12장 정규표현식

12.1 정규 표현식의 기본

12.1.1 정규 표현식

- 문자열의 패턴을 표현하기 위한 도구
- 유닉스의 스크립트 언어처리에 사용되던 표기법 (grep, sed, awk)
- 정규 표현식을 내장한 프로그래밍 언어들 (Perl, Java, Javascript, PHP, Python, Ruby)
- 자바스크립트의 정규표현식은 Perl의 정규 표현식을 받아들인 것

12.1.2 정규 표현식의 생성

- RegExp 생성자 혹은 정규표현식 리터럴로 RegExp 객체를 생성
- 정규표현식 리터럴을 강추 (p.457)
    - 이유: RegExp 생성자는 이스케이프(\)를 다루는게 까다로움

```
const reg = new RegExp("abc"); // 생성자
const reg = /abc/; // 리터럴
```

12.1.3 패턴 매칭

패턴 매칭 = 문자열이 정규표현식과 일치하는지 확인하는 작업

12.1.4 RegExp 객체의 메서드

정규표현식을 사용하는 메서드들

- String.prototype: match, replace, search, split
- RegExp.prototype: test, exec
    - test: 정규표현식과 문자열이 일치하는지 true, false를 반환
    - exec: 정규표현식과 일치하는 문자열을 검색하여 배열로 반환

```
const reg = /Script/;
console.log(reg.test("Javascript"));
console.log(reg.exec("Javascript"));
```

12.2 기본적인 정규 표현식 패턴 작성하기

12.2.1 리터럴 문자와 메타 문자

- 리터럴 문자 = 일반 문자 (p.456)
    - \0 \n \t \v \f \r
- 메타 문자 = 정규표현식에서 특별한 뜻을 갖는 구분자
    - ^ $ \ . * + ? ( ) { } |

    - 메타 문자를 리터럴 문자로 표기할 때는 문자 앞에 \를 붙여서 이스케이프한다.

12.2.2 문자 클래스와 부정문자 클래스

문자 클래스: [...]

- 안에 있는 문자가 한번 이상 사용되었는지 확인
- -를 통해 범위 표시: [a-z]
- -를 통해 문자 자체 표시 [-ab]

```
const reg = /[abc]/;
const rangeReg = /[a-c]/;
const wordReg = /[-ac]/;
console.log(reg.test('mac'));
console.log(reg.test('window'));
console.log(rangeReg.test('mac'));
console.log(rangeReg.test('window'));
console.log(wordReg.test('mac'));
console.log(wordReg.test('window'));
```

질문: [ab]와 [-ab]의 차이를 모르겠다.

부정문자 클래스: [^...]

- [...] 의 여집합
- []안에 있는 것으로만 이루어지면 안됨

```
const reg = /[^0-9]/;
console.log(reg.test("137")); // false
console.log(reg.test("137a")); // true
```

12.2.3 문자 클래스의 단축 표기

- . : \n을 제외한 임의의 문자
- \d : [0-9], 숫자
- \D : [^0-9], 숫자 외 문자
- \w: [a-zA-Z0-9], word
- \W: [^a-zA-Z0-9]
- \s: 공백문자
- \S: 공백문자가 아닌 문자
- \b: 영단어의 경계 위치
    - '\b': 백스페이스 *질문: 이게 뭐지? 출력 가능할까?*
    - /b/: 영단어의 경계 위치 (p.465)

12.2.4 반복패턴

- {m,n}: 앞의 요소를 최소 m번, 최대 n번 반복

- {m,}: 앞의 요소를 최소 m번 반복
- {,n}: 앞의 요소를 최소 0번 최대 n번 반복
- {n}: 앞의 요소를 최소 1번 최대 n번 반복
- ?: {0,1}
- +: {1,}
- *: {0,}

욕심 많은, 욕심없는 반복

```
const greedyReg = /Script.*/;
const nonGreedyReg = /Script.*?/;
console.log(greedyReg.exec('JavaScriptScriptScript')); //
ScriptScriptScript를 반환
console.log(nonGreedyReg.exec('JavaScriptScriptScript')); // Script 를 반환
const reg = /0*?1/;
console.log(reg.exec('000012')); // 00001을 반환
```

12.2.5 그룹화와 참조

- 그룹화: (...)
- 부분 정규표현식: 그룹화된 정규표현식
- 캡처링: 부분 정규표현식과 일치한 값을 별도로 저장하는 동작
    - \1을 통해 첫 번째 부분 정규 표현식을 참조
    - (?:...): ?:를 통해 캡처링 없는 그룹화도 가능. 이게 어디에 쓰이는 지는 잘 모르겠음.

```
const headerReg1 = /<h[1-6]>.*<\/h[1-6]>/;
const headerReg2 = /<(h[1-6])>.*<\/\1>/;
const headerReg3 = /<(h[1-6])>.*<\/\2>/;
const text1 = '<h1>Javascript</h1>';
const text2 = '<h1>Javascript</h2>';
const text3 = '<h1><h2>Javascript</h2>';
const text4 = '<h1><h2>Javascript</h2></h1>';
console.log(headerReg1.test(text1));
console.log(headerReg1.test(text2));
console.log(headerReg1.test(text3));
console.log(headerReg1.test(text4));
console.log('\n');
// \1이 부분 정규 표현식과 일치한 값을 참조
console.log(headerReg2.test(text1));
console.log(headerReg2.test(text2));
console.log(headerReg2.test(text3));
console.log(headerReg2.test(text4));
console.log('\n');
console.log(headerReg3.test(text1));
console.log(headerReg3.test(text2));
console.log(headerReg3.test(text3));
console.log(headerReg3.test(text4));
console.log('\n');
```

```
// 부분 정규 표현식 문자열도 표시
console.log(headerReg1.exec(text3));
console.log(headerReg2.exec(text3));
```

12.2.6 위치를 기준으로 매칭하기 | 앵커

앵커: 문자열의 위치를 패턴트로 지정

- ^: 문자열의 시작

```
const reg = /^Java/;
console.log(reg.test('Javascript'));
console.log(reg.test('ScriptJava'));
```

- $: 문자열의 마지막

```
const reg = /Java$/;
console.log(reg.test('Javascript'));
console.log(reg.test('ScriptJava'));
```

- \b: 영어 단어의 경계, 한글도 영어 단어 외의 문자로 간주

```
const reg1 = /cat/;
const reg2 = /\scat\s/;
const reg3 = /\bcat\b/;
console.log(reg1.test('저는 cat을 좋아합니다.')); // true
console.log(reg2.test('저는 cat을 좋아합니다.')); // false
console.log(reg3.test('저는 cat을 좋아합니다.')); // true
```

- \B: 영어 단어 경계 외의 위치

```
const Bdog = /\Bdog/;
const Bdog = /dog\B/;
console.log(Bdog.test("Blulldog"));
console.log(Bdog.test("dog"));
console.log(Bdog.test("I love dog"));
console.log(Bdog.test("doggy"));
console.log(Bdog.test("who likes the dog sound?"));
```

- x(?=pattern): 전방 탐색, x 다음에 pattern 이 나옴

```
const frontSearch = /Java(?=Script)/;
console.log(frontSearch.exec("JavaScript"));
```

```
console.log(frontSearch.exec("Java Script"));
console.log(frontSearch.exec("JavaWowScript"));
```

- x(?!pattern): 전방 부정 탐색, x 다음에 pattern 이 나오지 않음

```
const notFrontSearch = /Java(?!Script)/;
console.log(notFrontSearch.exec("JavaScript"));
console.log(notFrontSearch.exec("Java Script"));
console.log(notFrontSearch.exec("JavaWowScript"));
```

12.2.7 선택 패턴: |

```
const countFruits = /\b(\d+) (apple|peach|orange)s?\b/;
console.log(countFruits.exec('100 apples'));
console.log(countFruits.exec('100 peaches'));
```

12.2.8 플래그

- i: insensitive, 대소문자를 구별하지 않음
- g: global, 전역 검색
- m: multiple, 여러줄 검색
- y: 시작 위치 고정 검색
- u: 정규표현식 패턴을 유니코드 열로 처리

```
const reg1 = /^cat/ig;
const reg2 = new RegExp('^cat', 'im');
console.log(reg1.test("Dog\nCat\nMonkey"));
console.log(reg2.test("Dog\nCat\nMonkey"));
```