JavaScript

정규표현식 – 김근형 강사

정규표현식

- 정규표현식
  - 정규 표현식은 문자열에 나타는 특정 문자 조합과 대응시키기 위해 사용되는 패턴이다.
  - 자바스크립트에서, 정규 표현식 또한 객체이다.
  - 이 패턴들은 RegExp의 exec 메소드와 test 메소드 ,그리고 String의 match메소드 , replace메소드 , search메소드 , split 메소드와 함께 쓰인다.

정규표현식

- 정규식을 만드는 두가지 방법
  - 정규식 리터럴(슬래쉬"/"로 감싸는 패턴)을 사용하는 방법

```
var re = /ab+c/;
```

    - 정규식 리터럴은 스크립트가 불러와질 때 컴파일된다.
    - 만약 정규식이 상수라면, 이렇게 사용하는 것이 성능을 향상시킬 수 있다.

  - RegExp 객체의 생성자 함수를 호출하는 방법

```
var re = new RegExp("ab+c");
```

    - 생성자 함수를 사용하면 정규식이 실행 시점에 컴파일된다.
    - 정규식의 패턴이 변경될 수 있는 경우, 혹은 사용자 입력과 같이 다른 출처로부터 패턴을 가져와야 하는 경우에는 생성자 함수를 사용할 것.

정규표현식

- 정규식 패턴 작성하기
  - 정규식 패턴은 /abc/ 같이 단순 문자로 구성될 수도 있고, /ab*c/ 또는 /Chapter (\d+)\.\d*/와 같이 단순 문자와 특수 문자의 조합으로 구성될 수 있다.
  - 단순 패턴 사용하기
    - 단순 패턴은 문자열을 있는 그대로 대응시키고자 할 때 사용된다.
    - 위의 패턴은 "Hi, do you know your abc's?" 와 "The latest airplane designs evolved from slabcraft." 두가지 예에서 부분 문자열 'abc'에 대응된다. ' Grab crab ' 이라는 문자열에서 ' ab c ' 라는 부분 문자열을 포함하고 있지만, ' abc ' 를 정확하게 포함하고 있지 않기 때문에 대응되지 않는다.
  - 특수 문자 사용하기
    - 검색에서 하나 이상의 b들을 찾거나, 혹은 공백을 찾는 것과 같이 '있는 그대로의 대응' 이상의 대응을 필요로 할 경우, 패턴에 특수한 문자를 포함시킨다.
    - /ab*c/ 패턴은 'a' 문자 뒤에 0개 이상의 'b' 문자(* 문자는 바로 앞의 문자가 0개 이상이라는 것을 의미한다)가 나타나고 바로 뒤에 ' c ' 문자가 나타나는 문자 조합에 대응된다. 문자열 " cbbabbbbcdebc, " 에서 위의 패턴은 부분 문자열 ' abbbbc ' 와 대응된다.

정규표현식

- 정규식에서의 특수문자

캐릭터	뜻	캐릭터	뜻
\	특수문자 인식 (예 : \b, *)	x|y	'x' 또는 'y'에 대응된다.
^	입력 시작 부분 표시	{n}	앞 표현식이 n번 나타나는 부분에 대응된다
$	입력 끝 부분 표시	{n,m}	n과 m은 양의 정수이고, n <= m를 만족해야 한다. 앞 문자가 최소 n개, 최대 m개가 나타나는 부분에 대응된다. m이 생략된다면, m은 ∞로 취급된다.
*	앞의 표현식이 0회 이상 연속으로 반복	[xyz]	문자셋(Character set) – 괄호안의 어떤 문자에 대응할 경우 사용. 하이폰을 사용하여 문자 범위 지정이 가능하다.
+	앞의 표현식이 1회 이상 연속으로 반복	[^xyz]	부정 문자셋(negated character set) 또는 보충 문자셋(complemented character set) - 괄호 내부에 등장하지 않는 어떤 문자와도 대응된다.
?	앞의 표현식이 0회 혹은 1회 출력	[\b]	백스페이스(U+0008)에 대응된다
.	단일 문자 대응(개행 문자 제외)	\b	단어 경계에 대응된다.
(x)	x의 패턴에 대입되고 해당 내용을 기억한다.(포획괄호)	\B	단어 경계가 아닌 부분에 대응된다.
(?:x)	x의 패턴에 대입되고 해당 내용을 기억하지 않는다.(비포획괄호)	\cX	문자열 내부의 제어 문자에 대응됩니다. 여기서 X는 A에서 Z까지의 문자 중 하나이다.
x(?=y)	오직 'y'가 뒤따라오는 'x'에만 대응된다	\d	숫자 문자에 대응된다. [0-9]와 동일하다.
x(?!y)	'x'뒤에 'y'가 없는경우 에만 'x'에 일치한다	\D	숫자 문자가 아닌 문자에 대응됩니다. [^0-9]와 동일하다

정규표현식

- 정규식에서의 특수문자

캐릭터	뜻	캐릭터	뜻
\f	폼피드 (U+000C) 문자에 대응	\v	수직 탭(U+000B) 문자에 대응
\n	줄 바꿈 (U+000A) 문자에 대응	\w	밑줄 문자를 포함한 영숫자 문자에 대응된다. [A-Za-z0-9_] 와 동일
\r	캐리지 리턴(U+000D) 문자에 대응	\W	단어 문자가 아닌 문자에 대응된다. [^A-Za-z0-9_] 와 동일
\s	스페이스, 탭, 폼피드, 줄 바꿈 문자등을 포함한 하나의 공백 문자에 대응	\n	정규식 내부의 n번째 괄호에서 대응된 부분에 대한 역참조. 여기서, n은 양의 정수이다.
\S	공백 문자가 아닌 하나의 문자에 대응	\0	널 (U+0000)문자에 대응한다.
\t	탭 (U+0009) 문자에 대응		

정규표현식

- 괄호를 사용
  - 정규식 내부의 일부를 둘러싼 괄호는, 해당 부분에서 대응된 문자열을 기억하는 효과를 갖는다.
  - 기억된 문자열은 이후 패턴화된 부분 문자열 일치 사용하기에서 설명한 것처럼 다른 곳에서 사용하기 위하여 불러와질 수 있다.
  - 패턴 /Chapter (\d+)\.\d*/ 패턴으로 "Open Chapter 4.3, paragraph 6"를 대응 할 경우 '4'가 저장된다.
  - 부분 문자열을 대응시키면서도 해당 부분을 기억하지 않으려면, 괄호의 첫머리에 ?:패턴을 사용한다.

정규표현식

- 정규식 사용 메서드

메서드명	설명
exec	대응되는 문자열을 찾는 RegExp 메소드. 정보를 가지고 있는 배열을 반환한다. 대응되는 문자열을 찾지 못했다면 null을 반환한다.
test	대응되는 문자열이 있는지 검사하는 RegExp 메소드. true 나 false를 반환한다.
match	대응되는 문자열을 찾는 String 메소드. 정보를 가지고 있는 배열을 반환한다. 대응되는 문자열을 찾지 못했다면 null을 반환한다.
search	대응되는 문자열이 있는지 검사하는 String 메소드. 대응된 부분의 인덱스를 반환한다. 대응되는 문자열을 찾지 못했다면 -1을 반환한다.
replace	대응되는 문자열을 찾아 다른 문자열로 치환하는 String 메소드.
split	정규식 혹은 문자열로 대상 문자열을 나누어 배열로 반환하는 String 메소드.

  - 문자열 내부에 패턴과 대응되는 부분이 있는지 알고 싶다면, test 나 search 메소드를 사용한다.
  - 좀 더 많은 정보를 원하면 (대신 실행이 느림) exec 나 match 메소드를 사용한다.

정규표현식

- 정규식 사용 예제

```
const tel1 = '0101234567팔';
const tel2 = '987654321';

// 정규 표현식 리터럴
const myRegExp1 = /^[0-9]+$/;
const myRegExp2 = new RegExp("^[0-9]+$")

console.log(myRegExp1.test(tel1)); // false
console.log(myRegExp1.test(tel2)); // true

console.log(myRegExp2.test(tel1)); // false
console.log(myRegExp2.test(tel2)); // true
```

정규표현식

- 정규식 사용 예제

```
const targetStr = 'This is a pen.';
const regexr = /is/ig;

// RegExp 객체의 메소드
console.log(regexr.exec(targetStr)); // [ 'is', index: 2, input: 'This is a pen.' ]
console.log(regexr.test(targetStr)); // true

// String 객체의 메소드
console.log(targetStr.match(regexr)); // [ 'is', 'is' ]
console.log(targetStr.replace(regexr, 'IS')); // ThIS IS a pen.
// String.prototype.search는 검색된 문자열의 첫번째 인덱스를 반환한다.
console.log(targetStr.search(regexr)); // 2 ← index
console.log(targetStr.split(regexr));  // [ 'Th', ' ', ' a pen.' ]
```

```
▶ ["is", index: 2, input: "This is a pen.", groups: undefined]
true
▶ (2) ["is", "is"]
ThIS IS a pen.
2
▶ (3) ["Th", " ", " a pen."]
```

정규표현식

- exec를 통해 나오는 데이터

```
var myRe = /d(b+)d/g; // 또는, var myRe = new RegExp('d(b+)d', 'g');
var myArray = myRe.exec('cdbbdbsbz');
console.dir(myRe);
console.dir(myArray);
```

Object	Property or index	Description	In this example
myArray		매칭된 문자열과 기억된 부분 문자열	["abbd", "bb"]
	index	매칭된 문자열 인덱스 (0부터 시작)	1
	input	원래의 문자열	"cdbbdbsbz"
	[0]	매칭된 문자열	"abbd"
myRe	lastIndex	다음 매칭을 시작할 인덱스	5
	source	정규식 패턴	"d(b+)d"

```
▼/d(b+)d/g
    dotAll: (...)
    flags: (...)
    global: (...)
    ignoreCase: (...)
    multiline: (...)
    source: (...)
    sticky: (...)
    unicode: (...)
    lastIndex: 5
  ▶__proto__: Object
▼Array(2)
    0: "dbbd"
    1: "bb"
    index: 1
    input: "cdbbdbsbz"
    groups: undefined
    length: 2
  ▶__proto__: Array(0)
```

정규표현식

- 플래그
  - 정규식은 다섯 개의 플래그를 설정해줄 수 있으며, 이를 통해 전역 검색 또는 대소문자 구분 없는 검색을 수행할 수 있다.
  - 이 플래그들은 각기 사용될 수도 있고 함께 사용될 수도 있고 순서에 구분이 없다.
  - 사용방법은 아래와 같다.

```
// 정규식에 플래그 포함
var re = /pattern/flags;

// RegExp를 활용한 플래그 사용
var re = new RegExp("pattern", "flags");
```

정규표현식

- 플래그 종류

플래그명	설명
g	문자열 내의 모든 패턴을 검색
I	대소문자 구분 없는 검색
m	문자열의 행이 바뀌더라도 검색을 계속한다
s	.에 개행 문자도 매칭(ES2018)
u	유니코드; 패턴을 유니코드 코드 포인트의 나열로 취급한다.

정규표현식

- 플래그 예제

```
const targetStr = 'Is this all there is?';

// 문자열 is를 대소문자를 구별하여 한번만 검색한다.
let regexr1 = /is/;
let regexr2 = new RegExp("is");

console.log(targetStr.match(regexr1)); // [ 'is', index: 5, input: 'Is this all there is?' ]
console.log(targetStr.match(regexr2));

// 문자열 is를 대소문자를 구별하지 않고 대상 문자열 끝까지 검색한다.
regexr1 = /is/ig;
regexr2 = new RegExp("is","ig");

console.log(targetStr.match(regexr1)); // [ 'Is', 'is', 'is' ]
console.log(targetStr.match(regexr1).length); // 3

console.log(targetStr.match(regexr2)); // [ 'Is', 'is', 'is' ]
console.log(targetStr.match(regexr2).length); // 3
```

```
▶ ["is", index: 5, input: "Is this all there is?", groups: undefined]
▶ ["is", index: 5, input: "Is this all there is?", groups: undefined]
▶ (3) ["Is", "is", "is"]
3
▶ (3) ["Is", "is", "is"]
3
```