node.js è stato creato da Ryan Dahl a partire dal 2009, e il suo sviluppo e la manutenzione è sponsorizzata da Joyent, il suo ex datore di lavoro.
L'obiettivo originale di Dahl è stato quello di creare siti web con capacità di spinta, come visto in applicazioni web come Gmail. Dopo aver cercato soluzioni in diversi altri linguaggi di programmazione scelse JavaScript a causa della mancanza di una API di I/O. Questo gli ha permesso di definire una convenzione di I/O non bloccante e orientata agli eventi.

2.3.3 Dojo Toolkit

Dojo Toolkit è una libreria modulare open source di JavaScript progettata per facilitare il rapido sviluppo di applicazioni cross-platform basate su Ajax e siti Web. è stato avviato da Alex Russell, Dylan Schiemann, David Schontzler, e altri nel 2004 ed è a doppia licenza, sotto la licenza BSD oppure l'Academic Free License. La Fondazione Dojo è un non-profit creato con l'obiettivo di promuovere l'adozione del toolkit.

Come framework JavaScript Dojo mira a soddisfare le numerose esigenze di sviluppo web client-side su larga scala. Ad esempio, Dojo astrae le differenze tra i diversi browser per fornire API che funzionano su tutti loro,

fissa un quadro per la definizione di moduli di codice e la gestione delle loro interdipendenze, ma fornisce costruzione di strumenti per l'ottimizzazione di JavaScript e CSS, generazione di documentazione e test unit, supporta l'internazionalizzazione, la localizzazione e l'accessibilità, e fornisce una ricca suite di classi di utility e widget dell'interfaccia utente.

2.3.4 Altre Librerie

Prototype

Prototype JavaScript Framework è un framework JavaScript creato da Sam Stephenson nel febbraio 2005, inizialmente ideato come supporto Ajax in Ruby on Rails. Esso ha come scopo quello di facilitare lo sviluppo di applicazioni web dinamiche. In particolare offre supporto per l'utilizzo di AJAX e della Programmazione orientata agli oggetti in JavaScript. Inoltre è utilizzato come libreria di supporto per altri progetti JavaScript come script.aculo.us e Rico.

Ext JS

Ext JS è una libreria JavaScript per la costruzione di applicazioni web interattive, utilizzando tecnologie come Ajax, DHTML, e lo scripting DOM. Originariamente costruito come un add-on alla libreria YUI (Yahoo! UI Library) da Jack Slocum, Ext include l'interoperabilità con jQuery e Prototype. A partire dalla versione 1.1, Ext non ha alcuna dipendenza da librerie esterne, rendendo il loro uso invece facoltativo.

Script.aculo.us

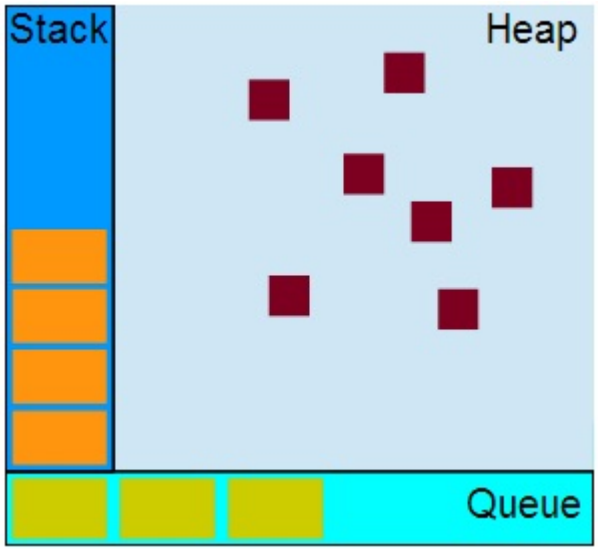
Script.aculo.us è una libreria JavaScript che permette di creare effetti grafici di grande effetto nei siti web utilizzando un sistema di funzioni molto semplici. La libreria si basa su Prototype ed in particolare contiene effetti grafici, una libreria per il cosiddetto Drag and drop e molto altro ancora. Sostanzialmente, grazie a poche righe di codice è possibile sviluppare effetti molto suggestivi come la scomparsa e l'apparizione degli oggetti, il loro trascinamento o l'autocompletamento dei form (come succede nella pagina di google).

Capitolo 3

Programmazione Asincrona in JavaScript

3.1 L'Event-Loop di JavaScript

JavaScript ha un modello di concorrenza basato su un ciclo di eventi o "event loop". Questo modello è molto diverso rispetto al modello in altri linguaggi come C o Java.



Stack

Le chiamate alle funzioni formano uno "stack" o una pila di frame:

```
function f(b) {
    var a = 12;
    return a + b + 35;
}

function g(x) {
    var m = 4;
    return f(m * x);
}
```

Quando si chiama *g*, viene creato un primo frame contenente gli argomenti e le variabili locali di *g*. Quando poi *g* chiama *f*, un secondo frame, contenente gli argomenti e le variabili locali di *f*, viene creato e messo sopra "push" al primo. Quando *f* restituisce il risultato, il frame viene eliminato dallo stack, lasciando solo il frame *g*. Quando anche *g* finische, lo stack è vuoto.

Queue

A runtime JavaScript contiene una coda (Queue) di funzioni, in pratica una lista di funzioni da elaborare. Quando lo stack è vuoto, una funzione viene estratta dalla coda ed elaborata. La funzione chiamata poi inserirà un frame nello stack. L'elaborazione termina quando lo stack torna vuoto.

Event-Loop

L'implementazione in pseudo codice dell'event-loop è questa:

```
while (queue.waitForFunction()) {
    queue.processNextFunction();
}
```

queue.waitForFunction attende sincronicamente l'arrivo di una funzione.

Ogni funzione viene elaborata completamente prima di passare a quella successiva. Questo offre diverse proprietà di funzionamento riguardo al programma che si vuole realizzare, compreso il fatto che ogni volta che viene eseguita una funzione, non può essere anticipata e si svolgerà interamente prima di qualsiasi altro punto del codice.
Un aspetto negativo di questo modello è che se una funzione richiede troppo tempo per essere completata, l'applicazione web non è in grado di elaborare le interazioni degli utenti, come click o scroll. Il browser attenua questo con la finestra di dialogo "uno script sta impiegando troppo tempo". Una buona pratica da seguire è quella di rendere l'elaborazione di una funzione breve. Se possibile, si può anche optare per la suddivisione di una funzione.

Nel browser web, le funzioni vengono aggiunte ogni volta che un evento si verifica e vi è un event listener collegato ad esso. Se non c'è l'event listener, l'evento viene perso.

Una proprietà molto interessante del modello event-loop è che JavaScript, a differenza di molti altri linguaggi, è **non bloccante**. Le funzioni di I/O vengono in genere eseguite tramite eventi e callback, così quando l'applicazione è in attesa, ad esempio, di una query, può ancora elaborare altre cose, come l'input dell'utente.

3.2 Programmazione ad Eventi

Javascript è stato concepito per essere un linguaggio single-thread dove vengono manipolate le attività asincrone attraverso gli eventi. quando ci sono potenzialmente pochi eventi, costruire codice event-base è più semplice che costruire codice multi-thread.

È concettualmente più elegante ed elimina la necessità di avvolgere il backup dei dati in mutex con semafori per rendersi thread-safe. Ma quando un evento è strettamente legato all'evento precedente la semplicità spesso da il posto ad una struttura così terrificante che è stata soprannominata *Pyramid of Doom*, una serie di funzioni annidate una dentro l'altra che rendono

il codice decisamente poco leggibile, e soprattutto, lo rendono difficilmente ampliabile e modificabile:

```
step1(function(result1) {
  step2(function(result2) {
    step3(function(result3) {
      // and so on...
    });
  });
});
```

Non si può ovviamente programmare in questo modo, ma il problema non è con il linguaggio stesso, ma con il modo in cui i programmatori utilizzano il linguaggio.

Ma come funzionano gli eventi? la confusione sui modelli di eventi asincroni in Javascript è molto datata, praticamente dalla nascita di JavaScript stesso.
Però gli eventi Javascript sono sia concettualemente eleganti sia pratici. Una volta accettato che il linguaggio è single-thread, il single-thread appunto è più una caratteristica che una limitazione. Ciò significa che il codice è non-bloccante e gli eventi sono pianificati in fila in modo ordinato

3.2.1 Schedulazione Eventi

Quando si vuole eseguire un pezzo di codice nel futuro in JavaScript, noi lo mettiamo in una callback. Una callback è una funzione ordinaria, tranne quando è passata ad una funzione come setTimeout o legata come proprietà a document.onready. Quando una callback viene eseguita vuol dire che l'evento a cui era stata associata ha avuto luogo e vuol dire inoltre che tutto il codice "sincrono" è finito, e quindi, dalla coda degli eventi è stata eseguita tale callback.

Andiamo ora a vedere come si comporta la funzione *setTimeout*, una prima definizione intuitiva di setTimeout potrebbe essere questa:"data una

callback e un ritardo di n millisecondi, setTimeout esegue la callback n millisecondi dopo".
Ma come descrizione è in realtà incompleta.

Vediamo ora un semplice esempio di utilizzo di setTimeout:

```
for (var i = 1; i <= 3; i++) {
  setTimeout(function(){ console.log(i); }, 0);
};

4
4
4
```

molti principianti del linguaggio si aspettano che questo loop produca in uscita 1,2,3 o comunque una giustapposizione di questi 3 numeri come i 3 millisicecondi. Per capire perché l'uscita è 4,4,4 bisogna capire un concetto fondamentale: *i gestori di eventi JavaScript non vengono eseguiti fino a che il Thread è libero.*

Vediamo ora un ulteriore esempio di utilizzo di setTimeout:

```
var start = new Date;
setTimeout(function(){
  var end = new Date;
  console.log('Time elapsed:', end - start, 'ms');
}, 500);
while (new Date - start < 1000) {};
```

Una mentalità ancora multithread si aspetta che dopo 500ms la funzione nel setTimeout venga eseguita. Ma questo vuol dire che il ciclo (di durata 1000ms) doveva essere interrotto. Invece se si esegue il codice si otterrà:
time elapsed: 1002ms
Quindi la funzione callback di setTimeout non potrà mai essere eseguita finché il while non termina.
Quando chiamiamo setTimeout un evento timeout è accodato. Poi l'esecuzione continua fino a che non ci sono più linee di codice. Solo allora la

virtual machine di JavaScript guarda la coda degli eventi.
Se c'è almeno un evento nella coda eseguibile la VM chiama il suo gestore (ad esempio, la funzione che abbiamo passato a setTimeout). Quando il gestore la restituisce torniamo alla coda.

Gli eventi di input funzionano allo stesso modo: quando un utente fa click su un elemendo DOM con un gestore di click allegato, un evento click viene messo in coda. Ma il gestore non verrà eseguito fino a quando tutto il codice in esecuzione è terminato.
La facilità di programmazione degli eventi in JavaScript è una delle caratteristiche più potenti del linguaggio.
Il codice JavaScript non può essere interrotto, in quanto gli eventi possono essere messi in coda solo durante l'esecuzione del codice; e non verranno mai eseguiti fino a che il codice non è terminato.

3.2.2 Funzioni Asincrone

Ogni ambiente JavaScript ha le proprie funzioni asincrone. Alcune come setTimeout e setInterval sono ovunque. Altre sono solamente nel Browser e altre solo lato-server. Generalmente però si suddividono in 2 categorie: **I/O** e **Timing**. Questi sono gli elementi di base che si usano per definire complessi comportamenti asincroni nelle applicazioni.

Funzioni di I/O

Node.js è stato creato per avere un event-driven server framework costruito con un linguaggio di alto livello. JavaScript è il linguaggio perfettto perché è adatto per I/O non bloccanti.
In JavaScript un loop come questo viene eseguito per sempre:

```
var ajaxRequest = new XMLHttpRequest;
ajaxRequest.open('GET', url);
ajaxRequest.send(null);
while (ajaxRequest.readyState === XMLHttpRequest.UNSENT) {
  // readyState can't change until the loop returns
};
```

La cosa giusta da fare invece è collegare un gestore e ritornare alla coda di eventi:

```
var ajaxRequest = new XMLHttpRequest;
ajaxRequest.open('GET', url);
ajaxRequest.send(null);
ajaxRequest.onreadystatechange = function() {
  // ...
};
```

Ogni volta che bisogna aspettare la pressione di un tasto dall'utente bisogna definire una callback. Altrimenti il nostro ambiente JavaScript farà una I/O sincrona che diventerà bloccante.

Funzioni Timing

Funzioni create per essere eseguite in un punto in futuro, per un'animazione o una simulazione. Le ben note funzioni temporali sono setTimeout e setInterval.
Purtroppo queste funzioni hanno diversi difetti. Come visto in precedenza uno di questi difetti è insormortabile: nessuna funzione di temportizzazione di JavaScript può causare l'esecuzione di codice, mentre un altro codice è in esecuzione nello stesso processo.

```
var fireCount = 0;
var start = new Date;
var timer = setInterval(function() {
  if (new Date - start > 1000) {
    clearInterval(timer);
    console.log(fireCount);
    return;
  }
  fireCount++;
}, 0);
```

Quando si pianifica un evento setInterval e un ritardo di 0ms, l'evento deve essere eseguito il più spesso possibile.

Ma in realtà setTimeout e setInterval sono lenti in base alla progettazione. Infatti le specifiche HTML hanno un minimo di Timeout/interval di 4ms.

Nel caso si desiderasse una grana di temporizzazione più fine ci sono due possibilità:

- In Node, process.nextTick consente i programmare un evento in esecuzione il più presto possibile
- I Browser moderni hanno una funzione requestAnimationFrame che serve ad un duplice scopo: permette animazioni JavaScript a 60+ frame/sec e conserva cicli di CPU, impedendo l'esecuzione delle animazioni delle schede in background.

3.2.3 Funzioni di Callbacks

Ogni funzione asincrona in JavaScript può far partire altre funzioni, sia sincrone sia asincrone e, ovviamente, qualsiasi funzione asincrona deve fornire il risultato di tale operazione in modo asincrono.
il termine asincrono è un termine un po' improprio, se si chiama una funzione, il programma semplicemente non continuerà fino a che la funzione non "ritorna". In JavaScript quando invece parliamo di funzione asincrona intendiamo che può causare un altra funzione (callback) da eseguire in seguito nella coda degli eventi.Un altro termine per funzione asincrona potrebbe essere funzione non bloccante.

In genere, funzioni che richiedono la callback la pretendono come ultimo argomento (setTimeout e setInterval sono eccezioni) ma alcune funzioni asincrone usano la callback indirettamente, restituendo una *promise*, cioè un oggetto che rappresenta un'attività con due possibili esiti (risolta o rifiutata) , o utilizzando *PubSub*, un modo appunto per pubblicare e propagare gli eventi in tutta l'applicazione.

Ci sono funzioni che sono asincrone in certi casi e sincrone in altri. Per esempio la funzione omonima *jQuery* può essere usata per ritardare una funzione fino a quando il DOM non è carico. Ma se il DOM è già carico

non c'è alcun ritardo e la funzione viene eseguita subito

Prendiamo come esempio la costruzione di un calcolatore browser-based che utilizza web worker per fare i calcoli in un altro thread:

```
var calculationCache = {},
    calculationCallbacks = {},
    mathWorker = new Worker('calculator.js');

mathWorker.addEventListener('message', function(e) {
  var message = e.data;
  calculationCache[message.formula] = message.result;
  calculationCallbacks[message.formula](message.result);
});

function runCalculation(formula, callback) {
  if (formula in calculationCache) {
    return callback(calculationCache[formula]);
  };
  if (formula in calculationCallbacks) {
    return setTimeout(function() {
      runCalculation(formula, callback);
    }, 0);
  };
  mathWorker.postMessage(formula);
  calculationCallbacks[formula] = callback;
}
```

In questo caso la funzione runCalculation è sincrona quando il risultato è memorizzato nella cache, mentre è asincrona negli altri casi.
Gli scenari possibili sono 3:

- La formula è già stata computata e il risultato è nella cache. In questo caso la funzione è sincrona
- La formula è stata spedita al worker ma il risultatno non è stato ancora ricevuto. In questo caso runCalculation imposta un timeout

per richiamarsi di nuovo. Il processo si rifà fino a quando il risultatno non è nella cache.

- La formula non è stata ancora spedita al worker. In questo caso noi invochiamo la callback dall'event listener del worker.

Negli scenari 2 e 3 noi aspettiamo che un task si completi in 2 modi differenti.

In runCalculation abbiamo aspettato che il worker finisse il suo calcolo ripetendo la stessa funzione con un timeout oppure memorizzando la callback. A prima vista potrebbe sembrare più facile utilizzare la ricorsione asincrona, eliminando l'oggetto *calculationCallbacks*. Spesso viene usato setTimeout per questo scopo perché assomiglia di più ad un idioma di un linguaggio single-thread.

Ma i Timeout non sono a "costo zero". Tanti timeout possono creare un significante carico computazionale, e nella ricorsione asincrona non c'è limite al numero di timeout che potrebbero essere in attesa. Per queste ragioni la ricorsione asincrona viene considerata un anti-pattern.

Callbacks Annidate

L'anti-pattern più comune in JavaScript è l'annidamento delle callback, prendiamo un semplice esempio in Node.js, una funzione che controlla che l'utente abbia inserito la password giusta:

```
function checkPassword(username, passwordGuess, callback) {
  var queryStr = 'SELECT * FROM user WHERE username = ?';
  db.query(selectUser, username, function (err, result) {
    if (err) throw err;
    hash(passwordGuess, function(passwordGuessHash) {
      callback(passwordGuessHash === result['password_hash']);
    });
  });
}
```

Qui la funzione asincrona *checkPassword* lancia un'altra funzione asincrona *db.query* la quale lancia un'altra funzione potenzialmente asincrona *hash*.
In questo momento il codice non è così complicato, ma se si vogliono aggiungere nuove funzioni ad esso (come la gestione di errore del database, la registrazione tentativi di accesso, limitazione e così via) diventa difficile.
Le callback annidate fanno in modo che per aggiungere nuove funzionalità dobbiamo scrivere più codice, piuttosto che attuare queste caratteristiche in pezzi riusabili e gestibili.

Questa implementazione equivalente evita questi problemi:

```
function checkPassword(username, passwordGuess, callback) {
  var passwordHash;
  var queryStr = 'SELECT * FROM user WHERE username = ?';
  db.query(selectUser, username, queryCallback);

  function queryCallback(err, result) {
    if (err) throw err;
    passwordHash = result['password_hash'];
    hash(passwordGuess, hashCallback);
  }

  function hashCallback(passwordGuessHash) {
    callback(passwordHash === passwordGuessHash);
  }
}
```

Questo approccio è più prolisso, ma si legge chiaramente ed è molto più facile da estendere.
Come regola generale, evitare più di due livelli di funzioni nidificate. La chiave è trovare un modo per memorizzare i risultati fuori dalla funzione che fa la chiamata asincrona, in questo modo le callback non devono essere nidificate.

3.3 Eventi Distribuiti

per una semplice gestione degli eventi è sufficiente collegare un gestore ad ogni evento e la nostra applicazione lo gestisce. Questo è vero, ma quando

un evento ha ripercussioni diverse, il principio "one event, one handler" obbliga i gestori a crescere fino a proporzioni gigantesche.
Se per esempio si sta costruendo un web-based word processor. Ogni volta che un utente preme un tasto, una serie di cose devono accadere: il nuovo carattere deve essere visualizzato sullo schermo, il cursore deve essere spostato, l'azione deve essere inserita nel locale "annullare storia" e sincronizzata con il server, potrebbe essere necessario eseguire il controllo ortografico, e il numero di parole o di pagine potrebbe essere aggiornato. La gestione e la realizzazione di tutti questi compiti ad opera di un singolo gestore è inutilmente complicata e pesante.
E oppurtuno sostituire il massiccio gestore di eventi con uno più malleabile e dinamico. Un gestore dove possiamo aggiungere attività e rimuovere attività in fase di esecuzione (runtime). In breve, vogliamo utilizzare gli eventi distribuiti, dove un singolo episodio può scatenare reazioni in tutta la nostra applicazione.

Con le Specifiche del 2000 il W3C ha aggiunto il metodo *addEventListener* nelle specifiche del DOM, e jQuery l'ha astratto con il metodo *bind*. Bind rende facile aggiungere molti handler a qualsiasi evento o a qualsiasi elemento, senza preoccuparsi di calpestare i piedi a qualcuno. Dal punto di vista dell'architettura software, jQuery pubblica per gli eventi l'elemento link per chiunque voglia iscriversi. Questo è chiamato *PubSub*.
In Node esiste una generica entità *PubSub* chiamata *EventMitter*. Per aggiungere un gestore di eventi al EventEmitter basta chiamare *on* con il tipo di evento e il gestore; Il metodo *Emit* andrà a chiamare tutti i gestori per lo stesso tipo di evento.

Anche se PubSub è una tecnica per la gestione di eventi asincroni, non c'è nulla di intrinsecamente asincrono su di essa. Si consideri questo codice:

```
$('input[type=submit]')
.on('click', function() { console.log('foo'); })
.trigger('click');
console.log('bar');
```

l'output sarà:

```
foo
bar
```

Dimostrando che il gestore click è stato chiamato subito da trigger. In effetti, ogni volta che un evento jQuery viene eseguito, tutti i suoi gestori saranno eseguiti senza interruzioni.
Quindi, cerchiamo di essere chiari: l'utente fa click sul pulsante invia, questo è un evento asincrono. Il primo handler del click viene eseguito dalla coda di eventi. Ma l'handler non ha nessun modo di sapere se viene eseguito dalla coda di eventi o dal codice dell'applicazione.

Modelli Evented

Quando un oggetto ha un interfaccia PubSub noi lo chiamiamo *evented object*. Un caso particolare è quando un oggetto utilizzato per memorizzare i dati (un modello) pubblica gli eventi ogni volta che viene modificato. I modelli sono la M in Model-View-Controller (MVC) che è diventato uno dei pattern più utilizzati nella programmazione JavaScript degli ultimi anni.
La vecchia scuola di JavaScript apporta modifiche al DOM direttamente dal gestore di eventi input, la nuova scuola apporta invece modifiche ai modelli, che poi emettono eventi che fanno aggiornare il DOM. In quasi tutte le applicazioni, questa separazione di risultati, risulta più elegante e con codice più intuitivo

Nella sua forma più semplice, MVC consiste nel cablare modelli a viste: "se questo modello cambia in questo modo, il DOM cambia in quest'altro modo" ma i maggiori profitti di MVC ci sono quando gli eventi di cambiamento salgono l'albero dei dati. Invece di sottoscrivere gli eventi in ogni foglia, si possono sottoscrivere le radici e i rami.

Capitolo 4

Le Promise

4.1 Promises e Deferred

Da jQuery 1.5 sono state introdotte per la prima volta le *promise*. Una Promise è un oggetto che rappresenta un'attività con due possibili esiti (successo o fallimento) e ha le callbacks che vengono lanciate quando si verifica uno dei due risultati:

```
var promise = $.get('/mydata');
promise.done(onSuccess);
promise.fail(onFailure);
promise.always(onAlways);
```

Se una chiamata AJAX ha molteplici effetti (animazioni, inserimento di HTML, blocco/sblocco input per l'utente ecc ecc), è difficile gestirle tutte da parte dell'applicazione.
È molto più elegante passare una promise. Passando una promise si sta dicendo che qualcosa sta accadendo, se poi si dà a questa promise una callback si saprà anche quando accade, È come un EventMitter.
Ma il più grande vantaggio di utilizzare le promise è che si possono ricavare facilmente nuove promise da quelle esistenti. Si potrebbero costruire due promise che rappresentano attività parallele per creare poi una promise che ci informa del loro completamento. Oppure si potrebbe fare una promise che rappresenta la prima attività di una serie e di conseguenza una che rappresenta il completamento finale di tutta la serie.

4.1.1 Creare Promise

Cerchiamo ora di creare una promise, prendiamo come esempio la costruzione di un Prompt per premere o Y (si) o N (no). Invece di creare direttamente una promise, creiamo una *deferred* che è un superset di promise con un'aggiunta fondamentale, è possibile attivare una deferred direttamente, infatti una promise pura permette solo di aggiungere callback che dovranno essere innescate da qualcun'altro. Le deferred invece possono essere innescate con i metodi *resolve* e *reject*.

```
var promptDeferred = new $.Deferred();
promptDeferred.always(function(){ console.log('A choice was made:'); });
promptDeferred.done(function(){ console.log('Starting game...'); });
promptDeferred.fail(function(){ console.log('No game today.'); });

$('#playGame').focus().on('keypress', function(e) {
  var Y = 121, N = 110;
  if (e.keyCode === Y) {
    promptDeferred.resolve();
  } else if (e.keyCode === N) {
    promptDeferred.reject();
  } else {
    return false;  // our Deferred remains pending
  };
});
```

Con questo codice se premo Y la deferred sarà resolved e verranno esequite le funzioni di callback always e done. Mentre se premo N la deferred sarà rejected e verranno eseguite le funzioni di callback always e fail.
Provando a premere ripetutamente Y o N non avremo cambiamenti in quanto una promise può essere o resolved o rejected solo una volta, da li in poi la promise è inerte. Si dice che la promise è in pending (attesa) quando non è ancora stata o rejected o resolved. Possiamo controllare lo stato della promise con i metodi di stato (*isResolved isRejected*)
Quando si sta effettuando una operazione asincrona one-shot con due grandi risultati (ad esempio, il successo/fallimento o accettare/declinare), una deferred dà una rappresentazione intuitiva del risultato.

Se si volesse avere una promise che non è una deferred basterebbe semplicemente chiamare il metodo promise della deferred:

```
var promptPromise = promptDeferred.promise();
var promise1 = promptDeferred.promise();
var promise2 = promptDeferred.promise();
console.log(promise1 === promise2);  // true
console.log(promise1 === promise1.promise());  // true
```

promptPromise è solo una copia di promptDeferred senza i metodi di resolved/rejected. Non importa se leghiamo un callback a una deferred o alla sua promise, perché condividono internamente le stesse funzioni di callback. Essi condividono anche lo stesso stato (in attesa, risolto, o rifiutata). Questo significa che la creazione di più promises per una singola deferred sarebbe inutile.
E chiamando promise su una promise jQuery darà un riferimento allo stesso oggetto.
L'unica ragione per usare il metodo promise è l'incapsulamento. Se passiamo promptPromise ma teniamo promptDeferred per noi, possiamo stare certi che nessuna delle nostre callbacks verrà eseguita fino a quando non lo vogliamo noi.
Per chiarire, ogni defererd contiene una promise e ogni promise rappresenta una Deferred. Quando si ha una Deferred, puoi manipolare il suo stato. Quando si dispone di una promise pura, si può leggere solo quello stato e allegare callback.

Ajax è un caso perfetto per utilizzare le promise: ogni chiamata a un server remoto sarà o un successo o un fallimento, e questi casi dovranno essere trattai in modo diverso a livello applicativo. Ma le promise possono essere altrettanto utili per le operazioni asincrone locali, come le animazioni.
In jQuery è possibile passare una callback per qualsiasi metodo di animazione ed essere notificati quando questo finisce, ma è anche possibile chiedere a un oggetto jQuery una promise, che rappresenta il completamento delle sue animazioni in corso e in sospeso.

```
var errorPromise = $('.error').fadeIn().promise();
errorPromise.done(afterErrorShown);
```

Animazioni applicate allo stesso oggetto jQuery sono in coda per l'esecuzione in sequenza, e la promise si risolve solo quando tutte le animazioni che erano in coda al momento della chiamata a promise sono state resolved. Quindi, questo genera due promise distinte che verranno risolte in sequenza.

Passare Argomenti alle Callback

Una Promise può dare alla rispettiva callback delle informazioni aggiuntive. quando una deferred viene resolved o rejected ogni argomento previsto viene mandato alla corrispondente callback:

```
var aDreamDeferred = new $.Deferred();
aDreamDeferred.done(function(subject) {
  console.log('I had the most wonderful dream about', subject);
});
aDreamDeferred.resolve('the JS event model');
```

```
I had the most wonderful dream about the JS event model
```

Ci sono inoltre dei metodi speciali per far partire le callbacks in contesti particolari: resolveWith e rejectWith. Basta passare il contesto come primo argomento e tutti gli altri argomenti in forma di array

```
var slashdotter = {
  comment: function(editor){
    console.log('Obviously', editor, 'is the best text editor.');
  }
};
var grammarDeferred = new $.Deferred();
grammarDeferred.done(function(verb, object) {
  this[verb](object);
});
grammarDeferred.resolveWith(slashdotter, ['comment', 'Emacs']);
```

```
Obviously Emacs is the best text editor.
```

Progress

Inoltre in jQuery è disponibile anche un tipo di promise con callbacks che possono essere richiamate un numero illimitato di volte, si chiama *progress*. Ovviamente una chiamata a progress farà partire la relativa callback fino a quando la deferred non verrà risolta o rejected.

4.1.2 Combinare Promise

L'esistenza delle Progress non cambia il fatto che, in ultima analisi, ogni Promise è o resolve o reject, oppure rimane in attesa per l'eternità.
Il caso più comune per combinare le promise è scoprire quando un insieme di attività asincrone è terminata. Per esempio si sta mostrando un video tutorial durante il caricamento di un gioco dal server. Si vuole iniziare il gioco non appena due cose sono successe, in qualsiasi ordine:

- Il video tutorial è finito
- Il gioco si è caricato

Data una promise che rappresenta ciascuno dei processi lo scopo finale è quello di iniziare il gioco quando entrambe le promise vengono risolte
Introduciamo quindi il metodo *when*:

```
var gameReadying = $.when(tutorialPromise, gameLoadedPromise);
gameReadying.done(startGame);
```

When agisce come un AND logico per la risoluzione delle promise. La Promise che genera viene risolta quando sono entrambe risolte, e viene rifiutata appena una delle due promesse viene rifiutata.
Un ottimo caso d'uso di *when* è quando si vogliono combinare più chiamate Ajax. Quando si vogliono eseguire due chiamate POST in una volta e ricevere una notifica quando esntrambe sono risolte, non c'è bisogno di definire una funzione di callback separata per ogni richiesta.

```
$.when($.post('/1', data1), $.post('/2', data2))
.then(onPosted, onFailure);
```

In caso di successo, when può avere accesso agli argomenti di callback da ognuna delle sue promise costituenti. Gli argomenti sono passati come una lista con lo stesso ordine con il quale le promise sono passate a when. Se una promise fornisce molteplici argomenti di callback, tali argomenti vengono convertiti in un array.

allegare le callbacks direttamente alle promises passate a when è il modo migliore per prenderne i risultati:

```
var serverData = {};
var getting1 = $.get('/1')
.done(function(result) {serverData['1'] = result;});
var getting2 = $.get('/2')
.done(function(result) {serverData['2'] = result;});
$.when(getting1, getting2)
.done(function() {
  // the GET information is now in serverData...
});
```

when e altri metodi in jQuery, che utilizzano promise, permettono di passare a non-Promise. Queste sono trattate come promise che si sono risolte con il corrispettivo argomento. Per esempio:

```
$.when('foo')
```

Restituirà una promessa che verrà immediatamente risolta con il valore "foo".
Ma come fa when a sapere se un argomento è una promise?
jQuery controlla ogni argomento con un metodo chiamato promise, se ne esiste uno, jQuery utilizza quindi il valore restituito da tale metodo. Il metodo promise di una promise restituisce, ovviamente, se stesso.
Quindi, ad esempio, se vogliarmo realizzare una promise che si risolverà quando abbiamo recuperato alcuni dati e l'animazione # loading è completata, tutto quello che dobbiamo fare è scrivere questo:

```
var fetching = $.get('/myData');
$.when(fetching, $('#loading'));
```

Bisogna solo ricordarsi di fare questo soltanto dopo che l'animazione è partita. Se la coda di # loading è vuota la promise si risolverà immediatamente.

4.2 La Funzione Pipe

Un grande motivo per cui l'esecuzione di una serie di attività asincrone è spesso scomoda in JavaScript è che non si possono associare handler per il secondo task fino a che il primo non è completo. Come esempio, prendiamo (GET) i dati da un URL e li mettiamo (POST) in un altro:

```
var getPromise = $.get('/query');
getPromise.done(function(data) {
  var postPromise = $.post('/search', data);
});
// Now we'd like to attach handlers to postPromise...
```

Non possiamo legare callback a postPromise finché l'operazione di GET non è finita. È stata creata una chiamata POST che non possiamo fare fino a quando non abbiamo i dati che stiamo ottenendo in modo asincrono dalla chiamata GET.
È per questo che in jQuery 1.6 hanno aggiunto il metodo *pipe* alle promise. In sostanza pipe prende una callback di una promise, e restituisce una promise che rappresenta il risultato della callback.

```
var getPromise = $.get('/query');
var postPromise = getPromise.pipe(function(data) {
  return $.post('/search', data);
});
```

Pipe accetta un argomento per ogni tipo di callback: done, fail, e il progress. In questo esempio, abbiamo appena fornito una callback che viene eseguita quando getPromise è risolta. Il metodo restituisce una nuova promise che è risolta/respinta quando la promise restituita dalla nostra callback è risolto/respinta.

Si può inoltre utilizzare pipe per filtrare una promise modificando gli argomenti di callback. Se la callback di una pipe ritorna qualcosa di diverso da una promise/deferred questo valore diventa l'argomento di callback. Per esempio, se si dispone di una promise che emette notifiche di avanzamento con un numero compreso tra 0 e 1, è possibile utilizzare pipe per creare un'altra promise identica che però emette notifiche di avanzamento con una stringa leggibile:

```
var promise2 = promise1.pipe(null, null, function(progress) {
  return Math.floor(progress * 100) + '% complete';
});
```

Riassumendo, ci sono due cose che si possono fare a partire da una pipe di callback:

- Se si restituisce una promise, la promise restituita dalla pipe imiterà se stessa
- Se si restituisce un valore non-promise (o niente), la promise restituita dalla pipe sarà immediatamente risolta, rifiutata o notificata con quel valore, in base a quello che è appena successo nella promise originale.

Pipe in Cascata

Pipe non richiede di fornire ogni possibile callback, in realtà di solito si vuole scrivere:

```
var pipedPromise = originalPromise.pipe(successCallback);
```

Oppure la si vuole associare al fail. Ma cosa succede quando non abbiamo dato a pipe una callback per quello che la promise originale fa?
La promise della pipe mima la promise originaria in quei casi. Possiamo dire che il comportamento della promise scende attraverso la pipe. Questa "cascata" è molto utile perché ci permette di definire le logiche di diramazione per le attività asincrone con il minimo sforzo. Supponiamo di avere un processo in 3 fasi:

```
var step1 = $.post('/step1', data1);
var step2 = step1.pipe(function() {
  return $.post('/step2', data2);
});
var lastStep = step2.pipe(function() {
  return $.post('/step3', data3);
});
```

Qui *lastStep* verrà risolto solo quando tutte le 3 chiamate Ajax avranno successo, e verrà rifiutato se anche una delle tre chiamate fallisce. Se ci interessa solo il processo nel suo insieme possiamo omettere le dichiarazioni di variabili per le fasi precedenti:

```
var posting = $.post('/step1', data1)
  .pipe(function() {
    return $.post('/step2', data2);
  })
    .pipe(function() {
      return $.post('/step3', data3);
    });
```

O possiamo includere, equivalentemente, la seconda pipe dentro alle altre:

```
var posting = $.post('/step1', data1)
  .pipe(function() {
    return $.post('/step2', data2)
    .pipe(function() {
      return $.post('/step3', data3);
    });
  });
```

Naturalmente questo ci riporta alla *pyramid off Doom*. È necessario essere consapevoli di questo stile, ma come regola, provare a dichiare pipe delle promise individualmente. I nomi delle variabili possono non essere necessari, ma rendono il codice più auto-documentativo.

4.2.1 Rimpiazzare le Callbacks con le Promise

In un mondo perfetto, ogni funzione che ha iniziato un'attività asincrona restituisce una promise. Purtoppo, la maggior parte delle API JavaScript (comprese le funzioni native disponibili in tutti i browser e in Node.js) sono callback-based, non promise-based.
Il modo più semplice per utilizzare le promise con un API Callback-based è quello di creare una deferred e passare la sua funzione trigger come argomento di callback. Ad esempio, con una semplice funzione asincrona come setTimeout, ci piacerebbe passare il nostro metodo resolve della deferred:

```
var timing = new $.Deferred();
setTimeout(timing.resolve, 500);
```

Nei casi in cui potrebbe verificarsi un errore, l'ideale sarebbe scrivere una callback che condizionalmente indirizza un resolve o un reject. Per esempio, con una callback node-style si potrebbe fare così:

```
var fileReading = new $.Deferred();
fs.readFile(filename, 'utf8', function(err) {
  if (err) {
    fileReading.reject(err);
  } else {
    fileReading.resolve(Array.prototype.slice.call(arguments, 1));
  };
});
```

Ma, il modo migliore è definire una funzione per generare una callback node-style da ogni deferred data:

```
deferredCallback = function(deferred) {
  return function(err) {
    if (err) {
      deferred.reject(err);
    } else {
      deferred.resolve(Array.prototype.slice.call(arguments, 1));
    };
  };
}
```

Così l'esempio precedente diventa:

```
var fileReading = new $.Deferred();
fs.readFile(filename, 'utf8', deferredCallback(fileReading));
```

Le promise stanno diventando sempre più popolari, di conseguenza, molte altre librerie di JavaScript seguiranno l'esempio di jQuery e restituiranno le promise dalle loro funzioni asincrone. Per il momento la soluzione migliore, se si vuole utilizzare un generatore di promise, rimane quella appena descritta.

Capitolo 5

Flow Control e Multithreading

Fin'ora l'argomento di questa testi è stato incentrato sull'utilizzo di astrazioni per gestire le attività asincrone in tutta l'applicazione. PubSub, per esempio, è un astrazione che ci permette di distribuire gli eventi; le promise sono un astrazione che cerca di rappresentare semplici task con oggetti che possono poi essere combinati per rappresentare task più complessi.

C'è ancora un problema che riguarda l'interazione. Nella costruzione di applicazioni distribuite nasce l'esigenza di eseguire più operazioni di I/O in serie o in parallelo, Questo è un problema comune nel mondo Node che prende il nome di controllo di flusso (Flow Control).
Una delle più famose librerie che si occupa di questo, soprattutto dal punto di vista Server è Async.js

Lato client invece fin'ora sono stati descritti gli eventi come alternativa al multiThreading. Il multiThreading però è un problema quando il codice in esecuzione nei diversi thread ha accesso agli stessi dati. Anche una linea semplice come:

i++;

Rischia di essere indeterminata quando può essere modificata da più thread nello stesso momento. Ma questo in JavaScript non è possibile. D'altra par-

te, la distribuzione dei compiti su più core della CPU è sempre più essenziale in quanto influisce notevolmente sulle prestazioni della nostra applicazione. Quindi abbiamo bisogno di multithreading, ma ciò non vuol dire abbandonare la programmazione event-based.
Durante l'esecuzione di un singolo thread non è ideale, anzi è addirittura peggio, distribuire l'applicazione su più core. I sistemi multicore diventano incredibilmente lenti quando devono continuamente interagire tra loro. È molto meglio dare ad ogni core un lavoro a parte e poi sincronizzarli quando è effetivamente necessario farlo.

Questo è esattamente ciò che i workers fanno in JavaScript. Dal Thread principale della mia applicazione, si può far eseguire del codice in un thread separato, il worker può inviare messaggi (e viceversa) che prendono la forma di callback eseguite dalla coda di eventi. In breve, si interagisce con diversi thread allo stesso modo in cui si fa I/O.

5.0.2 Lato Client - Web Worker

I Web Worker sono parte dello standar di HTML5, per crearne uno, si chiama il *worker constructor* con l'URL di uno script:

```
var worker = new Worker('worker.js');
worker.addEventListener('message', function(e) {
  console.log(e.data);  // echo whatever was sent by postMessage
});
```

Qui l'interesse è ovviamente incentrato sull'attributo *data* dell'evento messaggio. Se fossimo vincolati ad usare lo stesso gestore di eventi per più workers, potremmo usare *e.target* per determinare quale lavoratore ha emesso l'evento.

La comunicazione per i workers si fa quindi in questo modo. Per comodità, l'interfaccia per parlare con i workers è simmetrica: viene usato *worker.postMessage* per inviare, e il worker utilizza *self.addEventListener ('message', ...)*, per ricevere. Ecco un esempio completo:

```
// master script
var worker = new Worker('boknows.js');
worker.addEventListener('message', function(e) {
  console.log(e.data);
});
worker.postMessage('football');
worker.postMessage('baseball');

// boknows.js
self.addEventListener('message', function(e) {
  self.postMessage('Bo knows ' + e.data);
});
```

Restrizioni dei Web Workers

I Web Worker sono destinati principalmente per gestire calcoli complessi senza compromettere la capacità di risposta DOM. Le potenziali applicazioni includono:

- Decodifica del video in streaming
- Crittografia delle comunicazioni
- Analisi del testo in un editor web-based

Negli editor web-based quando si digita del testo bisogna eseguire alcune analisi sulle stringhe prima di aggiornare il DOM con adeguate evidenziazioni della sintassi. Nel browser moderni, l'analisi viene effettuata su un thread separato, garantendo la fluidità e la reattività dell'editor.

In genere, il worker invierà il risultato dei suoi calcoli al thread principale, che dovrà quindi aggiornare la pagina. il motivo per cui il worker non aggiorna direttamente la pagina è ovviamente legato al fatto che bisogna cerare di lasciare intatte le astrazioni asincrone di JavaScript. Se un worker potesse alterare il codice della pagina, ricadremmo nelle solite problematiche di race condition e quindi dovremmo applicare mutex e semafori.

Allo stesso modo, un worker non può vedere nessun oggetto nel thread principale (o in altri thread di lavoro). Quando un oggetto viene inviato attraverso postMessage, è serializzato e deserializzato; come *JSON.parse*.

Quindi, le modifiche all'oggetto originale non influenzeranno la copia nell altro thread.
Anche l'oggetto console non è disponibile per i worker. Tutto quello che un worker può vedere è il proprio *global object*, e tutto ciò che è impacchettato in esso: gli oggetti JavaScript standard come setTimeout e le funzioni math, oltre ai metodi di Ajax del suo browser.

Per quanto riguarda Ajax, un worker può utilizzare *XMLHttpRequest* liberamente. Si possono anche utilizzare le WebSocket se il browser le supporta. Ciò significa che il worker può estrarre i dati direttamente dal server. Nel caso in cui la mole di dati da elaborare sia molta, e spesso capita in quanto i worker tra i vari utilizzi si primari si occupano anche dello streaming video, il quale ha bisogno di essere decodificato, scaricare i dati direttamente dal server è ovviamente la cosa migliore da fare, piuttosto che scaricarli dal thread principale e poi passarli al worker con postMessage serializzandoli.

5.0.3 Lato Server

Async.js

Async è un modulo che fornisce potenti funzioni per lavorare con JavaScript asincrono. Sebbene originariamente progettato per l'uso con Node.js, può anche essere utilizzato direttamente nel browser. Async fornisce circa 20 funzioni che includono i funzionali (map, reduce, filter, each...), così come alcuni modelli comuni per il Flow Control asincrono (parallel, series, waterfall...). Vediamo ora come utilizzare questi modelli per il Flow Control nel migliore dei modi.

Supponiamo di voler leggere tutti i file nella directory ricette, in ordine alfabetico, e poi concatenare il contenuto in una singola stringa e visualizzarla. Potremmo farlo facilmente usando metodi sincroni.

```
var fs = require('fs');
process.chdir('recipes');  // change the working directory

var concatenation = '';

fs.readdirSync('.')
  .filter(function(filename) {
    // ignore directories
    return fs.statSync(filename).isFile();
  })
    .forEach(function(filename) {
      // add contents to our output
      concatenation += fs.readFileSync(filename, 'utf8');
    });

console.log(concatenation);
```

ma questa operazione è ovviamente inefficiente; il problema è che non possiamo sostituire:

```
concatenation += fs.readFileSync(filename, 'utf8');
```

Con l'analogo asincrono:

```
fs.readFile(filename, 'utf8', function(err, contents) {
  if (err) throw err;
  concatenation += contents;
});
```

Perchè non c'è alcuna garanzia che le callbacks di readFile vengano eseguite nell'ordine in cui sono state fatte le chiamate di reafFile. Molto probabilmente i file più brevi saranno letti più velocemente. il risultato di come vengono lette le ricette è imprevedibile.

Se cerchiamo di risolvere questo problema senza l'ausilio di ulteriori funzioni il risultato dovrebbe essere circa questo:

```
var fs = require('fs');
process.chdir('recipes');  // change the working directory
var concatenation = '';

fs.readdir('.', function(err, filenames) {
  if (err) throw err;

  function readFileAt(i) {
    var filename = filenames[i];
    fs.stat(filename, function(err, stats) {
      if (err) throw err;
      if (! stats.isFile()) return readFileAt(i + 1);

      fs.readFile(filename, 'utf8', function(err, text) {
        if (err) throw err;
        concatenation += text;
        if (i + 1 === filenames.length) {
          // all files read, display the output
          return console.log(concatenation);
        }
        readFileAt(i + 1);
      });
    });
  }
  readFileAt(0);
});
```

Questo codice però è molto più lungo, l'ideale sarebbe avere delle alternative asincrone ai metodi *filter* e *forEach*.
Async.js ci aiuta in questo e ci da diverse alternative:

- *async.filter* e *async.foreach* trattano l'array dato in parallelo
- *async.filterSeries* e *async.foreachSeries* trattano l'array dato in esecuzione sequenziale

I motivi per il quale Async.js mette a disposizione anche dei metodi "*series*" sono sicuramente riguardanti l'ordinamento appena citato nell'esempio precedente e il numero minimo di file che Node, o qualsiasi altro processo, possono gestire simultaneamente. Se si raggiunge questo limite il sistema operativo darà errore. Invece leggendo i file in serie non avremo questa limitazione.

Il codice dell'esempio precedente si può quindi adattare:

```
var async = require('async');
var fs = require('fs');
process.chdir('recipes');  // change the working directory
var concatenation = '';

var dirContents = fs.readdirSync('.');

async.filter(dirContents, isFilename, function(filenames) {
  async.forEachSeries(filenames, readAndConcat, onComplete);
});

function isFilename(filename, callback) {
  fs.stat(filename, function(err, stats) {
    if (err) throw err;
    callback(stats.isFile());
  });
}

function readAndConcat(filename, callback) {
  fs.readFile(filename, 'utf8', function(err, fileContents) {
    if (err) return callback(err);
    concatenation += fileContents;
    callback();
  });
}

function onComplete(err) {
  if (err) throw err;
  console.log(concatenation);
}
```

Ora il nostro codice si divide esattamente in due parti: il compito generale (nella forma delle chiamate async.filter e async.forEachSeries) e i dettagli di implementazione (nella forma di due funzioni di iterazione e una callback finale)

Filter e forEach non sono gli unici metodi che Async.js mette a disposizione, ci sono anche i seguenti:

- *Reject/rejectSeries*: l'inverso di filter
- *Map/mapSeries*: per trasformazioni 1:1
- *Reduce/reduceRight*: per trasformare un valore ad ogni passo
- *Detect/detectSeries*: per la ricerca di un valore corrispondente di un filtro
- *SortBy*: per generare una copia ordinata

- *Some*: per verificare che almeno un valore corrisponde al criterio dato
- *Every*: per verifycare che tutti i valori corrispondon al criterio dato

Organizzare i Task con Async.js

Async.js non offre solo metodi per lavorare con i dati parallelamente o in serie, offre anche strumenti per fare il dispatching delle funzioni asincrone raccogliendone i dati.
Supponiamo di avere una serie di fuzioni asincrone che vogliamo vengano eseguite in ordine. Senza l'uso di altre funzioni si potrebbe scrivere una cosa simile:

```
funcs[0](function() {
  funcs[1](function() {
    funcs[2](onComplete);
  })
});
```

Async.js ci da a disposizione però le funzioni Async.Series e Async.waterfall che prendono una serie di funzioni e le eseguono in modo sequenziale. Passando a ognuna una callback NodeStyle. La differenza tra i due è che async.series fornisce solo le callback per ogni task, mentre async.waterfall fornisce anche i risultati del task precedente. (per risultati si intende il valore nonerror che ogni task passa alla sua callback)
Vediamo un semplice esempio con timeOut:

```
var async = require ('async');

var start = new Date;

async.series([
  function(callback) { setTimeout(callback, 100); },
  function(callback) { setTimeout(callback, 300); },
  function(callback) { setTimeout(callback, 200); }
], function(err, results) {
  // show time elapsed since start
  console.log('Completed in ' + (new Date - start) + 'ms');
});
```

La chiamata a console.log verrà eseguita dopo circa 600ms perché ogni task nell'array è completato in ordine. La callback che async.js passa ad ogni funzione controlla semplicemente la presenza di un errore.

Async.js offre un analogo parallelo ad async.series, si chiama async.parallel. Prende come input una serie di funzioni nella forma *function(callback)...*, più un opzionale handler di completamento che viene lanciato dopo che l'ultima callback è stata eseguita.
L'esempio precedente del timeOut si può modificare con questo:

```
var async = require ('async');
var start = new Date;
async.parallel([
  function(callback) { setTimeout(callback, 100); },
  function(callback) { setTimeout(callback, 300); },
  function(callback) { setTimeout(callback, 200); }
], function(err, results) {
  console.log('Completed in ' + (new Date - start) + 'ms');
});
```

mentre async.series prende la somma dei timeout (600ms) asynce.parallel prende il massimo (300ms).
Convenientemente, async.js passa i risultati all'handler di completamento nell'ordine corrispondente all'array, non nell'ordine in cui sono stati generati i risultati. Così si ottengono i benefici delle prestazioni del parallelismo senza l'imprevedibilità.

Gestire le Code con Async.queue

Le funzioni viste in precedenza vanno bene nella maggiorn parte dei casi, ma async.series e async.parallel hanno però le loro limitazioni:

- L'array dei task è statico, una volta chiamati async.series o async.parallel non è possibile aggiungere o rimuovere task
- Non c'è un modo per sapere quanti compiti sono stati completati, è come avere una scatola nera, a meno che i compiti stessi non inviino degli aggiornamenti
- Si è limitati o ad una non-concorrenza, o ad una concorrenza illimitata. Questo è un problema importante quando si tratta di file I/O. Se stiamo operando su migliaia di file, non vogliamo essere inefficienti e fare una serie, ma se facciamo partire il parallelo probabilmente il sistema operativo si bloccherà

Async fornisce un metodo versatile che affronta ognuno di questi problemi: *async.queue*.

L'interfaccia di async.queue è un po' più complessa di quella di async.series e async.parallel. Prende una funzione chiamata worker (anziché un array di funzioni) ed un valore (il numero massimo di attività simultanee lavoratore può elaborare)e di conseguenza restituisce una coda. Ecco un esempio:

```
var async = require('async');

function worker(data, callback) {
  console.log(data);
  callback();
}
var concurrency = 2;
var queue = async.queue(worker, concurrency);
queue.push(1);
queue.push(2);
queue.push(3);
```

Non è importante il valore della variabile *concurrency*, l'output sarà sempre 1,2,3.
Il valore della variabile non incide sull'output ma con *concurrency* a 2, bisognerà fare due viaggi nella coda di eventi, con *concurrency* a 1, i viaggi sarebbero 3, con *concurrency* maggiore o uguale a 3 il viaggio attraverso la coda è uno solo. Una coda con la *concurrency* a 0 non farà nulla. È possibile ottenere la massima concorrenza assegnando a *concurrency* il valore *infinity*.

Async mette a disposizione anche il metodo *push* attraverso il quale è possibile aggiungere dinamicamente dei task alla coda attraverso il codice: *push(task,[callback])*; la callback viene chiamata una volta che il worker ha finito di elaborare il task. Inoltre, è possibile anche aggiungere un'array di funzioni invece di un singolo task, in questo caso la rispettiva callback verrà richiamata ogni volta che uno dei task passati termina.

Come con async.series e i suoi simili, possiamo dare ad async.queue un handler di completamento. Invece di passarlo come argomento, però, abbiamo bisogno di attaccarlo come una proprietà chiamata *drain*.

Async.queue fornisce oltre a *drain* anche altri eventi:

- Quando l'ultimo compito è stato avviato, la coda chiama *emtpy*. (Al termine dell'attività, la coda chiama drain)
- Quando il limite di concorrenza è stato raggiunto, queue chiama *saturated*

Async.queue è dunque un ottimo strumento che Async.js mette a disposizione dell'utente, è particolarmente indicato soprattutto quando si vogliono esegure un gran numero di attività asincrone con concorrenza limitata.

Capitolo 6

Conclusioni

JavaScript è uno dei linguaggi più usati a livello Web, soprattutto per la costruzione di Web-app. Il motivo di questa popolarità è da ricercare nella grande qualità di JavaScript di poter eseguire lo stesso codice su più macchine, anche architetturalmente molto diverse. Questo nel mondo eterogeneo di internet è un fattore decisamente importante.
In quest'elaborato sono stati visti i problemi principali della programmazione in JavaScript, si è cercato prima di capire a fondo l'event-loop e i problemi derivanti da esso, e successivamente si sono viste diverse soluzioni a casi d'uso comuni. Nella fattispecie si è visto come l'utilizzo di *PubSub* torna utile per la propagazione degli eventi in tutta l'applicazione e come le *promise*, che rappresentano una singola attività, siano semplici e intuitive da implementare e da combinare.
Successivamente invece si è visto come poter eseguire più compiti in parallelo anche in un linguaggio, JavaScript appunto, che è single-thread. Lato client si sono studiati i *Web Worker* e si è visto come poter eseguire compiti più complessi in altri thread senza avere i problemi dati dal parallelismo, ad esempio il race condition.
Lato Server invece si è visto come la libreria *Async.js* offre all'utilizzatore potenti strumenti per poter eseguire più funzioni asincrone, in parallelo o in serie, anche senza perdere l'ordinamento originale delle funzioni.

Tutte le soluzioni viste nell'elaborato vengono proposte per semplificare la vita al programmatore, in quanto la costruzione di web-app con un lin-

guaggio ad eventi come JavaScript è sicuramente poco intuitiva, soprattutto per chi è abituato a linguaggi come Java o C++. È per questo che negli ultimi anni sono stati sviluppati linguaggi alternativi a JavaScript come **DART** e **TypeScript** per cercare di risolvere le difficoltà di programmazione.
DART nasce da Google e il suo intento è quello di risolvere i problemi di JavaScript offrendo al tempo stesso migliori prestazioni, la possibilità di sviluppare più facilmente strumenti utili alla gestione di progetti di grandi dimensioni e migliori funzionalità legate alla sicurezza.
TypeScript invece è un superset di JavaScript in quanto estende la sua sintassi, in questo modo qualunque programma scritto in JavaScript è anche in grado di funzionare con TypeScript senza nessuna modifica. TypeScript nasce dal crescente bisogno di un linguaggio front-end per lo sviluppo di applicazioni JavaScript su larga scala. TypeScript estende JavaScript includendo classi, interfacce e moduli e rendendo quindi JavaScript più simile a linguaggi meglio conosciuti dai programmatori.

Ringraziamenti

Grazie a tutti quelli che mi hannno supportato e sopportato in questi anni
Grazie agli amici di sempre, ai compagni di università, agli amici d'infanzia ritrovati, ai Good Ole Boys, ai ragazzi del quarto e a tutti quelli che si vedono a San Pietro e poi decidono
Grazie alla Sofi per l'aiuto e la pazienza
Grazie alla mia famiglia per tutto il resto

Async JavaScript - Trevor Burnham 2012

Cos'era il Web 1.0? - FabMad www.fabmad.it 2013

Cos'è l'HTML - Bartolini www1.mat.uniroma1.it 2006

jQuery Mobile sposta il web in 320pixel - Mario Concina www.marioconcina.it 2010

Concurrency model and Event Loop - developer.mozilla.org 2013