美团点评

美团点评
2018 技术年货

CODE A BETTER LIFE

序

春节已近，年味渐浓。

又到了我们献上技术年货的时候。

不久前，我们已经给大家分享了技术沙龙大套餐，汇集了过去一年我们线上线下技术沙龙 99位讲师，85个演讲，70+小时 分享。

今天出场的，同样重磅——技术博客全年大合集。

2018年，是美团技术团队官方博客第5个年头， 博客网站 全年独立访问用户累计超过300万，微信公众号（meituantech）的关注数也超过了15万。

由衷地感谢大家一直以来对我们的鼓励和陪伴！

在2019年春节到来之际，我们再次精选了114篇技术干货，制作成一本厚达1200多页的电子书呈送给大家。

这本电子书主要包括前端、后台、系统、算法、测试、运维、工程师成长等7个板块。疑义相与析，大家在阅读中如果发现Bug、问题，欢迎扫描文末二维码，通过微信公众号与我们交流。

也欢迎大家转给有相同兴趣的同事、朋友，一起切磋，共同成长。

最后祝大家，新春快乐，阖家幸福。

目录 – 测试篇

Lego–美团接口自动化测试实践 4

智能支付稳定性测试实战 37

大众点评App的短视频耗电量优化实战 44

“小众”之美——Ruby在QA自动化中的应用 56

美团点评云真机平台实践 67

质量运营在智能支付业务测试中的初步实践 76

Lego-美团接口自动化测试实践

作者: 永达

一、概述

1.1 接口自动化概述

众所周知，接口自动化测试有着如下特点：

- 低投入，高产出。
- 比较容易实现自动化。
- 和UI自动化测试相比更加稳定。

如何做好一个接口自动化测试项目呢?

我认为，一个“好的”自动化测试项目，需要从**“时间”**、**“人力”**、**“收益”**这三个方面出发，做好“取舍”。

不能由于被测系统发生一些变更，就导致花费了几个小时的自动化脚本无法执行。同时，我们需要看到“收益”，不能为了总想看到100%的成功，而少做或者不做校验，但是校验多了维护成本一定会增多，可能每天都需要进行大量的维护。

所以做好这三个方面的平衡并不容易，经常能看到做自动化的同学，做到最后就本末倒置了。

1.2 提高ROI

想要提高ROI（Return On Investment，投资回报率），我们必须从两方面入手：

1. 减少投入成本。
2. 增加使用率。

针对“减少投入成本”

我们需要做到：

- **减少工具开发的成本。**尽可能的减少开发工具的时间、工具维护的时间，尽可能使用公司已有的，或是业界成熟的工具或组件。
- **减少用例录入成本。**简化测试用例录入的成本，尽可能多的提示，如果可以，开发一些批量生成测试用例的工具。
- **减少用例维护成本。**减少用例维护成本，尽量只用在页面上做简单的输入即可完成维护动作，而不是进行大量的代码操作。
- **减少用例优化成本。**当团队做用例优化时，可以通过一些统计数据，进行有针对性、有目的性的用例优化。

针对“增加使用率”

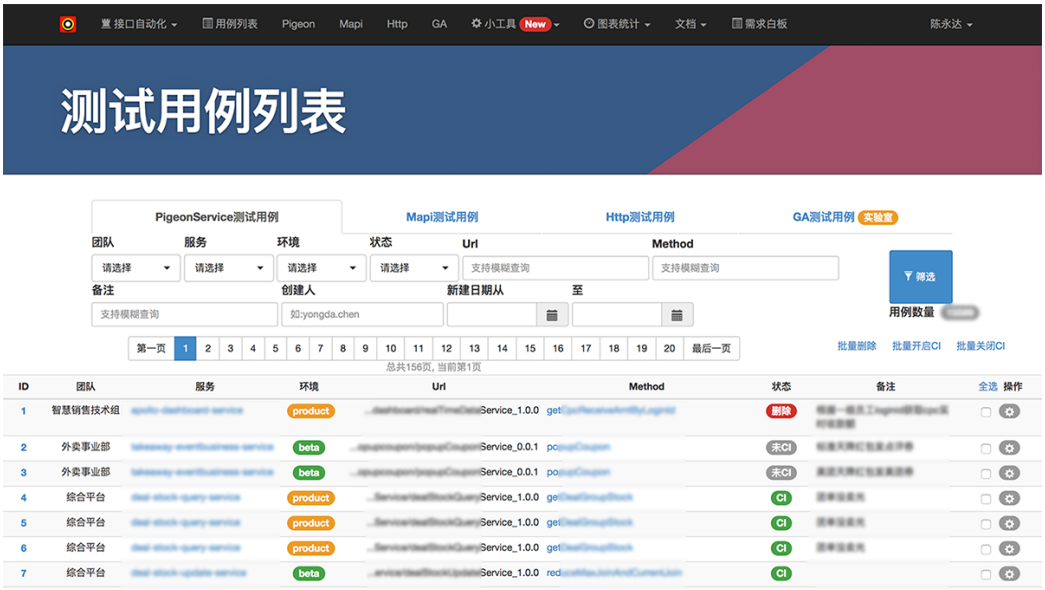
我们需要做到：

- **手工也能用。**不只是进行接口自动化测试，也可以完全用在手工测试上。
- **人人能用。**每一个需要使用测试的人，包括一些非技术人员都可以使用。
- **当工具用。**将一些接口用例当成工具使用，比如“生成订单”工具，“查找表单数据”工具。
- **每天测试。**进行每日构建测试。
- **开发的在构建之后也能触发测试。**开发将被测系统构建后，能自动触发接口自动化测试脚本，进行测试。

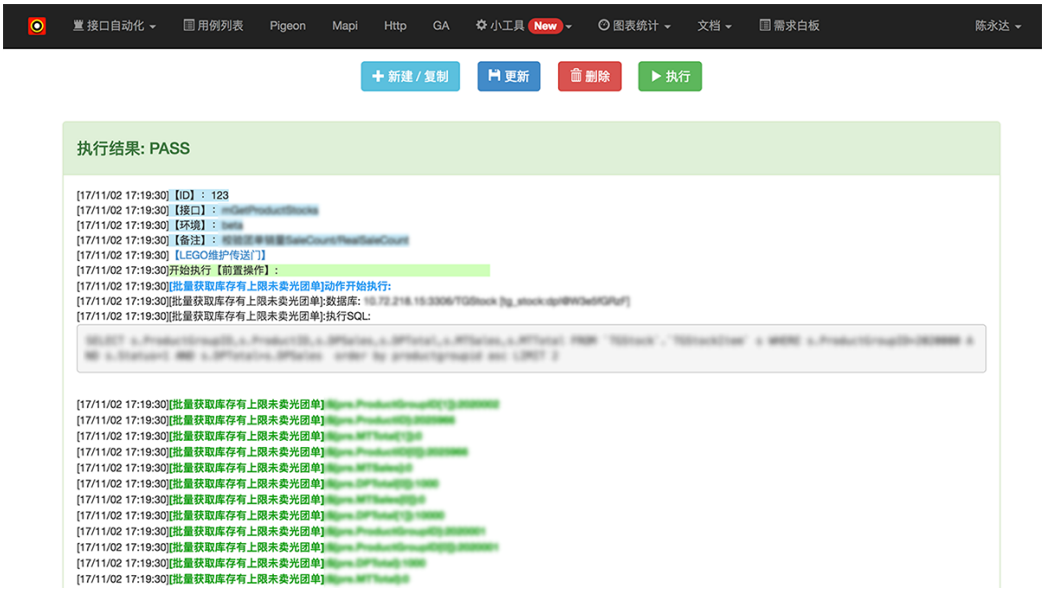
所以，我这边开发了Lego接口测试平台，来实现我对自动测试想法的一些实践。先简单浏览一下网站，了解一下大概是个什么样的工具。

首页

用例维护页面



自动化用例列表



在线执行结果

用例数量统计

1.3 Lego的组成

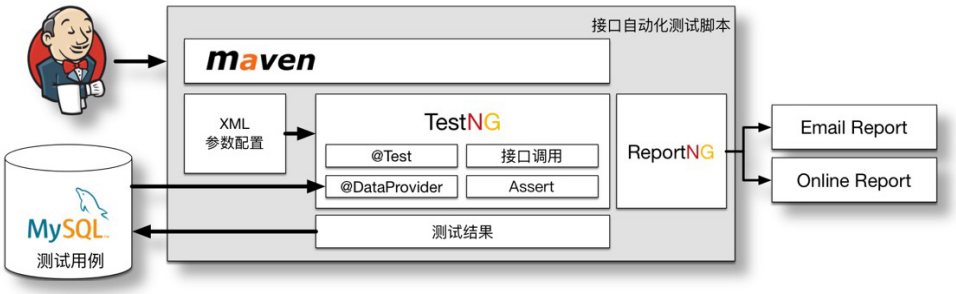
Lego接口测试解决方案是由两部分组成的，一个就是刚刚看到的“网站”，另一个部分就是“脚本”。

下面就开始进行“脚本设计”部分的介绍。

二、脚本设计

2.1 Lego的做法

Lego接口自动化测试脚本部分，使用很常见的Jenkins+TestNG的结构。



Jenkins+TestNG的结构

相信看到这样的模型并不陌生，因为很多的测试都是这样的组成方式。

将自动化测试用例存储至MySQL数据库中，做成比较常见的**“数据驱动”**做法。

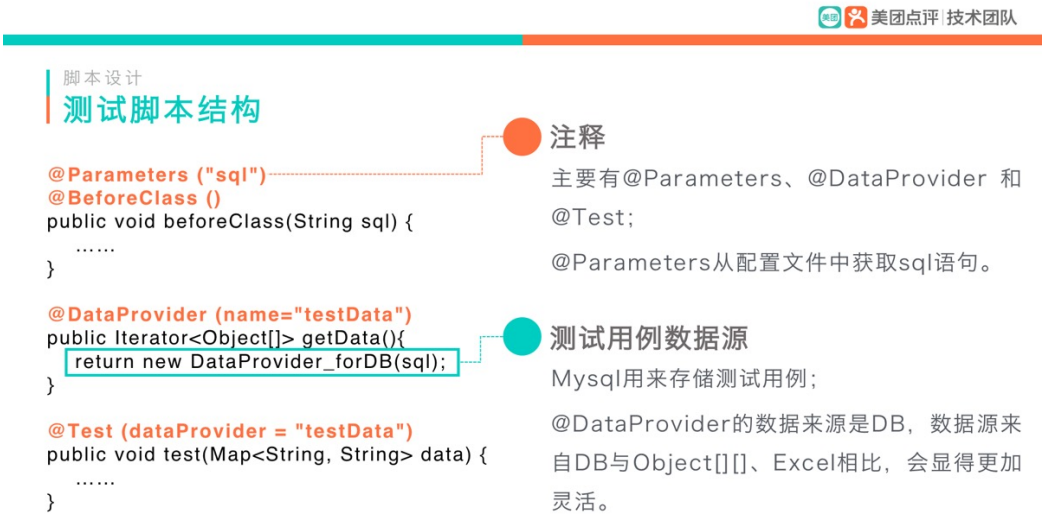
很多团队也是使用这样的结构来进行接口自动化，沿用的话，那在以后的“推广”中，学习和迁移成本低都会比较低。

2.2 测试脚本

首先来简单看一下目前的脚本代码：

```
public class TestPigeon {
    String sql;
    int team_id = -1;

    @Parameters({"sql", "team_id"})
    @BeforeClass()
    public void beforeClass(String sql, int team_id) {
        this.sql = sql;
        this.team_id = team_id;
        ResultRecorder.cleanInfo();
    }

    /**
     * XML中的SQL决定了执行什么用例，执行多少条用例，SQL的搜索结果为需要测试的测试用例
     */
    @DataProvider(name = "testData")
    private Iterator<Object[]> getData() throws SQLException, ClassNotFoundException {
        return new DataProvider_forDB(TestConfig.DB_IP, TestConfig.DB_PORT,
            TestConfig.DB_BASE_NAME,TestConfig.DB_USERNAME, TestConfig.DB_PASSWORD, sql);
    }

    @Test(dataProvider = "testData")
```

```
    public void test(Map<String, String> data) {
        new ExecPigeonTest().execTestCase(data, false);
    }

    @AfterMethod
    public void afterMethod(ITestResult result, Object[] objs) {...}

    @AfterClass
    public void consoleLog() {...}
}
```



测试脚本结构

有一种做法我一直不提倡，就是把测试用例直接写在Java文件中。这样做会带来很多问题：修改测试用例需要改动大量的代码；代码也不便于交接给其他同学，因为每个人都有自己的编码风格和用例设计风格，这样交接，最后都会变成由下一个同学全部推翻重写一遍；如果测试平台更换，无法做用例数据的迁移，只能手动的一条条重新输入。

所以“测试数据”与“脚本”分离是非常有必要的。

网上很多的范例是使用的Excel进行的数据驱动，我这里为什么改用MySQL而不使用Excel了呢?

在公司，我们的脚本和代码都是提交至公司的Git代码仓库，如果使用Excel……很显然不方便日常经常修改测试用例的情况。使用MySQL数据库就没有这样的烦恼了，由于数据与脚本的分离，只需对数据进行修改即可，脚本每次会在数据库中读取最新的用例数据进行测试。同时，还可以防止一些操作代码时的误操作。

这里再附上一段我自己写的 `DataProvider_forDB` 方法，方便其他同学使用在自己的脚本上：

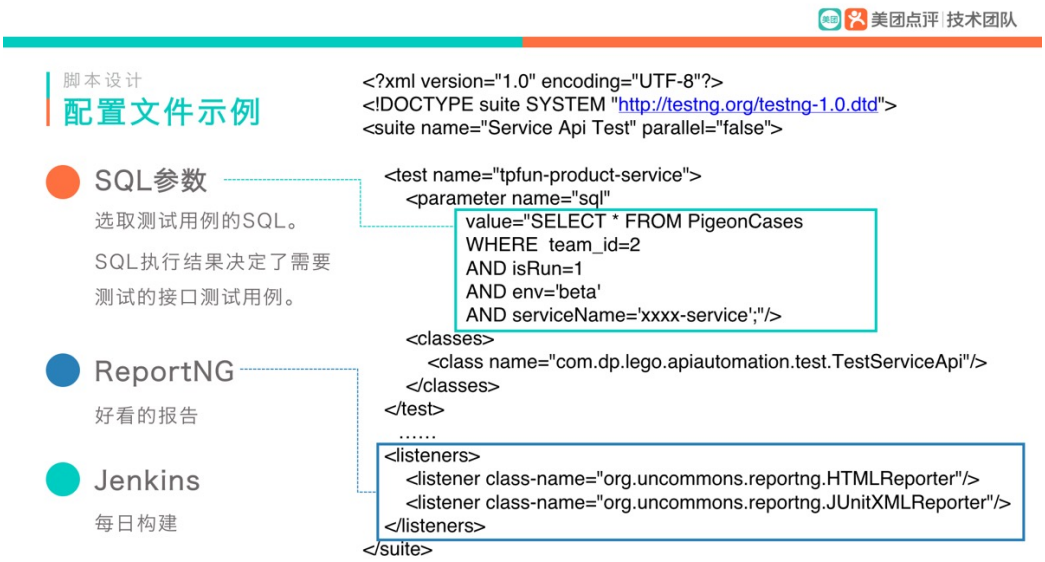
```
import java.sql.*;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

/**
 * 数据源 数据库
 *
 * @author yongda.chen
 */
public class DataProvider_forDB implements Iterator<Object[]> {

    ResultSet rs;
    ResultSetMetaData rd;
```

```
    public DataProvider_forDB(String ip, String port, String baseName,
        String userName, String password, String sql) throws ClassNotFoundException, SQLException {

        Class.forName("com.mysql.jdbc.Driver");
        String url = String.format("jdbc:mysql://%s:%s/%s", ip, port, baseName);
        Connection conn = DriverManager.getConnection(url, userName, password);
        Statement createStatement = conn.createStatement();

        rs = createStatement.executeQuery(sql);
        rd = rs.getMetaData();
    }

    @Override
    public boolean hasNext() {
        boolean flag = false;
        try {
            flag = rs.next();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return flag;
    }

    @Override
    public Object[] next() {
        Map<String, String> data = new HashMap<String, String>();
        try {
            for (int i = 1; i <= rd.getColumnCount(); i++) {
                data.put(rd.getColumnName(i), rs.getString(i));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        Object r[] = new Object[1];
        r[0] = data;
        return r;
    }

    @Override
    public void remove() {
        try {
            rs.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

2.3 配置文件

上面图中提到了“配置文件”，下面就来简单看一下这个XML配置文件的脚本：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="Pigeon Api测试" parallel="false">

    <test name="xxx-xxx-service">
        <parameter name="sql"
                   value="SELECT * FROM API_PigeonCases
                   WHERE team_id=2
                   AND isRun=1
                   AND service='xxx-xxx-service'
                   AND env='beta';"/>
        <classes>
            <class name="com.dp.lego.test.TestPigeon"/>
        </classes>
    </test>

    <listeners>
        <listener class-name="org.uncommons.reportng.HTMLReporter"/>
        <listener class-name="org.uncommons.reportng.JUnitXMLReporter"/>
    </listeners>
</suite>
```

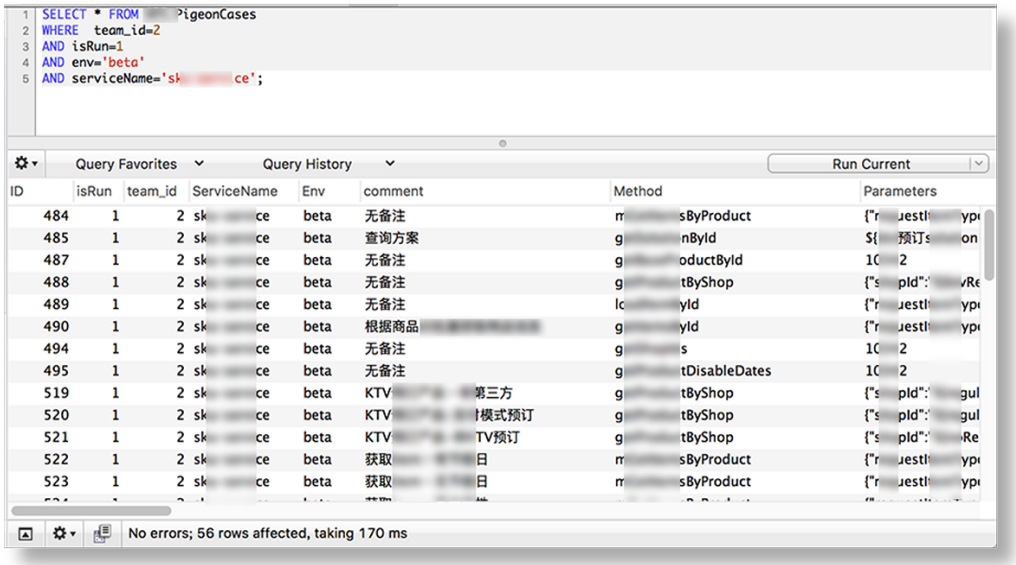


对照上图来解释一下配置文件：

- SQL的话，这里的SQL主要决定了选取哪些测试用例进行测试。
- 一个标签，就代表一组测试，可以写多个标签。
- “listener”是为了最后能够生成一个ReportNG的报告。
- Jenkins来实现每日构建，可以使用Maven插件，通过命令来选择需要执行的XML配置。

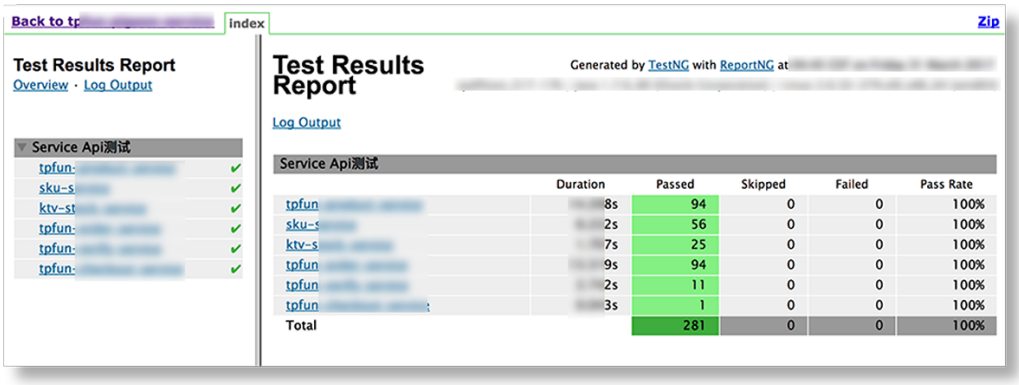
这样做有什么好处呢?

使用SQL最大的好处就是灵活



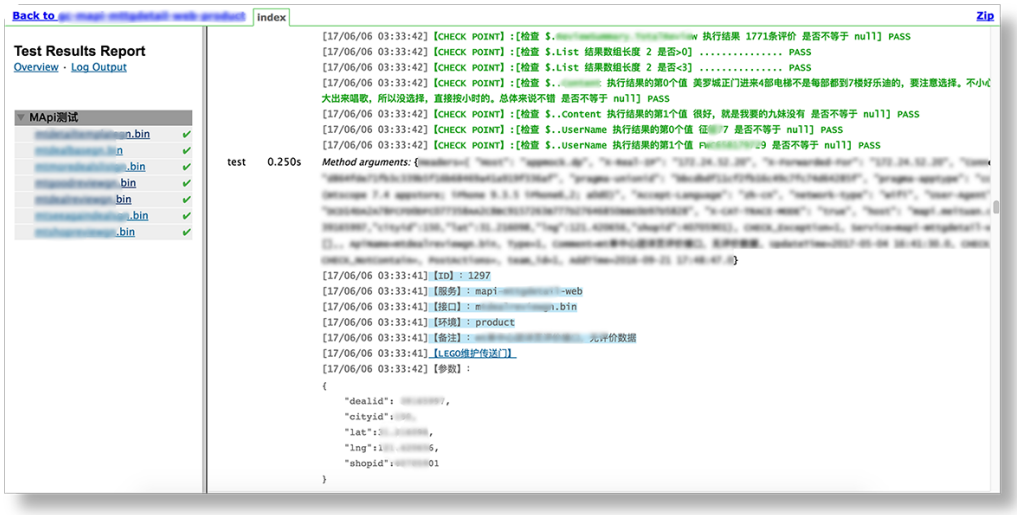
如上面的这个例子，在数据库中会查询出下面这56条测试用例，那么这个标签就会对这56条用例进行逐一测试。

多<test>标签时，可以分组展示

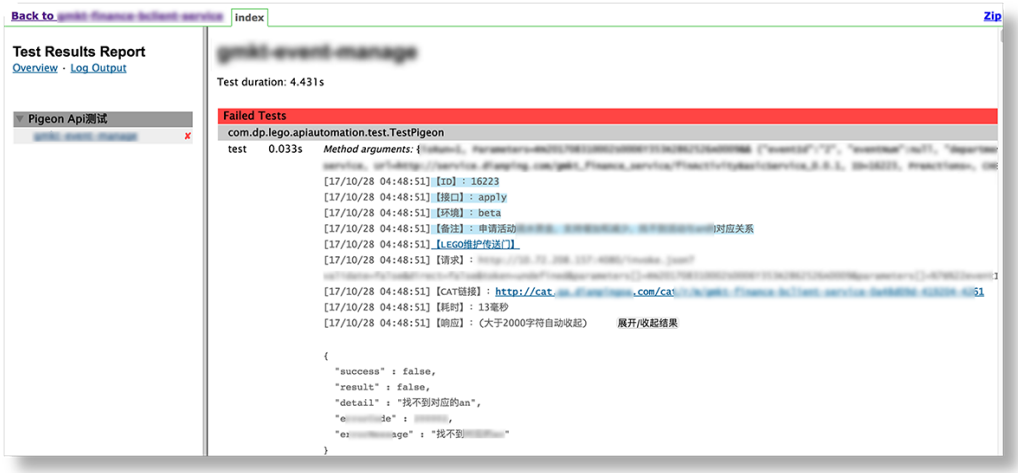


使用多个 `<test>` 标签来区分用例，最大的好处就是也能在最后的报告上，达到一个分组展示的效果。

报告更美观丰富

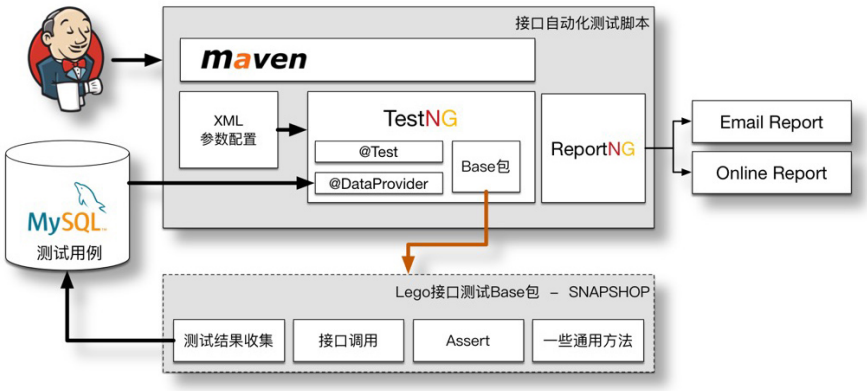


由于使用了 `ReportNG` 进行报告的打印，所以报告的展示要比TestNG自带的报告要更加美观、并且能自定义展示样式，点开能看到详细的执行过程。



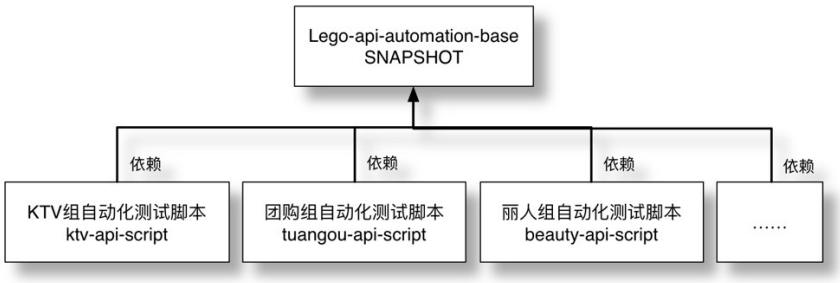
如果有执行失败的用例，通常报错的用例会在最上方优先展示。

支持多团队

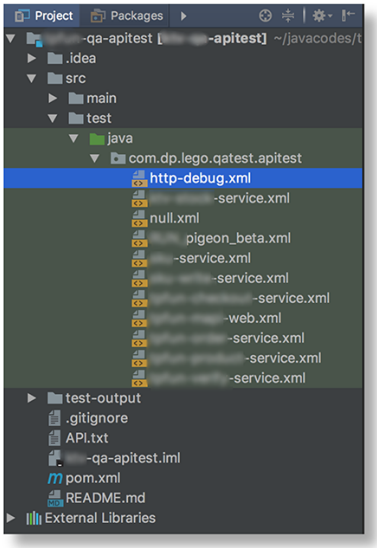


当两个团队开始使用时，为了方便维护，将基础部分抽出，各个团队的脚本都依赖这个Base包，并且将Base包版本置为“SNAPSHOT版本”。使用“SNAPSHOT版本”的好处是，之后我对Lego更新，各个业务组并不需要对脚本做任何改动就能及时更新。

当更多的团队开始使用后，比较直观的看的话是这个样子的：



每个团队的脚本都依赖于我的这个Base包，所以最后，各个业务团队的脚本就变成了下面的这个样子：



可以看到，使用了Lego之后：

- 没有了Java文件，只有XML文件
- xml中只需要配置SQL。
- 执行和调试也很方便。

- 可以右键直接执行想要执行的测试配置。
- 可以使用maven命令执行测试：
  - `mvn clean test -U -Dxml=xmlFileName` 。
  - 通过参数来选择需要执行的xml文件。
- 也可以使用Jenkins来实现定时构建测试。

由于，所有测试用例都在数据库所以这段脚本基本不需要改动了，减少了大量的脚本代码量。

有些同学要问，有时候编写一条接口测试用例不只是请求一下接口就行，可能还需要写一些数据库操作啊，一些参数可能还得自己写一些方法才能获取到啊之类的，那不code怎么处理呢?

下面就进入“用例设计”，我将介绍我如何通过统一的用例模板来解决这些问题。

三、用例设计

3.1 一些思考

我在做接口自动化设计的时候，会思考通用、校验、健壮、易用这几点。

通用

- **简单、方便**
  - 用例数据与脚本分离，简单、方便。
  - 免去上传脚本的动作，能避免很多不必要的错误和维护时间。
  - 便于维护。
- **模板化**
  - 抽象出通用的模板，可快速拓展。
  - 数据结构一致，便于批量操作。
  - 专人维护、减少多团队间的重复开发工作。
  - 由于使用了统一的模板，那各组之间便可交流、学习、做有效的对比分析。
  - 如果以后这个平台不再使用，或者有更好的平台，可快速迁移。
- **可统计、可拓展**
  - 可统计、可开发工具；如：用例数统计，某服务下有多少条用例等。
  - 可开发用例维护工具。
  - 可开发批量生成工具。

校验

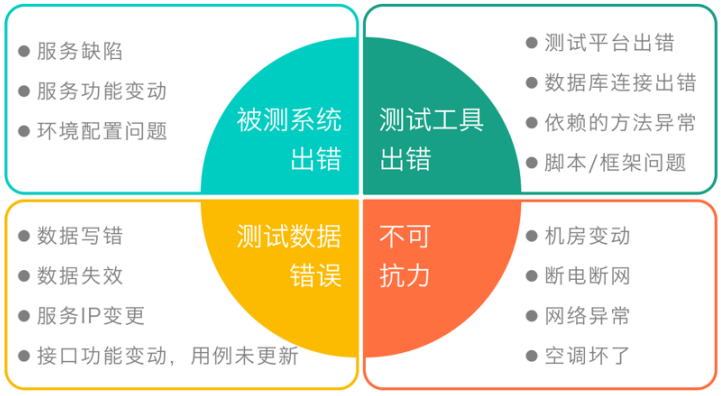
在写自动化脚本的时候，都会想“细致”，然后“写很多”的检查点；但当“校验点”多的时候，又会因为很多原因造成执行失败。所以我们的设计，需要在保证充足的检查点的情况下，还要尽可能减少误报。

- **充足的检查点**
  - 可以检查出被测服务更多的缺陷。
- **尽量少的误报**
  - 可以减少很多的人工检查和维护的时间人力成本。

- **还要**
  - 简单、易读。
  - 最好使用一些公式就能实现自己想要的验证。
  - 通用、灵活、多样。
  - 甚至可以用在其他项目的检查上，减少学习成本。

健壮

执行测试的过程中，难免会报失败，执行失败可能的原因有很多，简单分为4类：



- **被测系统出错**，这部分其实是我们希望看到的，因为这说明我们的自动化测试真正地发现了一个Bug，用例发挥了它的价值，所以，这是我们希望看到的。
- **测试工具出错**，这部分其实是我们不希望看到的，因为很大可能我们今天的自动化相当于白跑了。
- **测试数据错误**，这是我们要避免的，既然数据容易失效，那我在设计测试平台的时候，就需要考虑如果将所有的数据跑“活”，而不是只写“死”。
- **不可抗力**，这部分是我们也很无奈的，但是这样的情况很少发生。

那针对上面的情况：

- **参数数据失效**
  - 支持实时去数据库查询。
  - 支持批量查。
- **IP进场发生变更**
  - 自动更新IP。
- **灵活、可复用**
  - 支持批量维护。
  - 接口测试执行前生成一些数据。
  - 接口执行完成后销毁一些数据。
  - 支持参数使用另一条测试用例的返回结果。
  - 支持一些请求参数实时生成，如token等数据，从而减少数据失效的问题。

通过这些手段，提高测试用例的健壮性，让每一条自动化测试用例都能很好的完成测试任务，真正发挥出一条测试用例的价值。

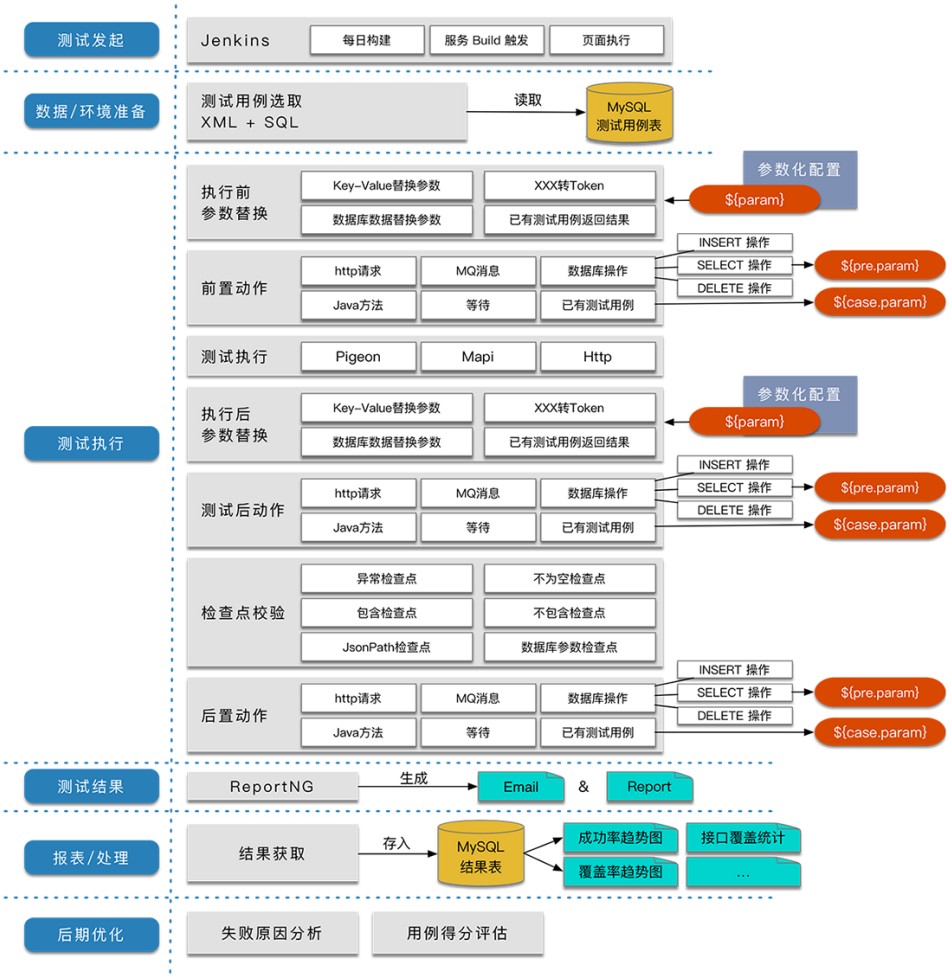
易用

- **简单**
  - 功能强大，但要人人会用。
  - 非技术人员也要会用。
- **减少代码操作**
  - 让自动化开发人员注意力能更多的放在用例本身，而不是浪费在无关紧要的开发工作上面。
- **还要**
  - 配置能复用。
  - 通用、易学。
  - 一些数据能自动生成。

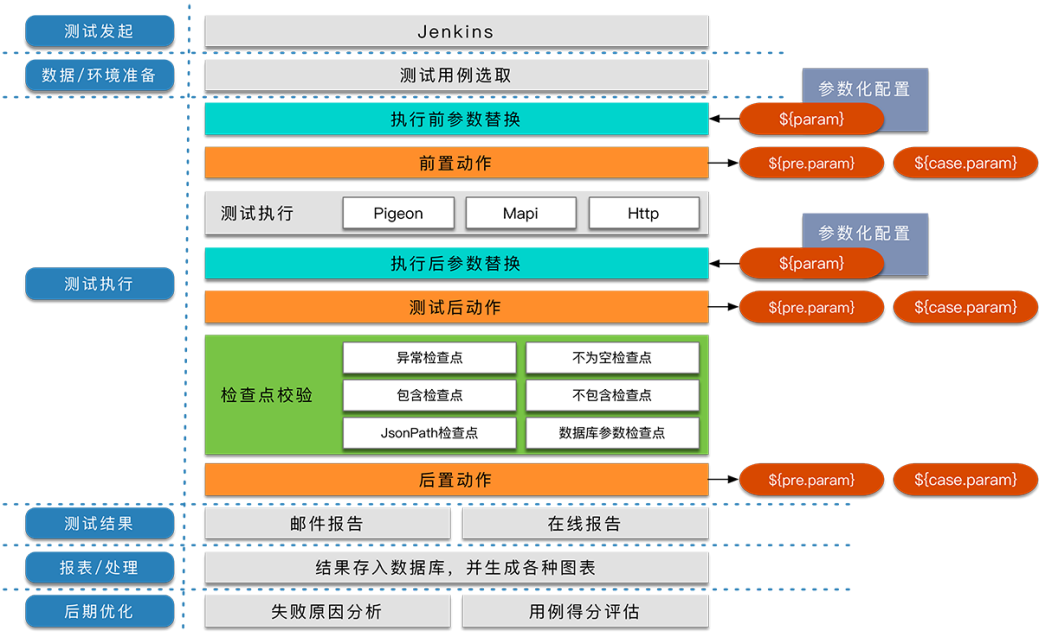
3.2 Lego接口自动化测试用例

说了这么多，那我们来看一下一条Lego接口测试用例的样子。

一条Lego自动用例执行顺序大概是如下图这样：



简单区分一下各个部分，可以看到：



那上面图中提到了两个名词：

- **“参数化”**
- **“前后置动作”**

下面会先对这两个名词做一个简单的介绍。

3.3 参数化

比如一个请求需要用到的参数。

```
{
        "sync": false,
        "cityId": 1,
        "source": 0,
        "userId": 1234,
        "productId": 00004321
}
```

这个例子中有个参数 `"productId": 00004321` ，而由于测试的环境中，表单00004321很可能一些状态已经发生了改变，甚至表单已经删除，导致接口请求的失败，那么这时候，就很适合对 `"productId": 00004321` 进行参数化，比如写成这样：

```
{
        "sync": false,
        "cityId": 1,
        "source": 0,
        "userId": 1234,
        "productId": ${myProductId}
}
```

所以对**“参数化”**简单的理解就是：

> **通过一些操作，将一个“值”替换掉测试用例里的一个“替代字符”**

`${myProductId}` 的值可以通过配置获取到：

- Key-Value

    - 配置 Value=00004321。
- **SQL获取**
    - 执行一个select语句来实时查询得到可用ID。
- **已有测试用例**
    - 某个接口接口测试用例的返回结果。

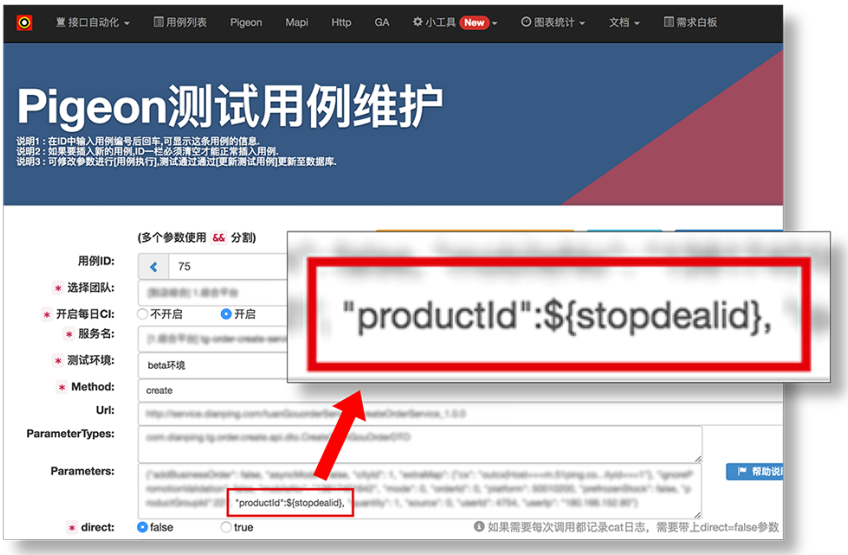
“参数化”实例

下面我们来看一个“参数化”的实例：

（1） 首先我们在参数化维护页面中新建一个参数化， `shopdealid` **。**

通过配置我们可以看到这个参数的值，是执行了一条SQL后，取用执行结果中 `DealID` 字段的值。

（2） 在用例中，将需要这个表单号的地方用${shopdealid}替代。



那在编写测试用例的时候，大家可以看一下这个放大的图片，在这里的ProductID的值并不是硬代码一个固定的表单号，而是选择了刚才配置的参数化数据。

（3） 执行结果中，${shopdealid} 变为实时查询数据库的来的一个真实的表单号。

从结果中可以看到，我们的这个参数被替换成了一个有效的值，而这个值就是我们刚刚配置的那个SQL实时查询而来的。

“参数化”的场景

多个测试用例使用同一个参数进行测试

如50条测试用例都使用同一个id作为参数进行测试，这时候我们需要变更这个id。

无参数化时：

- 需要修改50次，即每条测试用例中的id都得进行修改。
- 可能会有遗漏。 有参数化时：
- ID部分用 ${myID} 替代。
- 需要修改的话，在“参数化维护”页面中维护 ${myID}这条数据就可以。修改一次，所有使用${myID}的用例都配置完成。

测试数据过期导致测试用例执行失败

如一条用例参数需要传入Token，但是Token会因为时间问题而导致过期，这时候用例就失败了。

无参数化时：

- 经常修改Token，或是写一段ID转Token的代码。
- 方法可能会重复编写。
- 多个团队之间可能实现方式也不同。

有参数化时：

- 使用参数化工具，Lego统一管理。
- 维护一个参数化 如： `${测试用Token} = id:123` 。

数据库获取有效测试数据

参数中需要传入DealId作为参数，写死参数的话，如果这个DealId被修改引起失效，那这条测试用例就会执行失败。

不使用Lego时：

- 测试环境中，一个订单时常会因为测试需要被修改数据，导致单号失效，最后导致自动化失败。
- 编写相关代码来做好数据准备工作。
- 在代码中编写读取数据库的方法获取某些内容。

在Lego上的方案： – 使用参数化，实时获取sql结果，查询出一条符合条件的dealId来实现。 – 使用参数化，调用写好的“生成订单”接口用例实现，拿单号来实现。 – 前后置动作，插入一条满足条件的数据。

3.4 前后置动作

“前后置动作”的概念就比较好理解了：

在接口请求之前（或之后），执行一些操作

目前前后置动作支持6种类型：

- 数据库SQL执行
  - 有时候在执行接口请求前，为了保证数据可用，可能需要在数据库中插入或删除一条信息，这时候就可以使用前后置动作里的“执行SQL语句”类型，来编写在接口请求前（后）的 Insert 和 Delete 语句。
- 已有测试用例执行
  - 比如当前测试用例的请求参数，需要使用另一条测试用例的返回结果，这时候就可以使用“执行测试用例”类型，写上Lego上某条测试用例的ID编号，就可以在当前用例接口请求前（后）执行这条测试用例。
  - 前后置动作中测试用例的返回结果可以用于当前用例的参数，对测试用例返回结果内容的获取上，也支持JsonPath和正则表达式两种方式。
- MQ消息发送
  - 在接口请求前（后）发送MQ消息。
- HTTP请求
- 等待时间
- 自定义的Java方法
  - 如果上面的方法还满足不了需求，还可以根据自己的需要，编写自己的Java方法。
  - 可以在Lego-Kit项目中，编写自己需要的Java方法，选择“执行Java方法”，通过反射实现自定义Java方法的执行。

这里的SQL同时支持Select操作，这里其实也是做了一些小的设计，会将查询出来的全部的结果，放入到这个全局Map中。

比如查询一条SQL得到下表中的结果：

id | name | age | number :—: | :—: | :—: | :—: | :—: 0 | 张三 | 18 | 1122 1 | 李四 | 30 | 3344

那我们可以使用下面左边的表达式，得到对应的结果：

- `${pre.name}` —– 得到“张三”å
- `${pre.age}` —– 得到 18
- `${pre.number}` —– 得到 1122

也可以用：

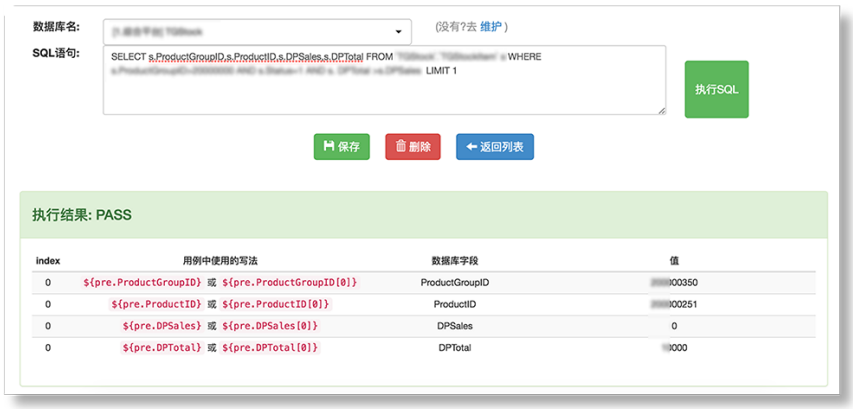
- `${pre.name[0]}` —– 得到“张三”
- `${pre.age[0]}` —– 得到 18
- `${pre.number[0]}` —– 得到 1122
- `${pre.name[1]}` —– 得到“李四”
- `${pre.age[1]}` —– 得到 30
- `${pre.number[1]}` —– 得到 3344

这样的设计，更加帮助在用例设计时，提供数据准备的操作。

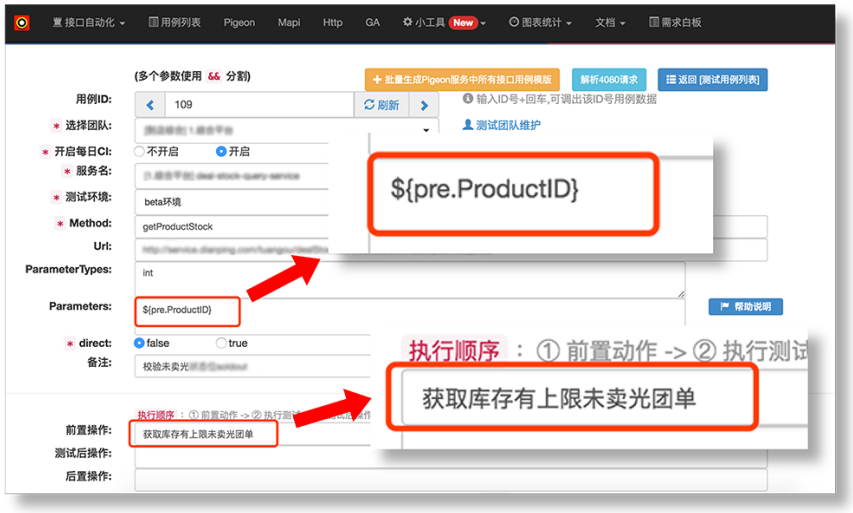
“前后置动作”实例

（1） 首先我们在前后置维护页面中新建一个动作， 获取库存上限未卖光团单 。

这个配置也是可以支持在线调试的，在调试中，可以看到可以使用的参数化：

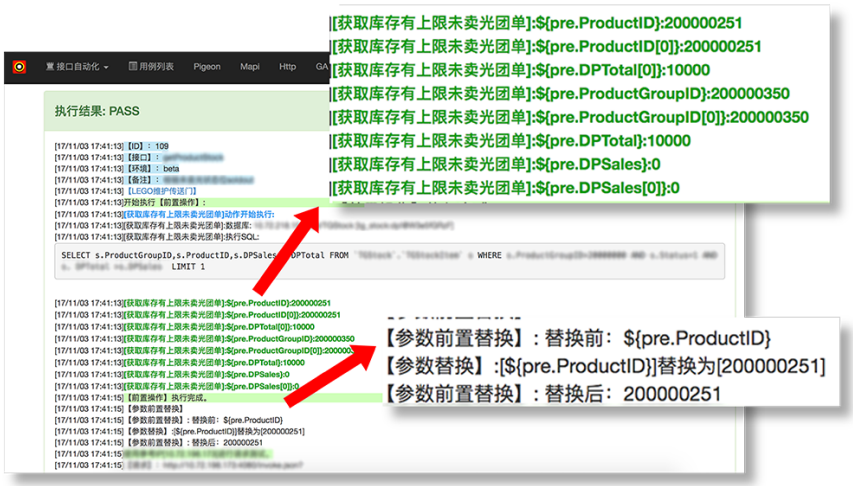


（2） 在测试用例中的前置动作，添加 获取库存上限未卖光团单 **。**



这样就可以在整个测试用例中，使用 `${pre.ProductID}` ，来替换掉原有的数据信息。

（3） 最后请求接口，返回了执行成功 。



Q & A

Q：那如果同样是获取三个参数，使用3个“参数化的Select操作”和使用1个“前置动作的Select操作”又有什么不同呢?

A： 不同在于执行时间上。 比如，我们查询最新的有效团单的“单号”“下单人”和“手机号”三个字段。 使用3个“参数化的Select操作”： 可能当执行${单号}的时候得到的订单号是“10001”，但是当执行到${下单人}的时候，可能有谁又下了一单，可能取到的下单人变成了“10002”的“李四”而不是“10001”的“张三”了，最后可能“单号”“下单人”和“手机号”三个字段去的数据并非同一行的数据。 而使用“前置动作的Select操作”： 就可以避免上面的问题，因为所有字段的数据是一次性查询出来的，就不会出现错位的情况。

Q：那“参数化的Select操作”和“前置动作的Select操作”这样不同的取值时机又有什么好用之处呢?

A： 由于“前置动作”一定是接口请求前执行，“参数化”一定是用到的时候才执行这样的特性。 所以在检查点中，如果要验证一个数据库字段在经过接口调用后发生了变更，那使用“前置动作”和“参数化”同时去查询这个字段，然后进行比较，不一致就说明发生了变化。 所以根据使用场景，选择合适的参数化方式，很重要，选择对了，能大大提升测试用例的测试数据健壮性。

3.5 执行各部分

回到一开始的流程图，可以按照一类一类来看执行过程。

测试发起

测试发起基本还是使用的Jenkins，稳定、成熟、简单、公司工具组支持，也支持从Lego的Web页面进行执行操作。

数据 / 环境准备

使用 @DataProvider 的方式，从DB数据库中读取测试用例，逐一执行进行测试。

测试执行

在正式执行测试用例之前，会先进行一波参数替换的动作，在调用接口之后，还会执行一次参数替换动作。

参数替换后会进行前置动作的执行，然后在调用接口之后还会执行测试后动作，最后执行后置动作。

接口请求这部分就没什么好说的了，就是通过接口请求的参数，请求对应的接口，拿到返回结果。

这里的话是为了方便通用，所以要求返回的结果都是使用的String类型。这样做最大的好处就是。比如说我现在有一种新的接口类型需要接入。那只需要写一个方法能够请求到这个接口，并且拿到String类型的返回结果，就可以很快将新的接口类型接入Lego测试平台进行接口测试。

检查点校验

检查点部分是一条自动化测试用例的精髓，一条自动化测试用例是否能真正的发挥它的测试功能，就是看QA对这条测试用例的检查点编写是否做了良好设计。在Lego平台上，目前我拥有的检查点有6种不同的类型。

> - **异常检查点**

    - 当返回结果为异常时，则会报错。
    - 但是有时候为了做异常测试，可以将这个检查点关掉。
- **不为空检查点**
    - 顾名思义，当出现”“、”[]“、”{}“、null 这样的的结果，都会报错。也可以根据自己用例的实际情况关闭。
- **包含检查点**
- **不包含检查点**
    - “包含”和“不包含”检查点是将接口的返回结果作为一个String类型来看，检查所有返回内容中是否“包含”或“不包含”指定的内容。
- **数据库参数检查点**
    - 顾名思义，不做过多的解释了。
- **JsonPath检查点**
    - 这是我在Lego上设计的最具有特色的一种检查点类型。

JsonPath的基本写法是：{JsonPath语法}==value

JsonPath的语法和XPath的语法差不多，都是根据路径的方法找值。这里也是主要是针对返回结果为JSON数据的结果，进行检查。

具体的JsonPath语法可以参考： https://github.com/json-path/JsonPath

说完了**“JsonPath的语法”**，现在说一下**“JsonPath检查点的语法”**，**“JsonPath检查点的语法”**是我自己想的，主要针对以下几种数据类型进行校验：

(1) 字符串类型结果检验

- 等于： `==`
- 不等于： `!==`
- 包含： `=`
- 不包含： `!=`

例如：

- `{$.[1].name}==aa` ： 检查返回的JSON中第2个JSON的name字段是否等于aa。
- `{$..type}=='14'` ： 检查返回的JSON中每一个JSON的name字段是否等于aa。
- `{$.[1].type}==14 && {$.[1].orderId}==106712` ： 一条用例中多个检查用&&连接。
- `{$..orderId}!==12` ： 检查返回的JSON中每个JSON的orderId字段是否不等于12。
- `{$..type}=1` ： 检查返回的JSON中每个JSON的type字段是否包含1。
- `{$.[1].type}!=chenyongda` ： 检查返回的JSON中第2个JSON的type字段是否不包含chenyongda。

(2) 数值校验

- 等于： `=`
- 大于： `>`
- 大于等于： `>=`
- 小于： `<`

- 小于等于： `<=`

例如：

- {$.[0].value}<5：检查返回的JSON中第1个JSON的value字段的列表是否小于3。
- {$.[1].value}>4：检查返回的JSON中第2个JSON的value字段的列表是否大于4。

(3) List结果检验

- list长度： `.length`
- list包含： `.contains(param)`
- list成员： `.get(index)`

例如：

- `{$..value}.length=3` ： 检查返回的JSON中每个JSON的value字段的列表是否等于3。
- `{$.[0].value}.length<5` ： 检查返回的JSON中第1个JSON的value字段的列表是否小于3。
- `{$.[1].value}.length>4` ： 检查返回的JSON中第2个JSON的value字段的列表是否大于4。
- `{$..value}.contains('222')` ： 检查返回的JSON中每个JSON的value字段的列表是否包含222字符串。
- `{$.[0].value}.contains(1426867200000)` ： 检查返回的JSON中第1个JSON的value字段的列表是否包含1426867200000。
- `{$.[0].value}.get(0)=='222'` ： 检查返回的JSON中第1个JSON的value字段的列表中第1个内容是否等于222。
- `{$..value}.get(2)='22'` ： 检查返回的JSON中每个JSON的value字段的列表中第3个内容是否包含22。

(4) 时间类型处理

时间戳转日期时间字符串： `.todate`

例如：

- `{$..beginDate}.todate==2015-12-31 23:59:59` ： 检查返回的JSON中beginDate这个时间戳转换成日期后是否等于2015-12-31 23:59:59。

当JsonPath返回的结果是列表的形式时

检查点	检查点等号左边	期望值	验证效果
{$.value}==“good”	[‘good’, ‘good’, ‘bad’, ‘good’]	“good”	作为4个检查点，会拿列表里的每个对象逐一和“期望值”进行检验，每一次对比都是一个独立的检查点。
{$.value}==[“good”]	[‘good’, ‘good’, ‘bad’, ‘good’]	[“good”]	作为1个检查点，作为一个整体做全量比对。
{$.value}==[‘a’, ‘b’]	[[‘a’, ‘b’],[‘a’, ‘b’],[‘a’, ‘b’, ‘c’]]	[‘a’, ‘b’]	作为3个检查点，道理和1一样，列表中的数据分别和期望值做比较。

除此之外，还有非常多的花样玩法

JsonPath中的检查支持“参数化”和“前后置动作”，所以会看到很多如：

{$.param}=‘${param}’ && {$.param}==${pre.param}

这样的检查点：

“参数化”和“前后置动作”也支持递归配置，这些都是为了能够让接口自动化测试用例写的更加灵活好用。

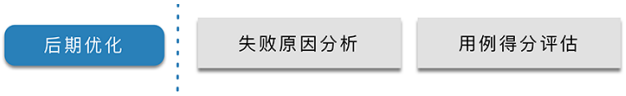
测试结果

使用ReportNG可以打印出很漂亮的报告。

报告会自定义一些高亮等展示方式，只需要在ReportNG使用前加上下面的语句，就可以支持“输出逃逸”，可使用HTML标签自定义输出样式。

```
System.setProperty("org.uncommons.reportng.escape-output", "false");
```

后期优化



当使用Jenkins执行后，通过Jenkins API 、和Base包中的一些方法，定时获取测试结果，落数据库，提供生成统计图表用。

四、网站功能

4.1 站点开发

既然打算做工具平台了，就得设计方方面面，可惜人手和时间上的不足，只能我一人利用下班时间进行开发。也算是担任了Lego平台的产品、后端开发、前端开发、运维和测试等各种角色。

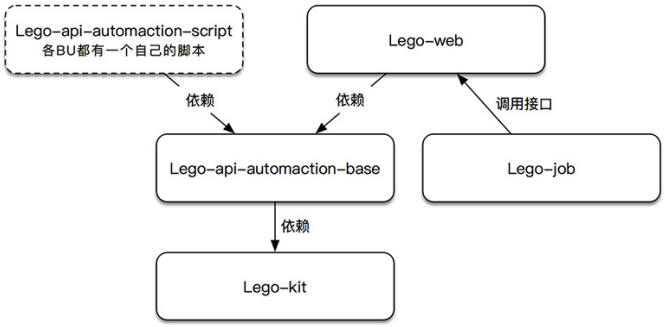
Jenkins+TestNG+ReportNG+我自己开发的基本接口自动化测试Base jar包，基本上没什么太大难度。但是站点这块，在来美团之前，还真没开发过这样的工具平台，这个算是我的第一个带Web界面的工具。边Google边做，没想到不久还真的架起来了一个简易版本。

使用 Servlet + Jsp 进行开发，前端框架使用Bootstrap，前端数据使用jstl，数据库使用MySQL，服务器使用的公司的一台Beta环境Docker虚拟机，域名是申请的公司内网域名，并开通北京上海两侧内网访问权限。

功能上基本都是要满足的，界面上，虽然做不到惊艳吧，但是绝对不能丑，功能满足，但是长得一副80年代的界面，我自己都会嫌弃去使用它，所以界面上我还是花了一些时间去调整和设计。熟练以后就快多

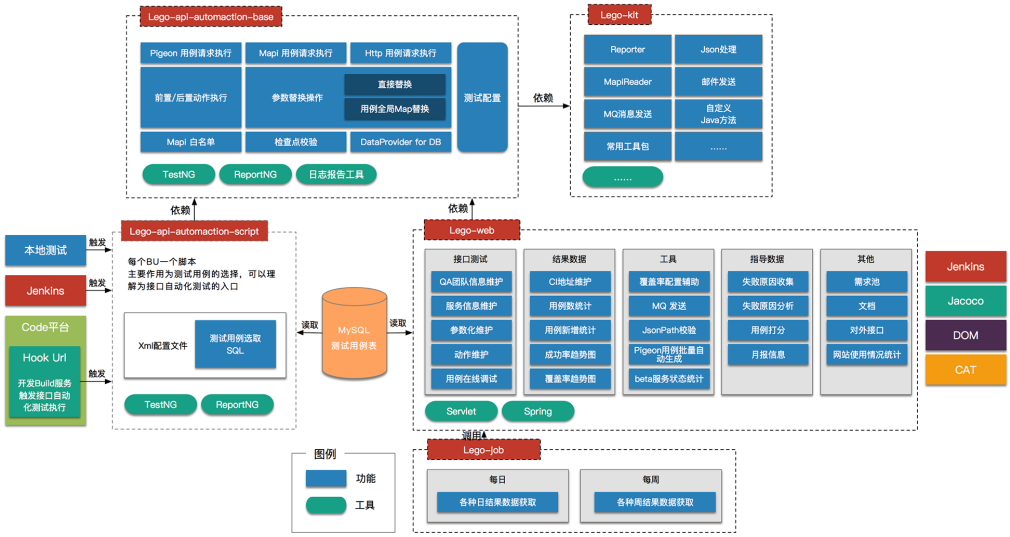
了。

4.2 整体组成



目前Lego由五个不同的项目组成，分别是“测试脚本”、“Lego-web页面项目”、“用于执行接口测试的base包”、“小工具集合Lego-kit”和“lego-job”，通过上图可以看出各项目间的依赖关系。

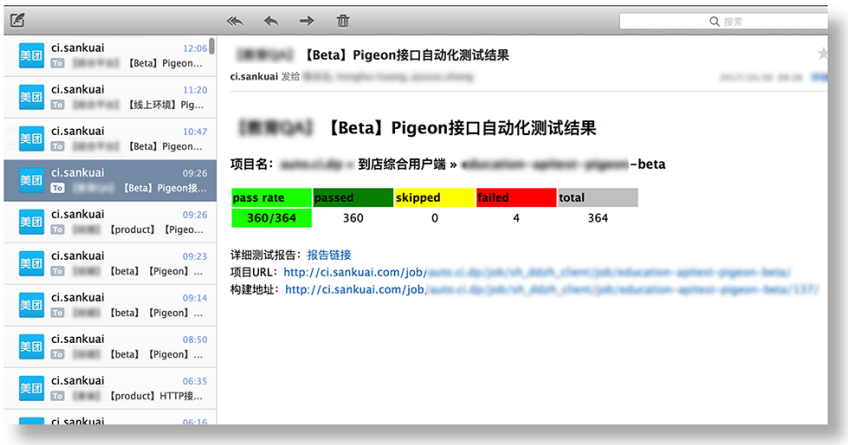
细化各个项目的功能，就是下图：



简单来说，网站部分和脚本是分离的，中间的纽带是数据库。所以，没有网站，脚本执行一点问题也没有；同样的，网站的操作，和脚本也没有关系。

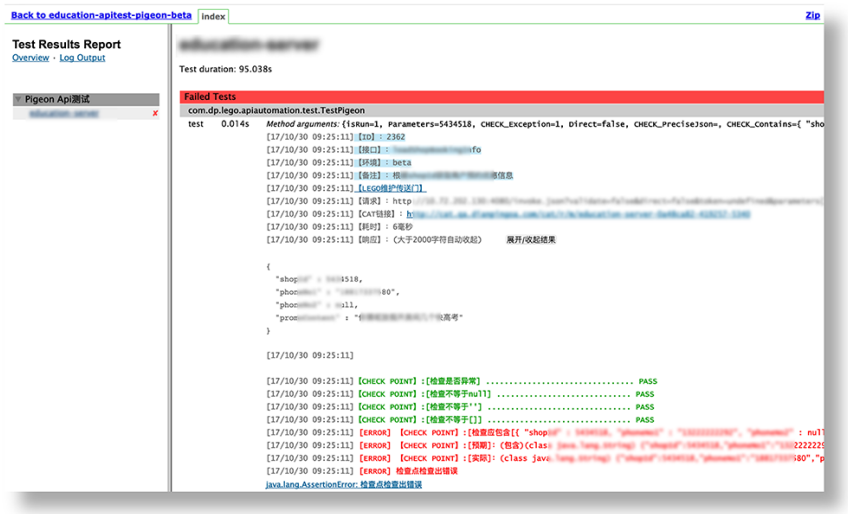
4.3 使用-日常维护

Step 1



每天上班来会收到这样的测试邮件，通过邮件能知道昨晚执行的情况。如果有报错，可以点击“详细报告链接”，跳转到在线报告。

Step 2



在现报告可以直接看到执行报错的信息，然后点击“LEGO维护传送门”，可以跳转到Lego站点上，进行用例维护。

Step 3

跳转到站点上以后，可以直接展示出该条测试用例的所有信息。定位，维护、保存，维护用例，可以点击“执行”查看维护后的执行结果，维护好后“保存”即可。

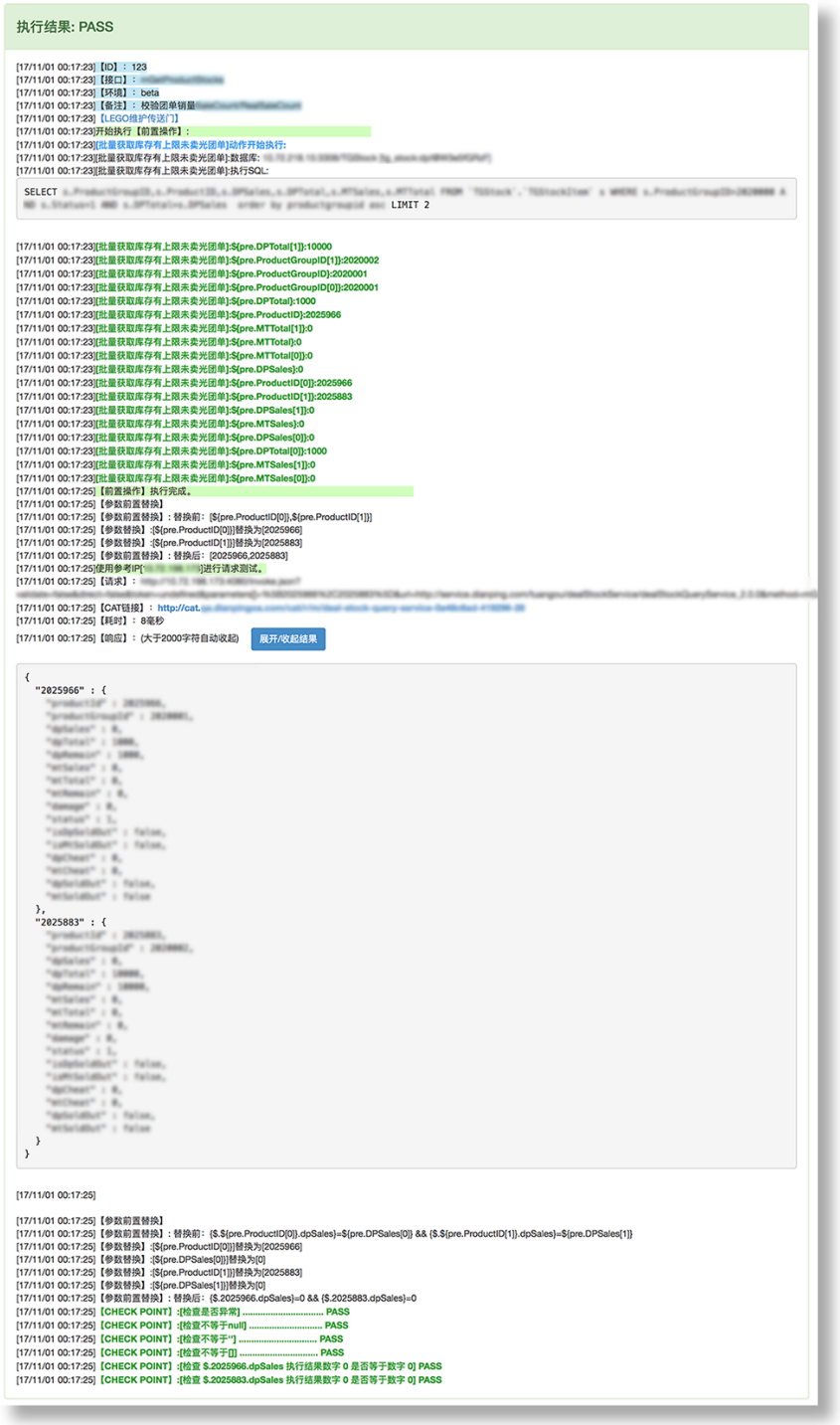
仅仅3步，1~2分钟即可完成对一条执行失败的用例进行定位、调试和维护动作。

4.4 用例编辑

通过页面，我们就可以对一条测试用例进行：

- 新建
- 复制
- 编辑
- 删除
- 是否放入每日构建中进行测试

4.5 在线调试



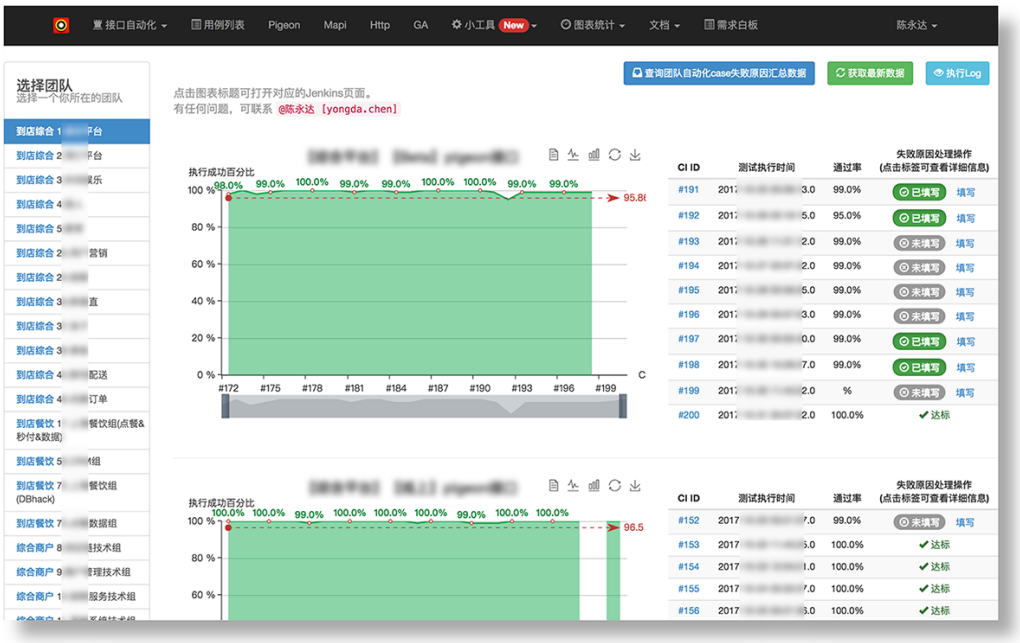
lego-web项目同样的使用base进行的用例执行，所以执行结果和打印都与脚本执行的一致的。

4.6 用例生成工具

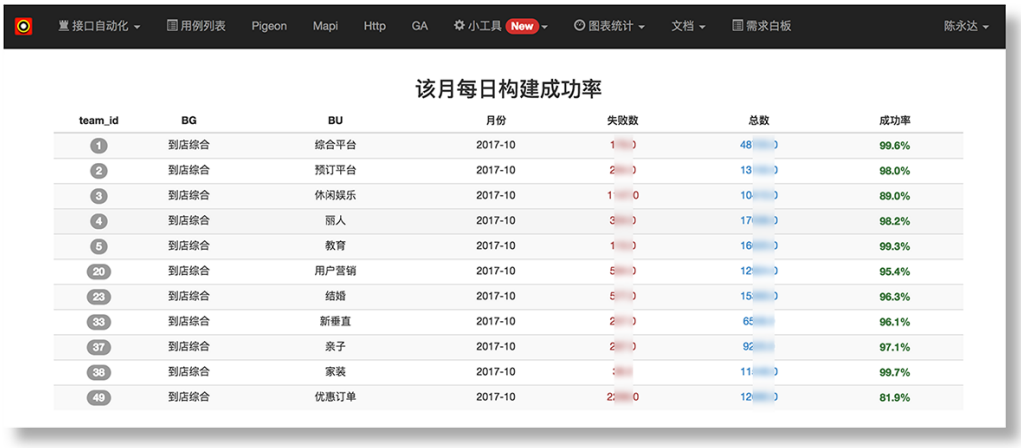
为了更方便的写用例，针对部分接口开发了一键批量生成用例的小工具。

4.7 执行结果分析

通过Jenkins接口、Base包中基础Test方法，将结果收集到数据库，便于各组对测试结果进行分析。



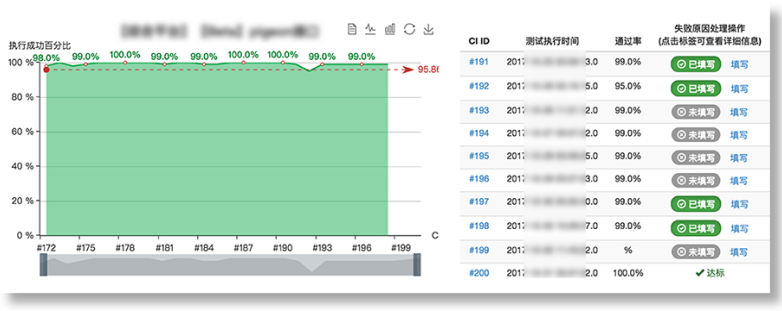
这是每天执行后成功率走势图：



也可以按月进行统计，生成统计的图表，帮助各个团队进行月报数据收集和统计。

4.8 失败原因跟踪

有了能直观看到测试结果的图表，就会想要跟踪失败原因。

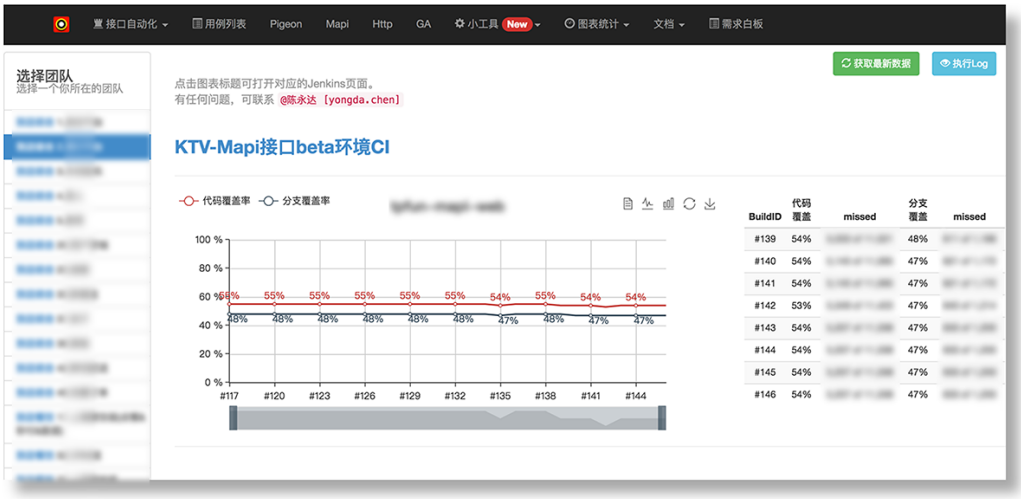


所以在成功率数据的右边，会有这样的跟踪失败原因的入口，也可以很直观地看到哪一些失败的原因还没有被跟踪。点开后可以对失败原因进行记录。

最后会有生成图表，可以很清晰地看到失败原因以及失败类型的占比。

4.9 代码覆盖率分析

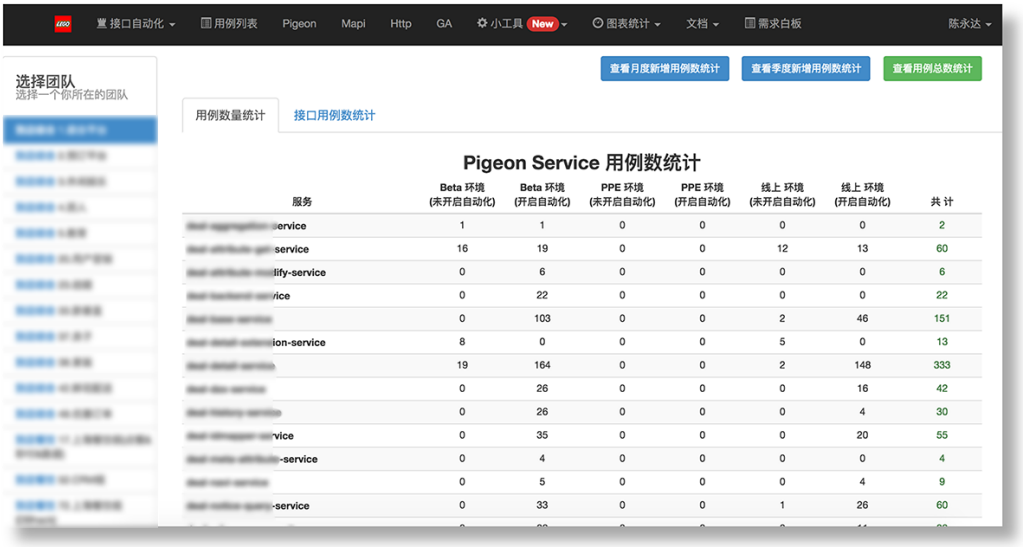
结合Jacoco，我们可以对接口自动化的代码覆盖率进行分析。



在多台Slave机器上配置Jacoco还是比较复杂的，所以可以开发覆盖率配置辅助工具来帮助测试同学，提高效率。

4.10 用例优化方向

除了上面的图表，还会给用例优化提供方向。



通过用例数量统计的图表，我们可以知道哪些服务用例还比较少，哪些环境的用例还比较少，可以比较有针对性的进行测试用例的补充。

通过失败原因的图表，我们可以改善自己用例中的“参数化”和“前后置动作”的使用，增加测试用例的健壮性。

通过线上接口调用量排序的图表。我们可以有效的知道优先维护哪些服务的测试用例，通过表格中，我们可以看到，哪些服务已经覆盖了测试用例，哪些没有被覆盖， 给各组的QA制定用例开发计划，提供参考。

同时在维护接口自动化测试的时候，都会看到用例评分的情况，来协助QA提高用例编写的质量。

4.11 收集反馈/学习

还做了“需求白板”，用来收集使用者的需求和Bug。除此之外，Lego平台已经不只是一个接口测试的平台，还可以让想学习开发的QA领任务，学习一些开发技巧，提高自己的代码能力。

五、总结

1. **为了减少开发成本**，使用比较常见的Jenkins+TestNG的脚本形式。
2. **为了简化code操作**，使用DB进行测试用例存储，并抽象出用例摸版。
3. **为了减低新建用例成本**，开发“用例维护页面”和“一键生成”等工具。
4. **为了减低维护成本**，加跳转链接，维护一条用例成本在几分钟内。
5. **为了增加用例健壮性**，设计了“参数化”、“前后置动作”等灵活的参数替换。
6. **为了易用和兼容**，统一“返回结果”类型，统一“检查点”的使用。
7. **为了接口自动化用例设计提供方向**，结合Jacoco做代码覆盖率统计，并开发相关配置工具
8. **为了便于分析数据**，从DOM、CAT、Jenkins上爬各种数据，在页面上用图表展示。
9. **为了优化用例**，提供“用例打分”、“线上调用量排行”等数据进行辅助。

将各部分像“乐高积木”一样组装在了一起，

使接口自动化在日常使用的过程中，

更灵活、高效、有趣和可靠。

本文介绍了我们的接口自动化测试平台Lego，欢迎感兴趣的同学扫描“美团技术团队”微信二维码，通过后台交流讨论。

智能支付稳定性测试实战

作者: 勋伟

> 本文根据美团高级测试开发工程师勋伟在美团第43期技术沙龙“美团金融千万级交易系统质量保障之路”的演讲整理而成。主要介绍了美团智能支付业务在稳定性方向遇到的挑战，并重点介绍QA在稳定性测试中的一些方法与实践。

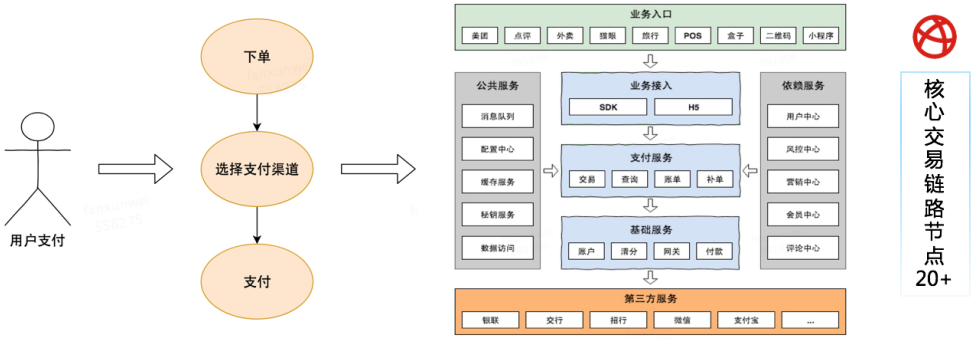
背景

美团支付承载了美团全部的交易流量，按照使用场景可以将其分为线上支付和智能支付两类业务。线上支付，支撑用户线上消费场景，处理美团所有线上交易，为团购、外卖、酒店旅游等业务线提供支付能力；智能支付，支撑用户到店消费场景，处理美团所有线下交易，通过智能POS、二维码支付、盒子支付等方式，为商家提供高效、智能化的收银解决方案。其中，智能支付作为新扩展的业务场景，去年也成为了美团增速最快的业务之一。

面临的挑战

而随着业务的快速增长，看似简单的支付动作，背后系统的复杂度却在持续提升。体现在：上层业务入口、底层支付渠道的不断丰富，微服务化背景下系统的纵向分层、服务的横向拆分，还有对外部系统（营销中心、会员中心、风控中心等）、内部基础设施（队列、缓存等）的依赖也越来越多，整条链路上的核心服务节点超过20个，业务复杂度可想而知。

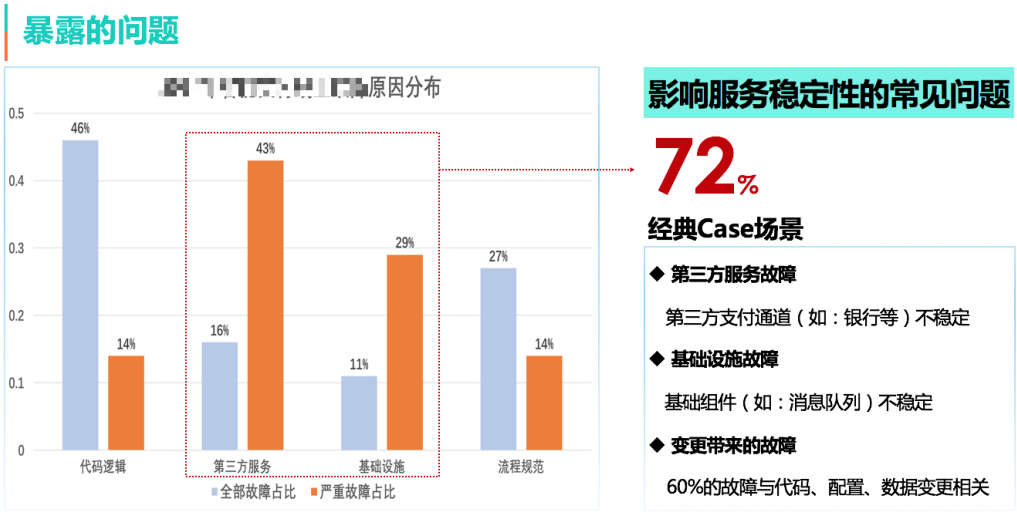
系统介绍 – 业务复杂度不断提升



此外，技术团队在短时间内就完成了从几个人到近百人规模的扩张，这也是一个潜在的不稳定因素。曾经在一段时间内，整个系统处在“牵一发而动全身”的状态，即使自身系统不做任何发版升级，也会因为一些基础设施、上下游服务的问题，业务会毫无征兆地受到影响。

痛定思痛，我们对发生过的线上问题进行复盘，分析影响服务稳定性的原因。通过数据发现，72%的严重故障集中在第三方服务和基础设施故障，对应的一些典型事故场景，比如：第三方支付通道不稳定、基

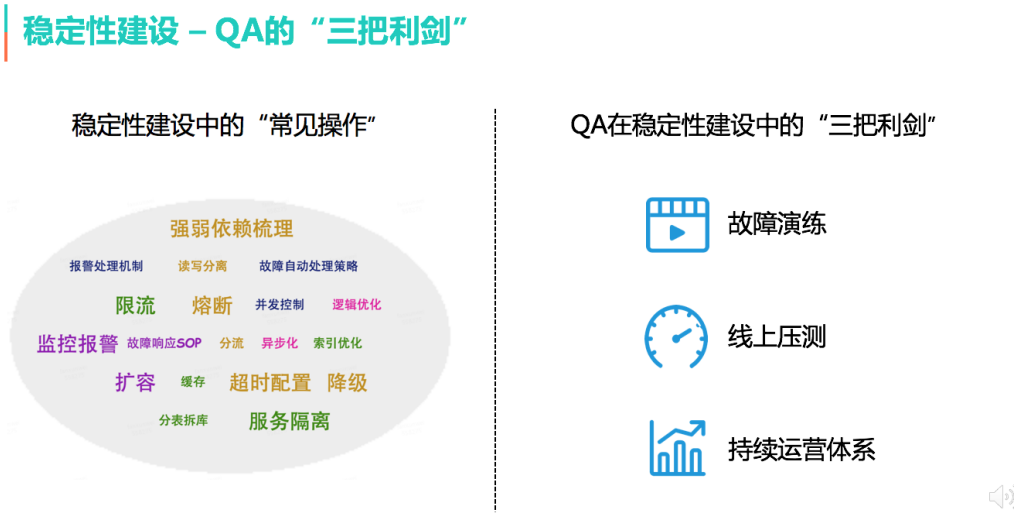
础设施（如消息队列）不稳定，进而导致整个系统雪崩，当依赖方故障恢复后，我们的业务却很难立即恢复。



解决方案

基于这些问题，我们开展了稳定性建设专项，目的很明确：提升服务的可用性。目标是逐步将系统可用性从2个9提升到3个9，再向4个9去努力。这个过程中最核心的两个策略：柔性可用，意思是尽可能保证核心功能可用，或在有损情况下尽可能保证核心用户体验，降低影响；另一个是快速恢复，即用工具或机制保证故障的快速定位和解决，降低故障修复时间。

围绕这两个策略，在稳定性建设中的常见操作：限流、熔断降级、扩容，用于打造系统的柔性可用；故障响应SOP、故障自动处理，用于故障处理时的快速恢复。而QA的工作更侧重于对这些“常见操作”进行有效性验证。基于经验，重点介绍“三把利剑”：故障演练、线上压测、持续运营体系。



故障演练的由来

举个真实的案例，在一次处理某支付通道不稳定的线上问题时，开发同学执行之前已经测试通过的预案（服务端关闭该通道，预期客户端将该支付通道的开关置灰，并会提示用户使用其他支付方式），但执行

中却发现预案无法生效（服务端操作后，客户端该支付通道仍处于开启状态）。非故障场景下预案功能正常，故障场景下却失效了。

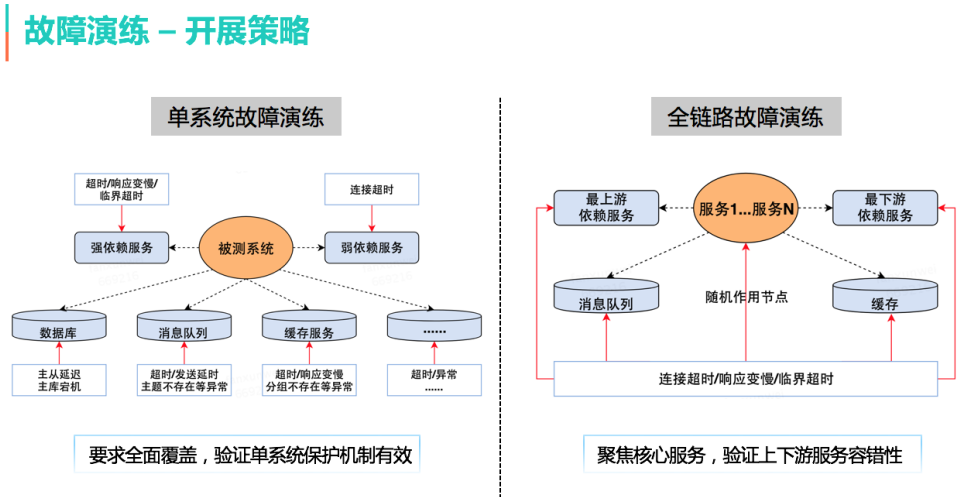
这就是故障演练的由来，我们需要尽可能还原故障场景，才能真正验证预案的有效性。

故障演练的整体方案

故障演练的整体方案，主要分为三部分：

- 负载生成模块，负责尽可能还原系统的真实运行场景（要求覆盖核心业务流程）。
- 故障注入模块，包含故障注入工具、故障样本库（涵盖外部服务、基础组件、机房、网络等各种依赖，并重点关注超时、异常两种情况）。
- 业务验证模块，结合自动化测试用例和各个监控大盘来进行。

为了更高效地开展故障演练，我们的策略是分为两个阶段进行。首先，针对单系统进行故障演练，从故障样本库出发，全面覆盖该系统所有的保护预案；在此基础上，进行全链路故障演练，聚焦核心服务故障，验证上下游服务的容错性。



故障演练的效果

事实证明，故障演练确实给我们带来了很多“惊喜”，暴露了很多隐患。这里列举三类问题：数据库主从延迟影响交易；基础设施故障时，业务未做降级；依赖服务超时设置不合理、限流策略考虑不足等。

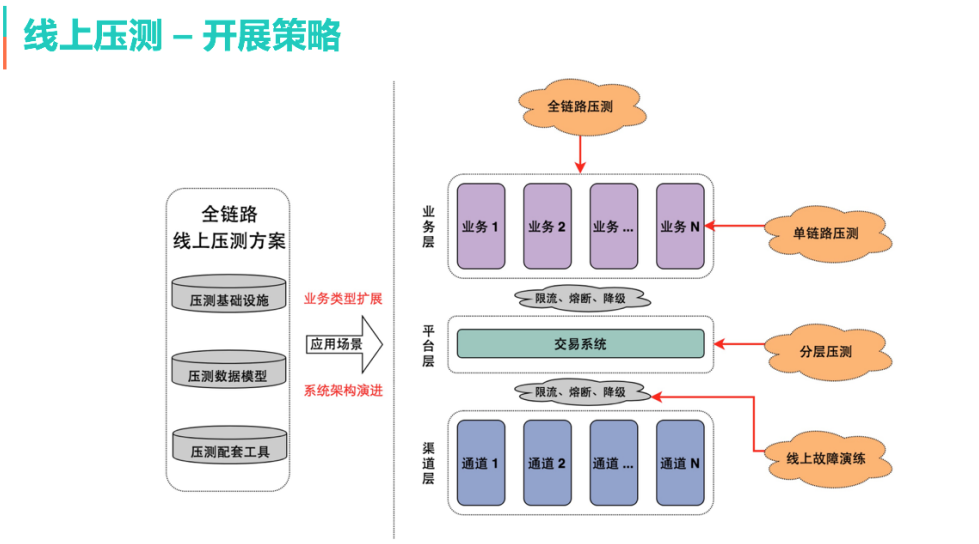
线上压测的由来

面对业务的指数级增长，我们必须对系统可承载的流量做到心中有数。对于QA来说，需要找到精准、高效的系统容量评估方法。我们碰到的难点包括：链路长、环节多、服务错综复杂，线下环境与线上差异大等等，基于测试有效性和测试成本考虑，我们决定要做线上压测，而且要实现全链路的线上压测。

线上压测的整体方案

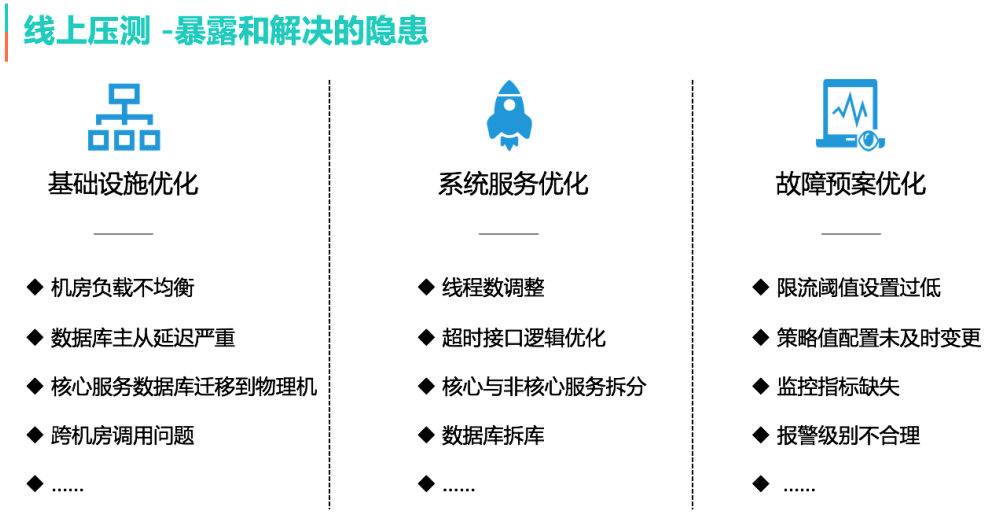
全链路压测的实现方案，与业界主流方案没有太大区别。根据压测流程，首先，场景建模，以便更真实的还原线上系统运行场景；其次，基础数据构造，应满足数据类型以及量级的要求，避免数据热点；之后，流量构建，读写流量构造或回放，同时对压测流量进行标记和脱敏；再之后，压测执行，过程中收集链路各节点的业务运行状态、资源使用情况等；最后，生成压测报告。

基于全链路线上压测方案，可以根据业务需求，灵活地进行单链路压测、分层压测等。更为重要的是，基于压测我们可以进行线上的故障演练，用于更加真实的验证系统限流、熔断等保护预案。



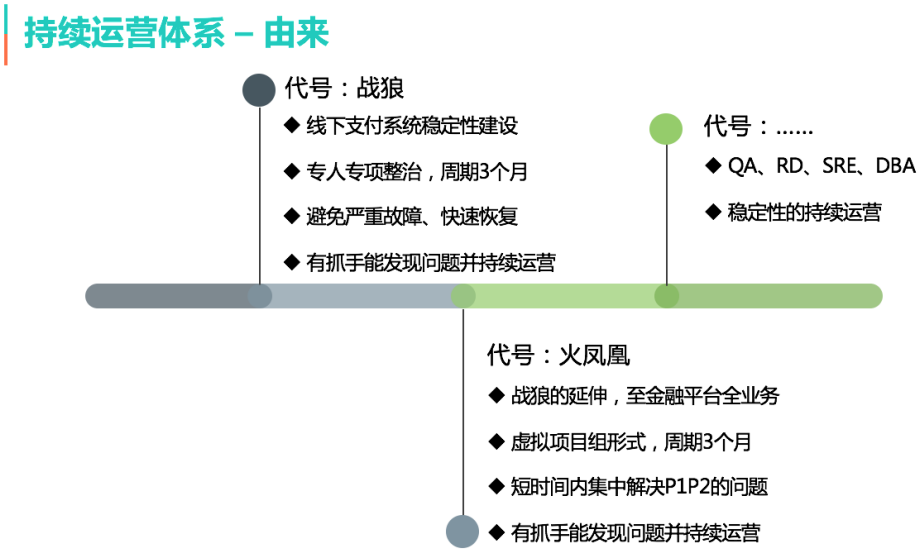
线上压测的效果

通过全链路线上压测，一方面让我们对系统容量做到心中有数，另一方面也让我们发现了线上系统运行过程中的潜在问题，而且这些问题一般都是高风险的。同样列举三类问题：基础设施优化，如机房负载不均衡、数据库主从延迟严重等；系统服务优化，如线程池配置不合理、数据库需要拆分等；故障预案优化，如限流阈值设置过低，有的甚至已经接近限流边缘而浑然不知等等。



持续运营体系的由来

智能支付的稳定性建设是作为一个专项在做，持续了近3个月的时间；在效果还不错的情况下，我们从智能支付延伸到整个金融服务平台，以虚拟项目组的方式再次运转了3个月的时间。通过项目方式，确实能集中解决现存的大部分稳定性问题，但业务在发展、系统在迭代，稳定性建设必然是一项长期的工作。于是，QA牵头SRE、DBA、RD，建立了初步的稳定性持续运营体系，并在持续完善。



持续运营体系的整体方案

下面介绍持续运营体系的三大策略：

流程规范工具化，尽可能减少人为意识因素，降低人力沟通和维护成本。

如：配置变更流程，将配置变更视同代码上线，以PR方式提交评审；代码规范检查落地到工具，尽可能将编码最佳实践抽取为规则，将人工检查演变为工具检查。

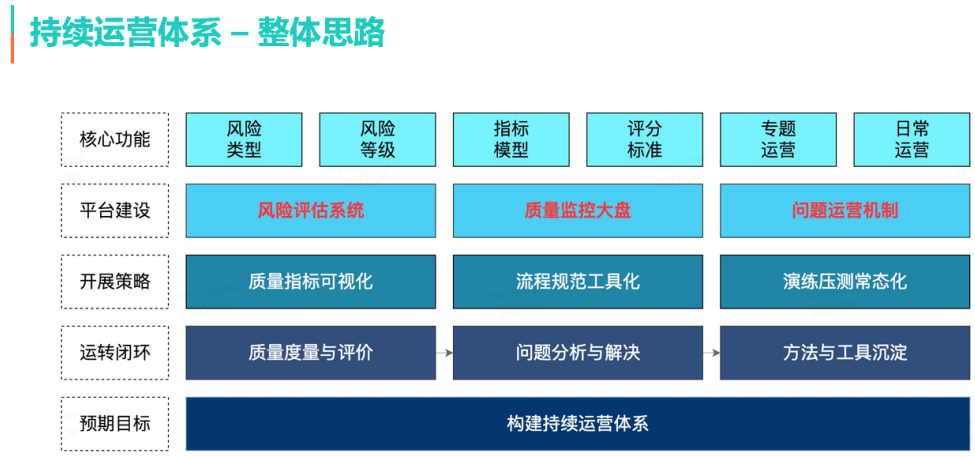
质量度量可视化，提取指标、通过数据驱动相关问题的PDCA闭环。

如：我们与SRE、DBA进行合作，将线上系统运维中与稳定性相关的指标提取出来，类似数据库慢查询次数、核心服务接口响应时长等等，并对指标数据进行实时监控，进而推进相关问题的解决。

演练压测常态化，降低演练和压测成本，具备常态化执行的能力。

如：通过自动化的触发演练报警，验证应急SOP在各团队实际执行中的效果。

基于以上三个策略，构建稳定性持续运营体系。强调闭环，从质量度量与评价、到问题分析与解决，最终完成方法与工具的沉淀；过程中，通过平台建设来落地运营数据、完善运营工具，提升运营效率。



持续运营体系的效果

简单展示当前持续运营体系的运行效果，包含风险评估、质量大盘、问题跟进以及最佳实践的沉淀等。

未来规划

综上便是智能支付QA在稳定性建设中的重点工作。对于未来工作的想法，主要有3个方向。第一，测试有效性提升，持续去扩展故障样本库、优化演练工具和压测方案；第二，持续的平台化建设，实现操作平台化、数据平台化；第三，智能化，逐步从人工运营、自动化运营到尝试智能化运营。

作者简介

- 勋伟，美团高级测试开发工程师，金融服务平台智能支付业务测试负责人，2015年加入美团点评。

招聘信息

如果你想学习互联网金融的技术体系，亲历互联网金融业务的爆发式增长，如果你想和我们一起，保证业务产品的高质量，欢迎加入美团金融工程质量组。有兴趣的同学可以发送简历到：fanxunwei#meituan.com。

大众点评App的短视频耗电量优化实战

作者: 倩云

前言

美团测试团队负责App的质量保证工作，日常除了App的功能测试以外，还会重点关注App的性能测试。现在大家对手机越来越依赖，而上面各App的耗电量，直接影响了手机的待机时间，是用户非常关心的一点。本文主要通过一个典型案例，介绍App性能测试中的电量测试，并总结了我们由此引发的一些思考。

一、案例分析

短视频作为已被市场验证的新内容传播载体，能有效增加用户停留时长。大众点评App从9.3版本开始推出短视频相关内容，在各页面新增了短视频模块。在短视频功能测试中，我们发现如果在视频列表页中播放视频，手机很快就会发烫。针对这种现象，我们马上拉取数据进行了分析，测试数据表明，视频列表页耗电量竟然是详情页的**11倍**。这是怎么回事儿呢?

目前行业内有很多电量测试的方法，我们采用的是 Battery Historian，这是Google推出的一款Android系统电量分析工具，支持5.0(API 21)及以上系统手机的电量分析。

1. 测试对象

短视频主要包括三个核心页面：视频列表页、视频详情页、作者页，本次的测试对象就是这三个页面。

2. 测试过程

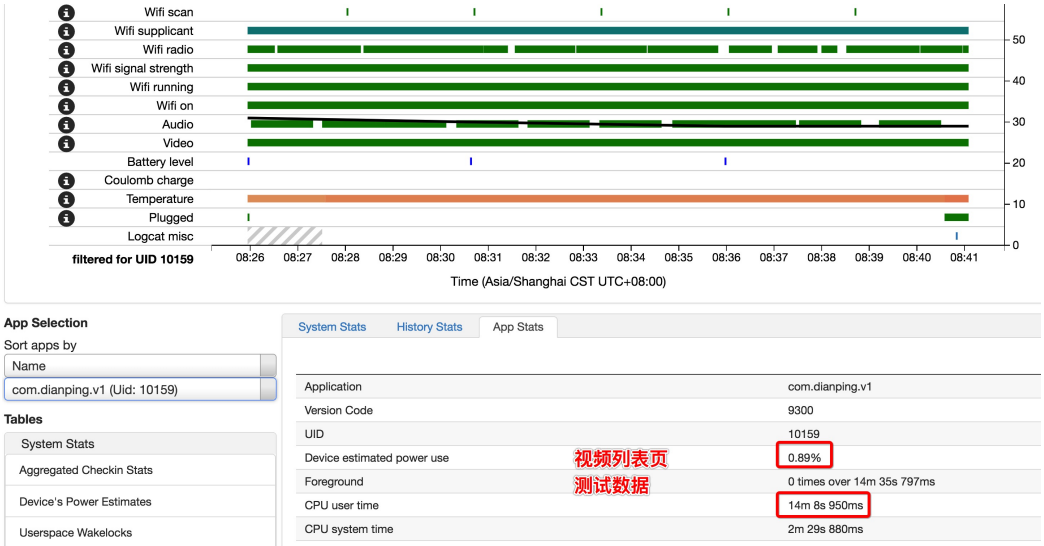
测试机型：华为Mate 9 Android 7.0 **电池容量**：4000mAh

播放的视频时长：1min15s **测试场景设计**：WiFi环境下，打开App，播放视频，通过点击“重新播放”，连

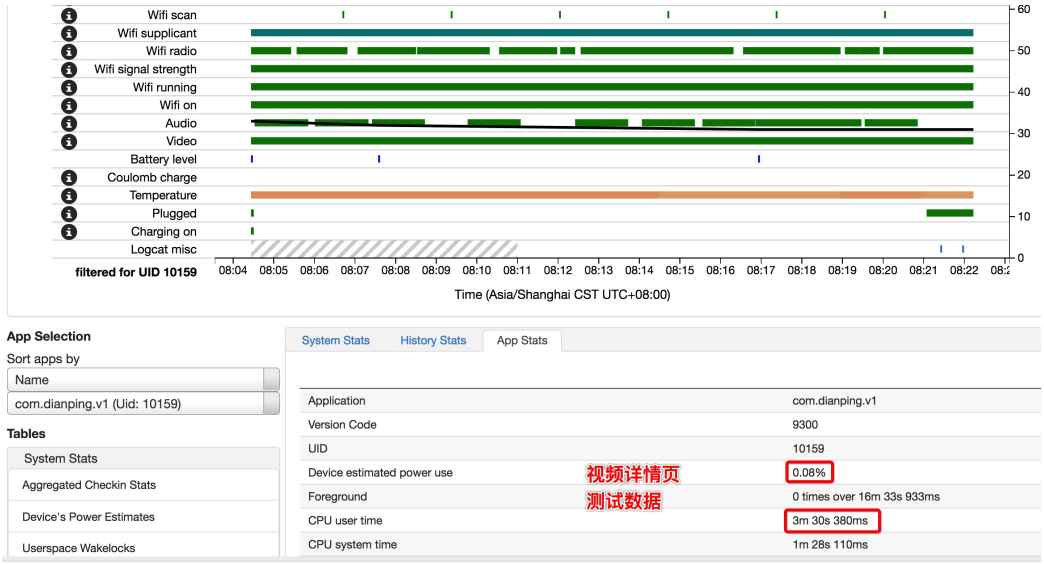
续播放10次 **对比场景**：停在App首页20min，手机不灭屏 **注意**：测试过程不充电，每次测试环境一致

3. 测试结果

如下是Battery Historian测试结果部分截图：



视频列表页



视频详情页

对测试结果数据进行汇总整理：

场景	测试页面	测试时长	消耗电量	CPU User Time
视频播放10次	视频列表页	14m35.797s	**0.89%**	**14m8.95s**
	视频详情页	16m33.933s	**0.08%**	**3m30.38s**
	作者页	17m4.332s	0.35%	10m32.36s
置于app首页	首页	20m47.26s	0.21%	6m52.23s

消耗电量：系统总电量的占比

从测试结果可以看到，短视频列表页耗电量特别高，是视频详情页的**11倍**。

4. 问题定位

视频列表页消耗电量过高，从测试数据可以很明显的看出来，视频列表页CPU占用时间高很多。从播放器布局来看，列表页和作者页比视频详情页只是多出了动画音符。如下图，红框中圈出的视频左下角的音符。

电量消耗差异这么大，是否跟动画音符有关呢。为了排除这个问题，重新编译了一个去掉动画音符的APK进行测试。测试结果：

场景	测试页面	测试时长	消耗电量	CPU User Time
视频播放10次	视频列表页 **有动画音符**	14m35.797s	**0.89%**	**14m8.95s**
视频播放10次	视频列表页 **无动画音符**	15m34.456s	**0.08%**	**3m6.42s**

从测试结果来看，CPU和耗电量很明显都下降了很多，因此确定是动画音符引起的。打开GPU视图更新的开关，查看三个页面的绘制情况。打开视频列表页，可以看到，动画音符每波动一次，会导致整个页面都在不停的绘制。如下是视频列表页绘制的情况：

从动图可以很明显看出该页面绘制十分异常，动画音符每波动一次，会导致整个页面都重新绘制一遍。

所以，到这里就明白了问题的原因，**因为页面上动画音符的实现方式有问题，动画音符波动时，导致整个页面会跟着一起不停的重新绘制。而页面的重复绘制，会使App CPU占用比正常情况下高出很多，进而导致耗电量高。**

5. 修复后验证

定位到原因之后，开发针对性的进行了修复。动画音符柱状图的实现，之前设计由多个可变化的单柱形View组成，单个柱形View重写了onMeasure & OnDraw方法，从外部柱状图View中初始化单个柱子的高度，然后自动根据一个函数式来变化高度。**因为每次都需要层层调用Measure和对应的Layout，所以造成外层控件的多次layout，进而造成CPU占用率增大**。修复之后，使用另一种方式实现，只重写了View的OnDraw方法，每次使用Canvas画出所有柱状图，使用ValueAnimator来计算变化柱状图高度，**也不再影响父控件的Layout**。如下是修复前后的核心代码：

AnimColumnView.java

```
AnimColumnView

    @Override
    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        int widthSize = MeasureSpec.getSize(widthMeasureSpec);
        int heightSize = MeasureSpec.getSize(heightMeasureSpec);
        int widthMode = MeasureSpec.getMode(widthMeasureSpec);
        if (widthMode == MeasureSpec.EXACTLY) {
            width = widthSize;
            height = heightSize;
        }
        if (mDynamicHeight == 0) {
            minHeight = height > 0 ? height / 10 : 5;
            mInterval = height / 12;
            mStartY = height;

            mInitHeight = mInitHeight > 0 ? Math.min(height, mInitHeight) : 0;
            mDynamicHeight = height - mInitHeight;

            mDelayTime = (int) (mInterval * 1000 / (mDrawFrequency * mDynamicHeight));
        }
        setMeasuredDimension(width, height);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        if (drawFlag) {
            mDynamicHeight -= mInterval;
            if (mStartY - mDynamicHeight >= height) drawFlag = false;
        } else {
            mDynamicHeight += mInterval;
            if (mStartY - mDynamicHeight <= minHeight) drawFlag = true;
        }
        postInvalidateDelayed(mDelayTime);
        canvas.drawLine( startX: 0, mStartY,  stopX: 0, mDynamicHeight, paint);
    }
}
```

修复前

VideoVisualizerView.java

```
VideoVisualizerView  onDraw()

        }

        valueAnimator = ValueAnimator.ofInt(0, 2 * varHeight);
        valueAnimator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {
            @Override
            public void onAnimationUpdate(ValueAnimator animation) {
                int animatorValue = (int) animation.getAnimatedValue();
                for (int i = 0; i < mColumnNum; i++) {
                    drawHeight[i] = fun( value: initHeight[i] + animatorValue, varHeight);
                }
                invalidate();
            }
        });
        valueAnimator.setInterpolator(new LinearInterpolator());
        valueAnimator.setDuration(mColumnDuration);
        valueAnimator.setRepeatMode(ValueAnimator.RESTART);
        valueAnimator.setRepeatCount(ValueAnimator.INFINITE);
        valueAnimator.setTarget(this);
    }

    private int fun(int value, int h) {
        int n = value / h / 2;
        return Math.abs(value - 2 * n * h - h);
    }

    @Override
    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        super.onMeasure(widthMeasureSpec, heightMeasureSpec);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        final int saveCount = canvas.getSaveCount();
        canvas.save();
        for (int i = 0; i < mColumnNum; i++) {
            canvas.drawRect( left: i * (mColumnWidth + mColumnHorizonGap), (this.getHeight() - (mColumnHeight / 4 + drawHeight[i])),
                    right: i * (mColumnWidth + mColumnHorizonGap) + mColumnWidth, this.getHeight(), mColumnPaint);
        }

        canvas.restoreToCount(saveCount);
    }
```

修复后

修复之后动画音符波动时的绘制区域：

修复之后，重新使用Battery Historian进行验证，测试结果：

场景	测试页面	测试时长	消耗电量	CPU User Time
视频播放10次	视频列表页 **修复前**	14m35.797s	**0.89%**	**14m8.95s**
视频播放10次	视频列表页 **修复后**	15m49.528s	**0.21%**	**6m12.9s**
视频播放10次	作者页 **修复前**	17m4.332s	**0.35%**	**10m32.36s**
视频播放10次	作者页 **修复后**	15m37.822s	**0.14%**	**5m40.35s**

从上面的测试结果，可以看到，视频列表页和作者页，**耗电情况得到明显的优化**。

总结一下，短视频耗电量的问题，**是由于错误的绘制方法，导致CPU占用过高，进而导致耗电量高**。那么因为动画音符导致耗电量异常的问题到这里就完美的解决了。CPU负载高，会导致耗电量高是显而易见的。但是还想深入探索一下，在手机系统各App耗电量排行榜中，耗电量是怎么计算的？还有哪些因素会影响耗电量呢？带着这些疑问，我们来看看系统计算耗电量的原理。

二、耗电量计算原理

根据物理学中的知识，功=电压*电流*时间，但是一部手机中，电压值U正常来说是不会变的，所以可以忽略，只通过电流和时间就可以表示电量。**模块电量(mAh)=模块电流(mA)*模块耗时(h)**。模块耗时比较容易理解，但是模块电流怎样获取呢，不同厂商的手机，硬件不同，是否会影响模块的电流呢。看一下系统提供的接口：./frameworks/base/core/java/com/Android/internal/os/PowerProfile.java

该类提供了public double getAveragePower(String type)接口，type可取PowerProfile中定义的常量值，包括POWER_CPU_IDLE（CPU空闲时），POWER_CPU_ACTIVE（CPU处于活动时），POWER_WIFI_ON（WiFi开启时）等各种状态。并且从接口可以看出来，每个模块的电流值，是从power_profile.xml文件取的值。PowerProfile.java只是用于读取power_profile.xml的接口而已，后者才是存储系统耗电信息的核心文件。power_profile.xml文件的存放路径是/system/framework/framework-res.apk。

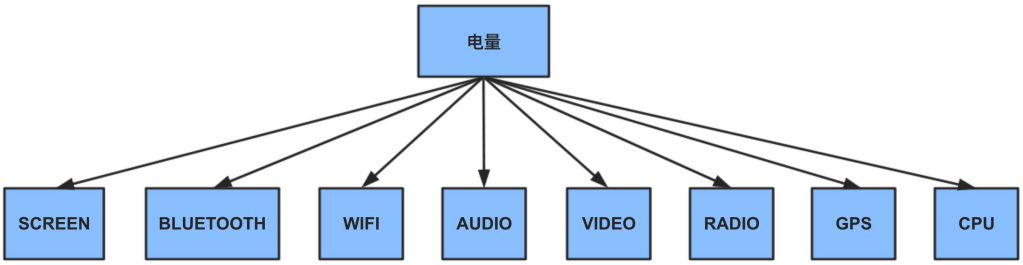
以Nexus 6P为例，在该路径获取到framework-res.apk文件。使用apktool，对framework-res.apk进行反解析，获取到手机里面的power_profile.xml文件，内容如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<device name="Android">
    <item name="none">0</item>
    <item name="screen.on">169.4278765</item>
    <item name="screen.full">79.09344216</item>
    <item name="bluetooth.active">25.2</item>
    <item name="bluetooth.on">1.7</item>
    <item name="wifi.on">21.21733311</item>
    <item name="wifi.active">98.04989804</item>
```

```
<item name="wifi.scan">129.8951166</item>
<item name="dsp.audio">26.5</item>
<item name="dsp.video">242.0</item>
<item name="gps.on">5.661105191</item>
<item name="radio.active">64.8918361</item>
<item name="radio.scanning">19.13559783</item>
<array name="radio.on">
    <value>17.52231575</value>
    <value>5.902211798</value>
    <value>6.454893079</value>
    <value>6.771166916</value>
    <value>6.725541238</value>
</array>
<array name="cpu.speeds.cluster0">
    <value>384000</value>
    <value>460800</value>
    <value>600000</value>
    <value>672000</value>
    <value>768000</value>
    <value>864000</value>
    <value>960000</value>
    <value>1248000</value>
    <value>1344000</value>
    <value>1478400</value>
    <value>1555200</value>
</array>
<array name="cpu.speeds.cluster1">
    <value>384000</value>
    <value>480000</value>
    <value>633600</value>
    <value>768000</value>
    <value>864000</value>
    <value>960000</value>
    <value>1248000</value>
    <value>1344000</value>
    <value>1440000</value>
    <value>1536000</value>
    <value>1632000</value>
    <value>1728000</value>
    <value>1824000</value>
    <value>1958400</value>
</array>
<item name="cpu.idle">0.144925583</item>
<item name="cpu.awake">9.488210416</item>
<array name="cpu.active.cluster0">
    <value>202.17</value>
    <value>211.34</value>
    <value>224.22</value>
    <value>238.72</value>
    <value>251.89</value>
    <value>263.07</value>
    <value>276.33</value>
    <value>314.40</value>
    <value>328.12</value>
    <value>369.63</value>
    <value>391.05</value>
</array>
<array name="cpu.active.cluster1">
    <value>354.95</value>
    <value>387.15</value>
    <value>442.86</value>
    <value>510.20</value>
    <value>582.65</value>
    <value>631.99</value>
    <value>812.02</value>
    <value>858.84</value>
    <value>943.23</value>
    <value>992.45</value>
    <value>1086.32</value>
    <value>1151.96</value>
    <value>1253.80</value>
    <value>1397.67</value>
</array>
<array name="cpu.clusters.cores">
    <value>4</value>
    <value>4</value>
</array>
<item name="battery.capacity">3450</item>
<array name="wifi.batchedscan">
    <value>.0003</value>
    <value>.003</value>
    <value>.03</value>
    <value>.3</value>
```

```
        <value>3</value>
    </array>
</device>
```

从文件内容中可以看到，power_profile.xml文件中，定义了消耗电量的各模块。如下图所示：



文件中定义了该手机各耗电模块在不同状态下的电流值。刚刚提到，电量只跟电流值和时间相关，所以通过这个文件，再加上模块的耗时，就可以计算出App消耗的电量，**App电量=ΣApp模块电量**。划重点，手机系统里面的电量排行，也是根据这个原理计算的。

了解原理对于平常在App耗电量的测试有很大的帮助。因为获取到手机power_profile.xml文件，就可以清楚的知道这个手机上，哪些模块会耗电，以及哪些模块在什么状态下耗电量最高。那么测试的时候，应该重点关注调用了这些模块的地方。比如App在哪些地方使用WiFi、蓝牙、GPS等等。

例如最近对比测试其他App发现，在一些特定的场景下，该App置于前台20min内，扫描了WiFi 50次，这种异常会导致App耗电量大大增加。并且反过来，当有case报App耗电量异常时，也可以从这些点去考虑，帮助定位问题。

三、电量测试方法总结

编号	测试方法	适用场景	优点	缺点
1	稳压电源+电流仪	整机电流	可以测试整机电流，并且数据精确	需要准备硬件工具，测试操作复杂，并且不能准确测试APP消耗电量
2	dumpsys batterystats	APP电量	有耗电量的详细数据	结果可读性比较差
3	系统“耗电排行”	APP电量	直观，跟用户看到的一致	没有详细的数据
4	Battery Historian	APP电量	结果直观，有耗电量的详细数据	适用于Android5.0及以上系统

如上，列出的一些常用的电量测试方法。综合各方法的优缺点，在定制个性化电量测试工具之前，目前采用的方法是Battery Historian。目前行业内，App耗电测试有很多种方案，如果仅仅测试出一个整体的电量值，对于定位问题是远远不够的。**借助Battery Historian，可以查看自设备上次充满电以来各种汇总统计信息，并且可以选择一个App查看详细信息**。所以QA的测试结果反馈从“这个版本App耗电量”高，变成“这个版本CPU占用高”“这个版本WiFi扫描异常”，可以帮助更快的定位到问题原因及解决问题。

当然，除了测试方法和测试工具，测试场景设计也非常重要。如果是在App内毫无规律的浏览，即使发现页面有问题，有很难定位到是哪个模块的问题。所以要针对性的设计场景，并且进行一些场景的对比，找出差异的地方。

四、总结

本文主要通过一个案例，介绍关于App电量测试中使用的一些基本方法和思路。电量测试采用的Battery Historian方法，虽然能初步解决问题，但是在实际的应用场景中还存在很多不足。目前美团云测平台，已经集成了电量测试方法，通过自动化操作，获取电量测试文件并进行解析，极大的提高了测试效率。目前每个版本发布之前，我们都会进行专门的电量测试，保障用户的使用体验。在电量测试方面，美团测试团队还在持续的实践和优化中。

作者简介

- 倩云，美团客户端测试开发工程师，2015年加入美团，主要负责大众点评App基础功能及Android专项测试工作。

招聘信息

点评平台技术部－平台质量中心，Base上海，主要负责大众点评平台入口和基础功能的质量保障。平台包括大众点评App、大众点评微信小程序、PC站：www.dianping.com、M站：m.dianping.com；主要业务涵盖：账号、POI、评价、视频、文章、会员社区、问答、运营活动、搜索推荐、通信链路、运营活动等基础业务。热忱期待各位QA、开发、算法人才加入点评平台技术部。联系邮箱：wanxia.wang#dianping.com。

“小众”之美——Ruby在QA自动化中的应用

作者: catfish

前言

关于测试领域的自动化，已有很多的文章做过介绍，“黑科技”也比比皆是，如通过Java字节码技术实现接口的录制，Fiddler录制内容转Python脚本，App中的插桩调试等，可见角度不同，对最佳实践的理解也不一样。这里想要阐述的是，外卖（上海）QA团队应用相对“小众”的Ruby，在资源有限的条件下实现自动化测试的一些实践与经验分享。

背景

加入外卖上海团队时，共2名QA同学，分别负责App与M站的功能测试，自动化测试停留在学习北京侧接口测试框架的阶段，实效上近乎为0，能力结构上在代码这部分是明显薄弱的。而摆在面前的问题是，回归测试的工作量较大，特别是M站渠道众多（4个渠道），移动端API的接口测试需区分多个版本，自动化测试的开展势在必行。在这样的条件下，如何快速且有效地搭建并推广自动化测试体系？在过去对自动化测试的多种尝试及实践的总结后，选择了Ruby。

Why Ruby?

简单点说就是：并不聪明的大脑加上“好逸恶劳”的思想，促使我在这些年的自动化测试实践中，不断寻找更合适的解决方案。所谓技术，其本质都是站在别人的肩膀上，肩膀的高度也决定了实现目标的快慢，而Ruby正符合所需的一些特征：

1. 效率。自身应该算是“纯粹”的测试人员，在“测试开发”这重职业并不普及的年代，一直希望有种语言可以让测试的开发效率超过研发，Ruby做到了；
2. 人性化的语法，各种糖。类似1.day.ago，简单的表达不需要解释；
3. 强大的元编程能力。基于此，DHH放弃了PHP而使用Ruby开发出了Rails，DSL也因此成为Ruby开发的框架中非常普通的特性，而这对于很多主流语言都是种奢望；
4. 对于测试来说足够充足的社区资源。不涉及科学计算，不涉及服务开发，在没有这些需求的情况下，Python和Java不再是必需。

脱离了开发语言的平台，但在不关注白盒测试的情况下并无太多不妥。当Ruby用于测试开发，基本“屏蔽”了性能上的劣势，充分展现了敏捷、易用的特点，也是选择这一技术路线的主要因素。

接口自动化框架Coral-API

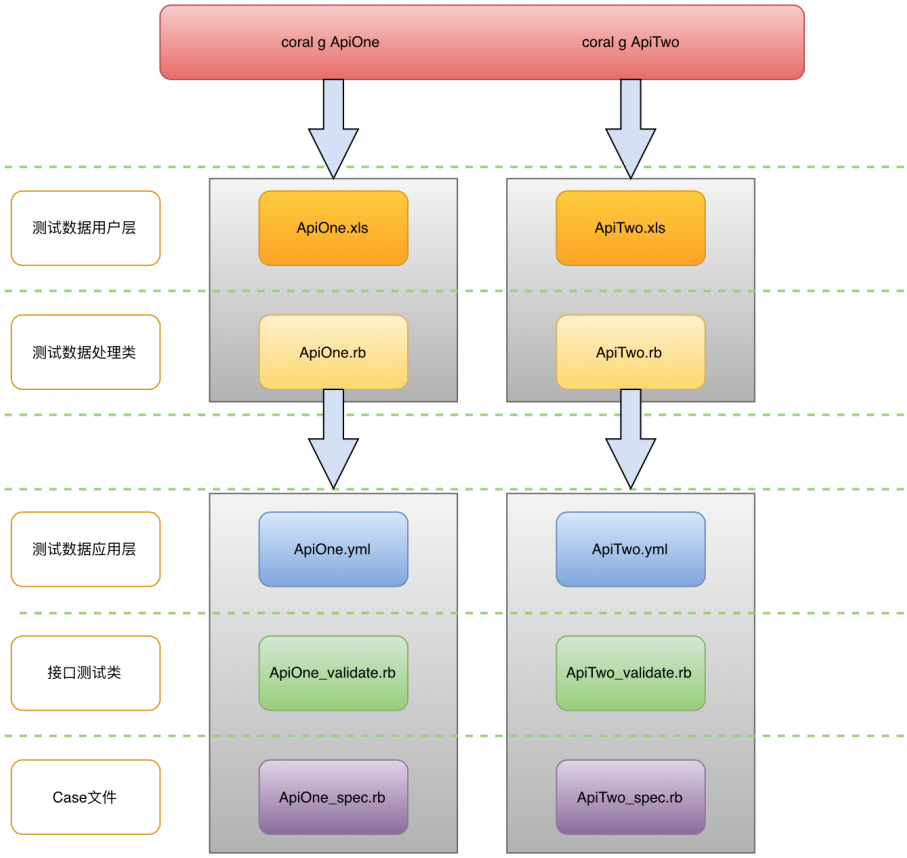
框架思路

接口自动化测试方案众多，个人认为它们都有自己的适用的范围和优缺点。UI类工具虽轻松实现无码Case，但在处理接口变动和全链路接口流程上多少会显得有些繁琐（尤其在支持数据驱动需求下），过多的规则、变量设置和编码也相差无几；录制类型的方案，更多还是适合回归，对于较全面的接口测试也

需要一定的开发量。基于这些权衡考虑，采用一种编码尽可能少、应用面更广的接口自动化框架实现方式，把它命名为Coral-API，主要有以下特点：

1. 测试数据处理独立
    - 预先生成测试所需的最终数据，区分单接口测试数据（单接口数据驱动测试）与链路测试数据
    - 通过命令行形式的语句解决了参数的多层嵌套及动态数据生成的问题
    - Excel中维护测试数据，最终转化为YML或存入DB，折中解决了JSON形式的数据难维护问题
2. 学习成本低
    - 框架提供生成通用结构代码的功能，使测试人员更关注于业务逻辑处理
    - DSL的书写风格，即便没有Ruby的语言基础，也可以较快掌握基本的接口测试用例编写
3. 扩展性
    - 支持Java平台的扩展
    - 支持HTTP/RPC接口，可根据开发框架扩展
    - 框架基于Rspec，支持多种验证方式（Build-In Matcher），及支持自定义Matcher，目前实现了JSON去噪的Diff，各种复合的条件比较

以单个接口测试编写为例，下图描述了具体流程：



coral-api框架

从图中可以看到，安装了Coral-API的gem后，可通过命令行“coral g {apiname}”，通过模板来生成测试数据XLS及对应的数据处理文件（例如ApiOne.rb文件），修改并执行ApiOne.rb文件，则可以生成最终

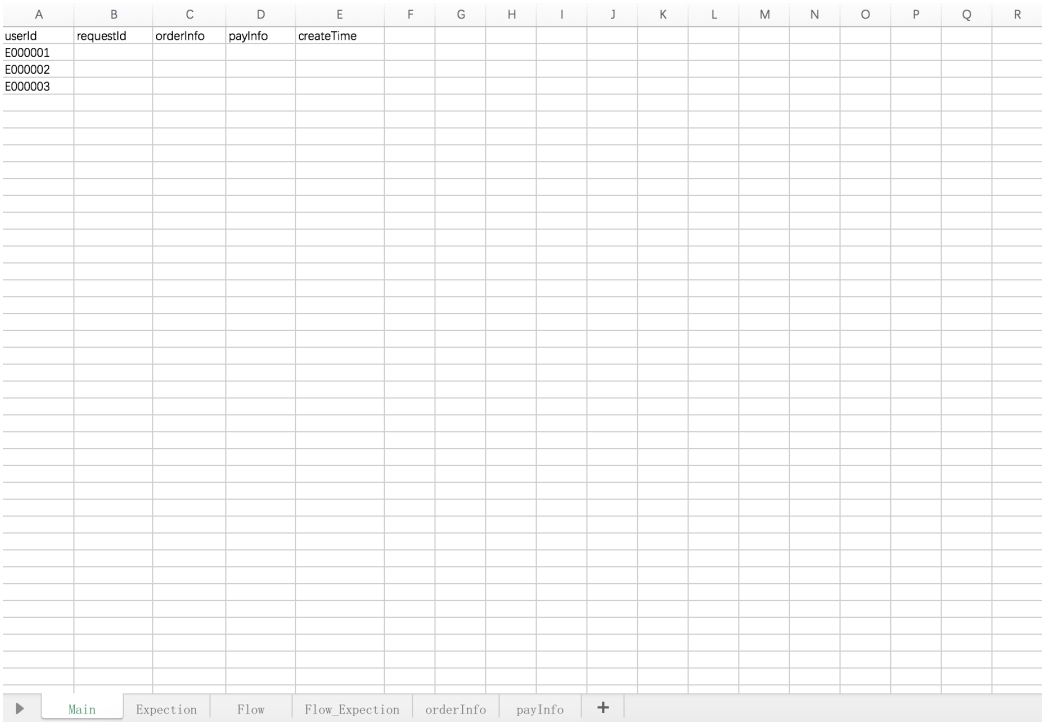
的测试数据（YML文件）及测试类和Case文件。如果开发框架支持（有途径可解析出参数），则可以通过脚本直接生成整个服务下所有接口的测试代码，实现自动化Case的同步开发。这种处理过程主要是一并解决了以下几个问题：

1. 复杂结构的测试数据构造
2. 动态参数的赋值
3. 测试数据的维护
4. 测试数据的加载

假设有以下这样一个接口请求格式，包含一个orderInfo的子节点，及payInfo的list，还需要解决一些变化值的问题，如各种id和time（暂且称为动态字段）。一般框架中会以JSON格式来作为测试用例的请求格式，在代码中按变量处理动态字段值。JSON作为请求数据的保存形式，存在一个很大的问题，就是后期维护，尤其是Case数量较多的时候。因此，考虑仍以Excel为数据维护的初始形式（使用上更直观），通过Sheet的嵌套来处理复杂结构，也便于后期接口参数变动后的Case维护。

```
userId: E000001
requestId: '1938670097'
orderInfo:
 orderId: '6778043386'
 count: '2'
 name: testgoods
payInfo:
- transactionId: '510455433082284'
 payTime: '2017-04-04 13:03:34'
 payType: BOC
- transactionId: '167338836018587'
 payTime: '2017-04-04 13:03:34'
 payType: Wallet
createTime: '2017-04-04 13:03:34'
```

测试数据的Excel做如下设计，Main中为第一层参数结构，预期响应另分一个Sheet，子节点和list节点的内容写在对应的Sheet中，动态值均置为空，在接口数据类中处理，orderInfo节点和payInfo节点均另写在新的Sheet中，用于单接口数据驱动的Case与链路回归用Case分开，当然这会增加一些Case维护的成本，可以选择是否区分。



示例的数据结构，通过以下语句即可实现，如果需要为后续接口测试提供前置步骤的数据，也可以同步实现，下例中为后续接口生成了5条请求数据。针对接口参数变动的情况，可以修改Excel和数据处理类文件，执行一遍即可，也提供了批量重新生成所有接口数据的脚本。

```
class Demo < ApiCaseBase

  update self.request,:requestId=>'gen_randcode(10)',:createTime=>'get_datetime'
  add_node self.request,"orderInfo",:orderId=>'gen_randcode(10)'
  add_list self.request,"payInfo",:transactionId=>'gen_randcode(15)',:payTime=>'get_datetime'

  sheetData={'ForApiOther'=>5}

  generate_data self,sheetData do
    update_force @data,:orderId=>'gen_randcode(10)',:createTime=>'get_datetime'
    add_node_force @data,"orderInfo",:orderId=>'gen_randcode(10)'
    add_list_force @data,"payInfo",:transactionId=>'gen_randcode(15)',:payTime=>'get_datetime'
  end
end
```

Excel作为Case的维护形式，缺点是Case较多情况下频繁读取比较影响时间。在这种情况下，考虑到把数据序列化到YML中，启动执行时接口测试类自动与测试数据进行绑定。在Case中可以直接使用形如DemoTest.request[1]的请求数据，提高了速度，结构上也清晰了不少。

接口测试类文件（HTTP接口调用为例）生成的模板如下，修改对应的接口信息即可，支持DB验证（代码块p这部分是目前唯一需要写Ruby代码的地方，当然这是非必需项）。

```
require 'apicasebase'

class PreviewTest

  include ApiTestBase

  set_cookie

  set_domain "Domain_takeaway"

  set_port 80

  set_path "/waimai/ajax/wxwallet/Preview"

  set_method "get"

  set_sql "select * from table"

  p = proc do |dbres|
    ## do something
    ## return a hash
  end

  set_p p

end
```

TestCase文件如下，原则上无需修改，只需要在测试数据的Excel中编写匹配规则及预期输出，基本上实现了单个接口无编码的数据驱动测试。

```
require 'Preview_validate'

RSpec.shared_examples "Preview Example" do |key,requestData,expData|

    it 'CaseNo'+ key.to_s + ': '+expData['memo'] do

      response = PreviewTest.response_of(key)

      expect(response).to eval("#{expData['matcher']} '#{expData['expection']}'")

    end
end

RSpec.describe "Preview接口测试",:project=>'api_m_auto',:author=>'Neil' do
```

```
  PreviewTest.request.each{|key,parameter|include_examples "Preview Example",key,PreviewTest.request[key],PreviewTest.expect
[key]}
end
```

接口流程Case编写就是各独立接口的业务逻辑串联，重点是Case的组织，把一些公用的Steps独立出shared_examples，在主流程的Case中include这些shared_examples即可，关联的上下游参数 通过全局变量来传递。

```
RSpec.describe "业务流程测试" ,:project=>'api_m_auto',:author =>'Neil' do
  let(:wm_b_client) { WmBClient.new('自配') }

  before(:context) do
    init_step
  end

  context "在线支付->商家接单->确认收货->评价" do
    include_examples "OrderAndPay Example",1
    include_examples "AcceptOrder Example"
    include_examples "CommentStep Example"
  end
end
```

通过上面的介绍，可以看到，Case的编写大部分可以通过代码生成实现（熟悉以后部分接口也可以根据需要进行操作步骤的取舍，如直接编写YML）。实践下来的情况是，从各方面一无所有，17个人日左右的时间，完成了M站API层接口自动化（业务流程9个，单个接口10个）及点评外卖移动端API的接口自动化（业务流程9个，单个接口20个），实现了外卖业务全链路接口回归，平均每个业务流Case步骤9个左右。期间也培养了一名之前未接触过Ruby的同学，在完成了第一版开发后，两名初级阶段的同学逐步承担起了框架的改进工作，实现了更多有效的验证Matcher，并支持了移动端API多版本的测试。之后的回归测试不仅时间上缩减了50%以上，也通过接口自动化3次发现了问题，其中一次API不同版本导致的Bug充分体现了自动化测试的效率。通过ci_reporter，可以方便地将Rspec的报告格式转为JUnit的XML格式，在Jenkins中做对应的展示。

Test Result : (root)

0次失败 (-1)

73个测试 (+1)
花了

所有的测试

Class	花的时间	失败	(区别)	跳过	(区别)	Pass	(区别)	总数	(区别)
AddrAdd	0.58 秒	0		0		2		2	
AddrDel	0.17 秒	0		0		1		1	
AddrEdit接口测试	0.38 秒	0		0		2		2	
AddrList接口测试	0.17 秒	0		0		1		1	
CityList接口测试	0.2 秒	0		0		1		1	
LastestAddrList接口测试	0.12 秒	0		0		1	+1	1	+1
MapSwitch接口测试	5.1 秒	0		0		1		1	
Menu接口测试	4.7 秒	0		0		1		1	
NewIndex接口测试	0.46 秒	0		0		1		1	
业务流程测试 在线支付(用券)->商家接单->确认收货->评价	10 秒	0	-1	0		10	+1	10	
业务流程测试 在线支付->取消订单	4.6 秒	0		0		5		5	
业务流程测试 在线支付->商家接单->申请退款->商家拒绝退款->确认收货->评价	16 秒	0		0		10		10	
业务流程测试 在线支付->商家接单->申请退款->放弃申请退款->确认收货->评价	15 秒	0		0		10		10	
业务流程测试 在线支付->商家接单->确认收货->评价	9.9 秒	0		0		8		8	
业务流程测试 在线支付->申请退款->商家同意退款->退款成功	15 秒	0		0		7		7	
业务流程测试 餐到付款->取消订单	11 秒	0		0		4		4	
业务流程测试 餐到付款->商家接单->确认收货->评价	8.8 秒	0		0		6		6	
添加和删除用户收货地址	0.6 秒	0		0		2		2	

测试报告jenkins展示

解决接口多版本测试的例子

移动端API自动化中存在的问题就是，一个接口会存在多个版本并存的情况，有header中内容不同的，或formdata内容不同的情况，在接口回归中必须都要照顾到，在Coral-API中我们采用以下方式进行处理。

在config.yml中定义各版本的header。

```
Domain_takeaway_header:
        v926: '{"connection":"upgrade","x-forwarded-for":"172.24.121.32, 203.76.219.234","mkunionid":"-113876624192351423","p
ragma-apptype":"com.dianping.ba.dpscope","mktunneltype":"tcp","pragma-dpid":"-113876624192351423","pragma-token":"e7c10bf5055
35bfddeba94f5c050550adbd9855686816f58f0b5ca08eed6acc6","user-agent":"MApi 1.1 (dpscope 9.4.0 appstore; iPhone 10.0.1 iPhone9,
1; a0d0)","pragma-device":"598f7d44120d0bf9eb7cf1d9774d3ac43faed266","pragma-os":"MApi 1.1 (dpscope 9.2.6 appstore; iPhone 1
0.0.1 iPhone9,1; a0d0)","mkscheme":"https","x-forwarded-for-port":"60779","X-CAT-TRACE-MODE":"true","network-type":"wifi","x-
real-ip":"203.76.219.234","pragma-newtoken":"e7c10bf505535bfddeba94f5c050550adbd9855686816f58f0b5ca08eed6acc6","pragma-appi
d":"351091731","mkoriginhost":"mobile.dianping.com","pragma-unionid":"91d9c0e21aca4170bf97ab897e5151ae0000000000040786871"}'
    v930: '{"connection":"upgrade","x-forwarded-for":"172.24.121.32, 203.76.219.234","mkunionid":"-113876624192351423","pragm
a-apptype":"com.dianping.ba.dpscope","mktunneltype":"tcp","pragma-dpid":"-113876624192351423","pragma-token":"e7c10bf505535bf
ddeba94f5c050550adbd9855686816f58f0b5ca08eed6acc6","user-agent":"MApi 1.1 (dpscope 9.4.0 appstore; iPhone 10.0.1 iPhone9,1; a
0d0)","pragma-device":"598f7d44120d0bf9eb7cf1d9774d3ac43faed266","pragma-os":"MApi 1.1 (dpscope 9.3.0 appstore; iPhone 10.0.1
iPhone9,1; a0d0)","mkscheme":"https","x-forwarded-for-port":"60779","X-CAT-TRACE-MODE":"true","network-type":"wifi","x-real-i
p":"203.76.219.234","pragma-newtoken":"e7c10bf505535bfddeba94f5c050550adbd9855686816f58f0b5ca08eed6acc6","pragma-appid":"3510
91731","mkoriginhost":"mobile.dianping.com","pragma-unionid":"91d9c0e21aca4170bf97ab897e5151ae0000000000040786871"}'
    ......
```

在接口测试类被加载时会进行全局变量赋值，同时替换header里对应节点的token，测试数据YML文件中则做这样的描述，每条数据的header则较方便地被替换。

```
---
Main:
  1: &DEFAULT
    headers: '<%= $v926 %>'
    host: mobile.51ping.com
    port: '80'
    path: "/deliveryaddresslist.ta"
    search: "?geotype=2&actuallat=31.217329&actuallng=121.415603&initiallat=31.22167778439444&initiallng=121.42671951083571"
    method: GET
    query: '{"geotype":"2","actuallat":"31.217329","actuallng":"121.415603","initiallat":"31.22167778439444","initiallng":"12
1.42671951083571"}'
    formData: "{}"
    scheme: 'http:'
  2:
    <<: *DEFAULT
    headers: '<%= $v930 %>'
  3:
    <<: *DEFAULT
    headers: '<%= $v940 %>'
  4:
    <<: *DEFAULT
    headers: '<%= $v950 %>'
  5:
    <<: *DEFAULT
    headers: '<%= $v990 %>'
```

解决RPC接口测试

HTTP接口的测试框架选择面还是比较多的，RPC调用的框架如何测试呢？答案就是JRuby + Java的反射调用，在Pigeon接口中我们已经试点了这种方式，证明是可行的，针对不同的RPC框架实现不同的Adapter（Jar文件），Coral-API传参（JSON格式）给Adapter，Adapter通过解析参数进行反射调用，这样对于框架来说无需改动，只需对部分文件模板稍作调整，也无需在Ruby中混写Java代码，实现了最少的代码量—2行。

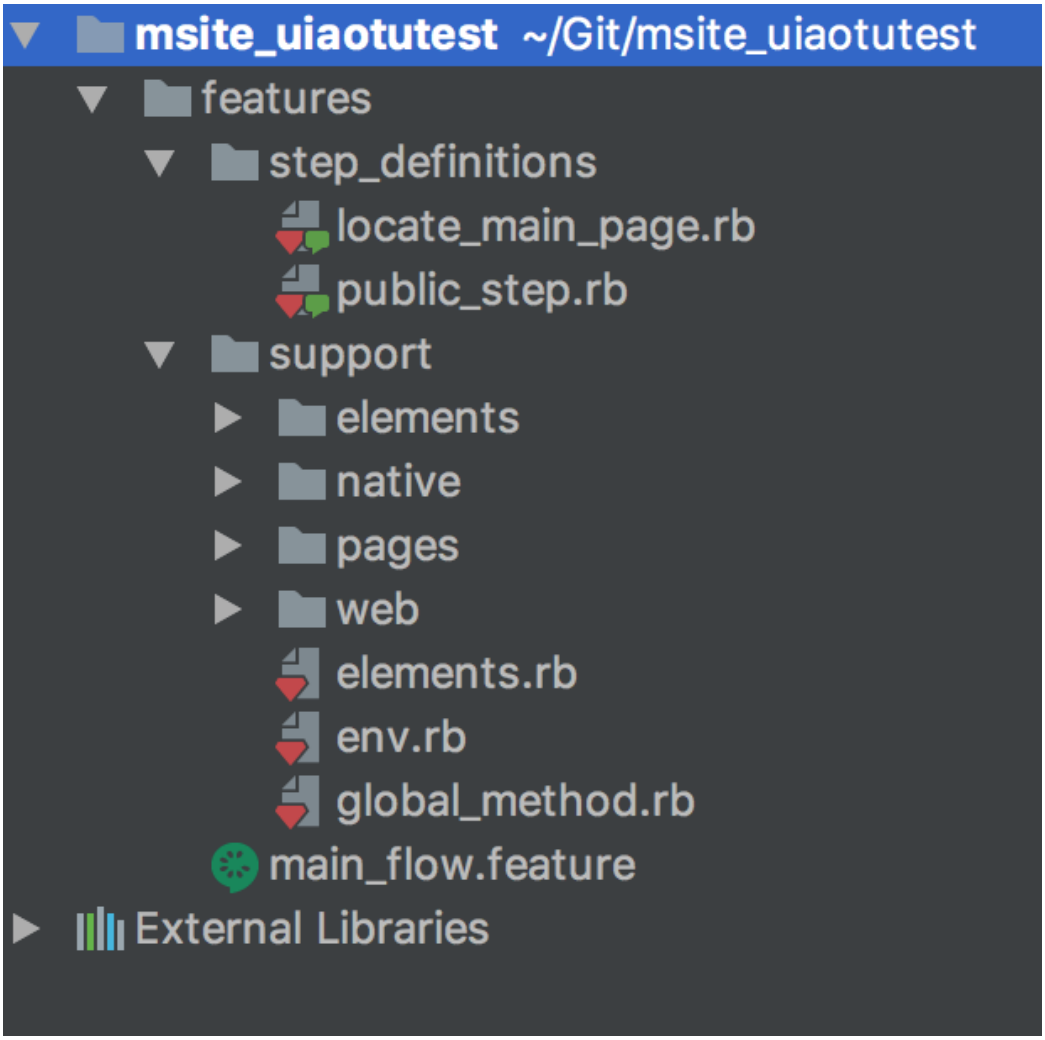
rpc调用

UI自动化框架Coral-APP

框架思想

App的UI自动化，Ruby的简便性更明显，尤其Appium提供了对Ruby良好的支持，各种UI框架的优劣就不在此赘述了。综合比较了Appium与Calabash后，选择了前者，测试框架选用了更适合业务流描述的Cucumber，沿用了以前在Web自动化中使用的对象库概念，将页面元素存储在CSV中，包括了Android与iOS的页面对象描述，满足不同系统平台的测试需要。在针对微信M站的UI自动化方案中，还需解决微信WebView的切换，及多窗口的切换问题，appium_lib都提供了较好的支持，下面介绍下结合了Appium及Cucumber的自动化框架Coral-APP。

框架结构如下图：



coral-app

step_definitions目录下为步骤实现，public_step.rb定义了一些公共步骤，比如微信测试需要用到的上下文切换，Webview里的页面切换功能，也可以通过support目录下的global_method.rb里新增的Kernel中的方法来实现。

support/native目录下为app测试的配置文件，support/web目录下为h5测试的配置文件。

support/env.rb 为启动文件，主要步骤如下：

```
$caps = Appium.load_appium_txt file: File.expand_path('../app/appium.txt', __FILE__), verbose: true

$caps[:caps].store("chromeOptions",{"androidProcess":"com.tencent.mm:tools"})

$driver = Appium::Driver.new($caps,true)

Elements.generate_all_objects

Before{$driver.start_driver}

After{$driver.quit_driver}
```

support/elements下为对象库CSV文件，内容如下图：

A	B	C	D	E	F
OBJNAME	ATTRIBUTE	ANDROID_IDENTITY	IOS_IDENTITY	MEMO	
微信我	xpath	//*[@text='我']		微信-我	
微信收藏	xpath	//*[@text='收藏']		微信-收藏	
微信收藏链接	xpath	//*[contains(@text,'takeaway')]		微信收藏-链接	
微信收藏链接URL	xpath	//*[contains(@text,'takeaway')]		微信收藏链接-URL	

对象库文件

support/elements.rb为对象库实现，将CSV中的描述转换为Elements模块中对象的功能，这样在Page中就可以直接使用类似“Elements.微信我”这样的对象描述了。

```
......

def self.define_ui_object(element)
  case $caps[:caps][:platformName].downcase
    when "android"
      idempotently_define_singleton_method(element["OBJNAME"]){$driver.find_element(:"#{element["ATTRIBUTE"]}","#{element["AN
DROID_IDENTITY"]}")}
    else
      idempotently_define_singleton_method(element["OBJNAME"]){$driver.find_element(:"#{element["ATTRIBUTE"]}","#{element["IO
S_IDENTITY"]}")}
  end
end

......
```

support/pages为Page层，实现了每个页面下的操作，目前把它实现为Kernel中的方法，采用中文命名，便于阅读使用。

```
module Kernel
  def 点击我
    Elements.微信我.click
  end

  def 点击收藏按钮
    Elements.微信收藏.click
  end

  def 点击收藏项
    Elements.微信收藏链接.click
  end
```

```
  def 点击收藏中的美团外卖链接
    Elements.微信收藏链接URL.click
  end
end
```

step里的步骤我们可以这样写，封装好足够的公共步骤或方法，Case的编写就是这么简单。

```
When /^进入美团外卖M站首页$/ do

  点击我

  点击收藏按钮

  点击收藏项

  点击收藏中的美团外卖链接

  等待 5

  step "切换到微信Webview"

  等待 15

  step "切换到美团外卖window"

end
```

最终Feature内容如下：

```
Feature: 回归下单主流程
  打开微信->进入首页->定位->进入自动化商户->下单->支付->订单详情
  Scenario:
    When 进入美团外卖M站首页
```

相对于其他的UI测试框架，使用接近自然语言的描述，提高了Case可读性，编写上也没有其他框架那么复杂。当然UI自动化中还是有一些小难点的，尤其是Hybrid应用，Appium目前还存在些对使用影响不大的Bug，在框架试用完成的情况下，将在微信入口体验优化项目结束后的进一步使用中去总结与完善。

质量工作的自动化

都知道在美团点评，QA还担负着质量控制的工作，当功能+自动化+性能+其他测试工作于一身，而且是1:8的测试开发比下，如何去关注质量的改进？答案只有：工具化、自动化。开发这样一个小系统，技术方案选择上考虑主要是效率和学习成本，符合敏捷开发的特点，基于这些因素，应用了被称为“Web开发的最佳实践”的Rails框架。

Rails的设计有些颠覆传统的编程理念，CRUD的实现上不用说了，一行命令即可，数据库层的操作，通过migration搞定，在Mail，Job等功能的实现上也非常方便，框架都有对应的模块，并且提供了大量的组件，Session、Cookie、安全密码、邮件地址校验都有对应的gem，感觉不像是在写代码，更像是在配置项目，不知不觉，一个系统雏形就完成了，整理了下项目中使用到的gem，主要有以下这些。

前端相关：

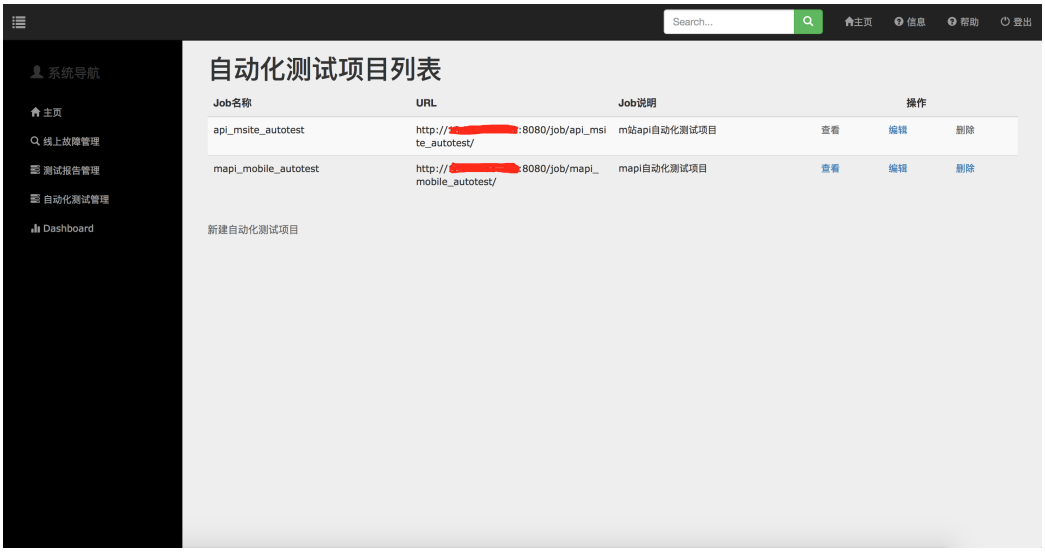
1. bootstrap-sass Bootstrap框架
2. jquery-rails jQuery框架
3. simple_form 优化的form组件
4. chartkick 堪称一行代码即可的图表组件
5. hightchart 图表组件

后端相关:

1. validates_email_format_of 邮件地址校验
2. has_secure_password 安全密码组件
3. mysql2 MySQL连接组件
4. cancancan 权限管理组件
5. sidekiq 队列中间件
6. sidekiq-cron 定时Job组件
7. rest-client Http And Rest Client For Ruby
8. will_paginate 分页组件

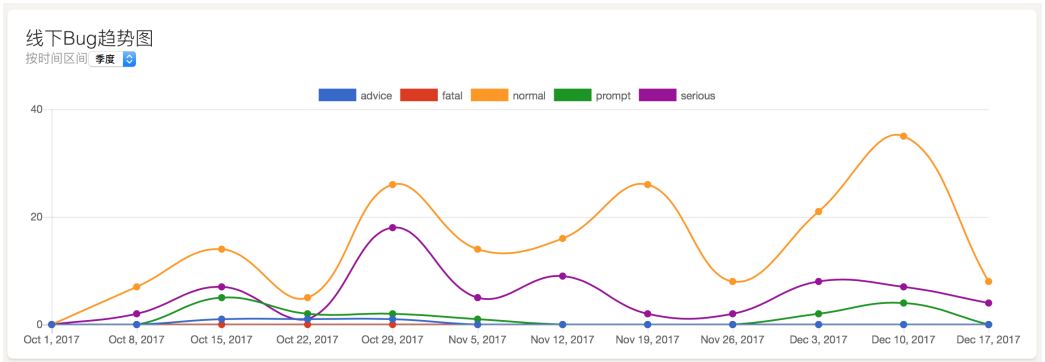
从搭建开发环境、写Demo，自己做产品、开发、测试、搭建生产环境、部署，边参阅文档边实现，总共18个人日左右，实现了平台基础功能、线上故障问题的管理及通知、测试报告的管理及通知、Sonar数据的抽取（Job及邮件）、Bug数据的抽取（Job）、自动化测试项目的接入、质量数据的Dashboard各类数据图表展示等功能，以下为系统功能的两个示例:

后台管理界面



shwmqp manager

线下缺陷周趋势



shwmqp manager

应用Rails，团队较快进入了可以通过数据进行质量分析的初级阶段，当然还有很长的路要走，在从0到1的这个过程中，还是较多地体会到了敏捷开发的特性，也充分感受到了DRY理念。

总结

以上为半年左右时间内，外卖上海QA团队在自动化工作上的一些实践，总的来说，达到一定预期效果，整理这篇文章分享一些心得。所谓的主流与小众并非绝对，主要从几个方面衡量：

1. 应用领域。Ruby因为性能问题，始终不太主流，但并不意味着它一无是处，用在测试领域，开发效率、DSL的友好性、语言的粘合性、使用者的学习低成本，都能发挥很大的优势。
2. 使用群体。不同的使用群体对于技能掌握的要求也是不同的，能达到同样效果甚至超过预期则就可以选择哪怕“小众”的方案。
3. 环境背景。其实有很多初创公司选择Ruby作为初期的技术栈有一定的道理，而这与我们当初的情景有相似之处，实际效果也体现了语言的特性。

当然应用“小众”技术，必然要面对不少挑战：如何迅速培养能掌握相关技术的同学，与其他语言平台的衔接问题，面对团队的质疑等。尤其Ruby属于易学难精的那种，从脚本语言应用层次上升到动态语言设计层次还是需要一定的学习曲线的，也就是说对于使用者来说是简单的，对于设计者的能力要求较高，就像流传的Ruby程序员的进阶过程就是魔法师的养成史。

正因为有特色的技术，才值得去研究和学习，就像它的设计者所说，目的就是为了让开发人员觉得编程是件快乐的事情。做了这么些年的测试，还能够不停止写代码的脚步，也是因为几年前开始接触Ruby。不论将来是否成为主流，它仍然是测试领域工具语言的不错选择，不管以后会出现什么样的技术，选型的标准也不会改变。技术的世界没有主流与小众，只有理解正确与否，应用得当与否。

招聘信息

最后插播一条广告，美团外卖上海研发中心长期招聘前端、客户端、后端、QA及数据、算法相关的工程师，欢迎有兴趣的同学发送简历到huangzhuolin02@meituan.com。

美团点评云真机平台实践

作者: 东初 李帅

背景

随着美团点评业务越来越多，研发团队越来越庞大，对测试手机的需求显著增长。这对公司来说是一笔不小的开支，但现有测试手机资源分配不均，利用率也非常有限，导致各个团队开发、测试过程中都很难做到多机型覆盖。怎么样合理、高效利用这些测试手机资源，是摆在我们面前的一道难题。

现有的方案

为了解决这些问题，业内也出现了一些手机管理和在线调试使用的工具或平台，比较常见的有:

- OpenSTF
- 百度MTC的远程真机调试
- Testin的云真机
- 腾讯WeTest的云真机
- 阿里MQC的远程真机租用

其中OpenSTF是开源项目，其他的平台大多也都是基于OpenSTF原理实现的。因此，我们对OpenSTF项目进行了深入研究。

遇到的问题

我们首先按照OpenSTF官方的方案进行了搭建，并进行了小规模的应用，但渐渐的我们发现了它的一些问题:

- 模块过多而且耦合紧密，解耦难度较大，每次修改需要更新所有模块，难以快速迭代开发。
- 部分技术选型落后。由于OpenSTF出现的时间比较早，部分技术已经落后于目前的主流。例如OpenSTF前端选用AngularJS 1.0进行开发，在生态链方面已经落后于其他流行的框架；数据库方面选用非关系型数据库RethinkDB，在数据计算和性能方面弱于MySQL等关系型数据库，同时RethinkDB资料较少，不便于开发与维护。
- OpenSTF屏幕图像传输采用图片单张传输的方式进行，而且画质不能由用户来调节，实际应用中占用带宽很高，在网络比较差的情况下会有严重的卡顿现象，体验很差。

我们的方案

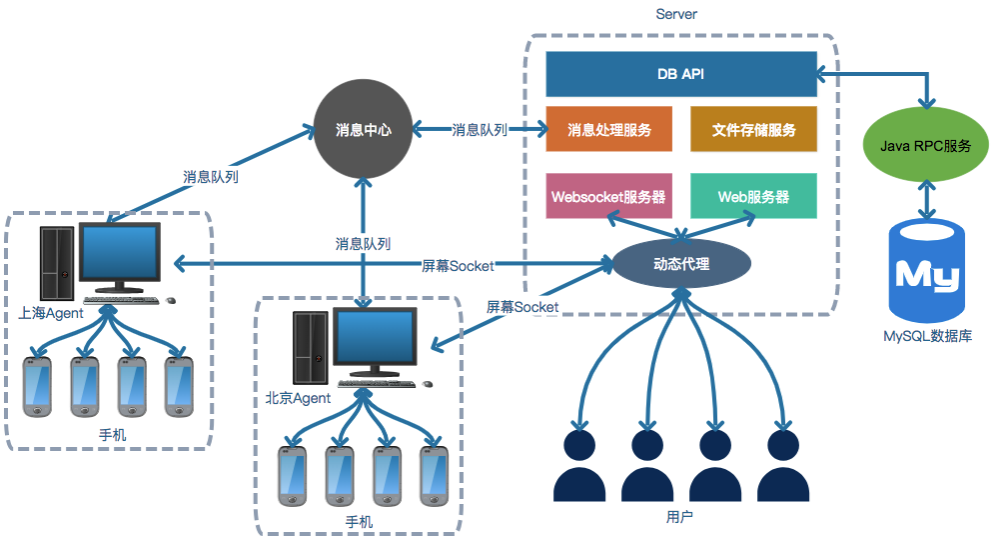
架构设计

根据业务场景的需要，并吸取了OpenSTF结构优点，我们采用Agent/Server模型的模块化设计方案。下面分别介绍主要模块的功能:

- Agent模块。Agent模块与OpenSTF的provider类似，部署在服务器上或者用户的电脑上，Agent连接真实的手机，并且将手机的屏幕图像通过Websocket动态代理到Websocket服务器上，然后通过消息中心来进行消息的传递。

- Server模块。Server用来集中管理和调度手机，与OpenSTF结构不同的是，我们的Server端包含Web服务器、Websocket服务器、动态代理以及消息处理服务等部分，Server将用户的访问动态代理到对应的Web服务器和Websocket服务器上，并通过消息处理模块向消息中心传递消息，实现用户与Agent端手机的交互。
- 数据存储模块。数据存储模块用来保存整个平台的数据，例如手机的状态、用户使用记录等。数据存储模块由MySQL数据库和一个RPC服务组成，Server不再直接读写数据库，而是通过一个RPC服务来进行数据的读写操作。
- 消息中心。与OpenSTF的triproxy功能类似，是连接手机和Server的枢纽，消息中心主要处理屏幕的操作以及手机的状态变更等消息。

通过模块化设计，项目结构比OpenSTF更加清晰。下面是整个系统的设计图：



架构的优势

Agent模块我们直接复用了OpenSTF的provider大部分功能，包括minicap、minitouch等。在此基础上，我们也扩展了一部分OpenSTF缺失的功能，比如：

- 在provider的基础上进行了二次开发，使其支持画质/帧率调节（下文会有详细说明）。
- 加入健康检测功能，检测手机网络是否正常、是否设置了网络代理等。
- 加入Inspector功能，方便获取UI控件树（下文会有详细说明）。
- 对Agent进行了版本区分，便于Web端根据不同的Agent版本对相应的功能展示和隐藏。

在Server模块中，我们引入了动态代理的机制，并且重新开发了Web部分，采用了Vue 2.0 + iView来实现，数据库采用了MySQL。

相比OpenSTF原生架构，总结下来有以下优势：

- 模块耦合程度低，开发和部署更方便。OpenSTF各个功能模块不仅数量多而且代码耦合紧密，在此基础上进行二次开发和部署非常困难。而我们将整个项目分为Server、Agent、消息中心、数据存储四个模块，四个模块功能和代码都是独立的，基本上没有耦合关系，支持快速迭代开发，部署也很方便。
- 前端框架更主流，开发和维护成本低。OpenSTF前端是使用AngularJS 1.0实现的，AngularJS 1.0已经处于废弃阶段，各种第三方组件基本已经停止支持，AngularJS 2.0的社区和生态并未成熟，而我们采用了Vue 2.0前端框架，Vue 2.0相对已经成熟，在美团侧也已经有大量应用，能够快速开发高质量的Web功能。
- 数据库性能强，设计灵活、维护方便。OpenSTF使用RethikDB作为数据库，RethikDB是一个NoSQL型数据库，它有非常多的缺点，比如处理大量数据时的性能很差，资料非常匮乏，排查问题和数据库维护都非常困难。而我们采用了

MySQL数据库，很好的解决了这些问题，并且实现了Server模块与数据读写的分离，这样在更新数据库表结构的时候无须同时修改Server端的代码，只要保证RPC服务的接口格式一致即可，开发和维护更加方便。

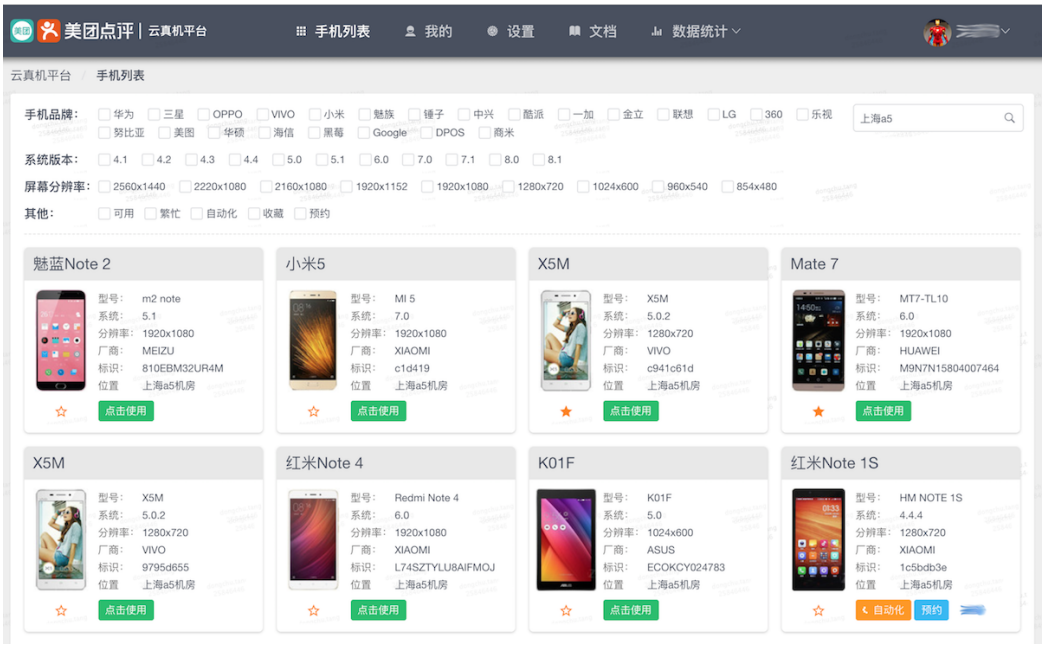
除了这些基础的功能之外，我们还开发了一些特色的功能，下面我们来详细介绍。

特色功能

与客户端自动化相结合

为了合理、高效利用测试手机资源，我们与客户端自动化进行了结合，主要有两个方面：

- 与集团内部的云测平台深度融合。在云测平台的服务器节点上部署Agent代码，为云测平台自动化任务创建者提供自动化过程展示和远程调试功能，同时将云测平台空闲的手机开放给更多人使用。
- 开放API。我们开放了一些API给普通用户，供用户查询手机状态、占用手机、连接adb调试等，用户可以使用脚本调用API，然后直接在平台的手机上进行自动化测试。



预约功能

当一台手机处于繁忙状态时，用户必须要等待手机空闲后才能使用，由于手机空闲时间不确定，就会给用户带来很大的不便。为了解决这个问题，我们开发了手机预约的功能，用户可以预约处于繁忙状态的手机，当手机空闲时，自动帮预约用户占用15分钟，并通过即时通讯工具通知预约人。

画质调节

远程调试平台的核心是实时获取屏幕图像，由于屏幕传输需要比较大的网络带宽，在网络不佳的情况下就会出现卡顿现象。因此，我们针对不同的网络做了一些流畅度的优化，下面来介绍一下其中的细节。

屏幕获取的原理是通过minicap来高速截图，每秒最高可达60张，然后将这些截图显示在Web上。因此，我们考虑从两个方面来优化网络带宽的占用，**第一个是调节截图的质量**，minicap本身支持调节画质（OpenSTF固定设置了80%的压缩比），关键代码如下：

```
var rate =  Number(match[6])
    if (rate > 2 && rate < 100) {
      log.info(rate)
      if (rate > 30) {
        options.screenJpegQuality = 80
      }else if (rate > 15) {
        options.screenJpegQuality = 50
      }else {
        options.screenJpegQuality = 20
      }

      frameProducer.restart()
      framerate = rate
    }
```

第二个是调节每秒发送的图片张数，也就是帧率，我们可以在Agent端控制发送图片的数量，关键代码如下：

```
# 首先修改帧率发送部分：
function wsFrameNotifier(frame) {
  if (latesenttime == 0 || Date.now()-latesenttime > 1000/framerate) {
    latesenttime = Date.now()
    return send(frame, {
      binary: true
    })
  }
}
# 再加入调整帧率的代码：
case 'rate':
  var rate =  Number(match[6])
  if (rate > 2 && rate < 100) {
    framerate = rate
  }
  break
```

那么，帧率和图片压缩比调节到多少才能满足不同网络环境的需要呢？我们先来看一组数据：

图片压缩比	图片尺寸
100%	69.82KB
80%	46.78KB
50%	41.87KB
20%	37.84KB
10%	35.84KB

表中是使用minicap做的图片压缩实验，从数据中我们可以看到当图片质量降低到80%时图片大小降低比较明显，而图片质量并没有明显的下降。继续降低图片质量，图片大小降低有限。我们再来看另外一组数据：

帧率	图片压缩比	实际流量
60	100%	4.02M/S
60	80%	2.74M/S
60	50%	2.41M/S
30	80%	1.43M/S

30	50%	1.22M/S
30	20%	1.10M/S
15	50%	0.63M/S
15	20%	0.55M/S
15	10%	0.52M/S

表中是各种帧率和压缩比组合产生的实际流量数据。

从数据中我们可以看到最高帧率和压缩比的组合下，流量达到了4M/S，而80%压缩比时流量减小到了2.7M/S，降低非常明显。考虑到实际网络情况，我们将**60帧、80%压缩作为了高画质选项**。

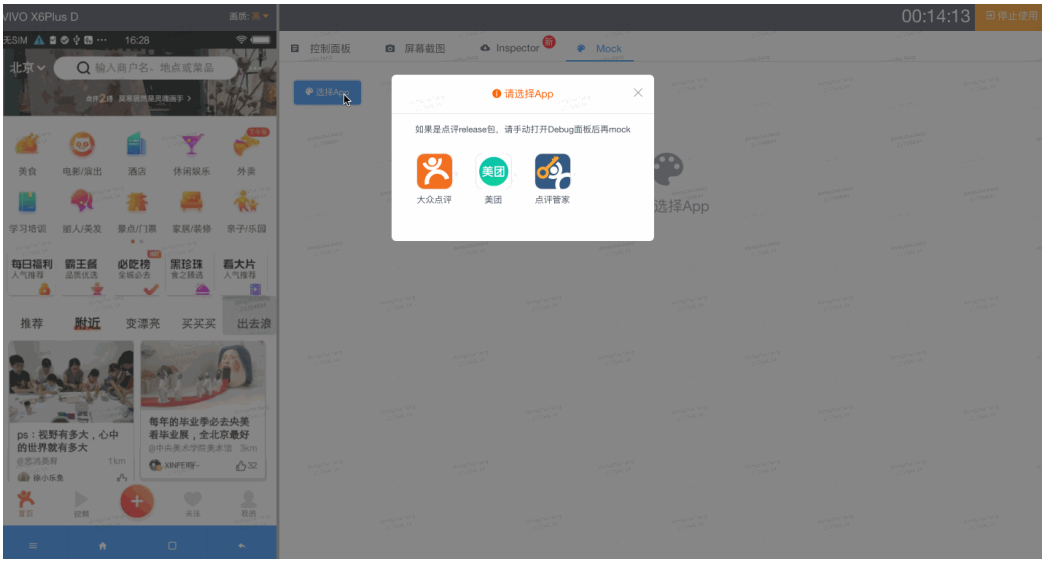
而图片质量从80%降低到50%时图片大小下降并不明显，此时降低帧率就成了很好的选择。当帧率降低到30帧时流量降低了一半，1.2M/S的流量能够满足大部分网络状况使用，30帧也能保证操作的流畅度，于是**30帧、50%压缩比成为了中画质的选项**。

低画质主要是为了保证在较差的网络环境能够正常使用，500K/S的流量是红线。我们将**15帧、20%压缩比作为低画质选项**，此时图片质量和帧率较低，但能够保证基本的使用体验。

除了通过降低图片质量和帧率来减小手机屏幕图像传输的流量外，将图像使用H264等编码压缩成视频传输也是一种有效降低流量的办法，相对于图片，图像的压缩率将会更高，用户的操作体验也会更好。但是图像压缩编码原理比较复杂，相关技术我们还在研究当中。

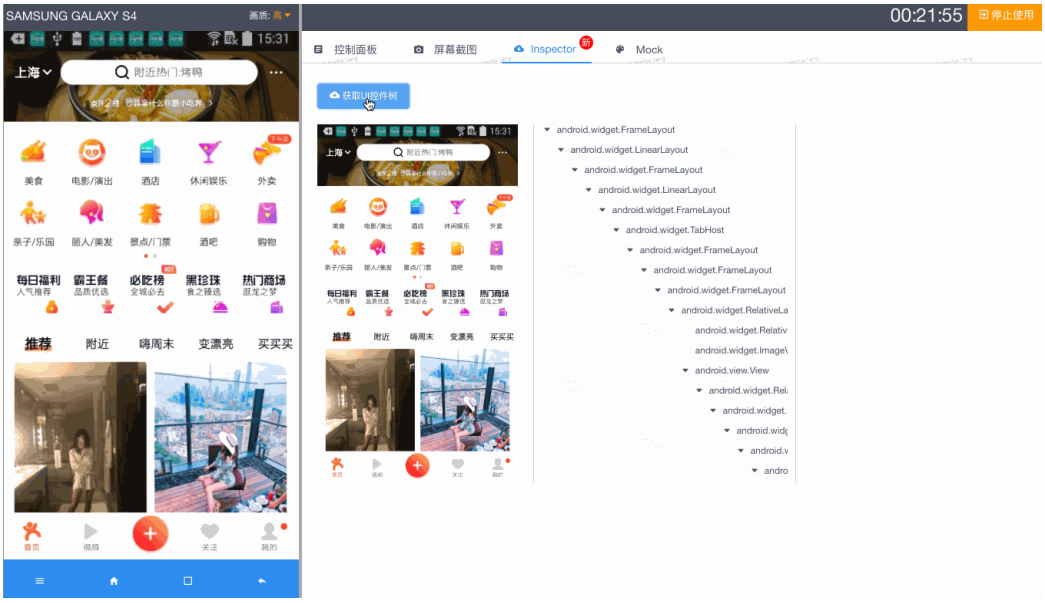
App Mock

在做App测试过程中经常需要抓包，一般情况下，我们通过修改WiFi的代理然后用抓包工具就可以实现。但是这样做的效率比较低，多个工具切换也非常不便。借助集团的Mock平台，我们开发了一键Mock功能，能够快速完成相应App的Mock操作。带来的好处是我们可以一边操作App，一边查看App发出的请求，大大提高了测试的效率。



App Inspector

App Inspector功能可以让用户在平台上使用真机的同时查看页面控件树及页面元素，并且支持Xpath，更加方便高效的查找页面元素，给UI自动化测试提供了很大便利。



这个功能我们是借助Uiautomator实现的。基本原理是写一个Uiautomator用例，用来获取当前页面的Hierarchy，然后将用例打包成一个JAR放到Agent端。当在Web端触发获取控件树时，Agent将JAR包推送到手机上并运行，此时会在手机端生成一个XML文件。然后再使用cat命令获取XML内容并在前端解析。用例核心代码如下：

```
public class launch extends UiAutomatorTestCase {

    public void testDumpHierarchy() throws UiObjectNotFoundException {
        File file = new File("/data/local/tmp/local/tmp/uidump.xml");
        UiDevice uiDevice = getUiDevice();
        String filename = "uidump.xml";
        uiDevice.dumpWindowHierarchy(filename);
    } }
```

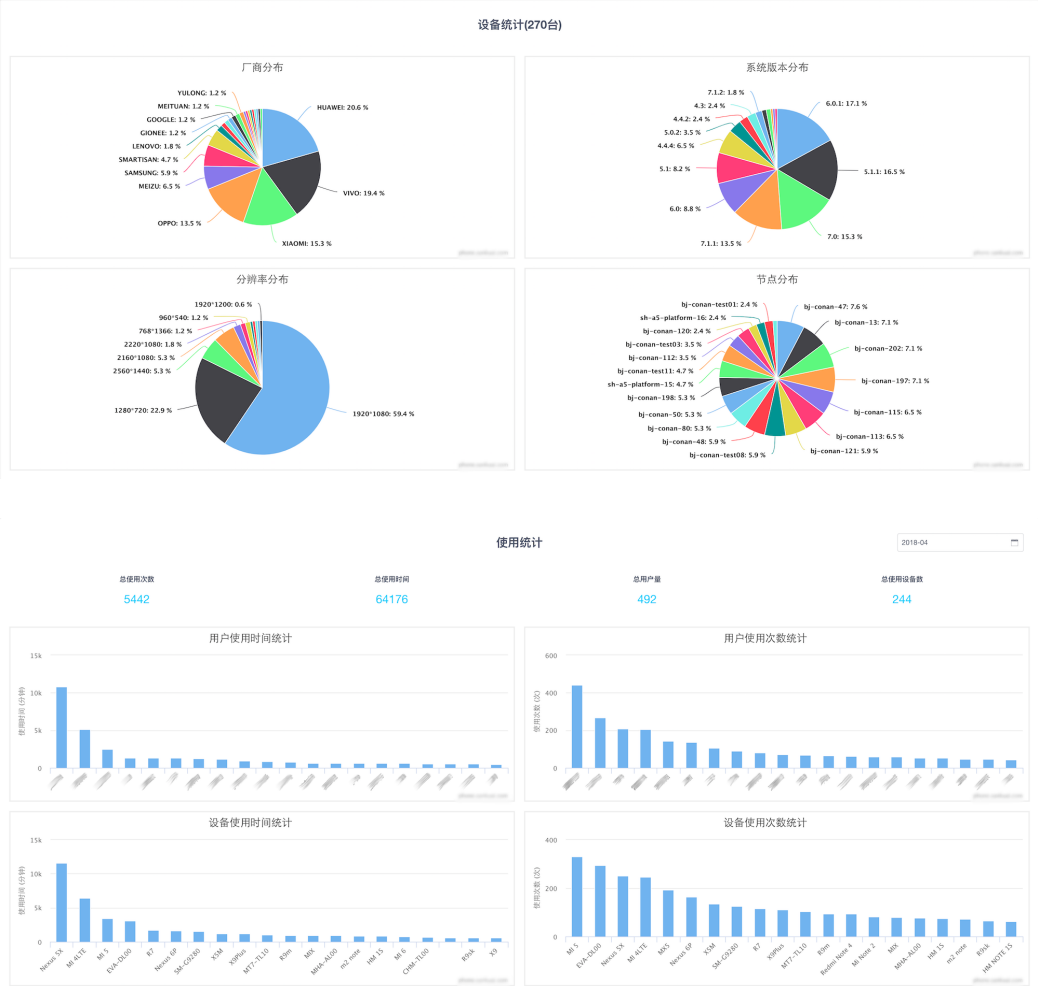
当然，你也可以用adb命令来获取Hierarchy：

```
adb shell uiautomator dump /data/local/tmp/uidump.xml
```

但这种方式不能获取动态页面，比如视频播放页面。

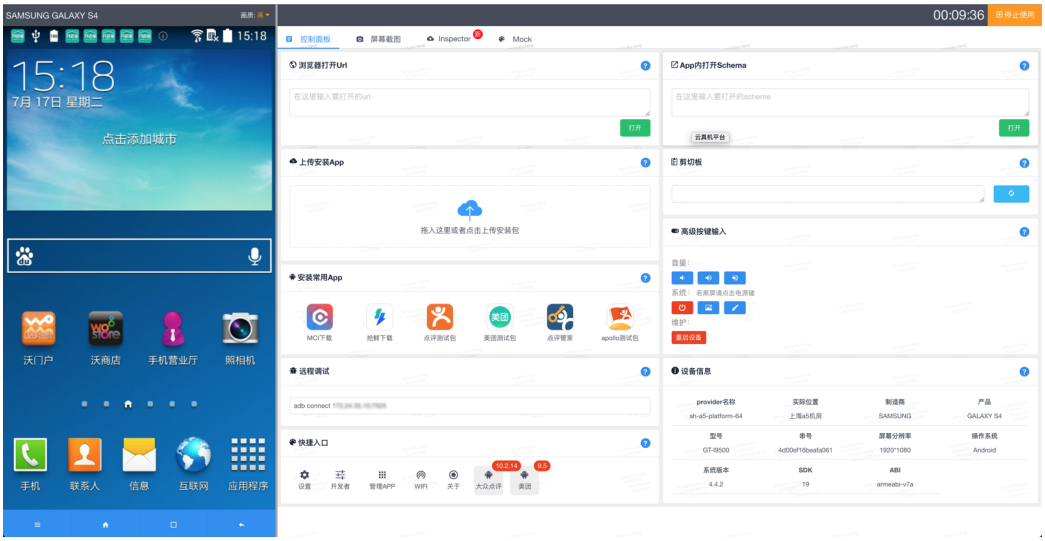
数据报表

为了更好的了解平台的运营情况，我们做了详细的数据统计，主要从使用次数、使用时间、设备数量、使用分布等方面进行统计。目前我们管理的手机近300台，平均每个月有超过500名研发人员在使用我们的平台，每天的使用次数达到280次，每天总使用时长超过60小时。



其他小功能

除了以上几个比较大的功能点，我们也做了一些贴心的小功能，比如：检测手机网络是否设置代理、检测手机已安装的应用版本及安装时间、快速安装最新版本的测试包、支持App内的Schema跳转等等。这些小功能为研发人员节省了很多时间，提升了他们的工作效率。



未来规划

iOS手机支持

目前，云真机平台只支持Android手机，对iOS手机的支持正在进行中。我们已经初步完成了主要功能的开发，预计很快将与大家见面。

产品优化

我们计划继续扩展产品功能，比如支持Log日志展示和性能数据采集等。目前云真机平台已经在美团点评内部平稳运行超过两年，我们会继续不断迭代版本、打磨产品，提供更好的使用体验。

作者简介

- 东初，大众点评平台质量工具组负责人。7年互联网行业测试、开发经验，2015年加入原大众点评。先后主导了云真机平台、单元测试平台、web安全实验平台等项目的开发，致力于用工具来提升研发团队的工作效率。
- 李帅，大众点评高级测试开发工程师。2015年加入原大众点评，主要负责云真机平台的开发以及客户端底层组件的测试。热衷于钻研测试领域的前沿技术，并推动了多项新技术落地。

招聘信息

点评平台－平台质量中心，Base上海，主要负责大众点评平台入口和基础功能的质量保障。平台包括大众点评App、大众点评微信小程序、PC站：www.dianping.com、 M站：m.dianping.com；主要业务涵盖：账号、POI、评价、视频、文章、会员社区、问答、运营活动、搜索推荐、通信链路、运营活动等基础业务。热忱期待各位QA、开发、算法人才加入点评平台技术部。联系邮箱：wenjie.pan#dianping.com。

质量运营在智能支付业务测试中的初步实践

作者: 勋伟

背景

毋庸置疑，质量是决定产品能否成功、企业能否持续发展的关键因素之一。对于“质量时代”下的互联网企业，如何在快速迭代的节奏中兼顾质量，真正落地“人人重视质量、人人创造质量、人人享受质量”，这是对QA的要求，也是整个产品技术团队面临的重要挑战。

质量运营，是将运营的思路注入到质量评估与改进工作中，它着眼于产品的全生命周期，以质量为中心，以数据为驱动，通过建设持续迭代的质量保障体系，最终提升交付质量。本文将聚焦研发过程中的提测阶段，以改进提测质量为例，从方案制定、策略应用、效果评估等几个方面，介绍质量运营在智能支付业务中的初步应用。

挑战

美团点评智能支付承担了整个公司所有线下交易，当前日交易量已经突破1000万单，是公司继外卖和摩拜之后，又一个千万级日订单的业务。业务高速增长、团队快速扩张的情况下，质量问题极易被放大化，如果不能及时得到处理，后续解决成本会越来越高。

存在的痛点

刚参与智能支付业务测试时遇到的几个问题，如下：

1. 缺陷严重级别高，提测打回时常发生，如：核心功能未实现或实现与需求不符。
2. 缺陷数量多，定位、修复、回归耗时长，如：越在上游引入的缺陷，修复的成本就越高，潜在的风险也越大。
3. 各类低级缺陷，团队彼此间的信任度降低，如：文案错误、变量引用错误等编码大意导致的低级缺陷。

解决的难点

在尝试去改善时发现难以推动的几个问题，如下：

1. 对暴露出的质量问题，如何更直观的在认知上达成一致？如：收到过很多类似的问题反馈：“xx的缺陷太多了，质量意识差”、“xx项目存在很大问题，需要尽快改进”，即使是基于事实得出的判断，这种偏主观的表达方式，对问题达成一致的认知带来很大的困难。
2. 对已公认的质量问题，如何更快速的进行分析和定位？如：缺陷发生在测试阶段，但缺陷的引入可能是在需求阶段、设计阶段、开发阶段；某个时间段内的异常质量数据，可能是A项目或B项目，可能是C团队或D团队。问题类型细分、数据钻取能力等等，在问题的快速分析和定位中至关重要。
3. 对已定位的质量问题，如何找到可以落地的改进措施？如：项目总结中常常会见到类似这样的描述：“加强自测”、“严格遵守项目流程”、“文档需要写的更详细些”，这种偏“形容词”的改进措施，很难实际操作，这也是整个质量改进过程中最大的一个难点。

思路

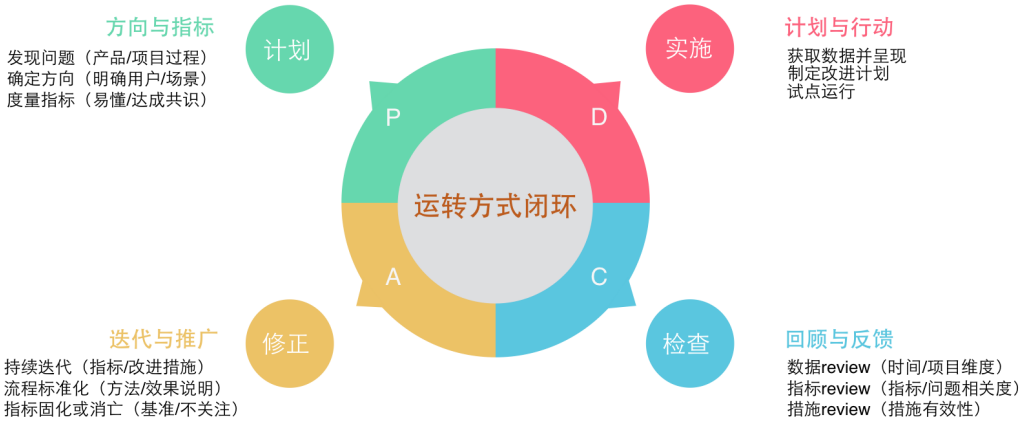
质量改进是一个持续迭代的过程，不可急于求成。按照质量运营的方法，基本思路为：分析痛点，找到抓手，持续运营，形成闭环。

基于提测阶段的质量问题特征，从痛点和难点中寻找突破点。大致思路为：

1. 目标应达成一致：质量改进的目标是QA的KPI，并应该与关键干系人在愿景、目标上达成一致。
2. 问题的客观呈现：提取核心度量指标，通过有效数据和典型案例说话。
3. 数据的灵活钻取：尽可能全的提供各种维度的数据，并分层级展示。
4. 改进措施可落地：对措施的多方Review、流程标准化到工具化的演进。

解决方案

解决方案的重中之重，是务必遵循PDCA来实现运转方式的闭环。具体如下：



目标/度量/方法/工具的持续迭代

确定问题与方向

通过痛点描述可知，缺陷是反映智能支付业务当前提测质量的最显著特征。提测质量的进一步分析，可通过**缺陷的数量、缺陷的严重程度、缺陷的生成原因**三个方向来展开。

缺陷数量

缺陷数量	具体说明
缺陷总数	指定统计规则下缺陷的数量，仅包含项目过程中提交的缺陷

缺陷的严重程度

严重级别	具体说明	使用范围
致命–Blocker	影响核心功能的缺陷	缺陷导致核心业务流程不可用，或产生较大影响
严重–Critical	造成较大影响的功能性缺陷	缺陷导致核心业务流程受影响，或导致非核心业务流程不可用
一般–Major	影响较小的功能性缺陷	缺陷导致非核心业务流程受影响，或导致用户体验类的问题

提示–Minor	非功能性缺陷	如：不影响正常功能的UI错误、无重大歧义的提示错误等
建议–Trivial	优化建议	非严格意义上的缺陷，一些可优化的点

缺陷的生成原因

生成原因	具体说明
实现与文档不符	RD实现与需求文档描述不一致
需求缺陷	需求文档中缺少相应描述；需求变更
技术方案考虑不足	前后端接口定义不一致；对边界、异常场景考虑不全等
环境问题	被测服务不稳定；服务器或测试设备配置等引起的问题
第三方依赖	依赖的外部系统引入的问题，如用户中心等
兼容性	不同设备上出现的功能或展示异常类的问题
性能问题	服务端性能：响应时间过长、CPU过高、GC频繁、没有分页、没有缓存等； 客户端功耗：包大小、冷启动时间、流量、内存泄漏OOM、加载时间、耗电量
安全问题	XSS注入，SQL注入等
Bugfix引入	由于修改Bug引入的缺陷
不是缺陷	无效Bug；不能复现

度量指标及标准

针对缺陷的三个分析方向，提取出可度量的指标、定义合理的标准值，并与整个团队达成一致。

指标提取策略

1. 典型性
   - 找最想要解决的问题。不追求全面，只针对Top问题提取指标，如：生成原因里最应该避免的是哪些。
   - 找最能反映问题的指标。如：缺陷数量有众多度量指标（新增数量、人均数量等等），为排除工作量影响，我们选择用千行代码缺陷率这个指标。
2. 有效性
   - 除非对绝对数量有明确要求，否则尽量使用百分比作为度量指标。
   - 指标数据的计算方式，要求简单易懂，并务必得到相关人的认可。

标准制定策略

1. 基于公司统一要求，如：Sonar千行代码严重问题数，统一标准为低于0.1。
2. 基于公司各业务现状，如：缺陷相关指标按照公司各业务部门排行，取Top5的值作为标准线。
3. 基于自身业务阶段持续调整，如：随着智能支付业务质量的持续改进，定义更严格的质量标准。

最终定义的指标与标准

度量指标	指标说明	标准值
千行代码缺陷率	（缺陷总数/代码行数）* 1000	移动端：< 0.45 前端：< 0.2 后端：< 0.15
Sonar千行代码严重问题数	（Sonar严重问题数/代码行数）* 1000	Blocker：0 总数：< 0.1
严重缺陷占比	严重缺陷总数/缺陷总数	< 3.5%
需求缺陷占比	需求缺陷总数/缺陷总数	< 10%
实现与文档不符缺陷占比	实现与文档不符缺陷总数/缺陷总数	< 10%

获取数据并展示

基于Metrics（美团点评工程质量中心提供的度量平台），能够快速的获取数据并展示。但要注意，部分指标的计算需要对Metrics提供的数据进行二次处理，以保证数据的精准性。如：在计算千行代码缺陷率时，需要排除掉开发自测缺陷等。

对于数据的展示形式，除了利用Metrics提供的各种图表外，最为关键的是要实现数据与问题（相关缺陷）的可关联，以便进行下一步分析。如下图所示（通过超链接方式进行关联）：

业务线	time	千行代码Bug率	总缺陷数	总代码行
智能支付	2018-第10周	0.144	31	215452
xxx	2018-第10周	0.056	8	141933

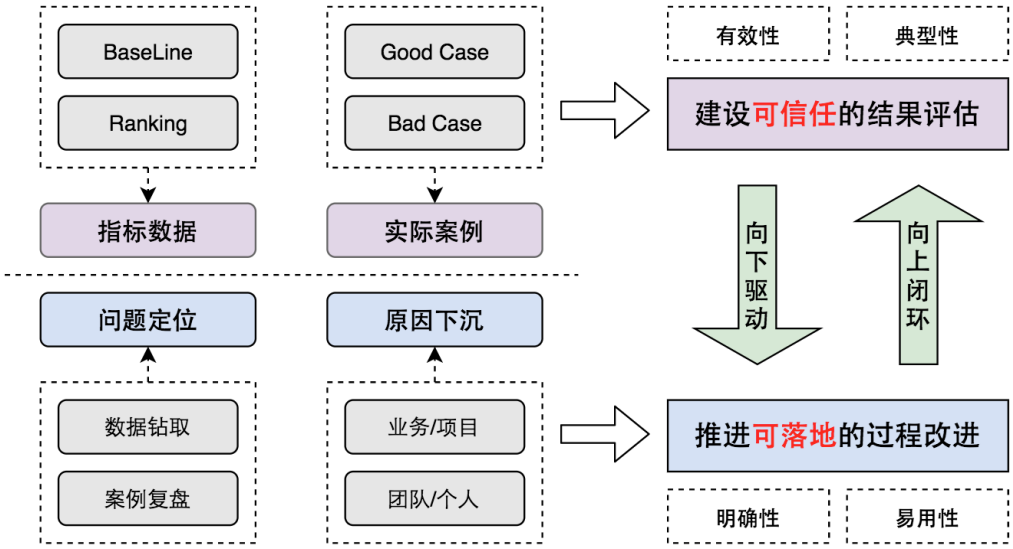
数据与问题可关联

制定计划并改进

改进措施的制定和实施是整个质量改进过程中的重中之重。基于经验，给出三个策略。

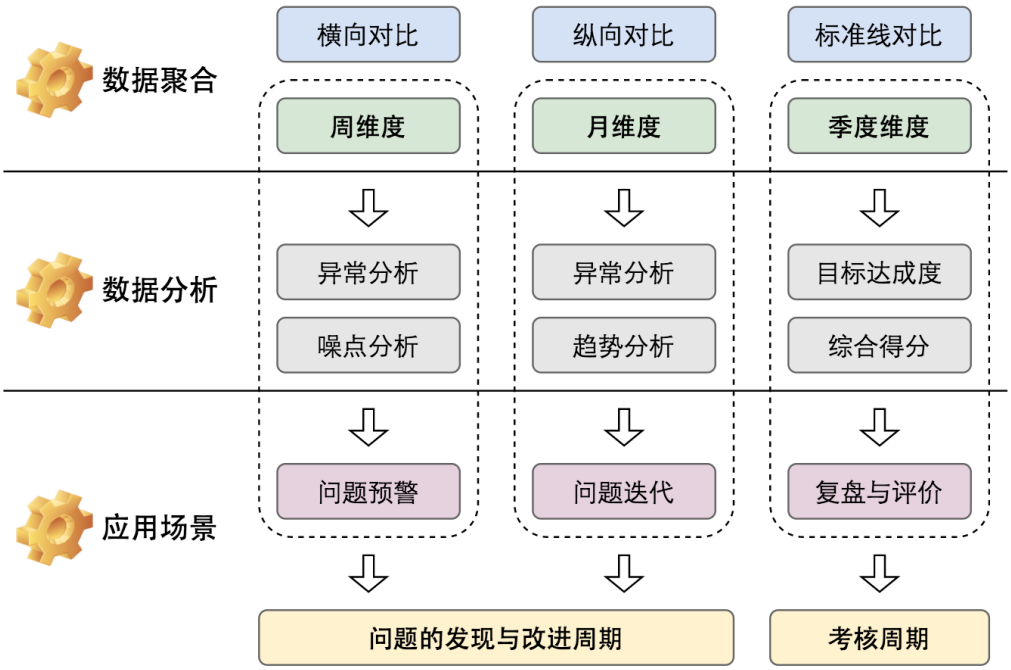
自上而下与自下而上相结合

- 自上而下：通过有效数据、典型案例，建设可信任的结果评估体系；以此为基础，利用每一个问题数据，在Leader层强化质量意识，借鉴向上管理的思路，实现质量改进的向下驱动。
- 自下而上：通过案例复盘、数据钻取，对问题进行明确定位，让问题方基于工具即可将问题下沉到具体项目或具体角色，进而推进可落地的过程改进，并持续利用结果评估体系衡量改进效果，实现质量改进的向上闭环。



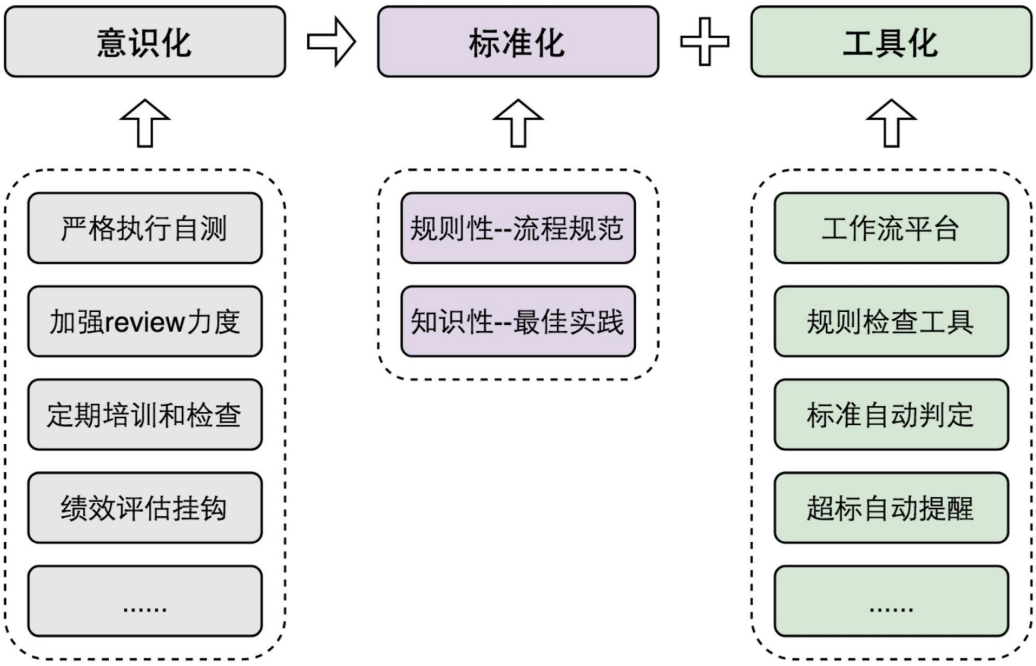
多维度的数据聚合与分析相结合

- 周维度数据聚合：对周数据中的异常进行分析，并排除掉因周期偏短导致的数据噪点，重在对问题进行风险预警。
- 月维度数据聚合：对月数据中的异常进行分析，并结合数据的变化趋势，重在对问题进行确认和改进。周维度和月维度相结合，构成了质量管理中的问题发现与改进周期。
- 季度维度数据聚合：对季度数据的分析，重在得出对质量目标的完成度并给出质量评分，并对过程中的问题进行回顾和总结，构成了质量管理中的考核周期。

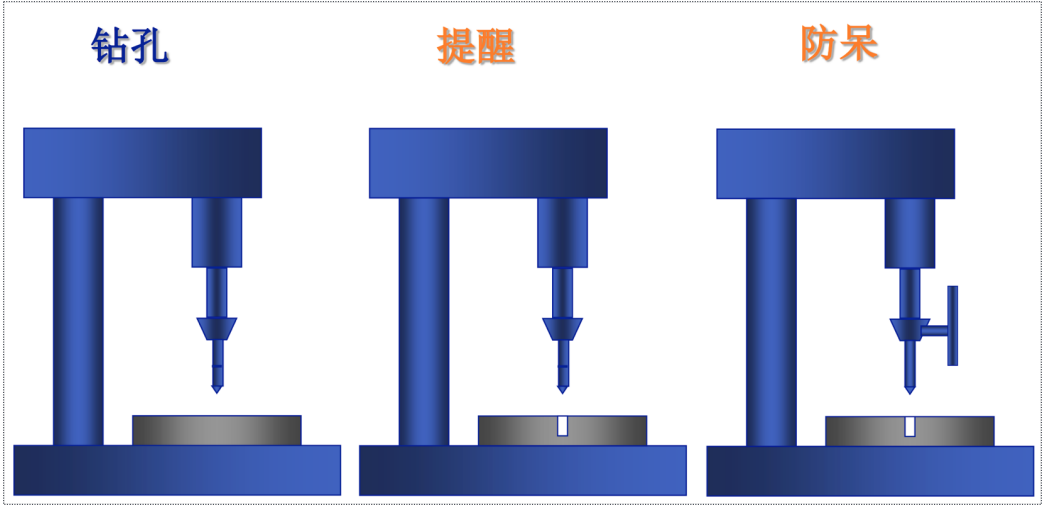


流程的标准化与工具化相结合

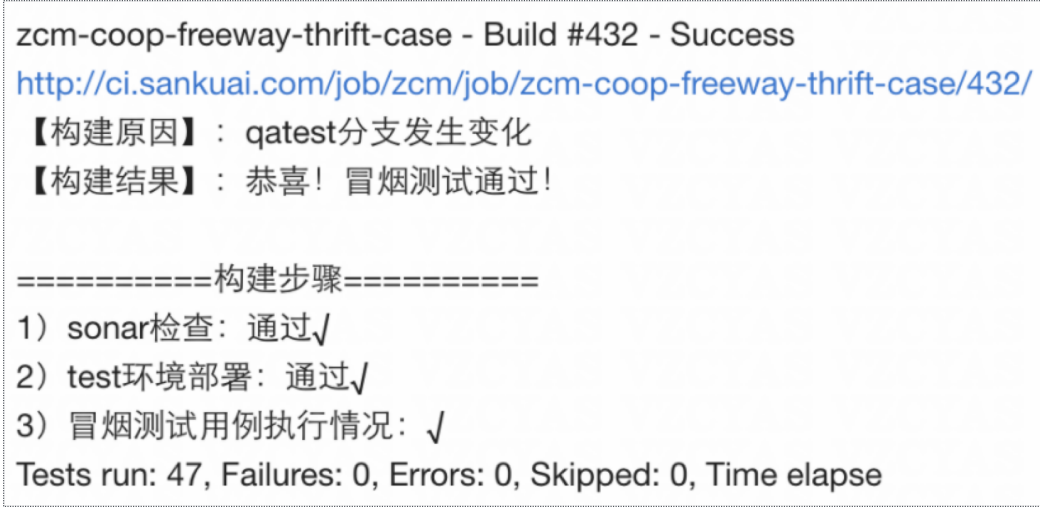
- 在改进提测质量的初期阶段，对于流程的优化经常出现在各个项目总结的改进措施中，但大多是通过意识、模板或者口头提醒来实现，这无疑增加了流程的接入成本、执行难度，进而降低了流程的约束力。
- 借鉴在制造业中常见的一种解决思路——“防呆措施”，在流程标准化之后，应尽可能将其工具化，提升流程的生命力。



- 防呆措施的目的之一是防止不符合流程的产出物交付到下游。
- 将防呆措施应用到提测流程，即应实现各项准入标准的自动化检查，类似如下提测时校验：



防呆措施



测试流程

回顾与反馈

主要从时间维度、项目维度两方面开展。

1. 时间维度，各类周会、双周会、季度总结，对质量数据进行Review。
2. 项目维度，重在项目复盘。复盘可以看成PDCA环和环之间的连接，有了它，PDCA才能环环相扣、周而复始。

迭代与推广

若改善有效，则进行推广。若改善无效，则分析原因，修改计划，重新启动另一轮PDCA。

1. 指标与标准的持续迭代，如：过程中对Sonar千行严重问题数的标准由0.1提升到0。
2. 度量模型与方法工具的推广，如：质量报表、Sonar在PR时触发检查不通过不允许Merge、提测准入自动化等等在其他业务的推广。

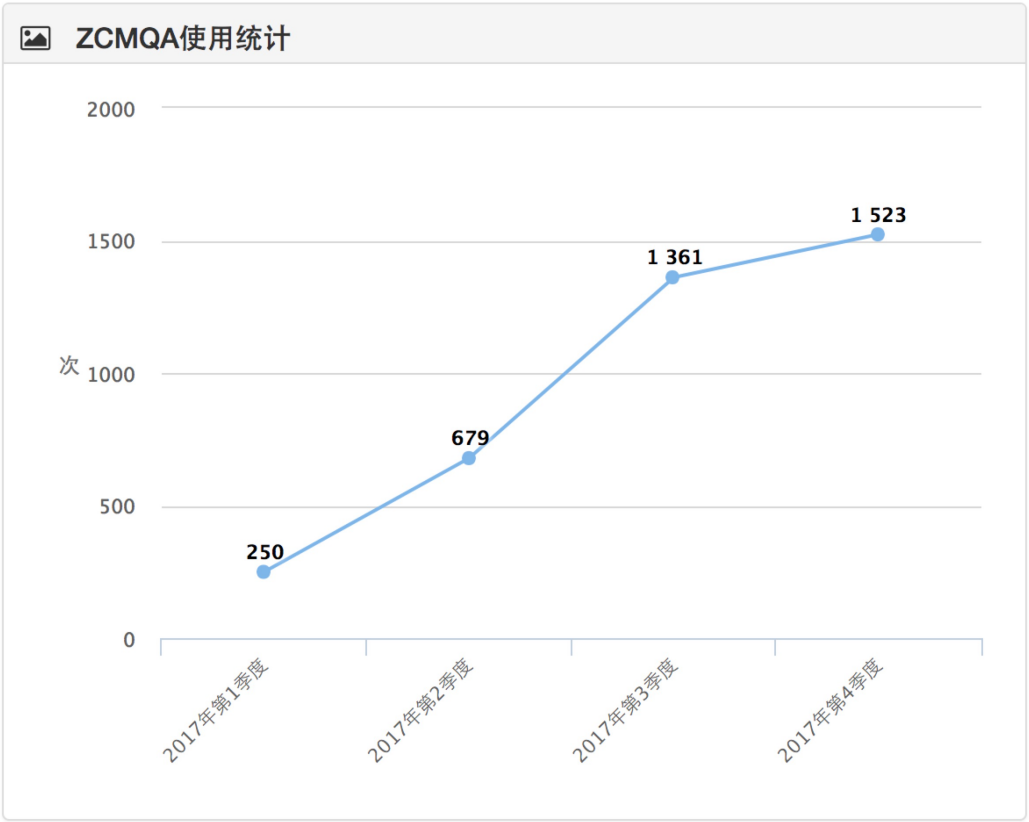
效果

对于效果的评估，主要从三方面进行说明。

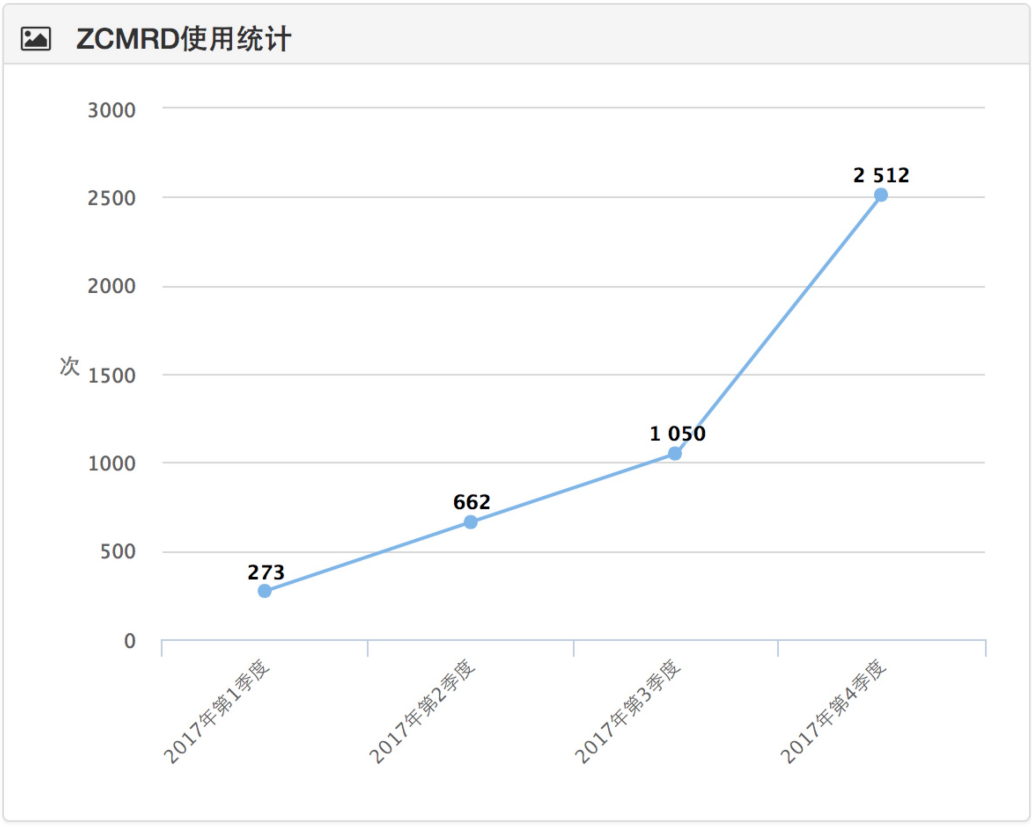
1. 质量数据的关注度
2. 质量指标的达成度
3. 过程质量改进对迭代效率的提升效果

质量数据关注度

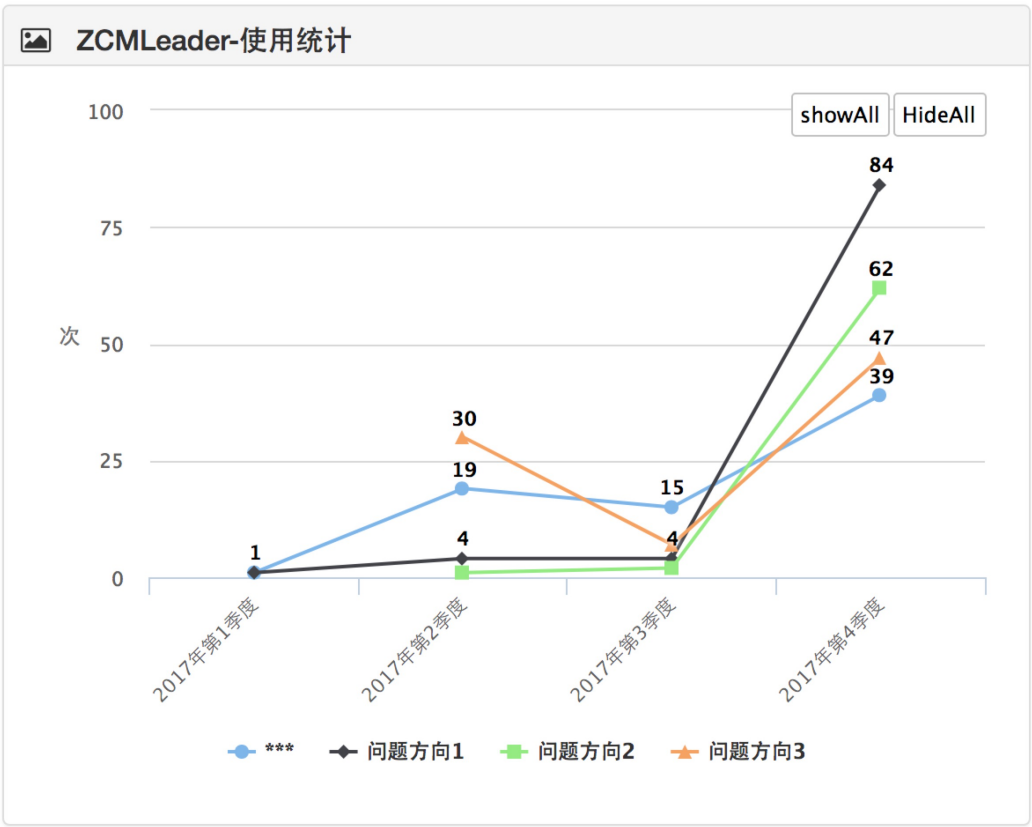
主要体现在团队各方对质量报表的使用率。以下三张图分别为：智能支付QA、智能支付RD、智能支付RD Leader对质量报表的使用率走势。



智能支付QA



智能支付RD



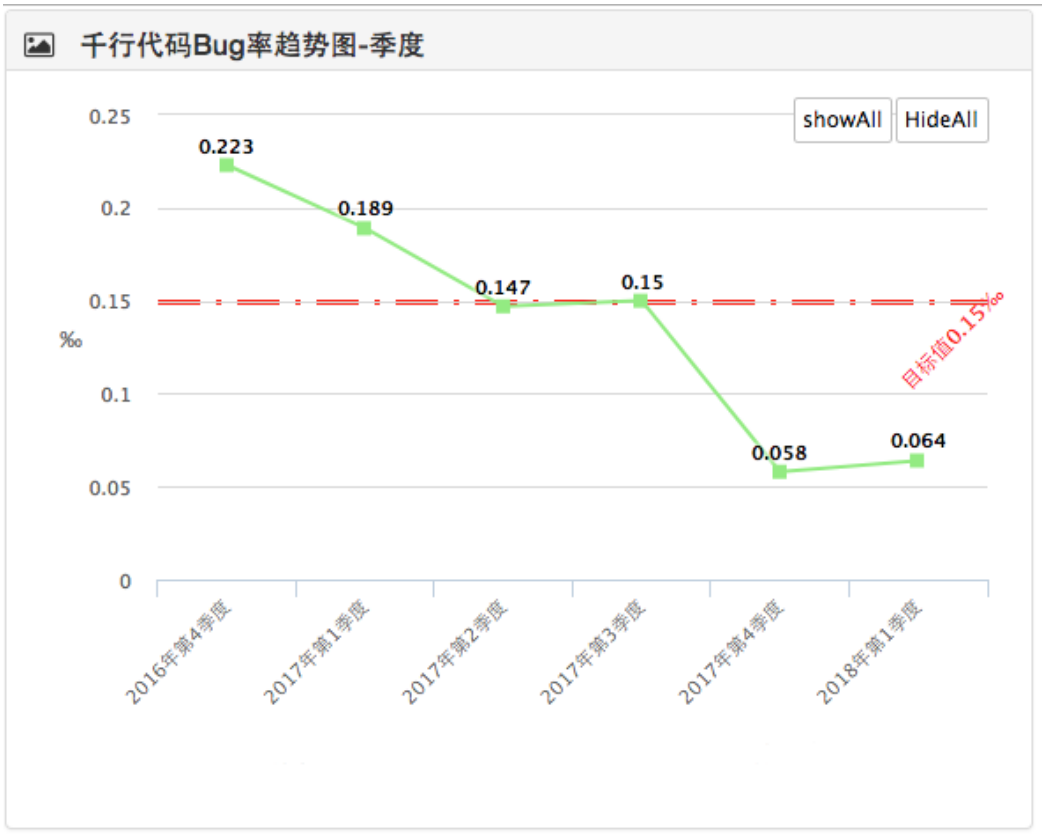
智能支付RD

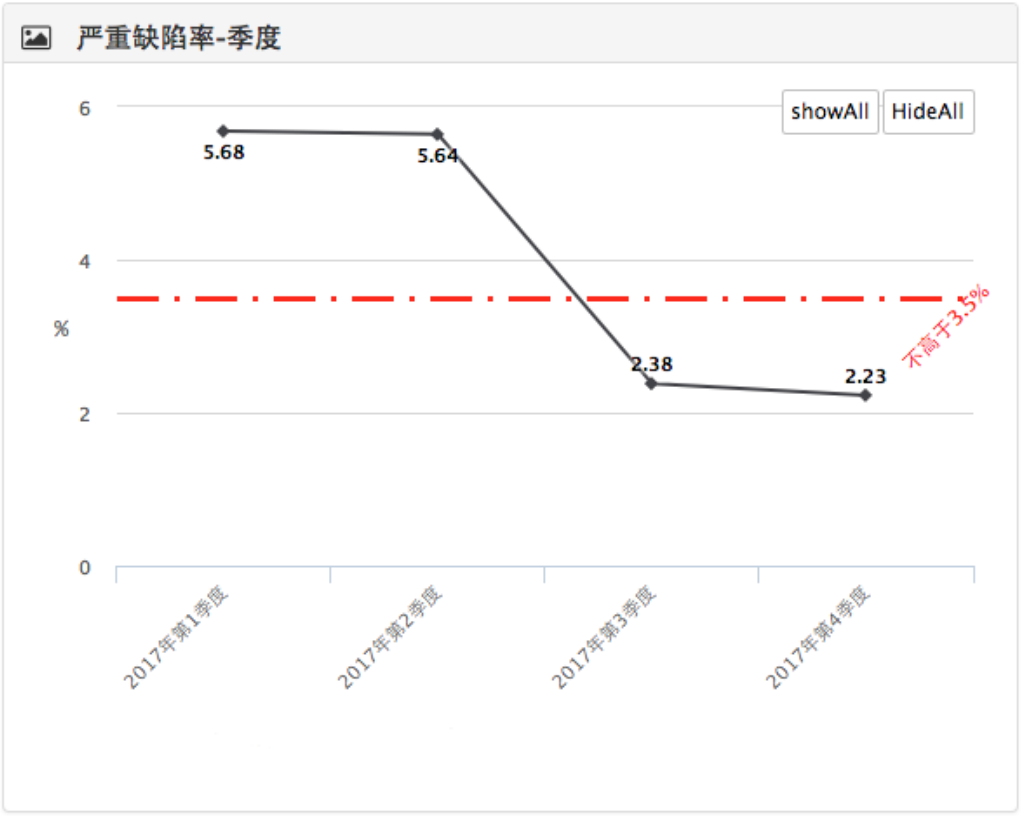
质量指标达成度

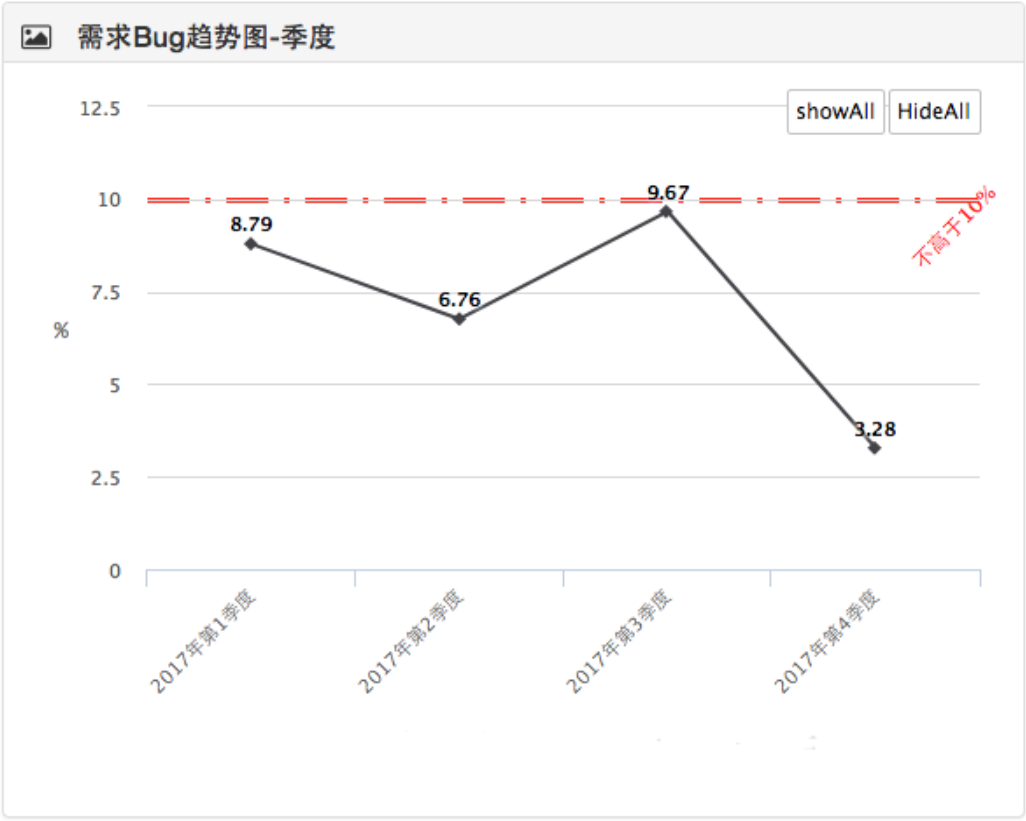
对质量指标的达成情况进行说明，其中：初始值为16年Q4的情况。

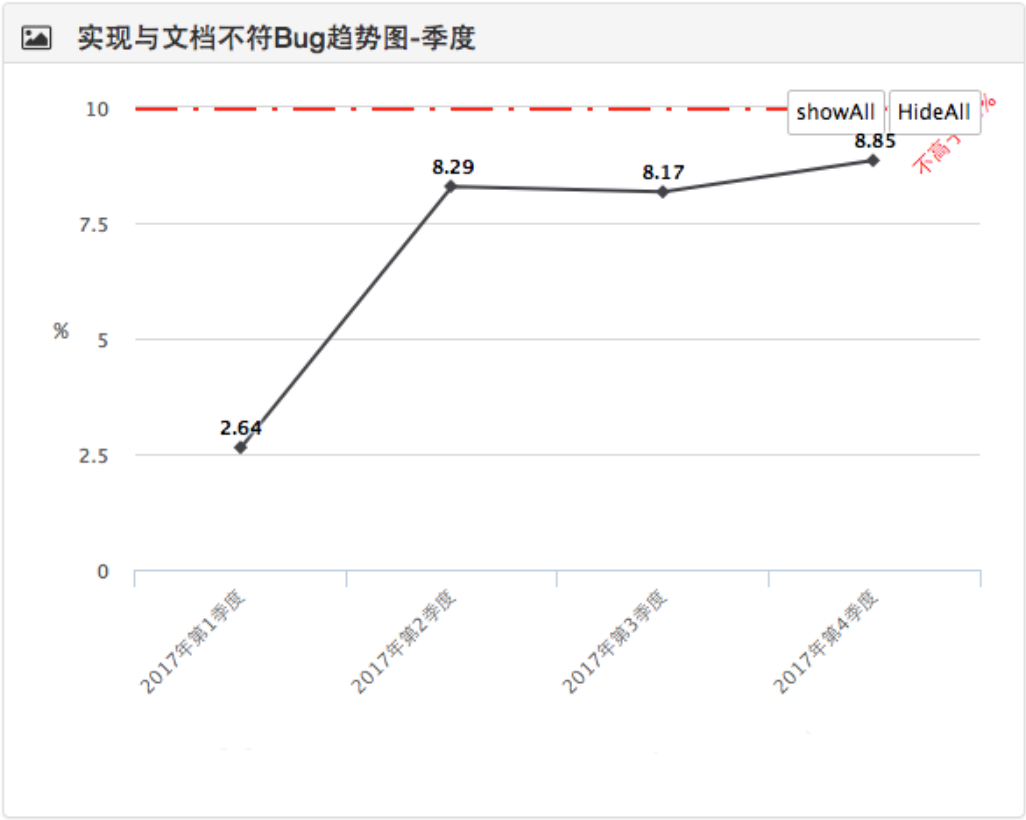
度量指标	初始值	标准值	目标完成度
千行代码缺陷率	移动端：0.7 前端：0.25 后端：0.3	移动端：< 0.45 前端：< 0.2 后端：< 0.15	**整体达标，但存在个别方向缺陷率较高**
Sonar千行代码严重问题数	Blocker > 0 总数：0.2	Blocker：0 总数：< 0.1	**达标**
严重缺陷占比	14%左右	< 3.5%	**达标**
需求缺陷占比	18%左右	< 10%	**达标**
实现与文档不符缺陷占比	< 10%	< 10%	**达标，但有升高趋势，接近标准值**

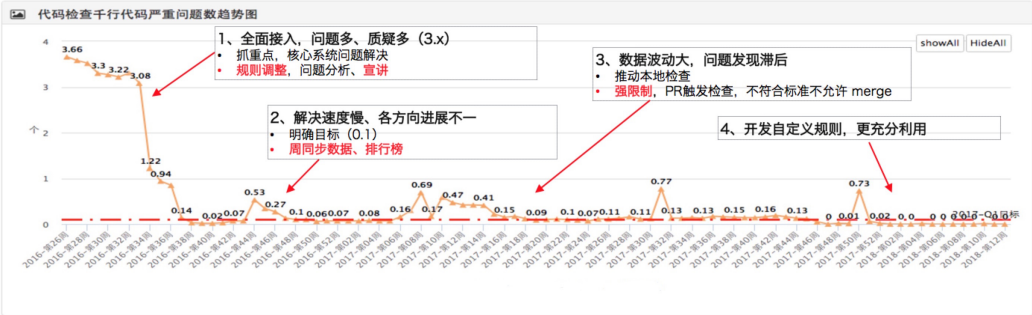
近几个Q的变化趋势，如下：




严重缺陷率-季度
showAll
HideAll
6
4
2
0
%
5.68
5.64
2.38
2.23
不高于3.5%
2017年第1季度
2017年第2季度
2017年第3季度
2017年第4季度

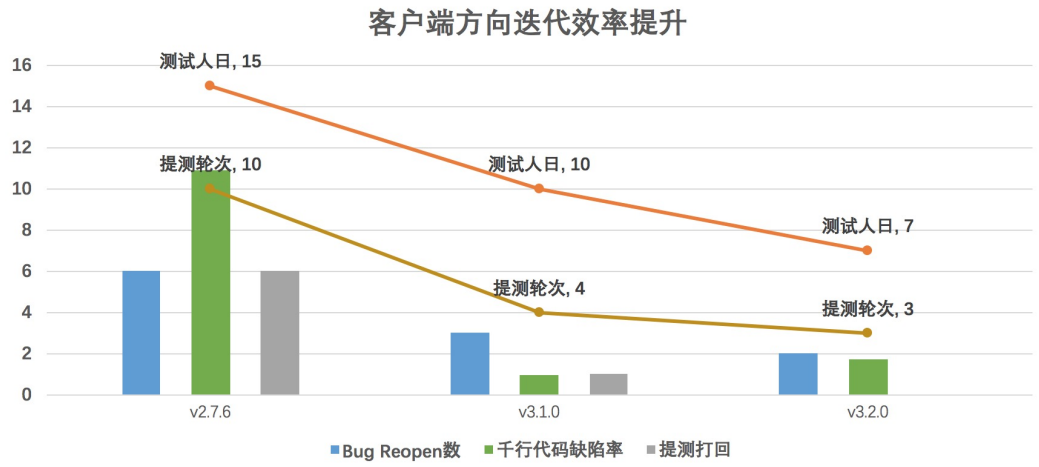

需求Bug趋势图-季度
showAll
HideAll
12.5
10
7.5
5
2.5
0
%
8.79
6.76
9.67
3.28
不高于10%
2017年第1季度
2017年第2季度
2017年第3季度
2017年第4季度





迭代效率提升效果

以客户端方向为例（之前过程质量存在较严重的问题），说明过程质量改进对迭代效率的提升效果。



总结

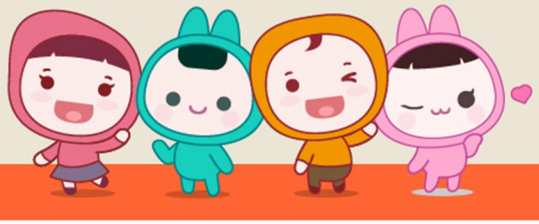
经过一段时间的摸索和实践，我们在提测质量上有了较明显的提升，过程中积累的方法、流程和工具也在推广使用。但提测质量只是全生命周期质量运营的一小部分，对于高速发展的智能支付业务，不仅要求整个质量保证体系的迭代优化，更要求全体成员不断提升质量思维、持续追求极致质量，进而形成一种质量文化，真正实现“人人重视质量、人人创造质量、人人享受质量”。

作者介绍

- 勋伟，美团点评高级测试开发工程师，金融服务平台智能支付业务测试负责人，2015年加入美团点评。

招聘

如果你想学习互联网金融的技术体系，亲历互联网金融业务的爆发式增长，如果你想和我们一起，保证业务产品的高质量，欢迎加入美团金融工程质量组。有兴趣的同学可以发送简历到：fanxunwei#meituan.com。



扫码关注技术团队
微信公众号

tech.meituan.com
美团技术博客

CODE A BETTER LIFE
一 行 代 码　亿 万 生 活