高效构建企业级精准测试体系

ZTEsoft中兴软创

—— 程海明

目录

01 精准测试

02 精准测试的关键

03 主流覆盖率工具及其实现逻辑简析

04 手工测试引入精准

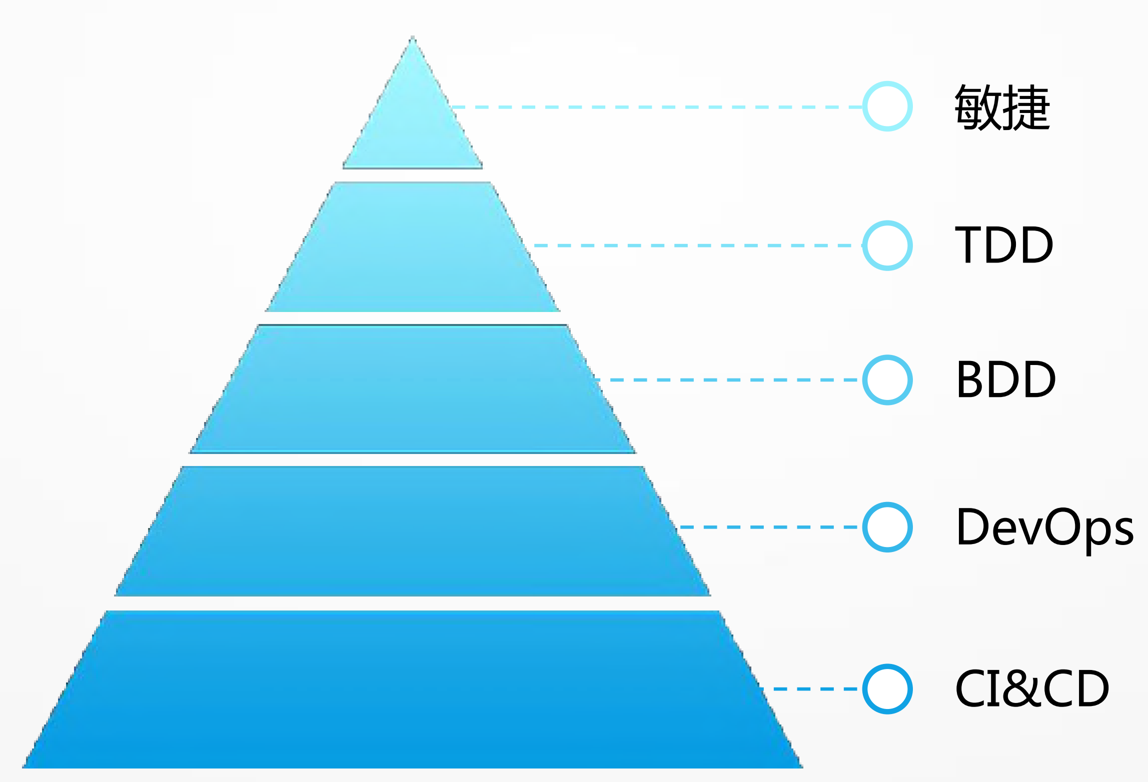
05 主流测试工具和自建测试平台如何做精准测试的基础数据建设

06 精准测试在CI&CD中的价值

07 测试用例重构

08 Q&A

革命浪潮



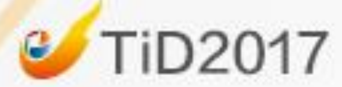
修改&重构的影响范围	测试用例设计质量的量化评估标准	
变化的测试外延界定	自动化回归的执行效率低下	
CI&CD	可视化管理	全链路的版本质量监控

什么是精准测试

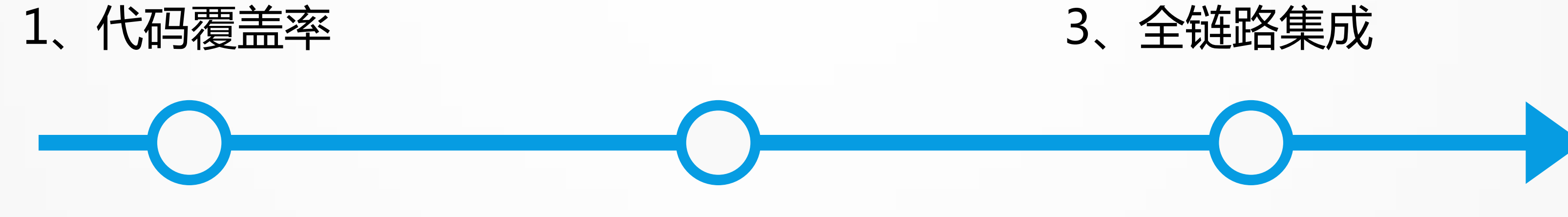
- 相对于普通测试，精准测试是在传统测试过程中，通过技术手段对被测试程序进行360度全景测试，将测试过程可视化、数字化、标准化，从而达到被测试上线稳定、无风险、维护成本低等优势。和传统测试比起来，精准测试需要通过程序自动的生成海量的、不能被篡改的原生态测试数据，通过这些数据的汇集、分析对测试进行带有只能性的指导，避免了传统测试过程中人力记录的时间成本与真实性的问题。

- 核心特性：测试示波器、崩溃捕获、路劲分析、全方位可视化的测试每一步、实时测试数据接收、测试用例和代码的双向关联与追溯、全面记录测试过程中的测试设备和测试人员以及测试用例与代码等多者之间的关联，并通过丰富的报表和技术债务与风险指标进行展示。

- 精准测试和传统测试的联系在于，整个测试过程中实际操作完全基于传统测试，并通过技术手段在传统的测试过程中自动产生原生态的测试数据。



精准测试的关键



主流覆盖率工具及其实现逻辑简析

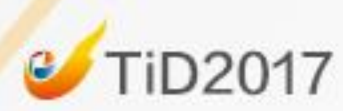
- 针对java，主流的是：

  jacoco，https://github.com/jacoco/jacoco

- 针对C++， 主流的是：

  gcov，https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

  http://gcovr.com/guide.html



代码覆盖率

Jacoco Features

Coverage analysis of instructions (C0), branches (C1), lines, methods, types and cyclomatic complexity.

Based on Java byte code and therefore works also without source files.

Simple integration through Java agent based on-the-fly instrumentation. Other integration scenarios like custom class loaders are possible through the API.

Framework agnostic: Smoothly integrates with Java VM based applications like plain Java programs, OSGi frameworks, web containers or EJB servers.

Compatible with all released Java class file versions.

Support for different JVM languages.

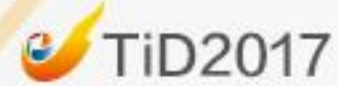
Several report formats (HTML, XML, CSV).

Remote protocol and JMX control to request execution data dumps from the coverage agent at any point in.

Ant tasks to collect and manage extimeecution data and create structured coverage reports.

Maven plug-in to collect coverage information and create reports in Maven builds.

PS：The probe does not record the number of times it has been called or collect any timing information



主流覆盖率工具的实现逻辑简析

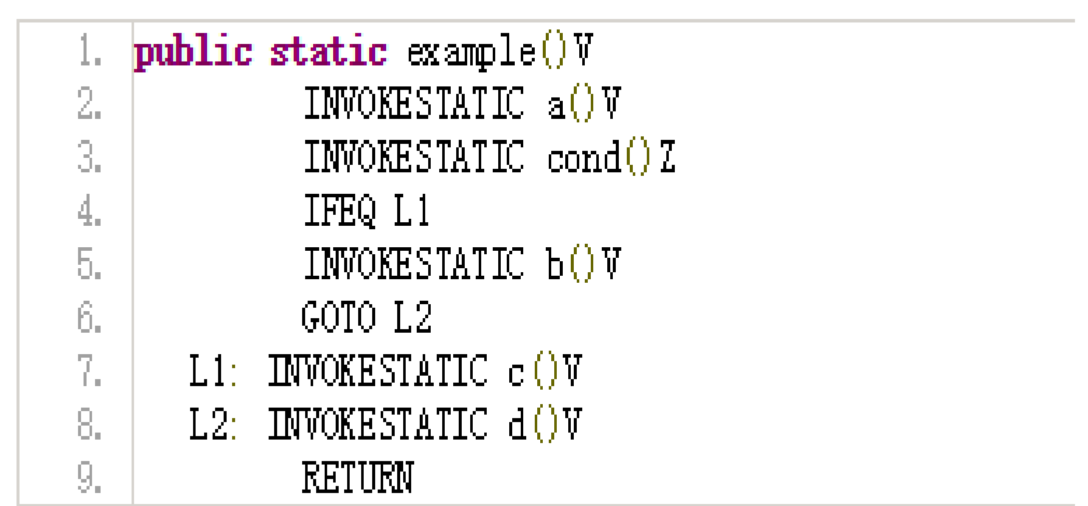
- 代码覆盖率工具，基本的实现方式是梳理代码的控制流程，在控制处加入探针，进而达到记录代码执行轨迹。针对java，通过对字节码的控制流梳理嵌入探针，完成对代码流向的轨迹记录。

Control Flow Graphs for Java Bytecode

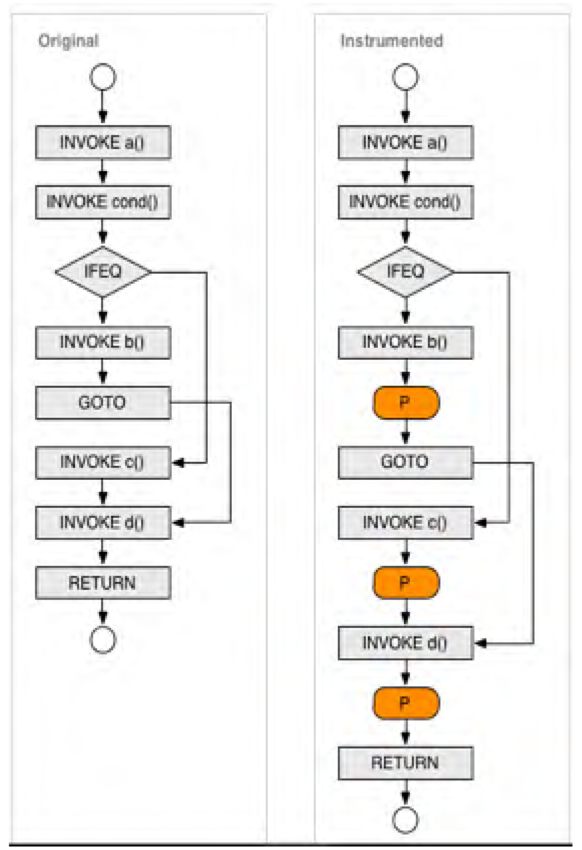
As an starting point we take the following example method that contains a single branching point:

```
1. public static void example() {
2.       a();
3.       if (cond()) {
4.             b();
5.       } else {
6.             c();
7.       }
8.       d();
9. }
```

主流覆盖率工具的实现逻辑简析



```
1. public static example()V
2.         INVOKESTATIC a()V
3.         INVOKESTATIC cond()Z
4.         IFEQ L1
5.         INVOKESTATIC b()V
6.         GOTO L2
7.   L1:  INVOKESTATIC c()V
8.   L2:  INVOKESTATIC d()V
9.         RETURN
```



- 详细见：http://www.jacoco.org/jacoco/trunk/doc/flow.html

TiD2017

代码覆盖率

Jacoco，ant的集成：

```
<jacoco:coverage output="file" destfile="./jacoco.exec" includes="com.ztesoft.*">
    <junit printsummary="on" fork="true" forkmode="once" showoutput="true">
      <classpath>
        <path>
          <fileset dir="/home/ztp/zcipclient/workpath/972/code/lib">
            <include name="**/*.jar"/>
          </fileset>
      </classpath>
      <formatter type="xml"/>
      <batchtest todir="${result.dir}">
        <fileset dir="/home/ztp/zcipclient/workpath/972/code/ZCIPTest/bin">
          <include name="**/unittest/**/*.class"/>
        </fileset>
      </batchtest>
    </junit>
  </jacoco:coverage>
```

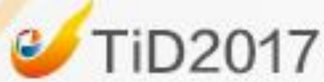
测试完成后会生成对应的覆盖率数据文件jacoco.exec，然后使用jacoco:report task完成格式化处理，生成相关的结果文件

代码覆盖率

Gcov

Gcov 需要在编译时增加编译选项：
CCFLAGS := ${CCFLAGS} -fprofile-arcs -ftest-coverage
LLDFLAGS := ${LLDFLAGS} -fprofile-arcs -ftest-coverage -lgcov

在测试执行完成后需发送停止程序的信号量，完成数据覆盖率数据的dump，最后通过lcov工具对结果数据进行格式化处理动作



代码覆盖率

JaCoCo Ant

JaCoCo Ant

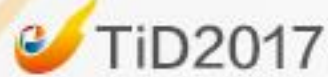
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Code		38%		26%	10,529	14,887	34,320	59,668	4,896	8,305	429	807
Total	172,179 of 277,311	38%	9,667 of 13,090	26%	10,529	14,887	34,320	59,668	4,896	8,305	429	807

```
    public int ciServiceDbRelaModify(List<CIServiceDbRelaDto> newSerDbRelass,
        List<CIServiceDbRelaDto> oldSerDbRelass, Connection conn, int staffId, int flowId) {
        try {
            List<CIServiceDbRelaDto> newSerDbRelas = new ArrayList<CIServiceDbRelaDto>();
            List<CIServiceDbRelaDto> oldSerDbRelas = new ArrayList<CIServiceDbRelaDto>();
            newSerDbRelas.addAll(newSerDbRelass);
            oldSerDbRelas.addAll(oldSerDbRelass);
            String oldValue = "";
            String newValue = "";
            boolean isNew = true;
            for (CIServiceDbRelaDto newSerDbRela : newSerDbRelass) {
                isNew = true;

                for (CIServiceDbRelaDto oldSerDbRela : oldSerDbRelass) {
                    if (mergeSerDbRela(oldSerDbRela, newSerDbRela)) {
                        oldSerDbRelas.remove(oldSerDbRela);
                        isNew = false;
                        break;
                    }
                }
                if (isNew) {
                    newValue = dealServiceDbReladto(newSerDbRela);
                    insertToManagerLog("", newValue, "insert", staffId, "CI_SERVICE_DB_RELA", flowId, conn);
                }
            }

            if (!oldSerDbRelas.isEmpty()) {
                for (CIServiceDbRelaDto oldNodeCfg : oldSerDbRelas) {
                    oldValue = dealServiceDbReladto(oldNodeCfg);
                    insertToManagerLog(oldValue, "", "delete", staffId, "CI_SERVICE_DB_RELA", flowId, conn);
                }
            }

            return 1;
        }
```



手工测试引入精准

- 手工测试引入精准，每次测试都能记录精准数据

手工测试覆盖率

选择被测试的IP地址 | 选择你的应用 | 请输入要测试的单号(可不填) | 开始测试

覆盖率报告请点击

我的历史记录

单号

合并

	时间	单号	覆盖率	来源	URL	导出
	★2017-06-14 21:11:56		lines......: 1.3% (771 of 55630 lines) methods....: 2.1% (161 of 7352 methods) branches...: 1.3% (169 of 12072 branches)	测试生成	http://[illegible]:8080/smartcenter/manualtest/[illegible]/20170614211156/report/index.html	
	★2017-05-03 17:02:11		lines......: 3.6% (2049 of 55630 lines) methods....: 6.2% (457 of 7352 methods) branches...: 2.6% (324 of 12072 branches)	测试生成	http://[illegible]:8080/smartcenter/manualtest/[illegible]/20170503170211/report/index.html	

手工测试如何引入精准

当前比对文件：DependProdOrderChecker.java，代码修改了12行,其中手工测试覆盖了0行,修改代码的行覆盖率为0%

共14块不同

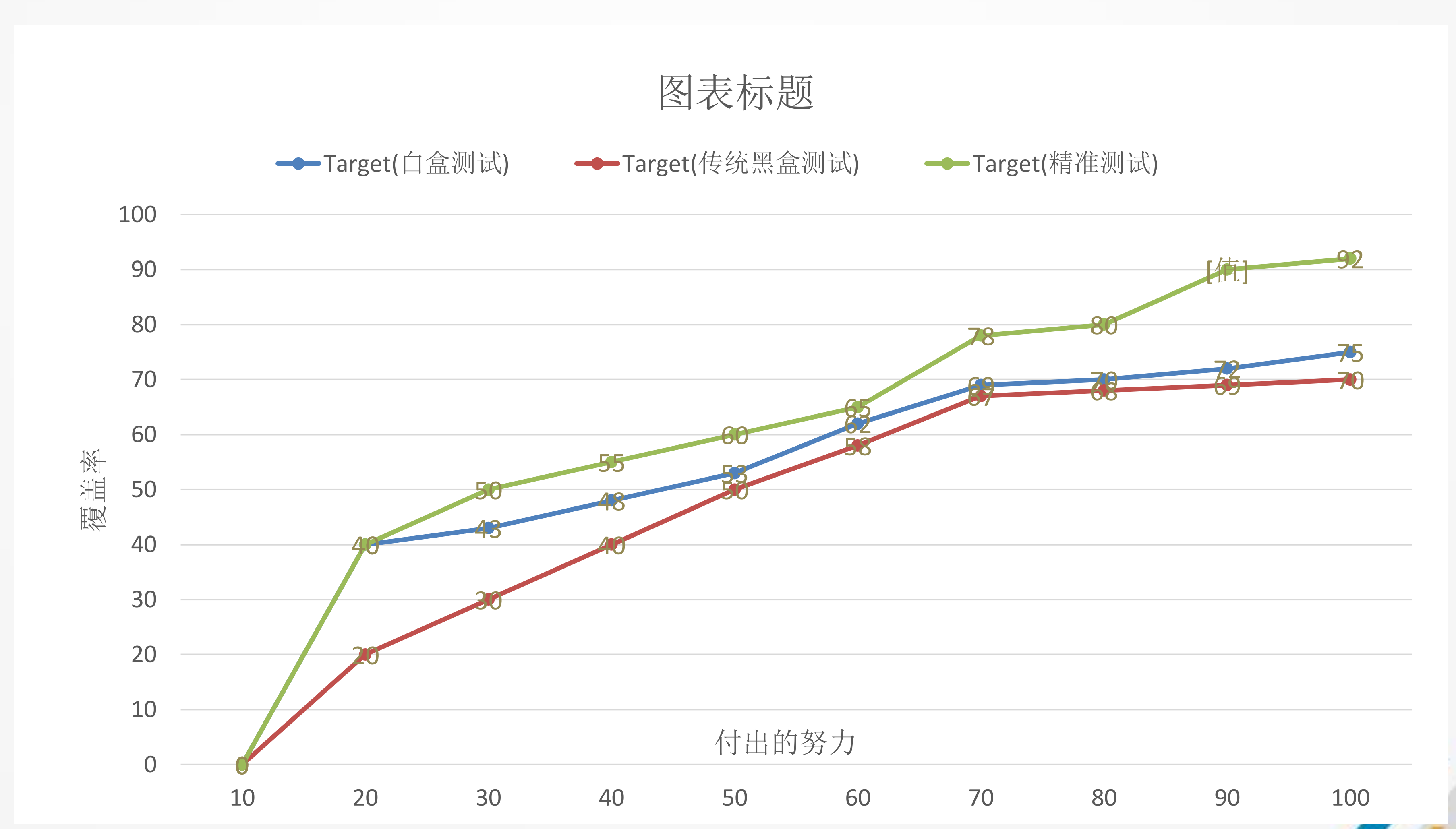
```
38   * congo对单个依赖产品订单的项目化校验
39   *
40   * @author &lt;br&gt;
41   * @version 1.0 &lt;br&gt;
42   * @taskId &lt;br&gt;
43   * @CreateDate &lt;br&gt;
44   * @since &lt;br&gt;
45   */
46  public class DependProdOrderChecker extends
47      com.ztesoft.zsmart.bss.order.cc.bll.base.validator.component.DependProdOrder<br>Checker {
48
49      /**
50       * logger &lt;br&gt;
51       */
52      private static ZSmartLogger logger = ZSmartLogger.getLogger(DependProdOrderC<br>hecker.class);
53
54      /**
```

```
31   * congo对单个依赖产品订单的项目化校验
32   *
33   * @author &lt;br&gt;
34   * @version 1.0 &lt;br&gt;
35   * @taskId 515216 &lt;br&gt;
36   * @CreateDate &lt;br&gt;
37   * @since V7.3 &lt;br&gt;
38   */
39  public class DependProdOrderChecker extends com.ztesoft.zsmart.bss.order.cc.bll.<br>base.validator.component.DependProdOrderChecker {
40
41      /**
42       * logger &lt;br&gt;
43       */
44      private static ZSmartLogger logger = ZSmartLogger.getLogger(DependProdOrderC<br>hecker.class);
45
46      /**
```

```
31   * 是否校验用户状态
32   */
33      private static String isNeedCheckStates4RelaProd = ConfigurationMgr.instance().getString("projectConfig.idea.profile.IsNeedCheckStates4RelaProd");
34      /**
35   * 根据依赖产品CODE做校验
36   * @author
37   * @taskId &lt;br&gt;
38   * @param subsOrder
39   * @param dependProdOrder
40   * @param dependProdSpecDto
41   * @throws BaseAppException &lt;br&gt;
42   */
43      public void checkDependProdOrderByCode(DynamicDict subsOrder, DynamicDict dependProdOrder,
44                                          ProdSpecDto dependProdSpecDto) throws BaseAppException {
45          if (OperationType.REMAIN.equals(dependProdOrder.getString("OPERATION_TYPE"))) {
46              return;
```

TiD2017

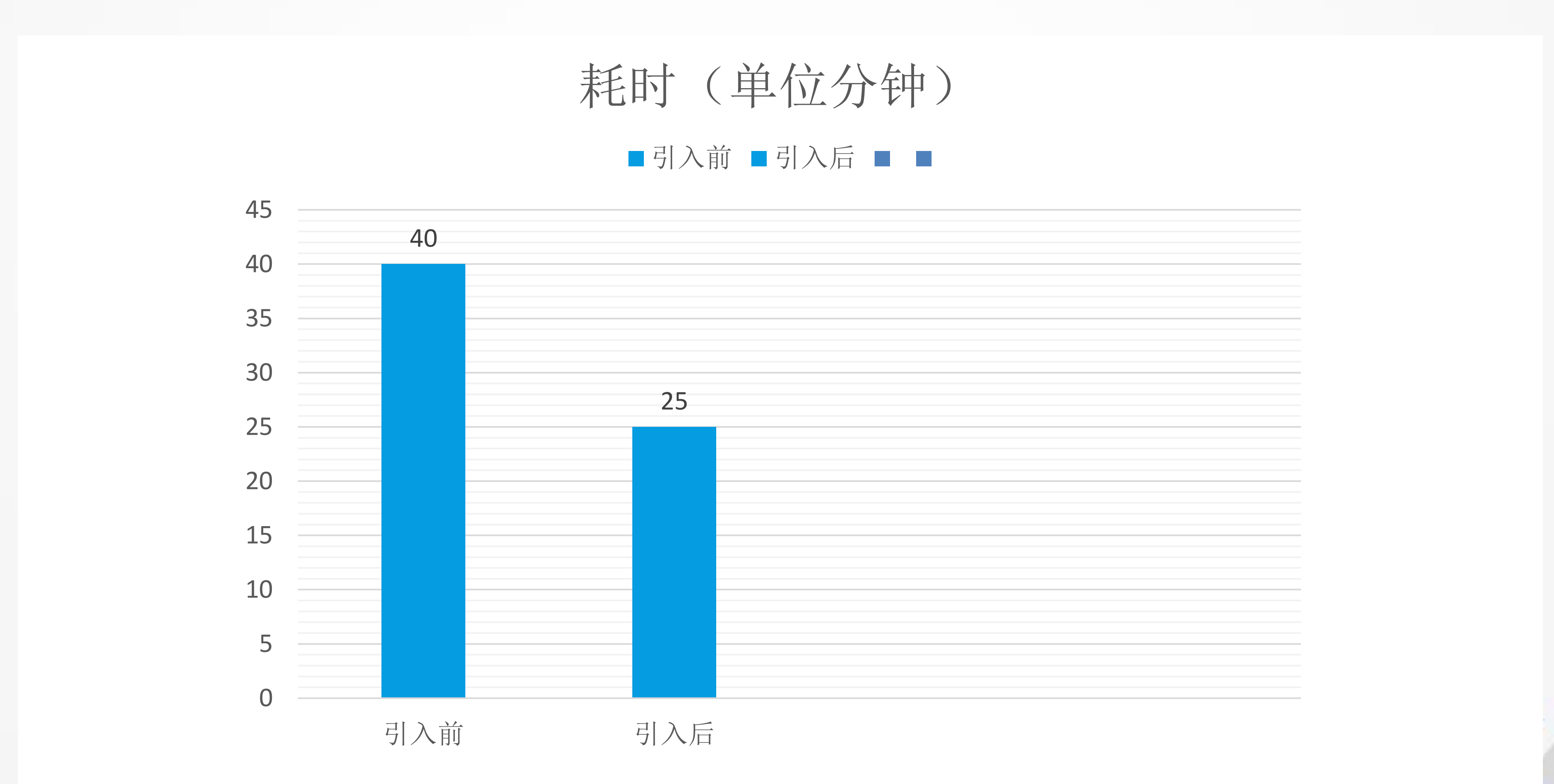
手工测试引入精准



TiD2017

手工测试引入精准

- 手工测试过程中Bug平均定位耗时



手工测试引入精准

手工测试中引入精准测试带来了哪些优势：

01 提高测试效率

02 可视化展示手工测试的覆盖率信息

03 差异化版本比对覆盖详情

04 测试设计质量的量化评估

05 多次合并测试覆盖率数据

06 打通开发和测试壁垒

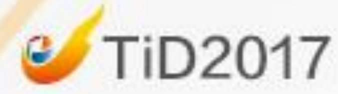
07 按需求汇总比对差异化结果

代码覆盖率

有了覆盖率，我们完成了以下动作：

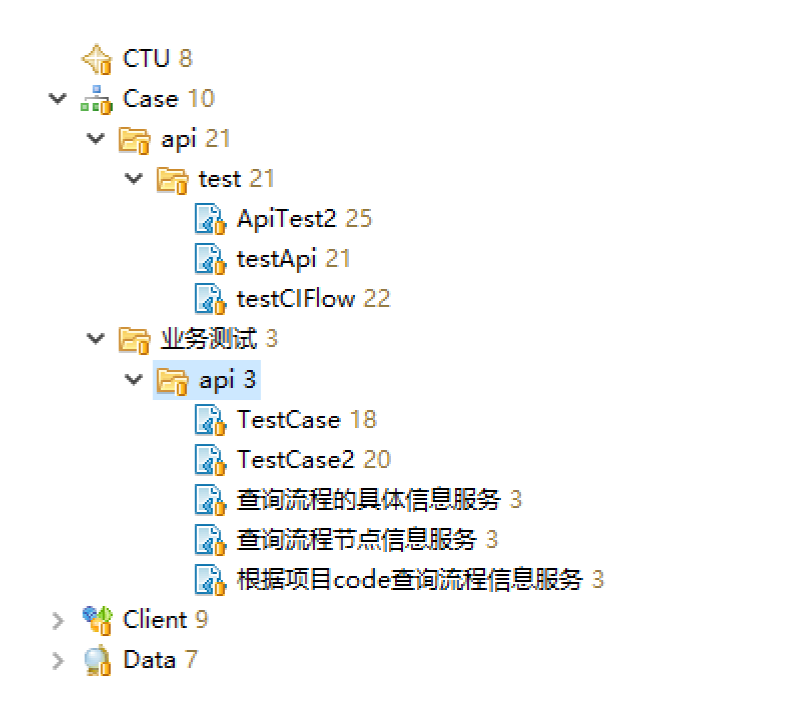
质量有了评估，我们还需要更快速的适应市场变化，快速的迭代版本和交付产品。

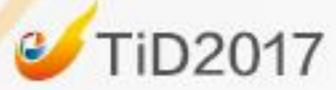
精准需更近一步



精准测试的关键

- case和code关联数据建设，基础测试用例数据的建设

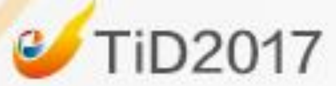




精准测试的关键

code —1——N→ case

文件	用例	时间
com/zsmart/core/common/Constants.java	Case/api/test/ApiTest2.case	2017-05-03
	Case/api/test/testApi.case	2017-05-03
	Case/api/test/testCIFlow.case	2017-05-03
	Case/业务测试/api/TestCase.case	2017-05-03
	Case/业务测试/api/TestCase2.case	2017-05-03
	Case/业务测试/api/查询流程的具体信息服务.case	2017-05-03
	Case/业务测试/api/查询流程节点信息服务.case	2017-05-03
	Case/业务测试/api/根据项目code查询流程信息服务.case	2017-05-03



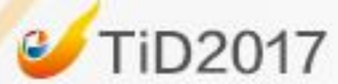
精准测试的关键

code N ← 1 case

用例	文件	时间
Case/业务测试/api/查询流程的具体信息服务.case	chm/test/api/controller/QueryFlowDescController.java	2017-05-03
	chm/test/api/service/impl/QueryFlowDescServiceImpl.java	2017-05-03
	chm/test/bll/CIFlowInstancebll.java	2017-05-03
	chm/test/dal/CIFlowInstanceDAO.java	2017-05-03
	chm/test/dal/CIFlowNodeDAO.java	2017-05-03
	chm/test/dto/CiFlowInstanceDto.java	2017-05-03
	chm/test/dto/FlowLabelDto.java	2017-05-03
	chm/test/socket/ServerMonitor.java	2017-05-03
	chm/test/socket/SingleSocketList.java	2017-05-03
	chm/test/socket/SocketSingle.java	2017-05-03
	com/zsmart/core/common/ConfigManager.java	2017-05-03
	com/zsmart/core/common/Constants.java	2017-05-03
	com/zsmart/core/common/DateHelper.java	2017-05-03
	com/zsmart/core/common/SqlLogCortroller.java	2017-05-03
	com/zsmart/core/common/SqlSession.java	2017-05-03
	com/zsmart/core/common/StringHelper.java	2017-05-03
	com/zsmart/core/common/UserSessionInfo.java	2017-05-03
	com/zsmart/core/common/Utils.java	2017-05-03
	com/zsmart/core/common/db/DAOFactory.java	2017-05-03
	com/zsmart/core/common/db/ZCIPDAO.java	2017-05-03

主流测试工具和自建测试平台如何做精准的基础数据建设

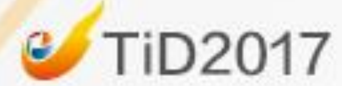
- 自动化测试用例的标准步骤



主流测试工具和自建测试平台如何做精准测试的基础数据建设

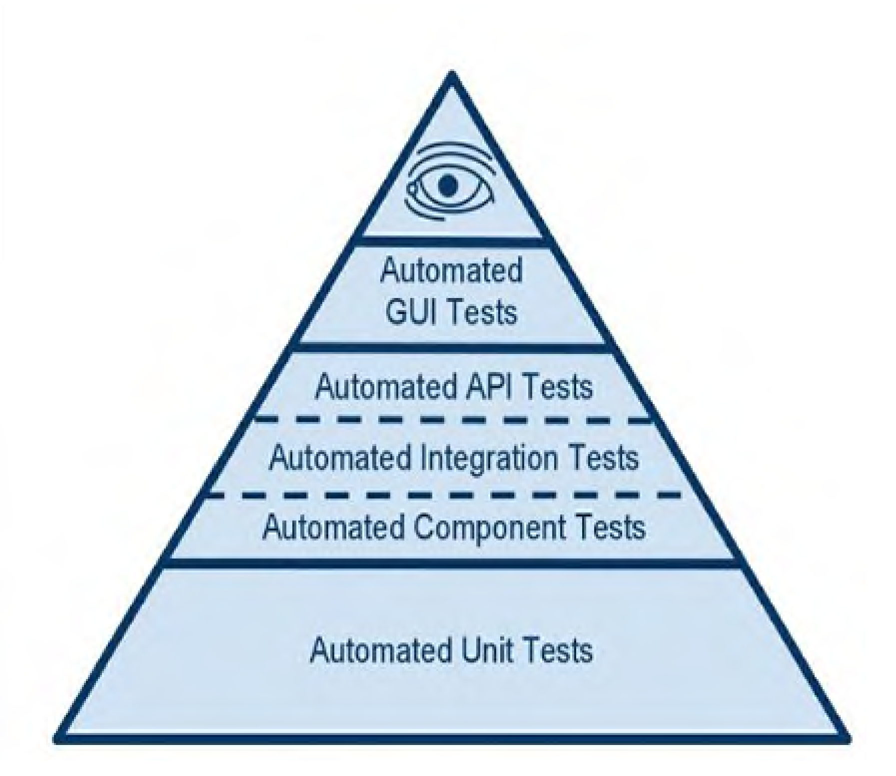
为了数据的准确性：

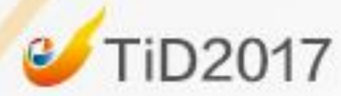
- 消除隔离第三方的操作的干扰
- TestCase执行前清除干扰数据
- TestCase执行结束后获取用例完整关联的覆盖率数据，并清除已经获　取的数据



主流测试工具和自建测试平台如何做精准测试的基础数据建设

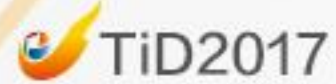
按照分层测试理论，测试可以分成不同的层，按照google测试的划分小型测试、中型测试、大型测试；但在精准测试覆盖率获取需求下只存在TestCase关联单个应用和TestCase关联多个应用的两种场景





TestCase 关联单个应用的实践

- Junit/testng等工具，为了精准，必须在每个测试用例执行完成后dump下覆盖率数据，针对junit4@Test注解的用例，我们可以通过修改BlockJUnit4ClassRunner类中的runChild方法，在每次执行完用例后进行覆盖率数据的dump.



TestCase 关联单个应用的实践

- 针对testng，testng提供了监听机制，在每次用例执行完成后会调用对应的监听动作，我们可以继承监听，编写自定义的函数完成对覆盖率数据的dump

```
public class TestListeners implements ITestListener {

        public void onTestStart(ITestResult result) {

        }

        public void onTestSuccess(ITestResult result) {
                System.out.println("success: " + result.getTestClass().getName() + " -- " + result.getMethod().getMethodName());
                downloadThreadingTestData(result);
        }

        public void onTestFailure(ITestResult result) {
                System.out.println("fail: " + result.getTestClass().getName() + " -- " + result.getMethod().getMethodName());
                downloadThreadingTestData(result);
        }



        public void onTestSkipped(ITestResult result) {

        }

        public void onTestFailedButWithinSuccessPercentage(ITestResult result) {
                // TODO Auto-generated method stub

        }
```

构建的集成

- Ant-junit

```
<jacoco:coverage output="tcpserver" address="${host.ip}" port="${jacoco.port}" includes="${coverage.class.list}">
  <junit printsummary="on" fork="true" forkmode="once" showoutput="true">
    <sysproperty key="junitCoverDir" value="${jacoco.single.dir}"/>
    <sysproperty key="jacocoPort" value="${jacoco.port}"/>
    <sysproperty key="jacocoAddress" value="${host.ip}"/>

    <classpath>
      <path>
        <fileset dir="/home/ztp/zcipclient/workpath/972/code/lib">
          <include name="**/*.jar"/>
        </fileset>
      </path>
      <pathelement path="/home/ztp/zcipclient/workpath/972/code/ZCIPTest/bin"/>
    </classpath>
    <formatter type="xml"/>
    <batchtest todir="${result.dir}">
      <fileset dir="/home/ztp/zcipclient/workpath/972/code/ZCIPTest/bin">
        <include name="**/unittest/**/*.class"/>
      </fileset>
    </batchtest>
  </junit>
</jacoco:coverage>
```

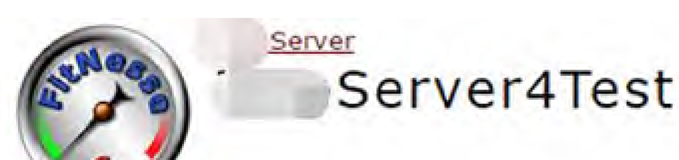
构建的集成

- Maven-testng

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.19.1</version>
  <configuration>
    <testFailureIgnore>true</testFailureIgnore>
    <includes>
      <include>**/*.java</include>
    </includes>
    <properties>
    <listener>com.ztesoft.zmsart.zcip.testng.TestListeners</listener>
    </properties>
    <systemPropertyVariables>
        <mavenTargetDir>${basedir}/target</mavenTargetDir>
        <junitCoverDir>${basedir}/mavenexec</junitCoverDir>
        <jacocoPort>6300</jacocoPort>
      <jacocoAddress>10.45.14.34</jacocoAddress>
    </systemPropertyVariables>
  </configuration>
</plugin>
```

TestCase 关联单个应用的实践

- Fitnesse及其他各种开源工具
  为了精准：
  需要自定义服务动作，在set up 中清除干扰数据
  在tear down 中获取用例完整关联的覆盖率数据



Contents:

- According To The File Database Update * ：按文件进行数据库更新
  - According To The File Database Update + ：数据库更新节点支持取其它流程制作版本生成的文件
  - Set Up
- Code Check * ：代码检查分层测试用例集合
  - Check Style + ：java checkstyle 节点分层测试用例
- Mail Test * ：zcip 平台 关于邮件模块的测试验证功能组
  - Mail Title Display Success Or Failure + ：邮件标题显示成功或失败
  - Set Up ：邮件功能分层测试准备条件
  - Tear Down ：邮件功能分层测试用例，清理工作
- Test Name + ：ddd大得多 ggg

TestCase 关联N个应用的实践

但是，在大规模应用中是否也能如此简单，有没有好的集成解决方案？

TestCase → Application-1
TestCase → Application-2
TestCase → Application-3
TestCase → Application-4

TestCase 关联N个应用的实践

- 一个TestCase关联多个应用，如何串联？
- 大量不同版本进行自动化回归，有没有统一的管理视图？
- 个性化需求
- CI&CD怎么集成精准，构建过程基础数据持续更新？

TestCase 关联N个应用的实践

01

个性化订制Jacoco,在TcpClient模式下启用注册监听机制，注册到jacocoServer

02

通过CI&CD平台或者其他流程控制平台，管理Job和应用的关联关系

03

jacocoServer提供覆盖率搜集服务和覆盖率报表服务，以及质量度量和分析服务

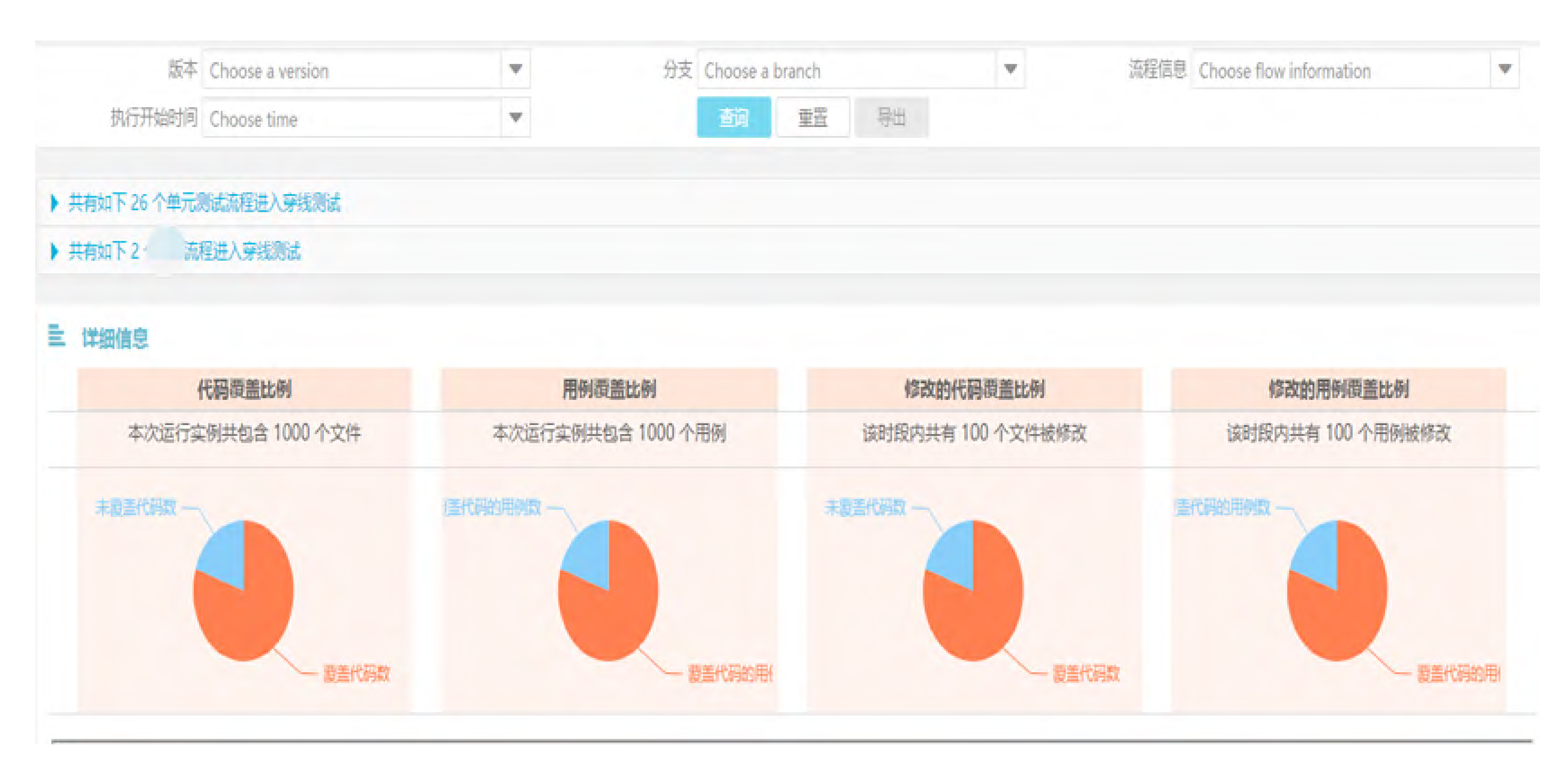
04

测试管理平台在TestCase执行完成后，发起覆盖率数据收集

TestCase 关联N个应用的实践

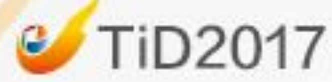
TiD2017

精准管理中心



code 与 case 精准关联

case和code精准关联数据建立了，我们可以做很多



精准测试在CI&CD中的价值

- 引入精准测试前后每次迭代的执行用例数和耗时

	引入前	引入后		
时间（小时）	16	1		
用例数	1000	60		

精准测试在CI&CD中的应用

- 提高大型项目的测试用例回归效率
- 根据版本变化动态甄别关联的自动化回归测试用例
- 辅助人工定位回归范围，降低人工选择测试的盲点
- CI&CD中的风险预警
- 双向回溯，确定版本迭代的影响范围

测试用例重构

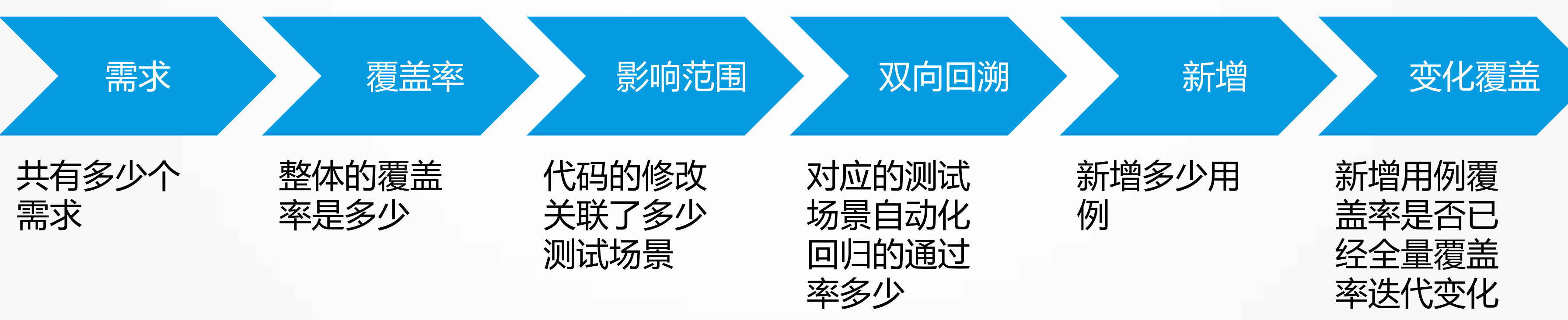
为什么要测试用例重构？

- 精准关联的依旧有很多，执行效率低下，无法实现快速迭代。
- 自动化测试用例后期维护工作，假设有2000个用例，每次自动化回归由于各种不稳定因素导致5%的错误，那么就有100个用例，每个错误用例分析需要2分钟/人，每次迭代就需要200分钟/人。
- 版本的不断重构和演进，有些老的用例可能已经没有什么重要的价值。

测试用例重构方法

- 基于测试用例覆盖率数据，精准比对发现包含或重叠的测试用例，然后再进行人工甄别
- 人工review，或是重写部分覆盖率用例，删除部分用例集，通过覆盖率数据比对来做校验
- 两个方法，一个覆盖90%和一个覆盖率10%对比两个50%，这两种哪个意义更大 ，正常的路径都被覆盖了
- 因为每个case有清晰的code关联数据，所以在删除时更有信心

全链路集成



限制

- 类被asm的等字节码工具动态的修改了
- Mock mock导致的数据无法收集
- 存在污染数据，比如常驻进程
- 第三方的操作污染
- 无法直接针对某次覆盖信息进行调用链路顺序分析

还有哪方面的应用

数据在这里，如何解决实际研发流程中的痛点，需要大家共同挖掘和探讨

TiD2017

谢谢

程海明

ZTEsoft中兴软创

软件创造价值