[通用]

遇到的问题，最后怎么解决，还有没有更好的办法？

[项目][设计]

系统开发前要确定用户量，数据量以及并发量

[Mybatis][Map]

以下是mybatis使用Map中其中一个key("content")进行foreach的例子。

<insert id="insertBatchByMap">

insert into cmdp\_ska\_cust\_group\_import (id, cust\_file\_id, cust\_name,

cust\_short\_name, cust\_code, board,

sale\_hq\_area, sale\_area, cust\_file\_status,

cust\_file\_main\_id, big\_area, area,

status, group\_id, bill\_code,

zone\_code, first\_visit\_time, month\_effect\_time,

cust\_level, res\_emp\_code, res\_emp\_name,

operation, check\_status, check\_result,

update\_time, update\_emp\_code)

values

<foreach collection ="paramMap['content']" item="custGroupImport" separator =",">

(#{custGroupImport.id,jdbcType=BIGINT}, #{custGroupImport.custFileId,jdbcType=VARCHAR}, #{custGroupImport.custName,jdbcType=VARCHAR},

#{custGroupImport.custShortName,jdbcType=VARCHAR}, #{custGroupImport.custCode,jdbcType=VARCHAR}, #{custGroupImport.board,jdbcType=VARCHAR},

#{custGroupImport.saleHqArea,jdbcType=VARCHAR}, #{custGroupImport.saleArea,jdbcType=VARCHAR}, #{custGroupImport.custFileStatus,jdbcType=VARCHAR},

#{custGroupImport.custFileMainId,jdbcType=VARCHAR}, #{custGroupImport.bigArea,jdbcType=VARCHAR}, #{custGroupImport.area,jdbcType=VARCHAR},

#{custGroupImport.status,jdbcType=VARCHAR}, #{custGroupImport.groupId,jdbcType=VARCHAR}, #{custGroupImport.billCode,jdbcType=VARCHAR},

#{custGroupImport.zoneCode,jdbcType=VARCHAR}, #{custGroupImport.firstVisitTime,jdbcType=VARCHAR}, #{custGroupImport.monthEffectTime,jdbcType=VARCHAR},

#{custGroupImport.custLevel,jdbcType=VARCHAR}, #{paramMap[empCode],jdbcType=VARCHAR}, #{custGroupImport.resEmpName,jdbcType=VARCHAR},

#{custGroupImport.operation,jdbcType=VARCHAR}, #{custGroupImport.checkStatus,jdbcType=VARCHAR}, #{custGroupImport.checkResult,jdbcType=VARCHAR},

now(), #{custGroupImport.updateEmpCode,jdbcType=VARCHAR})

</foreach>

</insert>

1. paramMap在foreach语句中获取value的方式跟在SQL语句中的方式不一样。

如果key是具体值，如"content",

foreach : paramMap['content']

SQL : paramMap[empCode]

如果key是变量(item)，如key

foreach : paramMap[key]

SQL : paramMap[${key}]

2. <foreach collection ="paramMap.keys" item="key"> 用于获取map的keys

3. <foreach collection ="paramMap.values" item="value"> 用于获取map的values

[Spring][加载配置文件]

<context:property-placeholder location="classpath:sys.properties" ignore-unresolvable="true"/>

sys.properties是配置文件名

[MySQL][存储过程][基本模板]

CREATE PROCEDURE ROLLING\_TABLE(

-- in p\_schema varchar(20),

-- in p\_table varchar(20)

)

BEGIN

-- 执行内容

END

[MySQL][存储过程][表名使用变量]

SET @statement = CONCAT("create table ", @table\_name\_tmp, ' SELECT \* from ', p\_table, ' where 1 = 2');

PREPARE stmt FROM @statement;

EXECUTE stmt;

[MySQL][存储过程][判断表是否存在]

SELECT COUNT(\*) INTO @cnt\_cur FROM information\_schema.statistics WHERE TABLE\_NAME=p\_table AND TABLE\_SCHEMA=p\_schema;

if @cnt\_cur > 0 then

-- 执行内容

end if;

[Java][Spring][annotation指定bean名字]

如果有多个实现类实现了同一个接口，在使用过程中，为了区分使用不同的实现类，操作如下：

先定义bean名字 ：@Service("accessLogKafkaManager")

注入时指定：　　　@Resource(name = "accessLogKafkaManager")

[Java][Json to Object][泛型]

GenericReq<MobileBody> req = JSON.parseObject(content,new TypeReference<GenericReq<MobileBody>>(){});

[maven][unpack]

assembly <unpack>true</unpack> //直接将所有依赖都已解包的形式放到jar里面

此时如果想exclude，表达如下：

<!-- <unpack>true</unpack> -->

<unpackOptions>

<excludes>

<exclude>\*\*/logback-test.xml</exclude>

<exclude>\*\*/\*.properties</exclude>

</excludes>

</unpackOptions>

[maven][useProjectArtifact]

assembly <useProjectArtifact>true</useProjectArtifact> //true将项目代码也打包进工程

[Spring][ControllerAdvice][Exception][Interceptor]

当throw Exception时，不会进入常规的Interceptor， 但可以进入添加了指定Exception的ControllerAdvice。

[Jmeter][限制QPS]

添加--->定时器--->Constant Throughput Timer

设置定时器的Target throughput为1200/分钟（20 QPS），设置Calculate Throughput based on 的值为All active threads。

Constant Throughput Timer只有在线程组中的线程产生足够多的request 的情况下才有意义，因此，即使设置了Constant Throughput Timer的值，也可能由于线程组中的线程数量不够，或是定时器设置不合理等原因导致总体的QPS不能达到预期目标。

Calculate Throughput based on：有5个选项, 分别是：

　　This thread only：控制每个线程的吞吐量，选择这种模式时，总的吞吐量/s = target Throughput \* 线程数量/60。

All active threads：设置的target Throughput 将分配在每个活跃线程上，每个活跃线程在上一次运行结束后等待合理的时间后再次运行。活跃线程指同一时刻同时运行的线程。总的吞吐量/s = target Throughput /60。

All active threads in current thread group：设置的target Throughput 将分配在当前线程组的每一个活跃线程上，当测试计划中只有一个线程组时，该选项和All active threads 选项的效果完全相同。

All avtive threads(shared)：与All active threads的选项基本相同。唯一区别是，每个活跃线程都会在所有活跃线程上一次运行结束后等待合理的时间后再次运行。

All active threads in current thread group(shared)：与All active threads in current thread group 基本相同，唯一的区别是，每个活跃线程都会在所有活跃线程的上一次运行结束后等待合理的时间后再次运行。

备注：

shared，就是全部跑完，一起等，再全部开始跑

in current thread group，只分配在当前线程组的每一个活跃线程上，测试时，有所有线程和active线程

[Jetty][IDEA][PORT]

修改jetty端口，VM参数中添加： -Djetty.http.port=9090，此参数在9.4版本后生效。（测试中发现9.2添加-Djetty.port=9090参数，还是无法修改端口，可能是bug）

[Maven][打包][疑问？]

在assembly文件中配置了unpack=true，在添加<unpackOptions><excludes>前后出现奇怪现象，如果曾经<excludes>， 就算去除这段配置，还是会exclude。但如果一开始没有加这个配置，文件不会被exclude。

[Maven][例子][依赖解包][exclude unpack]

<assembly>

<id>prod</id>

<formats>

<format>jar</format>

</formats>

<includeBaseDirectory>false</includeBaseDirectory>

<fileSets>

<fileSet>

<directory>src/main/assembly/prod/resources</directory>

<outputDirectory>/</outputDirectory>

<!-- 样例代码，此段exclude不是真实的

<excludes>

<exclude>\*\*/sys.properties</exclude>

</excludes>

-->

</fileSet>

</fileSets>

<dependencySets>

<dependencySet>

<outputDirectory>/</outputDirectory>

<useProjectArtifact>true</useProjectArtifact>

<unpack>true</unpack>

<!-- 将scope为runtime的依赖包打包到lib目录下。 -->

<scope>runtime</scope>

<unpackOptions>

<excludes>

<exclude>\*\*/logback-test.xml</exclude>

<!-- 不要随便用<exclude>\*\*/\*.properties</exclude>, 这样会把其他 unpack jar包里面的properties文件也过滤掉，导致某些类初始化失败 -->

<exclude>src/main/resources/\*.properties</exclude>

</excludes>

</unpackOptions>

</dependencySet>

</dependencySets>

</assembly>

[Zookeeper][连接]

使用ZooInspector连接zookeeper

[JStorm][Kafka][forceFromStart]

forceFromStart true，每次启动都从最开始消费；false，每次启动从上一次结束点开始消费

但如果从来都没有消费过，没有offset记录，则即使是false，不会消费任何消息。需要设置为true才能消费。

[Connection][一个]

1.一般的Connection不是线程安全的。如数据库connection，它在多线程环境中使用时,会导致数据操作的错乱,特别是有事务的情况.connection.commit()方法就是提交事务,你可以想象,在多线程环境中,线程A开启了事务,然后线程B却意外的commit,这该是个多么纠结的情况. -- 参考：https://blog.csdn.net/xwq911/article/details/49150043

2. 如果只有一个连接，该连接正在被使用（conn.commit()）,这时候如果其它线程希望使用该连接，需要等待，所以可以通过连接池在一开始多创建几个连接，还能减少使用过程中再来创建和销毁的资源消耗。

[批量更新数据]

批量更新数据时，可能会更新了脏数据（老数据），在Kafka->JStorm->ES的实践中，通过以下方法解决：

1. 相同id分发到相同的Bolt（防止不同bolt处理时，后消息的先被判定要写入，但此时尚未写入，接着前消息的在另一个Bolt被判定要写入，这样前消息会覆盖后消息）

2. 同一批次先做内部比较，取最新的（防止同一批次中后消息先传过来）

3. 写入es前把里面的时间拿出来跟待写入数据比较，只有待写入数据时间较新，才允许写入

[ThreadLocal]

ThreadLocal 将一些线程不安全的变量，将用同步的方式转为为每个线程创建一个对象来实现线程安全。

Thread和ThreadLocal之间的关系，能得出如下基础结论：

线程内部持有一个Map存放着自己的局部变量，这些变量的key是ThreadLocal对象（可以理解为什么叫做线程局部变量了吧）；

有一个变量想放入线程的局部变量的你就需要创建一个ThreadLocal对象；

ThreadLocal本身没有什么存储结构，只是提供了方法，所以变量不是存在ThreadLocal中的，ThreadLocal负责搬运。

<Thread, <ThreadLocal, Value>>

参考：https://blog.csdn.net/Justin\_zhao/article/details/78470713

[连接池][最大连接数]

当已取出的连接数大于设置的最大连接数时，取出会等待。

[连接池][参数]

maxTotal表示pool中对象最多能有多少。

maxIdle属性是指pool中最多能保留多少个空闲对象。

minIdle属性是指pool中最少有多少个空闲对象。

TestOnBorrow 取出连接时先校验连接是否能用，不能用销毁，每次借出链接都要检查，影响性能。

TimeBetweenEvictionRunsMillis 逐出扫描的时间间隔，定时把无用连接逐出默认-1，不扫描

testWhileIdle空闲时校验

numTestsPerEvictionRun 每次扫描校验的个数

仅设置testWhileIdle为true时，timeBetweenEvictionRunsMillis和numTestsPerEvictionRun必须设置。否则无效

testOnReturn为true时，这种验证发生在链接归还时，当间隔很久后再次被借出时，是一个已经失效的链接。

[Redis][连接池][error]

"Unexpected end of stream." 由于获取的Jedis不可用导致的。线程池中可以通过调整参数来解决。

TestOnBorrow

or

testWhileIdle为true时，timeBetweenEvictionRunsMillis和numTestsPerEvictionRun必须设置

[ES][聚合]

filter 再次过滤后聚合结果个数

cardinality 返回字段distinct后的个数

terms 返回字段每个值对应的结果个数,还能用于数组类型

"aggregations" : {

"<name>" : {

"terms": {

"field": "<field\_name>"

}

}

}

[Java][Throwable]

catch Throwable 能捕获比Exception更多的信息，包括Error。但一般不建议catch Throwable, 因为捕获Exception是为了让程序能自修复，但Error一般都是致命的错误。除非系统不能打印控制台log，在排查问题时就把错误打印到logger.error。

[Java][NoClassDefFoundError]

NoClassDefFoundError一般出现于静态内部类当中。通常由于外部类构造函数初始化失败导致（在第一次使用外部类时，就算是静态方法，也会调用构造函数）。

初始化失败原因可以是：

1. 依赖的配置文件不存在，可能被maven打包排除了

2. 依赖的类冲突，同名但版本不同的包，如果发现类冲突，如NoSuchMethodError，可以查找最后一层或者倒数第二层使用的类，看看是否有重复。

3. 其他各种导致构造函数不能运行的错误代码。

[Maven][安装自己的包]

mvn install:install-file -Dfile=erui-trace-1.3.0.1.jar -DgroupId=com.sf -DartifactId=erui-trace -Dversion=1.3.0.1 -Dpackaging=jar

[Java][Lambda][处理List]

1. 利用stream().forEach()循环处理List;

List<String> list = Lists.newArrayList();//新建一个List 用的google提供的Guava package com.google.common.collect;

list.add("1");

list.add("2");

list.add("3");

list.stream().forEach(string ->{

System.out.println(string);

});

运行结果：

1

2

3

2. 利用stream().map()处理List，并给另外一个List赋值：

List<String> list1 = Lists.newArrayList();

List<String> list2 = Lists.newArrayList();

list1.add("1");

list1.add("2");

list1.add("3");

list2 = list1.stream().map(string -> {

return "stream().map()处理之后：" + string;

}).collect(Collectors.toList());

list2.stream().forEach(string -> {

System.out.println(string);

});

运行结果：

stream().map()处理之后：1

stream().map()处理之后：2

stream().map()处理之后：3

3. 利用stream().filter()处理List,对该List进行过滤处理：

List<String> list1 = Lists.newArrayList();

List<String> list2 = Lists.newArrayList();

list1.add("1");

list1.add("1");

list1.add("2");

list1.add("3");

list2 = list1.stream().filter(s -> !s.equals("1")).collect(Collectors.toList());

System.out.println(list2.toString());

运行结果：

[2, 3]

Note： 除了 xxx->{xxxxx} 外，还有User::getName等表达方式，用于获取对象中的指定变量。

[MySQL][远程授权]

授权所有权限\*.\*, 给所有机器'%', 账号'root', 密码'Coolge98#'

GRANT ALL PRIVILEGES ON \*.\* TO 'root'@'%' IDENTIFIED BY 'Coolge98#' WITH GRANT OPTION;

[List][Map][Lambda]

1. 数据源是List<User>, 转换成Map<User.getName(), User>

方案：

使用Lambda表达式，list.stream().collect(Collectors.toMap(User::getName, user -> user));

优点：

不需要自己写循环，写法优雅。比起自己写一个方法更加灵活，Map的key和value都可以是不同类型

缺点：

执行效率没有迭代器和for循环快。

拓展：

如果只是想要提取所有User.name并组成集合的话，可以这么写,返回的是list：

list.stream().map(User::getName).collect(Collectors.toSet());

[ElasticSearch][字段多选]

ElasticSearch关于字段多选的可以如何加快筛选速度？

a) 传数组

b) 分词

然后都可以用terms来过滤

[Maven][xlsx文件损坏]

maven在打包时会编译项目中的文件，某些特殊格式的文件maven无法识别，会导致打包后文件损坏。解决办法是不编译这些文件，如下：

第一种方式，maven打包时，过滤不编译文件类型

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-resources-plugin</artifactId>

<configuration>

<nonFilteredFileExtensions>

<nonFilteredFileExtension>xlsx</nonFilteredFileExtension>

<nonFilteredFileExtension>xls</nonFilteredFileExtension>

<nonFilteredFileExtension>zip</nonFilteredFileExtension>

<nonFilteredFileExtension>cer</nonFilteredFileExtension>

<nonFilteredFileExtension>pfx</nonFilteredFileExtension>

<nonFilteredFileExtension>py</nonFilteredFileExtension>

<nonFilteredFileExtension>keystore</nonFilteredFileExtension>

</nonFilteredFileExtensions>

</configuration>

</plugin>

第二种方式，在过滤的时候排除掉相应的文件

<resource>

<directory>${project.basedir}/src/main/resources</directory>

<filtering>true</filtering>

<excludes>

<exclude>static/fonts/\*\*</exclude>

<exclude>static/file/\*\*</exclude>

</excludes>

</resource>

<resource>

<directory>${project.basedir}/src/main/resources</directory>

<filtering>false</filtering>

<includes>

<include>static/fonts/\*\*</include>

<include>static/file/\*\*</include>

</includes>

</resource>

[Git][修改 本地/IDEA 密码]

在git bash先执行以下命令：

git config --global credential.helper store

然后随便pull/push, 会提示重新输入密码，密码更新后，IDEA的git密码也会更新。

[Spring][placeholder]

使用说明：

1. 在spring的xml中加入以下配置：

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations">

<list>

<value>classpath:xxx.properties</value>

</list>

</property>

</bean>

2. 在需要使用${}的类上加入以下annotation：

@PropertySource("classpath:xxx.properties")

[Log][响应请求][错误原因]

关于日志，如果是服务化的error日志，里面必须有明显的内容来区分不同的请求，可以是单号、请求id甚至整个请求等，这样才方便定位问题。

当请求失败后，Resopnse不需要包括区分请求的内容，毕竟这个就是返回给调用方的。但必须包含succ字段，以及失败原因，可以是编码，也可以是文字，这个不要紧，只要日志记录清晰就能追查原因，调用方其实只要知道是成功还是失败，失败的基本原因就够了。

[Redis][setnx][锁][RedisTemplate]

如果锁的实现用：先get(),不存在就set()，这方式是不严谨的。因为在get()和set()中间有间隙，在间隙中，线程1还没set，线程2就尝试get，线程2就以为没有锁。所以get和set要用原子性操作来执行，比如：setnx(key, value)命令，当返回true，则表明key不存在（还没锁）并set(查询和set是整体的原子性操作)，当返回false则key已存在（锁住）。

在RedisTemplate中，setnx对应的方法是setIfAbsent, 但这个方法不是设置过期时间，需要执行后马上执行expire命令来定义过期时间。如果在执行expire命令出了异常，就可能导致这个锁永远存在，这是不合理的。为了优化，可以通过自定义lua脚本来把setnx和expire放到一个整体来运行，具体如下：

//lua脚本内容

String newSetIfAbsentScriptStr = " if 1 == redis.call('setnx',KEYS[1],ARGV[1]) then" +

" redis.call('expire',KEYS[1], KEYS[2])" +

" return 1;" +

" else" +

" return 0;" +

" end;";

public RedisScript<Boolean> newSetIfAbsentScript = new DefaultRedisScript<>(newSetIfAbsentScriptStr,Boolean.class);

//setIfAbsent的可调用方法

public boolean setIfAbsent(String key,String value,Long expireTime, TimeUnit timeUnit) {

try {

Long seconds = TimeoutUtils.toSeconds(expireTime, timeUnit);

return redisTemplate.execute(newSetIfAbsentScript, Lists.newArrayList(key, seconds.toString()), value);

} catch (Exception e) {

logger.error("Fail to setIfAbsent value to Redis.", e);

}

return false;

}

最好把RedisScript保存一下，这样可以每次脚本执行的时候，使用同一个sha1就可以访问到该script，而不必每次都重复上传script。

[职业发展][2018年终会议总结建议]

1. 公司架构，有哪些部分，大概多少人，分别做什么，多认识别的部门的人

2. 公司的业务，具体业务是做什么的

3. 技术发展，更深入的了解到底是怎么实现的

必备：后端，实时计算

加分：算法与模型

[Log][日志分隔]

使用换行符"\n"来分隔日志的不同内容, 这样就更加清晰。

[Maven][Testcase]

maven不能跑testcase的可能原因：

1. POM文件中有参数指定跳过testcase，如maven-surefire-plugin，或者自定义的properties

2. 防止testcase的路径不对，应该是src/test/java下面

[mock][PowerMockito]

Mockito可以模拟对象，PowerMockito可以模拟静态方法，但他们之间有些共用的代码，所以版本号要适当，否则会冲突。比如以下版本：

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-all</artifactId>

<version>2.0.2-beta</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.powermock</groupId>

<artifactId>powermock-api-mockito</artifactId>

<version>1.7.4</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.powermock</groupId>

<artifactId>powermock-module-junit4</artifactId>

<version>1.7.4</version>

<scope>test</scope>

</dependency>

使用PowerMockito时要注意：

1. 在测试类上标注@RunWith(PowerMockRunner.class)

2. 使用@PrepareForTest({xxx.class})来指定要mock的静态方法类

3. 如果需要和spring-test搭配的话，可以标注如下：（旧版的jar可能不支持）

@RunWith(PowerMockRunner.class)

@PowerMockRunnerDelegate(SpringJUnit4ClassRunner.class)

4. 使用PowerMockito会提示classloader错误，加入注解：@PowerMockIgnore(YourPackagePath.\*)来解决由于MockClassLoader造成的class加载错误，通过这个注解可以让报错的类使用系统的ClassLoader来加载报错类。如下：

java.security.NoSuchAlgorithmException: class configured for SSLContext: sun.security.ssl.SSLContextImpl$TLSContext not a SSLContext

解决：@PowerMockIgnore({"sun.security.\*", "javax.net.\*"})

ERROR StatusLogger Caught exception executing shutdown hook Shutdown callback for LoggerContext[name=org.powermock.core.classloader.MockClassLoader@12843fce] java.lang.LinkageError: javax/management/MBeanServer

解决：@PowerMockIgnore({"javax.management.\*"})

[反序列化][ClassNotFoundException]

序列化的类在反序列化后，需要转化为当初定义时指定的类，包括类名和包名需要一致。如果指定类被移走了，就会报ClassNotFoundException。

[单元测试]

1.单元测试主要测试的是类的方法，是最基本的测试，也是最小单元的测试。只判断方法的代码逻辑，不涉及业务逻辑。

2.测试驱动开发：

1. 先思考并定义需要单元测试的情况

2. 编写单元测试，并编写需要测试的类的接口

3. 进行测试，并只针对结果错误的部分进行修改

4. 直到全部测试通过，则此部分开发完成

3.关于代码中哪些部分需要进行单元测试，就是在代码中通过注释表明代码逻辑的部分，因为自己觉得这些部分可能不好理解，所以这些部分都需要在单元测试中被测到。

4.尽量将单元测试与被测试类放在同一个包，这样还可以方便地测试protected方法。

[单例与静态方法类]

单例可以继承和多态，而且单例在mock时比静态方法类更加灵活。

[大型网站架构模式][模式]

1. 分层，就是对同一个业务模块进行分层，如应用层，服务层和数据层，只负责单一职责，不允许跨层调用，也不允许逆向调用。

优势：便于合作开发和维护，各层的改动不需要其他层的调整。

2. 分割，按照业务逻辑，把系统分成独立的模块，每个模块内部都是完整的分层结构。

优势：1. 方便开发，维护，因为不同代码分开模块，代码更专一，清晰。

2. 发版灵活，不会因为新功能的发版，改动代码，影响到了其他功能（牵一发而动全身）。

3. 提高并发和功能扩展，针对并发高的模块，可以多部署几台机器。

3. 分布式，使用更多的机器完成同样的功能，提高并发量、计算能力和容量等。

4. 集群，多台服务器部署相同应用，提高系统可用性。

5. 缓存，加快处理速度，数据提取速度。

CDN，内容分发网络，在靠近用户的网络服务商，放置常用静态资源

反向代理，部署在后端服务器的前面，放置常用静态资源

本地缓存

分布式缓存，redis

6. 异步，通过消息队列存放待处理任务内容，消息队列消费者来处理任务。

提高系统可用性。就算消息者服务器挂了，任务就存在队列中，不影响新任务的进入，等服务器恢复，接着处理

加快响应速度。不需要等处理完再响应

消除并发高峰。当大量请求涌入时，都存放在队列中按序执行，不会瞬间对服务器造成太大压力。

7. 冗余，数据热备份，冷备份。

8. 自动化，代码管理？测试，安全检测，部署，监控，报警，失效转移，失效恢复，降级，分配资源

9. 安全，密码，手机校验码，验证码；加密；XSS、SQL注入；垃圾、敏感信息过滤；交易信息的风险控制。

[Lambda][Map<String, List<>>]

Map<String, Set<String>> result = (Map<String, Set<String>>) list.stream().collect(Collectors.groupingBy(Person::getId, Collectors.mapping(Person::getName, Collectors.toSet())));

[AOP][引入增强]

增强，可以让被代理类运行定义中没有实现过的接口功能。

1. 指定要实现的接口

public interface Love {

void display(String name);

}

2. 给这个接口实现advice

@Component

public class LoveAdvice extends DelegatingIntroductionInterceptor implements Love {

public void display(String name) {

System.out.println("You are my heart:" + name);

}

}

3. 关联被代理类和advice

<bean id="greetingProxy" class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="interfaces" value="com.aop.Love" />

<property name="target" ref="greetingImpl" />

<property name="interceptorNames" value="loveAdvice" />

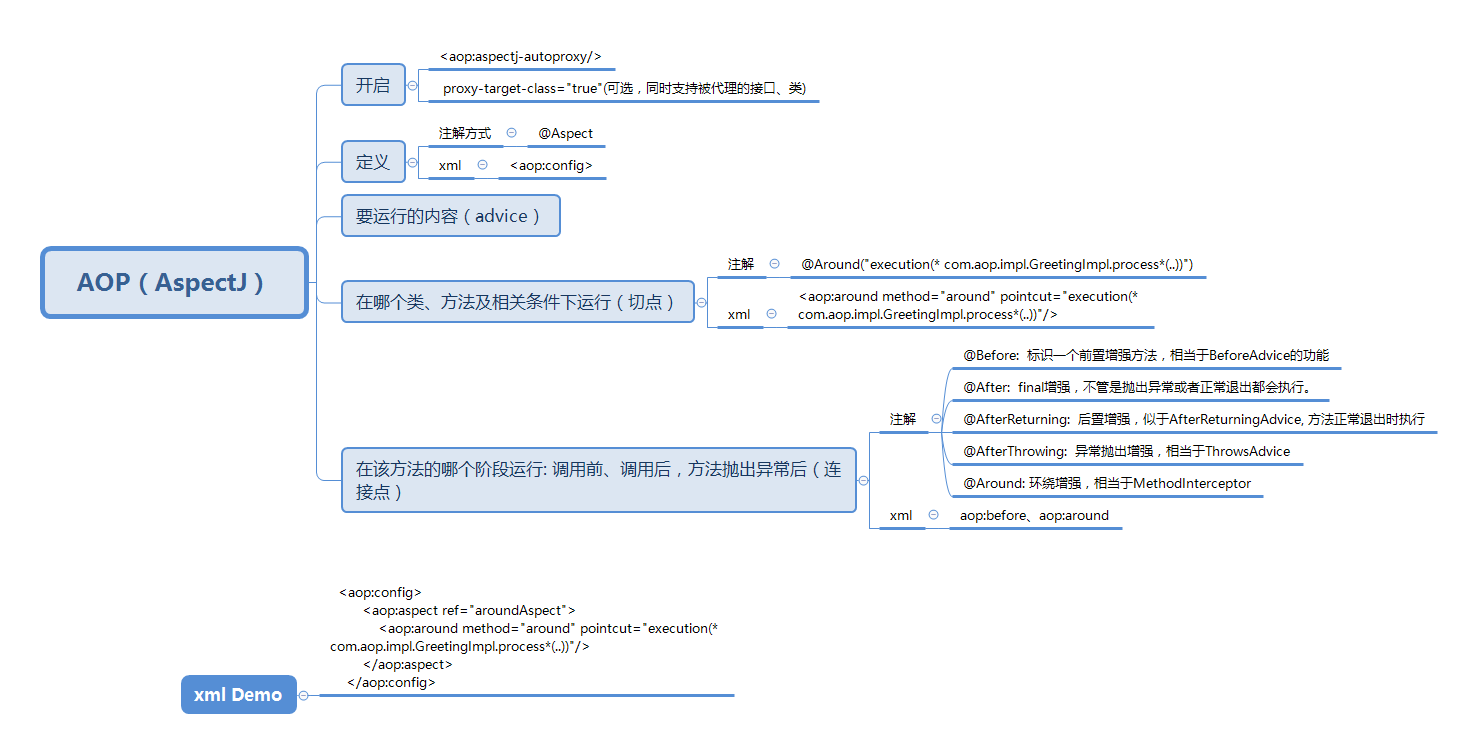
<property name="proxyTargetClass" value="true" />

</bean>

[ThreadLocal][初始化]

ThreadLocal不能只用set来初始化，因为set方法只是当前线程有效，针对当前线程进行赋值，所以这个方法不会被其他线程调用。我们可以重写initValue方法，initValue对所有线程都生效。这是因为在执行get方法时，如果获取不了值，会自动调用initValue方法，初始化一个值给新的线程。

[AOP][aspect][思维导图][简单汇总]



详细使用方式可参考：

<https://www.jianshu.com/p/830e799e099b>

[AOP][Spring][Aspect][内部调用]

Spring Aspec正常情况下不能代理内部方法，解决办法：

1. 添加属性<aop:aspectj-autoproxy **expose-proxy="true"**/>
2. 内部调用时用((XXX) AopContext.currentProxy())，不能用this或者隐式写法
3. 这样可能需要改动原本的代码，可以写一个Wrapper，继承本来的方法类，重写需要拦截的内部方法所在的外部方法，注入时就用Wrapper
4. ？？？ 在定义pointcut时，指定的类不能是wrapper，而是被代理的类名，不然会报错，在AopContext.currentProxy()时获取一个不是需要的proxy

[DrawApi][重构]

1. 创建BarRequestExceptionAdvice，BarRequestException
   1. 当抛出BarRequestException时，BarRequestExceptionAdvice会捕获并返回格式化的错误响应。
   2. 不需要自己捕获处理（RuntimeException），交由Advice处理
   3. 方便编写代码，对于常规无需处理的错误，直接抛出BarRequestException即可，不需要写try-catch，也不需要在service来构造Response，代码及逻辑清晰。
   4. BarRequestExceptionAdvice内部还支持输出error日志，日志也是经过格式化，通过BarRequestException构造的参数传递描述信息，还报告当前请求的body，方便排错。
   5. 简化代码，对于Advice能做到的所有错误处理，不需要重复编写，抛异常即可。
2. 创建MAPIHttpServletRequestWrapper，HttpServletRequestReplacedFilter
   1. 在请求的任何位置都可以通过MAPIHttpServletRequestWrapper.getParam(request) 来获取请求body。
   2. 原生SpringMVC框架中，因为已经消费过HttpRequest的请求内容，在Controller层开始不能通过HttpRequest获取。
3. 创建ESTimeLogAspect，SearchPlatformWrapper
   1. 对于SearchPlatformHelperInstance.invoke，WechatFlagManagerImpl.findTagsByPhoneService\*及WechatFlagManagerImpl.findBoundByPhoneServiceImpl(..)的运行时间进行记录
   2. 同一个请求中的所有时间记录进行统一的打印，监控清晰
   3. Wrapper继承被代理类，业务逻辑代码不需要改动，只要改写注入的bean即可。
4. 创建TestHelper，SpringMVCBaseTest
   1. 方便编写Controller层的功能测试，本地模拟Http请求。

[Spring in Action][第一章][Spring之旅]

|  |  |
| --- | --- |
| 1.什么是POJO  2-5.Spring降低复杂性的4种策略  6.什么是ApplicationContext  7.常见的ApplicationContext  8.bean的生命周期  9.Spring的组件 | 1. POJO就是JaveBean。 2. 基于POJO的轻量级和最小入侵性编程，不需要POJO继承任何接口或类，没有Spring痕迹，该类不管是否在Spring环境都可以使用。 3. 通过依赖注入和面向接口实现松耦合，基于接口可以方便替换同一接口的各种实例，通过xml或者注解装配实例，可以把new语句也除掉，实现依赖的松耦合。 4. 基于切面和惯例进行声明式编程。 5. 统一代码，方便更改。AOP将散乱相同的横向逻辑（日志，安全，事务）统一起来，如果要更改逻辑，改一个地方就好了。 6. 精简代码。核心逻辑中只有业务逻辑，横向逻辑被抽离出来。 7. 扩展功能且无需改动原来的代码。不管是切面还是被代理类，都无需在代码中进行任何关联，他们不知道对方的存在，只需要在spring上下文配置他们的关系即可。 8. 通过切面和模版减少样板式代码。使用模版代码，例如JdbcTemplate，可以简化工具使用的代码。 9. ApplicationContext是Spring容器的实现，相比BeanFactory，它还提供应用框架级别的服务，例如从属性文件解析文本信息以及把应用事件发给感兴趣的监听者。 10. AnnotationConfigApplicatioContextn从Java配置类中加载ApplicationContext，FileSystemXmlApplicationContext从文件系统中的XML中加载，ClassPathXmlApplicationContext从类路径下的XML中加载。 11. Bean装载到Spring ApplicationContext的生命周期过程： 12. 实例化 13. 属性赋值   以下是如果实现了前面的接口，会调用后面的方法   1. BeanNameAware， 将bean的ID传递给setBeanName()方法 2. BeanFactoryAware, setBeanFactory()方法 3. ApplicationContextAware，setApplicaitonContext()方法 4. BeanPostProcessor, postProcessBeforeInitialzation()方法 5. InitializingBean, afterPropertiesSet(), 如果bean使用init-method声明了初始化方法，该方法也会被调用 6. BeanPostProcessor, postPorcessAfterInitialization()方法 7. bean可以使用了 8. DisposableBean, detory(), 如果bean使用destory-method声明了销毁方法，该方法也会被调用。 9. Spring4.0的组件： 10. Spring Web Flow, 建立于SpringMVC上，支持基于流程的会话式Web应用(购物车，向导功能)。 11. Spring Web Service，提供契约优先的Web Service模型 12. Spring Security，安全 13. Spring Integration， 与其他应用进行交互 14. Spring Batch，提供定义批处理为Bean，以及任务间的依赖和串行、并行等运行方式。 15. Spring Data，访问SQL及NoSQL数据库 16. Spring Social，社交网络扩展模块 17. Spring Mobile 18. Spring for Android 19. Spring Boot，依赖于自动配置技术，消除大部分的Spring配置，还能减少工程构建文件的大小。 |
| 1. Spring的特点就是维护实例的创建，方便依赖的替换，增强功能或前后逻辑且无需改变原来的代码，通过模版工具类简化第三方工具的使用以及一系列不同领域的便捷开发。 2. 为代码开发添加更多的灵活性和便捷性。 | |

[Spring in Action][第二章][]

|  |  |
| --- | --- |
| 1.自动注入Bean的方式  2.自动扫描注入bean  3.@Autowired的作用域  4.Jave代码注入bean  5.XML注入bean | 1. 通过@Configuration注解类指定这是个配置类，@ComponentScan注解类表示要自动扫描，当没有任何注解的时候，默认扫描跟这个配置类在同一个包下的包含@Component类。在使用到的类中用@Autowired注入 2. @ComponentScan的value有几种常用方式，basePackages={“com.pojo”,”com.domain”}指定要扫描的包，basePackageClasses={CDPlayer.class, DVDPlayer.class}指定要扫描的类。如果指定的是包，对于代码重构需要迁移包时，那么所指定的基础包就可能会出现错误。可以考虑创建一个用来进行扫描的空标记接口，通过在类中实现接口，再通过basePackageClasses指定接口名，批量扫描，还不影响重构。 3. @Autowired除了可以注解属性，还可以注解方法，这时表示用容器里的bean作为方法的入参。 4. 通过Java代码装配bean，可以使用Java语法，支持判断逻辑等功能，更加强大。 5. 在@Configuration配置文件中定义bean 6. 通过@Bean注解来定义，返回bean的类型。当存在入参时，如果入参名字与bean一致，也会自动注入，类似@Autowired注解方法一样 7. XML注解 8. 构造器初始化    * 1. <constructor-arg>元素：<constructor-arg ref=”beanId” />      2. c-命名空间：         1. 引用传递：c:fieldName-ref=”beanId”，c:\_<fieldIndex>-ref=“beanId”(filedIndex指参数顺序，从0开始，如c\_0-ref=”beanId”), c:\_-ref=”beanId”(当只有一个参数时可以这么写，但之前试过不行)         2. 值传递：c:\_fieldName=”value”, c:\_fieldIndex=”value”， c:\_ =”value” 9. 参数初始化 10. <property>元素 11. p-命名空间：p:fieldName-ref=”beanId”, p:fieldName=”value” 12. <util>, <list><set>不能用于命名空间。可以使用<util:list>定义list。 13. <util:list id=”listBeanId”>   <value>xxx</value>  <value>yyy</value>  </util:list>  然后配置中就可以把”listBeanId”放到ref来使用   1. 除了util:list,还有util:set, util:map, util:constant等 2. 导入配置 3. 导入到Java配置 4. @import(CDPlayerConfig.class), 用于导入其他java配置；还支持导入普通的java类,并将其声明成一个bean 5. @importResource(“classpath:cd-config.xml”)，用于导入其他xml配置 6. 导入到xml配置 7. <bean class=”soundsystem. CDPlayerConfig”/>用于导入其他java配置 8. <import resource=”cd-config.xml”>，用于导入其他xml配置 9. 关联AB两个配置文件，不一定要在让A关联B，可以新增一个空配置文件C，里面仅仅把AB关联起来，这样A和B都可以使用对方定义的Bean |
| Spring注入bean有三种方式：自动化配置、基于Java的显示配置和基于XML的显示配置。推荐优先使用自动化配置，这样可以减少显示配置带来的维护成本，其次是Java显示配置，因为它更强大，支持逻辑。 | |

[logback][参数说明]

%M 方法名

%F 文件名

%L 行号

%-5level 日志级别，-5表示左对齐，用5个字符来输出

%logger{50} 当前类名，{50}不超过50个字符，超过了包名前面字段简写

[Spring in Action][第三章][]

|  |  |
| --- | --- |
| 2.存在重名bean，如何注入  3.Bean的作用域  4.非单例bean注入到单例bean  5.根据运行结果给bean的属性赋值 | 1. Spring通过profile来决定bean在应用时属于哪个环境(sit,prod)，不同于maven，各环境都是相同的包，具体参考3.1,3.2 2. 当Java定义中有两个或以上的同名的bean，可以使用@priority指定优先级；也可以在@Autowired的地方同时使用@qualifier（“xxx”）来指定实际类型，同时如果在类名上也通过@qualifier来指定别名的话，就可以做到与类名的松耦合，就算类名变了，也不影响注入。 3. Bean的作用域： 4. Singleton,单例 5. Prototype,原型，每次注入都创建一个新的实例 6. Session,每个会话创建一个bean实例，会话与给定的用户关联性最大 7. Request(每个请求创建一个bean实例) 8. 通过@Scope的proxyMode=ScopedProxyMode.INTERFACES，可以把非单例的bean注入到单例bean中，此时非单例bean会以代理的形式注入到单例bean中，可以编写代码调用它的方法，并且在不同的时刻，它可以是不同的值。 9. 运行时注入的方式： 10. 占位符 11. ${}； 12. @PropertySource(“配置文件路径“)声明配置源，注入Environment实例，使用getProperty()获取值； 13. @Value(“${xxx}”)，当使用自动扫描和自动注入时 14. SpEL, 非常灵活的表达式语言，用于运行时注入。 15. #{1}, 常量 16. #{systemProperties[‘sp.url’]} 系统属性 17. #{sgtPerppers.artilist}, bean的属性 18. #{sgtPerppers.selectArtist()}, bean的方法 19. T{java.lang.Math}.PI, 静态属性 20. T{java.lang.Math}.random(), 静态方法 |
| * 1. Spring可以在运行时给bean的属性赋值，方式十分灵活，除了指定值、配置文件值以外，还可以是方法的结果等。 | |

[MYSQL][别名][作用域]

当定义了别名如： select xxx as columnA 时，如果别名跟列名冲突，会在order by、group by会以别名的结果为准，但在select 和 where中以列名的结果为准。如果定义了别名跟列名一致，又想在order by使用列名，可以用以下方式：

Select xxx as columnA, columnA as org\_columnA from yyy order by org\_columnA

[Redis][lua][redisTemplate]

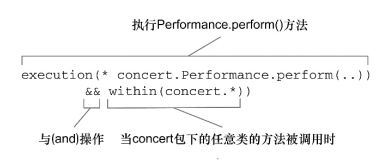
1. 通过lua脚本实现功能时，如果该参数是个数字，不能用JdkSerializationRedisSerializer来序列化，这样会解析不了。可以toString()以后再用StringRedisSerializer进行序列化。
2. RedisTemplate中的execute(RedisScript<T> script, List<K> keys, Object... args)，keys会用KeySerializer序列化，args会用ValueSerializer序列化。
3. 综上所述，如果需要的是数字，而且ValueSerializer是JdkSerializationRedisSerializer，KeySerializer是StringRedisSerializer的话，那么该参数只能放到keys里面，否则会报错：ERR value is not an integer or out of range

[大型网站技术架构][第六章][永无止境：网站的伸缩性架构]

|  |  |
| --- | --- |
| 1-3. 为什么系统需要伸缩性  4. 网站的伸缩性设计分裂  5. 应用服务器集群的伸缩性设计方案  6. 应用服务器的负载均衡算法  7. 缓存服务器和应用服务器在分布式上的区别  8. 缓存集群在伸缩时如何尽量高的保持命中率 | 1. 对于用户需求不明确的项目，或者是小创业团队项目，不可能一开始就构造一个大型系统，这需要十分大的成本。 2. 系统在某些时间段对于资源的需求量可能会大大增加（双十一），接着又回到原本的状态。   因此，   1. 系统的伸缩性显得十分必要。 2. 网站的伸缩性设计可分为两类： 3. 根据功能进行物理分离来实现伸缩，就是在不同的服务器上部署不同的功能模块。在这里又涉及到两种分离模式： 4. 纵向分离，就是在系统的整个处理流程中，将一、二、三等步骤分别部署在不同的机器上。也可以是分层分离，就是说将应用层服务，可复用服务、基础技术服务及数据库服务分别部署在不同的机器上。 5. 横向分离，就是在每一个步骤上，可能有多种可选服务、页面，将这些功能或者一个页面部署在不同机器上。 6. 单一的功能通过集群来实现伸缩，再强的服务器都是有限制的，如果可以通过集群，把单个功能部署到集群上，并且让该功能的处理能力与机器个数成线性关系，就不用担心用户数的不断扩展。 7. 应用服务器集群的伸缩性设计： 8. HTTP重定向负载均衡，就是用户请求先到达负载均衡服务器，负载均衡服务器通过HTTP302响应码重定向，让用户重定向请求目标服务器。   优点：实现简单  缺点：两次请求，性能较差。重定向服务器可能成为性能瓶颈   1. DNS域名解析负载均衡，就是通过DNS服务器中保存了多个目标服务器IP，它来决定具体要访问的目标服务器   优点：负载均衡交给DNS域名服务商管理，无需自己维护。DNS还支持地理位置的域名解析，可以得到最近的服务器地址，加快访问速度。  缺点：DNS是多级解析，就是如果某个服务器下线了，需要较长时间才能让多级的DNS服务器都生效。这时就可能会访问到无效的服务器。  尽管如此，大型网站还是会以DNS域名解析作为第一级的负载均衡手段，就是DNS会返回一组负载均衡服务器，接着这组负载均衡服务器再决定真实的目标服务器   1. 反向代理负载均衡，就是在反向代理服务器通过转发HTTP请求到目标服务器   优点：负载均衡和反向代理服务器集成到一起，部署简单  缺点：转发HTTP请求，这是应用层的负载均衡，该反向代理服务器可能成为性能瓶颈   1. IP负载均衡，在负载均衡服务器中通过内核进程获取网络数据包，通过修改源地址（负载均衡服务器IP）和目标地址（处理服务器IP），将请求发送到目标服务器来处理，处理完返回到负载均衡服务器，因为请求的地址是负载均衡服务器地址，目标服务器地址不一样，它不能跟用户直接通信。   优点：通过内核程序改IP，不需要进程处理，处理性能比反向代理好  缺点：所有请求都要经过这个负载均衡服务器，对于下载或者视频服务等大量数据传输服务来说，带宽成为瓶颈。   1. 数据链路层负载均衡，通过虚拟IP技术，使目标服务器集群的IP地址和负载均衡服务器IP地址一致，请求先到负载均衡服务器，通过内核程序修改mac地址到指定的目标服务器，处理完后目标服务器直接返回给用户。因为目标服务器IP和数据请求目的IP(负载均衡服务器IP)一致，不需要通过负载均衡服务器进行地址转换，可以直接返回到用户   优点：内核处理，处理性能好。目标服务器直接返回到用户，即使是大量数据的请求，不影响负载均衡服务器的带宽  Linux平台上最好的链路层负载均衡开源产品是LVS（Linux Virtual Server）   1. 负载均衡算法： 2. 轮询，依次分发到每台应用服务器上 3. 加权轮询，按照配置的权重分发到服务器上，高性能的服务器能分配到更多请求 4. 随机，包括简单随机和加权随机 5. 最少连接 6. 源地址散列，通过IP进行Hash得到服务器，这样同一个IP总是请求到同一个服务器上，该请求的上下文信息可以保存在这台服务器上 7. 分布式缓存集群跟应用集群不一样，不能分配到任意机器来处理，必须到存有需要数据的机器上。 8. 扩展缓存集群时，普通的Hash算法会在获取数据时因为机器数量变化了，导致解析路由错误，每增加一台机器的错误命中率为N/N+1(N为原有节点数)，但原有节点比较大时，错误概率接近1，最后压力都到数据库上了。解决方法： 9. 在访问量最小的时候扩展，并通过模拟请求预热缓存，使数据重新分布 10. 一致性Hash算法 11. 一致性Hash算法说明： 先构造一个[0,2^32-1]的整数环，根据节点IP的hash值取模后将节点放在环上，再根据key的hash值顺时针找到最近的一个节点。     1. 当节点发生变化时，可以尽量保持命中率。如下图：     在新增节点N4后，只有加粗段的key会受到影响，概率为25%，远低于普通求余75%。   * 1. 但是，上述方法还存在两个问题：      1. 不同的机器性能可能不一样，怎么有效的做到性能高的机器负载高也是个问题      2. N1到N4在环上的分布式基于ip % 2^32得出的，像这样节点不多时，4个可能连续紧密地分布在环上，导致数据倾斜      3. 只有N1被减少压力了，N2和N3没变，我们希望的是，在理想情况下，新增的节点能够均匀减少少有节点的压力   为了解决以上问题，我们引入了虚拟节点   * 1. 虚拟节点，就是给每个物理节点虚拟出很多个虚拟的节点，分布在这些虚拟节点上的数据，都会落到所属的物理节点上，它可以达到以下效果：      1. 让性能好的机器模拟更多的虚拟节点，负载更多的数据，实现权衡式负载均衡      2. 让圈上有足够多的节点，不至于节点太少，数据分布倾斜      3. 因为每新增一个节点，就能减少一个节点的压力，如果新增一堆虚拟节点，就能减少另一堆虚拟节点的压力，如果这堆虚拟节点恰好均匀分布在各个物理节点上，就能均匀的减少物理节点的压力。假设每台物理节点性能一样，每个物理节点受影响的数据量为其缓存数据量的X/(N+X),N为原有物理节点数，X为新加入的物理节点数   这个虚拟节点的倍数到底设置多少呢？除了根据机器的性能不一样设置大小不一样外，也不建议设置太大，这样会影响查询的性能，一般设置为150 |
|  | |

[Spring in Action][第四章][]

1. AOP的好处：
2. 每个关注点都集中于一个地方，而不是分散到多处代码中
3. 服务模块更简洁，只包含核心功能的代码，而次要的关注点的代码被转移到切面中
4. AOP的术语：
5. 通知，切面要执行的工作以及何时执行这个工作，如之前、之后等。
6. 连接点，执行通知的时机，具体指调用方法时、抛出异常时、甚至修改字段时等，它指能够应用通知的**所有点**。区别于通知，连接点的时机是描述什么情况会执行通知，而通知的时机指在这个情况下的之前还是之后执行。
7. 切点，定义通知在何处执行，通常是明确的类和方法名，或者正则表达式来匹配，它定义了哪些连接点会得到通知。
8. 切面，通知和切点的结合，定义了何时、何处完成什么功能。
9. 引入，向现有的类添加新方法或属性
10. 织入，切面应用到目标对象并创建新的代理对象的过程。AspectJ的织入编译器在编译期织入，AspectJ5的LTS在类加载期织入，Spring AOP在运行期织入
11. 定义切点的示例：





1. within(concert.\*)指定仅匹配concert包
2. bean(‘woodstock’)指定仅匹配bean woodstock
3. &&是与，xml中是and
4. ||或，xml or
5. !非, xml not
6. 启用AspectJ的方式
7. XML: <aop:aspectj-autoproxy>
8. JavaConfig: 在类中@EnableAspectJAutoProxy, 那这个config类是专门用于aop的吧，应该跟其他配置分开吗？应该不用，如果不是在aop代理类，@Bean也只是初始化普通bean。尽管是@Aspect，如果不是在@EnableAspectJAutoProxy下定义，就是一个普通的POJO

[Spring in Action][第五章][构建Spring Web应用程序]

|  |  |
| --- | --- |
| 1. 跟踪Spring MVC的请求过程 | 1. 跟踪Spring MVC的请求：      1. 请求离开浏览器时带上用户所请求内容的信息到达DispatcherServlet，至少包含URL，还可能有其他信息，如表单信息或Json内容 2. DispatcherServlet根据请求的URL来查询一个或多个处理器映射(handler mapping)来确定控制器 3. 控制器处理用户提交的信息，完成后产生的结果信息称为模型(model)，还会带上视图(view)的逻辑名称 4. 控制器将模型数据打包，并带上用于渲染输出的视图名返回给DispatcherServlet 5. DispatcherServlet使用视图解析器将视图名匹配为一个特定的视图实现，它可能是也可能不是JSP 6. 视图将使用数据模型渲染输出 7. 将输出通过响应对象返回给客户端 |
|  | |

[Mybatis][Mapper设计与查询范围]

1. Mapper.xml尽量通用，通过<if test="xxx != null" >进行where条件的添加，这样如果上游传null，就查询全部。针对可以查询全部的接口，controller层无需此变量入参，mapper方法调用时传null就好。针对只能通过变量查询的，在controller层用@requestParam控制为必填。
2. 针对in的条件，因为需要用foreach拼接，如果传入的list没有值，sql会报错。可以先校验当list.size()=0时，异常处理。当想查询全部时，整个list传null就好，因为<if test="xxx != null" >,就不会添加过滤条件。

[Lambda][分页][类型转换]

代码：

public static <T, R> PageInfo<R> pageTransform(List<T> source, Function<? super T, ? extends R> mapper) {

if(!(source instanceof Page)) {

throw new RuntimeException("source is not a Page!");

}

List<R> targeList = source.stream().map(mapper).collect(Collectors.toList());

PageInfo<T> sourcePageList = new PageInfo<>(source);

PageInfo<R> targetPageList = new PageInfo<>();

//这样可以保存原始的分页信息到新的结果集

BeanUtils.copyProperties(sourcePageList, targetPageList);

targetPageList.setList(targeList);

return targetPageList;

}

使用方式(ApiInfo -> ApiInfoVo, 其中ApiInfoVo内有ApiInfo作为入参的构造方法)：

PageUtils.startPage(currentPage, pageSize);

List<ApiInfo> sourceList = queryFromDB();

PageInfo<ApiInfoVo> targetPageList = PageUtils.pageTransform(sourceList, ApiInfoVo::new);

[MyBatis][读取类定义常量]

${@<类路径>@<变量名>}

e.g.

l.project\_name != ['${@com.sf.draw.openapi.constant.Constant@PLATFORM\_INVOKE\_LOG\_NAME}'](mailto:'${@com.sf.draw.openapi.constant.Constant@PLATFORM_INVOKE_LOG_NAME}')

注:这里不能用#

[公司项目][draw-realtime][maven build]

命令：mvn clean package -P prod

Build之前先把target文件夹删了，可能解决prod构建出sit配置文件的问题

[公司][OpenAPI][创建接口]

URL:

<http://profile.sf-express.com/aaa/api-list-manage/save-api>

Post:

{

"apiName" : "30天内散单用户",

"projectName" : "ESG-CEMP-CORE会员权益管理",

"apiType" : "个人标签",

"bigTableName" : "寄件标签",

"apiDefinition" : {

"inputParamsDef" : [

{

"encryptType" : "Mobile",

"field" : "mobile",

"type" : "equal"

}

],

"outputFieldsDef" : ["consignor\_30d"],

"unique":true

}

}

Post:

{

"apiStatus":"1",

"id":xx

}

[Lambda][好处与限制]

好处：

1. 具备了将函数逻辑作为参数传入方法的能力，实现code as data。逻辑可以作为入参的话，可以构建更加通用的方法，比如一个多步骤的方法，只有中间一步是不同类型要不同处理的，那可以把这个步骤抽出来作为方法入参，其他一致的步骤定义在方法里面就好。
2. Lambda Expression 使得代码量更少，因此代码更简洁、可读性更强（适合仅使用一次或几次、对复用要求不高的代码）。
3. JDK 8 中的 lambda expression 可以和 aggregate operations 一起使用，处理collection和map更加方便，还支持并行处理
4. 定义的逻辑不需要为它考虑方法名

劣势：

1. 如果lambda expression 可以和 aggregate operations 一起使用替代foreach的话，不方便调试，里面就一行的代码
2. 如果不是用并行处理，性能还没有foreach高。

[ElasticSearch][match]

当macth匹配keyword时，效果跟term一样，需要完全匹配

当match匹配text时，对入参进行分词，分词结果集中默认如果有任意一个分词属于该字段的分词集当中，则匹配。也可以设置操作符为AND，必须入参分词结果集中所有分词都属于该字段的分词集合中，才匹配。

e.g. :

"match" : {

“name” : “aa bb cc”

}

"match" : {

"baseDayTag" : {

"query" : "BD8-华东,BD8-513Y",

"operator" : "AND"

}

}

[Java][Spring][定时任务]

Spring开启定时任务的方式：

1. 在类上添加@EnableScheduling注解，spring4需要加上，不一定每一个定时任务类上面都要添加，但至少项目代码中至少有一个类被注解了。
2. 添加<task:scheduler id="dataScheduler" pool-size="5"/>，如果不加，系统默认单线程执行所有定时任务，如果有一个任务阻塞了，其他任务据无法执行。添加后可以多线程执行。
3. 在执行方法上添加@Scheduled(cron = "0/10 \* \* \* \* \*")注解，定义执行频率。

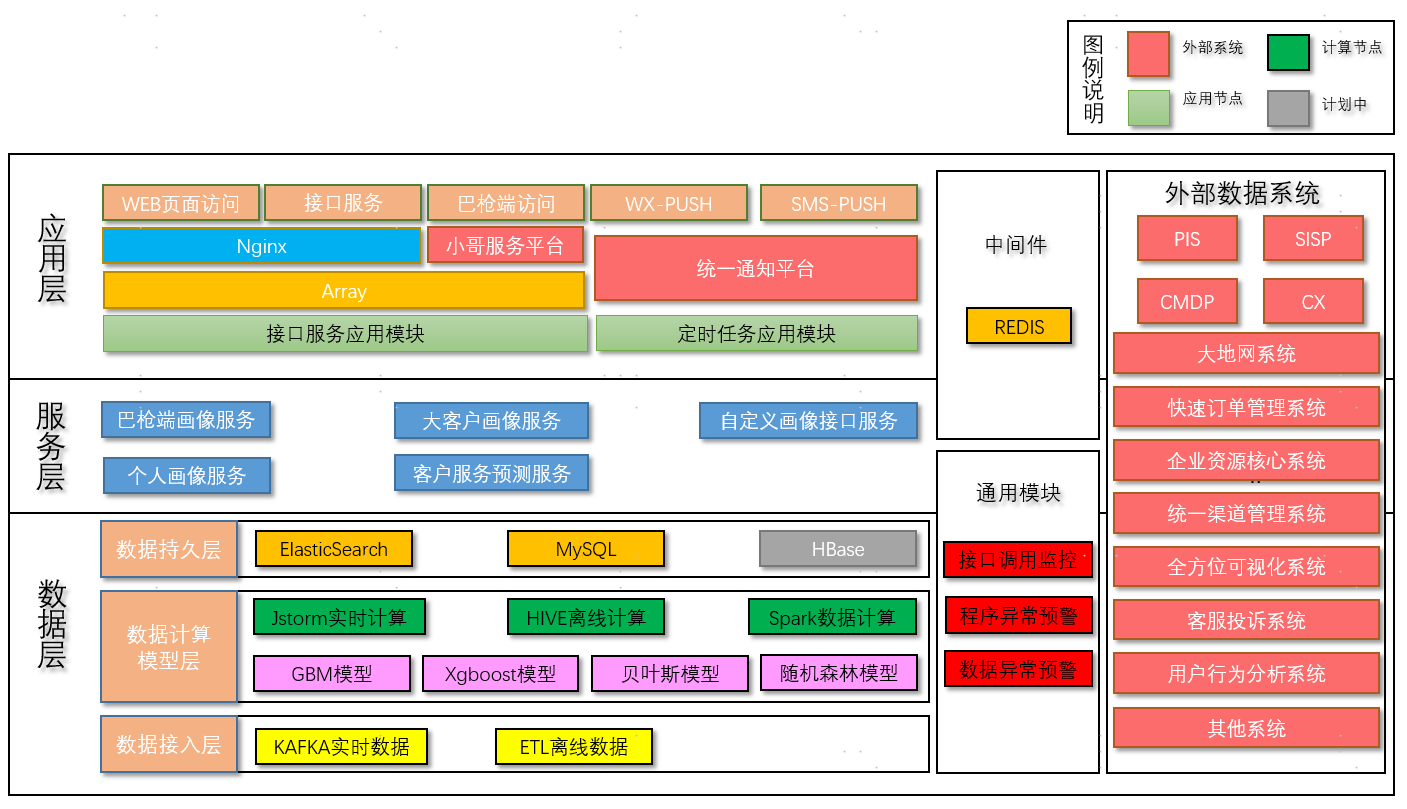
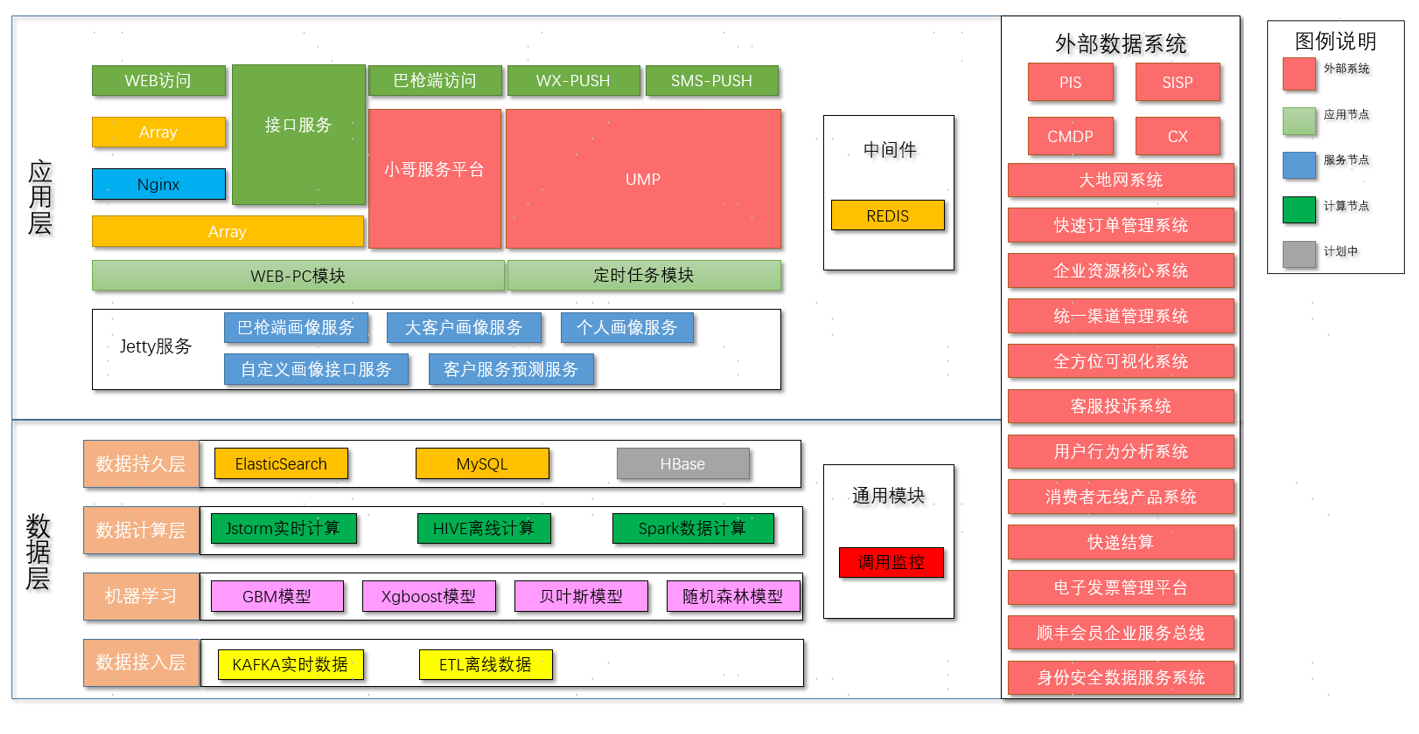
--https://cloud.tencent.com/developer/article/1383486

[ES][聚合]

在聚合的terms中，使用"min\_doc\_count"可以过滤结果中doc\_value(记录出现次数) < min\_doc\_count的数据。

[架构图][应用架构图][对比两次画图总结所得]

这是丰像的应用架构图的（**左边初稿，右边终稿**）：



通过调整总结如下：

1. 逻辑性，就是从上层是依次调用下层，下层是为上层提供服务。
   1. “WEB页面访问”和”接口服务”都会调用”Nginx”。就算是“接口服务“，通过访问公共地址(前端地址)来调用服务，而不是直接访问后端服务器域名，暴露后端
   2. 把“数据计算层“和”机器学习层“合并成”数据计算模型层“，因为它们是一个整体，在计算节点上，使用模型来生成结果数据。
   3. 把服务层独立出来
   4. 还知道了，原来前端包和nginx是在同一个服务器上面，nginx可以存放静态资源(CDN)，同时还可以用于转发
2. 整体性，各模块间尽量保持大小一致，保持整体性
   1. 把”中间件”和“通用模块“调大，让整个”系统内部模块” 、”中间件”、“通用模块“和”外部数据系统“看上去是一个整体
   2. 把“接口服务“、”小哥服务平台“和”UMP”调小，这样看起来跟旁边的标签是同一层的，
3. 紧凑度，一般来说每一个标签的大小相等，但如果空出太多空间，可以适当把该标签拉大，让整体更紧凑美观
   1. 把“统一通知平台“稍微调大，不至于上下留出太多空隙
   2. 各标签之前的空隙也调小了
4. 外部使用的源过多，如果影响美观的话，可以使用”其他系统”来表示

[Java][Web项目][对象命名]

VO(View Object)：页面对象，用于页面和Controller层之间的交互

DTO(Data Transfer Objct)：数据传输对象，泛指Controller层与service层之间的对象传输

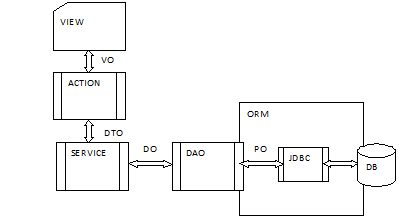
DO(Domain Object) : 领域对象，通常指与数据库或其他系统之间进行传输的对象

DAO(Data Access Object) : 数据访问对象，指用于访问数据库或其他数据存储接口的对象，在mybatis中与mapper对等

PO(Persistent Object) : 持久化对象，在数据库中，一行记录就是一个PO，记录中有多少个字段，该PO就有多少个属性

POJO(Plain Ordinary Java Object) ： 简单的Java对象, 以上的都是POJO

Web项目层级交互图：



[Kafka][性能][Kafka为了追求极致的性能，有这11大牛逼之处]

https://mp.weixin.qq.com/s?\_\_biz=MzU5ODUwNzY1Nw==&mid=2247484668&idx=1&sn=341162fc4531ea00876be22577b8fa6c&utm\_source=tuicool&utm\_medium=referral

|  |  |
| --- | --- |
| 1. 什么是broker  2. 批量处理  3. 客户端优化  4. 日志格式  5. 日志编码  6. 消息压缩  7. 建立索引  8. 分区  9. 一致性  10. 顺序写盘  11. 页缓存  12. 零拷贝 | 1. Broker是缓存代理，Kafka的一台或多台服务器统称Broker，它不保存订阅者的订阅状态，由订阅者自己保存 2. 对于生产者，发消息到broker，broker响应ACK，需要两次rpc；对于消费者，请求消息，broker发送消息，消费者响应ACK，需要3次rpc。在批量处理下，会累积到一定数据量再发送，如果是1000条消息，生产者只需要2次rpc，而在单条处理的情况下，生产者需要2000次rpc。Kafka的批量处理是微批次理念，目前比较流行 3. 客户端从单线程优化到双线程，主线程负责将消息放到客户端缓存，Sender线程负责从缓存发送消息。而消息在缓存中也会累积到一定批次再发送 4. Kafka日志格式有利于批量消息的处理。RecordBatch中的每一个record，不再保存绝对的timestamp和offset，而是保存delta，相对应于RecordBatch中的timestamp和offset 5. Kafka的日志中除了key、value外，还有一些其它字段，它们原本是固定大小的，在引入Varints和ZigZag编码后，实现这些字段的可变长。日志变小了，网络传输和存盘效率提高 6. 支持多种消息压缩方式:gzip、snappy、lz4。压缩后极大减少网络传输量，降低网络I/O，从而提升整体性能。不过这是一种用时间换空间的方式，如果对消息的时延有一定要求的话，不推荐压缩   --关于kafka的日志内容，  V1版本：  加入了timestamp（对内部而言，方便了日志保存、切分策略，对外部而言，方便了消息审计、端到端延迟等功能的扩展，虽然可以在消息体内部添加一个时间戳，但是解析变长的消息体(value是变长的)会带来额外的开销）  消息压缩  V2版本  加入了可变长编码  用相对值表示timestamp和offset  Kafka日志参考：  https://mp.weixin.qq.com/s?\_\_biz=MzU0MzQ5MDA0Mw==&mid=2247483983&idx=1&sn=1c2bd11df195f84e5433512f6b2695e8&chksm  =fb0be8dbcc7c61cda63c599769b38a78c08e51ab8a2943985040014dcbcc1cbe1547d3f70ee6&scene=21#wechat\_redirect   1. 每个日志分段文件对应两个索引文件，提高查找消息的效率 2. 分区是Kafka中的最小并行操作单元，对于生产者，每个分区的写入操作完全可以并行化；对于消费者，Kafka只允许单个分区的消息被一个消费者线程所消费，一个消费组的消费并行度完全依赖于所消费的分区数。但并不是分区越多，吞吐量就越高，当分区数达到一定阈值后，吞吐量不升反降 3. 为了保证一致性，Kafka使用了Pacific-A替代Raft一致性协议，这种模型会提升整理的效率 4. Kafka是典型的顺序写盘操作，采用文件追加的方式写入消息，只能在日志文件的尾部追加新的消息，不允许修改已写入的消息。这是操作系统会对线性读写做优化，如预读（read-ahead，提前将一个比较大的磁盘块读入内存）和后写（write-behind，将很多小的逻辑写操作合并成一个大的物理写操作），提高整体吞吐量 5. 大量使用了系统的页缓存机制，减少硬盘的访问。在读数据时，先在页缓存中查找，有则直接返回，没有则再读写硬盘，并存入页缓存。写数据也一样，先存入页缓存，再在合适的时间写入、更新数据到磁盘，以保持一致性。 6. 页缓存对比进程内部缓存的优势： 7. 进程内部缓存的对象开销大，一般是真实数据大小的几倍甚至更多。页缓存可以使用结构紧凑的字节码替代对象来节省更多的空间 8. Java的垃圾回收随着数据量增多而变慢 9. Kafka服务重启后，页缓存仍然有效，而进程内的缓存需要重建 10. 简化代码逻辑，把页缓存和文件之间的一致性交由操作系统负责，不用自己写代码 11. 使用了零拷贝技术, 数据传输中减少了两次拷贝，大大提高了传输性能 |
| 总结：  Kafka主要通过批量处理消息、优化日志大小、分区、使用更高效的一致性协议，以及利用系统自身的I/O优化技术来提高性能。  参考：  https://mp.weixin.qq.com/s?\_\_biz=MzU5ODUwNzY1Nw==&mid=2247484668&idx=1&sn=341162fc4531ea00876be22577b8fa6c&utm\_source=tuicool&utm\_medium=referral | |

[系统][IO][Zero Copy]

零拷贝技术大大提高了数据传输的速度，我们以数据往外发送为例：

1. 原始的拷贝方式：
2. 调用read，通过DMA将文件从磁盘copy到kernel buffer中（user context -> kernel context）
3. CPU将文件从kernel buffer copy到user buffer（kernel -> user）
4. 调动write，将文件从user buffer copy到kernel下的socket buffer中（user -> kernel）
5. 最后将kernel模式下的socket buffer的数据copy到网卡设备(protocol engine)中传送 (kernel -> user)

在这个过程当中，上下文切换了4次，文件也被拷贝了4次

1. Java NIO的FileChannal.transferTo()利用操作系统底层的sendFile实现后：
2. 通过DMA将文件从磁盘copy到kernel buffer中(user -> kernel)
3. 然后将kernel buffer 数据copy到socket buffer中(kernel)
4. 最后将socket buffer中的数据copy到网卡设备(protocol engine)中(kernel -> user)

这个过程，上下文切换了2次，文件拷贝了3次

1. Zero Copy:
2. 通过DMA将文件从磁盘copy到kernel buffer中(user -> kernel)
3. 向socket buffer中追加当前要发送的数据在kernel buffer中的位置及偏移量(kernel)
4. 根据socket buffer中的位置及偏移量，直接将kernel buffer中的数据copy到网卡设备(protocol engine)中(kernel -> user)

这个过程，上下文切换了2次，文件只被拷贝了2次

上下文切换4次->2次，文件拷贝4次->2次，大大提高了IO处理速度

参考：

[https://mp.weixin.qq.com/s?\_\_biz=MzU0MzQ5MDA0Mw==&mid=2247483913&idx=1&sn=2da53737b8e8908cf3efdae9621c9698&chksm=fb0be89dcc7c618b0d5a1ba8ac654295454cfc2fa81fbae5a6de49bf0a91a305ca707e9864fc&scene=21#wechat\_redirect](https://mp.weixin.qq.com/s?__biz=MzU0MzQ5MDA0Mw==&mid=2247483913&idx=1&sn=2da53737b8e8908cf3efdae9621c9698&chksm=fb0be89dcc7c618b0d5a1ba8ac654295454cfc2fa81fbae5a6de49bf0a91a305ca707e9864fc&scene=21" \l "wechat_redirect)

[系统][CPU][kernel模式与user模式]

kernel模式与user模式指的是CPU的两种模式。

CPU会通过这两种模式控制应用程序的行为，防止不了解系统的开发者通过应用程序对任何I/O设备拥有全部权限，执行一些危及系统安全的指令，导致系统崩溃。

user模式，开发者编写的应用程序代码是在此模式中执行，普通的数值计算或变量指派都可以在此模式完成，但不能执行特权级指令，就是不能与kernel、内存及硬件设备进行交互, 比如不能往磁盘写入数据，。执行的命令权限是少了，但能够保证系统的安全。

kernel模式，可以执行特权级指令(system call)，在执行实际操作之前，会有一系列的检查，包括应用程序的权限及操作的内容（是否在自己的存储器范围，是否对文件有读写权限，是否把数据往错误的装置送过去等等）。OS开发者藉由定义system call告诉开发者们，哪些行为必须经过OS的过滤才能执行，否则可能危害系统。这些命令在发布前必须经过严格的测试，并且在执行时，还会进行一系列的操作检查，以保证安全。

这是模式间的切换，而不仅仅只是kernel及user存储空间中的切换。

参考：

<https://blog.csdn.net/zhuxiaoyang2000/article/details/8052324>

[系统发布][主客渠][发版问题总结]

事情描述：

主客渠的特色经济标签发版，交付工程师按照包名来判断部署机器，部署后发现没有生效，验了几次都还是没有生效。过了一会，再验一次，发现有一个生效了(一共两个校验，同一套代码)。这时猜测是不是部署的服务器错了或者部署不全，我问交付，他说这是丰箱部署的，不会出错的，后来查了一下应用服务器的包的更新时间，发现里面的日期不是今天(当时自己也看错了，以为是6月份的)，错误判断是否丰箱出问题了，无法部署到应用服务器，可是又存在疑惑，为啥有个请求能通过。接着我打电话给第一个运维，描述说丰箱无法部署，她态度不大好，说自己现在看也没用，要不明天再看，接着我就问有没有丰箱的同事联系方式，她提供了一个。到后面感觉找丰箱的同事好像也没啥用，我又打了电话找第二个运维，我问应用服务器的地址，虽然他也还是不大乐意处理，但也在帮忙，可是他不是负责这个系统的，不熟悉，我是通过给他服务访问域名，想查出后面跟着的应用服务器，可是这个域名后面跟着的是一堆的nginx，nginx再转发到应用服务器，如果不一一查ng的配置，是找不到应用服务器地址的。搞了半个小时，还没有给到正确的地址。这个时候，我再确认了一下，发现服务器上面的包时对的，更新的时间是我提交代码的时间，我们把应用服务器关闭到只剩下一台，调接口时发现请求没有进去，这时能肯定是应用服务器部署错误或不全。我打电话给李喆林，他告诉我一共8台服务器，而我们之前只部署了四台，在沟通中，猜测部署错了集群，我看运维页面，部署的应用里面有sw关键字，还有另外一个没有sw关键字的，我当时觉得应该是部署错了到另一个模块，经过沟通确认是这样的，同时还告诉我验证脚本最下面有那8台应用服务器的地址(脚本很长，一开始没有看完，只拿了自己需要的内容)。最后我们先恢复sw模块，验证该模块，再发版这次的内容到正确的服务器上，再验证，发现接口是通了，可是没有结果，排查问题，通过log获取查询dsl，到堡垒机查询，也查不到结果，发现原来是数据缺失，数据不是我负责的，微服务没问题了，发版完成！

总结：

主客渠问题：

1. 没有告诉我应用服务器的ip地址
2. 说话态度很不积极，开发说有事情都不要找他，大不了第二天再发。问他有没有什么要注意的，他说什么都不用。这样使我不大好意思问他。运维也十分不积极，说她上线也没啥用，也会导致我不好意思问她过多的问题。
3. 两个不同模块的zip包名竟然是一样的，如果是区分开来，就不容易误会

交付工程师的问题：

1. 直接按zip包名字来找要部署的应用模块，应该先跟我确认到底要部署到哪个模块，而不是自己直接就部署上去

我的问题：

1. 没去问应用服务器的ip地址，如果一开始就知道，后面全部问题都不会发生。**以后必须知道系统所有组件的ip地址**
2. 看到包的更新时间时不细心，如果发现是我代码更新时间，就不会误认为是丰箱部署问题。**jar包的时间是打包的时间，不一定是当天**
3. 担心打扰到别人，或者别人认为我这都不懂。**不要不好意思，这就是他们该干的事情，该问就直接问，一开始可以礼貌，但如果他们态度还是不好，可以强硬一些。你最后没解决，别人还更加怀疑你的能力呢！**
4. 一直没重视发版的时间，到后面拖到客户上线的前一天才发版，导致发版紧急，必须当天解决。**以后必须安排好上线时间，至少比客户上线时间提前两个工作日，这样就算当天没解决，第二天也可以继续看，工作时间找别人帮忙也方便。**

[待验证][大数据组件][ES v.s. HBASE][key-value场景]

读数据过程：

ES：

1. 文档id通过hash得到对应的分片id。通过RoutingTable, 可以根据索引名称找到所有的分片，再通过hash找到对应的分片id。
2. 通过分片id得到对应的node节点
3. 在节点上，基于type和文档id合成一个键，在倒排索引中查到对应的segment及doc\_id(doc\_id是指在segment里面的第几行记录，区别于第一步的文档id)
4. 根据doc\_id在segment中从正向信息获取内容

总结：ES的返回时间比HBASE短，但需要的运行资源更多，因此并发量小于HBASE

缓存：

ES中缓存的是索引，在通过索引获知数据的具体位置后，还需要访问硬盘获取数据

HBASE在memstore中缓存的就是key-value数据，缓存中命中就直接返回结果，这会更快

IO：

为了提高性能，数据在磁盘中必须有序，如果使用B+树，在key跨度很大的情况，新插入的数据会存在磁盘很远的地方，产生大量的随机IO，磁盘寻到速度跟不上

ES会有许多散乱的文件？？？？，而且ES在写的时候还要维护倒排索引

HBASE使用了LSM树，在损耗一定读性能，来提高写性能，定期合并storefile，不产生过多的小文件。？？？HBASE在写入时不需要维护额外的索引

资源的使用：

硬盘：

ES会产生很多的倒排索引，这些索引甚至比原本的数据还要大很多

HBASE不会有这么多索引文件，key-value的映射本身就是一份索引，总的数据量远远小于ES

内存：

ES主要用于复杂的值过滤查询，在运行中，需要很多的资源包括缓存来加快这些复杂查询

HBASE主要就是做简单key-value查询的，不需要额外的资源做复杂操作，可以有更多的资源支持更高的并发量

还有的疑问：

1. ES中RoutingTable的具体内容，它如果通过id查到node
2. ES在缓存中放的是索引吗？还有其他什么内容？如果加快之前已做过的查询？
3. ES会产生大量的随机IO吗
4. HBASE的LSM树到底是啥
5. 对于资源的使用-内存的描述，是否正确

[待验证][大数据组件][ES][基本概念][vs MySQL]

|  |  |
| --- | --- |
| MySQL | ElasticSearch |
| Database | Index |
| Table | Type（在6.x后，一个index只能有一个type） |
| Row | Document |
| Column | Field |
| Schema | Mapping |
| Index | Everything is indexed |
| SQL | Query DSL |
| Select | GET |
| Update | UPDATE |
| Insert | PUT/POST |
| Delete | DELETE |

ES的写入过程

1. 新增的文档首先会被存放在内存的缓存中(此时数据不可读)
2. 当文档数足够多或者到达一定时间点时，就会进行refresh
   1. 生成一个新的segment，并请求写入磁盘
   2. 在写入磁盘前操作系统会先写入文件系统的缓存(一个内存空间)(这是操作系统的机制)，这个过程耗时极小

现象: 这个segment中的文档在refresh后可以很快被搜索到。因为处于文件系统的缓存也是可读的，所以在segment被真正保存到磁盘之前，就打开了对应的segment

* 1. 清空缓存，准备接收新的文档

1. 重复步骤2，当文件系统的缓存中的segment达到一定数量/时间(系统默认值或ES参数指定)，会执行操作系统的flush操作，等待所有segment写入磁盘
2. 磁盘中生成一个新的commit point，记录当前所有可用的segment
3. 清空translog日志

[大数据组件][ES][聚合terms不精确问题及解决方案]

ES的聚合terms会按照doc\_count排序，默认从高到低返回size条结果。

ES处理terms流程如下：

1. 每个分片内部先计算每个term的doc\_count
2. 排序doc\_count，把前shard\_size个结果发送到收集节点
3. 收集节点汇聚所有分片来的数据，再排序出前size个结果返回

这里会产生问题：

1. 每个分片上的前几名，并不一定是整体的前几名
2. 就算排名对了，算出来的doc\_count也可能不对，比如第一名的count在某一个分片中只有几个，它就不会被提交到收集节点

解决方案：

没有绝对好的解决方案，要看不同场景

方案一，只设置一个分片，适用于小数据量

优：实现简单，支持不同字段的terms聚合

劣：如果数据量太大，并发量也高，查询性能不高

方案二，把terms的字段作为判断分片的route，适用于只有一个字段需要terms聚合

优：如果单个查询的数据在同一个分片，简单查询的效率更高

劣：这只能解决当前字段的terms精确问题，如果其他字段也要terms统计，还是不准确

方案三，把shard\_size调大，适用于terms后结果个数不多的情况

优：实现简单

劣：shard\_size的大小对于性能影响很大，主要使用于terms后结果个数不多的情况。而且最大也只能是Integer.MAX\_VALUE，如果terms后个数大于Integer.MAX\_VALUE，结果还是可能不精确，但还是能够提高精确度.

[大数据组件][ES][思考][数据在单个分片vs多个分片]

考虑这个问题，需要清楚三个事情：

1. 查询的复杂度，复杂度包括命中数据量及dsl的复杂度，如果比较复杂，单台处理不过来，多个分片应该更好
2. 计算机的性能，如果计算机性能比较好，单台应该就能处理大部分请求，单个更好
3. 并发度，如果并发度很高，每个分片的处理需要一个线程，在多个分片的情况下，就需要多个线程来处理，但es的线程池有限，后面的请求可能要等待，而且线程的切换也有消耗，影响返回时间。单个分片更好

[大数据组件][ES][分片原则]

1. 为了防止机器损坏导致的数据丢失过多，恢复时间过长，分片大小不能太大
2. 分片太多的话，也会导致每一个请求为了触发全部分片，产生过多的线程，ES的线程池有限，在高并发量的情况下，部分请求响应延时
3. 所以一般分片大小在20-50G之间
4. 同时注意让每一台都相应的分片，且数据分布均匀，从而达到负载均衡，且所有服务器资源都能用上

[Git][Bash基本使用]

1. 获取项目

git clone <克隆地址>

1. 更改文件后，添加到git目录

git add <.|文件路径>

1. 提交到local repository

git commit –m “<提交内容描述>”

1. 同步到remote repository

git push <远程主机名(默认是origin)> <本地分支名>:<远程分支名>

(分支名相同，则本地分支和远程分支都可以省略 git push 即可)

其他使用：

1. 替换分支
   1. 查看所有分支

git branch –a

* 1. 创建本地分支并切换，切换到origin/release1.0分支命令，创建本地分支为”release1.0”

git checkout -b release1.0 origin/release1.0

* 1. 切换回已存在的分支

git checkout <本地分支名，如master>

1. 拉去更新

git pull <远程主机名(默认是origin)> <远程分支名>

git pull 不带参数，默认向当前跟踪的分支拉取

[ES][6.1.2][运行]

1. 先把环境变量的HADOOP\_HOME去掉，因为我只有一个文件，不完整

set HADOOP\_HOME=

1. 进入es的目录

cd D:\IdeaProject\elasticsearch-6.1.2

1. 机器已经配置了最新的Java，但该版本只支持JAVA8，设置JAVA8

set PATH=C:\Program Files\Java\jdk1.8.0\_152\bin;%PATH%

set JAVA\_HOME=C:\Program Files\Java\jdk1.8.0\_152

如果还不行，彻底一些，set PATH=C:\Program Files\Java\jdk1.8.0\_152\bin

1. 启动ES

gradlew run --debug-jvm

1. 如果还是遇到HADOOP\_HOME的问题，可以清了gradle在重试

gradle clean

1. 如果还是启动异常，尝试重启
2. 如果发现数据丢失，可以参照以下命令：
   1. 创建索引

<http://localhost:9200/index1>

* 1. 创建type

<http://localhost:9200/index1/type1/_mapping>

{

"type1": {

"properties": {

"name": {

"type": "keyword"

},

"age": {

"type": "keyword"

}

}

}

}

1. 原文地址，还有编译运行的方式

https://blog.csdn.net/wwwdc1012/article/details/81978966

[MySQL][启动命令]

net start mysql

[Java][Spring][对象初始化构造器后运行]

@PostConstruct

[Zookeeper][由配置系统触发的总结]

|  |  |
| --- | --- |
| 1-2 Zookeeper的介绍  3 常用节点  4-5 watch机制的触发时间及特点  6 Zookeeper的常见应用场景  7-8 Curator的使用原因，maven配置  9 自己实现的配置功能总结 | 1. Zookeeper是分布式应用程序的分布式开源协调服务 2. 它具有高可靠（冗余结构）、高性能（分布式）、有序性、原子性等特点，对于操作节点及其子节点增删查改，可以通过ACL控制 3. 常用节点： 4. 临时节点 EPHEMERAL，会话关闭后自动删除 5. 持久节点 PERSISTENT，不会自动删除 6. 临时顺序节点 EPHEMERAL\_SEQUENTIAL，临时且路径最后有序号 7. 持久顺序节点 PERSISTENT\_SEQUENTIAL，持久且路径最后有序号 8. 容器节点 CONTAINER，当它的所有子节点被删除时，它也会被自动删除 9. 持久定时节点 PERSISTENT\_WITH\_TTL，带TTL过期时间 10. 持久顺序定时节点 PERSISTENT\_SEQUENTIAL\_WITH\_TTL，带过期且路径有序号   最后三种在3.5.3-beta以上才有   1. Watch机制，对节点watch，并在以下事件发生时，可以触发预先写好的程序逻辑 2. NodeChildrenChanged，子节点被创建或删除时 3. NodeCreated, 该路径的节点被创建时(就是说可以监听一个不存在的节点路径) 4. NodeDataChanged, 该节点的数据被改动时 5. NodeDeleted，该节点被删除时 6. Watch的特点： 7. 主动推送，zk服务器主动推送事件，不需要客户端轮询 8. 一次性，保证事件只推送一次 9. 可见性，如果节点变化了，是先收到watch通知，然后才会看到节点的变化 10. 顺序性，保证多个watch的触发顺序与变更顺序一致 11. Zookeeper的常见应用场景 12. 配置服务器：通过watch机制，做到配置的动态更新，主动同步到每台服务器的内存配置中，无需重启服务器，无需每次访问远程配置，也无需应用自己轮询配置是否发生变化 13. 集群管理：及时感知集群中服务器的加入、移除和异常。通过watch临时节点来实现 14. 命名服务：通过名字获取服务URL，如微服务就用到 15. 分布式计数器，通过有序节点来实现分布式系统的统一计数 16. 发布订阅，基于watch 17. 分布式锁，基于zk的顺序性，通过创建有序节点，判断自己是否为当前序号最小的，来执行功能 18. 队列管理，基于watch和有序节点，监听自己的前一个序号节点，在前一个节点消失时执行逻辑，完了就删除本节点 19. Curator的使用原因 20. 原生的zookeeper JAVA API不好用，我遇到了连接后无故断开的情况，而且watch只能单次生效，事件发生后，还要主动调用watch功能，才能触发下一次 21. Curator封装的很好，使用方便 22. 无需手动多次触发watch功能，单次配置就好 23. 使用后连接一直正常 24. Curator的maven配置，如果是3.5.X，可以直接使用，如果是3.4.X，需要排除里面包含的zookeeper，并且自己引入zookeeper依赖，如下：   <dependency>  <groupId>org.apache.zookeeper</groupId>  <artifactId>zookeeper</artifactId>  <version>3.4.10</version>  </dependency>  <dependency>  <groupId>org.apache.curator</groupId>  <artifactId>curator-recipes</artifactId>  <version>4.0.0</version>  <exclusions>  <exclusion>  <groupId>org.apache.zookeeper</groupId>  <artifactId>zookeeper</artifactId>  </exclusion>  </exclusions>  </dependency>   1. 关于配置功能的总结： 2. Map<String, String> localConfig = new ConcurrentHashMap(); 优势：使用ConcurrentHashMap，支持多线程操作，读不加锁，提高并发性能 3. 在初始化时(@PostConstruct)，先从数据库读取固定前缀的全部配置，一一查询ZK是否存在，如果不存在则写入ZK，在把ZK的值写入本地内存，并且监听这些配置的路径。优势：启动时就加载并监听配置；如果要加新的配置，加入数据库就行；通过前缀过滤，支持多套系统；ZK存在就读取ZK的配置，因为ZK的稳定性比DB高，DB更可能更新失败 4. 配置更新时除了更新ZK，还会更新数据库，返回三种状态，告知成功、失败、ZK成功DB失败。优势：保留一份配置在数据库，在换ZK时，可以无缝使用；三种状态方便告知用户具体情况 5. 目前set功能不支持create节点，因为就算create并且监听了，也仅仅是执行这个代码的节点会收到通知 |
| Zookeeper有watch功能、临时节点、有序节点，支持高并发，顺序性，高可靠，原子性操作，基于这些特点，可以辅助分布式系统实现多种协调性的功能。  参考：https://blog.csdn.net/qq\_34021712/article/category/9278741 | |

[Flink][Flink 从 0 到 1 学习 —— Apache Flink 介绍]

|  |  |
| --- | --- |
|  | 1. Flink的基石(技术特点) 2. Checkpoint，基于Chandy-Lamport算法，实现了分布式一致性快照，提供了一致性的语义 3. State，丰富的State API。ValueState，ListState，MapState，BroadcastState 4. Time，实现了Watermask机制，乱序数据处理，迟到数据容忍 5. Window，开箱即用的滚动、滑动、会话窗口，以及灵活的自定义窗口 6. Flink APIs，支持SQL、DSL、DataStream / DataSet API(流式语法)及状态流(函数语法) 7. Flink的处理过程 8. Source数据源，接收本地集合、文件、socket及自定义的source，自定义包括kafka、RabbitMQ等等 9. Transformation，数据处理，有Map/KeyBy/Aggregations/Window/Window join/Split/Select等操作 10. Sink，发送结果，支持kafka、RabbitMQ、MySQL、ES、hdfs等，还能自定义sink 11. Flink的优势： 12. 支持批量和流式处理 13. 支持无序和延时数据 14. 状态化容错，同时在维护一次完整的应用状态时，能无缝修复错误 15. 分布式，支持上千个节点。高并发、高吞吐、低延迟 16. 灵活的窗口，基于时间、计数和session 17. 轻量级的容错能力，可以保持高并发，Flink以零数据丢失的方式从故障中恢复，但没有考虑可靠性和延迟之间的折衷 18. 通过checkpoint，能以无丢失状态和最短停机时间的方式更新应用或回退数据 19. Flink的程序可以并行运行 20. 自己管理JVM，更有效的内存管理 21. 丰富的库，支持机器学习、图形处理关系处理等 22. Flink运行组件介绍： 23. Job Client，负责接收用户的代码，创建数据流并交给Job Manager，最后将结果返回给用户 24. Job Manager， 协调和管理程序的执行，包括给Task Manager安排任务、管理checkpoint、故障恢复等。集群至少有一个master，多个时只有一个leader，其他standby 25. Task Manager，接收Job Manager的job，分配给内部的Task Slot执行。它是JVM中执行任务的工作节点，有一个或多个线程组成 26. Task Slot，执行sub task的一个JVM线程，Task Manager包含一个或多个Slot，它们之间内存隔离，CPU不隔离。允许子任务共享Slot，只要它们来自相同的job 27. Flink的Job构成，一个完整的任务叫job，一个job分成多个subtask，几个subtask可以在同一个slot运行，这几个subtask称为task. 28. Flink的Time机制：     1. Processing Time，指事件被处理时刻的系统时间     2. Ingestion Time，指事件进入Flink Data Source的时间     3. Event Time，事件发生的时间，一般就是数据本身携带的时间，由事件自身的一个字段指定   Event Time是业务事件时间，如果需要按照业务时间有序处理，就需要使用Event Time。Processing Time跟Ingestion Time都是Flink自动生成的时间，它们的区别在于，如果两条消息同时进入Flink，但由于延时，使用Processing Time的话，两条消息可能被不同的时间窗口处理，因为有一条来慢了，要计算时已经到了下一个窗口，而Processing Time机制又是根据计算时间来确定窗口的。使用Ingestion Time可以保证在同一个窗口处理 |
|  | |

[Java][DUBBO][DUBBO消费者接入]

Maven包依赖：

<!-- 引入服务提供方接口 -->

<!-- 引入dubbo, 注意跟服务提供方版本一致 -->

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>dubbo</artifactId>

<version>2.8.4</version>

<exclusions>

<exclusion>

<artifactId>javax.servlet-api</artifactId>

<groupId>javax.servlet</groupId>

</exclusion>

</exclusions>

</dependency>

<!--Zookeeper curator-->

<dependency>

<groupId>org.apache.curator</groupId>

<artifactId>curator-recipes</artifactId>

<version>4.0.1</version>

<exclusions>

<exclusion>

<groupId>org.apache.zookeeper</groupId>

<artifactId>zookeeper</artifactId>

</exclusion>

</exclusions>

</dependency>

<!-- 必须引入hessian，不然服务不可用，但不会提示缺少这个 -->

<dependency>

<groupId>com.caucho</groupId>

<artifactId>hessian</artifactId>

<version>4.0.38-SF-SNAPSHOT</version>

</dependency>

配置：

<!-- 1、指定当前服务/应用的名字（同样的服务名字相同，不要和别的服务同名），可任意指定-->

<dubbo:application name="user-profile" />

<!-- 2、指定注册中心的位置，

id：注册中心id，当使用多个注册时用于区分去不同地方查找服务

Protocol : 指定协议，一般是zookeeper

Address：注册中心地址

Group：指定服务组

-->

<dubbo:registry id="processIde" protocol="zookeeper" address="10.xx.xx.xx:2181" group="dubbo-bdp-ide" />

<!-- 3、声明需要调用的远程服务的接口；生成远程服务代理

Registry：指定注册中心id，跟dubbo:registry id的一致，不同注册中心可以通过这里指定

-->

<dubbo:reference id="clusterHdfsService" interface="com.sf.bdp.ide.api.service.hdfs.ClusterHdfsService" registry="processIde"/>

[Java][Eclipse][指定Java]

修改根目录下eclipse.ini文件，加入

-vm

C:\Program Files\Java\jdk1.8.0\_152\bin\javaw.exe

注意要在第一行加入！

[Http][Get & Post][GET 和 POST 到底有什么区别？]

|  |  |
| --- | --- |
| 1. GET、POST的幂等性 2. GET、POST的参数存放位置 3. GET、POST的安全性 4. URL的Percent Encoding 5. URL的长度限制   6-7. 请求头、请求体分开的原因及优化  8. REST对4种操作的定义  9. POST和PUT的选择  10. Ajax请求 | 1. GET请求应该具有幂等性，就是反复读取不会对系统造成副作用。POST可以不具有幂等性。 2. 对于浏览器而言，一个GET请求只能有一个URL触发，所以参数只能在queryString上。而POST请求会提交表单，表单格式可以是application/x-www-form-urlencoded用来传输简单的数据（key-value）；multipart/form-data格式用于传文件。而且POST也支持在URL上传queryString参数。而对于服务器交互而言，GET和POST都可以通过URL和body传参 3. 对于安全而言，虽然GET的请求参数在URL上，而POST的请求参数在BODY里面，但这仅仅能提高一点点安全效果，通过https加密会更加安全。而且安全上要处理的问题有很多，跨域、XSS、前端钓鱼等，POST和GET在安全这件事上仅仅是个小角色 4. 在URL上支持的字符是优先的，为了传特殊字符、中文等，使用了percent encoding。它能为URL不能识别的字符转换为可以识别的字符。但它不管字符集编码（比如中文到底是用UTF8还是GBK），早期不同浏览器用的编码不一样，开发者不可控，如果要传特殊字符，前端可以通过Ajax发送，它可以指定编码形式。目前，浏览器几乎都统一使用UTF-8。 5. URL有长度限制，目前Chrome是2M 6. 请求头(header)和请求体(body)分开的原因：    1. 对于HTTP代理：       1. 支持转发规则，nginx要先解释请求头在决定怎么做（转发proxy\_pass，重定向redirect，rewrite后重新判断等）       2. 用请求头的信息记录log       3. 如果代理规则不涉及请求体，可以用zero copy转发，提高上传文件效率    2. 对于HTTP服务器：       1. 通过请求头进行ACL权限校验控制       2. 通过Content-Length、Content-Type判断是否可以处理，不能就直接拦截。无需接收全部信息，这需要内存空间，如果并发量太高，如果导致服务器爆内存       3. 如果body的数据量很大，可以使用stream API，一批批处理 7. 一次POST的请求如果超过1KB，就先发请求头，否则一并发送。 8. 对于REST的规定，GET(获取)、POST(创建)、PUT(替换)、DELET(删除) 9. POST(创建)、PUT(替换)有时可以相互替代，PUT的规定是整体替换，既然叫替换，就需要知道id，如果id存在则替换，不存在则创建。对于可以知道对象的完整数据，特别是id的对象，可以用PUT；没有id或者id自动生成的则，用POST。 10. 浏览器Ajax请求，可以包含header和body，还可以指定编码形式，同时body的格式也支持多种，除了浏览器支持的表单格式外，还支持json、csv及自定义格式等，只有前后端沟通好。但GET请求有它的便捷性，只要在浏览器地址栏上就能触发访问，方便访问和调试，而且参数都在URL上，方便保存以便于重复使用。这是POST和Ajax请求不可替代的。 |
| 1. 虽然从规则来说，GET具有幂等性，而POST不具有，但也要看具体业务，很多时候，GET有幂等性是为了数据放缓存，读取方便，但如果数据量太大、变化太频繁，也不能总放缓存。POST有时也要实现幂等性，防止用户手抖多点几下。 2. 其实GET和POST都支持header和body，比如ES的GET。但浏览器由非Ajax触发的话，GET只能由URL触发，而URL有显式、字符支持和长度等限制，涉及这些问题时可以考虑POST，否则GET更加方便 3. GET和POST的区别：规则(定义、幂等性 8、1) -> 浏览器对GET的触发限制(2)-> URL的限制(4、5) -> BODY的优势、请求头、请求体分开的好处(6) -> 补充Ajax对GET的灵活性(10) -> GET的便捷性(10)   -- GET 和 POST 到底有什么区别？ - 大宽宽的回答 - 知乎 https://www.zhihu.com/question/28586791/answer/767316172 | |

[Java][MyBatis][自动commit]

当使用MyBatis-Spring时，就由spring来管理提交事务，这时不能用SqlSession.commit()，SqlSession.rollback() 或 SqlSession.close() 方法，否则会抛出 UnsupportedOperationException 异常。不管JDBC是否设置了自动提交，SqlSession的数据方法及Spring事务之外调用的任何mapper方法，都会被自动提交。如果想手动提交，可以使用PlatformTransactionManager手工管理事务。

[Java][事务][简单汇总及使用]

1. Spring的全注解引入事务方式：

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="multiDataSource" />

</bean>

<!-- 这句话运行在beans.xml配置的类中使用@Transactional来申明事务 -->

<tx:annotation-driven transaction-manager="transactionManager" />

引入后，通过在public方法上加入@Transactional注解，就可以对该方法进行事务管理，具体作用：

当里面有数据库的操作，如果该方法报出任何RuntimeException，数据库的操作就不会被commit。默认只作用于RuntimeException，除非在注解上加入属性rollbackFor=xxxException.class

使用细节：

1. 在public方法上加入@Transactional注解才能生效，并且该类是由spring容器初始化的
2. 默认只作用于RuntimeException，除非在注解上加入属性rollbackFor=xxxException.class
3. 方法里面调用的任何子方法(不管是public还是private)报出RuntimeException，都会自动rollback
4. 如果加入rollbackFor=xxxException.class后，在代码中catch处理了，不会自动rollback

等效于：

//事务属性 PROPAGATION\_REQUIRED

methodB{

……

}

main{

metodB();

}

//等于

main{

Connection con=null;

try{

con = getConnection();

con.setAutoCommit(false);

//方法调用

methodB();

//提交事务

con.commit();

} Catch(RuntimeException ex) {

//回滚事务

con.rollback();

} finally {

//释放资源

closeCon();

}

}

1. 事务有7中传播行为，用于控制两个方法在内部调用，正确和错误时的commit和rollback关系

一般的都比较容易理解，除了要具体分清PROPAGATION\_NESTED 与PROPAGATION\_REQUIRES\_NEW的区别：

PROPAGATION\_REQUIRES\_NEW时，内层事务与外层事务就像两个独立的事务一样，一旦内层事务进行了提交后，外层事务不能对其进行回滚。两个事务互不影响。

PROPAGATION\_NESTED时，外层事务的回滚可以引起内层事务的回滚。而内层事务的异常并不会导致外层事务的回滚。

1. 事务有4种隔离级别，分别用于无隔离、防脏读、不可重复读及幻读，级别越高，隔离越强，但锁的范围越大，性能越低

参考 ：

Spring事务管理（详解+实例）

https://blog.csdn.net/trigl/article/details/50968079

[Java][Maven][exclusion][\*号不生效]

exclusion jar包时，用\*表示artifactId有时不生效，尽管在IDEA已经看不到，但用mvn 打包后，还是有该jar 包。在解决logback和slf4j-log4j冲突时遇到，在pom里面排除了slf，IDEA和dependency:tree都没看到，可是打出来的包就是有

[Java][Maven][dependencyManagement]

通过<dependencyManagement>可以指定jar的版本号，然后继承这个POM的子POM文件，如果引入了指定过的jar包，就按照dependencyManagement来获取版本，就算在子POM指定了也没用。好处：

1. 统一Jar版本，防止项目中不同模块引入了不用版本导致冲突
2. 不需要通过在parent POM中引入这些jar来统一版本，不然子模块引入了不需要的jar会很冗肿

[Java][log4j]

当遇到以下异常：

nested exception is java.lang.NoClassDefFoundError: Could not initialize class org.apache.log4j.Log4jLoggerFactory

通常是因为log4j包冲突了，之前遇到logback和slf4j-log4j冲突，把slf4j-log4j去掉即可。

[公司][报销话费总结]

1. 两到三个月报销一次
2. 用电脑网页登陆网上营业厅，查询电子发票，选在下载到手机，会提示扫码保存到微信卡包
3. 手机丰声财务资助上传发票，在微信卡包里面的发票可以一次上传多张
4. 报销时内容填写”曹伟超报销xxxx年x、x月移动话费”
5. 通过在报销页面添加多个详情一次报销几个月份的

[Java][编码]

在Windows环境下，JVM默认编码为GBK。

Linux环境下默认编码为UTF-8。

运行时指定编码：

java -Dfile.encoding=UTF8 ...

(在java命令后面传入)

[Mavan][打包包含依赖]

在pom文件添加以下插件：

<!-- 打包时添加依赖 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-assembly-plugin</artifactId>

<version>3.0.0</version>

<configuration>

<descriptorRefs>

<descriptorRef>jar-with-dependencies</descriptorRef>

</descriptorRefs>

</configuration>

<executions>

<execution>

<id>make-assembly</id> <!-- this is used for inheritance merges -->

<phase>package</phase> <!-- 指定在打包节点执行jar包合并操作 -->

<goals>

<goal>single</goal>

</goals>

</execution>

</executions>

</plugin>

[Java][线程池][线程增加机制]

线程池里面有corePoolSize、maximumPoolSize、任务队列等三个概念。

它们的关系**并不是**说当corePoolSize满了，启动新的线程，直到maximumPoolSize也满了，任务就提交到任务队列中。

而是，当corePoolSize满了，任务就提交到任务队列中，这是也仅仅只有corePoolSize个线程在跑，直到任务队列也满了，才会启动新的线程，直到maximumPoolSize也满了，再根据饱和策略决定拒绝还是等待。

[Java][线程池][corePool的存活时间]

默认情况下，线程池会保存corePoolSize个线程，并且它们不是因为keepAliveTime过了而被销毁，它们会一直存活。

除非设置了allowCoreThreadTimeOut(true); 此时corePool的线程也会在空闲并过了keepAliveTime后被销毁，这是keepAliveTime必须大于0。

为了减少多余线程的产生，充分利用资源，可以把corePoolSize = maximumPoolSize; allowCoreThreadTimeOut(true); keepAliveTime足够小;任务队列无限大。这时，应用在初始化corePoolSize后，保留使用中的线程数，并且在需要更多线程时，新建线程。

[Maven][resources]

1. 通过在POM中的<build><resources><resource>指定resources文件夹后，打包时会从指定目录拷贝文件
2. 如果不指定，则默认从src/main/resources中拷贝，指定后是不会再从src/main/resources拷贝了。
3. <includes>指定需要的文件，不指定则默认全部需要
4. <excludes>排除不需要的文件
5. <filtering>true，把拷贝文件**里面的**占位符内容，都填充成POM中设定的值，可以在<profile><properties>中设置，这样可以给不同的环境配置不同的参数。默认是false

[MySQL][定位my.cnf]

1. locate my.cnf //列出系统中所有my.cnf文件
2. ps aux|grep mysql|grep 'my.cnf' //查找当前在使用的cnf文件
3. 如果没有指定，通过以下命令找到MySQL配置文件的优先级文件夹：

mysql --help|grep 'my.cnf'

1. 如果这些目录下都没有my.cnf，会使用默认配置，我们还可以手动创建一个配置文件放到上方的路径当中

[MySQL][Specified key was too long; max key length is 767 bytes]

这个错误发生在建表时，定义的字段过长导致。通过以下3步解决：

1. set global innodb\_large\_prefix=on;
2. set global innodb\_file\_format=Barracuda;
3. 建表语句最后加上ROW\_FORMAT=DYNAMIC，e.g.

CREATE TABLE `openapi\_project\_info\_niku1` (

`id` bigint(20) NOT NULL AUTO\_INCREMENT COMMENT '主键ID',

`project\_name` varchar(100) DEFAULT NULL COMMENT '项目名称',

PRIMARY KEY (`id`),

UNIQUE KEY `search01` (`project\_name`),

) **ROW\_FORMAT=DYNAMIC** ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='Open API项目信息表';

[SQL][limit]

limit关键字是执行在查询的最后，比如 select count(1) from xxx limit 100, 假设表有10000行，结果就是10000，而不是100。它是先执行完select count(1) from xxx，再进行limit的。

[Java][变长度参数]

1. 变长度参数只能放在参数列表的最后
2. 可以传任意个数的入参，甚至是**不传**

[Java][SpringMVC][代码规范][分层|定位]

1. Controller层
   1. 用于判断并调用不同的Service层服务
   2. 不做参数校验，因为不同的service可能对参数的要求不一样
2. Service层
   1. 用于实现业务相关的代码逻辑
   2. 参数校验
   3. 调用Manager层的方法
   4. 调用Mapper访问数据
   5. 不建议Service层调用Service层，减少耦合性
   6. vo、业务相关的pojo
3. Manager层
   1. 封装通用的方法提供给Service层调用，可以是第三方组件、通用方法
   2. 这些类都会初始化，可以注入其他spring bean
4. Mapper层
   1. 数据库相关的访问及实现
   2. po，数据库访问时的入参出参bean定义
5. Common层
   1. 常量
   2. Util，静态方法类
   3. 其他bean的定义，如excption、annotation以及一些自定义的业务无关的bean等

[前后端交互][接口定义]

一个接口是否能通用，需要考虑以下三个方面：

1. 入参
2. 出参
3. 功能、业务

如果这三方面有任意一个不同，都要新建额外的接口

[Hive][RLIKE|REGEXP]

Hive的的RLIKE支持正则表达式，且只要字段里面包含正则表达式的内容就能匹配成功.

e.g. where interest RLIKE '爱吃零食|生鲜客'

[Hive][CONCAT\_WS]

多个字段或数组合并成一个字段

eg. CONCAT\_WS(‘,’, ‘aaa’,’bbb’,’ccc’) , 第一个参数是结果拼接符， ‘aaa,bbb,ccc’ ； CONCAT\_WS(array)，如果不传结果拼接符，则直接拼接，可以拼接数组

[Java][HBase][HBaseTemplate]

Spring的HBaseTempl有个问题，就是每次查数据都要连接ZK，查询完关闭，在访问量大的情况下，这样会大大增加ZK的压力，影响业务的响应时间。建议使用原生的hbase-client来连接即可。连接要点：

1. 在初始化时建立连接，不关闭
2. 读完数据后关闭table

样例如下：

static {

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.property.clientPort", config.getProperty("hbase.zk.port"));

configuration.set("hbase.zookeeper.quorum", config.getProperty("hbase.zk.quorum"));

try {

conn = ConnectionFactory.createConnection(configuration);

} catch (IOException e) {

throw new RuntimeException(“Fail to create HBase connection!”);

}

}

public static Result getAsMap(String tableName, String rowKey, String family) {

Result result = null;

Table table = null;

try {

table = conn.getTable(TableName.valueOf(tableName));

Get get = new Get(rowKey.getBytes());

get.addFamily(family.getBytes());

result = table.get(get);

} catch (Exception e) {

logger.error("Fail while getting data in HBase", e);

} finally {

try {

if(null != table) table.close();

} catch (IOException e) {

logger.error("IOException while close table", e);

}

}

return result;

}

[Java][监控][Arthas]

一个Java程序监控工具，阿里的，开源

启动命令：

java -jar arthas-boot.jar

以下命令可以监控方法内一级子方法的调用时间，'#cost>100'指只返回超过100ms的，可不传。

trace com.xxx.classname xxxfunctionname '#cost>100'

**[Beyond Compare4][注册码]**

— BEGIN LICENSE KEY —

H1bJTd2SauPv5Garuaq0Ig43uqq5NJOEw94wxdZTpU-pFB9GmyPk677gJ

vC1Ro6sbAvKR4pVwtxdCfuoZDb6hJ5bVQKqlfihJfSYZt-xVrVU27+0Ja

hFbqTmYskatMTgPyjvv99CF2Te8ec+Ys2SPxyZAF0YwOCNOWmsyqN5y9t

q2Kw2pjoiDs5gIH-uw5U49JzOB6otS7kThBJE-H9A76u4uUvR8DKb+VcB

rWu5qSJGEnbsXNfJdq5L2D8QgRdV-sXHp2A-7j1X2n4WIISvU1V9koIyS

NisHFBTcWJS0sC5BTFwrtfLEE9lEwz2bxHQpWJiu12ZeKpi+7oUSqebX+

— END LICENSE KEY -----

或者：

w4G-in5u3SH75RoB3VZIX8htiZgw4ELilwvPcHAIQWfwfXv5n0IHDp5hv

1BM3+H1XygMtiE0-JBgacjE9tz33sIh542EmsGs1yg638UxVfmWqNLqu-

Zw91XxNEiZF7DC7-iV1XbSfsgxI8Tvqr-ZMTxlGCJU+2YLveAc-YXs8ci

RTtssts7leEbJ979H5v+G0sw-FwP9bjvE4GCJ8oj+jtlp7wFmpVdzovEh

v5Vg3dMqhqTiQHKfmHjYbb0o5OUxq0jOWxg5NKim9dhCVF+avO6mDeRNc

OYpl7BatIcd6tsiwdhHKRnyGshyVEjSgRCRY11IgyvdRPnbW8UOVULuTE

如果提示 这个授权密钥已被吊销 ，删除 C:\Users\Administrator\AppData\Roaming\Scooter Software\Beyond Compare 4 目录下的文件即可

**[Java][Redis][JedisPool资源池优化]**

参考：

https://www.alibabacloud.com/help/zh/doc-detail/98726.htm#section-m2c-5kr-zfb





为了方便使用，Jedis提供了JedisPoolConfig，它继承了GenericObjectPoolConfig在空闲检测上的一些设置。

public class JedisPoolConfig extends GenericObjectPoolConfig {

public JedisPoolConfig() {

// defaults to make your life with connection pool easier :)

setTestWhileIdle(true);

//

setMinEvictableIdleTimeMillis(60000);

//

setTimeBetweenEvictionRunsMillis(30000);

setNumTestsPerEvictionRun(-1);

}

}

## **关键参数设置建议**

**maxTotal**（最大连接数）

想合理设置**maxTotal**（最大连接数）需要考虑的因素较多，如：

* 业务希望的Redis并发量；
* 客户端执行命令时间；
* Redis资源，例如nodes （如应用个数等） \* maxTotal不能超过Redis的最大连接数；
* 资源开销，例如虽然希望控制空闲连接，但又不希望因为连接池中频繁地释放和创建连接造成不必要的开销。

假设一次命令时间，即borrow|return resource加上Jedis执行命令 （含网络耗时）的平均耗时约为1ms，一个连接的QPS大约是1000，业务期望的QPS是50000，那么理论上需要的资源池大小是50000 / 1000 = 50。

但事实上这只是个理论值，除此之外还要预留一些资源，所以**maxTotal**可以比理论值大一些。这个值不是越大越好，一方面连接太多会占用客户端和服务端资源，另一方面对于Redis这种高QPS的服务器，如果出现大命令的阻塞，即使设置再大的资源池也无济于事。

**maxIdle与minIdle**

**maxIdle**实际上才是业务需要的最大连接数，**maxTotal** 是为了给出余量，所以 **maxIdle** 不要设置得过小，否则会有new Jedis（新连接）开销，而**minIdle**是为了控制空闲资源检测。

连接池的最佳性能是**maxTotal**=**maxIdle**，这样就避免了连接池伸缩带来的性能干扰。但如果并发量不大或者**maxTotal**设置过高，则会导致不必要的连接资源浪费。

您可以根据实际总QPS和调用Redis的客户端规模整体评估每个节点所使用的连接池大小。

****使用监控获取合理值****

在实际环境中，比较可靠的方法是通过监控来尝试获取参数的最佳值。可以考虑通过JMX等方式实现监控，从而找到合理值。

****预热JedisPool，**可以减少**new Jedis（新连接）开销

**Note：参数testOnBorrow的配置是否为true,如果为true每次从连接池获取一个连接,都会判断这个连接是否可用,目前使用的jedis版本是2.6.2,判断某个分片集群的连接是否可用会遍历所有分片(这个集群是48个分片)发心跳判断连接是否可用,本来根据指定key调用只需要调用某个片,因此导致调用时间上升,redis连接数飙升。**

**建议：testOnBorrow改为false。**

**[Java][default访问修饰符]**

当不指定访问修饰符的时候，默认为default，此时只允许同一个类、同一个包之间的访问。

1. 当default修饰的是类时，只有同一个包里面的类才能继承，子类也能继承default父类的default变量，因为它们一定在同一个包内，否则也防问不了这个类，谈何继承。
2. 当子类和public父类在不同的包，子类不能继承父类的default变量
3. 可以继承变量要满足两个条件：
   1. 类定义中继承了父类
   2. 父类中某变量的访问修饰符，它的访问范围允许子类访问

**[Java][内部类与非内部类][初始化][区别]**

内部类的初始化方法：

创建静态内部类对象的一般形式为： 外部类类名.内部类类名 xxx = new 外部类类名.内部类类名()

创建成员内部类对象的一般形式为： 外部类类名.内部类类名 xxx = new 外部类对象名().new 内部类类名()

即使两个类A、B在同一个文件下定义，如果Class A 的{}没有保住Class B，那个Class B就不是 Class A的内部类。此时，初始化Class B， 只要 new B()就好

**[Windows][命令行][端口占用][关闭进程]**

查看端口占用命令：

netstat -ano | findstr "8080"

关闭进程：

taskkill /F /T /IM sqlservr.exe

taskkill /F /pid “4984”

备注：

/IM 指定要终止的进程的映像名称。通配符 '\*'可用来指定所有任务或映像名称。

/PID 指定要终止的进程的 PID。使用 TaskList 取得 PID。

/T 终止指定的进程和由它启用的子进程。

/F 指定强制终止进程。

**[Java][toString(), new String()的区别]**

toString()是执行该对象的toString方法，如果没有实现，则打印内存位置。

new String(xx)是把xx作为参数来初始化一个String对象

e.g.:

byte[] bytes = {97,98,99};

bytes.toString() ;//结果：[B@2265792c

new String(bytes); //结果：abc

bytes + “def”; //结果：[B@2265792cdef, 因为“+”拼接符与String拼接，默认调用toString()方法

new String(bytes) + “def”; //结果：abcdef

**[ElasticSearch][别名]**

一个别名可以对应多个索引，一个索引也可以对应多个别名 (多对多关系)

创建别名：

POST \_aliases

{

"actions" : [{"add" : {"index" : "test\_index\_name" , "alias" : "test\_alias\_name"}}]

}

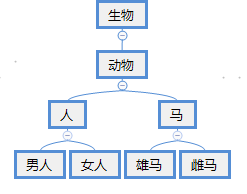
查询别名对应的索引：

GET test\_alias\_name/\_search

查询索引对应的别名：

GET test\_index\_name/\_search

**[Java][泛型][extend][super]**



1. ~~extend，<? extend 动物>, 则表示类型是动物的子类，如人、马。extend 能取不能存，那是因为，取出来的值都是动物的子类，可以赋值给动物。而如果要存，不能确定存的是人还是马还是其它类，而且人、马不能转换成动物，会失去特征，存进去就没了特征，意义不大。因此不能存。~~
2. ~~super，<? super 人>, 则表示类型是人的父类，就是动物。super能存不能取，存进去的时候，因为定义的时候，类型是人的父类，那么只要是~~**~~人或者他的子类，都可以存进去~~**~~，就像 void function(动物 ani)一样，这个方法可以填充ani及它的子类。而如果要取，赋值给什么类型呢？除了能确定可以赋值给Object以外，人的父类是是什么并不确定，可以是动物，也可以是生物等等，而赋值给Object一般没啥意义。因此不能取。~~
3. extend 和 super后面可以搭配类和接口
4. 把变量传入方法中，可以传入入参类型及其子类，那是因为，在方法里面，我们仅仅会调用入参类型的方法，而它的子类也有对应的方法实现
5. 把结果返回，只能返回结果出参类型及其父类，是为了防止我们调用了一个子类独有而父类并不存在的方法。
6. 结合4、5点，<? extend 动物>表示动物的子类。在get()方法中，我们可以赋值给动物。但在set(? extend 动物)中，我们仅仅知道它是动物的子类，但不同子类有自己独特的方法，如果传入了马，但调用了人才会的方法”上学”，这就会报错。所以extend能取不能存。
7. 结合4、5点，<? super 人>, 则表示类型是人的父类。在get()方法中，我们赋值给谁？虽然是人的父类，但除了知道肯定是Object的子类以为，我们还是不知道它是什么类型，赋值给Object，又失去了对象的特征；如果赋值给人，调用了一个人才有的方法“上学”，而实际对象是动物，动物不会上学，那就会报错了。但如果在set(? super 人)中，这个方法里面也只会调用人实现过的方法，这些方法在”人”、“男人”和“女人”中都实现了，那没问题。所以super能存不能取。

**[Java][Socket][绑定端口]**

Socket socket=new Socket();

socket.bind(new InetSocketAddress(12316)); //绑定本地端口

socket.connect(new InetSocketAddress("127.0.0.1", 12315)); //指定服务器地址和端口

**[Java][Socket[查询端口]**

查询本机：

socket.getLocalAddress().getHostName()

socket.getLocalPort()

查询指定服务器：

socket.getInetAddress().getHostName()

socket.getPort()

**[Java][Socket与IO Stream实验总结]**

1. socket.accept()会阻塞等待，使用多线程处理收发
2. 对于不同的socket，接收端会区分，不会串一起。除非只使用一个socket发送不同的业务，建议还是用不同的socket区分不同业务数据
3. 虽然socket可以绑定端口，不过一般也不需要，反正每次开启socket client都会自动使用不同的端口，开多个也不会冲突
4. 读字节流的时候，因为拆包和黏包的原因，不建议通过流available()获取流长度来初始化接收的byte[], 最好是发送端把长度告诉接收端
5. 因为不同的socket不会串一起，我们每次发送数据时，可以通过多个write()方法，把内容类型、字节长度、具体内容等信息发送给接收端，接收端通过循环read()来获取数据
6. 可以使用DataInputStream和DataOutputStream封装IO流，有更多便捷的流处理方法

具体代码参考如下：



[Java][从ReentrantLock的实现看AQS的原理及应用]

|  |  |
| --- | --- |
| 1 什么是AQS  2 什么是ReentrantLock  3 ReentrantLock与Synchronized块对比  4 AQS核心思想  5 AQS的同步状态  6 AQS的Node  7 自定义同步器要实现的方法  8 ReentrantLock的加锁、解锁流程 | 1. AQS全称AbstractQueuedSynchronizer，它是一种提供了原子式管理同步状态、阻塞和唤醒线程功能以及队列模型的简单框架。 2. ReentrantLock是可重入锁，一个线程可以对一个临界资源重复加锁。底层就是由AQS来实现的 3. ReentrantLock相比Synchronized块：      1. AQS核心思想，如果被请求的共享资源空闲，那就将请求资源的线程设置为有效的工作线程，将共享资源设置为锁定状态；如果共享资源被占用，就需要一定的阻塞等待机制来保证锁分配，这个机制主要用的队列是CLH变体的虚拟双向队列（FIFO） 2. AQS使用一个Volatile的int类型的成员变量state来表示同步状态，通过CAS完成对State值的修改。 3. AQS中最基本的数据结构——Node，Node即为上面CLH变体队列中的节点。      1. 自定义同步器需要实现以下方法：   一般来说，自定义同步器要么是独占方式，要么是共享方式，它们也只需实现tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared中的一种即可。     1. ReentrantLock的加锁、解锁流程   ReentrantLock |
|  | |

**[MYSQL][分页][MySQL分页数据错乱重复]**

MYSQL通过limit来分页时，我们都会添加一个排序的字段，这里有个问题，如果排序的字段在几个记录中的值是一致的，那么分页出来的结果排序就可能在不同limit设置中不一致，导致分页的结果重复或丢失。可以通过多加一些字段，保证排序字段结果的唯一性。

**[Java][MyBatis][生成工具][按照数据库定义生成字段名]**

mybatis generator插件中，如果 mysql数据表中的字段是用下划线划分的(个人一般都是喜欢这么创建表的字段，如：company\_name），那么生成的Vo中会自动对应为companyName；但是有时候表不是你设计的，全是这种格式的carBrandName或者是carbrandname，在生成对应的Vo时会是这样：carbrandname

解决：

在generatorConfig.xml中下的<table>里面添加 <property name="useActualColumnNames" value="true"/>

这个值原来默认是FALSE的，表示针对字段的下划线转换为驼峰写法，而为TRUE

**[Java][CAS]**

当调用admin/login时，会被过滤器CasTicketValidationFilter先处理，先获取之前的网址，登录成功后跳转回该网址。不会进入LoginController。

**[前端][302转发]**

Ajax/异步请求不能相应后端的302重定向，可以通过location.hred 、A标签等来请求后端302

**[Java][SpringMVC][初始页面]**

在SpringMVC中，一般有个Controller的RequstMapping(“/”)，用于处理不带指定URL的请求。

**[面试模式]**

1. 高并发系统的经验
2. 大数据处理的经验
3. 比价好的系统优化经验
4. Java知识

**[ElasticSearch][清空整个索引]**

GET kadm\_user2/**\_delete\_by\_query**

{

"query": {"match\_all": {}}

}

这个方法可以把结果集清空，当结果集是整表时，就可以清空全部

**[SFOPEN][创建新项目][1.31]**

1. 使用Eclipse新建项目
2. 选择maven项目
3. Archetype选择：com.sf sfopen-archetype 1.4.8
4. 在数据库中通过select UUID();生成并填写jwt-secret
5. 数据库准备脚本可以参考：<http://osfp.sf-express.com/index.php/44&page_id=750>

[Java][Hessian][使用总结]

1. Hessian配置
   1. 服务端
      1. web.xml加入hessian配置
      2. hessian\*.xml配置对外服务
         1. 提供服务的service参数使用Spring管理的bean，如果找不到，可以加入component-scan语句
      3. 提供API包
         1. API包里面包含服务接口、dto对象
         2. dto对象要实现Serializable接口
         3. dto对象主要指service用到的自定义入参和出参对象
         4. 服务返回值如无特殊情况，用Object，这样不需在API包定义一个dto
         5. 服务端和客户端的出入参数定义必须一致，就算一边是具体类，另一边是Object也会解析失败
         6. 一般上传到Nexus
            1. pom中配置<distributionManagement>
   2. 客户端
      1. hessian\*.xml配置接入服务
      2. import hessian.xml 到 Spring
      3. maven import 服务端API包

[Java][shiro][URL拦截]

Shiro的项目中，可能有配置shiro-config.xml拦截访问路径，添加controller时要注意把新的跟路径加上去