testharness.js 튜토리얼

들어가며

이 문서는 여러분이 GitHub Repository에서 클론(Clone)할 수 있는 W3C의 테스트 프레임워크 `testharness.js` 에 대한 튜토리얼을 제공합니다. 이 튜토리얼에서 여러분이 다른 테스트 프레임워크에 대해 반드시 친숙하다고 가정하고 있지는 않습니다만 자바스크립트 API를 사용하여 테스트를 작성하므로 이에 어느 정도 능숙하시기를 기대합니다.

만약 QUnit, Mocha나 Jasmine와 같은 다른 테스트 프레임워크에 친숙하다면 이 튜토리얼에서 상대적으로 쉽게 방법을 찾을 수 있을 것입니다. 사실 `testharness.js` 도 이들과 크게 다르지 않습니다만 브라우저의 API 구현에 대한 테스트를 모두 구현하는 방식보다는 자바스크립트로 표현하는데 훨씬 더 적합하게 개발된 많은 기능을 가지고 있습니다.

시작하기

다음 코드와 함께 시작해보도록 하죠! 여러분이 `testharness.js` 를 로딩하기 위해 가장 먼저 해야하는 것은 일반적인 방식대로 `script` 엘리먼트로 이를 포함하는 것입니다. 여러분은 자체적으로 복사본을 다운로드하여 원하는 곳에 로컬로 설치를 하거나 만약 W3C 서비스를 위한 테스트를 작성하고 있다면 다음과 같이 W3C 복사본을 사용할 수 있습니다.

```
<script src="/resources/testharness.js"></script>
<script src="/resources/testharnessreport.js"></script>
```

이쯤에서 자연스럽게 왜 2개의 파일이 존재하는지를 궁금할 것입니다. 이유는 단순합니다. 첫번째 것은 `testharness.js` 의 실제 구현 파일이고 두번째 것은 비어있는 파일입니다. 왜 비어있는 파일을 포함할까요? 특정한 업체가 여러분의 테스트 스윗을 사용할 때 `testharnessreport.js` 파일에 해당 내용을 덮어씌울 수 있도록 하는 것이 바로 그 목적입니다. 이를 통해 테스트 스윗 실행과 그들이 내부적인 테스트 리포팅 도구와 통합할 수 있습니다.

만약 결과를 자체적인 형식으로 렌더링하고 싶다면 여러분의 테스트에서 결과의 표시 위치에 ID `log` 를 가지는 HTML 엘리먼트를 포함해야 합니다. 이에 대한 예제는 이 문서의 마지막에 있습니다.

기본적인 테스트 방법

대다수의 기본적인 사용 방법은 실행하고자 하는 테스트를 위한 함수, 이름 및 추가적인 선택 사항으로 몇가지 옵션을 표현하는 상수 객체를 인자로 받는 `test()` 함수를 사용합니다.

```
test(function () {
    assert_true(true);
}, "True really is true");
```

주어진 함수는 반드시 하나 이상의 assertion을 포함하여야 합니다. 반대로 assertion들은 테스트 함수의 문맥에서 한번만 나타납니다. 완전하게 단일한 시험으로 고려된 경우라면 단일 테스트에서 여러개의 assertion을 포함할 수 있습니다. 즉, 실패에 대한 한개의 assertion은 테스트에 대한 실패를 뜻하고 반대로 모든 테스트에 대한 통과가 테스트의 성공의 필요조건이라고 얘기할 수 있습니다. 문서에는 원하는 만큼 많은 테스트들을 포함할 수 있습니다.

아래 예제는 2개의 assertion을 포함하고 있고 둘 다 통과할 것입니다.

```
test(function () {
    assert_true(true);
    assert_false(false);
}, "Truth is what you believe it to be");
```

그러나 아래 예제에서는 하나가 통과하였음에도 불구하고 다른 하나가 실패하였습니다. 이로 인해 이 테스트 전체가 실패로 보고될 것입니다.

```
test(function () {
    assert_true(true);
    assert_false(true);
}, "All opinions are equally valid.");
```

함수 및 이름 외에도 `test()` 는 또한 옵션들을 설정하기 위한 세번째 매개변수를 받을 수 있습니다. 아래의 메타데이터 소개 내에 이 옵션의 대부분이 문서화되어 있습니다.

여러분이 사용할 수 있는 일반적인 옵션은 `timeout` 입니다. 이는 밀리초 값(기본값 1000이며 1초와 동일합니다.)를 인자로 받습니다. 만약 테스트 내용이 실행하는데 `timeout` 보다 더 오랜 시간이 걸린다면 테스트는 중단되고 실패로 카운팅됩니다. 어떤 처리들는 실행에 1초보다 많은 시간이 걸릴 수 있으므로 만약 (초기 모바일 폰과 같은) 낮은 환경에서 복잡한 테스트를 실행할 때는 예시와 같이 타임아웃의 제한을 증가시키는게 유용할 것입니다.

```
test(function () {
    /* 뭔가 더 오래걸리고 느린 것을 여기에서 수행합니다. */
    assert_true(true);
}, "Long operation is successful", { timeout: 5000 });
```

다만 비동기 동작의 경우 이를 사용하지 말아야 한다는 것을 주의하시기 바랍니다. (이 기능은 비동기와는 무관하므로 동작하지 않습니다.) 이와 같은 경우는 이 주제에 대해 설명하는 아래 섹션을 보시기 바랍니다.

포함된 Assertion들

기본적으로, 여러분이 사용할 수 있는 훌륭한 assertion들이 있습니다. 대부분은 `assert_something(actu`

`al, expected, description)`의 형식을 취하고 있습니다만 일부는 다른 형태를 취하고 있습니다. 형식 중 일부가 `description`를 가지고 있다면 이 인자는 선택 사항이며 assertion이 실패했을 때의 제대로된 설명을 제공하기 위해 인간이 읽기 좋게 정의된 문자열입니다. 만약 이를 설정하지 않는다면 대신 기본 에러 메시지를 보게 될 것입니다.

assert_true(actual, description)

- `actual`이 `1`이나 `"문자열"`와 같은 "truthy"로써 처리되는 것이 아니라 반드시 자바스크립트의 `true` 인지를 엄격하게 확인합니다.
  - **역주**: truthy는 1, NOT null, true와 같이 조건식에서 참(True)의 의미로써 처리될 수 있는 값을 통칭하는 용어입니다.

```
test(function () {
    assert_true(true, "Truth is true");
    assert_true(1 === 1, "One is really one");
}, "Simple checks on truth");
```

assert_false(actual, description)

- `assert_true`와 동일하지만 반대로 동작합니다.
- `actual`이 (0이나 null과 같은) "falsy"로써가 아닌 자바스크립트의 `false`가 되어야 한다는 엄격함을 가지고 있습니다.
  - **역주**: truthy와는 반대로 falsy는 0, null, undefined, false와 같이 조건식에서 거짓(False)의 의미로써 처리될 수 있는 값을 통칭하는 용어입니다.

```
test(function () {
    assert_false(false, "Falsity is false");
    assert_false(1 === 0, "One is not zero");
}, "Simple checks on falsity");
```

assert_equals(actual, expected, description)

- `actual`과 `expected`가 동일한 값, 좀 더 정확히는 동일 객체인지를 확인합니다.
- 이를 사용할 때는 아래 사항들을 주의하여야 합니다.
  - 이러한 비교는 엄격하게 동작합니다.
  - 자바스크립트에서 제공되는 자동 형변환에는 의존하지 않아야 합니다.
  - *역주*: 이는 자동 형변환에서 발생할 수 있는 정밀도 상실이나 브라우저간의 차이 등에 의해 테스트의 결과값이 달라질 수 있기 때문입니다.

```
test(function () {
    assert_equals("dahut", "da" %2B "hut", "String concatenation");
    assert_equals(42, 6 * 7, "The ultimate answer");
}, "Simple checks on equality");
```

assert_not_equals(actual, expected, description)

- `assert_equals`과 동일하지만 반대로 동작합니다.
- `actual`과 `expected`가 같지 않아야함을 확인합니다.
- 비교의 엄격함에 대하여 동일한 규칙이 적용되므로 값이 유사해보여도 여전히 동일하지는 않습니다.

```
test(function () {
    assert_not_equals("dahut", "myth", "String comparison");
    assert_not_equals(42, "42", "The ultimate answer");
}, "Simple checks on unequality");
```

assert_in_array(actual, expected, description)

- `actual`이 `expected`에서 전달된 배열인지를 확인합니다.
- 어떠한 특이한 형태의 배열 멤버도 처리됩니다.
- 주의사항
  - 다중 배열인 경우 배열 내에서 재귀적으로 처리되지 않습니다. 즉, 1차원 배열만 지원됩니다.

```
test(function () {
    assert_in_array("dahut",
                    "chupacabra dahut unicorn".split(" "),
                    "Dahut hunting");
    assert_in_array(2017, [42, 47, 62, 2017] , "Lottery");
}, "Simple checks on membership");
```

assert_array_equals(actual, expected, description)

- `actual`과 `expected` 모두에 대해 배열을 취합니다.
- 같은 길이인지와 배열 내의 각 아이템이 다른 배열 내에서 해당되는 멤버와 `assert_equals`인지를 확인합니다.
- 이전의 assertion과 동일하게 1차원 배열이어야 합니다.

```
test(function () {
    assert_array_equals(["chupacabra", "dahut", "unicorn"],
                        "chupacabra dahut unicorn".split(" "),
                        "Dahut hunting");
    assert_array_equals([4, 9, 16],
                        [2, 3, 4].map(function (x) { return x * x; }),
                        "Square");
}, "Checks on identical membership");
```

assert_approx_equals(actual, expected, epsilon, description)

- 숫자인 `actual` 값을 받아 이것이 `expected`의 `epsilon` 인지를 확인합니다.

- 이는 여러분이 일어날 수 있는 추이들을 어느정도 알고 있을 경우의 실수 연산들에서 특히 유용하며 결과가 주어진 대략적인 범주에 있는지를 확인하고자 할 때 필요하지만 다른 경우에도 사용이 가능합니다.

```
test(function () {
    assert_approx_equals(Math.PI, 3.14, 0.01, "Roughly circular");
    assert_approx_equals(42, 47, 5, "47 is almost 42");
}, "Checks on epsilon equality");
```

assert_regexp_match(actual, expected, description)

- `actual`이 `expected`의 정규 표현에 정합하는지를 확인합니다.
- 여러분이 만들고자 하는대로 끝을 간단하거나 복잡하게 할 수 있으며 플래그에 의해 이를 생성할 수 있습니다.

```
test(function () {
    assert_regexp_match(document.title,
                        /^\w{5}-\w{10,12}\.js$/,
                        "That's my title");
    assert_regexp_match("A", /a/i, "Matching lowercase");
}, "Checks using regular expressions");
```

assert_own_property(object, property_name, description)

- 속성을 가진 `object`가 (프로토타입 체인을 상속받았는지가 아니라) 정말로 이를 가지고 있는지를 확인합니다.
- 자바스크립트 개발자는 이를 `hasOwnProperty`가 체크하는 것으로 이해할 것 입니다.
- 만약 이러한 중요한 메소드에 대해 알고 있지 않으시다면 MDN에서 이에 대해 읽어보시기 바랍니다.

```
test(function () {
    var gollum = { ring: "MIIIIINE!!!!" };
    assert_own_property(gollum, "ring", "Tricksy hobbitses!");
    /* `gollum`이 `toString`을 가지고 있음에도 불구하고 이는 실패할 것입니다. */
    assert_own_property(gollum,
                        "toString",
                        "I have that property, but it'ssss not mine.");
}, "Checks for property ownership");
```

assertinherits(object, propertyname, description)

- 객체 상에서 속성이 유효한지를 확인하는 것은 `assert_own_property`와 유사하지만 이를 보완할 수 있도록 상속된 프로토타입 체인 여부가 아니라 객체 자체의 속성이 아닌 경우에 assert가 발생합니다.

```
test(function () {
    var gollum = { ring: "MIIIIINE!!!!" };
    /* this will succeed here */
```

```
    assert_inherits(gollum,
                    "toString",
                    "I have that property, but it'ssss not mine.");
    assert_inherits(gollum,
                    "hasOwnProperty",
                    "This one works too.");
}, "Checks for property inheritance");
```

assert_idl_attribute(object, attribute_name, description)

- `assert_inherits`와 동일하며 이를 단순화한 alias입니다. 분명함을 위해 먼저 것을 유지하는 것이 더 나을 것입니다.

assertreadonly(object, propertyname, description)

- 주어진 `object` 상의 `property_name`이 실제로 읽기-전용이고 설정할 수 없는지를 확인합니다.

```
test(function () {
    assert_readonly(document, "nodeType", "You cannot change nodeType.");
}, "Checks for attribute readonlyness");
```

assert_throws(code, func, description)

- `func` 내의 코드가 어떤 예외를 꼭 발생하는지를 알고 있고 여러분이 기대하는 시간과 방식으로 코드가 (예외를) 발생하는지 확인하는 강력한 방법입니다.
- 이 assertion은 여러분이 `code`에 무엇을 전달하는지에 따라 다르게 동작합니다.
- 만약 `code`가 `null`이라면 모든 기존 예외가 동작할 것입니다. (다른 것들이 더 유용하므로 이는 특별히 확인할 필요는 없습니다.)

```
test(function () {
    assert_throws(null,
                  function () { document.appendChild(document); },
                  "Any exception.");
}, "Checks for exceptions (null)");
```

- 만약 `code`가 객체 종류라면 이의 `name` 속성을 확인합니다. 이 속성은 반드시 발생한 예외의 `name` 속성과 맞아야 합니다.
- 즉, 여러분이 특정한 `DOException` 객체를 여기에 전달할 수 있으며 만약 발생된 예외라면 이와 맞는 객체를 가져야 한다는 것을 의미합니다.

```
test(function () {
    assert_throws({ name: "Bad Kitten!" },
                  function () { throw { name: "Bad Kitten!"}; },
                  "Any exception with the right name.");
```

```
}, "Checks for exceptions (object)");
```

- `code`가 문자열이라면 이는 일반적으로 `DOMException` 명칭으로 인식될 수 있는 것 중의 하나여야 하며 `func`가 만든 연관된 `DOMExpception`인지가 확인됩니다.
- 기존 브라우저들과의 호환성을 위해 기존 예외와의 접촉이 지원되며 새로운 명칭으로 매핑됩니다.
- 따라서 예를들어 여러분이 `WrongDocumentError`를 의미하는 `WRONG_DOCUMENT_ERR`을 사용할 수 있습니다.
- 그렇지만 `WrongDocumentError` 스타일이 더 적합합니다.

```
test(function () {
    assert_throws("HierarchyRequestError",
                  function () { document.appendChild(document); },
                  "Specific DOM exception.");
}, "Checks for exceptions (string)");
```

assert_unreached(description)

- 어떠한 코드가 정말 도달할 수 없는지를 확인하기 위한 역할을 하는 아주 단순한 assertion입니다.
- 이는 설명(description)만을 인자로 받으며 단순하게 말하자면 이것이 호출될 때마다 항상 짜증스럽게 손을 들어야 하는 일을 집어치울 수 있습니다.
- 반대로 말하면 여기에서 여러분은 (건드리지도 않았으므로) 성공했다는 경우를 볼 수 있을 것입니다.

```
test(function () {
    if (true) return "where you came from";
    assert_unreached("Can't Touch This");
}, "Simple check on unreachability");
```

- 반면에 이는 코드가 이에 닿을 수 있기 때문에 실패할 것입니다.

```
test(function () {
    assert_unreached("Reaching where no coder has reached before");
}, "Failed check on unreachability");
```

비동기 테스트하기

전체가 동기적으로 처리되는 Web API는 점점 드물어지고 있습니다. 많은 수의 최근 API들이 동작에 약간의 시간이 걸리고 메인 스레드에서 멈추게 할 때마다 이를 비동기적으로 만드는데 많은 주의를 기울이고 있습니다.

고맙게도 `testharness.js`는 비동기 API를 테스트하는 것은 거의 동기 테스트만큼이나 쉬우며 모든 assertion들을 동일하게 하고 단지 어떻게 테스트를 설정하는지에 대한 몇가지 세밀한 부분들만이 다릅니다. 예제로 setTimeout 동작을 테스트하는 것으로 시작해보겠습니다.

먼저, `test(func, name, options)`를 호출하는 것 대신에 `async_test(name, options)`를 호

출하며 나중에 우리가 비동기 테스트의 흐름을 제어하기 위해 인터랙션할 객체인 이 함수의 반환값을 유지할 것입니다. 여러분이 보다시피 `async_test()`가 받는 `name`과 `options` 인자는 `test()`에서 사용되었던 인자였던 것과 `options`는 단지 선택값이라는 것까지 완전히 똑같습니다.

```
var stTest = async_test("Testing setTimeout()");
```

우리는 asserttion을 발생하기 위해 setTimeout 호출와 테스트가 끝났다는 flag를 사용할 것입니다. 이는 타임아웃을 취소하고 만약 assertion이 성공 한다면 (이 케이스는 평범합니다만) 테스트는 통과할 것입니다. 이는 다음과 같은 2가지 동작으로 수행됩니다. 첫번째는 (`test()`에서 사용된 첫번째 인자와 똑같이) 실행을 위해 개별적인 테스트를 정의하기 위해 사용되는 `step()` 메소드입니다. 두번째는 전체 테스트가 실행되었음을 `testharness.js`에 알려주기 위해 호출되는 `done()` 메소드입니다.

```
setTimeout(function () {
    stTest.step(function () {
        assert_true(true, "Truth is asynchronously true.");
    });
    stTest.done();
}, 10);
```

이는 가끔 객체의 특정한 `onfoo` 필드들에 이벤트 핸들러를 지정하는 것이 필요한 비동기 코드의 테스트에서의 경우입니다. 이는 `step()`으로 완료될 수 있습니다만 `step()` 호출은 그 자체를 함수 내로 감싸는 것이 필요하기 때문에 다루기가 좀 귀찮습니다. 다음과 같이 정확하고 손쉬운 사용례가 있습니다. `step_func`. `step()`과 같은 함수를 받지만 이벤트 핸들러로 바로 사용될 수 있는 함수를 반환합니다. 아래 XHR 예제는 상대적으로 이 편의성을 사용하도록 할 것입니다.

```
var xhrTest = async_test("Testing XHR access")
,   xhr
;
/* this in a step because it could throw */
xhrTest.step(function () {
    xhr = new XMLHttpRequest();
    xhr.open("GET", "using-testharness.html");
    xhr.onreadystatechange = xhrTest.step_func(function (ev) {
        assert_true(ev.isTrusted, "readystatechange is a trusted event");
        assert_false(ev.bubbles, "readystatechange is does not bubble");
        xhrTest.done();
    });
    xhr.send();
});
```

브라우저 접두사(vendor-prefixed)가 있는 기능의 테스트

새로운 기능이 더 이상 브라우저 접두사를 사용한 채로 배포되지 않기를 바라고 있는데도 여전히 테스트가 필요한 기능들이 브라우저 접두사가 있는 채로 존재합니다.

이러한 브라우저 접두사의 충격을 최소화하고 이러한 접두사 없이 테스트 스윗이 쉽게 실행되도록 만들기 위해 접두사가 있는 기능들의 사용과 요구사항을 선언할 수 있는 헌신적인 스크립트를 사용할 수 있습니다.

브라우저 접두사가 있는 기능을 위한 테스트 케이스를 작성하기 위해 다음과 같이 `testharness.js` 스크립트 **뒤에** 있을 스크립트를 추가해봅시다.

```
<script src="/common/vendor-prefix.js"></script>
```

이제 여러분은 브라우저 접두사를 사용하는 것이 필요한 기능들을 정의할 필요가 있습니다. 이는 저 스크립트 엘리먼트의 `data-prefixed-objects`와 `data-prefixed-prototypes` 속성을 설정하는 것으로 이를 처리할 수 있습니다.

이 두가지 속성은 접두사가 적용될 곳이 어딘지를 기술한 객체들의 배열을 JSON으로 인코딩된 상태로 받아들입니다. 이러한 객체 각각은 다음과 같이 `ancestors`와 `name` 속성을 가지고 있습니다. `ancestors`는 접두사를 가진 기능들이 존재하는 계층을 기술하는 객체 명칭들의 리스트를 포함하고 있으며 접두사가 없는 상태의 이름입니다.

만약 브라우저 접두사가 전역 네임스페이스에서 잘 알고 있는 객체에 적용되었을 경우 `data-prefixed-objects` 속성을 통해 이를 선언할 수 있습니다. 만약 브라우저 접두사가 잘 알고 있는 인터페이스로부터 초기화된 객체들에 적용되었다면 `data-prefixed-prototypes` 속성을 통해 이를 선언할 수 있습니다.

예를 들어, Media Capture and Streams API 테스트는 다음과 같이 접두사를 필요로 합니다.

- `navigator.getUserMedia` 메소드는 브라우저 접두사를 가진 상태로 배포되었습니다. (`navigator.mozGetUserMedia`와 `navigator.webkitGetUserMedia`가 그 예시입니다.)
- `HTMLMediaElement`의 `srcObject` 속성도 브라우저 접두사를 가진 상태로 배포되었습니다. (`vid.mozSrcObject`와 같은 것이 예시입니다.)

만약 이러한 기능들이 접두사를 가지고 있지 않고 있는 것처럼 테스트를 작성하는 것을 가능하도록 하기 위해 다음과 같이 아래 스크립트 선언에 한가지를 추가하여야 합니다.

```
<script src="vendor-prefix.js"
  data-prefixed-objects =  '[{"ancestors":["navigator"],
                             "name":"getUserMedia"}]'
  data-prefixed-prototypes='[{"ancestors":["HTMLMediaElement"],
                             "name":"srcObject"}]'>
</script>
```

메타데이터 포함하기

만약 여러분이 W3C Testing Framework내에 포함되기 위한 테스트를 작성 중이라면 (그리고 만약 여러분이 W3C 그룹을 위한 테스트들을 작성하고 있다면 반드시) 이 섹션이 흥미가 있으실 겁니다. 그렇지 않다면 그냥 넘어가셔도 됩니다.

Testing Framework로 통합하는 가장 훌륭한 방법은 여러분의 테스트가 메타데이터를 가지게 하는 것입니다. 만

약 HTML 파일 당 단 하나의 테스트를 가진다면 여러분의 테스트 메타데이터는 문서 내의 섹션에 포함되어야 하고 다시 말씀드리지만 여러분은 이 섹션을 넘겨도 괜찮습니다. 그러나 만약 하나의 문서 내에 여러분의 테스트 스윗 전체를 포함하고자 한다면 (물론 이러한 방식으로 테스트 스윗을 구성할 수 있습니다.) 이는 모든 `test()` 호출마다 테스트 메타데이터를 지정하는 것이 유용합니다.

이는 여러분이 이전에 보았던 `test()` 의 세번째 인자으로써 메타데이터를 제공하는 것으로 매우 쉽게 이를 처리할 수 있습니다. 여기에는 다음과 같이 3가지 필드를 사용할 수 있습니다. `help` 는 이 테스트가 시험되는 규격의 섹션을 지정하며 `assert` 는 여러분의 테스트가 포함하고 있는 assertion 설명들에 대한 배열입니다. 그리고 `author` 는 그냥 테스트의 저작자입니다.

```
test(function () {
        assert_true(true, "The spec says it's true.");
    },
    "True is true as per spec",
    {
        help:   "http://w3.org/TR/some-specification#truth-and-beauty",
        assert: ["Truth is true, you know."],
        author: "Robin Berjon "
    }
);
```

더 나은 사용 예제

대부분의 사용자는 이 섹션을 필요로 하지 않을 것입니다. 만약 동작하지 않는 복잡한 무언가를 처리하고자 하거나 궁금하지만 이것이 자주 필요하지 않을 것이므로 이해가 되지 않는 것처럼 보이더라도 걱정할 필요가 없다면 읽어보시기 바랍니다.

복잡한 설정

때로는 테스트가 실행되기 위한 복잡한 설정 동작을 수행하거나 테스트 실행의 전체 동작을 수정하는 것 모두가 중요할 수 있습니다. 이는 `setup(func, properties)` 를 사용하여 처리할 수 있습니다. 인자들은 선택적일 수도 아닐 수도 있으므로 `setup(func)` 나 `setup(properties)` 로 호출될 수도 있습니다.

일단 첫번째 테스트가 실행되고 나면 `setup()` 으로의 모든 호출은 즉시 제거될 것입니다. 따라서 이 섹션의 예제 중의 그 어떤 것도 실제로는 아무것도 하지 않지만 여전히 실제의 값을 보여주는데 유용할 것입니다.

근본적으로 `setup()` 의 함수 내에서 발생하는 모든 것은 (사용되었을 때) `setup()` 의 함수 내에 남아있습니다. 이는 여기서 실패가 대량으로 일어나더라도 테스트가 여전히 실행을 시도하는 것을 가능하게 합니다.

```
setup(function () {
    throw new Error("BOOM!");
});
```

properties 인자

`properties` 인자는 4개의 필드를 받을 수 있는 딕셔너리입니다.

timeout: ms

이는 단일 테스트보다는 페이지 내의 전체 테스트 셋을 위한 타임아웃값을 설정합니다. 개별 테스트들이 자체적인 타임아웃듯 발생하지 않을 지라도 전체 테스트가 느려지는 것을 우려한다면 이를 사용할 수 있습니다.

```
/* time out after 20 seconds */
setup({ timeout: 20000 });
```

explicit_done: true | false

일반적으로 테스트 실행은 document가 load 이벤트를 발생하거나 모든 동기적인 테스트들의 결과값(혹은 타임아웃)을 리포팅하고 만약 비동기 테스트가 있다면 그들이 잘 실행되었는지 혹은 타임아웃이 발생하였을 때 결과값을 리포팅하였을 때 완료되도록 고려되어야 합니다. (그리고 리포트를 생성하는 등) 다시 말해서 일반적인 동작 하에서 시스템은 테스트가 완전하게 실행되고 이를 전달해줄 필요가 없을 때를 어떻게 추정할 수 있는지를 알고 있어야 합니다. 여러분은 예를 들어 비동기적인 로딩이나 새로운 테스트의 생성과 같이 시스템을 혼란시킬 수 있는 무언가를 하고자 할 때 이 동작을 오버라이딩하고자 할 수 있습니다. 이 경우에 해당한다면 `explicit_done` 을 `true` 로 설정하고 실행하고자 하는 모든 테스트가 끝난 것을 인지하였을 때 여러분이 직접 글로벌 함수인 `done()` 을 호출합니다.

explicit_done

사용하고자 할 때 이를 이벤트 트리거들을 로딩하기 전에 true로 설정하는 것에 주의하는 것이 중요합니다. 아니면 여러분이 원하는대로 동작하는 것이 왜 아무것도 없는지를 확인하는데 놀라울 정도로 많은 시간을 소요하게 될 것입니다.

```
setup({ explicit_done: true });
/* ... at some point later... */
done();
```

output_document: Document

기본적으로 러너(runner)는 테스트 결과를 테스트가 실행 중인 동일 문서 내에 `log` ID를 가진 엘리먼트 내에 기록할 것입니다. 이는 대부분의 경우에 잘 동작합니다만 몇몇 상황에서는 여러분은 다른 방식으로 실행되기를 원할 수 있습니다. 테스트가 실행되는 문서는 아마 여러분이 테스트의 표시를 보기 위한 커다란 문서를 포함하고 있거나 새로운 팝업이나 새로운 탭을 열고 결과를 그곳에 출력하기를 원할 수 있습니다. 보다 일반적으로는 SVG 문서 내에서 테스트를 실행하고 있을 때 HTML로써 렌더링되지않는 `log` 엘리먼트의 제공은 약간 괜찮을 수 있습니다. 이 경우 출력을 HTML 문서로 리다이렉션(redirection)하는 것을 원할 것입니다. 이 모든 경우, Document 객체를 `output_document` 로 전달하기만 하면 됩니다.

```
setup({ output_document: window.parent.contentDocument });
setup({ output_document: document.getElementById
                         ("someIFrame").contentDocument });
```

explicit_timeout: true | false

몇몇 경우에 모든 타임아웃을 여러분이 직접 다루고 싶지 않습니다. (일반적으로는 여러분의 테스트가 더 큰 테

스트 러너의 컨텍스트에서 실행되고 있는 것은 여러분을 대신하여 타임아웃들을 제어하는 것입니다.) 이 경우 간단하게 `explicist_timeout` 을 `true` 로 설정하고 글로벌 함수 `timeout()` 을 직접 호출하여 타임아웃들을 조절할 수도 있습니다.

```
setup({ explicit_timeout: true });
/* ... at some point later if there really is a time out... */
timeout();
```

포맷 설정하기

여러분이 테스트 스윗에서 이슈들을 리포탕하고 싶거나 단순히 개발 도중 무엇을 기록하고 싶을 때마다 `toString()` 이 제공하는 것보다 더 인간-지향적인 결과를 생성하는 것은 매우 가치있는 것일 수 있다는 것은 대부분의 것들에서 제공되는 것과 마찬가지입니다. (여기에는 여러분이 `[object Object]` 에서 추론할 수 있는 것 정도만 있을 뿐입니다.) 이러한 때는 단순하게 전역 함수인 `format_value(value)` 를 사용합니다. 이는 어떻게 (재귀적인 탐색을 통해) 포맷 배열들, 제어 문자들을 가진 문자열들, 음수 0 그리고 이들보다 좀 더 중요한 DOM 노드 타입들을 포함하는 자바스크립트의 핵심 타입들을 알고 있습니다.

```
format_value(document);
format_value("foo bar");
format_value([-0, Infinity]);
```

테스트 설정하기

테스트를 작성하는 것은 매우 반복적인 노력일 수 있습니다. 가끔 실제적이고 기대하는 값들의 긴 리스트 상에서 그리고 어떻게 가볍게 만들 수 있을지가 고려되지 않은 테스팅 보일러플레이트의 부하가 뒤덮을 듯 발생했을 때도 여러분은 단순하게 동일한 assertion을 호출할 필요가 있습니다.

여러분이 더 편리하도록 하기 위해, `testhharness.js` 는 동일한 이름으로 각각이 기술되어 있는 실제값 및 기대값의 리스트 상에서 반복적으로 호출할 수 있는 아주 단순한 함수을 제공합니다. 이 함수의 형식은 `generate_tests(assert_something, [ [name, actual, expected], ...])` 입니다.

```
generate_tests(assert_equals, [
                              [ "Square of 2", 2 * 2, 4 ],
                              [ "Square of 3", 3 * 3, 9 ],
                              [ "Square of 4", 4 * 4, 16 ],
                              [ "Square of 5", 5 * 5, 25 ],
                              [ "Square of 6", 6 * 6, 36 ]
                            ]
);
```

콜백들

가끔 내부 동작에 반응하고 여러분만의 동작을 구축할 수 있도록 test harness 내에서 무엇이 어떻게 되고 있는지를 알 수 있다면 유용할 것입니다. (여러분만의 리포트 생성 혹은 여러분이 사용 중인 더 커다란 테스트 시스템과의 통합이 그 예입니다.)

이를 위해 `testharness.js` 는 여러분이 원한다면 알림을 설정할 수 있는 이벤트의 셋(Set)을 제공합니다. 만약 동일한 컨텍스트 내에서 테스트가 실행되고 있으며 이들이 시작 되기 전에 코드를 실행할 수 있다면 단순하게 몇가지 콜백들을 등록하는 것이 이러한 이벤트들을 인지할 수 있는 첫번째 방법입니다. 만약 테스트가 그 자체의 iframe (혹은 object 엘리먼트) 내에서 실행 중이라면 불려질 특정한 이름을 가진 함수의 생성에 의한 (어떠한 수준이던지) 여러분의 컨텍스트인 문서 안에 포함하는 것이 2번째 방법입니다. (후자의 접근 방법은 origin boundary 간에는 호출이 발생하지 않는다는 것을 주의하시기 바랍니다.)

start

설정이 이루어지고 첫번째 테스트가 생성되었을 때. 이는 여러분만의 설정을 수행할 수 있는 래퍼(wrapper)의 위치입니다.

```
/* in same context */
add_start_callback(function () {
    console.log("The tests have started running.");
});
/* in enclosing context */
function start_callback () {
    console.log("The tests have started running.");
}
```

result

결과가 생성되었을 때마다 발생합니다. 이는 동일 객체 상에서 존재하는 `PASS`, `FAIL`, `TIMEOUT` 혹은 `NOTRUN` 필드들(이들은 테스트가 성공, 실패, 타임 아웃 혹은 아예 실행되지 않았는지를 의미합니다.)과 비교할 수 있는 `status` 필드와 어떠한 에러라도 발생하면 이에 대한 에러 메세지를 제공하는 `message` 필드를 가진 `Test` 객체를 받습니다.

```
/* in same context */
add_result_callback(function (res) {
    console.log("Result received", res);
});
/* in enclosing context */
function result_callback (res) {
    console.log("Result received", res);
}
```

complete

테스트가 종료되었을 때 발생합니다. (성공이던 실패던). 이는 모든 결과들과 관련되어 있는 `result` 에 전달된 것과 같은 `Test` 객체들의 배열과 전체 실행의 상태를 기술하며 동일 객체 상에 존재하는 `OK`, `ERROR`

그리고 `TIMEOUT` 필드들(스윗이 완전하게 성공했는지 실패했는지 아니면 타임아웃이 발생했는지를 의미합니다.)과 비교할 수 있는 `status` 필드를 가진 상태 객체를 받습니다.

```
/* in same context */
add_completion_callback(function (allRes, status) {
    console.log("Test run completed", allRes, status);
});
/* in enclosing context */
function completion_callback (allRes, status) {
    console.log("Test run completed", allRes, status);
}
```

업데이트: 2014년 4월 10일 18:08 by Chang W. Doh