深度解密京东中台底层支撑框架

导读：近几年，除AIGC外，软件领域相关比较大的变化，就是各相关业务领域开始如火如荼地建设中台和去中台化了。本文不探讨中台对公司组织架构涉及的变化和影响，只是从中台化演进的思路，及使用的底层支撑技术框架进行分析探讨，重点对中台及前台协作涉及到的扩展点及热部署包的底层技术细节，结合京东实际落地情况，对涉及的核心技术框架进行源码初探分析，探讨技术框架的考虑点，拓宽大家的思路，欢迎大家审阅。

1、序言

中台及其建设，区别于单体应用建设及微服务建设，最大的差异在于中台建设有一个较为独特的概念，即前台角色。中台建设并采用标准协议，开放了一系列标准丰富的能力供前台角色去编排、扩展使用。从外部的视角来看，其实看不到中台和前台，单纯还是功能交付，外界的感知没有太大的差异；从产研的视角来看，功能交付是中台能力叠加一系列前台个性化能力的结合物，职责上期望通过中台底层能力建设和前台个性化能力增强，两方独立开发，再通过底层技术支撑框架来将两方能力进行结合，期望达到中台、前台角色分工清晰，独立交付，提升交付速度的美好愿望。

从上述定义就可以看到，在外部视角感知不强的情况下，交付速度其实是个很明显的衡量指标。中台建设成功的标准，重点并不在于对中台建设的丰富能力进行衡量，也不在于前台开发了多少独立的扩展能力，仍在于同之前未建设中台相比，交付速度是否有质的提升。若这个关键指标没有变化，预估此类建设思路也会出现相关的变化及转型，转型的下一步思路和方法也有不少，不在这里探讨。

后续的内容，均会聚焦中台的底层技术支撑框架（后文简称为Matrix），来将中台、前台两方能力进行结合的技术细节和考虑点进行展开。

在笔者看来，Matrix框架，作为京东实施PaaS化相关的技术解决方案，期望的是建立划分合理的业务领域，完成业务建模及抽象，分离出核心逻辑及高频个性化业务，并将个性化的逻辑隔离出来，期望交给名为前台的组织或者部门去共建开发，来达成中台逻辑稳定，前台可以并行开发，最终提升交付效率的目的。

当然，在京东或者其他商业化公司，应用系统覆盖的业务领域及技术方向存在

多种，包括但不限于各端的APP等前台、各端的web前台、各应用后端系统、各数据算法平台、各报表运营平台等等。目前的Matrix框架对其他某些领域可能存在不适应的情况，在笔者看来，着实属于正常情况。世界上的工具万万千，刀枪剑戟 斧钺钩叉 闲棍槊棒 鞭锏锤抓；各类解决方案千千万，很难存在一个锤子可以砸所有的钉子。但是只要总体的理念是合适的，并为其他各类适配此理念的工具提供合适成长的土壤，预判此解决方案可以逐步丰满成熟。

2、底层支撑框架分析
==========

分析前提：我们只聚焦与中台底层支撑框架，其他如中台的业务领域切分、领域建模等内容暂不涉及。需要注意的是，本次原理探究，是从使用人员及业务应用的角度出发，而非从真正对Matrix设计者的角度出发，对Matrix原理的探索难免会存在以偏概全的毛病，对其全貌也必然会存在描述不准确的情况。加之框架底层也在不断丰富和完善，很多技术内容存在不断变更和发展，本文描述的内容在短时间内也会存在描述不再准确的情况。但是好在基本原理应该是准确的，内容描述的过程中，如有错误，欢迎大家指正。

本文重点对底层支撑框架涉及的几个核心技术点进行展开：**前台包热部署设计原理、前中台隔离原理、前台业务身份设计原理**。

3、前台包热部署设计原理
==========

在中台化的实施路径上，伴随的节奏一般为：中台能力改造升级-->中台开放标准扩展能力-->前台使用标准扩展能力完成个性化开发-->前中台发布集成。其中发布集成有2种可选的方式，一种是中台和前台采用手工半自动化的方式进行集成，另外一种是中台和前台使用全自动的方式进行集成。全自动的方式，就免不了需要将前台的功能包，采用“热部署”的方式部署至中台相关的应用系统中。

本文对前台包发布、热部署的技术方案及思路进行探究，让我们的读者了解其设计理念。这样以后在工作中遇到类似前台包发布相关的问题，亦或者后续有机会使用并研究其他类似的技术框架，均可以做到举一反三，心中了然。

3.1、热部署基本原理

在开始介绍之前，假定我们作为底层框架的设计方，可以尝试回答下方的三个问题，并与后面给出的结果进行对比，看看与自己预判的答案是否一致。若不

一致，本文应该对你有些帮助，可以细细品鉴；若一致，那么恭喜你，说明你的技术功底相当扎实，对相关的原理及实践了解透彻。

问题：

1）使用前台扩展包，在发布平台操作完成（完成推送、生效）**后**，应用端进行扩容上线，扩展点包是否可以自动拉取加载、自动挂载运行?

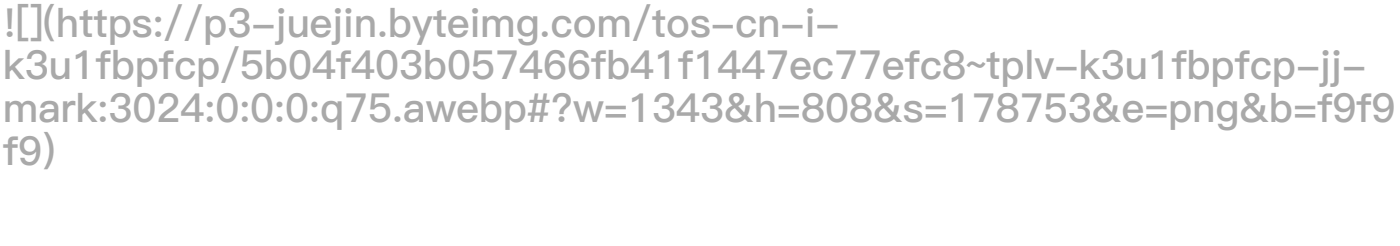
2）使用前台扩展包，在发布平台操作完成（完成推送、生效）**前**，应用端进行扩容上线，扩展点包是否可以自动拉取加载、自动挂载运行?

3）使用前台扩展包，在发布平台对部分容器完成前台包灰度推送，但没有触发生效，此时对这些容器执行重启操作，推送的前台包是否会挂载运行?

4）在测试环境亦或是其他环境，一不小心删除了热部署包路径及对应的文件，会有什么问题？如何解决?

在开始回答上面的问题之前，我们要先来给大家介绍下PaaS化下几个相关的名词，及这些名词背后的相互作用的关系。

可以先看下如下的架构设计图（笔者自己理解，与官方理解可能存在一定的差异）：



相关名词及对应的业务含义简要说明：

1）**控制台相关**：前台或者中台相关组织或角色在控制台操作上传前台包，控制台将相关的包信息写入至相关存储，并主动通知应用系统，应用系统内的Matrix SDK接收到通知后，执行相关的热更新及其他操作

2）**Matrix SDK相关**：热部署的包变更、发布等均依赖远端的控制台操作，在控制台操作审批通过后，远端的控制台将信息变更写入到相关存储，并发送变更通知到各个docker应用实例，各docker实例接受变更通知并执行如热部

署等不同的操作。docker实例执行热部署的工作主要有：从远程私服下载jar到本地目录，并执行一系列的操作流程，包括但不限于前台包卸载、前台包更新等，其中前台包更新包括但不限于：前台包类加载、前台包类初始化、前台包生命周期管理、前台包动态代理管理等。所谓万丈高楼平地起，Matrix平台框架的功能再复杂，实际的地基就是基于这一点，并在基础夯实叠加其他各类优化和加强的功能，大家可以仔细领悟。

3）**应用层执行相关**：中台应用层，在执行至扩展点相关的业务逻辑时，会动态代理至Matrix SDK，并由SDK接管逻辑，并确认业务是否命中相关的业务身份，在在业务身份命中后执行前台包的实际逻辑。

4）**监控及安全相关**：控制台及Matrix SDK相互合作，采集心跳、扩展点执行时间等各类统计维度数据，此处也是一个较为复杂的体系，不详细展开。

备注：以上架构图，只是笔者理解的Matrix技术框架的示意图，仅供参考。

前台包发布、热部署及管理相关的主要的链路示意图如下所示：

从图中可以看到，我们的前台包发布、热部署存在“推”、“拉”两条数据链路，两条数据链路也分别适配不同的业务场景，如下详细展开。

3.1.1、“推”链路

此链路的全过程可以表述为：软件实施人员在控制台，通过**推送**、**生效**等可视化界面工具操作完成后，控制台会将用户操作数据实时写入至相关的数据中心，数据中心依赖的核心组件DUCC（一种类似ZK的存储介质）会对外发布实时变更消息。集成了SDK的应用容器收到DUCC变更消息通知后，感知控制台的操作并依据操作类型不同做出差异化的反映：SDK依据操作的不同指令，在应用容器中执行不同的业务操作。

目前SDK中的操作策略指令包括但不限于如下4类：

1）**推送**：对应的功能可表述为：SDK接收到指令后，从远端文件服务器将相关版本的前台包下载至应用容器的固定目录中。此功能在Matrix技术框架中中对应的处理类为：PublishActionImpl。

2）**生效**：对应的功能可表述为：SDK接收到指令后，从远端文件服务器将相关版本的前台包下载至应用容器的固定目录中，并执行前台包的热加载功能（具体热加载的功能说明，可以参考第二篇文章，此处不在赘述）。此功能在Matrix技术框架中中对应的处理类为：TakeEffectActionImpl。

3）**卸载**：对应的功能可表述为：SDK接收到指令后，将相关的前台包从容器中卸载，并处理注销相关的数据（备注：此功能在藏经阁控制台默认情况下不会展示，我们平常情况下看不到这个功能菜单）。此功能在Matrix技术框架中中对应的处理类为：UnloadActionImpl。

4）**探活**：对应的功能可表述为：SDK接收到指令后，从当前的容器服务器发送相关的心跳包，用以采集监控当前容器服务器相关的实时状态数据。此功能在Matrix技术框架中中对应的处理类为：DoAliveActionImpl。

“推”链路适用的场景为：应用的场景为实时性要求较强的相关场景，包括但不限于新版本灰度及发布、版本回滚及控制、版本下线及监控等。此类场景要求在控制台执行相关的操作步骤后，控制台及相关的应用容器在短时间内（秒级或者毫秒级内）做出实时反映。主打的就是保障通知的及时性。

3.1.2、“拉”链路

此链路的全过程可以表述为：软件实施人员，在相关的容器部署平台，通过对容器进行扩容等操作，完成应用扩容。在对扩容的机器进行发布上线的时候，集成了SDK的应用容器，会自动从数据中心获取相关的应用元数据，在识别到元数据中存在部署版本的时候，自动从从远端文件服务器将**最新生效版本**的前台包下载至应用容器的固定目录中，并执行前台包的热加载功能。

此处需要注意一点，在京东现有环境内，容器扩容包含2步操作：1）部署平台创建新的docker环境，完成应用实例的创建，并处理复制更新好相关的镜像环境；2）软件实施人员对扩容出来的应用实例，执行发布上线等操作。其中“拉”链路出现的时机出现在步骤二中。从逻辑代码中，我们可以看到，在应用系统发布上线后，在spring的生命周期内，Matrix会执行相关“拉”逻辑。

“拉”链路适用的场景为：

1）对实时性要求不是很高的场景，如应用扩容等；

2）“推”链路无法触达或触达成本较高的场景。其中第二种场景有很多细分的业务场景，比如软件实施人员对容器中的部分目录文件执行了误删除等操作，或者容器中的部分目录文件出现了不可预知的损坏等异常场景。

3.1.3 、“推”和“拉”两条链路设计理念说明

从上面可以看到，“推”和“拉”两条链路都有其独自适应的应用场景，那么笔者提出一个问题，是否有可能只保留其中的一条链路呢？读者在不看后方的内容时可提取思考一下，为什么Matrix框架会执行如此设计呢?

在直接得出结论前，我们可以假定一下若只存在单链路，然后看看在单链路的情况下相关的复杂场景及其对应的解决方案，最后我们再来综合评定最优方案。

1、只存在“推”单链路：

需要应对的复杂场景：应用扩容。

复杂场景下需要解决的问题：在应用扩容时，在“推”链路下，数据中心如何能快速感知到扩容容器的存在，同时对此类场景实时下发处理消息降低实施成本。

可选的解决方案：在应用扩容发布时，应用端主动向数据中心发送相关的消息，获取数据中心最新的应用数据等；数据中心接收到消息后，再实时下发相关的处理消息。应用容器接收到消息后，再执行下载前台包、热更新等前台包部署操作。

弊端：应用扩容发布后，按一般情况，应用即可对外提供服务。为了防止出现应用已对外提供了服务，但是相关的前台包还没有拉取到应用本地，导致相关的扩展点出现“空转”等情况。为了解决此问题，上方提及的可选的解决方案，务必要保障实时性特别高，即务必保障在系统启动并对外提供服务前，快速完成解决方案。那么怎么保障？最好的办法就是在未处理成功前，设定系统没有启动成功，不能对外提供服务。如此，此方案落地层面，涉及SDK和数据中心一来一回两次交互，存在方案较为复杂，存在后续维护运维成本高的问题；同时还存在消息链路太长，导致整个项目的启动时长变得不可控，扩容体验变得极差的问题。

2、只存在“拉”单链路：

需要应对的复杂场景：软件实施人员在控制台实时操作“发布”、“推包”等操作。

复杂场景下需要解决的问题：软件实施人员执行的操作，在“拉”链路下，在应用端要可以及时感知到。

可选的解决方案：可选的方案有2个：1）控制台和应用端建立直接联系，在控制台完成相关操作后，直接通过联系渠道将操作传递给应用端；2）应用端定时获取数据中心的最新应用状态数据。但是方案一基本变相等同于是“推”链路，此处可以忽略不计。方案二理论上可行，但是也存在2个弊端：1）定时的时长不好控制，太长了，则操作可被感知具有一定的延迟性，存在客户体验不佳的情况；2）定时时长太短了，多个应用无脑同时请求数据中心，对数据中心的高可用提出了极高的要求。对此弊端，我们可选的方案有：制定业务可接收的延时时间，并定时去请求数据中心。并对请求时，请求数据和返回数据存在数据过大的情况，采取数据极限裁剪、数据压缩等方案。并设定独立专业团队，通过建立双数据中心等机制，保障数据中心的安全性及高可用性。

弊端：从根上来讲，相关的通知链路总会存在或多或少的时延，客户体验或多或少都有一定的影响；同时方案复杂性变高，维护成本变大。

小结：从上述两处推测方案可以看到，单纯只是采用一种解决方案，均存在较多的方案瑕疵。而将“推”和“拉”两个方案进行结合，并将方案应用在其适应的领域，则上述提及的弊端均可以以最小的成本来得以化解。而“推”、“拉”结合的解决方案，在业界的各类场景下，也存在广泛运用。大家可以多加领悟。

3.2、热部署相意事项

1、**需要注意控制大面积扩容的节奏**：从上述的表述中，我们可以看到，应用在扩容发布的时候，集成了SDK的应用服务器，会从远端服务器拉取最新的前台包至应用服务器。假定我们的前台包裁剪的大小比例不合理，一个前台包的大小在300M及以上，加之我们扩容采取的是大面积扩容发布上线应对线上紧急情况的场景，那么大促的某个瞬间，会存在同一时刻，应用容器集中请求文件服务器，可能导致文件服务器过载，进而导致下载失败和发布上线启动不成功的情况。这种情况，我是顶不住，不知道你顶不顶得住？

对于这种情况，推荐3类解决方案：1）每次只进行小批量扩容，保障扩容数量的合理性；2）仍然采取大批量扩容，但在应用发布启动时采用小批量的方式分批进行；3）在大面积扩容时，通知Matrix相关同事，要求其对文件服务器及扩容操作进行重点保障。至于具体落地方案怎么选择，大家可多加思考，并自由抉择。

2、**需要注意控制前台包的大小**：不管是“推”还是“拉”哪条链路，均会涉及到对前台包从远端服务器下载至本地的操作。从感官上来，此项操作对网络资源的消耗、应用服务器本地资源的读写均会在瞬间造成较大的影响。如此，就要求我们的共建方前台角色、中台角色严格把控前台包的大小，避免无关的包打入，控制前台包的大小在相对合理的区间。

3.3、问题解答及分析

依据上方的讲解，我们再来看下序言中提及的三个问题，来看看最终的答案，不知和你心中的答案是否一致呢?

问题1） 使用前台扩展包，在发布平台操作完成（完成推送、生效）**后**，应用端进行扩容上线，扩展点包是否可以自动拉取加载、自动挂载运行?

答案： 可以。此时应用端扩容上线，因为应用服务器上不存在任何前台包，会采取“拉”链路，将相关的前台包一次性拉取到本地进行加载、自动挂载运行。

问题2） 使用前台扩展包，在发布平台操作完成（完成推送、生效）**前**，应用端进行扩容上线，扩展点包是否可以自动拉取加载、自动挂载运行?

答案： 可以。此时仍然是“拉”链路在生效，但是需要注意一点，次数“拉”链路拉取的数据，为发布平台管理的上一次（如果有的话）上线的最新信息。在完成扩容上线后，如果我们没有在发布平台对新扩容的机器进行推包、发布等操作，线上应用会存在并运行两个版本的前台包。

问题3） 使用前台扩展包，在发布平台对部分容器完成前台包灰度推送，但没有触发生效，此时对这些容器执行重启操作，推送的前台包是否会挂载运行?

答案：不会。虽然说前台包已完成灰度推送，相关的前台包文件已在应用容器中存在，但是容器执行重启操作时，SDK会自动按分组、IP等检测最新已

生效的版本，若发现当前版本并没有生效，哪怕这个前台包文件在容器中已存在，也不会挂载运行。

问题4） 在测试环境亦或是其他环境，一不小心删除了热部署包路径及对应的文件，会有什么问题？如何解决？

答案： 分情况。

情况1：若是我们将前台包相关的目录进行了整个移除，则在应用容器再次进行启动的时候，会执行“拉”链路，将相关的前台包一次性拉取到本地进行加载、自动挂载运行。

情况2：若是我们讲前台包相关的目录中的部分文件进行移除（具体移除了哪类文件此处不详细展开），“拉”链路识别前台包文件已存在，“拉”链路不会得以执行。但是因为文件已损毁，相关的数据不完整，SDK在记载的过程中，会抛出相关的错误异常，并给出相关的错误信息，最终呈现的效果为整个应用启动失败。

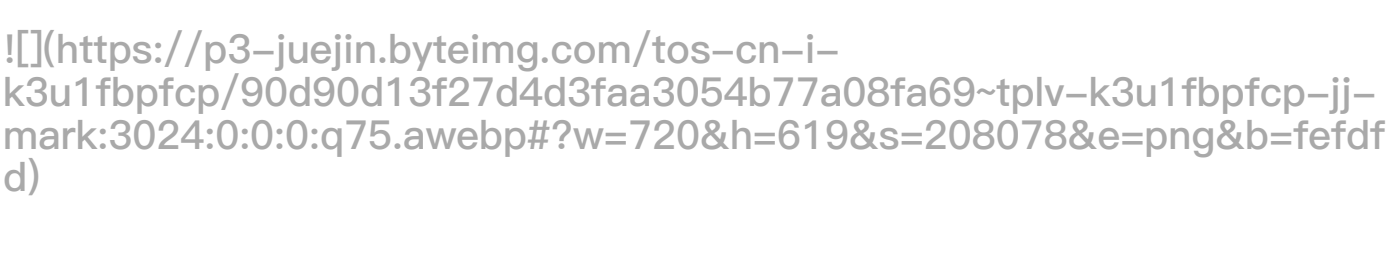
4、前中台隔离原理
==========

前文介绍的热部署操作完毕后，前台扩展包，就可以正确在中台相关的容器中完成部署了加载了。那么此处问题来了，前台和中台会共用某些相关的类，如何保障执行的安全性及可靠性呢。那么就不得不提到了类隔离机制，和隔离机制背后的原理：双亲委派模型。

类隔离机制：前台和中台使用不同的ClassLoader来加载实现隔离，中台使用默认的ClassLoader，前台使用自定义的ClassLoader，其中部分类为了避免冲突，前中台共享了默认的ClassLoader，这些类包括但不限于RPC框架、缓存框架等基础组件包。

笔者早期工作的时候，在web应用系统，使用Eclipse的osgi概念在开发处理相关的bundle包的时候，对于涉及web应用中登录的session等复杂的数据结构进行信息传递处理的时候，经常出现2类及以上的ClassLoader协同出现问题，导致应用系统出现一些莫名的、难以排查的ClassNotFoundException。表象为应用的工程包明明存在，在执行的时候就是会出现错误的奇特场景。当时初毕业，技术功底浅薄，对此类问题印象极其深刻。而在我们在京东的实际使用的过程中，暂无发现此类问题。如果大家有遇到此类问题，欢迎进一步交流。

说到类加载，就不得不提Java体系中大名鼎鼎的“双亲委派模型”（英文描述为：parents delegation model）。有关这个描述，此处多说一点。中文含义中的“双”，其实在英文描述中并没有对应的描述，也即“双”不“双”，其实不紧要，紧要的是“委派”二字。“双亲委派模型”的核心内容，用稍粗鄙点的言语可以表述为：有事找爸爸（爸爸如果也有事，可以找他爸爸的爸爸，以此类推）。此设计理念的好处内外网相关文章介绍得较多，此处不在赘述。



此模型的简要代码可以表述如下：

```
    protected Class<?> loadClass(String name,boolean resolve)
        throwsClassNotFoundException
    {
        synchronized(getClassLoadingLock(name)){
            // First, check if the class has already been loaded
            Class<?> c =findLoadedClass(name);
            if(c ==null){
                long t0 =System.nanoTime();
                try{
                    if(parent !=null){
                        c = parent.loadClass(name,false);
                    }else{
                        c =findBootstrapClassOrNull(name);
                    }
                }catch(ClassNotFoundException e){
                    // ClassNotFoundException thrown if class not found
                    // from the non-null parent class loader
                }

                if(c ==null){
                    // If still not found, then invoke findClass in order
                    // to find the class.
                    long t1 =System.nanoTime();
                    c =findClass(name);

                    // this is the defining class loader; record the stats
sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);
sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
                    sun.misc.PerfCounter.getFindClasses().increment();
```

```
                }
            }
            if(resolve){
                resolveClass(c);
            }
            return c;
        }
    }
```

所谓“双亲委派模型”，从上述代码可以看到，先检查类是否被加载过（第6行），若没有加载则调用父加载器进行加载（第11行），若父加载器为空则默认使用启动类加载器进行加载（低13行）。可以看到，若都失败，则调用当前类加载器进行加载（第24行）。是不是和上述找“爸爸”的描述，比较契合?

在Matrix框架下，中台相关的逻辑和前台相关的业务包，是由2个独立的ClassLoader类分开管理，其中前台相关的业务包，是由名为com.jd.matrix.core.classloader.BizClassLoader的类进行管理。我们先来看下这个类的具体逻辑：

```
public class BizClassLoader extends URLClassLoader{
    public BizClassLoader(URL[] urls){
        super(urls);
    }

    protected Class<?>loadClass(String name,boolean resolve)
throwsClassNotFoundException{
        Class<?> clazz =null;
        clazz =this.findLoadedClass(name);
        if(clazz !=null){
            return clazz;
        }else{
            try{
                ClassLoader jdkClassLoader
=ClassLoaderFactory.getJDKClassLoader();
                clazz = jdkClassLoader.loadClass(name);
                if(clazz !=null){
                    return clazz;
                }
            }catch(ClassNotFoundException exception){
            }

            if(ExportClassManager.match(name)){
```

```
                clazz
=ClassLoaderFactory.getInternalClassLoader().loadClass(name);
                if(clazz !=null){
                    return clazz;
                }
            }

            try{
                clazz =this.findClass(name);
            }catch(ClassNotFoundException exception){
            }

            if(clazz ==null){
                clazz
=ClassLoaderFactory.getInternalClassLoader().loadClass(name);
            }

            return clazz;
        }
    }

    protected Class<?> findClass(String name) throws
ClassNotFoundException{
        returnsuper.findClass(name);
    }}

```

从代码逻辑可以看到，Matrix提供的类加载器，其实是**破坏了**“双亲委派模型”，破坏点从13行开始，若当前类没有加载，没有调用父加载器进行加载，而是使用了jdkClassLoader去加载；同时若开启了Matrix包屏蔽的策略（21行），则会使用中台的ClassLoader去加载。

此处破坏的妙处在于：前台包可以找到并使用中台包对应的相关类；但是中台包不可以找到并使用前台包对应的相关类。粗俗点可以描述为：大哥不知道小弟，但是小弟在大哥的开放区域，可以有限了解大哥。

从此处也可以看到，Matrix对于类加载的管理与控制，与OSGI等标准规范相比，安全性等管控方面要松弛得多。从严格意义上来，此类管理的方式，安全性其实并不足，若中台开放的区域过大，加上前台包被有心人动手脚，其实是存在安全隐患，并防不胜防。

我们可以看到，Matrix会结合容器侧的spring配置里的exportClassConfig属性，使用自定义的类加载器(BizClassLoader)去加载垂直业务包里的类，主要有两个核心内容：

1、对于不同的前台业务包里的类，分别使用BizClassLoader的不同实例去加载，对于垂直业务包里的自己开发的业务类，即使不同的垂直业务包中存在全限定名相同的类，但因为加载他们的BizClassLoader实例不同，不会出现冲突问题，类隔离的要求能够得到满足。

2、对于exportClassConfig这个属性配置的相关类（包含但不限于：中台提供的sdk包所包含的类、基础RPC框架类），不会由1里所提及的自定义的classLoader的实例去加载，只会由同一个类加载器（加载中台容器业务类的appClassLoader）去加载，以此来保证中台容器调用扩展点实现时参数类型校验的一致性。

从上方的信息和代码中，我们可以看到， 假定前台和中台存在相同包名、相同类名的类，在执行前台扩展点的时候，分为前后2个步骤：1）执行前台包扩展点的业务身份匹配逻辑；2）执行前台包扩展点的实际业务逻辑。2者的区别，主要在于后者有一个主动切换当前类加载器的逻辑。也即在步骤1时，当前的类加载器为应用容器业务类的加载器；步骤2时，当前的类加载器切换为独立的类加载器。

5、 **前台业务身份设计原理**
==========

中台和前台合作的巧妙之处，在于中台开放了多种标准的、可扩展的能力，前台基于自己的不同的业务身份实现多种个性化业务场景。即一种中台的能力，可对应多个前台扩展部署包。那么运行时，中台如何知道应该使用哪些前台包呢?

这里就不得不提业务身份这个概念了，在中台的建设思想中，每个前台包都有自己独立的业务身份。中台系统运行的时候，依次匹配前台部署包，并选择业务身份命中的前台包执行相关逻辑。

业务身份的识别方式： Matrix业务身份有2种识别方式

1、**前台包手工编写识别业务身份**。即在前台自己在元数据（Annotation）定义自己独立的parserClass ，识别方式，由前台业务系统手工编写识别。这种方式的好处是纯手写代码，加上前中台组合review的时候，一般不会出现功能问题及性能瓶颈。但是也会存在很大一个弊端，即对某个业务而言，是不是属于自己的业务身份，是前台自己说了算，在一般情况下是没有问题的，但是如果前中台reivew的机制失效了，前台基于某些特定的原因，将前台的业务身份识别方式调整扩大或缩小，会导致影响的业务范围也会对

应的增大或者缩小，影响范围非常不可控。

从中台建设初期来看，本来预计这种方式，以后一定会废弃掉。从过去几年的实践经验来看，这个地方预判错了，这种形式目前仍然是主流的形式，只是不同的系统对同一个业务身份的识别，逻辑千差万别，在串联业务流程的过程，确实会存在较为痛苦的情况。

2、**中台统一管控识别业务身份**。在在前台自己在元数据（Annotation）设置autoParser的值为true，此时不用前台来判断处理和业务身份相关的控制逻辑了，这个逻辑会内置在中台，中台相关的业务身份的解析类为AutoBizCodeParser，此类的定义与实现框架与前台包定义的方式无异，只是实现了平台通用的能力：即由中台来决定是否可以命中某一个业务身份，在管理平台增加相关配置来匹配是否可命中业务身份。

核心的逻辑是Matrix内置定义了一个简单的脚本模板，在系统采用热部署指令，加载启动前台包的时候，识别到前台包包含App元数据（Annotation）是自动识别业务身份的时候，会在系统内按内置的脚本模板采用javassist框架来初始化缓存一套字节码。在实际去匹配前台业务身份的时候，实际执行的是此类字节码来动态判断业务身份是否可以命中。字节码的模板代码如下所示：

需要注意一点，目前底层框架生成字节码的时间点，并不是系统自动加载或者热部署生效的时候，而是尝试命中业务逻辑的时候。预判肯定会存在执行过程中第一次速度缓慢的情况，实际使用的时候可以多加注意。

```
#{ClassStart}
import java.util.*;
#{Package}
public class #{ClassName} {
    #{MethodInfo}
}
#{ClassEnd}

#{PackageStart}
import #{TypeName};

#{PackageEnd}
```

```
#{VarStart}
    #{Type} #{Var}=(#{Type})context.get("#{Var}");
#{VarEnd}

#{executeStart}
 public boolean execute(Map context){
       #{VarInfo}
       return #{Express};
    }
 #{executeEnd}


#{allStart}
    private boolean all(#{Var} ){

        Iterator iterator =#{Collection}.iterator();
         while (iterator.hasNext()) {
           #{ItemType} #{ItemVar}=(#{ItemType})iterator.next();
            if(!(#{Right})){
               return false;
             }

        }
        return true;
    }
#{allEnd}

#{anyStart}
    private boolean any(#{Var} ){

        Iterator iterator = #{Collection}.iterator();
        while (iterator.hasNext()) {
          #{ItemType} #{ItemVar}=(#{ItemType})iterator.next();
           if((#{Right})){
              return true;
           }

        }
        return false;
    }
#{anyEnd}

```

业务身份命中基本流程：此处其实无需多言，业务身份命中的逻辑其实隐含在前面介绍的前台包业务逻辑实际执行的地方，只是Matrix框架在执行逻辑前，创造了一个类似会话（Session）的概念，在会话中去对业务身份是否命中

执行相关匹配逻辑，实际就是对相关的前台业务包执行fiter方法，判断评估业务身份命中。在会话（Session）中对前台包依次执filter去匹配业务身份。

存在的风险：

1）部分前台包业务身份识别逻辑较重，会导致整体中台逻辑运行缓慢，存在性能风险；对应的建议解决方案：业务身份识别逻辑一定要轻量级；

2）目前Matrix框架默认仍然只是捕获了Exception类别的异常，对于**Throwable**级别的异常仍然是没有捕获的。部分前台包业务逻辑存在偶发性问题，执行业务身份匹配filter逻辑时，若抛出**Throwable**级别的异常，会导致其余业务身份的前台包也无法执行。这真是一粒老鼠屎，搞坏一锅粥。

业务身份基本原理：Matrix实际以业务身份来管理业务包与扩展点，同一个业务身份，可以命中一个垂直扩展点（代号为Y）和一些水平扩展点（代号为X）。针对同一个待识别的业务规则，比如同一个sku或者同一个订单而言，不可以命中多个业务身份，若实际命中了多个业务身份，Matrix会限制只会命中其中的某一个业务身份对应的扩展点方法。也即对这种业务场景，Matrix即使会命中多个业务身份的fiter方法，但是实际只会执行其中唯一的一个业务身份对应的扩展点方法，具体命中的业务身份，初看有一定的随机性，具体顺序参加下方的描述。

业务身份的顺序：从Matrix遍历业务身份处理扩展点的业务逻辑来看，业务身份的顺序至关重要，排名靠前的业务身份，其对应的扩展点方法执行的几率会大于排期靠后的业务身份。经排查，Matrix对业务身份排名的逻辑由业务身份（App）的三个属性来共同确定，具体为：priority（由大到小）、version（由大到小）、code（由大到小）。在满足上述排序规则的前提下确定业务身份的优先级。假定业务上存在如下场景：对同一个订单或者同一个sku的场景，同时可命中二级业务身份和三级业务身份，理论上业务期望三级业务身份被命中到。但是很不巧，priority、versionSpec等值设置不合理，导致二级业务身份被命中到出现与预期不一致的情况。对于此类场景，我们就需要制定严格的代码规范，对前台业务侧限定其priority、version、code的数值，确保业务逻辑的正确性。

重点提示：业务身份的定义与使用，是业务系统出现问题最多的地方。业务身份在Matrix中是一个很重要的概念，其设计理念期待整个业务身份是一颗节点互不交叉的树，但是在实际业务执行的时候，经常会存在业务身份重叠的情况。设计定义前台包的人员，属于两个不同的部门，属于两拨不同的人，2者很难意识到业务存在交叉重叠的情况，已经发生多次线上业务跑飞了出现与预期不一致的情况，后来花了大力气来解决的场景。常见出现问题的案例举例供大家参考：我们提供了一个扩展点供前台业务使用，其中一个前台包的业务身

份是大家电，另外一个前台包的业务身份是五星。结果在实际业务上，同一个sku或者同一个订单，既是大家电的业务，也是五星的业务。对于这种场景，如何解决呢？这个问题留给大家来思考。

Matrix对业务身份排名的具体代码如下所示，代码逻辑参见BizCodeSpec.compareTo方法：

```
    public int compareTo(BizCodeSpec o) {
        if (o == null) {
            return -1;
        } else if (this.priority != null && o.priority != null) {
            int ret = o.priority - this.priority;
            if (ret != 0) {
                return ret;
            } else if (this.versionSpec != null && o.versionSpec != null) {
                ret = this.versionSpec.compareTo(o.versionSpec);
                if (ret == 0) {
                    ret = o.bizCode.compareTo(this.bizCode);
                }

                return ret;
            } else {
                return 0;
            }
        } else {
            return 0;
        }
    }
```

最佳实践：

1、一般而言，我们在设计并编写前台包的时候，包名应该具有前台独立的业务属性，也即包名和中台的包名应该有一定的区隔。对于可能会和中台包冲突的业务场景，我们可以在框架中进行类名或者包名的排除。

2、中台开放给前台的包名范围，应该尽量限定在最小范围，防止前台不经意的恶意行为。

除这2点外，可能还存在其他各类场景对应的最佳实践，有待我们进一步探索。

6、思考
========

不管是中台化也好，还是去中台也好。核心思路，都在于如何以技术，服务好我们的业务。中台化，亦或者是品类差异化思路，均为交付过程中选择使用的差异化工具。但是工具背后，必然隐藏着对应的落地技术关键点，而这些关键点，道理都是相通的。作为技术人员，持续思考，如何从漫天繁花中，找到底层最核心的症结，不断打磨丰富我们的产品，提供优质的服务，值得我们持续探索。

> 作者：供应链研发 徐开廷
>
>