1. **http통신과 소켓통신은 어떻게 틀린가?**

HTTP는 HTML 문서와 같은 리소스들을 가져올 수 있도록 해주는 프로토콜입니다. HTTP는 웹에서 이루어지는 모든 데이터 교환의 기초이며, 클라이언트-서버 프로토콜이기도 합니다.

프로토콜이란 컴퓨터 내부에서, 또는 컴퓨터 사이에서 데이터의 교환 방식을 정의하는 규칙 체계입니다.

클라이언트-서버 프로토콜이란 (보통 웹브라우저인) 수신자 측에 의해 요청이 초기화되는 프로토콜을 의미합니다.

WebSocket이란 사용자의 브라우저와 서버 사이의 동적인 양방향 연결 채널을 구성하는 HTML5 프로토콜이다.

HTTP 통신

* Client의 요청(Request)이 있을 때만 서버가 응답(Response)하여 해당 정보를 전송하고 곧바로 연결을 종료하는 방식이다.
* Client가 요청을 보내는 경우에만 Server가 응답한다.
* Server로부터 응답을 받은 후에는 연결이 바로 종료된다. 실시간 연결이 아니고 필요한 경우에만 Server로 요청을 보내는 상황에 유용하다.
* 요청을 보내 Server의 응답을 기다리는 어플리케이션(Android or Ios)의 개발에 주로 사용된다.

Socket 통신

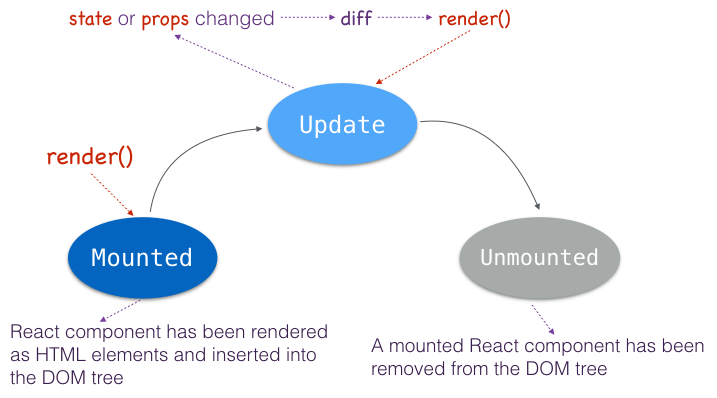
* Server와 Client가 특정 port를 통해 실시간으로 양방향 통신을 하는 방식이다.
* Server와 Client가 계속 연결을 유지한다.
* Server와 Client가 실시간으로 데이터를 주고받는 상황이 필요한 경우에 사용된다.
* 실시간 동영상 Streaming이나 온라인 게임 등과 같은 경우에 자주 사용된다.

출처 : <https://hees-dev.tistory.com/53>

<https://developer.mozilla.org/ko/docs/Web/HTTP/Overview>

1. **컴포넌트의 각 생명주기는 어떻게 되고 각 시점은 언제 호출되는가?**

컴포넌트의 생명주기는 아래와 같이 3가지의 시점(마운트, 업데이트, 언마운트)으로 나뉜다.



1. **마운트**

리액트 컴포넌트가 Instance로 생성되어(실제로 구현되고 메모리에 할당되면서) DOM tree에 삽입되어 브라우저 상에 나타나는 것을 마운트라고 한다.

(Instance란 설계도(class)를 바탕으로 소프트웨어 세계에 구현된 구체적인 실체이다. 실체화된 인스턴스는 메모리에 할당된다. Instance는 원본으로부터 생성된 복제본을 뜻한다.)

출처 : <https://dongkka.tistory.com/22>(인스턴스의 의미)

마운트의 경우 맨 처음 컴포넌트가 렌더될 때 이루어지며, 다시 props나 state가 변경되어 렌더될 때는 마운트가 되지 않는다.

마운트 시 호출하는 메서드

① constructor

컴포넌트를 새로 만들 때마다 호출되는 클래스 생성자 메서드이다. Constructor는 this.state의 초기값 적용, 인스턴스에 이벤트 처리 메서드를 바인딩하기 위해 사용한다.

주의!!! Constructor 구현 시에 super(props)를 가장 먼저 호출해야 한다.

Constructor 내부에서 setState를 호출하면 안된다.

② static getDerivedStateFromProps

Props에 있는 값을 state에 동기화하는 메서드이다. (react v 16.3 ~) 최초 마운트 시와 갱신 시 render( ) 메서드를 호출하기 직전에 호출되고 state 갱신을 위한 객체나 null을 반환한다. nextProps와 prevState에 따라 component의 작업여부를 결정한다.

주의!!! 이 메서드에서는 컴포넌트 인스턴스에 접근할 수 없다.

이 메서드는 부모 컴포넌트가 다시 렌더링을 발생시켰을 때 실행되고, 해당 컴포넌트 내에서 지역적인 setState가 발생한 경우 실행되지 않는다.

③ render

클래스 컴포넌트에서 구현하는 메서드로 UI를 렌더링하는 메서드이다. Render 메서드는 React 엘리먼트, 배열과 Fragment, Portals, 문자열과 숫자, Boolean 또는 null 중에 하나를 반환해야 한다.

주의!!! Render 함수는 컴포넌트의 state를 변경하면 안된다.

호출할 때마다 동일한 결과를 반환해야 한다.

브라우저와 직접 상호작용하지 않는다.

④ componentDidMount

컴포넌트가 DOM tree에 삽입된 직후에 호출되는 메서드이다. 외부에서 데이터를 불러와야 할 때 사용하기 적절하다.

주의!!! componentDidMount에서 setState를 호출하는 경우 render 함수가 두 번 호출되므로 성능 문제가 있을 수 있다.

1. **업데이트**

컴포넌트가 업데이트되는 경우는 다음의 4가지입니다.

1. Props가 바뀔 때
2. State가 바뀔 때
3. 부모 컴포넌트가 리렌더링될 때
4. 강제로 트리거를 발생시킬 때(this.forceUpdate)

리액트의 state나 props 변화가 있을 때는 렌더될 때 컴포넌트가 다시 마운트되는 게 아니므로, shouldComponentUpdate(nextProps, nextState)나, componentWillReceiveProps(nextProps) 쪽에서 그 변화를 감지해 관련 작업을 해줘야 하는 것이다.

업데이트 시 호출하는 메서드

①Static getDerivedStateFromProps

②shouldComponentUpdate

컴포넌트가 다시 렌더링을 해야 할 지 말아야 할 지 결정하는 메서드이다. 초기렌더링 또는 forceUpdate( ) 호출 시에 이 메서드는 호출되지 않는다. 이 메서드는 **렌더링을 방지하여 성능을 최적화**하는 목적으로 사용된다.

③Render

④getSnapshotBeforeUpdate

render 메서드 호출 후 DOM 변화를 반영하기 직전에 호출되는 메서드이다. 이 메서드에서 return하는 값을 componentDidUpdate에서 3번째 파라미터로 받아올 수 있다.

⑤componentDidUpdate

리렌더링을 완료한 후 실행되는 메서드이다. 최초렌더링에서는 호출되지 않는다. 컴포넌트가 업데이트되었을 시에 DOM을 조작하기 위해 사용한다.

주의!!! componentDidUpdate에서 setState를 사용하면 무한 렌더링이 될 수 있으므로 조건문을 잘 작성해야 한다.

1. **언마운트**

리액트 컴포넌트가 DOM 상에서 제거되는 것을 **언마운트**라고 한다.

언마운트 시 호출하는 메서드

componentWillUnmount

컴포넌트가 DOM에서 제거되기 직전에 호출되는 메서드이다. 타이머를 제거하거나 데이터 구독 해제 등의 목적으로 사용된다.

주의!!! componentWillUnmount가 호출된 컴포넌트는 다시 렌더링하지 않으므로 setState를 호출하면 안된다.

출처 : <https://medium.com/humanscape-tech/react-%EC%BB%B4%ED%8F%AC%EB%84%8C%ED%8A%B8-%EC%83%9D%EB%AA%85%EC%A3%BC%EA%B8%B0-c7f45ef2d0be>

<https://ideveloper2.tistory.com/97>

1. **자바스크립트에서 사용하는 자료구조에는 어떤 것들이 있는가?**

자료구조는 데이터의 표현 및 저장 방법을 의미한다. 자료구조에는 연결리스트, 스택, 큐 등이 있는데 자바스크립트의 배열은 훌륭하게도 이 모든 것을 표현할 수 있다.

① 연결 리스트(Linked List)

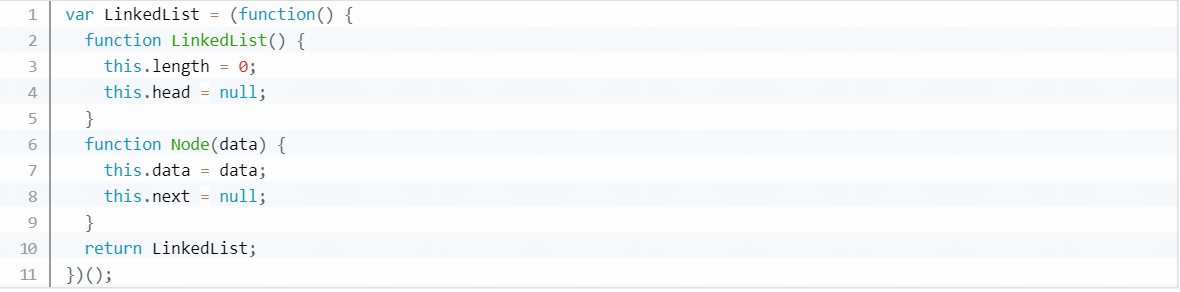
연결 리스트는 여러 개의 노드로 이루어져 있다. 각각의 노드는 데이터와 다음 노드가 무엇인지 알려주는 주소를 가지고 있다. 또한 연결 리스트는 새로운 데이터를 추가하거나, 데이터의 위치를 찾거나, 제거하는 기능이 있어야 한다.

1->2->3->4->5 라는 연결 리스트가 있다면 1, 2, 3, 4, 5는 데이터고 ->는 주소이다.

자바스크립트에서는 이미 연결 리스트가 구현되어 있다. 바로 **배열**이다.

[1, 2, 3, 4, 5]가 있으면 1, 2, 3, 4, 5는 데이터이고, array[0], array[1] 등은 데이터가 담긴 위치를 말해주고 있다. 또한, array.push( )를 통해 데이터를 추가할 수 있고, array.splice( )를 통해 데이터를 제거할 수 있다.

연결 리스트와 노드를 직접 구현해보면,



LinkedList에는 length와 head가 있다. Length는 노드의 개수를 표현하는 부분이고, head가 바로 첫 노드의 주소를 가리키는 부분이다.

이제 데이터를 추가, 검색, 삭제하는 메소드를 구현해보자.

var LinkedList = (function( ) {

function LikedList( ) {

this.length = 0;

this.head = null;

}

function Node(data) {

this.data = data;

this.next = null;

}

LinkedList.prototype.add = function(value) {

var node = new Node(value);

var current = this.head;

if (!current) { // 현재 아무 노드도 없으면

this.head = node; // head에 새 노드를 추가합니다.

this.length++;

return node;

} else { // 이미 노드가 있으면

while(current.next) { // 마지막 노드를 찾고.

current = current.next;

}

current.next = node; // 마지막 위치에 노드를 추가합니다.

this.length++;

return node;

}

};

LinkedList.prototype.search = function(position) {

var current = this.head;

var count = 0;

while (count < position) { // position 위치만큼 이동합니다.

current = current.next;

count++;

}

return current.data;

};

LinkedList.prototype.remove = function(position) {

var current = this.head;

var before;

var remove;

var count = 0;

if (position == 0) { // 맨 처음 노드를 삭제하면

remove = this.head;

this.head = this.head.next; // head를 두 번째 노드로 교체

this.length--;

return remove;

} else { // 그 외의 다른 노드를 삭제하면

while (count < position) {

before = current;

count++;

current = current.next;

}

remove = current;

before.next = remove.next;

// remove 메모리 정리

this.length--;

return remove;

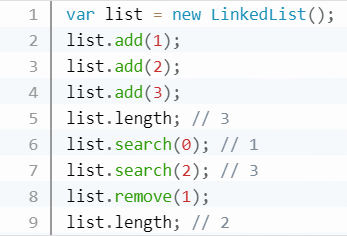
}

};

return LinkedList;

})();

테스트 결과

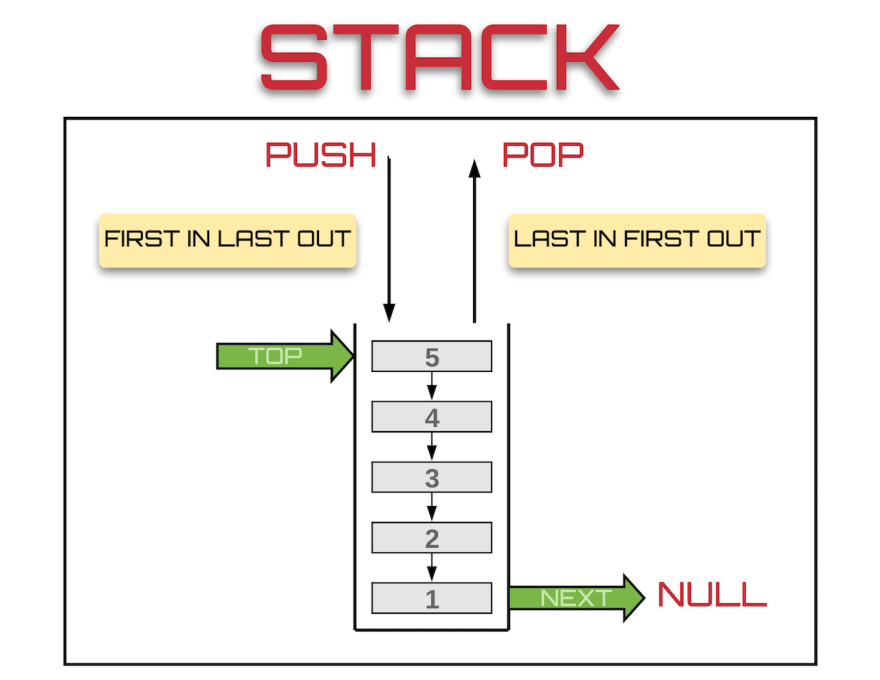


출처 : <https://www.zerocho.com/category/Algorithm/post/58008a628475ed00152d6c4d>

② 스택(Stack)

LIFO(Last In, First Out) 구조이다.

* 삽입: push
* 삭제: pop
* 읽기: peek



Data Structure in <https://dev.to/theoutlander/implementing-the-stack-data-structure-in-javascript-4164>

스택에서 데이터를 넣는 것을 push, 데이터를 꺼내는 것을 pop이라고 한다. 스택의 마지막요소를 알려주는 것을 stackTop으라고 한다. 그리고 push와 pop을 하는 위치를 top이라 하고, 스택의 가장 아랫부분을 bottom이라고 한다.

스택을 직접 객체를 사용해서 만들어보자.

var Stack = (function( ) {

function Stack( ) {

this.top = null;

this.count = 0;

}

function Node(data) {

this.data = data;

this.next = null;

}

Stack.prototype.push = function(data) {

var node = new Node(data);

node.next = this.top;

this.top = node;

return ++this.count;

};

Stack.prototype.pop = function( ) {

if (!this.top) { // stack underflow 방지

return false;

}

var data = this.top.data;

this.top = this.top.next;

// 예전 this.top의 메모리 정리

this.count--;

return data;

};

Stack.prototype.stackTop = function( ) {

return this.top.data;

};

return Stack;

})( );

테스트 해보자.

var stack = new Stack( );

stack.push(1);

stack.push(3);

stack.push(5);

stack.pop( );

stack.stackTop( );

출처 : <https://medium.com/@jinseok.choi/%EC%8A%A4%ED%83%9D-stack-48a22e52268b>

<https://www.zerocho.com/category/Algorithm/post/5800b79e1dfb250015c38db6>

③ 큐(Queue)

큐는 실생활에서 줄이라고 생각하면 된다. 우리가 순서를 기다릴 때 줄을 서죠? 새로 온 사람은 줄 맨 뒤에 서고, 제일 앞 사람은 필요한 행동을 한 후 빠진다. 이렇게 뒤에서 들어가고(enqueue) 앞에서 빠지는(dequeue) 구조입니다.

FIFO(First In, First Out) 구조이다.

* push(item) : item을 리스트의 끝부분에 추가한다. // enqueque
* unshift( ) : 리스트의 첫 번째 항목을 제거한다. // dequeque
* front : 제일 앞의 데이터를 알 수 있다.

이제 코드로 큐를 만들어보자.

var Queue = (function( ) {

function Queue( ) {

this.count = 0;

this.head = null;

this.rear = null;

}

function Node(data) {

this.data = data;

this.next = null;

}

Queue.prototype.enqueue = function(data) {

var node = new Node(data);

if (!this.head) {

this.head = node;

} else {

this.rear.next = node;

}

this.rear = node;

return ++this.count;

};

Queue.prototype.dequeue = function( ) {

if (!this.head) { // stack underflow 방지

return false;

}

var data = this.head.data;

this.head = this.head.next;

// this.head 메모리 클린

--this.count;

return data;

};

Queue.prototype.front = function( ) {

return this.head && this.head.data;

};

return Queue;

})( );

테스트해보자.

var queue = new Queue( );

queue.enqueue(1); // 1

queue.enqueue(3); // 2

queue.enqueue(5); // 3

queue.dequeue( ); // 1

queue.front( ); // 3

출처 : <https://www.zerocho.com/category/Algorithm/post/580b9b94108f510015524097>

④ Map

key-value로 이루어진 자료구조이다. O(1)의 시간이 소요된다. Object와 유사하지만 중요한 차이점이 있다. Object는 프로토타입을 가지고 있으므로, 의도치 않은 key 충돌이 있을 수 있다. 하지만 map은 명시적으로 제공된 key만을 가지고 있다. 또한 Object의 key는 삽입 순서를 보장하지 않지만, map은 삽입 순으로 순회한다. 또한 map의 key 값으론, 정말 아무거나 다 사용할 수 있다.

⑤ Set

map은 key를 넣으면 value를 준다.

set은 값만 모여있고, 중복값을 허용하지 않는다.

ex)

const foo = new Set([1, 2, 3, 4, 3, 2, 4, 5]);

console.log(foo); // Set(5) {1, 2, 3, 4, 5}

⑥ WeakMap, WeakSet ?

출처 : <https://junwoo45.github.io/2020-05-09-data_structure/>