异步系统的两种测试方法

原创 有赞技术 有赞coder 2018-04-08

文 | 孙军 on 测试

互联网软件系统一直随着需求、用户量上升等等的原因在演进，以求适应更复杂的业务场景，更高的性能要求等等。软件演进方式各种各样，系统异步化即为其中一种。

一般的，对于那些实时性要求不高，但却计算密集或者需要处理大数据量的耗时较长的任务，或是有较慢 I/O 的任务，选择异步化是一个不错的选择。在系统层面，像引入消息中间件来解耦系统，将耗时长的任务放在中间件后异步执行。在方法层面，像把耗时较长的任务放到其他线程中去异步执行。

与测试同步系统或方法不同，当我们测试异步系统（端到端测试、集成测试）或异步方法的时候（单元测试），由于测试线程不会被异步任务线程阻塞而让测试变得不可控，概率性失败，以单元测试为例，这样写异步测试是不稳定的：

```
@Test
public void testAsynchronousMethod() {
    callAsynchronousMethod();
    assertXXX(...);  //异步任务可能仍未完成，这时assert可能会失败
}
```

异步任务的两种类型：

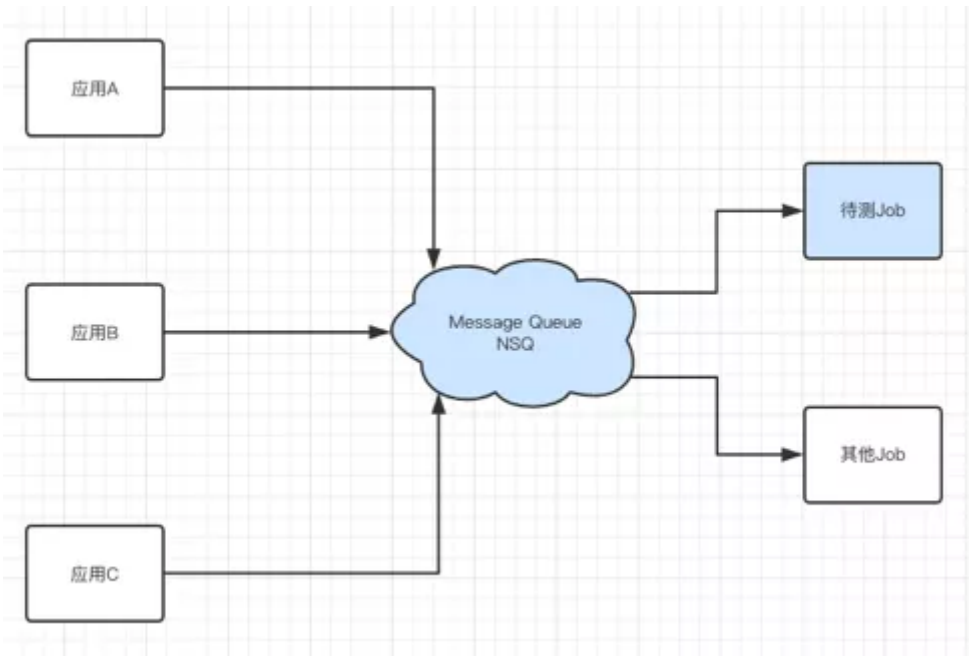
- 异步任务执行后对任务发起方或调用方有感知，比如发出一个事件或通知
- 异步任务执行后对任务发起方或调用方没有感知，只是改变了系统中的某些状态

对异步任务的测试也分以上两种类型讨论。对于第一种，我们可以采用 监听 方式测试：

```
import org.junit.Before;
import org.junit.Test;

public class ExampleTest {
    private final Object lock = new Object();

    @Before
    public void init() {
        new Thread(new Runnable() {
            public void run() {
                synchronized (lock) {  //获得锁
                    monitorEvent();    //监听异步事件的到来
                    lock.notifyAll();  //事件到达，释放锁
                }
            }
```

```
        }).start();
    }

    @Test
    public void testAsynchronousMethod() {
        callAsynchronousMethod();  //调用异步方法，需要较长一段时间才能执行完，并触发事件通知

        /**
         * 事件未到达时由于init已经获得了锁而阻塞，事件到达后因init中的锁释放而获得锁，
         * 此时异步任务已执行完成，可以放心的执行断言验证结果了
         */
        synchronized (lock) {
            assertTestResult();
        }
    }
}
```

这里的前提是事件通知会到来并被监听到，可要是不来呢（比如异常任务执行失败了）？我们就干等吗，其实我们还可以在测试中引入超时机制，这也引出了第二种类型的异常测试（可以称之为 轮询方式），假设我们有如下一个异步系统，应用发消息到 NSQ 消息中间件，一个待测试的 Job 监听这个消息并在消息到达后处理消息：



我们怎么测试呢，站在端到端测试的角度，可以测试从应用到 Job 的链路，消息是应用直接构造的 NSQ 消息，也可以是 Mysql binlog 经转化后构造的 NSQ 消息；站在集成测试的角度，我们可以缩小测试范围，直接在测试中构造 NSQ 消息，测试从消息中间件到 Job 的链路。长链路测试耗时长，且写测试前需要了解具体应用的消息触发逻辑，写测试也比较慢，无形中增加了很多测试成本。所以对于这样的系统，我们可以采用集成测试方法来测。

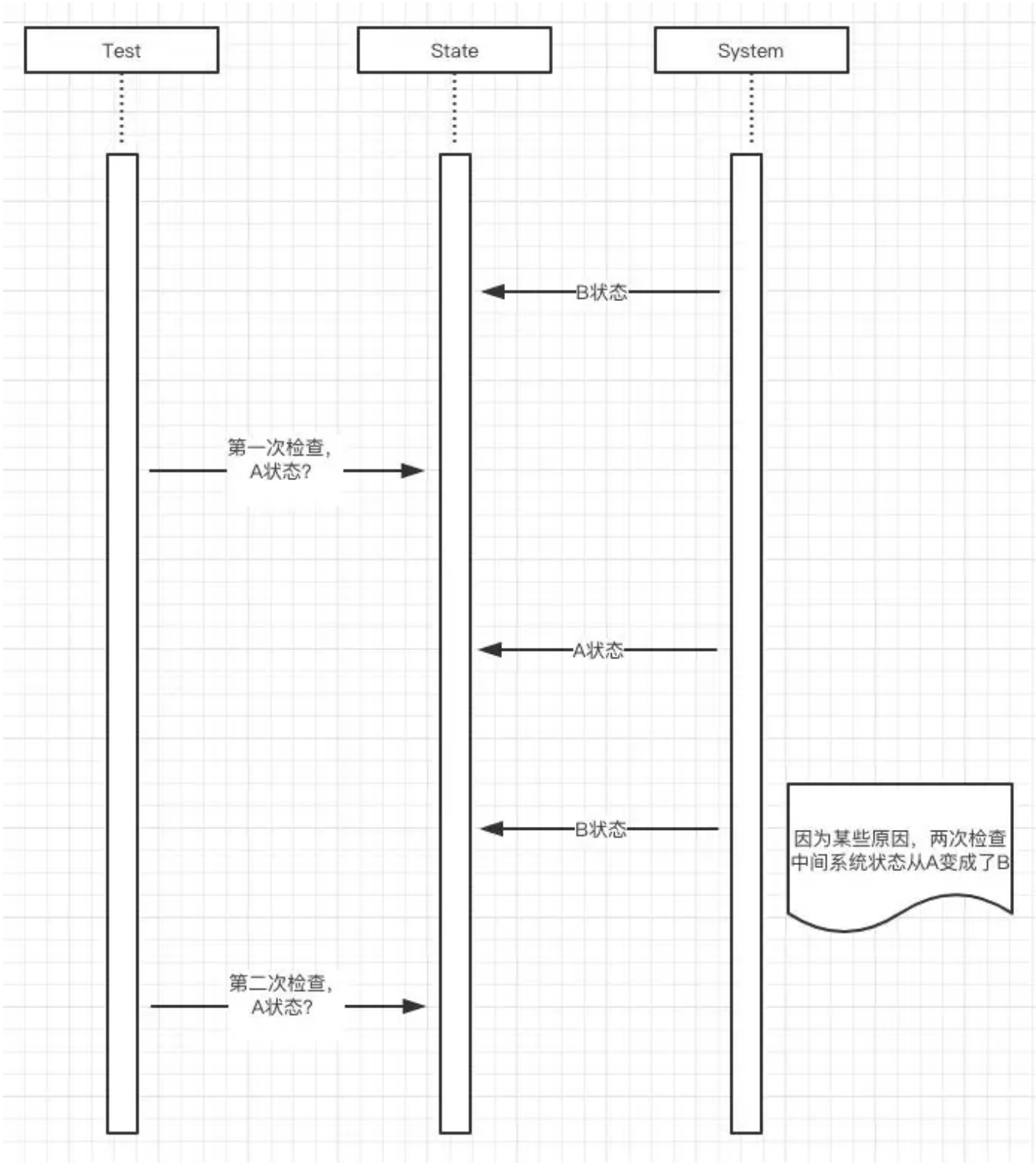
```
@Test
public void testAsynchronousJob() throws Exception {
    String msg = buildNsqMsg();     //构造NSQ消息
    nsqClient.send(TOPIC, msg, false);     //发送Nsq消息

    with().pollInterval(ONE_HUNDRED_MILLISECONDS).  //100ms后开始检查
            and().with().pollDelay(10, MILLISECONDS).  //此后每隔10ms检查一次
            await("description").  //描述信息
            atMost(1L, SECONDS).    //1s超时时间
            until(() -> xxxService.getState() == "changed");  //业务相关的断言逻辑
}
```

上述测试我们引入了 awaitility 工具类来做轮询操作，一个靠谱的轮询至少包含以下特性：

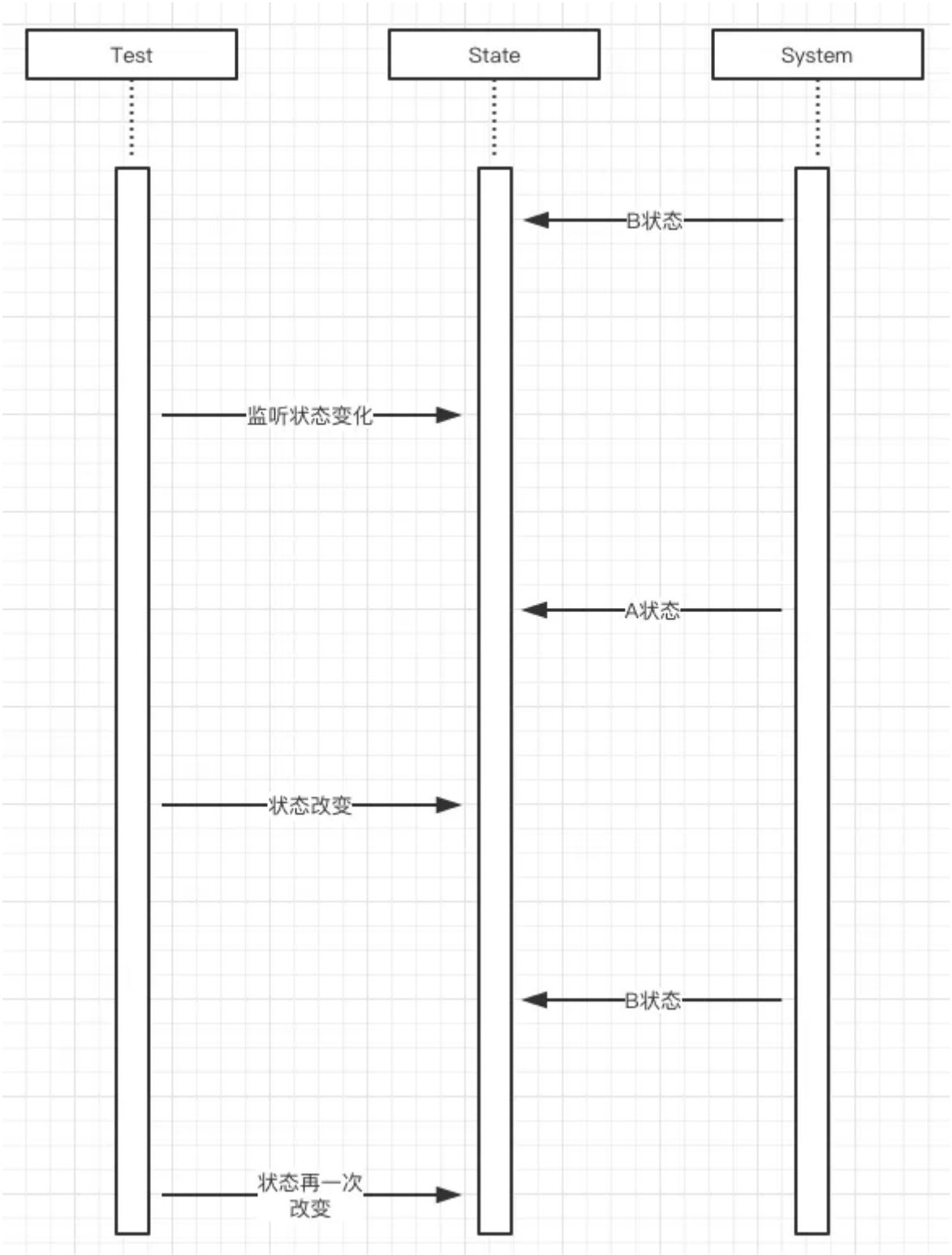
- 超时机制，不可能一直轮询
- 首次延迟轮询
- 轮询频率

最后，我们来讨论下测试结果可靠性问题。

假设一个异步系统采用轮询方式测试，触发异步任务后，当在两次轮询中间系统状态因为某些原因出现了抖动，下一次轮询时轮询方式可能会误以为异步操作还未完成或出现了异常，从而导致测试结果误判：



相对的，监听方式是不存在这样的问题的，只要系统状态改变，监听中的测试能立马感知到，并作出可靠的测试结果：



很多异步系统对外是没有回调的，这时候只能使用轮询方式测试异步任务，而轮询测试的可靠性取决于待测系统的可靠性。

可是，一个周期性执行的可靠系统同样会遇到上述问题，测试会因为代码周期性执行，系统状态周期性改变而变得不可靠。对于这样的系统，我们可以做一些可测性改造。将业务逻辑和周期执行逻辑剥离，并增加一个可以调用业务逻辑的入口，比如一个 restful 接口，这样测试时可以准确控制业务逻辑的执行时机和频率，也就可以可靠的测试了。

有赞已经在一些异步 Job 中采用上述轮询方式测试，我们在测试中主要碰到了两类 Job，一类是会和 Elasticsearch 搜索引擎交互的，由于 Elasticsearch 的刷新机制（有赞出于性能原因设置为 5 秒刷新一次数据），轮询方式因为测试时间太长而很局限，除非提高 Elasticsearch 的刷新频率；另一类则是跟 Mysql、Redis 交互的 Job，这类 Job 的测试可以工作的很好，测试基本可以在 150 毫秒内完成，也就意味着可以像普通测试一样置入持续集成的构建中了。