La rappresentazione dei dati nel web: XML e JSON

Rev 5.1 del 21/10/2024

XML e JSON

La codifica della pagina: Charset

I file html vengono normalmente salvati in formato **UTF-8 senza BOM** (intestazione).
UTF-8 è un formato che salva :

- i caratteri ascii base (dallo 0 al 127) su un byte
- i rimanenti caratteri su 2 bytes in formato Unicode

L'intestazione (BOM) serve ad avvisare il lettore riguardo al formato del file, però può interferire con la gestione server per cui normalmente viene omessa ed il formato utilizzato viene scritto in testa al file tramite un apposito meta tag html:

```
<meta charset="UTF-8">
```

Il tipo di contenuto: Internet Media Types

Il **media-type** indica il tipo di informazioni contenute all'interno del file :

```
<meta http-equiv="content-type" content="text/html”>
```

IANA manages the official registry of **media types** The identifiers were originally defined in **RFC 2046**, and were called **MIME types** because they referred to the non-ASCII parts of email messages that were composed using the MIME specification (**Multipurpose Internet Mail Extensions**).
They are also sometimes referred to as **Content-types**.
Their use has expanded from **email** sent through SMTP, to other protocols such as HTTP, and others.
New media types can be created with the procedures outlined in **RFC 6838**.

Text Type, for human-readable text and source code.

text/plain: Textual data; Defined in RFC 2046 and RFC 3676
text/html: HTML; Defined in RFC 2854
text/css: Cascading Style Sheets; Defined in RFC 2318
text/xml: Extensible Markup Language; Defined in RFC 3023
text/csv: Comma-separated values; Defined in RFC 4180
text/rtf: RTF; Defined by Paul Lindner
Elenco completo : http://en.wikipedia.org/wiki/Internet_media_type

text/javascript JavaScript; Defined in and made obsolete in RFC 4329 in order to discourage its usage in favor of **application/javascript.** However, text/javascript is allowed in HTML 4 and 5 and, unlike application/javascript, has cross-browser support.
application/json dati JSON serializzati

The "type" attribute of the <script> tag in HTML5 is optional and there is no need to use it at all since all browsers have always assumed the correct **default,** even before HTML5.

La trasmissione dei dati

I dati viaggiano attraverso la rete **sempre** in formato stringa.
Quando un client riceve una stringa di dati dalla rete, per poterli elaborare li deve trasformare in oggetto.
Quando invece deve trasmettere un oggetto in rete, prima di trasmetterlo lo deve trasformare in stringa.

- Il processo di trasformazione di una stringa in un oggetto si definisce **parsificazione** (parsing)
- Il processo inverso di trasformazione di un oggetto in stringa si definisce **serializzazione**

XML

Un documento XML è un documento strutturato a tag esattamente come i documenti HTML.
E' sostanzialmente costituito da un albero avente una **radice** (equivalente al tag **<html>**) con dei tag interni di **primo livello** che possono contenere al loro interno dei tag di **secondo livello** i quali possono contenere nodi di **terzo livello** e così via.

- Ogni tag può avere uno o più attributi.
- I valori terminali (cioè il contenuto finale di un nodo interno) sono detti **foglie** dell'albero

```
<bookstore>
      <book category="cooking">
            <title lang="en">Everyday Italian</title>
            <author>Giada De Laurentiis</author>
            <year>2005</year>
            <price>30.00</price>
      </book>

      <book category="children">
            <title lang="en">Harry Potter</title>
            <author>J K. Rowling</author>
            <year>2005</year>
            <price>29.99</price>
      </book>

      <book category="web">
            <title lang="en">XQuery Kick Start</title>
            <author>James McGovern</author>
            <author>Per Bothner</author>
            <author>Kurt Cagle</author>
            <author>James Linn</author>
            <author>Vaidyanathan Nagarajan</author>
            <year>2003</year>
            <price>49.99</price>
      </book>

      <book category="web" cover="paperback">
            <title lang="en">Learning XML</title>
            <author>Erik T. Ray</author>
            <year>2003</year>
            <price>39.95</price>
      </book>
</bookstore>
```

Parsificazione di una stringa XML in un oggetto XML

```
let xmlString = `<bookstore> …… </bookstore>`

let parser=new DOMParser();
let xmlDoc=parser.parseFromString(xmlString, "text/xml");
```

In questo modo viene creato un nuovo oggetto **xmlDoc** con all'interno l'intero albero XML.
xmlDoc è in pratica l'equivalente del **document** HTML.
Per trovare un qualunque tag si può partire da **xmlDoc**. querySelector() / querySelectorAll()

Serializzazione di un oggetto XML in stringa

```
let serializer = new XMLSerializer();
let xmlString = serializer.serializeToString(xmlDoc);
```

Navigazione di un albero XML

Un oggetto XML è un albero esattamente come il DOM di una pagina HTML ed è pertanto navigabile tramite gli stessi metodi e le stesse proprietà, compreso **innerHTML**

Per scorrere un albero XML occorre utilizzare un puntatore che punterà inizialmente alla root dell'albero e che poi scorrerà via via lungo tutti i nodi dell'albero.

Accesso alla radice dell'albero

```
let xmlRoot = xmlDoc.firstElementChild    // equivalente del tag <html>
let xmlRoot = xmlDoc.documentElement
let xmlRoot = xmlDoc.getElementsByTagName("bookstore")[0];
```

Navigazione

```
for (let i=0; i<xmlRoot.children.length;i++){
      let book = xmlRoot.children[i];
      let category =    book.getAttribute("category");
      let title = book.firstElementChild.textContent;
      let authors = "";
      for (let author of book.querySelectorAll("author")
                authors += author.textContent + " - "
      alert (title + '\n' + category + '\n' + authors);
```

Creazione di nuovi nodi

Occorre necessariamente partire da ***xmlDoc*** Partendo da ***document*** ovviamente **NON funziona !**

```
let element = xmlDoc.createElement("element");
xmlRoot.appendChild(element);
```

Creazione di un nuovo nodo a partire da una stringa

```
let book = '<book category="web" cover="paperback">
              <title lang="en">Learning XML</title>
              <author>Erik T. Ray</author>
            </book>'
```

si può procedere in 2 modi:

```
1)  xmlRoot.innerHTML += book
2)  const xmlBook = parser.parseFromString(book, "text/xml");
    xmRoot.appendChild(xmlBook);
```

Creazione di un nuovo oggetto xmlDoc

```
let xmlDoc = document.implementation.createDocument("", "", null);
```

Il primo parametro indica un eventuale namespace da anteporre al documento
Il secondo parametro rappresenta un ulteriore prefisso opzionale
Il terzo parametro indica il tipo di documento (Document Type)

```
let xmlRoot = xmlDoc.createElement("root");
xmlDoc.appendChild(xmlRoot);
```

Nota sulla gestione degli attributi

A differenza di **HTML** dove c'è un ***default state*** statico (accessibile tramite getAttribute / setAttribute) ed ***current state*** dinamico 'sovrastante' (accessibile tramite l'operatore puntino),
in **XML** esiste soltanto un default state statico, per cui gli attributi XML non sono accessibili tramite il puntino ma *soltanto* tramite getAttribute / setAttribute.

Vettori Associativi

Sono vettori che al posto dell'indice numerico usano una **chiave** alfanumerica (nell'esempio pippo, pluto e minnie).

```
let vect = new Array(); // oppure let vect=[] raccomandato perchè più veloce
vect['pippp']  = "descrizione di pippo";
vect['pluto']  = "descrizione di pluto";
vect['minnie'] = "descrizione di minnie";
```

I valori dei vettori associativi vengono salvati mediante una tecnica di **hash** legata alla chiave, cioè la **descrizione** viene posizionata in una ben precisa locazione di memoria (non sequenziale) la cui posizione è determinata univocamente dal valore della chiave.

Per cui il principale vantaggio di un vettore associativo rispetto ad un normale vettore enumerativo di record è la **possibilità di accedere ai contenuti in modo diretto tramite chiave**. Nel caso dei vettori enumerativi, per trovare 'pippo' occorre eseguire una ricerca sequenziale all'interno del vettore di record. Nel caso invece degli array associativi la descrizione di 'pippo' può essere acceduta in modo diretto:

Una oggetto simile in C# sono i Dictionary

Nota: E' anche possibile, una volta definito il vettore, definire alcune celle tramite chiave, ed altre tramite indice, cioè creare un vettore misto in cui il ciclo **`for in`** scandisce le celle associative, mentre il ciclo **`for of`** scandisce le celle enumerative. I due gruppi sono completamente distinti. Modalità pessima.

JSON : Java Script Object Notation

Per la scrittura di un vettore associativo esiste anche una sintassi alternativa, molto compatta, che è la cosiddetta **Java Script Object Notation** (abbreviato JSON). Le righe precedenti potrebbero essere riscritte anche nel modo seguente del tutto equivalente, in cui i vari campi vengono scritti in formato **`chiave : valore`** e separati da una virgola, definendo di fatto un nuovo record

```
let record = {
    "pippo" : "descrizione di pippo",
    "pluto" : "descrizione di pluto",
    "minnie" : "descrizione di minnie"
};
```

Questa sintassi è abbastanza simile alle **`struct`** del C, però mentre una **`struct`** definisce soltanto il tracciato di un record in cui **i dati** verranno inseriti successivamente, nel caso dei JSON struttura e dati vengono definiti allo stesso momento. Si tratta sostanzialmente di un Object che viene dichiarato ed istanziato allo stesso tempo cioè ancora **si definisce un oggetto a partire dal suo contenuto.**

La variabile record contiene (ovviamente) un **riferimento** all'oggetto.

Terminologia e sintassi

Si chiamano:

- **chiave** la stringa che identifica l'elemento
- **valore** il contenuto associato alla chiave

- **Le chiavi devono essere scritte con le virgolette doppie** (in taluni casi possono essere utilizzate anche le virgolette semplici o addirittura senza virgolette, ma è fuori standard)
- **I valori** possono utilizzare indifferentemente assegnazioni dirette oppure il contenuto di una variabile

Esempio : anagrafica di uno studente

```
let studente = {
    "nome" : "mario",
    "cognome" : "rossi",
    "eta" : 16,
    "studente" : true,
    "images" : ["smile.gif", "grim.gif", "frown.gif", "bomb.gif"],
    "hobbies" : [],                  // vettore al momento vuoto
    "pos": { "x": 40, "y": 300 },    // oggetto annidato

    "stampa" :   function () { alert("Hello " + this.nome); },
    "fullName" : function () { return this.nome + " " + this.cognome; }
};
```

In fase di scrittura diretta di un JSON (come nell'esempio precedente) **non è consentito parametrizzare una chiave con il nome di una variabile.** Anche se si omettono le virgolette, il testo viene comunque interpretato come "**Nome di chiave**" (oppure, in alcuni ambienti, genera un errore).
L'accesso parametrizzato tramite una variabile è invece possibile con la sintassi indicate di seguito.

Accesso in lettura ai campi di un JSON

Si può accedere **al valore di una chiave** tramite due sintassi equivalenti:

- la sintassi dei vettori associativi (parentesi quadre)
- la sintassi degli oggetti con il puntino come separatore di campo.

```
let eta = studente["eta"]; // sintassi dei vettori (più generale)
let eta = studente.eta;    // sintassi degli oggetti (no PHP, no TypeScript)
```

Il primo caso presenta il vantaggio di consentire un accesso parametrizzato al campo.
Cioè, invece di specificare direttamente il nome del campo, è possibile utilizzare una variabile :

```
let myVar = "nome";
alert(studente[myVar]);        // "mario"
```

Se si cerca di accedere in lettura ad una chiave inesistente, il risultato sarà **`undefined`**
Per cui prima di accedere ad un chiave bisognerebbe sempre controllare la sua esistenza in uno dei seguenti modi:

```
if(studente[key])   alert(studente[key]);
if(key in studente)  alert(studente[key]);
if(studente.hasOwnProperty(key))  // la chiave esiste e non è ereditata
```

Confronto con XML

In **XML** per ricercare una chiave (nodeElement) occorre utilizzare ***`querySelector()`*** il quale esegue sostanzialmente una ricerca sequenziale, mentre in JSON è possibile accedere in modo diretto ad una chiave attraverso un algoritmo di hash:

```
XML : let nome = xmlDoc.querySelector("strNome")
JSON: let nome = studente.strNome
```

Per cui, rispetto ad XML

- JSON consente un accesso diretto alle chiavi tramite hash, quindi molto più veloce
- JSON presenta una scrittura molto più concisa, dunque un file JSON pesa molto di meno e quindi è decisamente preferibile quando deve essere scambiato attraverso una rete.

Accesso in scrittura e aggiunta di nuove chiavi

```
studente["eta"]=18;  // sovrascrive il valore precedente
```

Per aggiungere una nuova chiave **NON è ammesso l'utilizzo del metodo `.push()`** (che vale solo per i vettori enumerativi) ma è sufficiente accedere in scrittura ad una chiave inesistente e la nuova chiave viene automaticamente creata.

```
studente["indirizzo"]="Fossano";
studente.indirizzo="Fossano"; // sintassi non ammessa in TypeScript per creare nuove chiavi
```

Attenzione che **non** è possibile creare direttamente nuove sotto-chiavi :

```
studente["indirizzo"]["citta"]="Fossano"
studente["indirizzo"]["via"]="San Michele, 68"
```

ma occorre necessariamente eseguire uno step intermedio;

```
studente["indirizzo"] = {}
studente["indirizzo"]["citta"]="Fossano"
studente["indirizzo"]["via"]="San Michele, 68"
```

Rimozione di una chiave

```
delete studente["eta"]
```

`delete` può anche essere utilizzato per cancellare una cella da un vettore enumerativo.
Però attenzione che, a differenza di splice, NON modifica le dimensioni dell'Array. Viene cancellato il contenuto ma la cella rimane con valore undefined. Es : `delete studente[3]`

Dichiarazione di un Array e di un JSON

1)
```
let persona = new Array();  // vettore generico (associativo o enumerativo)
let persona = [];  // equivalente
```

2)
```
let persona = new Object();   // JSON (vettore associativo)
let persona = {};  // equivalente
```

Le due sintassi non sono del tutto equivalenti. La 1° è preferibile per i vett enumerativi, la 2°per i json.

Parsing e Serializzazione di uno stream JSON

Il metodo statico **`JSON.parse(str)`** consente di parsificare una stringa jSON convertendola in oggetto
Il metodo statico **`JSON.stringify(obj)`** consente di serializzare un oggetto o un vettore di oggetti nella stringa corrispondente.

```
// server
let json = {"name": "John Doe", "age": 42}
let s = JSON.stringify(json);
// ------ trasmissione dei dati al client ----------
// client
let json = JSON.parse(s);
```

In realtà il metodo **stringify** presenta la seguente sintassi completa:

```
let jsonText = JSON.stringify(json, null, 2);
```

Il 2° parametro è detto **rimpiazzo** e consente di applicare un filtro sui campi da visualizzare (o rimpiazzarli con altri valori)
Il 3° parametro è detto **formattatore** e consente di formattare il json in un formato **human readable** in cui ogni livello interno presenta un livello di indentazione pari al valore indicato (nell'esempio 2 chr)

Le specifiche JSON

Le specifiche JSON sono definite all'interno dello standard **ECMA Script 5** (ES5).
Le principali regole sono le seguenti:

- Non è ammesso usare solo numeri come nome di chiave (*ovviamente*)
- I **nomi dei campi** DEVONO essere scritti come stringhe (cioè racchiusi tra doppi apici).
- Per le stringhe (chiavi e valori) NON sono ammessi gli apici singoli ma SOLO i **doppi apici**
  Ad esempio Express accetta SOLO stringhe json scritte in questo modo. Altri ambienti (es jQuery) accettano anche chiavi senza virgolette e ammettono l'utilizzo delle virgolette semplici.
- **Numeri** e **Booleani** devono essere scritti modo diretto (senza doppi apici). Attenzione che, aggiungendo i doppi apici, NON sono più NUMERI o BOOLEANI, ma diventano STRINGHE

Esempi di jSON valido

```
let person = {
      "name"      : "Nicolas",
      "age"       : 22,
      "date"      : [01, 09, 1992],   // vettore
      "student"   : true,
      "info"      : {"web": "myPage.it", "mail": "myMail@vallauri.edu"}
};
```

Anche le seguenti variabili sono considerati JSON validi

```
let n = 5
let s = "salve mondo"
```

questo perché in realtà viene eseguito un **boxing** automatico del tipo:

```
let N = new Object(5);
let S = new Object("salve mondo");
```

In entrambi i casi, al momento del **JSON.stringify()**, l'intero contenuto verrà racchiusa all'interno di un ulteriore apice singolo, apice singolo che verrà poi rimosso al momento del `JSON.parse()`.

Viceversa **`let s = 'salve mondo'`** non è considerato un JSON valido

Unboxing

```
let s = N.toString();      // "72"
let n = N.valueOf();       //  72
```

jsonformatter

Il sito **jsonformatter** consente di validare l'esatta sintassi di una qualunque stringa jSon.

Vettori di JSON

Sono sostanzialmente analoghi ai **vettori di record** disponibili nei linguaggi strutturati, però nel caso dei json ogni record può avere una sua struttura diversa dalla struttura degli altri record:

```
let people = [
   { "name" : "Nicolas",  "age" : 22  },
   { "name" : "Piero",    "age" : 29,  residenza="Fossano" }
];
```

Scansione di un vettore enumerativo di JSON

Si può utilizzare indifferentemente un ciclo ***FOR-OF*** oppure un ciclo ***forEach***
Il ciclo ***forEach*** è più veloce e consente di accedere anche all'indice **i**

```
for (let person of people)
     console.log(person.name)

let vet =["item1", "item2", "item3", "item4", "item6"]
vet.forEach ( function (item, i)  { console.log(item, i) });
```

Confronto fra JSON

A differenza delle variabili primitive, gli Object vengono passati alle funzioni per riferimento.
Questo vale anche per le assegnazioni che copiano il puntatore e NON l'intero record:

```
let studente2 = studente1;     // copia il riferimento
studente2.nome = "joe";        // anche studente1.nome conterrà "joe";
```

Allo stesso modo **occorre prestare attenzione nell'eseguire un confronto diretto fra due json**
Il confronto verrebbe eseguito sui puntatori e NON sul contenuto !
Il confronto può comunque essere eseguito sulle rispettive serializzazioni !

Il metodo includes

Il metodo includes tipico dei vettori enurativi, può operare anche su vettori di object, però attenzione che **confronta i puntatori** !

Si consideri la seguente istruzione:

```
if(people.includes({"name":"Mario",  "age" : 33 }))  => FALSE
perchè i puntatori non coincidono
```

invece:

```
let obj = {"name":"Piero",  "age" : 20 }
people.push(obj)
if(people.includes(obj))  => TRUE :  i due puntatori puntano allo stesso oggetto
```

Idem per **indexOf** e tutti gli altri metodi simili relativi ai vettori enumerativi.

Altra possibile sintassi per la scansione di un vettore di JSON

Anche il ciclo for-of consente di accedere, oltre all'item corrente, anche all'indice numerico i dell'item corrente. Per poter accedere anche all'indice occorre impostare il ciclo su **vet.entries**().

```
let vet =["item1", "item2", "item3", "item4", "item6"]
for(const [i, item] of vet.entries())  console.log(i, item)
```

La scansione delle chiavi

La scansione delle chiavi di un JSON (con relativo contenuto) può essere eseguita tramite il ciclo **for** key **in**

```
for (let key in studente)
    console.log(key, studente[key]);
```

Nota a differenza di C#, `key` non è un oggetto, ma una semplice stringa che contiene il nome della chiave. **Non** è pertanto possibile scrivere `alert(key.value)`
ma occorre scrivere `alert(studente[key]);`

Il vettore enumerativo delle chiavi

Il metodo statico **`Object.keys`** restituisce un **vettore enumerativo** di tutte le chiavi presenti all'interno di un vettore associativo:

```
let keys = Object.keys(studente);
keys.forEach(function(key){
    console.log(key, studente[key])  })
```

Sintassi alternativa per la Scansione delle chiavi

Esiste anche un ciclo che consente di leggere simultaneamente una chiave ed il suo relativo contenuto:

```
for(let [key, value] of Object.entries(studente))
    console.log(key, value)
```

in cui sostanzialmente abbiamo già il **`value`** senza doverlo accedere mediante `studente[key]`

Le dimensioni di un object

Nel caso dei vettori associativi / object la proprietà **`length`** non è definita (restituisce *undefined*)**.** Infatti, trattandosi di un oggetto, non ha senso parlare della sua lunghezza. Volendo però valutare il numero dei campi presenti, si può utilizzare il **vettore enumerativo delle chiavi** che, essendo un vettore enumerativo, dispone della proprietà length:

```
if(Object.keys(studente).length != 0)

// oppure in jQuery
jQuery.isEmptyObject({});
```

Forma Breve per la scrittura di un JSON

Utilizzo in scrittura

```
let residenza = "fossano"
let myJson = {"nome":"pippo", "eta":16, residenza};
```
la riga precedente crea una nuova chiave avente il nome della variabile e come contenuto il suo valore

```
let data = {nome, eta, res}   equivale a   {"nome":nome, "eta":eta, "res":res }
```

Utilizzo in lettura

Si supponga di partire dal precedente **myJson** contenente le tre chiavi nome, eta, residenza

```
const nome = myJson.nome          // pippo
const {nome, eta} = myJson        // pippo, 16
```

L'ultima riga crea **due variabili** ***nome*** e ***eta*** che avranno come valore il contenuto dei campi ***nome*** e ***eta*** all'interno del json. Se una chiave non esiste la variabile corrispondente sarà undefinded.

Esempio: sia ***data*** un json restituito da una richiesta ajax;
```
const data = await inviaRichiesta()
const {squadre, piloti} = data
```
Creo cioè due variabili `squadre e piloti` a partire da `data`

In lettura è anche possibile assegnare un nuovo nome alla variabile:
```
const {nome:nuovoNome, eta:nuovaEta} = myJson      // nuovo Nome = "pippo"
```

L'oggetto Map

Object e Map sono basate sullo stesso principio: salvano i dati in formato key:value

Object: `{1: 'smile', 2: 'cry', 42: 'happy'}`
Map: `{[1, "smile"], [2, "cry"], [42, "happy"]}`

Ogni item della map è costituito da un vettore enumerativo di due elementi: la chiave ed il valore

Differenze fra Object e Map

- Negli **Object** la chiave può essere soltanto un numero intero o una stringa. Nelle **Map** può essere qualunque cosa: un vettore, un altro object, etc.
- Nelle **Map** gli elementi sono ordinati sulla base dell'ordine di inserimento. Negli **Object** no.
- **Map** è una sottoclasse derivata da **Object**. Object rappresenta invece la superclasse da cui eredita Map. Per cui Map è una istanza di Object, mentre Object NON è una istanza di Map
- Un **Object** può essere istanziato in diversi modi:
```
let obj = {};          //Empty object
let obj = {id:1, name:"Test object"};
let obj = new Object(); //Empty Object
```
- Una **Map** può essere istanziata soltanto tramite costruttore che si aspetta come parametro un vettore enumerativo di coppie `[key,value]`
```
let map = new Map(); //Empty Map
let map = new Map([[1,2],[2,3]]); // map = {1=>2, 2=>3}
```

Quando è preferibile Object e quando Map

- L'**object** è la scelta ideale quando c'è bisogno di una struttura semplice e veloce, perché la creazione di un oggetto e l'accesso alle proprietà tramite una chiave specifica sono **molto più veloci** della creazione di Mappa e del relativo accesso
- L'**object** è preferibile anche laddove è necessario applicare una logica separata a una singola proprietà / elemento della collezione

```
let obj = {
    id: 1,
    name: "It's Me!",
    print: function(){
        return `Object Id: ${this.id}, with Name: ${this.name}`;
    }
}
```

Questo non è fattibile con le Map

- **Map** è preferibile in scenari che richiedono molte aggiunte e rimozioni di nuove chiavi e per grandi quantità di dati. Inoltre Map, a differenza di Object, mantiene l'ordine delle chiavi e quindi garantisce prestazioni stabili in tutti i browser.

Principali metodi dell'oggetto Map

```
map.get(1)        // 2
map.has(1);       // return true if key exists
map.set(4,5);     // {1=>2, 2=>3, 4=>5}
let ok = map.delete(1);  //{ 2=>3, 4=>5}
console.log(ok); // true
map.clear();      // remove ALL elements from a Map object
console.log(map.size);  // number of keys
```

Scansione di un oggetto Map

Le mappe sono iterabili, cioè supportano il metodo for of

```
for (const item of map){
    console.log(item);
    // Array[2,3]
    // Array[4,5]
}
```

oppure

```
for (const [key,value] of map){
    console.log(`key: ${key}, value: ${value}`);
    // key: 2, value: 3
    // key: 4, value: 5
}
```

oppure

```
map.forEach((value, key) => console.log(`key:${key}, value:${value}`))
    // key: 2, value: 3
    // key: 4, value: 5
```