我们都在使用敏捷开发，敏捷测试，维护着我们的项目，我们写着少量的test case，甚至不写一条case，敏捷宣言其中一条原则是工作的软件 “高于” 详尽的文档，详解文档包括各种计划书，总结报告，详尽测试用例等，我们将大量时间用在自动化测试以及手工探索性测试上面，而我们的用例则以BDD形式存在于代码之中，这样来帮助尽可能早的发现问题。

但最近我发现几个客户在质量问题，存在一些共性，这些基于黑盒测试的项目在测试过程中存在以下几个共同的问题：

（1）**大量的黑盒测试用例**，有的项目甚至用例数超过5w，测试工作大都是手工为主，受主观人为因素影响太大：每次版本发布，QA全凭个人经验来确定改动对系统影响范围，通常情况，要么测试范围定小了，造成漏测，要么测试范围过大，付出的代价过高，造成项目不能如期按时交付。

（2）**代码与测试没有数据可衡量**：没有单元测试，其他类型测试对代码覆盖程度，质量高低，没有数据能够衡量，例如我们说api测试覆盖率是100%，这个数据大多都是根据用例业务场景估算出的。QA只能增加更多的黑盒测试，而实际功能测试覆盖率随着时间和用例增多，便会触达覆盖率的天花板，更多的是重复的无效测试。

（3）**自动化测试无法发挥作用**：对于web/api或app 后端服务系统，测试人员对除手工测试外，我们将大量的时间与精力投放在api接口测试的实现上，随着项目的迭代，自动化用例积累越来越多，从几百到上千，这时候我们需要考虑测试稳定性，运行时长，大量重复测试场景与代码，整个测试ROI并不是随着用例数增多而上升，反而维护和排查问题成为QA日常工作的重担，疲于应付，没有精力将时间投入到更有用的探索性测试和分析工作中，进而造成bug频出，整个团队便对自动化失去信任，直至废弃，这也是很多传统行业无法规模化实施敏捷测试原因之一。

那么如何帮助团队树立信心，准确定位到变更影响的范围？精准测试。这个概念是最近几年逐渐兴起的话题。

先来谈谈什么是精准测试？

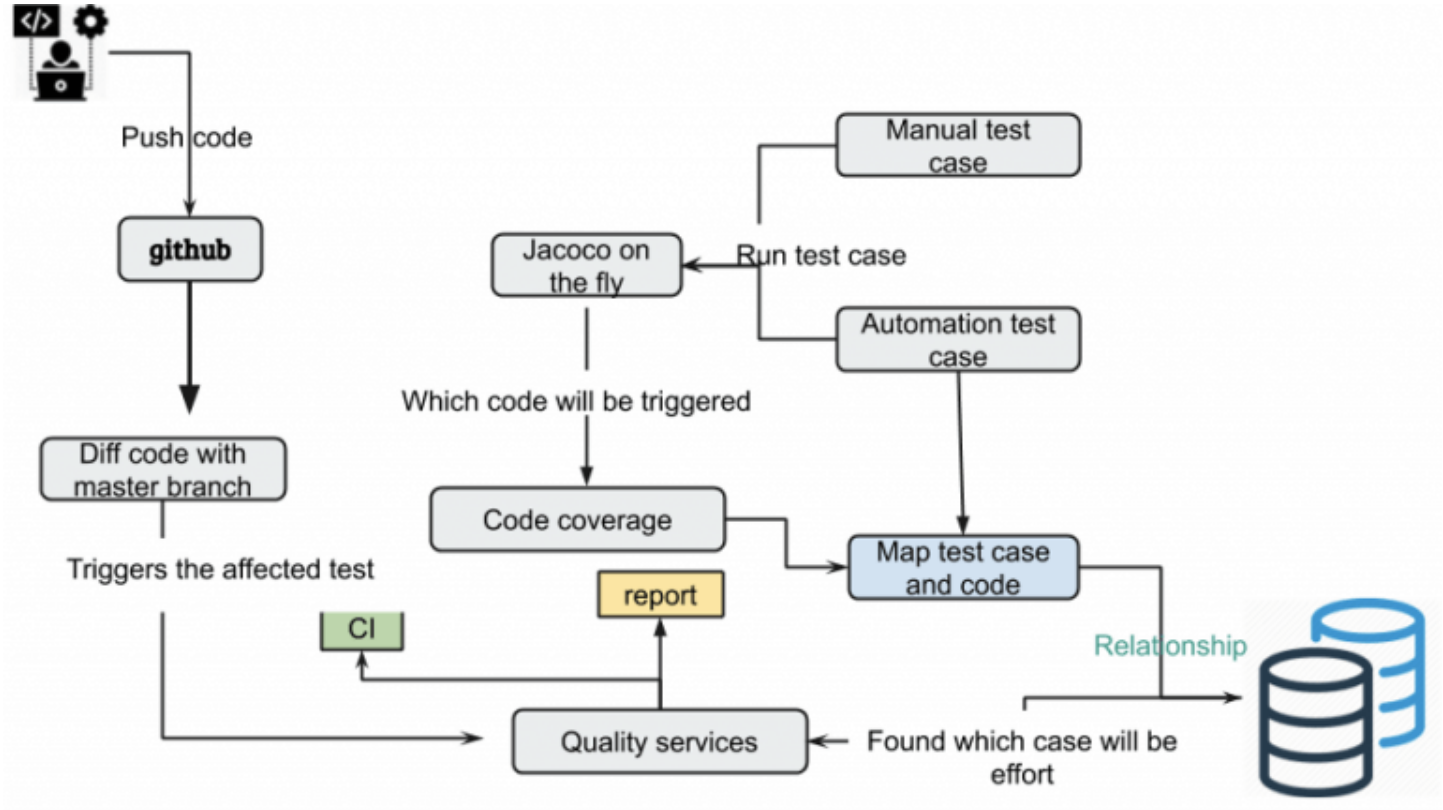
定义：利用技术手段对测试过程产生的数据进行采集存储，计算，汇总，可视化最终帮助团队提升软件测试的效率、并对项目整体质量进行改进和优化的这一系列操作。

通俗点讲：核心基于源代码变更分析，结合分析算法，确定影响范围，提升测试效率。

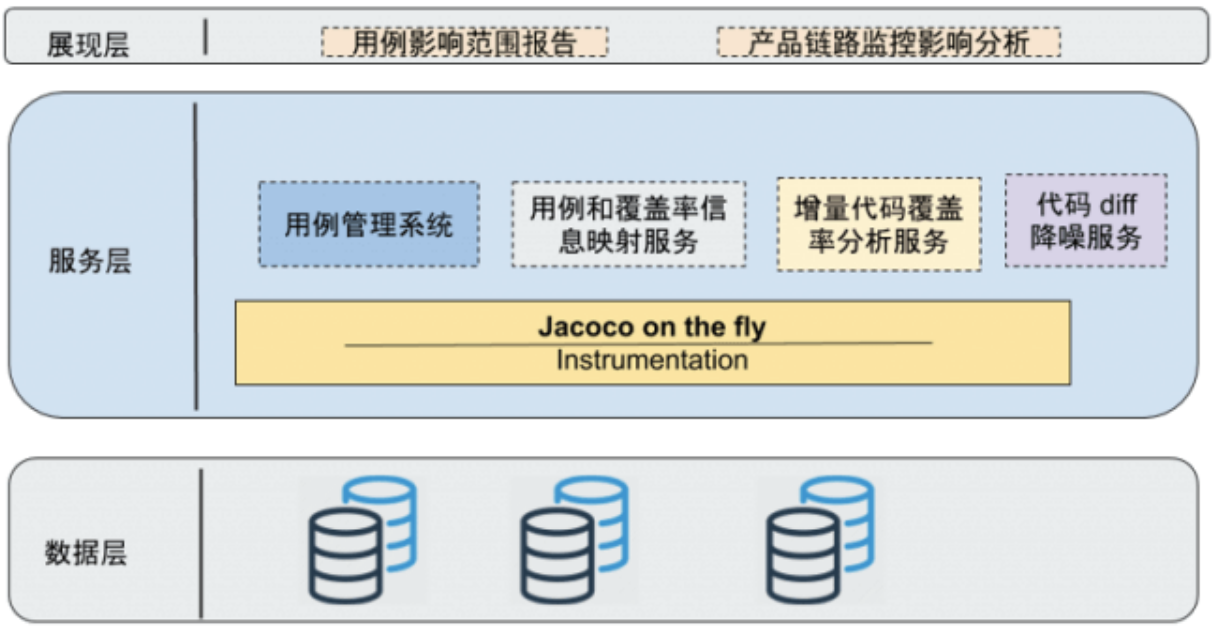
精准测试并没有改变传统的软件测试方法论，只不过是帮助我们将测试用例与程序代码之间的逻辑映射关系建立起来， 而这个过程则是通过算法和工具去采集测试过程执行的代码逻辑及测试数据，在测试过程加入采集过程，形成正向和逆向的追溯。

- 正向追溯，开发人员可以看到QA执行用例的代码细节，例如用例执行过程中，调用具体方法与实现类，方便进行缺陷的修复与定位。
- 逆向追溯，测试人员通过release前的增量代码快速确定测试用例的范围，极大减少回归测试的盲目性和工作量，提升ROI，达到测试覆盖率最大化。

精准测试原理



精准测试流程图



精准测试框架图

上图是我基于业界通用精准测试架构画出的架构图与流程图。

这套精准测试架构既可以用作手工测试，也可用在任意自动化测试上。
整个架构分为以下几部分：

1. 建立用例与代码覆盖率之间的映射关系
2. 影响面评估，分析识别增量与变更代码
3. 测试范围评估，用例筛选，链路分析

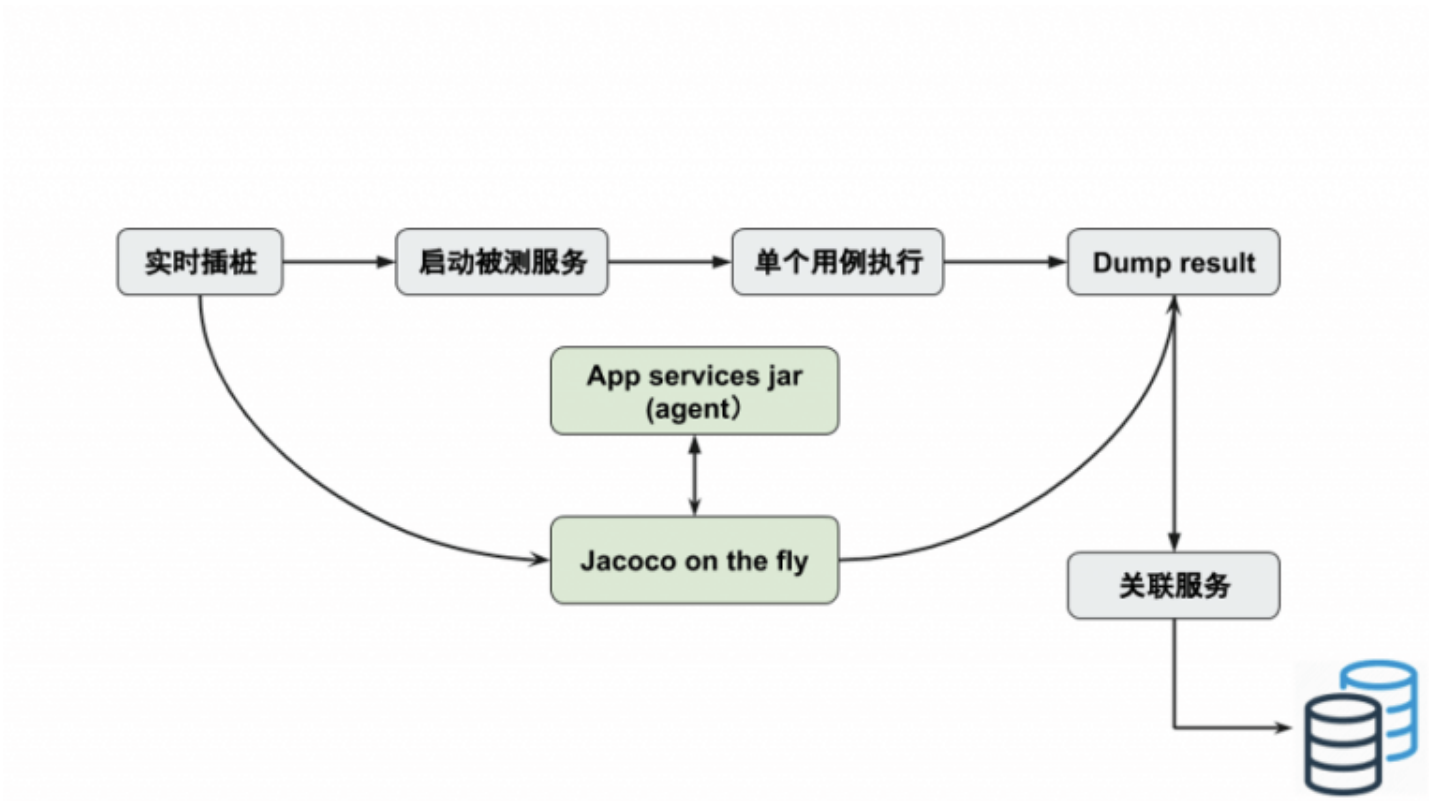
建立用例与代码覆盖率之间的映射关系

目前建立用例与代码之间的关系通过统计调用产生的覆盖率与路径进行关联。

在线模式

1. 功能测试关联方案

目前通用的解决方案是利用Jacocoon-the-fly模式基于 On-the-fly 方式无须入侵应用服务代码，启动脚本时，只需在 JVM 中通过 -javaagent 参数指定 jar 文件以启动 Instrumentation 的代理程序，该代理程序通过 Class Loader 装载一个 class 前判断是否需要注入 class 文件，再将统计代码插入 class ，测试覆盖率分析就可以在 JVM 执行测试的过程中完成。Jacoco提供了自己的Agent，完成插桩的同时，还提供了丰富的dump输出机制，如File,Tcp Server,Tcp Client。覆盖率信息可以通过文件或是Tcp的形式输出。通过外部服务在任意机器上通过api请求获取被测程序的覆盖率与执行路径。
对功能测试用例，我们可以通过执行单个用例，经过上述步骤后拿到该条用例影响代码的覆盖率与执行路径。在通过关联服务，将具体用例的ID与生成的覆盖率信息（类，方法，行等）建立映射关系，最后将关联数据存到数据库中保存。
基于Jacocoon-the-fly 模式获取单个用例覆盖率建立映射关系的流程如下：

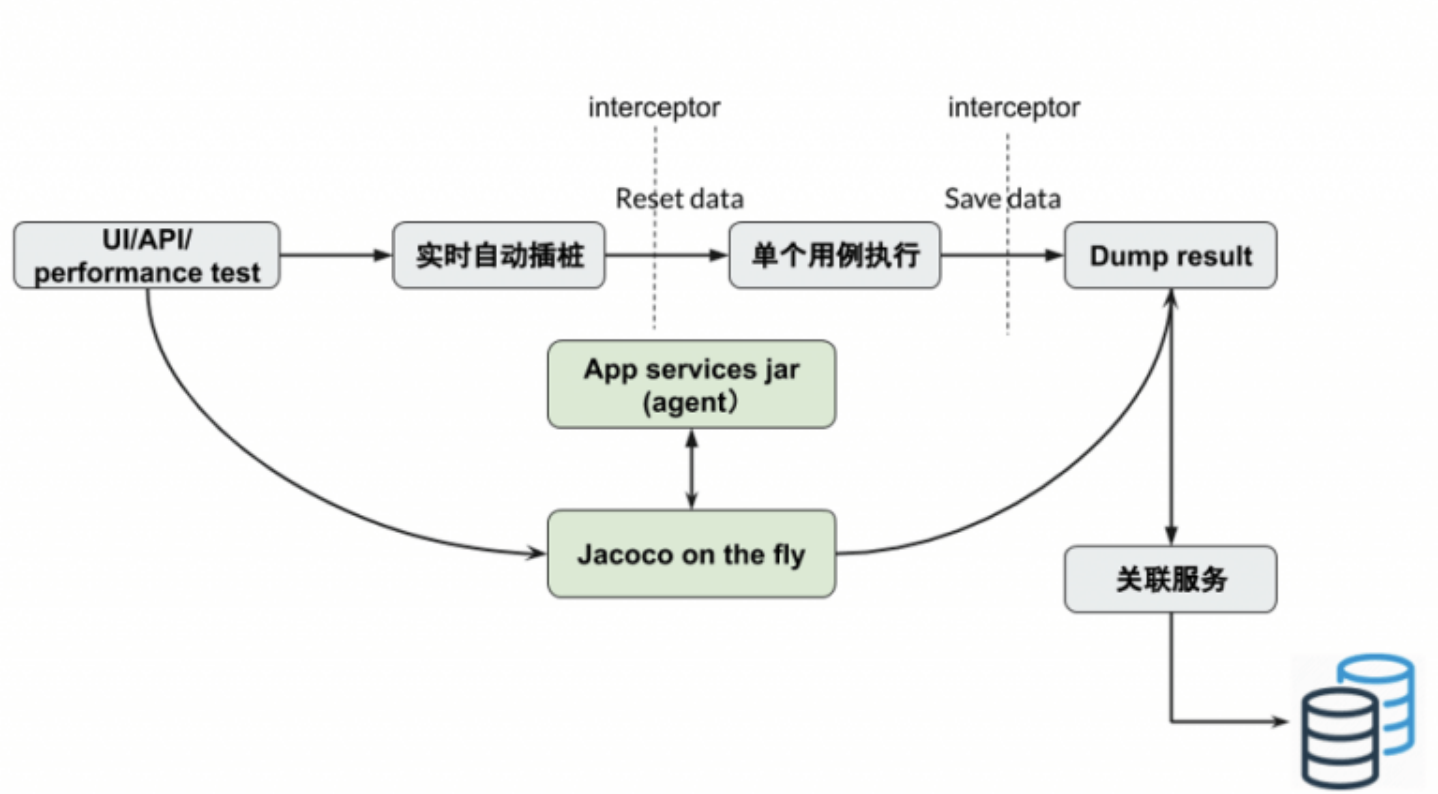


获取单个用例覆盖率

以上是基于java语言来关联用例与代码直接的关系，前端也有类似原理的工具，利用istanbul-middleware也可以实现同样的功能，具体请查阅一下这两篇文章，这里不再复述。

1. 自动化测试关联方案

利用AOP原理，在自动化框架的执行器加一个拦截器，在覆盖率收集开关打开时，请求执行前，这个过程类似于测试框架中的before hook：reset 被测服务桩数据，也就是上一次测试产生的dump 文件，请求执行后，这个过程类似于测试框架中的after hook：用api导出内存中的覆盖率数据，我们利用反射拿到用例方法名，利用关联服务将之与对应的 dump结果关联起来，实现自动插桩功能，快速帮助我们建立自动化用例与覆盖率之间的关系。



Offline模式（unit test采用模式，不是本文）：

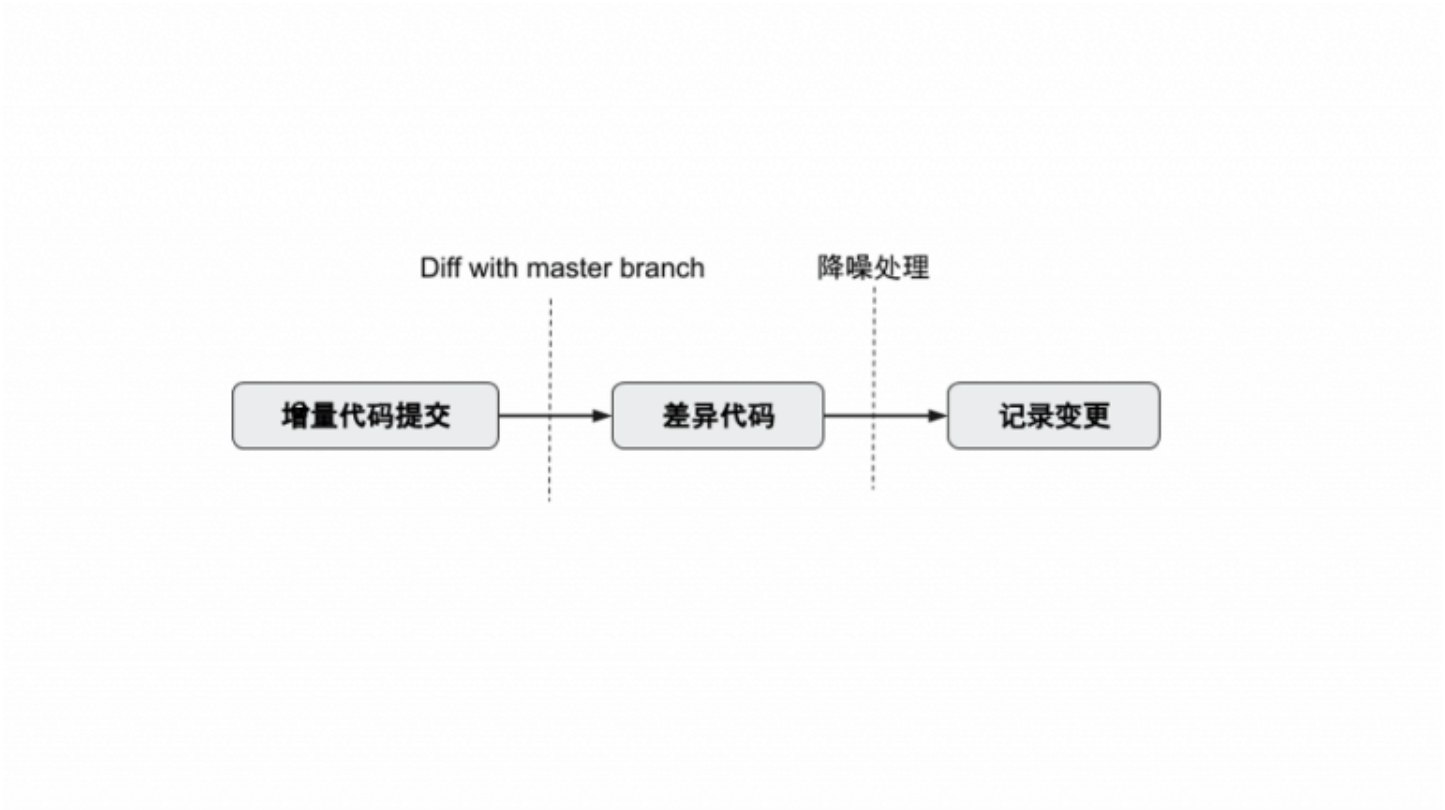
在测试前先对文件进行插桩，然后生成插过桩的class或jar包，测试插过桩 的class和jar包后，会生成动态覆盖信息到文件，最后统一对覆盖信息进行处理，并生成报告。

Offline模式适用于以下场景：

- 运行环境不支持java agent
- 部署环境不允许设置JVM参数
- 字节码需要被转换成其他虚拟机字节码，如Android Dalvik VM
- 动态修改字节码过程中和其他agent冲突
- 无法自定义用户加载类

影响面评估，分析识别增量与变更代码

我们有了代码与用例直接的关系的映射，我们需要将之用在开发流程中，首先我们需要得知我们的改动是什么，最直接的是通过 git diff 得知具体改动代码，但过于繁重，且太多干扰例如注释，空行等，最好的方法是实现比对算法，经过降噪处理，消除干扰，进而拿到处理后变更数据。



测试范围评估，用例筛选，链路分析

我们有了用例与代码之间的关系映射，有了提交增量代码差异记录，就可以实现逆向回溯。

利用代码的差异，通过查询服务就可以在上面提到关联关系数据库中反推影响的用例，以及上层的业务。这样可以帮助QA快速划分测试范围，减少过度测试。

总结精准测试的优点

1. 这种代码与用例，业务之间的关联关系，能够加深我们对被测系统及架构的了解，在不断的版本迭代过程中，能够实时了解任何类型测试对于当前版本的覆盖率，是否有遗漏的场景等，帮助团队更好的建立信心，使质量真正走上可持续化道路。
2. 精准测试在项目的中后期不在不依赖个人能力以及业务熟悉度等特点，大幅降低了团队测试的成本，使得团队QA能够有大量的时间做探索性测试以及质量度量上，提高QA对于团队的ROI，带给团队更清晰的质量数据。

但事物总是存在对立面的，获得巨大的收益同时，必然相应的存在缺点。否则也应当像UI自动化测试一样流行于各个公司以及团队中。

精准测试存在的问题

1. 基于手工测试的精准测试建立映射关系繁杂，如果需求改变频繁，用例维护以及之间的关系维护需要耗费大量时间精力。
2. 精准测试需要一定的自动化测试的覆盖，这样做起来更有意义，例如api自动化测试，如果本身用例过少，与代码之间关联关系不多时，变更代码后可能不会得出什么结果。
3. 最好有对应的用例管理系统，能够方便的帮助我们建立与代码之间的关系。
4. 需要投入开发能力强的QA或者测试开发建立整套系统环境，但长远考虑，将精准测试嵌入整个公司的质量平台中，不管对于新项目还说维护项目来说都是一种提升。
5. 项目生命周期需要较长，短期项目花费巨大精力开发和维护整套精准测试系统得不偿失。短期项目可以利用精准测试以api测试覆盖率作为衡量标准。不去建立繁杂的关系，只监控UI API测试覆盖率迭代时的变更来达到目的。

精准测试不是银弹，需要巨大的投入，用的好，能够成倍的提升质量，生产效率，用不好的话，就成了领导的KPI项目，弃之可惜，食之无味，鸡肋也。