快手工程架构治理大揭秘：告别崩溃，提效神器来袭！

> 在快手没有什么是不能release的，component、controller、runner都可以作为jar被release出来，我们写的每一个类都会通过复杂的依赖链路以光速形式扩散，扩散与恶化速度堪比奥密克戎，遗憾的是奥密克戎只影响了我们3年的生活，但快手的工程架构问题却困扰了我们远不止3年……

项目背景

01 项目背景

快手过去的工程架构，在多业务线都遵循了Component、Util、SDK、Runner、API的5层结构模式，导致了多业务线发展中工程包巨大、发布的SDK巨大，各业务间的代码依赖复杂。在多条业务线都暴露了工程劣化严重、影响效率和稳定性的问题。

在2023年的工程架构治理过程中，大部分Java后端同学在2023年都或多或少或主动或被动的在参与其中，由于快手存量的工程、服务、jar包数量都非常庞大，本文重点写下在治理过程中的一些提效方式与遇到的问题。

经验沉淀

01 IDEA卡顿问题

jar包/产物包治理是一个很宏观的工作，仅笔者所在团队就涉及到150个不达标的git仓库，我们在治理过程中总是会被IDEA卡顿的问题折磨，像笔者这种还停留在intel芯片的mac，IDEA在构建索引时回复消息都会卡顿，有时要分析一下调用链路，有时要临时修改几行代码，除了VIM这种比较geek的方式，我们尝试了两种解决办法。

1.1 源码在线浏览

我们首先尝试了OpenGrok来构建源码的索引关系，早期快手的codesearch工具也是基于这个来的，以某个近百万行代码工程为例，构建一次索引用时大概

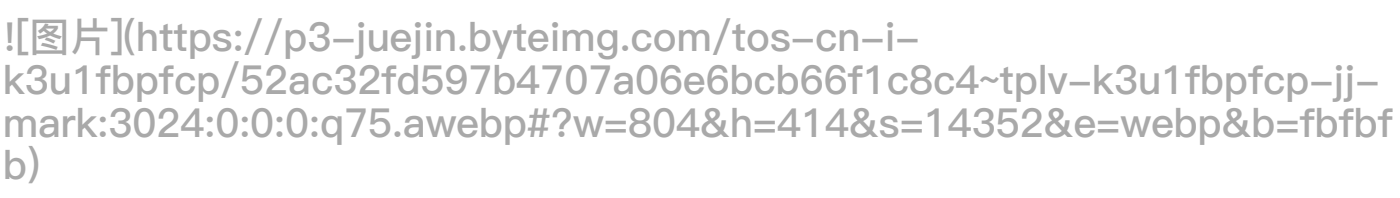
1分钟，缺点是只能索引仓库内的源码引用关系，下探到所依赖的jar包是不支持的，属于加强版的gitlab，不推荐。

1.2 云端IDEA

既然本地电脑性能是瓶颈，那就放到云端试试吧。

1.21 云主机申请与配置

我们申请了12核-24G的容器，镜像选java集成变成环境的，jdk git maven都集成好了。



1.22 云主机申请与配置

云主机登录配置：

首先将本地机器的公钥添加到云主机平台上，

如果没有创建过公钥，需要先生成公钥，一路回车

```
ssh-keygen -t rsa -C ”yourname@kuaishou.com”
```

如果已有公钥，直接按下一步获取

公钥获取， 公钥包含的是这个文件里的所有内容

```
cat ~/.ssh/id_rsa.pub
```

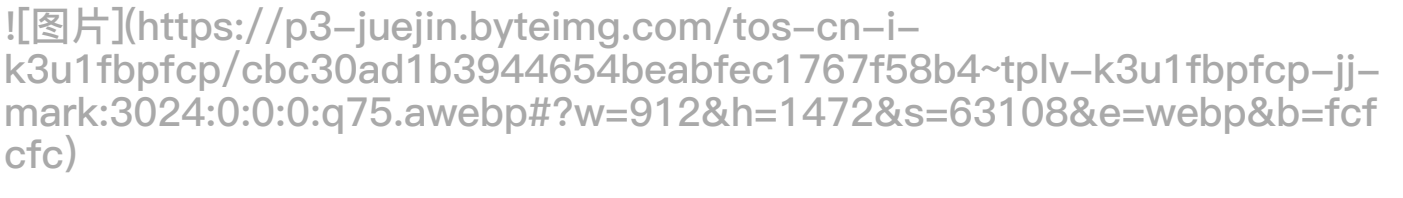
```

完成上述配置之后，可以通过ssh方式从本地机器直接登录云主机

```
设置用户信息
git config --global user.name "your name"
git config --global user.email "yourname@kuaishou.com"

生成公钥，遇见输入提示一律回车
ssh-keygen -t rsa -C "yourname@kuaishou.com"

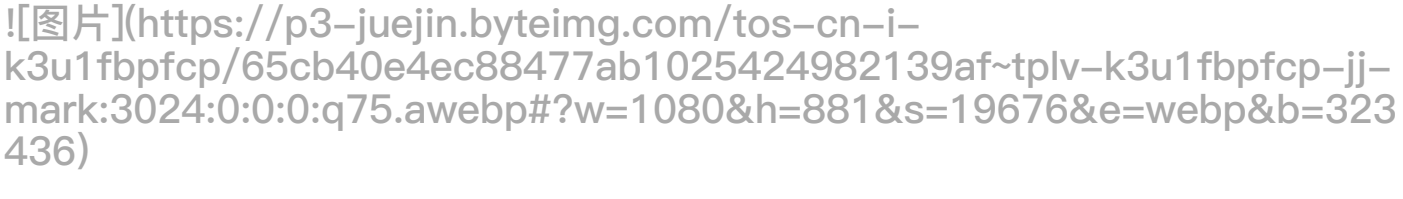
#获取ssh key
cat ~/.ssh/id_rsa.pub
```

然后将上一步复制的ssh key粘贴到公司的gitlab中

云主机maven配置：

```
#将通用的settring.xml文件拷贝过来
mkdir ~/.m2
cp /usr/local/kuaishou-build-
tools/src/main/resources/settings.xml ~/.m2
#maven默认集成的版本是3.6.3，有需要的话可以自己更新
```

云主机编码修改，默认编码有中文乱码问题：

```
#1> 在 ~/.bashrc末尾增加如下命令
export LANG=en_US.UTF-8
#2> 执行
source ~/.bashrc
```
```

本地mac配置：

1. 安装toolbox：[www.jetbrains.com/zh-cn/toolb...](http://cxyroad.com/”https://www.jetbrains.com/zh-cn/toolbox-app/”)
2. 安装Jetbrains GateWay，直接通过toolbox安装



3. 连接配置：SSH -> New Connection，然后点击Check .. Continue



4. 选择IDEA版本，可以默认，也可以通过Other Options选择下载方式。默认选项，云主机会自动下载。

也可以从本地上传，下载地址：[www.jetbrains.com/idea/downlo...](http://cxyroad.com/”https://www.jetbrains.com/idea/download/download-thanks.html?platform=linux”)

5. 选择你要打开的项目，如果云主机上还没有下载，可以直接点open an SSH termial直接登录云主机把要打开的项目clone下来
6. 点击Download and Start IDE，会有一段时间的等待，完成后就可以看到idea打开了

云端IDEA会受到网络延迟等问题的影响，丝滑度暂时还不能跟本地开发媲美，但是对于**应付在工程架构治理过程中需要临时打开一些工程、调整少量的依赖或代码的场景绰绰有余**，同时我们申请的云主机也可以帮助我们跑一些下文会用到的插件命令。

02 治理过程

治理思路主要如下:![图片](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/a116d583d7874795b4ae07070d0d4a78~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1080&h=639&s=31980&e=webp&a=1&b=ccf9f9)

这块我们主要推荐2个插件

2.1 MavenHelper

MavenHelper是一个IDEA插件，想必很多同学都使用过，在Marketplace搜索安装即可，主要用来查看maven的依赖树、依赖的传递链路。最重要的是，**在依赖治理的过程中MavenHelper可以帮助我们很快的定位哪些需要显示声明版本号的jar包是其他团队内部维护版本的**，这个会在下文的问题分析中详细说明。

2.2 自研依赖分析插件

> 二方包：内部私有包

> 独立依赖大小：jar包自身的体积+Maven仲裁后完整依赖树节点的总大小，大小均指磁盘占用空间

早期依赖治理过程中有一个问题很困扰我们，现有的插件本地运行只能分析类依赖关系，即依赖了哪个jar包的哪个类，但是对于优化包体积的目标，我们更希望能找出传递依赖大小topN的jar包，在这里我们分为两个部分。

1.采集所有二方包的独立依赖大小，这部分主要是公司工具链的同事实现的，原理就是所有二方包在发布的时候都会执行一下Maven的copy-dependencies采集二方包每个版本的大小。

2.自定义Maven插件，在Maven解析完依赖树之后，我们收集最终的jar列表，然后根据对应的groupId、artifactId、version去远程查询第一步中采集的数据，在本地进行排序后进行输出。

```
mvn clean -Denforcer.skip=true -DskipTests -Dmaven.test.skip=true -Dcheckstyle.skip com.kuaishou:operation-dependency-maven-plugin:RELEASE:ops-deps
```

执行结果如图所示

这样我们就可以在几十秒的时间内分析出本工程/jar包的头部依赖

03 收益预测工具

随着治理工作的推进，低垂的果实差不多都已经摘完了，治理进入深水区，在治理过程中我们遇到越来越多的工程，这些工程依赖的jar包传递依赖大小全部小于架构治理的最低标准，但是产物包/部署包体积仍然超标，对于这类工程有两种解法：

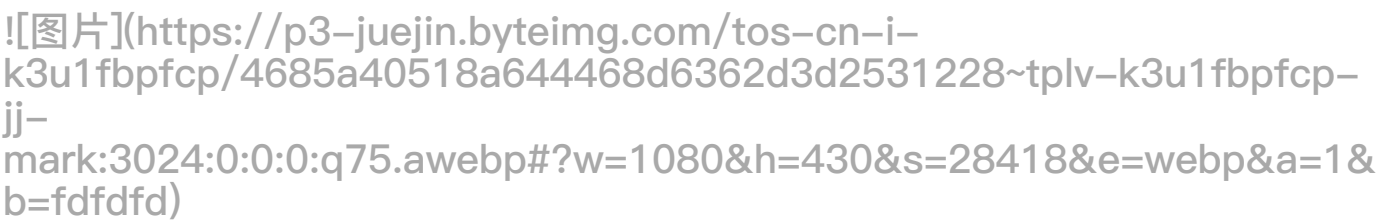
1.如果这个工程依赖的二方包过多，需要分析是不是个all in one的巨型单体工程，最直观的是统计部署的服务数，这类问题可以通过拆分工程来解决；

2.如果本身就是一个聚合服务，或者说工程不适合做拆分，我们就需要case by case的来分析下到底哪些包移除掉之后能够达标了；

第二种情况，对于业务侧来说，优化产物包体积的主要途径是优化依赖。业务侧每干掉一个依赖，都可能需经历代码改造、测试、上线等阶段，成本不可谓不高。所以，优化前，我们可能需要知道干掉哪个依赖包能最快的减小产物包体积。目前的依据可能主要有两个：

1. 依赖jar包本身体积大。
2. 依赖jar包传递依赖体积大。

以传递依赖体积大作为依据存在一个问题，如果某些传递依赖是公共的，那么优化掉某个jar包其实并不能优化掉这部分传递依赖，甚至可能新增依赖。一个典型的依赖图如下：



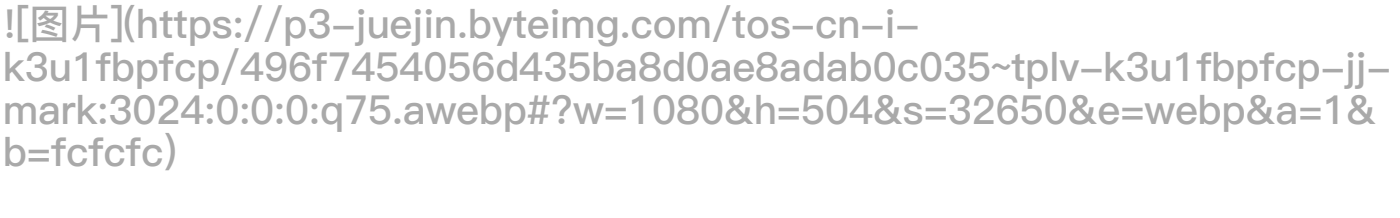
A、B是直接依赖，其它为间接依赖，引用顺序为从左到右。

假设干掉了依赖A，那么同时被干掉的是C，但是D和F不会被干掉，依赖E会从版本v1.0变成v2.0，同时会新增依赖G。有疑问的话可以看看依赖仲裁场景。优化完可能会有点绝望，期望150，实际1.5。

3.1 工具实现

3.1.1 目标分析

计算一个依赖被干掉后的产物包依赖，可以通过全局排除掉该依赖后按照maven的依赖分析规则重新计算产物包依赖，产物包原始依赖体积-重新计算后的体积即为收益。如下图所示：



产物包原先依赖为A C D E(1.0版本) F B，按照maven的依赖收集规则，干掉依赖A之后，依赖为B D F E(2.0版本) G。干掉依赖A之后的依赖总体积 - 干掉依赖A之前的依赖总体积，即为干掉A的收益。对于B、C、D、E、F包的分析逻辑也是如此。

3.1.2 代码实现

maven执行解析依赖的一般过程如下：

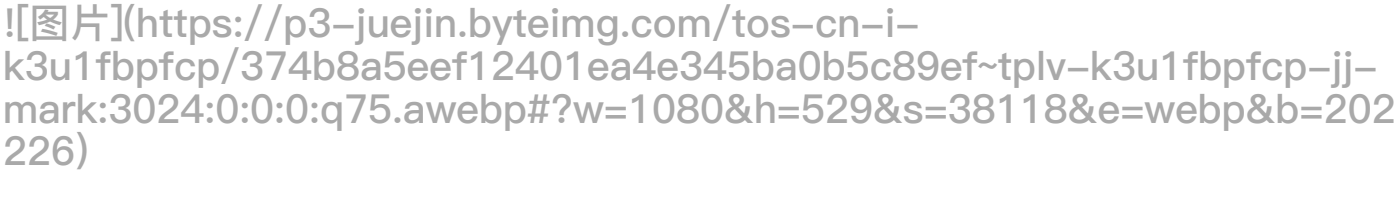
LDR -> LifecycleDependencyResolver

DPDR -> DefaultProjectDependencyResolver

DDC -> DefaultDependencyCollector

maven依赖解析的关键方法为：源码地址
`org.apache.maven.project.DefaultProjectDependenciesResolver#resolve`

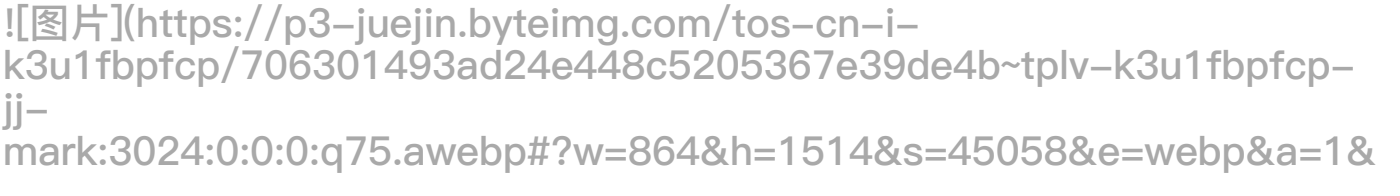
其中依赖收集的关键方法为：源码地址
`org.eclipse.aether.internal.impl.collect.DefaultDependencyCollector#collectDependencies`



`DefaultDependencyCollector#collectDependencies`包含了解析传递依赖以及依赖仲裁的逻辑，往该方法的`org.eclipse.aether.RepositorySystemSession`参数对象中设置`DependencySelector`可以全局排除指定依赖。

所以，全局排除某依赖后重新计算产物包依赖的逻辑如下图：

计算产物包所有依赖被排除后的收益过程如下：



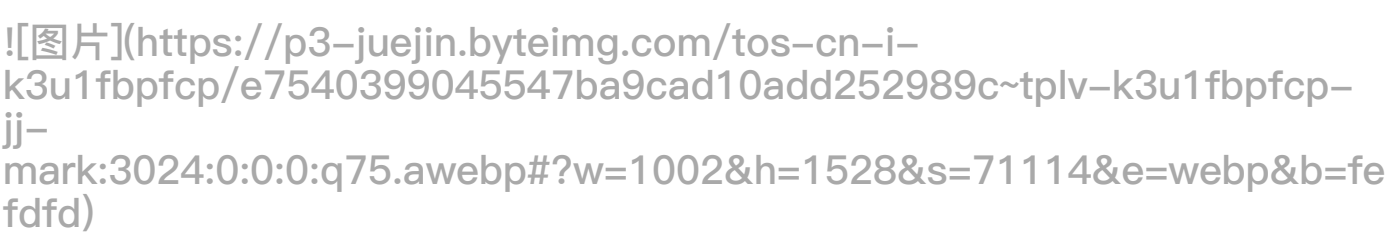
b=fdfdfd)

04

NoClassDefFoundError问题的解决

4.1 问题背景

在工程架构治理期间NoClassDefFoundError似乎成了一个热词。



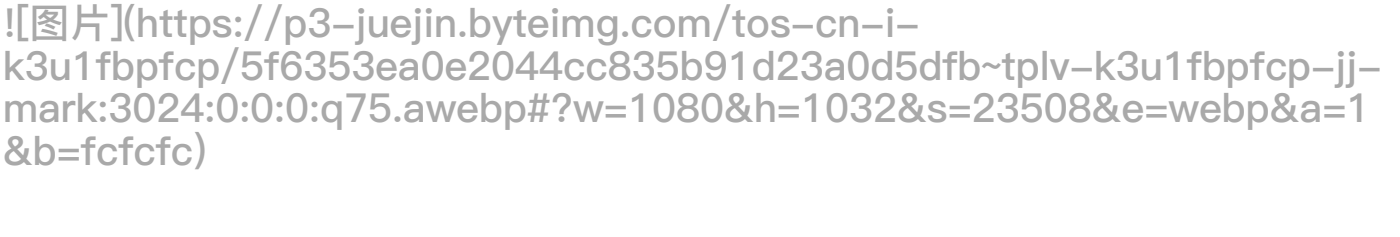
以下为示例

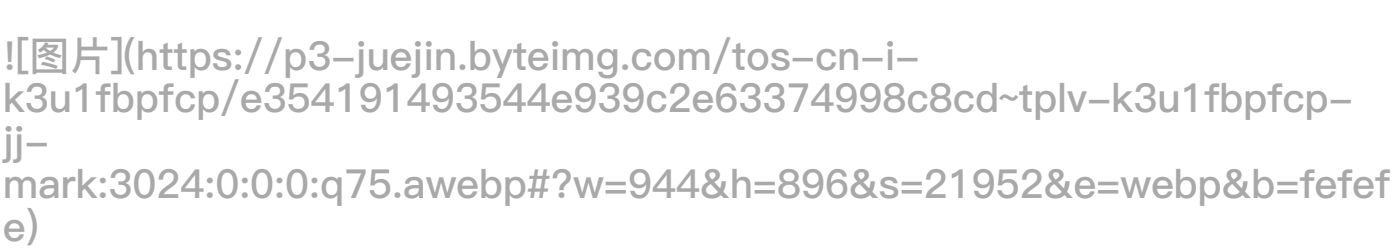
1. 假设有一个工程依赖了api-core-sdk里的UserService，UserServiceImpl中又引用了xxx-scope-sdk，但是在api-core-sdk的pom中并没有声明xxx-scope-sdk，而是通过photo-sdk间接引入的

2. 某天photo-sdk做瘦身，移除掉了xxx-scope-sdk，并进行发布

3. 由于kuaishou-xxx-api并没有直接引用xxx-scope-sdk中的类，且api-core-sdk已经是编译过的class文件，在kuaishou-xxx-api打包过程中不会发生任何编译问题

4. api服务上线，当业务逻辑走到api-->api-core-sdk--->UserCacheService这个链路的时候会发生NoClassDefFoundError





在这个过程中，

* photo-sdk(名字为示例)的作者正常进行瘦身工作，并且自身编译能通过，自身服务正常运行
* api-core-sdk(名字为示例)的作者没有做任何改动，甚至连打包的行为都没有

并且

* 这种问题编译期不会发现
* 如果其他工程通过其他路径间接引入了xxx-scope-sdk也不会有异常发生
* 这种问题往往被坑的是包体积较小的工程(引入的依赖少，缺少的jar大概率不会通过间接依赖传递进来)

临时解决办法

* kuaishou-xxx-api引入xxx-scope-sdk

问题的根原因于api-core-sdk是通过间接依赖引入的xxx-scope-sdk，这个问题需要api-core-sdk的维护方去调整依赖，才能彻底修复

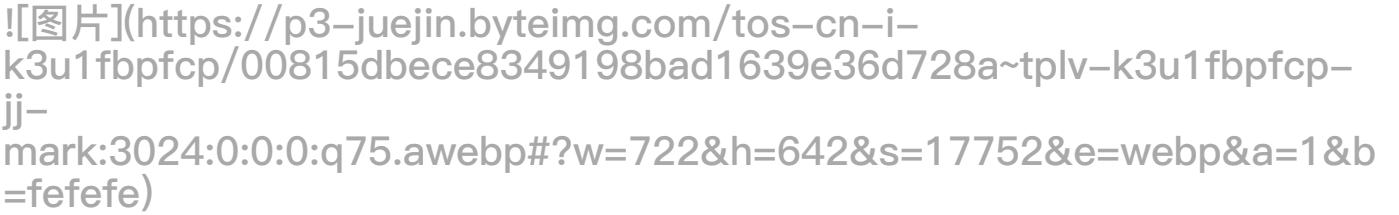
所以我们强烈建议主动声明依赖，排除工程构建隐患，这也是maven官方建议的做法，换言之，每个jar包都主动声明自己的依赖，将有效缓解这类问题。

但是由于早期团队并没有要求必须主动声明依赖的规范，加之历史存量的jar包较多，我们为了防止在线上出现NoClassDefFoundError问题，自研了「Maven类依赖检测插件」集成到流水线中，尽可能将问题前置到上线前暴露。

4.2 自研类依赖检测Maven插件

4.2.1 实现原理

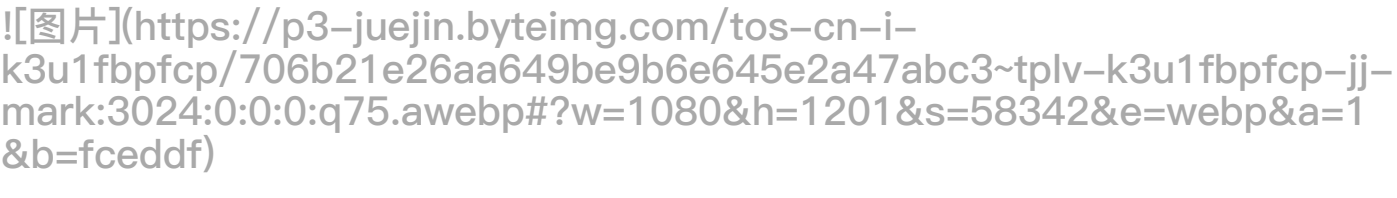
一个Spring项目的类依赖，主要包含两个部分，一个是项目类的直接依赖类以及通过传递依赖产生的间接依赖类，另一个是通过依赖注入方式产生的动态依赖。



直接依赖类会有直接依赖，也会有注入依赖，注入依赖类也是如此。把直接依赖类的分析称为静态分析，而注入依赖类的分析称为动态分析。

类依赖缺失分析的过程，就是静态分析+动态分析递归的过程。

完整处理流程如下图：



静态分析

以所有项目类为起点，获取直接依赖类，再根据上一步获取到的依赖类获取直接依赖类，循环这个过程，直到没有新的直接依赖类产生。这个过程中如果存在找不到的类，则认为该类缺失。

动态分析

动态分析需要完成的工作主要有两个：

1. 全局扫描，获取所有Spring bean并缓存。
2. 收集依赖注入的bean

总结

本文只是对架构治理过程中存在的问题以及庞大的工作量做了分析和工具建设，对于增量问题的控制，还需要建设一系列的准则和设计成熟的工程架构。

原文链接: https://juejin.cn/post/7371445601554186291