链滴

selenium 自动化监控实践

作者：xjlnjut730

原文链接：https://ld246.com/article/1544531672656

来源网站：链滴

最近公司有一套页面需要使用selenium来监控，确保页面功能在频繁发布之后从使用上正常的。经过两周的迭代，目前已经基本成形了，监控了近30个页面的近60个功能点。这里为这个项目做个总结。

业务梳理

接到任务的时候，业务方要求提供的报警信息是这样的：

页面 http://www.xx.com/xxx.html 访问异常，状态码：xxxx @所有人

一个资深的研发人员，接到需求，必须结合业务场景进行全面的分析与设计。针对这个需求，我觉得少需要考虑以下几点：

1. 什么情况下才算正常?
2. 测试使用的信息是否有限制？尤其像手机号、邮箱这种，怎么规定测试的资源?
3. 这个报警信息充分么?是否有充足的信息方便问题诊断?
4. 一个页面频繁访问是否会对正常业务造成影响?

接下来，我们进行逐个分析：

测试通过标准

这个很重要，需要根据这个标准来设计程序，必须跟业务方针对功能点进行逐个确认。这里由于涉及公司业务，就不深入展开，总体来说，使用了以下4种方式：

1. 访问URL成功
2. 表单提交，是否跳转至成功页面
3. 页面是否展示了成功元素
4. 页面某个元素是否已经不存在

测试资源限制

这个也需要跟业务方确认，因为有些页面必须用到手机号、邮箱、用户名、身份证号等业务数据。这数据如果不与正常的业务加以区分，很有可能会涉及到短信的推送、客服的跟进等，会对业务产生负的影响。测试用的手机号可以规定某一个号码或者某一个号段。前期我们使用某一个不常见的号段来行区分，经过后续的业务评估，觉得小概率事件仍然存在。所以最终，还是选择了一个不太可能使用手机号来进行测试，避免问题。

异常通知模板

一旦页面发现问题，究竟需要提供多少信息，这是个需要仔细考虑的问题。一个状态码肯定是不够的程序也很难精确定义各种状态码。根据Web开发多年的经验来看，遇到一个页面问题，我们一般会需去看这个页面当前展示的形式，也会去看控制台的输出，还会看网络传输的情况，还有页面的源码。般情况下，页面当前的截图、控制台输出、网络传输日志、页面源码、故障描述可以基本确定问题的因。所以我们确定了异常通知模样如下：

监控页面: http://xxxxxx/xxx.html
监控功能: 抢红包
错误描述: 抢红包后，20秒内未跳转至成功页面/未提示失败!
截图链接: http://xxxxxxxxx/snapshot/32356b27f5.zip
页面源码: http://xxxxxxxxx/source/1544090935036.txt
详细日志: http://xxxxxxxxx/log/1544090935036.log
监控耗时: 27522ms. @所有人

注意到，截图是一个打包的形式，里面包含了多张图片。在监控逻辑执行过程中，每一帧的截图会保下来，方便定位。当然也可以使用录屏的方式，但是开发起来会比较麻烦。后续会讲一下监控过程中图是如何实现的。

监控速度控制

一开始我们没有对监控做速度上的控制，5分钟跑一遍用例，给应用一天带来了将近1w多的监控数据有部分业务处理的比较慢，导致一些这些数据没能及时处理，一直在处理测试数据。一些需要当天通客户的短信、邮件也没能及时发出去，对业务产生了比较大的影响。最后，我们降低了监控速度，改了1小时1次，降低了访问频率。

Java or Python

为什么会有这个困扰呢？因为我在之前公司做这方面的工作时，采用的是python，开发起来效率比高，但新公司的技术栈基本都是Java。考虑到以下因素，最终采用了Java来操作：

1. 技术栈统一，可以多人协作开发，交流也比较顺畅。
2. Java本身的优势，拥有Spring Boot/Maven等强大的工具与库，开发效率并不逊色于Python。
3. 有IDEA，Java的重构也非常方便。
4. 考虑过用Go，但是觉得应用场景上Go并不太适合。

其实，无论哪种语言最终都可以实现页面监控的功能，所以必须根据团队的实际情况选择对应的技术。

集成spring

Selenium其实并不需要与Spring做集成，网上找的很多类似的文章，更多的是利用SpringBootTest做测试集成。这个项目并没有利用SpringBootTest来做，而是基于Quartz定时任务来做的，主要还因为需要定时去执行脚本。我这里说的与Spring集成，主要还是将ChromeDriver与WebDriverWait个实例交给Spring来托管，以实现一些特殊功能，后面会阐述。核心的类主要包括两个Proxy与一个Hlder，如下：

```
public class ChromeDriverProxy extends ChromeDriver {

    private String uuid;
    private Boolean snapshotWhenPossible;
    private Boolean cleanSnapshotWhenQuit;

    private Logger logger = Logger.getLogger(ChromeDriverProxy.class);

    public ChromeDriverProxy(ChromeOptions options) {
        super(options);
        uuid = UUID.randomUUID().toString().replace("-", "").substring(22);
    }

    // ...
}
// 每一个用例需要新的wait与driver对象，所以scope必须是SCOPE_PROTOTYPE
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class WebDriverWaitProxy extends WebDriverWait {
```

```
    private static final Integer WAIT_TIMEOUT = 20;

    private ChromeDriverProxy driver;

    public WebDriverWaitProxy(ChromeDriverProxy driver) {
        super(driver, WAIT_TIMEOUT);
        this.driver = driver;
    }

    // ...
}

// 每一个用例需要新的wait与driver对象，所以scope必须是SCOPE_PROTOTYPE
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class WebDriverHolderProxy {

    @Autowired
    private WebDriverHolderProxy wait;


    public WebDriverHolderProxy getWait() {
        return wait;
    }

    public ChromeDriverProxy getDriver() {
        return wait.getDriver();
    }
}
```

注意到上面的ChromeDriverProxy没有加注解，是因为ChromeDriverProxy不能直接用Spring来创，因为其构造方法需要传一个特定的ChromeOptions，怎么办呢？Spring还有使用FactoryBean来建实例。下面是ChromeDriverProxyFactoryBean：

```
@Component
public class ChromeDriverProxyFactoryBean implements FactoryBean<ChromeDriverProxy>,
eanPostProcessor {
    // chromeDriver默认延时
    public static final Integer DEFAULT_WAIT_TIMEOUT = 10;

    @Value("${webdriver.chrome.driver}")
    private String driverPath;

    @Value("${browser.maximize}")
    private Boolean shouldMaximize;

    @Value("${snapshot.when.possible}")
    private Boolean snapshotWhenPossible;

    private static ChromeOptions DEFAULT_OPTIONS = new ChromeOptions();

    static {
        LoggingPreferences preference = new LoggingPreferences();
        // 开始浏览器与性能日志
```

```
        preference.enable(LogType.BROWSER, Level.ALL);
        preference.enable(LogType.PERFORMANCE, Level.ALL);

        DEFAULT_OPTIONS.setCapability(CapabilityType.LOGGING_PREFS, preference);
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansE
xception {
        //factoryBean初始化完成后，设置chrome.driver路径
        System.setProperty("webdriver.chrome.driver", driverPath);
        // 加上，否则无法截图
        System.setProperty("java.awt.headless", "false");
        return bean;
    }

    /**
     * 初始化一个chromeDriver，主要包括：窗口最大化、日志配置、默认超时时间
     * @return
     */
    @Override
    public ChromeDriverProxy getObject() throws Exception {
        ChromeDriverProxy driver = new ChromeDriverProxy(DEFAULT_OPTIONS);

        driver.setSnapshotWhenPossible(snapshotWhenPossible);

        driver.setLogLevel(Level.ALL);
        driver.manage().timeouts().implicitlyWait(DEFAULT_WAIT_TIMEOUT, TimeUnit.SECONDS);
        if(shouldMaximize) {
            driver.manage().window().maximize();
        }

        return driver;
    }

    @Override
    public Class<?> getObjectType() {
        return ChromeDriverProxy.class;
    }
    // 这里指定了Scope，每次引用需要单独创建
    @Override
    public boolean isSingleton() {
        return false;
    }
}
```

然后就是用Quartz执行定时任务：

```
@Component
public class MonitorTask {

    @Autowired
    private MonitorExecute monitorExecute;
```

```
    private static Logger logger = Logger.getLogger(MonitorTask.class);
    // 重试次数
    private static final Integer MONITOR_RETRY_LIMIT = 2;

    // 用例之间间隔
    @Value("60")
    private Integer waitSecondsAfterCase;

    /**
     * 每小时执行一次监控关键业务
     */
    @Scheduled(initialDelay = 10 * 1000L, fixedRateString= "3600000")
    public void monitorPage() {
        logger.info("monitorPage start.");
        long startTime = System.currentTimeMillis();

        for(Map.Entry<String, Class<? extends MonitorCase>> entry : MonitorUrls.ZXBJ_URL_CA
E_MAP.entrySet()) {
            Class<? extends MonitorCase> executeClass = entry.getValue();
            String url = entry.getKey();

            try {
                // 执行测试用例，如果发现异常，则重试，重试次数达到阈值后仍然失败，则发送通知
                monitorExecute.doExecuteCase(executeClass, url, MONITOR_RETRY_LIMIT);

                TimeUnit.SECONDS.sleep(waitSecondsAfterCase);
            } catch (Exception e) {
                logger.error("url:" + url + ",executeClass:" + executeClass + " execute error!", e);
            }
        }

        logger.info("monitorPage end.cost:" + (System.currentTimeMillis() - startTime) / 1000 + "
.");
    }
}

/**
 * 用例执行服务
 * User: francis.xjl@qq.com
 * Create Time: 2018/12/5 13:57
 */
@Component
public class MonitorExecute {

    private static Logger logger = Logger.getLogger(MonitorExecute.class);

    private static final Integer SLEEP_SECONDS = 5;

    @Autowired
    private DingService dingService;

    @Autowired
    private ApplicationContext applicationContext;
```

```
    // 执行测试用例，如果发现异常，则重试，重试次数达到阈值后仍然失败，则发送通知
    public void doExecuteCase(Class<? extends MonitorCase> executeClass, String url, int retry
imit) {
        MonitorCase monitorCase = applicationContext.getBean(executeClass); // 这里必须proto
ype，否则driver会出问题
        MonitorResultMessage message = monitorCase.monitor(url);
        // 用例执行正常直接返回
        if(message.isSuccess()) {
            return;
        }

        retryLimit--;

        logger.warn(String.format("url:%s, retryLimit:%s failed. message:%s", url, retryLimit, mess
ge));
        // 重试次数达到阈值后仍然失败，则通知
        if(retryLimit <= 0) {
            dingService.notifyUrlUnusualWithSnapshot(message);
            return;
        }
        // 5秒后才重试
        try {
            TimeUnit.SECONDS.sleep(SLEEP_SECONDS);
        } catch (InterruptedException e) {}

        doExecuteCase(executeClass, url, retryLimit);
    }

}
```

开发测试用例的人员只要去实现MonitorCase就可以了，MonitorCase接口与实现如下所示：

```
/**
 * 监控脚本的抽象接口
 * User: francis.xjl@qq.com
 * Create Time: 2018/12/3 9:59
 */
public interface MonitorCase {

    /**
     * 一个监控脚本的主要运行逻辑在这里面，监控的页面通过参数传入，以便一个脚本能够支持多个
面。
     * @param url 监控的URL
     */
    MonitorResultMessage monitor(String url);
}


@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class MonitorCaseDemo implements MonitorCase {

    private Logger logger = Logger.getLogger(MonitorCaseDemo.class);
```

```
    @Autowired
    private WebDriverHolderProxy holder;

    @Override
    public MonitorResultMessage monitor(String url) {
        String mobile = Mobiles.instanceTestMobile();
        ChromeDriverProxy driver = holder.getDriver();
        WebDriverWait wait = holder.getWait();
        MonitorResultMessage message = new MonitorResultMessage(url, "领红包");

        try {
            // ...操作
            isSuccess = ...;

            if(!isSuccess) {
                message.setErrMsg("XXX功能测试未成功");
                Browsers.saveSceneData(driver, message);
            } else {
                driver.setCleanSnapshotWhenQuit(true);
            }
            message.setSuccess(isSuccess);
        } catch (Exception e) {
            message.setException(e);
            Browsers.saveSceneData(driver, message);
        } finally {
            Browsers.quitQuietly(driver);
        }

        return message;
    }
}
```

以上就是全部的Spring与Selenium整合内容，已经达到了预期。要特别注意ChromeDriver每次都必重新创建，否则测试之间会存在依赖，如果觉得ChromeDriver的频繁启动比较慢的话，可以使用ChrmeDriverService来实现对ChromeDriver的启停。

截图的处理

上面说到过，我们希望在执行过程中截图，又不想在每段代码前后加入截图的逻辑，那怎么操作呢？了上面Spring与Selenium整合的基础，方法就多了，可以使用AOP，也可以直接在对应的Proxy中对定方法执行前后来截图。本项目中使用的是后面的方法，在ChromeDriverProxy的findElement与WeDriverWaitProxy的until方法前后执行截图逻辑，且在ChromeDriverProxy的quit方法里完成对截图清理。可以满足大部分场景要求：

```
public class ChromeDriverProxy extends ChromeDriver {
    //...
    @Override
    public WebElement findElement(By by) {
        try {
            if(snapshotWhenPossible) {
                Browsers.saveSnapshotQuietly(this, uuid + "-" + System.currentTimeMillis() + "-FE1"
;
            }
```

```
            return super.findElement(by);
        } finally {
            Browsers.saveSnapshotQuietly(this, uuid + "-" + System.currentTimeMillis() + "-FE2");
        }
    }
    //...

    @Override
    public void quit() {
        super.quit();
        if(cleanSnapshotWhenQuit) {
            this.cleanSnapshot();
        }
    }

    /**
     * 清理driver自动生成的截图
     */
    private void cleanSnapshot() {
        String uuid = this.getUuid();
        String folder = MonitorApplication.CONTEXT.getEnvironment().getProperty("snapshot.sa
e.dir");
        String snapshotTmpDir = MonitorApplication.CONTEXT.getEnvironment().getProperty("s
apshot.tmp.dir");

        for(File file : FileUtils.listFiles(new File(folder), null, false)) {
            if(file.getName().contains(uuid)) {
                FileUtils.deleteQuietly(file);
            }
        }

        // clean windows的临时文件，否则会撑满硬盘
        for(File file : FileUtils.listFiles(new File(snapshotTmpDir), new String[]{"png"}, false )) {
            if(file.getName().startsWith("screenshot")) {
                FileUtils.deleteQuietly(file);
            }
        }

        logger.warn(uuid + " snapshots was cleaned!");
    }
}

@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class WebDriverWaitProxy extends WebDriverWait {
    //...
    @Override
    public <V> V until(Function<? super WebDriver, V> isTrue) {
        String name = driver.getUuid();

        try {
            if(driver.getSnapshotWhenPossible()) {
                Browsers.saveSnapshotQuietly(driver, name + "-" + System.currentTimeMillis() + "-
1");
```

```
            }
            return super.until(isTrue);
        } finally {
            if(driver.getSnapshotWhenPossible()) {
                Browsers.saveSnapshotQuietly(driver, name + "-" + System.currentTimeMillis() + "-
2");
            }
        }
    }
    //...
}
```

功能是OK了，但这种方式由于所有场景都会截图，对IO性能影响还是会有的，谨慎使用，由于本项并发不高，所以压力不太大。也可以考虑再对重复的截图做一些优化，避免不必要的IO。

下面是具体的截图逻辑，其实很简单：优先使用Robot截图，判断图片是黑屏后，再改用getScreensotAs进行截图。采用这样的策略主要是因为：getScreenshotAs方式有时会因为弹出框报错，为了尽能保证这时候也能截图，所以优先使用Robot截图。但Robot截图在屏幕灰掉的时候截取的是黑屏，以需要再做进一步判断。（如果监控的服务器没有屏幕，可以考虑反过来的逻辑，以避免过多的分支）

```
public static String saveSnapshotQuietly(ChromeDriver driver, String folder, String fileNamePr
fix) {
    if(driver == null) {
        logger.error("驱动为空，无法截图！");
        return "";
    }

    String fileName = fileNamePrefix + ".png";
    try {
        Dimension dimension = driver.manage().window().getSize();
        Point point = driver.manage().window().getPosition();
        Rectangle rect = new Rectangle(point.x, point.y, dimension.width, dimension.height);
        // 能用robot就用robot，因为getScreenshotAs不能绕过弹出框截图，可能存在问题
        // 但是如果没有桌面开着，robot也可能会截出黑屏；如果是黑屏，再尝试用getScreenshotAs
图
        BufferedImage img = new Robot().createScreenCapture(rect);
        if(!isAllBlack(img)) { //判断是否全黑，以确认截图是否正常
            ImageIO.write(img, "png", new File(folder + "/" + fileName));
            return fileName;
        }

        // 使用robot截图异常，则尝试使用chrome自身的截图功能
        logger.error("使用robot截图显示为全黑，尝试使用chrome自身的截图功能!");
        fileName = "chrome-" + fileName;

        // 如果有弹出框，暂时将弹出框内容记录在日志里
        switchToAlertIfExist(driver);

        File srcFile = driver.getScreenshotAs(OutputType.FILE);// 执行屏幕截取
        FileUtils.copyFile(srcFile, new File(folder, fileName));
    } catch (Exception e) {
        logger.error("截图时出现异常：" + e.getMessage(), e);
        return "";
```

```
    }

    return fileName;
}
```

尽量不要使用sleep

这个是刚开始接触selenium很容易犯的问题。那为什么不能这么做？直接的原因是为了尽可能避免问，所以sleep的时间伊往往设计得很长，这会延长脚本执行的时间，执行效率会低。间接的，很多情下，你是不需要使用sleep的，是因为不熟悉selenium，其实很多情况往往都有很多替代的方式。以是一些可以使用的常见方案：

1. wait.until(ExpectedConditions.visibilityOfElementLocated 等待元素可见
2. wait.until(ExpectedConditions.elementToBeClickable 等待元素可以点击（可用）
3. wait.until(ExpectedConditions.urlContains 等待URL包含
4. wait.until(ExpectedConditions.or 等待多个条件中出现一个
5. wait.until(ExpectedConditions.invisibilityOfElementLocated 等待元素不可见

其中第2点，也可用于判断输出框是否可填。刚开始接触的时候，我只知道1，不知道2，所以有些元可见了，但点击不了，用了sleep，部署到服务器上，由于网络原因，还是一直会出现明明元素已经见了，但是触发不了点击事件。还是要多去熟悉一下API。