移动 App 测试实战

顶级互联网企业软件测试和质量提升最佳实践

邱　鹏　陈　吉　潘晓明　著

机械工业出版社
China Machine Press

当邱鹏委托我为本书写推荐的时候，我也倍感压力，作为在测试及质量领域摸爬滚打十几年的老兵，深知这一领域的艰深与繁杂，也深深地知道在这个领域里，我们走得太快而沉淀太少；尤其在移动互联网领域，行业与业务一路飞奔，而相应的质量控制体系、方法和工具却远远没有跟上。邱鹏、陈吉与潘晓明合著的此书正好补缺了这一领域的空白。本书基本上涵盖了移动互联网产品测试的方方面面，从入门的功能测试、自动化到相对专项的弱网络、安全、稳定性等，都有所涉及。翻完此书，最大的感触是：实用！本书不会介绍什么“高大上”的概念，而是选择了很多非常实在的测试问题，给出了非常具体、可以实操的方案；而这些方案并不一定用了什么新颖的、昂贵的工具平台，很多的工具平台也许是大家耳熟能详的，稍加改造与适配就非常适合移动的场景。

总之，我相信，无论是测试的老兵，还是新人，你都可以从本书中获益良多。 对于测试的老兵而言，建议将本书看作工具手册，遇到某一领域的问题时，翻开看看，总能找到你想要的方案、工具。 而作为新人，自然建议通读一遍，对于某些领域的测试问题，也许暂时不会遇到，也许尚不能完全理解，也没关系，放于枕边，日后遇到时再翻出来看看，定会又有深一层的理解。

——李俊，蚂蚁金服资深专家

移动互联网高潮迭起，打造上千万乃至上亿用户的 APP 是每个开发者的梦想，Testin 作为全球最大的移动 APP 测试领域的专业厂家，我们深知每一个成功 APP 的背后都有一支强大的测试队伍，本书作者邱鹏成功领导过上亿用户 APP 的测试，他的实战经验对每一个渴望成功的开发者来说都是不可多得的宝贵经验。通过这本书，他把经历过的一切毫无保留地奉献给了我们，为我们揭开了移动 APP 测试的神秘面纱，其中的不少方法和经验，也是 Testin 一直以来所采用的。本书的精彩样例和讲解，由浅入深，无论是初学者还是专业测试人员都能毫无障碍地学习。

——戴亦斌，Testin 云测联合创始人 &COO

任何好的移动 App 其实都依赖着强大的后台服务支撑能力，Ricky 的书并不仅是针对移动客户端的测试实践的，其中还包括了网络协议、接口层、后台服务性能等全面的测试内容，也借此引出了行业内主流的互联网测试开展方法，涉及面虽广但因结合了具体实践案例而能很好地展示互联网测试的特点，在分层、精准测试的指导思路下，测试工程师既能如庖丁解牛般对项目中技术细节深入了解，又能兼顾到快速迭代发布的质量把关。因此本书对移动互联网测试工程师、在测试领域中努力探索的其他同行们以及移动互联网创业大潮中的团队都有着很强的参考学习意义，强烈推荐给大家！

——万巍，腾讯社交网络测试总监

在互联网高速发展的今天，测试作为技术研发流程中最重要的环节之一，其地位是不言而喻的；然而作为一个测试从业者应该具备什么样的技术能力、怎样快速地对互联网测试有一个全面的认识和学习、自己的职业怎样规划才是最合理的等问题，是每个测试从业者必须面对和考虑的，很多测试从业者由于经验比较少和缺乏一些专业人士的指导和建议，往往会走很多弯路。本书作者在腾讯、京东、阿里等大型互联网公司有多年丰富的测试研发和测试管理经验；结合大量的实战经验，对移动互联网测试在功能测试、自动化测试、专项测试、QA 等方面进行了全面而详细的讲解；能够帮助测试从业者快速和系统地了解和学习移动互联网测试架构。相信本书的内容无论对初级测试员、资深测试员还是测试管理者都有一些很好的启发和帮助。

——付学宝，京东无线开发总监

推荐产品经理阅读本书，书中有大量的实例帮助产品经理更全面地理解用户体验。用户不会愿意使用这样的产品：用十分钟就会让手机发热很厉害，或者在未知的时候被使用了大量的流量，或者在 2G 网络上完全打不开……本书还介绍了如何搭建自动化测试平台，帮助产品研发团队快速迭代产品，这在移动互联网时代尤为重要。

——Nikita Peng，携程机票产品总监

本书结合邱鹏先生多年的测试工作经历及对行业知识的理解，深入浅出地揭秘互联网测试。书中总结了通用的方法论、工具创新，并结合大量的实战项目案例分析，可以很好地帮助读者归纳总结。本书同样适合测试以外的互联网从业者学习，也有助于研发体系内其他的角色更好地理解测试，促进更多的换位思考和合作共赢，衷心推荐大家阅读学习。

——徐奇琛，京东无线运维总监

本书几位作者是我以前在腾讯、京东的同事，他们有非常丰富的测试经验。目前移动端是所有行业的未来发展方向，本书非常好，是通过丰富的项目实践摸索后的经验总结，是极有价值的宝贵资源，如果你对移动 App 测试和质量提升感兴趣的话，本书极具参考价值。

——朱永敏，口袋理财 CEO

本书从京东 / 腾讯的实际项目出发，系统介绍了 iOS 和 Android App 的整体质量保证实践，及全生命周期的方法、工具，并详细阐述了关键实现原理及相关代码，对开阔实际测试工作思路有非常大的帮助。作者们有几年的 APP 测试经验，以及在阿里巴巴技术大会等做技术分享的经验，有系统的总结梳理，本书值得互联网技术工程师参考。

——梁剑钊，天猫高级技术专家

做为一股汹涌而来的行业潮流，移动互联网在可以预见的未来仍然是一种不可阻挡的发展趋势。而针对移动互联网产品的测试一直是很多软件质量从业人员的困扰。本书构建了移动产品测试的整体框架，在一些有移动特性的环节进行了详细的阐述，比如电量、流量、弱网络等，同时辅以大量的实践案例。是一本实用性很强的好书！

——陈世宏，途牛旅游网无线中心总经理

对于移动互联网行业的创业者而言，更好的产品质量意味着更高的用户留存转化率，更少的用户投诉，从而实现更低的推广成本，最终在竞争激烈的市场中赢得先机。但是，大多数移动互联网创业团队在创业初期，并不具备完整的测试团队和质量管理能力；即使随着业务的发展，产品质量需求提上日程后，搭建质量管理体系的问题依然困扰着创业者。本书能够解决创业者对于质量管理体系的所有困扰：涵盖了从测试团队建设、测试流程与生命周期管理、常见测试工具与测试技术介绍等内容，面向 Android 与 iOS 两大主流移动 OS；无论测试的是客户端产品还是 Web 产品，均能够从中获益。本书特别针对移动互联网的使用场景特点，重点介绍了专项测试内容，能够发现传统测试无法覆盖的使用场景，可有效提升产品质量。同时，本书包含了丰富的案例与实用的代码供读者参考使用，是一本兼顾了理论性与实用性的好书！

——张勇，LBE 安全大师、创始人 &CEO

非常荣幸能成为本书的早期读者之一，我读完之后最深刻的感受就是：这正是我所经历过的，也正是我想总结给自己团队的内容。这是一本非常接地气的书，每一个章节阐述的方法几乎都可以在读完之后马上在项目中进行实际应用，在这本书上我看到很多我们在微信测试时曾经使用或仍在使用的方案和方法。

——夏凡，微信业务测试负责人

本书没有高大上的浮夸内容，都是在测试过程中针对一个个特定场景的很好的解决方案，更是作者从事测试行业多年宝贵经验的总结，很多你在工作中不知道如何去做的时候，可能从本书中找到答案，不管你是测试管理者还是执行者，书中的一些思路和方法，都值得去研读、借鉴。

——王胜，百度高级测试开发工程师

序　言 *Preface*

收到写序邀请时，我毫不犹豫答应了，因为终于有同仁开始把针对移动互联网测试领域的实践体系化整理了出来，然后分享给大家。另外很开心的是，Ricky 这个家伙是我的好友和同事，不少的实践和方案一定层面也代表了我们团队过去的真实测试管理和实践，要特别谢谢 Ricky，帮助实现了我们的愿望！

几年时间内，移动互联网席卷几乎所有行业和领域，移动开发技术也得到前所未有的快速发展。测试团队如何在移动互联网时代快速适应和应对相信是很多公司非常重视的事情，我们应该采取和之前哪些不一样的测试方法，应该保留哪些方面的技术，移动互联网领域的测试技术发展方向和各维度挑战又是什么样的？目前还没看到非常有代表和指引性的整体测试实践书籍，这是让我感到有点缺憾的地方，不同行业领域都各自有自己的核心技术和竞争力，移动互联网领域的测试技术不应该落后和滞后。通读了 Ricky 的整本书，让我很开心地看到体系化的移动互联网领域测试技术终于有一个很清晰的面貌放在了读者面前，也许这里的很多技术 / 工具方案并不代表是最好的，但在这本书中衍射出来各个维度 / 领域的测试实践和思路，对正在从事移动互联网测试的同行是一个极大的帮助和指路明灯。如果您也正从事移动领域的测试，强烈推荐您仔细用心阅读这本书。这本书可以按兴趣分章节来阅读，获取自己感兴趣的一部分技术，当然对于一些测试管理者，我也强烈推荐通读每个章节，在通读的过程中，同时建议思考这里的测试解决方案体系建设，同时完善自己的测试体系和思想。

借这本书，我也想简单分享一下我对国内测试行业现状的看法。过去近 20 年里，有些比较悲哀地看到大学教育仍然停滞在单纯的理论介绍上（这些理论大多都过时了），有些大学老师甚至对 IT 公司的测试工作没啥了解；而大部分企业里的测试团队，也相对比较落后，仍然基本聚焦在纯黑盒 / 功能测试上，在如何更好地提升测试效率以及深层次地提升测试

质量等方面很少开展；大批的同行可能在抱怨公司对测试岗位不重视的同时，很少去思考测试的核心竞争力到底在哪里，很少去分析测试的发展是否进入误区或要怎么改进。作为一名测试人，我一直想对其他同行呼吁和呐喊，希望不要糟蹋了自己良好的大学专业知识，不要让自己工作几年后因所开展的工作而缩小了自己的职业空间。但行业里需要更多志同道合的同仁一起来做这件事情，才能把测试领域的正确发展方向广泛传播。对于一名刚从事测试行业的同仁来说，我强烈推荐从下面几个领域来完善提升自己：1）针对开发语言或脚本语言的深度掌握和熟练使用；2）锻炼和提升自己的测试分析设计和评估能力，并不断完善自己的测试体系和思想；3）对产品的相关开发技术和设计架构，甚至深入到代码实现角度的深层次掌握和理解；4）坚实的自动化测试理解以及实践积累；5）对操作系统、网络等基础知识更深入的掌握和实践；6）保持对测试行业新技术的不断探索和对齐。 这些方面的能力，我们要在配合工作实践开展情况下，夯实，做深，做专，这是工作前 5 年里特别关键重要的沉淀，会直接影响自己未来 10 年甚至更久的职业发展空间。对于逐步走上管理岗位的测试同仁来说，我一直推崇技术管理的定位，直接说就是技术加管理两条腿都要继续保持，没有了持续的技术提升意识，自身的未来竞争力以及无法给团队很及时精准技术辅导的弊端会逐步显现出来，一旦到了不惑之年才醒悟，那才是真正的悲哀，后悔都来不及了。对于一名测试管理者，我也特别推荐从下面几个领域来丰富完善自己：1）建立自己清晰完善的测试解决方案体系和思想，配合工作管理，不断实施打磨，梳理完善自我的测试知识体系，培养出自己的一套测试解决方案体系和思想，如同我们讲的古人要悟出自己的一个道来一样；2）对质量和效率提升如何更加清晰的平衡和把关能力；3）完善和建设清晰的测试度量体系；4）关注和推动自动化测试，同时关注投资回报率（ROI）。

额，好吧，好像开始讲我自己的测试“大道”了，跑题了。最后回来再看这本书，我尤其希望读者能理解这本书所传递出来的思路和思想，然后您就清楚了 Ricky 的良苦用心，您也就正式悟道了！

最后送所有测试同仁的一句话仍就是：技术决定未来、没有技术没有未来！

吴凯华

腾讯公司社交网络质量部副总经理

2015-04-03

前　言 *Preface*

现在已经是移动互联网的时代，借助手机等移动设备，人们可以完成资讯的获取、社交、游戏，以及日常生活的各种应用，甚至很多工作的开展。有很多新兴的移动互联网公司在崛起，也有很多传统的 IT 公司在转型，更有大量传统行业的企业在借助移动互联网拓展自己的业务。对 IT 技术人员而言，这是一个非常好的时代，有大量的工作机会，因为有大量的移动互联网相关系统的研发需求。当然，这也意味着有很多新的技术和方法要去学习。有很多的研发人员快速转型到移动互联网领域，有大量的移动互联网产品被开发出来。在这个过程中，也会面临一个问题，那就是产品质量的参差不齐。在某种程度上，因为移动设备的特点，比如屏幕相比 PC 较小、电量有限、移动网络状况复杂，以及设备性能的问题，移动互联网产品对质量的要求其实更高。有过相关研发经验的人应该能体会，快速开发一个可用的移动 App 并不难，但是做一个高质量的 App 其实是一件非常有挑战的事情。

这本书写作的初衷就是希望给移动互联网产品的研发团队，包括测试团队，一个基于大量一线实践的比较系统性的参考。

我们毕业后工作的几家公司都比较重视产品质量，对测试的投入都比较大，对人员的要求也比较高，使得我们有机会比较系统地实践专业的测试工作。另一方面，近几年我们接触了很多规模较小的软件研发组织，甚至是一些创业公司，他们的团队和业务在快速发展，有非常强烈的意愿去深入了解体系化的测试和质量提升工作是如何开展的，但常常会感到一些迷茫。之前我们写过一些博客文章和培训材料，并参加了一些业界的技术交流，得到了非常多的正面反馈，让我们觉得这件事有价值，对同行们可能也有一些帮助。一两次的技术交流和零散的讨论总觉得不够系统，说不清楚。那不如写一本书吧，正好这些都是我们自己做过的事情，有过一些实践经验，也踩过一些坑。

本书的内容组织

基于以上的出发点，我们希望比较系统地介绍整个移动 App 的测试，其实广泛一点来讲是质量保证的工作，因为这本书里介绍的不少实践已经超出了单纯的测试的范畴。

首先我们会介绍一下典型的互联网产品的研发流程。就我们工作过的几家公司，每家都有一些不同，但是核心的做法其实非常类似。并不是简单地套用敏捷等流程方法，而是经过不断实践的摸索和调整，各家都找到一些适合产品特定以及互联网快速迭代要求的流程做法。这些也是后面讨论一些质量实践的基础。在第 1 章的第二部分我们介绍了功能测试中的一些实践，包括测试用例的设计和评审，以及测试进度的管理。

第 2 章介绍了自动化的方法，包括接口层面的自动化，这里我们重点介绍了一种实践过的轻量级方案，以及 App UI 层面的自动化，分布介绍了 Android 和 iOS 用到的一些技术方案。

第 3 章介绍了性能测试的方法，包括 Web 前端的性能，为了介绍这部分的性能问题，也介绍 HTTP 协议相关的知识，以及常用的测试方法。第二部分是 App 端的性能，包括 Android 和 iOS 内存相关的问题，以及内嵌 Web 组件的性能分析。最后介绍了后台服务的性能测试，包括了压力场景的建模、测试工具的介绍以及测试数据的收集和分析。

第 4 章重点介绍了几个针对 App 的测试方法，包括兼容性测试、流量测试、电量测试、弱网络测试、稳定性测试、安全测试和环境相关测试。这些方法，由于都是针对某个特殊方面或者问题的，所以我们统称为专项测试。

第 5 章介绍了代码静态扫描、代码覆盖率分析、接口 Mock 方法和 AOP 测试方法，这些是测试方法中非常有效的补充，我们称之为辅助测试方法。

第 6 章介绍了发布过程中的质量保证活动，包括持续集成的实践，以及发布环境的质量包括，包括发布系统的介绍。另外还专门讨论了内测和灰度这两个互联网产品比较常用的方法。

第 7 章介绍了质量的度量和推动方法。包括我们常用的一些质量分析的维度，QA 的角色和所做的工作，并专门讨论跨团队的质量推动。

第 8 章介绍了一些发布之后的质量管理工作，包括继续进行一些模块之间的交叉测试，发现一些之前没有发现的问题。另外，介绍了互联网产品的一些常见的监控维度，并重点介绍了适合测试团队开展的接口方面的自动化监控的实践做法。最后，讨论了关于外部用户问题反馈的收集和跟进的一些常见的做法。

第 9 章，最后，作为在软件测试领域工作多年的专业人员，我们也想借这个机会讨论一些我们对于软件测试、测试人员以及团队的看法和思考。因为前面介绍的所有实践，都

是这些人做出来的。

关于内容本身，如果只用一个词来形容其特点，我想那就是实战。除了个别知识点补充了一点点介绍性材料，这本书几乎所有的内容都是我们在真实的项目中实践过的，有很多材料都是直接来自真实的项目（当然做了一些敏感信息的过滤）。我们的原则是宁愿不全面，也不想误导。因为工作久了，我们发现有很多的理论似是而非，怎么说都有道理，比如凭空讨论一个企业应该专注一个领域还是应该多元化经营？这样的讨论可能会一直绕圈子，给不了真正有价值的参考；还不如介绍几个真实的企业是做什么的，处于什么样的状况，有什么优势，遇到什么实际问题，是怎么处理的。软件测试，甚至整个软件研发，都属于工程实践的范畴，最终是要有实际的产出，不是凭空的理论，所以我们觉得也应该用实践的态度来对待知识经验的分享。

谁适合阅读本书

说实话，在给这本书起名的时候我们有一些纠结，因为内容是围绕着一个移动 App 测试的各个方面来讲解的，但是有经验的读者会发现，这里介绍的测试技术和质量流程对于其他互联网产品同样适用。比如一些自动化和性能测试的方法，以及代码静态分析和覆盖率等技术手段，还有质量度量和推动的实践，都不局限于移动互联网方面，我们甚至觉得并不局限在互联网方面。就我们个人的经历而言，我们曾经参与过大型电信系统的开发，企业级服务器软件的测试，以及 PC 客户端的产品，后来转型到互联网领域，有很多在之前领域里好的实践可以被借鉴和应用，只不过要考虑实际产品和项目的特点来调整，我想反之也是一样。所以请大家不要被互联网或者所谓的互联网思维束缚，不是触了网就立即如何。我们仍然需要理解每一个技术的原理和优缺点，对于每一个质量提升的实践也是一样，然后结合自己所在项目的实践，优化和调整，这样会更加有效果。

基于本书的内容组织，这本书可能适合下面这些人：

- 希望将测试做得更加深入的一线测试人员，特别是互联网和移动互联网的测试人员，可以更加系统地了解相关的测试技术和方法。
- 希望提高代码质量的一线开发人员。本书有很多质量提升做法也可以用于开发，比如静态扫描和内存分析，在很多的团队中也确实如此应用。
- 测试团队的 leader，特别是一些接手互联网或者移动互联网测试团队时间不长的，可以比较系统地了解测试和质量管理工作的规划和思路。
- 希望提高产品质量和研发效率的研发团队负责人，可以作为对全流程质量提升的一些参考。

- 在校的大学生，了解到现在很多学校有测试相关的专业了，希望大家在校园里就可以了解到一些业界的实践做法。
- 其他任何关注移动互联网产品研发和质量提升的人员。

这是一本很全面的关于测试的书吗

嗯……我们很想说是，但很遗憾它可能还差很远。

一个人知道得越多，就知道自己不知道的越多，最近这几年工作的经历让我们深感如此。在工作中，我们不断遇到新的问题和挑战，新的技术和方法也在不断涌现出来。另外，我们在工作中接触了大量优秀的测试人员和各种深入的测试技术实践，也接触了许多业界同行，因此深知测试领域的博大精深。不过可能因为大家工作节奏都比较快，鲜有人系统化来做分享，所以本书也算是抛砖引玉，希望更多资深的业界同行把自己在一线的实战经验分享出来，共同推动国内的测试做得更加系统和深入，更加的有价值。也正是这样的想法激励着我们花费大量的业余时间，希望比较直接和鲜活地把我们在一线的实践分享出来。

本书阅读建议

对于移动 App 测试经验比较少的人，我们建议比较完整地阅读本书。对于有一定经验的人，请随手翻到你感兴趣的章节，因为本书的很多内容都有一定的独立性。本书的很多内容都结合了具体的实例讲解，因此我们也建议大家在看到相关的章节时动手实践。

关于作者

本书由三位作者：邱鹏（Ricky）、陈吉（Allen）、潘晓明（Shawn）共同完成，我们曾经在一个团队中长时间一起工作，对产品质量的持续提升和新的测试技术研究都一直怀有共同的热情。同时，我们背后有好几十位业务测试、测试开发和质量管理的同事都贡献了具体的项目实践和很多好的建议。本书具体内容的分工如下：

Ricky 规划了整本书的内容，并编写了接口自动化、Web 前端性能测试、后台服务性能测试、兼容性测试、部分流量测试内容、部分电量测试内容、弱网络测试、App 稳定性测试主要内容和接口 Mock 的部分内容，以及第 1 章、第 6 章、第 7 章、第 8 章、第 9 章。

Allen 作为资深的 Android 测试开发专家，编写了其中主要的 Android 相关内容，包括 Android UI 自动化、Android 内存测试和 WebView 性能、Android 代码静态扫描、Android 流量自动化部分、Android 的 ANR、安全测试、App 环境相关测试、Android 代码覆盖率、Mock Server、Android AOP 方法。

Shawn 作为资深的 iOS 测试开发专家，编写了其中主要的 iOS 相关章节，包括 iOS UI 自动化、iOS 内存和 WebView 性能分析、iOS 代码静态扫描、iOS 流量和电量的部分内容、iOS 代码覆盖率，iOS AOP 方法，以及持续集成中的 iOS 覆盖率案例。

分工协作是这本书得以完成的基础，不只是内容本身，也因为一个人无法承担繁忙的工作之余如此巨大的工作量，因为除了文字编写，每个案例都需要实践。协作本身就是一种精神力量，也是愉快的经历。

致谢

以下是一些我们共同想感谢的人：

首先我们想感谢曾经一起努力工作的腾讯、易迅和京东的同事们，他们对我们的测试工作给了非常多的支持和建议，促进我们不断提高。

我们也想特别感谢我们的编辑，机械工业出版社的吴怡编辑，她是推动这个写作计划变成现实的人，包括选题、内容的组织以及细节的文字方面都给了我们很多的帮助，是她的鼓励和肯定让这本书得以完成。我们也想感谢在写作本书的过程中，那些得知我们在进行这个长跑并给予支持和鼓励的人。

以下是几位作者分别想感谢的人：

Ricky 致谢：

首先我想感谢腾讯的吴凯华（Jeremy）先生，是他带我进入互联网测试领域，并给予非常多的指导，另外他还抽出宝贵的时间为这本书作序。为了这篇序，他向我仔细了解了内容组织背后的考虑，并阅读了本书的绝大部分内容，他这种认真负责的精神是一贯的，一直以来都令我非常敬佩，使人见贤思齐。这篇序本身也融入了他对于软件测试和个人发展的深入思考，非常值得一读。也特别感谢李俊（Jasper）先生，另一位在腾讯期间我的老板，也是非常的卓越、认真和正直，从他那里得到很多关于做好事情、带好团队的具体指导，至今受用。

感谢曾经在腾讯电商上海测试团队，以及京东无线测试部的每一位同事，这本书的内容是大家一起实践的一个小结，非常高兴能和大家一起共事，也为每个人取得的进步感到骄傲。谨以此书，致以：曹计昌先生、Joe Chen、Yun Zhang、John Li、Jicheng Wang、Tao Qian、Allen Wang、Enoch Huangfu、Fei Zou、付学宝、徐奇琛、彭晓虹、马弘烨、Scott Li、Step Tian、袁蓉蓉、党杰、朱永敏、幸锐、姚醒、王孝满、江川、盖美红、王宇、李松峰、陈保安、李伟奇、谭丁强、李大鹏、Nina Luo、Jessica、Shelly Hu、Victor Wan、Eddie Liu、Rocken Meng、Frank Xia、Allen Fang、Lampard Chen、Haison Tang、Allan Zhou 等老

师和朋友，在此表示深深感激。限于篇幅无法一一列举，但内心中对于所以给予过支持、建议和批评的人们深怀感激。

也借此机会感谢蚂蚁金服的同事们，感谢给予我新的机会让我可以迎接新的挑战。这本书包含了之前一些工作内容的沉淀，希望可以在新的领域有更多新的收获。

最后我想感谢我的家人，感谢他们对于我工作的理解，这本书的写作在工作之外又额外占去了一些本该陪他们的时间。儿童节快到了，把这本书献给我的女儿甜甜。

Allen 致谢：

首先感谢 Ricky 在 Android 专项测试工作中给予的支持和建议，此外感谢我的同事朱玮在 Android 代码覆盖率工作中的前期探索工作。本书 Android 部分的很多内容其实是源自前人的工作，加上我们自己的摸索和改进，总结出的一点经验。因此借此机会也感谢所有致力于 Android 测试的先驱者们。向你们致敬！

Shawn 致谢：

我首先要感谢我的 leader 邱鹏，在工作中给予了我不少帮助，得益于他的指导，让我能够在工作中自由地发挥，也使我个人在快速地成长。其次要感谢唐辰、王孝满两位开发同事在工作中给予我的支持和帮助，深深地让我体会到测试工作的开展离不开开发同事和测试人员的紧密配合。最后要感谢所有在工作中给予我帮助的同事们，是你们的帮助让我在工作中如鱼得水。再次感谢！

最后，想说明的是，虽然这本书几乎每一个技术点我们都在实际项目中实践和应用过，但即便如此，因为被测项目的特性差异，以及我们个人技术和视野的局限，难免有很多不完善和偏颇之处，所以我们也非常希望听到读者的反馈，帮助我们完善。你可以把意见和反馈发到这个邮箱：3007349@qq.com，谢谢！

目　录 *Contents*

第 1 章 Chapter 1

产品功能测试概述

人们在一起可以做出单独一个人所不能做出的事业。

——韦伯斯特

对于用户而言，移动互联网产品是一个可以在移动设备上安装的 App，或者一个可以为移动设备定制的页面，看起来比较简单，但是和 Web 互联网产品一样，任何一个功能丰富的移动互联网产品，背后都是有一个分工细致又密切合作的团队共同完成的。所以谈论移动互联网的测试首先就需要了解整个产品的研发流程，进而了解测试在其中的定位，以及和其他角色之间的协助。所以在本章开始我们会讨论一些常见的互联网研发流程，以及其中各个角色的分工协作。接下来会讨论在互联网产品研发快速迭代的节奏中，如何来做功能测试的一些基本实践，包含测试用例设计和评审，以及测试进度的管理和报告。

1.1 互联网产品常见的研发流程

对于每个研发组织，因为产品的特性、组织的特点和一些历史原因，对于产品研发流程的理解和设定都有不同的考虑。但是以我们工作过的几家互联网来说，因为互联网产品的一些共同点，大致的产品研发流程其实大同小异，或者是做类似的事情但叫法不同。考虑到本书的读者可能当前的工作范围不一定是互联网产品，或者还没有机会了解整个研发

流程，这里先做一些基本的介绍，也便于后面章节关于质量提升方面的讨论。

为了了解流程，首先需要介绍一下互联网产品研发相关的分工，主要的角色如下：

- 产品经理。负责产品方向和具体需求的规划，需求文档的编写。是待开发需求的提出方，或者代理方（来自业务部门等第三方的需求，由产品经理转化成研发团队的需求形式）。通常对于较大规模的产品，产品经理是一个团队，每个人分工负责部分功能模块的需求细节。
- 项目经理（以下简称 PM）。负责项目的立项和时间安排，并跟进项目研发的进展、变更和风险，以及各种跨团队的协调工作。在一个大的项目中，通常也会有多位项目经理分工协作。
- 设计师。负责产品的交互设计、视觉设计等方面。主要的产出是产品的交互原型和设计稿。
- 开发人员。负责产品的技术架构设计和代码编写，产出是可运行的实际产品。通常根据专业领域也进一步划分为架构师、后台开发、Web 前端开发、Android 开发、iOS 开发等多个岗位。
- 测试人员。负责产品的质量把关，包括功能、性能和稳定性等多方面的测试内容。进一步细分包括业务功能测试、测试工具和平台开发、专项技术测试等岗位。部分组织里面也将质量管理放在测试团队。
- 运维人员。负责产品的服务端运行环境的建设和维护，以及日常的配置管理、容量规划、网络和设备故障处理等工作，常常也包含监控平台的建设和管理。取决于研发组织是采用自建 IDC，租用 IDC 或者采用第三方云计算平台，运维团队的工作可能有所不同。
- 运营人员。负责业务和产品的推广和拓展。对于移动互联网产品，常见的工作范围包括 App 的推广，各类运营活动的规划和推动，同第三方一起开展的市场活动，以及运营平台的规划等方面。

在前面各个角色分工的基础上，图 1-1 展示了一个基于研发阶段和角色分工的流程图。也是在互联网研发中比较常见的一个流程，可以看出每个阶段要做的主要工作，以及对应角色在该阶段的产出物。

图 1-2 给出了一个以主要研发活动为线索的流程图，从中可以看出各个参与角色对应的研发活动的衔接。比如在需求评审完之后 PM 组织大家排期；开发自测完成之后交给产品经理体验；测试完成并发布测试报告，以及发布策略确定后进行发布上线。

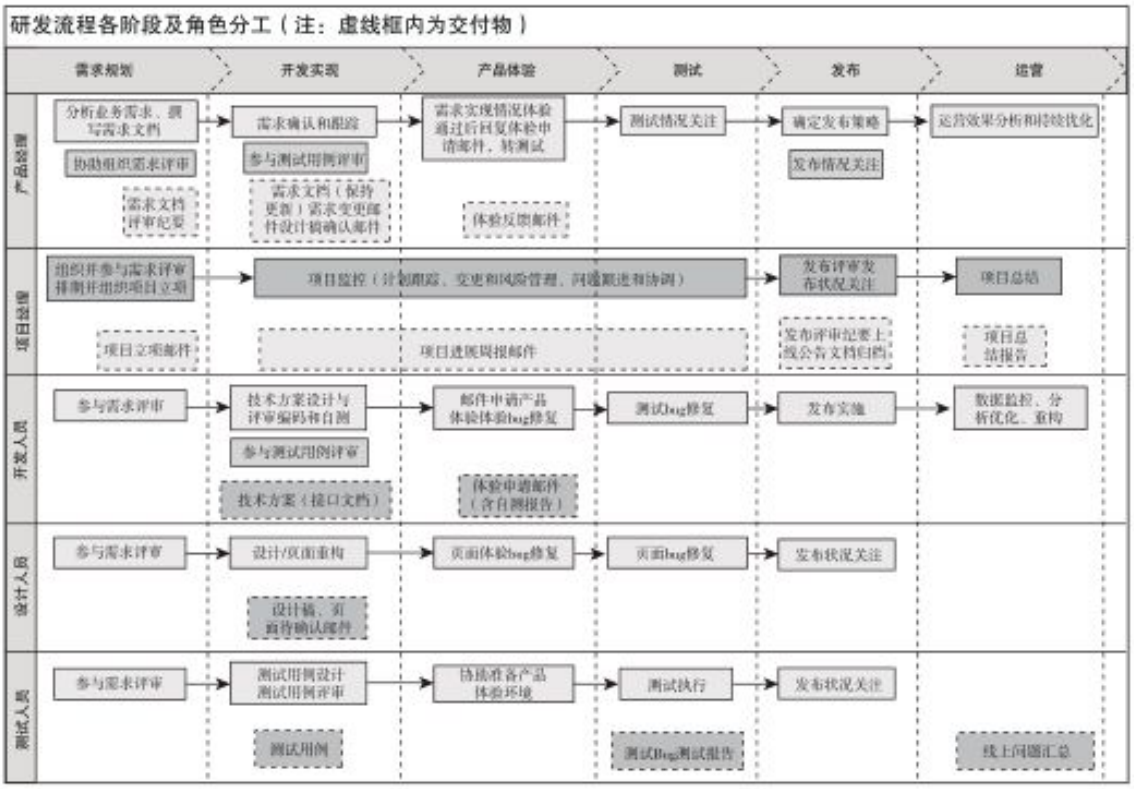


图 1-1　移动互联网研发流程及角色分工

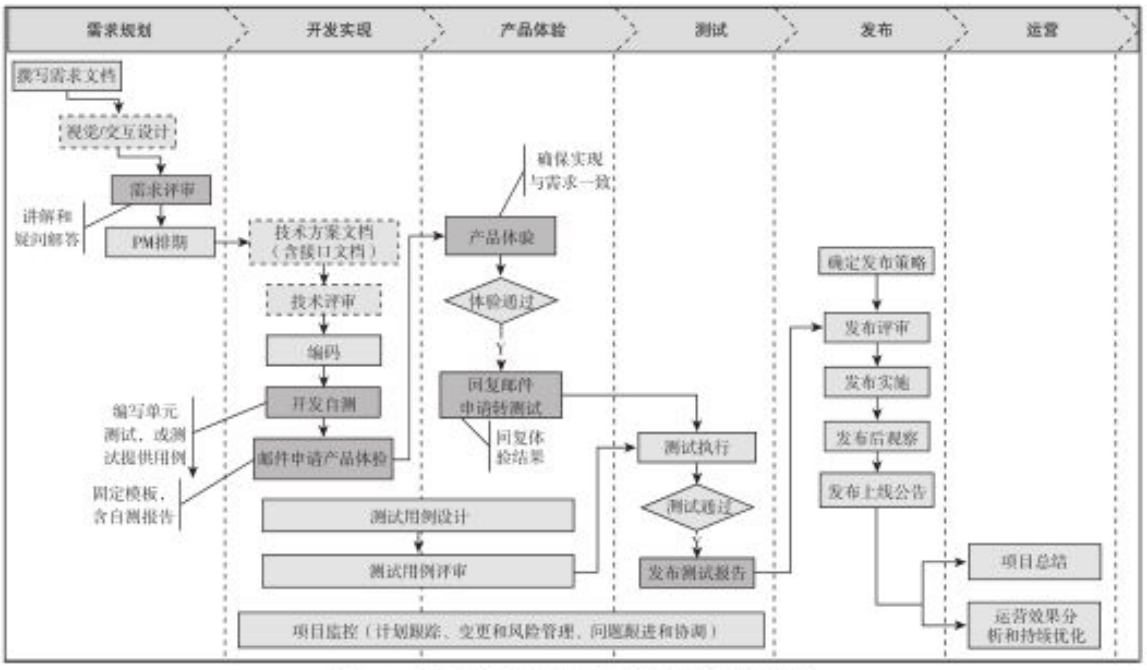


图 1-2　基于工作衔接的互联网研发流程图

为了适应互联网快速迭代的节奏，整个流量相对比较轻量。以上流程描述的是单个需求的处理过程，实际中，对于同一个 App 或者后台版本，一般是有多个需求并行的，而且不同版本的需求是有交叉重叠的。在一个版本研发的后期，通常会进行下一个版本需求的讨论和评审。

在本章的后续部分，以及后面关于质量管理和推动的章节，会对其中的多项重要流程实践做进一步深入的讲解。在这里我们讨论需求评审和技术方案评审。需求评审是一个比较常见的研发流程实践，但是实际上大部分人还是低估了其价值，执行的过程中也做得不够充分。对于互联网产品快速迭代的节奏，固然需求评审会花去一些宝贵的时间，但是事后来看，无论正面的案例还是反面的案例，这个时间花得还是非常值得的，正所谓磨刀不误砍柴功。需求评审，特别是现场会议形式的评审，是一个非常难得的多方沟通的机会。多个角色可以统一对需求的理解，有疑问的地方可以及时讨论。

从测试人员的角度，需求评审的价值主要在以下几个方面：

- 充分理解需求，为后续的测试用例编写打下基础。
- 基于对需求细节的了解，可以更准确地评估测试的要点和工作量。
- 发现需求中模糊不清的地方。从质量管理的角度，这也是一个非常好的缺陷预防的方法。

为了现场评审有更好的效果，并提高评审的效率，建议评审组织者在评审会召开之前将需求文档提前发给评审专家进行预审，在预审结束之前评审专家预先将发现的问题发给评审组织者，这样可以在会上针对问题进行评审，使评审更充分、更有效，否则需求评审会有可能会演变成一个需求讲解会，从而达不到预想的效果。

除了需求评审，对于一些偏技术性的需求，或者一个全新开发的功能，很有必要做技术方案方面的评审。请对应的开发人员来讲解一下准备采用的技术方案。一方面可以请其他资深的开发人员帮助评审，另一方面从测试人员的角度，了解基本的技术实现也有助于设计测试用例，并提前为可测性做一些准备。

下面这个例子可以帮助我们理解技术方案评审的价值。下面是一个偏技术类需求的例子，是一个自行开发的 App 端的 SDK，用于将 App 遇到的一些异常问题汇总上报，包括 crash 和各种错误，其他运营类数据的上报也是类似的逻辑。在第 8 章关于监控埋点的部分也会做相关的介绍。

图 1-3 是针对这个需求的一些测试用例。

从以上用例可以看出，如果不了解一些技术实现的细节，很多测试场景根本无法覆盖，比如：

- 未上报的临时数据是通过本地数据库来存储，就会有相关的测试场景。其中数据库文件大小需要有一个上限，这个其实也是评审过程中测试提出来的，避免在一些极端情况下占用过多的存储空间。

图 1-3　针对异常上报 SDK 的测试用例

- 上报进程是基于接口下发的策略来控制上报逻辑的。这个部分也会引起一系列可能出现的问题。
- 对于一些可能遇到的异常情况，设计上是如何考虑的，这些其实在技术方案评审的时候就可以考虑到。

类似的例子很多，对于一个全新的功能也是如此，不同的开发人员会有不同的设计上的考虑，哪些逻辑放在 App 端？哪些放在后台？同步的机制如何做？有哪些新增的接口？这些都是技术方案上的考虑，不同的设计会带来不同的测试场景和测试点。

不覆盖这些场景就会有很多质量问题的死角，而这些都是 App 性能和安全性的隐患。

关于流程，我们的观点会更偏向实用主义，并不在乎是否是纯粹或者经典的某种模式。比如上面的技术方案评审，也是来自于项目运作中实际的需求，讨论下来觉得部分功能有必要开展就加到流程里面。

另外在工作中我们发现一个特点，在我们了解的几个大的互联网公司里面，不太经常谈敏捷概念，而更多是根据业务运作的特点，以及团队不断的磨合，整理出一套相对有一定适应性的流程。

另外，还有一个观点，对于一个用户量很大的网站或者 App，当版本有大量的需求迭代，比较频繁地来做发布（网站更明显），每个需求的开发周期比较短，同时又是多个角色大量的人员合作，经历了几个版本，一段时间以后，大家就逐渐磨合出了一套适应这个需求的流程，并在过程中不断优化调整。所以，所谓的互联网的做法也并不是因为它比传统的软件流程先进，而是它更适互联网产品的运作方式而已，本质上还是产品形态和需求驱动的。

1.2 测试用例设计和评审

测试人员的一个基本工作，或者说是基本功，就是测试用例的编写。对于一些快速迭代的互联网产品，关于是否需要编写测试用例，也有一些讨论和争论。

就我们的观念，觉得还是需要，特别是对于 App 这样的产品，很多功能有一定的稳定性。类比来说，测试用例相当于电影的剧本，有场景、动作、台词，规划出一个基本的框架。测试用例也是一样，针对什么功能，在什么情况和使用场景下，做什么操作，用什么数据，期望有什么样的结果，进而和实际结果对比判断是否合理。

如果完全没有这样的剧本，测试会比较盲目，更重要的是，参考上面 SDK 测试用例的例子，如果不系统地把这些测试点提前记录下来，等到测试人员拿到可测的版本，如何保证能想起上面所有这些情况，并系统化地覆盖？

对各种场景和路径比较系统化的覆盖，是对一个专职或者专业测试人员的基本要求，

这一点使他们和普通用户的使用区分开来。也体现了测试的系统性、深度和效率，在很短的时间里覆盖足够多的场景。另外，还有很重要的一点，当我们把这些测试点记录下来，也可以请其他测试人员，以及产品和开发等一起来评审，确保测试用例的正确性和全面性。缺少用例，也不便于知识的积累和传递，因为现实中会有具体模块负责人员的变更和交接。

当然，虽然有了测试用例，实际执行过程中还是有一些临场发挥的成分在里面，就像电影剧本一样，对于同样的剧本，不同的人来演绎效果会非常不同。测试用例的执行也是一样，同样的用例，不同的人还是会发现不同的问题，因为人在执行的过程中观察的点会不同，操作的方式也会有些差别。

在是否编写用例上我们给出了自己的建议，这一点和传统的软件研发流程一样。但是细节的做法有差异，主要体现在以下几个方面：

1）**用例设计上的投入**。在我们曾经工作过的企业级产品的测试中，因为一个发布的版本是 6 个月以上的时间，所以测试用例设计的流程会做得比较严谨，或者以互联网的观点来看，比较重。之前的做法是对于每一个模块需要编写测试设计文档，讨论需要考虑的测试场景，然后进行开发测试内部评审。修订完了之后开始编写正式的测试用例，然后再召开测试用例评审会。这些固然严谨和全面，但对于互联网产品，很多模块只有几天的测试时间，无法负担，所以通常省去了测试设计思路的文档化过程。

2）**用例编写的详细程度**。严格来说，测试用例应该至少包括下面的这些要素：

- 用例的题目，一句话描述。
- 测试步骤，逐个写下，详细程度需要有少量经验的人也可以执行。
- 前置条件，此用例的执行需要哪些前置条件或者在什么条件下才会有预期的结果。
- 测试数据，此用例的执行需要什么样的测试数据，比如无货的商品，特殊的优惠券等，需要提前准备好并附上。
- 期望的测试结果。

同样，如果每个用例写到这样的程度，实际上对于很多项目来说也是无法承受的。对于上面给出的测试用例示例可以看出，里面每一个叶子节点其实对应的是一个用例，但是非常的概要，或者只能算是一个提示，告诉对应的测试人员需要考虑这样的情况。但是并没有详细地给出测试的步骤、数据和期望的结果。

某种程度上，这是一种妥协，但也是另一种工作方式，用例就是一个故事梗概，需要对应的测试人员了解需求的上下文，以及基本的实现方式，进而来执行。这一点上，有点像演讲时用的 PPT，可以把要说的话全部写在上面，也可以只写几个关键词，其他需要说的自己补充和发挥。

可以看出，这种方式的可行性，其实对测试人员提出了更高的要求，需要对负责的业

务功能细节非常了解，并且对测试环境和数据等方面也能把控。所以在人员的分工上，一段时间，对于一个功能模块，会有一个具体的测试负责人来跟进。

图 1-4 所示用例是针对 Android App 增量升级功能，也是类似的方式，非常简洁，对这个功能有一定了解的人会知道每一条用例代表的场景和如何执行。

图 1-4　针对 App 增量升级功能的部分用例

3）**表现形式**。这个差异主要是因为引入了思维导图的表现形式，基于常用的 Xmind、Freemind 等工具，上面两个示例都是在 Xmind 中编写的。传统上测试用例主要的载体是 Excel 表格，或者基于 Web 的测试用例管理平台。这两种形式都可以比较完整地表达上面提到的测试用例的各个要素。遇到的问题是：一方面编写的工作量比较大，考虑一个功能模块有超过 100 个用例是很普遍的；另一方面缺少逻辑关系，这一点也是思维导图的优势。

以上几种形式我们在不同的项目中都实际应用过，包括也试验过先用思维导图来编写用例，然后导出成 Excel 或者导入到平台等方式。实际应用下来，转换的过程体验并不好，也会存在变更后双向同步的问题。在目前实际项目的做法中，我们并没有严格要求测试人

员编写用例的形式。

关于常用的测试用例设计方法，这里不准备展开来讲，因为那需要写一本独立的书，另外业界也有很好的参考，包括但不限于下面几本：

- 《软件测试》（Software Testing：A Craftsman's Approach, Second Edition），作者是 Paul C.Jorgensen。这本书介绍了很多经典的测试设计方面的方法，非常的详细，很多思路依然有效。
- 《微软的软件测试之道》（How We Test Software at Microsoft），作者是 Alan Page、Ken Johnston 和 Bj Rollison。介绍了一些测试的基本概念，以及等价类划分（Equivalence Class Partitioning）、边界值分析（Boundary Value Analysis）、组合分析（Combinatorial Analysis）等经典测试方法，以及 Model-Based Testing 的案例。
- 《探索式软件测试》（Exploratory Software Testing），作者是 James A.Whittaker。这本书给出了很多可操作的探索性测试的方法，其中有很多方法已经沉淀下来变成可以借鉴的测试设计思路，而不再只是探索了。

除了以上的资料，以及个人的尝试和琢磨，最快速提高测试设计能力的实践是参加测试用例评审，特别是参加一些有丰富测试经验的人在场的用例评审，可以很快地打开思路，考虑得更加全面。

从项目和测试质量的角度，测试评审的帮助也会非常大，以实践的经验来看，这个实践除了能让测试人员考虑更加充分，覆盖更多必要的场景，也能帮助大家提前理清很多产品功能的细节和注意事项。在测试用例评审的时候，需要鼓励大家打破思维的局限，敢于思考各种可能会出现的复杂场景，以及异常情况。对于需求和实现模糊的地方，尽量不要做假设，而是需要和对应的人去确认。也是因为这个原因，往往还会发现很多需求和功能设计上考虑不周全的问题。因为当我们深入讨论一个测试场景的时候，就会发现我们不知道我们的产品经理和开发人员是怎么处理的，也可能他们还没有考虑到这种情况，也可能相关的信息没有同步导致大家理解不一致。

针对这样的情况，我们尝试过觉得比较好的做法是，把这些问题逐个记录下来，然后通过邮件集中发出来给对应的产品经理和开发人员来确认，进而让大家的理解达成一致。

对于团队成员测试设计能力的考察和提升，还有一个可以实践的方式是用例设计的 Workshop。就是让所有测试人员都集中在一起，然后给出一个功能场景请大家来写出自己认为需要的用例。选取的场景最好比较通用、大家都比较容易理解，比如上传用户头像，登录注册，电商里面的发表用户评论送积分等。现场让每个人写下用例，只需要简略地写出每个用例希望覆盖的测试点。

接下来请每个人讲解自己的用例，并在白板上记录，多个人考虑到的测试点可以记录

次数。

通过这样的活动，可以看出每个人对测试设计考虑的维度、深度以及全面性。因为是同一个功能，所以大家可以互相借鉴和参考，并发现自己考虑上的不足。通过这样的实践，可以不断提升测试设计的质量。

1.3 测试进度管理

在一个较大型的项目中，通常运作的方式是按照子项目或者功能模块来进行分工，每个功能模块有具体对应的设计、产品、运营、开发和测试人员。结合实际的项目情况，如果功能较大可能上面一个角色有多个人一起参与，反之也可能一个人同时负责多个功能模块。不管是哪种情况，实际项目在测试进行中，以上不同的角色，以及对应的各个团队 leader，甚至公司或部门管理层，都希望及时看到工作的进展，以及遇到的问题和风险。

而另一个方面，互联网产品的测试周期都比较短，一个模块的整个测试周期只有几天是非常常见的，使得我们不可能有大量的时间用在整理测试进度的报告本身。结合这两种情况，我们需要考虑一个比较清晰简洁的方式来反映出测试工作的进度，暴露出其中的问题让大家尽快关注到，同时让编写这样的进度报告的代价变得比较小，因为太多的文字工作是无法承受的。下面我们来看一下在实际的项目中我们用到的一些方式。

从大的方面，我们将测试报告分为两类：

1）**测试进度报告**：在测试阶段中间发出，告知测试工作的进度，发现的问题、风险，以及接下来的计划。

这个报告发送的频次依据具体的项目情况而定，对于比较重要的且时间比较短的项目，建议每天发出，让相关人员可以非常及时地了解进展和风险。对于一些周期比较长的或者重要性的不高的项目，可以考虑隔天或者每周发送，基于大家的讨论来约定。

2）**测试完成报告**：标志测试工作的结束，会给出对应的测试结果和结论，包含是否达到可发布的标准以及还有哪些遗留问题。这个报告一般在整个测试工作完成之后发出，针对某一个具体的模块或者整个的测试项目。

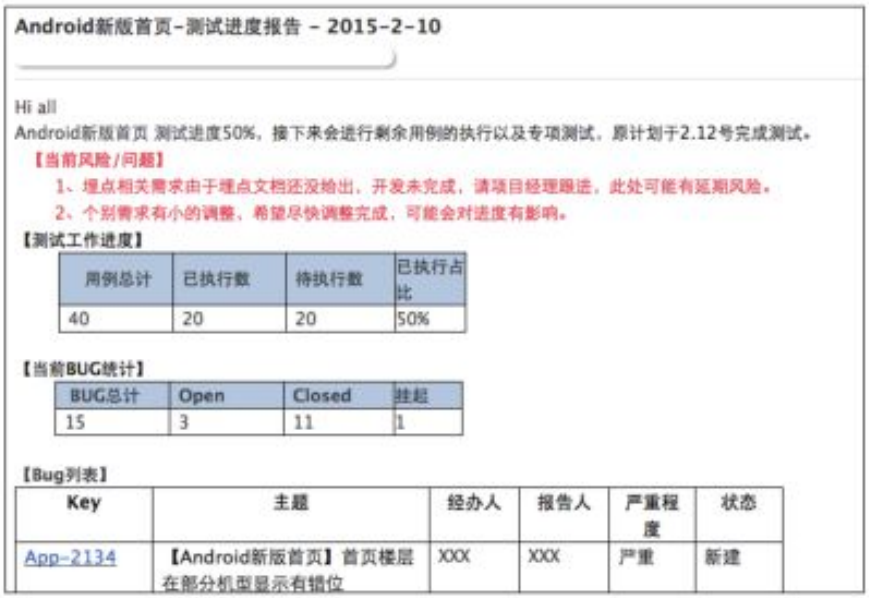
下面我们来看看这两种测试报告的具体内容。

1.3.1 测试进度报告

首先来看下测试进度报告，我们拿一个具体的邮件报告的例子来看，如图 1-5 所示。

从以上报告我们可以看出，主要的内容非常简洁，符合我们前面说的目标。主要侧重

以下几个方面：



Android新版首页-测试进度报告 - 2015-2-10

Hi all

Android新版首页 测试进度50%，接下来会进行剩余用例的执行以及专项测试，原计划于2.12号完成测试。

【当前风险/问题】

1、埋点相关需求由于埋点文档还没给出，开发未完成，请项目经理跟进，此处可能有延期风险。

2、个别需求有小的调整，希望尽快调整完成，可能会对进度有影响。

【测试工作进度】

用例总计	已执行数	待执行数	已执行占比
40	20	20	50%

【当前BUG统计】

BUG总计	Open	Closed	挂起
15	3	11	1

【Bug列表】

Key	主题	经办人	报告人	严重程度	状态
App-2134	【Android新版首页】首页楼层在部分机型显示有错位	XXX	XXX	严重	新建

图 1-5　测试进度报告邮件示例

- **风险和问题**：基于要事先说的原则，在邮件的一开始就把当前遇到的可能影响项目质量或者进度的问题列出来。如果是比较紧急的，可以标红或者加粗来引起收件人的注意。
- **测试工作进度**：这个可以给出一个大概的百分比，可以用测试用例的执行情况，也可以基于测试人员自己的工作估计。
- **当前 bug 统计**：一个简单的 bug 统计，让大家可以看到 bug 的总数和待处理的 bug 数量。
- **未关闭 bug 列表**：让大家可以比较直观地看到待解决 bug 的情况，从标题上就有个基本的了解，以及对应的状态、严重程度和相关的处理人。

以上测试进度报告的内容在项目实际运作过程中并不是严格限定的，相当于一个指导，大家可以基于项目的情况，补充项目的内容信息。

比如我们常遇到以下几种情况：

1）该项目包括多个子项目，每个子项目有独立的进展情况，但是整体在一个进度报告里面比较高效。

针对这种情况，可以在上面的“测试进度报告”里面写上各个子项目的进展情况，如表 1-1 所示。

表 1-1 按子项目划分的测试进度

测试项目	进　展
子项目 1	100%
子项目 2	70%
子项目 3	60%
子项目 4	80%
子项目 5	30%

2）专项测试进展。针对某个模块的测试，除了基本的黑盒功能测试之前，可能还需要进行其他针对性的测试，我们称之为专项测试，具体的专项测试内容和做法将在本书的后续章节展开讨论，这里为了便于说明，只简单列举部分测试类型，比如：兼容性测试、流量测试、电量测试、弱网络测试、代码覆盖率测试。

针对不同类型的模块，专项测试的开展情况可能不同，如果有相关的开展，建议也在测试进度报告中显示出来。可以在上述简单模板的基础上，补充类似下面的信息：

【专项测试进展】

兼容性测试：已完成 Android 2.3，4.0，以及 3 种不同分辨率的测试，整体进度 40%

流量测试：已完成，无异常

电量测试：已完成，无异常

弱网络测试：已完成，一个问题已提交 bug，bug 单号 ××××

代码覆盖率测试：未完成

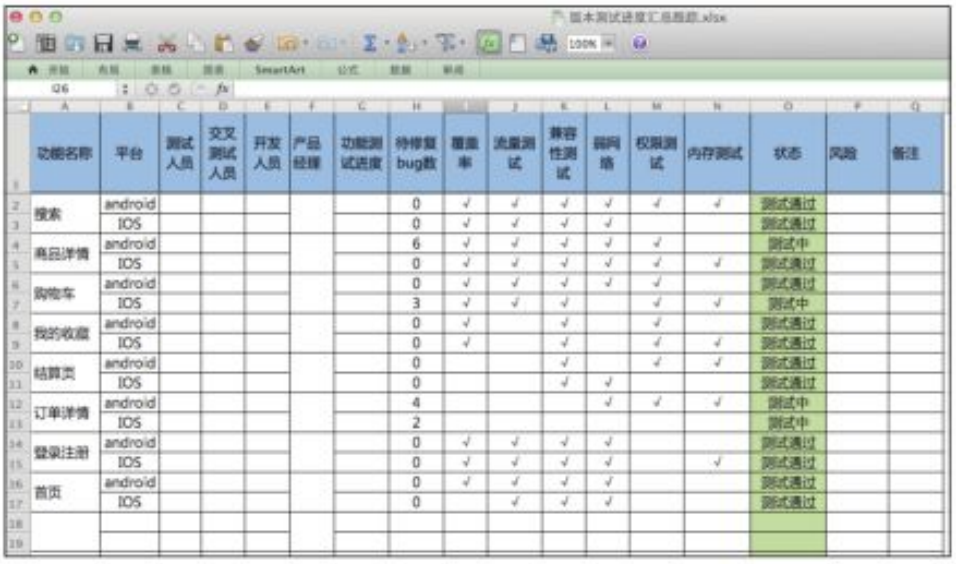
以上是单个项目的测试进展报告，为了便于全局了解整体的测试进度情况，可以将各个项目的进展汇总到一张表格中，如图 1-6 所示。

在这个聚合的整体测试进度报告中，信息已经被抽象，所以不像单个模块的测试进度报告一样可以看到比较细节的信息。如果需要了解对应模块的细节，需要查看该模块的测试人员发出的针对该模块的测试进度报告。

1.3.2 测试完成报告

当某个具体的功能模块测试完成后，对应模块的测试负责人会发出对应模块的测试报告，发给相关的项目经理 / 设计 / 产品 / 开发 / 运维同事，以及对应的团队 leader，标志着该功能通过了测试，可以进入发布（或者灰度）阶段。

图 1-7 所示是一个测试完成报告的样例。



功能名称	平台	测试人员	交叉测试人员	开发人员	产品经理	功能测试进度	待修复bug数	覆盖率	流量测试	兼容性测试	弱网络	权限测试	内存测试	状态	风险	备注
搜索	android						0	√	√	√	√	√	√	测试通过		
	IOS						0	√	√	√	√			测试通过		
商品详情	android						6	√	√	√	√	√		测试中		
	IOS						0	√	√	√	√	√	√	测试通过		
购物车	android						0	√	√	√	√	√		测试通过		
	IOS						3	√	√	√		√	√	测试中		
我的收藏	android						0	√		√		√		测试通过		
	IOS						0	√		√		√	√	测试通过		
结算页	android						0			√		√	√	测试通过		
	IOS						0			√	√			测试通过		
订单详情	android						4				√	√	√	测试中		
	IOS						2							测试中		
登录注册	android						0	√	√	√	√			测试通过		
	IOS						0	√	√	√	√		√	测试通过		
首页	android						0	√	√	√	√			测试通过		
	IOS						0		√	√	√			测试通过		

图 1-6　项目整体测试进度跟踪表

测试完成报告				
测试项目/项目名称	V5.0 Android版首页功能测试		测试负责人	Android测试小组
验证结果	通过	遗留问题数	1	
遗留问题	【遗留问题列表】			
测试范围	1. 核心功能：XXX 2. 基本功能：XXX 3. 新增临时需求如下： （1）XXX （2）XXX （3）XXX			
测试文档	1、需求文档：	【文档URL】		
	2、设计原型：	【文档URL】		
	3、其他文档：			
测试日期	测试开始时间		测试结束时间	
	2015/2/2 9:00		2015/2/6 20:00	
测试用例	【文档URL】			
BUG记录	【bug管理系统列表URL】			
专项测试结果				
备注				

图 1-7　测试完成报告的样例

实际中报告的具体信息可以根据团队项目管理的要求来做调整。在有专职项目经理来跟进功能和版本进度的情况下，通常项目经理会基于这样的测试完成报告来认定该模块的测试完成，常常也意味着该模块研发工作的完成，所以在发出该报告前也需要将遗留问题都评审过，大家认定当前模块质量达到了发布标准。

以上是单个模块的测试完成报告，对于整个 App，通常 Android 和 iOS 等平台分开来看，因为对外发布的单位是一个完整的 App，所以也需要一个完整的测试完成报告来给出测试的结论。格式和上面类似，只是测试范围和遗留问题的罗列可能会多一些。

1.3.3 系统化的方法

以上介绍的测试进度和测试完成报告，基本能满足测试项目管理的需求，在日常的测试工作中保持信息的及时同步。但是在信息的完整性和效率上也有进一步提升的空间，主要体现在下面几个方面：

1）以上报告里面的信息都是手工填写的。上述报告的里面的测试用例信息，bug 统计信息和列表，以及对应的需求点等信息，都是测试人员手工填写到报告中的，这个会需要耗费一些时间，同时信息的准确性有时也得不到保障。

2）一些时间维度的信息丢失。按前面讨论的，测试进度报告是定期发出的，每次看到都是一封独立的邮件，阅读报告的人无法快速地了解到时间维度的信息，比如：

- ❑ 该功能是什么时候提测的，有没有提测延期？
- ❑ 测试和预期有没有延期，偏差的原因。
- ❑ 测试的耗时情况。

这些信息也可以手工填写，但是非常容易因为理解不一致而出错。

3）沟通的成本。如果需要报告格式的变更，需要很多的沟通成本。

针对这样的一些问题，在一些比较成熟的测试团队，特别是一些有较强开发能力的测试团队，通常会借助自动化的系统来标准化测试进度的填写，以及相关信息的自动抽取，测试人员只需要触发相关的报告，并填写少量的主观信息，就可以发出标准化的报告。

图 1-8、图 1-9、图 1-10 是单个模块的测试报告样例。

主题：【测试结果反馈】活动平台1.0beta012

2014年03月10日，活动平台1.0beta012 已完成测试。

对应需求/迭代	需求/迭代链接
测试报告总结	测试完成。
版本特性	1. 【活动平台】活动平台添加用户签到组件 2. 【活动平台】活动平台添加抽奖功能
是否需要发布评审	是
报告类别	版本测试
测试结论	测试通过
测试分析检查结果	不涉及
安全测试结果	不涉及

图 1-8　测试进度报告-part 1

转测试质量	合格
编译部署正常	是
环境部署校验	通过
功能用例	新增 10 个 ， 回归 0 个　执行 10个
首轮测试用例通过率	共 10个 ， 通过 10 个 ， 通过率 100%　(通过率=通过数/执行用例总数*100%)
首轮测试用例通过率说明	
初始预计/实际提测时间	2014-03-05 / 2014-03-05
初始预计/实际发布时间	2014-03-07 / 2014-03-10
发布延期原因	构造测试数据比较复杂，且有外部依赖，影响了部分测试进度。
测试时间	2014-03-05 至 2014-03-10
测试耗时	15 小时
测试人员	xxx
开发人员	xxx;
产品经理	xxx;

图 1-9　测试进度报告-part 2

【兼容性测试】 无

【问题列表】

序号	BUG单号	BUG标题	创建者	BUG状态	严重级别
1	xxx	【活动平台】xxx	xxx	closed	一般
2	xxx	【活动平台】xxx	xxx	closed	一般
3	xxx	【活动平台】xxx	xxx	closed	一般
4	xxx	【活动平台】xxx	xxx	closed	一般
5	xxx	【活动平台】xxx	xxx	closed	一般

【建议列表】 无

【用例执行信息】

序号	用例ID	用例目录-用例名称	执行结果	执行者	用例执行备注
1	xxx	xxxxx	通过	xxx	
2	xxx	xxxxx	通过	xxx	
3	xxx	xxxxx	通过	xxx	

图 1-10　测试进度报告-part 3

以上报告的大部分信息都来自于功能测试平台，测试人员只需填写测试报告总结和发布延期原因等主观信息，以及针对一些选项做选择。整个模板是标准化且可定制的。

如果要做到上面的系统化方法，除了测试团队有一定的成熟度和开发能力之外，其实对整个研发组织的项目管理也提出了一定的要求。上面功能测试平台的信息粗略地说需要来自三个方面的系统：

- 版本需求管理平台。将产品经理或者研发自身提出的各个功能迭代，录入到对应的系统管理起来，需求评审 / 开发 / 走查 / 转测试 / 测试完成等各个状态都可以在系统中扭转，各个对应的研发角色可以参与进来操作，并完善相关的信息。这个比

Excel 和邮件的方式来管理需求要方便和高效很多。

- ❑ 测试用例管理平台。可以方便地进行用例的录入和管理，以及支持评审等功能。
- ❑ 缺陷管理平台。缺陷的创建、状态的扭转、已经数据的统计等功能。

以上三个平台都需要提供相关的数据接口，允许测试平台等外部程序拉取对应的信息来汇总。

而更近一步，如果平台上有了每个模块的上述基本信息，自然就会想到可以借助平台来自动化地聚合整个项目组，或者整个大的 App 版本所有模块的测试进展和完成情况。这样就可以非常具体地看到各种不同维度的测试项目的信息，对于测试团队的负责人来说，对整个测试情况也会有比较好的把握，也能及时发现项目的风险和问题，同时也提到了非常多的度量角度，对项目和团队管理上也是很好的参考。

1.4 本章小结

互联网产品的迭代速度非常快，所以对整个研发流程的效率要求非常高。而另一方面，一个大项的系统或者项目都是需要大量的不同角色的人员来协作完成，同时对质量的要求也很高。要保持项目长期良好地高效运转，这两者之间需要一定的平衡，过犹不及。测试作为衔接研发和发布的重要环节，其中的流程和效率也就非常重要。上面列出的是一些实践的做法，只是一个参考，并不是什么标准，实际中还是要以团队和项目的特点来找到适合自己的做法，总之，需要以高效、轻量、清晰和实用为原则。

第 2 章 Chapter 2

功能测试自动化

由于频繁地重复，许多起初在我们看来重要的事物逐渐变得毫无价值。

——叔本华

如果仅仅依靠纯手工的测试执行，很快测试就会面临瓶颈，因为每一个功能几乎都不是第一次提交测试后就测试通过的，所以就需要 bug 修复、验证，以及回归的过程。另外，有很多的测试工作手工做起来非常的繁琐，甚至不便，比如针对接口协议的验证。这些都依赖于测试自动化的开展。针对移动互联网的产品，本章主要介绍两个方面的自动化，一个是基于接口的自动化，另一个是基于 App UI 方面的自动化。关于接口自动化方面，下面会介绍一套我们在实际项目中应用过的基于开源组件的轻量级自动化方案。针对 App UI 自动化，主要介绍 Android 和 iOS 的 UI 自动化的常用技术，并结合一些实例来讲解。Web UI 方面的自动化技术这里不会涉及，网络上可以找到比较完善的资料和文档。

2.1 轻量接口自动化测试

无论 Web 互联网的产品还是移动互联网的产品都必须依赖大量的后台接口提供的服务，有很多的业务逻辑都是放在后台来处理的，所以非常有必要对这部分逻辑来做测试验证。技术方案上，也可以模拟用户的 UI 操作，从界面上发起相关的请求。但是实际中，会发现这样的做法效率不高而且稳定性不够，开发和维护的代价也比较大。针对这部分的测

试，最直接的方式还是从接口层面发起请求来验证。

就目前观察，对于一些比较稳定的基础性组件，比如底层平台、API、SDK 等，或者功能通用性高的产品，比如防火墙、邮件系统等，都可以做到比较高的自动化率，而且自动化测试开发的方案也相对比较明确。相比而言，偏重应用层业务的测试团队，通常在自动化方面的开展要困难很多，主要有以下几个问题：

- 首要的问题是版本的节奏，互联网产品版本节奏非常快，一个稍大的系统一周发布上百个功能特性是一件很常见的事情。团队成员有非常多的精力消耗在这些功能版本上，需要快速理解业务，构建测试环境，进行 bug 验证回归，以及发布和线上验证等，留给业务测试人员构建自动化用例的时间非常少。
- 自动化框架的开发代价比较大。自己开发或者维护过测试框架的人可能都深有体会，这绝不是一件容易的事情，其工作量和持续的时间往往会超出我们的预期，特别是对于大部分的互联网测试团队，尤其是一些初创团队。而另一方面，也很难找到一个现成的，能满足特定需求的平台。
- 实际的项目，特别是大型的项目，功能通常都非常的复杂，需要将业务逻辑通过框架的能力来表达，对构建用例来说也存在一定的门槛。而测试开发人员由于对各个业务细节的了解程度不够且精力有限，对于需要构建大规模自动化测试用例的系统也是一个挑战。

对于一些大的互联网研发团队，如果有一个比较大的测试部门，对应有比较强的测试开发能力，可以考虑自己开发一套自动化测试框架，然后可以在多个业务测试团队复用。对于一些人力有限的团队，则需要考虑一些轻量级的方式。这里不准备介绍如何开发一个自动化框架，那可能需要写一本书的篇幅。下面将要介绍的是适合小团队的一些测试自动化的方法，基于我们之前在一些实际项目中的实践。这套方法我们曾在几个不同类型的项目中得到应用，包括电商的后端 ERP 系统、Web 网站以及 App 的后台接口，都获得了比较好的实际效果，同时测试开发和用例编写的代价相对可以承受。

2.1.1 JMeter 关于自动化方面的特性介绍

这个轻量级方法的核心是采用开源测试工具 JMeter（http://jmeter.apache.org/）作为引擎，把发送接口请求，以及结果的解析和断言的工作都交给 JMeter 的基础功能来实现。这样的考虑涉及两个问题：一是为什么不从头开发一套完整的自动化框架？二是为什么采用 JMeter？

关于不从头开发一套完整的框架，主要的考虑如下：

1）首先主要是开发成本的考虑。用当前任何一个主流的开发语言，比如 Java/Python/

PHP 等，写一个协议层面的收发都非常简单，因为有了大量的通用的库。但是如果用于自动化测试，需要考虑的方面非常多，比如以 HTTP 协议为例：

- 需要考虑数据编码的问题
- 需要处理连接的建立，销毁的问题
- 需要考虑 Cookie 的问题，自动重定向的问题，是否用 keep-alive
- 对于 POST，需要处理发送前的数据准备问题
- 需要处理代理的问题
- 需要处理返回数据的解析
- 需要支持各种断言的形式

2）在自动化测试的开展过程中，会不断对底层框架提出新的要求，需要有持续的测试开发人力投入。

3）需要保证框架的高质量和稳定性。很多内部开发的框架都会遇到上面的问题，投入了大量人力，开发了很长时间，也需要持续的维护，但是一旦有大的产品和人员的变动，很多时候，框架或者框架的很多功能都会被废弃。

4）这可能不是团队的工作核心。开发一个 HTTP 等基础协议的收发处理的框架对很多业务测试团队来说，都不是核心的职责和最重要的事情，特别是人力有限的情况下。

基于这样的考虑，为了快速构建一套可用的自动化框架，我们决定寻找一个开源的工具，能完成协议层面的基本功能。最后我们选用了 JMeter，基于它来构建我们的自动化平台。

JMeter 是 Apache Software Foundation 下面的一个开源项目，有超过 10 年的历史了。它是一个纯 Java 编写的测试工具，最早主要用于 HTTP 协议性能测试（现在这仍是其主要用途之一），但是后来随着功能的逐渐丰富完善，它也成为一个接口协议测试工具，以及自动化测试工具。下面介绍下它的一些特点，主要从功能测试的角度，其他更进一步的信息可以参考它的官方文档，或者下载之后试验。

1. 支持多种不同类型的协议

从图 2-1 可以看出，JMeter 自带支持的接口协议有多种，可以直接应用，省去了上面提到的接口协议的数据收发方面的开发。

2. 对 HTTP 协议的支持比较全面

HTTP 是互联网最常用的协议，目前很多移动互联网产品也是通过 HTTP 协议来完成后台的交互。JMeter 对 HTTP 的支持比较全面，如图 2-2 所示，对于协议方法、请求的参数和选项、代理等方面都提供了支持。

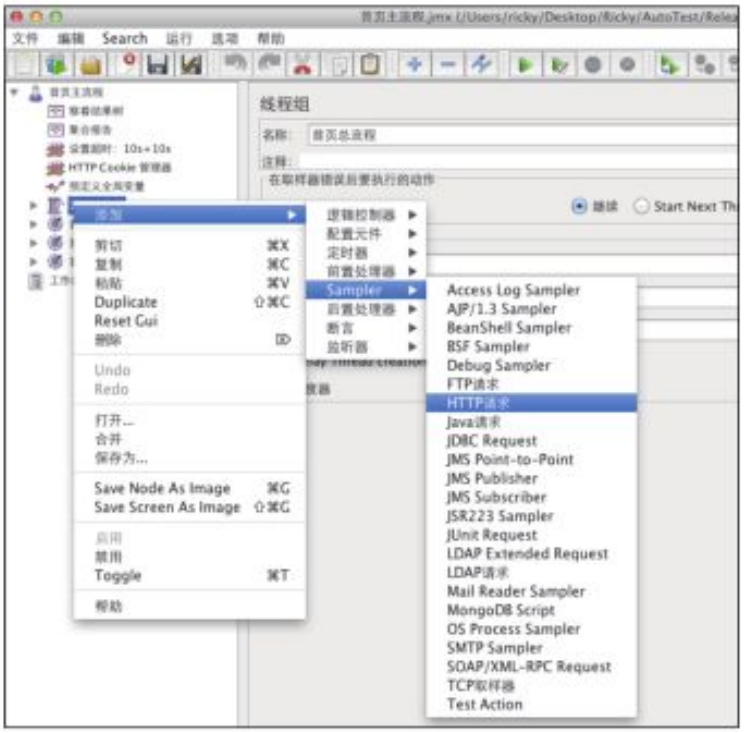


图 2-1 JMeter 支持的请求类型

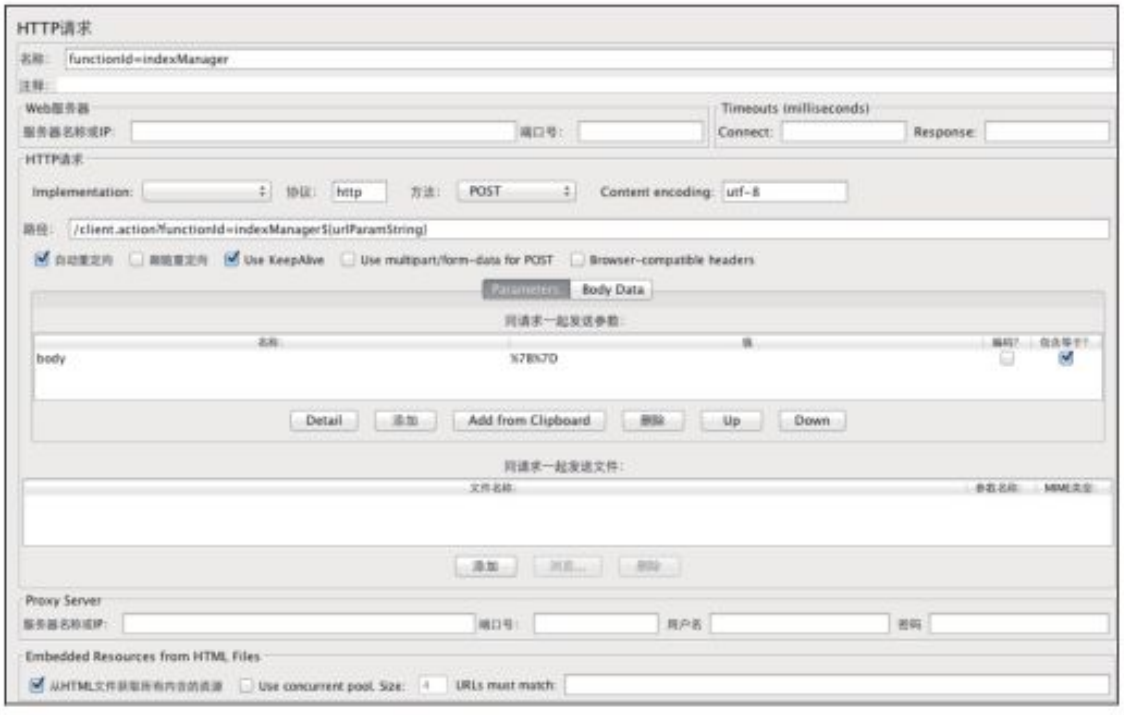


图 2-2 JMeter 中 HTTP 请求的参数填写界面

除此之外，还提供了 Cookie 管理、Cache 管理、消息头和授权管理等辅助功能，如图 2-3 所示。

图 2-3　JMeter 中对 HTTP 请求提供的辅助功能

3. 其他非直接支持的协议可以通过扩展方式实现

某些实际项目中，可能在 App 和服务端通信上采用非上述标准协议之外的其他协议，包括一些私有协议。对于这种情况，可以通过 JMeter 提供的通用组件来编写代码扩展，或者更进一步编写 JMeter 插件，类似于一个新的 Sampler。接下来我们通过一个例子看看如何通过现有组件来扩展。

对于其他私有通信协议的接口，我们可以通过 JMeter 的 OS Process Sampler 来进行桥接和测试，如图 2-4 所示。OS Process Sampler 可以用来启动一个可执行程序，由于是通过命令行方式启动，所以我们可以用任何语言编写一个测试用的可执行程序。在该可执行程序中调用我们的接口，调用完成后可以做简单的解析判断输出文本信息，也可以把返回的原始数据输出而交由 JMeter 做后续解析判断。具体做法是往该进程的标准输出流写入数据。之后在 JMeter 中即可读取这些数据。

以 C# 语言为例，在测试程序中我们可以用下列代码输出文本到标准输出流：

```
Console.WriteLine("[!Error!]");
```

在 JMeter 中我们可以使用 BeanShell PostProcessor 来处理可执行文件输出，如图 2-5 所示。

使用 BeanShell PostProcessor 来处理可执行文件输出，BeanShell PostProcessor 中获取输出的关键代码如下：

```
var res=prev.getResponseDataAsString();
var hasErr=res.contains("[!Error!]");
if(hasErr)
{
//这里可以增加断言或者进行变量赋值等常用操作
}
```

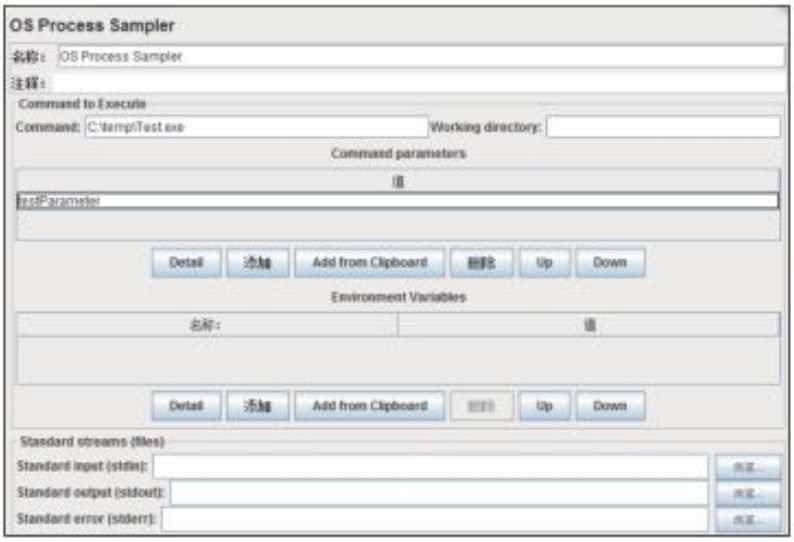


图 2-4　在 OS Process Sampler 中设置可执行文件路径和参数

图 2-5　在 JMeter 中使用 BeanShell PostProcessor

通过上述例子可以看到，测试用可执行文件的输出与 JMeter 的输入数据需要有一定的约定格式。该数据格式在项目中可以根据实际情况自由设计。

4. 支持丰富的断言

对于接口测试而言，断言是一个非常重要的功能，而且实际项目中，可能需要比较复杂的断言方式来判断结果是否正确。

图 2-6 所示是 JMeter 提供的响应断言，可以看出，其支持的检查内容比较多，包括响应代码、响应头和 Body 内容等方面，检查方式也支持多种不同的规则。

5. 支持内嵌自定义脚本

实际中，一个功能的自动化可能需要多个步骤，包含调用多个接口来完成。而多个接口之间有一些逻辑管理，比如后面的接口依赖前面接口的响应数据，甚至数据需要做一些处理才能为后面的接口所用。针对这种情况，可以通过 JMeter 内嵌支持的自定义脚本来实现，可以使用 JavaScript 和 Java 等语言。

图 2-6　在 JMeter 中使用响应断言

下面我们以 JavaScript 语言为例，通过一个实际的例子来看看，如图 2-7 所示。

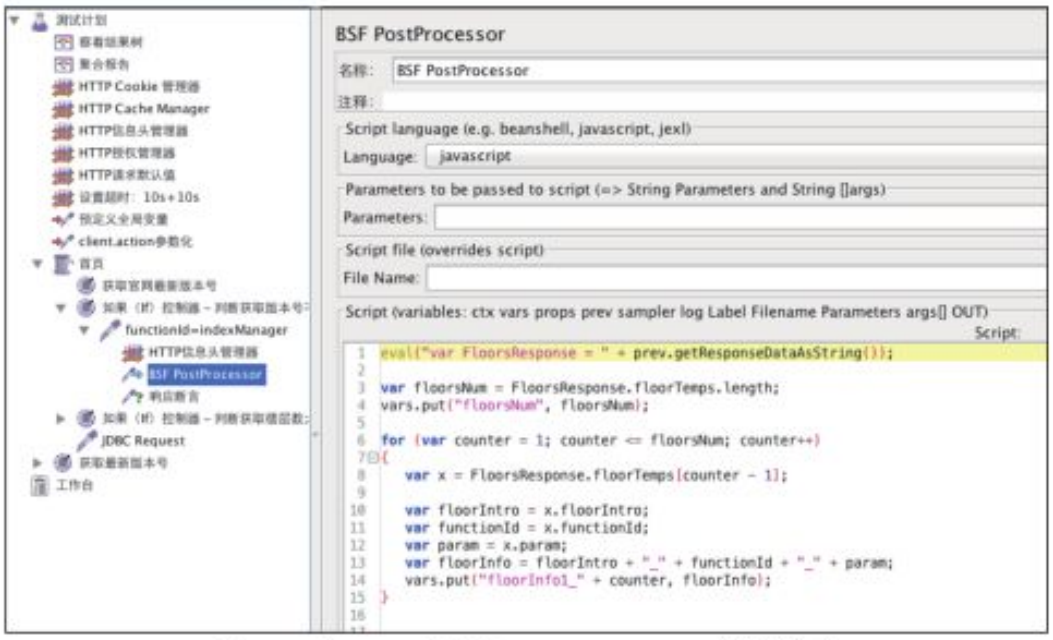


图 2-7　在 JMeter 中通过 BSF PostProcessor 来内嵌脚本

这个例子是针对一个电商 App 的首页楼层布局的检查，前面的接口返回了各个楼层的信息，这里的 BSF PostProcessor 里面的 JS 脚本以前面接口的返回作为参数，解析出其中各个楼层的信息，然后存入到数组中。之后后续的接口就可以读取这些变量，逐个地访问这些楼层的信息。

6. 可以直连 DB 检查数据

在实际的接口自动化中，除了通过接口层面来检查执行结果，有些不能在接口中返

回的，或者接口中信息比较少的，也可以通过直接查询 DB 的方式来检查数据和结果。在 JMeter 中可以通过 JDBC Request 这个 Sampler 来实现。图 2-8 所示是一个具体的例子。

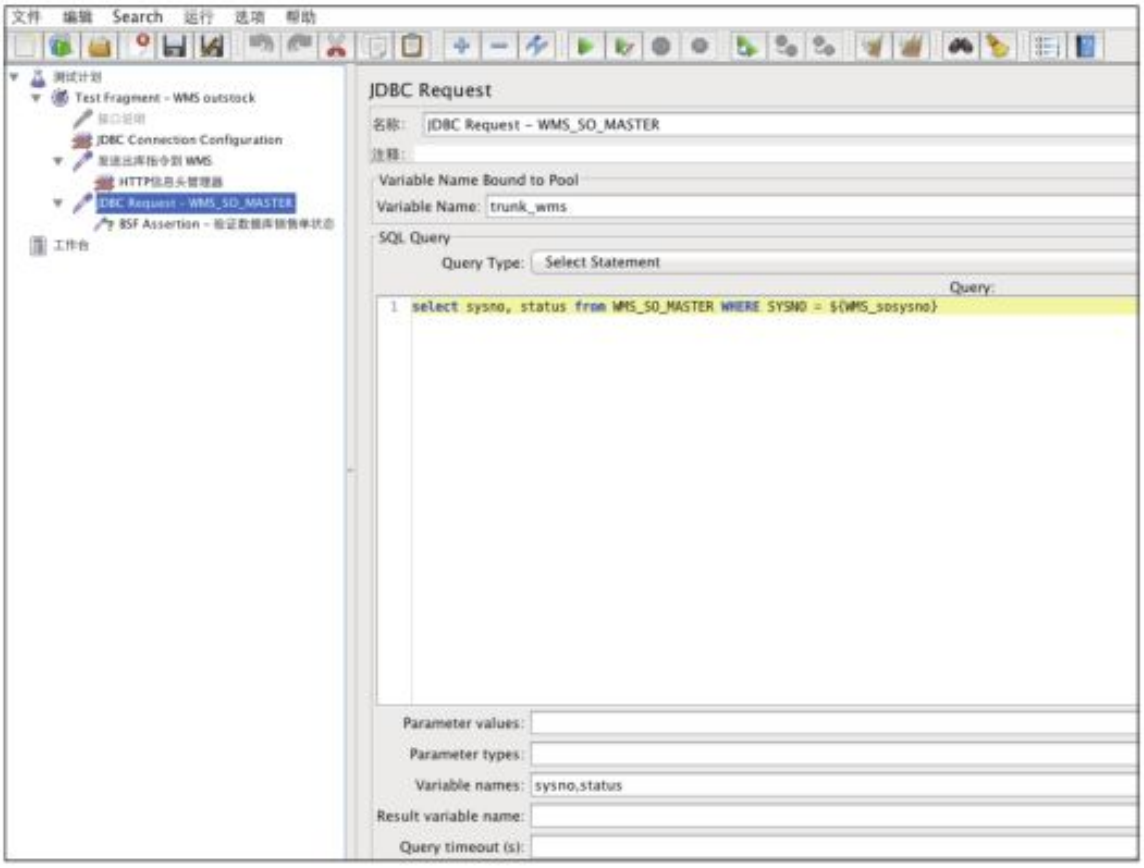


图 2-8　在 JMeter 中通过 JDBC 来操作数据库

可以直接在其中编写 SQL 脚本来访问对应 DB 中的数据，获取到数据后可以存入变量做进一步的断言和处理。

7. 可以嵌入执行第三方命令行

可以使用 OS Process Sampler 和 BeanShell Sampler 来执行一些外部的命令、用户环境和数据的准备等功能。

8. 文本的输入和输出

JMeter 的输入和输出都是文本的形式，包括界面编辑的配置存成 jmx 文本，需要的测试数据也可以通过文本的方式提供，另外，执行之后的结果也是存成文本的。这样的方式，

方便了后续其他脚本的处理和结果的解析。

9. 图形界面和命令行启动执行

JMeter 本身提供图形界面，极大地方便了用例制作和调试，不但提供请求配置和脚本编写的界面，也可以用于执行和调试，并有对应的多种形式的报告，让用例制作和调试的过程变得简单。当调试完成后，可以用命令行的方式来执行 JMeter 脚本，而不需要打开 GUI，也方便自动化框架的封装。

10. 工具本身非常稳定

最后一点也是一个很重要的方面，因为自动化会比较频繁的执行，而且可能执行的时间比较长，所以工具和框架本身的稳定性也非常重要。JMeter 作为一个开源工具，有非常广泛的用户基础，得到过比较充分的验证。新版本也在持续开发中，所以稳定性和可维护性方面比较有保障。

还有更多细节的功能这里不一一介绍了，有了上述这些特性后，就可以用于接口自动化的构建了。如果独自开发上面提到的这些功能，并在实际使用中稳定下来需要花费比较大的代价。借助已有的开源工作，围绕其做二次开发，是下面的自动化方案可以做到比较轻量级的主要原因。

2.1.2　基于 JMeter 的轻量接口自动化实践

有了上面这些基本的功能，接下来需要考虑如何整合成一套完整的方案，包括考虑用例的可复用和可扩展。

实践中我们的基本思路如下：

1. 用例的分层

为了更好的复用和管理，我们对用例的层次进行一些划分，并给出名称的定义。单次接口请求是一个最小粒度的操作，比如下面示例中的一次 HTTP 请求，在以下我们称之为 CGI。CGI 这个词的本意是通用网关接口，由于习惯的原因，常常用来代称单个业务接口。Function 是一个对外有逻辑意义的请求组合，比如提交订单、审核订单。TestCase 是一个成品，TestSuite 是一个用例的集合。基本的组织方式如图 2-9 所示。

通过这样的分层，整体逻辑就比较清楚了，CGI 对于一个具体的接口是最小粒度的元素，如果接口变化了，只需要修改对应的 CGI 用例。Function 层面是一个有逻辑意义的功能，可能需要多个 CGI 共同完成，相当于做了一层封装，为后面构建用例打下基础。

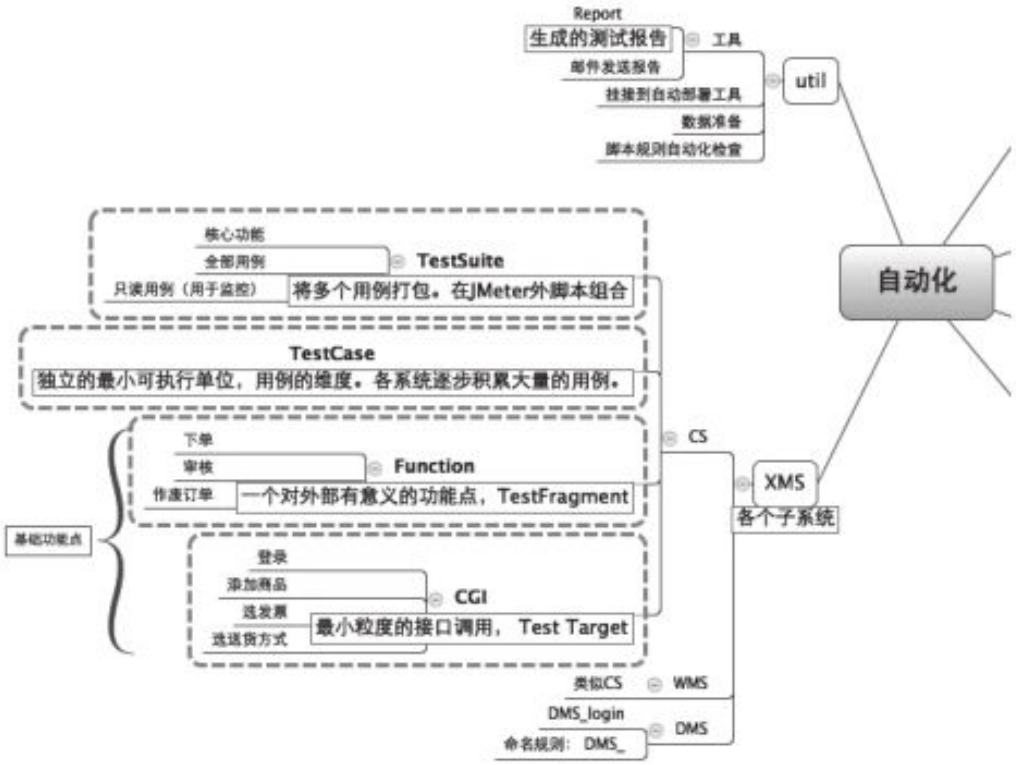


图 2-9　自动化用例的层次划分

每个子系统的目录结构如图 2-10 所示。

图 2-10　自动化用例的目录结构

这里有个需要注意的小地方，为了部署的时候更灵活，希望脚本中互相引用的文件是相对的路径，但是 JMeter 支持不太好，默认的根目录是 bin，所以简单的做法是把整个自动化用例的目录复制到 JMeter 的 bin 下面。

2. 完整的自动化用例结构

因为我们有多个系统，对应多个测试小组，大家各有专注，而整个业务又是一个长链条。Function 层就是我们复用的基础，里面包含了对 CGI 层接口的调用。注意一点，JMeter 2.10 开始建议用 TestFragment 来组织，而 TestFragment 是不能直接执行的，只是些积木，到了 TestCase 层面再用 ThreadGroup 线程组才可以执行。图 2-11 是一个 Test Fragment 的示例。

图 2-11　Test Fragment 示例

图 2-12 所示是一个完整的 TestCase，包含了本系统和其他系统的一些 Function 步骤。

实际多人协作的过程汇总，用例细节有很多需要规范的地方，比如命名规范，这样便于多人协作。

3. 数据的输出

JMeter 里面配置和编写用例执行完了之后，需要拿到输出的结果，外层的自动化框架才能进行解析和结果的输出。这里我们通过添加一个“察看结果树”的监听器，将所有的执行结果输出到一个文本文件里面，如图 2-13 所示。为了解析方便，通过变量指定了一个确定的文件名，数据结果方面选择了所有的列，便于得到完整的结果并处理。

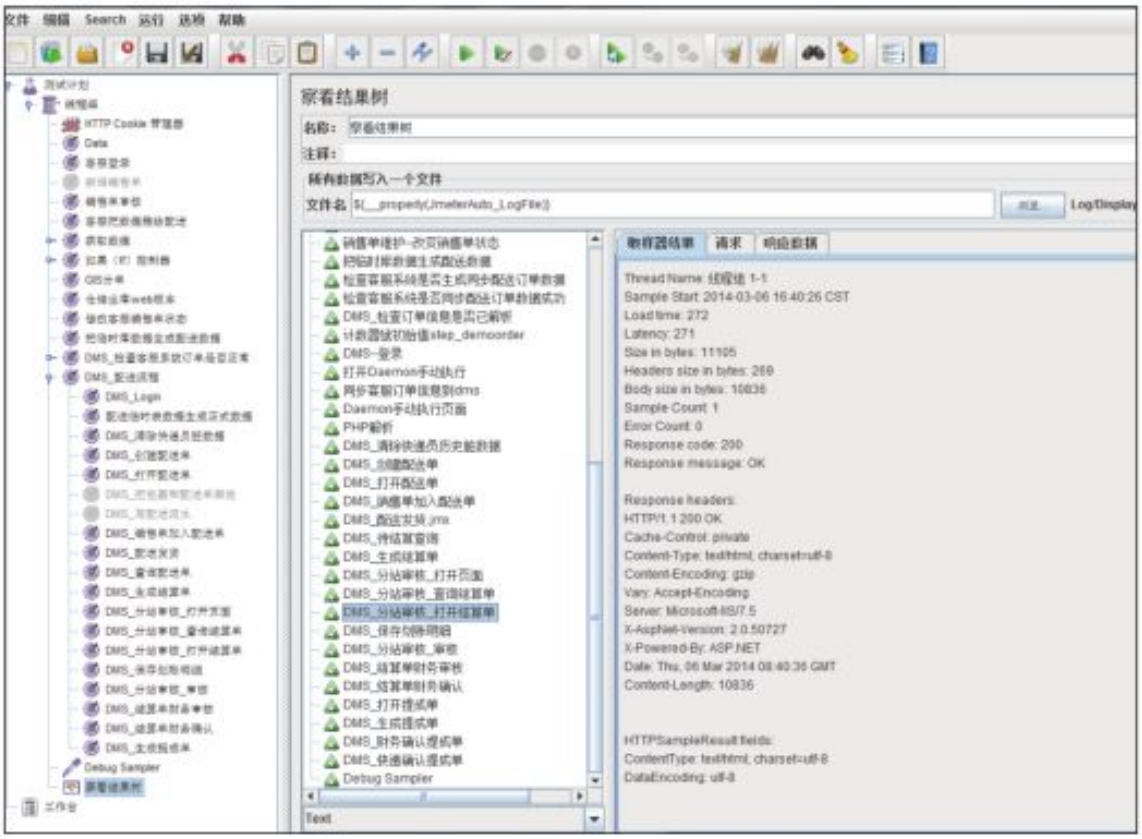


图 2-12　完整的自动化用例结构

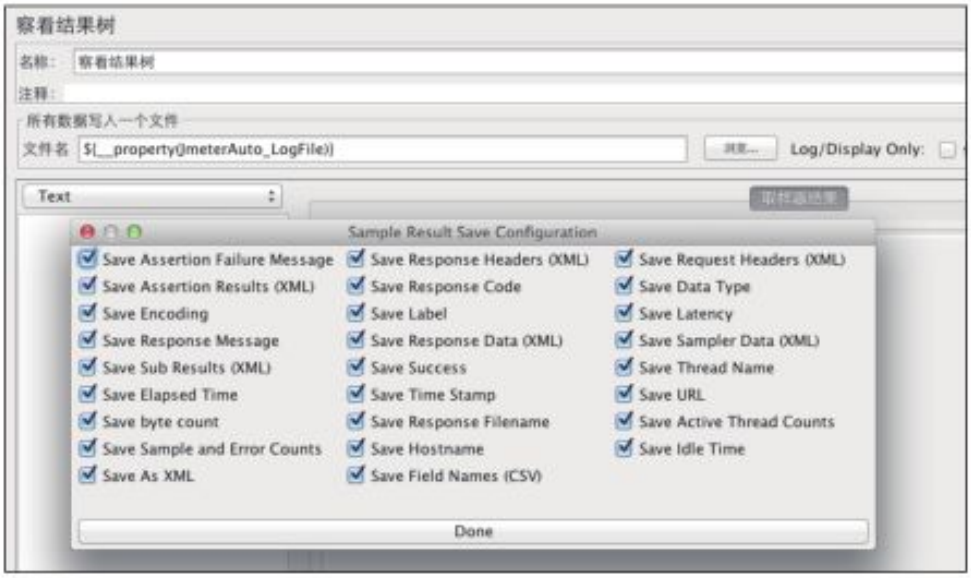


图 2-13　JMeter 中察看结果树的选型

4. 自动化的报告

基于上述步骤输出的文本结果，可以编写一个解析器，把结果解析成比较易于阅读的方式。然后生成对应的报告，每次执行完成之后自动地发邮件报告出来。图 2-14 是一个自动化测试报告的例子。

图 2-14　自动化测试报告邮件样本

5. 用例执行细节查询

除了上面汇总之后的结果报告，还需要一个地方能看到用例执行的细节，帮助定位问题。实际中我们构建了一个 Web 平台，测试人员可以在上面看到每次测试的结果，以及每个用例的执行细节。

为了更好地定位问题，需要把每个接口执行的细节也暴露出来，点击详情可以看到调用的情况，包括 request 和 response 的数据，如图 2-15 所示。

到这里为止，一个比较完整的可以在实际项目中应用的自动化框架就完成了。在实际的运作过程中，可以对业务测试人员进行一些 JMeter 方面的培训，让大家熟悉如何构造接口的请求，以及断言处理等模块，然后就可以编写对应的用例了。当然，如果多个人来准备测试用例，也需要知道哪些 CGI 和 Function 级别的封装是已有的，可以复用。

图 2-15 查看单个接口的执行详情

可见，除了 JMeter 以外的框架部分的开发工作外，工作量不是很大，很多较小规模的测试团队也可以承受。综合来看，基于这样的方案可以快速地把一个可用的自动化系统构建起来。

上述方案在一个项目中实践的情况如下：

1）覆盖了 4 个大的系统，200 多个 CGI 接口，以及 100 多个功能点。每个系统在构建后快速地通过以上自动化用例来进行回归，并发出邮件报告，框架整体比较稳定。

2）整个过程，不算制作用例的时间，我们实际投入的测试开发的人力约为一个人 / 月。

3）大部分业务测试人员都参与到了用例的制作，提升了对业务逻辑的理解，并且对部分人员来说，也学习了 HTTP 协议等基础知识，并编写了少量的脚本，提升了业务测试人员的自动化和测试开发能力。

总体来说，达到了轻量化来构建接口自动化的目标。

除了以上好处，也有一些不方便的地方需要指出：

- JMeter 在包含其他的 jmx 脚本后，不能直接在界面显示加载的内容，所以看不到被包含的脚本里面的步骤，调试的时候不方便。不过 JMeter 可以支持多个实例同时运行，所以在编写和调试阶段，可以同时打开多个 JMeter 窗口，一定程度上可以缓解这个问题。

- 最后的测试报告里面，不能控制显示结果的层次，是直接展开成每个接口的结果，当用例的步骤过多时查看起来不是特别方便。

除了技术层面的考虑，自动化执行的过程中还有几个建议值得考虑：

1）一定要强挂钩到测试和发布的环节。这一点看起来没那么重要，但是如果不希望自动化测试成为花瓶，必须要这样做。互联网产品的节奏非常快，如果自动化不能实际地在项目中发挥效果，就很容易被荒废。目前来看最合适的点是在每次自动部署后快速地把自动化用例执行一遍，这要在手工测试开始之前。这也是持续集成的思路，可以快速地发挥自动化的价值。

2）报告也需要自动化地发出来，并且邮件抄送给相关的开发人员、测试人员和团队负责人。这样可以让大家快速地关注到结果。

3）非 100% 成功的都要跟进。宁愿少而精，就像“破窗理论”指出的，如果能容忍一个用例失败，就会有 2 个、3 个，也会让自动化慢慢失去意义。我们的做法是只要有失败，对应系统的测试负责人员需要进行定位并邮件回复出问题原因和处理措施。

4）需要关注用例的细节。测试团队的负责人需要去关注用例的质量，而不只是用例的数量和执行情况，比如断言做到什么程度，哪些自动化数据是写死的，哪些参数化了，功能之间的复用情况，这些都是影响整个自动化的稳定性和可维护性的具体方面。

2.2　App UI 层面的自动化

除了上面介绍的基于接口的自动化，App UI 层面的自动化也是一个重要的自动化技术。可以帮助快速地进行 App 功能的回归。考虑到功能的变动和维护的代价，实际中投入产出比较高的方式是针对相对稳定的功能进行快速的回归。也可以和后面讨论的持续集成结合，做新构建的验证。除了功能的自动化验证之外，UI 自动化技术还有一些其他的价值，比如第 4 章专项测试中介绍了使用 UI 自动化技术和云测试平台来构造一套高效的兼容性测试方案，以及基于模糊测试思路和 UI 自动化建立的 App 稳定性测试平台。这些都是以 App UI 自动化技术为前提的。

另外，需要指出的是，对于 UI 自动化应该有一个合理的期望。很多刚接触 App 测试的人或者不了解 App UI 自动化的管理人员，可能提起 App 的测试技术，觉得最主要的就是 App 的自动化，而主要讨论的就是 UI 的自动化技术。其实在经历过一些大的项目之后，观念可能会有些不同。基于目前多个团队的实践来看，如果对于一个在快速发展中的 App，UI 自动化可能更适合一些基础功能的回归，而不是替代手工的功能测试，特别是对于新的功能点。

接下来讨论如何进行 App UI 的自动化测试，包括 Android 和 iOS 的做法。

2.2.1 Android 的 UI 自动化技术

本节介绍 Android UI 自动化测试的一些基本的实现方式。由于各个项目的产品特性不同，这里不涉及 UI 自动化框架的封装。

2.2.1.1 UI Automator Viewer

在进行 Android UI 自动化测试之前，作为测试人员，应当了解待测 App 的 UI，包括使用了哪些控件、控件的类名、控件的 id，等等。获知这些信息之后才能写测试脚本进行自动化测试。

Android SDK 提供了一个 UI Automator Viewer 工具来帮助我们获取这些信息。（需要安装 Android SDK Tools、Revision 21 或以上，Android SDK Platform、API 16 或以上）这个工具提供了一个可视化界面展示当前设备上的各个控件的属性。按照下列步骤使用这个工具：

- 链接 Android 设备到计算机。
- 打开 <Android SDK 目录 >/tools/。
- 运行命令：uiautomatorviewer
- 点击“Device Screenshot”按钮，鼠标悬停到某一 UI 元素上面即可查看对应控件的详细属性，如图 2-16 所示。

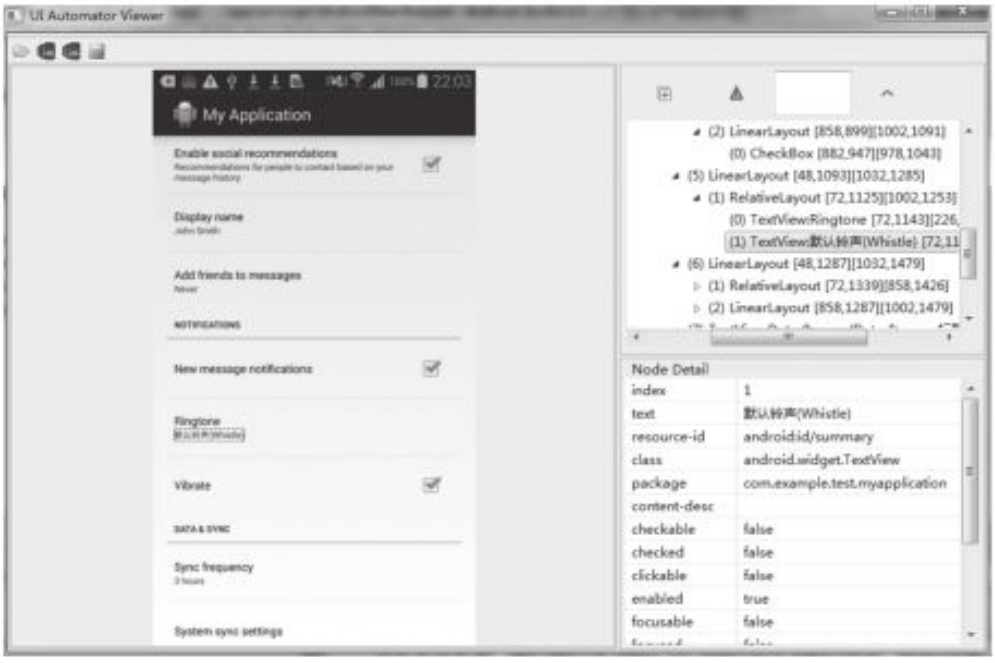


图 2-16 使用 UI Automator Viewer 工具浏览界面元素

注意，如果你在计算机上连接了多个设备，在运行 UI Automator Viewer 前需要通过设置环境变量指定需要查看的 UI 设备。具体方法如下：

- 通过命令 adb devices 找出设备序列号。
- 设置环境变量。
  - Windows

    set ANDROID_SERIAL=< 上一步获取的设备序列号 >
  - Linux

    export ANDROID_SERIAL=< 上一步获取的设备序列号 >

此外，UI Automator Viewer 还提供了一个功能，可以查看 UIAutomator 测试框架可能不支持的控件。只需要在界面上点击“ Toggle NAF Nodes”按钮（右上方感叹号标记的按钮）即可查看。如果在使用 UIAutomator 过程中发现某个控件不能被自动化驱动，可以对照 UI Automator Viewer 工具的结果排查问题。

在熟悉了待测 App 的 UI 以后，我们需要选择一项或者多项测试方法来进行 UI 自动化测试。Android 的 UI 自动化测试方法从技术角度来说，大致可以分为以下几种：

- Instrumentation。
- UIAutomator。
- 基于 Instrumentation/ UIAutomator 的封装。
- 基于系统事件的自动化测试。
- 基于图像识别的自动化测试。

下面分别介绍这几种方法。

2.2.1.2　Android JUnit 测试

Instrumentation 和 UIAutomator 都是基于 JUnit 的，所以在介绍它们之前，先以 Android Studio 1.0 为例，介绍如何进行 Android 的 JUnit 测试。

1）我们需要在项目源代码目录下创建一个包和测试类，如图 2-17 所示。

图 2-17　在项目中创建测试类

这个测试的代码很简单，判断变量 1 和变量 2 是否相等，如果不相等，断言出错。当然运行后的结果必然是出错。这里需要注意，由于是基于 Junit，测试方法名字必须以 test 开头，否则 JUnit 无法辨识。

2）点击 Run->Edit Configurations。在弹出窗口上点击左上角的“Add New Configurations”按钮（绿色十字图标）。选择 Android Tests，如图 2-18 所示。

图 2-18　配置选择界面

3）按图 2-19 的配置设置新建的测试。

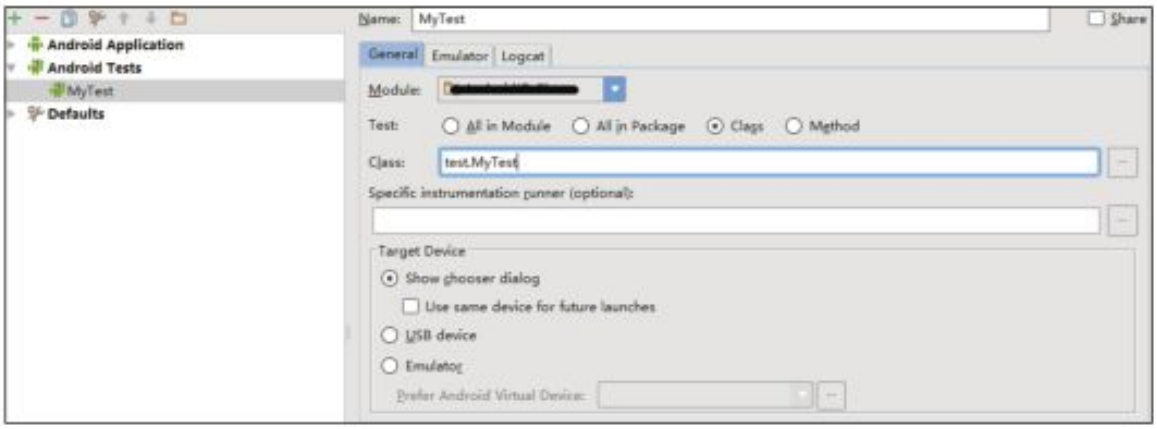


图 2-19　新的测试配置

其中 Modules 为源代码所在的 Module。Test 部分可以自由设置寻找测试类的范围，这里我们手动指定了测试的类。

4）在工具条上选择刚才新建的测试并点击运行，如图 2-20 所示。

图 2-20　运行测试

5）选择设备后等待执行结果，最后结果如图 2-21 所示。

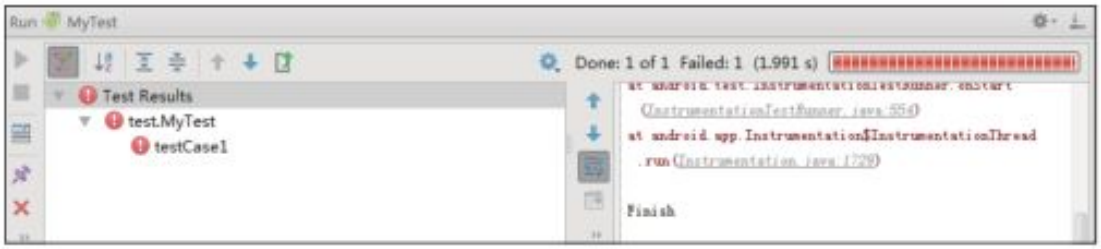


图 2-21　测试结果界面

从左边栏可以看到哪个测试方法出错，从右边栏可以看详细的调用栈。经过上面的步骤我们已经可以对 App 进行单元测试了。只需在测试类中引用相应的待测试类并调用其方法即可。

1. Instrumentation 测试框架

Instrumentation 可用来进行黑盒和白盒测试，是谷歌早期推出的 UI 自动化测试框架，所以向后兼容性最好，可以用来测试安装在低版本的 Android 操作系统上的 App。但相对来说，对使用此方法的测试人员的技术要求较高。主要缺陷是此方法只能用来测试单个 App，涉及跨 App 交互的情况此方法不能支持。(例如，测试场景为：点击 App A 上的一个按钮将导致 App B 的界面展现到屏幕上，再点击 App B 上的一个按钮后进行断言判断。)

需要使用 Instrumentation 来进行 UI 自动化测试时。我们只需修改前面 JUnit 测试部分的测试类的代码即可，其他步骤都相同。

一个简单的使用 Instrumentation 进行 UI 自动化测试的测试类例子如下：

```
public class MyTest extends ActivityInstrumentationTestCase2<SettingsActivity>
{
 SettingsActivity mMyActivity;
 protected void setUp() throws Exception {
     super.setUp();
      mMyActivity = getActivity();
 }
 protected void tearDown() throws Exception {  super.tearDown();}
 public MyTest() {
     super(SettingsActivity.class);
 }

 public void testCase1()
 {
   ActionBar mMyBar=  mMyActivity.getActionBar();
   String title= mMyBar.getTitle().toString();
   assertEquals(title, "My Application");
 }
}
```

我们的测试类需要继承 ActivityInstrumentationTestCase2<T>，T 为待测试的 Activity 类型。有两个方法值得关注，一个是 setUp 方法。这个方法在测试开始的时候会被调用一次。我们可以在这个方法中进行初始化操作。上述代码在这个方法中获取了当前 Activity 对象的引用。另外一个方法是 tearDown 方法。这个方法在测试结束的时候会被调用一次。如有需要，我们可以在这个方法中添加代码。

跟之前的 JUnit 测试类一样，我们的测试方法需要以 test 开头。在测试方法 testCase1 中，我们获取页面头部的文字描述，进行了校验。当页面标题不是“My Application”的时候，此测试失败。

上面的代码只是一个简单的例子，我们可以做更多的事情，例如使用 Activity 类的 findViewById 方法获取需要的控件并做一些操作，如按钮点击，给输入框添加输入等。在此不一一赘述。

2. UIAutomator 测试框架

UIAutomator 是谷歌新推出的 UI 自动化测试框架，需要 API level 16 和以上的操作系统才能使用。使用它能够测试跨 App 交互的场景，但是它也有缺点。除了兼容性外，对一些控件也很难定位到。其中最大的缺点是不支持 WebView 的自动化测试。如果你的 App 中大量使用了 WebView，并且需要对其进行 UI 自动化测试，那么你可能不得不选择 Instrumentation。

下面以 Android Studio 1.0 为例说明如何进行 UIAutomator 自动化测试：

1）在 <Android SDK 安装目录 >\platforms\android-XX 下面找到 uiautomator.jar。

2）将 uiautomator.jar 复制到项目的 app/libs 目录下。

3）测试项目 Module 的依赖项增加 libs 下的所有 jar：

```
dependencies {
    compile fileTree(include: ['*.jar'], dir: 'libs')
    compile 'com.android.support:support-v4:21.0.3'
}
```

4）假设我们需要测试的 Activity 是 SettingsActivity（使用 Android Studio 1.0 自带项目模板 Settings Activity 创建）。增加测试类如图 2-22 所示。

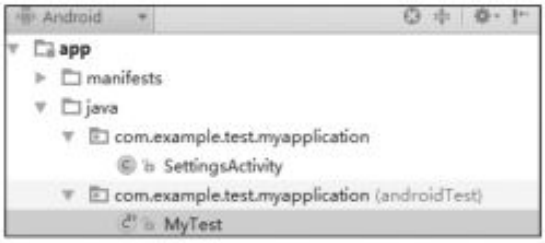


图 2-22　为 SettingsActivity 增加测试类

5）确保待测 App 已经安装到 Android 手机上。

6）修改测试类代码为：

```
public class MyTest extends UiAutomatorTestCase  {
    String packageName="com.example.test.myapplication";
    String activityName="SettingsActivity";
    protected void setUp() throws Exception {
       getUiDevice().pressHome();

       Runtime.getRuntime().exec("am start -a android.intent.action.MAIN -n
           "+packageName+"/."+activityName);
       UiObject mainUi=new UiObject(new UiSelector().packageName(packageName));
       assertTrue("App not started", mainUi.waitForExists(3000));
       super.setUp();

 }
protected void tearDown() throws Exception
{
super.tearDown();
}
    public void testCase1()
    {
       UiObject uiToTest=new UiObject(new UiSelector().className("android.
           widget.CheckBox").instance(1));
       try
       {
          uiToTest.click();
          getUiDevice().waitForIdle(3000);
          UiObject uiToAssert=new UiObject(new UiSelector().text("Ringtone").
              instance(0));
          assertFalse("CheckBox not work", uiToAssert.isEnabled());
       }
       catch(UiObjectNotFoundException ex)
       {
            assertTrue("Cannot find checkbox", false);
       }
   }
}
```

与 Instrumentation 类似，我们可以在 setUp 中写初始化的代码，在 tearDown 中写清理工作的代码。这里我们在 setUp 中进行了回到主界面的操作。以此作为测试的起点。这也是通常的做法。

另外，在初始化代码中，我们调用了 am 命令启动我们需要测试的 App 的 Activity。这个放在初始化的目的通常是为了节省测试的时间。一些情况下，启动 Activity 的工作只需要一次，我们可以写很多个测试方法来测试这个启动后的 Activity 的不同测试点。

在测试方法 testCase1 中，我们执行了如下的操作：

- 寻找第二个勾选框并点击。
- 判断 Ringtone 文本是否为不可用状态（外观变为灰色）。

这里有一些技巧。例如 getUiDevice().waitForIdle 可以保证 UI 变更后才执行后续代码，UiObjectNotFoundException 的捕捉等。

7）按上面“Android 的 JUnit 测试”描述的第 6 步执行后在选择设备界面时选择 Cancel。因为目前 Android Studio 1.0 不能很好地支持 UIAutomator。我们只用它来生成测试 apk 包。

8）项目的 build\outputs\apk 下可以找到带有 test 字样的 apk 包，如图 2-23 所示。

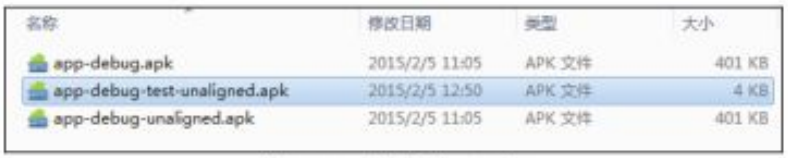


图 2-23 生成的 apk 包

9）将 Android 手机连接电脑，使用下列命令将 apk 包推送到手机上：

```
adb push app-debug-test-unaligned.apk /sdcard/test.apk
```

10）使用下列命令进行 UI Automator 测试：

```
adb shell uiautomator runtest /sdcard/test.apk
```

11）上述命令执行完毕后观察手机，能发现 UI 被自动进行了操作。等待操作完成后，查看测试结果，如图 2-24 所示。

```
INSTRUMENTATION_STATUS: numtests=1
INSTRUMENTATION_STATUS: stream=
com.example.test.myapplication.MyTest:
INSTRUMENTATION_STATUS: id=UiAutomatorTestRunner
INSTRUMENTATION_STATUS: test=testCase1
INSTRUMENTATION_STATUS: class=com.example.test.myapplication.MyTest
INSTRUMENTATION_STATUS: current=1
INSTRUMENTATION_STATUS_CODE: 1
INSTRUMENTATION_STATUS: numtests=1
INSTRUMENTATION_STATUS: stream=.
INSTRUMENTATION_STATUS: id=UiAutomatorTestRunner
INSTRUMENTATION_STATUS: test=testCase1
INSTRUMENTATION_STATUS: class=com.example.test.myapplication.MyTest
INSTRUMENTATION_STATUS: current=1
INSTRUMENTATION_STATUS_CODE: 0
INSTRUMENTATION_STATUS: stream=
Test results for WatcherResultPrinter=.
Time: 3.625

OK (1 test)
```

图 2-24 最终的测试结果

由于篇幅所限，更具体的 UIAutomator 测试方法不再进行介绍。读者可参考 Android 官方文档进行学习。

3. 基于 Instrumentation/ UIAutomator 的封装

除了上文提到的两个谷歌推出的原生 UI 自动化测试框架外，还有很多基于它们封装的 UI 自动化测试框架。其中 Robotium、Appium 是使用比较广泛的。

Robotium 其实严格来说算不上是一个测试框架，它只是一个类库，提供了更友好的定位控件的方法等，在使用 Instrumentation 进行自动化测试时，建议使用 Robotium 类库提升测试效率。

Appium 是一个重量级的测试框架，支持 iOS 和 Android 等平台的自动化测试，支持多语言编写测试脚本。在 Android 自动化测试的实现上，它也是基于 Instrumentation 和 UIAutomator 的。Appium 用 Ruby 写的测试脚本示例如图 2-25 所示。

Ruby　Java

```
cell_names = tags('android.widget.TextView').map { |cell|
cell.name }

cell_names[1..-1].each do |cell_name|
  wait { scroll_to_exact(cell_name).click }
  wait_true { ! exists { find_exact cell_name } }
  wait { back }
  wait { find_exact('Accessibility');
find_exact('Animation') }
end
```

图 2-25　Appium 的 Ruby 测试脚本

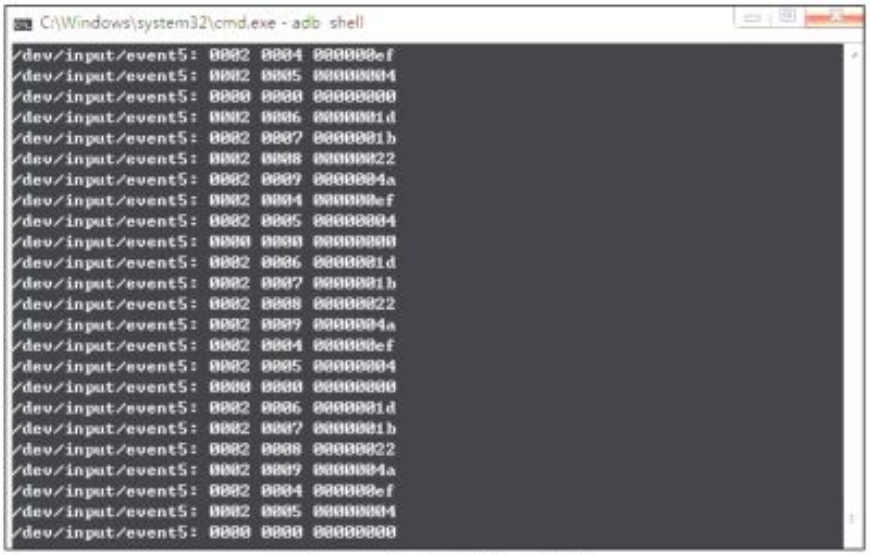
由于篇幅所限，本书不赘述它们具体的使用方法。有兴趣的读者可以自行搜索相关的资料。

2.2.1.3　基于系统事件的自动化测试

所有 Android 操作系统的输入事件都可以在进入 adb shell 后通过 getevent 获取（需要 su），相关示例如图 2-26 所示。

这些事件都会存储在 /dev/input/eventX 下。每个 eventX 代表一个输入外设。例如触摸屏、键盘等，而输出的每一行代表一个事件，例如某个键盘按下，等等（具体每个事件的三个部分代表什么含义请参考 Linux 相关资料）。因此，我们可以通过解析这些文件，来实现录制的功能。测试人员可以把自己的操作录制下来，用一种数据格式记录。

那么怎么回放录制的测试脚本呢？我们可以解析录制的数据，并在 adb shell 下使用 sendevent 命令向指定外设一个个发送事件，也可以在 adb shell 下用 input keyevent 命令来做。例如，在将 Android 手机连接计算机后打开一个记事本类程序，进入 adb shell，执行：



```
/dev/input/event5: 0002 0004 000000ef
/dev/input/event5: 0002 0005 00000004
/dev/input/event5: 0000 0000 00000000
/dev/input/event5: 0002 0006 0000001d
/dev/input/event5: 0002 0007 0000001b
/dev/input/event5: 0002 0008 00000022
/dev/input/event5: 0002 0009 0000004a
/dev/input/event5: 0002 0004 000000ef
/dev/input/event5: 0002 0005 00000004
/dev/input/event5: 0000 0000 00000000
/dev/input/event5: 0002 0006 0000001d
/dev/input/event5: 0002 0007 0000001b
/dev/input/event5: 0002 0008 00000022
/dev/input/event5: 0002 0009 0000004a
/dev/input/event5: 0002 0004 000000ef
/dev/input/event5: 0002 0005 00000004
/dev/input/event5: 0000 0000 00000000
/dev/input/event5: 0002 0006 0000001d
/dev/input/event5: 0002 0007 0000001b
/dev/input/event5: 0002 0008 00000022
/dev/input/event5: 0002 0009 0000004a
/dev/input/event5: 0002 0004 000000ef
/dev/input/event5: 0002 0005 00000004
/dev/input/event5: 0000 0000 00000000
```

图 2-26　获取系统输入事件

```
Input keyevent 34
```

你会发现效果等同于输入一个 F 键。

基于系统的事件无法对界面上的元素做判断，只能进行一些驱动类型的工作。因此这个方法不能单独使用来进行自动化测试。断言部分需要配合 Instrumentation/UIAutomator。

此方法的缺陷是：

- □ 不同的设备有不同的分辨率。在 A 设备下录制的脚本在 B 设备下执行可能会产生问题。
- □ 不同设备的 eventX 可能代表不同外设。

这些都是在使用这个方法进行自动化测试的时候需要考虑的问题。

2.2.1.4　基于图像识别的自动化测试

近年来图像识别技术有了一定的发展。因此测试人员也开始研究是否可以将此技术应用到自动化测试中。简单的想法是，如果一个 App 的各个元素外观变化不大，而位置变化较为频繁，或者内部实现（例如 UI 控件类的变化）改变较为频繁，使用图像识别来写测试脚本的稳定性可能会较高。另外如果断言是基于图像的，那么使用图像识别技术来做自动化测试更为自然和方便。

测试方法举例：

1）测试人员写测试脚本，指定需要操作哪些元素（指定图片相关的特征），并说明怎么操作。

2）使用 adb screencap 命令截图。

3）使用图像识别技术来识别出各个元素的位置。

4）解析测试脚本，找出下个操作步骤。如果测试结束，则退出生成报告。

5）采用某方法进行相应的操作（通过系统事件、机器人手臂等方式驱动），或者进行基于图像识别的断言。

6）返回步骤 2）。

但是由于使用此方法的优势只在特定场景能够体现，局限性较大，再加上图像识别准确率的问题，目前还没有被广泛接受和应用的测试框架出现。在此提及也是抛砖引玉，希望有一天能够有一个出色的基于图像识别的 Android 测试框架问世。

2.2.2　iOS 的 UI 自动化技术

关于 iOS 的 UI 自动化，这里结合实例介绍两个方面的内容，一个是基于 Instrument 的 UI 自动化，另一个是目前比较常用的 Appium 框架。

2.2.2.1　基于 Instrument 的 iOS UI 自动化

对于 iOS 的 UI 自动化，Instrument 提供了最基本的自动化测试功能。我们可以通过 Automation 工具实现基本的自动化测试需求。该工具支持真机和模拟器两种测试方式：

- 模拟器：执行速度快，无须证书，测试门槛较低，但对于一些特定功能如相机、跳转就无能为力了。
- 真机：App 的所有功能均能自动化实现，但需调试证书，且执行效率低。

下面用一个示例介绍整个使用过程。

1. 开始使用 Automation

下面我们以 Xcode 6 为例，介绍使用 Automation 的步骤。

1）选择要测试的 target，选定要编译安装的平台，若在真机上测试就选择真机设备，同时指定好调试证书及对应的 provision 文件。若在模拟器上测试就选择模拟器环境，我们这边指定 iPhone6 模拟器，如图 2-27 所示。

2）在刚才选择界面，点击 Edit Scheme，指定 Profile 选项以 Debug 模式编译，点击关闭，如图 2-28 所示。

3）点击 Product 菜单下的 Profile 选项来编译项目工程，如图 2-29 所示。

图 2-27 Xcode 设备选择

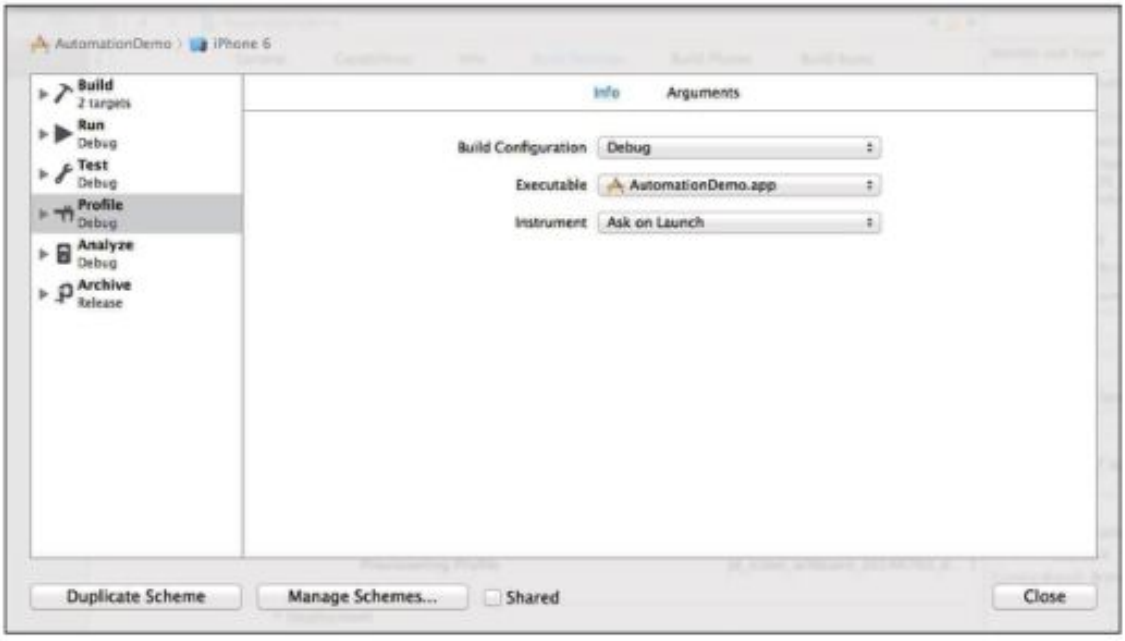


图 2-28 Scheme 设置界面

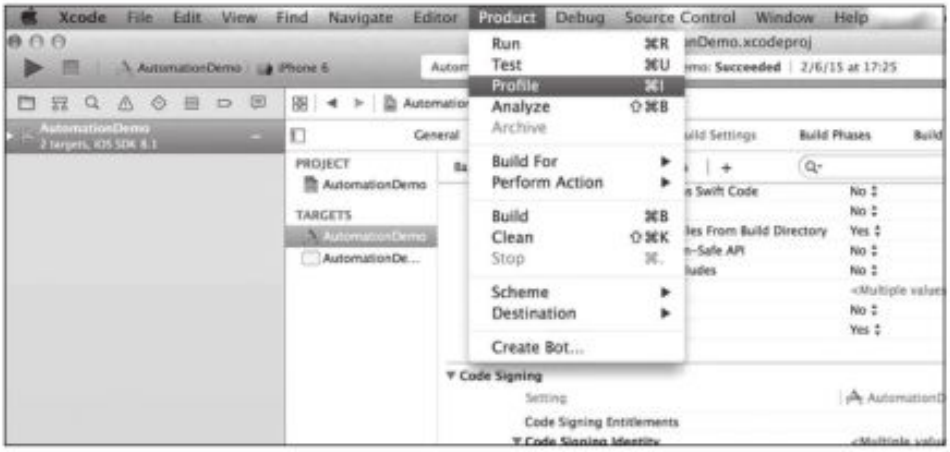


图 2-29　选择 Profile 模式编译项目工程

4）等待编译完成后，弹出 Instrument 界面，选择 Automation，如图 2-30 所示。

图 2-30　Instrument 主菜单界面

5）系统自动选择模拟器中的测试 App，如果要改成真机运行，修改界面上方的测试平台至真实设备，选择相应的 App 即可，如图 2-31 所示。

2. 录制脚本

我们可以通过录制的方式生成测试脚本，切换到 Script 界面，点击下方的录制按钮，App 在模拟器中就启动了，我们对 App 的操作，就会被自动录制成回放脚本，录制完成后，点击终止按钮，结束录制。点击运行按钮，就可以执行回放刚才录制的操作了，如

图 2-32 所示。

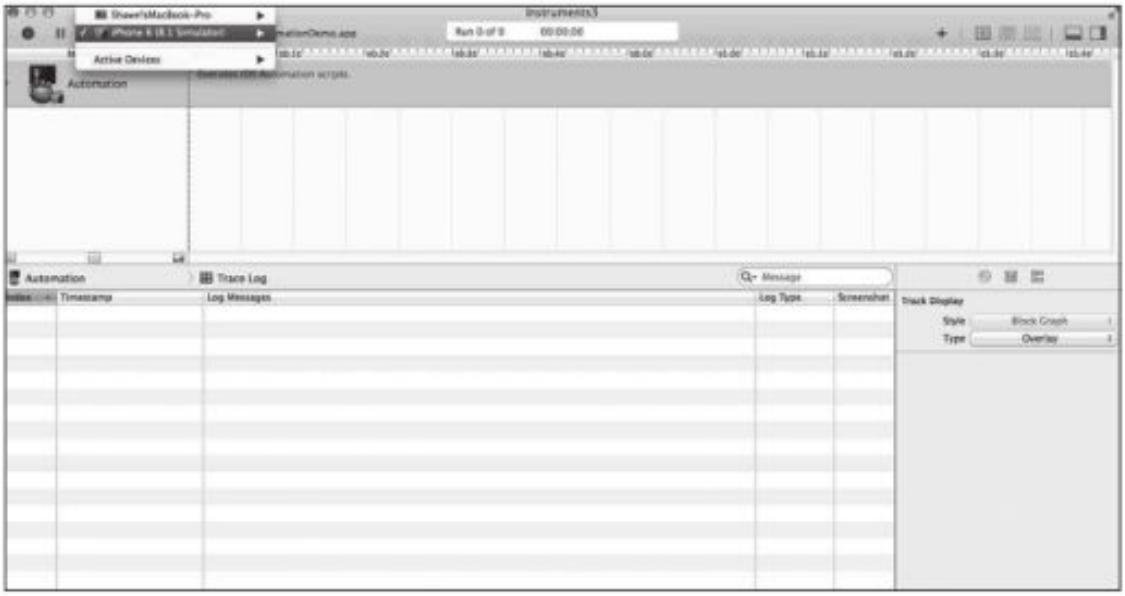


图 2-31　Automation 界面

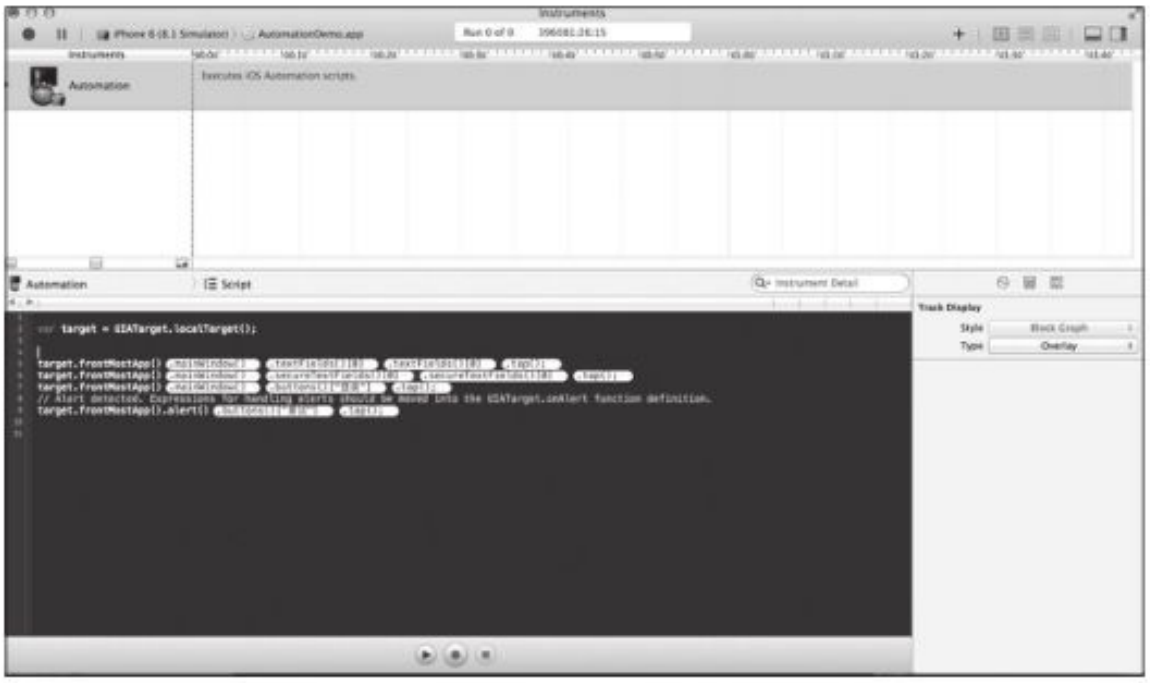


图 2-32　Automation 脚本录制编辑界面

3. 编写自己的脚本

录制的方式虽然很方便，但是缺乏灵活性。大多数时候，我们需要自己编写脚本。我们的测试 App 是一个很简单的登录界面，输入用户名 panxiaoming，密码 :Ilovetest，就会登录成功，弹出一个欢迎界面，如图 2-33 所示。

图 2-33　测试 App 主界面

如果输入错误的密码，会提示错误信息，如图 2-34 所示。

图 2-34　登录失败界面

输入正确的密码，提示欢迎信息，如图 2-35 所示。

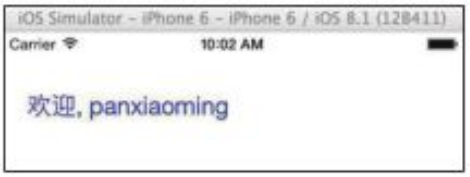


图 2-35　登录成功界面

我们可以通过 logElementTree 的方法查看 App 的 UI 元素布局。点击运行，在 Editor Log 界面中查看这些 UI 元素，如图 2-36 所示。

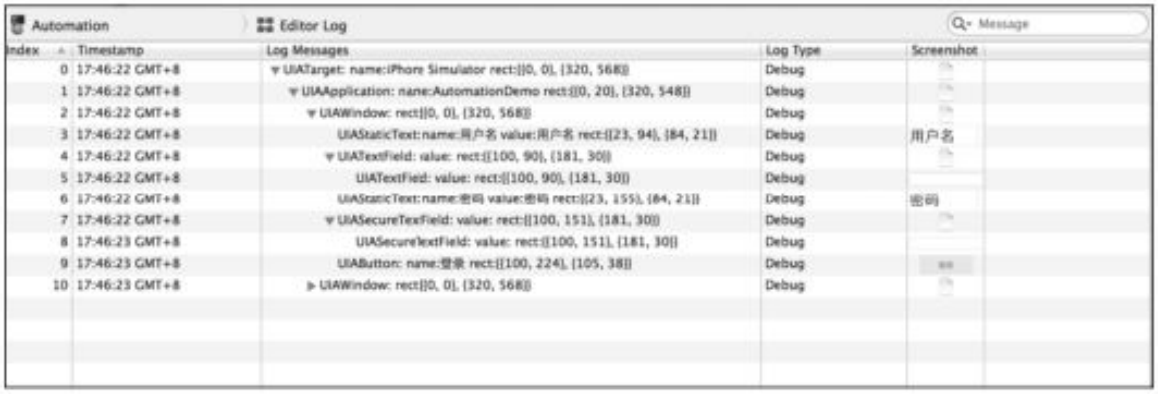


图 2-36　Editor Log 信息界面

Editor Log 很有用，在这里可以查看脚本中所有的 log 信息以及断言结果，同时它会自动记录点击操作和一些其他的操作，同时还有截图，方便查看。

于是我们就可以编写一个正常登录的测试脚本，脚本如图 2-37 所示。

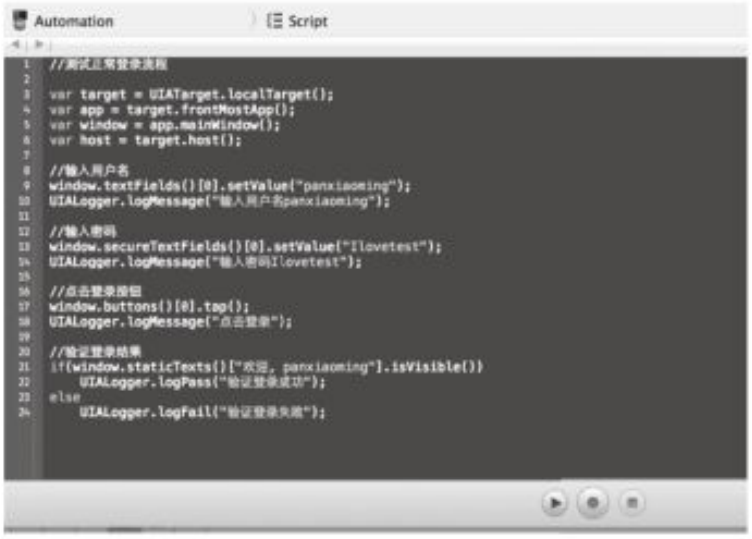


图 2-37　自动化脚本内容

执行结果如图 2-38 所示。

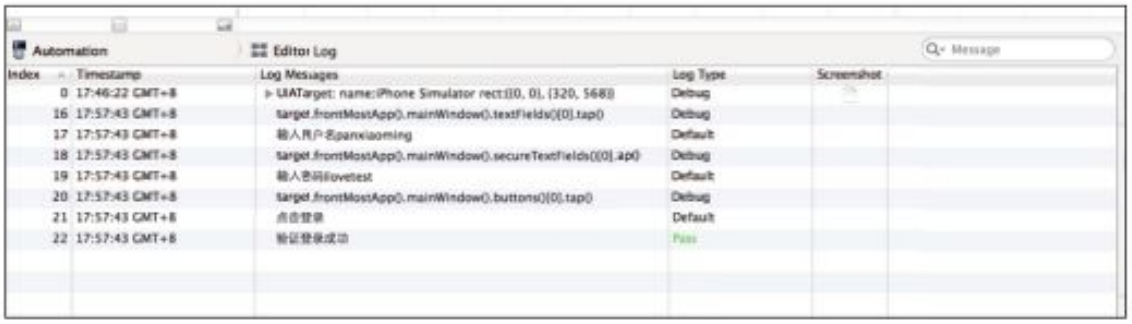


图 2-38　执行 Log 结果

我们看到在自己编写的脚本中加入逻辑和判断，使得脚本的可靠性更强，同时 log 信息可以帮助我们去分析定位问题，这种方式要比简单录制强上不少，脚本的语句类似 JavaScript，可以到苹果官网查看具体语法和 API，网址为：(https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/UsingtheAutomationInstrument/UsingtheAutomationInstrument.html)。

4. Automation 的高级用法

我们完全可以通过 shell 脚本去驱动执行 Automation 的测试用例，通过定时执行 shell 脚本来完成我们的自动化测试任务，具体方法如下：

```
instruments -t <automationTracetemplate路径> <模拟器App路径> -e UIASCRIPT
    <testcase路径>
```

我们只需要在自动化测试代码中使用 performTaskWithPathArgumentsTimeout 的方法把断言执行结果通过 shell 脚本写到外部文件中，就可以查看所有 case 的执行结果了。

5. Automation 的总结

Instrument 的 Automation 总体来说还是比较强大的，首先它不需要对 App 作任何修改、插桩，保证了 App 的原生功能。其次对 iOS 的控件支持非常好，同时支持暂停、截图等一系列功能，基本上能够满足我们对自动化测试的需要。不过它也有不少缺点，首先它没有 case 管理概念，其次是它的扩展性不是很好，还有就是它的执行经常会中断。

2.2.2.2　自动化测试框架之 Appium

Appium 是当前比较流行的一个自动化测试框架。它是一个开源的自动化测试框架，支持跨平台，支持原生和混合开发，支持真机和模拟器。它是一个 C/S 结构的设计，底层是基于 iOS 的 UIAutomation，它的特点是：

- 无需任何驱动桩的插入，可直接操作原 App。

- ❑ case 支持多种语言。
- ❑ 内含丰富的 API，支持更多的手机端的操作。
- ❑ 支持各种测试框架。

下面简单介绍一下 Appium 在模拟器上的使用方法：

1）先到 https://bitbucket.org/appium/appium.app/downloads/ 下载 appium.app 程序，记得下载 OS X 的版本。

2）安装该 App 到你的 Mac 机器上。

3）启动 App，界面显示如图 2-39 所示。

图 2-39　Appium 客户端界面

4）首先我们需要检测当前环境，点击  ，Appium 会自动帮你检测当前环境，是否有需要安装的依赖的工具。如果环境没问题会显示如图 2-40 所示界面（这里因为没有安装 Android 的环境，所以报错，可以忽略）。

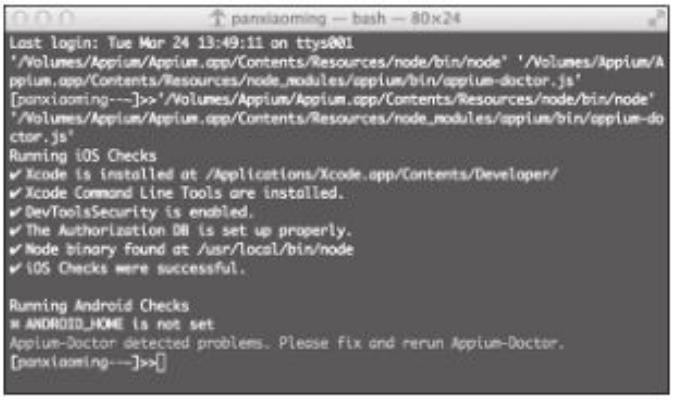


图 2-40　环境检测结果

5）因为我们需要测试 iOS 的 App，所以应勾选苹果的图标，点击该图标，进入 iOS 的设置界面，如图 2-41 所示。在这里设置我们要测试的项目工程编译完的 .app 文件的路径，以及模拟器和 iOS 版本。

图 2-41　iOS 设置界面

6）点击 进入常规设置，如图 2-42 所示，在这里设置 Appium Server 的地址和端口信息，并设置本地地址和默认端口，同时勾选回调的地址和端口。

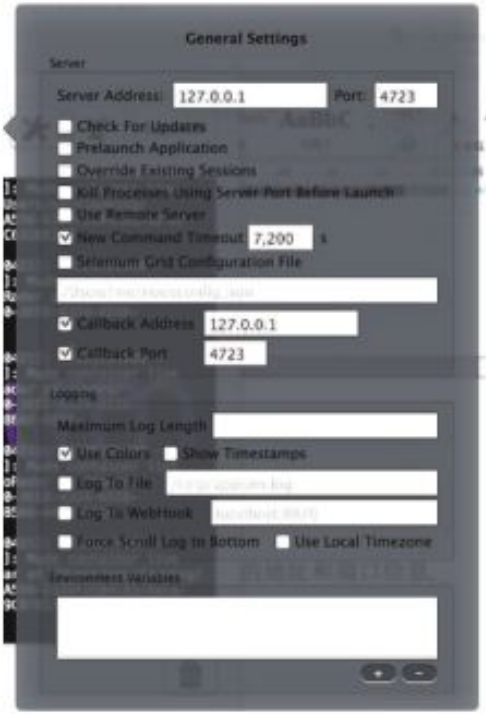


图 2-42　常规设置界面

7）点击 Lauch 就可以启动 Appium Server 了。点击  就可以去捕获 UI 控件了，我们可以看到模拟器被启动了，同时之前设置的 App 也被启动了。

8）在控件捕获界面，我们可以通过点击右边的截屏来获取控件的 xpath，如图 2-43 所示。

图 2-43　控件捕获界面

9）同时可以点击录制，自动生成操作代码，如图 2-44 所示。录制界面支持预设的手势操作。

图 2-44　录制自动生成代码

10）我们可以用自己熟悉的语言编写自己的测试用例，然后去执行这些用例，图 2-45 为在 Eclipse 里执行 Java 的测试用例。

本节大致介绍了 Appium 的基本功能和使用方法，由于篇幅有限，这里就不作详细介

绍了。有兴趣的读者可以去官网了解详情，网址为：http://appium.io。

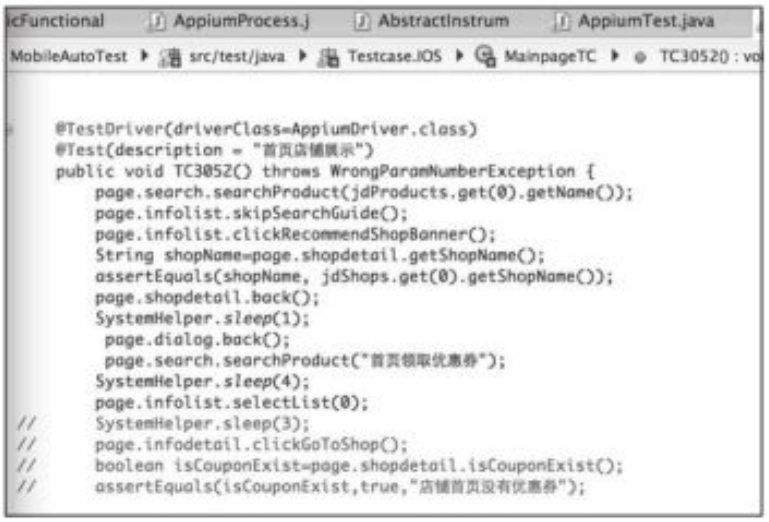


图 2-45　在 Eclipse 中管理执行测试用例

2.3　本章小结

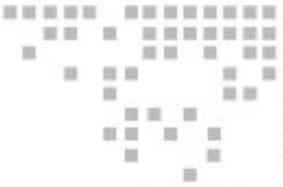
本章我们介绍一种基于开源工具的轻量级接口自动化的实践方法，并结合实例分别介绍了 Android 和 iOS 的 App UI 自动化测试的一些技术方案，可以作为无线测试团队开展自动化测试方面的一些参考。这两种方法看似独立，其实可以互为补充，甚至可以结合起来用。因为 UI 自动化测试中的断言通常是对 UI 层面的元素做出的，例如判断是否界面上某个文本控件的文本符合要求等。但是对于一些特殊情况，例如操作控件后 UI 并无可提供断言的变化，而是对后台的数据产生了变化，我们需要有对非 UI 进行断言的方法，这样对于测试检查的完整性很有必要。在这种情况下，可以借助接口自动化提供的功能，直接访问数据库的数据，或者通过访问接口，对接口的数据进行断言来判断 UI 操作的结果。

具体的实现方式可以使用测试工具串联 UI 自动化和非 UI 的断言。

举例来说，我们可以用 JMeter 来串联两种自动化方式。JMeter 提供了一个采样器叫“ OS Process Sampler”，可以用来执行命令行。基本上所有的 UI 自动化方法最终都可以用命令行来进行驱动。而 JMeter 本身又提供了多种采样器可以跟服务和数据库（通过 JDBC Request Sampler）交互。因此我们可以使用它来串联 UI 自动化和非 UI 的断言。我们可以

为测试团队准备通用的可复用的用来驱动 UI 自动化的“ OS Process Sampler”，测试人员只需准备自动化测试脚本，并跟平时一样，使用其他 JMeter 采样器和 JMeter 断言进行数据校验即可。

当然，也可以反过来在封装过的 UI 自动化的脚本中提供访问接口和后台 DB 的能力，来支持上面的需求。



第3章 Chapter 3

性能测试

如果你无法度量它，你就无法管理它。

——彼得·德鲁克

前面章节我们探讨了移动产品的功能测试，以及功能测试的自动化。通过这样的一些测试活动我们可以验证被测产品功能层面的正确性和可用性，但如果只是做得这样，可能还达不到一个高质量产品的要求，其中一个很重要的方面就是性能。比如我们在曾经用过的产品中遇到的一些情况：

- App 使用的时候会觉得非常卡顿不流畅。
- 查询一个信息或者执行一个操作，服务端需要好几秒才有响应结果。
- 当应用在使用高峰，比如电商的促销活动时，应用频繁报错。
- App 使用一段时间后内存占用过高，甚至出现闪退。
- 手机访问应用的界面打开非常慢。

以上这些都非常严重的影响用户的使用体验，也很有可能造成用户的流失，并造成很差的口碑，并影响业务的运营数据。

为了避免这样的风险，通常我们会在测试阶段进行性能测试，获得测试数据，这样便于对性能进行量化的分析，也便于改善后的对比。

性能测试的开展和被测系统的特点密切相关，针对移动互联网产品的构成，简单来说性能可以分为前端性能和后台接口性能，进一步，前端又分为 Web 页面和原生的 App 代码。对于 App，会分别针对 Android 和 iOS 来进行测试。

在这一章里面，我们将从三个方面来介绍性能测试。首先是 Web 前端的性能，为了介绍这部分的性能问题，我们会介绍 HTTP 协议相关的知识，以及常用的测试方法。第二部分是 App 端的性能，包括 Android 和 iOS 内存相关的问题，以及内嵌 Web 组件的性能分析。最后一部分会介绍关于后台服务的性能测试，包括了压力场景的建模、测试工具的介绍以及测试数据的收集和分析。因为移动产品的后台和传统 Web 产品的后台比较类似，所以这一部分的知识其实并不局限于移动产品，是比较通用的内容。

3.1 Web 前端性能测试

看到这一节，有些人可能会问，本书不是介绍移动互联网相关的测试吗，为什么要介绍 Web 性能测试的部分？

答案当然是因为移动互联网的产品也会直接涉及 Web 的部分，主要是两部分：

1）M 站，或者有些团队成为触屏版、Touch 版。

在 PC 浏览器和手机浏览器上输入同一个网站的同样的 URL，返回的内容完全不同，这个目前已经是非常普遍的做法，主要是考虑手机屏幕的大小和流量等情况，返回专门的 M 版本。比如我们在手机打开浏览器访问 www.sina.com.cn，然后会被自动跳转到 sina.cn，是针对手机浏览器的版本，如图 3-1 所示。大家可以试试自己常在 PC 浏览器上访问的一些网站，看是否有针对手机的适配版本。

图 3-1 针对手机浏览器自动适配的网页

如果不了解这个是怎么做到的，可以搜索下 User-Agent 这个 HTTP Header 字段，以及 HTTP 协议的 302 响应码对应的跳转机制。

2）很多 App 都是混合方式，既有原生的代码，也有内嵌的网页。

比如一些新闻类或者阅读类的 App，可以在登录 / 列表 / 订阅等功能采用原生的代码，但是在内容方面直接使用内嵌的 HTML 页面，这样灵活性很大，不用写死模板，为各种内容的展现提供了便利和灵活性。还有一类如电商的 App，因为会有各种各样的促销活动，这些活动内容和排版布局可能会有非常多的花样，而原生的模板通常有限，如果新增的话还需要开发、测试和发布，而 App 的发布和更新又是一个比较长和代价较大的动作，所以通常这一部分也是做成 HTML 页面的。

基于以上的原因，其实在移动互联网的开发和测试中，我们还是会需要大量接触到之前 Web 方面的内容，只是这时候页面在终端的载体从 PC 的浏览器变成了手机的浏览器或者 App 内嵌的浏览器组件，比如 WebView。

下面是一个具体的例子，这个是京东 App 里面的一个活动，点击后显示如图 3-2 所示。

图 3-2 App 内嵌 Web 运营活动页面示例

如果这个时候我们把手机 Wifi 的代理指向 PC 上，然后在 Fiddler 工具上看到打开上述活动时的 HTTP 请求，如图 3-3 所示，大家可以看到，有 HTML、CSS、JS、JPG 图片，和我们在 PC 网页抓包看到的非常类似。

图 3-3 App 上运营活动对应的 HTTP 请求

从这个角度来看，从 Web 互联网到移动互联网时代，很多 Web 前端的测试方法、测试工具都能继续发挥作用，性能测试方面也是一样。

3.1.1 HTTP 性能相关的技术要点

由于 Web 前端的元素获取是建立在 HTTP 协议之上的，在讨论 Web 前端性能测试之前，我们需要对 HTTP 协议有一些了解，包括基本的交互过程、协议构成。另外，为了更好地理解 Web 前端的一些性能要点，我们需要了解 HTTP 是如何高效地借助下层的 TCP 协议来运作的。最后，我们介绍 HTTP 协议中和性能直接相关的特性。了解这些知识点对于我们理解 Web 前端性能测试和优化是非常有必要的。

3.1.1.1 HTTP 协议介绍

接下来我们介绍 HTTP 协议的一些基本内容，这些对于理解后续内容非常有帮助。对此已经比较熟悉的读者可以略过这一部分。

HTTP 是 Hypertext Transfer Protocol（超文本传输协议）的简称，是整个互联网中广泛使用的协议，也是 Web 的基础。自 1990 年起，HTTP 就已经应用于 WWW 全球信息服务系统，已经有很长的一段历史。

HTTP 是一种请求 / 响应式的协议，基于 TCP 协议来进行数据传输，一个客户机与服务器建立连接后，发送一个请求给服务器，服务器接到请求后，给予相应的响应信息。

HTTP 的第一版本 HTTP/0.9 是一种简单的用于网络间原始数据传输的协议。HTTP/1.0 由 RFC 1945 定义，在原 HTTP/0.9 的基础上，进一步改进，允许消息以类 MIME 信息格式

存在，包括请求 / 响应范式中的已传输数据和修饰符等方面的信息。HTTP/1.1 由 RFC 2616 定义，要求更加严格以确保服务的可靠性，增强了在 HTTP/1.0 没有充分考虑到分层代理服务器、高速缓存、持久连接需求和虚拟主机等方面的能力。

安全增强版的 HTTP（即 S-HTTP 或 HTTPS）则是 HTTP 协议与安全套接口层 (SSL) 的结合，使 HTTP 的协议数据在传输过程中更加安全。本书不会讨论 HTTPS 相关的技术，相关内容请查阅对应的资料。

通过 HTTP 或者 HTTPS 协议请求的资源由统一资源标示符（Uniform Resource Identifiers，URI）来标识。HTTP URL 是一种特殊类型的 URI，包含了用于查找某个资源的足够信息，格式如下：

```
http://host[":"port][abs_path]
```

比如下面的 URL：

```
http://images.apple.com/v/macbook/b/overview/images/overview_design_large.jpg
```

- http 表示要通过 HTTP 协议来定位网络资源。
- host 表示合法的 Internet 主机域名或者 IP 地址。
- port 指定一个端口号，为空则使用默认端口 80。
- abs_path 指定请求资源的 URI；如果 URL 中没有给出 abs_path，那么当它作为请求 URI 时，必须以“/”的形式给出，通常这个工作由浏览器自动帮我们完成。

HTTP 是一个基于请求与响应模式的、无状态的、应用层的协议。每一次完整的交互过程都包含 HTTP 请求和 HTTP 响应。

HTTP 请求由三部分组成，分别是：请求行、消息报头、请求正文（可选）。

请求行以一个方法符号开头，以空格分开，后面跟着请求的 URI 和协议的版本，格式如下：

```
Method Request-URI HTTP-Version CRLF
```

其中：

- Method 表示请求方法。
- Request-URI 是一个统一资源标识符。
- HTTP-Version 表示请求的 HTTP 协议版本。
- CRLF 表示回车和换行（除了作为结尾的 CRLF 外，不允许出现单独的 CR 或 LF 字符）。

图 3-4 所示是一个 HTTP GET 请求的内容。包含了请求行和消息报头，并无请求正文。请求行：

```
GET / HTTP/1.1
```

GET 是方法；“/”是 Request-URI 请求的资源路径，是基于消息报头中 host 字段标识的主机的路径；HTTP/1.1 是协议和版本号。

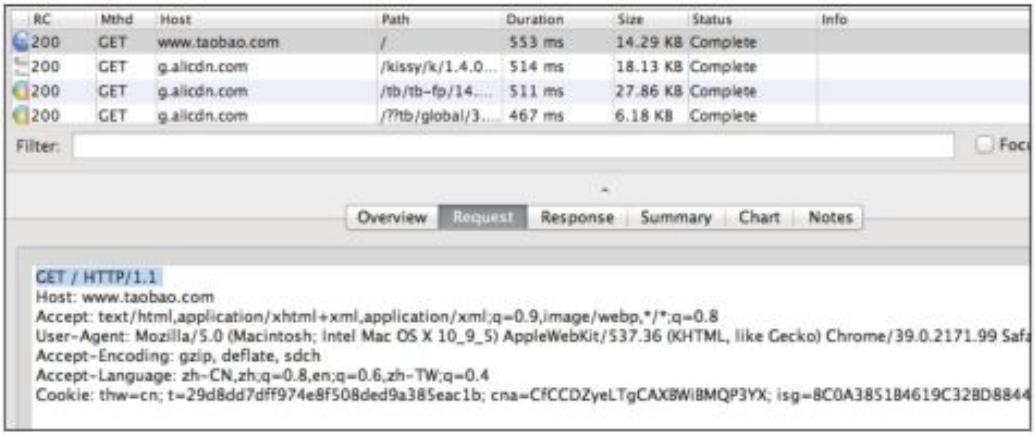


图 3-4 HTTP GET 请求内容示例

请求方法（所有方法全为大写）有多种，各个方法的解释如下：

- GET 请求获取 Request-URI 所标识的资源。
- POST 在 Request-URI 所标识的资源后附加新的数据。
- HEAD 请求获取由 Request-URI 所标识的资源的响应消息报头。
- PUT 请求服务器存储一个资源，并用 Request-URI 作为其标识。
- DELETE 请求服务器删除 Request-URI 所标识的资源。
- TRACE 请求服务器回送收到的请求信息，主要用于测试或诊断。
- CONNECT 保留将来使用。
- OPTIONS 请求查询服务器的性能，或者查询与资源相关的选项和需求。

其中 GET、POST、PUT、DELETE 比较常用。

图 3-5 所示是一个 HTTP POST 请求的内容示例，消息报头（header）和请求正文（body）之间用一个空行来分隔。其中乱码为工具无法解析加密内容所致。

HTTP 响应和请求类似，也是由三个部分组成，分别是：状态行、消息报头、响应正文。

状态行格式如下：

```
HTTP-Version Status-Code Reason-Phrase CRLF
```

例如：HTTP/1.1 200 OK（CRLF）

图 3-5 HTTP POST 请求内容示例

其中，HTTP-Version 表示服务器 HTTP 协议的版本；Status-Code 表示服务器发回的响应状态代码；Reason-Phrase 表示状态代码的文本描述；CRLF 表示回车和换行。

状态代码有三位数字组成，第一个数字定义了响应的类别，且有五种可能取值：

- 1xx：指示信息——表示请求已接收，继续处理。
- 2xx：成功——表示请求已被成功接收、理解、接受。
- 3xx：重定向——要完成请求必须进行更进一步的操作。
- 4xx：客户端错误——请求有语法错误或请求无法实现。
- 5xx：服务器端错误——服务器未能实现合法的请求。

常见状态代码、状态描述和说明如下：

- 200 OK // 客户端请求成功。
- 400 Bad Request // 客户端请求有语法错误，不能被服务器所理解。
- 401 Unauthorized // 请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用 。
- 403 Forbidden // 服务器收到请求，但是拒绝提供服务。
- 404 Not Found // 请求资源不存在，eg：输入了错误的 URL。
- 500 Internal Server Error // 服务器发生不可预期的错误。
- 503 Server Unavailable // 服务器当前不能处理客户端的请求，一段时间后可能恢复正常。

图 3-6 所示是一个 HTTP 响应的内容。

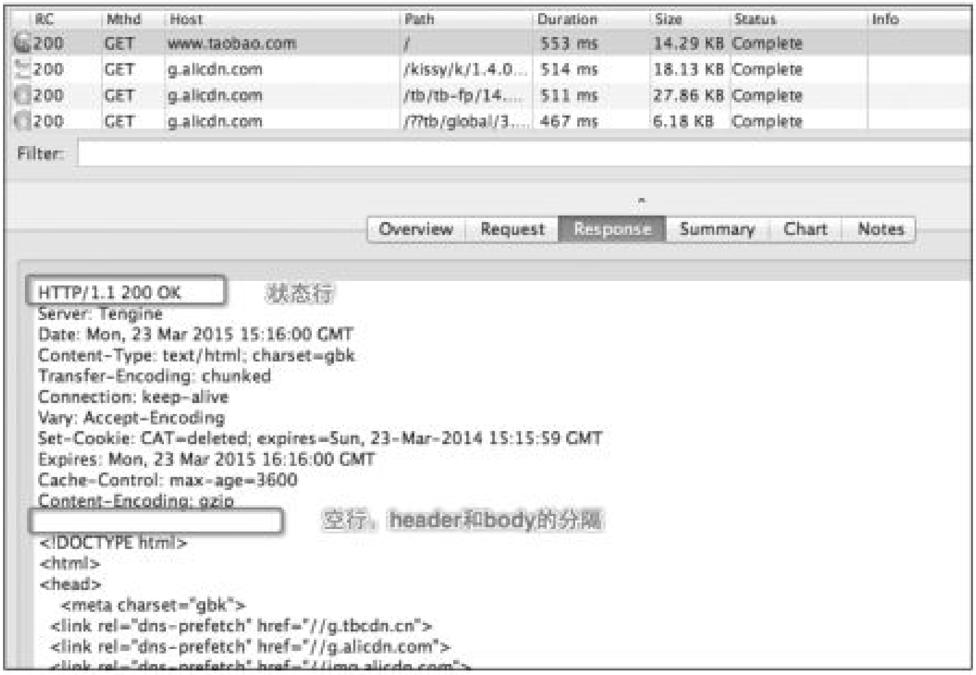


图 3-6 HTTP 响应内容示例

至此我们对 HTTP 协议有了一些基本的了解，更详细的内容可以参考对应的 RFC 文档。

3.1.1.2 在 TCP 协议层面的交互和并发

从计算机网络协议的分层来看，HTTP 协议是基于 TCP 协议来运作的，如图 3-7 所示。

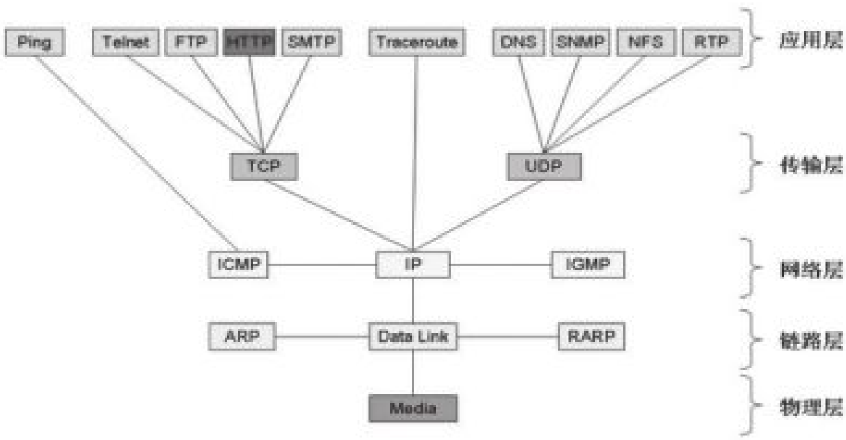


图 3-7 常见计算机网络协议的层次关系（图片来自网络）

接下来我们通过网络抓包工具 Wireshark 从 TCP 协议层面了解 HTTP 的交互过程。图 3-8 所示是一次完整的 HTTP 请求和响应在 TCP 层面看到的过程。

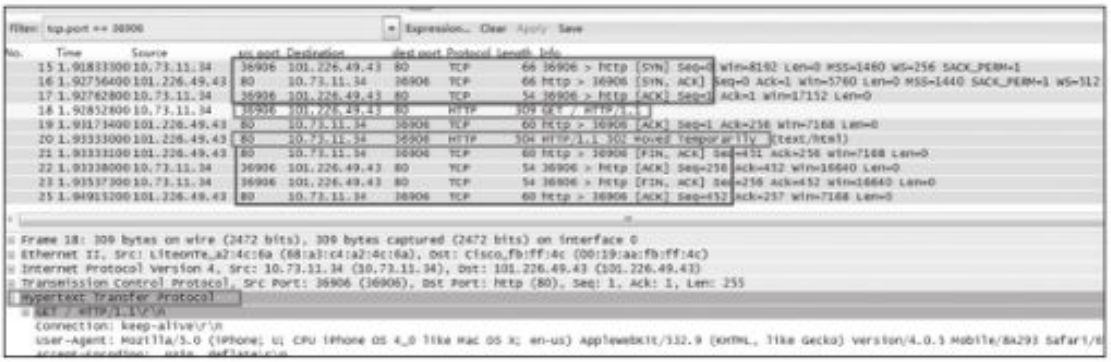


图 3-8 一次 HTTP 请求和响应在 TCP 层面的过程

从图中可以看到其中包含了 TCP 连接建立的过程（俗称三次握手）、HTTP GET 请求的发出（No.18）、HTTP 响应的返回（No.20），以及 TCP 连接断开的过程（俗称四次挥手）。

如果我们选取上面的 18 号包，可以进一步查看协议的层次，如图 3-9 所示，可以看到 HTTP - TCP - IP- Ethernet 的层次。

图 3-9 一个 HTTP 报文的协议层次图

在讨论了 HTTP 在 TCP 层面的交互过程之后，接下来我们看 HTTP 协议如果通过 TCP 层面的并发来达到高效获取资源的效果。对于这个并发的理解非常重要，而且不只是针对 M 版和内嵌网页部分，原生 App 应用如果使用 HTTP 协议获取数据，接口请求的封装也会涉及并发相关的参数。

针对这个问题，我们可以通过一个实验结果来分析。我们用浏览器访问新浪的首页（www.sina.com.cn），并用 Wireshark 录下所有的请求。如果按照上面单次 HTTP 请求和响应的 TCP 协议过程看到的结果，那么如果页面有 200 个元素，就应该新建和销毁 200 个 TCP 连接。而实际情况并不是如此，我们抓取了通过浏览器访问新浪首页的过程，进而分析了部分 HTTP 请求和响应。

如图 3-10 所示，我们发现 3828 作为 TCP 源端口出现了 6 次。

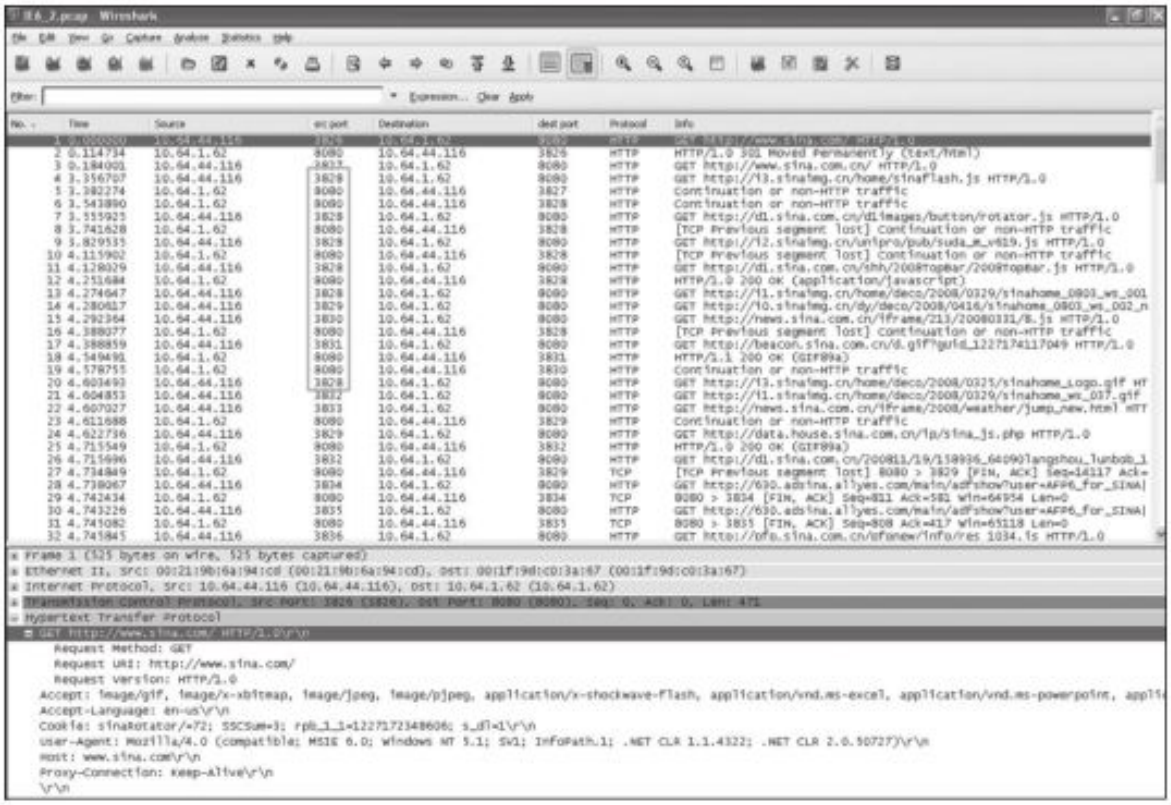


图 3-10　HTTP 多次请求中 TCP 连接复用的情况

基于对 TCP 协议的理解，对一个确定的远端 Server 和 port，本地 TCP 源端口不可能在短时间内复用。那么这里看到的 3828 作为源端口出现 6 次就表明，在这个 TCP 连接（10.64.44.116:3828–10.64.1.62:8080）上，连续完成了 6 次 HTTP 请求和响应，获取了 6 个 HTTP 元素。也就是说 HTTP 协议多次复用了这个 TCP 连接来串行发起多次请求。这也是

HTTP 1.1 版本相对于 1.0 的一个重要改进。TCP 连接的建立和消耗都是一个耗费资源和时间的过程，所以连接的复用对于性能的提升是很重要的。

下面这篇文章比较详细地对比了 HTTP 1.0 和 1.1 的区别，Key Differences between HTTP/1.0 and HTTP/1.1（http://www8.org/w8-papers/5c-protocols/key/key.html）。

在复用 TCP 连接获取多个资源的基础上，我们发现如果需要从一台主机上获取大量资源，浏览器会建立多个 TCP 连接来并行发起请求。

那么浏览器如何来决定建立多少并行的连接呢？这取决于三个方面：

- 需要从同一个主机获取的资源数量，这个决定了最大的并行数。
- 浏览器关于这个并行的设置。

在实际的浏览器中，我们找到了关于的设置。如图 3-11 所示是 Firefox 的并发数设置。

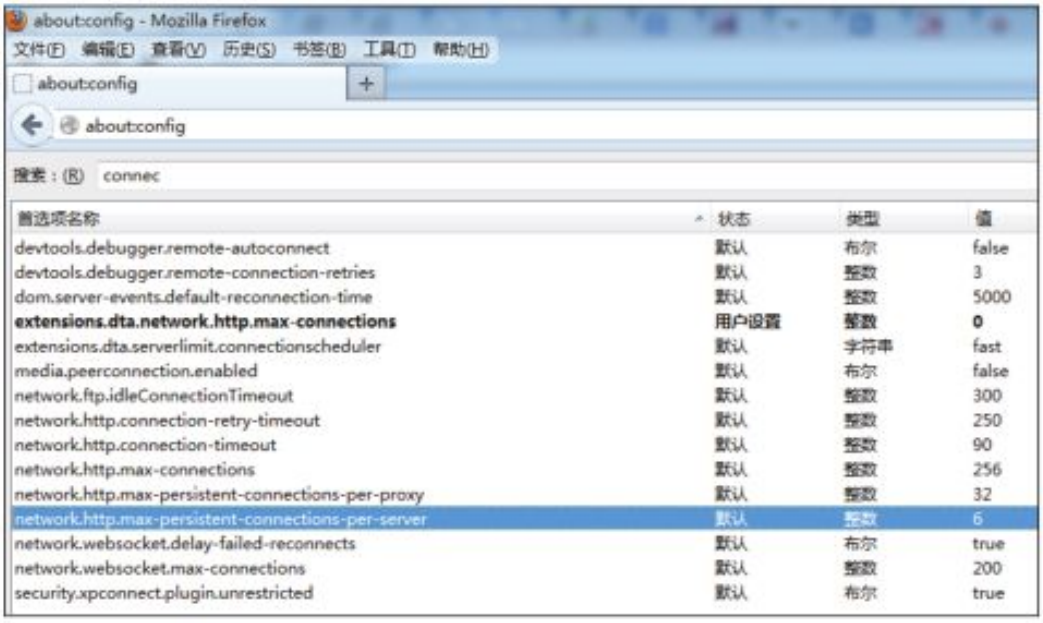


图 3-11 Firefox 浏览器中关于同一个 Server 并发连接数的设置

需要说明的是，这个参数限定了最大的并发连接数，并不意味着必须建立这么多连接。

在针对 HTTP 协议的测试中，很多工具也模拟浏览器支持用户定义这个参数值。图 3-12 是 Avalanche 测试工具中关于到单个 Server 最大连接数的设置。

图 3-13 是 JMeter 在 HTTP 请求中提供的设置。

针对这个参数的效果，我们做了一个对比实验，用 JMeter 打开一个 www.sina.com.cn 或者 www.qq.com 这种页面内容元素比较多的网站。注意要勾选上面选项中的“从 HTML 文件获取所有内含的资源”，除了基本的 HTML 文件之外，也和浏览器一样继续获取 CSS、

JS、JPG 等资源。

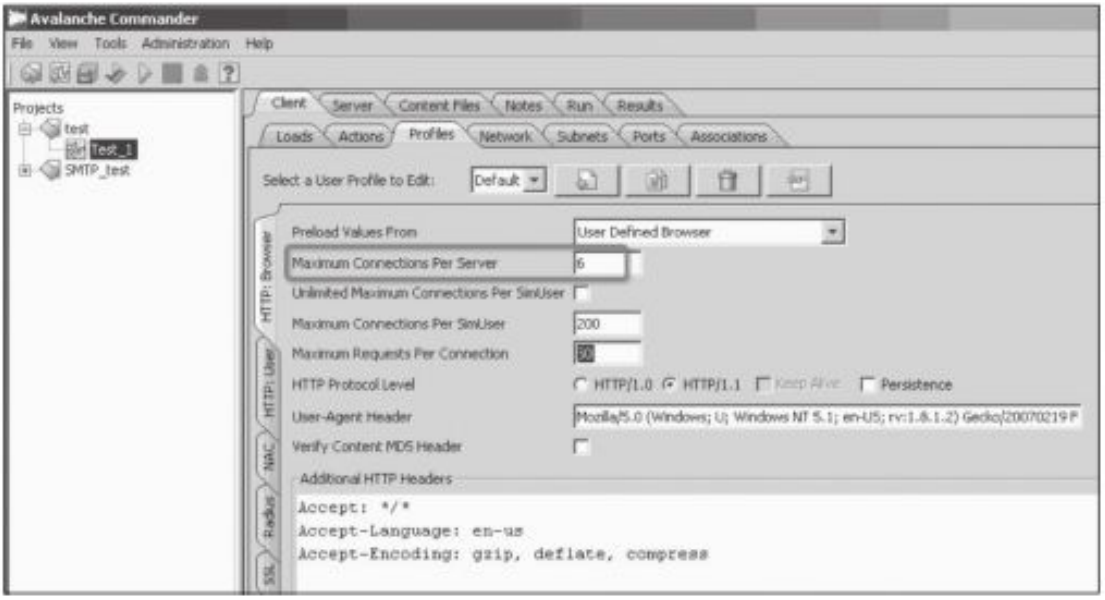


图 3-12 Avalanche 中到单个 Server 最大连接数的设置

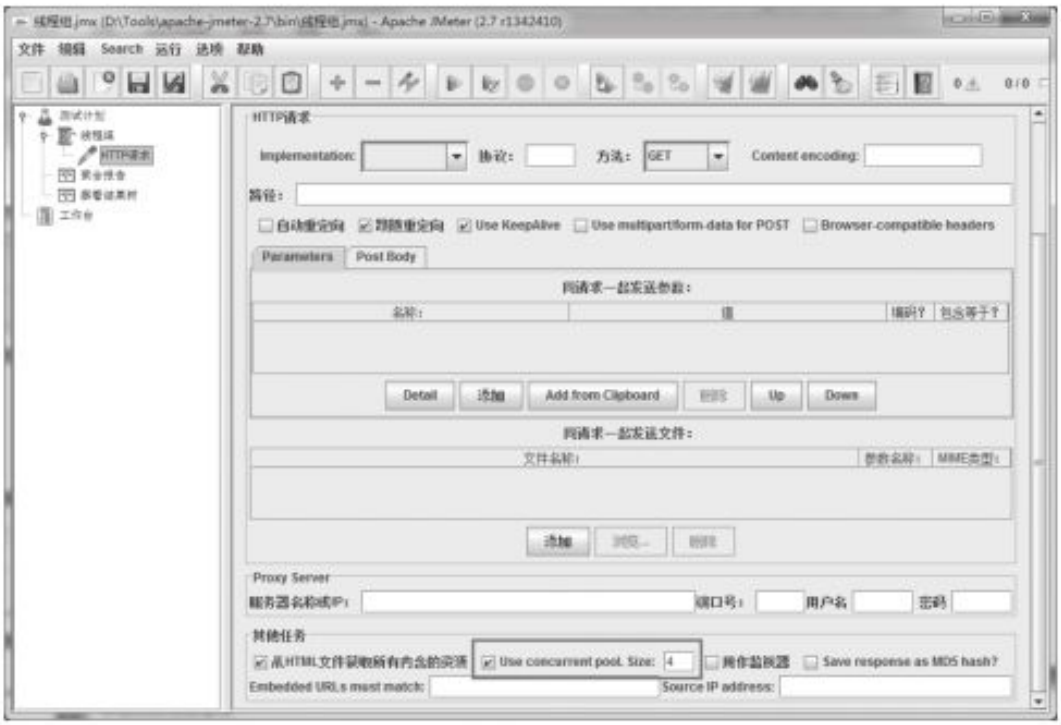


图 3-13 JMeter 中单个 Server 最大连接数的设置

下面的“HTTP 请求”是没有勾选“Use concurrent pool”选型的，而“HTTP 请求 2”勾选了，并将 pool size 设为 4。

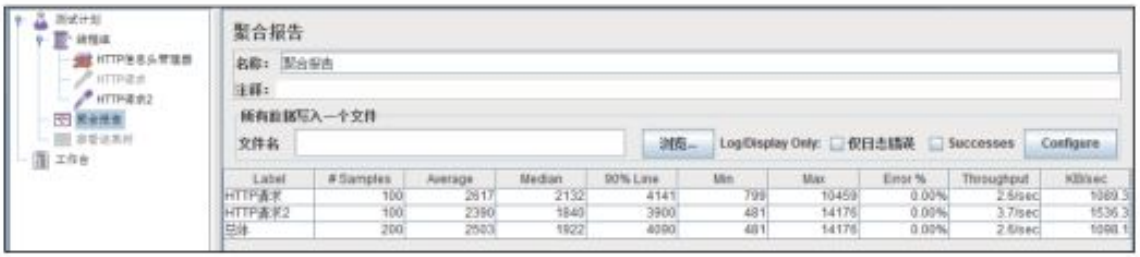


图 3-14 是否开启 TCP 连接并发的测试结果对比

从图 3-14 的结果我们可以看到，差别非常明显，设置了 pool size = 4 之后，每秒完成的请求数提高了 40% 多。通过前面的说明，我们就比较容易理解这个设置的意义，也帮助我们在性能测试中更准确地模拟用户行为。因为这个并发是针对同一个域名的，如果域名比较分散意义就不大，这也就是为什么 Web 性能优化的建议里面也包括不要有太多域名。当然，这个还受到其他参数的限制，比如整个浏览器能建立多少个链接。

为了更深入地理解前端性能，HTTP 协议中有很多细节的内容值得去了解。下面讨论的两个方面都和性能相关。

通常我们理解浏览器通过 HTTP 协议获取页面元素的过程是：浏览器会先取回这个页面的主 html 文件，然后解析内容，进而发起请求去取其他页面展示所需要的元素，比如图片、css、js 等。通常情况是如此，但并不一定每次都是这样。下面我们看一个具体的例子，图 3-15 所示是通过 Chrome 浏览器访问 www.qq.com 首页，过程中通过 Wireshark 抓取的网络包。

从图 3-15 可以看出，在 No.36 发起了获取首页（根目录）HTML 内容的 HTTP GET 请求。到 No.376 的时候，这个请求才返回。而在这个返回之前，浏览器发起了大量的页面其他元素的请求，包括 js 和图片等，介于 No.15 和 No.376 之间。

接下来我们对首页的响应进行进一步的分析。

如图 3-16 所示，我们发现由于首页内容较大，超过 TCP 单个 Segment 1460 字节的限制，所以整个首页响应约 58KB 的内容，分成 42 个 TCP segment 返回，然后在 HTTP 层面进行合并。

获取了 14 个 TCP segment（截止 Frame #75）后，开始发起内嵌资源的请求（Frame #80）。到 Frame #376，整个首页 html 返回完成。在这个过程之间，浏览器已经获取了大量的其他页面资源。

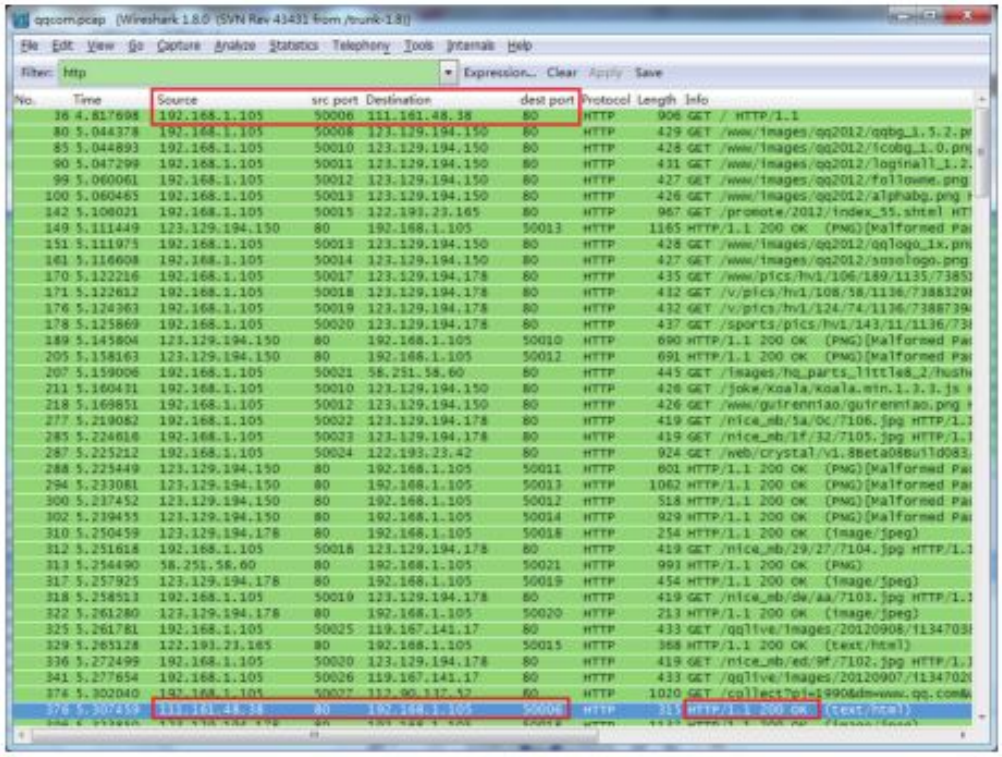


图 3-15 通过浏览器获取网站元素的 Wireshark 抓包

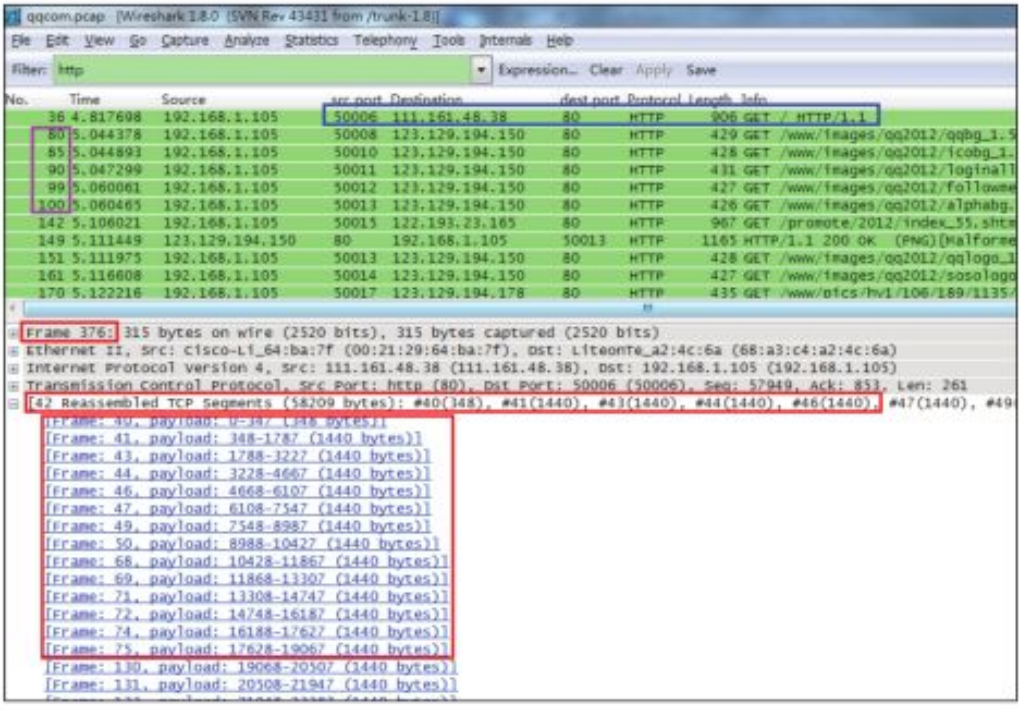


图 3-16 首页响应内容对应的 TCP Segments

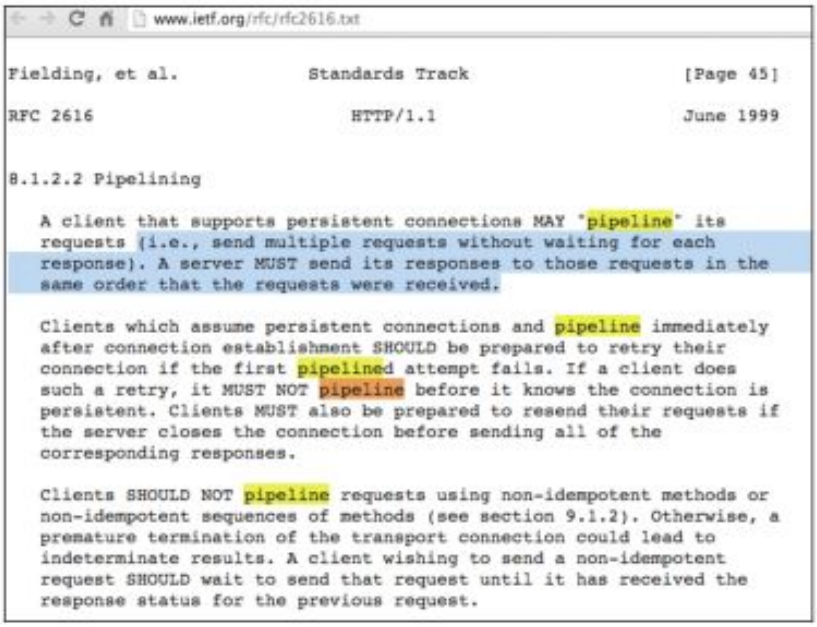
所以实际上，浏览器在获取了部分的 HTML 内容之后，就开始了解析，并基于解析的内容发起了针对其他页面元素的请求。这个过程比我们前面理解的要更加高效，并行度更高。

在前面的例子中，我们提到 HTTP 可以复用一个 TCP 连接发起多个请求。在上面的例子中，多个请求和响应是串行来完成的，从时间上来看，过程如下：[建立 TCP 连接] → [HTTP 请求 #1] → [HTTP 响应 #1] → [HTTP 请求 #2] → [HTTP 响应 #2] … → [HTTP 请求 #n] → [HTTP 响应 #n] → [断开 TCP 连接]。

以上过程相比每次发起 HTTP 请求都重新建立连接提高了效率，也减少了 TCP 连接数。但是进一步来看，它仍然存在不高效的地方，如果 HTTP 响应 #1 需要较长时间返回，其原因可能是服务器处理不够快或者网络延时较大，那么后面的请求只能逐个排队。

这个模型就好比一个按订单生产某种产品的工厂，每次只有交付了前一个订单才能接受下一个订单。那是否可以批量地把订单接收进来，处理完了之后逐个返回呢？因为这个看起来是更高效的方式。

实际上从前面提到的 HTTP 1.1 开始，就考虑了支持这种方式，称之为 pipeline。如图 3-17 所示，在 HTTP 1.1 对应的 RFC 2616 中，有关于 pipeline 的描述。


www.ietf.org/rfc/rfc2616.txt

Fielding, et al. Standards Track [Page 45]

RFC 2616 HTTP/1.1 June 1999

8.1.2.2 Pipelining

A client that supports persistent connections MAY "pipeline" its requests (i.e., send multiple requests without waiting for each response). A server MUST send its responses to those requests in the same order that the requests were received.

Clients which assume persistent connections and pipeline immediately after connection establishment SHOULD be prepared to retry their connection if the first pipelined attempt fails. If a client does such a retry, it MUST NOT pipeline before it knows the connection is persistent. Clients MUST also be prepared to resend their requests if the server closes the connection before sending all of the corresponding responses.

Clients SHOULD NOT pipeline requests using non-idempotent methods or non-idempotent sequences of methods (see section 9.1.2). Otherwise, a premature termination of the transport connection could lead to indeterminate results. A client wishing to send a non-idempotent request SHOULD wait to send that request until it has received the response status for the previous request.

图 3-17 HTTP 1.1 的 RFC 中关于 pipeline 的描述

但不幸的是，实际实现过程中，遇到了非常多的问题。以至于 Chrome 浏览器移除了相关的开关。在 Chromium 开发者站点有相关的说明：

“The option to enable pipelining has been removed from Chrome, as there are known crashing bugs and known front-of-queue blocking issues. There are also a large number of servers and middleboxes that behave badly and inconsistently when pipelining is enabled. Until these are resolved, it's recommended nobody uses pipelining. Doing so currently requires a custom build of Chromium.”

意思是 Chrome 不再支持 pipeline，因为很多已知的缺陷，以及很多 Web Server 对这个支持也有问题。

如图 3-18 所示，在 Firefox 中也有对应的选型，但是实际上并未开启。

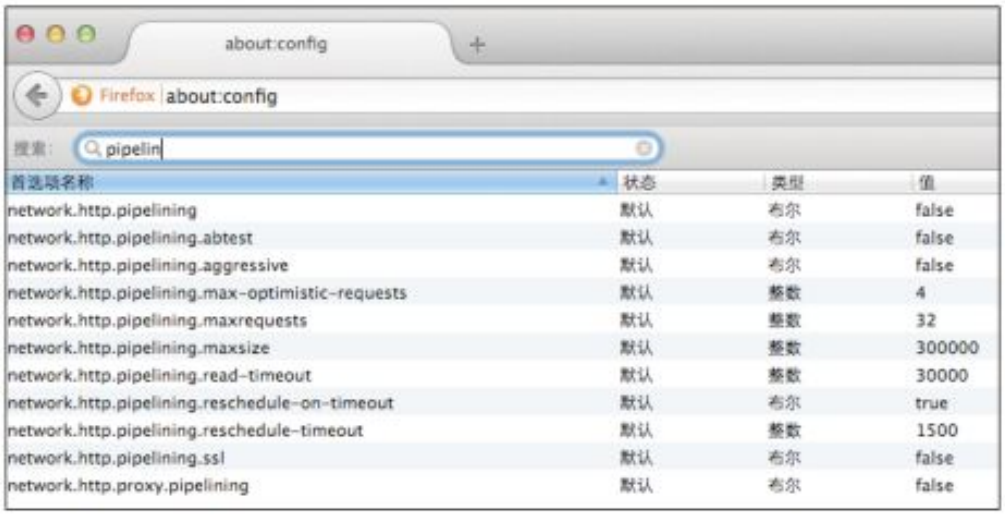


图 3-18　Firefox 中关于 pipeline 的设置

尽管如此，在同一个 TCP 连接上并行发送请求和响应仍然是一个明显的提升效率的方向。在 HTTP 2.0 的 IETF RFC 草案（https://datatracker.ietf.org/doc/draft-ietf-httpbis-http2）中这个被作为重要的改进列出。

相比 HTTP 1.1，2.0 中对于这种方式的实现给予了更明确的说明。

“In particular, HTTP/1.0 allowed only one request to be outstanding at a time on a given TCP connection. HTTP/1.1 added request pipelining, but this only partially addressed request concurrency and still suffers from head-of-line blocking. Therefore, HTTP/1.0 and HTTP/1.1 clients that need to make many requests use multiple connections to a server in order to achieve concurrency and thereby reduce latency.”

“HTTP/2 addresses these issues by defining an optimized mapping of HTTP's semantics to an underlying connection. Specifically, it allows interleaving of request and response

messages on the same connection and uses an efficient coding for HTTP header fields."

借助 frame 的概念，以及对于 frame 状态和处理方式的明确定义，HTTP 2.0 有望实现 TCP 连接的更高效利用，从而在既定的网络状况下进一步提升 HTTP 的性能。

3.1.1.3 性能相关的特性

HTTP 协议中还有一些直接和性能相关的特性值得关注。这里重点介绍两个方面，一个是 HTTP 传输过程中的数据压缩，另一个是客户端的缓存。

1. HTTP 传输过程中的数据压缩

首先我们通过一个具体的例子来看看。为了便于调整参数，我们使用 JMeter 来替代浏览器发起请求。

如图 3-19 所示，HTTP 头信息中有相关的字段 Accept-Encoding 来标识是否使用 gzip（一种开源的压缩格式）。只有客户端（比如浏览器）在 HTTP 请求头里面指明可以支持 gzip 压缩的时候，服务端才返回压缩的内容，避免客户端无法解析的情况。默认情况下，对于 HTML 文本这样比较易于压缩的内容，大部分网站都是开启的。

图 3-19 HTTP 头信息中关于压缩的设置

为了对比压缩的效果，如图 3-19 所示，我们制作了两个 HTTP 采样器，第一个开启 gzip 压缩，第二个不声明使用。

图 3-20 所示是开启压缩后，服务器返回的百度首页 HTML 内容的大小，为 12873 字节。

图 3-21 所示是禁用压缩后，服务器返回的百度首页 HTML 内容的大小，为 39514 字节。

对比可以发现，压缩后的内容大小为压缩前内容大小的三分之一左右，效果还是非常的明显。由于传输的内容大为减少，TCP Segments 的数量也减少很多，客户端可以更快地获取到页面内容，进而也会提升用户感知到的页面打开时间。

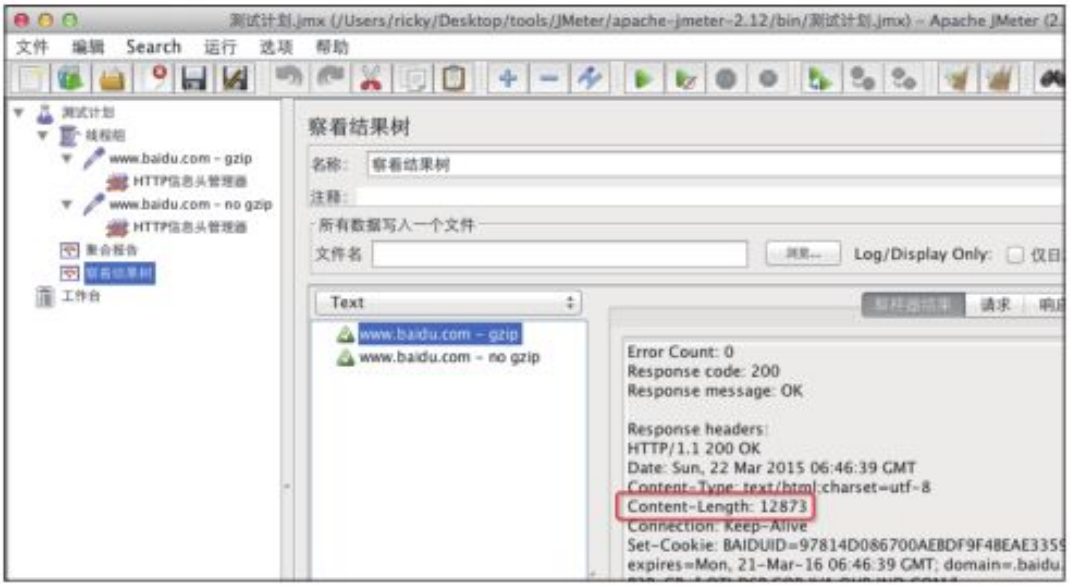


图 3-20 开启 gzip 压缩的后的百度首页文本大小

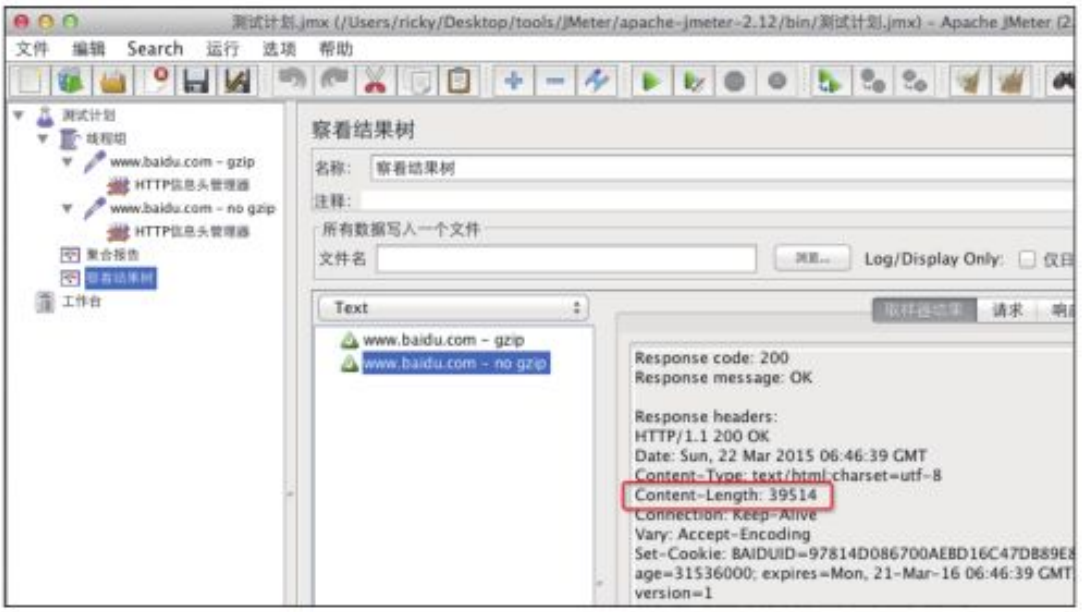


图 3-21 禁用 gzip 压缩的后的百度首页文本大小

需要说明的是，HTTP 传输中的压缩是一个双向的设置，一方面是上面提到的客户端要声明可以接收压缩后的内容，另一方面服务器也要开启压缩相关的设置。图 3-22 所示是 nginx 中关于数据压缩的部分设置。

```
nginx中关于压缩的设置
gzip              on;
gzip_comp_level   2;                          # 压缩比例，比例越大，
gzip_types        text/css text/javascript;   # 哪些文件可以被压缩
gzip_disable      "MSIE [1-6]\.";             # IE6对脚本压缩支持不好
```

图 3-22 nginx 中关于数据压缩的部分设置

2. 缓存的管理

用户在访问一个网页的时候，有一定概率可能会在一段时间之后再访问。而在一段时间内，网站上很多内容，特别是一些静态内容，如图片、JS 和 CSS 等，可能和上次访问的时候内容没有任何变化。如果后面每次都要重复从服务端拉取这些内容，就会浪费带宽，对于客户端的响应时间也有影响。所以对于没有变化的内容，在客户端缓存是一个很好的办法，HTTP 协议在设计的时候已经做了相应的考虑。下面我们通过一个实际的例子来看看。

首先，我们通过浏览器来访问 qq.com 首页，如果不是第一次访问，请先清除浏览器的缓存。如图 3-23 所示，在无缓存的情况下访问的时候，所有的元素都需要从服务端拉取。

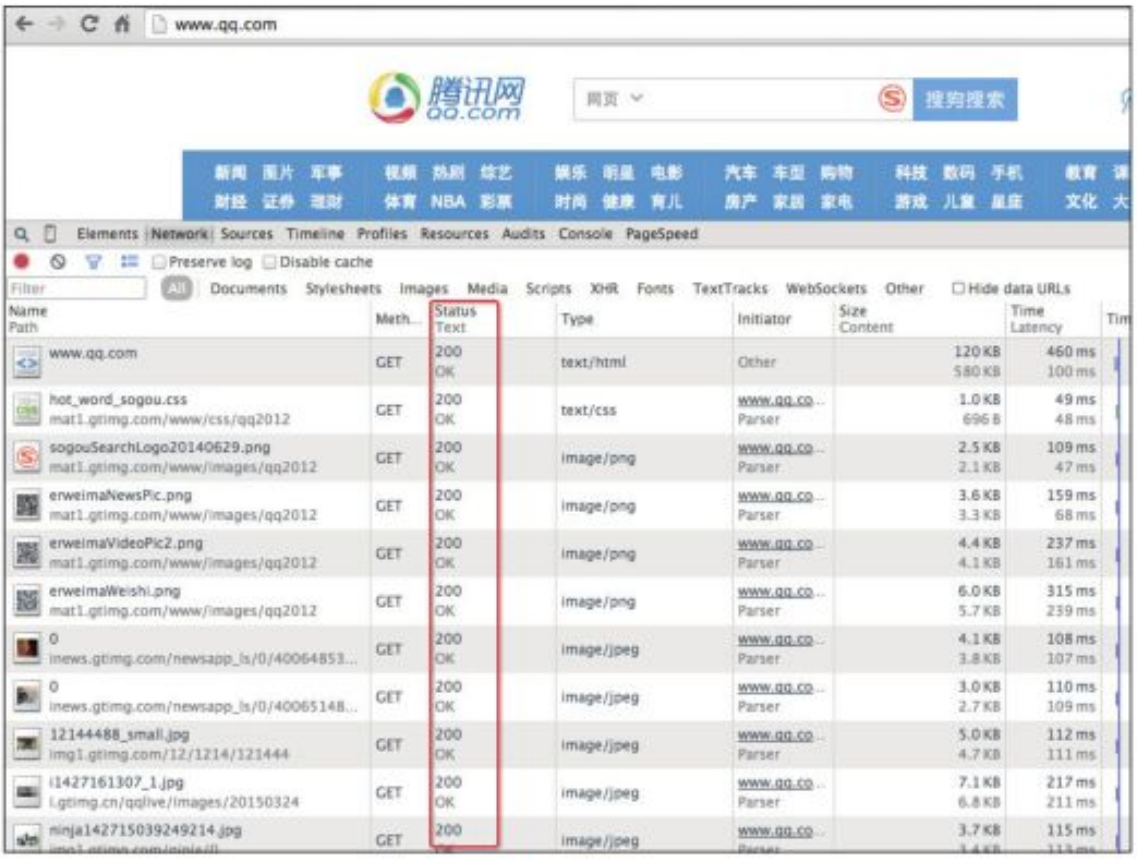


图 3-23 无缓存情况下访问 qq.com 首页

我们查看其中一个文件，ping.js，如图 3-24 所示，返回码是 HTTP 200。从响应头中可以看到几个缓存相关的字段。

- Cache-Control 设定了缓存的有效时间 max-age=600，表示 600 秒，相当于 10 分钟。
- Last-Modified 表示这个文件的内容最后一次在服务器上被修改的时间。Expires 表示这个文件可用于本地缓存的过期时间，默认使用的是 GMT 0，如果换算到中国时间（GMT +8），是 2015 年 3 月 24 日 13:19:45，也等同于响应头中的 Date 字段标识的时间 13:09:45，加上 Cache-Control 中的 10 分钟有效期。

这一次无缓存情况下服务端返回的这些头信息，为下一次客户端需要获取这个 URL 时是否可以用缓存打下了基础。

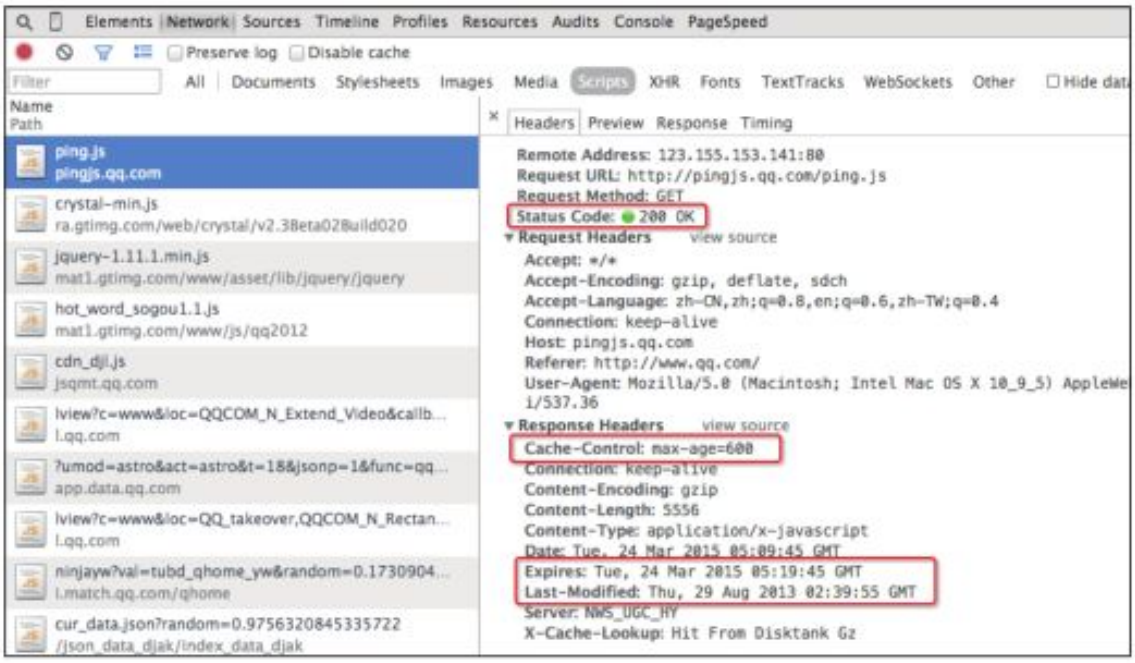


图 3-24　无缓存情况下单个请求和响应的消息头

在不主动清理浏览器缓存的情况下，我们第二次访问 qq.com 首页。如图 3-25 所示，我们发现除了 HTTP 200 之外，Size/Content 字段中，有大量的 URL 请求被标识为 from cache。这是 Chrome 浏览器用来表示本次访问时这些 URL 内容都是直接从本地缓存文件读取，而不是通过 HTTP 网络请求来获取的，缓存已经开始发挥作用了。

这些缓存都是在浏览器本机上的文件，不过做了整合和索引，如图 3-26 所示，是 Mac OS 下 Chrome 浏览器的缓存目录。

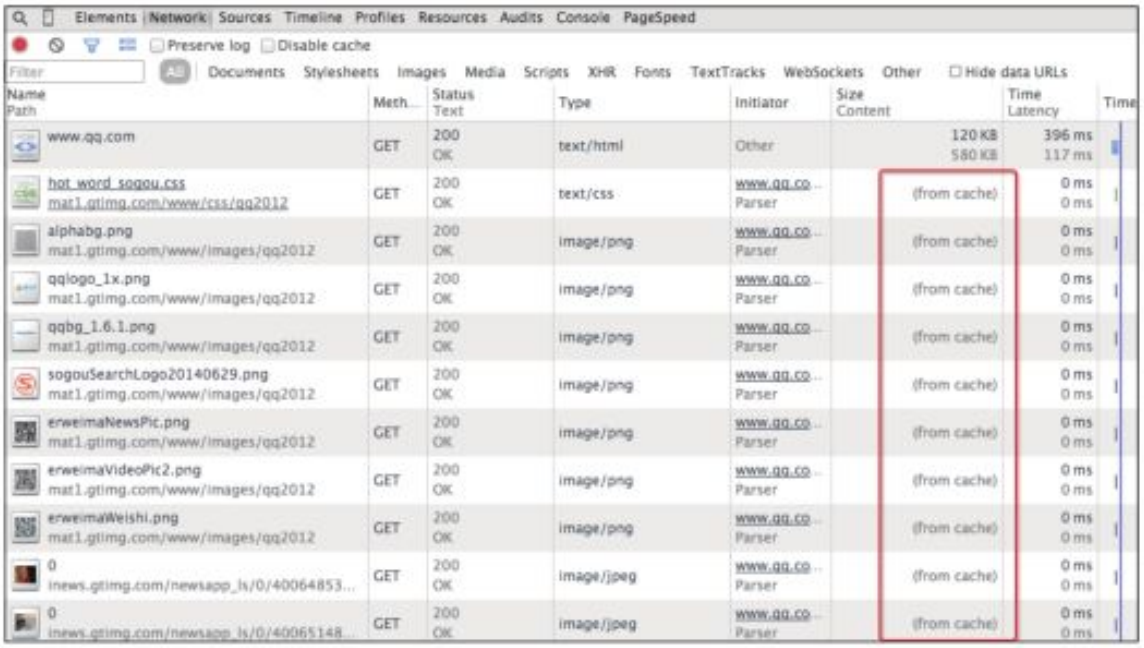


图 3-25 有缓存情况下访问 qq.com 首页

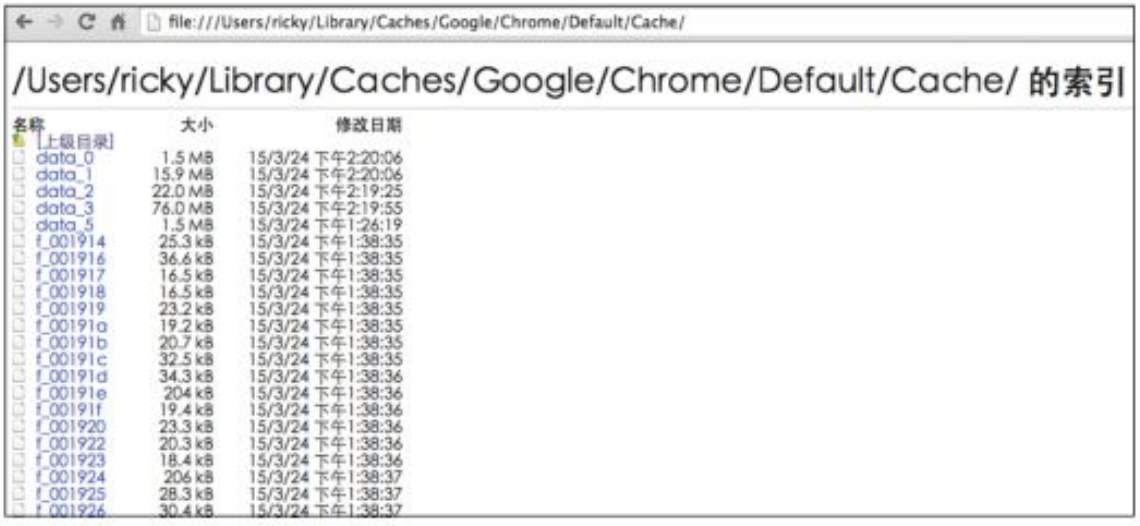


图 3-26 Mac 下 Chrome 浏览器的缓存目录

在有缓存访问的情况下，我们再来看看 ping.js 的请求情况，如图 3-27 所示，这一次，我们发现 Status Code 除了 200 之外，也是 from cache，表明这次并没有发送网络请求，而是直接读取的本地缓存文件。

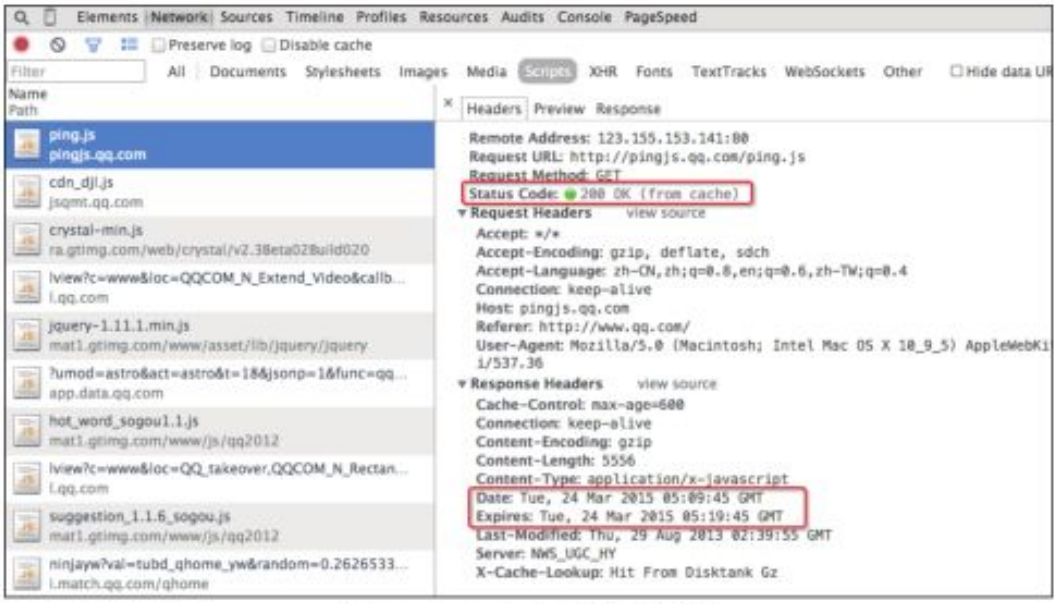


图 3-27　ping.js 的缓存使用情况

对比图 3-27 和图 3-24 会发现前者的详情里面没有 Timing 这个 tab，是因为根本没有发送网络情况，所以也没有必要记录网络相关的时间。

通过上面的分析，我们了解到上一次的缓存有效期是 10 分钟，到 13:19:45 过期。我们接下来等到这个时间之后再访问看看，是否缓存还会继续有效。图 3-28 所示是在 13:24:55 针对 ping.js 文件的试验结果。

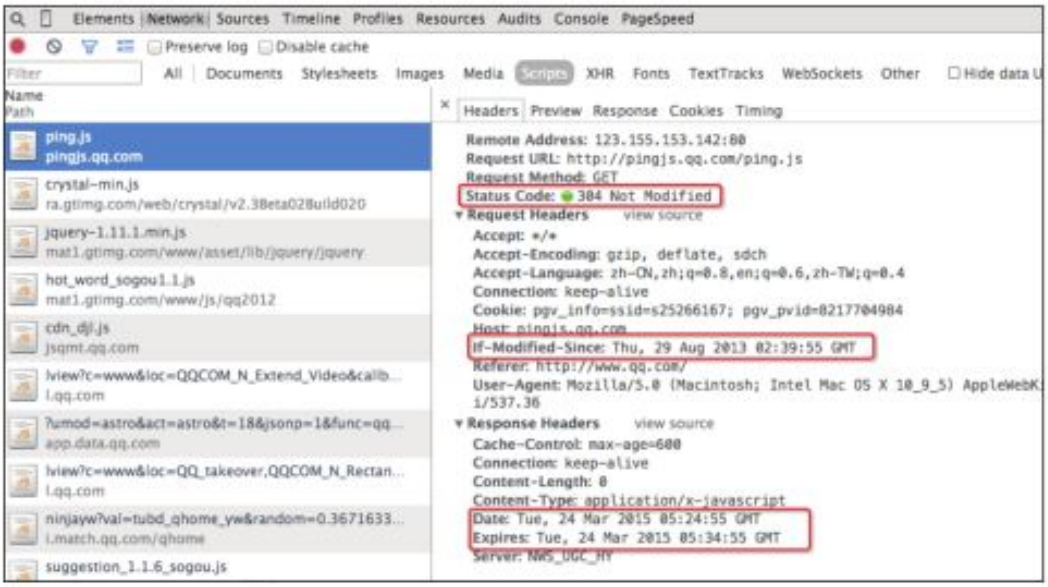


图 3-28　URL 缓存时间过期后继续再次访问的情况

从图中可以看到，这次没有直接取自本地缓存，表示缓存确实过期了，浏览器没有直接使用。但是另一方面，响应码也不是 HTTP 200，而是 304 Not Modified。

针对图 3-28 所示的 HTTP 304 返回，如果查看 Timing，如图 3-29 所示，也可以看到对应的各个阶段的耗时，表明还是发起了 HTTP 请求。

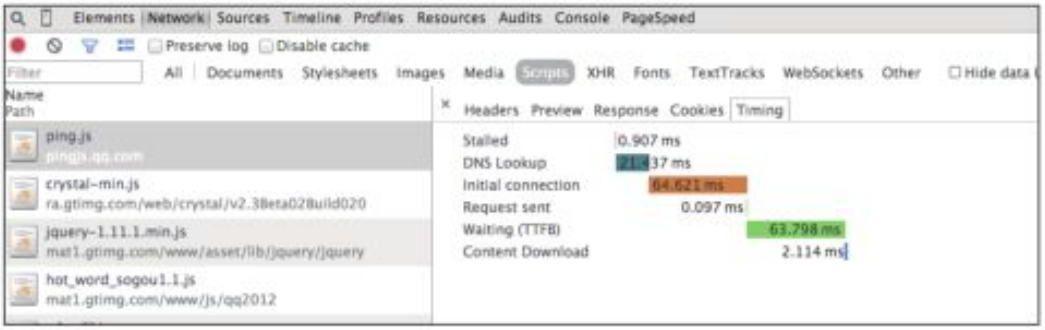


图 3-29 ping.js 在 HTTP304 情况下的 Timing 信息

这个响应码的意思是服务端接收到浏览器的请求，判断后认为服务端最新的该文件版本和客户端已有的一致，于是告诉客户端不需要再传输完整的内容了，可以直接用本地的版本。

这样做的好处是虽然客户端还是要发起请求，但是服务端只需通过 304 简单应答，而不必重新传输文件的内容，同样起到了缓存的目的。

那么这个是如何做到的？

从图 3-28 中，我们发现 HTTP 请求头中有一个字段：

```
If-Modified-Since:Thu, 29 Aug 2013 02:39:55 GMT
```

对比图 3-24，可以看到这个时间戳正好是第一次访问 ping.js 时，响应头中的 Last-Modified 字段的内容。

就是说客户端在第一次收到 ping.js 的响应时，将对应的信息存储起来。等到下次要请求这个同样的 URL 时，客户端发现缓存因为过期不能直接使用了（但是不会立即删除缓存文件），而会把这个时间戳回传给服务端，告诉服务端本地有这个版本的该文件。这样服务端收到这个信息后可以对比给出合适的响应。

上面我们通过一个实际的例子了解了 HTTP 缓存的效果，接下来我们了解几个重要的参数，除了上面提到的 Expires、Last-Modified 和 If-Modified Since。

Cache-Control 上面提到了，但实际上它有很多的参数可以选择，对于缓存的控制非常重要，参数包括：

- public：响应会被缓存，并且可以在多用户间共享。

- private：响应只能够作为私有的缓存，比如在一个浏览器中，不能在用户间共享，所以设置该参数后就不能被反向代理缓存了。
- no-cache：响应不会被缓存，而是实时向服务器端请求资源，这使得 HTTP 认证能够禁止缓存以保证安全性。实际中这个容易让人产生误解，字面理解是响应不被缓存，而实际上 no-cache 情况下也是会被缓存的，只是每次客户端都要向服务器评估缓存响应的有效性。
- no-store：在任何条件下，响应都不会被缓存，并且不会写入到客户端的磁盘里，这也是基于安全考虑的某些敏感的响应才会使用这个。
- max-age=[单位：秒]：设置缓存最大的有效时间，从服务端返回的时间开始计算。
- s-maxage=[单位：秒]：类似于 max-age，但是它只用于共享缓存，比如代理。

另外一个在缓存中会遇到的做法是通过 Etag 来决定缓存是否有效。引入 Etag 的目的是为了解决 Last-Modified 等机制无法解决的一些问题，比如有些情况下，服务器上的文件只是变化了修改时间，但是内容本身并没有任何变化，那么前面的缓存判断机制都会以为内容变了需要重新拉取，另外的问题就是 If-Modified-Since 的时间精度是到秒，如果在秒以下修改，是无法判断的。Etag 的基本原理就是通过算出文件的哈希值，便于判断文件内容是否修改。需要说明的是 Etag 的计算方式在规范中并没有明确规定，一般是取 inode+size+LastModified 进行哈希。

3.1.2 Web 前端性能测试方法

前面部分介绍了 HTTP 协议的一些基本交互过程，和 TCP 的协作，以及性能相关的特性，接下来讨论具体的 Web 前端性能测试方法。

HTTP 前端常用的性能测试工具非常多，比如 Fiddler、YSlow、HttpWatch、Firebug，以及各个浏览器基本都自带的开发者工具。在使用过不同的工具之后，会发现除了个别功能，大部分前端分析工具提供的信息都差不多。下面介绍部分常用的工具。

3.1.2.1 常用 Web 性能测试工具

首先最常用的是浏览器自带的开发者工具。图 3-30 是 Chrome 浏览器自带的 PageSpeed 分析工具，在页面加载完成后就可以看到分析的结果，并在数据分析的基础上提供了一个建议。

在 Web 前端性能方面，Yahoo 整理了比较全面的 35 条规则，可以作为非常好的参考，这里不再赘述了。见 https://developer.yahoo.com/performance/rules.html。

另外，这里介绍一个在线的工具 WebPageTest（http://www.webpagetest.org），可以快速获取到丰富的信息，并提供了一些统计功能。

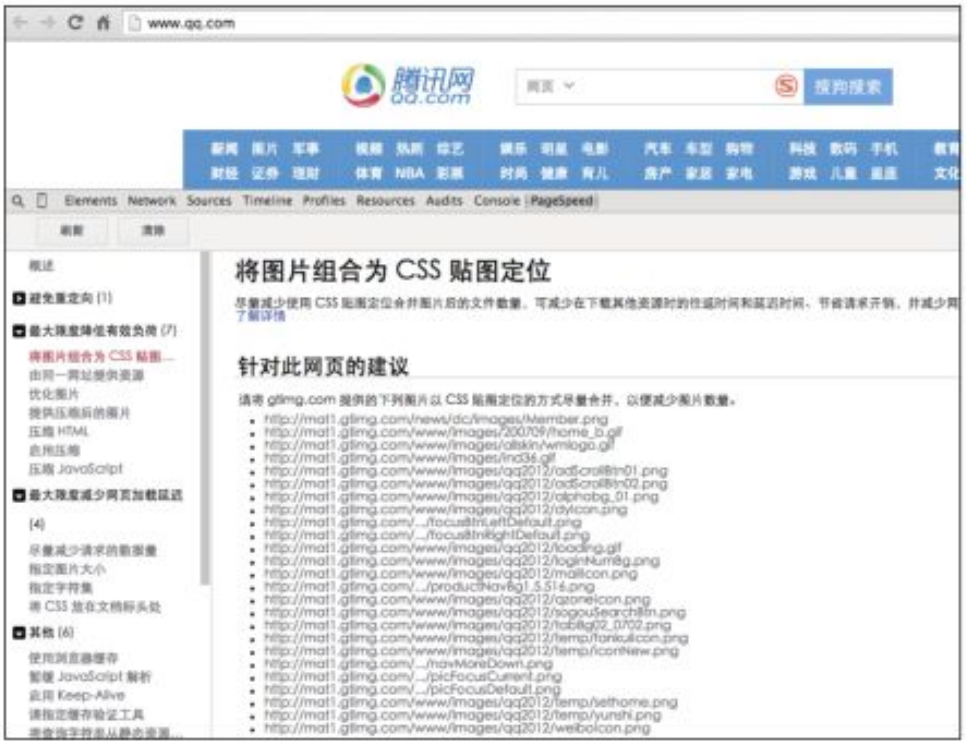


图 3-30 qq.com 首页的 PageSpeed 分析结果

图 3-31 所示是针对 qq.com 的一次测试结果的 Summary 信息。

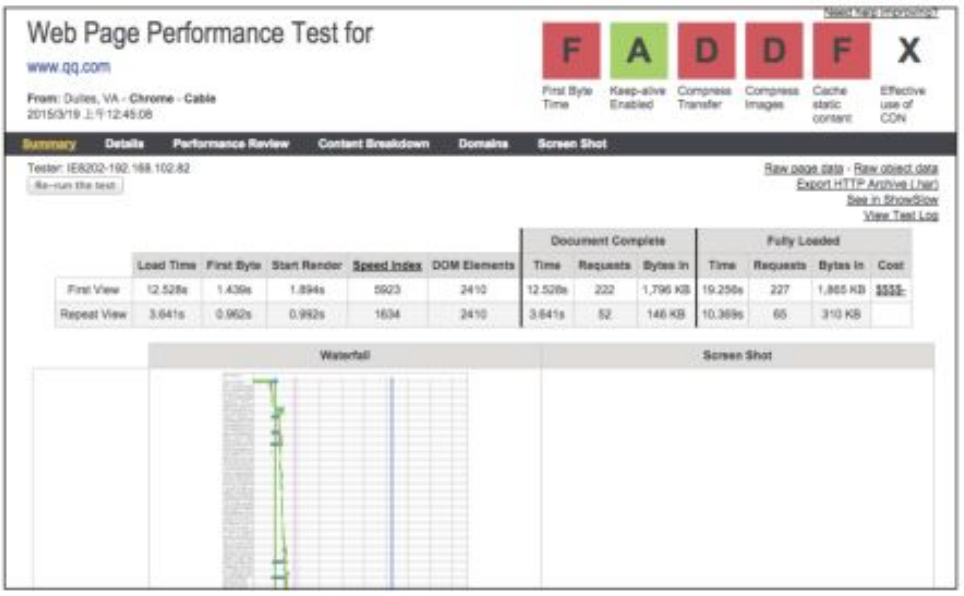


图 3-31 WebPageSpeed 提供的测试报告 -Summary

图 3-32 所示是对应的详细信息，可以看到单个请求的详细时间，包括 DNS 查询时间、连接建立时间、Time to First Byte 的时间（后面 3.3.3.1 节中有相关介绍）和内容下载时间。

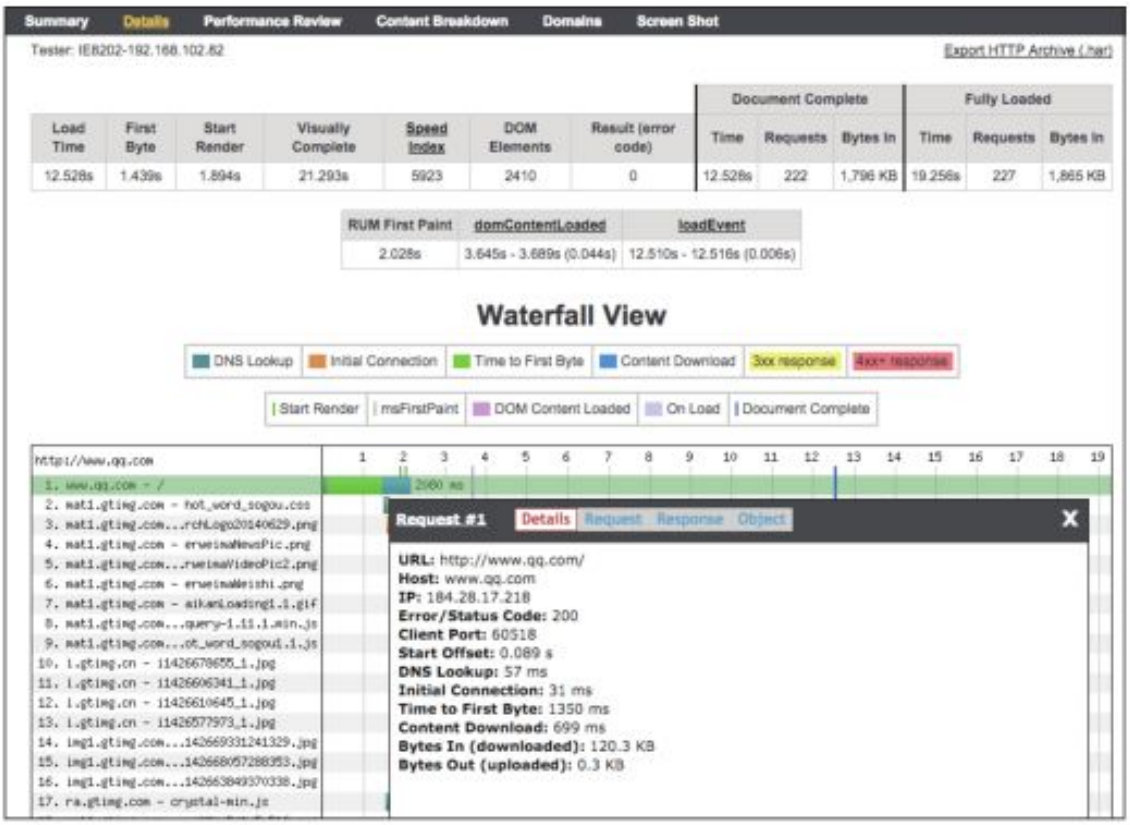


图 3-32　WebPageSpeed 提供的测试报告 - 详细信息

图 3-33 所示是根据域名来做的请求数量和传输内容大小的细分。

图 3-34 所示是根据内容的类型来做的请求数量和传输内容大小的细分。

通过以上统计信息和单个请求的维度，可以很快找到前端性能的瓶颈。当然，实际项目中，Web 前端性能的优化可能需要前后台多方面的努力。

3.1.2.2　Chrome PC/ 手机远程调试

目前大部分的前端分析工具都是针对 PC/Mac 机的，而如前面所提到的，手机上通过浏览器来访问网页也是普遍的需求，而且 M 站也是移动业务的重要产品。那么从测试角度，如何来获取手机浏览器上的类似调试工具给出的信息呢?

主要有两个方法，一是采用上面提到的 WebPageTest 之类的在线工具，另一个方法是采用 PC 浏览器 + 手机浏览器的远程调试模式。下面以 Chrome 浏览器为例介绍第二种方式的做法。

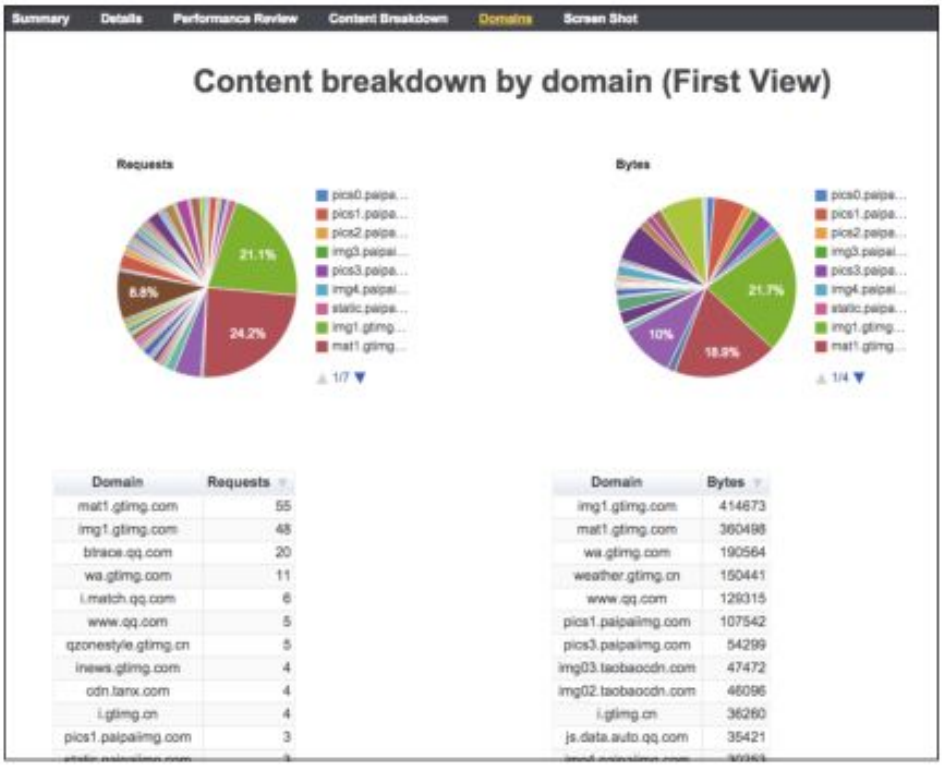


图 3-33 WebPageSpeed 提供的测试报告 – 按域名细分

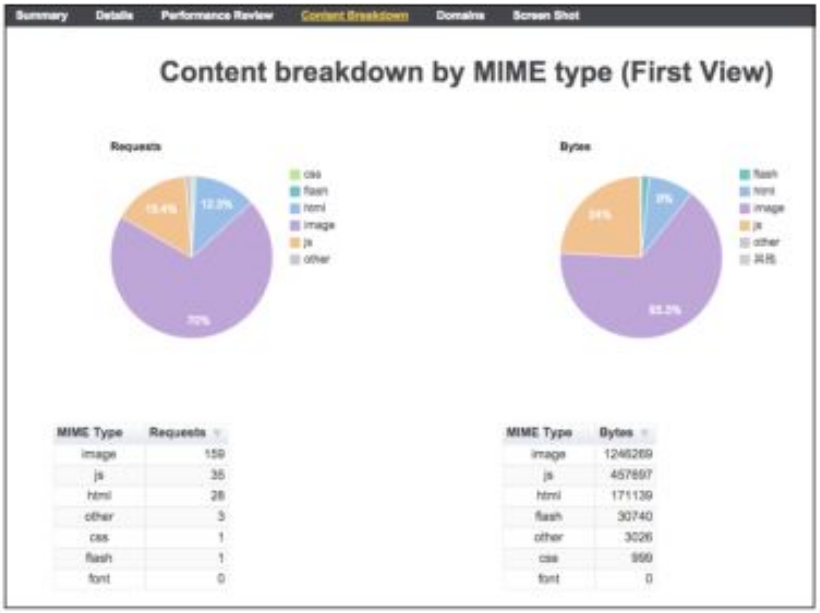


图 3-34 WebPageSpeed 提供的测试报告 -MIME type 细分

首先，需要在 PC/Mac 浏览器上安装 Chrome 的 ADB 插件，如图 3-35 所示。

图 3-35　Chrome 中的 ADB 插件

接下来将手机通过 USB 线和 PC/Mac 连接，在 PC/Mac 的 Chrome 浏览器里打开下面的 URL ： chrome://inspect/#devices 就能在 PC/Mac 浏览器的页面里看到连接的手机和当前手机上 Chrome 中打开的页面（注意必须是 Chrome 浏览器打开的页面），如图 3-36 所示。

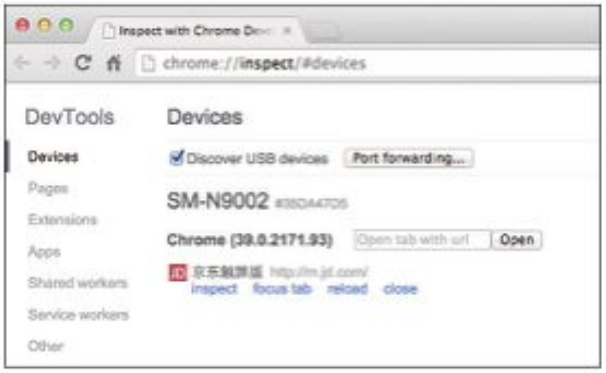


图 3-36　通过 ADB 连接手机浏览器的内容

然后，点击图 3-36 中对应页面下面的“ inspect”，就可以打开 Chrome 的开发者工具，如图 3-37 所示。

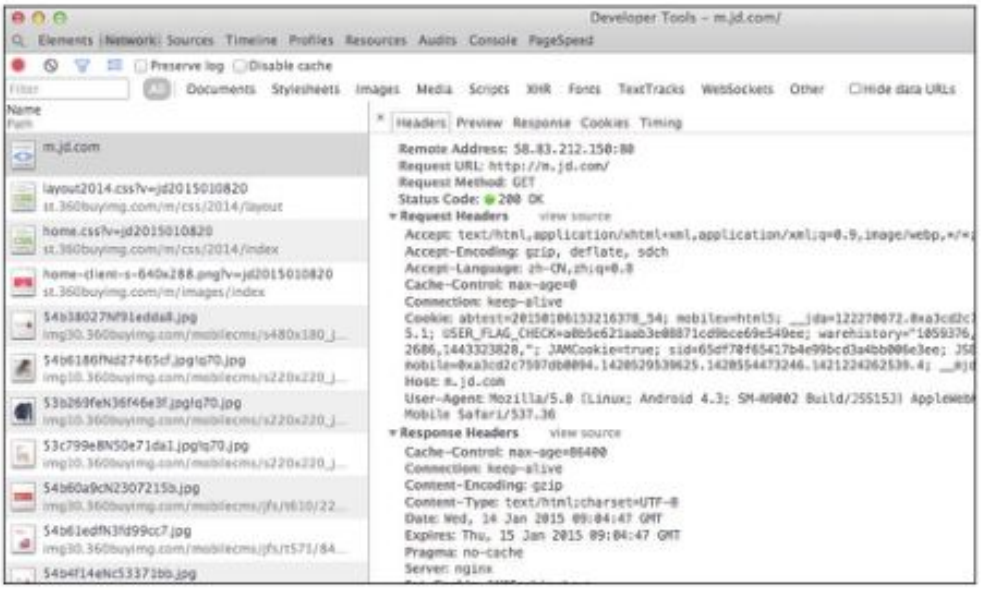


图 3-37　通过开发者工具看到的手机上页面的内容

在这种情况下，手机上的操作都会反映到这个开发者工具上面，就可以像分析 PC 浏览器页面一样的分析手机浏览器上各个请求。如图 3-38 所示是手机浏览器上某个请求的网络耗时分析。

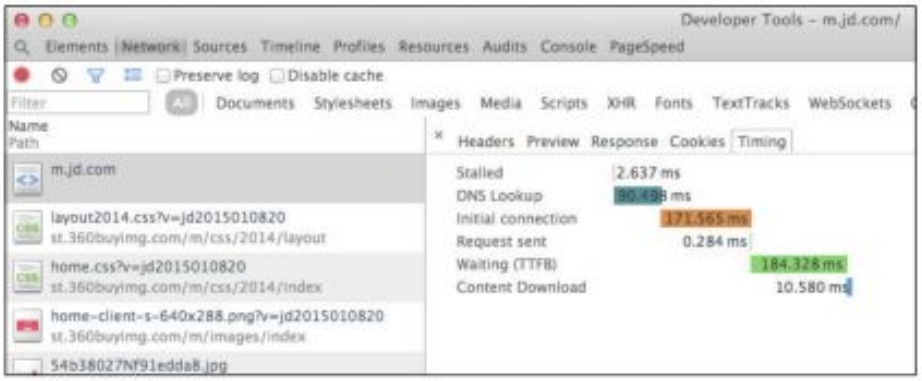


图 3-38 手机浏览器上某个请求的网络耗时分析

由于 PC Chrome 和手机 Chrome 之间是以 USB 的方式来通信的，所以这个模式也可以获取手机以 2G/3G/4G 移动网络方式获取的数据。

切换到移动网络，可以在 DNS 查询时间比较长的情况，如图 3-39 所示。

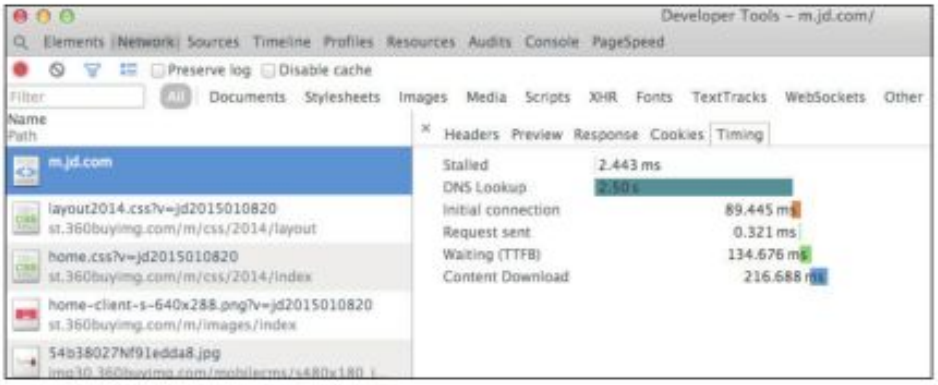


图 3-39 手机浏览器上请求的 DNS 耗时较长情况

相应的，同样可以使用 PageSpeed 功能来进行分析，和 Web 性能的最佳实践做一些对照，给出一些参考意见。

通过上面介绍，可以看出如果只是简单地使用工具，门槛其实非常低，更重要的是数据的解读和分析，而这方面依赖于对具体技术细节的理解，以及对被测产品的架构和部署等方面的深入了解。

图 3-40　通过 PageSpeed 分析手机浏览器所访问页面的性能

3.2　App 端性能测试

App 端本身的性能是影响用户体验的非常重要的方面，包含的内容非常多，例如 CPU、内存的使用情况，以及如何快速完成页面渲染，有很多也是和具体的 App 面向的领域相关，比如手游的 App 可能涉及游戏 2D、3D 引擎方面的问题。在这里我们重点讨论两个方面，一是内存问题的分析，二是 App 内嵌 Web 组件的性能分析。接下来分别就 Android 和 iOS 两个平台，针对以上两个方面结合示例来进行讨论。

首先我们来看内存方面的问题。一个好的 App 除了优秀的用户体验之外，程序的内存性能也是至关重要。虽然这个方面用户无法体验到，但是作为测试人员，我们却无法避开这个问题。因为一旦发生内存泄漏的问题，轻则影响到 App 的运行性能，严重的则会导致内存告警，程序崩溃。

3.2.1　Android 内存问题分析

就 App 实际项目的经验来看，Android 平台应用的内存问题比较容易出现，主要的症状是内存使用过高，以及因为内存不够而导致的崩溃。接下里我们针对基于 Java 的 Android 应用，了解一些内存管理方面的知识，并介绍如何收集内存 dump 文件，并用 MAT 工具进行分析，找出内存使用方面存在的问题。

3.2.1.1　Java 内存管理介绍

我们在分析 Android 内存的时候，首先需要对 Java VM 的垃圾回收机制有一定的了解

（Android 系统在这一部分，内部逻辑基本跟桌面的 Java VM 一致），这样才能知道某个对象当前驻留在内存中是正常现象还是异常现象。一些使用图片较多的 App，内存会比较大，但是不能因此认为这个 App 有内存方面的问题。此外，我们还需要了解 Java 中的几种引用类型。不同引用类型，在垃圾回收的时机上是不同的。而通常我们在做内存分析的时候只需要关注部分的引用类型即可。

1. 堆内存垃圾回收机制

Java 的堆内存回收机制比较复杂。限于篇幅，此处只介绍跟下文内存分析较为相关以及相对容易理解的基础知识。

在每个进程中，会有一个垃圾回收线程负责检查是否有没有引用到的对象可以被释放掉，我们称为标记过程，如图 3-41 所示。

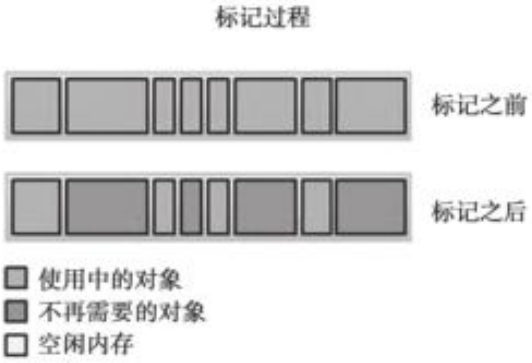


图 3-41 垃圾回收标记过程

这个过程中会从各个垃圾回收根出发，递归遍历它们引用的对象，生成由不可以被回收的对象组成的图的数据结构。生成完毕后，除了这些图状数据结构中的对象，其他对象占用的堆内存被认为是可以被回收的。垃圾回收根主要包括：

- 静态变量
- 栈上指向的堆内存对象
- 寄存器
- 其他（为了简化理解，不讲解其他）

静态变量很好理解。显然静态的对象和它引用到的其他对象是不能被释放的。栈上指向的堆内存对象是指在方法调用到一半的时候 CPU 时间给了垃圾回收线程执行，这时原来方法的本地变量还是存储在栈上。显然此时这些本地变量不能被释放，否则当执行原来方法的线程还原现场并继续执行后会产生问题。寄存器也是同样道理，方法的入参部分可能会由寄存器存储而不是存储在栈上，因此也不能被回收。

在区分了可以被回收和不可以被回收的内存后，垃圾回收线程可以回收内存对象。但是这么做会留下内存上的空白，我们称为内存碎片。内存分配器在分配新对象内存的时候需要根据本身维护的内存碎片的链表来进行内存分配，如图 3-42 所示。

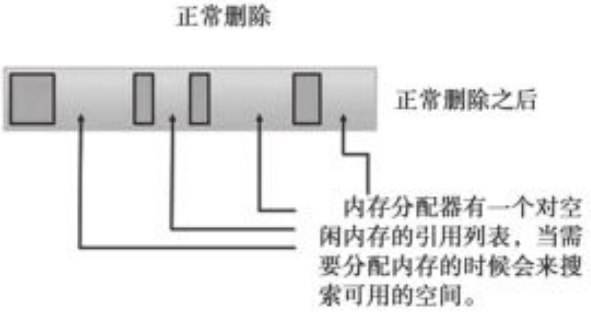


图 3-42　垃圾回收删除过程

在一些情况下，垃圾回收线程也会压缩掉内存碎片。但是这个过程较为耗时。所以压缩的触发不会非常激进。压缩完毕后如果有新对象需要分配内存，会从堆顶部进行分配，如图 3-43 所示。

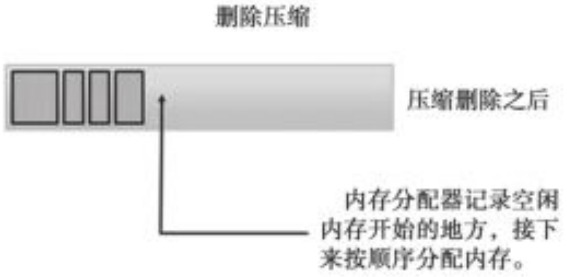


图 3-43　垃圾回收删除压缩过程

2. Java 的几种引用

Java 程序中开发可能会使用到如下几种引用：

- 强引用
- 弱引用
- 软引用
- 其他（为了简化理解，不讲解其他）

其中强引用是最为常见的。如下代码即使用了强引用：

```
ClassA a=new ClassA();
```

引用 a 即为一个强引用。在构建不可回收对象图的过程中，如果遍历到一个强引用，

那么这个引用指向的对象会被加入到图中。

如下代码使用了弱引用：

```
WeakReference<ClassA> a_weak=new  WeakReference<ClassA>(new ClassA());
```

即便 a_weak 作为垃圾回收根，它指向的堆内存空间也是会被回收的。我们可以通过：

```
a_weak.get();
```

判断引用指向的对象是否已经被回收，如果被回收则返回 null，否则返回引用的对象。在一些场景我们会用到弱引用。例如，我们可以弱引用机制来实现内存中大对象的缓存。

软引用跟弱引用类似。如下代码使用了软引用：

```
SoftReference<ClassA> a_soft=new  SoftReference<ClassA>(new ClassA());
```

用法也跟弱引用完全相同。唯一的区别是，垃圾回收器对软引用会采用比较保守的回收策略，只有当特定条件，例如进程内存占用非常高等等，才会对软引用指向的内存空间进行回收操作。所以，我们对一些更希望对象在内存中驻留的场景，可能更倾向使用软引用而非弱引用。

3.2.1.2 Android 内存占用分析实践

MAT 是一个强大的 heap dump 内存分析工具。在测试中我们可以使用它来发现一些内存问题。我们可以从官网上下载 MAT 工具 www.eclipse.org/mat/downloads.php。

下面以 Windows 环境为例，说明如何使用工具进行内存分析。

1. heap dump 抓取

首先我们需要抓取进程的 heap dump。Eclipse 的 DDMS 和 Android Studio 的 DDMS 都能进行抓取工作。图 3-44 所示是 Android Studio 1.0 的 DDMS 界面，我们可以选择进程并点击“Dump Java Heap”进行 heap dump 抓取。抓取到的 dump 文件以 .hprof 结尾。

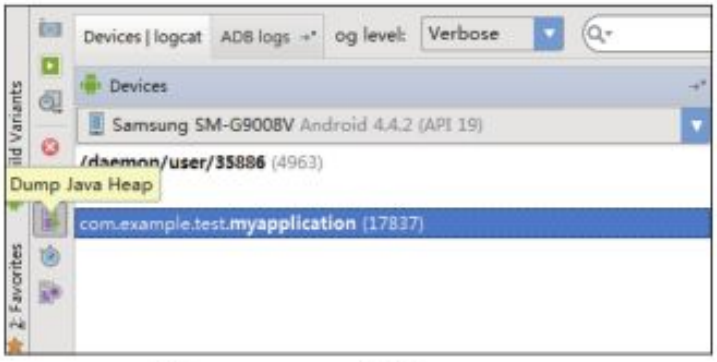


图 3-44 DDMS 中抓取 heap dump

2. heap dump 分析

运行 MemoryAnalyzer.exe，并打开抓取的 .hprof 文件，会出现图 3-45 所示界面。

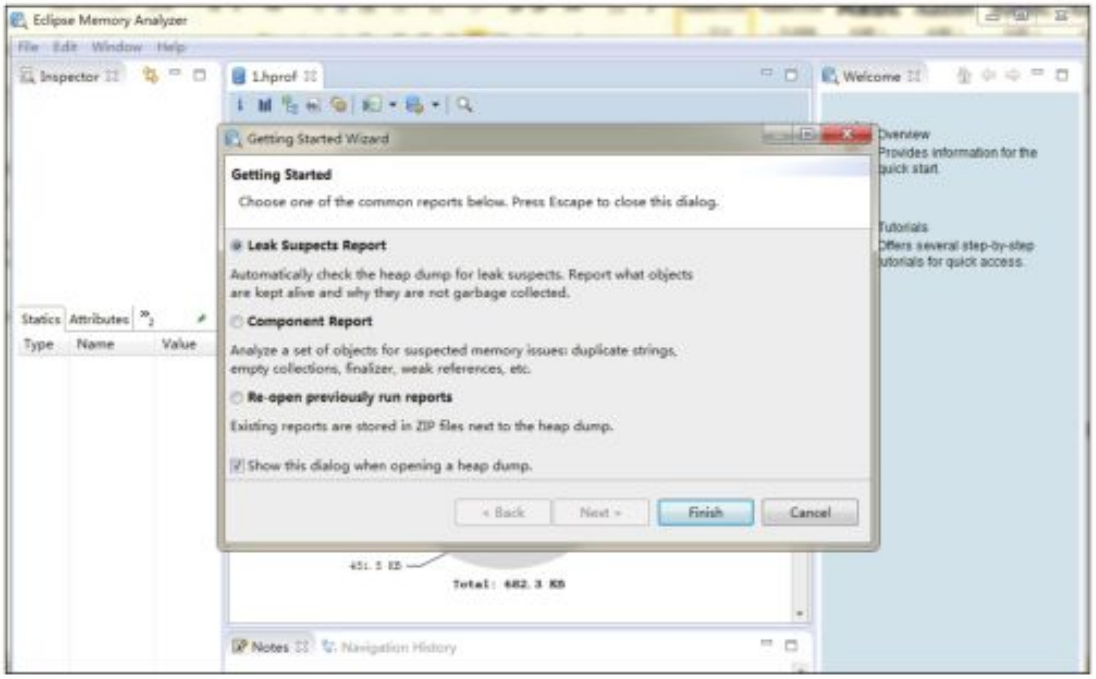


图 3-45　MAT 中打开 heap dump

这里我们可以选择让 MAT 工具帮我们自动分析。我们点击 Cancel。有兴趣的读者可以尝试自动分析并查看报告。

在进行分析时，我们主要关注以下两个问题：

- 大对象常驻内存。
- 内存泄漏。

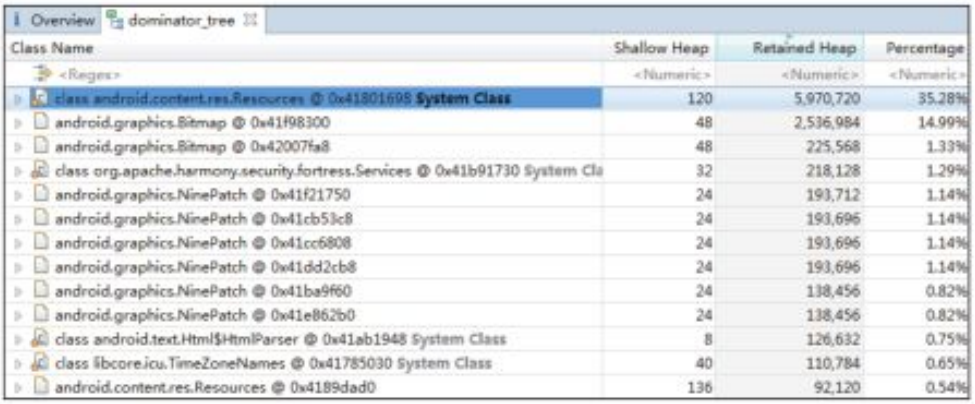
下面分别介绍。

A．常驻内存的大对象问题及解决方法。

如果在内存分析中发现有很大的内存对象存在，需要评估是否是必须的，是否可以优化。

通常，我们可以点击“ Open Dominator Tree for entire heap”按钮来查询内存中最大的对象，如图 3-46 和图 3-47 所示。

图 3-46　MAT 中打开 Dominaotor Tree



Overview | dominator_tree

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
class android.content.res.Resources @ 0x41801698 System Class	120	5,970,720	35.28%
android.graphics.Bitmap @ 0x41f98300	48	2,536,984	14.99%
android.graphics.Bitmap @ 0x42007fa8	48	225,568	1.33%
class org.apache.harmony.security.fortress.Services @ 0x41b91730 System Cla	32	218,128	1.29%
android.graphics.NinePatch @ 0x41f21750	24	193,712	1.14%
android.graphics.NinePatch @ 0x41cb53c8	24	193,696	1.14%
android.graphics.NinePatch @ 0x41cc6808	24	193,696	1.14%
android.graphics.NinePatch @ 0x41dd2cb8	24	193,696	1.14%
android.graphics.NinePatch @ 0x41ba9f60	24	138,456	0.82%
android.graphics.NinePatch @ 0x41e862b0	24	138,456	0.82%
class android.text.Html$HtmlParser @ 0x41ab1948 System Class	8	126,632	0.75%
class libcore.icu.TimeZoneNames @ 0x41785030 System Class	40	110,784	0.65%
android.content.res.Resources @ 0x4189dad0	136	92,120	0.54%

图 3-47 Dominaotor Tree 结果列表

在结果列表中，Shallow Heap 代表这个对象直接占用的内存大小。Retained Heap 表示这个对象直接占用的内存，以及由它出发，所有引用到的对象占用的内存大小总和。或者简单而言，就是一旦这个对象被回收，可够释放掉多少内存。在内存分析中，我们主要关注 Retained Heap。

那么如何判断大对象是否是可以优化掉的呢？通常我们需要注意以下几点：

- android.content.res.Resources 占用内存较大一般是正常现象，通常不用过于关注。此处的优化可以从资源文件（例如图片）的大小着手。
- 如果占用内存较大的前几位对象的包名是应用程序代码中的包名，则可重点关注。
- 是否有当前不在屏幕上的图片依然驻留内存。

这里主要着重介绍如何分析内存中的位图，因为目前大多数 App 的图片内存问题都是由于图像对象引起，所以这一分析方法非常有效。我们在 MAT 工具中左键点击一个 Bitmap 对象后，可以在 Attributes 视图中获取 mWidth，mHeight。分别为该图片的宽和高。我们还能获取 mBuffer，即图片的原始数据字节数组。可以通过右键 ->Copy->Save Value to File 将这个数组里的数据以文件形式保存下来，如图 3-48 所示。

我们把文件存为 < 文件名 >.rgba。

之后可以使用图片转换工具将此文件转为便于查看的图片格式。比如，我们可以使用 ImageMagick 工具。

ImageMagick 转换工具的使用例子：

```
<ImageMagick安装路径>\convert.exe" -size 1080x354 -depth 8
   <上一步保存的文件路径和文件名>.rgba 1.png
```

上述命令将当前目录下的 1.rgba 转化为 1.png。其中 1080 为用 MAT 工具获得的位图宽度，354 为用 MAT 工具获得的位图高度。

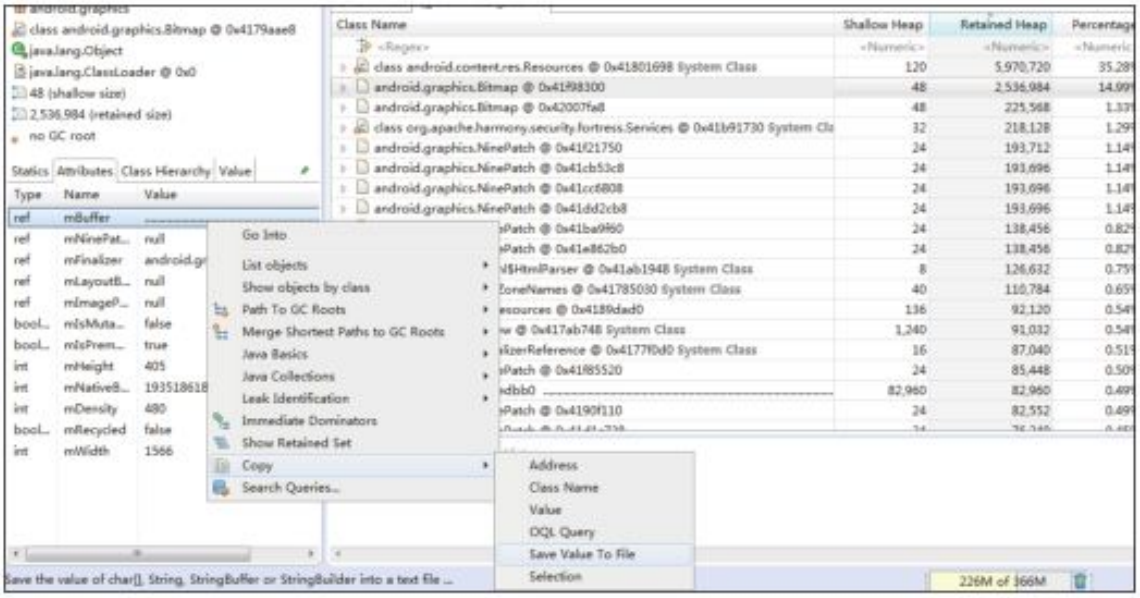


图 3-48　保存内存中的 Bitmap 到本地文件

之后查看转换后的图片我们就能知道 MAT 窗口中显示的 Bitmap 其实是什么图片了。我们可以进一步分析这个图片在抓取 dump 的时候驻留在内存中是否合理。如果不合理，可以进行优化。

找到可疑的对象或者确定可以优化的对象后，我们可以右键点击，选择 Path To GC Roots->exclude weak/soft references 寻找这些对象是被哪些对象引用的，进而能定位到代码层面，如图 3-49 所示。这里我们排除掉弱引用和软引用，因为它们会被内存回收，不用重点关注。在一些情况下，测试人员不一定能了解代码结构。用这个方法也可以作为探索代码的入口点。

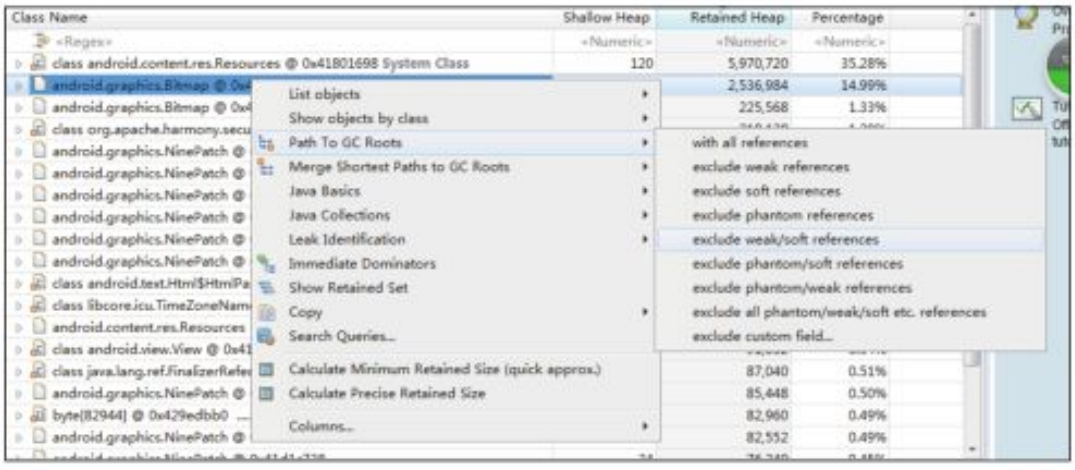


图 3-49　追踪 GC 根

我们会看到类似图 3-50 的表格。

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
android.graphics.Bitmap @ 0x41f98300	48	2,536,984
mBitmap android.graphics.drawable.BitmapDrawable$BitmapState @ 0x41	40	40
[249] java.lang.Object[510] @ 0x41d2fcd0	2,056	2,056
mValues android.util.LongSparseArray @ 0x41894f48	24	6,176
[0] android.util.LongSparseArray[2] @ 0x41894f28	24	5,946,736
sPreloadedDrawables class android.content.res.Resources @	120	5,970,720
[10] java.lang.Object[14] @ 0x429d0990	72	72
Σ **Total: 2 entries**		
mBitmap android.graphics.drawable.BitmapDrawable @ 0x42a5c078	72	96
mGlow android.widget.EdgeEffect @ 0x42a5bf58	136	368
mEdgeGlowTop android.widget.ListView @ 0x42a5b928	1,248	5,944
mServedView, **mNextServedView** android.view.inputmethod.Inp	120	800
mList com.example.test.myapplication.SettingsActivity @ 0x42a16	288	760
Σ **Total: 2 entries**		
Σ **Total: 2 entries**		

图 3-50 查看 GC 根路径

图中这个 Bitmap 对象有两条 GC 根路径，比如由资源部分引用链产生，以及由使用 android.widget.ListView 对象的引用链产生，等等。如果要彻底在内存中去除对象，需要打断此对象所有的 GC 根路径或者直接将此对象置为 null 才行。

B．内存泄漏问题及测试。

我们可以进行一些重复性的操作。例如反复进入 Activity 再回退，查看是否有不必要的对象依然驻留内存。

对于测试来说，一个高效的测试内存泄漏的方法是使用 MAT 中的 OQL。此功能可以通过点击 MAT 主界面工具栏上的 OQL 按钮打开，如图 3-51 所示。

图 3-51 MAT 中打开 OQL

我们可以通过 OQL 语句查询。这是一个类似 SQL 的语句，用来查询内存中的对象。例如，我们可以使用如图 3-52 所示的命令查询堆内存中所有类名包含 Bitmap 的对象。

OQL 的优点是，我们可以根据类名直接进行查询。对于一个测试人员来说，可能只需要关注部分的代码。这样使用 OQL 能够精准定位到该测试人员负责的代码部分，大大提高了分析的效率。通常，我们可以通过包名来查询自己负责的类。之后使用上文所述的方法查询占用内存较大的对象的 GC 根路径进行分析即可。

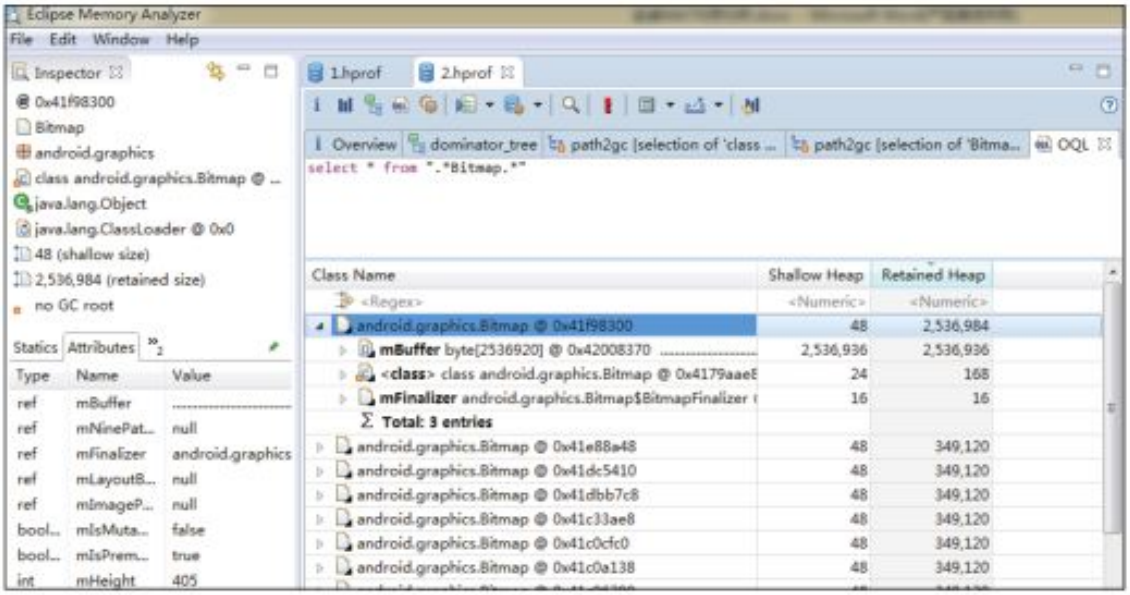


图 3-52　使用 OQL 查询内存中的对象

要了解更多 OQL 语法，可以按 F1 键调出帮助菜单，如图 3-53 所示。

图 3-53　查看 OQL 帮助文档

总之，MAT 是一款强大的分析 Android 应用堆内存的工具，我们可以用它来发现 App 的内存泄漏等问题。通过对图片的分析可以帮助我们定位出内存中的位图对象是哪个图片。使用 OQL 可以提升分析的效率。通过追踪 GC 根并结合阅读代码，我们可以进一步将内存问题定位到代码层面。

3.2.2　iOS 内存问题分析

前面讨论了 Android 方面的内存分析，接下来我们看看 iOS 平台的内存问题分析的方法。

1. iOS 的内存管理

iOS 的内存管理使用了引用计数器的概念。什么是引用计数器呢？简单来说就是内存空间被引用的次数，当该内存空间的引用计数器为空时，该内存空间才会被系统释放。那怎么样去申请内存，释放内存呢？我们知道，初始化一个变量，给一个空变量赋值，都会申请内存空间。在 iOS 中，我们需要关心三个关键词：alloc(或 new)、retain、copy。当我们用到了这三个关键词去初始化变量，或者给变量赋值时，我们有责任去释放变量。而使用其他方式的初始化或者赋值，我们就不用关心释放的问题了，因为 iOS 系统会自动帮我们释放掉它们。内存管理属于 iOS 开发人员必须了解的内容，同时通过了解这些内容，可以帮助测试人员更好地定位 App 中内存泄漏问题。

2. ARC 模式

苹果在 iOS5 中新推出了 ARC 的概念，ARC 指 Automatic Reference Counting，它的主要功能是自动帮我们做了释放的操作，开发代码可以省去很多释放的操作语句，大大地减少了内存泄漏的风险。但是使用了 ARC 的模式开发的 App，不代表就没有内存泄漏的问题了。

3. 内存泄漏测试的准备工作

要做 iOS 内存泄漏测试，我们需要准备这些工具：

- 项目工程源码。
- Instrument 工具。
- 开发调试证书。
- 苹果设备。

这里推荐使用真机调试而不是模拟器，因为很多问题在模拟器上无法发现，iOS 很多性能数据在真机和模拟器上，有时会差别很大。毕竟我们的 App 是安装在设备上使用的，在真机上测试能更好地发现问题。

4. 测试步骤 (以 Xcode6 为例)

在 Xcode 上部导航栏里选择要编译的 Target，选择连接的真实设备，然后点击 Edit Scheme，如图 3-54 所示。

在 Profile 设置栏里，选择 Debug 模式，点击 Close，如图 3-55 所示。

调试证书和 provision 文件的获取这里就不多介绍了，可以到苹果开发者网站上了解到更多信息 (https://developer.apple.com)。这里我们确保调试证书和 provision 文件都选择正确，如图 3-56 所示。

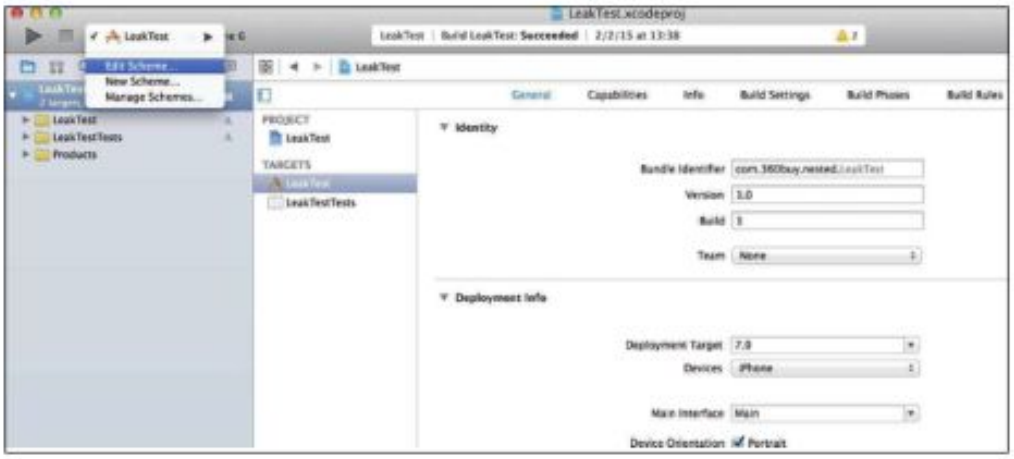


图 3-54　进入 Scheme 设置

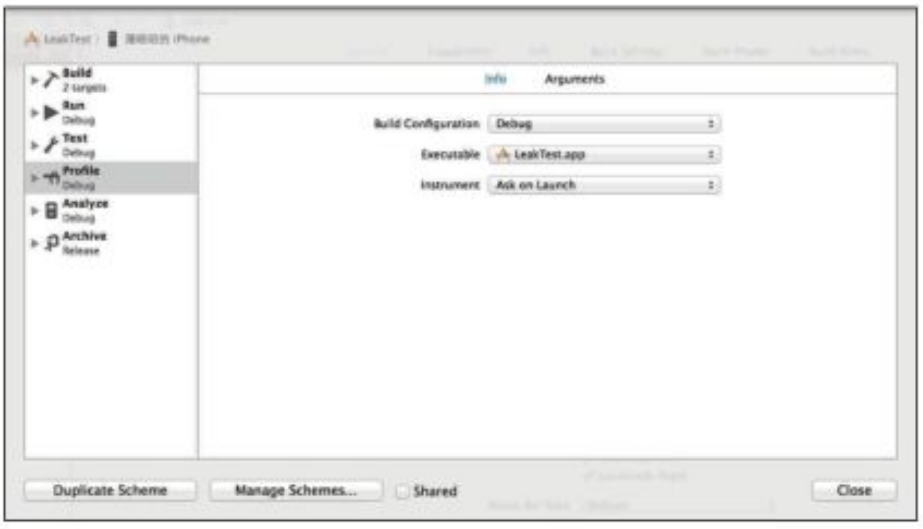


图 3-55　Scheme 设置界面，设置 Profile 编译模式

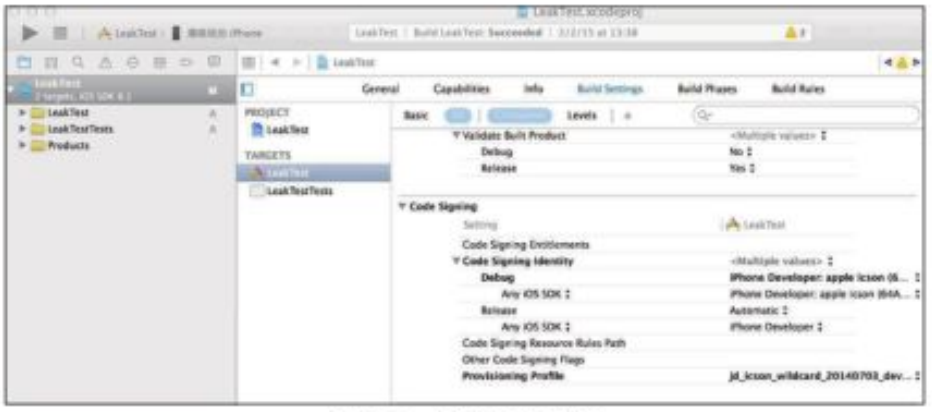


图 3-56　证书设置界面

点击 Product 菜单下的 Profile 来编译工程，如图 3-57 所示。

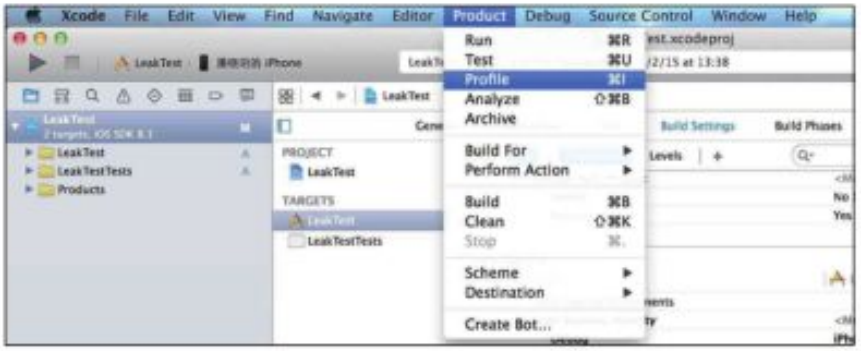


图 3-57 以 Profile 方式编译运行

等待项目工程的编译和安装完成后，会弹出 Instrument 的菜单界面，然后选择 Leaks 工具，如图 3-58 所示。

图 3-58 Instrument 主界面，选择 Leaks 功能

点击左上方的录制按钮，App 就启动了，我们可以边在真机上执行测试，边在 Leaks 上看到泄漏结果，如图 3-59 所示界面上可以看到泄漏对象、泄漏大小、申请内存操作等。

我们也能切到调用堆栈方式来查看泄漏详情，定位到代码级别，如图 3-60 所示。

5. 测试案例分析

A. 最常见的场景就是类成员变量在 dealloc 中未被释放。

解决方法：在 dealloc 方法中，添加对该成员变量的释放操作，如图 3-61 所示。

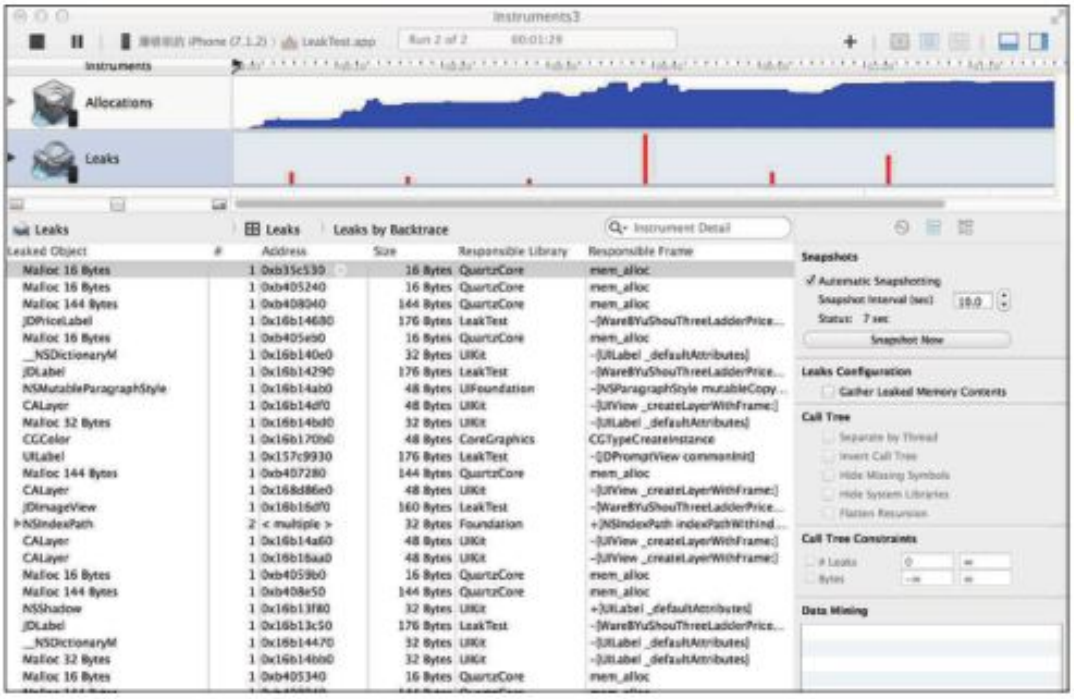


图 3-59　Leaks 按 leak 对象分类界面

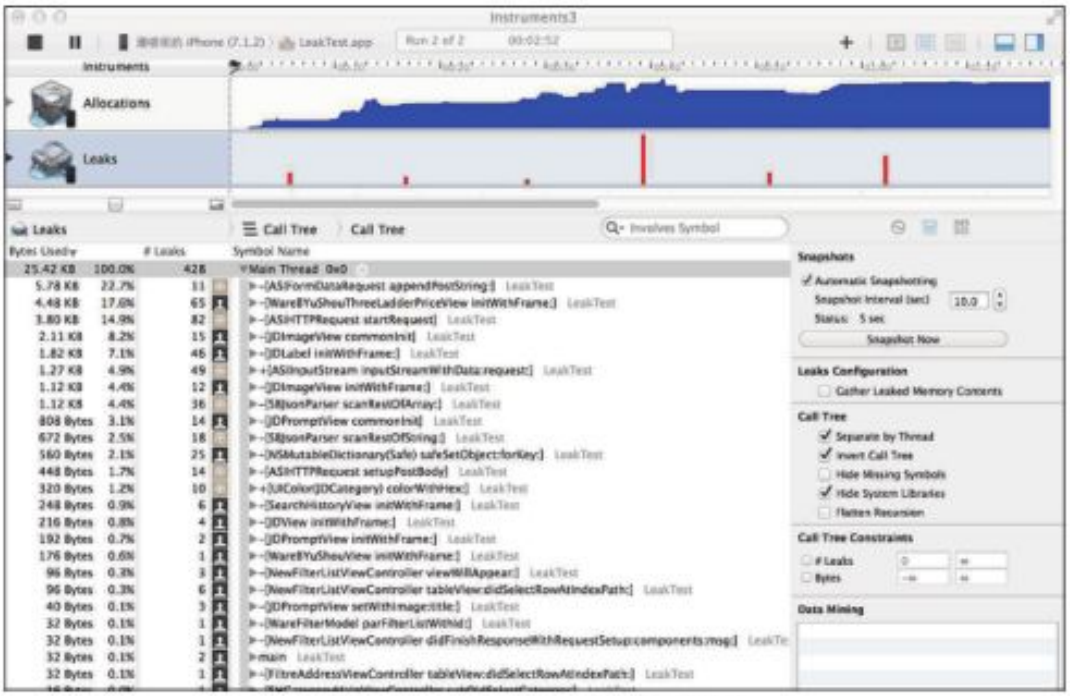


图 3-60　Leaks 按 leak 堆栈方式展示

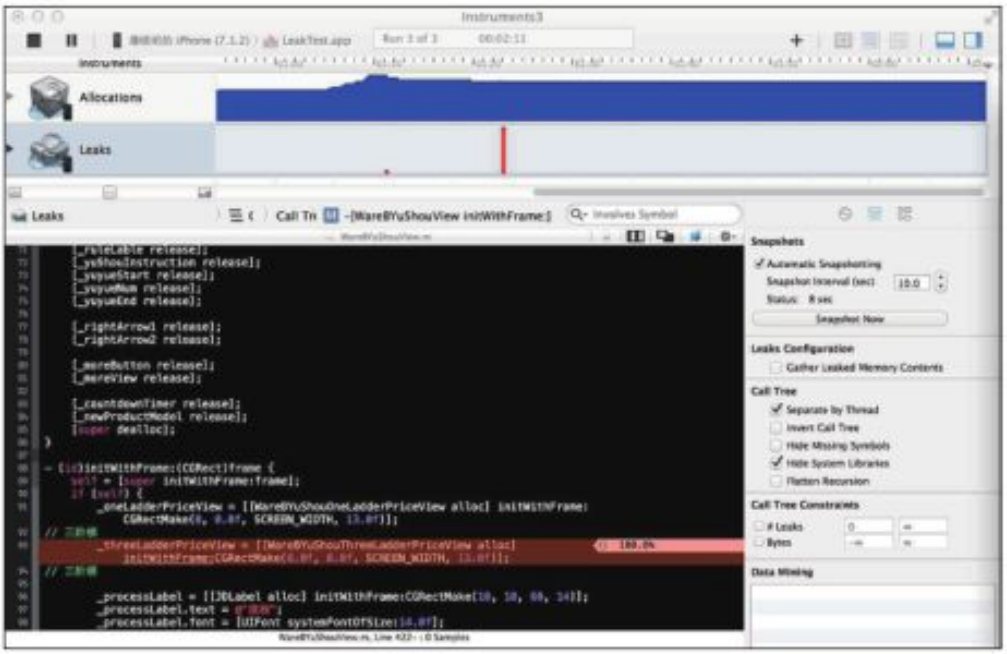


图 3-61 dealloc 方法中未释放成员变量

B. 方法中的局部变量，申明了增加计数器，但直到方法结束，都没有释放操作，如图 3-62 所示。

解决方法：在方法结束前，释放该变量。

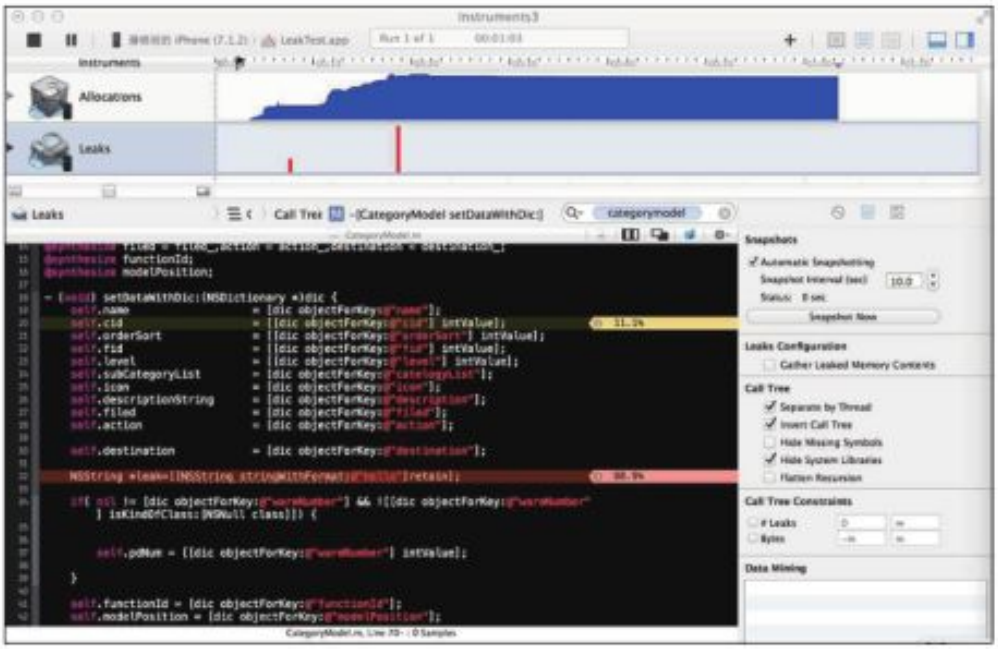


图 3-62 局部变量在添加引用计数后，未在方法中释放

C. 释放操作无效。未使用 iOS 自带 setter 方法的变量，直接等号赋值 nil 不会释放掉之前引用的内存地址，导致之前的值泄漏，如图 3-63 所示。

解决方法：使用 self. 的方法赋 nil 值，或者直接用变量释放操作。

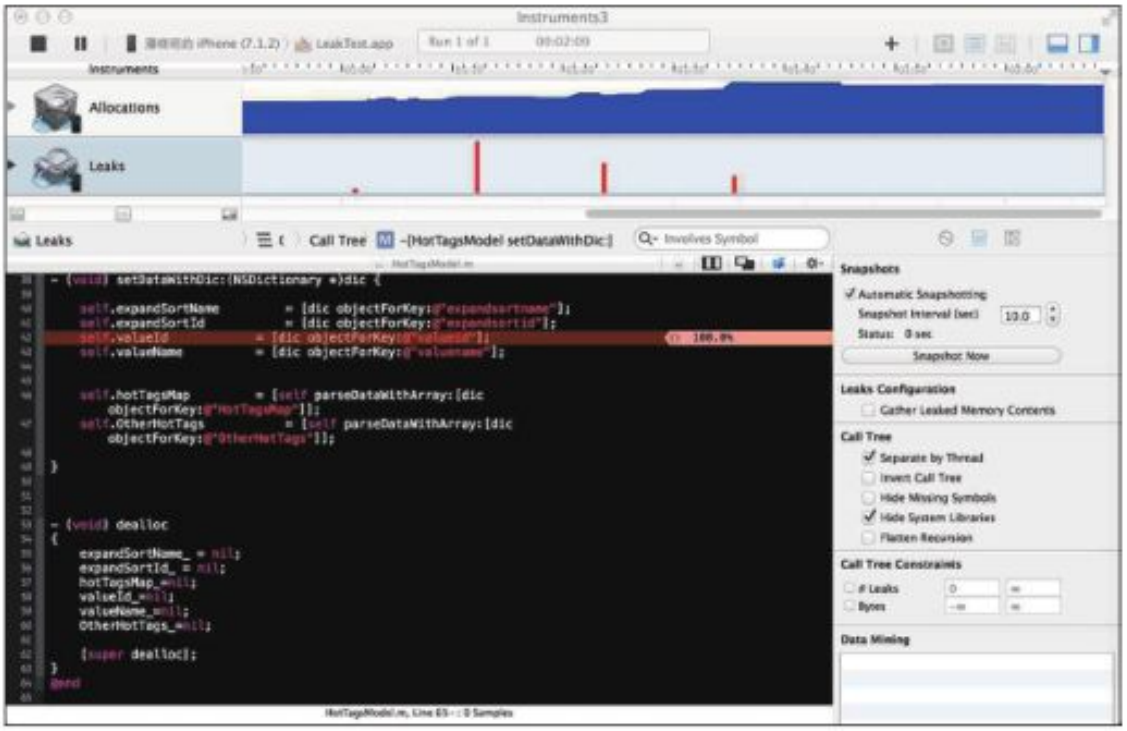


图 3-63　错误的释放操作

D. 使用 self. 方法赋值，同时使用 autorelease 方法以为释放正常。这个问题的根源在于 self. 的方法会 retain 一个计数，而 autorelease 只会减去一个计数而已，自然引发泄漏，如图 3-64 所示。

解决方法：在后面添加释放操作，或者不用 self. 的方法赋值。

E. 类成员变量在类中未被使用到，因此在 dealloc 方法中缺失该变量的释放操作。看起来没有什么大问题。不过如果该类成员变量在外部被赋值，则会导致泄漏，如图 3-65 所示。

解决方法：保证类中那些公共的非基本类型的成员变量都有释放操作。记住一个原则，谁对变量有拥有权，谁就有责任去释放它。

当然，我们也不能完全依赖 Leaks 标记的泄漏地方，因为有时候它的标记是不准确的。比如上面图 3-63 中出现的情况，Leaks 无法精确地标记到无效的释放操作，不过它却指出了这个类存在内存泄漏，我们只要顺藤摸瓜，在调用堆栈中一层层地分析，就能定位到内存泄漏的地方。

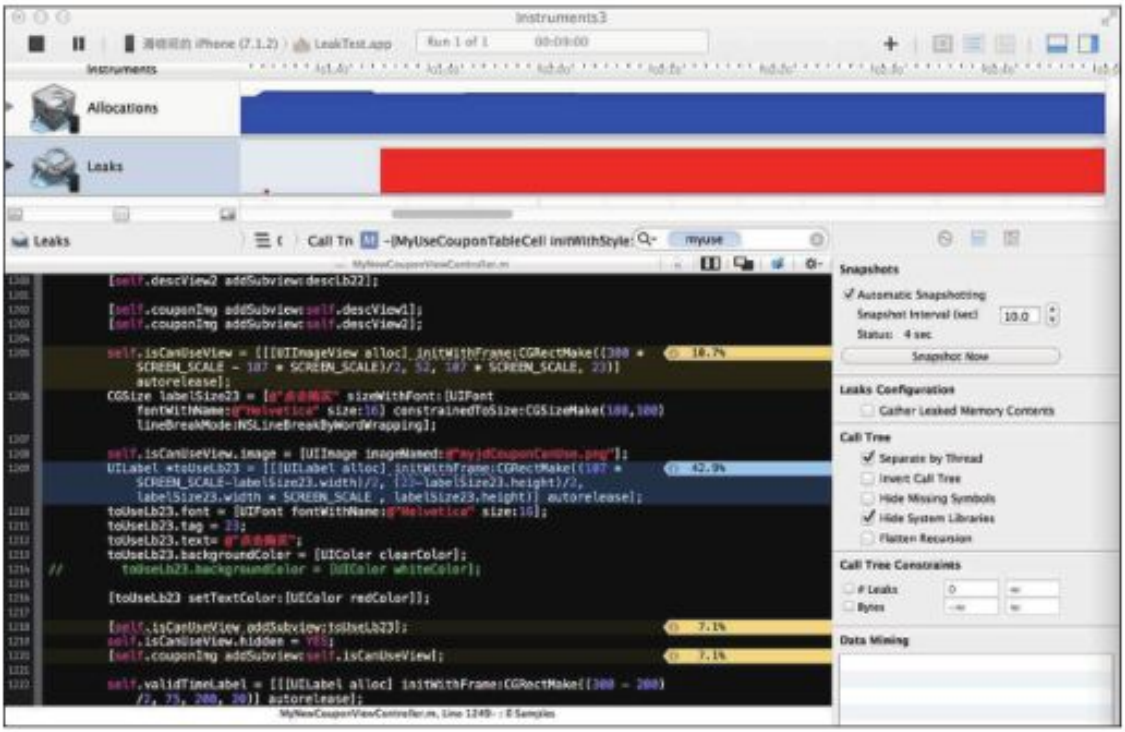


图 3-64 遗漏的释放操作

```
MessageCenterViewController *secondMsgList = [[MessageCenterViewController alloc] init];
secondMsgList.firstMsgModel = [_listDatas objectAtIndex:indexPath.row];
secondMsgList.msgViewTitle = msgViewTitle;
```

图 3-65 外部赋值导致泄漏

3.2.3 App 内嵌 Web 组件的性能分析

很多的 App 都是采用原生代码和嵌入页面混合的模式，有很多比较动态的内容都是通过加载 Web 页面的方式来展现。以电商的 App 为例，很多运营活动都是比较动态的，在 App 上线后再配置出来，而且因为不同的活动需要的模板不同，如果通过原生代码来做，代价会比较大，所以很多都是通过内嵌的页面来实现。从用户可见的功能点来说，这方面占了很大的比重，所以非常有必要关注这一部分的性能问题。接下来结合实际的案例，分别从 iOS 和 Android 的角度来做一些这方面测试的探讨。

3.2.3.1 Android webView 性能分析

如果是 4.4 以上版本的操作系统，可以直接使用 3.1.2.2 节提到的方法，通过 PC/MAC 上的 Chrome 进行调试，只需要在程序中添加下列代码即可开启调试开关：

```
if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
```

```
            webView.setWebContentsDebuggingEnabled(true);
        }
```

后续的分析方法跟 3.1.2.3 节中提到的一样。

除了这个方法外，我们可以使用 WebView 提供的以下两个 API 进行白盒测试：

- setWebChromeClient——主要可以提供通信相关的通知。
- setWebViewClient——主要可以提供 UI、JS 相关的通知。

我们可以通过修改源代码调用这两个 API 来实现。首先我们需要找到 WebView 的实例。然后在初始化阶段加入我们的测试代码。

下面是一个例子，获取加载页面和资源加载的时间。其中有底纹部分的代码是我们添加的测试代码：

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    WebView webView= (WebView)findViewById(R.id.webView);
    webView.setWebChromeClient(new WebChromeClient(){
        long elapsed=0;
        long startTime=0;
        public void onProgressChanged(WebView view, int progress){
            if(progress==0)
            {
                startTime=Calendar.getInstance().getTimeInMillis();
            }
            if(progress==100)
            {
            String webViewUrl=view.getUrl();
            long endTime=Calendar.getInstance().getTimeInMillis();
            elapsed=endTime-startTime;
             Log.d("myLog", elapsed+"ms "+webViewUrl+" onProgressChanged");
            }
        }
    });
    webView.setWebViewClient(new WebViewClient(){
        long elapsed=0;
        long startTime=0;
        public void onPageStarted(WebView view, String url, Bitmap favicon){
            startTime=Calendar.getInstance().getTimeInMillis();
        }
        public void onPageFinished(WebView view, String url){
            String webViewUrl=view.getUrl();
            long endTime=Calendar.getInstance().getTimeInMillis();
            elapsed=endTime-startTime;
            Log.d("myLog", elapsed+"ms "+webViewUrl+" onPageFinished");
        }
```

```
            public void onLoadResource(WebView view, String url){

Log.d("myLog", (Calendar.getInstance().getTimeInMillis()-startTime)+"ms "+url+"
    onLoadResource");
            }
        });
        webView.loadUrl("http://www.bing.com");
}
```

上述代码在一些关键的节点加入了输出日志的代码。以 onLoadResource 为例，我们可以在其中获得页面资源加载完成点。

通过上述代码我们可以用 Logcat 查看日志，如图 3-66 所示。

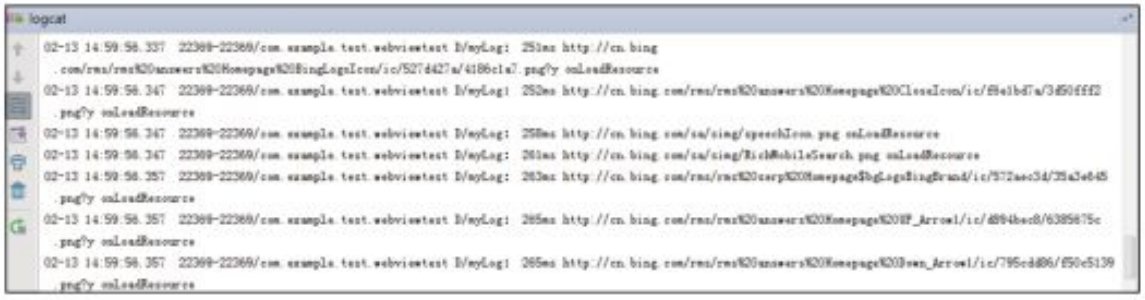


图 3-66 查看 Logcat 日志

我们也可以直接在界面上展示结果。下面图 3-67 是一个例子，在手机上展现用 WebView 访问的网页的加载时间：

```
public  class MyWebViewClient extends WebViewClient{
        Activity mActivity;
        long elapsed=0;
        long startTime=0;
        public void onPageStarted(WebView view, String url, Bitmap favicon){
            startTime=Calendar.getInstance().getTimeInMillis();
        }
        public void onPageFinished(WebView view, String url){
            String webViewUrl=view.getUrl();
            long endTime=Calendar.getInstance().getTimeInMillis();
            elapsed=endTime-startTime;

            Toast.makeText(mActivity, elapsed+"ms",  Toast.LENGTH_LONG).show();
}
```

在 Activity 的初始化 WebView 代码中增加：

```
webView.setWebViewClient(new MyWebViewClient(this));
```

效果如图 3-67 所示。

图 3-67 直接在 App 界面上展示网络访问时间

除了加载相关的信息外，我们还能通过这两个 API 获取更多的信息。例如 JavaScript 的异常信息，页面加载失败错误详情，等等。详情可以参考官方的 API 文档。

值得一提的是，由于一些情况下开发人员自己也会调用这两个 API，我们采用调用 webView 的 API 进行测试的时候需要特别注意，必须在不干扰原有代码逻辑的前提下加上我们的测试代码。在开发没有调用这两个 API 的情况下，我们可以考虑使用 AOP 等方式进行全局的代码添加以提高测试效率。

3.2.3.2 iOS webView 性能分析

iOS 的 webView 内部引擎和 Safari 不同，由于 webView 不支持 JavaScript 加速，所以我们会看到 webView 的性能比起 Safari 性能会差很多。好的 webView 设计能够让用户有非常好的操作体验，那么如何来测试，获取 webView 的性能指标呢?

1. webView 类的加载时间

我们可以在 webView 的 loadRequest 方法执行后埋入初始时间代码，如图 3-68 所示。

```
[_webView loadRequest:[NSURLRequest requestWithURL:[NSURL URLWithString:self.entryURL]]];
NSString* start=[NSString stringWithFormat:@"%.2f",[[NSDate date] timeIntervalSince1970]];
NSLog(@"开始加载页: %@",start);
```

图 3-68 在 webview 加载时，插入开始时间

接着在 webView 的 - (void)webViewDidFinishLoad:(UIWebView *)webView 委托方法中，插入结束时间代码，如图 3-69 所示。

```
- (void)webViewDidFinishLoad:(UIWebView *)webView
{
    NSString* end=[NSString stringWithFormat:@"%.2f",[[NSDate date] timeIntervalSince1970]];
    NSLog(@"结束加载: %@",end);
```

图 3-69 在 webView 完成加载时，插入结束时间

通过计算时间差，我们可以得到 webView 的加载时间。不过，这个加载时间只是单单 webView 的初始化过程的消耗时间。从用户角度来说，刚刚看到了这个 webView 的内容而已，毕竟这个 webView 如果内容很多很长，滑动下拉的过程中，资源请求，重新计算 DOM，重新渲染等性能数据就无法得知了。初始化过程太长会导致用户看到整个 Web 页是白屏的，影响用户体验。

2. webView 的请求资源性能分析

刚才提到 webView 的资源请求性能数据，我们知道，现在很多页面都实现了异步调用。比如显示的图片，只有用户下拉到了它的显示位置，App 才会去请求该资源。这样做的好处很明显：1）减少了 Web 页的初始化时间。 2）节约流量。那么怎么去获取这些资源请求的性能数据呢？我们可以通过 Fiddler 抓包去查看，如图 3-70 所示。

```
Request Count:    1
Bytes Sent:       347        (headers:347; body:0)
Bytes Received:   9,092              (headers:364; body:8,728)

ACTUAL PERFORMANCE
--------------
ClientConnected:      14:54:18.203
ClientBeginRequest:   14:54:18.208
GotRequestHeaders:    14:54:18.208
ClientDoneRequest:    14:54:18.208
Determine Gateway:    0ms
DNS Lookup:           2ms
TCP/IP Connect:       31ms
HTTPS Handshake:      0ms
ServerConnected:      14:54:18.242
FiddlerBeginRequest:  14:54:18.242
ServerGotRequest:     14:54:18.242
ServerBeginResponse:  14:54:18.273
GotResponseHeaders:   14:54:18.273
ServerDoneResponse:   14:54:18.273
ClientBeginResponse:  14:54:18.273
ClientDoneResponse:   14:54:18.273

       Overall Elapsed:     00:00:00.0650000
```

图 3-70　Fiddler 抓包查看接口的响应时间信息

在统计标签下，查看请求从开始请求到响应的消耗时间。当然，这个性能数据只是单单从网络层面的性能分析，如果这个性能数据不理想，那么极有可能是 Server 端的响应问题，或用户当前网络带宽偏慢，或网络中间哪个节点出了问题。体现在 webView 中就是图片区域白屏，图片长时间无法显示，影响用户体验。

3. webView 的渲染速度

我们可能还需要知道 webView 在计算布局、渲染过程的消耗时间。做 Web 开发的同学可能会利用浏览器开发者工具，比如 Safari 的 Web Inspector 的 Timeline 来查看页面加载性能。同样，我们可以利用 Web Inspector 来查看页面元素和布局。该工具支持查看 iOS 设备的 webView 元素、渲染时间等。其他浏览器通常也有类似的开发者工具可以使用。

打开 Safari 设置，在 Advanced 标签里，勾选“Show Develop menu in menu bar”选项，如图 3-71 所示。

图 3-71 Mac 的 Safari 的设置

然后在菜单中找到 Develop（开发者），可以看到设备以及当前的 Web 页，图 3-72 所示为针对京东 App 的 Web 页面进行分析。

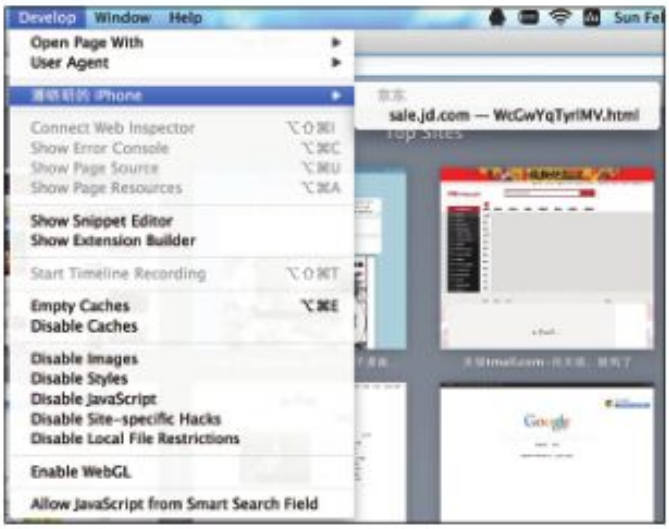


图 3-72 “开发者”菜单中的 Inspector 功能

点进去后，在 Web 中的操作都会被记录，包括网络请求、布局与渲染、JavaScript 执行，如图 3-73 所示。

我们可以进入布局和渲染这个部分，查看页面的渲染速度，如图 3-74 所示。

Web 页在滑动的过程中，布局会失效，也就是会出现大量式样重新计算的原因，然后再渲染。

提升 webView 的性能建议：

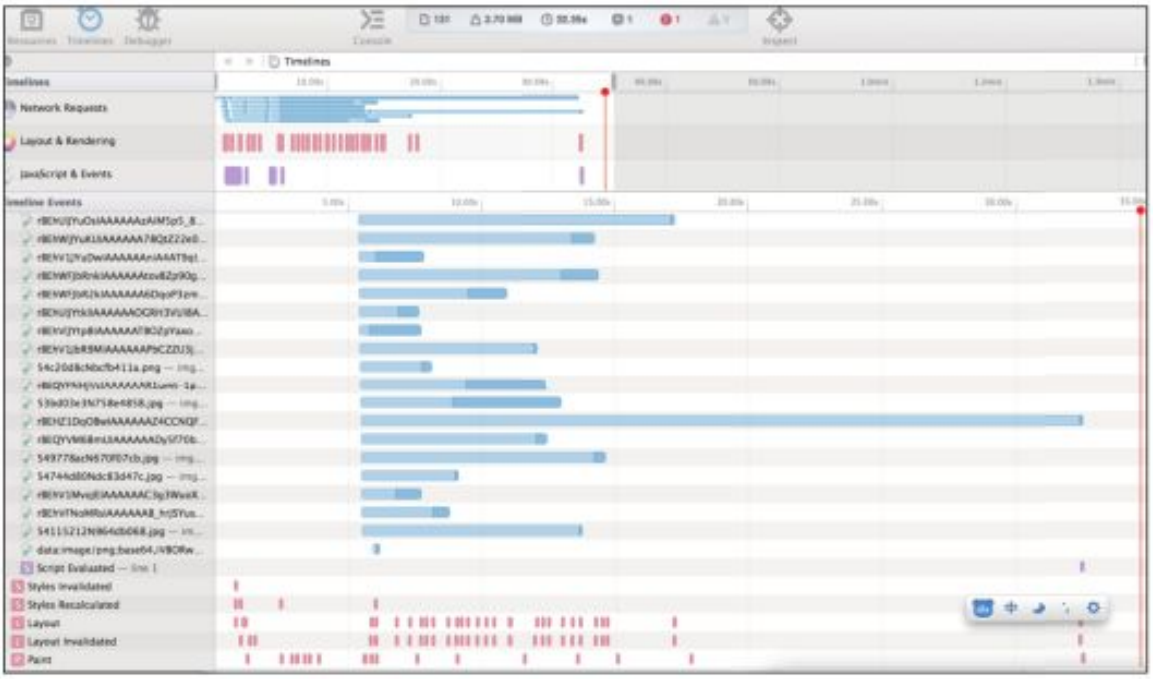


图 3-73 请求、资源渲染和 js 执行的 timing

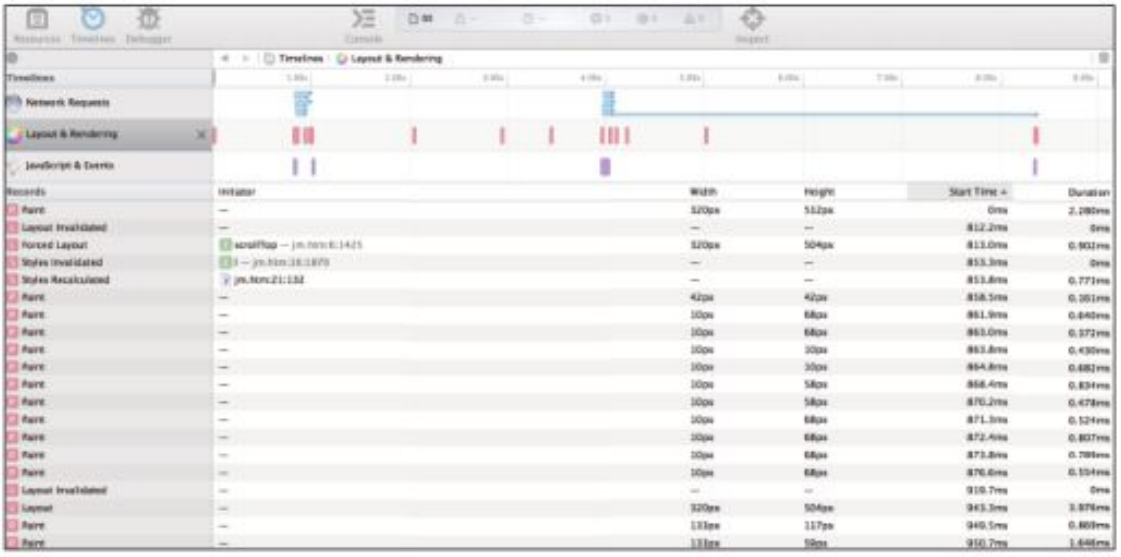


图 3-74 页面渲染的过程和时间

- 在符合需求的情况下，缓存一切能够缓存的东西，包括 html、js、图片等等。这部分的请求耗时与跨越度是最长的。缓存后，用户下次进入 Web 页，由于资源从内存中加载，加载速度是非常快的。

- Web 页面尽量做得小，页面元素尽量少。我们之前提过，如果 Web 页太长，用户需要一直下拉滑动操作的话，会导致大量的布局失效，然后重新计算式样、画图、渲染，这部分也是 webView 性能下降的原因，直接影响用户体验。
- 减少复杂的 JavaScript 执行。之前提过 webView 的 js 引擎和 Safari 不同，执行速度也会慢很多。
- 最后，要提一下，桌面端的高性能 Web 页的开发的注意点同样适用于移动客户端。

3.3 后台服务性能测试

前面我们介绍了 Web 前端性能和 App 前端性能的一些知识要点和测试方法，接下来我们讨论后台服务性能测试。从整体的性能来看，很大的部分是依赖于后台服务的能力，因为通常很多的功能都依赖于从后台服务接口返回的信息。如果后台服务响应比较慢，即使前端做了非常好的优化，用户感体验还是会不好。

这里不准备详细地从零开始介绍后台服务性能测试的各个方面内容，那样可能需要单独的一本书。这里会重点介绍一些比较容易被忽视的点，可以作为参考。如果想做好性能测试，可能还需要系统地学习很多相关的内容。

在介绍具体的知识点之前，有几个性能测试的概念值得探讨。

1. 性能测试的目标和类型

笼统地说，性能测试的目的是为了量化地评估被测系统的响应时间和容量等维度的指标。但是，实际执行过程中，根据侧重点的不同又可以分为几种类型的性能测试：

- 负载（Load）测试——测试系统在一个预先定义的负载情况下的性能表现，看数据结果是否在可接受的范围内。比如按照设计要求，某个系统需要支持每天 1000 万人使用，最高同时 10 万人在线。那么在系统上线前可能需要进行实际性能测试的评估，看能否满足设计要求。

关于如何把上面偏商业指标的要求转化成性能测试的参数设定后面章节会做一些讨论。

- 压力（Stress）测试——在上面既定目标的负载测试基础上，我们可能想进一步了解系统的性能情况。这样做有几个方面的价值：
  - 了解系统性能的极限，或者俗称的性能的拐点在哪里，在当前情况下最大可以支持到怎样的访问和交易量？
  - 当超过极限的时候，系统的行为是什么？拒绝服务、崩溃、以及是否会造成数据丢失等问题？

- 帮助我们找到系统的瓶颈。当性能到达上限的时候，进一步提升的阻碍在哪里，可能是架构、组件、编码或者配置的问题。

- 耐久性（Endurance）测试——由于互联网的系统都是7×24小时在线上运作的，所以需要从时间维度上考察性能，看是否具有可持续性。实际中我们会遇到在系统刚上线的时候性能比较好，但是运行了一段时间之后开始变慢，可能的原因有很多，常见的包括：数据量的累积，使得DB和日志等增大影响性能；内存泄漏等问题在累积一段时间后爆发；系统的线程管理、队列处理等设计不合理，在长时间运行，特别是遇到波峰情况后出现异常。

这些方面的问题可以通过耐久性角度的性能测试来发现，比如提前构造大量数据，以及通过贴近上限的负载长时间大数据量测试执行，类似于工业产品测试中快速老化的做法，让这些在实际项目运行中需要长时间累积爆发的问题在一个可接受的比较短的时间内暴露出来。

- 可扩展性（Scalability）测试——这一点对于互联网产品尤其重要，绝大部分产品在早期的时候无法精确预估可能会有多大的用户量，因而系统需要跟随业务量的增长一起成长。在这种情况下，我们需要进行可扩展性测试。在测试执行的过程中，通过调整后端的配置，比如增加新的服务器，或者改变负载均衡的设置等方式，观察测试数据是否有预期的增长。理想的情况下，增加新的服务器之后可以实现线性的增长，但是受限于负载均衡系统的能力，以及数据同步等问题，这一目标并非简单就可实现，所以需要实际测试的验证，找出哪里可能存在单点的性能瓶颈。

- 基准（Benchmark）测试——基准测试的特点在于有一套标准化的测试套件，甚至整个测试环境。进一步可以分为两类，一类是某个组织内部自己定义的一套针对某个产品的标准化的性能测试、测试环境、方法和参数配置等在内部达成一致；另一类是通用的对外公开的基准测试。

由于在性能测试中，环境、方法和参数等维度的细小差别很可能会导致结果的巨大差异，比如前面章节提到的HTTP请求在TCP层连接数并发的设置，所以如果有一套标准化的测试套件，会让性能测试结果更加容易做横行或者纵向的对比，使之成为一个大家比较容易理解的指标。

业界比较知名的两个基准测试的组织是TPC（http://www.tpc.org/）和SPEC（http://www.spec.org/）。他们针对一些硬件和软件系统提出了标准化的测试套件。以TPC为例，其中有一个基准测试套件TPC-E，针对在线交易处理系统（On-Line Transaction Processing，OLTP）。整个套件包含了一个虚拟的公司及对应的交易系统、测试数据和工具。

基于这个标准化的套件，就可以横向来对比不同服务器系统的性能，这里以每秒处理

的交易数（transactions per second，tps）来作为主要指标。图 3-75 所示是 TPC 网站上公布的 TPC-E 在某个时间段内性能前 10 名的服务器系统。

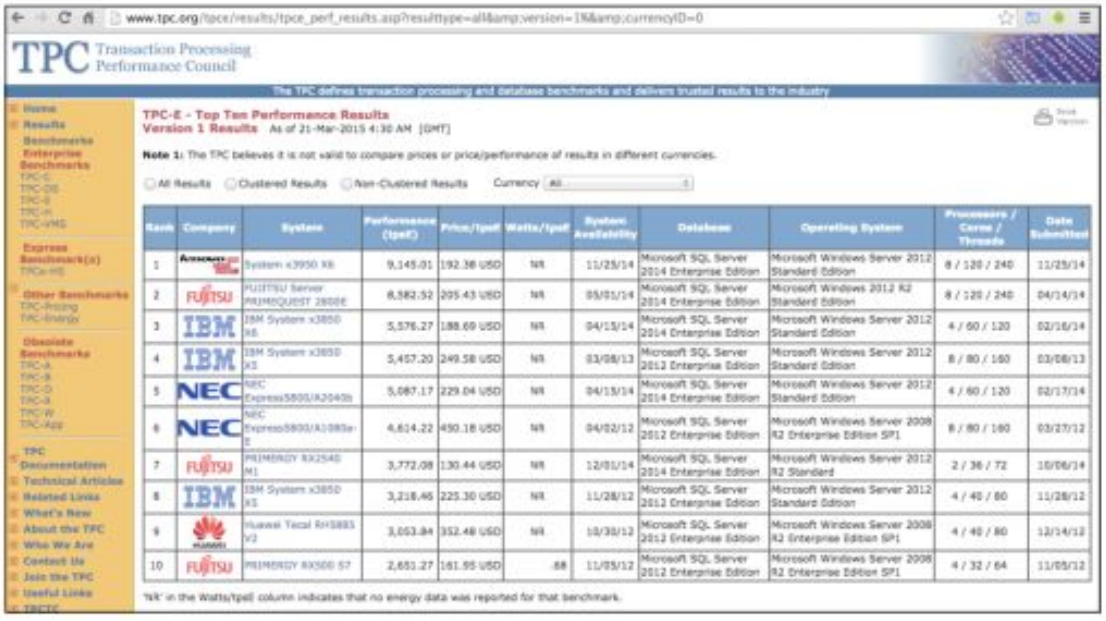


图 3-75　某时间点 TPC-E 前 10 名结果

基准测试的好处是标准化之后便于对比，需要说明的是，不要过于依赖基准测试，即便是内部开发的针对性的基准测试，也只能作为一个参考。因为其代价是将一些场景和参数固化，有些可能无法反映实际项目的情况，特别是对于一个复杂的业务系统。

综合来看，以上不同类型的性能测试目标其实有相关性，从测试技术方法上也不是隔离的。比如压力测试和耐久性测试可以通过在基本的负载测试的基础上提升压力，或者延长测试时间来实现。实际项目中，需要结合项目的需求，来针对性地进行测试。

2. 精确还是模糊

通常来说，性能测试最后都会给出一些明确的数字化的结果，于是可以认为性能测试相比功能测试而言是精确的。

这里以汽车燃料消耗（俗称油耗）的测试结果为例，在中国汽车燃料消耗量网站（http://chinaafc.miit.gov.cn/）上可以查询到各个型号汽车的燃料消耗情况。图 3-76 是一个具体汽车型号的数据。

其中列出了三种情况（市区、市郊、综合）下的每百公里油耗测试值，都是精确的数字。但是稍微有一些这方面常识的人都知道，这个数字其实并不能精确地代表现实中开这个车型 100 公里的实际油耗。原因是实际情况下油耗会受到诸多因素的影响，至少包括车

辆的载重情况、道路的情况（是否拥堵、上坡还是下坡等），以及驾驶者的习惯等方面。

图 3-76 汽车燃料消耗量标识

进一步查询以上标识中提到的国标 GB/T 19233—2008，可以看到测试的方法。如图 3-77 和图 3-78 所示。

图 3-77 关于汽车燃料消耗量计算的国标

轻型汽车燃料消耗量试验方法

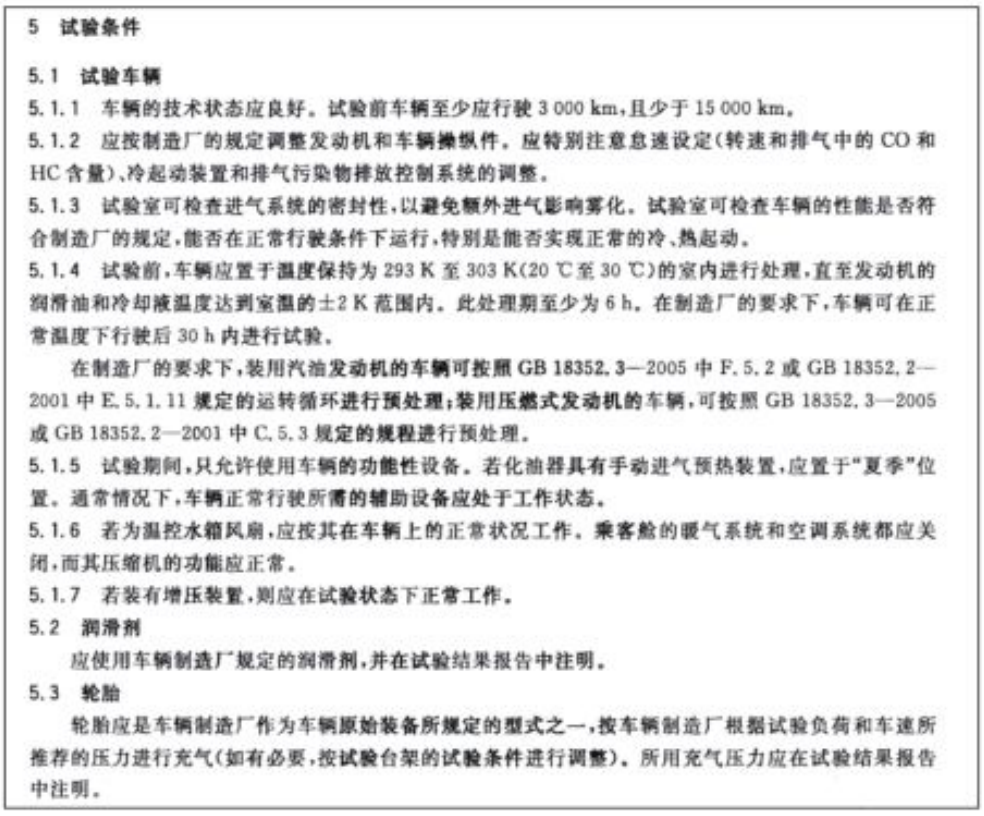
1 范围

1.1 本标准规定了通过测定汽车在模拟市区和市郊工况循环下的二氧化碳(CO_2)、一氧化碳(CO)和碳氢化合物(HC)排放量，并用碳平衡法计算燃料消耗量的试验和计算方法，以及生产一致性的检查和判定方法。

1.2 本标准适用于以点燃式发动机或压燃式发动机为动力，最大设计车速大于或等于 50 km/h 的 M_1 类、N_1 类和最大设计总质量不超过 3 500 kg 的 M_2 类车辆。

1.3 本标准不适用于不能燃用汽油或柴油的车辆。

图 3-78 试验方法说明



5 试验条件

5.1 试验车辆

5.1.1 车辆的技术状态应良好。试验前车辆至少应行驶 3 000 km，且少于 15 000 km。

5.1.2 应按制造厂的规定调整发动机和车辆操纵件。应特别注意怠速设定(转速和排气中的 CO 和 HC 含量)、冷起动装置和排气污染物排放控制系统的调整。

5.1.3 试验室可检查进气系统的密封性，以避免额外进气影响雾化。试验室可检查车辆的性能是否符合制造厂的规定，能否在正常行驶条件下运行，特别是能否实现正常的冷、热起动。

5.1.4 试验前，车辆应置于温度保持为 293 K 至 303 K(20 ℃至 30 ℃)的室内进行处理，直至发动机的润滑油和冷却液温度达到室温的±2 K 范围内。此处理期至少为 6 h。在制造厂的要求下，车辆可在正常温度下行驶后 30 h 内进行试验。

在制造厂的要求下，装用汽油发动机的车辆可按照 GB 18352.3—2005 中 F.5.2 或 GB 18352.2—2001 中 E.5.1.11 规定的运转循环进行预处理；装用压燃式发动机的车辆，可按照 GB 18352.3—2005 或 GB 18352.2—2001 中 C.5.3 规定的规程进行预处理。

5.1.5 试验期间，只允许使用车辆的功能性设备。若化油器具有手动进气预热装置，应置于"夏季"位置。通常情况下，车辆正常行驶所需的辅助设备应处于工作状态。

5.1.6 若为温控水箱风扇，应按其在车辆上的正常状况工作。乘客舱的暖气系统和空调系统都应关闭，而其压缩机的功能应正常。

5.1.7 若装有增压装置，则应在试验状态下正常工作。

5.2 润滑剂

应使用车辆制造厂规定的润滑剂，并在试验结果报告中注明。

5.3 轮胎

轮胎应是车辆制造厂作为车辆原始装备所规定的型式之一，按车辆制造厂根据试验负荷和车速所推荐的压力进行充气(如有必要，按试验台架的试验条件进行调整)。所用充气压力应在试验结果报告中注明。

图 3-79 燃料消耗量测试国标中关于试验条件说明

图 3-79 所示的测试方法非常类似于前面提到的基准测试，将一套完整的方案用于所有被测对象，得出可以对比的结果。基于上面试验条件的介绍，可以看出在测试燃料消耗的过程中，对于被测车辆的条件做了多项设定。

从上面的例子可以看出，所谓精确的测试结果，是在做了大量前提条件的设定下得出

的，并不能和真实场景中的个案一一对应。但是我们不能因为这个测试结果无法直接映射到真实环境下而否定这样的测试价值。因为通过这个结果可以知道实际结果所在的大致范围，另一方面，也可以提供横向的对比参考。理论上，在标识上同一工况下油耗更高的车型在现实中同样使用情况下油耗也会更高，可以给消费者提供参考。

回到软件性能测试，也是一样的道理，其数字结果是在包括被测系统、测试工具和实施方案等多个前提条件的设定下得出的。所以，对于结果的理解不能走向两个极端，完全的刻板认定或者否定测试结果的价值，而应把基于这些前提条件的测试结果作为实际系统的参考。

3. 宏观还是微观

关于性能测试还有一个比较常见的观念是认为性能测试是看大的方面，从整体系统的架构出发，而不用关心功能的细节。这个听起来有道理，特别是针对整体而非某个功能点的性能测试。但是实际上，有过一定的实际项目性能测试经验的人可能不这样认为。原因在于看似宏观的性能测试非常依赖于很多微观的细节，比如以下方面都可能会严重影响性能测试的结果：

- 系统负载均衡的策略设置。
- 系统的核心进程数 / 线程数设置。
- 系统的某个核心算法配置。
- 是否启用缓存。
- 日志级别，是否打开 debug。
- 测试工具的配置，比如请求间隔设置、是否打开 keep-alive 等。

以上的例子还可以举出很多，这些都是微观的设置，涉及某个配置文件中的一个值，或者一个 True or False。一个小的改变可能让性能测试的结果差别很大。知名的性能测试专家 Scott Barber 曾在一篇关于性能测试的文章“Macro to Micro And Back Again”中给出一段很有启发的说明：

> “Macro- and Micro-tests, macro strategies and micro-plans, macro-level application usage and micro-level usage implementation details, macro-level result summaries for executives and micro-level test results for developers... it sounds like a day in the life of a performance tester to me.”

当然，并不是所有的细节都需要关注，值得关注的是关键性细节（Critical Details），如何识别这样的细节需要对于被测系统、测试环境和测试工具有深入的了解，也可以咨询相应的人员。

接下来我们具体讨论整个性能测试环节中比较重要的一些知识点，并结合实例来说明。

3.3.1 压力场景的建模

在执行功能测试之前，我们会进行测试设计，思考要测试哪些方面，如何开展，具体的产出是测试设计文档或者用例。在性能测试的时候，其实一样，也需要考虑如何来开展，其中最基本的就是如何构造性能测试的压力场景，包括但不限于以下：

- 需要在性能测试中模拟哪些类型的请求。
- 需要多大的压力，对应到并发量。
- 需要构造怎样的测试数据。
- 对测试工具有什么要求。

这方面考虑不周也是刚开始接触性能测试不久的人容易出现的问题，很多人往往找到一个可以给被测系统发起请求的工具就急于开始测试的执行，很多参数都是比较随意的指定。这样可以给出一些测试的结果，但是很容易给出误导性的结果，也经不起评审和验证。

下面我们从几个方面来讨论如何尽可能在性能测试中模拟出真实用户的行为来产生并发的测试流量。

3.3.1.1 获取和模拟单个用户的行为

性能测试本质上和很多测试一样，也是一种模拟。我们通过工具和脚本模拟出真实用户的请求，通过并发的方式来放大流量测试后台服务的性能，并记录测试结果数据。所以，如何获取和通过工具模拟出单个用户的行为是一个必须首先完成的工作。

1. 获取用户操作对应的接口请求

下面我们结合一个实际的例子来说明。使用的过程是用户打开一个电商的 App，用关键词“手机”搜索商品，忽略前面打开 App 首页的过程。在使用过程中我们将手机 Wifi 的代理指向 Mac 机器上的代理软件 Charles，抓包分析上述操作对应的请求。

从图 3-80 可以看到，排除几个 GET 请求对应的图片下载，可以看到一共有三个搜索相关的 HTTP 请求。从 response 内容上看，上面选中的 ID 为 tip 的请求是用于拉取搜索时的关键词提示。如图 3-81 所示，在用户输入“手机”两个词，还未点击“搜索”按钮的时候，通过上面 tip 接口拉取的推荐词已经展示出来，这时用户可以选择继续搜索“手机”，或者提示里面的“手机壳”等关键词。

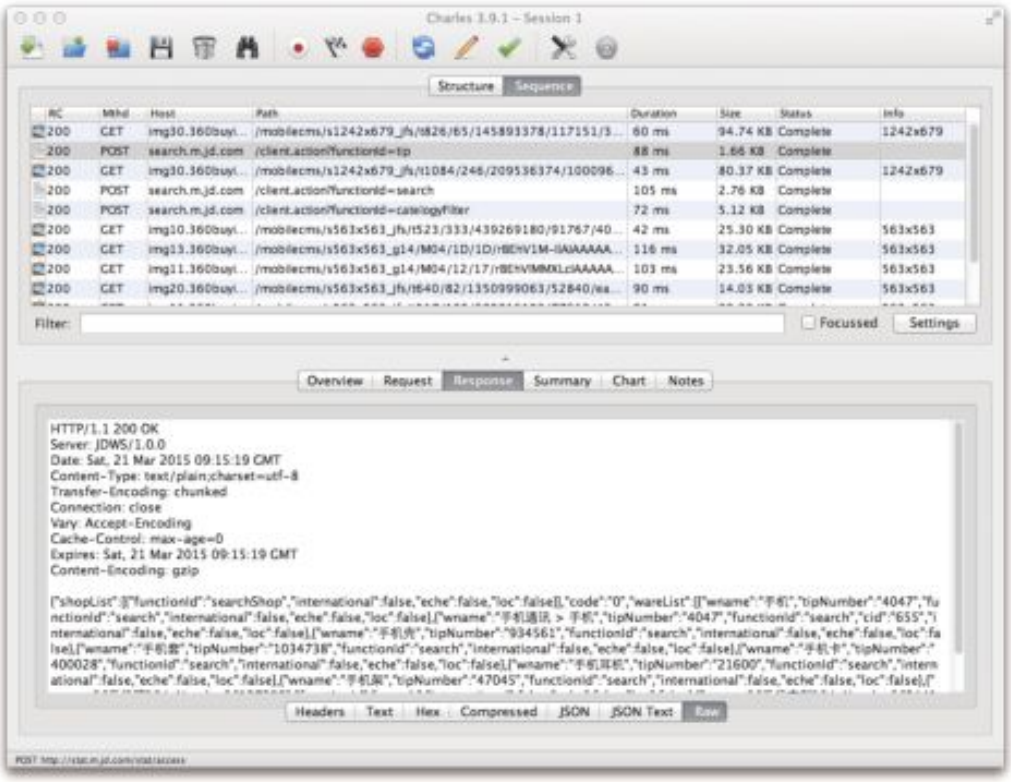


图 3-80 通过 App 搜索商品时的请求列表

图 3-81 搜索关键词和提示

接下来，使用前面输入的“手机”关键词，点击“搜索”按钮发起请求。

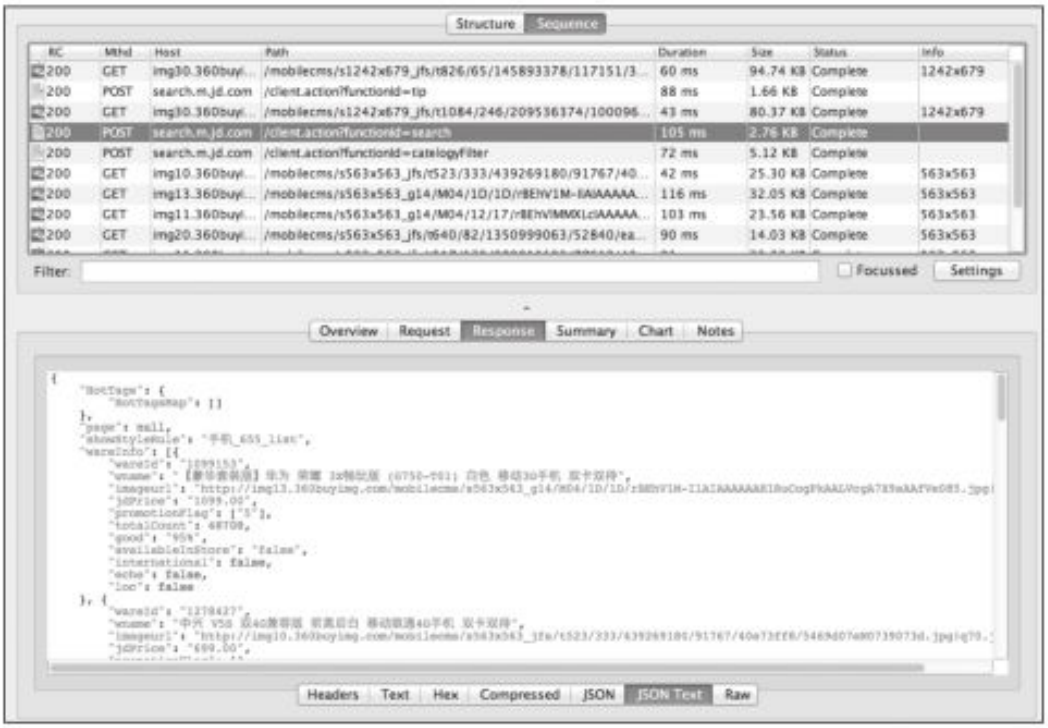


图 3-82　搜索接口对应的响应结果

图 3-82 高亮显示的 search 接口，根据用户输入的“手机关键词”返回了搜索结果，采用 JSON 数据格式。对应到手机 App 上的显示结果如图 3-83 所示。

图 3-83　手机 App 上的搜索结果展示

如果从上面使用过程来看，前面两个接口分别返回了搜索提示词和详细搜索结果，看起来已经完成了相关的请求。但是实际中，从抓包结果看到，即使在用户没有其他操作的情况下，App 在搜索的同时也自动发起了另一个请求，如图 3-84 高亮显示的 catelogyFilter。

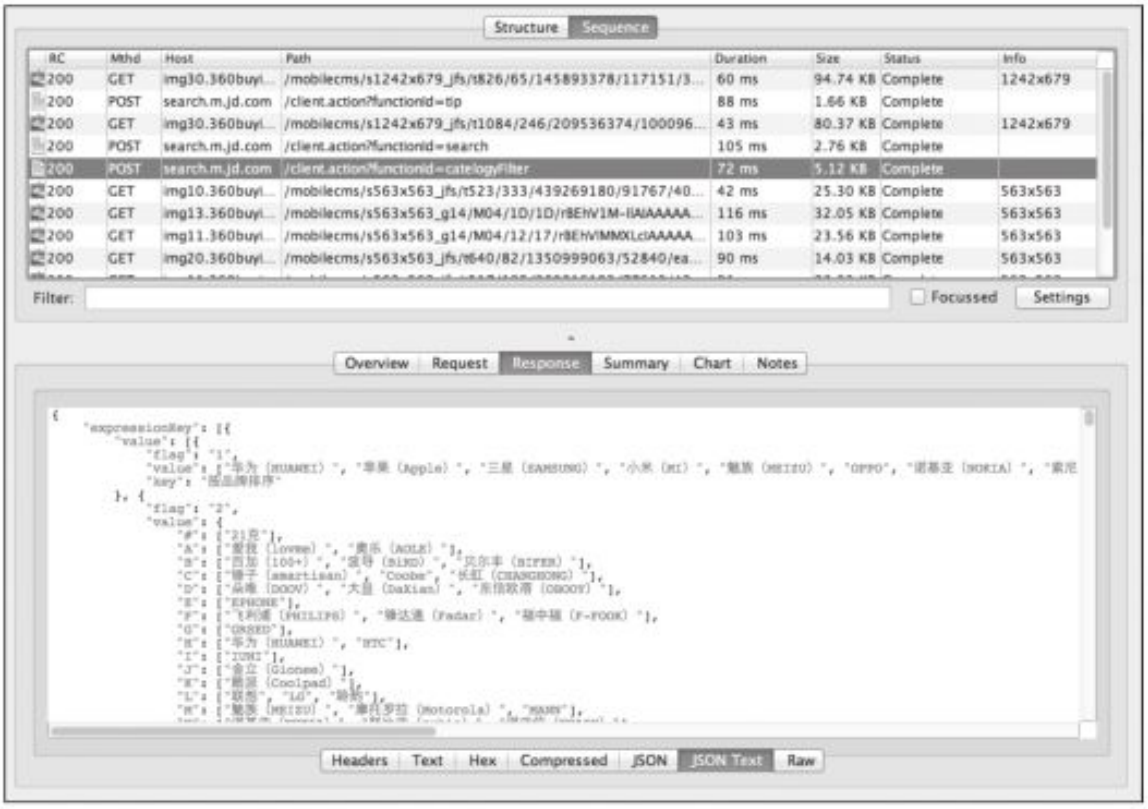


图 3-84 搜索筛选接口返回的内容

从内容来看，这个接口返回的是搜索筛选的选项。对应上面搜索结果图上右上角的“筛选”按钮内的内容，点击后可以看到图 3-85 所示的结果。第一个选项是品牌，选择按自己排序后对应的就是图 3-84 所示的返回结果。

通过以上真实用户行为的分析，我们可以看到，如果将用户在 App 上的搜索操作理解为只是发起一个 search 请求，就会遗漏 tip 和 catelogyFilter 两个请求。进而性能测试中的请求量也会有非常大的偏差，这两个接口的性能问题也无法暴露出来。

从这个例子我们也可以体会到，在单个真实用户行为分析上多花一些时间是值得的，这些也是前面提到的关键细节。这些如果不在性能测试脚本构造之前梳理清楚，很容易差之毫厘谬以千里。以上只是一个示例，实际性能测试中，每一个需要被覆盖的用户使用场景都需要进行类似的分析。

图 3-85　App 上的搜索筛选条件

2. 模拟用户请求

通过上面的分析我们获得了单个用户的行为，接下来我们需要在性能测试工具中模拟出这样的用户请求。基于具体的协议类型和工具提供的用法，常见的有两种做法，一是在工具中配置请求，二是通过代码的方式。下面分别来看看具体的例子。

JMeter 主要是第一种方式，对于 HTTP 请求可以直接配置各种参数，前面在接口自动化的时候也提到过这种方式。

除了直接在里面配置请求，JMeter 还提供了录制的功能，如图 3-86 所示，可以加入一个 HTTP 代理服务器组件，并将“目标控制器”设置为之前创建的线程组，点击下方的启动按钮就会启动代理服务器。这时候可以将手机的 Wifi 网络代理指向运行 JMeter 这台机器的 8080 端口，然后启动 App 操作，相关的 HTTP 请求就会被录制下来，创建到线程组里面。

录制的好处在于简化了创建请求的过程，像请求相关的参数在录制的时候 JMeter 会自动填好，如图 3-87 所示。

添加一个查看结果树的组件，就可以执行上面录制的请求，来验证是否可用。如图 3-88 所示，可以看到前面提到的搜索 search 接口执行后返回的 JSON 响应数据。

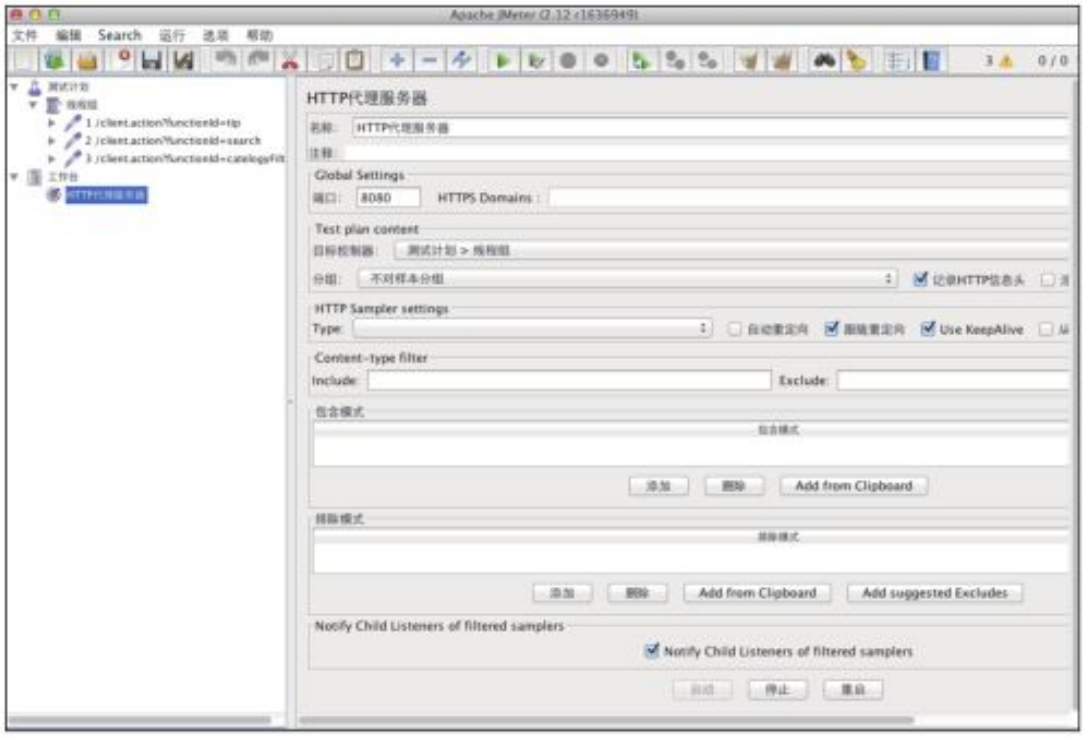


图 3-86 JMeter 请求录制代理

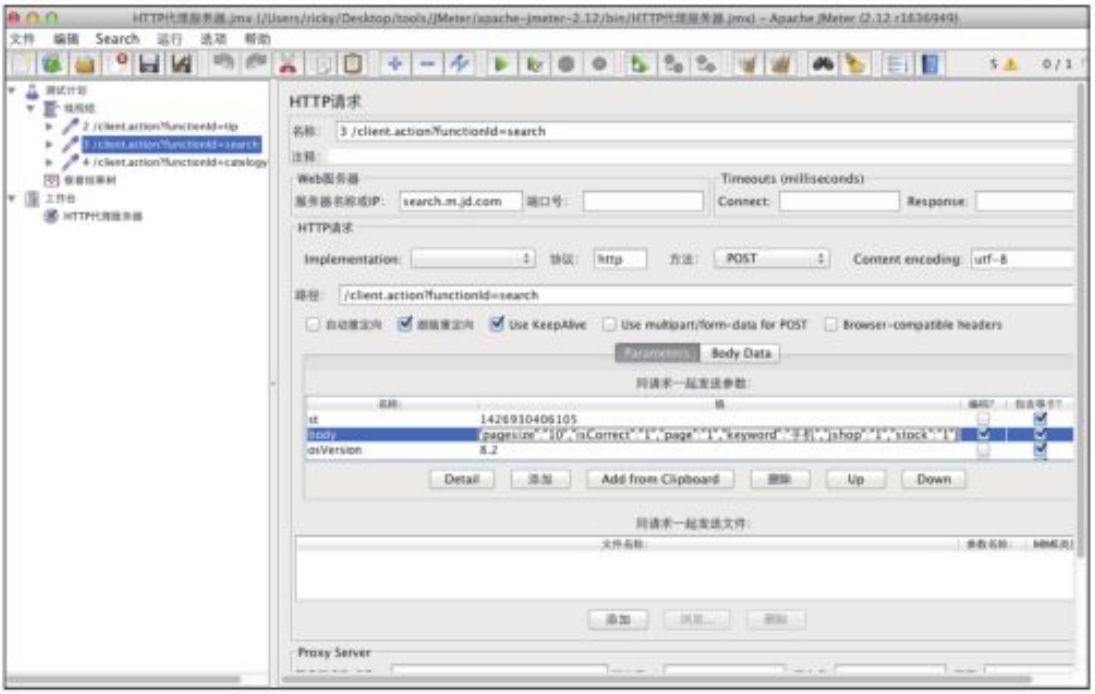


图 3-87 通过录制获得的接口请求

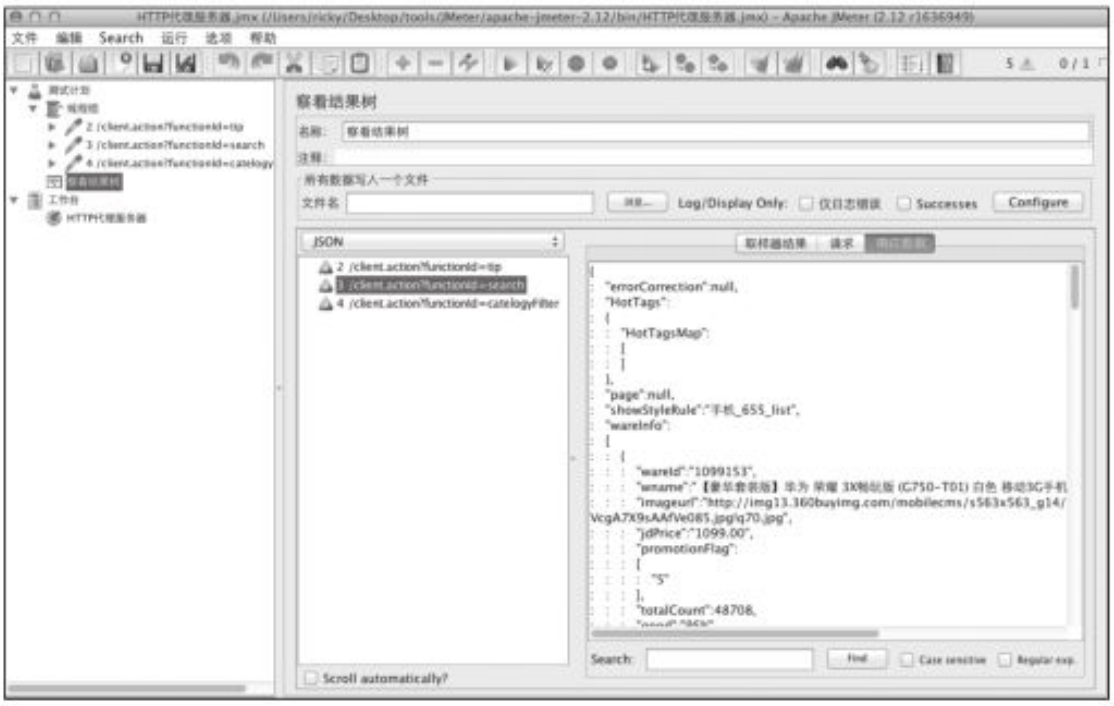


图 3-88 录制后请求的执行结果

以上是一个简单的示例，实际中的情况可能更加复杂，录制下来的请求不一定直接可用，因为部分请求数据可能是一次有效的 ID，或者有时效性的要求。那就需要对数据做参数化的处理。关于这部分内容不做详细介绍，请参考对应工具的文档。

第二种方式是通过脚本的方式来描述请求和对应的数据，以及控制逻辑。图 3-89 所示的 LoadRunner 是这方面很好的例子，它可以支持多种语言来编写虚拟用户的脚本。

以上脚本中包含了访问 HTML 页面，加入 Think Time（下面会讲解），以及提交表单等操作。

以上脚本可以通过 LoadRunner 自带录制功能获得，原理和前面提到的 JMeter 类似，都是代理的方式，也可以自己手工编写脚本。脚本整理完之后可以在 LoadRunner 中试运行一次，看结果是否正确。图 3-90 所示是上面脚本的执行结果日志。

3. Think Time

通过上面的步骤，我们完成了真实用户的请求分析，并通过测试工具模拟了对应的请求。这里有一个概念需要特别指出，那就是 Think Time。

设想这样的一个场景，一个真实的用户在某个电商网站上购物，一个简化的流程可能如下：

图 3-89 LoadRunner 中的虚拟用户脚本

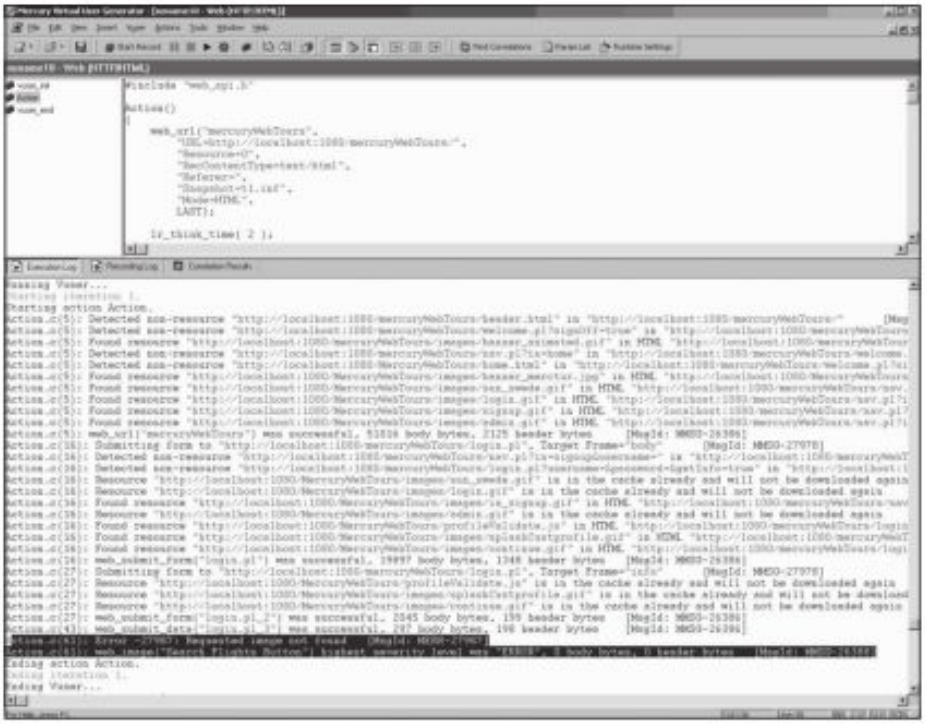


图 3-90 LoadRunner 中虚拟用户脚本的试运行

1）进入首页

2）搜索一个商品

3）查看商品详情

4）加入购物车

5）提交订单

6）完成支付

基于上面的讨论我们可以构造一系列 JMeter 请求，放在一个线程组里面。

如果我们要测试针对这样的用户，我们的系统可以支持多少人同时来购物。假设我们已经考虑了用户登录账号和购买商品的参数化问题，是否可以直接将线程组的数值设置到一个较大的数值，然后并发执行呢？

这样可以执行起来，但是有一个很大的问题。在于真实用户和脚本的不同。脚本如果是基于前面方法录制的，两个请求的执行时间之前是没有任何其他停顿的，其间隔只是依赖于上一个服务的响应时间和测试机发起请求所需的时间。但是显然真实的用户不是机器，他们在做上面每一个步骤的时候都有一个思考的时间，这也是 Think Time 这个词的意义来源。当然，这个思考时间也是泛指，包括了用户操作的时间，比如进入首页后，用户需要在搜索框中输入想购买商品的关键词，打开输入法并输入相关的词可能也需要少则几秒钟的时间，搜索结果出来之后用户需要浏览和选择，找到感兴趣的商品并点击查看详情，后面的步骤也是类似。

试想一下，对比请求连续执行和每个步骤间加入模拟真实用户的 Think Time 之后，对于同一个系统，能支持的同时在线购物人数必定会有绝大的差异，而有 Think Time 的做法显然更接近真实情况。

以下的示例是基于前面的搜索请求。我自己作为一个真实的用户在 App 的搜索框输入“手机”关键词后看了一下推荐词，然后点击了搜索按钮。图 3-91 所示是这个操作过程中抓包看到的请求和响应时间。可以看到，这个请求的发起时间是 17:39:50。

图 3-92 所示是实际的搜索请求的发起时间，是 17:39:52，其中间隔了 2 秒。这个可以视为某一个真实用户在这个步骤的 Think Time。

接下来我们看在 JMeter 中如何模拟这个时间。最简单的办法是在第二个请求中加入一个固定定时器，如图 3-93 所示，这样第一个请求执行完后会等待 2 秒再发起下一个请求。

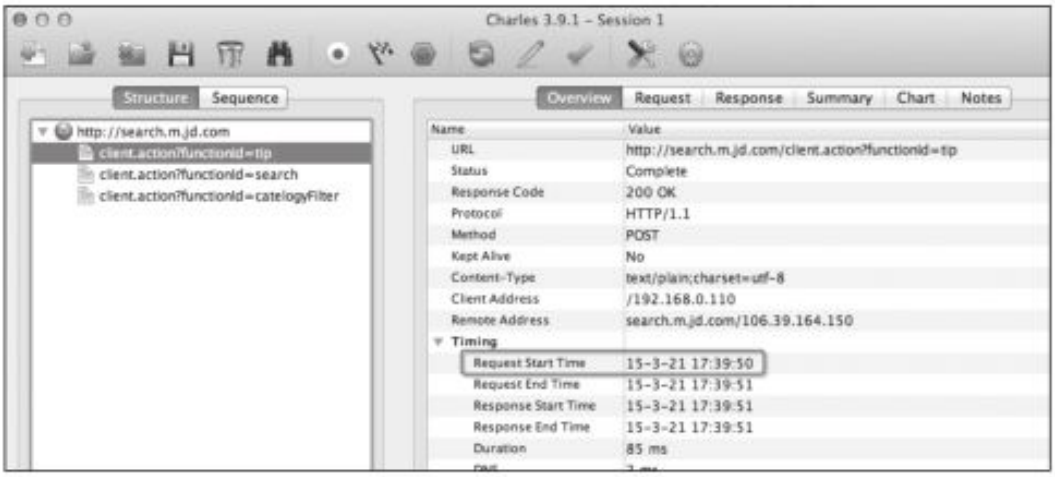


图 3-91 tip 接口的请求时间

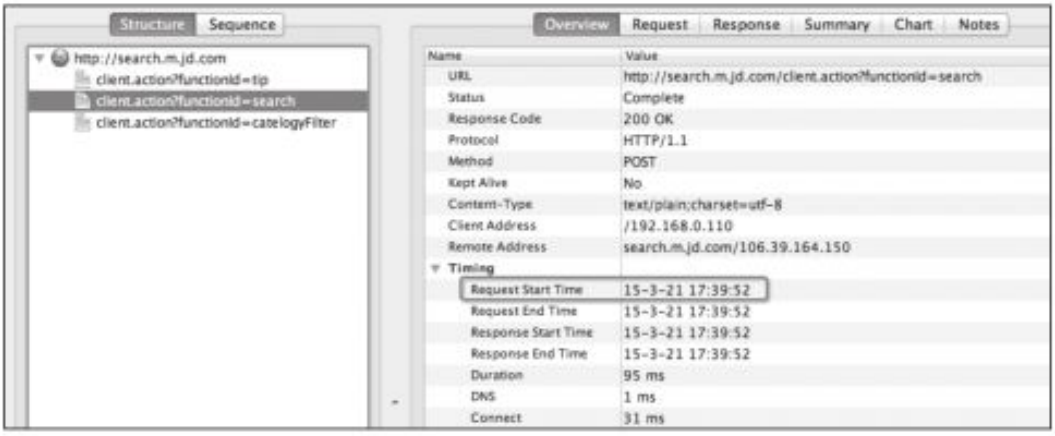


图 3-92 search 接口的请求时间

图 3-93 JMeter 中的固定定时器

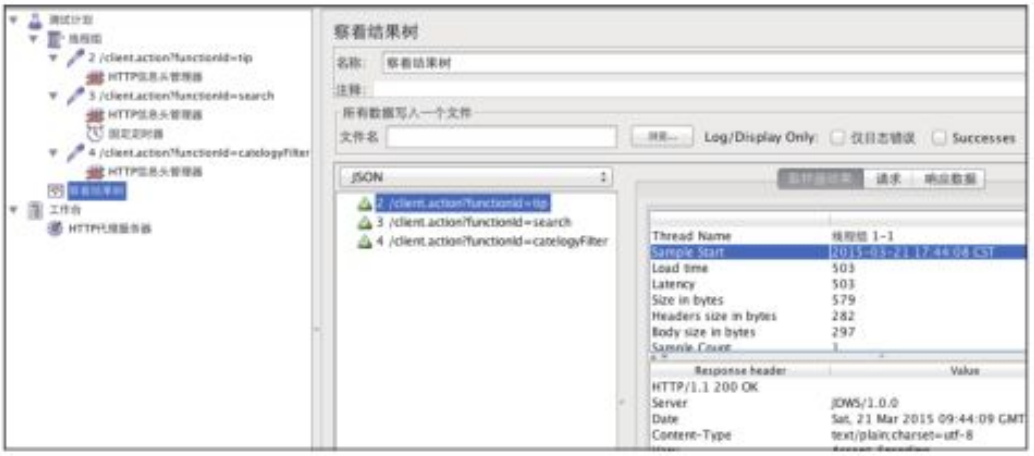


图 3-94 tip 接口的执行结果

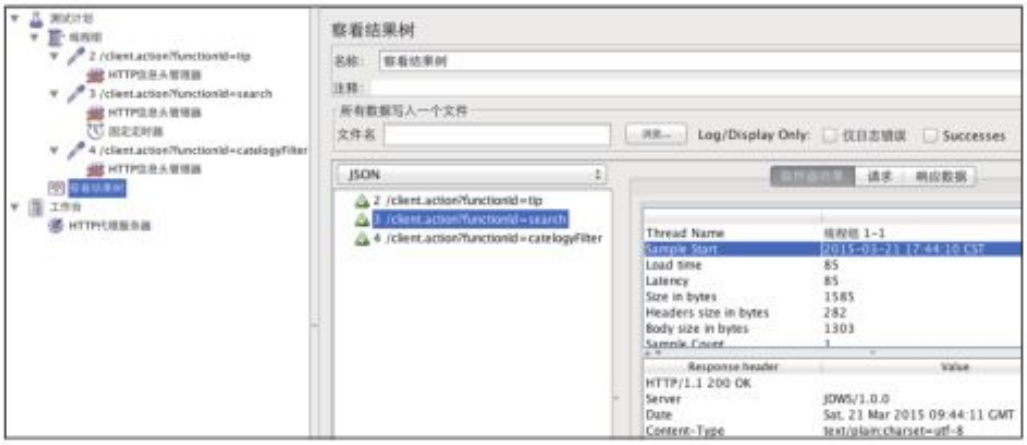


图 3-95 search 接口的执行结果

对比图 3-94 和图 3-95 中所示的两个请求的执行时间，可以看出之间的间隔也是 2 秒。这样我们就在测试中模拟出了一个真实用户的行为。

以上引入了一个固定定时器来模拟 Think Time，实际中每个用户的 Think Time 可能是不同的，所以也可以用随机的定时器，设置一个时间范围，在每次发起请求时随机选取。

从以上例子可以看出，Think Time 这个概念比较容易理解，但是实际中容易被忽视。是否有 Think Time 影响到请求模拟的真实性，直接影响到性能测试的压力情况，进而影响到可以支持多少并发用户的测试结果。

3.3.1.2 构建虚拟用户组

以上介绍了如何获取和模拟单个用户的行为，通过工具放大后其实代表了行为一致的一类用户。但是对于一个真实的被测系统，通常有很多种使用方式，并不是每个用户做的

步骤都一样，如果想看系统整体的性能，那就需要同时模拟多类不同的用户，这里我们称之为虚拟用户组。

以网上购物的系统为例，对于核心流程部分，主要的使用场景包括：

1）搜索商品。

2）访问商品详情。

3）将商品添加到购物车。

4）关注（收藏）商品。

5）查看购物车。

6）选择购物车中商品进入订单填写页面。

7）提交订单。

以上每一个使用场景都需要对应的一个或多个请求来完成。可以采用前面提到的方法逐个构造出每个场景的真实用户对应的请求。

如果要完整地测试整个核心流程的性能，需要把上面提到的环节都包含进去。如图3-96所示，可以在JMeter中创建多个线程组，来表示不同的虚拟用户组，这样也可以针对每个组来控制并发请求量。

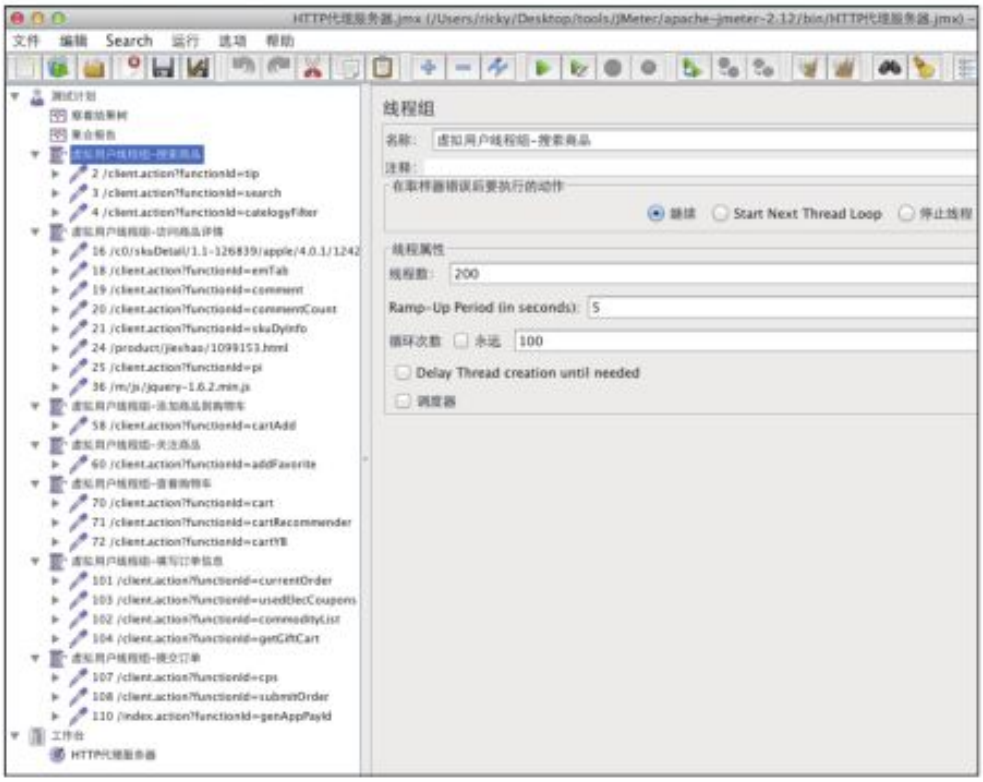


图3-96 在JMeter中模拟不同使用场景的虚拟用户组

这里的做法相当于不同类似用户操作的混合，那么接下来的问题是采用什么样的比例来进行混合。如果能有一些历史数据作为参考，会比较有说服力。这样的历史数据可以来源于一个旧系统的报告或日志，也可以参考业界的相关数据。实在没有参考的就只能通过相关人一起讨论来做合理的假设。

3.3.1.3 测试流量的设定

在构造好了虚拟用户组之后，接下来需要考虑测试流量的设定，除了被测产品的配置外，还有几个方面需要考虑。

1. 选择测试数据样本

这个是性能测试中一个非常有挑战性的，而且无法给出一个通用的简单办法。和前面提到的虚拟用户组比例的选取一样，最好能有一些实际系统的参考数据，这样做合理的假设也不至于盲目。

下面讨论的被测系统为邮件安全扫描服务器的例子。为了测试这个系统，我们需要发送大量的邮件，而不同的邮件大小和类型对于性能有着非常明显的影响，除了内容解析本身的耗时不同，也可能因为触发不同的策略而处理时间不同。

图 3-97 所示的数据是基于这样的方式给出的，先是参考一些真实系统中的概要数据，比如邮件大小和类型的比例，以及触发各种安全策略的邮件占比。基于这样的比例数据，我们在性能测试中再去构造对应的邮件样本。实际中，可能需要结合被测系统的特点来看如何更接近真实的构造测试数据。

component	detail	ratio (%)	rule match	avg size (KB)
spam		80	spam	4.6
phishing		1	phishing	5
virus	joke test virus	1	virus	46
keyword filter	body keyword match	1	content filter - body	3
attachment filter	dll attachment	1	content filter - attachment dll	40
scan exception	password protected	1	scan exception	15
WRS	URL check	1	WRS	2
DKIM	DKIM enforcement trigger	1	DKIM enforcement	5
attachment, normal		7		
	zip	1		357
	eml file	1		18
	excel	1		516
	jpg	1		323
	pdf	1		1330
	ppt	1		819
	word	1		501
plain text, normal		6		4-100
Total		100		46

图 3-97 邮件测试样本分布图

2. 并发量

在流量设置中有一个很重要的概念就是并发量，或称为并发度。比如上面提到的JMeter脚本中，如果将某个线程组的线程数设置为100，那是不是对于这个类型的请求，并发量就是100。从宏观的角度，这样理解也是对的，就好比请了100个人，每个人独立地完成一连串的工作，确实是100个在并行。但是对于服务器感受到的压力，可能就不是100了。图3-98的对比表达了对于这个模型理解上的差异。

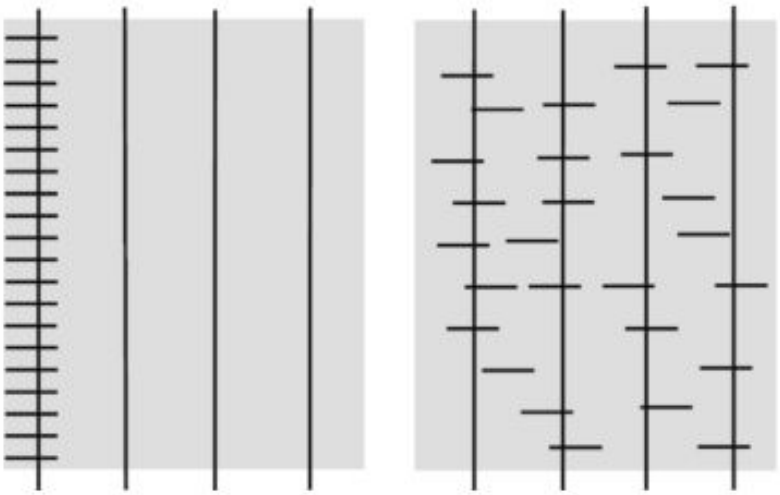


图3-98 两种不同的并发模型对比

这里所谓的100个并行对于服务端而言并不是左边图上严格的全部并行，因为每个虚拟用户执行的节奏是独立。假设这个操作需要3个请求完成，那么很有可能出现这样的情况：某个虚拟用户还在等待第一个请求的响应，但是另一个虚拟用户已经收到了第一个请求的响应并发起了第二个请求。那么对于服务端而言，在某一个时刻，无论是对于请求1还是请求2，并行度都没有到100。这个模型比较类似于图中右边部分所示的模型。理解这个模型对于并行的理解非常重要。

这里的差异在于宏观上的并行还是严格意义上的并行，比如就某个请求的严格意义上的并行或许可以通过TCP连接的保持数来看。

下面我们用一个实际的例子来看，基于前面制作的搜索商品的三个请求构成的一个使用场景，为了简化问题将其他场景的脚本暂时禁用。

在脚本运行的过程中，通过“netstat | grep ESTABLISH | wc -l”命令获得保持连接状态的TCP请求数量。执行JMeter前关闭本机上其他可能联网的软件，减少数据干扰，命令执行的效果如图3-99所示。

```
rickyqiu-nb-2:~ ricky$ netstat | grep ESTABLISH | wc -l
      15
rickyqiu-nb-2:~ ricky$
rickyqiu-nb-2:~ ricky$ netstat | grep ESTABLISH
tcp4       0      0  192.168.0.111.57570    114.80.174.50.http     ESTABLISHED
tcp4       0      0  192.168.0.111.57562    101.226.181.100.http   ESTABLISHED
tcp4       0      0  192.168.0.111.57559    101.226.181.100.http   ESTABLISHED
tcp4       0      0  192.168.0.111.57557    101.226.178.140.http   ESTABLISHED
tcp4       0      0  192.168.0.111.57555    101.226.178.100.http   ESTABLISHED
```

图 3-99　用于检查当前 TCP 连接数的脚本

图 3-100 为了持续观察 TCP 保持连接数的情况，我们在 netstat 命令外面加上了一个循环，每隔 1 秒打印一次。

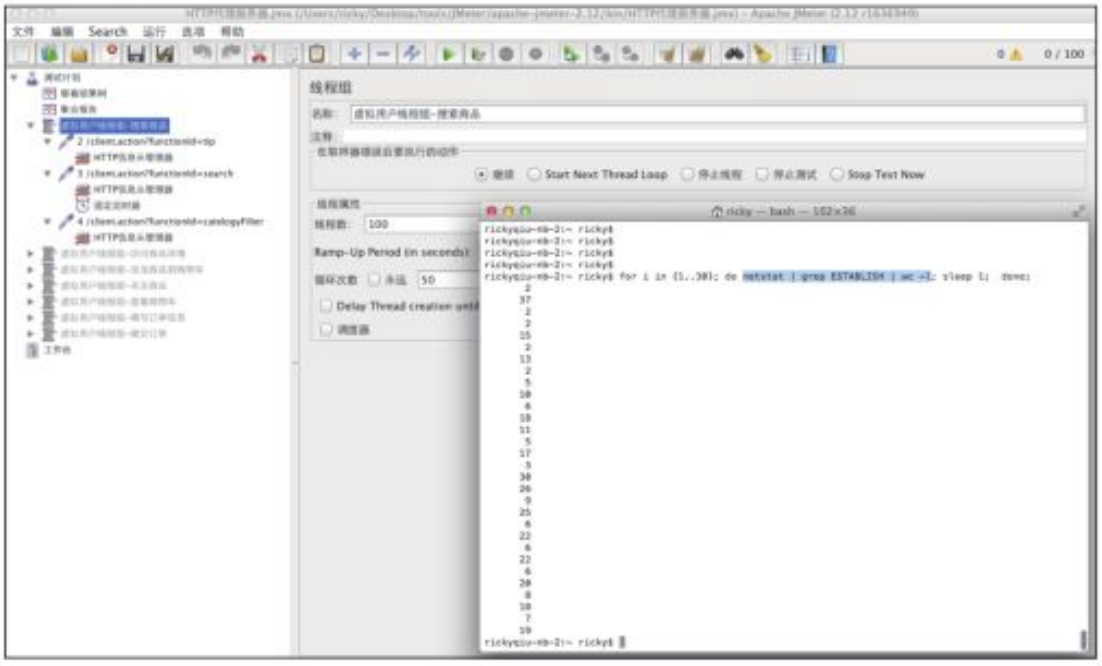


图 3-100　有 2 秒固定延时情况下的实际并发数

从图 3-100 可以看出，最大的协议层面的并发数在 37，更多集中在 30 以下，甚至一些在 10 以下。

接下来我们将用于 search 请求之前 2 秒 Think Time 的定时器暂时禁止，再执行一次上面的过程。

从图 3-101 可以看出，并行度有了很大的提升，集中在 50 上下，最高达到 77，即便如此，和理论上 100 的并发仍有一定的差距。这也验证了之前提到的 Think Time 的有无对于系统压力的不同。

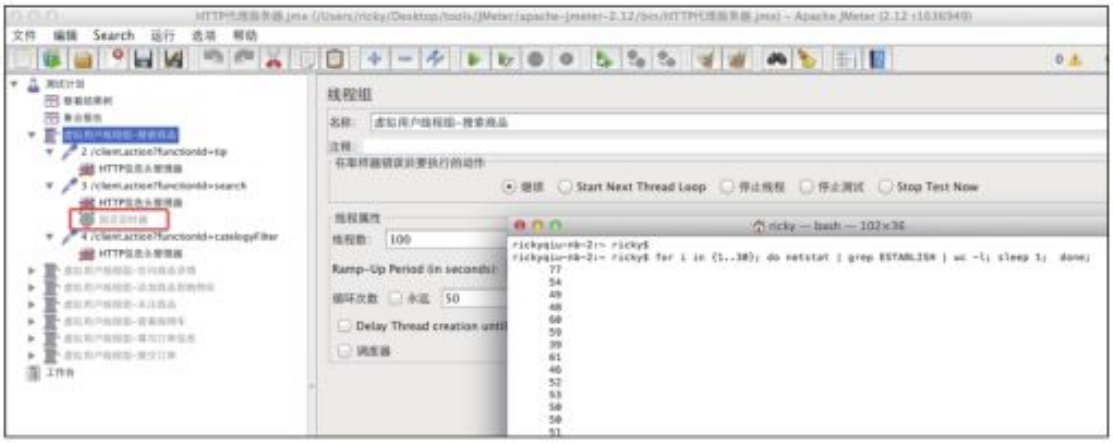


图 3-101 禁用 2 秒固定延时后的实际并发数

3. 实际系统的流量曲线

业务部门或者运营团队给出的系统性能描述通常包含这样的表达："系统需要能支撑每天 1000 万活跃用户的使用，每天完成 100 万笔交易。"从业务团队的角度，这里的说明是合理的，但是对于性能测试人员，这样不具有操作性，需要转化为工具可以配置的压力参数。那么如何映射到每分钟，甚至每秒的请求呢。

为了让我们的性能测试更加的真实，可能参考一些线上系统的流量曲线。以下两张图 3-102 和图 3-103 是某个线上系统连续 2 天的系统的流量曲线。

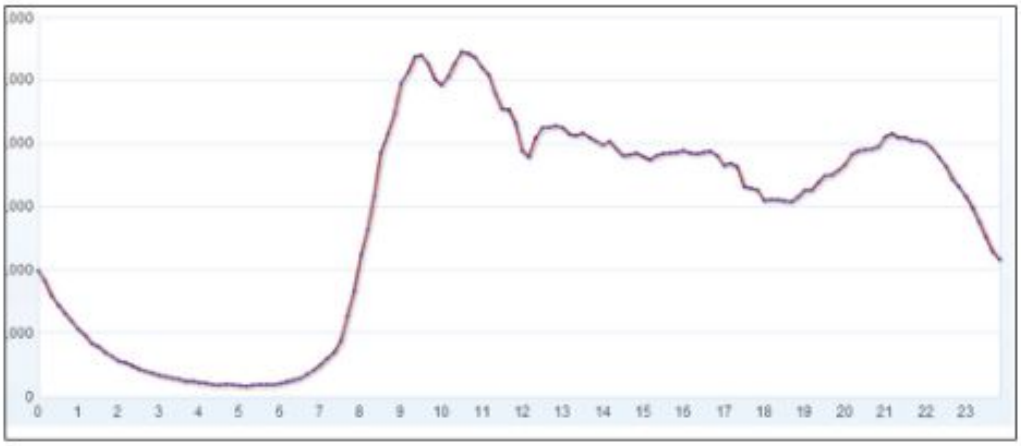


图 3-102 某系统 Day 1 的 24 小时流量曲线

从图可以看出，两天的曲线略有不同，但是整体的走势非常接近。都是早上 8 点多之后有一个高峰，持续到 11 点多，然后一直到晚上保持比较平稳，晚上 12 点后开始逐步下降，凌晨 3 点到 6 点左右处于最低谷。这样的图大概反映了大量真实用户的使用习惯。

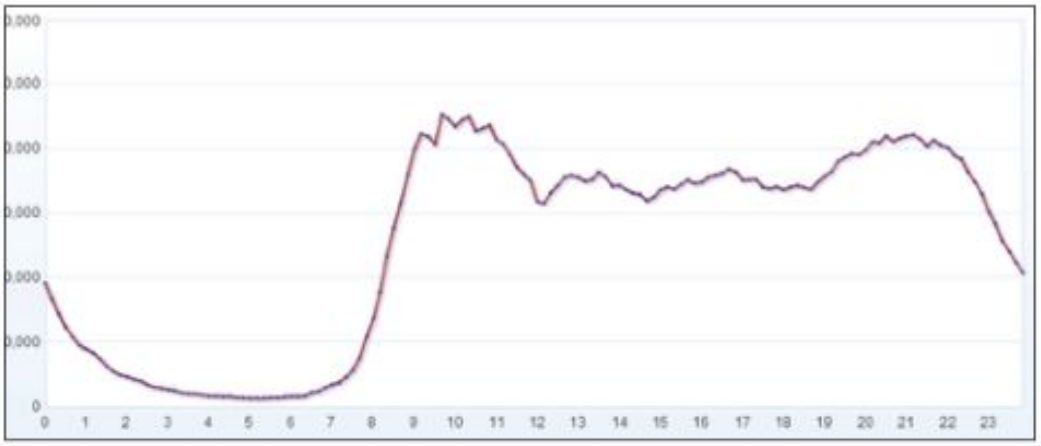


图 3-103 某系统 Day 2 的 24 小时流量曲线

所以对于性能测试而言，我们不能简单地将前面提到的每天的指标平摊到每个小的时间段，而是要考虑峰值的情况下系统是否能承受。上图的粒度还比较粗，大约是每 10 分钟一个采样点，如果要更精确地了解峰值的情况，可能需要更细粒度的曲线图，例如每 5 秒钟左右一个采样点。

3.3.2 测试工具

基于本书的定位，这里不会详细介绍某种测试工具的使用方法，而准备讨论一些共性的特点和需要注意的问题。

1. 测试工具的主要组成部分

性能测试工具各有不同，但是抽象来看其实都有一些共同点。图 3-104 所示是在使用过大量的商业和开源性能测试工具后的一个模块分析结果，可以帮助快速理解测试工具的各个组件，也可以作为自行开发性能测试工具的参考。

图中间的 DUT 表示被测系统，右上角的部分对应前面讲到的真实用户行为的模拟，通过手工编写脚本、录制或者界面配置等几种方法构建虚拟用户。完成这部分功能的组件称为虚拟用户生成器，其产出的数据 / 配置 / 脚本提供给流量发生器来执行。

流量发生器主要有三个功能：

- 完成和被测服务的协议交互。以 HTTP 为例，需要基于准备好的数据建立网络连接，发起 HTTP 请求，接收响应并处理数据。
- 通过多进程 / 多线程 / 多主机等并发方式将虚拟用户数量放大。
- 记录执行过程中的部分结果，比如协议的完成情况，包含结果和响应时间等，以及并行度等数据。

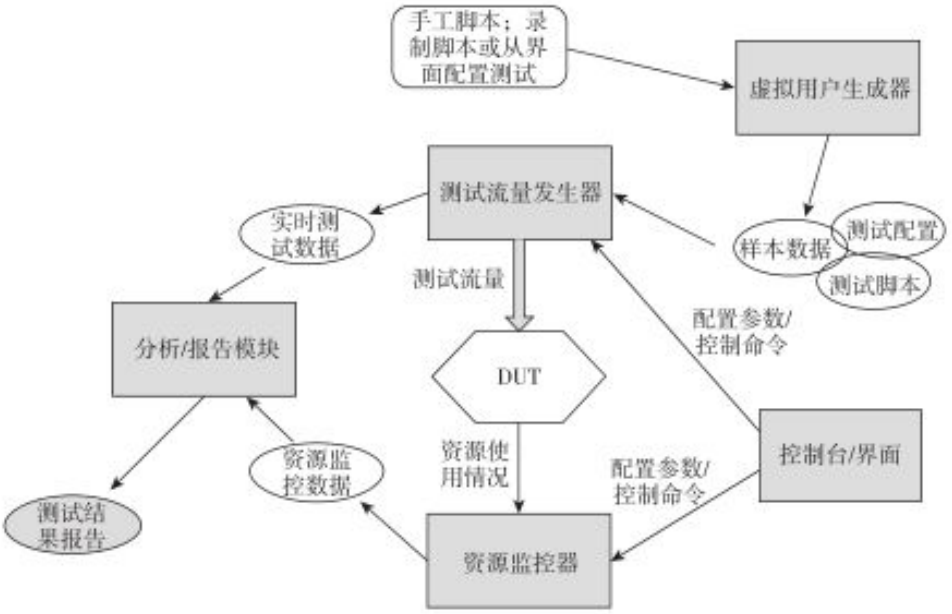


图 3-104 性能测试工具的常见组成部分

控制台和界面主要功能是提供测试人员的操作平台，并实时展示相关的数据。资源监控器的作用是在性能测试的执行过程中，同步收集被测系统的资源使用数据，包括 CPU、内存等方面。这些数据，加上之前流量发生器收集到的相关数据构成了比较完整的测试数据。这些原始数据通常可读性比较差，所以很多测试工具也提供了数据分析和报告模块，可以在性能测试过程中做一些实时的汇总统计，有些也在测试结束后进行异步的详细分析，最终给出一个比较结构化的测试报告。

需要指出的是，以上讨论的是一个相对比较完整的性能测试工具的组件，实际中每个工具提供的部分可能不同。

下面以 Apache AB 工具为例，它是一个开源的工具，使用起来非常简单，能满足最基本的静态 HTTP 并发测试的需求。简单的使用方法如下：

```
ab -c 10 -n 1000 http://127.0.0.1/index.php
```

ApacheBench 参数格式：

```
ab [options] [http://]hostname[:port]/path
```

参数说明：

```
-n requests Number of requests to perform
//在测试会话中所执行的请求个数（本次测试总共要访问页面的次数）。
-c concurrency Number of multiple requests to make
//一次产生的请求个数（并发数）。
```

```
-t timelimit Seconds to max. wait for responses
//测试所进行的最大秒数。其内部隐含值是-n 50000。
-p postfile File containing data to POST
//包含了需要POST的数据的文件，文件格式如 "p1=1&p2=2" .使用方法是 -p 111.txt
-T content-type Content-type header for POSTing
//POST数据所使用的Content-type头信息，如 -T "application/x-www-form-urlencoded"
-X proxy:port Proxyserver and port number to use
-k Use HTTP KeepAlive feature
```

如果按照上面模块划分来看，AB 通过简单的参数配置提供了一个很基础的虚拟用户生成器，有一个流量发生器来产生并发的请求。同时有一个简单的分析和报告模块，结果如图 3-105 所示，并不提供资源监控和可交互的控制台。

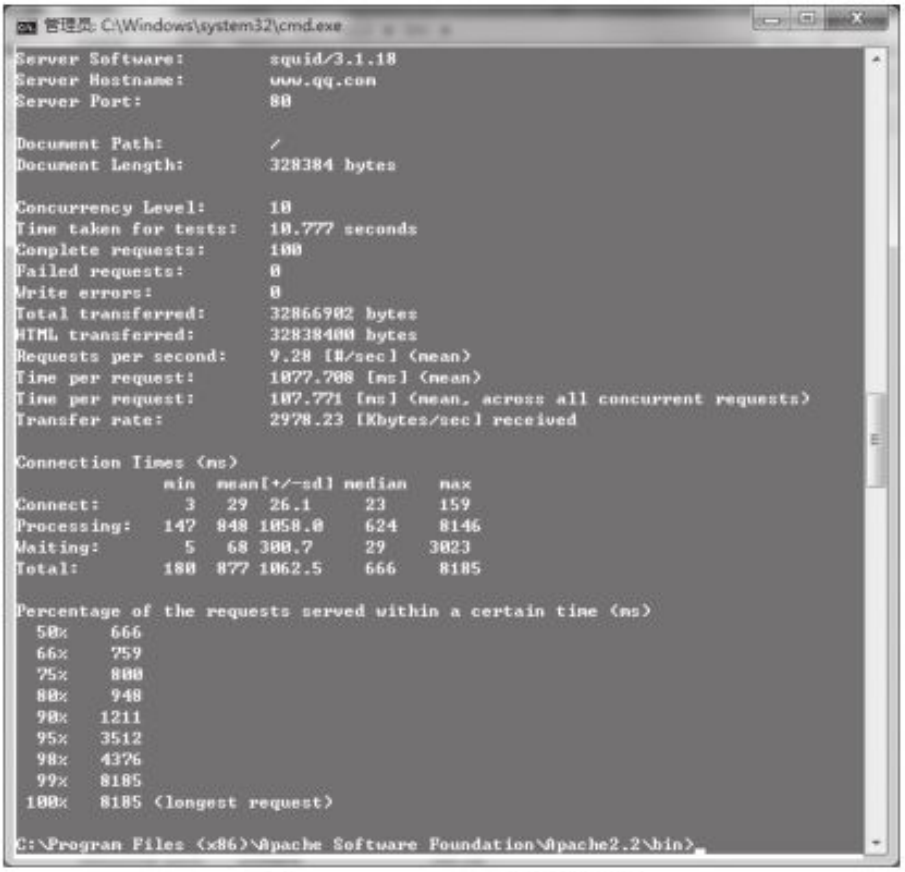


图 3-105 Apache AB 的执行结果

JMeter 是一个广泛使用的性能测试工具，提供的组件比较完善，虽然默认不提供资源使用的监控，但是也有一些第三方插件可以支持远程同步获取服务器的资源使用信息。类似 LoadRunner 这类成熟工具在以上各个方面都提供比较完整且比较强大的功能。

2. 性能测试工具的评估

如果我们觉得使用一个外部的工具，那么就需要进行测试工具的评估，从之前的经验来看，评估中需要考虑下面这些方面：

- 支持的协议和功能是否能满足测试的需求，如果不直接满足，是否有可扩展的方法。
- 数据分析和报告功能是否能满足对于测试指标的要求，如果不能，是否有数据文件可以导出进行二次分析。
- 工具本身的性能如何，就好比一杆称有自己的称重范围，能否支持到我们测试需要的并发量级，工具本身是否支持多机等并发方式。
- 工具是否成熟且有业界其他公司的应用经验。
- 是否需要付费，或者免费版有什么局限，如果需要付费 License 模型是怎样的。
- 公司内部或者客户对该工具的认可程度，当需要对外发布测试报告的时候更需要关注。
- 工具本身是否还在继续开发和维护中。
- 是否有良好的售后支持，或者成熟的技术社区可以获得支持。

3. 自研工具的注意事项

有时候我们会需要在内部开发专门的性能测试工具，不过在这样做之前我们需要做一些认真的考虑，因为实际中，开发和后续持续改进及维度的代价比较大。以下是一些需要考虑的方面：

- 为什么必须自行开发工具，是否广泛评估现有工具都不能满足要求。
- 是否可以在现有开源工具基础上进行二次开发。
- 性能测试工具本身也是需要高性能的，需要在这方面有一定的技术积累。
- 性能测试工具因为常常需要高并发长时间的运行，对稳定性也有很高的要求。
- 在工具开始使用的早期要非常仔细地去校验测试数据，排除工具本身缺陷导致的数据问题，否则会带来很大的误导。
- 可以和一些类似工具做对比测试来验证工具本身。
- 一开始就要考虑到后续的更新和维护的安排。

3.3.3 测试数据的收集

性能测试中还有一个很重要的工作是收集测试数据。大体来说，数据可以分为三种类型：

- 产品的主要性能指标。
- 被测产品的系统资源使用情况。
- 被测产品的内部指标。

接下来分别进行一些介绍和说明。

3.3.3.1 产品的主要性能指标

这部分的数据通常是由测试工具直接给出，直接和性能测试的指标相关。常见的有下面这些：

- 支持的最大并发用户数。
- 每秒处理的请求或事务（有逻辑意义的多个请求的组合）数。
- 请求响应时间。
- 支持的最大带宽。

针对这部分的性能指标，有几个方面需要注意：

1. 作为性能指标，业务逻辑的正确性是前提

当我们谈论性能测试指标，比如每秒处理的请求数，其实有一个隐含的前提是业务逻辑正常处理完成，严格来说是每秒成功处理的请求数。

为了保证这一点，需要设置合理的响应断言，比如对于很多接口响应，即便出错在 HTTP 协议层面，仍然是返回 Code 200。如果只是依赖工具自身 HTTP 层面的检查，那么出错的响应也都会被认为是正常的，如果这个时候大面积出错，性能测试的数据是完全不可信的。针对这种情况，需要添加对响应结果的断言。另外也可以在测试完成后检查被测系统的数据，比如对于提交订单的结果的性能测试，可以事后查看被测系统在那个时间段是否生成了对应数量的订单。还有一个方法是查看被测系统的错误日志，看性能测试阶段是否有错误产生。

2. 工具给出的指标和我们的理解是否一致

比如对比每秒处理的请求数量，不同的工具有多种表述方式，常见的有 page/s、request/s、tran/s（transaction/s 的简写）、hit/s 等。如果不确定是否和自己的理解一致，可以在低并发情况下跑少量的请求，然后人工去验证数据。

3. 对于异常数据的容忍度

在高并发的性能测试中，很难保证每一个请求都被正确的处理，会出现少量的超时或者错误。针对这样的情况，需要设定一个对于错误的容忍度，比如设定小于千分之一的响应错误为可以接受，此时的性能测试数据仍认为有效。当然，在实际项目中，这个数值的设定需要与相关的干系人达成一致。

4. 关于响应时间的理解

响应时间是一个很重要的性能指标，常见的响应时间有下面几种：

- **事务的响应时间**：这个响应时间通常是看整个事务处理完成的时间，比如提交一个订单到得到完整的响应。如上面所说，事务的响应时间计算可以根据业务逻辑的需要来设定计时的起点和终点。
- **页面响应时间**：这个是针对 Web 请求而言的，表示的是从发起请求到把整个页面的元素全部取回来的时间。
- **URL 响应时间**：通常一个页面包含不止一个元素，对应的 URL 也不止一个。URL 响应时间表示的是从收到请求的 ACK 到完整取回一个 URL 的数据的时间。
- **服务器响应时间**：这个是指服务器收到请求到开始返回请求的时间。其计算方法一是从服务器端来看，二是从客户端来看。从客户端来看的时候需要从发起请求到得到结果之间的时间中减去网络传输的时间。
- **连接建立时间**：从 TCP 协议发出 SYN 到收到 SYN、ACK 的时间，其衡量的是来回的网络延迟。
- **第一个数据字节的时间**（TTFB，Time to First Byte）：这个相比上面而言，是更细节的响应时间，简单来说就是指从发出 TCP 请求到第一个数据包返回的时间，因为一个 URL 的数据可能包含多个数据包。如果网络状况比较差，或者服务器已经过载导致丢包较多，那么重传的时间也会被计算在内，导致时间加长，所以这个响应时间对网络的延迟也比较敏感。

图 3-106 是一个 TCP 层面的示意图，可以看出上面几个响应时间指标的定义。

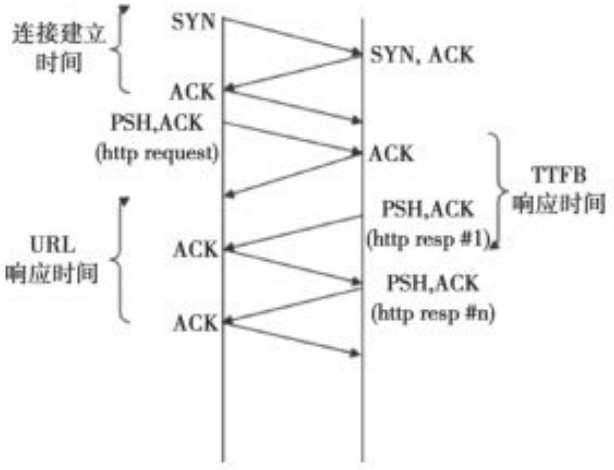


图 3-106 从 TCP 连接角度看响应时间

这样细分的一个好处是可以帮助进行问题的定位。如果发现事务的响应比较慢时，可以进一步查看上面提到的细分指标。如果发现建立链接时间过长，可能是因为服务器的连接数（可以对比服务端的这个资源使用情况）已经很多了，无法快速创建更多 TCP 链接，或者是否有内存不够或者端口耗尽的情况。当业务层逻辑处理很慢的时候可能表现为连接建立时间很短，但是 TTFB 很长。

3.3.3.2 服务器资源使用情况

性能测试数据收集中很重要的一部分是被测系统的资源使用情况，因为系统性能和资源使用是密切相关的，主要的目的有下面几个方面：

- 了解在当前压力下，系统各项资源的使用情况，也可以用于横向对比。
- 通过资源使用情况的分析可以看出当前是否测出了系统最大的性能。
- 是否有某项资源的使用已经到达上限，成为瓶颈。
- 是否有其他非被测系统的模块占用了资源。

通常在性能测试中，测试人员都会去收集 CPU、内存、网络等服务器资源使用情况，但是如果只是笼统地给出一个百分比是不够的，需要进一步细分来提供更多有参考价值的数据。下面以 CPU 和内存为例做一些细化分析。

1. CPU 使用率的细节

这里以 Linux 平台为例来讲解，Windows 上也可以做类似的细分。

图 3-107 所示是在一台 Linux 主机上执行 top 命令的结果，可以看到在 CPU 使用率那一行还有多个不同维度的指标，而不只是一个简单的百分比。

```
top - 23:52:27 up 7 days,  2:55,  1 user,  load average: 0.55, 0.75, 0.82
Tasks: 103 total,   2 running,  96 sleeping,   0 stopped,   5 zombie
Cpu(s): 17.3% us,  6.0% sy,  0.0% ni, 75.5% id,  1.2% wa,  0.0% hi,  0.0% si
Mem:   2074908k total,  2030604k used,    44304k free,   102040k buffers
Swap:  2031608k total,      208k used,  2031400k free,  1590300k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
26456 mysql     15   0  376m 196m 3908 S    2  9.7 135:35.48 mysqld
21338 nobody    15   0 10428 3548 1168 S    1  0.2   0:00.16 httpd
14279 nobody    15   0 10564 3604 1168 S    0  0.2   0:00.40 httpd
16986 nobody    15   0 10700 3712 1168 S    0  0.2   0:00.29 httpd
19691 nobody    15   0 10564 3628 1164 S    0  0.2   0:00.21 httpd
19917 nobody    15   0 10564 3604 1164 S    0  0.2   0:00.21 httpd
21335 nobody    15   0 10564 3648 1164 S    0  0.2   0:00.14 httpd
21370 nobody    15   0 10428 3564 1168 S    0  0.2   0:00.17 httpd
22977 nobody    15   0 10292 3452 1160 S    0  0.2   0:00.10 httpd
```

图 3-107 top 命令观察到的 CPU 使用情况

上面几个细分项的说明如下：

- us: user time，处于用户态的运行时间占比，不包含 nice 值为负进程。
- sy: system time，系统处于核心态的运行时间占比。

- ni: nice time，用户进程空间中改变过优先级的进程占用 CPU 百分比。
- id: idle time；CPU 空闲时间百分比。
- wa: io wait time，IO 等待时间占比。
- hi: hard irq time，CPU 处理硬中断所占用的时间百分比。
- si: software irq time，CPU 处理软中断所占用的时间百分比。

从以上几个细分项的说明，我们发现这些比一个简单的总体使用率能提供更多有价值的信息。当 CPU 使用率较高的时候，我们通过进一步观察上面各个单项，可以推断是哪部分占用比较高，比如是否因为 IO 等待时间长或者过多处理中断导致。

接下来我们看 CPU 使用率的计算过程，也可以帮助我们进一步理解这个指标。

在系统的 /proc/stat 文件中包含了所有 CPU 活动的信息，该文件中的所有值都是从系统启动开始累计到当前时刻。需要注意的是不同内核版本中该文件的格式可能略有差异。

以下是某台主机的相关信息，截取了其中一部分。

```
bash-3.00# cat /proc/stat | grep "cpu "
cpu  1129215 0 1311793 37097490 70380 4425 11038
```

以上数据分别对应 us、ni、sy、id、wi、hi、si，其单位是 jiffies。jiffies 是内核中的一个全局变量，用来记录自系统启动以来产生的节拍数。在 Linux 系统中，一个节拍大致可理解为操作系统进程调度的最小时间片，不同 Linux 内核可能会值不同，通常在 1ms 到 10ms 之间。

下面是计算方法：

1）采样两个足够短的时间间隔的 CPU 信息快照，分别记作 t1、t2，其中 t1、t2 的结构均为：(user、nice、system、idle、iowait、irq、softirq) 的 7 元组；

2）计算总的 CPU 时间片：

a）把第一次的所有 CPU 使用情况求和，得到 s1。

b）把第二次的所有 CPU 使用情况求和，得到 s2。

c）s2-s1 得到这个时间间隔内的所有时间片。

3）计算空闲时间 idle。

idle 对应第四列的数据：

i2 - i1 = 第二次的第四列 – 第一次的第四列

4）计算总 CPU 使用率

pcpu =(1 - (i2-i1)/(s2–s1)) * 100%

2. 内存使用率的细节

在性能测试中，内存的使用情况是一个很重要的监控点，不管是从资源使用的角度还

是从发现内存泄漏问题的角度。同样，内存也不只是一个简单的使用字节数或者百分比的指标，也需要进一步细分来看。

图 3-108 所示的是 Linux top 命令中关于内存的信息。

```
top - 23:52:27 up 7 days,  2:55,  1 user,  load average: 0.55, 0.75, 0.82
Tasks: 103 total,   2 running,  96 sleeping,   0 stopped,   5 zombie
Cpu(s): 17.3% us,  6.0% sy,  0.0% ni, 75.5% id,  1.2% wa,  0.0% hi,  0.0% si
Mem:   2074908k total,  2030604k used,    44304k free,   102040k buffers
Swap:  2031608k total,      208k used,  2031400k free,  1590300k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
26456 mysql     15   0  376m 196m 3908 S    2  9.7 135:35.48 mysqld
21338 nobody    15   0 10428 3548 1168 S    1  0.2   0:00.16 httpd
14279 nobody    15   0 10564 3604 1168 S    0  0.2   0:00.40 httpd
16986 nobody    15   0 10700 3712 1168 S    0  0.2   0:00.29 httpd
19691 nobody    15   0 10564 3628 1164 S    0  0.2   0:00.21 httpd
19917 nobody    15   0 10564 3604 1164 S    0  0.2   0:00.21 httpd
21335 nobody    15   0 10564 3648 1164 S    0  0.2   0:00.14 httpd
```

图 3-108　top 命令所示的内存使用情况

这里显示的整个系统层面汇总的内存使用信息，其中 Mem 表示物理内存，Swap 称为交换区。

交换区逻辑上可看作是内存的一部分，但它是从硬盘中划分出来的。概念上和 Windows 下的虚拟内存一样，当系统的物理内存不够用的时候，需要将物理内存中一部分暂时没有操作的程序占用空间释放出来，以供当前运行的程序使用。这些被释放的空间被临时保存到 Swap 磁盘文件中，等到那些程序要运行时，再从 Swap 文件中恢复保存的数据到内存中。系统是在物理内存不够时，才进行 Swap 交换，所以通过观察 Swap 的使用情况也可以分析物理内存是否充足。由于磁盘文件的读写速度比物理内存慢很多，所以当系统大量读写 Swap 文件的时候，也会导致性能下降。Swap 也是系统参数指定好的，如果 Swap 空间用光了，系统会直接报错。

和 CPU 使用情况类似，Linux 也提供了一个文件 /proc/meminfo 来记录内存使用情况，图 3-109 所示是其部分内容。

```
# cat /proc/meminfo
cat /proc/meminfo
MemTotal:         390992 kB
MemFree:          247176 kB
Buffers:               0 kB
Cached:            61808 kB
SwapCached:            0 kB
Active:            84832 kB
Inactive:          33940 kB
Active(anon):      71484 kB
Inactive(anon):        0 kB
```

图 3-109　meminfo 的部分结果

参数说明如下：

- total：总计物理内存的大小。
- used：已使用多大。
- free：可用有多少。
- Shared：多个进程共享的内存总额。
- Buffers/Cached：磁盘缓存的大小。从应用程序的角度来说，可用内存 = 系统 free memory+buffers+cached。
- Buffers：用来给块设备做的缓冲大小，它只记录文件系统的 metadata 以及 tracking in-flight pages。
- Cached：是用来给文件做缓冲，记录文件的内容。

以上讨论的是总体的内存使用情况，还可以从单个具体进程的角度来分析，数据来源于 /proc/[pid]/status 文件，其中 pid 是进程 ID。一个例子如图 3-110 所示。

```
root@rickyVPS:~ #
root@rickyVPS:~ # ps -ef | grep mysql
root     25143 25063  0 11:20 pts/0    00:00:00 grep mysql
root     25769     1  0  2013 ?        00:00:00 /bin/sh /usr/bin/mysqld_safe --dat
-socket=/var/lib/mysql/mysql.sock --pid-file=/var/run/mysqld/mysqld.pid --basedir=
mysql    25874 25769  0  2013 ?        03:21:16 /usr/libexec/mysqld --basedir=/usr
ysql --user=mysql --log-error=/var/log/mysqld.log --pid-file=/var/run/mysqld/mysql
ib/mysql/mysql.sock
root@rickyVPS:~ #
root@rickyVPS:~ # cat /proc/25874/status
Name:   mysqld
State:  S (sleeping)
Tgid:   25874
Pid:    25874
PPid:   25769
TracerPid:      0
Uid:    27      27      27      27
Gid:    27      27      27      27
Utrace: 0
FDSize: 256
Groups: 27
VmPeak:   727328 kB
VmSize:   727324 kB
VmLck:         0 kB
VmHWM:     34392 kB
VmRSS:     28048 kB
VmData:   667072 kB
VmStk:        88 kB
VmExe:      6532 kB
VmLib:      7328 kB
VmPTE:       308 kB
VmSwap:        0 kB
Threads:        10
SigQ:   0/3782
```

图 3-110 某个进程的内存使用情况

下面我们看一下其中主要的一些细分指标：

- VmSize(KB)：虚拟内存大小。整个进程使用虚拟内存大小，是 VmLib、VmExe、VmData 和 VmStk 的总和。
- VmRSS(KB)：虚拟内存驻留集合大小。这是驻留在物理内存的一部分，它没有交换到硬盘，包括代码、数据和栈。
- VmData(KB)：程序数据段的大小（所占虚拟内存的大小），堆使用的虚拟内存。
- VmStk(KB)：任务在用户态的栈大小，栈使用的虚拟内存。
- VmExe(KB)：程序所拥有的可执行虚拟内存的大小，代码段，不包括任务使用的库。
- VmLib(KB)：被映射到任务的虚拟内存空间的库的大小。

对于核心的服务进程，我们需要从上面细分的角度来分析内存使用情况，找出潜在的内存泄漏问题和瓶颈。

3.3.3.3 产品的其他指标数据

性能测试中收集的数据除了前面提到的产品性能指标，以及系统资源的使用情况，还有一类指标虽然不一定会直接出现在测试报告中，但是对于性能测试结果分析和问题定位非常有价值。这一类数据直接和被测系统的内部运行情况相关，比如：

- 主程序的进程 / 线程数，及动态增长情况。
- 数据队列的使用和增长情况。
- Cache/Buffer 的使用情况。
- 打开的文件句柄数。
- 目录下的文件数，日志大小等。

下面来看两个实际测试中的例子。

有很多服务端系统的进程数 / 线程数是可以随着待处理请求数动态调节的。从图 3-111，我们可以很直观地看到性能测试的执行过程，进程的变化情况，可以帮助我们了解系统在大压力下的运行情况。

以上图 3-112 和图 3-113 是在不同的压力级别下观察到的，测试 1 中队列里的文件数在不断波动，但是总体数量比较平稳，具有可持续性。而测试 2 中由于测试流量较大，文件来不及被处理，导致在队列中不断堆积，直至队列的上线。这样的性能数据是不具有可持续性的，如果测试时间过短，问题无法暴露出来。另一方面，也可以看到系统的部分瓶颈所在。

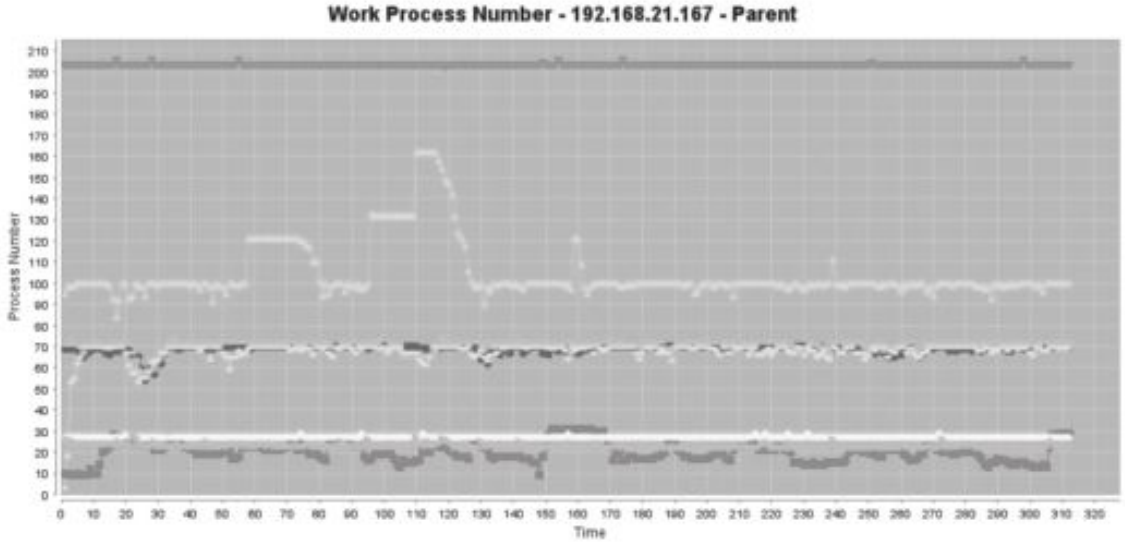


图 3-111 被测系统各个模块的核心进程数动态变化情况

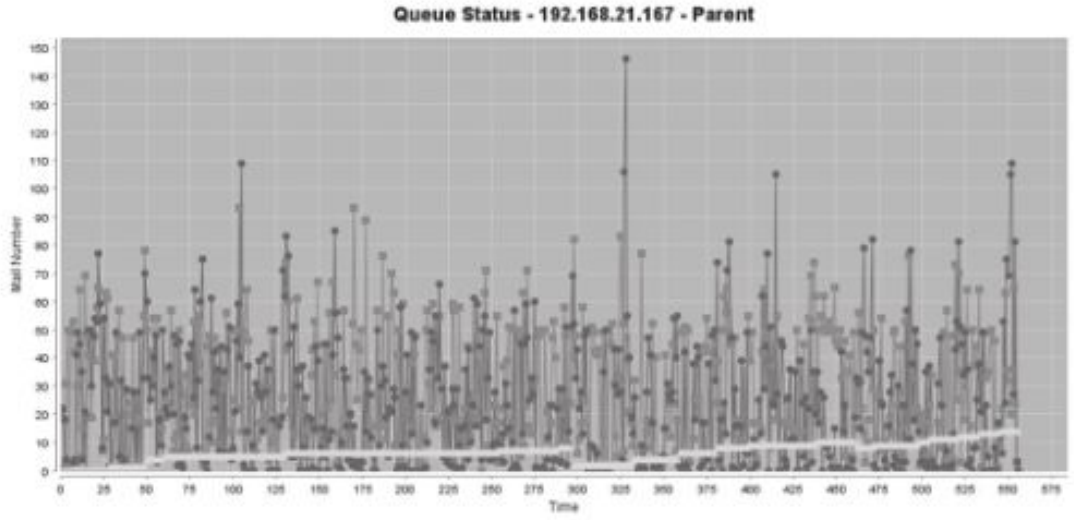


图 3-112 被测系统的队列中待处理文件数 – 测试 1

从以上例子可以看出，这一类的指标在性能测试中对于我们分析和定位问题也是非常有帮助，应该在测试过程中进行观察。

3.3.4 分析和报告

在性能测试执行完后，需要对数据进行整理分析并汇总输出测试报告。这里讨论两个比较共性的问题：一是数据的分析，二是性能测试报告编写的注意事项。

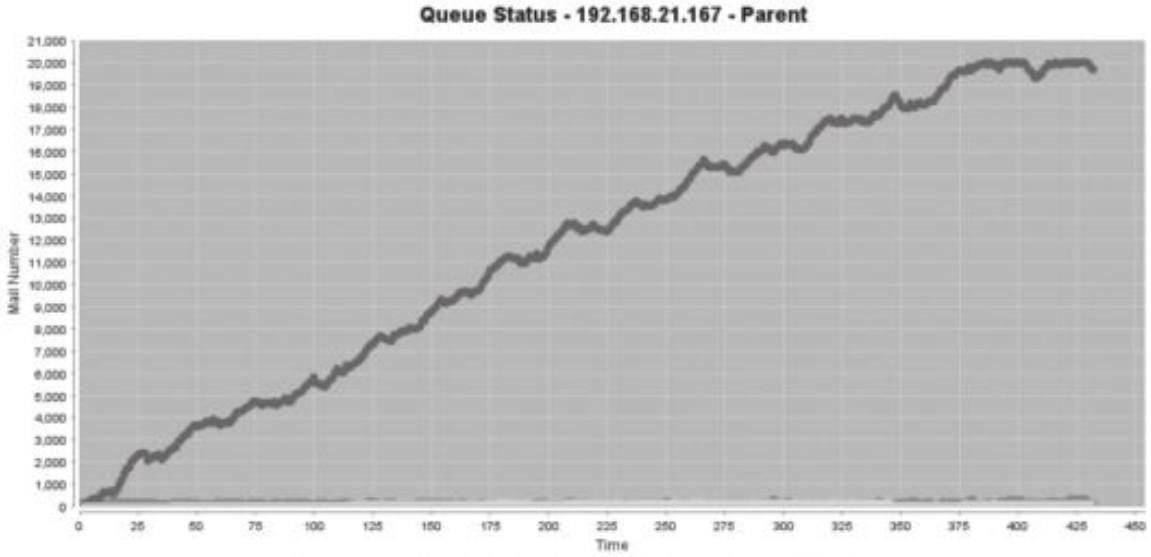


图 3-113 被测系统的队列中待处理文件数 – 测试 2

1. 性能数据的计算方式

常用的计算方式主要有下面几种：

- **平均值**——这个方式使用最普遍，它可以将大量的同类数据汇总成一个值，概念也比较容易理解，所以一般结果中会包含这个值。
- **标准差**——这是统计中常用的一种工具，用来反映个体的离散程度。

上面平均值的做法看起来非常的公平，但是实际上，很多信息被遗漏，有时并不能反映真实的情况。考虑下面的两组数据：

数据集 C：{198，220，204，199，210，206，202，212，205，213，204，209，218，200，219，225，206，196，215，211}

数据集 D：{150，231，175，186，225，247，172，182，263，242，195，236，229，210，168，274，160，225，210，192}

其平均值都为 Avg = 208.6

但是如果把这些点在坐标轴上画出来就会发现有差异，如图 3-114 和图 3-115 所示。

从图中可以直观看出，数据集 C 对应的数据更加靠近均值，波动比较小，而数据集 D 的数据更加离散。如果这些数值代表响应数据，那么 C 的稳定性更好。

以上是直观的感觉，从数学的角度，可以用标准差来衡量数据偏离均值的程度。标准差也称为标准偏差，或者实验标准差，公式为

$$\sigma - \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

如是总体（即估算总体方差），根号内除以 n（对应 excel 函数：STDEVP）；如是抽样（即估算样本方差），根号内除以 $n-1$（对应 excel 函数：STDEV）

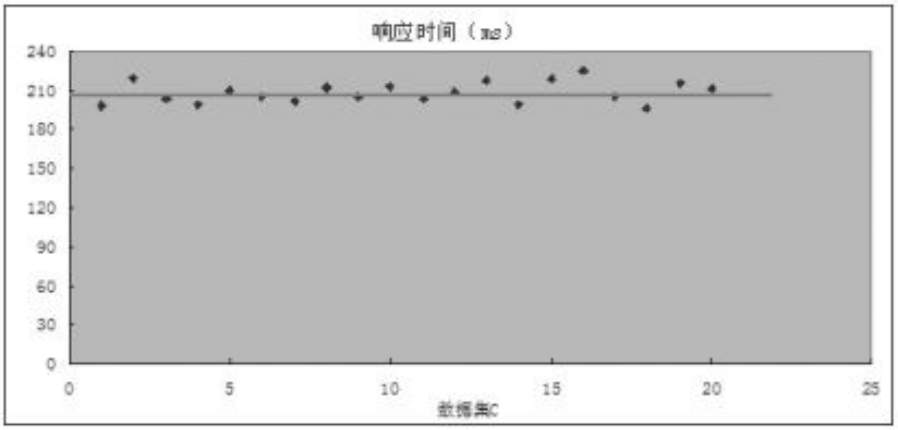


图 3-114 数据集 C 对应的坐标图

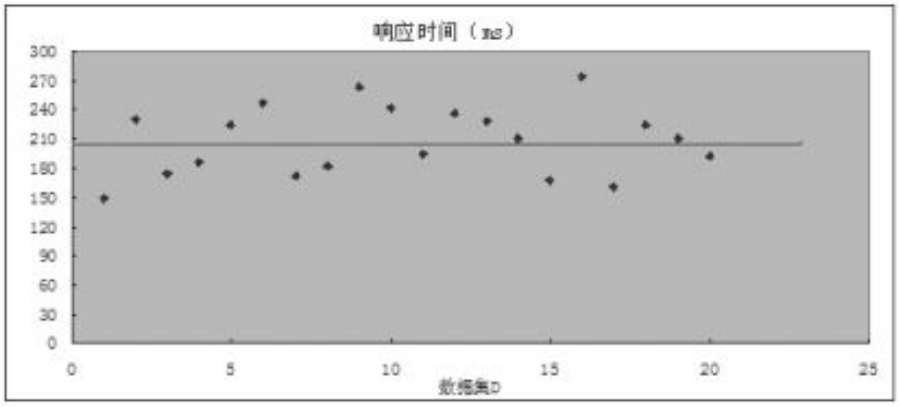


图 3-115 数据集 D 对应的坐标图

以上两组数据的标准差计算结果如下：

数据集 C：STDEVP = 7.83，STDEV = 8.03

数据集 D：STDEVP = 34.37，STDEV = 35.27

所以标准差也可以帮我们从另一个维度来观察数据：

- **最小 / 最大值**——这个指标在统计结果中也经常使用，而且一般测试工具也会直接给出，可以帮助了解极端的情况。不过由于它直接代表的是个案，当数据量非常大的时候参考意义不大，但是可以针对从日志中查看极端数据对应的问题。
- **中位值**（Median）——其计算方式是将所有数据从小到大或从大到小排列，奇数个数的话取中间的数字，偶数个数的话取中间两个数的平均数，然后得到的值。

需要理解标准差与平均值代表意义的区别，简单来说可以知道有一般的数据在这个之

上或之下。还有很多计算方式类似的演变形式，比如 90% 值，对应响应时间，通常代表 90% 的响应时间在这个数据以下；另外还有上四分位数和下四分位数。

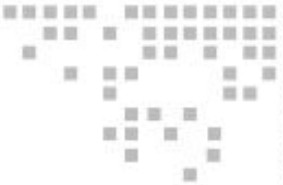
2. 测试报告的注意事项

性能测试报告是测试结论和产出的集中展示，也可能会作为一份正式的报告文件。编写的时候需要注意以下几个方面：

- 简要说明性能测试的被测系统，测试的目的。
- 在报告前面章节给出汇总后的测试结果摘要，以及对应的结论。因为对部分人可能不需要了解详细的数据。
- 在后续的场景需要对测试的场景、使用的测试方法和工具做简要的介绍，让阅读报告的人理解这些数据是在何种情况下给出的。
- 需要对测试环境以及被测系统的平台、版本等信息给出说明。
- 对于一些重要的参数配置，就是前面提到的关键细节，需要说明。
- 在性能测试过程中发现的问题也可以提交到 bug 管理系统来跟进，在报告上可以附上发现的问题。

3.4 本章小结

本章从三个方面介绍了一些性能测试方法和注意事项，包括 Web 前端的性能测试、App 前端的性能测试，以及后台服务的性能测试。整体来说，性能测试是一个非常综合性的测试活动，涉及面非常广，包括对于被测系统架构、实现和使用方式的理解，对网络、操作系统和数据库等通用的计算机技术的掌握，以及对于测试工具和常用方法的理解，另外还需要和不同角色的团队成员进行沟通协调，对测试人员的锻炼和考验也非常大。相对于功能测试给出一个通过与否的结论，性能测试是一个开放性的问题，需要去探索给出尽量准确的数据来给业务和研发部门提供有价值的参考信息。



第 4 章 Chapter 4

专项测试

幸福的家庭都是相似的，但不幸的家庭各不相同。

——托尔斯泰

在进行了手工的功能测试，也开发了一些自动化用例，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被反馈回来，比如：

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

这些问题都是使用前面章节的测试方法难以找出的，所以我们需要根据这些问题来探索不同的富有针对性的测试方法，借助这些方法来发现一些共性的深层次的问题。

在本章中我们会重点介绍以下几个测试方法：兼容性测试、流量测试、电量测试、弱网络测试、稳定性测试、安全测试和环境相关测试。以上这些方法都是针对某个特殊方面或者问题的，所以我们统称它们为专项测试。它们是无线互联网产品中重要的测试类型。实践经验证明，这些方法可帮助我们从不同维度发现很多深层次的问题，所以非常值得去了解和实践。

4.1 兼容性测试

兼容性问题是比较容易遇到的一类问题，特别是当 App 的用户量越来越大之后。而且，另一方面，终端设备的型号越来越多，也加剧了这方面的碎片化，使得兼容性成为一个不得不考虑的问题。因为一旦有这方面的问题，就会影响这一类的很多用户，对业务的影响会比较大。

针对这个问题的测试会考虑覆盖多种不同的场景，我们称之为兼容性测试。严格来说，兼容性测试本质上也是功能测试，只不过侧重在不同的软硬件环境。

4.1.1 兼容性测试的准备和手工测试

和所有的测试类型一样，不可能在有限的测试人力和时间情况下覆盖所有的场景，对于兼容性测试而言，这里的取舍更加明显。所以在讨论任何兼容性测试技术和方法之前，都必须考虑的一个问题是：如何圈定测试范围？

这个问题没有标准答案，因为这取决于产品本身所处的阶段，以及对质量的要求。不过有一个思路可以参考，那就是尽量覆盖该产品的主要用户，也就是常说的 Top X 原则。

关于 Top X，建议尽量获取产品自己的数据，因为业界报告给出来的数据是汇总了很多不同类型的 App 给出来的，可能和我们的 App 比例差异较大。具体的技术方案可以采用 App 内部埋点的方式，本书第 8 章关于监控的章节做了一些讨论。

下面给出的例子来自友盟的 demo App，如果通过自研的 SDK 和统计后台，获取的数据也类似。图 4-1 所示是按分辨率统计的前十。

图 4-1 新增用户按分辨率统计

图 4-2 所示是按操作系统版本统计的前十。

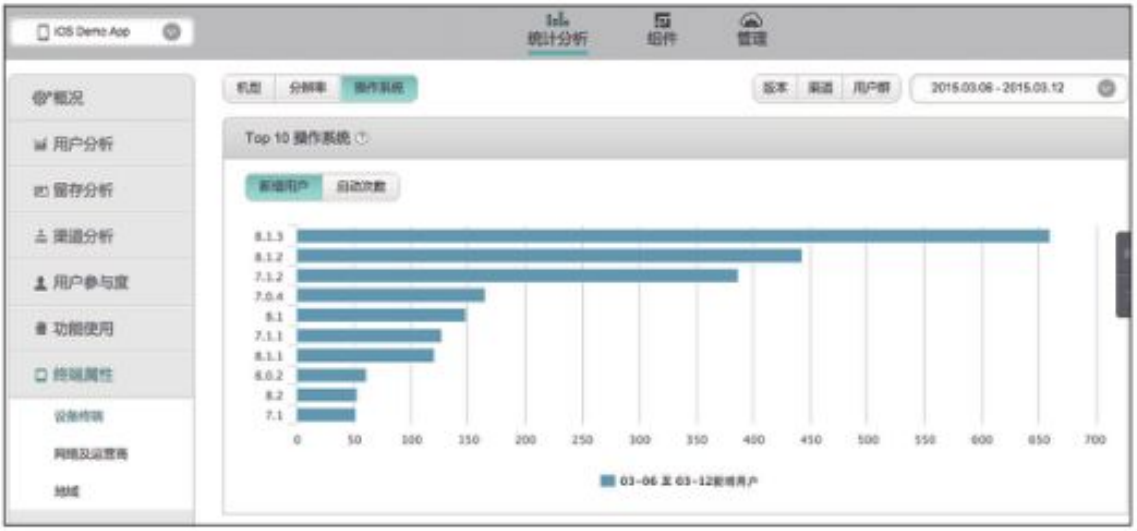


图 4-2 新增用户按操作系统版本统计

当获取到我们的 App 对应的数据时，就可以选取覆盖的范围。

不包括一些针对特定项目的情况，对于移动互联网的产品，常见的兼容性考虑主要有下面几个方面。

针对 App 通常会考虑这些方面：

1）操作系统版本。针对 iOS，截至本书写作时需要考虑 iOS 5.1.1（支持 64 位的最低版本），iOS 6、7、8

针对 Andoird，截至目前通常要考虑 2.3（有少量较老的机型）、4.x，以及 5。

针对每个操作系统大版本下的小版本，比如对于 Android 4，包括了下面这些子版本号：

4.0-4.0.2，4.0.3-4.0.4，4.1，4.2，4.3，4.4

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。

2）屏幕分辨率。由于显示屏技术的不断提升和更新，手机屏幕的分辨率也在逐步提升。以 Android 为例，截至目前，主流机型的分辨率大致经历了 800 × 480、960 × 640、1280 × 720（720p）、1920 × 1080（1080p）、2560 × 1440（2K）等几个阶段。对于 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。

3）不同厂家的 ROM。这个主要是 Android 系统的碎片化引起的问题，几乎每个

Android 手机厂商都对 Android 系统进行了或深或浅的定制。实际中我们也确实曾遇到一些不同厂家 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不同的问题。现实中我们会采购一些主流厂家的手机型号，并在上面验证功能。

4）网络类型。这也是一个需要考虑的问题，涉及 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：

- 主要是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。
- 另外需要考虑的就是屏幕分辨率的问题。

针对以上提到的兼容性问题，基本的做法就是根据 App 用户的特征挑选出要覆盖的范围，然后购买相关的测试设备，在功能测试中抽出一部分时间做兼容性测试。实际中为了效率，不太可能逐个测试用例在每个兼容性的维度来执行，因为功能用例数直接乘以设备数，其结果是无法承受的测试工作量。通常，我们会选择在少数主流设备上执行全量的用例，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地跟踪和了解进展，可以通过一个表格的形式来维护不同功能点的兼容性测试情况。在 bug 管理方面，可以增加对应的选项，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。

4.1.2 基于 UI 自动化脚本的云测试方案

上面提到的手工测试和普通的功能测试没有实质性区别，有对应的设备和时间就可以开展。但是这个方法也会有一定的局限性：

- 很多测试团队不一定有完备的所有类型的设备。特别是对于外部反馈的兼容性问题，立即去采购周期较长，而且可能使用率也会很低。
- 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，重复劳动也容易使人厌倦。
- 在不同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。

基于以上问题和新的 App 云测试技术的出现，我们采用了一些新的兼容性测试的方法，思路大致是这样的：

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反复多次操作。工作量随着需要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

机可供使用，后续还会更多。那么是否可以借助这些远程的真机来做我们的兼容性测试呢？

这个方案看起来是可行的，但是要做起来需要考虑以下几个方面的问题：

- 要借助 UI 自动化的方案，能用很小的代价快速写出针对某个具体功能的 UI 自动化脚本。
- 需要有一个有大量真机的云测试平台。
- 平台要提供一些能力，比如截图（对查看 UI 方面的兼容性问题非常重要）、日志（对于定位问题非常重要），以及一些基本的性能指标，比如安装时间、启动时间、CPU/ 内存使用情况等。
- 需要脚本足够简单，不需要复杂的框架，脚本也要足够直观，这样每个功能测试人员都能够快速地编写。只有这样，这个方法才有可能普及。

基于这样的思路和目标，下面是我们尝试过的一些实践。针对的是一个购物中叫做秒杀的模块，大家可以作为参考。

这里以 testin 为例，如果使用其他 App 云测试平台也是类似的思路。

1. 脚本的编写和本地调试

一个简单的 TestCase 封装，这部分框架是 testin 平台提供的，如图 4-3 所示。

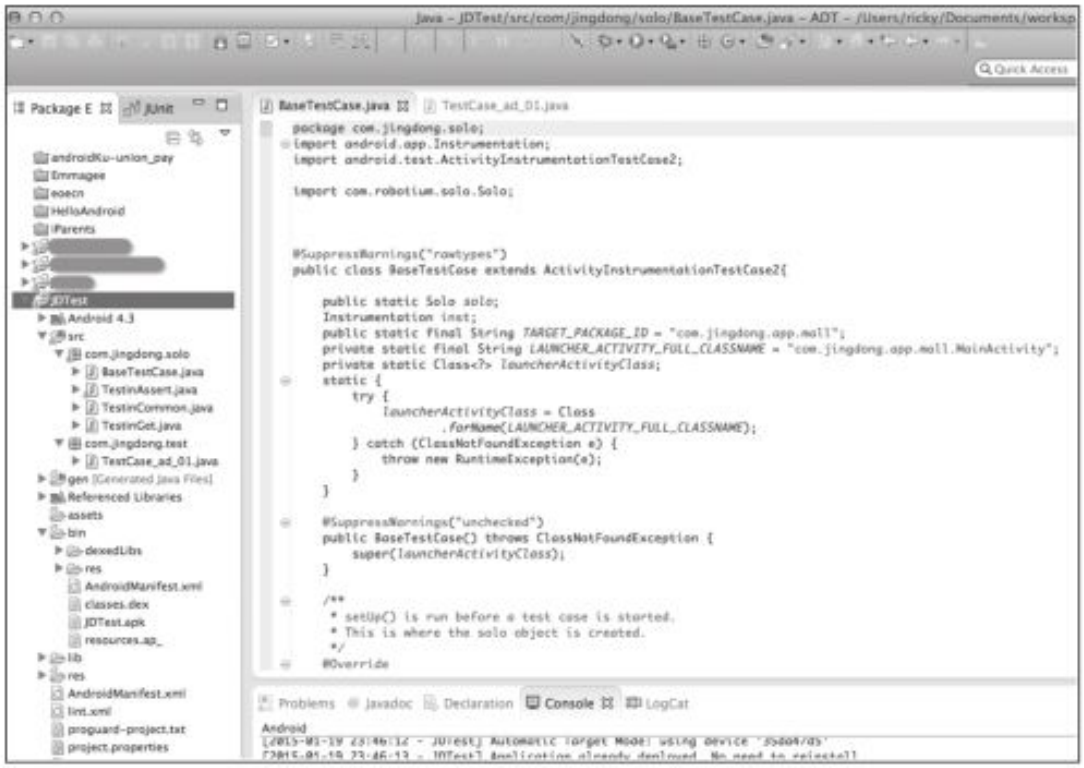


图 4-3 兼容性测试脚本框架

对于每个功能测试人员而言，只需要写如图 4-4 所示的简单代码，就可以驱动一步步 UI 的操作。

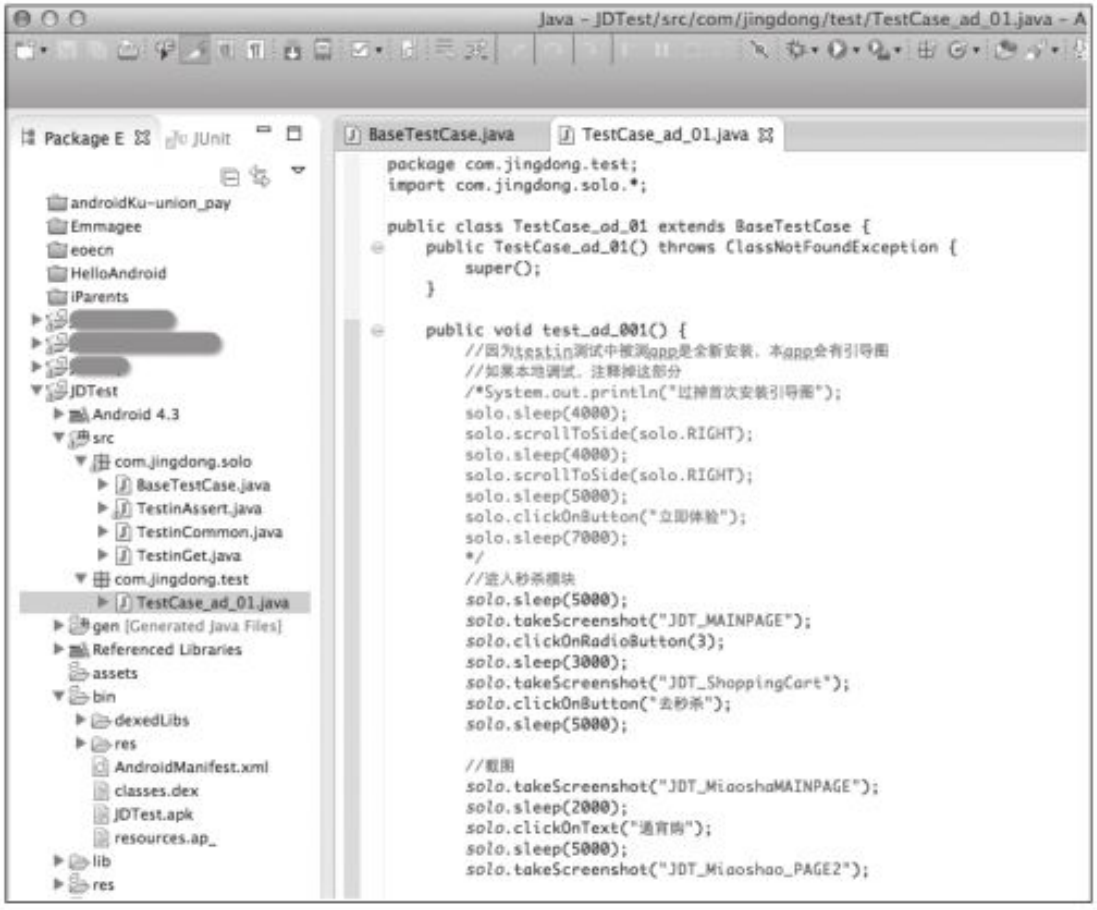


图 4-4　兼容性测试部分脚本

项目中会用到 robotium-solo 这个 jar 包，需要将其加入 build path，如图 4-5 所示。

以下是对于图 4-4 所示脚本的简单说明。

- `solo.sleep(5000);`

等待 App 启动后进入首页，也可以用更复杂一些的 solo.waitForActivity 方法。

- `solo.takeScreenshot("JDT_MAINPAGE");`

产生一张屏幕截图，之后在报告里面可以看到。如果是在云端跑，该图会在报告页面里看到；如果是本地真机调试，截图会存成手机上的图片文件，而里面的字符就是文件名。

- `solo.clickOnRadioButton(3);`

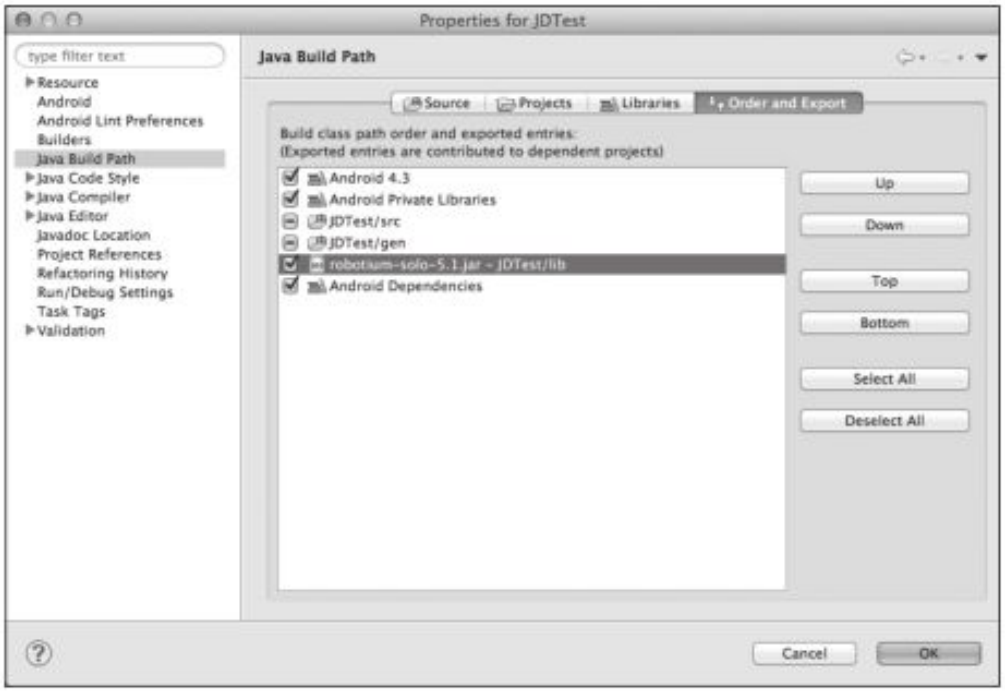


图 4-5 添加 robotium-solo 的包到编译路径

借助 android SDK 里面自带的 tools 下面的 uiautomatorviewer 工具获取 UI 元素的信息，便于代码来定位，如图 4-6 所示。这部分和前面第 2 章介绍的 Android UI 自动化的做法类似。

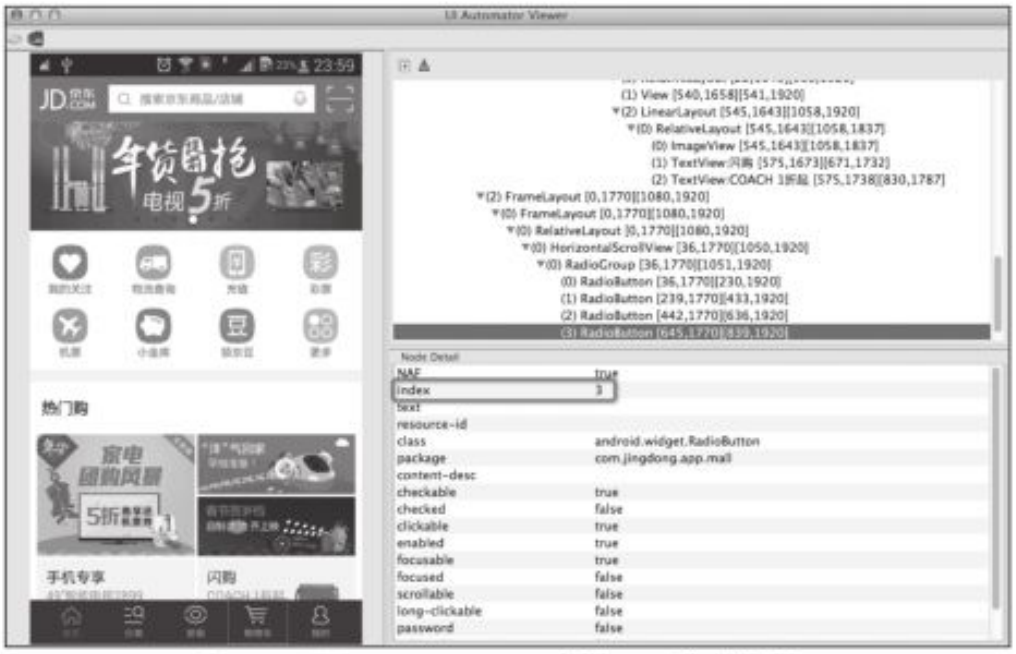


图 4-6 通过 uiautomatorviewer 获取 UI 元素定位信息

- `solo.clickOnButton("去秒杀");`

有些不便于用上述索引方式定位的也可以用文本信息来定位，比如这里的“去秒杀”，如图 4-7 所示。

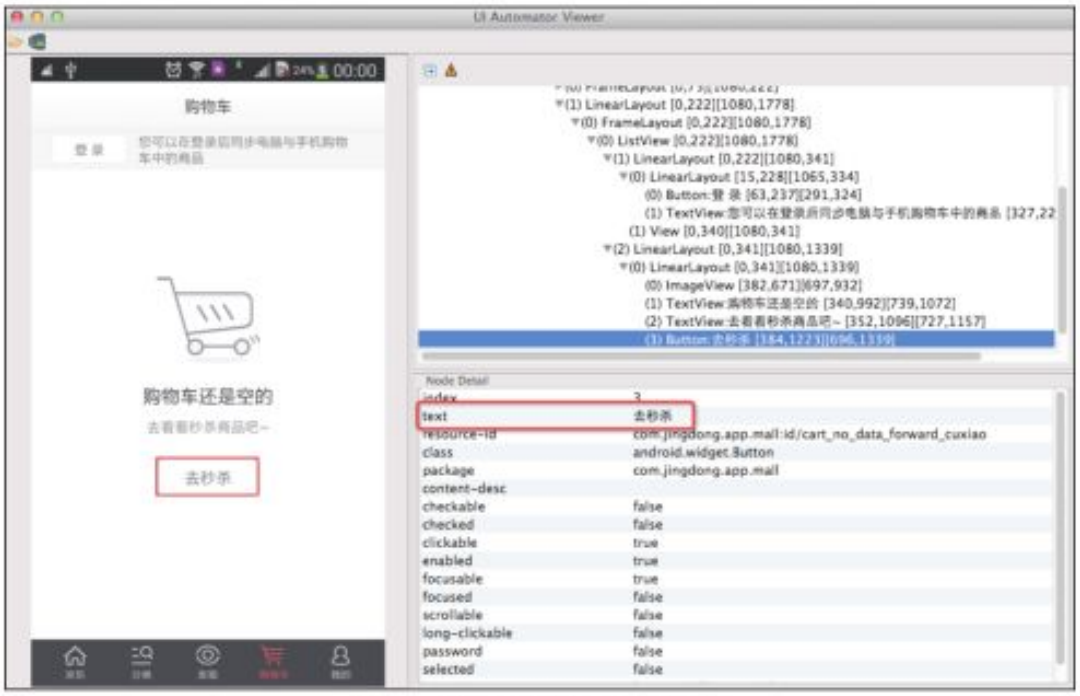


图 4-7 通过文本信息来定位元素

点击 Run As“Android JUnit Test”开始执行，如图 4-8 所示。

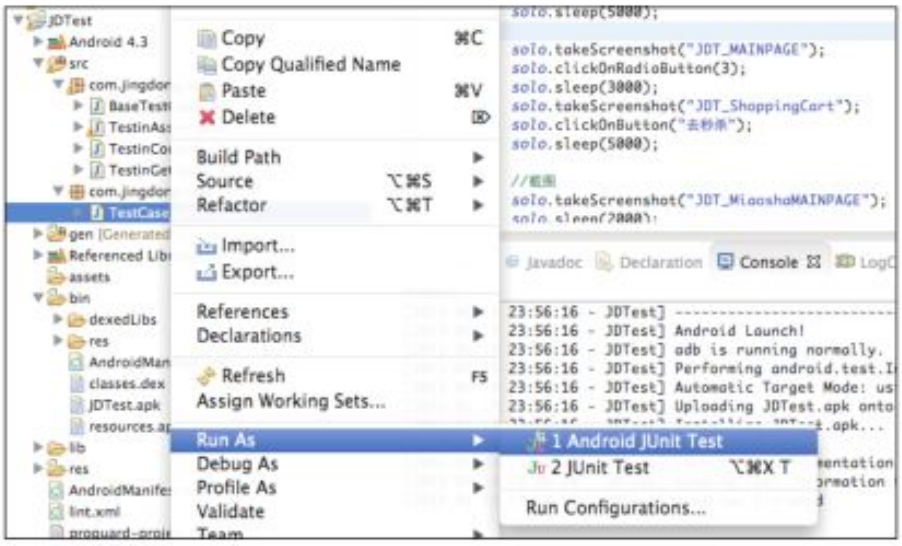


图 4-8 选择 Android JUnit Test 来执行脚本

成功执行后脚本会将被测的 App 启动起来，如图 4-9 所示。

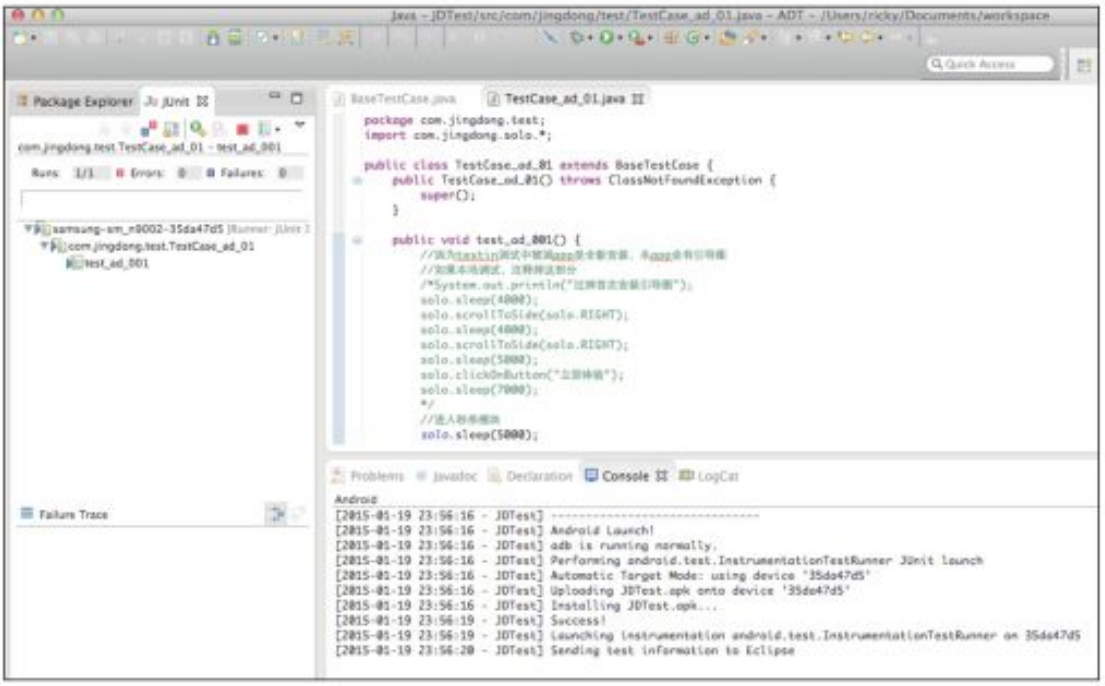


图 4-9 测试脚本执行中

执行结果如图 4-10 所示。

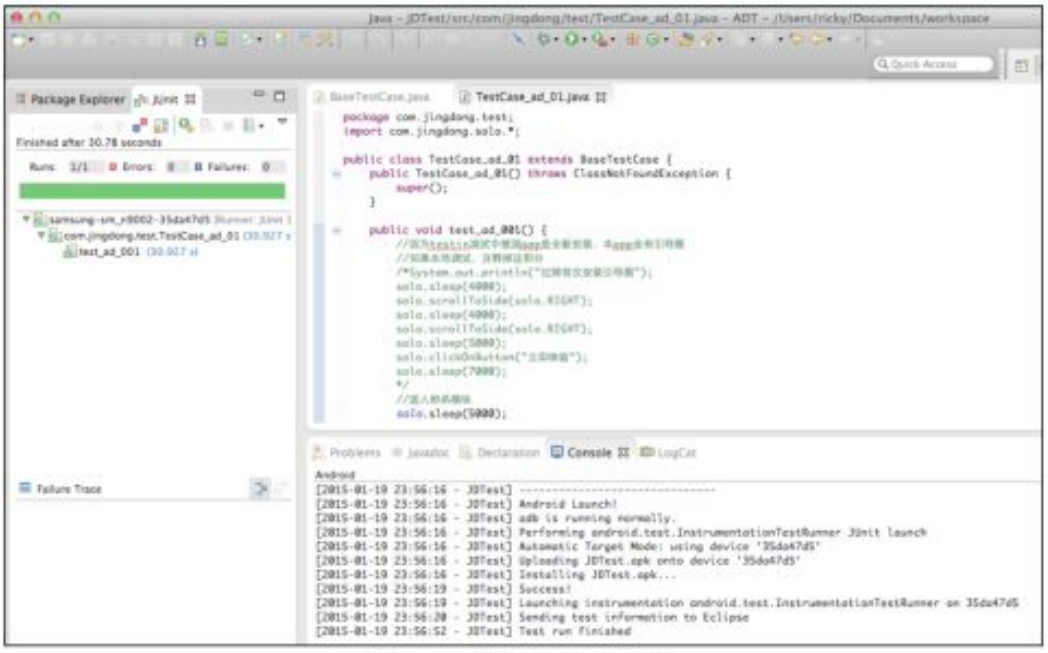


图 4-10 测试脚本执行结果

2. 上传到云平台运行

本地调试完了之后，我们就可以上传到云端来试试了，如图 4-11 所示。

图 4-11　选择自定义脚本兼容性测试

兼容性测试是不带脚本的测试方法，包括安装、启动后随机点击，然后卸载。这里我们有定制的脚本，所以选择功能测试，如图 4-11 所示。

如图 4-12 所示，这里需要上传两个包。上一个 App 安装包是我们的被测 App，debug 签名的。下一个自定义脚本是我们前面编译出来的测试的 apk 文件，在工程的 bin 目录下面，如图 4-13 所示。

选择好文件后上传，如图 4-14 所示。

图 4-12 安装包和脚本上传界面

图 4-13 打包后的测试脚本

图 4-14　上传被测 App 和测试脚本

然后选择想要覆盖的机型，可以从不同维度来选取，如图 4-15 所示。

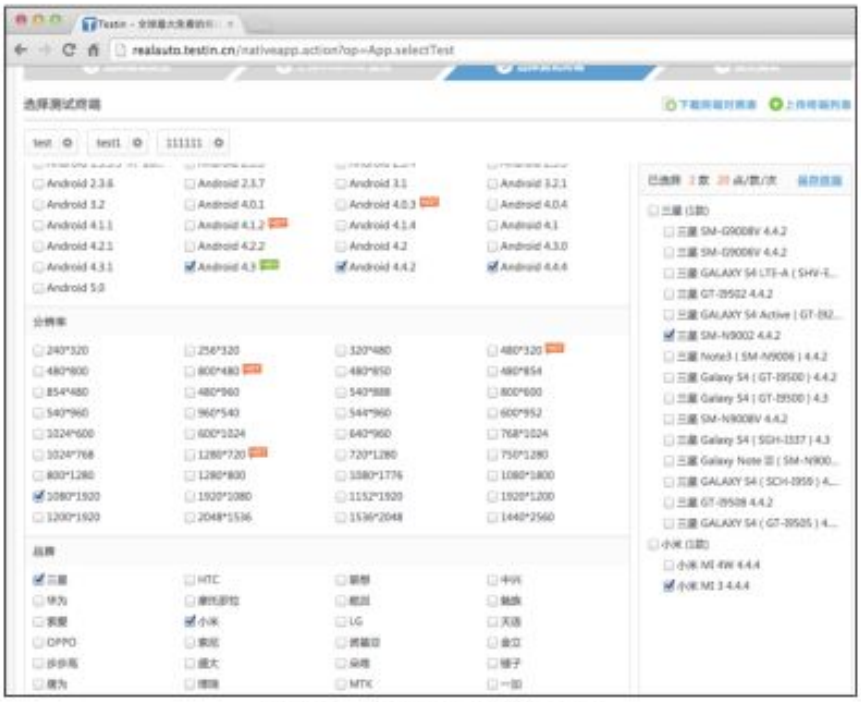


图 4-15　选择需要覆盖的机型

这里选取了两款手机作为示例，实际可以结合项目情况来选取需要的机型。然后提交执行，等待结果。

3. 查看测试报告

待测试结束后可以到平台上查看报告，如图 4-16 和图 4-17 所示。

图 4-16 测试结果概况

图 4-17 性能结果概况

可以查看刚才的脚本在每一台真机上执行后的屏幕截图，截图的时机就是前面代码里面那几个截图语句来设定的。针对 UI 相关的兼容性问题，这个时候可以一个个点击截图来查看，如图 4-18 所示。

图 4-18 测试结果屏幕截图

平台也提供了一些相关的日志，如图 4-19 所示。

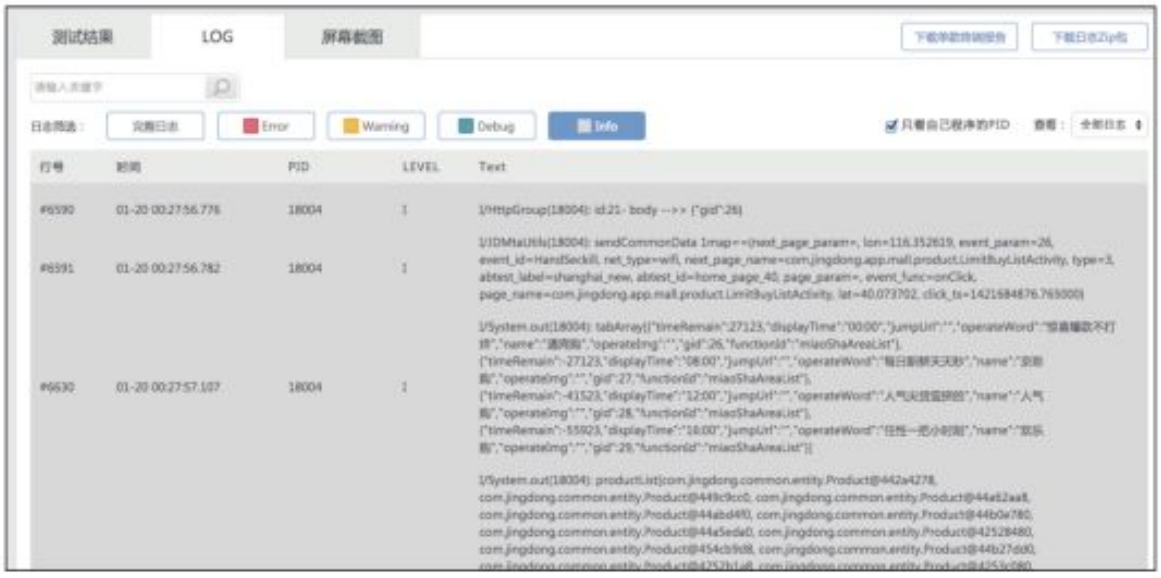


图 4-19 测试过程中的日志

从下载的日志文件中也能看到程序运行过程中发送的 HTTP 请求和接收到的 HTTP 响应的情况，如图 4-20 所示。

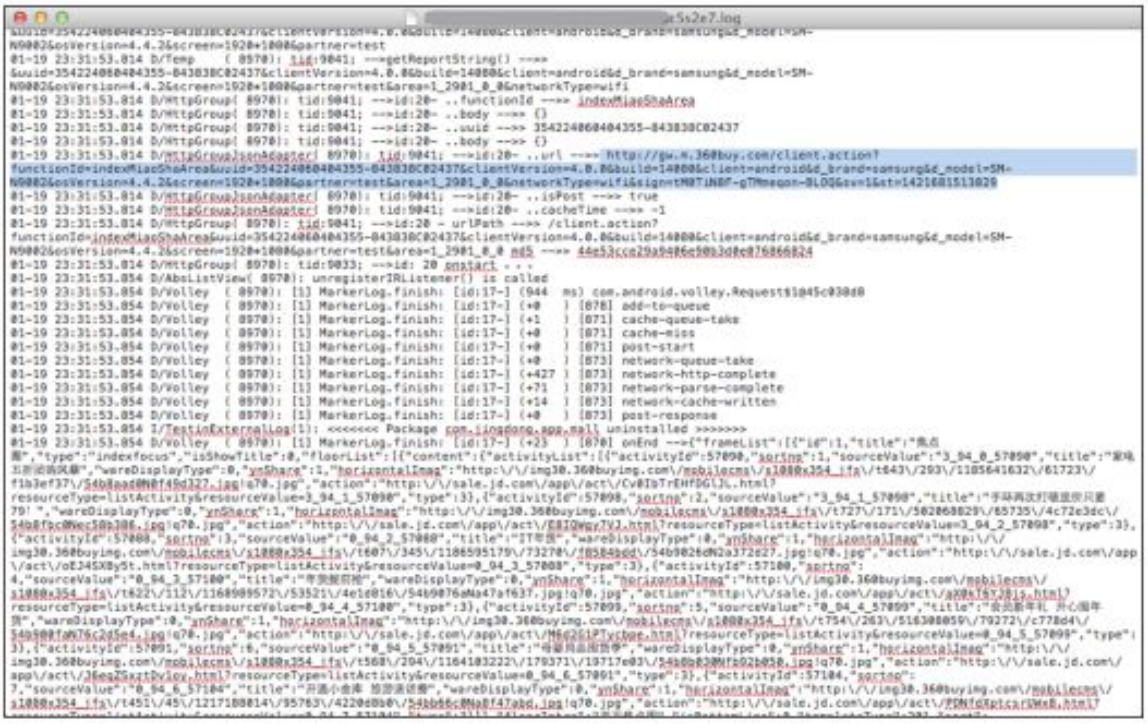


图 4-20 HTTP 请求和响应的日志

总的来说，这个方法的好处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真机资源，将一个流程在多台机器上执行，极大地节省了多平台覆盖的重复人力操作。从以上过程来看，技术门槛不高，但是需要测试团队有基本的 App 开发能力。另外，这样的云测试平台有些是需要付费购买服务的，需要准备一些相关的预算。

4.2 流量测试

移动互联网产品的一个好处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。这使得用户会关心他使用的 App 使用了多少流量，因为这直接关系着资费账单。一些不好的 App 设计，或者缺陷，比如频繁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。

所以在移动产品的测试中，很有必要对 App 使用的流量进行度量。大致来说，流量可以从用户使用的相关性角度来分为两类，一类是用户的操作直接导致的流量消耗，另一类

是后台，即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 系统来说更加容易出现，由于众所周知的原因，目前 Android 的消息推送机制不是借助统一的管道，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，这个部分在后面的章节我们会讨论。

接下来我们讨论如何针对 Android 和 iOS 平台进行流量测试。

4.2.1 Android App 特有的流量测试方法

下面我们介绍一些 Android 平台相关的流量测试方法，常用的技术方案主要有下面几种。

1. 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 系统上的两个文件的内容：

```
proc/uid_stat/{UID} /tcp_snd
proc/uid_stat/{UID} /tcp_rcv
```

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（Byte）为单位，tcp_rcv 表示接收到的数据累计大小。

图 4-21 以一个测试 App 为例展示了如何获取这个数据。操作之前需要将手机通过 USB 线连到 PC 上，然后通过 ADB 命令进入手机的 shell。

以上步骤首先通过 App 的包名获取当前该 App 进程的 PID，上面对应的是 1215，如果无法查看则需要先执行 App。接下来通过访问 /proc/{PID}/status 文件查找该 App 对应的 UID，为 10274。基于这个 UID 我们就可以访问上面提到的两个记录流量数据的文件，其中记录的数字就是对应的流量数据。当我们打开该 App 继续使用的时候，可以看到 tcp_rcv 数据的增长。

这种方法也是很多手机安全软件的做法，能比较简单地给出各个 App 的流量消耗情况。不过这个方法也有一些比较显而易见的局限，那就是只能统计总数据，没有其他维度的细分，不能提供更详细的参考。

2. 通过系统 API 来获取流量数据

上面是通过手工的方法来获取基本的流量数据，其实 Android 也提供了对应的 API 来获取 App 对应的流量数据。

图 4-22 所示的 TrafficStats 类提供了相应的多个方法来获取不同角度的流量数据。

```
root@hlte:/ # ps |grep com.rickyqiu
u0_a274   1215  313   925596 62400 ffffffff 400b66ec S com.rickyqiu.myAndroidPractice
root@hlte:/ #
root@hlte:/ #
root@hlte:/ # cat /proc/1215/status
Name:   AndroidPractice
State:  S (sleeping)
Tgid:   1215
Pid:    1215
PPid:   313
TracerPid:      0
Uid:    10274   10274   10274   10274
Gid:    10274   10274   10274   10274
FDSize: 256
Groups: 1028 3003 50274
VmPeak:   929608 kB
VmSize:   887116 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:     63424 kB
VmRSS:     62400 kB
VmData:    23532 kB
VmStk:       136 kB
VmExe:         8 kB
VmLib:     43584 kB
VmPTE:       204 kB
VmSwap:        0 kB
Threads:        14
SigQ:   1/16877
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000001204
SigIgn: 0000000000000000
SigCgt: 00000002000094e8
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: ffffffff00000000
Cpus_allowed:   f
Cpus_allowed_list:      0-3
voluntary_ctxt_switches:        2718
nonvoluntary_ctxt_switches:     11477
root@hlte:/ #
root@hlte:/ # cat /proc/uid_stat/10274/tcp_snd
4998
root@hlte:/ # cat /proc/uid_stat/10274/tcp_rcv
1784946
root@hlte:/ #
root@hlte:/ # cat /proc/uid_stat/10274/tcp_rcv
2157170
root@hlte:/ #
```

图 4-21 Android 系统自带的流量统计功能

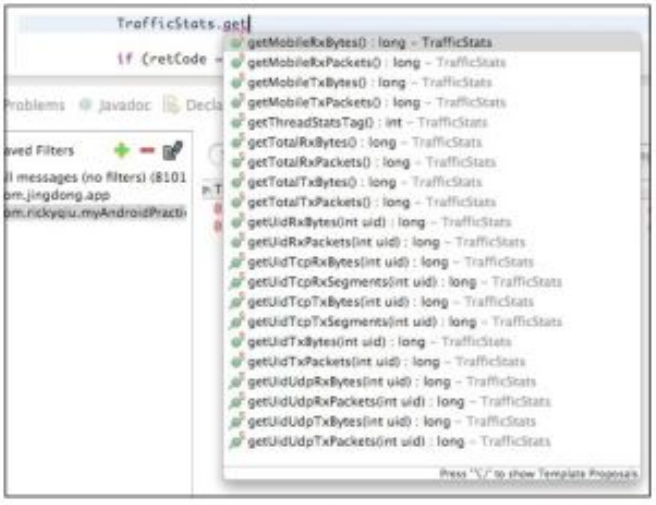


图 4-22 Android 中用于获取 App 流量数据的 API 接口

以图 4-23 所示通过 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。

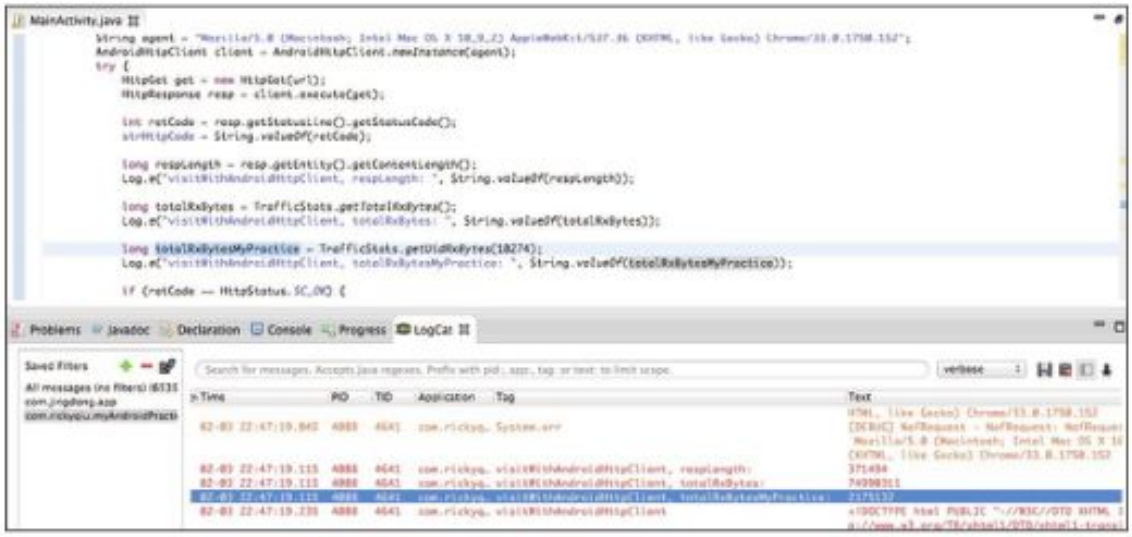


图 4-23　通过 TrafficStats.getUidRxBytes 方法获取该 App 对应流量

3. App 内部通过代码统计接口数据量

除了上面提到的借助 Android 系统的功能来获取流量外，还可以在网络访问的接口代码加上相关的统计功能来获取流量的数据。

图 4-24 所示是一个简单的示例。

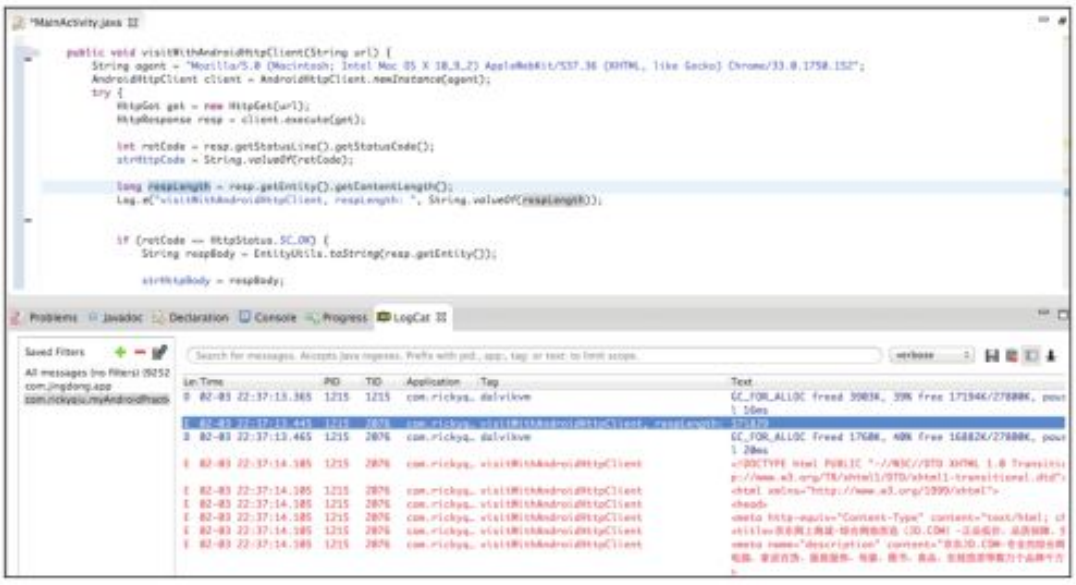


图 4-24　在接口访问代码中加入流量统计功能

以上代码中，resp 是一个 HttpResponse 对象，resp.getEntity().getContengLength() 方法可以获得本次响应的数据量。

实际应用中，很多 App 会将接口访问的功能做一些封装，上层的应用代码可以复用这些库。针对这种情况，可以在接口访问的代码封装中插入流量统计的功能。进一步，这个方法可以详细区分每个接口的访问次数和数据量。但是这种方法也会有一些局限性，主要是一些系统的 DNS 等流量无法统计，以及有些不使用接口封装的模块产生的流量会被遗漏。

4.2.2　iOS App 特有的流量测试方法

下面我们来看 iOS 流量统计的方法，主要有两种。

1. 通过 Instruments 自带的 Network 来查看网络流量

在 Xcode 中选择 Xcode → Open → Developer Tool，打开 Instruments，如图 4-25 所示。

图 4-25　Instrument 启动菜单

选择其中的 Network 功能。

可以按进程方式查看网络流量，如图 4-26 所示。

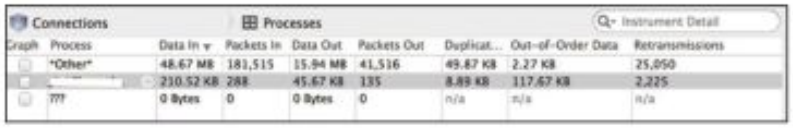
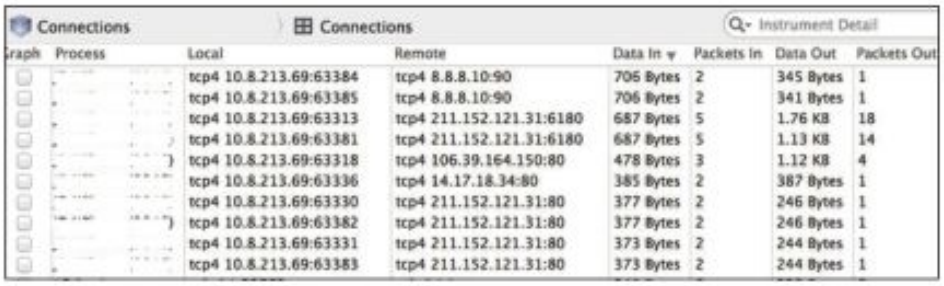


图 4-26　进程维度流量信息

可以看到当前测试 App 的网络吞吐情况、包数量等。大多数情况下，我们都会关注这个纬度的流量结果。

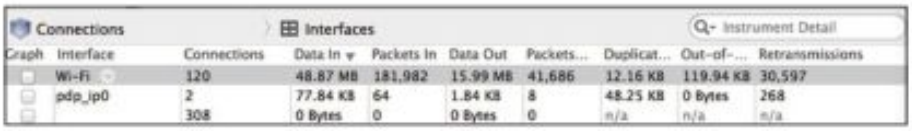
也可以按 TCP 连接来查看不同资源 (IP 地址) 请求的网络流量，如图 4-27 所示。



Graph	Process	Local	Remote	Data In	Packets In	Data Out	Packets Out
		tcp4 10.8.213.69:63384	tcp4 8.8.8.10:90	706 Bytes	2	345 Bytes	1
		tcp4 10.8.213.69:63385	tcp4 8.8.8.10:90	706 Bytes	2	341 Bytes	1
		tcp4 10.8.213.69:63313	tcp4 211.152.121.31:6180	687 Bytes	5	1.76 KB	18
		tcp4 10.8.213.69:63381	tcp4 211.152.121.31:6180	687 Bytes	5	1.13 KB	14
		tcp4 10.8.213.69:63318	tcp4 106.39.164.150:80	478 Bytes	3	1.12 KB	4
		tcp4 10.8.213.69:63336	tcp4 14.17.18.34:80	385 Bytes	2	387 Bytes	1
		tcp4 10.8.213.69:63330	tcp4 211.152.121.31:80	377 Bytes	2	246 Bytes	1
		tcp4 10.8.213.69:63382	tcp4 211.152.121.31:80	377 Bytes	2	246 Bytes	1
		tcp4 10.8.213.69:63331	tcp4 211.152.121.31:80	373 Bytes	2	244 Bytes	1
		tcp4 10.8.213.69:63383	tcp4 211.152.121.31:80	373 Bytes	2	244 Bytes	1

图 4-27　TCP 连接维度查看流量

当然也可以按活动网络接口来查看整体流量，如图 4-28 所示，支持 WiFi 和蜂窝网络接口，不过无法区分出当前的测试应用。



Graph	Interface	Connections	Data In	Packets In	Data Out	Packets...	Duplicat...	Out-of-...	Retransmissions
	Wi-Fi	120	48.87 MB	181,982	15.99 MB	41,686	12.16 KB	119.94 KB	30,597
	pdp_ip0	2	77.84 KB	64	1.84 KB	8	48.25 KB	0 Bytes	268
		308	0 Bytes	0	0 Bytes	0	n/a	n/a	n/a

图 4-28　网络接口维度查看流量

Instruments 给我们提供了基本的网络吞吐量的统计分析手段，可以帮助我们简单地分析应用的网络流量消耗。要注意的一点是，以上的统计都是基于 TCP/IP 连接来统计的，在数据上能做到比 HTTP 流量统计更精确一些，特别适合应用蜂窝数据时统计流量。

2. 代码方式实时统计流量

iOS 系统库提供了获取网络接口流量的方法，这里介绍一下我们自己实现的一个获取手机网络接口流量的功能。相关的代码和 App 本身的代码一起打包，在 App 启动后会以浮窗的形式显示，如图 4-29 所示。

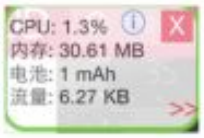


图 4-29　流量消耗实时显示

下面给大家介绍一下其中一个方法的主要核心代码：

```
#import <arpa/inet.h>
#import <net/if.h>
#import <ifaddrs.h>
-(int)getDataCounters
{
```

```
BOOL   success;
struct ifaddrs *addrs;
const struct ifaddrs *cursor;
const struct if_data *networkStatisc;

int WiFiSent = 0;
int WiFiReceived = 0;
int WWANSent = 0;
int WWANReceived = 0;
int output=0;
//此处判断当前网络类型，读者有兴趣可自行查阅资料
NetworkStatus status = [[Reachability reachabilityForInternetConnection]
    currentReachabilityStatus];
if(status==NotReachable){
    return 0;
}else{
    NSString *name=[[[NSString alloc]init]autorelease];
    success = getifaddrs(&addrs) == 0;
    if (success)
    {
        cursor = addrs;
        while (cursor != NULL)
        {
              // 如果是en0 则是 WiFi; 如果是pdp_ip0 ，则是蜂窝网络
            name=[NSString stringWithFormat:@"%s", cursor->ifa_name];
                          //这里我们只查看到网络层
            if (cursor->ifa_addr->sa_family == AF_INET)

            {
                if(status==ReachableViaWiFi){
                    if ([name hasPrefix:@"en0"])
                        {
                        //这里就是获取网络接口进出流量的核心代码
                        networkStatisc = (const struct if_data *) cursor-
                            >ifa_data;
                            //这里需要判断是新统计还是应用老的流量
                            if(original_WifiReceived==0&&original_WifiSent==0){
                            //WiFi发送字节大小
                            original_WifiSent=networkStatisc->ifi_obytes;
                            //WiFi接受字节大小
                            original_WifiReceived=networkStatisc->ifi_ibytes;
                        }else{
                            WiFiSent=networkStatisc->ifi_obytes-original_WifiSent;

                            WiFiReceived=networkStatisc->ifi_ibytes-original_
                                WifiReceived;
                        }
                        //这个输出即当前WiFi流量
                            output=WiFiSent+WiFiReceived;
                    }
                }
```

```
                else if(status==ReachableViaWWAN){
                    if ([name hasPrefix:@"pdp_ip0"])
                        {
                        networkStatisc = (const struct if_data *)
                            cursor->ifa_data;
                            if(original_3GReceived==0&&original_3GSent==0){
                            original_3GSent=networkStatisc->ifi_obytes;
                            original_3GReceived=networkStatisc->ifi_ibytes;
                        }else{
                            WWANSent=networkStatisc->ifi_obytes-
                                original_3GSent;
                            WWANReceived=networkStatisc->ifi_ibytes-
                                original_3GReceived;
                        }
                        output=WWANSent+WWANReceived;
                    }
                }

            }

            cursor = cursor->ifa_next;
        }
        freeifaddrs(addrs);
    }
    return output;
}
```

这个流量统计是网络接口纬度的，支持多个网络接口。但无法区分具体 App 使用了多少，测试的时候需要关闭一切影响因素，包括后台以及其他推送服务。同时该统计方法支持不同网络层面的统计，包括网络层、数据链路层。

4.2.3 通用的流量测试方法

除了上面提到的分别针对 Android 和 iOS 的流量统计方法之外，还有一些比较通用的方法，常用的有两种：在手机上抓包，通过网络代理来统计。下面分别来做一些介绍。

4.2.3.1 手机上抓包

在后台系统的开发和测试中，借助工具抓取网络包（俗称抓包）来进行网络层的分析是一种非常常用的技术手段，常用的抓包工具有 Windows 下的 Wireshark 工具和 Linux 下的 tcpdump。

由于 Android 本质上也是一个 Linux 系统，所以也有对应的 tcpdump 版本。iOS 系统也可以使用 tcpdump 工具来抓包。由于 tcpdump 需要比较高的系统权限来访问底层的网络包，所以对于 Android 系统需要 root 权限，对于 iOS 也需要越狱。下面以 Android 系统为例来看看。

使用一台 root 过的 Android 手机，我们用 adb shell 登录到手机上，并通过 su 命令进

入 root 状态。如果没有 tcpdump 可执行文件请先下载一个，并通过 adb push 上传到手机上。接下来我们就可以进入 tcpdump 所在目录执行这个命令来抓包，如图 4-30 所示。

```
root@hlte:/data #
root@hlte:/data # ./tcpdump -s 0 -w t3.pcap -v
tcpdump: listening on wlan0, link-type EN10MB (Ethernet), capture size 65535 bytes
Got 4283
```

图 4-30　手机上的 tcpdump 抓包

关于 tcpdump 详细的参数和使用方法，这里不做过多介绍，请大家参考命令手册。上面用到的几个参数简要介绍如下：

- -s 0 : 抓取数据包时默认抓取长度为 68 字节。加上 -S 0 后可以抓到完整的数据包。
- -w t1.pcap：表示将抓到的内容存入 t1.pcap 这个文件中。
- -v：表示在命令执行过程中显示当前抓到的包的数量。

tcpdump 还有大量的可选参数，可以设置基于网络类型、协议、IP 和端口等各种过滤条件，来灵活地抓取感兴趣的流量。

抓包结束后，可以将上面的 t1.pcap 文件传到 PC 上，接下来就可以借助 PC 上的 pcap 分析工具来查看和分析，这里用的是 Wireshark 工具。下面看一下简单的分析过程，如图 4-31 所示。

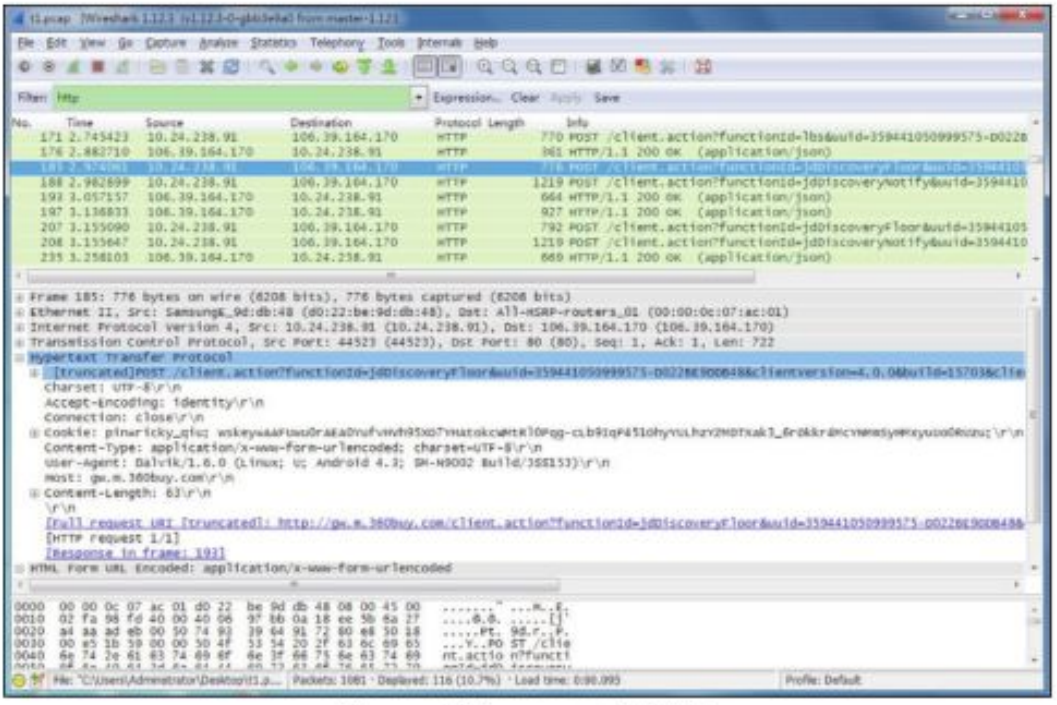


图 4-31　通过 Wireshark 分析抓包

通过 http 过滤词，可以筛选出所有的 HTTP 请求。接下来可以关注某一个连接的整个过程，如图 4-32 所示，查看对应的请求和响应的数据量，对应图中的 Length 字段。

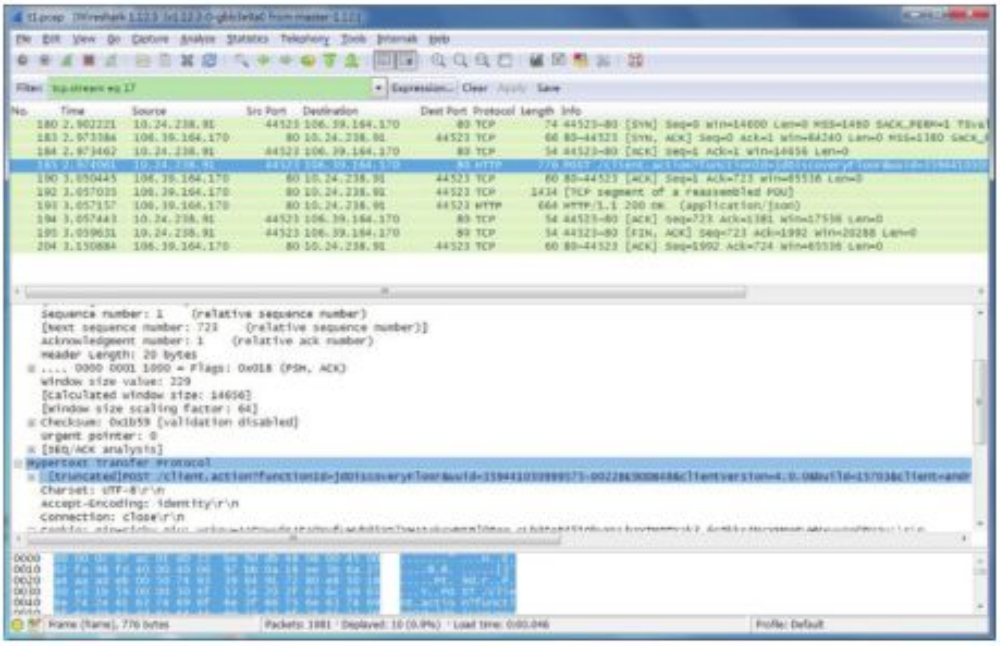


图 4-32　某一条 TCP 流的抓包结果

除了查看单个请求和响应的数据量之外，也可以统计所有请求的数据量，如图 4-33 所示。

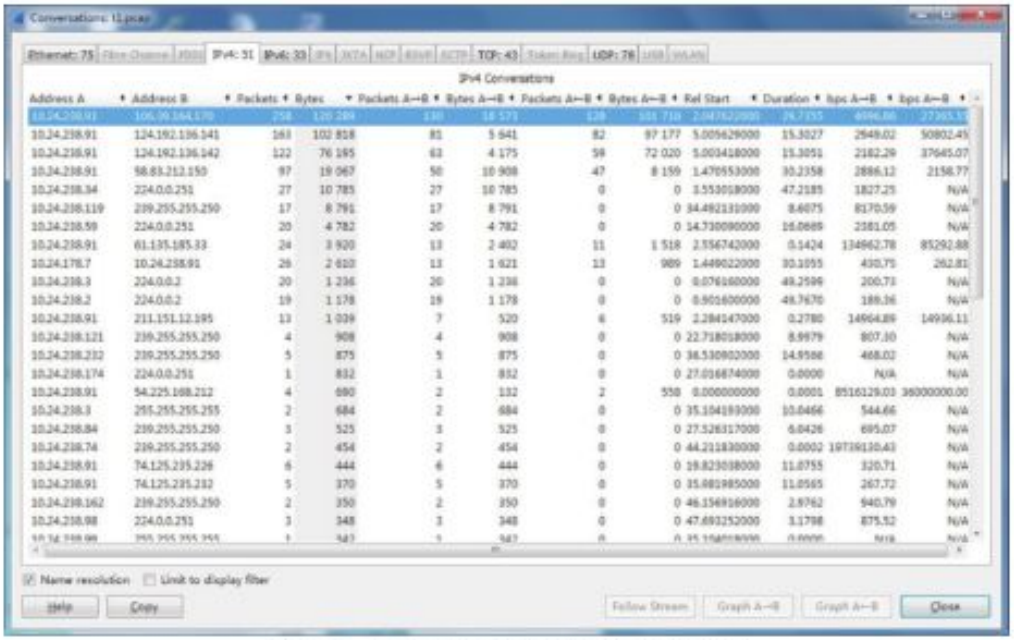


图 4-33　各个 TCP 连接的数据量统计结果

基于 Wireshark 提供的统计分析的功能，从图中可以看到各条 TCP stream 的进出流量情况。如果在抓包时没有做过滤，这里可以通过 IP 和端口来筛选出感兴趣的数据流并统计出对应的流量。

4.2.3.2　基于 WiFi 代理的方式获取流量数据

上面介绍的基于抓包的方法可以统计出数据流量的大小，但是整个操作步骤还是比较繁琐，需要先在手机上抓包，导出抓包结果文件，然后在 PC 上手工分析。实际工作中，还有一种更简便的方法就是通过代理工具来分析，对于以 HTTP 为主要协议的系统来说尤其合适。

图 4-34 和图 4-35 分别展示了在 iOS 和 Android 系统中如何设置代理。

图 4-34　在 iOS 中设置 WiFi 代理

对应的情况里，在 PC 上开启 Fiddler 代理工具，就可以看到手机上发出的请求和收到的响应，如图 4-36 所示。其中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。

图 4-35　在 Android 中设置 WiFi 代理

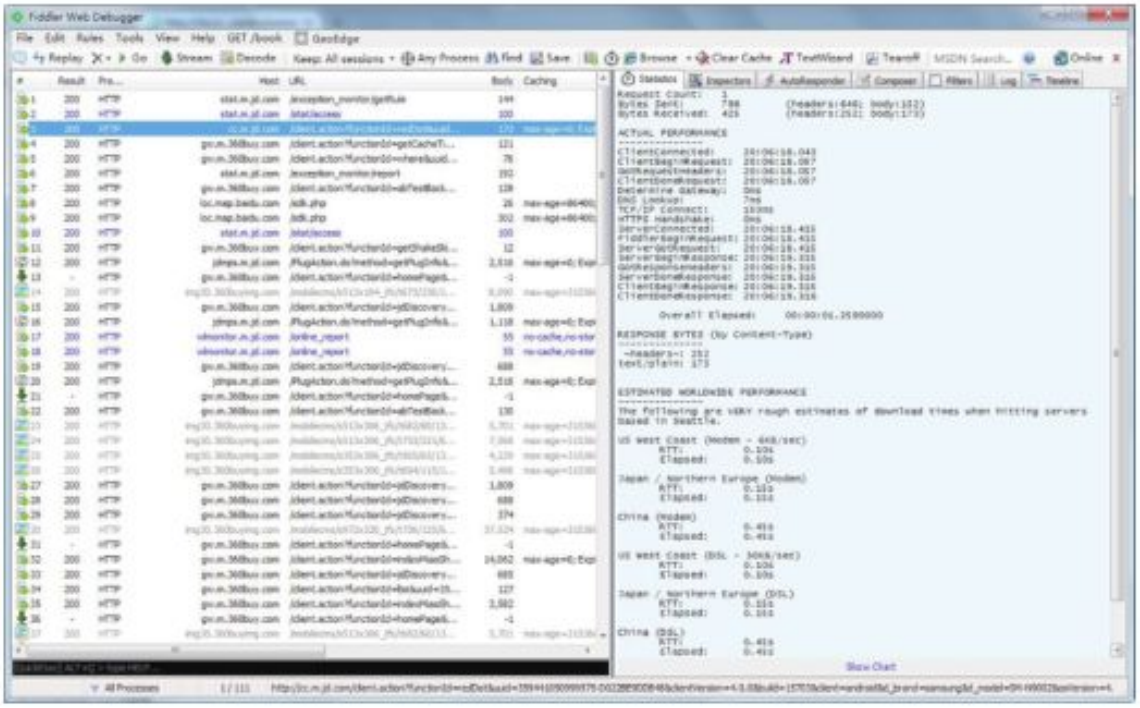


图 4-36　通过 Fiddler 分析请求流量

对应的情况是，在 Mac 上可以使用 Charles 作为代理，以及请求 / 响应显示的工具，如下图 4-37 所示。

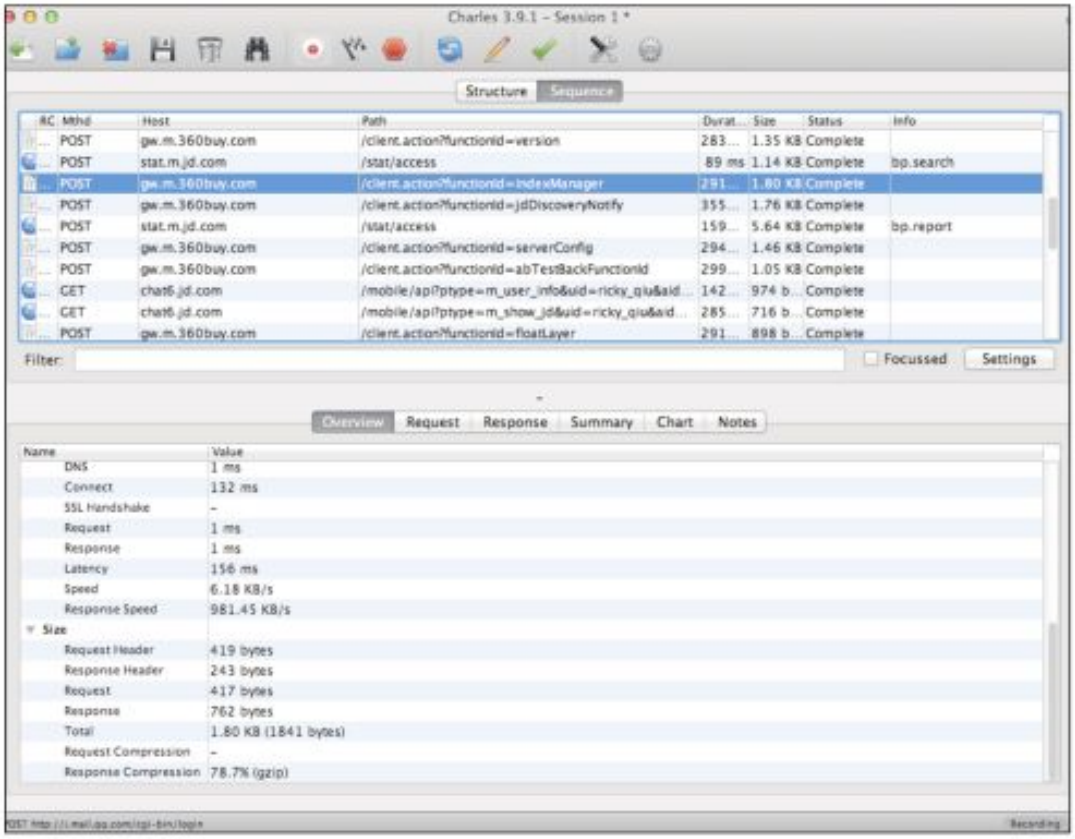


图 4-37　Mac 下的 Charles 分析工具

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。和抓包结果的对比分析，粗略估算下来，大概少算 10% 左右。当然，这个也取决于平均包大小的情况，如果普遍都是小包，那么协议 header 的占比就会比较高。

4.2.3.3　自动化的流量统计方案

在上面通过代理工具来获取流量数据的基础上，我们实践了一个自动化的方法，来让流量数据的获取和分析更加简便高效。主要的思路是基于 FiddlerCode 的二次开发。

我们通常会用 Fiddler 工具进行 HTTP 抓包并进行手动分析。其中的分析步骤大同小异，无非是将 Fiddler 抓包的原始数据导入一些数据处理工具中（例如 Excel），然后进行一些过滤、分组、排序等操作。最后根据处理后的数据分析流量瓶颈和优化点。下面列出了手动分析通常要做的事情：

- 按域名分组
- 按类型分组（图片、接口等）
- 按缓存与否分组
- 过滤域名
- 统计上下行流量大小
- 统计请求个数

像上述这些重复性的手动工作我们完全可以自动化完成，我们尝试通过基于 FiddlerCore 进行开发（关于 FiddlerCore 的详情请参考 5.3.3 节中关于 Mock Server 的部分）。下面是我们测试团队开发的基于 FiddlerCore 的自动化分析工具。图 4-38 是基于域名的自动化分析结果。

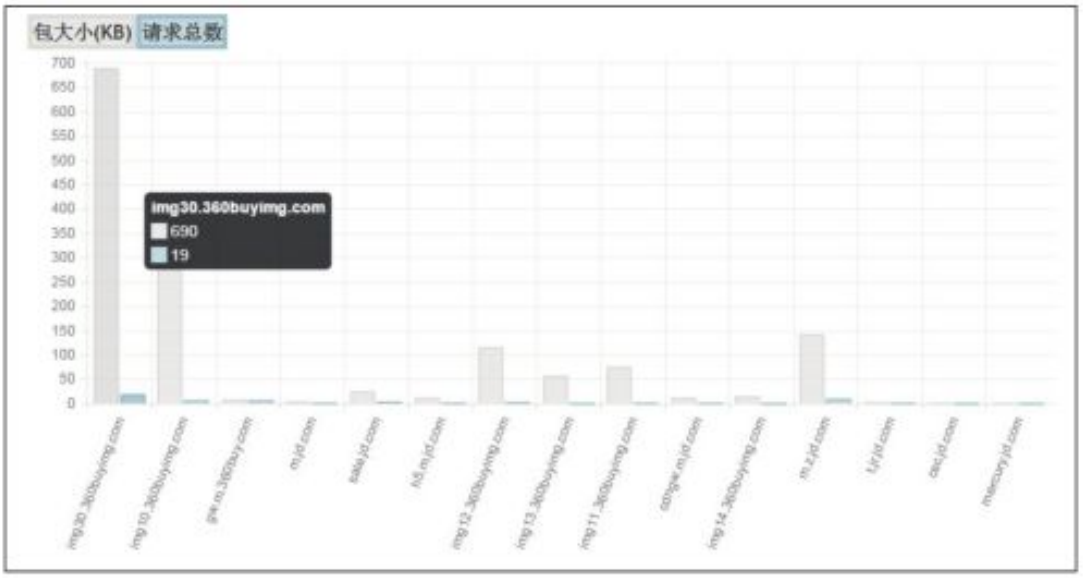


图 4-38　基于域名的流量自动分析结果

图 4-39 是基于数据类型的分析结果。

图 4-39　基于数据类型的分析结果

使用上述工具后，在测试中，我们只要在测试开始和结束时通知统计工具即可，最后的报告会在几秒内自动生成。该报告可以互动，例如点击上图中的“未缓存”部分会展示所有未缓存的图片会话，等等。这样能大大提高流量测试的效率，并能快速定位流量瓶颈，发现优化点。在实际使用中，我们利用这个工具发现了不少后台接口处理不当引起的流量问题，例如图片未缓存等。

除了可视化报告外，我们也可以导出原始数据以供其他用途使用（例如存储在数据库中追踪接口流量优化情况，等等）。

一旦有新的数据关注点，我们也可以通过修改统计工具一次性进行扩展工作，对测试人员来说，流量分析的操作方式依然保持不变。

值得一提的是，一些 App 的后台接口在 2G、3G、4G 网络下会返回不同的内容。此场景的流量测试我们无法通过 WiFi 连 Fiddler 的方式来进行。对于此场景，需要使用前面介绍过的 tcpdump 工具抓取网络包，之后可以用代码解析 tcpdump 所抓取网络包的数据，从中找出 HTTP 包并进行自动分析。

4.2.4 常见的流量节省方法

以上讨论了多种流量测试的方法，这里介绍一些常见的节省流量的方法。主要包含以下几个方面。

1. 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，以及图片的压缩。基于一些压缩算法，可以在 JPG、PNG 等文件格式的基础上进一步降低图片大小而图片质量不会明显下降。

2. 不同数据格式的采用

在传输相同信息的情况下，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 格式要小。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，特别是针对图片数据比较多的 App 而言，相比 JPG、PNG 格式，在同等尺寸和画质的情况下，WebP 格式的文件大小有很大的优势。这里有一篇 WebP 相关的对比分析和测试文章可以参考：http://isux.tencent.com/introduction-of-webp.html

针对 WebP，应用中需要考虑 App 和 Server 端的支持。对于 App 端，Android 4.0 以上的 BitmapFactory 可以直接支持，对于 4.0 以下版本，需要引入相关的 jar 包来支持，iOS 需

要引入相应的第三方库来支持。在服务端，需要有转换工具将之前的其他格式图片转换成 WebP 格式。

3. 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，这个主要是针对后台数据上报，PUSH 消息检查等定时机制的。

4. 只获取必要的数据

很多时候，我们会遇到 App 一页的内容非常多的情况，而用户可能只会查看一部分，过多地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。

5. 缓存

缓存也是一个非常常用且有效的方法，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。

6. 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 场景下，流量和带宽都不是问题，这时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等。

图 4-40 所示是一个实际 App 中的例子，打开这个开关后，在 2G/3G 网络情况下，默认不加载商品图片等大的图，而需要用户点击后按需下载，从而达到基本功能可用但是节省流量的目的。

图 4-40　关于节省流量的 App 设置

4.3 电量测试

对于移动设备而言，电量也是一个用户非常关注的方面。在电池技术没有巨大突破的前提下，这方面始终会存在一些瓶颈，如果一些 App 架构的设计不好，或者代码有缺陷，就可能会导致电量消耗比较高。当用户发现电池消耗过快的时候，就可能会去查看哪些应用消耗电量过多，消耗排名比较高且有替代品的 App 就很有可能被用户卸载。在当前移动互联网产品竞争比较激烈的情况下，争取新安装用户通常需要投入很多人力和物力，所以如果因为电量消耗过多的原因被用户卸载，那是非常遗憾的事情。为了避免这样的情况，我们需要针对电量消耗做一些针对性的测试。以下我们分别就 Android 和 iOS 的测试方法来做一些介绍。

4.3.1 Android 电量测试方法

这里我们介绍两种测试方法，第一种方法是基于硬件的测试方案，可以比较精确地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。

1. 基于硬件设备的方法

这个方法的基本做法是将手机的电池拿掉，并将电量表串接在手机和外部电源之间。这样手机的供电就会经过电量表，可以获得比较准确的电量数据，做法如图 4-41 所示。

图 4-41 借助电量表来测试手机电量消耗

为了便于数据的收集和分析，可以开发一个上位机软件，和电量表直接通信，在手机上的 App 使用过程中同步上传电量消耗情况，并在 PC 上显示。图 4-42 所示是基于以上测试方案收集到的电量数据。

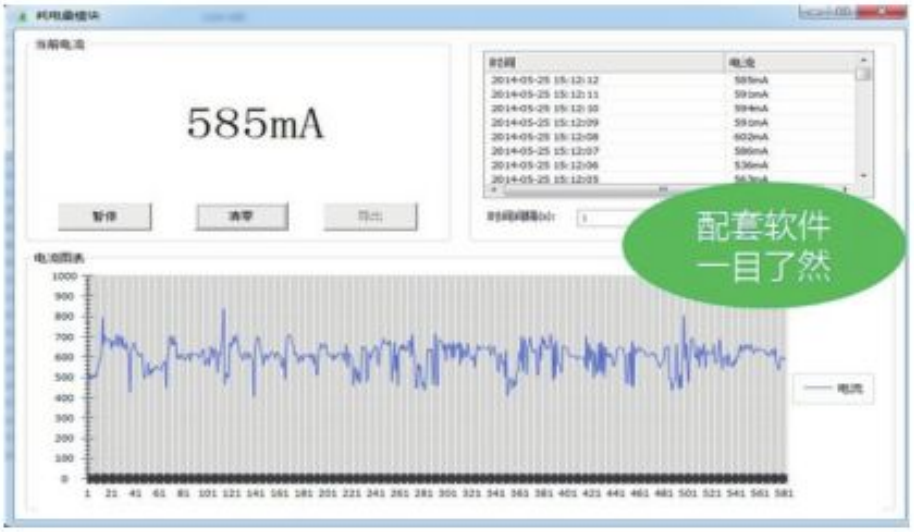


图 4-42 基于硬件电量测试设备的数据收集和展示⊖

通过这样的一套方案，可以在 App 手工测试或者自动化测试的过程中，同步采集电量消耗的信息，并和 CPU/ 内存等系统资源信息一起进行用户问题分析。

2. 基于 GSam Battery Monitor Pro 查看电量消耗

上面基于硬件的测试方法给出了具体的电量消耗数值，比较精确，但是实际使用中有几个问题需要进一步了解：

- 上面电量消耗的数值是整个手机的，而不只是被测 App 的。所以需要测试时尽量减少其他系统本身和其他 App 的干扰。
- 单独看这个数字其实无法得到直接的参考，不知道这个数值是偏高还是合理，可能需要一些对比参考。
- 不能给出更进一步的参考，比如具体某个 App 哪些方面消耗电量比较多。

针对这些问题，我们找到一个更加便捷的测试方法，直接借着 GSam Battery Monitor Pro（以下简称 GSam）这个工具，如图 4-43 所示。当然，也可以使用其他类似的工具。

电量是一个物理的概念，其消耗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。在上面 GSam 工具中，将一些不便于分摊到应用 App 的作为公共部分。可以分摊的部分会计入各个 App，然后计算在这一段时间内各个应用电量消耗的排名。图 4-44 是 GSam 工具展示的各个 App 的电量消耗情况。

⊖ 以上两张图片来自腾讯 WeTest 实验室，经同意授权使用，图片版权归 WeTest 所有，在此表示感谢。WeTest 是一个专注于游戏质量的一站式服务平台（http://wetest.qq.com/）。

图 4-43　GSam Battery Monitor Pro 测试工具

图 4-44　各个应用的电量消耗情况

在点击某一个 App 之后，可以看到该应用电量消耗的详情。下面以微信电话本为例，查看主要的电量消耗原因，如图 4-45 所示。

图 4-45 具体应用的电量消耗情况

从应用的角度，这里列出了 CPU 使用、网络流量、感应器，以及唤醒锁和保持唤醒的情况。从这些信息可以初步判断 App 的电量消耗的分布。对于图中提到的 CPU 使用和光线感应器比较容易理解，对唤醒锁这里我们稍作讨论。这里涉及两个 Android 概念，一个是 AlarmManager，另一个是 WakeLock。

AlarmManager 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 上这通常是用一个服务定时向服务器发起请求来实现的。从流量角度，这也是为什么手机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。AlarmManager 机制也可以用来定期执行 App 里面某个功能来实现到服务端的轮询。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，更好的方案还是选用 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 就是为这样的需求而产生的一种机制。

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，主要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。

以上是一个简单的介绍，关于 Android 后台运行和唤醒机制可以参考相关的文档。从以上介绍我们可以看出，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。而目前情况，Android 平台对应用并无严格的审核，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。

4.3.2 iOS 电量测试方法

针对 iOS 的电量测试，Instrument 提供了一套 Energy 工具来查看 App 的耗电情况，它能够提供多个纬度来查看分析 App 的耗电源头。

1）在 Xcode 中选择真机调试，设置 debug 下的开发者证书和 provision 文件，修改 scheme，如图 4-46 所示。

图 4-46 选择 Scheme

2）修改 Profile 的编译选项为 Debug，点击关闭，如图 4-47 所示。

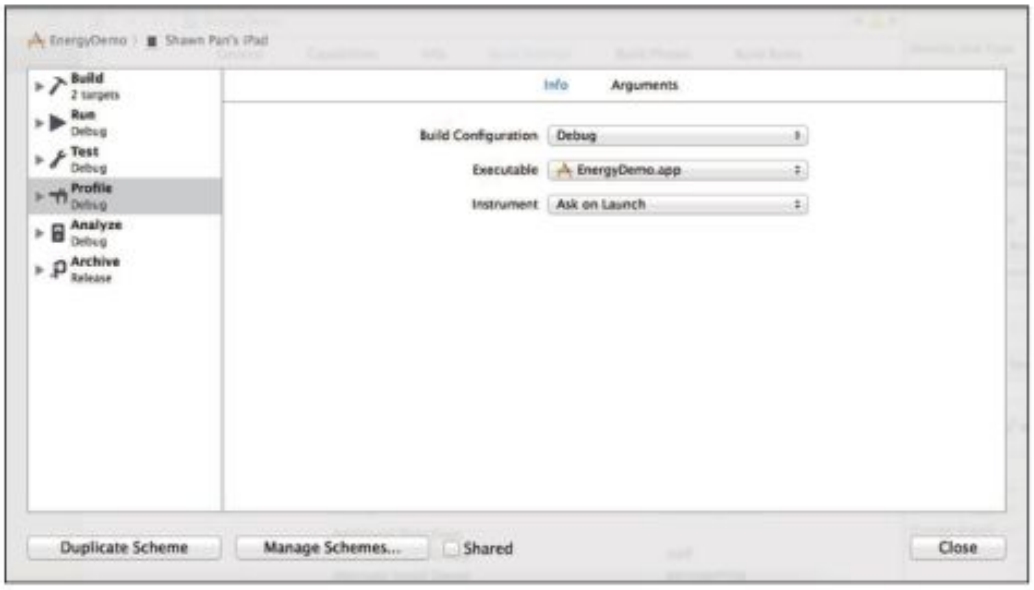


图 4-47　设置 Profile 的编译模式

3）点击 Profile，编译安装项目工程到设备，如图 4-48 所示。

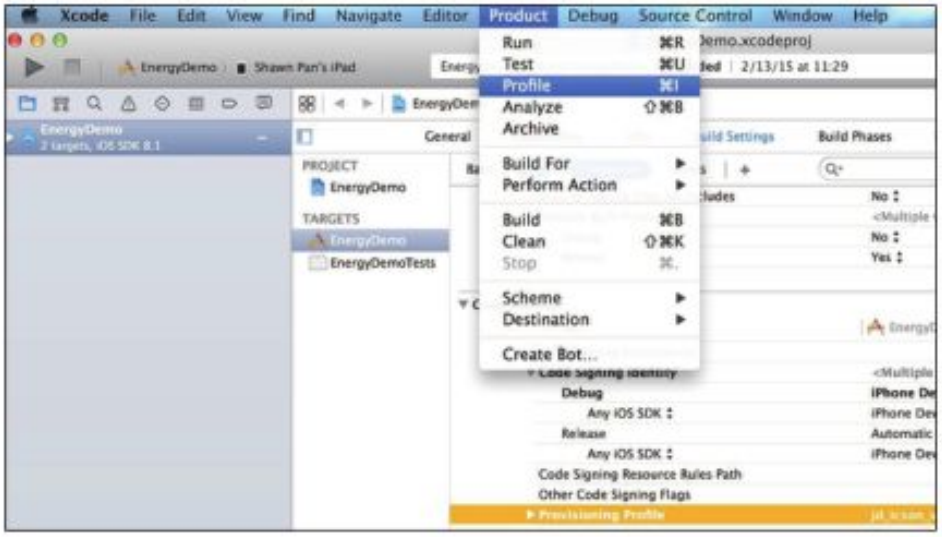


图 4-48　选择 Profile 模式编译

4）等待安装完成后，弹出 Instrument 对话框，选择 Energy Diagnostics，如图 4-49 所示。

图 4-49 Instrument 主菜单

5）点击上方的录制按钮，就可以启动 App 开始测试了，如图 4-50 所示。

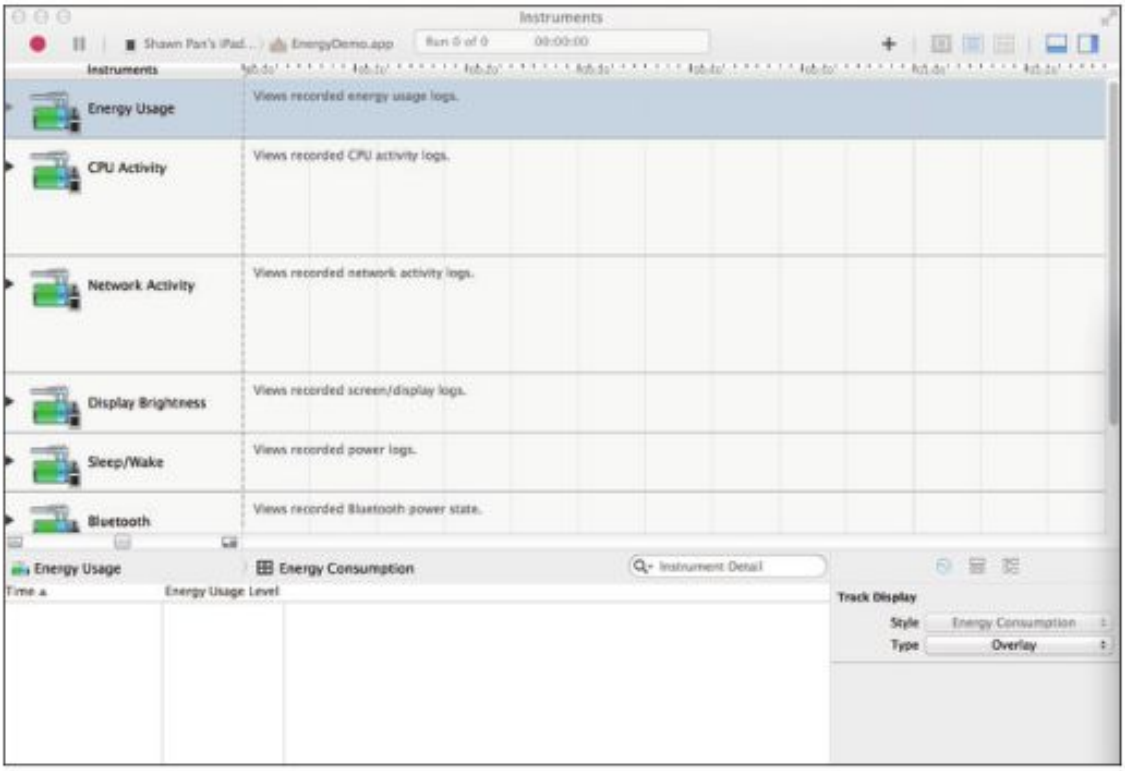


图 4-50 Energy 界面

6）电量消耗分析

A.Energy Usage

当前设备的电量消耗情况以时间间隔纬度显示，该刻度是苹果定制的，电量消耗刻度在 0 ~ 20 之间，每个刻度指定不同的消耗情况，从小到大，逐步增加，如图 4-51 所示。比如如果在某个时间段的电量消耗刻度为 19，说明当前的电量消耗很大。如果在某个时间段的电量消耗刻度为 1，说明当前电量消耗很少。由于该刻度没有具体的电量值参考，所以也无法判断这个刻度的具体消耗值是多少。

Energy Usage › Energy Consumption

Time ▲	Energy Usage Level
00:00 - 00:00	1/20
00:00 - 00:00	1/20
00:00 - 00:00	1/20
00:00 - 00:00	12/20
00:00 - 00:00.755	14/20
00:00.755 - 00:01.755	13/20
00:01.755 - 00:02.755	14/20
00:02.755 - 00:03.755	13/20
00:03.755 - 00:04.755	13/20
00:04.755 - 00:05.399	12/20
00:05.399 - 00:06.399	14/20
00:06.399 - 00:07.399	14/20
00:07.399 - 00:08.399	15/20
00:08.399 - 00:09.399	13/20
00:09.399 - 00:10.393	12/20
00:10.393 - 00:11.393	12/20
00:11.393 - 00:12.393	13/20
00:12.393 - 00:13.393	12/20
00:13.393 - 00:14.393	12/20
00:14.393 - 00:15.395	12/20

图 4-51　电量总体消耗

B. CPU Activity

如图 4-52 所示，以间隔时间为采样，记录了当前耗电量的各项 CPU 占比，包括设备总体活动百分比、当前活动 App、声音处理、图像渲染。

CPU Activity › CPU Activity　　Instrument Detail

Time ▲	Total Activity	Foreground App Activity	Audio Processing	Graphics
00:00 - 00:02.218	92.2%	37.6%	0%	16.5%
00:02.218 - 00:07.602	84%	43.2%	0%	21.7%
00:07.602 - 00:09.077	72.3%	13.2%	0%	33.8%
00:09.077 - 00:11.202	65.6%	2.2%	0%	15.6%
00:11.202 - 00:14.209	59.1%	6.4%	0%	20.9%
00:14.209 - 00:17.635	71.2%	29.7%	1.1%	34.3%
00:17.635 - 00:20.202	82%	27.6%	0%	15.8%
00:20.202 - 00:23.206	69.7%	29.4%	0%	24.6%
00:23.206 - 00:26.212	86.4%	31%	0%	33.8%
00:26.212 - 00:29.202	85.1%	28.5%	0%	35.8%
00:29.202 - 00:32.201	59.2%	2.1%	0%	10%
00:32.201 - 00:35.202	53%	0.7%	0%	4.7%
00:35.202 - 00:38.202	60.2%	9.1%	0%	14.4%
00:38.202 - 00:41.214	70.3%	26.3%	0%	29.3%
00:41.214 - 00:44.202	80.5%	35%	0%	13.2%
00:44.202 - 00:47.203	41.5%	0.4%	0%	4%
00:47.203 - 00:50.550	67.5%	35.2%	0%	15%
00:50.550 - 00:53.218	92.3%	55.7%	0%	27.9%

图 4-52　CPU 消耗

C. Network Activity

如图 4-53 所示，以间隔时间为采样，记录当前网络吞吐量，包括 WiFi 下的入包、出包数量，出口和入口字节大小，出口和入口错误数，同时支持蜂窝流量数据。

Network Activity 〉 Network Activity　　Instrument Detail

Time ▲	Wi-Fi Packets (in)	Wi-Fi Packets (out)	Wi-Fi Bytes (in)	Wi-Fi Bytes (out)	Wi-Fi Errors (in)	Wi-Fi Errors (out)	Co
00:00 – 00:00.229	9	0	4721	1056	7	0	0
00:00.229 – 00:01.231	37	0	37226	4219	8	0	0
00:01.231 – 00:02.252	18	0	1991	2830	11	0	0
00:02.252 – 00:07.323	145	0	98109	16798	41	0	0
00:07.323 – 00:09.101	9	0	865	1074	5	0	0
00:09.101 – 00:09.389	0	0	0	156	0	0	0
00:09.388 – 00:10.225	8	0	5928	1022	8	0	0
00:10.225 – 00:11.227	3	0	430	346	1	0	0
00:11.227 – 00:12.225	6	0	1449	0	1	0	0
00:12.225 – 00:13.226	8	0	4616	990	2	0	0
00:13.226 – 00:14.233	10	0	5645	1947	3	0	0
00:14.233 – 00:15.225	12	0	1360	1026	6	0	0
00:15.225 – 00:16.226	10	0	2648	975	5	0	0
00:16.226 – 00:17.659	10	0	4788	1392	5	0	0
00:17.659 – 00:18.240	27	0	12524	3761	9	0	0
00:18.240 – 00:19.315	135	0	122198	12123	29	0	0
00:19.315 – 00:20.226	28	0	8430	708	2	0	0
00:20.226 – 00:21.226	21	0	5524	1200	3	0	0

图 4-53　网络流量消耗

D. Display Brightness

如图 4-54 所示，以间隔时间为采样，获取设备的亮度百分比。

Display Brightness 〉 Brightness

Time ▲	Brightness Level
00:00 – 00:08.277	75%
00:08.277 – 00:08.301	70%
00:08.301 – 00:08.314	68%
00:08.314 – 00:08.331	65%
00:08.331 – 00:08.345	62%
00:08.345 – 00:08.362	59%
00:08.362 – 00:08.389	56%
00:08.389 – 00:08.397	53%
00:08.397 – 00:08.414	51%
00:08.414 – 00:08.466	47%
00:08.466 – 00:08.482	40%
00:08.482 – 00:08.502	38%
00:08.502 – 00:08.517	35%
00:08.517 – 00:08.528	33%
00:08.528 – 00:08.548	31%
00:08.548 – 00:08.564	30%
00:08.564 – 00:08.584	28%
00:08.584 – 00:08.612	27%

图 4-54　屏幕亮度

E. Sleep/Wake

如图 4-55 所示，以时间间隔为采样，获取到的设备休眠和唤醒状态。

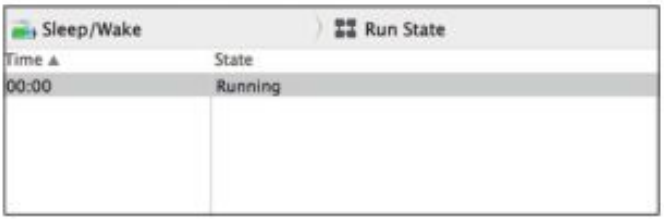


图 4-55 运行状态

F. Bluetooth

图 4-56 所示是以间隔时间为采样，获取到的设备蓝牙连接状态。这里只有 On/Off 两种状态。

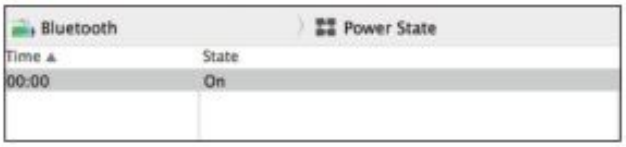


图 4-56 蓝牙状态

G. Wi-Fi

图 4-57 是以间隔时间为采样，获取到的设备无线网络连接状态，这里只有 On/Off 两种状态。

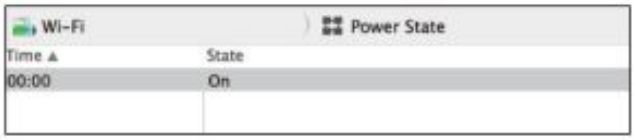


图 4-57 WiFi 状态

H. GPS

图 4-58 所示是以间隔时间为采样，获取设备 GPS 的运行状态，这里只有 On/Off 两种状态。

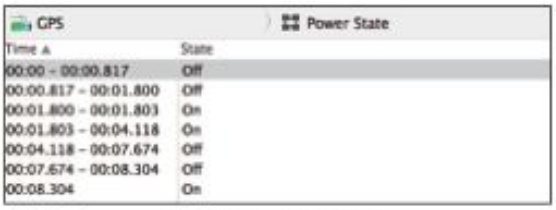


图 4-58 GPS 状态

我们可以通过捕获在 Energy Usage 中刻度较大的时间段，去分析该时间段其他指标的运行情况，比如网络流量、图像渲染等，找到耗电量大的根源。一般情况下，耗电量大头主要集中在 CPU 的高频率运算、GPU 的高频率运算、蜂窝网络芯片的持续运算等。所以，我们会发现 App 有扫码功能时，耗电量特别严重。玩大型游戏时，图像的渲染导致手机掉电特别严重。在使用 3G/4G 网络上网时，手机耗电也会特别严重。

通过苹果提供的类库获取当前设备电量信息

如果你想获取详细的关于电量值的信息，可以使用苹果提供的关于电量信息的类库，不过由于该库是属于苹果系统框架，所以如果你的 App 用到这部分可能无法通过 App Store 审核。而且苹果在 iOS8 中去除了该类库，所以我们只能在 iOS7 和 iOS6 上使用它。

```
iOS7:
/System/Library/PrivateFrameworks/PowerlogLoggerSupport.framework/
PLBatteryPropertiesEntry
iOS6:
/System/Library/PrivateFrameworks/GAIA.framework/OSDBattery
```

获取设备当前电量，单位是毫安时：

```
-(int)currentBattery{
      NSString* version=[[UIDevice currentDevice] systemVersion];
      NSString* value;
      id entry=[self BatteryInfo];
      if([version hasPrefix:@"6"]){
          value=[NSString stringWithFormat:@"%@", [entry _getBattery-
              CurrentCapacity]];
      }
      value=[NSString stringWithFormat:@"%@", [entry objectForKey:@"current_
          capacity"]];
      return value;
}
```

获取设备电池最大电量，单位毫安时：

```
-(int)BatteryMaxCapacity{
      NSString* version=[[UIDevice currentDevice] systemVersion];
      id entry=[self BatteryInfo];
      if([version hasPrefix:@"6"]){
          return [[NSString stringWithFormat:@"%@", [entry _
              getBatteryMaxCapacity]] intValue];;
      }
      return [[NSString stringWithFormat:@"%@", [entry objectForKey:@"max_
          capacity"]] intValue];
}
```

然后我们就可以通过比较两次当前电量的差，来获得设备耗电量的具体数值了。不过由于该方法是设备纬度的，无法精确应用于测试 App、后台服务，其他应用的网络流量都

可能是耗电来源。建议关闭后台所有的其他程序和消息通知，只留一个测试应用在前台运行，让系统保持一个比较干净的环境来测试，以此获得比较精确的数据。

4.4 弱网络测试

移动互联网产品相比 PC 互联网产品，有一个特点是前者使用的网络比较多样，除了 WiFi 很多时候是在移动网络下使用，而且随着移动网络的换代和普及，使用移动网络的情况会越来越多。移动网络遇到的情况比较复杂，比较容易遇到信号不好的地方（建筑物或者隧道），以及基站间切换，或者在体育场等人员密集场所基站容量跟不上的情况，所以网络不稳定的情况是比较容易发生的。就我们自己个人的体验和收到的用户反馈来看，很多时候，App 的一些问题是在复杂网络情况下才会暴露。与其让发布后用户遇到相关的问题而投诉，不如我们在测试阶段就尽量模拟这样的弱网络情况，及早发现和修复问题，提升产品的使用体验。下面我们探讨有哪些方案来帮助我们模拟弱网络情况。

4.4.1 借助手机自带的网络状况模拟工具

目前来看这个方案主要针对 iOS，对 Android 的支持还不是很好。

首先进入 iOS 系统中自带的开发者（Developer）选项。需要注意的是，只有用于调试过的真机才能看到这个选项，如图 4-59 所示，默认是隐藏的。

图 4-59 iOS 设置中的开发者选项

点进去之后能看到相关的具体设置，其中有一项叫做“NETWORK LINK CONDITIONER”，如图 4-60 所示，默认是关闭的状态。

点这个选项进去之后能看到具体的网络类型设置，如图 4-61 所示。默认提供了几个常用 profile 供选择。点 i 图标进去之后能看到具体的参数设置，关于进 / 出的带宽和延迟，以及丢包和 DNS 延迟，也可以选择规则适用的协议类型。

图 4-60 开发者选项具体内容

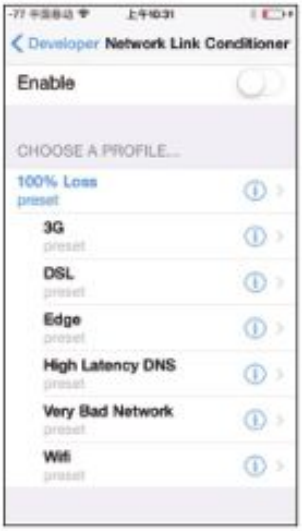


图 4-61 模拟网络类型选项

除了自带的几种网络状况之外，我们还可以选择最下面的“ Add a profile…”，自定义我们想要的网络状况，如图 4-62 所示。

以上参数可以参考移动网络，比如 2G/3G/4G 的一些常见指标，也可以基于之前真实网络抓包分析的结果，或者也可以跟进项目需要自定义设定。

在选好对应的 profile 之后，就可以选择 Enable 使其生效，如图 4-63 所示。

图 4-62 自定义网络状况

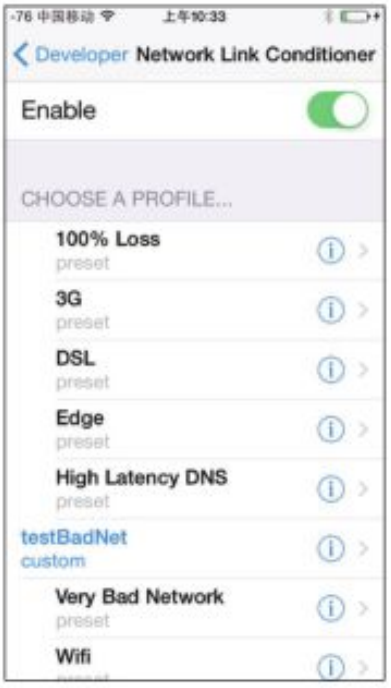


图 4-63 启用自定义网络状况设置

接下来就可以在这个手机上打开相应的 App 进行弱网络情况下的功能测试。

4.4.2 基于代理的弱网络的模拟

除了上面提到的 iOS 系统可以借助自带的网络状况工具来构造测试环境，我们还可以采用另一个更通用的做法，那就是通过网络代理的方式。主要的思路如下，手机和 PC/Mac 电脑都连接同一个 WiFi，在电脑上开启代理软件，然后修改手机上的网络设置，将代理指向电脑上对应的代理的 IP 和端口。这种情况下，由于手机流量经过电脑，电脑上的网络状况模拟就会影响到实际的手机网络。这种方式的另一个类似做法就是在电脑上通过双网卡的方式自行搭建一个 WiFi 热点，让手机直接连接这个热点。

这方面的工具非常多，当然也可以自行开发。下面就我们实际使用过的两个工具来讲解下，分别适用于 Windows 和 Mac 平台。将网络模拟工具跑在 PC/Mac 上，并打开代理软件，比如前面提到的 Fiddler 或者 Charles，然后手机连接同一个 WiFi 热点，并在 WiFi 的设置里把代理指向对应的 PC/Mac。

1. Windows 下的 Network Delay Simulator

首先我们介绍 Windows 下面用到的一个网络状况模拟工具：Network Delay Simulator。这是一个免费工具，可以从网络上下载。它的基本原理是安装时会添加一个新的网络驱动，进而控制网络状况。

其使用界面比较简单，如图 4-64 所示。

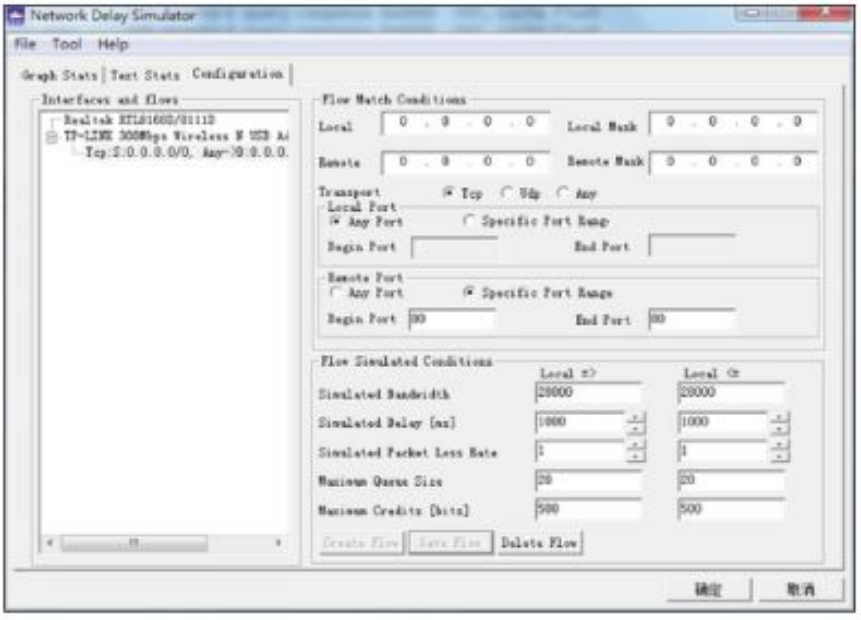


图 4-64 Network Delay Simulator 主界面

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的 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 的响应情况看到差别，另一方面，我们也可以在 PC/Mac 上抓包来看看具体的网络层的情况。

图 4-65 所示是一个简单的试验的结果，基于上图的参数，开启了上面的网络模拟之后，Wireshark 抓包的结果可以看出，因为丢包产生了大量的 TCP 重传，Info 字段显示 TCP Retransimission 的部分。

图 4-65 模拟弱网络下的抓包结果

进一步，如果我们通过 Follow TCP Stream 观察某一个 HTTP 请求对应的整个 TCP 连接的过程，也能看到重传，乱序（TCP out-of-order），如图 4-66 所示。从响应延迟上看，很多也是超过一秒的，基于下面 Time 字段的相对时间戳，第一个用于建立连接 TCP SYN 发出是 33.019918s，在 36.018776s 经历了一次重传，而对方的 SYN+ACK 回来时间是

38.468732s。另外，观察 584 号包的 HTTP POST 请求，以及对应的 674 号包的 response，也是超过了 1s。对比网络状况模拟之前的情况，可以明显地看到弱网络模拟确实生效了。

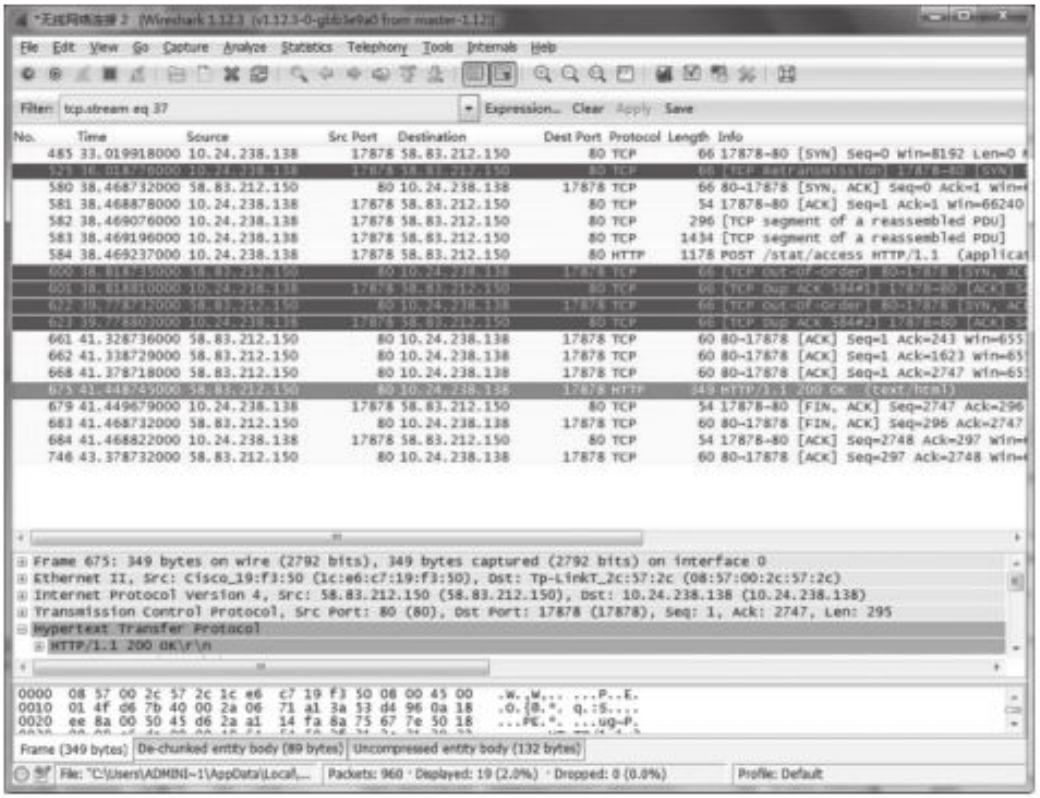


图 4-66 某一条 TCP Stream 在弱网络下的结果

在这种情况下，我们可以操作 App，观察 App 各功能的情况，也可以观察日志中有没有什么异常发生。

2. Mac 下的 Network Link Conditioner

接下来介绍的工具是运行在 Mac OS 上的 Network Link Conditioner，如图 4-67 所示，基本功能和上面介绍的 Windows 下的 Network Delay Simulator 类似。

手机通过前面流量测试中提到的方式，代理连到 Mac 上，同时打开代理和网络状况模拟程序。这里是 Charles 和 Network Link Conditioner 的组合。下面我们尝试模拟不同的情况，并观察 App 对应的接口层面的响应情况。

图 4-68 所示是网络为 WiFi 的情况，可以看出接口的延迟相对比较小。

图 4-67　Mac 下的 Network Link Conditioner

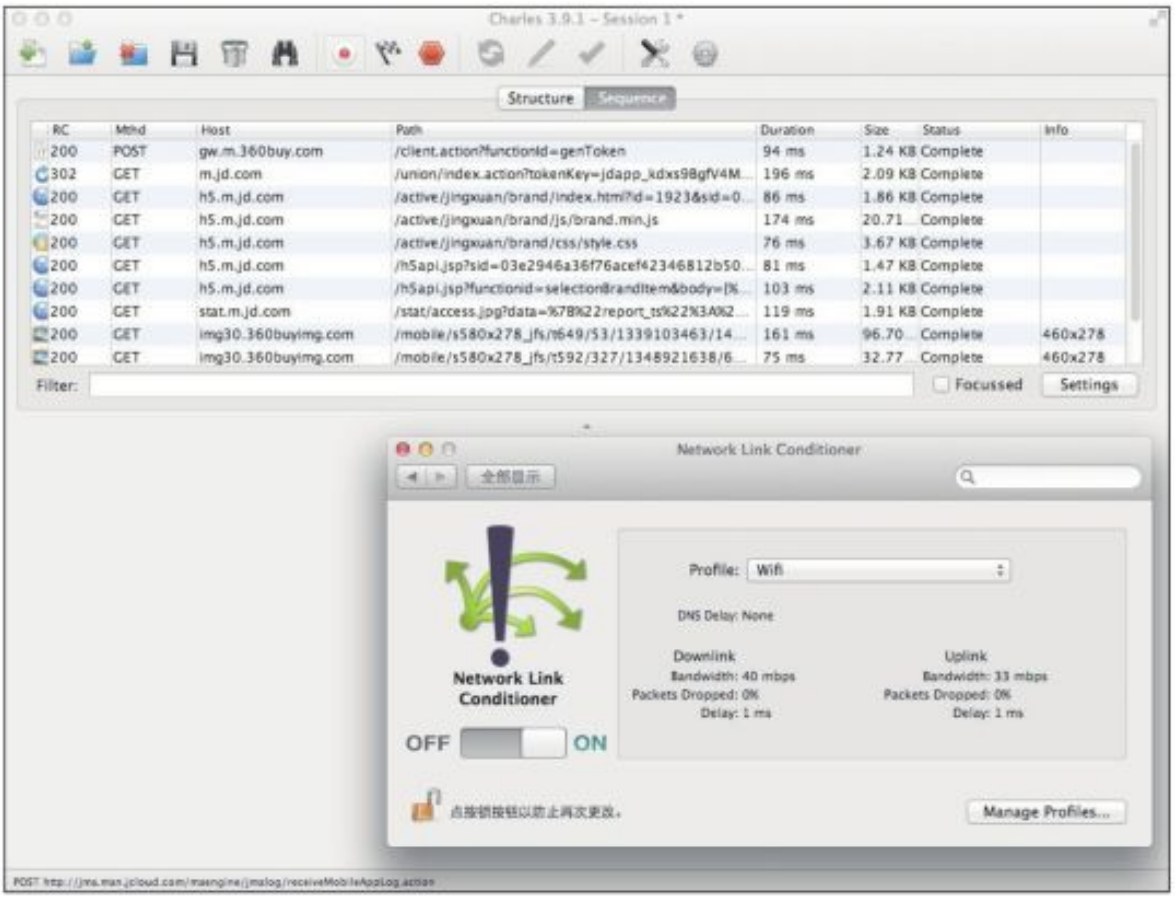


图 4-68　WiFi Profile 下的抓包结果

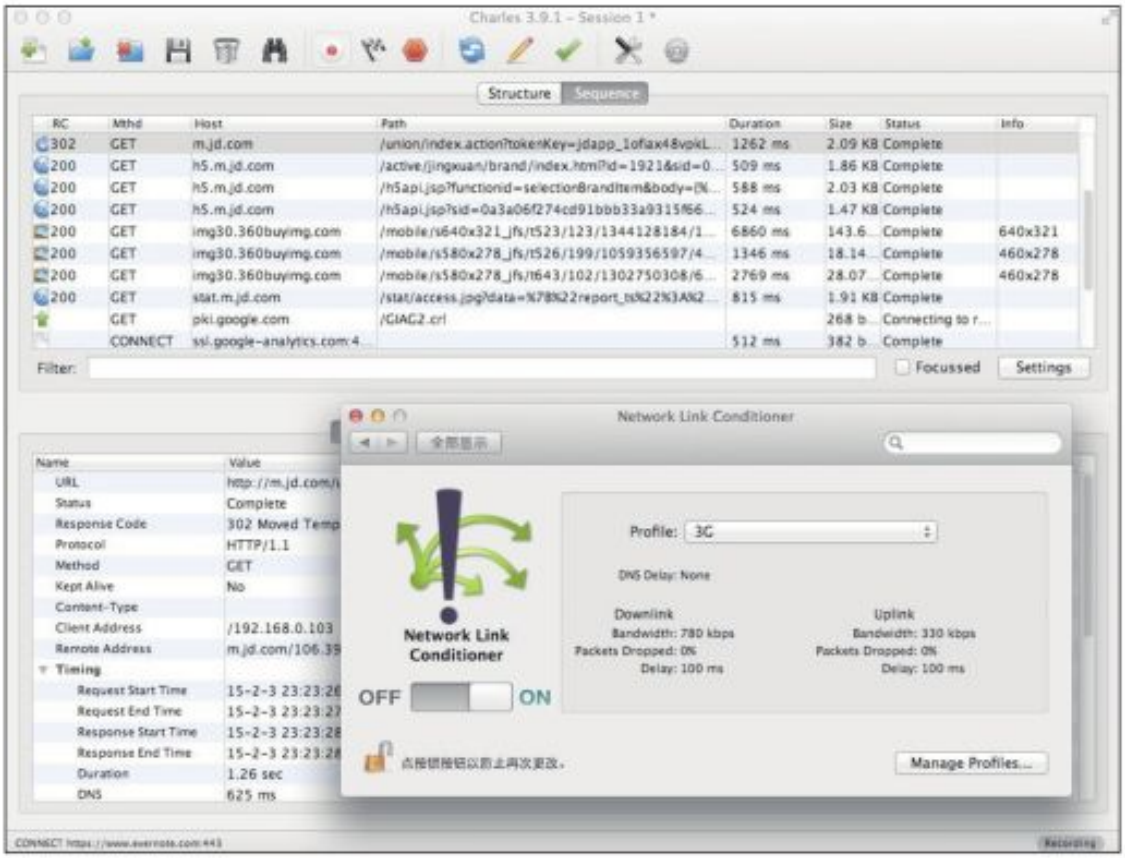


图 4-69 3G Profile 下的抓包结果

图 4-69 是模拟 3G 网络的情况，可以看出带宽要低很多，而且网络延时设置到了 100ms。从接口响应来看，很多接口的响应得到了 500ms 以上，个别达到了几秒。

图 4-70 是模拟 2G 网络的情况，带宽进一步降低，延迟也增大到 400ms，这种情况下所有接口的响应都达到 1.8s 以上，个别数据量较大的请求达到了 10 多秒。

在网络较差的情况，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 实际的行为，比如部分显示，或者提示加载进度等，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，实际中，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。工具本身使用的代价不大，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。

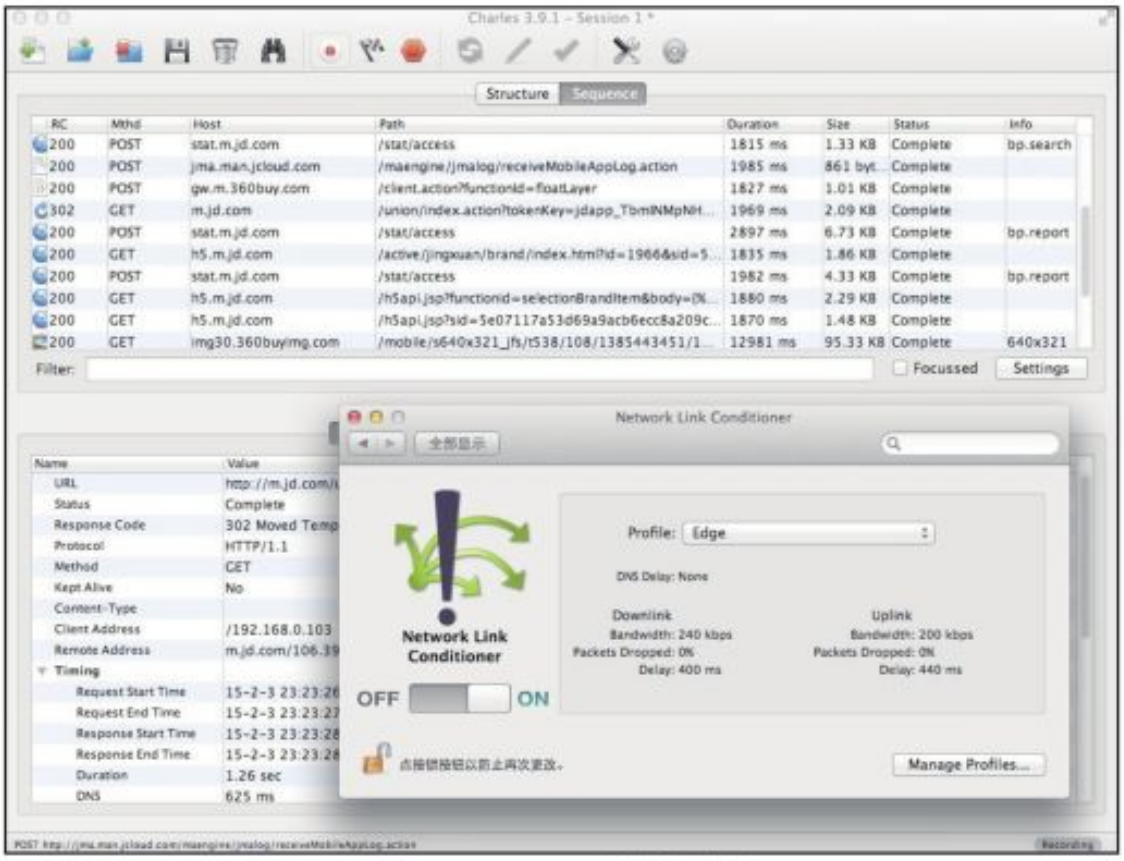


图 4-70　2G Profile 下的抓包结果

不过严格来说以上只是对弱网络情况的简单模拟，因为实际情况可能更加的多变和复杂。上面的模拟中，参数是固定的，而实际中肯定是多变的，另一方面，以上也没有包含网络类型切换，以及有网 / 无网的切换。

弥补这一部分的差异，可以通过实地的测试来进行。比如带上手机，装上被测的应用，然后到电梯，地铁等场景下来测试。

4.5　稳定性测试

在保证基本功能正确性的基础之上，App 的稳定性就显得非常重要，如果一个 App 经常出现闪退或者卡死，那么用户体验会受到很大伤害，在有竞品的情况下很容易造成用户的流失。

提起 App 的稳定性测试，很容易联想起 MonkeyTest，这个方法的使用门槛和成本非常的低，但是通常收获也比较有限。而其他专门针对稳定性的方法还比较少。当然，很多

稳定性的问题也是通过黑盒功能测试、灰度内测，以及用户问题反馈机制得以发现进而修复的。

在本节中我们会介绍 Monkey Test 的基本用法，对于不太了解的读者可以做一个参考，另外也会介绍一个我们在探索和实验中的方法，借助模糊测试的思路和 App UI 自动化的技术来做稳定性的测试。

需要说明的是，这里讨论的主要是 App 本身的稳定性测试，对于 App 所依赖的后台服务的稳定性这里并未谈及，那一部分和所有的后台服务一样，可以参考之前已经有的针对服务的测试方法和技术。

4.5.1 基于 Monkey 的稳定性测试

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

Monkey 程序由 Android 系统自带，使用 Java 语言编写，在手机的 Android 文件系统中的路径是：/system/framework/monkey.jar，如图 4-71 所示。

```
root@hlte:/system/framework # pwd
/system/framework
root@hlte:/system/framework #
root@hlte:/system/framework # ls -l monkey*
-rw-r--r-- root     root          313 1970-03-13 09:58 monkey.jar
-rw-r--r-- root     root       120800 1970-03-13 09:58 monkey.odex
root@hlte:/system/framework #
```

图 4-71 Android 自动的 Monkey 命令路径

另外，系统还提供了一个 shell 脚本来启动 Monkey 程序。shell 脚本在 Android 文件系统中的存放路径是：/system/bin/monkey。通常运行的时候我们都是直接使用这个 shell 脚本。图 4-72 所示是 Monkey 脚本的内容。

```
root@hlte:/system/bin # pwd
/system/bin
root@hlte:/system/bin #
root@hlte:/system/bin # ls -l monkey
-rwxr-xr-x root     shell         217 2013-09-11 23:29 monkey
root@hlte:/system/bin #
root@hlte:/system/bin # cat monkey
# Script to start "monkey" on the device, which has a very rudimentary
# shell.
#
base=/system
export CLASSPATH=$base/framework/monkey.jar
trap "" HUP
exec app_process $base/bin com.android.commands.monkey.Monkey $*

root@hlte:/system/bin #
```

图 4-72 monkey 脚本内容

Monkey 程序提供了很多的执行参数，如图 4-73 所示。

```
$ adb shell monkey [options]
```

```
root@hlte:/system # monkey
usage: monkey [-p ALLOWED_PACKAGE [-p ALLOWED_PACKAGE] ...]
              [-c MAIN_CATEGORY [-c MAIN_CATEGORY] ...]
              [--ignore-crashes] [--ignore-timeouts]
              [--ignore-security-exceptions]
              [--monitor-native-crashes] [--ignore-native-crashes]
              [--kill-process-after-error] [--hprof]
              [--pct-touch PERCENT] [--pct-motion PERCENT]
              [--pct-trackball PERCENT] [--pct-syskeys PERCENT]
              [--pct-nav PERCENT] [--pct-majornav PERCENT]
              [--pct-appswitch PERCENT] [--pct-flip PERCENT]
              [--pct-anyevent PERCENT] [--pct-pinchzoom PERCENT]
              [--pkg-blacklist-file PACKAGE_BLACKLIST_FILE]
              [--pkg-whitelist-file PACKAGE_WHITELIST_FILE]
              [--wait-dbg] [--dbg-no-events]
              [--setup scriptfile] [-f scriptfile [-f scriptfile] ...]
              [--port port]
              [-s SEED] [-v [-v] ...]
              [--throttle MILLISEC] [--randomize-throttle]
              [--profile-wait MILLISEC]
              [--device-sleep-time MILLISEC]
              [--randomize-script]
              [--script-log]
              [--bugreport]
              [--periodic-bugreport]
              COUNT
```

图 4-73　monkey 命令的参数

这里不详细展开讲解，常用的几个参数是：

- -p 指定被测的 App 包名。
- -v 显示执行时的信息。
- COUNT 发送的事件数目。

下面是一个 Monkey 常见用法的示例，它启动指定的 App，并向其发送 100 个伪随机事件。

```
$ adb shell monkey -p com.jingdong.app.mall -v 100
```

从图 4-74 所示的执行结果，以及在执行过程中观察手机上的动作，可以发现 monkey 模拟的事件非常广泛，除了对于被测 App 的点击、滑动、键盘输入等常见的操作之外，它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：

- 执行的次数到了。
- 如果限定了 Monkey 运行在一个或几个特定的包上，当监测到试图转到其他包的操作，会对其进行阻止。

图 4-74 monkey 执行结果

- ❑ 如果应用程序崩溃或接收到任何失控的异常，Monkey 将停止并报错。
- ❑ 如果应用程序产生了 ANR（应用程序不响应）的错误，Monkey 将会停止并报错。

MonkeyRunner

MonkeyRunner 和 Monkey 其实并无直接关联，但常常被认为是 Monkey 的一种延伸。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，控制发送模拟的击键操作，并可以对模拟器进行各种操作。

关于 MonkeyRunner 的详细介绍可以参考 :http://developer.android.com/tools/help/monkeyrunner_concepts.html。

4.5.2 Android 的 ANR

Android 系统的 ANR 也是一个比较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 Not Responding，也就是我们常说的 ANR：

- ❑ 输入事件 5 秒内未响应完成，例如点击屏幕，按键等操作。
- ❑ BroadcastReceiver10 秒内未执行完毕。

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。

出现 ANR 是十分影响用户体验的。作为测试，在测试过程中遇到 App 发生 ANR 的时候需要提供开发人员测试步骤和现场数据。测试步骤一般不难提供，现场数据怎么获取呢？Android 的 ANR 日志存放在 /data/anr/traces.txt。其中包括了 App 每个线程的堆栈信息。开发可以通过这些数据定位问题。我们在连接手机到电脑后，可以使用如下命令将日志导出到电脑上：

1. 首先进入 adb shell，使用下列命令将 ANR 日志文件复制到可导出的文件夹：

```
cp /data/anr/traces.txt /sdcard/traces.txt
```

2. 然后用 adb 命令导出到本地的文件：

```
adb pull /sdcard/traces.txt 本地文件路径
```

将此文件提供给开发人员即可进行后续排查。有兴趣的读者也可以查阅其中的内容进行分析。

4.5.3 基于模糊测试思路的稳定性测试方法探索

通过上面介绍的 Monkey 和 MonkeyRunner 工具，我们可以进行一些基本的稳定性测试。但是有一个问题，如果我们希望稳定性测试中的操作更加贴近用户的行为和日常的操作就需要自己动手来做一些控制，用解决功能自动化的方法。

下面要介绍的是一个比较实验性的方法，目前已经在实际项目中完成了试运行，还没有大规模开展。综合来看是 App 端稳定性测试的一种思路，同时也是 App UI 自动化技术的一个比较好的应用。

这个方法借鉴了模糊测试（Fuzzing Test）的思想，模糊测试常常用于做安全性测试，发掘安全漏洞。但其实这个测试方法本身并不局限于安全测试领域，对于稳定性也是一个很有效的测试方法。鉴于这个方法是一个很通用且重要的测试方法，而部分读者之前可能没有接触过，下面会先做一个简单的介绍。

4.5.3.1 模糊测试方法简介

模糊测试方法详细展开篇幅较大，各种细节可以参考一本针对性的书籍，《模糊测试：强制发掘安全漏洞的利器》，这里结合一个具体的例子做个基本的介绍。

模糊是相对于通常功能测试的“确定”而言的。在我们写测试用例的时候，由于是人工或者确定的自动化脚本来执行，需要把测试步骤和数据明确下来。即便用例本身的文字写得不明确，实际执行的时候还是会选取具体的执行路径和输入数据。

这里以测试一个电子邮件系统为例，如果我们的用例是发送一封邮件，然后查看接收

的结果，那么用例的执行步骤可能是这样的：

打开邮件客户端发送一封电子邮件，发件人 jeremy@sender.com，收件人 james@receiver.com 和 richard@receiver.com，标题是：测试邮件，正文内容为一段 200 字左右的文本。电子邮件的发送是基于 SMTP 协议的，以上的操作对应到 SMTP 协议层面的步骤如下：

```
S: 220 receiver.com Simple Mail Transfer Service Ready
C: EHLO sender.com
S: 250- receiver.com greets sender.com
S: 250-8BITMIME
S: 250-SIZE
S: 250-DSN
S: 250 HELP
C: MAIL FROM:<jeremy@sender.com>
S: 250 OK
C: RCPT TO:<james@receiver.com>
S: 250 OK
C: RCPT TO:<richard@receiver.com>
S: 250 OK
C: DATA
S: 354 Start mail input; end with <CRLF>.<CRLF>
C: Blah blah blah...
C: ...etc. etc. etc.
C: .
S: 250 OK
C: QUIT
S: 221 receiver.com Service closing transmission channel
```

也就是按照 EHLO，MAIL FROM，RCTP TO，DATA，.，QUIT 的命令序列与 Server 交互。

从以上的步骤来看，是一个比较确定的交互过程，而且每一步输入的数据也是确定的。这样的测试可以发现一些功能层面的问题，但是对路径和数据的覆盖非常有限。而模糊测试是针对这种情况的一个很大的扩展，技术上有一些创新，主要体现在以下几个方面：

- 单个步骤的数据引入随机，包括一些异常情况。这里的例子用到了一个模糊测试的框架 Sulley，其本身提供了一些默认的随机算法，有些是基于历史积累的容易出错的字符。
- 执行步骤的不确定性。有些步骤可以被反复执行，但是步骤本身之间有一定的逻辑关系，Sulley 框架本身也支持步骤的逻辑连接。
- 多个随机出来的步骤串行的持续运行。可以帮助发现一些累积性的问题，比如内存泄漏。
- 日志和辅助监控手段。当出现问题，比如程序崩溃的时候可以找到当时执行的步骤和数据，便于重现和定位。

基于这样的思路，下面是一个基于 Sulley 框架写的 SMTP 邮件发送过程的模糊测试的例子，基于 Python 语言编写的。这里为了简洁只考虑 ehlo 而忽略 helo。

```
#引入Sulley的库，
from sulley import *
#接受SMTP第一次连接时的greeting信息
def get_greeting_msg(sock):
    greet_message = sock.recv(1000)
    session.log("Greeting Message -->%s" % greet_message, 2)
#定义callback用于记录日志
def callback(session, node, edge, sock):
    session.log("Date sent -->%s" % node.render(), 2)
s_initialize("ehlo")
if s_block_start("ehlo"):
    s_static("EHLO ")
    s_delim(" ")
    s_random("xxx.com", 5, 30)
    s_static("\r\n")
s_block_end()

s_initialize("mail from")
if s_block_start("mail from"):
    s_static("MAIL FROM:")
    s_delim("<")
    s_string("jeremy")
    s_delim("@")
    s_string("sender.com")
    s_delim(">")
    s_static("\r\n")
s_block_end()

s_initialize("rcpt to")
if s_block_start("rcpt to"):
    s_static("RCPT TO:")
    s_string("james")
    s_delim("@")
    s_string("receiver.com")
    s_static("\r\n")
s_block_end()

s_initialize("pre_data")
if s_block_start("pre_data"):
    s_static("DATA\r\n")
s_block_end()

s_initialize("data_content")
if s_block_start("data_content"):
    s_static("Received:")
    s_string("test")
```

```
    s_static("\r\n")
    s_static("Subject:")
    s_string("test mail subject")
    s_static("\r\n")
    s_static("\r\n")
    s_string("test mail body")
    s_static("\r\n.\r\n")
s_block_end()

s = sessions.session(log_level=100)
target = sessions.target("127.0.0.1", 25)
s.add_target(target)
s.connect(sess.root, s_get("ehlo"), callback)
s.connect(s_get("ehlo"), s_get("mail from"), callback)
s.connect(s_get("mail from"), s_get("rcpt to"), callback)
s.connect(s_get("rcpt to"), s_get("pre_data"), callback)
s.connect(s_get("pre_data"), s_get("data_content"), callback)
s.fuzz()
```

为了对 SMTP 的 Server 做模糊测试，我们需要按照 SMTP 协议交互的过程，发送数据来和 Server 交互，每次命令发出的数据包称为 Request，比如 MAIL FROM 及其参数是一个 Request。在 Sulley 框架中，Request 是按照“块”（block）的方式来构造的，而每个 block 又是由“原语”（primitive）来构成的，也就是一些基本的数据类型，比如整数、字符、随机数 / 串等。

在普通的测试中，一个带有确定参数的 MAIL FROM 请求类似这样：

```
MAIL FROM:<jeremy@sender.com>
```

采用模糊测试方法后，上面的 Request 变成了下面的一个 block：

```
if s_block_start("mail from"):
    s_static("MAIL FROM:")
    s_delim("<")
    s_string("jeremy")
    s_delim("@")
    s_string("sender.com")
    s_delim(">")
    s_static("\r\n")
s_block_end()
```

s_static 表示该部分不做变化，“MAIL FROM：”固定出现，否则服务器会报协议错误，无法继续后面的流程。

s_delim 表示针对分隔符来做针对性的变化，会在多次的请求中变异出各种各样的分隔符，来测试服务器对各种情况的处理。这里针对“<”，“@”，“>”三个分隔符分别做了变异处理。

s_string 表示后面是一个随机出现的 string，这里的“ jeremy”相当于是一个随机种子，

框架会自动衍生出各种字符串，有各种特定字符的，以及长度超长的，等等。也可以根据测试的需求来做一些控制，这里不详细展开。

这样通过把一个请求的各个组成部分拆散成不变的和可变的，进而在不同的请求实例里面构造出各种复杂的数据，衍生出大量的用例。比如可能会出现这样的数据：

```
MAIL FROM:#jeremy&sender.com)
MAIL FROM:<s!#**#%$%@qqqqqqewrwerwerwerwetrytryurtytr.com)
```

如果服务器处理不当，就有可能出现缓冲区溢出或者特殊字符解析导致的程序错误，进而引发安全性和稳定性的问题。

以上是针对单个 Request 的变化，实际的协议通常都是由一系列有一定逻辑关系的 Request 构成的，比如上面提到的 SMTP 协议的顺序。在 Sulley 的定义里面，将一系列的 Request 称为一个 Session。上面示例代码里面的下列语句就是用来告诉 Sulley 框架，这些 Request 直接是如何串接起来的，简单理解其思路就是定义每个有逻辑意义的连接点，然后在实际执行过程中就可以沿着整个图来遍历，每个步骤也可以出现多次或者反复：

```
s.connect(sess.root, s_get("ehlo"), callback)
s.connect(s_get("ehlo"), s_get("mail from"), callback)
s.connect(s_get("mail from"), s_get("rcpt to"), callback)
s.connect(s_get("rcpt to"), s_get("pre_data"), callback)
s.connect(s_get("pre_data"), s_get("data_content"), callback)
```

了解了模糊测试方法的上面两个核心要点之后，我们对这个方法的思路有了一定的了解。基于上面两点就可以自动生成大量的测试用例，对于每个用例来说，就是选择一个 Session 路径，以及随机出每个 Request 里面每一项的具体参数，然后就可以像一个普通用例一样执行了。根据实际协议的 Request 的数量，以及可变的参数数量，和一些基本参数的设定，这样的方法可以自动构造出成千上万的测试用例，然后一个个执行下去。

4.5.3.2 基于模糊测试方法和 UI 自动化的稳定性测试方案

在上面小节我们借助一个实例了解了模糊测试方法的基本思路，接下来我们考虑如何把这个方法应用到 App 的稳定性测试上。在上面 SMTP 的例子中，每一个 Request 是一次网络请求，对应到 App 的稳定性上面，每一次“Request”就是一次在 App 上面的 UI 操作。关于 App 的 UI 操作我们可以借助前面自动化章节讨论到的技术来实现，这里不再赘述。

接下来我们讨论这个方案的具体思路，主要包含几个方面：

1）首先整理出一些用户的常见操作，类似于前面的一个 Session。

以电商的 App 为例，常见的操作路径包括：

- 首页 – 浏览焦点图活动 – 查看商品详情
- 首页 – 搜索一个商品 – 查看商品详情 – 查看评论 – 加入购车

- 首页 – 购物车 – 下单 – 结算 – 支付
- 首页 – 我的 – 查看订单列表 – 查看订单详情

实际可能的操作路径要多很多，这里只是列举了一些。以上的每一步操作我们都可以通过 App UI 自动化来表达。

2）将操作中的数据做一些随机化，比如：

- 搜索的关键词
- 添加到购物车的商品的数量（但是要满足产品的约束）
- 支付方式
- 浏览活动和商品列表的上下滑动的幅度和次数

3）上面每一个操作的步骤序列里面的步骤引入随机，比如浏览商品，可以连续浏览多个商品，并查看商品详页。

4）将多个随机产生的用例串联起来。可以用执行时间或者用例个数为限制持续运行下去。

5）完善的测试框架的日志，以及 App 层面的监控，包括对 App 界面的截图，自身的日志，以及 CPU 和内存使用情况。在出现问题的时候能够快速定位。

基于上面的思路，我们做了一些实践的探索，图 4-75 所示是目前的原型系统的执行日志。

Excute Time	Level	Error Location	Description
2015-02-13 11:40:24	INFO	Console.java:83	onExecutionStart
2015-02-13 11:41:08	INFO	Console.java:83	开始
2015-02-13 11:41:11	INFO	Console.java:83	测试用例开始执行
2015-02-13 11:41:11	INFO	Console.java:93	class Operation.FuzzingOperation.fuzzingTestOperationAndroid
2015-02-13 11:41:11	INFO	Console.java:83	flow_searchProduct
2015-02-13 11:41:11	INFO	Console.java:83	Excuting Flow 1
2015-02-13 11:41:11	INFO	Console.java:83	******** Excuting flow_searchProduct **********
2015-02-13 11:41:20	INFO	Console.java:83	[首页搜索框] 输入文本:samsung
2015-02-13 11:41:25	INFO	Console.java:83	点击[搜索按钮]
2015-02-13 11:41:28	INFO	Console.java:83	执行通过
2015-02-13 11:41:31	INFO	Console.java:83	测试用例开始执行
2015-02-13 11:41:31	INFO	Console.java:93	class Operation.FuzzingOperation.fuzzingTestOperationAndroid
2015-02-13 11:41:31	INFO	Console.java:83	flow_cart
2015-02-13 11:41:31	INFO	Console.java:83	Excuting Flow 2
2015-02-13 11:41:31	INFO	Console.java:83	******** Excuting flow_cart **********
2015-02-13 11:41:33	INFO	Console.java:83	get [首页] 中index:3的元素
2015-02-13 11:41:37	INFO	Console.java:83	get [首页] 中index:0的元素
2015-02-13 11:41:37	INFO	Console.java:83	执行通过
2015-02-13 11:41:40	INFO	Console.java:83	测试用例开始执行
2015-02-13 11:41:40	INFO	Console.java:93	class Operation.FuzzingOperation.fuzzingTestOperationAndroid
2015-02-13 11:41:40	INFO	Console.java:83	flow_story
2015-02-13 11:41:40	INFO	Console.java:83	Excuting Flow 3
2015-02-13 11:41:40	INFO	Console.java:83	******** Excuting flow_story **********
2015-02-13 11:41:41	INFO	Console.java:83	get [首页] 中index:2的元素
2015-02-13 11:41:44	INFO	Console.java:83	点击 [故事]
2015-02-13 11:41:47	INFO	Console.java:83	get [首页] 中index:0的元素
2015-02-13 11:41:47	INFO	Console.java:83	执行通过
2015-02-13 11:41:50	INFO	Console.java:83	测试用例开始执行

图 4-75　基于模糊测试的稳定性测试 Demo 结果

4.6 安全测试

传统的 Web 安全测试的方法，相对已经比较成熟，这里不做介绍，大家可以参考相关的书籍。下面从移动 App 产品的角度介绍一些测试点和测试方法。

4.6.1 安装包测试

以下是几个针对安装包的测试点。

1. 能否反编译代码

我们把移动应用发布出去后最终的用户获得的是一个程序安装包。我们需要关注的是用户能否从这个安装包中获取项目的源代码。为何要关注源代码泄漏问题呢？除了保护自己公司的知识产权外，还有安全方面的考虑。一些程序开发人员会在程序源代码中硬编码一些敏感信息，例如密码等。一旦这些敏感信息被获取可能导致被恶意使用，引起一些预期外的风险。另外程序内部的一些设计欠佳的逻辑也可能隐含漏洞。一旦源代码泄漏，带来的安全风险是十分高的。

为了避免这些问题，除了在代码审核的时候关注外，通常开发的做法是对代码进行混淆。混淆后的源代码通过反编译软件生成的源代码是人很难读懂的。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，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。

2. 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等问题。我们可以使用下列命令检查：

```
jarsigner -verify -verbose -certs apk包路径
```

如果运行后结果为“jar 已验证”，说明签名校验成功。

3. 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损坏，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。

4. 权限设置检查

一些用户对于自己的隐私问题十分敏感。因此，我们需要对 App 申请某些特定权限的必要性进行检查，例如访问通讯录等。对于没必要的权限，一般都建议开发直接去除。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，并结合需求逐一校验此权限是否为必须的。另外，对于 manifest 文件的修改也需要关注。在增加新权限前需要进行评估。

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，或是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控制器 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。

4.6.2 敏感信息测试

关于敏感信息，主要考虑以下几个方面。

1）数据库是否存储敏感信息。现今大部分移动 App 都会使用到数据库。如果在数据库中存储了敏感信息，一旦用户手机被他人获得，就可能造成用户的隐私泄露。特别是一些应用会把 cookie 类数据保存在数据库中。一旦此数据被他人获取，可能造成用户账户被盗用等严重问题。我们在测试中也需要关注这一问题。我们需要对各个数据库字段含义进行了解，并评估其中可能的安全问题。除了开发需要关注这点外，测试也需要把关。测试中，在跑完一个包含数据库操作的测试用例后，我们可以直接查看数据库里的数据，观察是否有敏感信息存储在内。一般来说，这些敏感信息需要在用户进行注销操作后删除。如果是 cookie 类数据，建议设置合理的过期时间。

2）日志中是否存在敏感信息。一般情况下开发在写程序的过程中都会加入日志帮助调试。在日志内可能会写入一些敏感信息。通常在应用发布版本不会使用日志。但也不排除一些特殊情况。对于发布版本中包含日志的应用，在测试的时候我们也需要关注日志中是否包含敏感信息。

3）配置文件是否存在敏感信息。与日志类似，我们需要检查配置文件中是否包含敏感信息。

4.6.3 软键盘劫持

如果用户安装了第三方软键盘，那么在用户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包含恶意代码，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漏出去的。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。测试中需要关注这点并给予开发建议。注：iOS8 以下的平台非越狱机器则无需担心这点，第三方软键盘程序不能安装。不过最新的 iOS8 已经支持第三方输入法。

4.6.4 账户安全

对于用户账户的安全性，我们一般需要关注以下几点：

- 密码是否明文存储在后台数据库。近几年有不少大型网站都爆出了在数据库中明文存储用户名和密码的问题，其所带来的负面影响是十分巨大的。在评审和测试中需要关注密码的存储。
- 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。
- 账户锁定策略。对于用户输入错误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。如果没有此类保护措施，用户账户很可能遭遇暴力破解。我们需要在评审中关注这点，并在测试中进行验证。注：一些常见的此类需求的设计缺陷需要被注意，例如后台对每个账号做次数限制可能会引起所有账号都被策略锁定等等。测试人员也需要有一定的对设计缺陷进行评估的能力。
- 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他的账号可能已经被泄漏了。在一定程度上能够提升用户体验。举个反例，一个聊天程序，如果支持同时登陆而又不提供同时会话的通知，用户 A 传给用户 B 的聊天内容很可能会被传给盗取用户 B 密码的恶意用户 C，而 B 完全不知道。对于没有实现此功能的应用，作为测试可以提出建议。
- 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。

4.6.5 数据通信安全

数据通信的安全性方面，测试中我们主要关注以下几点：

- 关键数据是否散列或加密。密码在传输中必须是加密的。其他敏感信息在传输前也需要进行散列或者加密，以免被中间节点获取并恶意利用。
- 关键连接是否使用安全通信，例如 HTTPS。在获知接口设计后我们需要评估是否其中内容包含敏感信息，如果未使用安全通信，需要知会开发修改。
- 是否对数字证书合法性进行验证。即便使用了安全通信，例如 HTTPS，我们也需要在客户端代码中对服务端证书进行合法性校验。测试中可以使用 Fiddler 工具模拟中间人攻击方法。如果客户端对于 Fiddler 证书没有校验而能正常调用，则存在安全隐患。
- 是否校验数据合法性。在一些情况下，我们需要有方法来确保服务端下发的明文数据不被篡改。通常开发侧的实现方式是对数据进行数字签名并在客户端进行校验。我们可以模拟后台返回进行相关的测试工作。此外，对于其他一些客户端未进行数据校验的接口，我们也需要有意识地思考如果不进行校验是否会产生问题，并通过模拟后台返回验证。

4.6.6 组件安全测试

这里主要是指 Android 平台各个组件是否能被外部应用恶意调用从而带来一些安全问题。包括 Activity、Service、ContentProvider、Broadcast 等等。采用的测试方法是通过使用 drozer 工具（https://www.mwrinfosecurity.com/products/drozer/）结合查看代码的方式。具体使用方法此处不赘述。建议读者阅读相关官方文档。

4.6.7 服务端接口测试

我们主要会关注我们的服务端接口是否存在以下问题：

- SQL 注入
- XSS 跨站脚本攻击
- CSRF 跨站请求伪造
- 越权访问

除了上述常见的服务端问题外，我们还需要结合实际的需求，设计和代码，分析是否需求或设计本身就会带来安全问题。

举个极端一点的例子，有一个购物的应用，下单的流程包含两个接口，接口 A 返回订

单详情，其中包括订单号码和金额总价。调用接口 A 后用户在客户端看到一个订单页面。用户点击提交后调用接口 B，客户端传给接口 B 的参数为接口 A 返回的订单号码和金额总价，接口 B 的后台根据传给接口 B 的金额总价从用户账户中扣款，扣款成功后即根据订单号码发货。这一设计有什么问题呢？那就是接口 B 完全信任了客户端传入的金额总价而未做校验。恶意用户可以直接调用接口 B，传入伪造的金额和真实订单号。这样就能以便宜的价格购物。

4.7 环境相关的测试

实际项目中，有一些缺陷我们发现是和 App 所处的运行环境相关的，所以在测试设计和执行的过程中也需要考虑这些方面的情况。这类场景非常多，也需要结合 App 的特点，比如有些 App 会用到摄像头、加速度感应等硬件设备，那么也需要考虑对应的情况。下面介绍几种相对比较通用、会普遍遇到的环境相关的问题，供大家参考。

4.7.1 干扰测试

在用户使用手机上 App 的时候，会遇到一些打断的情况，例如正在浏览页面的时候收到一个电话等等。如果开发代码写的不够严密，会导致我们的 App 在这些情况下发生一些异常的行为。我们在测试中也需要覆盖到这些干扰的情况。常见的场景有以下几种：

- 收到电话
- 收到短信
- 收到通知栏消息
- 无电提示框弹出
- 第三方安全软件告警框弹出

当然，由于测试时间和测试资源有限，我们一般不推荐对每个界面甚至每个功能点都进行干扰测试。在做此项测试之前需要先评估一下功能本身跟干扰的关联性。举个例子，如果我们的应用是一款游戏 App，那么我们需要保证用户在游戏中收到电话后能正常中断游戏并在挂断电话后能够正常恢复游戏。另外也需要考虑通知栏消息是否会覆盖掉界面上实时性高的游戏相关的文字等等。另外一个我们在实际中遇到的案例也很有参考性。在某种特定的支付场景下，需要用户输入手机收到的短信验证码才能完成支付环境。App 端开发在实现这一功能的时候，考虑支付的安全性，当用户按 Home 键将 App 切到后台后做了保护处理，不再显示输入验证码的界面。而根据不同手机和用户的设置，收到短信验证码的时候不一定会在通知栏顶部显示，那么用户就需要切到读取短信的地方再切换回来，但

是这时由于上面提到的保护措施，已经不可以再输入验证码了，这样用户就无法走完支付流程。这是一个很典型的 App 在手机上特有的场景，如果我们的测试人员开启了短信预览功能，或者使用两个手机，一个有 SIM 卡用于接收短信，另一个测试机用于执行 App，以上情况就会被掩盖，等到真实用户遇到这类问题就会投诉。

这些细节问题产品和开发可能不会一开始在定义需求和开发阶段想到，作为测试，我们需要有这方面的意识和能力，在需求评审中提出这类问题并进行后续的测试。

4.7.2 权限测试

一些用户在实际使用 App 的时候会有意识地阻止某些功能。例如有的用户感觉让某个 App 访问电话本或者相册可能泄漏隐私，就在手机的设置中禁止了该 App 访问相册的权限。

图 4-76 针对具体应用的权限设置

如图 4-76 所示，Android 用户可以在系统中禁止某个 App 获得相关的权限。很多手机上的安全软件也提供了相关的功能。

在 iOS 中也有类似的设置，图 4-77 所示是 iOS 系统中列出的隐私权限。

针对每一个涉及隐私的权限类型，都可以针对性地设置各个 App 是否可以获取该权限。图 4-78 所示是针对定位服务的例子。

图 4-77 iOS 中的隐私权限设置

图 4-78 各 App 针对定位服务的权限设置

进一步，如图 4-79 所示，可以设置针对某 App 授予该权限的细节。

图 4-79 设置针对某 App 授予该权限的细节

所以在 iOS 系统中，涉及隐私相关的权限，用户可以比较精确地针对每一个 App 来设定。

App 代码如果对此类情况处理不当，很可能产生 UI 不友好、进程崩溃等诸多问题。这些情况也是在测试中需要覆盖到的。我们需要考虑每个功能是否用到了这些用户可控制的权限，如果用到，则要增加相应的测试用例。在实践中，测试人员可能并不清楚具体功能实现所需要的权限。对此一个可行的方法是由开发在提测时提供一个需要的权限列表。这样测试人员就能有针对性地进行测试。我们在测试中通过权限测试也发现了一些问题。例如，在一个需要定位功能的模块中，当我们的测试人员关闭了定位服务权限后，界面上的位置显示为错误的默认值“北京”，而没有任何友好的提示信息。可见权限测试在实际工作中的确能够帮助我们发现不少问题。

注意 此处的权限测试跟安全测试一节中的“权限设置检查”描述的是同一系统功能。但是本节内容主要是从用户使用角度进行黑盒测试，安全测试一节的内容偏白盒，注重如何分析 App 本身提供了哪些需要权限的功能。

4.7.3 边界情况

手机环境本身也有其边界情况需要在测试中覆盖。常见的场景有：

- 可用存储空间过少。如果程序有存储操作，建议覆盖此场景。通常来说，可接受的底线是如果进程被系统杀掉，在可用存储空间增多后能正常重启并使用 App。
- 没有 SD 卡 / 双 SD 卡。如果程序使用了 SD 卡，需要覆盖此场景。
- 飞行模式。如果程序使用数据网络、蓝牙等与飞行模式相关的功能，建议覆盖此场景。

- 系统时间有误（晚于和早于标准时间）。通过此场景的测试经常能够发现一些设计和代码上的缺陷。建议对所有与系统时间有关的功能均手动调整系统时间进行测试。
- 第三方依赖。如果我们的 App 对其他 App 有依赖关系（例如 QQ，微信联合登录等），那么我们还需要测试第三方 App 没有安装的情况，以及版本过低的情况。

4.7.4 Android 定位测试

有时候，在测试中我们会遇到一些 App 行为与地理位置相互关联的情况。例如我们需要给国内某特定地区用户展示一种特殊样式的 UI。我们在测试中不太可能拿着手机跑到这些地区去实际测试，而是会用下列方式进行模拟测试。

- 白盒方式。由于定位代码最终获取的是一个位置对象，我们只需要在获取到位置对象后手动设置经纬度即可进行测试。程序中使用的位置对象一般是 android.location.Location 对象。调用它的 setLatitude 和 setLongitude 方法就可以设置经纬度。

每次修改都需要手动设置并且重新编译会比较麻烦，对此我们可以进一步考虑使用文件的方式来进行配置。并且可以考虑加上开关，通过文件配置在测试代码和真实代码之间来回切换。这样能避免重新编译，提高效率。

- 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。Android Studio 和 Eclipse 等 IDE 都带有 DDMS 工具，如图 4-80 所示。

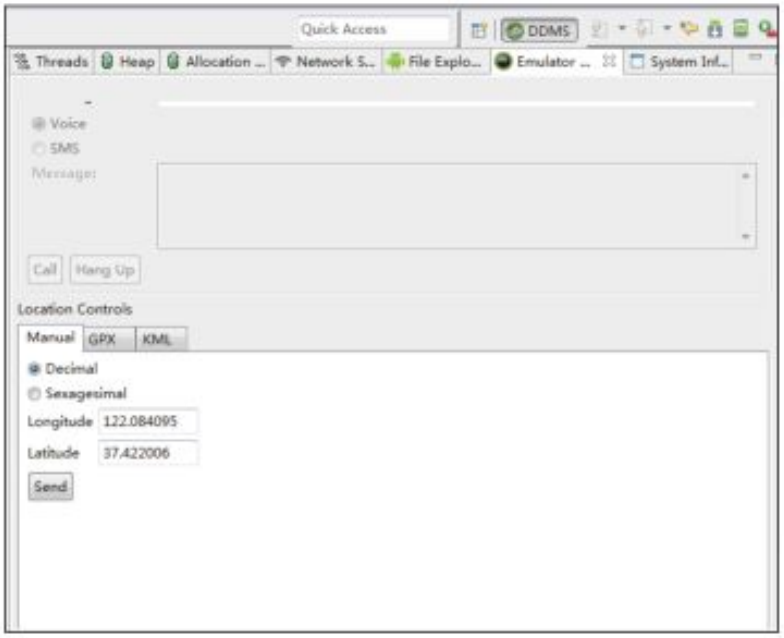


图 4-80 Android 模拟器中关于位置的模拟设置

其底层是调用了 adb shell 的 geo fix 命令。因此我们也可以手动使用：

```
geo fix [经度] [纬度]
```

来设置模拟器的经纬度。

- 自定义位置提供器。一些第三方 App 提供了基于自定义位置定位器的位置修改功能。要使用此功能需要在手机的开发者选项中设置允许虚拟位置，如图 4-81 所示。

图 4-81　Android 中关于位置模拟的开发者选项

通常这类 App 会提供一个地图界面给用户选择坐标，点击后就能修改机器的位置信息。

我们也可以自己写代码实现提供此功能的 App。下面是核心代码，将 GPS_PROVIDER 提供的位置设置为自定义的经纬度：

```
Location fakeLocation=new Location(LocationManager.GPS_PROVIDER);
manager.addTestProvider(LocationManager.GPS_PROVIDER,  true,
        false,
        false,
        false,
        true,
        true,
        true,
        Criteria.POWER_MEDIUM,
        Criteria.ACCURACY_FINE);
manager.setTestProviderEnabled(LocationManager.GPS_PROVIDER,  true);
```

```
manager.setTestProviderStatus(LocationManager.GPS_PROVIDER,  LocationProvider.
AVAILABLE,  null,  System.currentTimeMillis());
//设置为自定义的经纬度
    fakeLocation.setLatitude(23.129);
    fakeLocation.setLongitude(113.264);
    fakeLocation.setAccuracy(1);
    fakeLocation.setTime(System.currentTimeMillis());
    if(Build.VERSION.SDK_INT>16)
    {
    fakeLocation.setElapsedRealtimeNanos(SystemClock.elapsedRealtimeNanos());
    }
manager.setTestProviderLocation(LocationManager.GPS_PROVIDER, fakeLocation);
```

除了上述代码外，还需要在 manifest 文件中设置：

```
<uses-permission  android:name="android.permission.ACCESS_MOCK_LOCATION"></
uses-permission>
```

当其他程序使用：

```
LocationManager manager = (LocationManager) getSystemService(Context.LOCATION_
SERVICE);
Location location = manager.getLastKnownLocation(LocationManager.GPS_
PROVIDER);
```

获取位置的时候，将获取我们自定义的位置。

以上介绍了几种 App 常见的环境相关的问题，实际项目中可以在每个功能的测试用例设计中加入相关的场景考虑。也可以系统化的针对某一个维度来进行多个模块的测试，比如权限测试，可以把所有需要用到隐私相关权限的功能点罗列出来，然后在测试上禁止对应的权限，逐个查看对应的功能，看其行为是否可以接受。

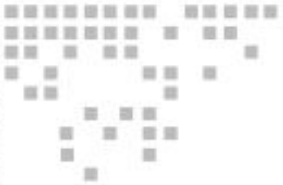
4.8 本章小结

在这一章，我们介绍了多个专项测试的方法。针对兼容性测试，我们介绍了如何在有限的资源情况下高效地开展手工的兼容性测试，以及如何借助 UI 自动化和云测试平台来进一步提升兼容性测试的效率和覆盖面。对于流量测试，我们介绍了 Android 和 iOS 平台相关的测试方法，并介绍了借助抓包和网络代理的通用测试方案，也讨论了我们基于代理开发的自动化工具，希望可以进一步提升效率。在电量测试方面，我们分别介绍了 Android 和 iOS 中用到的测试方法，也简单介绍了基于硬件的方案。对于弱网络测试方面，主要介绍了如何借助手机 OS 自带的工具，以及基于代理的第三方工具来模拟出我们想要的弱网络场景。对于稳定性测试，主要介绍了如何使用 Monkey，以及探索了借鉴模糊测试来开展的稳定性测试方案。在安全测试方面，主要介绍了 App 可能遇到的各个方面的安全问题，

可以针对这些问题来定向地进行安全测试。最后，针对 App 的运行环境的特点，介绍了测试中要注意的问题以及环境相关的一些问题。

本章介绍的测试方法是针对移动互联网产品中的 App 客户端来开展的，每个方式都是针对一类问题，相对比较独立，可以结合项目的情况单独开展。在业务团队中，就某一个专项测试的方面，可以让部分技术能力较好的成员，或者专职的测试开发人员先进行实践的探索，然后将方法和工具在团队内部推广。

显然，这里讨论的专项测试并不全面，还有很多的测试需求等待实施，也还有很多的测试技术和方法值得去探索。而且随着移动互联网产品的深入发展，还会有更多的产品技术形态出现，测试技术也需要不断地跟进和提升。



Chapter 5 第 5 章

辅助测试方法

发现一个问题往往比解决一个问题更为重要。

——爱因斯坦

在前面章节中，我们介绍了功能测试以及对应的自动化，还有性能测试和各种针对 App 的专项测试，这些都直接和某个功能点或者产品的业务目标效果。在实际测试中，还有一类方法并不直接针对某个功能，而是用横向的维度来分析一些代码和实现中可能出现的共性的问题。这些是我们测试方法中非常有效的补充，在这里我们称之为辅助测试方法。本章将要讨论的内容包括代码静态扫描、代码覆盖率分析、接口 Mock 方法和 AOP 测试方法。进一步来说，这些方法其实并不单单是针对移动互联网产品，其实非常通用，更多是和开发语言以及接口实现相关。灵活地运用这些方法，可以帮助我们更加高效地发现很多代码的问题，进一步提高产品的质量。

5.1 代码静态扫描

不用实际执行代码，而是仅通过工具扫描就发现问题是测试技术中重要的手段，也是各种开源和商业工具在不断追求的。这种方法也许并不完美（因为很多问题不一定通过静态的方法就能够找出），但却是非常有价值和高效的发现问题的方法，而且在代码刚写完，甚至编写的过程中就可以开展。这种方法的优点是，可以直接定位到问题代码，也不需要测试环境的安装和部署，等等。不过实践中也有几个问题需要注意：

- 静态扫描工具通常是编程语言强相关的，所以要选取适合项目所用语言的扫描工具。
- 针对一些主流的编程语言，比如 Java、C++，有多款商业的和开源的扫描工具，需要结合项目做一些评估。
- 扫描工具完成扫描和生成报告通常都比较快，但这只是开始，实际中更多的工作量在于分析扫描的结果，因此也不能排除误报的情况。
- 代码的静态扫描除了扫出代码缺陷外，也会涉及很多编程规范和风格的问题，所以需要持续进行，并进行相关的培训和宣导，借助静态扫描的结果持续提升代码质量，以及质量意识。

接下来我们针对 Android 和 iOS 的代码，介绍一些我们在实际项目中用到的方案，期望发现的问题主要是代码缺陷和冗余代码的问题。

5.1.1　针对 Android 的静态代码扫描

Android App 主要用到的编程语言有两种，一是用 Java 语言，另外一类是基于 NDK 用 C++ 来编写。后者多见于各种手机游戏 App，前者在大部分非游戏类 App 中占主流。由于代码静态扫描工具是语言相关的，所以需要选取合适的工具。在这里我们主要介绍基于 Java 开发的类型，C++ 的部分可以参考 C++ 相关的静态扫描工具。

下面介绍两个常用的免费工具，一个是 FindBugs，一个是 Lint。

5.1.1.1　FindBugs 工具的应用

FindBugs 是一款出色的 Java 静态代码扫描工具，能够帮助我们发现 Java 代码中隐含的问题。由于大部分 Android App 是用 Java 编写的，我们可以使用 FindBugs 来扫描 Android 的代码并发现问题。在实践中我们发现，FindBugs 能够帮助寻找出很多黑盒测试难以覆盖的问题，例如对空对象的引用未做检查等。

我们通常可以以三种方式使用 FindBugs：

- 插件方式——需要 IDE 支持，界面友好，使用方便。
- UI 界面方式——无需 IDE 依赖，需要一些手动配置。
- 命令行方式——适用于持续集成。

需要指出的是，FindBugs 发现的问题并不能直接认定为 bug。需要先对它发现的问题进行分析。一些优先级较低并且通过测试和开发都确定可以排除的问题，可以直接通过 FindBugs 的过滤器进行过滤。

根据我们的经验，FindBugs 发现的问题的 bug 转化率超过 60%，还是较为有效的。

1. 插件方式

下面是 Android Studio 1.0 下安装 FindBugs 插件的方法。

选择 File->Settings 打开设置面板，选择 Plugin，搜索 findbugs，点击结果视图中的 Browse 链接打开 Browse Repositories 面板，搜索 findbugs 并点击右侧的 Install plugin 按钮进行安装，如图 5-1 所示。

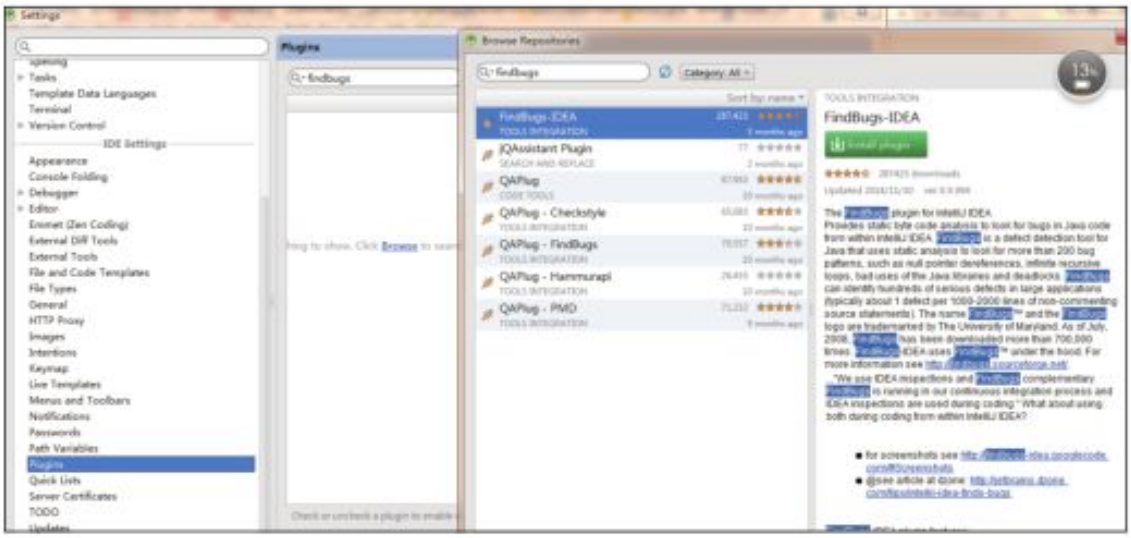


图 5-1　在 Android Studio 1.0 中安装 FindBugs 插件

安装完毕后重启 Android Studio，即可使用该插件。我们可以右键选择某个 Module，然后点击 Analyze Module Files，如图 5-2 所示。

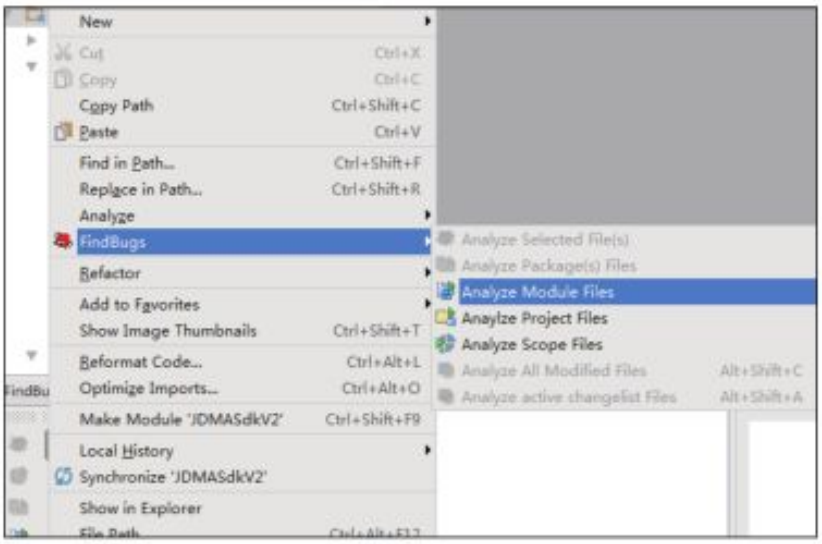


图 5-2　使用 FindBugs 插件分析代码

之后等待扫描完成，即可在 FindBugs 窗口下查看发现的问题。点击指定的问题即可跳转到对应的代码行并能够查看详细的问题描述，如图 5-3 所示。

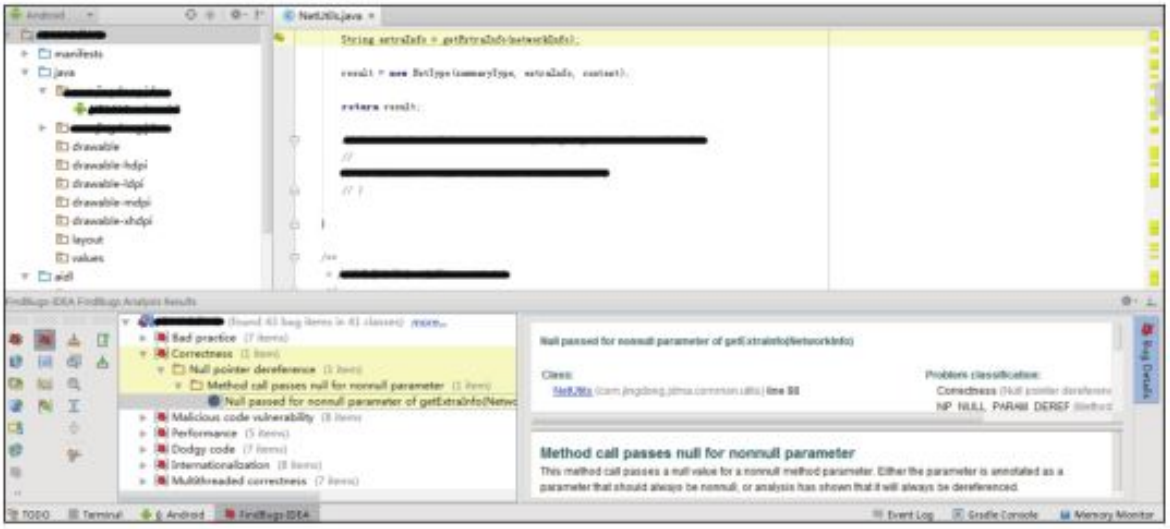


图 5-3　使用 FindBugs 插件查看分析结果

2. UI 界面方式

在 http://findbugs.sourceforge.net/downloads.html 下载安装 FindBugs 后，我们可以运行 FindBugs 安装路径下的 bing\findbugs.bat。之后点击“文件 -> 新建”，输入项目名、分析类包的目录、辅助类位置、源文件目录，如图 5-4 所示。

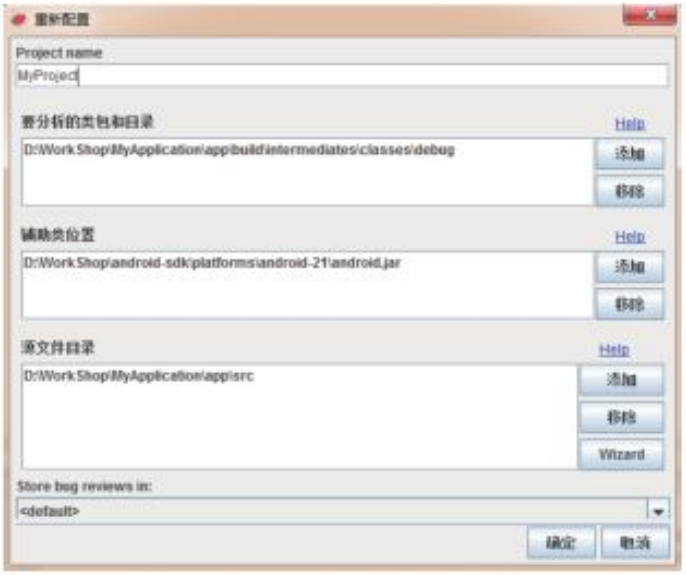


图 5-4　FindBugs UI 界面配置

按“确定”按钮进行分析即可查看结果，如图 5-5 所示。

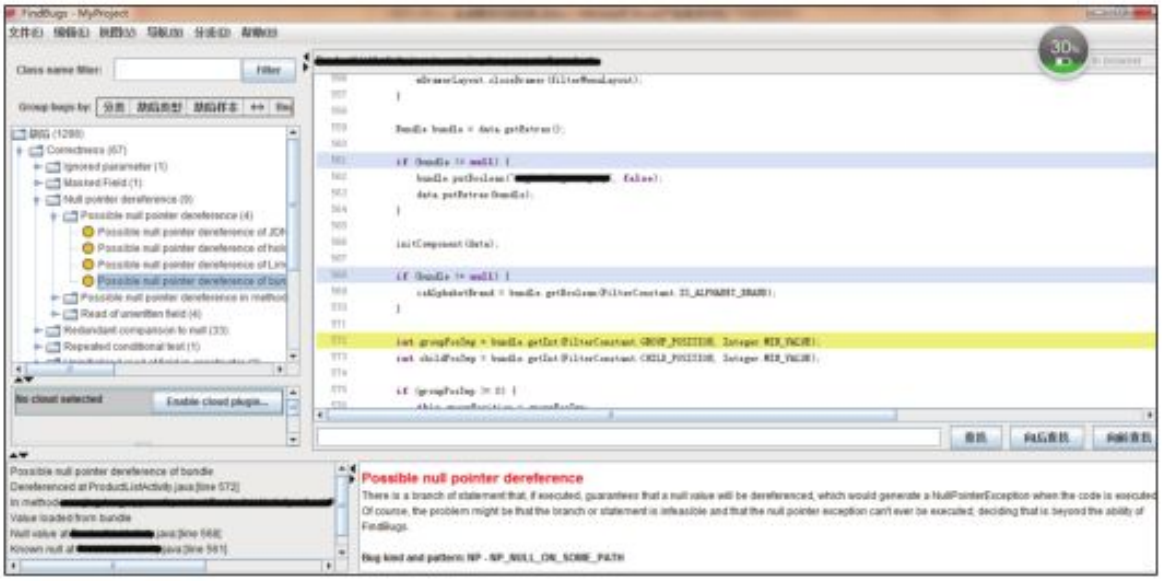


图 5-5　用 FindBugs UI 界面查看分析结果

完成后我们可以通过“文件 -> 另存为”保存分析配置，如图 5-6 所示。

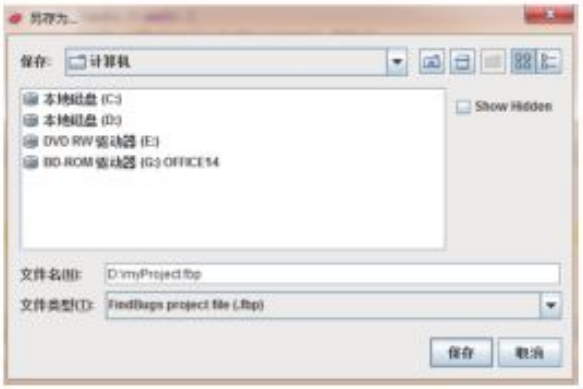


图 5-6　保存 fbp 文件

如果用文本编辑器打开这个 fbp 文件，可以发现它是一个 xml 格式的文件：

```
<Project projectName="MyProject">
    <Jar>要分析的类包和目录</Jar>
    <AuxClasspathEntry>辅助类位置</AuxClasspathEntry>
    <SrcDir>源文件目录 </SrcDir>
</Project>
```

这个 fbp 文件可以在 FindBugs UI 中打开，也可以用下面介绍的命令行方式打开。

3. 命令行方式

我们可以采用命令行方式使用 FindBugs。这种方式适合在持续集成中使用。下面是使用命令行调用 FindBugs 3.0.0 的一个例子：

```
java -jar %FINDBUGS_HOME%/lib/findbugs.jar -textui -output c:\temp\findbugsResult.xml
    -xml -project "C:\temp\myProject.fbp" -exclude C:\temp\fbFilter.xml
```

阴影部分依次为：

- Findbugs.jar 的路径。
- 分析结果文件的存储路径。
- 分析结果文件格式，支持 html、xml。
- 分析配置文件（格式见上一小节“UI 界面方式”）。
- 过滤器路径（格式见下一小节“过滤器”）。

执行完毕后能在输出文件中看到结果。以 xml 格式为例，打开输出文件，可以发现其中包含 BugInstance 的集合，包括问题类型、问题类名和代码行等。如图 5-7 所示。我们可以通过解析该文件来进行后续的扩展应用。

图 5-7　查看 XML 格式的 FindBugs 分析结果

4. 过滤器

过滤器是一个 xml 文件。我们在上述三种使用方式中都可以添加过滤器来过滤掉经过评估不用检查的部分，以便节省分析时间，提高测试和开发的效率。如下是一个 FindBugs 过滤器的例子，它过滤掉了对编译过程中产生的临时 Java 文件的检查以及一些特定模式的检查：

```
<FindBugsFilter>
     <Match>
         <Source name="~R\.java" />
     </Match>
     <Match>
     <Bug pattern="EI_EXPOSE_REP2"/>
     </Match>
```

```
 <Match>
<Bug pattern="EI_EXPOSE_REP"/>
</Match>
 <Match>
<Bug pattern="SF_SWITCH_NO_DEFAULT"/>
</Match>
 <Match>
<Bug pattern="SF_SWITCH_FALLTHROUGH"/>
</Match>
 <Match>
<Bug pattern="RCN_REDUNDANT_NULLCHECK_OF_NULL_VALUE"/>
</Match>
 <Match>
<Bug pattern="ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD"/>
</Match>
 <Match>
<Bug pattern="DMI_HARDCODED_ABSOLUTE_FILENAME"/>
</Match>
 <Match>
<Bug pattern="SE_COMPARATOR_SHOULD_BE_SERIALIZABLE"/>
</Match>
 <Match>
<Bug pattern="DM_EXIT"/>
</Match>
  <Match>
<Bug pattern="NN_NAKED_NOTIFY"/>
</Match>
  <Match>
<Bug pattern="UW_UNCOND_WAIT"/>
</Match>
<Match>
<Bug pattern="WA_NOT_IN_LOOP"/>
</Match>
 <Match>
<Bug pattern="BX_UNBOXING_IMMEDIATELY_REBOXED"/>
</Match>
 <Match>
<Bug pattern="BX_BOXING_IMMEDIATELY_UNBOXED_TO_PERFORM_COERCION"/>
</Match>

</FindBugsFilter>
```

5. 代码屏蔽

开发人员在评估特定问题后，也可以通过在代码的方法中或者类名前面增加一个标记来屏蔽某处特定的问题。例如：

```
@SuppressWarnings("DM_STRING_CTOR")
```

这样在代码扫描的时候 FindBugs 会忽略所指定的特定问题。由于代码屏蔽可能会让一些问题被掩盖，所以屏蔽内容最好能经过开发小组负责人的评审。

5.1.1.2　Lint 扫描工具

Lint 是官方的 Android 静态代码扫描工具。与 FindBugs 相比，Lint 能扫描出 Android 的特定问题，例如无用的资源文件残留，Android API 版本兼容问题，布局文件中的问题，等等。但是 FindBugs 能够发现的一些 Java 代码层面的问题 Lint 目前并不能发现。因此在实际使用中，我们可以考虑结合两种静态代码扫描工具的优点，共同使用。

对于使用 Gradle 来构建和集成的项目，我们可以在项目目录下直接执行下列命令行进行 Lint 扫描：

```
gradle lint
```

结束后会在 build\outputs 下生成 lint-results.html 和 lint-results.xml 两种格式的分析结果文件，图 5-8 是生成 html 报告的一个例子。

图 5-8　Lint HTML 格式的分析报告

对于没有使用 Gradle 来构建和集成的项目我们也可以使用 lint 命令进行静态代码扫描。

1. 代码屏蔽

跟 FindBugs 类似，lint 也提供了屏蔽特定错误的方式。我们可以在类或者方法前添加

如下标记屏蔽特定的错误：

```
@SuppressLint( "NewApi" )
```

2.XML 文件中的屏蔽

由于 lint 还会检查布局文件，我们有时候也需要对布局的特定错误进行屏蔽。例如：

```
<LinearLayout tools:ignore=" MergeRootFrame" >
```

3.Gradle 文件全局屏蔽

对于一个 gradle 文件，我们也可以在 gradle 文件中进行全局性的屏蔽。例如：

```
android{
    lintOptions{
            disable 'MergeRootFrame'
    }
}
```

更多有关 lint 的用法可以参考 lint 官网：

http://tools.android.com/tips/lint

5.1.2 针对 iOS 的静态代码扫描和分析

以上介绍了 Android 代码的静态扫描工具，接下来我们来看看 iOS 对应的工具。iOS 开发环境是高度集成的，这里的静态扫描也依赖于 Xcode 本身提供的功能。下面主要讨论常见的代码问题，以及一个我们自行开发的冗余资源扫描脚本。

5.1.2.1 iOS Analyze 静态分析

Xcode 提供了一个 Analyze 功能，可以静态分析代码中的潜在问题，如图 5-9 所示。

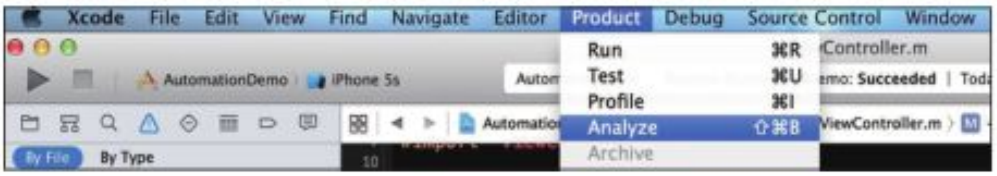
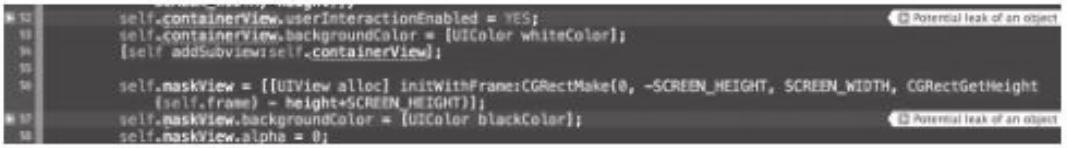


图 5-9 选择 Analyze 模式编译

通过静态分析发现的常见问题示例如下。

A. 未被使用的变量，如下所示：

B. 内存泄漏风险，如下所示：



C. 未遵循框架规范编码如下所示：

D. 不合适的类型返回如下所示：

E. API 误用如下所示：

F. 逻辑错误如下所示：

G. 空指针引用如下所示：

以上列举了一些常见的代码问题。静态代码扫描能够发现不少潜在的风险，不过它的扫描结果有时并不准确，需要我们仔细检查判断。尤其是在编码完成后，开发人员需要对自己的代码做一遍静态扫描，及时发现问题，以便在测试阶段不会暴露更多问题。

5.1.2.2　iOS 冗余扫描脚本

在 iOS 的项目工程中，由于版本的更新，日积月累，充斥了大量冗余的无用的代码文件、布局文件、图片文件和声音文件等。无用的代码可以通过静态扫描和覆盖率报告及时发现，那么资源文件怎么能够快速扫描发现呢？

我们知道，所有的资源文件都是在项目工程中包含的，开发时通过资源文件名去引用这些资源（布局文件同样如此），所以在 Xcode 里搜索资源名称，如果有被引用，就能被搜到（有些代码是通过拼接字符串去引用资源名的，那就只能通过部分关键词去搜），如图 5-10 所示。

图 5-10　Xcode 的全局搜索功能

因此如果没有搜到，就可以判断这个资源没有被使用到。那是不是就完成了呢？当然不是，因为资源没有被引用到，但仍有可能被打包到 App 里面。因此，我们还要检查该文件是否被编译打包了。我们可以到 Xcode-Build Phases 标签下的 Copy Bundle Resources 下查看该资源是否被打包了，如图 5-11 所示。

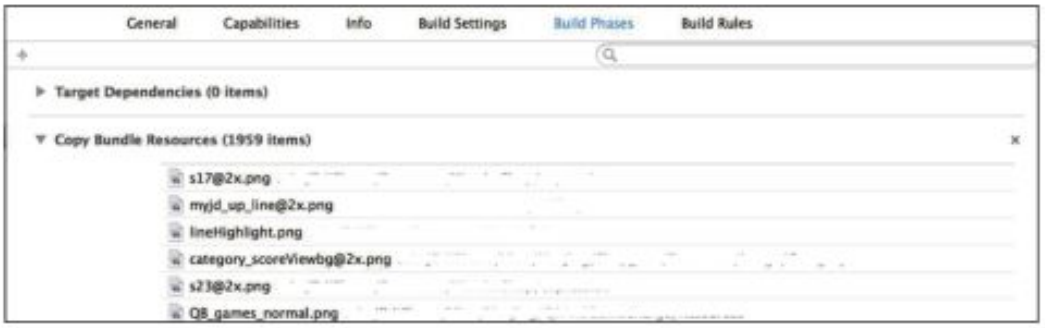


图 5-11　项目工程的编译资源列表

知道了这个方法，我们就可以通过 shell 脚本去快速遍历检查项目工程的这些资源文件，我们编写了一个 shell 脚本，供参考。

参数 1：参考的项目工程目录，包括 .m，.xib 文件。

参数 2：被扫描的资源文件目录，包括 .png，.jpg 文件。

参数 3：项目 .xcodeproj/project.pbxproj 文件路径。

```
#!/bin/sh

if [ -f ~/Desktop/resource_scan_result.txt ];then
     rm -f ~/Desktop/resource_scan_result.txt
fi
#找到所有的资源文件
files='find $1 -name "*.m" -o -name "*.xib" -o -name "*.mm" -o -name "*.plist"'
#遍历所有资源文件
```

```
for png in 'find $2 -name "*.png" -o -name "*.jpg" -o -name "*.wav"'
do
    basename='basename $png'
    if [ "${basename##*.}" == "png" ];then
    #过滤文件名后缀
        echo $basename|grep -q @2x.*.png
        if [ $? -eq 0 ];then
            name=${basename%%@2x*.png}
        else
            echo $basename|grep -q @3x.*.png
            if [ $? -eq 0 ];then
                name=${basename%%@3x*.png}
            else
                name=${basename%%.png}
            fi
        fi
    elif [ "${basename##*.}" == "jpg" ];then
        echo $basename|grep -q @2x.*.jpg
        if [ $? -eq 0 ];then
            name=${basename%%@2x*.jpg}
        else
            echo $basename|grep -q @3x.*.jpg
            if [ $? -eq 0 ];then
                name=${basename%%@3x*.jpg}
            else
                name=${basename%%.jpg}
            fi
        fi
    elif [ "${basename##*.}" == "wav" ];then
        name=${basename%%.wav}
    fi
    #过滤那些后缀用数字拼接的
    name=${name%%[0-9]*}
    if grep -q $name $files;then
        echo "$png"  找到了~~
    else
        if grep -q $basename $3;then
            echo "$png" 没有被引用到 >> ~/Desktop/resource_scan_result.txt
        fi
    fi
done
if [ -f ~/Desktop/resource_scan_result.txt ];then
    echo **********扫描结束，请到桌面下的resource_scan_result.txt下查看未被引用的资源
        **********
else
    echo **********扫描结束，没有发现未被引用的资源**************
fi
```

执行完毕后，所有未被引用的资源都会标记出来写入报告文件中。不过该方法没法检

查那些纯动态引用资源文件的代码。

5.2 代码覆盖率分析

代码覆盖率分析是一个很常用的测试手段，主要用于帮助我们评估测试执行的代码覆盖情况。针对移动互联网的产品，代码覆盖率仍然是一个很重要的参考。由于代码覆盖率是和语言相关的，下面我们分别介绍下 Android 和 iOS 代码覆盖率的技术实现，然后讨论如何高效地利用覆盖率工具。

5.2.1 Android 代码覆盖率技术方案

获得 Android 代码覆盖率数据，主要有两种方式：

- 使用现有的代码覆盖率框架，例如 emma 和 jacoco。
- 基于 asm 从头开始编写自己的代码覆盖率框架。

emma 和 jacoco 最初都是为桌面应用开发的，没有现成的 Android 支持，如果要在 Android 上使用需要进行一定的二次开发。asm 是处理 Java 字节码的一个类库，可用于进行代码覆盖率所必须的字节码插桩工作。

无论使用哪种方法，测试步骤都是类似的，测试步骤为：

1）生成插桩的 App 的 apk 包。

2）进行 UI 自动化测试或者手动测试。

3）在 Android 手机上生成代码覆盖率原始数据文件。

4）将原始数据文件导出到计算机进行处理并生成最终的报告。

如果是使用 Ant 进行编译，也可以用 jacoco 来做代码覆盖率，具体方法可以参考 jacoco 的官方文档（http://www.eclemma.org/jacoco/trunk/doc/ant.html）。这里用 gradle（http://gradle.org/）环境为例的原因是：

- 目前看来 gradle 是未来的趋势。
- 当前关于 gradle 文档较少，更加难以找到如何使用 gradle+jacoco 进行 Android 代码覆盖率的详细资料。

另外，部分测试人员之前可能使用过或者听说过 emma 这个代码覆盖率工具。我们之所以不介绍 emma，是因为 emma 已经长时间处于无人维护的状态，其最新版本是 2005 年的。而 jacoco 则是当前的主流 Java 代码覆盖率工具。

下面以 gradle2.2.1 编译环境为例，介绍如何使用 jacoco 进行 Android App 的代码覆盖率测试。

1. 准备工作

1）从官网（http://www.eclemma.org/jacoco/）下载 jacoco-0.7.1.201405082137.zip。

2）解压后找到 lib\jacocoagent.jar，解压 jacocoagent.jar 后导出 class 文件放入一个目录中。

3）Android SDK 中获取 android.jar，解压后导出所有 class 文件放入上述同一目录（主要是使用了 Android 的 LogCat 进行日志输出便于调试）。

4）从官网下载 Agent.java 源代码进行修改。这里修改的原因是，默认 jacoco 触发生成代码覆盖率测试文件的代码是通过 Runtime.getRuntime().addShutdownHook 来 hook 的。这个 hook 点在 Android 下是不能由进程关闭而触发的。因此我们必须进行修改。可以考虑在后台开启线程定期生成原始数据文件。一个可能的修改方案如下：

```
package org.jacoco.agent.rt.internal_932a715;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.net.InetAddress;
import java.util.concurrent.Callable;
import org.jacoco.agent.rt.IAgent;
import org.jacoco.agent.rt.internal_932a715.core.JaCoCo;
import org.jacoco.agent.rt.internal_932a715.core.data.ExecutionDataWriter;
import org.jacoco.agent.rt.internal_932a715.core.runtime.AbstractRuntime;
import org.jacoco.agent.rt.internal_932a715.core.runtime.AgentOptions;
import  org.jacoco.agent.rt.internal_932a715.core.runtime.AgentOptions.
OutputMode;
import org.jacoco.agent.rt.internal_932a715.core.runtime.RuntimeData;
import org.jacoco.agent.rt.internal_932a715.output.FileOutput;
import org.jacoco.agent.rt.internal_932a715.output.IAgentOutput;
import org.jacoco.agent.rt.internal_932a715.output.NoneOutput;
import org.jacoco.agent.rt.internal_932a715.output.TcpClientOutput;
import org.jacoco.agent.rt.internal_932a715.output.TcpServerOutput;
import android.util.Log;
import java.io.File;
public class Agent implements IAgent, Runnable

...
public void startup() {
          try {
          Log.w("mylog ", "called startup");
                  String sessionId = options.getSessionId();
                  if (sessionId == null) {
                          sessionId = createSessionId();
                  }
                  data.setSessionId(sessionId);
                  output = createAgentOutput();
                  output.startup(options,  data);
```

```
                    if (options.getJmx()) {
                            jmxRegistration = new JmxRegistration(this);
                    }
                            // Start a new thread to dump coverage data
    Log.w("mylog", "start thread");
                            thread=new Thread(new Agent(options, null));
                            thread.start();
            }
 catch (final Exception e) {
                    logger.logExeption(e);
            }
    }
    private static Thread thread;
    private static java.util.Random rl=new java.util.Random(999999999);
public void run() {
    try{
    int i=0;
    int uid=rl.nextInt();
      while(true){

org.jacoco.agent.rt.internal_932a715.Agent.getInstance().options.setDestfile
     ("sdcard/jacoco/"+uid+"_"+i+".exec");
      Log.w("mylog", "setDestfile");
      this.output = createAgentOutput();
      this.output.startup(org.jacoco.agent.rt.internal_932a715.Agent.getInstance().
          options,
      org.jacoco.agent.rt.internal_932a715.Agent.getInstance().data);
    this.dump(true);
    Log.w("mylog ", "called dump to generate new jacoco.exec");
    java.lang.Thread.sleep(30000);
    i++;
       }
    }
    catch(Exception ex)
    {
    Log.w("mylog ", ex);
    }
}
```

5）使用下列命令进行编译：

```
Javac -source 1.6 -target 1.6 -cp <步骤2和步骤3放置的目录> Agent.java
```

6）用生成的 .class 文件替换 jacocoagent.jar 内的相应文件。修改 jacocoagent.jar 内 MANIFEST.MF 文件为：

```
Manifest-Version: 1.0
Premain-Class: org.jacoco.agent.rt.internal_932a715.PreMain
Archiver-Version: Plexus Archiver
Build-Jdk: 1.5.0_22
Built-By: godin
```

```
Created-By: Apache Maven
Implementation-Title: JaCoCo Java Agent
Implementation-Version: 0.7.1.201405082137
Implementation-Vendor: Mountainminds GmbH & Co. KG
```

将修改后的jacocoagent.jar放入org.jacoco.agent-0.7.1.201405082137.jar（从下载的jacoco-0.7.1.201405082137.zip解压后可获得）。在下列路径放置修改后的org.jacoco.agent-0.7.1.201405082137.jar：

```
<Android SDK目录>\extras\android\m2repository\org\jacoco\ org.jacoco.
    agent\0.7.1.201405082137
```

7）在下列路径放置官网上下载的压缩包中的lib\org.jacoco.ant-0.7.1.201405082137.jar：

```
<Android SDK目录>\extras\android\m2repository\org\jacoco\ org.jacoco.
    ant\0.7.1.201405082137
```

注意　由于gradle代码变动，上述方法不一定有效。这时候需要借助其他工具找出gradle执行时使用的org.jacoco.ant-0.7.1.201405082137.jar是哪个并进行替换。例如用Windows下的Process Monitor工具等。

2. 插桩

准备工作完成后我们需要对项目进行一些配置以使用jacoco。

1）在项目的build.gradle增加下列配置：

```
apply plugin: 'jacoco'
android {
buildTypes {
    debug{
        testCoverageEnabled true
...
 }
...
}
```

注意，如果需要测试引用的类库的代码覆盖率时，需要在所引用类库项目的release而不是debug下添加配置：

```
buildTypes {
    release{
        testCoverageEnabled true
...
 }
```

因为gradle 2.2.1编译主工程debug版本时，拉取的是所引用类库的release版本的classses.jar。

2）运行 gradle assembleDebug 命令生成插桩后的 apk debug 包。

3）安装 apk 到手机。

完成后我们可以进行 UI 自动化测试或者手动测试。

3. 获取代码覆盖率文件

测试完毕后我们需要等待覆盖率文件生成。注意上面的实现是每隔 30 秒生成一个数据文件，所以可能需要等待 30 秒。

另外需要注意，由于生成代码覆盖率文件的代码是在进程后台线程运行，需要强制关闭进程才不会有新代码覆盖率文件生成。结束测试时请记得强制杀死进程，否则会有很多无用文件生成。

1）使用下列命令导出目录下所有代码覆盖率文件到本地 jacoco 目录下：

```
adb pull sdcard/jacoco jacoco
```

2）如果没有发现文件生成，请检查 App manifest.xml 是否包含写文件权限：

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

4. 合并代码覆盖率文件

在命令行当前文件夹下增加如下的 build.xml 并运行 ant merge 命令进行合并。其中高亮部分可根据实际情况替换。从上往下依次为：

- 存放 jacocoant.jar 的路径。
- 合并后的代码覆盖率文件路径。
- 存放待合并代码覆盖率文件的目录。

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="MainActivity" default="help" xmlns:jacoco="antlib:org.jacoco.
    ant">
    <taskdef uri="antlib:org.jacoco.ant" resource="org/jacoco/ant/antlib.xml">
        <classpath path="D:/WorkShop/jacoco/lib/jacocoant.jar"/>
    </taskdef>
<target name="merge">
    <jacoco:merge destfile=" C:/temp/merged.exec">
        <fileset dir="C:/temp/jacoco" includes="*.exec"/>
    </jacoco:merge>
</target>
</project>
```

5 . 生成代码覆盖率报告

生成代码覆盖率的部分我们需要用到 Apache ant。你可以从官网下载（http://ant.apache.org/）。

1）在项目根目录下放置如下 build.xml 文件。其中高亮部分可根据实际情况修改。从上

往下依次为：

- 存放 jacocoant.jar 的路径。
- 代码覆盖率文件路径。
- .class 文件存储路径。如果是 debug 包，需要指定 debug 的 .class 文件。
- 源代码存放路径。
- 报告生成文件夹的路径（相对 build.xml 路径）。

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="MainActivity" default="help" xmlns:jacoco="antlib:org.jacoco.
    ant">
    <taskdef uri="antlib:org.jacoco.ant" resource="org/jacoco/ant/antlib.xml">
        <classpath path="D:/WorkShop/jacoco/lib/jacocoant.jar"/>
    </taskdef>
    <target name="report">

    <jacoco:report>

        <executiondata>
            <file file=" C:/temp/merged.exec "/>
        </executiondata>

        <structure name="JD Project">
            <classfiles>
                <fileset dir="build/intermediates/classes/debug"/>
            </classfiles>
            <sourcefiles encoding="UTF-8">
                <fileset dir="src"/>
                </sourcefiles>
        </structure>

    <html destdir="report"/>
    </jacoco:report>
    </target>
</project>
```

2）命令行切换到项目根目录，运行 ant report。

3）按照上面配置后可以在项目下面发现一个 report 文件夹，其中 index.html 即为代码覆盖率报告。可以点击查看详细情况，包括每个包的覆盖率情况，如图 5-12 所示。

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.example.test.myapplication	96 of 262	63%	11 of 24	54%	25	38	29	77	15	26	11	13
Total	96 of 262	63%	11 of 24	54%	25	38	29	77	15	26	11	13

图 5-12　代码覆盖率汇总结果

也可以看到某个文件中的代码的覆盖率结果，如图 5-13 所示。

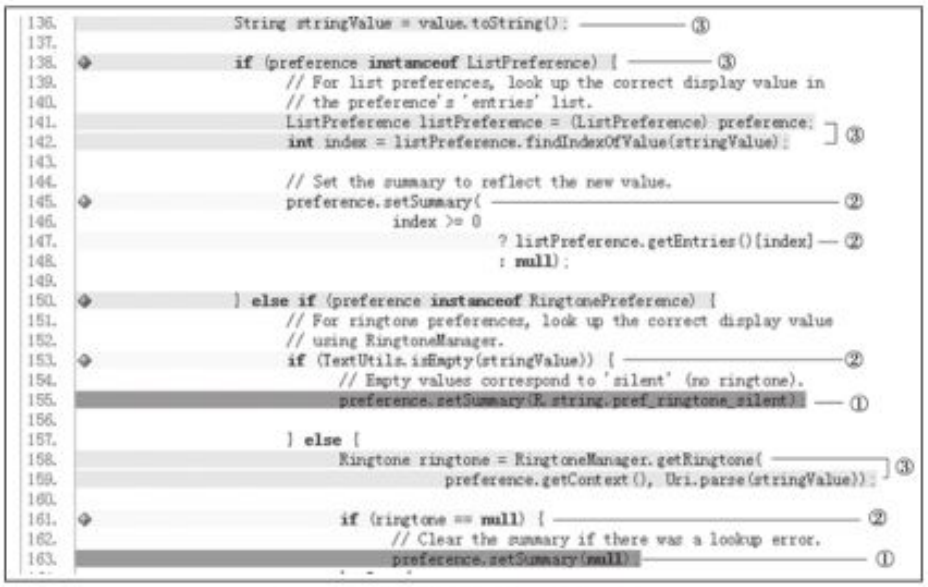


图 5-13 具体文件的代码覆盖率结果

其中阴影①部分为没有执行到的代码，阴影②部分为有一部分没有走到的分支，阴影②部分表示执行到的代码。

5.2.2 iOS 代码覆盖率技术方案

上面我们结合实例解释了如何生成 Android 的代码覆盖率，接下来我们看看 iOS 平台的实践方法。

1. iOS 的覆盖率介绍

Xcode 自带覆盖率编译选项，通过开启了覆盖率选项，编译器会添加覆盖率参数，通过插桩的方式收集代码的执行情况。关于 C 的覆盖率原理，大家可以自行查阅资料，这里不再赘述。iOS 会为项目中每个类生成两个文件，我们通过关联这两个文件获取覆盖率信息：

- gcno 文件：包含基本的块信息，以及代码行与块的映射关系。
- gcda 文件：包含代码行执行的情况，以及覆盖率的信息归纳。

2. 覆盖率数据的查看

gcno 和 gcda 文件是无法直接查看的，需要我们用工具去生成 .info 文件。.info 文件不但聚合了项目中所有类的 gcno 文件和 gcda 文件，同时将覆盖率的数据和 block 作了映射。我们通过报告制作工具，就可以直接利用 info 文件生成覆盖率报告。

3.iOS 代码覆盖率的生成过程

这里我们以 Xcode6 为例，详细介绍如何编译带有代码覆盖率的项目工程，以及如何生

成覆盖率报告。

1）在 Build Setting 设置里找到 Instrument Program Flow 和 Generate Test Coverage Files，把它们的值设成 YES，如图 5-14 和图 5-15 所示。

图 5-14　设置 Instrument Program Flow

图 5-15　设置 Generate Test Coverage Files

2）在 Info 标签中，添加如下选项：

- Application supports iTunes files sharing（App 共享文件夹设置），用来存放覆盖率数据并导出，如图 5-16 所示。

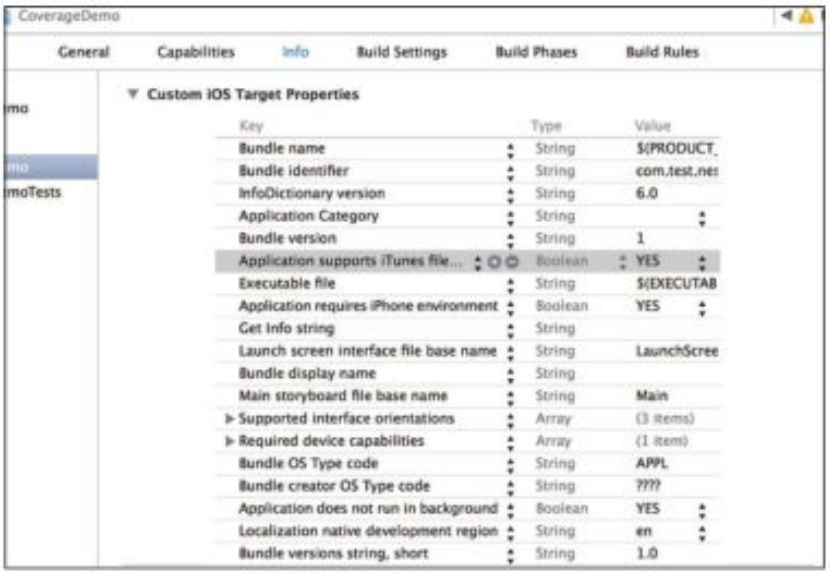


图 5-16　在 Info 标签中添加 Application supports iTunes file sharing

这个选项是针对真机测试覆盖率数据的，如果是模拟器测试环境，则无须设置此选项。

- Application does not run in background（不允许 App 在后台运行），通过按 home 键来

终止 App 运行使得覆盖率数据能够写入到文件 . 设置如图 5-17 所示。这里需要注意一个问题，如果你的 App 是有跳转功能的（比如 QQ 授权登录），那么该设置会让跳转功能失效，并且无法获取覆盖率数据。如果你需要跳转逻辑的覆盖率数据，就不能设置此选项。那么如何让 App 退出呢，可以使用 exit(0) 或者故意以 crash 方式让 App 自行终止。若是使用如此方式退出 App，需要捕获 exception 时，将覆盖率数据写入：

```
extern void __gcov_flush(void);
__gcov_flush();
```

这样 App 在切到后台时就不会终止，而终止 App 让覆盖率写入的方式变成了自定义让 App 退出的方式。当然这里只是我们举的一个例子，实际中你可以在任何指定的地方调用该方法来实现覆盖率数据的写入，只是需要谨慎选择入口并通知测试人员该入口在 UI 上的位置。

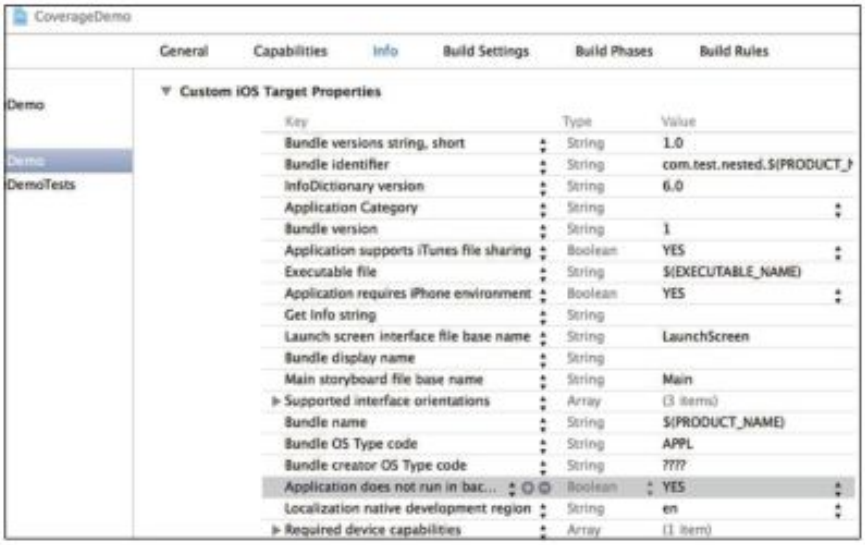


图 5-17　在 Info 标签中添加 Application does not run in background

3）设置覆盖率文件保存位置。

由于模拟器测试环境是纯 Mac 环境，所以覆盖率数据也存在 Mac 上，它的文件位置和 gcno 文件其实是在一起的（后面会提到）。

如果是真机测试环境，覆盖率数据是存在设备上的，我们需要指定覆盖率数据的保存位置，以便导出。由于无法访问苹果设备的文件结构（越狱的设备可以），我们只能访问苹果提供的针对每个 App 的对外的共享文件夹，我们可以在 iTunes 里查看到这些文件夹（前提是 App 设置了步骤 2 中的功能文件夹的选项）。

我们可以在 main 文件中设置覆盖率数据的保存位置，如图 5-18 所示，在 main 方法中添加：

```
const char* prefix = "GCOV_PREFIX"; //设置覆盖率前缀变量
const char* prefixValue = [[NSHomeDirectory() stringByAppendingPathComponent:@"
Documents/coverageData"] cStringUsingEncoding:NSASCIIStringEncoding];
          //设置覆盖率数据保存在共享文件夹下的coverageData目录下
const char* prefixStrip = "GCOV_PREFIX_STRIP";
          //设置跳过文件夹变量
const char* prefixStripVaule = "1";
          //设置跳过覆盖率父文件夹层数，由于默认的覆盖率数据文件夹层数很深，这边需要过滤
setenv(prefix,  prefixValue,  1);
setenv(prefixStrip,  prefixStripVaule,  1);
```

```
//
//  main.m
//  CoverageDemo
//
//  Created by 潘晓明 on 2/11/15.
//  Copyright (c) 2015 com.test.shawntest. All rights reserved.
//

#import <UIKit/UIKit.h>
#import "AppDelegate.h"

int main(int argc, char * argv[]) {
    const char* prefix = "GCOV_PREFIX";
    const char* prefixValue = [[NSHomeDirectory()
        stringByAppendingPathComponent:@"Documents/coverageData"]
        cStringUsingEncoding:NSASCIIStringEncoding];
    const char* prefixStrip = "GCOV_PREFIX_STRIP";
    const char* prefixStripVaule = "1";
    setenv(prefix, prefixValue, 1);
    setenv(prefixStrip, prefixStripVaule, 1);
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate
            class]));
    }
}
```

图 5-18　main 文件中设置覆盖率数据的保存位置

4）编译工程。我们可以选择模拟器或者真实设备来编译安装项目工程。真机测试的情况下，确保开发者证书和 provision 文件选择正确。这里的编译设置没有太多限制，debug 和 release 都支持（确保步骤 1 中设置的标志位应用到了你的编译设置）。当然，我们也可以将带有覆盖率设置的项目工程打成 ipa 包，分发给测试人员，进行覆盖率测试。每个人的覆盖率测试结果数据都保存在他的测试机上，我们在后面的制作覆盖率报告中会提到，如何汇总多个覆盖率数据。

5）测试。带有覆盖率的 App 和一般的 App 没有任何功能上的区别。我们只需按照我们常规的测试方法，进行黑盒测试即可。测试完成后，通过点击 home 键让 App 切到后台，此时 App 会被终止，同时覆盖率数据已写入文件中。

当然，我们可以重新打开 App 继续测试，新的覆盖率数据会继续在老的覆盖率数据上叠加。

6）获取覆盖率数据。测试完成后，我们需要拿到覆盖率数据，即 gcno 和 gcda 文件。我们可以点击 File → Project Setting 来查看项目工程的编译文件存放路径，如图 5-19 和图 5-20 所示。

一般如果我们没有特殊指定的话，默认路径是在 /Users/< 用户名 >/Library/Developer/Xcode/DerivedData/ 下，找到刚才编译的项目工程文件夹，如图 5-21 所示。

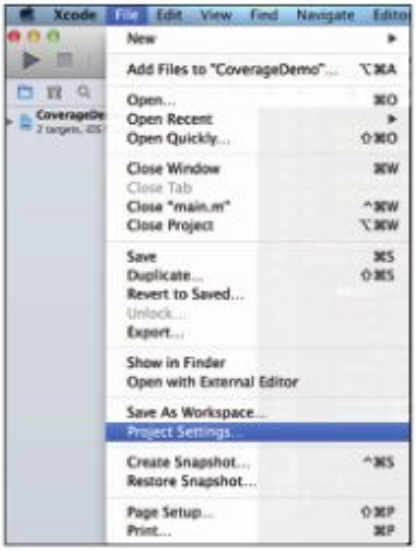


图 5-19 查看项目工程设置

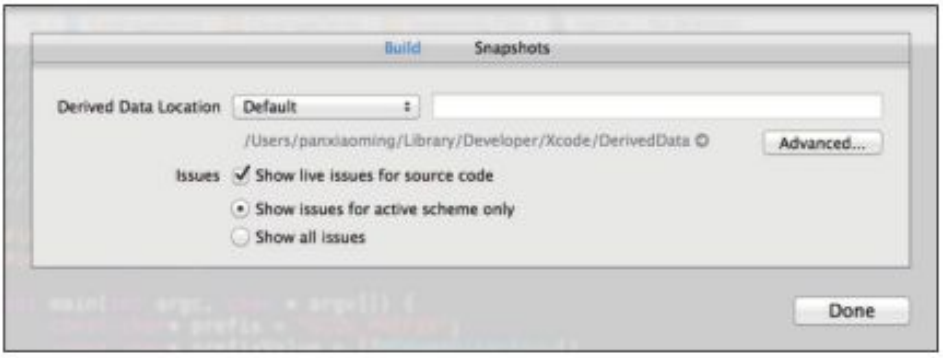


图 5-20 项目工程的编译输出文件路径设置界面

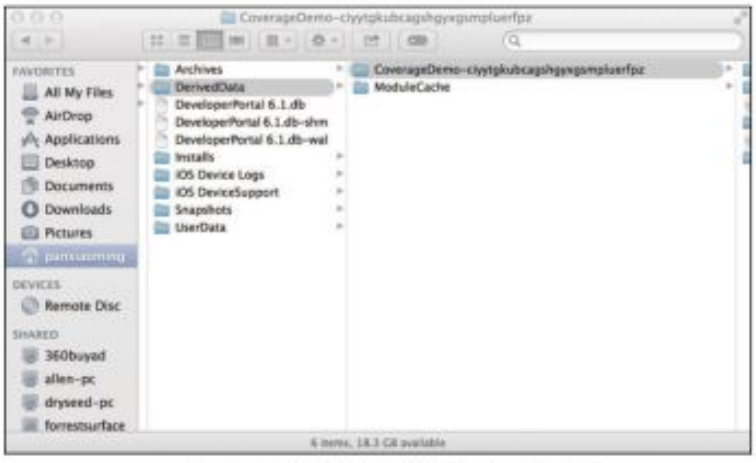


图 5-21 项目工程的编译输出文件路径

这个文件夹的层级很深，我们需要找到 gcno 文件，相对路径是 :./Build/Intermediates/< 项目名称 >.build/Debug-< 真机或模拟器标识 >/< 项目名称 >.build/Objects-normal/< 指令名称 >/，如图 5-22 所示。

注意　真机标识 :iphoneos 模拟器标识 :iphonesimulator。

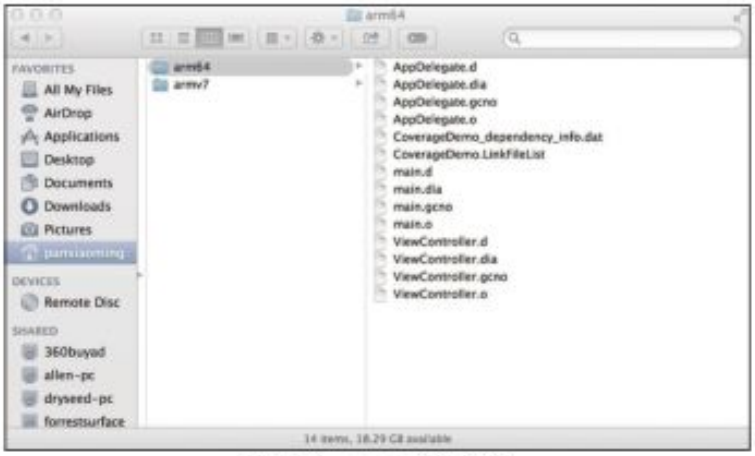


图 5-22　gcno 文件路径

我们将所有的 gcno 文件拷贝到一个临时文件夹中，比如在桌面上新建一个 tmp 文件夹，将所有 gcno 文件拷贝进去。

如果是模拟器环境的话，覆盖率数据文件 gcda 文件也同时在这里展示，我们可以将 gcda 和 gcno 一起复制到临时文件夹中。

如果是真机测试的话，我们把设备连接 mac，打开 iTunes，找到测试 App 的共享文件夹，如图 5-23 所示，将文件夹拷贝出来。

图 5-23　iTunes 的 App 共享文件夹

这个文件夹和之前一样，层级也比较深。这里提一下我们之前在 main 中添加的代码，其中有一项是关于父文件夹跳过的变量，如果我们设置了值，这里就过滤掉了值相应的文件夹层数。我们找到最里层的文件夹，可以看到 gcda 文件，将所有的 gcda 文件拷贝到刚才的临时文件夹中，如图 5-24 所示。

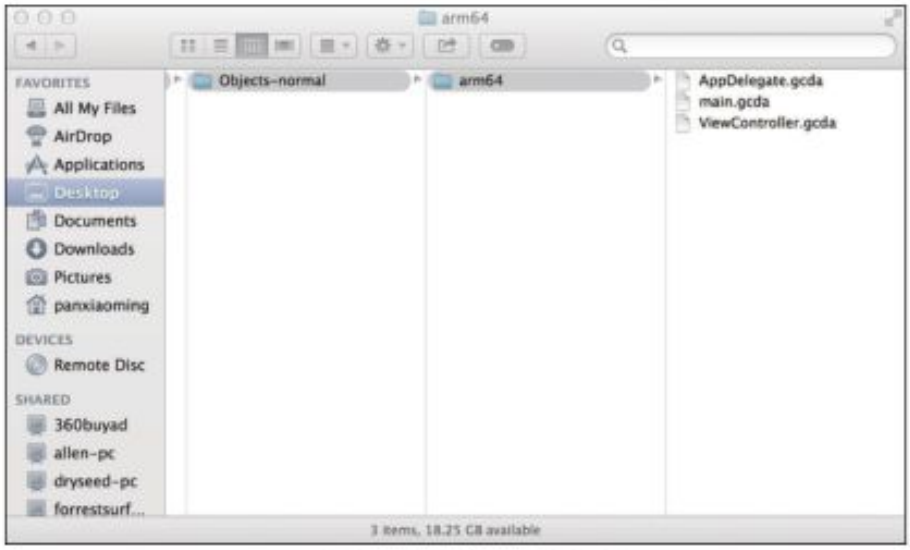


图 5-24 共享文件夹中的覆盖率数据路径

7）制作覆盖率报告。Lcov 是一个能够整合覆盖率数据，制作 info 文件，并且产出覆盖率 html 报告的工具。具体详情，可以参考。http://ltp.sourceforge.net/coverage/lcov.php

接着我们使用 lcov 工具中的 lcov 命令来制作 info 文件，具体命令：

```
lcov -c -d <临时文件夹路径> -b <项目工程路径> -o <输出info文件路径>
```

如下所示：

```
[panxiaoming---]>>/Users/panxiaoming/Dev/Tools/lcov-1.11/bin/lcov -c -d ~/Desktop/tmp -b /Use
rs/panxiaoming/Dev/IOS_work/CoverageDemo -o ~/Desktop/coverage.info
```

我们可以过滤掉项目中的某些文件夹，这样在制作报告的时候就不会引入：

```
lcov -remove <当前info文件路径> <需要过滤的项目文件夹路径> ... -o <新info文件路径>
```

如下所示：

```
[panxiaoming---]>>/Users/panxiaoming/Dev/Tools/lcov-1.11/bin/lcov --remove ~/Desktop/coverage
.info /Users/panxiaoming/Dev/IOS_work/CoverageDemo/CoverageDemoTests -o ~/Desktop/newCoverage
.info
```

我们也可以整合多个测试人员的覆盖率数据：

```
lcov -a <第一个info文件路径> -a <第二个info文件路径> -a ... -o <合并完后的info文件路径>
```

如下所示：

```
[panxiaoming---]>>/Users/panxiaoming/Dev/Tools/lcov-1.11/bin/lcov -a ~/Desktop/1.info -a ~/De
sktop/2.info -o ~/Desktop/combine.info
```

最后我们就可以生成覆盖率报告了，可以使用 lcov 工具中的 genhtml 命令来生成 html

报告，具体命令：

```
genhtml -t <报告标题> <info文件路径> -o <输出报告路径>
```

如下所示：

```
[panxiaoming-->>/Users/panxiaoming/Dev/Tools/lcov-1.11/bin/genhtml -t "覆盖率报告演示" ~/De
sktop/coverage.info -o ~/Desktop/report
```

8）查看报告。我们打开生成的报告文件夹中的 index.html，查看总体覆盖率报告，如图 5-25 所示。

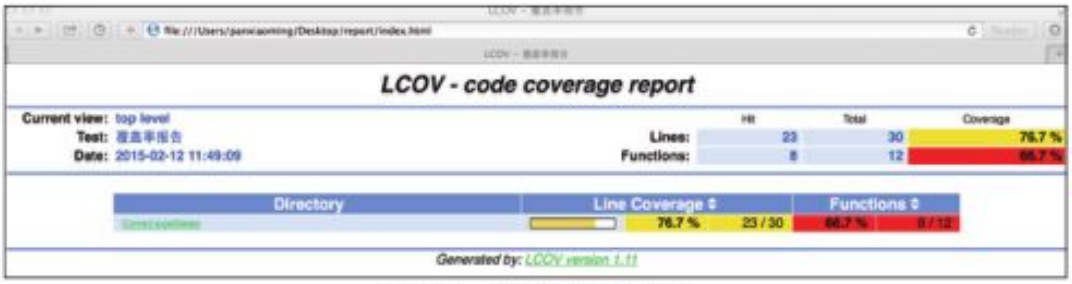


图 5-25　覆盖率报告根目录

总报告以文件夹的方式展现，点击后可以查看该文件夹下的文件覆盖情况，如图 5-26 所示。覆盖率报告按两种纬度呈现：代码行覆盖和方法数覆盖。

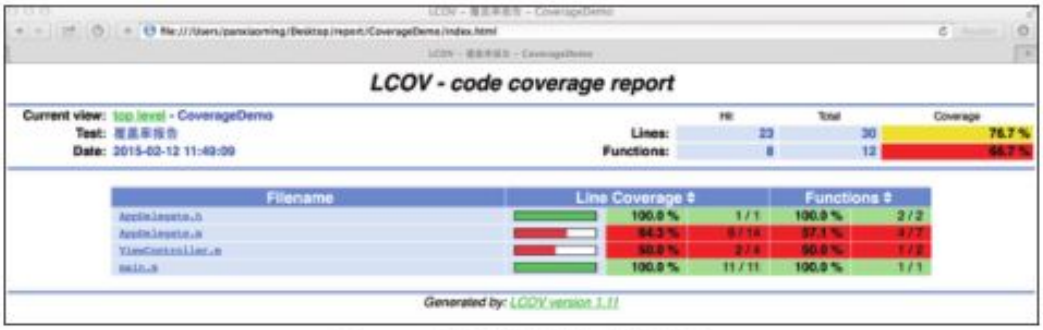


图 5-26　覆盖率报告文件夹目录

点击文件可以查看具体的覆盖情况。阴影①代码表示已经覆盖到的代码行，阴影②代码行表示未覆盖的代码行，左边的数字表示代码被执行的次数，如图 5-27 所示。

5.2.3　代码覆盖率的应用实践

以上两小节讨论的是如果分别获取 Android 和 iOS 的代码覆盖率数据。接下来我们讨论如何在项目中高效应用代码覆盖率工具。

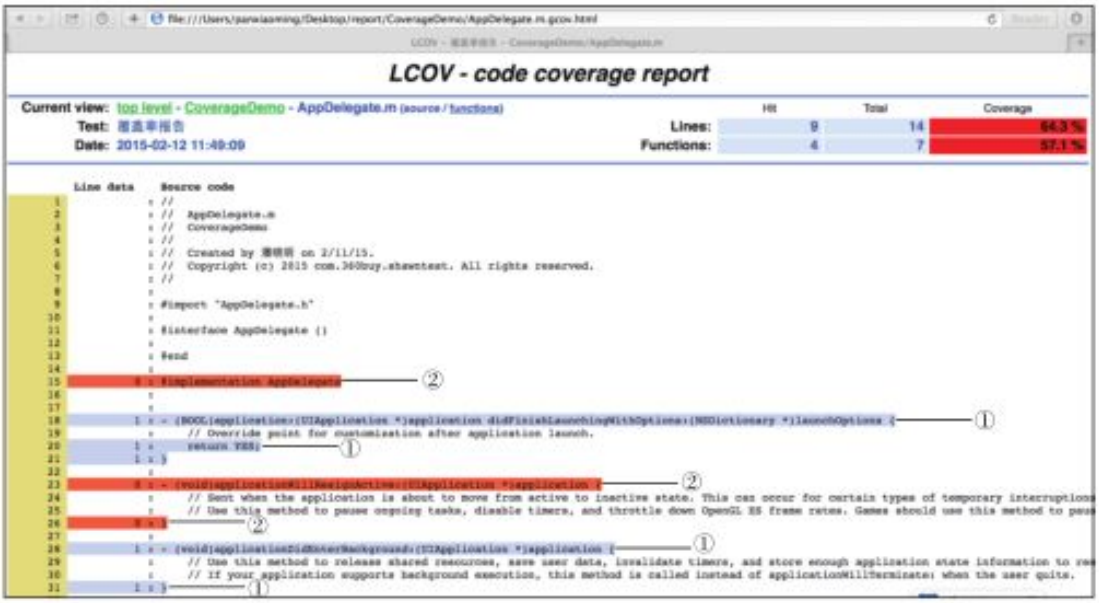


图 5-27 文件覆盖率具体内容

代码覆盖率工具由来已久，相信很多测试团队也实践过，但是实际中，这个工具很容易流于形式，因为如果强制一个代码覆盖率的指标，比如 90%，带来的结果很可能是大家为了完成覆盖率的指标而凑相关的用例，却不一定能带来实际的测试质量的提升。针对这样的情况，我们在项目中做了一些应用方式的调整，主要体现在以下几个方面：

1. 如果想要推广代码覆盖率，就必需让生成覆盖率的过程足够简单。

在实际的项目，特别是移动互联网的项目中，整个业务的节奏非常紧，如果为了获取代码覆盖率数据，需要业务测试人员额外做很多复杂的步骤，那么这个实践就很难推广，很容易被繁重的业务测试工作妥协。因为生成覆盖率数据只是完成了一部分工作，后面还需要花时间去阅读报告和分析未覆盖部分的原因。

针对这种情况，我们将 Android 的 iOS 代码覆盖率数据生成的过程高度自动化。主要的步骤是：测试人员可以通过一个统一的工具或者平台获取经过覆盖率插桩的 App 安装包；测试人员可以像普通测试包一样正常执行用例，无论 App 正常退出或者异常 Crash，通过上面的改造，覆盖率数据都可以及时写入本地文件而不会丢失；执行完测试用例后可以简便地导出覆盖率数据并生成 HTML 形式的报告，便于分析。通过这样的步骤，将生成覆盖率数据的门槛大大降低。

2. 按小功能来分析覆盖率数据

如果是整个 App 的代码覆盖率报告，因为涉及的代码量非常大，阅读和分析的代价也非常大，通常很难承受。所以实际中，我们按照测试人员和模块划分，让每个参与者每次

只聚焦一个小功能的代码覆盖率结果，通常只有几个到十来个文件，这样分析的功能量比较小，可以比较聚焦，也更便于给出有价值的分析结果。

3. 可以用于粗略的冗余文件分析

针对整个 App 维度，可以定期做一次冗余文件的分析，看哪些文件根本无法执行到，并和开发确认冗余文件。这也是覆盖率可以提供的一个有价值的参考维度。

4. 不给出强制的覆盖率指标，而是侧重对测试有什么帮助

测试覆盖率因为有具体的数值，所以比较容易给出一个强制的指标，但是这样也很容易让这个工具的导向改变。测试覆盖率本质上是一个辅助工具，帮助提供一个功能和代码之间的映射关系的参考，如果对于业务测试人员，可以提供一些有价值的参考，反到比较容易推行。实际中我们发现具体的帮助主要有两个方面。

一是帮助发现无用的代码。以下是我们在实践项目中遇到的一些情况：

1）部分为测试代码，在合入主版本之后会直接调用正式的模块，所以这些代码无法覆盖 。

2）开发在修复 Bug 时使用了另外一种解决方案，但是并未将之前的代码删除，导致这部分代码无法覆盖。

3）开发编写代码过程中遇到相似了页面，直接拷贝了已存在页面的代码，但是有些功能在新的页面里根本会走不到，所以出现了覆盖不到的情况。

针对以上几种情况，可能需要开发对代码进行处理。

二是帮助找到测试用例未覆盖的地方。以下也是实际项目中遇到的情况。通过分析覆盖率中未覆盖的情况，发现有几个异常处理的场景开发做了处理，但是测试未覆盖，需要新增用例：

1）xxx 接口返回 jason 格式文件不合法。

2）xxx 接口返回 code!=0。

3）xxx 接口返回分享的 url 非字符串。

4）xxx 接口返回 functionId 非字符串。

5）分享过程中出现网络异常时客户端的处理。

结合实际项目应用的情况来看，代码覆盖率如果合理应用会提供非常有价值的参考。

5.3　接口 Mock 方法

前面关于接口自动化的章节也提到，我们很多的 App 都非常依赖接口返回的数据来完成响应的功能。而实际运行的过程中，因为各种各样的原因，比如网络异常，接口服务异

常（可能因为后面的关联接口或者数据等方面出问题）等情况下，接口可能没有响应、返回错误信息或者有返回但内容不正确或者不完整。在这种情况下，我们可能不能期望功能还完全正常，但是应该保证下面几个方面：

- App 不会 Crash 或者点击无响应。
- 不能带来严重的影响，比如数据不完整等。
- 有比较友好的提示。

接口异常的情况不会频繁发生，实际测试中不可能等待相关的情况发生，为了高效地测试各种异常的场景，需要快速模拟和测试各种需要覆盖的情况。接下来我们介绍在实际项目中用到的几种方法。

另外，需要补充说明的是，这里介绍的方法不只是可以用在接口异常的模拟。在实际的开发过程中还有一个很有价值的用途，那就是对于前端的 App 和 M 站点开发人员，可以在后台接口还没有开发完成的时候，通过 Mock 工具提供相关的接口返回，并且可控，这样就可以在接口开发的过程中并行进行前台的开发和调试，提升研发效率，也为后面正式的前后台联调打下基础。

5.3.1 常见的接口异常模拟方法

为了测试上面提到的场景，需要模拟一些实际中不存在的接口返回，常用的技术从实现原理上可以分为三类。

1. 直接修改后台 Server，返回想要的值

这个方法听起来比较直接，就是直接修改后台 Server 的代码、数据或者配置文件，让它返回想要的数值，或者模拟出错不可用的情况。这个方法理论上肯定可行，但是实用性会有问题，主要是：

- 这样做的代价比较大，需要修改后端服务，操作起来比较复杂，而前端人员通常对后端接口的实现细节不是很了解，需要后台开发和运维人员配合。
- 可能会影响团队其他人的工作。因为在很多项目中，通常会多人共用一台开发或者测试环境，特别是当环境部署和维护比较复杂的时候。在这种情况下修改后台服务模拟异常情况会影响其他人的工作。

基于以上考虑这个方法在实际中比较少采用，当然，对于特定情况，也不失为一个可用的方法。

2. 通过一个测试 Server 返回想要的数据

第二种方法是搭建一个纯测试用的 Server，在接口协议上保持和真的 Server 一致，但

是没有任何后端负责的系统和逻辑，尽可能做得轻量和简单。

图 5-28 展示的是一个示例。

```
require 'socket'
require 'cgi'

server = TCPServer.new('127.0.0.1', 8081)
puts 'Listening on 127.0.0.1:8081'

loop {
 client = server.accept
 first_request_header = client.gets
 client.puts "HTTP/1.1 200 OK\nContent-Type: text/html\r\n\r\n<h1>Hello from ruby web server!</
     h1>"
 client.close
}
```

图 5-28　一个极简的 Web Server 代码

以上是一个用 Ruby 语言编写的 Web Server，只需要几行代码，当执行起来后，会在本机的 8081 端口监听。当前端通过浏览器或者直接 HTTP 协议访问这个端口的时候，就会获得一个 HTTP 响应，如图 5-29 所示。

图 5-29　极简 Web Server 的访问结果

相比而已，一个真实的站点可能使用 Apache 或 Tomcat 等服务期软件，以及可能有对应的负载均衡、缓存服务、其他依赖接口和数据库等。而这里只需要一个 Ruby 的运行时环境和简单的几行代码，我们就可以提供 HTTP 服务。

当然，以上只是一个示例，实际中为了模拟各种接口的响应，我们需要在这个基础上补充一些代码，来跟进不同的接口和参数提供不同的既定响应。也可以将这些请求和响应的数据按照一定的逻辑，提前放置在配置文件中，让它更加灵活。总的来说，这个代价是非常小的，而且可以随心所欲地提供各种想要的响应结果。

3. 在数据返回的途中修改

这个方法和上面提到的有些不同，既不是在原有真实 Server 的基础上修改，也不是凭空构造出一个测试用的 Server，而是在现有 Server 返回数据的基础上，按照测试需要修改对应的响应。

参考前面介绍的流量测试和弱网络测试的方法，在代理的基础上做相应的修改看起来是一个比较可行的做法。接下来我们会详细介绍实际中用到的基于代理工具的做法。

5.3.2 使用 Fiddler 作为 Mock Server

我们可以直接使用 Windows 下的 Fiddler 工具作为一个 Mock 工具。步骤为：

1）将手机与 Windows 机器连入同一局域网，并设置手机的 WiFi 代理 (代理的 IP 为 Windows 机器的 IP，端口为 Fiddler 的监听端口，默认为 8888)。

2）Fiddler 需要设置为允许远程连接，如图 5-30 所示。

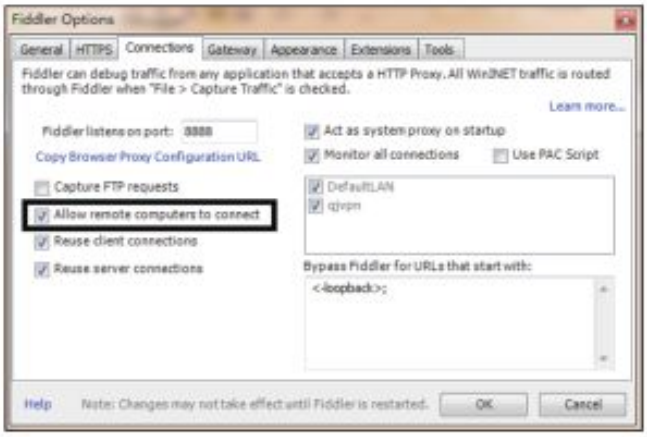


图 5-30 配置 Fiddler 允许远程连接

3）通过 Fiddler 的 AutoResponder 设置哪些请求需要返回哪些响应，如图 5-31 所示。

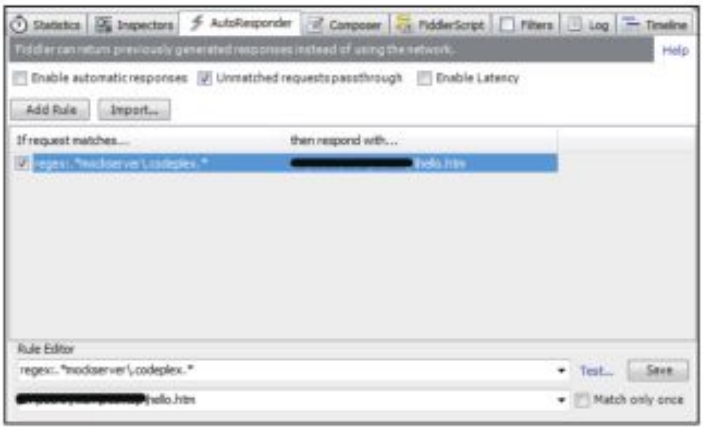


图 5-31 配置 Fiddler 自动响应

这里我们用正则表达式匹配了 url，如果 url 中含有“ mockserver.codeplex ”就返回 hello.htm 中的响应。

4）勾选“Enable automatic responses”即生效，如图 5-32 所示。

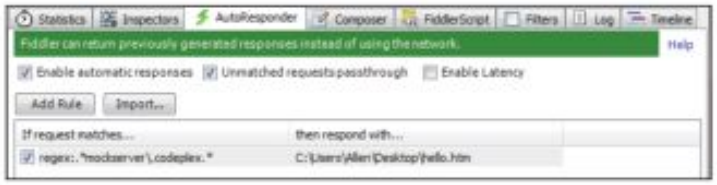


图 5-32　启用 Fiddler 允许自动响应

5）此时在手机端通过 WiFi 发送的 HTTP 请求中只要符合上面设置的匹配条件就会返回我们期望的响应数据。

我们可以使用 AutoResponder 测试以下场景：

- 非预期格式（例如接口正常返回为 JSON 格式，实际返回 xml 格式，字符串，等等）。
- 非 HTTP 200（通常情况下正常接口返回为 HTTP 200）。
- 字段缺失。
- 字段为 NULL。
- 字段格式不正确（例如应当是数字实际返回为字符）。
- 字段边界值。
- 字符型字段的文本过长。
- 返回数组为空。

模拟出上述情况后，我们可以在客户端观察实际处理的结果，根据用户看到的情况进行评估，确定是否开 BUG 跟进。

除了测试各种响应数据外，我们还可以人为增加响应延时，以测试接口响应超时情况下客户端的表现。

Fiddler 的 AutoResponder 可以方便地提供 Mock Server 功能。但是我们在实际使用中也能发现它的一些不足之处。例如：

- 书写匹配规则较为麻烦。
- 规则增加过多后不容易分辨所增加该规则的含义以及对应的接口。
- 经加密处理的请求无法匹配 HTTP Body。
- 经加密处理的响应篡改非常麻烦。
- 不支持替换部分响应内容。

上述部分缺陷可以使用 FiddlerScript 来解决。但是这个脚本的书写对测试人员要求较高，难以普及。那么，是否有一个更为简便的工具来弥补上述的不足呢？

5.3.3 基于 FiddlerCore 二次开发的 Mock 工具

FiddlerCore 是 Fiddler 工具的内核模块 (官网 http://www.telerik.com/fiddler/fiddlercore)。这个类库是公开的，我们可以基于此来自己写一个 Mock 工具来满足自己团队的测试需求，原理如图 5-33 所示。

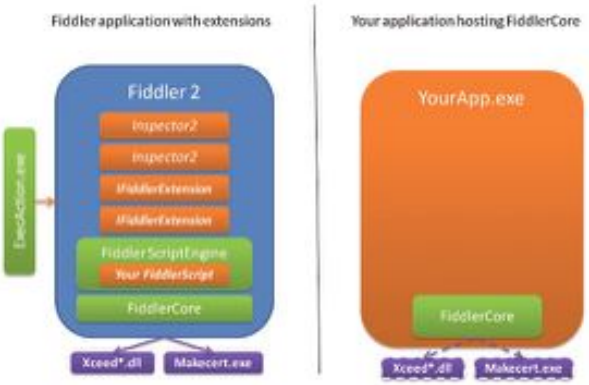


图 5-33 FiddlerCore 原理图

下面是一个使用 FiddlerCore 的简单示例 (C# 编写)。运行后所有使用 FiddlerCore 作为代理的程序发出 HTTP 请求的 URL 只要带有 bing 字符串，FiddlerCore 都会返回 hello world：

```
static void Main(string[] args)
    {
        int iProcCount = Environment.ProcessorCount;
        int iMinWorkerThreads = Math.Max(16,  6 * iProcCount);
        int iMinIOThreads = iProcCount;
        ThreadPool.SetMinThreads(iMinWorkerThreads,  iMinIOThreads);
         Fiddler.FiddlerApplication.SetAppDisplayName("TestControllerFCore");
         FiddlerCoreStartupFlags oFCSF = FiddlerCoreStartupFlags.Default;
         int iPort = 9988;
         Fiddler.FiddlerApplication.BeforeRequest += delegate(Fiddler.
             Session oS)
         {
             oS.bBufferResponse = true;
         };
         Fiddler.FiddlerApplication.BeforeResponse += delegate(Fiddler.
             Session oS)
         {
             if (oS.fullUrl.Contains("bing"))
             {
                 var newHeader = new HTTPResponseHeaders();
                 var finalRespBody = System.Text.Encoding.UTF8.GetBytes("hello
                    world!");
```

```
                oS.utilAssignResponse(newHeader, finalRespBody);
                oS.oResponse.headers["Content-Length"] = finalRespBody.Length.
                    ToString();
            }
        };
        Fiddler.FiddlerApplication.Startup(iPort, oFCSF);
        Console.WriteLine("Started");
        Console.ReadLine();
        try
        {
            Fiddler.FiddlerApplication.Shutdown();
        }
        catch (Exception) { }

    }
}
```

效果如图 5-34 所示。

图 5-34　使用 FiddlerCore 篡改响应的结果

我们在实际项目中基于 FiddlerCore 做了一定的二次开发工作，目前我们的 Mock 工具支持文件格式的规则定义，加密接口篡改等。图 5-35 所示是我们的 Mock 工具规则映射文件的示例，通过修改该文件即可驱动 HTTP 响应的篡改工作。

图 5-35　规则文件示例

这个 Mock 工具主要的功能特性如下：

1）根据用户配置的规则，拦截对于的响应，然后基于配置表中的数据文件，将响应修改成其中的内容。

2）可以支持 URL 的模糊匹配。

3）可以支持文本内容，也可以支持返回指定的图片文件。

4）可以支持网络延时。

实际使用中可以采用两种方式：

- Mock 工具本身是一个独立可以运行的 Windows 程序，可以直接监听某一个端口，作为代理使用，同时基于上述规则修改响应内容。
- 可以作为二级代理的方式使用，前面还是启动一个 Fiddler，再将 Fiddler 自身的代理指向 Mock 工具。从 Mock 功能上是一样的，但是这样的好处是在 Mock 的过程中可以继续从 Fiddler 上面看到请求和响应的内容。

我们编写的 Mock Server 的代码和相关文档可以从 http://mockserver.codeplex.com/ 下载。有兴趣的读者也可以基于我们的项目进行进一步的开发以支持自己的测试工作。

5.4 AOP 测试方法

AOP（Aspect Oriented Programming，面向方面编程）与 OOP(Object Oriented Programming，面向对象编程）对应，都是一种编程思想。从 OOP 角度来看，我们关注的是业务处理逻辑，是属于纵向的行为，从 AOP 角度分析，我们关注的是对象行为发生时的问题，是属于横向的行为。

下面我们通过一个示例来介绍什么是 AOP。

对于有些没有接触过 AOP 的人，乍一听完全让人摸不着头脑。这和我们通常所知的面向对象编程 OOP 有何区别？让我们来看一个简单的例子帮助理解。

假设已有一个 OOP 编写的系统，其中有下列类，如图 5-36 所示。

图 5-36 假定系统中包含的类

每个类代表不同的人做不同的事。然后有一个新需求需要实现：所有人做任何事情前都要上报董事长。那么 OOP 下应该如何实现呢。

实现方式 1:

即在每个类中的每个方法开头部分都调用上报董事长方法：

```
Void 测试()
{
    上报董事长();
…
}
Void 吃饭()
{
    上报董事长();
…
}
```

实现方式 2:

预见到将来可能的需求更改，将上报部分抽离到报告模块中。可预见的更改可能跟不同人有关，所以传入一个实例给报告模块以供可能的判断逻辑使用：

```
Void 测试()
{
    报告模块.报告(this);
…
}
Void 吃饭()
{
    报告模块.报告(this);
…
}
…
Void 报告模块.报告(Person  p)
{
    上报董事长();
}
```

显而易见，实现方式 2 的扩展性更好。但是这里的问题是，如何可以正确预见到将来的更改？只传调用者的实例是否足够?

假设又有一个需求更改来了：只有财务在做报销操作时候才需要上报董事长。那么用实现方式 2 又很难通过简单的代码变更来实现。

上述这些类型的需求我们称之为横切性需求。

这时候我们再回头看一下实现方式 1。我们可以发现这个方式的主要缺点就是在此类需求变更的时候，需要一个个删除 / 添加方法的调用。那么如果我们能够有一个自动化的框架，只要给出我们横切性需求的定义，让这个框架自动找到相应的代码，删除 / 添加方法调用，那实现方式 1 其实是完全可以接受的，而且能完全解耦，更为方便。

基于 OOP，通过横切性需求定义来进行编程的方法即为面向方面的编程 AOP。

AOP 本身是一个开发技术，诞生也有很长的一段时间了，但是从测试的角度，因为它

的上述特性，我们也可以把它作为一项测试技术，横向的来看测试过程中我们感兴趣的方面，从而希望能发现一些问题，达到提升产品质量的目的。以下 AOP 测试技术实践中关于 Android 的部分参考了之前腾讯 QQ 音乐测试团队的部分实践经验，就两个团队的测试结果来看，这个方法能提供一些非常有价值的发现。而且针对这些问题，发现的方式也比较高效。

下面分别介绍我们针对 Android 和 iOS 的 AOP 测试实践。

5.4.1 Android AOP 测试实践

AOP 技术是和编程语言强相关的，对于 Android 系统，除了一些手机游戏，大部分是采用 Java 语言来编写 App 端代码的，所以这里使用的框架也是基于 Java 的。下面我们结合一个具体的例子来看整个使用过程。

1. 什么是 AspectJ

AspectJ 是 Java 下实现 AOP 的一个框架。AspectJ 在 2002 年被转让给 Eclipse Foundation，从而成为开源社区中 AOP 技术的先锋，也是目前最为流行的 AOP 工具。以下是其中几种主要的概念：

- 切面 (Aspect): 对横切关注点的抽象。
- 连接点 (JoinPoint): 被拦截到的点，泛指方法。
- 切入点 (CutPoint): 对哪些连接点进行拦截的定义。
- 目标对象（Target Object）: 包含连接点的对象，也称为被通知或代理对象。

基本原理是解析横切性需求定义文件，找到需要修改的源文件，然后使用自己的 ajc 编译器进行编译。编译过程中，复制源代码到新的临时文件，修改新的临时文件，并编译修改的新源代码。对上一章节提出的需求：只有财务在做报销操作时候才需要上报董事长，我们可以用下列的 AspectJ 横切性需求定义来描述：

```
pointcut 财务报销 ():execution(* 财务类.报销(..));
    before():财务报销 ()
{
上报董事长()
}
```

AspectJ 也支持更为友好的 Annotation 语法定义横切性需求，并且支持编译后以及运行时的代码插入。更多的使用方式请参考 AspectJ 官方网站 (http://www.eclipse.org/aspectj/)。

2.AspectJ 在 Android 测试中的应用

我们可以在测试中使用 AOP 方法，只用很少的代码去实现一些横切性的测试需求。例如，我们可以用 AspectJ 在所有捕获和未捕获的异常抛出点进行日志记录工作，记录调用堆

栈等。通过分析日志，我们可以发现 App 崩溃的原因，定位到代码行，并且能够发现一些不正确的异常捕获逻辑。（未捕获异常会导致进程崩溃，捕获异常处理不当可能意味着性能和代码逻辑上的隐患。）

下面以 gradle 编译环境为例，说明如何使用 AOP 进行 Android App 的全局异常记录。

1）在项目的源文件目录下添加 com/example/aspectJ/Exceptions.aj 文件。内容如下：

```
package com.example.aspectJ;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
import android.content.Context;
import android.util.Log;
public aspect Exceptions {
    private static final Object fLock = new Object();
String logFilePath="/sdcard/myapp_Ex_Log";
// unhandled exceptions
    pointcut UHEPoint():execution(* *(..))&&!within(com.example.aspectJ.*);
// handled exceptions
    pointcut HEPoint(Throwable ex, Object exHandlerObject):handler(*)&&args(e
        x)&&this(exHandlerObject)&&!within(com.example.aspectJ.*);
    after()throwing(Throwable ex):UHEPoint()
    {
        String str="";
        str+= "==UH==="+"\n";
        str+=ex.getMessage()+"\n";
        str+=ex.getClass()+"\n";
        for (StackTraceElement ste : ex.getStackTrace())
        {
            str+= ste.toString()+"\n";
        }
            AppendLineToLogFile(str);
    }
    before(Throwable ex, Object exHandlerObject):HEPoint(ex, exHandlerObject)
    {
        String str="";
        str+= "==H==="+"\n";
        str+=ex.getMessage()+"\n";
        str+=ex.getClass()+"\n";
        for (StackTraceElement ste : ex.getStackTrace())
        {
            str+= ste.toString()+"\n";
        }
        AppendLineToLogFile(str);
```

```
    }
    void AppendLineToLogFile(String txt)
    {
        Writer output;
        synchronized (fLock)
        {
            try {
                File logFile = new File(logFilePath);
            if(!logFile.exists()) {
                logFile.createNewFile();
            }
                output = new BufferedWriter(new FileWriter  (logFilePath, true));
                output.append(txt);
                output.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

上述代码对程序中未捕获和捕获的异常都进行了抓取并写入日志。其中主要分两部分。第一部分是切面点的声明。以未捕获异常为例，代码如下：

```
pointcut UHEPoint():execution(* *(..))&&!within(com.example.aspectJ.*);
```

这是 AspectJ 的切面定义语法。UHEPoint() 是我们自己起的方法名字，说明符合我们设定的切面条件后调用哪个方法。execution(* *(..)) 的意思是代码中所有方法都符合我们的切面定义，即在所有方法调用的地方都要调用我们自己的定义的 UHEPoint() 方法。!within(com.example.aspectJ.*) 是指除了 com.example.aspectJ.* 包下的类外，所有方法都符合切面的定义。这主要是为了防止我们自己的 UHEPoint() 抛出异常后导致死循环的产生。

第二部分为方法的实现。即：

```
after()throwing(Throwable ex):UHEPoint()
    {
...
            }
```

这里声明了 after()throwing(Throwable ex)，表示切入点是方法执行中抛出异常的时候。UHEPoint() 是我们自己起的方法名，这样 AspectJ 引擎就能关联起切面定义和我们的切入点的实现。在 UHEPoint() 中，我们在切入点中在日志里记录了未捕获的异常信息。

2）在希望进行全局异常记录的 module 项目的 build.gradle 文件中添加如下内容：

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.uphyca.gradle:gradle-android-aspectj-plugin:0.9.+'

    }
}
apply plugin: 'android-aspectj'
```

3）（可选）在任意方法中添加一段测试代码抛出异常或者抛出异常并捕获。例如：

```
throw new OutOfMemoryError("test");}
```

4）在 module 目录下运行 gradle assembleDebug 编译并生成 apk 包。

5）安装 apk 包到手机并进行操作触发抛出异常的代码。

6）观察 /sdcard/myapp_Ex_Log，所有抛出异常的调用堆栈都会被记录在这个文件中，如图 5-37 所示。

```
root@klte:/sdcard # cat myapp_Ex_Log
cat myapp_Ex_Log
--UH--
test
class java.lang.OutOfMemoryError
com.example.test.myapplication.SettingsActivity.onPostCreate(SettingsActivity.ja
va:48)
android.app.Instrumentation.callActivityOnPostCreate(Instrumentation.java:1156)
android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2370)
android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2453)
android.app.ActivityThread.access$900(ActivityThread.java:173)
android.app.ActivityThread$H.handleMessage(ActivityThread.java:1303)
android.os.Handler.dispatchMessage(Handler.java:102)
android.os.Looper.loop(Looper.java:136)
android.app.ActivityThread.main(ActivityThread.java:5579)
java.lang.reflect.Method.invokeNative(Native Method)
java.lang.reflect.Method.invoke(Method.java:515)
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:1268)

com.android.internal.os.ZygoteInit.main(ZygoteInit.java:1084)
dalvik.system.NativeStart.main(Native Method)
```

图 5-37　使用 AOP 方法获取到的异常信息

通过上述的方法，我们在实践中确实发现了不少开发不正确处理的异常逻辑。同时，我们用这一方法也能获得崩溃日志。这对于一些测试中偶现的问题的排查工作来说是十分有帮助的。

5.4.2　iOS AOP 测试实践

类似前面讨论的 Android 的 AOP 方法，iOS 也有类似的技术。而且 iOS 有不少 AOP 的实现代码，大家可以到 Github 上查找。这里简单介绍一个 iOS 的 AOP 实现原理：通过

Cocoa 本身支持在 App 运行时，将拦截器注入给代理对象，使其干涉真实对象的执行顺序从而达到给代码增加“切面”的目的。网上很多的 AOP 实现类库，底层实际都使用到了这个技术。

这里我们使用一个叫 Aspects 的类库，(https://github.com/steipete/Aspects)，它支持在拦截方法前执行你的代码，拦截方法后执行你的代码，或者彻底用你的代码替换拦截方法。不过它不支持拦截静态方法，比较遗憾。

接下来我们使用 Aspects 的来实现 iOS 的 AOP，作为例子，我们这次来拦截 App 底层的 http 请求，获取它的请求和返回，然后显示到 App 浮窗上。浮窗的逻辑不是我们本节的重点，本节主要说明如何使用 AOP 以及使用 AOP 后的效果。

1）添加文件到项目工程。在 Xcode 中，点击 File 菜单，选择 Add files to 项目名，点击先把 Aspects.h，Aspects.m(从 github 下载 Aspects 即可找到) 这两个文件添加到项目工程，如图 5-38 所示。

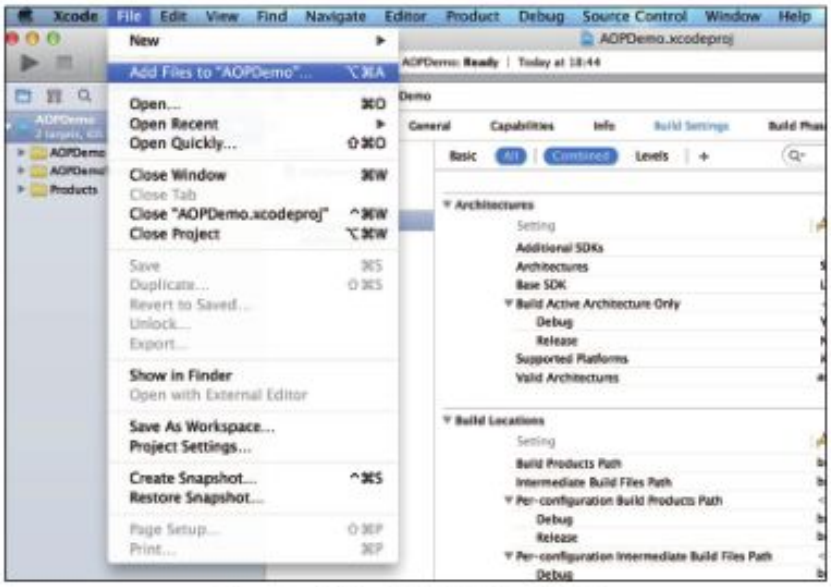


图 5-38　添加文件到项目工程选项

在弹出的文件窗口中选择 Aspects.h，Aspects.m 这两个文件，点击添加将这两个文件添加到项目工程中去，如图 5-39 示。

2）在项目 App Delegate 中，导入 Aspects.h 文件，如图 5-40 所示。

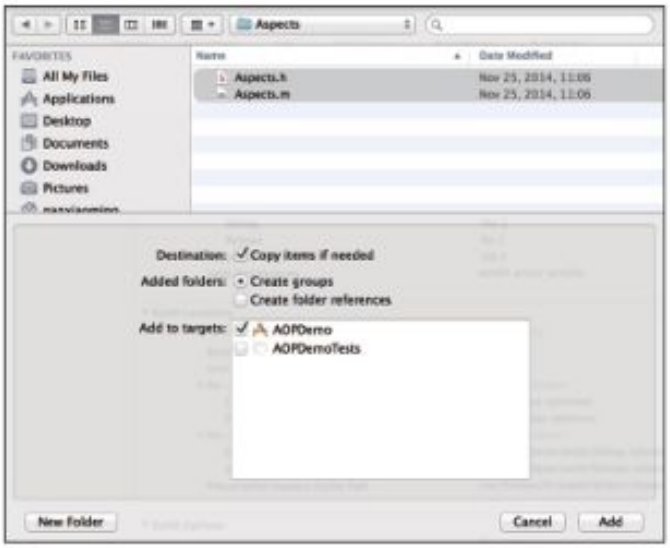


图 5-39　选择要添加的文件

图 5-40　导入头文件

3）在 App Delegate 类中，找到 -(BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions 方法，在里面添加 AOP 切面，如图 5-41 所示。

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
    (NSDictionary *)launchOptions {
    MyWindow *mw=[[MyWindow alloc] init];
    [HttpModel aspect_hookSelector:@selector(setResponseInfo:) withOptions:
        AspectPositionBefore usingBlock:^(id<AspectInfo> aspectInfo,NSDictionary*
        info){
        [mw getResponseInfo:info];
    } error:NULL ];
    // Override point for customization after application launch.
    return YES;
}
```

图 5-41　AOP 实现代码

代码中我们有一个浮窗类，叫 MyWindow，用来展示网络请求数据，我们在这里先把它初始化。接下来是重点，这里我们 hook 的对象叫做 HttpModel，这是一个用来管理 http 请求的类，它用来管理 http 的请求数据和头，以及 http 返回的数据和头，我们不用了解这个类的细节，我们只需知道，这个类有一个方法叫做 setResponseInfo: 它有个 NSDictionary

的参数，这个参数包含请求完成后的响应信息。我们在这个方法执行前，先截取它的传参，把这个参数传入到我们自己的 MyWindow 的 getResponseInfo: 方法里。通过解析这个字典来完成展示 http 的信息。

了解了切入目的后，我们就可以用 Aspects 的方法去做拦截操作了。这里我们用到了 Aspects 的 +(id<AspectToken>)aspect_hookSelector:(SEL)selector withOptions:(AspectOptions)options usingBlock:(id)block error:(NSError **)error 方法。看到前面的 +，知道这是一个静态方法，可以不用初始化 Aspects 直接使用该方法。这个方法返回一个满足 AspectToken 协议的对象，我们不用关心这个返回。接着看到第一个参数是 SEL，顾名思义就是我们要拦截的方法，因此结合前面的分析，我们在这里传入的是 @selector(setResponseInfo:) 紧接着便是 AspectOptions 参数，这是一个结构体，Aspects 支持 3 种拦截选项，分别是：

- AspectPositionAfter ＝0。默认选项，在拦截方法前执行自己的代码
- AspectPositionInstead = 1。用自己的代码完全替换拦截方法
- AspectPositionBefore ＝2。在拦截方法执行之后，执行自己的代码。

这里我们使用的是在拦截方法之前先执行我们的代码。

后面就是一个 block 的参数，这里便是我们自定义代码的存放位置。在这里需要注意的就是，由于 block 的作用域限制，非 block 本身传入的参数是无法识别的。在这里我们的 block 是这么定义的：

```
^(id<AspectInfo> aspectInfo, NSDictionary* info){
        [mw getResponseInfo:info];
}
```

block 传入的两个参数重点介绍一下，参数之一是遵循 aspectInfo 的对象，那么这个对象具体是哪个类型呢。这不就是之前提到的 HttpModel 这个类吗？因为是它被拦截啊。其实不对，我们之前简单提过 iOS 的运行时机制和代理对象，其实这就是一个代理对象，我们靠它来注入拦截。这里我们不用太过关心这个参数。参数之二是我们拦截的方法里的传入参数，我们需要捕获这个参数，然后在 block 实现中，把这个参数传入方法中。

最后一个参数是 error: 用来指定出错后的操作，这里我们设为空。

让我们来重新编译整个项目工程，查看 AOP 之后的效果，如图 5-42 所示。

可以看到，我们成功地拦截到了 HttpModel 这个类的方法，截取了 HTTP 的返回，并且把这些信息展示了出来。

图 5-42　通过 AOP 捕获 http 请求返回的信息

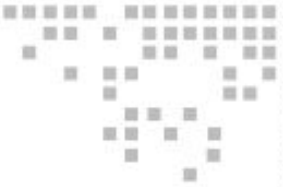
截至目前，我们分别介绍了 Android 和 iOS 平台的 AOP 测试实践。需要补充说明的是，AOP 不只是可以用于以上示例的关于 Exception 的横向捕获，还可以用于很多其他的方面，比如通过横向跟踪 activity 的 onStop 和 onResume 来获取整个使用路径，可以用于问题的复现和定位。进一步来看，AOP 其实是一个非常通用的技术，怎么使用要取决于项目的测试需求。

5.5　本章小结

本章我们重点介绍了几种在软件测试中常用的测试辅助方法。首先介绍的是代码静态扫描的方法，包括了分别针对 Android 和 iOS 的工具，以及具体的案例。对于所有比较通用且使用群体较大的编程语言，几乎都有对应的静态扫描工具，这些工具可以在不运行被测代码的情况下，根据之前积累的经验规则，快速地扫描出代码的潜在问题。根据我们之前的经验，虽然可能有一些误判，但是总体而言还是非常有价值的。而且因为静态扫描对运行环境的依赖比较少，所以也非常适合加入到持续集成（CI）里面，在开发人员提交代码后快速地发现问题。代码覆盖率分析是另一个非常通用的测试分析手段，针对 App 产品依然非常的有价值，可以让我们清晰地知道哪些代码被执行到了，是测试覆盖的一个重要参考，本章除了介绍工具的使用，也介绍了我们在这方面的一些实践。在实际项目中，为了提高开发的效率和模拟各种接口的返回，Mock 也是一个很常用的技术手段，本章也做了介绍，探讨了几种常用的方法，并深入讨论了基于 FiddlerCode 做二次开发的实现思路。最后

我们介绍了基于 AOP 的测试方法。因为 Spring 框架的流行，AOP 作为一个编程方法被广为使用，其实它也是一个很好的测试方法，可以帮助我们横向地快速拿到很多信息，用于测试分析。

总体而已，为了让我们的测试和质量保证做得更加深入，这些被实践证明过的辅助方法都有很大的价值，在实际的项目中，可以结合具体的情况来逐步开展，相信一定会对产品质量的提升有积极的帮助。



第 6 章 Chapter 6

发布过程中的质量管理

任何科学的雏形都有双重的形象：胚胎时的丑恶，萌芽时的美丽。

——雨果

互联网软件产品和传统软件产品有一个很大的差别在于：谁来运营这个系统？或者用一个比喻来说，传统的很多软件模式是买桶装水，虽然也会有一些服务支持，但是日常的运作和提供方没有直接的关系。而互联网软件就好比自来水，除了提供水，还需要铺设和维护管道，用户有自己的账号，还需要定期统计使用情况。互联网软件在这种情况下更像是一个服务，既然是服务，其触达用户的方式就有所不同，有一个发布的过程。

本章试图探讨在软件的发布过程中，我们有什么方式来保证发布本身的质量，以及是否可以借助这种触达方式，帮助提升产品本身质量。本章会讨论四个方面的内容：

- 持续集成，这和编译打包通常密切相关，提供发布物。
- 把控发布环节的质量。
- 通过开展内测尽早发现问题。
- 通过灰度机制，避免大范围一次性发布的风险，以及收集反馈。

以上这些做法都是在各个互联网研发团队中广泛使用的做法，非常值得去实践并不断在细节上的改进。

6.1 持续集成

持续集成这个实践经过最近几年的普及和推广，已经比较深入人心了，稍大一点儿的

研发组织，也包括很多的创业团队都有自己的持续集成平台。关于持续集成的概念这里做个简单的介绍，如果想比较系统地了解持续集成，可以参考《持续集成：软件质量改进和风险降低之道》。

6.1.1 持续集成简介

持续集成是一种软件开发实践，将新的开发成果快速地和已有的基础集合起来，快速地提供可用的版本，并进行快速的测试和验证。其中一个很重要的理念就是 Fail Fast，如果会失败就尽快暴露出来，如果有问题的话。通常的实践是每次集成都通过自动化的构建（包括编译、发布、自动化测试）来验证，从而尽快地发现开发中的错误。许多团队发现这个过程可以大大减少集成的问题，让团队能够更快地开发高质量的软件，也提升整体的效率。

持续集成的目标主要体现在以下几个方面：

1）及时暴露问题。开发团队在开发过程中，每天都会把一天的成果提交到代码管理系统。随着人员的扩张，时间的流逝，项目代码会越来越臃肿，谁都不能避免某一次的提交过程中引入了错误的代码。利用持续集成，我们能够在第一时间发现问题并暴露出来，使得开发团队能够非常快速地定位到问题，然后修复它。不把问题遗留到第二天，减少后期返工的次数和返工工作量，降低了项目的风险。

2）快速提供产品。开发团队在开发过程中，常常需要用到最新的版本，比如测试人员需要回归验证缺陷是否得到修复，产品经理需要体验新功能，这样就需要能否快速完整地构建开发中的产品。通过持续集成，产品经理和测试等角色能够随时随地查看最新的产品式样，及时地纠正产品在研发过程中出现的偏差和需求理解错误，提升产品的质量。也通过问题的快速修复来提高效率。

3）减少重复劳动。项目工程每天的编译、打包和一些基本的验证工作是枯燥且冗长的，通过持续集成，我们可以将上述过程定时或者通过代码提交等事件触发而自动执行，省去了大量重复的工作。

4）提高项目可见性，增强团队信心。持续集成通过快速的构建可用的系统，也可以在过程中通过各种检查手段快速地发现问题。通过持续这样的过程，可以让整个团队清楚哪些功能已经开发完成，是否可以编译打包，是否可以通过静态检查以及单元测试、功能测试等检查手段？通过快速、自动化地给出这样的结果，可以让研发团队和管理层及时知道项目的实际进展情况，从而可以提供辅助决策。而通过不断地完成可用的新功能点，也让整个研发团队看到清晰的进展和取得的成果，也会增强整个团队的信心。

下面我们结合一个项目中的实例来看看持续集成带来的效果。图 6-1 所示是某个系统

的编译失败情况，按月进行统计。

图 6-1　编译失败数量变化曲线

在 11 月份的时候开始将编译环节接入持续集成，并且在有编译失败的情况下自动发送邮件给整个开发和测试团队，附带编译失败的原因和错误日志。通过这样的实践，将编译失败的情况第一时间透明出来，并逐个定位到原因和对应的开发人员，进而讨论如何优化。到了 12 月份，编译失败的情况有了大幅度的降低。

持续集成这个实践本身并不能帮助我们定位引起编译失败的代码，以及优化。但是通过这样有代码提交后的快速构建，以及失败案例的周知机制，非常明显地推动了编译失败问题的改善。编译失败的大幅下降一方面节省了大家的时间，另一方面也提高了版本质量。有了这样的机制，也会隐性地提高整个团队的质量意识，因为谁也不想自己提交的代码导致项目编译失败。

6.1.2　持续集成实践

在讨论了持续集成的基本概念和益处之后，我们来看看常见的实施方案。业界有非常多的持续集成平台，这里以最知名和使用率最高的 Jenkins 工具为例来介绍。

Jenkins 支持多个操作系统，简单易用，提供了一套完整的从 svn 获取代码到编译打包分发的流程，同时，它支持很多扩展插件，适合不同的项目需求。详情可以参考官网：http://jenkins-ci.org。

以下我们通过一个实际的例子来看看基本的持续集成的构建过程，主要包括代码覆盖率插桩和打包。

1. 持续集成的设计框架

用户身份包括：

- 管理员：持续集成管理员创建和维护持续集成项目。
- 普通用户：测试人员每个人都有自己的账户，能够登录 Jenkins 并且自己去构建项目工程，下载打包完的 ipa 包，但是无权限修改和删除构建项目。

构建项目：

1）建立多个版本的项目工程，以便能够测试不同版本的 App。

2）在版本的基础上，建立不同的插桩项目工程。例如，原版的项目工程，带有代码覆盖率设置的构建项目，带有 AOP 插桩的构建项目。

3）通过参数化方式，让用户自行选择是在线安装或是创建 ipa 包。

4）通过附件上传方式，上传用户的覆盖率数据，在线制作 iOS 代码覆盖率报告。按登录用户和 svn 下载版本的命名方式，保存用户的覆盖率报告。

5）制作不同设备的构建项目，如 iphone 项目，ipad 项目。

6）扫描代码，可以静态分析代码，同时扫描项目工程中的无效资源。

通知邮件：1）构建成功：发送成功邮件，通知构建人员和管理员。2）构建失败：发送失败邮件，通知构建人员和管理员。

2. 具体实现

这里不讨论每个实现的细节，只是针对某些重要步骤进行简单的介绍。

1）创建用户并且设置权限。

进入全局安全设置，选择授权策略，如图 6-2 所示。

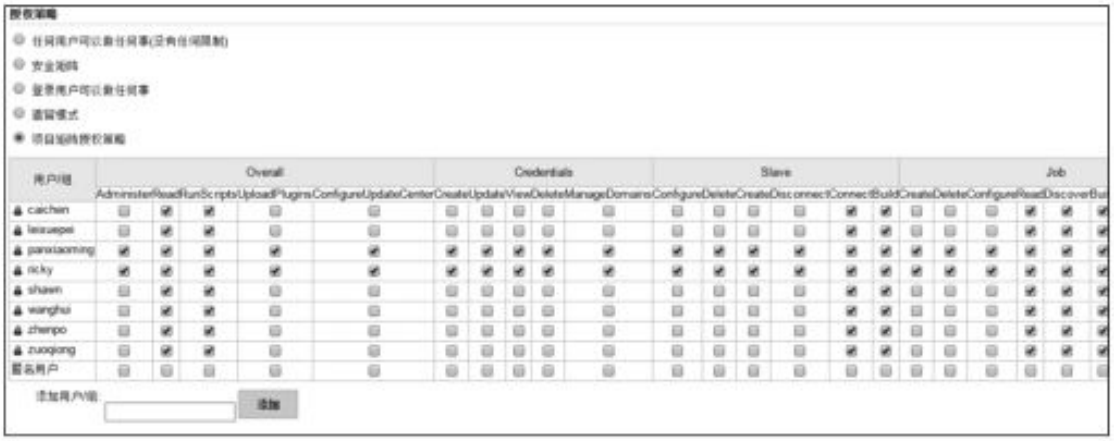


图 6-2　Jenkins 项目授权策略设置

2）创建 iOS 构建工程。设置参数，让用户选择是打包还是在线安装，如图 6-3 所示。

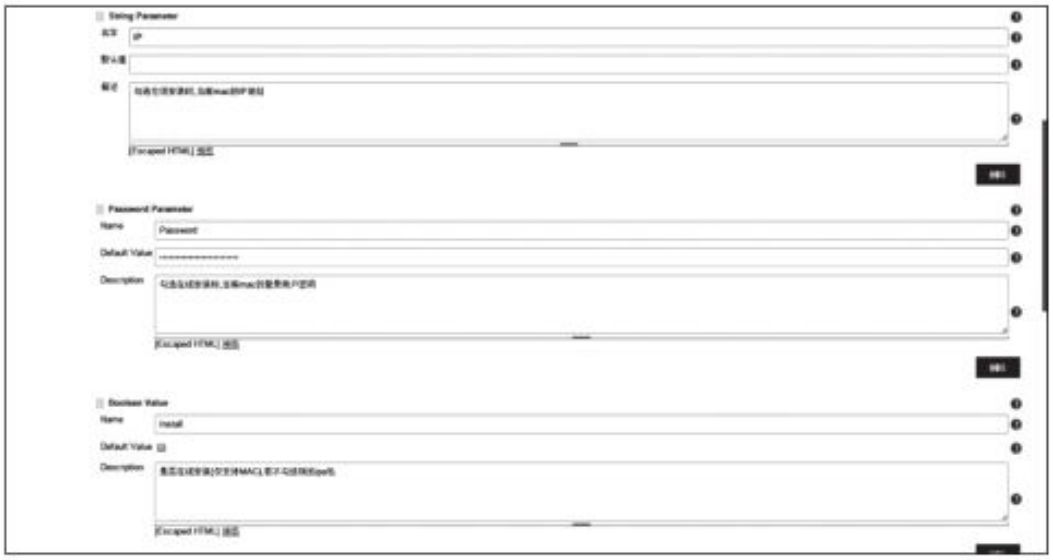


图 6-3　创建持续集成工程

参数 1: 文本输入框，输入远程用户电脑 IP。

参数 2: 密码输入框，输入远程用户的密码。

参数 3: 勾选框，提示是否在线安装。

设置 svn 路径，以及更新方式，如图 6-4 所示。

图 6-4　指定项目源码路径

设置项目编译，打包，安装过程，通过 shell 执。如图 6-5 所示。

iOS 的命令行编译以及 ipa 打包，可以参见第 5 章中 iOS 代码覆盖率技术方案小节。

发送邮件设置使用的是一个发送邮件的插件，支持设置发送内容，发送对象，触发器设置。如图 6-6 所示。

3) 用户使用。登录 Jenkins 后，选择构建项目，点击左边的 Build With Parameters，如

图 6-7 所示。

图 6-5 指定需要执行的脚本

图 6-6 构建后操作设置

图 6-7 Jenkins 中的项目主界面

如图 6-8 所示，输入完参数后，点击开始构建即可，等待构建完成。

这样我们就能从 Jenkins 中编译安装或者打包 iOS 项目了。我们可以通过远程脚本触发构建，也可以通过用户手动去构建。每个用户可以按需构建，也可以直接复用别人刚才构建的成果。测试人员就不必在本地维护项目工程代码和编译安装环境了。通过 Jenkins 统一平台，完成 App 包的更新下载。

图 6-8　项目的参数配置

以上只是一个简单的示例，在这个基础上可以挂接更多的持续集成任务，常见的如下：

- 编译打包，包含失败告警。
- 静态代码扫描日报。
- 包检查（大小、冗余文件等）。
- 代码覆盖率插桩（可选择出覆盖率的包）。
- 自动化测试及对应的报告。
- 流量测试，电量测试。
- 性能测试。

从以上的介绍可以看出，只要能将一个任务的入口做成一个可执行命令，就有可能挂接到持续集成平台上。当然，结合具体的任务，可能对测试环境和数据有一定的依赖。

6.2　发布环节的质量把控

在这一节，我们集中讨论在发布的过程中需要注意的一些问题和一些被证明过有帮助的做法。

在讨论具体的做法之前，有两个问题需要明确。首先我们需要发布的是什么？对于基于 Web 的互联网产品，需要发布的是网站的前后台，需要发布的文件都是在服务器上。对于移动互联网的产品，如果是手机页面形态，那还是一样；如果是 App 形态，那就包含了两种不同的发布形式，App 有 App 对应的发布渠道，而后台部分和之前 Web 形态的互联网产品一样。在下面的小节我们会针对这两种形态分开来讨论。

其次是关于谁来负责发布？不同的组织有不同的做法，以我目前所见到的有下面几种：

- 开发人员
- 产品经理或者运营（主要是针对 App）
- 运维人员
- 测试人员
- 专职的配置管理员

当然，也可能还有其他的角色参与其中。这个并没有对错，要依据各个研发组织自身的情况来看，目标还是侧重发布的质量和效率。

6.2.1 后台服务的发布

接下来我们讨论在后台发布过程中需要注意的一些问题。在一些创业团队或者项目在很早期的时候，往往采用的是人工发布，直接将需要发布的文件打包复制到对应的服务器上，然后手动启停服务来完成整个发布的过程。这个方式在发布不是很频繁、开发团队人数较小的时候是适用的，但是随着产品功能的不断增加，开发团队的不断扩大，以及外部用户量的快速增长，这种方式就完全不能满足需求了。在这种情况下，很多研发组织都有了自己的发布系统，可能是整个公司级统一的，对于大的公司，也可能是部门级的系统。就目前看到的情况，通常都是自研的系统，比较少见通用的标准化系统。这里我们不以具体的系统来说明，只是讨论一些为了保证发布的质量，发布系统应该提供的主要特性，以及一些实践的做法，供大家参考。

提到发布，我们需要对相关的环节有一个了解，便于后续的讨论。图 6-9 给出的是一个比较典型的互联网产品的后台发布所经过的环境，其他组织的环境可能局部有差异但大致比较相同。

图 6-9 中的源码管理系统和编译环境比较容易理解，接下来介绍下为什么需要其他几个环境。“开发环境”通常是给开发人员在正式提交代码之前做自测和调试使用。“测试环境”，顾名思义，是用来进行测试使用的，开放人员做完自测后将功能提交给测试人员，来进行比较完整的各方面测试。所以通常“测试环境”比较全面，能执行产品各个主要功能流程，并且有相关的测试数据。“预发布环境”是互联网产品特有的，通常可以这样理解，它本质上其实是线上环境，是其中的一台或者多台机器，但是这些机器从线上负载均衡配置中摘除了，使得外部用户的流量不会到这台机器上。也就是说这里只是内部人员用于新功能验证的地方，不对外部开发。通常，“预发布环境”对接的后台和数据库都是线上真实的，这有利有弊。好处是可以用线上真实的后台和数据来验证功能点，能发现一些测试环境因为配置和数据无法发现的问题，也可以看到更真实的产品效果。但是另一方面，因为是线上的系统，所以所有的操作都会影响到数据，有些操作因为影响实际系统运行不一定

可以执行。最后的“线上环境”是我们正式提供对外服务的地方，这里的任何改动都可能会直接影响到外部真实用户。

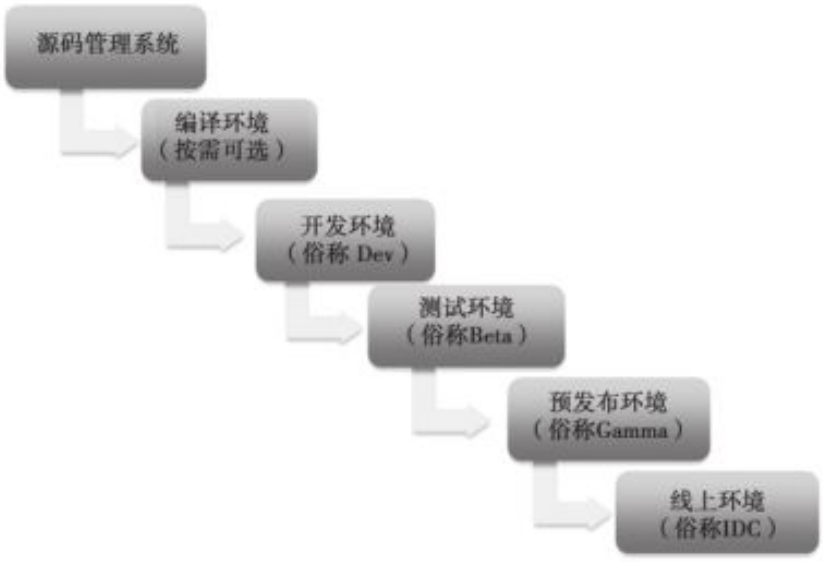


图 6-9　功能发布所涉及的环境

我们觉得一个成熟的发布系统应该具备以下一些能力。

1. 需要和前置的源码管理和编译打包系统对接

发布系统最本质的功能就是将本次发布需要更新的文件推送到各个线上的服务器上，那么最基本的问题就是怎么从源头获取到文件。根据所用的技术不同，形态也有些差异。

对于 PHP/Python 这类解释型的语言，直接将源文件发布到线上服务器即可，这种情况下发布系统可以直接和源码管理系统对接，程序自动拉取对应的文件。

对于 Java/C++ 等需要编译的语言，还需要一个编译环境，通常会编写对应的自动化编译和打包脚本，快速生成可以发布的文件。

2. 最好能和测试环境做对接

测试环境本身也是需要发布的，因为需要测试新的功能以及回归缺陷修复的情况，只是通常相比线上环境简单不少。所以一个完善的发布系统也包括对测试环境的发布。

另外还有一个需要面临的选择，那就是发布到预发布或者线上的文件是来自于测试环境还是直接来自于源码管理 / 编译环境。

如果是直接来自于源码管理 / 编译环境，就会遇到以下两个问题：

- 测试环境测过的版本和发布的版本可能不一样，因为在测试之后可能还做了改动。
- 如果有个别功能是不需要经过测试的，俗称免测，那么就可能会忘记将这部分功能

的改动发布到测试环境。带来的后果是测试环境因为遗漏了新的改动变得不可用或者不能体现真实情况了。

如果选择从测试环境拉取文件发布到后续环节，避开了上面的问题，也会有一些隐患，比如测试环境的文件可能会被不小心修改，以及代码覆盖率等工具可能会需要对文件插桩。

针对这样的问题，实际项目中一方面可以通过流程和习惯来做约束，另一方面也有一些辅助的技术手段，比如用文件的比对机制来减少这方面的影响。

3. 完备的权限管理

对于一个已经有成千上万的正式用户的线上系统来说，每一次的发布都是一件比较严肃的事情，因为总是会伴随一定的风险，相信每一位做过一段时间此类项目的同行都会有类似的体会。另一方面，大型的项目参与的人会非常多，几百人参与开发的系统也比较常见。基于这样的情况，发布系统就非常有必要进行严格的权限控制，也能避免很多的误操作带来的影响。权限管理主要考虑以下几个方面：

- 明确哪些人有发布权限，成熟的系统可以和个人在组织里面的域账号 /LDAP 账号做关联来设定。
- 对发布权限按模块做细分，建议每个人的权限只能发布自己相关的模块，避免误操作。当然也可以对少量人员留有更高权限去发布多个模块。
- 对发布系统本身的管理权限，建议收归到少数管理员手中。比如发布系统本身的配置，包括人员管理、模块管理、机器管理，以及发布机的操作系统登录权限。

4. 需要提供校验、重传等容错功能

发布系统本身也是一个文件传输和管理系统，而在传输的过程中会遇到各种各样的问题，比如网络问题、磁盘问题和权限问题等，所以也需要有一定的校验和容错机制。最基本的，需要对发布之后的目标文件和源文件做二进制比对，以确保复制和传输过程中没有问题。另外，对比传输失败的情况，也需要有重试的机制，重试不成功后必须有明确的提示，不能让发布人员误以为成功。

5. 需要有分布式发布的能力

在一个实际的较大型的项目中，线上的服务器通常有很多台，用来分担负载，所以对于发布系统来说，需要能够同时发布到多台机器。当机器数量较大时，需要有并行发布的能力，以提高发布的效率，缩短处在中间状态的时间。另外，还需要有选择性发布的能力，使得灰度发布成为可能。关于灰度发布的细节做法，稍后章节会来讨论。

6. 需要有自动回滚的能力

发布过程中难免会遇到一些问题，比较严重的是发布后发现了严重的缺陷，导致系统

不可用，可能是因为测试的遗漏，也可能是因为线上环境不同导致的。但是在影响了线上服务的紧急情况下，时间上通常不允许立即进行问题的定位和修复这类比较耗时且时间不确定的操作，最好的做法是立即回退，或者称为回滚，到发布之前的文件版本，快速地恢复服务。这个基本上是发布系统应该必备的功能。要做到准确和快速地回滚，有非常多的技术细节需要考虑，对发布系统本身的质量也有很高的要求。而这个功能本身也是建立在文件的备份和多版本管理的基础之上。

7. 需要有详细的发布历史和日志

发布作为一个重要的操作需要留下详细的发布历史和日志记录，便于后续对问题的回溯和审计。同时，如果发布过程中遇到问题，也是重要的定位问题手段，因为线上机器较多的情况下，难以一台台上去查看。

前面提到发布的工作可以由很多角色来承担，但是在我们的一些实践中，发布是由测试团队来完成的。更具体的有两种方式，一种是由一个专职的人来做，这种比较适合系统功能耦合较紧的情况，比如后端 ERP 系统，另一种是每个模块的测试负责人来发布自己所负责的需求。第二种情况之所以可以实现是在发布系统功能比较完善的情况下。这种情况下，发布这个比较耗费时间和精力的工作可以分散开来。

可能是站在测试的角度观念难免偏颇，但是由测试来做发布的工作对项目质量的提升是有帮助的，主要体现在以下几个方面：

1）研发环节对于功能细节比较了解的角色是产品、开发和测试人员。产品通常对技术操作的细节了解不多，不太适合做后台的发布。如果是开发来做发布，可能会比较随意，因为常会遇到需要小修小改的地方，这样就很有可能在其他人不知道的情况下产生外网发布，有些看似小的改动其实很有风险。测试其实是比较合适做发布的，一方面因为测试过功能的细节，比较了解将要发布的功能，且测试人员也有一定的技术背景。

2）发布的过程中也需要测试和验证。线上的运行环境和测试环境往往不同，所以会遇到测试环境验证通过的功能到线上出问题。这里面涉及的因素很多，比如服务器本身配置，基础软件版本，路由和负载均衡，以及正式环境对比测试环境的数据差异。当然，我们需要努力减少这种差异带来的影响，但是实际上难以完全避免。所以在做完发布之后，通常需要做一些相关的验证。如果是由测试人员来执行发布，这样过程就更加的无缝。如果不是由测试人员来做发布，也可以参与一起验证。

在发布完成之后，一个比较好的实践是发送一封上线完成的报告邮件，周知相关的各个项目团队成员，以及相关需求的干系人，包括管理层。同时，因为发布是一个重要的里程碑，表明该功能可以和用户见面了，也是告知产品和运营等角色可以继续后续的运作了。

以下是两个发布报告的样例。

图 6-10 所示的是一个人工编写的发布报告，包含了发布的功能列表，以及上线过程的一个简述。


0228发布报告

Hi，all:
今天发布从11:30开始推送到预发，到18:00推送完成。
发布过程大致如下:
1、今天发布内容包括

2、从11:30开始推送到预发，到15:20完成。
在预发环境更新了1次，14:20更新，原因:
在预发环境更新了1次，15:20更新，原因：发布包里多签入了代码，需要回滚。

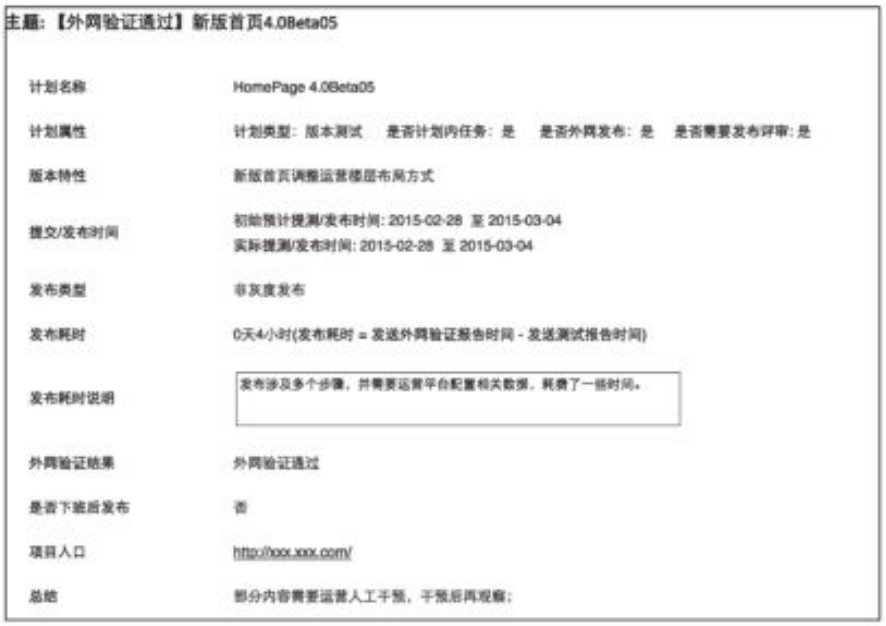
3、从15:15开始推送到线上，到18:00结束。
在正式发环境更新了1次，16:00更新，原因:并加一些日志跟踪。

本次上线内容列表

项目组	编号	项目	类型	标题	测试负责人	DIO（创建人）	开发负责人	验证负责人

图 6-10 人工编写的发布报告

图 6-11 是一个测试平台自动生成的发布报告，也包含了测试人员手工填写的部分信息。


主题:【外网验证通过】新版首页4.0Beta05

计划名称	HomePage 4.0Beta05
计划属性	计划类型：版本测试　是否计划内任务：是　是否外网发布：是　是否需要发布评审：是
版本特性	新版首页调整运营楼层布局方式
提交/发布时间	初始预计提测/发布时间: 2015-02-28 至 2015-03-04 实际提测/发布时间: 2015-02-28 至 2015-03-04
发布类型	非灰度发布
发布耗时	0天4小时(发布耗时 = 发送外网验证报告时间 - 发送测试报告时间)
发布耗时说明	发布涉及多个步骤，并需要运营平台配置相关数据，耗费了一些时间。
外网验证结果	外网验证通过
是否下班后发布	否
项目入口	http://xxx.xxx.com/
总结	部分内容需要运营人工干预，干预后再观察；

图 6-11 测试平台生成的发布报告

从以上信息可以看出，测试报告不仅能告知需求已经上线，而且可通过发布过程中的信息记录，对问题进行反馈，可以联合研发的各个环节来改进产品质量，提高发布环节本身的质量和效率。这些对于不断提升整个研发组织的能力和效率都是非常有帮助的。

6.2.2　App 的发布

在讨论完了后台发布之后，我们来看看 App 端的发布。相比较而言，App 端的发布要简单很多，主要是一个分发的过程，当然也包括后续的验证。下面以常见的 iOS 和 Android 两个平台来分别说明。

1. iOS App 的发布

iOS 平台的发布通常由对应的 iOS 开发人员来完成，发布的过程测试介入比较少。因为苹果的审核机制，App 提交审核后需要等约 1 周左右的时间来确定是否通过。相信做 App 时间长一些的团队都或多或少遇到 App 审核被拒的情况，其原因非常多，而且审核规则还在不断变化中，有些甚至不是特别透明，所以也需要不断积累经验，包括自身 App 过去的经验教训，以及网上别人遇到的案例。

在审核通过后，可以选择打开下载开关让用户来更新。测试在这时也会介入做一次类似线上验证的操作。主要覆盖以下场景：

- ❑ 全新下载安装，基础功能验证。
- ❑ 老版本升级安装，基础功能验证，以及登录态，收藏等数据的保留情况。

2. Android App 的发布

Android 情况不同，目前大部分市场基本没有审核的机制，但是发布的渠道要多很多。一般会考虑通过国内主流的应用市场来发布，另外，产品的官网也是重要的发布渠道。针对不同的市场，为了跟踪下载来源，便于了解各个渠道的效果，通常的做法是针对每个主要的市场打出带有单独标识的包。发布到个别市场，实际上也是一个 App 新版灰度的方式，关于这方面将在后面讨论。很多市场会有一些首发活动之类的运营方式，如果和这些市场有一些合作，也需要在发布环节做相应的控制。

6.3　内测

微软研发内部很早就有了 Eating our own dog food（吃自家的狗粮，意为公司内部使用自家开发的软件，特别是早期版本）的说法。这里我们把这个方法称为内测。内测是在前面提到的常规测试手段，主要是测试人员集中地进行功能、性能和专项测试之外的重要补充。其好处有很多，实践下来主要体现在以下几个方面。

1. 可以尽早发现更多的产品质量问题

这是最基本的目标。内测尽管是内部用户，但毕竟也是正式地使用，就会遇到很多真实用户会遇到的问题，这和测试阶段以测试为目标还是有所不同，有些问题因为涉及真实

数据，可能这时才会暴露出来。以 App 为例，测试阶段可能绝大部分的操作是在公司的办公环境，但是内测的用户可能带着最新版本的 App 回到家，或者在出差的路上使用。比如电商的 App，内测用户也会用它下真实的订单，完成整个购买的流程，而测试阶段可能只会走少量的真实订单。其他类型的 App 也是一样。由于是内部的用户，会知道明确的反馈问题的渠道，而且通常会非常耐心地描述问题，甚至可以当面拿着手机来讨论。

2. 发现一些用户体验类的问题

专职的测试人员对被测的 App 使用得非常熟练了，很多不好用的地方也习惯了，一些菜单也不觉得难找，会形成一些“审美疲劳”。而对于很多内测用户，新版的很多功能是比较新鲜的，甚至第一次接触，可以发现更多体验类的问题，而且因为参与的人数量更大，会有更多不同的角度。体验而非功能缺陷类的问题也会反馈出来，进而可以汇总给产品经理来看是否需要改善。

3. 增加大家对自家产品的了解，增进协作

通过内测，内部的同事对自己的产品有了更多的了解，这本身就很有价值，因为很多时候沟通的困难在于缺乏了解。对于很多大型的产品，涉及的模块和业务员非常多，常常不局限于无线产品研发团队本身，里面也会嵌入很多不同类型的功能模块。内测也是一个很好的增加产品了解的机会。

说了这么多好处之后下面我们来看看高效的内测要如何来开展。下面从几个方面来展开讨论。

6.3.1 内测的范围

首先我们讨论下内测的范围，包含两个方面：一个是从 App 研发的阶段来看，什么时候比较适合来做内测？第二是从参与的人来看，可以设定怎样的参与范围？

1. 何时启动内测

这个其实没有一个明确的标准，要结合产品单个版本的研发周期长短，以及产品的质量成熟度来考虑。对于内测，我们希望提供的是已经有了一定质量的产品，如果明知还有很多问题而发动较大规模的内测，可能会收到很多重复的缺陷报告，也浪费大家的时间，甚至对士气也有影响。另一方面，从时间上来看，如果马上就要正式对外发布了，启动内测也有点晚了，发现的问题已经流到了正式用户那里。

经过一段时间的摸索，对于 App 产品，我们的建议是发布前几天。这个时候主要的功能测试都已经完成，在最后的集成回归阶段，缺陷比较少。

2. 参与内测的对象

参与内测的对象也是一直需要去考虑的问题，可能包含下面几种：

- 部门内部的同事。部门内部的同事是最直接可以参与内测的，无论是否是直接参与 App 研发的。因为即使直接参与的同事，他涉及的也只是个别模块，一样可以去体验其他模块的功能。同部门的同事沟通更加方便，在启动时间上，甚至可以将更早期的版本用于这个范围的内测。比如只有部分主要功能测试完成，但是还有些功能在测试阶段，如果能明确和大家说清楚也是可以的。
- 公司内部其他部门的同事。对于公司内部其他部门的同事，沟通成本会增加，建议提供功能更完善、成熟度更高的版本。也需要更加明确地告知反馈的渠道。
- 外部粉丝用户。除了公司内部的同事，很多公司也在使用外部的粉丝群来作为内测的范围。这样的外部用户和灰度期间被选中的用户的差别在于内测是用户有意识地参与，而灰度对用户来说是被动的。反馈问题的积极性也会不同，对缺陷的容忍度也会不同。

内测是一个很好的质量提升手段，但是就过往的经验来看，不能对此有过高的期望，不同人反馈的问题基本是一个正态分布，有少数是比较重度的用户，另外有一些人少量使用，当然也还会有一些人并不会参与。当然这个也和内测的活动组织相关，包括宣传、激励等方面。下面也会做一些探讨。

另外，需要特别注意的是，对于部分涉及商业机密的功能，内测的范围要做严格控制，也可能需要做些开关处理，比较正式的可以考虑和参与的人签署 NDA（Non-Disclosure Agreement，保密协议）。

6.3.2　内测的实施

接下来我们讨论内测具体实施过程中的一些做法。包容 App 的分发、激励机制和反馈的收集。

1. 内部包的分发

App 和 Web 产品不同的是需要安装客户端，那么就存在一个测试包分发的问题。对于 Android 系统来说可以直接提供 APK 文件下载，参与的用户可以借助各种助手工具来安装，iOS 也可以借助 iTools 等工具来安装。在我经历的研发团队，内部开发了一套测试包的分包平台，实践下来是一套比较高效的方式。主要的特点如下：

- 提供手机网页版和 App 两种方式。手机网页版用户访问一个网址，然后用自己的个人域账号登录，就可以看到内测包的列表了，选中对应的版本来下载安装。App

的方式类似，这个平台本身有一个自己的 App，第一次测试的时候需要安装这个 App，并且登录。后续登录会保持，每次只要打开就可以看到内测包的列表了，一劳永逸的办法。

- ❑ 分发平台本身也带有反馈机制，用户可以借助这个平台来反馈问题。
- ❑ 分发平台也有一定的权限控制和审计的功能，控制哪些人可以参与，以及记录哪些人下载了最新的测试包。

另外，还可以借助二维码的方式来帮助内测包的分发，扫码下载也是一种很方便的方式。另外，也可以直接提供安装包的下载地址。

因为内测期间可能会需要多次更换测试包，所以上面的分发平台就能节省很多人的时间。很自然，这个平台其实不只是可以用于内测，对于测试阶段，需要更频繁地更换测试包，也有很大的帮助，只需要在每台测试机上提前装好分发平台的 App，后续测试包的更新就变得非常简便了。

在正式开始内测的时候，我们需要借助一些渠道来让参与的人知晓相关信息，结合实际情况，可以借助邮件、微信群、QQ 群等方式。提供的信息需要包括：

- ❑ 新版本特性说明：可以让大家知道内测的重点。
- ❑ 如何获取测试包：最好提供不同的方式。因为每个人的环境和习惯不同。
- ❑ 支持的平台和设备。
- ❑ 问题反馈方式：可以提供对应的内测接口人和收集反馈的邮件地址或者网址。
- ❑ 注意事项：有哪些需要注意的问题，或者明确的已知问题。

2. 内测激励机制

为了让内测更加有效和有趣，可以考虑一定的物质激励手段。对于内部用户和外部用户可以考虑不同的方式，这部分可以结合组织具体的情况来考虑。

3. 问题收集、跟进和反馈

通过前面一系列的工作，内测可以开展起来了，但是如果没有问题收集、跟进和反馈，整个价值也无法体现。测试团队可以作为内测问题的收集接口人，因为测试人员先对反馈的问题进行重现和判断。对于有多位测试人员的团队，可以设定一个接口人，然后就不同模块的问题进行分发。另外，如果研发团队有专职的质量管理（QA）团队，那么也可以请 QA 团队来做这个协调人。

如果没有问题的积极跟进，内测会流于形式。积极主动的问题跟进，除了高效地定位和解决问题之外，也是对内测人员劳动成果的一种尊重。

以图 6-12 所示是一个问题收集和反馈的邮件，去掉了具体的信息，供参考。

截止目前收到的内测反馈问题，请对应人员及时跟进并反馈进展！谢谢！

Hi All

主客户端V3.8 内测反馈问题收集

序号	反馈日期	平台	build版本	问题描述	反馈人	问题类型	跟进人	当前状态
1	2014/10/27	android	13082			非问题		
2	2014/10/27	android	13082			bug		已解决
3	2014/10/27	android	13082			bug		修复中
4	2014/10/27	android	13082			体验问题		修复中
5	2014/10/27	ios	15491			bug		已解决
6	2014/10/27	ios	15491			体验问题		已解决
7	2014/10/27	ios	15491			bug		

图 6-12　内测收集到的问题反馈汇总

6.4　灰度

在讨论了内测之后，我们来看一下在互联网研发领域广泛使用的一种质量管理的方法：灰度。灰度不是非黑即白，比如我们说一个显示器或者打印机有 16 级灰度，意味着从全白到全黑之间，还有 14 个颜色深浅的级别，从字面上也很形象的表达了其中的思路。其实这个方式在软件研发以外的领域早就广泛的应用。比如各种政策刚出台的时候通常有个试运行的过程，可以收集反馈做调整。新开通的一条地铁线也有一个试运行和设备调试的过程。这样的例子还可以举出很多很多。主要的思路就是让一个最终需要大规模发布的产品，先给一部分人（通常是一小部分人）用起来，看看反应和反响，然后考虑进一步的调整或者扩大范围，直至最后的全面铺开。

接下来我们分别看下 Android 和 iOS 的做法。

6.4.1　Android App 的灰度方法

对于 Android 来说，最直接想到的做法是基于不同的应用市场来做灰度，比如先只提交给一个小的应用市场，这样只有部分用户会收到。早期这样的做法确实可行，不过目前来看效果不太好，比较难以控制，特别是现在很多应用市场都有互相抓取的机制。

更实际的做法可以借助 App 的检查新版本机制来做。在目前的很多 App 中都有一个功能，就是在启动后检查自身是否有新的版本，进而提醒用户是否要下载安装。技术实现上也很简单，就是调用一个后台的 version 接口，请求中附带当前版本的信息，服务端返回最新版本的信息，客户端就可以来做比对得出判断。在这个基本机制的基础上，就可以衍生出一个 Android 灰度的做法，那就是服务端可以选择对哪些用户返回新版本。比如当前大部分用户安装的是 3.9 版本，而 4.0 版本已经测试完成可以发布了，但是我们希望只灰度给

部分用户试用。那么当用户手机上的 App 启动后向服务端询问最新版本的时候，我们给少部分用户的 App 返回 4.0，这部分 App 就会弹出提示询问用户是否更新。而对于大部分的 App 收到的结果还是 3.9，客户端就认为自己当前还是最新版本，这部分用户就不在灰度范围里面。粗略的描述是这样，实际中可以做一些更精细的控制，比如：

- 白名单机制——可以在后台设定哪些用户 ID 在或者不在灰度范围里面，进行比较精确的指定。通常我们会收集内部研发团队的账号 ID，放在这个白名单里面，让大家第一时间体验到正式版本中的新功能。
- 比例控制——最简单的办法可以基于用户 ID，后者设备 ID 的数字取模，得到一定的比例，比如 ID 除以 100 余数为 1 的用户，也就是这 1% 的用户会在范围里面。
- 有一定逻辑的筛选——为了让参与灰度的用户更加少而精，我们可能希望灰度的用户能够有足够的多样性，比如来自不同的地域，使用不同的网络类型，手机类型的多样性（厂家、型号、OS 版本、分辨率等）。这样在较小的灰度用户数量的基础上就能获得足够丰富的样本，灰度就更加高效，同时潜在的负面影响也可以被控制到很小的范围。当然，要做到这样的精选，对后台接口的逻辑也提出了更高的要求。
- 逐步灰度的能力——就像前面提到的，灰度也是分很多级的，对于我们这里提到的灰度也可以有一个逐步递进的过程，特别是对于有海量用户的系统，这显得更加重要。比如一开始，可以先灰度 1% 的用户，收集反馈信息，如果发现了一些问题可以快速修复提供一个新版本，如果反馈非常的正面可以进一步扩大灰度的范围到 5%，10%，逐步下去。针对这种情况，有一个问题需要注意，如果版本不断更新，尽量不要让一个用户处于多次灰度之中，避免在短时间内多次提示更新版本。
- 功能维度的灰度——前面讨论的是整体 App 的灰度，就是整个版本对用户可见或者不可见。还有一个更细粒度的做法是让 App 中的某一个功能对部分用户可见，对另一部分用户不可见。或者让两类用户看到的是不同的设置。这在业务上也是非常有价值的。技术的实现上通常是通过控制对应功能的显示入口来做到的，进一步也是通过控制接口返回的数据来控制的。

需要说明的一点是，虽然我们上面比较精确地设定了参与的人数或者比例，但是实际更新到新版本的人数会有很大差异，就好比一个触达消息存在一个转化率的问题，对于一个更新新版的提示，很多用户会选择忽略。所以设定的 1%，实际上可能只有千分之几。

6.4.2　iOS App 的灰度方法

iOS 的小范围分发借助企业账号来实现，很多内部使用的 App 也是通过这种方式进行正式的分发。但是对于外部用户的灰度，这种方式不太可行，而且会触发 Apple 相关的政策。

当 App Store 完成你的 App 审核以后，我们有时需要在内部灰度一下，而不是直接发布给终端用户。苹果支持灰度使用 App。我们可以向苹果申请 Promo Codes，这个 Promo Codes 是一串序列号，使用这个 Promo Codes 可以下载灰度 App。该序列号可以使用最长 4 周，直到当你发布了新版本的 App，或者你联系苹果服务人员去取消新版本。每一个序列号只能支持安装一台设备，你可以在申请的时候，申请最多 100 个 Promo Codes。详情参考苹果官网 : https://developer.apple.com/library/ios/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/Chapters/ProvidingPromoCodes.html

申请步骤如下 :

1）确保当前的 App 状态是等待开发发布或者准备上架。

2）用发布者的账号登录 iTunes Connect（发布流程和账号详情，这里就不再赘述）。

3）点击进入 My Apps，找到要申请的 App，点击“进入详情”。

4）点击上方的 version，在 Additional Information 中，点击 Promo Codes。

5）在弹出的对话框中，选择要申请的数量，点击 continue。

6）同意苹果公司的条款，接着下载一份文档，其中包含你所申请的 Promo Codes。

使用方法如下 :

1）进入 App store，下拉到底，点击“兑换”，如图 6-13 所示。

2）输入 Promo Codes，点击“兑换”即可下载该 App，如图 6-14 所示。

和前面内测章节讨论的一样，对于灰度我们同样需要及时地收集反馈信息，并跟进问题。

最后，想补充讨论一个观点。和一些人讨论的时候，甚至听到一个说法，因为有了灰度，可以不需要研发团队自己来做测试验证，交给用户就好了。这个听起来很美好，实际会有一些陷阱。

灰度不是“银弹”，不能太依赖这个方法。事实上我们所见到的面对巨大用户量的互联网产品也几乎没有不经过内部测试而在写完代码之后直接发布给用户的。

用灰度替代测试的想法很有诱惑力，特别是对于没有亲身经历过大型软件项目研发的人来说，但是实际上却行不通。主要有以下几个原因：

1）初次提交的产品质量往往达不到可商业化应用的要求。如果我们去分析研发过程中的各种 bug，会发现其中的原因有多方面，比如对于需求有不同的理解，开发人员自身的

编码能力和习惯不好，不断进入的新开发人员对技术和业务还不够熟悉，以及在编码的过程中并没有考虑到很多组合场景、复杂场景、异常情况。

图 6-13 App store 兑换券输入入口

图 6-14 Promotion Code 输入界面

2）如果很多用户在使用的过程中遇到问题，那本身就已经构成了伤害。而对 App 来说，因为发布更新的代价比较大，且周期比较长，特别是 iOS，伤害持续的时间比较长，

造成的影响会比较大。另外对于很多电商、金融等方面的 App，很多缺陷导致的损失是难以接受的。

3）用户会发现一些问题，但是大部分用户会沉默地选择弃用，能主动反馈问题只是一小部分用户，且反馈也会有时间差。

4）用户使用的路径比较窄，可能需要很大量的用户、很长时间的累积才能遍历到我们产品的各个功能点，这样发现问题的时间就会拉得非常长。相比而言，专业的测试人员因为有测试用例设计上的考虑，会在很短的时间覆盖各种不同的路径，虽然最终难免有所遗漏，但是整个效率要高很多。

当然，这个问题进一步展开是测试的价值和意义的问题，这里就不进一步展开讨论了，本书的最后一章也会有一些相关的讨论。

表 5-1 汇总了内测和灰度的特点，将几种范围的做法进行了比较。

表 5-1　内测和灰度做法对比

简要说明类型	公司内测 – 特性版本	公司内测 – 集成版本	外部内测 – 粉丝用户	外网灰度
进入标准	完成某个重要特性的版本，第一轮全部用例执行完毕且无严重待修复 bug	所有特性代码已合入主干，同时集成包完成最基本的验证，无阻塞性 bug	集成版本完成第一轮回归测试，无严重待修复 bug	达到发布标准的最终集成版本
范围	无线研发内部人员	公司内部人员	外部粉丝用户	一定比例的外部用户
发布方式	内部平台	内部平台	群文件，邮件等	灰度平台手动发布
问题反馈渠道	内部接口人	内部接口人	粉丝群消息，接口人	App 意见反馈，在线客服或客服电话
问题跟进人	特性测试负责人	App 整体测试负责人	粉丝群接口人 + 产品经理	QA+ 项目经理
退出标准	至集成版本内测开始	至全量发布	至全量发布	无严重问题，crash 率在较低水平

通过以上对比，我们可以看到对于几种不同类型的内测和灰度，在进入标准、参与范围和跟进方式上面都有一些不同。在实际项目中，可以结合具体产品的特点和团队的情况来调整。

6.5　本章小结

这一章我们重点讨论了从测试完成到全量提供给外部用户使用的过程中涉及的一些环节，以及对应的质量控制和提升的一些实践，我们统称为发布过程中的质量管理。持续集

成作为一个已经被实践证明有用并广泛使用的研发实践，对于移动互联网的产品研发依然非常有帮助，仍然是必不可少的。通常我们会非常关注发布后的质量，但是对于互联网产品，发布过程本身也会比较复杂，因为每次发布可能牵涉的文件和机器非常多，所以发布过程中的质量也非常值得关注。这里我们就后台和 App 发布讨论了发布过程中要注意的问题，以及打造一个良好发布系统的主要方面。

内测和灰度是互联网产品常用的两种重要的辅助手段，本章也讨论了一些具体开展的做法，包含一些实践和思考。这些做法看起来简单，不像前面的一些章节有“技术含量”，但是如果比较好地利用起来，会收到很好的效果，没有实践过的不妨在自己项目中做一些尝试，也可以采用灰度的策略，在个别项目或模块中尝试。

第 7 章 Chapter 7

质量的度量和推动

明智的人使自己适应世界；而不明智的人则坚持要世界适应自己。所以人类进步靠的是不明智的人。

——肖伯纳

这一章看起来没有什么技术含量，但我觉得对很多的测试团队，甚至是研发部门而言，如果想要快速提升研发质量，这些实践可能是能最快见到一些成效的方式，而不只是纯粹的软件测试的活动。因为从企业或者产品的角度，关注的是产品质量，而不是具体的方法。软件测试或者质量提升流程，这些都是为质量这个大目标服务的。也是基于这个原因，很多公司的测试部门称为质量管理部，或者技术质量部，质量提升部，等等，因为他们的职能已经不只是纯粹的软件测试。另外，质量，即便只是研发阶段的质量，虽然测试团队是承担了直接的责任，但是如果要做好，需要多个角色的协同努力。在本章，我们也介绍了在实际项目中采用过的一些实践。主要包括我们常用的一些质量分析的维度，以及从哪些方面来衡量整个研发阶段的质量和效率情况。基于这些数据，将会探讨一些我们实践过的质量推动的活动。接下来结合我们自己的实践，介绍了 QA 的角色和所做的工作。最后我们会专门讨论跨团队的质量推动，因为在一个大型的项目中，仅靠测试和质量团队几乎不可能做到高质量的产品发布，需要让相关的研发团队一起配合和执行。

7.1 质量的度量和推动概念

质量是一个比较笼统的概念，出现的问题可以通过一些现象反映出来，比如外部用户

的投诉比较多，发生线上故障或者事故的次数比较多，测试阶段发现的 bug 比较多，导致需要来回地出新版本和回归。如果不做一些量化的分析，比较难以准确地衡量。更重要的是，通过不同维度的衡量我们可以知道哪方面的问题比较多，进而找到改进优化的方向。另外，有了这些度量数据，我们也可以评估质量是不是在往好的方面改进。

质量的度量有很多不同的方面，如果按照问题的发现阶段来分，可以分为运营阶段的问题和研发阶段的问题。上线之后的问题我们将在下一章关于发布之后的质量管理来讨论，在这里我们重点讨论研发阶段的质量问题。

研发阶段的质量问题，最直接的是通过缺陷（俗称 bug）来体现。也许现在还有人在讨论要不要将 bug 记录到一个 bug 管理系统，如果直接口头地告知开发人员，然后快速地修复是不是效率更高？ 因为没有了 bug 的填写和回复过程。这个听起来有道理，但是实际项目运作一段时间之后就会发现其中的问题。主要的问题是：

- 没有一个记录跟踪的机制，问题很容易被遗漏，以及不知道后续修复的情况。
- 无法衡量有多少问题，对于整个研发项目的管理非常不利。
- 一些更深层次的影响，可能会让功能的提交和修复变得非常随意，看似口头的沟通很快，但是如果有大量这样的讨论其实效率更低。当然，验证这两种方式最好的办法就是在项目中试运行一段时间。就我们接触的研发组织，基本都选择了使用 bug 管理系统来记录研发阶段所发现的 bug。

7.1.1 质量数据的度量

bug 数据是研发过程中很重要的过程资产，不仅反映了测试的工作成果，从中也可以看出很多整个研发流程的问题，非常值得去挖掘分析。在一些稍大的研发团队，通常都有专职的质量管理角色，也就是所谓的 QA。

QA 通过不同维度来分析缺陷的数据。可以有非常多的维度来分析，建议结合研发团队自身的需求，每个分析维度都应该有具体的价值和意义，而不应该只是为了数据而分析数据。

数据的具体获取方法可能有下面几种：

- 缺陷管理系统本身有比较丰富的报表功能，并且可以根据条件来定制。
- 从缺陷管理系统导出数据表格，然后在 Excel 等软件中进行手工的分析
- 在上面第二种方法的基础上，开发一些自动分析的脚本，快速计算感兴趣维度的数据。

下面列举了一些我们在实践中采用的维度，作为参考。

1. 全局统计数据

图 7-1 所示是一个比较全局角度的 bug 数据样例。这样的数据可以让大家看到一个整

体的缺陷情况。

V3.0 版本Bug统计情况

- V3.0 版本总缺陷数 531个，有效缺陷510个，有效率96%。
- 未关闭bug（新建+已解决+重新开启）【排除挂起和已关闭】7个。
- 一般级别以上有效bug 452个，其中严重+致命bug 137个，占总数的25.8%

系统名称	已关闭	挂起	新建	已解决	重新开启	总计
iPhone客户端	181	13	2		1	197
Android客户端	210	18	1	1		230
服务端	72	7	1			80
CMS系统	22	1		1		24
总计	485	39	4	2	1	531

图 7-1　版本缺陷统计情况

整体的数据是重要的参考，但往往也因为数据笼统而难以有具体的推动。

通常的情况是这样的：

QA：最近这个版本 bug 很多。

开发 Leader：呵呵，是的。不过这个版本新增的功能也确实挺多的，中间还有不少的需求变更。另外，我们最近还来了很多新人，对业务和我们用到的技术也不是很熟悉。

QA：也是……

然后，就没什么然后了。

所以我们需要从更多的维度给出更加具体的信息。

2. 按 bug 类型来统计

通常在提交 bug 的时候都会有一个字段来选择 bug 类型。一般是测试人员在提交的时候填写，但是有可能测试一开始选择的 bug 类型并不准确，所以开发人员修改完代码，在系统里面回复 bug 的时候也可以调整这个字段的选择。

基于每个 bug 的类型，我们可以得到一些统计数据，如图 7-2 所示。这个数据可以帮助我们了解哪些类型的缺陷占比较高，这些都是我们可以去努力优化的方向。

3. 按模块和系统分布情况来统计

按功能模块来统计 bug 数量也是一个比较常见的维度，如图 7-3 所示，可以帮助我们知道哪些模块的 bug 比较多，进而可以重点分析这个模块存在的问题。

bug类型	个数
程序逻辑错误	109
页面显示错误	58
用户体验优化	45
程序接口错误	32
需求描述不清晰/不准确	31
性能问题	25
页面跳转错误	23
兼容性问题	22
需求未实现	21
页面加载异常	16
其他	15
需求遗漏	15
程序算法错误	14
测试数据问题	11
程序提示信息错误	10
程序校验错误	8
测试环境问题	4
需求变更	3
操作问题	3
发布类问题	3
安全类问题	2
总计	470

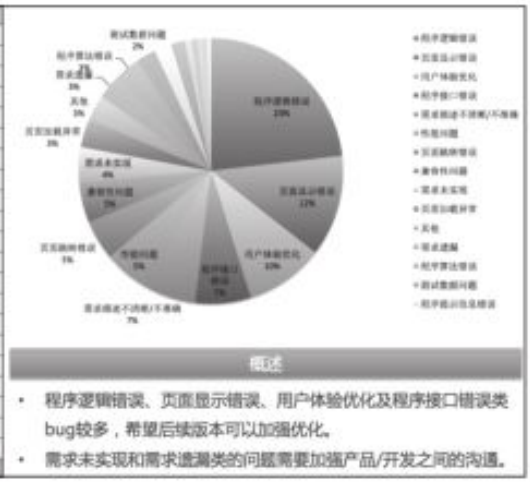


图 7-2 按 bug 类型的统计结果

一般级别以上bug按模块-系统分布					
模块	iPhone客户端	android客户端	服务端	CMS系统	总计
首页	13	17	5	6	41
商品详情也	21	26	9	8	64
购物车	18	14	11		43
搜索	14	16	7		37
订单页	9	11	8		28
订单查询	11	12	6		29
运营活动	20	21	3	13	57
分类	12	16	10		38
充值	4	6	3		13
我的收藏	3	4	2		9
评论	5	7	1		13

图 7-3 bug 按模块 – 系统分布统计结果

4. 按开发和测试人员的维度统计

这个维度是基于人来统计的，前提是统计所有的有效 bug，可以看出哪些测试人员提交的 bug 比较多，如图 7-4 所示，或者也可以统计哪些开发人员引入的 bug 比较多。针对数据比较突出的可以进一步来分析原因并制定改进措施。

通过以上数据我们可以清楚地看出哪些人发现的 bug 比较多。对于功能测试的工作量估计也有帮助。结合测试的时间和各模块的情况可以进一步分析我们测试主要的时间花在哪些方面，从而帮助进一步提升效率。

5. 按模块缺陷详细统计（严重程度、自测可发现和新引入）

这是在上面按模块统计的基础上的一个细化维度，如图 7-5 所示，除了 bug 严重程度

之外，还引入了一些提测质量的自定义的衡量指标，比如开发自测可发现 bug 的情况，以及修改引入的回归性 bug 的情况。关于开发自测可发现 bug 这个维度，后面也会有一些讨论。

报告人	个数	报告人	个数
Annedli	141	Gaoweili	36
Amydwang	124	Barretshen	18
Yoyoyang	112	Junglezhang	18
Allenjchen	100	Autowang	14
Shawnxmpan	93	Kevindlxie	12
Carozuo	76	Piaozhou	12
Joshuazhu	74	Jiaqizhang	12
Forrestguo	73	Lianchen	11
Emilylei	72	Jasoncui	11
Lemonguan	72	Winstonwu	10
Xunpingping	70	Xiaoftang	7
Liujunping	69	Wizardyao	7
Zhaowei	57	Donishen	6
Cececai	55	Yanrzhang	5
Joycezhen	52	Cocokong	3
Wanghui	51	Kennyfu	3
Wuhoushi	41	Bestfei	3
Rickyqiu	39	Johnwang	3

图 7-4　按测试人员维度统计缺陷

按模块详细缺陷分析

模块	平台	bug总数	一般以上bug总数	一般以上有效bug数	一般以上有效bug率	自测可发现bug数	在开发自测时外中	可自测发现率	修改新引入bug数	新引入bug率	致命+严重bug数	致命+严重bug率
应用组件	android	43	38	37	97%	7	3	18%	2	5%	13	35%
	iphone	31	30	28	93%	6	3	20%	3	10%	17	49%
	服务端	5	5	5	100%	2	0	40.00%	0	0%	3	60 %
购物车	android	34	30	29	97%	7	4	23%	7	23%	11	32%
	iphone	40	35	33	94%	2	1	6%	5	14%	9	23%
	服务端	5	4	4	100%	0	0	0%	0	0%	1	20%
	CMS	7	6	5	83%	0	0	0%	0	0%	2	29%
首页												
商详页												
结算页												
订单页												
我的订单												
列表												
搜索												

图 7-5　模块维度 bug 详细分析结果

6. 针对一些重点模块或者项目做更深入的分析

可以从 bug 的类型、根源和发现阶段等角度进行更深入的分析。这里不再详细举例说明。

7. 基于开发工作量的缺陷统计

实际项目中，如果我们拿着按模块区分的缺陷数据和开发 leader 讨论，很可能遇到的

一个挑战是，他们会觉得不公平，因为不同模块的工作量差别很大。比如 A 和 B 两个模块，A 有 10 个 bug，B 有 5 个 bug，但是 A 是两个人开发了两周，B 是一个人开发了两天。虽然 A 的 bug 绝对数量比 B 多，但是如果按比例来看，A 的开发质量要比 B 好。这样说也确实有道理。所以通常在分析中，会用到一个指标，叫做“千行代码缺陷率”。计算方式是用某个模块的缺陷数量，有的还会根据 bug 的严重等级（致命、严重、一般、轻微）进行相应的加权，除以变更的有效代码行数（非空非注释，以千行为单位）。这里相当于是用有效代码行数来作为一个工作量评估的替代品，因为工作量是一个很难精确度量的指标，软件开发是一个纯智力的劳动，不太合适用工作时间的长短来度量，而且不同的人因为能力或者熟悉程度的差异，单位时间的有效产出也会差别很大。在这种情况下，代码行数是一个相对有效的度量。通常不会有人刻意的去操纵代码行数，当然也可以做一些 review，另外统计的时候也要剔除一些明显异常的情况，比如代码目录管理的变更导致的大量更新，或者 copy 引入大量的其他代码。

8. crash rate 数据

对于 App 来说，crash rate 是一个重要的质量指标，因为 crash（俗称闪退）对用户体验的影响比较大，会导致用户操作的中断，可能引起数据的丢失，也非常影响产品的口碑。所以非常需要对这个方面的数据进行统计。

crash rate 的计算方式通常有两种：

1）一段时间累积的 crash 次数 / 活跃用户数

2）一段时间累积的 crash 用户数 / 活跃用户数

方式 2）和 1）的区别在于，一个用户可能会在一段时间内遇到多次的 crash，甚至是同一个问题导致的 crash。 所以通常 1）的数值要比 2）高。这个指标的选取可以结合团队的理解，因为是从不同的维度来反映问题，也可以两个指标都给出来。

从上面计算方式可以看出，这个指标的获取取决于两方面的数据：

a）活跃用户数，也就是通常说的 UV，这个可以从 App 的后台来获取。

b）crash 的数据，这个通常依赖于 App 的异常上报逻辑，因为 crash 是发生在 App 端。

关于这个数据的使用，首选需要区分 iOS 和 Android 平台，另外可以从不同的维度来参考：

- 和可获取的业界数据来比较。
- 和同一个平台之前的版本比较，看是否有改善。
- 按时间维度来看趋势，比如按天 / 周 / 月，来观察同一个版本在一段时间的走势。
- 更细粒度，可以按 build 号来统计，因为对于同一个大版本，在灰度期间可能会发

出多个小版本，所以需要更精确的度量。

7.1.2　质量推动的活动

上面我们把质量数据透明出来了，但是如果只是把这些邮件发出来，可能很多人不一定会仔细看，即便是直接相关的人也可能遗漏。质量的推动不能只停留在上面静态数据的提供，而需要一些互动的形式，引起足够的关注，以及通过沟通和讨论确认问题，共同找出改进的措施。以下是我们尝试过的三种类型的会议，包括针对所有研发 leader 的质量会议、质量周会，以及针对具体项目的总结会。

1. 针对所有研发 leader 的质量会议

这个会议是邀请所有项目管理、产品、开发和测试的 leader 参加，频率不用很高，比如一到两个月一次。会上可以同步主要版本的缺陷分布情况，以及遇到的主要质量问题，包含外部用户针对系统方面的投诉，以及 crash rate 等数据。并且可以讨论一些质量提升的流程，让大家从不同角度发表意见，进而达成一致。如果能得到各个直接参与团队 leader 的支持，后续质量提升措施的推动和推广都会更有效率。

2. 项目总结会

也称为 Postmortem，很多在外企做过研发的同学都应该对这个名词有些了解，在项目结束后一群人来做对应的总结和反思。这是一个很好的实践，集中项目的直接参与人，大家一起总结和反思项目过程中的得失，进而找到可以优化的措施。需要注意的是应该有具体的可以落实的做法，不求多，最终有几点具体的改变就非常有价值。

3. 质量周会

除了上面的项目总结会，实际过程中，我们有时会觉得等到项目结束再来反思已经有点太晚了，还有一个原因是互联网的产品，项目之间衔接比较紧，甚至有交叉重叠，所以通常大家无暇做太多回顾。而对于移动互联网产品，两个大版本之间的周期相对较长，于是在一些实践之后我们把这个质量会议过程放到了项目的研发过程之中。

会议上用到的材料是质量数据分析的结果和具体的 bug 列表，以及一些影响比较大或者典型的问题。

图 7-6 所示是一个质量数据分析会议的会议纪要。从中我们可以看出，这个质量会议的跟进措施是包括了产品、开发、测试、QA、项目经理多个角色，需要大家从不同的维度来提升质量和效率。

上海质量数据及流程讨论会			
会议时间	2015-2-12 17:00-18:00		
会议地点	上海：1204会议室		
主 持 人	【QA】		
记 录 人	【QA】		
参 会 人	【对应项目经理，产品，开发，测试，以及相关团队负责人】		
会议议题	议题	讨论要点	
	质量数据及流程讨论	1. 部分模块bug分析 2. Bug的建议以及处理流程	
会议结论	事项		负责人
	1、产品经理有需求变更，请不要找开发口头描述直接修改，请首先更新需求文档，然后同时通知开发和测试，避免测试提了bug后才发现有需求变更；		产品经理
	2、功能未全部开发完成或者存在外部依赖无法测试的，请开发同学尽量提前在提测邮件中明确，测试可以先不测这块不提此类bug；		开发人员
	3、项目经理组织测试用例评审，直接的开发人员和产品经理建议必须参加，可以对测试用例提前确认；		项目经理
	4、体验优化类bug请测试人员先与产品和视觉多沟通确认后再提bug；		测试人员
	5、类似代码规范或无用代码类的建议类bug、以及埋点类的bug测试人员可提在同一个bug中进行跟踪；		测试人员
	6、建议开发人员修复bug后，在Jira备注中备注一下问题的原因和解决方法；		开发人员
	7、建议开发同学在开发过程中可以使用静态代码扫描工具进行自检，提前将严重问题修复；		开发人员
	8、建议在Jira中能够区分功能模块历史遗留bug和当前版本引入的bug；		QA

图 7-6 质量数据分析会议结论示例

7.2 QA 的角色

QA 在整个研发组织的质量管理和提升方面是一个非常重要的参与者，甚至有时是主导者，但由于本书不是一本纯粹关于质量管理的书，所以这里我们只是对质量管理（通常称为 QA，或者 SQA）这个角色的工作做一个比较简单的介绍，主要也是从测试团队的角度出发。

关于 QA 具体应该做哪些工作，这方面同样没有什么标准，非常需要结合项目和团队的实际情况来确定。总的思路是相比通用的数据整理等？比较浅层的工作，贴近具体的开发和测试团队的 QA 角色应该提供更加深入和个性化的服务。以下是我们在移动互联网产

品的测试中对于 QA 工作内容的一个定义。

1. 跟踪研发流程落地情况

按功能点维度跟踪“开发自测”，“设计走查”和“产品走查”等质量提升实践的落地情况，包含开发自测用例执行情况，走查发现问题的数据等情况，并汇总相关数据。相关的统计数据会附加在版本 bug 分析报告中。

除了数据统计之外，对于那些没有执行相关的质量流程的模块，需要跳进去了解具体的原因，看是否遇到什么样的困难，以及流程上有什么需要调整的地方。

2. 测试内部流程执行的情况

按功能点维度跟踪“测试用例评审”，“开发自测用例”准备，“测试进度报告”以及”专项测试”的执行情况并进行汇总。这方面的数据和项目的进度数据一起按模块列出。

3. 组织新版本内测活动

在前面章节我们讨论了内测的实践，QA 是一个很合适来协调这个质量提升活动的角色。需要组织内测活动的开展，在内测开展期间按 build 版本按日期收集用户反馈，并推动问题的解决，最终保证对外发布版本的质量。同时，收集用户反馈并进行统计是进行有奖内测活动的基础。

4. 版本缺陷分析

这部分也是传统质量管理中的一个重要内容。细分下来包括以下方面：

1）bug 管理平台中 bug 属性的定制和日常维护。

2）bug 数据统计，参考上面提到的质量数据各种维度的度量。

3）版本发布前对未关闭 bug 进行分析和推动，让管理者及时了解 bug 解决情况，将是否可以进行版本发布进行量化，为领导决策提供指导。

4）版本发布前组织对挂起状态的 bug 进行集中审核，确定挂起 bug 的影响程度，另一方面督促继续跟踪历史挂起状态的 bug。

5. 质量推动活动

如前面章节讨论的，QA 可以来组织各种形式的质量 review 会议，不定期地发现流程执行中的问题，从而进一步优化。

6. 线上发现的问题的管理

QA 关注的不只是研发阶段的质量问题，同时也关注运营阶段的质量，更多的是外部用户反馈回来的问题。分为以下三部分：

1）制定流程规范，包括重大事故管理流程及线上问题处理流程等。

2）流程跟踪和实施推动：用户反馈及其他渠道反馈的重大问题录入事故管理系统并跟踪推动解决，督促开展事故分析会，对应的责任方需要发出问题根源报告，避免重复发生。

3）所有问题通过系统或表格记录，每月进行汇总分析，评估系统运营质量情况，也是测试漏测率数据的一个来源。

7. 组织交叉测试及测试内部激励评选

在后续章节我们会谈到相关的实践，这些也是质量提升的辅助手段。

从立场上，QA 应该站在一个更第三方的角度，即便当 QA 是归属于大的测试，或者质量管理部门。应该去审视和调查整个研发流程中容易产生质量问题的地方，并且尽量通过数据的方式展现出来，推动优化。而从测试团队负责人的角度，如果希望团队有提升，也应该鼓励暴露出测试自身的问题，比如 Bug 本身的质量是不是够高，以及外网问题是否漏测等。

7.3 跨团队的质量推动

按照我们前面的讨论，质量的提升不只是测试团队的事情。但是只是笼统地喊口号，可能并不能带来具体的效果，所以需要落实到具体可操作的措施，以下是一些实践证明可以应用的一些方法。

7.3.1 开发自测

比较理想的情况下，开发人员写完一个模块的代码之后，都会做自测，包括单元测试和模块集成到一起的测试，后者类似于专职测试人员做的黑盒测试部分。然而理想和现实之间总是有一些差距，实际的情况是，开发提交给测试的版本里面，常常 bug 较多，甚至有些比较极端一点的情况，主要功能流程都跑不通。类似的情况，做测试时间稍微长一点的人估计都会遇到。很显然，这样的情况会严重影响研发效率，而且通常如果提交测试的时候质量很差，测试阶段发现 bug 较多的话，最后发到线上的质量也不会太好，这就是典型的软件缺陷的群集现象。所以质量提升很重要的一点就是提高开发提交测试的版本的质量，正所谓质量是 built in 的，一开始就把事情做对才是更高效的方式。

除了开发人员自身的能力和经验提升，以及开发内部的各种提高质量的技术和流程方案，从测试角度，我们有什么方法可以提高开发的提测质量？仅仅停留在口号是不够的，接下来我们看看在实际中用到的一些方法。

1. 建立一个度量的指标

这里我们使用的指标是：开发自测可发现 bug 率。计算方式是这样的：

开发自测可发现 bug 率 =（测试提交的 bug 中认定为开发自测可发现的数量）/（测试提交的总 bug 数）

以上可以针对一个功能模块来统计，也可以针对整个版本来统计。

为了统计有多少 bug 是属于开发自测应该发现的，我们在缺陷管理系统里面自定义了一个字段，如下图 7-7 所示，“开发自测应发现”，测试人员在提交 bug 的时候需要根据实际情况进行选择，默认为否。

图 7-7　bug 填写中的开发自测可发现字段

接下来的问题是这个字段的选择，测试人员如何来判定：

- 在初期的时候我们让每位业务测试人员自己来判定，但是给出一些指导意见：那些在主功能流程上的 bug，或者显而易见的 bug。这样的结果难免会有一些主观，不过经过一段时间的此类 bug review 和调整，数据基本可用，这样的统计就建立起来了。
- 如果我们可以给开发人员一些自测用例，那么在这些自测用例里面的 bug 就属于开发人员自测可发现的 bug。这种方式的范围设定会更明确一些，但是需要依赖开发

人员自测用例的机制。

2. 提供自测用例给开发

大部分的开发人员也都非常希望自己提交的代码有比较好的质量，而且以之前的经验来看，很多开发人员也会在代码完成提交测试之前进行一些基本的验证，总体来说会做得比较零散，因此也有部分开发人员会参考测试人员的用例。但是测试人员在做测试用例设计的时候需要考虑各种各样复杂的情况，以及异常情况，所以对一个功能点来说，用例的数量通常比较大。如果直接给开发人员用于自测，可能耗费的时间会比较长，所以在实践中，通常会给出测试用例的一个子集。而且，采取比较灵活的做法，主要考虑以下几点：

- 一般我们只给出覆盖主要功能点的核心用例给开发人员做自测。
- 对于特殊模块我们会给出更进一层的用例，比如：开发人员是新人或者新接手这个模块，或者全新编写的模块。
- 通常我们会把全量的测试用例给开发人员，主要针对一些由开发自测保证的模块，另外对于一些上一版缺陷过多的模块也会采取这样的方法。

当然，这样的方法也必须得到研发管理层（包括开发负责人）的同意，需要事前进行相关的讨论和沟通。

在前面提到的研发流程中，通常需求评审通过之后，测试人员就开始编写用例。在开发编写代码的过程中，测试用例也完成了，这时测试人员会挑选一部分测试用例用于开发自测。为了后续的执行和结果填写的方便，用例通常采用表格的方式，另外，也会附上全量的测试用例给开发人员参考，如图 7-8 所示。

图 7-8　准备开发自测用例

有了开发自测用例的机制之后，开发在提测前要完成这些测试用例的执行，并在提测的邮件里面附上用例的执行结果，如图 7-9 所示。

图 7-9　包含自测用例结果的开发提测邮件

如果团队的自动化程度做得比较好，其实可以通过自动化测试来做提测的验收，就像前面第 6 章中持续集成小节谈到的。另外，我们也曾经尝试过更进一步的实践，那就是帮开发人员部署一套自动化测试，针对开发环境来运行，开发人员可以在测试人员不干预的情况下自己运行测试用例集，如果发现问题会先行修复，而不是等到提交给测试人员的时候被发现，这样就更进一步提高了效率。但是，如果要做到这样，对自动化用例集的稳定性和可维护性要求比较高。

3. 评估效果和反馈

如果开发和测试团队达成共识通过开发自测用例的形式来保证提测的质量，那么就取决于测试用例的执行情况。基于 Trust and Verify 的原则，从质量管理的角度会进一步从两个角度来衡量执行的效果：

1）开发自测可发现 bug 的数量和比例有没有下降？这个指标可以做横向的对比：哪些模块比较多，或者占比较高？ 以及纵向的对比：对应某一个模块的不同版本或者某一位开发人员的 bug，这一指标的变化情况如何？

2）有哪些具体的 bug 应该是通过自测用例可以发现的，但是还是在提测之后被测试人员发现？对于在这个列表里的 bug，需要拿出来逐个进行 review，看具体是什么原因导致本应该被自测用例发现的 bug 还是流到了测试阶段。 是因为开发和测试人员对于测试用例的理解有分歧？还是因为环境和数据的限制导致开发人员难以执行对应的用例？如果遇到这种情况，开发人员需要明确地标注，以及对应的沟通。

任何一个质量提升的流程，如果没有后续的数据度量和结果的闭环反馈机制，就往往容易流于形式，达不到相应的效果。反之，如果能像持续集成一样，有明确的数据统计和及时的反馈，对质量的推动很有帮助，同时，参与其中的人员也可以及时地了解到效果，也是很好的鼓励和促进。

7.3.2 设计走查

上面我们讨论了关于通过推动开发自测来提升研发阶段的版本质量。质量包含很多的维度，而整个产品研发的过程有很多不同的角色一起参与，那么他们是否可以一起帮助来提升质量？

实践证明确实可以，而且有些是其他角色无法替代的。在一个正式发布给外部用户的产品，特别是面向普通大众用户的产品，设计（包括交互和视觉）都是非常重要的部分，对整个产品的成功有非常重要的影响力。那么如何保证实际做出来的东西符合设计的需求？恐怕设计人员自己来验收最合适。于是，在研发流程中，我们推动引入了“设计走查”的环节。通常是在开发完成提交第一个版本的时候，请设计师来实际使用产品进行视觉和交互的走查。对于发现的问题，设计师会详细地记录下来，并且回复开发人员的提测邮件，进而开发人员再基于这些意见进行调整。当然，对于一些有争议的部分，可以拿出来讨论。

图 7-10 是一封设计师回复的走查结果邮件示例，内容模块特性做了部分模糊化处理，但是设计走查的问题描述是真实的。从这里可以看出，设计走查对于追求用户体验和设计感的 App 来说是非常重要的流程手段，同时也是产品开发团队和设计团队加强沟通的一个重要机制，因为有了这样的闭环，会有更多的交流和提升。

7.3.3 产品走查

与上面讨论的设计走查一样，我们在开发第一次提交之后也加入了一个质量环节，邀请产品经理来体验新开发完成的功能，看是否和自己的需求一致，有没有错误和理解偏差的地方，我们称之为“产品走查”。同样，也是在自测邮件的基础上回复，请对应的开发人员来跟进。

关于开展这个实践的时机，严格来讲应该是在测试开始之前就完成，且所有问题得到修复后再进入测试阶段。避免测试发现重复的问题，浪费时间。但是实际项目中，通常提测到发布的时间很短，如果把设计走查和产品走查都串行起来做，时间上难以保证。所以，通常我们会并行来做，但是会要求设计师和产品经理在开发提测后的一段时间之内完成相应的走查工作并给出邮件回复，通常我们建议 24 小时内给出回复。

和前面开发自测的实践一样，我们也需要一个数据的统计和反馈机制，所以在每一个

版本的研发阶段，QA 人员都会统计如下数据：

- 有哪些模块进行了设计走查和产品走查。
- 设计走查、产品走查发现的问题数。

【视觉走查报告】Android首页

Hi All
安卓版首页视觉走查结果如下表。
详细的对比走查报告在附件中，请查收。谢谢。

Android首页走查结果

页面	模块	问题描述	优先级	状态
	焦点图&搜索框	1.分类和扫一扫icon模糊了 2.搜索框的圆角矩形不是6px的圆角，高度也不是60px；搜索框内的icon被横向压扁了。 3.焦点图没有放左右切换的功能圆点。 4.搜索框内的文案字符颜色应该为#848689 5.顶部搜索框变灰的速度可以再快一些，在焦点图快不见时就完全变成黑色半透明状态。	P0	待修改
	今日推荐	1.分割线颜色不对，应该是#d7d7d7，两根分割线中间应该有#f3f5f7的灰色色块。 2.两个元素距离分割线的距离应该是20px，秒杀距离今日特价的距离应该是22。秒杀距左边边距应该是8px，和今日特价对齐。 5.倒计时的矩形和秒杀的距离应该是18px；"全部商品"的字号大了，应该是25，且颜色应该是#686868 6.向右箭头被拉伸过了，模糊了；商品图的圆形模糊了，且应该没有灰色描边；商品描述的字符控制在265px以内显示，超出部分显示省略号；	P0	待修改

图 7-10　产品走查结果示例

这些数据可帮助我们评估这两项实践开展的情况以及实际的效果。

7.4　本章小结

这一章的内容不多，看起来也比较简单，但是如果要按照质量提升的效果和付出的代价作为维度衡量性价比，那么我觉得这一章里面的一些实践估计是最高的。作为技术人员，我们一直坚信技术的力量，这样没有错。但是在实际项目的质量提升中，技术可能是从及格到优秀必备的，而上面提到的各种质量推动的流程，也许能比较快速地帮助一个质量较差的产品先做到及格。另外，在这个过程中，整个团队也能更好地磨合和协作。

进一步的思考，还有一个观念的问题，过往的经历，很多时候测试团队，特别是比较

技术导向的测试团队，往往容易把测试自身的技能提升当成唯一的质量提升的方式，完全的投入到测试技术和工具里面。这方面的努力是值得的，但是或许可以用更 open 的思路来做质量管理。而且，如果和研发流程的其他角色更多的沟通和交流，我们会发现，其实大家都非常的关注质量，而且很多工程师发自内心的也希望做出高质量的产品。所以应该更多的协作，在质量提升的不同方面和方法上。

第 8 章 Chapter 8

发布之后的质量管理

这不是结束，甚至不是结束的开始，而是开始的结束。

——丘吉尔

当我们的产品经过重重考验终于发布上线了，但其实这个时候我们所经受的真正的质量考验才刚刚开始，这是由互联网产品的特性决定的：7×24 小时的在线服务。也正是基于这样的产品特性，对于测试的一个思考是，我们不能局限于研发质量，而是要延伸到线上运营质量，用运营的思路来做移动产品的测试。在本章我们会介绍几个方面的实践。首先是上线后我们会继续做一些功能模块的交叉测试，期望在用户之前继续发现一些问题，对于测试团队内部也是知识和分工的交换。接下来我们会重点介绍监控的开展，也会针对接口的测试详细介绍我们的一些实践。对于互联网产品而言，监控不是一个辅助手段，而是一个非常必要的功能，而且在实际中，我们了解的每一个大型系统都有比较完善的监控机制。最后我们会来讨论如何收集外部用户的反馈，以及实践中我们常用的跟进机制。

8.1 发布后的交叉测试

实际项目和团队运作过程中，大家通常按照平台或者功能模块来划分测试人员的工作。这样做非常自然，也有很多的好处，比如一段时间一个人可以专注在一个小的领域，对于相应的业务功能和模块使用的技术都有比较深入的了解，便于测试做得比较深入，对于个人来讲也是一个积累的过程。但是凡事都有利弊。不利的地方可能包括如下问题：

- 某个人对一个功能测久了，很多东西会觉得习以为常，或者有通常所谓的“免疫”，有些缺陷发现不了。另外每个人有自己思维的盲点，很多路径考虑不到，测试用例评审也只能帮到一部分。
- 对于个人的提升也有瓶颈，接触的东西会比较单一，包括技术和业务方面。
- 对所分配的模块通常都会设计用例进行比较系统的测试，反而少了一些探索性的测试。
- 团队成员间对功能模块互相了解不够，遇到边界的问题容易遗漏，也不利于团队的工作轮换和备份机制。

基于上述的这些问题，一方面我们会定期做一些模块的轮岗，另一方面我们也可以尝试一些交叉测试的方法。所谓交叉测试，是指在测试的某个阶段，测试人员之间相互交换测试的模块，换一个人换一个思路，而且交换者也可以发现交叉测试的模块和之前测试的模块之间的联系，从而构建更多的测试场景，发现更多的问题。这对质量提升很有帮助，同时也可以促进团队之间的沟通以及知识的共享，是一种很好的方式。

接下来要考虑的是，交叉测试在什么时候引入比较合适的问题？一般情况下前期的功能测试是每个测试人员和对应功能模块的第一次接触，有很强的新鲜感，在对模块进行初步了解后可以快速发现较多 bug。反过来如果此时立即开展交叉测试，测试者对功能模块逻辑不熟悉，导致测试时间比较紧张，所以此时不宜进行交叉测试。但是在经过功能测试后，一般的 bug 都会被找出来，功能渐渐趋于稳定，产品逐渐定型，但是此时还可能会存在一些 bug，由于受到测试惯性的影响很可能会被忽略，此时引入交叉测试可以发现一些遗漏的 bug，同时发现和之前测试模块之间的联系，构建新场景，所以在功能测试后，发版之前引入交叉测试比较合适。但如果发版时间比较紧张，在发版之前来不及开展交叉测试的团队，也可以放在版本灰度期间或者下一个版本启动之前进行。因为对于移动互联网的产品有一个 App 的提交和发布时间，所以时间节点可能会比 Web 互联网的产品要更明确一些。在一个大的 App 版本发布后，会进行下一个版本的需求评审和测试用例准备等工作，可以安排出一小段时间来做交叉测试。以下是我们的一些实践。就结果来看，累计发现了几十个问题，除了一些纯体验问题，有效 bug 率在 80% 左右，参与的模块中有超过一半的模块有问题被提报。从发现问题的角度看这个实践是很有价值的。以下是具体操作过程中的一些思考和实践：

1）如何划分测试范围。是完全的散打，每个人随意挑选自己感兴趣的模块，还是事先分配好模块逐个地指定 owner？实际过程中我们选择了一个折中的做法，首先做跨小组的划分，比如 Android 和 iOS 测试小组互换，M 和 iPad 小组互换，这样可以在大的方面保证大家可以探索新的领域，另外如果有异地的团队也可以考虑按地域划分。在上面划分的基

础上，再请每个人来挑选想要去测的模块，先到先得。

2）时间安排的考虑。时间安排上也结合具体的项目情况，建议集中在 2~3 天的时间，这样效率比较高，也便于协调。

3）新发现问题的处理流程。交叉测试过程中发现的问题和研发阶段发现的问题一样记入 bug 管理系统，但最好首先分配给该模块原来的测试负责人。收到问题的测试人员需要像开发接到 bug 一样，及时地去重现和验证，如果确实是 bug，再转给对应的开发。这里要注意的是模块原测试负责人需要及时给出是否是问题，如何处理等反馈意见。这样不仅体现了跟进问题，而且也表达对别人贡献的尊重。

4）不要求全，鼓励探索。这个交叉测试的做法和正常的版本测试不一样，不要求系统和全面，不用编写测试用例。而且因为每个参与的人的时间分配不同，每个人都很难那么全面地参与。对于交叉测试，需要鼓励的是更探索性的方式，不用被测试用例束缚。

5）鼓励一定的竞争。交叉测试对于被发现问题模块的原负责人，也是一个鞭策。我们对此还专门设立了一个激励奖——“乐于助人奖”，发给交叉测试过程中发现 bug 最多的人，帮助发现别人模块的问题，其实就是在帮助别人。

交叉测试进行的阶段，可能是 App 已经全量发布了，或者还在少量灰度阶段中。如果在交叉测试中发现了严重问题，需要根据所处的阶段评估在哪一个版本修复？如果还在灰度阶段中可以替换灰度版本，如果已经发布了可以讨论是否出修补版本，或者将问题在下一个版本修复。

8.2　线上监控

这个主题其实非常的广泛，且有很多深入的内容，就我了解的几家大型互联网公司在上面投入了非常大的人力和物力，展开之后一定可以作为一本完整的书。这里限于篇幅，我们将讨论集中在两部分：1）常见的监控问题和做法；2）适合测试团队开展的监控方法。

监控的需求主要来自于以下几个方面：

1）应急问题响应的需要。因为有了大量的外部用户，所以研发团队希望非常清楚系统的运行情况。当出现问题和故障的时候，监控系统能第一时间告警，通知对应的人。这对于应急响应非常关键，因为等收到用户的投诉反馈，时间已经比较久了，而且影响面会非常大。

2）产品和业务运营的需要。监控系统除了对研发是重要的参考，实际项目中很多运营方面的数据也是通过监控系统给出的。

3）对研发质量提升的重要参考。监控的维度非常多，有很多方面可以帮助研发人员了

解系统的运行情况，进而可以不断地进行调整和调优，不断完善系统的性能、稳定性和可用性。

8.2.1 监控类型介绍

各个研发组织使用的监控平台、监控方法差异都非常大，但是可以从几个维度来进行划分。

8.2.1.1 面向用户的端到端的监控

这一类监控方法的特点是不需要在产品代码中增加监控的相关代码，没有侵入性，同时角度比较贴近用户，出现的问题可以比较容易对应到哪些功能失效。

这类监控方法有几种常见做法：

- 接口监控。接口是后台服务的俗称，无论是 Web 网站或者 App，都非常依赖于接口提供的服务，所以这个监控非常有必要。后面也会对接口监控的做法进行比较详细的介绍。
- 基于 UI 自动化的监控。模拟用户从 UI 的操作来执行监控。这个方面更贴近用户使用，但是因为 UI 自动化的执行效率原因，实际中应用并不多。

以上是从技术实现上来看，另外对于用户角度监控还有一些其他维度。我们再来看看监控平台的来源，也主要有以下两种类型：

- 内部开发的接口监控平台。稍后我们会结合实际的例子来介绍。
- 采用第三方的标准化监控服务，免费的或者付费的。业界也有多家公司提供相关的第三方监控服务。

稍大一些的研发组织多半会有内部开发的监控平台，当然，也可以采用第三方的作为补充。对于一些比较标准化的监控，借助第三方效率更高。

关于监控的地域分布，因为外部真实的用户可能来自不同的地域，使用不同的运营商网络，会带来实际访问体验上的差异，涉及网络质量、互联互通问题，以及多 IDC 和 CDN 的分布问题。所以监控方式也会遇到同样的选择，例如：

- 监控的请求发起集中在单台机器或者一个机房。
- 监控情况分布在不同地域、不同网络。

通常情况下，内部开发的系统较少大规模投入做非常广泛的监控点分布，所以这也是一些第三方监控服务提供商的一个价值点。多监控点的做法一般分两种：自建的监控点，或者借助合作的真实用户。

图 8-1 所示是某第三方监控平台提供的按地域划分的监控结果。

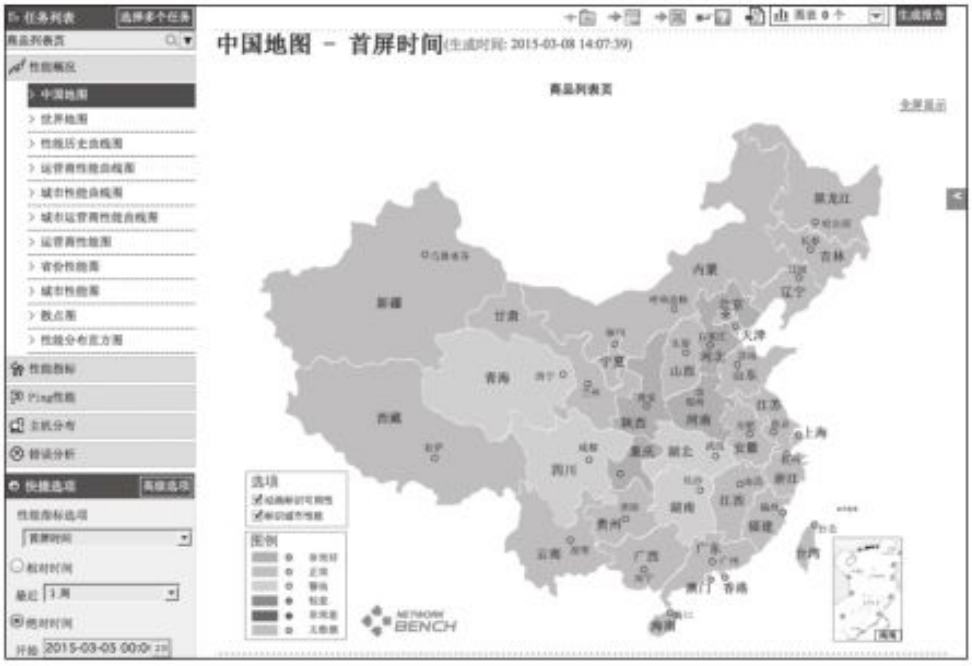


图 8-1　按地域划分的监控结果

8.2.1.2　产品中的埋点监控

上面提到的从用户角度的监控能发现很多性能和可用性的问题，但是也有一定的局限性，最主要的问题如下：

- 能知道哪个功能或者接口出问题了，但是不能进一步给出细节信息以帮助快速定位。
- 用户角度的监控本质上是采样的方式，是在巨大真实用户流量的基础上混入了少量的测试样本，严格来说并不能反映普遍的情况。也无法给出比较精确的全局统计信息。
- 无法提供一些内部模块间调用的信息。对于大型的系统，通常有非常多的子系统，而且通常也是多个团队协作开发和维护的。

基于这样的考虑，在实际项目中，通常会需要更细粒度的监控，而实现这种监控最直接的办法就是在正式的产品代码中添加监控相关的埋点代码。对于大型系统，由于涉及的模块和开发人员比较多，为了监控的标准化和高效，可能需要开发监控相关的库或者 SDK。也是由于这个原因，这种监控方式一般是开发团队来主导。

进一步，按照埋点的位置，可以细分为几类：后台埋点、网页埋点、App 埋点，下面分别讨论。

1. 后台埋点

前面讨论的基于用户角度的端到端的监控是从客户端（浏览器或者 App）可见的最外层接口来发起监控。这里讨论的方式是在后台接口内部插入一些监控代码来记录实际访问中的信息，俗称埋点。因为实际的大型项目中，通过一个最外层的服务，需要依赖很多层次的内部服务。比如电商的下订单服务，可能需要依赖以下信息：

- 用户信息服务，检查用户账号是否正常，信息是否正确。
- 地址服务，检查收货地址是否在可配送范围。
- 商品服务，检查商品信息。
- 库存服务，检查是否有足够的库存可供下单。
- 价格服务，当前价格是否正确和正常。
- 促销信息服务，检查订单使用的优惠券等促销方式是否满足对应的促销规则。
- 配送时效服务，结合以上信息，当前订单在什么时间可以履约配送。

以上只是列举了部分，实际项目还要复杂得多，所有这些接口，任何一个出问题都可能导致提交订单失败。这里的后台埋点监控就是要记录实际调用中，每一个被调用接口的表现情况，及时将结果上报到监控平台，反映出真实的情况。严格来说，每一个调用其他接口的服务都应该记录调用结果，并反馈上报。这样就可以第一时间发现问题，或者提供必要的统计信息。

统计信息通常包括：

- 本次请求的类型（记录核心参数）。
- 本次请求的结果（成功、失败以及返回码）。
- 本次请求的响应时间。

将这些信息汇总到一起后，可以进一步得出以下信息：

- 总请求次数，单位时间内的请求次数。
- 请求的总错误数，进而得到错误率。
- 响应时间统计，比如平均、最大、最小、90%、方差等维度。

结合时间维度，可以提供一些对比分析结果，比如：某个数据（比如登录量、访问量、下单量等？）相比昨天同一时段有没有明显波动，也可以和多天的历史平均值对比。这个指标可以帮助推测出一些异常情况，通过告警等方式引起关注，来进行进一步分析。

这方面的指标可以结合实际项目的特点和需求，衍生出很多。

从这里可以看出，对于整个监控系统，除了产品代码中的埋点之外，通常需要一个监控的后台，用于接收埋点上报的数据，进行必要的信息汇总和分析，提供相应的访问界面，以及通知和告警的功能，比如邮件、短信和微信等方式。

从这个角度，埋点监控本身已经成为系统的一个必要的功能，只是这个功能的需求方是研发自身。通常在这个功能的设计中，为了不因为监控系统自身的问题，比如获取信息失败，上报时间过长导致进程或线程被堵塞，一般都是通过单独的进程或者线程来进行异步的上报，包括实时的上报或者累积部分数据后集中上报。

2. 网页埋点

对于以 Web 形式提供服务的产品，可以通过在 Web 页面中埋点来监控用户访问的情况。实现方式可以在 html 中访问对应的 URL，或者调用相关的 JS，也可以用独立的 JS 方式来实现。基本的原理就是通过 URL 请求将数据反馈给对应的监控后台。可以采用 GA（Google Analytics）这样的通用第三方平台，国内有对应的百度统计、腾讯分析等平台，有些大的研发组织也可能会有自己研发的数据收集和分析平台。图 8-2 所示是百度分析的埋点代码。相比之前的用户端监控方式，这里的数据量要大很多，可以收集全量的埋点，也可以按百分比记录部分数据（如果数据量过大的话）。

图 8-2　网站统计埋点代码示例

从图 8-3 可以看出，这里提供的都是一些比较常见的标准化指标，如果需要定制比较细致的维度，可能需要在页面中埋入更复杂的代码，甚至自行开发对应的监控后台，这里不详细介绍了。

3. App 埋点

和前面网页埋点类似，App 埋点是将相关的监控代码放入 App 的代码里面，和 App 一起打包发布，当用户下载并安装了 App，打开使用的时候，相关的监控代码也会被执行，

记录相关的信息并通过接口上报到监控后台。

图 8-3 百度分析的报告示例

以下从应用角度以一个简单的例子来讲解。以下代码是在 Eclipse 开发环境中的 Android 应用中使用友盟 SDK 来开发的，详细使用信息可以参考其网站的说明和示例 http://www.umeng.com/，其他平台也是类似的做法。

首先，需要在友盟网站注册相关的开发者账号，获取 AppKey，并下载对应的 SDK。

接下来在 Eclipse 开发环境中，在 AndroidManifest.xml 中添加权限和一些配置，主要是 AppKey，这个是在注册好账号之后添加应用时获取的，Channel ID 是自己来定义。

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name= "android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name= "android.permission.READ_PHONE_STATE" />

……
<meta-data  android:value= "535f487******"  android:name="UMENG_APPKEY"></meta-
data >
<meta-data android:value="Test001" android:name="UMENG_CHANNEL" />
</application >
```

然后在 activity 里面添加对应的代码，MobclickAgent class。

```
import com.umeng.analytics.MobclickAgent;

protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mContext = this;
        MobclickAgent.updateOnlineConfig(this);
}

protected void onResume() {
```

```
        super.onResume();
        Log.e(TAG, "onResume");
        MobclickAgent.onResume(mContext);
}

protected void onPause() {
    super.onPause();
    Log.e(TAG, "onPause");
    MobclickAgent.onPause(mContext);
}
```

环境配置方面，需要把 umeng 的 jar 包加到工程里面，如图 8-4 所示。

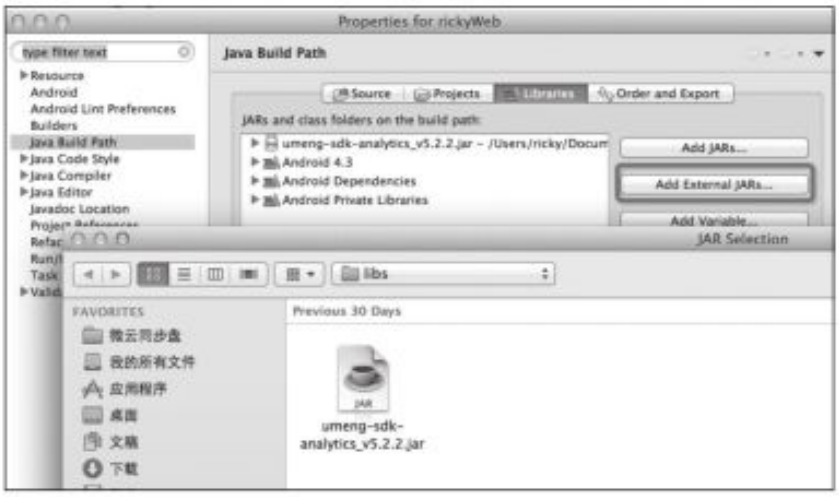


图 8-4　添加友盟 SDK 到编译路径

接下来，将 App 编译打包并在手机上执行起来，几分钟之后就可以看到数据了，如图 8-5 所示。

图 8-5　友盟统计分析的启动次数结果

这里因为是通过 onResume() 统计的，所以有一些细节需要注意，看启动次数是否需要调整，如图 8-6 所示。

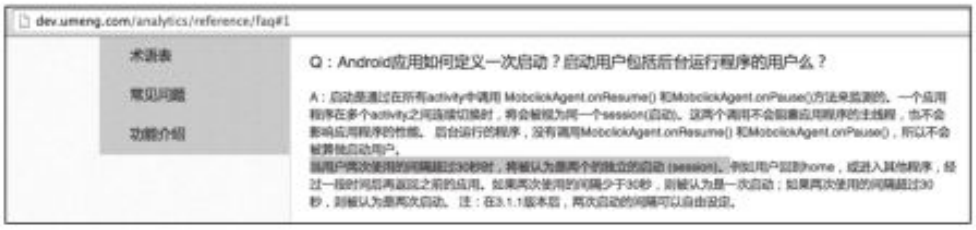


图 8-6　启动次数的计算方式细节

可以看到渠道的结果，如图 8-7 所示，就是之前在 AndroidManifest.xml 里面配置的。

图 8-7　新增用户的分析结果

设备的一些信息目前不能查看当天的结果，需要第二天来看，如图 8-8 所示。

图 8-8　新增用户的设备类型结果

另外也试验了自定义的 Event。我的 demo App 里面有两段访问 HTTP 接口的地方，分别通过 Apache HttpClient 和 Android HttpClient，想统计调用次数以及响应时间。

在调用前后埋上对应的代码：

```
MobclickAgent.onEventBegin(this, "Android_http_visit");
MobclickAgent.onEventEnd(this, "Android_http_visit");
```

并在平台上注册对应的事件 ID 和名称，如图 8-9 所示。

图 8-9　编辑自定义事件

接下来就可以看到对应事件的访问次数信息，如图 8-10 所示。

图 8-10　基于事件的统计结果

以上是一个简单的 App 端埋点或者监控信息的例子，实际项目中的代码要复杂得多，但是原理上是一样的。

大的研发组织，出于需求的定制和数据保密等方面的考虑，可能会独立开发自己的监控平台，以及对应的 App 端的 SDK。主要提供的数据包括以下三个方面：

- 运营层面的数据，比如 PV、UV，以及用户的访问路径、功能细节的访问情况等。
- 性能数据，比如某些功能的打开时间，或者接口的访问响应时间。
- 异常数据，将 App 端可能遇到的各种异常上报到后台，这样可以知道用户在使用 App 的过程中遇到的各种问题，帮助来进行排查和定位，以及后续的修复。前面提到的 crash 数据就是通过这种方式实现的。

如果是自行开发 SDK，需要注意以下几点：

- 在当前移动网络的环境下，用户通常对移动设备的流量使用是比较敏感的，所以需要控制监控上报带来的额外流量消耗。常用的做法是只上报必要的信息，并进行数据层面的压缩，重复的事件可以在 App 端聚合后再上报。
- 不应该干扰产品本身的功能和行为，不能因为 SDK 本身的逻辑影响到产品的稳定性。
- 资源使用方面的考虑。在某些极端的情况下，可能会产生大量的监控事件，比如用户频繁使用或者遇到异常。在设计上报 SDK 的时候，一般将记录和上报的过程异步来实现，先将数据整理后写到移动设备的文件系统，这样可避免数据累积带来大量的内存消耗，同时也避免网络状况带来的等待。除此之外，还会设定一些限制，包括参数的上限、内存的使用量、本地监控数据文件的大小等，在极端情况下，宁愿丢失部分数据，也不能影响 App 的运行或者移动设备的系统环境。

8.2.1.3 基础运维监控

以上提到的两个维度的监控都偏向产品的功能和应用层面，还有一类监控是面向基础设施和服务器的通用资源方面，我们称之为基础运维监控。这一类监控指标，通常和具体的业务并不强相关，面向的主要是一些通用的资源指标，比如：

- 机器负载，包括：CPU 使用情况和 Load、内存使用情况、磁盘 IO。
- 网络方面的情况，包括网络可用性、连接数、网络进出带宽等指标。
- 部分基础设备情况，比如路由器、交换机、负载均衡和存储等。
- 一些标准的服务器软件的情况，比如 Nginx、Tomcat、DB Server 等。

这些硬件和软件服务器设施是 IT 系统的基石，它们的稳定性和性能直接影响到上层的应用服务。所以需要对其进行密切的监控。在研发组织自己搭建的服务器环境中，通常由自己的运维人员来负责，对于购买的第三方云服务平台，通常由平台提供相应的监控和告警功能。

另外，这方面也有一些开源的第三方服务平台，可以在企业内部自行部署，比如常用

的 Ganglia 和 Nagios 平台。一些主要的基于操作系统或者常见服务器软件的指标都是相对比较标准化的，针对这些可以直接通过这样的平台获取。如果有一些内部的服务器软件或者自定义的指标也可以通过一些二次开发来补充。

8.2.2　接口自动化监控

以上介绍了实际项目中经常采用的不同维度的监控，可以看出有非常多的维度，对于稳定性和性能要求比较高的互联网服务，通常要求有比较全面和立体的监控系统。这些监控的开发和实施通常需要 IT、运维、开发和测试等多个团队共同完成。由于本书是基于测试团队的角度，所以接下来我们详细讨论比较适合测试团队来开展的基于接口自动化的监控。

对于接口监控不了解的人可能会有一个疑惑：为什么测试阶段通过并且发布时也验证没有问题的功能会发在布后出问题？主要原因有下面这些，这也是监控中会发现的一些问题：

1）互联网产品有运营的性质，系统本身有很多的配置可以调整，另外还可能有各种运营活动配置出来的功能。这部分在功能开发阶段是无法预料的，有很大运营的空间，也是可能出问题的地方。

2）基础设施方面的问题，比如 IDC 机房、CDN、后台服务器、反向代理配置和缓存服务等方面的问题。

3）不断的功能发布，迭代开发引入的新问题。

4）一些隐藏的之前未被发现的缺陷，可能在一些特定条件才会触发，或者系统数据量累积到一定程度才会出现。

5）依赖的第三方系统的问题。这一类也比较常见，任何一个大型的系统都会有很多的第三方依赖，公司内部的或者外部的服务，这些服务出问题我们可能并不会第一时间得到通知，如果有了针对自身服务的监控系统就可以快速地发现问题，并帮助定位。

考虑到开发的成本和工作的复用性，如果团队已经有了一套基于接口的自动化系统，那么在对应的接口监控方面就可以很大程度复用之前的工作成果。相关的接口自动化方面的讨论可以参照前面章节的讨论，下面的实例也是基于前面提到的以 JMeter 工具为基础的接口自动化。这里侧重介绍将其转化为监控需要注意的问题和一些实践的做法。

图 8-11 所示是监控系统的一个整体架构图。左下角部分，测试人员将本地调试好的基于 JMeter 的接口监控用例提交到 SVN（也可以是其他配置管理工具）服务器上。监控系统会定时到 SVN 拉取更新的用例文件，然后启动 JMeter，考虑到 JMeter UI 的开销可以采用命令行的方式运行。用例在执行的时候访问对应的接口，基于预先定义的数据，并执行相应的断言判断结果是否合法。待单次执行后，监控程序会来解析 JMeter 生成的结果文件，然后进行预定的解析，将解析好的数据汇总到数据库中。异步数据分析模块会定时来执行，

看异常数据是否达到告警的阈值，如果达到，会根据设定的告警方式和收件人，发送告警。另外，还有用于数据汇总的日报模块和提供监控数据详情访问的页面模块。

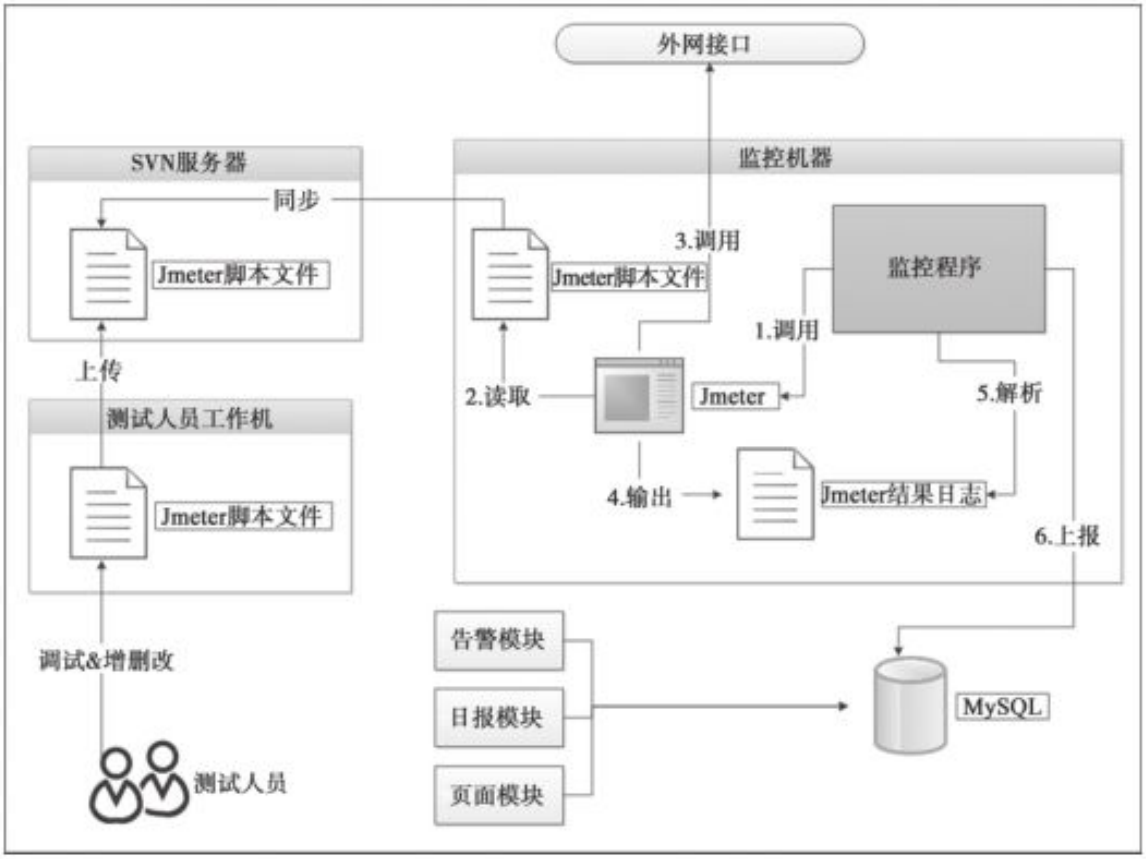


图 8-11 监控平台的整体架构图

上面介绍了整体架构，实际工作过程中还有非常多需要注意的方面。下面分别介绍。

1. 需要合理设定监控的范围

自动化测试因为目标是替代手工测试，所以用例尽量覆盖手工测试需要覆盖的各种情况，在资源充足的情况下尽量全面。但是对于监控，这样的思路不一定合适，而更应该少而精。原因主要是两点：

- 监控的目标不是为了发现功能性的问题，而且线上运行过程中的比较系统性的问题，所以监控的粒度可以比自动化粗一些。
- 每一个监控用例有大量的后续维护和告警处理的代价，过多的用例在实际应用中反而得不偿失。

以电商的商品搜索功能为例，功能测试可能需要覆盖各种非常细致的情况，比如中文、

英文、特殊字符、带有空格的多个关键词、热门和冷门的词语，以及各种组合条件，比如是否有货、付款方式、品牌、价格区间、屏幕尺寸（针对手机和电视等）等非常多的维度。如果我们针对搜索的接口监控覆盖这么多的维度将是非常繁琐的。

覆盖范围还有一个差异点是，监控通常不会去覆盖各种异常情况、边界和极端情况，这些不是持续不断运行的监控想要发现的主要问题。

2. 技术实现方面的考虑

1）监控请求和参数的动态化。线上真实的环境和测试环境不同，不是一个完全研发人员可控的环境，为了监控覆盖的更全面更准确，可能需要对部分请求和参数进行动态化。比如下面的例子：

- 对于一个电商的 App，首页有哪些运营活动通常是不确定的，运营人员可以动态的增加和调整。如果监控想要覆盖这部分活动的可用性和性能，那么就需要在用例中进行动态化处理，通常的做法是从前面的接口中获取有哪些活动，然后把这些活动列表提取出来，在后续通过循环的方式逐个访问遍历。这个在内部开发的自动化框架或者基于 JMeter 的平台中都可以比较方便的实现。其他领域的 App 也可能有类似的动态化的需求，处理的方式也是类似。
- 以下单为例，使用的商品可能因为无库存的原因无法购买，那么下单的监控就会失败。实际中的做法可以搜索某一大类商品，然后脚本从返回的结果里面挑选出有库存的商品为作为后续请求的参数。

2）必要的白名单机制。因为监控用例是在线上 7×24 小时不间断的执行，而且往往使用的账号数量有限，所以很有可能因为频繁的操作触发系统的安全防刷机制。针对这种情况，最好的办法是和对应的开发人员沟通，并获得审批，将测试账号加入系统的白名单。

3）定制的断言。对实际产品中的页面和接口响应有一些经验的人通常了解，HTTP 协议层面的返回码 200 并不能代表功能层面的正确性，很多错误页面的 HTML 返回时也是 200。另外，对于很多后台接口，然后在 HTTP 返回之外的内容里面指定一套自定义的返回码，比如 code=0 表示成功，其他表示各种错误情况。还有一些更复杂的情况，比如 code=0，应该是接口返回成功，但是实际的数据列表为空，而基于当前的请求返回的结果不应该为空，可能是某些内部服务出错导致的。 针对以上各种情况，需要设定合适的接口断言，以保证监控结果的可信度，避免误报和漏报。

4）需要设置超时机制。在后台服务或者网络出问题的情况下，监控的请求可能很长时间得不到响应，比如一分钟，这样可能会给监控系统本身的运行带来影响。这样因为某些接口的异常可能导致很多接口的执行间隔大大拉长，导致问题暴露的时间也变长。针对这样的情况，可以设定单个接口请求的超时时间，比如 10s，超过这一时间放弃请求并记录错

误，因为通常情况过长的响应时间本身也是问题。

5）用例的可扩展性。监控系统和很多系统一样，也是不断完善的，一开始可能只覆盖很少的功能和场景，所以平台的稳定性和价值得到认可，可能会不断增加新的用例。另外，在一个较大的团队中，基于功能模块的测试分工，可能需要很多的测试人员协助，各自完成自己负责模块的用例。为了更加系统化的开展监控，需要在监控平台的设计中考虑用例的可扩展性，以及对于用例的增加和修改的流程也必须比较高效。为了更好的配合，也可以制定监控用例的规范。

6）监控平台的大数据量支持。随着监控用例数量的不断增加和持续运行，监控系统的 DB 中累积的数据量会越来越大，所以也需要考虑系统自身的容量和稳定性。除了对应服务器本身硬件配置的保证外，也可以通过脚本定期清理历史数据，或者剔除部分正常数据而重点记录异常数据，以及对同类型错误的详情做聚合等方式减少数据量。

3. 业务统计数据的过滤

现在任何正式的商用系统，都会有对应的数据统计和分析系统，有些称为 BI（Business Intelligence）系统。如果监控的执行数量比较多，特别是一些监控的操作引起的数据变化涉及系统的核心指标，比如访问量、交易量等，那么需要和数据系统的开发人员合作，将监控产生的数据过滤掉，以免对数据的真实性造成影响。

4. 需要提供一套实时的告警机制

自动化测试通常只需要执行完之后有对应的结果就可以了，相关的测试人员会去查看，而监控产生的事件无法预测什么时候会发生，所以需要一些比较实时的告警机制。通常的方式有邮件，如图 8-12 所示，短信和 IM（比如微信）样例如图 8-13 所示。后两者的实时性更好，但是邮件可以提供更丰富的内容。

图 8-12　告警邮件的示例

图 8-13　告警微信的示例

实际中，还需要考虑的一个问题是告警的收敛机制，当一个问题持续存在的时候，如果每隔几分钟就发出大量的告警也可能会引起过多的打扰，另外监控系统也可能会有误判的情况，所以需要考虑一定的告警收敛机制。这个可以结合实际项目的特点来设定。

5. 需要提供定期的统计报表功能

接口监控的主要目标是第一时间发现问题并发出告警，但是从另一个角度，如果能定期统计一段时间内的接口可用性和性能的汇总情况，对于了解服务的整体情况和推动优化也是非常有帮助的。项目应用中，我们采取日报的机制来观察各个接口的情况，如图 8-14 所示。

对于异常率或者超时率超过一定阈值（比如 0.5%），可以将其标上底色以引起关注。

6. 辅助功能

告警内容的详情如图 8-15 所示。

接口的趋势图如图 8-16 所示。

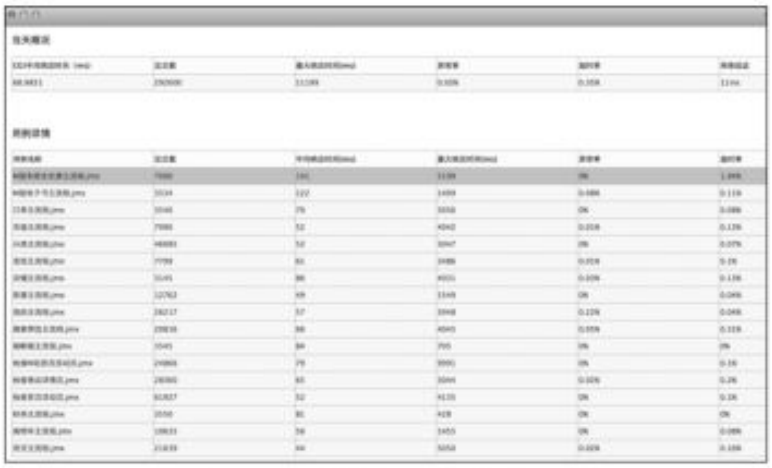


图 8-14　监控数据汇总日报的示例

图 8-15　告警发生时的接口请求和响应详情

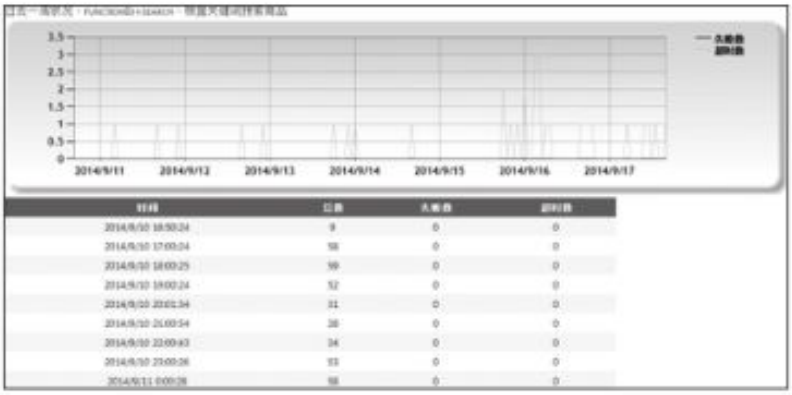


图 8-16　接口失败和超时趋势图

当一个告警报告出来的问题已经分配对应的人处理，或者有些问题短时间内无法处理，而又不想对所有告警的收件人造成过多的干扰，可以利用暂时屏蔽该告警的功能屏蔽一段时间。

7. 问题的响应机制

监控系统和之前的功能测试、自动化测试都不同，它本质上也是一个线上持续运行的系统，也会不断产生结果，包括告警和日报等汇总数据。如果只是开发完了系统，并上线产生结果，那其实并没有产生实际性的有价值的结果。要让整个接口监控和告警体系运作良好，需要考虑以下问题：

1）明确各个系统和模块的责任人，对于实时和统计数据中发现的问题及时跟进处理

2）应急响应机制，当收到告警的时候，谁来跟进处理，大致的流程是怎样的。这方面实际中也需要一定的实践和磨合。

3）问题的上升和告知，对于一些比较严重的突发问题，需要快速地告知相关的负责人，也可能需要上升出来让管理层知晓，以便及时协调和应对。部分组织也针对紧急问题成立了专门的应急小组。

8.3　外部用户问题反馈的收集和跟进

当产品发布出去，有一定的用户量之后，就可能会收到一些用户对问题的反馈。当然，反馈的问题可能不只是和软件系统相关，也可能和提供的服务本身等业务方面相关。不过因为我们的产品最终都是为用户提供服务，从用户角度看到的质量更直接和业务的发展相关。

从软件系统研发团队的角度，我们也非常希望知道用户对于服务质量的反馈，可以帮助我们及时发现问题，并不断优化。通常的反馈渠道主要有：

1）用户直接联系客服，通过电话或者邮件的方式。客服人员会解答用户的部分问题，一些系统相关的问题可能会被记录整理，进而反馈给研发团队。

2）很多 App 都带有用户反馈的功能。在很多 App 里面都可以找到对应的用户反馈功能，如图 8-17 所示，用户可以直接在里面填写问题信息然后提交，通常 App 也会顺带把一些基础信息上报，比如用户的手机型号、OS 版本和 App 的版本等，以帮助定位问题。

3）内测 / 粉丝用户群，或者朋友间的反馈。

4）用户在论坛和微博等公开渠道的反馈。

针对用户反馈的这些问题，需要一些系统化的跟进。对于较大的研发组织，常见的做法是安排专人来收集和跟进。以下是一些外部问题跟进的实践。

图 8-17 App 中的意见反馈功能

1. 定期的问题汇总和邮件周知

可以将用户反馈过来的零散的问题定期进行汇总，比如按周来统计。

图 8-18 所示是一个外部用户反馈问题的汇总周报，从中可以看出不同问题的数量分布情况。便于我们了解问题的影响程度，进而集中精力先解决反馈比较多或者影响比较大的。

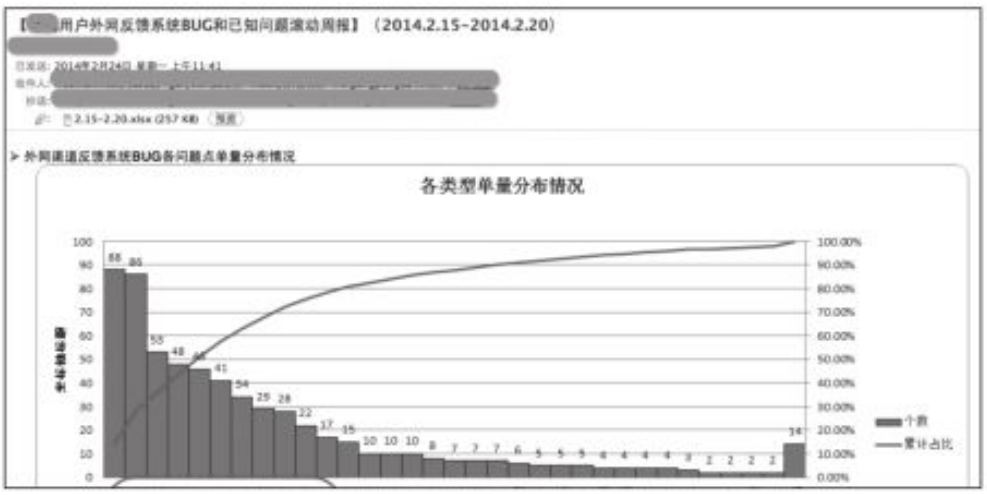


图 8-18 外部用户反馈问题走势

对应影响比较大、反馈数量比较多的问题要安排专门的人来及时跟进，并定期反馈结果。对于内部难以重现或者定位的，或者描述不清的问题，可以通过用户回访来获取更多信息。

2. 借助自动化的问题收集平台或者工具

这个做法主要针对论坛和微博等公开渠道，可以借助平台自身的功能，也可以借助第

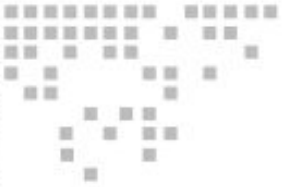
三方开发的分析工具，或者有部分研发实力较强的团队自行开发了相关的工具。这个自动化的方法可以更高效快速地收集问题，对于一些新版本爆发的问题也可以更快地感知。

3. 问题根源的跟进

针对定位为系统缺陷的外部问题，需要对应的研发人员给出问题根源的分析，以及可能的改进措施，可以用邮件发出。对于造成了较大影响的问题，需要召开专门的问题分析会，并通过会议纪要的形式发出后续的处理措施和提升计划。

8.4　本章小结

本章介绍了一些发布之后的质量管理工作，包括继续进行一些模块之间的交叉测试，发现一些之前没有发现的问题。另外，介绍了互联网产品的一些常见的监控维度，并重点介绍了适合测试团队开展的接口方面的自动化监控的实践做法。最后，讨论了关于外部用户问题反馈的收集和跟进的一些常见做法。可以依据产品和项目的特点来开展对应的实践。总的来说，这些发布之后的质量管理充当了一个闭环和反馈的功能，可以让我们看到有哪些问题从之前重重把关之中遗漏到了用户那里，以及可以反向推动开发，指出有哪些方面值得提升，也是对前面研发流程的重要促进。



Chapter 9 第 9 章

关于软件测试和测试团队

宿命论是那些缺乏意志力的弱者的借口。

—— 罗曼·罗兰

前面几章我们讨论了很多具体的测试技术，以及质量提升的方法流程，这些都是和我们的测试工作直接相关的部分。在这一章里面，我们试图跳出日常的工作，思考一些和测试相关的更普遍一点的问题。本章内容主要包含下面几个方面：

- 软件测试是否必需。通过分析测试这项活动本身存在的意义和价值，进而讨论哪些是需要专职的测试人员来做。
- 专职测试人员的价值。这里讨论专职测试人员所能提供的价值。
- 测试团队的管理和发展。IT 行业发展到现在的阶段，早已经告别了个人单打独斗的模式，做成任何一件有价值的事情，几乎都要依赖于团队的协作和努力。这一部分探讨测试团队在整个研发组织中的位置，测试团队内部不同的组织形式，以及测试人员的个人发展路径。

9.1 测试是否必需

如果要讨论测试的价值和意义，那么有一个更基本的问题不得不讨论，那就是测试存在的意义，或者说测试是必需的吗。更进一步的问题是：什么样的东西是必需的？这个问题看似简单，其实非常复杂。

比如对于一个人来说，谈到必需，大家直觉上会想到衣食住行这些基本的生活要素，说这些是必需的估计没有太大的争议。但是在这个社会上，对于一个普通人来说，必需的东西远不止这些，例如为了健康的成长，医疗和教育其实也是必需的。更进一步，也许友情、爱情也是必需的。这样的单子可以列出很长，随着每个人需求的层次不断提高，对于必需的要求也不断提高。

回到软件方面，其实也是一样的。如果我们开发一个系统，目标只是为了验证一个算法或者想法是否可行，那就好比只是最基本的生存需求。但是更进一步，如果我们希望这个软件可以有一些更实际的用途，那么就可能会需要处理一些实际的数据，产生一些有价值的结果，并保证正确性。如果我们希望很多人可以因为它而受益，那么可能需要解决很多人同时访问的问题。当它的重要性提升的时候，对它的要求就越来越高，包括各种情况下的正确性、异常情况下的稳定性、性能和使用体验等等方面。随着系统的发展，在基本的可运行的基础上，这些要求都会变成必需。

另外，对于质量的要求也和系统的类型有关。考虑下面几种不同类型的软件系统：

1）为课程设计开发的 demo 程序。

2）内部使用的 IT 系统。

3）面向公开用户的系统。

4）有海量用户的系统。

5）处理电子商务或者金融等关键业务的系统。

可以想象这些系统对于质量各个维度的要求都不同。这里隐含的是一个标准问题，即测试的必需性存在的一个前提就是对于品质的追求。

这里也折射出另一个问题，一个很现实的问题，谁来决定品质的标准，以及投入多少来达到这个标准？产品的定位和需求最终决定了后续的质量活动的投入度和深入程度。

任何对品质有要求的系统，哪怕只是对基本的可用性有要求的系统，都离不开测试活动。相对于理论的验证，测试是一个实践性质的活动，需要在一个实际可以运行的系统上验证各种维度的结果。理论验证可以提高效率，缩小测试的范围，但是实际的验证绝大多数情况下不可替代。大家可以看一下身边几乎所有的行业都存在各种正式和非正式的测试活动。就像我们的软件一样，有很简单的手工快速验证，也有用各种复杂的系统和设施来进行的辅助验证。风洞测试是一个很好的例子，它可以帮助验证飞行器或者一款高性能汽车的空气动力学方面的特性，而不是在量产之后把这个不确定性交给用户。

以上提到对高质量的追求提出了对于测试的需求，甚至深入测试的需求。但是需要注意的是，测试活动和测试作为一个独立的工种存在是有差别的。或者换个说法，必需的是测试这个事情本身，而不一定是测试作为一个独立的工种存在。从这个角度，也就能理解

为什么在很多的研发组织里面，有测试的需求但并没有专职的测试工程师存在。

为了进一步讨论这个问题，可以对比现实世界里面的测试工作和专职测试人员的依赖关系。看看下面的几个例子：

- 交管方面的速度测、酒精测试。因为工具成熟，使用简单，交通警察顺手做了这个测试工作，而不依赖于专职测试人员。
- 品酒师，古董鉴定专家。工具简单，但非常依赖于人的经验判断，需要非常专业和资深的人以及经验阅历的积累。
- 医学检查和化验。工具非常复杂，很依赖工具，但是也需要有一定专业知识的人来操作和判断，所以有专业的工具也要有专业的人。

类比来看，测试在不同的地方，是否作为一个独立的工种存在也是类似的考虑。目前来看，软件开发本身的标准化做得还不够，每一个系统基本都是独一无二的开发过程。当然，关于这一点也有不同的看法，一些人认为应该做到标准化但是现在还没有做到，另有一些人认为软件开发这样的高智力活动不应该考虑标准化。

总体来说，软件行业相对比较复杂。我们目前讨论的很多软件开发的形态是一种量身定做，宏观来看是一个比较初级的方式。很多企业，有自己的开发人员，几十、几百或者几千，做出的系统只是给自己的业务使用，而且累积下来的设计非常的复杂。这样使得通用的全自动的测试系统比较难实现，至少相对于传统行业而言，测试工具的通用性要差很多，类似 LoadRunner 这种算是比较通用的商业性能测试工具，其实也有非常大的二次开发和分析工作，而且对使用的人要求比较高（我们讨论的性能测试不只是录个脚本跑一下这个层次）。当然，对于一些比较标准化的产品，比如防火墙、路由器等，针对部分的性能指标有 RFC 3511 之类相对标准化做法和工具，但整体而言还是非常有限。

基于这样的现状，对于一个质量有一定要求的软件系统，我们还是需要测试，具体的情况有三种可能：

1）开发人员自己来做测试工作。

2）外包给别人来做测试工作。

3）有内部的专职测试人员。

实际上，很多组织里参与测试的人员是上面两种甚至三种的混合，因为混合的比例不同体现出一些差别。比如我们看到的几种模式如下：

- 开发做了比较完整的单元测试和基本的功能测试，使得测试人员极少。比如大家常说到的 Google 模式，他们的开发自测做得比较完善，测试人员的比例较低，以及部分使用外包。
- 开发自测较少，专职测试人员的比例比较大。有些研发团队的开发人员只做极少的

自测，大部分依赖专职的测试人员，这个其实比较常见，特别是在一些成熟度不是很高的研发组织中。

- 在上面的基础上大量使用外包，把很多基础的版本功能测试通过外包来完成，内部的测试人员更多来做自动化和其他测试技术方面的实践。

结合以上的几个维度，我们再来看测试是否必需，以及不同的投入考量，也许会更加清晰一些。

9.2　专职测试人员的价值

从以上的分析和实际中接触的研发组织内部人员的看法，大部分人对于测试活动本身的价值是认可的，存在争议的问题是：是否需要专职的测试人员？通常的观点包括：为什么不可以开发自测来保证？为什么不可以通过用户的小范围灰度测试来替代专职测试

关于第二个问题，在前面灰度相关的章节也做了一个基本的探讨，接下来我们看看专职测试人员能提供哪些价值。因为只有提供足够的价值才有存在的必要。

基于大量项目的实践来看，我们认为专职的软件测试有存在的价值，可以分为下面三个层次。

第一个层次：职位本身带来的价值

这个有点类似于工厂里的 QC，需要有专人来做检验的工作，这种价值和设立这个专门的职位有关。就像很多职业的分工，一旦设立了这个专门的职位，这个职位上的人就需要按照设定的要求去驱动某些事情。对测试而言就是产品在出去之前会被检验到，对项目经理而言就是会按照计划来驱动项目往前走。极端一点来讲是不需要通过这样的职位设定来驱动某些事情落实的，因为开发人员也可以自测，产品集成好了之后也可以从用户的角度来完整的测试，但是实际上如果没有这样的分工和专职的安排，很多事情会难以执行到位。独立的分工可以在某种程度保证对应的事情有专人来做。

从这个角度看，这有点像是通过分工来确保落实。而且因为设立了这样的专职工作，那么自然就有工作职责，需要对质量负责，而因为有这样的要求，测试人员被要求发现问题，提出质量提升的建议。一个是制度上的安排，一个是心理上的。反过来可以设想一下，如果在一个正式的商业产品中，没有测试人员或者类似的工种，很多时候对质量的要求容易流于形式，很容易被进度的压力妥协掉，而且测试是否充分本身就是个很模糊的概念，大家可能简单用一用觉得没有问题就发布出去了。

上面提到的其实是一个很基本的层次，相当于测试有而且做了而已，至于做得怎么样，那是另一回事。

第二个层次：做得更专业和深入

这个也容易理解，就好比饭店的主厨与一个只偶然在家里做饭的人相比，或者一个专业的赛车手与一位只把开车当出行方式的人相比。他们都可以做菜或者开车，但是实际上做得深入和专业程度完全不同。对于测试这个职业，也是一样，如果只是把功能都点到了，发现了 bug，那和普通用户其实没有太大的区别，也就没有专业的价值。其实，从这个角度看，本书的前面 8 章都是在谈论专业的测试人员有哪些专业的价值，主要是测试的深度，也就是效果，还有因为专业而带来的高效率。

如果能做到上面的两个层次，在目前情况下，已经是一个合格的测试人员了。但是如果进一步放到一个更大的范围来看，特别是从测试团队在整个研发系统，进而在整个公司的层面来看，做到以上层面也还是不够的。

第三个层次：提高整体产品的质量

前面的两个层次主要还是在找问题，着眼点还比较窄，仅从这个角度不能满足更高的要求。因为对于业务和产品而言，这样做有两个问题，一是事后才发现问题代价比较大，二是发布后才发现问题有时已经晚了，甚至没法修补。

其实还有更多质量提升方法，包含但不限于以下几点：

1）将发现 bug 变得更早。比如通过静态扫描、单元测试等技术手段，或者前面章节提到的各种研发质量提升的流程方法，以及持续集成等实践的开展，将更多质量问题提前暴露。

2）缺陷的预防。再往前走，在有缺陷的代码被写出来之前就发现问题。比如在需求阶段、设计阶段，甚至产品的规格制定的时候就发现问题。这类问题有很多，比如很多场景可能没有考虑到，有些可能和原来的客户或者产品的需求不一致，甚至有些地方不具有可测性。针对这种情况，研发团队可以及时的讨论和调整。这个时候的调整可能比产品出来之后发现几个 bug 更有价值，因为早期的错误可能到后面很难修复，或者修复的代价很大。

3）协助建立质量的文化。之所以说协助，是因为我觉得这个可能不只是靠测试人员就能做到，需要和开发人员以及产品的管理人员等一起来创建。质量不是测试出来的，而是写出来的，是设计出来的，是架构出来的，是规划出来的。整个研发流程的每一个环节的一些决策和实践都会深远的影响到质量，这个已经超出了测试人员最基本的工作要求范围，但却是能发挥更大价值的地方。

另外，从发展的角度来看，专职测试人员的出现其实也是软件行业的专业化细分带来的结果。

很多大的软件企业都是从创业的小公司成长起来的，包括我们所见到的所有大互联网企业。一开始，可能只有几个创始人，大家提出想法，自己动手，或者找几个初期的创

业伙伴来实现。这个时候，并没有细致的分工。随着业务和产品的逐渐发展壮大，慢慢有了专职的产品经理、运营人员、运维人员、设计师等角色，开发也逐步细分为后台开发、Web 前端和 App 开发等不同工种，测试其实也是这个专业细分中自然产生的。

其实，对于专职测试这个工作的某种偏见，或者叫误解，并不只是对于测试。在一个大的研发组织中，每个人大部分时间都专注于自己的工作内容，少有机会了解其他的合作方。以至于有很多对于其他工种专业性的不了解，在觉得测试就是点点鼠标的同时，也有人觉得产品经理就是写写需求文档，开发就是 coding，设计师是美工，运营就是配置下活动，运维就是配置服务器。这样的理解其实是非常的狭隘，因为每一个工作做到深入都有它非常专业的地方。

最终，专职的测试人员的价值和意义还是看他们创造了多少价值。

9.3　测试团队和发展

前面我们讨论了在不同情况下软件测试的必需性，以及专职测试人员的价值，接下来我们探讨关于测试团队的一些做法，以及人员的发展。

关于测试团队，不同组织里面的情况可能不同，因为在初创的企业里很多是没有测试人员的，而随着业务的发展和人员规模的扩大，逐渐有了测试人员。在整个研发组织人员较少的时候，测试可能还没有形成一个独立的团队，这种情况下一般是直接汇报给开发小组的负责人。这个时候的测试人数较少，主要是比较基础的黑盒功能测试。

等到整个研发体系发展壮大了之后，测试人员也慢慢多了起来，会形成一个单独的测试小组。进一步发展后，可能会出现针对每一个业务或产品线有对应的测试小组，进而就出现了测试团队的二级组织。这个时候随着测试团队规模的扩大，当然也是随着整个研发组织的壮大，以及业务方面提出的更多更高的质量要求，测试团队在客观上有了进一步提升的需求。另外，主观上因为较大规模的专职测试团队的出现，会不再满足于完成基本的功能测试工作，也有进一步提升的动力。一些测试的流程规范、用例设计、缺陷管理、质量数据分析、自动化的开展和工具平台等慢慢开始引入。接下来，可能到研发部门层面有一个完整的测试团队，进而可能是整个公司，或者事业部层面有一个完整的测试部门，或者中心。

就目前来看，出现了两种趋势，一种是把整个公司或者事业部的测试集中在一个大的测试部门，另一种是打散分到各个产品线。因为各个组织的情况和发展思路不同，很难简单地说哪一种好或者不好，从测试团队的角度来看各有利弊。集中在一个大的测试部门主要好处是：

- 因为资源的整合，可以减少各个团队的重复建设，集中来做一些平台建设，技术研究或者技术共享，有利于提升团队的技术深度。
- 从业务的角度，集中后测试可以横向地来看各个项目的质量情况，研发流程的过程执行和效率的情况。从整个组织的角度，对研发的质量和效率有促进的作用。
- 从测试人员个人发展的角度，因为整个测试组织有了更好的深度，个人发展的空间也会更大，无论技术还是管理方面。

将测试人员分到各个业务线的好处是：

- 和对应业务的产品和开发等团队在一个部门，可以减少跨部门协作的问题。
- 因为分到业务部门，可以针对业务的发展情况调配人员的编制，避免在大的测试部门的时候可能会面临的业务之间人力资源竞争的情况。
- 测试人员可以更加专注和贴近业务，持续深入地了解业务的需求，一些实践也可以做得更专深。

另外，介于这两者之间还有一种是矩阵式的管理。一方面，从组织架构上是归属于测试部门或者质量部，但是从日常工作上，是归属到具体业务线，也可以虚线管理，和对应的产品、开发团队密切配合，包括工作座位可能都在一起。

讨论完组织架构后，我们来看看测试团队内部的专业细分。

IT 行业发展几十年后，已经出现了大量非常细致的分工，有些分工甚至可以说是隔行如隔山。考虑对口的产品和技术形态，测试人员也有很大的差异了，比如测试通信设备的、测试 Web 前端的、测试后台服务的以及测试 App 的，其背景知识、工作流程等方面也有很大的差异。即使不考虑这方面，从专注的测试工作内容上看，也有进一步的细分。一般可以分为以下四个专业领域：

1）*业务测试*。也叫系统测试，这一部分的测试人员负责具体产品和业务的测试。当然，具体的工作内容并不局限在手工的功能测试，通常也包含自动化测试的开展，以及性能测试和各种专项技术测试的开展，总体的目标是为了所负责产品的质量提升。由于整体而言，测试还是为业务服务的，所以这一类的测试人员一般占测试团队的大多数。

2）*专项测试*。如果测试组织稍大，对测试的深度有更高的要求，而有些测试类型又需要比较长时间的积累，且技能有横向的共用性，那么就可能有专人来做，俗称专项测试，比如安全专项测试、性能专项测试等。

3）*测试开发*。当整个测试团队的规模比较大，需要很多共性的基础平台和工具建设，就可能会抽出部分人员专门来做这方面的事情，一般称为测试开发。其工作内容本身更接近软件开发人员，所做的主要工作也是开发系统，只不过是测试或者研发内部使用的系统。

4）质量管理。有些研发组织称之为 QA 或者 SQA，其专注在质量数据的收集、研发流程的管理和质量推动等事项上。在本书的第 7 章专门讨论到了相关的内容。根据组织的实际情况，有些组织将这个角色放在测试部门，有些组织可能放在研发管理等其他部门。

虽然有了上述工作方向的划分，但是也不应该将职责完全割裂开来。比如对业务测试和测试开发，有些测试团队甚至并不严格区分，如果业务测试人员的能力比较好，结合业务测试的需要抽出一部分时间来开发工具也是完全可行的。反之，如果测试开发人员完全脱离业务，可能做出来的测试工作或者平台并不能很好地符合业务的需求，所以有时候测试开发人员适当地参与产品功能测试也是很好的做法，可以了解实际项目中的测试流程、遇到的问题以及需要工具来帮助提升的地方，这样可能对于测试开发工作会很有帮助。

相信很多人在实际项目中，都会遇到测试资源不充足的问题。这确实是一个矛盾的问题，项目的进度都很紧，要快速找到大量合格的测试人员也很难。其实，关于这个问题，可以从另一个角度来看，是否所有的功能都需要经过测试，以及测试的投入力度。

实际项目中，测试资源永远都是不够的。通过前面讨论测试设计的章节，我们可以发现哪怕对于一个简单的功能，都可以设计出大量看起来很有必要的测试用例，总体来说执行成本是很高的。在一个比较成熟的研发组织中，应该敢于让部分项目，或者项目中的部分功能不经过测试，俗称免测。当然这个前提是开发团队进行必要的测试，保证一定的质量，整体的风险也比较可控。而专职的测试团队定位是少而精，主要精力放在重点的业务、重点的功能，或者重点功能的重要部分。

在整个项目流程中，特别是对于互联网这样快速迭代的项目，从人力和时间的消耗上来看，测试活动都是代价非常大的。所以过度依赖测试来保证产品的质量不仅代价很大，而且也可能因此影响产品的节奏。所以在实际项目中，要从研发环节各个流程来提升质量，将测试资源针对性地投入在重点项目上。

这里还有一个方面值得探讨，那就是测试投入的可伸缩性。有过测试经验的人比较容易理解，对于一个小的功能，比如电商系统中的商品详情页，根据不同的质量要求和测试粒度，一个测试人员可以测试一个小时，也可以测试一天，或者一周。当然这也取决于提交测试的版本的质量。所以在实际中，除了按照重点项目和功能来划分，也可以从测试粒度的不同来更有效地利用有限的测试资源。

接下来，我们探讨一下测试人员的发展。

目前来看，业界存在一个普遍的矛盾：一方面很多人觉得测试没有发展，另一方面，非常多的企业急需资深的专业测试人员。我们认识的大部分测试团队负责人都为招不到优秀的测试人员而苦恼。工作中，我们也见过一些测试人员转型为研发流程中的其他角色，

这里我们重点讨论继续在测试领域发展的测试人员的发展问题，换个角度，也可以从业界普遍希望招聘到什么样的测试人员这个角度来看。目前看来，比较资深的测试人员主要有三类：

1）有良好测试开发能力的人。通过测试技术和工具提升整个团队的测试深度和效率是一个持续追求的目标，所以有良好的测试开发能力的人一直都是很紧缺的。良好的代码功底不必说，而且需要有一定的测试思维，最好是有测试平台和工具的开发经验。很多人觉得能写代码就能做好这个，其实有差别，如果没有一定的测试思维和经验，做出来的东西可能会比较脱离实际，最后难以落地。

2）资深的业务测试人员，有丰富的业务测试经验、测试设计能力和质量推动能力很好的人。测试设计是一项很重要的能力，需要长期的训练和有意识的培养，而且大部分缺陷的发现也非常依赖于测试设计的全面性和深度。另外，还需要业务测试人员有全局观察和分析的能力，持续地推动质量而不只是发现 bug，所以也需要对研发流程非常了解，并且有很好的沟通协调和推动能力。

3）测试团队 Leader。一个好的测试团队 Leader 需要有良好的技术功底，因为带队人的技术能力和视野对团队的影响非常大，也会影响对人员和工具平台等方面的判断力，以及是否可以把团队的技术能力带到一个更高的高度。除了技术能力之外，还需要有良好的规划能力，能够系统性地考虑团队面临的问题、需要提升的方向和达成的节奏。另外，还需要有非常好的对于人的洞察力，以及沟通和协调能力。

以上列出的三个发展路线在目前来看是比较清晰的，当然，实际中并不局限于这三种。总体而言，专职的测试人员在整个研发组织中和其他角色一样有很大的发展空间。

9.4 本章小结

本章我们抛开各种测试技术、方法和流程，重点探讨了测试这项活动本身的意义，进而探讨了是否需要引入专职测试人员，以及对应的价值。关于测试团队方面，讨论了目前业界一些主要的组织结构和人员安排的做法，并进而探讨了测试人员的发展路线。这些方面不像工具和技术，可以比较明显地直接看到实际使用的效果，快速地判断出优劣，而更多取决于整个组织的状况，大家的观念，以及项目的类型和所处的情况。所以相对而言，这一部分其实是更加复杂，更需要团队负责人根据实际情况来选择和推动。

参考文献

[1] Rom Patton. 软件测试 [M]. 张小松，等译 .2 版 . 北京：机械工业出版社，2006.

[2] Paul C Jorgensen. 软件测试 [M]. 韩柯，等译 .2 版 . 北京：机械工业出版社，2003.

[3] Glenford J Myers，等 . 软件测试的艺术 [M]. 张晓明，等译 .3 版 . 北京：机械工业出版社，2012.

[4] Paul M Duvall，等 . 持续集成：软件质量改进和风险降低之道 [M]. 王海鹏，等译 . 北京：机械工业出版社，2008.

[5] Jez Humble，等 . 持续交付：发布可靠软件的系统方法 [M]. 乔梁，译 . 北京：人民邮电出版社，2011.

[6] Gerald M Weinberg. 完美软件：对软件测试的各种幻想 [M]. 宋锐，译 . 北京：电子工业出版社，2009.

[7] Alan Page, 等 . 微软的软件测试之道 [M]. 张爽，等译 . 北京：机械工业出版社，2009.

[8] Jams A Whittaker. 探索式软件测试 [M]. 方敏，等译 . 北京：清华大学出版社，2010.

[9] Michael.Sutton，模糊测试：强制发掘安全漏洞的利器 [M]. 段念，等译 . 北京：电子工业出版社，2013.

[10] Paco Hope，等 .Web 安全测试 [M]. 傅鑫，等译 . 北京：清华大学出版社，2010.

[11] 郭欣 . 构建高性能 Web 站点 [M]. 北京：电子工业出版社，2012.

[12] Steve Souders. 高性能网站建设指南 [M]. 刘彦博，译 . 北京：电子工业出版社，2008.

[13] Steve Senders. 高性能网站建设进阶指南：Web 开发者性能优化最佳实践 [M]. 口碑网前端团队，译 . 北京：电子工业出版社，2010.

[14] Gerald M Weinberg. 技术领导之路：全面解决问题的途径 [M]. 余晟，译 . 北京：电子工业出版社，2009.

[15] 安迪 · 格鲁夫 . 格鲁夫给经理人的第一课 [M]. 巫宗融，译 . 北京：中信出版社，2011.

[16] 彼得 · 德鲁克 . 卓有成效的管理者 [M]. 许是祥，译 . 北京：机械工业出版社，2009.

移动 App 测试实战

顶级互联网企业软件测试和质量提升最佳实践

邱　鹏　陈　吉　潘晓明　著

机械工业出版社
China Machine Press