下文以dubbo2.5.10 升级到2.7.3为例子。

案例来自携程

Ps:不要升级到2.7.0~1，这两个版本的bug较多。比如，向zk注册用永久节点。

# 升级 2.7.3 的几个理由

2.5.10 版本在携程只用了一年半左右，业务的应用也不算很多，这么快就大版本升级，主要是遇到了下面的几个问题。

## 2.5.10 的异步也是有阻塞的

2.5.10 版本只支持客户端异步，而且是基于 JDK 1.6 的 Future，并不是真正意义上的异步，本质上还是阻塞的，只不过是从 DubboClientHandler 线程切换到了业务线程。

## 支持服务端异步

对于微服务来说，一般又会调用外部服务，在网络 IO 比较多的场景下异步服务的优势会很明显，可以充分利用 CPU 资源，提高系统吞吐量，降低响应时间。部分机票、酒店等业务同学明确表示需要服务端异步。

## 现阶段不兼容问题带来的副作用较小

不兼容问题大概率是服务端和客户端的版本不一致，比如服务端 2.7.0，客户端 2.5.10。考虑到携程目前 CDubbo 服务的比例还不太高，早一点升级对业务的影响会比较小。

## 为之后的升级做铺垫

携程业务场景很广泛，部分业务已经明确表示需要 2.7.0 的服务端异步，也有业务在尝试 3.0 的 Reactive 了。如果先升级到 2.7.0，以后再升级 3.0 会比较容易些，如果直接从 2.5.10 升级到 3.0 版本，可能升级不过去，或者无法透明升级。

## 支持三中心

2.5.10 只有注册中心，注册数据和配置数据对注册中心的压力比较大。2.7.0 对模型重构，拆分成注册中心、元数据中心、配置中心，职责划分更合理。为了接入公司的测试平台，需要用到服务的元数据信息，2.7.0 正好提供了这个能力。

# 第一阶段升级及踩坑历程

注：为了表达方便，后续提到的 Apache 代表 org.apache 的 package，Alibaba 代表 com.alibaba 的 package。

第一阶段升级 2.7.0 是在 2019 年 3 月份左右，大概花了三周的时间，我们先来看下所遇到的几个问题吧。

## 变更 package 导致的不兼容

2.7.0 把 package 从 com.alibaba 改成了 org.apache，虽然对低版本做了兼容，但是还是会发现部分 class 找不到了，例如：Alibaba 的 DubboComponentScan 就已经被删掉了。取而代之的是 Apache 的 DubboComponentScan，不过这个问题在编译时就会报错了。

## Apache 的 Constants 常量类被拆分

升级到 2.7.0 版本之后，Alibaba package 的 Constants 还是没变，但是如果要用新功能升级到 Apache package，你会发现 Constants 被拆分成 RegistryConstants, CommonConstants, RemotingConstants 等多个常量类。新的常量类只是分散到不同的 class 中，只要换个引用就可以解决了。

## Apache 的 Router 接口新增了部分方法

如果扩展是基于 Alibaba 的 Router 接口，Dubbo 已经做了默认实现，应该不会存在兼容性问题。这次，我们直接换成了 Apache 的 Router 接口，因为新加了 isRuntime、isForce、getPriority 方法编译时就报错了。

## Apache 的 ProxyFactory 接口新增了 getProxy 方法

我们这次升级是把 Alibaba 的 ProxyFactory 换到了 Apache 的 package 下，2.7.0 版本中对该接口新增了 getProxy 方法，编译时会报错。如果不需要扩展这部分功能，可以通过 delegate 机制保留默认实现就可以了。

## 限制 ApplicationConfig 必须全局唯一

2.5.10 版本对于 ApplicationConfig 没有限制，服务端起多个服务时可以配置独立的 ApplicationConfig。但是从 2.7.0 开始 ApplicationConfig 就会要求全局唯一，如果一个应用定义了多个不同的 ApplicationConfig 就会报错。

Apache 的 ConfigManager 的 setApplication 会检查是否 duplicate



## JDK 1.8

2.7.0 为了支持真正的异步，用到了 JDK 1.8 的 CompletableFuture，也用到了 1.8 的 Supplier、Consumer 等操作符。如果业务的应用还是基于 JDK 1.7 打包的，升级后就会导致发布失败。由于我们这次是公司层面的整体升级，就需要所有业务应用都升级到 1.8 才可以发布。

## 默认升级到 Netty4

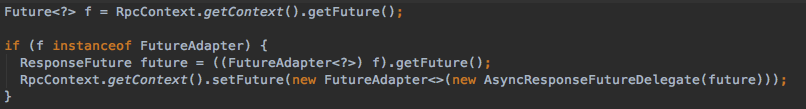
为了接入公司的 CAT 监控系统，需要把 Codec 的监控埋点数据通过 ThreadLocal 传递下去。

但是，2.7.0 把 Netty 的版本从默认的 Netty3 升级到了 Netty4，这两个版本的线程模型是不一样的，Netty3 的 decode 是在 New IO worker 线程，Netty4 是 NettyServerWorker 线程，导致原有逻辑的监控埋点数据传不过来。为了暂时解决这个问题，我们把默认的 Netty 版本降回了 Netty3。

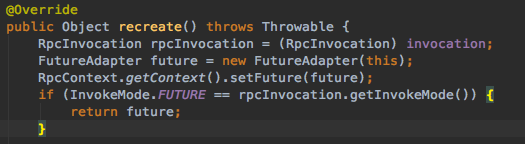
注：CAT 是点评开源的实时应用监控平台，目前在携程也有落地，在 Github （https://github.com/dianping/cat）上也已经超过 1 万颗星。

## 异步请求一直 hang 住

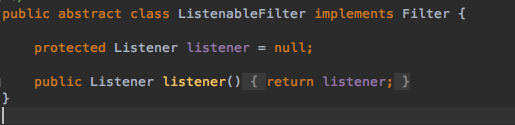
扩展 2.5.10 版本的时候，为了支持对客户端异步的埋点，我们对 RpcContext 的 Future 重新包装了，用户拿到的 Future 已经是被我们包装过的 FutureAdapter 了。



在 2.7.0 版本中， AsyncRpcResult 在 recreate 的时候也会给 RpcContext 设置 Future。导致用户拿到的 Future 跟实际的不是同一个，客户端一直拿不到响应，请求被 hang 住。



2.7.0 对这部分的重构很好，支持了异步 Filter 链，通过 ListenableFilter 回调机制比现在的代码结构更清晰，可以把同步和异步埋点的逻辑进行统一整合。



## 服务端无法指定客户端的调用方式

Issue：https://github.com/apache/dubbo/issues/3650

如果服务端设置了默认 ASYNC，升级到 2.7.0 版本后客户端会拿不到响应。例如：服务端配置了 async=true，客户端默认配置。

<dubbo:service interface="..." async="true">

<dubbo:reference interface="...">

2.5.10 版本的客户端，通过 client.sayHello() 会返回 null，RpcContext 的 Future 可以拿到响应。

基于 2.7.0 版本测试下来， client.sayHello 拿到了响应，但是 RpcContext 的 Future 却是 null。

经过研究发现，2.7.0 版本在 ClusterUtils 的 mergeUrl 过程中把服务端传递过来的 ASYNC\_KEY 给删掉了，所以客户端仍然以同步方式去调用。

这个是新老版本兼容性的 Bug，已经在 2.7.2 修复了，验证下来没问题了。

## @Service 注解无法设置 parameters 参数

Issue：https://github.com/apache/dubbo/issues/3778

用户通过 Annotation 方式启动服务，在 @Service 注解的 parameters 属性，服务端启动的时候拿不到用户配置的参数。

@Service(parameters = {"someKey","someValue"})

public class DemoServiceImpl implements DemoService {

}

并且报了下面这样的错误。

Caused by:

org.springframework.beans.ConversionNotSupportedException:

Failed to convert property value of type 'java.lang.String[]' to required type 'java.util.Map' for property 'parameters';

nested exception is java.lang.IllegalStateException:

Cannot convert value of type 'java.lang.String[]' to required type 'java.util.Map' for property 'parameters':

no matching editors or conversion strategy found

2.7.0 版本对这部分机制进行了重构，BeanDefinitionBuilder 把这段 parameters 的参数转换代码给漏掉了。加上这段逻辑之后，我们测试下来已经 OK 了，这个问题已经在 2.7.2 解决掉了。

private AbstractBeanDefinition buildServiceBeanDefinition() {

BeanDefinitionBuilder builder =rootBeanDefinition(ServiceBean.class);

...

// Convert parameters into map

builder.addPropertyValue("parameters",convertParameters(serviceAnnotationAttributes.getStringArray("parameters")));

}

## 当客户端发现服务时出现异常，即使服务端启动后也不会恢复

Issue：https://github.com/apache/dubbo/issues/4068

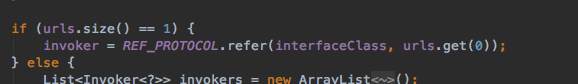
API 方式比 XML 和 Annotation 更灵活，可以在不重启进程的情况下多次初始化客户端。

服务端没有启动的情况下，通过 API 的方式启动了客户端，这个时候客户端会报 No Provider 的错误。然后启动服务端，客户端通过 API 的方式再次初始化，仍然会报 No Provider 的错误。

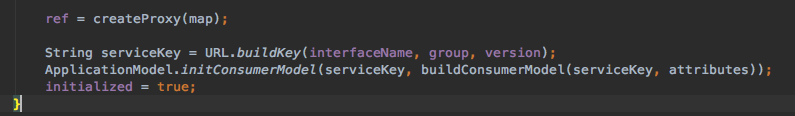
ReferenceConfigCache cache = ReferenceConfigCache.getCache();

DemoService demoService = (DemoService) cache.get(reference);

通过翻阅 ReferenceConfig 的代码，服务发现的时候可能会抛异常导致直接跳出 init 过程，但是 initialized 标志位已经被置为 true 了，导致下次不会再重新初始化。



修复方案：只有在 init 方法的最后，客户端代理创建完成才会设置 initialized 为 true。



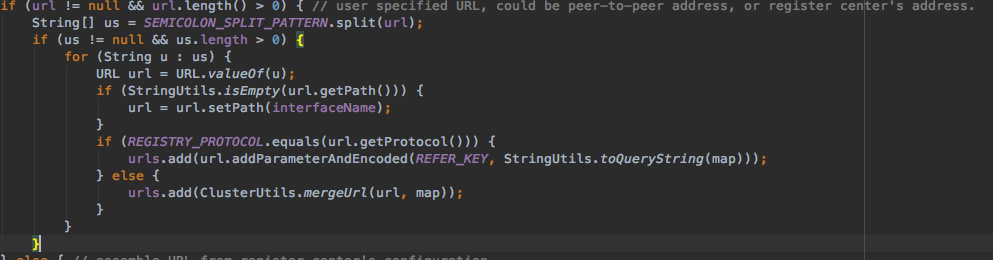
这个问题已经在 2.7.2 版本修复，验证下来已经 OK 了。

## 客户端服务发现失败，重试会有 OOM 的风险

Issue：https://github.com/apache/dubbo/issues/4107

如果服务端没有启动的情况下启动了客户端，客户端会报 No Provider 的错误，如果一直不停的重试可能会有 OOM 的风险。

Dubbo 在创建代理的时候会缓存 urls，每次启动失败都会把 url 加到 urls，但是由于 dubbo 的 URL 是有时间戳的，就导致 urls 队列不停的增长，甚至引起 Heap OOM 的风险。



解决方案：每次创建代理之前，都把 urls 给予清空，这个问题已经在 2.7.2 中解决了。

总结：第一轮升级过程中大概历时 3 周左右，发现的几个 Issue 导致我们的 Test Case 无法继续下去，升级过程暂停了两三个月。

# 第二阶段升级及踩坑历程

直到六月中旬，阿里团队把上述几个问题在 2.7.2 修复了，我们重新开始了第二轮的升级过程。

性能测试，吞吐量下降了 40%

服务端：8C24G 的物理机，响应报文大小为 10Bytes，queue 设置为 -1 无界队列。

客户端：10 台 4C8G 的 Docker，请求报文大小也是 10Bytes。

基于原生的 2.5.10 版本，我们的压测环境下可以达到 8 万 QPS 左右。由于 CDubbo 扩展了熔断、配置、监控等功能，吞吐量下降到 5.5 万 QPS 左右。

升级到 2.7.2 版本后，最高只压测到 3 万多，吞吐量下降了差不多 40% 左右。

这个问题是因为 JDK 1.8 的 Bug 导致的，JDK 1.8 的 CompletableFuture 在 get 时会等到 256 次 countDown 执行完毕，影响了性能。

Issue：https://github.com/apache/dubbo/issues/4279

总结：第二阶段遇到的性能下降的问题肯定要解决后才可以上线，问题反馈给阿里团队后，他们需要讨论新的 Hotfix 发布机制。

# 第三阶段升级及踩坑历程

这次改变了合作模式，跟阿里团队基于 2.7.3-SNAPSHOT 版本一起讨论，一起修复，一起验证。下面的几个问题都是基于 SNAPSHOT 验证过程中发现的问题，并且在正式版中修复掉了。

## 对于同步的请求，方法级超时不生效

Issue：https://github.com/apache/dubbo/issues/4435

如果服务级设置的 timeout 为 1000ms，sayHello 方法设置的 timeout 为 800ms。理论上来说，sayHello 方法的请求应该在 800ms 就会超时了，但是实际上我们发现直到 1000ms 才会超时。

<dubbo:reference id="demoService" interface="com.ctrip.Demo" timeout="1000">

<dubbo:method name="sayHello">

<dubbo:parameter key="timeout" value="800"/>

</dubbo:method>

</dubbo:reference>

同步的请求是在 AsyncToSyncInvoker 中执行了同步等待，修复前的代码如下，取的是整个服务的超时时间，也就是 1000ms。

asyncResult.get(getUrl().getParameter(TIMEOUT\_KEY, DEFAULT\_TIMEOUT), TimeUnit.MILLISECONDS);

解决方案：同步请求，除了 AsyncToSyncInvoker 在 get 时被设置了超时时间，DubboInvoker 的 CompletableFuture 也被设置了超时时间。其实，只要一个地方能够超时就足够了，所以 AsyncToSyncInvoker 被设置到 Integer.MAX\_VALUE 永不超时，所有的超时机制都通过 CompletableFuture 实现。

## 异步超时的情况下，不会回调 listener 的 onError 方法，导致埋点丢失

Issue：https://github.com/apache/dubbo/issues/4152

https://github.com/apache/dubbo/issues/4306

在修复前的版本中，ProtocolFilterWrapper 的 Filter 链中，只处理了正常的 onResponse 响应，并没有处理 onError 情况，就导致异常发生时不会回调 ListenableFilter 的 onError 方法。

修复后，会对正常响应和异常响应进行回调。

try {

if (t == null) {

listener.onResponse(r, filterInvoker, invocation);

} else {

listener.onError(t, filterInvoker, invocation);

}

} catch (Throwable filterError) {

t = filterError;

}

## @Reference 注解的方式，客户端不会初始化

Issue：https://github.com/apache/dubbo/issues/4330

基于 2.7.2 版本，如果用的 Annotation 的方式，先要把 DubboComponentScan 换成 Apache 的，不然编译时就会因为找不到 class 而报错 。

如果客户端的 @Reference 用的还是 Alibaba 的 package，所拿到的 proxy 代理是 null，导致 service.sayHello 调用时抛 NPE 的 exception。



这个问题，主要是由于 Apache 的 DubboComponentScan 没有兼容 Alibaba 的 @Reference 注解，目前 2.7.3 正式版实现了对 Alibaba 的 @Reference 和 @Service 的兼容。

## 服务端 executes 限流失效

Issue：https://github.com/apache/dubbo/issues/4277

我们的测试场景把服务的 executes 设置为 1，然后客户端多线程发起请求到服务端。第一次发起的多线程请求，只有一个请求能通过，符合预期。第二次再发起多线程请求，所有请求都通过了，并没有被限流。

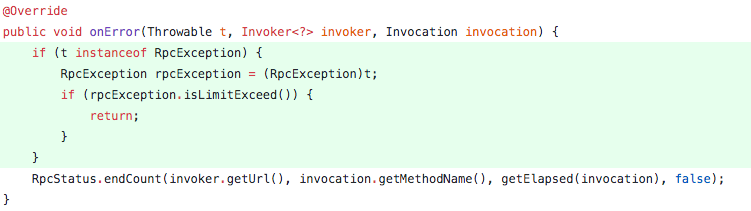
<bean id="demoService" class="com.xxx."/>

<dubbo:service interface="com.xxx" ref="demoService" executes="1">

</dubbo:service>

这是因为服务端抛异常的时候，除了正常请求结束后释放掉的计数器，异常处理时又减了一次，之后的限流一直处于失效的状态，所有请求都可以通过了。

这个问题已经在 2.7.3 解决了，解决方案就是在 onError 的时候不要重复减。



## 现有服务框架生成的 ListenableFutre 异步服务接口，Dubbo 无法支持

携程现有几千个 SOA 服务，服务端异步用的是 Guava 的 ListenableFuture，但是 2.7.0 支持的服务端异步用的是 CompletableFuture，这就导致现有服务接口迁移过来，无法支持 Dubbo 协议的服务端异步了。

针对这个问题，我们想到了几个方案。

方案 1：让 Dubbo 既支持 CompletableFuture 又支持 ListenableFuture

首先，需要 Dubbo 支持 ListenableFuture，这个改动成本比较高。其次，对用户多一个选择也会提高他们的学习成本，以及犯错的概率。

方案 2：只支持 CompletableFuture

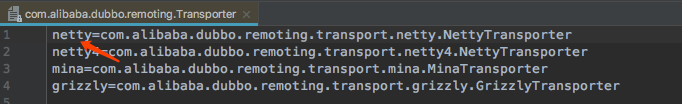
如果用户从 SOA 服务迁移到 CDubbo 框架，就需要把服务接口的 Future 类型改为 CompletableFuture。

最终跟业务沟通下来，选择了方案 2，业务迁移到 CDubbo 的时候手工修改服务接口的 Future 类型。

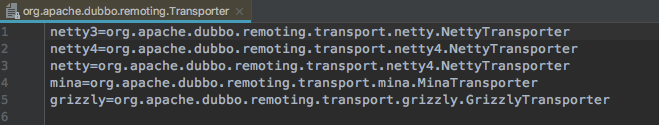
## 服务端新版本，客户端老版本，报 Netty3 找不到的异常

这个问题的根因是前面监控打点失败，我们把 Netty 默认版本降回了 Netty3。服务端 2.7.3 版本，客户端 2.5.10 版本的情况下会报 Netty3 找不到的异常。

在 2.5.10 版本中， Netty3 在 resource 配置文件中的名字叫 netty，具体如下图：



但是，2.7.3 版本把 Netty3 在 resource 配置文件中的名字改成了 netty3，而不是 netty 了。



服务端注册的时候会包括 Netty 版本，通过注册中心推送到了客户端，客户端的 2.5.10 版本不存在 netty3 的资源文件，通过 SPI 加载的时候因为找不到 netty3 而报错了。

解决方案：Netty 的版本不应该被推送到客户端，我们修改了动态配置的推送规则，不允许 Netty 参数推送到客户端，问题就解决了。

# 兼容性测试

第三轮测试把所有的 Test Case 都通过了，接着我们手工验证了新老版本的兼容性测试。以下场景都是基于服务端升级 2.7.3，客户端仍然是 2.5.10 场景下的测试验证。

## 注册发现机制

服务端可以正常注册到注册中心，客户端也可以发现到新版本的服务端。

## 同步请求是否正常

如果服务端返回的是 Response 对象，客户端以同步的方式可以正常调用。

## 异步请求是否正常

如果服务端返回的是 Response 对象，客户端以异步的方式可以正常调用。

## 监控打点

除了不支持 CompletableFuture，其他都正常。

## 服务端升级到新版本，客户端老版本，在超时场景下的异常测试

因为服务端抛的是 org.apache.dubbo.rpc.RpcException，这个 package 在 2.5.10 版本中是不存在的，就会报 java.lang.ClassNotFoundException 的错误。

这个错误似乎也没法避免，这也是我们优先升级 Dubbo 2.7.3 的原因，我们要忍受这种阵痛，等全部升级完就不存在这个问题了

# 性能压测

兼容性测试也通过了，我们紧接着开始了第二轮压力测试：(基于 2.7.3-SNAPSHOT)

从服务端的压测数据来看，在低于 4 万 QPS 的时候性能没啥区别，在 5 万左右的时候响应时间有所下降，主要是由于 YGC 导致的。

# 集成测试

## ApplicationConfig 的冲突再次出现

前面只是解决了 CDubbo 单个组件的 ApplicationConfig 冲突问题，在一个组件中保证只会有一个 ApplicationConfig。但是，不同组件在暴露本地服务的时候也需要设置 ApplicationConfig，用户可能会只引用一个组件，也可能两个同时引用，无法保证不同组件只初始化一个 ApplicationConfig。

看了 ApplicationConfig 的 equals 方法，可以知道冲突是因为 name 不一致，我们只要保证 name 一致就行了。



## 开源和订制版的冲突

在携程，大部分业务用的是我们提供的开源版的 Dubbo，还有部分业务使用的是基于 Dubbo 代码直接修改过的订制版本。

因为，我们这次是公司级的升级，用了订制版 Dubbo 的应用，如果引入公司其他中间件，这些中间件又依赖了开源版的 Dubbo，就会导致业务的应用类冲突。

对于这个问题，没有统一的解决方案，需要跟业务同事进行讨论来解决。

## 服务端启动时端口连不上

Issue：https://github.com/apache/dubbo/issues/4775

在集成测试时，服务端用的默认协议，客户端通过 20880 端口发起的连接，结果客户端报连接失败。后来看了下服务端的 20880 端口的确没有打开，本地打开的端口号是 20xxx。

经过调试代码发现拿到的默认协议是 QSchedule 组件设置的 ProtocolConfig。看下 ConfigManager 的代码，addProtocol 的时候会把第一个协议作为默认协议缓存下来了，之后再 getDefaultProtocol 的时候拿到的并不是默认的协议了。

public void addProtocol(ProtocolConfig protocolConfig) {

…

if (protocols.containsKey(key)&& !protocolConfig.equals(protocols.get(key))) {

logger.warn("…");

} else {

protocols.put(key, protocolConfig);

}

}

这个问题可以把 ProtocolConfig 默认值设置为 false，就不会被 put 到 protocols 作为默认协议了。但是，对于不知道背景的同学可能还是会掉坑里，不过这个问题会在 2.7.4 版本中修复。

# 感兴趣的几个话题

写到这里，我们已经通过了 Test Case、回归测试、压力测试和集成测试，发布了 SNAPSHOT 版本给到业务同事去试用，预计九月初会发布正式版本。

除了我们踩到的坑，下面可能也是你感兴趣的话题。

## 注册中心

注册中心，我们在去年落地 2.5.10 的时候就扩展了携程自己的注册中心。

服务端注册实例信息到注册中心，每隔 5s 发送一次心跳来续约，如果注册中心 30s 没有收到心跳，会将其从注册中心反注册，并通知到客户端。

客户端向注册中心发起订阅，当注册信息发生变化时会通过长连接推送到客户端。

这套机制是基于 2.5.10 扩展的，在升级 2.7.3 的过程中没有任何变更，可以完全兼容 2.7.3，服务端和客户端都可以正常的注册和发现。

## 从一个中心拆分成三中心

1）注册中心：前面已经提到，升级 2.7.3 没有变更，可以完全兼容。

2）配置中心：CDubbo 的 0.2.0 版本，为了接入携程自己的配置中心，就已经通过 override 协议实现了动态配置方案，这套机制目前没有发现问题，所以这次没有对接 2.7.3 提供的动态配置推送能力。方案可以参考：

http://dubbo.apache.org/zh-cn/docs/user/demos/config-rule-deprecated.html

3）元数据中心：TCP 协议的测试不像 HTTP 协议那么方便，服务提供者必须得自己写个 Client 才能测试验证，大多数业务同事都反馈过这个痛点。

在 2019 年 1 月份的时候，CDubbo 对接了携程的测试平台，支持 Dubbo 协议的测试。当时为了透明升级 2.7.0 版本，就已经提前把元数据中心的代码拷贝到内部的版本了，这次升级 2.7.3 版本很平滑，没有发现有啥问题。

## 为什么敢于升级大版本

业界对大版本升级的普遍做法就是，等其他大厂试试看，或者等发布几个 hotfix 之后再考虑。携程在这次升级过程中有一套自己的保障，事实也证明我们的单元测试和集成测试在 2.7.3 升级过程中发挥了重要作用。

1）单元和集成测试覆盖率 93%：刚开始落地 CDubbo 的时候就非常重视测试覆盖率。目前为止，我们的测试覆盖率仍然达到了 93%。

在这次升级过程中，很多藏的很深的 Bug 都是通过我们的测试代码发现的，前面谈到的升级过程中遇到的 Bug 基本都是通过测试代码发现的。不但保证了质量，也提高了我们升级的效率。

2）Benchmark 压测：我们有一套稳定的压测环境，服务端是一台 8C24G 的物理机，客户端是 10 台 4C8G 的 Docker 机器。服务器都比较稳定，测试结果也能真实准确的反应出性能的问题。

每次发布新功能的时候，都要经过 Benchmark 至少 5 万 QPS 以上持续一天的稳定性压测。