1.SpringBoot面试题

1.1.简单聊聊SpringBoot的作用是什么?

知识背景:

SpringBoot采用默认配置，帮助我们快速的构建和运行Spring项目:

- 简化spring初始搭建和开发过程
- 提供了大量的starter，集成了spring及大量第三方应用的自动配置
- 允许使用配置文件(properties或者yml文件)覆盖默认配置
- 创建独立的spring应用程序，使用main方法运行
- 内嵌Tomcat无需部署war包，直接打成jar包，用nohup java -jar – & 启动就好

面试话术:

SpringBoot是一个快速构建项目并简化项目配置的工具，内部集成了Tomcat及大多数第三方应用和Spring框架的默认配置。与我们学习的SpringMVC和SpringCloud并无冲突,SpringBoot提供的这些默认配置，大大简化了SpringMVC、SpringCloud等基于Spring的Web应用的开发。

1.2.SpringBoot的自动配置原理?

问题1：那你说说SpringBoot的自动配置是如何实现的?

面试话术:

一般我们的SpringBoot项目启动类都会添加`@SpringBootApplication`注解，而这个注解的其中一个二级注解是`@EnableAutoConfiguration`注解。而`@EnableAutoConfiguration`注解通过`@Import`注解，以`ImportSelector`接口的方法来导入classpath下的`META-INF/spring.factories`文件，这些文件中会指定需要加载的一些类名称。

这些类一般都加了`@Configuration`注解，并且完成了对某框架（例如Redis、SpringMVC）的默认配置，当这些类符合条件时，就会被实例化，其中的配置生效，那么自动配置自然生效了。

问题2：满足怎样的条件配置才会生效?

面试话术:

一般提供默认配置的类都会添加`@ConditionalOnXxx`这样的注解，例如：`@ConditionalOnClass`，`@ConditionalOnProperties`等。`@ConditionalOnClass`表示只有classpath中存在某些指定的类时，条件满足，此时该配置类才会生效。例如Redis的默认配置其实早就有了，但是只有你引入redis的starter依赖，才满足了条件，触发自动配置。

问题3：那如果我需要覆盖这些默认配置呢?

有两种方式可以覆盖默认配置:

- SpringBoot提供默认配置时，会在提供的Bean上加注解@ConditionalOnMissingBean，意思是如果这个Bean不存在时条件满足，那么我们只要配置了相同的Bean，那么SpringBoot提供的默认配

置就会失效
- SpringBoot提供默认配置时，一些关键属性会通过读取application.yml或者application.properties文件来获取，因此我们可以通过覆盖任意一个文件中的属性来覆盖默认配置。

1.3.有没有自定义过SpringBoot的stater?

面试话术：

有，项目中某些中间件的客户端（如Redis、ElasticSearch）会进行二次封装，并通过starter方式提供jar包，供大家使用。

一般定义starter包括下面几个子工程：

- xxx-spring-boot-starter：pom格式，管理当前starter中需要的各种依赖
- xxx-spring-boot-autoconfigure：jar格式，自动配置的核心代码

我以elasticsearch为例来说说autoconfigure中包含哪些

- elasticsearch的工具类
- 属性加载的类，一般通过@ConfigurationProperties注解读取yaml文件中的es地址
- 添加了@Configuration的配置类，作用是初始化elasticsearch工具类，初始化elasticsearch客户端，初始化一些其它必备的实例。
- resource下定义META-INF文件夹，并且文件夹下定义spring.factories文件，文件中是key-value形式
    - key是EnableAutoConfiguration这个注解的全路径名
    - value是我们自定义自动配置类（加了@Configuration的类），如果有多个以","隔开

1.4.SpringBoot项目的启动流程

话术：

SpringBoot项目启动第一步就是创建SpringApplication的实例，并且调用SpringApplication.run()这个方法。

创建SpringApplication实例主要完成三件事情：

- 记录当前启动类字节码
- 判断当前项目类型，普通Servlet、响应式WebFlux、NONE
- 加载/META-INF/spring.factories文件，初始化ApplicationContextInitializer和ApplicationListener实例

而后的run()方法则会创建spring容器，流程如下：

- 准备监听器，监听Spring启动的各个过程
- 创建并配置环境参数Environment
- 创建ApplicationContext
- `prepareContext()`：初始化ApplicationContext，准备运行环境
- `refreshContext(context)`：准备Bean工厂，调用一个BeanDefinition和BeanFactory的后处理器，初始化各种Bean，初始化tomcat
- `afterRefresh()`：拓展功能，目前为空

- 发布容器初始化完毕的事件

1.5.SpringBoot的配置加载优先级

面试话术：

SpringBoot参数配置方式很多，比较常用参数配置方式按照优先级从高到低分别是：

- 在命令行中传入的参数
- java 的系统属性，可以通过System.getProperties()获得的内容
- 操作系统的环境变量
- 针对不同{profile}环境的配置文件内容，例如 applicaiton-{profile}.yaml
- application.yml或application.proerties文件
- 在@Configration注解修改的类中，通过@PropertySource注解定义的属性

2.SpringCloud相关

2.1.SpringCloud和Dubbo的区别

聊聊看SpringCloud和Dubbo有什么区别?

面试话术：

两者都是现在主流的微服务框架，但却存在不少差异：

- 初始定位不同：SpringCloud定位为微服务架构下的一站式解决方案；Dubbo 是 SOA 时代的产物，它的关注点主要在于服务的调用和治理
- 生态环境不同：SpringCloud依托于Spring平台，具备更加完善的生态体系；而Dubbo一开始只是做RPC远程调用，生态相对匮乏，现在逐渐丰富起来。
- 调用方式：SpringCloud是采用Http协议做远程调用，接口一般是Rest风格，比较灵活；Dubbo是采用Dubbo协议，接口一般是Java的Service接口，格式固定。但调用时采用Netty的NIO方式，性能较好。
- 组件差异比较多，例如SpringCloud注册中心一般用Eureka，而Dubbo用的是Zookeeper

SpringCloud生态丰富，功能完善，更像是品牌机，Dubbo则相对灵活，可定制性强，更像是组装机。

相关资料：

SpringCloud：Spring公司开源的微服务框架，SpirngCloud 定位为微服务架构下的一站式解决方案，生态丰富，功能完善。

Dubbo：阿里巴巴开源的RPC框架，Dubbo 是 SOA 时代的产物，它的关注点主要在于服务的调用，流量分发、流量监控和熔断。

SpringCloudAlibaba

两者的生态对比：

功能	Dubbo	SpringCloud
服务注册中心	Zookeeper	Eureka(主流）、Consul、zookeeper
服务调用方式	RPC基于Dubbo协议	REST API 基于Http协议
服务监控	Dubbo-Monitor	Spring Boot Admin
熔断器	不完善	Spring Cloud Netflix Hystrix
服务网关	无	Spring Cloud Netflix Zuul、Gateway
分布式配置	无	Spring Cloud Config
服务跟踪	无	Spring Cloud Sleuth+Zipkin(一般)
数据流	无	Spring Cloud Stream
批量任务	无	Spring Cloud Task
信息总线	无	Spring Cloud Bus

Spring Cloud 的功能很明显比 Dubbo 更加强大，涵盖面更广，而且作为 Spring 的旗舰项目，它也能够与 Spring Framework、Spring Boot、Spring Data、Spring Batch 等其他 Spring 项目完美融合，这些对于微服务而言是至关重要的。

使用 Dubbo 构建的微服务架构就像组装电脑，各环节选择自由度很高，但是最终结果很有可能因为一条内存质量不行就点不亮了，总是让人不怎么放心，但是如果使用者是一名高手，那这些都不是问题。

而 Spring Cloud 就像品牌机，在 Spring Source 的整合下，做了大量的兼容性测试，保证了机器拥有更高的稳定性，但是如果要在使用非原装组件外的东西，就需要对其基础原理有足够的了解。

2.2.dubbo和Feign远程调用的差异

面试话术：

Feign是SpringCloud中的远程调用方式，基于成熟Http协议，所有接口都采用Rest风格。因此接口规范更统一，而且只要符合规范，实现接口的微服务可以采用任意语言或技术开发。但受限于http协议本身的特点，请求和响应格式臃肿，其通信效率相对会差一些。

Dubbo框架默认采用Dubbo自定义通信协议，与Http协议一样底层都是TCP通信。但是Dubbo协议自定义了Java数据序列化和反序列化方式、数据传输格式，因此Dubbo在数据传输性能上会比Http协议要好一些。

不过这种性能差异除非是达极高的并发量级，否则无需过多考虑。

相关资料：

Dubbo采用自定义的Dubbo协议实现远程通信，是一种典型的RPC调用方案，而SpringCloud中使用的Feign是基于Rest风格的调用方式。

1）Rest风格

REST是一种架构风格，指的是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是 RESTful。

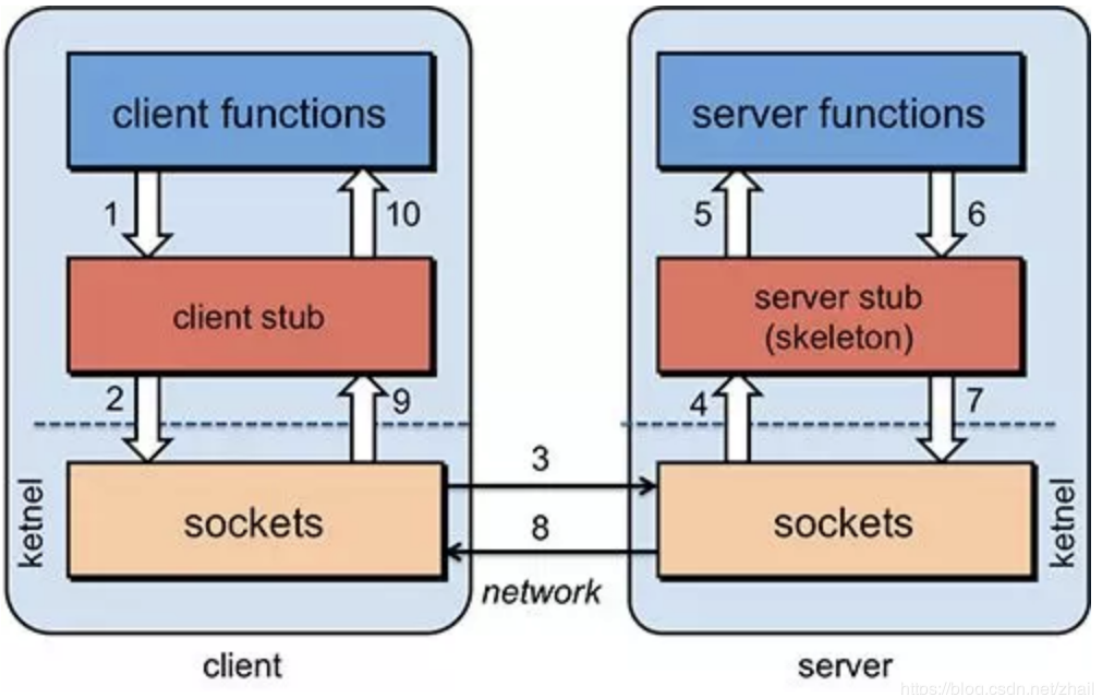
Rest的风格可以完全通过HTTP协议实现，使用 HTTP 协议处理数据通信。REST架构对资源的操作包括获取、创建、修改和删除资源的操作正好对应HTTP协议提供的GET、POST、PUT和DELETE方法。

因此请求和想要过程只要遵循http协议即可，更加灵活

SpringCloud中的Feign就是Rest风格的调用方式。

2）RPC

Remote Procedure Call，远程过程调用，就是像调用本地方法一样调用远程方法。RPC架构图：



RPC一般要确定下面几件事情：

- 数据传输方式：多数RPC框架选择TCP作为传输协议，性能比较好。
- 数据传输内容：请求方需要告知需要调用的函数的名称、参数、等信息。
- 序列化方式：客户端和服务端交互时将参数或结果转化为字节流在网络中传输，那么数据转化为字节流的或者将字节流转换成能读取的固定格式时就需要进行序列化和反序列化

因为有序列化和反序列化的需求，因此对数据传输格式有严格要求，不如Http灵活

Dubbo协议就是RPC的典型代表。

我们看看Dubbo协议和Feign的调用区别：

	Dubbo	**Feign（Http调用）**
传输协议	TCP	TCP
开发语言	java	不限
性能	好	一般
灵活性	一般	好

2.3.Eureka和Zookeeper注册中心的区别

面试话术：

SpringCloud和Dubbo都支持多种注册中心，不过目前主流来看SpringCloud用Eureka较多，Dubbo则以Zookeeper为主。两者存在较大的差异：

- 从集群设计来看：Eureka集群各节点平等，没有主从关系，因此可能出现数据不一致情况；ZK为了满足一致性，必须包含主从关系，一主多从。集群无主时，不对外提供服务
- CAP原则来看：Eureka满足AP原则，为了保证整个服务可用性，牺牲了集群数据的一致性；而Zookeeper满足CP原则，为了保证各节点数据一致性，牺牲了整个服务的可用性。
- 服务拉取方式来看：Eureka采用的是服务主动拉取策略，消费者按照固定频率（默认30秒）去Eureka拉取服务并缓存在本地；ZK中的消费者首次启动到ZK订阅自己需要的服务信息，并缓存在本地。然后监听服务列表变化，以后服务变更ZK会推送给消费者。

相关资料：

首先，Eureka和Zookeeper都是服务治理框架，但是设计上有一定的差别。

先看下CAP原则：C-数据一致性；A-服务可用性；P-服务对网络分区故障的容错性，这三个特性在任何分布式系统中不能同时满足，最多同时满足两个。

- Eureka满足AP，Zookeeper满足CP

  当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接down掉不可用。也就是说，服务注册功能对可用性的要求要高于一致性。但是Zookeeper和Eureka在一致性与可用性间做出了不同的选择。

  - `Zookeeper`：Zookeeper的设计追求数据的一致性，不保证服务的可用性。当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30 ~ 120s, 且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得zk集群失去master节点是较大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。
  - `Eureka`：Eureka追求的是服务的可用性，从而牺牲了数据的一致性。Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册或时如果发现连接失败，则会自动切换至其它节点，只要有一台Eureka还在，就能保证注册服务可用(保证可用性)，只不过查到的信息可能不是最新的(不保证强一致性)。除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况。
    - Eureka不再从注册列表中移除因为长时间没收到心跳而应该过期的服务
    - Eureka仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)
    - 当网络稳定时，当前实例新的注册信息会被同步到其它节点中

    因此， Eureka可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像zookeeper那样使整个注册服务瘫痪。
- Eureka集群各节点平等，Zookeeper中有主从之分
  - 如果Zookeeper集群中部分宕机，可能会导致整个集群因为选主而阻塞，服务不可用
  - eureka集群宕机部分，不会对其它机器产生影响
- Eureka的服务发现需要主动去拉取，Zookeeper服务发现是监听机制
  - eureka中获取服务列表后会缓存起来，每隔30秒重新拉取服务列表
  - zookeeper则是监听节点信息变化，当服务节点信息变化时，客户端立即就得到通知

2.4.SpringCloud中的常用组件有哪些?

Spring Cloud的子项目很多，比较常见的都是Netflix开源的组件：

- Spring Cloud Config
  集中配置管理工具，分布式系统中统一的外部配置管理，默认使用Git来存储配置，可以支持客户端配置的刷新及加密、解密操作。
- Spring Cloud Netflix
  Netflix OSS 开源组件集成，包括Eureka、Hystrix、Ribbon、Feign、Zuul等核心组件。
  - Eureka：服务治理组件，包括服务端的注册中心和客户端的服务发现机制；
  - Ribbon：负载均衡的服务调用组件，具有多种负载均衡调用策略；
  - Hystrix：服务容错组件，实现了断路器模式，为依赖服务的出错和延迟提供了容错能力；
  - Feign：基于Ribbon和Hystrix的声明式服务调用组件；
  - Zuul：API网关组件，对请求提供路由及过滤功能。
- Spring Cloud Bus
  用于传播集群状态变化的消息总线，使用轻量级消息代理链接分布式系统中的节点，可以用来动态刷新集群中的服务配置。
- Spring Cloud Consul
  基于Hashicorp Consul的服务治理组件。
- Spring Cloud Security
  安全工具包，对Zuul代理中的负载均衡OAuth2客户端及登录认证进行支持。
- Spring Cloud Sleuth
  Spring Cloud应用程序的分布式请求链路跟踪，支持使用Zipkin、HTrace和基于日志（例如ELK）的跟踪。
- Spring Cloud Stream
  轻量级事件驱动微服务框架，可以使用简单的声明式模型来发送及接收消息，主要实现为Apache Kafka及RabbitMQ。
- Spring Cloud Task
  用于快速构建短暂、有限数据处理任务的微服务框架，用于向应用中添加功能性和非功能性的特性。
- Spring Cloud Zookeeper
  基于Apache Zookeeper的服务治理组件。
- Spring Cloud Gateway
  API网关组件，对请求提供路由及过滤功能。
- Spring Cloud OpenFeign
  基于Ribbon和Hystrix的声明式服务调用组件，可以动态创建基于Spring MVC注解的接口实现用于服务调用，在Spring Cloud 2.0中已经取代Feign成为了一等公民。

2.5.微服务调用关系复杂，如何做监控和错误排查?

面试话术：

企业中对于微服务监控有一套东西，叫做APM。比如：SpringCloudSeluth+Zipkin，Pinpoint、Skywalking，可以实现性能监控、链路跟踪（精确到某个代码，某条sql）、CPU运行情况，链路运行耗时。

当然， 还可以借助于分布式日志管理系统。把项目运行的日志收集，形成统计报表，放入elasticsearch，便于搜索查看。比如：ELK技术栈、GrayLog

2.6.Hystix的作用是什么?

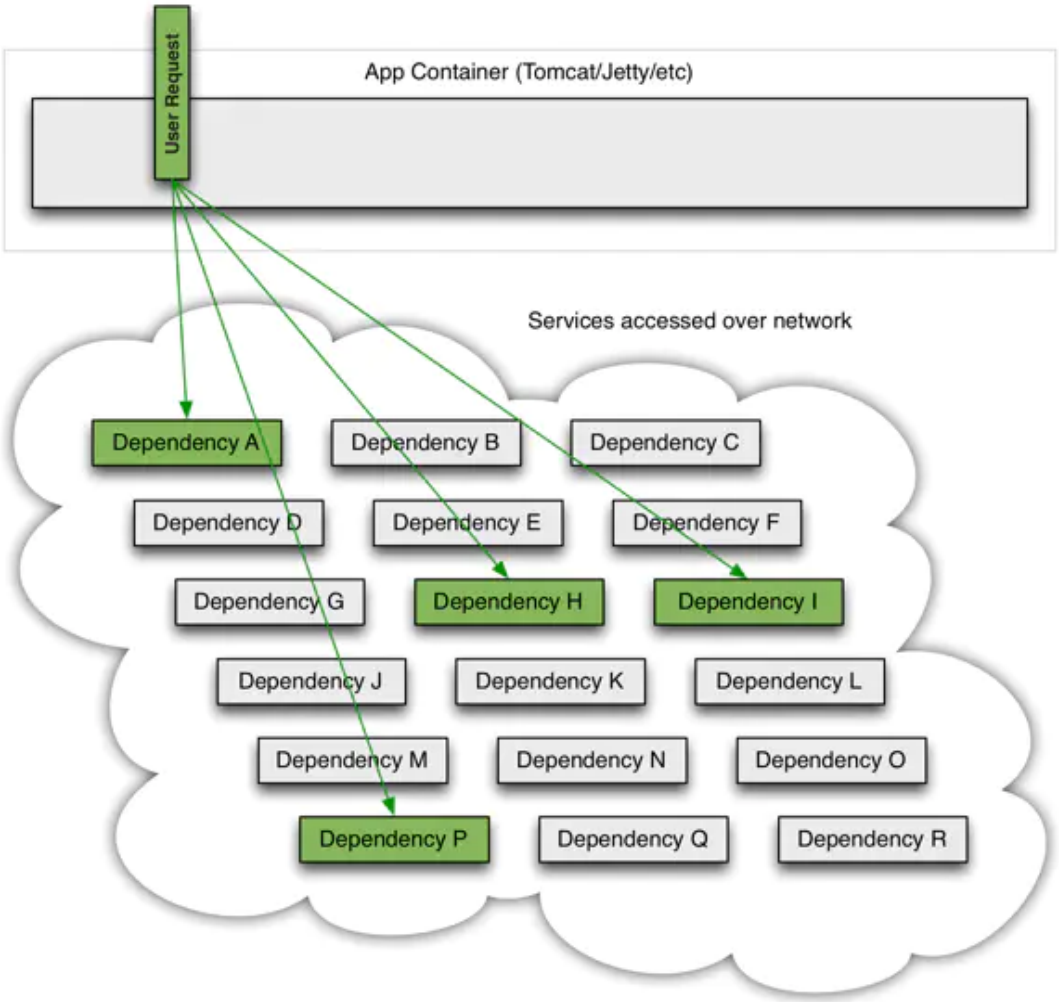
话术:

Hystix是Netflix开源的一个延迟和容错库，用于隔离访问远程服务、第三方库，防止出现级联失败。比较常用的手段就是线程隔离和服务熔断。

资料:

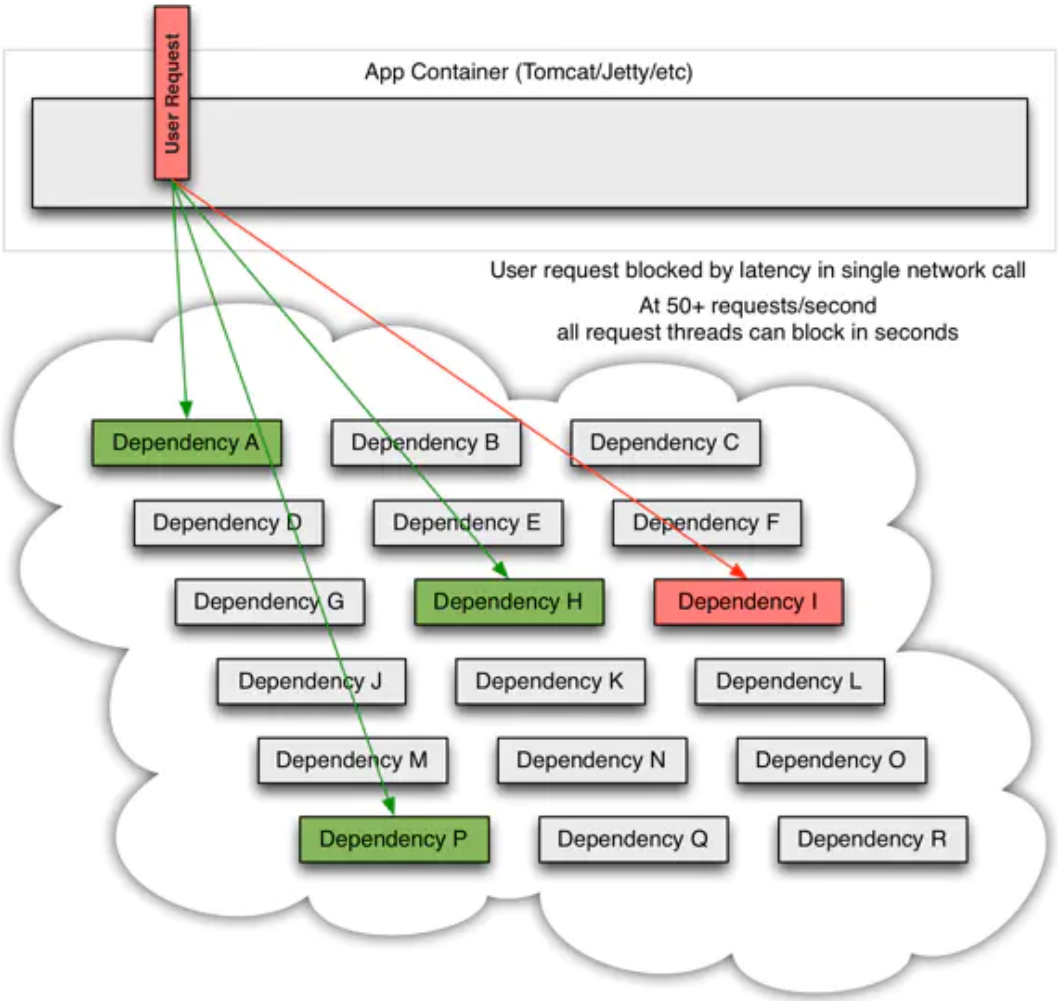
1）什么是雪崩问题（级联失败）？

在大中型分布式系统中，微服务间调用关系复杂，可能业务会调用多个其它服务，如下图:

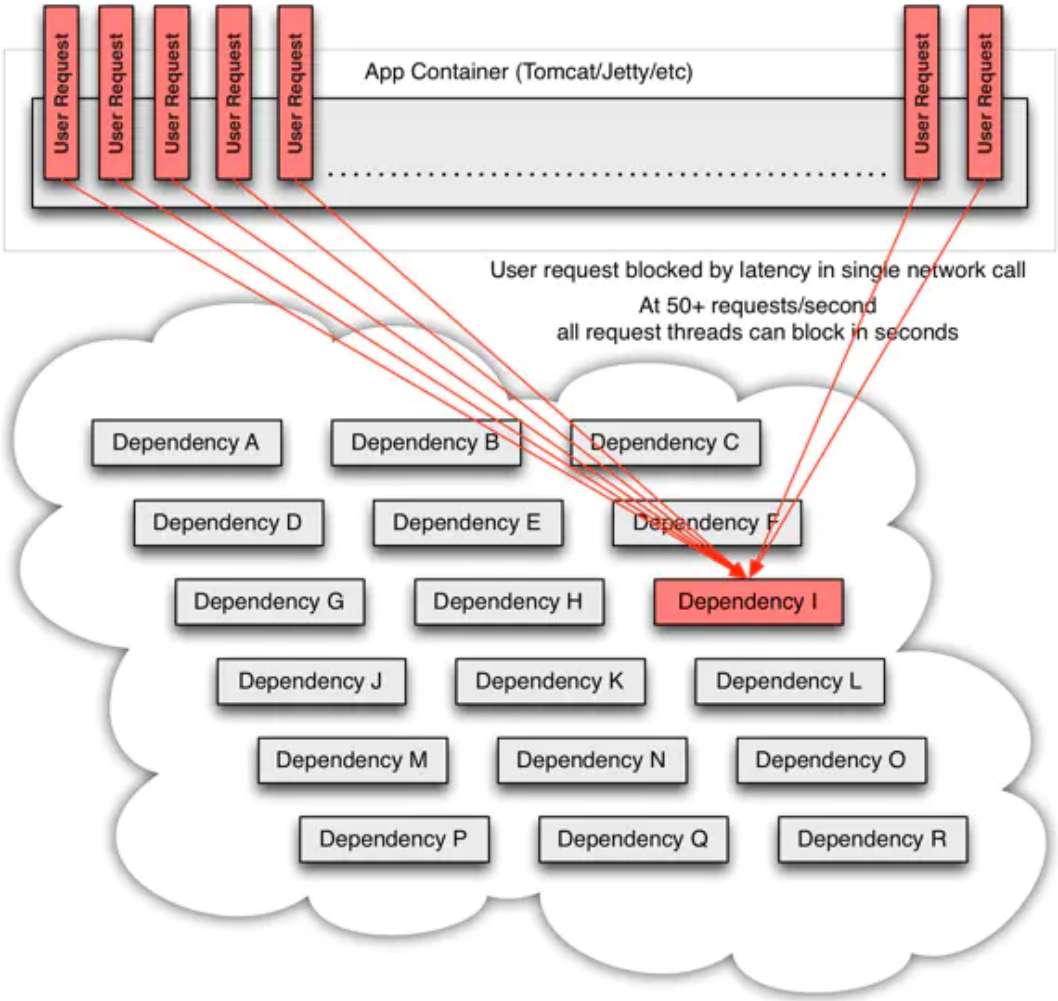


在高并发访问下,这些依赖的稳定性与否对系统的影响非常大,但是依赖有很多不可控问题:如网络连接缓慢，资源繁忙，暂时不可用，服务脱机等.

如下图：依赖 I 出现不可用，但是其他依赖仍然可用.访问服务A或H的不会有问题，但是访问服务I的请求会阻塞。

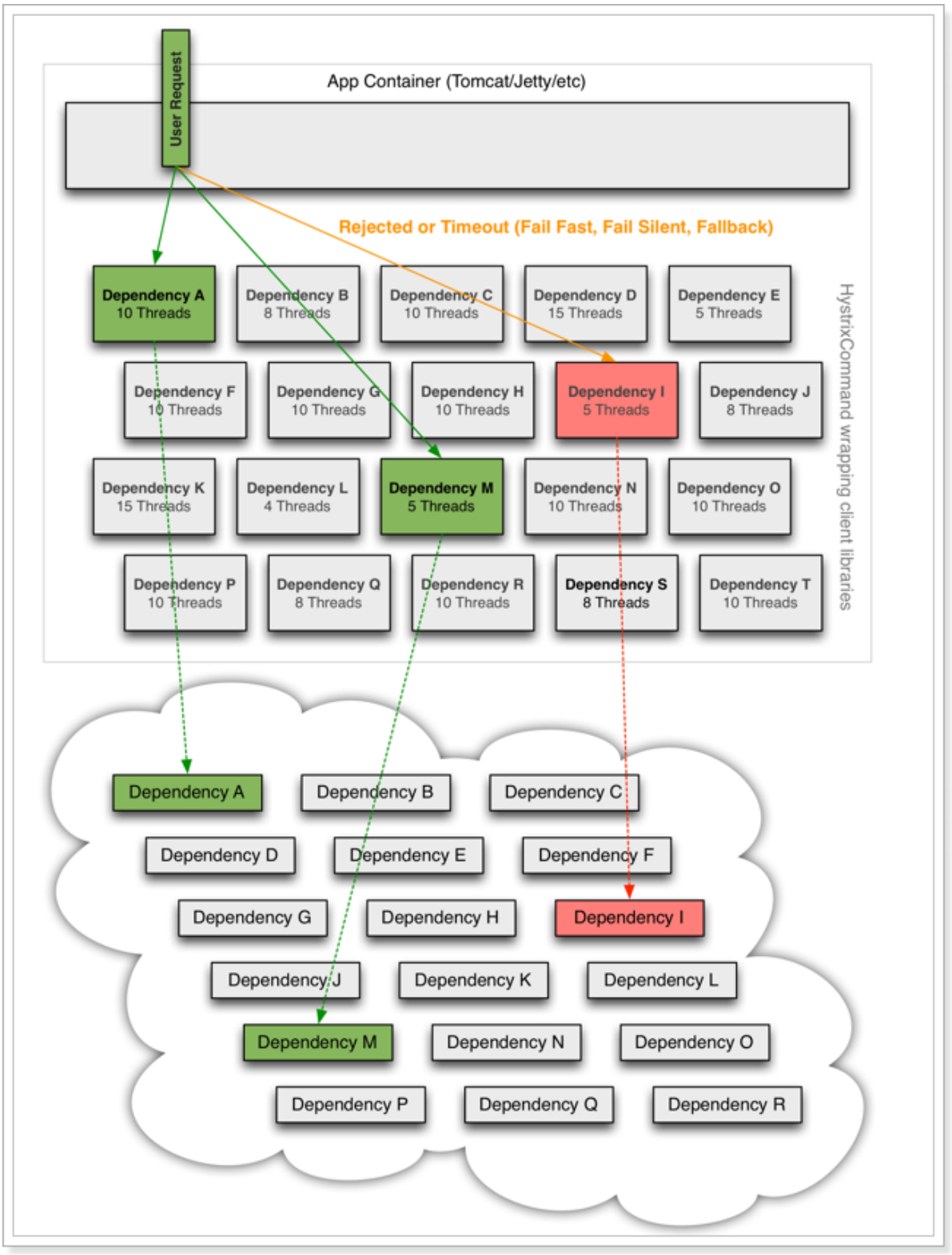


当依赖I 阻塞时,当前服务接收的请求会越来越多被阻塞，直到当前tomcat资源被耗尽，没有更多线程可用，导致整个服务崩溃。



2）线程隔离

线程隔离示意图：



解读：

Hystrix为每个依赖服务调用分配一个小的线程池，如果线程池已满调用将被立即拒绝，默认不采用排队.加速失败判定时间。

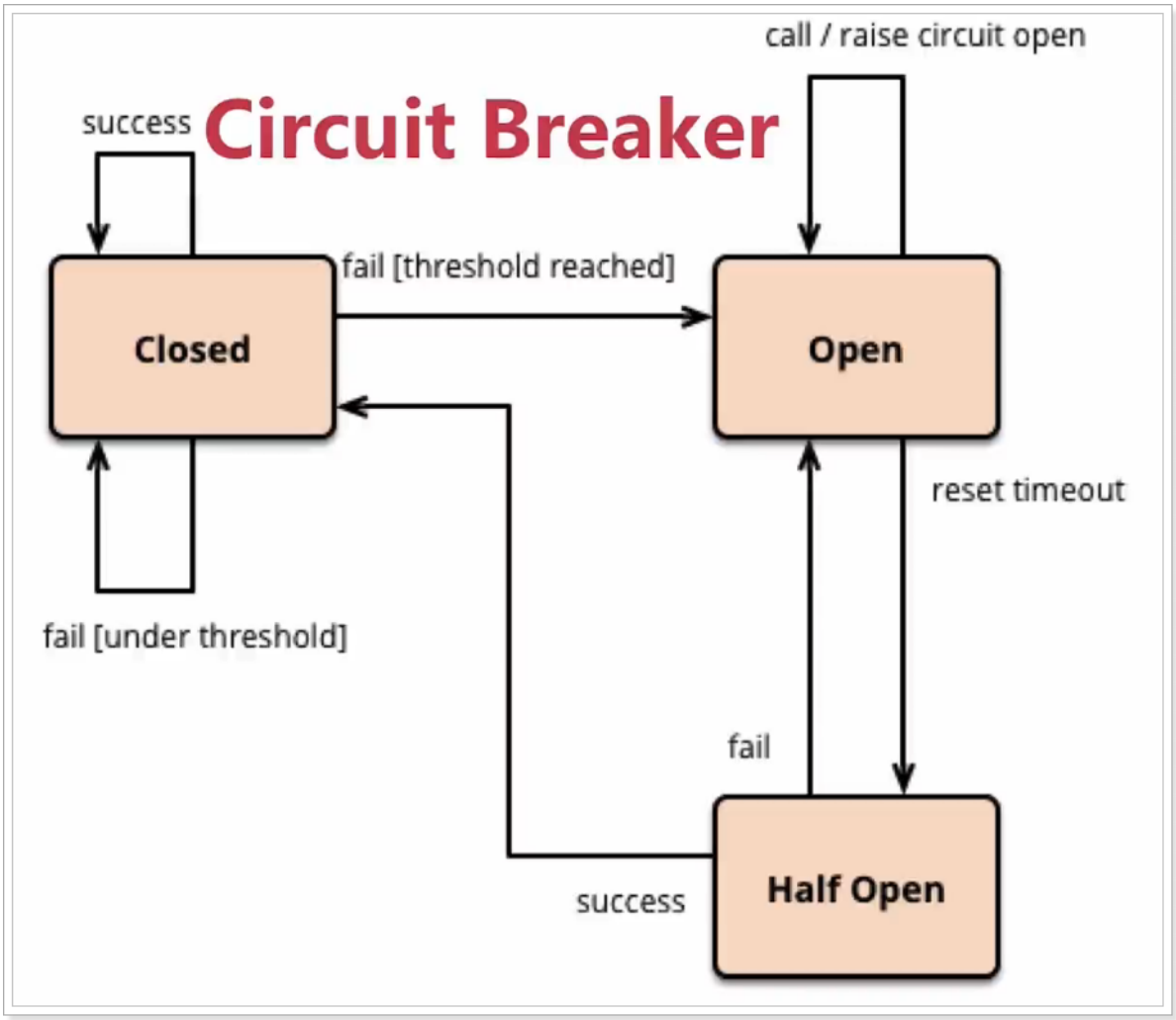
用户的请求将不再直接访问服务，而是通过线程池中的空闲线程来访问服务，如果**线程池已满**，或者**请求超时**，则会进行降级处理，执行fallback(降级)逻辑

3）服务熔断

尽管隔离可以避免服务出现级联失败，但是对于访问**服务I（异常服务）**的其它服务，每次处理请求都要等待数秒直至fallback，显然是对系统资源的浪费。

因此，当Hystix判断一个依赖服务失败比例较高时，就会对其做熔断处理，快速失败，不再阻塞等待。

Hystix的熔断状态机模型：



状态机有3个状态：

- Closed：关闭状态（断路器关闭），所有请求都正常访问。
- Open：打开状态（断路器打开），所有请求都会被降级。Hystix会对请求情况计数，当一定时间内失败请求百分比达到阈值，则触发熔断，断路器打开。默认失败比例的阈值是50%，请求次数最少不低于20次。
- Half Open：半开状态，open状态不是永久的，打开后会进入休眠时间（默认是5S）。随后断路器会自动进入半开状态。此时会释放1次请求通过，若这个请求是健康的，则会关闭断路器，否则继续保持打开，再次进行5秒休眠计时。

3.RabbitMQ常见面试题

3.1.你们公司为什么选择了RabbitMQ产品，而不是RocketMQ和Kafka（问区别）？

面试话术：

kafka是以吞吐量高而闻名，不过其数据稳定性一般，而且无法保证消息有序性。

阿里巴巴的RocketMQ基于Kafka的原理，利用Java代码打造，弥补了Kafka的缺点，继承了其高吞吐的优势，其客户端目前以Java为主。

RabbitMQ基于面向并发的语言Erlang开发，吞吐量不如Kafka，但是消息可靠性较好。也能有效的保证消息的有序性。因为Erlang的原因，集群搭建比较方便。支持多种协议，并且有各种语言的客户端，比较灵活。

综合考虑我们公司的并发需求以及稳定性需求，我们选择了RabbitMQ。

相关资料：

这个问题其实时问3种MQ的差别，先看一张图：

特性	ActiveMQ	RabbitMQ	RocketMQ	kafka
开发语言	java	erlang	java	scala
单机吞吐量	万级	万级	10万级	10万级
时效性	ms级	us级	ms级	ms级以内
可用性	高(主从架构)	高(主从架构)	非常高(分布式架构)	非常高(分布式架构)
功能特性	成熟的产品，在很多公司得到应用；有较多的文档；各种协议支持较好	基于erlang开发，所以并发能力很强，性能极其好，延时很低;管理界面较丰富	MQ功能比较完备，扩展性佳	只支持主要的MQ功能，像一些消息查询，消息回溯等功能没有提供，毕竟是为大数据准备的，在大数据领域应用广。

ActiveMQ现在已经很少使用，社区不太活跃，放弃。

RabbitMQ并发能力强、消息延时低、高可用、管理界面丰富，并且最重要的是：社区非常活跃，出现BUG都能及时解决。

Kafka和RocketMQ的特点都是高吞吐量，但是kafka消息可靠性比较一般，而且消息不保证有序性。RocketMQ弥补了Kafka的缺点，不过是阿里开源，社区不太活跃，文档也不够丰富。

3.2.在项目中哪些地方使用了MQ，解决了什么问题?

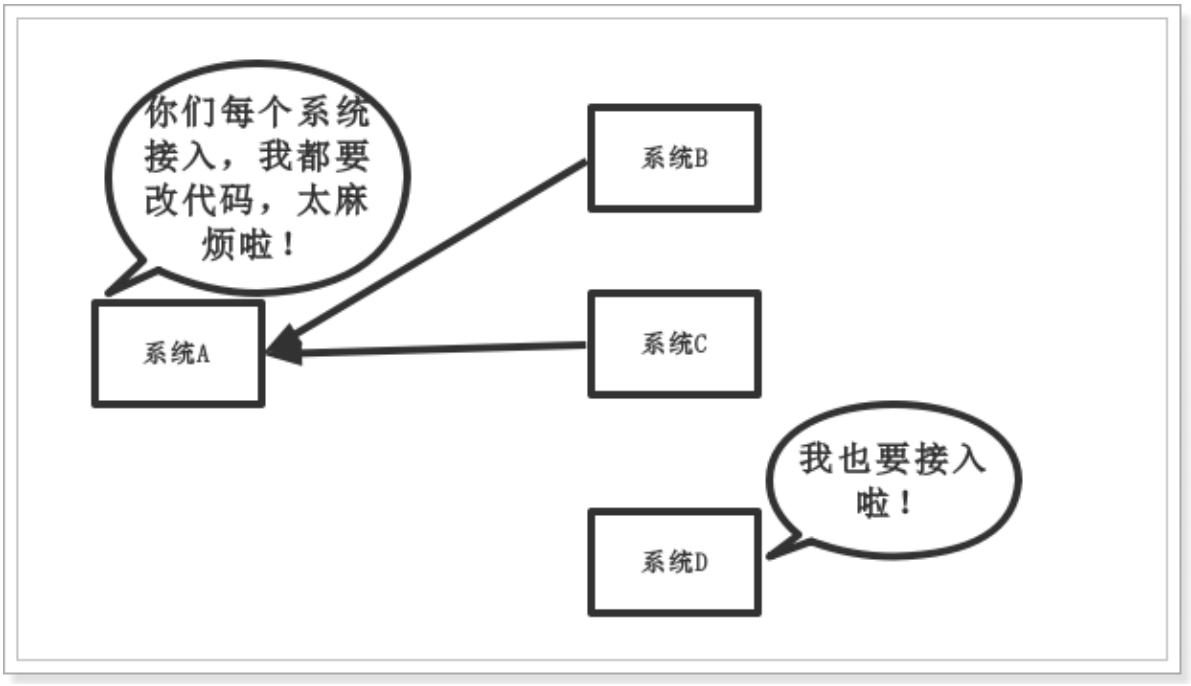
这个问题主要问的时MQ的作用，包括下列几点：

- 解耦合
- 流量削峰
- 异步执行
- 延迟队列

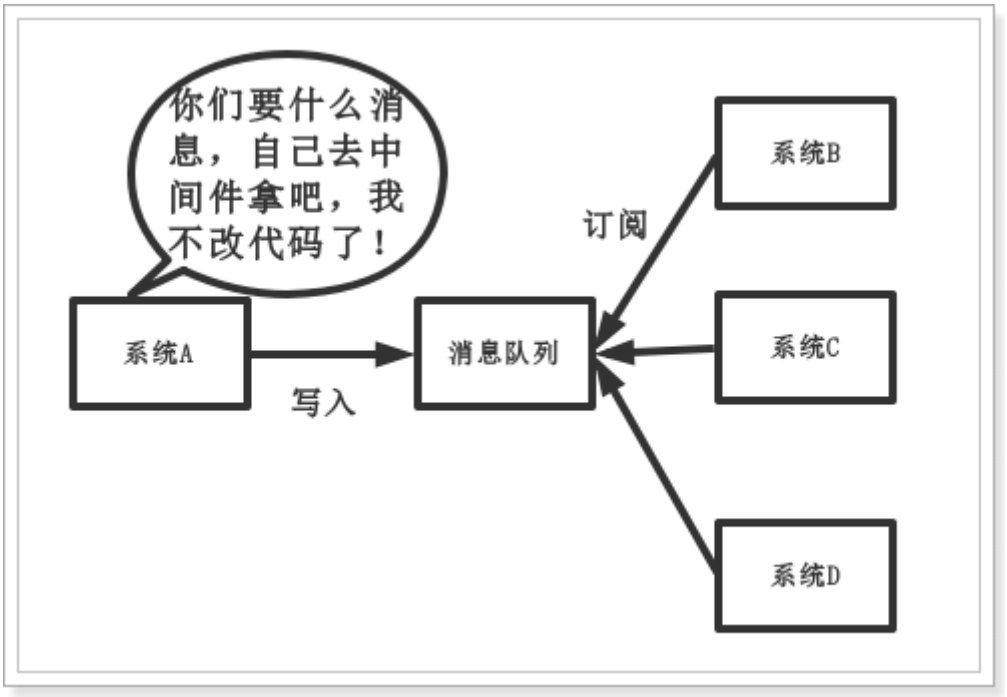
1）解耦合

例如系统A执行完业务，系统B需要得到系统A的业务结果，此时可以系统A中调用系统B（系统A中耦合了系统B的业务）。

此时如果系统C、系统D都有类似需求，那么系统A的业务逻辑还要继续修改，违反了开闭原则。



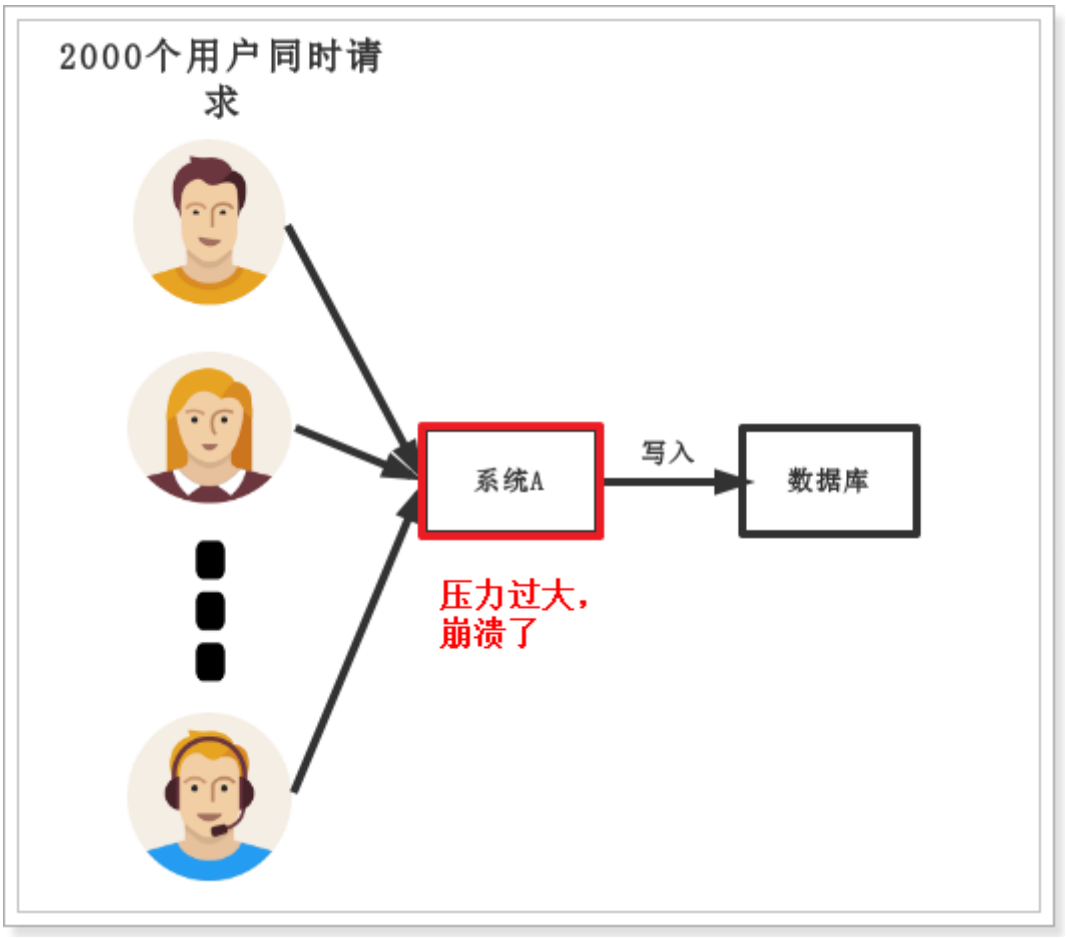
此时，可以利用MQ来解耦，让商品微服务发送消息通知，而相关的其它系统监听MQ即可：



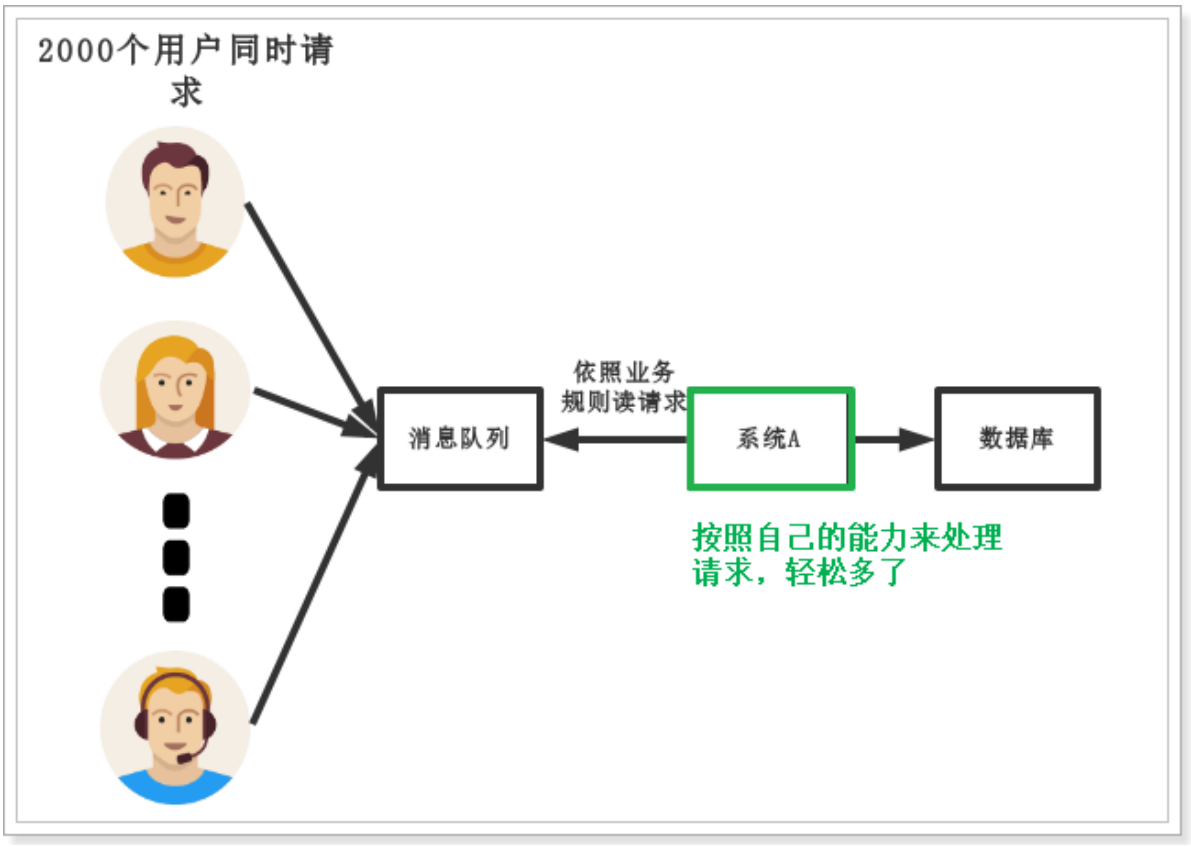
2）流量削峰

数据库的并发能力有限，往往称为业务执行的性能瓶颈。

例如我们的服务只能支持500的并发，然而又每秒1000甚至更高的服务流量涌入，服务肯定会崩溃的。

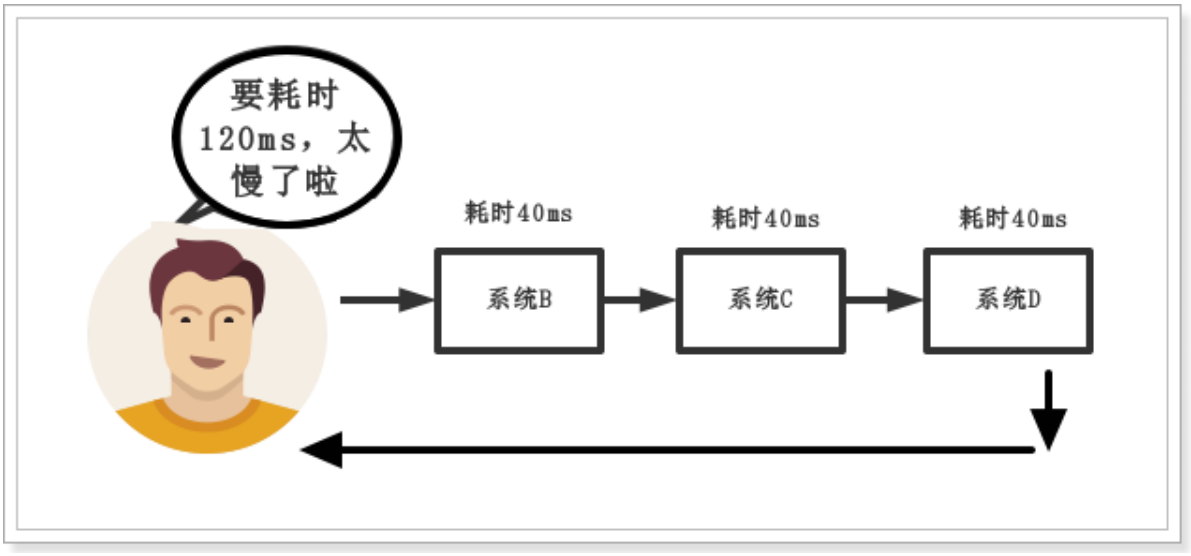


此时，利用MQ来作为缓冲，就像大坝一样，高并发流量涌入，先放到MQ中缓存起来，后续系统再慢慢取出并处理即可：

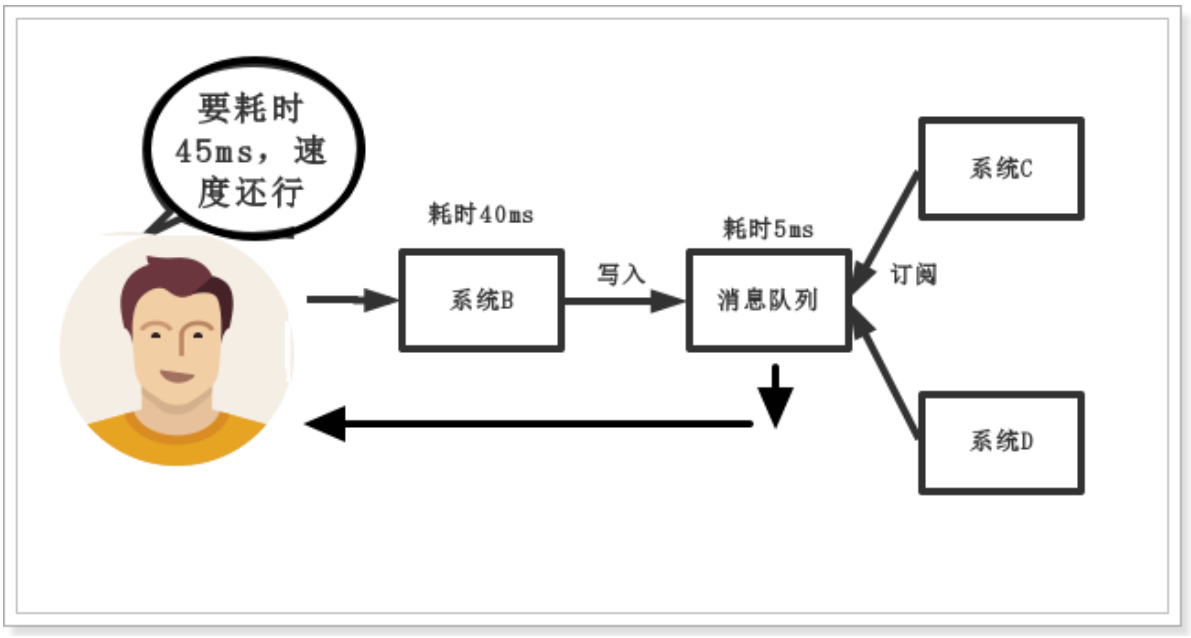


3）异步调用

如果一个业务执行中，需要调用多个其它服务，业务链路很长，同步调用的用时就时多个服务执行的总耗时，如图：



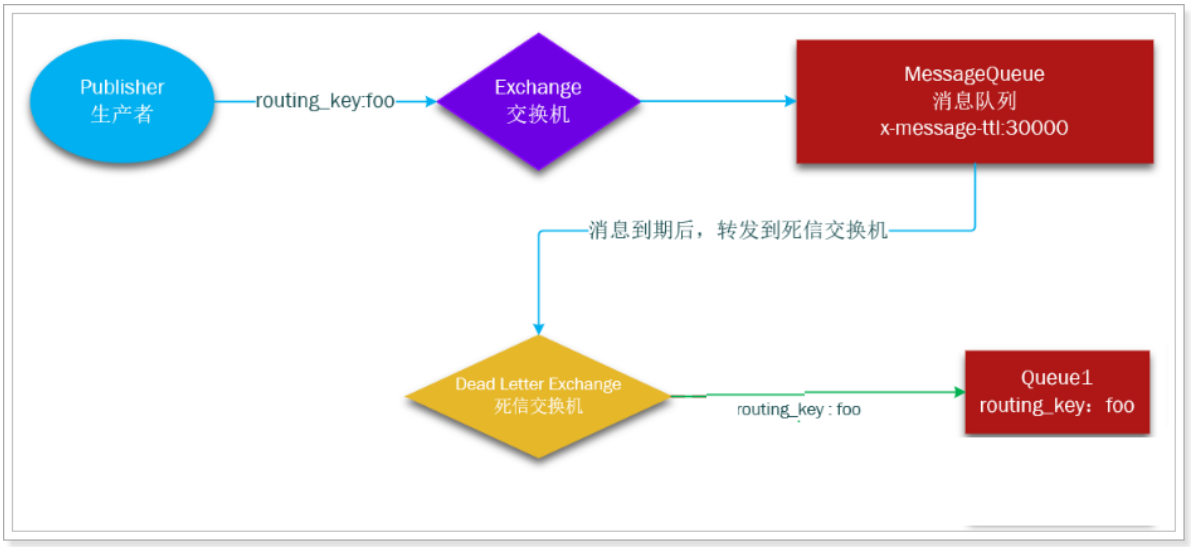
但是，我们如果再B系统执行完成后，利用MQ通知系统C和系统D去完成，直接返回结果给用户，就可以减少业务耗时。这样就把同步阻塞调用，变成了异步调用：



4）延迟队列

例如定时清理订单的需求，我们在下单完成后，如果用户超过半小时未支付，需要关闭订单。此时我们就需要发送延迟消息，即：发送消息半小时后，消费者才能收到消息，这就是延迟队列。

RabbitMQ的死信队列可以实现延迟队列效果，如图：



RocketMQ也可以实现延迟队列，但是RocketMQ的延迟队列的时间只能是固定时间间隔。

3.3.如何保证RabbitMQ的高可用

> RabbitMQ

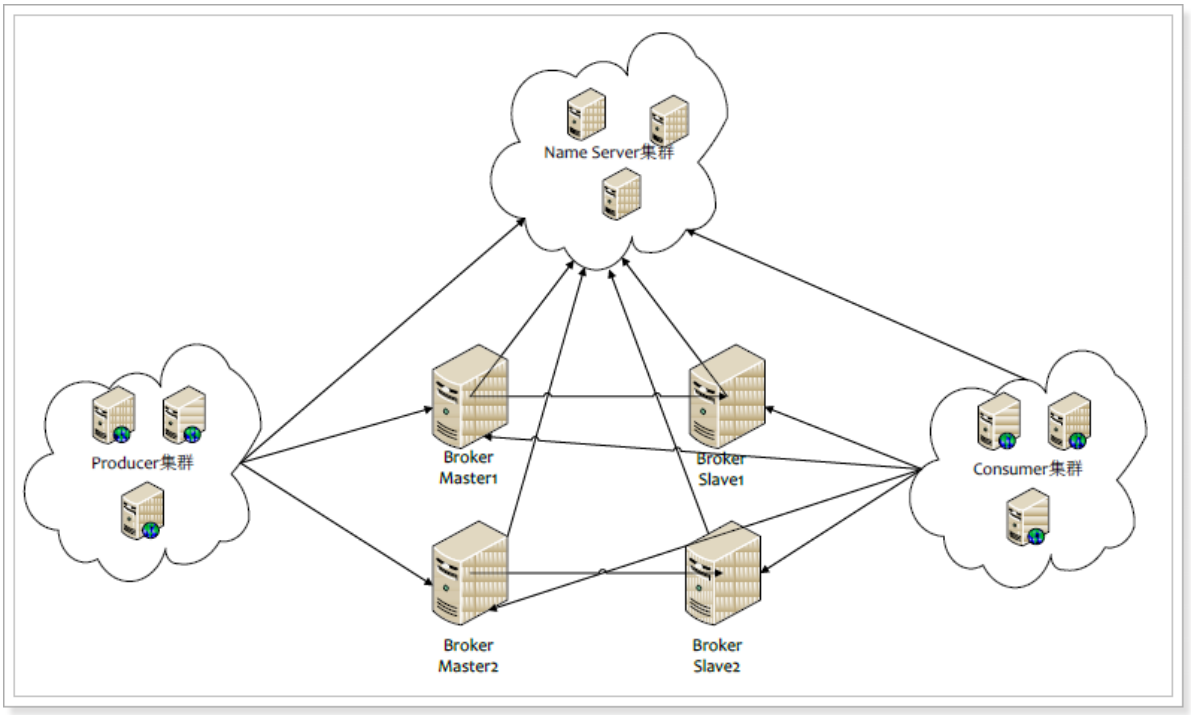
RabbitMQ底层基于Erlang语言，对分布式支持较好。并且官方也给出了搭建镜像机器的方式，可以把队列及其中的数据同步到镜像节点中，当队列所在节点故障时，镜像队列可以继续提供服务。

另外，MQ数据可以持久化，当节点恢复时，可以恢复数据。

可以参考资料：《centos搭建RabbitMQ集群》

> RocketMQ

RocketMQ中的核心组件是NameServer和Broker，这两部分都可以建立主从集群，保证高可用：



3.4.如何保证RabbitMQ的消息可靠性，防止消息丢失?

面试话术：

我们针对MQ消息丢失的几种不同情况，采用不同的应对方案：

- 生产者发送消息时可能因为网络问题导致消息丢失：
  - 利用RabbitMQ提供的publisher confirm机制，参考文档。
    - 生产者发送消息后，可以编写成功回调函数或失败超时回调函数
    - RabbitMQ接收消息成功并持久化会调用成功回调函数，通知消息的发送者
    - RabbitMQ接收消息时出现异常会调用失败回调函数，通知消息的发送者
    - 消息超时未发送成功也会调用失败回调函数
- MQ宕机导致丢失消息：
  - 消息持久化，RabbitMQ会将消息持久化到磁盘，宕机重启可以恢复消息
  - 镜像集群，主从备份，避免引宕机导致消息丢失
- 消费者丢失消息：避免引消费者异常或宕机导致消息丢失
  - 消费者的确认机制，在处理消息结束后，手动Acknowledge回执给MQ
  - MQ未接受到Acknowledge会认为消费失败，消息会保留在MQ

rabbitmq中消息消费后自动删除，不会永久保留，无法实现消息回溯.

3.5.如何防止MQ消息的重复消费?

消息重复消费产生的原因：

- 因为网络故障，导致生产者确认机制失败，生产者重发消息
- 因为网络故障，导致生产者确认机制失败，MQ重新投递消息

解决思路：业务处理时，保证处理消息接口的**幂等**性。

- 能根据具体的业务或状态来确定的，在消费端通过业务判断是否执行过，例如新增订单，看看订单ID是否已经存在
- 对于无法通过业务判断的，我们可以为每一条消息设置全局唯一id，保存到数据库消息表。消息处理前对ID进行判断即可

3.6.如何解决MQ的消息堆积问题?

面试话术：

RabbitMQ支持多消费者绑定同一队列，消息Broker会把消息轮询的发送给每一个消费者。通过同一个队列多消费者监听，实现消息的争抢，加快消息消费速度。

RabbitMQ也可以做集群，集群数据会分片效果，从而能堆积更多消息。

备选方案：也可以给单个消费者接收消息后放入队列，交给线程池去处理。

3.7.如何保证MQ消息的有序性?

某个业务发出了3条消息，要求这3条消息按照发送时的顺序执行。

- 业务对并发要求不高：
    - 保证消息发送时有序同步发送
    - 保证消息发送被同一个队列接收，MQ本身是先进先出，保证消息有序
    - 保证一个队列只有一个消费者，避免多个消费者争抢导致顺序混乱
- 业务同时对并发要求较高：
    - 满足上述第一个场景的条件
    - 可以有多个队列
    - 有时序要求的一组消息，通过hash方式分派到一个固定队列

3.8.如何利用RabbitMQ实现延迟队列

面试话术：

RabbitMQ中有一个**死信队列**设定：我们可以给一个队列**设置过期时间**，或者发送消息时给消息设置过期时间。过期的消息称为**死信**，队列会把死信转发给提前设置的**死信交换机**，而与死信交换机绑定的队列就可以拿到这些消息。

因为发送消息超过一定时间（过期）后，才会被队列拿到，这样就实现了延迟队列效果。

实现起来非常简单，不过也有一些缺陷：

- 如果延迟消息过多，可能导致MQ的消息堆积过多
- MQ消息无法删除，因此不能撤销延迟消息。

如果对上述问题有要求，可以利用Redis来实现延迟队列。

相关资料：

参考官方网站：https://www.rabbitmq.com/dlx.html

首先来看死信的概念。

> 死信

死信的英文是（Dead Letter），满足下列条件的消息被称为死信：

- 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。意思就是这个消息没有消费者需要了。
- 消息是一个过期消息（TTL到期），到期可以是**消息本身超时或者队列的TTL超时。**
- 消息的长度超过了其被投递的队列最大限制

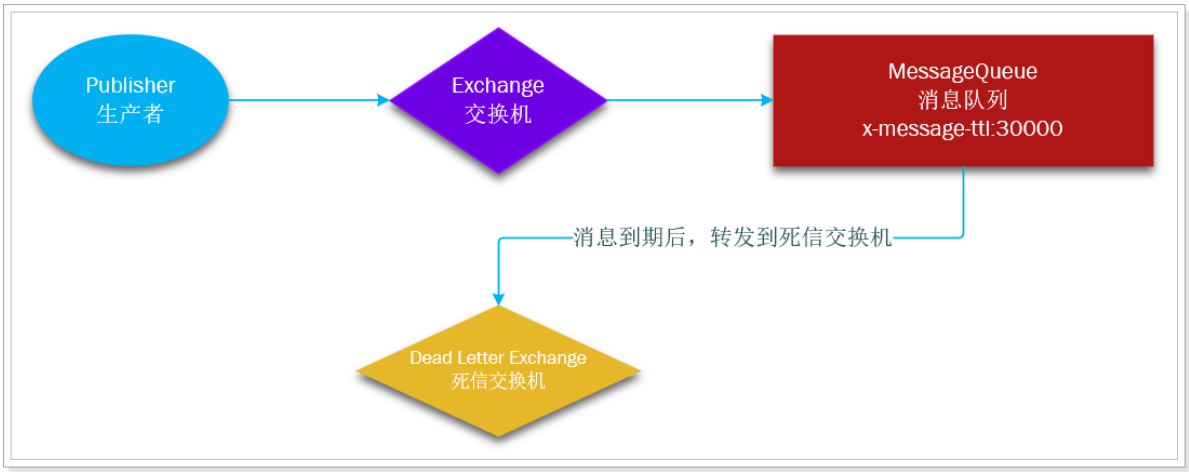
要实现延迟队列，我们肯定需要人为控制一个消息变为死信，因此我们一般采用上述的第二种方式：**让一个消息在一段时间后过期**，这种过期可以通过两种策略实现：

- 队列TTL：通过`x-message-ttl`属性给消息所在队列设置TTL（Time To Live），当队列中的消息存在时间超过TTL后就自动成为死信
- 消息TTL：消息的发送者在发送消息时，设置消息TTL属性。消息到达队列，TTL到期后成为死信、
- 如果一个消息具有TTL，同时所在队列也具备TTL，时间长度较小的会生效

由上面的概念可以知道，一个消息是不是死信，最终是由**消息所在的队列**来判断和处理的。当一个消息被判定为死信，它所在的队列会做怎样的处理呢？

队列会把**死信**交给提前指定的**死信交换机（Dead Letter Exchange）**。

如图：



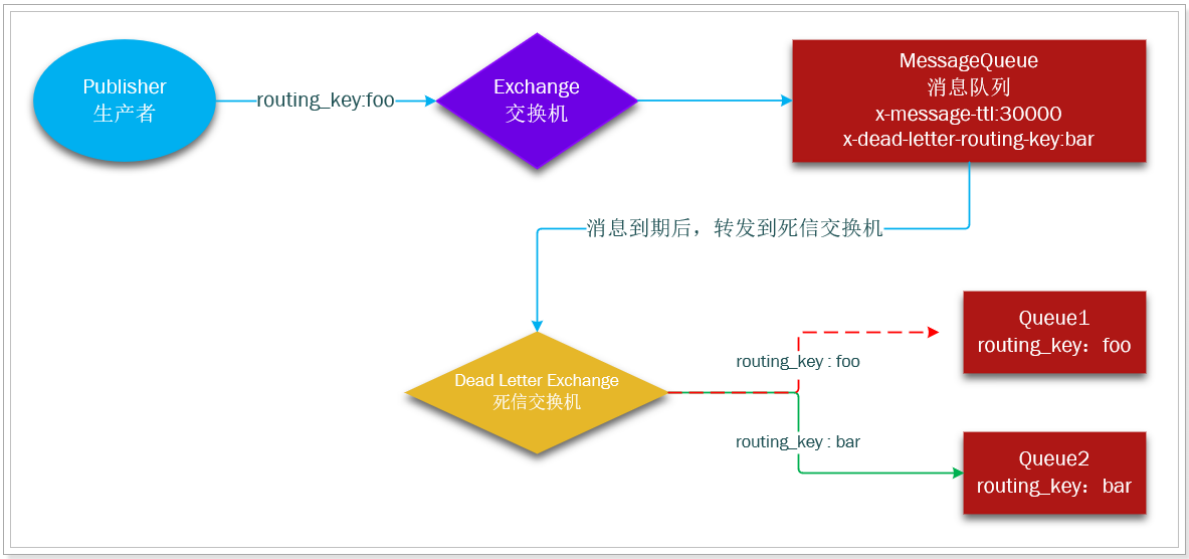
> 死信交换机Dead Letter Exchanges

死信交换机（Dead Letter Exchange）其实就是一个普通交换机，也具备以前学习的交换机的所有特征，例如可以设置交换机类型为：topic、direct等。它负责把消息根据routing key转发给绑定的队列。

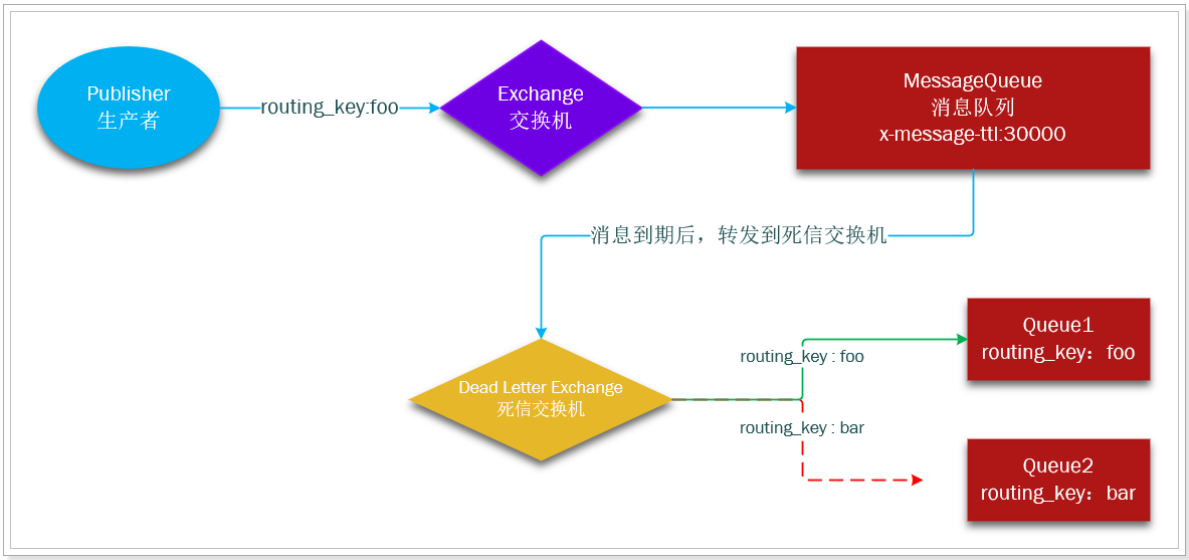
那什么样的交换机才可以叫死信交换机？需要队列在声明的时候，通过`x-dead-letter-exchange`属性指定一个交换机，被指定的交换机就是**死信交换机（Dead Letter Exchange）**。同时队列还可以指定一个`x-dead-letter-routing-key`（死信路由）作为死信的`routing_key`，死信交换机转发消息时会根据这个`routing_key`来转发消息。

死信交换机接收到消息以后，会根据消息的`routing_key`再次转发消息到绑定的队列，如果队列绑定到死信交换机时，会根据队列指定的`x-dead-letter-routing-key`来转发，如果队列没有绑定，则会根据消息来源时指定的`routing_key`来转发。

例如：现在publisher发送消息时指定`routing_key`为`foo`，队列绑定死信交换机时指定了**死信路由**为：`bar`，则死信交换机转发时，会使用`bar`作为`routing_key`，如图：



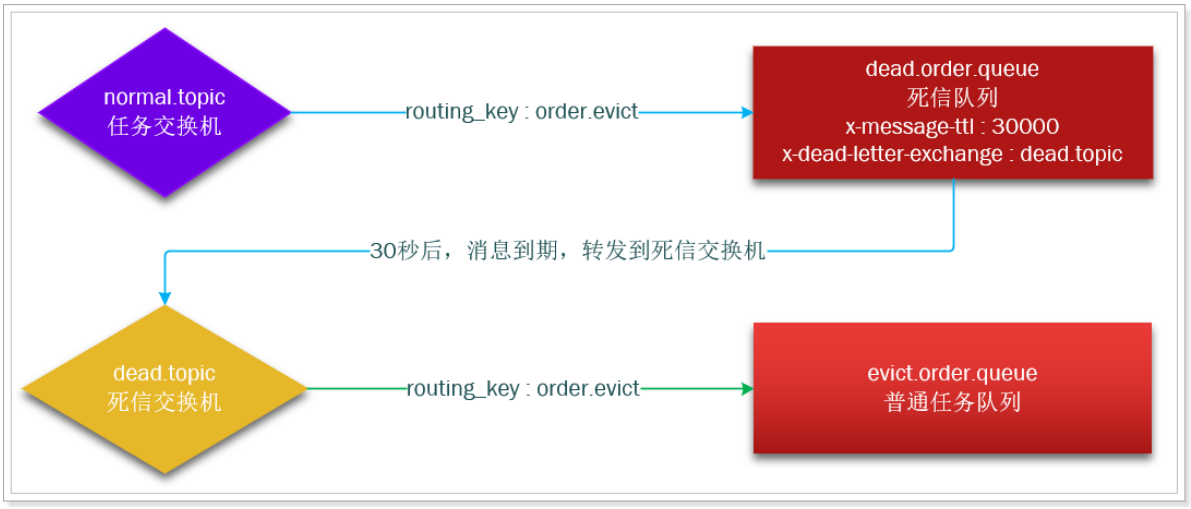
现在publisher发送消息时指定`routing_key`为`foo`，队列绑定死信交换机时没有指定**死信路由，则死信交换机转发时，会使用`foo`作为`routing_key`，如图：



现在，如果我们发送一个routingKey为foo的消息到达设置了过期时间为30秒的队列（图中的MessageQueue），30秒后消息过期，就会转发到死信交换机，然后就会发送到Queue1这个队列，我们的任务执行者监听Queue1，即可实现延迟队列了。

示例

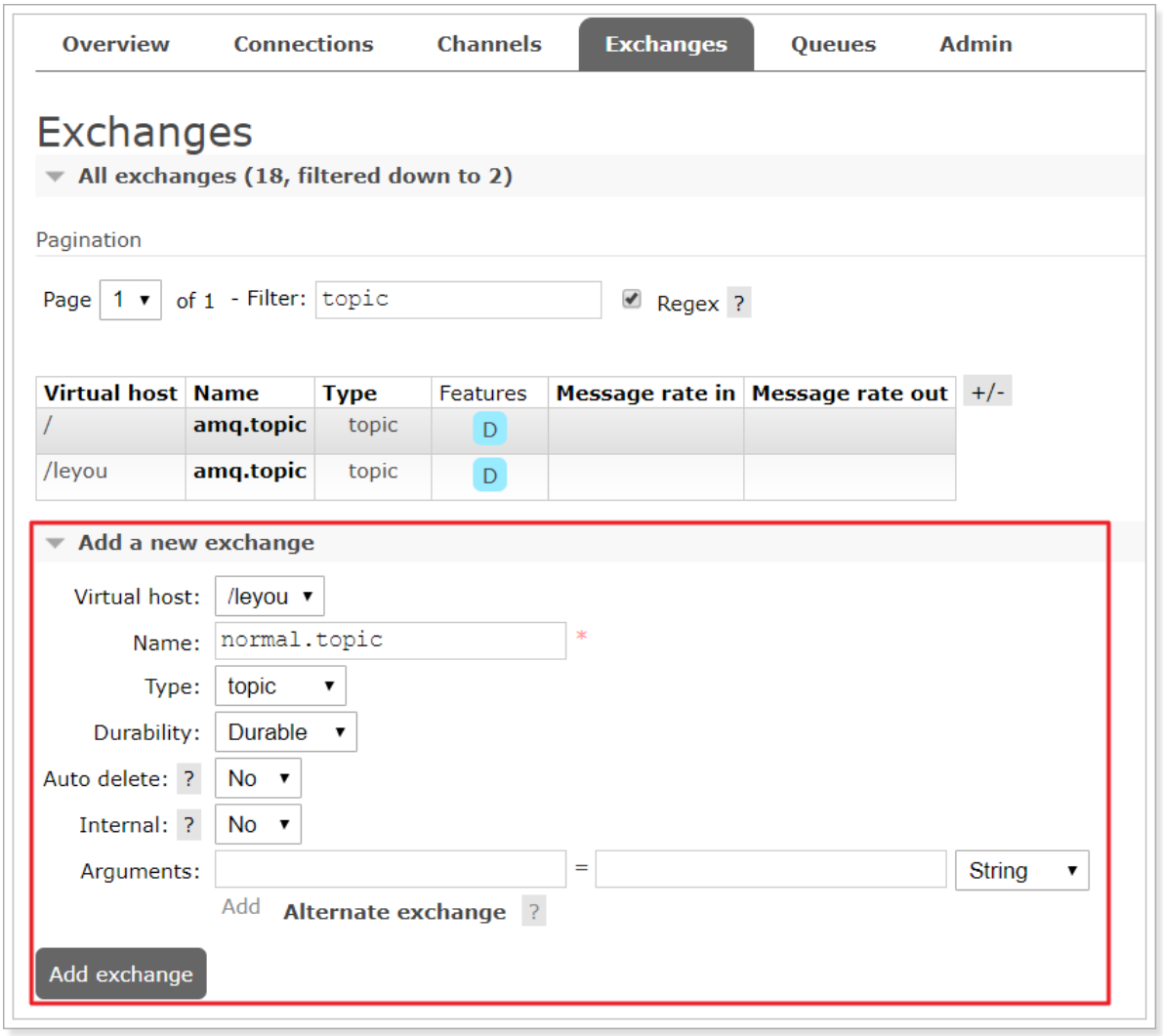
接下来，我们通过示例来展示下死信队列，如图：

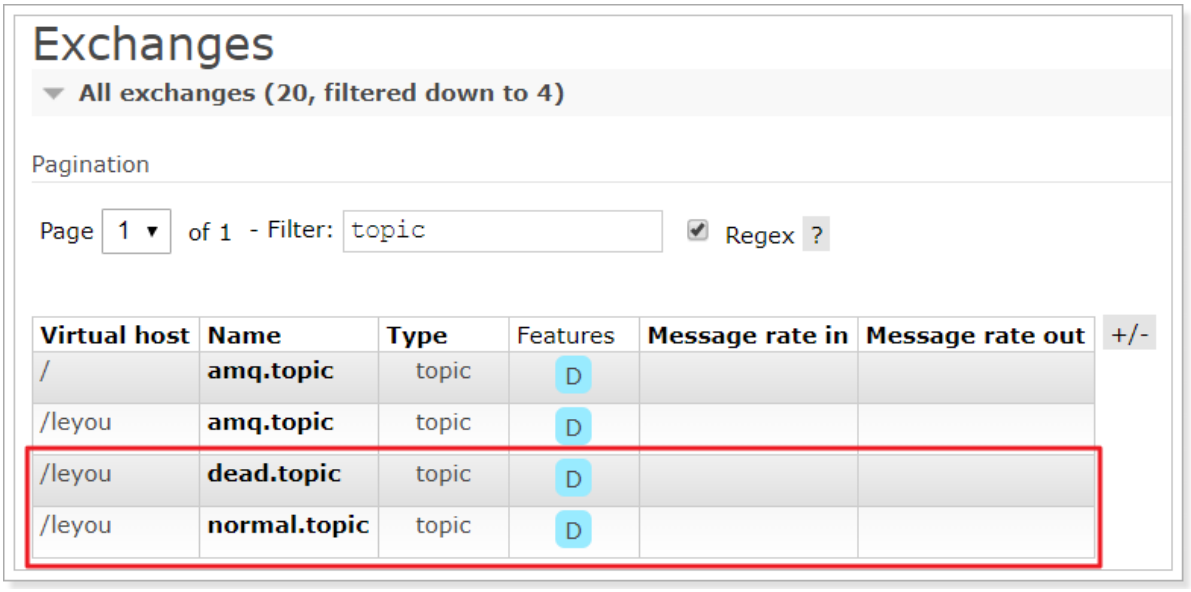


1）创建交换机

打开RabbitMQ的管理界面，然后先创建两个交换机：

- `normal.topic`：一个普通的topic类型的交换机
- `dead.topic`：一个普通topic类型的交换机，但是作为死信交换机来用



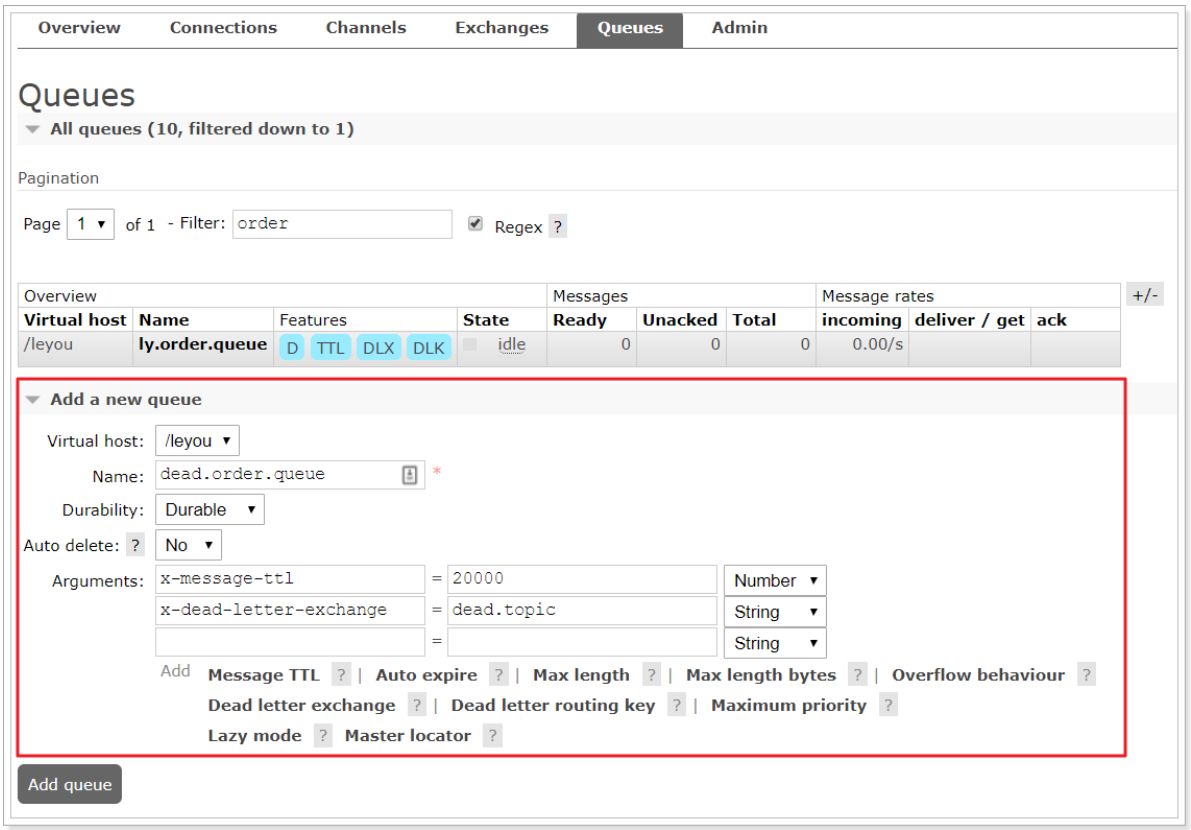


2）创建队列

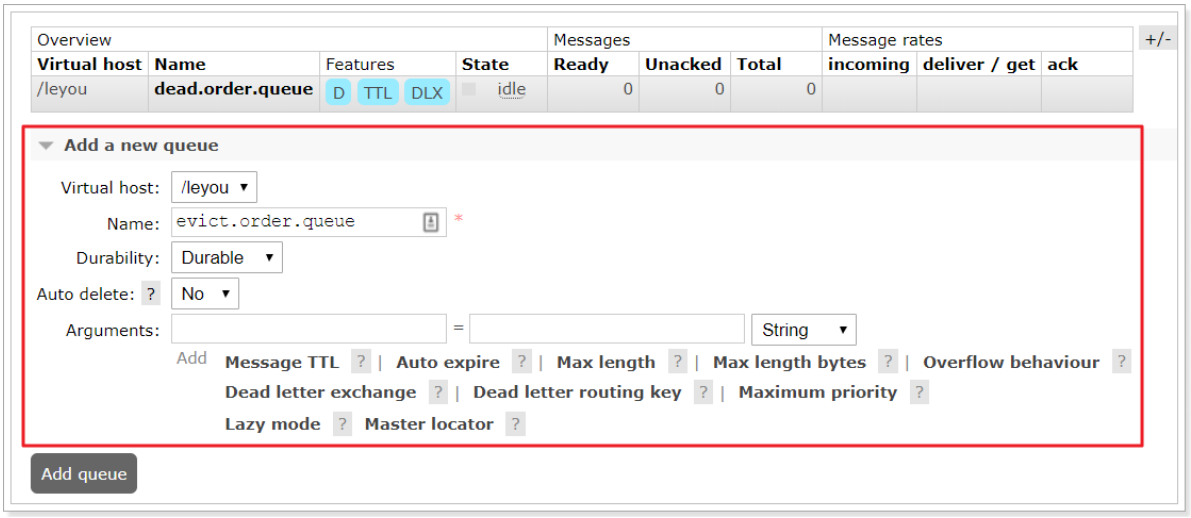
然后创建两个队列：

- `dead.order.queue`：死信队列，设置过期时间为20秒，
  - 与`normal.topic`交换机绑定，接收消息，routing_key为`order.evict`
  - 指定`x-dead-letter-exchange`为`dead.topic`这个死信交换机
  - 指定`x-message-ttl`设置消息过期时间
- `evict.order.queue`：普通任务队列，接收死信交换机转发过来的消息，将来推送给消费者
  - 与`dead.topic`交换机绑定，接收消息，routing_key为`order.evict`

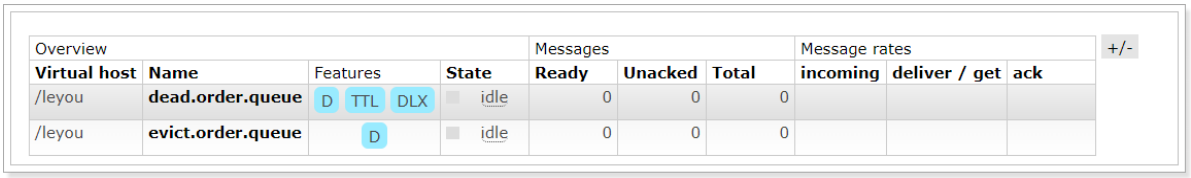
死信队列：



普通任务队列：

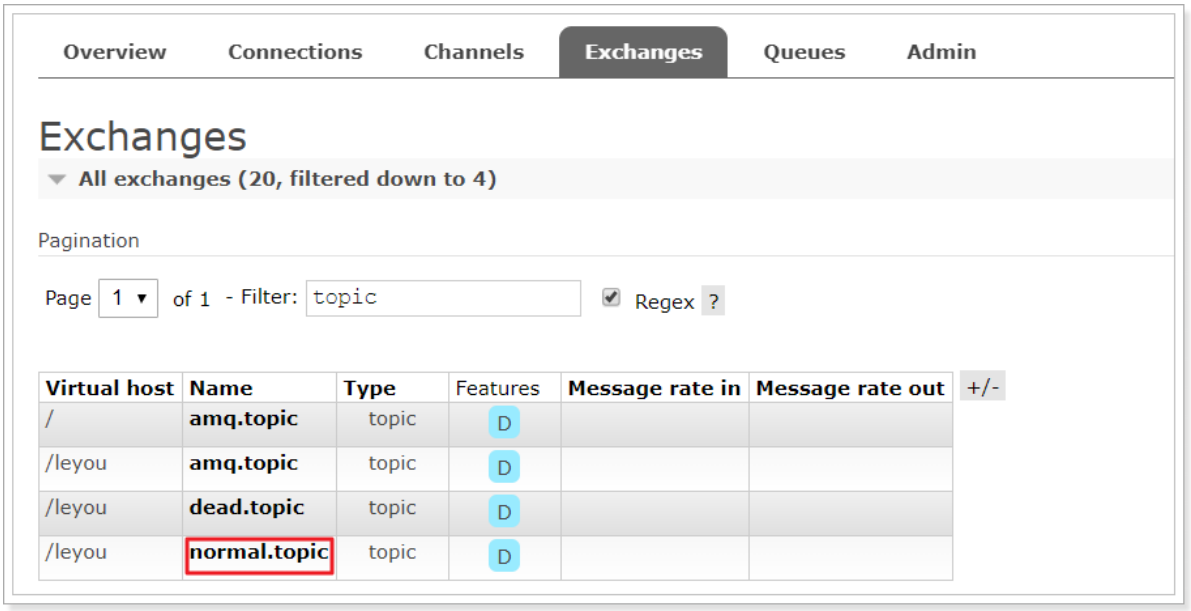


最终：

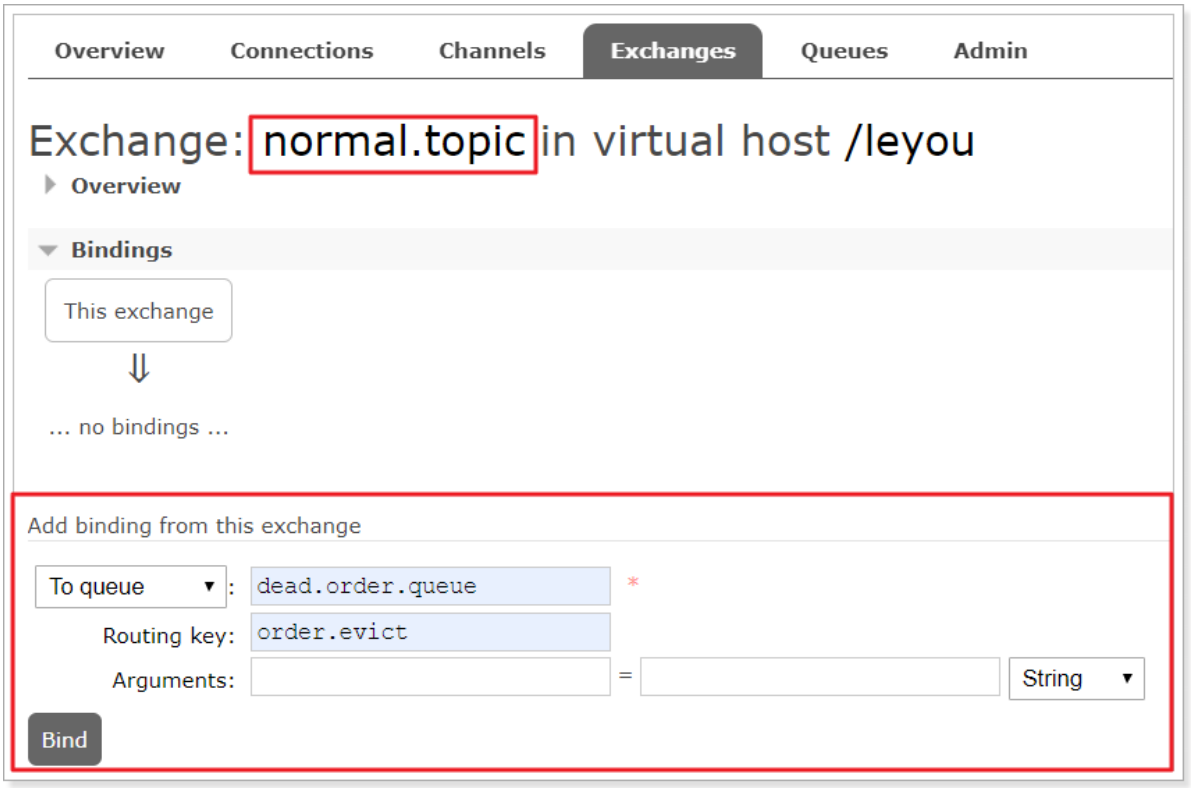


3）绑定普通队列与交换机

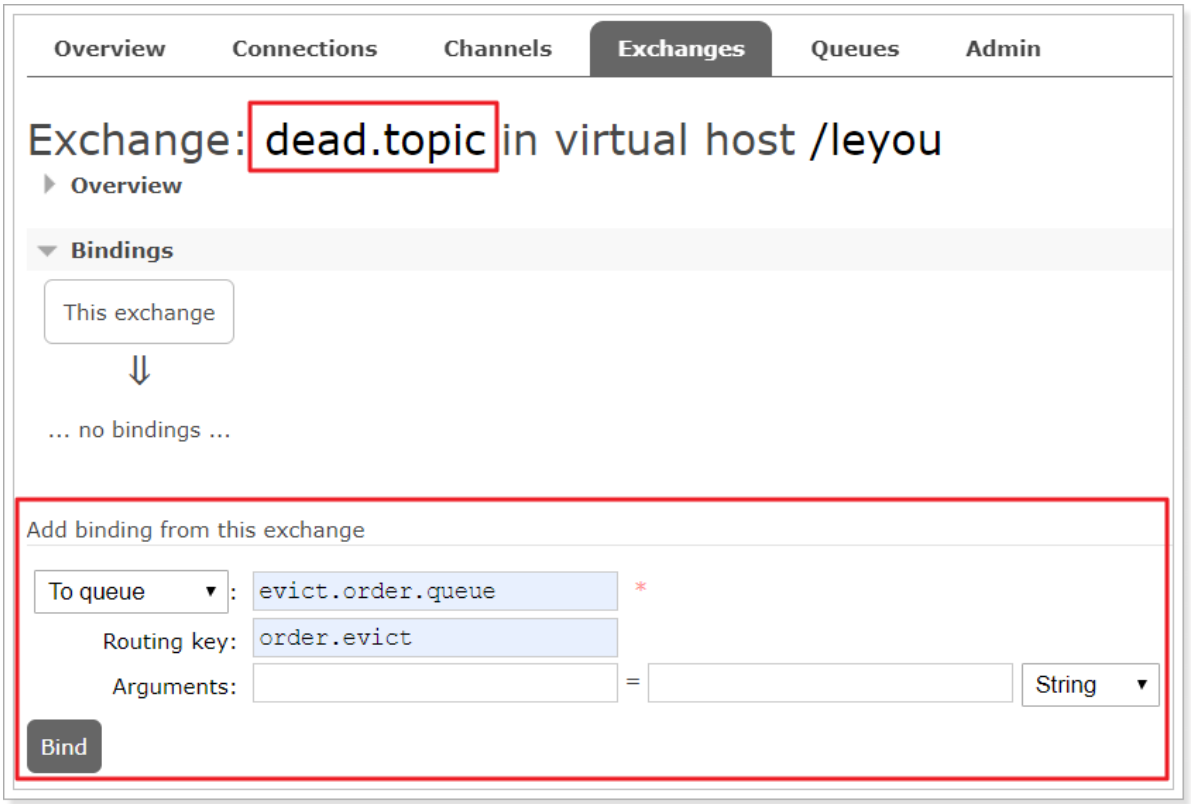
进入交换机界面，点击要绑定的交换机，例如：`normal.topic`：



在点开的界面填写要绑定的队列及routing_key：

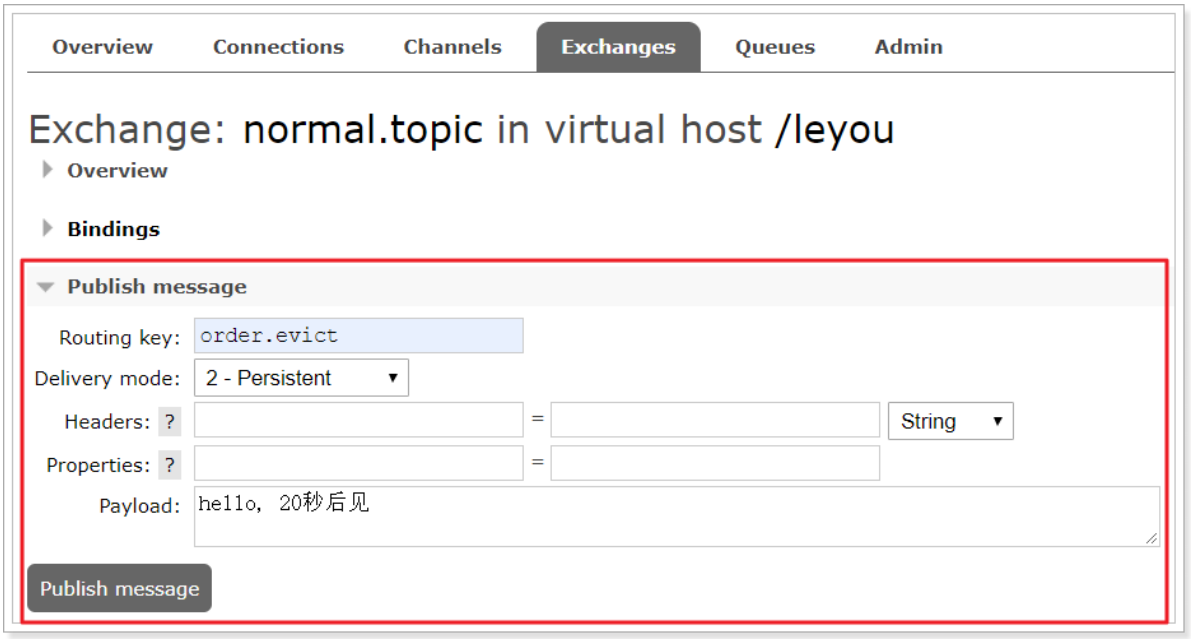


然后还要绑定`evict.order.queue`到`dead.topic`这个交换机：

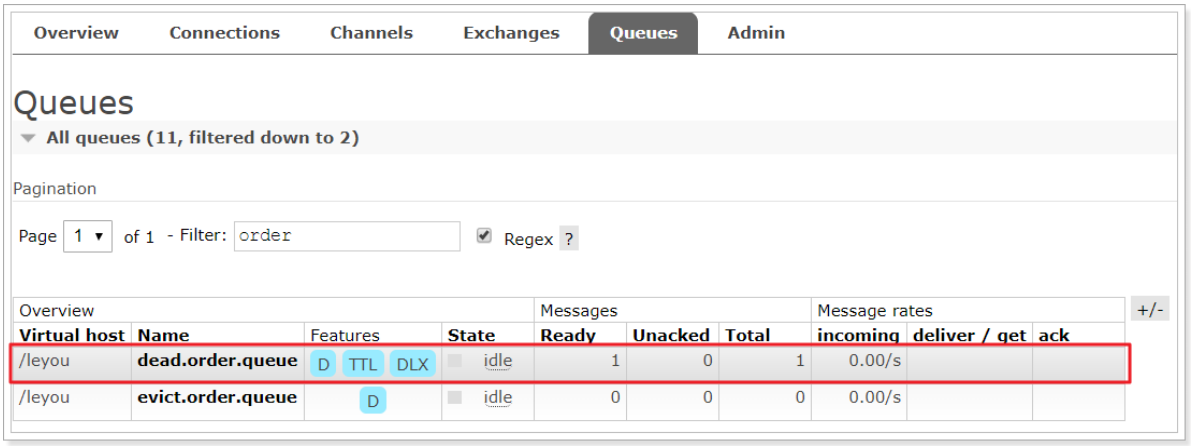


4）测试发送消息

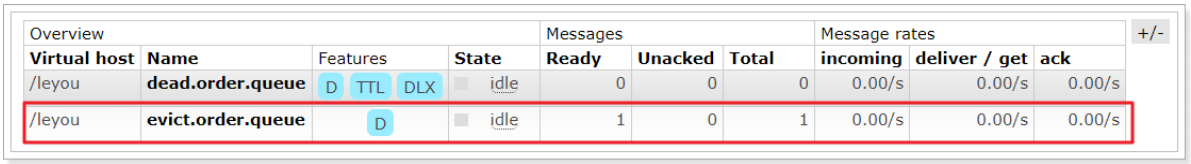
现在，向`normal.topic`交换机发送消息，指定`routing_key`为：`order.evict`



可以看到`dead.order.queue`中已经有消息了：



然后等待20秒后，看到消息到了`evict.order.queue`：



优缺点

RabbitMQ实现延迟队列的优缺点：

优点：

- 实现简单
- 可持久化
- 高可用集群
- 性能强
- 实时性好

缺点：

- 无法删除消息
- 如果是时间跨度非常大并且频率高的任务，不太适合

4.Redis相关问题

4.1.Redis与Memcache的区别?

- `redis支持更丰富的数据类型`（支持更复杂的应用场景）：Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，zset，hash等数据结构的存储。memcache支持简单的数据类型，String。
- `Redis支持数据的持久化`，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用，而Memecache把数据全部存在内存之中。
- `集群模式`：memcached没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据；但是 redis 目前是原生支持 cluster 模式的.
- `Redis使用单线程`：Memcached是多线程，非阻塞IO复用的网络模型；Redis使用单线程的多路 IO 复用模型。

对比参数	Redis	Memcached
类型	1、支持内存 2、非关系型数据库	1、支持内存 2、key-value键值对形式 3、缓存系统
数据存储类型	1、String 2、List 3、Set 4、Hash 5、Sort Set 【俗称ZSet】	1、文本型 2、二进制类型【新版增加】
查询【操作】类型	1、批量操作 2、事务支持【虽然是假的事务】 3、每个类型不同的CRUD	1、CRUD 2、少量的其他命令
附加功能	1、发布/订阅模式 2、主从分区 3、序列化支持 4、脚本支持【Lua脚本】	1、多线程服务支持
网络IO模型	1、单进程模式	2、多线程、非阻塞IO模式
事件库	自封装简易事件库AeEvent	贵族血统的LibEvent事件库
持久化支持	1、RDB 2、AOF	不支持

4.2.Redis的单线程问题

面试官：Redis采用单线程，如何保证高并发?

面试话术：

Redis快的主要原因是：

1. 完全基于内存
2. 数据结构简单，对数据操作也简单
3. 使用多路 I/O 复用模型，充分利用CPU资源

面试官：这样做的好处是什么?

面试话术：

单线程优势有下面几点：

- 代码更清晰，处理逻辑更简单
- 不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为锁而导致的性能消耗

- 不存在多进程或者多线程导致的CPU切换，充分利用CPU资源

4.2.Redis的持久化方案由哪些?

相关资料：

1）RDB 持久化

RDB持久化可以使用save或bgsave，为了不阻塞主进程业务，一般都使用bgsave，流程：

- Redis 进程会 fork 出一个子进程（与父进程内存数据一致）。
- 父进程继续处理客户端请求命令
- 由子进程将内存中的所有数据写入到一个临时的 RDB 文件中。
- 完成写入操作之后，旧的 RDB 文件会被新的 RDB 文件替换掉。

下面是一些和 RDB 持久化相关的配置：

- `save 60 10000`：如果在 60 秒内有 10000 个 key 发生改变，那就执行 RDB 持久化。
- `stop-writes-on-bgsave-error yes`：如果 Redis 执行 RDB 持久化失败（常见于操作系统内存不足），那么 Redis 将不再接受 client 写入数据的请求。
- `rdbcompression yes`：当生成 RDB 文件时，同时进行压缩。
- `dbfilename dump.rdb`：将 RDB 文件命名为 dump.rdb。
- `dir /var/lib/redis`：将 RDB 文件保存在 `/var/lib/redis` 目录下。

当然在实践中，我们通常会将 `stop-writes-on-bgsave-error` 设置为 `false`，同时让监控系统在 Redis 执行 RDB 持久化失败时发送告警，以便人工介入解决，而不是粗暴地拒绝 client 的写入请求。

RDB持久化的优点：

- RDB持久化文件小，Redis数据恢复时速度快
- 子进程不影响父进程，父进程可以持续处理客户端命令
- 子进程fork时采用copy-on-write方式，大多数情况下，没有太多的内存消耗，效率比较好。

RDB 持久化的缺点：

- 子进程fork时采用copy-on-write方式，如果Redis此时写操作较多，可能导致额外的内存占用，甚至内存溢出
- RDB文件压缩会减小文件体积，但通过时会对CPU有额外的消耗
- 如果业务场景很看重数据的持久性 (durability)，那么不应该采用 RDB 持久化。譬如说，如果 Redis 每 5 分钟执行一次 RDB 持久化，要是 Redis 意外奔溃了，那么最多会丢失 5 分钟的数据。

2）AOF 持久化

可以使用 `appendonly yes` 配置项来开启 AOF 持久化。Redis 执行 AOF 持久化时，会将接收到的写命令追加到 AOF 文件的末尾，因此 Redis 只要对 AOF 文件中的命令进行回放，就可以将数据库还原到原先的状态。

与 RDB 持久化相比，AOF 持久化的一个明显优势就是，它可以提高数据的持久性 (durability)。因为在 AOF 模式下，Redis 每次接收到 client 的写命令，就会将命令 `write()` 到 AOF 文件末尾。

然而，在 Linux 中，将数据 `write()` 到文件后，数据并不会立即刷新到磁盘，而会先暂存在 OS 的文件系统缓冲区。在合适的时机，OS 才会将缓冲区的数据刷新到磁盘（如果需要将文件内容刷新到磁盘，可以调用 `fsync()` 或 `fdatasync()` ）。

通过 `appendfsync` 配置项，可以控制 Redis 将命令同步到磁盘的频率：

- `always`：每次 Redis 将命令 `write()` 到 AOF 文件时，都会调用 `fsync()`，将命令刷新到磁盘。这可以保证最好的数据持久性，但却会给系统带来极大的开销。

- `no`：Redis 只将命令`write()`到 AOF 文件。这会让 OS 决定何时将命令刷新到磁盘。
- `everysec`：除了将命令`write()`到 AOF 文件，Redis 还会每秒执行一次`fsync()`。在实践中，推荐使用这种设置，一定程度上可以保证数据持久性，又不会明显降低 Redis 性能。

然而，AOF 持久化并不是没有缺点的：Redis 会不断将接收到的写命令追加到 AOF 文件中，导致 AOF 文件越来越大。过大的 AOF 文件会消耗磁盘空间，并且导致 Redis 重启时更加缓慢。为了解决这个问题，在适当情况下，Redis 会对 AOF 文件进行重写，去除文件中冗余的命令，以减小 AOF 文件的体积。在重写 AOF 文件期间， Redis 会启动一个子进程，由子进程负责对 AOF 文件进行重写。

可以通过下面两个配置项，控制 Redis 重写 AOF 文件的频率：

- `auto-aof-rewrite-min-size 64mb`
- `auto-aof-rewrite-percentage 100`

上面两个配置的作用：当 AOF 文件的体积大于 64MB，并且 AOF 文件的体积比上一次重写之后的体积大了至少一倍，那么 Redis 就会执行 AOF 重写。

优点：

- 持久化频率高，数据可靠性高
- 没有额外的内存或CPU消耗

缺点：

- 文件体积大
- 文件大导致服务数据恢复时效率较低

面试话术：

Redis 提供了两种数据持久化的方式，一种是 RDB，另一种是 AOF。默认情况下，Redis 使用的是 RDB 持久化。

RDB持久化文件体积较小，但是保存数据的频率一般较低，可靠性差，容易丢失数据。另外RDB写数据时会采用Fork函数拷贝主进程，可能有额外的内存消耗，文件压缩也会有额外的CPU消耗。

ROF持久化可以做到每秒钟持久化一次，可靠性高。但是持久化文件体积较大，导致数据恢复时读取文件时间较长，效率略低

4.3.Redis的集群方式有哪些?

面试话术：

Redis集群可以分为**主从集群**和**分片集群**两类。

主从集群一般一主多从，主库用来写数据，从库用来读数据。结合哨兵，可以再主库宕机时从新选主，**目的是保证Redis的高可用。**

分片集群是数据分片，我们会让多个Redis节点组成集群，并将16383个插槽分到不同的节点上。存储数据时利用对key做hash运算，得到插槽值后存储到对应的节点即可。因为存储数据面向的是插槽而非节点本身，因此可以做到集群动态伸缩。**目的是让Redis能存储更多数据。**

1）主从集群

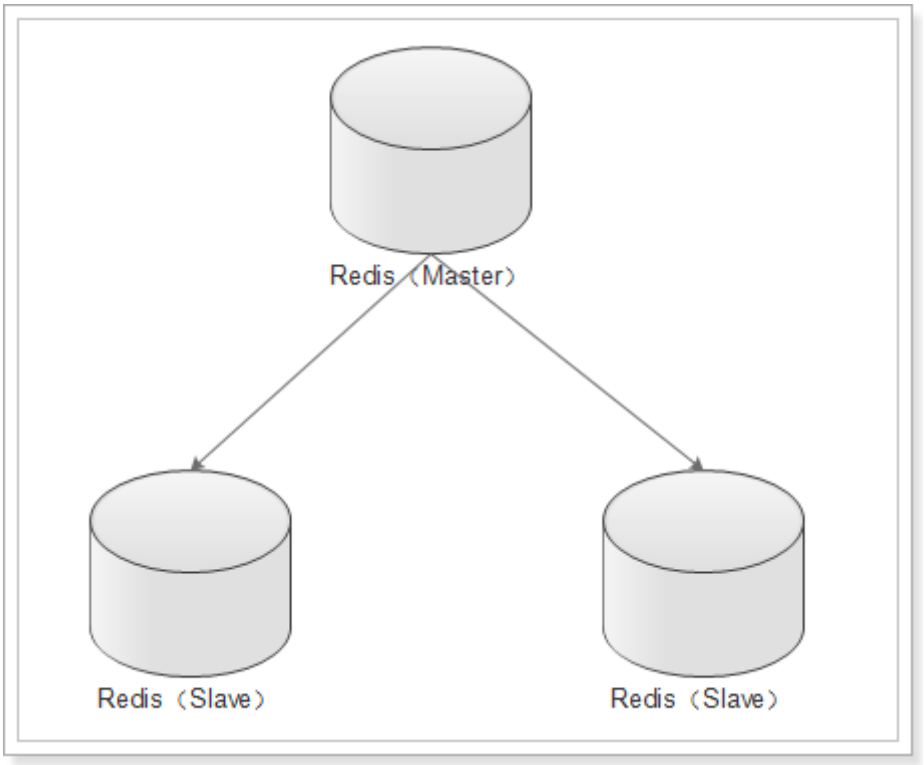
主从集群，也是读写分离集群。一般都是一主多从方式。

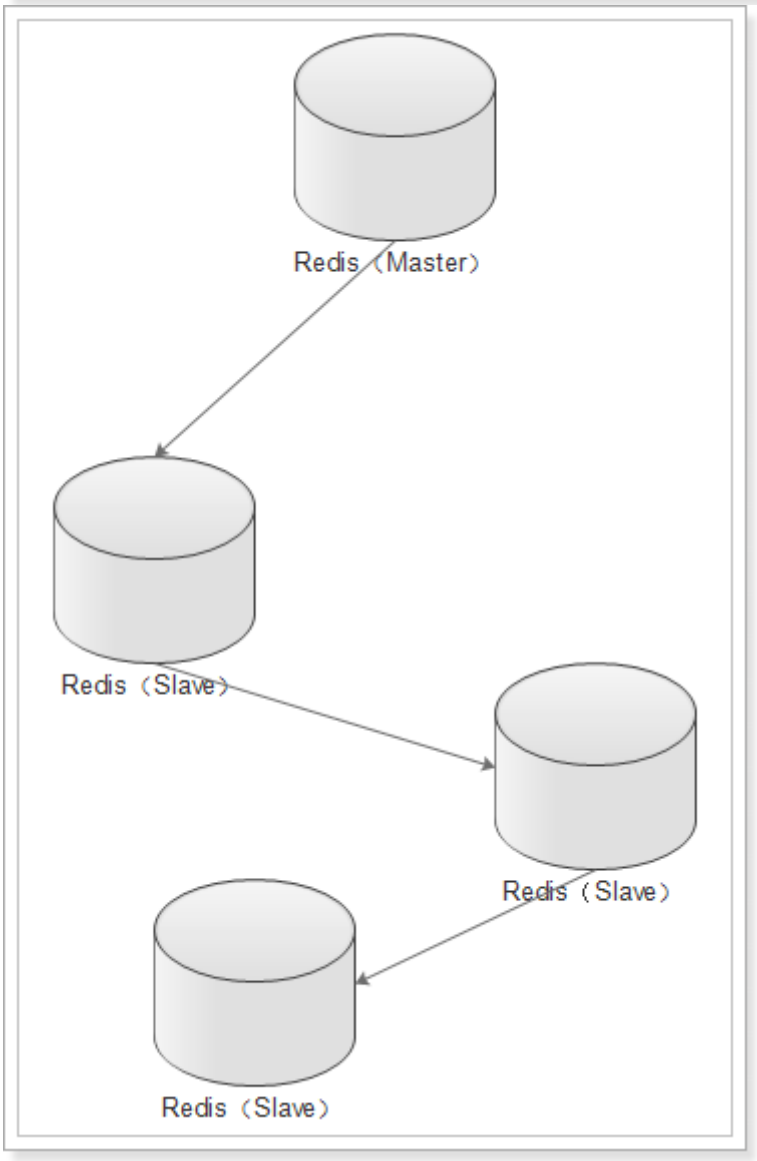
Redis 的复制（replication）功能允许用户根据一个 Redis 服务器来创建任意多个该服务器的复制品，其中被复制的服务器为主服务器（master），而通过复制创建出来的服务器复制品则为从服务器（slave）。

只要主从服务器之间的网络连接正常，主从服务器两者会具有相同的数据，主服务器就会一直将发生在自己身上的数据更新同步 给从服务器，从而一直保证主从服务器的数据相同。

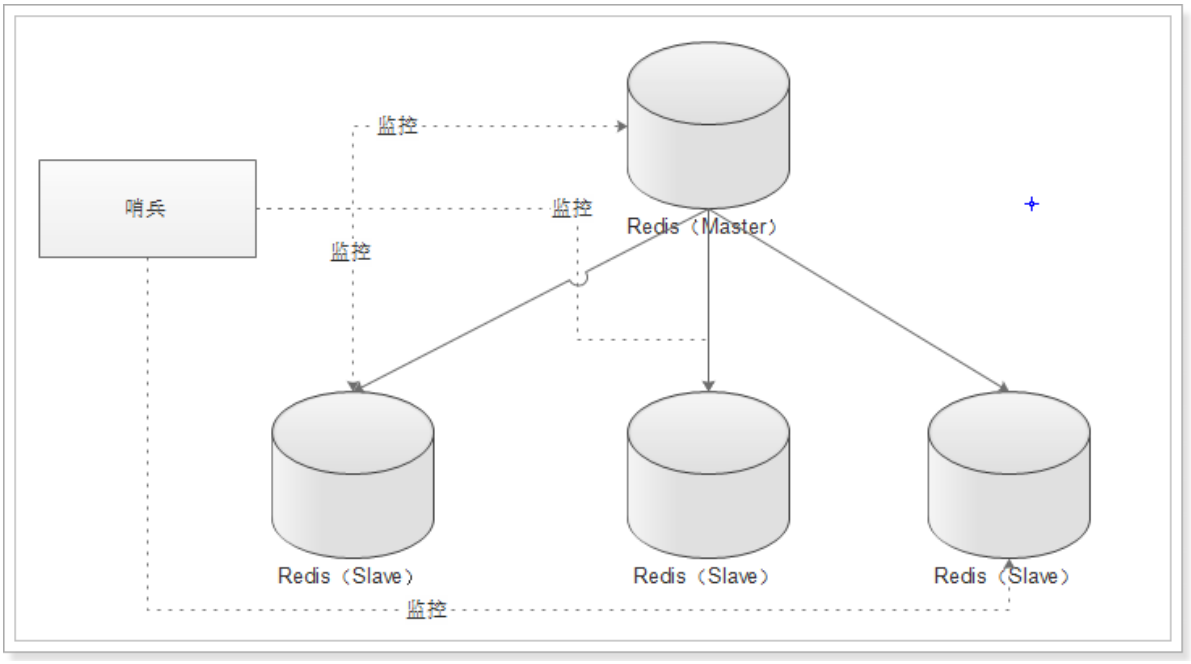
- 写数据时只能通过主节点完成
- 读数据可以从任何节点完成
- 如果配置了`哨兵节点`，当master宕机时，哨兵会从salve节点选出一个新的主。

主从集群分两种：


Redis（Master）
Redis（Slave）
Redis（Slave）

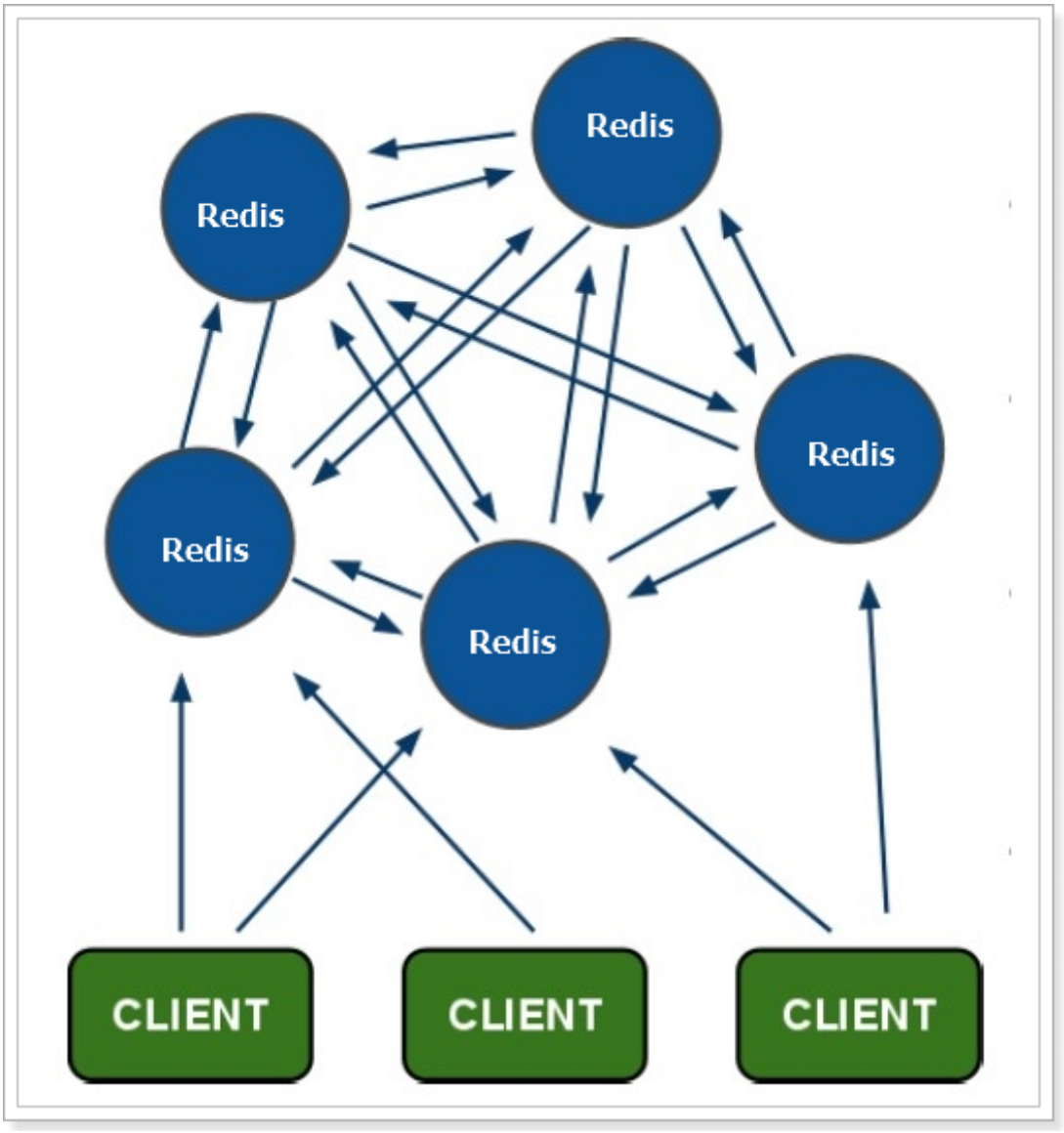

Redis（Master）
Redis（Slave）
Redis（Slave）
Redis（Slave）

带有哨兵的集群：



2）分片集群

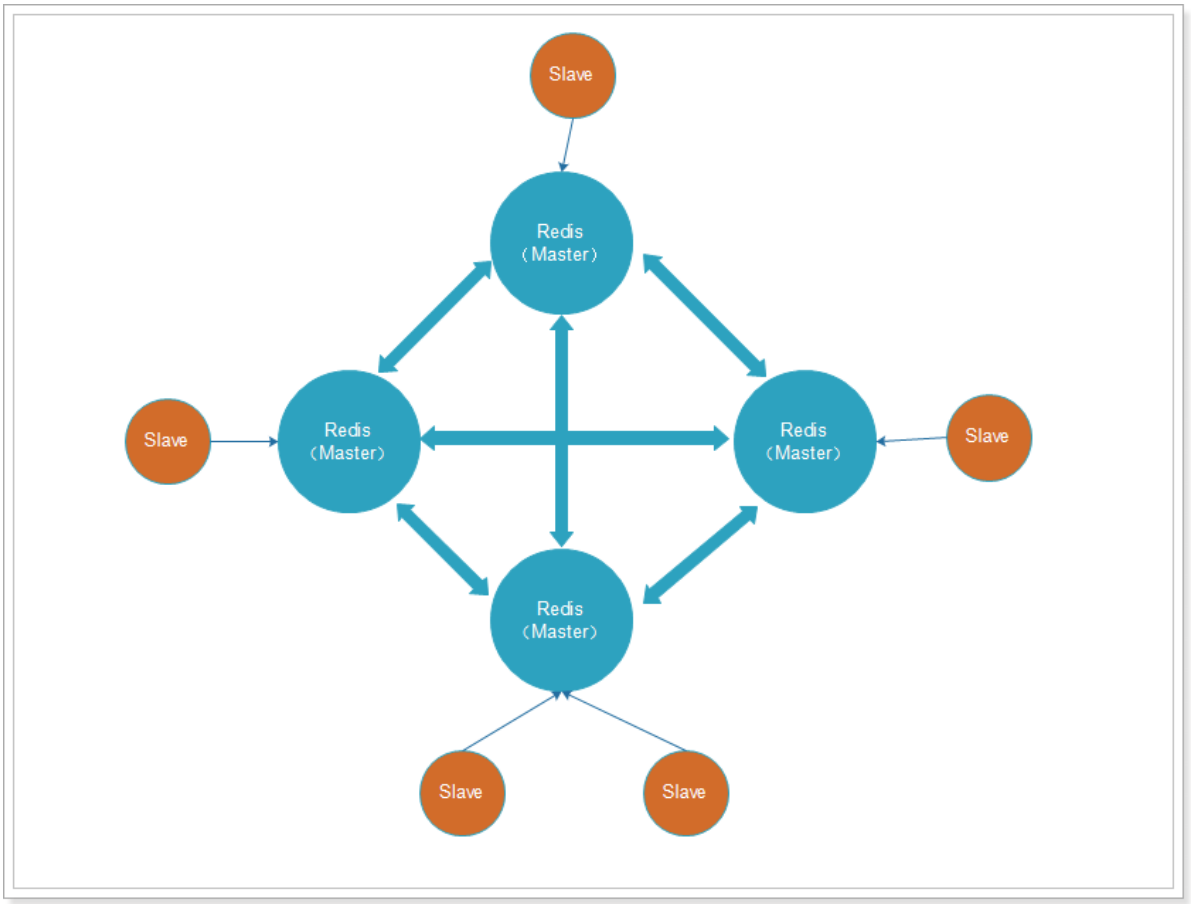
主从集群中，每个节点都要保存所有信息，容易形成木桶效应。并且当数据量较大时，单个机器无法满足需求。此时我们就要使用分片集群了。



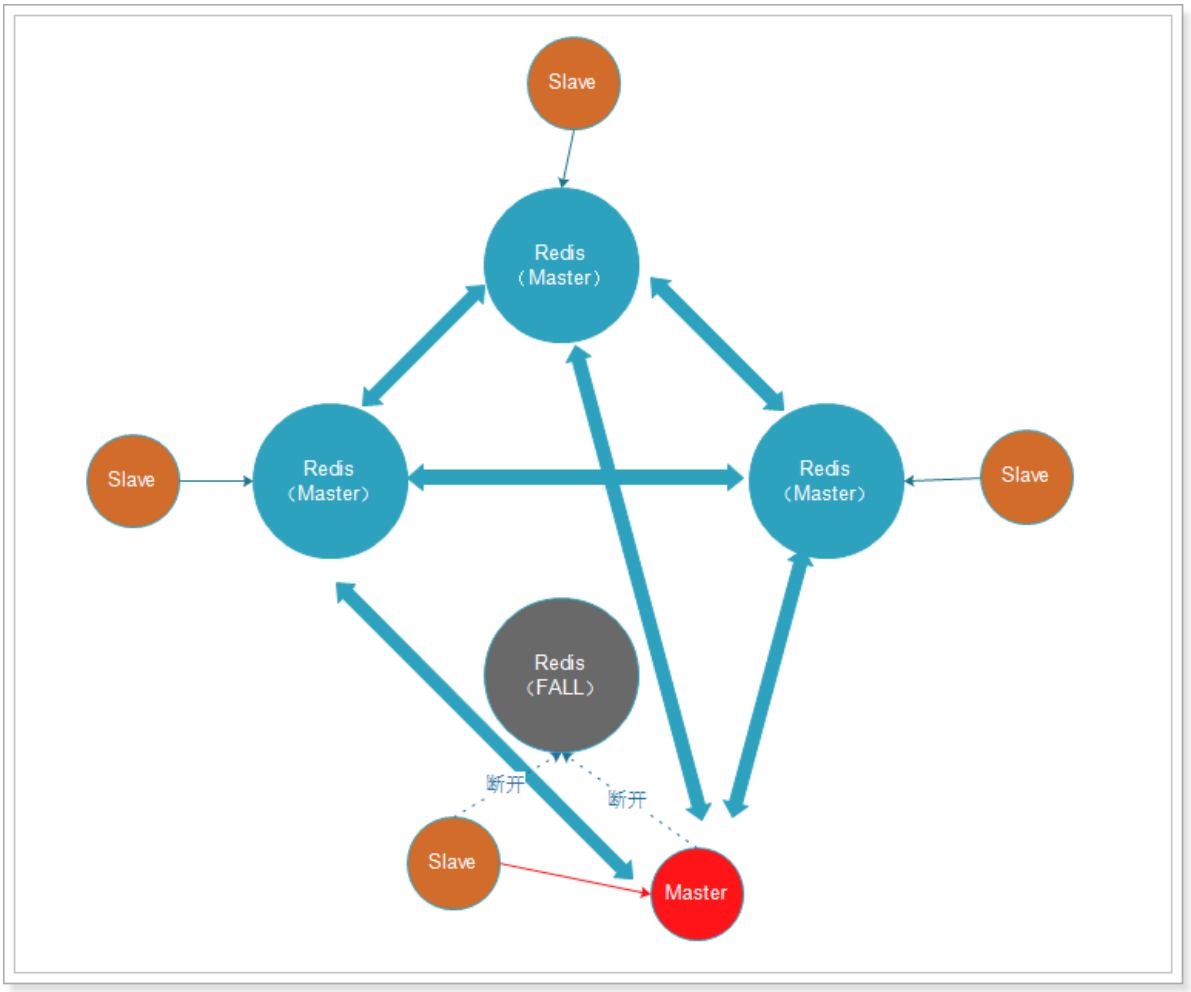
集群特征：

- 每个节点都保存不同数据
- 所有的redis节点彼此互联(PING-PONG机制),内部使用二进制协议优化传输速度和带宽.
- 节点的fail是通过集群中超过半数的节点检测失效时才生效.
- 客户端与redis节点直连,不需要中间proxy层连接集群中任何一个可用节点都可以访问到数据
- redis-cluster把所有的物理节点映射到[0-16383]slot（插槽）上，实现动态伸缩

为了保证Redis中每个节点的高可用，我们还可以给每个节点创建replication（slave节点），如图：



出现故障时，主从可以及时切换：



4.4.Redis的常用数据类型有哪些?

支持多种类型的数据结构，主要区别是value存储的数据格式不同：

- string：最基本的数据类型，二进制安全的字符串，最大512M。
- list：按照添加顺序保持顺序的字符串列表。
- set：无序的字符串集合，不存在重复的元素。
- sorted set：已排序的字符串集合。
- hash：key-value对格式

4.5.聊一下Redis事务机制

相关资料：

参考：http://redisdoc.com/topic/transaction.html

Redis事务功能是通过MULTI、EXEC、DISCARD和WATCH 四个原语实现的。Redis会将一个事务中的所有命令序列化，然后按顺序执行。但是Redis事务不支持回滚操作，命令运行出错后，正确的命令会继续执行。

- `MULTI`：用于开启一个事务，它总是返回OK。 MULTI执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个**待执行命令队列**中
- `EXEC`：按顺序执行命令队列内的所有命令。返回所有命令的返回值。事务执行过程中，Redis不会执行其它事务的命令。
- `DISCARD`：清空命令队列，并放弃执行事务， 并且客户端会从事务状态中退出
- `WATCH`：Redis的乐观锁机制，利用compare-and-set（CAS）原理，可以监控一个或多个键，一旦其中有一个键被修改，之后的事务就不会执行

使用事务时可能会遇上以下两种错误：

- 执行 EXEC 之前，入队的命令可能会出错。比如说，命令可能会产生语法错误（参数数量错误，参数名错误，等等），或者其他更严重的错误，比如内存不足（如果服务器使用 `maxmemory` 设置了最大内存限制的话）。
    - Redis 2.6.5 开始，服务器会对命令入队失败的情况进行记录，并在客户端调用 EXEC 命令时，拒绝执行并自动放弃这个事务。
- 命令可能在 EXEC 调用之后失败。举个例子，事务中的命令可能处理了错误类型的键，比如将列表命令用在了字符串键上面，诸如此类。
    - 即使事务中有某个/某些命令在执行时产生了错误， 事务中的其他命令仍然会继续执行，不会回滚。

为什么 Redis 不支持回滚（roll back）？

以下是这种做法的优点：

- Redis 命令只会因为错误的语法而失败（并且这些问题不能在入队时发现），或是命令用在了错误类型的键上面：这也就是说，从实用性的角度来说，失败的命令是由**编程错误**造成的，而这些错误应该在开发的过程中被发现，而不应该出现在生产环境中。
- 因为不需要对回滚进行支持，所以 Redis 的内部可以保持简单且快速。

鉴于没有任何机制能避免程序员自己造成的错误， 并且这类错误通常不会在生产环境中出现， 所以 Redis 选择了更简单、更快速的无回滚方式来处理事务。

面试话术：

Redis事务其实是把一系列Redis命令放入队列，然后批量执行，执行过程中不会有其它事务来打断。不过与关系型数据库的事务不同，Redis事务不支持回滚操作，事务中某个命令执行失败，其它命令依然会执行。

为了弥补不能回滚的问题，Redis会在事务入队时就检查命令，如果命令异常则会放弃整个事务。

因此，只要程序员编程是正确的，理论上说Redis会正确执行所有事务，无需回滚。

面试官：如果事务执行一半的时候Redis宕机怎么办?

Redis有持久化机制，因为可靠性问题，我们一般使用AOF持久化。事务的所有命令也会写入AOF文件，但是如果在执行EXEC命令之前，Redis已经宕机，则AOF文件中事务不完整。使用 `redis-check-aof` 程序可以移除 AOF 文件中不完整事务的信息，确保服务器可以顺利启动。

4.6.Redis的Key过期策略

参考资料：

为什么需要内存回收?

- 1、在Redis中，set指令可以指定key的过期时间，当过期时间到达以后，key就失效了；
- 2、Redis是基于内存操作的，所有的数据都是保存在内存中，一台机器的内存是有限且很宝贵的。

基于以上两点，为了保证Redis能继续提供可靠的服务，Redis需要一种机制清理掉不常用的、无效的、多余的数据，失效后的数据需要及时清理，这就需要内存回收了。

Redis的内存回收主要分为过期删除策略和内存淘汰策略两部分。

过期删除策略

删除达到过期时间的key。

- 1）定时删除

对于每一个设置了过期时间的key都会创建一个定时器，一旦到达过期时间就立即删除。该策略可以立即清除过期的数据，对内存较友好，但是缺点是占用了大量的CPU资源去处理过期的数据，会影响Redis的吞吐量和响应时间。

- 2）惰性删除

当访问一个key时，才判断该key是否过期，过期则删除。该策略能最大限度地节省CPU资源，但是对内存却十分不友好。有一种极端的情况是可能出现大量的过期key没有被再次访问，因此不会被清除，导致占用了大量的内存。

> 在计算机科学中，懒惰删除（英文：lazy deletion）指的是从一个散列表（也称哈希表）中删除元素的一种方法。在这个方法中，删除仅仅是指标记一个元素被删除，而不是整个清除它。被删除的位点在插入时被当作空元素，在搜索之时被当作已占据。

- 3）定期删除

每隔一段时间，扫描Redis中过期key字典，并清除部分过期的key。该策略是前两者的一个折中方案，还可以通过调整定时扫描的时间间隔和每次扫描的限定耗时，在不同情况下使得CPU和内存资源达到最优的平衡效果。

在Redis中，`同时使用了定期删除和惰性删除`。不过Redis定期删除采用的是随机抽取的方式删除部分Key，因此不能保证过期key 100%的删除。

Redis结合了定期删除和惰性删除，基本上能很好的处理过期数据的清理，但是实际上还是有点问题的，如果过期key较多，定期删除漏掉了一部分，而且也没有及时去查，即没有走惰性删除，那么就会有大量的过期key堆积在内存中，导致redis内存耗尽，当内存耗尽之后，有新的key到来会发生什么事呢？是直接抛弃还是其他措施呢？有什么办法可以接受更多的key?

内存淘汰策略

Redis的内存淘汰策略，是指内存达到maxmemory极限时，使用某种算法来决定清理掉哪些数据，以保证新数据的存入。

Redis的内存淘汰机制包括：

- noeviction: 当内存不足以容纳新写入数据时，新写入操作会报错。
- allkeys-lru：当内存不足以容纳新写入数据时，在键空间（`server.db[i].dict`）中，移除最近最少使用的 key（这个是最常用的）。
- allkeys-random：当内存不足以容纳新写入数据时，在键空间（`server.db[i].dict`）中，随机移除某个 key。
- volatile-lru：当内存不足以容纳新写入数据时，在设置了过期时间的键空间（`server.db[i].expires`）中，移除最近最少使用的 key。
- volatile-random：当内存不足以容纳新写入数据时，在设置了过期时间的键空间（`server.db[i].expires`）中，随机移除某个 key。
- volatile-ttl：当内存不足以容纳新写入数据时，在设置了过期时间的键空间（`server.db[i].expires`）中，有更早过期时间的 key 优先移除。

> 在配置文件中，通过maxmemory-policy可以配置要使用哪一个淘汰机制。

什么时候会进行淘汰?

Redis会在每一次处理命令的时候（processCommand函数调用freeMemoryIfNeeded）判断当前redis是否达到了内存的最大限制，如果达到限制，则使用对应的算法去处理需要删除的key。

在淘汰key时，Redis默认最常用的是LRU算法（Latest Recently Used）。Redis通过在每一个redisObject保存lru属性来保存key最近的访问时间，在实现LRU算法时直接读取key的lru属性。

具体实现时，Redis遍历每一个db，从每一个db中随机抽取一批样本key，默认是3个key，再从这3个key中，删除最近最少使用的key。

面试话术：

Redis过期策略包含定期删除和惰性删除两部分。定期删除是在Redis内部有一个定时任务，会定期删除一些过期的key。惰性删除是当用户查询某个Key时，会检查这个Key是否已经过期，如果没过期则返回用户，如果过期则删除。

但是这两个策略都无法保证过期key一定删除，漏网之鱼越来越多，还可能导致内存溢出。当发生内存不足问题时，Redis还会做内存回收。内存回收采用LRU策略，就是最近最少使用。其原理就是记录每个Key的最近使用时间，内存回收时，随机抽取一些Key，比较其使用时间，把最老的几个删除。

Redis的逻辑是：最近使用过的，很可能再次被使用

4.7.Redis在项目中的哪些地方有用到?

（1）共享session

在分布式系统下，服务会部署在不同的tomcat，因此多个tomcat的session无法共享，以前存储在session中的数据无法实现共享，可以用redis代替session，解决分布式系统间数据共享问题。

（2）数据缓存

Redis采用内存存储，读写效率较高。我们可以把数据库的访问频率高的热点数据存储到redis中，这样用户请求时优先从redis中读取，减少数据库压力，提高并发能力。

（3）异步队列

Reids在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得Redis能作为一个很好的消息队列平台来使用。而且Redis中还有pub/sub这样的专用结构，用于1对N的消息通信模式。

（4）分布式锁

Redis中的乐观锁机制，可以帮助我们实现分布式锁的效果，用于解决分布式系统下的多线程安全问题

4.8.Redis的缓存击穿、缓存雪崩、缓存穿透

1）缓存穿透

参考资料：

- 什么是缓存穿透
  - 正常情况下，我们去查询数据都是存在。那么请求去查询一条压根儿数据库中根本就不存在的数据，也就是缓存和数据库都查询不到这条数据，但是请求每次都会打到数据库上面去。这种查询不存在数据的现象我们称为**缓存穿透**。
- 穿透带来的问题
  - 试想一下，如果有黑客会对你的系统进行攻击，拿一个不存在的id 去查询数据，会产生大量的请求到数据库去查询。可能会导致你的数据库由于压力过大而宕掉。
- 解决办法
  - 缓存空值：之所以会发生穿透，就是因为缓存中没有存储这些空数据的key。从而导致每次查询都到数据库去了。那么我们就可以为这些key对应的值设置为null 丢到缓存里面去。后面再出现查询这个key 的请求的时候，直接返回null 。这样，就不用在到数据库中去走一圈了，但是别忘了设置过期时间。
  - BloomFilter（布隆过滤）：将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被 这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。在缓存之前在加一层 BloomFilter ，在查询的时候先去 BloomFilter 去查询 key 是否存在，如果不存在就直接返回，存在再走查缓存 -> 查 DB。

话术：

缓存穿透有两种解决方案：**其一**是把不存在的key设置null值到缓存中。**其二**是使用布隆过滤器，在查询缓存前先通过布隆过滤器判断key是否存在，存在再去查询缓存。

设置null值可能被恶意针对，攻击者使用大量不存在的不重复key ，那么方案一就会缓存大量不存在key数据。此时我们还可以对Key规定格式模板，然后对不存在的key做**正则规范**匹配，如果完全不符合就不用存null值到redis，而是直接返回错误。

2）缓存击穿

相关资料：

- 什么是缓存击穿?

key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：缓存被“击穿”的问题。

当这个key在失效的瞬间，redis查询失败，持续的大并发就穿破缓存，直接请求数据库，就像在一个屏障上凿开了一个洞。

- 解决方案：
    - 使用互斥锁(mutex key)：mutex，就是互斥。简单地来说，就是在缓存失效的时候（判断拿出来的值为空），不是立即去load db，而是先使用Redis的SETNX去set一个互斥key，当操作返回成功时，再进行load db的操作并回设缓存；否则，就重试整个get缓存的方法。SETNX，是「SET if Not eXists」的缩写，也就是只有不存在的时候才设置，可以利用它来实现互斥的效果。
    - 软过期：也就是逻辑过期，不使用redis提供的过期时间，而是业务层在数据中存储过期时间信息。查询时由业务程序判断是否过期，如果数据即将过期时，将缓存的时效延长，程序可以派遣一个线程去数据库中获取最新的数据，其他线程这时看到延长了的过期时间，就会继续使用旧数据，等派遣的线程获取最新数据后再更新缓存。

推荐使用互斥锁，因为软过期会有业务逻辑侵入和额外的判断。

面试话术：

缓存击穿主要担心的是某个Key过期，更新缓存时引起对数据库的突发高并发访问。因此我们可以在更新缓存时采用互斥锁控制，只允许一个线程去更新缓存，其它线程等待并重新读取缓存。例如Redis的setnx命令就能实现互斥效果。

3）缓存雪崩

相关资料：

缓存雪崩，是指在某一个时间段，缓存集中过期失效。对这批数据的访问查询，都落到了数据库上，对于数据库而言，就会产生周期性的压力波峰。

解决方案：

- 数据分类分批处理：采取不同分类数据，缓存不同周期
- 相同分类数据：采用固定时长加随机数方式设置缓存
- 热点数据缓存时间长一些，冷门数据缓存时间短一些
- 避免redis节点宕机引起雪崩，搭建主从集群，保证高可用

面试话术：

解决缓存雪崩问题的关键是让缓存Key的过期时间分散。因此我们可以把数据按照业务分类，然后设置不同过期时间。相同业务类型的key，设置固定时长加随机数。尽可能保证每个Key的过期时间都不相同。

另外，Redis宕机也可能导致缓存雪崩，因此我们还要搭建Redis主从集群及哨兵监控，保证Redis的高可用。

4.9.缓存冷热数据分离

背景资料:

Redis使用的是内存存储，当需要海量数据存储时，成本非常高。

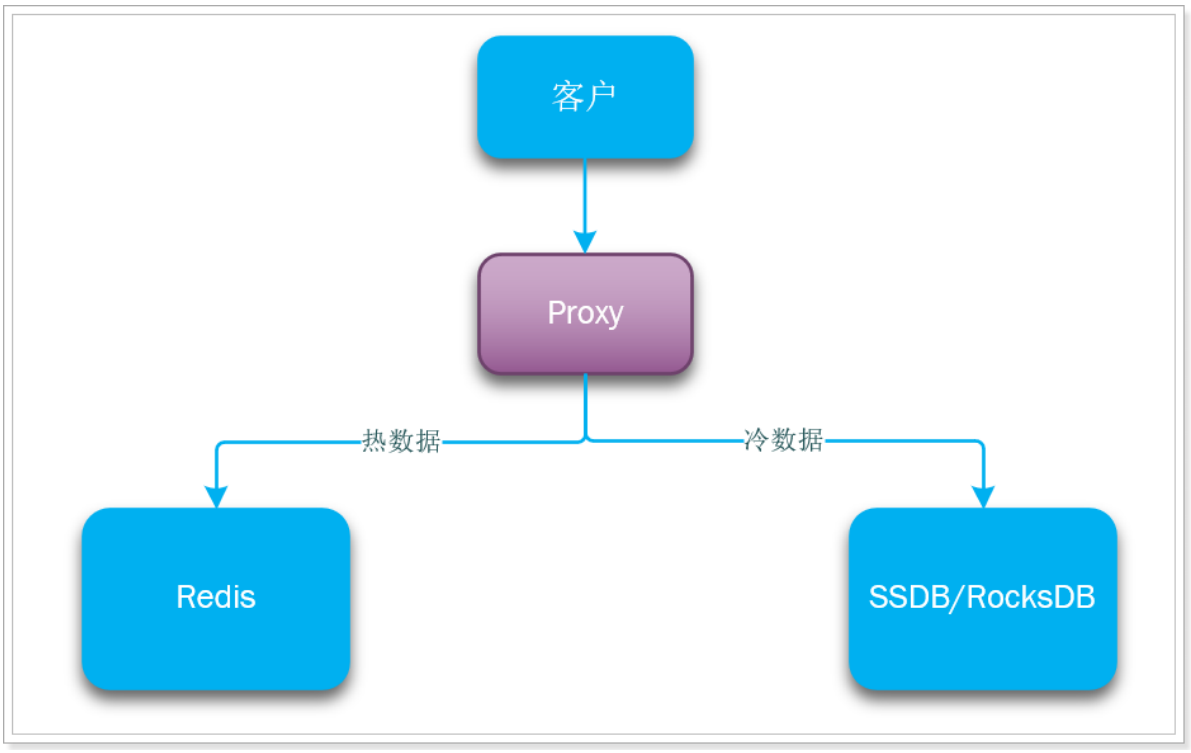
经过调研发现，当前主流DDR3内存和主流SATA SSD的单位成本价格差距大概在20倍左右，为了优化redis机器综合成本，我们考虑实现基于**热度统计 的数据分级存储**及数据在RAM/FLASH之间的动态交换，从而大幅度降低成本，达到性能与成本的高平衡。

基本思路：基于key访问次数(LFU)的热度统计算法识别出热点数据，并将热点数据保留在redis中，对于无访问/访问次数少的数据则转存到SSD上，如果SSD上的key再次变热，则重新将其加载到redis内存中。

目前流行的高性能磁盘存储，并且遵循Redis协议的方案包括:

- SSDB：http://ssdb.io/zh_cn/
- RocksDB：https://rocksdb.org.cn/

因此，我们就需要在应用程序与缓存服务之间引入代理，实现Redis和SSD之间的切换，如图:



这样的代理方案阿里云提供的就有。当然也有一些开源方案，例如：https://github.com/JingchengLi/swapdb

4.10.Redis实现分布式锁

分布式锁要满足的条件:

- 多进程互斥：同一时刻，只有一个进程可以获取锁
- 保证锁可以释放：任务结束或出现异常，锁一定要释放，避免死锁
- 阻塞锁（可选）：获取锁失败时可否重试
- 重入锁（可选）：获取锁的代码递归调用时，依然可以获取锁

1）最基本的分布式锁：

利用Redis的setnx命令，这个命令的特征时如果多次执行，只有第一次执行会成功，可以实现`互斥`的效果。但是为了保证服务宕机时也可以释放锁，需要利用expire命令给锁设置一个有效期

```
setnx lock thread-01 # 尝试获取锁
expire lock 10 # 设置有效期
```

面试官问题1：如果expire之前服务宕机怎么办？

要保证setnx和expire命令的原子性。redis的set命令可以满足：

```
set key value [NX] [EX time]
```

需要添加nx和ex的选项：

- NX：与setnx一致，第一次执行成功
- EX：设置过期时间

面试官问题2：释放锁的时候，如果自己的锁已经过期了，此时会出现安全漏洞，如何解决？

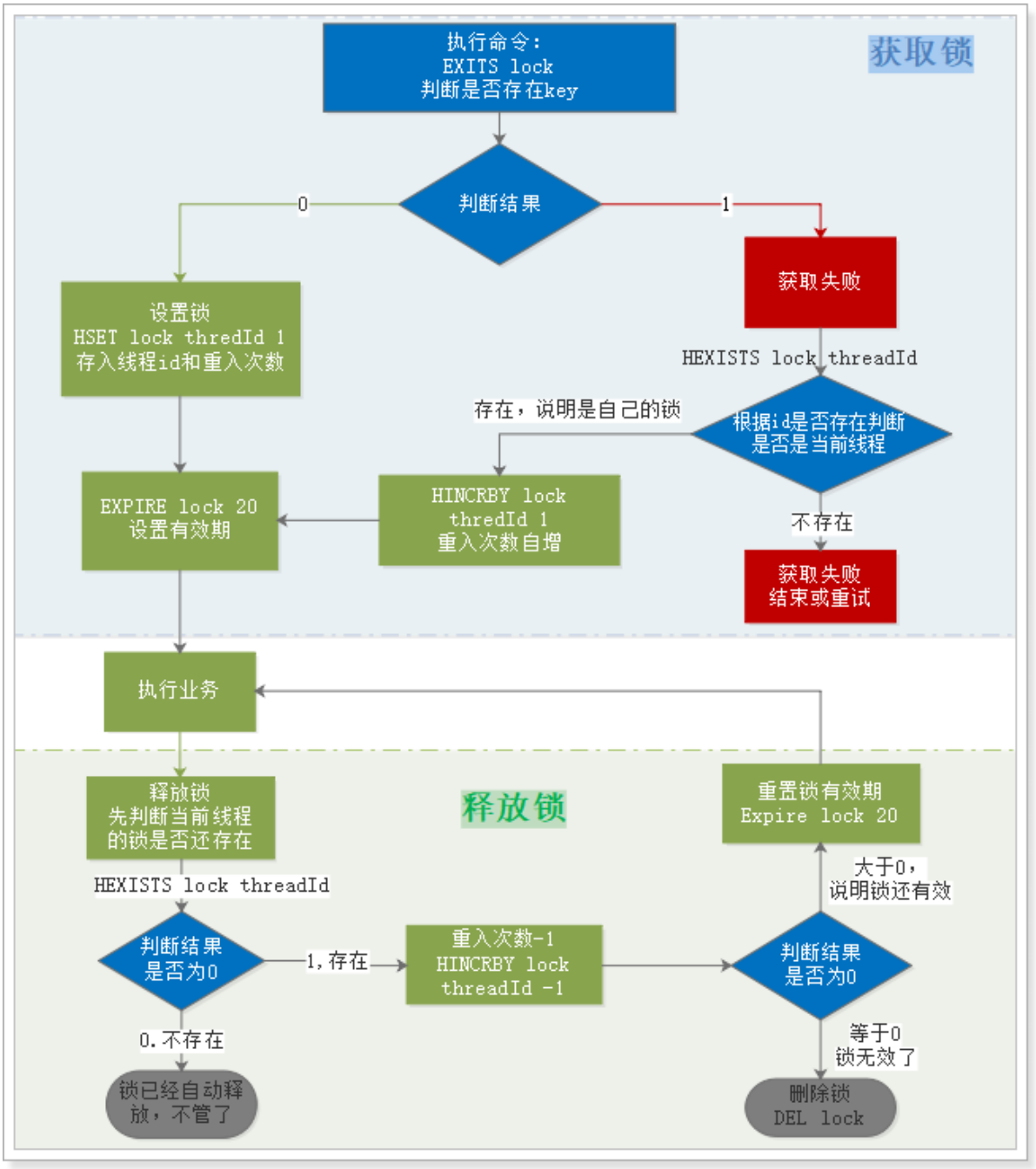
在锁中存储当前进程和线程标识，释放锁时对锁的标识判断，如果是自己的则删除，不是则放弃操作。

但是这两步操作要保证原子性，需要通过Lua脚本来实现。

```
if redis.call("get",KEYS[1]) == ARGV[1] then
    redis.call("del",KEYS[1])
end
```

2）可重入分布式锁

如果有重入的需求，则除了在锁中记录进程标识，还要记录重试次数，流程如下：



下面我们假设锁的key为“`lock`”，hashKey是当前线程的id：“`threadId`”，锁自动释放时间假设为20

获取锁的步骤：

- 1、判断lock是否存在 `EXISTS lock`
  - 存在，说明有人获取锁了，下面判断是不是自己的锁
    - 判断当前线程id作为hashKey是否存在：`HEXISTS lock threadId`
      - 不存在，说明锁已经有了，且不是自己获取的，锁获取失败，end
      - 存在，说明是自己获取的锁，重入次数+1：`HINCRBY lock threadId 1`，去到步骤3
  - 2、不存在，说明可以获取锁，`HSET key threadId 1`
  - 3、设置锁自动释放时间，`EXPIRE lock 20`

释放锁的步骤：

- 1、判断当前线程id作为hashKey是否存在：`HEXISTS lock threadId`
  - 不存在，说明锁已经失效，不用管了
  - 存在，说明锁还在，重入次数减1：`HINCRBY lock threadId -1`，获取新的重入次数

- 2、判断重入次数是否为0：
  - 为0，说明锁全部释放，删除key：`DEL lock`
  - 大于0，说明锁还在使用，重置有效时间：`EXPIRE lock 20`

对应的Lua脚本如下：

首先是获取锁：

```
local key = KEYS[1]; -- 锁的key
local threadId = ARGV[1]; -- 线程唯一标识
local releaseTime = ARGV[2]; -- 锁的自动释放时间

if(redis.call('exists', key) == 0) then -- 判断是否存在
    redis.call('hset', key, threadId, '1'); -- 不存在，获取锁
    redis.call('expire', key, releaseTime); -- 设置有效期
    return 1; -- 返回结果
end;

if(redis.call('hexists', key, threadId) == 1) then -- 锁已经存在，判断threadId是否是
自己
    redis.call('hincrby', key, threadId, '1'); -- 不存在，获取锁，重入次数+1
    redis.call('expire', key, releaseTime); -- 设置有效期
    return 1; -- 返回结果
end;
return 0; -- 代码走到这里,说明获取锁的不是自己，获取锁失败
```

然后是释放锁：

```
local key = KEYS[1]; -- 锁的key
local threadId = ARGV[1]; -- 线程唯一标识
local releaseTime = ARGV[2]; -- 锁的自动释放时间

if (redis.call('HEXISTS', key, threadId) == 0) then -- 判断当前锁是否还是被自己持有
    return nil; -- 如果已经不是自己，则直接返回
end;
local count = redis.call('HINCRBY', key, threadId, -1); -- 是自己的锁，则重入次数-1

if (count > 0) then -- 判断是否重入次数是否已经为0
    redis.call('EXPIRE', key, releaseTime); -- 大于0说明不能释放锁，重置有效期然后返回
    return nil;
else
    redis.call('DEL', key); -- 等于0说明可以释放锁，直接删除
    return nil;
end;
```

3）高可用的锁

`面试官问题`：redis分布式锁依赖与redis，如果redis宕机则锁失效。如何解决？

此时大多数同学会回答说：搭建主从集群，做数据备份。

这样就进入了陷阱，因为面试官的下一个问题就来了：

面试官问题：如果搭建主从集群做数据备份时，进程A获取锁，master还没有把数据备份到slave，master宕机，slave升级为master，此时原来锁失效，其它进程也可以获取锁，出现安全问题。如何解决?

关于这个问题，Redis官网给出了解决方案，使用RedLock思路可以解决：

> 在Redis的分布式环境中，我们假设有N个Redis master。这些节点完全互相独立，不存在主从复制或者其他集群协调机制。之前我们已经描述了在Redis单实例下怎么安全地获取和释放锁。我们确保将在每（N)个实例上使用此方法获取和释放锁。在这个样例中，我们假设有5个Redis master节点，这是一个比较合理的设置，所以我们需要在5台机器上面或者5台虚拟机上面运行这些实例，这样保证他们不会同时都宕掉。
>
> 为了取到锁，客户端应该执行以下操作:
>
> 1. 获取当前Unix时间，以毫秒为单位。
> 2. 依次尝试从N个实例，使用相同的key和随机值获取锁。在步骤2，当向Redis设置锁时,客户端应该设置一个网络连接和响应超时时间，这个超时时间应该小于锁的失效时间。例如你的锁自动失效时间为10秒，则超时时间应该在5-50毫秒之间。这样可以避免服务器端Redis已经挂掉的情况下，客户端还在死死地等待响应结果。如果服务器端没有在规定时间内响应，客户端应该尽快尝试另外一个Redis实例。
> 3. 客户端使用当前时间减去开始获取锁时间（步骤1记录的时间）就得到获取锁使用的时间。当且仅当从大多数（这里是3个节点）的Redis节点都取到锁，并且使用的时间小于锁失效时间时，锁才算获取成功。
> 4. 如果取到了锁，key的真正有效时间等于有效时间减去获取锁所使用的时间（步骤3计算的结果）。
> 5. 如果因为某些原因，获取锁失败（*没有*在至少N/2+1个Redis实例取到锁或者取锁时间已经超过了有效时间），客户端应该在所有的Redis实例上进行解锁（即便某些Redis实例根本就没有加锁成功）。

4.11.如何实现数据库与缓存数据一致?

面试话术：

实现方案有下面几种：

- 本地缓存同步：当前微服务的数据库数据与缓存数据同步，可以直接在数据库修改时加入对Redis的修改逻辑，保证一致。
- 跨服务缓存同步：服务A调用了服务B，并对查询结果缓存。服务B数据库修改，可以通过MQ通知服务A，服务A修改Redis缓存数据
- 通用方案：使用Canal框架，伪装成MySQL的salve节点，监听MySQL的binLog变化，然后修改Redis缓存数据

5.秒杀相关

5.1.锁，减库存

1）悲观锁

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java中synchronized和ReentrantLock等独占锁就是悲观锁思想的实现。

以减库存为例来说：

减库存先要查询库存，判断库存是否充足，然后再减库存。

如果我们查询库存后，判断库存是充足的，此时有人修改了库存，则我们的判断就不准确了，此时写数据就会又库存超卖的风险。

必须保证从查询开始就锁定数据，保证其它人无法操作，可以通过下列方式实现：

- 同步方法或Synchronized：适用于单点项目，分布式下会失效。
- 分布式锁：把整个减库存方法通过分布式锁锁定，不允许他人执行减库存逻辑
- 数据库锁：在执行查询语句时，在语句后面跟上 for update，则查询即会对数据加锁，其它人就无法操作了。

2）乐观锁

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和CAS算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于write_condition机制，其实都是提供的乐观锁。在Java中java.util.concurrent.atomic包下面的AtomicInteger就是使用了乐观锁的一种实现方式CAS实现的。

以减库存为例来说：

减库存先要查询库存，判断库存是否充足，然后再减库存。

如果我们查询库存后，判断库存是充足的，此时有人修改了库存，则我们的判断就不准确了，此时写数据就会又库存超卖的风险。

我们假设自己减库存时没有其它人在操作，不过执行sql时对库存检查即可。

方式1：版本号

在库存表添加version字段，每次修改数据都对version执行+1操做。

减库存步骤：

- 查询version值，例如此时version是20
- 执行减库存，在where条件中判断 version值是否等于查询到的version值：
    - UPDATE tb_stock SET stock = stock - 1, version = 21 WHERE id = 101 AND version = 20

方式2：判断库存

因为库存本身就是数值，可以用库存来做检查，代替版本号：

- 查询库存，例如值是20，需要减库存值为 2
- 减库存：UPDATE tb_stock SET stock = 18 WHERE id = 101 AND stock = 20

这种方式可能有安全漏洞，即CAS这ABA问题，比如我查询的时候是20，有人购买了一个商品，变成了19，然后又有人退货，库存恢复为20。我们认为库存没变，其实此时已经有人修改了数据了。

方式3：无符号数

库存是数字，如果我们把库存变成无符号数字，则数据库默认不能为负，如果减库存传入的值为负数，数据库直接报错，因此减库存时无需做特殊判断，直接减库存即可。

3）使用场景

乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行retry，这样反倒是降低了性能，所以一般多写的场景下用悲观锁就比较合适。

5.2.秒杀思路

秒杀问题的难点：

- 高并发，服务端tomcat并发能力有限
- 减库存多线程执行容易超卖
- 下单业务流程、业务链路较长，耗时较久，服务QPS低
- 秒杀页面请求量大
- 秒杀机器人防范

具体实现：

5.2.1.前端优化

前端是直接与用户交互的地方，并发最高，一般有下面手段去处理：

1）页面静态化

将秒杀商品页面静态化处理，少量动态数据通过ajax异步加载，访问商品页面无需去数据库查询商品信息，大大提高页面加载的速度。

2）CDN服务

静态化可以让页面响应速度增加，但是如果我们的静态资源服务器压力过大，也可以考虑购买CDN服务，将静态资源部署到CDN服务，一方面提高响应速度，另一方面减轻对服务端压力

3）秒杀限流

秒杀按钮点击时，不立即向服务端发送请求，而是要求回答验证问题答案，回答结束才发送秒杀请求。好处有2点：

- 用户回答问题耗时不同，把用户发送请求分散到不同事件段
- 限制秒杀机器人或爬虫的恶意访问

4）动态秒杀按钮

为了避免秒杀开始前有人提前获取秒杀地址并编写秒杀机器人，我们可以把秒杀按钮利用JS绑定，秒杀开始前对应的JS文件内容设置为点击后禁止发送请求。

秒杀开始时，我们再修改对应的JS文件内容，填写真实发送请求地址，这样开始前不会有人知道秒杀的地址信息。

5）避免重复连续点击

点击秒杀后，按钮禁用，一定时间后开启使用

5.2.2.网关

如果采用Nginx作为网关，则可以再Nginx中对用户请求限流，只放行部分用户请求到达微服务群。

5.2.3.微服务

1）限流

RateLimiter是guava提供的基于令牌桶算法的限流实现类，通过调整生成token的速率来限制用户频繁访问秒杀页面，从而达到防止超大流量冲垮系统。（令牌桶算法的原理是系统会以一个恒定的速度往桶里放入令牌，而如果请求需要被处理，则需要先从桶里获取一个令牌，当桶里没有令牌可取时，则拒绝服务。

2）预减库存

请求到达服务端，并发依然很高，数据库直接处理肯定难以接受。

我们可以把秒杀的商品库存存入Redis，利用Redis中的库存判断秒杀商品是否充足，再Redis中完成抢购资格判断、减库存行为。

但是，尽管redis单线程运行，执行Redis的Java代码依然有线程安全风险，所以为了保证redis中减库存判断的安全性，这里推荐使用Lua脚本编写相关逻辑，保证代码执行的原子性。

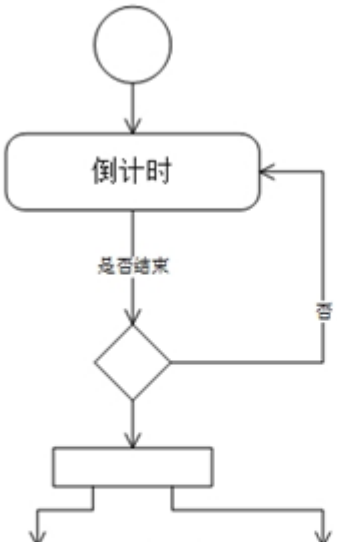
3）流量削峰，异步写数据

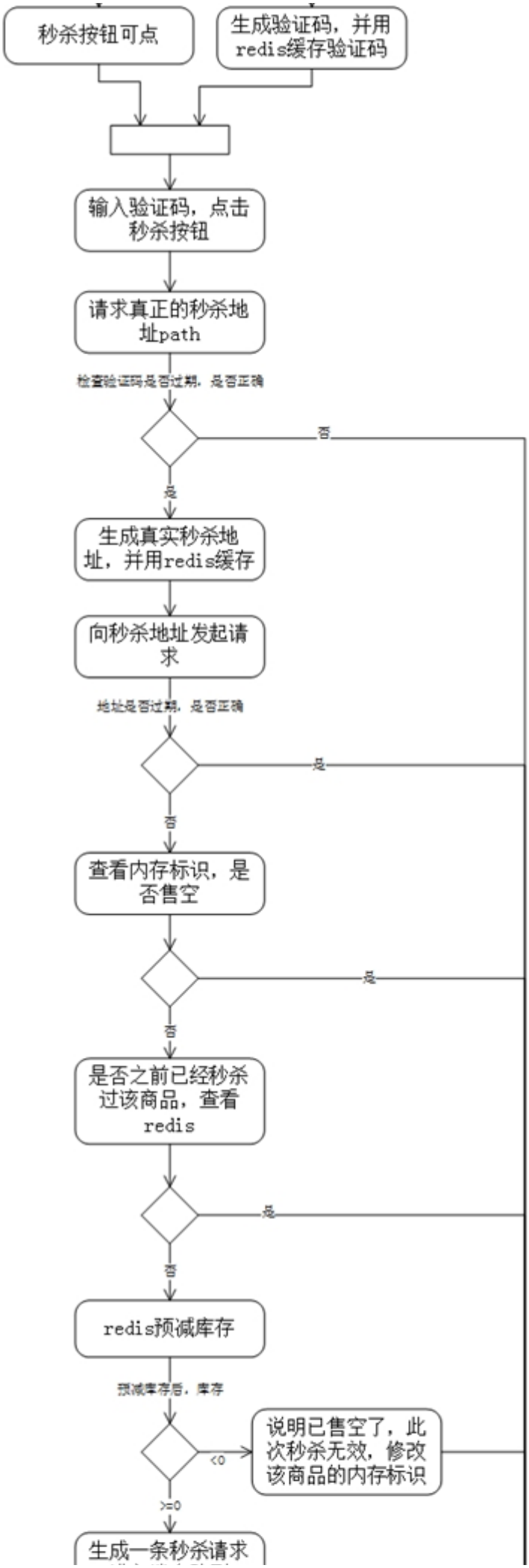
经过Redis的判断处理，单个商品放行的请求数量基本就是库存剩余量，请求大大减少，但是如果参与秒杀商品较多，用户并发依然很高，数据库可能依然难以处理，所以还需要把下的业务异步执行，实现流量削峰。

用户在redis中获取下单资格后，不要去执行下单逻辑，而是把用户及资格信息发送到MQ中，然后就返回用户抢购成功的结果。

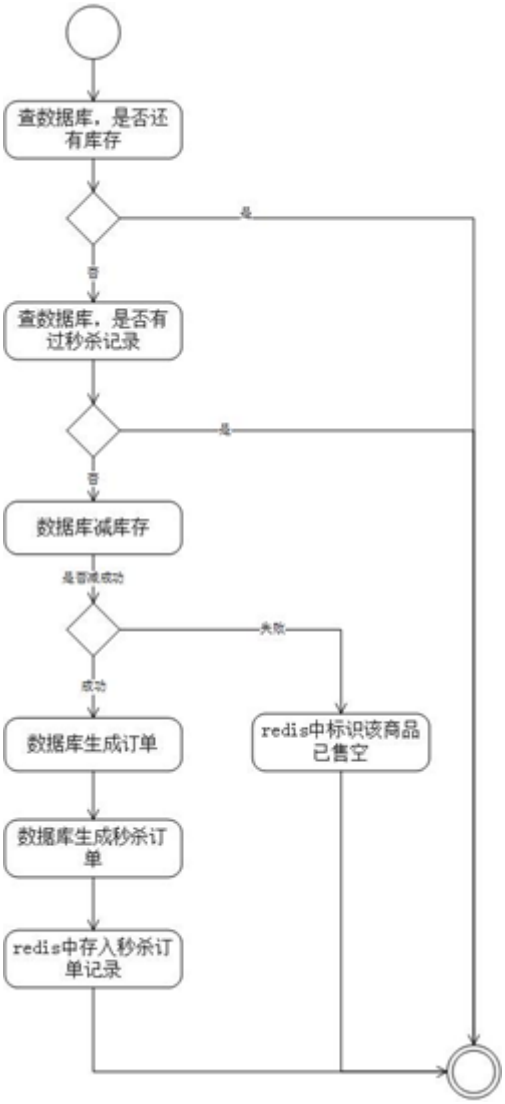
此时服务端下单的业务监听RabbitMQ，逐个处理MQ中的下单消息，利用MQ来缓存高并发的流量。变同步写数据为异步写数据，大大缩短业务链路，提高并发。

流程图：




秒杀按钮可点
生成验证码，并用redis缓存验证码
输入验证码，点击秒杀按钮
请求真正的秒杀地址path
检查验证码是否过期，是否正确
否
是
生成真实秒杀地址，并用redis缓存
向秒杀地址发起请求
地址是否过期，是否正确
是
否
查看内存标识，是否售空
是
否
是否之前已经秒杀过该商品，查看redis
是
否
redis预减库存
预减库存后，库存
<0
说明已售空了，此次秒杀无效，修改该商品的内存标识
>=0
生成一条秒杀请求

监听到MQ后的处理逻辑，关键时如何防止库存超卖，这一点我们在上面的5.1中已经讲过，不再赘述。



秒杀项目相关资料：https://github.com/qiurunze123/miaosha