# 一．Java基础

## 1.0 io与nio

### 1.01 io分类与基础使用

主要分为字节流，字符流，对象流等；字节流是以继承inputStream,outputStream两抽象类的子类如：FileInputStream,BufferedInputStream,FileOutputStream,BufferedOutputStream等；

字符流是以继承Reader,Writer两抽象类的子类如：FileReader,BufferedReader, InputStreamReader

FileWriter,BufferedWriter,OutputStreamWriter等。

使用场景：1.io流是万能流，能传输任意文件； 2.字符流的使用仅仅局限于文本文件，且字符流在处理文件的时候对编码方式的要求严格

Io流的使用注意事项：1.使用流要注意使用完释放资源即关闭连接；2.流的关闭顺序也很重要，理论上是先打开的先关闭，a依赖b，就先关闭a; 3.使用输出流的时候文件不存在，程序会自己创建文件，但不会自己创建文件夹，需要手动去创建文件夹； 4.io不是并发的流，是阻塞的流，某一时刻要么执行读操作要么执行写操作！

### 1.02 nio非阻塞流

概述：面向缓冲的非阻塞流。

核心：通道，缓冲，选择器，非阻塞io

通道类型：FileChannel,DatagramChannel(udp读写网络数据),SocketChannel（tcp）,ServerSocketChannel

缓冲：CharBuffer DoubleBuffer IntBuffer LongBuffer ByteBuffer ShortBuffer FloatBuffer

选择器：适用于单个线程处理多个通道的监听

## 1.1 线程

### 1.1.1创建线程的方式

1. 直接继承Thread（缺点：单继承不易扩展）
2. 实现Runable接口（有点：可以实现多个线程共享资源）常见！！

### 1.1.2线程状态

1.创建：new后的线程

2.就绪：start()的线程，随时可以被cpu调度执行

3.运行：线程获得cpu执行权，只能由就绪状态到运行状态

4.阻塞：某种原因线程放弃cpu执行权，阻塞的情况有三种

a.等待阻塞，调用wait()方法，等待唤醒

b.同步阻塞, 线程在获取synchronized同步锁失败

c.其他阻塞，sleep(),join()等

5.死亡：run完毕或被Kill掉

### 1.1.3 多线程如何避免死锁

1.死锁例子：

**public void** mthod1(){  
 **synchronized** (String.**class**){  
 System.***out***.println(**"一线程获得string的锁，并锁住"**);  
 **synchronized** (Integer.**class**){  
 System.***out***.println(**"线程获得Integer的锁"**);  
 }  
 }  
 System.***out***.println(**"该线程已经执行完程序，没有锁住！"**);  
}  
**public void** method2(){  
 **synchronized** (Integer.**class**){  
 System.***out***.println(**"一线程获得Integer的锁，并锁住"**);  
 **synchronized** (String.**class**){  
 System.***out***.println(**"线程获得Stringd的锁"**);  
 }  
 }  
}

1. 解决死锁：不让线程之间相互占据所需要的锁对象

## 1.2线程通信

1. **synchronized同步，这种方式，本质上就是“共享内存”式的通信。多个线程需要访问同一个共享变量，谁拿到了锁（获得了访问权限），谁就可以执行。**
2. **wait/notify机制**

### 1.2.1 volatile synchronized Lock ReentrantLock

1. volatile和synchronized的区别
2. 锁提供了两种主要特性：互斥（一线程一次只能持有一个特定的锁）和可见性（锁释放之前对共享数据的操作是可见的）
3. 都是为了保证多线程读取数据的一致性
4. Volatile本质是告诉jvm当前变量的值不确定，需要从主存中读取，synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞
5. Volatile仅使用在变量级别，synchronized可以使用在变量，方法
6. Volatile不会造成线程的阻塞，而synchronized可能会造成线程的阻塞
7. Volatile不具备原子性，不适合在对该变量的操作依赖于自己如n++

### 1.2.2 可重入锁 ReentrantLock

*ReentrantLock逼格满满，完全可以替代Synchronized，可中断响应、锁申请等待限时、公平锁*

1. 乐观锁：每次访问数据线程都是运行状态，提高数据吞吐量，实现方式是CAS
2. 悲观锁 ：每次访问数据线程都是处于阻塞状态，提高了安全性。但有缺点：1.可能会导致优先级高的线程等待优先级低的线程导致优先级倒序；2.加锁，放锁次数过多，引起性能问题
3. 阻塞锁：让线程进入阻塞状态等待，当获得相应的信号（唤醒，时间）时，才可以进入线程的准备就绪状态

## 1.3 集合

### 1.3.1 Linkedhashmap与hashmap的区别

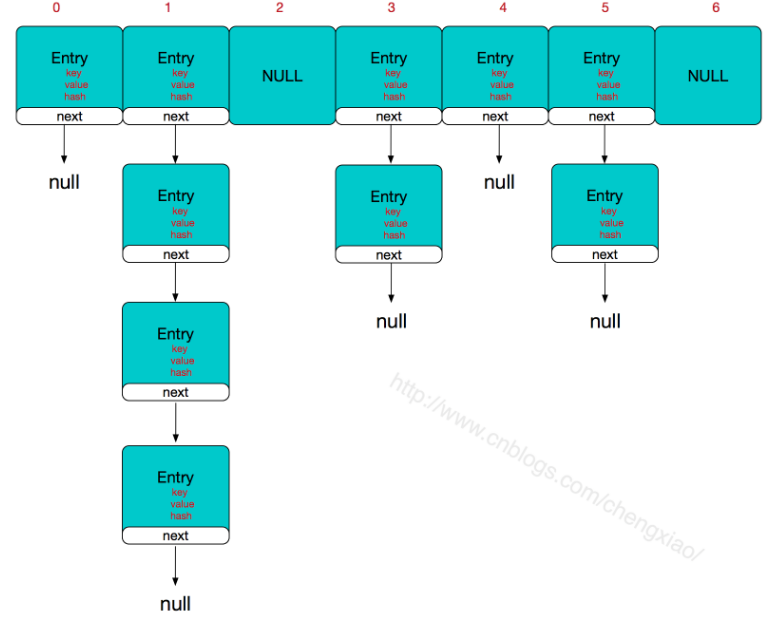
>1.hashMap的结构与原理：

a.hashMap是由数组（主体）+单向链表，数组中存放着entity

b.entity中存放着key-value，hash和指向下一个实体的指针

c.元素是由hash函数计算存储地址放到数组中的

d．哈希冲突：两个同元素通过hash函数得到的实际存储地址一样，hashMAP采用链地址法解决冲突。



>LinkedHashMap是在hashMap的基础上增加了顺序（数组+双向链表）

### 1.3.2 二叉树

有道云笔记链接地址：

http://note.youdao.com/noteshare?id=86d7a1bce6dfe0dc61218a2cc58d82dc&sub=63ECF61753B74B42A6BF2A82A258A8D5

## 1.4 面向对象的几大特征

>1.抽象：忽略一个主题中与当前目标无关的东西,专注的注意与当前目标有关的方面

>2.封装：把过程和数据包围起来,对数据的访问只能通过特定的界面

>3.继承：为了重用父类代码,同时为实现多态性作准备

>4.多态：是指允许不同类的对象对同一消息做出响应,如子类对父类的重载，同一类中方法的重写

## 1.5 反射（框架设计的灵魂）

>1.简述：在运行状态中，对任意一个类中，都能获得这个类的属性和方法，通俗点讲，反射就是把java类中的各种成分映射成一个个的Java对象。通过反射，该类对我们来说是完全透明的，想要获取任何东西都可以。

>2.反射先得获得类的class（字节码），获得字节码的方法有三种，Class clazz1 = Class.forName("全限定类名");Class clazz2  = Person.class; Class clazz3 = p.getClass();

>3.jdk代理，cglib代理就用到了反射

## 1.6 枚举

>1.什么是枚举？枚举是jdk5以后得新的类型

>2.枚举的格式？public enum Color{RED(“red”,1),BLUE(“blue”,2),YELLOW(“yellow”,3);public String color;public int index;setter,getter…}

>3.枚举的用途？a.可以用在controller里状态码

## 1.7 设计模式

>创建型模式 5种:工厂方法模式，抽象工厂模式，单例模式，建造者模式，原型模式

>结构型模式7种：适配器模式，装饰器模式，代理模式，外观模式，桥接模式，组合模式，享元模式

>行为型模式11种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式

## 1.8 代理

## 1.9泛型

# 二．javaWeb专题

## 2.1 jdbc dbutil c3p0

## 2.2 过滤器

## 2.3 xml

## 2.4 tomcat

## 2.5 json

## 2.6 jetty

# 三．HTTP TCP UPD 专题

## 3.1 HTTP

### 3.1.1 http请求报文&http请求响应头

>request:请求行(get/url HTTP1.1)，请求头（headrs如User-Agent：xx Accept：xx），消息主体(post提交的参数)

>http响应（http报文头）:状态行（HTTP-version Status-code ok?），消息报头(Date:,Content-Type:,Content-Length:)，响应正文

### http请求方法

>get,post,put,head

### 3.1.3 http请求过程

>1.建立tcp连接🡪🡪2.浏览器向服务器发送请求命令（GET /URL HTTP1.0）🡪🡪3.浏览器发送请求头信息🡪🡪4.服务器应答🡪🡪5.服务器向浏览器发送应答消息报头🡪🡪6.服务器发送响应正文🡪🡪服务器关闭tcp连接

### 3.1.4 GET post put区别

>get 将参数写在url后面，有大小限制，不安全

>post 将参数封装在请求数据中，以Key-value的形式，数据大小没有限制

### 3.1.5 http状态码

>1xx:只是信息—表示已经接收，继续处理

>2xx：成功—表示已经接收，处理

>3xx：重定向—要完成请求必须进一步操作

>4xx：客户端错误—请求资源不存在

>5xx:服务端错误—服务器未能实现合法的请求

### 3.1.6长连接 短连接，http协议是无状态

>短连接：http1.0默认采取短连接，每次访问一个css,jps,html等资源都重新建立连接-传输数据-关闭连接。耗资源

>长连接：http1.1默认采取长连接，Connection:keep-alive,在每次服务器与客户端建立连接后不主动去关闭连接。

>http协议无状态：是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。无状态连接严重阻碍了应用程序的实现，因此想要保持连接状态的技术应运而生，Cookie和session

### 3.1.7转发和重定向

>转发：是服务器的行为，浏览器的地址信息不变

>重定向:是客户端的行为，浏览器的地址发生变化

## 3.2 TCP

>

### 3.2.1 TCP头部

### 3.2.2 tcp三次握手

### 3.2.3 tcp如何保证可靠性传输

### 3.2.4 ip报文 地址类别 私有地址

### 3.2.5网络攻击 dns欺骗 sql注入 ddos攻击

### 3.2.6 socket编程

### 3.2.7 https ssl

## 3.3 UDP

# 四．数据库专题

## 4.1 sql语句

4.1.1 sql主要分为两大类：dml(数据操作语句：增删查改等),ddl(增加删除表，定义索引等)

4.1.2 简单sql语句（太简单，只写一点）：

a>去重 select distinct cols from tab

b> where/ and /or /order by /insert /delete /update /like /betwwen /in 不总结

c>constraints约束：

1.NOT NULL约束强制列非空

2.UNIQUE约束唯一表示数据库中的每条记录

3.PRIMARY KEY 约束唯一表示数据库中的每条记录，每个表只能有一个主键

4. foreign key 一个表中的外键指向另一个表中的主键 constraint fk perOrders foregin key (id\_p) references person(id\_p)

5.CHECK check约束用于限制列中值的范围 CHECK(id\_p>10)

6.DEFAULT

4.1.3 高级sql

a>group by having

分组查询并带上过滤条件，#--分组查询--#

SELECT \* FROM `work` GROUP BY dept\_id HAVING salary>=800

b>子查询

1.1->相关子查询（子查询查询依赖外部查询）：

一条sql查询的结果作为另一个sql查询的条件在再进行查询。

SELECT \* FROM `work` a WHERE salary=(SELECT MAX(salary) FROM `work` b WHERE a.dept\_id=b.dept\_id);

1.2->嵌套子查询（子查询依赖外部查询）：

1.3->语句执行顺序：

标准的sql解析顺序：

(1) FROM 子句 组装来自不同数据源的数据

(2) WHERE 子句 基于指定的条件对记录进行筛选

(3) GROUP BY 子句 将数据划分为多个分组

(4) 使用聚合函数进行计算

(5) 使用HAVING子句筛选分组

(6) 计算所有的表达式

(7) 使用ORDER BY对结果集进行排序

不包含子查询如->select \* from t where col=xx 这样一条sql语句，分析器先找到select，然后跳到from将t表导入内存，并通过指针p1找到第一条记录，接着找到where关键字计算它的表达式，如果条件为真就将这条记录装进虚拟表，P1指针一直指向其他记录，直到检索完整个表。

相关子查询如->SELECT \* FROM `work` a WHERE salary=(SELECT MAX(salary) FROM `work` b WHERE a.dept\_id=b.dept\_id);会依次一行行地拿外部查询的结果给子查询使用，子查询查到的记录再返回给外部显示

嵌套查询如->SELECT a.\* FROM `work` a,(SELECT dept\_id,MAX(salary) as salary FROM `work` GROUP BY dept\_id) t

WHERE t.dept\_id=a.dept\_id AND t.salary=a.salary;会先把子查询一次性检索查询完毕，把子查询的结果返回给外部查询，再进行外部查询。

1.4->mysql查看执行计划explain:

a.使用explain关键字可以模拟优化器执行sql查询语句，从而知道mysql是如何处理你的sql，分析你的查询语句或是表结构的性能瓶颈

b.其中重要的字段id,type,key,rows,Extra,select\_type……

id=>select查询的序列号，包含一组数字，表示查询中执行select子句或操作表的顺序。Id相同执行顺序由上至下，Id越大优先级越高

select\_type=>查询的类型，主要用于区分普通查询，联合查询，子查询等复杂的查询。SIMPLE:简单的select查询；PRIMARY:查询中包含任何复杂的子部分，最外层查询被标记为PRIMARY;SUBQUERY:在select或where列表中包含子查询；DERIVED:在from列表中包含的子查询被标记为DERIVED(衍生)；UNION:若第二个select出现在union之后，则被标记为union;

type=>访问类型，sql查询优化的重要指标，扫描表的方式。

Possible\_keys=>查询涉及到的字段上存在的索引

Key=>实际使用的索引

Ref=>显示索引的哪一列被使用

Rows:大致估算出找到需要的记录需要读取的行数

Extra=>不适合在其他字段中显示，但是十分重要的额外信息

c>连接查询

1. 内连接inner join
2. 左连接left join
3. 右连接right join
4. 全连接full join

e>事务处理

1. 支持事务的数据库引擎。
2. 事务的主要命令begin,commit,rollback,savepoint..
3. 隔离级别：READ UNCOMMITED READ COMMITED REPEATABLE READ SERIALIZABLE

## 4.2 索引

4.2.1>作用：在不读取整张表的情况下，索引可以更快地查询数据

4.2.2>创建：1直接创建索引.create INDEX i\_k on tab(col) 2.间接创建索引：在定义表时

4.2.3>索引的种类：

## 4.3 分库分表

### 4.3.1 概念与垂直水平分表

1.切分原因：单表的数据量达到1000w或100G以后，由于查询维度较多，即使添加从库，优化索引，性能都会严重下降，为了减轻数据库的负担，需要对表进行切分。

2.切分分类：垂直切分和水平切分

>垂直切分有垂直分库和垂直分表，根据业务耦合性，将关联度低的不同表存储在不同的数据库。做法与大系统拆分为多个小系统类似，按业务分类进行独立划分；基于数据库中的"列"进行，某个表字段较多，可以新建一张扩展表，将不经常用或字段长度较大的字段拆分出去到扩展表中，即垂直分表。

>水平切分，水平切分分为库内分表和分库分表，是根据表内数据内在的逻辑关系，将同一个表按不同的条件分散到多个数据库或多个表中，每个表中只包含一部分数据，从而使得单个表的数据量变小，达到分布式的效果

3.切分的优缺点：

>优点：垂直切分=>1.解决业务系统层面的耦合，业务清晰 2. 与微服务的治理类似，也能对不同业务的数据进行分级管理、维护、监控、扩展等 3. 高并发场景下，垂直切分一定程度的提升IO、数据库连接数、单机硬件资源的瓶颈

水平切分=>1.不存在单库数据量过大、高并发的性能瓶颈，提升系统稳定性和负载能力 2. 应用端改造较小，不需要拆分业务模块

>缺点：垂直切分=>1.部分表无法join，只能通过接口聚合方式解决，提升了开发的复杂度2.分布式事务处理复杂3.依然存在单表数据量过大的问题（需要水平切分）

水平切分=>1.跨分片的事务一致性难以保证 2.跨库的join关联查询性能较差 3.数据多次扩展难度和维护量极大

### 4.3.2 如何解决分库分表带来的坏处

## 4.4 数据库锁

### 4.4.1 封锁，封锁协议

### 4.4.2 死锁 活锁

## 4.5 优化

# 五．Java框架专题

## 5.1 spring

### 5.1.1搭建并理解各配置

### 5.1.2 aop ioc di

### 5.1.3解读源码，是如何做到解读xml实现反射等功能

## 5.2 springmvc

### 5.2.1搭建并理解各配置

### 5.2.2解读源码

## 5.3 mybatis

### 5.3.1搭建并理解各配置

### 5.3.2 generator一键生成

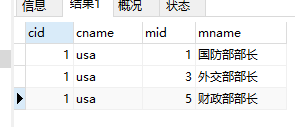
### 5.3.3清楚Po-dao-mapper之间映射

### 5.3.4 sql

### 5.3.5 mybatis做表关联查询

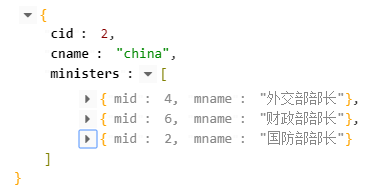
关联查询的结果接收有两种：

1. 用自定义dto接收，但是不是那种很直观的展示结果



1. 定义一个resultMap来接收，是直观的依赖关系的展示结果

<resultMap type="com.provider.demo.entity.Country" id="countryMapper">  
 <id column="cid" property="cid" jdbcType="INTEGER"/>  
 <result column="cname" property="cname" jdbcType="VARCHAR"/>  
 <collection property="ministers" ofType="com.provider.demo.entity.Minister">  
 <id column="mid" property="mid" jdbcType="INTEGER"/>  
 <result column="mname" property="mname" jdbcType="VARCHAR"/>  
 </collection>  
</resultMap>



## 5.3(新) jpa

### 5.3（新）.1 jpa五大接口类

>Repository仅仅是一个标识，表明任何继承它的均为仓库接口类，方便Spring自动扫描识别

>curdRepository 继承Repository，实现了一组CRUD相关的方法

>pagingAndSortingRepository 继承CrudRepository，实现了一组分页排序相关的方法

>JpaRepository 继承PagingAndSortingRepository，实现一组JPA规范相关的方法

>JpaSpecificationExecutor 比较特殊，不属于Repository体系，实现一组JPA Criteria查询相关的方法

## 5.4 springboot

 Spring Boot使编码变简单

Spring Boot使配置变简单

Spring Boot使监控变简单

Spring Boot使部署变简单

### 5.4.1> springboot的搭建与启动

<parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.1.1.RELEASE</version>  
 <relativePath/> <!-- lookup parent from repository -->  
</parent>

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.mybatis.spring.boot</groupId>  
 <artifactId>mybatis-spring-boot-starter</artifactId>  
 <version>1.3.2</version>  
</dependency>

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
</dependency>

新建application.yml(格式百度!)

启动:新建DemoApplication.java

@SpringBootApplication  
public class DemoApplication {  
 public static void main(String[] args) {  
 SpringApplication.*run*(DemoApplication.class, args);  
 }  
}

### 5.4.2> springboot集成mybatis

<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <scope>runtime</scope>  
</dependency>

<!-- 添加druid 依赖包（操作数据源） -->  
<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid</artifactId>  
 <version>${druid.version}</version>  
</dependency>

配置文件application.yml:

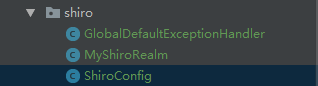
**spring:  
 datasource:  
 url:** jdbc:mysql://localhost:3306/train?useUnicode=true&characterEncoding=utf-8&useSSL=true&serverTimezone=UTC  
 **username:** root  
 **password:** root  
 **driver-class-name:** com.mysql.cj.jdbc.Driver

*## 该配置节点为独立的节点，有很多同学容易将这个配置放在spring的节点下，导致配置无法被识别***mybatis:  
 mapper-locations:** classpath:mapper/\*.xml *#注意：一定要对应mapper映射xml文件的所在路径* **type-aliases-package:** com.example.demo.entity *# 注意：对应实体类的路径*

### 5.4.3> springboot集成shiro

<dependency>  
 <groupId>org.apache.shiro</groupId>  
 <artifactId>shiro-spring</artifactId>  
 <version>1.4.0</version>  
</dependency>

创建如下类（全局异常捕捉-主要针对权限拦截，shrio域，shiro配置文件）



### 5.4.4> 自定义拦截器

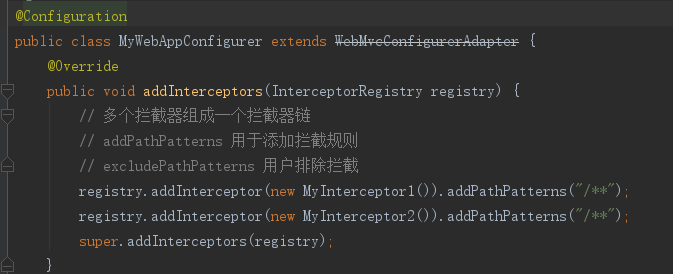
实现自定义拦截器只需要3步：

1、创建我们自己的拦截器类并实现 HandlerInterceptor 接口。

2、创建一个Java类继承WebMvcConfigurerAdapter，并重写 addInterceptors 方法。

2、实例化我们自定义的拦截器，然后将对像手动添加到拦截器链中（在addInterceptors方法中添加）。





## 5.5 springcloud

### 5.5.1> 整合到springboot（父项目有预引进依赖）

A．需要有三个子项目：eureka注册中心，provider提供者，consumer消费者

B．eureka的配置：

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-eureka-server</artifactId>  
 </dependency>

server.port=8088  
##页面  
spring.thymeleaf.prefix=classpath:/templates/  
spring.mvc.view.suffix=.html  
##springcloud  
#eueka 主机名  
eureka.instance.hostname=localhost  
# 不向注册中心注册自己  
eureka.client.register-with-eureka=**false**# 不需要检索服务  
eureka.client.fetch-registry=**false**eureka.client.serviceUrl.defaultZone=http://localhost:${server.port}/eureka/

C．provider配置：

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

#服务名  
spring.application.name=ticket-provider  
#使用ip进行注册  
eureka.instance.prefer-ip-address=**true**#注册地址  
eureka.client.service-url.defaultZone=http://localhost:8088/eureka/

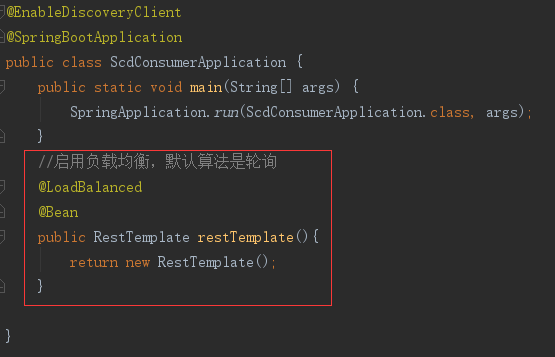
C．consumer配置中心：

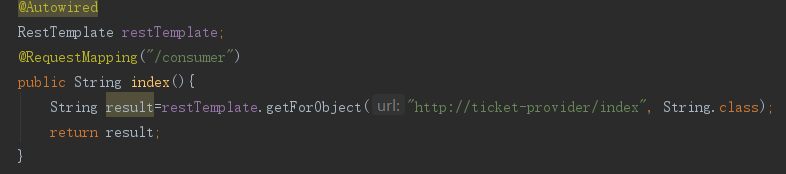
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-eureka</artifactId>  
 </dependency>

##springcloud  
spring.application.name=ticket-customer  
eureka.instance.prefer-ip-address=**true**#注册地址  
eureka.client.service-url.defaultZone=http://localhost:8088/eureka/

### 5.5.2> 消费者调用服务提供的方式

1.restTemplate方式:





2.fegin(像本地方法样调用，也是针对controller)

### 5.5.4> 与dubbo的异同？

<据我所知！>

1.dubbo是针对service的微服务框架，消费者和服务者之间有较强的依赖关系, 而springcloud是针对controller,有点httpClient的感觉。

## 5.6 安全框架shiro

# 六．分布式，缓存专题

## 6.1 dubbo微服务

## 6.2 zookeeper

## 6.3 redis缓存

### 6.3.1 redis的基础知识

A. Redis可以存储键与5种不同数据结构类型之间的映射，这5种数据结构类型分别为String（字符串）、List（列表）、Set（集合）、Hash（散列）和 Zset（有序集合）

B. >1..redis支持16个数据库实例，数据库实例角标0-15，默认登录0号数据库,登录其他数据库语法：select index

>2.事务:开启事务：multi 提交事务：exec 回滚事务：discard 监听事务：watc（乐观锁）

c.redis持久化:

>1.rdb（默认的）： 缺省情况下，Redis会将数据集的快照dump到dump.rdb文件中。此外，我们也可以通过配置文件来修改Redis服务器dump快照的频率，在打开redis.conf文件之后，我们搜索save，可以看到下面的配置信息：  
    save 900 1         #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存照。  
    save 300 10      #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存照。  
    save 60 10000   #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内照。

>2.aof（性能低于rdb）

### 6.3.2 redis在项目中的使用(redistemplate)

a.在spring项目中用redistemplate模板（就我而言），RedisTemplate位于spring-data-redis包下，如何使用？🡺

>1.创建redis.properties和redis.xml

>2.redis.xml里注入

<!—连接池-->

<bean id="poolConfig" class="redis.clients.jedis.JedisPoolConfig">  
 <property name="maxIdle" value="${redis.maxIdle}"/>  
 <property name="maxWaitMillis" value="${redis.maxWait}"/>  
 <property name="testOnBorrow" value="${redis.testOnBorrow}"/>  
</bean>

<!-- redis连接工厂 -->  
<bean id="connectionFactory" class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory">  
 <property name="poolConfig" ref="poolConfig"/>  
 <property name="port" value="${redis.port}"/>  
 <property name="hostName" value="${redis.host}"/>  
 <property name="password" value=""/>  
 <property name="timeout" value="${redis.timeout}"></property>  
</bean>

<!—redisTemplate模板-->

<bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate">  
 <property name="connectionFactory" ref="connectionFactory"/>  
 <!--序列化key-->  
 <property name="keySerializer">  
 <bean class="org.springframework.data.redis.serializer.StringRedisSerializer"/>  
 </property>  
 <!--序列化value-->  
 <property name="valueSerializer">  
 <bean class="org.springframework.data.redis.serializer.JdkSerializationRedisSerializer"/>  
 </property>  
</bean>

<!—redisutil看情况吧，一般是要配合util使用-->

<bean id="redisUtil" class="com.github.Duankan.utils.RedisUtil">  
 <property name="redisTemplate" ref="redisTemplate"/>  
</bean>

>3.redisTemplate的使用总结：增删查改（序列化操作，放取都需序列化）, redisTemplate定义了五种数据结构操作（具体操作百度）：

redisTemplate.opsForValue();//操作字符串

redisTemplate.opsForHash();//操作hash

redisTemplate.opsForList();//操作list

redisTemplate.opsForSet();//操作set

redisTemplate.opsForZSet();//操作有序set

ValueOperations<Serializable, Object> operations = redisTemplate.opsForValue();//序列化操作，放取都需序列化  
operations.set(key, value);

Operations.get(key);

### 6.3.3 redis的几种主流序列化与反序列化



1.jdk序列化与反序列化

//jdk序列化  
public static byte[] serializer\_jdk(Object object){  
 ObjectOutputStream oos=null;  
 ByteArrayOutputStream baos=null;  
 try{  
 baos=new ByteArrayOutputStream();  
 oos=new ObjectOutputStream(baos);//包装模式？  
 oos.writeObject(object);  
 byte[] bytes=baos.toByteArray();  
 return bytes;  
 }  
 catch (Exception e){  
 e.printStackTrace();  
 }  
 return null;  
}  
//jdk反序列化  
public static Object unSerializer\_jdk(byte[] bytes){  
 ObjectInputStream ois=null;  
 ByteArrayInputStream bais=null;  
 try{  
 bais=new ByteArrayInputStream(bytes);  
 ois=new ObjectInputStream(bais);  
 return ois.readObject();  
 }  
 catch (Exception e){  
 e.printStackTrace();  
 }  
 return null;  
}

2.protostuff序列化与反序列化

3.kryo序列化与反序列化（\*我写了一整套obj list map set的序列与反序列化\*）

3.1 自定义object的序列与反序列化

*/\*\*  
 \** ***@param*** <*T*> *obj 要序列化的对象  
 \** ***@return*** *String  
 \** ***@desc*** *kryo序列化object*

*1.采用包装模式，实际上操作的流:baos  
 \*/*public static <T extends Serializable> String serializationObject\_kryo(T obj) {  
 Kryo kryo = new Kryo();  
 kryo.setReferences(false);//???  
 kryo.register(obj.getClass(), new JavaSerializer());//注册序列化类型  
 ByteArrayOutputStream baos = new ByteArrayOutputStream();  
 Output output = new Output(baos);  
 kryo.writeClassAndObject(output, obj);//将obj写到out  
 output.flush();  
 output.close();  
 byte[] b = baos.toByteArray();//字节数组输出流转换为字节码  
 try {  
 baos.flush();  
 baos.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 return new String(new Base64().encode(b));  
}

*/\*\*  
 \** ***@param*** *obj redis中取出的string  
 \** ***@param*** <*T*> *clazz要序列化输出的对象类型  
 \** ***@return*** *T 序列化输出的对象类型  
 \** ***@desc*** *kryo反序列化object  
 \*/*public static <T extends Serializable> T deserializationObject(String obj, Class<T> clazz) {  
 Kryo kryo = new Kryo();  
 kryo.setReferences(false);  
 kryo.register(clazz, new JavaSerializer());  
 ByteArrayInputStream bais = new ByteArrayInputStream(new Base64().decode(obj));  
 Input input = new Input(bais);  
 return (T) kryo.readClassAndObject(input);  
}

3.2 list的序列化与反序列化

*/\*\*  
 \** ***@param*** *obj 要序列化的list  
 \** ***@param*** *clazz List中元素的类型  
 \** ***@return*** *String  
 \** ***@desc*** *kryo序列化list  
 \*/*public static <T extends Serializable> String serializationList(List<T> obj, Class<T> clazz) {  
 Kryo kryo = new Kryo();  
 kryo.setReferences(false);  
 kryo.setRegistrationRequired(true);  
  
 CollectionSerializer serializer = new CollectionSerializer();//注册序列化类型  
 serializer.setElementClass(clazz, new JavaSerializer());  
 serializer.setElementsCanBeNull(false);  
 kryo.register(clazz, new JavaSerializer());  
 kryo.register(ArrayList.class, serializer);  
  
 ByteArrayOutputStream baos = new ByteArrayOutputStream();  
 Output output = new Output(baos);  
 kryo.writeObject(output, obj);  
 output.flush();  
 output.close();  
 byte[] b = baos.toByteArray();  
 try {  
 baos.flush();  
 baos.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 return new String(new Base64().encode(b));  
}

*/\*\*  
 \** ***@param*** *obj redis中取得的string  
 \** ***@param*** *clazz 序列化输出的对象类型  
 \** ***@return*** *List  
 \** ***@desc*** *kryo反序列化list  
 \*/*public static <T extends Serializable> List<T> deserializationList(String obj, Class<T> clazz) {  
 Kryo kryo = new Kryo();  
 kryo.setReferences(false);  
 kryo.setRegistrationRequired(true);  
  
 CollectionSerializer serializer = new CollectionSerializer();  
 serializer.setElementClass(clazz, new JavaSerializer());  
 serializer.setElementsCanBeNull(false);  
  
 kryo.register(clazz, new JavaSerializer());  
 kryo.register(ArrayList.class, serializer);  
 ByteArrayInputStream bais = new ByteArrayInputStream(new Base64().decode(obj));  
 Input input = new Input(bais);  
 return (List<T>) kryo.readObject(input, ArrayList.class, serializer);  
}

3.3 map的序列化与反序列化

*/\*\*  
 \** ***@param*** *obj  
 \** ***@param*** *clazz  
 \** ***@param*** <*T*>  
 *\** ***@return*** *String  
 \** ***@desc*** *kryo序列化Map  
 \*/*public static <T extends Serializable> String serializationMap(Map<String, T> obj, Class<T> clazz) {  
 Kryo kryo = new Kryo();  
 kryo.setReferences(false);  
 kryo.setRegistrationRequired(true);  
 MapSerializer serializer = new MapSerializer();  
 serializer.setKeyClass(String.class, new JavaSerializer());  
 serializer.setKeysCanBeNull(false);  
 serializer.setValueClass(clazz, new JavaSerializer());  
 serializer.setValuesCanBeNull(true);  
 kryo.register(clazz, new JavaSerializer());  
 kryo.register(HashMap.class, serializer);  
 ByteArrayOutputStream baos = new ByteArrayOutputStream();  
 Output output = new Output(baos);  
 kryo.writeObject(output, obj);  
 output.flush();  
 output.close();  
 byte[] b = baos.toByteArray();  
 try {  
 baos.flush();  
 baos.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 return new String(new Base64().encode(b));  
}

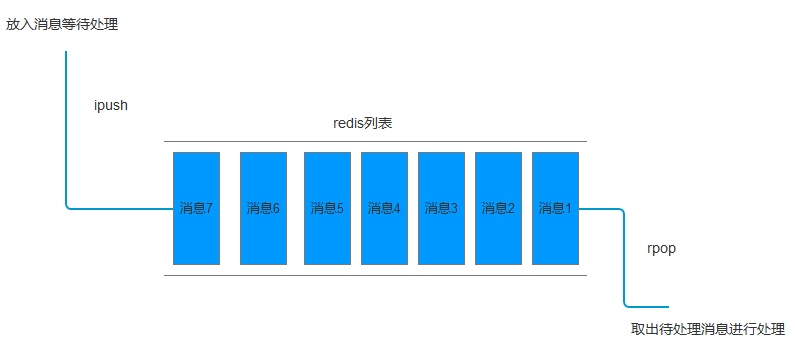
*/\*\*  
 \** ***@param*** *obj  
 \** ***@param*** *clazz  
 \** ***@param*** <*T*>  
 *\** ***@return*** *Map  
 \** ***@desc*** *kryo反序列化Map  
 \*/*public static <T extends Serializable> Map<String, T> deserializationMap(String obj, Class<T> clazz) {  
 Kryo kryo = new Kryo();  
 kryo.setReferences(false);  
 kryo.setRegistrationRequired(true);  
 MapSerializer serializer = new MapSerializer();  
 serializer.setKeyClass(String.class, new JavaSerializer());  
 serializer.setKeysCanBeNull(false);  
 serializer.setValueClass(clazz, new JavaSerializer());  
 serializer.setValuesCanBeNull(true);  
 kryo.register(clazz, new JavaSerializer());  
 kryo.register(HashMap.class, serializer);  
 ByteArrayInputStream bais = new ByteArrayInputStream(  
 new Base64().decode(obj));  
 Input input = new Input(bais);  
 return (Map<String, T>) kryo.readObject(input, HashMap.class,  
 serializer);  
}

3.4 set的序列化与反序列化

*/\*\*  
 \** ***@param*** *obj  
 \** ***@param*** *clazz  
 \** ***@param*** <*T*>  
 *\** ***@return*** *String  
 \** ***@desc*** *kryo序列化Set  
 \*/*public static <T extends Serializable> String serializationSet(Set<T> obj, Class<T> clazz) {  
 Kryo kryo = new Kryo();  
 kryo.setReferences(false);  
 kryo.setRegistrationRequired(true);  
 CollectionSerializer serializer = new CollectionSerializer();  
 serializer.setElementClass(clazz, new JavaSerializer());  
 serializer.setElementsCanBeNull(false);  
 kryo.register(clazz, new JavaSerializer());  
 kryo.register(HashSet.class, serializer);  
 ByteArrayOutputStream baos = new ByteArrayOutputStream();  
 Output output = new Output(baos);  
 kryo.writeObject(output, obj);  
 output.flush();  
 output.close();  
 byte[] b = baos.toByteArray();  
 try {  
 baos.flush();  
 baos.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 return new String(new Base64().encode(b));  
}

*/\*\*  
 \** ***@param*** *obj  
 \** ***@param*** *clazz  
 \** ***@param*** <*T*>  
 *\** ***@return*** *Set  
 \** ***@desc*** *kryo反序列化Set  
 \*/*public static <T extends Serializable> Set<T> deserializationSet(String obj, Class<T> clazz) {  
 Kryo kryo = new Kryo();  
 kryo.setReferences(false);  
 kryo.setRegistrationRequired(true);  
 CollectionSerializer serializer = new CollectionSerializer();  
 serializer.setElementClass(clazz, new JavaSerializer());  
 serializer.setElementsCanBeNull(false);  
 kryo.register(clazz, new JavaSerializer());  
 kryo.register(HashSet.class, serializer);  
 ByteArrayInputStream bais = new ByteArrayInputStream(  
 new Base64().decode(obj));  
 Input input = new Input(bais);  
 return (Set<T>) kryo.readObject(input, HashSet.class, serializer);  
}

### 6.3.4 redis做消息队列



Redis的列表使用双向链表，保存了头尾，在两边插曲元素效率比较高。所以可以直接使用Redis的List实现消息队列，只需简单的两个指令lpush和rpop或者rpush和lpop

实例模型如下：

消息生产者：

消息消费者：

### 6.3.5 发布订阅

## 6.4 elasticsearch搜素引擎

## 6.5 hadoop大数据

## 6.6 nginx负载均衡

## 6.7 docker

# 七．事务，队列，工作流等专题

## 7.1 Jdbc事务

## 7.2 ssm框架事务

## 7.3分布式消息中间件

## 7.3.1 rabitmq

### 7.3.1.1 rabbitmq简介

消息队列，是一种应用程序对应用程序的通信方法。在项目中，将一些无需即时返回且耗时的操作提取出来，进行了异步处理，节省了服务器的请求响应时间，提高了系统的吞吐量。

### 7.3.1.2 rabbitmq安装

>windows下：1.先安装Erlang,并配置环境变量

2.下载rabbitmq-server-3.6.5

3.进入rabbitmq-plugins.bat目录，执行enable rabbitmq\_management

4.net stop RabbitMQ && net start RabbitMQ启动rabbitmq

### 7.3.1.3 springboot整合rabbitmq

>1.引进依赖：

<!--rabbitmq依赖-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-amqp</artifactId>  
</dependency>

>2.提供者配置：

>application配置:

##rabbitmq相关配置  
spring.rabbitmq.host=localhost  
spring.rabbitmq.port=5672  
spring.rabbitmq.username=guest  
spring.rabbitmq.password=guest

>发送者示例：

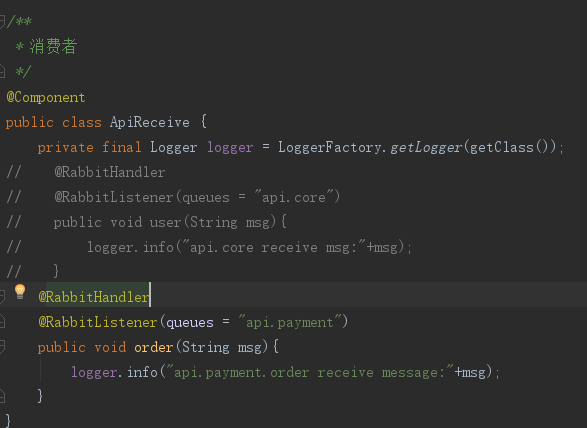


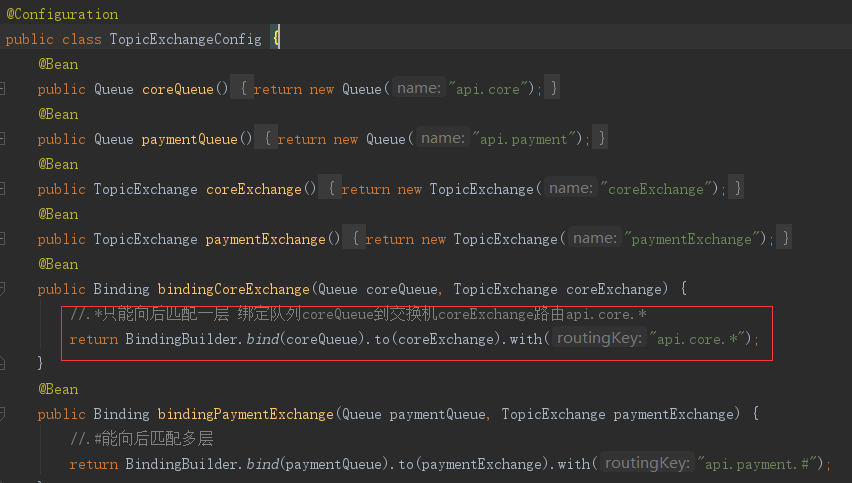
>3.消费者配置：

>application配置：

##rabbitmq相关配置##  
spring.rabbitmq.host=localhost  
spring.rabbitmq.port=5672  
spring.rabbitmq.username=guest  
spring.rabbitmq.password=guest

>消费类和交换机配置:





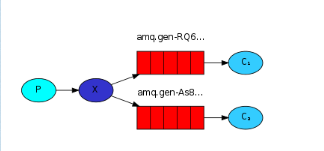
### 7.3.1.4 消息，队列，交换机，路由

>消息：传输的文本等数据

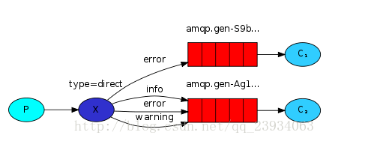
>队列:queue生产者将消息通过绑定Key交换到各个队列，存储消息

>交换机Exchage:

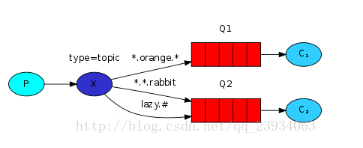
1.fanout 路由规则很简单，它会把所有发送到该exchange绑定的队列中



2.direct 会把消息路由到绑定key和路由Key完全匹配的队列中



1. **Topic 把消息路由到能通过模糊匹配和匹配规则的队列中**



>路由：

### 7.3.1.5 rabbitmq使用常见问题

>1.如何禁止大量的消息涌到消费者？

## 7.3.2 activemq

## 7.4工作流：activity

# 八．linux专题

# 九．jvm db调优专题

## 9.1 栈内存 堆内存

9.1.1 栈内存（stack）: 用来存储局部变量，基本数据类型和对象的引用的地址

9.1.1.1 优缺点：速度快，空间小，用完就释放空间（数组，对象）

9.1.2 堆内存(heap)：动态运行分配内存，空间大，速度慢

如下一段代码1：

Private int a=6;

Sout(“before change a is ”+a);

Change(a); //6

Sout(“after change a is ”+a); //6

Public void Change(int v){

v=8;

}

图解1：

栈内存开辟空间：

a=6

b=8

如下代码2：

Private Some some=new Some();

Some.name=”test”;

Some.addr=”武汉市江夏区”;

Sout(“before change name is”+some.name);//test

Change(Some some);

Sout(“afterchange name is”+some.name);//to2

Public void Change(Some sm){

Sm.name=”to2”;

}

图解2：

栈内存： 堆内存：

0x1a

some 都指向 开辟空间

0x1a name

addr

sm

0x1a

如下代码3：

Private Some sm1=new Some();

sm1.name=”sm1”;

sm1.addr=”武汉市江夏区”;

Sout(“before change sm1.name is ”+sm1.name);//sm1

Change(sm1);

Sout(“after change sm1.name is ”+sm1.name);//sm1

Public void Change(Some some){

Some sm2=new Some();

sm2.name=”sm2”;

some=sm2;

}

图解3：

栈内存： 堆内存：

sm1 01xa sm1 “sm1”

01xa “江夏区”

Some 01xa

01xb “sm2”

sm2 01xb sm2

9.1.3 总结：

1.一个方法不能不变传入基本类型的参数值

2.一个方法不能改变传入对象类型的参数的引用地址

3.一个方法能改变传入对象参数的某个属性

# 十．前端专题

## 10.1 vue ivew

## 10.2 js高级

## 10.3 jquery

## 10.4 bootstrap layer

## 10.5 css html布局

# 十一.开发工具专题

## 11.1 idea

## 11.2 git

## 11.3 maven

## 11.4 mysql

## 11.5 svn

## 11.6 tomcat

## 11.7 linux虚拟机

## 11.8 vscode