

# Spring Cloud

## 黑马篇

简介

认识微服务

远程调用

  Hello World

  Eureka

    环境搭建

    实现负载均衡

  Ribbon负载均衡

    饥饿加载

  Nacos

    分级存储模型

  NACOS统一配置管理

    多环境共享

  Nacos集群配置

  Feign

    性能优化

      最佳实践

  统一网关GateWay

    全局过滤器GlobalFilter

    跨域问题处理

  Docker

    Hello World

    常用命令

    数据卷

    Dockerfile 自定义镜像

    DockerCompose

    自定义镜像仓库

  MQ

    RabbitMQ安装

  ES

    数据类型

    索引【index】

    文档【\_doc】

    RestClient

      索引

      文档

      批量增加

      高级查询

      简单查询

- 算法排序
- 布尔查询
- 搜索结果处理
- RestClient操作
  - 结果处理
- ES 高级部分
  - 数据聚合
  - DSL
  - RestClient
- 拼音分词器
  - 安装
  - 使用
- 自动补全
- 案例索引映射
- 数据同步
- 提出问题与解决方案

集群

Sentinel

- 介绍与安装
- 介绍
- 依赖与配置
- 限流规则
  - 防止一直点
- 隔离和降级
  - 整合
  - 线程隔离
- 熔断降级
- 授权规则
- 规则持久化

分布式事务【基于seata】

- 理论基础
- seata
- 注册
- XA模式
- AT模式
- TCC
- saga
- 对比

分布式缓存

- RDB和AOF
- 主从架构
- 哨兵
- Redis分片集群

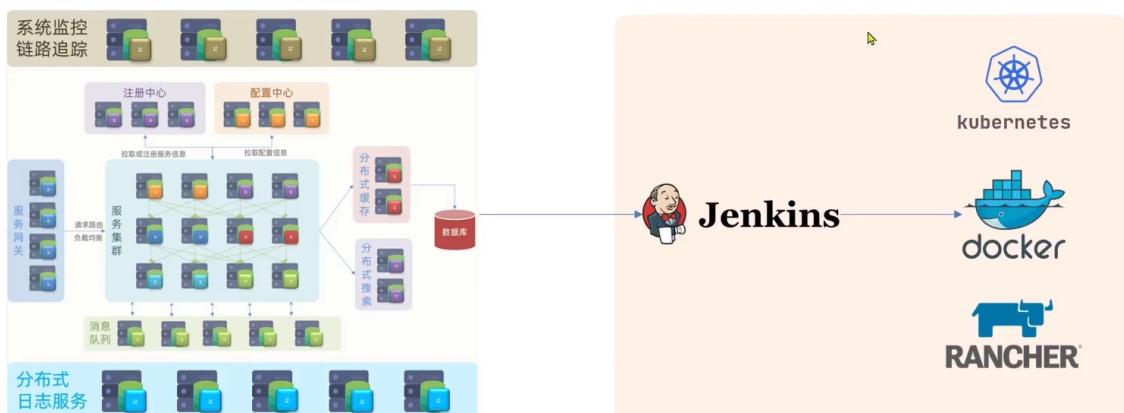
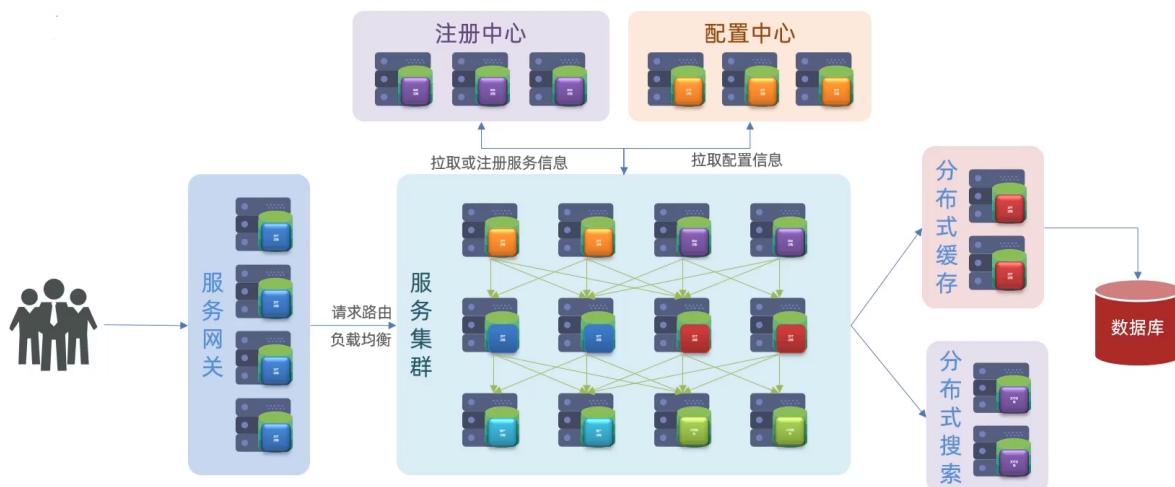
多级缓存

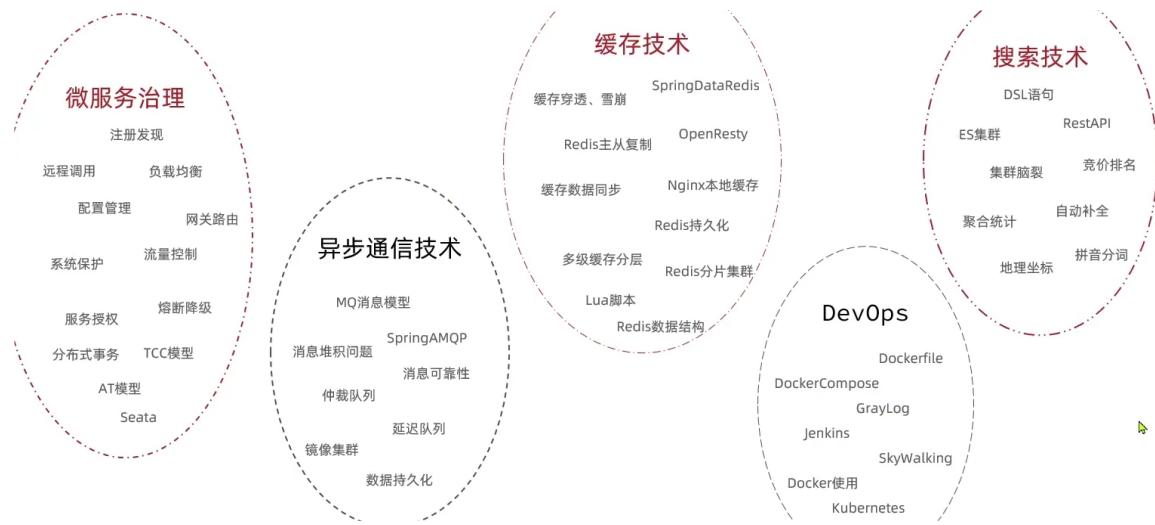
- 进程缓存

Lua  
OpenResty  
Canal  
  
MQ  
生产者消息确认  
持久化  
消费者消息确认  
失败重试机制  
死信交换机  
TTL  
延迟队列  
消息堆积问题

## 黑马篇

# 简介





# 认识微服务

简单来说就是单体应用耦合度高，低代码量的时候没有影响，当代码量组件上升成一个大型的缺点就显现出来，不便于维护

而微服务架构就有所缓解，根据业务将各个模块进行拆分，每一个业务模块作为独立开发

- 降低耦合度
- 有利于提升扩展性

那么服务之间作为单独的模块将如何进行调用呢？怎么知道你挂没挂呢？

- 不同微服务，不要重复开发相同业务
- 微服务数据独立，不要访问其他服务的数据库
- 微服务可以将自己业务暴露为接口，供其他服务使用

# 远程调用

## Hello World

```
@Autowired
private OrderService orderService;
```

```
private final static String USERHOST =
"http://localhost:8081/user/";

@Autowired //需要优先注册到容器中
private RestTemplate restTemplate;

@GetMapping("{orderId}")
public Order
queryOrderByUserId(@PathVariable("orderId") Long orderId) {
    // 查询到 Order 对象
    Order order =
orderService.queryOrderById(orderId);
    // 根据Order对象的ID获取user对象
    User user = restTemplate.getForObject(USERHOST
+ order.getUserId(), User.class);
    System.out.println(user);
    return order;
}
```

## Eureka

EurekaServer：服务注册中心，会记录服务信息，监控心跳【30秒一次】

EurekaClient：注册在Eureka的服务都称之为客户端

消费者：根据服务名称在EurekaServer中拉取服务列表，基于负载均衡选取一个

提供者：注册信息到EurekaServer，每隔30秒发送心跳续约

## 环境搭建

### 父工程

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0
.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<parent>
    <artifactId>cloud-demo</artifactId>
    <groupId>cn.itcast.demo</groupId>
    <version>1.0</version>
</parent>
<modelVersion>4.0.0</modelVersion>

<artifactId>order-service</artifactId>

<dependencies>
    <dependency>

<groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-
web</artifactId>
        </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-
java</artifactId>
        </dependency>
    <!--mybatis-->
    <dependency>

<groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-
starter</artifactId>
        </dependency>

    </dependencies>
    <build>
        <plugins>
            <plugin>

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

```
<artifactId>spring-boot-maven-
plugin</artifactId>
    </plugin>
</plugins>
</build>
</project>
```

新建子工程引入

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

boot主启动类添加 `EnableEurekaServer` 开启服务注册中心

配置application.yaml

```
eureka:
  client: # eureka的地址信息
    service-url: # 如果是集群 多个之间逗号隔开
      defaultZone: http://127.0.0.1:10086/eureka
```

客户端的pom文件

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

客户端的配置文件暂时和上面一样

启动服务

打开服务端【eurekaServer】的端口

## 当前在 Eureka 注册的实例

应用	朋友们	可用区	地位
EUREKASERVER	不适用 (1)	(1)	上 (1) - 192.168.159.1:eurekaserver:10086
订购服务	不适用 (1)	(1)	上 (1) - 192.168.159.1: 订购服务: 8080
用户服务	不适用 (1)	(1)	上 (1) - 192.168.159.1: 用户服务: 8081

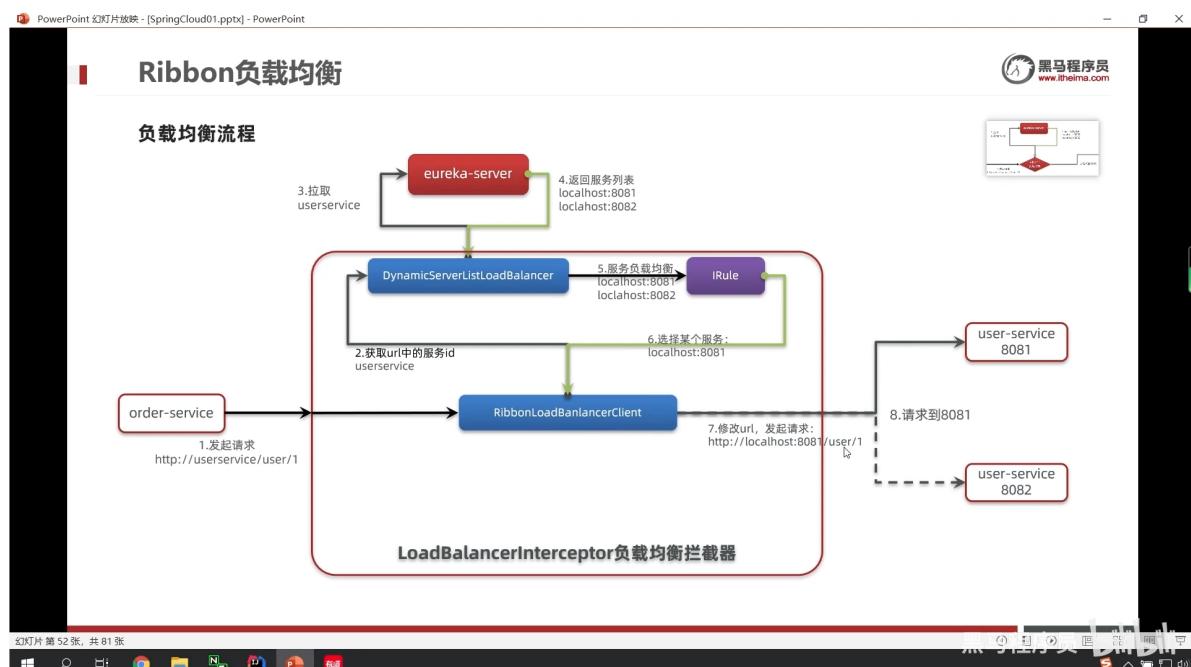
## 实现负载均衡

消费者方

```
// 修改为在注册中心的实例名称  
private final static String USERHOST =  
"http://userservice/user/";  
  
@Bean  
@LoadBalanced // 配合注册中心开启负载均衡  
public RestTemplate restTemplate(){  
    return new RestTemplate();  
}
```

此时用IDEA执行 `-Dserver.port=端口号` 同时执行两个一样的服务，便可以直观的在控制台发现一人一次的 负载均衡 效果

## Ribbon负载均衡



内置负载均衡规则类	规则描述
RoundRobinRule	简单轮询服务列表来选择服务器。它是Ribbon默认的负载均衡规则。
AvailabilityFilteringRule	对以下两种服务器进行忽略： (1) 在默认情况下，这台服务器如果3次连接失败，这台服务器就会被设置为“短路”状态。短路状态将持续30秒，如果再次连接失败，短路的持续时间就会几何级地增加。 (2) 并发数过高的服务器。如果一个服务器的并发连接数过高，配置了AvailabilityFilteringRule规则的客户端也会将其忽略。并发连接数的上限，可以由客户端的<clientName><clientConfigNameSpace>.ActiveConnectionsLimit属性进行配置。
WeightedResponseTimeRule	为每一个服务器赋予一个权重值。服务器响应时间越长，这个服务器的权重就越小。这个规则会随机选择服务器，这个权重值会影响服务器的选择。
ZoneAvoidanceRule	以区域可用的服务器为基础进行服务器的选择。使用Zone对服务器进行分类，这个Zone可以理解为一个机房、一个机架等。而后再对Zone内的多个服务做轮询。
BestAvailableRule	忽略哪些短路的服务器，并选择并发数较低的服务器。
RandomRule	随机选择一个可用的服务器。
RetryRule	重试机制的选择逻辑

## 调整负载均衡制度的方式

### 1 配置bean注入容器 【在消费者】

```
@Bean  
public IRule getIRule(){  
    return new RandomRule(); // 随机  
}
```

### 2 在消费者的application配置文件中

```
userservice: # 注意 这一同一个服务的实例名  
ribbon: # 随机  
    NFLoadBalancerRuleClassName:  
com.netflix.loadbalancer.RandomRule
```

## 饥饿加载

Ribbon默认采用懒加载，第一次访问时才会取创建LoadBalanceClient，请求时间会很长。而饥饿加载则会在项目启动时创建，降低第一次访问时的耗时，通过配置开启饥饿加载

```
ribbon:  
    eager-load:  
        clients: userservice # 服务实例名  
        enabled: true  
# 以上就是对一个实例 不同的服务开启饥饿加载 -clients是一个List
```

## Nacos

```
startup.cmd -m standalone // 1版本可用 2版本要集群和  
MySQL  
# Linux  
.bin/startup.sh -m standalone
```

父工程添加

```
<dependency>  
    <groupId>com.alibaba.cloud</groupId>  
    <artifactId>spring-cloud-alibaba-  
dependencies</artifactId>  
    <version>2.2.5.RELEASE</version>  
    <type>pom</type>  
    <scope>import</scope>  
</dependency>
```

除去子模块的eureka依赖,然后添加

```
←!—服务注册/发现中心依赖→  
<dependency>  
    <groupId>com.alibaba.cloud</groupId>  
    <artifactId>spring-cloud-starter-alibaba-nacos-  
discovery</artifactId>  
</dependency>
```

配置文件

```
spring:  
  cloud:  
    nacos:  
      discovery:  
        cluster-name: localhost:8848
```

@LoadBalanced 可以和其配合负载均衡的

## 分级存储模型

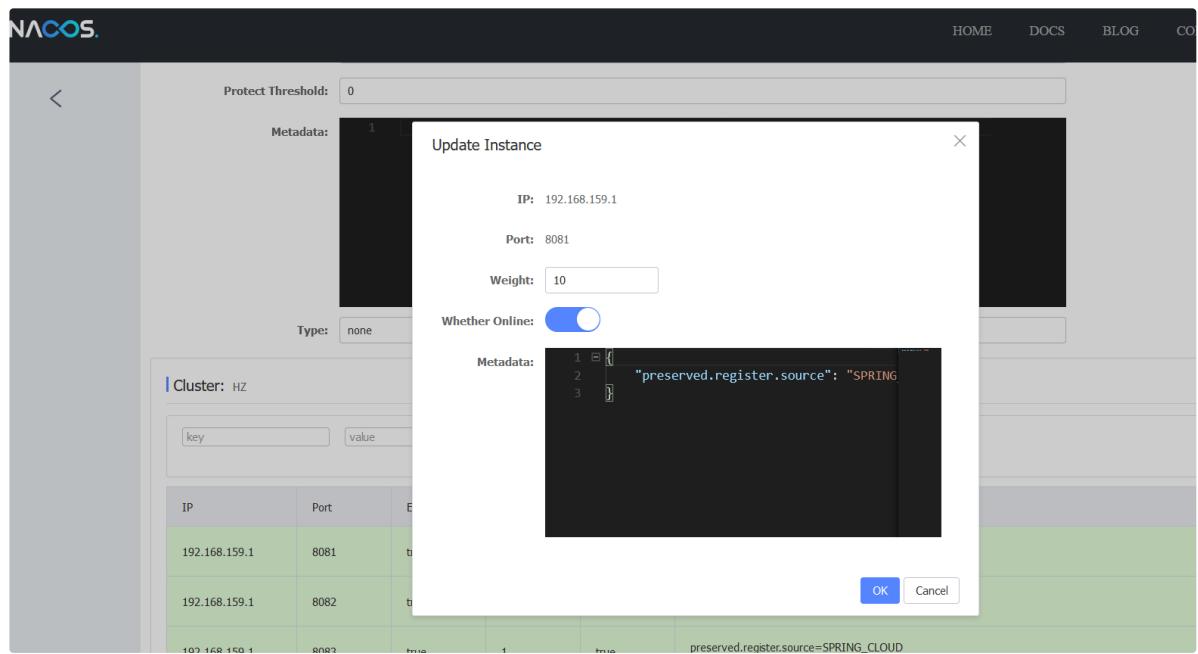
```
cloud:  
  nacos:  
    server-addr: localhost:8848  
    discovery: # 配置集群  
      cluster-name: HZ
```

```
userservice: # 集群名称  
ribbon: # 配置优先访问同集群  
  NFLoadBalancerRuleClassName:  
    com.alibaba.cloud.nacos.ribbon.NacosRule
```

此时消费者访问同一种服务时，会优先访问同集群的，但是在本地集群选择随机

本集群生产者死了访问其他

可视化修改权重



【最好设置为0~1】

## nameSpace

Namespaces	Namespace ID	Number of Configurations	Actions
public(to retain control)		0	<a href="#">Details</a> <a href="#">Delete</a> <a href="#">Edit</a>

```
cloud:
  nacos:
    server-addr: localhost:8848
    discovery:
      cluster-name: HZ
      namespace: 03961c13-1622-42cb-b0a2-6f0035762fb3 # 命名空间的ID
```

环境隔离： 不同空间下不可见

临时实例： 服务提供者主动发心跳 容机主动剔除

非临时实例： Nacos主动询问 容机等待恢复健康

Nacos会主动推送消息 和 消费者定时拉取配合

```
cloud:  
  nacos:  
    server-addr: localhost:8848  
    discovery:  
      cluster-name: HZ  
      ephemeral: false # 设置为非临时实例
```

IP	Port	Ephemeral	Weight	Healthy	Metadata
192.168.159.1	8081	false	1	true	preserved.register.source=SPRING_CLOUD

## 1. Nacos与eureka的共同点

- ① 都支持服务注册和服务拉取
- ② 都支持服务提供者心跳方式做健康检测

## 2. Nacos与Eureka的区别

- ① Nacos支持服务端主动检测提供者状态：临时实例采用心跳模式，非临时实例采用主动检测模式
- ② 临时实例心跳不正常会被剔除，非临时实例则不会被剔除
- ③ Nacos支持服务列表变更的消息推送模式，服务列表更新更及时
- ④ Nacos集群默认采用AP方式，当集群中存在非临时实例时，采用CP模式；Eureka采用AP方式

# NACOS统一配置管理

< 创建配置

\* 资料编  一般是‘服务名称-环境.yaml/properties’  
号:

\* 团体:  可以默认  
高级选项

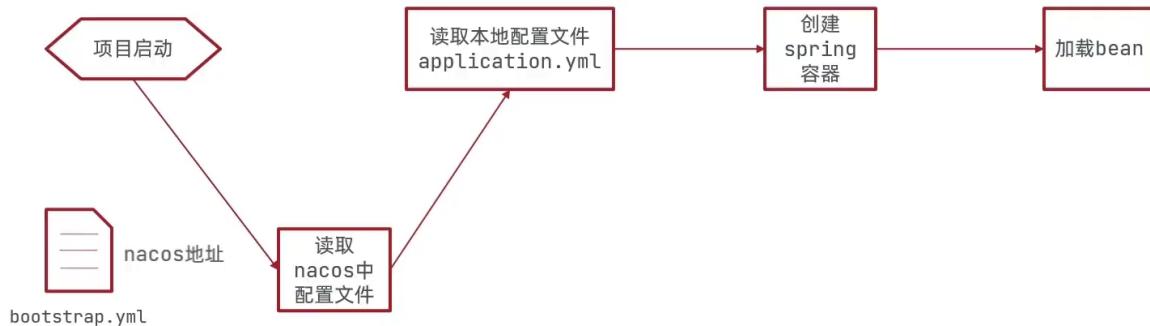
描述:

格式:  文本  JSON  XML  YAML  HTML  特性  
一般有热更新需求的配置

\* 配置内容    
② :

## 统一配置管理

配置获取的步骤如下：



## 引入客户端配置管理依赖

```

<!— nacos配置管理依赖—>
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-
config</artifactId>
</dependency>
  
```

## 在服务提供者方

新建一个 `bootstrap.yaml/yml` 文件

```
spring:
  application: # 服务名称
    name: userservice
  profiles: # 环境
    active: dev
  cloud:
    nacos:
      server-addr: localhost:8848
      discovery:
        cluster-name: HZ
      config: # 文件后缀名
        file-extension: yaml
# 去除服务配置application.yml 中相同的配置
```

配置热更新

感知方式一

```
@RequestMapping("/user")
@RefreshScope
public class UserController {

  @Autowired
  private UserService userService;
  @Value("${pattern.dateformat}")
  private String pattern;
```

感知方式二[推荐]

```
@Data
@Component
@ConfigurationProperties(prefix = "pattern")
public class PropertiesConfig {
  private String dateformat;
}
```

## 多环境共享

微服务启动时会从nacos读取多个配置文件：

- [spring.application.name]-[spring.profiles.active].yaml, 例如: userservice-dev.yaml
- [spring.application.name].yaml, 例如: userservice.yaml

无论profile如何变化, [spring.application.name].yaml这个文件一定会加载, 因此多环境共享配置可以写入这个文件

云端配置文件加环境名》云端配置文件通用》本地

## Nacos集群配置

1. 搭建数据库, 舒适化数据库表结构
2. 下载nacos安装包与配置
3. 启动nacos集群
4. Nginx反向代理

## Feign

消费者导入依赖

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

```
@FeignClient("userservice")
public interface UserFeignClient {
    @GetMapping("/user/{id}")
    User getUser(@PathVariable("id") Long id);
}
```

此时, 自动注入Web层 可以直接根据查出来的order对象的id获取User对象

## 日志配置

```
# feign的日志配置
feign:
  client:
    config:
      default: # 这里可以写默认，也可以写某种具体的实例名
      称
      loggerLevel: FULL
```

## java代码实现

```
public class FeignConfig {

  @Bean
  public Logger.Level getLogger(){
    return Logger.Level.FULL;
  }

}

// 开启配置类
// 开启Feign客户端
@EnableFeignClients(defaultConfiguration =
FeignConfig.class)
// 日志级别指定当前目录有效
@FeignClient(value = "userservice",configuration =
FeignConfig.class)
```

## 性能优化

# Feign的性能优化

Feign底层的客户端实现：

- URLConnection：默认实现，不支持连接池
- Apache HttpClient：支持连接池
- OKHttp：支持连接池

因此优化Feign的性能主要包括：

- ① 使用连接池代替默认的URLConnection
- ② 日志级别，最好用basic或none

```
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-httpclient</artifactId>
</dependency>
```

开启配置

```
# feign的日志配置
feign:
  client:
    config:
      default: # 这里可以写默认，也可以写某种具体的实例名称
        loggerLevel: BASIC
  httpclient:
    enabled: true # 开启连接池
    max-connections: 200 # 最大连接数
    max-connections-per-route: 50 # 每个路径的最大连接数
```

# 最佳实践

## Feign的最佳实践

方式一（继承）：给消费者的FeignClient和提供者的controller定义统一的父接口作为标准。

- 服务紧耦合
- 父接口参数列表中的映射不会被继承

```
public interface UserAPI{  
    @GetMapping("/user/{id}")  
    User findById(@PathVariable("id") Long id);  
}
```

```
@FeignClient(value = "userservice")  
public interface UserClient extends UserAPI{}
```

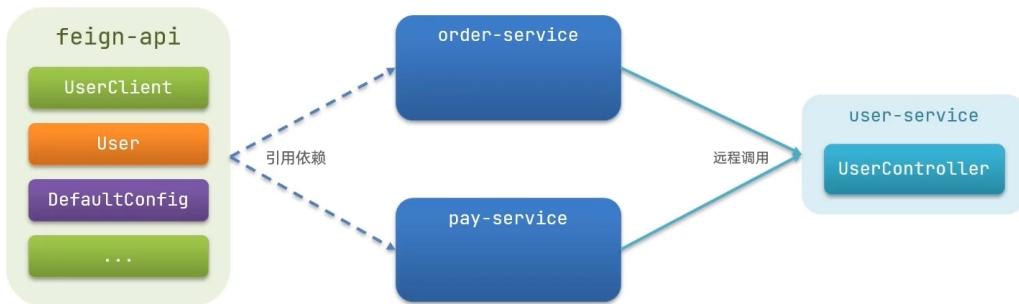
```
@RestController  
public class UserController implements UserAPI{  
    public User findById(@PathVariable("id") Long id){  
        // ... 实现业务  
    } }
```



It is generally not advisable to share an interface between a server and a client. It introduces tight coupling, and also actually doesn't work with Spring MVC in its current form (method parameter mapping is not inherited).

## Feign的最佳实践

方式二（抽取）：将FeignClient抽取为独立模块，并且把接口有关的POJO、默认的Feign配置都放到这个模块中，提供给所有消费者使用



## Feign的最佳实践

当定义的FeignClient不在SpringBootApplication的扫描包范围时，这些FeignClient无法使用。有两种方式解决：

方式一：指定FeignClient所在包

```
@EnableFeignClients(basePackages = "cn.itcast.feign.clients")
```

方式二：指定FeignClient字节码

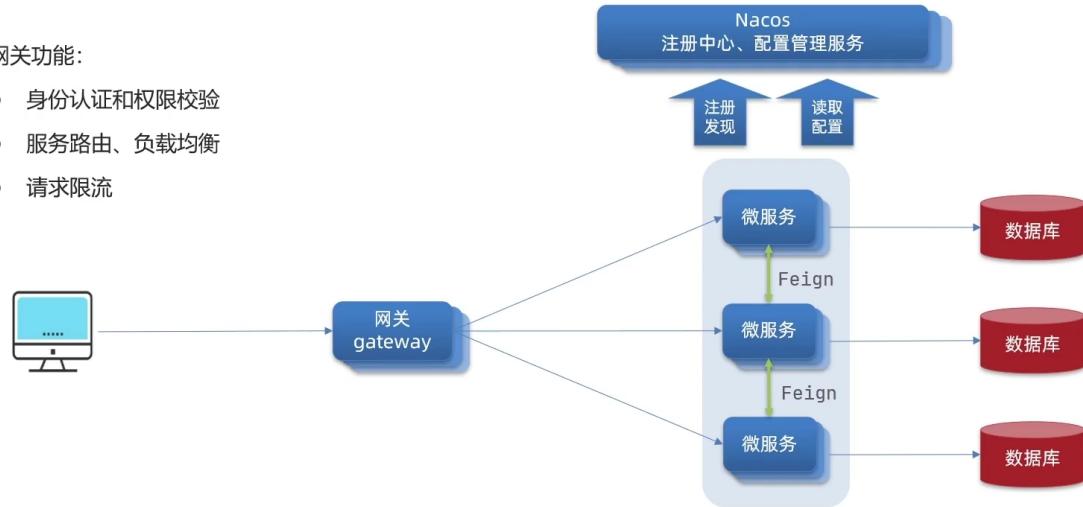
```
@EnableFeignClients(clients = {UserClient.class})
```

# 统一网关GateWay

## 为什么需要网关

网关功能：

- 身份认证和权限校验
- 服务路由、负载均衡
- 请求限流



## 网关的技术实现

在SpringCloud中网关的实现包括两种：

- gateway
- zuul

Zuul是基于Servlet的实现，属于阻塞式编程。而SpringCloudGateway则是基于Spring5中提供的WebFlux，属于响应式编程的实现，具备更好的性能。

## 新建一个模块

←!—

引入网关依赖→

```
<dependency>
```

←!—

org.springframework.cloud</groupId>

  <artifactId>spring-cloud-starter-

gateway</artifactId>

```
</dependency>
```

←!—

将自己注册到nacos的服务当中→

```
<dependency>
```

  <groupId>com.alibaba.cloud</groupId>

  <artifactId>spring-cloud-starter-

alibaba-nacos-discovery</artifactId>

```
</dependency>
```

## 并且编写启动类

### 配置文件

```
spring:  
  application:
```

```

name: gateway
cloud:
nacos:
  server-addr: 192.168.87.129:8848
gateway: # 网关路由配置
routes:
  - id: user_service #路由ID 自定义不重复
    # 路由的地址 前面的lb负载均衡 loadBalnd
    uri: lb://userservice
    predicates: # 【路由断言】
      - Path=/user/** # 断言你请求的路径是符合规
则的 以User开头
      - id: order-service
        uri: lb://orederservice
        predicates:
          - Path=/order/**

server:
port: 10010

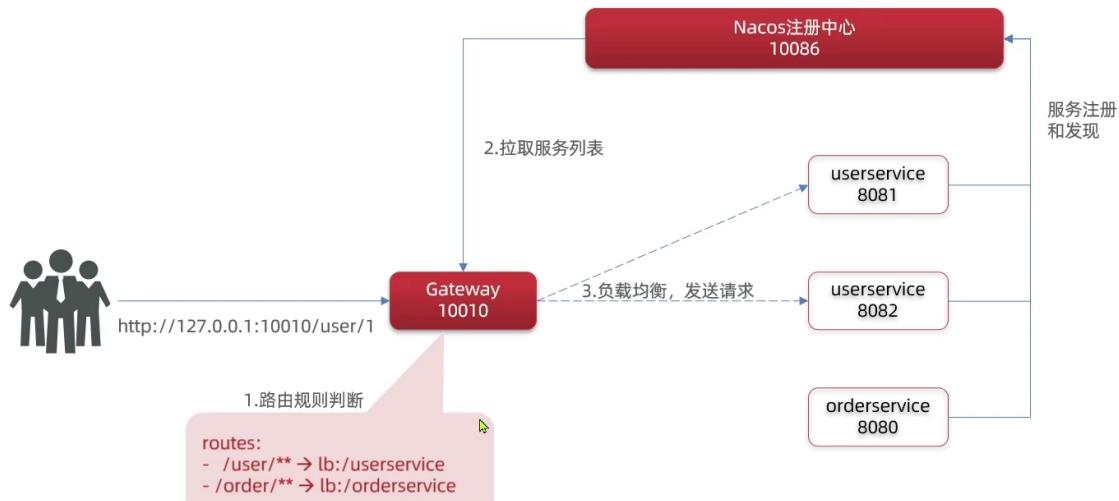
```

此时启动服务

<http://localhost:10010/order/101>

/order/101 便可以进入

搭建网关服务



## 网关搭建步骤：

1. 创建项目，引入nacos服务发现和gateway依赖
2. 配置application.yml，包括服务基本信息、nacos地址、路由

## 路由配置包括：

1. 路由id：路由的唯一标示
2. 路由目标（uri）：路由的目标地址，http代表固定地址，lb代表根据服务名负载均衡
3. 路由断言（predicates）：判断路由的规则，
4. 路由过滤器（filters）：对请求或响应做处理

Spring提供了11种基本的Predicate工厂：

名称	说明	示例
After	是某个时间点后的请求	- After=2037-01-20T17:42:47.789-07:00[America/Denver]
Before	是某个时间点之前的请求	- Before=2031-04-13T15:14:47.433+08:00[Asia/Shanghai]
Between	是某两个时间点之前的请求	- Between=2037-01-20T17:42:47.789-07:00[America/Denver], 2037-01-21T17:42:47.789-07:00[America/Denver]
Cookie	请求必须包含某些cookie	- Cookie=chocolate, ch.p
Header	请求必须包含某些header	- Header=X-Request-Id, \d+
Host	请求必须是访问某个host（域名）	- Host=*.somehost.org, *.anotherhost.org
Method	请求方式必须是指定方式	- Method=GET, POST
Path	请求路径必须符合指定规则	- Path=/red/{segment}, /blue/**
Query	请求参数必须包含指定参数	- Query=name, Jack或者- Query=name
RemoteAddr	请求者的ip必须是指定范围	- RemoteAddr=192.168.1.1/24
Weight	权重处理	

Spring提供了31种不同的路由过滤器工厂。例如：

名称	说明
AddRequestHeader	给当前请求添加一个请求头
RemoveRequestHeader	移除请求中的一个请求头
AddResponseHeader	给响应结果中添加一个响应头
RemoveResponseHeader	从响应结果中移除一个响应头
RequestRateLimiter	限制请求的流量
...	

cloud:

```
nacos:
  server-addr: 192.168.87.129:8848
  gateway: # 网关路由配置
  routes:
    - id: user-service #路由ID 自定义不重复
      # 路由的地址 前面的lb负载均衡 loadBalnd
      uri: lb://userservice
      predicates: # 【路由断言】
        - Path=/user/** # 断言你请求的路径是符合规则的 以User开头
          # 给单个服务添加请求头
      filters: #过滤器
        - AddRequestHeader=Wjl, WjlNB #添加请求头
    - id: order-service
      uri: lb://orderservice
      predicates:
        - Path=/order/**
      filters: #过滤器
        - AddRequestHeader=Wjl, WjlNB #添加请求头
          # 默认对所有的过滤器都生效
  default-filters:
    - AddRequestHeader=Wjl, WjlNB #添加请求头
```

## 全局过滤器GlobalFilter

自己写代码提供逻辑，前面的过滤器是固定逻辑的

## 全局过滤器 GlobalFilter

全局过滤器的作用也是处理一切进入网关的请求和微服务响应，与GatewayFilter的作用一样。区别在于GatewayFilter通过配置定义，处理逻辑是固定的。而GlobalFilter的逻辑需要自己写代码实现。定义方式是实现GlobalFilter接口。

```
public interface GlobalFilter {  
    /**  
     * 处理当前请求，有必要的话通过{@link GatewayFilterChain}将请求交给下一个过滤器处理  
     *  
     * @param exchange 请求上下文，里面可以获取Request、Response等信息  
     * @param chain 用来把请求委托给下一个过滤器  
     * @return {@code Mono<Void>} 返回标示当前过滤器业务结束  
     */  
    Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain);  
}
```

```
/**  
 * GlobalFilter 实现该接口配置网关过滤器处理逻辑  
 */  
// 放置在容器中  
@Component  
public class GateWayFilter implements GlobalFilter, Ordered {  
  
    @Override  
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {  
        // 获取参数  
        MultiValueMap<String, String> queryParams =  
            exchange.getRequest().getQueryParams();  
        String authorization =  
            queryParams.getFirst("authorization");  
        // 判断  
        if ("admin".equals(authorization)) {  
            // 放行  
            return chain.filter(exchange);  
        }  
        // 设置状态码 UNAUTHORIZED 401 未登录  
  
        exchange.getResponse().setStatus(HttpStatus.UNAUTHORIZED);  
        // 结束请求  
        return  
            exchange.getResponse().setComplete();  
    }  
}
```

```

    }

    @Override
    public int getOrder() {
        return -1;
    }
}

// 此时 必须携带参数才能获取 // 不然会报401

```

### 过滤器执行顺序

请求进入网关会碰到三类过滤器：当前路由的过滤器、DefaultFilter、GlobalFilter

请求路由后，会将当前路由过滤器和DefaultFilter、GlobalFilter，合并到一个过滤器链（集合）中，排序后依次执行每个过滤器



### 过滤器执行顺序

- 每一个过滤器都必须指定一个int类型的order值，**order值越小，优先级越高，执行顺序越靠前。**
- GlobalFilter通过实现Ordered接口，或者添加@Order注解来指定order值，由我们自己指定
- 路由过滤器和defaultFilter的order由Spring指定，默认是按照声明顺序从1递增。
- 当过滤器的order值一样时，会按照 defaultFilter > 路由过滤器 > GlobalFilter的顺序执行。

可以参考下面几个类的源码来查看：

`org.springframework.cloud.gateway.route.RouteDefinitionRouteLocator#getFilters()`方法是先加载defaultFilters，然后再加载某个route的filters，然后合并。

`org.springframework.cloud.gateway.handler.FilteringWebHandler#handle()`方法会加载全局过滤器，与前面的过滤器合并后根据order排序，组织过滤器链

## 跨域问题处理

## 跨域问题处理

跨域：域名不一致就是跨域，主要包括：

- 域名不同： www.taobao.com 和 www.taobao.org 和 www.jd.com 和 miaosha.jd.com
- 域名相同，端口不同： localhost:8080和localhost8081

跨域问题：浏览器禁止请求的发起者与服务端发生跨域ajax请求，请求被浏览器拦截的问题

解决方案：CORS

```
spring:  
  cloud:  
    gateway:  
      globalcors: # 全局的跨域处理  
        add-to-simple-url-handler-mapping: true #  
解决options请求被拦截问题  
      corsConfigurations:  
        '/**':  
          allowedOrigins: # 允许哪些网站的跨域请求  
            - "http://localhost:8090"  
            - "http://www.leyou.com"  
          allowedMethods: # 允许的跨域ajax的请求方式  
            - "GET"  
            - "POST"  
            - "DELETE"  
            - "PUT"  
            - "OPTIONS"  
          allowedHeaders: "*" # 允许在请求中携带的头  
信息  
          allowCredentials: true # 是否允许携带  
cookie  
          maxAge: 360000 # 这次跨域检测的有效期
```

## Docker

# Hello World

为了解决：

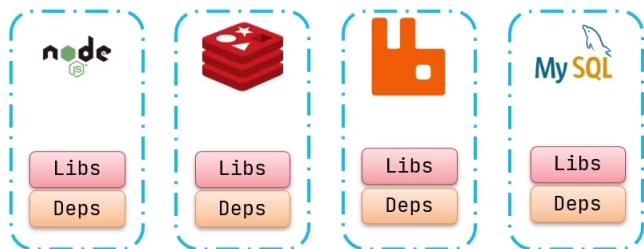
开发：我能运行！

运维： 放屁！

## Docker

Docker如何解决依赖的兼容问题的？

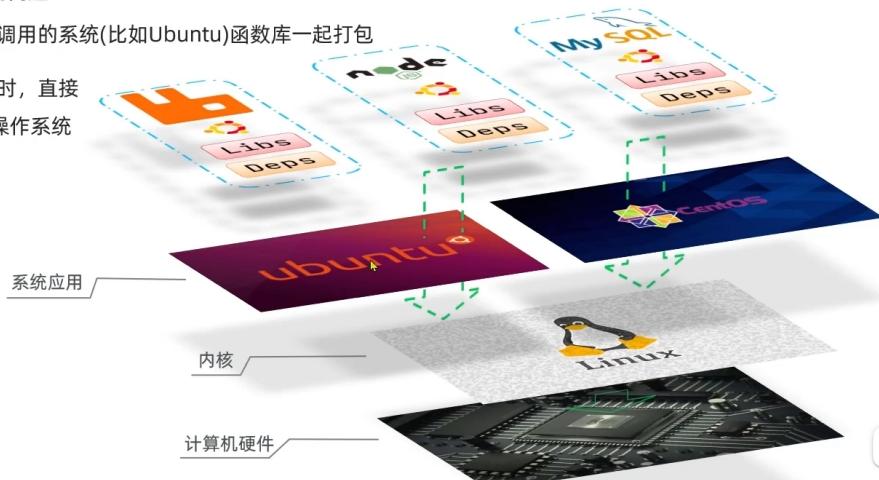
- 将应用的Libs（函数库）、Deps（依赖）、配置与应用一起打包
- 将每个应用放到一个隔离容器去运行，避免互相干扰



## Docker

Docker如何解决不同系统环境的问题？

- Docker将用户程序与所需要调用的系统(比如Ubuntu)函数库一起打包
- Docker运行到不同操作系统时，直接基于打包的库函数，借助于操作系统的Linux内核来运行



## Docker

Docker如何解决大型项目依赖关系复杂，不同组件依赖的兼容性问题？

- Docker允许开发中将应用、依赖、函数库、配置一起打包，形成可移植镜像
- Docker应用运行在容器中，使用沙箱机制，相互隔离

Docker如何解决开发、测试、生产环境有差异的问题

- Docker镜像中包含完整运行环境，包括系统函数库，仅依赖系统的Linux内核，因此可以在任意Linux操作系统上运行



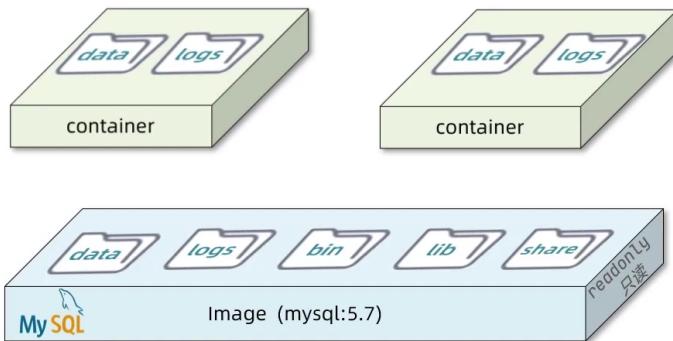
## Docker和虚拟机的差异：

- docker是一个系统进程；虚拟机是在操作系统中的操作系统
- docker体积小、启动速度快、性能好；虚拟机体积大、启动速度慢、性能一般

## 镜像和容器

**镜像（Image）**：Docker将应用程序及其所需的依赖、函数库、环境、配置等文件打包在一起，称为镜像。

**容器（Container）**：镜像中的应用程序运行后形成的进程就是容器，只是Docker会给容器做隔离，对外不可见。



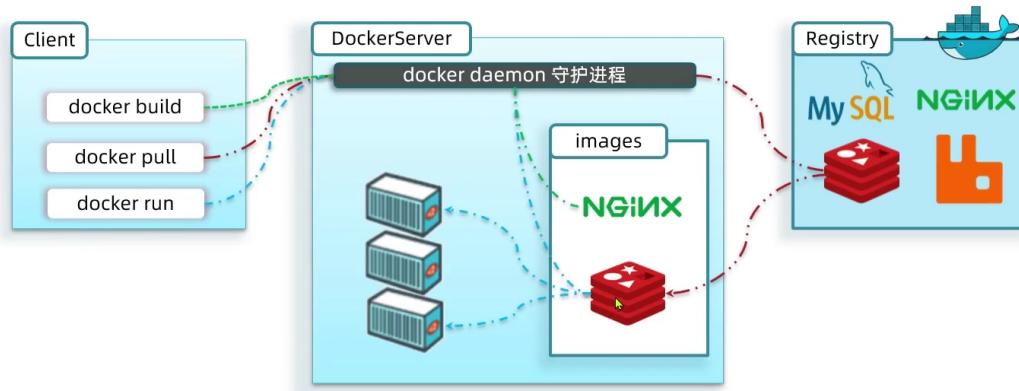
- DockerHub: DockerHub是一个Docker镜像的托管平台。这样的平台称为Docker Registry。
- 国内也有类似于DockerHub 的公开服务，比如 网易云镜像服务、阿里云镜像库等。



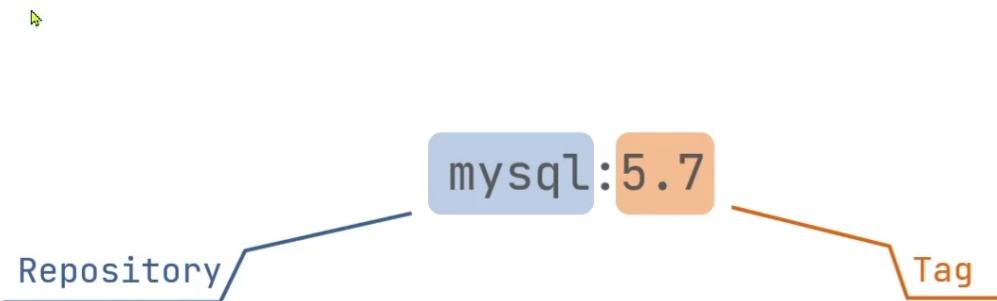
### docker架构

Docker是一个CS架构的程序，由两部分组成：

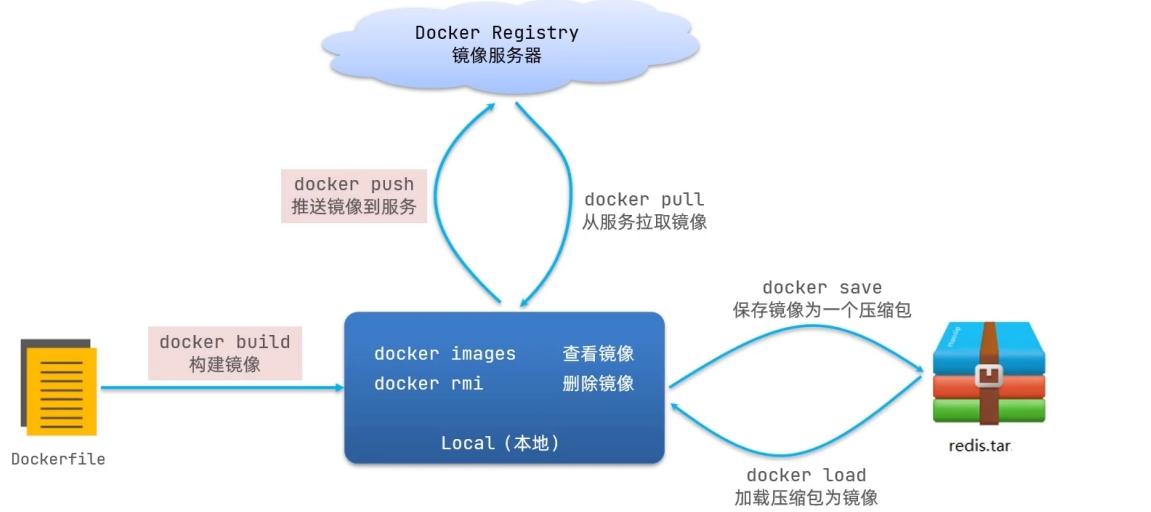
- ◆ 服务端(server): Docker守护进程，负责处理Docker指令，管理镜像、容器等
- ◆ 客户端(client): 通过命令或RestAPI向Docker服务端发送指令。可以在本地或远程向服务端发送指令。



- 镜像名称一般分两部分组成：[repository]:[tag]。
- 在没有指定tag时，默认是latest，代表最新版本的镜像



## 镜像操作命令



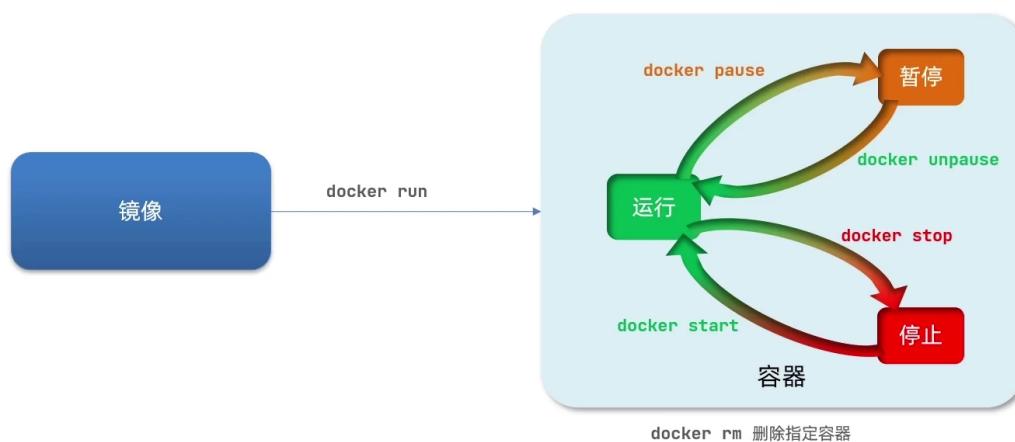
## 常用命令

```

docker images # 查看镜像
docker load -i `镜像文件`
docker save -o `导出文件` `镜像:tag`
docker rmi `镜像:tag`
docker pull `镜像:tag` # 拉取镜像 不带tag默认最新

```

## 容器相关命令



```
docker run --name ng -p 80:80 -d nginx
```

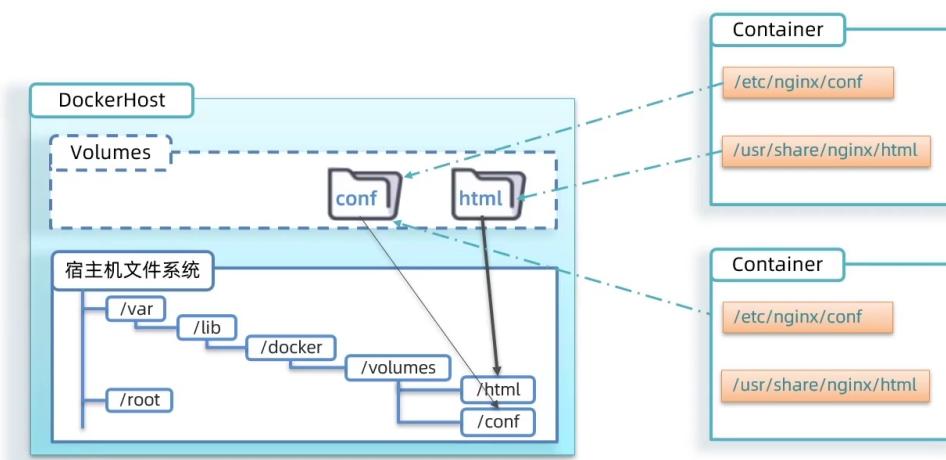
起名 别名 进入 机器端口: 软件端口 后台运行 镜像名

```
docker run --name rs -p 6379:6379 -d redis:latest  
redis-server --appendonly -yes  
# 运行持久化的redis
```

## 数据卷

### 数据卷

数据卷（volume）是一个虚拟目录，指向宿主机文件系统中的某个目录。



### 操作数据卷

数据卷操作的基本语法如下：

```
docker volume [COMMAND]
```

docker volume命令是数据卷操作，根据命令后跟随的command来确定下一步的操作：

- ◆ **create**              创建一个volume
- ◆ **inspect**            显示一个或多个volume的信息
- ◆ **ls**                  列出所有的volume
- ◆ **prune**             删除未使用的volume
- ◆ **rm**                删除一个或多个指定的volume

我们在创建容器时，可以通过 `-v` 参数来挂载一个数据卷到某个容器目录

#### 举例说明

```
docker run \
--name mn \
-v html:/root/html \
-p 8080:80
nginx \
```

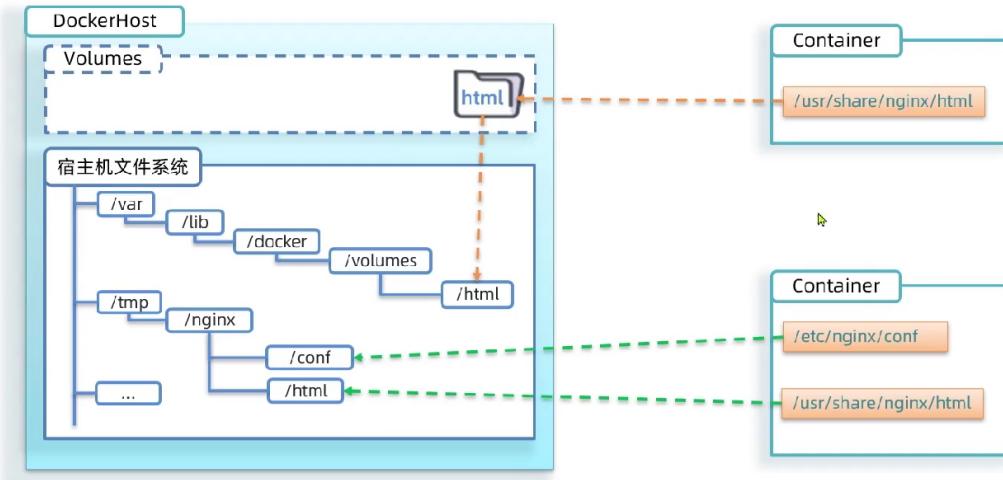
`docker run` : 就是创建并运行容器  
`--name mn` : 给容器起个名字叫mn  
`-v html:/root/html` : 把html数据卷挂载到容器内的/root/html这个目录中  
`-p 8080:80` : 把宿主机的8080端口映射到容器内的80端口  
`nginx` : 镜像名称

```
docker run --name ng -p 80:80 -v
html:/usr/share/nginx/html -d nginx:latest
# 将html数据卷【挂载】到Nginx容器的
或者说volume卷就是通用文件夹?
类似于Vue的暗箱操作
```

#### 挂载MySQL

```
docker run \
--name mysql \
-e MYSQL_ROOT_PASSWORD=123456 \
-p 3307:3306 \
-v
/tmp/mysql/conf/hmy.cnf:/etc/mysql/conf.d/hmy.cnf \
-v /tmp/mysql/data:/var/lib/mysql \
-d \
mysql

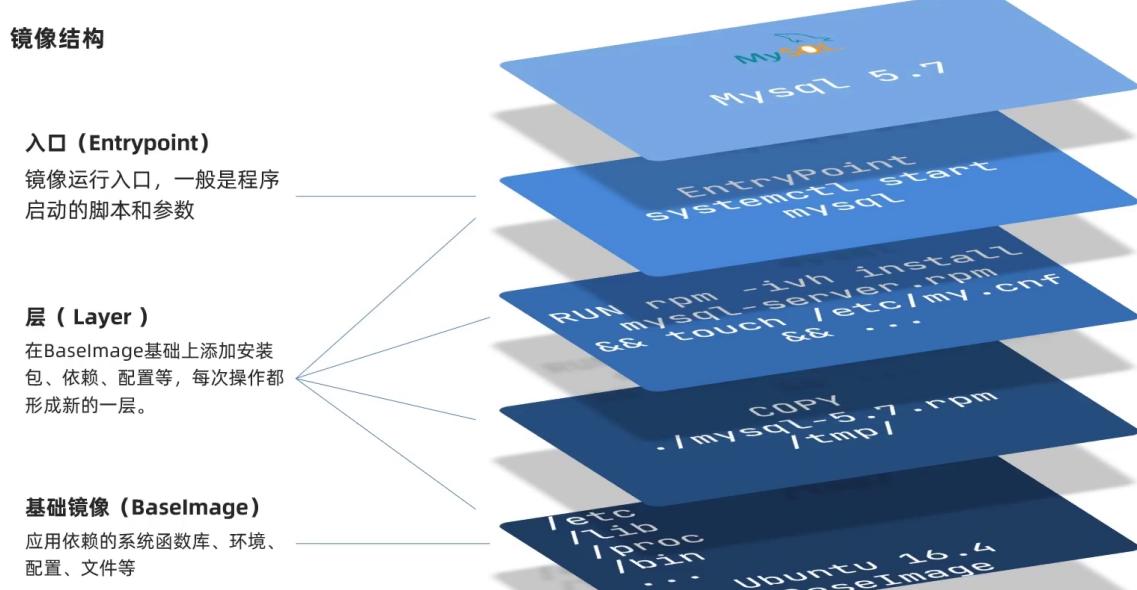
docker run --name mysql -e
MYSQL_ROOT_PASSWORD=123456 -p 3306:3306 -v
/tmp/mysql/conf/hmy.cnf:/etc/mysql/conf.d/hmy.cnf -
v /tmp/mysql/data:/var/lib/mysql -d mysql:latest
```



## 1. docker run的命令中通过 -v 参数挂载文件或目录到容器中：

- ① -v volume名称:容器内目录
- ② -v 宿主机文件:容器内文件
- ③ -v 宿主机目录:容器内目录

## Dockerfile 自定义镜像



## 什么是Dockerfile

Dockerfile就是一个文本文件，其中包含一个个的**指令(Instruction)**，用指令来说明要执行什么操作来构建镜像。每一个指令都会形成一层Layer。

指令	说明	示例
FROM	指定基础镜像	FROM centos:6
ENV	设置环境变量，可在后面指令使用	ENV key value
COPY	拷贝本地文件到镜像的指定目录	COPY ./mysql-5.7.rpm /tmp
RUN	执行Linux的shell命令，一般是安装过程的命令	RUN yum install gcc
EXPOSE	指定容器运行时监听的端口，是给镜像使用者看的	EXPOSE 8080
ENTRYPOINT	镜像中应用的启动命令，容器运行时调用	ENTRYPOINT java -jar xx.jar

更新详细语法说明，请参考官网文档：<https://docs.docker.com/engine/reference/builder>

```
# 指定基础镜像
FROM ubuntu:16.04
# 配置环境变量，JDK的安装目录
ENV JAVA_DIR=/usr/local

# 拷贝jdk和java项目的包
COPY ./jdk8.tar.gz $JAVA_DIR/
COPY ./docker-demo.jar /tmp/app.jar

# 安装JDK
RUN cd $JAVA_DIR \
&& tar -xf ./jdk8.tar.gz \
&& mv ./jdk1.8.0_144 ./java8

# 配置环境变量
ENV JAVA_HOME=$JAVA_DIR/java8
ENV PATH=$PATH:$JAVA_HOME/bin

# 暴露端口
EXPOSE 8090
# 入口，java项目的启动命令
ENTRYPOINT java -jar /tmp/app.jar

# 指定基础镜像
FROM java:8-alpine
```

```
COPY ./docker-demo.jar /tmp/app.jar
# 暴露端口
EXPOSE 8090
# 入口, java项目的启动命令
ENTRYPOINT java -jar /tmp/app.jar
```

```
docker build -t javaweb:1.0 .
```

```
docker run --name web -p 8090:8090 -d javaweb:1.0
```

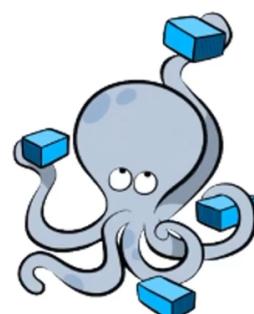
1. Dockerfile的本质是一个文件，通过指令描述镜像的构建过程
2. Dockerfile的第一行必须是FROM，从一个基础镜像来构建
3. 基础镜像可以是基本操作系统，如Ubuntu。也可以是其他人制作好的镜像，例如：java:8-alpine

## DockerCompose

### 什么是DockerCompose

- Docker Compose可以基于Compose文件帮我们快速的部署分布式应用，而无需手动一个个创建和运行容器！
- Compose文件是一个文本文件，通过指令定义集群中的每个容器如何运行。

```
version: "3.8"
services:
  mysql:
    image: mysql:5.7.25
    environment:
      MYSQL_ROOT_PASSWORD: 123
    volumes:
      - /tmp/mysql/data:/var/lib/mysql
      - /tmp/mysql/conf/hmy.cnf:/etc/mysql/conf.d/hmy.cnf
  web:
    build: .
    ports:
      - 8090: 8090
```



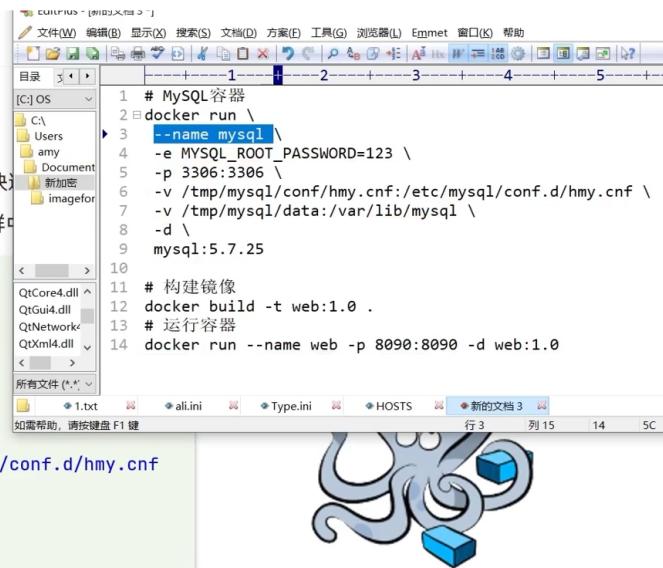
# DockerCompose

## 什么是DockerCompose

- Docker Compose可以基于Compose文件帮我们快速搭建集群
- Compose文件是一个文本文件，通过指令定义集群中各服务的运行环境

```
version: "3.8"

services:
  mysql:
    image: mysql:5.7.25
    environment:
      MYSQL_ROOT_PASSWORD: 123
    volumes:
      - /tmp/mysql/data:/var/lib/mysql
      - /tmp/mysql/conf/hmy.cnf:/etc/mysql/conf.d/hmy.cnf
  web:
    build: .
    ports:
      - 8090: 8090
```



```
[root@wj108 alp]# docker-compose --help
Define and run multi-container applications with
Docker.
```

### Usage:

```
docker-compose [-f <arg>...] [options] [COMMAND]
[ARGS...]
docker-compose -h|--help
```

### Options:

<b>-f, --file</b> FILE	Specify an alternate compose file  (default: docker- compose.yml)
<b>-p, --project-name</b> NAME	Specify an alternate project name  (default: directory name)
<b>--verbose</b>	Show more output
<b>--log-level</b> LEVEL	Set log level (DEBUG, INFO, WARNING, ERROR, CRITICAL)
<b>--no-ansi</b>	Do not print ANSI control characters
<b>-v, --version</b>	Print version and
<b>exit</b>	
<b>-H, --host</b> HOST	Daemon socket to connect to

```
--tls                                Use TLS; implied by -  
-tlsverify  
--tlscacert CA_PATH                 Trust certs signed  
only by this CA  
--tlscert CLIENT_CERT_PATH          Path to TLS  
certificate file  
--tlskey TLS_KEY_PATH               Path to TLS key file  
--tlsverify  
the remote  
--skip-hostname-check              Don't check the  
daemon's hostname against the  
name specified in the  
client certificate  
--project-directory PATH            Specify an alternate  
working directory  
                                  (default: the path of  
the Compose file)  
--compatibility  
attempt to convert deploy  
keys in v3 files to  
their non-Swarm equivalent
```

#### Commands:

build	Build or rebuild services
bundle	Generate a Docker bundle from the Compose file
config	Validate and view the Compose file
create	Create services
down	Stop and remove containers, networks, images, and volumes
events	Receive real time events from containers
exec	Execute a command in a running container
help	Get help on a command
images	List images
kill	Kill containers
logs	View output from containers
pause	Pause services

port	Print the public port <b>for</b> a
port binding	
ps	List containers
pull	Pull service images
push	Push service images
restart	Restart services
rm	Remove stopped containers
run	Run a one-off command
scale	Set number of containers <b>for</b> a
service	
start	Start services
stop	Stop services
top	Display the running processes
unpause	Unpause services
up	Create and start containers
version	Show the Docker-Compose
version information	

## 自定义镜像仓库

### 在私有镜像仓库推送或拉取镜像

推送镜像到私有镜像服务必须先tag，步骤如下：

① 重新tag本地镜像，名称前缀为私有仓库的地址：192.168.150.101:8080/

```
docker tag nginx:latest 192.168.150.101:8080/nginx:1.0
```

② 推送镜像

```
docker push 192.168.150.101:8080/nginx:1.0
```

③ 拉取镜像

```
docker pull 192.168.150.101:8080/nginx:1.0
```

```
# 打开要修改的文件  
vi /etc/docker/daemon.json  
  
{  
    "registry-mirrors":  
    ["https://uc51lue1.mirror.aliyuncs.com"],  
    "insecure-registries":  
    ["http://192.168.87.129:8080"]  
}
```

```
# 重加载  
systemctl daemon-reload  
# 重启docker  
systemctl restart docker
```

新建一个文件夹

```
mkdir registry-ui  
新建文件  
touch docker-compose.yml
```

图形化版本

```
version: '3.0'  
services:  
    registry:  
        image: registry  
        volumes:  
            - ./registry-data:/var/lib/registry  
    ui:  
        image: joxit/docker-registry-ui:static  
        ports:  
            - 8080:80  
        environment:  
            - REGISTRY_TITLE=WJL私有仓库  
            - REGISTRY_URL=http://registry:5000  
        depends_on:
```

## - registry

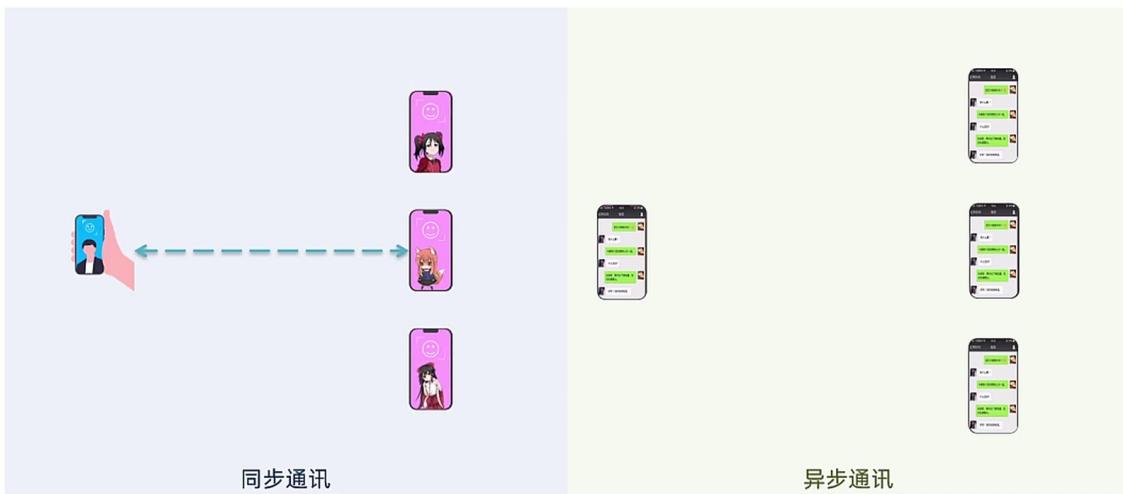
非图形化界面

```
docker run -d \
--restart=always \
--name registry \
-p 5000:5000 \
-v registry-data:/var/lib/registry \
registry
```

```
docker-compose up -d
# 构建
```

# MQ

同步通讯和异步通讯





### 什么是MQ

MQ (MessageQueue)，中文是消息队列，字面来看就是存放消息的队列。也就是事件驱动架构中的Broker。

	RabbitMQ	ActiveMQ	RocketMQ	Kafka
公司/社区	Rabbit	Apache	阿里	Apache
开发语言	Erlang	java	Java	Scala&Java
协议支持	AMQP, XMPP, SMTP, STOMP	OpenWire,STOMP, REST,XMPP,AMQP	自定义协议	自定义协议
可用性	高	一般	高	高
单机吞吐量	一般	差	高	非常高
消息延迟	微秒级	毫秒级	毫秒级	毫秒以内
消息可靠性	高	一般	高	一般

## RabbitMQ安装

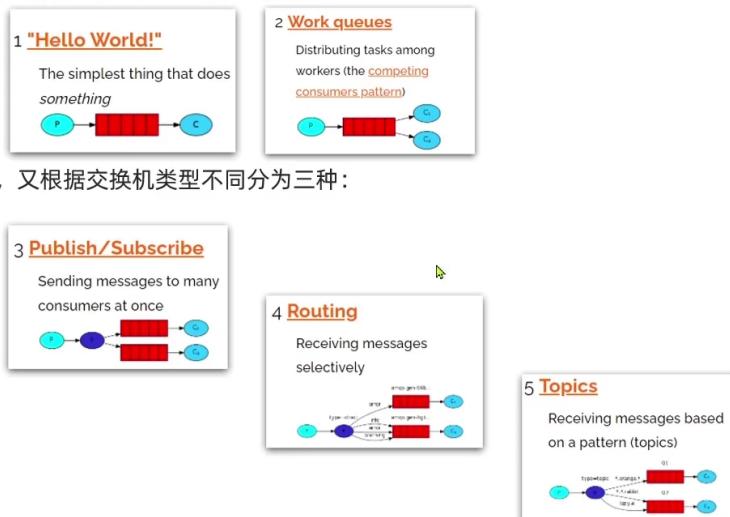
RabbitMQ中的几个概念：

- channel：操作MQ的工具
- exchange：路由消息到队列中
- queue：缓存消息
- virtual host：虚拟主机，是对queue、exchange等资源的逻辑分组

## 常见消息模型

MQ的官方文档中给出了5个MQ的Demo示例，对应了几种不同的用法：

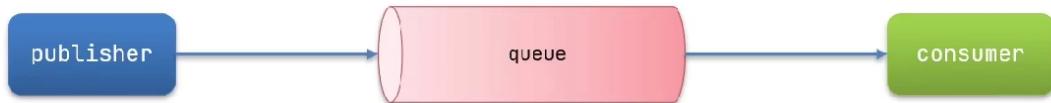
- 基本消息队列（BasicQueue）
- 工作消息队列（WorkQueue）



## HelloWorld案例

官方的HelloWorld是基于最基础的消息队列模型来实现的，只包括三个角色：

- publisher: 消息发布者，将消息发送到队列queue
- queue: 消息队列，负责接受并缓存消息
- consumer: 订阅队列，处理队列中的消息



←!—AMQP依赖，包含RabbitMQ—→  
<dependency>

```
<groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

## Work模型的使用：

- 多个消费者绑定到一个队列，同一条消息只会被一个消费者处理
- 通过设置prefetch来控制消费者预取的消息数量

```
spring:  
    rabbitmq:  
        password: 123456  
        username: wjl  
        host: 192.168.87.129  
        virtual-host: /  
        port: 5672  
        listener:  
            simple: # 设置每次只能处理一条消息 处理完成才能获取  
                下一条  
            prefetch: 1 # 也只能能者多劳
```

```
@RabbitListener(bindings = @QueueBinding(  
    value = @Queue(name="blue"),  
    exchange = @Exchange(name = "dexchange", type =  
ExchangeTypes.DIRECT),  
    key = {"blue", "red"}  
)  
public void Direct1(String msg){  
    log.info("Listener监听[blue] 【{}】 ",msg);  
}  
  
@RabbitListener(bindings = @QueueBinding(  
    value = @Queue(name = "yellow"),  
    exchange = @Exchange(name = "dexchange", type =  
ExchangeTypes.DIRECT),  
    key = {"yellow", "red"}  
)  
public void Direct2(String msg){
```

```
    log.info("Listener监听[yellow] {}", msg);
}
```

## 消息转换器

Spring对消息对象的处理是由org.springframework.amqp.support.converter.MessageConverter来处理的。而默认实现是SimpleMessageConverter，基于JDK的ObjectOutputStream完成序列化。

如果要修改只需要定义一个MessageConverter类型的Bean即可。推荐用JSON方式序列化，步骤如下：

- 我们在publisher服务引入依赖

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
    <version>2.9.10</version>
</dependency>
```

- 我们在publisher服务声明MessageConverter：

```
@Bean
public MessageConverter jsonMessageConverter(){
    return new Jackson2JsonMessageConverter();
}
```

## 消息转换器

我们在consumer服务引入Jackson依赖：

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
    <version>2.9.10</version>
</dependency>
```

我们在consumer服务定义MessageConverter：

```
@Bean
public MessageConverter jsonMessageConverter(){
    return new Jackson2JsonMessageConverter();
}
```

然后定义一个消费者，监听object.queue队列并消费消息：

```
@RabbitListener(queues = "object.queue")
public void listenObjectQueue(Map<String, Object> msg) {
    System.out.println("收到消息: [" + msg + "]");
}
```

ES

## 正向索引和倒排索引

elasticsearch采用倒排索引：

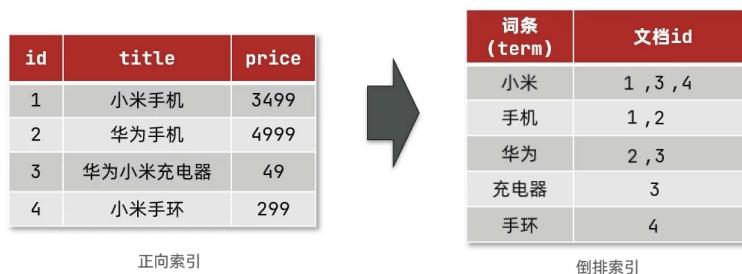
- 文档 (document)：每条数据就是一个文档
- 词条 (term)：文档按照语义分成的词语



## 正向索引和倒排索引

elasticsearch采用倒排索引：

- 文档 (document)：每条数据就是一个文档
- 词条 (term)：文档按照语义分成的词语



elasticsearch是面向文档存储的，可以是数据库中的一条商品数据，一个订单信息。

文档数据会被序列化为json格式后存储在elasticsearch中。

The diagram illustrates the relationship between forward indexing and inverted indexing. On the left, a table shows four documents with fields: id, title, and price. The documents are:

id	title	price
1	小米手机	3499
2	华为手机	4999
3	华为小米充电器	49
4	小米手环	299

An arrow points from this table to a JSON object on the right.

```
{
  "id": 1,
  "title": "小米手机",
  "price": 3499
}
{
  "id": 2,
  "title": "华为手机",
  "price": 4999
}
{
  "id": 3,
  "title": "华为小米充电器",
  "price": 49
}
{
  "id": 4,
  "title": "小米手环",
  "price": 299
}
```

## 索引 (Index)

- 索引 (index) : 相同类型的文档的集合
- 映射 (mapping) : 索引中文档的字段约束信息, 类似表的结构约束

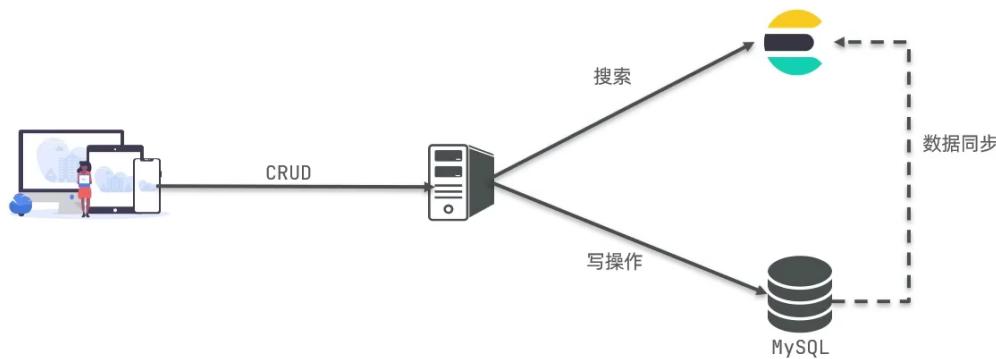


MySQL	Elasticsearch	说明
Table	Index	索引(index), 就是文档的集合, 类似数据库的表(table)
Row	Document	文档 (Document), 就是一条条的数据, 类似数据库中的行 (Row), 文档都是JSON格式
Column	Field	字段 (Field), 就是JSON文档中的字段, 类似数据库中的列 (Column)
Schema	Mapping	Mapping (映射) 是索引中文档的约束, 例如字段类型约束。类似数据库的表结构 (Schema)
SQL	DSL	DSL是elasticsearch提供的JSON风格的请求语句, 用来操作 elasticsearch, 实现CRUD

## 架构

Mysql: 擅长事务类型操作, 可以确保数据的安全和一致性

Elasticsearch: 擅长海量数据的搜索、分析、计算



# 数据类型

## mapping属性

mapping是对索引库中文档的约束，常见的mapping属性包括：

- type：字段数据类型，常见的简单类型有：
  - 字符串：text（可分词的文本）、keyword（精确值，例如：品牌、国家、ip地址）
  - 数值：long、integer、short、byte、double、float、
  - 布尔：boolean
  - 日期：date
  - 对象：object
- index：是否创建索引，默认为true
- analyzer：使用哪种分词器
- properties：该字段的子字段

```
{  
    "age": 21,  
    "weight": 52.1,  
    "isMarried": false,  
    "info": "黑马程序员Java讲师",  
    "email": "zy@itcast.cn",  
    "score": [99.1, 99.5, 98.9],  
    "name": {  
        "firstName": "云",  
        "lastName": "赵"  
    }  
}
```

## RestClient操作索引库

### 步骤 步骤2：分析数据结构

mapping要考虑的问题：

字段名、数据类型、是否参与搜索、是否分词、如果分词，分词器是什么？

```
CREATE TABLE `tb_hotel` (  
    `id` bigint(20) NOT NULL COMMENT '酒店id',  
    `name` varchar(255) NOT NULL COMMENT '酒店名称；例：7天酒店',  
    `address` varchar(255) NOT NULL COMMENT '酒店地址；例：航头路',  
    `price` int(10) NOT NULL COMMENT '酒店价格；例：329',  
    `score` int(2) NOT NULL COMMENT '酒店评分；例：45，就是4.5分',  
    `brand` varchar(32) NOT NULL COMMENT '酒店品牌；例：如家',  
    `city` varchar(32) NOT NULL COMMENT '所在城市；例：上海',  
    `star_name` varchar(16) DEFAULT NULL COMMENT '酒店星级，从低到高分别是：1星到5星，1钻到5钻',  
    `business` varchar(255) DEFAULT NULL COMMENT '商圈；例：虹桥',  
    `latitude` varchar(32) NOT NULL COMMENT '纬度；例：31.2497',  
    `longitude` varchar(32) NOT NULL COMMENT '经度；例：120.3925',  
    `pic` varchar(255) DEFAULT NULL COMMENT '酒店图片；例：/img/1.jpg',  
    PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```



黑马程序员  
www.itheima.com

#### 小提示

ES中支持两种地理坐标数据类型：

- geo\_point：由纬度（latitude）和经度（longitude）确定的一个点。例如：“32.8752345, 120.2981576”
- geo\_shape：有多个geo\_point组成的复杂几何图形。例如一条直线，“LINESTRING (-77.03653 38.897676, -77.009051 38.889939)”

#### 小提示

字段拷贝可以使用copy\_to属性将当前字段拷贝到指定字段。示例：

```
{"all": {  
    "type": "text",  
    "analyzer": "ik_max_word"  
},  
"brand": {  
    "type": "keyword",  
    "copy_to": "all"  
}}
```

The screenshot shows the Elasticsearch Dev Tools interface. In the top navigation bar, there are tabs for 'Console', 'Search Profiler', 'Grok Debugger', and 'Painless Lab' (BETA). The 'Console' tab is selected. Below the tabs, there are 'History', 'Settings', and 'Help' buttons. The main area contains a code editor with a JSON document. The JSON document includes fields like 'city', 'business', 'location', 'pic', and 'all'. To the right of the code editor, the search results are displayed, showing a single hit for the index 'heima'. The result includes fields such as '\_index', '\_type', '\_id', '\_version', '\_seq\_no', '\_primary\_term', '\_found', '\_source', '\_info', '\_email', and '\_name'. The response status is '200 - OK' and the time taken is '24 ms'.

```

134 },
135 "city": {
136   "type": "keyword"
137 },
138 "starName": {
139   "type": "keyword"
140 },
141 "business": {
142   "type": "keyword",
143   "copy_to": "all"
144 },
145 "location": {
146   "type": "geo_point"
147 },
148 "pic": {
149   "type": "keyword",
150   "index": false
151 },
152 "all": {
153   "type": "text",
154   "analyzer": "ik_max_word"
155 }
156 }
157

```

```

1 * {
2   "_index": "heima",
3   "_type": "doc",
4   "_id": "1",
5   "_version": 11,
6   "_seq_no": 13,
7   "_primary_term": 2,
8   "found": true,
9   "source": {
10     "_info": "黑马程序员java讲师",
11     "_email": "ZYun@itcast.cn",
12     "_name": {
13       "firstName": "云",
14       "lastName": "赵"
15     }
16   }
17 }
18

```

# 索引 [index]

## 创建索引库

ES中通过Restful请求操作索引库、文档。请求内容用DSL语句来表示。创建索引库和mapping的DSL语法如下：

The diagram illustrates the creation of an index and its mapping using Elasticsearch's DSL. It consists of two side-by-side code snippets separated by a large arrow pointing from left to right.

**示例**

**左侧代码（示例）：**

```

PUT /索引库名称
{
  "mappings": {
    "properties": {
      "字段名": {
        "type": "text",
        "analyzer": "ik_smart"
      },
      "字段名2": {
        "type": "keyword",
        "index": "false"
      },
      "字段名3": {
        "properties": {
          "子字段": {
            "type": "keyword"
          }
        }
      }
    }
  }
}

```

**右侧代码（示例）：**

```

PUT /heima
{
  "mappings": {
    "properties": {
      "info": {
        "type": "text",
        "analyzer": "ik_smart"
      },
      "email": {
        "type": "keyword",
        "index": "false"
      },
      "name": {
        "properties": {
          "firstName": {
            "type": "keyword"
          }
        }
      }
    }
  }
}

```

## 查看、删除索引库

查看索引库语法：

```
GET /索引库名
```

示例：

```
GET /heima
```

删除索引库的语法：

```
DELETE /索引库名
```

示例：

```
DELETE /heima
```

## 修改索引库

索引库和mapping一旦创建无法修改，但是可以添加新的字段，语法如下：

```
PUT /索引库名/_mapping
{
  "properties": {
    "新字段名": {
      "type": "integer"
    }
  }
}
```

示例：

```
PUT /heima/_mapping
{
  "properties": {
    "age": {
      "type": "integer"
    }
  }
}
```

## 文档【\_doc】

## 添加文档

新增文档的DSL语法如下：

```
POST /索引库名/_doc/文档id
{
    "字段1": "值1",
    "字段2": "值2",
    "字段3": [
        "子属性1": "值3",
        "子属性2": "值4"
    ],
    // ...
}
```

## 查看、删除文档

查看文档语法：

```
GET /索引库名/_doc/文档id
```

示例：

```
GET /heima/_doc/1
```

删除索引库的语法：

```
DELETE /索引库名/_doc/文档id
```

示例：

```
DELETE /heima/_doc/1
```

## 修改文档

方式一：全量修改，会删除旧文档，添加新文档

```
PUT /索引库名/_doc/文档id
{
    "字段1": "值1",
    "字段2": "值2",
    // ... 略
}
```

方式二：增量修改，修改指定字段值

```
POST /索引库名/_update/文档id
{
    "doc": {
        "字段名": "新的值",
    }
}
```

RestClient

1. 引入es的RestHighLevelClient依赖:

```
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>elasticsearch-rest-high-level-client</artifactId>
</dependency>
```

2. 因为SpringBoot默认的ES版本是7.6.2, 所以我们需要覆盖默认的ES版本:

```
<properties>
    <java.version>1.8</java.version>
    <elasticsearch.version>7.12.1</elasticsearch.version>
</properties>
```

3. 初始化RestHighLevelClient:

```
RestHighLevelClient client = new RestHighLevelClient(RestClient.builder(
    HttpHost.create("http://192.168.150.101:9200"))
);
```

### @Configuration

```
public class RestClientConfig extends
AbstractElasticsearchConfiguration {
    @Override
    @Bean
    public RestHighLevelClient elasticsearchClient() {
        final ClientConfiguration clientConfiguration =
        ClientConfiguration.builder()
            .connectedTo("172.16.91.10:9200")
            .build();
        return
        RestClients.create(clientConfiguration).rest();
    }
}
```

## 索引

创建索引库代码如下：

```
@Test  
void testCreateHotelIndex() throws IOException {  
    // 1. 创建Request对象  
    CreateIndexRequest request = new CreateIndexRequest("hotel");  
    // 2. 请求参数, MAPPING_TEMPLATE是静态常量字符串, 内容是创建索引库的DSL语句  
    request.source(MAPPING_TEMPLATE, XContentType.JSON);  
    // 3. 发起请求  
    client.indices().create(request, RequestOptions.DEFAULT);  
}
```



- 删除索引库代码如下：

```
@Test  
void testDeleteHotelIndex() throws IOException {  
    // 1. 创建Request对象  
    DeleteIndexRequest request = new DeleteIndexRequest("hotel");  
    // 2. 发起请求  
    client.indices().delete(request, RequestOptions.DEFAULT);  
}
```

- 判断索引库是否存在

```
@Test  
void testExistsHotelIndex() throws IOException {  
    // 1. 创建Request对象  
    GetIndexRequest request = new GetIndexRequest("hotel");  
    // 2. 发起请求  
    boolean exists = client.indices().exists(request, RequestOptions.DEFAULT);  
    // 3. 输出  
    System.out.println(exists);  
}
```

## 文档

先查询酒店数据，然后给这条数据创建倒排索引，即可完成添加：

```
@Test  
void testIndexDocument() throws IOException {  
    // 1. 创建request对象  
    IndexRequest request = new IndexRequest("indexName").id("1");  
    // 2. 准备JSON文档  
    request.source("{\"name\": \"Jack\", \"age\": 21}", XContentType.JSON);  
    // 3. 发送请求  
    client.index(request, RequestOptions.DEFAULT);  
}
```



根据id查询到的文档数据是json，需要反序列化为java对象：

```
@Test
void testGetDocumentById() throws IOException {
    // 1. 创建request对象
    GetRequest request = new GetRequest("indexName", "1");
    // 2. 发送请求，得到结果
    GetResponse response = client.get(request, RequestOptions.DEFAULT);
    // 3. 解析结果
    String json = response.getSourceAsString();

    System.out.println(json);
}
```

GET /indexName/\_doc/1

```
{
  "_index": "users",
  "_type": "_doc",
  "_id": "1",
  "_version": 1,
  "_seq_no": 0,
  "_primary_term": 1,
  "found": true,
  "_source": {
    "name": "Jack",
    "age": 21
  }
}
```

修改文档数据有两种方式：

方式一：全量更新。再次写入id一样的文档，就会删除旧文档，添加新文档

方式二：局部更新。只更新部分字段，我们演示方式二

```
@Test
void testUpdateDocumentById() throws IOException {
    // 1. 创建request对象
    UpdateRequest request = new UpdateRequest("indexName", "1");
    // 2. 准备参数，每2个参数为一对 key value
    request.doc(
        "age", 18,
        "name", "Rose"
    );
    // 3. 更新文档
    client.update(request, RequestOptions.DEFAULT);
}
```

POST /users/\_update/1

```
{
  索引库名、 文档id
  "doc": {
    "name": "Rose",
    "age": 18
  }
}
```

要修改的字段

## 批量增加

需求：批量查询酒店数据，然后批量导入索引库中

思路：

1. 利用mybatis-plus查询酒店数据
2. 将查询到的酒店数据（Hotel）转换为文档类型数据（HotelDoc）
3. 利用JavaRestClient中的Bulk批处理，实现批量新增文档，示例代码如下

```
@Test
void testBulk() throws IOException {
    // 1. 创建Bulk请求
    BulkRequest request = new BulkRequest();
    // 2. 添加要批量提交的请求：这里添加了两个新增文档的请求
    request.add(new IndexRequest("hotel")
        .id("101").source("json source", XContentType.JSON));
    request.add(new IndexRequest("hotel")
        .id("102").source("json source2", XContentType.JSON));
    // 3. 发起bulk请求
    client.bulk(request, RequestOptions.DEFAULT);
}
```

```

for (Hotel hotel : hotels) {
    HotelDoc hotelDoc = new HotelDoc(hotel);
}

// 1. 创建Request
BulkRequest request = new BulkRequest();
// 2. 准备参数，添加多个新增的Request
for (Hotel hotel : hotels) {
    // 转换为文档类型HotelDoc
    HotelDoc hotelDoc = new HotelDoc(hotel);
    // 创建新增文档的Request对象
    request.add(new IndexRequest("hotel")
        .id(hotelDoc.getId().toString())
        .source(JSON.toJSONString(hotelDoc), XContentType.JSON));
}

request.add(new IndexRequest("hotel").id("61038").source(source: "json", XContentType.JSON));
request.add(new IndexRequest("hotel").id("61038").source(source: "json", XContentType.JSON));
}

public HotelDoc(Hotel hotel) {
    this.id = hotel.getId();
    this.name = hotel.getName();
    this.address = hotel.getAddress();
    this.price = hotel.getPrice();
    this.score = hotel.getScore();
    this.brand = hotel.getBrand();
    this.city = hotel.getCity();
    this.starName = hotel.getStarName();
    this.business = hotel.getBusiness();
    this.location = hotel.getLatitude() + ", " + hotel.getLongitude();
    this.pic = hotel.getPic();
}

```

## 高级查询

### DSL Query的分类

Elasticsearch提供了基于JSON的DSL ([Domain Specific Language](#)) 来定义查询。常见的查询类型包括：

- 查询所有：查询出所有数据，一般测试用。例如：match\_all
- 全文检索 (full text) 查询：利用分词器对用户输入内容分词，然后去倒排索引库中匹配。例如：
  - match\_query
  - multi\_match\_query
- 精确查询：根据精确词条值查找数据，一般是查找keyword、数值、日期、boolean等类型字段。例如：
  - ids
  - range
  - term
- 地理 (geo) 查询：根据经纬度查询。例如：
  - geo\_distance
  - geo\_bounding\_box
- 复合 (compound) 查询：复合查询可以将上述各种查询条件组合起来，合并查询条件。例如：
  - bool
  - function\_score

# 简单查询

## 全文检索查询

match查询：全文检索查询的一种，会对用户输入内容分词，然后去倒排索引库检索，语法：

```
GET /indexName/_search
{
  "query": {
    "match": {
      "FIELD": "TEXT"
    }
  }
}
```

multi\_match：与match查询类似，只不过允许同时查询多个字段，语法：

```
GET /indexName/_search
{
  "query": {
    "multi_match": {
      "query": "TEXT",
      "fields": ["FIELD1", "FIELD12"]
    }
  }
}
```

```
7
8  PUT /test
9  {
10 "mappings": {
11   "properties": {
12     "id": {
13       "type": "integer"
14     },
15     "title": {
16       "type": "keyword",
17       "copy_to": "all"
18     },
19     "desc": {
20       "type": "text",
21       "copy_to": "all"
22     }
23   }
24 }
```

```

25
26 GET /test/_search
27 {
28   "query": {
29     | "match_all": {}
30   }
31 }
32
33 GET /test/_search
34 {
35   "query": {
36     | "match": {
37       | "all": "出"
38     }
39   }

```

▶ ↻

```

13   "value" : 1,
14   "relation" : "eq"
15 },
16 "max_score" : 1.1106448,
17 "hits" : [
18   {
19     "_index" : "test",
20     "_type" : "_doc",
21     "_id" : "4",
22     "_score" : 1.1106448,
23     "_source" : {
24       "id" : 4,
25       "title" : "熊出没",
26       "desc" : "垃圾"
27     }

```

PicGo

## 精确查询-语法

精确查询一般是根据id、数值、keyword类型、或布尔字段来查询。语法如下：

term查询：

```
// term查询
GET /indexName/_search
{
  "query": {
    "term": {
      "FIELD": {
        "value": "VALUE"
      }
    }
  }
}
```

range查询：

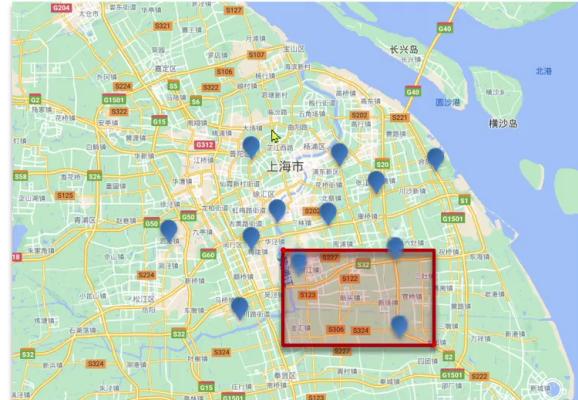
```
// range查询
GET /indexName/_search
{
  "query": {
    "range": {
      "FIELD": {
        "gte": 10,
        "lte": 20
      }
    }
  }
}
```

## 地理查询

根据经纬度查询，[官方文档](#)。例如：

- geo\_bounding\_box：查询geo\_point值落在某个矩形范围的所有文档

```
// geo_bounding_box查询
GET /indexName/_search
{
  "query": {
    "geo_bounding_box": {
      "FIELD": {
        "top_left": {
          "lat": 31.1,
          "lon": 121.5
        },
        "bottom_right": {
          "lat": 30.9,
          "lon": 121.7
        }
      }
    }
  }
}
```

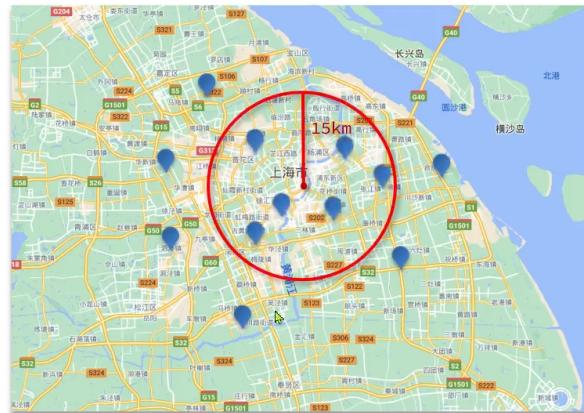


## 地理查询

根据经纬度查询，[官方文档](#)。例如：

- geo\_distance：查询到指定中心点小于某个距离值的所有文档

```
// geo_distance 查询
GET /indexName/_search
{
  "query": {
    "geo_distance": {
      "distance": "15km",
      "FIELD": "31.21,121.5"
    }
  }
}
```



## 算法排序

### 相关性算分

当我们利用match查询时，文档结果会根据与搜索词条的关联度打分（\_score），返回结果时按照分值降序排列。

例如，我们搜索 "虹桥如家"，结果如下：

```
[
  {
    "_score" : 17.850193,
    "_source" : {
      "name" : "虹桥如家酒店真不错",
    }
  },
  {
    "_score" : 12.259849,
    "_source" : {
      "name" : "外滩如家酒店真不错",
    }
  },
  {
    "_score" : 11.91091,
    "_source" : {
      "name" : "迪士尼如家酒店真不错",
    }
  }
]
```

$$TF(\text{词条频率}) = \frac{\text{词条出现次数}}{\text{文档中词条总数}}$$

$$\begin{aligned} \text{TF-IDF算法} \\ \text{IDF}(\text{逆文档频率}) &= \log\left(\frac{\text{文档总数}}{\text{包含词条的文档总数}}\right) \\ \text{score} &= \sum_i^n \text{TF}(\text{词条频率}) * \text{IDF}(\text{逆文档频率}) \end{aligned}$$

$$\begin{aligned} \text{BM25算法} \\ \text{Score}(Q, d) &= \sum_i^n \log\left(1 + \frac{N-n+0.5}{n+0.5}\right) \cdot \frac{f_i}{f_i + k_1 \cdot (1-b + b \cdot \frac{d_i}{avgdl})} \end{aligned}$$

### Function Score Query

使用 [function score query](#)，可以修改文档的相关性算分（query score），根据新得到的算分排序。

```
GET /hotel/_search
{
  "query": {
    "function_score": {
      "query": { "match": { "all": "外滩" } },
      "functions": [
        {
          "filter": { "term": { "id": "1" } },
          "weight": 10
        }
      ],
      "boost_mode": "multiply"
    }
  }
}
```

原始查询条件，搜索文档并根据相关性打分(query score)

过滤条件，符合条件的文档才会被重新算分

算分函数，算分函数的结果称为function score，将来会与query score运算，得到新算分，常见的算分函数有：

- weight：给一个常量值，作为函数结果(function score)
- field\_value\_factor：用文档中的某个字段值作为函数结果
- random\_score：随机生成一个值，作为函数结果
- script\_score：自定义计算公式，公式结果作为函数结果

加权模式，定义function score与query score的运算方式，包括：

- multiply：两者相乘。默认就是这个
- replace：用function score替换query score
- 其它：sum、avg、max、min

# 布尔查询

## 复合查询 Boolean Query

布尔查询是一个或多个查询子句的组合。子查询的组合方式有：

- must: 必须匹配每个子查询，类似“与”
- should: 选择性匹配子查询，类似“或”
- must\_not: 必须不匹配，不参与算分，类似“非”
- filter: 必须匹配，不参与算分

The screenshot shows a search interface for '黑马旅游' (www.theima.com). It includes a search bar and a sidebar with filters for '全部结果' (All results), '城市' (City), '星级' (Star Rating), '品牌' (Brand), and '价格' (Price). The '城市' filter has options for Shanghai, Beijing, Shenzhen, and Hangzhou. The '星级' filter has options for 2-star, 3-star, 4-star, 5-star, and 3-star. The '品牌' filter lists 7 Days Hotel, JI Hotel, Super 8, Holiday Inn Express, and Hanting Hotel. The '价格' filter shows ranges: 100元以下, 100-300元, 300-600元, 600-1500元, and 1500元以上.

```
GET /hotel/_search
{
  "query": {
    "bool": {
      "must": [
        {"term": {"city": "上海"}},
        {"term": {"brand": "皇冠假日"}, "term": {"brand": "华美达"}}
      ],
      "must_not": [
        {"range": {"price": {"lte": 500}}}
      ],
      "filter": [
        {"range": {"score": {"gte": 45}}}
      ]
    }
  }
}
```

## 案例 利用bool查询实现功能

需求：搜索名字包含“如家”，价格不高于400，在坐标31.21,121.5周围10km范围内的酒店。

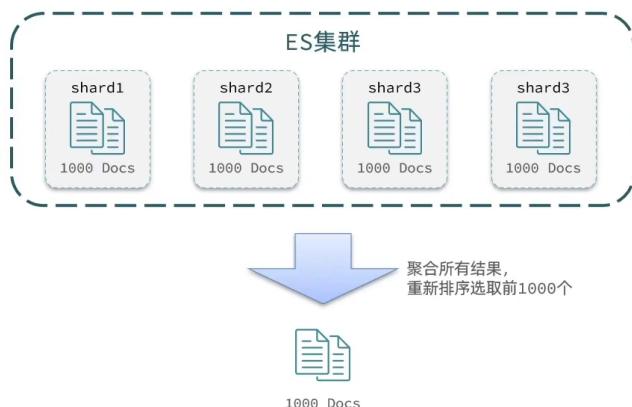
```
GET /hotel/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {"name": "如家"}}
      ],
      "must_not": [
        {"range": {"price": {"gt": 400}}}
      ],
      "filter": [
        {"geo_distance": {
          "distance": "10km", "location": {"lat": 31.21, "lon": 121.5}}
        }
      ]
    }
  }
}
```

# 搜索结果处理

## 深度分页问题

ES是分布式的，所以会面临深度分页问题。例如按price排序后，获取from = 990, size =10的数据：

1. 首先在每个数据分片上都排序并查询前1000条文档。
2. 然后将所有节点的结果聚合，在内存中重新排序选出前1000条文档
3. 最后从这1000条中，选取从990开始的10条文档



### from + size:

- 优点：支持随机翻页
- 缺点：深度分页问题，默认查询上限（from + size）是10000
- 场景：百度、京东、谷歌、淘宝这样的随机翻页搜索

### after search:

- 优点：没有查询上限（单次查询的size不超过10000）
- 缺点：只能向后逐页查询，不支持随机翻页
- 场景：没有随机翻页需求的搜索，例如手机向下滚动翻页

### scroll:

- 优点：没有查询上限（单次查询的size不超过10000）
- 缺点：会有额外内存消耗，并且搜索结果是非实时的
- 场景：海量数据的获取和迁移。从ES7.1开始不推荐，建议用 after search方案。

## 高亮

高亮：就是在搜索结果中把搜索关键字突出显示。

原理是这样的：

- 将搜索结果中的关键字用标签标记出来
- 在页面中给标签添加css样式

语法：

```
GET /hotel/_search
{
  "query": {
    "match": {
      "FIELD": "TEXT"
    }
  },
  "highlight": {
    "fields": { // 指定要高亮的字段
      "FIELD": {
        "pre_tags": "<em>", // 用来标记高亮字段的前置标签
        "post_tags": "</em>" // 用来标记高亮字段的后置标签
      }
    }
  }
}
```

The screenshot shows the 'Styles' tab in the developer tools of a browser. It highlights a CSS rule for the 'em' element, which is used to style the search results. The rule is defined in the 'element.style' section of the 'Styles' tab.

```
344 GET /hotel/_search
345 {
346   "query": {
347     "match": {
348       "all": "如家"
349     }
350   },
351 },
352 "highlight": {
353   "fields": {
354     "name": {
355       "require_field_match": "false"
356     }
357   }
358 }
```

```
29   "name" : "如家酒店(北京良乡西路店)",
30   "pic" : "https://m.tuniucdn.com/fb3/s1/2n9c
31   /3Dpgf5RTTzrxpeN5y3RLnRVtxMEA_w200_h200_c1_t0.
32   "score" : 46,
33   "starName" : "二钻"
34 },
35 "highlight" : {
36   "name" : [
37     "<em>如家</em>酒店(北京良乡西路店)"
38   ]
39 },
40 },
41 {
42   "_index" : "hotel",
43   "type" : "doc".
```

## 搜索结果处理整体语法：

```
GET /hotel/_search
{
  "query": {
    "match": {
      "name": "如家"
    }
  },
  "from": 0, // 分页开始的位置
  "size": 20, // 期望获取的文档总数
  "sort": [
    { "price": "asc" }, // 普通排序
    {
      "_geo_distance": { // 距离排序
        "location" : "31.040699,121.618075",
        "order" : "asc",
        "unit" : "km"
      }
    }
  ],
  "highlight": {
    "fields": { // 高亮字段
      "name": {
        "pre_tags": "<em>", // 用来标记高亮字段的前置标签
        "post_tags": "</em>" // 用来标记高亮字段的后置标签
      }
    }
  }
}
```

## RestClient操作

我们通过match\_all来演示下基本的API，再看结果的解析：

```
@Test
void testMatchAll() throws IOException {
  // ... 略
  // 4. 解析结果
  SearchHits searchHits = response.getHits();
  // 4.1. 查询的总条数
  long total = searchHits.getTotalHits().value;
  // 4.2. 查询的结果数组
  SearchHit[] hits = searchHits.getHits();
  for (SearchHit hit : hits) {
    // 4.3. 得到source
    String json = hit.getSourceAsString();
    // 4.4. 打印
    System.out.println(json);
  }
}
```

The diagram illustrates the mapping between the Java code and the resulting Elasticsearch JSON response. It shows three main components: 1) A light green box containing the Java code for querying and printing results. 2) A light blue box containing the Elasticsearch JSON response. 3) Colored arrows indicating the flow from specific code snippets to their corresponding JSON fields. A teal arrow points from the line 'long total = searchHits.getTotalHits().value;' to the 'total' field in the JSON. A purple arrow points from the line 'SearchHit[] hits = searchHits.getHits();' to the 'hits' array in the JSON. A green arrow points from the line 'String json = hit.getSourceAsString();' to the '\_source' field within one of the hit objects.

```
"took" : 0,
"timed_out" : false,
"hits" : {
  "total" : {
    "value" : 2,
    "relation" : "eq"
  },
  "max_score" : 1.0,
  "hits" : [
    {
      "_index" : "heima",
      "_type" : "_doc",
      "_id" : "1",
      "_score" : 1.0,
      "_source" : {
        "info" : "Java讲师",
        "name" : "赵云",
      }
    },
    ...
  ]
}
```

## 精确查询

精确查询常见的有term查询和range查询，同样利用QueryBuilders实现：

```
// 词条查询
QueryBuilders.termQuery("city", "杭州");
// 范围查询
QueryBuilders.rangeQuery("price").gte(100).lte(150);
```

```
GET /hotel/_search
{
  "query": {
    "term": {
      "city": "杭州"
    }
  }
}
GET /hotel/_search
{
  "query": {
    "range": {
      "price": { "gte": 100, "lte": 150 }
    }
  }
}
```

## 复合查询-boolean query

精确查询常见的有term查询和range查询，同样利用QueryBuilders实现：

```
// 创建布尔查询
BoolQueryBuilder boolQuery = QueryBuilders.boolQuery();
// 添加must条件
boolQuery.must(QueryBuilders.termQuery("city", "杭州"));
// 添加filter条件
boolQuery.filter(QueryBuilders.rangeQuery("price").lte(250));
```

```
GET /hotel/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "term": { "city": "杭州" }
        }
      ],
      "filter": [
        {
          "range": {
            "price": { "lte": 250 }
          }
        }
      ]
    }
  }
}
```

```
@Test
void testBool() throws IOException {
    // 1. 准备Request
    SearchRequest request = new SearchRequest(...indices: "hotel");
    // 2. 准备DSL
    // 2.1. 准备BooleanQuery
    BoolQueryBuilder boolQuery = QueryBuilders.boolQuery();
    // 2.2. 添加term
    boolQuery.must(QueryBuilders.termQuery(name: "city", value: "杭州"));
    // 2.3. 添加range
    // boolQuery.filter(QueryBuilders.rangeQuery("price").lte(250)); I

    request.source().query(boolQuery);
    // 3. 发送请求
    SearchResponse response = client.search(request, RequestOptions.DEFAULT);
    // 4. 解析响应
    handleResponse(response);
}
```

## 结果处理

```
void testPageAndSort() throws IOException {
    // 页码, 每页大小
    int page = 2, size = 5;

    // 1. 准备Request
    SearchRequest request = new SearchRequest(...indices: "hotel");
    // 2. 准备DSL
    // 2.1.query
    request.source().query(QueryBuilders.matchAllQuery());
    // 2.2.排序 sort
    request.source().sort(name: "price", SortOrder.ASC);
    // 2.3.分页 from、size
    request.source().from((page - 1) * size).size(5); // 3. 发送请求
```

```
// 1.准备Request
SearchRequest request = new SearchRequest(...indices: "hotel");
// 2.准备DSL
// 2.1.query
request.source().query(QueryBuilders.matchQuery(name: "all", text: "如家"));
// 2.2.高亮
request.source().highlighter(new HighlightBuilder().field("name").requireFieldMatch(false));
// 3.发送请求
SearchResponse response = client.search(request, RequestOptions.DEFAULT);
// 4.解析响应
handleResponse(response);
```

高亮的结果处理相对比较麻烦：

The diagram illustrates the flow of data from Java code to JSON results. On the left, Java code is shown for handling search results:

```
// 获取source
HotelDoc hotelDoc = JSON.parseObject(hit.getSourceAsString(),
    HotelDoc.class);
// 处理高亮
Map<String, HighlightField> highlightFields =
hit.getHighlightFields();
if (!CollectionUtils.isEmpty(highlightFields)) {
    // 获取高亮字段结果
    HighlightField highlightField = highlightFields.get("name");
    if (highlightField != null) {
        // 取出高亮结果数组中的第一个，就是酒店名称
        String name = highlightField.getFragments()[0].string();
        hotelDoc.setName(name);
    }
}
```

An orange box highlights the line `HighlightField highlightField = highlightFields.get("name");`. A blue box highlights the entire `highlightFields` map. An orange arrow points from this code to the corresponding JSON field in the result on the right.

On the right, the resulting JSON document is shown:

```
{
    "_index": "hotel",
    "_type": ".doc",
    "_id": "339952837",
    "_score": 2.8947515,
    "_source": {
        "id": 339952837,
        "name": "如家酒店(北京良乡西路店)",
        "price": 159,
        "score": 46,
        "brand": "如家",
        "city": "北京",
        "location": "39.73167, 116.132482",
        "pic": "t0.jpg"
    },
    "highlight": {
        "name": [
            "<em>如家</em>酒店(北京良乡西路店)"
        ]
    }
}
```

A pink box highlights the `highlight` field in the JSON. An orange arrow points from the highlighted code in the Java code to this field in the JSON result.

## 距离排序

距离排序与普通字段排序有所差异，API如下：

```
// 价格排序  
request.source().sort("price", SortOrder.ASC);  
  
// 距离排序  
request.source().sort(SortBuilders  
    .geoDistanceSort("location", new GeoPoint("31.21, 121.5"))  
    .order(SortOrder.ASC)  
    .unit(DistanceUnit.KILOMETERS)  
);
```

```
GET /indexName/_search  
{  
  "query": {  
    "match_all": {}  
  },  
  "sort": [  
    {  
      "price": "asc"  
    },  
    {  
      "_geo_distance": {  
        "FIELD": "纬度, 经度",  
        "order": "asc",  
        "unit": "km"  
      }  
    }  
  ]  
}
```

## // 2. 算分函数查询

```
FunctionScoreQueryBuilder  
functionScoreQuery =  
QueryBuilders.functionScoreQuery(  
    boolQuery, // 原始查询, boolQuery  
    new  
FunctionScoreQueryBuilder.FilterFunctionBuilder[]{  
    // function数组  
    new  
FunctionScoreQueryBuilder.FilterFunctionBuilder(  
    QueryBuilders.termQuery("isAD", true), // 过滤条件  
    ScoreFunctionBuilders.weightFactorFunction(10) //  
    算分函数  
    )  
}  
);
```

## 组合查询-function score

Function Score查询可以控制文档的相关性算分，使用方式如下：

```
// 7.function score
FunctionScoreQueryBuilder functionScoreQueryBuilder =
    QueryBuilders.functionScoreQuery(
        QueryBuilders.matchQuery("name", "外滩"),
        new FunctionScoreQueryBuilder.FilterFunctionBuilder[]{
            new FunctionScoreQueryBuilder.FilterFunctionBuilder(
                QueryBuilders.termQuery("brand", "如家"),
                ScoreFunctionBuilders.weightFactorFunction(5)
            )
        }
    );
sourceBuilder.query(functionScoreQueryBuilder);
```

```
GET /hotel/_search
{
  "query": {
    "function_score": {
      "query": {
        "match": {
          "name": "外滩"
        }
      },
      "functions": [
        {
          "filter": {
            "term": {
              "brand": "如家"
            }
          },
          "weight": 5
        }
      ]
    }
  }
}
```

# ES 高级部分

## 数据聚合

### DSL

#### 聚合的分类

聚合 (aggregations) 可以实现对文档数据的统计、分析、运算。聚合常见的有三类：

- 桶 (Bucket) 聚合：用来对文档做分组
  - TermAggregation: 按照文档字段值分组
  - Date Histogram: 按照日期阶梯分组，例如一周为一组，或者一月为一组
- 度量 (Metric) 聚合：用以计算一些值，比如：最大值、最小值、平均值等
  - Avg: 求平均值
  - Max: 求最大值
  - Min: 求最小值
  - Stats: 同时求max、min、avg、sum等
- 管道 (pipeline) 聚合：其它聚合的结果为基础做聚合

TEXT数据类型不聚合



## DSL实现Bucket聚合

现在，我们要统计所有数据中的酒店品牌有几种，此时可以根据酒店品牌的名称做聚合。

类型为term类型，DSL示例：

```
GET /hotel/_search
{
  "size": 0, // 设置size为0，结果中不包含文档，只包含聚合结果
  "aggs": {
    "brandAgg": {
      "terms": {
        "field": "brand", // 参与聚合的字段
        "size": 20 // 希望获取的聚合结果数量
      }
    }
  }
}
```

## Bucket聚合-限定聚合范围

默认情况下，Bucket聚合是对索引库的所有文档做聚合，我们可以限定要聚合的文档范围，只要添加query条件即可：

```
GET /hotel/_search
{
  "query": {
    "range": {
      "price": {
        "lte": 200 // 只对200元以下的文档聚合
      }
    }
  },
  "size": 0,
  "aggs": {
    "brandAgg": {
      "terms": {
        "field": "brand",
        "size": 20
      }
    }
  }
}
```

## DSL实现Metrics 聚合

例如，我们要求获取每个品牌的用户评分的min、max、avg等值。

我们可以利用stats聚合：

```
GET /hotel/_search
{
  "size": 0,
  "aggs": {
    "brandAgg": {
      "terms": {
        "field": "brand",
        "size": 20
      },
      "aggs": { // 是brands聚合的子聚合，也就是分组后对每组分别计算
        "score_stats": { // 聚合名称
          "stats": { // 聚合类型，这里stats可以计算min、max、avg等
            "field": "score" // 聚合字段，这里是score
          }
        }
      }
    }
  }
}
```

Console   Search Profiler   Grok Debugger   Painless Lab BETA

History   Settings   Help

200 - OK   68 ms

```
1 GET _search
2 {
3   "query": {
4     | "match_all": {}
5   }
6 }
7 GET /hotel/_mapping
8 GET /hotel/_search
9 {
10   "size": 0,
11   "aggs": {
12     "jiudain": {
13       "terms": {
14         "field": "business",
15         "size": 10,
16         "order": {
17           "_count": "desc"
18         }
19       },
20       "aggs": {
21         "score": {
22           "stats": {
23             "field": "score"
24           }
25         }
26       }
27     }
28   }
29 }
30 }
31 }
32 }
33 }
34 }
35 }
36 }
37 }
38 }
39 }
40 }
41 }
```

```
17 "hits": [],
18 },
19 "aggregations": {
20   "jiudain": {
21     "doc_count_error_upper_bound": 0,
22     "sum_other_doc_count": 160,
23     "buckets": [
24       {
25         "key": "浦东金桥地区",
26         "doc_count": 5,
27         "score": {
28           "count": 5,
29           "min": 45.0,
30           "max": 47.0,
31           "avg": 46.0,
32           "sum": 230.0
33         }
34       },
35       {
36         "key": "科技园",
37         "doc_count": 5,
38         "score": {
39           "count": 5,
40           "min": 36.0,
41           "max": 47.0,
42         }
43     }
44   }
45 },
46 },
47 },
48 },
49 },
50 },
51 },
52 },
53 },
54 },
55 },
56 },
57 },
58 },
59 },
59 },
60 },
61 },
62 },
63 },
64 },
65 },
66 },
67 },
68 },
69 },
70 },
71 },
72 },
73 },
74 },
75 },
76 },
77 },
78 },
79 },
80 },
81 },
82 },
83 },
84 },
85 },
86 }
```

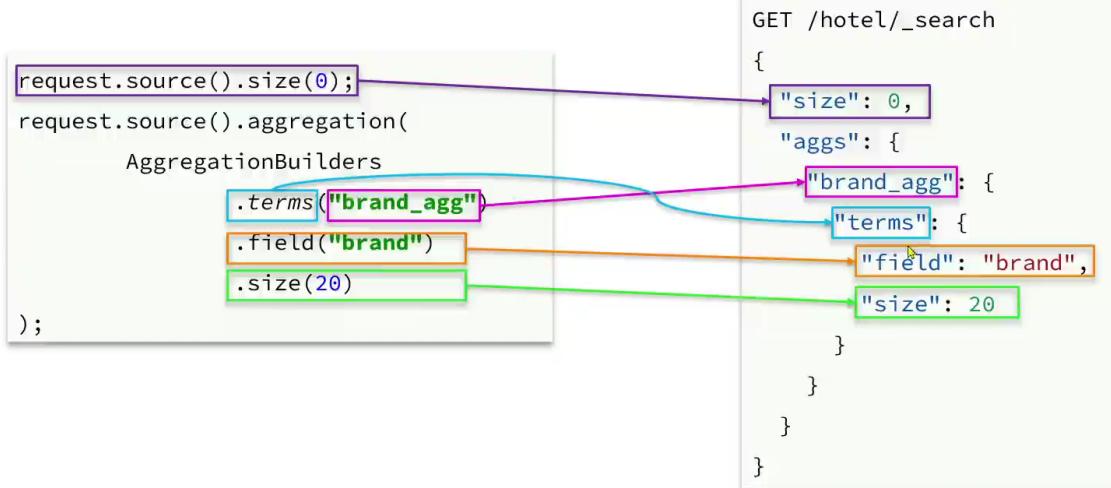
Click to send request

Click to send request

## RestClient

## RestAPI实现聚合

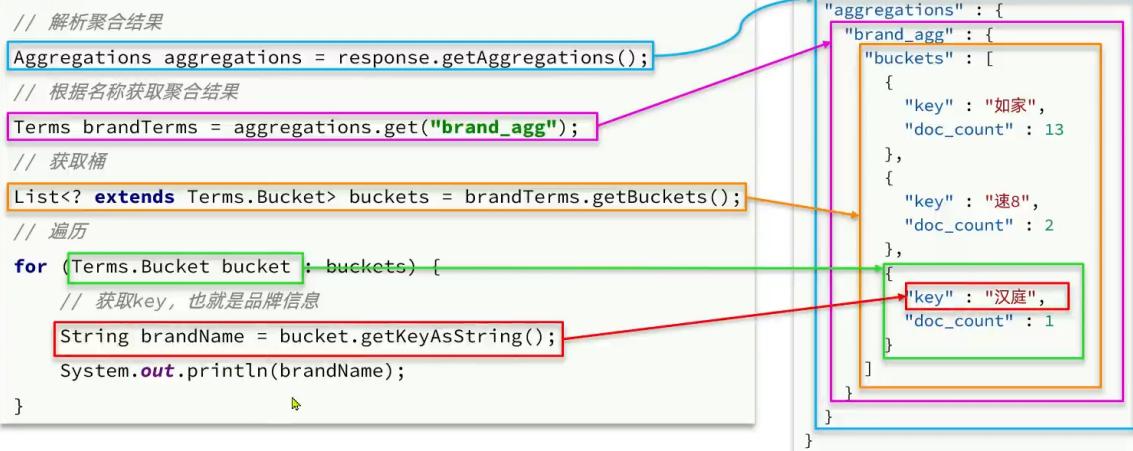
我们以品牌聚合为例，演示下Java的RestClient使用，先看请求组装：



```
@Test  
void testAggregation() throws IOException {  
    // 1. 准备Request  
    SearchRequest request = new SearchRequest(...indices: "hotel");  
    // 2. 准备DSL  
    // 2.1. 设置size  
    request.source().size(0);  
    // 2.2. 聚合  
    request.source().aggregation(AggregationBuilders  
        .terms(name: "brandAgg")  
        .field("brand")  
        .size(10)  
    );
```

## RestAPI实现聚合

再看下聚合结果解析



需求：搜索页面的品牌、城市等信息不应该是在页面写死，而是通过聚合索引库中的酒店数据得来的：

The screenshot shows a search interface for '黑马旅游' (www.theima.com). At the top is a logo and the website address. Below it is a search bar with an orange '搜索' (Search) button. Underneath the search bar is a section titled '全部结果' (All Results) containing four filter tables:

- 城市**: 上海, 北京, 深圳, 杭州
- 星级**: 四星, 五星, 二钻, 三钻, 四钻, 五钻
- 品牌**: 7天酒店, 如家, 速8, 皇冠假日, 华美达, 万怡, 喜来登, 万豪, 和颐
- 价格**: 100元以下, 100-300元, 300-600元, 600-1500元, 1500元以上

在IUserService中定义一个方法，实现对品牌、城市、星级的聚合，方法声明如下：

```
/**  
 * 查询城市、星级、品牌的聚合结果  
 * @return 聚合结果，格式：{"城市": ["上海", "北京"], "品牌": ["如家", "希尔顿"]}  
 */  
Map<String, List<String>> filters();
```

## 拼音分词器

### 安装

解压放置在es安装目录的plugin文件夹下面重启后测试是否成功

```
# 测试是否安装成功  
POST /_analyze  
{  
  "text": "拼音分词器",  
  "analyzer": "pinyin"  
}
```

### 结果

```
{  
  "tokens" : [  
    {  
      "token" : "pin",  
      "start_offset" : 0,  
      "end_offset" : 0,  
      "type" : "word",  
      "position" : 0  
    },
```

```
{  
    "token" : "pyfcq",  
    "start_offset" : 0,  
    "end_offset" : 0,  
    "type" : "word",  
    "position" : 0  
},  
{  
    "token" : "yin",  
    "start_offset" : 0,  
    "end_offset" : 0,  
    "type" : "word",  
    "position" : 1  
},  
{  
    "token" : "fen",  
    "start_offset" : 0,  
    "end_offset" : 0,  
    "type" : "word",  
    "position" : 2  
},  
{  
    "token" : "ci",  
    "start_offset" : 0,  
    "end_offset" : 0,  
    "type" : "word",  
    "position" : 3  
},  
{  
    "token" : "qi",  
    "start_offset" : 0,  
    "end_offset" : 0,  
    "type" : "word",  
    "position" : 4  
}  
]  
}
```

# 使用

## 自动补全

### 自定义分词器

elasticsearch中分词器（analyzer）的组成包含三部分：

- character filters: 在tokenizer之前对文本进行处理。例如删除字符、替换字符
- tokenizer: 将文本按照一定的规则切割成词条（term）。例如keyword，就是不分词；还有ik\_smart
- tokenizer filter: 将tokenizer输出的词条做进一步处理。例如大小写转换、同义词处理、拼音处理等



## 在官方文档中有很多的配置选项 详情

The screenshot shows a detailed configuration page for the Pinyin analyzer. The page contains a large amount of text describing various configuration options:

- keep\_first\_letter when this option enabled, eg: 刘德华 > ldh, default: true
- keep\_separate\_first\_letter when this option enabled, will keep first letters separately, eg: 刘德华 > l, d, h, default: false, NOTE: query result maybe too fuzziness due to term too frequency
- limit\_first\_letter\_length set max length of the first letter result, default: 16
- keep\_full\_pinyin when this option enabled, eg: 刘德华 > [ liu , de , hua ], default: true
- keep\_joined\_full\_pinyin when this option enabled, eg: 刘德华 > [ liudehua ], default: false
- keep\_none\_chinese keep non chinese letter or number in result, default: true
- keep\_none\_chinese\_together keep non chinese letter together, default: true, eg: DJ音乐家 -> DJ, yin, yue, jia, when set to false, eg: DJ音乐家 -> D, J, yin, yue, jia, NOTE: keep\_none\_chinese should be enabled first
- keep\_none\_chinese\_in\_first\_letter keep non Chinese letters in first letter, eg: 刘德华AT2016 -> ldhat2016, default: true
- keep\_none\_chinese\_in\_joined\_full\_pinyin keep non Chinese letters in joined full pinyin, eg: 刘德华2016 -> liudehua2016, default: false
- none\_chinese\_pinyin\_tokenize break non chinese letters into separate pinyin term if they are pinyin, default: true, eg: liudehuaalibaba13zhuanghan -> liu , de , hua , a , li , ba , 13 , zhuang , han , NOTE: keep\_none\_chinese and keep\_none\_chinese\_together should be enabled first
- keep\_original when this option enabled, will keep original input as well, default: false
- lowercase lowercase non Chinese letters, default: true
- trim\_whitespace default: true
- remove\_duplicated\_term when this option enabled, duplicated term will be removed to save index, eg: de的 > de , default: false, NOTE: position related query maybe influenced
- ignore\_pinyin\_offset after 6.0, offset is strictly constrained, overlapped tokens are not allowed, with this parameter, overlapped token will be allowed by ignore offset, please note, all position related query or highlight will become incorrect, you should use multi fields and specify different settings for different query purpose. if you need offset, please set it to false. default: true.

1.Create a index with custom pinyin analyzer

## 自定义分词器

我们可以在创建索引库时，通过settings来配置自定义的analyzer（分词器）：

```
PUT /test
```

```
{  
  "settings": {  
    "analysis": {  
      "analyzer": { // 自定义分词器  
        "my_analyzer": { // 分词器名称  
          "tokenizer": "ik_max_word",  
          "filter": "pinyin"  
        }  
      }  
    }  
  }  
}
```



```
PUT /test
```

```
{  
  "settings": {  
    "analysis": {  
      "analyzer": { // 自定义分词器  
        "my_analyzer": { // 分词器名称  
          "tokenizer": "ik_max_word",  
          "filter": "py"  
        }  
      },  
      "filter": { // 自定义tokenizer filter  
        "py": { // 过滤器名称  
          "type": "pinyin", // 过滤器类型，这里是pinyin  
          "keep_full_pinyin": false,  
          "keep_joined_full_pinyin": true,  
          "keep_original": true,  
          "limit_first_letter_length": 16,  
          "remove_duplicated_term": true,  
          "none_chinese_pinyin_tokenize": false  
        }  
      }  
    }  
  }  
}
```

// 自定义拼音分词器

```
PUT /test
```

```
{
```

```
  "settings": {  
    "analysis": {  
      "analyzer": {  
        "my_analyzer": {  
          "tokenizer": "ik_max_word",  
          "filter": "py"  
        }  
      },  
      "filter": {  
        "py": {  
          "type": "pinyin",  
          "keep_full_pinyin": false,  
          "keep_joined_full_pinyin": true,  
          "keep_original": true,  
          "limit_first_letter_length": 16,  
          "remove_duplicated_term": true,  
          "none_chinese_pinyin_tokenize": false  
        }  
      }  
    }  
  }  
}
```

```
POST /test/_doc/1
```

```
{  
    "id": 1,  
    "name": "狮子"  
}  
POST /test/_doc/2  
{  
    "id": 2,  
    "name": "虱子"  
}  
  
GET /test/_search  
{  
    "query": {  
        "match": {  
            "name": "掉入狮子笼咋办"  
        }  
    }  
}  
  
// 自动补全的索引库  
PUT test  
{  
    "mappings": {  
        "properties": {  
            "title":{  
                "type": "completion"  
            }  
        }  
    }  
}  
// 示例数据  
POST test/_doc  
{  
    "title": ["Sony", "WH-1000XM3"]  
}  
POST test/_doc  
{  
    "title": ["SK-II", "PITERA"]  
}  
POST test/_doc  
{
```

```
        "title": ["Nintendo", "switch"]
    }

// 自动补全查询
POST /test/_search
{
    "suggest": {
        "title_suggest": {
            "text": "s", // 关键字
            "completion": {
                "field": "title", // 补全字段
                "skip_duplicates": true, // 跳过重复的
                "size": 10 // 获取前10条结果
            }
        }
    }
}

// 酒店数据索引库
PUT /hotel
{
    "settings": {
        "analysis": {
            "analyzer": {
                "text_analyzer": {
                    "tokenizer": "ik_max_word",
                    "filter": "py"
                },
                "completion_analyzer": {
                    "tokenizer": "keyword",
                    "filter": "py"
                }
            },
            "filter": {
                "py": {
                    "type": "pinyin",
                    "keep_full_pinyin": false,
                    "keep_joined_full_pinyin": true,
                    "keep_original": true,
                    "limit_first_letter_length": 16,
                    "remove_duplicated_term": true,

```

```
        "none_chinese_pinyin_tokenize": false
    }
}
},
{
  "mappings": {
    "properties": {
      "id": {
        "type": "keyword"
      },
      "name": {
        "type": "text",
        "analyzer": "text_analyzer",
        "search_analyzer": "ik_smart",
        "copy_to": "all"
      },
      "address": {
        "type": "keyword",
        "index": false
      },
      "price": {
        "type": "integer"
      },
      "score": {
        "type": "integer"
      },
      "brand": {
        "type": "keyword",
        "copy_to": "all"
      },
      "city": {
        "type": "keyword"
      },
      "starName": {
        "type": "keyword"
      },
      "business": {
        "type": "keyword",
        "copy_to": "all"
      },
      "location": {
        "type": "geo_point"
      }
    }
  }
}
```

```

        "type": "geo_point"
    },
    "pic": {
        "type": "keyword",
        "index": false
    },
    "all": {
        "type": "text",
        "analyzer": "text_analyzer",
        "search_analyzer": "ik_smart"
    },
    "suggestion": {
        "type": "completion",
        "analyzer": "completion_analyzer"
    }
}
}
}
}

```

```

6  |     "analysis": {
7  |         "analyzer": {
8  |             "my_analyzer": {
9  |                 "tokenizer": "ik_max_word",
10 |                 "filter": "py"
11 |             }
12 |         },
13 |         "filter": {
14 |             "py": {
15 |                 "type": "pinyin",
16 |                 "keep_full_pinyin": false,
17 |                 "keep_joined_full_pinyin": true,
18 |                 "keep_original": true,
19 |                 "limit_first_letter_length": 16,
20 |                 "remove_duplicated_term": true,
21 |                 "none_chinese_pinyin_tokenize": false
22 |             }
23 |         }
24 |     }
25 |
26 | }
27 GET /test/_analyze
28 {
29     "text": "你好",
30     "analyzer": "my_analyzer"
31 }

```

```

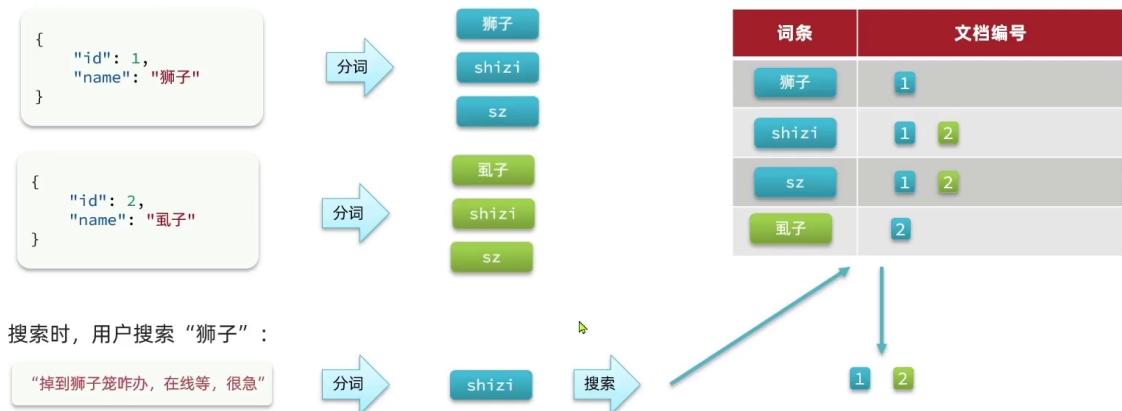
2  | {
3  |     "tokens": [
4  |         {
5  |             "token": "你好",
6  |             "start_offset": 0,
7  |             "end_offset": 2,
8  |             "type": "CN_WORD",
9  |             "position": 0
10 |         },
11 |         {
12 |             "token": "nihao",
13 |             "start_offset": 0,
14 |             "end_offset": 2,
15 |             "type": "CN_WORD",
16 |             "position": 0
17 |         },
18 |         {
19 |             "token": "nh",
20 |             "start_offset": 0,
21 |             "end_offset": 2,
22 |             "type": "CN_WORD",
23 |             "position": 0
24 |         }
25 |     ]
26 |
27

```

## 自定义分词器

拼音分词器适合在创建倒排索引的时候使用，但不能在搜索的时候使用。

创建倒排索引时：



## 自定义分词器

因此字段在创建倒排索引时应该用my\_analyzer分词器；字段在搜索时应该使用ik\_smart分词器；

```
PUT /test
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_analyzer": {
          "tokenizer": "ik_max_word", "filter": "py"
        }
      },
      "filter": {
        "py": { ... }
      }
    }
  },
  "mappings": {
    "properties": {
      "name": {
        "type": "text",
        "analyzer": "my_analyzer",
        "search_analyzer": "ik_smart"
      }
    }
  }
}
```

## 如何使用拼音分词器？

- ① 下载pinyin分词器
- ② 解压并放到elasticsearch的plugin目录
- ③ 重启即可

## 如何自定义分词器？

- ① 创建索引库时，在settings中配置，可以包含三部分
- ② character filter
- ③ tokenizer
- ④ filter

## 拼音分词器注意事项？

- 为了避免搜索到同音字，搜索时不要使用拼音分词器

# 自动补全

## completion suggester查询

elasticsearch提供了[Completion Suggester](#)查询来实现自动补全功能。这个查询会匹配以用户输入内容开头的词条并返回。为了提高补全查询的效率，对于文档中字段的类型有一些约束：

- 参与补全查询的字段必须是completion类型。
- 字段的内容一般是用来补全的多个词条形成的数组。

```
// 创建索引库
PUT test
{
  "mappings": {
    "properties": {
      "title": {
        "type": "completion"
      }
    }
  }
}

// 示例数据
POST test/_doc
{
  "title": ["Sony", "WH-1000XM3"]
}
POST test/_doc
{
  "title": ["SK-II", "PITERA"]
}
POST test/_doc
{
  "title": ["Nintendo", "switch"]
}
```

# 案例索引映射

```
// 酒店数据索引库
PUT /hotel
{
  "settings": {
    "analysis": {
      "analyzer": {
```

```
        "text_analyzer": {
            "tokenizer": "ik_max_word",
            "filter": "py"
        },
        "completion_analyzer": {
            "tokenizer": "keyword",
            "filter": "py"
        }
    },
    "filter": {
        "py": {
            "type": "pinyin",
            "keep_full_pinyin": false,
            "keep_joined_full_pinyin": true,
            "keep_original": true,
            "limit_first_letter_length": 16,
            "remove_duplicated_term": true,
            "none_chinese_pinyin_tokenize": false
        }
    }
},
{
    "mappings": {
        "properties": {
            "id": {
                "type": "keyword"
            },
            "name": {
                "type": "text",
                "analyzer": "text_analyzer",
                "search_analyzer": "ik_smart",
                "copy_to": "all"
            },
            "address": {
                "type": "keyword",
                "index": false
            },
            "price": {
                "type": "integer"
            },
            "score": {

```

```
        "type": "integer"
    },
    "brand": {
        "type": "keyword",
        "copy_to": "all"
    },
    "city": {
        "type": "keyword"
    },
    "starName": {
        "type": "keyword"
    },
    "business": {
        "type": "keyword",
        "copy_to": "all"
    },
    "location": {
        "type": "geo_point"
    },
    "pic": {
        "type": "keyword",
        "index": false
    },
    "all": {
        "type": "text",
        "analyzer": "text_analyzer",
        "search_analyzer": "ik_smart"
    },
    "suggestion": {
        "type": "completion",
        "analyzer": "completion_analyzer"
    }
}
}
```

## RestAPI实现自动补全

再来看结果解析：

```
// 4. 处理结果
Suggest suggest = response.getSuggest();
// 4.1. 根据名称获取补全结果
CompletionSuggestion suggestion = suggest.getSuggestion("hotelSuggestion");
// 4.2. 获取options并遍历
for (CompletionSuggestion.Entry.Option option : suggestion.getOptions()) {
    // 4.3. 获取一个option中的text, 也就是补全的词条
    String text = option.getText().string();
    System.out.println(text);
}
```

```
{
    "took" : 1,
    "timed_out" : false,
    "_shards" : {...},
    "hits" : {...},
    "suggest" : [
        "title_suggest" : [
            {
                "text" : "s",
                "offset" : 0,
                "length" : 1,
                "options" : [
                    {
                        "text" : "SK-II",
                        ...
                    },
                    {
                        "text" : "Sony",
                        ...
                    },
                    {
                        "text" : "switch",
                        ...
                    }
                ]
            }
        ]
    }
}
```

## RestAPI实现自动补全

再来看结果解析：

```
// 4. 处理结果
Suggest suggest = response.getSuggest();
// 4.1. 根据名称获取补全结果
CompletionSuggestion suggestion = suggest.getSuggestion("hotelSuggestion");
// 4.2. 获取options并遍历
for (CompletionSuggestion.Entry.Option option : suggestion.getOptions()) {
    // 4.3. 获取一个option中的text, 也就是补全的词条
    String text = option.getText().string();
    System.out.println(text);
}
```

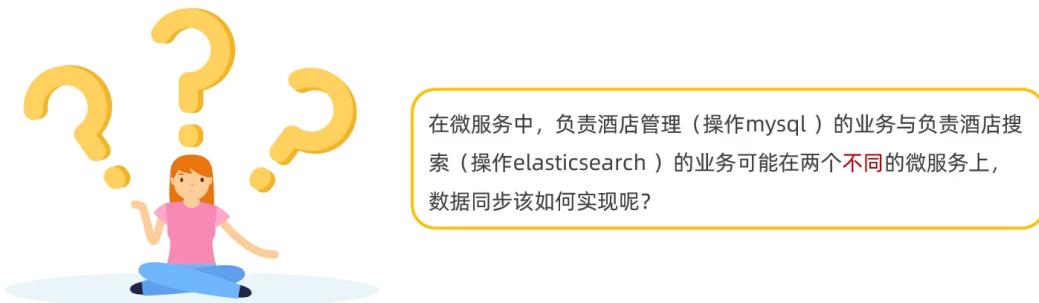
```
{
    "took" : 1,
    "timed_out" : false,
    "_shards" : {...},
    "hits" : {...},
    "suggest" : [
        "title_suggest" : [
            {
                "text" : "s",
                "offset" : 0,
                "length" : 1,
                "options" : [
                    {
                        "text" : "SK-II",
                        ...
                    },
                    {
                        "text" : "Sony",
                        ...
                    },
                    {
                        "text" : "switch",
                        ...
                    }
                ]
            }
        ]
    }
}
```

## 数据同步

### 提出问题与解决方案

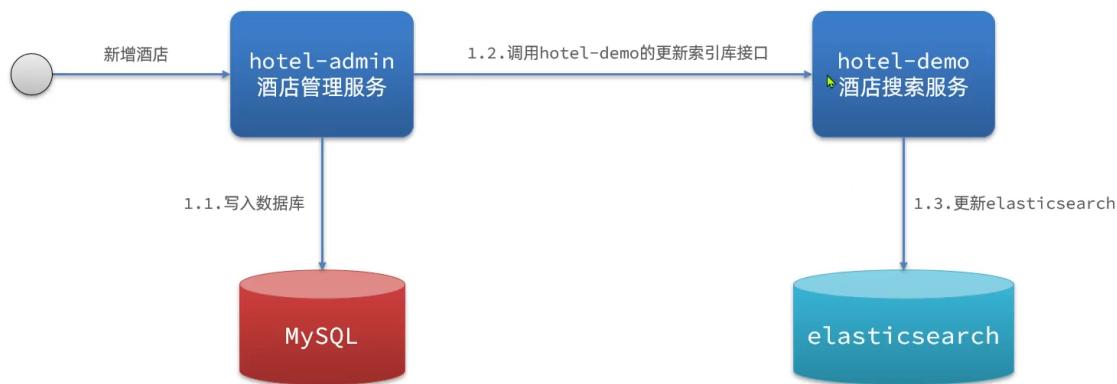
## 数据同步问题分析

elasticsearch中的酒店数据来自于mysql数据库，因此mysql数据发生改变时，elasticsearch也必须跟着改变，这个就是elasticsearch与mysql之间的**数据同步**。



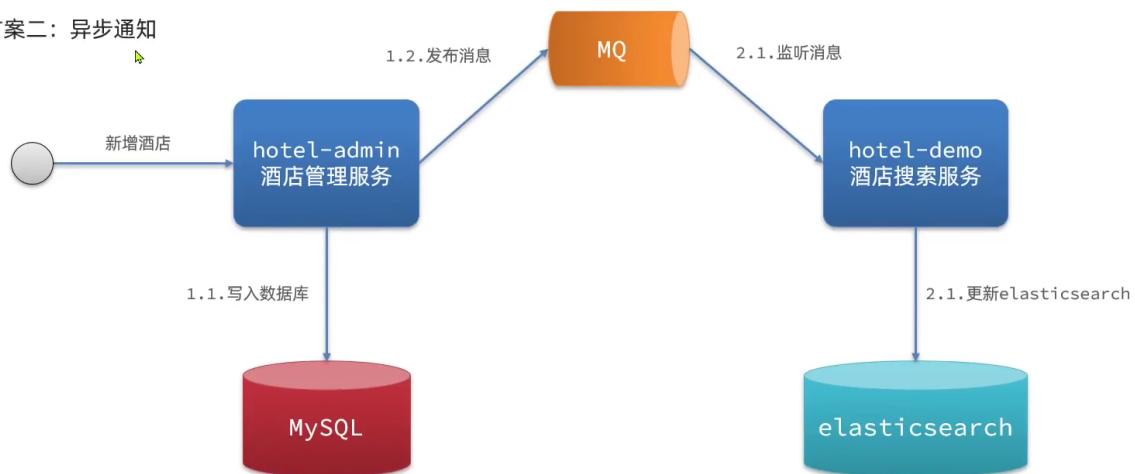
## 数据同步问题分析

### 方案一：同步调用



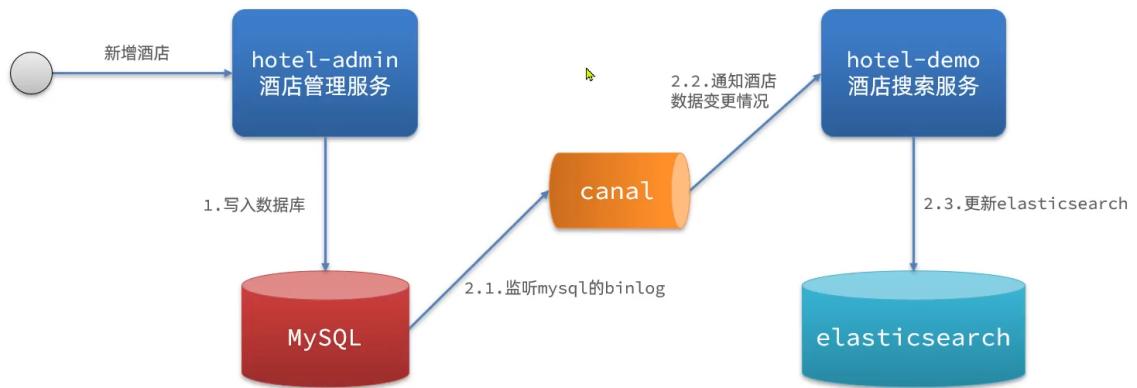
## 数据同步问题分析

### 方案二：异步通知



## 数据同步问题分析

方案三：监听binlog



### 方式一：同步调用

- 优点：实现简单，粗暴
- 缺点：业务耦合度高

### 方式二：异步通知

- 优点：低耦合，实现难度一般
- 缺点：依赖mq的可靠性

### 方式三：监听binlog

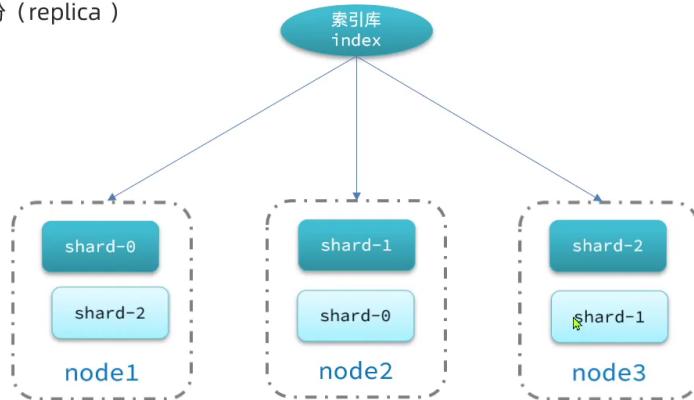
- 优点：完全解除服务间耦合
- 缺点：开启binlog增加数据库负担、实现复杂度高

集群

## ES集群结构

单机的elasticsearch做数据存储，必然面临两个问题：海量数据存储问题、单点故障问题。

- 海量数据存储问题：将索引库从逻辑上拆分为N个分片（shard），存储到多个节点
- 单点故障问题：将分片数据在不同节点备份（replica）



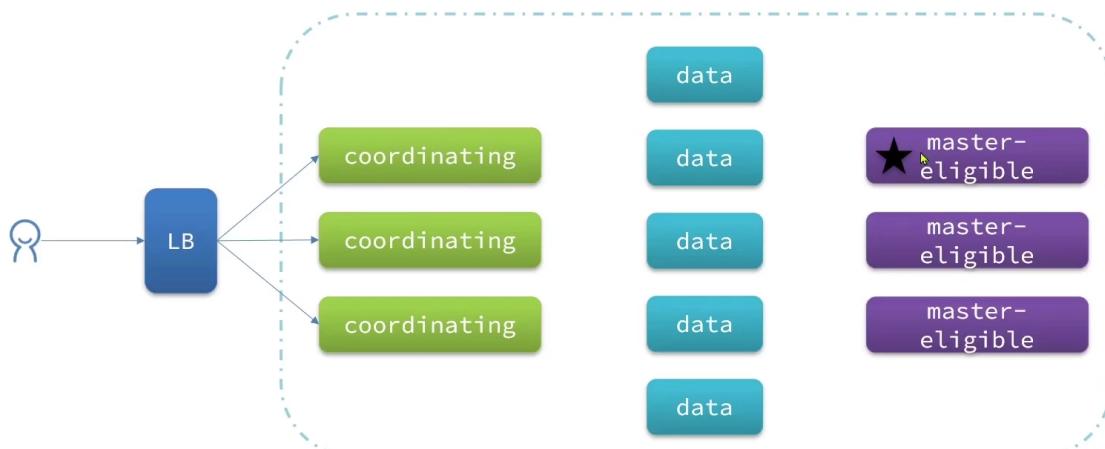
## ES集群的节点角色

elasticsearch中集群节点有不同的职责划分：

节点类型	配置参数	默认值	节点职责
master eligible	node.master	true	备选主节点：主节点可以管理和记录集群状态、决定分片在哪个节点、处理创建和删除索引库的请求
data	node.data	true	数据节点：存储数据、搜索、聚合、CRUD
ingest	node.ingest	true	数据存储之前的预处理
coordinating	上面3个参数都为false 则为coordinating节点	无	路由请求到其它节点 合并其它节点处理的结果，返回给用户

## ES集群的分布式查询

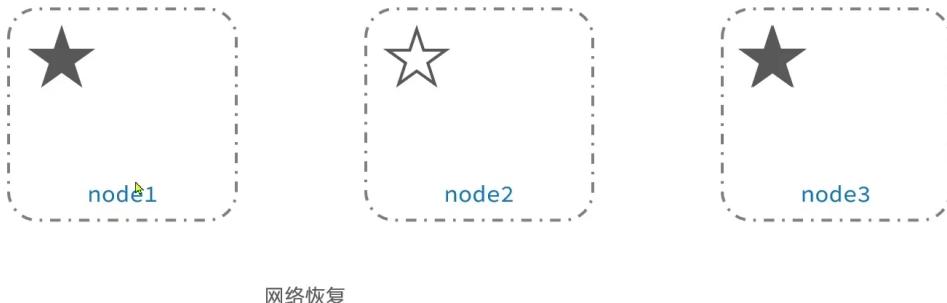
elasticsearch中的每个节点角色都有自己不同的职责，因此建议集群部署时，每个节点都有独立的角色。



## ES集群的脑裂

默认情况下，每个节点都是master eligible节点，因此一旦master节点宕机，其它候选节点会选举一个成为主节点。当主节点与其他节点网络故障时，可能发生脑裂问题。

为了避免脑裂，需要要求选票超过  $(\text{eligible节点数量} + 1) / 2$  才能当选为主，因此eligible节点数量最好是奇数。对应配置项是discovery.zen.minimum\_master\_nodes，在es7.0以后，已经成为默认配置，因此一般不会发生脑裂问题



master eligible节点的作用是什么？

- 参与集群选主
- 主节点可以管理集群状态、管理分片信息、处理创建和删除索引库的请求

data节点的作用是什么？

- 数据的CRUD

coordinator节点的作用是什么？

- 路由请求到其它节点
- 合并查询到的结果，返回给用户

## ES集群的分布式存储

当新增文档时，应该保存到不同分片，保证数据均衡，那么coordinating node如何确定数据该存储到哪个分片呢？elasticsearch会通过hash算法来计算文档应该存储到哪个分片：

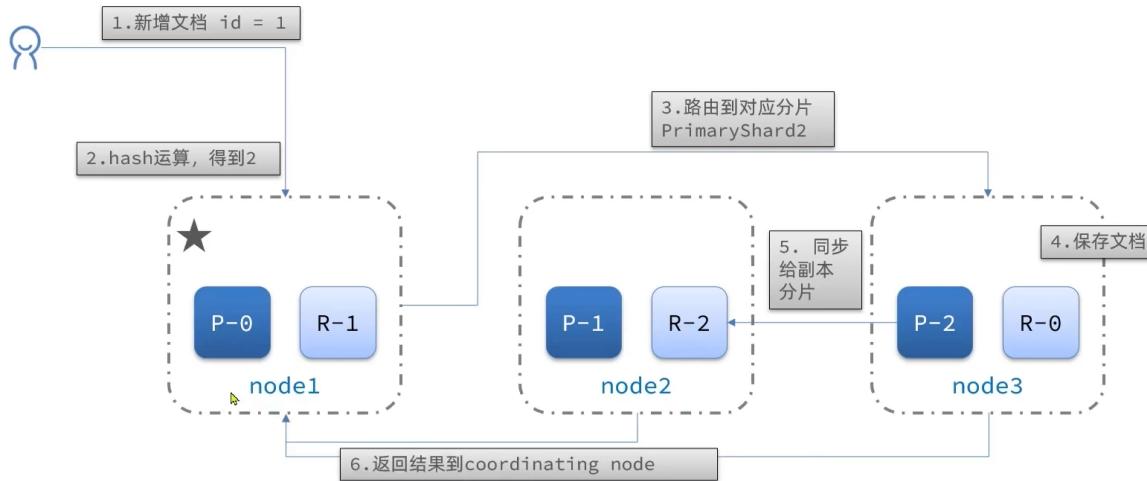
```
shard = hash(_routing) % number_of_shards
```

说明：

- \_routing默认是文档的id
- 算法与分片数量有关，因此索引库一旦创建，分片数量不能修改！

## ES集群的分布式存储

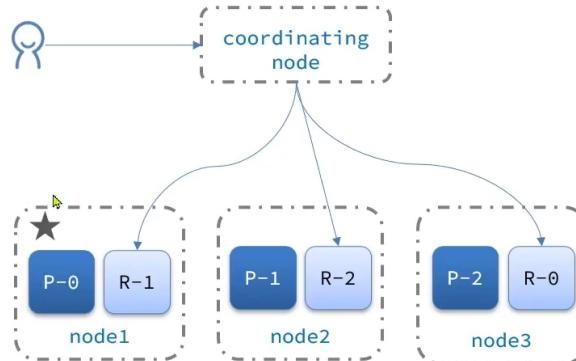
新增文档流程：



## ES集群的分布式查询

elasticsearch的查询分成两个阶段：

- scatter phase: 分散阶段, coordinating node会把请求分发到每一个分片
- gather phase: 聚集阶段, coordinating node汇总data node的搜索结果, 并处理为最终结果集返回给用户



## 分布式新增如何确定分片？

- coordinating node根据id做hash运算, 得到结果对shard数量取余, 余数就是对应的分片

## 分布式查询：

- 分散阶段: coordinating node将查询请求分发给不同分片
- 收集阶段: 将查询结果汇总到coordinating node , 整理并返回给用户

## 4.3. 创建索引库

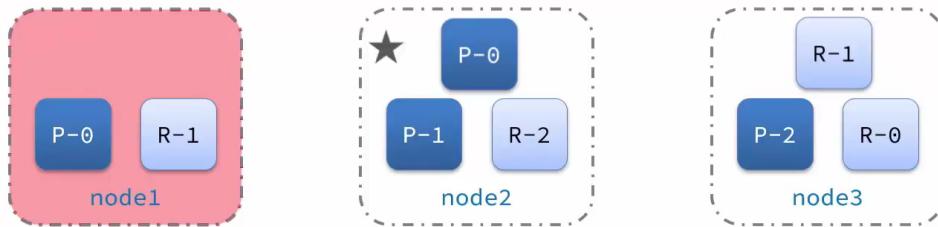
### 1) 利用kibana的DevTools创建索引库

在DevTools中输入指令：

```
PUT /itcast
{
  "settings": {
    "number_of_shards": 3, // 分片数量
    "number_of_replicas": 1 // 副本数量
  },
  "mappings": {
    "properties": {
      // mapping映射定义 ...
    }
  }
}
```

#### ES集群的故障转移

集群的master节点会监控集群中的节点状态，如果发现有节点宕机，会立即将宕机节点的分片数据迁移到其它节点，确保数据安全，这个叫做故障转移。



#### 故障转移：

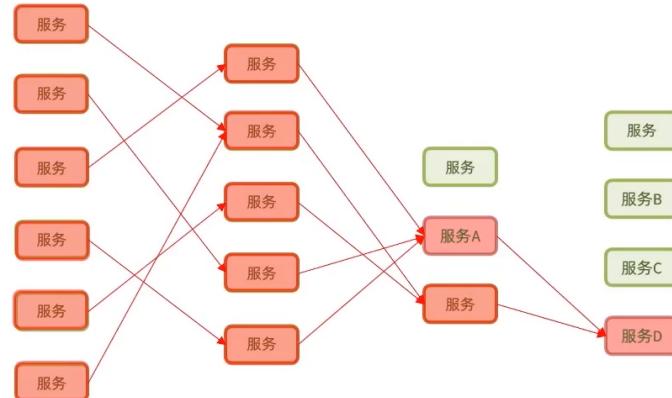
- master宕机后，EligibleMaster选举为新的主节点。
- master节点监控分片、节点状态，将故障节点上的分片转移到正常节点，确保数据安全。

Sentinel

# 介绍与安装

## 雪崩问题

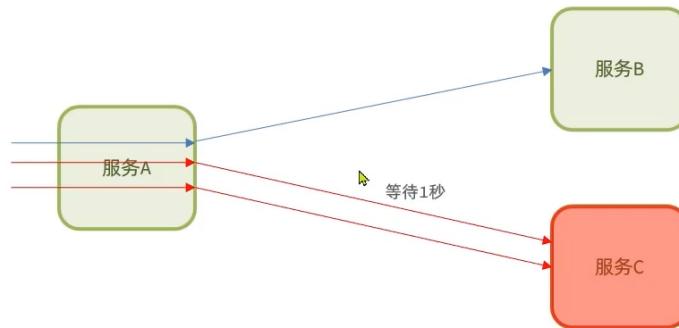
微服务调用链路中的某个服务故障，引起整个链路中的所有微服务都不可用，这就是雪崩。



## 雪崩问题

解决雪崩问题的常见方式有四种：

- 超时处理：设定超时时间，请求超过一定时间没有响应就返回错误信息，不会无休止等待



## 雪崩问题

解决雪崩问题的常见方式有四种：

- 超时处理：设定超时时间，请求超过一定时间没有响应就返回错误信息，不会无休止等待
- 舱壁模式：限定每个业务能使用的线程数，避免耗尽整个tomcat的资源，因此也叫线程隔离。
- 熔断降级：由**断路器**统计业务执行的异常比例，如果超出阈值则会**熔断**该业务，拦截访问该业务的一切请求。
- 流量控制：限制业务访问的QPS，避免服务因流量的突增而故障。



## 服务保护技术对比

	<b>Sentinel</b>	<b>Hystrix</b>
隔离策略	信号量隔离	线程池隔离/信号量隔离
熔断降级策略	基于慢调用比例或异常比例	基于失败比率
实时指标实现	滑动窗口	滑动窗口（基于 RxJava）
规则配置	支持多种数据源	支持多种数据源
扩展性	多个扩展点	插件的形式
基于注解的支持	支持	支持
限流	基于 QPS，支持基于调用关系的限流	有限的支持
流量整形	支持慢启动、匀速排队模式	不支持
系统自适应保护	支持	不支持
控制台	开箱即用，可配置规则、查看秒级监控、机器发现等	不完善
常见框架的适配	Servlet、Spring Cloud、Dubbo、gRPC 等	Servlet、Spring Cloud Netflix

## 介绍

### 认识Sentinel

Sentinel是阿里巴巴开源的一款微服务流量控制组件。官网地址：<https://sentinelguard.io/zh-cn/index.html>

Sentinel具有以下特征：

- 丰富的应用场景：**Sentinel 承接了阿里巴巴近 10 年的双十一大促流量的核心场景，例如秒杀（即突发流量控制在系统容量可以承受的范围）、消息削峰填谷、集群流量控制、实时熔断下游不可用应用等。
- 完备的实时监控：**Sentinel 同时提供实时的监控功能。您可以在控制台中看到接入应用的单台机器秒级数据，甚至 500 台以下规模的集群的汇总运行情况。
- 广泛的开源生态：**Sentinel 提供开箱即用的与其它开源框架/库的整合模块，例如与 Spring Cloud、Dubbo、gRPC 的整合。您只需要引入相应的依赖并进行简单的配置即可快速地接入 Sentinel。
- 完善的 SPI 扩展点：**Sentinel 提供简单易用、完善的 SPI 扩展接口。您可以通过实现扩展接口来快速地定制逻辑。例如定制规则管理、适配动态数据源等。

## 官网

### ##### 控制台安装

## 安装Sentinel控制台

sentinel官方提供了UI控制台，方便我们对系统做限流设置。大家可以在[GitHub](#)下载。课前资料提供了下载好的jar包：

 sentinel-dashboard-1.8.1.jar Executable Jar File 20,745 KB

1. 将其拷贝到一个你能记住的非中文目录，然后运行命令：

```
java -jar sentinel-dashboard-1.8.1.jar
```

2. 然后访问：localhost:8080 即可看到控制台页面，默认的账户和密码都是sentinel



## 安装Sentinel控制台

如果要修改Sentinel的默认端口、账户、密码，可以通过下列配置：

配置项	默认值	说明
server.port	8080	服务端口
sentinel.dashboard.auth.username	sentinel	默认用户名
sentinel.dashboard.auth.password	sentinel	默认密码

## 依赖与配置

### 整合sentinel的依赖

→

```
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
```

```
cloud:
  sentinel:
    transport:
      # sentinel 控制台信息
      dashboard: 192.168.0.104:8080
```

然后随便访问某个开启的断点，触发监控。

# 限流规则

## 簇点链路

簇点链路：就是项目内的调用链路，链路中**被监控**的每个接口就是一个资源。默认情况下sentinel会监控SpringMVC的每一个端点（Endpoint），因此SpringMVC的每一个端点（Endpoint）就是调用链路中的一个资源。

流控、熔断等都是**针对簇点链路中的资源**来设置的，因此我们可以点击对应资源后面的按钮来设置规则：

The screenshot shows the Sentinel UI interface for managing flow rules. On the left, there's a sidebar with navigation items like '首页', '实时监控', '簇点链路' (highlighted with a red box), '流控规则', '降级规则', '热点规则', '系统规则', '授权规则', '集群流控', and '机器列表'. The main content area is titled 'orderservice' and shows a table for '簇点链路'. The table has columns: 资源名 (Resource Name), 通过QPS (Throughput QPS), 拒绝QPS (Rejected QPS), 线程数 (Threads), 平均RT (Average RT), 分钟通过 (Throughput per minute), 分钟拒绝 (Rejected per minute), and 操作 (Operations). There are four rows: 'sentinel\_default\_context', 'sentinel\_spring\_web\_context', and two entries under '/order/{orderId}'. The last entry, '/order/{orderId}', has its '操作' (Operations) row highlighted with a red box, showing buttons for '+ 流控' (Flow Control), '+ 降级' (Degradation), '+ 热点' (Hotspot), and '+ 授权' (Authorization).

## 快速入门

点击资源/`/order/{orderId}`后面的流控按钮，就可以弹出表单。表单中可以添加流控规则，如下图所示：

The dialog box is titled '新增流控规则' (Add Flow Control Rule). It contains the following fields:

- 资源名 (Resource Name): /order/{orderId}
- 针对来源 (Target Source): default
- 阈值类型 (Threshold Type):
  - QPS
  - 线程数 (Threads)
- 单机阈值 (Single Machine Threshold): 1
- 是否集群 (Is Cluster):

At the bottom, there are three buttons: '新增并继续添加' (Add and Continue) (green), '新增' (Add) (green), and '取消' (Cancel) (red).

其含义是限制 `/order/{orderId}`这个资源的单机QPS为1，即每秒只允许1次请求，超出的请求会被**拦截并报错**。

## 流控规则入门案例

需求：给 /order/{orderId}这个资源设置流控规则， QPS不能超过 5。然后利用jmeter测试。

### 1. 设置流控规则：

资源名	/order/{orderId}
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 线程数
单机阈值	5

### 2. jmeter测试：

线程属性	
线程数：	20
2秒内发送20个请求	
Ramp-Up时间（秒）：	2
QPS是10	
循环次数	<input type="checkbox"/> 永远 1

## 流控模式

在添加限流规则时，点击高级选项，可以选择三种流控模式：

- 直接：统计当前资源的请求，触发阈值时对当前资源直接限流，也是默认的模式
- 关联：统计与当前资源相关的另一个资源，触发阈值时，对当前资源限流
- 链路：统计从指定链路访问到本资源的请求，触发阈值时，对指定链路限流

资源名	/order/{orderId}
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 线程数
单机阈值	2
是否集群	<input type="checkbox"/>
流控模式	<input checked="" type="radio"/> 直接 <input type="radio"/> 关联 <input type="radio"/> 链路

## 流控模式-关联

- 关联模式：统计与当前资源相关的另一个资源，触发阈值时，对当前资源限流
- 使用场景：比如用户支付时需要修改订单状态，同时用户要查询订单。查询和修改操作会争抢数据库锁，产生竞争。业务需求是有限支付和更新订单的业务，因此当修改订单业务触发阈值时，需要对查询订单业务限流。

资源名	/read
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 线程数
单机阈值	单机阈值
是否集群	<input type="checkbox"/>
流控模式	<input type="radio"/> 直接 <input checked="" type="radio"/> 关联 <input type="radio"/> 链路
关联资源	/write
流控效果	<input checked="" type="radio"/> 快速失败 <input type="radio"/> Warm Up <input type="radio"/> 排队等待

当/write资源访问量触发阈值时，就会对/read资源限流，避免影响/write资源。

## 流控模式-链路

链路模式：只针对从指定链路访问到本资源的请求做统计，判断是否超过阈值。

例如有两条请求链路：

- /test1 → /common
- /test2 → /common

如果只希望统计从/test2进入到/common的请求，则可以这样配置：

资源名	/common
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 线程数
	<input type="radio"/> 单机阈值 <input checked="" type="radio"/> 单机阈值
是否集群	<input type="checkbox"/>
流控模式	<input type="radio"/> 直接 <input type="radio"/> 关联 <input checked="" type="radio"/> 链路
入口资源	/test2

## 流控效果

流控效果是指请求达到流控阈值时应该采取的措施，包括三种：

- 快速失败：达到阈值后，新的请求会被立即拒绝并抛出FlowException异常。是默认的处理方式。
- warm up：预热模式，对超出阈值的请求同样是拒绝并抛出异常。但这种模式阈值会动态变化，从一个较小值逐渐增加到最大阈值。
- 排队等待：让所有的请求按照先后次序排队执行，两个请求的间隔不能小于指定时长

流控模式	<input type="radio"/> 直接 <input type="radio"/> 关联 <input checked="" type="radio"/> 链路
入口资源	/order/query
流控效果	<input checked="" type="radio"/> 快速失败 <input type="radio"/> Warm Up <input type="radio"/> 排队等待

## 流控效果-warm up

warm up也叫预热模式，是应对服务冷启动的一种方案。请求阈值初始值是 threshold / coldFactor，持续指定时长后，逐渐提高到threshold值。而coldFactor的默认值是3.

例如，我设置QPS的threshold为10，预热时间为5秒，那么初始阈值就是  $10 / 3$ ，也就是3，然后在5秒后逐渐增长到10.



## 流控效果-排队等待

当请求超过QPS阈值时，快速失败和warm up会拒绝新的请求并抛出异常。而排队等待则是让所有请求进入一个队列中，然后按照阈值允许的时间间隔依次执行。后来的请求必须等待前面执行完成，如果请求预期的等待时间超出最大时长，则会被拒绝。

例如：QPS = 5，意味着每200ms处理一个队列中的请求；timeout = 2000，意味着预期等待超过2000ms的请求会被拒绝并抛出异常



## 流控效果有哪些？

- 快速失败：QPS超过阈值时，拒绝新的请求
- warm up：QPS超过阈值时，拒绝新的请求；QPS阈值是逐渐提升的，可以避免冷启动时高并发导致服务宕机。
- 排队等待：请求会进入队列，按照阈值允许的时间间隔依次执行请求；如果请求预期等待时长大于超时时间，直接拒绝

## 防止一直点

## 热点参数限流

之前的限流是统计访问某个资源的所有请求，判断是否超过QPS阈值。而热点参数限流是分别统计参数值相同的请求，判断是否超过QPS阈值。



配置示例：

The configuration interface shows a resource named `hot` in `QPS 模式`. The `参数索引` is set to `0`, and the `单机阈值` is set to `5` with a window of `1 秒`.

代表的含义是：对hot这个资源的0号参数（第一个参数）做统计，每1秒相同参数值的请求数不能超过5

## 热点参数限流

在热点参数限流的高级选项中，可以对部分参数设置例外配置：

The configuration interface shows an exception configuration for the `long` type parameter index `0`. It lists two exceptions:

参数值	参数类型	限流阈值	操作
100	long	10	删除
101	long	15	删除

结合上一个配置，这里的含义是对0号的long类型参数限流，每1秒相同参数的QPS不能超过5，有两个例外：

- 如果参数值是100，则每1秒允许的QPS为10
- 如果参数值是101，则每1秒允许的QPS为15

### 注意

热点参数限流对默认的SpringMVC资源无效

所以要在MVC资源上添加 `@SentinelResource("ID")` 注解填写资源名称

`web-context-unify: false # 关闭context整合`

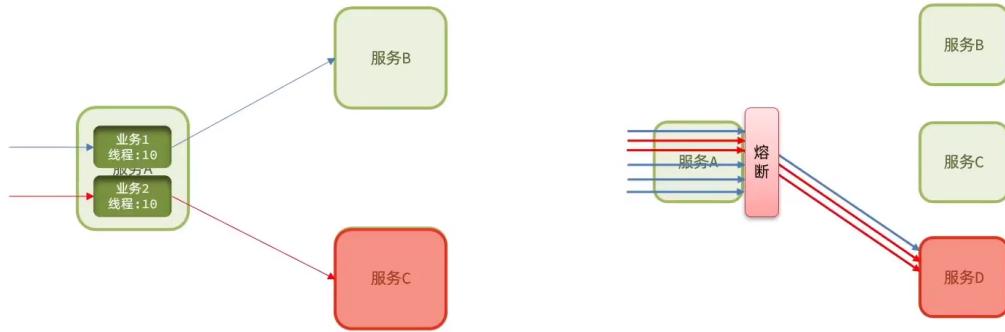
# 隔离和降级

## 整合

### 隔离和降级

虽然限流可以尽量避免因高并发而引起的服务故障，但服务还会因为其它原因而故障。而要将这些故障控制在一定范围，避免雪崩，就要靠线程隔离（舱壁模式）和熔断降级手段了。

不管是线程隔离还是熔断降级，都是对**客户端（调用方）**的保护。



### Feign整合Sentinel

SpringCloud中，微服务调用都是通过Feign来实现的，因此做客户端保护必须整合Feign和Sentinel。

#### 1. 修改OrderService的application.yml文件，开启Feign的Sentinel功能

```
feign:  
  sentinel:  
    enabled: true # 开启Feign的Sentinel功能
```

#### 2. 给FeignClient编写失败后的降级逻辑

- ① 方式一：FallbackClass，无法对远程调用的异常做处理
- ② 方式二：FallbackFactory，可以对远程调用的异常做处理，我们选择这种

```

# feign的日志配置
feign:
  client:
    config:
      default: # 这里可以写默认，也可以写某种具体的实例名
      称
        loggerLevel: BASIC
  httpclient:
    enabled: true # 开启连接池
    max-connections: 200 # 最大连接数
    max-connections-per-route: 50 #每个路径的最大连接
    数
    # 开启feign的sentinel功能
  sentinel:
    enabled: true

```

## Feign整合Sentinel

步骤一：在feing-api项目中定义类，实现FallbackFactory：

```

@Slf4j
public class UserClientFallbackFactory implements FallbackFactory<UserClient> {
  @Override
  public UserClient create(Throwable throwable) {
    // 创建UserClient接口实现类，实现其中的方法，编写失败降级的处理逻辑
    return new UserClient() {
      @Override
      public User findById(Long id) {
        // 记录异常信息
        log.error("查询用户失败", throwable);
        // 根据业务需求返回默认的数据，这里是空用户
        return new User();
      }
    };
  }
}

```

## Feign整合Sentinel

步骤二：在feing-api项目中的DefaultFeignConfiguration类中将UserClientFallbackFactory注册为一个Bean：

```

@Bean
public UserClientFallbackFactory userClientFallback(){
  return new UserClientFallback();
}

```

步骤三：在feing-api项目中的UserClient接口中使用UserClientFallbackFactory：

```

@FeignClient(value = "userservice", fallbackFactory = UserClientFallbackFactory.class)
public interface UserClient {
  @GetMapping("/user/{id}")
  User findById(@PathVariable("id") Long id);
}

```

```
/**  
 * 业务降级的处理逻辑  
 * 实现FallbackFactory<UserFeignClient> 接口 泛型为调用者  
 */  
  
@Slf4j  
public class UserClientFallbackFactory implements  
FallbackFactory<UserFeignClient> {  
    @Override  
    public UserFeignClient create(Throwable cause)  
{  
        return new UserFeignClient() {  
            @Override  
            public User getUser(Long id) {  
                // 编写降级逻辑  
                log.info("查询用户为空");  
                return new User();  
            }  
        };  
    }  
}  
  
@Bean  
public UserClientFallbackFactory  
userClientFallbackFactory(){  
    return new UserClientFallbackFactory();  
}  
  
// 日志级别指定当前目录有效  
@FeignClient(value = "userservice",  
// 指定降级所处理的字节码文件  
fallbackFactory = UserClientFallbackFactory.class)  
public interface UserFeignClient {  
    @GetMapping("/user/{id}")  
    User getUser(@PathVariable("id") Long id);  
}
```

注意

```
@EnableFeignClients(clients =  
{UserFeignClient.class},  
// 指定配置
```

```
defaultConfiguration = FeignConfig.class) //指定字节  
码文件
```

Sentinel支持的雪崩解决方案：

- 线程隔离（仓壁模式）
- 降级熔断

Feign整合Sentinel的步骤：

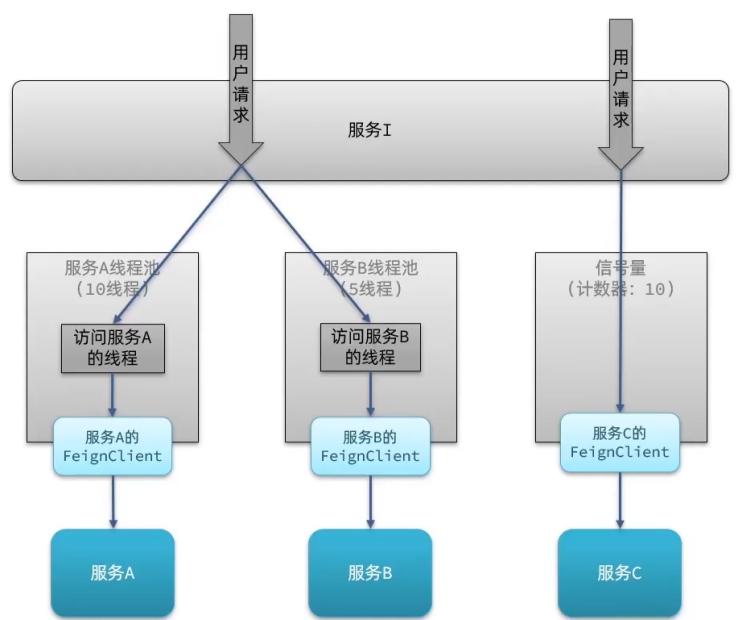
- 在application.yml中配置：feign.sentinel.enable=true
- 给FeignClient编写FallbackFactory并注册为Bean
- 将FallbackFactory配置到FeignClient

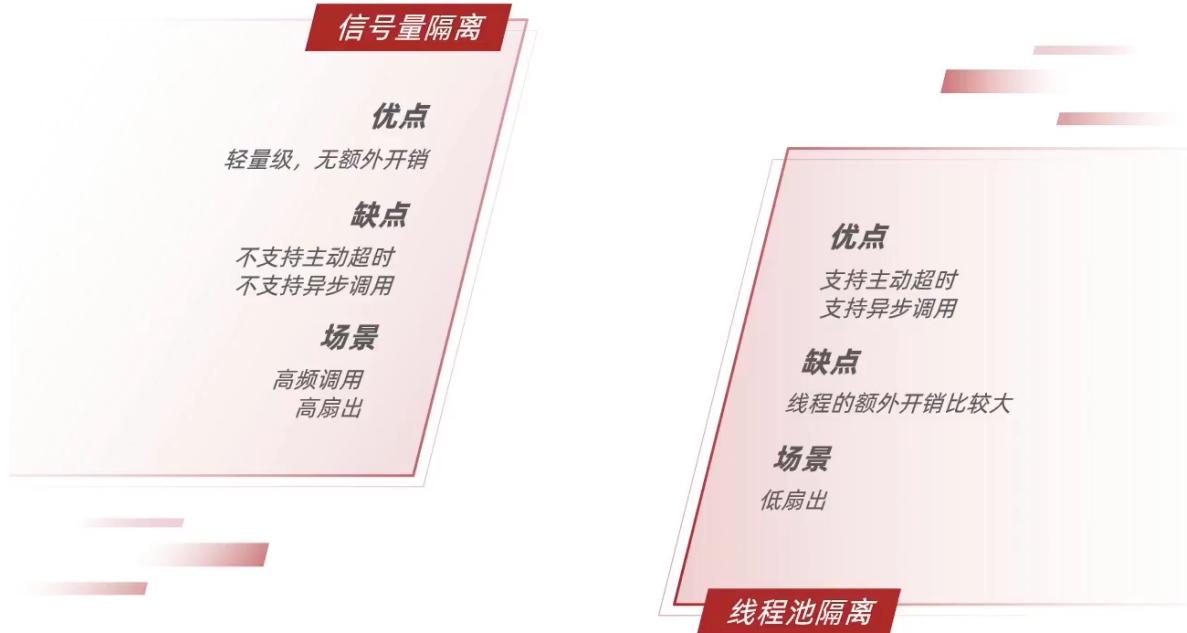
## 线程隔离

### 线程隔离

线程隔离有两种方式实现：

- 线程池隔离
- 信号量隔离（Sentinel默认采用）





## 线程隔离（舱壁模式）

在添加限流规则时，可以选择两种阈值类型：

资源名: /order/{orderId}

针对来源: default

阈值类型:  QPS  线程数      单机阈值: 5

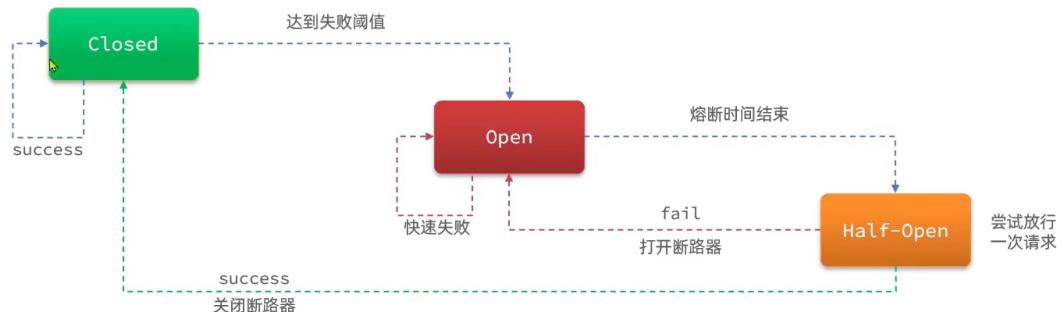
是否集群:

- QPS: 就是每秒的请求数，在快速入门中已经演示过
- 线程数: 是该资源能使用的tomcat线程数的最大值。也就是通过限制线程数量，实现**舱壁模式**。

## 熔断降级

### 熔断降级

熔断降级是解决雪崩问题的重要手段。其思路是由**断路器**统计服务调用的异常比例、慢请求比例，如果超出阈值则会**熔断**该服务。即拦截访问该服务的一切请求；而当服务恢复时，断路器会放行访问该服务的请求。



## 熔断策略-慢调用

断路器熔断策略有三种：慢调用、异常比例、异常数

- 慢调用：业务的响应时长（RT）大于指定时长的请求认定为慢调用请求。在指定时间内，如果请求数量超过设定的最小数量，慢调用比例大于设定的阈值，则触发熔断。例如：

The screenshot shows the 'Add Degradation Rule' dialog. The 'Resource Name' field is set to '/test'. Under the 'Slow-Call Strategy' section, the 'Slow-Call Ratio' radio button is selected. The 'Max RT' is set to 500 ms, and the 'Ratio Threshold' is 0.5. The 'Circuit Breaker Duration' is 5 seconds, and the 'Minimum Request Count' is 10. The 'Statistical Duration' is 10000 ms.

解读：RT超过500ms的调用是慢调用，统计最近10000ms内的请求，如果请求数量超过10次，并且慢调用比例不低于0.5，则触发熔断，熔断时长为5秒。然后进入half-open状态，放行一次请求做测试。

## 熔断策略-异常比例、异常数

断路器熔断策略有三种：慢调用、异常比例或异常数

- 异常比例或异常数：统计指定时间内的调用，如果调用次数超过指定请求数，并且出现异常的比例达到设定的比例阈值（或超过指定异常数），则触发熔断。例如：

The screenshot shows two side-by-side configurations of the 'Add Degradation Rule' dialog. Both have 'Resource Name' set to '/test'. The left configuration has 'Exception Ratio' selected, with 'Ratio Threshold' set to 0.4. The right configuration has 'Exception Number' selected, with 'Exception Number' set to 2. Both configurations share the same other parameters: 'Circuit Breaker Duration' is 5 seconds, 'Minimum Request Count' is 10, and 'Statistical Duration' is 1000 ms.

解读：统计最近1000ms内的请求，如果请求数量超过10次，并且异常比例不低于0.5，则触发熔断，熔断时长为5秒。然后进入half-open状态，放行一次请求做测试。

## 授权规则

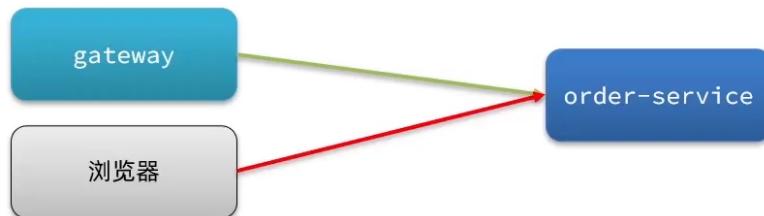
## 授权规则

授权规则可以对调用方的来源做控制，有白名单和黑名单两种方式。

- 白名单：来源 (origin) 在白名单内的调用者允许访问
- 黑名单：来源 (origin) 在黑名单内的调用者不允许访问

资源名	资源名称
流控应用	指调用方，多个调用方名称用半角英文逗号 (,) 分隔
授权类型	<input checked="" type="radio"/> 白名单 <input type="radio"/> 黑名单

例如，我们限定只允许从网关来的请求访问order-service，那么流控应用中就填写网关的名称



## 授权规则

Sentinel是通过RequestOriginParser这个接口的parseOrigin来获取请求的来源的。

```
public interface RequestOriginParser {  
    /**  
     * 从请求request对象中获取origin, 获取方式自定义  
     */  
    String parseOrigin(HttpServletRequest request);  
}
```

例如，我们尝试从request中获取一个名为origin的请求头，作为origin的值：

```
@Component  
public class HeaderOriginParser implements RequestOriginParser {  
    @Override  
    public String parseOrigin(HttpServletRequest request) {  
        String origin = request.getHeader("origin");  
        if(StringUtils.isEmpty(origin)){  
            return "blank";  
        }  
        return origin;  
    }  
}
```

## 授权规则

我们还需要在gateway服务中，利用网关的过滤器添加名为gateway的origin头：

```
spring:  
cloud:  
    gateway:  
        default-filters:  
            - AddRequestHeader=origin,gateway # 添加名为origin的请求头，值为gateway
```

## 授权规则

我们还需要在gateway服务中，利用网关的过滤器添加名为gateway的origin头：

```
spring:  
  cloud:  
    gateway:  
      default-filters:  
        - AddRequestHeader=origin,gateway # 添加名为origin的请求头，值为gateway
```

给/order/{orderId} 配置授权规则：

资源名	/order/{orderId}
流控应用	gateway
授权类型	<input checked="" type="radio"/> 白名单 <input type="radio"/> 黑名单

```
/**  
 * 监控  
 */  
// 放在容器中 不然不生效  
@Component  
public class HeaderOriginParser implements  
RequestOriginParser  
{  
    @Override  
    public String parseOrigin(HttpServletRequest  
httpServletRequest) {  
        String origin =  
httpServletRequest.getHeader("origin");  
        if (StringUtils.isEmpty(origin)) {  
            origin = "bank";  
        }  
        return origin;  
    }  
}
```

## 自定义异常结果

默认情况下，发生限流、降级、授权拦截时，都会抛出异常到调用方。如果要自定义异常时的返回结果，需要实现 BlockExceptionHandler接口：

```
public interface BlockExceptionHandler {  
    /**  
     * 处理请求被限流、降级、授权拦截时抛出的异常: BlockException  
     */  
    void handle(HttpServletRequest request, HttpServletResponse response, BlockException e) throws Exception;  
}
```

## 自定义异常结果

而BlockException包含很多个子类，分别对应不同的场景：

异常	说明
FlowException	限流异常
ParamFlowException	热点参数限流的异常
DegradeException	降级异常
AuthorityException	授权规则异常
SystemBlockException	系统规则异常

## 自定义异常结果

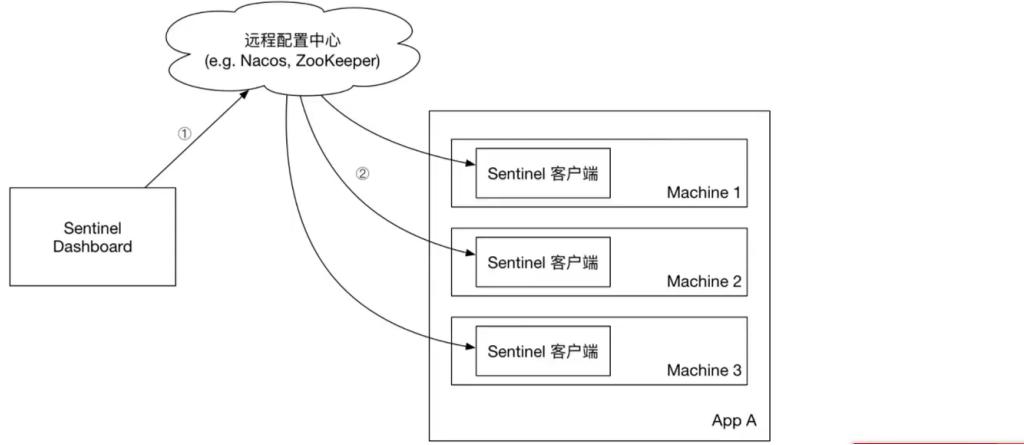
我们在order-service中定义类，实现BlockExceptionHandler接口：

```
@Component  
public class SentinelBlockHandler implements BlockExceptionHandler {  
    @Override  
    public void handle(  
        HttpServletRequest httpServletRequest,  
        HttpServletResponse httpServletResponse, BlockException e) throws Exception {  
        String msg = "未知异常";  
        int status = 429;  
        if (e instanceof FlowException) {  
            msg = "请求被限流了！";  
        } else if (e instanceof DegradeException) {  
            msg = "请求被降级了！";  
        } else if (e instanceof ParamFlowException) {  
            msg = "热点参数限流！";  
        } else if (e instanceof AuthorityException) {  
            msg = "请求没有权限！";  
            status = 401;  
        }  
        httpServletResponse.setContentType("application/json;charset=utf-8");  
        httpServletResponse.setStatus(status);  
        httpServletResponse.getWriter().println("{\"message\": \"" + msg + "\", \"status\": " + status + "}")  
    }  
}
```

# 规则持久化

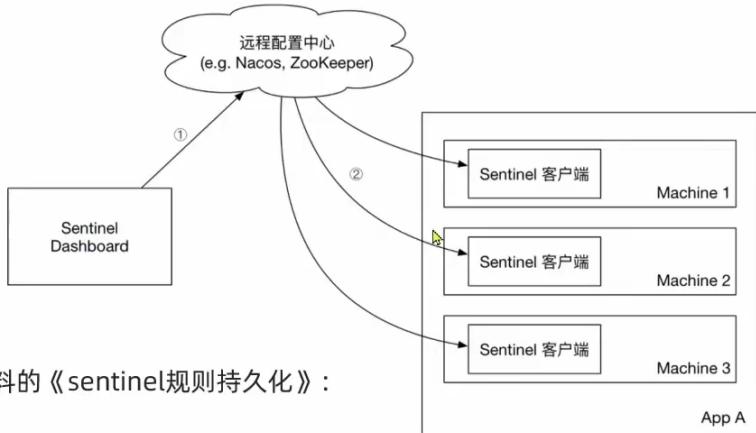
## 规则管理模式-push模式

push模式：控制台将配置规则推送到远程配置中心，例如Nacos。Sentinel客户端监听Nacos，获取配置变更的推送消息，完成本地配置更新。



## 实现push模式

push模式实现最为复杂，依赖于nacos，并且需要修改Sentinel控制台源码。

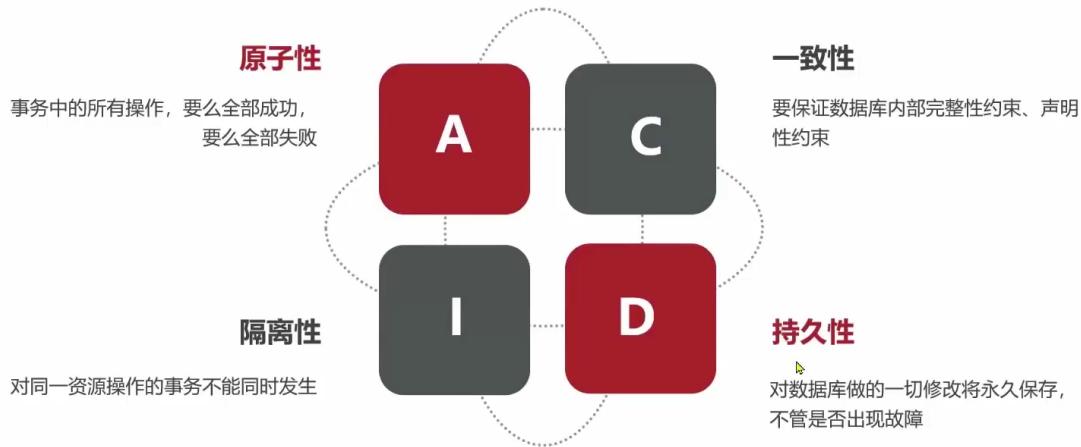


详细步骤可以参考课前资料的《sentinel规则持久化》：



# 分布式事务【基于seata】

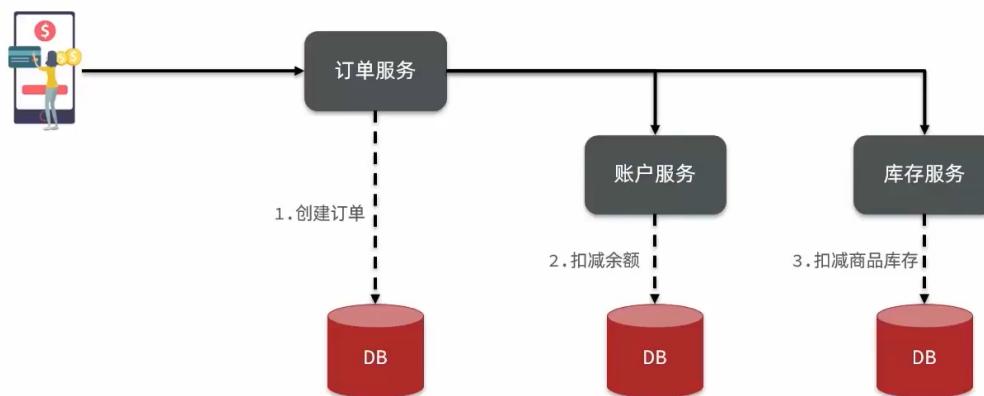
## 事务的ACID原则



## 分布式服务案例

微服务下单业务，在下单时会调用订单服务，创建订单并写入数据库。然后订单服务调用账户服务和库存服务：

- 账户服务负责扣减用户余额
- 库存服务负责扣减商品库存



## 理论基础

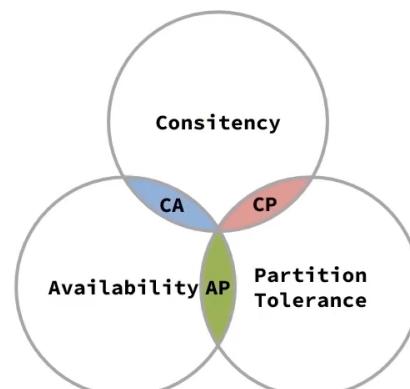
### CAP定理

1998年，加州大学的计算机科学家 Eric Brewer 提出，分布式系统有三个指标：

- Consistency (一致性)
- Availability (可用性)
- Partition tolerance<sup>分区容错性</sup> (分区容错性)

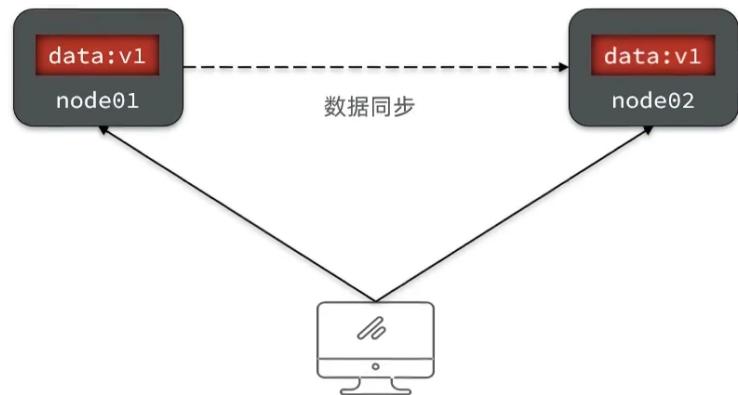
Eric Brewer 说，分布式系统无法同时满足这三个指标。

这个结论就叫做 CAP 定理。



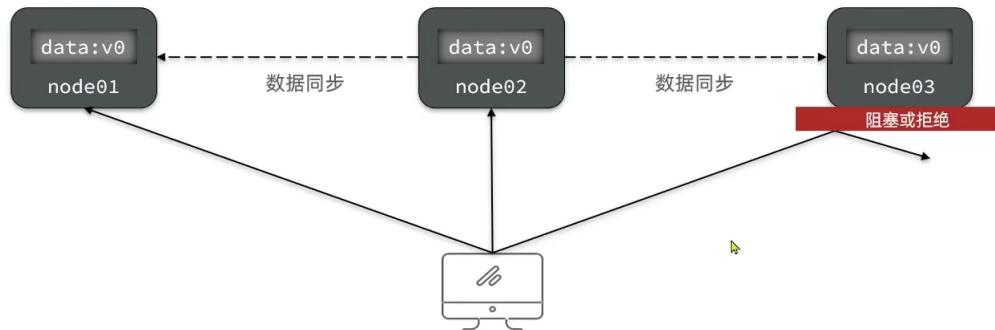
## CAP定理- Consistency

Consistency (一致性)：用户访问分布式系统中的任意节点，得到的数据必须一致



## CAP定理- Availability

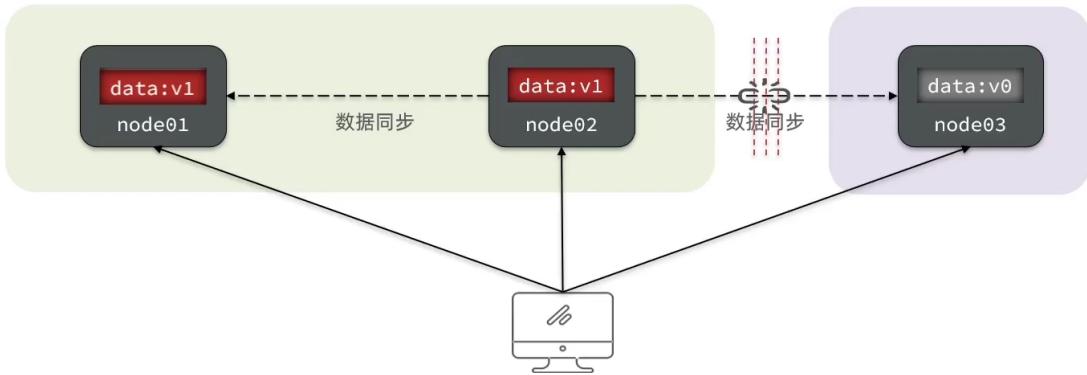
Availability (可用性)：用户访问集群中的任意健康节点，必须能得到响应，而不是超时或拒绝



## CAP定理-Partition tolerance

Partition (分区)：因为网络故障或其它原因导致分布式系统中的部分节点与其它节点失去连接，形成独立分区。

Tolerance (容错)：在集群出现分区时，整个系统也要持续对外提供服务



简述CAP定理内容？

- 分布式系统节点通过网络连接，一定会出现分区问题（P）
- 当分区出现时，系统的一致性（C）和可用性（A）就无法同时满足

思考：elasticsearch集群是CP还是AP？

## BASE理论

BASE理论是对CAP的一种解决思路，包含三个思想：

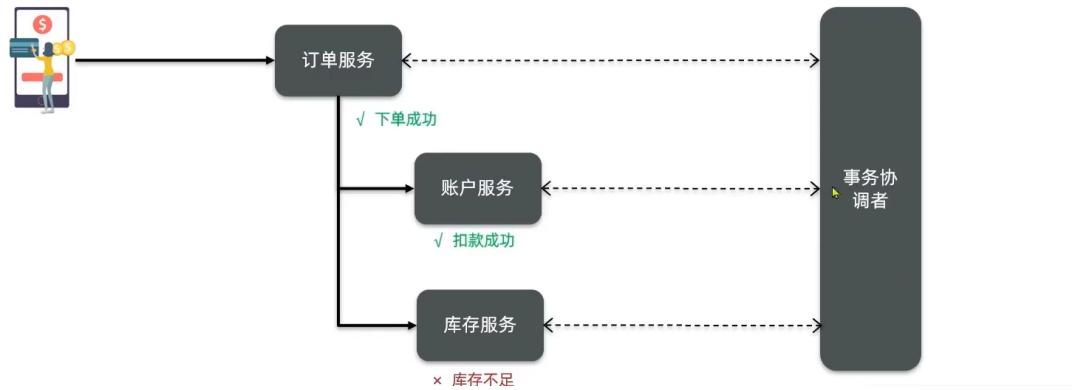
- **Basically Available (基本可用)**：分布式系统在出现故障时，允许损失部分可用性，即保证核心可用。
- **Soft State (软状态)**：在一定时间内，允许出现中间状态，比如临时的不一致状态。
- **Eventually Consistent (最终一致性)**：虽然无法保证强一致性，但是在软状态结束后，最终达到数据一致。

而分布式事务最大的问题是各个子事务的一致性问题，因此可以借鉴CAP定理和BASE理论：

- AP模式：各子事务分别执行和提交，允许出现结果不一致，然后采用弥补措施恢复数据即可，实现最终一致。
- CP模式：各个子事务执行后互相等待，同时提交，同时回滚，达成强一致。但事务等待过程中，处于弱可用状态。

## 分布式事务模型

解决分布式事务，各个子系统之间必须能感知到彼此的事务状态，才能保证状态一致，因此需要一个事务协调者来协调每一个事务的参与者（子系统事务）。



## 简述BASE理论三个思想：

- 基本可用
- 软状态
- 最终一致

## 解决分布式事务的思想和模型：

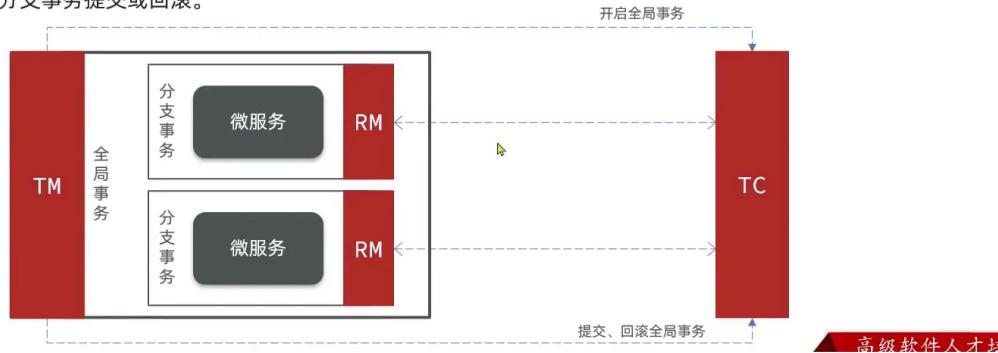
- 全局事务：整个分布式事务
- 分支事务：分布式事务中包含的每个子系统的事务
- 最终一致思想：各分支事务分别执行并提交，如果有不一致的情况，再想办法恢复数据
- 强一致思想：各分支事务执行完业务不要提交，等待彼此结果。而后统一提交或回滚

seata

## Seata架构

Seata事务管理中有三个重要的角色：

- **TC (Transaction Coordinator) - 事务协调者：**维护全局和分支事务的状态，协调全局事务提交或回滚。
- **TM (Transaction Manager) - 事务管理器：**定义全局事务的范围、开始全局事务、提交或回滚全局事务。
- **RM (Resource Manager) - 资源管理器：**管理分支事务处理的资源，与TC交谈以注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚。



## 初识Seata

Seata提供了四种不同的分布式事务解决方案：

- XA模式：强一致性分阶段事务模式，牺牲了一定的可用性，无业务侵入
- TCC模式：最终一致的分阶段事务模式，有业务侵入
- AT模式：最终一致的分阶段事务模式，无业务侵入，也是Seata的默认模式
- SAGA模式：长事务模式，有业务侵入

## 注册

### 使用前的配置

```
registry {
    # 注册中心类型
    # file 、nacos 、eureka、redis、zk、consul、etcd3、
    sofa
    type = "nacos"

    nacos {
        application = "seata-server"
        serverAddr = "192.168.0.101:8848"
        #最好和所要使用的事务微服务一组
        group = "DEFAULT_GROUP"
        namespace = ""
        cluster = "SH" # 集群 和nacos一致就行
    }
}
```

```
    username = "nacos"
    password = "nacos"
}
eureka {
    serviceUrl = "http://localhost:8761/eureka"
    application = "default"
    weight = "1"
}
redis {
    serverAddr = "localhost:6379"
    db = 0
    password = ""
    cluster = "default"
    timeout = 0
}
zk {
    cluster = "default"
    serverAddr = "127.0.0.1:2181"
    sessionTimeout = 6000
    connectTimeout = 2000
    username = ""
    password = ""
}
consul {
    cluster = "default"
    serverAddr = "127.0.0.1:8500"
    aclToken = ""
}
etcd3 {
    cluster = "default"
    serverAddr = "http://localhost:2379"
}
sofa {
    serverAddr = "127.0.0.1:9603"
    application = "default"
    region = "DEFAULT_ZONE"
    datacenter = "DefaultDataCenter"
    cluster = "default"
    group = "SEATA_GROUP"
    addressWaitTime = "3000"
}
```

```
file {
    name = "file.conf"
}

config {
    # 存放文件的位置 最好交给服务作为管理
    # file、nacos、apollo、zk、consul、etcd3
    type = "nacos"

    nacos {
        serverAddr = "192.168.0.101:8848"
        namespace = ""
        group = "SEATA_GROUP"
        username = "nacos"
        password = "nacos"
        # 配置文件名同nacos的云端配置yaml 自己先添加
        dataId = "seataServer.properties"
    }
    consul {
        serverAddr = "127.0.0.1:8500"
        aclToken = ""
    }
    apollo {
        appId = "seata-server"
        ## apolloConfigService will cover apolloMeta
        apolloMeta = "http://192.168.1.204:8801"
        apolloConfigService =
"http://192.168.1.204:8080"
        namespace = "application"
        apolloAccesskeySecret = ""
        cluster = "seata"
    }
    zk {
        serverAddr = "127.0.0.1:2181"
        sessionTimeout = 6000
        connectTimeout = 2000
        username = ""
        password = ""
        nodePath = "/seata/seata.properties"
    }
}
```

```
etcd3 {  
    serverAddr = "http://localhost:2379"  
}  
file {  
    name = "file.conf"  
}  
}
```

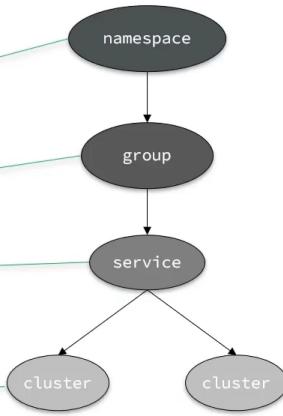
```
<dependency>  
    <groupId>com.alibaba.cloud</groupId>  
    <artifactId>spring-cloud-starter-  
alibaba-seata</artifactId>  
    <exclusions>  
        <!—排除低版本—>  
        <exclusion>  
            <groupId>io.seata</groupId>  
            <artifactId>seata-spring-boot-  
starter</artifactId>  
            </exclusion>  
        </exclusions>  
    </dependency>  
    <!— seata starter—>  
    <dependency>  
        <groupId>io.seata</groupId>  
        <artifactId>seata-spring-boot-  
starter</artifactId>  
        <version>${seata.version}</version>  
    </dependency>
```

## 微服务集成Seata



2. 然后，配置application.yml，让微服务通过注册中心找到seata-tc-server：

```
seata:  
  registry: # TC服务注册中心的配置，微服务根据这些信息去注册中心获取tc服务地址  
    # 参考tc服务自己的registry.conf中的配置，  
    # 包括：地址、namespace、group、application-name，cluster  
    type: nacos  
    nacos: # tc  
      server-addr: 127.0.0.1:8848  
      namespace: ""  
      group: DEFAULT_GROUP  
      application: seata-tc-server # tc服务在nacos中的服务名称  
      tx-service-group: seata-demo # 事务组，根据这个获取tc服务的cluster名称  
    service:  
      vgroup-mapping: # 事务组与TC服务cluster的映射关系  
        seata-demo: SH
```



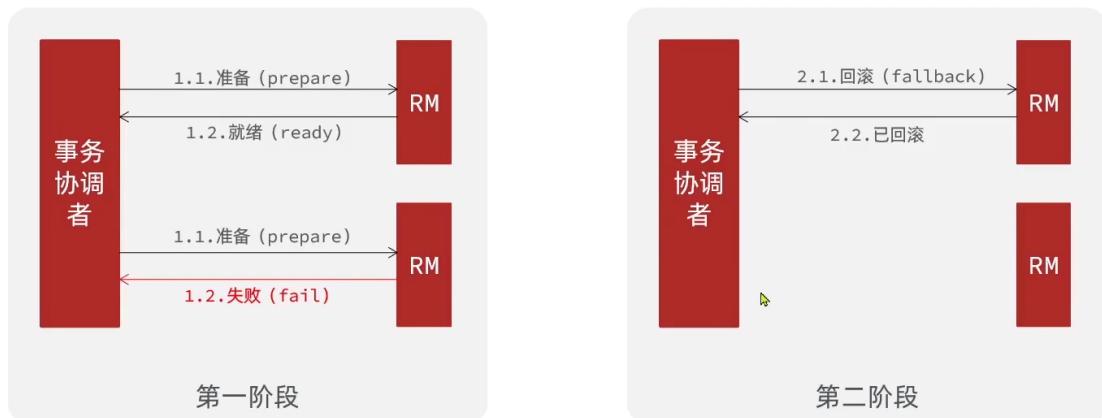
## 配置文件

```
seata:  
  registry:  
    nacos:  
      application: seata-server # 注意和seata配置文件  
一致  
      group: DEFAULT_GROUP  
      namespace: ""  
      server-addr: 127.0.0.1:8845  
      username: nacos  
      password: nacos  
    type: nacos  
    tx-service-group: seata-demo  
  service:  
    vgroup-mapping:  
      seata-demo: SH
```

## XA模式

## XA模式原理

XA 规范是 X/Open 组织定义的分布式事务处理（DTP，Distributed Transaction Processing）标准，XA 规范描述了全局 TM 与局部的 RM 之间的接口，几乎所有主流的数据库都对 XA 规范提供了支持。



## seata的XA模式

seata的XA模式做了一些调整，但大体相似：

RM一阶段的工作：

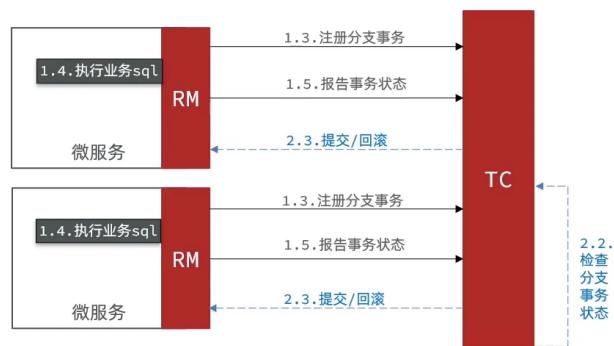
- ① 注册分支事务到TC
- ② 执行分支业务sql但不提交
- ③ 报告执行状态到TC

TC二阶段的工作：

- TC检测各分支事务执行状态
  - a. 如果都成功，通知所有RM提交事务
  - b. 如果有失败，通知所有RM回滚事务

RM二阶段的工作：

- 接收TC指令，提交或回滚事务



## XA模式的优点是什么？

- 事务的强一致性，满足ACID原则。
- 常用数据库都支持，实现简单，并且没有代码侵入

## XA模式的缺点是什么？

- 因为一阶段需要锁定数据库资源，等待二阶段结束才释放，性能较差
- 依赖关系型数据库实现事务

## 实现XA模式

Seata的starter已经完成了XA模式的自动装配，实现非常简单，步骤如下：

1. 修改application.yml文件（每个参与事务的微服务），开启XA模式：

```
seata:  
  data-source-proxy-mode: XA # 开启数据源代理的XA模式
```

2. 给发起全局事务的入口方法添加@GlobalTransactional注解，本例中是OrderServiceImpl中的create方法：

```
@Override  
 @GlobalTransactional  
 public Long create(Order order) {  
     // 创建订单  
     orderMapper.insert(order);  
     // 扣余额 ... 略  
     // 扣减库存 ... 略  
     return order.getId();  
 }
```

3. 重启服务并测试

## AT模式

### AT模式原理

AT模式同样是分阶段提交的事务模型，不过弥补了XA模型中资源锁定周期过长的缺陷。

阶段一RM的工作：

- 注册分支事务
- 记录undo-log（数据快照）
- 执行业务sql并提交
- 报告事务状态

阶段二提交时RM的工作：

- 删除undo-log即可

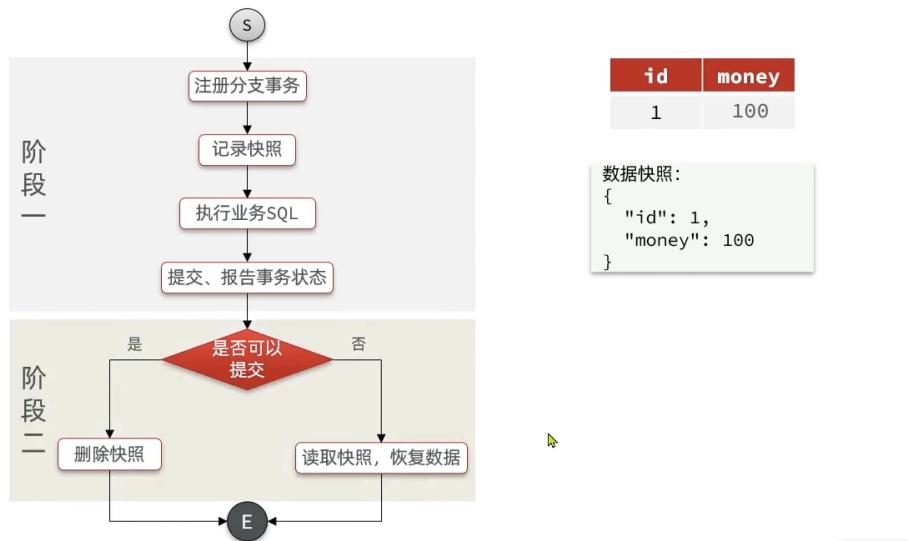
阶段二回滚时RM的工作：

- 根据undo-log恢复数据到更新前



## AT模式原理

例如，一个分支业务的SQL是这样的：update tb\_account set money = money - 10 where id = 1

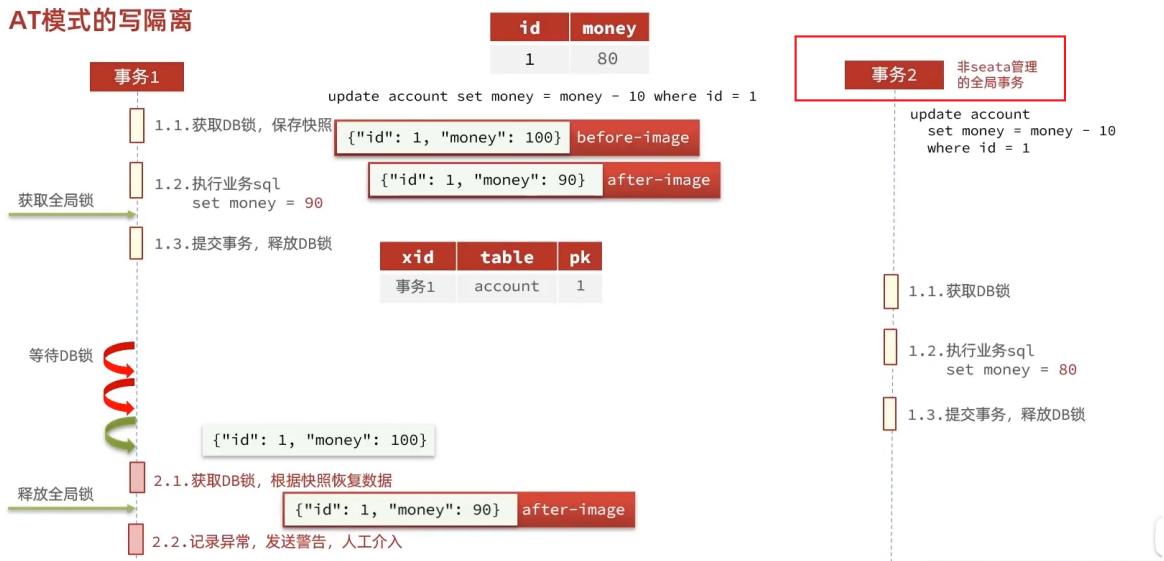
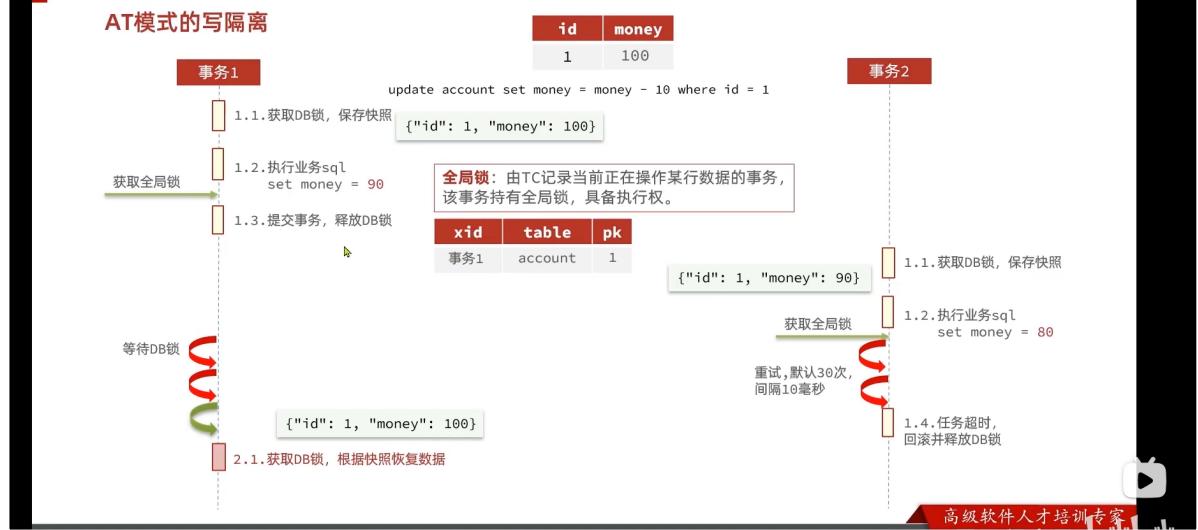


## 简述AT模式与XA模式最大的区别是什么？

- XA模式一阶段不提交事务，锁定资源；AT模式一阶段直接提交，不锁定资源。
- XA模式依赖数据库机制实现回滚；AT模式利用数据快照实现数据回滚。
- XA模式强一致；AT模式最终一致

## AT模式的脏写问题





## AT模式的优点：

- 一阶段完成直接提交事务，释放数据库资源，性能比较好
- 利用全局锁实现读写隔离
- 没有代码侵入，框架自动完成回滚和提交

## AT模式的缺点：

- 两阶段之间属于软状态，属于最终一致
- 框架的快照功能会影响性能，但比XA模式要好很多

## 实现AT模式

AT模式中的快照生成、回滚等动作都是由框架自动完成，没有任何代码侵入，因此实现非常简单。

- 导入课前资料提供的Sql文件：seata-at.sql，其中lock\_table导入到TC服务关联的数据库，undo\_log表导入到微服务关联的数据库：



seata-at.sql

- 修改application.yml文件，将事务模式修改为AT模式即可：

```
seata:  
  data-source-proxy-mode: AT # 开启数据源代理的AT模式
```

- 重启服务并测试

```
/*  
Navicat Premium Data Transfer  
  
Source Server : local  
Source Server Type : MySQL  
Source Server Version : 50622  
Source Host : localhost:3306  
Source Schema : seata_demo  
  
Target Server Type : MySQL  
Target Server Version : 50622  
File Encoding : 65001  
  
Date: 20/06/2021 12:39:03  
*/  
  
SET NAMES utf8mb4;  
SET FOREIGN_KEY_CHECKS = 0;  
  
--  
-- Table structure for undo_log  
--  
DROP TABLE IF EXISTS `undo_log`;  
CREATE TABLE `undo_log` (  
  `branch_id` bigint(20) NOT NULL COMMENT 'branch transaction id',  
  `xid` varchar(100) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT 'global transaction id',  
  `log_id` bigint(20) NOT NULL COMMENT 'log sequence number',  
  `transaction_id` varchar(100) NOT NULL COMMENT 'transaction id',  
  `rollback_info` longblob NOT NULL COMMENT 'rollback info',  
  `log_status` int(11) NOT NULL COMMENT 'log status',  
  `log_created` datetime NOT NULL COMMENT 'log created time',  
  `log_modified` datetime NOT NULL COMMENT 'log modified time',  
  `ext_info` longblob NOT NULL COMMENT 'ext info'
```

```
    `context` varchar(128) CHARACTER SET utf8 COLLATE
utf8_general_ci NOT NULL COMMENT 'undo_log
context,such as serialization',
    `rollback_info` longblob NOT NULL COMMENT
'rollback info',
    `log_status` int(11) NOT NULL COMMENT '0:normal
status,1:defense status',
    `log_created` datetime(6) NOT NULL COMMENT
'create datetime',
    `log_modified` datetime(6) NOT NULL COMMENT
'modify datetime',
    UNIQUE INDEX `ux_undo_log`(`xid`, `branch_id`)
USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE =
utf8_general_ci COMMENT = 'AT transaction mode undo
table' ROW_FORMAT = Compact;
```

```
-- -----
-- Records of undo_log
-----
```

```
-- -----
-- Table structure for lock_table
-----
DROP TABLE IF EXISTS `lock_table`;
CREATE TABLE `lock_table` (
    `row_key` varchar(128) CHARACTER SET utf8 COLLATE
utf8_general_ci NOT NULL,
    `xid` varchar(96) CHARACTER SET utf8 COLLATE
utf8_general_ci NULL DEFAULT NULL,
    `transaction_id` bigint(20) NULL DEFAULT NULL,
    `branch_id` bigint(20) NOT NULL,
    `resource_id` varchar(256) CHARACTER SET utf8
COLLATE utf8_general_ci NULL DEFAULT NULL,
    `table_name` varchar(32) CHARACTER SET utf8
COLLATE utf8_general_ci NULL DEFAULT NULL,
    `pk` varchar(36) CHARACTER SET utf8 COLLATE
utf8_general_ci NULL DEFAULT NULL,
    `gmt_create` datetime NULL DEFAULT NULL,
```

```
`gmt_modified` datetime NULL DEFAULT NULL,  
PRIMARY KEY (`row_key`) USING BTREE,  
INDEX `idx_branch_id`(`branch_id`) USING BTREE  
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE =  
utf8_general_ci ROW_FORMAT = Compact;
```

```
SET FOREIGN_KEY_CHECKS = 1;
```

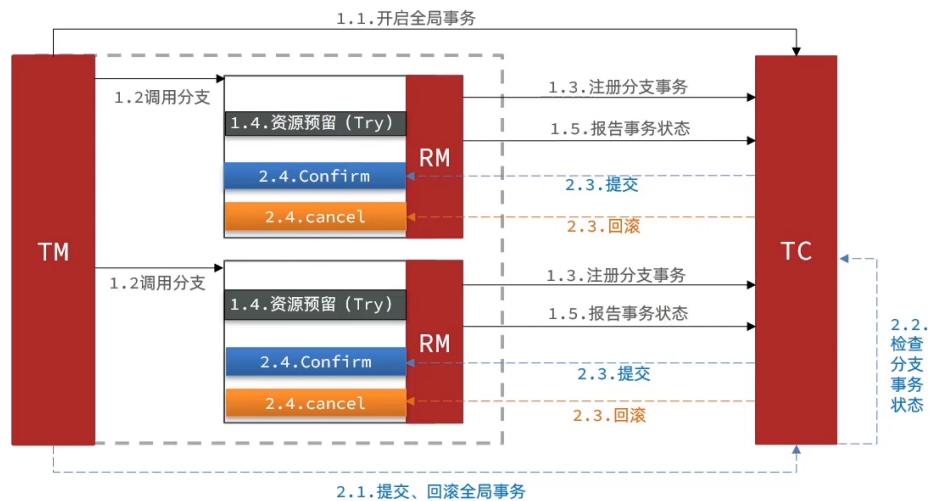
## TCC

### TCC模式原理

TCC模式与AT模式非常相似，每阶段都是独立事务，不同的是TCC通过人工编码来实现数据恢复。需要实现三个方法：

- Try: 资源的检测和预留；
- Confirm: 完成资源操作业务；要求 Try 成功 Confirm 一定要能成功。
- Cancel: 预留资源释放，可以理解为try的反向操作。

TCC的工作模型图：



## TCC模式的每个阶段是做什么的？

- Try: 资源检查和预留
- Confirm: 业务执行和提交
- Cancel: 预留资源的释放

## TCC的优点是什么？

- 一阶段完成直接提交事务，释放数据库资源，性能好
- 相比AT模型，无需生成快照，无需使用全局锁，性能最强
- 不依赖数据库事务，而是依赖补偿操作，可以用于非事务型数据库

## TCC的缺点是什么？

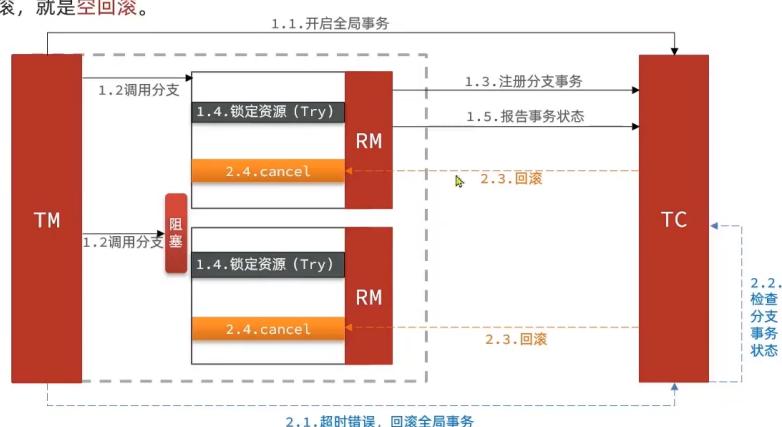
- 有代码侵入，需要人为编写try、Confirm和Cancel接口，太麻烦
- 软状态，事务是最终一致
- 需要考虑Confirm和Cancel的失败情况，做好幂等处理



## TCC的空回滚和业务悬挂

当某分支事务的try阶段阻塞时，可能导致全局事务超时而触发二阶段的cancel操作。在未执行try操作时先执行了cancel操作，这时cancel不能做回滚，就是**空回滚**。

对于已经空回滚的业务，如果以后继续执行try，就永远不可能confirm或cancel，这就是**业务悬挂**。应当阻止执行空回滚后的try操作，避免悬挂



## 业务分析

为了实现空回滚、防止业务悬挂，以及幂等性要求。我们必须在数据库记录冻结金额的同时，记录当前事务id和执行状态，为此我们设计了一张表：

### Try业务

- 记录冻结金额和事务状态到 account\_freeze 表
- 扣减 account 表可用金额

```
CREATE TABLE `account_freeze_tbl` (
  `xid` varchar(128) NOT NULL,
  `user_id` varchar(255) DEFAULT NULL COMMENT '用户id',
  `freeze_money` int(11) unsigned DEFAULT '0' COMMENT '冻结金额',
  `state` int(1) DEFAULT NULL COMMENT '事务状态, 0:try, 1:confirm, 2:cancel',
  PRIMARY KEY (`xid`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8 ROW_FORMAT=COMPACT;
```

### Confirm业务

- 根据xid删除 account\_freeze 表的冻结记录

### Cancel业务

- 修改 account\_freeze 表，冻结金额为0, state为2
- 修改 account 表，恢复可用金额

### 如何判断是否空回滚

- cancel 业务中，根据 xid 查询 account\_freeze，如果为 null 则说明 try 还没做，需要空回滚

### 如何避免业务悬挂

- try 业务中，根据 xid 查询 account\_freeze，如果已经存在则证明 cancel 已经执行，拒绝执行 try 业务

## 声明TCC接口

TCC的Try、Confirm、Cancel方法都需要在接口中基于注解来声明，语法如下：

```
@LocalTCC
public interface TCCService {
    /**
     * Try 逻辑，@TwoPhaseBusinessAction 中的 name 属性要与当前方法名一致，用于指定 Try 逻辑对应的方法
     */
    @TwoPhaseBusinessAction(name = "prepare", commitMethod = "confirm", rollbackMethod = "cancel")
    void prepare(@BusinessActionContextParameter(paramName = "param") String param);
    /**
     * 二阶段 confirm 确认方法、可以另命名，但要保证与 commitMethod 一致
     *
     * @param context 上下文，可以传递 try 方法的参数
     * @return boolean 执行是否成功
     */
    boolean confirm (BusinessActionContext context);
    /**
     * 二阶段回滚方法，要保证与 rollbackMethod 一致
     */
    boolean cancel (BusinessActionContext context);
}
```

### @LocalTCC

```
public interface AccountTccService {

    @TwoPhaseBusinessAction(name="prepare",
                           commitMethod ="confirm",
                           rollbackMethod = "cancel")
    // 设置字符串可以方便 BusinessActionContext 获取
    void
    prepare(@BusinessActionContextParameter(paramName =
    "userId") String userId,

    @BusinessActionContextParameter(paramName =
    "money") int money);

    Boolean confirm(BusinessActionContext context);
```

```
        Boolean cancel(BusinessActionContext context);  
    }
```

## 实现类

```
@Service  
public class AccountTccServiceImpl implements  
AccountTccService {  
  
    @Autowired  
    private AccountMapper accountMapper;  
  
    @Autowired  
    private AccountFreezeMapper freezeMapper;  
  
    @Override  
    // 加上事务注解防止扣减余额失败导致扣错  
    @Transactional  
    public void prepare(String userId, int money) {  
        // 获取事务id  
        String xid = RootContext.getXID();  
        // 悬挂的问题  
        if (freezeMapper.selectById(xid) != null) {  
            // 代表有冻结的记录 执行过了 直接结束任务  
            return;  
        }  
        // 1. 扣减可用余额  
        accountMapper.deduct(userId, money);  
        // 2. 记录冻结金额  
        AccountFreeze freeze = new AccountFreeze();  
        freeze.setXid(xid);  
        freeze.setUserId(userId);  
        freeze.setFreezeMoney(money);  
        freeze.setState(AccountFreeze.State.TRY);  
  
        freezeMapper.insert(freeze);  
    }  
  
    @Override
```

```
    public Boolean confirm(BusinessActionContext context) {
        // 1,. 获取事务id 根据id删除冻结记录
        String xid = context.getXid();
        int count = freezeMapper.deleteById(xid);
        // 证明是否删除成功
        return count == 1;
    }

    /**
     * 恢复冻结的钱 这些可以直接从数据库获取
     * 也可以从上下文对象获取
     * @param context
     * @return
     */
    @Override
    public Boolean cancel(BusinessActionContext context) {
        AccountFreeze freeze =
freezeMapper.selectById(context.getXid());

        // 空回滚的判断
        if (freeze == null) {
            // 获取id通过注解修饰的字符串
            String userId =
context.getActionContext("userId").toString();
            // 证明try没有执行 此时插入一条数据证明回滚过
            freeze = new AccountFreeze();
            freeze.setXid(context.getXid());
            freeze.setUserId(userId);
            freeze.setFreezeMoney(0);

            freeze.setState(AccountFreeze.State.CANCEL);
            freezeMapper.insert(freeze);
            return true;
        }
        // 判断幂等性 是否执行过
        if
(freeze.getState() == AccountFreeze.State.CANCEL){
            return true;
        }
    }
}
```

```

    // 恢复钱

    accountMapper.refund(freeze.getUserId(),freeze.get
FreezeMoney());
    // 将状态改为cancel 冻结金额清零

    freeze.setState(AccountFreeze.State.CANCEL);
    freeze.setFreezeMoney(0);

    int count =
freezeMapper.updateById(freeze);
    return count == 1;
}
}

```

注意 之后将web层的service置换为该接口  
由业务的不同所以逻辑是不一样的 只是一个demo

## saga

### Saga模式

Saga模式是SEATA提供的长事务解决方案。也分为两个阶段：

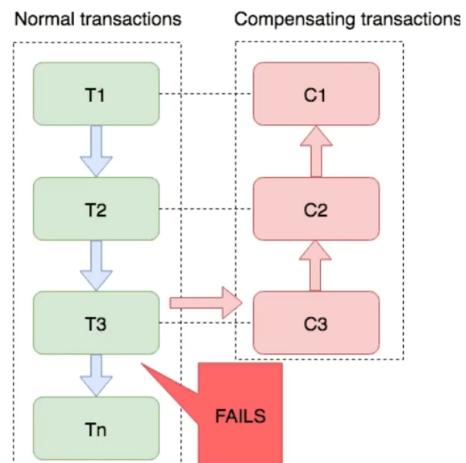
- 一阶段：直接提交本地事务
- 二阶段：成功则什么都不做；失败则通过编写补偿业务来回滚

Saga模式优点：

- 事务参与者可以基于事件驱动实现异步调用，吞吐高
- 一阶段直接提交事务，无锁，性能好
- 不用编写TCC中的三个阶段，实现简单

缺点：

- 软状态持续时间不确定，时效性差
- 没有锁，没有事务隔离，会有脏写



# 对比

## 四种模式对比

	XA	AT	TCC	SAGA
一致性	强一致	弱一致	弱一致	最终一致
隔离性	完全隔离	基于全局锁隔离	基于资源预留隔离	无隔离
代码侵入	无	无	有，要编写三个接口	有，要编写状态机和补偿业务
性能	差	好	非常好	非常好
场景	对一致性、隔离性有高要求的业务	基于关系型数据库的大多数分布式事务场景都可以	<ul style="list-style-type: none"><li>对性能要求较高的事务。</li><li>有非关系型数据库要参与的事务。</li></ul>	<ul style="list-style-type: none"><li>业务流程长、业务流程多</li><li>参与者包含其它公司或遗留系统服务，无法提供 TCC 模式要求的三个接口</li></ul>

# 分布式缓存

## RDB和AOF

### RDB

RDB全称Redis Database Backup file（Redis数据备份文件），也被叫做Redis数据快照。简单来说就是把内存中的所有数据都记录到磁盘中。当Redis实例故障重启后，从磁盘读取快照文件，恢复数据。

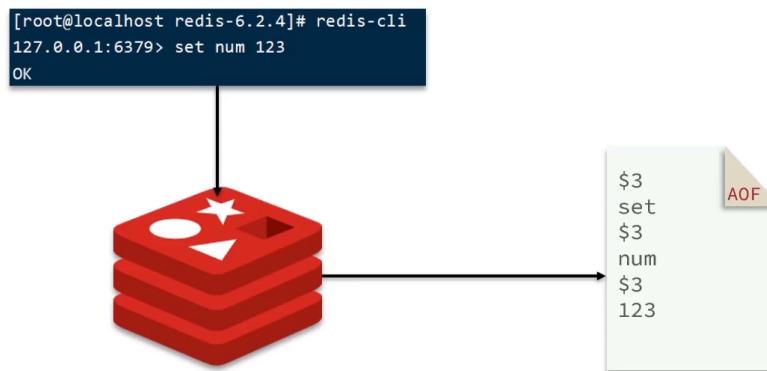
快照文件称为RDB文件，默认是保存在当前运行目录。

```
[root@localhost ~]# redis-cli
127.0.0.1:6379> save  #由Redis主进程来执行RDB，会阻塞所有命令
ok
127.0.0.1:6379> bgsave  #开启子进程执行RDB，避免主进程受到影响
Background saving started
```

Redis停机时会执行一次RDB。

## AOF

AOF全称为Append Only File（追加文件）。Redis处理的每一个写命令都会记录在AOF文件，可以看做是命令日志文件。



## AOF

AOF默认是关闭的，需要修改redis.conf配置文件来开启AOF：

```
# 是否开启AOF功能，默认是no  
appendonly yes  
# AOF文件的名称  
appendfilename "appendonly.aof"
```

AOF的命令记录的频率也可以通过redis.conf文件来配：

```
# 表示每执行一次写命令，立即记录到AOF文件  
appendfsync always  
# 写命令执行完先放入AOF缓冲区，然后表示每隔1秒将缓冲区数据写到AOF文件，是默认方案  
appendfsync everysec  
# 写命令执行完先放入AOF缓冲区，由操作系统决定何时将缓冲区内容写回磁盘  
appendfsync no
```

配置项	刷盘时机	优点	缺点
Always	同步刷盘	可靠性高，几乎不丢数据	性能影响大
everysec	每秒刷盘	性能适中	最多丢失1秒数据
no	操作系统控制	性能最好	可靠性较差，可能丢失大量数据

## AOF

因为是记录命令，AOF文件会比RDB文件大的多。而且AOF会记录对同一个key的多次写操作，但只有最后一次写操作才有意义。通过执行bgrewriteaof命令，可以让AOF文件执行重写功能，用最少的命令达到相同效果。



Redis也会在触发阈值时自动去重写AOF文件。阈值也可以在redis.conf中配置：

```
# AOF文件比上次文件 增长超过多少百分比则触发重写  
auto-aof-rewrite-percentage 100  
# AOF文件体积最小多大以上才触发重写  
auto-aof-rewrite-min-size 64mb
```

## AOF

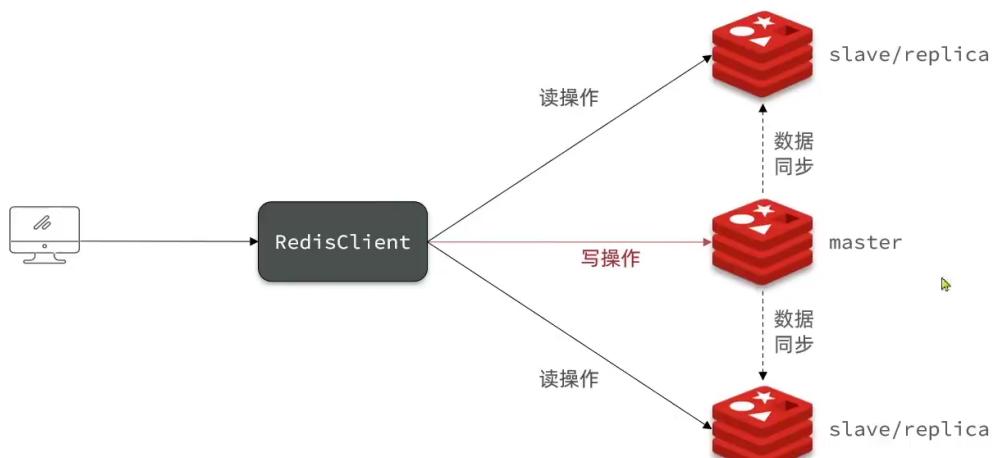
RDB和AOF各有自己的优缺点，如果对数据安全性要求较高，在实际开发中往往会结合两者来使用。

	RDB	AOF
持久化方式	定时对整个内存做快照	记录每一次执行的命令
数据完整性	不完整，两次备份之间会丢失	相对完整，取决于刷盘策略
文件大小	会有压缩，文件体积小	记录命令，文件体积很大
宕机恢复速度	很快	慢
数据恢复优先级	低，因为数据完整性不如AOF	高，因为数据完整性更高
系统资源占用	高，大量CPU和内存消耗	低，主要是磁盘IO资源 但AOF重写时会占用大量CPU和内存资源
使用场景	可以容忍数分钟的数据丢失，追求更快的启动速度	对数据安全性要求较高常见

## 主从架构

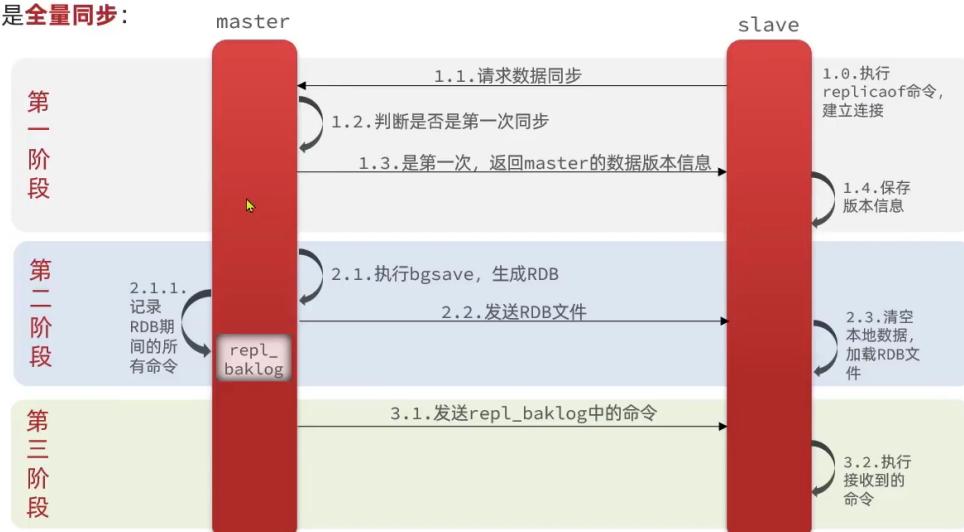
### 搭建主从架构

单节点Redis的并发能力是有上限的，要进一步提高Redis的并发能力，就需要搭建主从集群，实现读写分离。



### 数据同步原理

主从第一次同步是**全量同步**：



## 简述全量同步的流程？

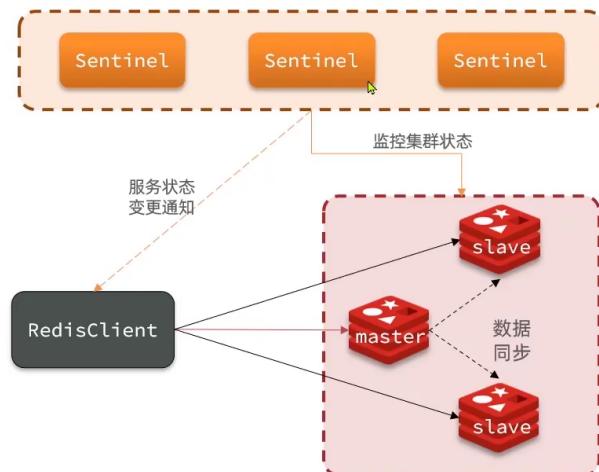
- slave节点请求增量同步
- master节点判断replid，发现不一致，拒绝增量同步
- master将完整内存数据生成RDB，发送RDB到slave
- slave清空本地数据，加载master的RDB
- master将RDB期间的命令记录在repl\_baklog，并持续将log中的命令发送给slave

## 哨兵

### 哨兵的作用

Redis提供了哨兵（Sentinel）机制来实现主从集群的自动故障恢复。哨兵的结构和作用如下：

- **监控**: Sentinel 会不断检查您的master和slave 是否按预期工作
- **自动故障恢复**: 如果master故障，Sentinel会将一个slave提升为master。当故障实例恢复后也以新的master为主
- **通知**: Sentinel充当Redis客户端的服务发现来源，当集群发生故障转移时，会将最新信息推送 给Redis的客户端



### 选举新的master

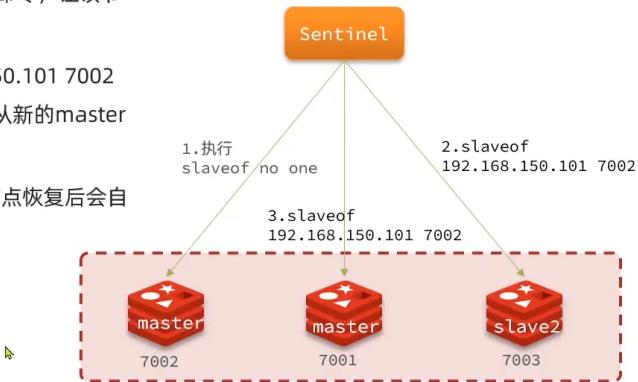
一旦发现master故障，sentinel需要在salve中选择一个作为新的master，选择依据是这样的：

- 首先会判断slave节点与master节点断开时间长短，如果超过指定值（down-after-milliseconds \* 10）则会排除该 slave节点
- 然后判断slave节点的slave-priority值，越小优先级越高，如果是0则永不参与选举
- 如果slave-priority一样，则判断slave节点的offset值，越大说明数据越新，优先级越高
- 最后是判断slave节点的运行id大小，越小优先级越高。

## 如何实现故障转移

当选中了其中一个slave为新的master后（例如slave1），故障的转移的步骤如下：

- sentinel给备选的slave1节点发送slaveof no one命令，让该节点成为master
- sentinel给所有其它slave发送slaveof 192.168.150.101 7002命令，让这些slave成为新master的从节点，开始从新的master上同步数据。
- 最后，sentinel将故障节点标记为slave，当故障节点恢复后会自动成为新的master的slave节点



## RedisTemplate的哨兵模式

- 在pom文件中引入redis的starter依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

- 然后在配置文件application.yml中指定sentinel相关信息：

```
spring:
  redis:
    sentinel:
      master: mymaster # 指定master名称
      nodes: # 指定redis-sentinel集群信息
        - 192.168.150.101:27001
        - 192.168.150.101:27002
        - 192.168.150.101:27003
```

- 配置主从读写分离

```
@Bean
public LettuceClientConfigurationBuilderCustomizer configurationBuilderCustomizer(){
    return configBuilder -> configBuilder.readFrom(ReadFrom.REPLICA_PREFERRED);
}
```

这里的ReadFrom是配置Redis的读取策略，是一个枚举，包括下面选择：

- MASTER：从主节点读取
- MASTER\_PREFERRED：优先从master节点读取，master不可用才读取replica
- REPLICA：从slave (replica) 节点读取
- REPLICA\_PREFERRED：优先从slave (replica) 节点读取，所有的slave都不可用才读取master

## Redis分片集群

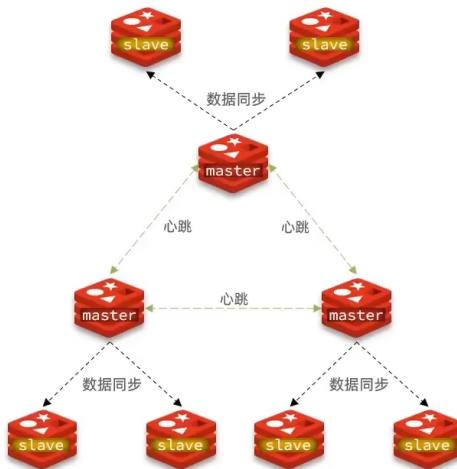
## 分片集群结构

主从和哨兵可以解决高可用、高并发读的问题。但是依然有两个问题没有解决：

- 海量数据存储问题
- 高并发写的问题

使用分片集群可以解决上述问题，分片集群特征：

- 集群中有多个master，每个master保存不同数据
- 每个master都可以有多个slave节点
- master之间通过ping监测彼此健康状态
- 客户端请求可以访问集群任意节点，最终都会被转发到正确节点



## 散列插槽

Redis会把每一个master节点映射到0~16383共16384个插槽 (hash slot) 上，查看集群信息时就能看到：

```
M: f5fc58defbebb957e47fb0d8327a09dc4f1678f5 192.168.150.101:7001
  slots:[0-5460] (5461 slots) master
M: afaaa70d6528fc72490e0f3f7b32731a12c12bb8 192.168.150.101:7002
  slots:[5461-10922] (5462 slots) master
M: 1c00e5f9e158b169f199f15884ab43bc433b1a06 192.168.150.101:7003
  slots:[10923-16383] (5461 slots) master
```

数据key不是与节点绑定，而是与插槽绑定。redis会根据key的有效部分计算插槽值，分两种情况：

- key中包含 "{}"，且 "{}" 中至少包含1个字符，“{}”中的部分是有效部分
- key中不包含 "{}"，整个key都是有效部分

例如：key是num，那么就根据num计算，如果是{itcast}num，则根据itcast计算。计算方式是利用CRC16算法得到一个hash值，然后对16384取余，得到的结果就是slot值。

## Redis如何判断某个key应该在哪个实例？

- 将16384个插槽分配到不同的实例
- 根据key的有效部分计算哈希值，对16384取余
- 余数作为插槽，寻找插槽所在实例即可

## 如何将同一类数据固定的保存在同一个Redis实例？

- 这一类数据使用相同的有效部分，例如key都以{typeld}为前缀

## 添加一个节点到集群

redis-cli --cluster提供了很多操作集群的命令，可以通过下面方式查看：

```
[root@localhost ~]# redis-cli --cluster help
Cluster Manager Commands:
  create      host1:port1 ... hostN:portN
               --cluster-replicas <arg>
  check       host:port
               --cluster-search-multiple-owners
  info        host:port
  fix         host:port
               --cluster-search-multiple-owners
               --cluster-fix-with-unreachable-masters
```

比如，添加节点的命令：

```
[root@localhost ~]# redis-cli --cluster help
Cluster Manager Commands:
  create      host1:port1 ... hostN:portN
               --cluster-replicas <arg>
  add-node    new_host:new_port existing_host:existing_port
               --cluster-slave
               --cluster-master-id <arg>
```

## 故障转移

当集群中有一个master宕机会发生什么呢？

- 首先是该实例与其它实例失去连接
- 然后是疑似宕机：

```
1fa6d68d590827c24c237b1c490b78e5c7fe2ca9 192.168.150.101:8003@18003 slave f5fc58defbebb957e47fb0d8327a09dc4f1678f5 0 1625207711535 8 connected
f5fc58defbebb957e47fb0d8327a09dc4f1678f5 192.168.150.101:7001@17001 myself,master - 0 1625207710000 8 connected 0-5460
afaaa70d6528fc72490e0f3f7b32731a12c12bb8 192.168.150.101:7002@17002 master,fail? - 1625207705198 1625207703000 10 disconnected 5461-10922
6ec60fb5af950a465f05c8024bf8f75d809b014 192.168.150.101:8002@18002 slave lc00e5f9e158b169f199f15884ab43bc433b1a06 0 1625207710000 3 connected
1c00e5f9e158b169f199f15884ab43bc433b1a06 192.168.150.101:7003@17003 master - 0 1625207711000 3 connected 10923-16383
7b6d5ffc9a985d614dc5aeb2ee3abac1adfd3e22 192.168.150.101:8001@18001 slave afaaa70d6528fc72490e0f3f7b32731a12c12bb8 0 1625207709420 10 connected
```

- 最后是确定下线，自动提升一个slave为新的master：

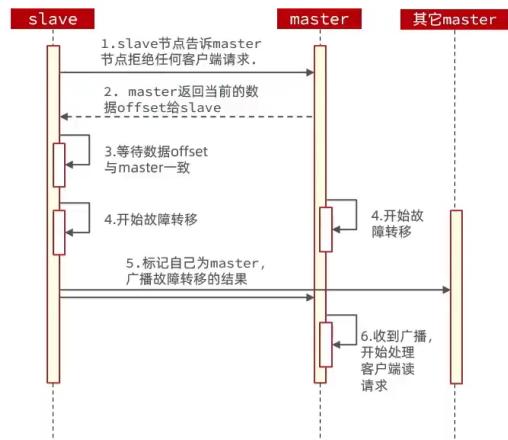
```
1fa6d68d590827c24c237b1c490b78e5c7fe2ca9 192.168.150.101:8003@18003 slave f5fc58defbebb957e47fb0d8327a09dc4f1678f5 0 1625208023157 8
f5fc58defbebb957e47fb0d8327a09dc4f1678f5 192.168.150.101:7001@17001 myself,master - 0 1625208022000 8 connected 0-5460
afaaa70d6528fc72490e0f3f7b32731a12c12bb8 192.168.150.101:7002@17002 master,fail? - 1625207705198 1625207703000 10 disconnected
6ec60fb5af950a465f05c8024bf8f75d809b014 192.168.150.101:8002@18002 slave lc00e5f9e158b169f199f15884ab43bc433b1a06 0 1625208021035 3
1c00e5f9e158b169f199f15884ab43bc433b1a06 192.168.150.101:7003@17003 master - 0 1625208022084 3 connected 10923-16383
7b6d5ffc9a985d614dc5aeb2ee3abac1adfd3e22 192.168.150.101:8001@18001 master - 0 1625208023000 11 connected 5461-10922
```

## 数据迁移

利用cluster failover命令可以手动让集群中的某个master宕机，切换到执行cluster failover命令的这个slave节点，实现无感知的数据迁移。其流程如下：

手动的Failover支持三种不同模式：

- 缺省：默认的流程，如图1~6步
- force：省略了对offset的一致性校验
- takeover：直接执行第5步，忽略数据一致性、忽略master状态和其它master的意见



## RedisTemplate访问分片集群

RedisTemplate底层同样基于lettuce实现了分片集群的支持，而使用的步骤与哨兵模式基本一致：

1. 引入redis的starter依赖
2. 配置分片集群地址
3. 配置读写分离

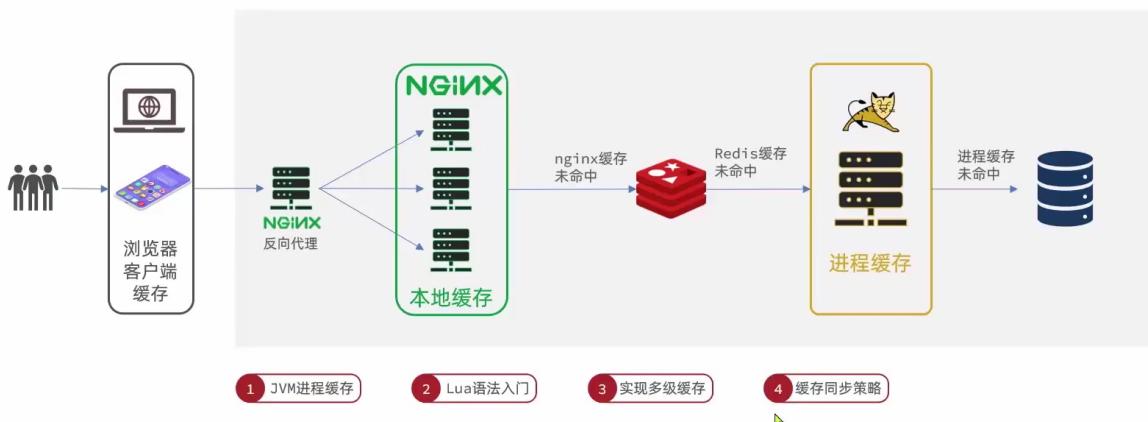
与哨兵模式相比，其中只有分片集群的配置方式略有差异，如下：

```
spring:  
  redis:  
    cluster:  
      nodes: # 指定分片集群的每一个节点信息  
        - 192.168.150.101:7001  
        - 192.168.150.101:7002  
        - 192.168.150.101:7003  
        - 192.168.150.101:8001  
        - 192.168.150.101:8002  
        - 192.168.150.101:8003
```

# 多级缓存

## 多级缓存方案

用作缓存的Nginx是业务Nginx，需要部署为集群，再有专门的Nginx用来做反向代理：



# 进程缓存

## 本地进程缓存

缓存在日常开发中启动至关重要的作用，由于是存储在内存中，数据的读取速度是非常快的，能大量减少对数据库的访问，减少数据库的压力。我们把缓存分为两类：

- 分布式缓存，例如Redis：
  - 优点：存储容量更大、可靠性更好、可以在集群间共享
  - 缺点：访问缓存有网络开销
  - 场景：缓存数据量较大、可靠性要求较高、需要在集群间共享
- 进程本地缓存，例如HashMap、GuavaCache：
  - 优点：读取本地内存，没有网络开销，速度更快
  - 缺点：存储容量有限、可靠性较低、无法共享
  - 场景：性能要求较高，缓存数据量较小

## 本地进程缓存

Caffeine是一个基于Java8开发的，提供了近乎最佳命中率的高性能的本地缓存库。目前Spring内部的缓存使用的就是Caffeine。GitHub地址：<https://github.com/ben-manes/caffeine>



Caffeine提供了三种缓存驱逐策略：

- 基于容量：设置缓存的数量上限

```
// 创建缓存对象
Cache<String, String> cache = Caffeine.newBuilder()
    .maximumSize(1) // 设置缓存大小上限为 1
    .build();
```

- 基于时间：设置缓存的有效时间

```
// 创建缓存对象
Cache<String, String> cache = Caffeine.newBuilder()
    .expireAfterWrite(Duration.ofSeconds(10)) // 设置缓存有效期为 10 秒，从最后一次写入开始计时
    .build();
```

- 基于引用：设置缓存为软引用或弱引用，利用GC来回收缓存数据。性能较差，不建议使用。

在默认情况下，当一个缓存元素过期的时候，Caffeine不会自动立即将其清理和驱逐。而是在一次读或写操作后，或者在空闲时间完成对失效数据的驱逐。

### @Configuration

```
public class CaffeineConfig {
```

#### @Bean

```
public Cache<Long, Item> itemCache(){
    return Caffeine.newBuilder()
        .initialCapacity(100) // 设置初始化值
```

```

        .maximumSize(10_000) // 设置最大上
        .build();
    }

    @Bean
    public Cache<Long, ItemStock> itemStockCache(){
        return Caffeine.newBuilder()
            .initialCapacity(100)
            .maximumSize(10_000)
            .build();
    }

}

```

## Lua

### 数据类型

数据类型	描述
nil	这个最简单，只有值nil属于该类，表示一个无效值（在条件表达式中相当于false）。
boolean	包含两个值：false和true
number	表示双精度类型的实浮点数
string	字符串由一对双引号或单引号来表示
function	由 C 或 Lua 编写的函数
table	Lua 中的表 (table) 其实是一个"关联数组" (associative arrays)，数组的索引可以是数字、字符串或表类型。在 Lua 里，table 的创建是通过"构造表达式"来完成，最简单构造表达式是{}，用来创建一个空表。

## 变量

Lua声明变量的时候，并不需要指定数据类型：

```
-- 声明字符串  
local str = 'hello'  
-- 声明数字  
local num = 21  
-- 声明布尔类型  
local flag = true  
-- 声明数组 key为索引的 table  
local arr = {'java', 'python', 'lua'}  
-- 声明table, 类似java的map  
local map = {name='Jack', age=21}
```

访问table：

```
-- 访问数组, lua数组的角标从1开始  
print(arr[1])  
-- 访问table  
print(map['name'])  
print(map.name)
```

## 循环

数组、table都可以利用for循环来遍历：

- 遍历数组：

```
-- 声明数组 key为索引的 table  
local arr = {'java', 'python', 'lua'}  
-- 遍历数组  
for index,value in ipairs(arr) do  
    print(index, value)  
end
```

- 遍历table：

```
-- 声明map, 也就是table  
local map = {name='Jack', age=21}  
-- 遍历table  
for key,value in pairs(map) do  
    print(key, value)  
end
```

## 条件控制

类似Java的条件控制，例如if、else语法：

```
if(布尔表达式)
then
  --[ 布尔表达式为 true 时执行该语句块 --]
else
  --[ 布尔表达式为 false 时执行该语句块 --]
end
```

与java不同，布尔表达式中的逻辑运算是基于英文单词：

操作符	描述	实例
and	逻辑与操作符。若 A 为 false，则返回 A，否则返回 B。	(A and B) 为 false。
or	逻辑或操作符。若 A 为 true，则返回 A，否则返回 B。	(A or B) 为 true。
not	逻辑非操作符。与逻辑运算结果相反，如果条件为 true，逻辑非为 false。	not(A and B) 为 true。

# OpenResty

## 初识OpenResty

OpenResty® 是一个基于 Nginx 的高性能 Web 平台，用于方便地搭建能够处理超高并发、扩展性极高的动态 Web 应用、Web 服务和动态网关。具备下列特点：

- 具备Nginx的完整功能
- 基于Lua语言进行扩展，集成了大量精良的 Lua 库、第三方模块
- 允许使用Lua自定义业务逻辑、自定义库

官方网站：<https://openresty.org/cn/>



## OpenResty快速入门，实现商品详情页数据查询

商品详情页面目前展示的是假数据，在浏览器的控制台可以看到查询商品信息的请求：

```
▼ General
Request URL: http://localhost/api/item/10001
Request Method: GET
Status Code: 502 Bad Gateway
Remote Address: 127.0.0.1:80
Referrer Policy: strict-origin-when-cross-origin
```

而这个请求最终被反向代理到虚拟机的OpenResty集群：

```
upstream nginx-cluster{
    server 192.168.150.101:8081;                                nginx-cluster集群，就是将来的nginx业务集群
}
server {
    listen 80;
    server_name localhost;
}

location /api {
    proxy_pass http://nginx-cluster;                            监听/api路径，反向代理到nginx-cluster集群
}
```

需求：在OpenResty中接收这个请求，并返回一段商品的假数据。

## 步骤一：修改nginx.conf文件

- 在nginx.conf的http下面，添加对OpenResty的Lua模块的加载：

```
# 加载lua 模块  
lua_package_path "/usr/local/openresty/lualib/?.lua;;";  
# 加载c模块  
lua_package_cpath "/usr/local/openresty/lualib/?.so;;";
```

- 在nginx.conf的server下面，添加对/api/item这个路径的监听：

```
location /api/item {  
    # 响应类型，这里返回json  
    default_type application/json;  
    # 响应数据由 lua/item.lua这个文件来决定  
    content_by_lua_file lua/item.lua;  
}
```

## 步骤二：编写item.lua文件

- 在nginx目录创建文件夹：lua

```
[root@node1 nginx]# pwd  
/usr/local/openresty/nginx  
[root@node1 nginx]# mkdir lua
```

- 在lua文件夹下，新建文件：item.lua

```
[root@node1 nginx]# pwd  
/usr/local/openresty/nginx  
[root@node1 nginx]# touch lua/item.lua
```

- 内容如下：

```
-- 返回假数据，这里的ngx.say()函数，就是写数据到Response中  
ngx.say('{"id":10001,"name":"SALSA AIR"}')
```

- 重新加载配置

```
nginx -s reload
```

## OpenResty获取请求参数

OpenResty提供了各种API用来获取不同类型的请求参数：

参数格式	参数示例	参数解析代码示例
路径占位符	/item/1001	# 1. 正则表达式匹配: location ~ /item/(\d+) { content_by_lua_file lua/item.lua; }  -- 2. 匹配到的参数会存入ngx.var数组中, -- 可以用角标获取 local id = ngx.var[1]
请求头	id: 1001	-- 获取请求头, 返回值是table类型 local headers = ngx.req.get_headers()
Get请求参数	?id=1001	-- 获取GET请求参数, 返回值是table类型 local queryParams = ngx.req.get_uri_args()
Post表单参数	id=1001	-- 读取请求体 ngx.req.read_body() -- 获取POST表单参数, 返回值是table类型 local postParams = ngx.req.get_post_args()
JSON参数	{"id": 1001}	-- 读取请求体 ngx.req.read_body() -- 获取body中的json参数, 返回值是string类型 local jsonBody = ngx.req.get_body_data()

## nginx内部发送Http请求

nginx提供了内部API用以发送http请求：

```
local resp = ngx.location.capture("/path", {  
    method = ngx.HTTP_GET, -- 请求方式  
    args = {a=1,b=2}, -- get方式传参数  
    body = "c=3&d=4" -- post方式传参数  
})
```

返回的响应内容包括：

- resp.status: 响应状态码
- resp.header: 响应头, 是一个table
- resp.body: 响应体, 就是响应数据

**注意：**这里的path是路径，并不包含IP和端口。这个请求会被nginx内部的server监听并处理。

但是我们希望这个请求发送到Tomcat服务器，所以还需要编写一个server来对这个路径做反向代理：

```
location /path {  
    # 这里是windows电脑的ip和Java服务端口, 需要确保windows防火墙处于关闭状态  
    proxy_pass http://192.168.150.1:8081;  
}
```

高级

## 封装http查询的函数

我们可以把http查询的请求封装为一个函数，放到OpenResty函数库中，方便后期使用。

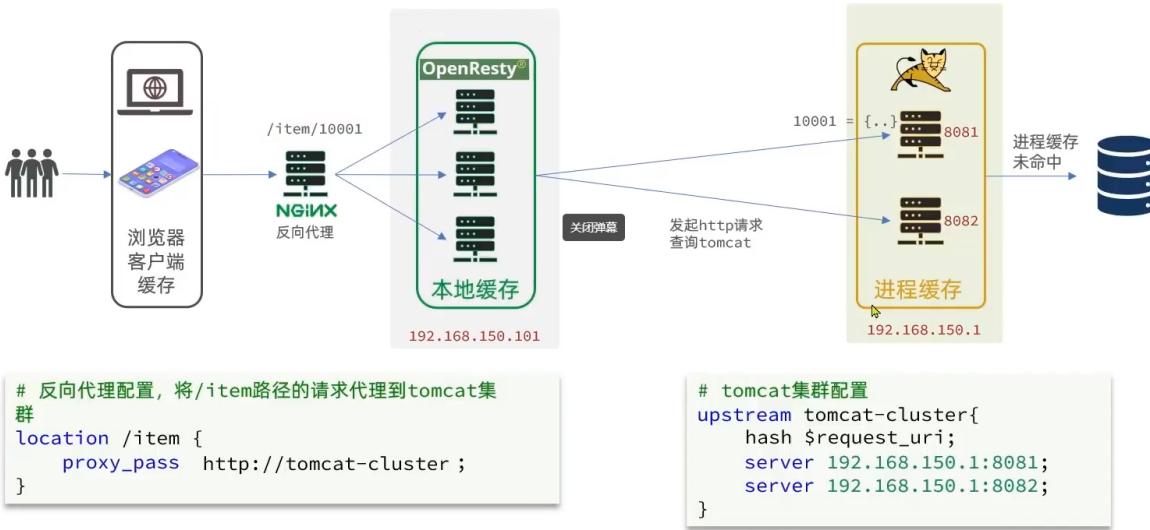
1. 在/usr/local/openresty/lualib目录下创建common.lua文件：

```
vi /usr/local/openresty/lualib/common.lua
```

2. 在common.lua中封装http查询的函数

```
-- 封装函数, 发送http请求, 并解析响应  
local function read_http(path, params)  
    local resp = ngx.location.capture(path, {  
        method = ngx.HTTP_GET,  
        args = params,  
    })  
    if not resp then  
        -- 记录错误信息, 返回404  
        ngx.log(ngx.ERR, "http not found, path: ", path, ", args: ", args)  
        ngx.exit(404)  
    end  
    return resp.body  
end  
-- 将方法导出  
local _M = {  
    read_http = read_http  
}  
return _M
```

## Tomcat集群的负载均衡



## 冷启动与缓存预热

**冷启动**: 服务刚刚启动时, Redis中并没有缓存, 如果所有商品数据都在第一次查询时添加缓存, 可能会给数据库带来较大压力。

**缓存预热**: 在实际开发中, 我们可以利用大数据统计用户访问的热点数据, 在项目启动时将这些热点数据提前查询并保存到Redis中。

### 1. 利用Docker安装Redis

```
docker run --name redis -p 6379:6379 -d redis redis-server --appendonly yes
```

### 2. 在item-service服务中引入Redis依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

### 3. 配置Redis地址

```
spring:
  redis:
    host: 192.168.150.101
```

### 4. 编写初始化类

```
@Component
public class RedisHandler implements InitializingBean {
    @Autowired
    private StringRedisTemplate redisTemplate;
    @Override
    public void afterPropertiesSet() throws Exception { // 初始化缓存 ... }
}
```

## OpenResty的Redis模块

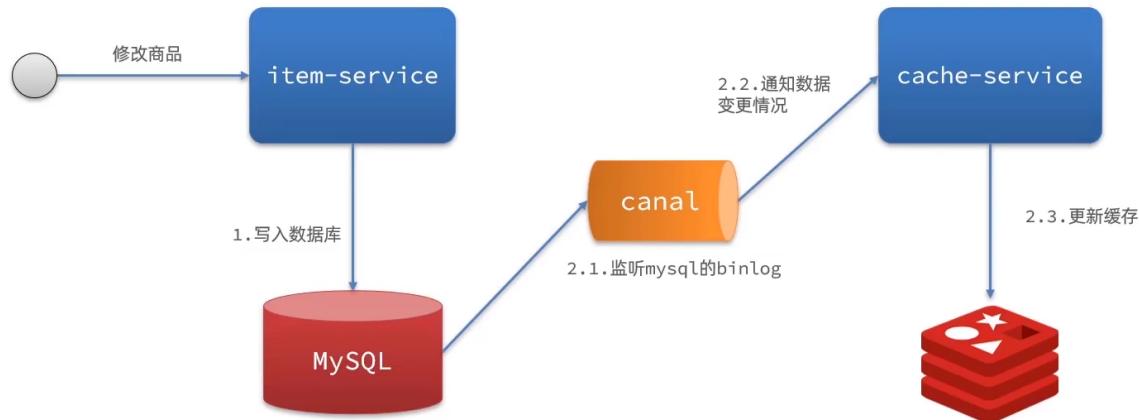
OpenResty提供了操作Redis的模块，我们只要引入该模块就能直接使用：

- 封装函数，从Redis读数据并返回

```
-- 查询redis的方法 ip和port是redis地址, key是查询的key
local function read_redis(ip, port, key)
    -- 获取一个连接
    local ok, err = red:connect(ip, port)
    if not ok then
        ngx.log(ngx.ERR, "连接redis失败 : ", err)
        return nil
    end
    -- 查询redis
    local resp, err = red:get(key)
    -- 查询失败处理
    if not resp then
        ngx.log(ngx.ERR, "查询Redis失败: ", err, ", key = " , key)
    end
    --得到的数据为空处理
    if resp == ngx.null then
        resp = nil
        ngx.log(ngx.ERR, "查询Redis数据为空, key = " , key)
    end
    close_redis(red)
    return resp
end
```

## 缓存同步策略

基于Canal的异步通知：



## Canal

和数据库的协同安装查看文档

## Canal客户端

Canal提供了各种语言的客户端，当Canal监听到binlog变化时，会通知Canal的客户端。不过这里我们会使用GitHub上的第三方开源的canal-starter。地址：<https://github.com/NormanGyllenhaal/canal-client>

引入依赖：

```
<!--canal-->
<dependency>
    <groupId>top.javatool</groupId>
    <artifactId>canal-spring-boot-starter</artifactId>
    <version>1.2.1-RELEASE</version>
</dependency>
```

编写配置：

```
canal:
  destination: heima # canal实例名称，要跟canal-server运行时设置的destination一致
  server: 192.168.150.101:11111 # canal地址
```

## Canal客户端

编写监听器，监听Canal消息：

```
package com.heima.item.canal;

@Component
public class ItemHandler implements EntryHandler<Item> {

    @Override
    public void insert(Item item) {
        // 新增数据到redis
    }

    @Override
    public void update(Item before, Item after) {
        // 更新redis数据
        // 更新本地缓存
    }

    @Override
    public void delete(Item item) {
        // 删除redis数据
        // 清理本地缓存
    }
}
```

## Canal客户端

Canal推送给canal-client的是被修改的这一行数据（row），而我们引入的canal-client则会帮我们把行数据封装到Item实体类中。这个过程中需要知道数据库与实体的映射关系，要用到JPA的几个注解：

```
@Data
@TableName("tb_item")
public class Item {
    @TableId(type = IdType.AUTO)
    @Id
    private Long id; // 标记表中的id字段

    @Column(name = "name")
    private String name; // 标记表中与属性名不一致的字段

    // ... 其它字段略
    private Date updateTime;
    @TableField(exist = false)
    @Transient
    private Integer stock; // 标记不属于表中的字段
    @TableField(exist = false)
    @Transient
    private Integer sold;
}
```

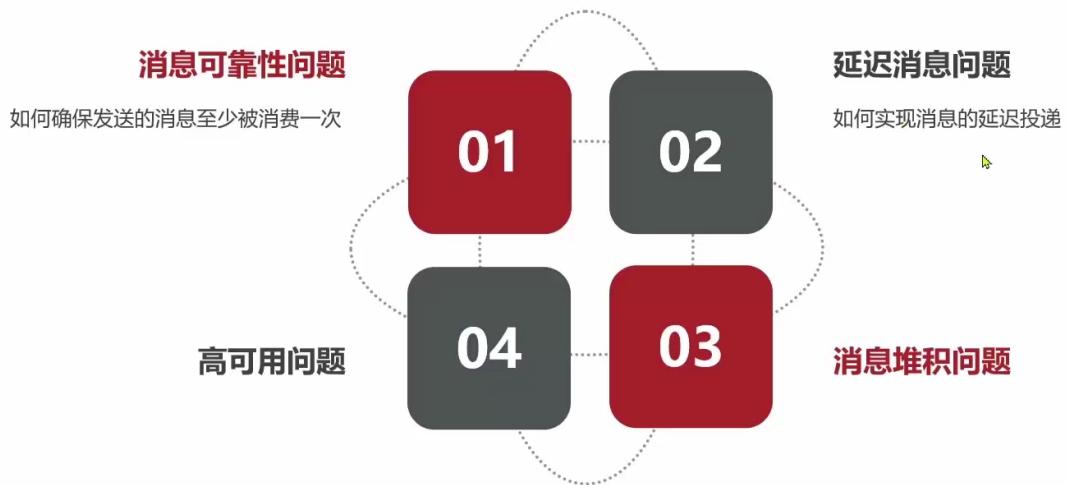
# MQ

## 缓存同步策略

缓存数据同步的常见方式有三种：

- **设置有效期：**给缓存设置有效期，到期后自动删除。再次查询时更新
  - 优势：简单、方便
  - 缺点：时效性差，缓存过期之前可能不一致
  - 场景：更新频率较低，时效性要求低的业务
- **同步双写：**在修改数据库的同时，直接修改缓存
  - 优势：时效性强，缓存与数据库强一致
  - 缺点：有代码侵入，耦合度高；
  - 场景：对一致性、时效性要求较高的缓存数据
- **异步通知：**修改数据库时发送事件通知，相关服务监听到通知后修改缓存数据
  - 优势：低耦合，可以同时通知多个缓存服务
  - 缺点：时效性一般，可能存在中间不一致状态
  - 场景：时效性要求一般，有多个服务需要同步

## MQ的一些常见问题



# 生产者消息确认

## 生产者确认机制

RabbitMQ提供了publisher confirm机制来避免消息发送到MQ过程中丢失。消息发送到MQ以后，会返回一个结果给发送者，表示消息是否处理成功。结果有两种请求：

- publisher-confirm，发送者确认
  - 消息成功投递到交换机，返回ack
  - 消息未投递到交换机，返回nack
- publisher-return，发送者回执
  - 消息投递到交换机了，但是没有路由到队列。返回ACK，及路由失败原因。

## SpringAMQP实现生产者确认

1. 在publisher这个微服务的application.yml中添加配置：

```
spring:  
  rabbitmq:  
    publisher-confirm-type: correlated  
    publisher-returns: true  
    template:  
      mandatory: true
```

配置说明：

- publish-confirm-type：开启publisher-confirm，这里支持两种类型：
  - simple：同步等待confirm结果，直到超时
  - correlated：异步回调，定义ConfirmCallback，MQ返回结果时会回调这个ConfirmCallback
- publish-returns：开启publish-return功能，同样是基于callback机制，不过是定义ReturnCallback
- template.mandatory：定义消息路由失败时的策略。true，则调用ReturnCallback；false：则直接丢弃消息

## SpringAMQP实现生产者确认

2. 每个RabbitTemplate只能配置一个ReturnCallback，因此需要在项目启动过程中配置：

```
@Slf4j  
@Configuration  
public class CommonConfig implements ApplicationContextAware {  
    @Override  
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {  
        // 获取RabbitTemplate  
        RabbitTemplate rabbitTemplate = applicationContext.getBean(RabbitTemplate.class);  
        // 设置ReturnCallback  
        rabbitTemplate.setReturnCallback((message, replyCode, replyText, exchange, routingKey) -> {  
            log.info("消息发送失败，应答码{}，原因{}，交换机{}，路由键{}，消息{}",  
                replyCode, replyText, exchange, routingKey, message.toString());  
        });  
    }  
}
```

## SpringAMQP实现生产者确认

### 3. 发送消息，指定消息ID、消息ConfirmCallback

```
@Test
public void testSendMessage2SimpleQueue() throws InterruptedException {
    // 消息体
    String message = "hello, spring amqp!";
    // 消息ID, 需要封装到CorrelationData中
    CorrelationData correlationData = new CorrelationData(UUID.randomUUID().toString());
    // 添加callback
    correlationData.getFuture().addCallback(
        result -> {
            if(result.isAck()){
                // ack, 消息成功
                log.debug("消息发送成功, ID:{}" , correlationData.getId());
            }else{
                // nack, 消息失败
                log.error("消息发送失败, ID:{}, 原因{}", correlationData.getId(), result.getReason());
            }
        },
        ex -> log.error("消息发送异常, ID:{}, 原因{}", correlationData.getId(), ex.getMessage())
    );
    // 发送消息
    rabbitTemplate.convertAndSend("amq.direct", "simple", message, correlationData);
}
```

```
@Test
```

```
public void testSendMessage2SimpleQueue()
throws InterruptedException {
    String routingKey = "simple.test";
    String message = "hello, spring amqp!";

    CorrelationData correlationData = new
CorrelationData(UUID.randomUUID().toString());

    correlationData.getFuture().addCallback(
        // 成功 到交换机
        new
SuccessCallback<CorrelationData.Confirm>() {
            @Override
            public void
onSuccess(CorrelationData.Confirm confirm) {
                // 判断是否成功?
                if (confirm.isAck()) {
                    log.info("发送成功");
                }else {
                    log.info("消息投递到交换机
失败 id为{}", correlationData.getId());
                }
            }
        },
        // 失败 到队列
    );
}
```

```

        new FailureCallback() {
            @Override
            public void onFailure(Throwable
throwable) {
                log.info("消息失败 原因:
{}", throwable);

                //rabbitTemplate.convertAndSend("amq.topic",
routingKey, message, correlationData);
            }
        });

        // 到了交换机 但是没有到队列会触发
returnedMessage
        rabbitTemplate.convertAndSend("amq.topic",
routingKey, message, correlationData);
    }
}

```

## 持久化

### 消息持久化

MQ默认是内存存储消息，开启持久化功能可以确保缓存在MQ中的消息不丢失。

1. 交换机持久化：

```

@Bean
public DirectExchange simpleExchange(){
    // 三个参数：交换机名称、是否持久化、当没有queue与其绑定时是否自动删除
    return new DirectExchange("simple.direct", true, false);
}

```

2. 队列持久化：

```

@Bean
public Queue simpleQueue(){
    // 使用QueueBuilder构建队列，durable就是持久化的
    return QueueBuilder.durable("simple.queue").build();
}

```

## 消息持久化

MQ默认是内存存储消息，开启持久化功能可以确保缓存在MQ中的消息不丢失。

### 1. 交换机持久化：

```
@Bean  
public DirectExchange simpleExchange(){  
    // 三个参数：交换机名称、是否持久化、当没有queue与其绑定时是否自动删除  
    return new DirectExchange("simple.direct", true, false);  
}
```

### 2. 队列持久化：

```
@Bean  
public Queue simpleQueue(){  
    // 使用QueueBuilder构建队列，durable就是持久化的  
    return QueueBuilder.durable("simple.queue").build();  
}
```

3. 消息持久化，SpringAMQP中的的消息默认是持久的，可以通过MessageProperties中的DeliveryMode来指定的：

```
Message msg = MessageBuilder  
    .withBody(message.getBytes(StandardCharsets.UTF_8)) // 消息体  
    .setDeliveryMode(MessageDeliveryMode.PERSISTENT) // 持久化  
    .build();
```

高级软件人才培训

## 默认都是持久化

## 消费者消息确认

RabbitMQ支持消费者确认机制，即：消费者处理消息后可以向MQ发送ack回执，MQ收到ack回执后才会删除该消息。

而SpringAMQP则允许配置三种确认模式：

- manual: 手动ack，需要在业务代码结束后，调用api发送ack。
- auto: 自动ack，由spring监测listener代码是否出现异常，没有异常则返回ack；抛出异常则返回nack
- none: 关闭ack，MQ假定消费者获取消息后会成功处理，因此消息投递后立即被删除

配置方式是修改application.yml文件，添加下面配置：

```
spring:  
  rabbitmq:  
    listener:  
      simple:  
        prefetch: 1  
        acknowledge-mode: none # none, 关闭ack; manual, 手动ack; auto: 自动ack
```

▶

## 失败重试机制

## 消费者失败重试

当消费者出现异常后，消息会不断requeue（重新入队）到队列，再重新发送给消费者，然后再次异常，再次requeue，无限循环，导致mq的消息处理飙升，带来不必要的压力：

Overview				Messages			Message rates			+/-
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
simple.queue	classic	D	running	0	1	1	0.00/s	3,370/s	0.00/s	

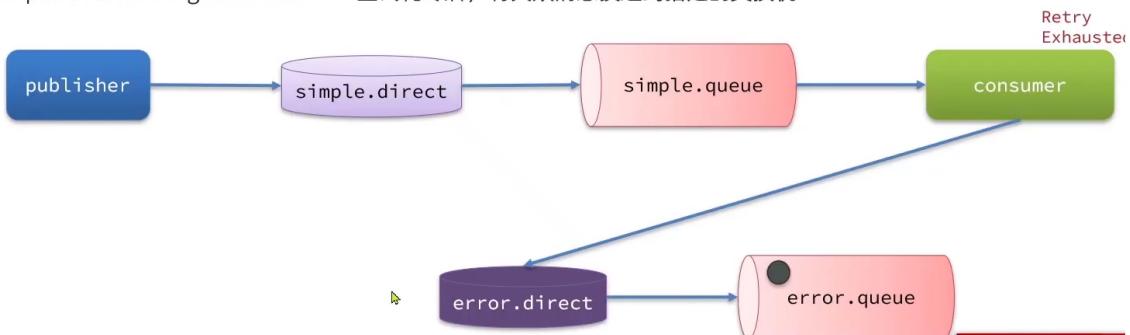
我们可以利用Spring的retry机制，在消费者出现异常时利用本地重试，而不是无限制的requeue到mq队列。

```
spring:
  rabbitmq:
    listener:
      simple:
        prefetch: 1
        retry:
          enabled: true # 开启消费者失败重试
          initial-interval: 1000 # 初识的失败等待时长为1秒
          multiplier: 1 # 下次失败的等待时长倍数, 下次等待时长 = multiplier * last-interval
          max-attempts: 3 # 最大重试次数
        stateless: true # true无状态; false有状态。如果业务中包含事务, 这里改为false
```

## 消费者失败消息处理策略

在开启重试模式后，重试次数耗尽，如果消息依然失败，则需要有MessageRecoverer接口来处理，它包含三种不同的实现：

- `RejectAndDontRequeueRecoverer`: 重试耗尽后, 直接reject, 丢弃消息。默认就是这种方式
  - `ImmediateRequeueMessageRecoverer`: 重试耗尽后, 返回nack, 消息重新入队
  - `RepublishMessageRecoverer`: 重试耗尽后, 将失败消息投递到指定的交换机



消费者失败消息处理策略

测试下RepublishMessageRecoverer处理模式：

- 首先，定义接收失败消息的交换机、队列及其绑定关系：

```
@Bean
public DirectExchange errorMessageExchange(){
    return new DirectExchange("error.direct");
}
@Bean
public Queue errorQueue(){
    return new Queue("error.queue", true);
}
@Bean
public Binding errorBinding(){
    return BindingBuilder.bind(errorQueue()).to(errorMessageExchange()).with("error");
}
```

- 然后，定义RepublishMessageRecoverer：

```
@Bean  
public MessageRecoverer republishMessageRecoverer(RabbitTemplate rabbitTemplate){  
    return new RepublishMessageRecoverer(rabbitTemplate, "error.direct", "error")  
}
```

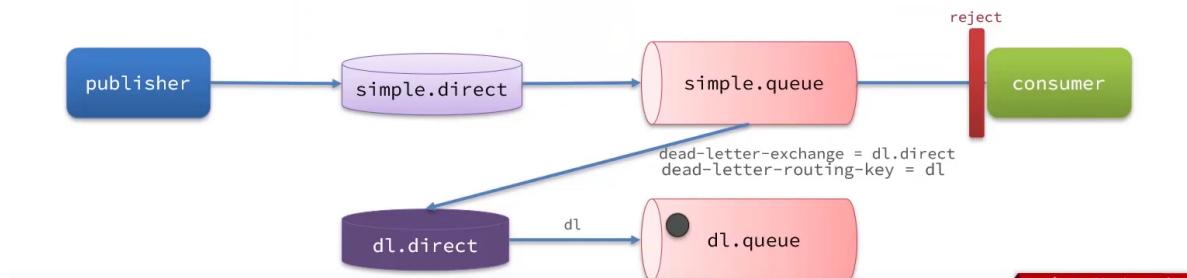
# 死信交换机

## 初识死信交换机

当一个队列中的消息满足下列情况之一时，可以成为死信（dead letter）：

- 消费者使用basic.reject或 basic.nack声明消费失败，并且消息的requeue参数设置为false
- 消息是一个过期消息，超时无人消费
- 要投递的队列消息堆积满了，最早的消息可能成为死信

如果该队列配置了dead-letter-exchange属性，指定了一个交换机，那么队列中的死信就会投递到这个交换机中，而这个交换机称为死信交换机（Dead Letter Exchange，简称DLX）。

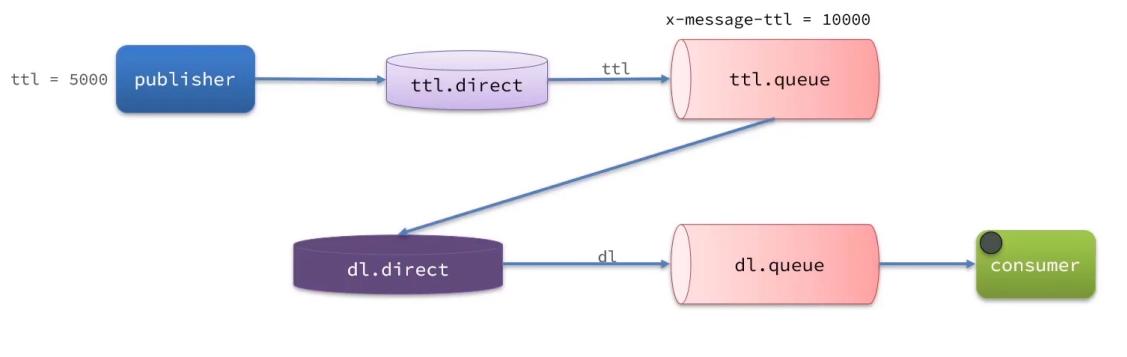


# TTL

## TTL

TTL，也就是Time-To-Live。如果一个队列中的消息TTL结束仍未消费，则会变为死信，ttl超时分为两种情况：

- 消息所在的队列设置了存活时间
- 消息本身设置了存活时间



## TTL

要给队列设置超时时间，需要在声明队列时配置x-message-ttl属性：

```
@Bean
public DirectExchange ttlExchange(){
    return new DirectExchange("ttl.direct");
}
@Bean
public Queue ttlQueue(){
    return QueueBuilder.durable("ttl.queue") // 指定队列名称，并持久化
        .ttl(10000) // 设置队列的超时时间，10秒
        .deadLetterExchange("dl.direct") // 指定死信交换机
        .deadLetterRoutingKey("dl") // 指定死信RoutingKey
        .build();
}
@Bean
public Binding simpleBinding(){
    return BindingBuilder.bind(ttlQueue()).to(ttlExchange()).with("ttl");
}
```

## TTL

发送消息时，给消息本身设置超时时间

```
@Test
public void testTTLMsg() {
    // 创建消息
    Message message = MessageBuilder
        .withBody("hello, ttl message".getBytes(StandardCharsets.UTF_8))
        .setExpiration("5000")
        .build();
    // 消息ID，需要封装到CorrelationData中
    CorrelationData correlationData = new CorrelationData(UUID.randomUUID().toString());
    // 发送消息
    rabbitTemplate.convertAndSend("ttl.direct", "ttl", message, correlationData);
}
```

// 监听延时消息

```
@Slf4j
@Component
```

```
public class DlMsgListen {
```

```
    @RabbitListener(bindings = @QueueBinding(
        value = @Queue(name = "dl.queue",
durable = "true"),
        exchange = @Exchange(name =
"dl.direct"),
        key = "dl"
    ))
    public void dlMsg(String msg){
        log.info("接收到延时消息：【{}】",msg);
    }
}
```

```

@Configuration
public class TTLMsgConfig {

    @Bean
    public Queue ttlQueue(){
        return QueueBuilder.durable("ttl.queue")
            .ttl(10000)
            // 超时消息的交换机
            .deadLetterExchange("dl.direct")
            // 超时消息的Key
            .deadLetterRoutingKey("dl")
            .build();
    }

    @Bean
    public DirectExchange ttlExchange(){
        return new DirectExchange("ttl.direct");
    }

    @Bean
    public Binding ttlBindDing(){
        return
BindingBuilder.bind(ttlQueue()).to(ttlExchange()).w
ith("ttl");
    }
}

```

## 延迟队列

### SpringAMQP使用延迟队列插件

DelayExchange的本质还是官方的三种交换机，只是添加了延迟功能。因此使用时只需要声明一个交换机，交换机的类型可以是任意类型，然后设定delayed属性为true即可。

基于注解方式：

```

@RabbitListener(bindings = @QueueBinding(
    value = @Queue(name = "delay.queue", durable = "true"),
    exchange = @Exchange(name = "delay.direct", delayed = "true"),
    key = "delay"
))
public void listenDelayedQueue(String msg){
    log.info("接收到 delay.queue的延迟消息: {}", msg);
}

```

基于java代码的方式：

```
@Bean
public DirectExchange delayedExchange(){
    return ExchangeBuilder
        .directExchange("delay.direct") // 指定交换机类型和名称
        .delayed() // 设置delay属性为 true
        .durable(true) // 持久化
        .build();
}

@Bean
public Queue delayedQueue(){
    return new Queue( name: "delay.queue");
}

@Bean
public Binding delayedBinding(){
    return BindingBuilder.bind(delayedQueue()).to(delayedExchange()).with( routingKey: "delay");
}
```

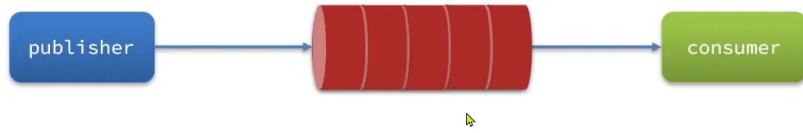
然后我们向这个delay为true的交换机中发送消息，一定要给消息添加一个header: x-delay，值为延迟的时间，单位为毫秒：

```
@Test
public void testDelayedMsg() {
    // 创建消息
    Message message = MessageBuilder
        .withBody("hello, delayed message".getBytes(StandardCharsets.UTF_8))
        .setHeader("x-delay", 10000)
        .build();
    // 消息ID，需要封装到CorrelationData中
    CorrelationData correlationData = new CorrelationData(UUID.randomUUID().toString());
    // 发送消息
    rabbitTemplate.convertAndSend( exchange: "delay.direct", routingKey: "delay", message, correlationData);
    log.debug("发送消息成功");
}
```

## 消息堆积问题

## 消息堆积问题

当生产者发送消息的速度超过了消费者处理消息的速度，就会导致队列中的消息堆积，直到队列存储消息达到上限。最早接收到的消息，可能就会成为死信，会被丢弃，这就是消息堆积问题。



解决消息堆积有三种种思路：

- 增加更多消费者，提高消费速度
- 在消费者内开启线程池加快消息处理速度
- 扩大队列容积，提高堆积上限



## 惰性队列

从RabbitMQ的3.6.0版本开始，就增加了Lazy Queues的概念，也就是惰性队列。

惰性队列的特征如下：

- 接收到消息后直接存入磁盘而非内存
- 消费者要消费消息时才会从磁盘中读取并加载到内存
- 支持数百万条的消息存储

而要设置一个队列为惰性队列，只需要在声明队列时，指定x-queue-mode属性为lazy即可。可以通过命令行将一个运行中的队列修改为惰性队列：

```
rabbitmqctl set_policy Lazy "^lazy-queue$" '{"queue-mode":"lazy"}' --apply-to queues
```

用SpringAMQP声明惰性队列分两种方式：

- @Bean的方式：

```
@Bean
public Queue lazyQueue(){
    return QueueBuilder
        .durable("lazy.queue")
        .lazy() // 开启x-queue-mode为lazy
        .build();
}
```

- 注解方式：

```
@RabbitListener(queuesToDeclare = @Queue(
    name = "lazy.queue",
    durable = "true",
    arguments = @Argument(name = "x-queue-mode", value = "lazy"))
)
public void listenLazyQueue(String msg){
    log.info("接收到 lazy.queue的消息: {}", msg);
}
```

