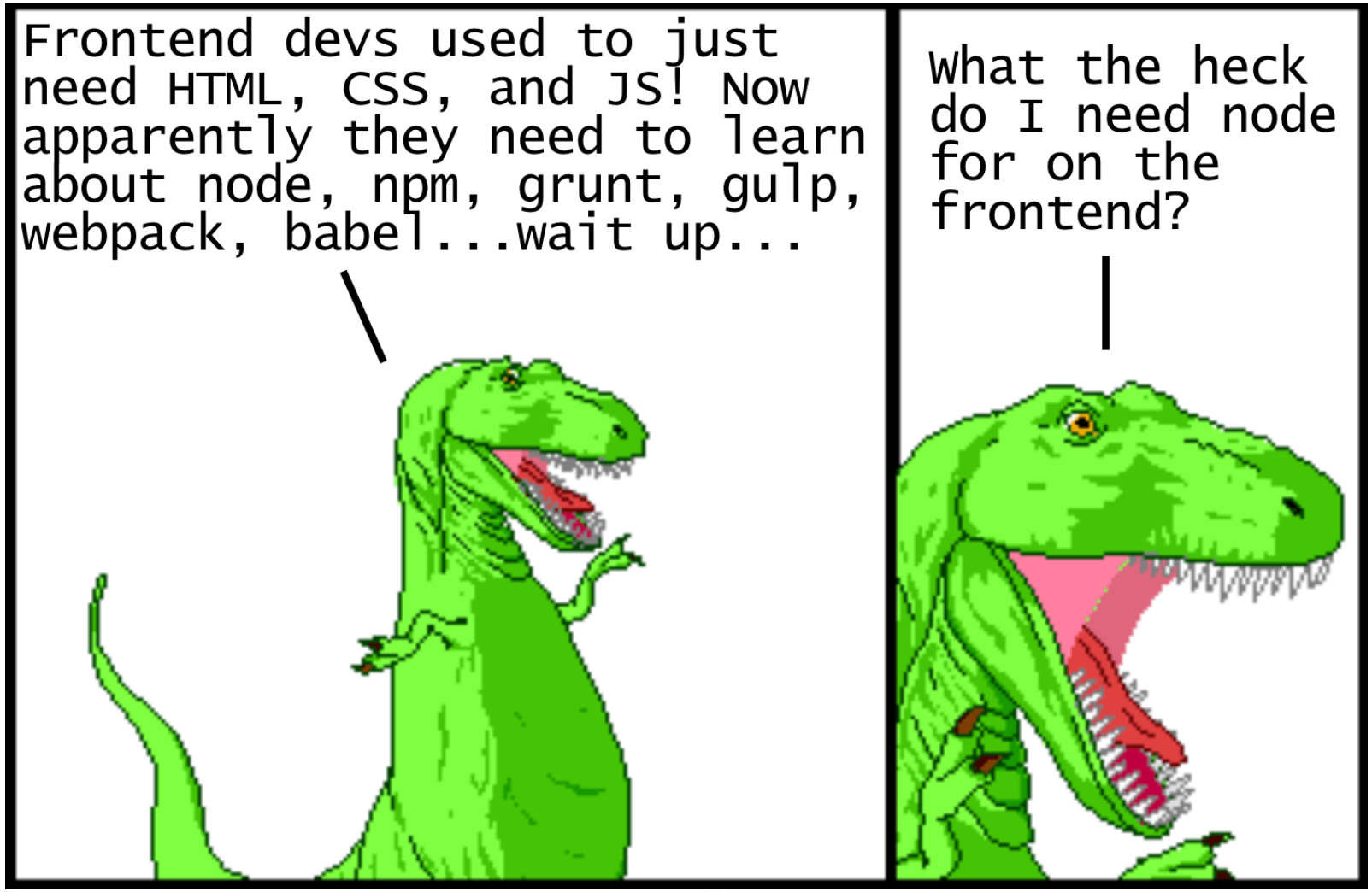
Modern JavaScript

모듈과 번들, 트랜스파일러.

조새나라



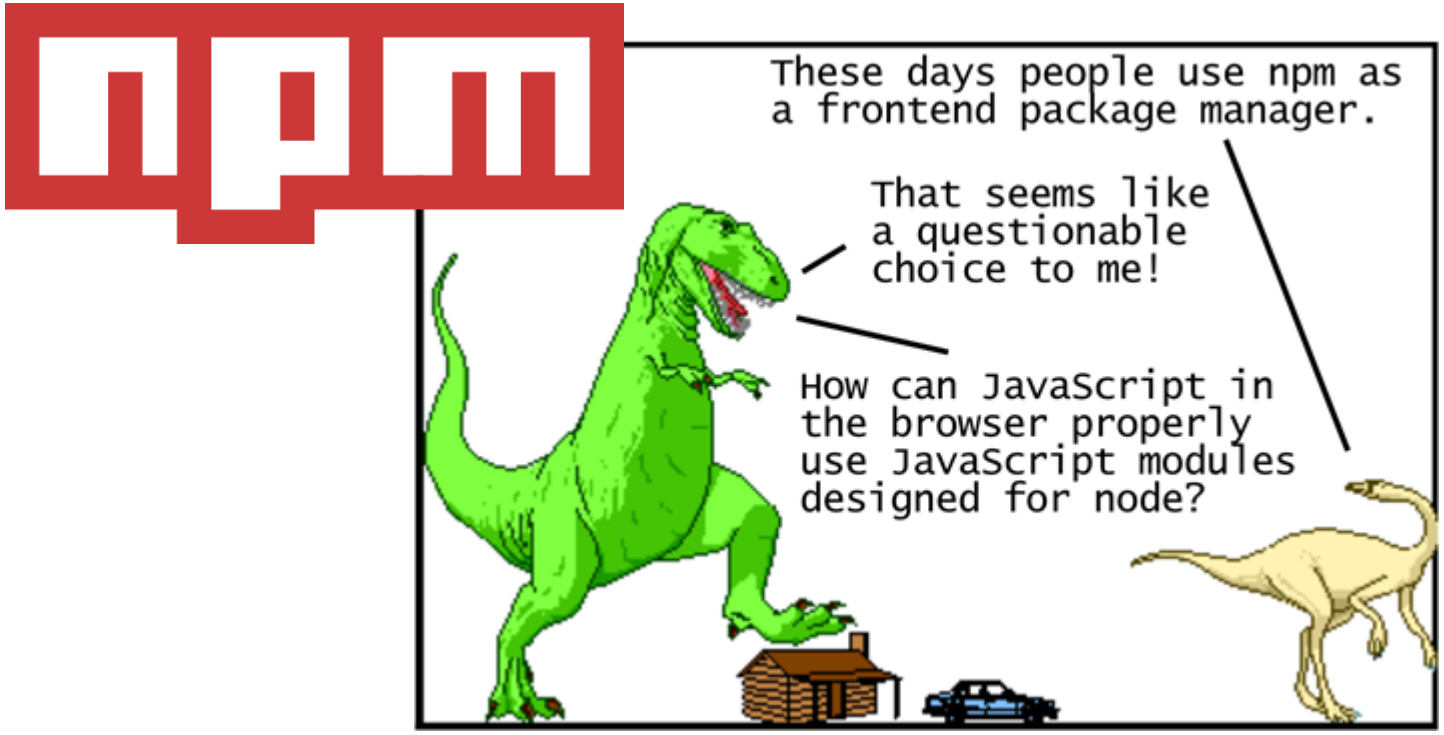
과거에는 프론트엔드 개발자는 단지 Html, CSS, JS(Jquery) 정도만 알면 됐다.
하지만 현재는...

Node.js는 2009년 Ryan Dahl이 개발한 자바스크립트 기반의 런타임 환경이다.

JS를 브라우저 환경이 아닌 서버 환경에서도 사용할 수 있게 하고자 고안해내었으며 따라서 CommonJs 명세를 일부 따르고 있다.

큰 특징은 자바 스크립트가 웹 브라우저에서 실행되는 것이 아닌 웹 서버에서 실행된다는 점이다.

왜 프론트엔드 개발자가 웹 서버에서 실행되는 Node.js를 알아야 할까?



그것은 바로 NPM(Node Package Manager)과 관련이 있다.

NPM은 자바스크립트를 위한 패키지 관리자이다.
여타 패키지 관리자들과 같이 커맨드 라인으로 사용할 수 있는 것이 특징.

NPM 역시 자바스크립트로 작성되어있으며 "모듈 패키징" 이 엉망으로 관리가 되는 것을 본 Isaac Z. Schlueter가 2011년에 개발하였다.

여기서 "자바스크립트 모듈" 이란 또 무엇인가?

자바스크립트 모듈화는 기본적으로 범용적인 목적, 그러니까 브라우저 밖에서
쓰기 위해서 시작된 행위이며 그렇게 사용하기 위한 필요조건이다.

왜 필요조건이 되었을까? 바로 다음과 같은 서버 사이드 자바스크립트의 문제점 때문이다.

- 서로 호환되는 표준 라이브러리가 없다.
- 데이터베이스에 연결할 수 있는 표준 인터페이스가 없다.
- 다른 모듈을 삽입하는 표준적인 방법이 없다.
- 코드를 패키징해서 배포하고 설치하는 방법이 필요하다.
- 의존성 문제까지 해결하는 공통 패키지 모듈 저장소가 필요하다.

더 풀어서 설명하자면 다른 언어들과 달리 자바스크립트는 하나의 코드에서 다른 코드를 불러올 수가 없다.

브라우저 환경에서 구동될 목적으로 만들어졌으며 따라서 보안상의 이유로 파일 시스템에 접근할 수 없기 때문이다.

그래서 그간 자바스크립트는 "전역 변수" 에 할당하여 공유하는 방식을 사용하였다.

이 방식은 큰 문제점이 있다. 각자의 독립적인 스코프가 없으며 접근 권한이 있는지 확인도 불가능해 코드의 충돌과 자원의 낭비, 보안상의 문제가 빈번히 일어나게 된다.

이런 문제의 해결은 바로 "자바스크립트 모듈화" 를 통해 가능하게 되며 모듈화를 통해 자바스크립트를 범용화 시키고자 나온 것이 "CommonJS 프로젝트" 이다.

2009년에 시작된 CommonJS 프로젝트는 자바스크립트를 브라우저 밖에서 사용하기 위해 시작된 프로젝트이다.

자바스크립트의 범용적인 생태계 구성을 위해 시작되었으며 거의 대부분의 명세가 모듈의 정의와 모듈의 사용법으로 되어있을 만큼 이 부분에 있어서 자바스크립트가 큰 약점을 지니고 있다는 것을 방증하며 언어의 범용적인 사용에 있어 가장 중요한 부분이 바로 이 모듈화로 볼 수 있다.

CommonJS에서 정의한 모듈화는 아래의 세 부분으로 정의된다.

- 스코프(Scope): 모든 모듈은 자신만의 독립적인 실행 영역이 있어야 한다.
- 정의(Definition): 모듈 정의는 exports 객체를 이용한다.
- 사용(Usage): 모듈 사용은 require 함수를 이용한다.

- 스코프(Scope)

모듈은 독립적인 실행 영역이 있어야 한다. 기존과 달리 서버사이드 영역일 경우 파일 단위의 독립적인 스코프를 보유하므로 파일 하나에 모듈 하나를 매칭시켜 사용하게 된다.

```
// a.js
var a = 1;
var b = 2;

//b.js
var a = 3;
var b = 2;

//index.js
console.log(a,b);
```

우측의 a.js와 b.js를 받아오는 index.js가 있을 경우
모듈을 사용하지 않은 경우 전역변수 처리가 되어 변수의
충돌이 일어나 a가 선언된 순서에 따라 달라질 것이다.

그러나 모듈을 사용하게 되면 이러한 사용은 index.js에서
변수 a,b를 선언한 모듈에 따라서만 달라지게 된다.

- 정의(Definition)와 사용(Usage)

exports 객체를 이용하여 정의하고 require 함수를 통해 사용한다.

```
//FileA.js
var a = 3;
b=4;
//FileA의 sum 함수를 exports를 통해 공개적으로 정의한다.
exports.sum = function(c, d) {
return a + b + c + d;
}


//FileB.js
var a = 5;
b = 6;
var moduleA = require("FileA"); //FileA.js를 모듈로 사용.
/*
 * FileA.js에서 공개적으로 정의된 함수 sum을 사용한다.
 * FileA 환경의 전역변수 a,b와 c,d의 인자 FileB의 5,6을 받아
 * 충돌없이 연산하여 3+4+5+6 = 18 의 결과값을 얻을 수 있다.
 */
moduleA.sum(a,b);
```

이러한 CommonJS 모듈 명세를 따라 구현된 것 중 가장 유명한 것이 바로 node.js.

그러나 지금까지의 모든 내용은 서버사이드 방면에서 사용하기 위한 모듈화이다.

이 모듈화덕에 자바스크립트의 단점들을 보완해내고 require 함수와 같이 편리한 사용을 맛본 사람들은 슬슬 프론트엔드 개발 시에도 이러한 장점을 이용하고 싶다는 욕구를 가지게 되었다.

더군다나 NPM과 같은 프로젝트 매니저의 존재는 이전의 프론트엔드 개발 환경에선 없었다!

"프론트엔드는 언제까지 패키지를 직접 설치할 것이며 자동화 관리를 하지 않을 것인가?"
"모듈화를 통한 장점을 프론트엔드에도 이용할 수 없을까?"

이러한 개발자들의 욕구로 프론트엔드에서도 이 장점들을 사용할 수 있는 방식이 고안되었다.

이런 욕구로 등장한 것이 바로 웹팩이다.

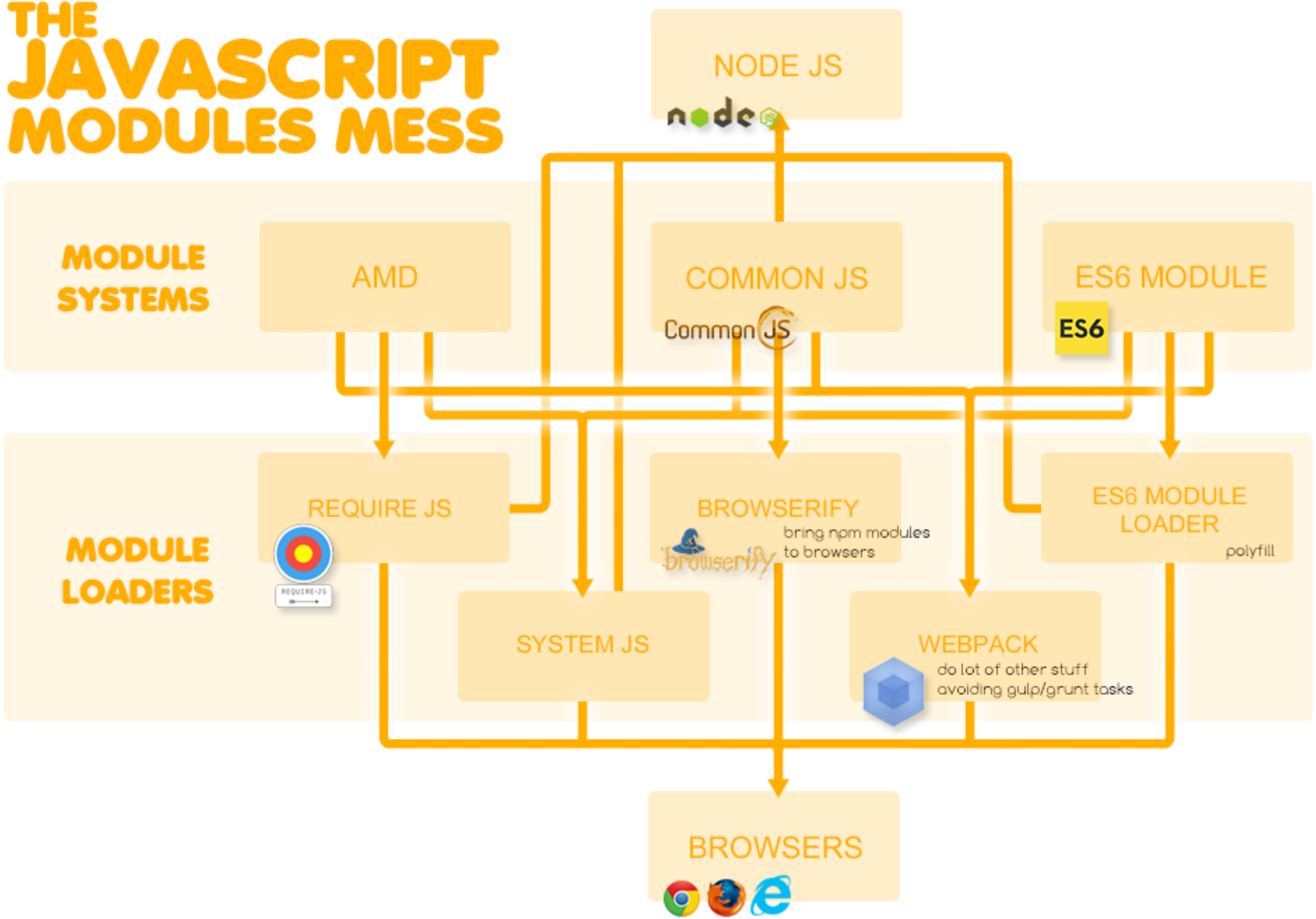
NPM은 웹팩과 별개로 JS로 짜여진 패키지 매니저이기 때문에 얼마든지 프론트엔드에서 사용이 가능했다. 다만 실제로 담겨있는 node_modules 폴더의 깊은 경로까지 들어갔어야만 수동으로 HTML에 불러올 수 있었다.

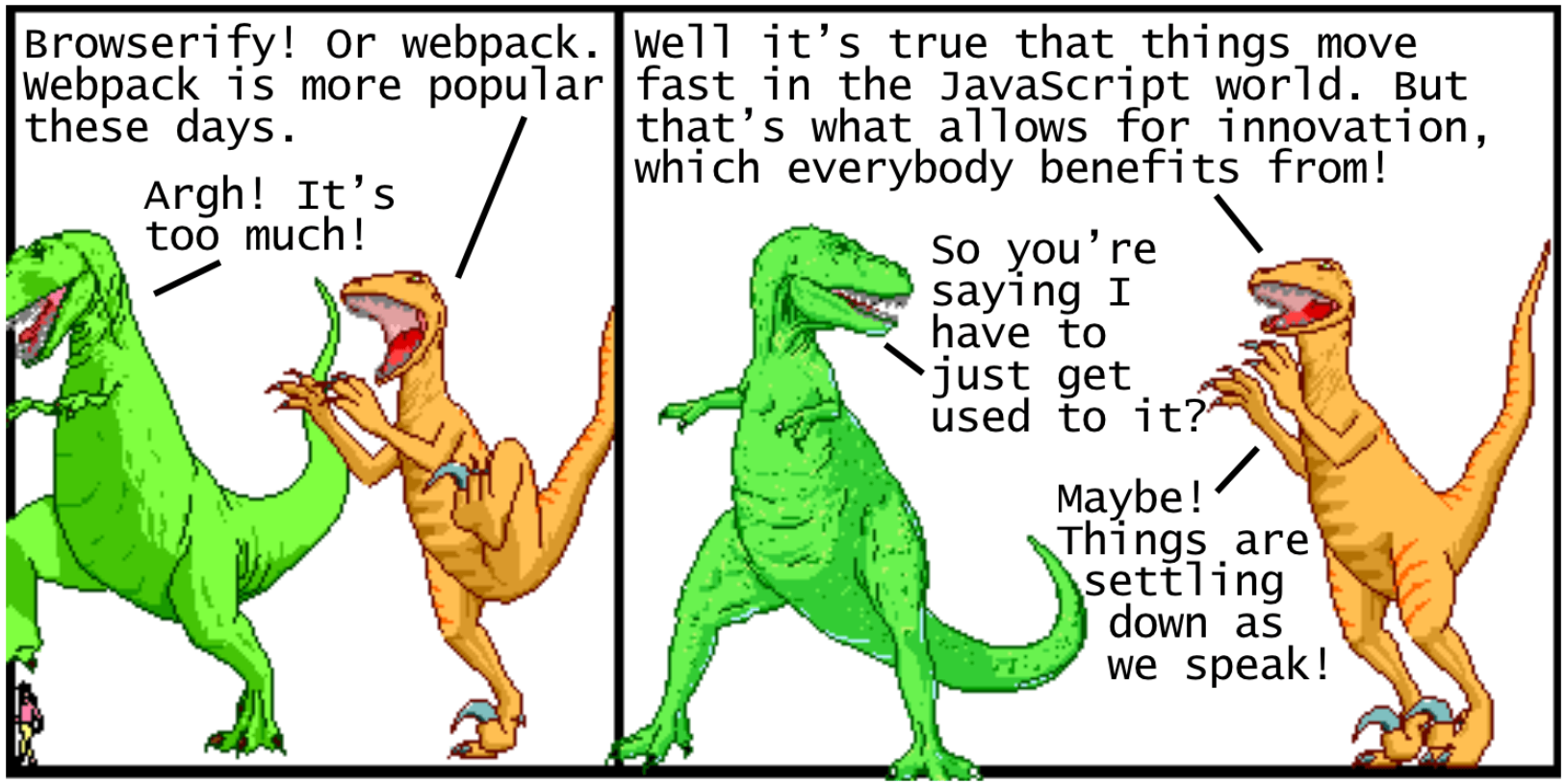
그리고 모듈의 사용법 중 require 함수를 브라우저에서 동일한 코드로 사용 시 오류가 발생한다.
브라우저에는 파일시스템에 접근하는 require 함수가 정의되지 않았기 때문이다.
따라서 이 방식으로는 브라우저에서 모듈을 불러올 수가 없다.

이런 부분들에서 웹팩과 같은 모듈 번들러의 필요성이 재기된다.
이러한 모듈 번들러들은 파일 시스템에 접근이 가능한 빌드 단계에서 브라우저와 호환되는 최종 결과물을 "번들" 파일로 만들어내는 도구이다.

이 덕분에 NPM을 편하게 사용 가능케 끔 하며, require 문법에 대해서도 처리가 가능해진다.

THE JAVASCRIPT MODULES MESS





이제 모듈 번들러는 정식으로 프론트엔드 개발의 워크플로우에 정착된 상황이며 React가 각광받는 이유도 이를 가장 잘 활용하는 프론트엔드 라이브러리이기 때문이다.

이제 모듈 번들러를 통해 프론트엔드에서 빌드 단계가 추가되면서 프론트엔드를 개발자들은 더 강력한 기능들에 대해서 생각하였다.
특히 언어의 새로운 기능들에 대한 지원을 늦장 대응하는 브라우저를 고려하지 않아도 될 수 있다는 점에서 새로운 전환을 맞이한 것.

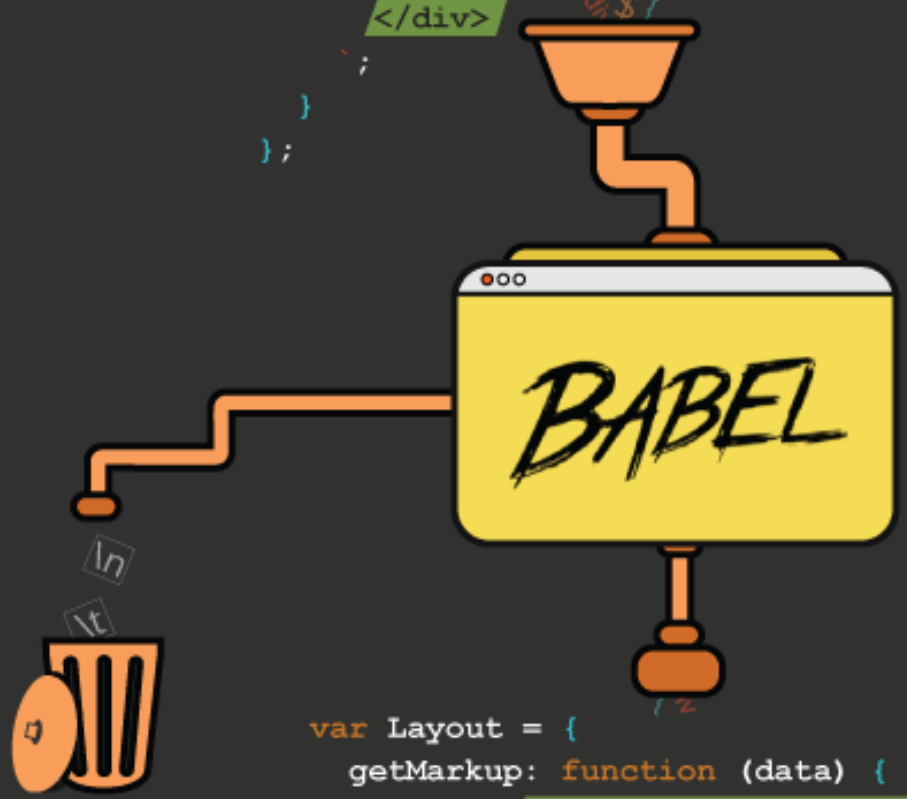
이러한 언어의 새로운 기능을 제공하기 위해서 등장한 것이 트랜스파일러이다.
트랜스파일링은 코드를 다른 코드로 변환한다는 뜻으로 이는 프론트엔드에선 매우 중요하다.

최초의 등장은 자바스크립트의 문법적인 발전을 위해 등장하였지만 점차 시간이 지나며
빠르게 변화해가는 ES 문법을 대부분의 브라우저에서 호환되는 ES5 문법으로 변환시키기 위해서
주로 사용되고 있으며 대표적으로 BABEL이 쓰이고 있다.

Java와 같이 정적 형지정(typing)을 제공하는 타입스크립트도 각광을 받고 있지만
프론트엔드 방면에서는 바닐라JS에 가장 가까운 BABEL이 많은 사람들에게 더 선택받고 있다.

BABEL은 보통 앞서 설명한 웹팩에서 빌드 시에 추가하여 사용하는 것이 일반적이지만
StandAlone 버전을 통하여 단독으로 페이지 로딩 시에도 사용할 수 있다.

```
var Layout = {
  getMarkup: function (data) {
    return nowhitespace`
      <div id="wrapper">
        <div id="cta">
          <a href="${data.href}"></a>
        </div>
      </div>
    `;
  }
};
```




```
var Layout = {
  getMarkup: function (data) {
    return '<div id="wrapper"><div id="cta"><a
href="' + data.href + '"></a></div></div>';
  }
};
```

결론

이것이 모던 자바스크립트다!

기존의 HTML, JS, CSS 개발에서 패키지 매니저를 사용하여 써드 파티 패키지들을 자동으로 관리하고, 모듈 번들러를 사용해 번들화 시킨 자바스크립트 파일을 만들며, 트랜스파일러를 이용해 최신의 자바스크립트 기능들을 사용하며 여기서 더 나아가 태스크러너를 통해 빌드 과정을 자동화까지 하고 있다.

우리에겐 시작조차 하지 못한 이야기이기 때문에 다소 생소할 수 있지만 이 내용이 언급된 것도 벌써 2년도 더 된 이야기이다.

웹 개발이 초보자들에게 REPL적인 성격 덕분에 쉽게 배우고 볼 수 있어서 장점이 있었지만 현재는 반대로 그만큼 다양한 도구와 급변하고 있는 성질 덕에 다소 어려움이 있다고 볼 수 있다.

그러나 현재 그 생태계가 안정화되고 있으며 일관되게 진행되고 있기 때문에 오히려 워크플로우가 편해지고 있다고 생각하는 것이 더 옳다고 볼 수 있다.

오히려 진짜 개발자에겐 흥미롭고 재밌는 요소들이 아닐까?

감사합니다.

- 출처

구닥다리 공룡을 위한 오늘날의 JavaScript
https://steemit.com/javascript/@march23hare/javascript

JavaScript 표준을 위한 움직임: CommonJS와 AMD
https://d2.naver.com/helloworld/12864

Modern JavaScript Explained For Dinosaurs
https://medium.com/the-node-js-collection/modern-javascript-explained-for-dinosaurs-f695e9747b70

Javascript: The module terms and librairies mess
https://bertrandg.github.io/the-javascript-module-mess/

node.js, NPM, CommonJS
https://ko.wikipedia.org/