目 录

Contents

2016 年 08 月

联系邮箱：editor@51testing.com

Android 内存泄漏从入门到精通

◆作者：姚潮生

一、基本概念

1、首先以一个内存泄露实例来开始本节基础概念的内容:

实例 1: (单例导致内存对象无法释放而泄露)

```
@Override
protected void onResume() {
    super.onResume();

    LogRecordUtil a = LogRecordUtil.getInstance(this);
}


 public static LogRecordUtil getInstance(Context c){
     if(instance == null){
         instance = new LogRecordUtil(c);
     }
     return instance;
 }
```

可以看出 LogRecordUtil 这个工具类是一个单例，并引用了 activity 的 context。

试想这个场景，应用起来以后，转屏。转屏以后，旧 Activity 会 destroy，新 Activity 会重建，导致单例 LogRecordUtil 重新 getInstance。很不幸的是，由于 instance 已经不是空的了，所以 LogRecordUtil 不会重建，还持有之前的 context，也就是之前的那个 Activity 实例的 context，因此会造成两个问题:

功能问题: 使用 LogRecordUtil 访问 context 相关内容时可能会发生异常（因为当前 context 并不是当前 activity 的 context）;

内存泄露: 旧 context 被生命周期更长的静态变量持有而导致 activity 无法释放造成泄漏!（因此静态变量是很容易因此内存泄露的!）

2、内存泄露，我们要研究的泄露对象到底是什么？

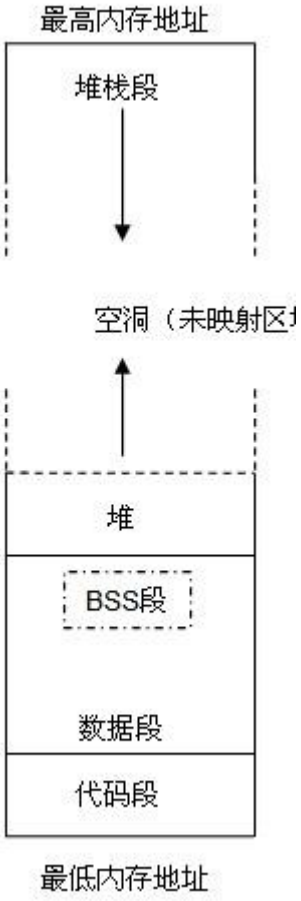
(1) 首先我们来了解程序运行时，所需内存的分配策略:

按照编译原理的观点，程序运行时的内存分配有三种策略，分别是静态的，栈式的，和堆式的，对应的，三种存储策略使用的内存空间主要分别是静态存储区（也称方法区）、堆区和栈区。他们的功能不同，对他们使用方式也就不同。

静态存储区（方法区）: 内存在程序编译的时候就已经分配好，这块内存在程序整个运行期间都存在。它主要存放静态数据、全局 static 数据和常量。

栈区: 在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

堆区: 亦称动态内存分配。程序在运行的时候用 malloc 或 new 申请任意大小的内存，程序员自己负责在适当的时候用 free 或 delete 释放内存（Java 则依赖垃圾回收器）。动态内存的生存期可以由我们决定，如果我们不释放内存，程序将在最后才释放掉动态内存。 但是，良好的编程习惯是：如果某动态内存不再使用，需要将其释放掉。



(2) 接下来我们集中说下堆和栈的区别:

在函数中（说明是局部变量）定义的一些基本类型的变量和对象的引用变量都是在函数的栈内存中分配。当在一段代码块中定义一个变量时，java 就在栈中为这个变量分配内存空间，当超过变量的作用域后，java 会自动释放掉为该变量分配的内存空间，该内存空间可以立刻被另作他用。

堆内存用于存放所有由 new 创建的对象（内容包括该对象其中的所有成员变量）和数组。在堆中分配的内存，由 java 虚拟机自动垃圾回收器来管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，在栈中的这个特殊的变量就变成了数组或者对象的引用变量，以后就可以在程序中使用栈内存中的引用变量来访问堆中的数组或者对象，引用变量相当于为数组或者对象起的一个别名，或者代号。

堆是不连续的内存区域（因为系统是用链表来存储空闲内存地址，自然不是连续的），堆大小受限于计算机系统中有效的虚拟内存（32bit 系统理论上是 4G），所以堆的空间比较灵活，比较大。栈是一块连续的内存区域，大小是操作系统预定好的，

windows 下栈大小是 2M（也有是 1M，在编译时确定，VC 中可设置）。

对于堆，频繁的 new/delete 会造成大量内存碎片，使程序效率降低。对于栈，它是先进后出的队列，进出一一对应，不产生碎片，运行效率稳定高。

举一个关于变量存储位置的实例 2:

```
public class A{
    int a = 1;
    person p = new person();

    public void method(){
        int a2 = 1;
        person p2 = new person();
    }
}
A aObject = new A();
```

A类内的局部变量都存在于栈中，包括基本数据类型a2和对象引用p2。
而p2指向的person对象是存在于堆中。

aObject对象实体存放在堆中，包括这个对象的所有成员变量a和p；
而p这个引用指向的person类对象也是存在堆中。
而aObject这个引用存在与栈中。

(3) 结论:

局部变量的基本数据类型和引用存储于栈中，引用的对象实体存储于堆中。

——因为它们属于方法中的变量，生命周期随方法而结束。

成员变量全部存储与堆中（包括基本数据类型，引用和引用的对象实体）

——因为它们属于类，类对象终究是要被 new 出来使用的。

回到我们的问题：内存泄露需要关注的是什么？

我们这里说的内存泄露，是针对，也只针对堆内存，他们存放的就是引用指向的对象实体（包括其中的所有成员变量)!!!!

3、那么第二个问题就是，内存为什么会泄露？

为了判断 Java 中是否有内存泄露，我们首先必须了解 Java 是如何管理（堆）内存的。Java 的内存管理就是对象的分配和释放问题。在 Java 中，内存的分配是由程序完成的，而内存的释放是由垃圾收集器(Garbage Collection，GC)完成的，程序员不需要通过调用函数来释放内存，但它只能回收无用并且不再被其它对象引用的那些对象所占用的

空间。

Java 的内存垃圾回收机制是从程序的主要运行对象(如静态对象/寄存器/栈上指向的堆内存对象等)开始检查引用链，当遍历一遍后得到上述这些无法回收的对象和他们所引用的对象链，组成无法回收的对象集合，而其他孤立对象（集）就作为垃圾回收。GC 为了能够正确释放对象，必须监控每一个对象的运行状态，包括对象的申请、引用、被引用、赋值等，GC 都需要进行监控。监视对象状态是为了更加准确地、及时地释放对象，而释放对象的根本原则就是该对象不再被引用。

在 Java 中，这些无用的对象都由 GC 负责回收，因此程序员不需要考虑这部分的内存泄露。虽然，我们有几个函数可以访问 GC，例如运行 GC 的函数 System.gc()，但是根据 Java 语言规范定义，该函数不保证 JVM 的垃圾收集器一定会执行。因为不同的 JVM 实现者可能使用不同的算法管理 GC。通常 GC 的线程的优先级别较低。JVM 调用 GC 的策略也有很多种，有的是内存使用到达一定程度时，GC 才开始工作，也有定时执行的，有的是平缓执行 GC，有的是中断式执行 GC。但通常来说，我们不需要关心这些。

至此，我们来看看 Java 中需要被回收的垃圾:

```
{
Person p1 = new Person();
}
```

引用句柄 P1 的作用域是从定义到“} ”处，执行完这对大括号中的所有代码后，产生的 Person 对象就会变成垃圾，因为引用这个对象的句柄 P1 已超过其作用域，P1 失效，在栈中被销毁，因此堆上的 Person 对象不再被任何句柄引用了。因此 person 变为垃圾，会被回收。

从上面的例子和解释，可以看到一个很关键的词：引用。

通俗的讲，通过 A 能调用并访问到 B，那就说明 A 持有 B 的引用，或 A 就是 B 的引用，B 的引用计数+1.

（ 1 ）比如 Person p1 = new Person(); 通过 P1 能操作 Person 对象，因此 P1 是 Person 的引用;

（ 2 ）比如类 O 中有一个成员变量是 I 类对象，因此我们可以使用 o.i 的方式来访

问 I 类对象的成员，因此 o 持有一个 i 对象的引用。

GC 过程与对象的引用类型是严重相关的，我们来看看 Java 对引用的分类 Strong reference，SoftReference，WeakReference，PhatomReference

级别	回收时机	用途	生存时间
强	从来不会	对象的一般状态	JVM 停止运行时终止
软	在内存不足时	联合 ReferenceQueue 构造有效期短/占内存大/生命周期长的对象的二级高速缓冲器（内存不足才清空）	内存不足时终止
弱	在垃圾回收时	联合 ReferenceQueue 构造有效期短/占内存大/生命周期长的对象的一级高速缓冲器（系统发生 gc 则清空）	gc 运行后终止
虚	在垃圾回收时	联合 ReferenceQueue 来跟踪对象被垃圾回收器回收的活动	gc 运行后终止

讲多一步，这里的软引用/弱引用一般是做什么的呢？

在 Android 应用的开发中，为了防止内存溢出，在处理一些占用内存大而且声明周期较长的对象时候，可以尽量应用软引用和弱引用技术。

软/弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java 虚拟机就会把这个软引用加入到与之关联的引用队列中。利用这个队列可以得知被回收的软/弱引用的对象列表，从而为缓冲器清除已失效的软/弱引用。

另外可以根据对象是否经常使用来判断选择软引用还是弱引用。如果该对象可能会经常使用的，就尽量用软引用。如果该对象不被使用的可能性更大些，就可以用弱引用。

回到我们的问题，为什么内存会泄露？

堆内存中的长生命周期的对象持有短生命周期对象的强/软引用，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，这就是 Java 中内存泄露的根本原因。

二、排查方式

1、最原始的内存泄露测试

重复多次操作关键的可疑的路径，从内存监控工具中观察内存曲线，是否存在不断上升的趋势且不会在程序返回时明显回落。

这种方式可以发现最基本，也是最明显的内存泄露问题，对用户价值最大，操作难度小，性价比极高。

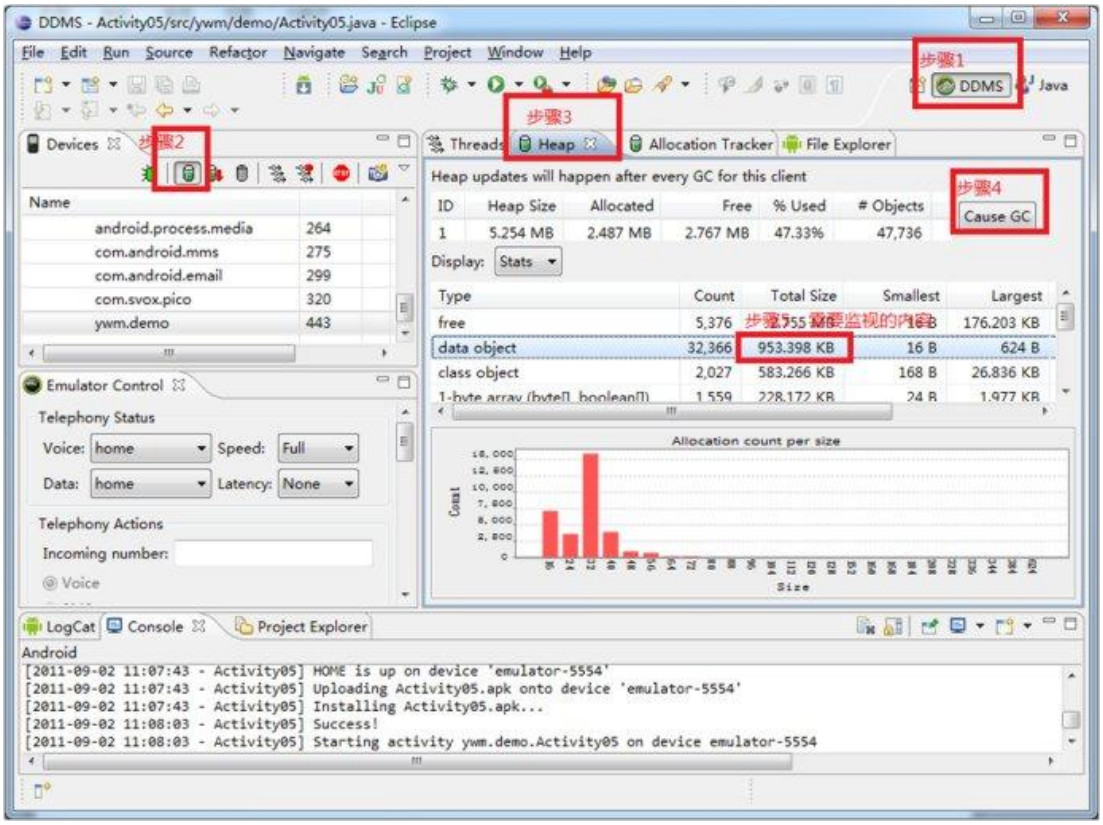
2、MAT 内存分析工具

（1）MAT 分析 heap 的总内存占用大小来初步判断是否存在泄露

在 Devices 中，点击要监控的程序。

- 点击 Devices 视图界面中最上方一排图标中的“Update Heap”
- 点击 Heap 视图
- 点击 Heap 视图中的“Cause GC”按钮

到此为止需检测的进程就可以被监视。



Heap 视图中部有一个 Type 叫做 data object，即数据对象，也就是我们的程序中大量存在的类类型的对象。在 data object 一行中有一列是“Total Size”，其值就是当前进程中所有 Java 数据对象的内存总量，一般情况下，这个值的大小决定了是否会有内存泄漏。可以这样判断：

进入某应用，不断的操作该应用，同时注意观察 data object 的 Total Size 值

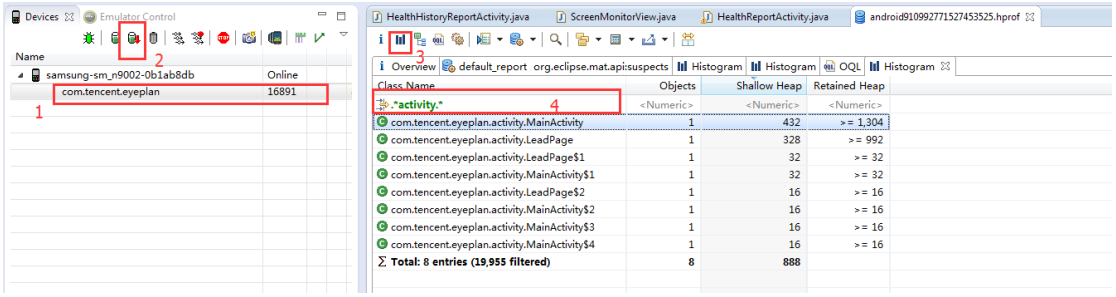
正常情况下 Total Size 值都会稳定在一个有限的范围内，也就是说由于程序中的的代码良好，没有造成对象不被垃圾回收的情况，

所以说虽然我们不断的操作会不断的生成很多对象，而在虚拟机不断的进行 GC 的过程中，这些对象都被回收了，内存占用量会会落到一个稳定的水平；反之如果代码中存在没有释放对象引用的情况，则 data object 的 Total Size 值在每次 GC 后不会有明显的回落。随着操作次数的增多 Total Size 的值会越来越大，直到到达一个上限后导致进程被杀掉。

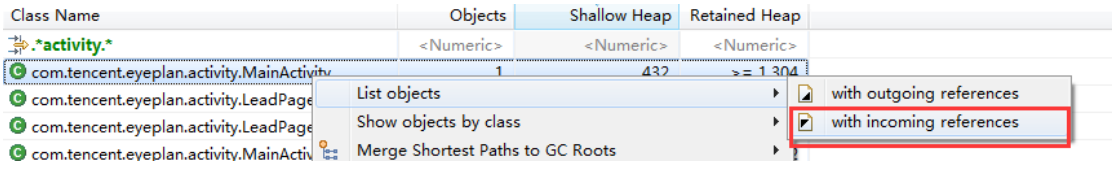
（2）MAT 分析 hprof 来定位内存泄露的原因所在。

这是出现内存泄露后使用 MAT 进行问题定位的有效手段。

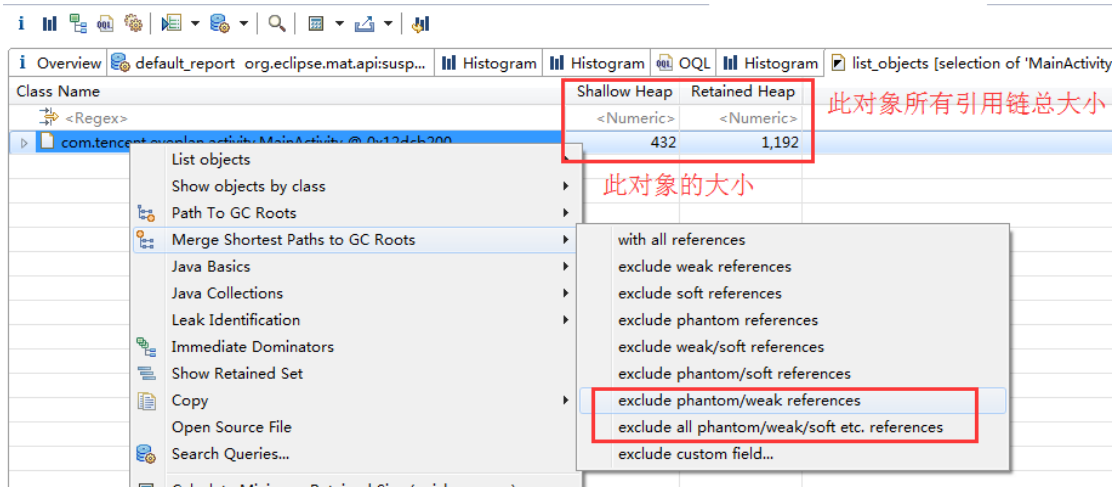
Dump 出内存泄露当时的内存镜像 hprof，分析怀疑泄露的类：



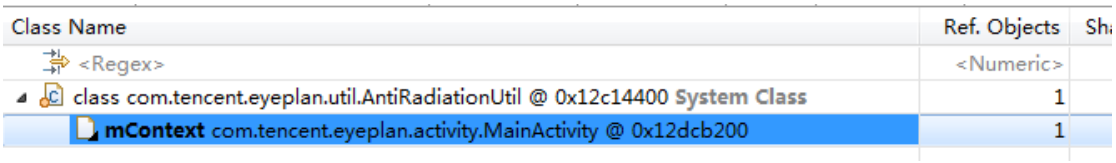
分析持有此类对象引用的外部对象



分析这些持有引用的对象的 GC 路径



逐个分析每个对象的 GC 路径是否正常



从这个路径可以看出是一个 antiRadiationUtil 工具类对象持有了 MainActivity 的引用导致 MainActivity 无法释放。此时就要进入代码分析此时 antiRadiationUtil 的引用持有是否合理（如果 antiRadiationUtil 持有了 MainActivity 的 context 导致节目退出后 MainActivity 无法销毁，那一般都属于内存泄露了）。

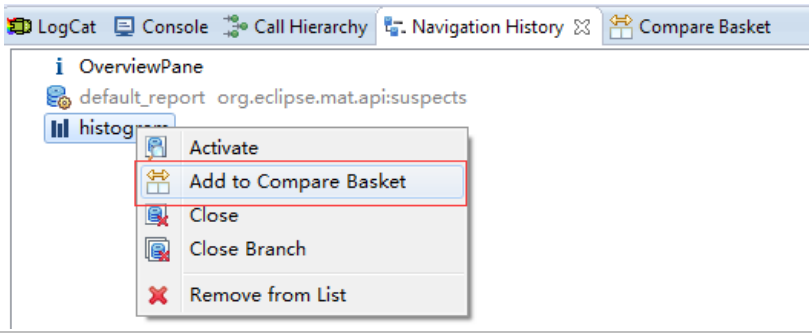
（3）MAT 对比操作前后的 hprof 来定位内存泄露的根因所在。

为查找内存泄漏，通常需要两个 Dump 结果作对比，打开 Navigator History 面板，将两个表的 Histogram 结果都添加到 Compare Basket 中去

A）第一个 HPROF 文件(usingFile > Open Heap Dump).

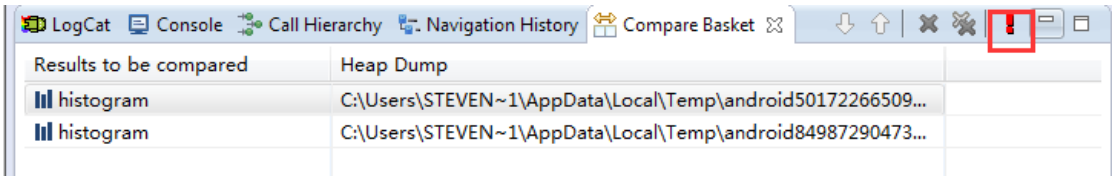
B）打开 Histogram view.

C）在 NavigationHistory view 里 (如果看不到就从 Window >show view>MAT-Navigation History)，右击 histogram 然后选择 Add to Compare Basket .

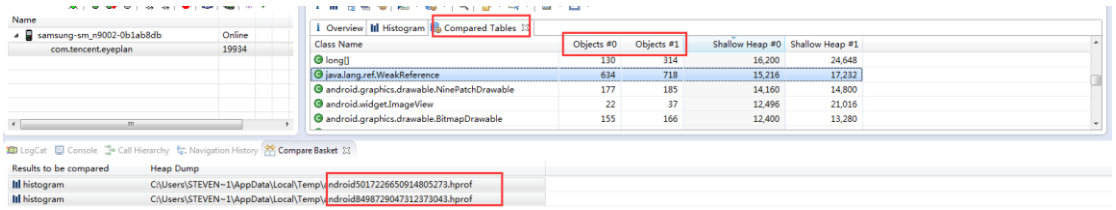


D）打开第二个 HPROF 文件然后重做步骤 2 和 3.

E）切换到 Compare Basket view，然后点击 Compare the Results (视图右上角的红色"!"图标)。



F）分析对比结果



可以看出两个 hprof 的数据对象对比结果。

通过这种方式可以快速定位到操作前后所持有的对象增量，从而进一步定位出当前操作导致内存泄露的具体原因是泄露了什么数据对象。

注意:

如果是用 MAT Eclipse 插件获取的 Dump 文件，不需要经过转换则可在 MAT 中打开，Adt 会自动进行转换。

而手机 SDk Dump 出的文件要经过转换才能被 MAT 识别，Android SDK 提供了这个工具 hprof-conv (位于 sdk/tools 下)

首先，要通过控制台进入到你的 android sdk tools 目录下执行以下命令:

./hprof-conv xxx-a.hprof xxx-b.hprof

例如 hprof-conv input.hprof out.hprof

此时才能将 out.hprof 放在 eclipse 的 MAT 中打开。

手机管家内存泄露每日监控方案

AspectJ+MLD+UIAutomator

这中方案能每日自动测试并得出是否存在疑似泄露的报告结果，不论泄露对象的大

小。其中 MLD 自研工具，原理是利用虚引用来追踪对象生命周期，网上有不少类似的开源工具，在此不详述，不同产品应根据自己的特点和要求选择适合的方案。

三、常见原因

1. 集合类。

集合类如果仅仅有添加元素的方法，而没有相应的删除机制，导致内存被占用。如果这个集合类是全局性的变量(比如类中的静态属性，全局性的 map 等即有静态引用或 final 一直指向它)，那么没有相应的删除机制，很可能导致集合所占用的内存只增不减。

2. 单例模式。

不正确使用单例模式是引起内存泄露的一个常见问题，单例对象在被初始化后将在 JVM 的整个生命周期中存在(以静态变量的方式)，如果单例对象持有外部对象的引用，那么这个外部对象将不能被 jvm 正常回收，导致内存泄露

3. Android 组件或特殊集合对象的使用。

BraodcastReceiver，ContentObserver，FileObserver，Cursor，Callback 等在 Activity onDestroy 或者某类生命周期结束之后一定要 unregister 或者 close 掉，否则这个 Activity 类会被 system 强引用，不会被内存回收。

不要直接对 Activity 进行直接引用作为成员变量，如果不得不这么做，请用 private WeakReference mActivity 来做，相同的，对于 Service 等其他有自己声明周期的对象来说，直接引用都需要谨慎考虑是否会存在内存泄露的可能。

4. Handler

要知道，只要 Handler 发送的 Message 尚未被处理，则该 Message 及发送它的 Handler 对象将被线程 MessageQueue 一直持有。由于由于 Handler 属于 TLS(Thread Local Storage)变量，生命周期和 Activity 是不一致的。因此这种实现方式一般很难保证跟 View 或者 Activity 的生命周期保持一致，故很容易导致无法正确释放。如上所述，Handler 的使用要尤为小心，否则将很容易导致内存泄露的发生。

5. Thread 内存泄露

线程也是造成内存泄露的一个重要的源头。线程产生内存泄露的主要原因在于线程

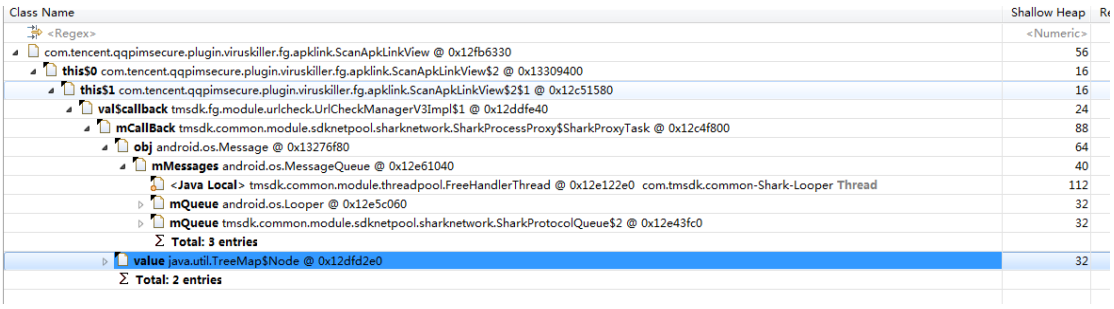
生命周期的不可控。比如线程是 Activity 的内部类，则线程对象中保存了 Activity 的一个引用，当线程的 run 函数耗时较长没有结束时，线程对象是不会被销毁的，因此它所引用的老的 Activity 也不会被销毁，因此就出现了内存泄露的问题。

6. 一些不良代码造成的内存压力

有些代码并不造成内存泄露，但是它们，或是对没使用的内存没进行有效及时的释放，或是没有有效的利用已有的对象而是频繁的申请新内存。

6.1 管家内存泄露实例原因归类:

(1) callback 只有 add 操作，没有注销 remove



可以看到 view 被 callback 引用，而 callback 被 sharkprotocolQueue 等引用导致泄漏。

(2) 发送延时消息时，如果该消息未处理，在退出页面后会导致该页面无法回收。

Android 应用启动的时候会创建 UI 主线程的 LOOPER 对象，它存在于整个应用的生命周期，用于处理消息队列里的 message。而这些 message 会引用发送该消息的 handler 对象。

泄露原因：Handler向MessageQueue中发送消息，Looper对MessageQueue中的消息进行轮询并进行分发（如图2.5），如果MessageQueue中的消息对Activity有引用，并且消息处理有时间延迟，就会导致消息的生命周期超过Activity的生命周期，造成泄露。

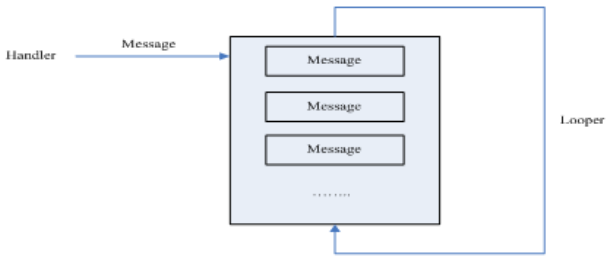


图2.5 Handler，MessageQueue,Looper的关系

那么问题来了，如果这些 handler 是 activity 的内部类，那么当这些 handler 的消息未处理完或者消息本身是延时消息的话，就会导致 activity 退出后，从 activity 到 handler 到 message 到 looper 的引用链条一直存在，从而导致 activity 的泄露！

(3) 手机瘦身的微信聊天图片解析耗时太长，退出手机瘦身后，如果解析未结束，将导致扫描任务—手机瘦身界面资源无法释放。

这种情况是典型的线程对象导致的内存泄露。原因也很简单，线程 Thread 对象的 run 任务未执行完之前，对象本身是不会释放的。因此 activity 等组件对象内的线程对象成员如果有耗时任务（一般也都是耗时任务），就会导致一直持有组件本身的引用导致内存泄露！

(4) 静态变量使用不当导致泄漏。

在解决沉浸式通知栏的一个 bug 的时候，UI 库中引入了一个 workaround，它使用了静态变量引用 page，但是 page 退出的时候没有清理，是这里导致了问题。

Static变量的生命周期伴随着APP，如果Static对象对Activity有引用则会导致Activity不能释放。这种泄露引用关系的特点：引用关系的根部为static变量（变量左边图片有小红点），GC roots 为System Class。

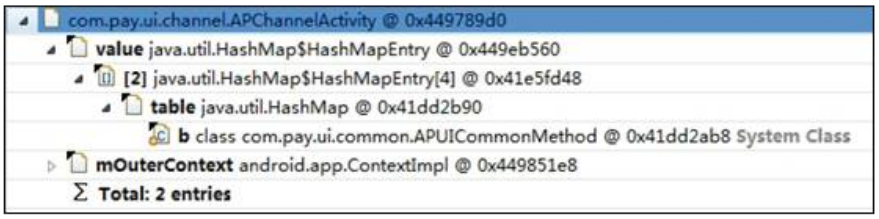


图2.7 Static导致的泄露

泄露原因：Static变量的生命周期超过了Activity的生命周期。

解决方法：在Activity退出时需要断开Static变量和Activity的引用关系。

6.1 Bitmap 没调用 recycle()

Bitmap 对象在不使用时，我们应该先调用 recycle()释放内存，然后才它设置为 null.

虽然 recycle()从源码上看，调用它应该能立即释放 Bitmap 的主要内存，

但是测试结果显示它并没能立即释放内存。但是我它应该还是能大大的加速 Bitmap 的主要内存的释放。

6.2 构造 Adapter 时，没有使用缓存的 convertView

外部补充:

Activity 泄露类型	解决方法
监听器	退出 Activity 时将监听器注销 方法：调用 removelistener()，removeobserver()进行注销
Handler	退出 Activity 时将 Handler 的 Message 或者 Callback 移除 方法：调用 removeCallbacksAndMessages(null)
Thread	退出 Activity 时将 Thread 移除或者停止 方法：Timer 线程泄露则调用 timer.cancel() 将 Runnable 从线程中移除 提供线程停止接口，当 Activity 退出时将线程停止
Static	退出界面时将 Static 变量置空
Context	使用 AppContext
中间界面	进入下级页面入口处调用 finish()
循环访问（A-B-A-B..）	设置循环访问次数

HTTP 接口测试还可以这么玩

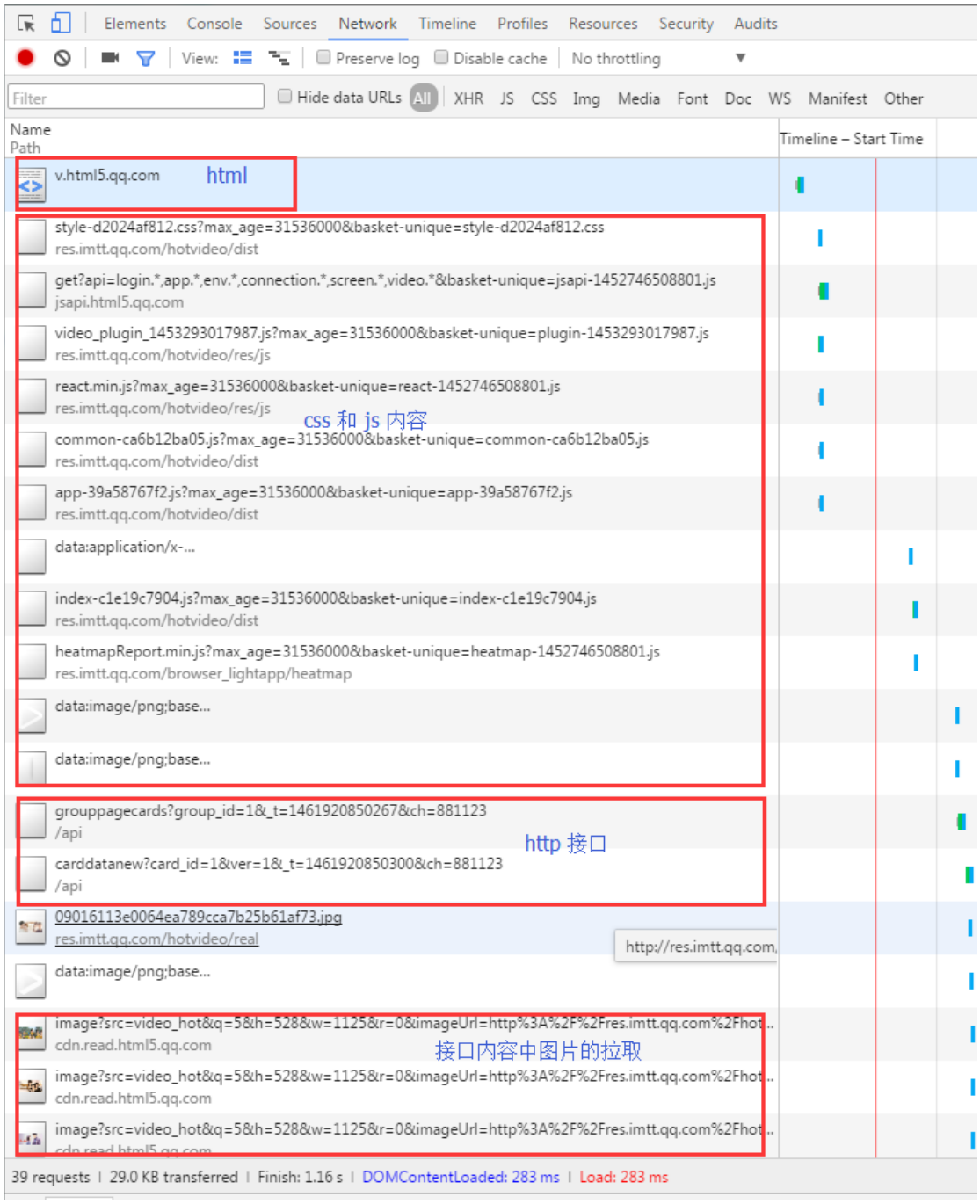
◆ 作者：赵田

一、背景

随着 H5 在各行业领域的运用，无论是在 APP 内嵌入 H5 页面的 hybrid 应用还是直接在微信公众号或者轻应用中使用 H5 页面都是非常的常见（比如前端页面通过 HTTP 接口调用拉取数据进行交互，实现前后台分离）。 而随着此类技术的应用和发展，作为一个测试人员，跟上时代的变化，除了保证前端页面 UI 的正确性，也要保证 HTTP 接口的正确性，从而保证了整个业务功能逻辑的正确性，而接口如果手工测试，不仅工作量很大，而且效率比较地下，而它的特点更适合通过搭建自动化框架来测试，既能提升效率，又能保证质量。

备注:

HTTP 接口一般有两种请求方式，一种是 POST，一种是 GET，需要关注发起请求 headers（POST 请求还要关注 post 数据）和响应的 headers 和 body，一般情况下返回的数据都是 json 格式。从 Chrome 的 Network 去分析一个网页的请求加载顺序大概就能看出，目前很多网页的请求顺序都是先去请求 html，从 html 里得到 css 和 js 的地址，去请求 css 和 js，从 js 里的 http 接口去请求相关的数据，如果拉取回来的数据还有很多图片或其他地址，在继续请求图片，回填内容到 html 网页里，网页内容不断更新变化，其实也就是接口拉取出来数据的变化，页面的样式基本都是一样的:



1.1 手工测试 hold 不住的问题

1) 如上图，视频分类很多，电影、电视剧、综艺、动漫等，每次都把各个频道测试一遍，比较耗时；

2) 在进行视频组合查询时，各种条件组合能拉取回不同的数据，而组合的方式有上千种，如何都保证查询过滤的正确性；

3) 前端页面都是正常的，可用户总反馈有时候拉取不到数据，到底哪里出了问题；

4) 写了用例，但是发现覆盖不全，因为组合场景太多，每个组合场景都测试，工作量又太大；

5) 线上出现问题了我们却不是第一个知道出问题了，没法对页面的内容进行很好的监控，因为用户场景变化多端；

1.2 怎么来通过 HTTP 接口测试很好的解决上面问题呢

1) 抽取接口（chrome 爬取？http 工具分析？手工提取）

2) 拿到接口后，怎么获取接口参数所有的值（通过线上数据去挨个查找？从运营平台获取数据？）

3) 怎么把所有线上接口都全部抓取并监控起来

4) 发现问题的反馈处理

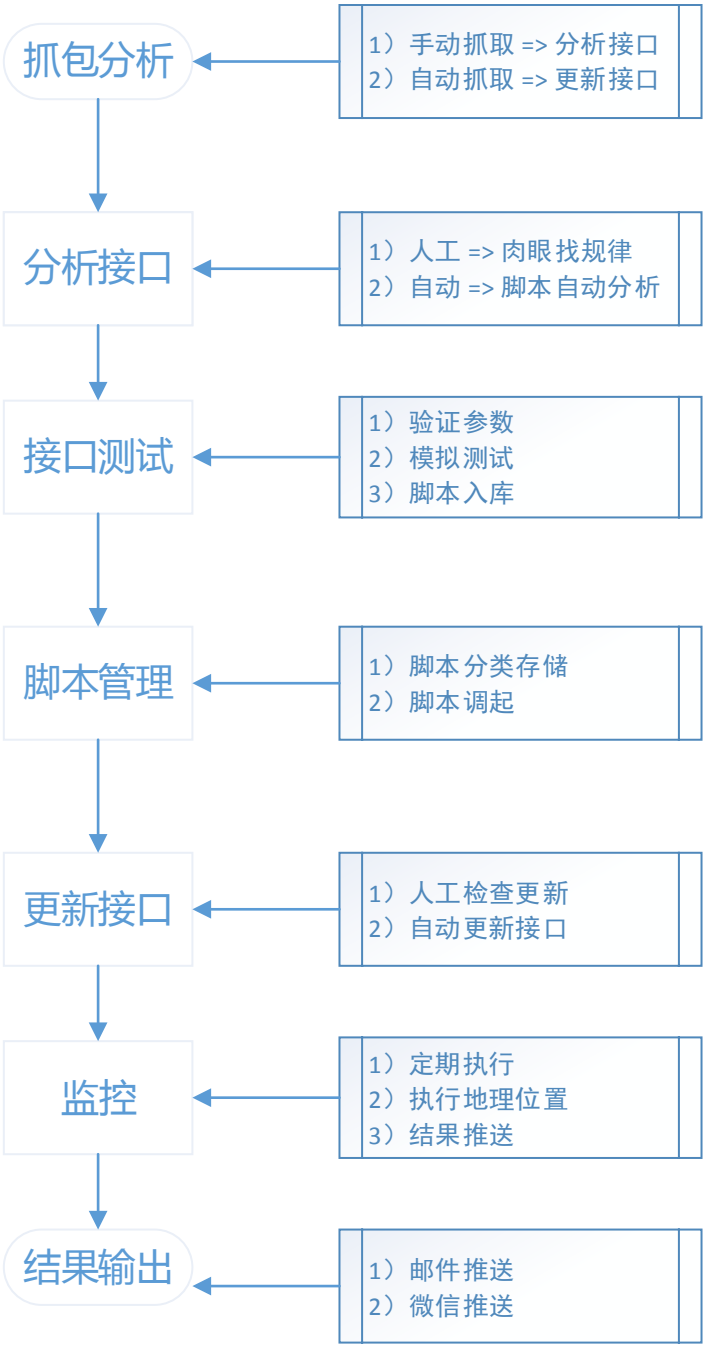
5) 接口修改维护

带着这些问题，进行了下面整个自动化接口测试平台的搭建。

二、接口自动化测试思路

2.1 整个测试流程的梳理

根据上面 1.2 所描述将会遇到的问题，整理测试设计思路，每个大项划分出要完成的子项，流程图如下:



2.2 运行时机

设计这个测试，是为了更好的更快的发现问题，能尽早的完成测试闭环，找出产品缺陷，反馈开发同学，加快整个迭代的速度。具体有以下场景：

(1) 开发自测：开发同学开发完新的接口后，不知道对其他接口有没有影响，可以跑一遍接口测试来确定；

(2) 冒烟测试：开发提测后，可以把所有接口和参数都运行一遍，所需要修改域名为测试环境域名和新增接口；

(3) 线上监控：对已上线业务进行监控，当某些组合条件查询不到数据或者某些接口拉取不到数据时，能够及时提醒相关测试和开发人员。

三、HTTP 接口自动化测试平台搭建

3.1 技术选型

1) 前端和后台逻辑：根据目前所熟悉的框架和语言，选择 Python+Django+Bootstrap

2) 存储：使用 Mysql 存储所有接口数据，分为 3 块数据（抓取回来所有接口数据、唯一接口数据、参数化接口数据）

3) 接口监控任务调起管理：Jenkins

3.2 接口数据抓取

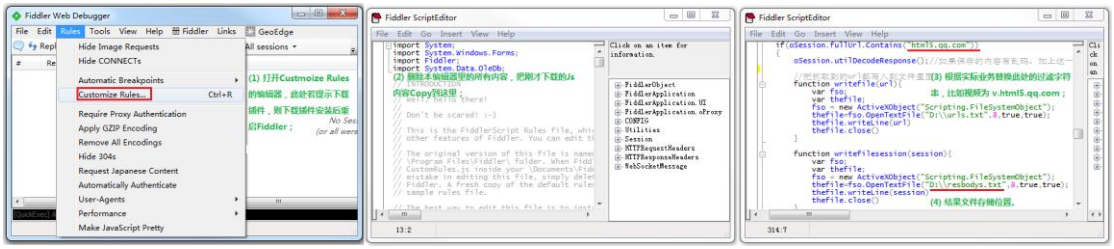
1) 手工抓取 （模块、标签是为了方便从业务角度管理 http 接口脚本）

2) 自动批量抓取：测试人员在手机上访问业务，手机通过笔记本商的 Fiddler 来代理上网，这样 Fiddler 可以抓取到所有数据，相关设置可以参考 Fiddler 手机抓包教程 http://jingyan.baidu.com/article/03b2f78c7b6bb05ea237aed2.html，抓包并分析出 HTTP 接口数据的流程如下：

a、设置 Fiddler 的 Customize Rules…

b、修改 Script 内容，具体代码和相关描述如下图，注意你需要过滤的主要域名，因为手机上有很多其他的请求也会被抓出来，通过域名过滤可以过滤出来当前域名的请求；

c、在这个脚本中，还可以定义请求中以 html、gif、css、js、jpg 等等其他和接口无关的请求；



3) 把所有有效的接口请求头，请求 body，返回头，返回 body 全部存储到文件里，等待下一步分析；

抓取到的文件数据如下：

```
GET http://v.xxxx.qq.com/api/grouppagecards?group_id=1&_t=1442997551517&ch=881123 HTTP/1.1
Host: v.xxxx.qq.com
Connection: keep-alive
Accept: */*
Q-GUID: noqb30f8e9bce440d3cd4ee1a4f3681d
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.99
Safari/537.36
Referer: http://v.xxxx.qq.com/
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8
Cookie: _gscu_661903259=3865725108h53282; _gscbrs_661903259=1; qz_gdt=iann2vimciaj7pchn35a;
3g_guest_id=-9108492734137626624; qqmusic_uin=; qqmusic_key=; qqmusic_fromtag=;
IED_LOG_INFO2=userUin%3D547365261%26nickName%3Dfabulous%26userLoginTime%3D1442309943; qm_username=547365261;
qm_sid=8731bdb1a59e676dfb08ec256152d2e6,cK4Of3l_uo_w.; pt_clientip=03757f000001fa70; pt_serverip=3bef0a825967459e;
RK=zBHCL2iIGE; pt2gguin=o0547365261; uin=o0547365261; skey=@fsLieXmrN; ptisp=ctc;
ptcz=db93eb2161cf47c45b517f8654d452d80f5bb5208256aee754d4a94553c30f5f; pgv_info=ssid=s8252699135&pgvReferrer=;
pgv_pvid=2392760601; o_cookie=547365261


HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Content-Length: 635
Content-Type: text/json
Cache-Control: no-cache
Pragma: no-cache

{"cards":[{"card_id":1,"template_type":1},{"card_id":105,"template_type":5},{"card_id":12,"template_type":2},{"card_id"
:84,"template_type":2},{"card_id":534,"template_type":6},{"card_id":501,"template_type":13},{"card_id":101,"template_ty
pe":2},{"card_id":102,"template_type":2},{"card_id":10,"template_type":2},{"card_id":104,"template_type":2},{"card_id":
308,"template_type":2},{"card_id":78,"template_type":2},{"card_id":80,"template_type":2},{"card_id":116,"template_type"
:2},{"card_id":14,"template_type":6},{"card_id":7,"template_type":4},{"card_id":8,"template_type":10}],"msg":"CARD_SUCC
ESS","name":"精选","ret":0,"size":17}
```

3.3 分析接口

1）接口清洗:

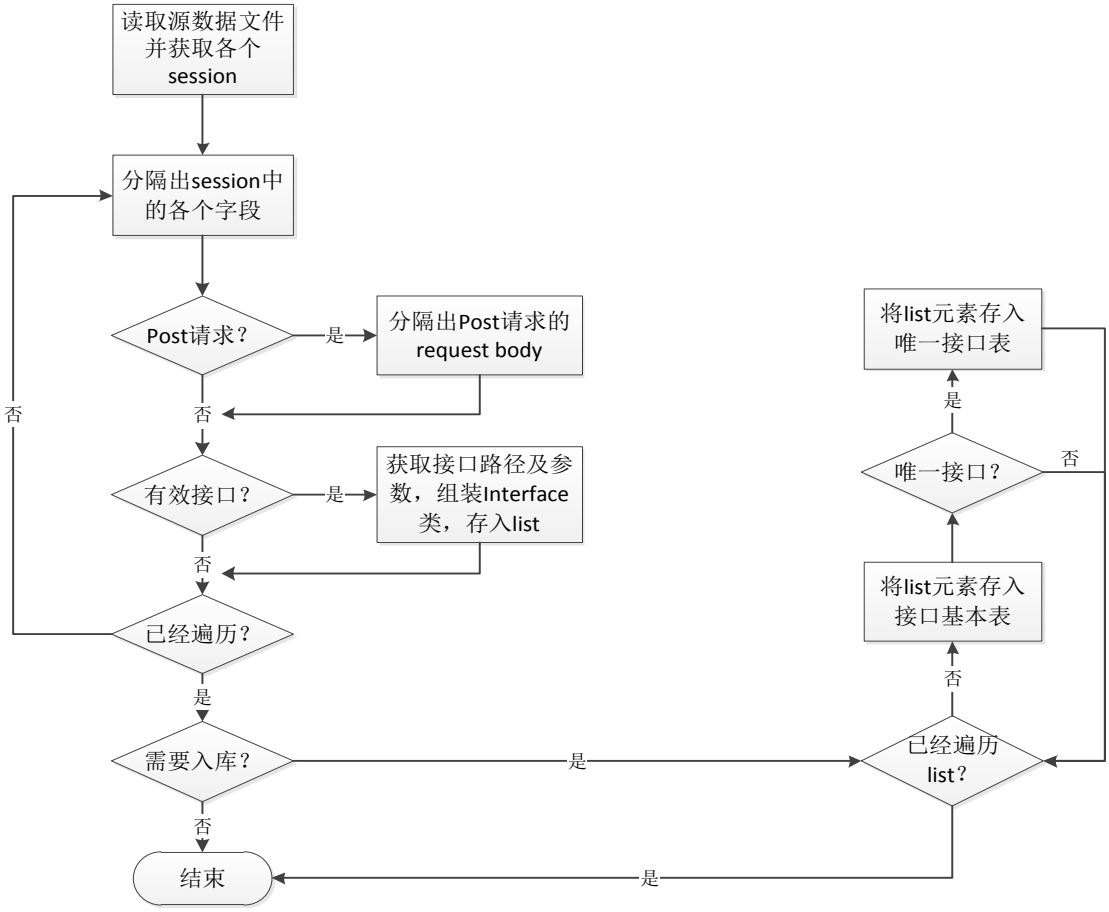
- 作用：接口回放，回归测试；
- 过滤掉提取的 http session 中的 js、css、图片等杂质；
- Post 请求：过滤掉经过加密请求（暂不考虑），其他 session 保留；

- Get 请求：api 返回数据都是 json 类型，根据 response 中的“Content-Type”字段是否为 json 判定是否为有效接口；
- 保留返回码为 301/302 跳转的 http session;

2）唯一接口过滤：

- 作用：接口回放，回归测试；
- 过滤掉提取的 http session 中的 js、css、图片等杂质；
- Post 请求：过滤掉经过加密请求（暂不考虑），其他 session 保留；
- Get 请求：api 返回数据都是 json 类型，根据 response 中的”Content-Type”字段是否为 json 判定是否为有效接口；
- 保留返回码为 301/302 跳转的 http session;

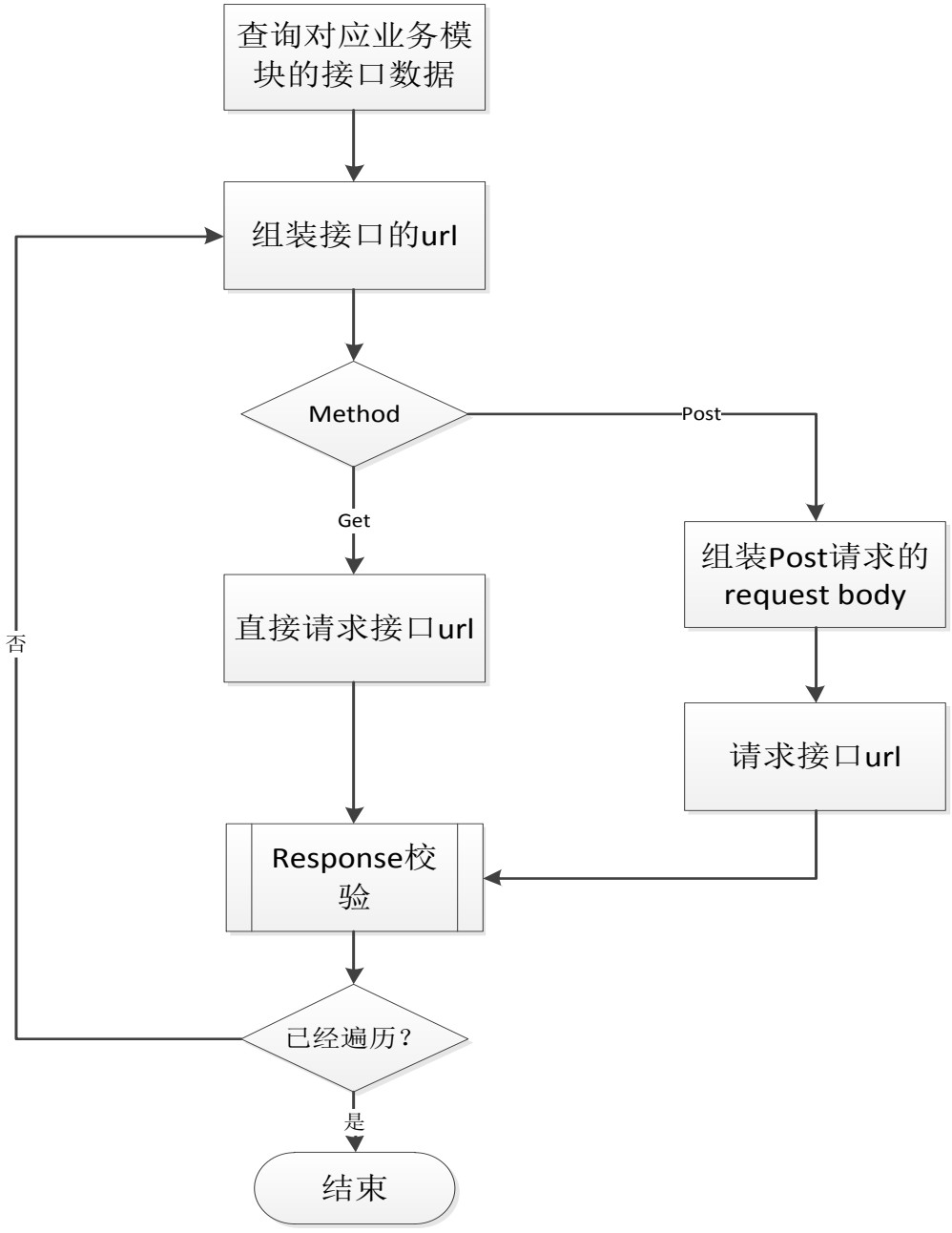
3）接口清洗流程



3.4 接口测试

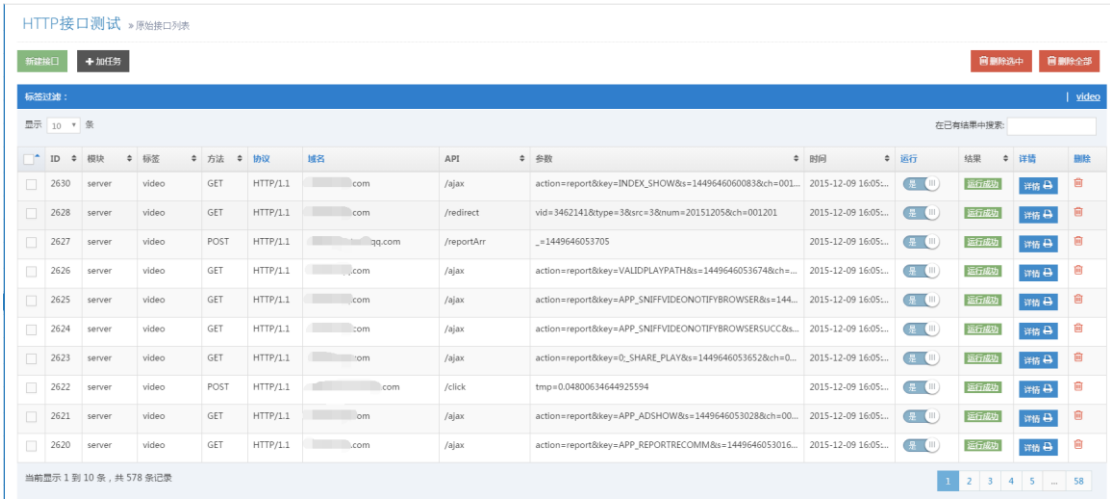
1）对清洗的接口进行测试，测试通过后，接口再做入库处理;

2）接口调用的处理流程:

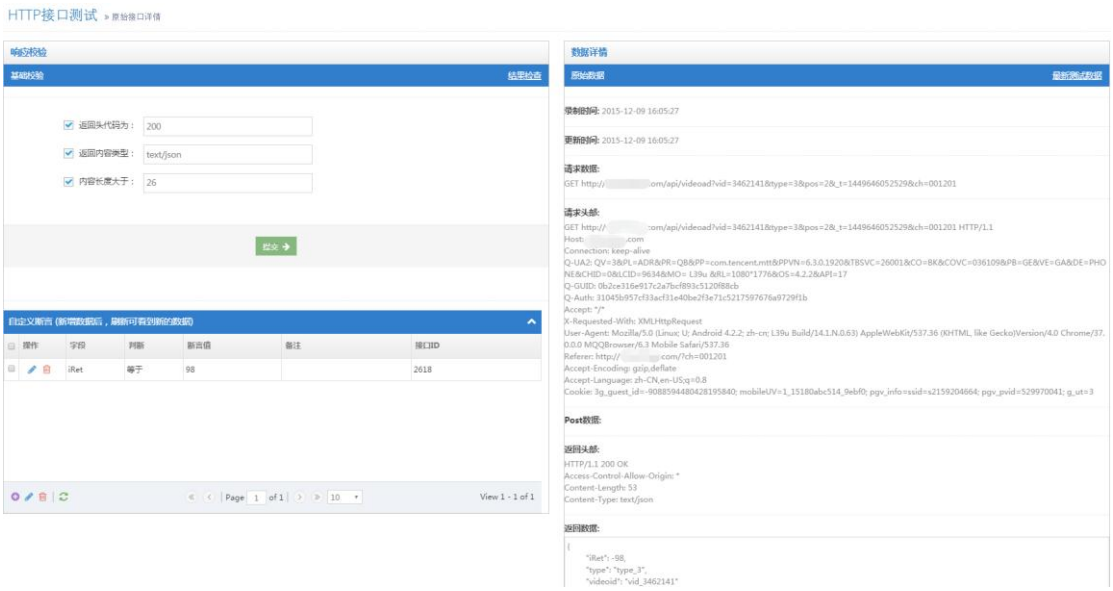


3.5 脚本管理

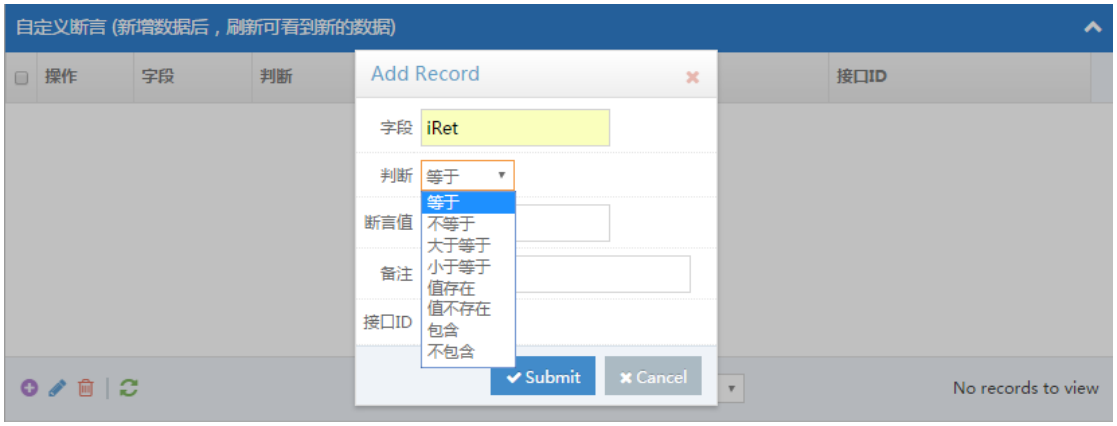
1） 可以对测试任务进行管理，设置在批量运行时是否要进行运行，运行的状态展示，运行详情设置以及删除操作，因为接口数量众多，可以选择删除选中或是全部删除:



2） 右侧可以看到所有请求的原始数据（请求时间、接口更新时间、请求数据、请求头部、响应头部、响应内容数据），左侧可以对响应进行校验，分为基础校验和自定义断言，基础校验可以校验返回头代码、返回内容类型、内容长度，自定义断言可以自己添加任何返回数据的字段并设置对比方式和值进行对比，可设置多个字段:



自定义断言:



3）接口参数化，在获取到接口后，可以通过接口参数的 key，加上从开发或者运维那里获取到的参数值列表，进行快速参数化，所有参数进行排列组合，生成该接口全集，进行回放测试;

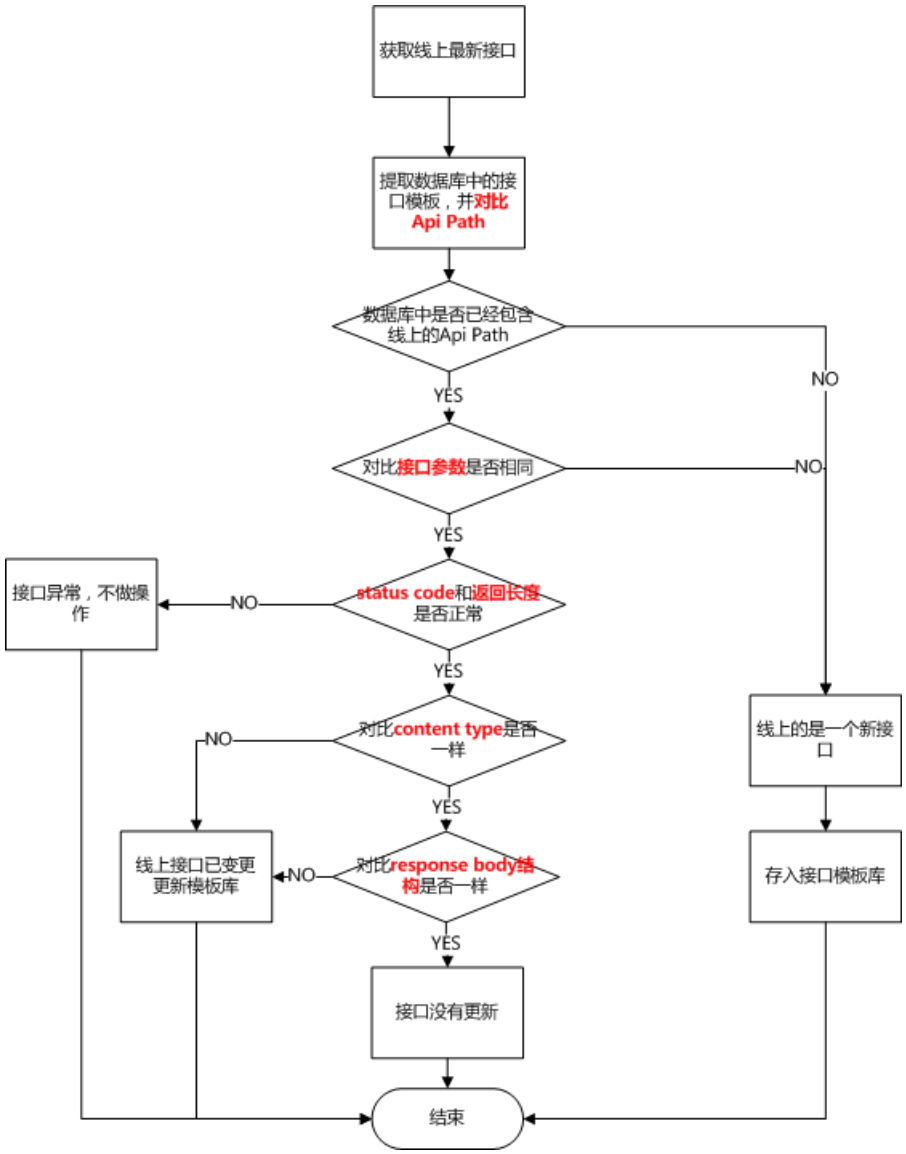
3.6 更新接口

在使用过程中，会遇根据由于业务变动来 新增、修改、删除 HTTP API 的情况，所以在接口自动化测试时，我们可以通过下面两种情况来处理接口的变动;

1) 从开发那里得知有更改变化的接口，手动通过接口管理页面进行参数的删减或者直接手工新建接口，添加到队列里;

2) 如果是不知道接口是否有变动，目前采用半自动去分析接口（和初始录制接口一样，测试人员在手机上操作一遍业务，Fiddler 录制所有接口，再自动识别接口的变更，来更数据库中的接口);

3) 具体实现流程图:

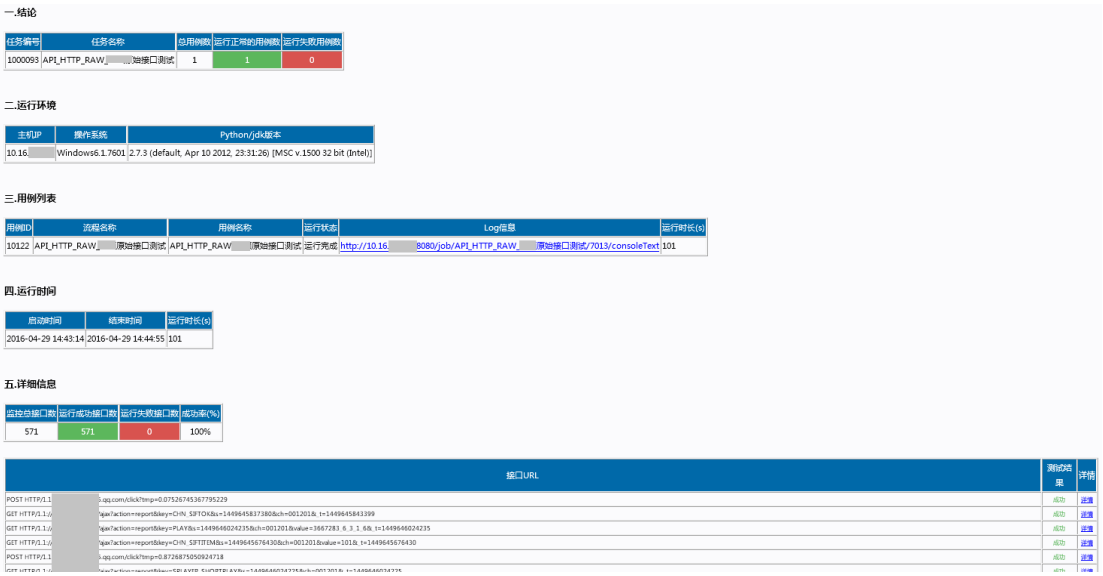


3.7 日常监控 和 结果输出

日常监控可以使用 Jenkins 来做后台管理，通过前端页面提交任务表单后，自动根据提交数据在 Jenkins 里建立新的 job，可以手动触发执行任务或者自动定时触发任务:

1) 在前端页面填写 form 表单（包括任务名称、业务分类、运行计划、结果邮件推送列表等），提交后，自动在后台添加任务到 Jenkins 里，如下图;

2) 根据运行计划执行后生成监控邮件结果，通知项目相关人员，此处可以配置为当失败时在进行通知，线上的监控一般都是每半小时执行一次，这样能够尽快的获得线上接口运行情况;



3) 发现问题的闭环：打通缺陷管理系统，如果发现监控失败的接口，可以点击详情页查看具体失败详情，确认是缺陷，可以一键提交缺陷到缺陷管理系统，方便后面对该问题的跟踪处理;

当我们讨论流畅度的时候,我们究竟在说什么?

◆作者：罗小松

一、前言：那些年我们用过的显示性能指标

注：Google 在自己文章中用了 Display Performance 来描述我们常说的流畅度，为了显得有文化，本文主要用“显示性能”一词来代指“流畅度”（虽然两者在概念上有细微差别）。

从 Android 诞生的那一刻起，流畅度就为众人所关注。一时之间，似乎所有人都在讨论 Android 和 iOS 谁的流畅度更好。但是，毫不夸张的说，流畅度绝对是 Android 众多性能维度中最为奇葩的一个。因为，为了刻画这一性能维度，业界设计了各式各样的指标来对其进行衡量。可以说弄清了这些指标我们就明白了什么是流畅度，可是这似乎并不太容易。

笔者简单搜集了一些业界中提及的显示性能指标，大家可以来品评一下：

指标名称：FPS
相关资料：Android 性能测试之 fps 获取
指标名称：Aggregate frame stats（N 多个指标）
相关资料：Testing Display Performance
指标名称：Jankiness count、Max accumulated frames、Frame rate
相关资料：JankTestBase.java
指标名称：SM、Skipped frames
相关资料：Android 应用性能评测调优

面对如此之多的显示性能指标，想必大家也会跟笔者一样，心中难免疑惑丛生。其实，我们只需要依次弄清楚以下三个哲学问题，所有的问题也许就会迎刃而解：

- 你是谁——这些指标具体反映了什么问题
- 你从哪儿来——这些指标数值是怎么得到的
- 你要到哪儿去——这些指标如何落地来指导优化

因此，本文将尝试依次从上诉三个问题来逐步分析和探讨各个显示性能指标。

1、你是谁——这些指标具体反映了什么问题

总所周知，脱离了具体的应用背景，所有的指标都是没有意义的。所以，为了彻底弄清楚各个显示性能指标的具体身份，我们势必得从 Android 的图像渲染流程说起。

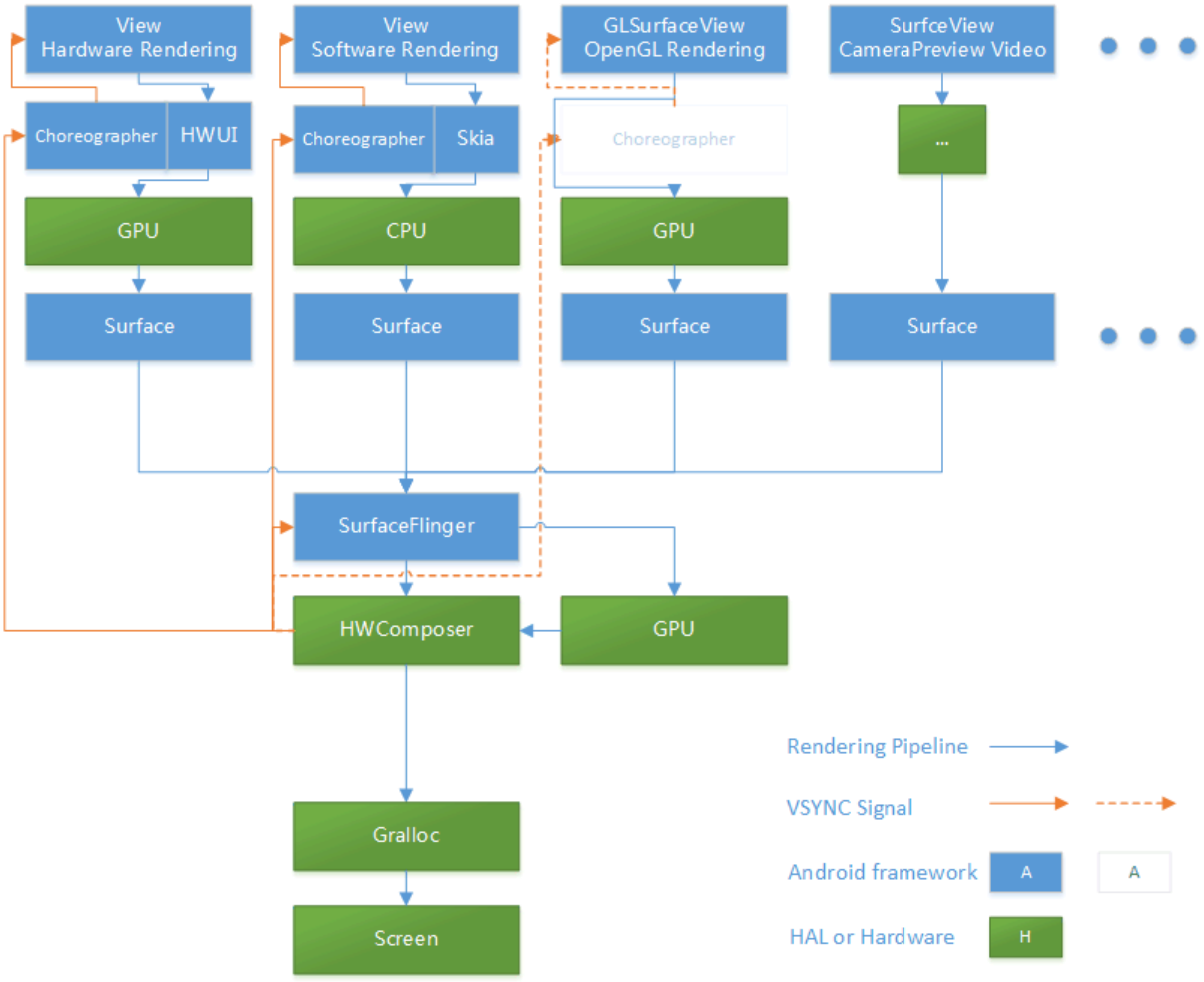
具体展开之前，首先需要说明的是，为了降低复杂程度和本章篇幅，在这个环节之中，我们只讨论图像渲染流程中的各个具体环节所对应的指标有哪些。而指标的具体定义，由第二章《你从哪儿来——这些指标数值是怎么得到的》进行讨论。

1.1. Android 图像渲染流程

下图是笔者结合各类资料（主要是是源码及官方文档），在根据自己的理解梳理出的几种常见场景下的图像渲染流程：

PS 1：笔者个人技术水平有限，若存在理解有误的地方还望指正。

PS 2：本文主要讨论的 Android 源码为 Android 6.0



备注：基于 OpenGL 的应用可以使用 Choreographer 中的 VSYNC 信号来进行图像渲染工作的安排。

上面这幅图涉及的概念较多，要完全吃透估计得费不少时间。不过好在我们只是想弄明白显示性能各种指标的含义，所以我们只需要理清下面两大关系即可：

(1) **SurfaceFlinger、HWComposer 与 Surface 的关系**

Surface：可以理解为 Android 系统中的一个基本显示单元。只要使用 Android 任意一种 API 绘图，绘制的结果都将反映在 Surface 上。

SurfaceFlinger：服务运行在 System 进程中，用来统一管理系统的帧缓冲区设备，其主要作用是将系统中的大部分 Surface 进行合成。SurfaceFlinger 主要使用 GPU 进行 Surface 的合成，合成的结果将形成一个 FrameBuffer。

HWComposer：即 Hardware Composer HAL，其作用是将 SurfaceFlinger 通过 GPU 合成的结果与其他 Surface 一起最终形成 BufferQueue 中的一个 Buffer。此外，HWComposer 可以协助 SurfaceFlinger 进行 Surface 的合成，但是否进行协助是由 HWComposer 决定的。

值得注意的是，有的 Surface 不由 WindowManager 管理，将直接作为 HWComposer 的输入之一与 SurfaceFlinger 的输出做最后的合成。

(2) **Choreographer、SurfaceFlinger、HWComposer 与 VSYNC 的关系**

VSYNC：Vertical Synchronization 的缩写，它的作用是使 GPU 的渲染频率与显示器的刷新频率（一般为固定值）同步从而避免出现画面撕裂的现象。

HWComposer：VSYNC 信号主要由 HWComposer 通过硬件触发。

Choreographer：当收到 VSYNC 信号时，Choreographer 将按优先级高低依次去调用使用者通过 postCallback 提前设置的回调函数，它们分别是：优先级最高的 CALLBACK_INPUT、优先级次高的 CALLBACK_ANIMATION 以及优先级最低的 CALLBACK_TRAVERSAL。

SurfaceFlinger：Surface 的合成操作也时基于 VSYNC 信号进行的。

简单来说，Android 图像渲染流程主要由以下特征：

(1) 我们可以简单把 Android 图像渲染架构分为应用（Surface）、系统（SurfaceFlinger）、硬件（Screen）三个层级，其中绘制在应用层，合成及提交上屏在系统层，显示在硬件层；

(2) 无论应用（Surface）、系统（SurfaceFlinger）、硬件（Screen）都是当且仅当绘制内容发生改变，才会对绘制内容进行处理;

(3) 系统中的 SurfaceFlinger 以及绝大部分 Surface 都是按照 VSYNC 信号的节奏来安排自己的任务;

目前，绝大部分 Surface 都属于 Hardware Rendering。

1.2.各个指标在 Android 图像渲染流程所代表的意义

大致梳理了 Android 的图像渲染流程之后，我们需要做的一件事情，就是看看上面提到的指标，都对应了渲染流程的哪些阶段，这样对于我们了解各个指标所反映的具体物理意义及其优势劣势都有极大帮助。再次强调，在这个环节之中，我们的讨论仅限于只讨论指标所对应的渲染流程的具体阶段，各指标的具体定义由第二章具体展开。

1.2.1.系统层级（SurfaceFlinger）的显示性能指标

(1) 基础数据：SurfaceFlinger 合成次数

(2) 指标意义：系统合成帧率：FPS

(3) 特别说明:

- SurfaceFlinger 仅在显示区域内的 Surface 有提交内容更新时才会进行合成（上屏），因此，系统合成帧率低并不一定意味着图像显示性能差，有可能是因为当前并没有任何的内容更新所导致。
- 若显示区域内的某个待测 Surface 持续进行更新时， SurfaceFlinger 的合成（上屏）的频率可以在某种程度上反映该 Surface 的显示性能，但从理论上分析该指标并不一定准确。这是因为，若显示区域内尚存在其他 Surface，它们也会影响 SurfaceFlinger 的合成（上屏）的行为，从而干扰结果。
- 若某个 Surface 的合成不在 SurfaceFlinger 中进行（如 Camera Preview），则该 Surface 的显示性能无法用这类指标进行衡量。

1.2.2.应用层级（Surface）的显示性能指标

(1) 基础数据：绘制过程中每一帧的关键时间点（如开始绘制时间、结束绘制时间等）

(2) 指标意义:

- 应用绘制帧率：Frame rate
- 应用绘制轮询频率：SM
- 应用绘制超时（跳帧）的次数：Aggregate frame stats、Jankiness count、Skipped frames
- 应用绘制超时（跳帧）的幅度：Aggregate frame stats、Max accumulated frames、Skipped frames

(3) 特别说明:

- 与 SurfaceFlinger 类似， Surface 也仅在有内容更新时才会进行绘制，因此，绘制频率低并不一定意味着图像显示性能差，有可能是因为当前并没有任何的内容更新所导致。
- Aggregate frame stats 由于其基础数据仅在硬件绘制（Hardware Rendering）过程中进行统计，属于 HWUI 的功能，所以非硬件绘制的 Surface 自然无法使用这类指标进行衡量;
- Jankiness count、Max accumulated frames、Frame rate 这类指标的基础数据来自于 FrameTracker ，适用范围最广。
- SM、Skipped frames 这类指标，由于其基础数据取自 Choreographer，若 某些 Surface 的绘制不依赖于 Choreographer ，则这些指标无法衡量该 Surface 的显示性能。

1.3.小结

评价显示性能的各个指标，可以按其在图像渲染流程中的作用，分为以下两类:

1) 系统层级的指标仅有 FPS 一根独苗，它的限制是 Surface 的和合成需要在 SurfaceFlinger 中进行;

2) 应用层级的指标较多，它们之中又可以分为三类:

➢ Aggregate frame stats 属于 HWUI 的功能， 需要 Surface 的绘制由 HWUI 进行才能进行分析;

- Jankiness count、Max accumulated frames、Frame rate 看上去适用范围最广。
- SM、Skipped frames 需要 Surface 依赖 Choreographer 进行绘制，才能正常工作;

2、你从哪儿来——这些指标数值是怎么得到的

第一章的内容仅仅是站在整个图像绘制流程的高度来简单分析各个指标的，本章将进一步分析各个指标的基础数据来源以及具体计算方式。

2.1、基础数据：系统层级（SurfaceFlinger）的合成（上屏）的次数

前面说到，在 Android 系统中，SurfaceFlinger 扮演了系统中所有 Surface 的管理者的角色，当应用程序所对应的 Surface 更新之后，绝大多数的 Surface 都将在 SurfaceFlinger 之中完成了合并的工作之后，最终才会在 Screen 上显示出来。

当然，SurfaceFlinger 的执行也是由 VSYNC 信号驱动的，这也决定了每秒钟合成次数的上限就是 60 次。当 SurfaceFlinger 接收到 Surface 更新通知的时候，将会由 SurfaceFlinger::handleMessageRefresh 函数进行处理，其中包含重建可见区域、初始化、合成等步骤。这里，我们主要关注 SurfaceFlinger::doComposition() 这个方法。

```
void SurfaceFlinger::handleMessageRefresh() {
    ...
    if (CC_UNLIKELY(mDropMissedFrames && frameMissed)) {
        // Latch buffers, but don't send anything to HWC, then signal another
        // wakeup for the next vsync
        preComposition();
        repaintEverything();
    } else {
        preComposition();
        rebuildLayerStacks();
        setUpHWComposer();
        doDebugFlashRegions();
        doComposition(); //重点关注对象
        postComposition();
    }
    ...
```

```
}
```

在 doComposition 中，完成 Surface 的合成之后，都会调用 DisplayDevice::flip()，它会使用变量 mPageFlipCount 统计我们进行合成的次数，这个变量就是我们统计 FPS 的核心原始数据。mPageFlipCount 记录了 SurfaceFlinger 一共进行了多少次合成，也可以简单理解为，SurfaceFlinger 向屏幕提交了多少帧的数据。

```
void SurfaceFlinger::doComposition() {
    ATRACE_CALL();
    const bool repaintEverything = android_atomic_and(0, &mRepaintEverything);
    for (size_t dpy=0 ; dpy<mDisplays.size() ; dpy++) {
        const sp<DisplayDevice>& hw(mDisplays[dpy]);
        if (hw->isDisplayOn()) {
            ...
            hw->flip(hw->swapRegion);//重点关注对象
            ...
        }
        // inform the h/w that we're done compositing
        hw->compositionComplete();
    }
    postFramebuffer();
}
void DisplayDevice::flip(const Region& dirty) const {
    ...
    mPageFlipCount++;
}
```

不仅如此，Android 还为我们获取这个基础数据提供了比较方便的方法。通过执行 adb 命令：service call SurfaceFlinger 1013，我们就可以得出当前的 mPageFlipCount。

```
C:\Users\xiaosongluo>adb shell
shell@cancro:/ $ su
su
root@cancro:/ # service call SurfaceFlinger 1013
service call SurfaceFlinger 1013
Result: Parcel(00aea4f4    '....')
```

2.1.1、FPS 的计算方法

根据 FPS 的定义，我们不难逆推得出 FPS 的计算方法:

在 t1 时刻获取 mPageFlipCount 的数值 v1，在在 t2 时刻获取 mPageFlipCount 的数值 v2，FPS 的计算公式:

FPS = (v2 - v1) / (t2 - t1);

需要注意的是: mPageFlipCount 的原始数据是 16 进制的，一般而言计算之前需要先进行进制转换。

2.2、基础数据: 应用层级（Surface）的绘制过程中每一帧的关键时间点（FrameInfo）

请大家先注意 FrameInfo 是由 Android 6.0（具体来讲是 Android M Preview） 引入到 HWUI 模块中的统计功能。 因此，目前来讲绝大多数系统上的大多数应用都暂时无法获取这一基础数据。不过 This IsTheFuture。

我们再来仔细瞧瞧 Google 给出的显示性能测试的十全大补丸 《Testing Display Performance : Aggregate frame stats》 。其中，特别值得关注的是 adb shell dumpsys gfxinfo <PACKAGE_NAME> framestats 这一条命令。通过这条命令，我们获取每一帧绘制过程中每个关键节点的耗时情况，从而仔细的分析潜在的性能问题。

不得不说，按照 Google 给出的这种测试方法进行测试得到的显示性能数据是非常全面的。

这些基础数据都是记录在 FrameInfo 之中，由 CanvasContext 在 doFrame（）时进行记录。相关的主要源码如下:

```
//源码: FrameInfo.cpp
#include "FrameInfo.h"
#include <cstring>

namespace android {
  namespace uirenderer {
    const std::string FrameInfoNames[] = {
      "Flags",
```

```
            "IntendedVsync",
            "Vsync",
            "OldestInputEvent",
            "NewestInputEvent",
            "HandleInputStart",
            "AnimationStart",
            "PerformTraversalsStart",
            "DrawStart",
            "SyncQueued",
            "SyncStart",
            "IssueDrawCommandsStart",
            "SwapBuffers",
            "FrameCompleted",
        };

        void FrameInfo::importUiThreadInfo(int64_t* info) {
            memcpy(mFrameInfo, info, UI_THREAD_FRAME_INFO_SIZE * sizeof(int64_t));
        }
    } /* namespace uirenderer */
} /* namespace android */
```

2.2.1、Aggregate frame stats 指标的计算方法

首先需要说明的是 Aggregate frame stats 不是一个指标，而是一系列指标集合。我们来看一个具体的 Aggregate frame stats 的例子:

```
Stats since: 752958278148ns
Total frames rendered: 82189
Janky frames: 35335 (42.99%)
90th percentile: 34ms
95th percentile: 42ms
99th percentile: 69ms
Number Missed Vsync: 4706
Number High input latency: 142
Number Slow UI thread: 17270
```

```
Number Slow bitmap uploads: 1542
Number Slow draw: 23342
```

以上统计信息的实现可以详见源码：GfxMonitorImpl.java

在 Android M 以上的系统上，上述信息的获取十分方便（事实上也只有这些系统能够获取这些信息）。仅需要执行以下命令即可：

```
adb shell dumpsys gfxinfo <PACKAGE_NAME>
```

2.3、基础数据：应用层级（Surface）的绘制过程中每一帧的关键时间点（FrameTracker）

我们先来看一段 FrameTracker 的自我介绍：

FrameTracker tracks information about the most recently rendered frames. It uses a circular buffer of frame records, and is NOTthread-safe-mutexing must be done at a higher level if multi-threaded access is possible.

其实 FrameTracker 是按照图层（Layer / Surface）统计的，而针对每个 Layer ，FrameTracker 都记录了它最近 128 帧的信息。

获取 FrameTracker 的信息相当方便，仅需要执行以下命令即可：

```
adb shell dumpsys SurfaceFlinger --latency <LAYER_NAME>
```

该条命令最终会调用如下源码，打印出绘制过程中的关键时间点：

```
void FrameTracker::dumpStats(String8& result) const {
    Mutex::Autolock lock(mMutex);
    processFencesLocked();
    const size_t o = mOffset;
    for (size_t i = 1; i < NUM_FRAME_RECORDS; i++) {
        const size_t index = (o+i) % NUM_FRAME_RECORDS;
        result.appendFormat("%" PRId64 "\t%" PRId64 "\t%" PRId64 "\n",
            mFrameRecords[index].desiredPresentTime,
            mFrameRecords[index].actualPresentTime,
            mFrameRecords[index].frameReadyTime);
    }
    result.append("\n");
```

```
}
```

其中 desiredPresentTime、actualPresentTime 以及 frameReadyTime 的含义已经非常清晰，这里就不画蛇添足了。

2.3.1、Jankiness count、Max accumulated frames、Frame rate 指标的计算方法

这里需要特别指出的是， FrameTracker 只保存了 Surface 最近渲染的 128 帧的信息，因此，Jankiness count、Max accumulated frames、Frame rate 也仅仅是针对这 128 帧数据所计算出来的结果,它们的具体含义分别是:

- Jankiness count: 根据相邻两帧绘制时间的差值，“估计”是否存在跳帧并进行跳帧次数的统计;
- Max accumulated frames: 根据相邻两帧绘制时间的差值，“估计”这 128 帧绘制过程中可能形成的最大连续跳帧数;
- Frame rate: 计算所得平均（绘制）帧率。

注: 以上数据主要依据 actualPresentTime 数据来计算，desiredPresentTime 以及 frameReadyTime 实际并未参与上述指标的计算过程。

其实 Jankiness count、Max accumulated frames、Frame rate 指标的基础数据适用范围很是广泛，而且也绕过了 FPS 无法统计具体应用显示性能的缺点。但可惜的是，FPS 无法反映静态应用的显示性能的缺点，这类指标也无法完全避免，而且由于 Jankiness count、Max accumulated frames 的数值来源于“估计”，这也限制了这一指标的使用。

如果你对具体的计算过程感兴趣，可以参考详见源码: JankTestBase

2.4、基础数据: 应用层级（Surface）的绘制过程中每一帧的关键时间点（Choreographer）

先说一句有点绕口的话: Choreographer 是依据 Choreographer 绘制的 Surface 在 UI 绘制过程中最为核心的机制。

Choreographer 的工作机制简单来说就是，使用者首先通过 postCallback 在 Choreographer 中设置的自己回调函数:

- CALLBACK_INPUT: 优先级最高，和输入事件处理有关。

- CALLBACK_ANIMATION：优先级其次，和 Animation 的处理有关。
- CALLBACK_TRAVERSAL：优先级最低，和 UI 等控件绘制有关。

那么，当 Choreographer 接收到 VSYNC 信号时，Choreographer 会调用 doFrame 函数依次对上述借口进行回调，从而进行渲染。

那么显然，doFrame 的执行效率（次数、频率）也就是我们需要的显示性能数据。而这样的基础数据，Choreographer 自身也进行了记录。如下面代码中， jitterNanos 记录了绘制前后两帧所间隔的时间差， 而 skippedFrames 则记录了 jitterNanos 这段时间 doFrame 错过了多少个 VSYNC 信号，即跳过了多少帧。

```
// Set a limit to warn about skipped frames.
// Skipped frames imply jank.
private static final int SKIPPED_FRAME_WARNING_LIMIT =
SystemProperties.getInt("debug.choreographer.skipwarning", 30);


void doFrame(long frameTimeNanos, int frame) {
   ...
   final long jitterNanos = startNanos - frameTimeNanos;
   if (jitterNanos >= mFrameIntervalNanos) {
      final long skippedFrames = jitterNanos / mFrameIntervalNanos;
      if (skippedFrames >= SKIPPED_FRAME_WARNING_LIMIT) {
         Log.i(TAG, "Skipped " + skippedFrames + " frames! " + "The application may be doing too
much work on its main thread.");
      }
      ...
      final long lastFrameOffset = jitterNanos % mFrameIntervalNanos;
      ...
      frameTimeNanos = startNanos - lastFrameOffset;
   }
   ...
}
```

上述数据的获取并不是那么的直接，所以需要一定的手段。方法一共有三种，都不难：

Logcat 方案

缺点：该方案需要系统授权“Adb Root”权限，用于修改系统属性；对于丢帧信息只能统计分析，无法进行实时处理。

优点：设置完成后，可以获取系统中所有应用各自的绘制丢帧情况（丢帧发生的时间以及连续丢帧的数量）。

其实，仔细观察代码，我们就可以注意到 Choreographer 源码中本身就有输出的方案：

```
if (skippedFrames >= SKIPPED_FRAME_WARNING_LIMIT) {
    Log.i(TAG, "Skipped " + skippedFrames + " frames! " + "The application may be doing too much work on its main thread.");
}
```

唯一阻碍我们获取数值的是：skippedFrames 的数值只有大于 SKIPPED_FRAME_WARNING_LIMIT 才会输出相关的警告。而 SKIPPED_FRAME_WARNING_LIMIT 的数值可以由系统参数 debug.choreographer.skipwarning 来设定。

注意：初始条件下，系统中不存在 debug.choreographer.skipwarning 参数，因此 SKIPPED_FRAME_WARNING_LIMIT 将取默认值 30。因此，正常情况下，我们能够看见上诉 Log 出现的机会极少。

因此，如果我们修改（设定）系统属性 debug.choreographer.skipwarning 为 1，Logcat 中将打印出每一次丢帧的 Log。需要说明的是，由于为 SKIPPED_FRAME_WARNING_LIMIT 赋值的代码段由 Zygote 在系统启动阶段加载，而其他应用都是在拷贝复用 Zygote 中的设定，因此设定系统属性后需要重启 Zygote 才能使得上述设定生效。

具体的设置方法如下：

```
setprop debug.choreographer.skipwarning 1
setprop ctl.restart surfaceflinger; setprop ctl.restart zygote
```

设定完成以后，我们可以直接通过 Logcat 中的信息得到系统中所有应用的绘制丢帧信息，包括丢帧发生的时间以及连续丢帧的数量。不过由于 Logcat 信息的滞后性，以上信息我们几乎只能进行在测试完成后进行统计分析，而无法进行实时处理。

Choreographer.FrameCallback 方案

缺点：该方案需要将测试代码与待测应用打包在一起，因此理论上仅能测试自己开发的应用。

优点：可以对丢帧信息进行实时处理

我们先来看看 Choreographer.FrameCallback 的定义。

Implement this interface to receive a callback when a new display frame is being rendered. The callback is invoked on theLooper thread to which the Choreographeris attached.

通过这个接口，我们可以在每一帧被渲染的时候记录下它开始渲染的时间，这样在下一帧被处理是，我们不仅可以判断上一帧在渲染过程中是否出现掉帧，而整个过程都是实时处理的，这为我们可以及时获取相关的调用栈信息来辅助定位潜在的性能缺陷有极大的帮助。

代码注入方案

缺点：该方案需要通过注入程序为指定应用注入测试代码，因此需要系统为注入程序授权 "应用 Root" 权限。

优点：与 Choreographer.FrameCallback 方案一致。

该方案可以简单理解为通过注入的方式来实现与 Choreographer.FrameCallback 方案一样的目的。因此，这里我们主要讨论两者在实现方式上的区别。

显而易见，我们需要注入的对象是 Choreographer ，因此理论上任何第三方应用都是可以被注入的。但是随着 Android 系统对"应用 Root" 权限管理越来越严格，所以该方案可用的范围越来越小。

2.4.1、SM 指标的计算方法

根据定义，SM 其实类似于 FPS，它被设计为可以衡量应用平均每秒执行 doFrame（） 的次数。我们可以认为它是在衡量 Surface 渲染轮询的次数。

针对 **Logcat 方案**，我们只需统计测试过程中目标进程一共掉了多少帧，由于对于绝大多数应用在没有丢帧的情况下会针对每一次 VSYNC 信号执行一次 doFrame（），而 VSYNC 绝大多数情况下每秒会触发 60 次，因此我们可以反向计算得出 SM 的数值：

SM = (60* totalSeconds - totalSkippedFrames) / totalSeconds;

针对 **Choreographer.FrameCallback 方案** 以及 **代码注入方案**，我们需要在代码中自己进行统计输出（可以是设计成实时的，也可以设计成测试结束后进行统计计算的）。

2.4.2、Skipped frames 指标的计算方法

这个指标的就是指当前应用在丢帧发生时的丢帧帧数。

针对 **Logcat 方案**，该数值直接在 Logcat 中输出，并且带有时间信息。

04-18 16:31:24.957 I/Choreographer(24164): Skipped 4 frames! The application may be doing too much work on its main thread.

04-18 16:31:25.009 I/Choreographer(24164): Skipped 2 frames! The application may be doing too much work on its main thread.

针对 **Choreographer.FrameCallback 方案**以及**代码注入方案**，我们可能很方便的通过计算前后两帧开始渲染的时间差获得这一数值，同样方便。同样与 **Logcat 方案**不同的是，它也是可以设计成实时计算的。

2.5、小结

通过对各个显示性能指标的分析，我们可以知道，虽然目前指标众多，但其实有本质区别的指标确很少：

- 系统层面：

合成（上屏）帧率：FPS

- 应用层面：

跳帧次数：Aggregate frame stats、Jankiness count、Skipped frames

跳帧幅度：Aggregate frame stats、Max accumulated frames、Skipped frames

绘制帧率：Frame rate

绘制轮询频率：SM

更为重要的是，我们从上述的分析中知道了各个指标都有着自己的优势和不足，这也从根本上决定了它们各自有各自的用法。

3、你要到哪儿去——这些指标如何落地来指导优化

其实指标的用法也是多种多样的，为了方便讨论，我们仅从日常监控、缺陷定位以及数据上报三个方面来讨论各个显示性能指标是如何落地的。

3.1、日常监控

- FPS：数据形式最为直观（FPS 是最早的显示性能指标，而且在多个平台中都有着类似的定义），且对系统平台的要求最低（API level 1），游戏、视频等连续绘制的应用可以考虑选用，但不适用于绝大多数非连续绘制的应用；
- SM：数据形式与 FPS 类似，可以很好的弥补 FPS 无法准确刻画非连续绘制的应用显示性能的缺陷；
- Aggregate frame stats：除了对系统平台有较高的要求以外，其采集方式最为简单（系统自带功能）；
- Skipped frames：与 Aggregate frame stats 类似， 信息量相对较少，但可适用范围更广

特别说明：Jankiness count、Max accumulated frames、Frame rate 只统计了 128 帧的信息（约 2~3 秒），而且 Jankiness count、Max accumulated frames 对于掉帧情况的计算并非是一个准确值，因此这些指标都不太适用于日常监控

举个栗子，笔者服务的某个产品使用如下的一些指标来监控产品与竞品的性能变化情况：

测试指标	场景 1	场景 2	场景 3	场景 4
FPS	58	58	58	58
SM	59	59	59	59
Num of 6+ Skipped Frames	0	0	0	0
Num of 3+ Skipped Frames	0	0	2	0

备注：

1.Num of x+ Skipped Frames 代表测试过程中发生连续丢 x 帧（及以上）的次数；

2.至于为什么我们选择关注连续丢 3 帧以及连续丢 6 帧的的次数，在【缺陷定位】部分有相关的分析讨论；

3.2、缺陷定位

Skipped frames：基于 Choreographer.FrameCallback 方案实现的 Skipped frames 指标，可以在卡顿出现的时刻获取应用堆栈信息，可以在一定程度上进行缺陷定位

特别说明：

FrameInfo 相关指标无法直接进行缺陷定位，但 FrameInfo 当中包含了大量详尽的绘制基础数据，对于缺陷定位也有较大帮助；

关于缺陷定位过程中连续掉帧阈值的选取，可参考维基百科中提到几个重要的帧率数值：

- 10-12 fps：由于人类眼睛的特殊生理结构，如果所看画面之帧率高于每秒约 10-12 帧的时候，就会认为是连贯的
- 24 fps：有声电影的拍摄及播放帧率均为每秒 24 帧，对一般人而言已算可接受
- 30 fps：早期的高动态电子游戏，帧率少于每秒 30 帧的话就会显得不连贯，这是因为没有动态模糊使流畅度降低
- 60 fps：在实际体验中，60 帧相对于 30 帧有着更好的体验

以上各数据分别对应： 0 帧、1 帧、2.5 帧、5~6 帧。（这就是为啥选择 3/6 的原因）

举个栗子， 笔者的同事万大师（yuwan）基于上述原理自研了一款性能分析工具。该工具在集成于待测应用之后，可以自动保存如下的性能缺陷信息：

Frame lost:（连续丢帧数量，一般我们会设定一个阈值，例如 6 帧以上我们才会进行记录）
User action: ...（当前用户操作）
Stack trace: ...（当前堆栈信息）

有了这个工具之后，我们可以收集应用的各个潜在“卡顿”点，用于进一步的分析和优化。

3.3、数据上报

- Aggregate frame stats：除了对系统平台有较高的要求以外，其采集方式最为简单（系统自带功能）、数据也比较清晰，相信基于这类指标实现性能数据上报是特别方便的
- Skipped frames ： 基于 Choreographer.FrameCallback 方案实现的 Skipped

frames 指标，采集方式简单，实现基础性能数据上报、卡顿数据上报也是很方便的

这方面应用，笔者所服务的产品暂时没有涉及，就不举例子了。如果各位观众感兴趣，建议参考 Android ANR 的设计理念。

3.4、小结

发现了没有 Skipped frames 的用处很大有没有？ 而且通读全篇，你会发现 Aggregate frame stats、Jankiness count、Max accumulated frames 这些指标都有提供类似的功能。

至于为什么，这就不是本文需要讨论的内容了，如果大家比较感兴趣，笔者这里给出两份相关的链接以供各位参考:

Fps Versus Frame Time 量化和优化用户与 Android 设备之间的交互

附、现有显示性能指标对比

本来写到这里本文的主要内容就应该结束了。但是如果不对比一下显示性能指标神马的，总会让人觉得缺少了一些什么。

友情提示：下述内容相对主观，建议各位读者依据项目情况自行进行选择。

指标名称	指标意义	基础数据来源	采集方式	适用系统	适用应用	用途
FPS	系统合成帧率	SurfaceFlinger	adb shell	1.0+	略	监控
Aggregate frame stats	应用跳帧次数、幅度	FrameInfo	adb shell	6.0+	HW Rendering	监控/上报
Jankiness count	（估算）应用跳帧次数	FrameTracker	adb shell	1.0+	Almost All	定位
Max accumulated frames	（估算）应用跳帧幅度	FrameTracker	adb shell	1.0+	Almost All	定位
Frame rate	应用绘制帧率	FrameTracker	adb shell	1.0+	Almost All	定位
SM	应用绘制轮询频率	Choreographer	多种方式	4.2+	SW/HW Rendering 及 部分 OpenGL Rendering	监控
Skipped frames	应用跳帧次数、幅度	Choreographer	多种方式	4.2+	SW/HW Rendering 及 部分 OpenGL Rendering	监控/定位/上报

牛刀小试-Xposed 在测试中的小应用

◆ 作者：蒋翠翠

一、开端:

作为测试人员，想必多有遇到过这样的情景：跟开发 G 沟通测试发现的问题，尤其是性能问题，有时会被质疑证据不充分——“产品模块那么多，凭啥说是我这块引起的？”。这不，前不久本宝宝就遇到了这么一例：被测产品引入了第三方的 SDK，出于对产品的负责，开发 G 向我们提出了测试需求——评测引入 SDK 对原产品性能的影响。对比测试发现，引入 SDK 后，产品 Wifi 扫描的频次和时长明显增加。这意味着，产品的电量消耗将大幅增长（Wifi 扫描、GPS 定位等硬件相关的模块，耗起电来，还是蛮吓人的）。跟开发 G 说明情况后，开发 G 快速拉了 SDK 的接口人（简称“Y 童鞋”，即外部童鞋），一起来定位这个 Bug。

本以为证据确凿——引入了 SDK 才增长的，那不就是 SDK 的问题嘛！可 Y 童鞋坚持：SDK 的逻辑中不可能有那么多 Wifi 扫描；自测也没发现 SDK 有问题；再说了，你们产品本身也有 Wifi 扫描，凭啥说多出来的一定是 SDK 中的？唇枪舌战十多个来回，也没能把问题扯清楚。不过，倒是受了 Y 童鞋的启发——“凭啥说多出来的一定是 SDK 中的”——若是能把 Wifi 扫描的代码触发路径找到，到底是谁家的，不就一清二楚，铁证如山了？那，如何寻求证据呢？

求证思路:

Android Wifi 扫描需调用到系统的 android.net.wifi.WifiManager.startScan()方法。通过劫持该方法，在方法被调时打印其调用关系，即可知道最初的调用是在哪块代码。说到方法劫持，就该 Xposed 出手了。

二、Xposed 简介:

Xposed 是一个针对 Android 平台的方法劫持开源项目（链接:

https://github.com/rovo89/XposedBridge/wiki/Development-tutorial)。它的强大之处在于可以 hook 方法的调用，改变系统和应用的行为，而不需要对 APKs 做任何修改。如果是通过反编译 APK 来修改应用，可以在反编译后代码的任何地方插入或修改内容，但是需要重新编译 APK 并签名，并且改变仅针对单个 APK 包。使用 Xposed 的 hook，虽不能改变方法内部的代码(因为在方法内部难以寻址和定义改变的内容)，但可以在被 hook 的方法调用前后注入自己的代码(方法是 java 中的最小单元，方便寻址)。其基本思路及原理为:

Android 中每个应用进程启动时都由 Zygote 进程复制(fork)而来，获取 Zygote 进程的 Dalvik 虚拟机实例拷贝，并与 Zygote 共享 java 运行时库。

Zygote 进程在手机开机时由/init.rc 脚本启动，执行/system/bin/app_process（一个二进制文件，负责加载所需类，并触发相关的初始化方法）后进程启动完成，如下:

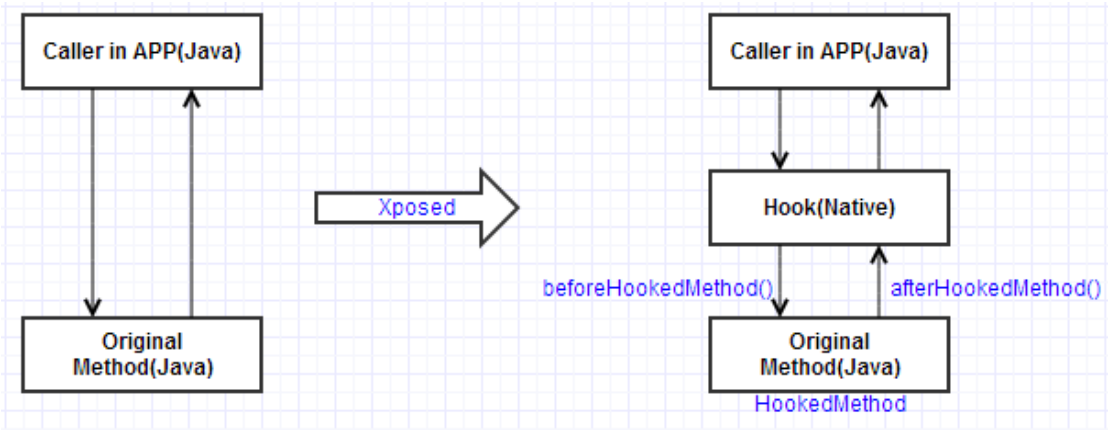
```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
```

Xposed 的切入点就在 app_process: 安装 Xposed 框架后，一个扩展了的 app_process 文件会替换原有文件，添加额外的 XposedBridge.jar 到 classpath 中。这样，通过 Zygote 复制而来的应用进程也都加载了 XposedBridge.jar。

XposedBridge.jar: 位于/data/data/de.robv.android.xposed.installer/bin/目录下。其中的 XposedBridge 类的 main 方法在进程启动时会被调用。一些初始化的工作和 Xposed 模块的加载都在该 main 方法中执行。

XposedBridge 类的 hookMethodNative()方法可以将目标方法(被 Hook 的方法)的 type 属性改为"native"（native 方法的调用将跳转至方法的地址去执行），并将方法的地址链接到自定义的 native 通用方法。也就是说，每次目标方法被调用时，都会转而调用自定的通用方法。在自定义的方法中，XposedBridge 类的 handleHookedMethod()方法被调用，该方法将参数和 “this reference” 传递给目标方法，并执行已注册的回调函数。回调函数中可以改变传递给目标方法的参数、实例、静态变量，触发其他方法，改变目标方法返回的结果...或者直接跳过——非常灵活。

简图如下:



关于 Xposed 的更多信息，可参考:

"Android Hook Xposed 原理与源码分析":

http://blog.csdn.net/wxyyxc1992/article/details/17320911

“Xposed 开发教程”:

https://github.com/rovo89/XposedBridge/wiki/Development-tutorial

三、Xposed 牛刀小试:

寻求证据的过程，也即 Xposed 的使用过程。在 Dalvik 环境下，Xposed 的使用资源包含两大块：Xposed installer 和 Xposed module。两者都是 app 的形式。Xposed.installer.apk 提供固有的框架，完成 app_process 的替换、classpath 的修改，并提供 UI 来进行框架、模块的管理和日志的查看。Xposed module 为自定义的 hook 模块，指定需要 Hook 的方法以及 Hook 到方法后相应的操作。

1、Xposed 使用步骤

1) 安装 xposed.installer.apk;

2) 开发并安装 Xposed module;

3) 加载并激活 module(by reboot) ；

4) 操作被测应用，即可看到 hook 到的日志信息。日志路径为 /data/data/de.robv.android.xposed.installer/log/error.log。

2、Xposed module 开发

Xposed module 和普通 app 的差别在于，其 AndroidManifest.xml 有几个特殊的 meta

data 标签，并且有一个特殊的配置文件 xposed_init，用来使 Xposed 框架识别 module。使用 Eclipse 开发 Xposed module，步骤如下:

Step1、新建 android project

Step2、编辑 AndroidManifest.xml

在 AndroidManifest.xml 文件中添加 3 个 meta data 节点, (name，value)分别为:(xposedmodule，true)、(xposeddescription，(module 的简单描述，自定义))、(xposedminversion，(所使用的 XposedBridge.jar 的版本))。示例如下:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.test.xposeddemo"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="14"
        android:targetSdkVersion="18" />

    <uses-permission
     android:name="android.permission.WRITE_EXTERNAL_STORAGE"
     android:maxSdkVersion="18" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >

        <meta-data android:name="xposedmodule" android:value="true"/>

        <meta-data                    android:name="xposeddescription"
android:value="Xposed IO"/>
```

Step3、导入 XposedBridgeApi.jar

因为需要使用 XposedBridge 中的 API，所以下载 XposedBridgeApi-<version>.jar 并将其加入到 Build Path。(note: 确保编译 APK 时，该 jar 包只是 referenced，而不是 included，否则会报 IllegalAccessError。工程的 libs 文件夹下的文件编译时默认 included，所以不要将该 jar 包放在 libs 文件夹，而是放到 lib 文件夹下。)

Step4、coding

Xposed module 有几个不同的入口点，可以根据自己的需求，在 Android 系统启动、apk 包加载、或 apk 资源初始化的时候使用 Xposed Hook 方法。所有的入口点都是扩展了 IXposedMod 接口的子接口，分别为:

IXposedHookZygoteInit

IXposedHookLoadPackage

IXposedHookInitPackageResources

自定义的 module 模块中，实现上述几个接口中的一个，静态导入 de.robv.android.xposed.XposedHelpers.findAndHookMethod 方法，hook 目标方法，并 Override 相关的方法，即可。

3、Xposed 使用实例——Wifi 扫描的 Hook:

前面说过，Android Wifi 扫描需调用到系统的 android.net.wifi.WifiManager.startScan() 方法。函数原型如下:

```
public boolean startScan ()
```

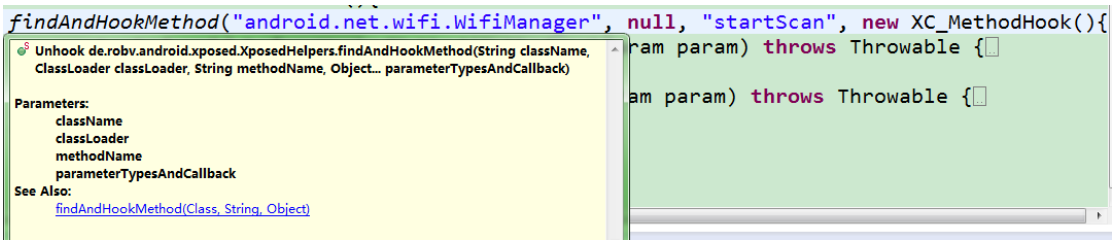
Returns	
boolean	true if the operation succeeded, i.e., the scan was initiated

整个 Hook startScan()的代码，也即 Xposed Module 的主体很简单，整体结构如下:

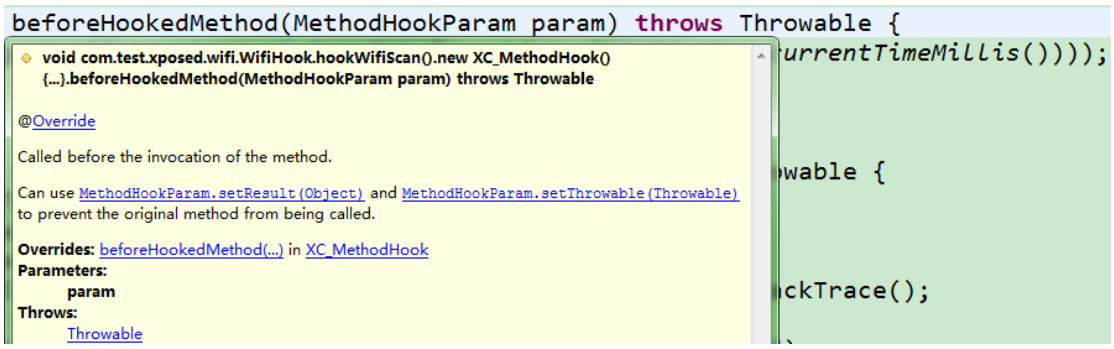
```
import static de.robv.android.xposed.XposedHelpers.findAndHookMethod;
import de.robv.android.xposed.IXposedHookLoadPackage;
import de.robv.android.xposed.XC_MethodHook;
import de.robv.android.xposed.XposedBridge;
import de.robv.android.xposed.callbacks.XC_LoadPackage.LoadPackageParam;

import java.text.SimpleDateFormat;
import java.util.Date;

import com.test.uitl.Util;

public class WifiHook implements IXposedHookLoadPackage{
    private String pkgName;
    private static int countWifiScan;
    private static SimpleDateFormat myFmt = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

    public WifiHook(){
        pkgName = Util.getTargetPkgName();
        countWifiScan = 0;
    }

    @Override
    public void handleLoadPackage(LoadPackageParam lpparam) throws Throwable {
        if(lpparam.packageName.equalsIgnoreCase(pkgName)){
            hookWifiScan();
        }
    }

    public static void hookWifiScan(){
        findAndHookMethod("android.net.wifi.WifiManager", null, "startScan", new XC_MethodHook(){
            protected void beforeHookedMethod(MethodHookParam param) throws Throwable {

            protected void afterHookedMethod(MethodHookParam param) throws Throwable {
        });
    }
}
```

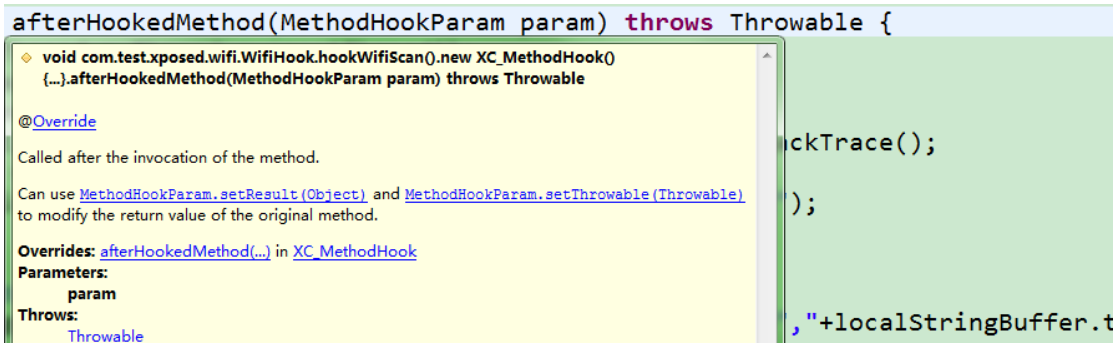
因为只关注被测应用中的 wifi 扫描情况，所以这里选择在 apk 加载的时候 Hook startScan()方法。自定义 WifiHook 类实现 IXposedHookLoadPackage 接口，并重写其 handleLoadPackage()方法。该方法中，首先判断所加载的包是否是被测应用的 apk 包名，如果是，则 Hook 其中的 startScan()方法。

找到并 hook 目标方法，通过 findAndHookMethod()实现。该方法的参数分别为目标方法所在类的 className，classLoder，方法名，方法参数的对象类型（没有参数的情况下不需指定），以及 Hook 到方法后的回调函数。



对最后结果起作用的就是这里的回调函数 XC_MethodHook。通过重写 XC_MethodHook 的两个方法 beforeHookedMethod()和 afterHookedMethod()来指定目标方法执行前后分别进行怎样的操作。

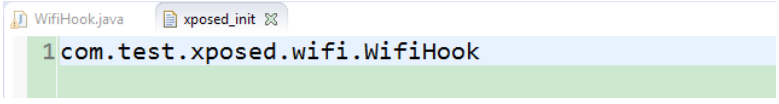




在 startScan()调用前，记录其开始的时间；调用后，获取该方法当前所在进程，使用 Thread.getStackTrace()方法获取该方法的调用关系，并记录其结束的时间；借助 XposedBridge 的日志类将上述信息输出到日志文件。

```
public static void hookWifiScan(){
    findAndHookMethod("android.net.wifi.WifiManager", null, "startScan", new XC_MethodHook(){
        @Override
        protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
            XposedBridge.log("startScan at "+myFmt.format(new Date(System.currentTimeMillis())));
        }

        @Override
        protected void afterHookedMethod(MethodHookParam param) throws Throwable {
            Thread curThread = Thread.currentThread();
            String threadName = curThread.getName();
            StringBuffer localStringBuffer = new StringBuffer();
            StackTraceElement[] arrayOfStackTraceElement = curThread.getStackTrace();
            for (int i = 4; i < arrayOfStackTraceElement.length; i++) {
              localStringBuffer.append(arrayOfStackTraceElement[i] + "->\t");
            }

            countWifiScan++;
            XposedBridge.log(countWifiScan+" times at thread "+threadName+","+localStringBuffer.toString());
            Boolean ret = (Boolean) param.getResult();
            XposedBridge.log("startScan return "+ret+", end at "+myFmt.format(new Date(System.currentTimeMillis())));
        }
    });
}
```

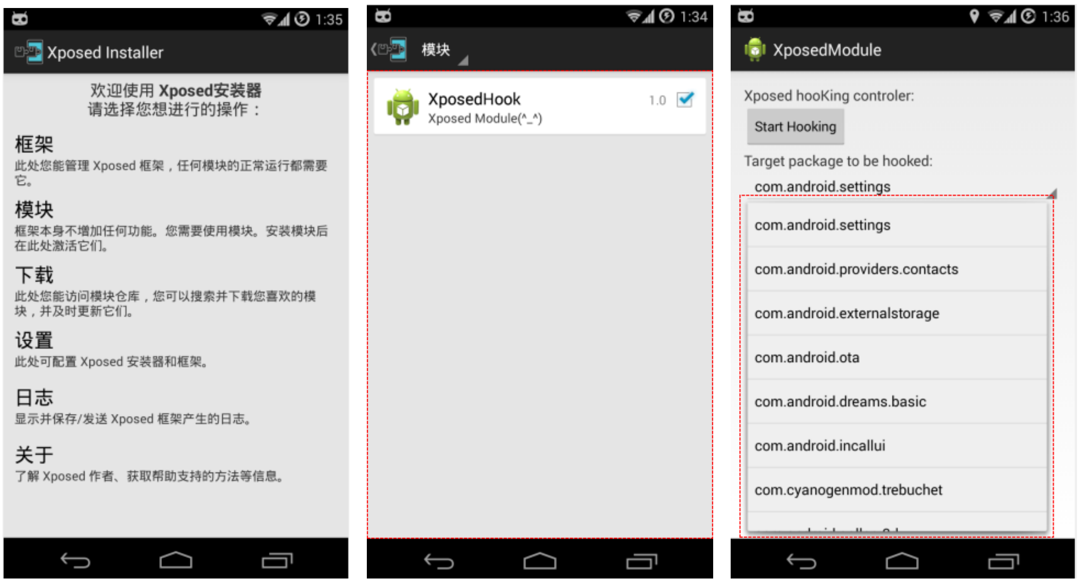
还记得前面提过的配置文件 xposed_init 吧。在 assets 中新建资源文件 xposed_init，其中只需加入一行，即所编写的 Xposed Module 的完整类名，以使 Xposed 框架 Hook 相应目标函数。

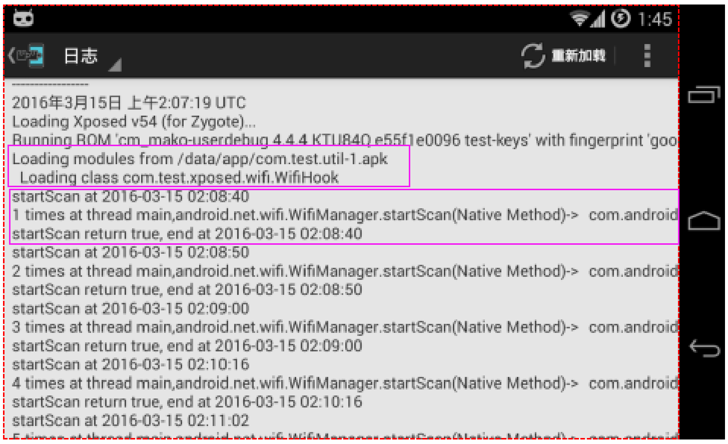


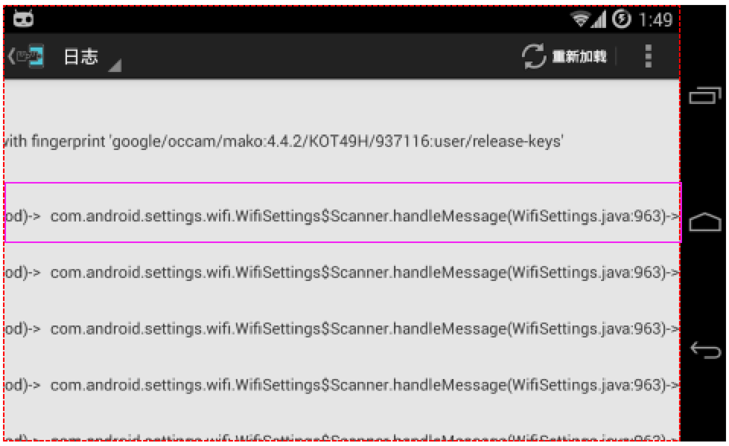
最后，按照普通 apk 的打包流程，打包成 apk，安装使用即可。

4、Wifi 扫描的 Hook 效果

如图，加载自定义的 Xposed 模块之后，从日志中可以看到模块相关的信息以及 Hook 到的 startScan()方法的调用情况。图中以系统应用”设置（com.android.settings）”为例，Hook 其 Wifi 扫描方法，可以追踪到 startScan()方法是在源码文件的 WifiSetting.java 的 963 行调用的。







5、结尾

将目标应用设为我们的被测产品，操作之后获取日志，并将日志文件发给开发 G 和 Y 童鞋。终于可以理直气壮地得出结论——多出的 Wifi 扫描的确是 SDK 中触发的，而且是在源码哪些地方触发的。证据确凿，还找到了问题的所在，不得不感慨 Xposed 的强大。

缺陷分析的正逆向

◆作者：王琳

网上流传一句话：世界上最难的事有两件，一是把自己的思想装进别人的脑袋，二是把别人的钱装进自己的口袋。当然后来又有神续：前者成功的是老师，后者成功的是老板，两者都成功的是寺庙。当然这是开玩笑，说这个是为了引入我们今天的话题：缺陷分析。

作为测试工程师，最常接触的就是各种缺陷，如何将缺陷变废为宝，对今后的测试工作产生指导价值，这是我们做缺陷分析的终极价值，也相当于把别人口袋的钱（代码bug）装进自己的口袋（指导价值）。至于另外的一件把自己的思想放进别人脑子里的事，也是比较高难度的，今天笔者就尝试给大家分享一种缺陷分析的思路，如果读者读过本文如能产生共鸣或者火花，那这也算是完成传播思想这个难事了。

言归正传，缺陷分析如果按照现实工作的模式，大致应该分为两个过程：缺陷定位和缺陷报告。这两个过程可以具体化表示，缺陷定位是一种从表现到内在原因的查找过程，类似几何证明题里的分析过程，由最后的结果逆向求证需要的条件；而缺陷报告就是根据那些分析再从正向梳理整个问题发生的过程并总结收益，类似几何证明题里最后的证明步骤，呈现出来的是从触发条件一路到最后的结果的正向演进。针对两个过程，笔者以手机 QQ 浏览器（iPhone）项目案例和大家分享下思路。

一、缺陷定位（逆向求证）

几何证明题中，要求证一个结果，例如 A=B，就要根据 A=B 的这个结果去寻找成立的条件，一步步的条件往上推演，直到找到已知的对应条件。这个过程就是一个逆向的求证过程。在缺陷分析中也要借鉴这种思路，根据缺陷现象（来自测试发现或者用户反馈），分析产生缺陷的可能原因，从直接原因到间接原因，再到根因，逐步推演到实际的代码层面上。

在这个逆向求证过程中，笔者推荐使用中医的望闻问切的四诊法。《难经》有曰：“望而知之者，望见其五色，以知其病。闻而知之者，闻其五音，以别其病。问而知之

者，问其所欲五味，以知其病所起所在也。切脉而知之者，诊其寸口，视其虚实，以知其病，病在何脏腑也。”换成测试语言来说就是要通过“望”这种观察方式查看现象，确定是否是 bug；通过“闻”这种体验观察缺陷的特征，为后面的判断做辅助依据；通过“问”这种验证方式判断缺陷可能产生的原因和时间；通过“切”这种方式定位缺陷产生的根本原因。整体思路如图 1-1 所示，说的比较抽象，接下来结合实际案例来解释下。

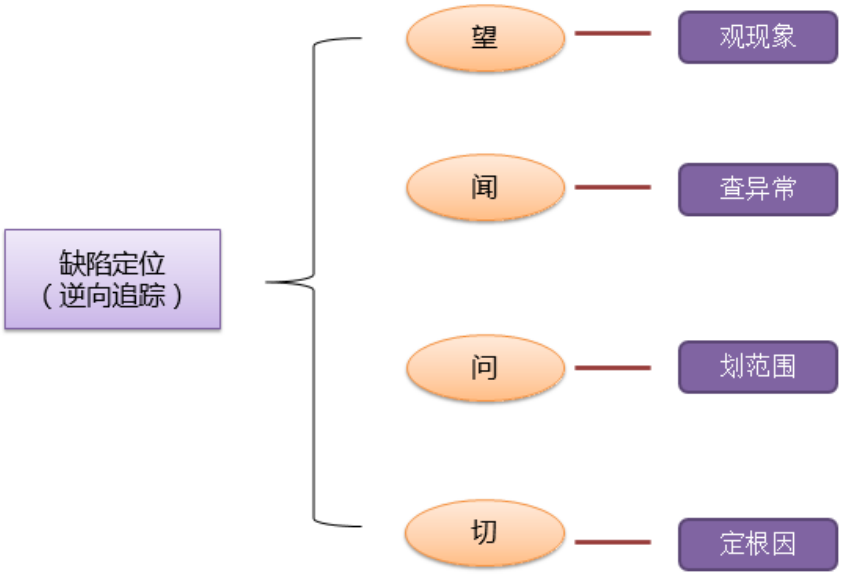


图 1-1

案例：掉粉 30 万事件

1.1 望

手机 QQ 浏览器 6.5 版本 appstore 发布后，发现日活突然少了 30 万左右，持续多日。这就可以判断是“有问题”了，因为观察到日活的减少时与新版本发布有关，且是突然下跌几十万，这与以往的某天突然下跌日活不同，可以排除线上运营活动或者其它的非版本问题的原因，产生的异常与新发布的版本质量有关系，是个可以进行分析的版本质量缺陷。

在望这个环节上，我们得到抽象的信息。6.5 版本用户日活减少了三十万。

1.2 闻

知道了病症的表现，还得通过闻的方式来观察这个症状。医学上是听五音，闻味道。在 IT 行业里就是要逐个可能的维度里去观察异常，如果各个维度都正常，那还得

扩展维度。如果某个维度（例如某种机型、系统、版本等等）存在问题，就可以获得有价值的线索。

这个“闻”的成本跟被观察的缺陷类型有关系。在掉粉 30 万这个案例中，由于现象比较抽象，原因可能很多，逐一检测，耗时较长，到最后找到异常的特征，前后花了将近三天的排查时间。发现 iOS7 系统的用户在发布了新版本后，日活从之前的三十多万将至百人以内，如图 1-2 所示。

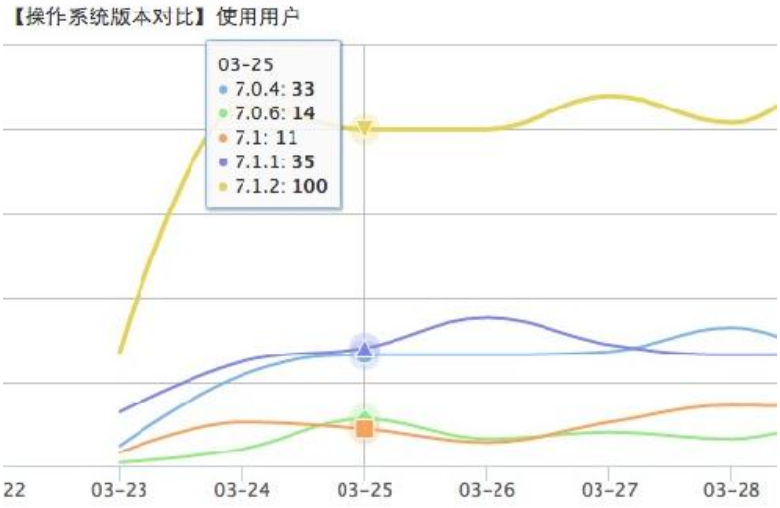


图 1-2

在闻这个环节上，我们获取了进一步的信息，iOS7 系统的用户在 6.5 版本发布后出现了问题。

1.3 问

“望”和“闻”都是测试或者开发人员没有经过与缺陷的亲密接触而获取到的信息。要想定位具体问题还是要“问”。如果说“望”是直观得到的信息，“闻”可能还需要开发帮忙梳理可能的维度，那么“问”这个维度更多的考验测试人员自身的功力。“问”的主要目的是确认缺陷所在区域、缺陷产生的历史时期、缺陷的影响范围。通过测试人员与缺陷的亲自交互，为项目组提供专业的缺陷定位分析。

在案例 1 中，iOS7 系统用户日活上报陡降，可能有两个原因：一是 iOS7 系统的用户装不上新版本或者无法正常启动；二是 iOS7 系统用户能够正常使用，只是上报信息出现问题。这是需要测试人员来验证的，首先在上线前测试时验证过 iOS7 系统的安装与启动，其次从 appstore 渠道下载最新版，在 iOS7 各个子系统和机型上尝试，都不会

出现无法启动的问题，原因一排除。通过连接灯塔系统，发现 iOS7 系统在所有的上报信息都无法查找到。判断原因二有很大的嫌疑。进一步确认是终端版本没有上报，还是灯塔系统出现故障。通过抓包确认，终端确实没有上报信息到后台，因此判断为版本质量问题。这便确定了缺陷所在的区域是终端版本。

接下来判断缺陷产生的历史时期，6.5 版本存在这个问题，那么自然的要确认 6.4 版本和即将发布的 6.6 版本有没有同样的问题，经过同样的验证方法发现 6.4 和 6.6 版本都没有此问题，缺陷就发生在 6.5 版本。

最后还需要判断下缺陷的影响范围。按理在 6.5 引入缺陷后，没有被发现之前，开发是不会修复的，6.6 版本也应该存在此问题。可是这里却要存疑，何故 6.6 版本“自动”修复了缺陷？

在经过“问”这个环节后，我们获取到了更进一步的信息。缺陷只存在于 iOS7 系统上 6.5 版本，终端不能够上报数据，灯塔系统正常。提出可疑点，6.6 何时自动修复了缺陷。

1.4 切

当测试人员把问这个环节的获取的信息提交给项目组后，就需要项目组进行 CodeReview，来查找所有涉及 iOS7 网络上报的修改代码。开发根据提供的信息找到可疑代码如下图 1-3 所示。

图 1-3

进一步展开分析，发现在修改小说网络层缓存时影响到了 iOS7 的网络上报。如下图 1-4 所示。

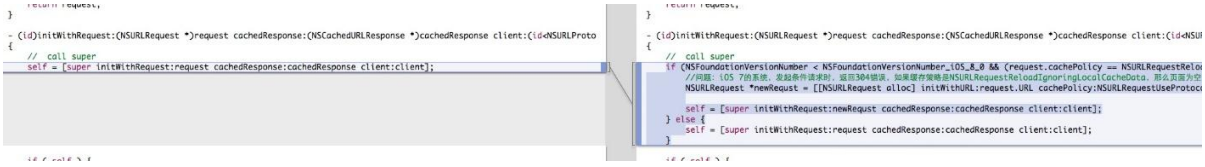


图 1-4

别忘了，还有一个疑问点，为何 6.6 自动修复了缺陷？再“切脉”发现有一位开发当时进行 CR 时觉得上图的代码写的不够好，可能存在隐患，就给重构了，也就避免了 iOS7 上报出现问题，但是这位开发自身也是不清楚之前还存在这个问题，所以缺陷就在悄无声息中解决了。

到了切这个环节，我们已经基本定位出了缺陷产生的根本原因。新手开发在修改小说的问题时，影响到了 iOS7 系统的网络上报，导致了 iOS7 系统用户在安装 6.5 版本后无法上报信息到灯塔，从而导致日活陡降 30 万。解决方法只能靠快速发布 patch 版本来进行修复。

纵观整个望闻问切的过程，就是从现象逆向追踪到本质的一个定位，也是一般 bug 分析的必经过程。很多时候我们写出来的报告像是事后诸葛亮的感觉，就是仿佛已经知道了原因，再来分析影响。还原缺陷分析的真实过程对我们再进行分析能够起到很好的梳理作用，明白自己所处的阶段。

二、缺陷报告（正向演进）

对项目组开发来说，一般从缺陷能够逆向追踪到产生的代码原因，解决了基本上工作就结束了。但是我们测试人员在这个过程中还可以挖掘更多的有价值的点。这也是要写缺陷报告的原因，通过上一章的逆向追踪，我们已经可以知道缺陷相关的情况，如何变废为宝，将缺陷的价值挖掘到最大，并指导今后的测试就看这个正向演进的过程。

如何呈现一份好的缺陷分析报告，也是一个考验功力的时候。本文笔者建立一个缺陷分析模型如下图 2-1 所示。既要聚焦于当前的问题，又要有宏观分析的视角。从现象到思考逐层深入分析，每一层又对应着宏观的分析。通过该模型的方式演进缺陷分析，能够将问题剖析的相对清晰，指导今后测试也有借鉴意义。

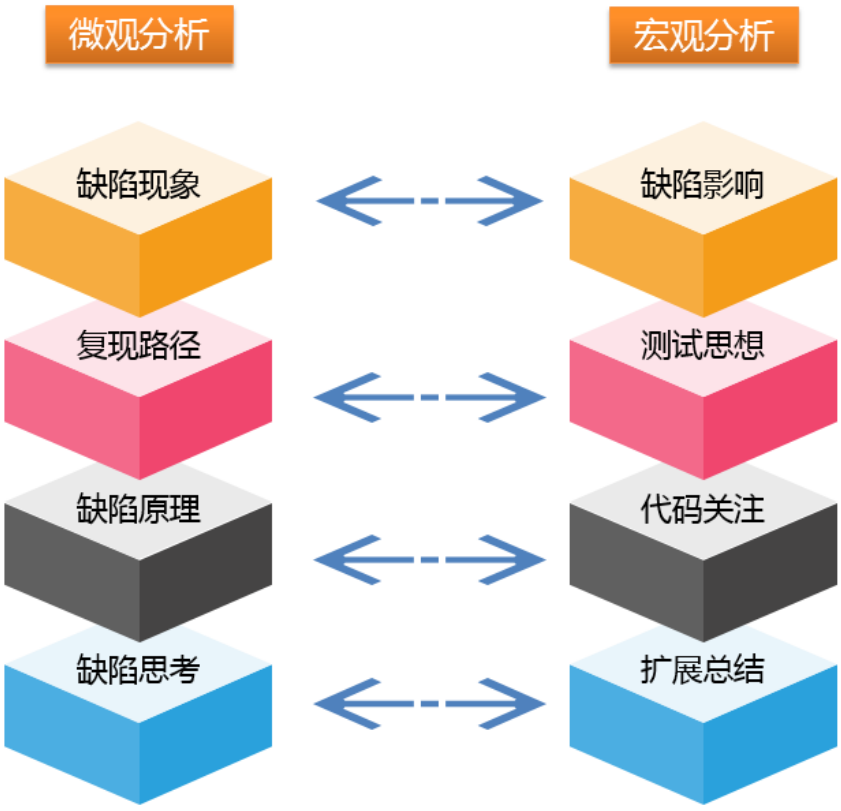


图 2-1

图 2-1 所示的模型主要是说明缺陷报告要分为四层，分别是缺陷现象、复现路径、缺陷原理、缺陷思考，这四层是我们做一份完整的缺陷报告不可缺少的，也是层层递进的一个逻辑关系。右侧的是宏观方面的对应，就是除了要就事论事分析 bug，还要跳出当前 bug，看的更加宏观一些，思考广角要大，宏观的四层也是一次递进的关系。具体可见下文讲解。本节内容搞的案例承接上文，仍然采用手机浏览器掉粉事件。

2.1、缺陷现象——缺陷影响

首先要描述缺陷发生时的现象，这个容易做。站在宏观的角度来看，还要列出缺陷产生的影响，如表 2-1 所示。

表 2-1

影响维度	影响程度
必现程度	必然出现，大概率出现，很难出现
机型系统	全系统都出现，新系统出现（例如 iOS9），少数低端系统出现（例如 iOS6），常用系统出现（例如 iOS7、iOS8）
路径易触发性	较少使用，频繁触发
严重程度	闪退，重启或者重试可恢复的 bug，一直不消失的 bug
修复成本	通过后台发布更新来修复，或者发 patch 版本来修复

对应本文案例记录如下。

缺陷现象： 6.5 版本的 iOS7 系统用户无法上报使用数据。

缺陷影响： 如表 2-2 所示。

表 2-2

影响维度	影响程度
必现程度	必然出现
机型系统	iOS7
路径易触发性	必然触发
严重程度	不影响用户使用，只影响统计
修复成本	发 patch 版本来修复

总结一下这一层面的描述要求：

缺陷原理的描述要求： 有图有真相，闪退附日志。（如果可以现象图，最好附上，便于读者能够清晰认识到缺陷的现象，如果是闪退了，还需要附上日志）

缺陷影响的描述要求： 机型必现度，修复影响度。（主要是指表格中的项都要有完整说明，对于了解缺陷在宏观层面上的影响有所帮助）

2.2、复现路径——测试思想

接下来要描述复现 bug 的路径，每个复现路径都蕴含着一定的测试思想。这里为了抽象，可以引用探索式测试全局测试策略中的一些方法，例如下图 2 -2 所示的几个举例。实际情况是可能是几种测试思路的组合。

图 2-2

对应本案例记录如下:

复现路径：在 iOS7 手机上，安装 6.5 版本（全新或者升级安装），启动浏览器，无上报信息。

测试思想：指南针测试法（按照需求进行验收，iOS7 系统用户上报是必备的）。

总结一下这一层面的描述要求:

复现路径的描述要求：言简意要明，完整路径全。（语言简洁明了，不能大篇叙述，必要时可以考虑用流程图等不同形式来展示，其次路径需要描述完整，如果存在多个复现路径或者对应多种现象都需要注明，不要遗漏。）

测试思想的描述要求：路径含思想，关联知识库。（这个是指复现 bug 的操作应该蕴含一定的测试思想，用可理解简要词汇表示，可以记录到 bug 的知识库里，也可以从 bug 知识库里寻找历史相关的 bug，重新整合思考。）

2.3、缺陷原理——代码关注

在微观上了解了缺陷现象和复现路径，还需要深入说明缺陷原理，缺陷原理一般落实到代码层，有时候考虑到成本问题可以只到逻辑层。可以采用 5W 法，提问逐层分析问题产生的原因。这个 5W 涉及的问题可以是在逆向求证的时候思考过的，也可以是就测试人员为何漏测或者开发人员为何写出缺陷代码进行提问。

定位缺陷发生的原理后，开发和测试人员都要提炼出关于今后写代码需要格外关注的点，将相关逻辑进行完整排查，确保没有类似的问题。

对应本案例的描述如下:

缺陷原理:

1、为什么手机浏览器日活近期掉粉 30 万？

因为发布 6.5 版本，安装了 6.5 版本后，iOS7 系统上的日活陡降 30 万。

2、是什么原因导致的日活陡降？

因为 iOS7 系统用户使用 6.5 版本浏览器后无法上报日志到灯塔系统。

3、所有 iOS7 系统有这个问题吗？会不会跟机型有关系？

所有 iOS7 系统都存在此问题，iOS8、iOS9 都是没问题的。与机型无关。

4、在 6.4 版本和即将发出的 6.6 版本没这个问题吗？

旧版本和即将发出的新版本都没问题，问题只存在于 6.5 版本上。说明是 6.5 引入的，在 6.6 上被无意中修复了。影响范围就限于 6.5 版本，需要发布 patch。

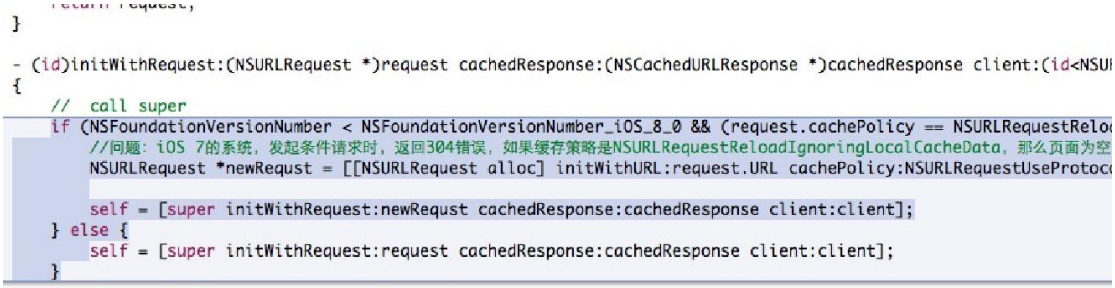
5、是 6.5 版本什么时候引入的，什么情况下引入的？

在 6.5 的灰度阶段引入的，为了修改小说页面刷新后空白的 bug，如下图 2-3 所示修改了网络层缓存机制，导致 iOS7 系统的网络上报存在问题。

图 2-3

6、开发的 CR 为何没有发现问题？

当时是过了 CR，但是没有发现可能存在隐患，只是觉得如下图 2-4 所示的逻辑实现不是很优雅，就暂时没动。后来在 6.6 的时候另外一个开发看着觉得逻辑比较乱，就重构了一下，自动修复了这个问题。



```
}

- (id)initWithRequest:(NSURLRequest *)request cachedResponse:(NSCachedURLResponse *)cachedResponse client:(id<NSUI
{
    //  call super
    if (NSFoundationVersionNumber < NSFoundationVersionNumber_iOS_8_0 && (request.cachePolicy == NSURLRequestReloa
        //问题：iOS 7的系统，发起条件请求时，返回304错误，如果缓存策略是NSURLRequestReloadIgnoringLocalCacheData，那么页面为空
        NSURLRequest *newRequst = [[NSURLRequest alloc] initWithURL:request.URL cachePolicy:NSURLRequestUseProtoco

        self = [super initWithRequest:newRequst cachedResponse:cachedResponse client:client];
    } else {
        self = [super initWithRequest:request cachedResponse:cachedResponse client:client];
    }
```

图 2-4

7、测试为何没有暴露问题？

这个问题反映了测试上报时候对系统的不关注问题。在问题发生之前，测试统计上报的测试都是不分系统，只要在一个系统上做了验证即可。理论上对上报类的只跟网络层有关系，与系统没有关系，但是从修改的代码中可以看出是针对 iOS7 系统做了单独

处理，应该进行多关注。今后测试统计上报，会适当兼顾各个系统的情况。

代码关注：

开发需要关注网络层修改的问题，尤其是灰度后需要多找几个人进行 CR，新人开发更是要多注意。测试人员在发生这个 bug 后，需要关注开发提交的影响点中涉及系统的网络层修改，在不同系统上对统计上报做个排查测试。

总结一下这一层面的描述要求：

缺陷原理的描述要求： 5 W深究到底，深入代码层。

代码关注的描述要求：开发关注点，逻辑完整查。

2.4、缺陷思考——扩展总结

在经过前面三个步骤后基本上就能够对缺陷有个全维度的审视和认知。但是要做文章开头说的把别人口袋里的钱放到自己口袋里，还缺最后一个很关键的环节，就是微观上关于缺陷的思考和宏观上的扩展总结，这些也是要呈现在正向演进的报告里，提高测试报告的“含金量”。缺陷思考主要是反问自己是否真的解决了问题，这个看似简单，其实最容易出问题。开发修复了代码后 ，要做个全方位的验证，对相关模块也要做关联测试。宏观上的扩展总结是可选项，可以对历史上类似的 bug 做个梳理，有能力和时间的话还可以对相关的架构做个完整梳理。

对应本案例的描述如下：

缺陷思考：经过修改后，在 6.6 上测试发现 iOS7 上上报 ok，日活在灰度时也会到正常水平。继续做关联性测试，既然日活上报没问题，那么深入看看各个业务在不同系统上的具体上报，发现 6.6 灰度后引入了一个新问题，就是只有初始化状态的上报，没有后续的功能使用上报，于是再次修复。通过关联测试增强了测试人员挖掘缺陷的能力，对类似问题进行了围堵式打击。

扩展总结：

梳理历史上关于统计上报的知识库。过往只有某些业务上报存在不准确或者漏报的问题，现在可以增加对系统的覆盖检查，上线前的 checklist 里也可以增加一列。

总结一下这一层面的描述要求：

缺陷思考的描述要求：确认不复现，关联性测试。

扩展总结的描述要求：历史相关联，架构梳理清。

由此正向演进的过程梳理完毕。

三、写在最后（讨论分析）

如下图 3-1 所示，概括了是本文的总体思路。缺陷分析的正逆向分析，逆向求证重点在于从缺陷现象通过望闻问切四诊法寻找到问题的本因，也即缺陷定位，而正向演进侧重报告输出和逻辑理顺，从得知根因开始，从根因开始理顺影响和整理测试思路，完善知识库，指导今后更好的工作。

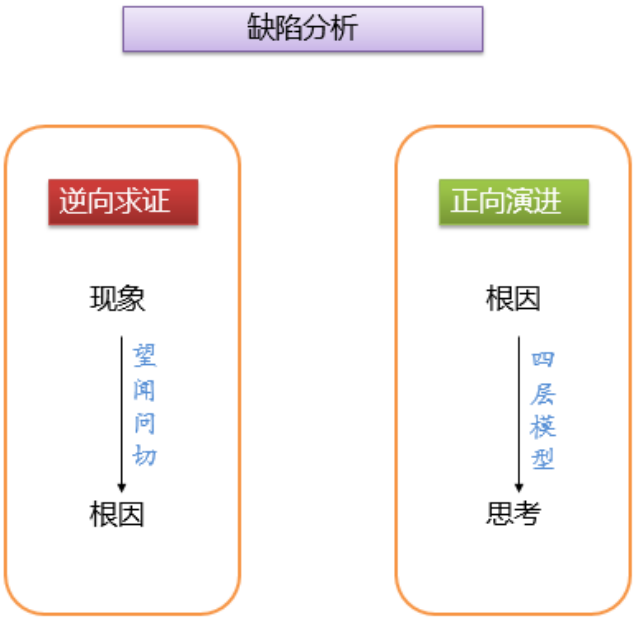


图 3-1

在日常的测试工作中，我们与开发之间的互动主要体现在逆向求证的相互帮助和讨论中，逆向求证是能够体现团队协作和提高测试影响力的过程。最终的输出是缺陷报告的形式，可以参考本文提供的模型来进行梳理总结和反思。

在文章的最后，还是要讨论三个老生常谈的问题，任何一个理论或者模型的使用必然伴随着一个挑战，就是投入产出比如何？适用人群是怎样的？适用阶段是什么时候？本文就这三个问题和大家再进行一次讨论。

3.1、投入产出比

投入产出比我们分两个方面来分析，首先是线上反馈或者研发过程中的严重问题（例如闪退、功能障碍），不存在投入产出比的顾虑，因为这些问题都是必须验证和解

决的。这里涉及投入产出比主要针对的是对日常的研发 bug 或者线上反馈的普通问题。笔者推荐如下思路。

缺陷定位（逆向求证）如何降低投入:

通过望的方式决定是否要进行下一步，如果问题判断影响不大或者没必要做进一步的分析，就可以在这一步终止。如果问题比较严重，就继续下一步。

在闻的这一环节，确认异常发生的位置，如果能找到，可以安排新人或者外包去进行“问”这一环节，例如验证发生版本、影响机型范围等。在这里可以减少一部分人力成本。

在切的这一环节，如果情况比较复杂，就需要联系开发进行处理，测试人员要视情况判断是否要深入参与，一般性的问题，经过前三个环节后基本可以交给开发来处理了。要紧的问题可以由测试人员继续跟进，直到结束。

缺陷报告（正向演进）如何提高产出:

练习本文的四层模型或者自己认为合理的模式进行缺陷报告总结，形成固有套路，提高报告撰写效率。

学习探索式测试思想，对测试过程中的操作能够快速找到指导思想，方便后续知识库的维护。

日常多思考，一个报告的缺陷思考和扩展总结方面可以邀请他人一起参与讨论，碰撞出思想火花，集思广益，提高思考的产出。

结合测试人员实际的经验进行判断，通过这种一升一降的方式，提高自己在缺陷分析方面的投入产出比。

3.2、适用人群

这个很多会问，这种缺陷分析是否适用于外包测试同学，新加入的正式员工合适可以开始做缺陷分析？我们一一来讨论下。表 3-1 所示，“✓”代表适用，“*”代表不适用。这也只是个参考，具体还得因人而异，总体来说，各个角色都是可以通过学习和练习的方式逐步提高缺陷分析能力。手机 QQ 浏览器（iPhone）项目上从正式到外包同学启动了全员缺陷分析模式，效果还是不错的。

表 3-1

适用人群	逆向求证				正向演进			
	望	闻	问	切	缺陷现象	复现路径	缺陷原理	缺陷思考
新人	✓	✓	✓	×	✓	✓	✓	×
有经验者	✓	✓	✓	✓	✓	✓	✓	✓
外包骨干	✓	✓	✓	×	✓	✓	✓	✓
外包新人	×	×	✓	×	✓	✓	×	×

3.3、适用阶段

适用阶段这个因项目而异，在 iPhone 浏览器项目组中，和项目组达成的共识是在集成测试后发现的闪退问题都要做缺陷分析，不管是因为集成引入的还是之前测试遗漏，都要做分析并同步出来。另外在每个版本结束到下个版本开始，每个外包测试同学会对自己负责的模块的典型的 bug 做根因分析，以此作总结回顾，并思考接下来的测试侧重点。

探索式测试体系--缘来就是你

◆ 作者：王琳 赵燕 谢琳

一、前言

在做了一段的测试工作后，测试人员一般都会遇到这样一种情况：自由测试中往往比执行用例时更加容易发现问题，尤其是在进行过一轮用例测试后，后面的回归和集成测试中通过执行用例能够发现的问题几乎很少（“农药悖论”理论）。很多人提议放弃用例执行，改为自由测试。此时问题就来了，已解决的 bug 又出现了怎么办，自由测试中怎么度量测试质量，不同人执行测试的差异怎么平衡，测试经验如何传承，测试覆盖范围如何保证不遗漏……

带着这些问题，我们来介绍下业界非常火热的探索式测试，并重点阐述我们的测试工作中是怎样运用探索式测试思想来进行实践的。

二、缘定三生

在上个世纪 60 年代，人们就发现偏离预设的用例进行测试能够发现更多的问题。很多时候，我们称这种行为为自由测试，随机测试，monkey 测试。后来发展到一定的阶段，经历了方法论建设，这种行为又被演绎为探索式测试。所有的概念都要先了解形成过程，才能更好的理解现在的发展情况。探索式测试也不是凭空而来，到目前为止，大致经历了四个阶段（下图中 2.0，其中 ET 是 Exploratory Testing 的缩写）。

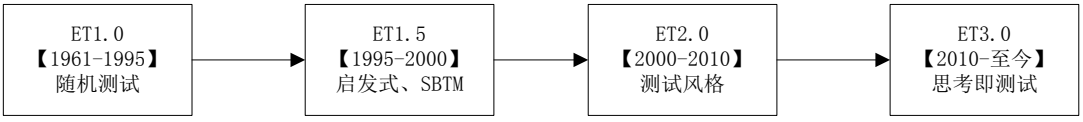


图 2.0

2.1、ET 1.0

【1961-1995】“Put aside your scripts and look at the product! Interact with it! Find

bugs!”人们发现在做没有用例的自由测试时能够发现更多和更好的 bug。探索式测试作为一种术语和理论的第一次迭代旨在脱离脚本的束缚，给“更好的测试”腾出空间。“把你的脚本放到一边去，观察产品，与它交互，发现错误!”探索式测试的拥护者将探索式测试视为一种技术，直到现在仍然有许多人持有此观点。当然从现在的视角来看这种方式表达探索式测试是错误的，但在当时具有一定的指导意义。

2.2、ET 1.5

【1995-2000】“Compare and contrast the important structures of scripted and exploratory testing and the relationships between them”这个阶段主要是在纯粹的随机测试中加入了引导性因子。启发式探索式测试在此阶段提出。启发式探索式测试为实时测试者提供了全面的底层测试模型作为参考，引导和规范探索式测试活动。人们开始比较脚本和探索式测试的结构及关系，而不是再将其视作感觉上的不同活动。James 在此期间还尝试区分随机测试和探索式测试，探索式测试是有技能的，更具备描述性的活动。2000 年的时候引入了基于测程的测试管理（SBTM），提高探索式测试在项目上的可管理性。

2.3、ET 2.0

【2000-2010】“This is a sliding bar on which testing ranges from completely exploratory to completely scripted.”在这个阶段人们认为所有的测试活动都是在纯探索式和纯脚本测试之间的范围。停止将探索式测试称为一种技术，取而代之的是称之为适用于技术的测试风格。2006 年定义了“同时进行学习、测试设计和测试执行”的探索式测试的简单定义。还提出了“检查”是可以自动化的，而“测试”是不可自动化的，以及非智慧”和“智慧”行为的差异性。

2.4、ET 3.0

【2010-2015】“The differences between mimeo-morphic actions and poli-morphic actions.”发展到这个阶段，只要是富有责任感带着思考力的去做测试，那么就是在做探索式测试，在测试过程中利用线索，积极去思考，抛弃固化的思维进行测试。另外一方面，脚本测试可以对测试人员进行帮助，或者将脚本测试视为一种可用的工具或者技术。最终人们做的是一种有脚本化的探索式测试。

看完了探索式测试的前世今生，是不是觉得其实早已相识了？这说明你与探索式测

试早已缘定三生呢，值得寻找今世的缘分！

好了，言归正传，现阶段比较通用的探索式测试应该还是 ET2.0 以及之前的内容，ET3.0 固然与时俱进，但是在指导现阶段的测试工作来说还是有些务虚的成分。怎么说呢，我们可以从思想高度上达到 ET3.0 的境界，也即测试就是探索式测试，否则就是没有思考力的机械行为。但是实际指导测试工作，我们可以结合 ET2.0 之前的方式开展。

三、初恋的味道

初恋是甜蜜的，与众不同的，只有恋过的人才会知道，就像初识探索式测试一样。在学习探索式测试的过程中，也会有酸甜苦辣，只有了解它的人才知道这种味道。不妨和探索测试一起再回味一下初恋的味道。接下来笔者将详细介绍如何入门探索式测试。

3.1、方法介绍

有些人会觉得探索式测试无异于自由测试，其实不然，探索式测试是有一套成熟的方法论体系，其中的方法就是掌握探索式测试法宝。前人总结的探索式测试方法同样适用于移动互联网测试如下表 3-1 所示，根据移动 APP 的实际情况可以重新进行梳理归类。

表 3-1

基础类方法	指南法、反叛法、懒汉法、破坏法、取消法、极限法、强迫症法、测一送一法
深入测试方法	地标法、快递法、长路径法、深夜法、通宵法、遍历法、收藏家法
分区域测试方法	卖点法、配角法、恶邻法、上一版本法、深巷法

◆ **基础类测方法**

1) 指南法：指南法顾名思义就是按照需求文档、用户手册或是测试建议进行的测试，检查产品功能是否有按照预期实现。

2) 反叛法：输入最不可能的数据，或是已知恶意的输入，检查程序对于异常输入数据的处理能力。包括非法输入内容和错误的输入顺序。

3) 懒汉法：测试人员做尽量少的工作，接受所有默认值，检测应用处理默认值和

空白值的能力

4) 保持输入字段为空，或不勾选任何选项。

5) 破坏法：破坏应用运行的环境，数据、资源或权限，再去执行相关的操作，检查应用的表现

6) 取消法：启动操作后在停止它，针对比较耗时的操作，检查应用的自我清除能力，重新启动操作能正常开始并结束

7) 极限法：向软件提供极限条件或难以回答的问题，查看软件的处理能力，包括数据极限，操作极限，时间极限。

8) 强迫症法：重复输入相同数据，反复操作同一个按钮

9) 测一送一法：用户同时在不同地方操作同一个用户数据对象

◆ **深入测试方法**

1) 地标法：把应用中的功能点当作地标，从一个地标执行到另一个地标来探索应用程序

2) 快递法：确认特性所使用的内部数据，通过操作软件得到该数据走遍其相关特性，测试人员使用该方法时重点关注数据的流动是否始终正确。

3) 长路径法：1.确定测试目标，到达目的地之前尽量多地在应用程序中穿行；2.埋在应用程序最深处的界面作为测试目标

4) 深夜法：卖点特性停止运行后，测试其它维护任务如数据归档、备份文件等

5) 通宵法：让程序一直保持运行而不去关闭（重复运行自动化脚本且从不停机）

6) 遍历法：最短路径来过完软件的所有明显的功能

7) 收藏家法：收集软件的输出，越多越好

◆ **分区域测试方法**

1) 卖点法：产品主打的亮点功能，按照产品演示的步骤执行

2) 配角法：找到和主要特性一起显示或运行的特性，从而对这些特性进行额外的测试和关注

3) 恶邻法：bug 多、相关模块

4) 上一版本法：上一版本支持的场景，在新版本的表现

5) 深巷法：关注最不可能被使用或最不吸引用户的特性和小功能

3.2、用例设计

ET 的测试用例更像一种思维导图，或者思维引导，没有具体的形态，ET 需要的就是一种测试思维，测试经验，不需要罗列具体的测试步骤.测试管理者在分配任务的时候，可以指出测试的切入点，以及可能出现的问题点。

测试人员按照 ET 的思维导图中的测试点进行测试执行，分为检查和探索两个步骤，检查主要是检查产品功能是否有按需求实现，探索主要是按照漫游测试方法探索软件的各种路径和场景，如图 3-1 所示是探索式测试用例设计。

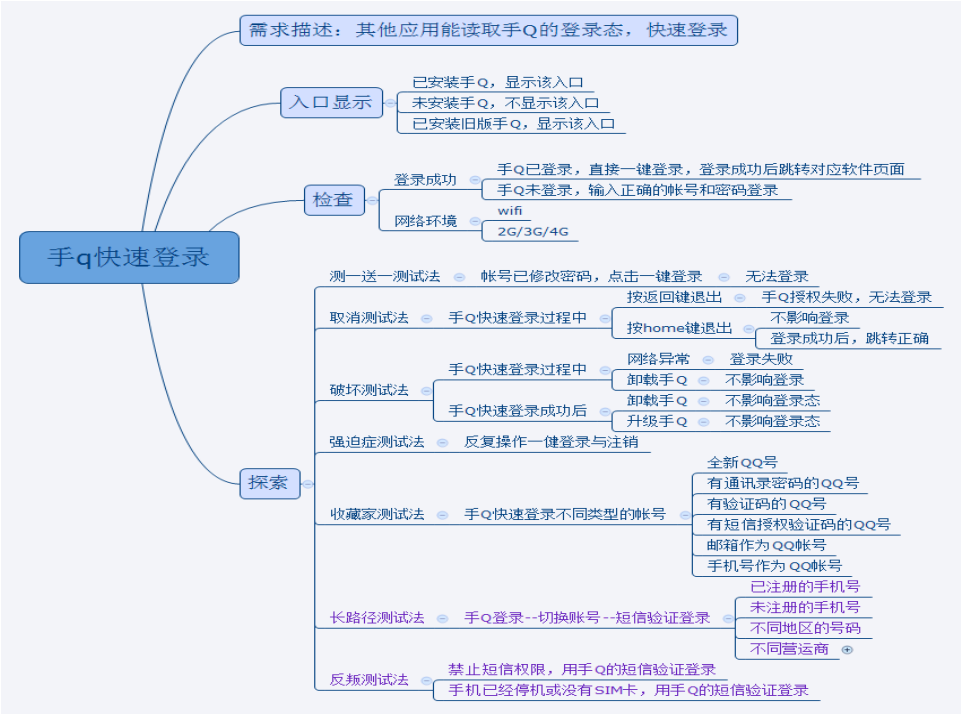


图 3-1

3.3、探索章程

这部分介绍如何快速制定探索式测试章程。

第一步走：圈地盘

确定探索测试的目标，资源以及相关的信息，目标尽量控制不要过大，举例如下。

“目标：探索微信朋友圈发照片功能。创造不同的测试数据和场景，以图发现朋友圈发照片的权限“

第二步走：找变量

确定影响测试的因子，如下表 3-2 所示

表 3-2

照片来源	照片大小	网络	照片格式
系统拍照	大于 10M	WIFI	png
第三方软件保存	5M	4G	JPEG
截图	2M	3G	TIFF
缓存图片	200K	2G	BMP

第三步走：再探索

根据影响测试的因子，进行组合测试法，组合各种 Case。

四、蜜月之旅

读完上一章，想必大家和探索测试一起回味过初恋的味道，那么接下来我们就和探索测试来一场蜜月旅行吧，在蜜月的旅途中我们能进一步增进和探索测试的情感交流和融合。

所谓的和探索测试的“蜜月”——就是在学习和实践探索测试方法的过程中，我们会增加一写趣味性的实践活动，我们将测试变得有趣的理论落地，同时提供了详细的指导方案，我们总结了一下四种趣味的“蜜月”活动：

4.1、脑风暴活动

在一个和谐的气氛中集体测试，重复发挥漫游探索测试的特点，汇集不同的测试同学思维方式，给模块测试同学提供更广阔的测试思路，如图 4-1 所示。

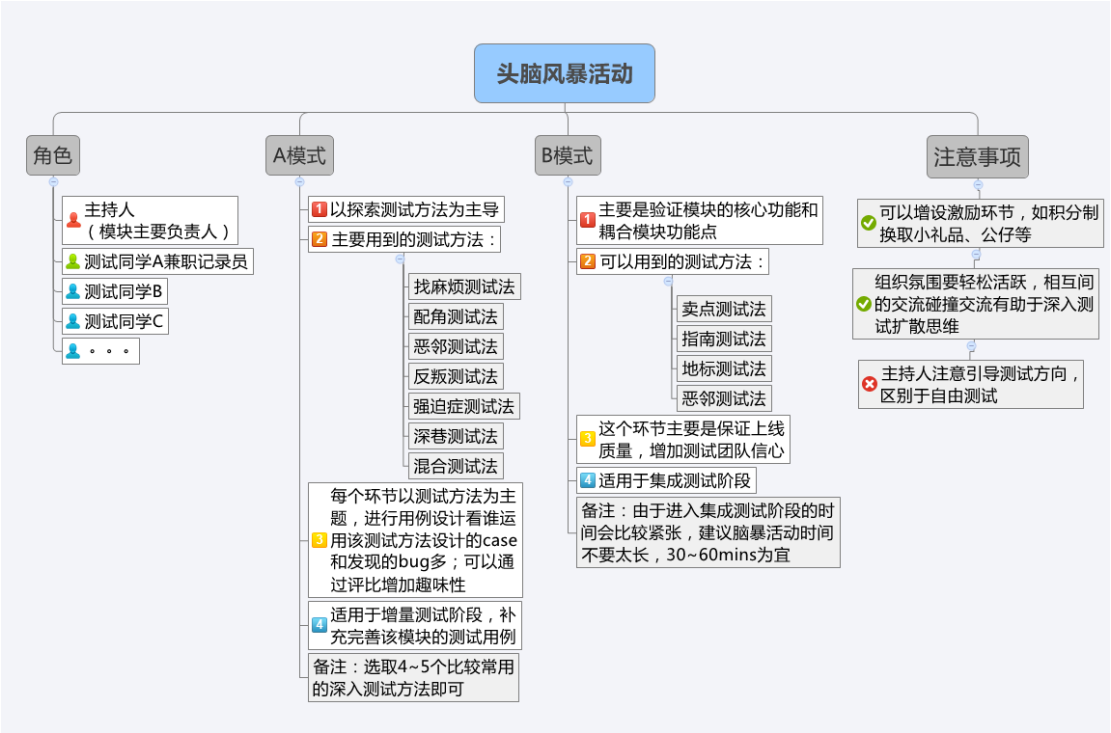


图 4-1

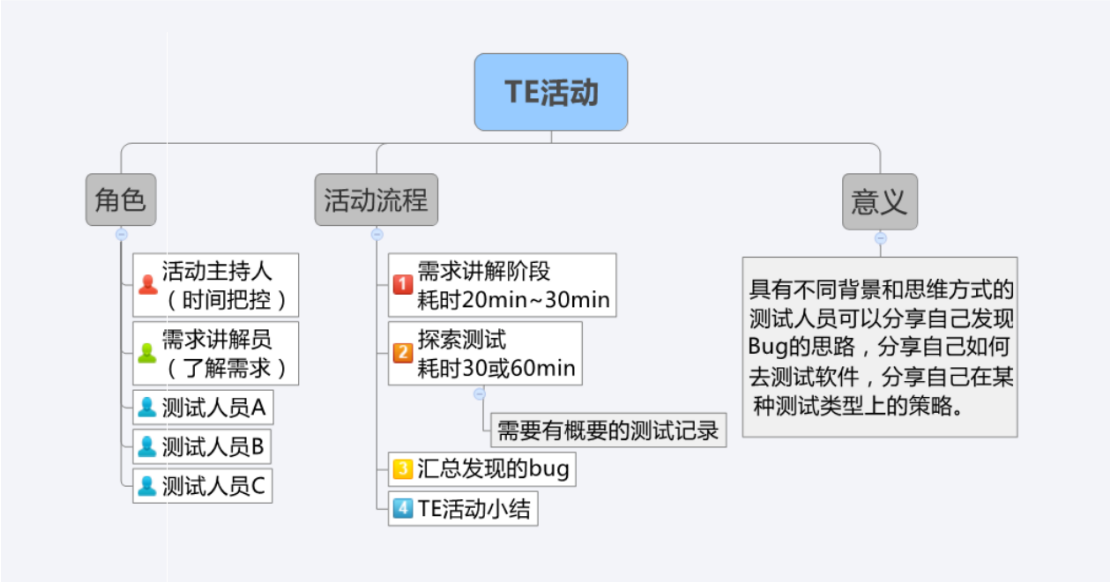
A 模式:（适用于增量测试阶段，补充完善该模块的测试用例）每个环节以测试方法为主题，进行用例设计看谁运用该测试方法设计的 case 和发现的 bug 多；可以通过评比增加趣味性;

B 模式:（适用于集成测试阶段）这个环节主要是保证上线质量，增加测试团队信心;

通过简短的探索式测试的头脑风暴，可以产生多样化的测试策略和测试用例，深入挖掘 bug。

4.2、Team Explore

集体专项的探索测试活动 Team Explore，又简称为 TE 活动，集中测试但在测试过程中互相不被打扰，提供一个紧张、竞争的测试气氛，为激发测试同学的测试激情和集中注意力;



在限时的压力下，大家各自运用探索测试方法，进行集中测试，活动的主持人必须把控好时间，还需要对被测模块了解，能够判定是否为缺陷或是已知 bug。在这个活动中会相对于其他实践活动较为严肃和紧张，效率也会相对高一些，大家的测试记录也比较灵活，可以手工简单纪要，活动场景如图 4-2 所示，测试结果记录如图 4-3 所示。

图 4-2

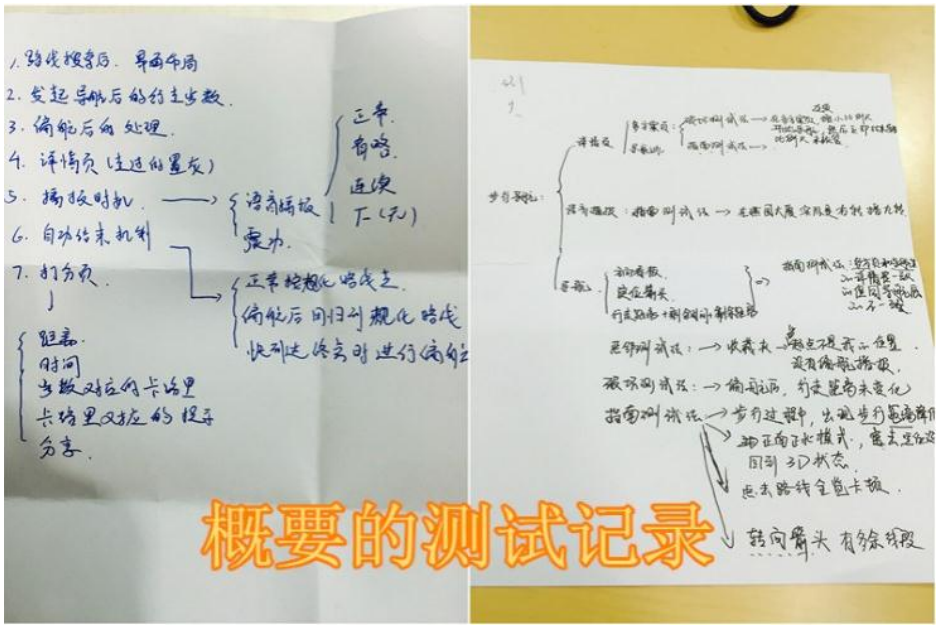


图 4-3

4.3、Bug Bash

Bug Bash 是短期的全员测试活动，是常规测试的有效补充。集体模式的探索测试是相比找到的缺陷，它更有意义的是增加团队合作和凝聚力。

项目组成员都可以使用探索式方法，来体验产品，时间可控制在 1 小时内，利用运用各自的技能和职业背景，集中精力来搜寻软件的缺陷，相信可以给产品保驾护航。

4.4、结对测试

由两名或多名测试人员在一起同时测试同一个模块，并相互给出建议和指导，通过共享使用探索测试方法，从而不断的积累系统知识和测试经验，达到共同进步的目的。

探索测试不但有助于测试人员更加深入理解产品，也更深刻地理解测试思想和测试技术。在学习探索式测试的道路上，动手实践并积极反思总是优于等待观望。专项活动也同时会给枯燥无味的测试工作带来一些趣味和灵感，积极的团队学习行为同时也可以沉淀知识和增强凝聚力。

最后附上整理的探索测试专项活动模板图 4-4 所示，献给大家~~~

专项探索测试活动模式

	测试模块	活动负责人（由）	测试版本号	活动地点和时间	目标	准备资料（需求文档等）	活动记录	发现bug数	经验积累	备注
TE活动（Team Explore）	双击编辑	Frank	版本号	双击编辑	1、提早发现bug，降低软件风险； 2、. . . 3、. . .	所需文件拖放在这里	1 可以是照片 2 可以是笔记 3 可以是视频	严重： 一般： 建议：	1、哪个方法发现了比较多的问题？ 活动小结 2、测试活动还有哪些不足之处？ 3、. . . 实际耗时：	如需要提前准备的物料
头脑风暴活动	双击编辑	Brian	版本号	双击编辑	1、提早发现bug，降低软件风险；	所需文件拖放在这里	1 可以是照片 2 可以是笔记 3 可以是视频	建议： 一般： 严重：	1、哪个方法发现了比较多的问题？ 活动小结 2、测试活动还有哪些不足之处？ 3、. . . 实际耗时：	如需要提前准备的物料
缺陷大扫除活动（BugBash）	双击编辑	Vivian	版本号	双击编辑	1、提早发现bug，降低软件风险；	所需文件拖放在这里	1 可以是照片 2 可以是笔记 3 可以是视频	严重： 一般： 建议：	1、哪个方法发现了比较多的问题？ 活动小结 2、测试活动还有哪些不足之处？ 3、. . . 实际耗时：	如需要提前准备的物料
结对测试	双击编辑	Vivian	版本号	双击编辑	1、提早发现bug，降低软件风险；	所需文件拖放在这里	1 可以是照片 2 可以是笔记 3 可以是视频	严重： 一般： 建议：	1、哪个方法发现了比较多的问题？ 活动小结 2、测试活动还有哪些不足之处？ 3、. . . 实际耗时：	如需要提前准备的物料

图 4-4

五、生活浪漫曲

在经历了初恋的味道，蜜月的旅行，当一切回归平常时，探索式测试可能依然陪伴我们走过岁月？答案是肯定的。

在不同的项目迭代阶段，探索式测试并不是固定的实践形式，而是可以结合项目迭代特点进行多样化实践。本章是生活浪漫曲，主要讲述在测试设计和增量测试中，探索式测试能给我们带来的浪漫因子。

5.1、测试设计

大军未动，粮草先行。在一切测试活动开始之前，都要有一个比较好的测试设计。那么探索式测试在这个环节能够起到什么作用呢？主要是在两个环节上：章程设计和思维碰撞。

5.1.1、章程设计

测试章程根据复杂程度又分为测试说明和规划图。

简版的是测试说明，根据提测需求和开发产品沟通后进行确认的文档，类似于普通的需求说明书，只不过形式更为灵活和简洁，能达到逻辑描述精简即可，对于一些未知的逻辑，也可以在测试说明中进行记录。这个测试说明将为测试执行阶段的一个 tips，激发测试人员的灵感，同样的测试人员也可以在测试执行过程中不断补充和修改这个说明，达到边测试边设计的工作方式。具体案例可以参考下文的增量测试中的举例。

较为具体点的是测试规划图，这个需要找有相关测试经验，对产品比较熟悉的测试人员进行设计。将被测对象放在整个产品上、放在整个平台上进行审视，敲定需要哪些

测试点影响范围。也可以像用例评审一样，召开会议，共同补齐规划图中的遗漏点或者调整逻辑.以下图 5-1 为手机地图安卓版 5.5 路况播报测程规划，可以看出里面的内容十分详细，从工具准备到数据流都有罗列，这样的测程规划图比较适用于特性在版本迭代中作为完整资料保存，对一些关键路径也可以做较好的记录，指导新手做探索式测试也是十分不错的方式。

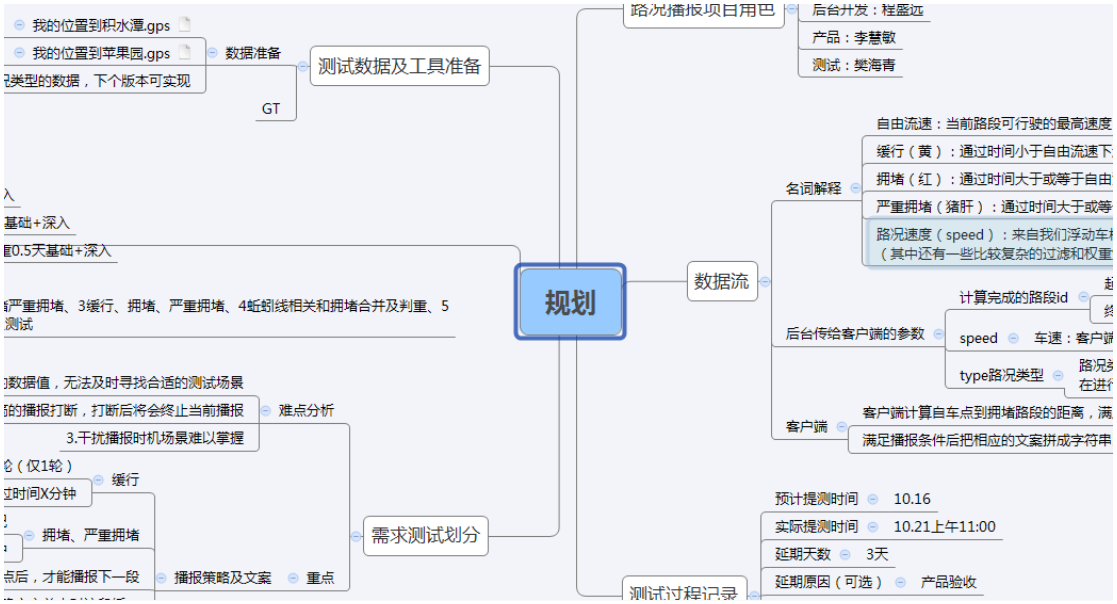


图 5-1

5.1.2、思维碰撞

上面说到了思路补充，其实还有一个很重要的环节是测试设计后的用例评审，大家坐在一起共同评审已完成用例。多人的交流得到的收获远大于主次负责人的补充，因此可以称为之为思维碰撞。

测试、产品、开发一起完成这个脑暴过程。测试人员可以起个主导作用，引导产品和开发往一些场景上思考，让熟悉代码的开发尽快的想起逻辑实现的缺陷，让产品尽快联想到需求设计缺陷。

探索式测试在这里面起到的是利用系统性的测试思维进行交流碰撞，引导更多的场景设想。可以用于用例补充，或者测试说明的补充，还可以将规划图做的更完整。

5.2、增量测试

在增量测试中，我们仍保留部分用例测试，加上探索式专项测试。也就是在纯脚本测试和纯探索式测试之间寻找一个点，如图 5-2 所示，取得最佳的效果。

图 5-2

5.2.1、基础测试

用例测试也被称之为基础测试，主要用于检查需求是否都按照预期实现了。这部分工作和传统的脚本测试无异，不加赘述。属于逻辑验证部分，这部分后期可以考虑做成自动化监控的形式。

5.2.2、探索测试

这里的探索测试其实就是专项探索式测试。在确认了基础需求验证通过后，结合风险点，展开探索式测试。可参考上一篇《探索式测试基础系列——蜜月旅行》。

本章以 QQ 浏览器（iPhone）的一次测试为例，开发重构了工具栏相关的代码，并罗列了可能涉及的影响模块。测试人员根据影响点采用二维表的组织方式，同时罗列出测试说明的问题，可以在各个交叉点中进行探索式测试实践，并回答测试说明中的关注点或者疑问点。

附：二维表（规划图）如下：

功能/基础特性	起始页前进后退	网页前进后退	每日头条：文章和图片阅览	微云收藏	yiya语音	我的视频：大家都在看	小说
全屏切换	pass	pass	pass	fail	fail	pass	pass
第三方调起	pass	pass	pass	NA	NA	pass	fail
换肤	pass	pass	pass	pass	pass	pass	pass
……	……	……	……	……	……	……	……

附 1：本次测试说明：

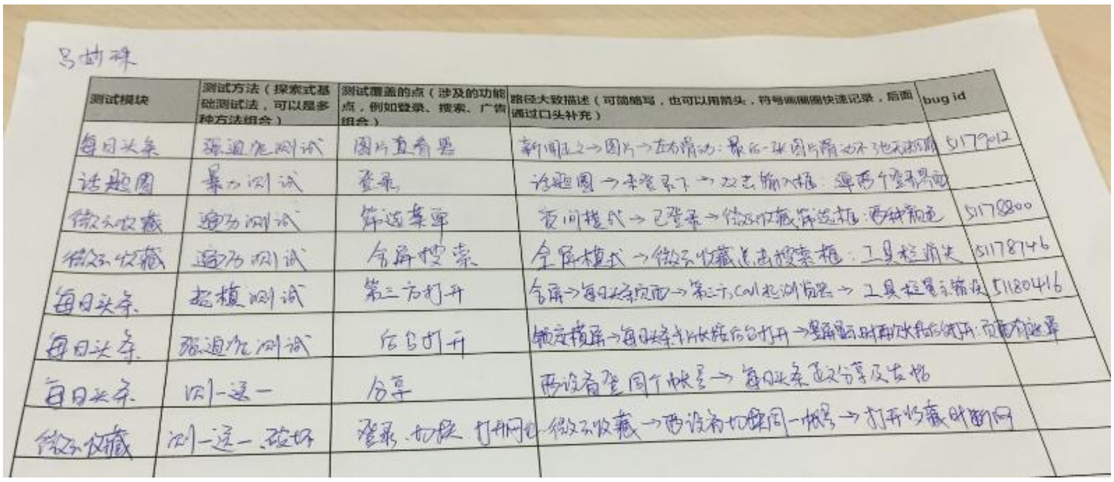
地址栏工具栏是否显示正常（前进后退状态，布局，动画）

地址栏，工具栏背景是否正确，是否可正常交互

前进后退滑动手势是否正确

业务场景切换的时候，过度动画是否正常

附 2：测试过程记录



测试模块	测试方法（探索式基础测试法，可以是多种方法组合）	测试覆盖的点（涉及的功能点，例如登录、搜索、广告组合）	路径大致描述（可简略写，也可以用箭头，符号画圈快速记录，后面通过口头补充）	bug id
每日头条	强迫症测试	图片查看器	新闻正文→图片→左右滑动：最后一张图片滑动不了也无提示	5179012
话题圈	暴力测试	登录	话题圈→未登录下→发表输入框：弹两个登录界面	
微云收藏	遍历测试	筛选菜单	夜间模式→已登录→微云收藏筛选框：两种颜色	5178800
微云收藏	遍历测试	全屏搜索	全屏模式→微云收藏点击搜索框：工具栏消失	51178746
每日头条	[illegible]测试	第三方打开	全屏→每日头条页面→第三方call起浏览器→工具栏[illegible]	51180416
每日头条	强迫症测试	后台打开	锁定横屏→每日头条长按后台打开→[illegible]	
每日头条	问一送一	分享	两设备登同个帐号→每日头条[illegible]分享及发帖	
微云收藏	问一送一、破坏	登录、切换、打开网	微云收藏→两设备切换同一帐号→打开收藏时断网	

附 3：测试报告

测试结果：不通过

发现问题：

1、横屏开启无痕模式，再竖屏，底部菜单栏没有无痕图案

2、yiya 语音网页搜索，全屏模式下搜索框与系统时间，电量栏重叠

3、网页搜索框，点击键盘上的语音输入无反应

……

六、生活协奏曲

前文讲过，探索式测试能为平常的生活带来浪漫因子，在浪漫一段时间后，新奇感消失，但效果仍在，探索式测试与日常测试真正融为一体，深刻作用于产品质量保证，共同演奏出协奏曲。接着上章，我们来讲下集成测试和上线前测试的两个环节中的探索式测试。

6.1、集成测试

集成测试阶段，各项功能（FT）都合入，且经过了测试，质量趋于稳定。也正是因

为这种合入，可能导致新旧功能之间产生不可知的影响。因此集成测试应该是一次完整的质量体检。我们的集成测试分成三个部分：指南测试、专项测试、系统探索。

6.1.1、指南测试

在探索式测试基础方法中有一种方法叫做指南针测试法，就是根据需求来做测试。我们把验证需求实现的用例称之为一级基础用例。因此指南测试其实也是用例测试，不过这个用例只是基础用例，覆盖了基础需求，只包含正常逻辑的用例。

举例来说 QQ 浏览器（iPhone）各个模块完整用例共计 3700 多条，包含了需求验证类型不含覆盖安装的基础用例（1 级用例），也包含了其他的用例（2 级用例）例如模块之间复杂交互和极限情况的用例、覆盖安装用例等。如图 6-1 所示。

优先级	测试点	条件	步骤	预期结果
1	显示	无历史记录	进入历史页面，查看历史记录	页面显示：没有历史记录，右下角“清空”按钮被置灰（没有清空按钮）
1		有历史记录	进入历史页面，查看历史记录	页面显示：历史记录，每条记录左侧显示对应的网址图标，右侧显示标题名，下方显示对应网址。页面右下角“清空”按钮为可用状态
1			点击任意历史记录	正确跳转至相应网页，网页在当前窗口打开，不新建窗口
2		历史记录有未拉取下来的网页记录的图标	拉取历史时网络不好，未拉取到网页记录的图标	显示默认的图标
2		历史记录有标题名过长的记录	查看的标题名过长的记录	过长的标题名在末尾截断，显示“...”
2		历史记录有URL过长的记录	查看的URL过长的记录	过长的URL在末尾截断，显示“...”
1	去重	已访问多个网址，并记录历史	再次访问当天访问过的网址	同一天内相同的历史记录，只保留最后一条记录
1	按日期显示	有今天的历史记录	进入历史页面，查看当天历史记录	显示为：今天+日期，下方显示显示对应日期的所有历史记录
2		有昨天的历史记录	进入历史页面，查看当天历史记录	显示为：昨天+日期，下方显示显示对应日期的所有历史记录
2		有3天前的历史记录	进入历史页面，查看当天历史记录	显示为：三天前的日期，下方显示对应日期的所有历史记录
2		有4天前的历史记录	进入历史页面，查看当天历史记录	显示为：四天前的日期，下方显示对应日期的所有历史记录
2		有5天前的历史记录	进入历史页面，查看当天历史记录	显示为：五天前的日期，下方显示对应日期的所有历史记录

图 6-1

这个用例筛选可以从两个时机入手，第一个时机是在设计用例的时候，直接按照需求标识出 1 级用例和 2 级用例。如果一开始没有做这样的用例分级，可以再集成前测试人员先按照需求进行分级，再约上不同的开发负责人逐一进行评审，确保基础需求的验证用例没有遗漏。

在 QQ 浏览器（iPhone）实际测试中，700 条用例，5 个测试人力，大约需要 1 天的时间进行。

6.1.2、专项测试

把 2 级用例中涉及覆盖安装的用例抽离出来，作为专项测试内容。如下图 6-2 所示：

验证点	结果
书签（本地书签、网络书签、PC书签、书签文件夹）	pass
首页书签	pass
历史记录、最常访问（包括搜索历史记录）	pass
个人中心身份态（保持、注销、切换）关注微信帐号	pass
微云收藏	pass
微云文件	pass
下载（状态、大小，个数）	pass
文件（查看，删除等）	pass
搜索引擎（升级后保持、切换）	pass
设置（字体大小、皮肤、消息管理、无痕浏览、夜间模式等抽测）	pass
分享	pass
省流量	pass
导航更新	pass
现场恢复	pass
cookies保存	pass
我的书架-收藏的书、设置、读书进度	pass
我的视频--最近观看、缓存任务、我的收藏	pass
热门视频、QQ空间、小说等的更新	pass

图 6-2

单独列出这项测试是因为移动 APP 的覆盖安装比较耗时，如果在指南测试中进行，将会不断出现等待升级的时间，我们将所有涉及覆盖安装的用例集中到一个时间段进行，通过一次升级就可以检查多个数据在新旧版本上的完整性和正确性。涉及到的探索式测试策略包括：上一版本测试法、快递测试法。

另外还有一个机型系统的适配问题，移动端的系统差异往往会影响其上的 APP 功能。实际集成测试每个测试人员负责的机型系统不同，因此我们还需要对一些核心功能进行全量的系统覆盖。也把这部分单独抽离出来作为专项测试。下表 6-1 所示。涉及到的探索式测试策略包括：遍历测试法、超模测试法。

表 6-1

加强对入口级功能的重视，在集成阶段要做到对此类功能点的全系统覆盖		
	功能点	备注

1	第三方调用打开	重点考察调起逻辑是否通畅，页面显示是否正常，包括横屏、皮肤、放大模式等因素下的菜单栏、通知栏显示
2	与音乐兼容	QQ 音乐、酷狗、酷我等播放器在后台播放音乐时，打开浏览器后音乐照常播放
3	通知栏 push	点击 push 进入浏览器（包括冷启动和热启动）页面显示是否异常，包括通知栏、菜单栏等
*	******	*******

专项测试阶段在 QQ 浏览器（iPhone）上的耗时为 5 人*0.5 天。

6.1.3、系统探索

这个阶段在基础用例+覆盖安装用例之后，是一次大规模的探索式测试。

首先将浏览器基础特性作为一个维度，将各个 FT 作为另外一个维度，形成如下图所示的二维表。这个表的目的是将探索式测试的自由度限制在一个框架内，不至于偏离主题，在横纵交叉点中测试人员可以充分发挥自己的自由度去做“边测试边设计”的工作。

	浏览器features业务交叉关系								
	业务模块 功能模块	FT模块							
		每日头条	小说书架	轻应用	文件管理（包括微云、微收藏）	我的视频（包含下载播放）	微云收藏	皮肤	导航卡片	书签快链
基础能力	QQ帐号登录	pass	fail	NA	pass	pass	pass	pass	pass	pass
	微信登录	pass	pass	NA	na	pass	na	pass	pass	pass
	分享（跨屏、啾啾、复制网址、生成二维码等	pass	pass	NA	na	pass	pass	pass	pass	pass
	跨屏穿越	na	pass	NA	pass	pass	pass	pass	pass	pass
	地址栏	fail	fail	NA	na	pass	pass	pass	pass	pass
	全屏	pass	pass	NA	pass	pass	na	pass	pass	pass
	底栏菜单	pass	pass	NA	pass	pass	pass	pass	pass	pass
	长按菜单	pass	NA	NA	pass	pass	pass	pass	pass	pass
	夜间模式	pass	pass	NA	pass	pass	pass	pass	pass	pass
	旋屏	pass	pass	NA	pass	pass	pass	pass	pass	pass

图 6-3

上图 6-3 所示是二维表，还可以进一步演绎为多维表，将每个 FT 与整个浏览器乃至整个操作平台的特性关联起来，形成多维规划图。整个操作过程建议做测试记录和交流总结。

在 QQ 浏览器（iPhone）上的这个阶段耗时大约是 5 人*1.5 天。

6.2、上线测试

上线测试一般时间相对有限。我们的测试就分为检查点测试和风险点的测试。

6.2.1、检查点测试

检查点非常类似于集成测试中的指南测试，不过这里关注的是基础特性是否受到影响。如下图 6-4 所示是 QQ 浏览器（iPhone）在上线前的检查点，基本涵盖基础功能验证。

序号	来源	检查点
1	基础要求	系统push（小说更新push、跨屏push）
2	基础要求	升级提示（3种方式：系统通知栏、提示升级、检查升级）
3	基础要求	起始页配置拉取正确
4	基础要求	手Q身份调用
5	基础要求	能够覆盖安装(采用第二个覆盖安装的sheet，不用每个人都验证一轮，四个人每人验证一部分合起来就行）
6	基础要求	检查后台数据（QUA、渠道号、相关PV数）
7	基础要求	问题反馈的链接是否正确
8	基础要求	视频下载可离线观看
9	基础要求	帮助关于正确
9	基础要求	帐号中心登录后，QQ空间等自有业务会有快速登录的提示
9	基础要求	手Q微信第三方调起（记录login_type)
9	基础要求	icloud
9	核心能力	mttf达标
9	核心能力	内存和速度是否达到要求
9	用户反馈痛点	与QQ音乐兼容性
9	用户反馈痛点	QQ农场、牧场等应用的一健偷菜 一健播种等操作
9	用户反馈痛点	微云收藏、视频离线也能观看
9	用户反馈痛点	小说、精阅可读

图 6-4

6.2.2、风险点测试

每次提交上线，都有一些修改的代码，这些修改的代码涉及的影响点，也是上线前测试阶段探索式测试的着力点。

根据 svn 日志中查找修改点或者开发 PM 罗列出风险，或者像回归测试中的用到的精准测试那样输出测试点，以这些为测试章程进行测试，也即风险点测试。

在这两个阶段的探索式测试落地实践已经算是比较成熟和完备的阶段，如果运用得到，将会事半功倍，改革旧有模式，非常值得尝试。

七、生活进阶曲

在探索式测试落地实践中奏出了协奏曲后进入到高级阶段，如何在问题定位和经验积累中发挥作用，也可以理解为在生活达到非常和谐后，如何孕育一个后代并为其提供良好的环境，因此本章的名字叫做生活进阶曲，表明在本章内容结束后生活将发生了质的改变，有了良好的传承。

7.1、反馈跟踪

前面讲的都是开发迭代过程，在实际中我们还有很重要的一个环节就是上线后的用户反馈跟踪。通过各种渠道，我们可以收集到各种用户反馈，能否将用户反馈复现出来直接影响到问题的定位和解决，另外一方面，随着用户反馈问题的复现，我们可以回顾反思漏测问题。

7.1.1、路径复现

拿到一条用户反馈，我们就要尝试去复现。举例来说。灰度后，收到一条用户反馈“视频横屏播放后，无法竖屏。”

寻找复现路径，根据视频规划图，可以确认影响视频播放的影响点：网络、横竖屏、锁屏、弹幕、小窗口等等，如下图 7-1 所示。结合这些测试点，根据不同的探索式测试基础方法进行尝试。

图 7-1

最终发现复现路径为：任意视频源，小窗口横屏观看视频，点击暂停，等待手机自动黑屏，再次打开手机进入播放界面，竖屏旋转手机，视频播放界面无法横屏。

采用的测试策略包含：长路径测试（多种操作连续进行）、清晨测试法（屏幕解锁重新进入视频）。

7.1.2、漏测分析

在复现了用户反馈的问题，我们还可以回溯到问题产生的根源。主要是几个步骤：

【时机】

(1) 引入 bug 的代码时机（开发）

(2) 发现 bug 的时机（测试）

两个时间差可以用来评估漏测的时间成本。

【缺陷类型】

(1) 必现程度：必然出现，大概率出现，很难出现。

(2) 机型系统：全系统都出现，新系统出现（例如 iOS9），少数低端系统出现（例如 iOS6），常用系统出现（例如 iOS7、iOS8）

(3) 严重程度：闪退，重启或者重试可恢复的 bug，一直不消失的 bug

(4) 修复成本：通过后台发布更新来修复，发 patch 版本来修复

上述不同维度结合起来可以评估缺陷的类型，定位漏测带来的损失程度。

【回顾反思】

如下图 7-2 所示，逐个环节提问漏测的原因。

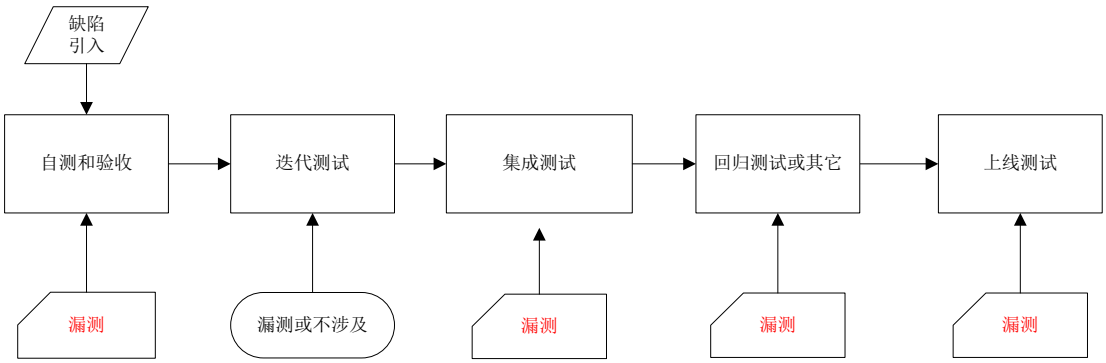


图 7-2

(1) 5W：通过层层递进的提问和回答进行分析；通过并发时多维度提问和回答进行分析。

(2) 扩展思考：缺陷在每个阶段是如何被漏过，今后应该怎样关注。

7.2、经验库积累

经验传承问题不光是探索测试的专利,但基于探索式测试的研究,我们仍然重点关注了这个环节。将前人的测试经验已库的方式存储起来，以便后人参考查阅。并且从个性化和通用角度分成了一级和二级。

7.2.1、一级经验库

一级经验库主要是基于各项目自身，总结出本项目核心功能及重点功能的测试方法，这里面包含项目个性化的功能，比如地图产品，导航是个性化又核心得功能，所以对于导航功能测试方法提取，适用于各个地图产品，但并不适用于其他类型产品。如下图 7-3 所示是同步助手的以及经验库。

测试模块	测试思路	测试场景	发现问题	Bug分析
照片备份				
登录态	指南测试法	有登录态进入照片备份		
	反叛测试法	无登录态进入照片备份触发登录、登录手机帐号	ID： 50969044 【手Q会员合作回归一】非会员： “照片备份” 触发登录，输入手机号仍能顺利进入照片备份功能	功能实现遗漏
本地相册	指南测试法	本机照片，扫描后成生故事相册（按云控分类命名）	ID： 50639196【6.1 迭代一回归二】照片管理- 照片扫描：手机中只有两张刚拍的照片，扫描卡住	编码实现错误
	指南测试法	一键收藏相册、收藏相册部分照片生成稍后处理相		
	强迫症测试法	稍后处理相册多次操作选择照片收藏与删除		
	极限测试法	手机没有照片、只有截图、所有照片已上传到云端		
	极限测试法	手机有大量照片，生成10个相册	ID： 50651696 【6.1 迭代一回归三】照片管理：相册有1000+张照片时，本地加载时间持续三四分钟才显示照片。建议优化 ID： 50656499 【6.1 迭代一回归三】照片管理- 相册分	性能问题 编码实现错误
	破坏测试法	禁止读写系统照片，进入照片备份		
	上一版本测试	上一版本禁止读写系统照片，覆盖安装进入照片备份	ID： 50749780 【6.1 上线前测试】覆盖安装：3.7.1的包已登录和对照片权限未选择 “允许” 或 “禁止” ，安装	编码实现错误

图 7-3

7.2.2、二级经验库

二级经验库主要是基于整个互联网产品，总结出互联网产品上面通用功能的测试方法，这个适用于多个互联网产品，需要从互联网产品结构进行分析，提取难度更大些，比如几乎所有产品都包含的登录功能，同步功能，列表功能，联网读取功能等。如下图 7-4 所示是登录功能的经验库部分截图。

测试模块	测试思路	测试场景	发现问题	Bug分析
登录				
登录状态	指南测试法	显示登录入口（未登录/登录态过		
	指南测试法	已登录（显示登录信息，可注销）	ID： 50382345【3.7 集成测试】更多	编码实现错误
手Q快速登录	指南测试法	有安装手Q（有登录/无登录）		
	懒汉测试法	无安装手Q		
	旧版本测试法	安装旧版手Q		
	测一送一测试法	帐号已修改密码，点击一键登录		
	取消测试法	手Q快速登录过程中断（返回/Home键）	ID:50351741 【wtlogin-回归一测试】在手机QQ登录页面点击 “返回” ，提示错误码0	界面逻辑问题
	破坏测试法	手Q快速登录过程中断(中断网络/卸		
	破坏测试法	手Q快速登录后kill应用	ID:50342663【wtlogin-集成测试】手机QQ一键登录，kill掉同步助手，再次进入同步助手登录态丢失	登录态仍没有保存到userInfo中于是 在收到成功通知时在saveAccount函数中再
	强迫症测试法	反复操作一健登录与注销	ID:50351067 【wtlogin-回归一测试】手机QQ一键登录，同步完成后注销	编码问题，一些前置变量没设置好

图 7-4

至此探索式测试体系基本讲述完毕，缘来就是你，期待读者朋友能够与探索式测试在工作中相亲相爱，相得益彰！

应用宝基于 Robotium 自动化测试

◆作者：陈航特

一、背景目的

应用宝项目组采用 FT(Feature Team)模式，整个项目组分为多个 FT，而每个 FT 又同时有多个需求分支在并行运作着，几乎每天都有多新特性合入主干，项目节奏快、变更频繁，且又希望能够短周期内快速地对外发布新版本，做到快速交付、持续交付。

为了支撑项目组的这种研发模式，测试侧需要在 FT 分支上及主干上做大量的测试，而其中在 FT 分支的 rebase 测试、合流后验证、主干灰度测试等等阶段还包括大量的重复性测试，因此有必要在这些环节加入自动化测试，以持续验证新特性未破坏原有系统。

二、框架选择

如 0 所示，对比了目前业界常用的几个可用于 Android 端的自动化测试框架:

表 1.Android 自动化测试框架对比

框架	Robotium	Espresso	UIAutomator	Appium
支持的 API 版本	All	8，10，15 >=	16 >	All
成熟度/活跃度	成熟/活跃	成熟/Google	较成熟/活跃	较成熟/活跃
支持的语言	Java	Java	Java	Almost Any
WebView/X5 WebView	支持/支持	不支持	不支持	支持/支持
UIAutomation	API >=18, 支持结合 UIAutomation	API >=18, 支持结合 UIAutomation	不支持	未知
用例执方式/adb 依赖性	adb shell am	同 Robotium	adb shell uiautomator runtest	Client Server 发送命令模式

主要优点	1.通过 id 识别控件，手机适配好 2.基于 Instrumentation，执行速度快、稳定性好 3.可使用 Android 及被测工程的 API	同 Robotium	跨应用支持好	1.不需要对被测应用进行修改 2.脚本支持多种语言进行编写 3.支持跨平台，可用于 IOS、Android
主要缺点	1.测试 apk 签名需要与被测 apk 一致 2.跨应用能力弱	同 Robotium	1.只支 API>16 2.需要被测控件有 android:hint 等属性	1.API4.2 以下只支持 Selendroid 方式 2.一台 MAC 机只能运行一个 Instrument 实例
执行速度	4 星	4 星	4 星	3.5 星
稳定性	4 星	4 星	3.5 星	3.5 星

通过对比可以发现不同的测试框架各有优缺点，且基本都不能独自满足所有的测试场景，正所谓菜刀、牛刀、剪刀、指甲刀，刀刀皆有自己的用途与适用场景，我们需要根据项目情况及实际的使用场景来选择最适用的工具。

Robotium 基于原生 Android Instrumentation 扩展而来，因此基于 Robotium 的测试既可以使用 Robotium 本身的 API，还可以使用 Android 原生的丰富 API，可扩展性更强，且基于 Robotium 的测试在执行速度、稳定性上有一定优势，而应用宝在手机端只有 Android 版本，也没有跨平台的需求，综合考虑，因此选择了 Robotium 框架。

三、环境搭建

3.1、基础环境搭建

测试工程使用了 Robotium，采用了的是 Android Junit 工程，因此需要搭建基础的 Android 开发环境，包含 JDK、Android SDK、Eclipse + ADT 插件 + SVN 插件等等开发工具，具体可搜“Android 开发环境搭建”搭建基础环境。

3.2、导入测试工程

1、使用 Eclipse 导入项目

2、配置 Build Path

3、配置 keystore

在实际项目中，如果是自家的项目，显然是不希望对被测 App 进行重签名的，有如下原因:

1) 每日进行测试的包众多，一一进行重签名影响效率

2) 如微信、应用宝等应用做了签名防护措施，重签名后将导致应用部分功能不可用甚至直接无法启动

测试工程需要与被测工程签名一致，因此测试工程需要将 keystore 配置成应用宝的签名。Window——Preferences——Android——Build，如 0 所示，点击 Browser，选择应用宝的 debug.keystore 签名，配置完成后，用 Eclipse 调试时，测试工程打出的 apk 即是应用宝的签名了，可以测试应用宝对外发布的任何包。

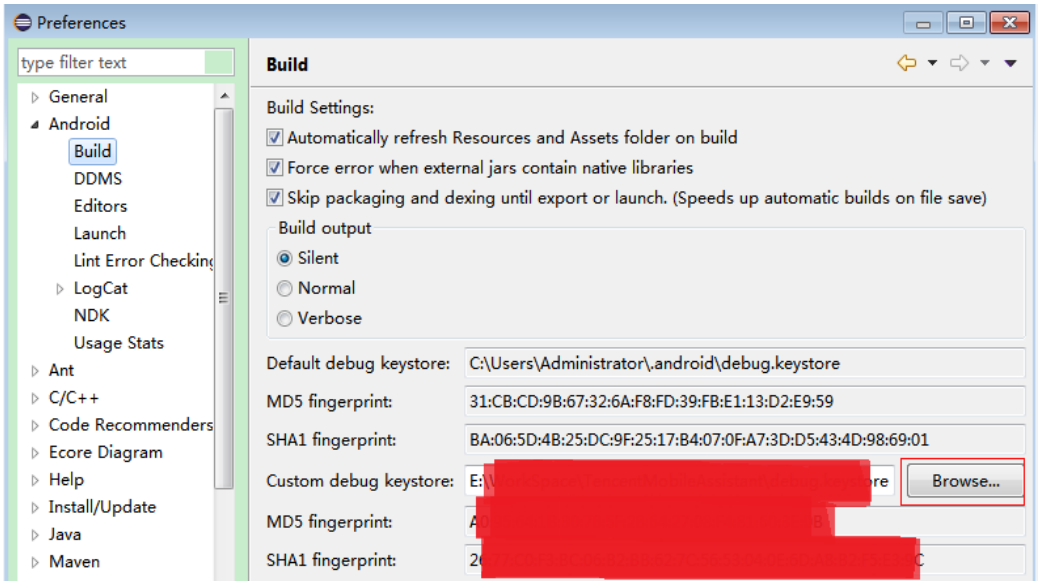


图 1 Eclipse 中配置自定义签名

4、配置编码

新导入工程后，工程可能有许多红点，此时工程任意有注释的 java 文件，如果注释为乱码则是因为编码不一致导致。此时需要将工程编码设置为 utf-8。也可右键选择测试工程，仅设置该工程为 UTF-8 编码

3.3、Eclipse 设置

工欲善其事，必先利其器，测试工程使用 Eclipse 作为 IDE，而为了编写代码可以更高效，有必要进行一些提高效率的设置。

1、配置输入联想

为了提高测试用例的代码编写效率，很有必要配置输入联想，在 Eclipse 中 Preferences——Java——Editor——Content Assist 配置输入联想

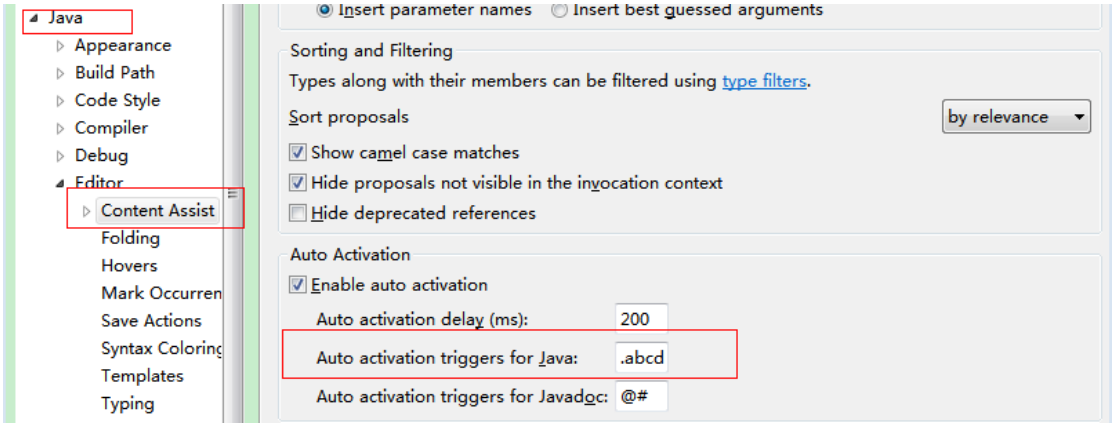


图 2 Eclipse 中配置代码自动提示

其中 Auto activation triggers for java 中默认只有.符号，即输入.时才会有代码联想出来，为了充分利用代码联想功能，需要在该输入框中把 abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ 这 26 个字母及.号输进去，这样，当键入.号或 26 个字母时，就会有自动提示，提高代码输入效率。

2、配置 Eclipse 的 JVM 参数

Eclipse 的 JVM 默认设置参数较小，因此可能造成各种卡慢现像，而我们的开发机配置一般较高，可以通过调整 JVM 参数充分利用机器资源提高 Eclipse 运行的流畅度，修改 Eclipse 安装目录下的 eclipse.ini 文件即可，具体可参考如下配置（修改-vmargs 参数后面的配置）:

```
-vmargs
-Dosgi.requiredJavaVersion=1.6
-Xms2048M
-Xmx2048M
-Xmn656M
-XX:PermSize=512M
-XX:MaxPermSize=1024M
-XX:+UseParallelGC
-XX:CMSInitiatingOccupancyFraction=85
```

-Xverify:none

-Xnoclassgc

-XX:+CMSClassUnloadingEnabled

-XX:+CMSPermGenSweepingEnabled

-XX:+DisableExplicitGC

3、关联源码

a) 关联内引用 jar 包的源码

导入测试工程后，libs 下的 Robotium 和 Uiautomator 两个 jar 使用了 properties 配置，默认就已关联上了 sources 目录下的源码，如 0 所示:

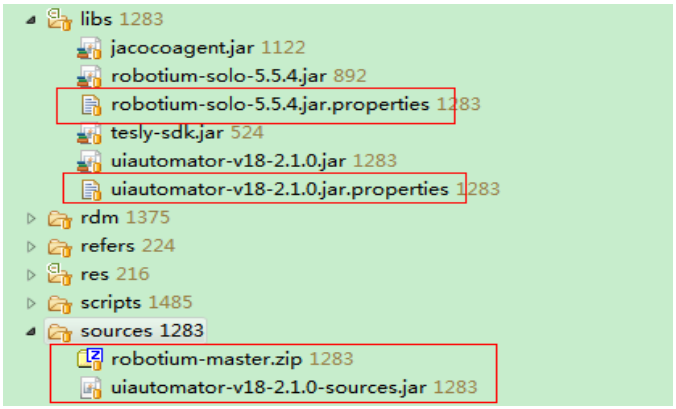


图 3 Eclipse 中配置关联 private jar

b) 关联外引用 jar 包的源码

关联外引用 jar 包的源码，这里主要关联 Android SDK 中的源码，右键 android.jar，进入 Java Source Attachment 选项，关联 sdk 中的源码，如 0 所示:

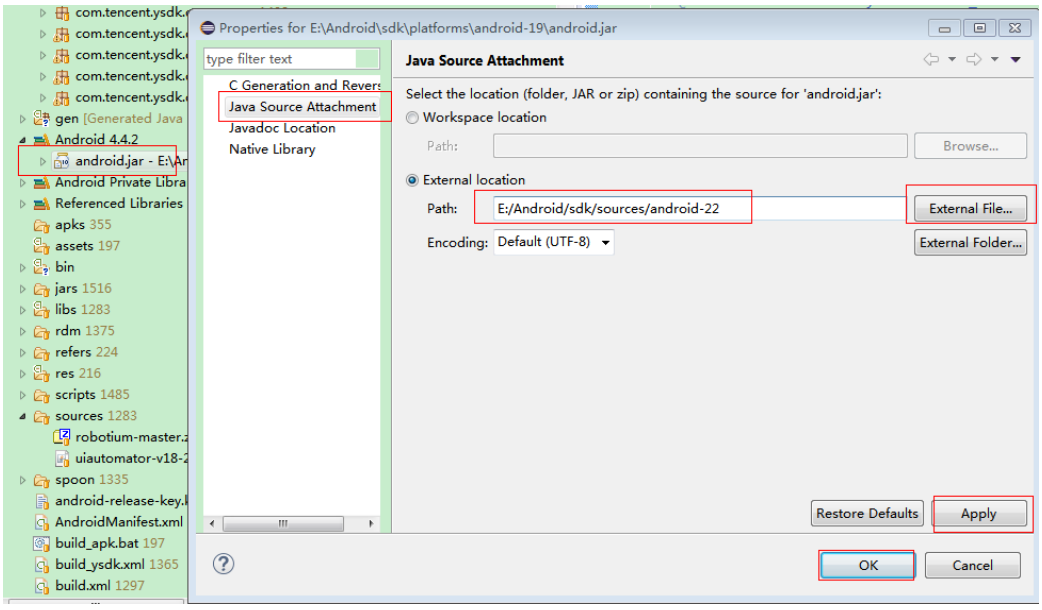


图 4 Eclipse 中配置关联外引用的 jar

至此，不论进入的是 Android SDK 还是 Robotium 中的 class 类，均可以查看到其源码实现。

四、Robotium

4.1、Robotium 介绍

Robotium 对外主要提供以下几个类:

- By: //Web 元素的选择器
- Condition: //接口类，用于等待
- RobotiumUtils: //工具类
- Solo: //对外提供各种 API
- Solo.Config: //Solo 配置类
- SystemUtils: //系统级工具类
- TimeOut: //Solo 配置类
- WebElement: //Web 元素的抽象类

其中 Solo 类是主要对外提供各种 API 的类，Solo 类采用中介者模式，持有 com.robotium.solo 包下的其它类的的实例对象，当我们调用 Solo 类中的 API 时，实则大

多数是转而调用 com.robotium.solo 包下其它类的方法。com.robotium.solo 包下主要有以下类:

- Getter: //提供控件获取相关 API
- ActivityUtils: //提供 Activity 相关 API
- Asserter: //提供断言相关的 API
- Clicker: //提供模拟点击相关的 API
- ScreenshotTaker: //提供截图相关的 API
- Scroller: //提供滚动相关的 API
- Searcher: //提供控件搜索相关的 API
- ViewFetcher: //提供控件过滤相关的 API
- Waiter: //提供控件等待相关的 API
- WebUtils: //提供 Web 支持相关的 API

Robotium 为了简化测试用例的编写，将以上的这些类都置为了 protected，对外只提供 Solo 类，因此，在编写测试用例时，主要实例化 Solo 类即可

Robotium 为一款支持黑盒测试也支持白盒测试的自动化测试框架，简单易用，提供了获取控件、发送点击事件、断言等等 API。

主要 API 如 0 所示:

表 2 Robotium 主要 API

返回值	方法及说明
View	getView(String id) 根据 id 获取控件
ArrayList<View>	getCurrentViews() 获取当前界面或弹框中所有的控件
ArrayList<T>	getCurrentViews(Class<T> classToFilterBy, View parent) 获取父控件 parent 下所有控件类型为 classToFilterBy 的控件
void	clickOnView(View view) / clickLongOnView(View view)

	点击指定的 view 控件 / 长按指定的 view 控件
void	clickOnScreen(float x, float y) / clickLongOnScreen(float x, float y) 根据坐标 x，y 点击屏幕 / 根据坐标 x，y 长按屏幕
void	enterText(EditText editText, String text) 在指定的 editText 中输入文本 text
void	typeText(EditText editText, String text) 在指定的 editText 中键入文本 text
void	drag(float fromX, float toX, float fromY, float toY, int stepCount) 从起始 x,y 坐标滑至终点 x,y 坐标；通过 stepCount 参数指定滑动时的步长
void	scrollToTop() / scrollToBottom() / scrollUp() / scrollDown() 滚动至顶部 / 滚动至底部 /向上滚动屏幕 / 向下滚动屏幕
boolean	waitForView(int id) / waitForText(String text) 等待指定控件出现 /等待指定文本出现
boolean	waitForActivity(String name) 等待指定的 Activity 出现
void	takeScreenshot(String name) 截图，图片名称为指定的 name 参数，图片默认路径/sdcard/Robotium-Screenshots/
void	finishOpenedActivities() 关闭当前已打开的所有的 Activity
void	goBack() / goBackToActivity(String name) 点击返回键 / 不断地点击返回键直至返回至指定的 Activity
ArrayList<WebElement>	getCurrentWebElements(By by) 通过 By 根据指定的元素属性获取当前 WebView 的所有 WebElement 元素
void	clickOnWebElement(By by) 通过 By 根据指定的元素属性点击 WebElement
void	clickOnWebElement(WebElement webElement) 点击指定的 WebElement
void	assertCurrentActivity(String message, String name) 断言当前界面是否为 name 参数指定的 Activity，若不是将抛出一个带有 message 提示的 Throwable 异常

通过如上 API 文档也可以发现，Robotium 框架提供的 API 是有限的，且也只是使用了一小部分 Android 的 API 封装而成。使用 Android、Java 丰富的类库我们可以开发出微信、手 Q、应用宝等等众多 App，同样地，我们也可以使用这些丰富的类库去扩展测试框架。

因此，选择 Robotium 测试框架，不只是选择的一个测试框架，而是选择的一种测试模式，即基于 Android、基于 Junit 的测试模式。

4.2、Native 控件获取与处理

1、uiautomatorviewer

可以使用%ANDROID_HOME%\tools 目录下的 uiautomatorviewer.bat 工具直接获取当前界面的控件结构及其 id。

注：uiautomatorviewer 只有在 Android 较高版本（4.3 及以上）才能直接获取控件 id 值。

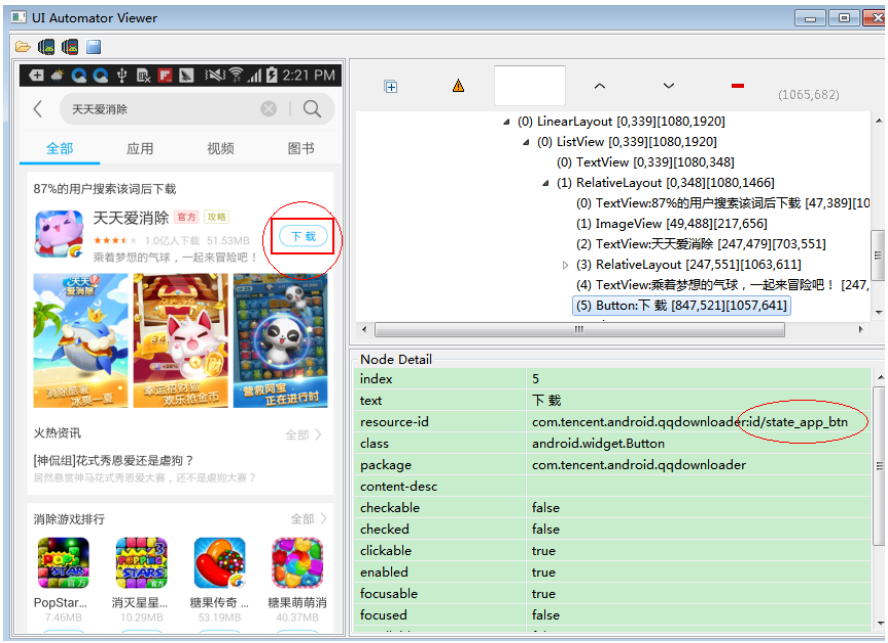


图 5 Uiautomatorviewer 查看控件

2、处理唯一 id 的控件

如果当前界面该控件 id 是唯一的，则处理起来很简单，如下：

```
Button loginBtn = (Button) solo.getView("loginBtn");
solo.clickOnView(loginBtn)
```

3、处理 id 相同的控件

在 Android 中，列表 ListView 采用的是 Adapter 形式，所以列表中的控件 id 都是相同的。

此时，需要先获取节点控件的父视图，通过父视图再查找相应的子视图。

4、控件过滤

测试过程中最常见的方式就是控件过滤

API: getCurrentViews(Class<T> classToFilterBy, View parent)

例如想获取某一个区域内的所有文本:

ArrayList<TextView> textViews = holo.getCurrentViews(TextView.class,parentView);

其中 parentView 为父视图，获取 parentView 下所有的 TextView

4.3、WebView 控件获取与处理

WebElement 元素获取可以通过 Chrome Mobile Emulation 模式获取。安装有较新版本 Chrome 浏览器，按 F12 即可进入 Chrome Mobile Emulation 模式。

输入 H5 页面链接，如：http://xxx.xxx.xxx/index.html

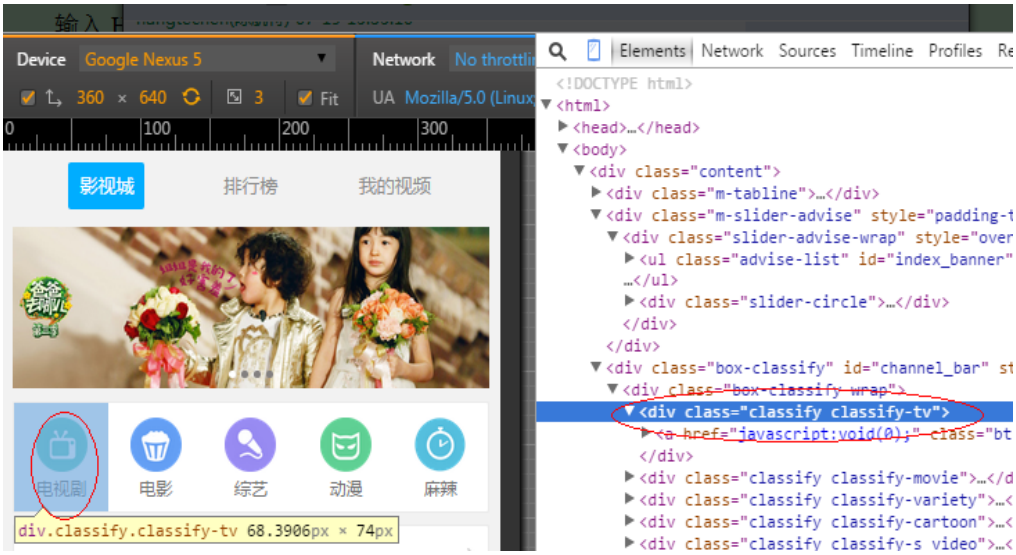


图 6 Chrome 按 F12

如 0 所示可以看到‘电视剧’拥有 class=“classify classify-tv”的唯一属性。

对于有些无法通过 PC 浏览器打开的 H5 页面，可以通过 Chrome DevTools 连接手机

端直接进行调试。首先，确保手机安装有可调试的 APP，即 APP 在 AndroidManifest.xml 中的设置为 android:debuggable="true"，例如应用宝打的 dev 模式的包即是可调试的包。

然后，在 Chrome 浏览器地址栏中输入“chrome://inspect/#devices”

打开应用宝，进入含有 WebView 的页面，例如进入娱乐 TAB，如 0 所示，可以看到出现了可以 inspect 的页面，点击 ‘inspect’ 按钮即可对该页面进行调试，如 0 所示，展示了该页面的元素。

图 7 Chrome 中使用 DevTools

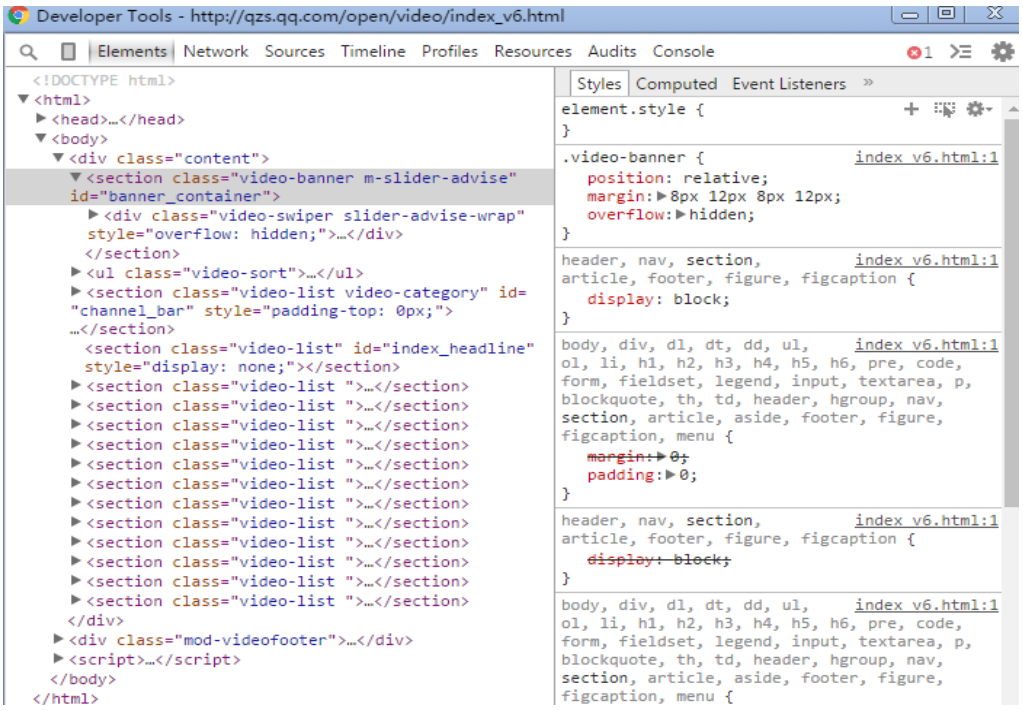


图 8 DevTools 点击 inspect 后展示页面元素

知道了元素的属性后，即可调用相应 API 获取 WebElement 对象，通过 By 方式获取：getCurrentTxWebElements(By by)；By 方式支持通过 classname、id、textcontent 等方式，如：

holo.getCurrentTxWebElements(By.className(className));

获取到了相应的 WebElement 对象后，即可对 Web 元素进行操作:

clickOnWebElement(By by)

clickOnWebElement(WebElement webElement)

waitForTxWebElement(By by,int timeout,boolean scroll)

4.4、断言

1、Assert 中的断言

使用 junit.framework.Assert 包中的断言：断言条件的 true 或 false、是否为空等等。如 0 所示:

From class junit.framework.Assert

static void	assertEquals(short expected, short actual) Asserts that two shorts are equal.
static void	assertEquals(String message, int expected, int actual) Asserts that two ints are equal.
static void	assertEquals(String message, short expected, short actual) Asserts that two shorts are equal.
static void	assertEquals(char expected, char actual) Asserts that two chars are equal.
static void	assertEquals(String message, String expected, String actual) Asserts that two Strings are equal.
static void	assertEquals(int expected, int actual) Asserts that two ints are equal.

图 9 Assert 中的断言

2、ViewAsserts 中的断言

使用 android.test.ViewAsserts 包中的断言：包括断言控件是否左对齐、右对齐、父视图是否包含某子视图等等。

Summary

Public Methods	
static void	assertBaselineAligned (View first, View second) Assert that two views are aligned on their baseline, that is that their baselines ar
static void	assertBottomAligned (View first, View second) Assert that two views are bottom aligned, that is that their bottom edges are on
static void	assertBottomAligned (View first, View second, int margin) Assert that two views are bottom aligned, that is that their bottom edges are on
static void	assertGroupContains (ViewGroup parent, View child) Assert that the specified group contains a specific child once and only once.
static void	assertGroupIntegrity (ViewGroup parent) Assert the specified group's integrity.
static void	assertGroupNotContains (ViewGroup parent, View child) Assert that the specified group does not contain a specific child.
static void	assertHasScreenCoordinates (View origin, View view, int x, int y) Assert that a view has a particular x and y position on the visible screen.

图 10 ViewAsserts 中的断言

五、跨应用（结合 UiAutomator2.0）

2015 年 3 月 Android Developers 团队宣布了 UiAutomator 2.0 版本的发布，这个版本最重要的就是 UiAutomator 终于可以基于 Instrumentation 了，使用 Instrumentation test runner 即可运行 UiAutomator，反过来，也即在基于 Instrumentation 的 test 中也能使用 UiAutomator。因此测试工程可同时使用 Robotium 和 UiAutomator 进行更丰富地测试。

新版的 UiAutomator 随 Android Support Repository 发布，可通过 SDK Manager 下载，以 2.1.0 版本为例，位于如下所示的路径中:

%ANDROID_HOME%\extras\android\m2repository\com\android\support\test\uiautomator\uiautomator-v18\2.1.0

新的测试支持库基本都是基于 Android Studio 库，文件以 aar 结尾而非 jar 结尾，本小节为方便在 Eclipse 中介绍需要将 aar 转化成 jar。

如 0 所示，使用压缩工具打开 uiautomator-v18-2.1.0.aar 文件，里面的 classes.jar 文件即是可用于 Eclipse 的 UiAutomator jar 包。提取出该 classes.jar 文件并重命名为方便记忆的 jar 包文件，导入至使用了 Robotium 的测试工程即可。

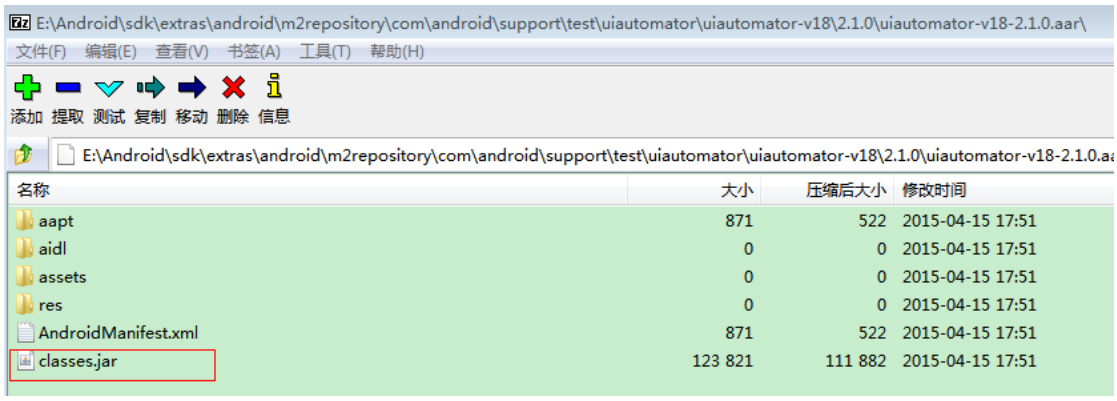
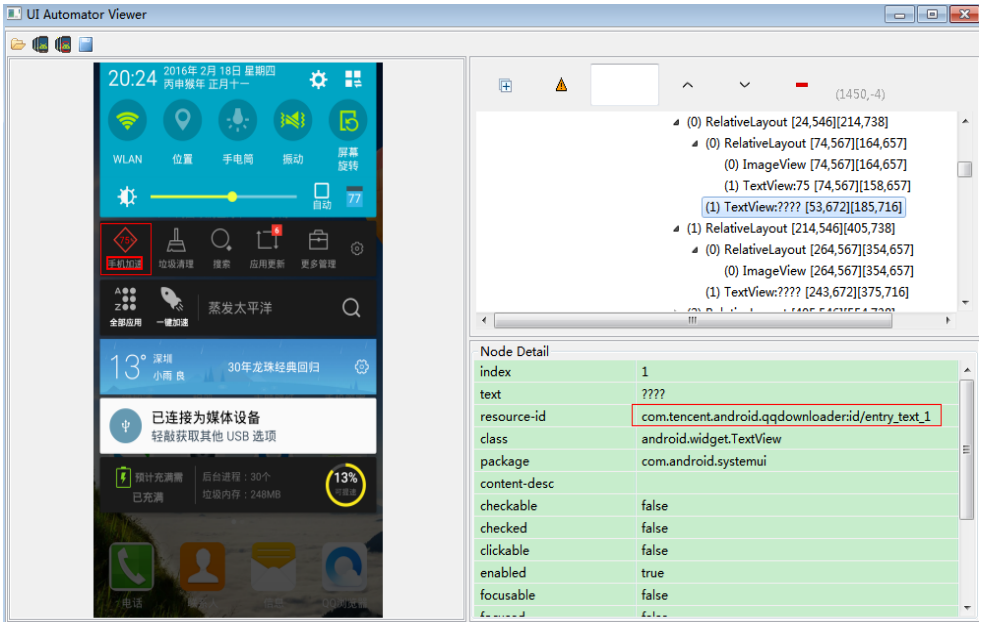


图 11 解压 aar 文件

如 0 所示，应用宝在通知栏中开启了快捷工具栏，测试此功能时需要开启通知栏，并点击工具栏中的按钮，这样的操作仅通过 Robotium 框架是无法完成的，此时就可以结合 UiAutomator 来实现。



0 应用宝快捷工具栏

UiAutomator 发布 2.0 版本后，可以通过传入 Instrumentation 对象获得 UiDevice 对象。

```
UiDevice.getInstance(instrumentation);
```

通过 UiDevice 对象可以完成点击 Home 键、打开通知栏，并通过 UiDevice 的 findObject 方法可以根据文本、资源 id 等等查找控件，并通过 UiObject 对象完成点击操作。

使用的 findObject 方法得到的为 UiObject 对象，此外也可以通过 By 的方式获取

UiAutomator 中的 UiObject2 对象，例如：

```
uiDevice.findObject(By.res("com.tencent.android.qqdownloader", "entry_text_1")).click();
```

UiAutomator2.0 还有许多更丰富更强大的功能，这里就不再一一介绍，总之，通过与 Instrumentation 结合可以方便地在测试工程中完成跨应用的操作，进行更丰富地测试。

六、测试工程

6.1、测试工程概览

使用 Robotium 进行自动化测试，测试工程为一个 Android Junit Test 工程，可以依赖被测工程，与可以选择独立存在。

关联被测工程源码的好处在于可以调用被测工程的代码，因此可以更容易地获取到被测应用内部的状态，例如拿到被测应用 ListView 内部填充的数据等等。而这样也会带来一些弊端:

1、测试工程的自动化编译打包也需要关联被测工程，脚本复杂度及维护成本增加

2、如果采用 R.id.xxx 方式获取控件的话，被测工程增加、删除布局文件都可能影响到测试工程的编译结果

3、如果被测应用进行了代码混淆，引用被测工程的代码复杂度将大大提高

鉴于此，应用宝采用的是脱离被测工程的方式，同一份测试 apk 可以同时测试多个版本的被测应用，另外，即使大家选择有源码的方式，也不建议使用 R.id.xxx 的方式获取控件。

测试工程需要在 AndroidManifest.xml 文件中注册 instrumentation 用于指定被测应用:

```
<instrumentation
    android:targetPackage="com.robotium.android.notepad"
    android:name="android.test.InstrumentationTestRunner" />
```

在同一个测试工程中我们可以只注册一个 instrumentation，也可以同时注册多个，例如当被测应用有多个，而测试工程又不想分别建立多个时，则可以使用注册多个的方法。首先，先编写一个继承自 android.test.InstrumentationTestRunner 的自定义

InstrumentationTestRunner，然后同样地在 AndroidManifest.xml 中注册:

```
<instrumentation
    android:targetPackage="com.robotium.android.anothernotepad"
    android:name=".instrumentation.InstrumentationTestRunner" />
```

6.2、测试用例

6.2.1、测试用例生命周期

测试用例基于 Android Junit，每个用例遵循以下三个步骤:

1) 首先，执行 setUp()方法，用于初始化。

2) 然后，执行以 public 且方法名以 test 开头的用例方法。

3) 最后，执行 tearDown()方法，用于释放资源等

另外，由于许多用例都需要拥有同样的功能特点，例如需要能够进行出错重试与出错截图等等，因此，可以编写一个共有的测试基类，应用宝测试工程中所有的测试类均继承自 SingleLaunchActivityTestCase2 这个类。

6.2.2、测试用例编写

测试用例编写的质量直接关系到用例的稳定性、维护成本以及是否能发现有效问题等等，因此是自动化测试中的关键一环。

(1) 首先，是确定测试用例的来源。

当开始准备编写自动化测试用例时，需要确定测试用例的来源，即需要明确例如以下几个方面:

哪些功能是主要功能、哪些功能可以自动化

用例的优先级、作用的测试阶段

测试一个功能需要哪些验证点

不同的项目组需要思考的点可能不一样，但目的是一致的，需要明确测试用例的来源，而不是任意地开始编写用例。应用宝中采用 CheckList 的形式，通过与各业务线讨论评审的方式确定关键功能、是否自动化、用例优先级、测试验证点等等。

(2) 然后，应该合理地去设计自动化测试用例

在设计自动化测试用例时，除了实现用例来源中的功能步骤外，用例的原子性是需要额外注意的，这将影响到多个用例在一起时是否可以高效稳定地运行。用例的原子性，即指用例间应该保持相对独立，不因用例执行的先后顺序而彼此干扰。

(3) 此外，应该以工程的视角去看待测试用例

测试代码也应该以工程的视角去看待，包括配置管理、结构管理、项目化运作等等。在编写测试用例过程中也应该尽可能地从工程角度在代码易用性、维护性方面去多加考虑。测试代码也应该要有代码规范，包含命名规范、编写规范、注释规范等等，以使测试用例能高效有质量地运转起来。

(4) 最后，应该验证测试用例的有效性

自动化测试用例本身也是需要经过验证与测试的，一个测试用例本身运行通过了并不一定代表用例就是有效的。例如可能因为检查点判断有问题导致该用例始终通过，而一般当用例开始交付运行后，如果一直是通过的，那么往往就不会有人关注，且测试人员会认为该模块已经有自动化测试去保障从而容易忽略基本的测试，所以常常无效的自动化测试用例比没有自动化测试更可怕，需要警惕出现无效的测试用例。在编写测试用例时需要验证用例的有效性，在测试用例交付使用后，也应该定期地关注测试用例的运行情况及其有效性。

6.2.3、测试用例执行

1、传统命令行执行方式

(1) **Running all tests:** adb shell am instrument -w com.tencent.assistant.qa/android.test.InstrumentationTestRunner

(2) **Running a single testcase:** adb shell am instrument -w -e class com.tencent.assistant.qa.testcase.nuclear.MobileAccelerateTest com.tencent.assistant.qa/android.test.InstrumentationTestRunner

(3) **Running a single test:** adb shell am instrument -w -e class com.tencent.assistant.qa.testcase.nuclear.MobileAccelerateTest#testMgr_MobileAccelerate com.tencent.assistant.qa/android.test.InstrumentationTestRunner

(4) **Running multiple tests:** adb shell am instrument -w -e class com.tencent.assistant.qa.testcase.pangu.FoundTabActivityTest,com.tencent.assistant.qa.testcase.nuclear.AssistantTabActivityTest com.tencent.assistant.qa/android.test.InstrumentationTestRunner

2、使用 spoon 执行

```
java -jar spoon-runner-1.1.3-SNAPSHOT-jar-with-dependencies.jar \
--apk example-app.apk \
--test-apk example-tests.apk \
--class-name com.tencent.assistant.qa.testcase.common.SearchTest
```

Options:

--apk	被测 APK 包所在的路径
--fail-on-failure	当出现 failure 时，发现非 0 的退出码
--output	测试报告的输出路径，默认为 spoon-output
--sdk	Android SDK 的路径，若已配置可不填
--test-apk	测试 APK 的路径
--title	测试报告显示的标题
--class-name	测试用例类名，需要为带包名的全称
--method-name	测试用例方法名
--no-animations	禁止进行截图的 gif 生成
--size	只运行包含相应注解的用例 (small, medium, large)
--adb-timeout	设置每个用例支持的超时时间（默认为 10 分钟）

3、在 Eclipse 中执行

选择一个测试类后，右键 Run As —— Android Junit Test 执行即可

注：在 Run Configuration 中，如设置有多个 Instrumentation runner，则需要指定 InstrumentationRunner，如 0 所示：

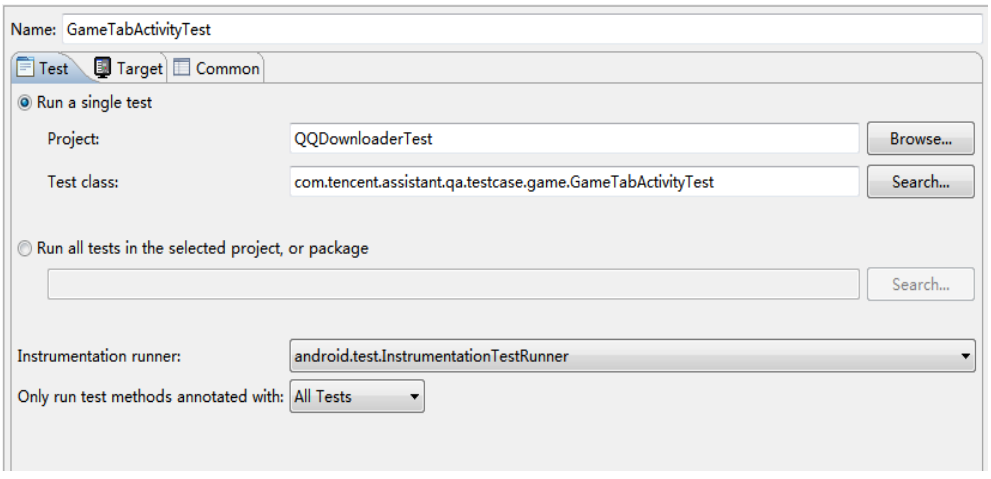


图 13 配置 Run Configuration

6.2.4、测试用例管理

当编写了较多测试用例时，就需要将测试用例分类管理起来，以方便统一维护及用例分级。基于 Junit 的测试可以使用 TestSuite 的方式进行管理。

由于在测试执行时，不同的用例执行时间长短不同，且作用的测试阶段也各不相同阶，因此在进行用例管理时，需要明确用例的级别，例如区分是核心功能用例还是普通用例，从而将不同级别的用例放于一处进行管理，在执行时才可以有针对性地进行测试。

6.3、测试报告

6.3.1、Spoon 报告

Spoon 是一个由主导有 okhttp、retrofit、leakcanary 等众多优秀开源项目的 Square 公司在 GitHub 上的开源项目，志力于改善基于 Instrumentation 的测试。通过分布式地在多台手机上同时执行基于 Instrumentation 的测试用例，并且在测试完成后生成统一的拥有测试结果概览、截图、运行时日志等等功能的 HTML 形式测试报告，Spoon 可以更加快速有效地对 Android 终端进行自动化测试。

项目开源地址：https://github.com/square/spoon

测试采用的 Spoon 生成，生成报告如 0 所示，其中绿条表示用例通过，红条表示用例失败:

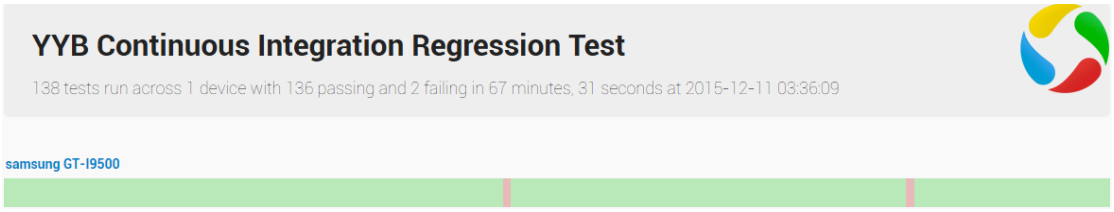


图 14 Spoon 报告首页

点击红条可跳转至失败用例的报告详情页，如 0 所示:

图 15 失败用例的报告详情页

用例采用出错重试并截图机制，当用例失败时进行截图，并往后开启截取一系列运行时的图片，每个用例右边有四个按钮，分别为将截图以 gif 格式播放、展示多台手机下同一用例运行情况、运行时日志查看、javadoc 文档链接。

例如点击右 3 按钮查看运行时日志，如 0 所示:

Found Tab, Bi Bei, Scroll List To Bottom And Assert Bottom Tips

Test failed in 3 minutes, 38 seconds on samsung GT-I9500

Timestamp	Level	Tag	Message
12-11 03:39:25.927	info	TestRunner	started: testFoundTab_BiBei_ScrollListToBottomAndAssertBottomTips(com.tencent
12-11 03:39:25.932	info	SingleLaunchActivityTestCase	setUp
12-11 03:39:25.932	info	SingleLaunchActivityTestCase	getName:testFoundTab_BiBei_ScrollListToBottomAndAssertBottomTips
12-11 03:39:25.932	info	SingleLaunchActivityTestCase	srcProtocolPath:/storage/emulated/0/Tencent/tassistant/log/ProtocolCost.txt
12-11 03:39:25.937	info	SingleLaunchActivityTestCase	batteryPct:100.0
12-11 03:39:25.937	info	SingleLaunchActivityTestCase	currentTestClass:com.tencent.assistant.qa.testcase.pangu.GroupListActivityTest
12-11 03:39:25.942	info	SingleLaunchActivityTestCase	isScreenShot:true
12-11 03:39:25.942	info	SingleLaunchActivityTestCase	runCount:0
12-11 03:39:25.982	info	System.out	pool-1-thread-4 calls detatch()
12-11 03:39:26.442	info	Jumper	tab name:发现
12-11 03:39:26.617	info	System.out	pool-2-protocalManager-thread-1 calls detatch()
12-11 03:39:27.137	info	Waiter	listViews.size() in waitForListView:2
12-11 03:39:27.737	info	ViewFetcher	views.size():2
12-11 03:39:27.737	info	ViewFetcher	locationOnScreen x:0
12-11 03:39:27.737	info	ViewFetcher	locationOnScreen x:38
12-11 03:39:27.777	debug	AbsListView	unregisterIRListener() is called
12-11 03:39:28.292	info	Waiter	listViews.size() in waitForListView:2
12-11 03:39:28.862	info	ViewFetcher	views.size():0
12-11 03:39:28.862	info	QQdownloaderTest	webview null

图 16 运行时日志

6.3.2、历史数据聚合报告

Spoon 会类似单元测试形式的 XML 报告文件，因此其他测试平台可以通过解析 junit-reports 目录下的 XML 报告获取用例执行的详情数据，对每次的测试进行入库存储，积累日常的测试数据，生成历史记录的测试报告页面。

图 17 历史数据聚合报告

七、持续集成

7.1、Jenkins 介绍

Jenkins，原名 Hudson，2011 年改为现在的名字，它是一个开源的实现持续集成的软件工具。官方网站：http://jenkins-ci.org/。

Jenkins 能实施监控集成中存在的错误，提供详细的日志文件和提醒功能，还能用图表的形式形象地展示项目构建的趋势和稳定性。

7.1.1、参数化构建

Jenkins 支持多种参数化构建，如 0 所示:

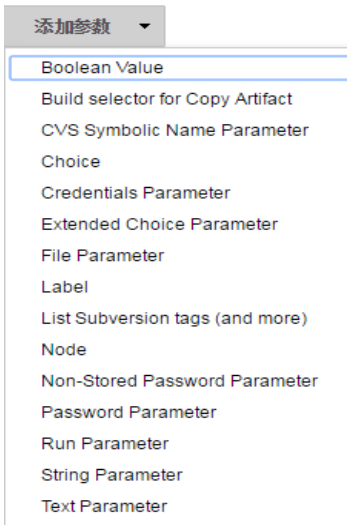


图 18 参数化构建

7.1.2、构建前

构建前可关联 SVN，设置定时触发器等等常规操作。

此外，安装相应插件后，构建前也可以删除 workspace 中的指定文件、设置当超时的时候是否停止构建、向 workspace 事先拷贝文件等等操作

7.1.3、构建

构建可以增加如 0 所示的诸多构建步骤:

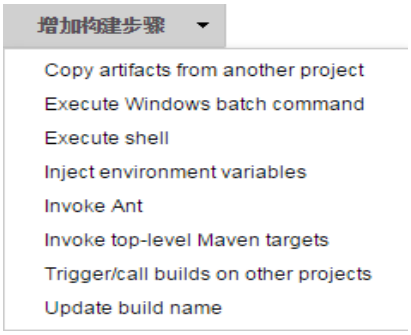


图 19 构建步骤

常用的有 Execute shell（在 Linux 机器中执行时），用于执行 shell 脚本

Execute Windows batch command（在 Windows 机器中执行时），用于执行 bat 批处理脚本

7.1.4、构建后

构建后可以选择如 0 所示的构建后步骤，常用的有邮件发送、触发新的构建任务、传递参数等等功能。

Aggregate downstream test results
Archive the artifacts
Build other projects
Jenkins Text Finder
Publish JUnit test result report
Publish Javadoc
Record Emma coverage report
Record fingerprints of files to track usage
Execute a set of scripts
Copy files back to the job's workspace on the master node
E-mail Notification
Editable Email Notification
Editable Email Notification Templates
Trigger parameterized build on other projects
Delete workspace when build is done

图 20 构建后步骤

7.2、整体流程图

由 7.1 节可知，Jenkins 支持参数化构建、关联 SVN、能设定触发时机、支持执行 Shell 或 bat 脚本、支持执行后邮件反馈、支持分布式运行等等一系列持续集成的流程。且 Jenkins 包含丰富的插件可以用于扩展功能，结合实际项目，因此应用宝使用 Jenkins 来做自化测试的持续集成，整体流程如 0 所示。

BVT 自动化测试根据不同的分支支持定时触发、分支监控及手动上传三种方式触发测试。任务创建后，将根据所选择的测试节点执行测试，测试用例采用基于 Robotium 框架编写，测试执行采用基于 Spoon 框架执行，因此支持在单台手机上执行也支持同时在多台手机上同时执行。测试执行完成后数据报告将回传服务端进行数据处理，数据处理完成将在相应平台上展示数据报告并邮件反馈相关负责人。

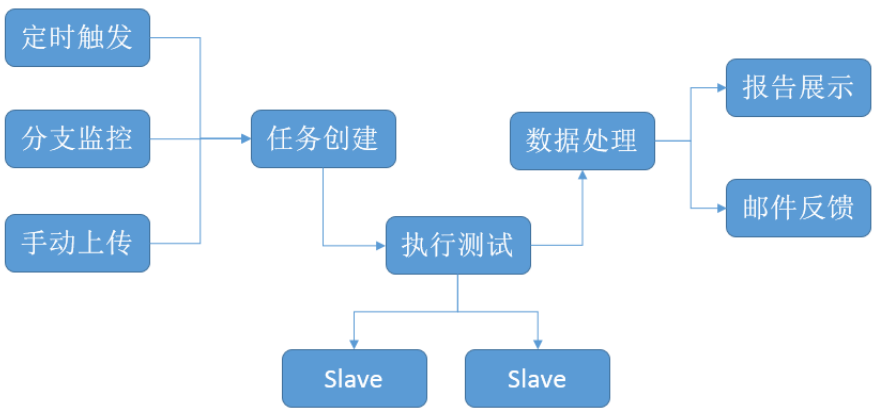


图 21 整体流程图

(1) 定时触发：用于主干每日夜里执行全量用例

(2) 分支监控：用于监控 DB 分支，当 DB 分支有新的构建时，就拉取相应 apk 进行 BVT 测试

(3) 手动上传：支持各 FT 及发布分支手动上传 apk 文件，手动触发 BVT 测试

(4) 任务创建：任务创建时会将测试工程进行编译打包生成测试.apk，并会将测试工程中需要用到地脚本文件、jar 包插件等统一拷贝至服务端的一个根据 job 名称命名的临时目录。

(5) 执行测试：在执行测试前，会将服务端该临时目录下的所有文件 push 至 Slave 执行机，然后执行相应的初始化脚本，例如卸载安装应用、清理手机中的残留数据等。

(6) 数据处理：在执行测试完成后，执行相应脚本，从手机中 pull 出测试产物，例如代码覆盖率用的 ec 文件、性能监控数据、协议日志数据、内存快照文件等。然后使用相应的 jar 包插件解析测试报告、上传数据至数据库等操作。

(7) 邮件反馈：调用邮件模版将测试报告发送指定的收件人。

在这种模式下，BVT 测试支持自动执行也支持手动触发执行，在 Jenkins 中以模版形式既可较灵活地进行配置，且配置维护也较为容易。另外任意能连接成为 Jenkins 节点的 PC 都可以迅速成为节点 PC 机，在节点 PC 上挂上手机即可成为系统的一部分，可以执行 BVT 自动化测试任务。

解放程序猿（媛）的双手
——iOS UI 自动化测试

◆ 作者：王琳

一、前言

随着移动互联网时代的蓬勃发展，移动终端的自动化测试也在业界日益活跃，总体来看在 Android 平台上的自动化工具和实践比较多，但是说到 iOS 平台无论从自动化工具的数量还是质量上就陡降了。究其原因，无外乎是 iOS 系统的封闭性，加上相对 Android 用户的数量少，导致对这个平台系统的自动化进展缓慢，据笔者了解到的情况，很多 iOS 平台的测试人员还处于纯手工测试模式，自动化程度和 Android 平台无法相论，更别提和 PC 端相比了。

然而越是困难的事，越是研究的少，就越具有挑战性。有挑战性的事大多又会带来不菲的收益，如果能在 iOS 上做出大规模可持续运行的自动化测试，那么对 iOS 的测试演进无疑是一次大的推动。

手机 QQ 浏览器（iPhone）测试小组的同学在比对和实践了业界已有的 iOS 自动化工具，总结提炼了对比，如下表所示。

工具	测试脚本语言	是否要原码	是否支持 webview	基于什么框架开发
Frank	Ruby	是	否	Cucumber
KIF	OC	是	否	XCUnit
Subliminal	OC	是	否	XCUnit
Calabash	Ruby	是	是	Cucumber

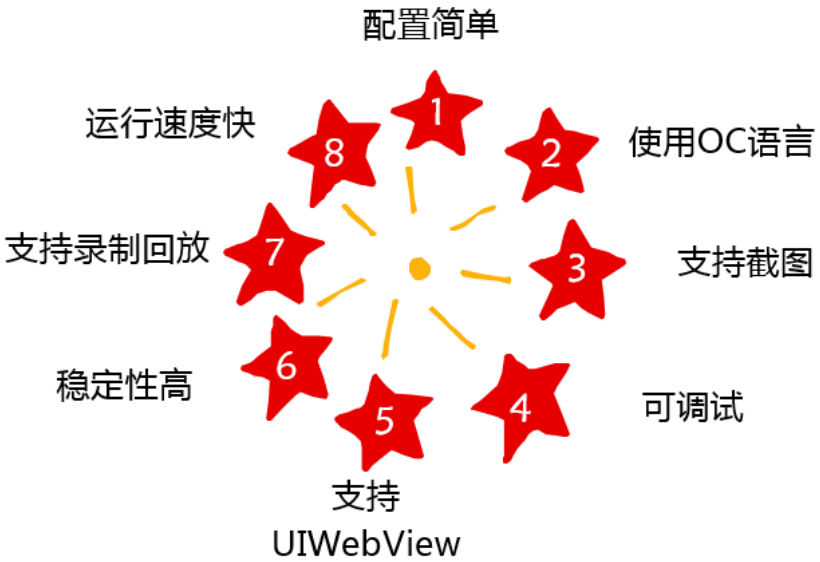
Appium	Ruby，C#,Java,JS,OC,PHP,Python,Perl	否	是	Agnostic
UI Testing	OC,Swift	是	是	XCUnit
UIAutomation	JS	否	是	原生

表中没有列的内容是稳定性，实践中看来上述除 UITesting 之外的工具稳定性都相对一般，会出现自身框架导致的各种闪退，以及性能越来越差的问题。

因此 iOS 平台上除了 Monkey 测试采用了自动化方式，以及部分性能测试轻度使用了一些自动化工具，大部分功能测试还是依赖于人的操作。移动 APP 更多的场景是面向用户，面向界面操作，UI 测试就非常直观和重要，因此我们致力于寻找一种稳定性高，且易于掌握的自动化测试工具。

二、久旱逢甘霖

苹果公司在 2015 年推出的 Xcode7 中引入了 UI Testing 工具，该工具配置相对简单，还支持录制回放功能，运行速度很快，测试代码也可以调试，使用 OC 作为脚本测试语言兼容性较好，支持 UIWebView，支持截图功能，最重要的是稳定性高。



其实选用一个工具，最看重的就是稳定性和可调试。可调试性不必多说，就是刚需。而很多工具没有被很好的应用就是因为稳定性较差，UI Testing 是苹果开发的，和被测程序的兼容性好，实践证明，确实能够彻夜稳定运行脚本，未见因工具自身原因导

致的闪退。性能方面也影响较小，后期发现一些截图操作会有一点点影响速度，但是整体运行还算良好，没有明显变差。

在大量工具都无法在 iOS 上施展拳脚时，UI testing 姗姗而来，带给我们惊喜和希望。

三、必经之路

工具是好工具，但是如果想在项目中实际应用起来，还是会遇到许多问题和困难，这里给大家分享下我们的磨合期。

第一个挑战：门槛要低

要使用这个工具，扑面而来的问题就是怎样快速上手，也即学习成本和投入产出比的问题。地球人都知道 OC 语言并不是一种容易快速上手学习的语言，加上底层是 XCTest 接口，录制后能看到的实现就是下图这样的，看着很凌乱有没有？

```
XCUIApplication *app = [[XCUIApplication alloc] init];
[app.buttons[@"\U7acb\U5373\U4f53\U9a8c"] tap];
[app.scrollViews.otherElements.scrollViews.otherElements.otherElements[@"\U5934\U6761"] tap];
[app.buttons[@"\U519b\U4e8b"] tap];
```

因此我们需要做的就是进行封装。封装包含接口封装和特殊控件的封装两部分。

接口封装有两层，一层主要处理弹窗，异常，超时等待，手势处理等操作，二层是把控件类型和手势合二为一，提供脚本使用。实际脚本编写者就可以直接调用，命名和使用方式都具备高可读性和可用性，另一方面底层的接口修改，不会影响到上层的函数表现形式，这对自动化用例编者来说也降低了门槛，不必关注底层接口的细节变动。

例如上图所示，浏览器多窗口界面向左滑动就可以删除页面，可以选择删除第一个页面。将这个操作封装起来如下图所示，后续使用即可直接调用该函数，只需传入对应参数即可。

```
//多窗口左滑动
- (void)swipeLeftMultiWindow:(NSUInteger)index VariableParameter:(NSString *)VariableParameter{
    XCUIApplication *app = [[XCUIApplication alloc] init];
    XCUIElement *element = [[[[[[app.otherElements containingType:XCUIElementTypeNavigationBar
        identifier:@"UIBrowser"] childrenMatchingType:XCUIElementTypeOther].element
        childrenMatchingType:XCUIElementTypeOther].element childrenMatchingType:XCUIElementTypeOther
        ].element childrenMatchingType:XCUIElementTypeOther] elementBoundByIndex:1];
    XCUIElementQuery *scrollViewsQuery = element.scrollViews;
    XCUIElement *multiElement = [[[[scrollViewsQuery childrenMatchingType:XCUIElementTypeOther]
        elementBoundByIndex:index] childrenMatchingType:XCUIElementTypeOther].element
        childrenMatchingType:XCUIElementTypeButton].element;
    NSString *description = [self doVariableParameter:VariableParameter];
    [self stepLogStart:description];
    if(![self swipeLeftXCUIElement:multiElement]){
        [self errorLog:description];
    }
    else{
        if (warnFlag) {
            [self screenShotMemory];
            [self warnLog:description];
        }
        else{
            [self rightLog:description];
        }
    }
    warnFlag = FALSE;
    [self stepLogEnd];
}
```

特殊控件的封装。一般应用于控件无法寻找到对应的唯一标识，或者层级比较多的情况下，通过录制的方式，读取控件属性，将其打包成一个整体，再后续使用时可以直接调用。

例如上图所示是浏览器多窗口的管理界面，右下角有个返回按钮，通过查看 xml 结构无法获知唯一标识，通过录制的方式确定控件结构。对录制的内容进行加工处理后，封装为特殊控件，如下图所示，存放于指定文件内，方便后续使用。

```
    return element;
}
//获取多窗口返回按钮
-(XCUIElement *) getMutiBackElement;{
        XCUIApplication *app = [[XCUIApplication alloc] init];
    XCUIElement *toolbar = [[[[[[[[[app.otherElements containingType:XCUIElementTypeNavigationBar
        identifier:@"UIBrowser"] childrenMatchingType:XCUIElementTypeOther].element
        childrenMatchingType:XCUIElementTypeOther].element childrenMatchingType:XCUIElementTypeOther
        ].element childrenMatchingType:XCUIElementTypeOther] elementBoundByIndex:1]
        childrenMatchingType:XCUIElementTypeOther].element childrenMatchingType:
        XCUIElementTypeToolbar] elementBoundByIndex:1];
    XCUIElement *button2 = toolbar.buttons[@"退出多窗口"];
    return button2;
}
```

使用中，可以直接调用已经封装好的接口，每个接口都包含一个或者多个固定参

数，和一个可变参数。固定参数按照所给出的类型传值就可以了，而可变参数，需要我们按照一定的格式进行参数传递。举例可变参数格式：VariableParameter:@” StepName=切换小说书架为官格模式&LogLevel=WARN”,其中多个参数使用&符号连接，每一个单独的参数使用 KEY=VALUE 的形式。

第二个挑战：灵活编译

在自动化测试中经常遇到的问题是配置环境和单例运行的问题。

首先环境配置会不会很复杂，很大程度上制约着使用者的使用频率。相比较 APPium 的复杂设置，UI testing 的配置就简单明了多了。

(1) 添加服务端配置。需要将 UITestServer.xcodeproj 添加到目标工程的 thridparty 文件夹下面。UITestServer.xcodeproj 是自研服务器，主要功能是：嵌入被测试 APP 中，实现端口监听，服务开启，消息获取，消息处理，各种事件功能（截屏，获取被测试 APP 日志信息，获取内存，cpu，网速，流量等功能），还可以扩展。

(2) 配置被测 APP。需要在被测 APP 的主加载函数中加入监听服务端口。为了不影响发布版本的使用，我们采用 DEBUG 模式。

(3) 配置 QBUITests（名字自定义）组件部分，该部分主要是我们的自动化测试框架部分，包括各种自动化组件，自动化脚本，配置信息等。需要将自研文件夹 UITestUtils，SystemResources，SpecialElement，ScreenShot，Log，ScriptInterface 全部拖动到 QBUITests Target 下面。

经过上面的配置，框架配置基本完成。

接下来看单例运行问题，如下图所示是小说模块的自动化脚本头部，包含开头的初始化操作，直接可以运行单例 “test311001”，也可以进行正常的调试，也可以指定运行全部用例或者部分脚本。

```
#import "MttScriptInterface.h"

@interface MttNovelScript : MttScriptInterface

@end

@implementation MttNovelScript

- (void)setUp {
    [super setUp];
    // Put setup code here. This method is called before the invocation of each test method in the class.

    // In UI tests it is usually best to stop immediately when a failure occurs.
    self.continueAfterFailure = NO;
    // UI tests must launch the application that they test. Doing this in setup will make sure it happens for
        each test method.

    [[[XCUIApplication alloc] init] launch];

    // In UI tests it's important to set the initial state - such as interface orientation - required for your
        tests before they run. The setUp method is a good place to do this.
}

- (void)tearDown {
    // Put teardown code here. This method is called after the invocation of each test method in the class.
    [super tearDown];
    [self endScriptLog];
}

- (void)test311001 {
    //首次进入小说书架，查看书架
    [self WriteUITestLog:@"MttNovelScript" scriptName:@"311001" scriptDescription:@"首次进入小说书架，查看书架"];
    [self tapScrollViewsOtherElementsButtons:@"小说" VariableParameter:@"StepName=进入小说书架"];
    [self existStatics:5 VariableParameter:@"StepName=判断存在5本以上小说"];

    //未登录显示
    [self tapNavigationBarsButtonWithLable:@"novel shelf login btn@2x" VariableParameter:@"StepName=点击进入个人中
        心"];
    [self existStaticsWithLable:@"个人中心" VariableParameter:@"StepName=判断存在个人中心标签"];
    [self tapButtonWithLableQuery:@"返回" XCUIElementIndex:0 VariableParameter:@"StepName=点击书架"];
```

第三个挑战：结果可查

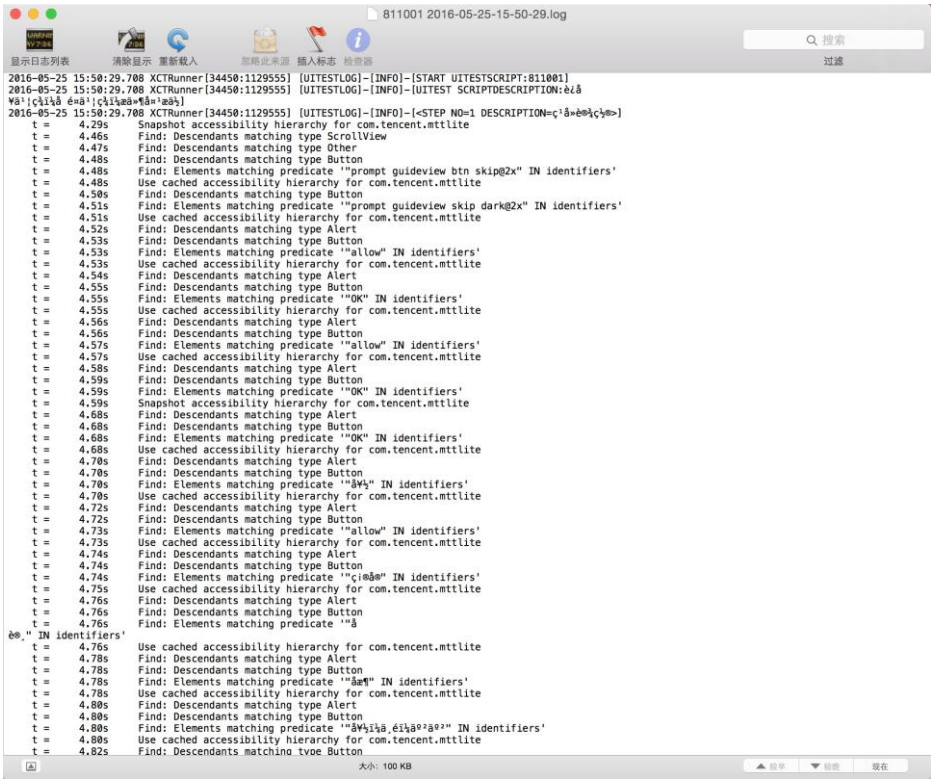
做 UI 自动化也好，做监控或者准入测试也罢，最终都要有个输出结果。一般来说对于自动化测试的结果要具备两点：截图和日志。截图可以时时截图，也即可以让自动化设计者或者测试人员随时获取想要的结果图片，也可以设置发生错误时自动截图。日志系统就要详细记录操作过程，在确认问题是可以一步步通过日志追溯。

UI Testing 除了控件识别和简单操作外，并没有提供屏幕截图功能，我们需要自己完成屏幕截图功能，而且还要能够在各种封装好的函数中灵活使用截图功能。如下图所示，在自动化用例脚本中使用函数时，可以随时在可变参数里设置 LogLevel=WARN，即可在操作执行过程中进行截图。

```
[self tapButtonWithLable:@"美图" VariableParameter:@"StepName=进入美图tab&LogLevel=WARN"];
[self swipeDownWithXCUIElement:[self getFanyeElement] VariableParameter:@"StepName=下拉列表&LogLevel=WARN"];
```

当然在程序运行异常或者元素找不到的时候也会自动截图。这些截图的操作都默认放在封装函数里了，使用者不必单独设置。

系统日志的获取分为两种，一种是过程中的操作记录，一种是内存之类的性能日志。日志的获取需要用到跨进程通信，前者主要是记录自动化用例执行过程中的操作步骤和截图信息，后者主要是获取设备的系统性能数据，例如内存、CPU、网络等。生成的本地 log 如下图示。



最终整合到日志系统（后文讲解）下图。

四、发展壮大

在初次磨合后，基本上有了本地单机运行自动化的能力，但是要真正对项目有所帮助，就必须有比较好的运行方式和结果展示，因此对这个自动化的研究还需要进一步发展。

1）整体架构

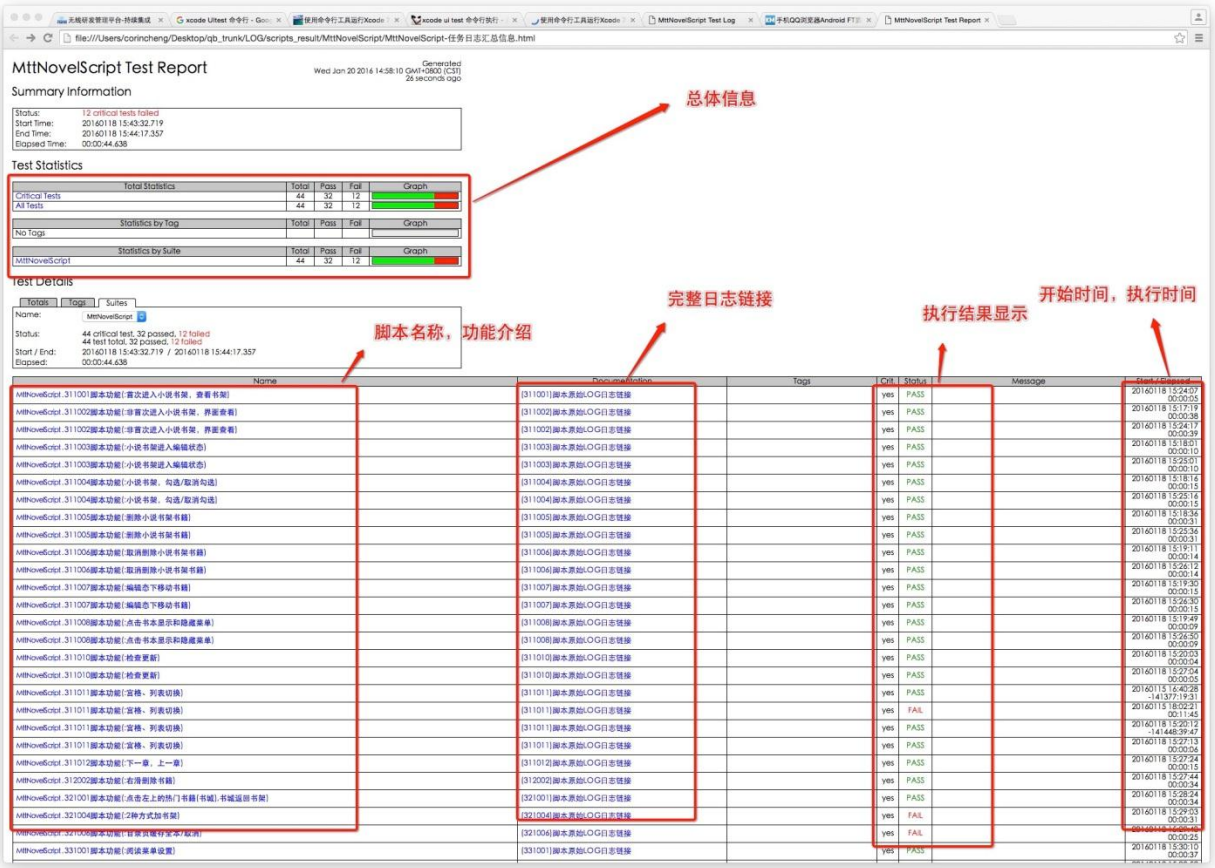
首先要对基于 UI Testing 的自动化整体架构有个了解。如下图所示，使用系统提供的 XCTest 接口、消息处理、驱动模块、系统资源获取，在中间层进行封装，包括控件调用封装，特殊控件封装，截图模块，日志处理模块。这些内容在上文都有讲述。

关于整体架构的内容在图中的最上层。一个是集成在 XCODE 里边的自动化运行框架和脚本，另一个是分析 log 日志的自动化 log 日志分析系统。如上图所示是在基于控件调用驱动的基础上，使用自动化脚本和配置文件完成自动化测试的工作。然后使用日志分析系统，包含日志分析、展示、邮件等，给到项目团队以完整的可视化报告。

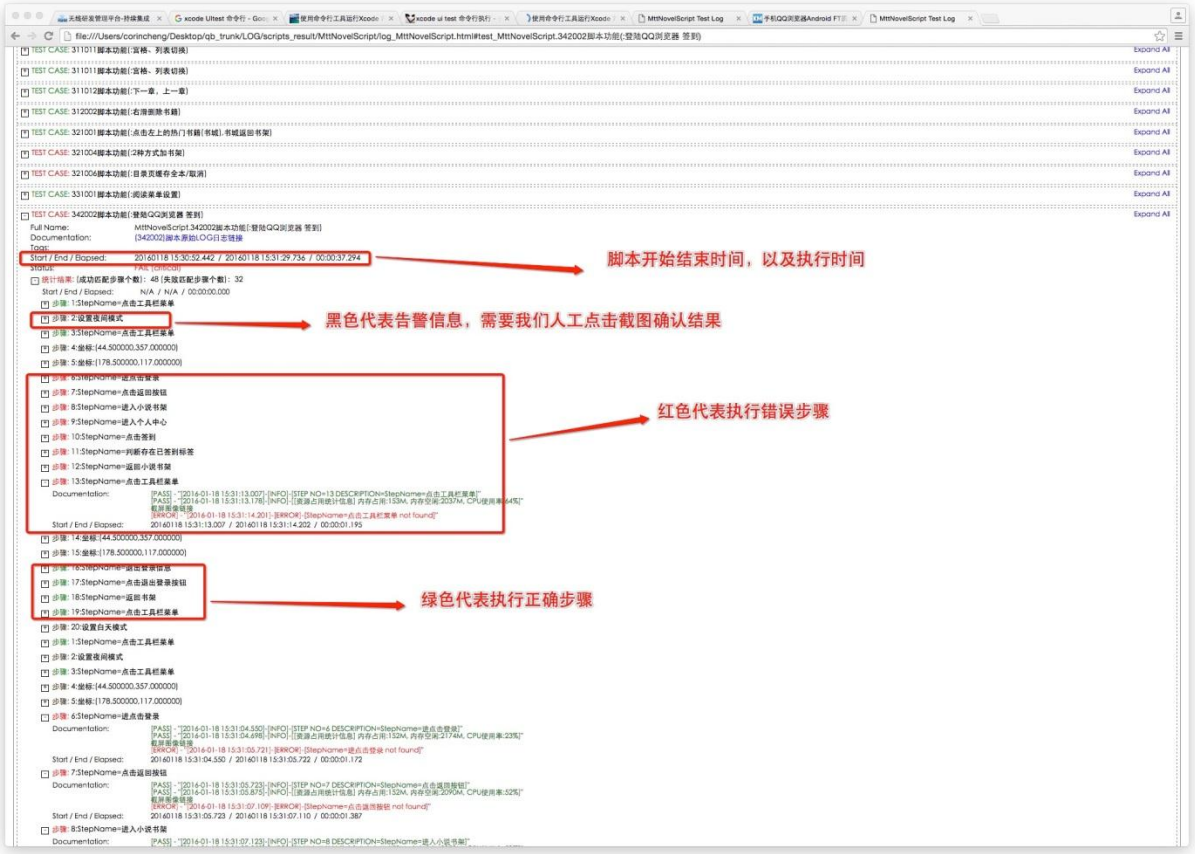
2）日志系统

日志分析系统是使用 python 开发的系统，所以其是独立于 XCODE 框架的，当自动化脚本执行完成后，需要把 log 日志传到分析系统下，然后使用该系统分析出日志，方便用户查看脚本错误信息，分析定位问题。

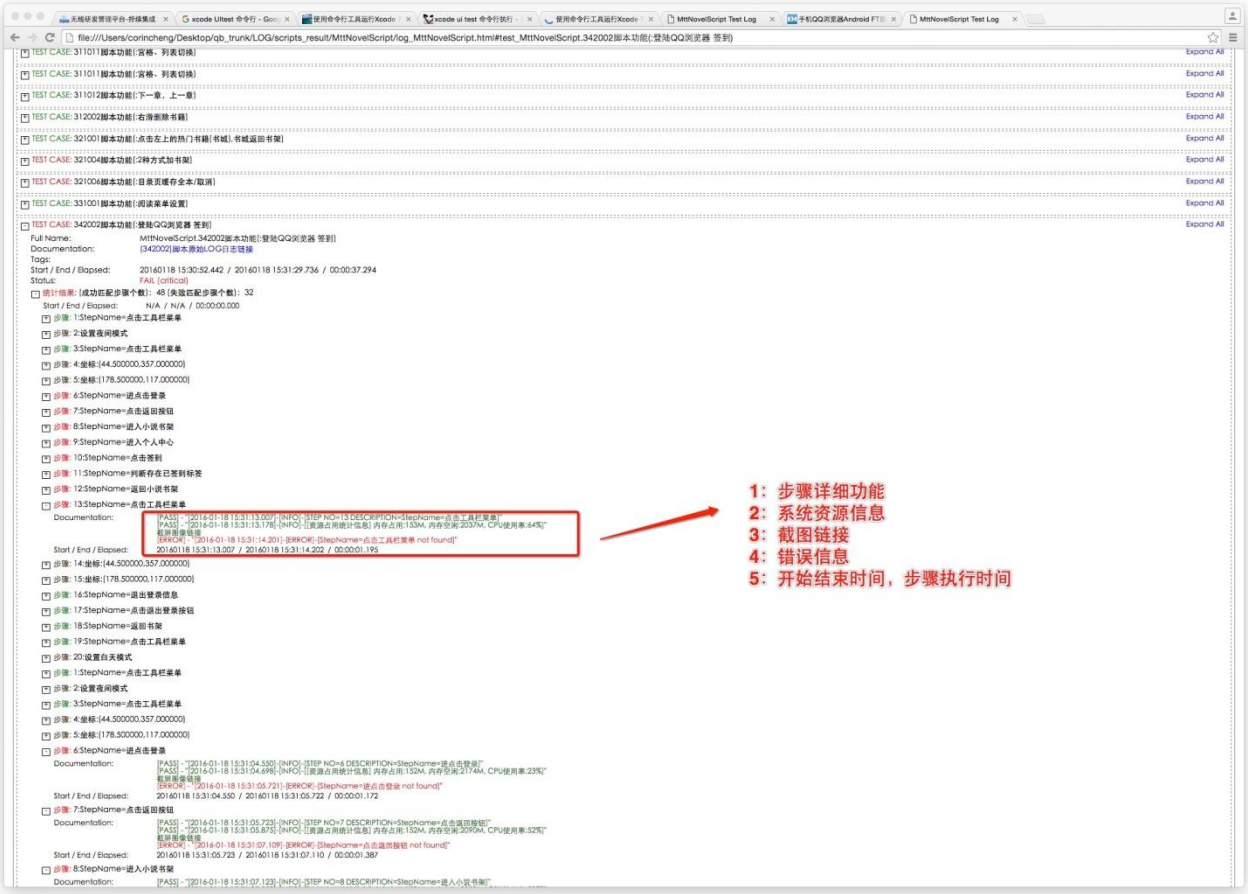
自研的日志分析系统可以将每次运行的日志升成 html 格式，如下图所示。



详细脚本日志介绍，如下图所示。



步骤详细信息，如下图所示，一般只需关注红色错误信息。



通过该日志系统就可以获得良好的可视效果，方便问题追踪和回溯。

3）持续集成

功能的自动化测试，最终的期望还是能够随着版本进行持续测试。有些比较好的应用场景：准入测试、开发自测、回归测试。不管是哪种应用场景，最佳方式都是将自动化框架合入主线，方便项目组的不同成员随时在任何开发机上运行。

但是！这里是 iOS 平台，有一种忧伤叫做苹果审核不通过，有一种致命拒绝叫做查出私有 API。虽然配置简单，合入主线跟随代码构建看起来不是个难事，但是难就难在需要打包升成版本的时候不能将自研部分的服务器代码编译进去，这部分会经常含有私有 API（一些功能的实现必须要用到私有 API）。因此我们采用了动态关联的方式，在主函数所在的文件中加入下图所示内容。既能顺利将框架合入开发主线，又可以在编包发布时不编译这部分代码。

五、实践效果

凡事都讲究个投入产出比，前期做了大量的预研和实践工作，那究竟在项目实践中

能发挥怎样的效果呢？接下来为大家展示一下。

1）部署情况

目前手机 QQ 浏览器（iPhone）项目上，已经采用这种基于 UI Testing 的自动化测试方法进行 BVT 建设，每天晚上测试白天提交到主线的最新代码，保障主线质量稳定，并为第二天早上的提测包做一个准入测试。

在部署时考虑到版本迭代以及 UI 变更大的问题，主要是在浏览器基础 FT 上进行了大量自动化测试部署，还对用户访问的 TOP 页面进行检测，如下图所示。确保合入主线的代码不会影响到浏览器基础功能，及时发现问题，避免提测后再暴露问题，造成时间和资源的浪费。

BVT统计信息

FT名称	失败用例数	通过用例数	开始时间	结束时间	用例通过率
视频	0	4	20160527 18:19:02.279	20160527 18:23:10.182	100.0%
主页书签	0	5	20160527 17:37:44.718	20160527 17:39:53.630	100.0%
TOP100站点检测	0	23	20160527 17:49:59.647	20160527 17:58:11.539	100.0%
主页feeds	0	13	20160527 17:41:19.817	20160527 17:49:56.438	100.0%
今日头条	2	7	20160527 17:58:14.557	20160527 18:03:03.260	77.78%
搜索	0	10	20160527 18:15:45.330	20160527 18:18:59.269	100.0%
小说	0	29	20160527 18:03:06.273	20160527 18:15:41.501	100.0%
全屏	0	3	20160527 17:39:56.725	20160527 17:41:16.751	100.0%
多窗口	0	5	20160527 18:23:13.191	20160527 18:27:35.533	100.0%

发现有问题的地方还可以进入详情查看，如下图所示点击测试 LOG 链接。

详细脚本执行统计信息

FT名称	脚本	执行结果	功能	开始时间	结束时间	测试LOG链接
今日头条	500007	失败	将正文页内容保存为书签	20160527 18:02:10.848	20160527 18:02:29.117	500007
今日头条	500005	失败	正文页横竖屏转换	20160527 18:01:23.899	20160527 18:01:55.115	500005
视频	111003	成功	查看土豆视频网站的播放情况	20160527 18:19:02.279	20160527 18:20:34.826	111003
视频	111009	成功	横屏播放界面，关注，开启弹幕设置	20160527 18:20:38.413	20160527 18:22:09.655	111009
视频	111010	成功	视频全屏下，直播显示测试	20160527 18:22:12.823	20160527 18:22:38.267	111010
视频	111011	成功	视频播放常驻测试	20160527 18:22:41.403	20160527 18:23:10.182	111011
主页书签	0811003	成功	同步书签	20160527 17:37:44.718	20160527 17:38:12.522	0811003

另外对一些浏览服务 SDK 也进行了自动化测试监控，如下图所示。

TBS-BVT统计信息

FT名称	失败用例数	通过用例数	开始时间	结束时间	用例通过率
TBS-UIWEBVIEW功能	0	19	20160527 10:05:56.628	20160527 10:11:45.957	100.0%
TBS-WKWEBVIEW功能	1	18	20160527 10:30:52.674	20160527 10:36:04.985	94.74%

2）投入产出比

投入产出比的问题，要看两个方面，好比天平的两端，一端是投入，一段是产出。得产出重过投入才是一个好项目，值得长期运营。

如上图所示，我们的投入成本可以分成两块，分别是一次性成本和线性成本。我们以应用于实际项目的 iOS 上的 BVT 来说明。

一次性成本：框架研发（1 个人 2 个月）+配置和部署（4 人 0.5 天）+学习成本（3 人 1 天）。这个一次性成本主要消耗在框架的研发上，以及测试人员的初始培训上，后续只有新加入测试人员才会增加这个成本。事实证明在一次性成本上的投入非常值当，好的框架可以保证提高后期运维阶段的稳定性和使用的简易性。

线性成本：自动化用例编写（平均 14 分钟/条），每日需要维护的成本（8 分钟左右）。线性成本随着时间的推移可能会产生变化，例如自动化用例随着测试人员的熟练程度单条用例的编写时间会减少，每日需要维护的成本随着用例数的增多和需求变动增多会增高，这些都在预期范围内。

产出方面，我们的评估分为客观和主观两方面。

客观：

前置 bug 暴露时间。BVT 每日运行，因此总会提前于正式提测前暴露问题。目前是部署在主线上，主线的提测频率大约 40 天能提测 10 次左右，也即平均下来差不多 4 天才能提测一次，一旦 BVT 发现问题，平均能前置三天发现。由于数量都是平均来算，

因此只能大约估值，例如发现了 29 次问题，前置时间就是 29*3 天=87 天。

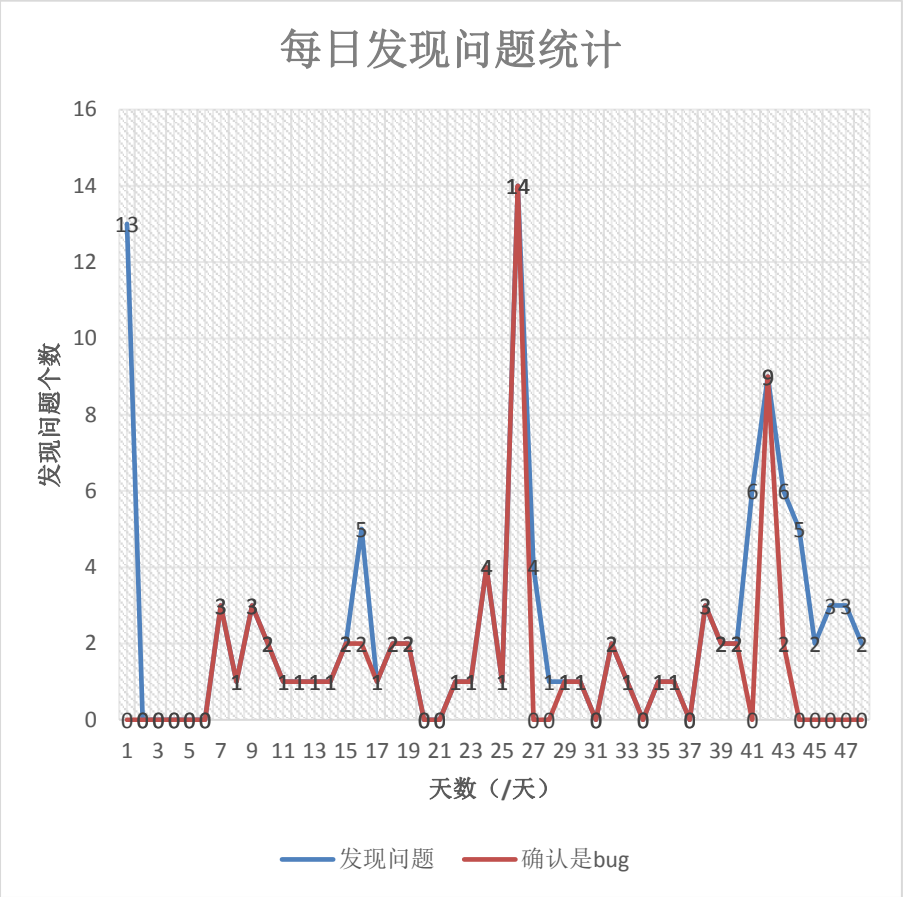
减少提测拒绝次数，节省人力时间成本。由于 BVT 里的自动化用例全部是基础核心用例，一旦出现运行问题，就是不符合准入测试标准的。在没有 BVT 的时代，提测前都是开发手工自测和测试手工验证的方式进行，一旦发现不符合测试条件的 bug，就会打回，这种情况下就会消耗不少的人力和时间。有了 BVT 后，这种情况大大减轻，开发可以自己运行自动化脚本做基础功能自测，测试每日监控也在运行检测。 目前手工运行一遍自动化的用例大约需要 200 分钟，机器在夜晚用 60 分钟就搞定，每个版本的拒测次数大约是 2 次左右。如果用狂野的算法应该是节约了 200*2=400 分钟，低调点的算法，应该是“从开始测试到发现导致拒测的 bug 所用的时间”*拒测次数，但是这个数据暂时来说不可考。

主观:

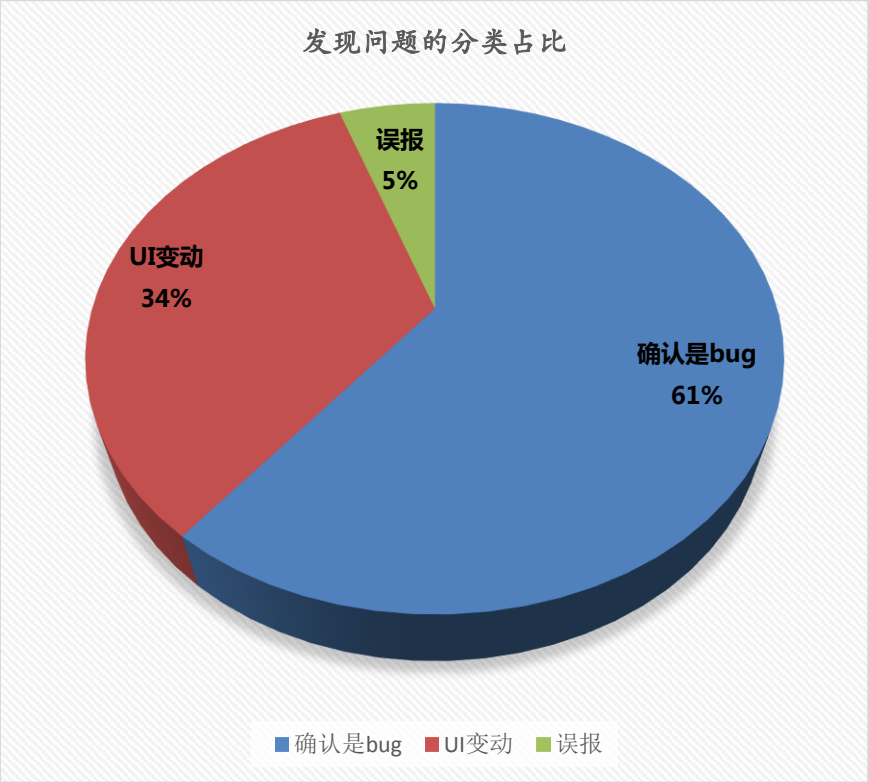
为什么要放上主观收益呢，因为客观上节省的时间，在主观上还要有个内心“更淡定”了的感受。总体来说 BVT 的部署，大大提高了测试在项目组的影响力，从此 iOS 上的测试从纯手工迈入了新时代，每日版本质量也有了持续稳定的检验，全项目组的内心也更加淡定了。

3）运营数据

如下图所示是每日发现问题的统计，金色的线是确认是 bug 的数量，蓝色的是 BVT 每日报出来的问题，可以看出两线基本重合的点比较多，这反映了误报率相对维持在一个比较低的水平。



发现的问题中主要分为三类，如下图所示，分别是纯误报（因为脚本的稳定性导致的）、UI 变动（包含被测元素变动、需求变更）和真实 bug。如下图所示是统计的两个月的数据，可以发现 UI 变动导致的问题占总发现问题的比例相对较低。这些数据是在没有与开发约定代码规范的时候，随着后期的合作，这部分 UI 变动导致的问题中的元素属性变动问题将会降低，但是纯需求变动的问题还是保持一定的比例。



六、写在最后

最后，还是需要和读者交流下这个自动化方案的问题和未来。

目前已知的问题是基于 UI Testing 的自动化测试方案需要的硬件要求。联机操作或者模拟器，得有 iMac，操作系统得是 OS10.10.5 及以上版本，Xcode 版本得是 7.1 及以上，Python 版本 2.7 及以上，内存 2G 及以上为佳。没有这个配置，恐怕无法良好的运行自动化测试。

还有一个性能问题，已经封装好的工具脚本总数理论上控制在 1000 个以内，可稳定运行 10 小时以上。在这稳定运行的 10 小时内，可以满足我们的自动化需求，但是时间超过几个小时后，还是会有些许减速，可以通过修改底层消息传递机制，提升传递效率，减少截图运行时间，也是今后自动化测试继续优化的方向。

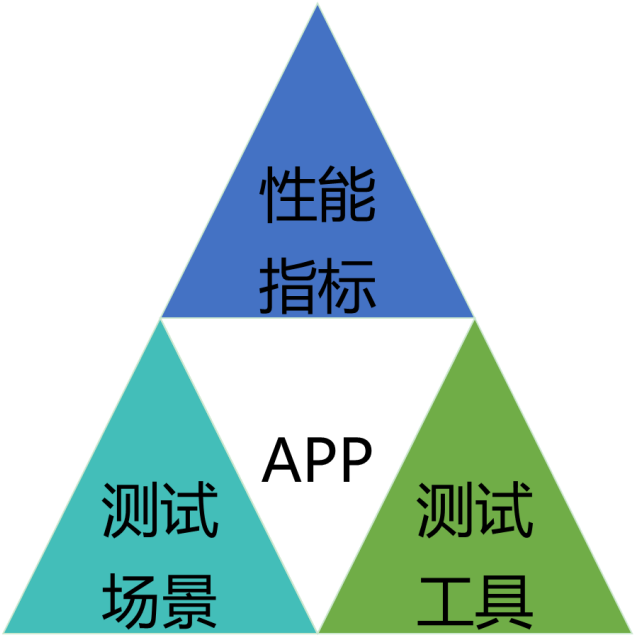
未来呢，继续扩大脚本覆盖范围，对特殊控件做封装，对函数的灵活性做优化，降低自动化测试的准入门槛，提升自动化测试的效率，期待更多同行业的牛人一起探讨！

探秘 APP 性能三角区

◆作者：刘燕

APP 要做性能测试，什么样的数据能反应应用的性能情况，如何评估应用的性能状态？不知道该如何入手？一起来分析下如何给 APP 做性能测试。

性能测试三角：性能指标、测试场景、测试工具。



首先要思考选哪些指标来评估性能：内存、cpu、电量还是什么？接着，选择你需要测试的场景，测试场景描述了你需要在何种场景下取性能数据，要测试 APP 何种功能等等。最后，根据你的指标和场景选择适合你的测试工具。

下面就从这三方面来具体分析。

一、性能指标

常见的性能指标有：内存、CPU、电量、流量、速度/耗时。这里从 2 个角度分析：

(1) 为什么选这个指标？

(2) 指标常用单位有哪些？

着重讲下关注最多的：内存、CPU。

1、内存

为什么要选内存呢？需要知道 Android 的 OOM 和 Low Memory Killer。

OOM：Out Of Memory，顾名思义是说内存不够用或者耗尽了，进程会被强制终止。安卓框架限制了每个应用进程所占用的最大内存值。关注内存的一个目的就是避免内存使用过大，出现 OOM。主要关注内存使用较多时的场景，例如游戏 app 正在游戏中。

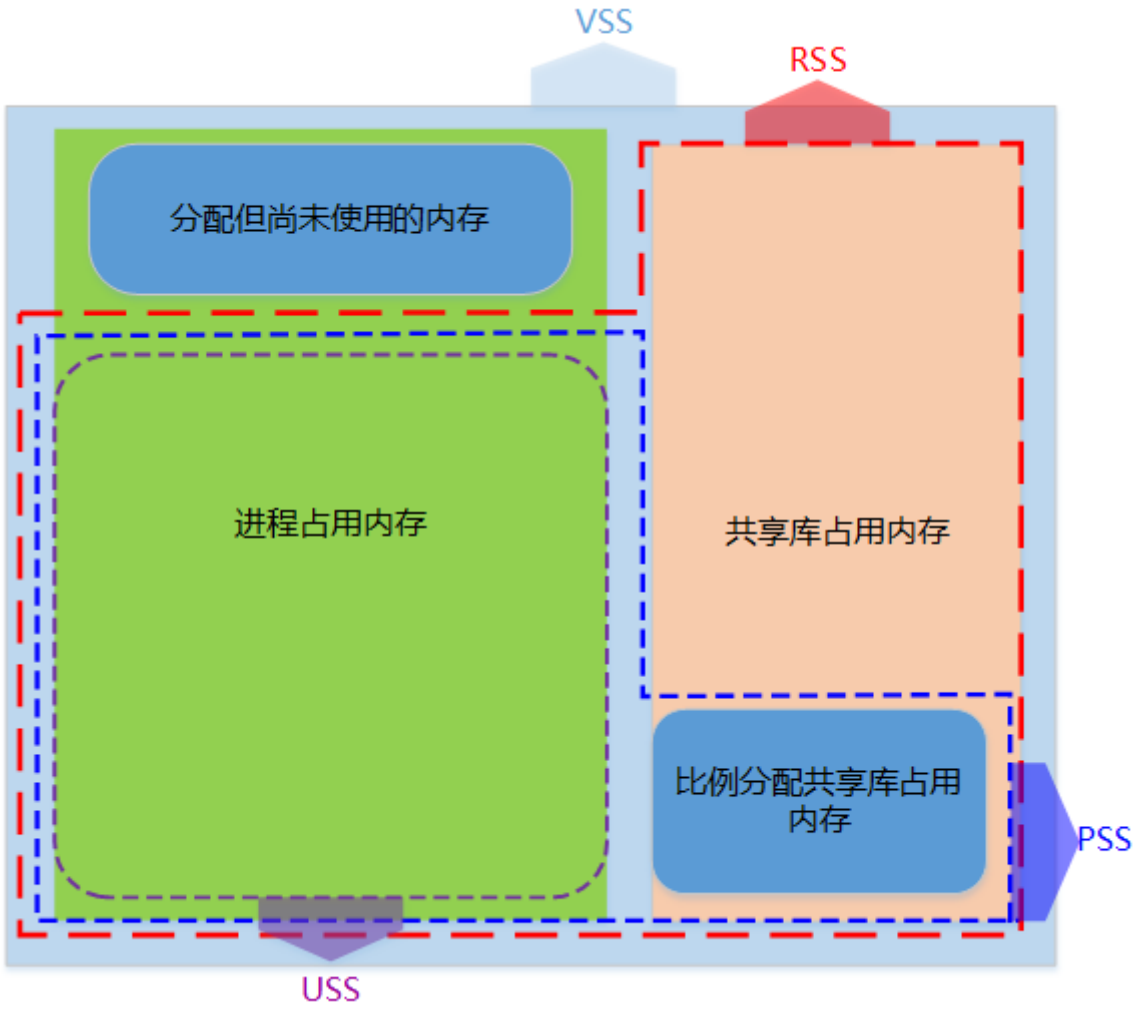
Low Memory Killer：Low Memory Killer 在用户空间中指定了一组内存临界值，当其中的某个值与进程描述中的 oom_adj 值在同一范围时，该进程将被 Kill 掉。如果你的 APP 某个进程需要一直保存存活，你需要保持你的进程优先级足够高，并且占用比较小，因为 Low Memory Killer 在工作时，同一优先级的进程会先 kill 那个占用最大的。性能测试时主要关注待机时的内存是不是够小。

这里再补充一点：Low Memory Killer 的工作可能致系统变卡。为什么呢？因为它 kill 了一些进程，然而现在市面的很多 APP 为了保活都会自启，刚刚被 kill，立刻又起来。启动占用大量内存（还有 CPU），又触发 Low Memory Killer。频繁的被 kill 和启动形成了恶性循环，so…系统变的很卡。

内存指标有 VSS、RSS、PSS、USS。差别如下：

- VSS- Virtual Set Size 虚拟耗用内存（包含共享库占用的内存）
- RSS- Resident Set Size 实际使用物理内存（包含共享库占用的内存）
- PSS- Proportional Set Size 实际使用的物理内存（比例分配共享库占用的内存）
- USS- Unique Set Size 进程独自占用的物理内存（不包含共享库占用的内存）

一般来说内存占用大小有如下规律：VSS >= RSS >= PSS >= USS。



测试中比较常见的的选择是 PSS total，这种算法共享库内存按比例分配，对 APP 来说比较公平。依据 APP 关注点，也可选择其他指标例如 USS，或者将其他指标也一起统计，用于分析。

```
** MEMINFO in pid 27465 [com.tencent.qqpimsecure] **
                   Pss  Private  Private  Swapped     Heap     Heap     Heap
                 Total    Dirty    Clean    Dirty     Size    Alloc     Free
                ------   ------   ------   ------   ------   ------   ------
  Native Heap     2190     2168        0        0     3900     2890      345
  Dalvik Heap     4287     3744        0        0    19772    18563     1209
Dalvik Other      4638     4548        0        0
       Stack       488      488        0        0
      Ashmem         4        4        0        0
   Other dev        20        0       20        0
    .so mmap       666      300       92        0
    .jar mmap       72        0        0        0
    .apk mmap       46        0        0        0
    .dex mmap     3947      300      916        0
   Other mmap       30        8        0        0
      Unknown      320      320        0        0
        TOTAL    16708    11880     1028        0    23672    21453     1554
```

例如用 dumpsys meminfo 命令看到，手机管家进程的 pss total = 16708 kb。

2、CPU

为什么要关注 CPU?

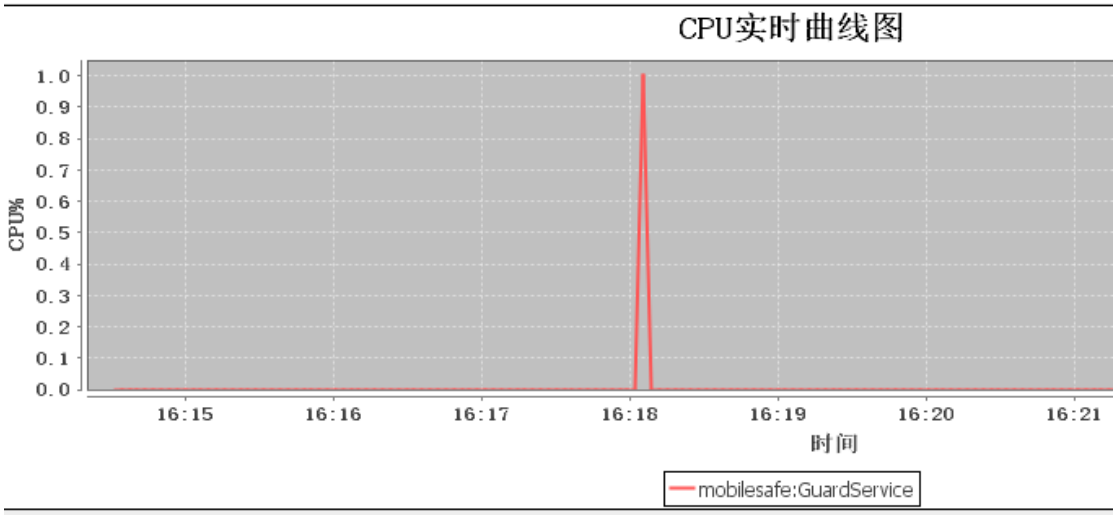
（1）CPU 使用率

想必你肯定有这样的经历：玩某个游戏或者 APP 的时候，手机发热发烫。是的，CPU 的频繁使用，会让你的手机发烫，让你的手机变卡（CPU 资源不足）。如果让用户发现你的 APP 用起来发烫，那就等着他的吐槽和卸载吧。

也就是说 CPU 性能，我们需要关注 APP 使用中 CPU 消耗情况，通常会使用 CPU 使用率这个指标。

（2）CPU jiffies

如果 APP 在退出界面后还有进程长期运行，那你需要关注下待机场景的 CPU。待机场景下 CPU 的消耗一般不会很大，例如手机管家，可能消耗经常是 0%，1%，长时间平均下，可能只有 0.1%、0.2%，看看竞品，也是差不多，好像没有太大区别。



那么 CPU 消耗这么少是不是就不用管 CPU 了呢，然而即使是平均值很小，但是长时间待机，例如安全类工具，CPU 的消耗还是不容忽视。那么这种情况如何评估 CPU 呢，这里引入一个更精确的指标：CPU 时间片：jiffies!

Jiffies：为 Linux 核心变数(unsigned long)，它被用来记录系统自开机以来，已经过了多少 tick。一定时间占用的 jiffies 可以反映出此进程的 CPU 消耗。

Android 系统可以获取到 APP 进程当前的 CPU jiffies（这个数值不断累加）。我们测试时常用的单位有：消耗 XX jiffies/分钟；半/1 小时共增加 XX jiffies。

Tips: Jiffies 与变频

Linux 内核中提供了 performance 、powersave 、userspace、conservative 和 ondemand 五种变频模式供用户选择使用，它们在选择 CPU 合适的运行频率时使用的是各自不同的标准并分别适用于不同的应用场景。那么，测试 CPU jiffies 的时候是否需要固定 CPU 频率呢？理论来讲，固定 CPU 频率肯定是可以的，那么不固定会怎样呢？

经测试数据验证，保持手机同环境情况下，不固定 CPU 频率对于测试无太大影响。

jiffies/分钟	第1次	第2次	第3次	第4次	第5次	平均值	标准差
测试数据	25.57	28.23	28.84	25.43	27.7	27.15	1.56

这组测试数据是没有固定 CPU 频率情况下测试了 5 次的 jiffies 数据（每次 30min），可以看出，标准差为 1.56，波动并不大，基本可以排除变频的影响。

3、电量

手机电池资源有限，电量的重要性就不必说了。现在很多手机都有电量排行，如果你的 APP 总是排在前面，小心被卸载哦。电量通常的单位是：mAs 或者 mAh。

4、流量

手机的一个特点就是有移动网络。移动网络下的流量消耗需要特别关注，wifi 下的流量优先级略低。流量单位：kb，M。

5、速度/耗时

可用性原则里面有个 2 秒原则：一个松散的原则，即用户没有必要对某些系统响应等待 2 秒以上的时间，比如应用程序转换和开始的响应时间。对于启动 APP，进入某页面，这些操作时间都应不超过 2 秒，且越短用户体验越好。

当然，2 秒并不是绝对的，对于一些用户感知明显的功能，例如垃圾扫描，病毒查杀，可能需要更多的时间，但是操作进行期间，需要给用户适当的感知和预期，避免用户因等待过久而离开。当然，用户是期望能够又准又快。

二、测试场景

性能要测哪些场景呢？用户实际使用中，场景是多种多样的。拿手机管家来说，有的用户每天喜欢拉拉小火箭；有的用户经常安装卸载 app；有的用户喜欢用它清理垃

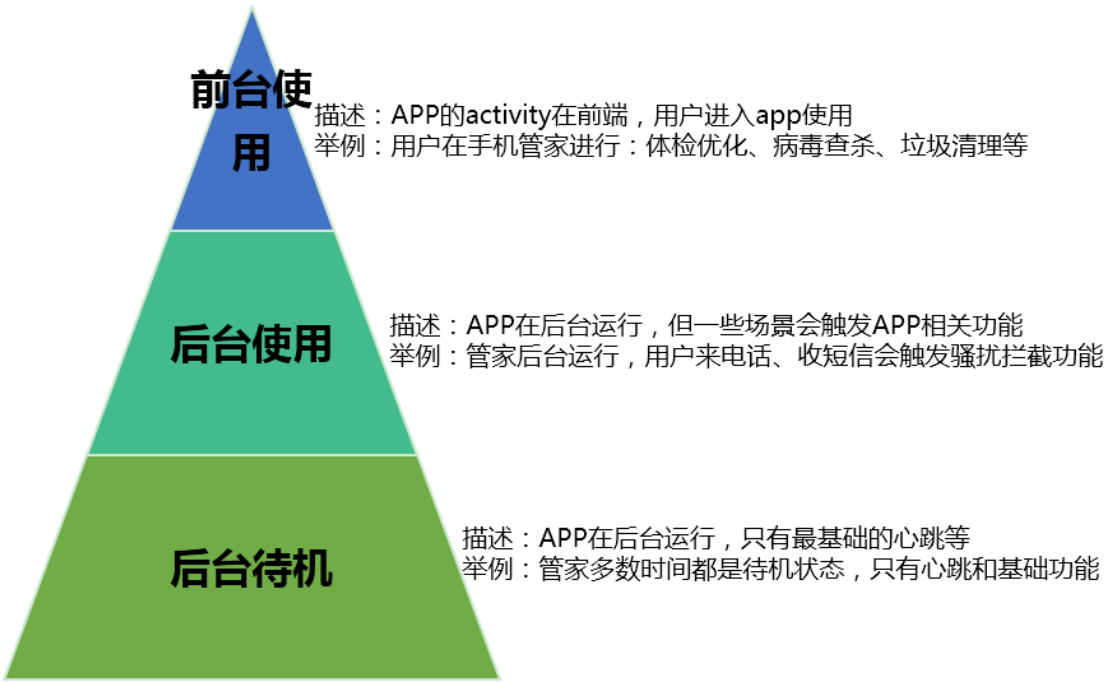
圾；又有的用户有很多骚扰电话和短信；还有用户喜欢用它连免费 wifi；另一些用户安装后不怎么使用。用户多种多样，功能多种多样，场景多种多样。要测试性能，这么多场景全部覆盖是不太可能的，要选择什么场景比较合适呢？

选择性能测试场景前，我们可以先将上述说的这些场景来分分类。

从用户使用 APP 时 APP 的 activity 是否在最前端，可将 APP 的使用场景分为：前台、后台。

在后台时根据 APP 只是保持心跳等最基础功能，还是一些场景触发了相关功能，可分为：后台待机和后台使用场景。

于是我们有了下面三个层次的场景：后台待机、后台使用、前台使用，场景模型如下：



1、场景与性能指标

先说说这些场景要关注些哪些性能指标。

(1) 后台待机

指标：重点关注待机内存和待机耗电情况。流量有需要则关注。

内存和耗电可取实际测试值作为数据。例如待机内存 19M；24h 待机电量 3mAh。如果你的 APP 的耗电模块主要是 CPU，你可以考虑使用 CPU 时间片（jiffies）来评

估。

测试时长：这个场景的耗电和 CPU 消耗会比较小，测试时需要考虑较长时间的测试数据以便反应 APP 的性能情况。

(2) 后台使用

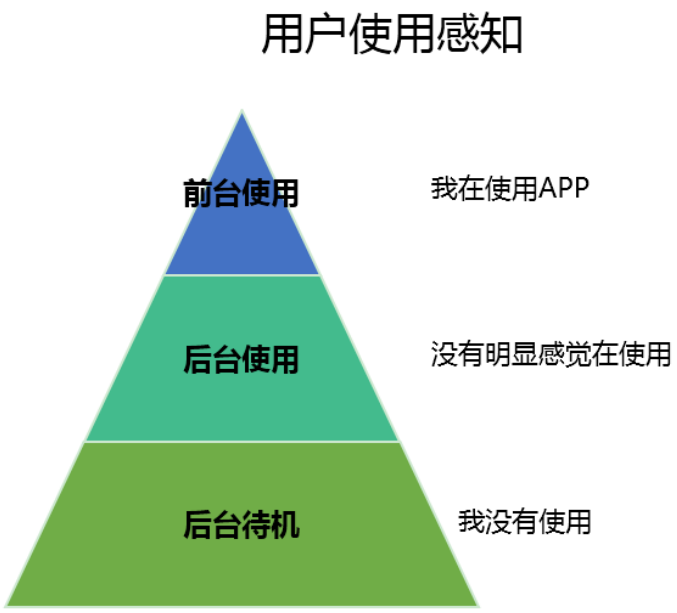
指标：重点关注一个场景触发 APP 功能时的内存、CPU 使用率或电量。

内存通常会取增量，即：触发功能后的内存-触发前的待机内存，作为某功能消耗的内存。

(3) 前台使用

常关注使用时的内存、CPU 使用率、流量。如果是某个操作需要一些时间，需要关注速度和耗时。

2、场景与用户感知



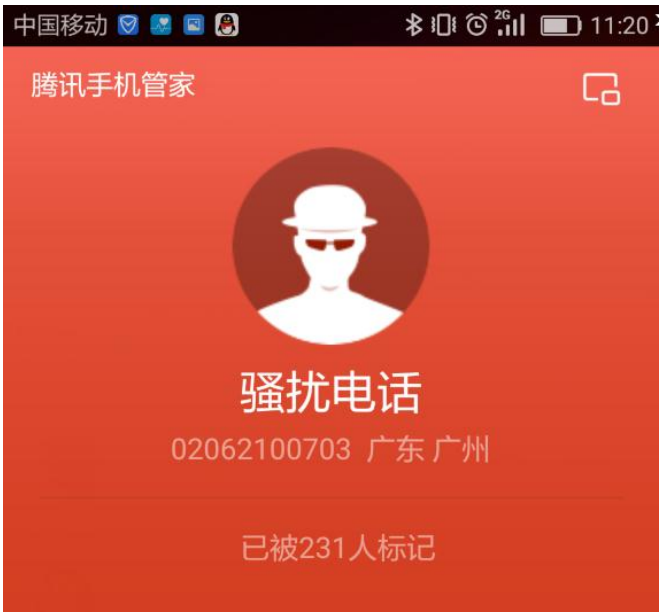
(1) 后台待机

这个场景下用户的感知是：我没有使用这个 APP，因此这种场景如果有性能消耗的情况下，一定是非常小的，否则用户会认为：我没用都占这么大内存！耗这么多电！让用户有这种感受是非常危险的。

例如：手机管家只在通知栏有个小图标，用户没有感觉自己在使用。

(2) 后台使用

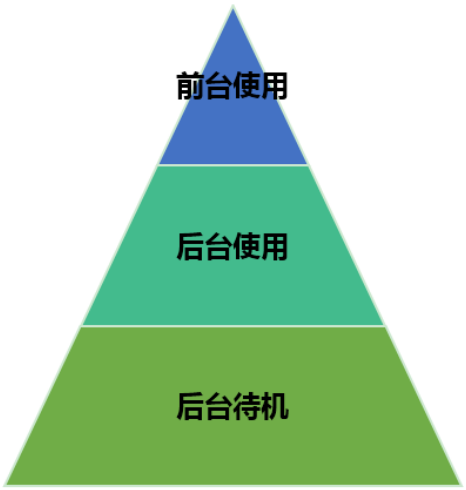
这个场景下 APP 确实进行了一些工作，但是用户对于自己使用了 APP 并不会有特别明显的感知。例如来电话时手机管家会进行电话的识别以判断是不是骚扰电话等，用户看到的是一个来去电悬浮窗，但是用户并没有主动使用，因此这种情况下性能消耗也不可以过高。



(3) 前台使用

这个场景是用户打开了 APP 进行使用，此时的性能消耗也是比较大的，但是用户的容忍度也会相对比较大。但是 OOM 导致闪退、手机发烫这些现象是绝对不允许的。性能消耗当然是越小用户越喜欢。

(4) 场景与测试优先级



场景分层图中，三个层次场景成金字塔形状，他们的占用面积同时反映了他们在用户侧使用时占用的时长。

这么多场景，时间有限，哪个场景更重要，我应该先测哪个呢？下面说说如何评估这些场景的重要程度和优先级。

原则：用户在该场景停留越久，该场景越重要；场景被用户使用到的频率越高，该场景优先级越高。**评估方法有：**

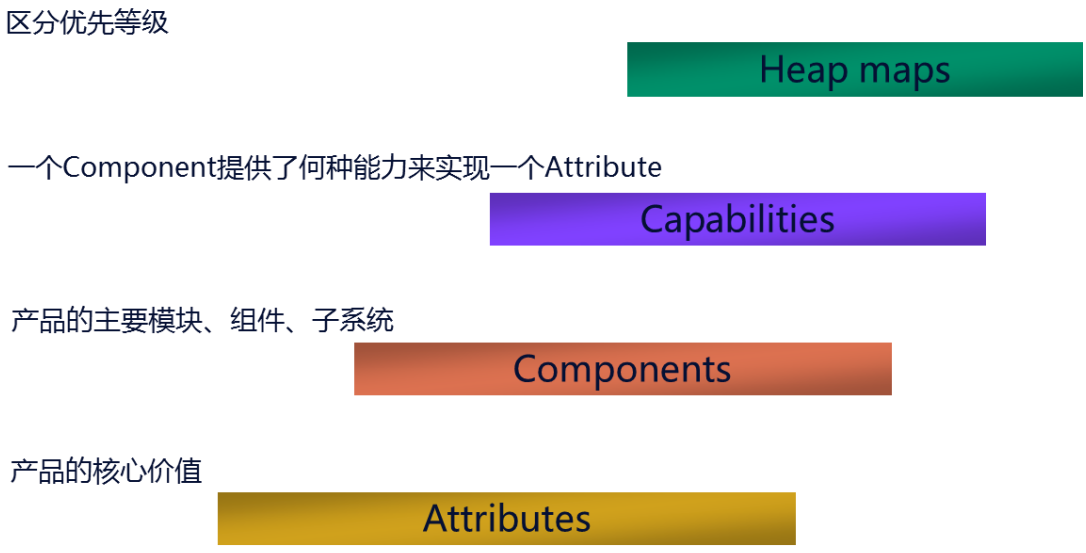
(1) 运营数据评估

对于后台待机场景，我们使用时长来评估优先级。重要程度=时长（数值越高越重要）。从运营数据中我们可以得到场景 1、2、3 在用户侧使用的时长 T1、T2、T3。例如用户平均每天亮屏 5 小时，灭屏 19 小时，那么灭屏待机场景重要程度=19，高于亮屏的 5。

对于后台使用和前台使用场景，我们用使用频率来评估，优先级=N，N=几天 1 次（数值越小优先级越高）。例如：收短信场景，这个场景每个用户每天都会遇到；而安装/卸载 APP 这个场景，用户可能平均 5 天操作一次。那么收短信场景的优先级=1，安装卸载 app 场景优先级=5。

(2) ACC 测试建模

当我们没有获得运营数据时，可以考虑使用 ACC 建模思路来帮助区分性能场景优先级：分析产品的核心价值；分析产品的主要模块、系统；FENIX 每个系统提供何种能力实现产品价值；区分优先级。



三、测试工具

测试工具的本质是获取性能数据，当然一些工具在使用和观察数据上有差别。工具分 2 类：现有工具，其他获取方案。这里列举下目前我们常用的工具共选择和参考：

性能测试	获取方法	优点	缺点
内存	APT（http://code.tencent.com/apt.html）	适合小白，有趋势图，直观	适合手动测试
	命令：dumpsys meminfo/top	有内存各项数据	
CPU	APT（http://code.tencent.com/apt.html）	适合小白，有趋势图，直观	jiffies只能看一段时间的累计增长
	命令：dumpsys cpuinfo/top	实时cpu使用率	
	读文件/proc/<pid>/stat	可用于jiffies的获取	
电量	电量仪：稳压器+示波器	真实电量	1.成本高 2.测的是整机电量
	PowerStats（http://gt.tencent.com/download.html）	数据分APP，分模块	需root
	命令：dumpsys batterystats	便于自动化	数据需自己解析
流量	Wireshark	简便	简便
	/proc/net/xt_qtaguid/stats	准，包含了tcp和udp的流量数据	4.0以上的系统
	TrafficStats类	方便，只包含tcp的流量数据	部分系统返回为0
	读文件/proc/uid_stat/	方便，只包含tcp的流量	一些系统（4.4以上）和机型没有该文件
速度	秒表		不够准确
	录屏分帧	贴近用户感受时间	难自动化
	Hook相关系统函数	准确，方便自动化	
	uiautomator 方法 clickAndWaitForNewWindow() waitForExists(long timeout)	方便自动化	时间小于2s时不够准确

手工和自动化测试

如果是新手或者只需测试几次，可考虑手工进行性能测试，建议选择方便直观易用的工具。测试内存和 CPU 使用率，推荐使用 APT，它最为 eclipse 插件，可实时监控 Android 手机上多个应用的 CPU、内存数据曲线，直观方便。

如果是需要长期测试的内容，需要考虑自动化测试。自动化测试方案我们常用 UIAutomator 来实现，其中获取性能数据的方案，可查看上表。

这里需要特别提下，在获取 CPU 时间片（jiffies）数据时需要注意，测试工具应该做尽量少的事情（不要同时用 dumpsys meminfo 获取内存会增加该进程的 CPU 消耗），减少对被测 app 性能的影响，选择性能消耗小的方式。建议：工具获取方式从系统文件 /proc/(pid)/stat 读取 CPU jiffies，不做其他测试内存等等事情。

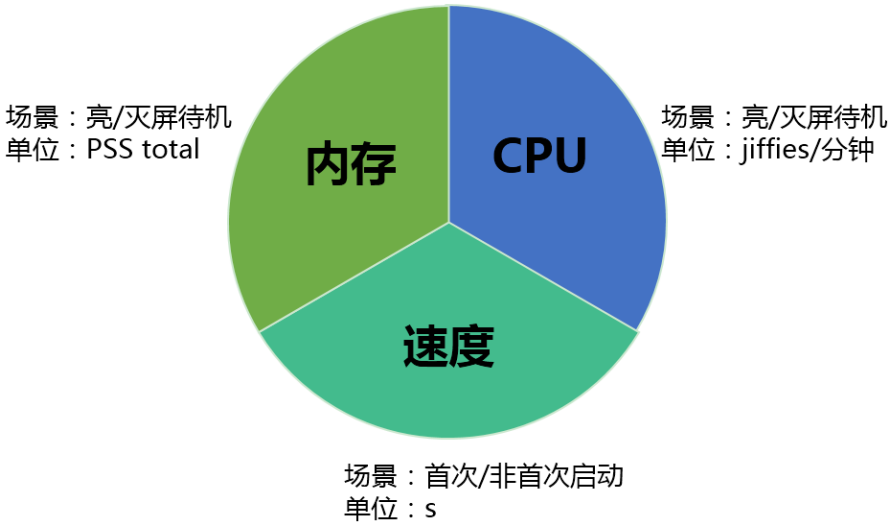
四、性能测试构建

到这里为止，对于 APP 性能测试的指标选择，场景选择，用什么工具有了一定了解。我们可以构建性能测试了。

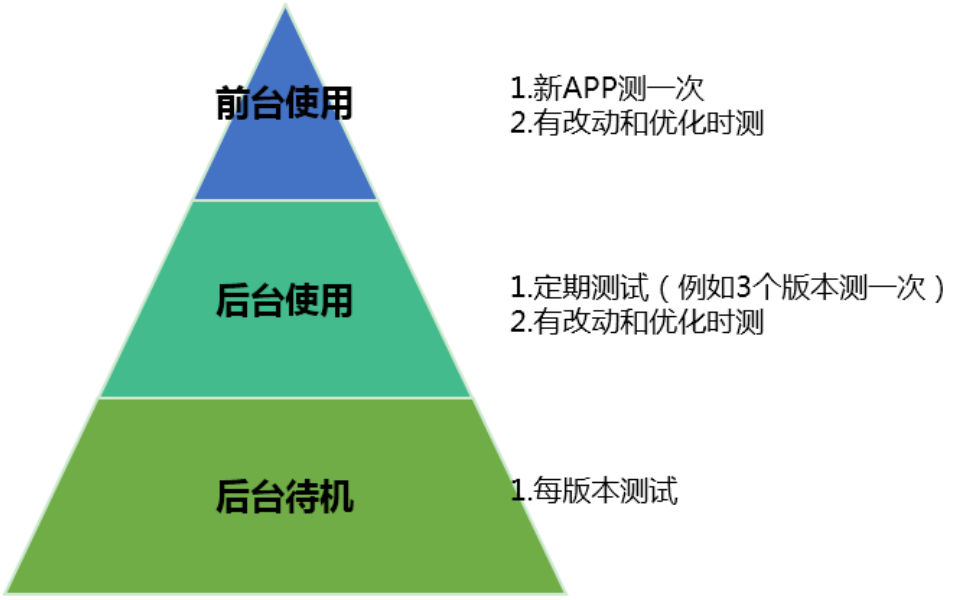
例如，手机管家需要长期关注一些重点性能指标，指标则选取了：内存和耗电，启动速度。由于用户在使用管家过程中，大部分时间都是处于“后台待机”场景，故我们选择测试的场景是：灭屏待机，亮屏待机。

内存使用 PSS total，耗电由于主要耗电模块是 CPU，因此我们选择使用 CPU 来评估耗电，待机 CPU 消耗小，故使用了 CPU 时间片 jiffies。

基础指标性能测试构建如下：



当然，除了这些基础指标，我们还需要测试其他场景，但可能不是每个版本都测。对于手机管家，三个层次的场景测试频率如下：



具体每个场景的分析，测试频率参考：

测试频率	测试周期	场景选择
高	每版本测试	最核心和基础场景
中	定期测试（几个版本测一次）	重要但优先级不那么高的场景
低	有改动时测试	各场景

最后，测试数据如果是单次、单个是没意义的，我们通常用两种方法做对比：历史版本对比、竞品对比。

当你要着手给 APP 做性能测试时，记得分析 APP 性能测试的三要素：性能指标、测试场景、测试工具，体系化构建你的性能测试任务，让 APP 性能保持优秀，运行更顺畅。