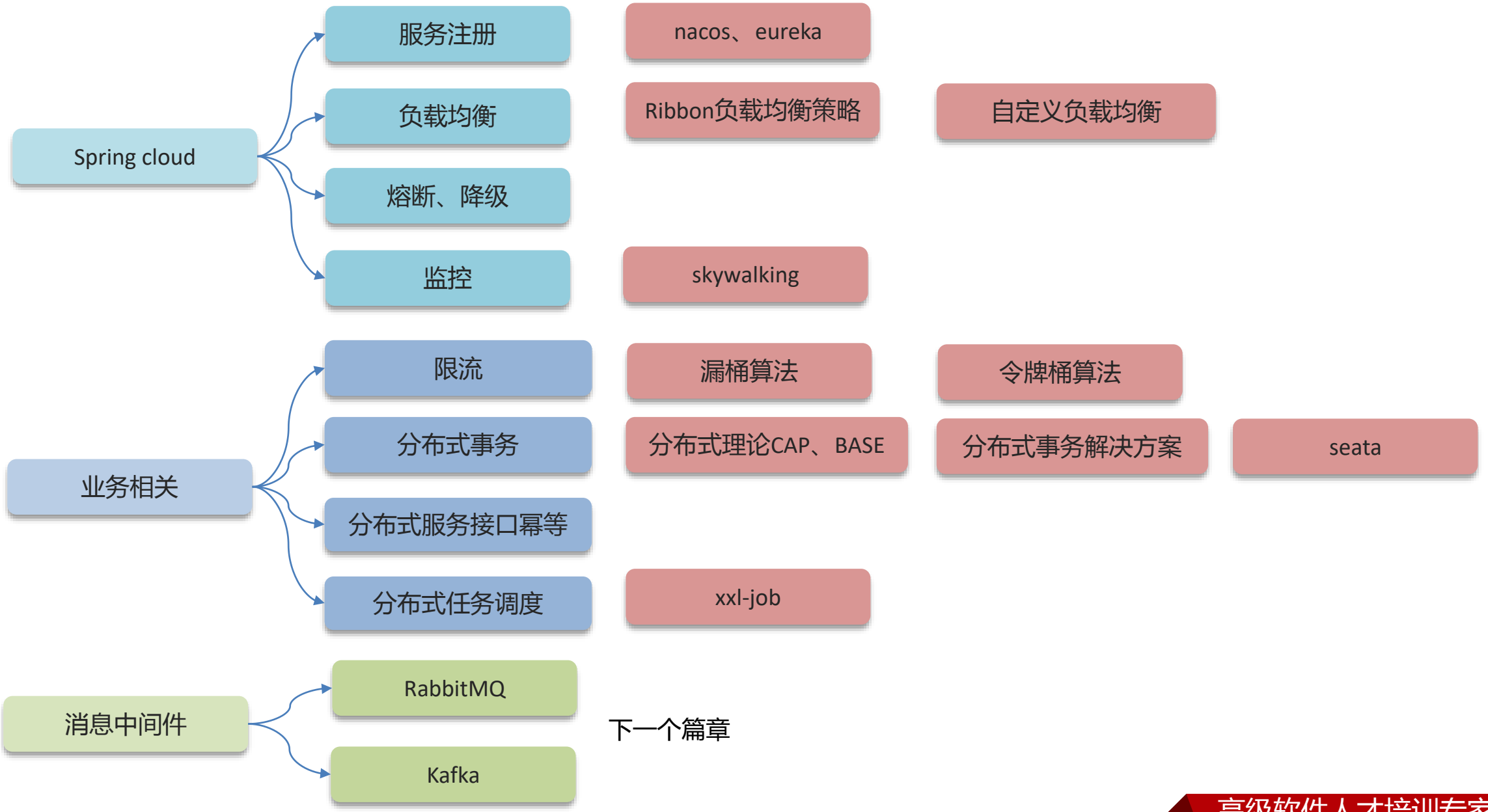
微服务篇

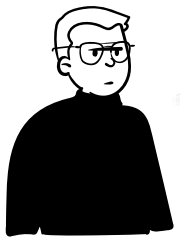
黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

多一句没有，少一句不行，用更短时间，教会更实用的技术！

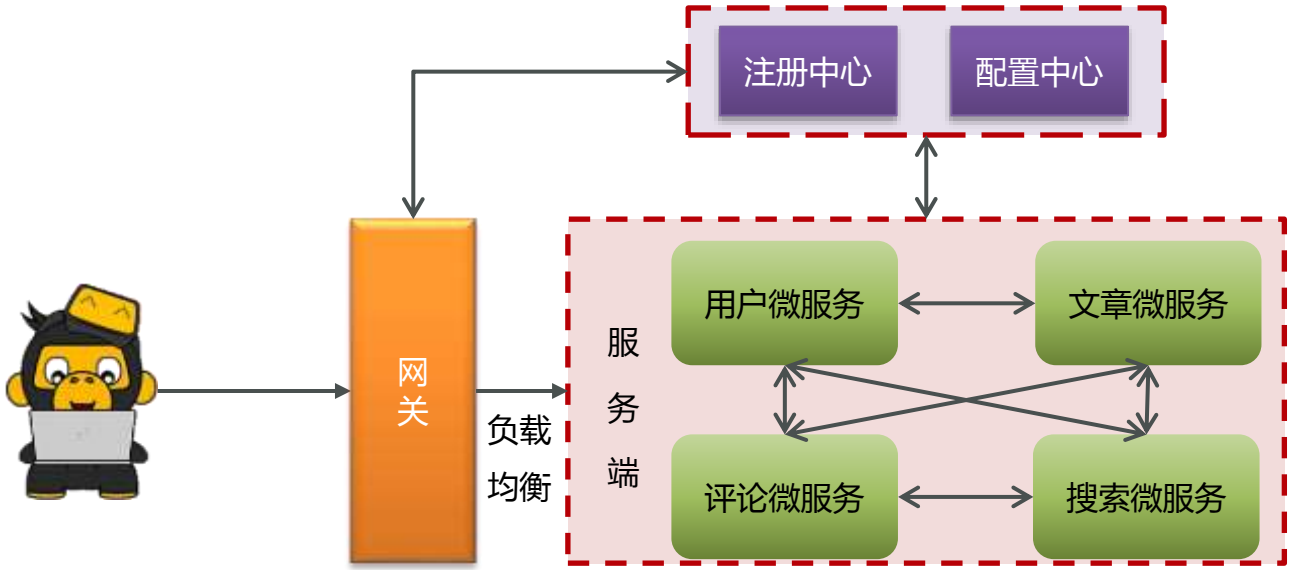


Spring Cloud

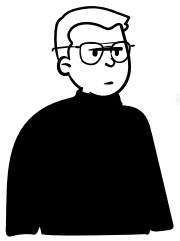


Spring Cloud 5大组件有哪些？

- 基础的内容考察
- 回答原则：简单的问题**不能**答错（一道面试题就能淘汰一个人）新手和老手都要注意



- Eureka ：注册中心
- Ribbon ：负载均衡
- Feign ：远程调用
- Hystrix： 服务熔断
- Zuul/Gateway ：网关



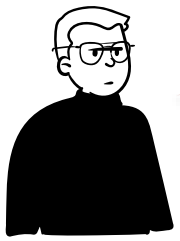
Spring Cloud 5大组件有哪些？

通常情况下：

- Eureka ：注册中心
- Ribbon ：负载均衡
- Feign ：远程调用
- Hystrix： 服务熔断
- Zuul/Gateway ：网关

随着SpringCloudAlibba在国内兴起，我们项目中使用了一些阿里巴巴的组件

- 注册中心/配置中心 Nacos
- 负载均衡 Ribbon
- 服务调用 Feign
- 服务保护 sentinel
- 服务网关 Gateway

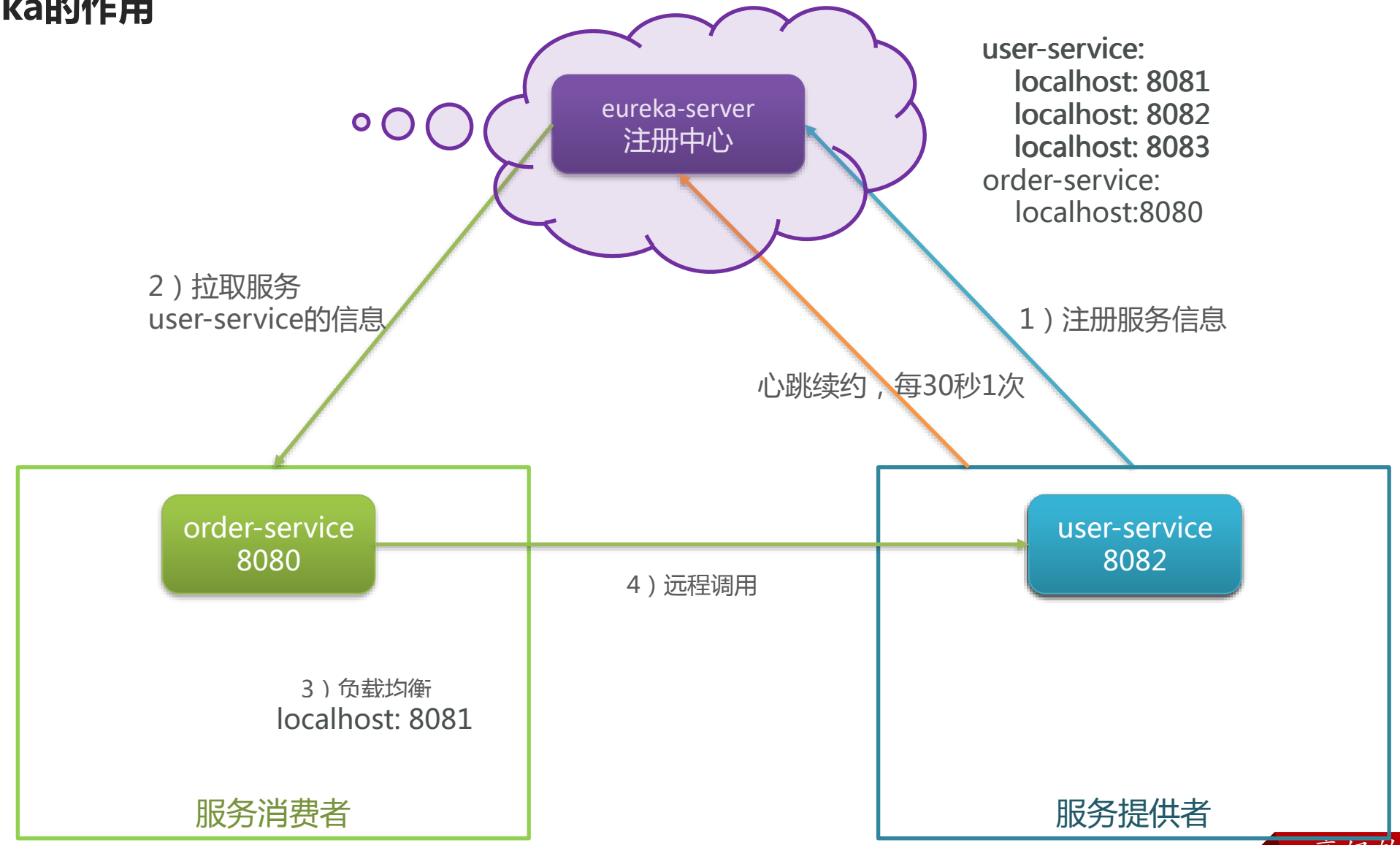


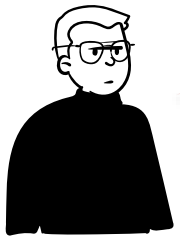
服务注册和发现是什么意思？Spring Cloud 如何实现服务注册发现？

- 微服务中必须要使用的组件，考察我们使用微服务的程度
- 注册中心的核心作用是：服务注册和发现
- 常见的注册中心：eureka、nocas、zookeeper

我做过的哪个微服务项目，使用了哪个注册中心

Eureka的作用


eureka-server
注册中心
user-service:
localhost: 8081
localhost: 8082
localhost: 8083
order-service:
localhost:8080
2）拉取服务
user-service的信息
1）注册服务信息
心跳续约，每30秒1次
order-service
8080
user-service
8082
4）远程调用
3）负载均衡
localhost: 8081
服务消费者
服务提供者



服务注册和发现是什么意思？Spring Cloud 如何实现服务注册发现？

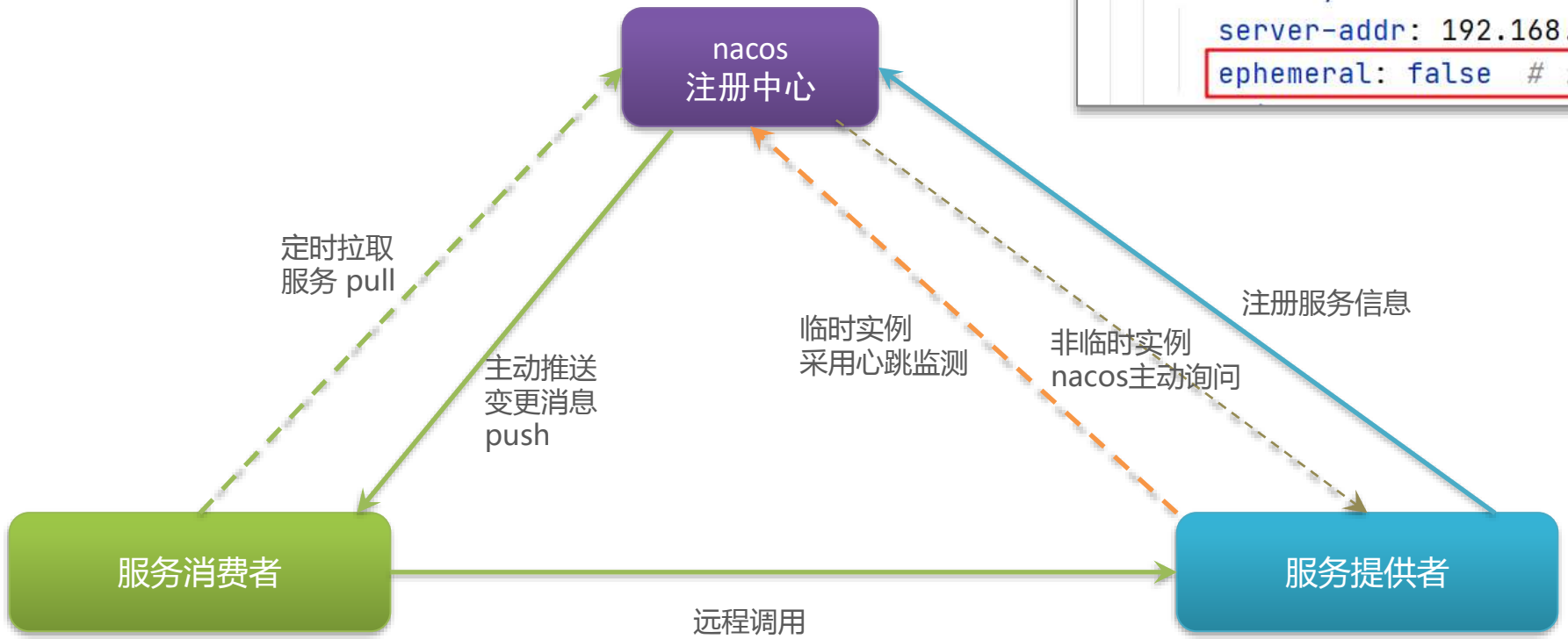
- 我们当时项目采用的eureka作为注册中心，这个也是spring cloud体系中的一个核心组件
- **服务注册**：服务提供者需要把自己的信息注册到eureka，由eureka来保存这些信息，比如服务名称、ip、端口等等
- **服务发现**：消费者向eureka拉取服务列表信息，如果服务提供者有集群，则消费者会利用负载均衡算法，选择一个发起调用
- **服务监控**：服务提供者会每隔30秒向eureka发送心跳，报告健康状态，如果eureka服务90秒没接收到心跳，从eureka中剔除

我看你之前也用过nacos、你能说下nacos与eureka的区别？

- 简历上有体现
- 面试官比较熟悉nacos和eureka

Nacos的工作流程

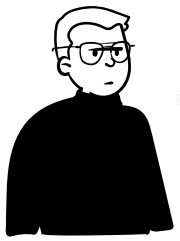
```yaml
spring:
  cloud:
    nacos:
      discovery:
        server-addr: 192.168.200.130:8848
        ephemeral: false  # 设置为非临时实例
```



user-service:
localhost:8081
...
服务列表缓存

我看你之前也用过nacos、你能说下nacos与eureka的区别？

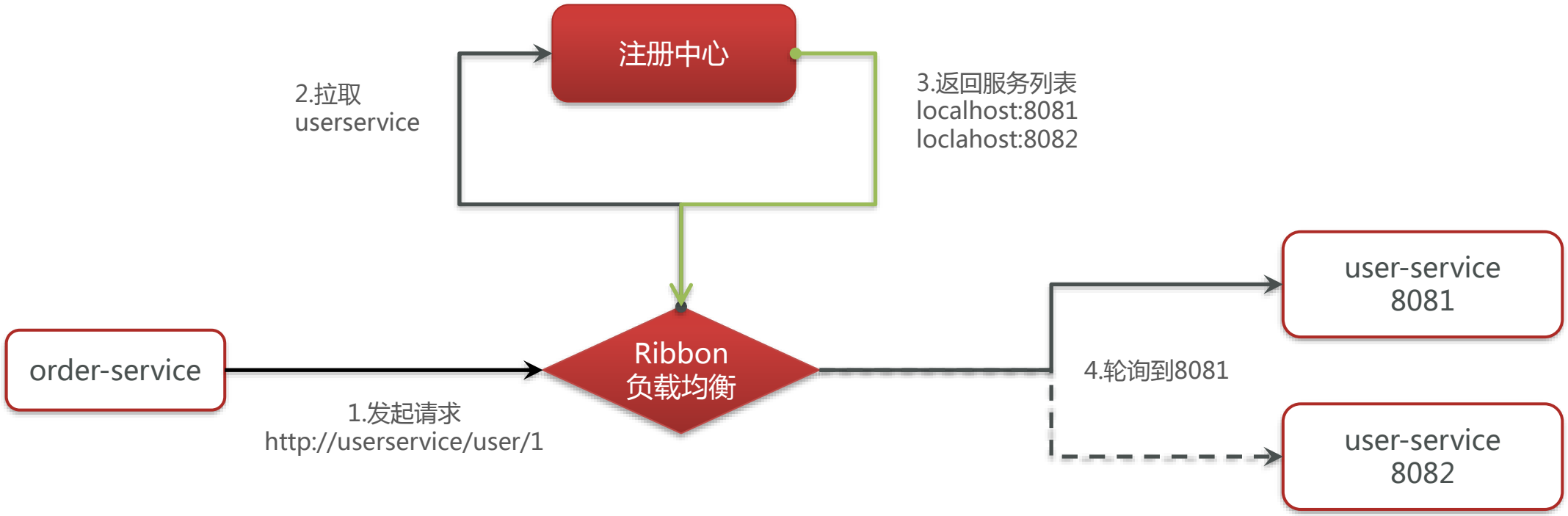
- Nacos与eureka的共同点（注册中心）
  - ① 都支持服务注册和服务拉取
  - ② 都支持服务提供者心跳方式做健康检测
- Nacos与Eureka的区别（注册中心）
  - ① Nacos支持服务端主动检测提供者状态：临时实例采用心跳模式，非临时实例采用主动检测模式
  - ② 临时实例心跳不正常会被剔除，非临时实例则不会被剔除
  - ③ Nacos支持服务列表变更的消息推送模式，服务列表更新更及时
  - ④ Nacos集群默认采用AP方式，当集群中存在非临时实例时，采用CP模式；Eureka采用AP方式
- Nacos还支持了配置中心，eureka则只有注册中心，也是选择使用nacos的一个重要原因

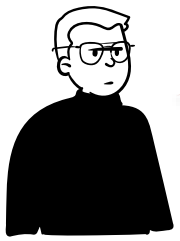


你们项目负载均衡如何实现的？

- 负载均衡 Ribbon，发起远程调用feign就会使用Ribbon
- Ribbon负载均衡策略有哪些？
- 如果想自定义负载均衡策略如何实现？

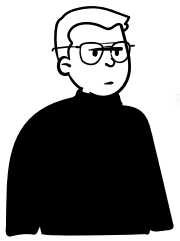
Ribbon负载均衡流程





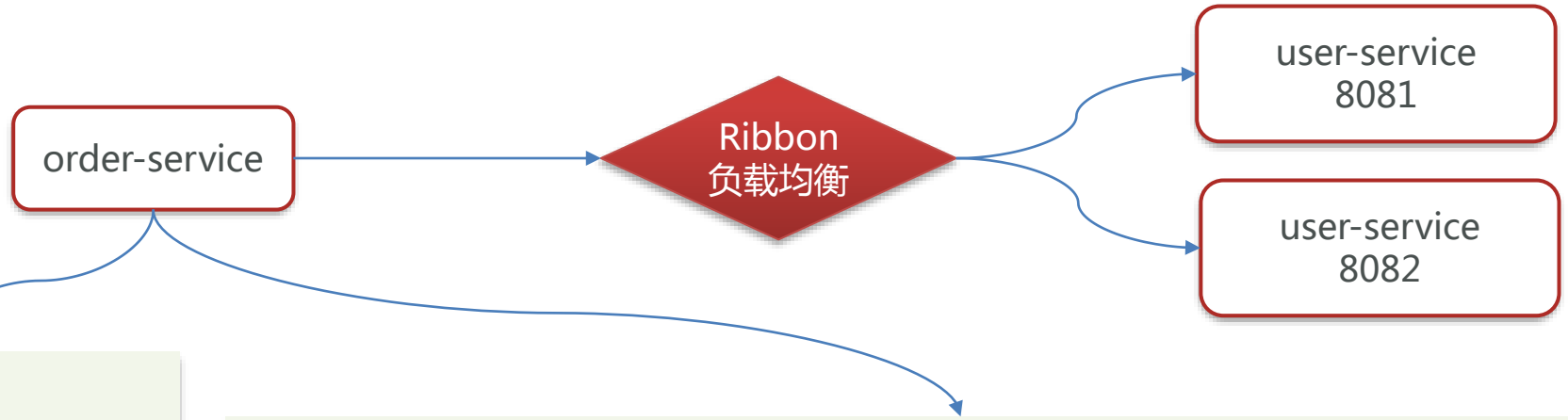
Ribbon负载均衡策略有哪些？

- RoundRobinRule：简单轮询服务列表来选择服务器
- WeightedResponseTimeRule：按照权重来选择服务器，响应时间越长，权重越小
- RandomRule：随机选择一个可用的服务器
- BestAvailableRule：忽略那些短路的服务器，并选择并发数较低的服务器
- RetryRule：重试机制的选择逻辑
- AvailabilityFilteringRule：可用性敏感策略，先过滤非健康的，再选择连接数较小的实例
- 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



如果想自定义负载均衡策略如何实现？

可以自己创建类实现IRule接口，然后再通过配置类或者配置文件配置即可 ，通过定义IRule实现可以修改负载均衡规则，有两种方式：




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

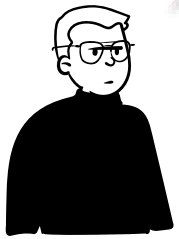
全局生效

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

局部生效

你们项目负载均衡如何实现的？

微服务的负载均衡主要使用了一个组件Ribbon，比如，我们在使用feign远程调用的过程中，底层的负载均衡就是使用了ribbon



Ribbon负载均衡策略有哪些？

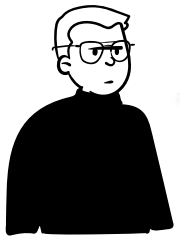
- RoundRobinRule：简单轮询服务列表来选择服务器
- WeightedResponseTimeRule：按照权重来选择服务器，响应时间越长，权重越小
- RandomRule：随机选择一个可用的服务器
- ZoneAvoidanceRule：区域敏感策略，以区域可用的服务器为基础进行服务器的选择。使用Zone对服务器进行分类，这个Zone可以理解为一个机房、一个机架等。而后再对Zone内的多个服务做轮询(默认)

如果想自定义负载均衡策略如何实现？

提供了两种方式：

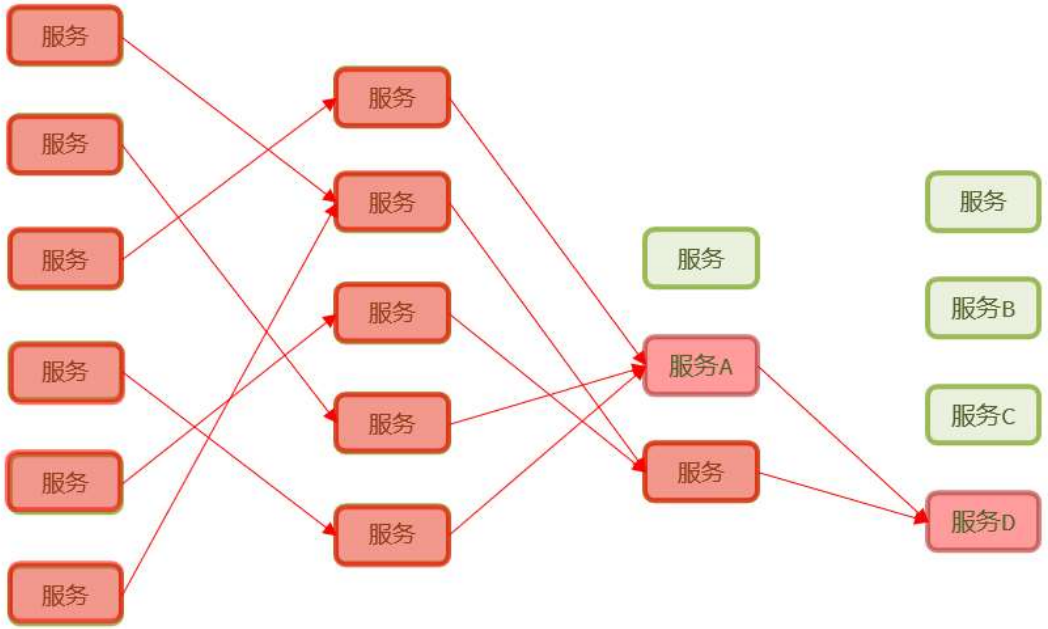
1，创建类实现IRule接口，可以指定负载均衡策略（全局）

2，在客户端的配置文件中，可以配置某一个服务调用的负载均衡策略（局部）



什么是服务雪崩，怎么解决这个问题？

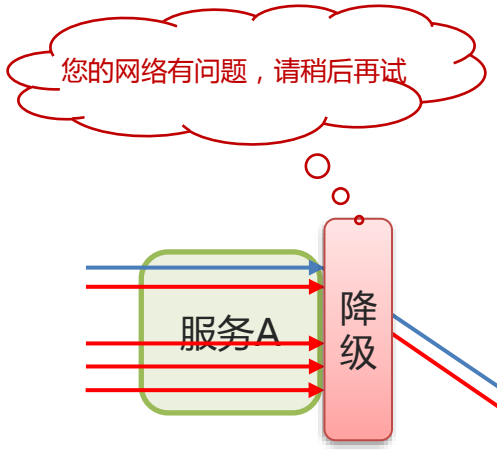
- 什么是服务雪崩？
- 熔断降级（解决）　Hystix 服务熔断降级
- 限流（预防）



雪崩：一个服务失败，导致整条链路的服务都失败的情形

服务降级

服务降级是服务自我保护的一种方式，或者保护下游服务的一种方式，用于确保服务不会受请求突增影响变得不可用，确保服务不会崩溃



如果降级太多，则会触发熔断机制

```java
@FeignClient(value = "leadnews-article",fallback = IArticleClientFallback.class)
public interface IArticleClient {

    @PostMapping("/api/v1/article/save")
    public ResponseResult saveArticle(@RequestBody ArticleDto dto);
}
```

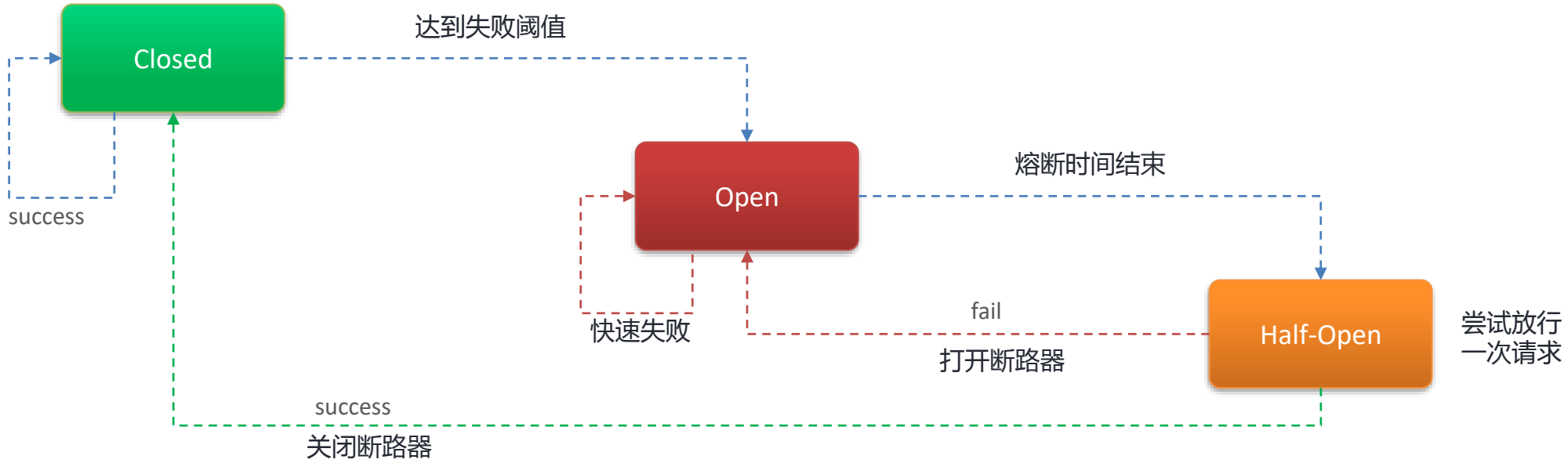
```java
@Component
public class IArticleClientFallback implements IArticleClient {
    @Override
    public ResponseResult saveArticle(ArticleDto dto)  {
        return ResponseResult.errorResult(AppHttpCodeEnum.SERVER_ERROR,"获取数据失败");
    }
}
```

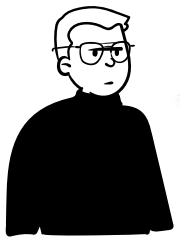
/api/v1/article/save

服务熔断

Hystrix 熔断机制，用于监控微服务调用情况， 默认是关闭的，如果需要开启需要在引导类上添加注解：@EnableCircuitBreaker

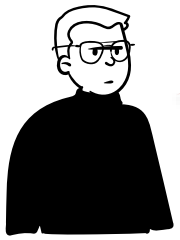
如果检测到 10 秒内请求的失败率超过 50%，就触发熔断机制。之后每隔 5 秒重新尝试请求微服务，如果微服务不能响应，继续走熔断机制。如果微服务可达，则关闭熔断机制，恢复正常请求





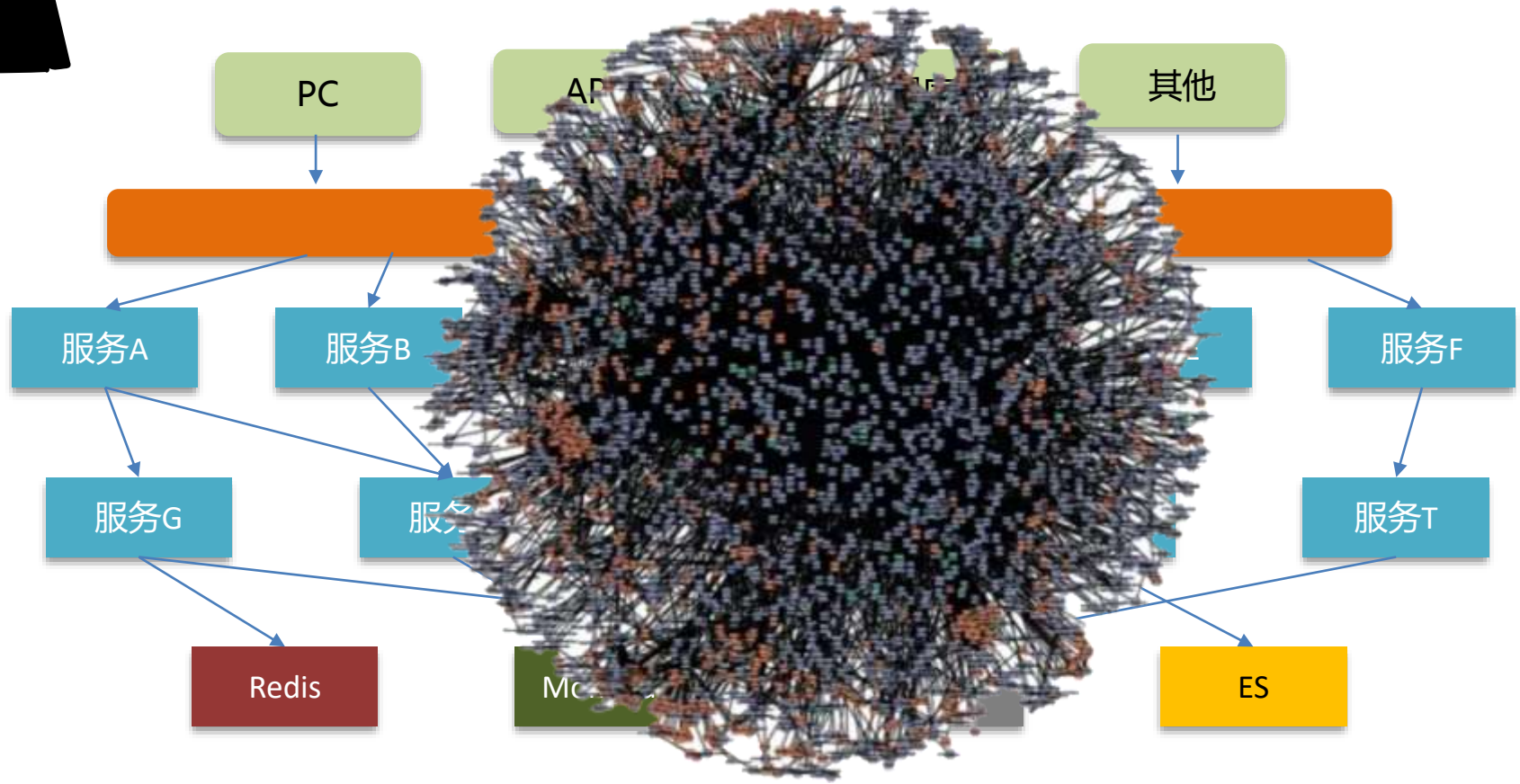
什么是服务雪崩，怎么解决这个问题？

- 服务雪崩：一个服务失败，导致整条链路的服务都失败的情形
- 服务降级：服务自我保护的一种方式，或者保护下游服务的一种方式，用于确保服务不会受请求突增影响变得不可用，确保服务不会崩溃，一般在实际开发中与feign接口整合，编写降级逻辑
- 服务熔断：默认关闭，需要手动打开，如果检测到 10 秒内请求的失败率超过 50%，就触发熔断机制。之后每隔 5 秒重新尝试请求微服务，如果微服务不能响应，继续走熔断机制。如果微服务可达，则关闭熔断机制，恢复正常请求



你们的微服务是怎么监控的？

为什么需要监控？



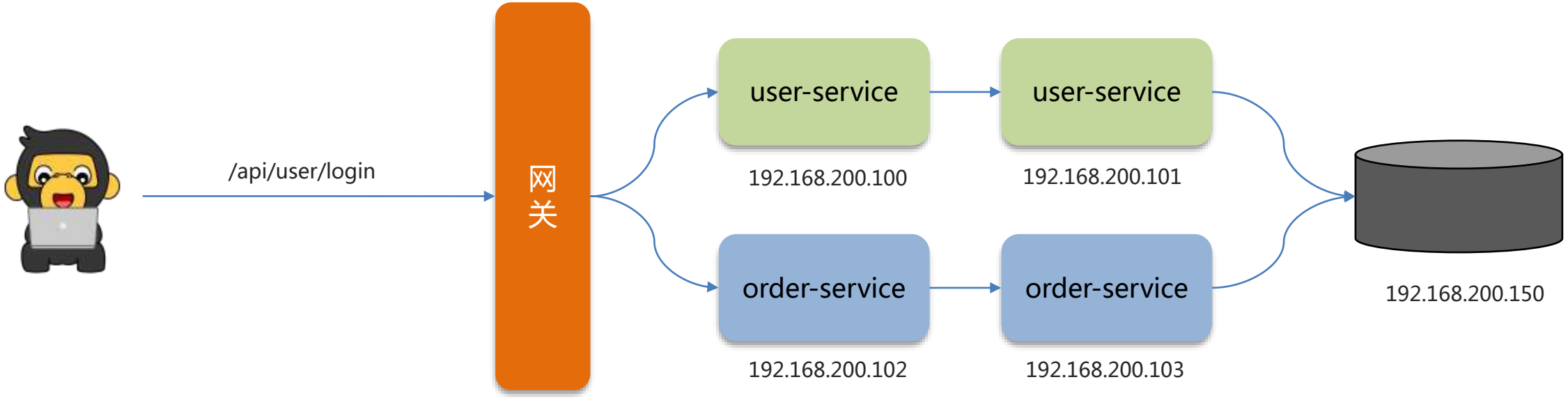
- 问题定位
- 性能分析
- 服务关系
- 服务告警

- Springboot-admin
- prometheus+Grafana
- zipkin
- skywalking

链路追踪工具

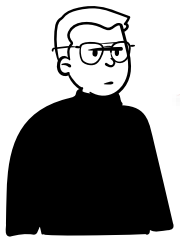
skywalking

一个分布式系统的应用程序性能监控工具（ Application Performance Managment ），提供了完善的链路追踪能力，apache的顶级项目（前华为产品经理吴晟主导开源）



- 服务（service）：业务资源应用系统（微服务）
- 端点（endpoint）：应用系统对外暴露的功能接口（接口）
- 实例（instance）：物理机

skywalking的详细部署和使用请查看今天讲义



你们的微服务是怎么监控的？

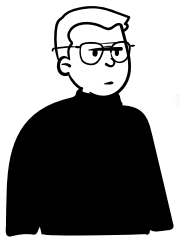
我们项目中采用的skywalking进行监控的

1，skywalking主要可以监控接口、服务、物理实例的一些状态。特别是在压测的时候可以看到众多服务中哪些服务和接口比较慢，我们可以针对性的分析和优化。

2，我们还在skywalking设置了告警规则，特别是在项目上线以后，如果报错，我们分别设置了可以给相关负责人发短信和发邮件，第一时间知道项目的bug情况，第一时间修复

业务相关


业务相关
限流
漏桶算法
令牌桶算法
分布式事务
分布式理论CAP、BASE
分布式事务解决方案
seata
分布式服务接口幂等
分布式任务调度
xxl-job

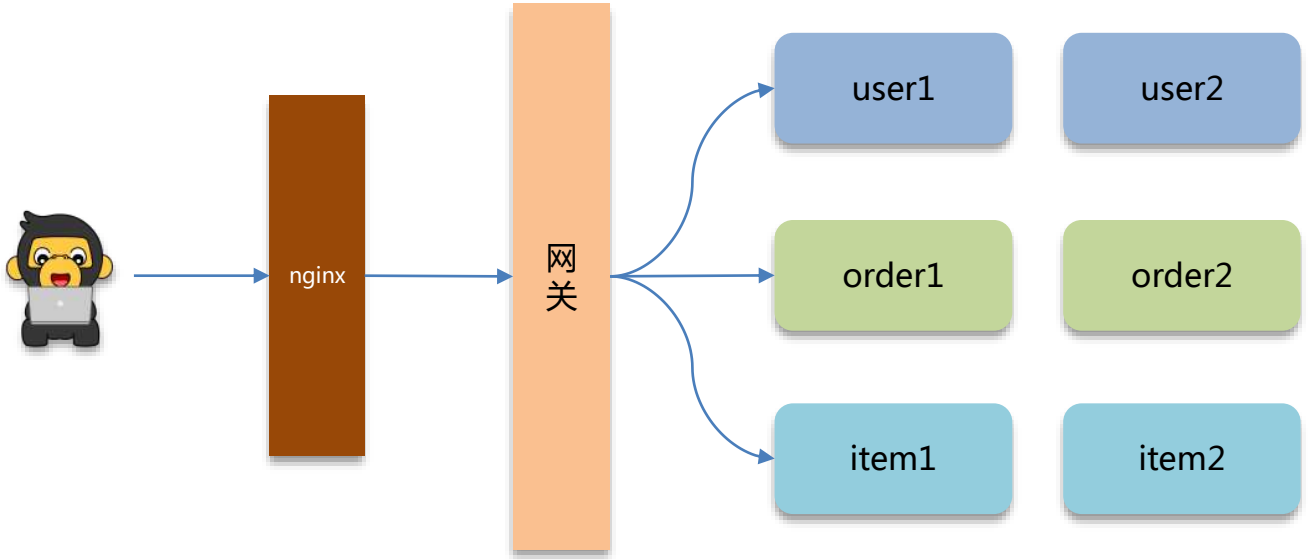


你们项目中有没有做过限流？怎么做的？

为什么要限流？

1，并发的确大（突发流量）

2，防止用户恶意刷接口



限流的实现方式：

- Tomcat：可以设置最大连接数
- Nginx，漏桶算法
- 网关，令牌桶算法
- 自定义拦截器

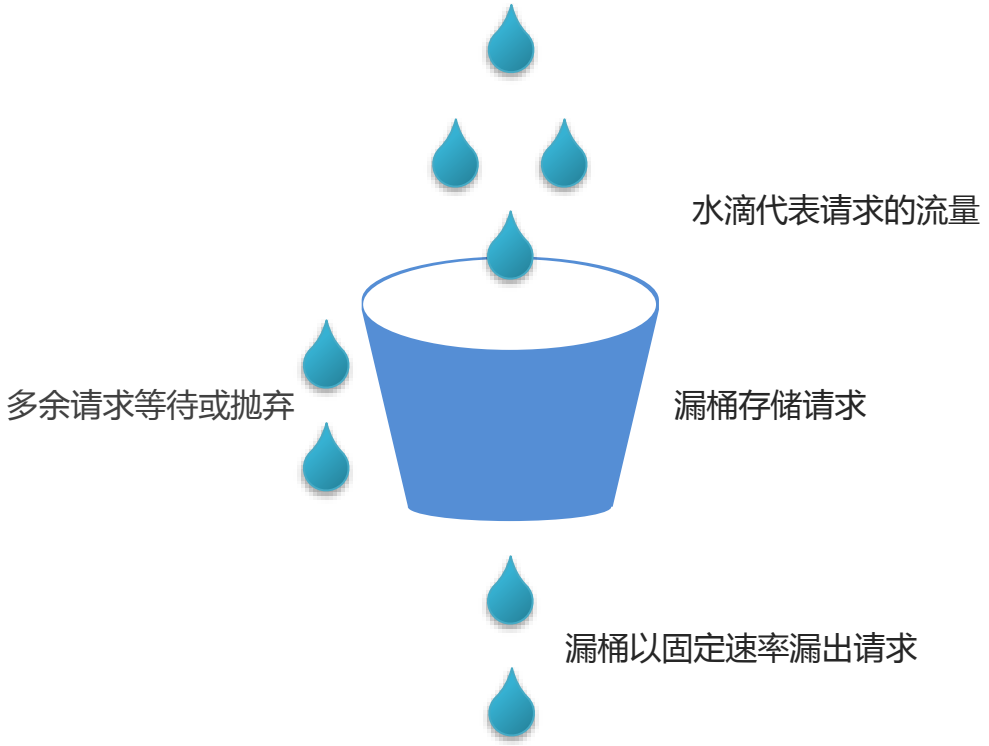
```
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           maxThreads="150"
           redirectPort="8443" />
```

可以设置最大连接数

Nginx限流

控制速率（突发流量）

```
http {
    limit_req_zone $binary_remote_addr zone=service1RateLimit:10m rate=10r/s
    server {
        listen       80;
        server_name  localhost;
        location / {
            limit_req_zone = service1RateLimit burst=20 nodelay;
            proxy_pass http://targetserver;
        }
    }
}
```



- 语法 :limit_req_zone key zone rate
- key:定义限流对象，binary_remote_addr就是一种key，基于客户端ip限流
- Zone：定义共享存储区来存储访问信息，10m可以存储16wip地址访问信息
- Rate：最大访问速率，rate=10r/s 表示每秒最多请求10个请求
- burst=20：相当于桶的大小
- Nodelay：快速处理

Nginx限流

控制并发连接数

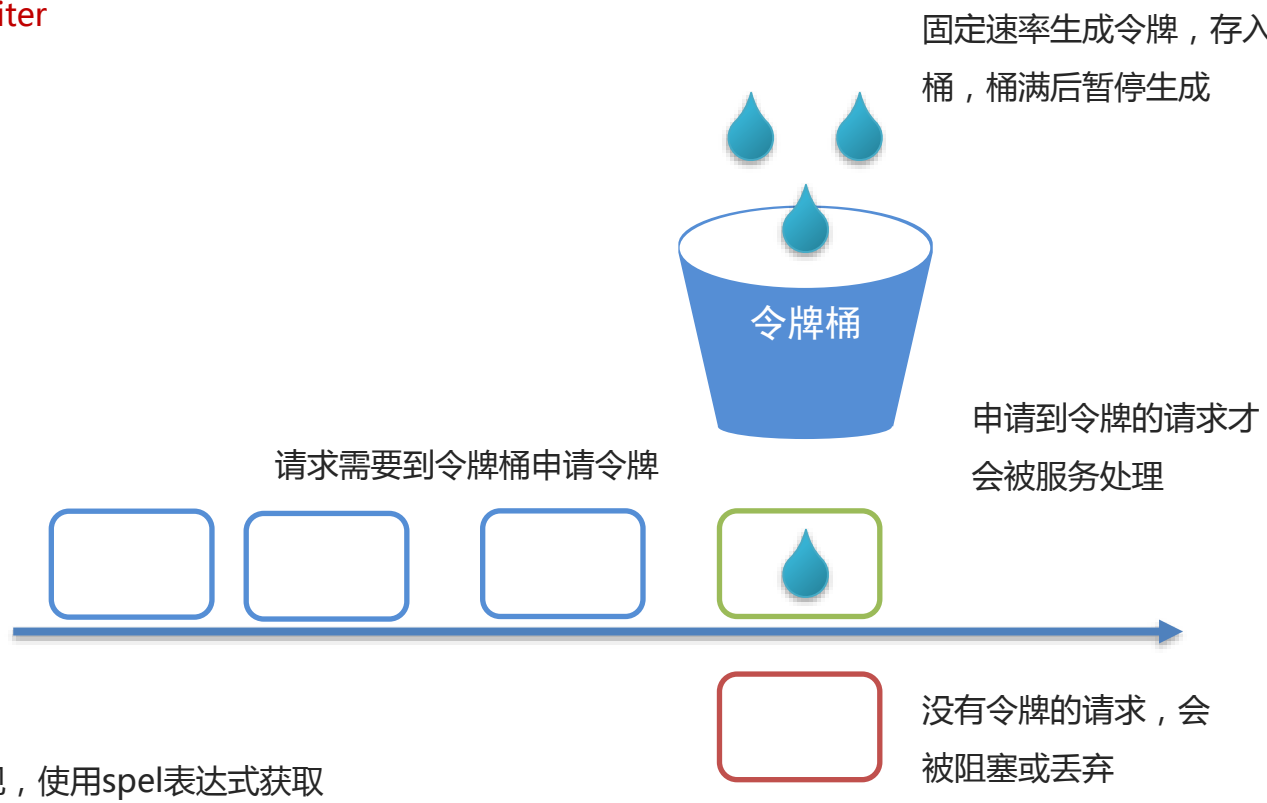
```
http {
    limit_conn_zone $binary_remote_addr zone=perip:10m;
    limit_conn_zone $server_name zone=perserver:10m;
    server {
        listen       80;
        server_name  localhost;
        location / {
            ...
            limit_conn perip 20;
            limit_conn perserver 100;
            proxy_pass http://targetserver;
        }
    }
}
```

- limit_conn perip 20：对应的key是 $binary_remote_addr，表示限制单个IP同时最多能持有20个连接。
- limit_conn perserver 100：对应的key是 $server_name，表示虚拟主机(server) 同时能处理并发连接的总数。

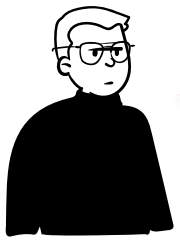
网关限流

yml配置文件中，微服务路由设置添加局部过滤器RequestRateLimiter

```yaml
- id: gateway-consumer
  uri: lb://GATEWAY-CONSUMER
  predicates:
  - Path=/order/**
  filters:
    - name: RequestRateLimiter
      args:
        # 使用SpEL从容器中获取对象
        key-resolver: '#{@pathKeyResolver}'
        # 令牌桶每秒填充平均速率
        redis-rate-limiter.replenishRate: 1
        # 令牌桶的上限
        redis-rate-limiter.burstCapacity: 3
```



- key-resolver ：定义限流对象（ ip 、路径、参数），需代码实现，使用spel表达式获取
- replenishRate ：令牌桶每秒填充平均速率。
- urstCapacity ：令牌桶总容量。



你们项目中有没有做过限流？怎么做的？

1，先来介绍业务，什么情况下去做限流，需要说明QPS具体多少

- 我们当时有一个活动，到了假期就会抢购优惠券，QPS最高可以达到2000，平时10-50之间，为了应对突发流量，需要做限流
- 常规限流，为了防止恶意攻击，保护系统正常运行，我们当时系统能够承受最大的QPS是多少（压测结果）

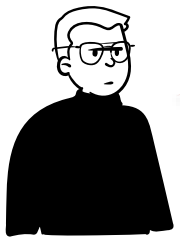
2，nginx限流

- 控制速率（突发流量），使用的漏桶算法来实现过滤，让请求以固定的速率处理请求，可以应对突发流量
- 控制并发数，限制单个ip的链接数和并发链接的总数

3，网关限流

- 在spring cloud gateway中支持局部过滤器RequestRateLimiter来做限流，使用的是令牌桶算法
- 可以根据ip或路径进行限流，可以设置每秒填充平均速率，和令牌桶总容量

限流常见的算法有哪些呢？



解释一下CAP和BASE

- 分布式事务方案的指导
- 分布式系统设计方向
- 根据业务指导使用正确的技术选择

- CAP定理
- BASE理论

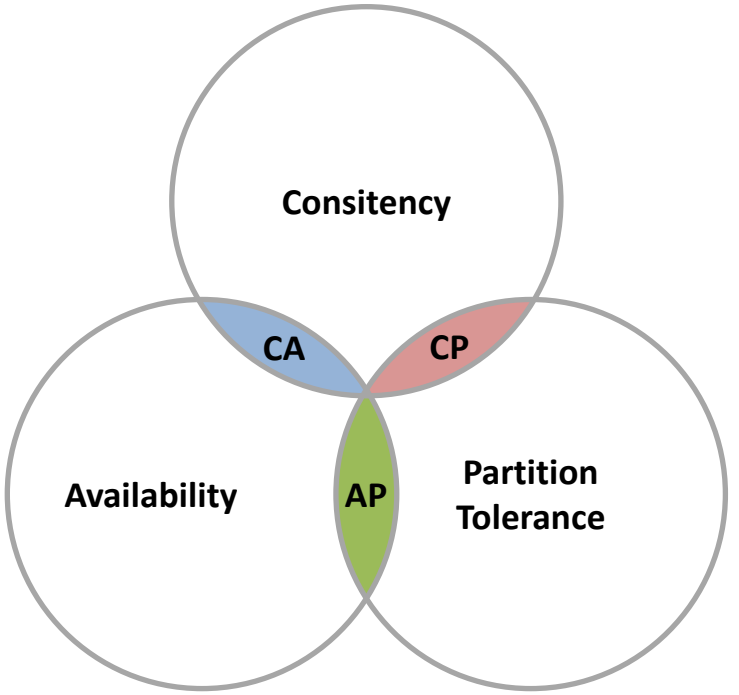
CAP定理

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

- Consistency（一致性）
- Availability（可用性）
- Partition tolerance （分区容错性）

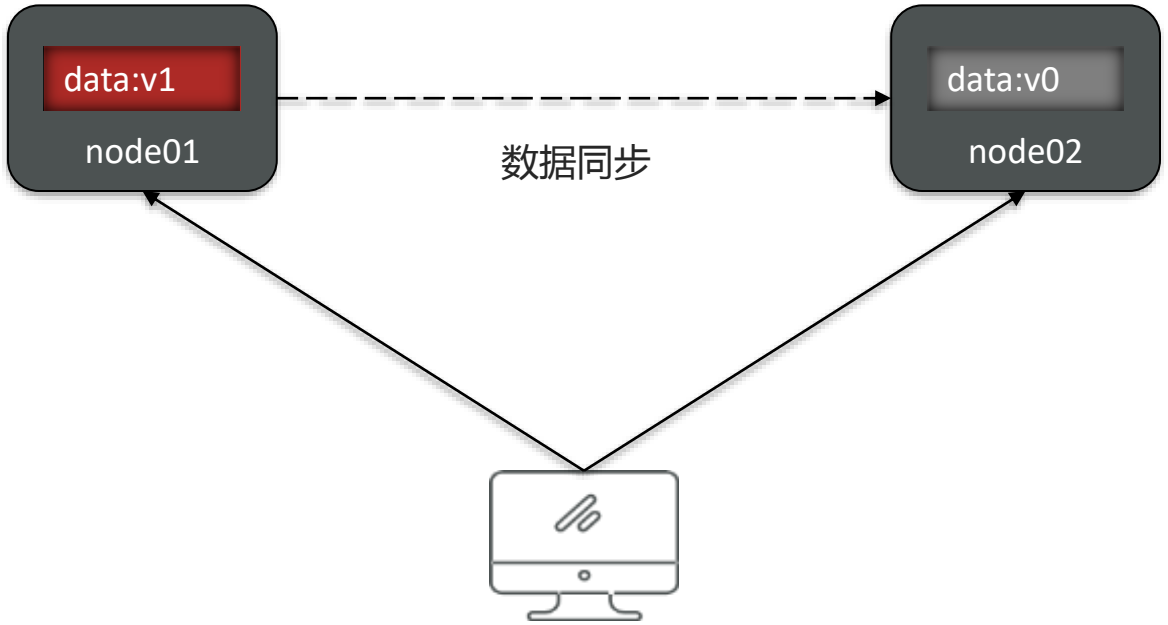
Eric Brewer 说，分布式系统无法同时满足这三个指标。

这个结论就叫做 CAP 定理。



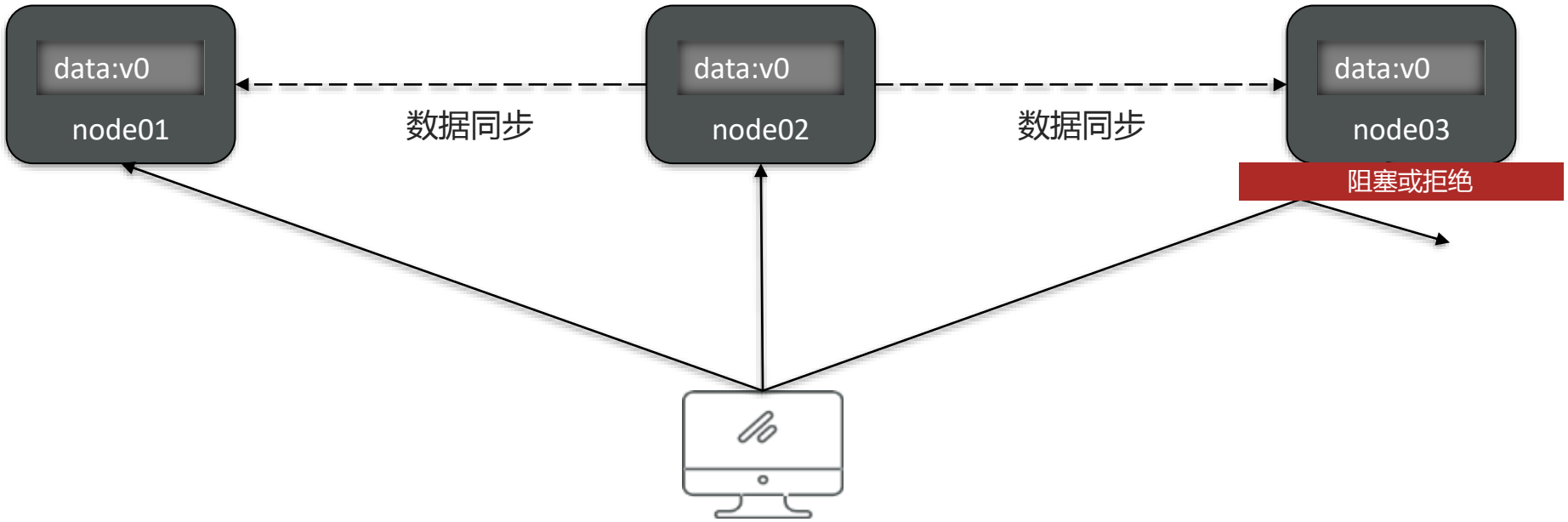
CAP定理- Consistency

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



CAP定理- Availability

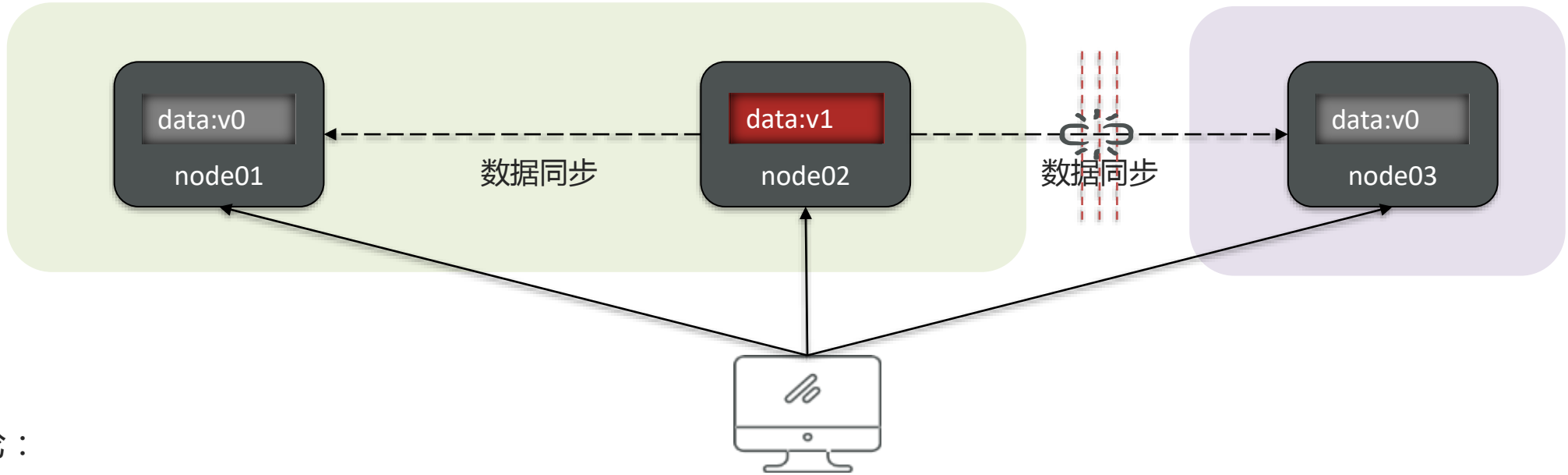
Availability （可用性）：用户访问集群中的任意健康节点，必须能得到响应，而不是超时或拒绝



CAP定理-Partition tolerance

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

Tolerance（容错）：在集群出现分区时，整个系统也要持续对外提供服务



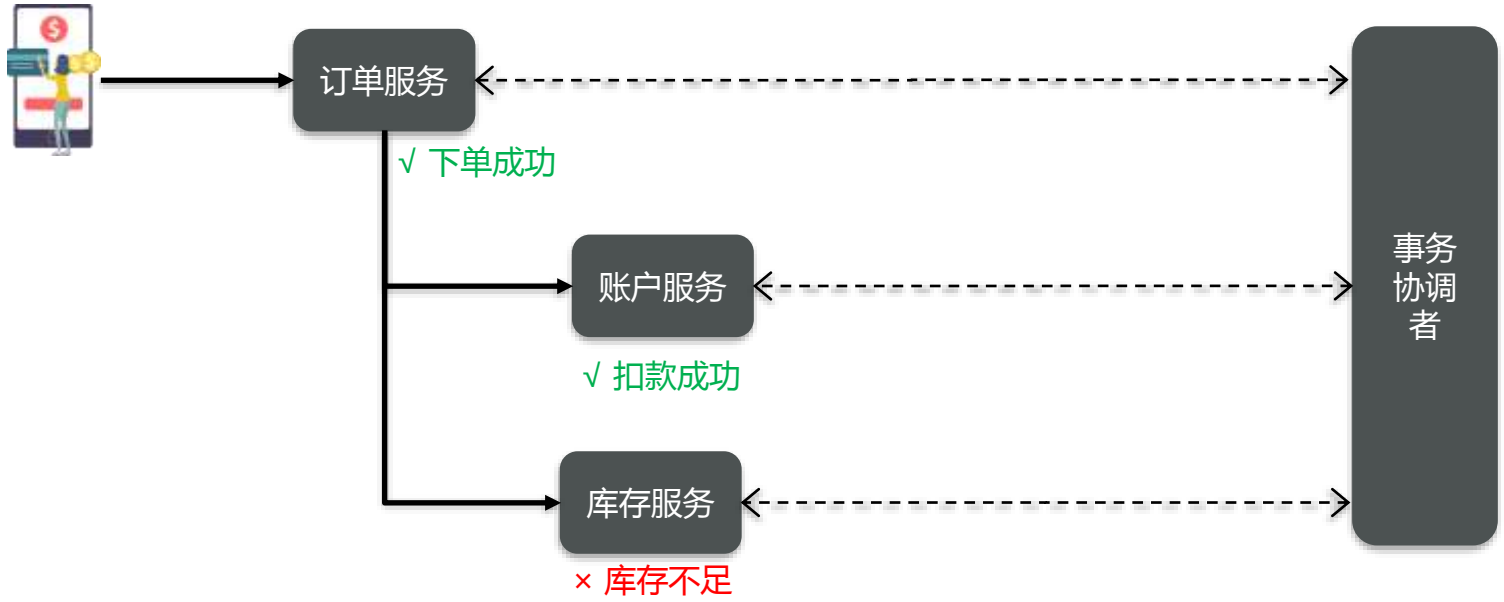
结论：

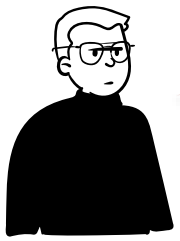
- 分布式系统节点之间肯定是需要网络连接的，分区（P）是必然存在的
- 如果保证访问的高可用性（A）,可以持续对外提供服务，但不能保证数据的强一致性--> AP
- 如果保证访问的数据强一致性（C）,就要放弃高可用性 --> CP

BASE理论

BASE理论是对CAP的一种解决思路，包含三个思想：

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





解释一下CAP和BASE

- CAP 定理(一致性、可用性、分区容错性)

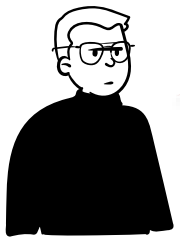
1. 分布式系统节点通过网络连接，一定会出现分区问题（P）
2. 当分区出现时，系统的一致性（C）和可用性（A）就无法同时满足

- BASE理论

1. 基本可用
2. 软状态
3. 最终一致

- 解决分布式事务的思想和模型：

1. 最终一致思想：各分支事务分别执行并提交，如果有不一致的情况，再想办法恢复数据（AP）
2. 强一致思想：各分支事务执行完业务不要提交，等待彼此结果。而后统一提交或回滚（CP）



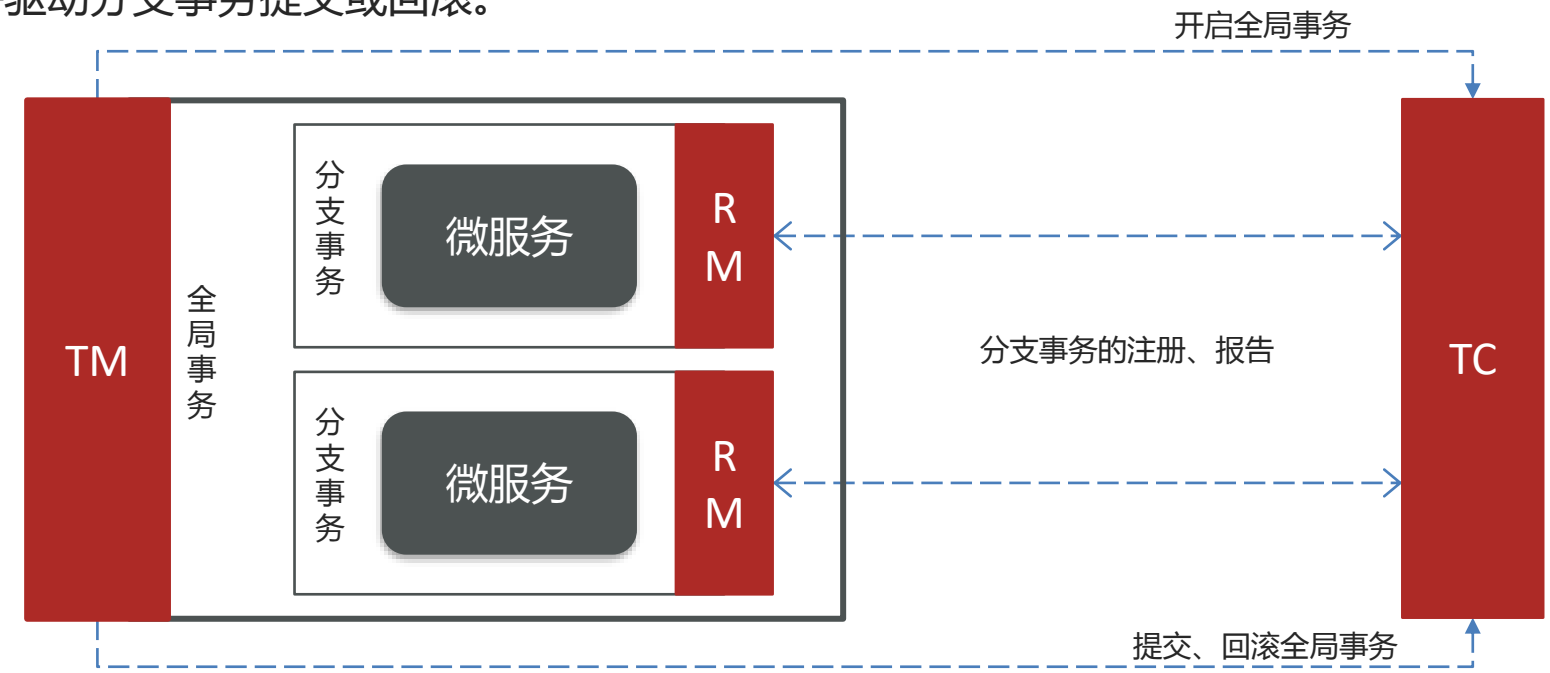
你们采用哪种分布式事务解决方案？

- 简历上写的是微服务项目
- Seata框架(XA、AT、TCC)
- MQ

Seata架构

Seata事务管理中有三个重要的角色：

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



seata的XA模式

RM一阶段的工作：

① 注册分支事务到TC

② 执行分支业务sql但不提交

③ 报告执行状态到TC

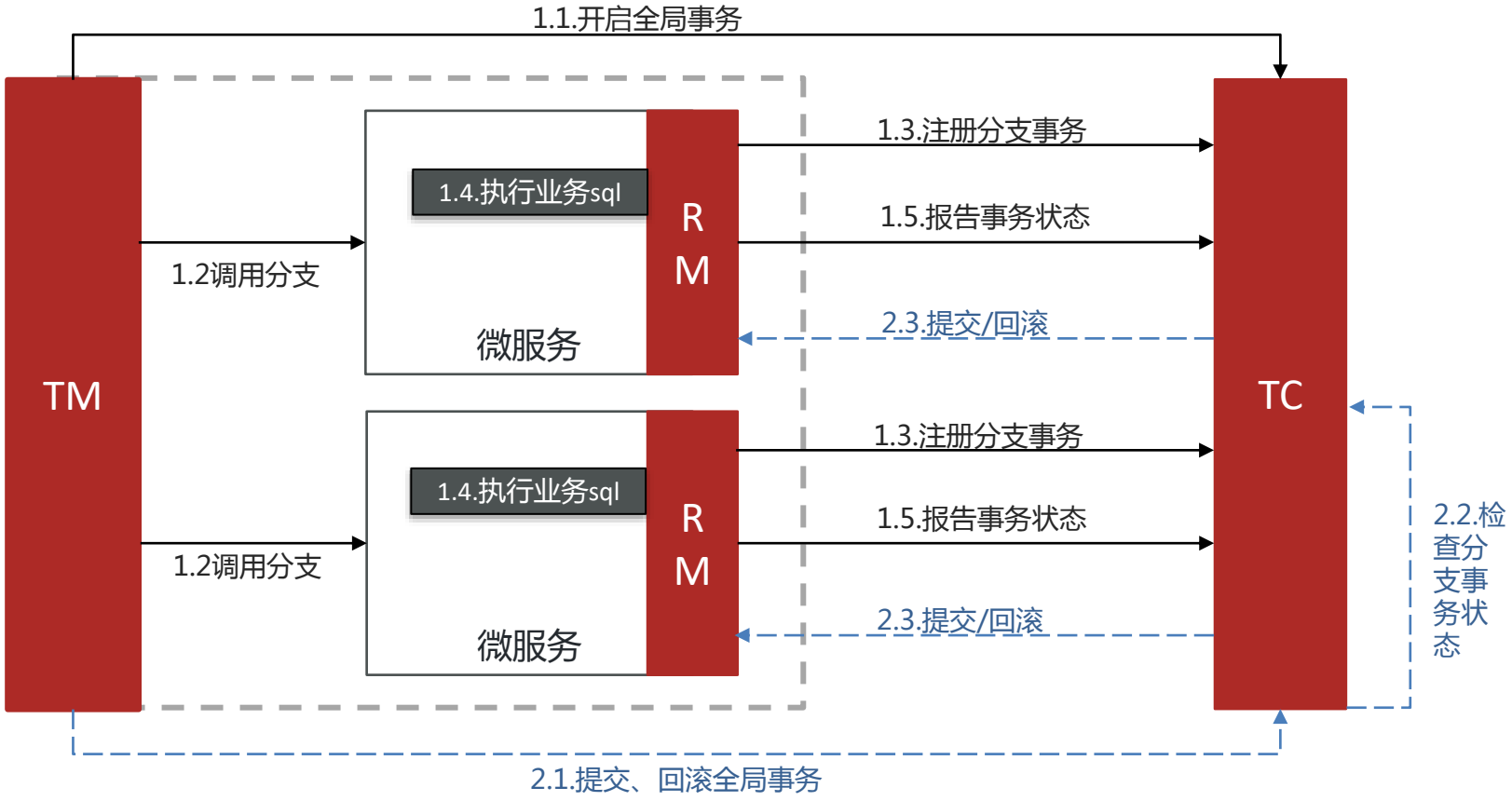
TC二阶段的工作：

- TC检测各分支事务执行状态

a. 如果都成功，通知所有RM提交事务
b. 如果有失败，通知所有RM回滚事务

RM二阶段的工作：

- 接收TC指令，提交或回滚事务



AT模式原理

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

阶段一RM的工作：

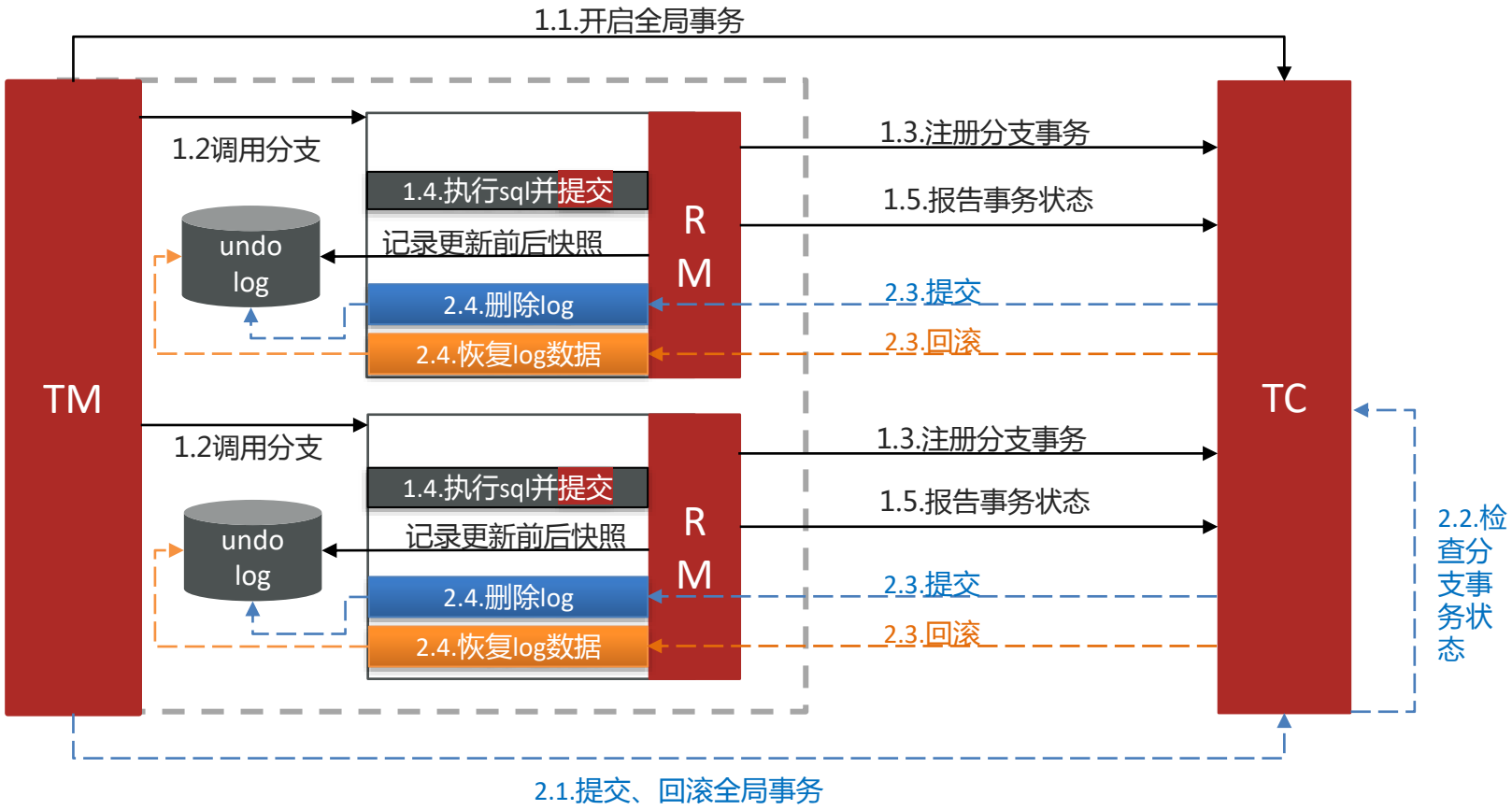
- 注册分支事务
- 记录undo-log（数据快照）
- 执行业务sql并提交
- 报告事务状态

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

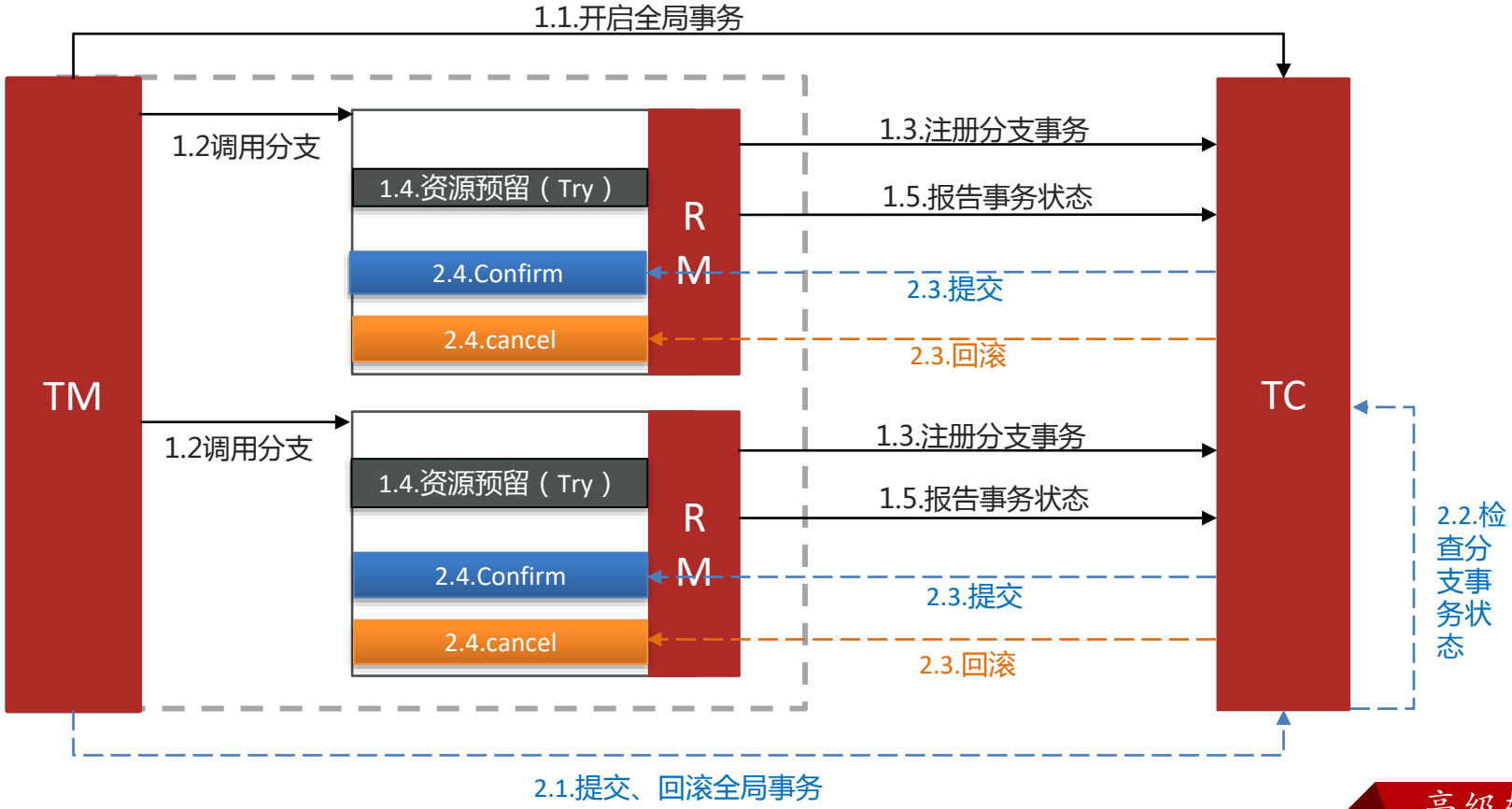


TCC模式原理

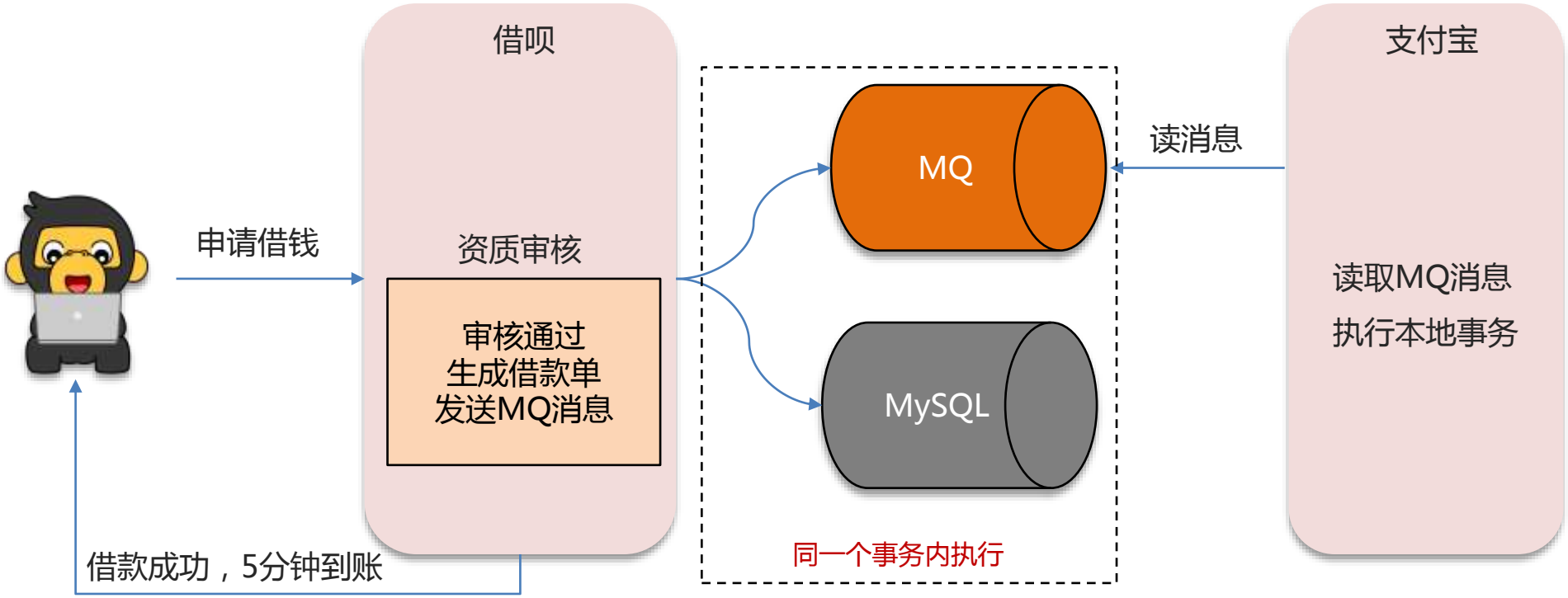
1、Try：资源的检测和预留；

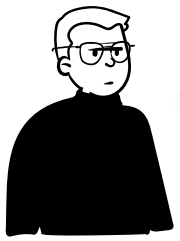
2、Confirm：完成资源操作业务；要求 Try 成功 Confirm 一定要能成功。

3、Cancel：预留资源释放，可以理解为try的反向操作。



MQ分布式事务


借呗
支付宝
申请借钱
资质审核
审核通过
生成借款单
发送MQ消息
MQ
MySQL
读消息
读取MQ消息
执行本地事务
同一个事务内执行
借款成功，5分钟到账

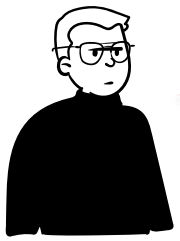


你们采用哪种分布式事务解决方案？

- 简历上写的微服务，只要是发生了多个服务之间的写操作，都需要进行分布式事务控制
- 描述项目中采用的哪种方案（seata | MQ）

1. seata的XA模式，CP，需要互相等待各个分支事务提交，可以保证强一致性，性能差　银行业务
2. seata的AT模式，AP，底层使用undo log 实现，性能好　互联网业务
3. seata的TCC模式，AP，性能较好，不过需要人工编码实现　银行业务
4. MQ模式实现分布式事务，在A服务写数据的时候，需要在同一个事务内发送消息到另外一个事务，异步，性能最好　互联网业务

四选一



分布式服务的接口幂等性如何设计？

幂等: 多次调用方法或者接口不会改变业务状态，可以保证重复调用的结果和单次调用的结果一致。

需要幂等场景

- 用户重复点击(网络波动)
- MQ消息重复
- 应用使用失败或超时重试机制

接口幂等

基于RESTful API的角度对部分常见类型请求的幂等性特点进行分析

请求方式	说明
GET	查询操作，天然幂等
POST	新增操作，请求一次与请求多次造成的结果不同，不是幂等的
PUT	更新操作，如果是以绝对值更新，则是幂等的。如果是通过增量的方式更新，则不是幂等的
DELETE	删除操作，根据唯一值删除，是幂等的

```
update t_item set money = 500 where id = 1;
update t_item set money = money + 500 where id = 1;
```

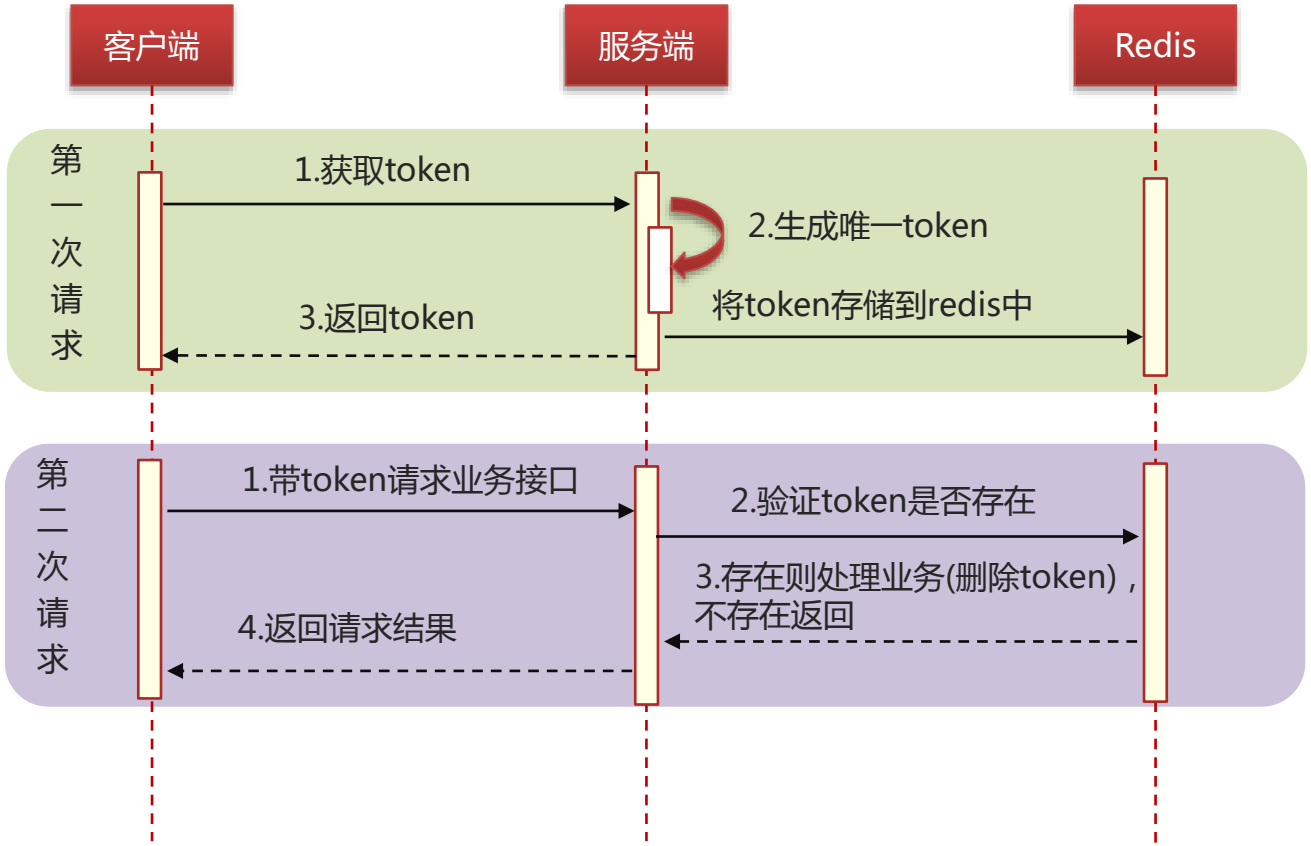
数据库唯一索引 新增

token+redis 新增、修改

分布式锁 新增、修改

token+redis

创建商品、提交订单、转账、支付等操作



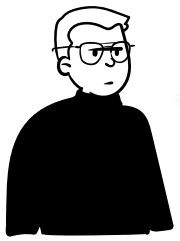
生成token

带token验证

分布式锁

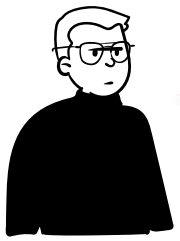
```
public void saveOrder(Item item) throws InterruptedException {
   //获取锁（重入锁），执行锁的名称
  RLock lock = redissonClient.getLock("heimalock");
   //尝试获取锁，参数分别是：获取锁的最大等待时间（期间会重试），锁自动释放时间，时间单位
  boolean isLock = lock.tryLock(10, TimeUnit.SECONDS);
    try {
       //判断是否获取成功
    if (!isLock) {
          log.info("下单操作获取锁失败,order:{}",item);
          throw new RuntimeException("新增或修改失败");
       }
       //下单操作

    } finally {
       //释放锁
    lock.unlock();
    }
}
```

- 快速失败（抢不到锁的线程）
- 控制锁的粒度



分布式服务的接口幂等性如何设计？

- 幂等: 多次调用方法或者接口不会改变业务状态，可以保证重复调用的结果和单次调用的结果一致
- 如果是新增数据，可以使用数据库的唯一索引
- 如果是新增或修改数据
  - 分布式锁，性能较低
  - 使用token+redis来实现，性能较好
    - 第一次请求，生成一个唯一token存入redis，返回给前端
    - 第二次请求，业务处理，携带之前的token，到redis进行验证，如果存在，可以执行业务，删除token；如果不存在，则直接返回，不处理业务



你们项目中使用了什么分布式任务调度

xxl-job

首先，还是要描述当时是什么场景用了任务调度

xxl-job解决的问题

- 解决集群任务的重复执行问题
- cron表达式定义灵活
- 定时任务失败了，重试和统计
- 任务量大，分片执行

1. xxl-job路由策略有哪些？
2. xxl-job任务执行失败怎么解决？
3. 如果有大数据量的任务同时都需要执行，怎么解决？

xxl-job路由策略有哪些？

1.FIRST（第一个）：固定选择第一个机器；

2.LAST（最后一个）：固定选择最后一个机器；

3.ROUND（轮询）

4.RANDOM（随机）：随机选择在线的机器；

5.CONSISTENT_HASH（一致性HASH）：每个任务按照Ha

6.LEAST_FREQUENTLY_USED（最不经常使用）：使用频

7.LEAST_RECENTLY_USED（最近最久未使用）：最久未

8.FAILOVER（故障转移）：按照顺序依次进行心跳检测

9.BUSYOVER（忙碌转移）：按照顺序依次进行空闲检测

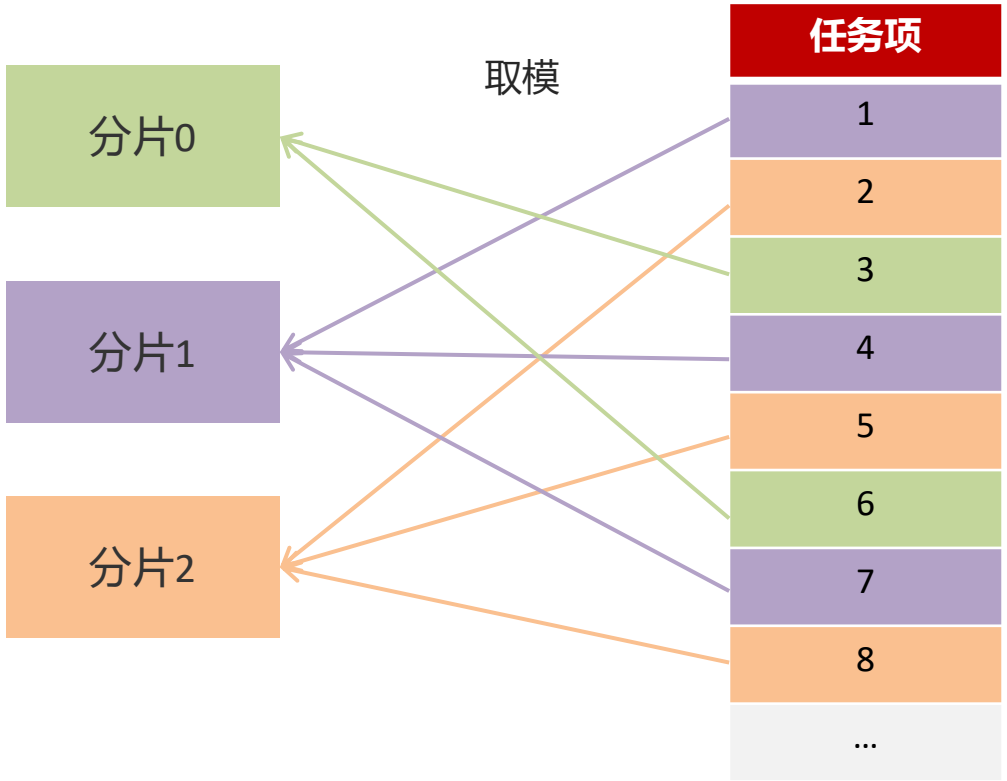
10.SHARDING_BROADCAST(分片广播)：广播触发对应集

xxl-job任务执行失败怎么解决？

故障转移+失败重试，查看日志分析----> 邮件告警

如果有大数据量的任务同时都需要执行，怎么解决？

执行器集群部署时，任务路由策略选择分片广播情况下，一次任务调度将会广播触发对应集群中所有执行器执行一次任务



分片参数

- index：当前分片序号(从0开始)，执行器集群列表中当前执行器的序号；
- total：总分片数，执行器集群的总机器数量；

```
@XxlJob("shadingSample")
public void shardingJobHandler() throws Exception {
    // 分片参数
    int shardIndex = XxlJobHelper.getShardIndex();
    int shardTotal = XxlJobHelper.getShardTotal();
    XxlJobHelper.log("分片参数：当前分片序号 = {}, 总分片数 = {}", shardIndex, shardTotal);
    // 业务逻辑
    List<Integer> list = getList();

    for (Integer integer : list) {
        if(integer % shardTotal == shardIndex){
            System.out.println("第"+shardIndex+"分片执行，执行数据为："+integer);
        }
    }
}
```

xxl-job路由策略有哪些？

xxl-job提供了很多的路由策略，我们平时用的较多就是：轮询、故障转移、分片广播...

xxl-job任务执行失败怎么解决？

- 路由策略选择故障转移，使用健康的实例来执行任务
- 设置重试次数
- 查看日志+邮件告警来通知相关负责人解决

如果有大数据量的任务同时都需要执行，怎么解决？

- 让多个实例一块去执行（部署集群），路由策略分片广播
- 在任务执行的代码中可以获取分片总数和当前分片，按照取模的方式分摊到各个实例执行


黑马程序员
www.itheima.com
传智教育旗下高端IT教育品牌