Istio 在 FreeWheel 微服务中的实践

FreeWheel 用 Go 语言和 k8s 重构单体系统，起初通过自研的 gateway 来提供统一的认证、授权、限流、监控功能，但由于 gateway 的中心化反向代理和复杂配置，成为制约微服务系统的瓶颈。因而选择 Istio 作为解决方案，实现 service mesh 架构。

Istio 架构包括四个主要模块：Istio Proxy，用于劫持 Pod 之间的通信；Pilot，为 Proxy 提供动态配置管理；Citadal，用于自动维护 mTLS 密钥；Mixer，分为两组：Policy 提供授权、Quota 等能力，Telemetry 提供监控数据收集能力。在 Pod 部署之后，这些组件以 SideCar 为纽带协同配合，Proxy 通过反向代理注入 SideCar，Citadal 自动刷新 secrets，Pilot 和 SideCar 建立连接、管理动态配置，Mixer 同样和 SideCar 连接并提供授权、限流、审计等服务。

FreeWheel 对原生的 Istio 做了一些改造，拓展了其功能。主要包括 SideCar 的扩展，加入了对业务系统的认证支持，将用户等信息以 header 的形式传入，并与 Istio 的原生功能对接；并扩展了 Mixer 的功能，支持对部分流量选择性的提供服务。

Jvm 问题定位典型案例分析

本节课讲了 jvm 问题定位的 4 个经典案例。

第一个是类加载死锁。案例中线程 Dump 没有检测到死锁而有不少线程阻塞在 Class.forName()方法下，经 jstack 分析，可知在代码中没有显示的使用锁，但在类加载过程中隐式地加了锁。这样是为了防止多个线程加载同一个类的时候重复分配内存。当类加载器的 loadclass 方法中包含有加锁的逻辑时，多线程情形下就会导致死锁。

第二个是 finalReference 的堆积。FinalReference 用于 GC 过程，指示一个对象是否可以被回收掉。类加载的时候根据类里是否含有非空的 void finalize()方法决定对象是否能进入 ReferenceQueue。只有队列中的 finalize 方法执行后，对象才能被回收。当队列中 finalReference 堆积过多时，就会导致对象清理缓慢，内存难以释放。

第三个问题是堆外内存泄漏。主要是指 heap 利用率低但依旧出现内存溢出的问题，触发这一问题的场景包括不足以触发 GC 的小对象（DirectByteBuffer）关联了较大的内存，导致内存无法释放，或者大对象被移入老年代，无法被 young GC 即时清除.

第四个问题是 Young GC 拉长。Young GC 分为 mark，copy 和 sweep 三部分。其中标记对象的多少直接影响耗时的长短。在模拟的场景中，堆积的 StringTable 会导致 Young GC 的拉长，而当 StringTable 被 Full GC 清理后，Young GC 又变短。

通过软引用和弱引用提高内存使用效率。

强引用（比如 new 出来的对象）是最难被 GC 回收的，宁可虚拟机抛出异常，中断程序，也不回收强引用指向的实例对象。软引用对象在堆内存充足的情况下不会被垃圾收集器回收，反之则会被回收。弱引用对象则更弱一些，不论内存是否充足，都会被回收。

利用其特性，软引用可以被用作缓存，读取软引用对象比从数据库中读取速度更快。内存空间足够时，无需回收软引用；回收时并不影响业务流程。弱引用对象会被自动回收的特性，可以带来“自动更新”的好处，而不用手动解除关联关系。

Java 自动内存管理技术的现状和未来

Java 内存管理涵盖三个方面：内存的分配、管理和回收，这三者都是通过 GC 来实现的。世界上第一个 GC 算法诞生于 1959 年，近些年先后出现了 CMS，G1，ZGC 等垃圾收集器。如今 Java 系统愈发复杂，给内存管理带来了新的挑战：复杂的生命对象周期，难以预料的内存使用方式，以及多线程等。针对这一现状，互联网公司有若干选择：使用 Java 自带的收集器，深度定制，或者商业版收集器。

理论上最好的 GC 是不用做 GC。相应的尝试包括自动寻址的 Hbase Bucket Cache 和 GCIH。针对不同的场景还有不做 GC 和手动 GC 等想法。

如何降低内存管理的开销。这里提到了 RI 的 ZGC 收集器，可以降低停顿时间，相应的会牺牲 CPU 的通透性。还有一种想法是把对象的引用链压缩在一起，从而减少需要管理的对象数目。作者提到了阿里的一些探索，像把 VM 和应用结合在一起，实现局部有序的对象分布和回收。在此基础上又展望了性能更高的新硬件对内存管理的颠覆。

总体而言，目前的现状是垃圾回收仍旧依赖暂停，并没有通用的收集器。可以期待的是标准 GC 接口，大堆无暂停的 GC 等。

微服务——构建持续交付与 DevOps 架构

对于客户需求的快速反应，推动了持续交付的产生。持续交付的质量不但取决于完备的工具链，也受到软件架构的影响。适合持续交付的软件架构需要易于部署和调整，而微服务架构则满足这一特性。微服务架构具有部署独立、时间短、步骤简单、零宕机等特性，相对于单体系统更易于调整，因而适合持续交付。当然微服务也有一些缺陷，比如说服务数量过多，服务间复杂的协调交互问题，技术栈的多样性带来的维护成本提高等问题。可以通过强化代码测试环节，以及引入问题定位工具等来处理这些问题。并非所有场景都适合微服务化，如果微服务带来的复杂性等问题超过了好处，那么这种转变是得不偿失的；需要因地制宜，选用最合适的架构；引入 DevOps 有利于提高微服务系统持续交付的能力。

内部方法调用时，为什么 Spring AOP 增强不生效

在类的内部方法中调用带有 AOP 注解的方法，AOP 增强不会生效。这是因为 AOP 是通过动态代理对 Bean 实现增强，作用于动态对象之上，内部方法调用并不经过这一增强逻辑。如果想要使增强生效，就不能使用 this 调用，而是应该选择增强过的 Proxy 对象。具体实现方式包括：把自身的 Bean autowire 进来，通过 ApplicationContextAware 获取到 Bean，或者通过 AOP 的一些辅助方法（AopContext.currentProxy()）获取 Proxy 对象。@Lazy 注解可以解决 Autowire 自身 Bean 时的问题。

携程容器云弹性能力构建之路

携程容器云的技术栈经历了一个从 Mesos 到 k8s 的迁移过程。其架构分为三层：基于 PaaS 的用户产品层，基于微服务和 Controllers 的中间层，基于云基础设施的基础层。其发展愿景是从资源提供者向服务提供者转型。

短期来看，坚持 Mesos 成本低，但长远来看 k8s 更有优势。在 k8s 的 Deployment 和 Statefulset 模型中，携程选择了后者，因为有固定 IP 的优势。在架构方面，要求同时支持 Mesos 和 k8s，以及单应用透明跨集群。应用架构在保留对 Mesos 支持的前提下，为存量向 k8s 的迁移埋下伏笔。迁移过程需要克服无用户参与、跨系统操作事务性、自动化、用户沟通的等挑战。

新的架构实现了调度、存储插件、网络、外部 Agent 等方面的功能扩展。迁移后分析资源池状况，发现了 CPU 利用率低和内存利用率周期性差的问题。可以根据实例的多少将应用划分为热点和长尾应用。为了提升资源利用率，可以对热点应用功能进行横向缩容或 hpa 整合，长尾应用纵向缩容或调整 xms，此外还有超分混部等策略。

应用线程过多时，会遇到 CPU throttle 的问题。可以通过调整 jvm 参数、放宽 CPU-quota 等来解决。对于 k8s 稳定性的问题，可以通过搭建破坏性测试环境、持续社区跟踪、防御型架构来规避风险。

容器云的弹性可以从三个角度来描述：基础设施的管理和运维，用户的交付，成本的控

制。k8s 有利于实现应用和基础设施的服务化，从而实现运维的自动化；也在用户和基础设施之间勾画了清晰的界限，有利于用户专注于业务，而成本控制则是构建弹性能力的必然结果。

从微服务到 Serverless 架构应用与实践。

所谓的传统架构包括单体系统和服务化框架，具有开发方便、部署管理简单等优势，但随着系统的成长和扩张，架构日益庞杂和混乱，带来了诸多问题。为了解决传统架构造成的不便，微服务架构应运而生，它的特点包括统一接口、容错处理、全链路服务监控、服务注册发现、统一代码框架以及服务分级等。微服务架构具有解耦业务逻辑实现原子化服务，服务独立部署运维，支持多语言开发等优势。

但随着业务的快速变化，轻量级的微服务也会越变越大，直到优势不再。它带来的是一种“虚假的安宁”。微服务架构的推进需要过渡时间，单体应用仍旧不可或缺，微服务长期迭代同样需要重构，以及开发测试的复杂性增加，这些都是微服务难以避免之殇。

微服务架构的主要痛点在于如何更快地迭代更新、即时上线，如何将开发和部署运维解耦，如何降低编程框架的复杂性，如何避免累积迭代后的重构，如何实现运维的智能化。为此又产生了 Serverless 架构，它可以实现动态的扩缩容，从而提高资源利用率，对于复杂的微服务系统能有效的降低成本。同程艺龙的 Serverless 平台能够集成原有的容器平台，实现容器级隔离，通过 gateway 访问应用，根据流量自动拉起服务并动态扩缩容，并可以实现编程框架和 IDE 的云化。主要用于支持网页和活动推广，以及变化量大的后台。

GraalVM 及其生态系统

Java 源码由 javac 编译成可跨平台运行的字节码，字节码在 VM 中有两种执行方式：解释器解释执行和 JIT 即时编译器编译成机器码后执行。Graal 编译器被设计来取代 c2 编译器，与 c 语言写的 c2 不同，Graal 基于 Java 语音编写，更易于理解和维护，相对于老版的 c2 性能有明显的提升。

Graal 不只是支持 Java 的编译器，所有可以被编译成字节码的语言都可以运行在 GraalVM 上，比如 Scala，以及 Truffle 框架支持的 JavaScript、R、Ruby 等，使得不同语言之间可以共享数据结构。

Graal 支持针对特定场景机器码的去优化，可以切换回解释执行，能够节省编译时间，易于执行。Graal 支持基于图的中间表达形式，能动态地展示编译器的编译过程；可以实现部分逃逸分析，减少堆空间的使用，降低 GC 频率。Graal 实现了模块化的架构，把 jit 从 VM 中抽提出来，通过 JVMCI 与虚拟机交互，编译器可以向 VM 分发请求，获取元数据，以及部署代码；与 c2 相比，相对于上层应用是透明的，可以直接编译测试机器码。

Graal 的生态系统主要包括 Truffle 和 Substrate VM。