07 | 如何基于流量回放实现读服务的自动化测试回归?

在本模块的前四讲里，我向你介绍了可以直接落地的、能够支撑百万并发的读服务的系统架构，包含懒加载缓存、全量缓存，以及数据同步等方案的技术细节。

基于上述方案及细节，你可以直接对你所负责的读服务进行架构升级，将性能进一步提升。在升级系统架构时，有一个很重要的点容易被研发同学忽略——**只评估了升级的工作量，就直接开干**。最后提测的时候，发现**改造范围太大，测试根本回归不完，升级重构上线遥遥无期**。

随着业务的发展，系统相应地进行升级重构是不可避免的，但与此同时也带来了巨大的测试回归量。在本讲里，我将介绍一种**自动化的测试回归架构方案**，它能够极大地降低读业务因升级重构而带来的回归问题。此外，还能适配日常需求上线的回归工作量，真正**做到了回归自动化、研发自助化，避免用户场景全覆盖遗留导致漏测的问题**。

为什么要自动化?

首先我们来看针对架构升级的场景。 不管是因为技术还是业务导致的系统重构架构升级，它的改造量和范围都是非常大的。对于此类系统级的重构，测试回归的工作量至少都是以月为单位，对于人力的消耗巨大。一种应对方案是，先不改造，到系统实在扛不住了再想办法。另一种应对方案是，先暂停需求，全力进行改造。但在实际工作场景中，上述应对策略往往很难实现。

再来看看针对日常需求的场景。 对于后台系统，基本上都是微服务架构，对外会提供一至多个接口。一般一个需求只会涉及一个或者几个接口的某些场景的修改，测试同学也只会对改造的接口及对应的场景进行测试。

但在实际案例里，需求涉及的接口或场景上线后均不会出现 Bug，但同一个服务里的其他接口，即当次未涉及修改的被测接口，就比较容易出现 Bug。因为虽然对外的接口是不同的，但底层的逻辑很可能是复用了相同的代码，导致相互影响。为了避免此情况，对于任何一个需求，测试同学都需要回归所有接口，这个工作量是巨大的。

针对上述问题，通过自动化的方式提升效率便是我们的不二选择。

如何实现自动化回归?

首先来看一下自动化回归方案的**基本原理**。读服务能够实现测试回归自动化有两个前置条件。

1. 读服务均是查询，它是无状态的。

无状态是指每次请求都是**无副作用的**、**可以重复的**。同样的请求入参，在后台数据无变化的情况下，多次重试的结果均一样。相比而言，写或者扣减业务就不行。比如用户的余额变化就会产生副作用。具体来说，上一次购物订单支付成功，第二次和上一次使用同样的金额进行支付就不一定能成功。因为上一次的支付产生了副作用，用户的余额减少了，导致此次支付不成功。

2. 不管是架构升级还是日常需求，读服务对外接口的出入参格式是没有变化的。

不管服务内部的逻辑如何变化，只要接口出入参格式不变（下图 1 中标记的 1、2 表示架构升级，但对外接口未变），就可以利用读服务的无状态特性进行流量回放。

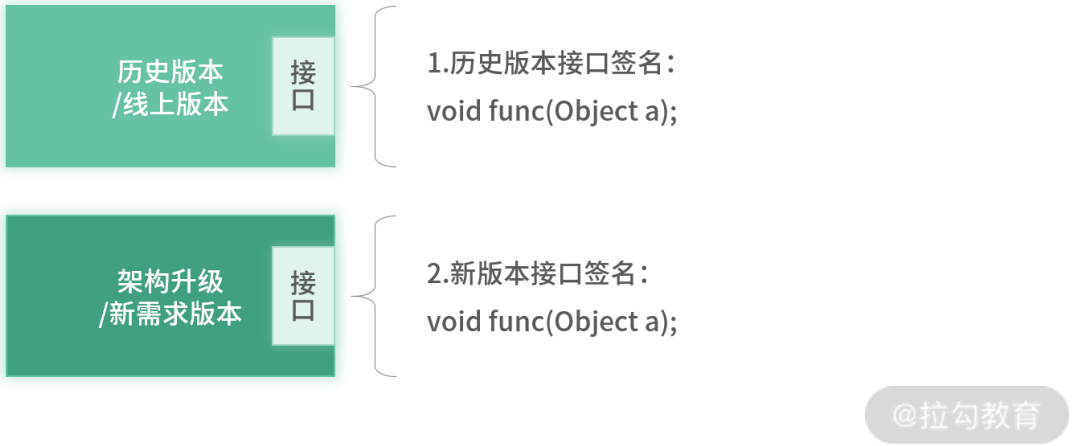

图 1：新老版本的接口未变架构图

整体架构

下图 2 是读服务的自动化测试回归的整体架构：

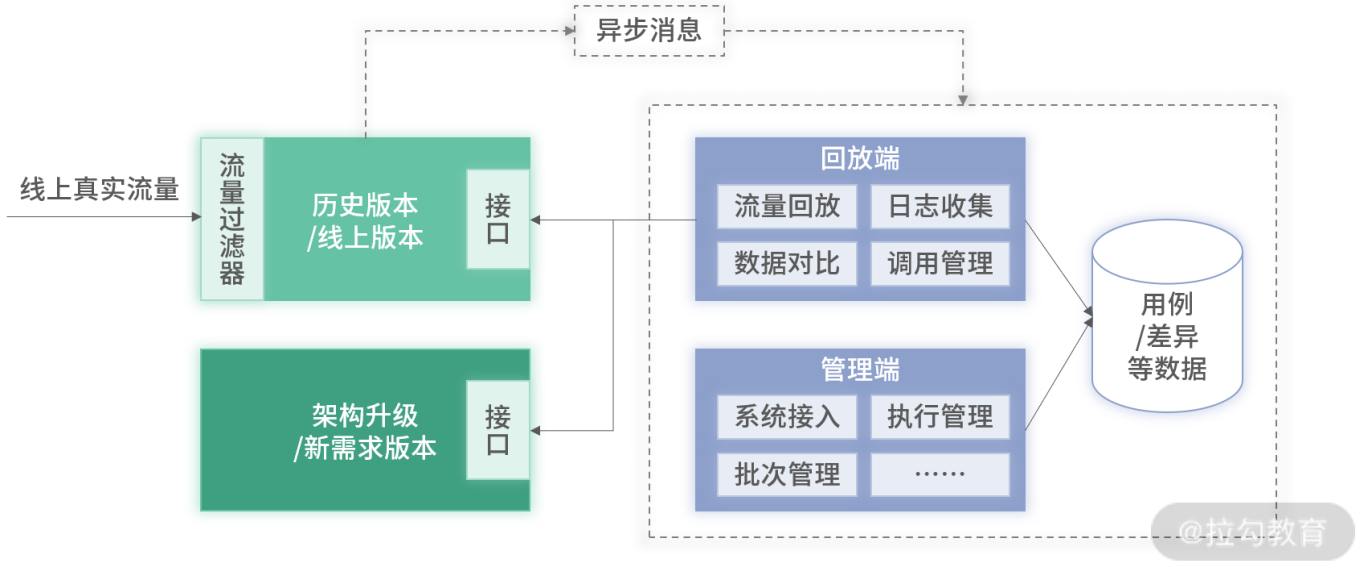

图 2：自动化测试回归整体架构图

在这个架构里包含三大模块，分别为**日志收集、数据回放，以及差异对比**。它们主要有以下 3 个功能：

1. 日志收集，主要作用是收集被测系统的真实用户请求，基于一定规则处理后作为系统用例；
2. 数据回放是基于收集的用例，对被测系统进行数据回放，发起自动化测试回归；
3. 差异对比，类似人工测试发现 Bug 的过程，通过差异对比自动发现与预期不一致的用例，进而确定 Bug。

日志收集

不管是提供 HTTP 形式还是 RPC 形式接口的后台读服务，都可以通过现在比较成熟的过滤器进行日志收集，比如 Spring 里的 Interceptor 过滤器、Servlet 里的 Filter 过滤器等。采用过滤器的架构如下图 3 所示：

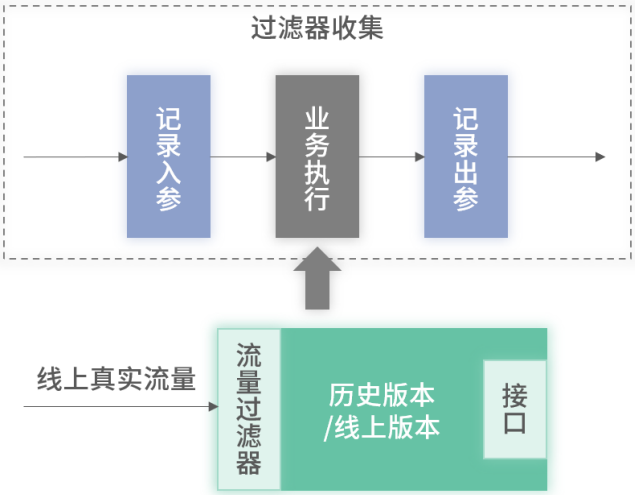

图 3：基于过滤器的日志收集架构图

在过滤器中，会对所有请求的入参和出参进行记录，并通过 MQ 发送出去。记录的数据格式类似如下：

```
{"应用名":"XXX","接口方法名":"RPC记录的接口方法名/HTTP记录域名及路径","入参":"XXX","出参":"XXX"}
```

对于发送出去的 MQ，自动化回归的消费服务会按应用和接口进行处理。对于应用和接口方法的标识数据可以存储在数据库里，对于入参和出参可以存储在 NoSQL 里，如 HBase。之所以将出入参数据存储在 HBase 里，是因为入参和出参数据量较大，存储在数据库里查询性能会比较差。

即使将参数存储在原生支持分布式的 NoSQL 里，也要对采集的日志进行一些过滤、去重以及无效数据定期清理等操作。此外，对于压测等非业务场景，需要关闭日志采集。毕竟，日常业务请求带来的数据量可能一天都上亿或十几亿次，如果不做管控，对于存储的消耗将是巨大的。

当前日志收集采用的过滤器是和业务应用同属于一个进程，它会占用业务应用的内存资源，同时对于业务也存在一定的侵入性。当遇到此种情况，可以将日志收集独立出来，采用单独进程进行运行。升级后的架构如下图 4 所示：

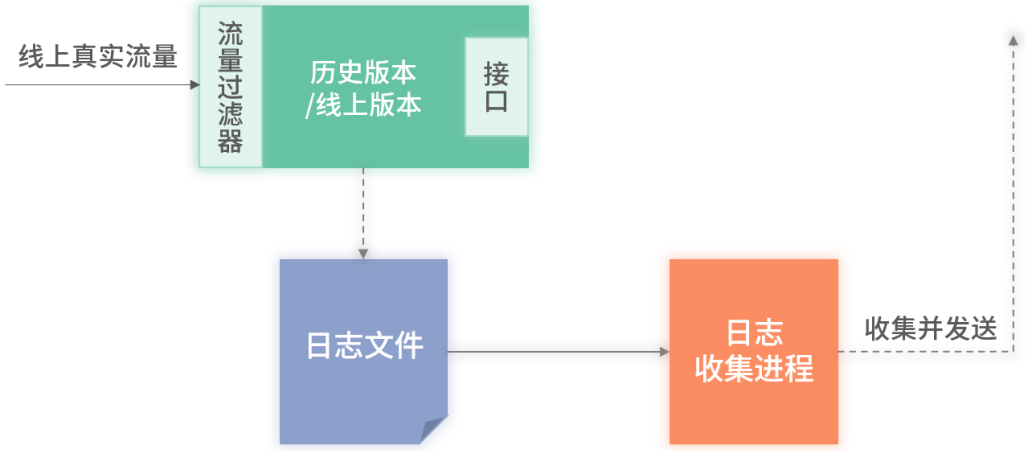

图 4：单独进程的日志收集架构图

采用单独进程后，业务应用需要按上述格式将出入参日志打印到指定文件。单独的数据收集进程只需要对此文件进行监控并将变化数据发送至 MQ 即可。在操作系统里，可以单独对进程设置资源占用的限制。为了保证业务应用不被日志采集所影响，可以对采集进程设置内存、CPU 等限制，并配置资源占用报警等。

数据回放

有了上面的出入参之后，便可以在测试时进行回放了。**数据回放主要的作用是基于日志里记录的接口信息（HTTP 便为域名）和入参，去实时调用被测系统，并存储实时回放返回的出参信息**。整体架构如下图 5 所示：

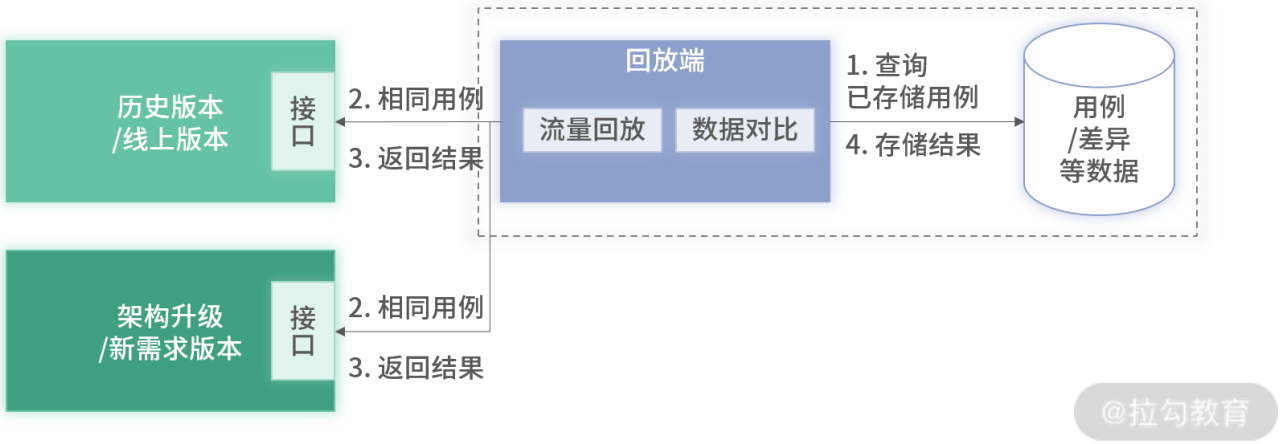

图 5：实时数据回放架构图

实时回放时，如果是 HTTP 形式的接口，采用主流的 HTTP 客户端即可。如果是 RPC 形式接口，需要使用该 RPC 框架提供的能够调用任意被测接口的客户端。

差异对比

在完成了数据回放后，便可以对回放产生的结果数据与预期数据（比如收集日志里的出参）进行比较。数据对比有很多形式，比如基于二进制校验和、基于文本等。二进制校验和的方式只是告诉你数据是否整体一致，而不会展示具体哪里不一致。即使展示了，但因为是二进制，人工也无法查看。

此处数据对比采用文本形式，先将数据转换为 JSON 格式，再进行对比。使用文本格式，可以看到数据整体不一致，以及具体是何处导致的不一致。如下图 6 所示：

```
{
    "name": "一本书",
    "sku": 123
}
```

```
{
    "name": "一本书吗",
    "sku": 123
}
```


图 6：返回数据对比图

可以看到，同一个 SKU 的名称，在两边出现了不一致。如果只是进行了系统重构，相同的 SKU 对应的数据应该是一样，此处出现了不一致，代表出现了 Bug。**采用文本对比，可以直观地看到哪个字段数据有差异，从而更快定位到问题**。

正常情况下，只要存在差异的数据，均可认为是 Bug，是需要进行修复的。但有些时候，比如接口会对每次请求产生一个唯一标识，用此标识来打印日志并返回给调用方。在差异对比时，此类接口每次都会因为唯一标识不同而导致对比出现差异，但此差异又不会对业务产生影响。对于此类场景，需要差异对比工具具备忽略能力，在对比时忽略唯一标识字段。

通过数据收集、回放和差异对比，能够实现自动化、可视化，将读业务的测试回归完全自动化。

回放的三种模式

上述讲解了自动化测试的整体流程和实现步骤，关于回放，我们只讲解了技术实现，接下来我再介绍几种需要你重点关注的模式。

1. 离线回放模式

离线回放模式是指在回放时只调用进行改造的新服务，将新服务返回的数据与收集日志里的出参进行比较。架构如下图 7 所示：

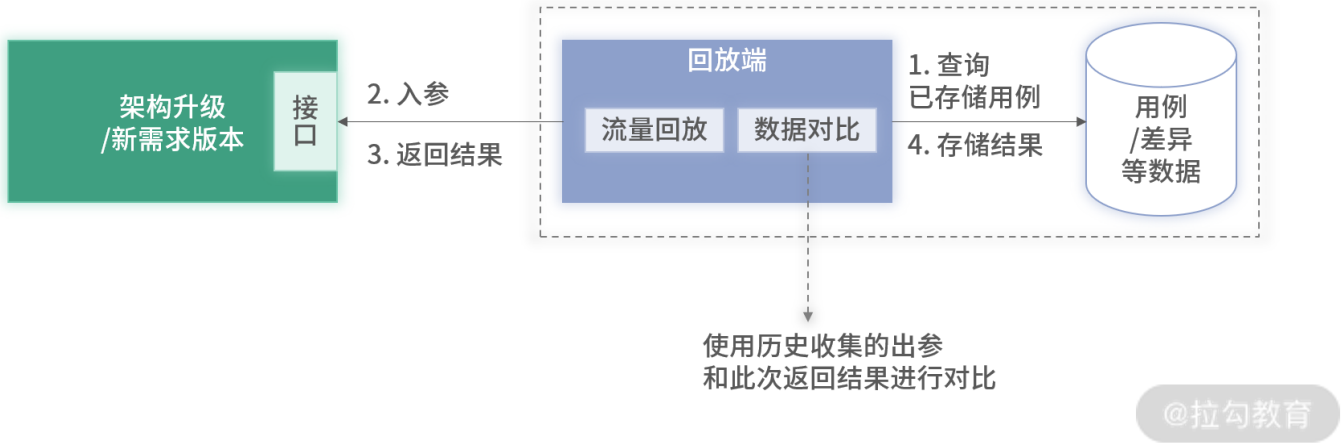

图 7：离线回放模式架构图

采用离线回放的好处是，减少了对于线上老版本的调用量，避免对线上产生影响也节约了资源。但存在一个问题，如果后台存储中的数据已经发生了变化，此时就不能使用收集的日志里的出参。因为从新版本实时查出的数据与历史收集的日志数据已经不准了。

2. 实时回放模式

为了解决离线回放模式里，因为数据变化导致收集的日志里的出参无效问题，可以采用实时回放的模式，如下图 8 所示的架构图。上述架构在收集的日志里，只记录入参而不记录出参，收集流程见下图 8 中的标记 3。

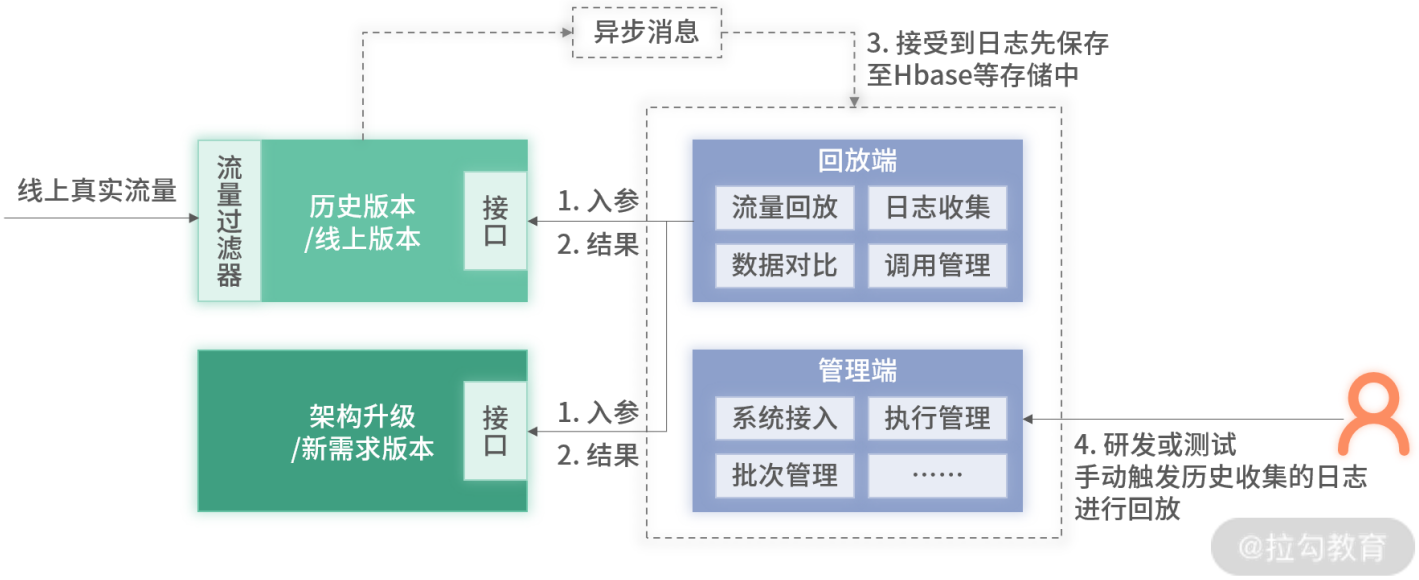

图 8：基于录制的实时回放模式

实时回放的模式会在上图 8 中的标记 4，研发或测试手动触发回放功能后，使用入参实时的调用新老版本的被测应用，并对比双方返回的出参，通过此方式可以规避数据变更的问题。

3. 无录制的实时回放模式

不管是离线回放还是实时回放都存在一个问题，我们是对接口的入参进行录制（存储至 NoSQL 里，如上图 8 里的 HBase）再回放的。因存储容量有限，只能存储一定数量的数据，很多日志用例可能会被丢弃。这就可能导致有些重要场景会被漏测。针对这个问题，可以采用无录制的实时回放模式，架构如下图 9 所示：

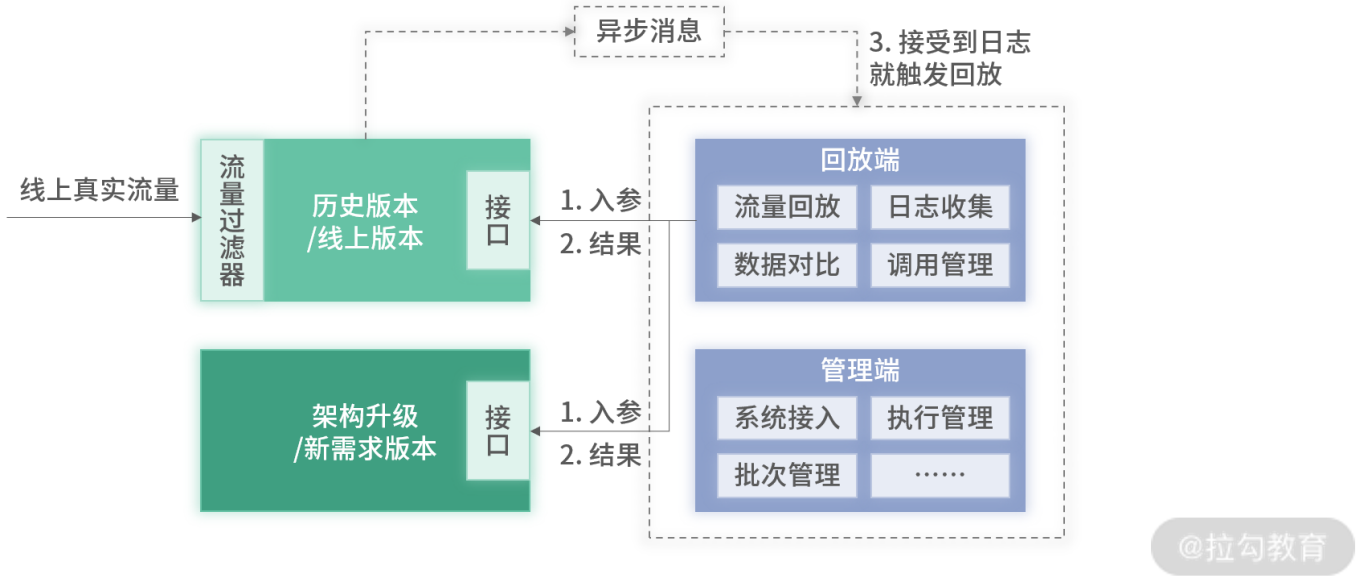

图 9：无录制的实时回放模式

无录制实时回放不再记录入参数据了（见图 9 和图 8 里的标记 3 的差异）。当日志消费模块接收到收集的日志用例后，实时调用新老版本被测服务并进行数据对比。使用此方式，进行几周或者更久的回放，基本能够覆盖全部场景了。

下面我们再来总结一下**注意事项**。在线上部署及使用自动化回归工具时，也有一些需要注意与规避的点。

1. **在进行自动化回归时，写接口一定要屏蔽**。以注册用户举例，如果在回放时没有屏蔽，使用线上入参进行回放，将会产生很多垃圾用户，给后续的业务流程带来巨大影响。
2. 上述几种回放模式里，**除了离线模式外，实时回放模式和无录制的实时回放模式都会对线上系统产生一定的流量压力**。假设被测系统的性能比较差或者机器数较少，自动化回放的流量会把线上系统打挂，进而影响业务不可用。特别无录制的实时回放模式，带来的流量更大且持续时间更长。
3. 即使采用了无录制的实时回放模式，也只是通过更长时间的回放尽可能地覆盖更多的业务场景。但也并没有足够的证据表明，一定不会出现漏测。对于此种问题，**可以借助一些代码覆盖率的工具，如 Java 里的 JaCoCo，来统计一次回放后被测系统的代码覆盖率，通过数据来判断是否存在可能的漏测**。

总结

本讲介绍了读业务在系统重构及日常需求开发时，均存在的**测试回归耗时长**和**可能存在漏测**的问题。根据读业务无状态及可重复执行的特点，针对上述问题我们构建了一套**基于业务日志的自动化回归平台，主要包含三大子模块，分别是日志采集、数据回放及差异对比**。

希望你学习完本讲内容之后，对比你所在的团队，思考是否存在测试回归耗时长及漏测导致线上问题的场景。如果有的话，可以考虑尝试采用此方案及其变种，来提升你所在团队的效率。

最后，留一道思考题给你。本讲介绍的方案是基于读回放场景，请你思考写回放和读回放有什么区别？欢迎在留言区留言，我们一起讨论。

下一讲我们介绍 08 | 如何使用分库分表支持海量数据的写入？