# 第三篇 分布式系统

## 什么是分布式架构

多个节点组成的系统，节点通过消息传递协同工作。

需求：互联网的兴起，大型网站的各种高并发访问、海量数据处理等场景越来越多。(突破单机上线)

目标：实现网站的高可用、易伸缩、可扩展、安全等。

### 3.1.1 大型网站架构演化

大型网站：高并发+海量数据+业务复杂



#### 单体架构

当网站流量很小时，只要一个应用，所有功能都部署在一台机器，方便维护，成本又低。访问量增加，可做集群。

特点：一般采用昂贵的大型主机,结构简单，不用考虑各节点之间的分布式协作问题。

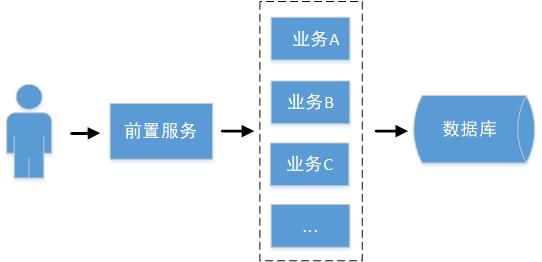
问题：单体架构发展大一定程度，项目复杂度高，难于维护、扩展性差，影响开发效率等。

实例理解

企业软件设计需要满足多种多样的业务需求。因此，一个特定的应用软件会包括有几百个功能项，而所有这些功能项都打包进了一个单体的应用中。典型的例子有，ERP、CRM等其他各种各样的软件。对于这种野兽级别的软件应用、部署、排错、扩展和升级工作都是一个个噩梦。

#### 垂直架构

又叫业务拆分。将复杂的业务拆分成不同的产品线，分而治之，以降低复杂度，提升效率。



如图垂直架构最多的还是通过同一DB整合业务。

* 拆分：拆分后的应用独立部署维护。
* 整合：超链接/消息队列/同一个DB。或者这几种方式的组合。

问题

随着业务拆分越来越小，复杂度呈指数级增加，部署维护越来越困难。

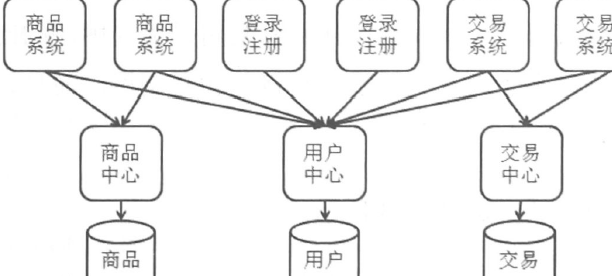
1）跨应用的交叉调用越来越复杂

2）db连接数目是服务器规模的平方。导致db资源不足。

#### 分布式架构

抽取核心业务，作为独立服务，使用独立的DB。

大型网站架构演化到这里，大多数技术问题得以解决。因此，目前一些云计算平台，将计算作为基础资源出售，中小网站按需付费。



架构的本质

从单体架构到服务化架构，应用数量都在不断的增加，基础组建下沉，业务系统上浮。

变化的原因:用户规模，业务带来的复杂度变化（并发量/数据量/业务复杂度）。

目前通过引用jar包，耦合度较高。

#### SOA

面向服务的架构：就像面向对象的程序由一个一个类组成。1）SOA的软件可以理解为由一个一个服务组成。2）每个服务独立发布独立运行）。

问题：服务越来越多，交互越发复杂，不可避免出现资源浪费（计算机资源）的情况。

目的：服务治理，提高集群利用率。

方法：增加调用中心，基于访问压力实时管理集群容量。

在这个阶段可以使用WebService或者dubbo来服务治理。

#### 微服务

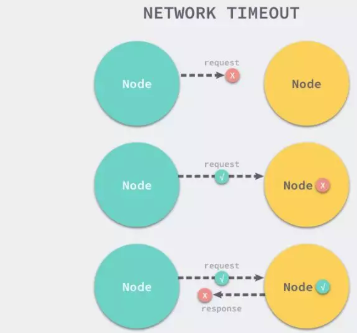
微服务架构是一种架构模式，强调彻底的服务化（拆分粒度更小）和组件化

### 3.2.1 网络请求

网络通信经常是不稳定的，网络通信可能会包含成功、失败以及超时三种情况。

在分布式架构中，服务之间基于网络通信，所以会出现障碍。

由此带来的问题



比如网络延迟，重复处理，消息丢失等问题。

**重复消费**

下游业务接口保证幂等。

Kafka提供exactly once但consumer的下游必须是kafka。

**消息丢失**

上游：重试

网络请求超时是造成分布式复杂的根本原因。

* 可靠性：节点通过网络来交换信息，网络超时引发一系列问题。
* 复杂度：由于可靠性问题，增加系统处理的复杂度（比如分布式事务）。
* 单点故障：一般采用普通计算机，服务器数量多，因此宕机概率大。

分布式 VS 集群

分布式：一个业务分拆多出个子业务，部署在不同的服务器上

集群：同一个业务，部署在多个服务器上

通过接口分解系统解耦性，不同子系统通过相同的接口描述进行服务调用。

### 3.2.3 数据治理

SpringBoot+MongoDB

Todo

注册中心

[Reference](https://mp.weixin.qq.com/s?__biz=MzIyMzgyODkxMQ==&mid=2247483885&idx=1&sn=73321a5574045c47fccb3013e47ec0e9&chksm=e8190f2ddf6e863b6016bd90b396cf9eaa2365ff42d4120299de2ec5b4f510ad071ce1d079f8&scene=21#wechat_redirect)

需求：1）根据负载，动态调整服务节点数量。2）节点down掉，保证服务可用。

方案一 Redis：

1) 频繁轮询

2) 缓存解决轮询，带来滞后性问题

3) 节点新增或者删除（down掉或kill）服务节点，无法及时清除

方案二 zookeeper

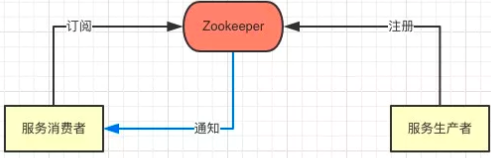
Zookeeper

Zookeeper是一个分布式一致性框架。主要用来解决分布式集群中应用系统的一致性问题。

本质：缓存+推送

观察者模式

缓存数据+订阅发布



服务生产者删除时，zookeeper通知消费者更新本地缓存并删除保存的节点信息。

使用场景

配置中心：统一的配置中心，不需要在每个服务器单独配置（通知模式）。

分布式锁：瞬时有序节点

TODO

机器间存在依赖关系时，就需要处理协调问题。如指定一台作为master主节点接收任务，并将任务分为其他从节点，涉及到主节点选举和主节点是否存活的判断。另外这些机器作为RPC服务提供者提供服务时，需要提供服务注册查询功能。当我们想实现一个分布式锁时。这些都是Zookeeper可以发挥的场景。总结来说，

* 服务发现
* 分布式锁
* 分布式应用可以基于它实现同步，配置管理，集群管理，命名名空间
* 崩溃检查
* 存储分布式元数据
* 选举主节点

节点类型**:**永久节点;瞬时节点

是否可以做分布式的数据库？

https://www.zhihu.com/question/22116083

CMD

/zkServer.sh start/status

路由策略

考虑重点：1.尽量保证各节点的连接均匀。2.增删节点是否要做 Rebalance。

均衡算法

* 轮询：会出现新增节点分配不均。
* Hash取模：类似HashMap，也会出现轮询的问题。可做rebalance，索引客户端重连。
* 权重：基于监控，根据负载调整，高就调低权重，低就提高权重。

推送路由

服务监控

* 各节点OS，
* JVM,GC等
* 业务监控：比如消息系统的在线数。

## 微服务

### 3.2.1 微服务基础

（源于2014年的一篇文章“Microservices”，本章内容将基于SpringCloud框架体系讲述）

#### 什么是微服务？

微服务是一种架构理念，提出微服务的设计原则，强调彻底的服务化和组件化。一个组件就是一个产品，可独立对外提供服务,独立运行空间，独立的DB。微指组件拆分粒度更小。

微服务架构从互联网企业兴起的，互联网企业特点：

1)用户规模大（并发量高数据量大）。2)发布/迭代周期短（不同子系统发布周期不同）。

微服务 VS SOA

不同时代的产物，要解决的具体问题不同。但没有本质的区别，并且微服务是由SOA演化过来的。在SOA用于服务治理，提高集群资源效率。微服务架构从互联网企业兴起的。解决出现的新问题。

[微服务架构，多“微”才合适？](https://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651961970&idx=1&sn=60631fc64b42bff46dcf2bc4eab60b13&chksm=bd2d0fae8a5a86b87861b5cf71abb334d42e8cff3315f13dc5aac560c2c350e425fbd8bdfd90&mpshare=1&scene=1&srcid=&pass_ticket=cW2JCTQ4rkqDxvQI35OMgWpyjAos1512GIFXJP3ta6rRXyKD1LcWRqRcSA3RsKxL#rd)

特点

从下面特点感受微服务是什么样子

* 通信轻量级：通常基于HTTP的RESTful API
* 技术异构：具体到一个服务，根据需要选择不同语言、工具。
* 复杂度：不同于接口直接调用，通过REST需考虑故障，过载，消息丢失等

为什么用？

无需关心过多底层细节，而可以聚焦于业务本身，快速构建行业应用。消除信息孤岛，形成良性的业务生态。

好处

* 拆分力度更小更彻底，单一职责，分工明确。
* 微服务架构带来更好的扩展性，可靠性等

什么时候用

微服务治理框架，基于Spring Boot开发。

#### 服务治理

服务的管理，可视化治理界面，服务构建发布，分布式事务，流量控制，监控告警，服务契约，链路跟踪，灰度发布，服务降级等等。

SOA服务治理

1）注册中心：管理所有服务

2）路由选择、负载均衡及容错处理；

3）服务监控与统计；

4）服务过滤（黑名单、白名单）；

5）服务升降级，权重调整；

6）服务状态检测、监测；

7）服务权限控制。

8）服务依赖关系；

### 3.2.2 SpringCloud

服务治理框架。

* 微服务是一种架构理念，从理论为具体的技术落地提供了指导思想。
* Spring Boot是一套快速配置脚手架，可以基于Spring Boot快速开发单个微服务；
* Spring Cloud基于Spring Boot实现服务治理工具包，关注全局的服务治理框架。

#### 核心组件

Eureka：用于服务的注册于发现。

Feign：服务的调用以及均衡负载。

Hystrix：处理服务的熔断防止故障扩散。

Spring Cloud Config：服务配置。

Gateway：服务网关

#### 支撑服务

Dapper/zipkin+Sleuth：服务链路追踪

ELK：日志聚合。Elasticsearch+Logstash+Kibana

Spring Boot Admin：服务监控。

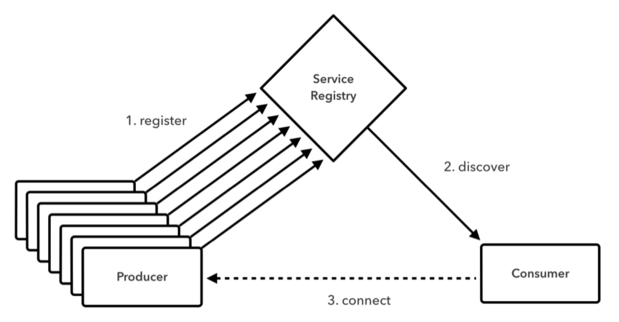
Docker：部署发布。

项目根据实际需求和业务场景，选择使用到的组件。

Eureka

分布式环境，实现服务的注册和发现，满足节点动态扩展的需求。

* 动态调整服务节点数量，服务自动注册/删除，不用修改配置文件。
* 负载均衡在客户端实现，流量不经节点。
* 节点down掉，保证服务可用。



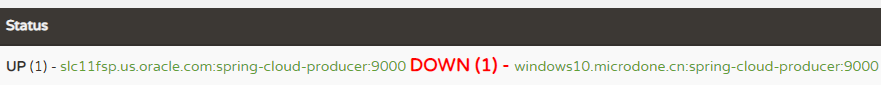
可用性

保证AP，withOutC，更适用注册发现场景。因为注册服务更重要的是可用性，可以接受短期内达不到一致性的状况。

Zookeeper: 场景：分布式协调,保证CP，withoutA, zookeeper节点点数据是同步的，极端环境可能丢弃一些请求保证一致性(client需要重新请求)。例如：当master失联时，剩余节点选举期间整个zk集群都是不可用的。

自我保护模式

节点短时间丢失大量心跳连接(可能发生了网络故障)，这个节点会进入“自我保护模式”，同时保留那些“心跳死亡”注册信息。以防还有客户端向其发起请求。故障恢复，这个节点会退出此模式。Eureka的哲学是，同时保留“好数据”与“坏数据”总比丢掉任何数据要更好。



集群

zookeeper和eureka集群都是一种去中心化架构，每个节点保存完整实例注册服务信息, 是平等的。

原理

Eureka Server：在运行过程中，Server之间会定时同步实例的注册信息。

Service Consumer：启动时，从Server获取服务信息并定时更新。

Feign

基于http的rpc服务调用框架，简化服务调用。整合了Ribbon，Hystrix和RestTemplate，。

配置

可直接用配置Ribbon的方式来自定义调用的参数。负载均衡策略

* RoundRobin：轮询
* Random
* WeightedResponseTime
* ZoneAware

feign声明式Web客户端

1. @FeignClient(name= "spring-cloud-producer",fallback = HelloHystrix.**class**)
2. **public** **interface** HelloRemote {
3. @RequestMapping(value = "/hello")
4. **public** String hello(@RequestParam(value = "name") String name);
5. }

Request Interceptor

1. @Configuration
2. **public** **class** HmilyFeignInterceptor **implements** RequestInterceptor {
3. @Override
4. **public** **void** apply(**final** RequestTemplate requestTemplate) {
5. Context Context = ContextLocal.get();
6. requestTemplate.header("context", GsonUtils.toJson(context));
7. }
8. }

Todo

Spring Cloud Loadbalance

Spring Cloud Config

随着项目规模变大，每个项目都散落着各种配置文件，如果采用分布式的开发模式，需要的配置文件随着服务增加而不断增多。某一个基础服务信息变更，都会引起一系列的更新和重启。

* 集中管理各环境的配置文件
* 配置文件修改之后，可以快速的生效。（通知）
* 可以进行版本管理

springboot项目只有在启动的时候才会获取配置文件的值，更新时需手动refresh

Spring Cloud Bus

本质是利用了MQ的广播机制在分布式的系统中传播消息，目前常用的有Kafka和RabbitMQ。

* Hystrix

熔断器，用于服务容错，处理服务的熔断防止故障扩散。

[Reference](http://www.ityouknow.com/springcloud/2017/05/16/springcloud-hystrix.html)

如果它在一段时间内侦测到许多类似的错误，会强迫其以后的多个调用快速失败，不再访问远程服务器，从而防止应用程序不断地尝试执行可能会失败的操作，使得应用程序继续执行而不用等待修正错误，或者浪费CPU时间去等到长时间的超时产生。熔断器也可以使应用程序能够诊断错误是否已经修正，如果已经修正，应用程序会再次尝试调用操作。

熔断器模式就像是那些容易导致错误的操作的一种代理。这种代理能够记录最近调用发生错误的次数，然后决定使用允许操作继续，或者立即返回错误。

* Gateway

Gateway将外部request均衡分发给后台服务端。

* 简化调用：后端实例动态变化，即服务的访问地址。网关简化外部客户端调用。
* 安全：在网关认证，无需在每个微服务API认证。

在网关层实现非业务逻辑，比如监控，提高业务灵活性。此外在网关中实现 日志/限流等

注册中心 VS Gateway

* 注册中心：构建内部服务，。
* Gateway用于服务外部调用者。

Nginx API Gateway

API Gateway诞生为解决云端问题，而Nginx仅作为反向代理，没有上面的功能。

* Spring Cloud Zuul

简单使用

1. zuul.routes.hello.path=/hello/\*\*
2. zuul.routes.hello.url=http://localhost:9000/

通过url映射的方式来实现zull的转发有局限性，比如每增加一个服务就需要配置一条内容，另外后端的服务如果是动态来提供，就不能采用这种方案来配置了。

服务化

服务名与服务实例地址的关系在eureka中已存在，将Zuul注册到eureka即可发现其他服务。

1. zuul.routes.api-a.path=/producer/\*\*
2. zuul.routes.api-a.serviceId=spring-cloud-producer

serviceId是服务名，zuul默认路由规则如下：<http://zuul_host:zuul_port/serviceId/path>,

比如 http://localhost:8888/spring-cloud-producer/hello?name=%E5%B0%8F

路由熔断

后端服务出现异常的时候，我们不希望将异常抛出给最外层。

断路器就只有在该服务的所有实例都无法运作的情况下才能起作用。这种时候，断路器的形式更像是提供一种友好的错误信息，或者假装服务正常运行的假象给使用者。

Spring Cloud Gateway

Zuul（1.x） 基于 Servlet，使用阻塞 API，它不支持任何长连接，如 WebSockets，Spring Cloud Gateway 使用非阻塞 API，支持 WebSockets，支持限流等新特性。

设计准则

服务大小

不是越小越好，

设计原则

* 单一职责
* 保证微服务设计能支持服务的敏捷/独立地开发和部署；

微服务划分的粒度？**M**

数据管理

每个服务有自己的DB,去中心化的数据管理

服务治理

去中心化的治理，服务发现可以在API网关级别实现。

跨服务调用

ELK日志处理

请求加traceID记录日志。集群环境也会遇到，

ELK工具

Elasticsearch：分布式搜索和分析引擎。全文检索，结构化和分析。

Logstash：实时传输数据的管道。可🏠过滤等

Kibana ：es的展示界面，展示数据，分析数据

Zipkin捕捉应用中未捕捉的异常，加了Hystrix看不出异常，

Sleuth+Dapper/zipkin

分布式服务追踪。主要三部分 数据收集🡪存储🡪展示

Spring Boot Admin

服务监控，监控内容。

Spring Boot Actuator 提供了对单个 Spring Boot 的监控，信息包含：应用状态、内存、线程、堆栈等等，比较全面的监控了 Spring Boot 应用的整个生命周期。

Spring Cloud Stream

[Reference](https://mp.weixin.qq.com/s?__biz=MzI4ODQ3NjE2OA==&mid=2247484734&idx=1&sn=9cb886cf26c286bf1541a75b3459b823&chksm=ec3c9859db4b114f1564bba38f993b911319e9c1338747871751050778d8c2a735cfee0718a8&mpshare=1&scene=1&srcid=#rd)

### 3.2.3 Spring Boot

简化应用配置，更容易使用spring。方便搭建项目或构建一个微服务。

* 简化配置：封装常用套件，比如mybatis、hibernate、redis、mongodb等
* 自动管理依赖。
* 部署简单：内嵌Web容器，如 Tomcat

缺点是集成度较高，使用过程中不太容易了解底层。

注解说明

[Reference](https://blog.csdn.net/javaloveiphone/article/details/52182899)

@Configuration 相当于xml中的<beans>，用于配置spring容器。

spring-boot 2.0

#### Starter

Starter集成框架并提供缺省参数，是实现自动化配置的关键

spring-boot-starter-web

集成Spring MVC,Tomcat，Jackson等库

spring-boot-starter-data-jpa

基于hibernate的JPA框架。包含datasource，driver/事务管理器等。

Spring boot Actualtor

Actualtor比较全面的监控了Spring Boot应用的整个生命周期。

监控内容：应用状态、内存、线程、堆栈等等

查看方式：接口返回json

Spring Boot Admin

封装actuator接口，用UI页面显示，方便查看。可直接修改logger的level

package结构

com.example.myproject

+- Application.java

+- domain

+- service

+- controller

#### Spring Boot Test

[Reference](https://fanlychie.github.io/post/spring-boot-testing.html)

装载 Spring 应用程序的上下文资源

SpringBootTest代替spring中的ContextConfiguration

@SpringBootTest

webEnvironment

* MOCK 不启动容器
* RANDOM\_PORT 启动容器，使用随机端口
* DEFINED\_PORT 启动容器，使用定义端口
* NONE

## 中台战略

### 什么是中台？

在传统的前台/后台架构中，各个项目相对独立，许多项目都在重复发明轮子，开发效率低。

分类

按功能和角色中台可以划分成4个维度

业务中台

技术中台

数据中台

算法中台

技术中台

中台能力不是只堆砌技术组件，不是缺消息组件装个kafka，然后就抛给业务开发去使用，而是思考如何让业务开发更好更快速地上手使用。能非常快速的提供给业务和其他技术部门使用，维护成本也低，这才是真正能为团队和业务带来价值的地方。

成本

对于企业来说，方案牛不牛逼也不重要，关键是能帮助用户找到效率、质量与成本的平衡点，或许才算是一个合格的「技术中台」

参考

[什么是中台](https://mp.weixin.qq.com/s?__biz=MzU0OTE4MzYzMw==&mid=2247487400&idx=2&sn=524b08946b0f9aec29ac9669e01a1499&chksm=fbb28656ccc50f403478d5ff3e001f034e4dd2bf7d6d4c829b6c2b4bf7f7d8b4858008d31147&mpshare=1&scene=1&srcid=&sharer_sharetime=1568889891725&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=cc53f48e7cf7aa7e8c3cfbd9e2b4887f104a3e9a3a5f73d52834f01c6dea97965e19cdcf0a8f14a64ac00d082c5cfc3685c1120c9f76629fb5c32ef9f8ebc9b40234f5de450ce5c11b7a78fd22979964&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62060833&lang=zh_CN&pass_ticket=b9Rj%2BB3zvC3erxrn0M%2BHi2SXUlNByZMhxuDzh4FBR4Zwi4r3btNcw23znOcd%2BUUu)

[企业微服务中台落地实践和思想之我见](https://mp.weixin.qq.com/s?__biz=MzAxNjk4ODE4OQ==&mid=2247485436&idx=1&sn=6dd74854b3a8dda85d964a1ccdb949c3&chksm=9bed268eac9aaf988773d3a7528964491a81893a9592784018b4bf7a201d9a4efb7b60200161&mpshare=1&scene=1&srcid=#rd)



### 3.4 部署平台

Docker

容器即应用，面向容器开发

如你所知，为了加快项目进度，你有一个待办事项清单：

* 拉取代码基线
* 安装外部工具，如数据库、缓存和一些额外的工具和服务
* 给外部工具打补丁和升级
* 配置数据库和服务以使他们可以通信
* 开始祈祷（可能需要好几次）
* 调试问题（至少要30几次）

想象一下，如果在将应用程序的代码基线拉出存储库之后，您只需运行几个命令行（可能只有一个），可以使整个应用程序的环境准备就绪。听起来很酷，对吧？

#### 3.3.1 DevOps

[Reference](https://zh.wikipedia.org/wiki/DevOps)

* 更小、更频繁的变更──意味着更少的风险
* 让开发人员更多地控制生产环境
* 更多地以应用程序为中心来理解基础设施
* 定义简洁明了的流程
* 尽可能地自动化
* 促成开发与运维的协作

##### Docker

屏蔽程序运行环境的差异。

3.3.2 Jenkins

持续集成

#### 中间件

#### 后台服务

后台服务：主要包括消息系统，分布式缓存，分布式数据访问层和任务调度系统。（中间件？）

分布式数据访问层：分表分库

## 3.5 消息队列

[Reference](https://mp.weixin.qq.com/s?__biz=MzIwMTY0NDU3Nw==&mid=2651938493&idx=1&sn=d0a18b9dd5d587726196310df1811d8d&chksm=8d0f31f3ba78b8e5541ea73b3d093db52b92ff3e190bb7f594f03c9b16778db00a4e15f3b52a&mpshare=1&scene=1&srcid=#rd)

消息队列是分布式系统中重要的组件。

### 使用场景

* 系统集成：分布式系统集成方式除了直接服务调用还可以用消息队列。
* 服务解耦：基于订阅发布，需要消息的系统直接订阅。比如下单通知，耦合每个通知对象。
* 异步：非必要业务异步执行，加快响应速度。比如下单成功，发短信，发红包。
* 削峰限流：请求加入消息队列，将处理的请求控制到server能处理的范围内，比如秒杀系统类型。比Redis实现的功能强大。

MQ VS Queue

JDK Queue都是简单的内存队列。MQ 用于分布式环境（思考hashmap和redis）。使用MQ增加系统复杂度，需要考虑可用行，一致性，传输可靠性，消息重复消费等问题。

MQ产品

* ActiveMQ：同RabbitMQ老牌的。份额较少，下一代改名Apollo。
* RabbitMQ：发布频率，社区活跃，适合中小型软件公司，简单，性能佳。
* Rocketmq：阿里开源，适合数据量大，高并发场景，适合大型软件公司，有足够资金搭建分布式环境，有定制化开发能力和精力。
* kafka：同样适合分布式环境。目前最火
* ZeroMQ：轻量级消息系统。
* redis：

集群

RabbitMQ: [Reference](https://www.jianshu.com/p/79ca08116d57)

### Kafka

消息投递语义

出现网络请求超时问题，消息发送者有两种选择

重试：消息的发送者通过重试对消息进行重发，可能会导致消息的重复发送与处理

不重试：可能导致消息的丢失

* 最多一次（At-Most Once）
* 最少一次（At-Least Once）
* 正好一次（Exactly Once）

分布式系统与消息投递

[Reference](https://mp.weixin.qq.com/s?__biz=MzU0OTE4MzYzMw==&mid=2247485980&idx=1&sn=a33f504b4ef6265c4ec9c2d18eaa5380&chksm=fbb283e2ccc50af415b1a4bdafd3e26b9d284c02194cc2d5d4414b93d6aa3acccd49851a0e7f&mpshare=1&scene=1&srcid=&pass_ticket=rMfcWI31qoBc%2BJleH2)

重复消费

可靠性传输

丢失消息：

Producer

MQ

Consumer

实现可靠性消息服务，保证最终一致性。[Reference](https://my.oschina.net/xliangbo/blog/1545040)

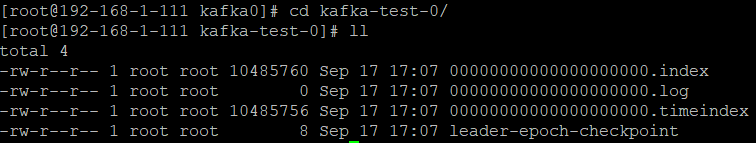
消息顺序性

消息顺序放同一队列，只用一个consumer去消费队列

Kafka

日志处理

日志存储机制



Log Retention Policy

关于Kafka的日志

日志分为两种：

数据日志

操作日志

ActiveMQ

基于JMS

## 3.6 分布式事务

从广义上来看，分布式事务其实也是事务，比如：垂直拆分Product Order User到三个库。

刚性事务

* 2PC：强一致性。阻塞提交，而且对性能影响太大；

柔性事务

一般在应用层实现，最终一致性。

* TCC：缺点要写很多补偿的代码，开发成本高。
* 本地消息表：消息表会耦合到业务系统，不太优雅；
* MQ事务：依赖于有事务消息的MQ中间件。同本地消息表不适合下游业务需要回滚场景。
* Saga：

Reference

[关于分布式事务、两阶段提交协议、三阶提交协议](https://www.hollischuang.com/archives/2591)

[柔性事务与服务模式](https://www.hollischuang.com/archives/2591)

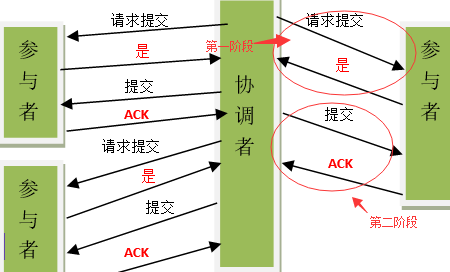
除了XA ，其他方案都会侵入业务层，需要设计实现正向和反向的幂等接口，往往会导致很高的研发和维护成本。

### 3.4.3 2PC

二阶提交协议，主要保证了分布式事务的原子性。

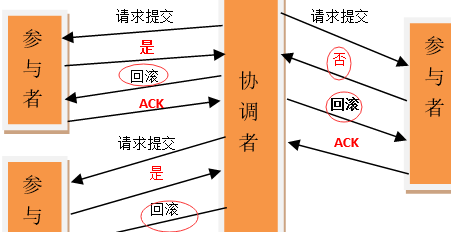
第一阶段：参与者收到协调者请求提交后即提交本地事务，但没commit

第二阶段：各节点执行commit



参与者：RM，协调者：TM

任何节点的响应消息为“中止“或询问超时，所有节点在第二阶段结束事务。



几个问题：

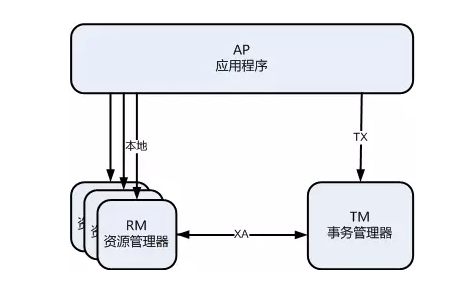
单点故障：协调者故障，导致所有参与者一直阻塞。

数据不一致：协调者故障（第二阶段），只有部分参与者收到提交请求并执行。

XA规范

XA是分布式事务处理（DTP）的标准协议，保证分布式事务的ACID特性，像本地事务一样。

DTP模型



* TM : Transaction Manager。
* RM : Resource Manager，由DB实现，比如Oracle、DB2都实现了XA接口
* XA 协议描述了 TM 与 RM 之间的接口

CRM ：通信资源管理器，比如消息中间件。

AP：Application Program。

二阶提交协议和三阶提交协议是XA的具体实现。

### 3.4.4 TCC

采用补偿机制，对每个操作注册一个与对应的确认和补偿（撤销）操作。

* try：业务校验和资源预留（DML）。
* confirm：确认执行业务操作（Commit）。
* cancel：取消执行业务操作（Rollback）。

1.为什么要try不直接一步完成？

单体数量多出错概率高。

2.为什么需要要资源预留？

2PC的阻塞并不适用高并发场景，TCC在应用层实现，单体资源预留即可提交，实现最终一致性。

case：同步调用

实现步骤

1. **public** Transaction interceptor(ProceedingJoinPoint point) {
2. Transaction trans=preTry();//1.buid trans
3. **try**{
4. returnValue = point.proceed();////2.execute try
5. }**catch**{
6. cancel(trans);
7. }
8. confirm(trans); //confirm
9. }

Aop

1. **public** Transaction preTry() {
2. Transaction trans = buildTransaction(point);//PRE\_TRY
3. saveTransaction(trans);
4. **return** trans;
5. }

TODO

[分布式事务的实现原理](https://mp.weixin.qq.com/s?__biz=MzAxNjk4ODE4OQ==&mid=2247485028&idx=1&sn=6473cc9f4854ebb689449e62391c337a&chksm=9bed2716ac9aae00e438a1221d22e1f0ba4f6ffe2f9808d1384d50eba8d9f02217a1c420e03c&mpshare=1&scene=1&srcid=#rd])

[增加一个协调服务](https://www.jianshu.com/p/2d2867349f66)

### 3.4.5 MQ事务/可靠消息服务

单体的事务直接提交🡪mq🡪所有的参与者

* 没有补偿发生异常将日志记录，admin查看
* 发起者service里做好业务校验，尽量确保全局事物成功提交。
* 每个单体都要有张事务表。

全局事务

角色：发起者和参与者

事务状态：开始,已提交,失败

Producter

1. **try**{
2. begin:tranId=save to tran\_table
3. exeService(commit)
4. updateTranTable(COMMIT)
5. }**catch**{
6. updateTranTable(FAILED)
7. }**finally**{
8. pushToMQ(tranId)//  只要调用过的下游方法且需要纳入同一tran的都要push
9. }

Schedule Check: 查询tran\_table中status=begin的,防止在2之后down机。

**Consumer**

push/pull from MQ

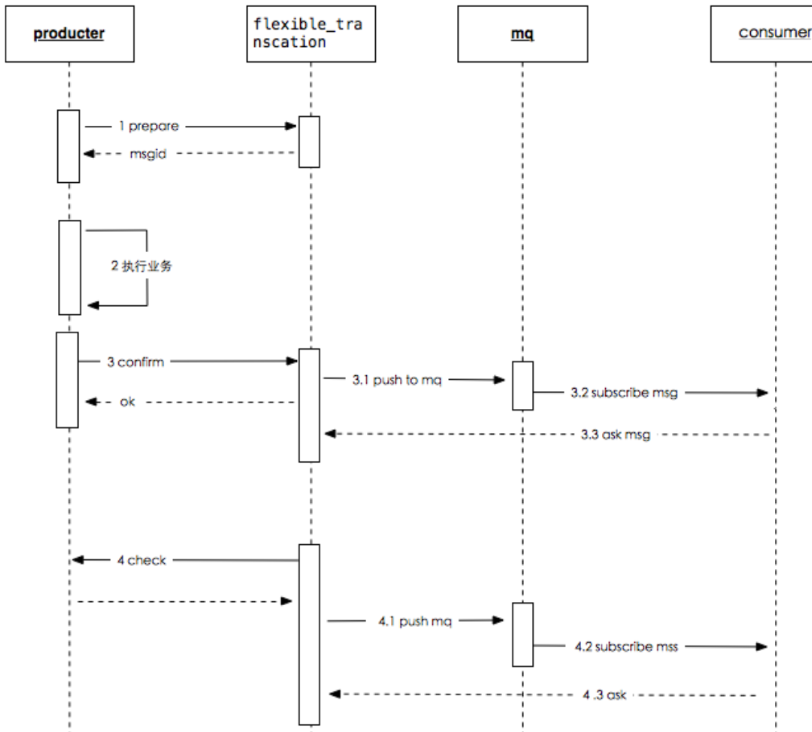
缺点：

* 实现难度大，主流MQ不支持。
* 下游业务需要回滚需要tcc补偿机制（同本地消息表）。

校验提到发起者

能确保事务消息发送成功且一定被消费。主流MQ不支持事务消息，比如Kafka和RabbitMQ。

[Reference](https://github.com/smartxing/flexible-transcation)



大致的思路：

每个单体应用都create tran\_table。对应用没有侵入。

本地消息表



优点： 一种非常经典的实现，避免了分布式事务，实现了最终一致性。

缺点： 消息表会耦合到业务系统中，如果没有封装好的解决方案，会有很多杂活需要处理。

Paxos算法

Chubby是google的paxos算法的实现，用来实现分布式锁、Master选举等功能。zookeeper是Chubby的开源实现，在Hadoop、Storm、Kafka等众多开源软件中都使用了zookeeper，各大互联网公司也都使用zookeeper作为分布式协调服务。

Saga

[Reference](https://mp.weixin.qq.com/s?__biz=MzAxNjk4ODE4OQ==&mid=2247485028&idx=1&sn=6473cc9f4854ebb689449e62391c337a&chksm=9bed2716ac9aae00e438a1221d22e1f0ba4f6ffe2f9808d1384d50eba8d9f02217a1c420e03c&mpshare=1&scene=1&srcid=#rd])

todo

JTA事务

Java事务的类型：JDBC事务、JTA(Java Transaction API)事务、容器事务。 常见的容器事务如Spring事务，容器事务主要是J2EE应用服务器提供的，容器事务大多是基于JTA完成，这是一个基于JNDI的，相当复杂的API实现。

JDBC事务有一个局限：一个 JDBC 事务不能跨越多个数据库.

### 3.4.1 理论支持

#### CAP定理

[Reference](https://www.hollischuang.com/archives/666)

一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance）这三项中的两项。

分区容错性可理解为分区

Warning

1. CAP的前提是P，有P才有CA的抉择，舍弃等同于单机系统。
2. 这里的一致性是强一致性。下面讲得牺牲一致性指强一致性，但可到达最终一致性。

CAP

1）Consistency

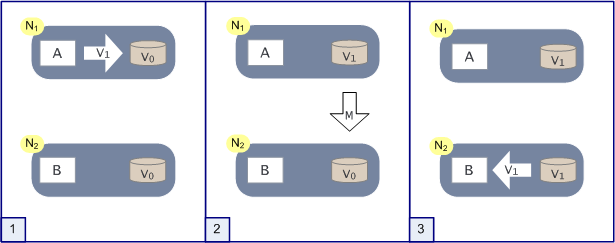
all nodes see the same data at the same time。强一致性。

2）Availability

Reads and writes always succeed，即服务一直可用，而且是正常响应时间。

3）Partition Tolerance

分区：指分布式，分成N个节点。分区容错性指分布式系统在遇到某节点或网络分区故障的时，单个节点仍能对外提供满足CA服务。



节点：N1,N2

一致性: N1和N2的数据应一样。V0=V0 。

可用性：请求N1和N2应得到立即响应。

分区容错性：N1和N2任何一方故障，彼此正常运转。

CAP权衡

P是分布式的基础，舍弃等同于单机系统。提升基础设施的稳定性才能保障P。

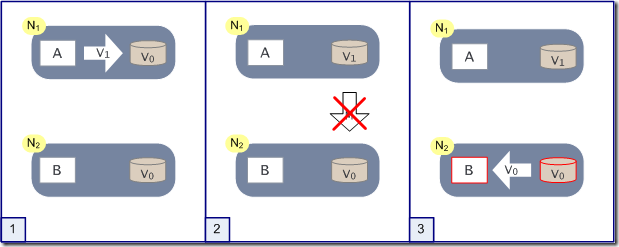
Without A

舍弃可用性。在分布式系统要想获得强一致性就要牺牲可用性。

典型的就是很多分布式数据库，在发生极端情况时，优先保证数据的强一致性，代价就是舍弃系统的可用性。如Redis、HBase等，还有分布式系统中常用的Zookeeper也是在CAP三者之中选择优先保证CP的。

Without C

舍弃一致性。



使用场景

在12306买票，查看时有票的，下单提示余票不足。牺牲一致性会影响一些用户体验，但是不至于造成流程的严重阻塞。准确的说这里舍弃的只是强一致性。退而求其次保证了最终一致性。

#### BASE理论

Base理论是对CAP理论的延伸。核心思想：即使无法做到强一致性，应用可以采用适合的方式达到最终一致性。

1）Basically Available

基本可用是指分布式系统在出现故障的时候，允许损失部分可用性，保证核心可用。

电商大促时，为了应对访问量激增，部分用户可能会被引导到降级页面，服务层也可能只提供降级服务。这就是损失部分可用性的体现。

2）Soft State

软状态是指允许系统存在中间状态。

分布式存储中一般一份数据至少会有三个副本，允许不同节点间副本同步的延时就是软状态的体现。MySQL replication的异步复制也是一种体现。

3）Eventual Consistency

最终一致性是指系统中的所有数据副本经过一定时间后，最终能够达到一致的状态。弱一致性和强一致性相反，最终一致性是弱一致性的一种特殊情况。

ACID VS BASE

ACID是传统数据库设计理论，追求强一致性。这里的C指数据的和数据关系的正确性。

BASE支持的是大型分布式系统，提出牺牲强一致性获得高可用性。

架构设计，偏理论，M2本阶段以架构设计为目标，预期时间为2年。

# 微服务

## 发布微服务

Docker create image

Build：docker build -t ndong211/my:v1 .（注意这里有点）

K8s创建示例

Pod：kubectl run my-test --image=ndong211/my:v1 --port=8080

Expose service：kubectl expose deployment my-test --type=NodePort

查看

kubectl get pods --selector="run=<my-test> " --output=wide

## 集群管理

### k8s集群管理

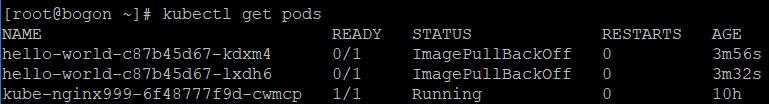
get

-o/output=json/yaml/wide|

#### 管理pods

查看pod

kubectl get pods [--all-namespaces]



删除：kubectl delete pods [pod-name]

Describe：kubectl describe pod <pod-name>

创建pod

kubectl run <pod-nane> --image=IMAGE [--port=port] [--replicas=NUM]

--replicas 副本数量

示例：

kubectl run nan-my1 --image=nan/my:v1 --port=8080

创建过程

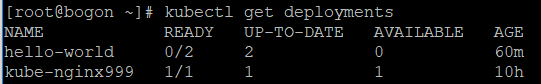
[Reference](https://www.kubernetes.org.cn/5759.html)

* 缺省创建资源deployment,缺省参数–restart-policy=Always
* 创建过程：kubectl http--> kube-apiserver --> deployment 持久化到etcd 创建ReplicaSet和Pod，每个 Pod 都被调度到合适的节点  Kubelet按PodSpec定义，创建并启动容器

#### 管理Deployment

查看

kubectl get deploy/deployments [name]



* 创建：kubectl create deployment NAME --image=image [--dry-run]
* 删除：kubectl delete deployments <deploy-name>
* 扩容：kubectl scale deployment nan-my1 --replicas 2
* 更新镜像：kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
* 回滚：kubectl rollout undo deployment/nginx-deployment
* 显示 ReplicaSet 的信息

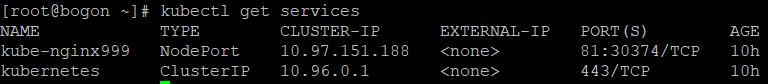
kubectl get replicasets/rs

kubectl describe replicasets

#### 管理Service

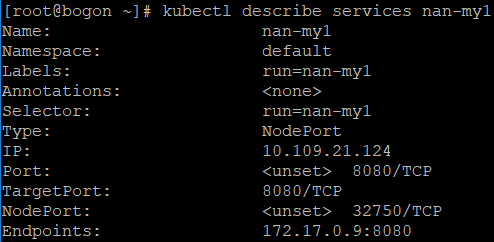
查看service

kubectl get svc/services



删除：kubectl delete services <name>

descripbe：kubectl describe services [label]



发布service

expose <-f fileName | type name> [--port=port] [--target-port=number-or-name] [--type=type]

target-port 容器端口

将资源暴露为新的Kubernetes Service，资源包括：

pod（po），service（svc），replication controller（rc），deployment（deploy），replica set（rs）

示例：

1. 将pod暴露给外网

kubectl expose deployment nan-my1 --type=NodePort

2.为rc创建service，并通过Service的80端口转发至容器的8000端口上。

kubectl expose rc nginx --port=80 --target-port=8000

3.由“nginx-controller.yaml”中指定的type和name标识的RC创建Service

kubectl expose -f nginx-controller.yaml --port=80 --target-port=8000

minikube查看service

minikube service kube-nginx999 [–url] [-n NAMESPACE]



#### 查看logs

查看指定pod的日志

kubectl logs <pod\_name>

kubectl logs -f <pod\_name> #类似tail -f

指定容器

kubectl logs <pod\_name> -c <container\_name>

Docker日志：docker logs <container\_id>

minikube查看日志：minikube logs

#### kubectl create

create -f fileName

通过配置文件名或stdin创建一个集群资源对象。支持json和yaml格式的文件。

示例：

kubectl create -f ./pod.json

#### 查看集群

kubectl cluster-info

/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

查看组件状态

kubectl get cs

kubectl get ep/endpoints

查看未授权的 CSR 请求

kubectl get csr (Certificate Signing Request)

#### Java Client

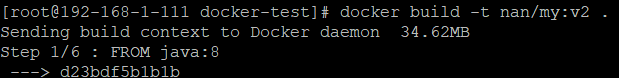
[Reference](https://blog.csdn.net/Ay_Ly/article/details/90404445)

### Docker

#### 3.1.1 镜像

创建镜像

docker build -t ndong211/my:v1 .（注意这里有点）



* --t/tag: 镜像的名字及标签
* 最后的 . 代表在当前目录下寻找 dockerfile,也可-f指定位置。

Dockerfile

FROM java:8

VOLUME /tmp

COPY ./my-1.0-SNAPSHOT.jar my.jar

RUN bash -c "touch /my.jar"

EXPOSE 8080

ENTRYPOINT ["java","-jar","my.jar"]

~

查看镜像

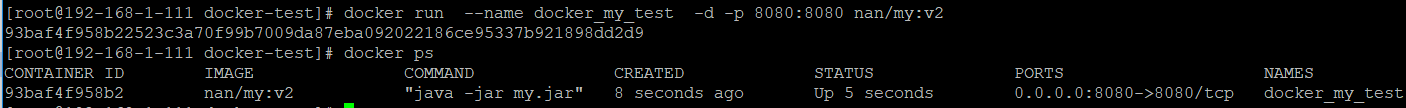
docker images [name]



#### 3.1.2 容器

创建容器

docker run --name names -d -p 80:80 repository:tag



* -d 后台运行
* -p 端口映射 -P容器端口随机

Docker run 用镜像创建容器 + docker start。

启动容器

启动：docker start/restart container\_id

停止：docker stop container\_id或container\_name

删除：docker rm container\_id

管理容器

logs

查看内部标准输出 docker logs -f bf08b7f2cd89

管理容器 docker exec -it docker\_nginx\_v1 bash

ps

docker ps [-a]

container id image command port names

d11a0358e403 nginx:v1 "nginx -g 'daemon ..." 0.0.0.0:88->80/tcp docker\_nginx\_v2

查看容器信息

docker inspect container\_id

#### 3.1.3 docker hub

[Reference](https://yeasy.gitbooks.io/docker_practice/repository/dockerhub.html)

docker pull/push <image-name>

push示例

docker push ndong211/my:v2

docker search +name

# 第二章DevOps

## 2.1 kubernetes

[Reference](https://jimmysong.io/kubernetes-handbook/concepts/deployment.html)

### 管理对象

Objects

|  |  |
| --- | --- |
| 类别 | 名称 |
| 资源对象 | Pod、ReplicaSet、ReplicationController、Deployment、StatefulSet、DaemonSet、Job、CronJob、HorizontalPodAutoscaling、Node、Namespace、Service、Ingress、Label、CustomResourceDefinition |
| 存储对象 | Volume、PersistentVolume、Secret、ConfigMap |
| 策略对象 | SecurityContext、ResourceQuota、LimitRange |
| 身份对象 | ServiceAccount、Role、ClusterRole |

* Pod：一组关联的容器集合
* Namespace：对一组资源和对象的抽象集合
* Node：Pod真正运行的主机。K8s为了管理Pod，
* service：对一组提供相同功能的Pods的抽象
* DaemonSet 守护进程集。DaemonSet保证在特定或所有Node节点上都运行一个Pod实例，常用来部署一些集群的日志采集、监控或者其他系统管理应用。典型的应用包括:

日志收集，比如fluentd，logstash等

系统监控，比如Prometheus Node Exporter，collectd等

系统程序，比如kube-proxy, kube-dns, glusterd, ceph，ingress-controller等

* Ingress：Kubernetes中的负载均衡我们主要用到了以下两种机制：

Service：使用Service提供集群内部的负载均衡，Kube-proxy负责将service请求负载均衡到后端的Pod中

Ingress Controller：使用Ingress提供集群外部的负载均衡

#### Node

Node是kubernetes集群的工作节点，可以是物理机也可以是虚拟机

Node的状态

Node包括如下状态信息：

* Address

HostName：可以被kubelet中的--hostname-override参数替代。

ExternalIP：可以被集群外部路由到的IP地址。

InternalIP：集群内部使用的IP，集群外部无法访问。

* Condition

OutOfDisk：磁盘空间不足时为True

Ready：Node controller 40秒内没有收到node的状态报告为Unknown，健康为True，否则为False。

MemoryPressure：当node有内存压力时为True，否则为False。

DiskPressure：当node有磁盘压力时为True，否则为False。

* Capacity

CPU

内存

可运行的最大Pod个数

* Info：节点的一些版本信息，如OS、kubernetes、docker等

#### Namespace

在一个Kubernetes集群中可以使用namespace创建多个“虚拟集群”，这些namespace之间可以完全隔离，也可以通过某种方式，让一个namespace中的service可以访问到其他的namespace中的服务

kubectl get namespaces



对象创建

创建K8s的Object可用yaml文件配置。

示例

kubectl create -f nginx-deployment.yaml --record

1. apiVersion: apps/v1beta1
2. kind: Deployment
3. metadata:
4. name: nginx-deployment
5. spec:
6. replicas: 3
7. template:
8. metadata:
9. labels:
10. app: nginx
11. spec:
12. containers:
13. - name: nginx
14. image: nginx:1.7.9
15. ports:
16. - containerPort: 80

* apiVersion - 创建该对象所使用的 Kubernetes API 的版本
* kind - 对象的类型
* metadata - 帮助识别对象唯一性的数据，包括一个 name 字符串、UID 和可选的 namespace
* spec.replicas字段代表了受此RC管理的Pod，需要运行的副本数。
* template模块用于定义Pod，包括Pod的名字，Pod拥有的label以及Pod中运行的应用。

#### Pod概览

Pod封装应用容器(一个或多个)，存储、独立的网络IP，Pod代表着部署的一个单位：k8s中应用的一个实例，可能由一个或者多个容器组合在一起共享资源。

K8s管理的是Pod而不是直接管理容器，集群中Pod有如下两种使用方式：

* 一个Pod中运行一个容器(常用)。
* 在一个Pod中同时运行多个容器。

Pause容器

又叫Infra容器。主要为每个业务容器提供以下功能：

* 在pod中担任Linux命名空间共享的基础；
* 启用pid命名空间，开启init进程。
* 解析：pause容器将内部的80端口映射到宿主机的8880端口

### K8s架构

k8s架构



Master架构



Node架构



每个Node节点上至少需要运行container runtime（Docker）、kubelet和kube-proxy服务。

Kubelet

集群每个 Node 节点都启动一个Kubelet服务进程。处理 Scheduler下发到本节点任务/管理Pod生命周期。

这意味着它将处理 Pod 与 Container Runtime之间所有的转换逻辑，包括挂载卷、容器日志、垃圾回收以及其他重要事件。

可以把 Kubelet当成一种特殊的 Controller，它每隔20秒向kube-apiserver查询 Pod，过滤 NodeName 与自身所在节点匹配的 Pod 列表。

Kubelet启动容器过程

拉取容器的镜像

通过 CRI 创建容器

启动容器

### 控制器

Kubernetes中内建了很多controller（控制器），这些相当于一个状态机，用来控制Pod的具体状态和行为。

#### Deployment

Deployment 为 Pod 和 ReplicaSet 提供了一个声明式定义(declarative)方法，用来替代以前的ReplicationController 来方便的管理应用。典型的应用场景包括：

定义Deployment来创建Pod和ReplicaSet

* 滚动升级和回滚应用
* 扩容和缩容
* 暂停和继续Deployment

ReplicaSet

#### Horizontal Pod Autoscaling

[Reference](https://jimmysong.io/kubernetes-handbook/concepts/horizontal-pod-autoscaling.html)

应用的资源使用率通常都有高峰和低谷的时候，如何削峰填谷，提高集群的整体资源利用率，让service中的Pod个数自动调整呢？这就有赖于Horizontal Pod Autoscaling了，顾名思义，使Pod水平自动缩放。这个Object（跟Pod、Deployment一样都是API resource）也是最能体现kubernetes之于传统运维价值的地方，不再需要手动扩容了，终于实现自动化了，还可以自定义指标，没准未来还可以通过人工智能自动进化呢！

### 服务发现

Kubernetes中为了实现服务实例间的负载均衡和不同服务间的服务发现，创造了Serivce对象，同时又为从集群外部访问集群创建了Ingress对象。

#### Service

#### Ingress

### 存储

### 集群安全

用户

k8s 中有2种用户，一般用户和serviceAccount。

* 一般用户: 给集群外部用户使用，例如集群管理员通过 kubectl这个客户端来操作集群，使用的就是一般用户，这个用户通过 k8s 的 RBAC 权限系统获得相应权限。
* serviceAccount: 给集群内的资源使用，例如pod访问apiserver。每个pod都有一个 default serviceAccount。

Kublet的认证授权

* 匿名访问，调用 kubelet API 的能力应受到限制
* 启用 X509 客户端证书身份验证：
* 启用 API bearer token（包括 service account token）。

kubelet bootstrap

kubelet 使用低权限的 bootstrap token 跟 api server 建立连接后，要能够自动向 api server 申请自己的证书，并且 api server 要能够自动审批证书。

kubectl create clusterrolebinding kubelet-bootstrap --user=kubelet-bootstrap --clusterrole=system:node-bootstrapper

给bootstrap token代表的用户 system:bootstrap:abcdef 赋予 clusterole certificatesigningrequests.certificates.k8s.io/nodeclient 和 system:node-bootstrapper，让该用户可以访问 csr API 以及自动审批其创建的 csr

给新的 work node 代表的用户 system:node:test-node 赋予 clusterrole system:certificates.k8s.io:certificatesigningrequests:selfnodeclient，让它发送的证书 renew 的请求能被自动审批

kubectl 审批

# 通过 CSR 请求

kubectl certificate approve <name>

kubectl certificate deny <name>

命令扩展：

kubectl delete csr 节点名称 #删除单个节点的请求

kubectl delete csr --all #删除所有节点请求

kubectl delete nodes node名称 #删除加入的节点

kubectl delete nodes --all #删除所有节点

RoleBinding

如果想限制user用户行为，需要使用 RBAC创建角色绑定以将该用户的行为限制在某个或某几个 namespace 空间范围内，例如

赋予用户kubelet-bootstrap角色

kubectl create clusterrolebinding kubelet-bootstrap --clusterrole=system:node-bootstrapper --user=kubelet-bootstrap

kubectl get clusterrolebindings

否则报异常node exception：

server.go:271] failed to run Kubelet: cannot create certificate signing request: User "kubelet-bootstrap" cannot create certificatesigningrequests.certificates.k8s.io at the cluster scope. (post certificatesigningrequests.certificates.k8s.io)



roles

kubectl get clusterroles



RBAC-基于角色的访问权限控制

-authorization-mode=RBAC启动API Server

## 2.2 Docker

### Docker Remote API

远程管理docker,允许通过unix socket通信操作Docker daemon，也可以通过HTTP调用其Rest API，完成容器的查询，创建、销毁等等操作。

# 第三章 软件安装

### 3.1 Docker

#### 安装

[Reference](https://www.runoob.com/docker/centos-docker-install.html)

yum install docker-ce docker-ce-cli containerd.io

启动：systemctl start docker

vi /lib/systemd/system/docker.service

仓库地址配置/etc/yum.repos.d/docker-ce.repo

Uninstall

yum remove -y docker docker-latest docker-ce docker-common \

docker-selinux container-selinux \

docker-engine \

docker-ce-cli \

docker-client docker-client-latest \

docker-logrotate docker-latest-logrotate

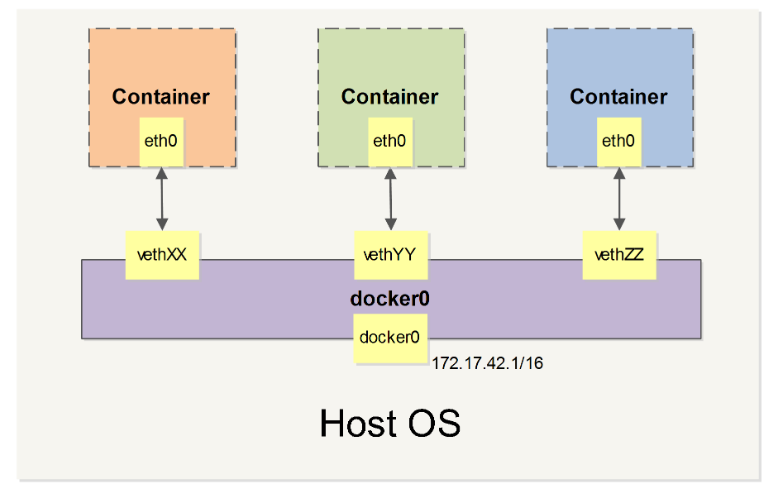
containerd.io

delete conf

/var/lib/docker

/etc/docker

#### 3.1.3 网络



### 6.3.2 k8s

安装

[Reference](https://kubernetes.io/docs/tasks/tools/install-kubectl/)

配置对多集群的访问:[Reference](https://kubernetes.io/zh/docs/tasks/access-application-cluster/configure-access-multiple-clusters/)

kubectl version

#### kubeadm

[Reference](https://blog.csdn.net/weixin_41806245/article/details/89381752)

install docker

yum -y install kubelet-1.14.0 kubeadm-1.14.0 kubectl-1.14.0

Init Master

kubeadm init

--apiserver-advertise-address 192.168.0.110 指定监听地址

--pod-network-cidr ：Specify range of IP addresses for the pod network.

--kubernetes-version=v1.14.0

示例

kubeadm init apiserver-advertise-address 192.168.0.110 --pod-network-cidr=10.244.0.0/16

Dashboad

kubectl delete -f dashboard.yaml

[mini Dashboad](http://192.168.0.8:8001/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/#/overview?namespace=default)

Node

mkdir -p $HOME/.kube

cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

chown $(id -u):$(id -g) $HOME/.kube/config

kubeadm join 192.168.0.110:6443 --token balro4.c1svv2f8rcz3bveb \

--discovery-token-ca-cert-hash sha256:26d141131d93adadcb535…

Token

kubectl -n kube-system describe $(kubectl -n kube-system get secret -n kube-system -o name| grep namespace) | grep token

1镜像问题

[Reference](https://juejin.im/post/5c24189d518825282222a480)

docker pull mirrorgooglecontainers/kube-proxy:v1.14.0

docker tag /mirrorgooglecontainers/kube-proxy:v1.14.0 k8s.gcr.io/kube-proxy:v1.14.0

docker rmi mirrorgooglecontainers/kube-proxy:v1.14.0

2Exception

[ERROR DirAvailable--etc-kubernetes-manifests]: /etc/kubernetes/manifests is not empty

[ERROR FileAvailable--etc-kubernetes-kubelet.conf]: /etc/kubernetes/kubelet.conf already exists

kubeadm reset

#### Minikube

[Reference](https://yq.aliyun.com/articles/221687)

安装

**1.下载minikube**

curl -Lo minikube http://kubernetes.oss-cn-hangzhou.aliyuncs.com/minikube/releases/v1.4.0/minikube-linux-amd64 && chmod +x minikube && sudo mv minikube /usr/local/bin/

启动

minikube start --vm-driver=none

查看状态：minikube status

minikube dashboard

kubectl proxy --port=8001 --address='192.168.0.5' --accept-hosts='^.\*' &

更新

如需更新minikube，需要更新 minikube 安装包

minikube delete 删除现有虚机，删除 ~/.minikube 目录缓存的文件

vm driver

~/.minikube/machines/minikube/config.json

#### 3.2.2 kubectl

kubectl version：Print the client and server version information

Client Version: version.Info{Major:"1", Minor:"6", GitVersion:"v1.6.0", GoVersion:"go1.7.5", Platform:"linux/amd64"}

Server Version: version.Info{Major:"1", Minor:"5", GitVersion:"v1.5.2", GitTreeState:"clean", GoVersion:"go1.7.4", Compiler:"gc", }

##### kubeconfig

kubeconfig文件记录k8s集群的各种信息，对集群构建非常重要。

* kubectl从~/.kube/config，即kubectl的kubeconfig文件中获取访问kube-apiserver的地址，证书和用户名等信息
* kubelet/kube-proxy等在Node上的程序进程同样通过bootstrap.kubeconfig和kube-proxy.kubeconfig上提供的认证与授权信息与Master进行通讯

查询集群地址和凭证

kubectl config view

1. apiVersion: v1
2. clusters:
3. - cluster:
4. certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSU....
5. server: https://192.168.0.110:6443
6. name: kubernetes
7. contexts:
8. - context:
9. cluster: kubernetes
10. user: kubelet-bootstrap
11. name: default
12. current-context: default
13. kind: Config
14. preferences: {}
15. users:
16. - name: kubelet-bootstrap
17. user:
18. token: 5123398c0116eb8617398e4c38c3be

certificate-authority-data 是 ca.pem 的base64加密的内容

client-key-data:

kubectl config get-contexts

##### Volume存储数据卷

exception

Orphaned pod "cdb27b68-9e8d-4708-8179-b5579dab861f" found, but volume paths are still present on disk.

rm -rf /var/lib/kubelet/pods/\*

#### 手动安装

[Reference](https://o-my-chenjian.com/2017/04/26/Deploy-Master-Of-K8s/)

##### flannel

master和node上都要安装Flannel网络

配置

vi /usr/lib/systemd/system/flanneld.service

vi /etc/sysconfig/flanneld

etcdctl --endpoints=${ETCD\_ENDPOINTS} put /kubernetes/network/config '{"Network":"'${CLUSTER\_CIDR}'", "SubnetLen": 24, "Backend": {"Type": "vxlan"}}'

etcdctl --endpoints=${ETCD\_ENDPOINTS} get /kubernetes/network/config

--cacert=/etc/kubernetes/ssl/ca.pem --key=/etc/etcd/ssl/etcd-key.pem --cert=/etc/etcd/ssl/etcd.pem

通过http向etcd增加key

curl -X PUT http://192.168.0.110:2379/v2/keys/kubernetes/network/config -d value='{"Network":"172.30.0.0/16", "SubnetLen": 24, "Backend": {"Type": "vxlan"}}'

##### master节点

安装kubernetes： yum -y install kubernetes

kube-apiserver

vi /usr/lib/systemd/system/kube-apiserver.service

vi /etc/kubernetes/apiserver

systemctl daemon-reload

systemctl restart kube-apiserver.service

kube-controller-manager

vi /usr/lib/systemd/system/kube-controller-manager.service

systemctl daemon-reload

systemctl restart kube-controller-manager.service

查看节点

curl http://192.168.0.110:8080/api/v1/nodes

kube-scheduler

vi /usr/lib/systemd/system/kube-scheduler.service

systemctl enable kube-scheduler

##### Node节点

1.安装kubernetes

yum -y install kubernetes&vi /etc/kubernetes/config

* KUBE\_LOGTOSTDERR="--logtostderr=true"
* KUBE\_LOG\_LEVEL="--v=0"
* KUBE\_ALLOW\_PRIV="--allow-privileged=false"
* KUBE\_MASTER="--master=http://k8s-master:8080"

kubelet

vi /usr/lib/systemd/system/kubelet.service

vi /etc/kubernetes/kubelet

* KUBELET\_ADDRESS="--address=0.0.0.0"
* KUBELET\_HOSTNAME="--hostname-override=k8s-node-1"
* KUBELET\_API\_SERVER="--api-servers=http://k8s-master:8080"
* KUBELET\_POD\_INFRA\_CONTAINER="--pod-infra-container-image=registry.access.redhat.com/rhel7/pod-infrastructure:latest" #确认此镜像存在，否则pod无法启动
* KUBELET\_ARGS=""

/etc/kubernetes/kubelet.conf

–client-ca-file

3.启动服务

systemctl daemon-reload

systemctl restart kubelet

kube-proxy

/usr/lib/systemd/system/kube-proxy.service

Vi

##### etcd

etcdctl version

ETCDCTL\_API=3

配置

vi /usr/lib/systemd/system/etcd.service

vi /etc/etcd/etcd.conf

#[Member]

ETCD\_DATA\_DIR：etcd数据保存目录

ETCD\_LISTEN\_CLIENT\_URLS：供外部**客户端**使用的url

ETCD\_ADVERTISE\_CLIENT\_URLS：暴露给外部**客户端**使用的url

ETCD\_NAME：etcd实例名称

#[Clustering]

ETCD\_LISTEN\_PEER\_URLS：集群内部通信使用的URL

ETCD\_INITIAL\_ADVERTISE\_PEER\_URLS：广播给集群内其他成员访问的URL

ETCD\_INITIAL\_CLUSTER：初始集群成员列表

ETCD\_INITIAL\_CLUSTER\_TOKEN：集群的名称

ETCD\_INITIAL\_CLUSTER\_STATE：初始集群状态，new为新建集群

rm -rf /var/lib/etcd/default.etcd

systemctl daemon-reload

systemctl restart etcd

etcdctl cluster-health

配置TSL

etcdctl member list --endpoints=https://192.168.0.111:2379 --cacert=/etc/kubernetes/ssl/ca.pem --key=/etc/etcd/ssl/etcd-key.pem --cert=/etc/etcd/ssl/etcd.pem

CMD

* -w, --write-out="simple" 输出格式(fields, json, protobuf, simple, table)
* --endpoints=[127.0.0.1:2379] ：gRPC endpoints

Restful api

curl http://192.168.0.110:2379/v2/keys/

查看集群

etcdctl endpoint status

etcdctl [--write-out=table] endpoint health

etcdctl member list

etcdctl endpoint health --endpoints=https://127.0.0.1:2379 --cacert=/etc/kubernetes/ssl/ca.pem --key=/etc/kubernetes/ssl/etcd-key.pem --cert=/etc/kubernetes/ssl/etcd.pem

示例：

etcdctl --endpoints=http://192.168.0.110:2379 member list -w table

集群成员

跟集群成员相关的命令如下：

member add Adds a member into the cluster

member remove Removes a member from the cluster

member update Updates a member in the cluster

member list Lists all members in the cluster

示例

etcdctl member list -w table



CRUD

* 增：etcdctl --endpoints=$ENDPOINTS put foo "Hello World!"
* 查：etcdctl [--endpoints=$ENDPOINTS] [--write-out="json"] get foo
* 删：etcdctl --endpoints=$ENDPOINTS del key

Failed to connect to apiserver: the server has asked for the client to provide credentials

Failed to connect to apiserver: Forbidden: "/healthz?timeout=1s

#### 集群安全设置

[Reference](https://o-my-chenjian.com/2017/04/25/Security-Settings-Of-K8s/)

#### grafana

[监控k8s](https://blog.csdn.net/lindao99/article/details/79977794)

# todo

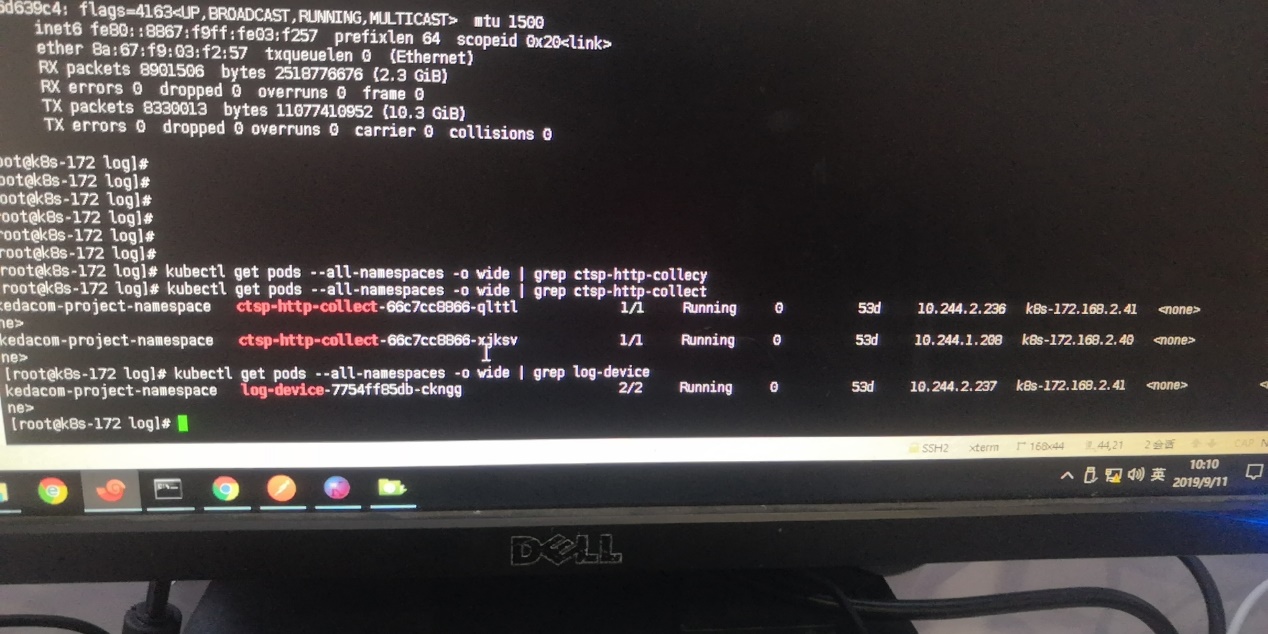
市局二类网服务器，查看设备日志的服务

Kubectl get pods --all-namespaces -o wide |grep ctsp-http-collect

kubectl –n kedacom-project-namespace logs ctsp-http-collect-66c7cc8866-qlttl > ctsp-http-collect-01.log

kubectl -n kedacom-project-namespace logs ctsp-http-collect-66c7cc8866-xjksv > ctsp-http-collect-02.log

kubectl get pods --all-namespaces |grep log-device



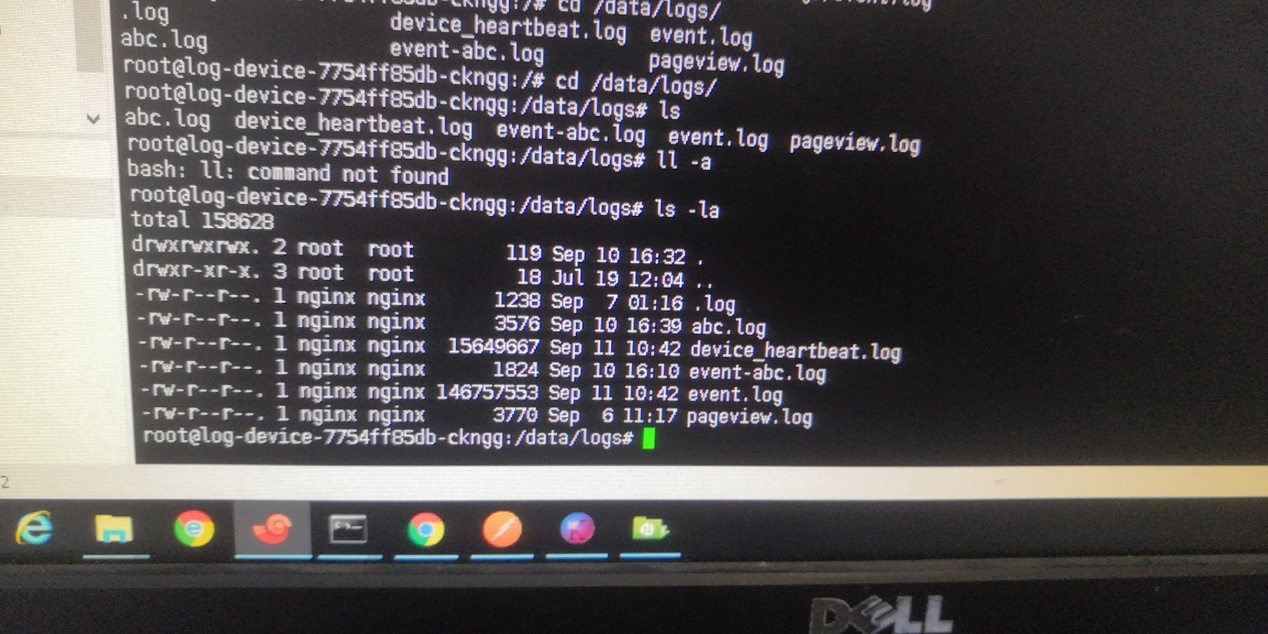
kubectl -n dolphin exec -it log-device-7885d65c9b-9l2w8（此处为下图的名字） bash

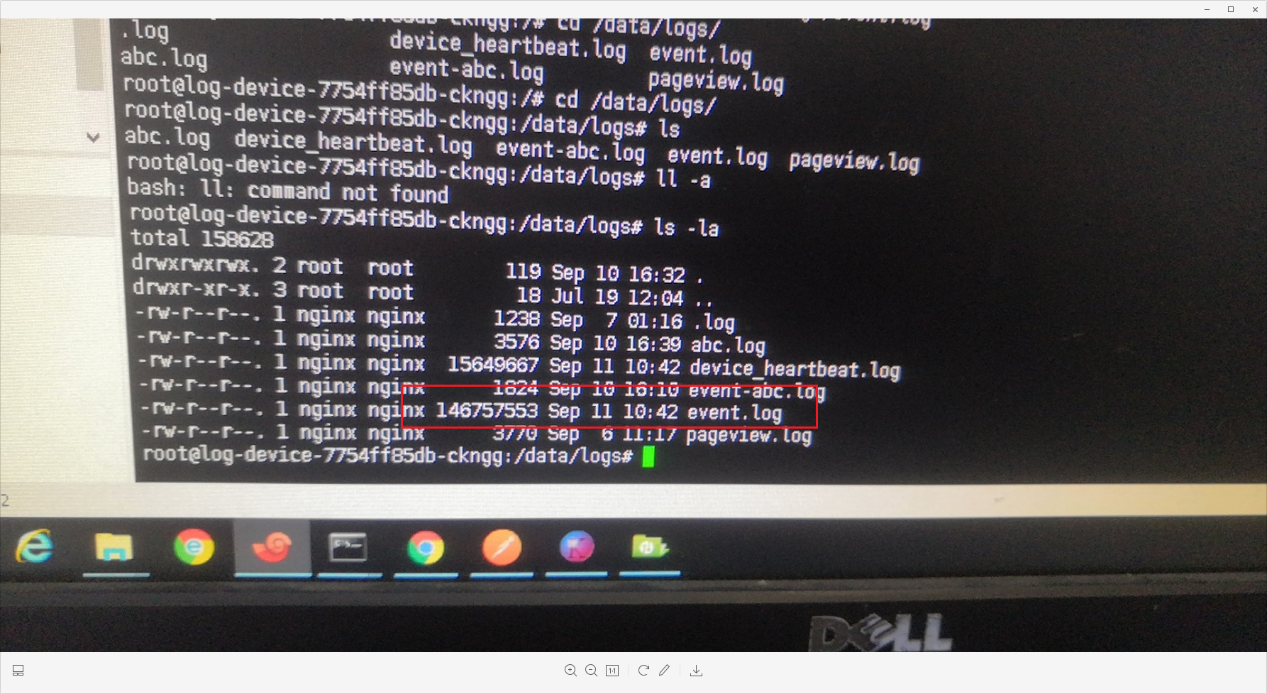
C:\Users\keda\AppData\Local\Temp\WeChat Files\10ce15cb149112aadf1e09a6d0c2bdb.png

进入容器之后

ls /data/logs/event.log -la

看下这个文件





kubectl -n dolphin exec -it log-device-7885d65c9b-9l2w8 bash -c flume-collect

用这个进flume服务 看下