## 7.29你们的系统使用了哪种服务框架？为什么要这样技术选型？

**优秀作业1：**

1.服务框架当时准备从dubbo和spring cloud两个中选型，最终选定springcloud.

2.选型对比维度：

1）整体解决方案 Dubbo没有自带的配置中心，注册中心基于zookeeper springcloud自 身相对比较完善，有注册中心、配置中心、负载均衡、服务治理 ；

2）底层调用 Dubbo基于rpc，速度快 springcloud基于http，访问相对dubbo慢 ；

3）技术前沿性（趋势） Dubbo已经比较成熟 springcloud技术相对前缘 ；

4）开发语言及开放度 都是Java，开源 ；

5）社区活跃度 Dubbo官方已没有维护，目前一开始重新维护，但基本上能碰到的问题，网上能找到解决方案。 springcloud英文的资料比较少。虽然可能会趟坑，但是业务简单，基本不会遇到什么大问题。

基于自身业务场景，主要只是展示功能，所以速度慢点，影响不大。

优先考虑整体解决方案和技术趋势，最终选择springcloud,虽然可能会趟坑，但基本不会影响业务。真有问题，也是开源，可以查代码解决

**优秀作业2：**

架构的完整度

dubbo的未来定位是专注于RPC领域，成为微服务生态系统的一个重要组件

spring cloud提供了一个微服务的全面解决方案。

1.功能对比： dubbo vs springcloud

服务注册中心 ZK eureka

服务调用方式 RPC http

服务监控 dubbo-admin spring boot admin

断路器 不完善 hystrix

服务网关 无 zuul

分布式配置 无 config

服务跟踪 无 sleuth

消息总线 无 bus

数据流 无 stream

2、 社区活跃度 dubbo

阿里已经重启对dubbo的维护，

spring cloud:提供了一个微服务的全面解决方案，活跃度优于dubbo

3、 学习成本

由于spring cloud都是基于springboot搭建的，其学习成本较低

对于新人来说，能很快的融入团队； 而dubbo的学习成本较大。

总结：因此对于一个业务迭代迅速的团队来说，直接使用市面上的整套spring cloud架构足以可以解决问题；并且spring boot可以跟任何中间件搭配使用，例如ng,rocketmq等，都可以实现业务要求。

至于性能问题，都是通过内网进行Http调用，其效率也可满足公司需要

**优秀作业3：**

前两年我们也对 spring cloud 和 dubbo 进行了技术选型：

1. 从性能效率上来看：

dubbo 使用了 netty 比 spring cloud 的 http 快很多。大概是 2 倍的差距。

2. 从开发效率上来看：

公司同学一般都会 spring，入门 spring cloud 的门槛低且与 spring 融合度更高。dubbo 相对会花一些学习成本。

3. 从社区活跃度来看：

spring cloud 的活跃度明显高于 dubbo。而后续 dubbo 看到了微服务的流行，也逐渐 恢复了更新。

4. 从生态圈上来看：

spring cloud 周边组件很多，通过 starter 自动装配即拿即用。而 dubbo 得自己集成其 他的框架。

综合来看，当年公司选择 spring cloud 的原因是以下几点：

1. 对于性能没有极致的要求。

2. 毕竟是小公司，想尽快产出，减少学习成本，增加开发效率，减少企业成本。

**优秀作业4：**

1.服务框架当时准备从dubbo和srpingcloud两个中选型，最终选定springcloud.

2.选型对比维度：

1）整体解决方案

Dubbo没有自带的配置中心，注册中心基于zookeeper

springcloud自身相对比较完善，有注册中心、配置中心、负载均衡、服务治理

2）底层调用

Dubbo基于rpc，速度快

springcloud基于http，访问相对dubbo慢

3）技术前沿性（趋势）

Dubbo已经比较成熟

springcloud技术相对前缘

4）开发语言及开放度

都是Java，开源

5）社区活跃度

Dubbo官方已没有维护，但基本上能碰到的问题，网上能找到解决方案 ；目前已 重新开始维护；

springcloud英文的，资料比较少。虽然可能会趟坑，但是业务简单，基本不会遇到什么大问题。 基于自身业务场景，主要只是展示功能，所以速度慢点，影响不大。

优先考虑整体解决方案和技术趋势，最终选择springcloud,

虽然可能会趟坑，但基本不会影响业务。真有问题，也是开源，可以查代码解决

## 7.30 自己独立画出Dubbo的底层架构原理图Dubbo底层架构原理的图

服务端启动流程：

1.服务提供者向注册中心注册服务。

2.并将 URL 和对应的 Impl 映射关系存到 exporterMap 中。

3.最后绑定端口和启动服务。 消费者请求流程：

(1) 消费者向注册中心获取服务注册信息存至 Directory 中。

1. 生成动态代理对象，该对象通过 router 路由和 loadbalance 负载均衡从 Directory 中选择一个服务地址。

(3) 通过 Exchange 组装好请求。

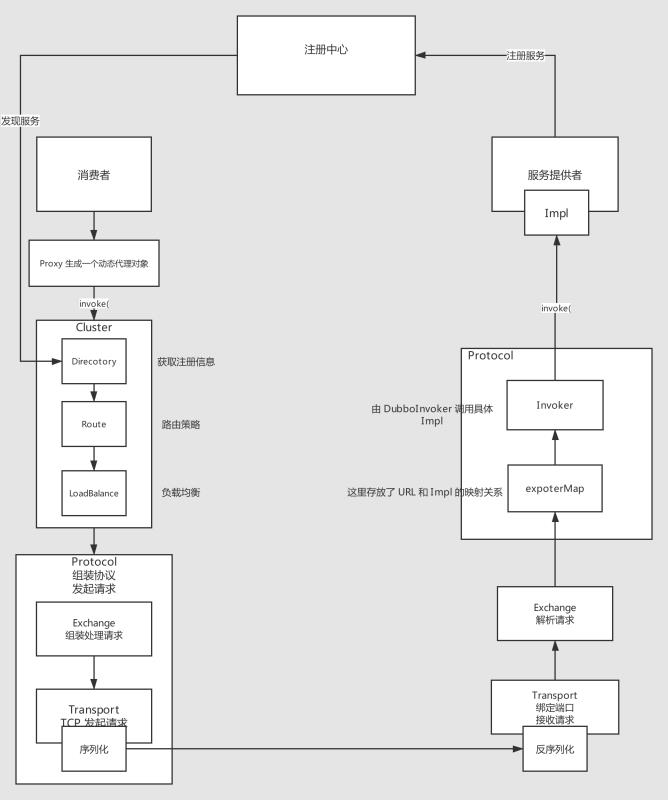
(4) 通过 Transport 层将序列化后的请求发送给服务端。

服务端响应流程：

1.服务端 Exchange 解析请求。

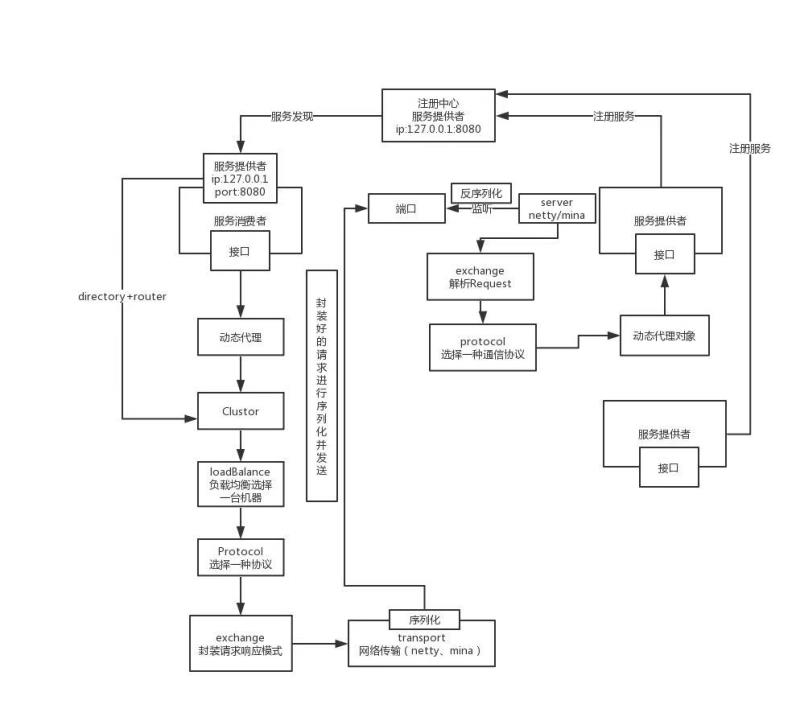
2.在 exporterMap 中请求获取对应的 Impl 实例。

3.调用 Impl 实例，最后再通过 Exchange, Transport 返回给消费者。



## 7.31

***作业1: 设计一个rpc框架画出设计图***

  
  
1: 首先需要service层，提供接口，给提供方和消费端实现。  
  
2: 然后提供一个注册中心，用于服务注册与发现，将服务方配置信息注册到注册中心进行管理。  
  
3: 如果消费方调用生产者，先通过动态代理，生成代理对象。  
  
4: 通过clustor组件中（directory 获取服务配置信息 router获取路由信息）。  
  
5:通过负载均衡策略（random loadbalance、roundrobin loadbalance、leastactive loadbalance、consistanthash loadbalance）选择一台机器进行服务调用。  
  
6: 选择一种通信协议（dubbo、http、rmi）进行服务之间交互。  
  
7: 将请求参数通过exchange组件封装。  
  
8: 通过网络通信框架netty将数据进行序列化发送到指定的端口。  
  
9: 服务方监听发送过来的请求并处理。  
  
10: 解析请求对象进行反序列化。  
  
11: 选择一种通信协议生成动态代理调用接口。  
  
12:结果处理通过动态代理进行返回。

1、动态代理层：RPC框架的一切调用细节都是动态代理方法里实现的。可选用Java原生Proxy类方法或Spring的Cglib实现。生成接口类的代理类，重写方法，实现真实调用。

2、注册中心：提供者启动时进行注册，消费者对某些服务进行订阅，服务发现。

① zookeeper， dubbo service-A service-B provider cosumer provider consumer IP+port IP+port IP+port IP+port IP+port IP+port IP+port IP+port 如上图设计好的树状结构来存储

Providers启动时在Zookeeper中某个服务节点下创建临时子节点，节点名称使用IP、端口、权重等信息拼接字符串，这样实现服务注册。

Consumer监听某个服务节点服务的子节点，并且拉取子节点列表。实现服务订阅和发现。

当Provider上线或下线时，临时节点发生新增或删除。所以Consumer那边监听会收到通知，拉取最新的注册表数据，更新本地注册表缓存，这样实现服务发现。

② 也可以采用Eureka的实现方式，采用纯内存ConcurrentHashMap来存储注册表，Provider启动时将IP、端口、权重等信息注册。实现服务注册。

Consumers定时去拉取注册表缓存在本地。实现服务发现。然后各服务采用定时心跳的方式来告诉注册中心自己是否存活。 实现故障感知。

比如用Map<String,Map<String,Lease<InstanceInfo>>>这样的结构来保存，服务名为key，Value Map的key为实例ID，Lease中维护最近的心跳时间，InstanceInfo中保存实例的IP、端口等信息。这是内存注册表。

并且可以加入多级缓存来优化并发读写冲突。 配置好合适的注册表拉取时间和心跳时间，比如30秒，则可以单机承载大规模系统日千万级的访问量。

综上：Zookeeper方式显然更加实时感知集群变动。而Eureka的方式因为是定时拉取，所以消费端对集群变更感知会有一定延时性。

3、Cluster层：实现路由和负载均衡、集群事件处理，比如故障切换等。必备轮询、加权轮询、随机、加权随机、一致性Hash等路由或负载均衡方式。

加权可以根据权重增加虚拟节点实现。一致性Hash，采用Hash环方式，并加入一定数量的虚拟节点来使节点更加均衡的分布在Hash环上，顺时针旋转时，继而请求分布更加均匀。可以采用ConcurrentSkipListMap来实现。

4、数据发送部分：Exchange，Portocol、序列化反序列化机制、Transport网络IO。 Exchange封装为Request/Response，结合BlockingQueue将Netty异步通信转化同步。

协议可以是HTTP、dubbo等 序列化加入多种选择，比如Json、Protobuf、Protostuff、Hessian、Kryo等、Java序列化等等。

5、网络IO部分，可以使用BIO、NIO。BIO性能极差！推荐使用NIO框架Netty来实现，并处理好数据粘包问题。

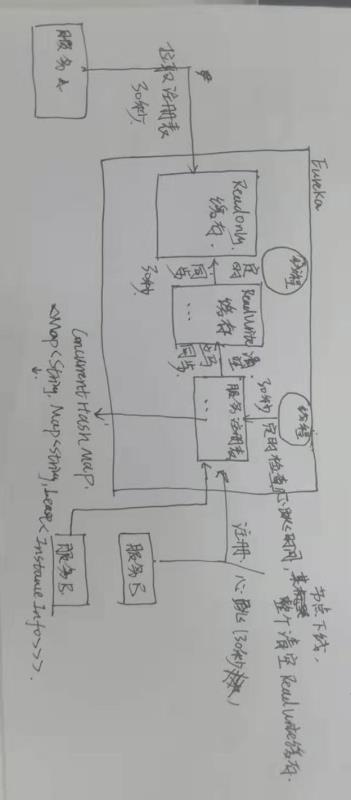
使用多路复用IO机制，服务端有一组Acceptor线程，ServerSocketChannel监听某个端口，而Acceptor线程通过多路复用器Selector来轮询监听ServerSocketChannel的Accept事件。

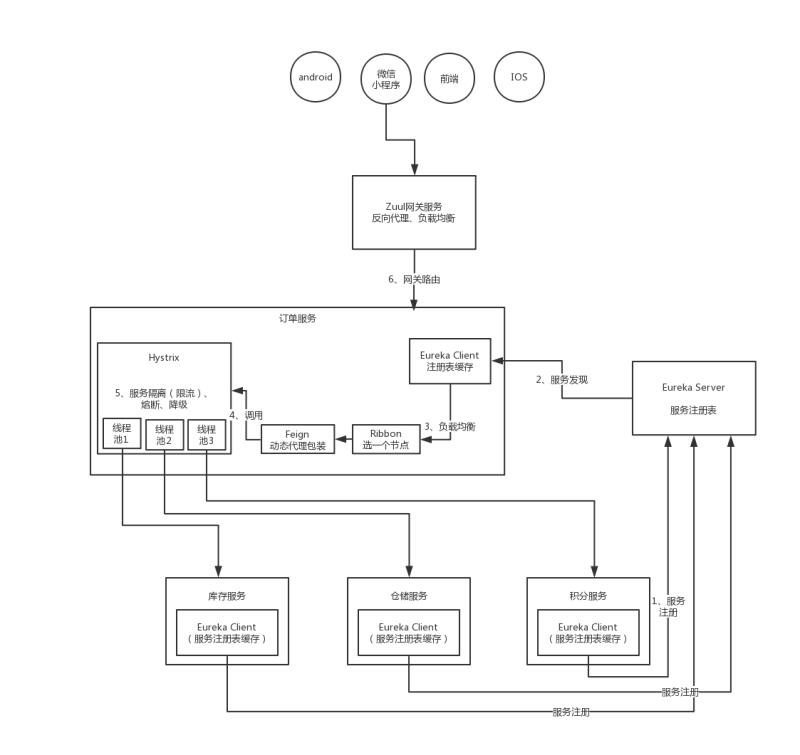
当客户端的netty请求连接时，此时会创建对应的SocketChannel。被Processor线程组通过Selector轮询监听，来处理实际的IO。

当客户端发来数据时，服务端SocketChannel触发事件，由Processor线程去解析请求，经历反序列化和Exchange、Protocol层层解析后，经由代理类找到对应的服务实现方法处理。再经过SocketChannel返回给客户端。

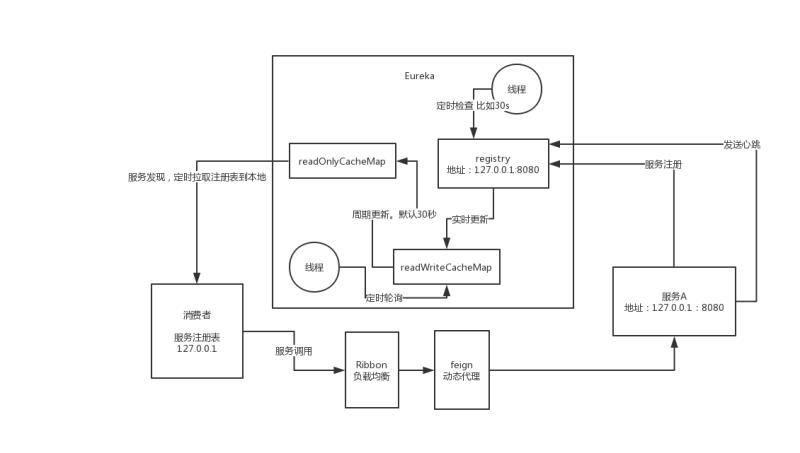
客户端同样是processor线程组监听到SocketChannel的事件，读取响应给程序。

综上所述： 划分层次的话，就是10层 Service、Config、Proxy、Registry、Cluster、Protocol、Exchage、Transport、Serialize、Monitor





***作业2：画Spring Cloud底层原理图***

  
  
1: 首先有个服务接口，IP地址127.0.0.1，端口8080，服务注册并暴露(Erueka)。  
  
2: spring Cloud Eureka注册中心有个注册服务表记录服务注册上来的服务地址端口等信息。  
  
3: Eureka注册表registry会立马向读写缓存readWriteCache同步注册信息。  
  
4: readWriteCache会通过一个线程定时（30s）向readOnlyCache同步一次注册配置信息。  
  
5: 消费方会从readOnlyCache中定时同步注册信息到本地缓存注册表。  
  
6: 消费方会通过本地缓存route信息通过负载均衡Ribbon选择一台机器。  
  
7: 选择完机器后就通过feign组件调用服务接口进行交互。  
  
8: eureka还有检测服务心跳与故障功能，会定时扫描registry服务注册上来的心跳，如果90s内没有心跳，认为服务已挂，会remove注册表信息。  
  
9: 故障的注册配置信息清空后也会立马同步给readWriteCache缓存，缓存做清空操作。  
  
10: 此时readOnlyCache可能并没有和readWriteCache同步，导致消费者调用故障地址所在的服务调用失败。  
  
11: 当readOnlyCache定时同步比对writeOnlyCache时候，发现readWriteCache已经被清空，那么readOnlyCache也被清空。  
  
12: 消费者来readaOnlyCache已经清空则去readWriteCache同步，发现还是为空，则去register拉去注册服务，然后同步给readWriteCache和readOnlyCache缓存。  
  
13：消费者通过正常机器的地址去调用服务接口正常。  
  
提问：  
 老师，您好，关于今天讲解的Eureka原理有个疑问：就是某个接口所在的服务器故障后，register感知后，则把该故障的配置移除，然后立马同步给readWriteCache  
  
 这个过程中，readWriteCache会做全量清空的操作,后续的readOnlyCache定时比对readWriteCache后，也会做全量清空的操作?  
  
 为什么这几个缓存直接的数据同步不做增量同步的操作呢?哪些不一致就清空哪些不可以么?这种设计的优点会是什么?麻烦请教下老师 谢谢!  
  
  
问题解答：  
 缓存最麻烦的就是缓存更新操作，缓存增量更新会稍微复杂一点；  
  
 其次在生产环境下服务很少会出现下线等操作，所以清空缓存实际不会那么频繁，直接删除缓存就好了，等下次拉取注册表用到的时候再进行缓存

Spring Cloud 基本原理图 服务注册和发现流程：

1. 服务提供方会注册到 eureka 上，写入注册表中。

2. 注册表内容有变化，会将 writeOnly 清空。

3. 有一个定时线程，每隔 30 秒将 writeOnly 的内容同步至 readOnly。

4. 有一个心跳定时检测线程，每隔一定时间移除没有心跳的节点信息。更新注册表，然后再走 2 的步骤。

5. eureka client 会每隔 30 秒读取 readOnly，保存至本地。 采用了 readOnly 和 writeOnly 是为了减少锁的竞争，让服务快速注册上 eureka，而不用等待锁阻塞。

这里有一个疑问：老师，您在视频讲解的时候说，Eureka 启动后，服务第一次注册时，会立马把注册表数据同步给 writeOnly。

但是如果有一个节点重启完成后，注册表上会有一个新的节点，此时注册表是清空 writeOnly。

我的疑问是 只有Eureka 启动后，服务第一次注册会立马同步数据给 writeOnly吗？此后，注册表变动，都是清空 writeOnly 吗？

服务调用流程：

1. Feign 会对加了注解的接口生成一个动态代理对象。

2. Ribbon 会从 eureka client 注册表通过负载均衡策略，拿到一个请求地址。

3. Feign 拿到 Ribbon 负载均衡后得到的请求地址，并发起 HTTP 请求。

RPC 框架架构设计图 服务注册发现阶段：

1. 服务提供者向注册中心注册服务。

2. 注册中心保存这个服务提供者的节点信息。

3. 服务调用者向注册中心订阅服务，并获取节点信息。

4. 当节点信息变化时，注册中心要主动通知给服务调用者，服务调用者及时同步本地注册表。

服务调用阶段：

1. 对服务提供者的接口生成动态代理对象。

2. 这个动态代理对象中有 “熔断降级 Mock 等机制” ，“服务注册表” 和 “负载均衡”。

3. 动态代理对象会根据是否熔断降级和 Mock 走相应的逻辑。如果是真实请求，那么通过负载均衡机制从注册表中选一个请求地址和端口，并发起请求。

4. 请求后，过滤器链中会记录请求的行为，做好监控。

5. 接下来协议层进行封装协议。

6. 传输层将序列化后的数据发给服务提供者。

消息接收阶段:

1. 服务提供者拿到数据进行反序列化和解析协议。

2. 通过过滤器链统计请求处理信息，进行监控。

3. 提取出请求数据里面的 impl 类名和方法参数，进行反射调用。

Netty 基本流程说明：

1. bossGroup 中取一个 nioEventLoop 把 ServerSocketChannel 注册到 selector 上。

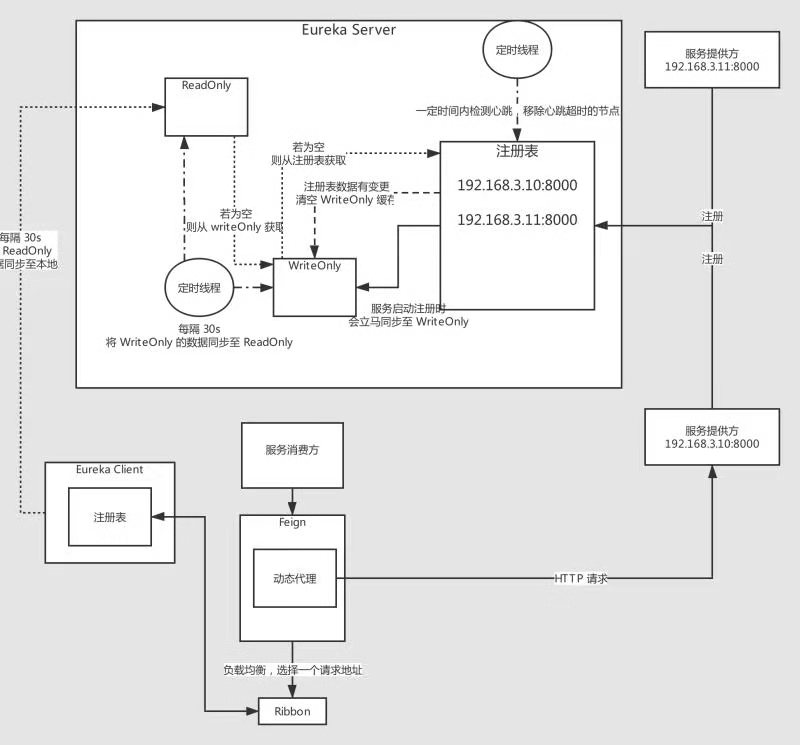
2. nioEventLoop 通过 ServerSocketChannel 绑定端口然后注册 OP\_ACCEPT 事件。

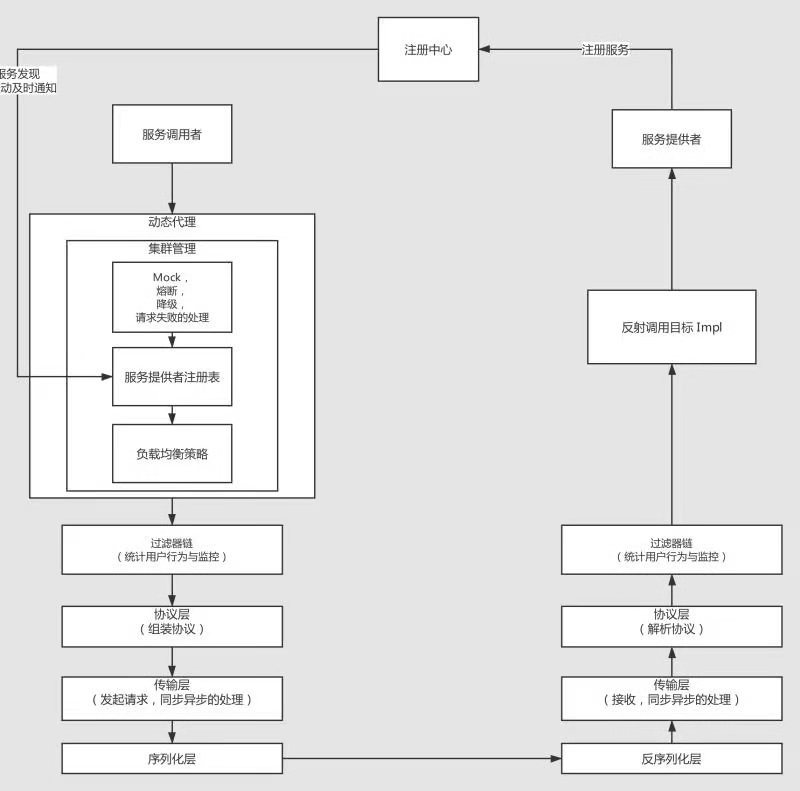
3. 当有新连接进入的时候，ServerSocketChannel 将新连接交给 childGroup 来处理。

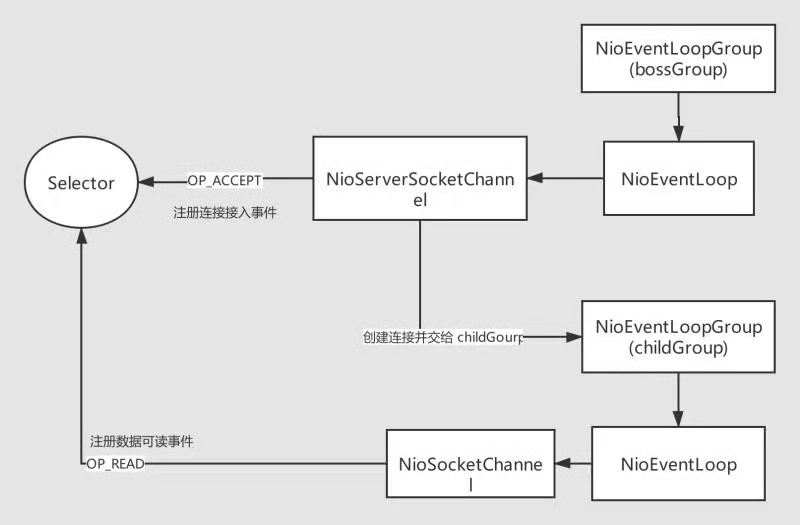
4. childGroup 选一个 nioEventLoop 来负责这个 SocketChannel。

5. 接下来，nioEventLoop 把 SocketChannel 注册到 selector 上，并关注 OP\_READ 事件。

6. 之后有数据读事件来时，SocketChannel 就可以来处理了。







问题回答：

服务实例发生注册、下线、故障的时候，也就是注册表发生变动时都会去刷新readWriteCacheMap，将对应的缓存给过期掉

## 8.1

**说一下自己公司的服务注册中心怎么技术选型的？生产环境中应该怎么优化？**

**1. 画出公司的服务注册中心原理图！**

**2. 考虑一下数据一致性**

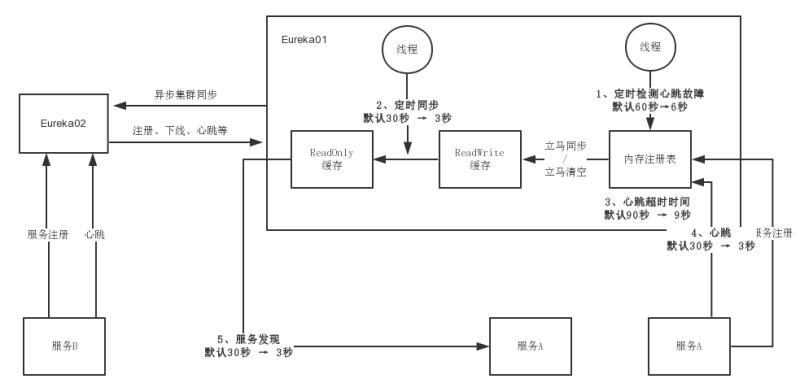
**3. 看一下公司的服务注册、故障发现的时效性是多长？**

**4. 注册中心最大能支撑多少服务实例？**

**5. 生产环境是如何部署的？几台机器？每台机器的配置？**

1. **如何保证的可用性？服务注册、故障、发现的时效性是否有做过优化？**

注册中心原理如上图



1. 选型：

我们公司使用的Spring Cloud服务框架，所以注册中心采用Eureka，Eureka选择了CAP中的AP，也就保证牺牲强一致性保证可用性。

由于是peer-to-peer方式，所有Eureka地位都是对等的，平等对外提供服务，服务往一个Eureka节点注册时，会异步同步给集群中其他Eureka节点。

如果同步失败，也不会影响可用性，会出现短暂的数据不一致。直到同步心跳信息时，发现数据不一致，然后重新同步缺失的注册信息，实现最终一致性。

1. 问题优化：

开始采用默认的参数的时候，遇到一些问题，发现服务上线延时不稳定，有时会1分钟以上，而服务下线延时更长，最高会达到几分钟，时效性太差！

后来我优化了相关参数，现在将上线延时控制和下线延迟控制在秒级。时效性提高了10倍，虽然Eureka请求压力大了10倍，但是依然很轻松。

①、上线部分：因为时效性主要是：ReadOnly缓存的默认定时同步间隔是30秒 + 服务拉取注册表默认30秒 + Ribbon更新本地缓存一个时间，所以极端下，延时超过1分钟。所以我把上图中2处，ReadOnly缓存的默认定时同步间隔改成3秒；服务拉取时间改为3秒。这样实现秒级上线，几秒钟。

②、下线部分：因为下线实现性主要是 线程定时检测服务心跳故障的时间间隔，默认60秒（另外认定心跳过期的时间，默认90秒）+ ReadOnly缓存的默认定时同步间隔是30秒+服务拉取注册表默认30秒，极端情况下，延时了几分钟了。

所以我把上图1处线程定时检测服务心跳故障的时间改为6秒；认定心跳过期时间改为9秒；心跳间隔改为3秒。这样实现服务下线秒级感知， 极端情况下最高也只有十几秒。然后关闭自我保护机制，避免了其代码中的bug。

1. 规模：

因为系统每天QPS只有300-400，部署了2个节点，4核8G配置。并发、容量、可用性都可以保证。

1. 容量计算：

假如系统是几百个服务，2000个实例的大型规模系统，默认拉取注册表间隔为3秒，心跳时间为3秒，这样Eureka压力为每秒1333次请求

由于Eureka是纯内存操作，并且多级缓存避免并发读写冲突，并且多节点分担读负载，所以使用高性能的服务器还是可以支撑的。

但是由于Eureka集群是点对点模式，服务和心跳信息要进行集群同步，多节点可以分担服务发现负载，但是不能分担服务注册和心跳负载。

综合考虑，最好还是控制在1000节点内，配合上述参数优化，可以实现秒级上下线感知。

老师您看，我这个项目经历能符合互联网高级工程师吗？P5的。硕士，2年1个月工作经历。

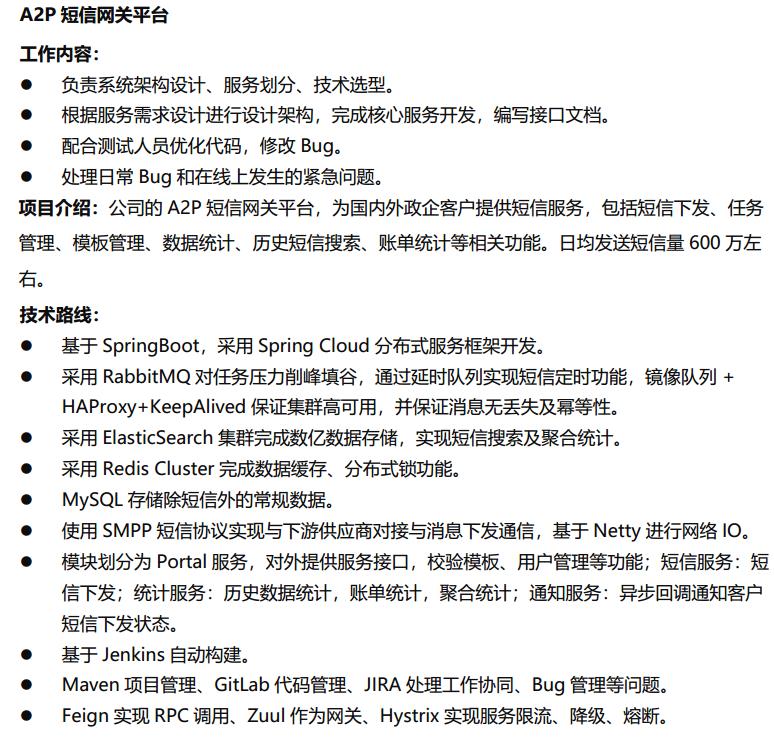
【A2P短信网关，项目上线4个月，日均短信量600万，Elasticsearch积累了6亿数据量220G，ES使用5台6核64G机器，部署5个shard，2个副本。ES堆内存分配20G，剩下40G留给文件系统缓存，将数据全部缓存到os cache，实现毫秒级搜索。】

【RabbitMQ3台镜像集群，4核16G】 【集群部署了2个Eureka、2个Zuul、3个Portal服务、6个短信发送服务、3个通知服务，都是4核16G内存。】

老师别笑话我啊，传统通信公司，是在没有太大的QPS，写项目经历都愁。。。这个量已经是在吹牛了，不知道吹的靠不靠谱？有啥漏洞吗？

点评：

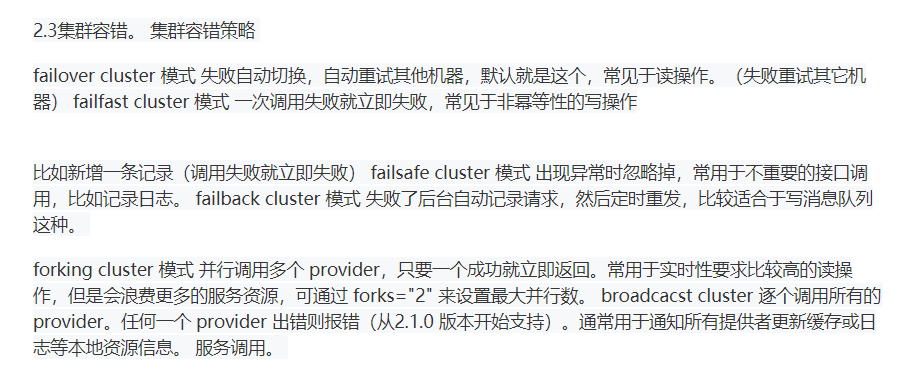
很不错。如果可以把训练营的一些技术融入到项目中，没问题的！

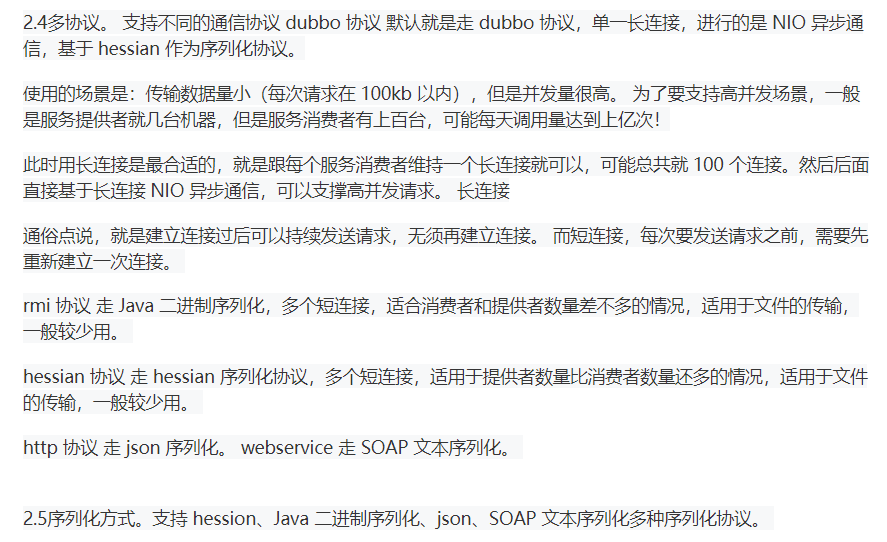


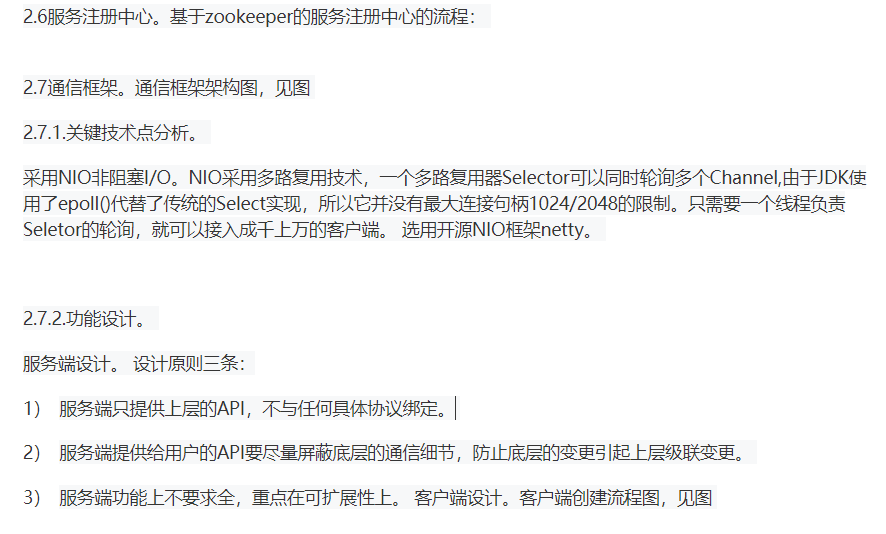
## 8.2ce7e17ed5534d8cf8b1308e947ec2e47_

## 8.5

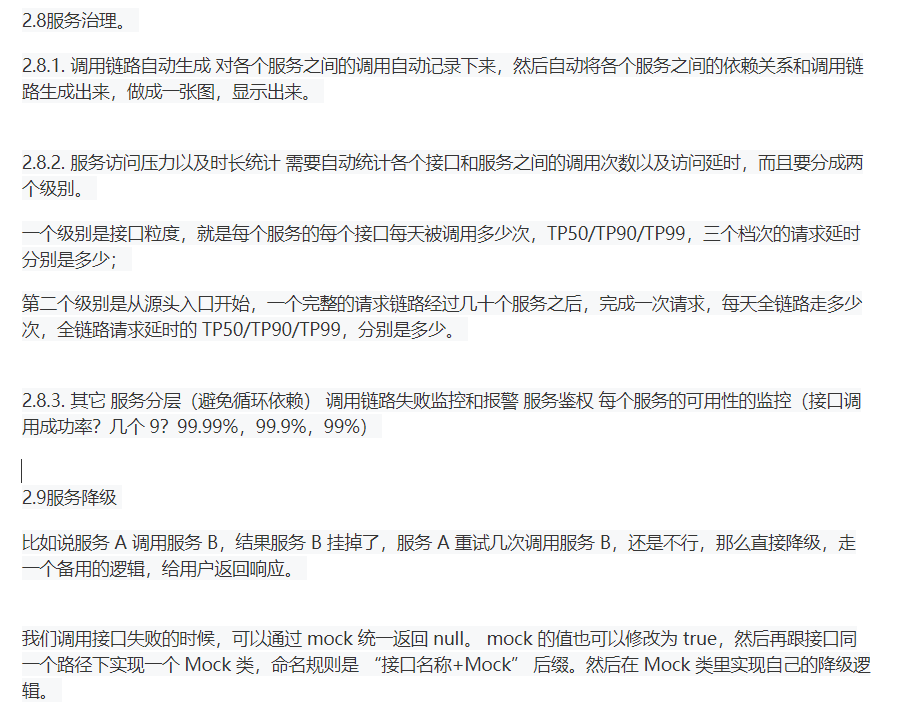


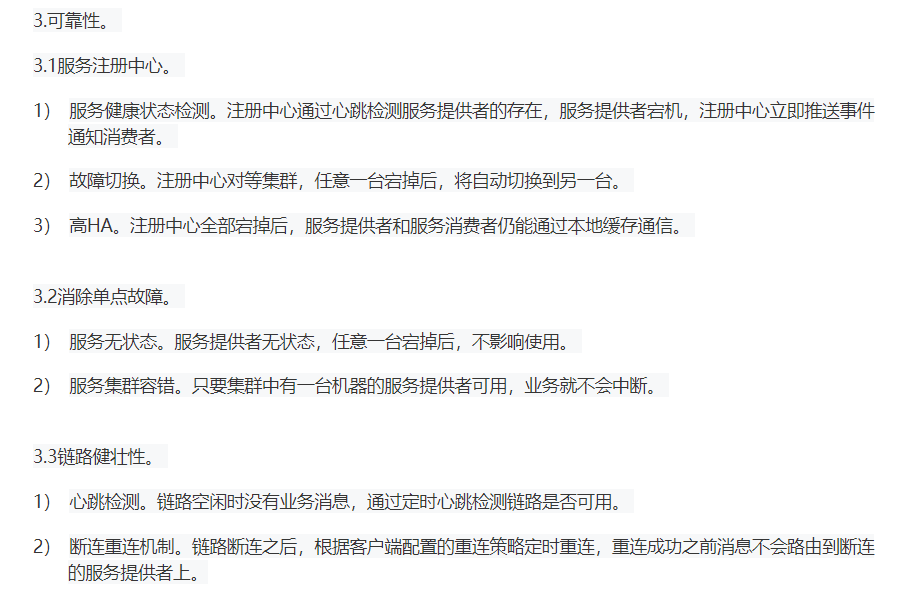


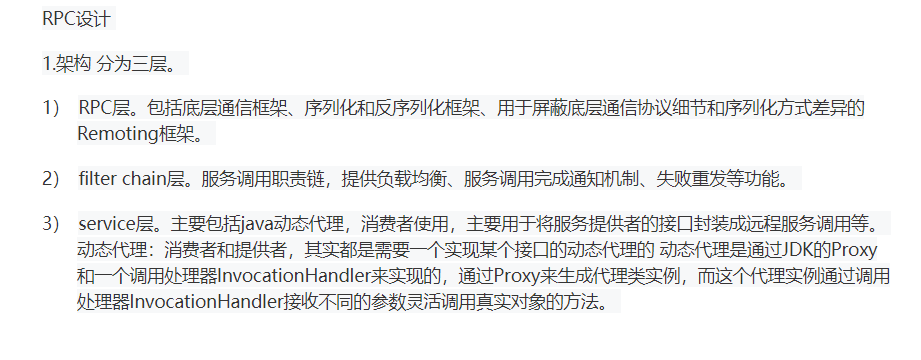


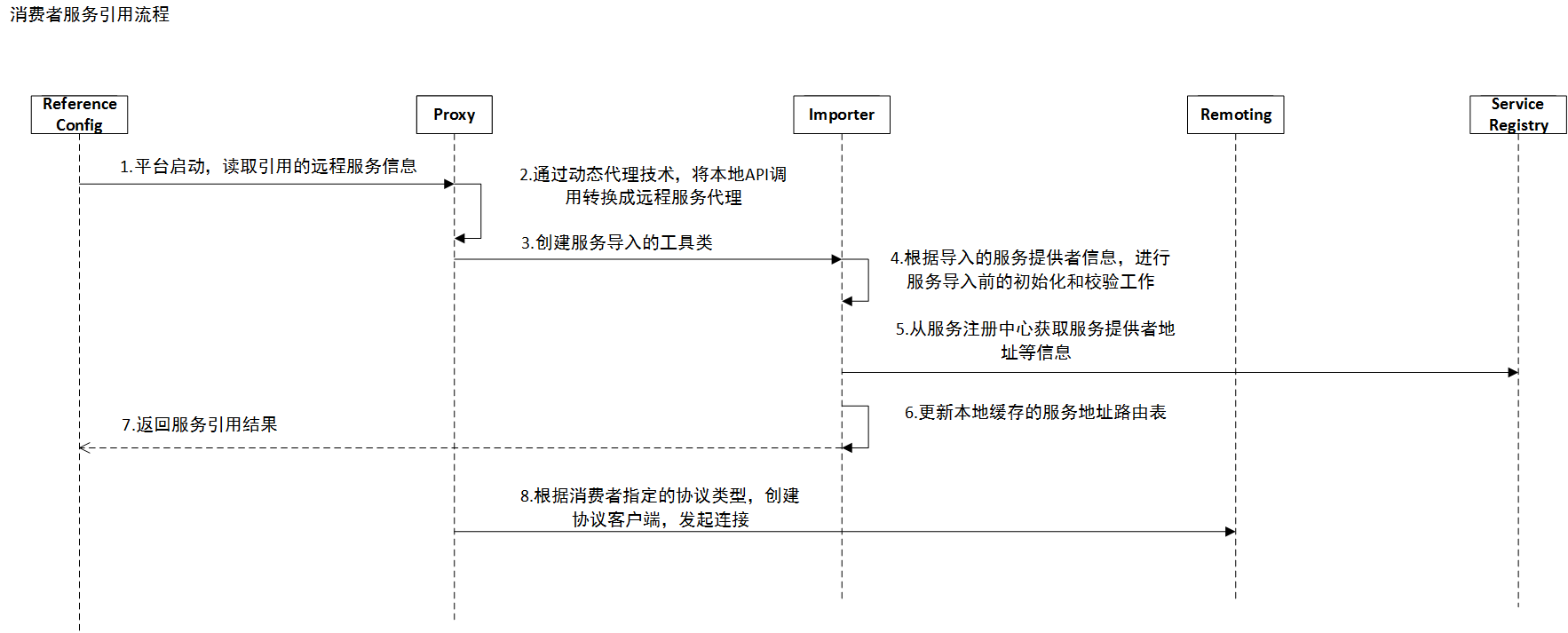


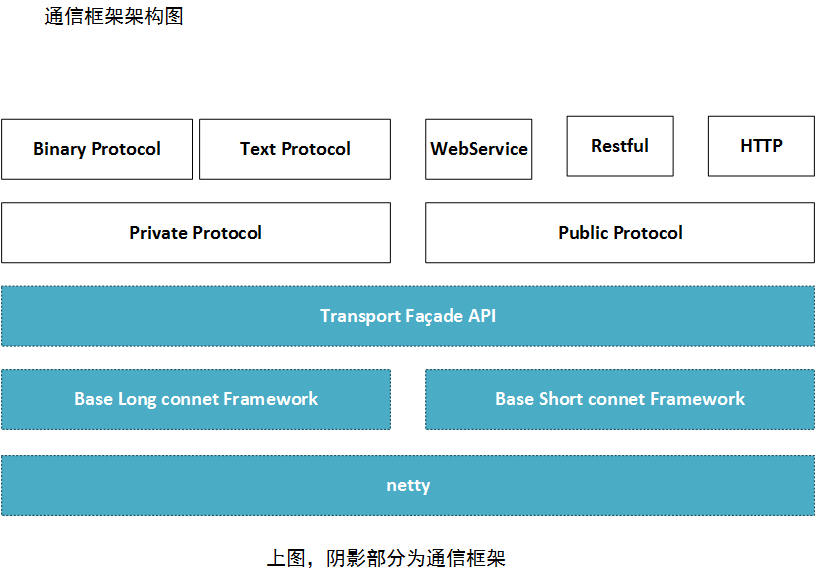


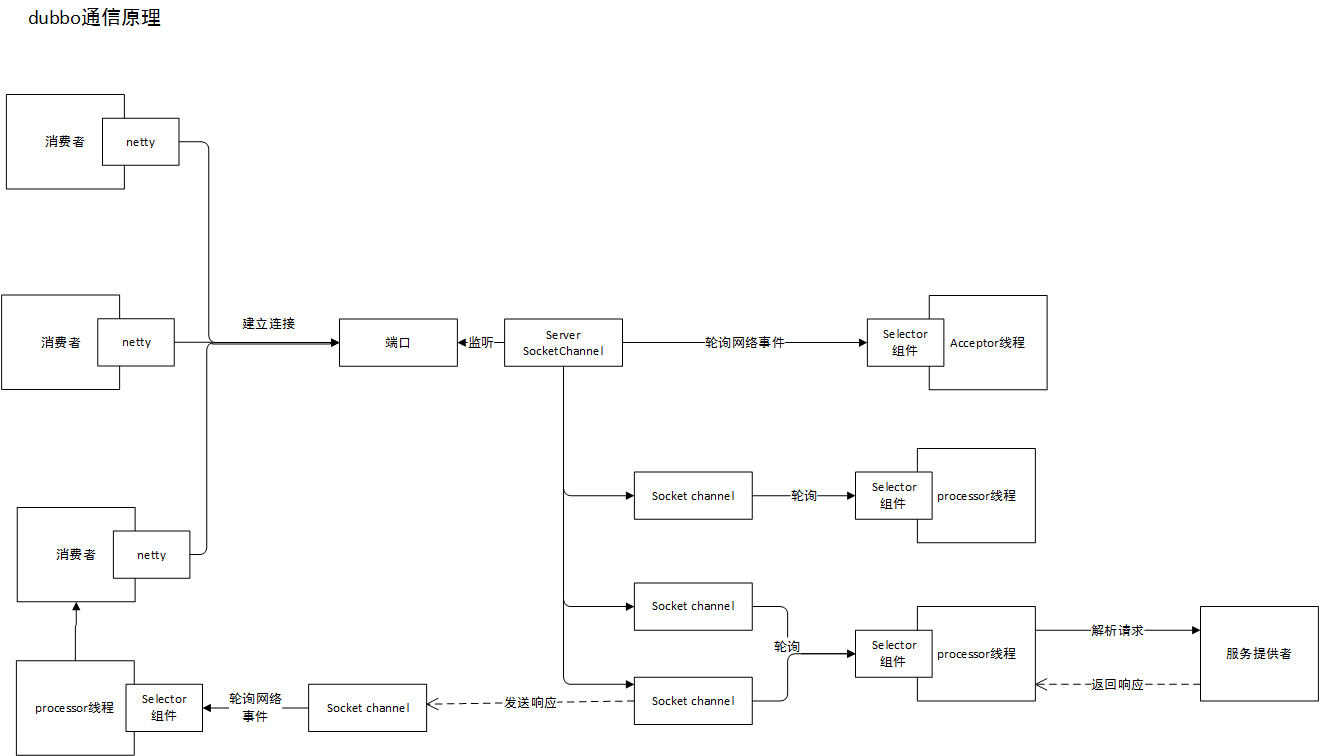


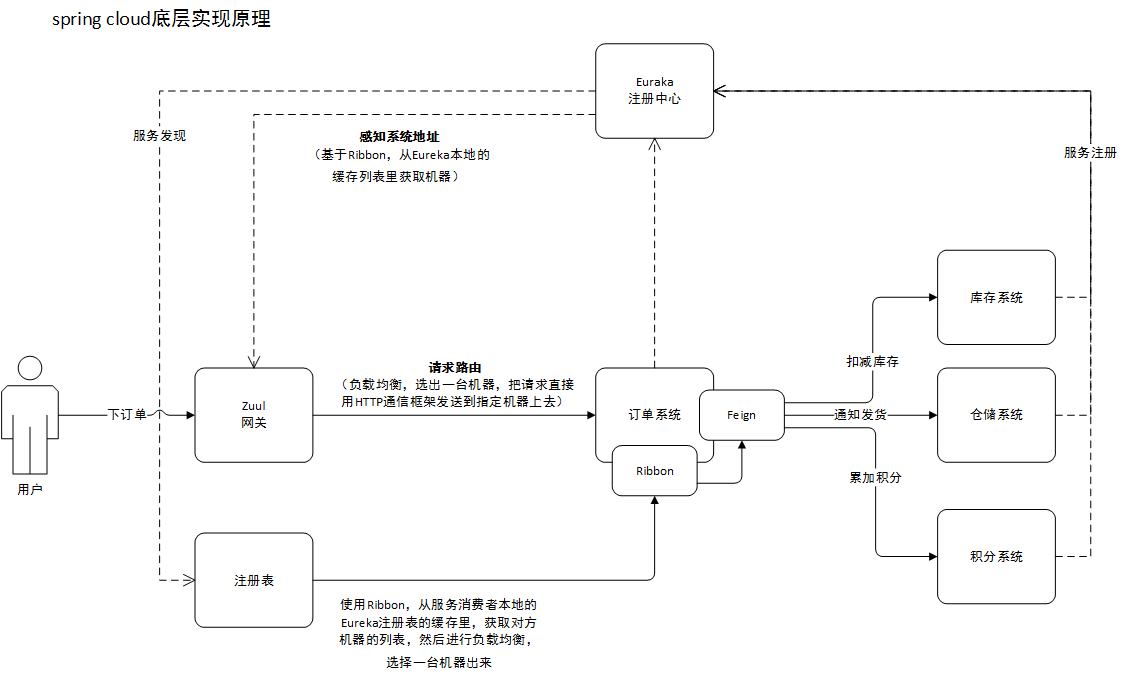


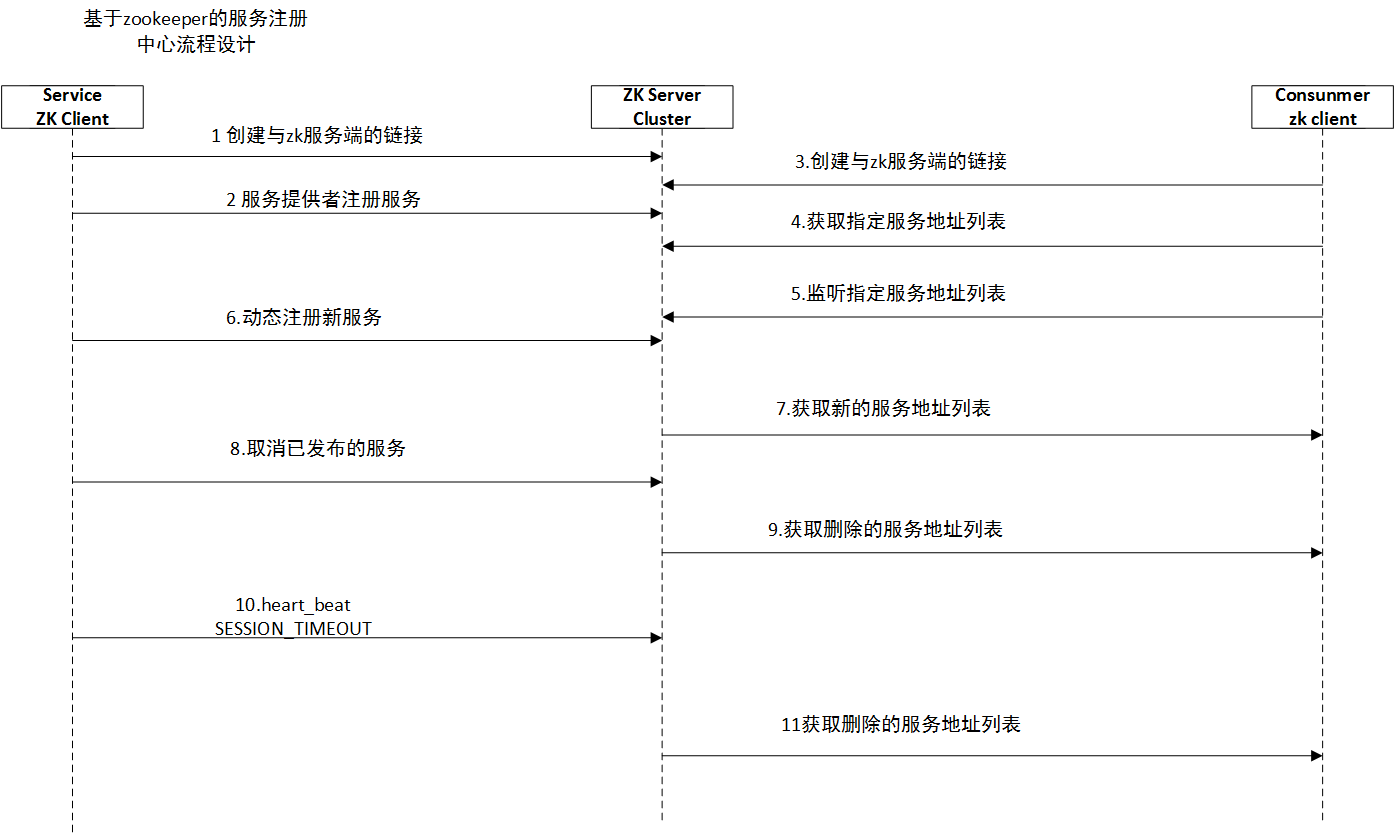


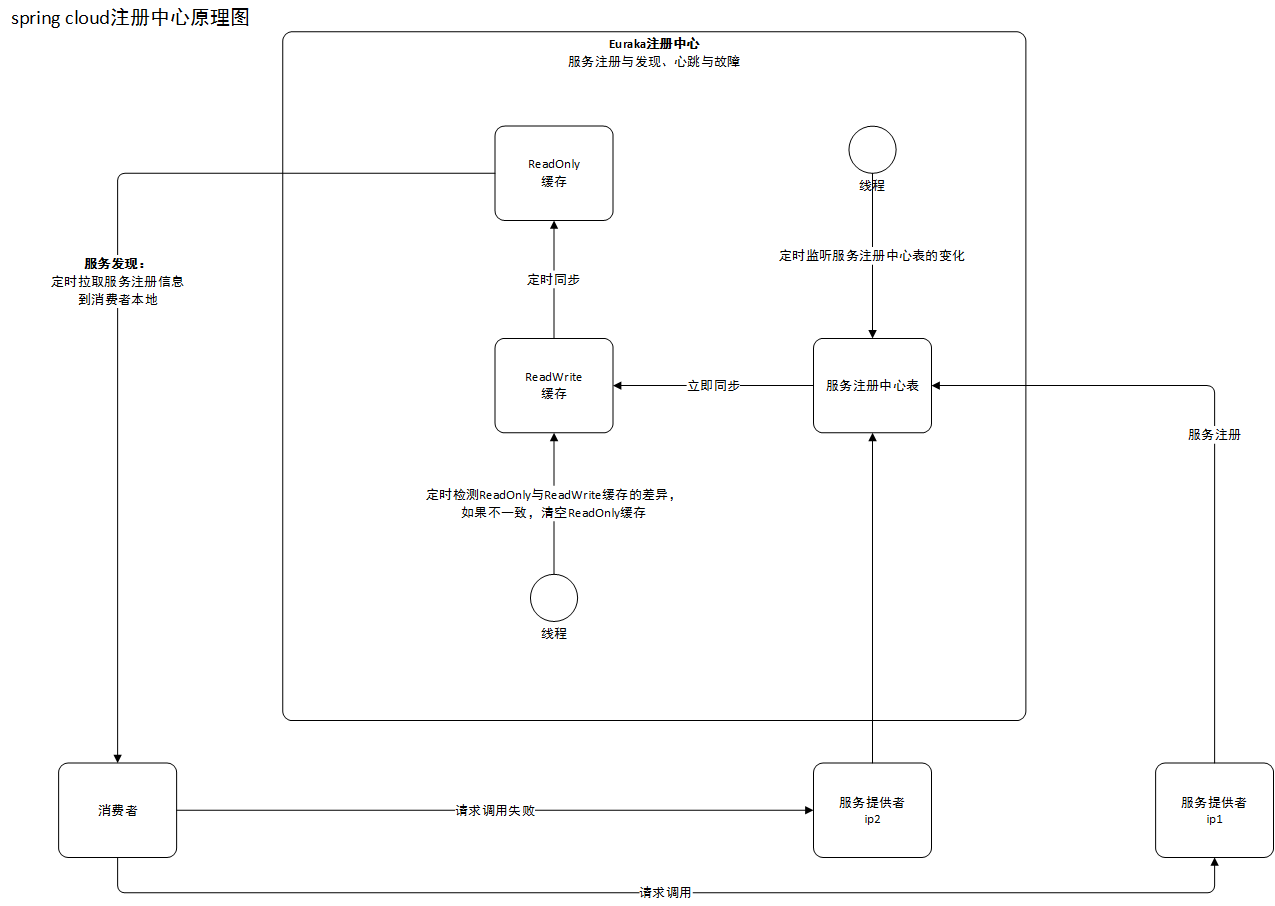


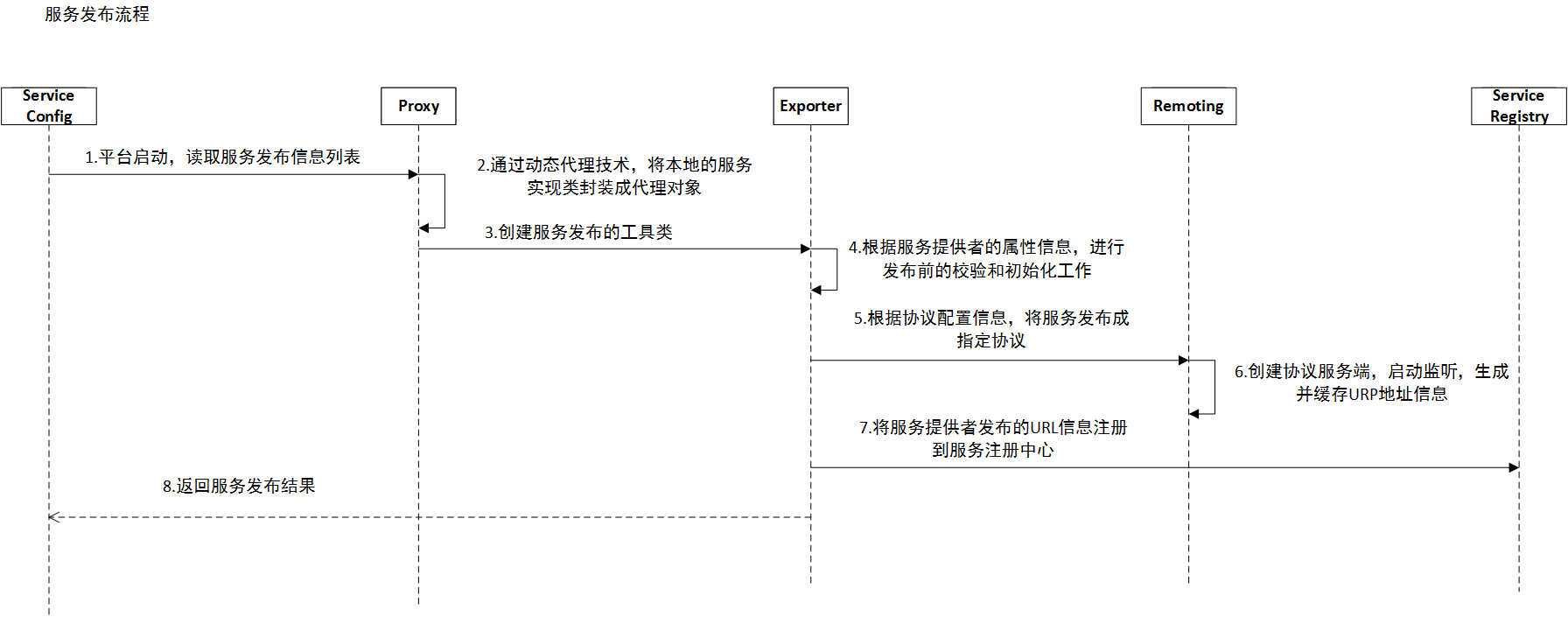




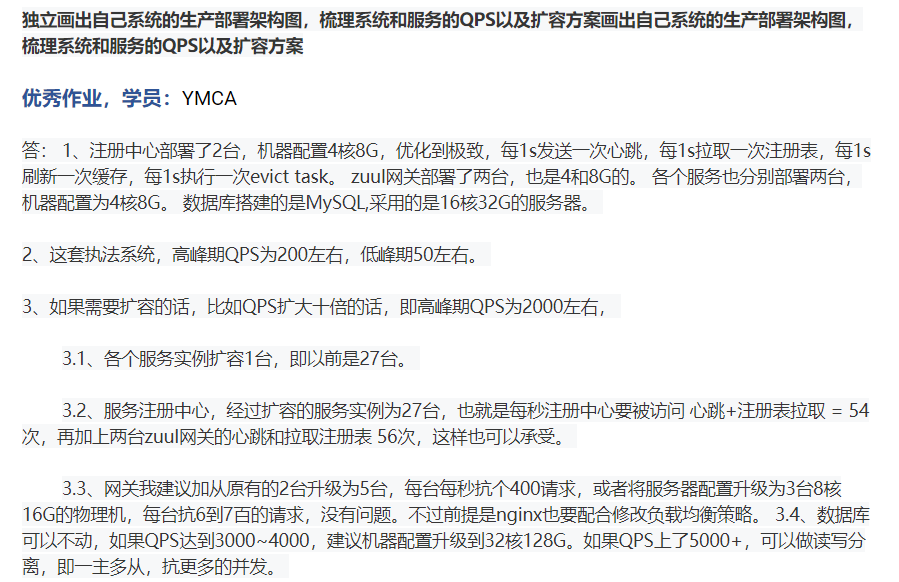


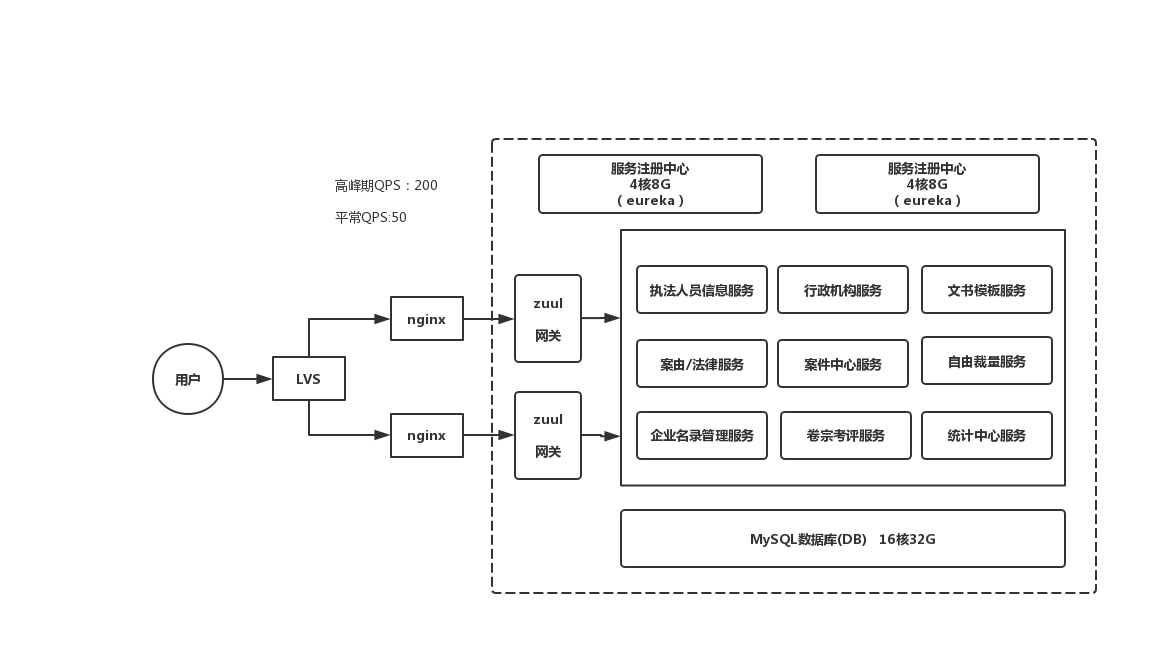




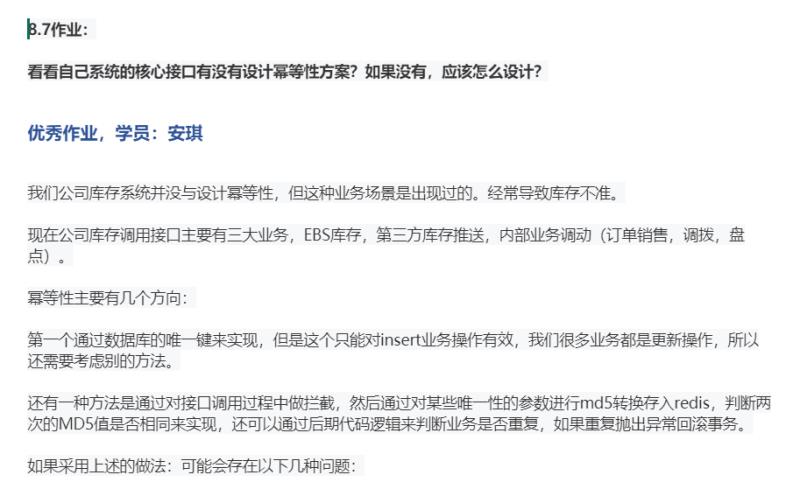


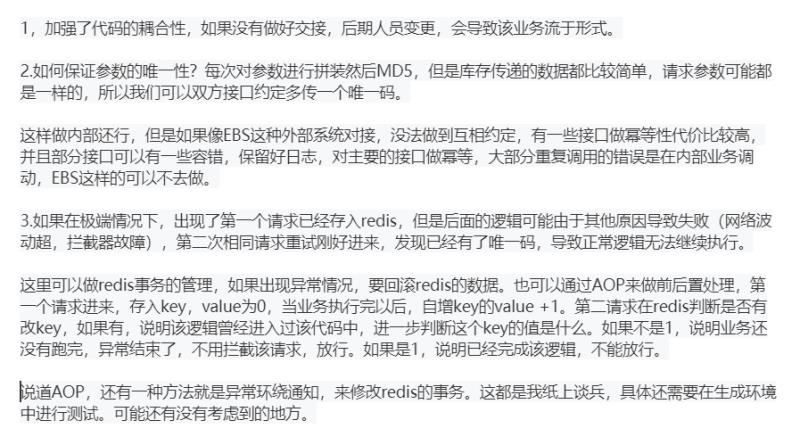
## 8.6

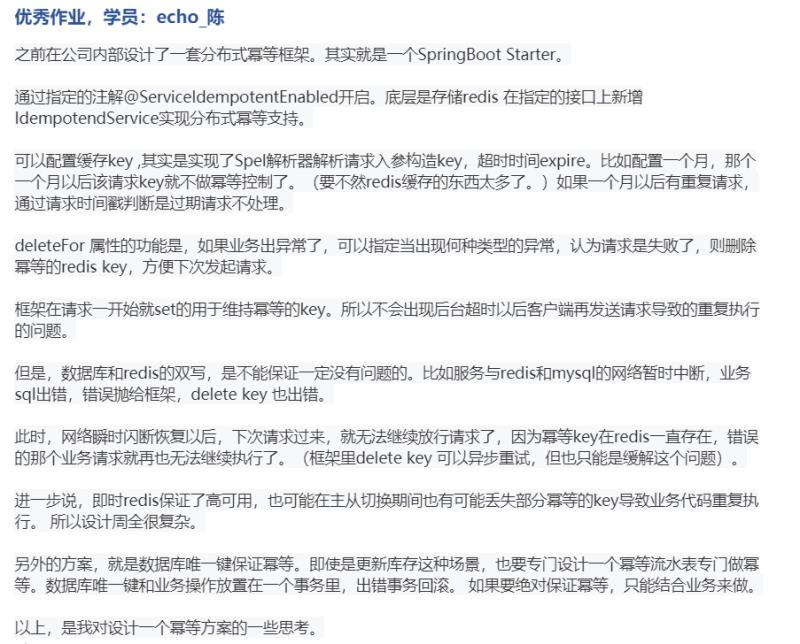




## 8.7





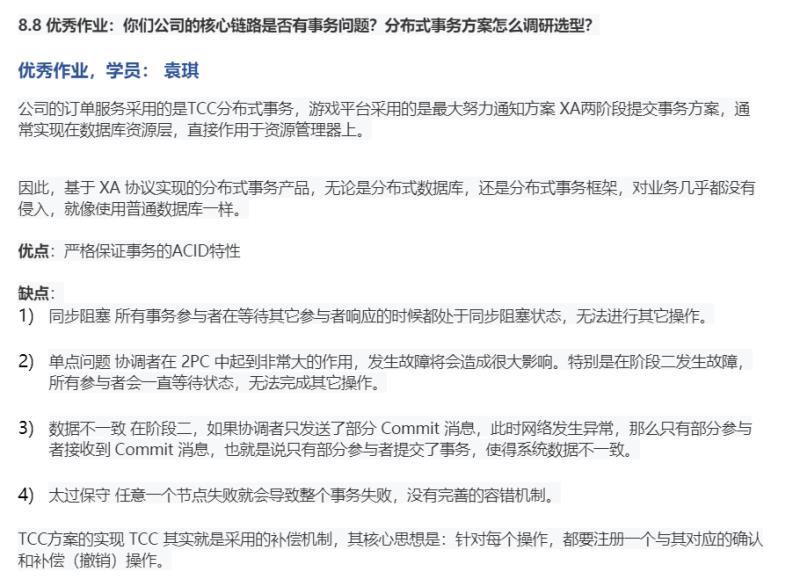


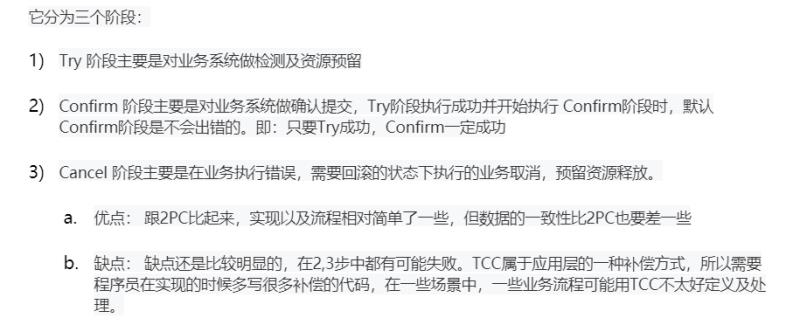


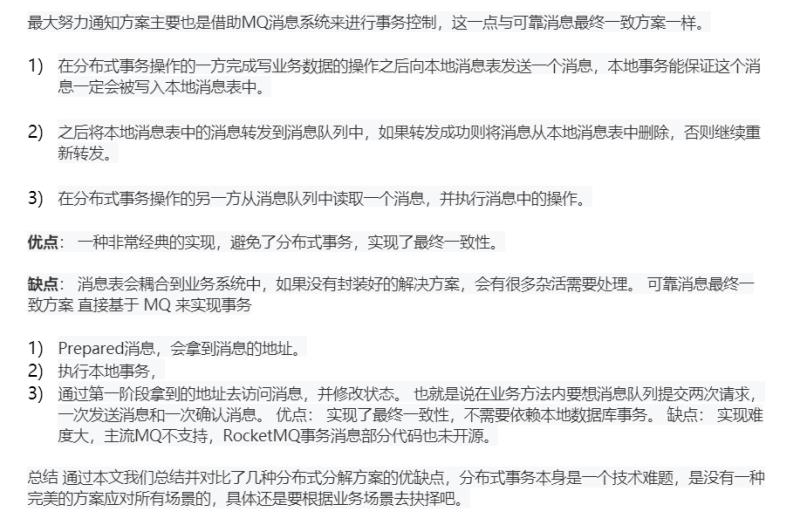


http://wuwenliang.net/2019/03/28/%E8%B7%9F%E6%88%91%E5%AD%A6RocketMQ%E4%B9%8B%E6%B6%88%E6%81%AF%E5%B9%82%E7%AD%89/

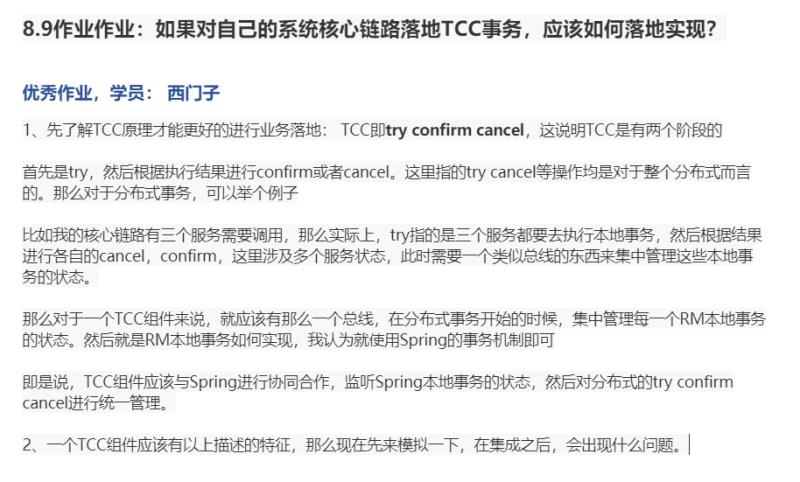
## 8.8

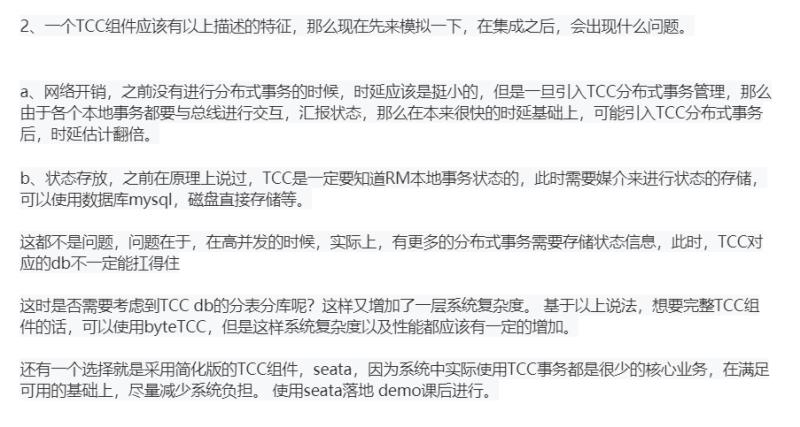




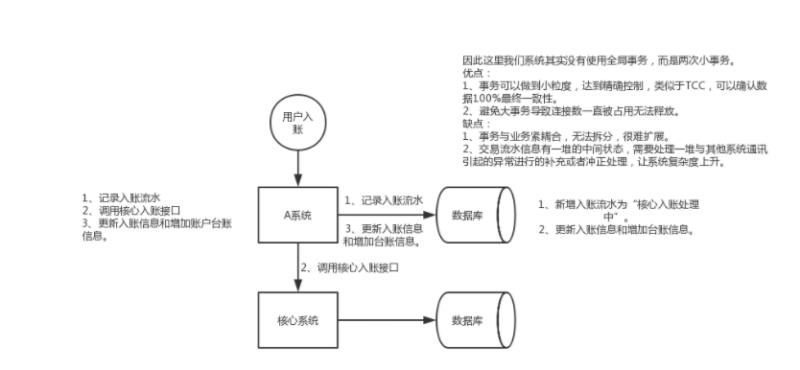


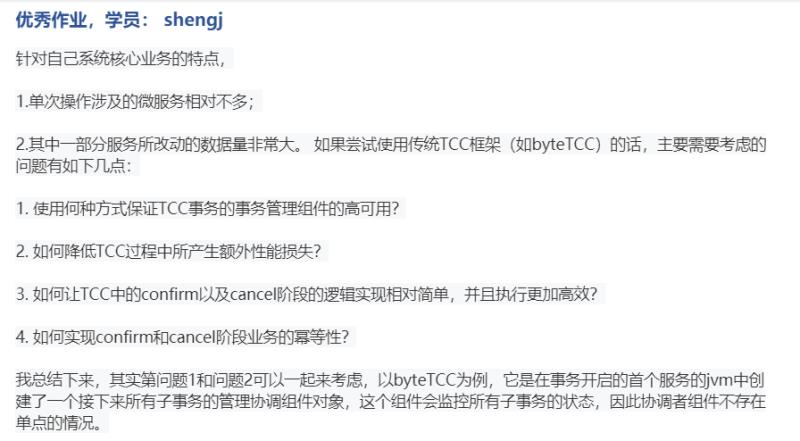
## 8.9

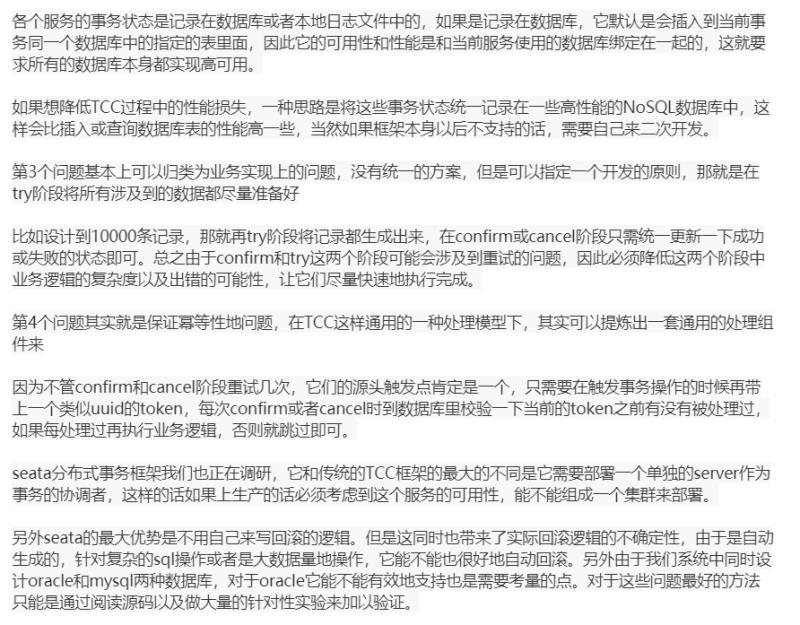




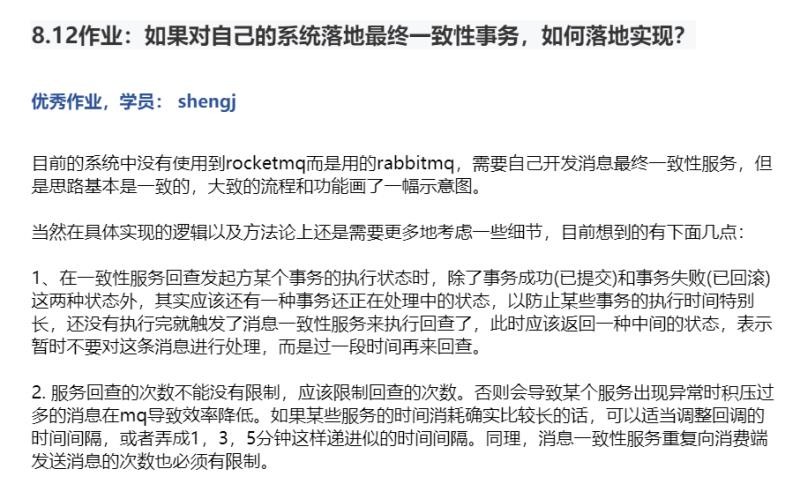


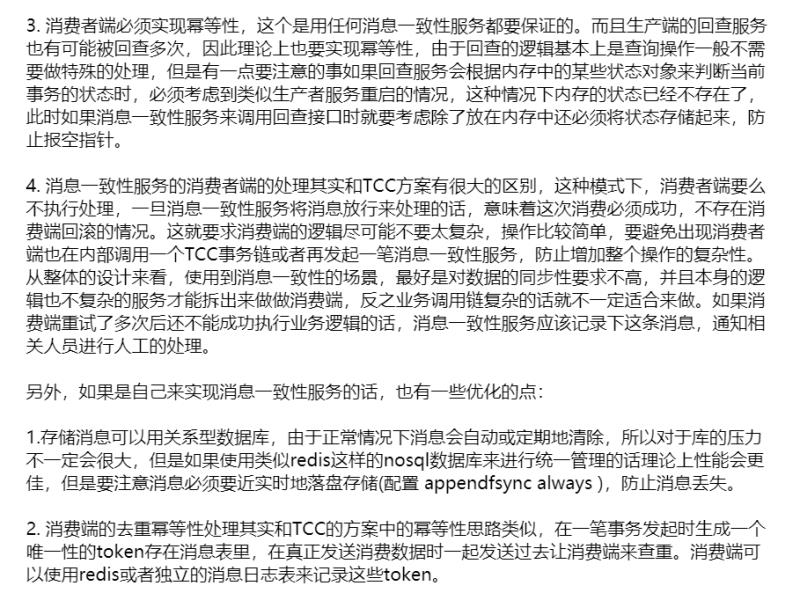


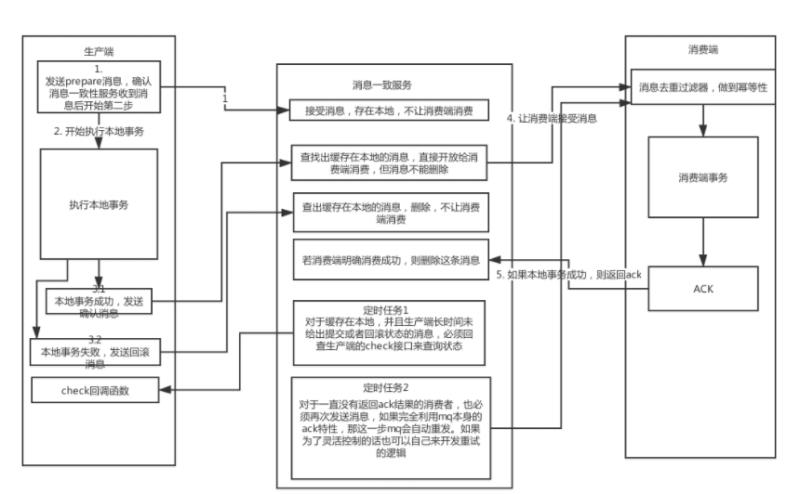




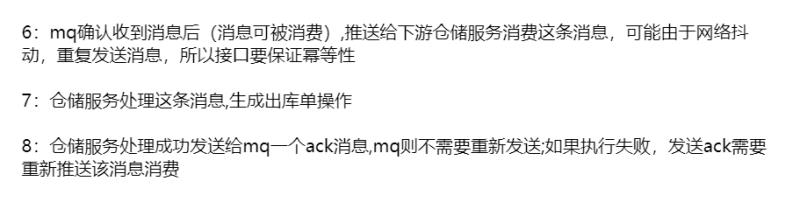
## 8.12

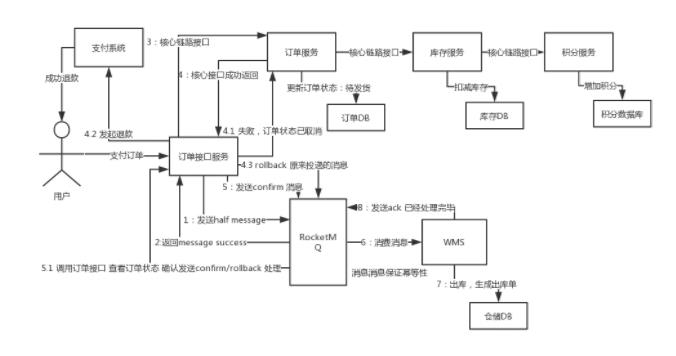












## 8.14

