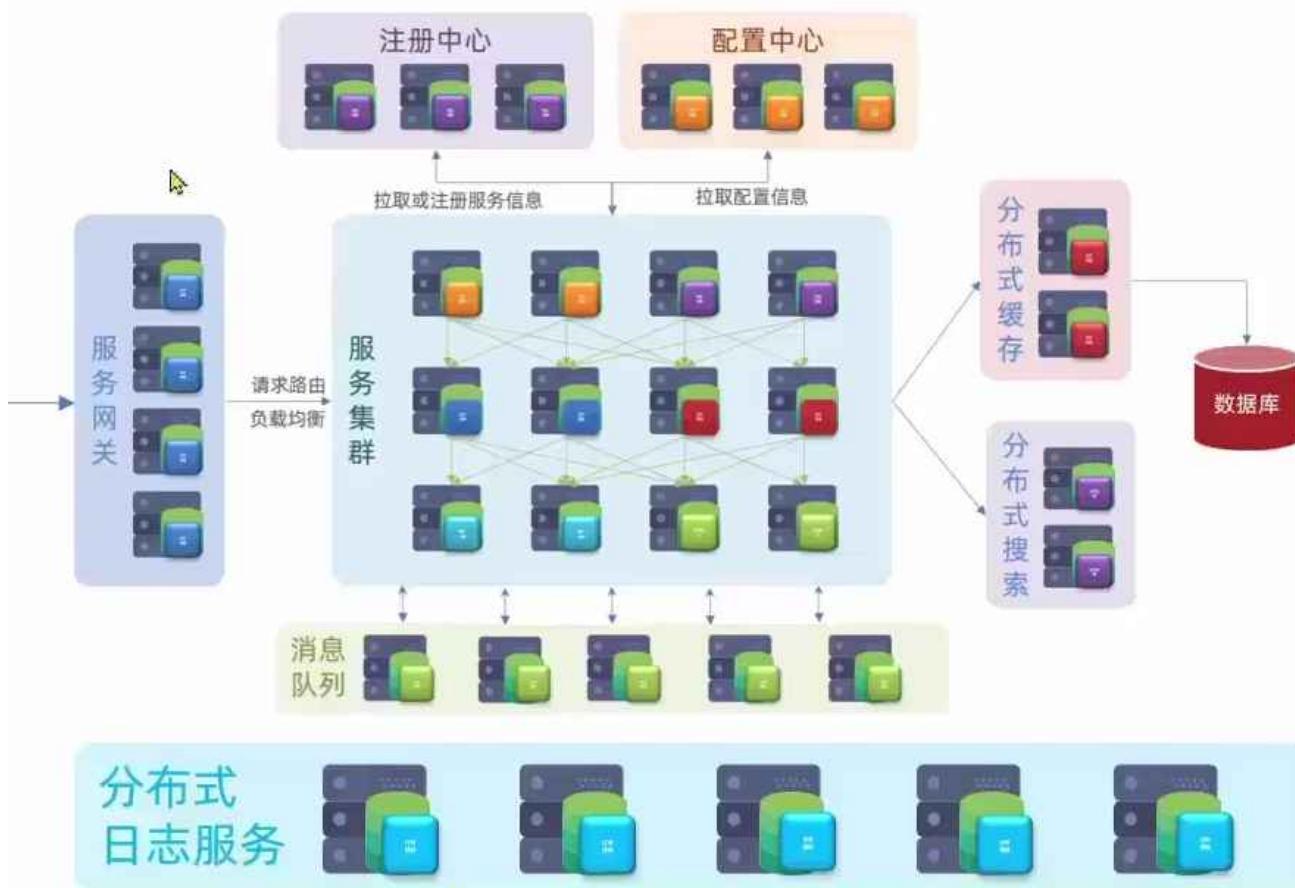


Springcloud



系统监控 链路追踪



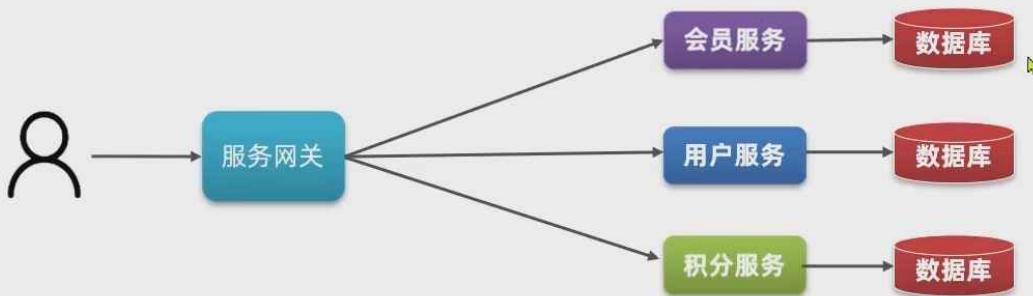
分布式 日志服务



微服务

微服务是一种经过良好架构设计的**分布式**架构方案，微服务架构特征：

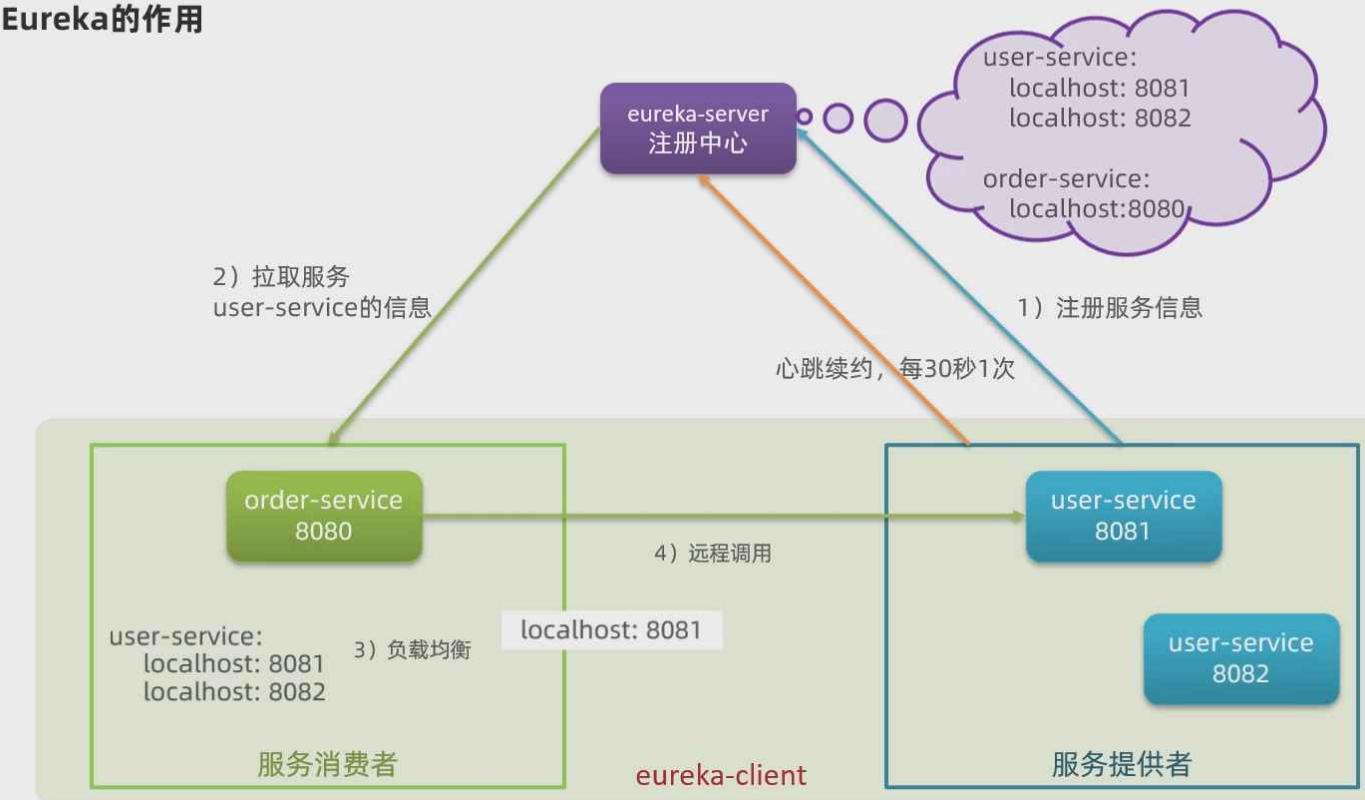
- 单一职责：微服务拆分粒度更小，每一个服务都对应唯一的业务能力，做到单一职责，避免重复业务开发
- 面向服务：微服务对外暴露业务接口
- 自治：团队独立、技术独立、数据独立、部署独立
- 隔离性强：服务调用做好隔离、容错、降级，避免出现级联问题



eureka

Eureka注册中心

Eureka的作用



在Eureka架构中，微服务角色有两类：

- **EurekaServer:** 服务端，注册中心
 - ◆ 记录服务信息
 - ◆ 心跳监控
- **EurekaClient:** 客户端
 - ◆ Provider: 服务提供者，例如案例中的 user-service
 - ◆ 注册自己的信息到EurekaServer
 - ◆ 每隔30秒向EurekaServer发送心跳
 - ◆ consumer: 服务消费者，例如案例中的 order-service
 - ◆ 根据服务名称从EurekaServer拉取服务列表
 - ◆ 基于服务列表做负载均衡，选中一个微服务后发起远程调用

1. 搭建EurekaServer

- 引入eureka-server依赖
- 添加@EnableEurekaServer注解
- 在application.yml中配置eureka地址

2. 服务注册

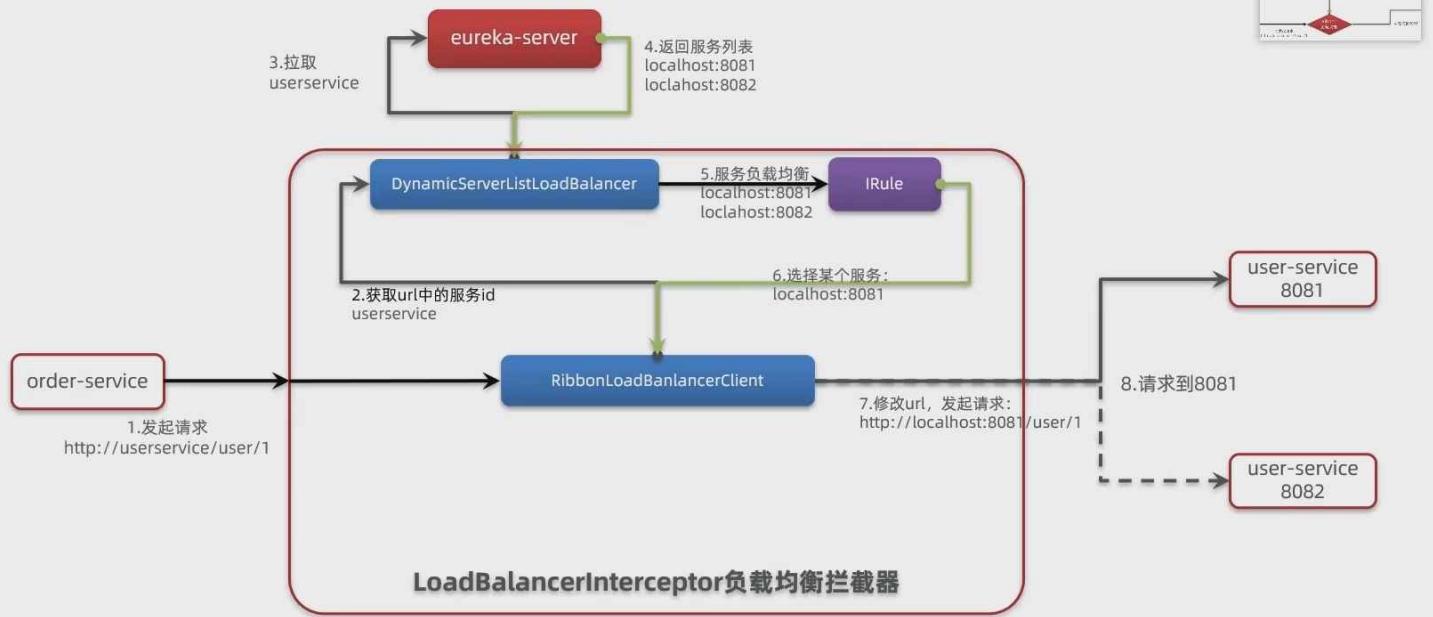
- 引入eureka-client依赖
- 在application.yml中配置eureka地址

3. 服务发现

- 引入eureka-client依赖
- 在application.yml中配置eureka地址
- 给RestTemplate添加@LoadBalanced注解
- 用服务提供者的服务名称远程调用

Ribbon

负载均衡流程



负载均衡策略

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 <code><clientName>.<clientConfigNameSpace>.ActiveConnectionsLimit</code> 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

负载均衡策略

通过定义`IRule`实现可以修改负载均衡规则，有两种方式：

1. 代码方式：在`order-service`中的`OrderApplication`类中，定义一个新的`IRule`：

```
@Bean  
public IRule randomRule(){  
    return new RandomRule();  
}
```

2. 配置文件方式：在`order-service`的`application.yml`文件中，添加新的配置也可以修改规则：

```
userservice:  
  ribbon:  
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule# 负载均衡规则
```

饥饿加载

Ribbon默认是采用懒加载，即第一次访问时才会去创建`LoadBalanceClient`，请求时间会很长。

而饥饿加载则会在项目启动时创建，降低第一次访问的耗时，通过下面配置开启饥饿加载：

```
ribbon:  
  eager-load:  
    enabled: true # 开启饥饿加载  
    clients: userservice # 指定对userservice这个服务饥饿加载
```

nacos

注册中心

1. Nacos服务搭建

- ① 下载安装包
- ② 解压
- ③ 在bin目录下运行指令：startup.cmd -m standalone

2. Nacos服务注册或发现

- ① 引入nacos.discovery依赖
- ② 配置nacos地址spring.cloud.nacos.server-addr

1. Nacos服务分级存储模型

- ① 一级是服务，例如userservice
- ② 二级是集群，例如杭州或上海
- ③ 三级是实例，例如杭州机房的某台部署了userservice的服务器

2. 如何设置实例的集群属性

- ① 修改application.yml文件，添加spring.cloud.nacos.discovery.cluster-name属性即可

. NacosRule负载均衡策略

- ① 优先选择同集群服务实例列表
- ② 本地集群找不到提供者，才去其它集群寻找，并且会报警告
- ③ 确定了可用实例列表后，再采用随机负载均衡挑选实例

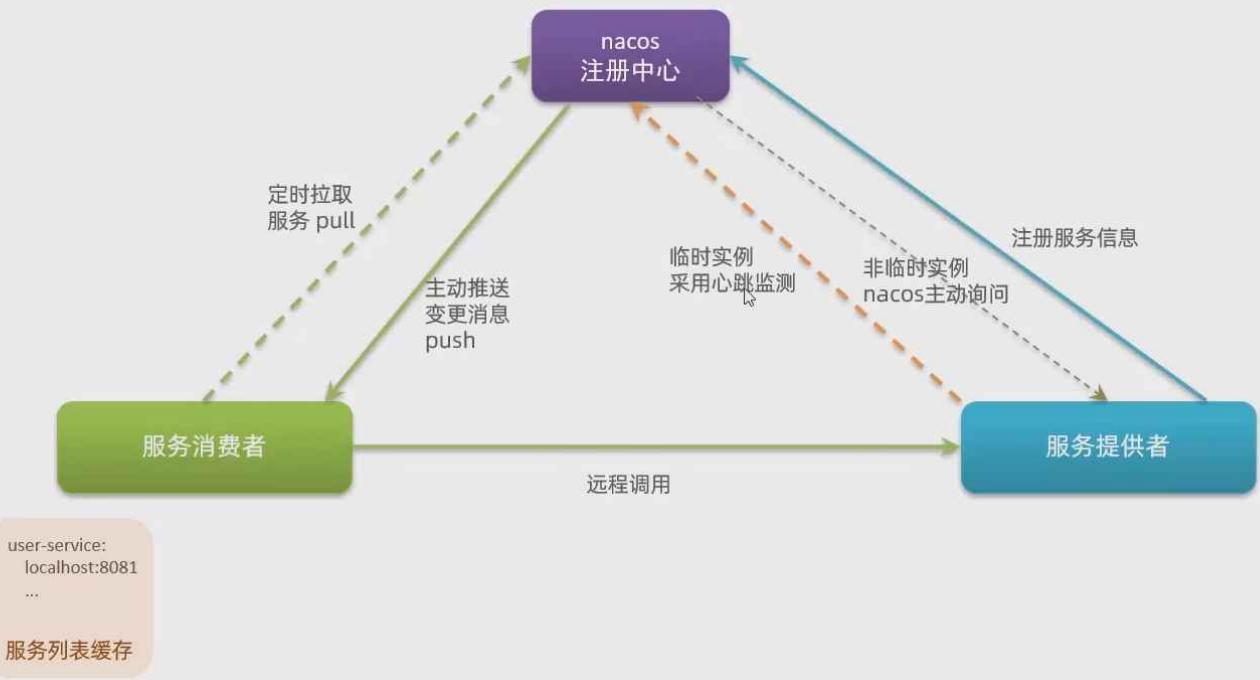
1. 实例的权重控制

- ① Nacos控制台可以设置实例的权重值，0~1之间
- ② 同集群内的多个实例，权重越高被访问的频率越高
- ③ 权重设置为0则完全不会被访问

1. Nacos环境隔离

- ① namespace用来做环境隔离
- ② 每个namespace都有唯一id
- ③ 不同namespace下的服务不可见

nacos注册中心细节分析



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

将配置交给Nacos管理的步骤

- ① 在Nacos中添加配置文件
- ② 在微服务中引入nacos的config依赖
- ③ 在微服务中添加bootstrap.yml，配置nacos地址、
当前环境、服务名称、文件后缀名。这些决定了程序
启动时去nacos读取哪个文件

Nacos配置更改后，微服务可以实现热更新，方式：

- ① 通过@Value注解注入，结合@RefreshScope来刷新
- ② 通过@ConfigurationProperties注入，自动刷新

注意事项：

- 不是所有的配置都适合放到配置中心，维护起来比较麻烦
- 建议将一些关键参数，需要运行时调整的参数放到nacos配置中心，一般都是自定义配置

微服务会从nacos读取的配置文件：

- ① [服务名]-[spring.profile.active].yaml, 环境配置
- ② [服务名].yaml, 默认配置, 多环境共享

优先级：

- ① [服务名]-[环境].yaml > [服务名].yaml > 本地配置

集群搭建步骤：

- ① 搭建MySQL集群并初始化数据库表
- ② 下载解压nacos
- ③ 修改集群配置（节点信息）、数据库配置
- ④ 分别启动多个nacos节点
- ⑤ nginx反向代理

Feign

使用Feign的步骤如下：

3. 编写Feign客户端：

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

主要是基于SpringMVC的注解来声明远程调用的信息，比如：

- 服务名称：userservice
- 请求方式：GET
- 请求路径：/user/{id}
- 请求参数：Long id
- 返回值类型：User

Feign的使用步骤

① 引入依赖

② 添加@EnableFeignClients注解

③ 编写FeignClient接口

④ 使用FeignClient中定义的方法代替RestTemplate

Feign的日志配置：

1. 方式一是配置文件，`feign.client.config.xxx.loggerLevel`
 - ①如果xxx是default则代表全局
 - ②如果xxx是服务名称，例如userservice则代表某服务
2. 方式二是java代码配置`Logger.Level`这个Bean
 - ①如果在`@EnableFeignClients`注解声明则代表全局
 - ②如果在`@FeignClient`注解中声明则代表某服务

Feign的优化：

1. 日志级别尽量用basic
2. 使用`HttpClient`或`OKHttp`代替`URLConnection`
 - ① 引入`feign-httpClient`依赖
 - ② 配置文件开启`httpClient`功能，设置连接池参数

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})
```

gateway

网关的作用：

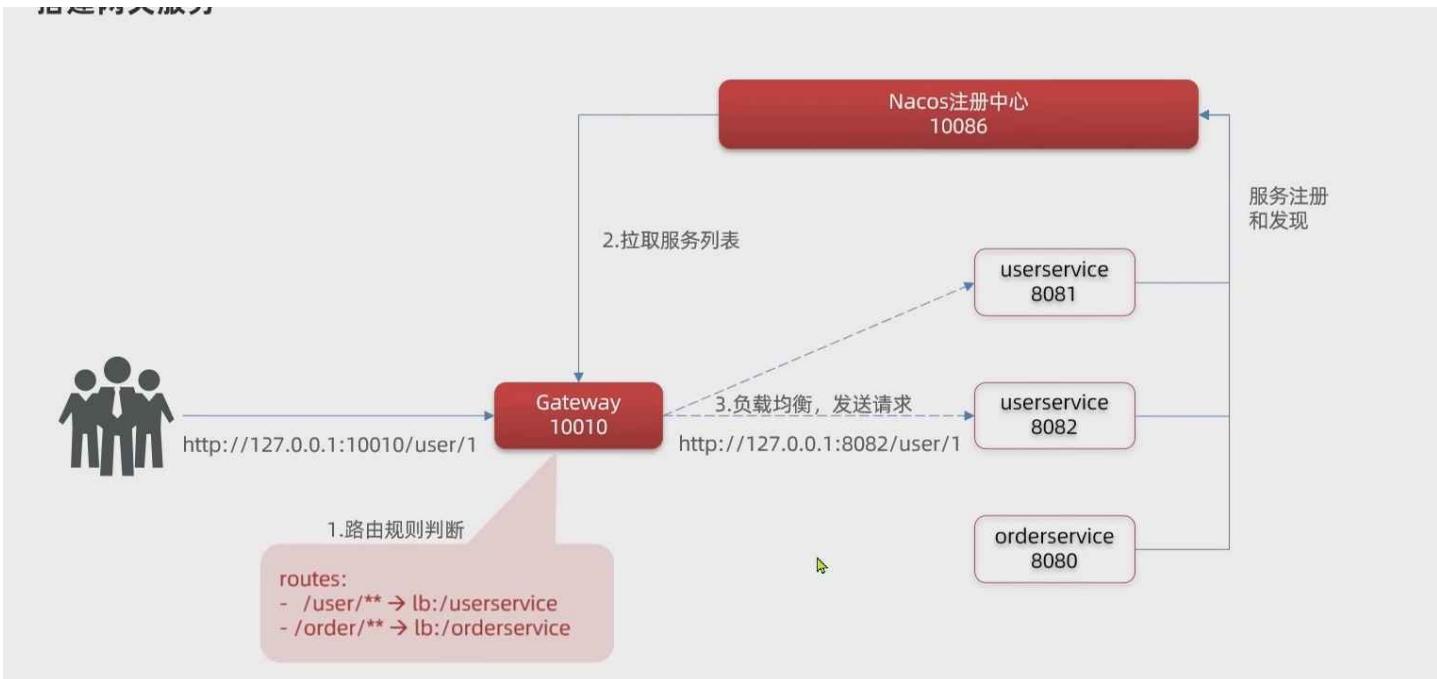
- 对用户请求做身份认证、权限校验
- 将用户请求路由到微服务，并实现负载均衡
- 对用户请求做限流

2. 编写路由配置及nacos地址

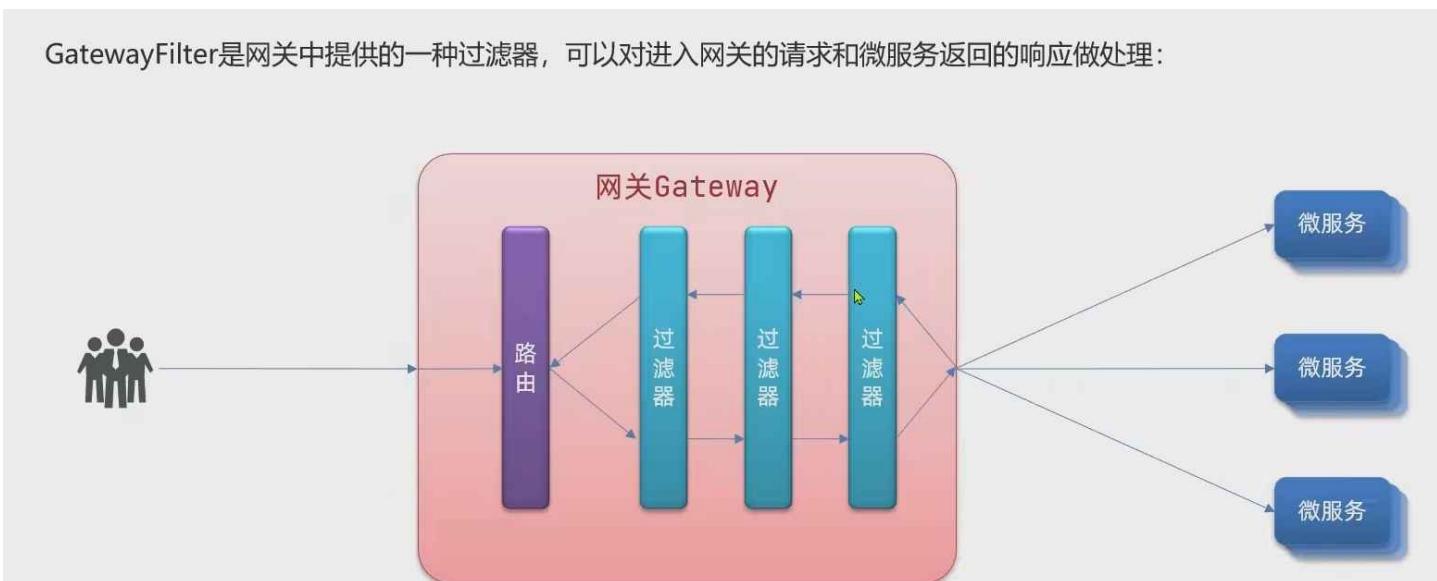
```
server:  
  port: 10010 # 网关端口  
spring:  
  application:  
    name: gateway # 服务名称  
  cloud:  
    nacos:  
      server-addr: localhost:8848 # nacos地址  
    gateway:  
      routes: # 网关路由配置  
        - id: user-service # 路由id, 自定义, 只要唯一即可  
          # uri: http://127.0.0.1:8081 # 路由的目标地址 http就是固定地址  
          uri: lb://userservice # 路由的目标地址 lb就是负载均衡, 后面跟服务名称  
          predicates: # 路由断言, 也就是判断请求是否符合路由规则的条件  
            - Path=/user/** # 这个是按照路径匹配, 只要以/user/开头就符合要求
```

路由断言工厂Route Predicate Factory

- 我们在配置文件中写的断言规则只是字符串，这些字符串会被Predicate Factory读取并处理，转变为路由判断的条件
- 例如Path=/user/**是按照路径匹配，这个规则是由org.springframework.cloud.gateway.handler.predicate.PathRoutePredicateFactory类来处理的
- 像这样的断言工厂在SpringCloudGateway还有十几个



GatewayFilter是网关中提供的一种过滤器，可以对进入网关的请求和微服务返回的响应做处理：



● 过滤器的作用是什么？

- ① 对路由的请求或响应做加工处理，比如添加请求头
- ② 配置在路由下的过滤器只对当前路由的请求生效

● defaultFilters的作用是什么？

- ① 对所有路由都生效的过滤器

步骤1：自定义过滤器

自定义类，实现GlobalFilter接口，添加@Order注解：

```
@Order(-1)
@Component
public class AuthorizeFilter implements GlobalFilter {
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        // 1. 获取请求参数
        MultiValueMap<String, String> params = exchange.getRequest().getQueryParams();
        // 2. 获取authorization参数
        String auth = params.getFirst("authorization");
        // 3. 校验
        if ("admin".equals(auth)) {
            // 放行
            return chain.filter(exchange);
        }
        // 4. 拦截
        // 4.1. 禁止访问
        exchange.getResponse().setStatusCode(HttpStatus.FORBIDDEN);
        // 4.2. 结束处理
        return exchange.getResponse().setComplete();
    }
}
```

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。

跨域问题处理

网关处理跨域采用的同样是CORS方案，并且只需要简单配置即可实现：

```
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

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**

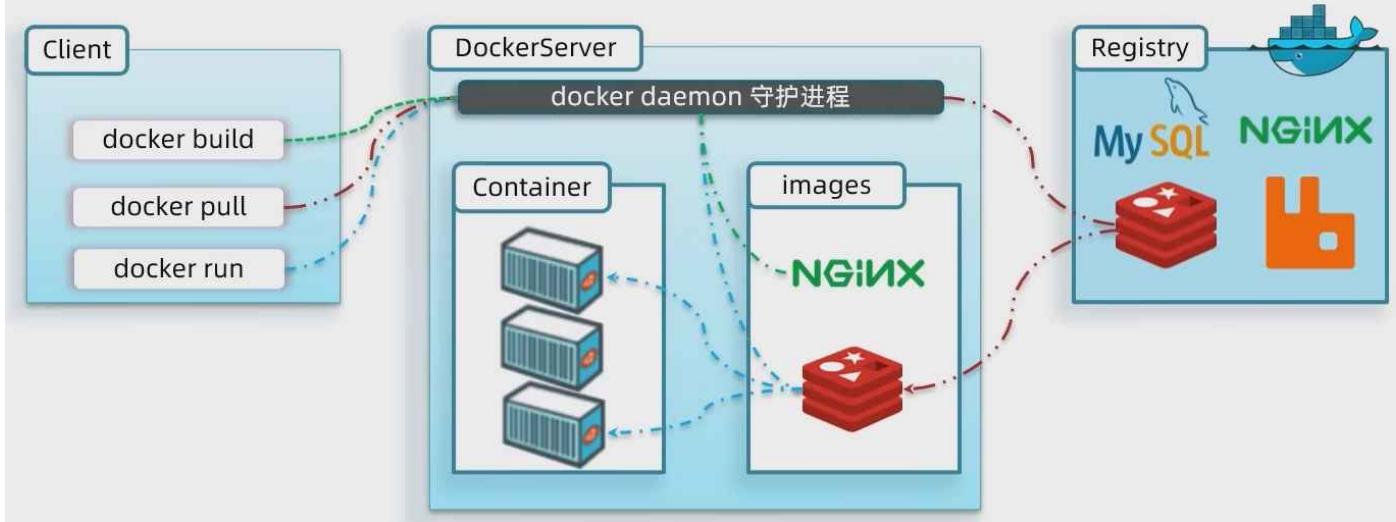
Docker如何解决开发、测试、生产环境有差异的问题

- 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：

- 将应用程序及其依赖、环境、配置打包在一起

容器：

- 镜像运行起来就是容器，一个镜像可以运行多个容器

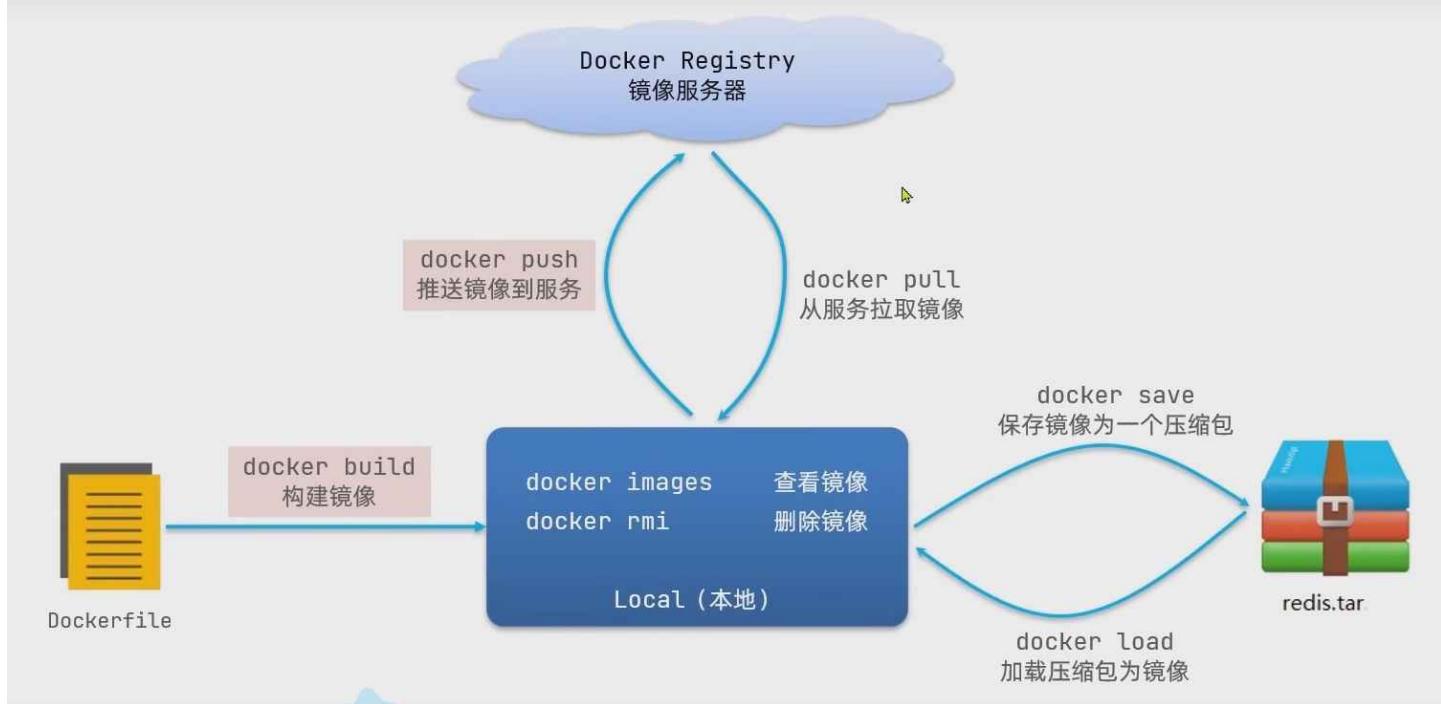
Docker结构：

- 服务端：接收命令或远程请求，操作镜像或容器
- 客户端：发送命令或者请求到Docker服务端

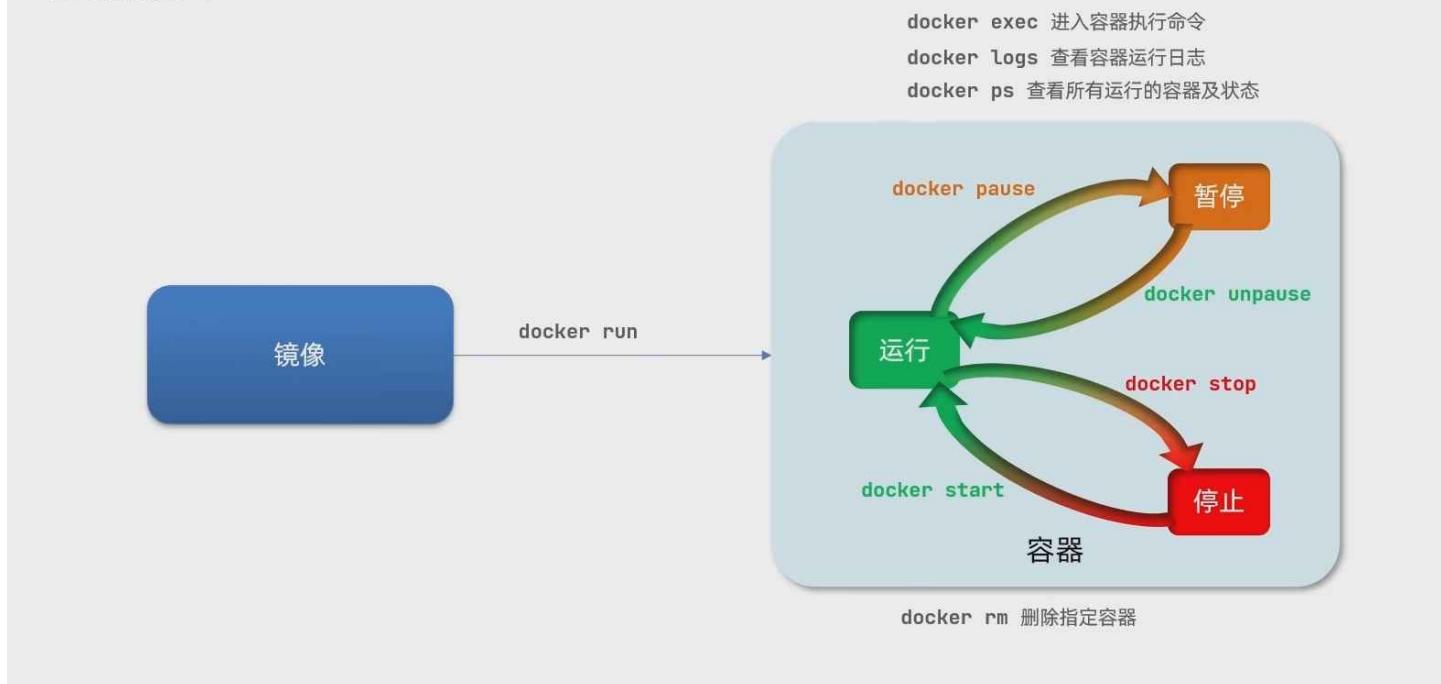
DockerHub：

- 一个镜像托管的服务器，类似的还有阿里云镜像服务，统称为 DockerRegistry

镜像操作命令



容器相关命令



docker run命令的常见参数有哪些？

- --name: 指定容器名称
- -p: 指定端口映射
- -d: 让容器后台运行

查看容器日志的命令：

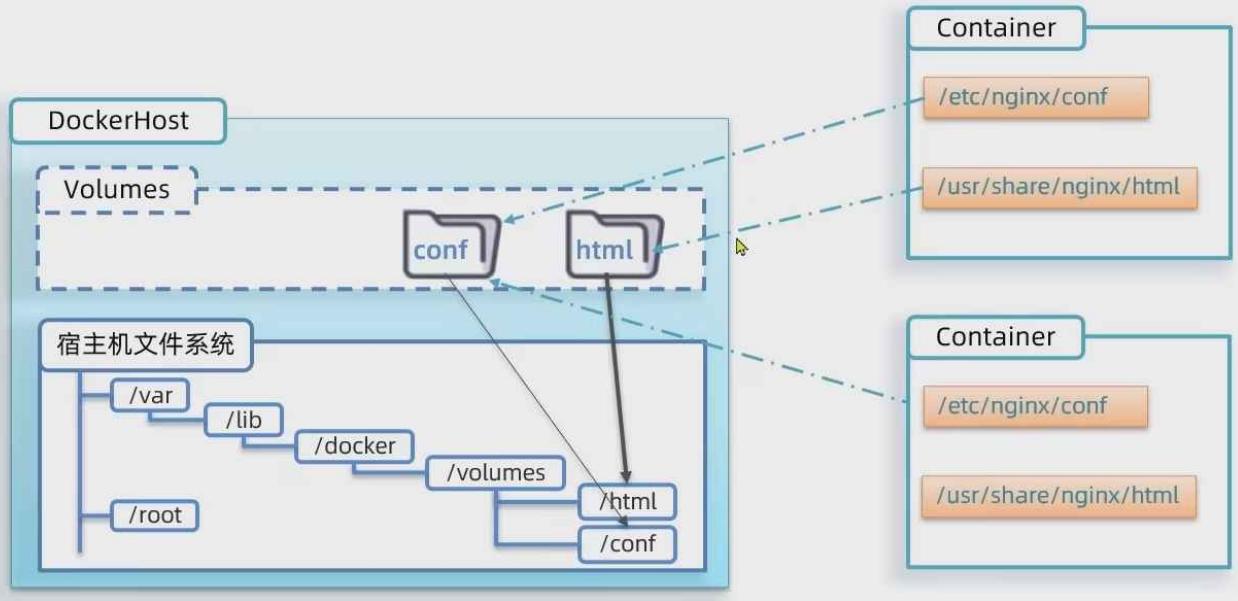
- docker logs
- 添加 -f 参数可以持续查看日志

查看容器状态：

- docker ps



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。



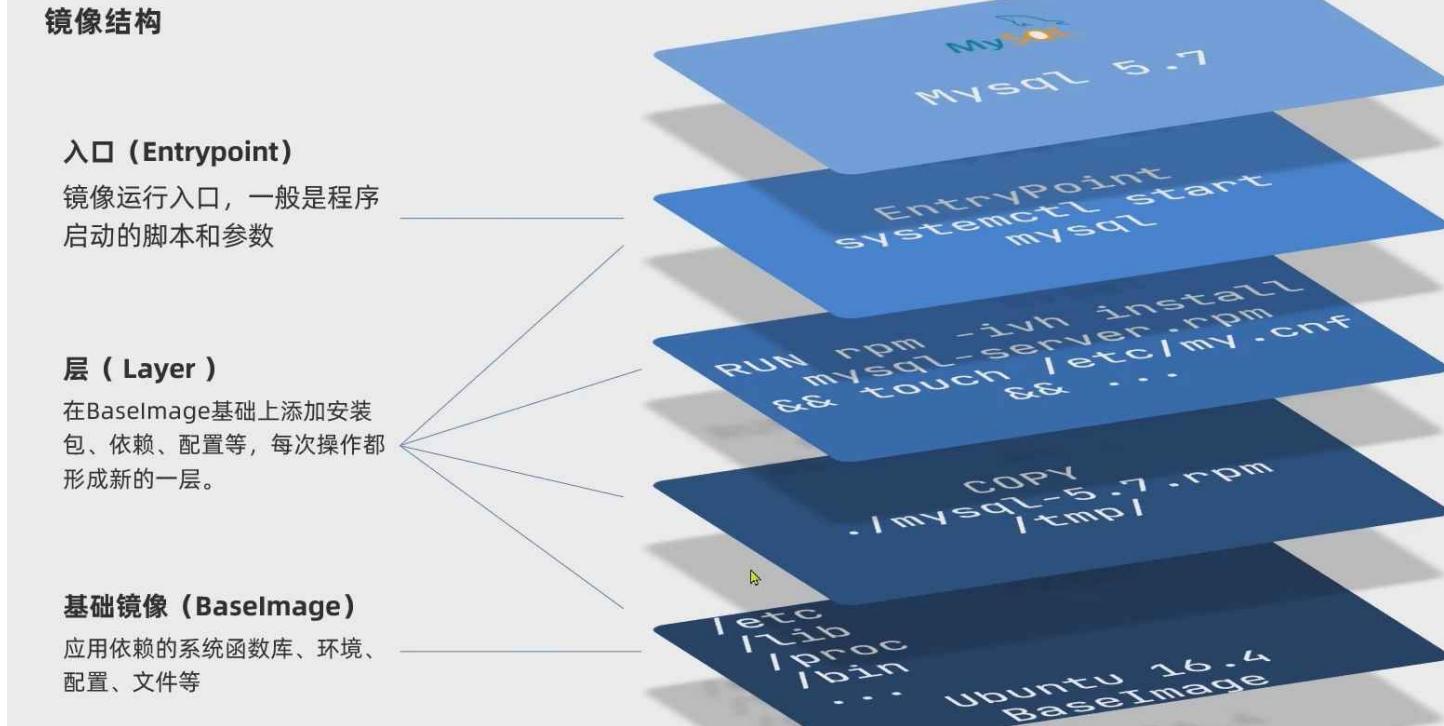
数据卷的作用：

- 将容器与数据分离，解耦合，方便操作容器内数据，保证数据安全

数据卷操作：

- docker volume create
- docker volume ls
- docker volume inspect
- docker volume rm
- docker volume prune

镜像结构



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。

1. 推送本地镜像到仓库前都必须重命名(docker tag)镜像，以镜像仓库地址为前缀
2. 镜像仓库推送前需要把仓库地址配置到docker服务的daemon.json文件中，被docker信任¹
3. 推送使用docker push命令
4. 拉取使用docker pull命令

同步调用的优点：

- 时效性较强，可以立即得到结果

同步调用的问题：

- 耦合度高
- 性能和吞吐能力下降
- 有额外的资源消耗
- 有级联失败问题

异步通信的优点：

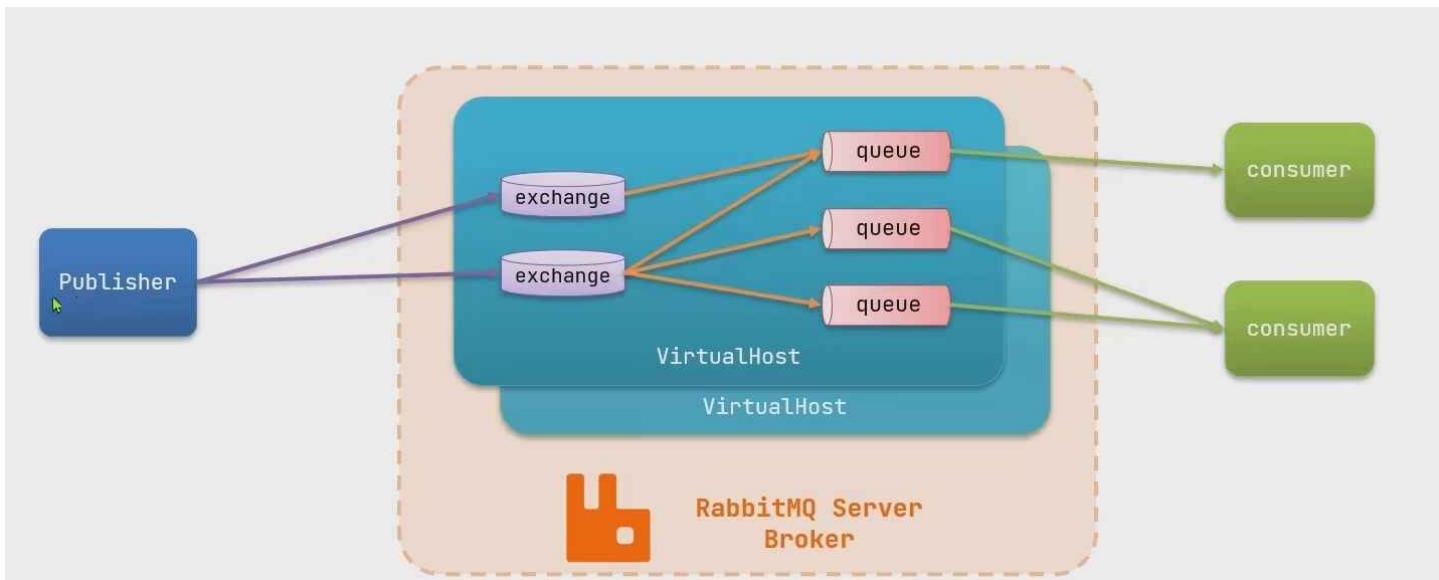
- 耦合度低
- 吞吐量提升
- 故障隔离
- 流量削峰

异步通信的缺点：

- 依赖于Broker的可靠性、安全性、吞吐能力
- 架构复杂了，业务没有明显的流程线，不好追踪管理

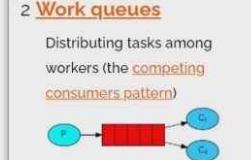
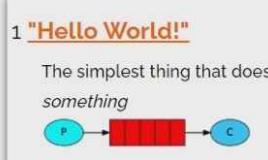
MQ (MessageQueue) , 中文是消息队列，字面来看就是存放消息的队列。也就是事件驱动架构中的Broker。

	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



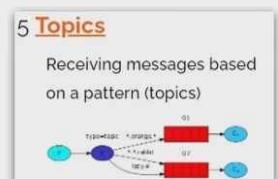
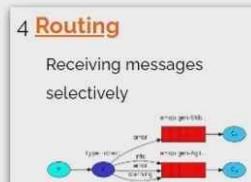
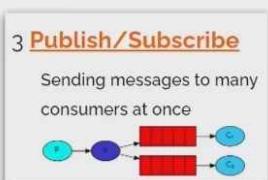
MQ的官方文档中给出了5个MQ的Demo示例，对应了几种不同的用法：

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



- 发布订阅 (Publish、Subscribe) , 又根据交换机类型不同分为三种：

- Fanout Exchange: 广播
- Direct Exchange: 路由
- Topic Exchange: 主题



什么是AMQP?

- 应用间消息通信的一种协议，与语言和平台无关。

SpringAMQP如何发送消息?

- 引入amqp的starter依赖
- 配置RabbitMQ地址
- 利用RabbitTemplate的convertAndSend方法

步骤3：在consumer中编写消费逻辑，监听simple.queue

1. 在consumer服务中编写application.yml，添加mq连接信息：

```
spring:  
  rabbitmq:  
    host: 192.168.150.101 # 主机名  
    port: 5672 # 端口  
    virtual-host: / # 虚拟主机  
    username: itcast # 用户名  
    password: 123321 # 密码
```

2. 在consumer服务中新建一个类，编写消费逻辑：

```
@Component  
public class SpringRabbitListener {  
  
    @RabbitListener(queues = "simple.queue")  
    public void listenSimpleQueueMessage(String msg) throws InterruptedException {  
        System.out.println("spring 消费者接收到消息 : [" + msg + "]");  
    }  
}
```

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

步骤1：在consumer服务声明Exchange、Queue、Binding

在consumer服务常见一个类，添加@Configuration注解，并声明FanoutExchange、Queue和绑定关系对象Binding，代码如下：

```
@Configuration
public class FanoutConfig {
    // 声明FanoutExchange交换机
    @Bean
    public FanoutExchange fanoutExchange(){
        return new FanoutExchange("itcast.fanout");
    }
    // 声明第1个队列
    @Bean
    public Queue fanoutQueue1(){
        return new Queue("fanout.queue1");
    }
    // 绑定队列1和交换机
    @Bean
    public Binding bindingQueue1(Queue fanoutQueue1, FanoutExchange fanoutExchange){
        return BindingBuilder.bind(fanoutQueue1).to(fanoutExchange);
    }
    // ... 略，以相同方式声明第2个队列，并完成绑定
}
```

交换机的作用是什么？

- 接收publisher发送的消息
- 将消息按照规则路由到与之绑定的队列
- 不能缓存消息，路由失败，消息丢失
- FanoutExchange的会将消息路由到每个绑定的队列

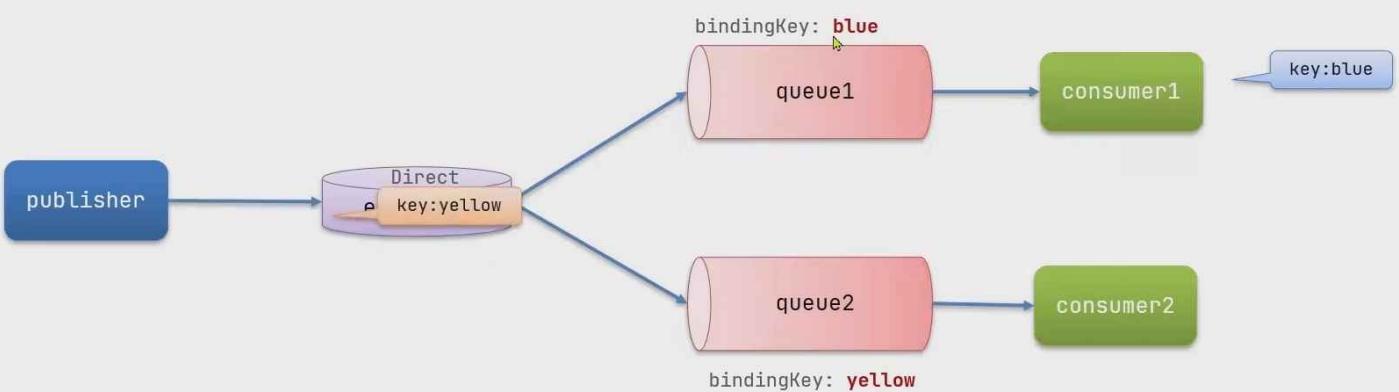
声明队列、交换机、绑定关系的Bean是什么？

- Queue
- FanoutExchange
- Binding

发布订阅-DirectExchange

Direct Exchange 会将接收到的消息根据规则路由到指定的Queue，因此称为路由模式（routes）。

- 每一个Queue都与Exchange设置一个BindingKey
- 发布者发送消息时，指定消息的RoutingKey
- Exchange将消息路由到BindingKey与消息RoutingKey一致的队列



- 在consumer服务中，编写两个消费者方法，分别监听direct.queue1和direct.queue2，
- 并利用@RabbitListener声明Exchange、Queue、RoutingKey

```

@RabbitListener(bindings = @QueueBinding(
    value = @Queue(name = "direct.queue1"),
    exchange = @Exchange(name = "itcast.direct", type = ExchangeTypes.DIRECT),
    key = {"red", "blue"})
)
public void listenDirectQueue1(String msg){
    System.out.println("消费者1接收到Direct消息: ["+msg+"] ");
}

@RabbitListener(bindings = @QueueBinding(
    value = @Queue(name = "direct.queue2"),
    exchange = @Exchange(name = "itcast.direct", type = ExchangeTypes.DIRECT),
    key = {"red", "yellow"})
)
public void listenDirectQueue2(String msg){
    System.out.println("消费者2接收到Direct消息: ["+msg+"] ");
}

```

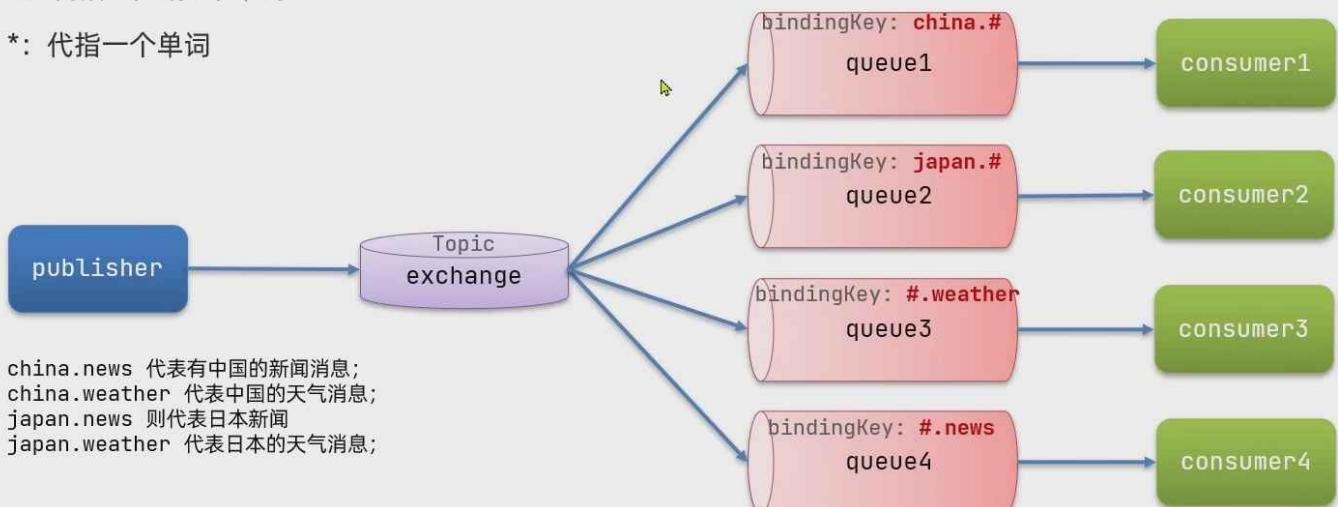
发布订阅-TopicExchange

TopicExchange与DirectExchange类似，区别在于routingKey必须是多个单词的列表，并且以.分割。

Queue与Exchange指定BindingKey时可以使用通配符：

#：代指0个或多个单词

*：代指一个单词



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();
}
```

SpringAMQP中消息的序列化和反序列化是怎么实现的？

- 利用MessageConverter实现的，默认是JDK的序列化
- 注意发送方与接收方必须使用相同的MessageConverter

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

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

正向索引

词条 (term)	文档id
小米	1 ,3 ,4
手机	1 ,2
华为	2 ,3
充电器	3
手环	4

倒排索引



什么是文档和词条？

- 每一条数据就是一个文档
- 对文档中的内容分词，得到的词语就是词条

什么是正向索引？

- 基于文档id创建索引。查询词条时必须先找到文档，而后判断是否包含词条

什么是倒排索引？

- 对文档内容分词，对词条创建索引，并记录词条所在文档的信息。查询时先根据词条查询到文档id，而后获取到文档

概念对比

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

分词器的作用是什么？

- 创建倒排索引时对文档分词
- 用户搜索时，对输入的内容分词

IK分词器有几种模式？

- ik_smart: 智能切分，粗粒度
- ik_max_word: 最细切分，细粒度

IK分词器如何拓展词条？如何停用词条？

- 利用config目录的IkAnalyzer.cfg.xml文件添加拓展词典和停用词典
- 在词典中添加拓展词条或者停用词条

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"  
    }  
}
```

索引库操作有哪些？

- 创建索引库：PUT /索引库名
- 查询索引库：GET /索引库名
- 删除索引库：DELETE /索引库名
- 添加字段：PUT /索引库名/_mapping

文档操作有哪些？

- 创建文档：POST /索引库名/_doc/文档id { json文档 }
- 查询文档：GET /索引库名/_doc/文档id
- 删除文档：DELETE /索引库名/_doc/文档id
- 修改文档：
 - 全量修改：PUT /索引库名/_doc/文档id { json文档 }
 - 增量修改：POST /索引库名/_update/文档id { "doc": {字段} }

步骤4：创建索引库

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);  
}
```

返回的对象中包含
索引库操作的所有方法

PUT /hotel 请求路径, 索引库名称

```
{  
    "mappings": {  
        "properties": {  
            "id": {  
                "type": "keyword"  
            },  
            "name": {  
                "type": "text",  
                "analyzer": "ik_max_word"  
            },  
            "address": {  
                "type": "geo_point"  
            },  
            "price": {  
                "type": "float"  
            },  
            "score": {  
                "type": "float"  
            },  
            "brand": {  
                "type": "string"  
            },  
            "city": {  
                "type": "string"  
            },  
            "starName": {  
                "type": "string"  
            },  
            "business": {  
                "type": "string"  
            },  
            "location": {  
                "type": "geo_point"  
            },  
            "pic": {  
                "type": "keyword",  
                "index": false  
            }  
        }  
    }  
}
```

DSL

索引库操作的基本步骤：

- 初始化RestHighLevelClient
- 创建XxxIndexRequest。XXX是CREATE、Get、Delete
- 准备DSL (CREATE时需要)
- 发送请求。调用RestHighLevelClient#indices().xxx()方法
，xxx是create、exists、delete

文档操作的基本步骤：

- 初始化RestHighLevelClient
- 创建XxxRequest。XXX是Index、Get、Update、Delete
- 准备参数（Index和Update时需要）
- 发送请求。调用RestHighLevelClient#.xxx()方法，xxx是index、get、update、delete

es搜索

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

match和multi_match的区别是什么？

- match：根据一个字段查询
- multi_match：根据多个字段查询，参与查询字段越多，查询性能越差



精确查询常见的有哪些？

- term查询：根据词条精确匹配，一般搜索keyword类型、数值类型、布尔类型、日期类型字段
- range查询：根据数值范围查询，可以是数值、日期的范围

elasticsearch中的相关性打分算法是什么？

- TF-IDF：在elasticsearch5.0之前，会随着词频增加而越来越大
- BM25：在elasticsearch5.0之后，会随着词频增加而增大，但增长曲线会趋于水平

function score query定义的三要素是什么？

- 过滤条件：哪些文档要加分
- 算分函数：如何计算function score
- 加权方式：function score 与 query score如何运算

bool查询有几种逻辑关系？

- must: 必须匹配的条件，可以理解为“与”
- should: 选择性匹配的条件，可以理解为“或”
- must_not: 必须不匹配的条件，不参与打分
- filter: 必须匹配的条件，不参与打分

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。

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
      }
    }
  }
}
```

查询的基本步骤是：

1. 创建SearchRequest对象
2. 准备Request.source(), 也就是DSL。
 - ① QueryBuilders来构建查询条件
 - ② 传入Request.source() 的 query() 方法
3. 发送请求，得到结果
4. 解析结果（参考JSON结果，从外到内，逐层解析）

全文检索的match和multi_match查询与match_all的API基本一致。差别是查询条件，也就是query的部分。

同样是利用QueryBuilders提供的方法：

```
// 单字段查询
QueryBuilders.matchQuery("all", "如家");

// 多字段查询
QueryBuilders.multiMatchQuery("如家", "name", "business");
```

```
GET /hotel/_search
{
  "query": {
    "match_all": {}
  }
}

GET /hotel/_search
{
  "query": {
    "match": {
      "all": "如家"
    }
  }
}

GET /hotel/_search
{
  "query": {
    "multi_match": {
      "query": "如家",
      "fields": ["brand", "name"]
    }
  }
}
```

- 所有搜索DSL的构建，记住一个API：
SearchRequest的source()方法。
- 高亮结果解析是参考JSON结果，逐层解析

es其他

什么是聚合？

- 聚合是对文档数据的统计、分析、计算

聚合的常见种类有哪些？

- Bucket：对文档数据分组，并统计每组数量
- Metric：对文档数据做计算，例如avg
- Pipeline：基于其它聚合结果再做聚合

参与聚合的字段类型必须是：

- keyword
- 数值
- 日期
- 布尔

aggs代表聚合，与query同级，此时query的作用是？

- 限定聚合的的文档范围

聚合必须的三要素：

- 聚合名称
- 聚合类型
- 聚合字段

聚合可配置属性有：

- size: 指定聚合结果数量
- order: 指定聚合结果排序方式
- field: 指定聚合字段

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



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

自动补全对字段的要求：

- 类型是completion类型
- 字段值是多词条的数组

RestAPI实现自动补全

先看请求参数构造的API:

```
// 1. 准备请求
SearchRequest request = new SearchRequest("hotel");
// 2. 请求参数
request.source()
    .suggest(new SuggestBuilder().addSuggestion(
        "mySuggestion",
        SuggestBuilders
            .completionSuggestion("title")
            .prefix("h")
            .skipDuplicates(true)
            .size(10)
    ));
// 3. 发送请求
client.search(request, RequestOptions.DEFAULT);
```

```
// 自动补全查询
GET /test/_search
{
    "suggest": {
        "mySuggestion": {
            "text": "h", // 关键字
            "completion": {
                "field": "title", // 补全字段
                "skip_duplicates": true,
                "size": 10 // 获取前10条结果
            }
        }
    }
}
```

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

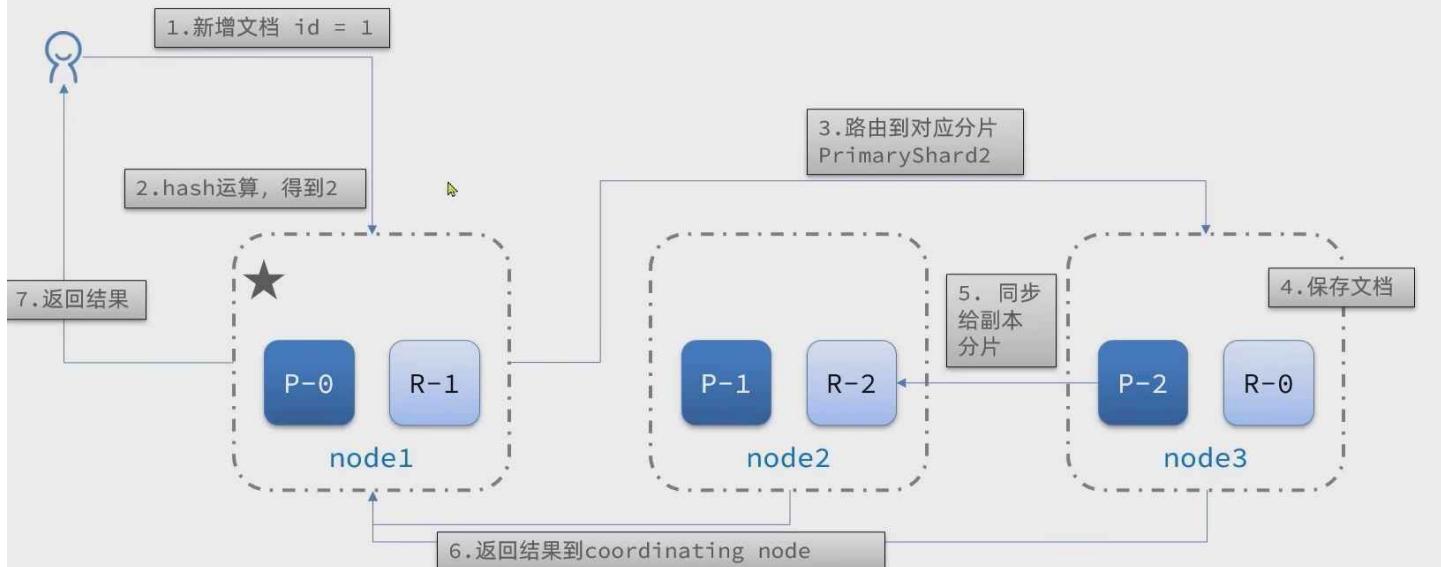
方式三：监听binlog

- 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

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

新增文档流程：



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



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。

微服务保护

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。



什么是雪崩问题？

- 微服务之间相互调用，因为调用链中的一个服务故障，引起整个链路都无法访问的情况。

如何避免因瞬间高并发流量而导致服务故障？

- 流量控制

如何避免因服务故障引起的雪崩问题？

- 超时处理
- 线程隔离
- 降级熔断

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

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

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

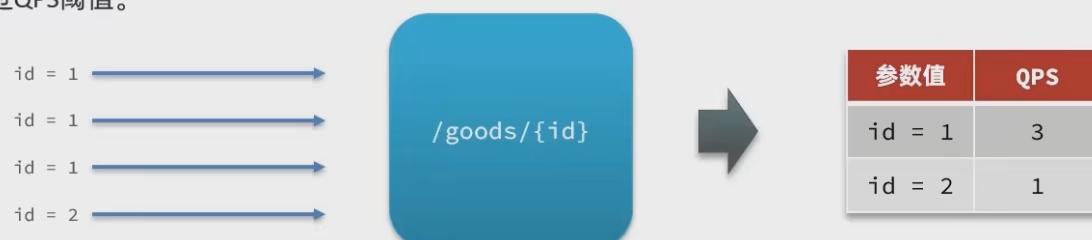
当请求超过QPS阈值时，快速失败和warm up会拒绝新的请求并抛出异常。而排队等待则是让所有请求进入一个队列中，然后按照阈值允许的时间间隔依次执行。后来的请求必须等待前面执行完成，如果请求预期的等待时间超出最大时长，则会被拒绝。

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



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。



Sentinel支持的雪崩解决方案：

- 线程隔离（仓壁模式）
- 降级熔断

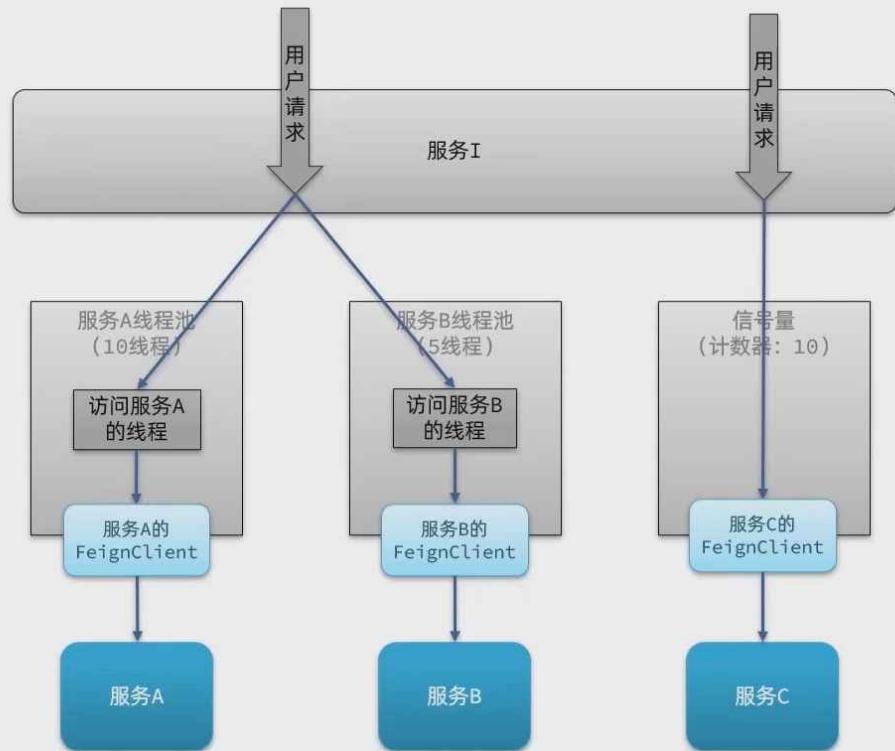
Feign整合Sentinel的步骤：

- 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）



线程隔离的两种手段是？

- 信号量隔离



- 线程池隔离

信号量隔离的特点是？

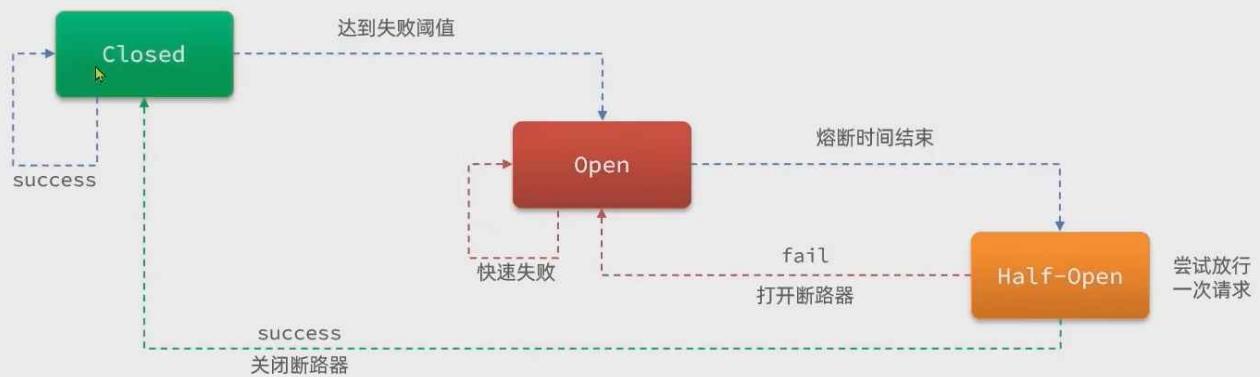
- 基于计数器模式，简单，开销小

线程池隔离的特点是？

- 基于线程池模式，有额外开销，但隔离控制更强

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。



Sentinel熔断降级的策略有哪些？

- 慢调用比例：超过指定时长的调用为慢调用，统计单位时长内慢调用的比例，超过阈值则熔断
- 异常比例：统计单位时长内异常调用的比例，超过阈值则熔断
- 异萂数：统计单位时长内异常调用的次数，超过阈值则熔断

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

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;  
}
```

Sentinel的三种配置管理模式是什么？

- 原始模式：保存在内存
- pull模式：保存在本地文件或数据库，定时去读取
- push模式：保存在nacos，监听变更实时更新

分布式事务

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：

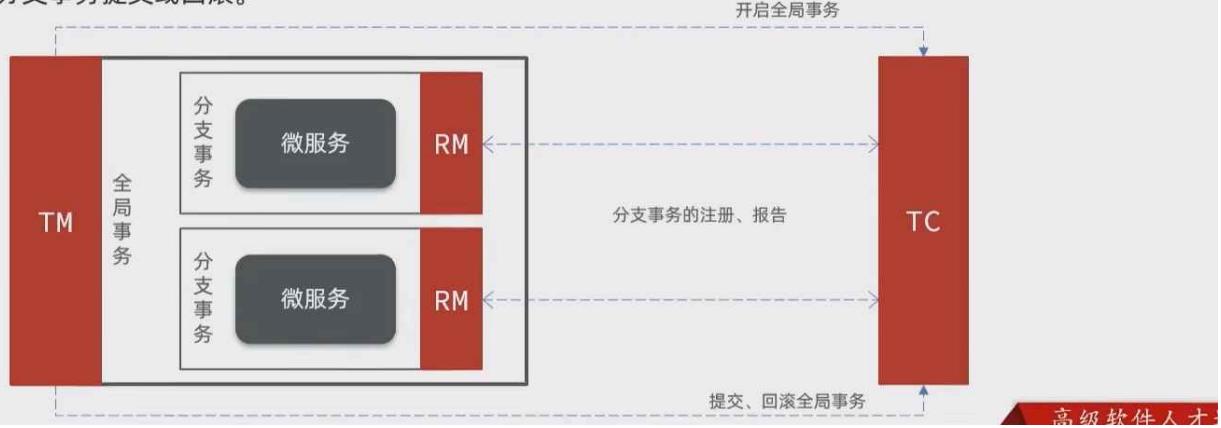
- AP模式：各子事务分别执行和提交，允许出现结果不一致，然后采用弥补措施恢复数据即可，实现**最终一致**。
- CP模式：各个子事务执行后互相等待，同时提交，同时回滚，达成**强一致**。但事务等待过程中，处于弱可用状态。

您好，这是Bing。我可以帮您了解一下什么是CAP理论。根据我的搜索结果，CAP理论是一系统的理论，它指出对于一个分布式系统来说，不可能同时满足以下三个性质：

- **一致性 (Consistency)**：所有节点访问同一份最新的数据副本。
- **可用性 (Availability)**：每次请求都能获取到非错误的响应，不保证数据为最新。
- **分区容忍性 (Partition tolerance)**：系统在网络分区或故障的情况下仍能继续运行。

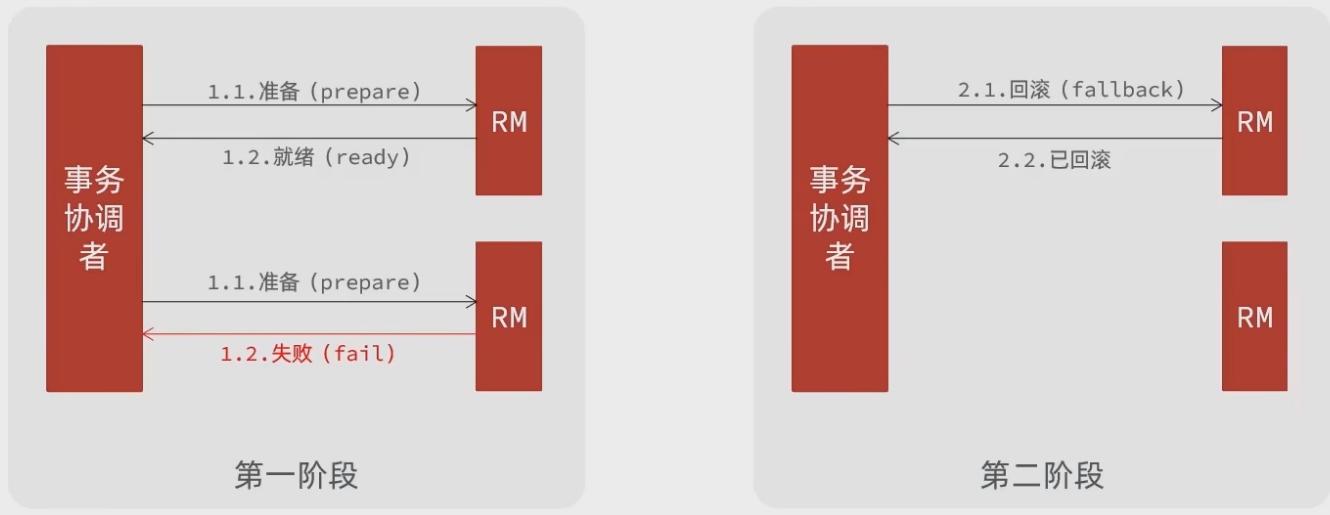
Seata事务管理中有三个重要的角色：

- **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。

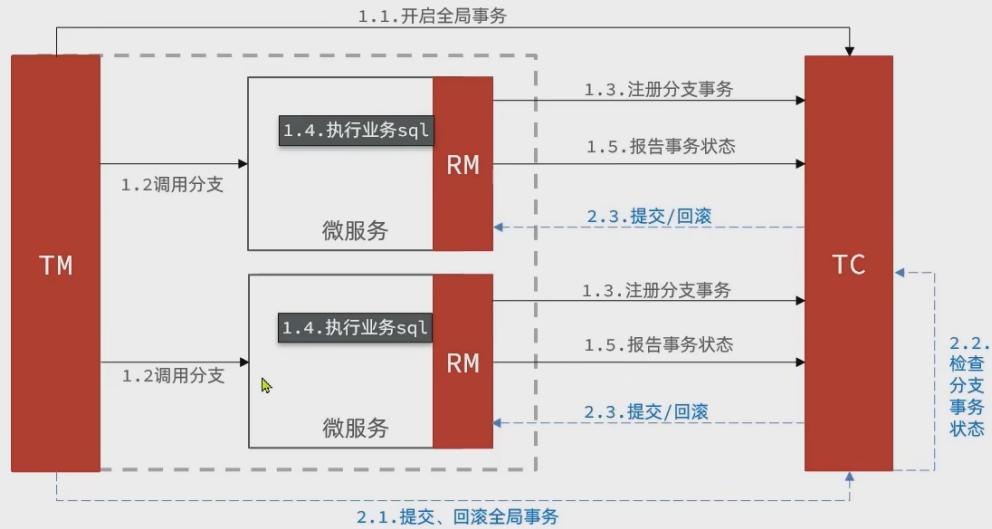


XA模式原理

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 规范描述了全局的 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模式做了一些调整，但大体相似：



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：

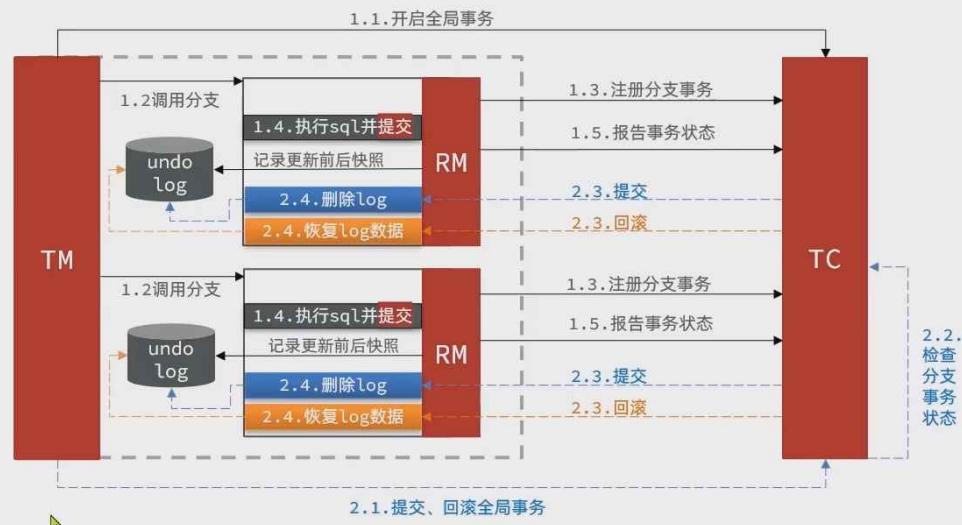
- 注册分支事务
- 记录undo-log（数据快照）
- 执行业务sql并提交
- 报告事务状态

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



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模式的写隔离



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：



- 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

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

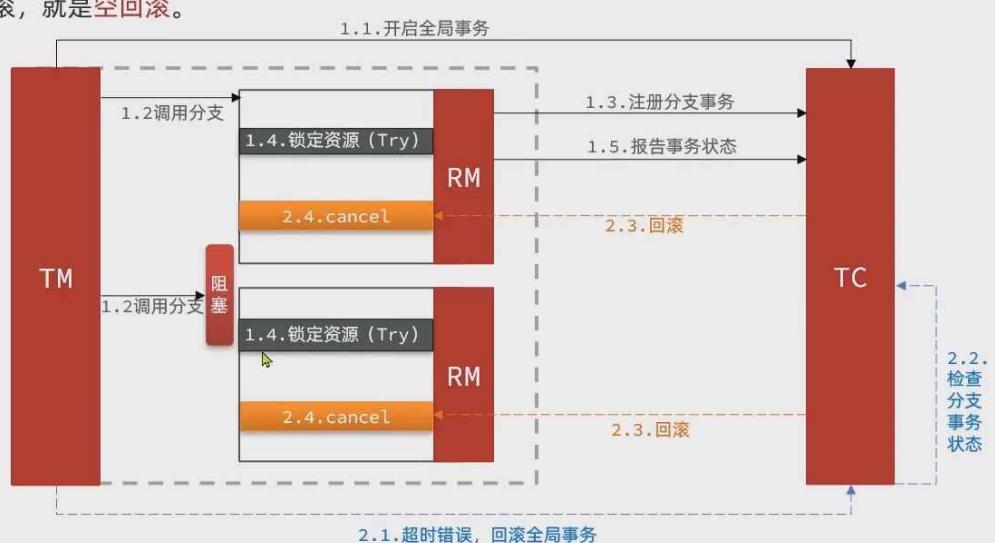
TCC的缺点是什么？

- 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



综上所述，TCC相比于XA如何减少资源的锁定时间，主要是通过以下两个方面：

- TCC不需要等待协调者的指令来提交或回滚事务，而是根据自己的业务逻辑来决定何时释放资源。
- TCC不需要对所有的资源进行锁定，而是只对必要的资源进行预留或锁定。

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：

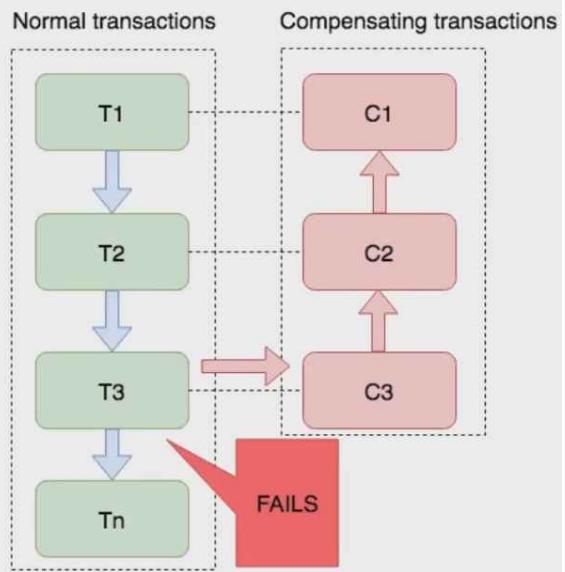
- 一阶段：直接提交本地事务
- 二阶段：成功则什么都不做；失败则通过编写补偿业务来回滚

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

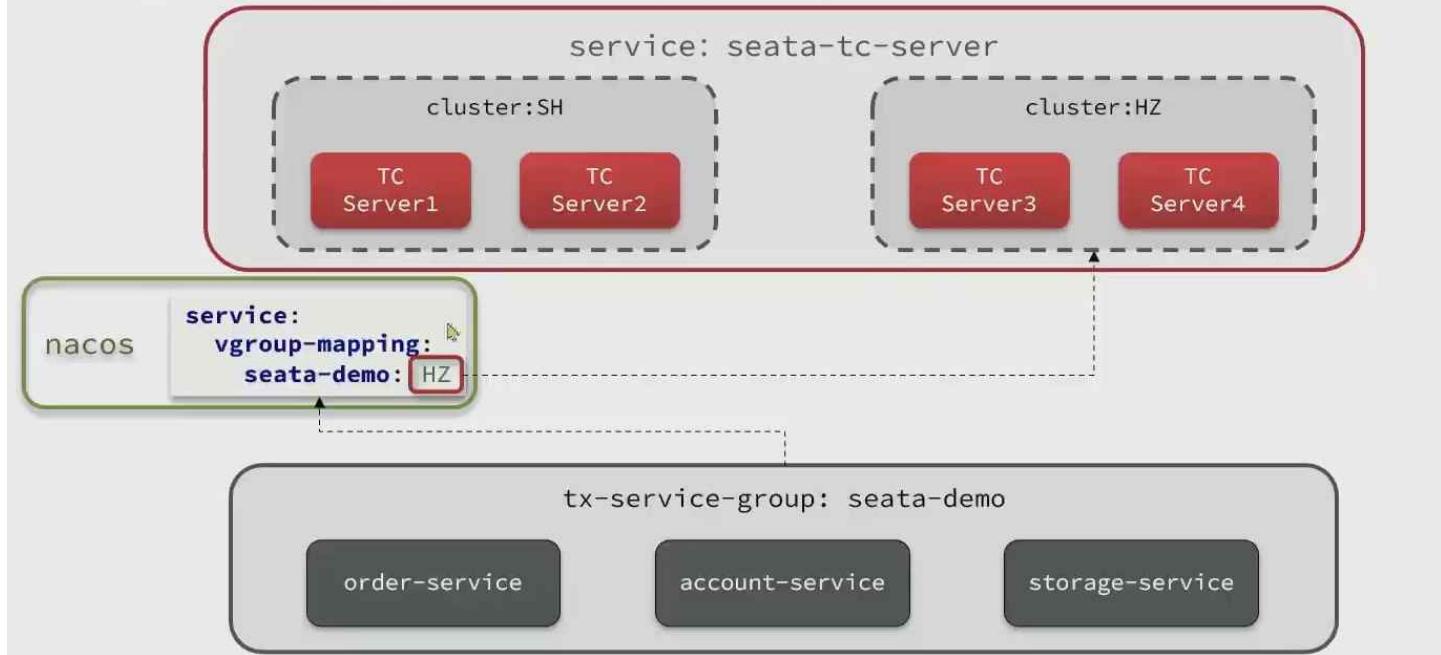


四种模式对比

	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">• 对性能要求较高的事务。• 有非关系型数据库要参与的事务。	<ul style="list-style-type: none">• 业务流程长、业务流程多• 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

TC的异地多机房容灾架构

TC服务作为Seata的核心服务，一定要保证高可用和异地容灾。



Redis

单点Redis的问题

数据丢失问题

实现Redis数据持久化

并发能力问题

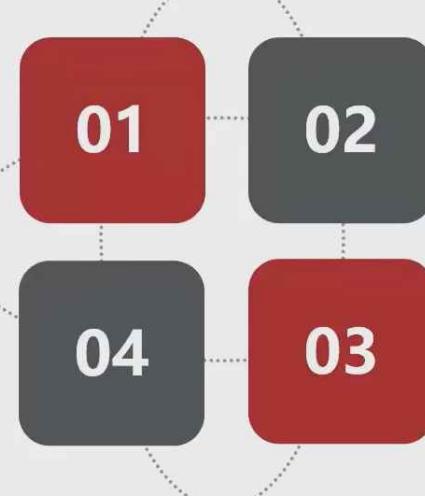
搭建主从集群，实现读写分离

存储能力问题

搭建分片集群，利用插槽机制实现动态扩容

故障恢复问题

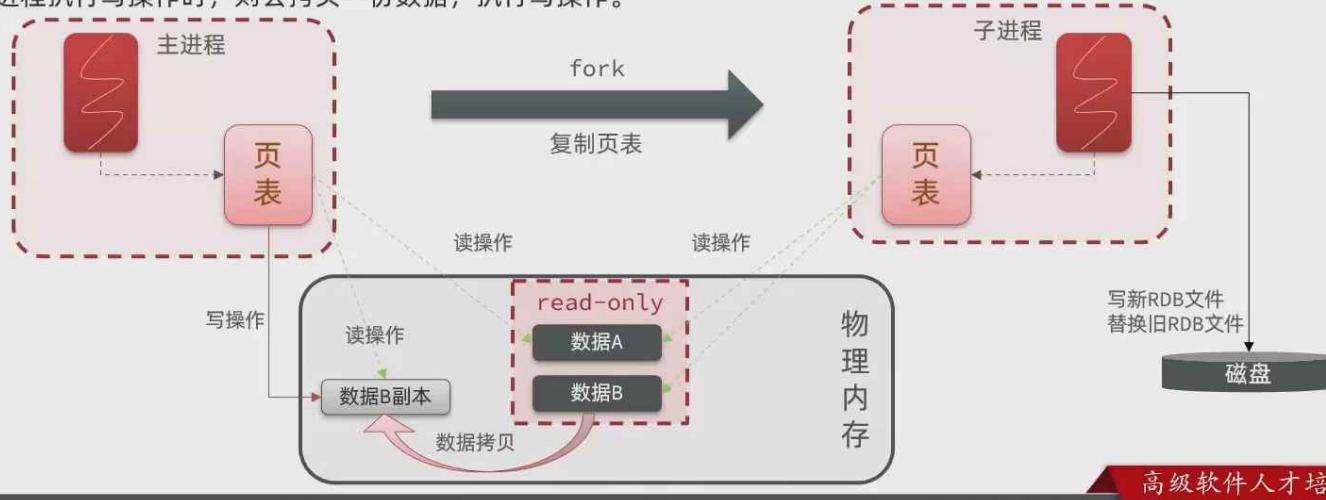
利用Redis哨兵，实现健康检测和自动恢复



RDB

bgsave开始时会fork主进程得到子进程，子进程共享主进程的内存数据。完成fork后读取内存数据并写入 RDB 文件。
fork采用的是copy-on-write技术：

- 当主进程执行读操作时，访问共享内存；
- 当主进程执行写操作时，则会拷贝一份数据，执行写操作。



RDB方式bgsave的基本流程？

- fork主进程得到一个子进程，共享内存空间
- 子进程读取内存数据并写入新的RDB文件
- 用新RDB文件替换旧的RDB文件。

RDB会在什么时候执行？ save 60 1000代表什么含义？

- 默认是服务停止时。
- 代表60秒内至少执行1000次修改则触发RDB

RDB的缺点？

- RDB执行间隔时间长，两次RDB之间写入数据有丢失的风险
- fork子进程、压缩、写出RDB文件都比较耗时

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
```

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

2.4.开启主从关系

现在三个实例还没有任何关系，要配置主从可以使用replicaof或者slaveof（5.0以前）命令。

有临时和永久两种模式：

- 修改配置文件（永久生效）
 - 在redis.conf中添加一行配置： `slaveof <masterip> <masterport>`
- 使用redis-cli客户端连接到redis服务，执行slaveof命令（重启后失效）：

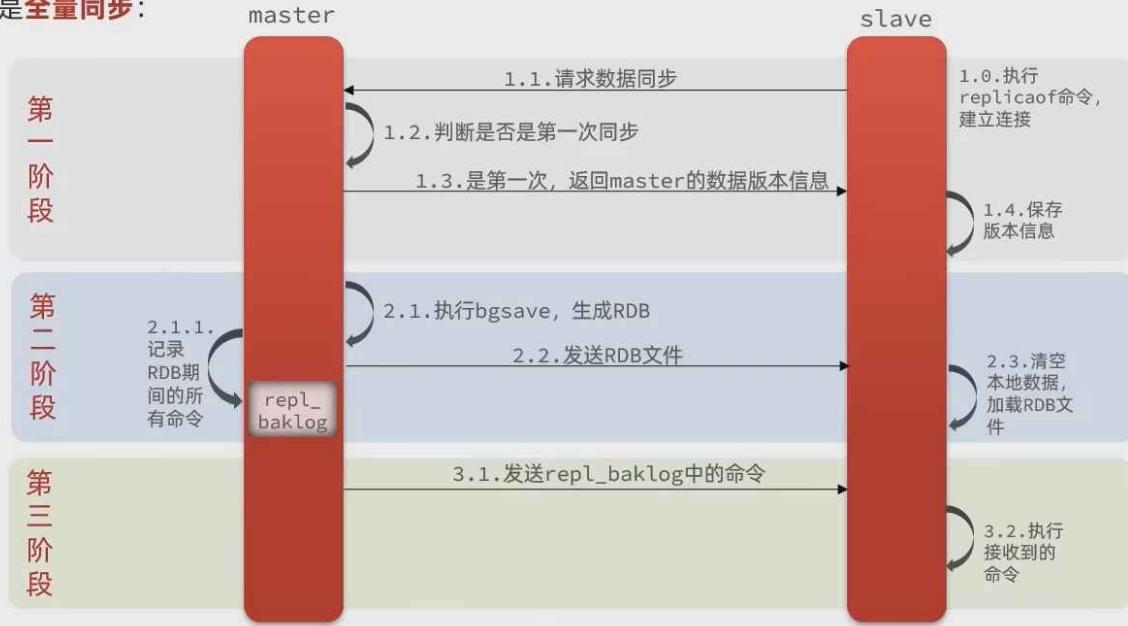


```
1 slaveof <masterip> <masterport>
```

注意：在5.0以后新增命令replicaof，与salveof效果一致。

数据同步原理

主从第一次同步是**全量同步**：



主从第一次同步是全量同步

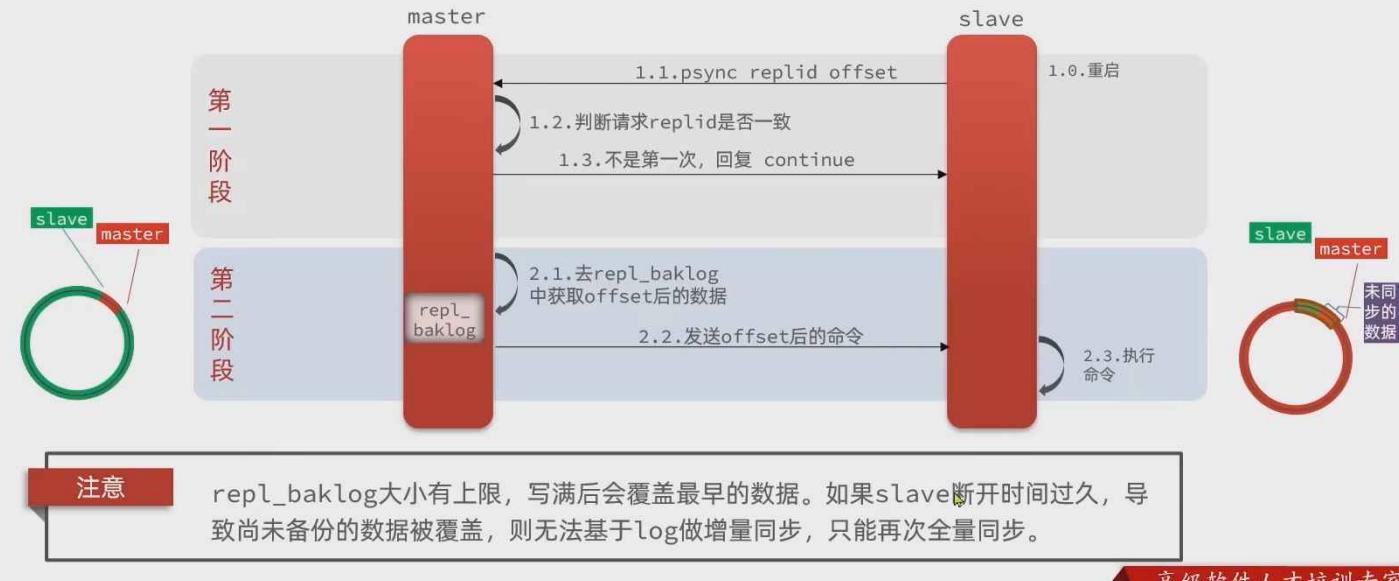


简述全量同步的流程？

- 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
- slave执行接收到的命令，保持与master之间的同步

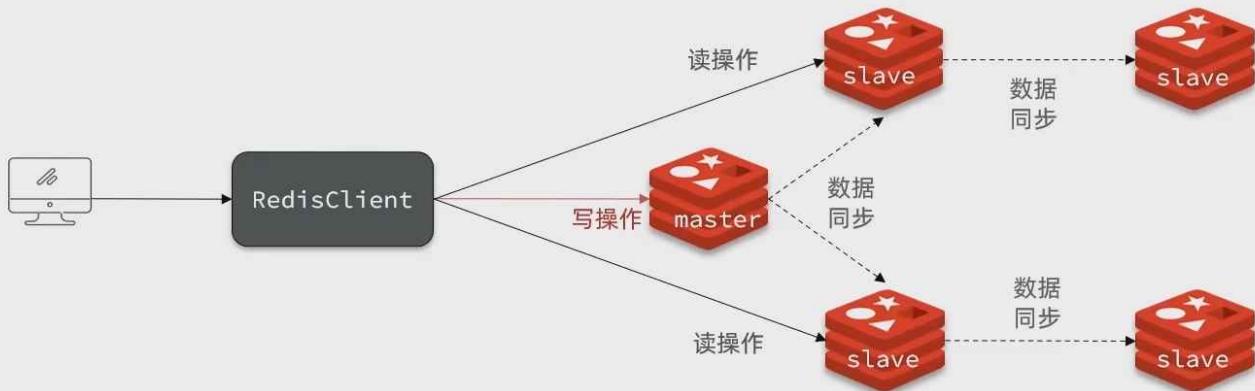
数据同步原理

主从第一次同步是全量同步，但如果slave重启后同步，则执行增量同步



可以从以下几个方面来优化Redis主从就集群：

- 在master中配置repl-diskless-sync yes启用无磁盘复制，避免全量同步时的磁盘IO。
- Redis单节点上的内存占用不要太大，减少RDB导致的过多磁盘IO
- 适当提高repl_baklog的大小，发现slave宕机时尽快实现故障恢复，尽可能避免全量同步
- 限制一个master上的slave节点数量，如果实在是太多slave，则可以采用主-从-从链式结构，减少master压力



简述全量同步和增量同步区别？

- 全量同步：master将完整内存数据生成RDB，发送RDB到slave。后续命令则记录在repl_baklog，逐个发送给slave。
- 增量同步：slave提交自己的offset到master，master获取repl_baklog中从offset之后的命令给slave

什么时候执行全量同步？

- slave节点第一次连接master节点时
- slave节点断开时间太久，repl_baklog中的offset已经被覆盖时

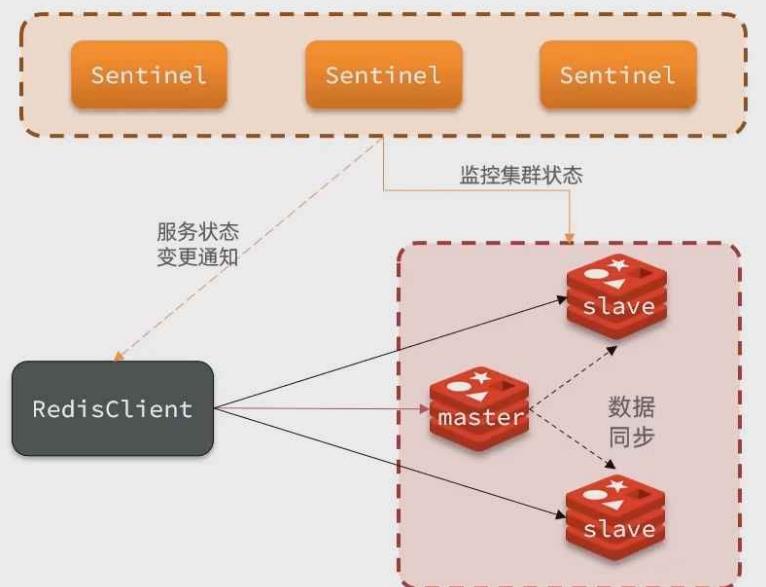
什么时候执行增量同步？

- slave节点断开又恢复，并且在repl_baklog中能找到offset时

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：

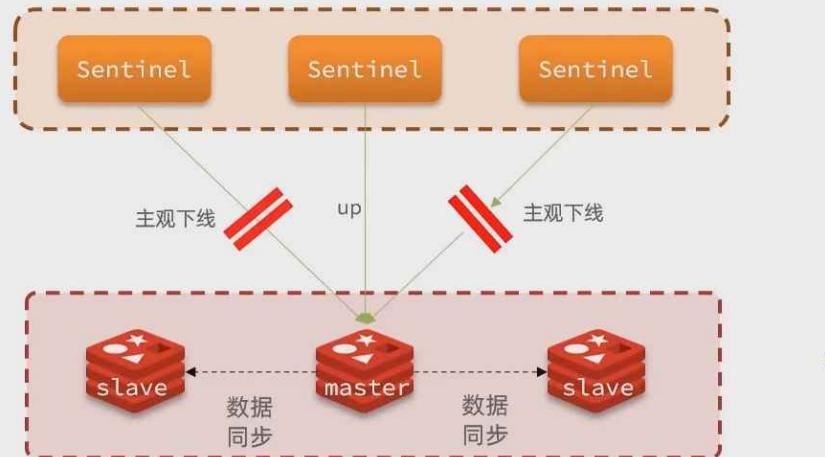
- **监控**：Sentinel会不断检查您的master和slave是否按预期工作
- **自动故障恢复**：如果master故障，Sentinel会将一个slave提升为master。当故障实例恢复后也以新的master为主
- **通知**：Sentinel充当Redis客户端的服务发现来源，当集群发生故障转移时，会将最新信息推送給Redis的客户端



服务状态监控

Sentinel基于心跳机制监测服务状态，每隔1秒向集群的每个实例发送ping命令：

- 主观下线：如果某sentinel节点发现某实例未在规定时间响应，则认为该实例**主观下线**。
- 客观下线：若超过指定数量（quorum）的sentinel都认为该实例主观下线，则该实例**客观下线**。quorum值最好超过Sentinel实例数量的一半。



选举新的master

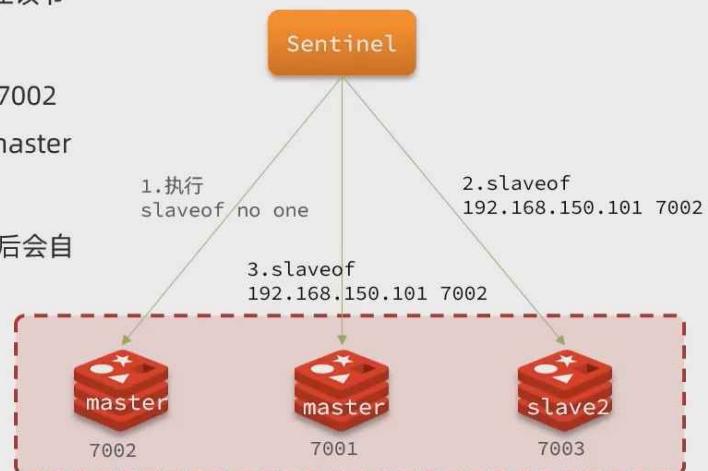
一旦发现master故障，sentinel需要在slave中选择一个作为新的master，选择依据是这样的：

- 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



Sentinel的三个作用是什么？

- 监控
- 故障转移
- 通知

Sentinel如何判断一个redis实例是否健康？

- 每隔1秒发送一次ping命令，如果超过一定时间没有响应则认为是主观下线
- 如果大多数sentinel都认为实例主观下线，则判定服务下线

故障转移步骤有哪些？

- 首先选定一个slave作为新的master，执行slaveof no one
- 然后让所有节点都执行slaveof 新master
- 修改故障节点配置，添加slaveof 新master

在Sentinel集群监管下的Redis主从集群，其节点会因为自动故障转移而发生变化，Redis的客户端必须感知这种变化，及时更新连接信息。Spring的RedisTemplate底层利用lettuce实现了节点的感知和自动切换。

RedisTemplate的哨兵模式

3. 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

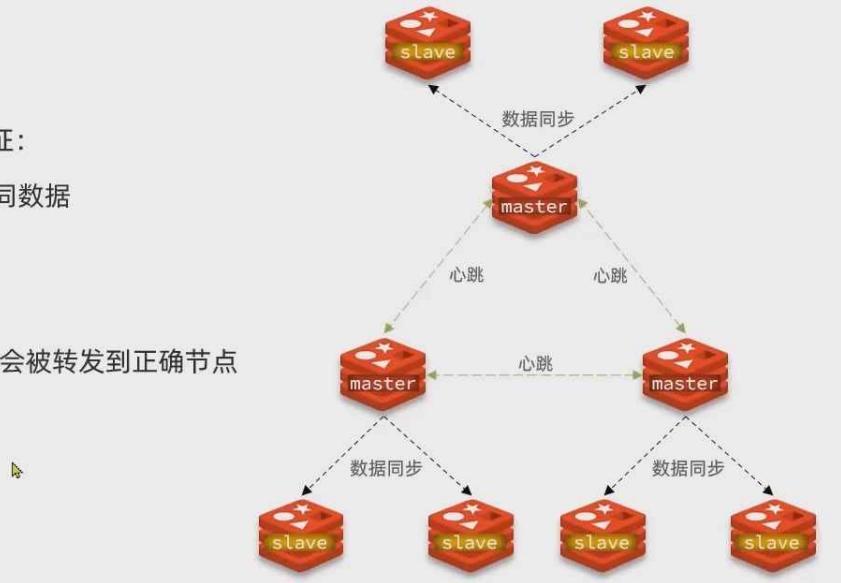
分片集群结构

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{typelid}为前缀

故障转移

当集群中有一个master宕机会发生什么呢？

1. 首先是该实例与其它实例失去连接
2. 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,slave - 0 1625207710000 8 connected 0-5460
afaaa70d6528fc72490e0f3f7b32731a12c12bb8 192.168.150.101:7002@17002 master,fail? - 1625207705198 1625207703000 10 disconnected 5461-10922
6ec60fb5afdf50a465f05c8024bf8f75d809b014 192.168.150.101:8002@18002 slave 1c00e5f9e158b169f199f15884ab43bc433b1a06 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
```

3. 最后是确定下线，自动提升一个slave为新的master：

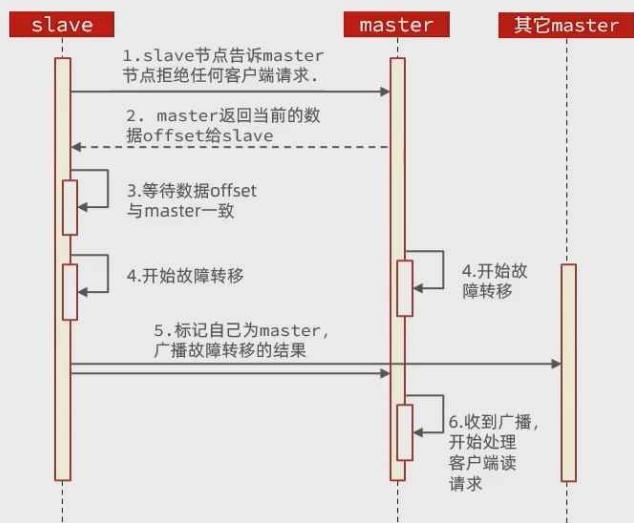
```
1fa6d68d590827c24c237b1c490b78e5c7fe2ca9 192.168.150.101:8003@18003 slave f5fc58defbebb957e47fb0d8327a09dc4f1678f5 0 1625208023157 8
f5fc58defbebb957e47fb0d8327a09dc4f1678f5 192.168.150.101:7001@17001 myself,master - 0 1625208022000 8 connected 0-5460
afaaa70d6528fc72490e0f3f7b32731a12c12bb8 192.168.150.101:7002@17002 master,fail - 1625207705198 1625207703000 10 disconnected
6ec60fb5afdf50a465f05c8024bf8f75d809b014 192.168.150.101:8002@18002 slave 1c00e5f9e158b169f199f15884ab43bc433b1a06 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

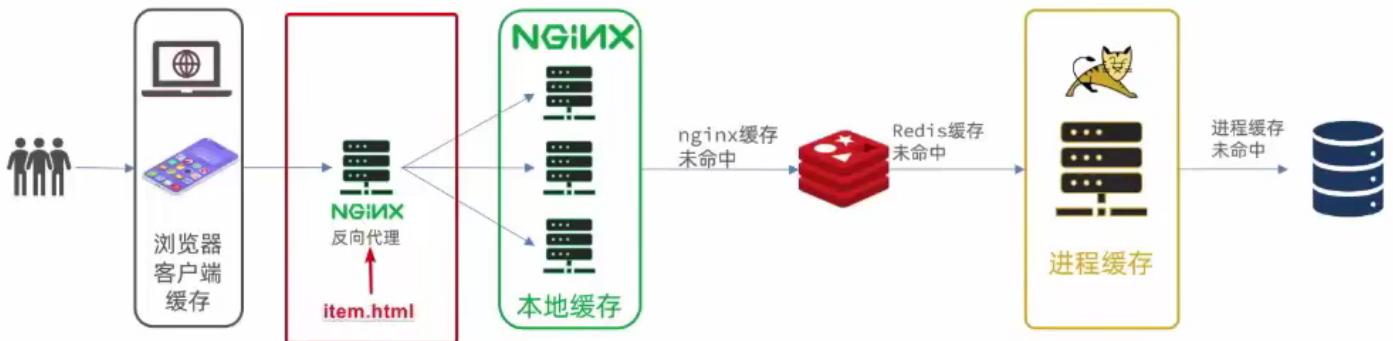
利用cluster failover命令可以手动让集群中的某个master宕机，切换到执行cluster failover命令的这个slave节点，实现无感知的数据迁移。其流程如下：

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



多级缓存



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

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。

OpenResty获取请求参数

OpenResty提供了各种API用来获取不同类型的需求参数:

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

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：

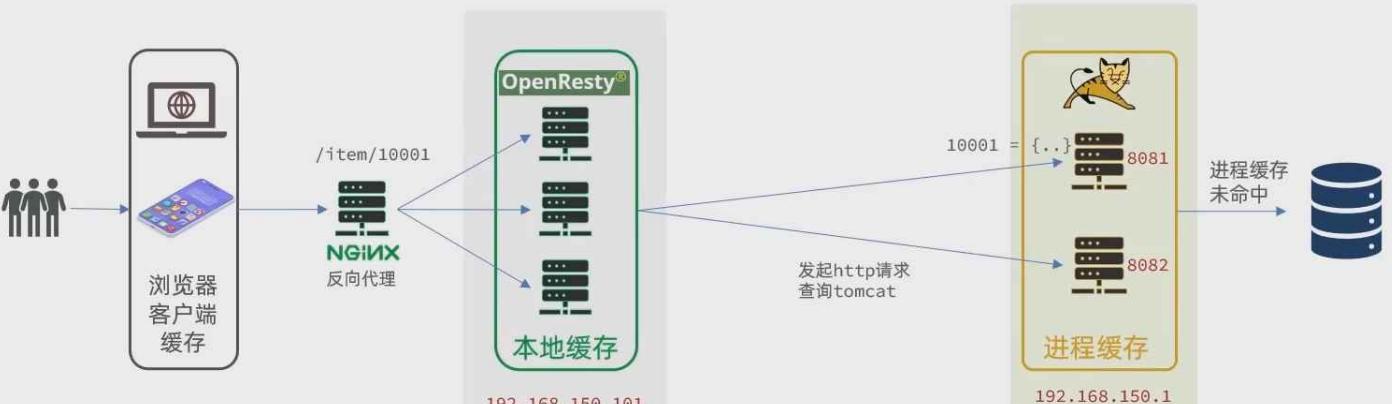
- resp.status: 响应状态码
- resp.header: 响应头，是一个table
- resp.body: 响应体，就是响应数据

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;  
}
```

Tomcat集群的负载均衡



```
# 反向代理配置，将/item路径的请求代理到tomcat集群  
location /item {  
    proxy_pass http://tomcat-cluster;  
}
```

```
# tomcat集群配置  
upstream tomcat-cluster{  
    hash $request_uri;  
    server 192.168.150.1:8081;  
    server 192.168.150.1:8082;  
}
```

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。

我们数据量较少，可以在启动时将所有数据都放入缓存中。

nginx本地缓存

OpenResty为Nginx提供了**shard dict**的功能，可以在nginx的多个worker之间共享数据，实现缓存功能。

- 开启共享字典，在nginx.conf的http下添加配置：

```
# 共享字典，也就是本地缓存，名称叫做：item_cache，大小150m
lua_shared_dict item_cache 150m;
```

- 操作共享字典：

```
-- 获取本地缓存对象
local item_cache = ngx.shared.item_cache
-- 存储，指定key、value、过期时间，单位s，默认为0代表永不过期
item_cache:set('key', 'value', 1000)
-- 读取
local val = item_cache:get('key')
```

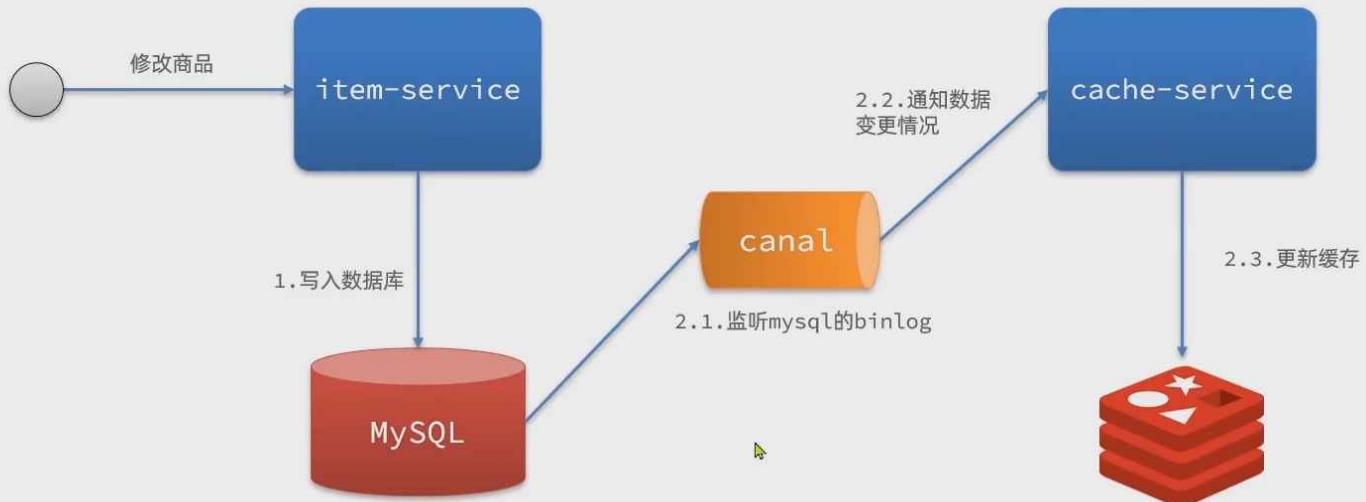
缓存同步策略

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

缓存同步策略

基于Canal的异步通知：

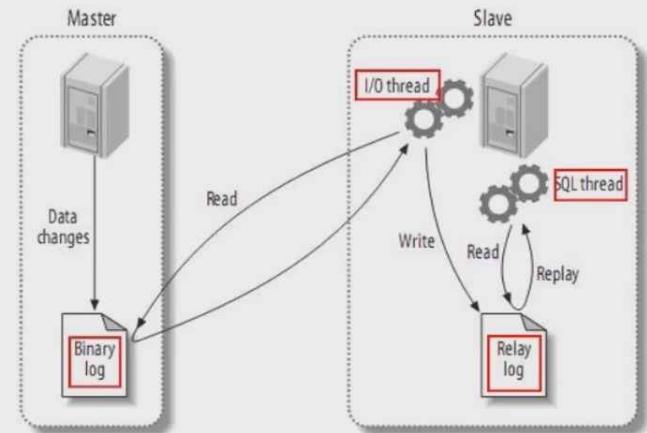


初识Canal

Canal [kə'næl]，译意为水道/管道/沟渠，canal是阿里巴巴旗下的一款开源项目，基于Java开发。基于数据库增量日志解析，提供增量数据订阅&消费。GitHub的地址：<https://github.com/alibaba/canal>

Canal是基于mysql的主从同步来实现的，MySQL主从同步的原理如下：

- MySQL master 将数据变更写入二进制日志(binary log)，其中记录的数据叫做binary log events
- MySQL slave 将 master 的 binary log events拷贝到它的中继日志(relay log)
- MySQL slave 重放 relay log 中事件，将数据变更反映它自己的数据



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。



编写监听器，监听Canal消息：

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

@CanalTable("tb_item")
@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
    }
}
```

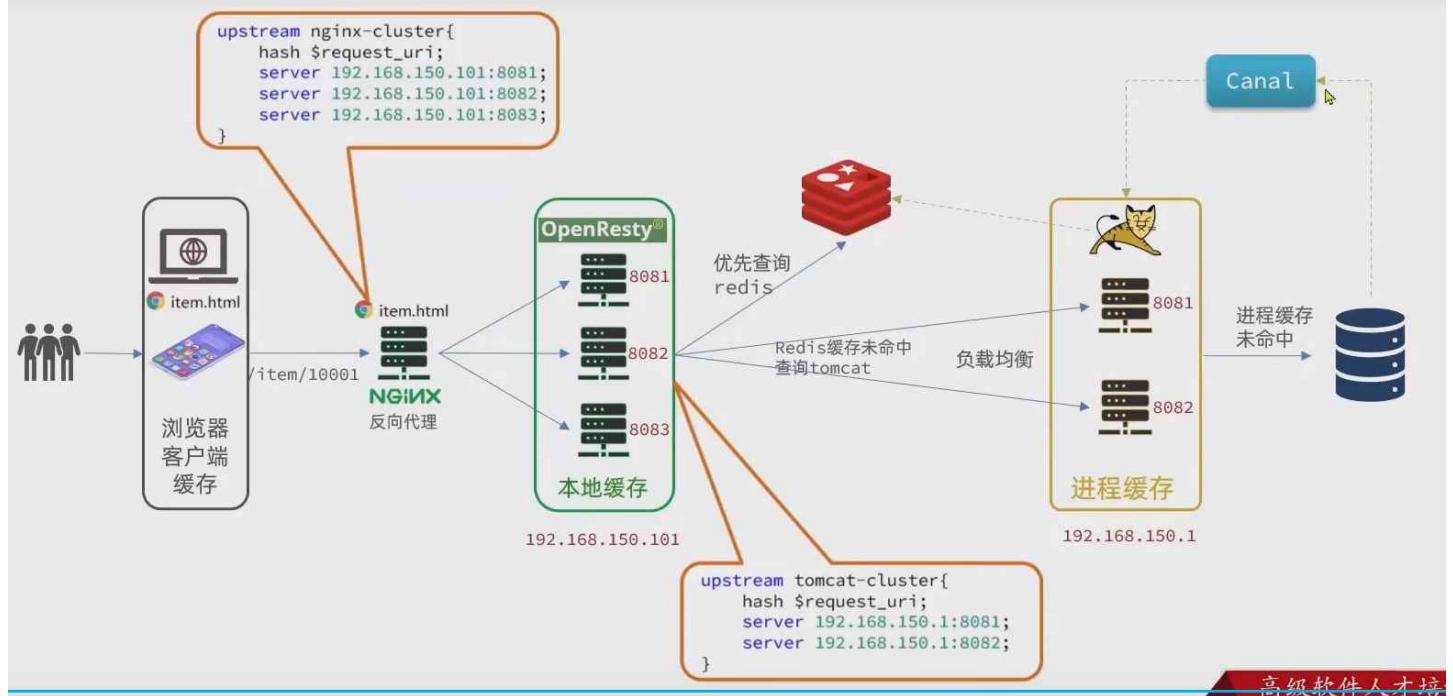
指定要监听的表
指定表关联的实体类
监听到数据库的增、改、删 的消息

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;
    @Column(name = "name")
    private String name;
    // ... 其它字段略
    private Date updateTime;
    @TableField(exist = false)
    @Transient
    private Integer stock;
    @TableField(exist = false)
    @Transient
    private Integer sold;
}
```

标记表中的id字段
标记表中与属性名不一致的字段
标记不属于表中的字段

多级缓存总结



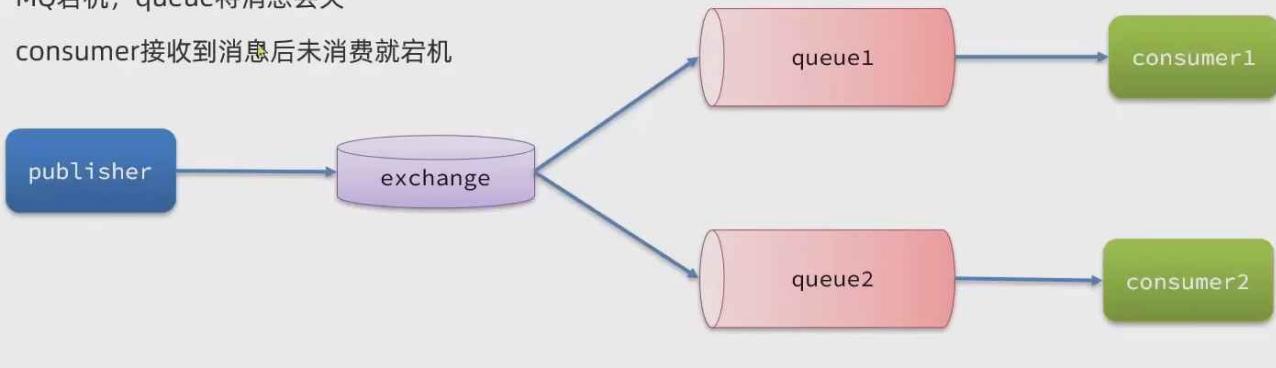
高级软件人才培训

MQ高级

消息可靠性问题

消息从生产者发送到exchange，再到queue，再到消费者，有哪些导致消息丢失的可能性？

- 发送时丢失：
 - 生产者发送的消息未送达exchange
 - 消息到达exchange后未到达queue
- MQ宕机，queue将消息丢失
- consumer接收到消息后未消费就宕机



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

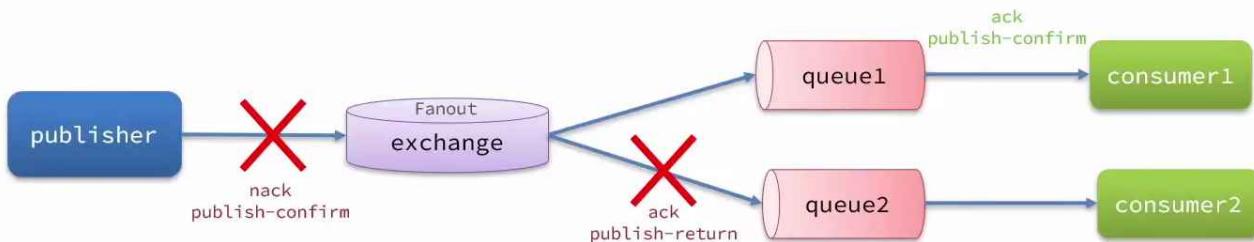
- 消息成功投递到交换机，返回ack
- 消息未投递到交换机，返回nack

注意

确认机制发送消息时，需要给每个消息设置一个全局唯一id，以区分不同消息，避免ack冲突

- 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中处理消息确认的几种情况：

- publisher-comfirm:

- 消息成功发送到exchange，返回ack
- 消息发送失败，没有到达交换机，返回nack
- 消息发送过程中出现异常，没有收到回执

- 消息成功发送到exchange，但没有路由到queue，

调用ReturnCallback

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

Spring中，交换机，队列，消息都是默认持久化的

消费者确认

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
```

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。

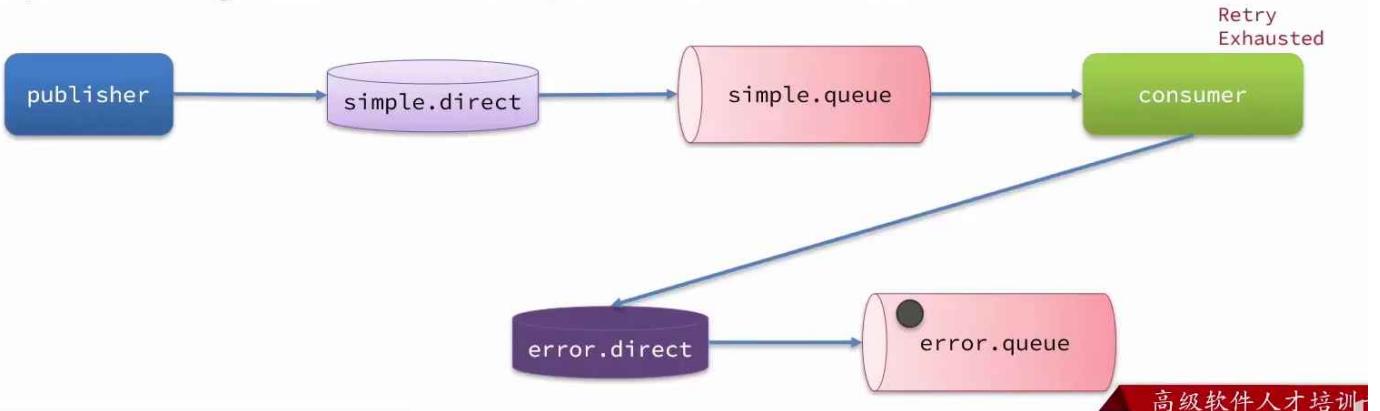
```
spring:  
  rabbitmq:  
    listener:  
      simple:  
        prefetch: 1  
        retry:  
          enabled: true # 开启消费者失败重试  
          initial-interval: 1000 # 初次的失败等待时长为1秒  
          multiplier: 2 # 下次失败的等待时长倍数，下次等待时长 = multiplier * last-interval  
          max-attempts: 3 # 最大重试次数  
          stateless: true # true无状态; false有状态。如果业务中包含事务，这里改为false
```

1 2 4
2x1 8

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



如何确保RabbitMQ消息的可靠性？

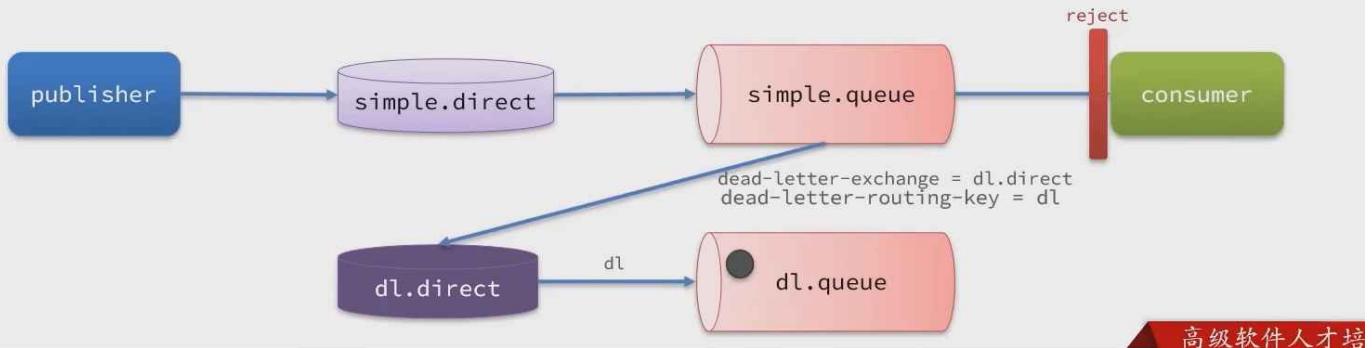
- 开启生产者确认机制，确保生产者的消息能到达队列
- 开启持久化功能，确保消息未消费前在队列中不会丢失
- 开启消费者确认机制为auto，由spring确认消息处理成功后完成ack
- 开启消费者失败重试机制，并设置MessageRecoverer，多次重试失败后将消息投递到异常交换机，交由人工处理

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）。



什么样的消息会成为死信？

- 消息被消费者reject或者返回nack
- 消息超时未消费
- 队列满了

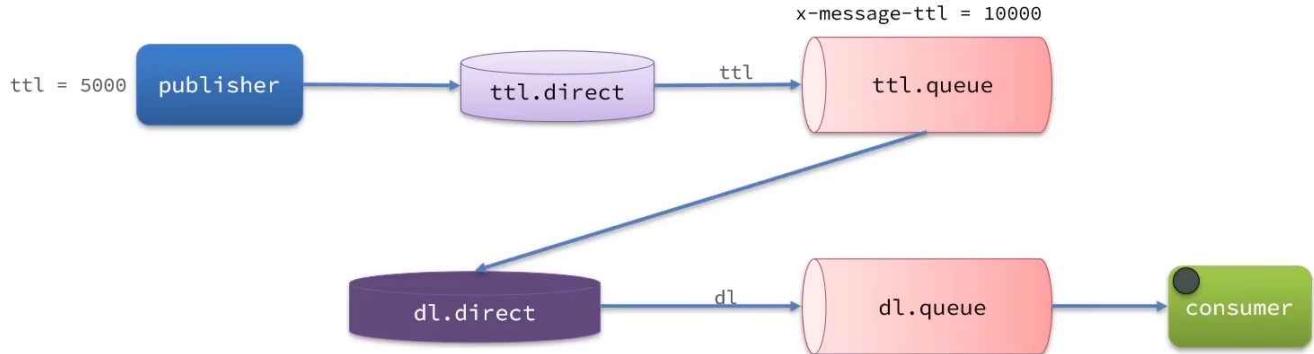
如何给队列绑定死信交换机？

- 给队列设置dead-letter-exchange属性，指定一个交换机
- 给队列设置dead-letter-routing-key属性，设置死信交换机与死信队列的RoutingKey

TTL

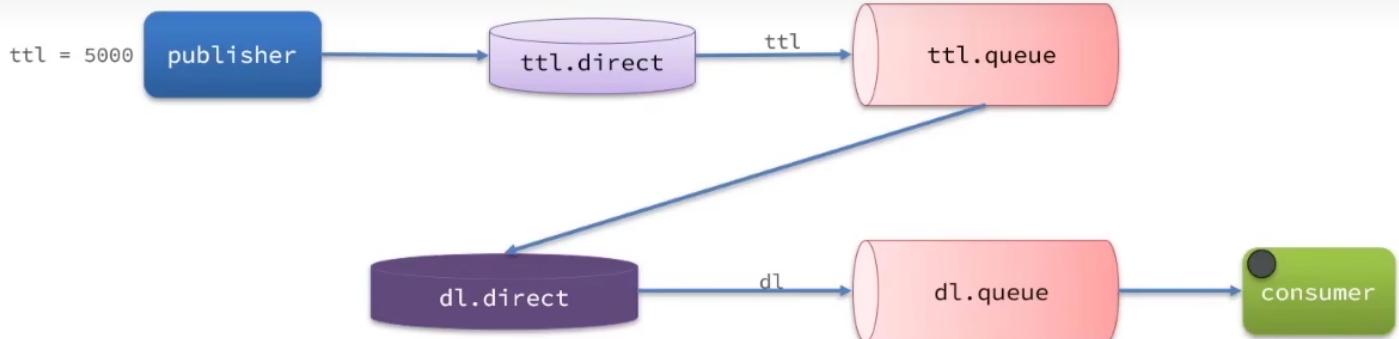
TTL，也就是Time-To-Live。如果一个队列中的消息TTL结束仍未消费，则会变为死信，ttl超时分为两种情况：

- 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



消息超时的两种方式是？

- 给队列设置ttl属性，进入队列后超过ttl时间的消息变为死信
- 给消息设置ttl属性，队列接收到消息超过ttl时间后变为死信
- 两者共存时，以时间短的ttl为准



如何实现发送一个消息20秒后消费者才收到消息？

- 给消息的目标队列指定死信交换机
- 消费者监听与死信交换机绑定的队列
- 发送消息时给消息设置ttl为20秒

延迟队列插件的使用步骤包括哪些？

- 声明一个交换机，添加delayed属性为true
- 发送消息时，添加x-delay头，值为超时时间

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
```

消息堆积问题的解决方案？

- 队列上绑定多个消费者，提高消费速度
- 给消费者开启线程池，提高消费速度
- 使用惰性队列，可以在mq中保存更多消息

惰性队列的优点有哪些？

- 基于磁盘存储，消息上限高
- 没有间歇性的page-out，性能比较稳定

惰性队列的缺点有哪些？

- 基于磁盘存储，消息时效性会降低
- 性能受限于磁盘的IO



集群分类

RabbitMQ是基于Erlang语言编写，而Erlang又是一个面向并发的语言，天然支持集群模式。RabbitMQ的集群有两种模式：

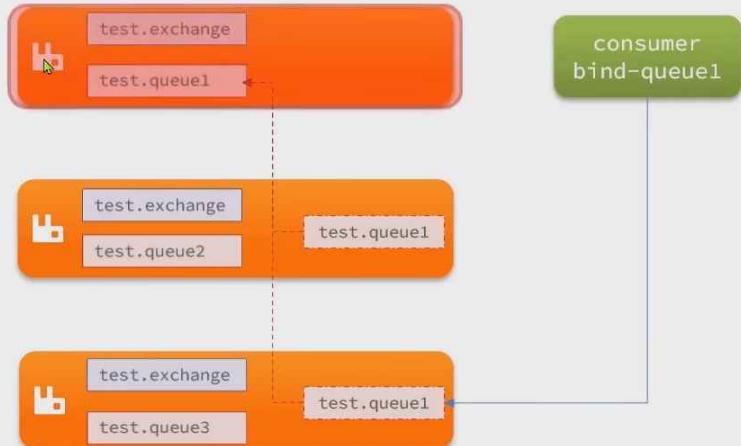
- **普通集群**：是一种分布式集群，将队列分散到集群的各个节点，从而提高整个集群的并发能力。
- **镜像集群**：是一种主从集群，普通集群的基础上，添加了主从备份功能，提高集群的数据可用性。

镜像集群虽然支持主从，但主从同步并不是强一致的，某些情况下可能有数据丢失的风险。因此在RabbitMQ的3.8版本以后，推出了新的功能：**仲裁队列**来代替镜像集群，底层采用Raft协议确保主从的数据一致性。

普通集群

普通集群，或者叫标准集群（classic cluster），具备下列特征：

- 会在集群的各个节点间共享部分数据，包括：交换机、队列元信息。不包含队列中的消息。
- 当访问集群某节点时，如果队列不在该节点，会从数据所在节点传递到当前节点并返回
- 队列所在节点宕机，队列中的消息就会丢失

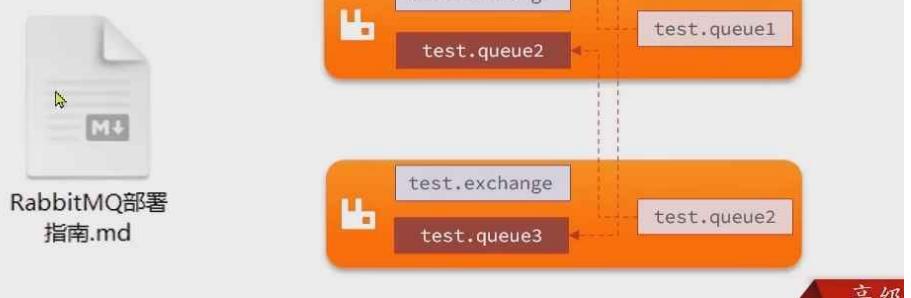


镜像集群

镜像集群：本质是主从模式，具备下面的特征：

- 交换机、队列、队列中的消息会在各个mq的镜像节点之间同步备份。
- 创建队列的节点被称为该队列的**主节点**，备份到的其它节点叫做该队列的**镜像节点**。
- 一个队列的主节点可能是另一个队列的镜像节点
- 所有操作都是主节点完成，然后同步给镜像节点
- 主宕机后，镜像节点会替代成新的主

详细的搭建步骤可以参考课前资料：



仲裁队列

仲裁队列：仲裁队列是3.8版本以后才有的新功能，用来替代镜像队列，具备下列特征：

- 与镜像队列一样，都是主从模式，支持主从数据同步
- 使用非常简单，没有复杂的配置
- 主从同步基于Raft协议，强一致

SpringCloud常见组件有哪些？

问题说明：这个题目主要考察对SpringCloud的组件基本了解

难易程度：简单

参考话术：

SpringCloud包含的组件很多，有很多功能是重复的。其中最常用组件包括：

- 注册中心组件：Eureka、Nacos等
- 负载均衡组件：Ribbon
- 远程调用组件：OpenFeign
- 网关组件：Zuul、Gateway
- 服务保护组件：Hystrix、Sentinel
- 服务配置管理组件：SpringCloudConfig、Nacos

Nacos的服务注册表结构是怎样的？

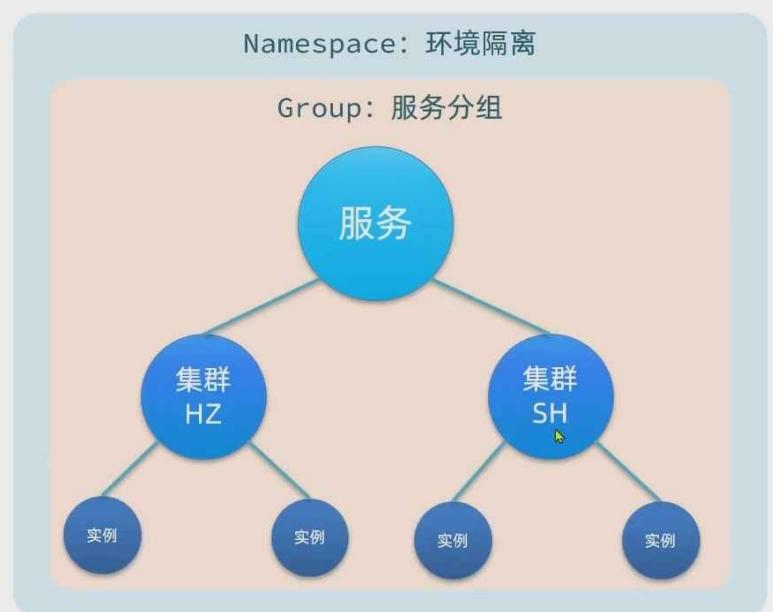
要了解Nacos的服务注册表结构，需要从两方面入手：

- 一是Nacos的分级存储模型
- 二是Nacos的服务端源码

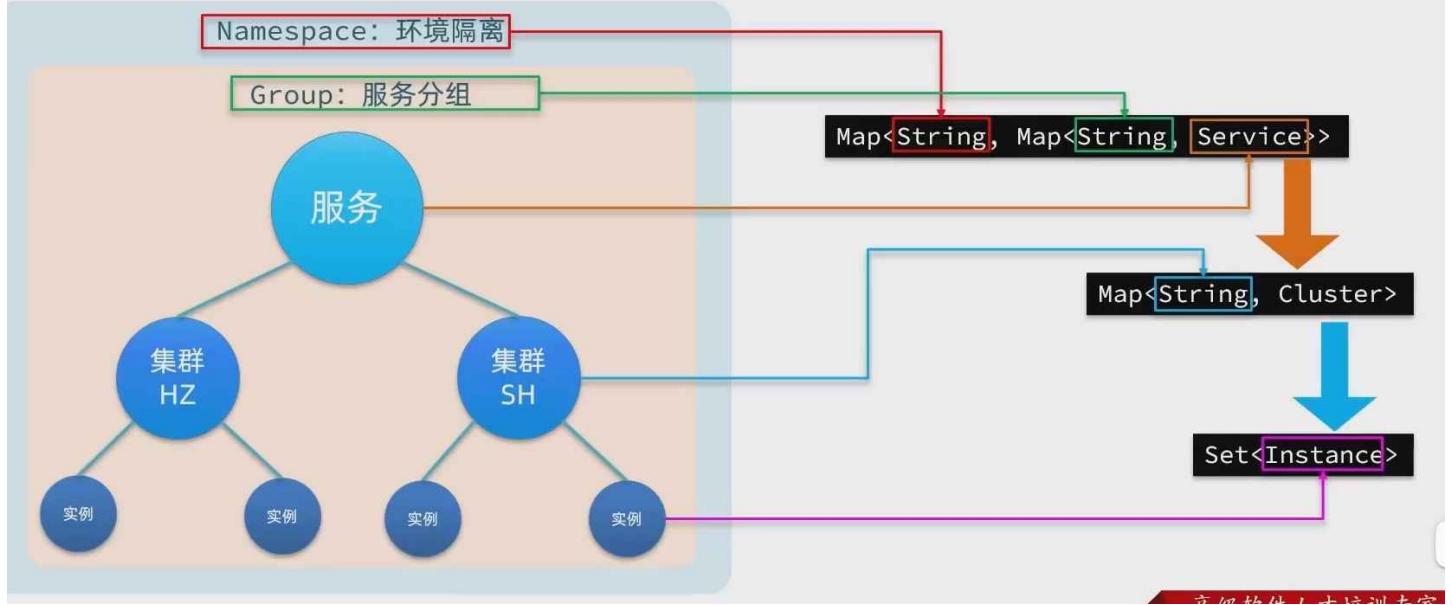
源码部分，可以参考课前资料提供的文档：



Nacos源码分析.md



Nacos的服务注册表结构是怎样的？



Nacos的服务注册表结构是怎样的？

问题说明：考察对Nacos数据分级结构的了解，以及Nacos源码的掌握情况

难度程度：一般

参考话术：

Nacos采用了数据的分级存储模型，最外层是Namespace，用来隔离环境。然后是Group，用来对服务分组。接下来就是服务（Service）了，一个服务包含多个实例，但是可能处于不同机房，因此Service下有多个集群（Cluster），Cluster下是不同的实例（Instance）。

对应到Java代码中，Nacos采用了一个多层的Map来表示。结构为`Map<String, Map<String, Service>>`，其中最外层Map的key就是namespaceId，值是一个Map。内层Map的key是group拼接serviceName，值是Service对象。Service对象内部又是一个Map，key是集群名称，值是Cluster对象。而Cluster对象内部维护了Instance的集合。

```
/**  
 * Map(namespace, Map(group::serviceName, Service)).  
 */  
private final Map<String, Map<String, Service>> serviceMap = new
```

Nacos如何支撑数十万服务注册压力？

问题说明：考察对Nacos源码的掌握情况

难度程度：难

参考话术：

Nacos内部接收到注册的请求时，不会立即写数据，而是将服务注册的任务放入一个阻塞队列就立即响应给客户端。然后利用线程池读取阻塞队列中的任务，异步来完成实例更新，从而提高并发写能力。

```
/**  
 * Map(namespace, Map(group::serviceName, Service)).  
 */  
private final Map<String, Map<String, Service>> serviceMap = new
```

```
public void doRegister(RegisterRequest request) {  
    Service service = getOrCreateService(request);  
    service.setIpPort(request.getIpPort());  
    service.setWeight(request.getWeight());  
}
```

