# 专项测试

在进行了手工的功能测试，也开发了一些自动化的测试用例，并且做了性能测试之后，测试工作看似比较完整了，但是当我们的APP在大量的用户那里被安装和使用的时候，还是会有很多我们之前没有预料到的问题，比如：

1. Crash的问题
2. 设备兼容性的问题
3. 流量使用过多的问题
4. APP导致用户手机电量消耗过快的问题
5. 在不同网络情况下不稳定，比如卡死和白屏的问题

## 兼容性测试测试面

### 操作系统

1.ios8及以上

2.Android4.4-8.0.

如果逐个去覆盖，工作量太大，投入产出比太低。除非有明确的直接影响APP的特性变动，否则不会逐个去考虑每个小版本

### 屏幕分辨率

主流机型的分辨率：800\*480,960\*640,1280\*720,1920\*1080,2k。针对iOS，相对简单一些，可以主要考虑最近几代机型对应的分辨率。

分辨率的兼容性是一个非常容易遇到的 问题，如果代码没有对不同的分辨率做适配处理，就会出现错位、遮挡、留白、拉伸和模糊等各种问题。一方面需要测试去实际验证，另一方面从设计和代码（比如使用相对布局）层面就需要做考虑。

### 不同厂家的ROM

这个主要是Android系统碎片化引起的问题，几乎每个Android手机厂商对Android系统进行了或深或浅的定制。实际中我们也确实遇到一些不同厂家ROM导致的问题，比如调用相机和一些底层服务出现的不兼容，以及“摇一摇”之类的功能遇到不同手机对于方向和重力传感器灵敏度设置出现不同的问题。

### 网络类型

这个问题设计APP中对不同网络的策略，以及对于不同网络的带宽、延迟和稳定性的处理。目前我们通常会考虑WiFi、2G、3G、4G下的功能情况。

### 针对手机M版网站：

1. 主要考虑不同的浏览器类型，包括主流厂商的手机上自带的浏览器，以及主流的第三方浏览器。
2. 另外需要考虑的就是屏幕分辨率的问题

小节：

通常我们会选择少数主流设备上执行全量的测试用例，在其他兼容性范围内的设备上覆盖主要功能的用例。项目执行过程中为了更好的跟踪和了解进展，可以通过一个表格的形式来维护不同功能点的兼容性情况。在bug管理方面，可以增加对应的测试选项，在测试人员提交bug的时候记录，以便后面进行统计和经验的总结。

### 基于UI的云平台兼容性测试

方法的局限性:

1. 很多测试团队不一定有完备的所有类型的设备，特别是对外部反馈的兼容性问题，立即去草狗周期较长，而且可能使用率也会很低
2. 为了覆盖不同维度的兼容性，需要测试人员手工在多台设备上执行重复的用例，效率比较低下。
3. 在不同的设备上发现的问题需要手工截图和记录日志，也是一个比较耗时的工作

兼容性测试方法思路：

1. 如果是针对上述兼容性问题，需要将一些比较基本的UI操作步骤在不同的手机上多次反复操作。工作量随着测试的机型的数量线性增长
2. 通常我们手头的测试机比较有限，目前一些较大的APP云测试平台提供了几千台真机供使用。

**UI自动化测试基本知识**

总的来说，这个方法的好处就是可以基于轻量级的UI自动化脚本，以及借助云端平台大量的针剂资源，讲一个流程在多台机器上执行，极大的节省了多平台覆盖的重复人力操作。

## 流量测试：

移动互联网产品的一个好处就是几乎可以随时随地使用，这给用户带来了很大的便利，但是同样也会带来一个问题，在目前以及未来的一段时间内，移动网络的带宽还比较有限，流量也是要付费的，这使得用户会关心它使用的APP使用了多少流量，因为这直接关系到用户的资费账单。一些不好的APP设计，或者缺陷，比如频繁的在后台联网去服务端获取信息，可能带来意料之外的流量消耗。这会直接伤害用户，导致用户卸载APP，或者引起投诉。另外流量的减少不只是减少用户的流量消耗，通常也会因为更少的网络传输带来更好的性能和响应。

流量可以从用户使用的相关性角度进行分类，一类是用户的操作直接导致的流量消耗，另一类是后台，既在用户没有直接使用情况下的流量消耗。后一种情况对于Android用户更加容易出现。目前Android的消息推送机制不是借助统一的管理，而是各个APP定时启动后台进程到自己的服务端去询问是否有新的消息，有的话就拉取到客户端，而这个询问的过程本身就会带来流量的消耗，当然也会带来额外的电量消耗。

### Android流量测试

#### Android APP基于系统自带的统计功能

这个也是目前最简单的方法，直接读取Android系统上两个文件的内容：

1. proc/uid\_stat/(UID)/tcp\_and
2. proc/uid\_stat/(UID)/tcp\_rev

其中，UID是每个Android APP在安装时分配的一个唯一编号，用于识别该APP。tcp\_snd文件中的数据表示发送的数据累计大小，以字节为单位，tcp\_rcv表示接收到的数据累计大小。

#### Android APP通过API来获取流量数据

使用TrafficStats类提供了相应的多个方法来获取不同角度的流量数据。通过TrafficStats.getUidRxBytes方法获取该App对应的接收流量数据，然后显示在日志里面。

#### APP内部通过代码统计接口数据量

使用代码，resp是一个HTTPResponse对象，resp.getEntity().-----------------getContentLength()方法可以获得本次响应的数据量。

### iOS流量测试方法

#### 通过Instruments自带的Network来查看网络流量

#### 代码方式实时统计流量

### 通用的流量测试方法

#### 手机上抓包

常用的抓包工具有Windows下的Wireshark工具和linux下的tcpdump.

#### 基于WiFi的方式来获取流量数据

在PC机开机Fiddler代理工具，就可以看到手机上发出的请求和收到的响应，Fiddler界面包含了每个请求和响应的流量大小，可以直接统计或者复制到Excel等软件中来计算。

以上方法可以获得每个接口的数据流量，需要指出的是，这种方式会少算一些底层的协议层header的流量。和抓包结果的饿对比分析，粗略估算下来，大概会少算10%左右，当然，这个也取决于平均包的大小，如果普遍都是小包，那么协议header的占比就会比较高。

#### 自动化的流量统计方案

### 常见的流量节省的方法

#### 数据的压缩

减少传输的数据量是一个最基本的节省流量的方法，在尽量不影响功能和体验的情况下，压缩是一个能直接减少流量的方法。压缩包含接口文本数据的压缩、js文件的压缩图片的压缩，基于一些压缩算法，可以在JPG、PNG等文件格式上的基础上进一步降低图片大小而图片质量不会明显下降。

#### **不同数据格式的采用**

在传输信息相同的情况下，采用更精简的文件格式也是一个常用的减少流量的方法。比如采用JSON格式作为接口数据返回格式通常比XML文件要小。另外在图片方面，近来开始比较广泛使用的WebP格式也是一个节省流量的办法。

#### **控制访问的频次**

减少流量消耗的另一个角度是减少访问的频次，这个主要是针对和后台的数据上报，PUSH消息检测等定时机制

#### **只获取必要的数据**

很多时候，我们会遇到APP一页的内容比较多的情况，而用户可能只会查看一部分，过多的从后台拉取数据就是一种浪费，所以可以采用分屏加载或者懒加载的方式来减少流量消耗。

#### **缓存**

缓存也是一个非常常用且非常有效的方法，做法和浏览器的缓存类似，可以将一些图片、js等之前访问过的数据暂时缓存起来，等以后使用到相关功能的时候就不用再去拉取相关的数据。同样，也需要控制缓存的有效期和更新策略。另外，由于手机的存储空间有限，通常也需要控制整个缓存的大小，并给用户提供清理缓存的选项。

#### **针对不同网络类型设计不同的访问策略**

目前来看，主要是用户对于在移动网络下的流量消耗比较敏感而在WiFi下，流量和带宽都不是问题，这是应该以更好的用户体验为导向。针对这两种情况下需求的差异，可以通过不同的策略来控制，通过判断当前的网络状况，控制数据访问的频率、预加载策略和图片的质量**。**

## 电量测试

### Android电量测试方法

一种是基于硬件的测试方案，可以比较准确的测试手机的电量消耗情况。第二种方法是借助一个第三方的APP来评估手机上各个APP的电量消耗。

基于GSam Battery Monitor Pro 查看电量消耗

电量是一个物理的概念，其消耗的最后对应的都是物理器件，包括CPU、内存、显示屏、网络、电话射频、存储设备、相机以及各种传感器。

唤醒锁：设计两个Android概念。一个是AlarmManager，另一个是WakeLock.

AlarmManager是在Android系统中主要用来定时处理一个事件，比如闹钟应用就是使用AlarmManager来实现的。另外，很多APP都带有消息服务或者定时和后台服务同步信息的功能，在Android上这通常采用一个服务定时向服务器发起请求来实现的。从流量角度看，这也是手机为什么在锁屏没有操作的情况下仍然会不断产生流量的原因。从电量的角度，这也会带来一定的消耗。当然对于定期执行任务也可以用Timer和TimerTask来实现，或者开一个Service在Thread里面以while循环来实现。但是那样对系统的消耗会更大，更好的方法还是选用AlarmManager。

在我们使用一些APP，比如微信、QQ的时候，如果有新消息到达，手机即使在锁屏状态下也会亮起并提示声音。这样的功能非常有必要，从技术角度需要有机制将对应的处于后台的APP唤醒。wakeLock就是这样一种机制。

各种锁的类型对CPU、屏幕、键盘的影响如下：

PARTIAL\_WAKE\_LOCK：保持CPU运转，屏幕和键盘灯有可能是关闭的。

Screen\_DIM\_WAKE\_LOCK：保持CPU运转，允许保持屏幕高亮显示但有可能是灰的，允许关闭键盘灯。

SCREEN\_BRIGHT\_WAKE\_LOCK:保持CPU运转，允许保持屏幕高亮显示，允许关闭键盘灯。

FULL\_WAKE\_LOCK：保持CPU运转，保持屏幕高亮显示，键盘灯也保持亮度。

ACQUIRE\_CAUSES\_WAKEUP：强制屏幕亮起，只要是针对一些必须通知用户的操作。

ON\_AFTER\_RELEASE：当锁被释放时，保持屏幕亮起一段时间

如果一个APP有比较多的后台唤醒，特别是不合理的频次和锁的设置，那么将导致很大的电量消耗。就目前情况，Android平台对用用并无严格的审核，就导致一些在这方面处理不好的应用成为耗电大户，对专项测试而言，我们需要验证我们的应用是否有这方面的问题。

## 弱网络测试

借助手机自带的网络状况模拟工具

基于代理的弱网络的模拟

Windows下的NetWork Delay Simulator

通过Flow Match Condition部分的设定可以选择对哪些网络连接进行控制，包括本地和远程的IP段、协议类型和端口，这样可以针对要测试的链路模拟网络状况，同时不影响电脑和手机上的其他软件正常使用。

可以模拟的网络状况包括双向的带宽、网络延迟、丢包率等维度。配置好对应的参数，然后选择Save Flow就开始生效了。

接下来我们就开始用手机打开被测得APP，进行弱网络情况下的测试。如果想验证弱网络的条件是否生效，一方面可以从APP的响应情况看出差别。另一方面，我们也可以在PC/MAC上抓包来看看具体的的情况。

Mac 下单的Network Link Conditioner

手机通过流量测试的方式，代理到MAC上，同时打开代理和网络状况模拟程序。这里是charles的Network Link Conditioner。

小节：

在网络较差的情况下，显示速度等体验方面肯定会有损失，这是无可避免的，但是也可以观察APP实际行为，比如部分显示，或者提示加载速度等，也可以从产品和用户体验的角度来观察这种情况是否可以接受，这个方面没有一个统一的标准，可能需要结合实际项目的情况，由产品经理和研发人员一起来判断。如果觉得体验不可接受，再进一步看是否在数据获取或者界面渲染的逻辑上做改进。

以上工具可以帮助我们模拟弱网络的情况，实际上，我们通过这样的模拟发现了很多APP层面的问题，包括对异常处理不当导致的应用崩溃，以及各种用户体验方面的问题。工具本身使用的贷记不大，但是发现的问题都非常有价值。所以这样的测试还是非常有必要的。

## 稳定性测试

### 基于monkey的稳定性测试

Monkey命令随机的向目标程序发送各种键盘事件流，并且可以自定义发送的次数，来观察被测应用的稳定性和可靠性。Monkey工具使用起来比较简单，只需要一行命令就可以开始执行。

Monkey模拟的时间非常广泛，除了对于被测APP的点击、滑动、键盘输入等常见的操作之外，他还大量模拟了各种手机系统的操作，比如调整音量、打开通知栏、改变网络状态、锁屏等操作。

Monkey测试停止条件主要有下面几种：

1. 执行的次数到了
2. 如果限定了Monkey运行在一个或者几个特定的包上，当监测到试图转到其他报的操作。会对其进行阻止。
3. 如果应用程序崩溃或者接收到任何失控的异常，Monkey将停止并报错
4. 如果应用程序产生了ANR（应用程序不响应）的错误，Monkey将停止并报错

MonkeyRunner是一个工具包，提供了一些API，可以在Android代码之外控制Android设备和模拟器。相比Monkey不可控的模拟事件，MonkeyRunner可以通过自己的代码来启动APP，控制发送模拟的点击操作，并对模拟器进行操作。

### Android的ANR

Android系统的ANR是一个特别容易遇到的稳定性问题。当一个Android APP遇到以下两种情况之一时，会产生Application Responding，也就是我们常说的ANR:

1. 输入事件5秒内未响应完成，或点击屏幕、按键等操作。
2. BroadcastReceiver10秒内未执行完成

常见的造成ANR的原因之一就是在主线程执行了耗时过长的操作。发生ANR时，Android系统会弹出一个应用程序无响应的对话框，用户可以选择等待或者强制终止该APP。

## 安全测试

### 安装包测试

#### 能否反编译代码

在测试中，我们可以直接使用反编译工具查看源代码，看是否进行了代码混淆，是否包括了显而易见的敏感信息，等等。例如，在Android测试中，常用的反编译方法是使用dex2jar工具并结合jd-gui工具查看源代码。

#### 安装包是否签名

这一点iOS平台可能不必考虑，因为iOS的每一个APP都有正式的发布证书来签名。当发布到APP store时，APP store都会做校验，保证该APP是合法开发者发布的。对于Android来说，由于发布渠道多种多样，没有此类权威检查，我们需要在发布前校验一下签名使用的key是否正确，以防被恶意第三方应用覆盖安装的问题。

jarsigner -verify -verbose -certs apk包路径

如果运行后结果为“jar包已验证”，说明签名校验成功

### 完整性校验

为确保安装包不会在测试完成到最终交付过程中因为各种问题发生文件破坏、需要对安装包进行完整性校验。通常做法是检查文件的MD5值，而且一般可以通过自动化做校验。

#### 权限设置检查

对于没有必要的权限，一般建议开发者直接去除。Android平台上我们可以直接检查manifest文件来读取应用所需的全部权限，并结合需求主意校验此权限是否为必须的。另外，对于mainfest文件的修改也需要关注。在增加新权限前需要评估。

在iOS平台上没有类似的manifest文件来查看APP的用户权限，iOS的APP用户权限管理只有在用户使用APP到了需要使用的权限时，系统才会弹出提示框，提示用户当前APP需要访问照片 ，或者联系人列表等类似的公共组件，用户可以选择接受或者拒绝，我们可以扫描代码来查看项目工程中有哪些权限设置。通过搜索关键类名，如通讯录一般需要访问ABAddressBookRef；照片操作一般需要访问UIImagePickerController等，我们可以判断程序中是否有相应权限的调用。如果是纯黑盒测试，则必须覆盖到所有代码路径才能保证没有遗漏，也可使用代码覆盖率测试判断是否覆盖。

### 敏感信息测试

1. 数据库中是否存储敏感信息
2. 日志中是否存储敏感信息
3. 配置文件是否存在敏感信息

### 软键盘劫持

如果用户安装了第三方软键盘，那么用户在使用我们的应用进行一些敏感信息输入的时候，一旦使用第三方软键盘输入，输入内容就能被第三方软键盘截获。如果该第三方软键盘含有恶意代码，可能引起用户数据被盗取并造成用户损失。一般用户并不清楚是由于这个软键盘导致的，而会认为是我们的应用把他的敏感信息泄露出去的。对此我们在一些特别敏感的需要输入的地方可以做检查，例如金融类APP登录界面的用户名和密码输入框等，看是否支持第三方输入法。对于非常敏感的输入，一般建议使用应用内的软键盘，或者至少提供用户这一选项，这样能避免可能的键盘劫持的风险。

### 账户安全

1. 密码是否明文存储在后台数据库
2. 密码传输是否加密。测试中我们需要查看密码是否被明文传输。如果是HTTP接口我们可以使用Fiddler等工具直接查看。
3. 账户锁定策略。对于用户输入密码次数过多的情况，一些应用会将账户临时锁定。这是对用户账户安全的一种保护措施，一般是推荐使用的。
4. 同时会话。一些应用对同时会话会有通知功能。这样至少可以让用户知道它的账户可能已经被泄露了，在一定程度上能够提升用户体验。
5. 注销机制。在客户端注销后，我们需要验证任何的来自该用户的，需要身份验证的接口调用都不能成功。

### 数据通信安全

1. 关键数据是否散列或加密
2. 关键连接是否使用安全通信
3. 是否对数字证书合法性进行验证
4. 是否校验数据的合法性

### 组件安全测试

### 服务端接口测试

1. SQL注入
2. XSS跨站脚本攻击
3. CSRF跨站请求伪造
4. 越权访问

## 环境相关测试

### 干扰测试

1. 收到电话
2. 收到短信
3. 收到通知栏信息
4. 无电提示框弹出
5. 第三方安全告警弹出

### 权限测试

### 边界情况

1. 可用存储空间过少
2. 没有SD卡/双SD卡
3. 飞行模式
4. 系统时间有误（晚于或早于标准时间）
5. 第三方依赖

### Android定位测试

1. 白盒测试。由于定位代码最终获取的是一个位置对象，我们只需要在获取位置对象后手动设置经纬度即可进行测试。
2. 模拟器模拟。使用模拟器进行测试的时候可以使用DDMS进行经纬度设置。
3. 自定义位置提供器。