Five Patterns to Help You Tame Asynchronous JavaScript

개요

- 당신이 비동기 동작에 직면하기 전에는 자바스크립트 10줄도 작성하지 못한다.
- 우리는 야수를 길들이는데(또는 방지하는데?) 도움이 되는 5가지 패턴에 대해 살펴볼 것이다.
- 각 패턴을 완벽히 커버하기보단 충분한 정보를 제공하는것이 목표이다.
- 사실 모든 패턴들은 동기화 컨텍스트에서 잘 작동한다(not only asynchronous).

'비동기'가 무엇인가? 속성 입문..

- JavaScript는 단일 스레드상에서 Event Loop를 사용
  - Event Loop : 동작을 요청한뒤에 콜백을 지정하여 동작이 완료되면 콜백이 실행되는 방식
- 현실은 모든 자바스크립트가 동기적으로 실행
  - 이벤트 루프는 루프가 작업 대기중인 코드의 실행이 완료된 후에 사용할 수 있을때까지 발생하지 않은 작업을 대기열에 넣는것을 허용
  - it's the event loop that allows you to queue up an action that won't take place until the loop is available some time after the code that queued the action has finished executing.
  - 그래서 이벤트 루프가 사용가능해진 후 언젠가 실행하기 위해 대기 중일 때 코드가 비동기적으로 실행할 수 있다고 한다.

> "Events can be queued while code is running, but they can't fire until the runtime is free." - Trevor Burnham

- 예제

```
function maybe() {
    console.log("…execute async, maybe?");
}

function proveIt() {
    setTimeout(maybe, 0);
    console.log("Hey, you just invoked me, and this is crazy…");
    console.log("But I'll queue you up");
    return "and you'll…";
}

proveIt();
```

- 결과

```
Hey, you just invoked me, and this is crazy…
But I'll queue you up
"and you'll…"
…execute async, maybe?
```

    - 이벤트 루프가 사용가능해진 0 밀리초 후에 실행할 maybe 함수를 대기열에 넣기 위해 setTimeout 함수를 사용
    - 이벤트 루프는 현재 실행 스택을 완료할 때까지 사용할 수 없다.
    - proveIt함수가 리턴하는 즉시, 런타임은 해제되고 대기중인 maybe함수 실행 가능
    - proveIt이 완료된 후, 대기중인 maybe 함수의 출력이 나타난다.

- setTimeout함수가 처음이거나, Javascript에 익숙치 않은 경우를 대비하여, 두개의 비동기 'primitives'는 모든 주요 런타임에 걸쳐 존재:
    - setTimeout(fn, x) : 함수 f가 x ms후에 호출(실제 지연시간이 더 길어질수 있음)
    - setInterval(fn, x) : x ms마다 함수 fn를 반복적으로 호출
- 일부환경에선 추가적인 비동기 기본형식을 제공
    - node.js는 process.nextTick이 제공
    - 많은 브라우저가 requestAnimationFrame을 지원
- more info
    - setTimeout 및 setInterval에 대한 MDN 기사 참고

Callbacks

- 콜백 == 흐름 == 기본적인 빌딩블록 == 비동기성의 핵심 역할
- ~~랭선이 여기저 어디로 가든, 높은 수준의 추상화는 단지 평범한 콜백을 감싸는 래퍼이다.~~
- 자바스크립트 입문자에겐 함수가 인수로 전달되는 것이 조금 생소할 수 있다.

```
// plain, non-jQuery version of hooking up an event handler
var clickity = document.getElementById("clickity");
clickity.addEventListener("click", function (e) {
    //console log, since it's like ALL real world scenarios, amirite?
    console.log("Alas, someone is pressing my buttons…");
});

// the obligatory jQuery version
$("#clickity").on("click", function (e) {
    console.log("Alas, someone is pressing my buttons…");
});
```

- addEventListener와 on 모두 콜백 인수를 취한다.
- 두번째 인자로 전달
- click 이벤트 발생 시 함수 호출
- DOM 엘리먼트(아마도 버튼)는 우리가 전달하고 이벤트 발생한 후에 호출할 콜백을 저장하는 기본기능을 가진다 점을 활용(옵저버 패턴의 구현).

```
var worker = {
    updateCustomer: function (customerInfo, cb) {
        $.ajax({
            url: "/some/api",
            type: "POST",
            data: customerInfo
        }).done(function (data) {
            // let's pretend there's some DOM to update…
            $("#update-status").removeClass("editing").addClass("saved");
            cb(null, data);
        }).fail(function (data) {
            // let's pretend there's some DOM to update…
            $("#update-status").removeClass("editing").addClass("failed");
            cb(data);
        });
    }
    // other methods, properties, etc
};

// and somewhere else, the calling code:
// (assuming we already have a currentCustomer instance, etc)
worker.updateCustomer(currentCustomer, function (err, data) {
    alert(err || data); // *sigh*, who really uses alerts these days?!
});
```

- 위의 worker 인스턴스에 몇가지 문제점이 있지만 일단 pass(it's tightly coupling AJAX & DOM updates, for one).
- 자바스크립트에서 함수는 first class citizen이다. => 다른 함수에 인수로 전달 가능
- updateCustomer 메소드는 두개의 인수를 사용
- cb는 콜백 함수로 updateCustomer가 실행 완료되면 호출 가능
- 이것은 콜백을 전달하고, 콜백 인수를 사용하는 메서드를 쓰는 것의 핵심 - it doesn't get much lower-level than this.
- "continuation passing style(CPS) : 콜백 == "continuation"
    - CPS : 호출의 끝에 원래의 context로 돌아가지 않고, 새로이 불릴 함수를 caller가 넘겨주는 프로그래밍 스타일
- 나는 콜백함수의 "context"를 지정하고자 하는 상황이 종종 실행되는 것을 언급해야 한다. (즉, what this will be inside the function when invoked by the method to which you are passing it)
- (대부분의 경우)함수를 인수로 전달하기 이전에 함수를 바인딩

```
// assuming we have a notifier instance, with a "showAlert" method on it...

// ES5 bind function (or using a shim)
worker.updateCustomer(currentCustomer, function(err, data) {
    this.showAlert(err || data);
}.bind(notifier));

// using underscore/lodash
worker.updateCustomer(currentCustomer, _.bind(function(err, data) {
    this.showAlert(err || data);
}, notifier));


// using jquery
worker.updateCustomer(currentCustomer, $.proxy(function(err, data) {
    this.showAlert(err || data);
}, notifier));
```

- 위의 예제에서, 우리는 notifier 인스턴스가 될 콜백의 "this"컨텍스트를 바인딩

- 콜백 함수 내의 this.showAlert는 notifier.showAlert를 호출하는 액세스 권한을 가진 것과 동일. 그런데 뭐 어쩌라고?
- 우리가 콜백을 전달하는 메소드는 그것을 호출하여 개체의 지식을 갖고 있지 않은 인스턴스에 속한다 할 수있다. 잉?
- 이것을 관심사의 분리 & 캡슐화라고 부른다.
- 그러나, 우리가 콜백(실제로 호출하는 객체 상의 메소드)을 전달하는 경우, 우리는 컨텍스트를 바인드 해야할 것이다.

```
var updateForm = {
    submit: function() {
        // get the data and store it in currentCustomer
        worker.updateCustomer(currentCustomer, this.showAlert.bind(this));
    },
    showAlert: function(err, data) {
        // I don't care how, just show an alert :-)
    }
};
```

- more info : "Getting Into Context Binds"

callback conclusion

- 장점
    - 간단하다 : 콜백은 간단하다(주의할 점은 당신이 필요한 경우 올바른 컨텍스트를 유지할 책임이 있다는 것). 미래의 어떤 시점에서 호출되는 함수를 전달
    - 가볍다 : 별도의 라이브러리가 필요없다. 함수로써의 first-class citizen은 언어에 내장. 작동을 위한 별도의 추가코드가 필요 없다.
- 단점
    - 불충분한 추상화를 할수있다. - 때때로 당신은 추가적인 'sugar'를 필요로 한다. 이곳에서 각기 다른 방법으로 콜백의 기초를 기반으로 하여 2-5차례 발생하게 될 것이다.
    - 중첩시 복잡함 - 콜백은 깊이 중첩될수록 읽거나, 디버그, 관리 하기가 어렵다.

the pyramid of doom

```
var allTheCustomerThings;
$("#getCustomer").click(function (cust) {
    var id = $("#cust-id").val();
    getCustomer(id, function (cust) {
        allTheCustomerThings = cust;
        getContacts(id, function (contacts) {
            allTheCustomerThings.contacts = contacts;
            getOrders(id, function (orders) {
                allTheCustomerThings.orders = orders;
                getAccountsRecv(id, function (ar) {
                    allTheCustomerThings.ar = ar;
                    // OK - we got all the data, NOW WHAT?! :-)
                });
            });
        });
    });
});
```

- Javascript가 처음인 개발자들은 자신의 응용프로그램을 제대로 계획하고 구조화하는데에 많은 시간을 할애하지 않는다.
- 그것은 종종 맹목적으로 Promise을 사용하기 위해 반사적인 이유를 예로 들고 있다는 점에서, 허수아비 공략법이다.
- 적절한 주의없이, Promise가 중첩을 완화하는데 사용되는 경우에도 마찬가지이다.(물론, Promise를 사용하는 합법적이고 설득력있는 이유가 있다. But I can just as easily slap promises on a terribly written code base only to eliminate nesting.)
- 낮은 수준의 패턴은 높은 수준의 디자인 문제를 해결하지 못한다 - 그래서 당신은 깊은 중첩 콜백으로 끝날 경우 코드에서 문제를 모델링하는 방법을 통해 생각해야 한다.
- and consider it both a code and design smell that should prompt you to address the systemic issue(s) with the overall design and not just the nesting strategy.

Observer Pattern (a.k.a. Events)

- 발행/구독 패턴
- subject라 불리는 객체(object)가 observer라 불리는 객체들의 리스트를 관리하고, 상태변화가 있으면 알아서 옵저버 객체들에게 알려주는(종종 옵저버의 메서드 methods를 호출함으로써 알려주는) 디자인 패턴(software design pattern)
- 1:n 의 조건을 성립, 주제 객체가 존재해야 한다.
    - 한 객체의 상태가 바뀌면 그 객체에 의존하는 다른 객체들한테 연락이 가고 자동으로 내용이 갱신되는 방식으로 일대다(one-to-many) 의존성을 정의
- 객체간의 느슨한 결합이 이루어져야 한다.
- 객체의 상태 변화를 관찰하는 관찰자(옵져버)를 객체에 등록(구독)하여, 객체에 상태 변화가 있을때마다 등록된 목록의 관찰자들에게 통지(발행)
- 모듈간의 밀접한 커플링을 조율하여, 모듈의 재사용성을 높인다.
- 옵저버 패턴이 필요한 경우
    - 만약 우리가 타겟 메소드 호출 시마다 콜백 전달을 유지하기를 원하지 않는다면 어떻게 될까?
    - 만약 우리가 호출 코드의 다른 부분도 타겟 메서드를 호출한 경우 호출할 자신의 콜백을 원한다면 어떻게 할까?
- 당신은 이벤트를 발하는 대상을 갖고 있다("event emitter")
- 그리고 옵저버는 이벤트가 언제 발생하는지에 대해 통보할 대상과 함께 콜백을 등록

- Doug Neiner와 필자가 Halo 4를 한다고 가정해보자.
    - Alex Robson이 내게 Doug가 언제 스틸질을 했는지 알고 싶다고 말한다.
    - 이 때, 나는 대상이 되고, Alex는 옵저버가 된다.
    - 불행하게도, Alex는 수많은 이벤트 알림을 받을 것이기 때문에 자신의 콜백을 debounce(지연?)하길 원할 것이다.

- 염두에 두어야 할 중요사항
    - 옵저버는 대상에 대한 직접 참조가 있어야 하다.
    - 대상은 가입자 콜백의 내부 상태를 유지 관리할 책임이 있다.
    - 이 패턴의 자바스크립트 구현은 일반적으로 0-n 인수의 가입자 콜백 메서드 Signature을 포함 (즉, 모든 이벤트는 다른 Signature을 가질 수 있다.)

- Halo 4를 예를 들어 설명

```
// This observer object can be mixed into any object, giving it the basic
// API necessary to add and remove subscribers as well as emit events
var observer = {
    // 'subscribers' will keep track of subscribers by event name
    // each event name subscribed to will be a member name on
    // this object, w/ the value as an array of objects containing
    // the subscriber callback and optional function context
    subscribers: {},

    // the 'on' method is used by subscribers to add a callback
    // to be invoked when a specific event is emitted
    on: function (event, cb, context) {
        this.subscribers[event] = this.subscribers[event] || [];
        this.subscribers[event].push({
            callback: cb,
            context: context
        });
    },

    // 'off' allows subscribers to remove their callbacks
    off: function (event, cb, context) {
        var idx, subs, sub;
        if ((subs = this.subscribers[event])) {
            idx = subs.length - 1;
            while (idx >= 0) {
                sub = subs[event][idx];
                if (sub.callback === cb && (!context || sub.context === context)) {
                    subs[event].splice(idx, 1);
                    break;
                }
                idx--;
            }
```

```
        }
    },

    // iterates over the subscriber list for
    // a given event and invokes the callbacks
    emit: function (event) {
        var subs, idx = 0,
            args = Array.prototype.slice.call(arguments, 1);
        if ((subs = this.subscribers[event])) {
            while (idx < subs.length) {
                sub = subs[idx];
                sub.callback.apply(sub.context || this, args);
                idx++;
            }
        }
    }
};
// We're using jQuery's extend function to copy the observer
// object's members over a new object, creating the "jim" instance
var jim = $.extend({
    dangItDoug: function (numberStolen) {
        this.emit("stolenkill", numberStolen);
    }
}, observer);

var alex = {
    tauntJim: function(numStolen, dt) {
        $('body').append("<div>Jim, your incompetence has cost you " +
            numStolen + (numStolen > 1 ? " kills" : " kill") + ".</div>");
    }
};
// the "jim" instance is a subject, and we're subscribing to it, to
// have alex.tauntJim invoked any time the 'stolenkill' event occurs
jim.on("stolenkill", alex.tauntJim);
var i = 0;
jim.dangItDoug(++i);
jim.dangItDoug(++i);
```

```
jim.dangItDoug(++i);
jim.dangItDoug(++i);
```

- 많은 인기있는 라이브러리들은 이미 이벤트-발생 동작기능이 내장되어 제공(ex. backbone.js & jQuery)

Event Conclusion

- 장점
  - Encourages Good De-coupling
    - Observer 패턴을 사용하면 응용 프로그램의 동작을 나누는데 도움이 될 것
    - 잘 캡슐화된 구성요소에 연결되고, 일반적으로 설계 결과는 깊게 중첩된 콜백의 함수에 대한 저항력이 강해진다.
  - Lends Itself to Testability
    - 이 접근방식과 함께 디커플링을 향한 편견으로 인해, 대부분의 경우 별개로 테스트 가능하고 쉽게 모방할 수 있는 컴포넌트를 끝낸다.
- 단점
  - Direct Reference Required
    - 이것은 event-emitter에 아킬레스 건
    - 당신이 대부분의 응용프그램의 대부분이 흥미로운 주제를 가지고 있다면, 당신은 어디서나 그것에 대한 참조를 전달
    - This begins to undermine any 'loose coupling' gains you've racked up, creating a kind of tight coupling that can be harder to detect until you have to change the subject's API and break most of your app in the process.

Elephant, Meet Room

그럼 이제 지금까지 회피해왔던 것들을 해결해보자

- 많은 유명한 라이브러리(jQuery같은)들은 Observer Pattern의 구현 모습을 제공. 그것들은 관찰자와 주제 사이의 중재자 역할 수행
- 이것은 문제가 되지 않는다. 그저 어떻게 작동하는지 알고 있으면 된 것이다.
- 이러한 구현의 대부분은 event-emitting 동작, API가 프로토타입이라는 의도, 또는 인스턴스 멤버로 혼합 등을 제공
- 그러나 많은 개발자는 이 event-emitting 라이브러리를 가지고 단일 인스턴스를 생성하고 일반화 "emitter"로 이 인스턴스를 사용
- 그래서 더이상 대상은 on/off/등의 호출 자체가 없다.

적용 예

- jQuery에서 '.on()'(이벤트 핸들러 생성), 'trigger()'(이벤트 발생), 'off()'(이벤트 핸들러 제거)를 사용
- 실제 메모리에 DOM 요소를 올려서 사용
- 이벤트 핸들러는 DOM 구조에 붙였다가 DOM의 해당 요소가 삭제되면 자동적으로 이벤트도 삭제

Messaging

> Jonathan Creamer는 Alex가 친구들이 Halo 4를 하고 있는것을 지켜보면서,
> 어떻게 진행되는 것인지 듣고싶어 한다는것을 알아냈다고 밝혔다.
> 유일한 문제점은 그가 우리집에 오는 방법을 몰랐고, 사실 그가 집에서 나오기 싫어했다.
> 그래서 그는 Alex에게 전화했고 우리가 이기거나 질때마다, 게임을 시작할때,
> 메달을 얻었을때 알려달라고 요청했다. 알렉스는 이제 중개자 혹은 브로커 역할을 한다.
> 조나단은 나(Jim Cowart)나 Doug에게 바로 설명하지 않고, 알렉스에게 그가 관심있어 하는 것에 대해 말해줬다.
> 그리고 알렉스가 다시 조나단에게 모든 관련 정보를 전달한다.
> 해당 정보의 출처가 어디든 말이다. 이것이 자바스크립트 "메세징"의 핵심이다.

- 이벤트 발생에 관심있는 옵저버를 가졌다는 점에서 옵저버 패턴과 유사
- 그러나 우리는 구독관리를 처리하고 올바른 구독자에게 이벤트를 매칭시켜주는 서드파티를 소개했다(message).
- 여기에 적용하기에 가장 좋은 패턴 이름에 대한 약간의 논쟁이 있었다.
- 그러나 여기에 합법적으로 주장할 수 있는 몇 가지 설명이 다음과 같이 되어있다:
  - Mediator : 객체 간의 통신은 중재자 내에서 캡슐화. 객체는 더 이상 서로 직접 통신하지 않고 대신 중재자를 통해 통신
  - Event Aggregator : 단일 객체에 여러 객체의 채널 이벤트는 클라이언트에 대한 등록을 단순화
  - (Client-side) Message Broker : 여러 대상에서 메세지를 수신, 정확한 목적지를 결정&메세지 발송

실제 사용되는 패턴에 상관없이, 사람들은 여전히 "PubSub(Publish/Subscribe)"라고 부른다.

- 우리가 완전한 서버측 메시지 브로커를 다시 만들려고하지 않는 것을 인식하는 것이 중요

- 대신,이 방법은 옵저버 패턴으로 나타날 수있는 약점을 수용한다.
- 특히 대규모 응용 프로그램은 다른 일반적인 모든 subscriber 및 사용할 publisher에 대한 종속성이 아니라 대상에 대한 직접 참조는 모든 응용 프로그램에 전달될 필요가 있다.
- 자바 스크립트 메시지 버스 구현의 몇 가지 일반적인 기능
    - "페이로드" 메세지는 콜백에 인수를 전달한다기 보다는 구조화된 봉투?를 사용한다고 할수 있다.
        - 이 봉투는 메시지 데이터 뿐만 아니라 데이터 메시지에 대한 기타 메타 데이터를 포함한다.(이벤트명, 채널명, 타임스탬프 등)
        - 메세지에 Including 동작은 비추한다. 이상적으로, JSON.stringify에 전달하는 경우, 전체 메시지 봉투는 보존되어야한다.
    - 가입자 콜백은 일관된 서명을 가지고 있다.
        - 위의 특징 덕분에 가입자 콜백은 일반적으로 1~2인자를 받는 메소드 시그니처(메소드의 특성과 메소드 이름, 파라미터, 반환값의 테이터타입을 가진 형태 - 모든 메세지에 대해 일관적임)로 끝난다.
    - Where the Observer pattern might have "event" names, messages in JavaScript usually have "topics" that can often be hierarchical (or namespaced). This opens the door for wildcard bindings - we'll see that in a moment.
    - 일부 구현에서는 항목이 '채널'로 인해 분할될 수 있다. - 항목의 논리적 그룹핑 - 성능 및 구조상의 이유로 말이다.

Why Would I Choose This Over Events?

postal.js - 자바스크립트 메세지 버스 구현

```
var jonathan = (function() {
    // postal allows you to group topics by channel.
    // Here we get a channel for "xbox" related topics.
    // From it we can publish and subscribe.
    var xboxChannel = postal.channel("xbox");
    var reactions = {
        newgame: function(data, env) {
            return "I hope this goes better than the last one did!";
        },
        newmedal: function(data, env) {
            return "Nice, a new medal - about time!";
        },
        lostgame: function(data, env) {
            return "Ouch, you got pwned!";
        },
        wongame: function(data, env) {
            return "Stroke of luck, perhaps?";
        }
    };
    var module = {
        handleMessage: function(data, env) {
            // if we have a reaction for this kind
            // of message we'll call makeComment.
            if(reactions[env.topic]) {
                this.makeComment(reactions[env.topic](data, env));
            }
        },
        makeComment: function(msg) {
            $('body').append("<div>" + msg + "</div>");
        }
    };

    // we are subscribing using the "#" wildcard
    // which would match ANY topic of any length
    // (postal follows the AMQP binding approach)
    // notice we're using a 'fluent' configuration
    // method to provide the function context we
    // want to be present (the value of "this")
    // in our subscriber callback
    xboxChannel
        .subscribe("#", module.handleMessage)
        .withContext(module);

    return module;
}());

var chan = postal.channel("xbox");
// we're briding a socket.io connection to our message bus
var socket = io.connect(location.origin);
_.each(['newgame', 'newmedal', 'lostgame', 'wongame'], function(evnt){
    socket.on(evnt, function(data){
        chan.publish(evnt, data);
    });
});
```

- 위 예제는 그저 메세징 맛보기에 불과하다.
- 우리가 newgame, newmedal, lostgame 및 wongame 이벤트를 등록하고있는 동안 웹소켓에 연결된다.

- 우리가 그 웹소켓 이벤트를 처리한 것처럼, 우리는 로컬 메시지 버스에 수신된 데이터를 공개하고, 모든 가입자는 웹소켓 연결에 대해 아무것도 알 필요 없고, 동일한 데이터에 어떤 다른 가입자를 몰라도 메세지를 받을 것이다.
- This kind of loose coupling lends itself to good testability, portability (across boundaries such as iframes and Web Workers) and composability.
- 메세징이 모듈 간 통신 접착제의 역할을 하는 경우, 메세지는 계약이다.
- 계약을 존중하는 한 당신은 임의의 수의 드롭 인 (drop-in) 모듈의 응용 프로그램을 작성할 수 있다.
- 모듈은 그들이 필요로하는 주제를 등록하고 게시 내용에 대한 메시지 계약을 준수한다.

Messaging Conclusion

- 장점
  - Promotes clean separation of concerns: 대형 어플을 구축할 필요 없다. 대신, 여러 개의 작은 것들을 구축하고 그들 사이의 통신을 위해 메시징를 사용한다.
  - Very testable: 메세지를 속이거나 메시지 출력을 테스트 하기 매우 쉽다.
  - Complements events nicely: 옵저버 패턴 '이벤트'는 로컬 문제에 대해 잘 작동한다. 메세징은 이벤트가 응용프로그램 수준의 메세지로 승격해야 할 경우 이를 보완하는 것이다.
- 단점
  - Can be prone to "boilerplate proliferation": 응용프로그램 특정 헬퍼를 작성함으로써 완화할 수 있다.
  - Can be confusing to devs who are new to the concept in general: 설명하고 가르치기 위해 적절한 시간이 걸린다.

Promises

```
// fictional viewmodel for a mobile login screen
// the ugly nested callback version
var loginViewModel = (function () {
  var login = function () {
    var username = $('#loginUsername').val();
    var password = $('#loginPassword').val();
    el.Users.login(
      username,
      password,
      function () {
        usersModel.load(
          function () {
            mobileApp.navigate(
              'views/notesView.html',
              function () {
                // YAY! We made it!
              },
              function (err) {
                showError(err.message);
              });
          },
          function (err) {
            showError(err.message);
          });
      },
      function (err) {
        showError(err.message);
      });
  };
}());
```

```
// Converted to use promises
var loginViewModel = (function () {
  var login = function () {
    var username = $('#loginUsername').val();
    var password = $('#loginPassword').val();
    el.Users.login(username, password)
      .then(function () {
        return usersModel.load();
      })
      .then(function () {
        mobileApp.navigate('views/notesView.html');
      })
      .then(
        null, // YAY! We made it!
        function (err) {
          showError(err.message);
        }
    );
  };
  return {
    login: login
  };
}());
```

Promises Conclusion

- 장점
  - 중첩&흐름의 복잡성 감소: 동기화 스타일에서 비동기 코드의 특정 측면을 표현할 수있는 것은 거의 모든 수준에서 개발자가 코드 섹션이 무엇을하고 있는지 즉시 이해하는데 도움
  - 결과는 "캐시"가 될수 있다: 해결된 promise를 유지해야 하는경우, 추가적인 성공 핸들러는 promise를 추가한다. promise는 결과값을 호출하고 전달하는 역할을 수행한다. 일부 상용구를 저장할 수 있을 뿐 아니라 불필요한 작업을 방지할 수 있다.(예를 들어, 중복된 ajax 요청과 같이)
- 단점
  - 결과는 "캐시"가 될수 있다: 이것은 양날의 검이다. promise 인스턴스를 유지하는 경우, 자주 변경되는 값에 대해 이렇게 하지 말아라.
- Opinionated on OSS APIs: 당신이 일반적인 소비를 위한 라이브러리를 작성하는 오픈소스 저자라면, 사양-호환 라이브러리를 사용해라! 그렇지 않는다면, 당신은 개발자들에게 매우 독단적인 종속성을 강요하는 것이다.
- Many Non-Spec-Adhering Implementations Exist: 다음 중 하나를 선택하면 과거의 것을 얻을 수 있다 :
  Q
  when.js
  RSVP.js

One More Thought

- promise는 콜백의 유일한 대안이 아니다.
- 경험상, promise는 동작의 단위를 간결히 하기 위한 강력한 도구라는 것을 알았다.
- 그러나, promise가 오래 실행되는 워크플로우에 사용되는 것을 보았다.
- FSM이 이런경우 더 적합한 작업이다.
- 또한 뷰 사이의 높은 수준의 UI를 관리엔 뭔가 더 반응성이 좋은(events, messaging) 것이 선택하는 것이 좋다.

Finite State Machines

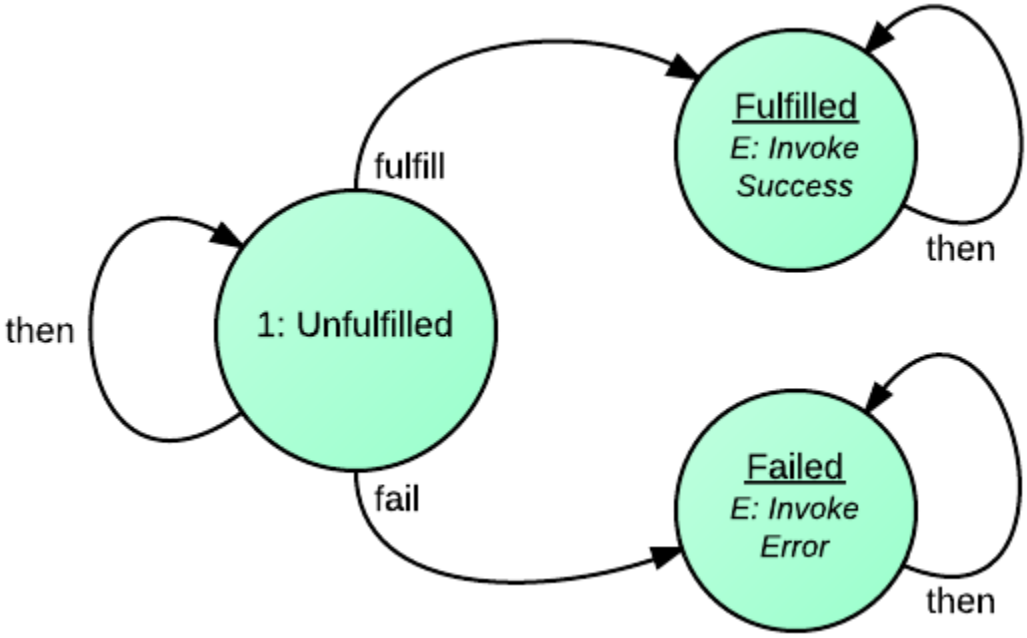
- 유한 개의 상태로 구성된 프로그램
- 때때로 동일한 입력에 다르게 반응할 수 있는 추상화가 필요
- However, once they are authenticated, the same attempts to navigate should actually take them to the desired view(s), etc.
- Finite State Machine은 간결하고 확장 가능한 방식으로 동작의 종류를 관리
- FSM은 "계산 추상화"이다 - 즉, 그것은 우리가 질문에 yes/no 대답을 계산하기 위해서 사용하는 것이다. - 그것은 내부 상태의 변화에 따라, 동일한 입력에 대해 다르게 반응 할 수 있다.
- FSM의 가장 기본은 간단한 스위치 문으로 생성 할 수 있다.

```
var fsmLite = (function(){
    var _state = "notReady";
    var _result;
    // We're watching a subject somewhere to know
    // when we can move to ready state
    someSubject.once("ready", function() {
        _state = "ready";
    });
    return {
        doSomething: function(callback) {
            switch(_state) {
                // if someone wants us to "doSomething"
                // but we aren't ready, we invoke their
                // callback and pass an error message.
                // Other strategies are applicable here
                // as well - such as queueing up any
                // callbacks to be invoked once we reach
                // the "done" state, etc.
                case "notReady":
                    callback("Not ready yet!", null);
                break;
                // we actually do the work
                case "ready":
                    worker.doWork(function(result) {
                        // Do other work, etc. and then:
                        _result = result;
                        _state = "done";
                        callback(null, _result);
                    });
                break;
                // The work has already been done, we just
                // invoke the callback and pass the result
                case "done":
                    callback(null, _result);
                break;
                default:
                    console.log("Invalid state for FSM!");
                break;
            }
        }
    };
}());
```

- 위의 예제에서 _state 값에 따라 동일한 입력(doSomething)에 대해 다른게 응답한다.
- 그것은 또한 몇 가지 다른 FSM 특성을 보여준다 :
    - 존재할 수 있는 상태의 수가 제한적이다.
    - 주어진 시간에 하나의 상태만 존재할 수 있다.
    - 입력을 받아들인다.
    - 출력을 생성할 수 있다.(예제에선 _result 값을 callback에 전달)
    - 상태 전환이 가능하다.

Promise는 특화된? 상태 머신이다.

- promise는 FSM이다.
- 세 가지 상태 중 하나: unfulfilled, fulfilled or failed.
- "then" 메소드를 사용하여 호출자의 입력을 받아들인다.
- 그것은 unfulfilled에서 fulfilled나 failed로 전환, 그리고 다신 전환되지 않는다.
- promise-FSM의 방향 그래프 표현



- 각 원은 promise가 존재할 수 있는 상태
- 화살표는 각 상태가 처리하는 입력(then, fulfill, fail).
- 화살표가 다시 상태를 가리키고 있는 경우, 이것은 입력이 새로운 상태로의 전이를 일으키지 않는 것을 의미
- 입력이 전환을 야기한다면, 화살표가 새로운 상태를 가리킨다.
- fulfilled와 failed상태는 둘 다 "entry actions"를 가진다. - 동작은 우리가 상태를 입력하자마자 실행된다.

이걸 언제 사용하나?

- Online/Offline connectivity
- Managing Views : Backbone 뷰는 "accepting-input"또는 "not-accepting-input" 상태일 것이다. 뷰는 여전히 모든 호출코드에 동일한 API를 제공한다. 하지만 상태에 따라 다르게 응답할 것이다. 양식이나 다른 시나리오의 번호를 제출하는 경우 유용하다.
- Initialization/Bootstrapping : Standing up a single page app, or a RabbitMQ client in node.js (just to name two examples) can involve a number of asynchronous calls that need to be performed in a certain order, and may involve a need to replay part or all of that workflow.
- State-based Routing : 당신의 클라이언트의 라우터는 앱의 여러 상태에 따라 다르게 작동 할 수 있습니다.
- Wizard/Step-Driven UI : 당신은 마법사 형식의 흐름을 제어하는 FSM을 사용할 수 있다. - 심지어 마법사 FSM과 병행하여 sibling/reactive FSM을 가진다.

Can I Get Some Help Here?

- 몇가지 강력한 기능을 가진 낮은 수준의 FSM 유틸리티를 원했기 때문에 machina.js를 만들었다. switch 기반의 FSM을 변환하여 봅시다.

```
var fsm = new machina.Fsm({

    // This will be the FSM's starting state
    initialState: "notReady",

    // initialize is invoked as soon as the
    // constructor has completed executing
    initialize: function() {
        var self = this;
        someSubject.on("ready", function() {
            self.handle("someSubject.ready");
        });
    },

    // The states object lets you organize your
    // states and input handlers. Each top level
    // member name is a state. The state object
```

```
// for each state contains function handlers
// OR a string value, indicating the only
// action is to transition to the state name
// specified in the string value.
states: {
    notReady: {
        "do.something" : function() {
            // deferUntilTransition is one of my
            // favorite machina features. It queues
            // the input up to be replayed in the
            // specified state name. The argument
            // is optional, and if not provided
            // machina will try to replay this
            // input again on the next transition
            this.deferUntilTransition("done");
        },
        // machina gives you a shortcut by allowing
        // the input handler value to be string state
        // name if the only reaction to an input is
        // to transition to a new state
        "someSubject.ready": "ready"
    },
    ready: {
        "do.something" : function() {
            worker.doWork(function(result) {
                // Do other work, etc. and then:
                this.result = result;
                // machina FSMs are event-emitters!
                this.emit("result.ready", this.result);
                this.transition("done");
            });
        }
    },
    done: {
        "do.something" : function() {
            this.emit("result.ready", this.result);
        }
    }
},

// this is a top level method that wraps the "handle"
// call that machina provides.
doSomething: function(callback) {
    // we could have easily required that callers
    // subscribe to events themselves, but we're
    // trying to be nice here
    this.once("result.ready", function(result){
        callback(result);
    });
    this.handle("do.something");
```

```
    }
});
```

- 따라서, machina는 다음과 같은 기능을 제공한다(이제 제한되지 않음):
    - 사용자 지정 건설 후 초기화 동작을 제공
    - 호출자가 아무것도 하지 않아도, 나중에까지 입력 지연
    - 예기치 않은 입력에 맞게 "범용" 처리기를 제공하는 능력
    - FSM은 이벤트 이미터(이벤트를 발생시키는 객체)이다. 그래서 다른 구성 요소들은 FSM을 관찰할 수 있다.
    - 모든 상태에 대한 _onEnter와 _onExit 핸들러

FSM Conclusion

- 장점
    - 매우 다양하다: FSM은 많은 실제 문제를 모델링할 수 있으며, 제대로 작동하지 않는 경우를 찾기 어렵다.
    - 워크플로우: FSM은 장기 실행, 비동기 워크플로우를 잘 처리
- 단점
    - 덜 알려진 패턴: 자바 스크립트 개발자 커뮤니티에서 FSM에 대한 인식이 낮은 것으로 나타난다.

Complementary Patterns

- FSM은 이벤트 뿐 아니라 continuation-passing을 이용하고 있다는 것을 알 수 있다.
- 사실, 5가지 패턴을 통해 우리의 전체 여행은 점차적으로 더 높은 수준의 추상화를 향해 계속해서 나아간다.
- 당신이 필요로 하는 추상화를 생성하기 위해 이 패턴들을 혼용하는 것을 보고 놀라지 마라.
- 오히려, 낮은 수준의 콜백 전달 패턴에 초점을 맞추고, 중첩 제거를 위해 promise를 사용하는 것이 피해를 주는 것이다. 이러한 패턴을 파악하는 것 외에도 우리는 함께 작업하는 이러한 패턴에서 만들 수있는 높은 수준의 추상화의 가능성을 확인해야한다.

마무리

- 위에 소개된 5가지 패턴이 비동기 코드를 길들이는데 도움이 되는 유일한 방법은 아니다.
- 그러나 이것은 "알고 있어야" 한다.

References

- http://tech.pro/blog/1402/five-patterns-to-help-you-tame-asynchronous-javascript