React + Decorator + HOC = Fantastic!!

천민호
kesakiyo@naver.com

REACT SEOUL

경 력

2016	제 1, 2회 삼성 대학생 프로그래밍 경시대회 입상
	한국 대학생 프로그래밍 경시대회 은상
2016 ~	조이코퍼레이션 프론트엔드 개발자

목 차

Decorator란?

선언된 클래스와 프로퍼티들을

디자인 시간에 변경하는 편리한 문법

Decorator란?

그래서 어떻게 사용하는걸까??

Decorator란?

```
@withSuperEngine
class Car {
  ...
  @readOnly
  manufacturer = 'ZOYI'
  ...
}
```

Decorator란?

클래스의 속성이 한눈에 보이나요?

Decorator의 선언 및 사용 방법

- Decorator는 사실 함수 표현식
- 함수를 선언한 뒤 '@' 키워드를 이용해 선언된 함수를 Decorator로 사용
- @withSuperEngine, @readOnly, @say.hello, @hello(…) 등의 형태

Decorator란?

클래스, 클래스의 프로퍼티에 적용이 가능

클래스 프로퍼티의 Decorator

프로퍼티의 descriptor를 인자로 받아

새로운 descriptor를 반환하는 형태

클래스 프로퍼티의 Decorator

```
function readonly(target, property, descriptor) {
  descriptor.writable = false
  return descriptor
}

class Car {
  @readonly
  manufacturer = 'ZOYI'
}

const myCar = new Car()
myCar.manufacturer = 'JOY' // 새로운 값을 할당하려고 한다면 에러가 납니다.
```

```
function nonenumerable(target, property, descriptor) {
  descriptor.enumerable = false
  return descriptor
}

class Car {
  @nonenumerable
  acceleration = 10

  manufacturer = 'ZOYI'
}

const myCar = new Car()
for (let key in myCar) {
  console.log(key)
  // manufacturer 만 출력이 된다. acceleration는 열거 대상에서 제외된다.
}
```

클래스 프로퍼티의 Decorator

단 몇 줄만으로 클래스의 프로퍼티 속성을 변경

클래스 프로퍼티의 Decorator

- 이 외에도 많은 속성들을 조정할 수 있음
- 메모이제이션, 자동으로 class에 bind된 메서드 등등
- core-decorator.js 는 수 많은 형태를 미리 정의해 둠

클래스의 Decorator

타겟 클래스의 생성자를 인자로 받아

새로운 생성자로 변경한 뒤 반환하는 형태

클래스의 Decorator - 프로토 타입의 추가

```
function setAnimalSound(sound) {
  return (target) => {
    target.prototype.sound = sound
    return target
  }
}
```

클래스의 Decorator - 프로토 타입의 추가

```
@setAnimalSound('oink')
class Pig {
  say() {
    return this.sound
  }
}

const pig = new Pig()
console.log(pig.say()) // 'oink' 출력
```

```
@setAnimalSound('quack')
class Duck {
  say() {
    return this.sound
  }
}

const duck = new Duck()
console.log(duck.say()) // 'quack' 출력
```

클래스의 Decorator - 프로토 타입의 추가

클래스 내부에 동물 울음소리를 정의하지 않고 Decorator를 사용해 정의

~~이런 코드는 설계 건점에서 봤을 때는 바람직하지 못함~~

클래스의 Decorator - 생성자 바꿔치기

```
function withBus(target) {
  return class Bus {
    say() {
      return 'I am bus'
    }
  }
}
```

클래스의 Decorator - 생성자 바꿔치기

```
@withBus
class Car {
  say() {
    return 'I am car'
  }
}


const car = new Car()
console.log(car.say()) // ‘I am bus’ 출력
```

클래스의 Decorator - 프로토 타입의 추가

클래스 자체를 하이재킹 하는 재미있는 구현

HOC를 이용한 컴포넌트 하이재킹에 적극 사용

HOC란?

- Higher - Order - Component의 약자
- 리액트 컴포넌트 로직을 재사용할 수 있는 고급 기법
- 리액트 API가 아니라 단순히 아키텍쳐

HOC란?

```
function withSay(WrappedComponent) {
  return class extends React.Component {
    say() {
      return 'hello'
    }

    render() {
      return (
        <WrappedComponent
          {...this.props}
          say={this.say} />
      )
    }
  }
}
```

- WrappedComponent를 인자로 받음
- 기존의 props에 원하는 속성들을 결합
- 새로운 컴포넌트로 재탄생

HOC란?

```
@withSay
class withOutSay extends React.Component {
  render() {
    <div>
      {this.props.say()}
    </div>
  }
}
```

props를 준 적이 없어도 say 메소드를 사용할 수 있음

HOC는 상속이 안되는 React에서
단비와 같은 존재

Cross Cutting Concern

~~횡단 관심사~~

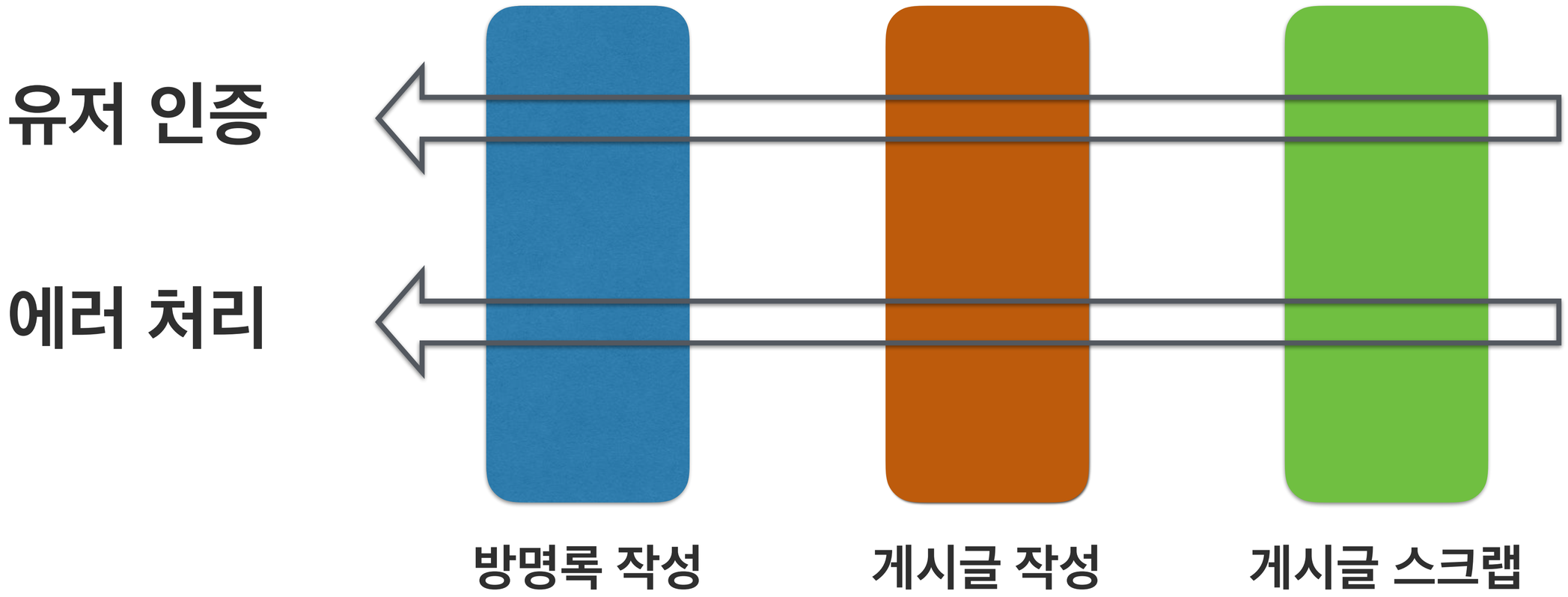
Cross Cutting Concern

모듈화가 쉽지 않은 로직

개발 전반에 걸쳐 자주 등장

계속해서 반복되는 코드

Cross Cutting Concern



Cross Cutting Concern

스파게티 코드의 시작

Cross Cutting Concern

하지만 우리에게는

HOC와 **Decorator**가!!

유저 인증 문제를 HOC로 해결

```
class TeamChat extends React.Component {
  constructor() {
    super()
    this.state = {
      unAuthenticated: false
    }
  }

  componentWillMount() {
    if (!this.props.user) {
      this.setState({
        unAuthenticated: true
      })
    }
  }

  render() {
    if (this.state.unauthenticated) {
      return <UnAuthenticatedComponent />
    }
    return <div>I'm TeamChat</div>
  }
}
```

- 전통적인 if-else 구문으로 유저 인증
- 인증을 해야하는 컴포넌트가 많아지면?
- 인증 로직이 바뀌기라도 한다면?

```
function mustToAuthenticated(WrappedComponent) {
  return class extends React.Component {
    constructor() {
      super()
      this.state = {
        unAuthenticated: false
      }
    }

    componentWillMount() {
      if (!this.props.user) {
        this.setState({
          unAuthenticated: true
        })
      }
    }

    render() {
      if (this.state.unauthenticated) {
        return <UnAuthenticatedComponent />
      }
      return <WrappedComponent {...this.props} />
    }
  }
}
```

유저 인증 문제를 HOC로 해결

```
@mustToAuthenticated
class TeamChat extends React.Component {
  render() {
    return <div>I'm TeamChat</div>
  }
}

@mustToAuthenticated
class UserChat extends React.Component {
  render() {
    return <div>I'm UserChat</div>
  }
}
```

- 확장이 용이한 인증 로직이 탄생!!
- 단 한줄만으로 동일한 로직을 적용
- 인증 로직이 바뀌어도 변경할 곳은 단 하나

i18n 서비스 구현을 HOC로 깔끔하게

```
@connect(state => ({
  locale: getLocale(state)
}))
class Channel extends React.Component {
  render() {
    const locale = this.props.locale
    const translate = TranslateService.get(locale)
    return (
      <div>
        <div>{translate.title}</div>
        <div>{translate.description}</div>
      </div>
    )
  }
}
```

i18n 서비스 구현을 HOC로 깔끔하게

프로젝트가 **커질수록**

번역키를 사용하는 컴포넌트는 **증가**

i18n 서비스 구현을 HOC로 깔끔하게

```
function withTranslate(WrappedComponent) {

  @connect(state => ({
    locale: getLocale(state)
  }))
  class DecoratedComponent extends React.Component {
    render() {
      const locale = this.props.locale
      const translate = TranslateService.get(locale)

      return (
        <WrappedComponent
          {...this.props}
          translate={translate} />
      )
    }
  }

  return DecoratedComponent
}
```

이제는 공식같은

HOC 구현 방법

i18n 서비스 구현을 HOC로 깔끔하게

```
@withTranslate
class Channel extends React.Component {
  render() {
    const translate = this.props.translate
    return (
      <div>
        <div>{translate.title}</div>
        <div>{translate.description}</div>
      </div>
    )
  }
}
```

컴포넌트 전반에 걸쳐 i18n 서비스를 제공해도 단 한줄로!!

Decorator는 중첩도 가능!!

```
@mustToAuthenticated
@withTranslate
class Channel extends React.Component {
  render() {
    const translate = this.props.translate
    return (
      <div>
        <div>{`Hello!! ${this.props.user.name}`}</div>
        <div>{translate.title}</div>
        <div>{translate.description}</div>
      </div>
    )
  }
}
```

우아하고 아름다운 설계를 할 수 있는 방법을 제공

React + Decorator + HOC = Fantastic!!

Fantastic 하지만 만능은 아님

- 과도한 HOC 중첩으로 인해 디버깅이 힘들 수 있음
- WrappedComponent에 직접적으로 ref를 달 수 없어 우회 방법을 사용해야 함
- 비동기 작업과 같이 사용하다가 예상치 못한 결과를 만날 수 있음
- 실제 제품에 사용하다 보면 만나는 몇가지 문제들

React + Decorator + HOC = Fantastic!!