测试并发程序

在开发高质量应用程序的时候，测试是一个很重要的工具。在过去，并发在应用程序架构中还没有那么重要，测试就相对简单。随着这几年的发展，并发设计模式已愈发重要了，想要测试好，已成了一个不小的挑战.

测试并发代码最主要的困难在于程序或者信息流不再反映在调用堆栈上了。函数并不会立即返回给调用者结果，而是通过回调函数，闭包（Block），通知或者一些类似的机制来推迟返回结果，这就让测试变得更加困难。

然而，测试异步代码也会带来一些好处，比如可以提早暴露一些较差的设计决定，让最终的实现变得更加清晰。

异步测试的问题

首先，我们来看一个简单的同步单元测试：两个数求和。

+ (int)add:(int)a to:(int)b {

return a + b; }

测试这个方法很简单，只需要比较该方法返回的值和期望的值是否相同，不相同，则测试失败。

- (void)testAddition {

int result = [Calculator add:2 to:2];

STAssertEquals(result, 4, nil); }

接下来，我们利用Block将该方法改变成异步返回结果。当然了，我们也会添加一个Bug,让测试失败。

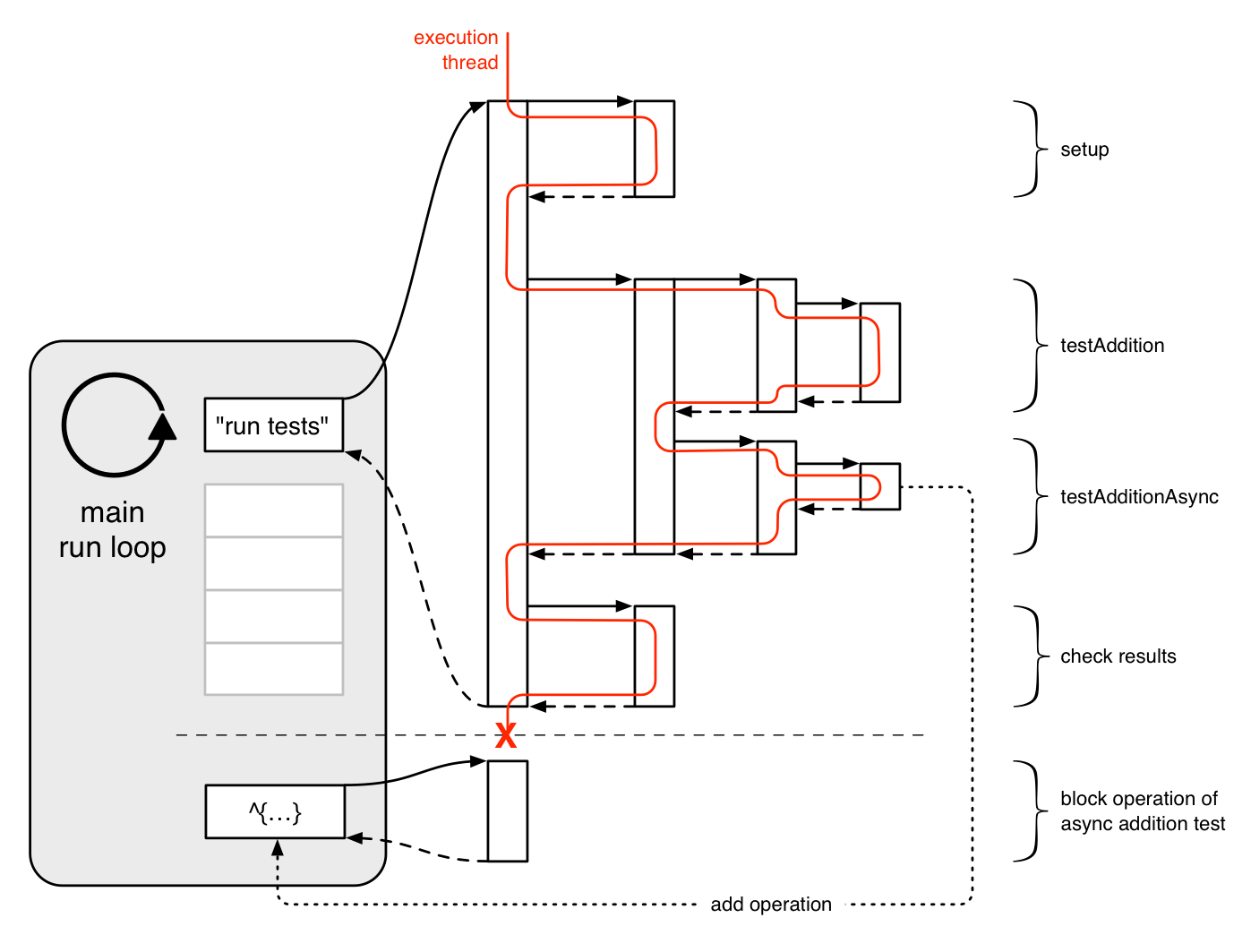
+ (int)add:(int)a to:(int)b block:(void(^)(int))block{

[[NSOperationQueue mainQueue] addOperationWithBlock^{ block(a - b); // Buggy implementation }]; }

虽然这是一个人为的例子，但是它却真实的反应了在编程中可能遇到的问题，只不过实际过程更复杂罢了。我们能想到的最简单测试方法就是把断言放到Block的实现中，然而，这种情况下，测试绝不会失败，Bug却依然存在。断言为什么会失败的？

关于SenTestingKit

XCode4使用的测试框架是基于开源的OCUnit, 为了能更好的理解异步测试，我们需要了解一下各种测试方法之间执行顺序的不同。下图展示了一个简化的流程。



测试程序从主run loop开始后，执行顺序主要有以下几步:

1、配置一个测试套件包含所有的测试(如在工程的scheme中配置)

2、运行测试套件，主要是调用以test开头的所有方法。运行结束后会返回一个对象，它包含所有执行的单个测试的结果。

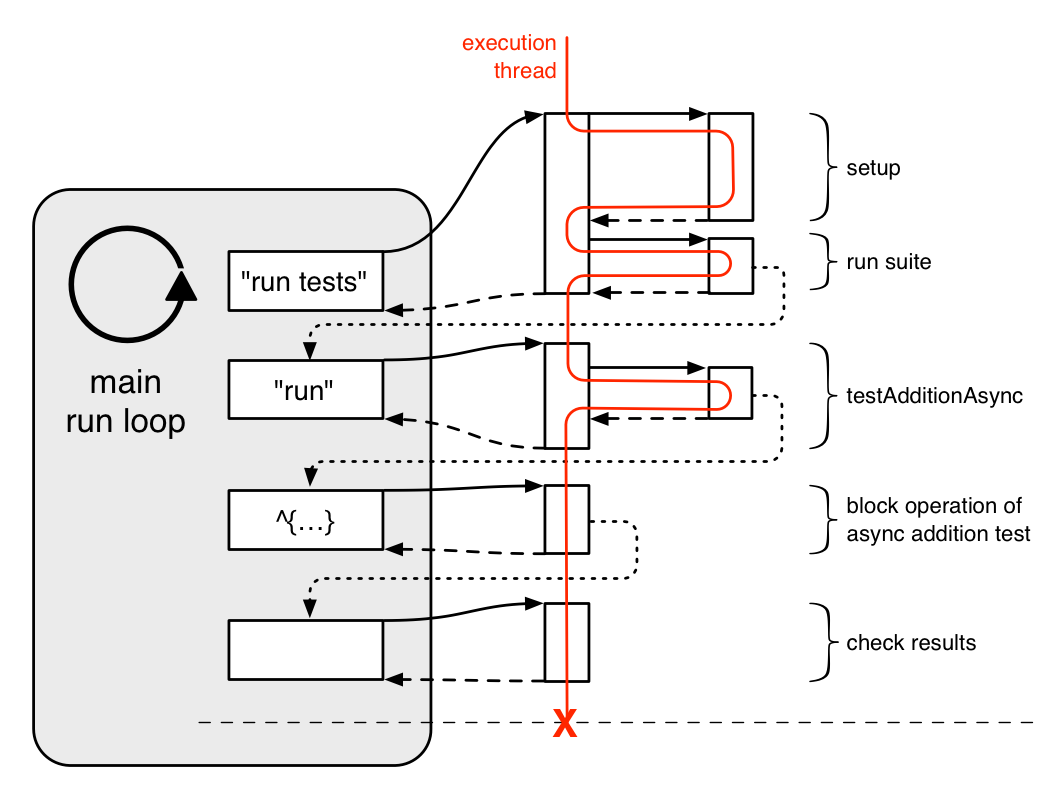
3、调用exit()方法，退出测试。

我们最感兴趣的是单个测试方法是如何被调用的。在异步测试中，包含断言的Block在主run loop中排队。当所有的单个测试执行完毕后，测试框架就会退出测试，而block却从来没有被调用，因此不会触发测试失败。

有很多种方式可以来解决这个问题，问题的核心在于：在测试方法未返回结果，框架也还未检查测试结果之前，都必须运行主run loop和处理加入主run loop队列操作。

Kiwi用轮询的方式来解决，GHUnit用一个单独的测试类，它会在测试方法内初始化，结束时发送一个通知。这两种方式都是通过代码来确保异步测试方法在测试结束之前都不会返回。SenTestingKit的异步扩展

我们的解决方式是对 SenTestingKit添加一个扩展，它将同步执行封装在堆栈上，在主线程上将每一个部分作。which winds up the synchronous execution on the stack and enqueues each part as a block on the main queue.正如下图所见，验证异步测试失败或者成功的方法被放在一个block内，它在框架查看测试结果之前被加入到了主run loop队列中。这种执行顺序允许我们开启一个测试并等待它的测试结果。



如果测试方法以Async结尾，框架就会认为该方法是异步测试。在异步测试中，我们在Block中添加一个宏来表示测试成功，为了防止Block永远不会被调用，我们还添加了一个超时方法。之前的错误的测试方法修改后如下所示：

- (void)testAdditionAsync {

[Calculator add:2 to:2 block^(int result){

STAssertEquals(result, 4, nil);

STSuccess(); // Calling this macro reports success }];

STFailAfter(2.0, @"Timeout"); }

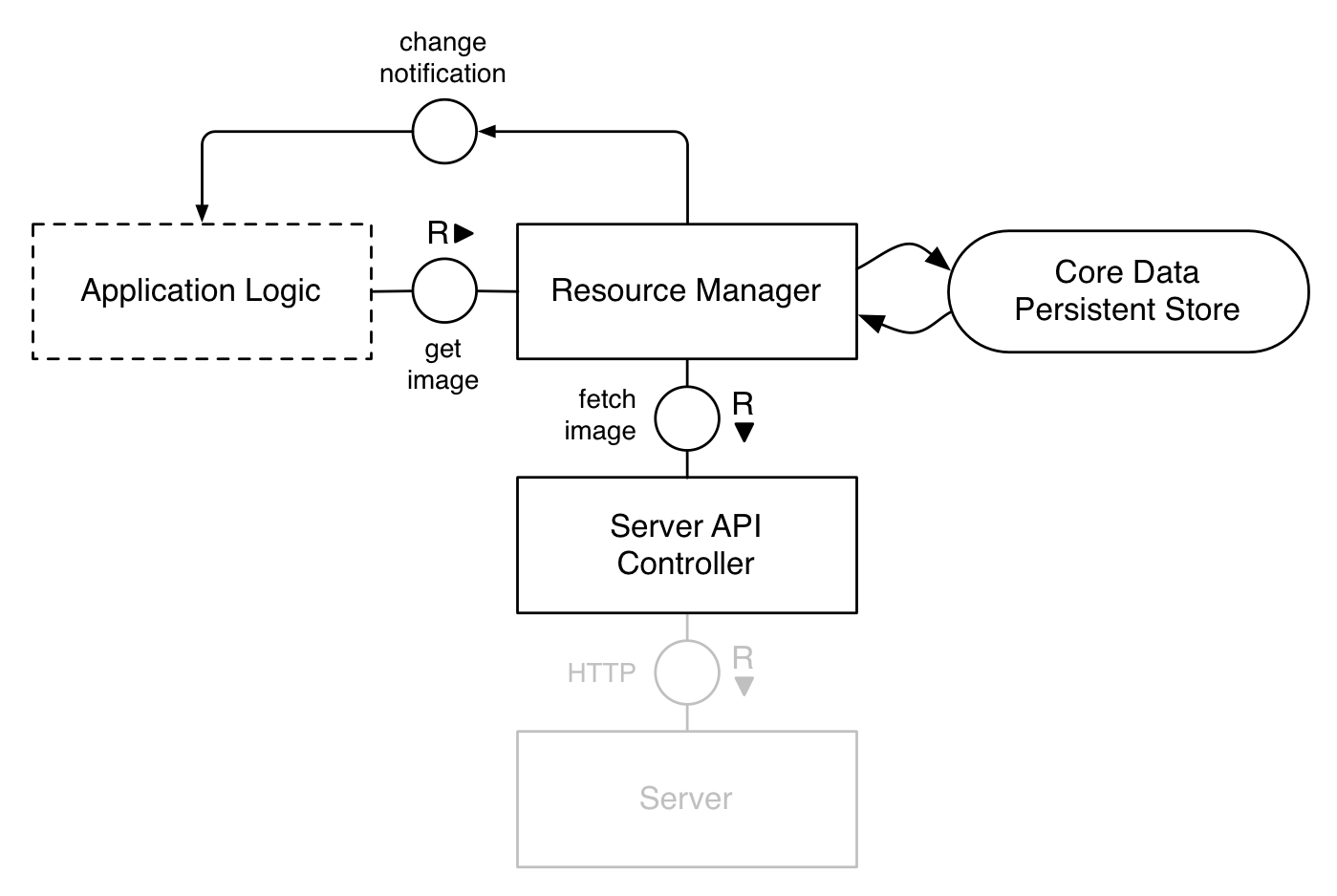
设计异步测试

就像同步测试一样，异步测试也应该比被测试的功能更简单。复杂的测试并不会改进代码的质量，反而会给测试本身带来Bug.在以测试驱动开发的情况下，简单的测试会让我们对组件，接口以及架构的行为有更清醒的认识。

示例工程

综上所述，我们创建了一个示例框架：PinacotecaCore，它从一个虚拟的服务器获取图像信息。框架中包含一个资源管理器，它对外暴露一个可以根据图像Id获取图像对象的接口，该接口的工作原理是资源管理器从服务器获取图片对象的信息，并更新到本地数据库。

虽然这个示例框架只是为了演示，但在我们自己开发的App中我们也是这么做的。



大致浏览了一下该框架后，让我们来详细了解一下。大体来讲，有三个组件我们需要测试：

1、模型层

2、服务器接口控制器（API Controller），包含所有对服务器的请求

3、资源管理器，管理core data堆栈，连接模型层和服务接口控制器

模型层

模型层应该尽量用同步的方式来测试。在不同的被管理的对象上下文中，只要没有太多的依赖，测试用例应该根据上下文在主线程上设置它自己的core data堆栈。在这个例子中，我们就是在setUp方法中设置core data堆栈，然后检查PCImage对象是否存在，不存在则构造一个，并更新它的值。当然这和异步测试没有关系，我们就不深入细说了。

服务器接口控制器

它主要处理服务器请求以及服务器API到模型的映射关系。让我们来看一下下面这个方法：

- [PCServerAPIController fetchImageWithId:queue:completionHandler:]

调用它需要传入一个图片对象Id，所在的执行队列以及一个完成后的回调方法。

当服务器根本不存在时，一个比较好的做法就是伪造一个代理服务器，正好OHHTTPStubs可以解决这个问题。在它的最新版本中，可以在请求响应中包含一个bundle，发送给客户端。

为了能接管请求，在测试类初始化时或者setUp方法中，对OHHTTPStubs进行配置。首先，我们需要加载一个包含请求响应对象（response）的bundle.

NSURL \*url = [[NSBundle bundleForClass:[self class]] URLForResource:@"ServerAPIResponses" withExtension:@"bundle"]; NSBundle \*bundle = [NSBundle url];

然后我们从bundle中加载请求响应对象,作为请求的响应值。

OHHTTPStubsResponse \*response; response = [OHHTTPStubsResponse responseNamed:@"images/123" fromBundle:responsesBundle responseTime:0.1]; [OHHTTPStubs stubRequestsPassingTest:^BOOL(NSURLRequest \*request) { return YES /\* true, if it's the expected request \*/; } withStubResponse:^OHHTTPStubsResponse \*(NSURLRequest \*request) { return response; }];

经过如上设置后，测试服务器接口控制器的简化版如下：

- (void)testFetchImageAsync {

[self.server fetchImageWithId:@"123" queue:[NSOperationQueue mainQueue]

completionHandler:^(id imageData, NSError \*error){

STAssertEqualObjects([NSOperationQueue currentQueue], queue, nil);

STAssertNil(error, [error localizedDescription]); STAssertTrue([imageData isKindOfClass:[NSDictionary class]], nil); // Check the values of the returned dictionary. STSuccess(); }]; STFailAfter(2.0, nil); }

资源管理器

它把服务器接口控制器和模型层联系了起来,而且还管理着core data堆栈。下面我们想测试获取一个图片对象的方法：

-[PCResourceManager imageWithId:usingManagedObjectContext:queue:updateHandler:]

该方法根据id返回一个图片对象。如果图片在数据库中不存在，它会创建一个只包含id的新对象，然后通过服务器接口控制器获取图片对象的详细信息。

资源管理器并不依赖服务器接口控制器，我们可以用OCMock来模拟,下面是代码实现：

OCMockObject \*mo; mo = [OCMockObject partialMockForObject:self.resourceManager.server];

id exp = [[serverMock expect] andCall:@selector(fetchImageWithId:queue:completionHandler:) onObject:self]; [exp fetchImageWithId:OCMOCK\_ANY queue:OCMOCK\_ANY completionHandler:OCMOCK\_ANY];

上面的代码实际上它并没有真正调用服务器接口控制器的方法，而是调用我们写在测试类中的方法。

用上面的做法，对资源管理的测试就变得很直观，当我们调用资源管理器获取资源时，实际上调用的是我们模拟的服务器接口控制器方法。这样我们也能检查调用服务器接口控制器时参数是否正确。在调用了获取图像对象的方法后，资源管理器会更新模型，然后回调我们测试中的回调函数。

- (void)testGetImageAsync {

NSManagedObjectContext \*ctx = self.resourceManager.mainManagedObjectContext;

\_\_block PCImage \*img;

img = [self.resourceManager imageWithId:@"123" usingManagedObjectContext:ctx queue:[NSOperationQueue mainQueue] updateHandler:^(NSError \*error){ // Check if the error is nil and // if the image has been updated. STSuccess(); }];

STAssertNotNil(img, nil);

STFailAfter(2.0, @"Timeout"); }

总结

测试并发设计模式开发的应用程序是一个挑战，但是一旦你理解了它们的不同，并建立最佳实践，一切都会变得简单而有趣。在nxtbgthng项目中，我们用SenTestingKitAsync框架来测试。但是像Kiwi 和GHUnit也都是不错的异步测试框架。你可以都试用一下，找到适合自己的测试工具。