## jvm

### 1.1 jvm(类加载机制、jvm调优)

类加载机制：加载、链接（验证、准备、解析）、初始化、使用、卸载

加载：通过包名、类名获取定义这个类的二进制字节流

把二进制字节流的静态存储结构转化为方法区可运行的数据结构

在方法区分配类的内存空间，设置初始值。

验证：jvm验证.class文件，确保符合jvm的需求。

准备：为类变量分配内存并设置初始值。

解析：将符号引用转换成直接引用。

初始化：赋值

使用：

卸载：gc垃圾回收。

jvm调优：调整内存大小（jvm初始内存和最大内存）

stack overflow：栈内存溢出

栈溢出解决方案：  
方案1.调整栈内变量内存分配，限制在1M内  
方案2.调整程序栈大小

### Jvm内存区域

程序计数器、虚拟机栈、本地方法栈、堆、方法区

程序计数器是一块很小的内存空间，它是线程私有的，可以认作为当前线程的行号指示器。

虚拟机栈：同计数器也为线程私有，生命周期与相同，就是我们平时说的栈。存储基本数据类型与对象引用的地址。

本地方法栈：执行native方法，可能调用底层的c或者c++，与底层进行交互。

堆：为对象实例以及数组分配内存地址。线程共享

方法区：用于存储已被虚拟机加载的类信息、常量、静态变量，线程共享。

### 1.3 Jvm常用内存参数设置：

-Xms ：初始堆内存大小

-Xmx：最大堆内存大小

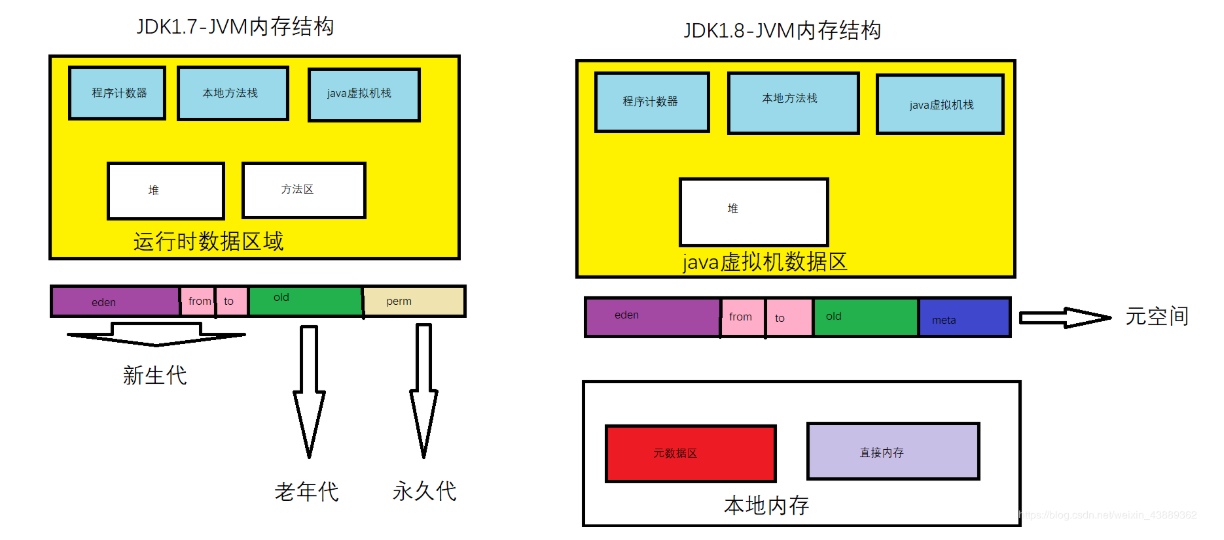
-Xmn：堆中Young区大小

-Xss：java每个线程的Stack(栈)大小

-XX:PermSize：持久代（方法区）的初始内存大小。

-XX:MaxPermSize：持久代（方法区）的最大内存大小。

### 1.4 Jvm在jdk1.7与jdk1.8的区别



**从图中我们可以看到，jdk1.7相较jdk1.8多了一个方法区（一般这么说，其实是永久代），而jdk1.8则多了一个元空间。**

首先，我们应该了解方法区只是JVM规范定义（即jdk1.7和jdk1.8都有方法区这个概念），而永久代和元空间则是方法区的实现

**永久代和元空间的区别：**

永久代：在运行时数据区域开辟空间实现方法区

元空间：在本地内存区域开辟空间实现方法区

**为什么要移除永久代呢？**

永久代中的元数据信息在每次FullGc时可能被收集，为永久代分配多少空间很难确定，超出指定空间容易造成内存泄漏

**元空间的特点：**

1.类及相关的元数据的生命周期与类加载器的一致

2.每个加载器有专门的存储空间

3.只进行线性分配

4.不会单独回收某个类

5.省掉了GC扫描及压缩的时间

6.元空间里的对象的位置是固定的

如果GC发现某个类加载器不再存活了，会把相关的空间整个回收掉

### 1.5 jvm栈、堆、方法区各存什么?

栈用来存储基本类型变量和对象的引用变量的地址。

堆用来存储new出来的对象和数组。

方法区存储方法和static变量。

### 1.6 GC分类

针对HotSpot VM的实现，它里面的GC其实准确分类只有两大种：

Partial GC：并不收集整个GC堆的模式

Young GC：只收集young gen的GC。

Old GC：只收集old gen的GC。只有CMS的concurrent collection是这个模式。

Mixed GC：收集整个young gen以及部分old gen的GC。只有G1有这个模式。

Full GC：收集整个堆，包括young gen、old gen、perm gen（如果存在的话）等所有部分的模式。

### 1.7 GC回收过程

Jvm的堆中分为新生代（占1/3）、老年代（占2/3）

新生代：

分为Eden区、Servivor区（from、to）；比例为8:1:1

新生代通过复制算法（Minor GC）回收；老年代通过标记-删除算法（Full GC或者Major GC）回收；

新生代的GC

GC开始时，对象只存在Eden区，经过Minor GC将存活的对象存到to 区，from区经过Minor GC，将仍存活的对象根据年龄判断去向，年龄达到一定阈值（默认是15），被移到老年代，没有达到阈值移到to区；将from与to互换角色，重复Minor GC，直到to 区被填满，将对象移到老年代。

老年代中的Major GC

**第一次标记**：如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链，那它将会被第一次标记；

**第二次标记**：第一次标记后接着会进行一次筛选，筛选的条件是此对象是否有必要执行finalize()方法。在finalize()方法中没有重新与引用链建立关联关系的，将被进行第二次标记。

将第二次标记过的对象卸载。

永久代(方法区)的GC

主要是收集常量池和对类型进行卸载； 通常情况下，这个区域的收集效率比较低。

常量的回收比较容易

类信息的回收需要满足的条件：

     1 . 该类所有的实例已经被回收，JVM中没有任何类的实例；

     2.  加载该类的ClassLoader被回收；

     3.  该类对应的java.lang.class没有地方引用

### 1.8 Jvm中对象创建过程

1. 当虚拟机遇到一条new指令时候，首先去检查这个指令的参数是否能**在常量池中能否定位到一个类的符号引用**（即类的带路径全名），并且检查这个符号引用代表的类是否已被加载、解析和初始化过，即验证是否是第一次使用该类。如果没有（不是第一次使用），那必须先执行相应的类加载过程（class.forname()）。

2. 在类加载检查通过后，接下来虚拟机将 **为新生的对象分配内存**。对象所需的内存的大小在类加载完成后便可以完全确定，为对象分配空间的任务等同于把一块确定大小的内存从Java堆中划分出来，目前常用的有两种方式，根据使用的垃圾收集器的不同使用不同的分配机制：

　　2.1. **指针碰撞（Bump the Pointer）**：假设Java堆的内存是绝对规整的，所有用过的内存都放一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅把那个指针向空闲空间那边挪动一段与对象大小相等的距离。

　　2.2. **空闲列表（Free List）**：如果Java堆中的内存并不是规整的，已使用的内存和空间的内存是相互交错的，虚拟机必须维护一个空闲列表，记录上哪些内存块是可用的，在分配时候从列表中找到一块足够大的空间划分给对象使用。

3. 内存分配完后，虚拟机需要将分配到的内存空间中的数据类型都 **初始化为零值**（**不包括对象头**）；

4. 虚拟机要 **对对象头进行必要的设置**，例如这个对象是哪个类的实例（即所属类）、如何才能找到类的元数据信息、对象的哈希码、对象的GC分代年龄等信息，这些信息都存放在对象的对象头中。

5. **调用对象的init()方法**,根据传入的属性值给对象属性赋值。

6. 在线程 **栈中新建对象引用** ，并指向堆中刚刚新建的对象实例。

总结：

1.查看常量池中是否存在这个类的符号引用，并且检查这个符号引用所代表的类是否已被加载、解析和初始化过。如果没有的话，就要执行相应的类加载过程。

2.为对象分配内存空间（指针碰撞、空闲列表）。

3.JVM将分配到内存空间的数据类型都初始化零值（int=0，String=null，boolean=false等）

4.JVM对对象值进行必要的设置（将一些信息[对象是哪个类的实例，对象的哈希码、对象的GC分代年龄等]存放到对象头）

5.调用对象的init()方法，给对象属性值赋值。

6.在栈中新建对象引用，并指向该对象。

## 线程

线程状态：新建、就绪、运行、阻塞、死亡；

阻塞状态分为：等待阻塞、同步阻塞、其他阻塞

了解的线程池：ExecutorService pool = Executors.newFixedThreadPool(10);

线程池中主线程与子线程的执行顺序：

先执行完子线程，再执行主线程。全部的子线程都执行完，主线程再关闭。

线程开启的是start方法，实际上执行的是run方法。

多线程同步：

synchronized 关键字

互斥锁

怎么新建一个线程

继承thread；实现Runnable接口。

两者的区别：

继承thread：单继承，不适合资源共享。

实现Runnable接口：增加程序的健壮性，代码可以被多个线程共享。

## 多线程 (同步、锁、线程池)

线程状态：新建、就绪、运行、阻塞、死亡；

阻塞状态分为：等待阻塞、同步阻塞、其他阻塞

线程池： newCachedThreadPool、newFixedThreadPool

ExecutorService pool = Executors.newFixedThreadPool(10); 创建固定大小的线程池

## 线程池的作用：减少内存开销；

线程池怎么加入线程：execute()方法；方法里新建线程

## 线程池各个参数的意义：

corePoolSize:最小线程数（就算是空闲状态，也会保持该数量线程

）

maximumPoolSize：最大线程数（线程的增长始终不会超过该值

）

long keepAliveTime：线程最大的存活时间

workerQueue：缓存队列

先 达到 corePoolSize，然后 优先放入队列，然后在到MaxPollSize;然后拒绝。

4种拒绝策略：

AbortPolicy （中止） 直接抛出异常

DiscardPolicy （抛弃） 直接抛弃新任务，不执行

DiscardOldestPolicy （抛弃最旧） 抛弃优先级最高的任务，来执行新任务

CallerRunsPolicy （调用者运行）

## java基本框架（spring、mybatis的工作原理）

spring:

Spring是一个轻量级的IoC和AOP容器框架.

springIOC的注入方式：构造器注入、setter方法注入、根据注解注入

AOP:一般称为面向切面，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块,减少系统中的重复代码，

降低了模块间的耦合度,可用于权限认证、日志、事务处理。

使用构造器注入的好处：

1. 保证依赖不可变（final关键字）
2. 保证依赖不为空（省去了我们对其检查）
3. 保证返回客户端（调用）的代码的时候是完全初始化的状态
4. 避免了循环依赖

AOP代理方式：静态代理、动态代理

静态代理：会在编译阶段将AspectJ(切面)织入到Java字节码中，运行的时候就是增强之后的AOP对象。

动态代理：AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，

并回调原对象的方法。

@Autowired和@Resource之间的区别

(1)@Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在。

(2)@Resource默认是按照名称装配注入的，只有当找不到与名称匹配的bean才会按照类型来装配注入。

BeanFactory与ApplicationContext的区别

BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。

ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。

声明式事务与编程式事务的区别：

声明式事务：加上@Transactional注解就可以，最细粒度到方法级别

编程式事务：使用TransactionTemplate，最细粒度到方法快级别

mybatis的工作原理：mybatis应用程序通过SqlSessionFactoryBuilder从配置文件中构建出SqlSessionFactory，然后，SqlSessionFactory的实例直接开启一个SqlSession，

再通过SqlSession实例获得Mapper对象并运行Mapper映射的SQL语句，完成对数据库的CRUD和事务提交，之后关闭SqlSession。

## spring是怎么注入bean的?

由Spring容器(比如 ApplicationContext)来实现,Spring启动时会把所有需要的bean扫描并注册到容器里，

在这个过程当中Spring会根据我们定义的bean之间的依赖关系来进行注入，依赖关系的维护方式有两种即XML配置文件或者注解，

Spring启动时会把这些依赖关系转化成Spring能够识别的数据结构BeanDefinition,并根据它来进行bean的初始化，依赖注入等操作。

## Spring事务

Spring事务其实说的是数据库事务

一般采用声明式事务，也就是注解 **@Transactional ，用在service层**

**@Transactional** 放在类上和方法上的区别：

方法上的**@Transactional 会覆盖类上的 @Transactional**

**@Transactional 的参数解释：**

@Transactional(propagation=Propagation.REQUIRED) : 有事务就处于当前事务中，没事务就创建一个事务

@Transactional(propagation=Propagation.NOT\_SUPPORTED)针对某个方法不开启事务

isolation=Isolation.DEFAULT：事务[数据库](http://lib.csdn.net/base/mysql)的默认隔离级别

readOnly=false：可写 针对 增删改操作

readOnly=true：查询使用只读

在Spring 的世界里面我们一般使用@Transactional 注解在对应方法上面声明为一个事务方法。  
但是在默认不写@Transactional(rollbackFor = Exception.class)默认回滚RuntimeException。

## Hibernate

一级缓存与二级缓存：

 一级缓存：基于Session的缓存，缓存内容只在当前session有效，session关闭，缓存内容失效。作用范围较小！ 缓存的时间短。

二级缓存：（SessionFactory缓存）

Hibernate提供了基于应用程序级别的缓存即为二级缓存，可以跨多个session，即不同的session都可以访问缓存数据。

注：session的生命周期(session什么时候会失效)

　 1. 服务器会把长时间没有活动的Session从服务器内存中清除，此时Session便失效。Tomcat中Session的默认失效时间为30分钟。

2. 调用Session的invalidate方法。

## Mybatis

Mybatis的一级、二级缓存：

一级缓存基于sqlSession默认开启,在操作数据库时需要构造SqlSession对象，在对象中有一个HashMap用于存储缓存数据。如果sqlSession执行了DML操作，并且提交到数据库，MyBatis则会清空SqlSession中的一级缓存。

当一个SqlSession结束后该SqlSession中的一级缓存也就不存在了。

二级缓存是mapper级别的缓存。使用二级缓存时，多个SqlSession使用同一个Mapper的sql语句去操作数据库，得到的数据会存在二级缓存区域，它同样是使用HashMap进行数据存储。相比一级缓存SqlSession，二级缓存的范围更大，多个Sqlsession可以共用二级缓存，二级缓存是跨SqlSession的。

## Mysql

Mysql的执行引擎：

innoDB：

优点：InnoDB是一个事务型的存储引擎, 提供了对数据库ACID事务的支持，并且实现了SQL标准的四种隔离级别。

缺点：不支持全文索引，而且它没有保存表的行数，当SELECT COUNT(\*) FROM TABLE时需要扫描全表。

Mysql事务的隔离级别：



Mysql的事务特性：

原子性：原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生；

一致性：事务前后数据的完整性必须保持一致；

隔离性：事务的隔离性是指多个用户并发访问数据库时，一个用户的事务不能被其它用户的事务所干扰，多个并发事务之间数据要相互隔离；

持久性：持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响。

Mysql中的脏读、幻读

脏读：

脏读发生在一个事务A读取了被另一个事务B修改，但是还未提交的数据。假如B回退，则事务A读取的是无效的数据

幻读：

幻读发生在当两个完全相同的查询执行时，第二次查询所返回的结果集跟第一个查询不相同。

不可重复读与幻读的区别：

不可重复读：不可重复读的重点是修改: 同样的条件, 你读取过的数据, 再次读取出来发现值不一样了。

幻读：幻读的重点在于新增或者删除 (数据条数变化)。同样的条件, 第1次和第2次读出来的记录数不一样。

Mysql默认的隔离级别是 可重复读；oracle默认的隔离级别是 未提交。

## 数据库优化（数据库连接池）

dbcp、c3p0

优化：创建索引、不查\*、exists代替in

锁机制（乐观锁、悲观锁）

数据库的执行计划：

## 数据库中索引为什么快？

本质是-平衡树（b树或b+树）索引中是包含一个表中列的值（或前缀的字符编码）和它的物理地址的值，通过二分法快速查找对应的值，通过地址查找到数据。

优势：查询快

劣势：创建、维护索引需要耗费时间；占用物理空间；对表的数据进行增删改时，索引也需要维护。

## Ehcache

Element与java程序是绑在一起的，java程序活着，它就活着。譬如，写一个独立程序放数据，再写一个独立程序拿数据，那么是拿不到数据的。只能在独立程序中才能拿到数据。

CacheManager：是缓存管理器，可以通过单例或者多例的方式创建，也是Ehcache的入口类。  
Cache：每个CacheManager可以管理多个Cache，每个Cache可以采用hash的方式管理多个Element。  
Element：用于存放真正缓存内容的。

Ehcache的缓存数据淘汰策略:

LFU：最少被使用，缓存的元素有一个hit属性，hit值最小的将会被清出缓存。

LRU：最近最少使用，缓存的元素有一个时间戳，当缓存容量满了，而又需要腾出地方来缓存新的元素的时候，那么现有缓存元素中时间戳离当前时间最远的元素将被清出缓存。

Ehcache的缓存数据过期策略:

Ehcache采用的是懒淘汰机制，每次往缓存放入数据的时候，都会存一个时间，在读取的时候要和设置的时间做TTL比较来判断是否过期。

存储类型：序列化的对象

## Redis

Redis属于独立的运行程序，需要单独安装后，使用JAVA中的Jedis来操纵。

Redis的存储类型：String,List,hash,set,sort set

Redis的优势：单线程，执行速度快

Redis的持久化：

Rdb: 保存redis某个时间点上的数据集。 缺点：可能会丢失一些实时数据。（将Reids在内存中的数据库记录定时 dump到磁盘上的RDB持久化）

Aof: 设置fsync策略，可以每秒一次；保证实时数据。(原理是将Reids的操作日志—主要是写操作以追加的方式写入文件)

Redis设置过期时间，过了过期时间，key会被删除吗？

aof是以记录命令（写操作）方式，所以如果还没过期，那么不会有变更的命令，如果过期，会在aof日志插入一条del命令。

rdb的话是内存快照方式，如果持久化时，key已经过期，那么不会持久化，如果在过期之前就已经持久化了，那么在恢复数据时，会判断key是否过期，如果过期不会导入。

Redis与ehcache的区别：

**redis是通过socket访问到缓存服务，效率比ecache低，比数据库要快很多，处理集群和分布式缓存方便，有成熟的方案。**

**ehcache直接在jvm虚拟机中缓存，速度快，效率高；但是缓存共享麻烦，集群分布式应用不方便。**

Redis的删除策略：

被动删除：key过期后不会立即删除，再次访问此key的时候，会检查是否过期，过期即删除。

主动删除：key过期后立即删除，这样对cpu压力比较大。

定时删除：定时统一删除过期的key。这个是折中方案。

## redis设置了过期时间，过了过期时间后，会删除key吗？

aof是以记录命令方式，所以如果还没过期，那么不会有变更的命令，如果过期，会在aof日志插入一条del命令。

rdb的话是内存快照方式，如果持久化时，key已经过期，那么不会持久化，如果在过期之前就已经持久化了，那么在恢复数据时，会判断key是否过期，如果过期不会导入。

## jdk1.8特性：

lamdba表达式

default关键字：在接口中实现方法

Date API更新 : DateTimeFormatter.ofPattern("yyyy-MM-dd hh:mm:ss"); 配合LocalDateTime使用

调用方式： ：： 例如：A a = A::new; A::aa()[aa方法是A类中的类方法]

## HashMap

hashMap怎么put、get

put:首先通过key获取hash值，这个是数组的下标，把entry元素放到数组对应的位置，这个entry元素包括key、value、还有一个指针，指向下一个entry元素。（如果下标一致，在对应的位置向下插入，指针指向原有位置的entry）

get:通过key获取hash值，找到数组对应下标的位置，再根据key找到对应的元素。

Jdk1.7：数组+链表

Jdk1.8：数组+链表/红黑树

当链表的长度大于8时，将链表转化成红黑树

hashMap源码：一个Node<K,V>，属性：int hash🡪代表数组下标位置，K key，V value，Node<K,V> next 指向的下一个节点

执行顺序：先获取key对应的hashcode值，在新建一个数组（长度为16），然后根据hash & (数组长度-1) 🡪与运算，获得数组的下标位置，将hashMap放在对应的位置，如果数组的该位置上有元素，比较key值是否相等，相等就直接覆盖value，不相等链表就增加一个元素，当链表元素大于8时，转换成红黑树；当数组长度大于12时，就执行resize方法进行扩容，新建长度为32的数组，根据上述方法得到hashmap对应的数组下标，依次对号入座就行。

HashMap中数组长度为什么是16？

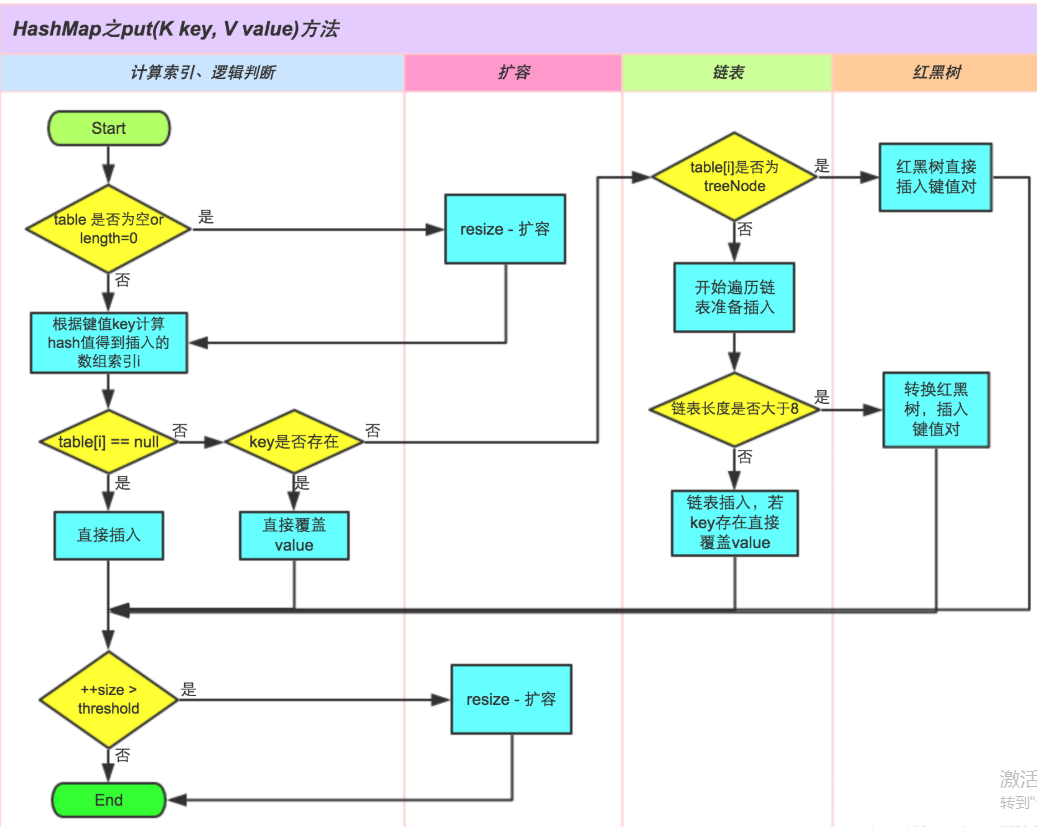
总结： 1 减少hash碰撞 (通过key的hashcode值和数组长度-1的与运算得到数组下标)

         2 提高map查询效率

         3 分配过小防止频繁扩容

         4 分配过大浪费资源

链表长度为8时变成红黑树，长度为6时又变成链表。



HashMap与hashTable、concurrentHashMap的区别：

hashMap线程不安全；初始化容量16；键值对可以为**空**；性能高；位运算符；最大扩容2(30)。

hashTable线程安全；初始化容量**11**；键值对不能为空；每个方法前都有 synchronized 关键字，使用一把锁处理并发问题，当有多个线程访问时，需要多个线程竞争一把锁，导致阻塞。

concurrentHashMap 中用cas算法实现乐观锁（compareAndSwapInt（目标值，原值，写入的新值）），cas是非阻塞算法的一种常见实现。

## Shiro

shiro的权限控制执行流程: application -----> subject -----> securityManager -----> realm -----> 安全数据;。

要控制按钮级别的权限的话，Jsp页面可以使用shiro标签；前后端分离的话可以把本用户对应角色的权限拿到前端进行比较。

Springboot中通过ShiroFilterFactoryBean来创建shiroFilter ,对访问的url 的访问进行控制。

Shiro集成单点：

搭建Cas-server服务器，在cas-client(实际上就是shiro项目)，配置登录路径，直接跳转到cas-server中登录，返回一个带有user信息的ticket，cas-client根据ticket信息进行授权。

## 中间件

rabitMQ：

a:生产者 b:消费者

5种队列：简单模式（1--1） work模式（1-多）、发布-订阅模式（生产者首先将消息发送到交换机，交换机绑定多个队列，然后被消费）

路由模式：

主题模式：

生产者发送消息到队列 1.获取连接以及mq通道、2.从连接中创建通道、3.创建队列、4.存入信息、5.关闭通道和连接

消费者从消息队列消费 1.获取连接以及mq通道、2.从连接中创建通道、3.定义队列的消费者、4.监听队列、5.获取消息、6.关闭通道和连接

kafka:

生产：1.创建生产者者（声明集群地址，设置key、value的序列化，声明broker地址） 2.根据topic发送消息

消费：1.创建消费者（声明集群地址，设置key、value的序列化，声明broker地址，声明group） 2.通过topic获取信息

## Springboot中的各个注解的解释：

@Service: 注解在类上，表示这是一个业务层bean

@Controller：注解在类上，表示这是一个控制层bean

@Repository: 注解在类上，表示这是一个数据访问层bean

@Component： 注解在类上，表示通用bean ，value不写默认就是类名首字母小写

@Autowired：按类型注入.

@Configuration

@Resource默认按照名称方式进行bean匹配。

@RestController与@Controller的区别

@RestController注解相当于@ResponseBody ＋ @Controller合在一起的作用。直接返回json数据，无法返回页面

@Controller可以返回指定的jsp或html页面，要返回json数据的话，要加上@ResponseBody注解。

## Tomcat的配置：

修改内存大小：

Windows下是在bin目录的catalina.bat文件中，添加

JAVA\_OPTS='-Xms256m -Xmx512m'，初始内存是256m，最大内存是256m

Linux下是修改bin目录下的catalina.sh文件。

## Springboot中怎么进行面向切面的使用：

写一个类，上面写上

*@Aspect注解*

*@Component*

对包下的所有类都进行切面控制

*@Pointcut("execution (\* com.topsec.hg..controller..\*.\*(..)) && !execution(\* com.topsec.hg..controller.AuditController.\*(..))")  
public void webLog() {  
}*

在方法中写上环绕

*@Around("webLog()")*

Spring AOP @Around @Before @After的区别

@Before是在所拦截方法执行之前执行一段逻辑。@After 是在所拦截方法执行之后执行一段逻辑。@Around是可以同时在所拦截方法的前后执行一段逻辑。

Aspect类内部的advice按照以下的顺序进行执行

@Around--@before—Method--@Around--@After--@AfterReturning

## 日志级别：

优先级从高到低

Error—Warn—Info--Debug

Logback.xml的配置：

<appender><filter><level></level></filter></>标签和<root><appender-ref ref=’appender里的name’></>标签

## 对springboot-starter的理解：

作用吧：

一键搞定所依赖的jar包，只需写 比如：mybatis-spring-boot-starter

自动加入mybatis的所有依赖，并且解决版本冲突

自动装配：

Springboot项目启动时，starter会自动加载所有的bean(类上标有@Configuration)。

## Es与关系型数据库的区别

## Hashcode与equals的区别

Hashcode相等，equals不一定相等

Equals相等，说明是同一个对象；hashcode不一定相等

为了便于好记，hashcode相当于人名，人相当于对象；equals相等，是同一个人，名字相同；名字相同的人有多个。

## springCloud分布式

eureka：服务注册中心

eureka-server：注册server服务

eureka-client：注册client服务（通过RestTemplate调用server服务的controller）

zuul：配置路由服务，相当于nginx中的反向代理

ribbon：负载均衡的作用，多个服务端可以服务名相同，端口号不同，发布到服务注册中心会显示同一行的两个。

config-server：用来从git/svn上下载文件内容，发布到eureka。

config-client：从config-server中获取在远端的配置文件内容。

## nginx

反向代理配置：

server {

listen 80;

server\_name localhost;

location / {

proxy\_pass http://OrdinaryPolling;

index index.html index.htm index.jsp;

}

}

通过自定义的url（host:port）,映射到服务的url以及开始页面，这样地址栏输入自定义的url，直接跳转到真正服务的页面。

## spring中的各个模式的使用场景

1. 单例模式

单个实例引用，单例模式有且只有一个类实例，并且提供唯一的访问入口。不应该由人来控制，该由代码来限制，强制单例。

使用场景：Spring中的bean都是单例模式 ，数据库连接池一般也采用单例模式。

优势：避免了对共享资源的多重占用。对于一些需要频繁创建和销毁的对象来说可以提高系统的性能。

2.代理模式：

代理模式（Proxy）为另一个对象提供一个替身或占位符以控制对这个对象的访问，其实用最简单的方式来解释代理模式，那就是中介，客户不想直接引用对象，这个“中介”也就是代理对象就起到了一定的作用。在客户端和目标端之间起到一定的作用。

避免直接调用对象，客户端通过代理类调用来调用目标端的方法，并对此方法进行增强。

静态代理与动态代理：

静态代理（实现方式）：目标类与代理类同时实现一个接口，在代理类中实例化目标类，并调用增强后的方法。

jdk动态代理（实现方式）：代理类通过实现InvocationHandler 接口，重写invoke 方法，来实现调用增强后的方法。

jdk动态代理总结：通过类加载器拿到真实实现类和真实实现类接口的字节码文件，通过代理类来调用真实实现类的方法。

3.工厂模式：

使用场景：

1．工厂模式的目的是为了实现解耦，将对象的创建和使用分开，即应用程序将对象的创建和初始化职责交给工厂对象。

2．降低代码重复。如果对象B的创建过程比较复杂，并且很多地方都用到了，那么很可能出现很多重复的代码，通过统一将创建对象B的代码放到工厂里面统一管理，可以减少代码的重复率，同时也方便维护。

在spring 中可以用工厂模式创建 bean ，用Factory-bean 动态工厂创建bean

Spring使用工厂模式配置bean的三种方法

1. 通过静态工厂方法 StaticCarFactory。
2. 通过实例工厂方法 InstanceCarFactory。
3. 通过实现 FactoryBean 接口，通过factorybean 来配置bean。

## Git 与 SVN 的区别

Git是分布式的，svn不是分布式的。

Git 可以不依赖网络做任何事情，分支是存在本地的，只要不合并主干，可以随便玩；

Svn 必须依赖网络才能提交，分支是存在远端的。

## Linux常用命令

查看linux磁盘占用情况： df -lh

## 时间复杂度

顺序查找：O(n)

二分法查找：O(logn)

## Java中静态方法与非静态方法的调用

静态方法是属于类的，即静态方法是随着类的加载而加载的，在加载类时，程序就会为静态方法分配内存，而非静态方法是属于对象的，对象是在类加载之后创建的，也就是说静态方法先于对象存在，当你创建一个对象时，程序为其在堆中分配内存，一般是通过this指针来指向该对象。静态方法不依赖于对象的调用，它是通过‘类名.静态方法名’这样的方式来调用的。而对于非静态方法，在对象创建的时候程序才会为其分配内存，然后通过类的对象去访问非静态方法。因此在对象未存在时非静态方法也不存在，静态方法自然不能调用一个不存在的方法。

## 过滤器与拦截器的区别

1．拦截器不依赖与servlet容器，过滤器依赖与servlet容器。

2．**拦截器只能对action请求起作用，而过滤器则可以对几乎所有的请求起作用。**

**3.深度不同：Filter只在Servlet前后起作用，而拦截器能够深入到方法前后。**

4. 在action的生命周期中，拦截器可以多次被调用，而过滤器只能在容器初始化时被调用一次。

## 批处理

jdbc的批处理：

Statement.addBatch(sqlArr[i]);

Statement.executeBatch();

mybatis的批处理：

Bath Update方法

<update id="batchUpdate " parameterType="java.util.List">

<foreach collection="list" index="index" item="item" separator=";">

INSERT INTO t\_test (

        id,

        name,

        del\_flag

        )

        VALUES

        <foreach collection ="list" item="test" separator =",">

         (#{test.id}, #{test.name}, #{test.delFlag})

        </foreach >

    </insert>

  </foreach>

</update>

Springboot+jpa的批处理：

注入

**@PersistenceContext**

EntityManager em;

通过调用 em.persist(*list*.get(i)); //批量插入

em.merge(*list*.get(i)); //批量修改

## Spring AOP中Aspect方法执行顺序

@before around

@before

-- sayHello() -- 执行方法

@after around

@after

@after return

@around 之前之后的顺序是以 JoinPoint.proceed() 为分割的。

## 获取linux用户操作历史

USER\_IP=`who -u am i 2>/dev/null| awk '{print $NF}'|sed -e 's/[()]//g'`

HISTFILESIZE=4000

HISTSIZE=4000

HISTORYFILE=/var/log/history.log

HISTTIMEFORMAT="%F %T ${USER\_IP} `whoami` "

export HISTTIMEFORMAT

export PROMPT\_COMMAND='{ date "+%Y-%m-%d %T $(history 1 | { read x cmd; echo "ip=172.21.5.181,user=$USER,ppid=$PPID,pwd=$PWD,from=$SSH\_CLIENT,command=$cmd"; })"; } >> $HISTORYFILE'

设置 HISTTIMEFORMAT 之后，在控制台输入 history 就会出现类似历史的输出。

设置 PROMPT\_COMMAND 之后，会将linux 的操作历史追加到 history.log中。

## Logstash

启动命令：./bin/logstash -f ../conf/logstash.conf &

logstash.conf内容：

input {

file {

path => "/var/log/history.log"

}

}

filter {

grok {

match => {"message" => "ip=(?<destIp>\S+),user=(?<username>\w+),ppid=(?<ppid>\d+),from=(?<srcIp>\S+)\s+(?<srcPort>\d+)\s+(?<destPort>\d+),pwd=(?<pwd>\S+),command:(?<command>.\*)"}

}

ruby {

code => "event.set('LOAD\_TIME', event.get('@timestamp').time.localtime.strftime('%Y-%m-%d %T'))"

}

}

output {

elasticsearch {

hosts => ["http://172.21.5.181:9200"]

index => "systemlog\_%{+YYYY-MM-dd}"

#user => "elastic"

#password => "changeme"

}

}

主要分为input , filter, output 三部分。

Logstash中 filter 的grok中的匹配用到的是正则匹配。

Ruby 是将默认的timestamp时间字段以固定格式赋值给要新建的字段。

## Springboot+jpa多表查询，并将查询结果转换成DTO

//注入EntityManager  
**@PersistenceContext  
private EntityManager** entityManager;

//拼sql  
**String** sql = **"SELECT c.name,b.parent\_id from testc c ,test\_b b where b.id=c.id"**;

//使用注入EntityManager进行查询,并将结果转换成DTO,前提是DTO的字段名称与查询到的一致  
**List**<**TestManyFieldModel**> list = entityManager.createNativeQuery(sql).getResultList();

## Springboot+Mybatis的多表查询

写两个resultMap即可

例子：（学生与班级）

**ClazzMapper.xml**

|  |
| --- |
| <resultMap type="com.gec.domain.Clazz" id="baseClazzResultMap">  <id column="id" property="clazzId"/>  <result column="clazz\_name" property="clazzName"/>  <result column="code" property="code"/>  </resultMap>  <resultMap type="com.gec.domain.Clazz" id="queryClazzListResultMap" extends="baseClazzResultMap">  <collection property="studentList" javaType="ArrayList"  column="id" ofType="com.gec.domain.Student"  select="com.gec.mapper.StudentMapper.queryStudentByClazzId"  >  </collection>  </resultMap>  <select id="queryClazzList" resultMap="queryClazzListResultMap">  select \* from tbl\_clazz;  </select> |

**StudentMapper.xml**

|  |
| --- |
| **<**resultMap type="com.gec.domain.Student" id="baseStudentResultMap">  <id column="id" property="studentId"/>  <result column="name" property="name"/>  <result column="sex" property="sex"/>  <result column="age" property="age"/>  </resultMap>  <resultMap type="com.gec.domain.Student" id="queryStudentListResultMap" extends="baseStudentResultMap">  <association property="clazz" javaType="com.gec.domain.Clazz">  <id column="cls\_id" property="clazzId"/>  <result column="clazz\_name" property="clazzName"/>  <result column="code" property="code"/>  </association>  </resultMap>  <select id="queryStudentList" resultMap="queryStudentListResultMap">  SELECT s.\*,c.id AS cls\_id,c.clazz\_name,c.code  FROM tbl\_student s,tbl\_clazz c WHERE s.clazz\_id=c.id;  </select>  <!-- 根据班级的id，获取学生列表 -->  <select id="queryStudentByClazzId" resultMap="baseStudentResultMap">  select \* from tbl\_student where clazz\_id=#{id};  </select> |

多对多的多表查询里面改成 collection就行了。

## Java中的异常

Throwable 是所有 Java 程序中错误处理的父类 ，有两种资类： Error 和 Exception 。

Error ：表示由 JVM 所侦测到的无法预期的错误，由于这是属于 JVM 层次的严重错误 ，导致 JVM 无法继续执行，因此，这是不可捕捉到的，无法采取任何恢复的操作，顶多只能显示错误信息。

Exception ：分为runtime exception 和 checked exception 。

runtime exception ：运行时异常，当出现这样的异常时，总是由虚拟机 接管。一般常见的NullPointerException 异常。

checked exception ：经常遇到的 IO 异常，以及 SQL 异常都是这种异常。需要捕获并进行处理。

## 基本数据类型与引用数据类型的区别

**声明变量时不同的内存分配：**

基本类型值：存储在栈中的简单数据段，它们的值直接存储在**变量访问的位置**。

引用类型值：存储在堆中的对象，**存储在变量处的值是一个指针，指向存储对象的内存地址。**

**参数传递的不同：**

基本类型值：只是把变量里的值传递给参数，之后参数和这个变量互不影响。

引用类型值：对象变量它里面的值是这个对象在堆内存中的内存地址，因此**它传递的值也就是这个内存地址。**

## Jpa、hibernate、MyBatis的区别

JPA是规范，Hibernate是框架，JPA是持久化规范，而Hibernate实现了JPA。

如果抛开JPA直接使用Hibernate的注解来定义一个实例，很快发现了几个问题：

1.jpa中有Entity, Table，hibernate中也有，但是内容不同

2.jpa中有Column,OneToMany等，Hibernate中没有，也没有替代品

想用hibernate注解, 是一定会用到jpa的。

Hibernate的设计理念是完全面向POJO的，而MyBatis不是。

Hibernate基本不再需要编写SQL就可以通过映射关系来操作数据库，是一种全表映射的体现；程序员不用精通SQL，只要懂得操作POJO就能够操作对应数据库的表。

在管理系统时代，首先是实现业务逻辑，然后才是性能，所以Hibernate在当时是主流。

在移动互联网时代，MyBatis是首选，不屏蔽SQL，程序员可以自己制定SQL规则，能更加精确定义SQL，从而优化性能。更符合移动互联网高并发，大数据，高性能，高响应的要求。

## 补充shiro

1. 在shiroFilter过滤器中配置登录的url、登录成功的url、以及无权限的url。

bean.setLoginUrl(**"/login"**);  
bean.setSuccessUrl(**"/index"**);  
bean.setUnauthorizedUrl(**"/403"**);

1. 在shirofilter中配置访问权限，过滤静态资源，以及/login、/index、/logout，其他的都设置为authc。

## Activity工作流

首先设置自动建立23张表，都是act\_开头的：

act\_re前缀的包含了流程定义的流程静态资源。

Act\_hi为前缀的包含了流程历史记录数据。

Act\_ru为前缀的这些运行时的表，包含流程实例，任务，变量，异步任务，等运行中的数据。

几个service:

比如TaskService，是activiti的任务服务类。可以从这个类中获取任务的信息。

而HistoryService，则是activiti的查询历史信息的类。在一个流程执行完成后，这个对象为我们提供查询历史信息。

1. 首先画流程图，首先一个流程包括bpmn文件和一个png图片
2. 将文件和图片一起部署：

repositoryService.createDeployment()

加载文件的key、文件、图片，一起部署，部署成功后会在act\_re\_procdef表中新建一条数据。

1. 开启任务实例（提交申请）
2. 处理任务

首先查询待办任务：通过查询act\_ru\_task、act\_ru\_variable查询某人的待办任务

select t.\* from act\_ru\_task t where t.ASSIGNEE\_='employee'

or t.EXECUTION\_ID\_ in (select v.EXECUTION\_ID\_ from act\_ru\_variable v where v.TEXT\_='employee')

查询到的是 employee 的待办任务

在处理任务

1. 一级一级处理，直到完结

期间有退回的任务，没有退回的任务是没有实际意义的：

用 exclusiveGateway 标签来设置同意、退回的操作。

提交给单人处理的在bpmn中设置assignee=${assign}；还有多人处理的，在图中设置 Multi Instance🡪colleection=${users}

多人提交时往map中设置list即可。到时候会在act\_re\_variable 中添加两条数据，与act\_ru\_task是以execution\_id相关联的。

## Kafka与rabbitMQ的区别

RabbitMQ是由内在高并发的erlanng语言开发，用在实时的对可靠性要求比较高的消息传递上。

kafka是采用Scala语言开发，它主要用于处理活跃的流式数据,大数据量的数据处理上

rabbitMQ的工作模式：

简单模式（1-1）

Work模式（1-多）一个生产者，多个消费者

发布-订阅模式（生产者首先将消息发送到交换机，交换机绑定多个队列，然后被消费）

Kafka：不同组名可以重复消费（通过group.id设置）

Producer：配置信息—>构建topic—>发送信息（信息是以<key,value>方式存在的）

Consumer：设置kafka地址—>设置消费组id(group\_id)—>确定topic—>从topic中拉取数据。

## 补充jpa中的继承映射

父类需要有注解 @Inheritance

可以选择继承策略：例如：

**@Inheritance**(strategy = **InheritanceType**.***JOINED***)

* JOINED：继承的子类各自生成一张表
* SINGLE\_TABLE：只生成一张表，用一个字段当鉴别器
* TABLE\_PER\_CLASS：每个类都生产一张表，包括基类

一般用最后一个，都生成一张表，并且子类生成的表中带有父类的属性。但添加时只往子类中添加数据，父类中不添加数据。

## Springcloud补充

通过Feign互相调用接口

定义一个servcie

**@FeignClient**(value = **"eureka-hello2"**)//是被调用方的服务名称  
**@Service  
public interface ClientService** {  
 **@RequestMapping**(**"/aaa"**)  
 **String** aaa(**@RequestParam**(name = **"name"**) **String** *name*);  
}

在controller中调用：

**@Autowired  
 private ClientService** clientService;  
 **@RequestMapping**(**"/aaa"**)  
 **public String** aaa(**String** *name*){  
// return clientService.aaa(name);  
 **String** aa = clientService.aaa(*name*);  
 **System**.***out***.println(**"---"**+aa);  
 **return** aa;  
 }

用template调用接口：

**//在启动类中注入restTemplate**

**@Bean  
@LoadBalanced** //表示开启客户端负载均衡  
**RestTemplate** restTemplate() {  
 **return new** RestTemplate();  
}

在controller中调用：

**@RequestMapping**(**"/testaa"**)  
**public String** testaa(**@RequestParam**(name = **"name"**) **String** *name*){  
 **String** body = restTemplate.getForEntity(**"http://eureka-hello2/aaa?name="**+*name*,**String**.**class**).getBody();  
 **return** body;  
}

两者的区别：

Feign是一个声明式的REST客户端，它的目的就是让REST调用更加简单。

Feign提供了HTTP请求的模板，通过编写简单的接口和插入注解，就可以定义好HTTP请求的参数、格式、地址等信息。

Template是将一个template作为bean注入到启动类。在controller中调用，并且需要加上请求地址（地址中不包括ip和端口，要有服务名）。

解决springcloud中一个服务挂掉，eureka中仍显示up的情况。

关闭注册中心的自我保护机制，防止已关闭的实例无法从注册中心剔除。

添加如下配置，关闭自我保护

Eureka server application.yml

|  |
| --- |
| eureka:  server:  enableSelfPreservation: false  eviction-interval-timer-in-ms: 1000 (单位是毫秒，默认是60\*1000) |

Service application.yml

|  |
| --- |
| eureka:  instance:  leaseRenewalIntervalInSeconds: 1  leaseExpirationDurationInSeconds: 2 |

Hystrix熔断器（容错机制、降级处理）

先添加依赖

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>  
</dependency>

FeignClient的熔断处理：

#application.yml中加入

feign:  
 hystrix:  
 enabled: true

接口类上加入的注解中添加属性fallback,指定回调类

@FeignClient(value = "eureka-hello2",fallback = FeignClientFallback.class)

创建回调类：

@Component  
public class FeignClientFallback implements ClientService {  
 @Override  
 public String aaa(String name) {  
 System.out.println("熔断处理!");  
 return "熔断处理!";  
 }  
}

RestTemplate+ribbon的熔断：

加入依赖，同上。

在启动类中加入注解：@EnableHystrix

在controller的请求方法上加上@HystrixCommand注解，并指定容错方法：

**@HystrixCommand**(fallbackMethod = **"hiError"**)

在类中添加容错方法（与调用的方法一致（参数格式和类型一致））：

public String hiError(@RequestParam(name = "name") String name){  
 System.out.println("请求方法失败，进入熔断方法！");  
 return "请求方法失败，进入熔断方法！";  
}

zuul路由网关

加入依赖

<dependency>

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

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

</dependency>

启动类中引入注解

@EnableEurekaClient

@EnableZuulProxy

修改zpplication.yml文件

server:

port: 8769

spring:

application:

name: service-zuul

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

zuul:

routes:

api-a:

path: /api-a/\*\*

service-id: service-ribbon

api-b:

path: /api-b/\*\*

service-id: service-feign

8769 端口的所有 api-a 请求会转发到 service-ribbon 工程，api-b 到 service-feign

路由网关起到一个反向代理的作用，请求到配置的path，经过zuul转发到服务。

分布式配置

从远端（github）或获取文件，开发或生产文件，获取文件里对应的属性值。