# 微服务

springclould是一套分布式解决方案

## eureka

注册中心,类似于zookeeper,保存服务的ip地址

### 服务器配置

pom引入eureka server

<dependency>

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

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

</dependency>

主启动类加注解

@EnableEurekaServer

yml配置自我保护

eureka:

server:

# 开起自我保护模式

# eureka 的自我保护状态：心跳失败的比例，在15分钟内是否低于85%,

# 如果出现了低于的情况，Eureka Server会将当前的实例注册信息保护起来，同时提示一个警告，

# 一旦进入保护模式，Eureka Server将会尝试保护其服务注册表中的信息，不再删除服务注册表中的数据。

# 也就是不会注销任何微服务

enable-self-preservation: false

eureka:

instance:

hostname: eureka1

### 客户端配置

引入eureka client

<dependency>

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

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

</dependency>

设置eureka地址

eureka:

client:

service-url:

defaultZone: <http://eureka1:2001/eureka,http://eureka1:2002/eureka>

主启动类加客户端注解

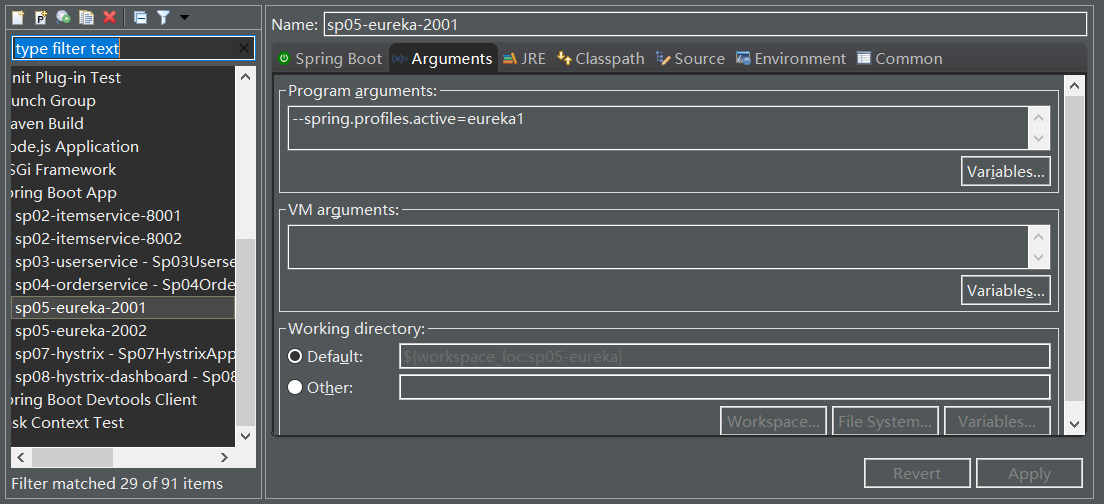
@EnableDiscoveryClient

### 集群

分区配置,服务器右键open config设置启动参数

--spring.profiles.active=eureka1

不执行时尝试手打,复制可能有特殊字符不报错



分别设置另一台服务器的地址用于注册,提前配置hosts文件

spring:

application:

name: eureka

server:

port: 2001

eureka:

server:

# 关闭自我保护模式

enable-self-preservation: false

---

spring:

profiles: eureka1

server:

port: 2001

eureka:

instance:

hostname: eureka1

client:

service-url:

defaultZone: http://eureka2:2002/eureka

---

spring:

profiles: eureka2

server:

port: 2002

eureka:

instance:

hostname: eureka2

client:

service-url:

defaultZone: <http://eureka2:2001/eureka>

register-with-eureka: true #向另一台eureka注册自己

fetch-registry: true #拉取另一台eureka的注册表

这两个配置默认都是开,如果不配置地址eureka也会默认的不停向端口号8761注册自己(但是没有另一台服务器就会失败)

### eureka与zookeeper区别

* eureka
  + AP - 强调可用性,eureka在服务不可用时会保护ip,并且在宕机时服务可以使用本地缓存
  + 集群 - 对等结构
* zookeeper
  + CP - 强调一致性
  + 集群 - 主从结构

## RestTemplate

RestTemplate是String Boot提供的工具

类似 HttpClient, Jsoup,是一个http远程连接,远程调用工具

### Spring Clould远程调用与dubbo对比



### 相关方法

* getForObject(url, 转换类型, 提交的参数)
* postForObject(url, 提交的数据, 转换的类型)

## ribbon

### 简介

提供远程连接服务,支持超时重试,需要配置超时时间,重试次数和重试服务器的个数,使用的工具就是RestTemplate

### 配置RestTemplate

// 创建 RestTemplate 实例，并存入 spring 容器

@Bean

@LoadBalanced

public RestTemplate getRestTemplate() {

// 使用工厂设置超时

SimpleClientHttpRequestFactory f = new SimpleClientHttpRequestFactory();

// 连接服务器的超时时间

f.setConnectTimeout(1000);

// 响应等待超时

f.setReadTimeout(1000);

return new RestTemplate(f);

}

使用spring管理, @LoadBalanced表示启用负载均衡,设置超时时间只能用工厂而不可以用yml

### 使用restTemplatet发起请求

自动注入restTemplatet就可使用

@Autowired

private RestTemplate restTemplatet;

// 向指定微服务地址发送 get 请求，并获得该服务的返回结果

// {1}是占位符代表第一个参数(orderId)

return rt.getForObject("http://localhost:8001/{1}", JsonResult.class, orderId);

return rt.postForObject("http://localhost:8001/decreaseNumber", items, JsonResult.class);

### ribbon负载均衡

添加依赖(此时使用的才是ribbon功能)

eureka 依赖中已经包含了 ribbon,如果使用eureka则无需配置

<dependency>

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

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

</dependency>

### ribbon重试

添加依赖

<dependency>

<groupId>org.springframework.retry</groupId>

<artifactId>spring-retry</artifactId>

</dependency>

yml配置

ribbon:

# 重试的次数

MaxAutoRetries: 1

# 一台服务器无应答后重试的服务器个数

MaxAutoRetriesNextServer: 2

# 对get和post请求都重试,默认是false,开启后要注意避免重复提交表单

OkToRetryOnAllOperations: true

这个配置最多访问3台服务器各2次共6次

### 程序设置延时

// 设置随机延迟

if(Math.random()<0.6) {

long t = new Random().nextInt(5000);

log.info("item-service-"+port+" - 暂停 "+t);

Thread.sleep(t);

}

## hystrix

### 降级和熔断

支持降级和熔断

降级:请求失败后执行的一段其他代码,生成结果返回给客户端, 超时、出错、不可到达时，对服务降级，返回错误信息或者是缓存数据

熔断:类似于保险丝的机制,当一段时间内的请求过多而成功率过低时,将所有请求直接降级

### 超时

hystrix自身也支持超时,默认一秒钟.这个时间应当设置为比ribbon的超时总时间更长, 因为熔断后ribbon会继续向后台发起请求.

例如ribbon设置超时1s,重试1次最多2个服务器,最多请求6次总延时就是6s,hystrix最好设置大于6s,例如设置为10s

yml修改这个值,单位是ms

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds

### 降级方法

为原方法添加 @HystrixCommand 注解，fallbackMethod属性指定降级方法名,降级方法定义在控制器中和原方法参数类型返回值一致

原方法

@GetMapping("/item-service/{orderId}")

@HystrixCommand(fallbackMethod = "getItemsFB")

public JsonResult<List<Item>> getItems(@PathVariable String orderId) {

// 向指定微服务地址发送 get 请求，并获得该服务的返回结果

// {1} 占位符，用 orderId 填充

return restTemplatet.getForObject([http://item-service/{1}](http://item-service/%7b1%7d)  
, JsonResult.class, orderId);

}

降级方法

// 降级方法的参数和返回值，需要和原始方法一致，方法名任意

public JsonResult<List<Item>> getItemsFB(String orderId) {

return JsonResult.err("获取订单商品列表失败");

}

### 配置

添加 hystrix 起步依赖

<dependency>

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

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

</dependency>

主启动类添加注解@SpringCloudApplication

其在@SpringBootApplication之上集合了@EnableDiscoveryClient和@EnableCircuitBreaker(回路中断),支持断路器

### 修改默认设置

hystrix默认的设置如下

* 10秒20次请求
* 50%失败,执行降级
* 满足上面两个条件会触发熔断
* 5秒后进入半开状态,会尝试请求后台服务,如果失败,断路器继续保持断开状态(请求仍被降级); 如果请求成功,关闭断路器(请求正常)

hystrix.command.default.circuitBreaker.requestVolumeThreshold

10秒内请求数量，默认20，如果没有达到该数量，即使请求全部失败，也不会触发断路器打开

hystrix.command.default.circuitBreaker.errorThresholdPercentage

失败请求百分比，达到该比例则触发断路器打开

hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds

断路器打开多长时间后，再次允许尝试访问（半开），仍失败则继续保持打开状态，如成功访问则关闭断路器，默认 5000

## actuator

### 简介

Spring Boot提供的一个数据监控工具,可以暴露 hystrix 的监控端点

可以监控:

* 健康状态
* 运行环境
* Spring容器中所有的对象
* ……

节点中包括health,环境变量等等,请求返回的是json

hystrix.stream显示了降级和熔断的数据

### 配置

编辑起步依赖, 添加 actuator 依赖

<dependency>

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

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

</dependency>

暴露节点,可以填带引号的”\*”,代表全部节点

management:

endpoints:

web:

exposure:

include: hystrix.stream

### hystrix dashboard

专门监控hystrix的数据的项目,是基于actuator节点的仪表盘,和其他项目无关,pom直接引入hystrix dashboard

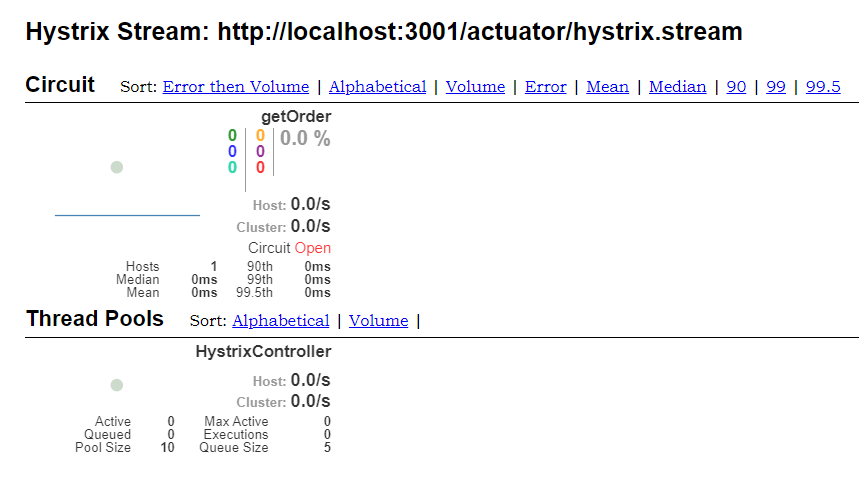
主程序添加 @EnableHystrixDashboard 和 @EnableDiscoveryClient,

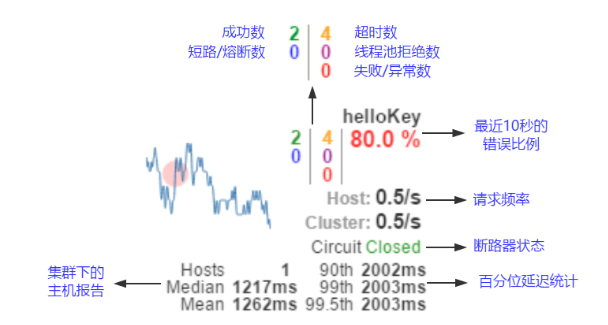
访问路径

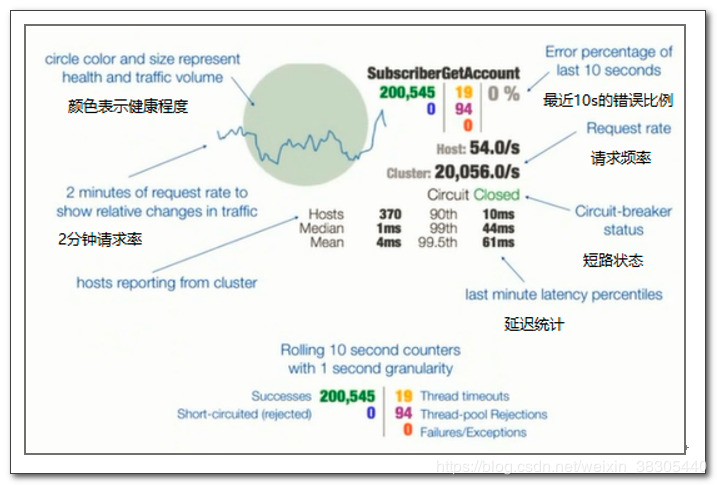
http://localhost:4001/hystrix

要填入节点的地址

<http://localhost:3001/actuator/hystrix.stream>







### apache 的并发访问测试工具 ab

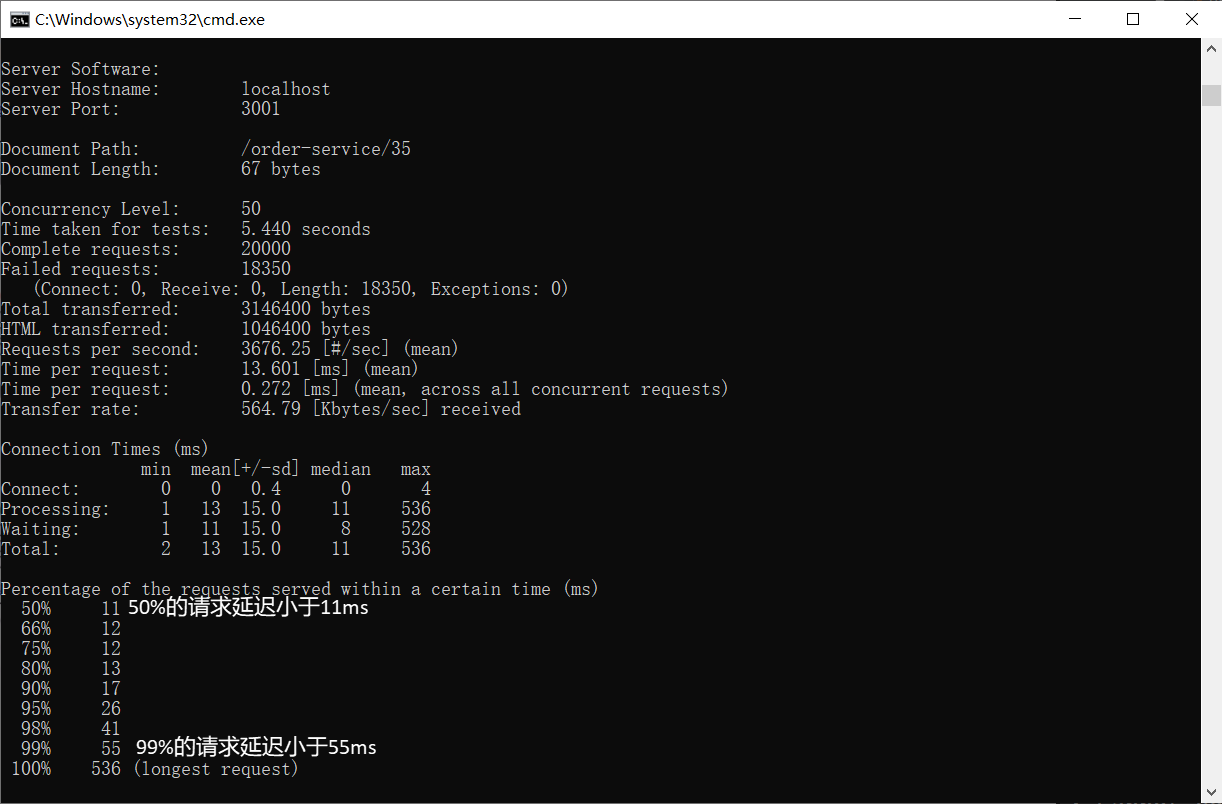
下载地址

<http://httpd.apache.org/docs/current/platform/windows.html#down>

bin目录下调出cmd窗口执行指令

ab -n 20000 -c 20 <http://localhost:3001/item-service/35>

-n代表20000个访问,-c代表20并发,相当于20个人同时访问



## feign

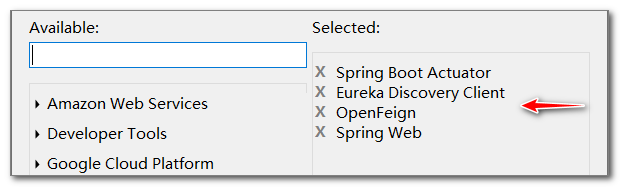
### 简介

feign实现声明式客户端,简化了调用的代码,用来在多个服务之间实现链式调用,整合ribbon和hystrix.

feign中定义了service接口,就像实际的接口一样调用,使用起来类似于dubbo,将控制器的常用注解用于service的方法,用于指定请求的url以及参数,基于http远程调用向后台的控制器发出请求.

### 配置

* 项目



* 主程序

主程序添加 @EnableDiscoveryClient 和 @EnableFeignClients

* service

添加注解@FeignClient("微服务名")指定连接的服务

@FeignClient("item-service")

* 方法

示例程序:

@GetMapping("/{userId}/score")

JsonResult addScore(@PathVariable Integer userId,   
@RequestParam Integer score);

当这样调用该方法：

service.addScore(7, 100);

那么 feign 会向服务器发送请求：

http://用户微服务/7/score?score=100

注意：如果 score 参数名与变量名不同，需要添加参数名设置：

@GetMapping("/{userId}/score")

JsonResult addScore(@PathVariable Integer userId,  
 @RequestParam("score") Integer s);

### feign + ribbon负载均衡和重试

fegin默认启用了ribbon,免配置

重试的默认配置参数:

ConnectTimeout=1000

ReadTimeout=1000

MaxAutoRetries=0

MaxAutoRetriesNextServer=1

yml中ribbon.xxx 全局配置

item-service.ribbon.xxx 对特定服务实例的配置

ribbon:

ConnectTimeout: 1000

ReadTimeout: 1000

item-service:

ribbon:

MaxAutoRetries: 1

MaxAutoRetriesNextServer: 2

ConnectTimeout: 1000

ReadTimeout: 500

### feign + hystrix降级

feign 默认没有启用 hystrix，添加配置，可以启用 hystrix的部分功能(如降级)

降级的动作不需要额外依赖

默认超时时间1s,可以修改

feign:

hystrix:

enabled: true

hystrix:

command:

default:

execution:

isolation:

thread:

timeoutInMilliseconds: 500

在只有hystrix时,降级方法定义在hystrix的控制器中,在原方法上使用注解标识降级方法名

使用feign + hystrix时, feign的控制器是正常调用,sevice是接口,需要定义接口的实现类编写降级方法

配置方法:在原方法的@FeignClient 注解上添加属性fallback = 接口实现类的类型, 降级类需要实现远程接口并添加@Component

...

@FeignClient(name="item-service", fallback =   
ItemFeignServiceFB.class)

public interface ItemFeignService {

...

降级类

@Component

public class ItemFeignServiceFB implements ItemFeignService {

@Override

public JsonResult<List<Item>> getItems(String orderId) {

return JsonResult.err("无法获取订单商品列表");

}  
}

### feign + hystrix熔断

feign 没有包含完整的 hystrix 依赖

右键点击项目,编辑起步依赖,添加hystrix依赖

<dependency>

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

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

</dependency>

熔断需要主程序添加@EnableCircuitBreaker

加上@EnableDiscoveryClient和@SpringBootApplication,也可以使用@SpringCloudApplication

### feign + actuator监控

需要 actuator 依赖

<dependency>

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

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

</dependency>

yml 暴露 hystrix.stream 监控端点

management:

endpoints:

web:

exposure:

include: hystrix.stream

访问hystrix dashboard:

http://localhost:4001/hystrix

填入 feign 监控路径:

<http://localhost:3001/actuator/hystrix.stream>

ab指令

ab -n 20000 -c 50 http://localhost:3001/item-service/35

### fegin + order service 微服务间调用

order service 调用商品库存服务和用户服务

1. pom.xml

添加依赖

actuator

feign

hystrix

1. application.yml

启用feign,暴露监控端点

feign:

hystrix:

enabled: true

management:

endpoints:

web:

exposure:

include: hystrix.stream

1. 主程序

@EnableFeignClients

@SpringCloudApplication

1. ItemFeignService
2. UserFeignService
3. ItemFeignServiceFB
4. UserFeignServiceFB

添加在order-service项目中

1. OrderServiceImpl

@Override

public Order getOrder(String orderId) {

//调用user-service获取用户信息

JsonResult<User> user = userService.getUser(7);

//调用item-service获取商品信息

JsonResult<List<Item>> items = itemService.getItems(orderId);

Order order = new Order();

order.setId(orderId);

order.setUser(user.getData());

order.setItems(items.getData());

return order;

}

@Override

public void addOrder(Order order) {

//调用item-service减少商品库存

itemService.decreaseNumber(order.getItems());

//TODO: 调用user-service增加用户积分

userService.addScore(7, 100);

log.info("保存订单："+order);

}

## turbine

### hystrix + turbine 集群聚合监控

turbine 可以汇集监控信息，将聚合后的信息提供给 hystrix dashboard 来集中展示和监控

turbine 项目依赖Turbine和eureka client

yml中要配置eureka和监控的应用名称,用逗号分隔,下面指定集群的名称

turbine:

app-config: feign,order-service

cluster-name-expression: new String("default")

主程序添加 @EnableTurbine 和 @EnableDiscoveryClient 注解

turbine 监控路径,默认开启了这个监控节点

http://localhost:5001/turbine.stream

在 hystrix dashboard 中填入turbine 监控路径，开启监控

<http://localhost:4001/hystrix>

## zuul

### 简介

为微服务应用提供统一的对外访问接口

### 配置

依赖zuul和eureka client,公共类

yml配置路由设置,默认设置是将服务id作为路径

zuul:

routes:

item-service: /item-service/\*\*

user-service: /user-service/\*\*

order-service: /order-service/\*\*

主程序添加 @EnableZuulProxy 和 @EnableDiscoveryClient 注解

### zuul + ribbon负载均衡

zuul 已经集成了 ribbon，默认已经实现了负载均衡

### zuul + ribbon 重试

zuul默认不支持重试,需要 spring-retry 依赖

yml设置开启,还可以设置ribbon的超时时间和重试次数

zuul:

retryable: true

### zuul + hystrix 降级

定义降级类实现ItemServiceFallback接口, 添加@Component交由Spring管理

@Component

public class ItemServiceFallback implements FallbackProvider {

/\*\*

\* @author Ericson

\* @date 2020/03/12 00:32

\* @description 指定应用此降级类的服务id，星号或null值可以通配所有服务

\*/

@Override

public String getRoute() {

return "item-service";

}

/\*\*

\* @author Ericson

\* @date 2020/03/12 00:32

\* @param route

\* @param cause

\* @return

\* @description 向客户端返回封装降级响应的response对象

\*/

@Override

public ClientHttpResponse fallbackResponse(String route,

Throwable cause) {

return response();

}

/\*\*

\* @author Ericson

\* @date 2020/03/12 00:38

\* @return response

\* @description ClientHttpResponse中封装降级响应

\*/

private ClientHttpResponse response() {

return new ClientHttpResponse() {

// 下面三个都是协议号相关的方法

@Override

public HttpStatus getStatusCode() throws IOException {

return HttpStatus.OK;

}

@Override

public int getRawStatusCode() throws IOException {

return HttpStatus.OK.value();

}

@Override

public String getStatusText() throws IOException {

return HttpStatus.OK.getReasonPhrase();

}

@Override

public void close() {}

@Override

public InputStream getBody() throws IOException {

log.info("fallback body");

String s = JsonResult.err()

.msg("后台服务错误")

.toString();

return new ByteArrayInputStream(s.getBytes("UTF-8"));

}

@Override

public HttpHeaders getHeaders() {

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

return headers;

}

};

}

}

### zuul + hystrix 数据监控

zuul 已经包含 actuator 依赖,yml中暴露监控端点

management:

endpoints:

web:

exposure:

include: hystrix.stream

启动 sp08-hystrix-dashboard，填入 zuul 的监控端点路径，开启监控

http://localhost:4001/hystrix

填入监控端点：

<http://localhost:3001/actuator/hystrix.stream>

### zuul + turbine 聚合监控

turbine 项目的yml中可以直接添加zuul

turbine:

app-config: order-service, zuul

cluster-name-expression: new String("default")

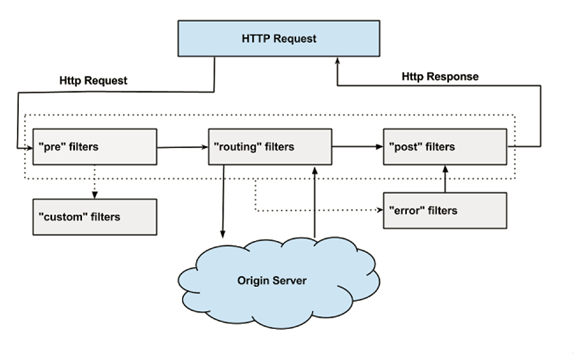
turbine监控端点地址

<http://localhost:5001/turbine.stream>

注意:turbine只监测zuul时没有数据,必须联合zuul的被访问者

### zuul 请求过滤

可以添加过滤器一次过滤所有服务的请求,例如用户登录的校验



定义过滤器，继承 ZuulFilter,添加@Component

/\*\*

\* @author Ericson

\* @class AccessFilter

\* @date 2020/03/12 01:30

\* @version 1.0

\* @description zuul的自定义过滤器

\*/

@Component

public class AccessFilter extends ZuulFilter {

/\*\*

\* @author Ericson

\* @date 2020/03/12 01:30

\* @see com.netflix.zuul.IZuulFilter#shouldFilter()

\* @description 方法决定是否执行下面的run方法,true执行,false不执行

\*/

@Override

public boolean shouldFilter() {

// 对指定的serviceid过滤，如果要过滤所有服务，直接返回 true

RequestContext ctx = RequestContext.getCurrentContext();

String serviceId = (String)ctx.get(FilterConstants.SERVICE\_ID\_KEY);

if (serviceId.equals("item-service")) {

return true;

}

return false;

}

/\*\*

\* @author Ericson

\* @date 2020/03/12 01:31

\* @return 这个值目前没有使用

\* @throws ZuulException

\* @see com.netflix.zuul.IZuulFilter#run()

\* @description 具体的过滤处理方法,关键是setSendZuulResponse

\*/

@Override

public Object run() throws ZuulException {

RequestContext ctx = RequestContext.getCurrentContext();

HttpServletRequest req = ctx.getRequest();

String token = req.getParameter("token");

if (token == null) {

// 此设置会阻止请求被路由到后台微服务

ctx.setSendZuulResponse(false);

// 向客户端的响应

ctx.setResponseStatusCode(200);

ctx.setResponseBody(JsonResult.err()

.code(JsonResult.NOT\_LOGIN)

.toString());

}

// 目前该返回值没有被使用

return null;

}

/\*\*

\* @author Ericson

\* @date 2020/03/12 01:34

\* @description 制定了过滤器的类型

\*/

@Override

public String filterType() {

return FilterConstants.PRE\_TYPE;

}

@Override

public int filterOrder() {

// 该过滤器顺序要 > 5，才能得到 serviceid

return FilterConstants.PRE\_DECORATION\_FILTER\_ORDER + 1;

}

}

## config 配置中心

### 原理

yml 配置文件保存到 git 服务器，例如 github.com, 配置服务启动时，从git服务器克隆配置信息仓库保存到本地,其他服务访问配置服务获取配置文件

### 配置文件

创建普通项目config,就是一个文件夹,放入所有的yml配置,修改对应的名称

item-service-dev.yml

user-service-dev.yml

order-service-dev.yml

zuul-dev.yml

-dev规定为开发环境,类似的还有-test和-prod代表测试环境和产品环境,也可自定义后缀或无后缀,可以使用/user-service/dev或者/user-service-dev.yml两种形式访问

可以设置禁止配置中心的配置将客户端配置覆盖掉,保证多个端口启动

spring:

......

cloud:

config:

override-none: true

将 config 项目分享到本地仓库并上传到 github远程仓库

### config 服务器设置

config 配置中心从 git 下载所有配置文件,而其他微服务启动时从 config 配置中心获取配置信息。

依赖config server,eureka client

yml

cloud:

config:

server:

git:

uri: https://github.com/hozuo/config-repo

searchPaths: config

主程序添加 @EnableConfigServer 和 @EnableDiscoveryClient

### 微服务客户端设置

添加 config client 依赖,添加bootstrap.yml，引导配置文件，先于 application.yml 加载,设置配置服务器名,自己的名和选择的开发环境(dev)

spring:

cloud:

config:

discovery:

enabled: true

service-id: config-server

name: item-service

profile: dev

eureka:

client:

service-url:

defaultZone: <http://eureka1:2001/eureka>,

<http://eureka2:2002/eureka>

### 配置刷新

微服务可以在不停机的情况下刷新一些自定义配置,向端点发送post请求微服务会向配置中心拉取新的配置并更新

添加actuator依赖,暴露端点

<dependency>

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

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

</dependency>

management:

endpoints:

web:

exposure:

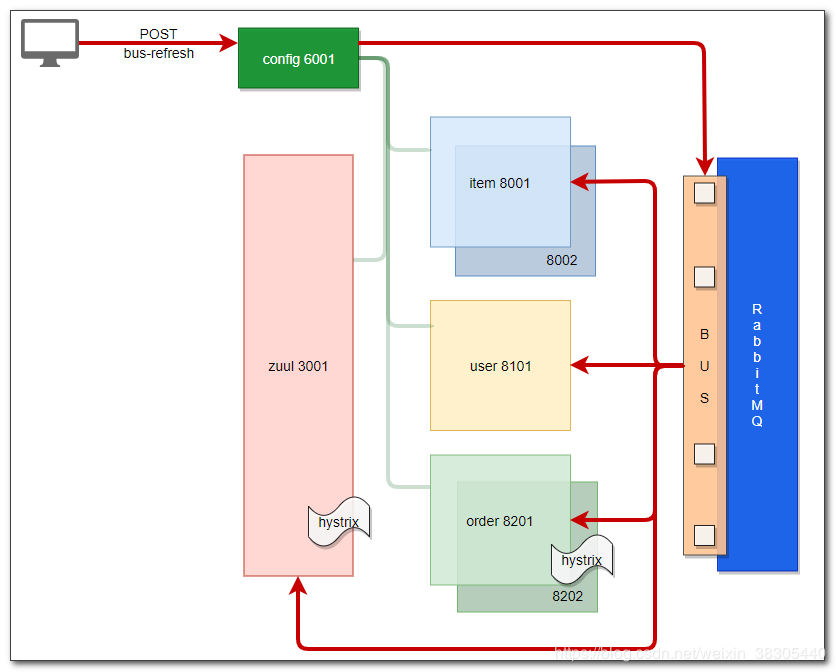
include: refresh

UserServiceImpl 添加 @RefreshScope 注解

只允许对添加了 @RefreshScope 或 @ConfigurationProperties 注解的 Bean 刷新配置，可以将更新的配置数据注入到 Bean 中

使用 postman 向刷新端点发送 post 请求可刷新,postman必须使用一个新的标签

### 消息总线配置刷新



一个微服务接到刷新请求后会向rabbitMQ发送消息,接着转发给所有微服务用于更新

所有参与的微服务要添加spring cloud bus依赖和rabbitmq 连接信息

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-cloud-bus</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-stream-binder-rabbit</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.amqp</groupId>

<artifactId>spring-rabbit-test</artifactId>

<scope>test</scope>

</dependency>

yml配置rabbitMQ连接信息

spring:

......

rabbitmq:

host: 192.168.64.140

port: 5672

username: admin

password: admin

接收第一个刷新请求的微服务要暴露端点,需要actuator依赖

management:

endpoints:

web:

exposure:

include: bus-refresh

向<http://localhost:6001/actuator/bus-refresh>发送post即可刷新

如果刷新指定的微服务，可按下面格式访问：

<http://localhost:6001/actuator/bus-refresh/user-service:8101>

### config 本地文件系统

配置文件也可以存在本地,配置 spring.profiles.active = native 来激活本地文件系统

本地路径默认：[classpath:/, classpath:/config, file:./, file:./config],选用这些路径则无需配置(下面的示例程序还是配置了)

spring:

application:

name: config-server

profiles:

active: native

cloud:

config:

server:

native:

search-locations: classpath:/config

## 链路跟踪

### 原理

四个微服务的控制台日志中，可以看到以下信息：

[服务id,请求id,span id,是否发送到zipkin]

请求id：请求到达第一个微服务时生成一个请求id，该id在调用链路中会一直向后面的微服务传递

span id：链路中每一步微服务调用，都生成一个新的id

[zuul,6c24c0a7a8e7281a,6c24c0a7a8e7281a,false]

[order-service,6c24c0a7a8e7281a,993f53408ab7b6e3,false]

[item-service,6c24c0a7a8e7281a,ce0c820204dbaae1,false]

[user-service,6c24c0a7a8e7281a,fdd1e177f72d667b,false]

### 配置

参加链路跟踪的微服务添加sleuth依赖

<dependency>

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

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

</dependency>

### sleuth + zipkin 链路分析

zipkin 可以收集链路跟踪数据，提供可视化的链路分析,微服务将链路数据发送至rabbitMQ,再转发给zipkin服务器.默认 10% 的链路数据会被发送到 zipkin 服务。可以配置修改链路数据抽样比例

启动 zipkin，连接到 rabbitmq

微服务添加zipkin依赖,如果没有配置过spring cloud bus，还需要添加 rabbitmq 依赖和连接信息

<dependency>

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

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

</dependency>

# 全文检索

## 配置环境

### linux配置jdk

解压jdk,

tar -xvf file.tar

-x解包

-v详情

-f指定名

-c打包

配置环境变量

vim /etc/profile

末尾后面添加

unset i

unset -f pathmunge

JAVA\_HOME=/usr/jdk1.8.0\_212

PATH=$JAVA\_HOME/bin:$PATH

export JAVA\_HOME PATH

source /etc/profile生效

## solr配置

### 安装solr

把 solr-8.1.1.tgz 传到服务器/usr/local 目录并解压缩

cd /usr/local

# 上传 solr-8.1.1.tgz 到 /usr/local 目录

tar -xzf solr-8.1.1.tgz

开放端口,root用户强制启动

# 开放 8983 端口

firewall-cmd --zone=public --add-port=8983/tcp --permanent

firewall-cmd --reload

# 不建议使用管理员启动 solr,加 -force 强制启动

bin/solr start -force

浏览器访问 solr 控制台

http://192.168.2.131:8983

### 创建core

core包含了数个字段,类似java的pojo,对应数据库中一行数据

创建core需要提前创建目录结构

# solr目录/server/solr

# |-pd

# |-conf

# |-data

cd /usr/local/solr-8.1.1

mkdir server/solr/pd

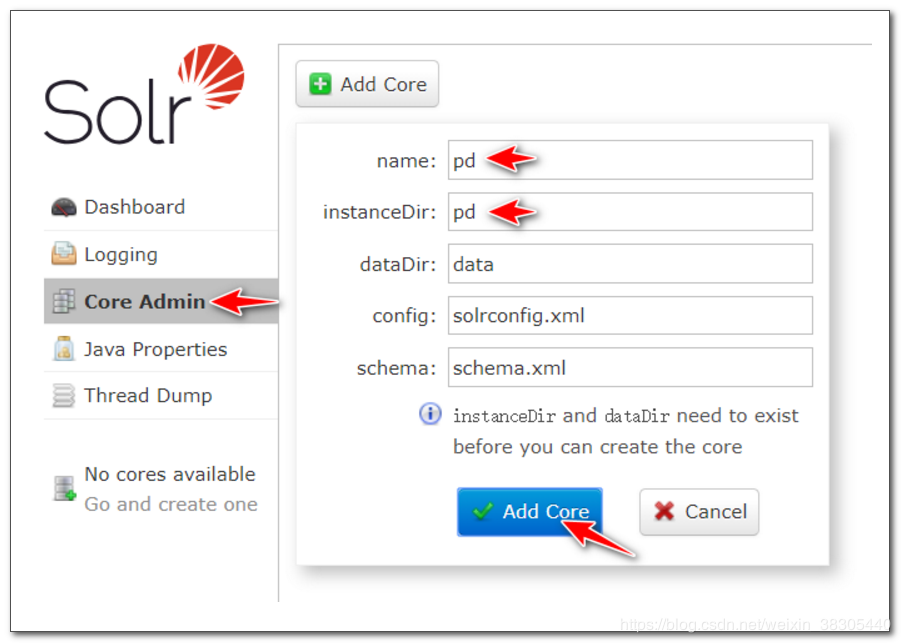
mkdir server/solr/pd/conf

mkdir server/solr/pd/data

conf 目录是 core 的配置目录, 存储一组配置文件,以默认配置为基础, 后续逐步修改,复制默认配置

cd /usr/local/solr-8.1.1

cp -r server/solr/configsets/\_default/conf server/solr/pd



## ik分词器

### ik-analyzer

<https://github.com/magese/ik-analyzer-solr>

下载 ik-analyzer 分词 jar 文件,传到 solr目录/server/solr-webapp/webapp/WEB-INF/lib

为了后续操作方便,把后面用到的jar文件一同传到服务器,包括四个文件:

ik-analyzer-8.1.0.jar

mysql-connector-java-5.1.46.jar

solr-dataimporthandler-8.1.1.jar

solr-dataimporthandler-extras-8.1.1.jar

复制6个文件到solr目录/server/solr-webapp/webapp/WEB-INF/classes

resources/

IKAnalyzer.cfg.xml

ext.dic

stopword.dic

stopwords.txt

ik.conf

dynamicdic.txt

# classes目录如果不存在,需要创建该目录

mkdir /usr/local/solr-8.1.1/server/solr-webapp/webapp/WEB-INF/classes

修改 solr目录/server/solr/pd/conf/managed-schema,添加 ik-analyzer 分词器(添加到末尾标签结束前)

<!-- ik分词器 -->

<fieldType name="text\_ik" class="solr.TextField">

<analyzer type="index">

<tokenizer class="org.wltea.analyzer.lucene.IKTokenizerFactory" useSmart="false" conf="ik.conf"/>

<filter class="solr.LowerCaseFilterFactory"/>

</analyzer>

<analyzer type="query">

<tokenizer class="org.wltea.analyzer.lucene.IKTokenizerFactory" useSmart="true" conf="ik.conf"/>

<filter class="solr.LowerCaseFilterFactory"/>

</analyzer>

</fieldType>

重启 solr 服务

cd /usr/local/solr-8.1.1

bin/solr restart -force

使用 text\_ik 分词器

### 停止词

设置停止词

上传停止词配置文件到 solr目录/server/solr-webapp/webapp/WEB-INF/classes

stopword.dic

stopwords.txt

重启服务,观察分词结果中,停止词被忽略

bin/solr restart -force

## mysql 数据库数据

### 跨网络访问权限

授予 root 用户 跨网络访问权限

grant all on \*.\* to 'root'@'%' identified by 'root';

授权 所有权限 在 所有库.所有表 给 root用户@任何ip地址 登录密码是 root

## solr使用

### 设置字段

和数据库里面的表字段一样,注意格式

title text\_ik

sellPoint text\_ik

price plong

barcode string

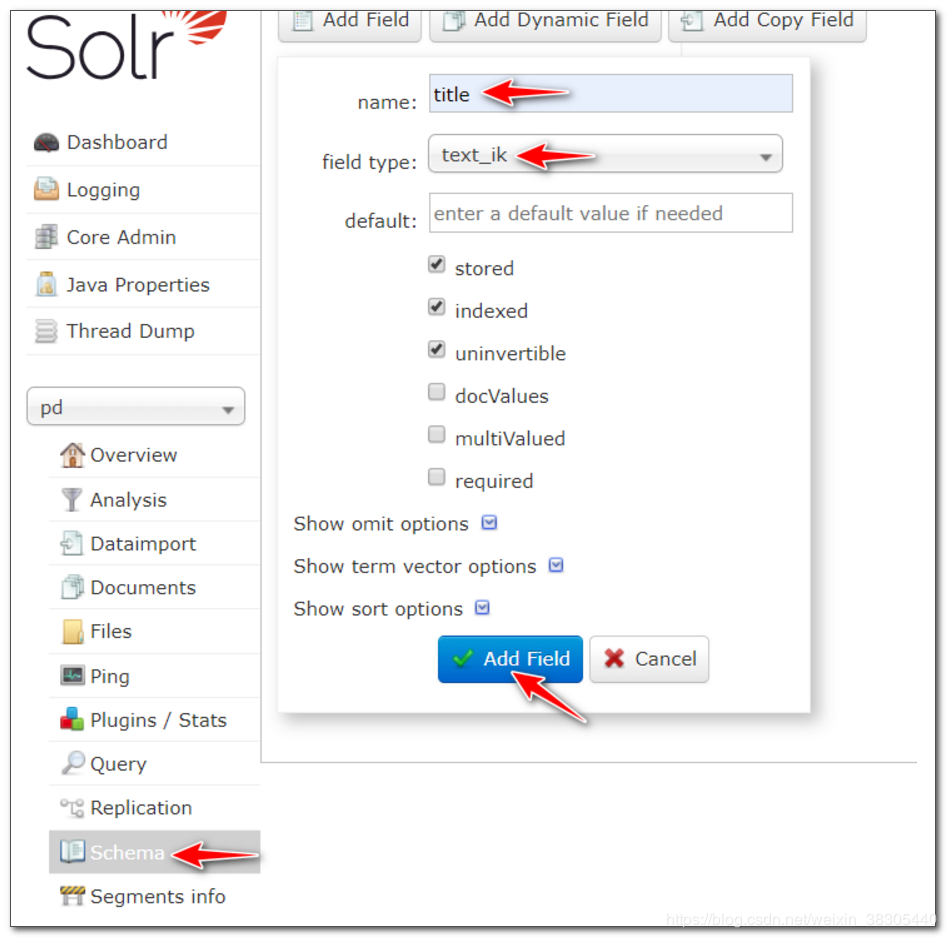
image string

cid plong

status pint

created pdate

updated pdate



可以在solr目录/server/solr/pd/conf/managed-schema的末尾看到一堆field



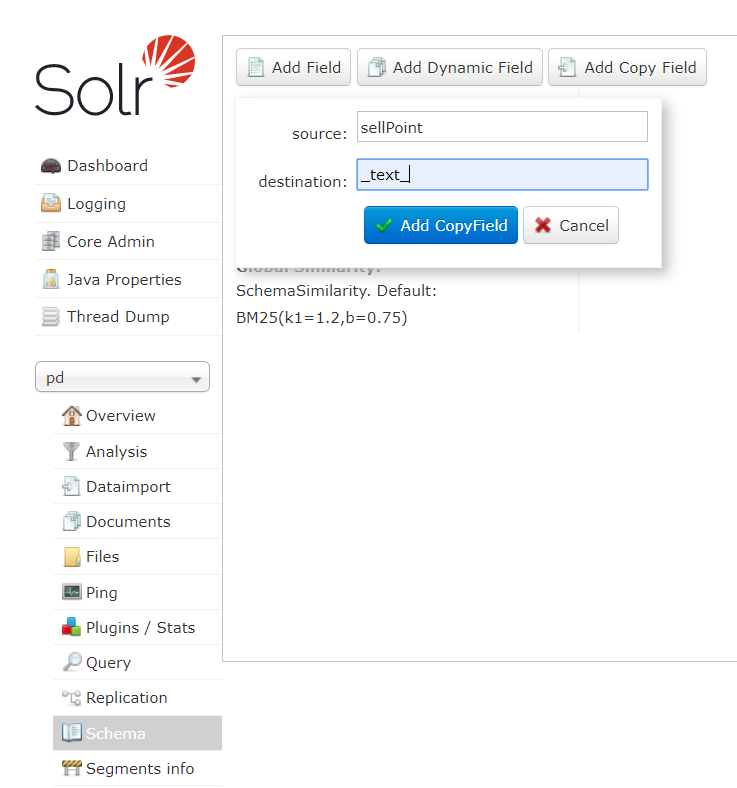
### 默认字段

查询时需要按字段查询,例如 title:电脑,什么字段都不写会使用默认字段进行查询,默认字段的指定设置在solr目录/server/solr/pd/confsolrconfig.xml中定义,具体设置仍然在managed-schema中



### Copy Field 副本字段

可以将多个字段的值合并到一个字段进行查询,默认查询字段 \_text\_,将 title 和 sellPoint 复制到 \_text\_ 字段,就可以一起搜索标题和卖点



### Data Import Handler 配置

Data Import Handler 的 jar 文件存放在 solr目录/dist 目录下(也可以用本地备份)

solr-dataimporthandler-8.1.1.jar

solr-dataimporthandler-extras-8.1.1.jar

复制这两个文件和 mysql 的 jar 文件(mysql-connector-java-5.1.46.jar)到 solr目录/server/solr-webapp/webapp/WEB-INF/lib

### 导入数据

dih-config.xml修改 mysql 的 ip 地址,传到solr目录/server/solr/pd/conf

<dataConfig>

<dataSource type="JdbcDataSource"

driver="com.mysql.jdbc.Driver"

url="jdbc:mysql://192.168.2.1:3306/pd\_store"

user="root"

password="root"

batchSize="100"

autoCommit="false" />

<document name="item">

<entity name="item" pk="id"

query="SELECT id,title,sell\_point sellPoint,price,barcode,image,cid,`status`,created,updated FROM pd\_item"

deltaQuery="SELECT id,title,sell\_point sellPoint,price,barcode,image,cid,`status`,created,updated FROM pd\_item WHERE updated &gt; date\_add(str\_to\_date('${dih.last\_index\_time}','%Y-%m-%d %H:%i:%s'),interval 8 hour)"

transformer="RegexTransformer">

</entity>

</document>

</dataConfig>

url用虚拟网络拥有数据库的主机ip,最后一位默认填1,query指定从数据库查询什么, deltaQuery设置每8小时查一次

solrconfig.xml 中添加 DIH 配置,名字就是刚才文件的名字,添加到末尾标签结束前

<requestHandler name="/dataimport" class="org.apache.solr.handler.dataimport.DataImportHandler">

<lst name="defaults">

<str name="config">dih-config.xml</str>

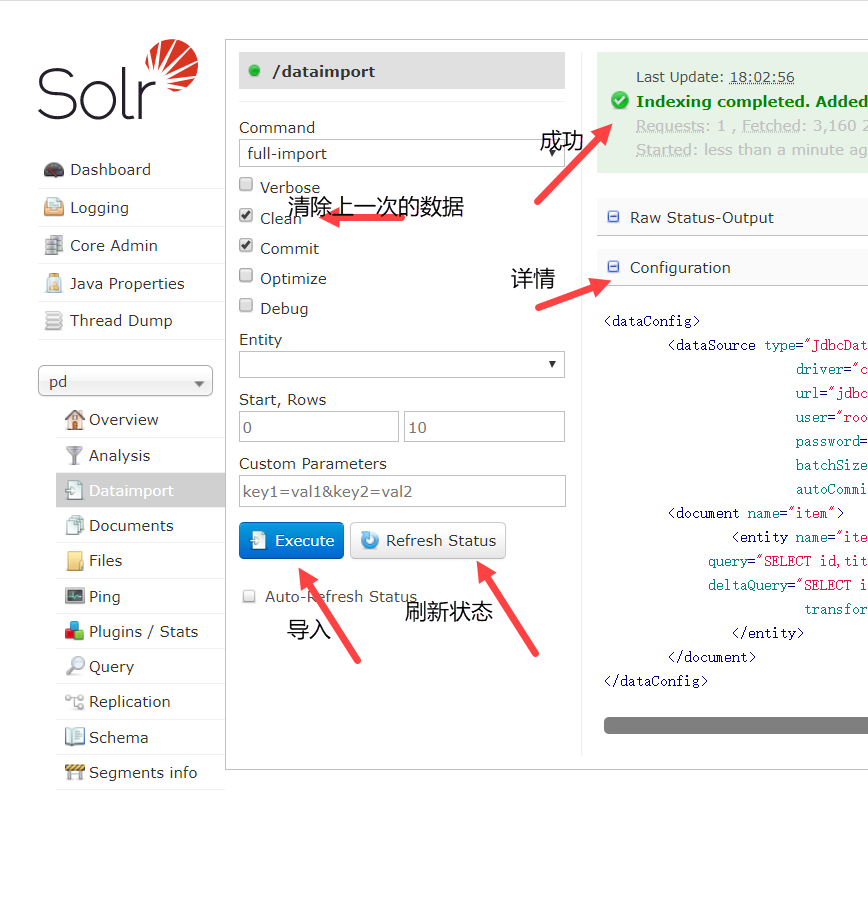
</lst>

</requestHandler>

重启solr

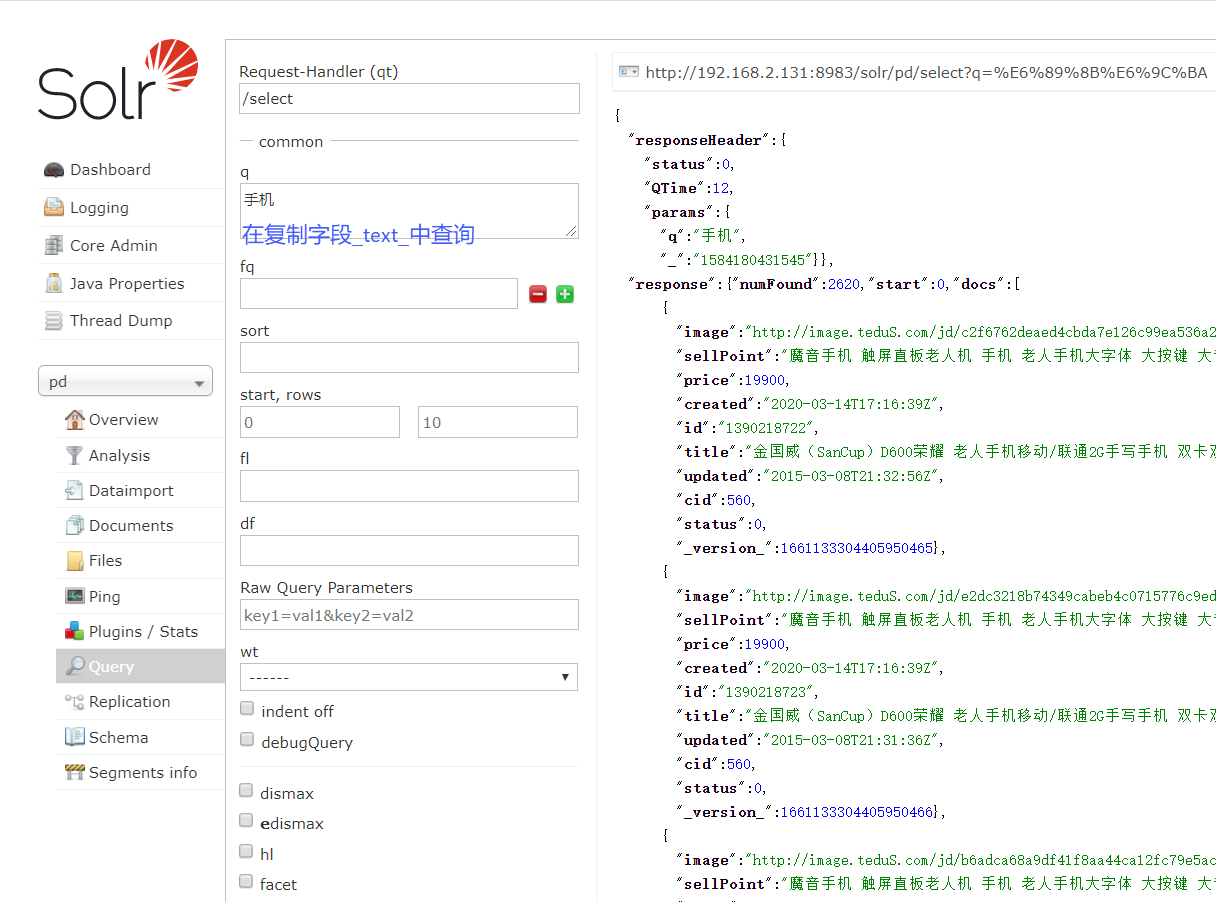
cd /usr/local/solr-8.1.1

bin/solr restart -force

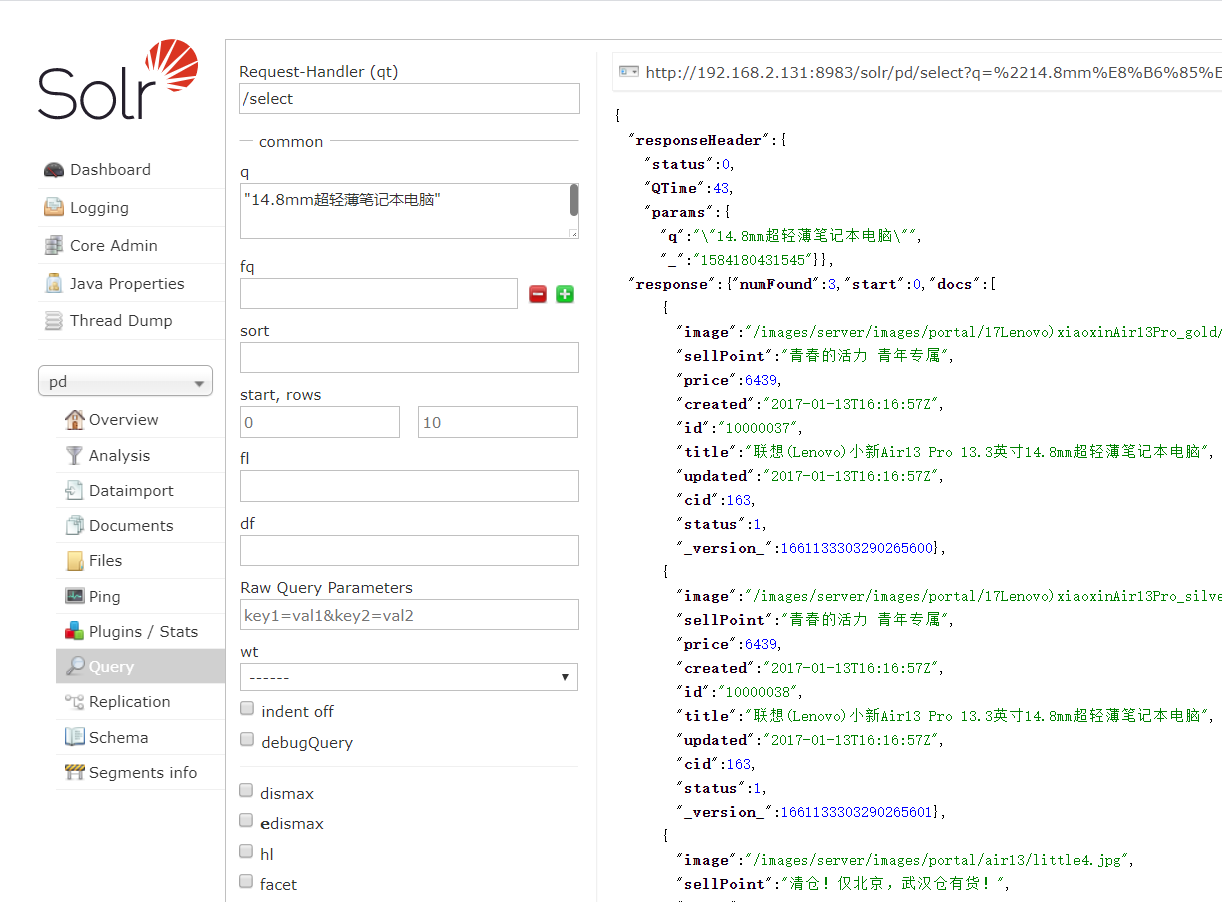


### 查询

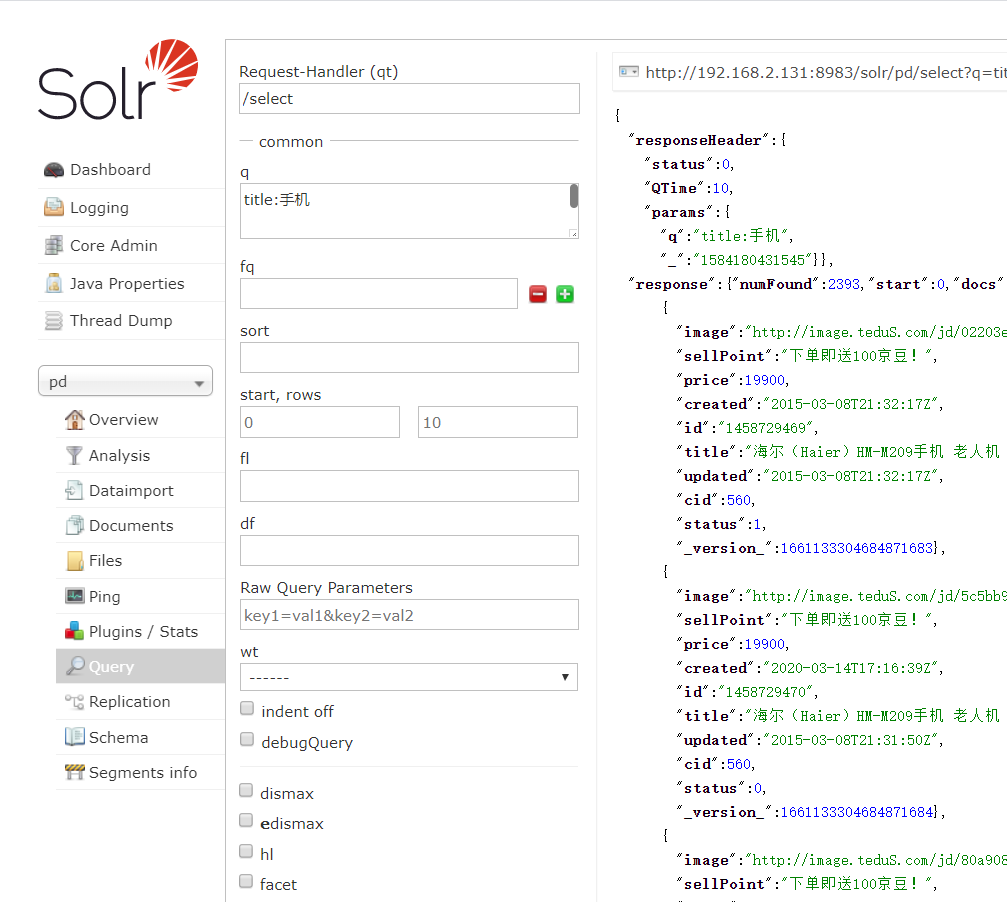
* 直接输入在默认字段\_text\_中查询,结合了标题和卖点



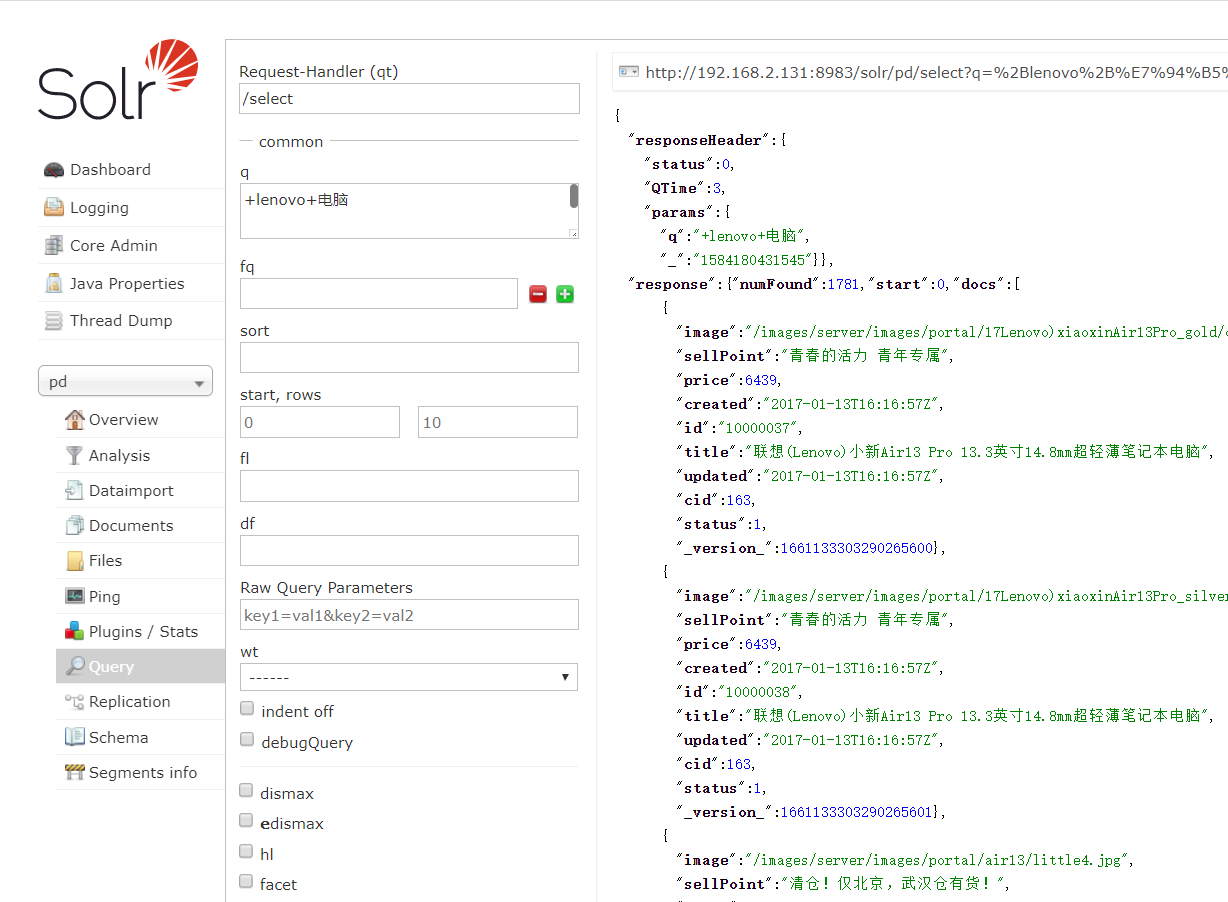
* 加“”查询全文



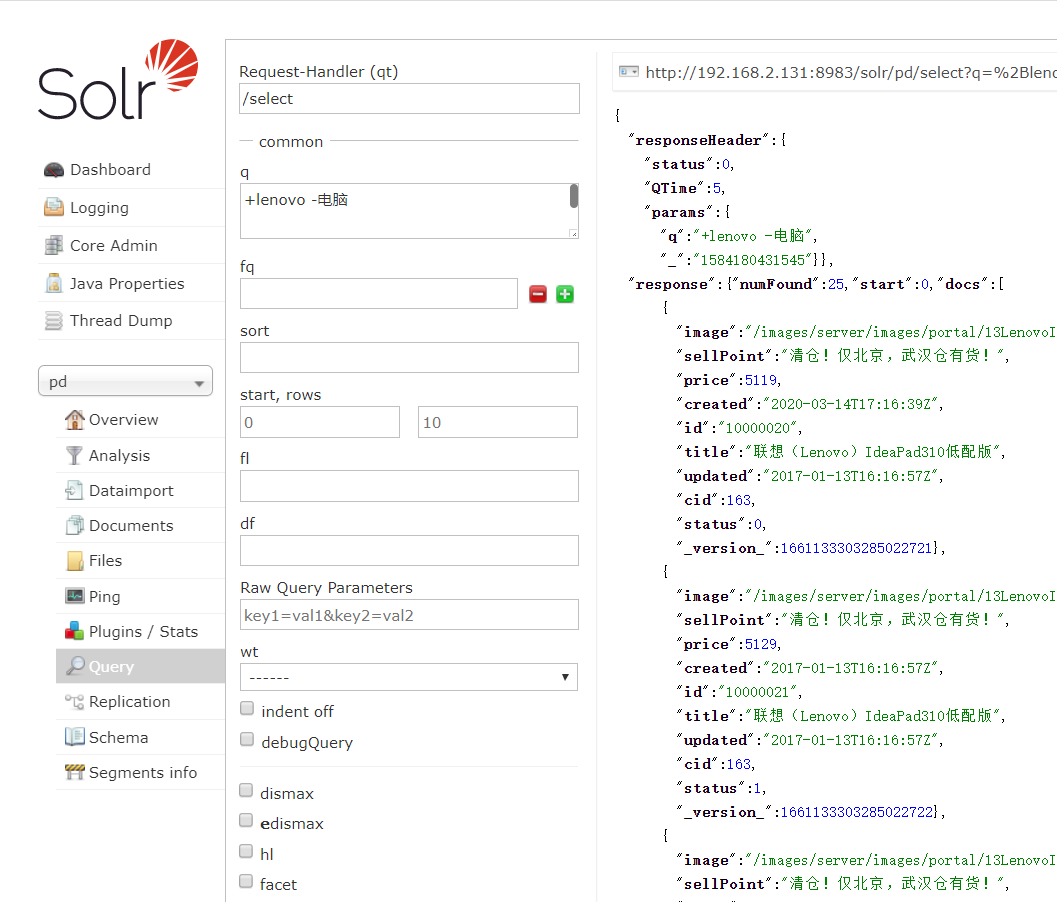
* 查询标题



* 多关键字同时出现



* 多关键字不出现



* 商品分类统计,搜索\*:\*全部

|  |  |
| --- | --- |
|  |  |

* 输入facet.mincount=100 只返回大于100条的数据

|  |  |
| --- | --- |
|  |  |

* 价格范围

在 Raw Query Parameters 输入框中填入以下内容:

facet.range=price&facet.range.start=0&facet.range.end=10000&facet.range.gap=2000

价格从0到10000每2000是一档

|  |  |
| --- | --- |
|  |  |

* 多字段查询,查询每个分类有多少件商品上架和下架

在 Raw Query Parameters 输入框中填入以下内容:

facet.pivot=cid,status

|  |  |
| --- | --- |
|  |  |

# Docker

hub.docker.com

下载镜像的网址

## 命令

### 查看镜像

docker image ls

### 查看容器

查看运行中的容器

docker ps

查看所有容器

docker ps -a

### 删除容器

docker container rm 容器编号的前三位

docker container rm -f 54b 899 5bd 6d3

### 运行容器

docker run 容器名称

docker run 容器名称:容器版本

-d 后台运行容器

-i 交互式操作

-t 终端。

docker run -it centos:7 bash

docker run -dit centos:7 bash

exit

### 进入容器

docker exec -it 54be8 bash

### 挂载主机目录

将一个宿主机的本地目录挂载到容器内,形成映射,容器的该路径数据会保存在外部路径中,删除容器时数据不会删除

命令格式 外部路径:容器路径

docker run -it -v /usr/app:/opt/app centos:7 bash

## 数据卷

### 定义

* 数据卷 是一个可供一个或多个容器使用的特殊目录
* 数据卷 可以在容器之间共享和重用
* 数据卷 默认会一直存在，即使容器被删除
  + 对数据卷的修改会立刻生效
  + 对数据卷的更新，不会影响镜像

### 创建数据卷

docker volume create my-vol

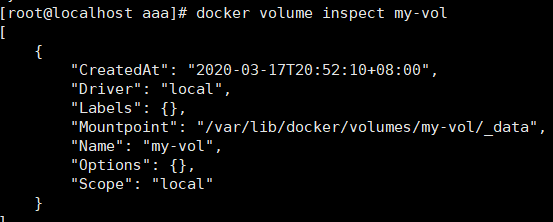
### 查看数据卷信息

查看所有数据卷

docker volume ls

查看指定数据卷

docker volume inspect my-vol



### 挂载卷

-v 卷名:容器目录

docker run -it -v my-vol:/opt/app centos:7 bash

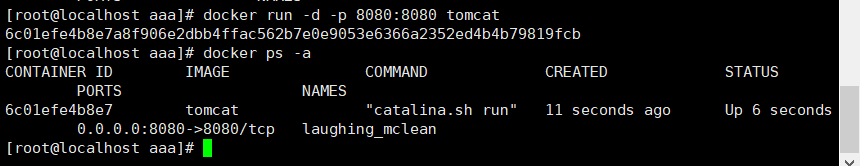
## 端口映射

容器的端口要映射到宿主机的端口上对外通信

### 映射指定端口

-p 本机端口:容器端口

docker run -d -p 8080:8080 tomcat



## 容器互联

同一宿主机下可以建立虚拟网络使容器相互连接,容器会被自动分配虚拟网络的地址

### 新建网络

新建网络

-d 指定网络类型 默认是桥接,可以选择overlay（Swarm mode）

docker network create -d bridge my-net

docker network create --subnet=172.18.0.0/24 dockernet

ifconfig

### 列出网络

docker network ls

### 查看网络信息

docker inspect 67d

### 连接容器

docker run -it --name app1 --network my-

net centos:7

docker run -it --name app2 --network my-net centos:7

可以使用ping app1

### 删除网络

docker network rm my-net

## 构建镜像

### Dockerfile

FROM centos:7

ADD jdk-8u212-linux-x64.tar.gz /opt/

ADD apache-tomcat-7.0.96.tar.gz /usr/

RUN mv /usr/apache-tomcat-7.0.96 /usr/tomcat

ENV JAVA\_HOME=/opt/jdk1.8.0\_212 \

CATALINA\_HOME=/usr/tomcat \

PATH=$PATH:/opt/jdk1.8.0\_212/bin:/usr/tomcat/bin

EXPOSE 8080

ENTRYPOINT /usr/tomcat/bin/startup.sh && tail -F /usr/tomcat/logs/catalina.out

### 构建指令

docker build -t tomcat:85 ./

## 搭建完整项目环境

### 加载 mariadb 镜像

docker load < mariadb-docker-image.gz

### 创建数据卷

docker volume create mysql-data

docker run -d --name mysql --net dockernet --ip 172.18.0.11 -v mysql-data:/var/lib/mysql -e MYSQL\_ROOT\_PASSWORD=root -p 3306:3306 mariadb

### 启动数据库

-d 后台运行

--name 指定容器名称

--net 连接的虚拟网络

--ip 为自己指定ip

-v 挂载卷

-e 指定密码

-p 映射端口

启动 mariadb 容器,并挂载数据卷

docker run -d --name mysql --net dockernet --ip 172.18.0.11 -v mysql-data:/var/lib/mysql -e MYSQL\_ROOT\_PASSWORD=root -p 3306:3306 mariadb

docker ps -a

docker inspect mysql

### 启动tomcat

docker run -d --name tomcat --net dockernet --ip 172.18.0.12 -v /opt/webapps:/usr/tomcat/webapps -p 8080:8080 tomcat:7

连接池中配置了数据库的ip地址

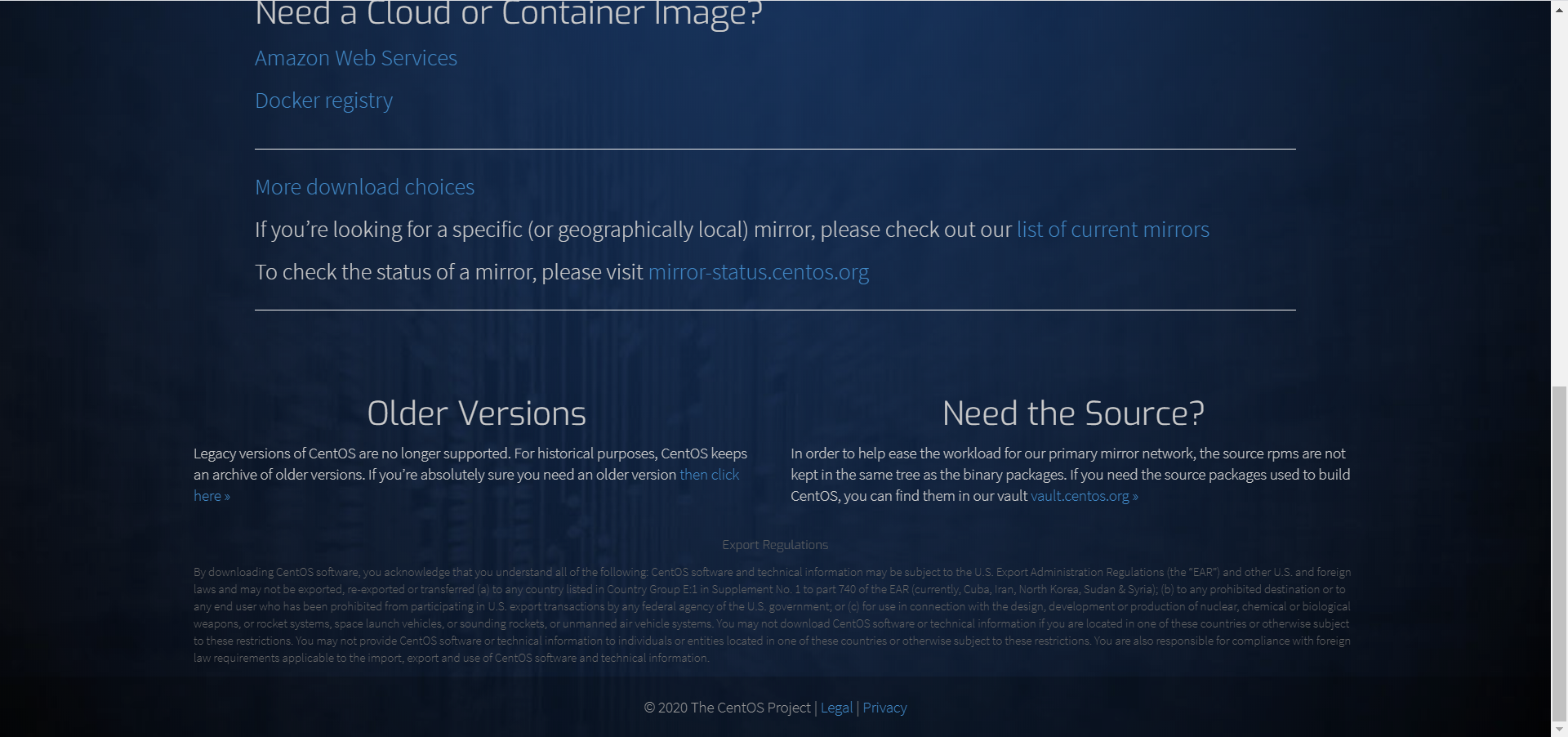
# kubernetes

### 下载centos

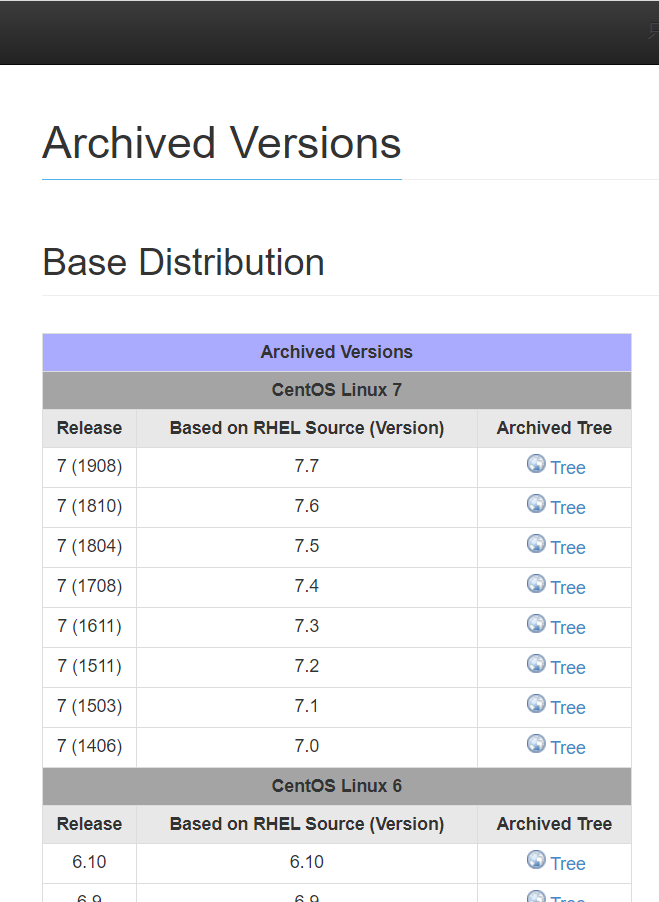
最新版本下载

访问centos 官网 https://www.centos.org/， 点击 Get CentOS Now,选择镜像

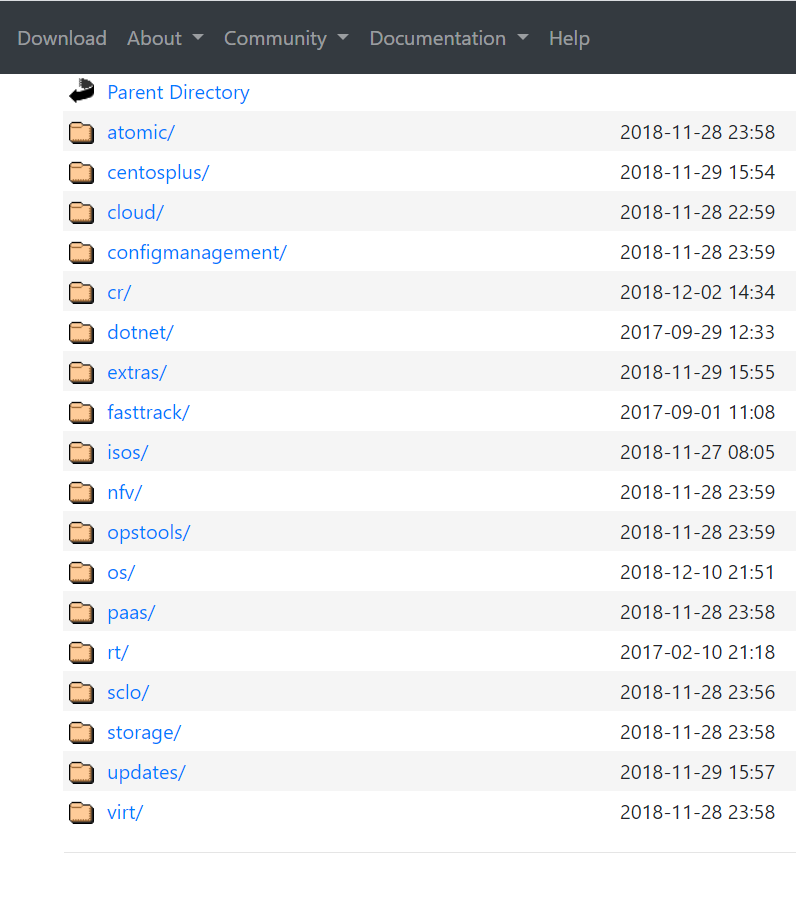
历史版本下载, https://www.centos.org/download/,Older Versions, then click here »



Archived Versions,Base Distribution,选择版本,tree



isos





CentOS-7-x86\_64-DVD-1804.iso 标准安装版，一般下载这个就可以了（推荐）

CentOS-7-x86\_64-NetInstall-1804.iso 网络安装镜像（从网络安装或者救援系统）

CentOS-7-x86\_64-Everything-1804.iso 对完整版安装盘的软件进行补充，集成所有软件。（包含centos7的一套完整的软件包，可以用来安装系统或者填充本地镜像）

CentOS-7-x86\_64-LiveGNOME-1804.iso GNOME桌面版

CentOS-7-x86\_64-LiveKDE-1804.iso KDE桌面版

CentOS-7-x86\_64-Minimal-1804.iso  精简版，自带的软件最少

### linux设置静态ip

cd /etc/sysconfig/network-scripts/

vim ifcfg-ens33

bootproto改为static

BOOTPROTO="static"

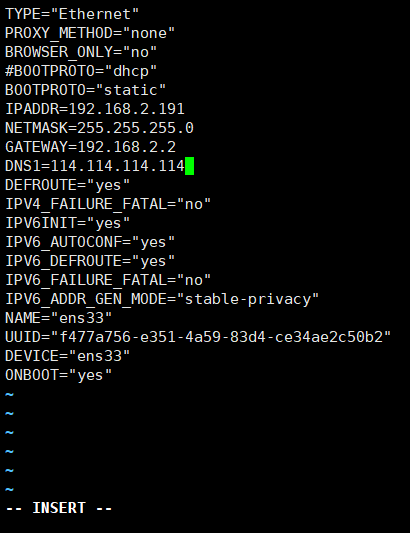
IPADDR=192.168.2.191

NETMASK=255.255.255.0

GATEWAY=192.168.2.2

DNS1=114.114.114.114

service network restart



### 准备离线安装环境

在CentOS7虚拟机中执行下面操作

cd ~/

# 下载 kubeasz 的自动化安装脚本文件: easzup,如果已经上传过此文件,则不必执行这一步

export release=2.0.3

curl -C- -fLO --retry 3 https://github.com/easzlab/kubeasz/releases/download/${release}/easzup

# 对easzup文件设置执行权限

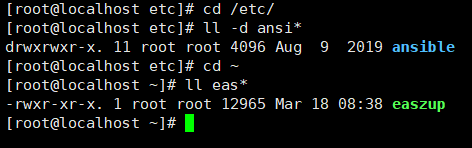
chmod +x ./easzup

# 下载离线安装文件,并安装配置docker,

# 如果离线文件已经存在则不会重复下载,

# 离线安装文件存放路径: /etc/ansible

./easzup -D



# 启动kubeasz工具使用的临时容器

./easzup -S

# 进入该容器

docker exec -it kubeasz sh

# 下面命令在容器内执行

# 配置离线安装

cd /etc/ansible

sed -i 's/^INSTALL\_SOURCE.\*$/INSTALL\_SOURCE: "offline"/g' roles/chrony/defaults/main.yml

sed -i 's/^INSTALL\_SOURCE.\*$/INSTALL\_SOURCE: "offline"/g' roles/ex-lb/defaults/main.yml

sed -i 's/^INSTALL\_SOURCE.\*$/INSTALL\_SOURCE: "offline"/g' roles/kube-node/defaults/main.yml

sed -i 's/^INSTALL\_SOURCE.\*$/INSTALL\_SOURCE: "offline"/g' roles/prepare/defaults/main.yml

exit

# 安装 python,已安装则忽略这一步

yum install python -y

### 在master上继续配置安装环境

# 安装pip,已安装则忽略这一步

wget -O /etc/yum.repos.d/epel-7.repo https://mirrors.aliyun.com/repo/epel-7.repo

yum install git python-pip -y

# pip安装ansible(国内如果安装太慢可以直接用pip阿里云加速),已安装则忽略这一步

pip install pip --upgrade -i https://mirrors.aliyun.com/pypi/simple/

pip install ansible==2.6.12 netaddr==0.7.19 -i https://mirrors.aliyun.com/pypi/simple/

# 在ansible控制端配置免密码登陆其他节点服务器

ssh-keygen -t ed25519 -N '' -f ~/.ssh/id\_ed25519

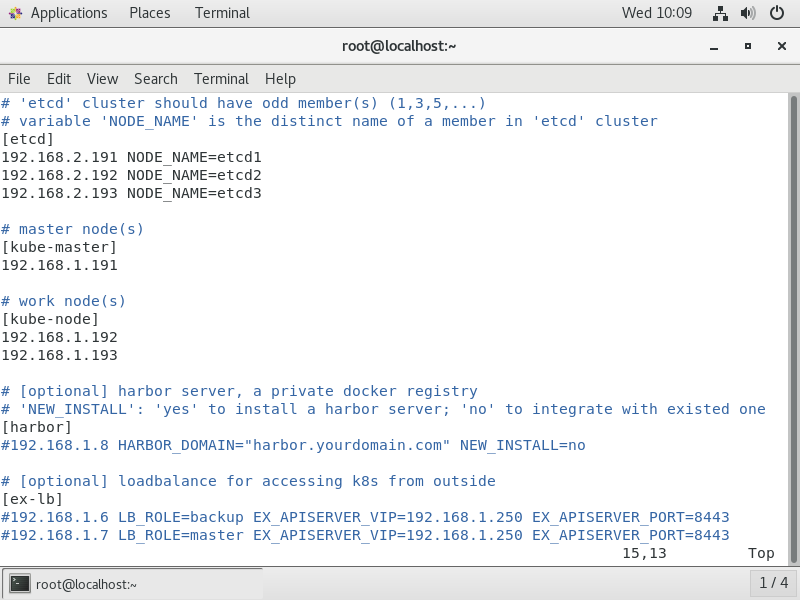
# 公钥复制到所有节点,包括master自己

# 按提示输入yes和root管理员的密码

ssh-copy-id 192.168.2.191

ssh-copy-id 192.168.2.192

ssh-copy-id 192.168.2.193



### 使用python3.7

#安装相应的编译工具

yum -y groupinstall "Development tools"

yum -y install zlib-devel bzip2-devel openssl-devel ncurses-devel sqlite-devel readline-devel tk-devel gdbm-devel db4-devel libpcap-devel xz-devel

yum install -y libffi-devel zlib1g-dev

yum install zlib\* -y

# 下载

wget https://www.python.org/ftp/python/3.7.3/Python-3.7.3.tgz

# 解压

tar -zxf Python-3.7.3.tgz

# 安装依赖包

yum install zlib-devel bzip2-devel openssl-devel ncurses-devel sqlite-devel readline-devel tk-devel gcc libffi-devel

# 进入python目录

cd Python-3.7.3

# 编译

./configure --prefix=/usr/local/python3.7 --with-ssl

#安装

make && make install

关于Python3.7以上的版本，需要多安装一个依赖包:

yum install -y libffi-devel

否则会出现ModuleNotFoundError: No module named '\_ctypes'的报错。

在make install后执行

echo $?

为0表示没有出错。如果没有报错，在/usr/local会生成python3.7目录。

然后将系统默认的python2备份

mv /usr/bin/python /usr/bin/python.bak

创建新的软连接

ln -s /usr/local/python/bin/python3.7 /usr/bin/python

### 更改yum配置

因为yum需要使用python2，将/usr/bin/python改为python3后，yum就不能正常运行了，因此需要更改一下yum的配置。

vim /usr/bin/yum

vim /usr/libexec/urlgrabber-ext-down

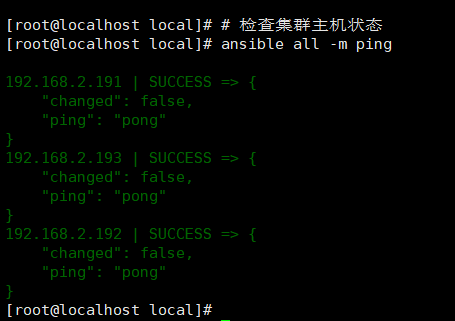
编辑这两个文件，将文件头的#!/usr/bin/python改为#!/usr/bin/python2即可。

### 添加pip到path

echo 'export PATH=/usr/local/python3.7/bin:$PATH' >>~/.bashrc

source ~/.bashrc

/usr/local/python3.7/bin



### 错误

fatal: [192.168.2.191]: FAILED! => {"changed": true, "cmd": "systemctl daemon-reload && systemctl restart etcd", "delta": "0:00:00.207916", "end": "2020-03-19 03:09:19.262195", "msg": "non-zero return code", "rc": 1, "start": "2020-03-19 03:09:19.054279", "stderr": "Job for etcd.service failed because the control process exited with error code. See \"systemctl status etcd.service\" and \"journalctl -xe\" for details.", "stderr\_lines": ["Job for etcd.service failed because the control process exited with error code. See \"systemctl status etcd.service\" and \"journalctl -xe\" for details."], "stdout": "", "stdout\_lines": []}

# SSO单点登录