记得录屏

# 分布式事务产生背景

[database transaction]

数据库事务要满足几个要求：ACID

Atomic(原子性) 事务必须是原子的工作单元

Consistent(一致性) 事务完成时，必须使所有数据都保持一致状态

Isolation(隔离性) 并发事务所做的修改必须和其他事务所做的修改是隔离的

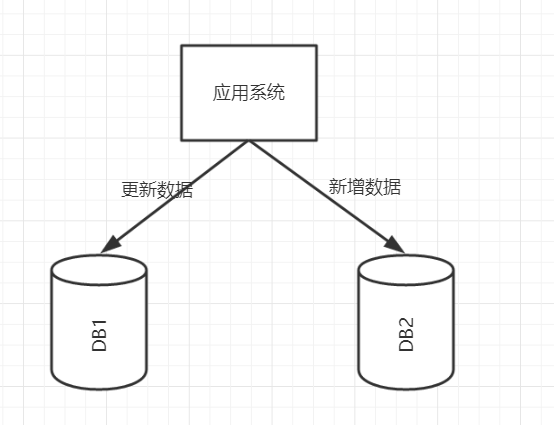
Duration（持久性） 事务完成之后，对系统的影响是永久性的

Mysql里的事务处理过程

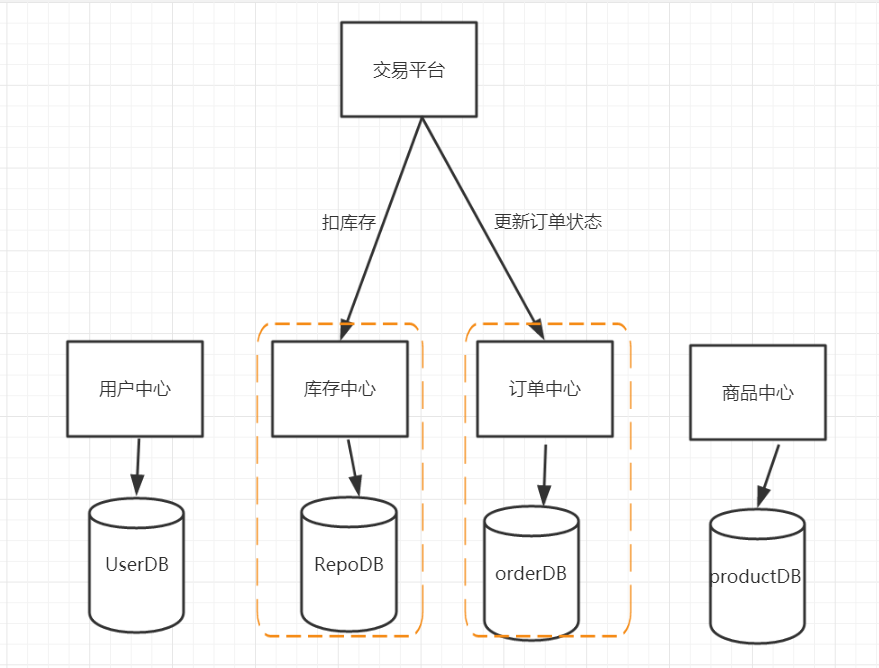
1. 记录redo和undo log文件，确保日志在磁盘上的持久化
2. 更新数据记录
3. 提交事务 ，redo 写入commit记录

# 分布式事务

## 数据库分库分表



## SOA化



# X/OpenDTP事务模型

X/Open Distributed Transaction Processing Reference Model

X/Open是一个组织机构，定义出的一套分布式事务标准， 定义了规范的API接口

2PC（two -phase-commit）, 用来保证分布式事务的完整性

J2EE 遵循了X/open DTP规范，设计并实现了java里面的分布式事务编程接口规范-JTA

XA是X/Open DTP定义的中间件与数据库之间的接口规范。 XA接口函数由数据库厂商提供

## X/OpenDTP 角色

AP application

RM resouces manager 资源管理器。 数据库

TM transaction manager 事务管理器，事务协调者

# 2PC（two -phase-commit）

(CAP)

## 阶段一：提交事务请求（投票）

1. TM向所有的AP发送事务内容，询问是否可以执行事务的提交操作，并等待各个AP的响应
2. 执行事务

各个AP节点执行事务操作，将undo和redo信息记录到事务日志中，尽量把提交过程中所消耗时间的操作和准备都提前完成后确保后续

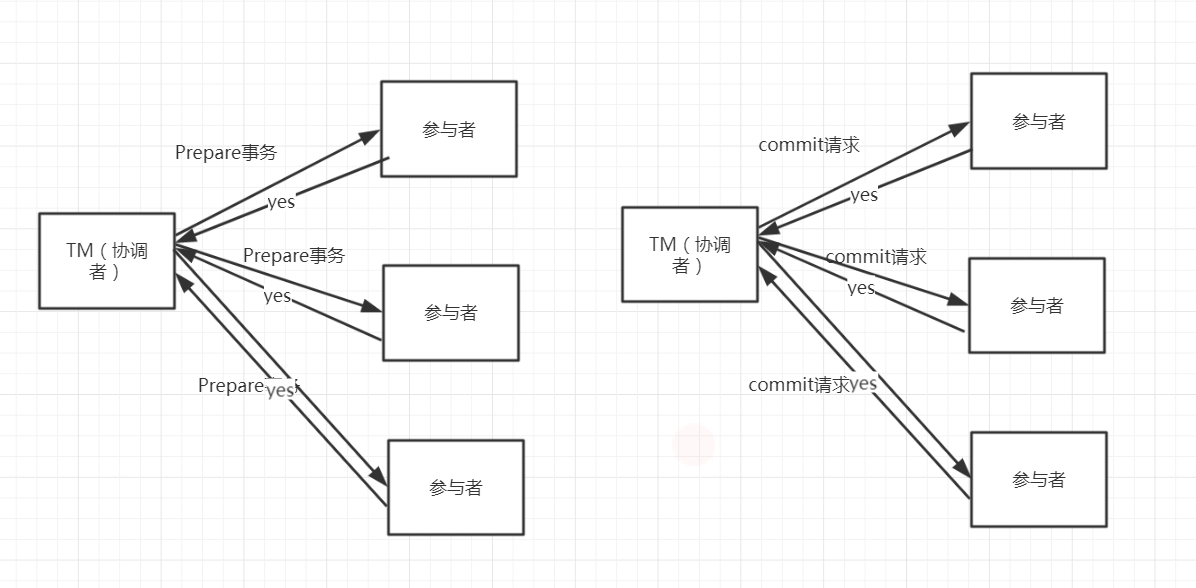
事务提交的成功率

1. 各个AP向TM反馈事务询问的响应

各个AP成功执行了事务操作，那么反馈给TM yes的response；如果AP没有成功执行事务，就反馈TM no的response

## 阶段二：执行事务提交

### 执行提交事务

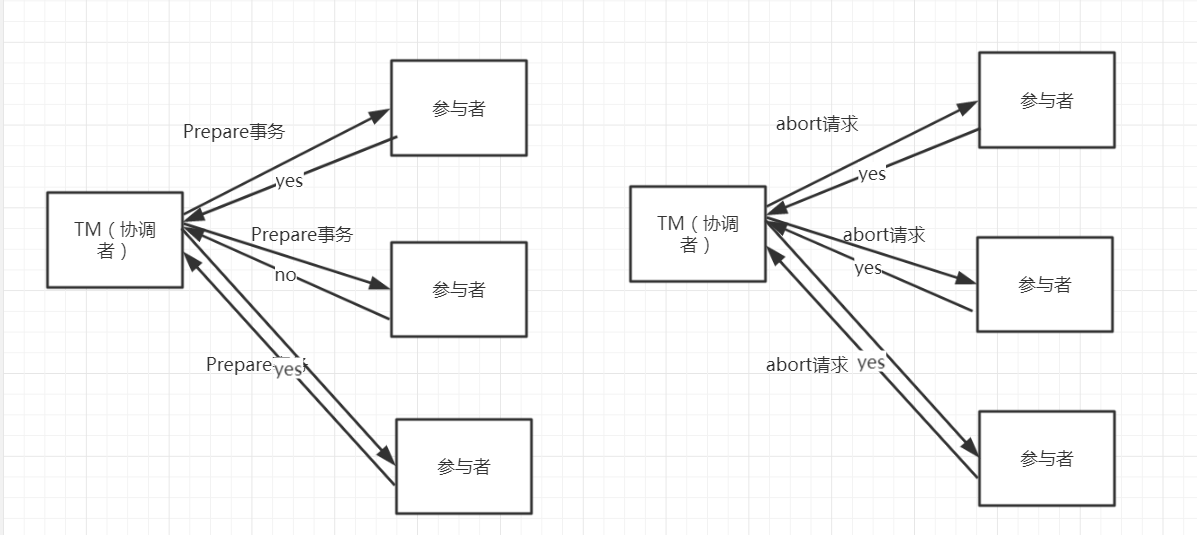


假设一个事务的提交过程总共需要30s， 其中prepare操作需要28（事务日志落地磁盘及各种io操作），而真正commit只需要2s

那么，commit阶段发生错误的概率和prepare相比， 2/28 (<10%) .只要第一个阶段成功，那么commit阶段出现失败的概率就非常小

大大增加了分布式事务的成功概率

### 中断事务提交



# 2pc存在的问题

1. 数据一致性问题
2. 同步阻塞

# 3PC(three phase commit)

阶段一：canCommit

阶段二：preCommit

阶段三：doCommit

# 分布式事务的实现

JOTM(java open transaction manager)

Atomikos

# 互联网的分布式事务的解决方案

# 知识点回顾

## 分布式事务产生的原因

* 1. 数据库分库分表
  2. 服务soa化
  3. *在分布式系统中，每一个机器节点虽然都能明确的知道自己执行的事务是成功还是失败，但是却无法知道其他分布式节点的事务执行情况。因此，当一个事务要跨越多个分布式节点的时候（比如，下单流程，下单系统和库存系统可能就是分别部署在不同的分布式节点中），为了保证该事务可以满足ACID，就要引入一个协调者（Cooradinator）。其他的节点被称为参与者（Participant）。协调者负责调度参与者的行为，并最终决定这些参与者是否要把事务进行提交。*

# X/OPEN DTP

AP application， 应用程序

RM resource manager ，资源管理，一般表示数据库，必须实现XA定义的接口

TM transaction manager 事务管理器，负责协调和事务管理

## 2pc提交协议

1. 提交事务请求（投票）
2. 执行事务请求（提交或中断）

## 2pc的数据一致性问题

1. 数据不一致。在二阶段提交的阶段二中，当协调者向参与者发送commit请求之后，发生了局部网络异常或者在发送commit请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了commit请求。而在这部分参与者接到commit请求之后就会执行commit操作。但是其他部分未接到commit请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据部一致性的现象。
2. 同步阻塞问题。执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态
3. 二阶段无法解决的问题：协调者在发出commit消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交
4. 单点故障。由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去

## 3pc提交协议

1. canCommit
2. preCommit
3. doCommit

改进点

1. 增加了超时机制
2. 第二阶段，如果协调者超时没有接受到参与者的反馈，则自动认为失败，发送abort命令
3. 第三阶段，如果参与者超时没有接受到协调者的反馈，则自动认为成功开始提交事务（基于概率）

## 3pc的问题

相对于2PC，3PC主要解决的单点故障问题，并减少阻塞，因为一旦参与者无法及时收到来自协调者的信息之后，他会默认执行commit。而不会一直持有事务资源并处于阻塞状态。但是这种机制也会导致数据一致性问题，因为，由于网络原因，协调者发送的abort响应没有及时被参与者接收到，那么参与者在等待超时之后执行了commit操作。这样就和其他接到abort命令并执行回滚的参与者之间存在数据不一致的情况。

# XA/JTA

XA 就是 X/Open DTP 定义的事务管理器与资源管理器的接口规范（即接口函数），XA 接口函数由数据库厂商提供。

JTA是基于X/Open DTP模型开发的java transaction APi规范

# 概述

通过2pc的方式去完成分布式事务，虽然通过这种方式能够达到预期的效果，但是我们在现实中很少会用到2pc方式的提交的XA事务，有几个原因

1. 互联网电商应用的快速发展，对事务和数据的绝对一致性要求并没有传统企业应用那么高
2. XA事务的介入增加了TM中间件，使得系统复杂化
3. XA事务的性能不高，因为TM要等待RM回应，所以为了确保事务尽量成功提交，等待超时的时间通常比较长，比如30s到几分钟，如果RM出现故障或者响应比较慢，则整个事务的性能严重下降

# 互联网的分布式事务解决方案

目前互联网领域里有几种流行的分布式解决方案，但都没有像之前所说的XA事务一样形成X/OpenDTP那样的工业规范，而是仅仅在具体的行业里获得较多的认可

# 业务接口整合，避免分布式事务

这个方案就是把一个业务流程中需要在一个事务里执行的多个相关业务接口包装整合到一个事务中，比如我们可以讲A/B/C整合为一个服务D来实现单一事务的业务流程服务

# 最终一致性方案之ebay模式

eBay在2008年公布了一个关于BASE准则提到一个分布式事务解决方案。eBay的方案其实是一个最终一致性方案，它主要采用消息队列来辅助实现事务控制流程，方案的核心是将需要分布式处理的任务通过消息队列的方式来异步执行，如果事务失败，则可以发起人工重试的纠正流程。人工重试被更多的应用于支付场景，通过对账系统对事后问题进行处理

比如一个很常见的场景：某个用户产生了一笔交易，那么需要在交易表中增加记录，同时需要修改用户表的金额（余额），由于这两个表属于不同的远程服务，所以就会涉及到分布式事务与数据一致性的问题

|  |  |
| --- | --- |
| user(id, name, amt\_sold, amt\_bought)  transaction(xid, seller\_id, buyer\_id, amount) | begin;  INSERT INTO transaction VALUES(xid, $seller\_id, $buyer\_id, $amount);  UPDATE user SET amt\_sold = amt\_sold + $amount WHERE id = $seller\_id;  UPDATE user SET amt\_bought = amt\_bought + $amount WHERE id = $buyer\_id;  commit; |

那么在这里可以使用消息队列（MQ）来做

先启动一个事务，更新交易表（transaction）后，并不直接更新user表，而是将要对user表进行的更新插入到消息队列中。

目标系统收到该消息以后，启动本地事务去对用户表的余额做调整

伪代码

bool result=dao.update();

if(result){

mq.send();

}

根据上面的伪代码的实现方案，可能出现几种情况

1. 数据库操作成功，向MQ中投递消息也成功
2. 操作数据库失败，不会向MQ中投递消息
3. 操作数据库成功，但是向MQ中投递消息时失败，向外抛出异常。数据库操作回滚

对于上面几种情况，问题都不大。那么我们分析小消费端的问题

1. 消息出队列以后，消费者对应的业务操作要执行成功。如果执行失败，消息不能失效或者丢失。需要保证消息和业务操作一致
2. 尽量避免消息重复消费，如果重复消费，也不能影响业务的执行结果

对于第一个问题，如何保证消息不丢失

现在用的比较普遍的MQ都具有持久化消息的功能，如果消费者宕机或者消费失败，都可以执行重试机制

对于如何避免消息的重复消费

1. 保证消费者的幂等性；**也就是说如果队列中的消息因为网络异常导致发送多次的情况下，仍然需要保证消息被应用多次与应用一次产生的效果是一样的**
2. 通过消费日志表来记录消费状态；**增加一个message\_applied（msg\_id）表，用来记录已经被成功应用的消息。在目标系统执行更新操作之前，先检测该消息是否已经被消费过，消费完成后通过本地事务控制来更新这个“消费表状态”，用来避免消息重复消费问题**

上面这种方式是非常经典的实现，基本避免了分布式事务，实现了“最终一致性”。

*各大知名的电商平台和互联网公司，几乎都是采用类似的设计思路来实现“最终一致性”的。这种方式适合的业务场景广泛，而且比较可靠。不过这种方式技术实现的难度比较大*

## 保证最终一致性的模式

1. 查询模式

任何一个服务操作都提供一个查询接口，用来向外部输出操作执行的状态。服务操作的使用方可以通过接口得知服务操作执行的状态，然后根据不同状态做不同的处理操作

为了能够实现查询，每个服务操作都需要有唯一的流水号

1. 补偿模式

有了查询模式，我们就能够得知操作所处的具体状态，如果整个操作处于不正常状态，我们需要修正操作中的出现问题的子操作。也许是要重新执行，或者取消已完成的操作。通过修复使得整个分布式系统达到最终一致。这个过程就是补偿模式

根据发起形式又分为

自动恢复：通过对发生失败操作的接口自动重试或者回滚已经完成的操作

通知运营：如果程序无法自动完成恢复，则通过运营人员手动进行补偿

通知技术：通过监控或者告警通知到技术人员，通过技术手段进行修复

# X/OpenDTP模型的支付宝的DTS架构

DTS(Distributed Transaction Service)框架是由支付宝在X/OpenDTP模型的基础上改进的一个设计，定义了类似2PC的标准两阶段接口，业务系统只需要实现对应的接口就可以使用DTS的事务功能。DTS最大的特点是放宽了数据库的强一致约束，保证了数据的最终一致性。

具体的流程是

|  |  |
| --- | --- |
|  | TCC分为三个阶段TRYING-CONFIRMING-CANCELING。每个阶段做不同的处理。  TRYING、CONFIRMING、CANCELIING大致可以理解为SQL事务中的LOCK、COMMIT、ROLLBACK  TRYING 阶段主要是对业务系统做检测及资源预留  CONFIRMING 阶段主要是对业务系统做确认提交，TRYING阶段执行成功并开始执行CONFIRMING阶段时，默认 CONFIRMING阶段是不会出错的。即：只要TRYING成功，CONFIRMING一定成功。  CANCELING 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。  以上所有的操作需要满足幂等性，幂等性的实现方式可以是：  1、通过唯一键值做处理，即每次调用的时候传入唯一键值，通过唯一键值判断业务是否被操作，如果已被操作，则不再重复操作  2、通过状态机处理，给业务数据设置状态，通过业务状态判断是否需要重复执行 |

如何更通俗的理解TCC事务模型

|  |
| --- |
| *支付系统接收到会员的支付请求后，需要扣减会员账户余额、增加会员积分（暂时假设需要同步实现）增加商户账户余额*  *会员系统、商户系统、积分系统是独立的三个子系统，无法通过传统的事务方式进行处理。*  *TRYING阶段：我们需要做的就是会员资金账户的资金预留，即：冻结会员账户的金额（订单金额）*  *CONFIRMING阶段：我们需要做的就是会员积分账户增加积分余额，商户账户增加账户余额*  *CANCELING阶段：该阶段需要执行的就是解冻释放我们扣减的会员余额* |

开源的tcc框架

tcc-transaction

bytetcc

# 最大努力通知型

做过支付宝交易接口的同学都知道，我们一般会在支付宝的回调页面和接口里，解密参数，然后调用系统中更新交易状态相关的服务，将订单更新为付款成功。同时，只有当我们回调页面中输出了success字样或者标识业务处理成功相应状态码时，支付宝才会停止回调请求。否则，支付宝会每间隔一段时间后，再向客户方发起回调请求，直到输出成功标识为止。

其实这就是一个很典型的补偿例子，跟一些MQ重试补偿机制很类似。

# 源码解读要点

首先我们要关注的是服务的发布和服务的消费这两个主要的流程，那么就可以基于这个点去找到源码分析的突破口。那么自然而然我们就可以想到spring的配置

## Spring对外留出的扩展

dubbo是基于spring 配置来实现服务的发布的，那么一定是基于spring的扩展来写了一套自己的标签，那么spring是如何解析这些配置呢？

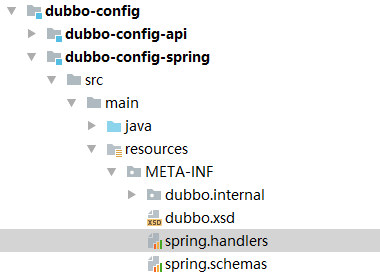
在spring中定义了两个接口

NamespaceHandler: 注册一堆BeanDefinitionParser，利用他们来进行解析

BeanDefinitionParser:用于解析每个element的内容

Spring默认会加载jar包下的META-INF/spring.handlers文件寻找对应的NamespaceHandler。

Dubbo-config模块下的dubbo-config-spring



## Dubbo的接入实现

Dubbo中spring扩展就是使用spring的自定义类型，所以同样也有NamespaceHandler、BeanDefinitionParser。而NamespaceHandler是DubboNamespaceHandler

|  |
| --- |
| **public class DubboNamespaceHandler extends NamespaceHandlerSupport {  static {  Version.*checkDuplicate*(DubboNamespaceHandler.class);  }  public void init() {  registerBeanDefinitionParser("application", new DubboBeanDefinitionParser(ApplicationConfig.class, true));  registerBeanDefinitionParser("module", new DubboBeanDefinitionParser(ModuleConfig.class, true));  registerBeanDefinitionParser("registry", new DubboBeanDefinitionParser(RegistryConfig.class, true));  registerBeanDefinitionParser("monitor", new DubboBeanDefinitionParser(MonitorConfig.class, true));  registerBeanDefinitionParser("provider", new DubboBeanDefinitionParser(ProviderConfig.class, true));  registerBeanDefinitionParser("consumer", new DubboBeanDefinitionParser(ConsumerConfig.class, true));  registerBeanDefinitionParser("protocol", new DubboBeanDefinitionParser(ProtocolConfig.class, true));  registerBeanDefinitionParser("service", new DubboBeanDefinitionParser(ServiceBean.class, true));  registerBeanDefinitionParser("reference", new DubboBeanDefinitionParser(ReferenceBean.class, false));  registerBeanDefinitionParser("annotation", new DubboBeanDefinitionParser(AnnotationBean.class, true));  } }** |

BeanDefinitionParser全部都使用了DubboBeanDefinitionParser，如果我们向看<dubbo:service>的配置，就直接看DubboBeanDefinitionParser中

这个里面主要做了一件事，把不同的配置分别转化成spring容器中的bean对象

application对应ApplicationConfig

registry对应RegistryConfig

monitor对应MonitorConfig

provider对应ProviderConfig

consumer对应ConsumerConfig

…

为了在spring启动的时候，也相应的启动provider发布服务注册服务的过程，而同时为了让客户端在启动的时候自动订阅发现服务，加入了两个bean

ServiceBean、ReferenceBean。

分别继承了ServiceConfig和ReferenceConfig

同时还分别实现了InitializingBean、DisposableBean, ApplicationContextAware, ApplicationListener, BeanNameAware

**InitializingBean**接口为bean提供了初始化方法的方式，它只包括afterPropertiesSet方法，凡是继承该接口的类，在初始化bean的时候会执行该方法。

**DisposableBean** bean被销毁的时候，spring容器会自动执行destory方法，比如释放资源

**ApplicationContextAware** 实现了这个接口的bean，当spring容器初始化的时候，会自动的将ApplicationContext注入进来

**ApplicationListener** ApplicationEvent事件监听，spring容器启动后会发一个事件通知

**BeanNameAware** 获得自身初始化时，本身的bean的id属性

那么基本的实现思路可以整理出来了

1. 利用spring的解析收集xml中的配置信息，然后把这些配置信息存储到serviceConfig中
2. 调用ServiceConfig的export方法来进行服务的发布和注册

# 服务的发布

## ServiceBean

serviceBean是服务发布的切入点，通过afterPropertiesSet方法，调用export()方法进行发布。

export为父类ServiceConfig中的方法，所以跳转到SeviceConfig类中的export方法

**delay的使用**

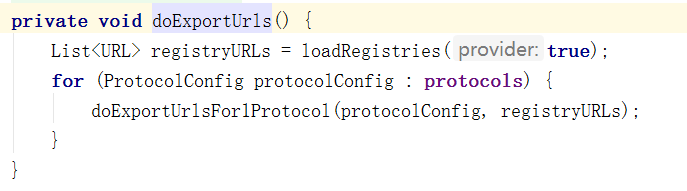


我们发现，delay的作用就是延迟暴露，而延迟的方式也很直截了当，Thread.sleep(delay)

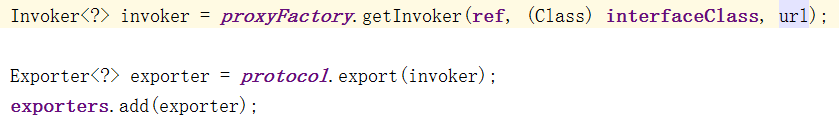
1. export是synchronized修饰的方法。也就是说暴露的过程是原子操作，正常情况下不会出现锁竞争的问题，毕竟初始化过程大多数情况下都是单一线程操作，这里联想到了spring的初始化流程，也进行了加锁操作，这里也给我们平时设计一个不错的启示：初始化流程的性能调优优先级应该放的比较低，但是安全的优先级应该放的比较高！
2. 继续看doExport()方法。同样是一堆初始化代码

## export的过程

继续看doExport()，最终会调用到doExportUrls()中：



最终实现逻辑



在上面这段代码中可以看到Dubbo的比较核心的抽象：Invoker， Invoker是一个代理类，从ProxyFactory中生成。这个地方可以做一个小结

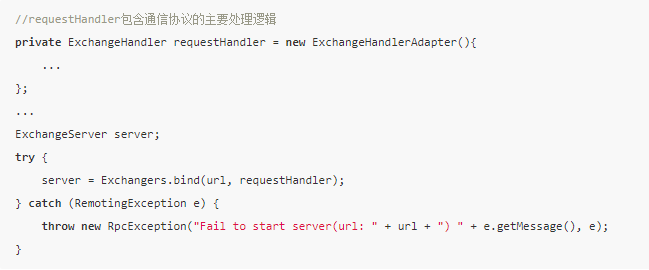
1. Invoker -执行具体的远程调用
2. Protocol – 服务地址的发布和订阅
3. Exporter – 暴露服务或取消暴露

## protocol发布服务

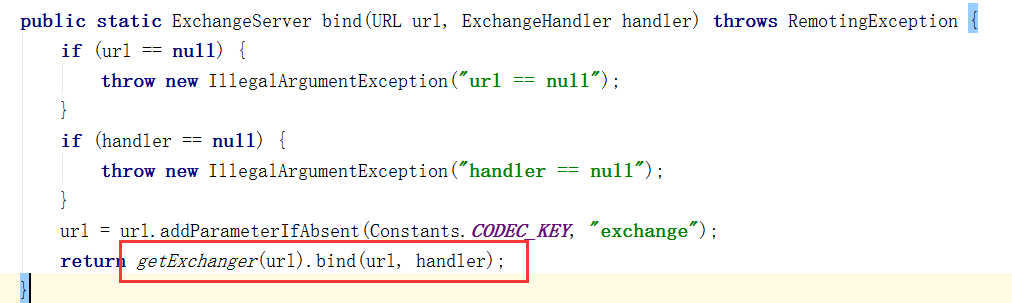
我们看一下dubboProtocol的export方法：openServer(url）

接着调用openServer， 继续createServer 创建服务

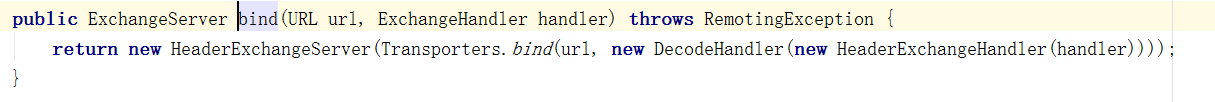
继续看其中的createServer方法：



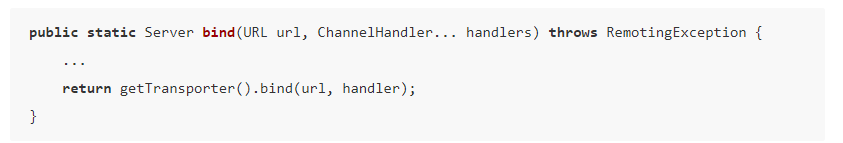
发现ExchangeServer是通过Exchangers创建的，直接看Exchanger.bind方法



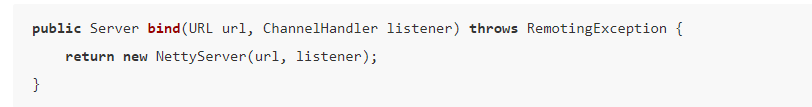
getExchanger方法实际上调用的是ExtensionLoader的相关方法，这里的ExtensionLoader是dubbo插件化的核心，我们会在后面的插件化讲解中详细讲解，这里我们只需要知道Exchanger的默认实现只有一个：HeaderExchanger。上面一段代码最终调用的是：



可以看到Server与Client实例均是在这里创建的，HeaderExchangeServer需要一个Server类型的参数，来自Transporters.bind()：



getTransporter()获取的实例来源于配置，默认返回一个NettyTransporter：



# 服务消费

## ReferenceBean

和serviceBean发布一样，也是使用NamespaceHandler作为切入点，调用ReferenceBean里面的afterPropertiesSet方法

**方法调用顺序afterPropertiesSet() -> getObject() -> get() -> init() -> createProxy()**

afterPropertiesSet方法中都是确认所有的组件是否都初始化好了，都准备好后我们进入生成Invoker的部分。这里的getObject会调用父类ReferenceConfig的init方法完成组装：





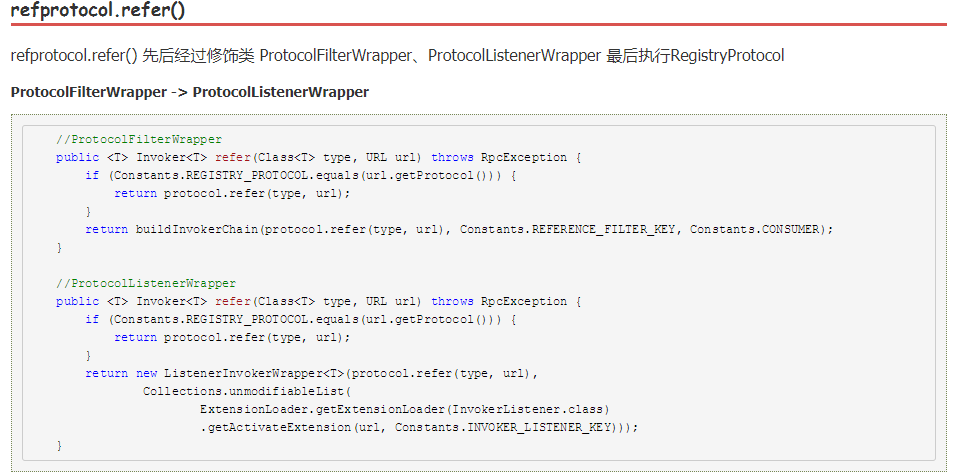


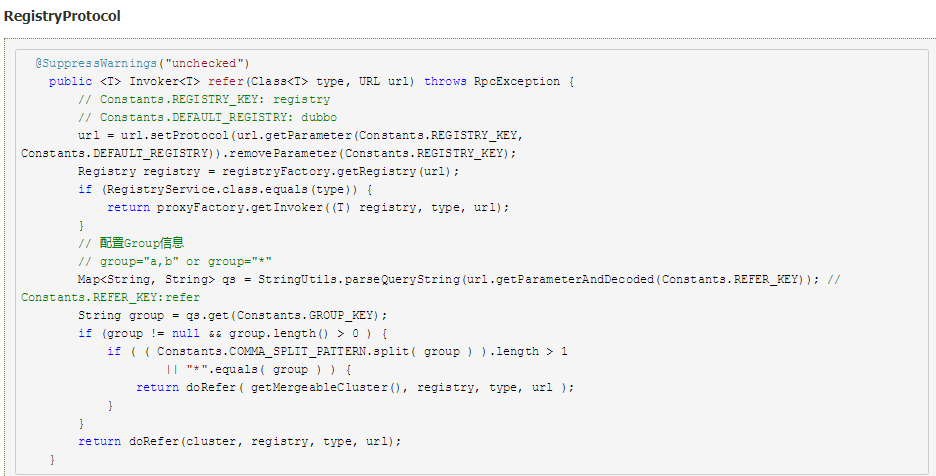
## createProxy方法





## refprotocol.refer





至此Reference在关联了所有application、module、consumer、registry、monitor、service、protocol后调用对应Protocol类的refer方法生成InvokerProxy。当用户调用service时dubbo会通过InvokerProxy调用Invoker的invoke的方法向服务端发起请求。客户端就这样完成了自己的初始化。

这个代理实例中仅仅包含一个handler对象（InvokerInvocationHandler类的实例），handler中则包含了RPC调用中非常核心的一个接口Invoker<T>的实现，Invoker接口的的的定义如下：

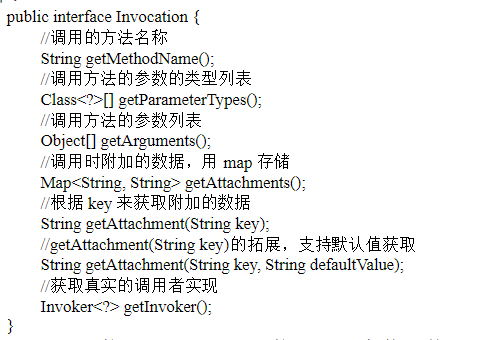
public interface Invoker<T> extends Node {

Class<T> getInterface(); //调用过程的具体表示形式

Result invoke(Invocation invocation) throws RpcException;

}

Invoker<T>接口的核心方法是invoke(Invocation invocation)，方法的参数Invocation是一个调用过程的抽象，也是Dubbo框架的核心接口，该接口中包含如何获取调用方法的名称、参数类型列表、参数列表以及绑定的数据，定义代码如下：



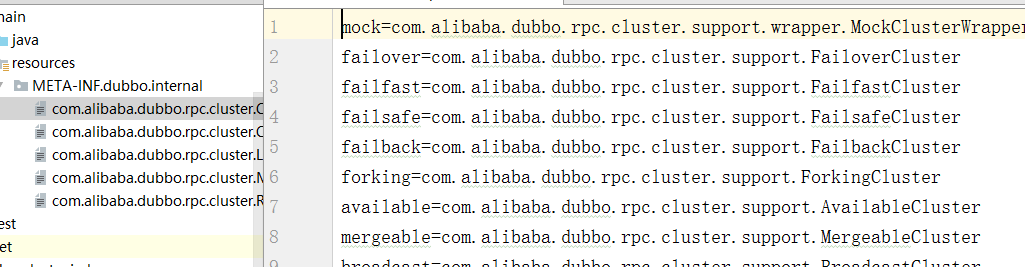
代理中的handler实例中包含的Invoker<T>接口实现者是MockClusterInvoker，其中MockClusterInvoker仅仅是一个Invoker的包装，并且也实现了接口Invoker<T>，其只是用于实现Dubbo框架中的mock功能，我们可以从他的invoke方法的实现中看出

# Dubbo插件化

Dubbo的插件化实现非常类似于原生的JAVA的SPI：它只是提供一种协议，并没有提供相关插件化实施的接口。用过的同学都知道，它有一种java原生的支持类：ServiceLoader，通过声明接口的实现类，在META-INF/services中注册一个实现类，然后通过ServiceLoader去生成一个接口实例，当更换插件的时候只需要把自己实现的插件替换到META-INF/services中即可。

## Dubbo的“SPI”

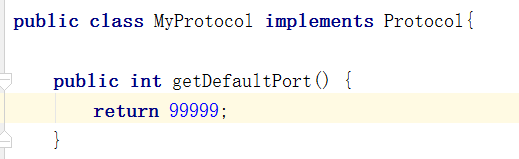
Dubbo的SPI并非原生的SPI，Dubbo的规则是在META-INF/dubbo、META-INF/dubbo/internal或者META-INF/services下面以需要实现的接口去创建一个文件，并且在文件中以properties规则一样配置实现类的全面以及分配实现的一个名称。我们看一下dubbo-cluster模块的META-INF.dubbo.internal：



## 实现自己的扩展点

假如我们使用自己定义的协议MyDefineProtocol

1. 在resources目录下新建META-INF/dubbo/com.alibaba.dubbo.rpc.Protocol文件，文件内容为com.\*\*\*.MyDefineProtocol
2. 实现类的内容



1. 最后在main方法中调用



1. 通过结果可以看到我们已经找到

## 源码分析

dubbo的扩展点框架主要位于这个包下：

com.alibaba.dubbo.common.extension

大概结构如下：

1. com.alibaba.dubbo.common.extension
2. |
3. |--factory
4. |     |--AdaptiveExtensionFactory   #稍后解释
5. |     |--SpiExtensionFactory        #稍后解释
6. |
7. |--support
8. |     |--ActivateComparator
9. |
10. |--Activate  #自动激活加载扩展的注解
11. |--Adaptive  #自适应扩展点的注解
12. |--ExtensionFactory  #扩展点对象生成工厂接口
13. |--ExtensionLoader   #扩展点加载器，扩展点的查找，校验，加载等核心逻辑的实现类
14. |--SPI   #扩展点注解

其中最核心的类就是ExtensionLoader，几乎所有特性都在这个类中实现。

ExtensionLoader没有提供public的构造方法，但是提供了一个public static的getExtensionLoader，这个方法就是获取ExtensionLoader实例的工厂方法。其public成员方法中有三个比较重要的方法：

getActivateExtension ：根据条件获取当前扩展可自动激活的实现

getExtension ： 根据名称获取当前扩展的指定实现

getAdaptiveExtension : 获取当前扩展的自适应实现

|  |  |
| --- | --- |
|  | 该方法需要一个Class类型的参数，该参数表示希望加载的扩展点类型，该参数必须是接口，且该接口必须被@SPI注解注释，否则拒绝处理。检查通过之后首先会检查ExtensionLoader缓存中是否已经存在该扩展对应的ExtensionLoader，如果有则直接返回，否则创建一个新的ExtensionLoader负责加载该扩展实现，同时将其缓存起来。可以看到对于每一个扩展，dubbo中只会有一个对应的ExtensionLoader实例。 |

接下来看下ExtensionLoader的私有构造函数：

|  |  |
| --- | --- |
|  | 这里保存了对应的扩展类型，并且设置了一个额外的objectFactory属性，他是一个ExtensionFactory类型，ExtensionFactory主要用于加载扩展的实现： |

ExtensionFactory主要用于加载扩展的实现：

|  |  |
| --- | --- |
|  | ExtensionFactory有@SPI注解，说明当前这个接口是一个扩展点。从extension包的结构图可以看到。Dubbo内部提供了两个实现类：SpiExtensionFactory和AdaptiveExtensionFactory。不同的实现可以以不同的方式来完成扩展点实现的加载。 |

默认的ExtensionFactory实现中，AdaptiveExtensionFactotry被@Adaptive注解注释，也就是它就是ExtensionFactory对应的自适应扩展实现(每个扩展点最多只能有一个自适应实现，如果所有实现中没有被@Adaptive注释的，那么dubbo会动态生成一个自适应实现类)，也就是说，所有对ExtensionFactory调用的地方，实际上调用的都是AdpativeExtensionFactory，那么我们看下他的实现代码：

|  |  |
| --- | --- |
|  | 这段代码，其实就相当于一个代理入口，它会遍历当前系统中所有的ExtensionFactory实现来获取指定的扩展实现，获取到扩展实现，遍历完所有ExtensionFactory实现，调用ExtensionLoader的getSupportedExtensions方法来获取ExtensionFactory的所有实现 |

从前面ExtensionLoader的私有构造函数中可以看出，在选择ExtensionFactory的时候，并不是调用getExtension(name)来获取某个具体的实现类，而是调用getAdaptiveExtension来获取一个自适应的实现。那么首先我们就来分析一下getAdaptiveExtension这个方法的实现吧：

|  |  |
| --- | --- |
|  | 首先检查缓存的adaptiveInstance是否存在，如果存在则直接使用，否则的话调用createAdaptiveExtension方法来创建新的adaptiveInstance并且缓存起来。也就是说对于某个扩展点，每次调用ExtensionLoader.getAdaptiveExtension获取到的都是同一个实例。 |

createAdaptiveExtension方法

|  |  |
| --- | --- |
|  | 在createAdaptiveExtension方法中，首先通过getAdaptiveExtensionClass方法获取到最终的自适应实现类型，然后实例化一个自适应扩展实现的实例，最后进行扩展点注入操作 |

|  |  |
| --- | --- |
|  | 他只是简单的调用了getExtensionClasses方法，然后在判adaptiveCalss缓存是否被设置，如果被设置那么直接返回，否则调用createAdaptiveExntesionClass方法动态生成一个自适应实现，关于动态生成自适应实现类然后编译加载并且实例化 |

先看getExtensionClasses方法

|  |  |
| --- | --- |
|  | 在getExtensionClasses方法中，首先检查缓存的cachedClasses，如果没有再调用loadExtensionClasses方法来加载，加载完成之后就会进行缓存。也就是说对于每个扩展点，其实现的加载只会执行一次。我们看下loadExtensionClasses方法： |

|  |  |
| --- | --- |
|  | 从代码里面可以看到，在loadExtensionClasses中首先会检测扩展点在@SPI注解中配置的默认扩展实现的名称，并将其赋值给cachedDefaultName属性进行缓存，后面想要获取该扩展点的默认实现名称就可以直接通过访问cachedDefaultName字段来完成，比如getDefaultExtensionName方法就是这么实现的。从这里的代码中又可以看到，具体的扩展实现类型，是通过调用loadFile方法来加载，分别从一下三个地方加载：  META-INF/dubbo/internal/  META-INF/dubbo/  META-INF/services/ |

调用loadFile方法，代码比较长，主要做了几个事情，有几个变量会赋值

cachedAdaptiveClass : 当前Extension类型对应的AdaptiveExtension类型(只能一个)

cachedWrapperClasses : 当前Extension类型对应的所有Wrapper实现类型(无顺序)

cachedActivates : 当前Extension实现自动激活实现缓存(map,无序)

cachedNames : 扩展点实现类对应的名称(如配置多个名称则值为第一个)

当loadExtensionClasses方法执行完成之后，还有以下变量被赋值：

cachedDefaultName : 当前扩展点的默认实现名称

当getExtensionClasses方法执行完成之后，除了上述变量被赋值之外，还有以下变量被赋值：

cachedClasses : 扩展点实现名称对应的实现类(一个实现类可能有多个名称)

其实也就是说，在调用了getExtensionClasses方法之后，当前扩展点对应的实现类的一些信息就已经加载进来了并且被缓存了。后面的许多操作都可以直接通过这些缓存数据来进行处理了。

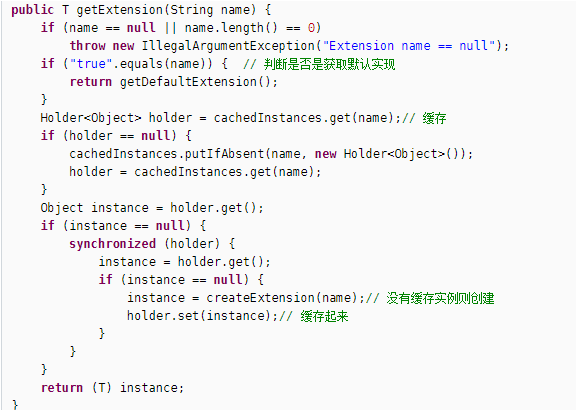
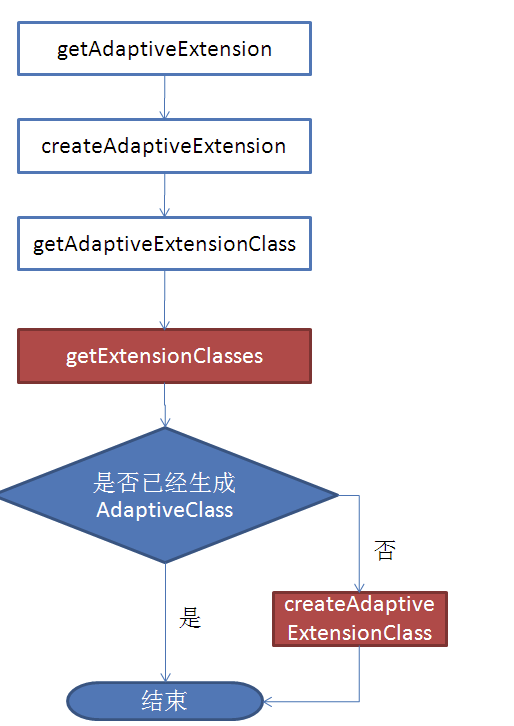
回到createAdaptiveExtension方法，他调用了getExtesionClasses方法加载扩展点实现信息完成之后，就可以直接通过判断cachedAdaptiveClass缓存字段是否被赋值盘确定当前扩展点是否有默认的AdaptiveExtension实现。如果没有，那么就调用createAdaptiveExtensionClass方法来动态生成一个。在dubbo的扩展点框架中大量的使用了缓存技术。

创建自适应扩展点实现类型和实例化就已经完成了，下面就来看下扩展点自动注入的实现injectExtension

|  |  |
| --- | --- |
|  | 这里可以看到，扩展点自动注入的一句就是根据setter方法对应的参数类型和property名称从ExtensionFactory中查询，如果有返回扩展点实例，那么就进行注入操作。到这里getAdaptiveExtension方法就分析完毕了。 |

## getExtension

这个方法的主要作用是用来获取ExtensionLoader实例代表的扩展的指定实现。已扩展实现的名字作为参数，结合前面学习getAdaptiveExtension的代码

# 总结

在整个过程中，最重要的两个方法getExtensionClasses和createAdaptiveExtensionClass

**getExtensionClasses**

这个方法主要是读取META-INF/services 、META-INF/dubbo、META-INF/internal目录下的文件内容

分析每一行，如果发现其中有哪个类的annotation是@Adaptive，就找到对应的AdaptiveClass。如果没有的话，就动态创建一个

**createAdaptiveExtensionClass**

该方法是在getExtensionClasses方法找不到AdaptiveClass的情况下被调用，该方法主要是通过字节码的方式在内存中新生成一个类，它具有AdaptiveClass的功能，Protocol就是通过这种方式获得AdaptiveClass类的。