Reactjs

2015년 8월 31일 월요일 오전 9:53

React.js 란?

읽기전에...

이 문서는 koba04님이 작성한 React.js Advent Calendar를 번역한 것입니다. 본래 원문서는 캘린더 형식으로 소개하지만 여기에서는 회를 나눠 작성할 생각입니다. 또한, React 버전 0.12.1 때 작성된 문서이기 때문에 현 버전과 다른 점이 있을 수 있습니다. 최대한 다른 부분을 노트로 작성할 생각이지만, 만약 생략된 부분이 있다면 댓글로 알려주시면 감사하겠습니다.

올해(2014년) 들어 갑자기 대세가 된 듯한 React.js 지만, "조금 전까지만 해도 Angular.js가 대세라고 하더니!"라며 혼란스러워하는 사람도 많을 거라 생각해서 Advent Calendar 형식으로 간단히 소개하고자 합니다. React.js에서 중요한 개념인 VIRTUAL DOM(가상돔) 별도의 Adevent Calendar에 작성돼 있으니 꼭 봐주시길 바랍니다.

왜 이 라이브러리가 뜨거운 감자가 됐는지는 솔직히 잘 모르겠지만, 필자는 개인적으로는 Github의 atom에서 성능 향상의 이유로 React.js를 사용하기로 했다는 기사(Moving Atom To React)를 보고 흥미를 갖게 됐습니다.

React.js는 Facebook이 만들고 있는 이른바 MVC 프레임워크에서의 뷰 부분을 컴포넌트로 만들기 위한 라이브러리입니다. Handlebars 같은 템플릿 엔진이 아닙니다. Facebook은 물론 instagram, AirBnb, Yahoo, Atlassian 등 여러 곳에서 사용하고 있습니다.

특징

공식 사이트에서는 특징을 크게 세 가지로 나눠 소개하고 있습니다.

- JUST THE UI
- VIRTUAL DOM
- DATA FLOW

이 세 가지 특징에 관해서 간단히 설명해 드리겠습니다.

JUST THE UI

React.js는 UI 컴포넌트를 만들기 위한 라이브러리입니다. 컴포넌트 지향 프레임워크는 여러 가지가 있지만 React.js는 정말 UI 컴포넌트만 지원합니다. 비록 지원하는 범위는 작지만, 애플리케이션을 만드는 방법을 크게 바꿀 수 있다는 점이 재미있습니다. 또한, 이해 비용이 적어 도입하기 쉬우며 Backbone.js의 뷰 부분을 React.js로 구현하거나 Angular.js의 directives를

React.js를 사용해 구현하는 등 여러 환경과 조합해 사용할 수 있습니다.

VIRTUAL DOM

React.js는 자바스크립트 내에 DOM Tree와 같은 구조체를 VIRTUAL DOM으로 갖고 있습니다. 다시 그릴 때는 그 구조체의 전후 상태를 비교하여 변경이 필요한 최소한의 요소만 실제 DOM에 반영합니다. 따라서 무작위로 다시 그려도 변경에 필요한 최소한의 DOM만 갱신되기 때문에 빠르게 처리할 수 있습니다.

DATA FLOW

React.js는 단방향 데이터 흐름을 지향합니다. 따라서 Angular.js의 양방향 데이터 바인딩을 사용할 때처럼 작성할 코드의 양이 확연히 줄거나 하지는 않습니다. 그렇지만, 애플리케이션의 데이터를 관리하는 모델 컴포넌트가 있고 그 데이터를 UI 컴포넌트에 전달하는 단순한 데이터 흐름으로 이해하고 관리하기 쉬운 애플리케이션을 만들 수 있습니다.

역자노트

과거엔 데이터가 변경되면 전체를 새로 그리는 간편하고 단순한 방법으로 애플리케이션을 구현했습니다. 현대에 들어 애플리케이션을 개발하는 방법이 많이 복잡해졌다고 생각합니다. Angular.js의 양방향 데이터 바인딩은 코드를 줄여주고 사용하기 편하지만, 규모가 커질수록 데이터의 흐름을 추적하기 힘듭니다. React.js는 근원(根源)으로 돌아가는 개발 방법입니다. 그리고 그 과정에서 발생하는 비효율적인 부분, 예를 들어 DOM 전체를 갱신해야하는 문제를 VIRTUAL DOM과 비교(diff)로 해결했습니다.

기타

JSX

추가로 두 가지 더 설명하겠습니다. 첫 번째로 JSX입니다. React.js에서는 JSX라고 하는 XML과 비슷한 문법을 이용할 수 있습니다. 이는 선택적으로 사용할 수 있는 문법이므로 JSX가 마음에 들지 않는다면 자바스크립트로 작성할 수도 있습니다. JSX는 다음에 자세히 소개하겠습니다.

역자노트

JSX는 페이스북에서 스펙을 정의한 ECMAScript 친화적인 XML 스러운 문법입니다. React.js에서는 이 문법을 VIRTUAL DOM(또는 컴포넌트의 계층)을 선언적(또는 명시적)으로 서술하여 표현하기 위해 사용됐습니다. 선택적으로 사용할 수 있다고는 하나 JSX가 React.js를 사용하는 이유 중 하나이기 때문에 사용하지 않는 건 개인적으로 추천하지 않습니다.

Flux

React.js에 조금 관심이 있는 분은 React.js와 Flux를 세트로 구성하는 방법에 관해 들은 적 있을 겁니다. 이것은 MVC와 같은 아키텍처를 구성하는 이야기이며 단지 Flux의 구성 요소로서 React.js를 사용하는 방법일 뿐 React.js에 포함된 것은 아닙니다. 이와 관련한 내용도 다음에 자세히 소개

하겠습니다.
다음 절에서는 Hello World를 컴포넌트로 만드는 간단한 예제와 함께 React.js를 사용하는 방법을 소개하겠습니다. 예제 코드를 작성할 때는 아래 공식 jsfiddle 링크를 사용하면 더욱 쉽게 테스트할 수 있으니 참고하시길 바랍니다.

- React JSFiddle
- React JSFiddle without JSX

Hello React.js

이번 절에서는 Hello World를 컴포넌트로 만들어보겠습니다. 기본적으로 React.createClass로 컴포넌트를 만듭니다. 그리고 그 컴포넌트들을 조합해 페이지를 만들고 React.render를 이용해 DOM과 짝을 맞춰 출력합니다.

JSX 사용

JSX에 관해서는 다음 절에서 조금 더 자세히 소개할 예정입니다. 보통 아래와 같은 느낌으로 자바스크립트 내에 XML과 비슷한 마크업을 직접 사용할 수 있습니다.

```
var React = require('react');
var Hello = React.createClass({
  render: function() {
    return (
      <div className="container">Hello {this.props.name}</div>
    );
  }
})
React.render(<Hello name="React" />, document.getElementById("app"));
```

위 코드를 브라우저에서 실행하면 당연히 에러가 발생합니다. 따라서 react-tools를 사용하여 사전에 컴파일하거나 JSXTransformer를 불러와야 합니다. 또한, browserify와 reactify를 조합해 사용하는 변환 방법도 있습니다. 참고로 말씀드리면 div(division)은 흔히 우리가 생각하는 HTML 태그가 아니라 React의 컴포넌트입니다.

역자노트

JSX에서 보이는 div, a 등과 같은 HTML 태그는 사실 HTML 태그가 아니라 모두 React.js의 컴포넌트입니다. 기본 HTML 태그를 React.js에서 미리 컴포넌트로 작성해 제공할 뿐입니다. JSX로 작성되는 모든 요소는 React.js 컴포넌트로 보시면 됩니다.

JSX + ES6, 7의 문법(일부)

JSX의 transform에는 harmoney 옵션이 있습니다. 이 옵션을 켜면 ES6, 7의

문법을 일부 사용할 수 있습니다. ES6의 문법인 Arrow function은 map, filter 등과 조합해 사용하면 정말 편리합니다.

```
var items = this.props.items.map((item) => {
  return <div>{item.name}</div>;
});
```

역자노트

위에서 언급된 react-tools와 JSXTransformer는 곧 Babel로 옮겨집니다. (참고) 자바스크립트 발전 속도에 대응하기 힘들었던 거로 보입니다. 반면, Babel은 잘 대응하고 있고 또 많은 개발자가 기본적으로 채택하는 빌드 도구이기 때문에 결정한 것 같습니다. 따라서 이를 사용하기보단 Babel과 함께 프로젝트를 구성하길 바랍니다.

without JSX

JSX 없이 코드를 작성하면 아래와 같습니다. Hello 컴포넌트의 render 메서드 이외에도 React.render에 Hello 컴포넌트를 전달하는 방식도 바뀌었습니다.

```
var React = require('react');
var Hello = React.createClass({
  render: function() {
    return React.DOM.div({className: 'container', 'Hello ' + this.props.name);
  }
})
React.render(
  React.createFactory(Hello)({name: 'React'}), document.getElementById("app")
);
```

이 Advent Calendar에서 소개하는 코드는 JSX에서 harmony 옵션을 켠 상태에서 작성했음을 알려드립니다. 그럼, 다음 절에서 JSX에 관해 조금 더 넓게 설명하도록 하겠습니다.

React.js의 JSX

위 Hello React.js 절에서 JSX를 잠깐 소개했습니다. 이번에는 조금 더 넓게 살펴보도록 하겠습니다.

JSX

```
var Hello = React.createClass({
  render: function() {
    return (
      <div>Hello {this.props.name}</div>
    );
  }
});
```

위 코드에서 한눈에 HTML로 보이는 부분 ... 이 JSX 문법입니

다. XML과 비슷한 형태로 태그를 작성해 나가면 됩니다. 따로 학습하고 기억해야 할 내용은 거의 없습니다. 이 문법에 관한 자세한 설명은 JSX Specification에 작성돼 있습니다. 하나 주의해야 할 점으로는 JSX는 HTML이 아니므로 div에 container라는 클래스를 지정하고 싶은 경우, ... 가 아니라 ... 로 작성해야 한다는 것입니다. 자바스크립트의 예약어 문제를 회피하기 위해서 이런 문법으로 디자인됐습니다. 추가로 label의 for 속성은 htmlFor로 작성해야 합니다. 이와 관련한 내용은 Tags and Attributes에 정리돼 있습니다. HTML은 태그가 제대로 닫히지 않아도 에러가 발생하지 않지만 JSX는 태그를 닫지 않은 경우 에러가 발생하므로 문법 문제를 쉽게 알아차릴 수 있습니다.

사용법

Realtime로 변환

JSX Transformer를 불러오면 JSX 문법을 실시간으로 변환할 수 있습니다. 다만, 이 방법을 제품(서비스) 환경에서 사용하는 것은 성능면에서 좋지 않아 권장하지 않습니다. 보통 개발 및 디버깅의 편의를 위해 사용합니다.

Precompile로 변환

npm install -g react-tools 커맨드 라인으로 react-tools를 설치하면 jsx 명령을 사용할 수 있습니다.

```
$ jsx src/build/
```

파일을 감시하는 것도 가능합니다.

```
$ jsx --watch src/build/
```

browserify나 webpack으로 변환

browserify와 reactify를 사용하여 변환할 수 있습니다.

```
"browserify": {
  "transform": [
    ["reactify", {"harmony": true} ]
  ]
}
```

node-jsx로 변환

Server-Side rerendering과 같이 ndoe.js 환경에서 변환하고 싶은 경우엔 node-jsx를 사용할 수 있습니다. require하고 install 하는 것으로 간단히 변환할 수 있습니다.

역자노트

Browserify와 babelify(babel)를 조합하면 ES6는 물론 React.js의 JSX 문법을 사용할 수 있습니

다. ECMAScript 6 compatibility table의 core.js + babel 항목을 보면 사용할 수 있는 ES6 문법을 확인할 수 있습니다. 여기에 Grunt를 이용해 자동화하면 조금 더 안락한 개발 환경을 구성할 수 있습니다.

JSX 사용 의미

JSX를 사용하면 HTML 문법과 비슷한 느낌으로 작성할 수 있어 비엔지니어도 이해하기 쉽다는 장점이 있습니다. 개인적인 성향 차이가 있다고는 하지만 개인적으로 React.DOM.div(null, 'hello') 보다 hello 와 같은 방식이 더 좋다고 생각합니다. 또, 버전 0.12는 버전 0.11에 비해 React.createClass의 동작 방식(인터페이스)이 바뀌었지만(이것에 관한 내용은 다음 절에서 소개하겠습니다.) JSX를 사용하고 있는 경우엔 코드를 그대로 사용 가능합니다. 즉, JSX에 대한 지원이 조금 더 좋습니다. JSX를 사용했을 때의 이 점은 이 정도로 생각하고 있으므로 자바스크립트로 작성하고 싶은 사람은 굳이 JSX를 사용하지 않아도 괜찮을 것 같습니다. JSX 이외로 커피스크립트 환경을 고려해 만들어진react-kup도 있습니다.

변환 코드 확인

특정 코드를 변환한 결과를 확인하고 싶은 경우엔 아래 문서를 참고하세요.

- JSX Compiler Service
- HTML to JSX

ES6, 7

harmony 옵션을 켜면 JSX의 변환할 시 Class나 Arrow Function 등 ES6, 7의 기능 일부를 사용할 수 있습니다. 개인적으로 아래와 같이 ES6, 7 문법으로 작성하는 것을 좋아해 옵션을 켜두고 사용하고 있습니다.

```
var Items = React.createClass({
  itemName(item) {
    return `${item.name}:${item.count}`;
  },
  render() {
    var items = this.props.items.map(item => <span>{this.itemName(item)}</span>);
return (
      <div>{items}</div>
    );
  }
});
```

아래와 같은 기능들을 사용할 수 있습니다.(참고)

- es6-arrow-functions
- es6-object-concise-method

- es6-object-short-notation
- es6-classes
- es6-rest-params
- es6-templates
- es6-destructuring
- es7-spread-

React.js를 처음 접하면 JSX라는 불가사의한 언어를 사용할 필요가 있어서 꺼리는 사람도 있을 것 같습니다만 컴포넌트를 알기 쉽게 정의하기 위한 문법이므로 조금 더 가볍게 생각했으면 좋겠습니다. 다음 절에서는 컴포넌트에 관해 조금 더 설명하겠습니다.

React.js의 컴포넌트

이번에는 컴포넌트를 소개하겠습니다. React.js에서는 기본적으로 컴포넌트를 만들고 조합하여 애플리케이션을 만듭니다.

render

컴포넌트는 React.createClass()에 render 메서드를 가진 리터럴 객체를 전달해 작성할 수 있습니다.

```
var Hello = React.createClass({
  render() {
    return (
      <div><span>hello</span></div>
    )
  }
});
```

그러면서 render()는 컴포넌트를 하나만 반환해야 합니다. 아래 처럼 복수의 컴포넌트를 반환할 수 없습니다.

```
// NO
render() {
   return (
     <div>title</div>
     <div>contents</div>
   );
}

// OK
render() {
  return (
    <div>
```

```
      <div>title</div>
      <div>contents</div>
    </div>
  );
}
```

또, render()는 어떤 타이밍에 몇번 호출될지 모르기 때문에 반드시 멱등성을 지키는 방법으로 구현해야합니다.

Separation of concerns?

React.js는 컴포넌트로써 마크업과 뷰의 로직을 createClass()의 안에 작성합니다. 하지만 마크업은 HTML이나 mustache로 작성하고 뷰의 로직은 자바스크립트로 나눠서 작성하는 기존의 방식을 취하지 않아 마음에 들지 않는 사람도 있을 것 같습니다. 이 사안에 대해 React.js의 개발자인 Pete Hunt는 "그것은 관심사의 분리(Separation of concerns)가 아니라 기술의 분리(Speparation of technologies)"라며 마크업과 뷰의 로직은 긴밀해야 한다고 언급했습니다. 거기에 템플릿의 문법으로 불필요하게 코드를 작성하는 것보다 자바스크립트로 작성하는 것이 더 좋다고 말하고 있습니다.

역자노트

HTML, CSS, 자바스크립트를 분리하는 건 관심사의 분리가 아니라 단순한 기술의 분리일 뿐, 그래서 React.js는 관심사를 컴포넌트 단위로 해석했다고 이해할 수 있습니다.

컴포넌트 간의 상호작용

Prop을 I/F로써 외부와 주고 받을 수 있습니다. <Hello name="foo" /> 처럼 작성하면, this.props.name 으로 참조할 수 있습니다.

```
var Hello = React.createClass({
  render() {
    return (
      <div>Hello {this.props.name}</div>
    )
  }
});
// <Hello name="React" />
// <div>Hello React</div>
```

Prop에 관해서는 다음 편에서 소개할 예정입니다.

동적으로 갱신

유저의 액션이나 Ajax 요청 등으로 값이 동적으로 변화하는 경우는 State를 사용합니다. 특정 this.state.xxx을 갱신할 때는 this.state를 사용해 갱신하는 것이 아니라 반드시 this.setState를 사용해 갱신합니다.

```
var Counter = React.createClass({
```

```
  getInitialState() {
    return {
      count: 0
    };
  },
  onClick() {
    this.setState({count: this.state.count + 1});
  },
  render() {
    return (
      <div>
        <div>count:{this.state.count}</div>
        <button onClick={this.onClick}>click!</button>
      </div>
    );
  }
});
```

State에 관한 내용은 다음 편에서 소개할 예정입니다.

React.createClass

React.createClass()는 컴포넌트를 작성할 때 사용하는 함수입니다. 이 함수는 버전 0.12에서 동작 방식이 바뀌었습니다. 0.11에서는 컴포넌트의 정의하고 컴포넌트의 엘리먼트를 반환하는 두 가지의 일을 담당했지만 0.12부터 컴포넌트를 정의하는 작업만 담당하도록 분리됐습니다. 즉, 엘리먼트가 아니므로 사용할 때는 React.createElement(Component, {name: 'xxx'}) 처럼 React Element로 변환할 필요가 있습니다. 이 작업은 React.createFactory(Component)로 해도 같습니다. 다만, JSX를 사용하고 있는 경우는 이전과 똑같이 React.createClass의 반환 값을 로 직접 전달해도 괜찮습니다.

```
var Hello = React.createClass({
  render() {
    return <div>{this.props.name}</div>;
  }
});

React.render(React.createElement(Hello, {name: "foo"}), document.body);
// or
React.render(React.createFactory(Hello)({name: "foo"}), document.body);
// JSX는 이전과 같은 방식
React.render(<Hello name="foo" />, document.body);
```

이 변경은 createClass()라는 이름 외에 또 다른 일을 담당하고 있었다는 문

제를 해결하기도 하지만, createElement를 통해 컴포넌트를 만들도록 함으로써 최적화할 수 있도록 하고 장기적으로 React.createClass로 작성한 문법을 ES6의 class로 대체 할 수 있도록 하려는 뜻도 있습니다.

역자노트

최근에 릴리즈된 버전 0.13에는 ES6의 class 문법을 사용해 컴포넌트를 정의할 수 있게 됐습니다. (참고) ES6 Classes 문법을 이용해 컴포넌트를 작성할 때 몇 가지 주의점이 필요합니다. 이런 사항은 천천히 소개해 드리겠습니다.

정리

여기까지 React.js의 기본적인 특징과 컴포넌트를 명시적으로 서술하기 위한 JSX 문법 등을 알아봤습니다. 컴포넌트는 이 밖에도 Lifecycle을 이용해 hook을 하는 방법도 있습니다. 그 방법에 대해서는 추후 천천히 소개하도록 하고 다음편에서는 Prop와 State 그리고 이 두 속성을 이용해 컴포넌트를 작성하는 방법을 소개하겠습니다.

React.js의 Prop

읽기전에...

이 문서는 koba04님이 작성한 React.js Advent Calendar를 번역한 것입니다. 본래 원문서는 캘린더 형식으로 소개하지만 여기에서는 회를 나눠 작성할 생각입니다. 또한, React 버전 0.12.1 때 작성된 문서이기 때문에 현 버전과 다른 점이 있을 수 있습니다. 최대한 다른 부분을 노트로 작성할 생각이지만, 만약 생략된 부분이 있다면 댓글로 알려주시면 감사하겠습니다.

전편에서 잠깐 등장한 Props을 소개하겠습니다.

기본 사용법

Prop은 컴포넌트의 속성(어트리뷰트)으로 정의하고 컴포넌트 내에서는 this.props.xxx로 참조해 사용합니다. 이것이 전부입니다. Prop으로는 객체, 함수 등 어떤 타입이든 지정할 수 있습니다.

```
var Avatar = React.createClass({
  render() {
    var avatarImg = `/img/avatar_${this.props.user.id}.png`;
return(
      <div>
        <span>{this.props.user.name}</span>
        <img src={avatarImg} />
      </div>
    );
  };
});
var user = {
```

```
  id: 10,
  name: "Hoge"
};
// <Avatar user={user} />
```

I/F(인터페이스)로써의 Prop

Prop은 외부에서 전달한 값이지 그 컴포넌트가 자체적으로 관리하는 값이 아니므로 내부에서 변경하면 안 됩니다. 컴포넌트가 관리할 필요가 있는 값은 다음 절에서 소개할 State로 정의해야 합니다. 즉, Prop은 Immutable(불변) 하며 외부와 I/F로써 작용합니다.

PropTypes

컴포넌트의 Prop은 외부로부터 값을 지정받기 때문에 검증(벨리데이션)이 필요합니다. 이때 React.js에서는 PropsTypes으로 Prop에 대한 타입 제약을 지정할 수 있습니다. 화려하진 않지만 좋은 기능입니다.

```
var Avatar = React.createClass({
  propTypes: {
    name:    React.PropTypes.string.isRequired,
    id:      React.PropTypes.number.isRequired,
    width:   React.PropTypes.number.isRequired,
    height: React.PropTypes.number.isRequired,
    alt:     React.PropTypes.string
  },
  render() {
    var src = `/img/avatar/${this.props.id}.png`;
    return (
      <div>
        <img src={src} width={this.props.width} height={this.props.height} alt={this.props.alt} />
        <span>{this.props.name}</span>
      </div>
    );
  }
});
<Avatar name="foo" id=1 width=100 height=100 />
```

위와 같은 느낌으로 작성합니다. PropTypes을 지정하는 것으로 컴포넌트의 I/F를 조금 더 명확하게 표현할 수 있습니다. PropTypes의 지정은 아래와 같은 느낌으로 유연하게 지정할 수 있습니다.

```
React.PropTypes.array            // 배열
React.PropTypes.bool.isRequired // Boolean, 필수
React.PropTypes.func             // 함수
React.PropTypes.number           // 정수
```

```
React.PropTypes.object          // 객체
React.PropTypes.string          // 문자열
React.PropTypes.node            // Render가 가능한 객체
React.PropTypes.element         // React Element
React.PropTypes.instanceOf(XXX) // XXX의 instance
React.PropTypes.oneOf(['foo', 'bar']) // foo 또는 bar
React.PropTypes.oneOfType([React.PropTypes.string, React.PropTypes.array]) // 문자열 또는 배열
React.PropTypes.arrayOf(React.PropTypes.string)  // 문자열을 원소로 가지는 배열
React.PropTypes.objectOf(React.PropTypes.string) // 문자열을 값으로 가지는 객체
React.PropTypes.shape({                          // 지정된 형식을 충족하는지
  color: React.PropTypes.string,
  fontSize: React.PropTypes.number
});
React.PropTypes.any.isRequired  // 어떤 타입이든 가능하지만 필수
// 커스텀 제약도 정의 가능
customPropType: function(props, propName, componentName) {
  if (!/^[0-9]/.test(props[propName])) {
    return new Error('Validation failed!');
  }
}
```

아래와 같이 제일 처음 소개한 예제 코드에 PropTypes를 정의할 수 있습니다.

```
var Avatar = React.createClass({
  propTypes: {
    user: React.PropTypes.shape({
      id:   React.PropTypes.number.isRequired,
      name: React.PropTypes.string.isRequired
    })
  },
  render() {
    var avatarImg = `/img/avatar_${this.props.user.id}.png`;
    return(
      <div>
        <span>{this.props.user.name}</span>
        <img src={avatarImg} />
      </div>
    );
  }
});
```

주의점으로는 React.js의 제약은 성능적인 이유로 실 서비스 환경에서는 검증하지 않습니다. 또 개발 환경에서도 에러가 발생하는 것이 아닌 console.warm으로 출력됩니다. 에러가 발생하도록 변경해 달라는 issue도

등록됐었기 때문에 앞으로 어떻게 변경될진 모르겠습니다.

역자노트

ES6에서 PropTypes을 지정하는 방식은 다음과 같습니다.

```
class Avatar extends React.Component {
  render() {
    var avatarImg = `/img/avatar_${this.props.user.id}.png`;
return(
      <div>
        <span>{this.props.user.name}</span>
        <img src={avatarImg} />
      </div>
    );
  }
}
Avatar.propTypes =  {
  user: React.PropTypes.shape({
    id:   React.PropTypes.number.isRequired,
    name: React.PropTypes.string.isRequired
  })
};
export default Avatar;
```

기본값 지정

getDefaultProps()에서 리터럴 객체를 반환하면 기본값으로 지정됩니다. 이는 컴포넌트 인스턴스가 만들어질 때 호출되는 것이 아니라 컴포넌트가 정의될 때만 호출되므로 주의가 필요합니다. 다음 절에서 소개할 getInitialState()은 다릅니다.

```
var Hello = React.createClass({
  getDefaultProps() {
    return {
      name: 'React'
    };
  },
  render() {
    return <div>Hello {this.props.name}</div>
  }
});
// <Hello />
```

역자노트

ES6에서 기본 값을 지정하는 방식은 다음과 같습니다.

```
class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>
```

```
  }
}
Hello.defaultProps = {
  name: 'React'
};
export default Hello;
// <Hello />
```

setProps & replaceProps

컴포넌트에 새로운 Prop을 전달하고 다시 rerender 하고 싶은 경우엔 setProps()와 replaceProps()를 사용합니다. 이 메서드를 이용하면 Prop을 갱신하면서 rerender 할 수 있습니다.

```
var Test = React.createClass({
  getDefaultProps: function() {
    return {
      id: 1
    };
  },
  render: function() {
    return (
      <div>{this.props.id}:{this.props.name}</div>
    );
  }
});
var component = React.render(<Test name="bar" />, document.body);
component.setProps({ name: "foo" });       // <div>1:foo</div>
component.replaceProps({ name: "hoge" }); // <div>:hoge</div>
```

setProps()은 기존의 Prop과 새로운 Prop을 합치(merge)지만 replaceProps()는 대체합니다. 그리고 각각 두 번째 인수에 콜백 함수를 지정할 수 있습니다.

역자노트

replaceProps()는 ES6에서 사용할 수 없으며, 곧 제거될 예정입니다.

여기까지 React.js의 Prop을 살펴봤습니다. 다음 절에서는 State를 소개하겠습니다.

React.js의 State

Porp은 Immutable하지만 State는 Mutable(이변)한 값을 정의할 수 있습니다.

기본 사용법

getInitialState()을 이용해 state의 초기값을 반환하고 데이터 변경이 있는

경우 this.setState()로 갱신합니다. 상태가 갱신되면 컴포넌트가 rerender 되어 UI가 갱신됩니다. 이때, 자식 컴포넌트도 함께 rerender 됩니다.

```
var Counter = React.createClass({
  getInitialState() {
    return {
      count: 0
    };
  },
  onClick() {
    this.setState({ count: this.state.count + 1});
  },
  render() {
    return (
      <div>
        <span>{this.state.count}</span>
        <button onClick={this.onClick}>click</button>
      </div>
    );
  }
});
```

setState()의 두 번째에 인수에는 setProps() 처럼 콜백 함수를 지정할 수 있습니다. 또 replaceProps()와 비슷한 replaceState()도 있습니다.

역자노트

replaceState()는 ES6에서 사용할 수 없으며, 곧 제거될 예정입니다.

State를 사용한 UI

State는 텍스트 필드 같은 컴포넌트 내에서 사용자 인터렉션에 따라 변경되는 값을 관리하는 경우에 가장 자주 사용됩니다. 또 컴포넌트 내에서 Ajax로 데이터를 요청하고 성공 시 콜백 함수에서 응답 데이터를 setState() 하는 방식으로도 사용합니다.

주의할 점

state의 값을 프로퍼티로 접근해 직접 변경하면 안 되고 반드시 setState()를 사용해 갱신해야 합니다. 이는 setState()가 호출되어야 rerender 되기 때문입니다. this.state 값 자체도 Immutable 하다라고 생각하는 것이 좋습니다. 만약, this.state.list라는 배열이 있고 list에 요소를 추가하고 싶은 경우도 push()하고 setState()하는 것이 아니라 this.setState({list: this.state.list.concat([value])}로 새로운 값(배열)을 지정하는 것이 좋습니다. 이 방법이 shouldComponentUpdate()로 성능 최적화 할 때와 undo의 구

현 시에 좀 더 유용합니다.

State는 최소화

Prop만 가지고 있는 Immutable한 컴포넌트가 조작하거나 이해하기 쉬우므로, 기본적으로는 Prop을 고려하고, State를 가진 컴포넌트는 최소화 하는 게 좋습니다. 최상위 컴포넌트만 State를 갖게 하고, 하위 컴포넌트는 전부 Prop만을 가지는 Immutable한 컴포넌트로 구성하여 어떤 변경이 있을 때 최상위 컴포넌트에서 setState()하여 rerender 하는 설계도 가능합니다. 이는 VirtualDOM의 기술을 이용한 설계 방법입니다. 이와 관련된 내용은 다음에 소개하겠습니다.

여기까지 State를 소개했습니다. 다음으로 Prop과 State를 사용한 컴포넌트 간의 상호작용을 소개하겠습니다.

Prop과 State를 사용한 컴포넌트 상호작용

이번에는 지금까지 소개한 Prop과 State를 사용해 컴포넌트 간 상호작용 하는 방법에 대해서 작성하겠습니다.

부모의 State를 자식의 Prop으로 전달

컴포넌트 설계 시 인터페이스를 고려해서 Prop을 설계하고 그 컴포넌트가 관리할 값 중 변경되는 값을 추려 State로 정의합니다. 컴포넌트 간의 부모와 자식 관계를 의식해서 설계해야 합니다. 부모는 State를 갖고 있고, 자식의 Prop으로 값을 전달하는 것이 기본 흐름입니다. 자식은 값을 사용하기만 할 뿐 관리는 부모가 합니다.

```
var User = React.createClass({
  propTypes: {
    name: React.PropTypes.string.isRequired,
    id:   React.PropTypes.number.isRequired
  },
  render() {
    return (
      <div>{this.props.id}:{this.props.name}</div>
    );
  }
});

var request = require('superagent');

var Users = React.createClass({
  getInitialState() {
```

```
    return {
      users: [ {id: 1, name: "foo"}, {id: 2, name: "bar"} ]
    }
  },
  componentDidMount() {
    request.get('http://example.com/users/', (res) => {
      this.setState({users: res.body.users});
    });
  },
  render() {
    var users = this.state.users.map((user) => {
      return <User id={user.id} name={user.name} key={user.id}/>
    });
    return (
      <div>
        <p>사용자 목록</p>
        {users}
      </div>
    );
  }
});
```

자식의 이벤트를 부모에서 처리

자식 컴포넌트 내에서 발생하는 이벤트를 부모에서 처리하고 싶은 경우엔 자식이 이벤트를 처리하기 위한 함수를 Prop 즉, I/F로 공개하고 부모가 자식의 Prop을 이용해 리스너를 전달하는 형태로 처리합니다. 예를 들어 TodoList에서 각 Todo는 자식 컴포넌트가 되고 자식 컴포넌트에 삭제나 편집 기능이 있을 때 삭제와 편집 처리 로직은 부모 컴포넌트에 정의하고 이벤트는 자식 컴포넌트에서 버블링되는 느낌으로 동작합니다.

```
var Todo = React.createClass({
  propTypes: {
    todo: React.PropTypes.shape({
      id: React.PropTypes.number.isRequired,
      text: React.PropTypes.string.isRequired
    }),
    // 삭제 처리를 I/F로 정의
    onDelete: React.PropTypes.func.isRequired
  },
  // 부모에게 이벤트 처리를 위임한다.
  _onDelete() {
    this.props.onDelete(this.props.todo.id);
  },
```

```
  render() {
    return (
      <div>
        <span>{this.props.todo.text}</span>
        <button onClick={this._onDelete}>delete</button>
      </div>
    );
  }
});

var TodoList = React.createClass({
  getInitialState() {
    return {
      todos: [
        {id:1, text:"advent calendar1"},
        {id:2, text:"advent calendar2"},
        {id:3, text:"advent calendar3"}
      ]
    };
  },
  // TodoList는 이 컴포넌트가 관리하고 있으므로 삭제 처리도 여기에 존재한다.
  deleteTodo(id) {
    this.setState({
      todos: this.state.todos.filter((todo) => {
        return todo.id !== id;
      })
    });
  },
  render() {
    var todos = this.state.todos.map((todo) => {
      return <li key={todo.id}><Todo onDelete={this.deleteTodo} todo={todo} /></li>;
    });
    return <ul>{todos}</ul>;
  }
});

React.render(<TodoList />, document.body);
```

State 초기값을 Prop에서 전달

State의 초기값을 Prop에서 전달해야 하는 경우엔 아래와 같이 처리합니다.

```
var Counter = React.createClass({
  propTypes: {
    count: React.PropTypes.number
```

```
  },
  getDefaultProps() {
    return {
      count: 0
    };
  },
  getInitialState() {
    return {
      count: this.props.count
    }
  },
  onClick() {
    this.setState({ count: this.state.count + 1 });
  },
  render() {
    return (
      <div>
        <p>{this.state.count}</p>
        <button onClick={this.onClick}>click</button>
      </div>
    );
  }
});

// <Counter count=10 />
```

하지만 위와 같은 형태로 작성하면 값이 증가할 때마다 Prop의 count도 함께 증가할 것으로 보이기 때문에 Prop을 초기값으로 사용할 때는 의도를 명확하게 드러내는 이름으로 작성합니다.

```
var Counter = React.createClass({
  propTypes: {
    initialCount: React.PropTypes.number
  },
  getDefaultProps() {
    return {
      initialCount: 0
    };
  },
  getInitialState() {
    return {
      count: this.props.initialCount
    }
  },
  onClick() {
```

```
    this.setState({ count: this.state.count + 1 });
  },
  render() {
    return (
      <div>
        <p>{this.state.count}</p>
        <button onClick={this.onClick}>click</button>
      </div>
    );
  }
  :
});

// <Counter initialCount=10 />
```

ref

컴포넌트 내에서 ref 프로퍼티를 사용하여 자식 컴포넌트를 참조할 수 있습니다. 이 프로퍼티를 사용하면 부모에서 자식의 메서드를 호출할 수 있습니다. 하지만 한번 사용하기 시작하면 컴포넌트 간의 관계를 알기 어려워지므로 기본적으로 div나 button 등과 같은 내장 컴포넌트를 참조할 때만 사용하는 게 좋습니다. 보통 다음 절에서 설명할 getDOMNode()와 함께 사용하는 경우가 많습니다.

```
var Test = React.createClass({
  componentDidMount() {
    console.log(this.refs.myDiv.props.children);   // xxx
  },
  render() {
    return (
      <div ref="myDiv">xxx</div>
    );
  }
});
```

역자노트

React v0.14 베타 버전부터 refs는 VIRTUAL DOM이 아닌 진짜 DOM 객체를 참조하도록 변경 됐습니다. (참고), 따라서 getDOMNode()를 조합해 사용할 필요가 없습니다.

getDOMNode

React.js에서 DOM은 VirtualDOM에 감춰져 있어서 직접 DOM을 조작하지 않습니다. 하지만 focus 하거나, jQuery Plugin을 쓰고자 할 때는 직접 DOM을 조작해야 하는 경우도 있습니다. 그런 경우에는 ref와 함께 getDOMNode()를 사용하여 DOM을 참조합니다. 다만, DOM을 직접 수정

하게 되면 VirtualDOM과의 관계가 틀어지기 때문에 읽기 전용으로 사용해야 합니다.

```
var Focus = React.createClass({
  componentDidMount() {
    this.refs.myText.getDOMNode().focus();
  },
  render() {
    return (
      <div>
        <p>set focus</p>
        <input type="text" ref="myText" />
      </div>
    );
  }
});
```

역자노트

getDOMNode()는 deprecated 됩니다.(참고) 대신 다음과 같이 사용하세요.

```
componentDidMount() {
  React.findDOMNode(this.refs.myText).focus();
}
```

Props.children

<myComponent>xxx</myComponent>와 같이 작성할때 xxx를 얻고자 할 때는 this.props.children 프로퍼티를 사용합니다.

```
var Hello = React.createClass({
  render() {
    return <div>{this.props.children}</div>;
  }
});

console.log(
  React.render(
    <Hello>xxx</Hello>,
    document.body
  ).props.children
);
// => xxx

console.log(
  React.render(
     <Hello><span>1</span><span>2</span></Hello>,
     document.body
  ).props.children
```

```
);
// => [React.Element, React.Element]
console.log(
  React.render(
    <Hello></Hello>,
    document.body
  ).props.children
);
// undefined
```

위와 같이 props.children은 지정 방식에 따라 문자열이거나 원소가 React Element로 이뤄진 배열이거나 undefined 일 수도 있습니다. 그래서 배열이라는 가정에 따라 원소의 개수를 확인하기 위해 children.length 한 경우 만약 문자열이 전달되면 String.length의 값이 반환되므로 chdilren을 사용할 때는 어떤 타입인지 검사할 필요가 있습니다. React.Children에는 count, forEach, map, only 등 편리한 함수를 제공하고 있습니다. 이 메서드를 잘 사용하면 자식을 조작할 때 발생하는 문제를 잘 회피할 수 있습니다.

```
var Hello = React.createClass({
  render() {
    return <div>{this.props.children}</div>;
  }
});
[
  <Hello>xxx</Hello>,
  <Hello><span>1</span><span>2</span></Hello>,
  <Hello></Hello>
].forEach( jsx => {
  var children = React.render(jsx, document.body).props.children;
  console.log("#########" + children + "##########");
  console.log(React.Children.count(children));
  React.Children.forEach(children, (child) => { console.log(child) });
});
// #########xxx##########
// 1
// xxx
// #########[object Object],[object Object]##########
// 2
// ReactElement {type: "span", key: null, ref: null, _owner: null, _context: Object...}
// ReactElement {type: "span", key: null, ref: null, _owner: null, _context: Object...}
// #########undefined##########
```

위 예제를 보면 알 수 있듯이 React.Children의 메서드를 사용하여 배열과 문자열의 문제를 해결하고 있습니다. 참고로 React.Children.only는

children의 React.element가 하나 이상일 때 오류를 발생시키는 함수입니다.

정리

여기까지 Prop과 State를 알아보고 그 속성을 사용해 컴포넌트에서 상호작용하는 방법도 알아봤습니다. prop과 state는 컴포넌트에서 데이터와 상태를 관리하는 데 중요한 속성이므로 꼭 기억해두시길 바랍니다. 다음편에서는 React 컴포넌트 작성 시 유용하게 사용할 수 있는 Lifecycle와 이벤트 등을 소개하겠습니다.

Component Lifecycle

읽기전에...

이 문서는 koba04님이 작성한 React.js Advent Calendar를 번역한 것입니다. 본래 원문서는 캘린더 형식으로 소개하지만 여기에서는 회를 나눠 작성할 생각입니다. 또한, React 버전 0.12.1 때 작성된 문서이기 때문에 현 버전과 다른 점이 있을 수 있습니다. 최대한 다른 부분을 노트로 작성할 생각이지만, 만약 생략된 부분이 있다면 댓글로 알려주시면 감사하겠습니다.

이번에는 컴포넌트의 라이프사이클(Lifecycle)을 소개하겠습니다. React.js는 컴포넌트의 상태 변화에 맞춰 호출되는 여러 가지 메서드를 제공 합니다. 그 메서드를 사용해 초기화나 후처리 등을 할 수 있습니다. 자주 사용하는 메서드는 componenetDidMount()나 componentWillUnmount() 입니다. componentDidMount()에서 이벤트를 등록하고 componentWillUnmount()에서 이벤트를 해제하는 패턴을 많이 사용합니다.

componentWillMount()

컴포넌트가 DOM 트리에 추가되기 전 한 번만 호출됩니다. 초기화 처리를 하는 데 사용할 수 있습니다. 이 안에서 setState하면 render 시에 사용됩니다.**Server-side rendering 시에도 호출되므로 어느 쪽에서도 동작할 수 있는 코드를 작성해야 합니다.**

역자노트

Server-side rendering 시에도 호출 되므로 대도록 이 Lifecycle 메서드에서 DOM을 컨트롤 하는 브라우저에서만 동작하는 로직을 작성하면 안됩니다. Node.js 환경에서는 DOM이 없으므로 에러가 발생하게 됩니다.

componentDidMount()

컴포넌트가 DOM 트리에 추가된 상태에 호출됩니다. DOM과 관련된 초기화를 하고 싶을 때 편리하게 사용할 수 있습니다. componentWillMount()와 다른 게**Server-side rendering 시에 호출되지 않습니다.** 따라서 DOM

을 다루는 처리 외에, Ajax 요청이나 setInterval 등의 Server-side rendering 시에는 불필요한 초기화 처리는 이 메서드를 통해 진행합니다.

componentWillReceiveProps(nextProps)

Prop이 갱신될 때 호출됩니다. 컴포넌트가 새로운 DOM 트리에 추가될 때는 호출되지 않습니다. 부모 컴포넌트의 State가 Prop으로 전달되고, 그 값이 변화한 할 때 화면의 표시 이외 Notification 같은 추가 작업을 이 메서드를 통해 할 수 있습니다. 마지막으로 Prop의 값에 따라 State의 값을 갱신 할 때에도 사용합니다.

shouldComponentUpdate()

이 메서드는 다른 메서드 Lifecycle 메서드와 달리 true나 false를 반환할 필요가 있습니다. 컴포넌트가 rerender 하기 전에 호출되며, 만약 false를 반환하면 VirtualDOM 비교를 하지 않고 rerender도 하지 않습니다. 즉, 독자적으로 Prop이나 State 비교 처리를 구현하는 것으로 불필요한 계산을 하지 않을 수 있습니다. 보통 성능 향상을 목적으로 사용합니다. 이 메서드가 반환하는 기본값은 true 이므로 재정의 하지 않으면 항상 rerender 합니다. 강제적으로 rerender 하고자 할땐 forceUpdate()를 사용합니다.
forceUpdate()가 호출되는 경우엔 shouldComponentUpdate()는 호출되지 않습니다.
Porp과 State가 Immutable한 데이터라면 다음과 같이 단순한 객체 비교로 구현이 가능합니다.

```
shouldComponentUpdate: function(nextProps, nextState) {
  return nextProps.user !== this.props.user || nextState.user !== this.state.user;
}
```

componentWillUpdate(nextProps, nextState)

컴포넌트가 갱신되기 전에 호출됩니다. 최초엔 호출되지 않습니다. 이 안에서는 setState를 호출할 수 없으므로 Prop의 값을 이용해 setState 하고 싶은 경우엔 componentWillReceiveProps()를 사용합니다.

componentDidUpdate(prevProps, prevState)

컴포넌트가 갱신된 뒤에 호출됩니다. 최초엔 호출되지 않습니다. DOM의 변화에 hook 하여 또 다른 작업을 하고 싶을 때 사용할 수 있습니다.

componentWillUnmount()

컴포넌트가 DOM에서 삭제될 때 호출됩니다. 이벤트 해제 같은 clean-up

처리 시 할 때 사용합니다. ComponentDidMount()에서 등록한 Timer의 처리나 DOM의 이벤트 등은 여기에서 해제해야 합니다.

추가

isMounted()

개발 시 Ajax를 요청하고 그 결과를 setState 하는 패턴이 자주 발생합니다. 그때 Ajax의 응답이 왔을 때 컴포넌트가 이미 Unmount 된 경우가 있는데, 바로 setState나 forceUpdate를 호출하면 에러가 발생하게 됩니다. 따라서 isMounted()를 사용해 방어 코드를 작성할 필요가 있습니다.

```
componentDidMount() {
  request.get('/path/to/api', res => {
    if (this.isMounted()) {
      this.setState({data: res.body.data});
    }
  });
}
```

역자노트

isMounted()는 React.Component를 상속한 ES6 클래스에서 사용할 수 없으며 미래의 버전에서 삭제될 예정입니다.

여기까지 컴포넌트의 Lifecycle를 소개했습니다. 다음절에서는 이벤트를 소개하겠습니다.

React.js의 이벤트

이번 절에서는 DOM 이벤트 처리를 소개하겠습니다.

SyntheticEvent

React.js에서는 DOM을 VIRTUAL DOM으로 랩핑한 것처럼 DOM의 이벤트 객체도 SyntheticEvent이라는 객체로 랩핑하여 크로스 브라우저에 대응하고 있습니다. SyntheticEvent의 인터페이스는 아래와 같습니다.

- **boolean** bubbles
- **boolean** cancelable
- **DOMEventTarget** currentTarget
- **boolean** defaultPrevented
- **Number** eventPhase
- **boolean** isTrusted
- **DOMEvent** nativeEvent
- **void** preventDefault()

- **void** stopPropagation()
- **DOMEventTarget** target
- **Date** timeStamp
- **String** type

이처럼 preventDefault()나 stopPropagation() 그리고 target 등을 지금까지 다뤘던 방식으로 사용할 수 있습니다. 추가로 이벤트 리스너에서 false를 반환하는 방법으로 이벤트의 전파를 정지할 수 있었지만, 이 방법은 이해하기 어렵다는 이유로 React.js 버전 0.12에서는 사용할 수 없도록 변경됐습니다.

이벤트 핸들러

기본적인 이벤트는 모두 지원하고 있습니다. 예를 들어 click 이벤트를 처리하고 싶은 경우엔 아래와 같이 작성합니다.

```
var Counter = React.createClass({
  getInitialState() {
    return {
      count: 0
    };
  },
  onClick(e) {
    // e is SyntheticEvent
    this.setState({ count: this.state.count + 1 });
  },
  render() {
    return (
      <div>
        <span>click count is {this.state.count}</span>
        <button onClick={this.onClick}>click!</button>
      </div>
    );
  }
});
```

onClick={this.onClick}으로 클릭 이벤트를 받고 있습니다. 이때 React.js는 컴포넌트의 문맥을 리스너에 bind 해주므로 따로 this.onClick.bind(this)와 같은 별도의 바인딩 작업이 필요하지 않습니다. 따라서 리스너 내에서 바로 this.setState()와 같은 메서드를 사용할 수 있습니다. 참고로 자동으로 this를 바인딩하는 동작은 앞으로 ES6의 ArrowFunction를 사용하도록 권고하고 지원하지 않을 수 있습니다.

역자노트

객체 리터럴로 컴포넌트를 생성할때는 실행 문맥 바인드가 필요 없지만 ES6 Classes 문법으로 작성할 땐 필요합니다.(참고)

Event delegation

Event Delegation은 jQuery에도 널리 알려진 대중적인 개념입니다. React.js는 자동으로 최상위 요소에만 이벤트를 등록하고 그곳에서 이벤트를 취합하여 내부에서 관리하는 맵핑 정보를 바탕으로 대응하는 컴포넌트에 이벤트를 발행합니다. 이때 이벤트는 캡처링, 버블링 되는데, 각 리스너마다 SyntheticEvent의 객체가 만들어지기 때문에 메모리의 얼로케이트를 여러 번 할 필요가 있습니다. 이 문제를 해결하기 위해 React.js는 객체를 풀(pool)로 관리하고 재사용하여 가비지 컬렉터의 횟수를 줄일 수 있도록 구현돼 있습니다. 추가로 DOM에 설정된 data-reactid을 사용해서 맵핑하고 있는 것 같습니다. 그리고 id로 부모와 자식 관계를 알 수 있도록 디자인돼 있습니다.(참고)

```
<ul class="nav nav-pills nav-justified" data-reactid=".1px6jd5i1a8.1.0.0.0.1.0">
  <li class="" data-reactid=".1px6jd5i1a8.1.0.0.0.1.0.0">
    <a href="/artist" data-reactid=".1px6jd5i1a8.1.0.0.0.1.0.0.0">Artist</a>
  </li>
  <li class="" data-reactid=".1px6jd5i1a8.1.0.0.0.1.0.1">
    <a href="/country" data-reactid=".1px6jd5i1a8.1.0.0.0.1.0.1.0">Country</a>
  </li>
</ul>
```

Not provided event

React.js가 지원하는 기본적인 이벤트 외에 window의 resize 이벤트나 jQuery Plugin의 독자 포멧 이벤트를 사용하고 싶은 경우 componentDidMount()에서 addEventListener를 통해 이벤트를 등록하고 componentWillUnmount()를 이용해 removeEventListener 하여 이벤트를 해제해 사용합니다.(참고) 참고로 이 경우 역시 this를 자동으로 bind 합니다.

```
var Box = React.createClass({
  getInitialState() {
    return {
      windowWidth: window.innerWidth
    };
  },
  handleResize(e) {
    this.setState({windowWidth: window.innerWidth});
```

```
  },
  componentDidMount() {
    window.addEventListener('resize', this.handleResize);
  },
  componentWillUnmount() {
    window.removeEventListener('resize', this.handleResize);
  },
  render() {
    return <div>Current window width: {this.state.windowWidth}</div>;
  }
});
React.render(<Box />, mountNode);
```

글로벌 이벤트를 선언하는 방법에 여러 논의가 있었습니다. 이슈285을 참고하세요.

touch event

터치 이벤트는 기본적으로 비활성화 돼 있습니다. 활성화하고 싶은 경우엔 React.initializeTouchEvents(true) 를 호출합니다.

여기까지 Event를 정리했습니다. 다음으로 Form을 다루는 방법을 소개하겠습니다.

React.js에서 폼 다루기

이번 절에서는 React.js에서 폼을 다루는 방법을 소개하겠습니다. React.js에서는 아래와 같이 Input 폼을 작성하면 변경할 수 없는 텍스트 필드가 생성됩니다.(데모)

```
<input type="text" value="initial value" />
<input type="text" value={this.state.textValue} />
```

Controlled Component

Controlled Component는 State에 따라 값을 관리하는 Componenet 입니다. 이를 이용해 텍스트 필드를 재작성합니다.

```
var Text = React.createClass({
  getInitialState() {
    return {
      textValue: "initial value"
    };
  },
  changeText(e) {
    this.setState({textValue: e.target.value});
  },
  render() {
```

```
    return (
      <div>
        <p>{this.state.textValue}</p>
        <input type="text" value={this.state.textValue} onChange={this.changeText} />
      </div>
    );
  }
});
```

value를 State로 관리하고, onChange()에서 setState()하여 명시적으로 값을 갱신하고 전달합니다.

UnControlled Component

UnControlled Componenent는 반대로 값을 관리하지 않는 컴포넌트로 초기값을 설정한 값은 defaultValue로 지정합니다. 이 경우는 앞 절에서처럼 onChange()에서 항상 값을 state에 반영해도 되고, 반영하고 싶을 때만 DOM에서 value를 취득하여 갱신하는 것도 가능합니다.

```
var LiveText = React.createClass({
  getInitialState() {
    return {
      textValue: "initial value"
    };
  },
  changeText(e) {
    this.setState({textValue: this.refs.inputText.getDOMNode().value });
  },
  render() {
    return (
      <div>
        <p>{this.state.textValue}</p>
        <input type="text" ref="inputText" defalutValue="initial value" />
        <button onClick={this.changeText}>change</button>
      </div>
    );
  }
});
```

textarea

textarea의 경우도 텍스트 필드와 마찬가지로 value를 지정합니다. HTML처럼 <textarea>xxx</textarea> 으로 작성하면 xxx는 defaultValue로 취급됩니다.(데모)

```
var OreTextArea = React.createClass({
  getInitialState() {
```

```
    return {
      textAreaValue: "initial value"
    };
  },
  onChangeText(e) {
    this.setState({textAreaValue: e.target.value});
  },
  onClick() {
    this.setState({textAreaValue: this.refs.textArea.getDOMNode().value});
  },
  render() {
    return (
      <div>
        <div>{this.state.textAreaValue}</div>
        <div>
          <textarea value={this.state.textAreaValue} onChange={this.onChangeText} />
        </div>
        <div>
          <textarea ref="textArea">this is default value</textarea>
          <button onClick={this.onClick}>change</button>
        </div>
      </div>
    );
  }
});
```

셀렉트 박스

셀렉트 박스도 역시 value를 지정합니다. multiple={true} Prop을 지정하면 요소를 복수로 선택할 수 있습니다.(데모)

```
var OreSelectBox = React.createClass({
  getDefaultProps() {
    return {
      answers: [1, 10, 100, 1000]
    };
  },
  getInitialState() {
    return {
      selectValue: 1,
      selectValues: [1,100]
    };
  },
  onChangeSelectValue(e) {
    this.setState({selectValue: e.target.value});
  },
```

```
  // 더 좋은 방법이 있을지...
  onChangeSelectValues(e) {
    var values = _.chain(e.target.options)
      .filter(function(option) { return option.selected })
      .map(function(option) { return +option.value })
      .value()
    ;
    this.setState({selectValues: values});
  },
  render() {
    var options = this.props.answers.map(function(answer) {
      return <option value={answer} key={answer}>{answer}</option>;
    });
    return (
      <div>
        <div>selectValue: {this.state.selectValue}</div>
        <div>
          <select value={this.state.selectValue} onChange={this.onChangeSelectValue}>
            {options}
          </select>
        </div>
        <div>selectValues: {this.state.selectValues.join(",")}</div>
        <div>
          <select multiple={true} defaultValue={this.state.selectValues}
onChange={this.onChangeSelectValues}>
            {options}
          </select>
        </div>
      </div>
    );
  }
});
```

LinkedStateMixin

LinkedStateMixin이라는 addon을 사용하면 앞에서 처럼 onChange()를 일일이 구현하지 않아도 state에 반영할 수 있습니다. 체크박스에 사용할 때는 checkLink를 사용합니다.

```
var React = require('react/addons');
var LinkedStateMixin = React.createClass({
  mixins: [React.addons.LinkedStateMixin],
  getInitialState() {
    return {
      textValue: "initial value"
```

```
    }
  },
  render() {
    return (
      <div>
        <div>value: {this.state.textValue}</div>
        <input type="text" valueLink={this.linkState('textValue')} />
      </div>
    );
  }
});
```

이 mixin이 하고 있는 것은 간단합니다. 내부 로직을 한번 살펴보는 것도 재미있을 것 같습니다.

LinkedStateMixin의 동작 방식

우선 Mixin해서 사용하는 linkState의 내부 로직을 보면 value와 무엇인가 작성한 Setter를 전달해서 ReactLink 객체의 인스턴스를 생성해 반환하고 있습니다.(참고)

```
linkState: function(key) {
  return new ReactLink(
    this.state[key],     ReactStateSetters.createStateKeySetter(this, key)
  );
}
```

ReactStateSetters.createStateKeySetter의 내부를 보면 전달된 State의 키에 대응해서 setState를 하는 함수를 반환하고 있습니다.(참고)

```
  createStateKeySetter: function(component, key) {
    // Memoize the setters.
    var cache = component.__keySetters || (component.__keySetters = {});
    return cache[key] || (cache[key] = createStateKeySetter(component, key));
  }
};
function createStateKeySetter(component, key) {
  // Partial state is allocated outside of the function closure so it can be
  // reused with every call, avoiding memory allocation when this function
  // is called.
  var partialState = {};
  return function stateKeySetter(value) {
    partialState[key] = value;
    component.setState(partialState);
  };
}
```

ReactLink의 Constructor(생성자)에서는 값(value)과

requestChange(createStateKeySetter에서 반환한 함수)를 프로퍼티로 설정합니다.(참고)

```
function ReactLink(value, requestChange) {
  this.value = value;
  this.requestChange = requestChange;
}
```

여기에서, valueLink의 Prop을 살펴보면 requestChange에 전달하는 인자는 e.target.value 라는 사실을 알 수 있습니다.(참고)

```
function _handleLinkedValueChange(e) {
  /*jshint validthis:true */
  this.props.valueLink.requestChange(e.target.value);
}
/**
  * @param {SyntheticEvent} e change event to handle
  */
function _handleLinkedCheckChange(e) {
  /*jshint validthis:true */
  this.props.checkedLink.requestChange(e.target.checked);
}
```

input의 컴포넌트를 보면, onChange 이벤트에 valueLink가 있으면 _handleLinkedValueChange를 호출하여 그 결과, setState 한다는 것을 알 수 있습니다.(참고1, 참고2)

```
getOnChange: function(input) {
  if (input.props.valueLink) {
    _assertValueLink(input);
    return _handleLinkedValueChange;
  } else if (input.props.checkedLink) {
    _assertCheckedLink(input);
    return _handleLinkedCheckChange;
  }
  return input.props.onChange;
}
_handleChange: function(event) {
  var returnValue;
  var onChange = LinkedValueUtils.getOnChange(this);
  if (onChange) {
    returnValue = onChange.call(this, event);
  }
```

여기까지 폼을 다루는 방법을 소개했습니다. 마지막에 간단한 Mixin을 살펴봄으로써 Mixin이 동작하는 방식도 알 수 있을 것이라 생각합니다. 다음 절에서는 React.js의 VIRTUAL DOM 구현에서 중요한 역할을 맡고 있는 key

속성을 소개하겠습니다.

React.js에서 중요한 key

이번 절에서는 React.js의 Virtual DOM 구현의 내에서도 유저가 인지할 수 있는 Key를 소개하겠습니다. React.js에서는 Prop에 key라는 값을 지정할 수 있고 컴포넌트의 리스트를 렌더링할 때 이를 지정하지 않으면 Development 환경에서 아래와 같은 경고가 출력됩니다.

Each child in an array should have a unique "key" prop. Check the render method of KeyTrap. See http://fb.me/react-warning-keys for more information.

이 key는 VIRTUAL DOM과 비교하여 실제 DOM에 반영할 때 최소한으로 변경하기 위해 사용됩니다. key를 사용하는 예는 다음과 같습니다.(참고)

```
var KeySample = React.createClass({
  getInitialState() {
    return {
      list: [1,2,3,4,5]
    };
  },
  add() {
    this.setState({ list: [0].concat(this.state.list) });
  },
  render() {
    var list = this.state.list.map(function(i) { return <li key={i}>{i}</li> });
    return (
      <div>
        <ul>{list}</ul>
        <button onClick={this.add}>add</button>
      </div>
    );
  }
});
```

위와 같은 원소로 유니크한 ID가 지정돼 있는 배열을 리스트로 출력하는 컴포넌트가 있다고 했을때, 새로 추가 시 배열의 앞에 0을 추가하면 DOM에도 실제로 변경이 필요한 부분만 반영됩니다. 만약 key를 사용하지 않으면 이런 비교가 불가능하여 전체 리스트를 갱신하게 됩니다. 이 예제에는 문제가 있는데 한번 추가한 후 다시 추가하면 0이라는 key를 가진 배열이 계속 추가되므로 실제로 변경된 사항이 없다 판단하여 DOM은 바뀌지 않습니다. 이러 형태의 문제가 발생했을때는 아래와 같은 경고가 출력됩니다.

Warning: flattenChildren(...): Encountered two children with the same key, .$0. Child keys must be unique; when two children share a key, only the first child will be used.

key를 제거하고 예제를 실행하면 같은 값을 가지는 엘리먼트가 계속 추가됩니다. 이와 비슷한 아이디어는 Angular.js의 track by와 Vue.js의 trackby 등 다른 라이브러리나 프레임워크에서도 만날 수 있습니다.

key must by unique

위 경고로 알 수 있듯이 key는 해당 리스트에서 반드시 유니크한 값으로 지정할 필요가 있습니다. 예를 들어 사용자 목록을 출력한다면 사용자의 ID가 key로 사용될 수 있습니다. 배열의 index를 key로 지정하는 것은 사실 큰 의미가 없습니다.

ReactCSSTransitionGroup

React.js에는 CSS 애니메이션을 위한 addon이 있습니다. 이는 애니메이션 대상이 되는 요소가 1개인 경우에도 key를 지정해야합니다. 이는 ReactCSSTransitionGroup에서 요소의 추가, 삭제를 추적해야 하기 때문에 key를 필요로 하는 것 같습니다.(실제 구현을 살펴보진 않았습니다.) ReactCSSTransitionGroup에 관해서는 추후 다시 소개하겠습니다.

추가 내용

마지막으로 React.js and Dynamic Children - Why the Keys are Important 을 참고해 key에 관해 생략된 부분을 소개하겠습니다.

```
<CountriesComponent>
  <TabList />  {/* 나라 리스트 */}
  <TabList />  {/* 위 나라에 해당하는 도시 리스트 */}
</CountriesComponent>
```

위와같은 컴포넌트를 구성하고 있고 TabList는 각각 활성화된 탭의 index를 State로 가지고 있다고 합시다. 그리고 국가 목록을 변경했을 때 도시 목록의 활성화된 index도 0으로 되돌리고 싶지만 의도한대로 동작하지 않습니다. getInitialState()에 활성화 index가 0으로 초기화 되도록 작성돼 있습니다. 따라서 나라가 변경됐을 때 도시 목록의 TabList는 나라에 대응한 도시의 리스트로 갱신되면서 초기화 될 것으로 보이지만 실제로 TabList를 재사용하므로 목록만 갱신됩니다. 즉, getInitialState()가 호출되지 않아 활성화 index가 갱신되지 않아 발생하는 문제입니다.

이 문제는 TabList에 key를 지정하고 국가가 달라졌을 때 도시 컴포넌트가 다시 생성되도록 하는 방식으로 해결할 수 있습니다.즉, key를 명시함으로써 새로운 컴포넌트를 만들도록 할 수 있습니다.

```
<CountriesComponent>
  <TabList key='countriesList'/>
```

```
  <TabList key={this.state.currentCountry} />
</CountriesComponent>
```

위 블로그에도 언급돼 있지만 이런 경우엔 TabList 컴포넌트에서 활성화 index를 State로 관리하는게 아니라 ContriesComponent가 관리하고 Prop으로 활성화 index를 TabList 컴포넌트에 전달하는게 더 맞는 방법인 것 같습니다.

정리

여기까지 React.js 컴포넌트의 Lifecycle과 이벤트 그리고 폼과 Key를 소개했습니다. 다음 편에서는 VIRTUAL DOM의 장점과 믹스-인 등을 소개하겠습니다.

React.js가 VIRTUAL DOM을 채택하고 있어 좋은 점

읽기전에...

이 문서는 koba04님이 작성한 React.js Advent Calendar를 번역한 것입니다. 본래 원문서는 캘린더 형식으로 소개하지만 여기에서는 회를 나눠 작성할 생각입니다. 또한, React 버전 0.12.1 때 작성된 문서이기 때문에 현 버전과 다른 점이 있을 수 있습니다. 최대한 다른 부분을 노트로 작성할 생각이지만, 만약 생략된 부분이 있다면 댓글로 알려주시면 감사하겠습니다.

이번에는 React.js의 VIRTUAL DOM을 간단히 소개하겠습니다. VIRTUAL DOM의 자세한 설명은 VirtualDOM Advent Calendar 2014(일본어)를 참고하세요. 사실 이 캘린더 만으로도 Virtual DOM을 충분히 이해할 수 있지만 흐름 상 한번 다뤄야 할 것 같아 작성합니다.

VIRTUAL DOM의 좋은 점

자바스크립트를 사용해 DOM을 조작하여 UI를 변경하는 애플리케이션의 경우, 사용자 경험을 해치지 않기 위해서라도 갱신되는 DOM을 최소한으로 유지합니다. 예를 들어 Backbone.js를 사용한다면 기본적으로 뷰 단위로 렌더링 하므로 뷰를 아주 잘게 나누는 것이 중요합니다. 그러면 뷰의 개수가 늘어나고 관계가 복잡해져 관리하기 힘듭니다. Angular.js의 경우는 Dirty Checking 하여 변경이 있을 때 다시 랜더링 되는 형식입니다. 이런 방법은 감시 대상이 늘어날수록 성능이 떨어지는 문제가 있습니다.(이 성능 문제를 개선하고자 버전 2부터는 Object.observe를 사용하도록 변경됩니다.)

React.js의 경우는 setState(forceUpdate)가 호출되면 그 컴포넌트와 하위 컴포넌트가 다시 랜더링되는 대상이 됩니다. 이 말을 듣게 되면 매번 광범위하게 DOM이 갱신된다고 느껴지지만 React.js에서는 VIRTUAL DOM이라

고 하는 형태로 메모리상에 DOM의 상태를 유지하고 있고 전/후 상태를 비교하여 달라진 부분만 실제 DOM에 반영합니다. 참고로 CSS도 마찬가지로 객체 형식으로 지정해 변경된 Style만 갱신합니다.

```
var Hoge = React.createClass({
  getInitialState() {
    return {
      style: {
        color: "#ccc",
        width: 200,
        height: 100
      }
    };
  },
  onChange() {
    var style = _.clone(this.state.style);
    style.color = "#ddd";
    this.setState({ style: style});
  },
  render() {
    return (
      <div style={this.state.style} onClick={this.onChange}>xxx</div>
    );
  }
}
```

이러한 방식으로 성능 문제를 해결한 것은 물론, 성능이 중요하지 않은 애플리케이션에서도 상위 레벨의 요소에 애플리케이션의 상태를 갖게 하고 그것을 setState()로 점점 갱신하는 것과 같은 조금은 거친 느낌으로 아키텍처도 할 수 있습니다. 서버 사이드의 렌더링과 비슷하네요. 즉, DOM을 다룰 때 신경 써야 하는 귀찮고 성능에 영향을 주는 부분을 React.js에 맡기는 것으로 애플리케이션의 구현을 단순하게 할 수 있는 특징이 있습니다.
애플리케이션 개발자가 VIRTUAL DOM을 직접 신경 쓰는 경우는 key 속성 지정과 성능 향상의 목적으로 shouldComponentUpdate()를 구현할 때입니다.

shouldComponentUpdate

shouldComponenetUpdate()에 관해서는 Component Lifecycle을 다룰 때 설명했습니다. 이 메서드를 구현(재정의)하지 않는 경우엔 UI를 항상 갱신하도록 구현돼 있습니다. 이 메서드가 false를 반환하면 그 컴포넌트와 하위 컴포넌트의 UI를 갱신하지 않습니다.(참고)

```
var shouldUpdate =
      this._pendingForceUpdate ||
      !inst.shouldComponentUpdate ||
      inst.shouldComponentUpdate(nextProps, nextState, nextContext);
```

최소한의 DOM만 갱신되는 메커니즘으로 인해 항상 UI를 갱신하도록 구현해도 문제가 안 될 것 같지만, 매번 VIRTUAL DOM 트리를 만들어 실제 DOM을 비교하는 작업을 하게 되므로 실제 DOM은 갱신되지 않더라도 비용 들어 갑니다. 따라서 컴포넌트의 State와 Prop의 전/후 상태를 비교하여 변경이 있는 경우에만 컴포넌트와 하위 컴포넌트의 VIRTUAL DOM의 트리를 만들어 실제 DOM과 비교하여 UI를 갱신하도록 하는 것이 조금 더 비용을 낮추는 방법입니다.

React.js 이 외의 VIRTUAL DOM

React.js 외에도 VIRTUAL DOM을 채용하고 있는 라이브러리로는 mercury와Mithril 등 여러 가지가 있고, Ember.js도 버전 2.0에서 VIRTUAL DOM의 구현을 검토(참고)하고 있습니다. 또한, 구현에 관해 알고 싶은 사람들은 vdom이나deku의 소스부터 살펴나가는 것을 추천합니다.
여기까지 VIRTUAL DOM을 소개했습니다. 다음 절에서는 spread attributes를 사용하여 컴포넌트를 작성하는 방법을 소개하겠습니다.

Spread Attributes

이번에는 기존의 컴포넌트를 Spread Attributes를 사용하여 간단하게 컴포넌트를 확장하는 방법을 가볍게 소개하려고 합니다. Spread Attributes는 React.js 버전 0.12에 추가된 기능입니다.

텍스트와 함께 출력되는 이미지 컴포넌트

예로써, 텍스트와 이미지를 한데 묶은 ImageText 컴포넌트를 사용합니다. 이 컴포넌트의 I/F는 이미지 경로와 텍스트를 전달할 수 있도록 디자인했습니다.

```
var ImageText = React.createClass({
  render() {
    return (
      <span>
        {this.props.text}
        <img src={this.props.src} width={this.props.width} height={this.props.height} />
      </span>
    );
  }
```

```
});
<ImageText text="이름" src="/img/foo.png" width="100" height="200" />
```

위와 같은 느낌으로 단순하게 구현할 수 있습니다. 하지만 이미지 태그를 표기할 때는 alt 어트리뷰트가 필요합니다. 여기에 또 추가하자니 귀찮습니다. 이런 문제는 Spread Attributes를 사용하면 다음과 같이 작성할 수 있습니다.

```
var ImageText = React.createClass({
  render() {
    var {text, ...other} = this.props;
    return (
      <span>{text}<img {...other} /></span>
    );
  }
});
```

Spread Attributes를 이용해 text와 ohter를 나누어 전달하면 이미지 어트리뷰트 갯수나 형식에 상관없이 사용할 수 있습니다. 자바스크립트로도 _.omit()을 이처럼 사용할 수 있습니다. 하지만 이렇게 작성할 경우 컴포넌트의 I/F를 알기 어려워지므로 PropTypes를 될 수 있으면 지정해두는 편이 좋다고 생각합니다.

클릭 이벤트 발생 시 Ajax 요청

이번에는 클릭 이벤트 발생 시 Ajax를 요청하도록 해보겠습니다.

```
var request = require('superagent');
var ImageText = React.createClass({
  onClick() {
    request.get("/click_img", { img: this.props.src });
  },
  render() {
    var {text, ...other} = this.props;
    return (
      <span>{text}<img {...other} onClick={this.onClick} /></span>
    );
  }
});
```

위와 같이 onClick()을 추가하면 Prop의 값과 자동으로 merge 합니다. 만약 {...other} 앞에 onClick()을 선언하면 Prop의 onClick을 우선시하여 덮어쓰므로 주의가 필요합니다.

```
var Hello = React.createClass({
    onClick() {
        alert('inner');
```

```
    },
    render: function() {
        var {name, ...other} = this.props;
        // 클릭시 inner 출력
        return <div>Hello <span {...other} onClick={this.onClick}>{name}</span></div>;
        // 클릭시 outer 출력
        return <div>Hello <span onClick={this.onClick} {...other}>{name}</span></div>;
    }
});
function onClick() {
    alert('outer');
}
React.render(<Hello name="World" onClick={onClick}/>, document.getElementById('container'));
```

Spread Attributes는 JSX 없이도 _.extend(), Object.assign() 등을 사용하여 구현할 수 있습니다. 하지만 JSX의 spread attributes 사용하는 편이 조금 더 편리한 것 같습니다. 다음 절에서는 mixin을 소개하겠습니다.

React.js의 믹스-인

이번에는 컴포넌트의 믹스-인 기능을 소개하겠습니다. 보통 믹스-인은 이름 그대로 기능을 수집하는 수단을 말하고 React.js에서 믹스-인은 컴포넌트의 공통 로직을 Object로 분리하여 공통적으로 사용할 수 있도록 하는 기능 뜻합니다. React.js 자체도 LinkedStateMixin이나 PureRenderMxin 등의 믹스-인을 제공하고 있습니다. 덧붙여 Marionette.js에서는 Behavior로, Vue.js에서는 믹스-인이라는 이름으로 같은 기능이 존재합니다.

읽기전에...

아쉽지만 ES6 문법에서는 Mixin을 사용할 수 없습니다.(참고), react-mixin으로 사용할 수 있지만, 개인적으로 깔끔하진 않은 거 같습니다.

사용 방법

Object를 배열로 지정하는 방식으로 사용합니다. 배열을 보면 알 수 있듯이 복수 지정이 가능합니다.

```
var Logger = {
  logging(str) {
    console.log(str);
  },
  componentDidMount() {
    this.logging("component did mount");
  }
};
var Hello = React.createClass({
  mixins: [Logger],
```

```
  render() {
    this.logging("render");
    return <div>Hello</div>
  }
});
```

믹스-인이 로드되는 순서

복수의 믹스-인을 지정할 수 있다고 말씀드렸습니다. 그럼 어떤 순서로 로드될까요? 예상대로 배열의 순서대로 믹스-인이 호출된 후 마지막에 컴포넌트의 메서드가 호출되는 것을 확인할 수 있습니다.

```
var MixinA = {
  componentWillMount() {
    console.log("mixinA");
  }
};

var MixinB = {
  componentWillMount() {
    console.log("mixinB");
  }
};

var Hello = React.createClass({
  mixins: [MixinA, MixinB],
  componentWillMount() {
    console.log("hello");
  },
  render() {
    return <div>hello</div>
  }
});
React.render(<Hello />, document.body);
// mixinA
// mixinB
// hello
```

Conflict State or Prop

getInitialState와 getDefaultProps 등을 믹스-인으로 지정하면 어떻게 될까요?

getInitialState

아래 예제를 보면 알 수 있듯이 State 값을 합칩니다.

```
var Mixin = {
  getInitialState() {
```

```
    return {
      mixinValue: "mixin state"
    };
  }
};
var Hello = React.createClass({
  mixins: [Mixin],
  getInitialState() {
    return {
      componentValue: "component state"
    };
  },
  render() {
    console.log(this.state);
    return <div>hello</div>
  }
});
React.render(<Hello />, document.body);
//  Object {mixinValue: "mixin state", componentValue: "component state"}
```

getDefaultProps

Props도 State와 마찬가지로 값을 합칩니다.

```
var Mixin = {
  getDefaultProps: function() {
    return {
      mixinValue: "mixin prop"
    };
  }
};

var Hello = React.createClass({
  mixins: [Mixin],
  getDefaultProps: function() {
    return {
      componentValue: "component prop"
    };
  },
  render: function() {
    console.log(this.props);
    return <div>hello</div>
  }
});

React.render(<Hello />, document.body);
 Object {mixinValue: "mixin prop", componentValue: "component prop"}
```

getInitialState에서 같은 key를 지정

만약 믹스-인과 같은 key를 지정할 경우엔 에러가 발생합니다.

```
var Mixin = {
  getInitialState() {
    return {
      value: "mixin state"
    };
  }
};

var Hello = React.createClass({
  mixins: [Mixin],
  getInitialState() {
    return {
      value: "component state"
    };
  },
  render() {
    console.log(this.state);
    return <div>hello</div>
  }
});

React.render(<Hello />, document.body);
//  Uncaught Error: Invariant Violation: mergeObjectsWithNoDuplicateKeys(): Tried to merge two objects with the same key: `value`. This conflict may be due to a mixin; in particular, this may be caused by two getInitialState() or getDefaultProps() methods returning objects with clashing keys.
```

메서드 재정의

믹스-인과 동일한 이름의 메서드를 컴포넌트에서 선언해 재정의 할때도 에러가 발생합니다.

```
var Mixin = {
  foo: function() {
    console.log("mixin foo");
  }
};

var Hello = React.createClass({
  mixins: [Mixin],
  foo: function() {
    console.log("component foo");
  },
  render: function() {
    return <div>hello</div>
```

```
  }
});

React.render(<Hello />, document.body);
// Uncaught Error: Invariant Violation: ReactCompositeComponentInterface: You are attempting to define `foo` on your component more than once. This conflict may be due to a mixin.
```

믹스-인을 이용하면 코드를 줄일 수 있습니다. 로직을 어렵게 하지 않을 수준에서 잘 사용하길 바랍니다. 여기까지 믹스-인을 소개했습니다. 다음 절에서는 애드온을 소개하겠습니다.

React.js의 애드온

이번에는 에드온을 소개하겠습니다. 엔드온은 코어에 들어갈 수준은 아닌 편리한 믹스-인이나 테스트 유틸, 성능 측정 도구 등을 모아 놓은 부가 기능입니다.

사용 방법

애드온은 require하거나 js 파일을 로드하는 것으로 사용할 수 있습니다.

```
var React = require('react/addons');
<script src="//cdnjs.cloudflare.com/ajax/libs/react/0.12.1/react-with-addons.js"></script>
```

애드온

TransitionGroup and CSSTransitionGroup

애니메이션을 하기 위한 애드온입니다. 이 애드온은 다음에 자세히 소개하겠습니다.

LinkedStateMixin

이 애드온은 이전에 한번 소개한 폼을 다룰 때 양방향 데이터 바인딩과 같은 로직을 간결하게 작성하기 위한 믹스-인입니다.

ClassSet

className 지정을 쉽게 하기 위한 애드온입니다. {className: boolean} 형식으로 지정할 수 있고 boolean이 true인 className만 적용됩니다. Angular.js나 다른 프레임워크에도 있는 기능입니다.(참고) 이 애드온은 곧 삭제될 예정입니다. 대신 classnames 같은 별도의 npm 모듈을 사용하도록 권고하고 있습니다.

```
var classSet = React.addons.classSet;

var Hello = React.createClass({
  getInitialState() {
    return {
      isWarning: false,
```

```
      isImportant: false
    };
  },
  toggleWarning() {
    this.setState({ isWarning: !this.state.isWarning });
  },
  toggleImportant() {
    this.setState({ isImportant: !this.state.isImportant });
  },
  render() {
    var style = classSet({
      'is-warning': this.state.isWarning,
      'is-important': this.state.isImportant
    });
    return (
      <div>
        <button onClick={this.toggleWarning}>warning</button>
        <button onClick={this.toggleImportant}>important</button>
        <p className={style}>( ´ ▽ ` )ﾉ</p>
      </div>
    );
  }
});
```

TestUtils

React.js를 테스트할 때 편리하게 사용할 수 있는 애드온이며 개발 환경에서만 사용할 수 있습니다. click 이벤트와 같은 이벤트를 시뮬레이터 하는 TestUtils.Simurate나 isElementOfType과 isDOMComponent 등 컴포넌트의 상태를 확인할 수 있는 함수까지 여러 가지 있습니다.(React.js 테스트는 추후 다시 소개하겠습니다.)

cloneWithProps

이 애드온을 사용하는 경우는 많지 않습니다. 어떤 컴포넌트에서 다른 Prop에 의한 새로운 컴포넌트를 만들고 싶을 때 사용합니다.

```
var cloneWithProps = React.addons.cloneWithProps;

var Item = React.createClass({
  render: function() {
    var text = this.props.text + (this.props.index != null ? ":" + this.props.index : '');
    return <div>{text}</div>
  }
});
```

```
var Loop = React.createClass({
  render: function() {
    var items = _.map(_.range(this.props.count), function(i) {
      return cloneWithProps(this.props.children, { key: i, index: i });
    }.bind(this));
    return <div>{items}</div>
  }
});

React.render(<Loop count="10"><Item text="hoge" /></Loop>, document.body);
```

위는 횟수만큼 children 컴포넌트를 만드는 과정을 cloneWithProps 애드온을 사용해 작성한 것입니다.

update

Object를 Immutable하게 조작하기 위한 애드온입니다. 뒤에서 설명할 PureRenderMixin() 또는 Prop과 State를 비교해 최적화하는 용도의 shouldComponentUpdate와 함께 조합해서 사용하면 편리합니다.

```
var update = React.addons.update;
var obj = {
  list: [1,2,3],
};
var obj2 = update(obj, {
  list: {
    $push: [4]
  }
});
console.log(obj2.list);      // ['a','b','c','d']
console.log(obj === obj2);  // false
```

참고로 페이스북은 별도의 Immutable.js를 만들고 있습니다. 이를 다음과 같이 사용할 수도 있습니다.

```
var obj = Immutable.Map({
    list: Immutable.List.of(1, 2, 3)
});
var obj2 = obj.set('list', obj.get('list').push(4));
console.log(obj2.get('list').toArray()); // ['a','b','c','d']
console.log(obj === obj2); // false
```

PureRenderMixin

성능을 최적화하기 위한 믹스-인입니다. 아래 코드를 살펴보겠습니다.(참고)

```
var ReactComponentWithPureRenderMixin = {
  shouldComponentUpdate: function(nextProps, nextState) {
    return !shallowEqual(this.props, nextProps) ||
```

```
            !shallowEqual(this.state, nextState);
  }
};
```

위 믹스-인이 사용하는 shallowEqual은 다음과 같이 작성돼 있습니다. 중첩된 값까지는 고려하지 않고 단순하게 비교합니다.(참고)

```
function shallowEqual(objA, objB) {
  if (objA === objB) {
    return true;
  }
  var key;
  // Test for A's keys different from B.
  for (key in objA) {
    if (objA.hasOwnProperty(key) &&
        (!objB.hasOwnProperty(key) || objA[key] !== objB[key])) {
      return false;
    }
  }
  // Test for B's keys missing from A.
  for (key in objB) {
    if (objB.hasOwnProperty(key) && !objA.hasOwnProperty(key)) {
      return false;
    }
  }
  return true;
}
```

Perf

성능 측정을 위한 애드온입니다. 개발 환경에서만 사용할 수 있습니다.

Perf.start()와 Perf.stop()으로 성능을 측정하고 싶은 로직을 둘러싸고 수치화할 수 있습니다.

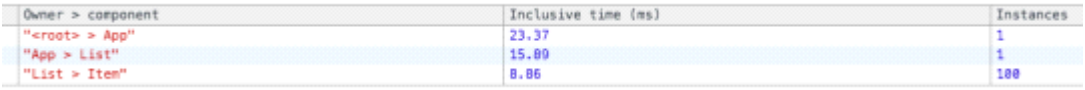
```
React.addons.Perf.start();
this.setState({ items: items }, function() {
  React.addons.Perf.stop();
  React.addons.Perf.printInclusive();
});
```

어떤 식으로 수치화되는지 확인하기 위해 Item 컴포넌트를 100개 추가하는 로직을 성능 측정하는 예제를 작성했습니다. 측정 결과는 개발자 콘솔에서 확인할 수 있습니다.

측정 결과의 수치가 매우 작은 경우엔 출력이 무시되니 참고바랍니다.

printInclusive

측정 중인 컴포넌트 처리에 걸린 시간을 알기 쉽게 출력합니다.

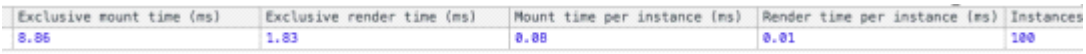


Owner > component	Inclusive time (ms)	Instances
"<root> > App"	23.37	1
"App > List"	15.09	1
"List > Item"	8.86	100

<그림 1 printInclusive>

printExclusive

컴포넌트 처리에 걸린 시간을 더 상세히 출력합니다.



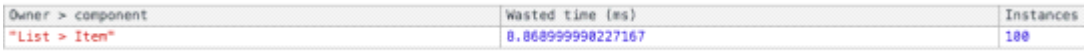
Exclusive mount time (ms)	Exclusive render time (ms)	Mount time per instance (ms)	Render time per instance (ms)	Instances
8.86	1.83	0.08	0.01	100

<그림 2 printExclusive>

printWasted

실제 렌더링 처리 이외에 걸린 시간을 출력합니다.

shouldComponenetUpdate()를 적용하는 타이밍을 찾기 위한 단서로 사용합니다.

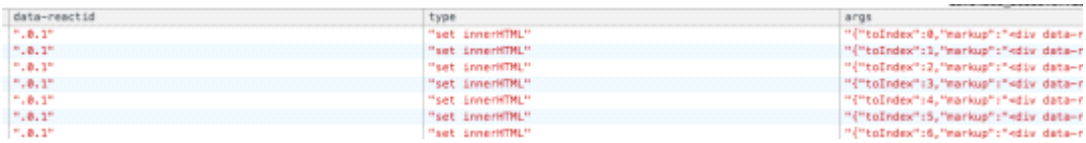


Owner > component	Wasted time (ms)	Instances
"List > Item"	0.868999998227167	100

<그림 3 printWasted>

printDOM(measurements)

돔을 추가하거나 삭제한 내역을 출력합니다.



data-reactid	type	args
".0.1"	"set innerHTML"	"{"toIndex":0,"markup":"<div data-r
".0.1"	"set innerHTML"	"{"toIndex":1,"markup":"<div data-r
".0.1"	"set innerHTML"	"{"toIndex":2,"markup":"<div data-r
".0.1"	"set innerHTML"	"{"toIndex":3,"markup":"<div data-r
".0.1"	"set innerHTML"	"{"toIndex":4,"markup":"<div data-r
".0.1"	"set innerHTML"	"{"toIndex":5,"markup":"<div data-r
".0.1"	"set innerHTML"	"{"toIndex":6,"markup":"<div data-r

<그림 4 printDOM>

getLastMeasurements

성능 측정 결과를 Object 형식으로 가져올 수 있습니다. 서버에 결과를 보내거나 위에서 소개한 각 메서드에 값을 넘겨줄 수도 있습니다. 측정 후 보기 좋게 정리하기 위해서도 사용할 수 있습니다.

React.js에서 애니메이션 처리하기

이번에는 React.js에서 애니메이션을 다루는 방법을 소개하겠습니다. React.js에서는 애니메이션을 Addon으로 지원하고 있으며 CSS 애니메이션과 CSSTransitionGroup addon을 사용하는 방식과 컴포넌트의 Lifecycle 메서드와 같은 메서드에서 훅(hook)하여 작성하는 두 가지 패턴으로 애니메이션을 처리할 수 있습니다.

CSSTransitionGroup

CSSTransitionGroup을 이용하면 컴포넌트를 추가/삭제 시 CSS 애니메이션을 줄 수 있습니다. 방법은 Angular.js와 Vue.js와 비슷합시다. 추가/삭제 시 클래스를 추가하여 CSS 애니메이션을 처리하는 방식입니다.

{transitionName}-{enter, leave} 패턴으로 클래스 명이 추가된 뒤, 다음 이

벤트 루프에서 {transitionName}-{enter, leave}-active의 className이 추가되는데 이때 이 클래스 명을 사용하여 CSS애니메이션을 처리합니다.(참고)

```
var CSSTransitionGroup = React.addons.CSSTransitionGroup;

var Hello = React.createClass({
  getInitialState: function() {
    return {
      value: '(´・ω・`)'
    };
  },
  onClick: function() {
    var value = this.state.value === '(´・ω・`)' ? '( `・ω・´)ゞ' : '(´・ω・`)';
    this.setState({ value: value });
  },
  render: function() {
    var value = <span className="sample" key={this.state.value}>{this.state.value}</span>;
    return (
      <div>
        <div>Animation!!<button onClick={this.onClick}>click!!</button></div>
        <CSSTransitionGroup transitionName="sample">
          {value}
        </CSSTransitionGroup>
      </div>
    );
  }
});

React.render(<Hello />, document.body);
.sample-enter {
     -webkit-transition: 1s ease-in;
}
.sample-enter.sample-enter-active {
    font-size: 80px;
}
.sample-leave {
    -webkit-transition: .5s ease-out;
}
.sample-leave.sample-leave-active {
    font-size: 10px;
}
```

주의할 점

애니메이션 되는 요소에는 반드시 key를 지정해야 합니다. 애니메이션 되

는 요소가 1개라도 반드시 지정해야 합니다. 이는 컴포넌트가 추가됐는지 아니면 갱신됐는지를 알려주기 위함입니다. 이것을 이용하면 앞에서 소개한 예처럼 컴포넌트가 1개라도 key를 변경하는 것으로 애니메이션을 적용할 수 있습니다.(key를 변경했다는 뜻은 컴포넌트를 추가[또는 갱신]/삭제했다는 뜻이므로)

애니메이션은 추가(enter) 시와 삭제(leave) 시 두 경우 모두에 지정할 필요가 있습니다. 만약 한 경우에만 애니메이션을 지정하고 싶다면 transitionEnter={false}, transitionLeave={false}를 지정합니다.

```
<CSSTransitionGroup transitionName="sample" transitionLeave={false}>
  {value}
</CSSTransitionGroup>
```

CSSTransitionGroup의 컴포넌트는 애니메이션 시작 시엔 이미 랜더링 돼 있어야 합니다. 추가되는 요소와 함께 CSSTransitionGroup의 컴포넌트를 추가하면 애니메이션하지 않습니다. 예를 들어 아래의 경우 처음 click 시엔 CSSTransitionGroup이 없으므로 애니메이션하지 않습니다.

```
var Hello = React.createClass({
  getInitialState: function() {
    return {
      value: ''
    };
  },
  onClick: function() {
    var value = this.state.value === '(´・ω・`)' ? '( `・ω・´)ゞ' : '(´・ω・`)';
    this.setState({ value: value });
  },
  render: function() {
    var value ;
    if (this.state.value) {
      value = (
        <CSSTransitionGroup transitionName="sample">
          <span className="sample" key={this.state.value}>{this.state.value}</span>
        </CSSTransitionGroup>
      );
    }
    return (
      <div>
        <div>Animation!!<button onClick={this.onClick}>click!!</button></div>
          {value}
      </div>
    );
```

```
  }
});
```

ReactTransitionGroup

CSS 애니메이션이 아니라 직접 유연하게 애니메이션 작성하고 싶은 경우엔 ReactTransitionGroup을 사용합니다. componentWillEnter(callback), componentDidEnter(), componentWillLeave(callback), componentDidLeave() 이 4개의 Lifecycle 메서드를 이용해 작성합니다. 또 ReactTransitionGroup은 기본으로 span 요소를 DOM에 추가하는데 <ReactTransitionGroup compoenent='ul'> 문법으로 추가하는 요소를 지정할 수 있습니다.(참고)

```
var TransitionGroup = React.addons.TransitionGroup;
var duration = 1000;
var AnimationComponent = React.createClass({
  componentWillEnter: function(callback) {
    console.log("component will enter");
    $(this.getDOMNode()).hide();
    callback();
  },
  componentDidEnter: function() {
    $(this.getDOMNode()).show(duration);
    console.log("component did enter");
  },
  componentWillLeave: function(callback) {
    console.log("component will leave");
    $(this.getDOMNode()).hide(duration, callback);
  },
  componentDidLeave: function() {
    console.log("component did leave");
  },
  render: function() {
    return <div>{this.props.text}</div>
  }
});

var Hello = React.createClass({
  getInitialState: function() {
    return {
      value: '(´・ω・`)'
    };
  },
  onClick: function() {
```

```
    var value = this.state.value === '(´・ω・`)' ? '(`・ω・´)ゞ' : '(´・ω・`)';
    this.setState({ value: value });
  },
  render: function() {
    var value = <AnimationComponent key={this.state.value} text={this.state.value} />;
    return (
      <div>
        <div>Animation!!<button onClick={this.onClick}>click!!</button></div>
        <TransitionGroup>
          {value}
        </TransitionGroup>
      </div>
    );
  }
});

React.render(<Hello />, document.body);
```

주의할 점

componentWillEnter()와 componentWillLeave() 처리가 끝나게 되면 반드시 callback을 호출해야 합니다.

여기까지 애니메이션에 관해서 간단히 소개했습니다. 이러한 방식으로 애니메이션을 처리하는데 익숙치 않기 때문에 보통 쓰기 어려운 감이 들 수 있습니다. React.js 측에서도 향후 개선점으로 애니메이션도 다루고 있으므로 앞으로는 더욱 쉬워질 것이로 생각합니다.

정리

이번 편에서는 React.js에서 VIRTUAL DOM을 채택해서 가능한 메커니즘과 간단하게 Props를 전달할 수 있는 Spread Attribute 그리고 믹스-인과 애드온, 마지막으로 애니메이션을 처리하는 방법을 소개했습니다. 여기까지 기본적인 React.js 사용법은 모두 소개했습니다. React.js를 이해하는 비용은 그리 비싸지 않습니다. 이 정도의 특징만 숙지해도 큰 무리 없이 컴포넌트를 개발할 수 있습니다. React.js 자체를 사용하는 것보다 컴포넌트를 설계하는 것이 더 어렵고 개발자의 역량에 따라 컴포넌트 효율성이나 디자인이 크게 좌우될 수 있습니다. 많이 만들고 고민해서 좋은 컴포넌트를 만들 수 있길 바랍니다.