개발 development

자바스크립트(Javascript)에서 옵저버 패턴(observer pattern)으로 모듈 간 의존성 제거하기

강상민 · 2016. 8. 24. 22:42

자바스크립트로 웹앱을 작성하다 보면 구현된 다수의 모듈들이 서로를 참조하는 구조가 자연스럽게 발생합니다. 이런 구조가 전혀 없을 수는 없지만 모듈 간의 의존성을 최소화시켜야 전체적인 프로그램 구조의 복잡성을 낮추고 전반적인 코드의 품질을 향상시킬 수 있습니다. 이는 꼭 자바스크립트뿐만 아니라 어떤 언어와 환경을 사용하던 한 번씩 되살펴 봐야 하는 부분입니다.

모듈 간의 상호 의존성은 왜 발생할까?

먼저 의존성 자체에 대해 생각해봅시다. 일반적으로 모듈은 특정 기능을 수행하는 단위로 구상/구현됩니다. 최초 모듈 구현시에는 작게 시작되지만 기능이 추가될수록 모듈의 기능 범위가 넓어지게 됩니다. 그러다 보면 모듈에 있어서는 안되는 함수(function)나 프로퍼티(property)가 슬슬 끼어들게 됩니다. 이런 요소들이 발견되었을 때가 바로 리펙토링을 수행해야 할 시점이 됩니다. 리펙토링으로 분리된 모듈은 기존 모듈의 입장에서는 연관성이 덜하여 분리되었지만 전체적인 구조에서는 기존 모듈과 밀접한 관계를 맺게 됩니다. 그만큼 두 모듈의 결합도는 높아지게 되고 상호 참조가 많은 부분에서 발생합니다. 단일 모듈로 구성된 프로그램이 아닌 이상 모듈 간 의존성은 자연스러운 부분입니다.

의존성을 최소화시키려면 어떻게 해야 될까?

의존성을 최소화하기 위해서는 근본적으로 각 세부 모듈들의 범위를 조절하는 것이 가장 중요합니다. 적절히 필요한 단위로 모듈을 구성하여 모듈 간의 인터페이스를 최소화하는 것이죠. 상호 참조가 과하다는 점은 두 모듈이 합쳐질 수 있는 여지가 있다는 신호이기도 합니다. 그럼에도 불구하고 다수의 모듈과 상호 작용이 필요한 경우라면 옵저버 패턴을 사용하여 상호 의존성을 제거할 수 있습니다.

옵저버 패턴(Observer pattern)이란?

옵저버 패턴은 디자인 패턴을 살펴보신 분이라면 아주 익숙한 디자인 패턴 중 하나입니다. 각 모듈의 중간에서 서로의 상태 변화를 관찰하는 관찰자 객체를 만드는 것 입니다. 각 모듈은 서로 알아야 하는 상태변화가 있을 때마다 상대 모듈에게 필요한 데이터와 함께 상태가 변했음을 관찰자(observer)를 통하여 통보(notify) 합니다. 이 통보를 받은 관찰자는 통보한 모듈에서 넘겨준 데이터를 인자로 해당 상태변화와 연관된 각 모듈의 핸들러들을 호출하여 각 모듈이 필요한 작업을 수행할 수 있도록 해줍니다. 물론 각 모듈에서는 어떤 상태변화에 어떤 핸들러가 호출이 되어야 하는지 미리 옵저버에게 등록해두어야 합니다. 글로 적으니 엄청 복잡한 것 같은데 우리에게 익숙한 코드로 세부 내용을 알아보겠습니다.

옵저버(Observer) 구현

```
var observer = {};   // 옵저버 객체를 작성합니다.
```

개발자입니다.


```
  handlers: {}    // 각 모듈에서 등록할 핸들러들을 담아둘 내부 변수를 선언합니다.
};


var observer = {
  handlers: {},

  // 상태 변화 이벤트가 발생하면 실행될 이벤트를 등록하는 함수를 작성합니다.
  // 핸들러 등록시 context를 함께 전달받아
  // 내부에서 this를 사용시 적절한 컨텍스트에서 실행될 수 있도록 합니다.
  register: function (eventName, handler, context) {
    // 해당 이벤트로 기존에 등록된 이벤트들이 있는지 확인합니다.
    var handlerArray = this.handlers[eventName];
    if (undefined === handlerArray) {
        // 신규 이벤트라면 새로운 배열을 할당합니다.
        // 핸들러를 바로 넣지 않고 배열을 할당하는 이유는
        // 한 이벤트에 여러개의 핸들러를 등록할 수 있도록 하기 위함입니다.
        handlerArray = this.handlers[eventName] = [];
    }

    // 전달받은 핸들러와 컨텍스트를 해당 이벤트의 핸들러배열에 추가합니다.
    handlerArray.push({
      handler: handler,
      context: context
    });
  }
};


var observer = {
  handlers: {},
  register: function (eventName, handler, context) {
    var handlerArray = this.handlers[eventName];
    if (undefined === handlerArray) {
        handlerArray = this.handlers[eventName] = [];
    }
    handlerArray.push({ handler: handler, context: context });
  },

  // 등록된 핸들러를 해제하는 함수를 작성합니다.
  unregister: function (eventName, handler, context) {
    var handlerArray = this.handlers[eventName];
    if (undefined === handlerArray)
      return ;

    // 삭제할 핸들러와 컨텍스트를 배열에서 찾습니다.
    for (var hidx = 0; hidx < handlerArray.length; hidx++) {
      var currentHandler = handlerArray[hidx];

      // 찾았다면 배열에서 삭제하고 함수를 종료합니다.
      if (handler === currentHandler['handler']
       && context === currentHandler['context']) {
        handlerArray.splice(hidx, 1);
```

```
    }
  }
};


var observer = {
  handlers: {},
  register: function (eventName, handler) {
    var handlerArray = this.handlers[eventName];
    if (undefined === handlerArray) {
        handlerArray = this.handlers[eventName] = [];
    }
    handlerArray.push({ handler: handler, context: context });
  },
  unregister: function (eventName, handler, context) {
    var handlerArray = this.handlers[eventName];
    if (undefined === handlerArray)
      return ;
    for (var hidx = 0; hidx < handlerArray.length; hidx++) {
      var currentHandler = handlerArray[hidx];
      if (handler === currentHandler['handler']
       && context === currentHandler['context']) {
        handlerArray.splice(hidx, 1);
        return ;
      }
    }
  }

  // 특정 상태가 변했을때 이벤트를 통보할 함수를 작성합니다.
  notify: function (evnetName, data) {
    // 통보된 이벤트에 등록된 핸들러가 있는지 확인합니다.
    var handlerArray = this.handlers[eventName];
    if (undefined === handlerArray)    // 없다면 함수를 리턴하여 종료합니다.
      return;

    // 핸들러 배열에 등록되어있는 핸들러들을 하나씩 꺼내 전달받은 데이터와 함께 호출합니다.
    for (var hidx = 0; hidx < handlerArray.length; hidx++) {
      var currentHandler = handlerArray[hidx];
      currentHandler['handler'].call(currentHandler['context'], data);
      // 전달받은 함수를 바로 호출하지 않고 call을 사용하여 호출하는 이유는
      // 미리 등록시 함께 전달된 context 객체를 함수내부에서 this로 사용할 수 있게끔
      // 함수 내부로 전달하기 위함입니다.
      // 자바스크립트에서 this를 사용할때는 상당히 주의해야 합니다.
    }
  }
};
```

각 모듈의 상태변화와 데이터를 주고받을 수 있는 observer 객체가 만들어졌습니다. 이제 실제 상황에서 어떻게 활용할 수 있는지 살펴보겠습니다.

옵저버(Observer) 활용

```
var manager = new Person();       // 팀장도 있고,
var programmer = new Person();    // 개발자도 있습니다.

// 서로 연관성을 한번 만들어 보겠습니다.

// 사장님이 말씀을 하십니다.
boss.speak = function(comment) { // 훈하 말씀을 담아서
  alert(comment);
  observer.notify("bossSpeak", comment);    // '사장이 말한다' 이벤트를 발생시킵니다.
};

// 이제 사장님 이하 직원들이 이야기를 새겨 들어야겠죠?
manager.listen = function (comment) {
  this.bossComment = comment;
  // 팀장님은 팀장님 답게 사장님의 훈하말씀을 본인의 마인드에 새겨놓습니다.

  // * call을 사용하여 context가 넘겨지지 않았다면 이 부분에서 this가
  // manager를 지칭할 것이라는 보장이 없게 됩니다.(자바스크립트의 특징으로 주의해야합니다.)
  // 그래서 notify에서 call을 사용하여 핸들러를 실행시키고
  // 첫번째 인자인 context로 manager를 넘겨주는 것입니다.
  // 그렇게 넘겨진 manager는 함수내에서 this에 할당되어 접근할 수 있게 됩니다.
};
observer.register("bossSpeak", manager.listen, manager);
// 옵저버에 등록하여 언제든 사장님의 말씀을 새겨들을 준비를 합니다.

// 자.. 그럼 우리 개발자는?
programmer.drop = function (comment) {    // 한귀로 듣고
  return comment;  // 한귀로 흘려 봅시다.
};
observer.register("bossSpeak", programmer.drop, programmer);
// 무념무상이 준비되었습니다.

boss.speak("... for an hour ...");    // 이제 사장님이 뭐라고 (한시간 동안)말씀을 하시면,
// 옵저버에 등록한 대로 자동으로
// manager.listen("... for an hour ...");     // 팀장님은 새겨듣고,
// programmer.drop("... for an hour ...");    // 개발자는 흘려 듣게 됩니다.
```

구현상 크게 어려운 부분은 없습니다. call을 사용하여 핸들러들을 호출하는 부분만 주의해주시면 됩니다. 옵저버 패턴을 이용하면 각각의 모듈이 어떤 상관관계를 갖는지 명시적으로 표현할 수 있습니다.(console.log(observer.handlers)로 살펴볼 수도 있습니다.) 한 모듈 안에서 여러 다른 모듈의 함수를 호출하게 되면 우리가 흔히 말하는 스파게티 코드를 만들어 내게 됩니다. 예제의 boss.speak 함수에서 다른 두 함수의 manager.listen과 programmer.drop을 호출한다고 생각해보세요. 물론 그렇게 만들어도 동작은 하겠지만, 중간에 manager 객체가 빠진다고 생각해보면 옵저버를 사용하지 않은 경우에 manager.listen을 호출하는 boss.speak 함수까지 확인하여 관련 부분을 제거해 주어야 하지만 옵저버를 사용해서 구현됐다면 그저 manager와 그에 관련된 옵저버 등록 부분만 제거해주면 다른 boss나 programmer 객체에는 어떠한 수정도 가해지지 않습니다. 이처럼 옵저버 패턴을 사용하면 모듈 간(객체 간) 연관성은 유지하면서 소스 코드 상의 상호 의존성은 제거하는 좀 더 간결한 구조를 만들어 낼 수 있습니다.

맺음

개발을 하다 보면 일정상의 이유로 혹은 다른 수없이 많은 이유로 깔끔하지 못한 코드를 작성하게 되는 경우가 많습니다. 이는 나중에 언젠가는 갚아야 할 부채로 (우리 마음속에) 남아 언젠가는 우리를 괴롭히게 됩니다. 옵저버 패턴은 이런 부채를 탕감할 수 있는 손

개발자입니다.

이 블로그에서 검색

#자바스크립트 #javascript #옵저버 #observer #패턴 #pattern #의존성 #dependence #제거 #removal

공감 1 | 댓글 7 | 인쇄

이 블로그 개발 development 카테고리 글 전체글 보기

이전 다음 ▲TOP

강상민
(c_ist82)
개발자 입니다.
프로필 쪽지

카테고리
전체보기 (19)
사색 (7)
개발 development (7)
iOS (5)

태그 최근 | 인기
ios , api , Auto , celery , django , drive , google , Layout , 개발 , 구글 , 드라이브 , 레이아웃 , 셀러리 , 스택 , 오토
▶모두보기