《我的奋斗》

目录

[1. Java基础 3](#_Toc651)

[2.Java高级 3](#_Toc31683)

[3.Java虚拟机 3](#_Toc7908)

[4.Spring 3](#_Toc15895)

[5.Mybatis 3](#_Toc26603)

[6.SpringBoot 3](#_Toc12442)

[7.SpringCloud 3](#_Toc11963)

[8.Redis 3](#_Toc27578)

[9.ElasticSearch 3](#_Toc3781)

[10.Zookeeper 4](#_Toc22265)

[10.1简介 4](#_Toc27665)

[10.2 Zookeeper文件夹主要目录介绍 4](#_Toc7844)

[10.3 Zookeeper配置文件介绍 4](#_Toc16525)

[10.4 Zookeeper的作用 6](#_Toc16476)

[10.5 Zookeeper命令 6](#_Toc6399)

[10.6 Session的基本原理 10](#_Toc11474)

[10.7 watcher机制 10](#_Toc26194)

[10.7.1 Watcher事件类型 11](#_Toc12382)

[10.8 Acl权限控制 13](#_Toc11845)

[11.Dobblo 16](#_Toc8542)

[12.MySQL 16](#_Toc16090)

[13.MyCat 16](#_Toc2687)

[14.RocketMQ 16](#_Toc25076)

[15.Linux 16](#_Toc30814)

[16.设计模式 16](#_Toc15092)

[17.分布式锁 16](#_Toc3364)

[18.事务 17](#_Toc1642)

[18.1数据库本地事务：ACID 17](#_Toc21001)

[18.2 InnoDB 实现原理 17](#_Toc29706)

[18.3分布式事务 18](#_Toc15839)

[18.3.1分布式的理论CAP 18](#_Toc21891)

[18.3.2 BASE理论 18](#_Toc24859)

[18.4分布式事务的解决方案 19](#_Toc28918)

[18.4.1 2PC/3PC 19](#_Toc27377)

[18.4.2 SAGA事务 20](#_Toc31758)

[18.4.3 TCC 22](#_Toc6267)

[18.4.4 MQ事务 22](#_Toc11542)

# Java基础

## 1.1泛型

Java的泛型（generics）。它是JDK 5中引入的一个新特性，允许在定义类和接口的时候使用类型参数（type parameter）。声明的类型参数在使用时用具体的类型来替换。

Java中的泛型基本上完全在 *编译器中实现*，由 *编译器执行类型检查* 和 *类型推断* ，然后生成普通的**非泛型的字节码**。这种实现技术称为**擦除（erasure）**（编译器使用泛型类型信息保证类型安全，然后在生成字节码之前将其清除）。

问题来了：

如果想做的通用的话，怎么办呢？例如下面的代码

**class** Fruit { }  
**class** Orange **extends** Fruit { }

List<Fruit> list = **new** ArrayList<Orange>();

它们之间会报错的说Incompatible types。

原因是：编译器认为的容器之间没有继承关系，因此不能这样做。

为了解决该问题，于是引出了<? extens T> 和 <? super T>的办法，让他们有关系。

### 1.1.1上界通配符

<? extends T> 上界只能外围取，不能往里放

将代码修改修改，此时第二行的代码就不会报错了。

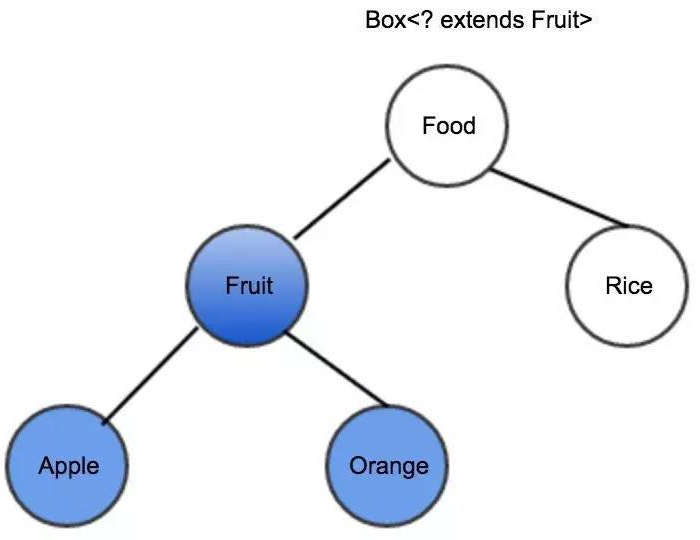
List<Fruit> list = **new** ArrayList<Orange>();

List<? **extends** Fruit> fruits = **new** ArrayList<Orange>();

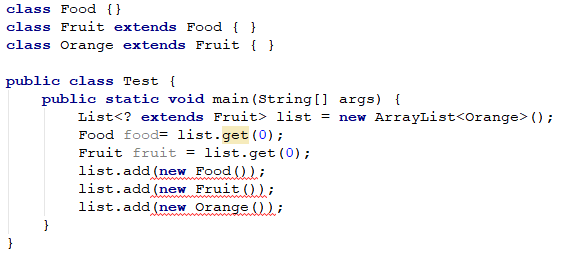
此时编译器会把它们看成是由继承关系的两个类型了。

? extends Fruit 即？是Fruit的子类或者该类。

取值范围即上界只能外围取，不能往里放 如图所示：



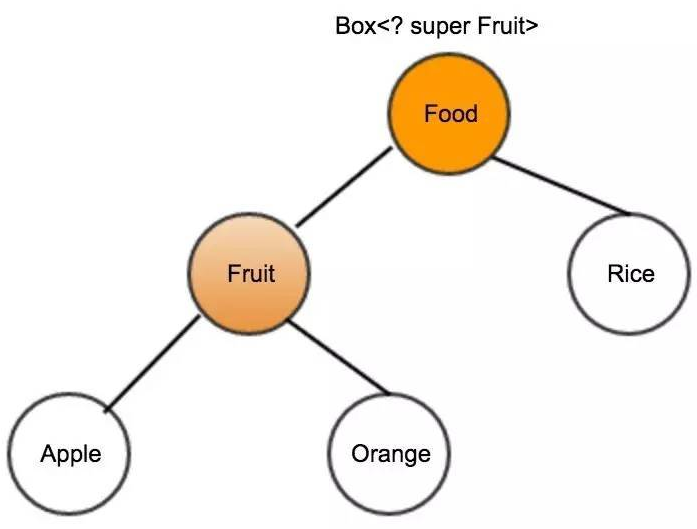
结合代码展示的话，如下图所示：

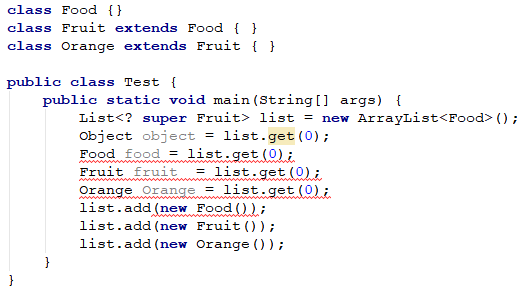


**原因**：Java编译器只知道容器内是Fruit或者它的派生类，但是不知道是什么类型。可能是Fruit、可能是Orange、可能是Apple？当编译器在看到box用Box赋值后，它就把容器里表上占位符“AAA”而不是“水果”等，当在插入时编译器不能匹配到这个占位符，所有就会出错。

### 1.1.2下界通配符

<? super T> 下界不影响往里存，往外取只能放在Object 对象里





**原因**：下界规定元素的最小的粒度，实际上是容器的元素的类型控制。所以放比Fruit粒度小的如Orange、Apple都行，但往外取时，只有所有类的基类Object对象才能装下。但是这样的话，元素的类型信息就全部消失了。

# Java高级

## AOP

AOP是一种编程范式,不是编程语言,为了解决特定的问题,而不是所有的问题的。

初衷：为了解决代码重复,关注点分离的问题。

好处：集中处理某一关注点/横切面逻辑,可以很方便添加/删除关注点

应用场景：权限控制,缓存控制,事务控制......

### 2.1.1切面表达式

pointcut expression - 切面表达式

expression:

designators(指示器)

匹配方法：

execution()

匹配注解：

@target()

@args()

@within()

@annotaion()

匹配包/类型

within()

匹配对象

this()

bean()

target()

匹配参数

args()

wildecards(通配符)

\* （匹配任意数量的字符）

+ （匹配制定数量的类及其子类）

.. （一般用于匹配任意数量的子包或参数）

operators(运算符)

&& （与）

|| （或）

! （非）

Advice注解

1.@Before 前置通知

2.@After 后置通知,方法执行完后

3.@AfterReturning 返回通知,成功执行后

4.AfterThrowing 异常通知,抛出异常后

5.@Around 环绕通知

### 2.1.2 AOP原理

AOP织入的时机

1.编译期(Aspectj)

2.类加载时(AspectJ 5+)

3.运行时（Spring AOP）

运行时织入是如何实现的呢？——答案是代理实现。

代理:

静态代理

动态代理:

通过接口实现的代理

通过继承实现的代理

静态代理的缺点: 代理越多,重复越多。

动态代理：

JDK代理：

只能代理接口的

通过java. lang.reflect.Proxy去动态生成代理类

实现java.lang.reflect.InvocationHandler完成织入

cglib代理：

通过继承的方式实现代理类

因为基于继承的方式实现代理,所以无法对static,final类进行代理

因为基于继承的方式实现代理,所以无法对static,private方法进行代理

Spring如何选择代理bean?

如果目标对象实现了接口,则默认采用JDK动态代理

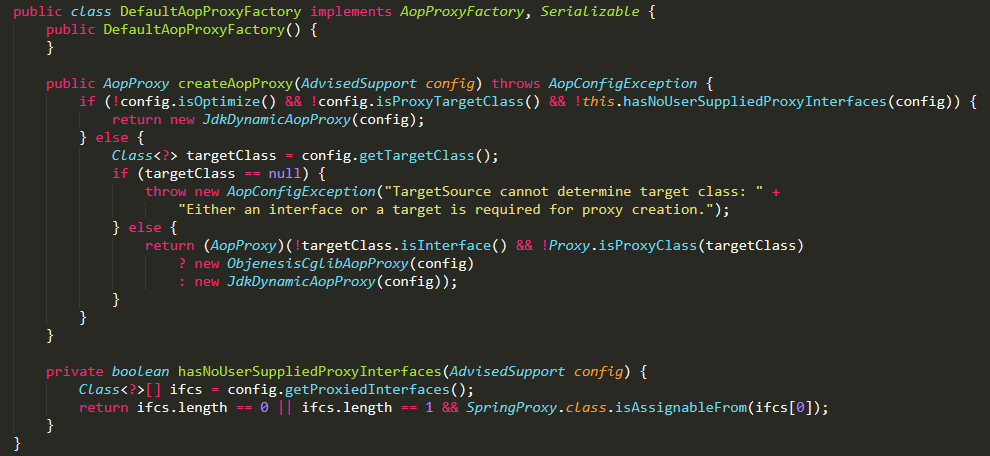
如果目标对象没有实现接口,则采用Cglib进行动态代理

如果目标对象实现了接口,且强制cglib代理,则使用cglib代理

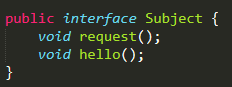
在SpringBoot中,启动类加上以下注解强制使用Cglib代理,该注解默认是false

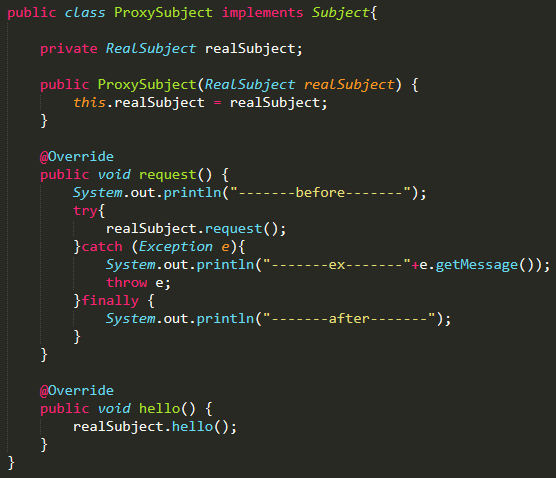
@EnableAspectJAutoProxy(proxyTargetClass = true)

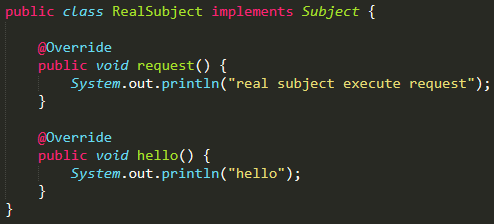
具体可以看Spring源码



### 2.1.3代理设计模式





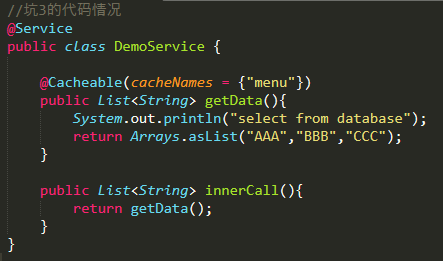


### 2.1.4 Spring AOP的一些坑

1.不宜把重要的业务逻辑放到aop中处理

2.无法拦截static、final方法、private方法

3.无法拦截内部方法调用



问题的解决方案及原因：



# 3.Java虚拟机

# 4.Spring

# 5.Mybatis

# 6.SpringBoot

# 7.SpringCloud

# 8.Redis

# 9.ElasticSearch

# 10.Zookeeper

## 10.1简介

Zookeeper是一个中间件,用于协调服务,作用于分布式系统,可以为大数据提供服务。支持Java和C语言的客户端Api。

**Zookeeper的特性：**

|  |  |
| --- | --- |
| **一致性** | 数据一致性,数据按照顺序分批入库。 |
| **原子性** | 事务要么成功,要么失败,是不会局部化的。 |
| **单一视图** | 客户端连接集群中的任一 zk节点,数据都是一致的。 |
| **可靠性** | 每次对zk的操作状态都会保存在服务端。 |
| **实时性** | 客户端可以读取到zk服务端的最新数据 |

**Zookeeper基本数据模型：**

* zk的数据模型也可以理解为linux/unix的文件目录: /usr/local/...
* 每一个节点都称之为znode ,它可以有子节点,也可以有数据。
* 每个节点分为临时节点和永久节点,临时节点在客户端断开后消失。
* 每个zk节点都各自的版本号,可以通过命令行来显示节点信息。
* 每当节点数据发生变化,那么该节点的版本号会累加(乐观锁)。
* 删除/修改过时节点,版本号不匹配则会报错。
* 每个zk节点存储的数据不宜过大,几K即可。
* 节点可以设置权限acl ,可以通过权限来限制用户的访问。

## 10.2 Zookeeper文件夹主要目录介绍

|  |  |
| --- | --- |
| **bin** | 主要的一些运行命令 |
| **conf** | 存放配置文件,其中我们需要修改zk.cfg |
| **contrib** | 附加的一些功能 |
| **dist-maven** | mvn编译后的目录 |
| **docs** | 文档 |
| **lib** | 需要依赖的jar包 |
| **recipes** | 案例demo代码 |
| **src** | 源码 |

## 10.3 Zookeeper配置文件介绍

clientPort

客户端连接server的端口,即对外服务端口,一般设置为2181。

dataDir

存储快照文件snapshot的目录。默认情况下,事务日志也会存储在这里。建议同时配置参数dataLogDir, 事务日志的写性能直接影响zk性能。

tickTime

ZK中的一个时间单元。ZK中所有时间都是以这个时间单元为基础,进行整数倍配置的。例如,session的最小超时时间是2\*tickTime。

dataLogDir

事务日志输出目录。尽量给事务日志的输出配置单独的磁盘或是挂载点,这将极大的提升ZK性能。

globalOutstandingLimit

最大请求堆积数。默认是1000。ZK运行的时候, 尽管server已经没有空闲来处理更多的客户端请求了,但是还是允许客户端将请求提交到服务器上来,以提高吞吐性能。当然,为了防止Server内存溢出,这个请求堆积数还是需要限制下的。

preAllocSize

预先开辟磁盘空间,用于后续写入事务日志。默认是64M,每个事务日志大小就是64M。如果ZK的快照频率较大的话,建议适当减小这个参数。

snapCount

每进行snapCount次事务日志输出后,触发一次快照(snapshot), 此时,ZK会生成一个snapshot.\*文件,同时创建一个新的事务日志文件log.\*。默认是100000.（真正的代码实现中,会进行一定的随机数处理,以避免所有服务器在同一时间进行快照而影响性能）

traceFile

用于记录所有请求的log,一般调试过程中可以使用,但是生产环境不建议使用,会严重影响性能。

maxClientCnxns

单个客户端与单台服务器之间的连接数的限制,是ip级别的,默认是60,如果设置为0,那么表明不作任何限制。请注意这个限制的使用范围,仅仅是单台客户端机器与单台ZK服务器之间的连接数限制,不是针对指定客户端IP,也不是ZK集群的连接数限制,也不是单台ZK对所有客户端的连接数限制。

clientPortAddress

对于多网卡的机器,可以为每个IP指定不同的监听端口。默认情况是所有IP都监听 clientPort 指定的端口。

minSessionTimeoutmaxSessionTimeout

Session超时时间限制,如果客户端设置的超时时间不在这个范围,那么会被强制设置为最大或最小时间。默认的Session超时时间是在2 \*  tickTime ~ 20 \* tickTime 这个范围

fsync.warningthresholdms

事务日志输出时,如果调用fsync方法超过指定的超时时间,那么会在日志中输出警告信息。默认是1000ms。(Java system property:  fsync.warningthresholdms )

autopurge.purgeInterval

在上文中已经提到,3.4.0及之后版本,ZK提供了自动清理事务日志和快照文件的功能,这个参数指定了清理频率,单位是小时,需要配置一个1或更大的整数,默认是0,表示不开启自动清理功能。(No Java system property)

autopurge.snapRetainCount

这个参数和上面的参数搭配使用,这个参数指定了需要保留的文件数目。默认是保留3个。

electionAlg

在之前的版本中,这个参数配置是允许我们选择leader选举算法,但是由于在以后的版本中,只会留下一种“TCP-based version of fast leader election”算法,所以这个参数目前看来没有用了,这里也不详细展开说了。

initLimit

Follower在启动过程中,会从Leader同步所有最新数据,然后确定自己能够对外服务的起始状态。Leader允许F在 initLimit 时间内完成这个工作。通常情况下,我们不用太在意这个参数的设置。如果ZK集群的数据量确实很大了,F在启动的时候,从Leader上同步数据的时间也会相应变长,因此在这种情况下,有必要适当调大这个参数了。(No Java system property)

syncLimit

在运行过程中,Leader负责与ZK集群中所有机器进行通信,例如通过一些心跳检测机制,来检测机器的存活状态。如果L发出心跳包在syncLimit之后,还没有从F那里收到响应,那么就认为这个F已经不在线了。注意：不要把这个参数设置得过大,否则可能会掩盖一些问题。

leaderServes

默认情况下,Leader是会接受客户端连接,并提供正常的读写服务。但是,如果你想让Leader专注于集群中机器的协调,那么可以将这个参数设置为no,这样一来,会大大提高写操作的性能。

server.x=[hostname]:nnnnn[:nnnnn]

这里的x是一个数字,与myid文件中的id是一致的。右边可以配置两个端口,第一个端口用于F和L之间的数据同步和其它通信,第二个端口用于Leader选举过程中投票通信。

group.x=nnnnn[:nnnnn]weight.x=nnnnn

对机器分组和权重设置,可以  参见这里(No Java system property)

cnxTimeout

Leader选举过程中,打开一次连接的超时时间,默认是5s

zookeeper.DigestAuthenticationProvider.superDigest

ZK权限设置相关

skipACL

对所有客户端请求都不作ACL检查。如果之前节点上设置有权限限制,一旦服务器上打开这个开头,那么也将失效。

forceSync

这个参数确定了是否需要在事务日志提交的时候调用 FileChannel .force来保证数据完全同步到磁盘。

jute.maxbuffer

每个节点最大数据量,是默认是1M。这个限制必须在server和client端都进行设置才会生效。

## 10.4 Zookeeper的作用

1.master节点选举,主节点挂了以后,从节点就会接手工作并且保证这个节点是唯一的,这也是所谓首脑模式,从而保证我们的集群是高可用的。

2.统一配置文件管理,即只需要部署一台服务器,则可以把相同的配置文件同步更新到其他所有服务器,此操作在云计算中用的特别多(假设修改了redis统一配置)。

3.发布与订阅,类似消息队列MQ ( amq , rmq... ),dubbo发布者把数据存在znode上,订阅者会读取这个数据。

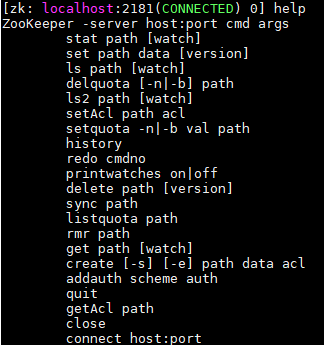
4.提供分布环境中不同进程之间争夺资源,类似于多线程中的锁。

5.集群管理,集群中保证数据的强一致性。

## 10.5 Zookeeper命令

**help命令**

help命令用于查询客服端所支持的所用的命令,执行help,输入如下：



**connect命令**

连接zk服务端,与close命令配合使用可以连接或者断开zk服务端

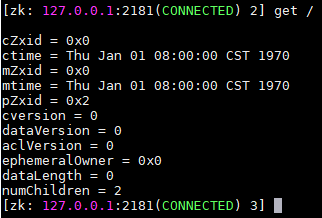
***connect 127.0.0.1:2181***

**close命令**

　　close命令用于关闭与服务端的链接

**get命令**

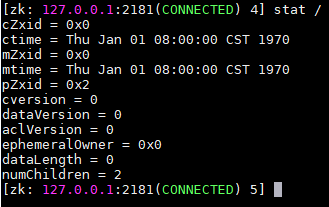
　　get命令用于获取节点的信息,注意节点的路径必须是以/开头的绝对路径。如get /



|  |  |
| --- | --- |
| cZxid | 节点创建时的zxid |
| ctime | 节点创建时间 |
| mZxid | 节点最近一次更新时的zxid |
| mtime | 节点最近一次更新的时间 |
| cversion | 子节点数据更新次数 |
| dataVersion | 本节点数据更新次数 |
| aclVersion | 节点ACL(授权信息)的更新次数 |
| ephemeralOwner | 如果该节点为临时节点,ephemeralOwner值表示与该节点绑定的session id. 如果该节点不是临时节点,ephemeralOwner值为0 |
| dataLength | 节点数据长度,本例中为hello world的长度 |
| numChildren | 子节点个数 |

**stat命令**

stat命令用于查看节点的状态信息,如stat /



该命令的结果参数说明同get命令

**set命令**

　　set命令用于设置节点的数据,如：

***set /usergrid hellUsergrid***

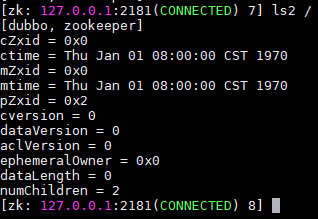
**ls命令**

ls命令用于获取路径下的节点信息,注意路径为绝对路径,如:



**ls2命令**

ls2命令是ls命令的增强版,比ls命令多输出本节点信息,如：



**listquota命令**

　　listquota命令用于显示配额,如listquota /storm

absolute path is /zookeeper/quota/storm/zookeeper\_limits

quota for /storm does not exist.

**setquota命令**

　　setquota命令用于设置节点个数以及数据长度的配额,如：

setquota -n 4 /zookeeper/node 设置/zookeeper/node子节点个数最大为4

setquota -b 100 /zookeeper/node 设置/zookeeper/node节点长度最大为100

**delquota命令**

　 delquota命令用于删除配额,

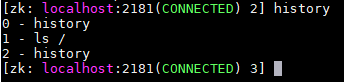
-n为子节点个数,

-b为节点数据长度,

如：delquota -n 2

**history命令**

　　history用于列出最近的命令历史,可以和redo配合使用。如history



**redo命令**

　　redo命令用于再次执行某个命令,使用方式为redo cmdid 如 redo 20

　　常与history配合使用

**create命令**

create命令用于创建节点,其中

-s为顺序充点,

-e临时节点

create /zookeeper/node1 "test\_create" world:anyone:fdsfds

**delete命令**

　　delete命令用于删除节点,如delete /nodeDelete

**addauth命令**

　　addauth命令用于节点认证,使用方式：如addauth digest username:password

**setAcl命令**

　　setAcl命令用于设置节点Acl

Acl由三部分构成：

1为scheme

2为user

3为permission

一般情况下表示为scheme:id:permissions

**getAcl命令**

　　获取节点的Acl,如getAcl /node1

scheme和id

|  |  |
| --- | --- |
| **world** | 它下面只有一个id, 叫anyone, world:anyone代表任何人,zookeeper中对所有人有权限的结点就是属于world:anyone的 |
| **auth** | 它不需要id, 只要是通过authentication的user都有权限(zookeeper支持通过kerberos来进行authencation, 也支持username/password形式的authentication) |
| **digest** | 它对应的id为username:BASE64(SHA1(password)),它需要先通过username:password形式的authentication |
| **ip** | 它对应的id为客户机的IP地址,设置的时候可以设置一个ip段,比如ip:192.168.1.0/16, 表示匹配前16个bit的IP段 |
| **super** | 在这种scheme情况下,对应的id拥有超级权限,可以做任何事情(cdrwa) |

**Permissions**

|  |  |
| --- | --- |
| **CREATE (c)** | 创建权限,可以在在当前node下创建child node |
| **DELETE (d)** | 删除权限,可以删除当前的node |
| **READ (r)** | 读权限,可以获取当前node的数据,可以list当前node所有的child nodes |
| **WRITE (w)** | 写权限,可以向当前node写数据 |
| **ADMIN (a)** | 管理权限,可以设置当前node的permission |

**sync命令**

　　sync命令用于强制同步,由于请求在半数以上的zk server上生效就表示此请求生效,那么就会有一些zk server上的数据是旧的。sync命令就是强制同步所有的更新操作。

**printwatchers命令**

　　printWatchers命令用于设置和显示监视状态,值为on或则off

**quit命令**

　　退出客户端

## 10.6 Session的基本原理

客户端和服务段之间的连接存在会话,每一个会话可以设置一个超时时间,心跳结束,session则过期。Session过期,临时节点会被抛弃。

心跳机制：客户端向服务端的ping包请求。

## 10.7 watcher机制

1.针对每个节点的操作,都会有一个监督者 -> wathcer。

2.当监控的某个对象( znode )发生了变化,则触发watcher事件。

3.zk中的watcher是**一次性**的,发后立即销毁。

4.父节点,子节点增删改都能够触发其watcher

5.针对不同类型的操作,触发的watcher事件也不同:

1. (子)节点创建事件

2. (子)节点删除事件

3. (子)节点数据变化事件

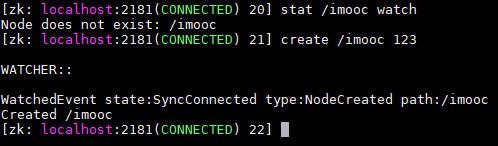
通过get path [watch]设置watcher

父节点增删改操作触发watcher

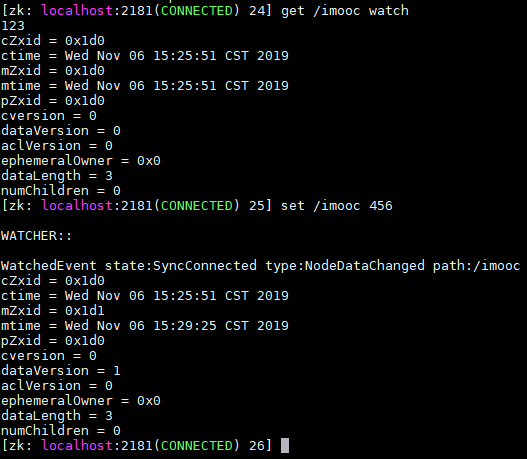
子节点增删改操作触发watcher

### 10.7.1 Watcher事件类型

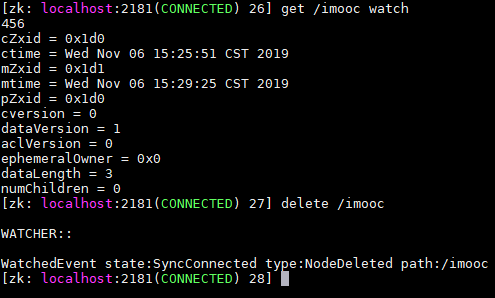
创建**父节点**触发: **NodeCreated**



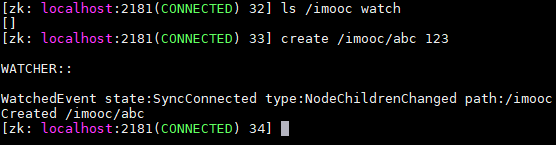
修改**父节点**数据触发: **NodeDataChanged**



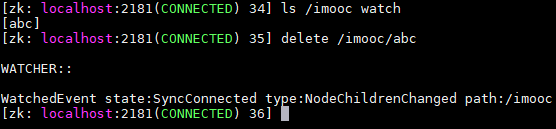
删除**父节点**触发: **NodeDeleted**



Is为父节点设置watcher ,创建子节点触发:**NodeChildrenChanged**

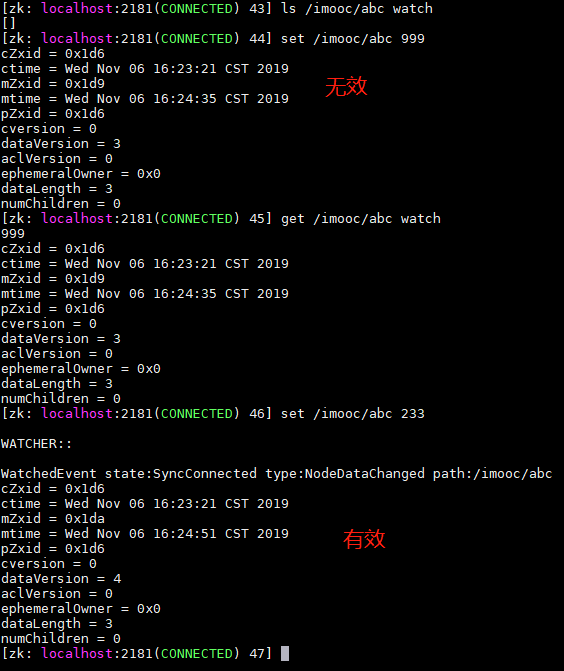


Is为父节点设置watcher ,删除子节点触发: **NodeChildrenChanged**



Is为父节点设置watcher ,修改子节点不触发事件.

这里是需要把子节点当作父节点来使用



Watcher使用场景：

1. 统一资源配置

## 10.8 Acl权限控制

ACL：Access Control List 访问控制列表,针对节点可以设置相关读写等权限,目的为了保障数据安全性,权限permissions可以指定不同的权限范围以及角色。

Acl权限控制是使用***Schema****：****ID****：****Permission***来控制

权限模式（**Scheme**）：授权的策略

授权对象（**ID**） : 授权的对象

权限（**Permission**） : 授予的权限

ZooKeeper的权限控制是基于每个znode节点的,需要对每个节点设置权限。每个znode支持设置多种权限控制方案和多个权限。子节点不会继承父节点的权限,客户端无权访问某节点,但可能可以访问它的子节点。

**schema采用何种方式授权**

|  |  |
| --- | --- |
| **world** | 默认方式,相当于全部都能访问 |
| **auth** | 代表已经认证通过的用户(cli中可以通过addauth digest user:pwd 来添加当前上下文中的授权用户) |
| **digest** | 即用户名:密码这种方式认证,这也是业务系统中最常用的。用 username:password 字符串来产生一个MD5串,然后该串被用来作为ACL ID。认证是通过明文发送 |
| **username** | password 来进行的,当用在ACL时,表达式为username:base64 ,base64是**password**的SHA1摘要的编码。 |
| **ip** | 使用客户端的主机IP作为ACL ID 。这个ACL表达式的格式为addr/bits ,此时addr中的有效位与客户端addr中的有效位进行比对。 |

**ID给谁授予权限**

授权对象ID是指,权限赋予的用户或者一个实体,例如：IP 地址或者机器。授权模式 schema与授权对象ID之间。

|  |  |
| --- | --- |
| **world** | 它下面只有一个id, 叫anyone, world:anyone代表任何人,zookeeper中对所有人有权限的结点就是属于world:anyone的 |
| **auth** | 它不需要id, 只要是通过authentication的user都有权限(zookeeper支持通过kerberos来进行authencation, 也支持username/password形式的authentication) |
| **digest** | 它对应的id为username:BASE64(SHA1(password)),它需要先通过username:password形式的authentication |
| **ip** | 它对应的id为客户机的IP地址,设置的时候可以设置一个ip段,比如ip:192.168.1.0/16, 表示匹配前16个bit的IP段 |
| **super** | 在这种scheme情况下,对应的id拥有超级权限,可以做任何事情(cdrwa) |

**permission授予什么权限**

CREATE、READ、WRITE、DELETE、ADMIN 也就是 增、删、改、查、管理权限,

这5种权限简写为crwda

delete是指对子节点的删除权限,其它4种权限指对自身节点的操作权限

|  |  |  |
| --- | --- | --- |
| CREATE | C | 可以创建子节点 |
| DELETE | D | 可以删除子节点（仅下一级节点） |
| READ | R | 可以读取节点数据及子节点列表 |
| WRITE | W | 可以设置节点数据 |
| ADMIN | A | 可以设置节点访问控制列表权限 |

| **四字命令** | **功能描述** |
| --- | --- |
| conf | 打印出服务相关配置的详细信息。 |
| cons | 列出所有连接到这台服务器的客户端全部连接/会话详细信息。包括"接受/发送"的包数量、会话id、操作延迟、最后的操作执行等等信息。 |
| crst | 重置所有连接的连接和会话统计信息。 |
| dump | 列出那些比较重要的会话和临时节点。这个命令只能在leader节点上有用。 |
| envi | 打印出服务环境的详细信息。 |
| reqs | 列出未经处理的请求 |
| ruok | 测试服务是否处于正确状态。如果确实如此,那么服务返回"imok",否则不做任何相应。 |
| stat | 输出关于性能和连接的客户端的列表。 |
| srst | 重置服务器的统计。 |
| srvr | 列出连接服务器的详细信息 |
| wchs | 列出服务器watch的详细信息。 |
| wchc | 通过session列出服务器watch的详细信息,它的输出是一个与watch相关的会话的列表。 |
| wchp | 通过路径列出服务器watch的详细信息。它输出一个与session相关的路径。 |
| mntr | 输出可用于检测集群健康状态的变量列表 |

# 11.Dobblo

# 12.MySQL

## 12.1索引

索引是一种用排序和协助查询的数据结构,通常的使用**B-Tree**或**B+Tree**实现。索引又分聚簇索引和非聚簇索引。

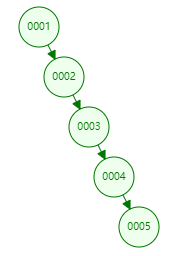
**聚簇索引和非聚簇索引的区别**：索引和数据是否在一起,在则是聚簇,反之非聚簇。因为聚簇索引的特性,所以查询速度肯定是比非聚簇索引快的。

像Innodb是聚簇索引,MyIsam是非聚簇索引。

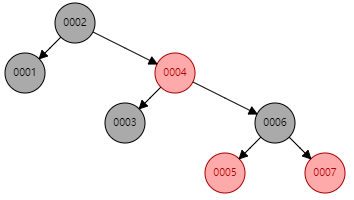
### 12.1.1索引的实现原理

索引可以用二叉树,红黑树,Hash,B-Tree或B+Tree实现。MySQL之所以选择用B-Tree或B+Tree来实现,是因为其他有一定的局限性。

**二叉树**来说,它不是自平衡树,在一定情况下会形成一种“链表”的结构,比如自增的情况下。查询效率会极大的下降,变成了遍历全部的数据。这显然不适合做索引的。

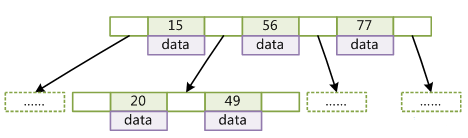


**红黑树**,是自平衡树的二叉树,解决的二叉树在一些情况下变成“链表”的问题,但是为什么不选择它来实现索引呢？这是因为红黑树在数据量增长的情况下,树的深度也会递增的。此时,查询的次数也会增加,如图深度是3。MySQL想控制深度在一定的范围,因此红黑树就不适合用来实现索引了。



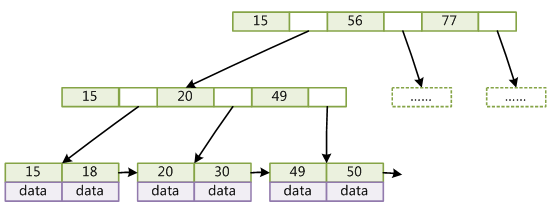
**B-Tree**,自然是为了解决红黑树的局限性。通过扩展节点存储的数量来保证树的深度。二叉树,红黑树节点存储的是一个元素,而B-Tree存储的一组数据。因此在数量增长的情况下,依然能保证树的深度了。下面的B-Tree的特性：

* 叶节点具有相同的深度,叶节点的指针为空
* 所有索引元素不重复
* 节点中的数据索引从左到右递增排列



**B+Tree**,是B-Tree的变种,MySQL还增加了顺序访问指针,如图所示。相比于B-Tree,B+Tree树更适合做索引。非子叶节点之存储key不存储数据,此外还有顺序访问指针（MySQL实现的）,很方便做范围查询。

* 非叶子节点不存储data,只存储索引(冗余),可以放更多的索引
* 叶子节点包含所有索引字段
* 叶子节点用指针连接,提高区问访问的性能



**Hash索引**,就通过hash函数处理索引字段,形成key-value存储。查询的时候,通过hash函数处理一次就可以查询到对应的数据了,可以说,效率是非常高的。但是,为什么不怎么用呢？是因为Hash索引无法做范围查询,模糊查询。这么一来Hash索引的适用场景就比较少了。得益于其查询的高效性,还是有一定的适用场景的。

采用Innodb存储引擎的时候,必须要有主键,这是因为MySQL实现索引是采用B+Tree的。即使没有,MySQL也会建立一个主键的,这个主键我们是看不到的。此外,还推荐主键是整形,自增的。首先,整型的比较肯定快于字符串的,此外存储容量也小很多。还有,自增的情况下,B+Tree不会出现分裂重构树的情况,减少不必要的维护。

# 13.MyCat

# 14.RocketMQ

分布式事务

[https://www.infoq.cn/article/2018%2F08%2Frocketmq-4.3-release](https://www.infoq.cn/article/2018/08/rocketmq-4.3-release)

# 15.Linux

# 16.设计模式

# 17.分布式锁

# 18.事务

事务提供一种机制将一个活动涉及的所有操作纳入到一个不可分割的执行单元,组成事务的所有操作只有在所有操作均能正常执行的情况下方能提交,只要其中任一操作执行失败,都将导致整个事务的回滚。简单地说,事务提供一种 "要么什么都不做,要么做全套（All or Nothing）" 机制。

## 18.1数据库本地事务：ACID

ACID是四个单词首字母的组合,即使Atomicity（原子性）,Consistency（一致性）,Isolation（隔离性）和Durability（持久性）。

**Atomicity（原子性）**

要么全部完成,要么全部不完成。

**Consistency（一致性）**

事务执行成功,所有变化（一般指数据）均正确操作。

**Isolation（隔离性）**

在并发环境下,不同事务操作相同数据,每个事务都有各自完整的数据空间。 事务不会查看到中间数据。

**Durability（持久性）**

事务只要成功结束,那么对数据库数据的更新便会永久保存下来。

## 18.2 InnoDB 实现原理

InnoDB是MySQL的一种存储引擎,它的实现是基于InnoDB的日志和锁来保证的。其中,**隔离性**是通过锁来保证的,**原子性**和**一致性**是**Undolog**来保证的,而**持久性**是通过**RedLog**保证的。

**UndoLog**：在操作任何数据前,将数据备份到一个地方（数据存储备份的地方叫做 UndoLog）。然后进行修改,如果事务执行失败了,那么系统可以利用备份数据恢复到事务执行前的状态。

**RedoLog**：记录新数据的备份,在事务提交前,只要将 RedoLog 持久化即可,不需要将数据持久化。当系统崩溃时,系统会根据 RedoLog 的内容,将数据更新至最新状态。

## 18.3分布式事务

数据库本地事务这个好控制,但是分布式事务就复杂多了,它是由多个操作组成,而这些操作不是基于一个数据库进行的,这便是它的难点。要么都全部成功,要么都全部失败。既All or Nothting。

分布式事务需求由来是因为微服务架构的兴起,兴趣的根本原因是单机无法处理大的数据量了,需要进行分库分表,分布在不同的机器一起进行。而事务这块是必不可少的,因而带来分布式事务的问题。

### 18.3.1分布式的理论CAP

CAP定律,又成为布鲁尔定理,CAP来自三个英文单词首字母的组合,既

**C** Consistency （一致性）

*所有数据备份,在同一时刻是否同样的值*

**A** Availability（可用性）

*集群中的部分节点出现故障,集群整体是否还能相应客户端读写*

**P** Partition tolerance（分区容错性）

*分区相当于对通信的时限要求,系统如果不能在指定时限内达成数据一致性,就意味着发生了分区,需要在 C 与 A 之间选择。*

CAP不是三剑客,三者不能共存,最多保证两种,其中P分区容错性是一样要保证的。这一点是因为无法100%的保证网络可靠,所以分区容错性必要得保证的。

CAP最多保证两种,因此有3套方案,如下：

1. CA 理论上不可能存在。由于网络无法 100% 可靠,故分布式系统中必定会有分区。分区发生时,为了保证 C,这个时候一定要拒绝请求,但是 A 不允许。

2. CP 放弃可用性,追求强一致

3. AP 放弃一致性（强一致性）,追求分区容错性和可用性。BASE 也是根据 AP 的拓展。

### 18.3.2 BASE理论

BASE其实讲的三点,如下：

1.Basically Available(基本可用)

*分布式系统在出现故障时,允许损失部分可用功能,保证核心功能可用。*

2.Soft State(软状态)

*允许系统中存在中间状态,这个状态不影响系统可用性,这里指的是CAP中的不一致。*

3.Eventually Consistent(最终一致性)

*最终一致是指经过一段时间后,所有节点数据都将会达到一致。*

上面讲过,CASE理论是CAP中AP方案的扩展。BASE解决了CAP中理论没有网络延迟,在BASE中用软状态和最终一致,保证了延迟后的一致性。BASE和ACID是相反的,它完全不同于ACID的强一致性模型,而是通过牺牲强一致性来获得可用性,并允许数据在一段时间内是不一致的,但最终达到一致状态。

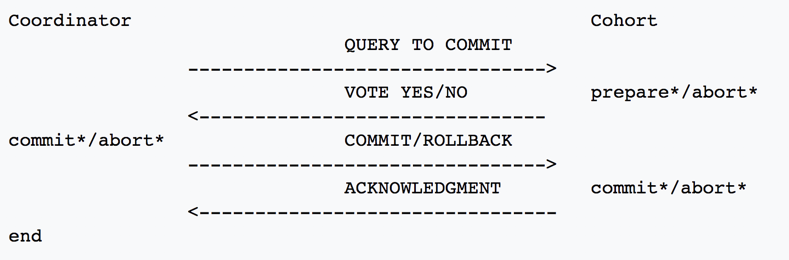
## 18.4分布式事务的解决方案

解决分布式事务,目前了解到的有4种,分别是*2PC/3PC,TCC,MQ事务,SAGA事务*。下面一一讲解这4种方案。

### 18.4.1 2PC/3PC

2PC/3PC其实是两个不同的的方案,2PC既二阶段提交,3PC自然就是3阶段提交了,这里合在一起讲解,因为3PC是2PC扩展来的。

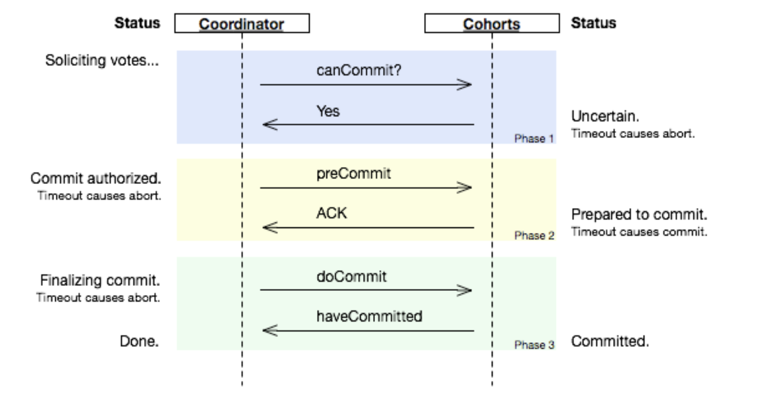
先来讲解2PC二阶段提交。2PC 把事务的执行分为两个阶段,如图所示。



**第一个阶**段即 prepare阶段,这个阶段实际上就是投票阶段,协调者向参与者确认是否可以共同提交。

**第二阶段**是得到全部参与者的所有回答后,协调者向所有的参与者发布共同提交或者共同回滚的指令,用以保证事务达到一致性。

但是分布式系统中的所有通信均存在着三种状态：***成功****,****失败****,****超时***。其中,超时状态的存在是我们在设计分布式系统时所面对的永远的痛,2PC同样存在问题,尤其是在发送完可以提交的指令后,参与者在没有收到提交或者回滚的指令时,面对已经上锁的资源,面对已经写出去的undo或者redo日志,参与者会一时陷入手足无措的状态,为了解决这个问题,3PC 应运而生,如下图所示。



3PC在commit之前增加了**preCommit**的过程,使得在参与者在收不到确认时,依然可以从容commit或者rollback,避免资源锁定太久导致浪费。但是3PC同样存在着很多问题。实现起来非常复杂,因为很难通过多次询问来解决系统间分歧问题,尤其是存在超时状态互不信任的分布式网络中。

优点：在于已经有较为成熟的实现方案,比如 XA。

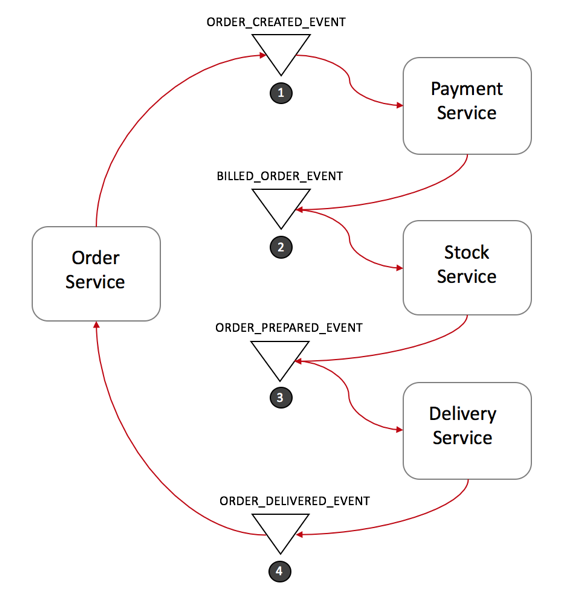
缺点：XA是一个阻塞协议。服务在投票后需要等待协调器的决定,此时服务会阻塞并锁定资源。由于其阻塞机制和最差时间复杂度高,因此,这种设计不能适应随着事务涉及的服务数量增加而扩展的需要,很难用于并发较高以及子事务声明周期较长 (long-running transactions) 的分布式服务中。

### 18.4.2 SAGA事务

SAGA算法是一种异步的分布式事务解决方案,假设所有事件按照顺序推进,总能达到系统的最终一致性,因此SAGA需要服务分别定义提交接口以及补偿接口,当某个事务分支失败时,调用其它的分支的补偿接口来进行回滚。

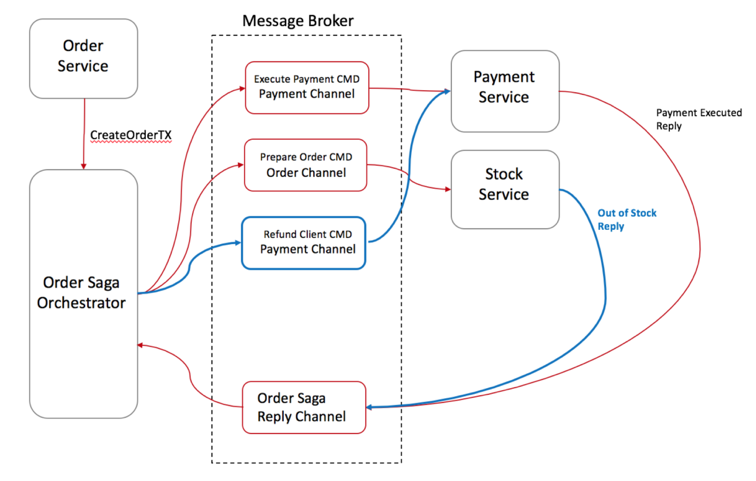
SAGA的具体实现分为两种：Choreography以及Orchestration。

**Choreography**这种模式下不存在协调器的概念,每个节点均对自己的上下游负责,在监听处理上游节点事件的同时,对下游节点发布事件。



**Orchestration**存在中心节点的模式,该中心节点,即协调器知道整个事务的分布状态,相比于无中心节点方式,该方式有着许多优点：

* 能够避免事务之间的循环依赖关系。
* 参与者只需要执行命令/回复(其实回复消息也是一种事件消息),降低参与者的复杂性。
* 开发测试门槛低。
* 在添加新步骤时,事务复杂性保持线性,回滚更容易管理。因此大多数saga模型实现均采用了这种思路。

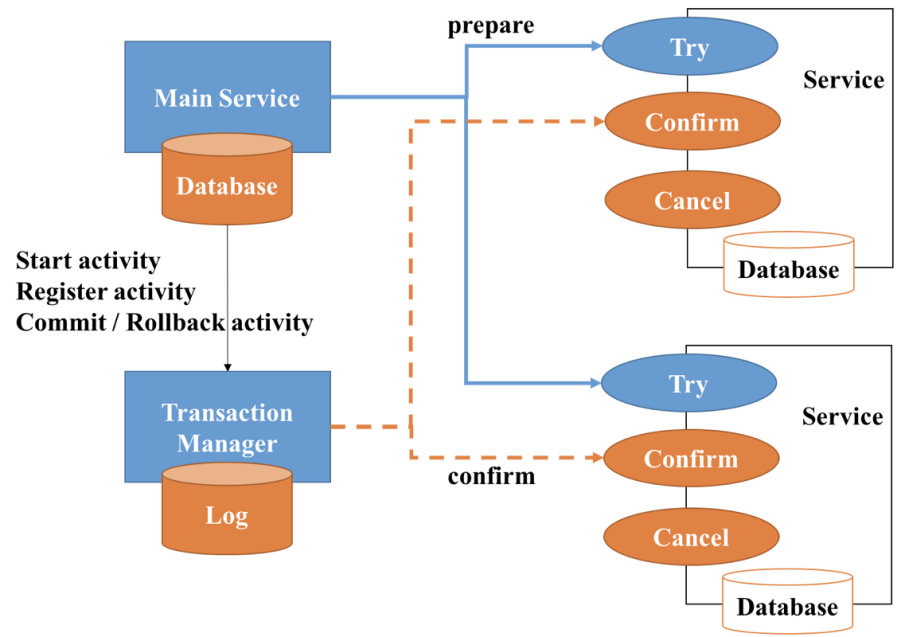


总结一下：SAGA模型的优点在于其降低了事务粒度,使得事务扩展更加容易,同时采用了异步化方式提升性能。但是其缺点在于很多时候很难定义补偿接口,回滚代价高,而且由于SAGA在执行过程中采用了先提交后补偿的思路进行操作,所以单个子事务在并发提交时的隔离性很难保证。

### 18.4.3 TCC

TCC(Try-Confirm-Concel) 模型是一种补偿性事务,主要分为三个阶段：

**Try**:检查、保留资源,**Confirm** :执行事务,**Concel**:释放资源,如图所示。



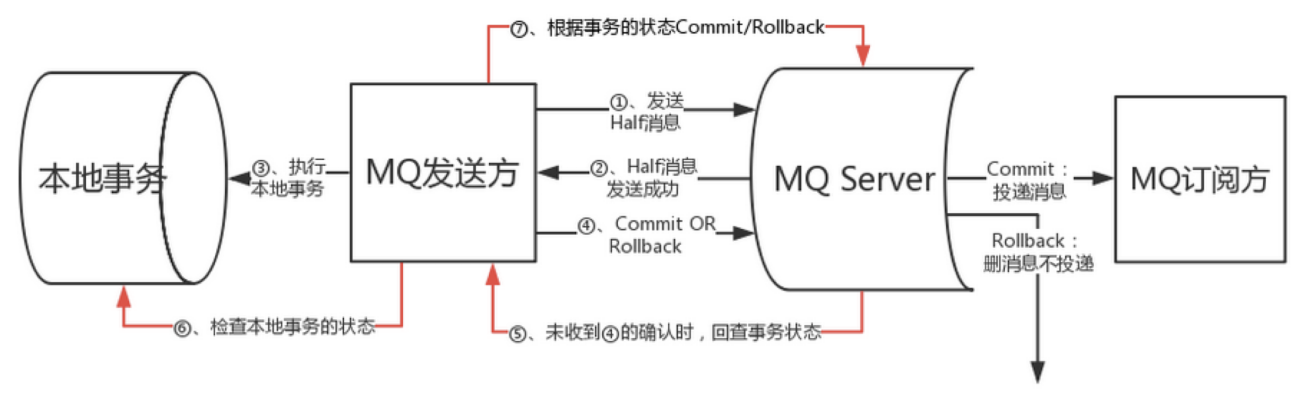
其中,活动管理器记录了全局事务的推进状态以及各子事务的执行状态,负责推进各个子事务共同进行提交或者回滚。同时负责在子事务处理超时后不停重试,重试不成功后转手工处理,用以保证事务的最终一致性。

总结一下,相比于SAGA模型,其优点在于尝试阶段仅仅只是对业务系统做检测,并保留业务资源,并没有真正提交,所以后续SAGA需要针对提交的事务做补偿,而TCC则仅仅需要释放保留资源,降低了补偿成本；并且,由于在Try阶段对资源进行了保留锁定,所以相比于SAGA模式,TCC模式拥有更高的隔离性。

缺点：相比于SAGA模式,TCC模式多增加了一个状态,导致在业务开发过程中,复杂度上升,而且协调器与子事务的通信过程增加,状态轮转处理也更为复杂。

### 18.4.4 MQ事务

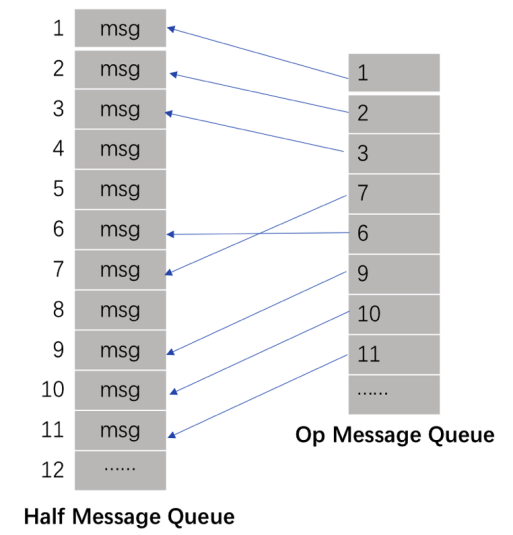
RocketMQ事务消息的设计流程同样借鉴了两阶段提交理论,如图所示：



1. 事务发起方首先发送prepare消息到MQ。
2. 在发送prepare消息成功后执行本地事务。
3. 根据本地事务执行结果返回commit或者是rollback。
4. 如果消息是rollback,MQ将删除该prepare消息不进行下发,如果是commit消息,MQ将会把这个消息发送给consumer端。
5. 如果执行本地事务过程中,执行端挂掉,或者超时,MQ将会不停的询问其同组的其它producer来获取状态。
6. Consumer端的消费成功机制有MQ保证

RocketMQ 事务消息在实现上充分利用了 RocketMQ 本身机制,在实现零依赖的基础上,同样实现了高性能、可扩展、全异步等一系列特性。

在具体实现上,RocketMQ 通过使用 Half Topic 以及 Operation Topic 两个内部队列来存储事务消息推进状态



Half Topic对应队列中存放着prepare消息,Operation Topic对应的队列则存放了prepare message对应的commit/rollback消息,消息体中则是prepare message对应的offset,服务端通过比对两个队列的差值来找到尚未提交的超时事务,进行回查。

# 19 Netty

## 19.1 IO模型

IO模型：BIO,NIO,AIO

**BIO**：

同步并阻塞(传统阻塞型),服务器实现模式为一个连接一个线程,即客户端有连接请求时服务器端就需要启动一个线程进行处理,如果这个连接不做任何事情会造成不必要的线程开销。

BIO 方式适用于连接数目比较小且固定的架构,这种方式对服务器资源要求比较高,并发局限于应用中,JDK1.4以前的唯一选择,但程序简单易理解。

**NIO**：

同步非阻塞,服务器实现模式为一个线程处理多个请求(连接),即客户端发送的连接请求都会注册到多路复用器上,多路复用器轮询到连接有 I/O 请求就进行处理。

NIO 方式适用于连接数目多且连接比较短（轻操作）的架构,比如聊天服务器,弹幕系统,服务器间通讯等。编程比较复杂,JDK1.4 开始支持。

**AIO**：

异步非阻塞,AIO 引入异步通道的概念,采用了Proactor模式,简化了程序编写,有效的请求才启动线程,它的特点是先由操作系统完成后才通知服务端程序启动线程去处理,一般适用于连接数较多且连接时间较长的应用。

AIO 方式使用于连接数目多且连接比较长（重操作）的架构,比如相册服务器,充分调用 OS 参与并发操作,编程比较复杂,JDK7 开始支持

## 19.2 BIO基本介绍

Java BIO就是传统的Java IO编程,其相关的类和接口在java.io包下

BIO(blocking I/O)：同步阻塞,服务器实现模式为**一个连接一个线程**,即客户端有连接请求时服务器端就需要启动一个线程进行处理,如果这个连接不做任何事情会造成不必要的线程开销,可以通过线程池机制改善(实现多个客户连接服务器)。

BIO 方式适用于连接数目比较小且固定的架构,这种方式对服务器资源要求比较高,并发局限于应用中,JDK1.4以前的唯一选择,程序简单易理解。

**BIO编程流程的梳理：**

1) 服务器端启动一个ServerSocket

2) 客户端启动 Socket对服务器进行通信,默认情况下服务器端需要对每个客户 建立一个线程与之通讯。

3) 客户端发出请求后, 先咨询服务器是否有线程响应,如果没有则会等待,或者被拒绝

4) 如果有响应,客户端线程会等待请求结束后,在继续执行

**Java BIO 问题分析：**

1) 每个请求都需要创建独立的线程,与对应的客户端进行数据 Read,业务处理,数据 Write 。

2) 当并发数较大时,需要创建大量线程来处理连接,系统资源占用较大。

3) 连接建立后,如果当前线程暂时没有数据可读,则线程就阻塞在 Read 操作上,造成线程资源浪费。

## 19.3 NIO介绍

Java NIO全称 java non-blocking IO,是指 JDK 提供的新 API。从 JDK1.4 开始,Java 提供了一系列改进的输入/输出的新特性,被统称为 NIO(即 New IO),是**同步非阻塞**的。NIO相关类都被放在**java.nio包及子包**下,并且对原java.io包中的很多类进行改写。

NIO三大核心部分：**Channel(通道),Buffer(缓冲区)**,**Selector(选择器)**

NIO 是**面向缓冲区**编程的。数据读取到一个它稍后处理的缓冲区,需要时可在缓冲区中前后移动,这就增加了处理过程中的灵活性,使用它可以提供非阻塞式的高伸缩性网络

Java NIO的非阻塞模式,使一个线程从某通道发送请求或者读取数据,但是它仅能得到目前可用的数据,如果目前没有数据可用时,就什么都不会获取,而不是保持线程阻塞,所以直至数据变的可以读取之前,该线程可以继续做其他的事情。 非阻塞写也是如此,一个线程请求写入一些数据到某通道,但不需要等待它完全写入,这个线程同时可以去做别的事情。

HTTP2.0 使用了多路复用的技术,做到同一个连接并发处理多个请求,而且并发请求的数量比 HTTP1.1 大了好几个数量级

### 19.3.1 NIO和BIO的比较

BIO以流的方式处理数据,NIO以块的方式处理数据,块I/O的效率比流I/O高很多。

BIO是阻塞的,NIO 则是非阻塞的

BIO基于字节流和字符流进行操作,而NIO 基于 Channel(通道)和 Buffer(缓冲区)进行操作,数据总是从通道读取到缓冲区中,或者从缓冲区写入到通道中。Selector(选择器)用于监听多个通道的事件（比如：连接请求,数据到达等）,因此使用单个线程就可以监听多个客户端通道。

### 19.3.2 NIO三大核心的关系

1)每个Channel都会对应一个 Buffer（Channel一对一Buffer）

2)Selector对应一个线程,一个线程对应多个Channel(连接)

3)程序切换到哪个Channel是有事件决定的, Event就是一个重要的概念

4)Selector 会根据不同的事件,在各个通道上切换

5)Buffer就是一个内存块,底层是有一个数组

6)数据的读取写入是通过Buffer, 这个和BIO不同,BIO中要么是输入流,或者是

输出流, 不能双向。但是 NIO 的 Buffer 是可以读也可以写, 需要flip方法切换

channel是双向的, 可以返回底层操作系统的情况, 比如Linux,底层的操作系统

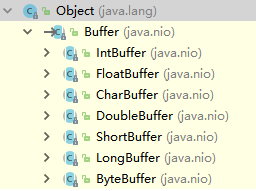
通道就是双向的。

**线程** <-**1**---**1**-> **Selector** <-**1**---**N**-> **Channel** <-**1**---**1**-> **Buffer**

### 19.3.3 Buffer(缓冲区)

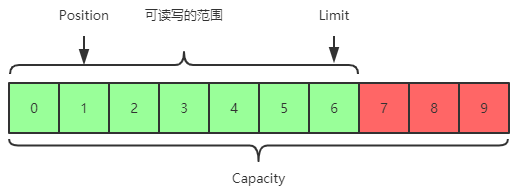
缓冲区本质上是一个**可以读写数据的内存块**,可以理解成是一个容器对象(含数组),该对象提供了一组方法,可以更轻松地使用内存块,缓冲区对象内置了一些机制,能够跟踪和记录缓冲区的状态变化。

Buffer的子类：（没有Boolean）



Buffer四个关键属性：

|  |  |
| --- | --- |
| Capacity | Buffer的容量,创建后不能修改 |
| Limit | Buffer读写的极限,该极限可读写,但读写不能超过该极限 |
| Position | 下一个读写的位置 |
| Mark | 标记 |



### 19.3.4 Channel(通道)

NIO的通道类似于流,但有区别：

通道可以同时进行读写,流只能读或者只能写

通道可以实现异步读写数据

通道可以从缓冲读数据,也可以写数据到缓冲:

BIO中的stream是单向的,NIO 中的通道(Channel)是双向的,可以读操作,也可以写操作。

Channel是一个接口

**public interface** Channel **extends** Closeable {}

常用的Channel类有：

FileChannel(用于文件的数据读写)

DatagramChannel(用于UDP的数据读写)

SocketChannel(用于TCP的数据读写)

ServerSocketChannel(用于TCP的数据读写)

### 19.3.5 Selector(选择器)

Selector能够检测多个注册的通道上是否有事件发生(多个Channel 以事件的方式可以注册到同一个Selector),如果有事件发生,便获取事件然后针对每个事件进行相应的处理。这样就可以只用一个单线程去管理多个通道,也就是管理多个连接和请求。

只有在连接通道真正有读写事件发生时,才会进行读写,就大大地减少了系统开销,并且不必为每个连接都创建一个线程,不用去维护多个线程。

避免了多线程之间的上下文切换导致的开销。

**Selector的特点**：

1)Netty的IO线程NioEventLoop聚合了Selector(选择器,也叫多路复用器),可以同时并发处理成百上千个客户端连接。

2)当线程从某客户端Socket通道进行读写数据时,若没有数据可用时,该线程可以进行其他任务。

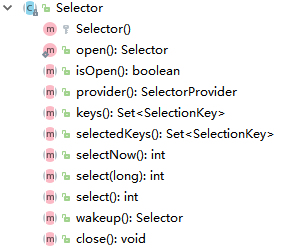
3)线程通常将非阻塞IO的空闲时间用于在其他通道上执行IO操作,所以单独的线程可以管理多个输入和输出通道。

4)由于读写操作都是非阻塞的,这就可以充分提升IO线程的运行效率,避免由于频繁I/O阻塞导致的线程挂起。

5)一个I/O线程可以并发处理N个客户端连接和读写操作,这从根本上解决了传统同步阻塞I/O一连接一线程模型,架构的性能、弹性伸缩能力和可靠性都得到了极大的提升。

Selector是一个抽象类

**public abstract class** Selector **implements** Closeable {}



**Selector相关方法说明**

selector.**select**() // 阻塞

selector.**select**(1000);// 阻塞 1000 毫秒,在 1000 毫秒后返回

selector.**wakeup**(); // 唤醒 selector

selector.**selectNow**(); // 不阻塞,立马返还

**SelectionKey**

SelectionKey,表示Selector和网络通道的注册关系:

int OP\_READ： 代表读操作,值为 1

int OP\_WRITE： 代表写操作,值为 4

int OP\_CONNECT：代表连接已经建立,值为 8

int OP\_ACCEPT： 有新的网络连接可以accept,值为 16

源码中：

**public static final int *OP\_READ*** = 1 << 0;

**public static final int *OP\_WRITE*** = 1 << 2;

**public static final int *OP\_CONNECT*** = 1 << 3;

**public static final int *OP\_ACCEPT*** = 1 << 4;

### 19.3.6 NIO与零拷贝

零拷贝基本介绍

1. 零拷贝是网络编程的关键,很多性能优化都离不开。
2. 在Java程序中,常用的零拷贝有 **mmap(内存映射)**和**sendFile**。
3. 零拷贝是从 操作系统的角度来说的。因为内核缓冲区之间,没有数据是重复的（只有kernel buffer 有一份数据）。
4. 零拷贝不仅仅带来更少的数据复制,还能带来其他的性能优势,例如更少的上下文切换,更少的CPU缓存伪共享以及无CPU校验和计算。
5. 提示：零拷贝从操作系统角度,是没有cpu拷贝。

**mmap 优化**

mmap通过内存映射,将文件映射到内核缓冲区,用户空间可以共享内核空间的数据。这样,在进行网络传输时,就可以减少内核空间到用户空间的拷贝次数。

**sendFile优化**

Linux 2.1 版本 提供sendFile 函数,其基本原理如下：

数据根本不经过用户态,直接从内核缓冲区进入到Socket Buffer,同时,由于和用户态完全无关,就减少了一次上下文切换

Linux在2.4 版本中,做了一些修改,避免了从内核缓冲区拷贝到Socket buffer的操作,直接拷贝到协议栈,从而再一次减少了数据拷贝。

这里其实有一次CPU拷贝kernel buffer -> socket buffer

但是,拷贝的信息很少,比如 lenght,offset,消耗低,可以忽略。

**mmap和sendFile的对比**

1)mmap适合小数据量读写，sendFile适合大文件传输。

2)mmap需要4次上下文切换，3次数据拷贝；sendFile需要3次上下文切换，最少2次数据拷贝。

3)sendFile可以利用DMA方式，减少CPU拷贝，mmap则不能（必须从内核拷贝到Socket缓冲区）。