学习 单元测试

Chai断言库

概述

Chai提供了BDD/TDD风格的断言库，并且支持在node和浏览器端搭配任意测试框架进行使用。

安装

Node.js

```
1. npm install chai
```

Brower

```
1. <script src="chai.js" type="text/javascript"></script>
```

通过script标签引入，然后作为全局变量使用。

断言风格

Assert

TDD风格的断言，和node 的 assert断言相似，并且进行了扩展和浏览器兼容。可读性较差。

Should&&Expect

和Expect都是BDD风格，可读性强。两者使用相同的可连缀的语法来构建断言，两者的不同之处在于Expect是使用构造函数来创建断言，
例：`expect(foo).to.be.a('string');`，而Should通过在Object.prototype上新增方法来进行断言，例：`answer.should.to.equal(42);`
注意点：

- 引用

```
var chai = require('chai')
  , expect = chai.expect
  , should = chai.should();
```

- 兼容性
  Should不兼容IE浏览器，Should在使用上有一些坑，例如断言不存在的对象obj，可能会这样写：`obj.should.not.exist`，但是既然obj已经是undefind了，因此会报错。

Expect/Should API一览

Language Chains
为了让测试用例更加可读，提供了无断言功能的语言连接词。

- to
- be
- been
- is
- that
- which
- and
- has
- have
- with
- at
- of
- same
- but
- does

.not
连接在其他断言如`.equal`之前，表示“非”的意思，建议进行正面断言，即断言结果是什么，而不是去断言结果不是什么。

.deep
作为`.equal`，`.include`，`.members`，`.keys`，`.property`的前缀，使得断言将使用深比较而不是严格相等（`===`）。

.nested
允许在`.property`,`.include`中使用`.`语法和`[]`语法。

```
1. expect({a: {b: ['x', 'y']}}).to.have.nested.property('a.
   b[1]');
```

> 如果属性字面量中包含.或者[]使用`//`进行转译。

.own
作为`.property`,`.include`的前缀，使得断言将忽略继承属性，而只检查自身属性。

> `.own`不能连接到`.nested`后

.ordered
作为`.members`的前缀，使members断言时要求顺序一致。

.any
作为`.keys`的前缀，断言目标有所给keys中的任意一个。

.all
作为`.keys`的前缀，断言目标有所给的全部keys。

> `.all`是`.keys`的默认前置状态，即当`.keys`前不跟`.all`和`.any`时返回`.all.keys`的结果

.a(type[, msg])

- @param { String } type 类型（如string | object | error | promise）
- @param { String } msg *optional* 可选描述，断言失败后显示

> - 可以前置`.not`,但是建议正面断言，而不是反面断言，因为往往正面的type更少，而反面的type难以覆盖。例如应该断言‘aaa’是一个string，而不是应该断言它不是一个object。
> - `.a`在做为语言连接词时不起断言作用，只是提高可读性，并且可以使用`.an`代替。

.include(val[,msg])

- @param { Mixed } val 任意类型的值

- @param { String } msg *optional*

根据target的类型进行“包含”断言：

- string
  断言是否包含所传入子串subtring。
- object
  断言所传入对象的属性是否为target的属性的子集。
- Set 或 WeakSet
  断言所传入的值包含在是target的成员（使用SameValueZero 算法进行比较，即两个NaN和正负零都视为相同）。
- Map
  断言传入的值是target的值中的一个。

> - 由于`.inculde`的行为是根据target来表现的，因此在这之前检查target的类型是很重要的，因此在`.include`之前使用`.a(type)`进行断言。
> - 默认使用严格相等（===）进行比较，可以在之前使用`.deep`来使用深比较的方式
> - 当target为object时断言会检查对象的原型链，可以使用`.own`作为前缀排除原型属性
> - `.contain`，`.contains`作为`.include`的别名使用
> - `.include`也可以作为语言连接词使用

.ok
断言target是`==`true的，更多时候建议使用`===`或者深比较来代替

.true .false .null .undefined
断言target是`===`true false null undefined

.NaN
断言target为NaN

.exist
断言target不`===`null或者undefined

.empty

- target为string或者array时，断言length属性`===`0
- target为map或者set，断言size属性`===`0

.arguments
断言target是一个`arguments`对象

.equal(val[, msg])

- @param { Mixed } val 值
- @param { String } msg *optional* 断言描述信息

断言target`===`所给值

> `.equals`，`.eq`是`.equal`的别名

.eql(obj[, msg])
断言target与所给值深度相等。

> - `.eqls`是`.eql`的别名
> - 和`.deep.equal`的唯一区别是`.eql()`后面不能再连缀

.above(n[, msg]) .least(n[, msg]) .below(n[, msg]) .most(n[, msg])

- @param { Number } n
- @param { String } msg *optional*

断言target（number或者date）`>`，`>=`，`<`，`<=`所给值

> 最好断言这个值是什么,即使用`.equal`

.within(start, finish[, msg])

- @param { Number } start lower bound inclusive
- @param { Number } finish upper bound inclusive
- @param { String } msg *optional*

断言target（number或者date）处于某个闭区间[start,finish]

.instanceof(constructor[, msg])

- @param { Constructor } constructor
- @param { String } msg *optional*

断言target是所给构造函数的实例

> 当使用bable或typescript的时候可能会有问题

.property(name[, val[, msg]])

- @param { String } name
- @param { Mixed } val (optional)
- @param { String } msg *optional*

断言target有所给键或者键值对

> - 默认使用`===`,可以使用`.deep`来使用深度比较
> - 默认包含可枚举和不可枚举属性，如果只想断言可枚举属性可前置`.own`

.lengthOf(n[, msg])

- @param { Number } n
- @param { String } msg *optional*

断言target的length属性`===`所给值

.match(re[, msg])

- @param { RegExp } re
- @param { String } msg *optional*

断言target匹配所给正则

.string(str[, msg])

- @param { String } str
- @param { String } msg *optional*

断言target String包含所给子字符串

> 可以使用contain()代替

.keys(key1[, key2[, …]])

- @param { String | Array | Object } keys

断言target object,array,map或者set有所给的keys

> 当传入的值是对象的时候，对象的值将被忽略

.throw([errorLike], [errMsgMatcher], [msg])

- @param { Error | ErrorConstructor } errorLike

- @param { String | RegExp } errMsgMatcher error message
- @param { String } msg *optional*
- @see https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Error#Error_types

断言target函数抛出的错误

.respondTo(method[, msg])

- @param { String } method
- @param { String } msg *optional*
  断言target是否有所给method（自身或者继承）

target为构造函数时,断言这个构造函数有所给method方法

```
1. function Foo(){}
2. expect(new Foo()).to.respondTo('method')
```

target为函数时,断言函数的**原型上**有所给方法（自身的或者继承的）

```
1. expect(Foo).to.respondTo('method')
```

> 两者的区别是前者直接断言本身，后者断言target的prototype。可以通过前置 `.itself` 使得断言函数时断言其本身

.satisfy(matcher[, msg])

- @param { Function } matcher
- @param { String } msg *optional*

将target作为matcher的第一个参数传入，断言matcher函数的返回值为true

.closeTo(expected, delta[, msg])

- @param { Number } expected
- @param { Number } delta
- @param { String } msg *optional*

断言target是否在[expected-delta,expected+delta]的范围内

.members(set[, msg])

- @param { Array } set
- @param { String } msg *optional*

断言target 数组中包含所给数组

.oneOf(list[, msg])

- @param { Array.<*> } list
- @param { String } msg *optional*

断言target是所给list的一个成员

.change(subject[, prop[, msg]])

- @param { String } subject
- @param { String } prop name *optional*
- @param { String } msg *optional*

断言target函数调用前后，subject的值，或者subject.prop的值相等

.increase(subject[, prop[, msg]]) .decrease(subject[, prop[, msg]])

- @param { String | Function } subject
- @param { String } prop name *optional*
- @param { String } msg *optional*

断言target函数调用前后subject的值，或者subject.prop的值增加或减少

.by(delta[, msg])

- @param { Number } delta
- @param { String } msg *optional*

连缀在`.increase`，`.decrease`之后，断言增加减少的值

.extensible
断言target是可扩展的（可以添加新的属性）

.sealed
断言target是封闭的（不能添加新属性，不能重新分配或删除已有属性，但是可以对已有属性重新赋值）

.frozen
断言target是冻结的（不能添加新属性，不能重新分配已有属性，删除已有属性或对已有属性重新赋值）

.finite
断言target是number，并且不是NaN或者正负无穷

.fail(actual, expected, [message], [operator])

- @param { Mixed } actual
- @param { Mixed } expected
- @param { String } message
- @param { String } operator

抛出一个错误