**分布式事务：**

事务的概念：

事务是以一种可靠的，一致的方式，访问和操作数据库中数据的程序单元。

1.分布式事务定义：在分布式系统中，给用户提供一个可靠的服务，让它可以通过一致的方式来访问数据。

分布式事务原则：

分布式事务实现原则

Spring分布式事务

分布式事务实现的几种模式：

消息驱动，事件朔源，TCC

**Spring的事务机制**

1.Spring事务管理：

（1）提供的统一的API接口支持不同的资源

（2）提供声明式的事务

（3）方便的与spring框架集成

（4）多个资源的事务管理，同步

Spring事务抽象：

1.PlatformTransactionManager：事务管理器接口

2.TransactionDefinition：事务的定义，包括传播属性，隔离属性（设置事务的属性）

3.TransactionStatus：事务的运行状态

代码实例：

public interface PlatformTransactionManager{

TransactionStatus getTransaction(TransactionDefinition definition)

commit(TransactionStatus status)

rollback(TransactionStatus status)

}

public interface TransactionDefinition{

int getPropagationBehavior();

int getIsolationLevel();

String getName();

int getTimeOut();

boolean isReadOnly();

}

public interface TransactionStatus{

boolean isNewTransaction();

boolean hasSavepoint();

void setRollbackOnly();

boolean isRollbackOnly();

boolean isCompleted();

}

TransactionDefinition --事务隔离级别

TransactionDefinition.ISOLATION\_DEFAULT

TransactionDefinition.ISOLATION\_READ\_COMMITTED

TransactionDefinition.ISOLATION\_READ\_UNCOMMITTED

TransactionDefinition.ISOLATION\_REPEATABLE\_READ

TransactionDefinition.ISOLATION\_SERIALIZABLE

TransactionDefinition -----事务的传播机制

是指service层两个方法互相调用，双方都存在事务（注意：以下ServiceA，ServiceB都存在事务）

**1.TransactionDefinition.PROPAGATION\_REQUIRED(default)（假设为B的默认事务传播机制）**

当ServiceA调用ServiceB时，ServiceB要求事务，若A不存在事务，则B会新建一个事务，相反B则不会新建事务

**2.TransactionDefinition.PROPAGATION\_SUPPORTS（假设为B的默认事务传播机制）**

A调用B时，当A在事务中时，B也会在事务中执行，若A不在事务中，则B也不会在事务中执行

**3.TransactionDefinition.PROPAGATION\_MANDATORY（假设为B的默认事务传播机制）**

A调用B时，B必须在事务中执行，否则报错

**4.TransactionDefinition.PROPAGATION\_REQUIRED\_NEW（假设为B的默认事务传播机制）JTA事务管理器中使用**

A调用B时，无论A是否在一个事务中，B都会新建一个事务来进行操作，若A存在事务，则会将A的事务挂起

**5.TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED JTA事务管理器中使用**

A调用B时，无论A是否在一个事务中，当B执行时，若A存在事务，则会将A的事务挂起，B不支持事务，待B执行完时，在执行A

**6.TransactionDefinition.PROPAGATION\_NEVER**

A调用B时，B绝不在一个事务中进行

**7.TransactionDefinition.PROPAGATION\_NESTEDED （JDBC3.0中使用）**

嵌套事务，A调用B时，相当于在A的事务中开启一个小事务，设置存盘点，方便回滚

**Spring事务编码实现：**

@Autowired

PlatformTransactionManager transactionManager；

public Customer create(Custer cu){

DefaultTransactionDefinition def=new DefaultTransactionDefinition();

def.setIsolactionLevel(TransactionManager.ISOLACTION\_SERIALIZABLE);

def.setPropagationBehavior(TransactionMananger.PROPAGATION\_SUPPORTS);

TransactionStatus ts=transactionMananger.getTransaction(def);

try{

service.save(cu);

transactionManager.commit(ts);

}catch(Exception e){

transactionMaanager.rollback(ts);

throw e;

}

}

SpringJMS事务：Session原生事务：

流程：

后续补充：

**Spring 外部事务和JTA**

1.Spring 内部事务和外部事务

2、JTA 与Spring JTA的实现

3.JTA 与 XA

4. JTA分布式事务实例

**本地事务：**

1.Spring容器管理事务的生命周期

2.通过Spring事务接口调用

3.业务代码与具体事务的实现无关

**执行流程为：**

调用者--> Spring事务接口 (用户通过接口调用管理事务)--> Spring事务实现（实际上调用相应事务的具体方法） --> 资源管理器（Spring的事务实现调用资源管理器的事务管理方法）--->数据库

**外部（全局）事务：**

1.外部事务管理器提供事务管理

2.通过Spring事务接口，调用外部管理器

3，使用JNDI等方式获取外部事务管理器的实例

4.外部事务管理器一般由应用服务器提供，Jboss

**外部（全局）事务- JTA**

1.外部事务管理器提供JTA事务管理

2.JTA事务管理器可以管理多个数据资源

3.通过2阶段提交实现多数据源的事务

**执行流程：（应用服务器）**

调用者--> Spring的事务接口（用户通过接口调用管理事务【spring容器】）--> Spring JTA(实际上调用Spring JTA实现事务管理【spring容器】) ---> JTA资源管理器（SpringJTA 通过应用服务器的JTA管理器管理事务【服务器】）-->资源管理器（应用服务器的JTA管理器调用资源的事务方法）【应用服务器】-->数据库

**执行流程- （不使用应用服务器）**

调用者 --> Spring 事务接口 (用户通过接口调用管理事务【spring容器】) ---->Spring JTA(实际上调用Spring JTA实现事务管理【spring容器】) --> JTA资源管理器（调用Spring容器内的某个JTA事务管理器【spring容器】）--> 资源管理器 ----> 数据库

**XA 与JTA（XA规范的JAVA实现即为JTA）:**

XA是指一种分布式事务规范，此规范定义了全局的事物管理器接口（Transaction Manager 接口）以及对数据资源封装的接口（XA Resource） ，通过ResourceManager 对多个Resourcce进行控制，而TransactionManager 可以对多个ResourceManager进行管理

1.Trasaction Mananger

2.XA Resource

3. 两阶段提交

**流程图：（JTA Transaction Manager是指Java实现的XA规范的接口，XA Resource 即Java针对XA Resource的接口）**

Transaction manager （JTA Transaction Manager）--------->ResourceManager(多个) -------------->Resource (多个，XA Resource)

**JTA具体内容（以下为JTA中的接口）：**

1. Transaction Manager 接口

2. XAResource 接口

3. XID 事务ID接口

**JTA 事务管理的缺点：不使用JTA的原因**

1.两阶段提交（不同数据源事务提交的快慢影响）

2.事务时间太长，锁数据的时间太长

3，低性能，低吞吐量

**不使用JTA实现多数据源的事务管理（类似JTA，但并不能真正满足需求）**

1.Spring事务同步机制

2，多个数据源上实现近似事务一致性

3.高性能，高吞吐量

多个事务的同步方法：

1.XA与最后资源博弈：

2.共享资源

两个数据源共享同一个底层资源

ActiveMq使用DB作为存储

使用DB上的connection控制事务提交

需要数据源支持

3.最大努力一次提交

依次提交事务，可能会出错，然后通过AOP或listener实现事务的直接同步

JMS最大努力一次性提交+重试

适用于其中一个数据源是MQ，并且事务由读MQ消息开始

利用MQ的消息重试机制，但是重试时需要考虑重复消息

数据库操作出错，消息会被重新放回MQ，重新出发该方法

4.链式事务

spring定义一个事务链，多个事务在一个事务管理器里一次提交，但是有可能出错；

Spring分布式事务的实现：

选择依据（根据一致性要求）：

1.强一致性事务：JTA（性能最差，只适用于单个事务）

2.弱，最终一致性事务：最大努力一次提交，链式事务（设计模式相应的错误处理机制）

根据场景选择：

1.MQ+DB:(最大努力一次提交+重试)

2.多个DB：（链式事务管理）DataSourceTransactionManager

链式的事务管理器：ChainedTransactionManager

eg：

ChainedTransactionManager ch=new ChainedTransactionManager(userMn,OrderMn);

//事务提交顺序：OrderMn->userMn

当有数据库宕掉的时候，链式事务管理就会出问题

创建步骤：

【1】DataSourceProperties

【2】DataSource ==》 DataSourceProperties().initializeDataSourceBuilder().type(DataSource.class).build();

【3】ChainedTransactionManager

3.多个数据源（链式事务，或其他事务同步）

==============

**微服务设计原则：**

1.单一职责原则

2.服务自治原则：服务自治，即每个微服务应该具备独立的能力，依赖与运行环境

3.轻量级通信原则：

4.接口明确原则：每个微服务对外接口应明确，尽量保持不变

**Dubbo相关文档描述，官网：**[**http://dubbo.apache.org/zh-cn/docs/user/quick-start.html**](http://dubbo.apache.org/zh-cn/docs/user/quick-start.html)

**Dubbo配置：**

1.多版本控制，主要用于服务升级

<dubbo:service interface="com.bao.inter.demo" version="0.0.1" ref="demo1"/>

<dubbo:service interface="com.bao.inter.demo" version="0.0.2" ref="demo2"/>

2,.服务分组，则是用于服务的共存

<dubbo:service interface="com.bao.inter.demo" group="par.weixin" ref="demo1"/>

<dubbo:service interface="com.bao.inter.demo" group="par.ali" ref="demo2"/>

<!--不同的接口实现，业务逻辑不同-->

**3.多协议支持**

目前dubbo支持多协议，9个协议，如下所示：

**协议选取：若存在大数据量进行传输，则选择短连接协议，小数据了传输可以选择 长链接协议【即保持默认协议，dubbo协议】**

1.dubbo协议

默认的传输协议，单链接，长链接，传输协议TCP，采用NIO异步传输

适用于消费者多于服务者的情况

2.rmi协议

多连接，短连接，传输协议TCP，采用BIO传输

适用于消费者和提供者数量差不多的情况。可以传输文件

3.hession协议

多连接，短连接，传输协议Http，采用BIO传输

适用于消费者比提供者少的情况。可以传输文件

4.http协议

多连接，短连接。传输协议Http，采用BIO传输。

适用于消费者比提供者少的情况，不支持传输文件。

5.webService协议

6.thrift协议

7.memcached协议与redis协议

8.rest协议

实例：

**<dubbo:protocol name="dubbo" port="20881"/>**

**<dubbo:protocol name="rmi" port="1099"/>**

**<dubbo:protocol name="hession" port="20882">**

**消费者：**

**<dubbo:reference id="othservice" interface="com.bao.inter.demo" group="pay.ali" protocol="dubbo,rmi" />**

**提供者“**

**<dubbo:service interface="com.bao.inter.demo" group="pay.ali" protocol="dubbo">**

**dubbo负载均衡支持：**

1.在src路径下建立dubbo.properties文件

用于指定telnet连接dubbo的端口号

里面添加内容：dubbo.application.qos.port=33333

2.dubbo内置的负载均衡算法：

【1】random

默认算法，其存在服务堆积的可能。

【2】roundrobin

轮询算法，按照设定好的权重进行调度。权重是设置在提供者端。

【3】leastactive

最少活跃度算法，即调用次数最少，优先级越高，被调用的几率就越高

【4】consistenthash

此算法可以保证，相同参数的请求其会调度到相同的提供者，默认只对提供者方法的第一个参数进行hash。

配置实例：

<dubbo:service interface="com.bao.inter.demo" group="pay.weixin" ref="otherService" protocol="dubbo">

<dubbo:method name="hello">

<!---表示对第二个参数进行hash，方法哪个参数位置为1则表示取谁进行hash-->

<dubbo:paratemer key="hash.arguments" value="0,1,0" >

<!--表示能够最多设置的虚拟主机个数-->

<dubbo:parameter key="hash.node" value="320"/>

</dubbo:method>

</dubbo:service>

注意：

若经过hash后的落点提供者主机发生宕机， 则dubbo会自动创建一个对应的虚拟主机，使相同参数的请求仍能落入到该提供者（虚拟机），然后Dubbo会将这些请求均匀分布到其他仍能提供服务的生产者中。dubbo默认创建160个虚拟主机节点、

【2】roundrobin轮询随机算法，配置点不同，产生的效果也不一样

1、配置在消费端

<dubbo:reference id=" demo1" interface="com.bao.inter.demo" protocol=“dubbo” group="pay.weixin" loadbalance=“roundrobin” weight=“100” />

解析：配置在消费端，表示只有该接口在调度生产者时，会采用此负载轮询算法，其余没有该配置的消费者则不会进行负载轮询算法

2.配置在生产端

<dubbo:service interface="com.bao.inter.demo" protocol="dubbo" loadbalance="roundrobin" group="pay.weixin" weight=“200”/>

解析：配置在服务端，表示只有该接口在生产调度时，所有调度该生产者的消费者都会采用该负载均衡算法。

**Dubbo内置的容错机制：**

**1.FailOver**

**故障转移机制，当调用提供者集群中的某个服务器失败时，其会自动尝试着调用其他服务器。**

**该机制一般用于读操作。其重试次数可以设置，【但会存在延时，因为调用重试原因】**

可以设置在服务提供者端或消费端：

实例：

**消费端设置：**

<dubbo:reference id="demo1" interface="com.bao.inter.demo" group="pay.weixin" loadbalance="roundrobin" protocol="dubbo" retries="2"/>

**解析：**

**retries设置为2，表示该消费者最多可以调用该指定服务3次，一次正常调用，2次重试。**

**【即第一次调用集群中某个指定生产者失败，则会切换调用下一个同类型的生产者，若还是失败则同理，直到重试次数到达设置好的次数】**

**服务端设置：**

<dubbo:service interface="com.bao.inter.demo" group="pay.weixin" loadbalance="roundrobin" protocol="dubbo" retries="2"/>

解析：

**retries设置为2，表示当前服务无论谁来访问，最多访问3次，一次正常访问，两次重试【表示该生产者第一次被消费者调用失败后，消费者不会切换到其他同类型生产服务，而是继续对该服务进行调用，直到达到设置的重试次数，达到重试次数后，才可能切换到下一个同类型服务】**

**2.FailFast**

**快速失败机制，消费端只发出一次调用，若失败则立即报错，通常用于非幂等性的写入操作**。

延伸：

**幂等性：**

**它是分布式系统中对请求的一种描述符。**

错误理解：当客户端向一个分布式系统发出系统的请求，可以获取到相同的结果，则该请求相对于当前分布式系统来说是幂等的

**正确的理解：当客户端向一个分布式系统发出相同的请求，对当前的分布式系统产生系统的影响，则该请求相对于当前分布式系统来说是幂等的。**

**常见的HTTP请求有： GET ， POST , PUT ,DELETE【以下情况为一般性情况】**

**GET请求：属于幂等性请求【没有对数据库造成改变】**

**POST请求：属于非幂等性请求【如新增记录】**

**PUT请求：属于幂等性请求 【如多次更新一条记录】**

**DELETE请求：属于幂等性请求【如多次删除同一条记录】**

**3.Fialback**

失败自动恢复机制，当消费者调用提供者失败后，dubbo会记录下该失败请求，并定时自动重新发送请求，该机制常用于实时性要求不高的服务，eg：消息通知

**4.Forking**

**并行机制。一次性向集群中所有的生产者服务节点发送请求，只要有一个成功，就立即返回。对于实时性要求很高的服务。但是会消耗大量的系统资源。**

**5.Broadcast**

**广播机制。消费者以广播的形式调用所有的提供者，只要任意一个提供者服务器报错，则消息报错。通常用于通知所有提供者更新缓存，更新日志等资源。**

**6.fastsafe**

**失败安全机制，若消费者调用提供者发生异常，则直接忽略本次消费操作。一般用于执行相对不太重要的服务。例如：写入审计日志等。**

**Dubbo服务降级：**

1.常见服务降级有：

【1】：部分服务暂停，页面可以访问，但部分服务不能访问

【2】部分服务延迟：页面可以访问，服务也可以执行，但向用户返回响应慢

【3】全部服务暂停：整个系统不能访问

【4】随机拒绝服务：随机某个服务不能访问

服务降级埋点设置：

一般设置在：

【1】反向代理服务器

【2】服务消费端

【3】数据缓存层

【4】消息中间件

【5】服务提供者

**Dubbo声明式缓存：**

概要：主要是为了进一步提高消费者对用户的响应速度，Dubbo提供了基于结果的声明式缓存。该缓存是基于消费者端的，所以使用很简单。只需要修改消费者配置文件即可，与提供者无关，

配置实例：

<dubbo:reference id="demo1" interface="com.bao.inter.demo" protocol="dubbo"

retries="2" loadbalance="roundrobin" cache="true"/>

基于方法设置声明式缓存：消费端设置

<dubbo:reference id="demo1" interface="com.bao.inter.demo" protocol="dubbo"

retries="2" loadbalance="roundrobin">

<dubbo:method name="hello" cache="lru"></dubbo:method>

</dubbo:reference>

Dubbo提供了三种缓存机制：

【1】lru：服务级别缓存的默认机制，该机制默认缓存1000个结果，若超出1000个缓存结果，将采用最近最少使用原则来删除缓存，以保证热点数据。

【2】threadlocal：当前线程缓存。当多个线程要对当前线程进行统一操作时首先需要查询当前线程的某个信息，通过线程缓存，可以有效减少查询。

【3】jcache：可以桥接各种缓存实现，及第三方缓存产品

查询缓存结果的应用场景：

dubbo的结果查询缓存可以应用在查询结果不变的场景，即不能使用在如下场景，消费者A调用的业务方法后从DB查询到一个结果a，然后消费者B对该DB中的结果相关数据进行了修改，以使该查询结果变为b。但是由于使用了结果缓存，消费者A中调用业务方法后的查询结果将长时间为a。直到该结果由于缓存空间满了而被消除，否则永远无法得到更新过的结果

**Dubbo服务延迟暴露**

1. 即在Dubbo服务启动过程中，需要warmup事件（预热事件，在启动一段时间后才能达到最佳状态），eg：初始化缓存，等待相关资源就位。等，可以使用deplay进行延迟暴露

只需要在<dubbo:service>标签中添加deplay属性，其值若为正数，单位为毫秒。则表示在多少毫秒后在发布服务。若为-1，则表示在spring加载完毕之后在暴露服务。

eg：

<dubbo:service interface="com.bao.inter.demo" protocol="dubbo" ref="demo1" retries="2"

deplay="1">

========================

**SpringCloud章节：【在SpringBoot基础之上快速构建分布式系统的工具集】**

1.微服务概念：

【1】每个微服务可以独立的运行在自己的进程里

【2】一系列独立运行的微服务共同构建了整个系统

【3】每个服务为独立的业务开发，一个微服务一般完成某个特定的功能

【4】微服务之间通过一些轻量的通信机制进行通讯

微服务的核心就是将传统的一站式应用，按业务拆分成一个个服务，彻底地去耦合，每一个微服务提供单个业务功能的服务。一个服务做一件事，从技术角度来看就是一种小而独立的处理过程，类似进程概念，能够自行单独启动或销毁，拥有自己独立的数据库。

2.微服务的优点：

【1】每个微服务足够内聚，足够小，功能点和需求点集中，一个服务只干单独的一件事

【2】微服务是松耦合的，是有功能意义的服务，每个阶段都是独立的。

【3】微服务可以使用不同的语言开发。

【4】 每个微服务可以有自己独立的数据库，微服务只是业务逻辑代码，不会和CSS，HTML等其他界面组件耦合。

3.微服务的缺点：

【1】分布式系统的复杂性

【2】多服务运维难度，随着服务的增加，运维的压力也增大

【3】系统部署依赖

【4】服务间通信成本

【5】数据一致性问题

【6】系统性能监控，集成测试等问题。

一个分布式系统需要具备哪些维度：

服务熔断：hystrix

服务路由：zuul，gateway

服务配置中心管理：springCloudConfig

消息队列：kafaka,RabbitMq

服务注册：zookeeper与eureka

服务负载均衡：ribbon，nginx

服务监控：metrics

服务配置与管理：archains，diamond

服务调用：Rest，RPC，GRPC，接口调用feign

等

springCloud是分布式微服务架构下的一站式解决方案，是各个微服务架构落地技术的集合体，俗称微服务全家桶。

2.SpringCloud和Dubbo的区别

最大的区别：SpringCloud抛弃了Dubbo的RPC通信，采用的是HTTP的REST方式。

两者各有优势，从一定程度上，springCloud牺牲了服务调用的性能，单页避免上面提到的原生RPC带来的问题，而且REST相比RPC更为灵活，服务提供方和调用方的依赖只靠一纸契约。不存在代码级别的强依赖，这在强调快速演化的微服务环境下，先得更为合适。

3.SpringBoot和SpringCloud的关系

springboot专注于快速方便的开发一个个微服务，springcloud是关注于全局的微服务协调整理治理框架。他将springboot开发的一个个单体微服务整合管理起来，为各个服务之间提供，配置管理，服务发现，断路器，路由，微代理，事件总线，全局锁，决策选举，分布式会话等集成服务。

springboot不依赖于springcloud，springCloud依赖于springboot。

4.什么是服务熔断？什么是服务降级

熔断机制，是应对雪崩效应的一种微服务链路保护机制，当扇出的链路的某个微服务不可用或相应时间太长，会进行服务降级，进而熔断该节点微服务的调用，快速返回“错误”的相应信息

服务降级:含义就是指：整体资源快不够了，先将某些服务停掉，待避过高峰期，在开启回来。

注意：

服务降级处理是在客户端完成的，与服务端没有关联

5.微服务优缺点

2.微服务的优点：

【1】每个微服务足够内聚，足够小，功能点和需求点集中，一个服务只干单独的一件事

【2】微服务是松耦合的，是有功能意义的服务，每个阶段都是独立的。

【3】微服务可以使用不同的语言开发。

【4】 每个微服务可以有自己独立的数据库，微服务只是业务逻辑代码，不会和CSS，HTML等其他界面组件耦合。

3.微服务的缺点：

【1】分布式系统的复杂性

【2】多服务运维难度，随着服务的增加，运维的压力也增大

【3】系统部署依赖

【4】服务间通信成本

【5】数据一致性问题

【6】系统性能监控，集成测试等问题。

**Eurake的自我保护机制：**

1.产生原因：

某个时刻某一个微服务不可用了，eureka不会立刻清理，依旧会对该微服务的信息进行保存。

默认情况下，若eurekaServer在一定时间内没有接收到某个微服务实例的心跳，EurekaServer将会注销该实例（默认90秒），但当网络分区故障时，微服务和Eureka之间无法正常通信，以上行为可能变得非常危险，因为微服务本身其实是健康的，此时本不应该注销到这个微服务。

Eureka通过自我保护模式，来解决这个问题：当EurekaServer节点在短时间内丢失过多的客户端时（可能产生网络分区故障），那么这个节点就会进入自我保护模式，一旦进入该模式，EurekaServer就会保护服务注册表的信息，不在删除服务注册表中的数据（即不会销毁任何微服务），当网络故障恢复后，该Eureka Server 节点会自动退出自我保护模式。

在自我保护模式中，Eureka Server会保护服务注册表中的信息，不再销毁任何服务实例，当他接收到的心跳数重新恢复到阀值以上时，该Eureka server 节点就会自动退出自我保护模式，即宁可保留错误的服务注册信息，也不盲目的注销任何可能健康的服务实例。

禁用自我保护 模式：

eureka.serever.enable-self-preservation = false

**注意yml：**

**yml配置中，spring.application.name: 服务程序名**

**此配置对应于eureka的服务实例名称，集群负载均衡时候，每个服务应用名称必须一致。**

2.Eureka集群配置：

server端配置：

eureka:

instance:

hostname: eureka7001.com #eureka服务端的实例名称

client:

register-with-eureka: false

fetch-registry: false

service-url:

defaultZone: http://eureka702.com:7002/eureka/ , http://eureka703.com:7003/eureka/

客户端配置：【生产者为例】

eureak:

client:

service-url:

defaultZone: http://eureka701.com:7001/eureka/, http://eureka702.com:7002/eureka/, http://eureka703.com:7003/eureka/

instance:

instance-id: microservicecloud-8001 #自定义服务名称信息（注册的id信息）

prefer-ip-address: true

**Ribbon负载均衡（Feign负载均衡）：客户端负载均衡组件**

**注意：**

**Ribbon和eureka整合之后，可以直接调用服务而不在关心地址和端口号**

**Ribbo调用是：Ribbon+RestTemplate**

1.实操

【1】客户端(消费者)pom的maven依赖

<!-- ribbon相关依赖 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-ribbon</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

【2】添加eureka的yml配置：

eureka:

client:

register-with-eureka: false

service-url:

defaultZone: http://eureka.7001.com/eureka/,http://eureka.7002.com/eureka/.http://eureka.7003.com/eureka/

【3】代码实例：【默认依旧是轮询算法】

@Configuration

public class Config {

@Bean

@LoadBalanced //基于Ribbon的负载均衡

public RestTemplate restTemplate(){

return new RestTemplate();

}

@Bean

public IRule roundRobinRule(){ //当要配置新的算法时，就在此配置类添加，以覆盖默认的算法

return new RoundRobinRule();

}

}

业务访问类：

@RestController

public class DeptController {

@Autowired

private RestTemplate template;

//private String REST\_URL\_PREFIX="http://localhost:8001";

private String REST\_URL\_PREFIX="http://SPRINGCLOUD-PROVIDER-DEPT";//修改使用微服务的名字进行

@RequestMapping("/consumer/getOne/{id}")

public Dept getOne(Integer id){

Map<String,Integer> map=new HashMap<>();

map.put("id", id);

return template.getForObject(REST\_URL\_PREFIX+"/getOne/{id}", Dept.class, map);

}

@RequestMapping("/consumer/add")

public boolean add(Dept dept){

return template.postForObject(REST\_URL\_PREFIX+"/add", dept, Boolean.class);

}

@SuppressWarnings("unchecked")

@RequestMapping("/consumer/getAll")

public List<Dept> getAll(){

return template.getForObject(REST\_URL\_PREFIX+"/getAll",List.class);

}

}

2.Robbin负载均衡算法实现

<https://github.com/Netflix/ribbon/blob/master/ribbon-loadbalancer/src/main/java/com/netflix/loadbalancer/RoundRobinRule.java>

**Ribbon自定义算法：**

**【1】主启动类添加@RibbonClient注解，主要是为在启动时，就去加载我们自定义的Ribbon配置类，使配置生效**

@RibbonClient（name=“MICROSERVER-DEPT” ,configuration=MySelfRule.class）

**注意：**

**自定义的算法类，不能放在@ComponentScan所扫描的当前包下以及子包下，否则自定义的配置类将会被所有的Ribbon客户端所共享，从而达不到特殊定制的目的【即不能和主启动类在统同一包下】**

【2】新建包名，同时添加自定义规则类

@Configuration

public class MyRule{

@Bean

public IRule randomRule(){

return new RandomRule();

}

}

**Feign相关知识**

Feign是一个声明式的WebService客户端，使用Feign能让编写WebService客户端更加便捷。

由来：目前大部分开发习惯了面向接口编程，eg：webService接口，Dao接口，这是大家的规范

方便多渠道开发，适应其他客户的开发习惯，采用统一的面向接口编程

【1】通过微服务的名字获取调用地址（spring.application.name）

【2】通过接口+注解，获得调用方的调用地址

1. 引入依赖（客户端的pom文件）

<!-- feign相关 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-feign</artifactId>

</dependency>

<!-- ribbon相关依赖 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-ribbon</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

【2】对api工程进行修改，就对接口进行修改，添加@FeignClien注解，同时指定要对应的微服务名称value【spring.application.name】

@FeignClient(value="SPRINGCLOUD-PROVIDER-DEPT-8001")

public interface DeptService {

@RequestMapping(value="/dept/getOne/{id}",method=RequestMethod.GET)

public Dept get(@PathVariable("id") int id);

@RequestMapping(value="/dept/list",method=RequestMethod.GET)

public List<Dept> list();

@RequestMapping(value="/dept/addOne",method=RequestMethod.POST)

public boolean add(Dept dept);

}

【3】修改业务访问类：controller层

面向接口编程

eg;

@Controller

public class deptController{

@Autowire

public DeptService deptService;

@RequestMapping(value="/dept/list",method=RequestMethod.GET)

public List<Dept> list(){

deptService. list();

}

}

**【4】 主启动类添加@EnableFeignClients(basePackages= {"com.bao.springcloud"})注解**

小结：

Feign通过接口的方法调用Rest服务，该请求发送至Eureka服务器，通过Feign直接找到服务接口，由于在进行服务调用的时候，融合ribbon技术，所以也支持负载均衡算法。

**Hystricx相关知识**

1.服务雪崩：

多个微服务之间调用的时候，假设服务A调用服务B和服务C，微服务B和微服务C又调用其他服务。这就是所谓的扇出，若扇出的某个连路上的某个微服务的调用响应时间过长或者不可用，对微服务A的调用就会 占用越来越多的系统资源。进而引起系统崩溃，所谓的雪崩效应。

对于高流量的应用来说，单一的后端依赖可能导致所有服务器上的所有资源都在几秒中内饱和。Hysricx是一个用于处理分布式系统的延迟和容错的开源库，在分布式系统里，许多依赖不可避免的会调用失败，eg:超时，异常等，Hystricx能够保证一个依赖出问题的情况下，不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性。

2.Hystricx本身是一种开关装置，当某个服务单元发生故障之后，通过断路器的故障监控，向调用方法返回一个符合预期的，可处理的备选响应（fallBack），而不是长时间的等待或者抛出调用方无法处理的异常。这就保证了服务调用方的线程不会被长时间的占用，避免了故障在分布式系统中蔓延，乃至雪崩。

**2.服务熔断**

熔断机制，是应对雪崩效应的一种微服务链路保护机制，当扇出的链路的某个微服务不可用或相应时间太长，会进行服务降级，进而熔断该节点微服务的调用，快速返回“错误”的相应信息，当检测到该节点微服务调用相应正常后恢复调用链路，在springCloud中，熔断机制是通过Hystricx实现，Hystrix会监控微服务之间的调用情况，当失败的调用达到一定的阀值， 缺省是5秒内20次调用失败就会启动熔断机制，

熔断机制的注解是： @HystrixCommand

@HystrixCommand就是对报异常后的方法处理。

一旦抛出异常，会自动调用HystrixCommand标好的fallBackMethod调用类的指定的方法。

实操：

【1】修改Pom文件，在provider的pom文件修改：

<!-- hystrix支持 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

【2】在provider端controller层指定的方法添加@HystrixCommand注解，调用的consumer采用RestTemplate访问

@RestController

public class DeptController {

@Autowired

DeptService service;

/\* @Autowired

private DiscoveryClient client;\*/

@GetMapping("/dept/findone/{id}")

@HystrixCommand(fallbackMethod="preocessHystrix\_Get")

public Dept getOne(@PathVariable("id") Integer id){

Dept dept=service.findOne(id);

if(null==dept){

throw new RuntimeException("该ID"+id+"没有对应的信息");

}

return dept;

}

public Dept preocessHystrix\_Get(@PathVariable("id") Integer id){

return new Dept().setId(id).setDept\_name("该ID没有对应的部门名称").setData\_source("没有对应的数据源信息");

}

}

【3】在provider主启动类上添加注解@EnadbleCircuitBreak即可

**3.服务降级**

【1】含义就是指：整体资源快不够了，先将某些服务停掉，待避过高峰期，在开启回来。

注意：

服务降级处理是在客户端完成的，与服务端没有关联

一般是从整体的负荷去考虑，当某个服务熔断之后，服务器将不再被调用，此时客户端可以自己准备一个本地的fallback回调，返回一个缺省值，虽然服务水平下降，但好歹可用，总比直接挂掉强。

【2】实操：

1.修改api工程，在此处添加一个实现了fallbackFactory接口的实现类，然后在对应的添加每个接口方法的回调方法（用于熔断）

@Component //这个注解一定要加

public class DeptServiceFallbackFactory implements FallbackFactory<DeptService> {

@Override

public DeptService create(Throwable cause) {

return new DeptService(){

@Override

public Dept get(int id) {

return new Dept().setId(id).setDept\_name("该ID没有对应的部门名称").setData\_source("没有对应的数据源信息");

}

@Override

public List<Dept> list() {

return null;

}

@Override

public boolean add(Dept dept) {

return false;

}

};

}

}

2.在对应修改接口，指定接口方法添加注解@**FeignClient，并指定fallbackFactory**

**这样避免了耦合（熔断和controller层）**

@FeignClient(value="SPRINGCLOUD-PROVIDER-DEPT-8001",fallbackFactory=DeptServiceFallbackFactory.class)

public interface DeptService {

@RequestMapping(value="/dept/getOne/{id}",method=RequestMethod.GET)

public Dept get(@PathVariable("id") int id);

@RequestMapping(value="/dept/list",method=RequestMethod.GET)

public List<Dept> list();

@RequestMapping(value="/dept/addOne",method=RequestMethod.POST)

public boolean add(Dept dept);

}

**HystrixDashboard相关【服务监控】**

**1.引入Maven依赖**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>

</dependency>

**2.修改yml，指定端口**

**3.修改主启动类，添加注解@EnableHystrixDashboard**

访问:http://localhost:9001/hystrix

**Zuul路由网关相关**

1.含义

Zuul包含了对请求的路由和过滤两个最主要的功能：

其中路由功能负责将外部请求转发到具体的微服务实例上，是实现外部访问统一入口的基础，而过滤功能则负责对请求的处理过程进行干预，是实现请求校验，服务聚合等功能的基础。

Zuul和eureka进行整合，将Zuul自身注册为Eureka服务治理下的应用，同时从Eureka中获得其他微服务的消息，也即以后的访问微服务都是通过Zuul跳转后获得。

注意：Zuul最终还是会注册到Eureka上。

提供=代理+路由+过滤

2.实操

【1】添加GAV坐标：

<!-- zuul路由网关 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zuul</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

【2】修改yml

server:

port: 9527

spring:

application:

name: microservicecloud-zuul-gateway

eureka:

client:

service-url:

defaultZone: http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka,http://eureka7003.com:7003/eureka

instance:

instance-id: gateway-9527.com

prefer-ip-address: true

zuul:

#ignored-services: microservicecloud-dept #限制使用微服务名称访问，只能通过网关访问

prefix: /atguigu #添加访问前缀，限制必须带上该前缀访问

ignored-services: "\*" #批量限制访问，全部只能通过网关实现

routes:

mydept.serviceId: microservicecloud-dept #路由映射规则,映射某个微服务

mydept.path: /mydept/\*\*

info:

app.name: atguigu-microcloud

company.name: www.atguigu.com

build.artifactId: $project.artifactId$

build.version: $project.version$

外部访问地址：http:// gateway-9527.com:9527/mydept/list | http:// gateway-9527.com:9527/mydept/get/2

【3】修改主启动类，添加注解@EnableZuulProxy

**SpringCloud Config分布式配置中心相关：**

1.SpringCloud Config为微服务架构的微服务提供集中化的外部配置支持，配置服务器为各个不同的微服务应用的所有环境提供了一个中心化的外部配置。

2.SpringCloud config 分为服务端和客户端两个部分。

3.服务端也称为分布式配置中心，它是一个独立的微服务应用，用来连接配置服务器并为客户端提供获取配置信息，加密/解密等访问接口。

4.客户端则是通过指定的配置中心来管理应用资源，以及业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息，配置服务器默认采取git来存储配置信息，这样就有助于对环境配置进行版本管理，并且可以通过git客户端工具来方便的管理和访问配置内容。

其他客户端获取配置文件信息时，会自动获取新的配置，无需重启

【2】服务端实操配置

1.添加GAV坐标

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

<!-- 避免Config的Git插件报错：org/eclipse/jgit/api/TransportConfigCallback -->

<dependency>

<groupId>org.eclipse.jgit</groupId>

<artifactId>org.eclipse.jgit</artifactId>

<version>4.10.0.201712302008-r</version>

</dependency>

2.修改yml文件

server:

port: 3344

spring:

application:

name: microservicecloud-config

cloud:

config:

server:

git:

uri: git@github.com:zzyybs/microservicecloud-config.git #GitHub上面的git仓库名字

3.主启动类添加注解@EnableConfigServer

4.配置读取规则：

/{applicaiton}/{profile}[/{label}]

/{application}-{profile}.yml

/{label}/{application}-{profile}.yml

/{application}-{profile}.properties

/{label}/{application}-{profile}.properties

【3】客户端实操配置：

1.添加GAV坐标：

<!-- SpringCloud Config客户端 -->

<!-- SpringCloud Config客户端 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jetty</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>springloaded</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

2.新增bootstrap.yml,在根路径下创建，内容如下：

注意：

application.yml 是用户级别的资源配置项

bootstrap.yml 是系统级别的资源配置，优先级更高

SpringCloud会创建一个“BootStrap Context”，作为Spring应用的Application Context的父上下文，初始化的时候，“BootStrap Context” 负责从外部源加载配置属性并解析配置。这两个上下文共享一个从外部获取的Environment。

bootstrap属性有最高优先级，默认情况下，他们不会被覆盖。 bootstrap context 和 Application context有着不同的约定。

故新增一个bootstrap.yml文件，保证了Bootstrap context和Application Context配置的分离。

spring:

cloud:

config:

**name: microservicecloud-config-client** #需要从github上读取的资源名称，注意没有yml后缀名【配置文件】

**profile: test #本次访问的配置项**

**label: master #哪个分支**

**uri: http://config-3344.com:3344 #本微服务启动后先去找3344号服务，通过SpringCloudConfig获取GitHub的服务地址**

3.application.yml配置内容如下：

spring:

application:

name: microservicecloud-config-client

注意添加：远端git服务器配置文件信息，yml：

spring:

profiles:

active:

-dev

----

**概念分析：**

1.服务降级： 在高并发情况下，防止用户一直等待，使用服务降级方式（直接返回一个友好的提示给客户端，调用fallback（）,一般是在客户端处理）--针对用户体验和高并发限流

2.服务熔断：熔断的目的是为了保护服务，在高并发情况下，若达到一定极限（用户设置）或流量超出设置阀值，则直接拒绝访问，保护当前服务；（服务降级和服务熔断一起使用）

3.服务隔离：雪崩效应产生的原因：默认情况下，只有一个线程池维护所有的服务接口，若大量的请求访问同一个接口，达到tomcat线程池默认极限，可能会导致其他服务无法访问；

解决服务雪崩效应：使用服务隔离机制（线程池方式和信号量）

服务隔离的原理：每个服务（接口）都有自己的线程池，每个线程池相互独立互不影响；

4.服务限流：

**【1】滑动窗口计数器算法原理**

传统计数器实现：定义一个原子类，针对于某一个服务实现次数记录，一旦到达阀值之后，这时可以直接走服务降级（返回一个友好的错误提示）===》会存在临界问题

eg：场景

限制每分钟内只能接受10个请求，超过则拒绝访问；

将60秒分成6个窗口，每个窗口有独立的计数器，同时父窗口只能包含6个子窗口；所以每个10秒中移动到下一个子窗口，最终实现临界值包含在60秒内，即只接受60秒内的请求

**【2】令牌桶算法：**

优点：能接受突然高并发请求；

1.可以开启一个独立的线程往桶【ArrayBlockQueue】里面存放以固定速率【每秒生成2个令牌】存放令牌,但是桶内的令牌不能无效制增长；

2.客户端只有在获取令牌成功的时候才可以访问服务，否则执行降级处理；

**使用RateLimiter实现令牌桶限流**

RateLimiter是guava提供的基于令牌桶算法的实现类，可以非常简单的完成限流特技，并且根据系统的实际情况来调整生成token的速率。

通常可应用于抢购限流防止冲垮系统；限制某接口、服务单位时间内的访问量，譬如一些第三方服务会对用户访问量进行限制；限制网速，单位时间内只允许上传下载多少字节等。

下面来看一些简单的实践，需要先引入guava的maven依赖。

依赖：

<parent>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-parent</artifactId>

            <version>2.0.0.RELEASE</version>

      </parent>

      <dependencies>

            <dependency>

                  <groupId>org.springframework.boot</groupId>

                  <artifactId>spring-boot-starter-web</artifactId>

            </dependency>

            <dependency>

                  <groupId>com.google.guava</groupId>

                  <artifactId>guava</artifactId>

                  <version>25.1-jre</version>

            </dependency>

      </dependencies>

代码：

 @RestController

**public** **class** IndexController {

      @Autowired

      **private** OrderService orderService;

      // 解释：1.0 表示 每秒中生成1个令牌存放在桶中

      RateLimiter rateLimiter = RateLimiter.*create*(1.0);

      // 下单请求

      @RequestMapping("/order")

      **public** String order() {

            // 1.限流判断

            // 如果在500秒内 没有获取不到令牌的话，则会一直等待

            System.***out***.println("生成令牌等待时间:" + rateLimiter.acquire());

            **boolean** acquire = rateLimiter.tryAcquire(500, TimeUnit.***MILLISECONDS***);

            **if** (!acquire) {

                  System.***out***.println("你在怎么抢，也抢不到，因为会一直等待的，你先放弃吧！");

                  **return** "你在怎么抢，也抢不到，因为会一直等待的，你先放弃吧！";

            }

            // 2.如果没有达到限流的要求,直接调用订单接口

            **boolean** isOrderAdd = orderService.addOrder();

            **if** (isOrderAdd) {

                  **return** "恭喜您,抢购成功!";

            }

            **return** "抢购失败!";

      }

**[3].漏桶算法**

**漏桶作为计量工具（The Leaky Bucket Algorithm as a Meter）时，可以用于流量整形（Traffic Shaping）和流量控制（TrafficPolicing），漏桶算法的描述如下：**

一个固定容量的漏桶，按照常量固定速率流出水滴；

如果桶是空的，则不需流出水滴；

可以以任意速率流入水滴到漏桶；

**如果流入水滴超出了桶的容量，则流入的水滴溢出了（被丢弃），而漏桶容量是不变的。**

**令牌桶和漏桶对比：**

**令牌桶是按照固定速率往桶中添加令牌，请求是否被处理需要看桶中令牌是否足够，当令牌数减为零时则拒绝新的请求；**

**漏桶则是按照常量固定速率流出请求，流入请求速率任意，当流入的请求数累积到漏桶容量时，则新流入的请求被拒绝；**

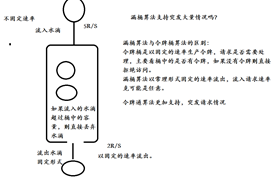
**令牌桶限制的是平均流入速率（允许突发请求，只要有令牌就可以处理，支持一次拿3个令牌，4个令牌），并允许一定程度突发流量；**

**漏桶限制的是常量流出速率（即流出速率是一个固定常量值，比如都是1的速率流出，而不能一次是1，下次又是2），从而平滑突发流入速率；**

**令牌桶允许一定程度的突发，而漏桶主要目的是平滑流入速率；**

两个算法实现可以一样，但是方向是相反的，对于相同的参数得到的限流效果是一样的。

另外有时候我们还使用计数器来进行限流，主要用来限制总并发数，比如数据库连接池、线程池、秒杀的并发数；只要全局总请求数或者一定时间段的总请求数设定的阀值则进行限流，是简单粗暴的总数量限流，而不是平均速率限流。



4.应用层限流：nginx+（http\_limit）

**应用级限流**

**限流总并发/连接/请求数**

对于一个应用系统来说一定会有极限并发/请求数，即总有一个TPS/QPS阀值，如果超了阀值则系统就会不响应用户请求或响应的非常慢，因此我们最好进行过载保护，防止大量请求涌入击垮系统。

如果你使用过Tomcat，其Connector其中一种配置有如下几个参数：

acceptCount：如果Tomcat的线程都忙于响应，新来的连接会进入队列排队，如果超出排队大小，则拒绝连接；

maxConnections：瞬时最大连接数，超出的会排队等待；

maxThreads：Tomcat能启动用来处理请求的最大线程数，如果请求处理量一直远远大于最大线程数则可能会僵死。

详细的配置请参考官方文档。另外如MySQL（如max\_connections）、Redis（如tcp-backlog）都会有类似的限制连接数的配置。

**限流总资源数**

如果有的资源是稀缺资源（如数据库连接、线程），而且可能有多个系统都会去使用它，那么需要限制应用；可以使用池化技术来限制总资源数：连接池、线程池。比如分配给每个应用的数据库连接是100，那么本应用最多可以使用100个资源，超出了可以等待或者抛异常。

**限流某个接口的总并发/请求数**

如果接口可能会有突发访问情况，但又担心访问量太大造成崩溃，如抢购业务；这个时候就需要限制这个接口的总并发/请求数总请求数了；因为粒度比较细，可以为每个接口都设置相应的阀值。可以使用[Java](http://lib.csdn.net/base/javaee)中的AtomicLong进行限流：

适合对业务无损的服务或者需要过载保护的服务进行限流，如抢购业务，超出了大小要么让用户排队，要么告诉用户没货了，对用户来说是可以接受的。而一些开放平台也会限制用户调用某个接口的试用请求量，也可以用这种计数器方式实现。这种方式也是简单粗暴的限流，没有平滑处理，需要根据实际情况选择使用；

**限流某个接口的时间窗请求数**

即一个时间窗口内的请求数，如想限制某个接口/服务每秒/每分钟/每天的请求数/调用量。如一些基础服务会被很多其他系统调用，比如商品详情页服务会调用基础商品服务调用，但是怕因为更新量比较大将基础服务打挂，这时我们要对每秒/每分钟的调用量进行限速；一种实现方式如下所示：

**平滑限流某个接口的请求数**

之前的限流方式都不能很好地应对突发请求，即瞬间请求可能都被允许从而导致一些问题；因此在一些场景中需要对突发请求进行整形，整形为平均速率请求处理（比如5r/s，则每隔200毫秒处理一个请求，平滑了速率）。这个时候有两种算法满足我们的场景：令牌桶和漏桶算法。Guava框架提供了令牌桶算法实现，可直接拿来使用。

Guava RateLimiter提供了令牌桶算法实现：平滑突发限流(SmoothBursty)和平滑预热限流(SmoothWarmingUp)实现。

**接入层限流**

接入层通常指请求流量的入口，该层的主要目的有：负载均衡、非法请求过滤、请求聚合、缓存、降级、限流、A/B测试、服务质量监控等等，可以参考笔者写的《使用Nginx+Lua(OpenResty)开发高性能Web应用》。

对于Nginx接入层限流可以使用Nginx自带了两个模块：连接数限流模块ngx\_http\_limit\_conn\_module和漏桶算法实现的请求限流模块ngx\_http\_limit\_req\_module。还可以使用OpenResty提供的Lua限流模块lua-resty-limit-traffic进行更复杂的限流场景。

limit\_conn用来对某个KEY对应的总的网络连接数进行限流，可以按照如IP、域名维度进行限流。limit\_req用来对某个KEY对应的请求的平均速率进行限流，并有两种用法：平滑模式（delay）和允许突发模式(nodelay)。

**ngx\_http\_limit\_conn\_module**

limit\_conn是对某个KEY对应的总的网络连接数进行限流。可以按照IP来限制IP维度的总连接数，或者按照服务域名来限制某个域名的总连接数。但是记住不是每一个请求连接都会被计数器统计，只有那些被Nginx处理的且已经读取了整个请求头的请求连接才会被计数器统计。

==========

**分布式Session的解决方案：**

1、使用cookie代替session

2. 使用nginx（反向代理）ip绑定，同一个IP只能访问在指定的机器上访问（没有负载均衡）

3. 使用数据库，效率不高

4.tomcat内置支持session的同步，同步会产生延迟

5.使用Spring-session框架，相当于把我们的session缓存到Redis

6.使用Token替代Session

**注意：**

**【1】sessionId是通过域名来保存的，浏览器时从对应的域名下【cookie的保存方式】获取sessionId**

====================

5个事件 –

-Connect (连接建立)

-Disconnect (连接断开）

– MessageReceived (消息已接受)

– Send(消息已发送)

– Exception Caught(异常) • 派发策略 –

5个事件共享同一个线程池 – Connect disconnect 使用独立线程池(size为1)

– 全部不派发线程池，IO线程处理

样例：

Thread Pool Size

• 设置多少线程合适？

有这样一个模块

cpu 计算时间 18ms （ running

）

查询数据库，网络 io 时间 80ms （ waiting

）

解析结果 2ms 如果服务器 2CPU ，大家看看这里多少线程合适

充分利用cpu资源： 线程数量=100/20\*2=10

从CPU角度而言 线程数量=（cpu时间+cpu等待时间）/cpu时间\*cpu数量 • 线程数量的设置就是由CPU决定的？ 有这样一个模块：

线程同步锁 ( 数据库事务锁 )50ms cpu 时间 18ms 查询数据库，网络 io 时间 80ms 解析结果 2ms 如果服务器有 2 个 CPU ，这个模块线程多少合适？

CPU计算为瓶颈，计算线程数量

线程数=(18 + 2 + 50 + 80) / 20 \* 2 = 15

• 以线程同步锁为瓶颈，计算线程数

线程数=(50 + 18 + 2 + 80) / 50 \* 1/1 = 3

Thread Pool Size

公式一： 线程数量=（线程总时间/瓶颈资源时间）

\* 瓶颈资源的线程并行数

准确的讲 瓶颈资源的线程并行数=瓶颈资源的总份数/单次请求占用瓶颈资源的份数

约束： 在计算的时候，对同一类资源的消耗时间进行合并

公式二： QPS=1000/线程总时间\*线程数

注意： 如果线程数不够，则QPS减少。 线程本身也要消耗资源，如果线程太多，同样QPS 会下降。

其他优化 – Lock free data structure – Buffer copy (Zero Copy) – JVM GC tuning – Context Switch – 同步转异步 – JavassistProxy改进JDK proxy • 注意性能的短板效应，避免过度优化 – 优化的代价，通常是牺牲未来的可能性。