iOS开发中的单元测试（三）——URLManager中的测试用例解析

作者：gaosboy | 时间：August 29, 2013 | 分类：猪在写代码

本文首发于 InfoQ中文站

URLManager是一个基于UINavigationController和UIViewController，以URL Scheme为设计基础的导航控件，目的是实现ViewController的松耦合，不依赖。

准备框架，定义基类

首先按照之前的两篇文章介绍的方法导入单元测试框架和匹配引擎框架，建立好测试Target，并配置编译选项。

定义测试用例基类：UMTestCase（代码1），其他用例全部继承自UMTestCase。

```
#import
@interface UMTestCase : GHTestCase
@end
```

代码1，UMTestCase，用例基类

构建用例

URLManager工具类（UMTools）测试用例（UMToolsTestCase）。UMTools中扩展了NSURL，NSString和UIView，方法涉及到给URL添加QueryString和从QueryString中读取参数，对字符串做子串判断，进行URL的编码和解码，对UIView的x，y，width和height的直接读写等。需要在用例中定义测试过程中会使用到属性（代码2），并在setUpClass中初始化他们（代码3）。

代码2，定义属性

```
// 普通字符串，带有字母和数字
@property   (strong, nonatomic)     NSString    *string;
// 普通字符串，仅带有字母
@property   (strong, nonatomic)     NSString    *stringWithoutNumber;
// 将被做URLEncode的字符串，含有特殊字符和汉字
@property   (strong, nonatomic)     NSString    *toBeEncode;
// 把 toBeEncode 编码后的串
@property   (strong, nonatomic)     NSString    *encoded;
// 普通的URL，带有QueryString
@property   (strong, nonatomic)     NSURL       *url;
// 去掉上边一个URL的QueryString
@property   (strong, nonatomic)     NSURL       *noQueryUrl;
// 一个普通的UIView
@property   (strong, nonatomic)     UIView      *view;
```

```
(void)setUpClass
{
    self.string                 = @"NSString For Test with a number 8848.";
    self.stringWithoutNumber    = @"NSString For Test.";
    self.toBeEncode             = @"~!@#$%^&*()_+=-[]{}:;\"'<>.,/?123qwe汉字";
    self.encoded                = @"%7E%21%40%23%24%25%5E%26%2A%28%29_%2B%3D-
%5B%5D%

7B%7D%3A%3B%22%27%3C%3E.%2C%2F%3F123qwe%E6%B1%89%E5%AD%97";
    self.url                    = [NSURL URLWithString:@"http://example.com
                                        /patha/pathb/?p2=v2&p1=v1"];
    self.noQueryUrl             = [NSURL URLWithString:@"http://example.com
                                        /patha/pathb/"];
    self.view                   = [[UIView alloc] initWithFrame:CGRectMake(10.0f,
                                        10.0f, 100.0f, 100.f)];
}
```

代码3，初始化属性

使用单元测试框架中的断言处理简单用例

单元测试是白盒测试，要做到路径覆盖（代码4）。 对“ContainsString”的测试进行正向和反向两种情况（即YES和NO两种返回结果）。

```
#pragma mark - UMString

- (void)testUMStringContainsString
{
    NSString *p = @"For";
    NSString *np = @"BAD";
    GHAssertTrue([self.string containsString:p],
                 @"\"%@\" should contains \"%@\".",
                 self.string, p);
    GHAssertFalse([self.string containsString:np],
                  @"\"%@\" should not contain \"%@\".",
                  self.string, p);
```

代码4，字符串测试用例

同时单元测试又要对功能负责，因此在路径覆盖之外还要尽量照顾到完整的功能。例如，对URLEncode的测试（代码5），要对尽量全面的特殊字符进行测试，而不是从源码实现中取出枚举的字符。

```
(void)testUrlencode
{
    GHAssertEqualStrings([self.toBeEncode urlencode], self.encoded,
                         @"URLEncode Error.",
                         self.toBeEncode, self.encoded);
    GHAssertEqualStrings([self.encoded urldecode], self.toBeEncode,
                         @"URLDecode Error.",
                         self.encoded, self.toBeEncode);
}
```

代码5，URLEncode测试用例

在进行这个测试之前，urlencode的实现忽视了对“~”的编码，正是由于单元测试用例所取的特殊字符是单独列举，并非从实现枚举中获取，检查出了这个错误。

引入匹配引擎，使用匹配引擎默认规则

前文提到过匹配引擎可以使测试用例中的断言更加丰富，URLManager的用例中也使用了匹配引擎：OCHamcrest。

在此前的介绍中提到，引入OCHamcrest可以通过定义“HC_SHORTHAND”来开启匹配引擎的简写模式。因为开启简写模式后匹配规则中的“containsString”规则和上述例子（代码5）中的“containsString:”方法命名冲突，导致测试程序无法正常运行，所以这个工程直接使用了类似“HC_asserTaht”这样带有HC前缀的完整命名。

我建议使用匹配引擎的开发者谨慎开启简写功能，OCHamcrest的匹配规则简写通常是很常见的单词，非常容易与工程中的类定义或方法定义重名。即使当下没有规则和方法名发生冲突，随着工程代码量的增加，一旦出现命名冲突的情况，重构的成本将非常高。

匹配引擎可以提供更丰富的断言，最简单的例如，URLManager的UMURL扩展支持向一个URL上添加参数，对这个方法测试断言就用到了匹配某个字符串是否包含某子串的规则（代码6）。

```
#pragma mark - UMURL

- (void)testAddParams
{
    NSURL *queryUrl = [self.noQueryUrl addParams:@{@"p1":@"v1",@"p2":@"v2"}];
    HC_assertThat(queryUrl.absoluteString, HC_containsString(@"p1=v1"));
    HC_assertThat(queryUrl.absoluteString, HC_containsString(@"p2=v2"));
}
```

代码6，URL参数测试用例

匹配规则中的陷阱

由于匹配规则的粒度较细，所以对于某些运行结果需要考虑到多种情况，否则正常的结果也可能会断言失败。

例如测试用例期望得到一个空容器（例如：NSArray），而SDK则认为这个容器已经没有存在的必要而释放了他，返回的是一个nil。对removeAllSubviews的测试中，对一个view调用removeAllSubviews方法，期望view.subviews为空。在SDK 6.x甚至SDK 7 DP1之前，都是没问题的，但在SDK 7 DP3中，SDK会把所有清空的容器和对象释放，以回收系统资源。在这种条件下view.subviews返回的就是nil，如果只是做类似HC_empty()这样的匹配，断言会失败，所以在断言之前做一个subviews属性的空判断（代码7）。

```
(void)testRemoveAllSubviews
{
    UIView *subViewA = [[UIView alloc] init];
    UIView *subViewB = [[UIView alloc] init];

    [self.view addSubview:subViewA];
    [self.view addSubview:subViewB];
    HC_assertThat(self.view.subviews, HC_containsInAnyOrder(subViewA, subViewB,
nil));

    [self.view removeAllSubviews];
    if (nil != self.view.subviews) {
        HC_assertThat(self.view.subviews, HC_empty());
    }
}
```

代码7，removeAllSubviews用例

另外，在默认匹配规则中会有一些容易产生歧义的命名，以collection的containsInAnyOrder为例：匹配对象是一个collection对象（也就是遵循NSFastEnumeration协议的对象，NSArray等），给出若干个匹配规则或元素。期待这个规则匹配该对象是否包含给出的若干元素，且不关心顺序。但在实际测试过程中会发现，这个规则要求给出的元素必须是该collection对象的完备集，也就是说要求给出的元素列表和要匹配的容器对象中的元素必须是相等的结合，但允许不关注顺序。

对UMNavigationController的测试中，需要判断增加一项URL Mapping是否生效，如果使用该匹配规则，就不能单纯判断config是否包含增量的URL，要断言成功必须连同此前config属性初始化写入的值一起考虑，使用一个完整的元素集合进行匹配（代码8）。

```
(void)testAddConfig
{
    [UMNavigationController setViewControllerName:@"ViewControllerA" forURL:@"
um://viewa2"];
    NSMutableDictionary *config = [UMNavigationController config];
    NSLog(@"%@", [config allKeys]);
    HC_assertThat([config allKeys],
                  HC_containsInAnyOrder(HC_equalTo(@"um://viewa2"), HC_equalTo(@"
um://viewa"),
                                        HC_equalTo(@"um://viewb"), nil));
    GHAssertEqualStrings(config[@"um://viewa2"], @"ViewControllerA",
                         @"config set error.");
}
```

代码8，AddConfig用例

自建匹配规则

上述例子表明匹配规则往往无法恰好满足测试需求，需要对默认规则进行升级。

升级一个匹配规则，首先阅读OCHamcrest默认规则源码，找到无法满足需求的代码。上述HC_containsInAnyOrder的例子中，个性需求是某个collection是否包含某几个元素（而非完整集合），而默认规则只能匹配完整集合。阅读源码（代码9）可以发现，在maches:describingMismatchTo:函数中，对规则对象的collection属性（要进行匹配的容器对象）进行遍历，并逐个调用matches:方法。matches:方法

中针对每个collection属性中的元素遍历匹配规则集合（matchers），并从规则集合（matchers）中移除匹配成功的规则。当给出的规则集合（matchers）全部成功匹配过之后，matchers属性已经为空。若此时对collection属性的遍历继续进行，matches:方法就不会进入匹配逻辑，直接跳出循环返回NO，导致匹配失败。

```
(BOOL)matches:(id)item
{
    NSUInteger index = 0;
    for (id matcher in matchers)
    {
        if ([matcher matches:item])
        {
            [matchers removeObjectAtIndex:index];
            return YES;
        }
        ++index;
    }
    [[mismatchDescription appendText:@"not matched: "]
appendDescriptionOf:item];
    return NO;
}

- (BOOL)matches:(id)collection describingMismatchTo:(id)
mismatchDescription
{
    if (![collection conformsToProtocol:@protocol(NSFastEnumeration)])
    {
        [super describeMismatchOf:collection to:mismatchDescription];
```

代码9，HC_containsInAnyOrder规则中的两个核心方法

我们的需求是，当匹配规则列表全部成功匹配之后就是此次匹配成功的标志。所以需要修改matches:方法中的匹配逻辑，当匹配列表为空则返回YES。

升级方案是继承HCIsCollectionContainingInAnyOrder创建一个新的匹配规则类HCIsCollectionHavingInAnyOrder；重新定义匹配规则HC_hasInAnyOrder；重写调用matches:方法的matches:describingMismatchTo:方法（代码10）；更新的核心是定义一个HCMatchingInAnyOrderEx类，按照个性需求定义matches:方法（代码11）。使用这个修改过的匹配规则就可以判断一个Collection是否包含某个几个元素了。

```
@implementation HCIsCollectionHavingInAnyOrder

- (BOOL)matches:(id)collection describingMismatchTo:(id)
mismatchDescription
{
    if (![collection conformsToProtocol:@protocol(NSFastEnumeration)])
    {
        [super describeMismatchOf:collection to:mismatchDescription];
        return NO;
    }

    HCMatchingInAnyOrderEx *matchSequence =
    [[HCMatchingInAnyOrderEx alloc] initWithMatchers:matchers
                                mismatchDescription:mismatchDescription];
    for (id item in collection)
        if (![matchSequence matches:item])
            return NO;

    return [matchSequence isFinishedWith:collection];
}

@end
```

代码10，HCIsCollectionHavingInAnyOrder实现

```
(BOOL)matches:(id)item
{
    NSUInteger index = 0;
    BOOL matched = (0 >= [self.matchers count]);
    for (id matcher in self.matchers)
    {
        if ([matcher matches:item]) {
            [self.matchers removeObjectAtIndex:index];
            matched = YES;
            return YES;
        }
        ++index;
    }
    return matched;
}
```

代码11，更新过的matches:方法

```
(void)testAddConfig
{
    [UMNavigationController setViewControllerName:@"ViewControllerA" forURL:@"um://
viewa2"];
    NSMutableDictionary *config = [UMNavigationController config];
    HC_assertThat([config allKeys],
                  HC_hasInAnyOrder(HC_equalTo(@"um://viewa2"), nil));
    GHAssertEqualStrings(config[@"um://viewa2"], @"ViewControllerA",
                         @"config set error.");
}
```

代码12，使用新规则的测试用例

另一个方面，在测试过程中会出现各种逻辑，有时默认规则根本无法覆盖，需要完全自建规则。例如对CGPoint和CGSize的相等匹配，如代码13中对UMView的size和origin方法测试。OCHamcrest的默认规则中根本没有提供任何针对CGPoint和CGSize两个结构体的匹配规则，所以要完成这个测试就需要自己定义针对这两种数据结构的匹配规则。

```
#pragma mark - UMView

    HC_assertThat(NSStringFromCGSize(self.view.size),
                  HC_equalToSize(self.view.frame.size));
    HC_assertThat(NSStringFromCGPoint(self.view.origin),
                  HC_equalToPoint(CGPointMake(self.view.frame.origin.x, self.
view.frame.origin.y)));
```

代码13，UMView测试用例片段

自定义匹配规则的详细说明可以参见上一篇《iOS开发中的单元测试（二）》，本文只对开发自定义规则中遇到的问题和需要特殊处理的方面进行解释。

OCHamcrest的匹配规则要求被匹配的必须是一个有强引用的对象，所以当被匹配的是一个struct结构（如CGPoint）需要进行一次转换，如代码14中定义的这个规则扩展——OBJC_EXPORT id HC_equalToPoint(CGPoint point)。 在CGPoint相等匹配的规则中，需要先把CGPoint转为字符串后传入断言方法，规则会把这个字符串储存起来，并与后续给出的CGPoint进行比较。匹配引擎对传入的需要进行匹配的参数类型没做任何限制，所以规则可以直接传入CGPoint。

开发自定义规则一般建议同时定义SHORTHAND，即使当前单元测试中不会用到（例如本文中的测试），但这个规则被其他复用的时候，可能会用到SHORTHAND命名。

```
#import

OBJC_EXPORT id HC_equalToPoint(CGPoint point);

#ifdef HC_SHORTHAND
#define equalToPoint HC_equalToPoint
#endif

@interface HCIsEqualToPoint : HCBaseMatcher

+ (id)equalToPoint:(CGPoint)point;
- (id)initWithPoint:(CGPoint)point;

@property (nonatomic, assign)       CGFloat     x;
@property (nonatomic, assign)       CGFloat     y;

@end
```

代码14，扩展匹配规则HC_equalToPoint定义

在匹配规则的过程中，有一个点需要特别注意，即对匹配对象类型和完整性的判断。往往开发者把注意力都放在对对象值的匹配上，而忽略了类型和完整性这类判断，最终导致整个用例运行失败，但无法准确定位出错的位置。上面提到的对subviews是否为空的判断也是这样的一个例子。所以在自定义的匹配规则中就需要考虑到这方面的问题，如代码15的matches:方法中，先要对传入的泛型对象item校验是否为字符串，后再转化为CGPoint对象，并进行相应比对。示例中给出的是一种较简单的情况，在更复杂的情况下，除了对泛型对象的类进行校验，还要校验其是否响应某方法，属性类型，空判断，等。

```
#import "HCIsEqualToPoint.h"
#import

id  HC_equalToPoint(CGPoint point)
{
    return [HCIsEqualToPoint equalToPoint:point];
}

@implementation HCIsEqualToPoint

+ (id)equalToPoint:(CGPoint)point
{
    return [[self alloc] initWithPoint:point];
}

- (id)initWithPoint:(CGPoint)point
{
    self = [super init];
    if (self) {
        self.x = point.x;
        self.y = point.y;
    }
    return self;
```

代码15，扩展匹配规则HC_equalToPoint实现

一个操作多个测试方法

以上提到的几个例子中所测试的都是非常简单的操作，所以一个测试方法覆盖了一个或多个操作，但对于较复杂的操作，往往需要多个测试方法，循序渐进的断言。例如测试通过URL生成UMViewController的用例，生成一个UMViewController实例由简单到复杂可以有三种简单方式：简单的URL生成，带参数的URL生成和带Query字典的URL生成，此外还有URL参数和Query字典共用的方式。所以对于这个操作至少需要使用4个测试方法（代码16）分别进行测试。

```
(void)testViewControllerForSimpleURL
{
    self.viewControllerA = (ViewControllerA *)[self.navigator
                                               viewControllerForURL:
                                               [NSURL
URLWithString:@"um://viewa"]
                                               withQuery:nil];

    HC_assertThat(self.viewControllerA, HC_instanceOf([UMViewController
class]));
    HC_assertThat(self.viewControllerA, HC_isA([ViewControllerA class]));
}

- (void)testViewControllerForURLWithArgs
{
    self.viewControllerA = (ViewControllerA *)[self.navigator
                            viewControllerForURL:[NSURL
URLWithString:@"um://viewa?
p1=v1&p2=v2"]
                            withQuery:nil];

    HC_assertThat(self.viewControllerA, HC_instanceOf([UMViewController
class]));
```

代码16，测试通过URL生成UMViewController的用例

一个测试方法多次断言

除了一个操作需要多个测试方法的情况，在同一个测试方法中也会有对一个结果进行多次断言的情况（上述用例代码16中已经是这种情况，一下用例更具代表性）。这种情况发生在操作结果较为复杂的情况下，例如生成一个UMNavigationController（代码17）就是这种情况：UMNavigationController的初始化方法是带RootViewController参数的，所以初始化的实例除了判断其本身是否为UINavigationController的子类和UMNavigationController实例外，还要判断rootViewController的合法性，以及viewControllers数组的正确性。

```
(void)testInitWihtRootViewControllerURL
{
    UMNavigationController *navigator = [[UMNavigationController alloc]
       initWithRootViewControllerURL:[NSURL URLWithString:@"um://viewb"]];

    HC_assertThat(navigator, HC_instanceOf([UINavigationController class]));
    HC_assertThat(navigator, HC_isA([UMNavigationController class]));

    HC_assertThat(navigator.rootViewController,
                            HC_instanceOf([UMViewController class]));
    HC_assertThat(navigator.rootViewController, HC_isA([ViewControllerB class]));

    HC_assertThatInteger(navigator.viewControllers.count, HC_equalToInteger(1));
    HC_assertThat(navigator.viewControllers,
                  HC_hasInAnyOrder(HC_instanceOf([UMViewController class]), nil));
    HC_assertThat(navigator.viewControllers,
                  HC_hasInAnyOrder(HC_isA([ViewControllerB class]), nil));
    HC_assertThat(navigator.viewControllers,
                  HC_hasInAnyOrder(HC_is(navigator.rootViewController), nil));
}
```

代码17，测试生成UMNavigationController的用例

总结

本文一共取了URLManager中的17段代码片段作为例子，介绍了从利用测试框架提供的断言方法进行简单的测试，一直到使用自定义匹配引擎规则创建较复杂测试用例，并且提到了部分测试引擎和匹配引擎使用过程中会遇到的陷阱。旨在推动开发者能够在开发过程中更简单高效的使用单元测试，为提升代码质量增加一份保障。读者可以在URLManager的工程中阅读更多的测试用例代码。

标签：ios, urlmanager

添加新评论

称呼 *

Email *

网站

http://

内容 *

提交评论