**支持微服务架构的工业应用开发框架**

**用户手册**

****

南京创联智软信息科技有限公司

2020年06月

目录

[1 引言 1](#_Toc12059)

[1.1 编写目的 1](#_Toc1948)

[1.2 适用范围 1](#_Toc1594)

[2 系统概述 1](#_Toc10479)

[3 系统详细介绍 2](#_Toc88)

[3.1 系统架构 2](#_Toc10738)

[3.2 第三方数据的接入 3](#_Toc4692)

[3.3 业务数据配置 4](#_Toc2957)

[4 三员功能 4](#_Toc8944)

[4.1 系统管理员（sysadmin） 4](#_Toc17678)

[4.1.1 用户管理（不含权限） 4](#_Toc4496)

[4.1.2 字典管理 7](#_Toc4203)

[4.1.3 定时任务管理 8](#_Toc12263)

[4.1.4 领域管理 8](#_Toc25017)

[4.1.5 流程管理 8](#_Toc11812)

[4.2 安全保密员（secadmin） 9](#_Toc11056)

[4.2.1 用户管理（权限分配） 9](#_Toc13654)

[4.2.2 新增角色 9](#_Toc23154)

[4.2.3 设置密码安全策略 10](#_Toc2595)

[4.3 安全审计员（auditadmin） 10](#_Toc8436)

[5 关键技术解决方案 10](#_Toc8710)

[5.1 分布式事务解决方案 10](#_Toc26803)

[5.1.1 分布式事务概念 10](#_Toc10159)

[5.1.2 分布式事务解决方案之2PC 11](#_Toc8100)

[5.1.3 Seata实现2PC事务 11](#_Toc3901)

[5.1.4 基于Seata分布式事务示例 13](#_Toc6006)

[5.1.5 分布式事务之TCC 15](#_Toc8554)

[5.1.6 TCC事务解决方案 16](#_Toc25767)

[5.1.7 Hmily实现TCC事务 18](#_Toc23768)

[5.1.8 分布式事务解决方案之可靠消息最终一致性 19](#_Toc16120)

[6 常用组件安装 22](#_Toc18221)

[6.1 消息中间件Rabbitmq 22](#_Toc14017)

[6.2 缓存型数据库Redis 22](#_Toc21402)

[6.3 Devops 22](#_Toc28093)

[6.3.1 代码库建设 22](#_Toc29437)

[6.3.2 持续集成（CI） 23](#_Toc14136)

[7 Kubernetes 24](#_Toc16836)

[7.1 基本对象概述 24](#_Toc25300)

[7.2 系统架构及组件功能 25](#_Toc10810)

[7.3 集群部署 25](#_Toc20022)

[7.3.1 环境规划 25](#_Toc17894)

[7.3.2 集群安装准备 26](#_Toc32092)

[7.3.3 Docker安装 29](#_Toc24346)

[7.3.4 基于Kubeadm安装K8S集群 31](#_Toc13935)

[7.3.5 安装Flannel网络插件 32](#_Toc15348)

[7.3.6 制品库安装 33](#_Toc17735)

[7.3.7 安装Kuboard 34](#_Toc26198)

[7.4 负载均衡 35](#_Toc27651)

[7.4.1 Nginx安装步骤 35](#_Toc156)

[7.4.2 Nginx使用步骤 36](#_Toc29022)

[7.4.3 Nginx实现负载均衡 37](#_Toc29987)

[7.4.4 Nginx配置高可用集群 37](#_Toc678)

# 引言

## 编写目的

根据《支持微服务架构的工业应用开发框架技术要求书》，此文档的主要目的辅助用户对支持微服务架构的工业应用开发框架的使用，明确操作步骤。

## 适用范围

本文档针对的读者主要是支持微服务架构的工业应用开发框架的使用人员，包括系统的维护人员也可以参考此文档对系统进行维护。

# 系统概述

随着业务场景越来越复杂，传统的单体应用架构的代码库不断膨胀，复杂性高、技术债务、部署频率低、可靠性差、扩展能力受限、阻碍技术创新等缺点愈发显著。微服务架构由于具备组件化、解耦化的特点，服务可独立部署、独立扩展，同时具备可靠性强、技术栈不受限等优点，使得微服务架构成为当前信息系统架构的发展趋势。随着工业互联网的发展，工业应用呈现出平台化、组件化的发展趋势，这恰好与微服务架构的思想是一致的。

微服务把一个大型的单体应用或服务拆分为多个支持微服务，它可扩展单个组件而不是整个应用程序堆栈，从而满足服务等级协议，提高了系统横向扩展、快速更新升级的能力，**微服务架构具有如下的优势**：

（1）技术的异构性：在一个由多个服务相互协作的系统中，可以在不同的服务中使用最适合该服务的技术。

（2）弹性：在单块系统中，如果服务不可用，那么所有的功能都会不可用，然而微服务系统本身就能够很好地处理服务不可用和功能降级问题。

（3）扩展：庞大的单块服务只能作为一个整体进行扩展。即使系统中只有一小部分存在性能问题，也需要对整个服务进行扩展。如果使用较小的多个服务，则可以只对需要扩展的服务进行扩展。

（4）与组织结构相匹配：微服务架构可以很好地将架构与组织结构相匹配，避免出现过大的代码库，从而获得理想的团队大小及生产力。

（5）可组合性：在微服务架构中，系统会开放很多接口供外部使用。当情况发生改变时，可以使用不同的方式构建应用，而整体化应用程序只能提供一个非常粗粒度的接口供外部使用。

（6）对可替代性的优化：使用微服务架构的团队可以在需要时轻易地重写服务，或者删除不再使用的服务。

任何事物都有它的双面性，都有它不利的一面，微服务也不例外，同单体架构相比，**微服务架构存在以下的缺点：**

（1）服务管理困难：单个服务的开发和维护相对来说是很容易的，但从整个系统上看，这是一件很麻烦的事情，因为系统从单体变成了分布式，多个服务分布在不同的服务器中，需要完善的服务监控和管理的能力。

（2）来自分区数据库带来的实现问题：每一个服务都有自己的数据库，这样才能

达到真正系统微服务化的目的。由此会带来一个最为突出的问题就是分布式事务，如何保证系统数据最终的一致性是使用微服务需要格外注意的问题。

1. 服务间调用的成本更高：由于服务都是分布式部署，服务之间的调用相比传统的本地方法调用，需要更大的成本，调用过程中还会遇到安全、网络抖动等外在的问题。

# 系统详细介绍

## 系统架构

为了支持部门业务的可持续拓展以及部门业务的不断迭代更新，开发框架的研制工作采用微服务的架构完成系统的设计。如图1所示为系统的架构设计图：



**图 1 支持微服务架构的工业应用开发框架系统架构图**

具体要求如下：

* 系统需要内置三员管理用户
* 实现统一鉴权
* 实现外部用户接入（Xbase系统）
* 系统应当具备完善的权限管理
* 系统应当具备第三方数据接入功能
* 具备服务监控功能
* 系统应当具备高度的可拓展性
* 系统应当具备完善的日志管理模块，能够准确的记录用户的操作内容

## 第三方数据的接入

以Alm数据管理系统为例，Alm同步管理主要负责完成Alm的领域建立、项目建立和alm配置管理的工作，点击配置管理，输入如下的信息，如图2所示：



**图2 alm数据接入配置**

点击保存按钮，完成了alm的配置工作。

## 业务数据配置

在字典管理中可以灵活的配置测试级别、测试类型、系统配置项名称、文档类型、批注类型的参数值，如图3所示：



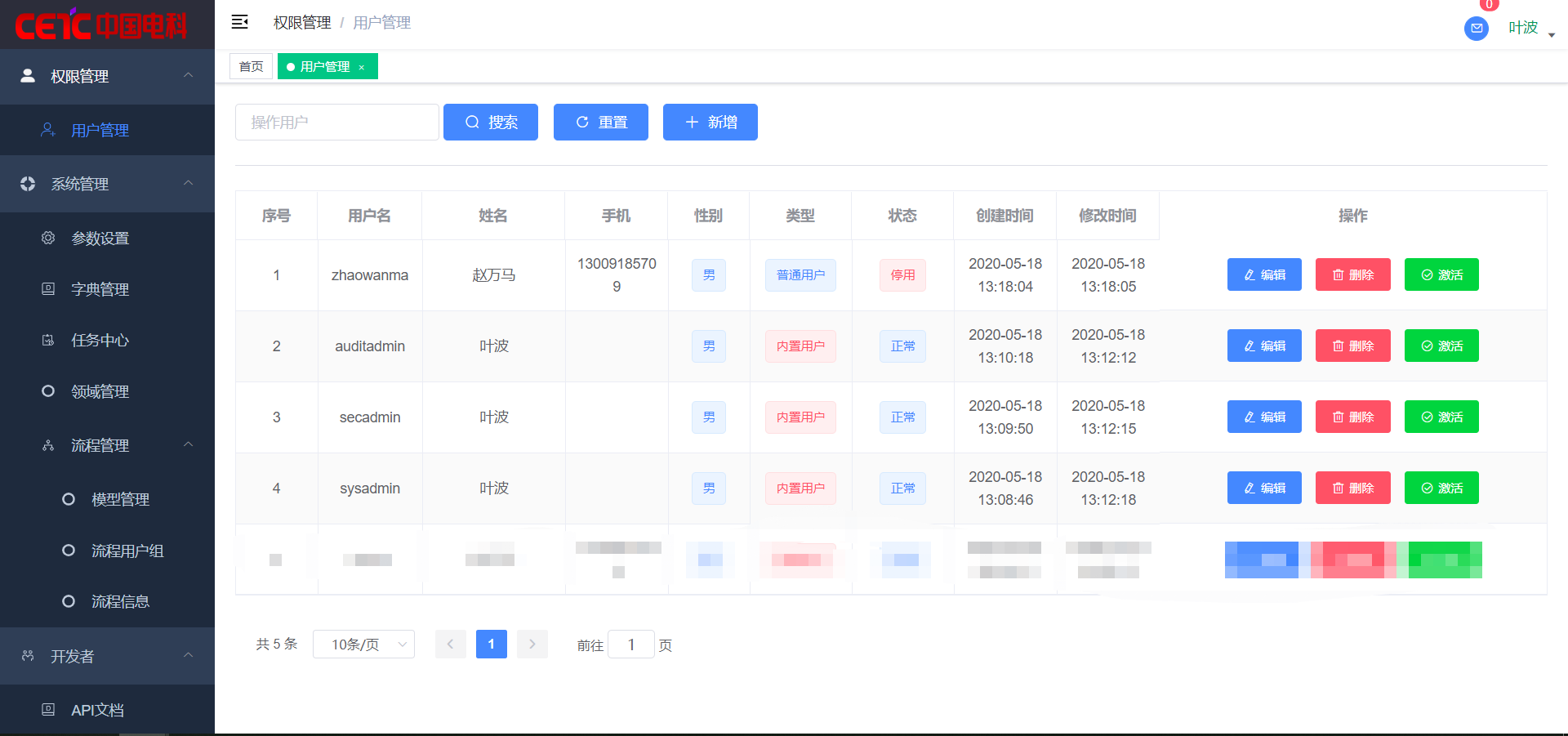
**图3 业务数据配置界面**

# 三员功能

## 系统管理员（sysadmin）

### **用户管理（不含权限）**

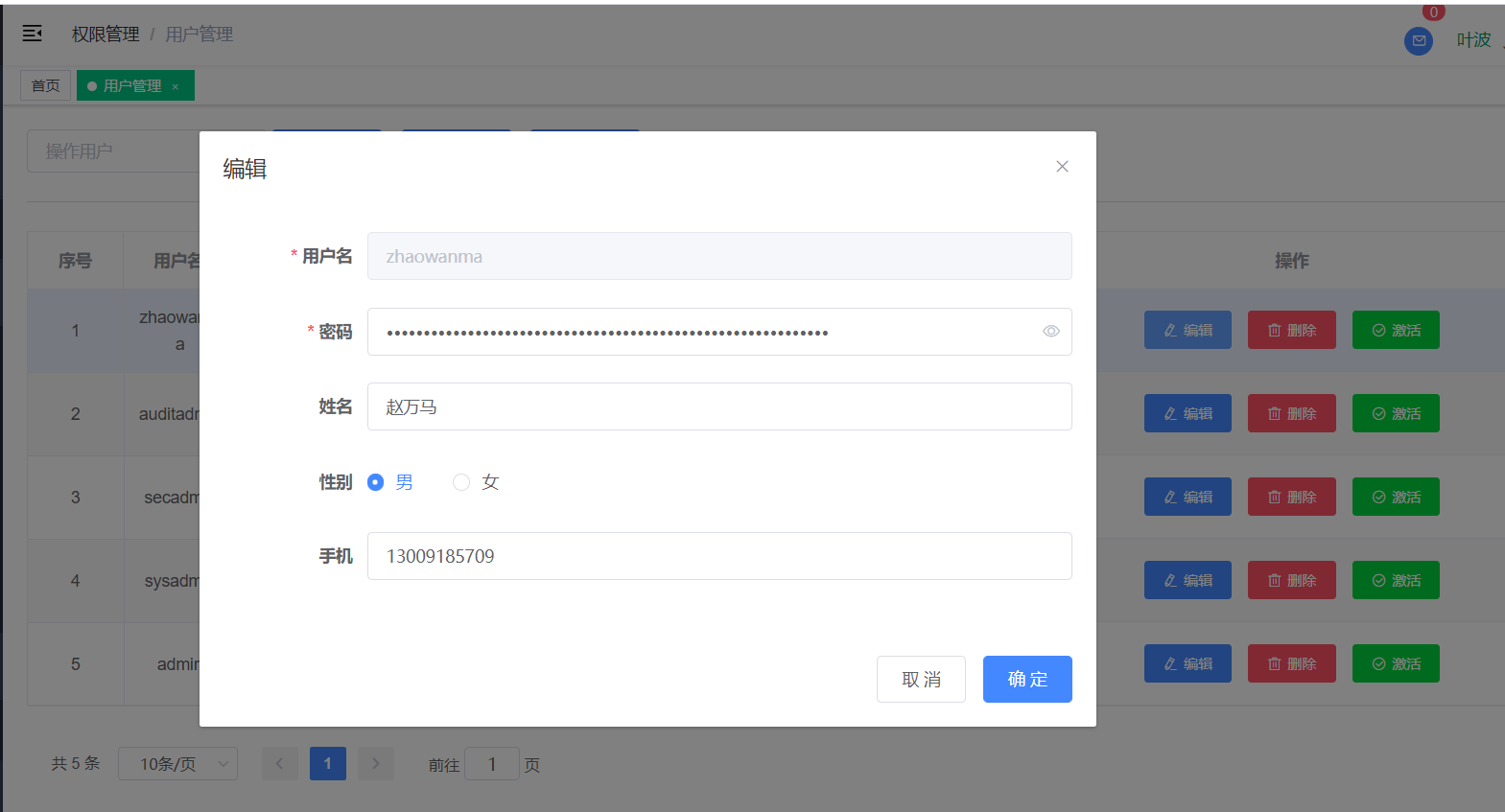
sysadmin系统管理员账户，主要为系统日常运行提供维护、配置管理。系统管理员登录后，默认具有系统管理模块的访问权限，并可维护组织结构、管理数据导入、控制工作流程以及启动或终止定时任务等。



**图4 系统管理员登录界面**

如图4所示为系统管理员的登录页面，系统管理员的登录后展示的页面如图所示，系统管理员具备用户管理、字典管理、任务中心，领域管理，流程管理等模块。

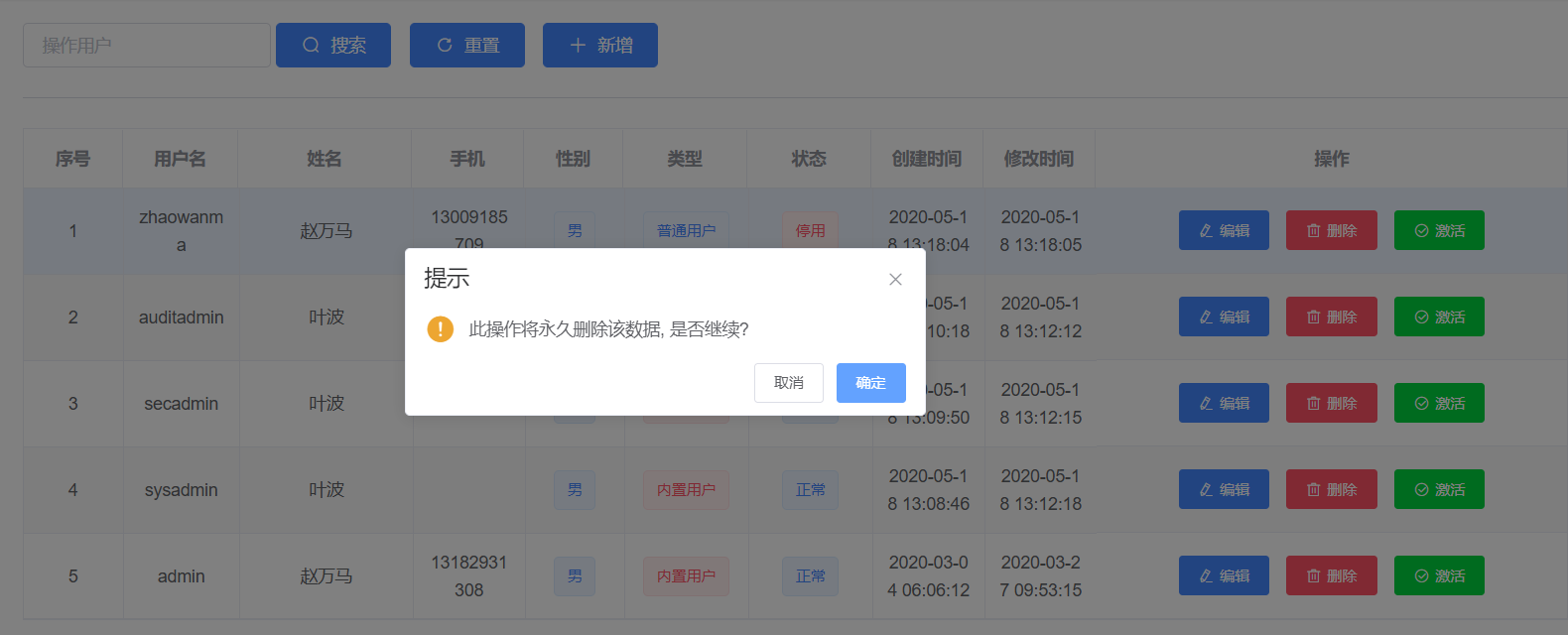
如图5所示，点击按钮，如图5所示分别为普通用户和内置用户的结果，其中内置用户会提示不可编辑，普通用户可以编辑。

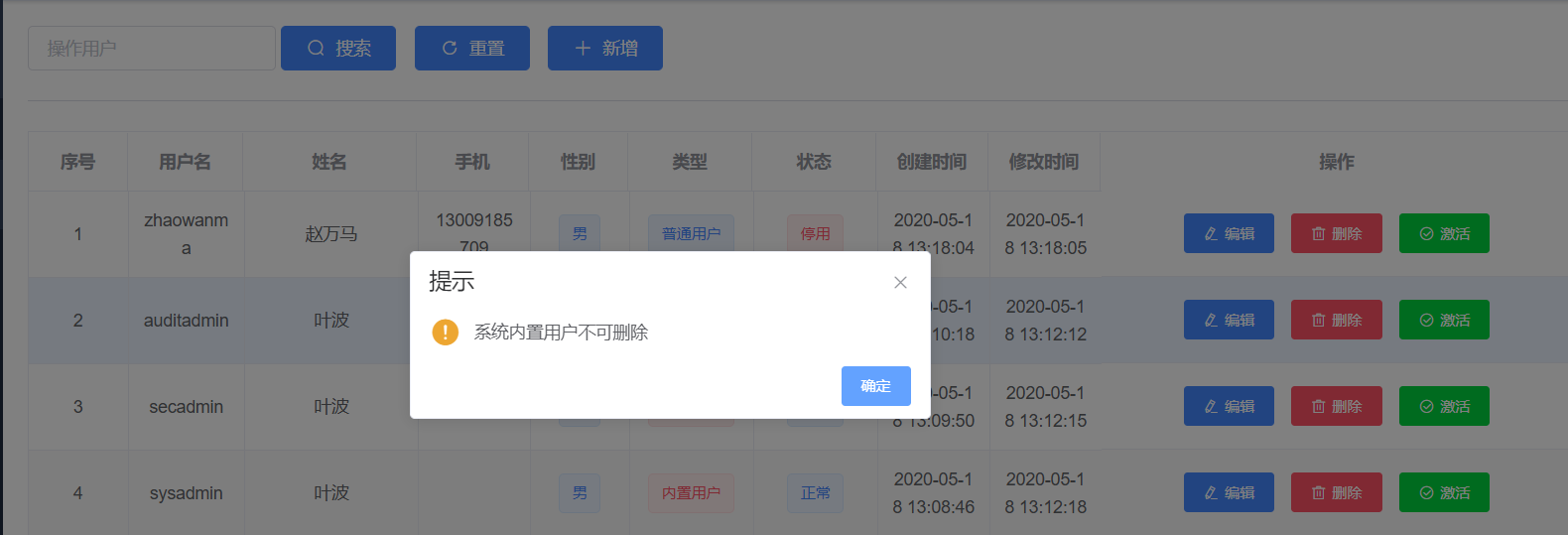




**图5 用户编辑功能**

如图6所示，点击 按钮，如图所示分别为普通用户和内置用户的结果，其中内置用户会提示不可删除，普通用户可以删除。

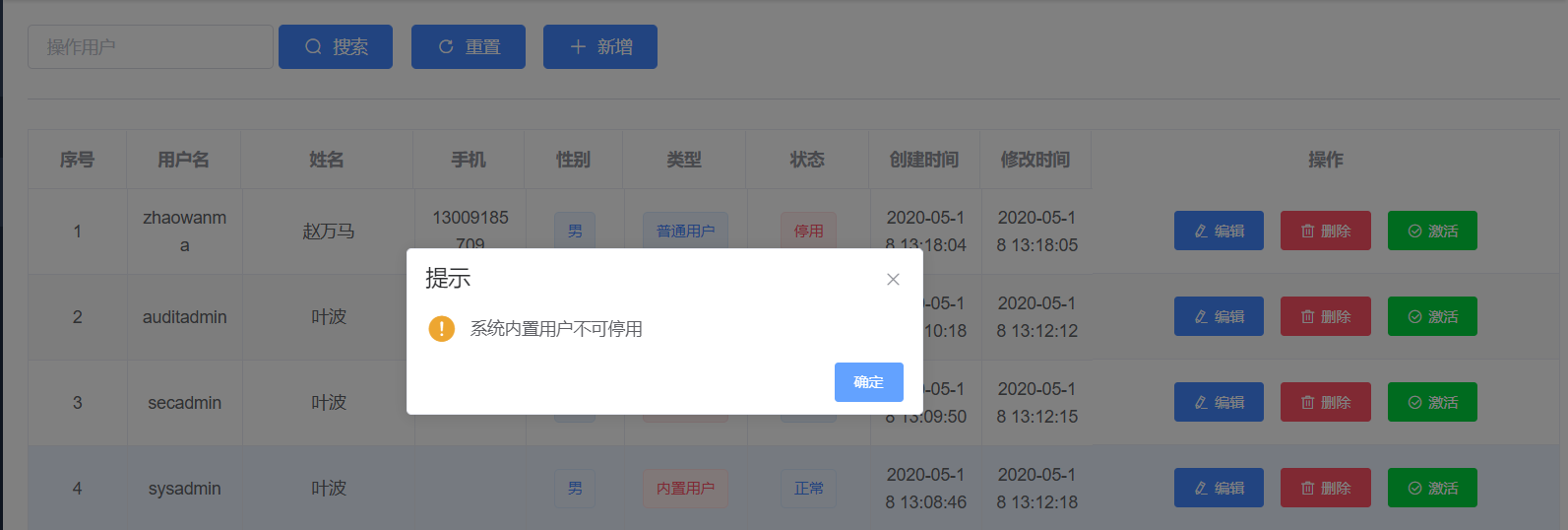




**图6 系统管理员删除用户**

如图7所示，点击按钮，如图所示分别为普通用户和内置用户的结果，其中内置用户会提示不可停用，普通用户可以停用。

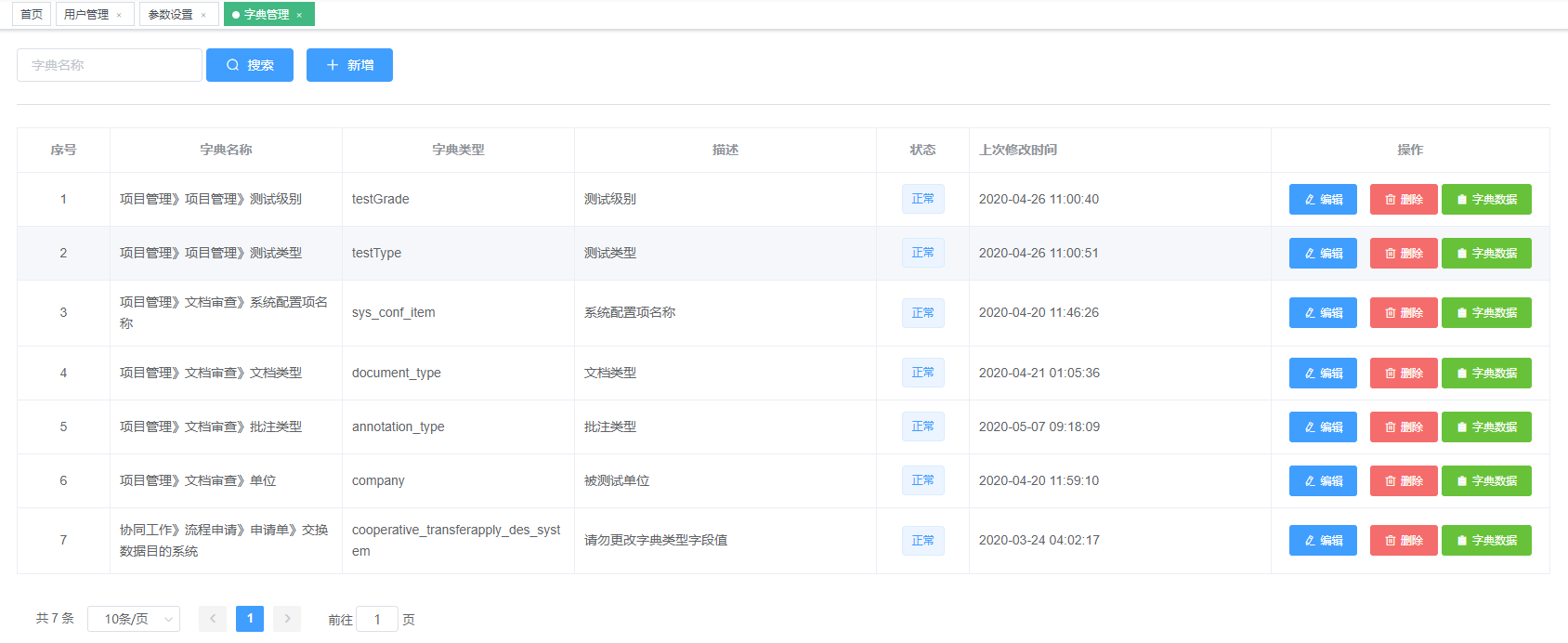




**图7 系统管理员停用用户**

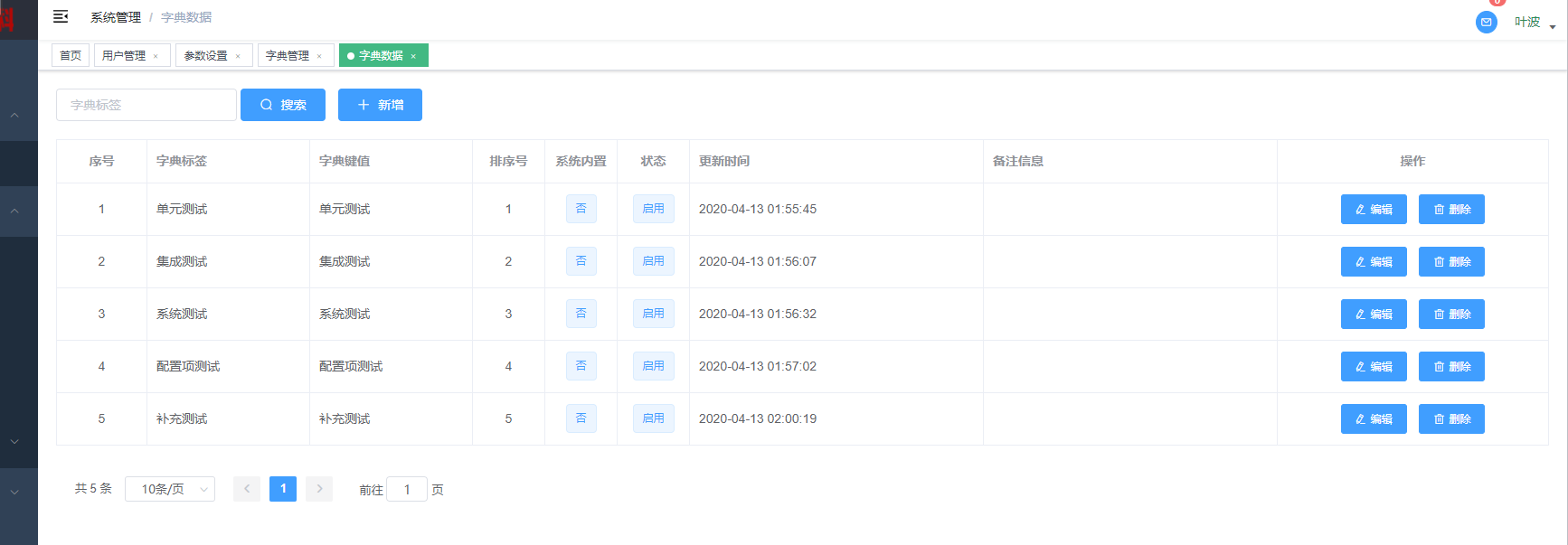
### **字典管理**

系统管理员具有维护系统字典的权限，如图8所示为系统的字典管理视图。



**图8 系统的字典管理视图**

以维护测试级别为例，系统管理员点击字典数据按钮按钮，弹出如下界面。



**图9 维护字典数据视图**

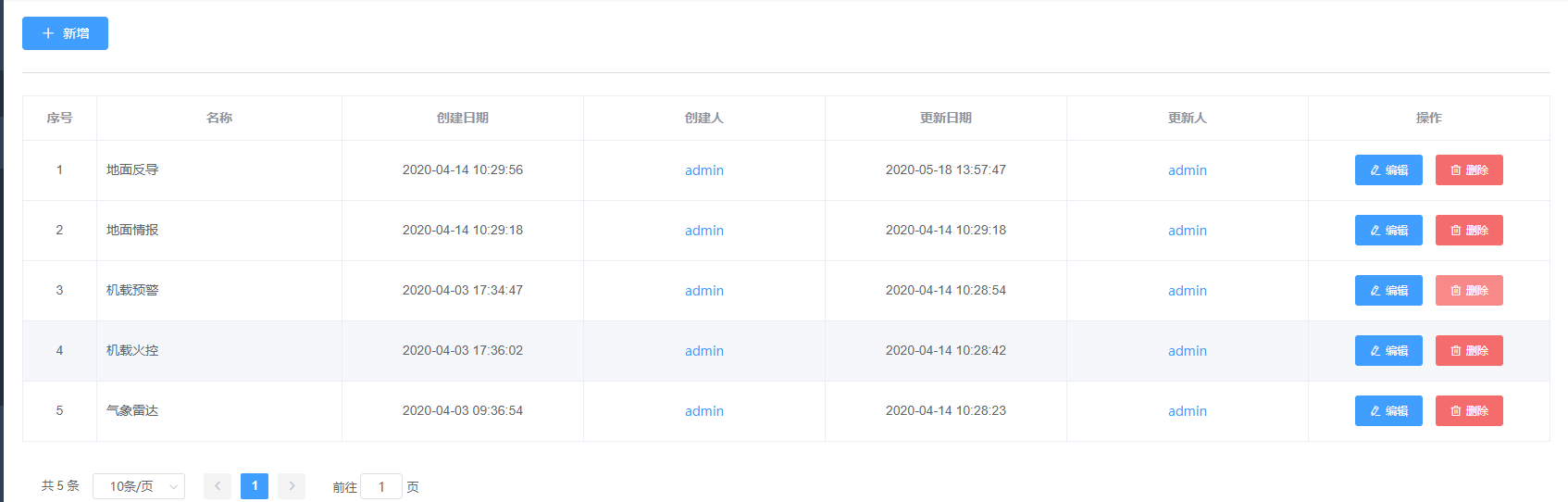
系统管理员可以新增或者修改测试级别，修改保存后会和项目管理中的新建项目功能进行联动。

### **定时任务管理**

系统管理员具备启动、停用定时任务的权限，目前系统不存在需要维护的定时任务。

### **领域管理**

系统管理员具备维护项目领域的权限。



**图10 领域维护视图**

### **流程管理**

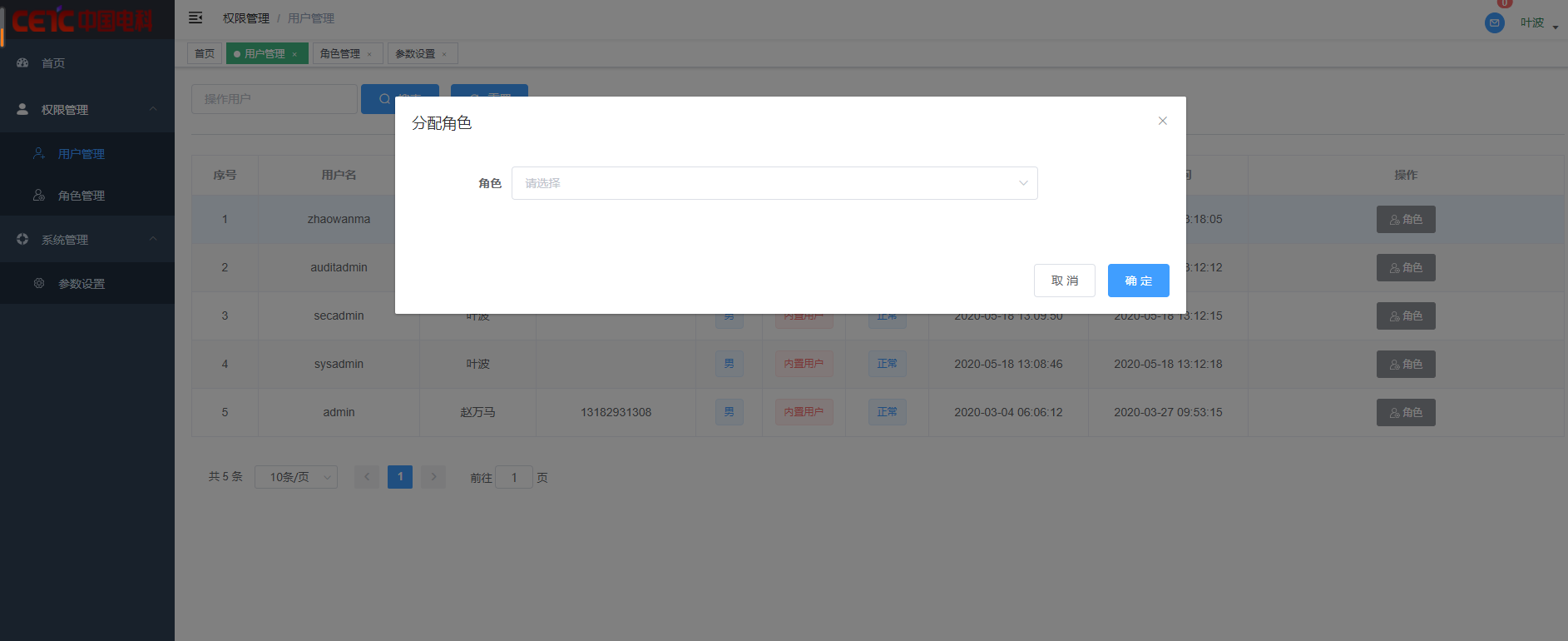
系统管理员具备维护系统流程的权限，包括流程模型的维护，流程用户组的维护以及流程信息的维护。

## 安全保密员（secadmin）

secadmin系统安全保密员账户，安全保密员可以分配权限组，解锁系统管理员和审计管理员，设定密码安全策略，安全保密员登录后，默认具有用户管理、角色管理和参数设置的功能。

### **用户管理（权限分配）**

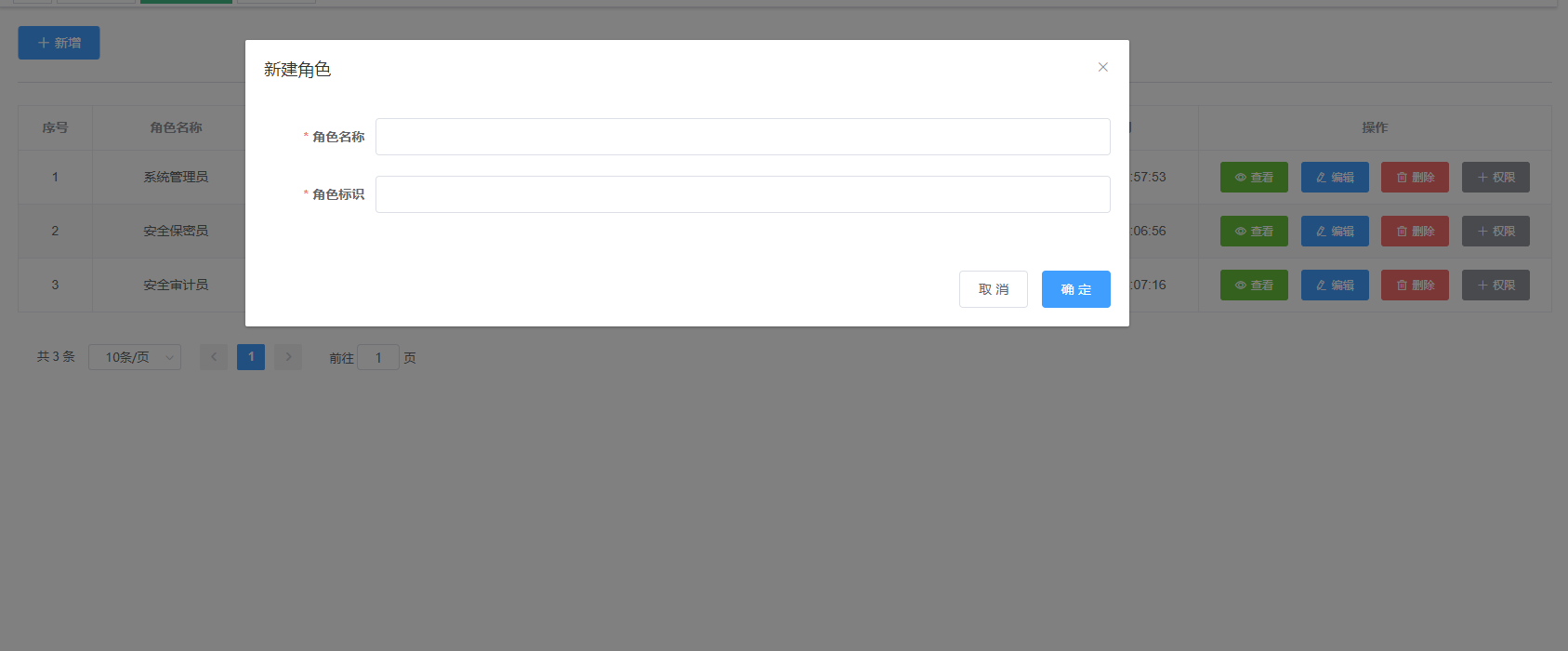
安全保密员登录后，具有为用户分配角色的权限，如图11所示。



**图11 分配角色**

### **新增角色**

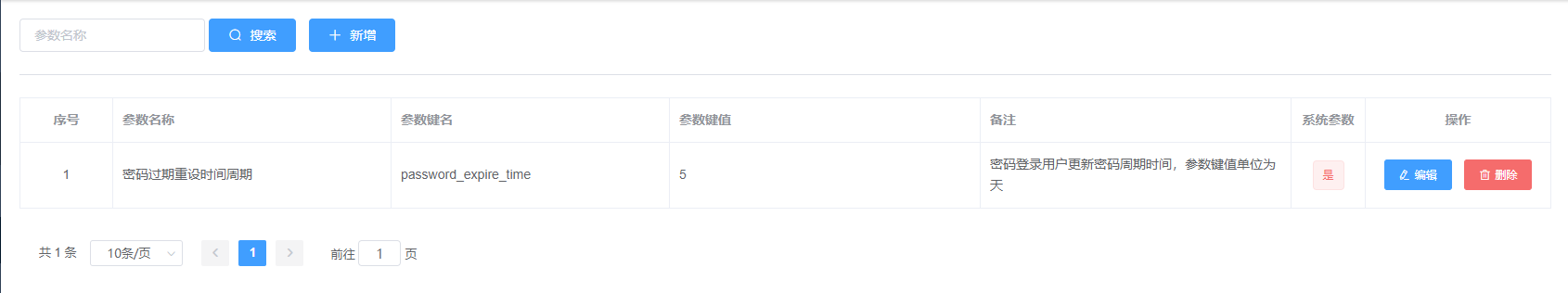
安全保密员登录后，具有为新增用户角色的权限，如图12所示。



**图12 新增角色**

### **设置密码安全策略**

安全保密员登录后，具有设定密码安全策略的权限，如图13所示。



**图13 修改密码安全策略**

## 安全审计员（auditadmin）

auditadmin系统安全审计员，通过日志审计系统管理员的操作行为，如图14所示为安全审计员登录展示内容。



**图14 日志管理系统**

# 关键技术解决方案

## 分布式事务解决方案

### **分布式事务概念**

分布式系统会把应用拆分为可独立部署的多个服务，因此需要服务与服务之间远程协作才能完成事务操作，这种分布式环境下由不同的服务之间通过网络远程协作完成事务称之为分布式事务，例如创建订单减库存事务，银行跨行转账事务等都是分布式事务。

分布式事务产生的场景：

* 多服务访问多数据库；
* 单体系统访问多个数据库实例；
* 多服务访问同一个数据库实例。

目前分布式事务常见的解决方案有2PC、TCC、可靠消息最终一致性、最大努力通知这几种。

### **分布式事务解决方案之2PC**

两阶段提交又称2PC（two-phase commit protocol），是将整个事务流程分为两个阶段，准备阶段（prepare phase）、提交阶段(commit phase)。准备阶段和提交阶段都是由事务管理器（协调者）发起的，协调的对象是资源管理器（参与者）。二阶段提交协议的概念来自 X/Open 组织提出的分布式事务的规范 XA 协议，协议主要定义了（全局）事务管理器和（局部）资源管理器之间的接口。XA 接口是双向的系统接口，在事务管理器以及一个或多个资源管理器之间形成通信桥梁。在计算机中部分关系型数据库如Oracle、MySQL支持两阶段提交协议。

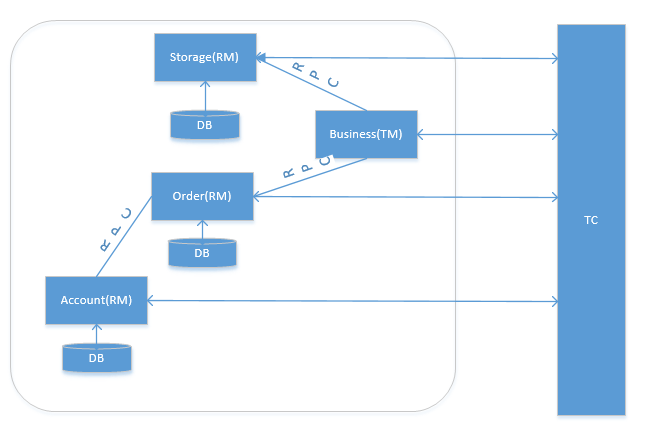
两阶段如下：

准备阶段：协调者向参与者发起指令，参与者评估自己的状态，如果参与者评估指令可以完成，参与者会写 redo 和 undo 日志，然后锁定资源，执行操作，但是并不提交。（Undo日志是记录修改前的数据，用于数据回滚，Redo日志是记录修改后的数据，用于提交事务后写入的数据文件）。

提交阶段：如果每个参与者明确返回准备成功，也就是预留资源和执行操作成功，协调者向参与者发起提交指令，参与者提交资源变更的事务，释放锁定的资源；如果任何一个参与者明确返回准备失败，也就是预留资源或者执行操作失败，协调者向参与者发起中止指令，参与者取消已经变更的事务，执行 undo 日志，释放锁定的资源。

### **Seata实现2PC事务**

今年初，阿里发布了开源分布式事务框架Fescar，后来跟蚂蚁TCC方案整合后改名为Seata，之所以采用阿里的Seata框架作为系统的分布式事务支撑技术，首先，阿里的开源技术往往是内部久经考验的，稳定性较好，其次，阿里的圈内影响力较高，Seata一经发布便形成了非常完善的技术生态，最后，目前由大厂发布的标准分布式事务框架只有阿里，以商品支付为例，整个分布式事务控制的原理如图11所示。



如上图，对于每一个服务而言都有两种角色（TM，RM）,在全局事务的过程中，它们会与TC进行通信协助完成整个事务，可以简单介绍下每个角色的作用。

* TC：Transaction Coordinator，事务协调器：监视每个全局事务的状态，负责全局

事务的提交和回滚。

* TM：Transaction Manager, 事务管理者：向TC发起，提交，回滚全局事务的请求。
* RM： Resource Manager, 资源管理器：服务向TC发起，提交，报告分支事务的

请求，并且服务本地事务的回滚，提交。

**Seata处理一个全局事务的流程如下：**

* TM向TC请求发起一个全局事务，TC返回一个代表这个全局事务的XID；
* XID在rpc中传播给每一个调用链中的服务；
* 每个RM拿到XID后向TC发起一个分支事务，TC返回一个代表这个分支事务的

XID；

* RM完成本地分支的业务，提交本地分支，并且报告给TC；
* 全局事务调用链处理完毕，TM根据有无异常向TC发起全局事务的提交或者回滚。**回滚：**
* 假设某个RM本地事务失败。该RM自身驱动本地事务回滚，并且报告给TC；
* TM检测到了某个分支事务失败，向TC发起全局事务回滚；
* TC给每一个RM发送消息，通知它们全部回滚；
* TC将全局事务回滚的结果发送给TM，全局事务结束。

### 基于**Seata分布式事务示例**

为了方便开发者掌握seata的使用方式，在框架中开发了CETC\_TX聚合模块，聚合了两个微服务模块，模拟了银行跨行转账的业务，实现了对分布式事务的控制，本示例的流程如下：

* 请求bank1进行转账，传入转账金额；
* bank1减少转账金额，调用bank2，传入转账金额；
* 跨行转账成功/失败。

工程步骤如下。

* 安装分布式事务协调器，安装步骤见附录；
* 新建两个数据库bank1和bank2，导入sql文件；
* 新建cetc-tx模块，在pom文件中添加版本控制信息；
* 在cetc-tx模块下新建cetc-seata-demo-bank1和cetc-seata-demo-bank2模块，分别引入第三方依赖spring-cloud-alibaba-seata；
* 在cetc-seata-demo-bank1和cetc-seata-demo-bank2模块下引入配置文件file.conf和registry.conf，注意在file.conf文件中需要修改ip地址，如下

service {  
 #vgroup->rgroup  
 vgroup\_mapping.cetc-seata-demo-bank1-fescar-service-group = "default"  
 #only support single node  
 default.grouplist = "192.168.0.108:31361"  
 #degrade current not support  
 enableDegrade = false  
 #disable  
 disable = false  
 #unit ms,s,m,h,d represents milliseconds, seconds, minutes, hours, days, default permanent  
 max.commit.retry.timeout = "-1"  
 max.rollback.retry.timeout = "-1"  
}

* 在cetc-seata-demo-bank1和cetc-seata-demo-bank2模块中配置数据源代理；

@Configuration  
public class DatabaseConfiguration {  
 private final ApplicationContext applicationContext;  
 public DatabaseConfiguration(ApplicationContext applicationContext) {  
 this.applicationContext = applicationContext;  
 }  
 @Bean  
 @ConfigurationProperties(prefix = "spring.datasource.ds0")  
 public DruidDataSource ds0() {  
 DruidDataSource druidDataSource = new DruidDataSource();  
 return druidDataSource;  
 }  
 @Primary  
 @Bean  
 public DataSource dataSource(DruidDataSource ds0) {  
 DataSourceProxy pds0 = new DataSourceProxy(ds0);  
 return pds0;  
 }  
}

* 发起一个张三向李四转账的事务，首先扣减张三的金额，再调用李四微服务的转账接口，代码如下；

@Transactional  
@GlobalTransactional//开启全局事务  
@Override  
public void updateAccountBalance(String accountNo, Double amount) {  
 //扣减张三的金额  
 accountInfoDao.updateAccountBalance(accountNo,amount \*-1);  
 //调用李四微服务，转账  
 String transfer = bank2Client.transfer(amount);  
 if("fallback".equals(transfer)){  
 //调用李四微服务异常  
 throw new RuntimeException("调用李四微服务异常"+RootContext.*getXID*());  
 }  
 if(amount == 2){  
 //人为制造异常  
 throw new RuntimeException("bank1 make exception..");  
 }  
}

@Transactional  
 @Override  
 public void updateAccountBalance(String accountNo, Double amount) {  
 *log*.info("bank2 service begin,XID：{}",RootContext.*getXID*());  
 //李四增加金额  
 accountInfoDao.updateAccountBalance(accountNo,amount);  
 if(amount==3){  
 //人为制造异常  
 throw new RuntimeException("bank2 make exception..");  
 }  
 }  
}

在上述代码中一共设计了三种异常来验证分布式事务，将李四微服务关闭模拟网络故障，当转账金额为2时人为制造张三微服务故障，当转账金额为3时人为制造李四微服务故障，通过postman可以模拟请求发送，观察数据库便能验证分布式事务的控制。

### **分布式事务之TCC**

TCC是Try、Confirm、Cancel三个词语的缩写，TCC要求每个分支事务实现三个操作：预处理Try、确认Confirm、撤销Cancel。Try操作做业务检查及资源预留，Confirm做业务确认操作，Cancel实现一个与Try相反的操作即回滚操作。

**TCC分为三个阶段：**

1. Try 阶段是做业务检查(一致性)及资源预留(隔离)，此阶段仅是一个初步操作，它和后续的Confirm 一起才能真正构成一个完整的业务逻辑。

2. Confirm 阶段是做确认提交，Try阶段所有分支事务执行成功后开始执行 Confirm。通常情况下，采用TCC则 认为 Confirm阶段是不会出错的。即：**只要Try成功，Confirm一定成功。若Confirm阶段真的出错了，需引入重试机制或人工处理。**

3. Cancel 阶段是在业务执行错误需要回滚的状态下执行分支事务的业务取消，预留资源释放。**通常情况下，采用TCC则认为Cancel阶段也是一定成功的。若Cancel阶段真的出错了，需引入重试机制或人工处理。**

**TCC需要注意三种异常处理分别是空回滚、幂等、悬挂:**

**空回滚：**

在没有调用 TCC 资源 Try 方法的情况下，调用了二阶段的 Cancel 方法，Cancel 方法需要识别出这是一个空回滚，然后直接返回成功。出现原因是当一个分支事务所在服务宕机或网络异常，分支事务调用记录为失败，这个时候其实是没有执行Try阶段，当故障恢复后，分布式事务进行回滚则会调用二阶段的Cancel方法，从而形成空回滚。

解决思路是关键就是要识别出这个空回滚。思路很简单就是需要知道一阶段是否执行，如果执行了，那就是正常回滚；如果没执行，那就是空回滚。前面已经说过TM在发起全局事务时生成全局事务记录，全局事务ID贯穿整个分布式事务调用链条。再额外增加一张分支事务记录表，其中有全局事务 ID 和分支事务 ID，第一阶段 Try 方法里会插入一条记录，表示一阶段执行了。Cancel 接口里读取该记录，如果该记录存在，则正常回滚；如果该记录不存在，则是空回滚。

**幂等：**

通过前面介绍已经了解到，为了保证TCC二阶段提交重试机制不会引发数据不一致，要求 TCC 的二阶段 Try、Confirm 和 Cancel 接口保证幂等，这样不会重复使用或者释放资源。如果幂等控制没有做好，很有可能导致数据不一致等严重问题。

解决思路在上述“分支事务记录”中增加执行状态，每次执行前都查询该状态。

**悬挂：**

悬挂就是对于一个分布式事务，其二阶段 Cancel 接口比 Try 接口先执行。

出现原因是在 RPC 调用分支事务try时，先注册分支事务，再执行RPC调用，如果此时 RPC 调用的网络发生拥堵，通常 RPC 调用是有超时时间的，RPC 超时以后，TM就会通知RM回滚该分布式事务，可能回滚完成后，RPC 请求才到达参与者真正执行，而一个 Try 方法预留的业务资源，只有该分布式事务才能使用，该分布式事务第一阶段预留的业务资源就再也没有人能够处理了，对于这种情况，我们就称为悬挂，即业务资源预留后没法继续处理。

解决思路是如果二阶段执行完成，那一阶段就不能再继续执行。在执行一阶段事务时判断在该全局事务下，“分支事务记录”表中是否已经有二阶段事务记录，如果有则不执行Try。

### **TCC事务解决方案**

Hmily是一个高性能分布式事务TCC开源框架，基于java语言来开发，支持Dubbo,SpringCloud等RPC框架进行分布式事务。

Hmily利用AOP对参与分布式事务的本地方法与远程方法进行拦截处理，通过多方拦截，事务参与者能够透明的调用到另一方的Try、Confirm、Cancel方法；传递事务上下文；并记录事务日志，酌情进行补偿，重试等。Hmily不需要事务协调服务，但需要提供一个数据库来进行日志存储。

**创建数据库：**

创建hmily数据库，用于存储hmily框架记录的数据。

**举例，场景为 A 转账 30 元给 B，A和B账户在不同的服务。**

**方案1**

账户A

try：

检查余额是否够30元

扣减30元

confirm：

空

cancel：

增加30元

账户B

try：

增加30元

confirm：

空

cancel：

减少30元

**方案1说明：**

（1）账户A，这里的余额就是所谓的业务资源，按照前面提到的原则，在第一阶段需要检查并预留业务资源，因此，我们在扣钱 TCC 资源的 Try 接口里先检查 A 账户余额是否足够，如果足够则扣除 30 元。Confirm 接口表示正式提交，由于业务资源已经在 Try 接口里扣除掉了，那么在第二阶段的 Confirm 接口里可以什么都不用做。Cancel接口的执行表示整个事务回滚，账户A回滚则需要把 Try 接口里扣除掉的 30 元还给账户。

（2）账号B，在第一阶段 Try 接口里实现给账户B加钱，Cancel 接口的执行表示整个事务回滚，账户B回滚则需要把Try接口里加的30元再减去。

**方案1的问题分析：**

（1）如果账户A的try没有执行在cancel则就多加了30元。

（2）由于try，cancel、confirm都是由单独的线程去调用，且会出现重复调用，所以都需要实现幂等。

（3）账号B在try中增加30元，当try执行完成后可能会被其它线程给消费了。

（4）如果账户B的try没有执行在cancel则就多减了30元。

问题解决：

（1）账户A的cancel方法需要判断try方法是否执行，正常执行try后方可执行cancel。

（2）try，cancel、confirm方法实现幂等。

（3）账号B在try方法中不允许更新账户金额，在confirm中更新账户金额。

（4）账户B的cancel方法需要判断try方法是否执行，正常执行try后方可执行cancel。

**优化方案：**

**账户A：**

try：

try幂等校验

try悬挂处理

检查余额是否够30元

扣减30元

confirm：

空

cancel：

cancel幂等校验

cancel空回滚处理

增加可用余额30元

**账户B**

try：

空

confirm：

confirm幂等校验

正式增加30元

cancel：

空

### **Hmily实现TCC事务**

**业务说明：**

本实例通过Hmily实现TCC分布式事务，模拟两个账户的转账交易过程。

两个账户分别在不同的银行(张三在bank1、李四在bank2)，bank1、bank2是两个微服务。交易过程是，张三给李四转账指定金额。上述交易步骤，要么一起成功，要么一起失败，必须是一个整体性的事务。

**创建数据库：**

* 创建hmily数据库，用于存储hmily框架记录的数据,执行脚本见附录x。
* 创建bank1库，并导入以下表结构和数据(包含张三账户)，执行脚本见附录x。
* 创建bank2库，并导入以下表结构和数据(包含李四账户)，执行脚本见附录x。

每个数据库都创建try、confifirm、cancel三张日志表，

**引入Maven依赖：**

<dependency>

<groupId>org.dromara</groupId>

<artifactId>hmily‐springcloud</artifactId>

<version>2.0.4‐RELEASE</version>

</dependency>

**配置hmily（见附录）：**

**TCC总结：**

如果拿TCC事务的处理流程与2PC两阶段提交做比较，2PC通常都是在跨库的DB层面，而TCC则在应用层面的处理，需要通过业务逻辑来实现。这种分布式事务的实现方式的优势在于，可以让应用自己定义数据操作的粒度，使得降低锁冲突、提高吞吐量成为可能。**而不足之处则在于对应用的侵入性非常强**，业务逻辑的每个分支都需要实现try、confifirm、cancel三个操作。此外，其实现难度也比较大，需要按照网络状态、系统故障等不同的失败原因实现不同的回滚策略。

### **分布式事务解决方案之可靠消息最终一致性**

**什么是可靠消息最终一致性事务**

可靠消息最终一致性方案是指当事务发起方执行完成本地事务后并发出一条消息，事务参与方(消息消费者)一定能够接收消息并处理事务成功，此方案强调的是只要消息发给事务参与方最终事务要达到一致。 此方案是利用消息中间件完成，事务发起方（消息生产方）将消息发给消息中间件，事务参与方从消息中间件接收消息，事务发起方和消息中间件之间，事务参与方（消息消费方）和消息中间件之间都是通过网络通信，由于网络通信的不确定性会导致分布式事务问题。

因此可靠消息最终一致性方案要解决以下几个问题：

**1.本地事务与消息发送的原子性问题**

本地事务与消息发送的原子性问题即：事务发起方在本地事务执行成功后消息必须发出去，否则就丢弃消息。即实 现本地事务和消息发送的原子性，要么都成功，要么都失败。本地事务与消息发送的原子性问题是实现可靠消息最终一致性方案的关键问题。

先来尝试下这种操作，先发送消息，再操作数据库：

begin transaction；

//1.发送MQ

//2.数据库操作

commit transation;

这种情况下无法保证数据库操作与发送消息的一致性，因为可能发送消息成功，数据库操作失败。你立马想到第二种方案，先进行数据库操作，再发送消息

begin transaction；

//1.数据库操作

//2.发送MQ

commit transation;

这种情况下貌似没有问题，如果发送MQ消息失败，就会抛出异常，导致数据库事务回滚。但如果是超时异常，数据库回滚，但MQ其实已经正常发送了，同样会导致不一致。

**2、事务参与方接收消息的可靠性**

事务参与方必须能够从消息队列接收到消息，如果接收消息失败可以重复接收消息。

**3、消息重复消费的问题**

由于网络2的存在，若某一个消费节点超时但是消费成功，此时消息中间件会重复投递此消息，就导致了消息的重复消费。要解决消息重复消费的问题就要实现事务参与方的方法幂等性。

**解决方案**

* 本地消息表方案

本地消息表这个方案最初是eBay提出的，此方案的核心是通过本地事务保证数据业务操作和消息的一致性，然后通过定时任务将消息发送至消息中间件，待确认消息发送给消费方成功再将消息删除。

下面以注册送积分为例来说明：下例共有两个微服务交互，用户服务和积分服务，用户服务负责添加用户，积分服务负责增加积分。

**1、用户注册**

用户服务在本地事务新增用户和增加 “积分消息日志”。（用户表和消息表通过本地事务保证一致）

下边是伪代码 ：

begin transaction；

//1.新增用户

//2.存储积分消息日志

commit transation;

这种情况下，本地数据库操作与存储积分消息日志处于同一个事务中，本地数据库操作与记录消息日志操作具备原子性。

**2、定时任务扫描日志**

如何保证将消息发送给消息队列呢？

经过第一步消息已经写到消息日志表中，可以启动独立的线程，定时对消息日志表中的消息进行扫描并发送至消息中间件，在消息中间件反馈发送成功后删除该消息日志，否则等待定时任务下一周期重试。

**3、消费消息**

如何保证消费者一定能消费到消息呢？

这里可以使用MQ的ack（即消息确认）机制，消费者监听MQ，如果消费者接收到消息并且业务处理完成后向MQ 发送ack（即消息确认），此时说明消费者正常消费消息完成，MQ将不再向消费者推送消息，否则消费者会不断重试向消费者来发送消息。

积分服务接收到”增加积分“消息，开始增加积分，积分增加成功后向消息中间件回应ack，否则消息中间件将重复投递此消息。由于消息会重复投递，积分服务的”增加积分“功能需要实现幂等性（通过redis）。需要注意的是，在实际生产中，需要设置ack机制的最大重试次数，防止消息中间件由于队列堵塞产生内存泄漏的现象，同时，消费方需要做集群。

# 常用组件安装

## 消息中间件Rabbitmq

RabbitMQ 官网上提供了 6 中工作模式：简单模式、工作队列模式、发布/订阅模式（广播模式）、路由模式、主题模式（通配符模式） 和 RPC 模式。这么多交换机中，最常用的交换机有三种：direct、topic、fanout，分别是：“直接连接交换机”，“主题路由匹配交换机”，“无路由交换机”。

## 缓存型数据库Redis

* 配置文件上传

上传redis的conf文件（改文件来源于redis的官网），注释掉bind，想要远程连接，首先把绑定到本地地址的配置注释掉，关闭保护模式protected-mode no，开启redis持久化，appendonly yes。

* 启动redis

# docker run -d -p 6379:6379 -v /usr/redis/redis.conf:/etc/redis/redis.conf -v /usr/redis/data:/data --name myredis redis:3.2 redis-server /etc/redis/redis.conf

-d：容器后台运行，并返回容器ID

-p：指定端口映射，格式为：主机（宿主）端口：容器端口

-v：绑定一个卷，资源映射

--name：给容器命名

redis-server  /etc/redis/redis.conf：指定容器启动时执行的命令

## Devops

### **代码库建设**

* 启动gitlab

docker run -d -p 444:443 -p 8082:80 -p 222:22 --name gitlab --restart always -v /home/gitlab/config:/etc/gitlab -v /home/gitlab/logs:/var/log/gitlab -v /home/gitlab/data:/var/opt/gitlab gitlab

* 按上面的方式，gitlab容器运行没问题，但在gitlab上创建项目的时候，生成项

目的URL访问地址是按容器的hostname来生成的，也就是容器的id。作为gitlab服务器，我们需要一个固定的URL访问地址，于是需要配置gitlab.rb（宿主机路径：/home/gitlab/config/gitlab.rb），修改配置。

$ vim /home/gitlab/config/gitlab.rb

* # 配置http协议所使用的访问地址,不加端口号默认为80

external\_url 'http://192.168.199.231'

* # 配置ssh协议所使用的访问地址和端口

gitlab\_rails['gitlab\_ssh\_host'] = '192.168.199.231'

gitlab\_rails['gitlab\_shell\_ssh\_port'] = 222

# 此端口是run时22端口映射的222端口:wq #保存配置文件并退出

* # 重启gitlab容器

$ docker restart gitlab

### **持续集成（CI）**

* 配置docker环境，将docker配置成外网访问模式

ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock

* 上传jenkins的镜像并执行加载命令

docker load -i ~

* 创建目录

mkdir /var/jenkins\_home，上传maven及其私服到jenkins\_home，配置setting

执行命令sudo chown -R 1000 /var/jenkins\_home

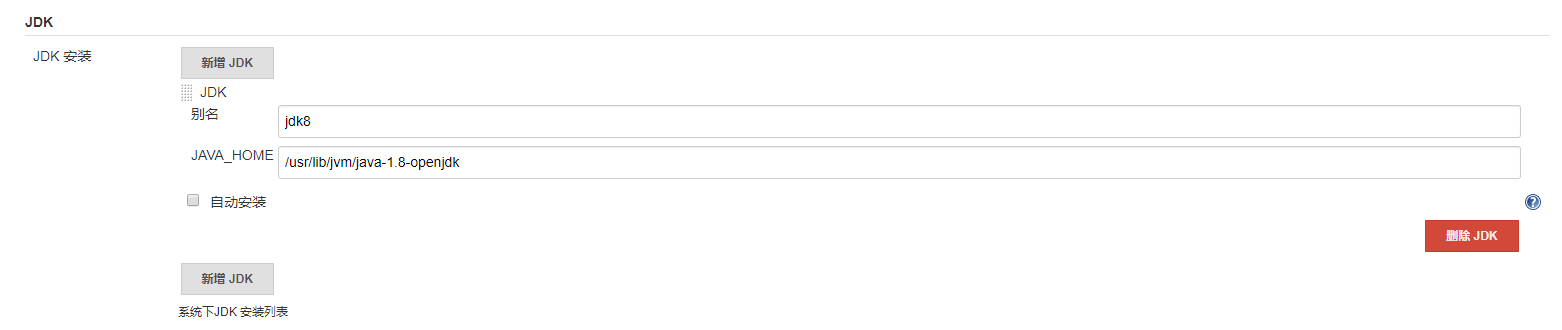
* 启动一个jenkins容器，jenkins\_home为jenkins的映射卷

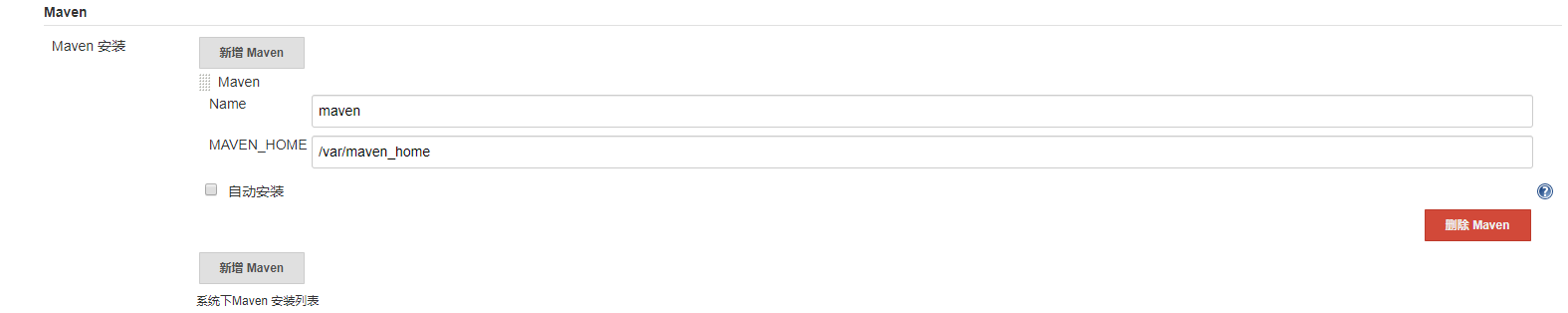
docker run -di --name jenkins -p 8081:8080 -p 50000:50000 -v /var/jenkins\_home/:/var/jenkins\_home/ jenkinsci/blueocean

* 进入jenkins的容器，根据登录页提示获取密钥

docker exec -it jenkins bash

* 上传汉化插件和系统自带插件，重启，完成jenkins的安装





# Kubernetes

## 基本对象概述

* Pod

Pod是最小部署单元，一个Pod有一个容器或多个容器组成，Pod中容器共享存储和网络，在同一台Docker主机上运行。

* Service

Service一个应用服务抽象，定义了Pod逻辑集合和访问这个Pod集合的策略。

Service代理Pod集合对外表现是一个访问入口，分配一个集群IP地址，来自这个IP的请求将负载均衡转发后端Pod中的容器。

Service通过Lable Selector选择一组Pod提供服务。

* Volume

数据卷，共享Pod中容器使用的数据。

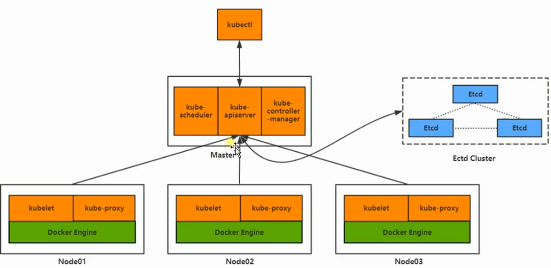
* Namespace

命名空间将对象逻辑上分配到不同的Namespace，可以是不同项目、用户等区分管理，并设定控制策略，从而实现多租户。命名空间也称为虚拟集群。

* Lable

标签用于区分对象（比如Pod、Service），键、值对存在，每个对象可以有对个标签，通过标签关联对象。

## 系统架构及组件功能



## 集群部署

### **环境规划**

|  |  |
| --- | --- |
| 软件 | 版本 |
| Linux操作系统 | Centos7.4\_x64 |
| Kubernetes | 1.15.1（2019年8月发布） |
| Docker | 18.xx-ce |
| Etcd | 3.0 |

|  |  |  |  |
| --- | --- | --- | --- |
| 角色 | IP | 组件 | 推荐配置 |
| master | xx | kube-apiserver  kube-controller-manager  kube-scheduler  etcd | CPU 2核心  +4G内存 |
| node01 | xx | kubelet  kube-proxy  docker  flannel  etcd |
| node02 | xx | kubelet  kube-proxy  docker  flannel  etcd |

### **集群安装准备**

* 设置系统主机名 hostnamectl set-hostname k8s-master01(包括node节点)
* 修改hosts文件 vi /etc/hosts (包括node节点)

192.168.0.108 k8s-master01

192.168.0.111 k8s-node01

192.168.0.113 k8s-node02

* 在Linux之间相互拷贝文件scp /etc/hosts root@k8s-node01:/etc/hosts
* 安装依赖包 yum install --downloadonly --downloaddir=temp conntrack ntpdate ntp ipvsadm ipset jq iptables curl sysstat libseccomp wgetvimnet-tools git

导入temp文件夹。

* 进入temp文件夹进行安装 # rpm -ivh --force --nodeps \*.rpm
* 设置防火墙为Iptables并设置空规则

systemctl stop firewalld && systemctl disable firewalld

yum install --downloadonly --downloaddir=iptables iptables-services（外场使用）

导入iptables文件夹

* 进入iptables文件夹进行安装# rpm -ivh --force --nodeps \*.rpm
* 执行开启命令

systemctl start iptables && systemctl enable iptables && iptables -F && service iptables save

* 关闭虚拟内存和SELINUX

swapoff -a && sed -i '/ swap / s/^\(.\*\)$/#\1/g' /etc/fstab

setenforce 0 && sed -i 's/^SELINUX=.\*/SELINUX=disabled/' /etc/selinux/config

* 调整内核参数，对于k8s

cat > kubernetes.conf <<EOF

net.bridge.bridge-nf-call-iptables=1

net.bridge.bridge-nf-call-ip6tables=1

net.ipv4.ip\_forward=1

net.ipv4.tcp\_tw\_recycle=0

vm.swappiness=0 # 禁止使用 swap 空间，只有当系统 OOM 时才允许使用它vm.overcommit\_memory=1 # 不检查物理内存是否够用

vm.panic\_on\_oom=0 # 开启OOM

fs.inotify.max\_user\_instances=8192

fs.inotify.max\_user\_watches=1048576

fs.file-max=52706963

fs.nr\_open=52706963

net.ipv6.conf.all.disable\_ipv6=1

net.netfilter.nf\_conntrack\_max=2310720

EOF

cp kubernetes.conf /etc/sysctl.d/kubernetes.conf

sysctl -p /etc/sysctl.d/kubernetes.conf

* 调整系统时区

# 设置系统时区为中国/上海

timedatectl set-timezone Asia/Shanghai

# 将当前的 UTC 时间写入硬件时钟

timedatectl set-local-rtc 0

# 重启依赖于系统时间的服务

systemctl restart rsyslog

systemctl restart crond

* 关闭系统不需要的服务 systemctl stop postfix && systemctl disable postfix
* 设置rsyslogd和systemd journald

mkdir /var/log/journal # 持久化保存日志的目录

mkdir /etc/systemd/journald.conf.d

cat > /etc/systemd/journald.conf.d/99-prophet.conf <<EOF

[Journal]

# 持久化保存到磁盘

Storage=persistent

# 压缩历史日志

Compress=yes

SyncIntervalSec=5m

RateLimitInterval=30s

RateLimitBurst=1000

# 最大占用空间 10G

SystemMaxUse=10G

# 单日志文件最大 200M

SystemMaxFileSize=200M

# 日志保存时间 2 周

MaxRetentionSec=2week

# 不将日志转发到 syslog

ForwardToSyslog=no

EOF

systemctl restart systemd-journald

* 升级系统内核为4.44

CentOS 7.x 系统自带的 3.10.x 内核存在一些 Bugs，导致运行的 Docker、Kubernetes 不稳定

#rpm -Uvh <http://www.elrepo.org/elrepo-release-7.0-3.el7.elrepo.noarch.rpm>

# 安装完成后检查 /boot/grub2/grub.cfg 中对应内核 menuentry 中是否包含 initrd16 配置，如果没有，再安装一次！

#yum --enablerepo=elrepo-kernel install --downloadonly --downloaddir=core kernel-lt

进入core包 rpm -ivh --force --nodeps \*.rpm

# 设置开机从新内核启动

grub2-set-default 'CentOS Linux (4.4.189-1.el7.elrepo.x86\_64) 7 (Core)'

reboot

* kube-proxy开启ipvs的前置条件

modprobe br\_netfilter

cat > /etc/sysconfig/modules/ipvs.modules <<EOF

#!/bin/bash

modprobe -- ip\_vs

modprobe -- ip\_vs\_rr

modprobe -- ip\_vs\_wrr

modprobe -- ip\_vs\_sh

modprobe -- nf\_conntrack\_ipv4

EOF

chmod 755 /etc/sysconfig/modules/ipvs.modules && bash /etc/sysconfig/modules/ipvs.modules && lsmod | grep -e ip\_vs -e nf\_conntrack\_ipv4

### **Docker安装**

* 上传docker安装包，解压 tar -xvf docker-18.06.3-ce.tgz，将解压出来的docker文件内容移动到/usr/bin/目录下 cp docker/\* /usr/bin/，将docker注册为service vi /etc/systemd/system/docker.service，将下列配置加到docker.service中并保存。

[Unit]

Description=Docker Application Container Engine

Documentation=https://docs.docker.com

After=network-online.target firewalld.service

Wants=network-online.target

[Service]

Type=notify

# the default is not to use systemd for cgroups because the delegate issues still

# exists and systemd currently does not support the cgroup feature set required

# for containers run by docker

ExecStart=/usr/bin/dockerd

ExecReload=/bin/kill -s HUP $MAINPID

# Having non-zero Limit\*s causes performance problems due to accounting overhead

# in the kernel. We recommend using cgroups to do container-local accounting.

LimitNOFILE=infinity

LimitNPROC=infinity

LimitCORE=infinity

# Uncomment TasksMax if your systemd version supports it.

# Only systemd 226 and above support this version.

#TasksMax=infinity

TimeoutStartSec=0

# set delegate yes so that systemd does not reset the cgroups of docker containers

Delegate=yes

# kill only the docker process, not all processes in the cgroup

KillMode=process

# restart the docker process if it exits prematurely

Restart=on-failure

StartLimitBurst=3

StartLimitInterval=60s

[Install]

WantedBy=multi-user.target

执行docker的启动命令

chmod +x /etc/systemd/system/docker.service #添加文件权限并启动docker

systemctl daemon-reload #重载unit配置文件

systemctl start docker #启动docker

systemctl enable docker.service #设置开机自启动

验证

systemctl status docker #查看Docker状态

docker -v #查看docker版本

uname -r查看当前内核版本

## 创建 /etc/docker 目录

mkdir /etc/docker

# 配置 daemon.

cat > /etc/docker/daemon.json <<EOF

{

"exec-opts": ["native.cgroupdriver=cgroupfs"],

"log-driver": "json-file",

"log-opts": {

"max-size": "100m"

}

}

EOF

mkdir -p /etc/systemd/system/docker.service.d 主要用来存放docker的配置文件

# 重启docker服务systemctl daemon-reload && systemctl restart docker && systemctl enable docker

### 基于Kubeadm安装K8S集群

* 安装Kubeadm（主从配置）

上传kubelet文件夹

进入kubelet文件夹，执行命令：rpm -ivh --force --nodeps \*.rpm

设置成开机自启 systemctl enable kubelet.service

上传kubeadm-basic.images.tar.gz和load-images.sh到master节点解压

复制文件到node节点

scp -r kubeadm-basic.images load-images.sh root@k8s-node01:/root/

chmod 777 ./\*.sh

执行./load-images.sh

* 初始化主节点

获得kubeadm默认的初始化模板 kubeadm config print init-defaults > kubeadm-config.yaml

vim kubeadm-config.yaml

修改localAPIEndpoint:advertiseAddress:物理机地址

修改kubernetsVersion: v1.15.1

添加networking:

dnsDomain: cluster.local

podSubnet: "10.244.0.0/16"

serviceSubnet: 10.96.0.0/12

添加

---

apiVersion: kubeproxy.config.k8s.io/v1alpha1

kind: KubeProxyConfiguration

featureGates:

SupportIPVSProxyMode: true

mode: ipvs

开始初始化：kubeadm init --config=kubeadm-config.yaml --experimental-upload-certs | tee kubeadm-init.log

* To start using your cluster, you need to run the following as a regular user:

mkdir -p $HOME/.kube

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

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

### 安装Flannel网络插件

mkdir install-k8s

mv kubeadm-init.log kubeadm-config.yaml install-k8s/core

mkdir plugin

cd plugin

mkdir flannel

cd flannel

上传kube-flannel.yml到/flannel目录下

上传flannel.tar到

执行kubectl create -f kube-flannel.yml

查询是否执行成功 kubectl get pod -n kube-system

kubeadm join 192.168.0.108:6443 --token abcdef.0123456789abcdef \

--discovery-token-ca-cert-hash sha256:e268d05d76b3a128e3c07b2ab76c2901c3a17826e4b8ed98ddd90da44da5bdc4

### **制品库安装**

配置/etc/docker/daemon.json

{

"exec-opts": ["native.cgroupdriver=cgroupfs"],

"log-driver": "json-file",

"log-opts": {

"max-size": "100m"

},

"insecure-registries":["https://hub.atcetc.com"]

}

此配置在所以安装docker的机器上都得配置

上传docker-compose，执行mv docker-compose /usr/local/bin

执行命令chmod a+x /usr/local/bin/docker-compose

执行命令mv harbor /usr/local/

执行命令cd /usr/local/harbor/

执行命令vi harbor.cfg

修改hostname = hub.atcetc.com和ui\_url\_protocol = https

执行命令mkdir -p /data/cert

执行命令cd /data/cert

openssl genrsa -des3 -out server.key 2048

openssl req -new -key server.key -out server.csr

cp server.key server.key.org

openssl rsa -in server.key.org -out server.key

openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt

chmod a+x \*

cd /usr/local/harbor/ 执行./install.sh

mkdir /data/certchmod -R 777

在master节点上运行 kubectl run eureka-deployment --image=hub.atcetc.com/library/cetc\_eureka:v1 --port=6868 --repicas=1

开启三个副本kubectl scale --replicas=3 deployment/eureka-deployment

为deployment开启service kubectl expose deployment eureka-deployment --port=6868 --target-port=6868

查看所有的虚拟ip的映射 ipvsadm -Ln

查看svc kubectl get svc

修改svc kubectl edit svc

修改type为 NodePort

### **安装Kuboard**

* 上传kuboard镜像

docker load < kuboard.tar

* 为镜像重新添加标签

docker tag eipwork/kuboard

* 上传kuboard.yaml,在master节点上执行kuboard的安装命令

kubectl apply -f kuboard.yaml

* 查看kuboard运行状态

kubectl get pods -l k8s.eip.work/name=kuboard -n kube-system

* 获取管理员用户的Token

kubectl -n kube-system get secret $(kubectl -n kube-system get secret | grep kuboard-user | awk '{print $1}') -o go-template='{{.data.token}}' | base64 -d

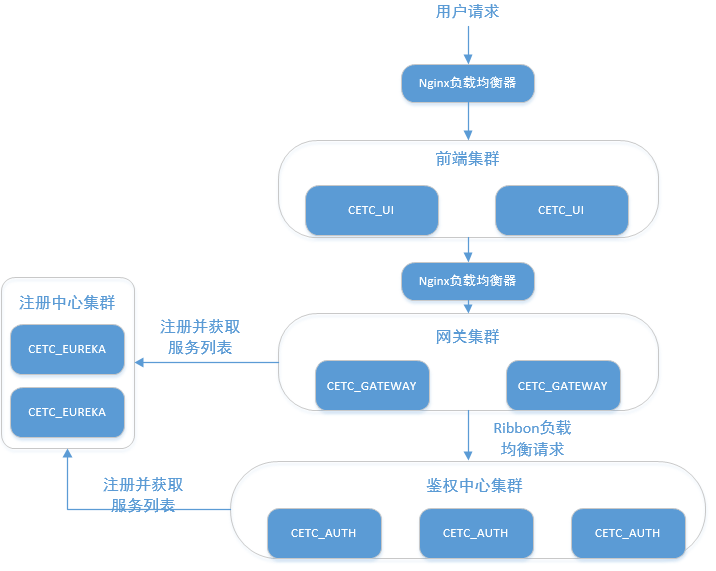
* 访问Kuboard

Kuboard Service 使用了 NodePort 的方式暴露服务，NodePort 为 32567。

## 负载均衡

负载均衡是指将负载分摊到多个执行单元上，常见的负载均衡有两种方式。一种是独立进程单元，通过负载均衡策略，将请求转发到不同的执行单元上，属于反向代理，例如Ngnix。另一种是将负载均衡逻辑以代码的形式封装到服务消费者的客户端上，服务消费者客户端维护了一份服务提供者的信息列表（以固定时间间隔从Eureka获取注册信息列表），有了信息列表，通过负载均衡策略将请求分摊给多个服务提供者，从而达到负载均衡的目的（例如Ribbon）。

课题对这两种模式均有使用，Ribbon的使用范畴属于开发阶段，内置到代码中，而Nginx是在部署实施阶段使用，例如部署了两个网关模块，ip地址分别为192.168.0.110和 192.168.0.120，只需要在Nginx配置相关的代理信息即可，最终暴露给前端的是一个虚拟地址，如图1所示为系统的负载均衡高可用设计。



### **Nginx安装步骤**

（1）安装 pcre 依赖

第一步 联网下载 pcre 压缩文件依赖

wget http://downloads.sourceforge.net/project/pcre/pcre/8.37/pcre-8.37.tar.gz

第二步 解压压缩文件

使用命令 tar xvf pcre-8.37.tar.gz

第三步 编译安装

./configure 完成后，回到 pcre 目录下执行 make，最后执行 make install

（2）安装openssl、zlib、gcc 依赖

yum -y install make zlib zlib-devel gcc-c++ libtool openssl openssl-devel

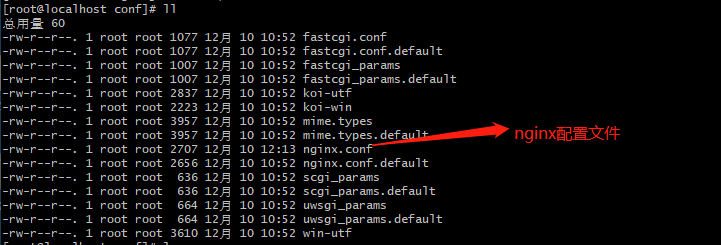
（3）安装 nginx

### **Nginx使用步骤**

Nginx常用命令如下：

* 查看版本号./nginx -v
* 停止 ./nginx -s stop
* 启动./nginx
* 重新加载 ./nginx -s reload

完成Nginx的安装后，修改Nginx 的配置文件：



配置文件中的内容主要包含三部分内容：  
（1）全局块：配置服务器整体运行的配置指令，比如 worker\_processes 1，处理并发数的配置，建议设置成和CPU相等数量的值。

（2）events 块：影响 Nginx 服务器与用户的网络连接，比如 worker\_connections 1024，支持的最大连接数为 1024。

（3）http 块还包含两部分：http全局块 server 块。

### **Nginx实现负载均衡**

浏览器中输入一个地址，通过负载均衡能够将请求平均（使用轮询策略）分配到不同的服务器上，在nginx的配置文件中进行负载均衡的配置如下。



Nginx分配服务器策略主要有如下四种：

第一种轮询（默认）：每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器down掉，能自动剔除。

第二种weightweight：代表权重默认为1,权重越高被分配的客户端越多。

第三种ip\_hash：每个请求按访问ip的hash结果分配，这样每个访客固定访问一个后端服务器。

第四种fair（第三方）：按后端服务器的响应时间来分配请求，响应时间短的优先分配。

### **Nginx配置高可用集群**

配置高可用的准备工作：

（1）需要两台服务器192.168.17.129 和192.168.17.131；

（2）在两台服务器安装nginx；

（3）在两台服务器安装keepalived。

keepalivied安装步骤：

（1）使用yum命令进行安装yum install keepalived –y；

（2）安装之后，在etc里面生成目录keepalived，有文件keepalived.conf。

完成高可用配置

（1）修改/etc/keepalived/keepalivec.conf配置文件，见附录；

（2）在/usr/local/src添加检测脚本；

#!/bin/bash

A=`ps -C nginx –no-header |wc -l`

if [ $A -eq 0 ];then

/usr/local/nginx/sbin/nginx

sleep 2

if [ `ps -C nginx --no-header |wc -l` -eq 0 ];then

killall keepalived

fi

fi

（3）把两台服务器上nginx和keepalived启动

启动nginx：./nginx

启动keepalived：systemctl start keepalived.service

最终测试

（1）在浏览器地址栏输入虚拟ip地址，能够访问到Nginx；

（2）停掉主服务器的的Nginx，重复步骤（1），依然能够访问到Nginx；

（3）完成高可用的测试。