微服务面试篇

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

目录

Contents

- 分布式事务
- 注册中心
- 远程调用
- 服务保护
- 其它面试题

分布式事务

目录

Contents

- CAP和BASE
- AT模式的脏写问题
- TCC模式
- 最大努力通知

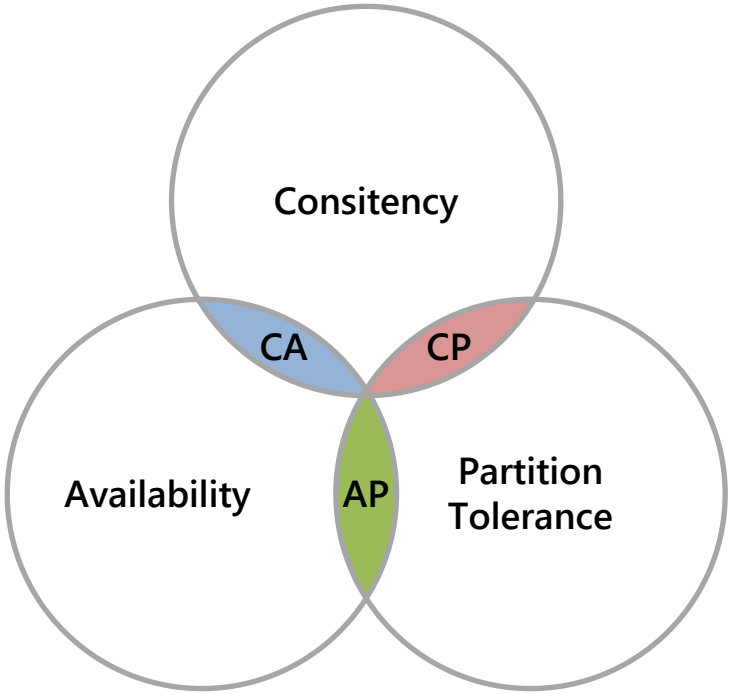
CAP定理

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）

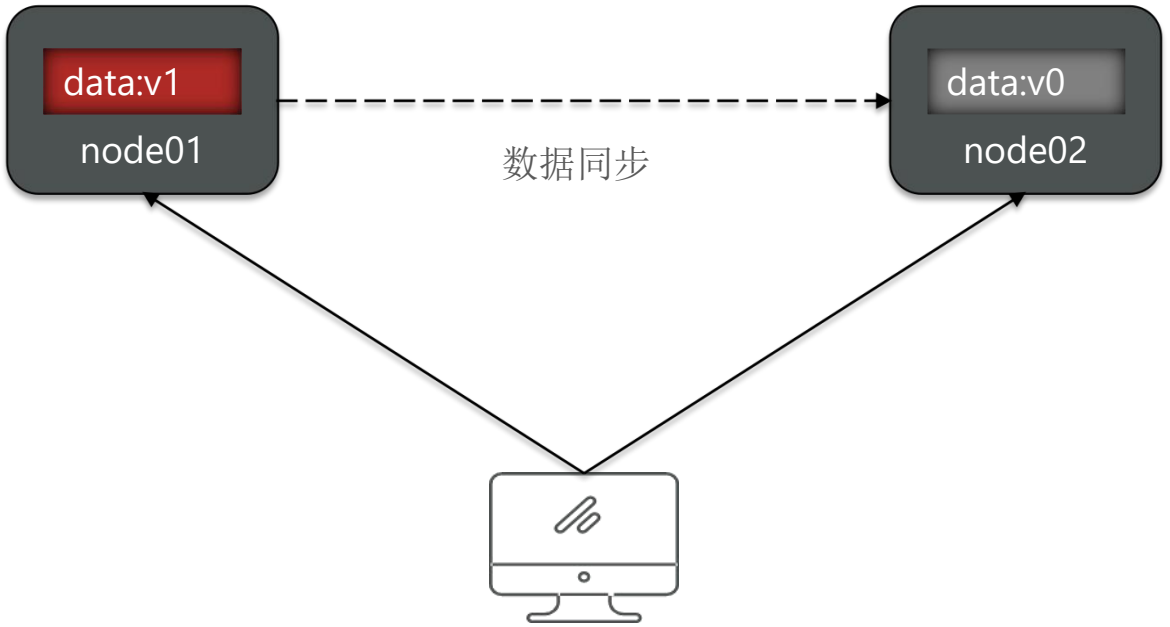
Eric Brewer 说，分布式系统无法同时满足这三个指标。

这个结论就叫做 CAP 定理。



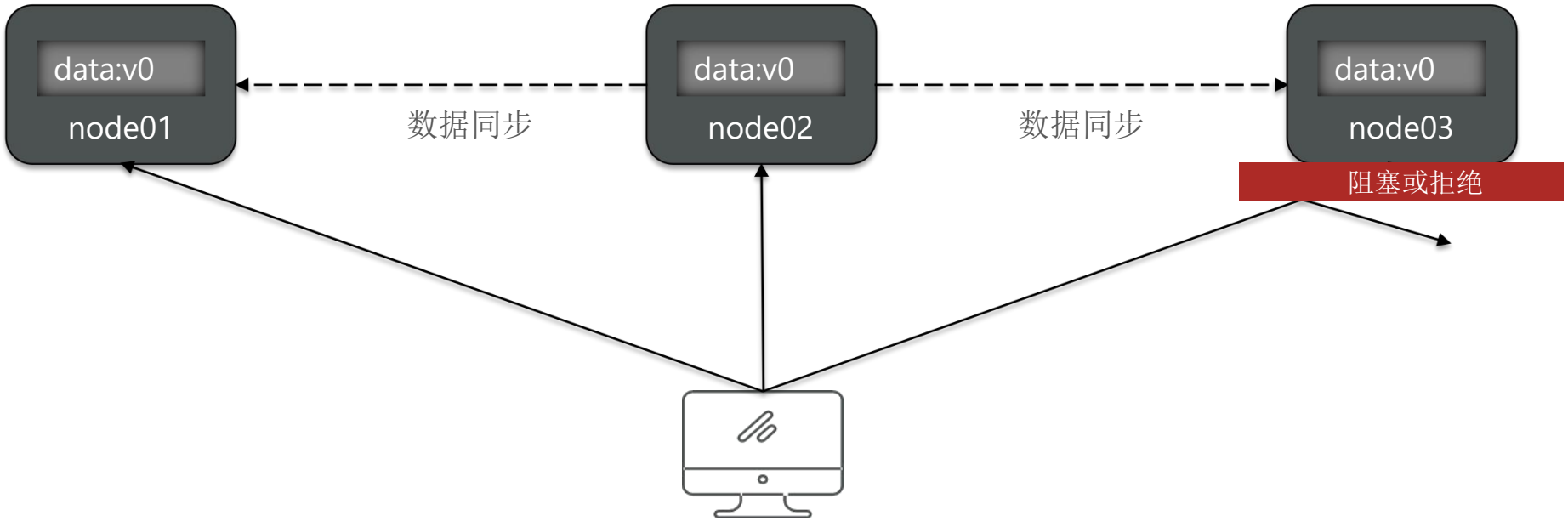
CAP定理- Consistency

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）：用户访问分布式系统时，读或写操作总能成功。只能读不能写，或者只能写不能读，或者两者都不能执行，就说明系统弱可用或不可用。

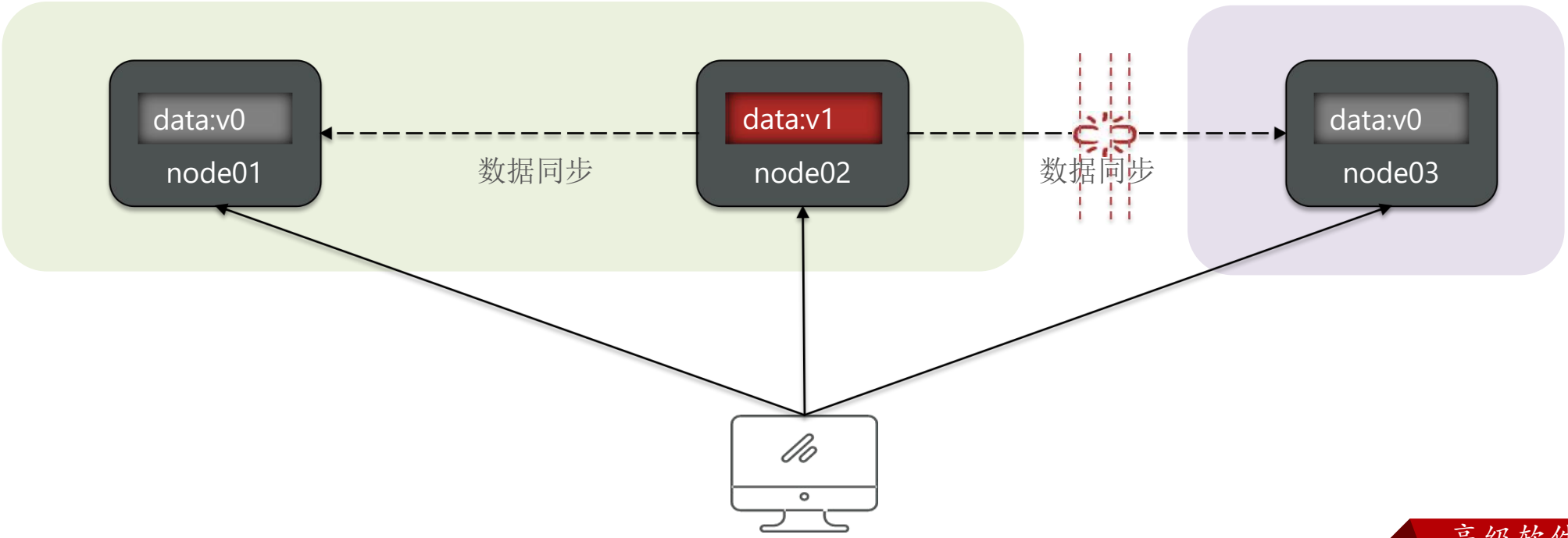


CAP定理-Partition tolerance

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）：系统要能容忍网络分区现象，出现分区时，整个系统也要持续对外提供服务

- 如果此时只允许读，不允许写，满足所有节点一致性。但是牺牲了可用性。符合CP
- 如果此时允许任意读写，满足了可用性。但由于node3无法同步，导致数据不一致，牺牲了一致性。符合AP



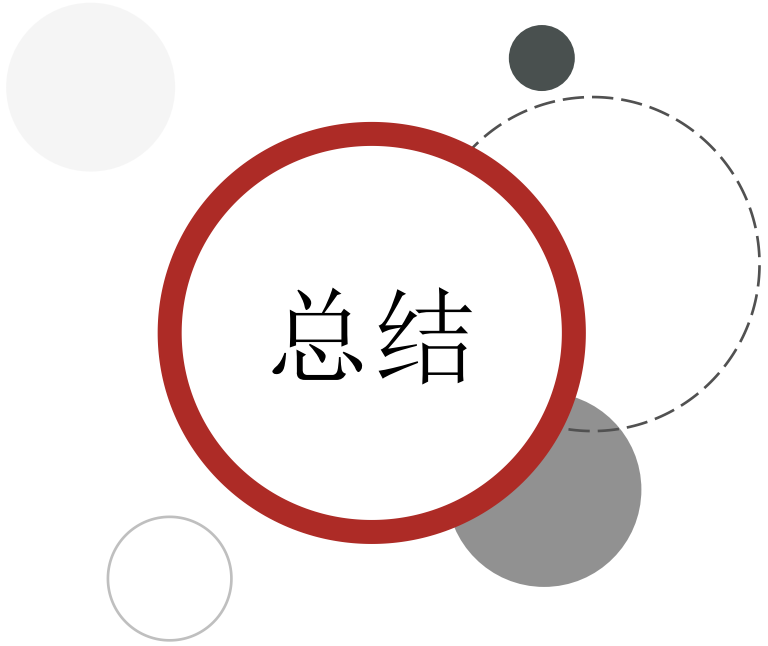
BASE理论

BASE理论是对CAP的一种解决思路，包含三个思想：

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

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

- 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。
- 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。



简述CAP定理内容？

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

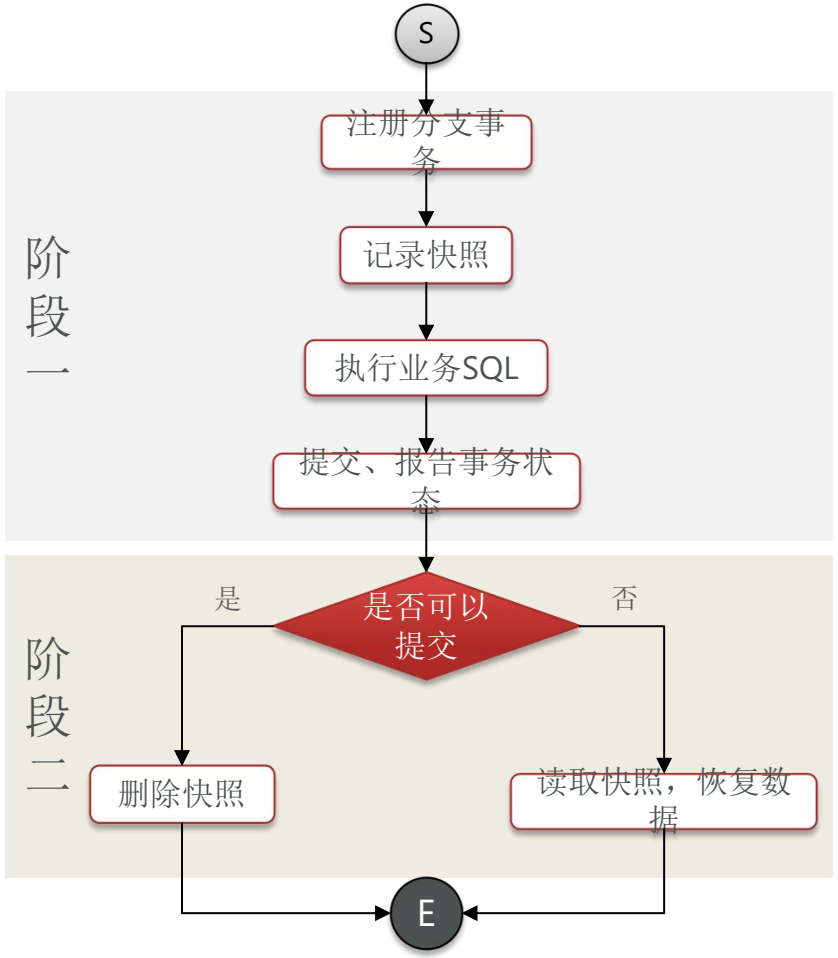
目录

Contents

- CAP和BASE
- AT模式的脏写问题
- TCC模式
- 最大努力通知

AT模式原理

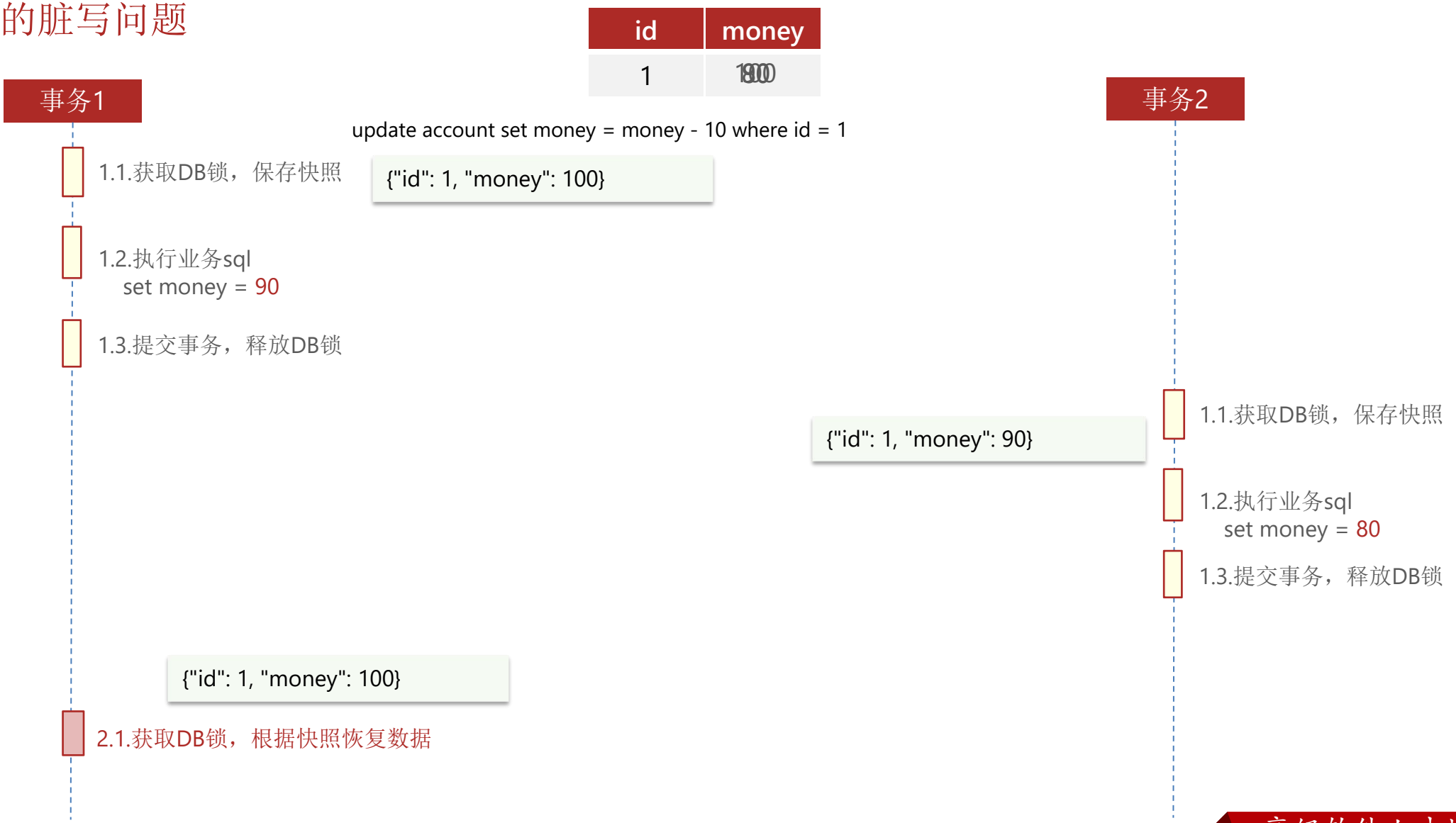
例如，一个分支业务的SQL是这样的：`update tb_account set money = money - 10 where id = 1`



id	money
1	100

```
数据快照：
{
  "id": 1,
  "money": 100
}
```

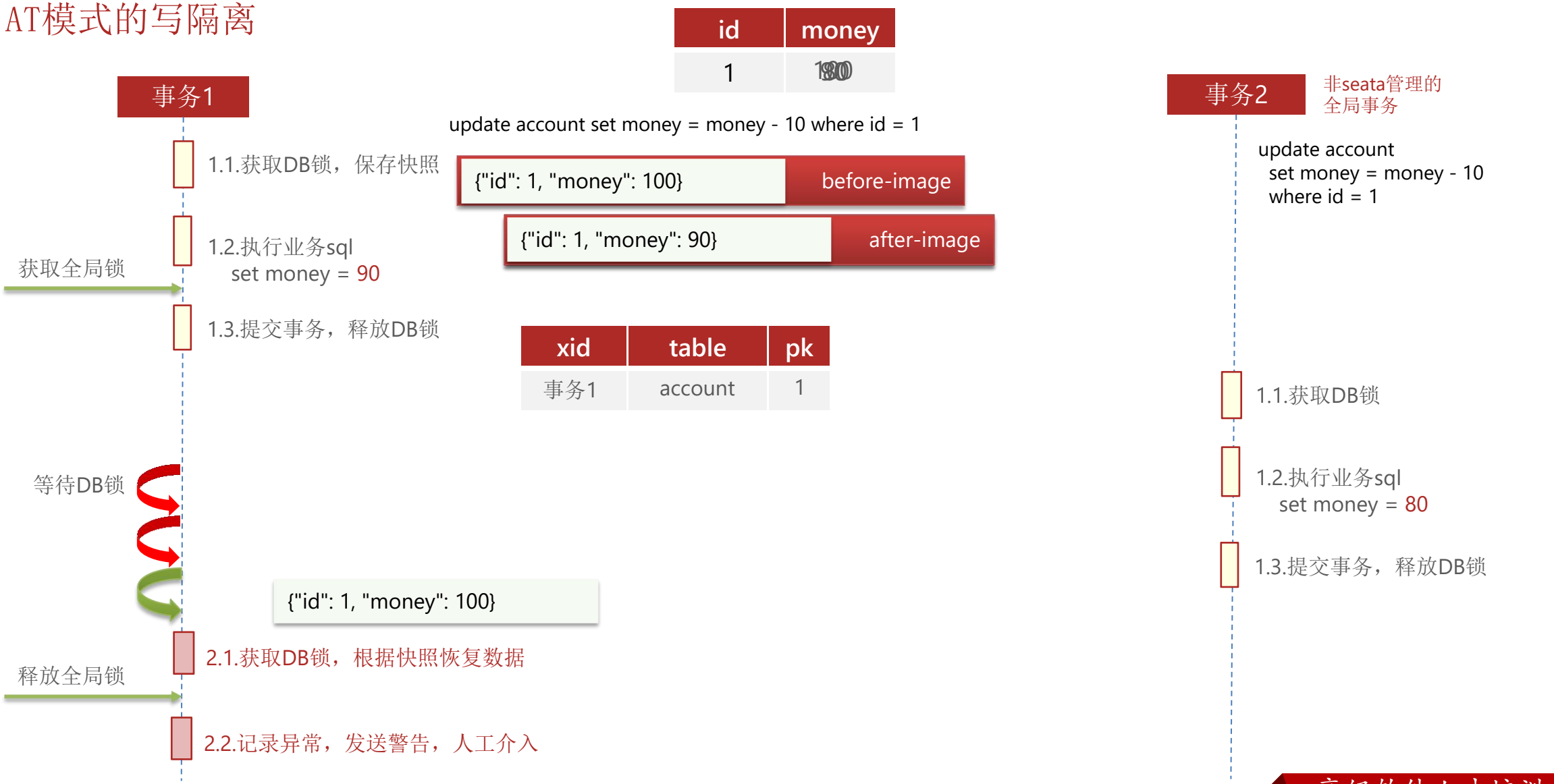
AT模式的脏写问题


id
money
1
100
事务1
update account set money = money - 10 where id = 1
1.1.获取DB锁，保存快照
{"id": 1, "money": 100}
1.2.执行业务sql
set money = 90
1.3.提交事务，释放DB锁
事务2
{"id": 1, "money": 90}
1.1.获取DB锁，保存快照
1.2.执行业务sql
set money = 80
1.3.提交事务，释放DB锁
{"id": 1, "money": 100}
2.1.获取DB锁，根据快照恢复数据

AT模式的写隔离

id	money
1	1900

AT模式的写隔离


id
money
1
100
事务1
update account set money = money - 10 where id = 1
1.1.获取DB锁，保存快照
{"id": 1, "money": 100}
before-image
1.2.执行业务sql
set money = 90
{"id": 1, "money": 90}
after-image
获取全局锁
1.3.提交事务，释放DB锁
xid
table
pk
事务1
account
1
事务2
非seata管理的
全局事务
update account
set money = money - 10
where id = 1
1.1.获取DB锁
1.2.执行业务sql
set money = 80
1.3.提交事务，释放DB锁
等待DB锁
{"id": 1, "money": 100}
2.1.获取DB锁，根据快照恢复数据
释放全局锁
2.2.记录异常，发送警告，人工介入

目录

Contents

- CAP和BASE
- AT模式的脏写问题
- TCC模式
- 最大努力通知

TCC模式

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

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

TCC模式原理

举例，一个扣减用户余额的业务。假设账户A原来余额是100，需要余额扣减30元。

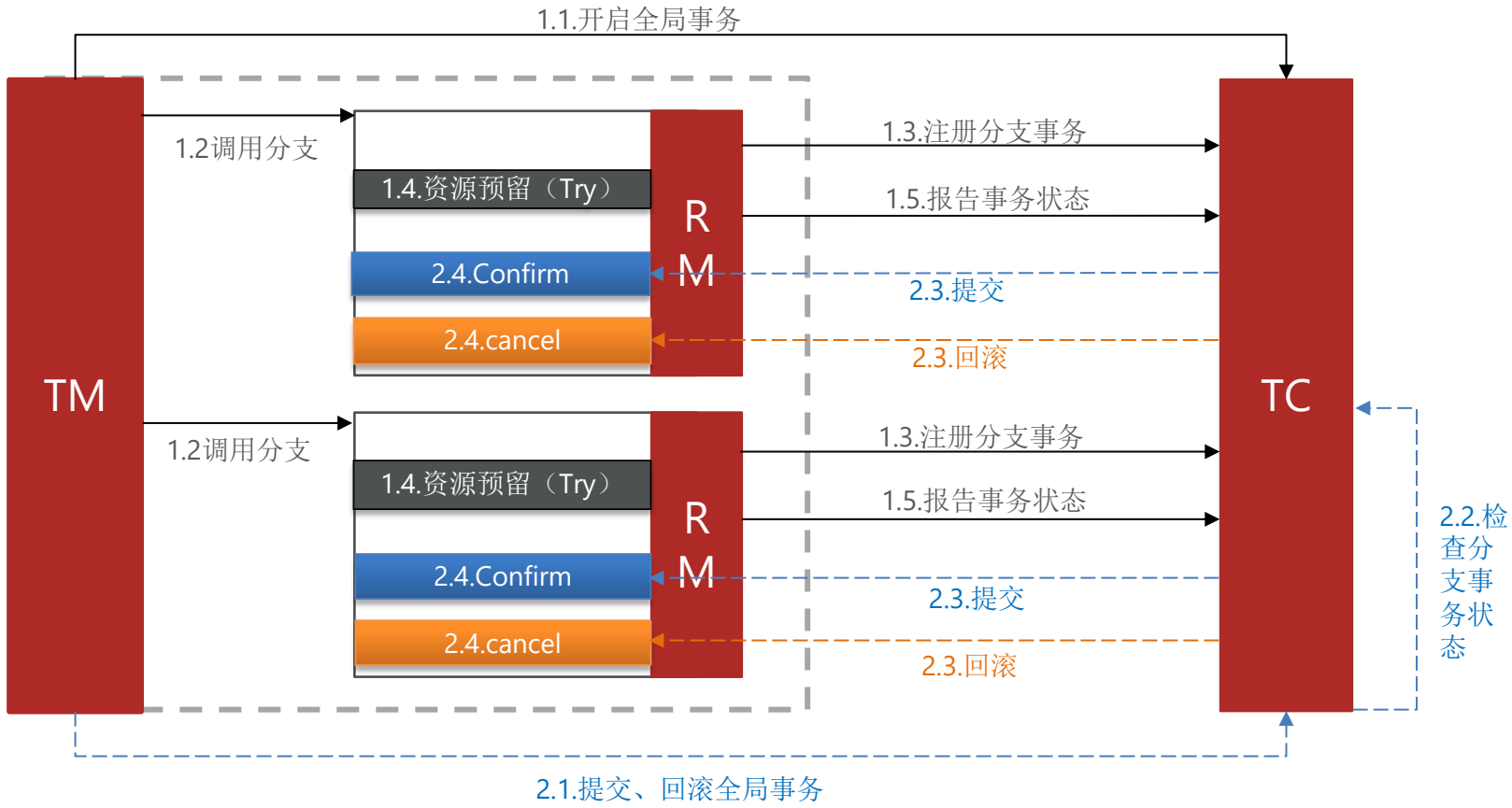
- 阶段一（ Try ）：检查余额是否充足，如果充足则冻结金额增加30元，可用余额扣除30

- 阶段二：假如要提交（Confirm），则冻结金额扣减30

- 阶段二：如果要回滚（Cancel），则冻结金额扣减30，可用余额增加30

TCC模式原理

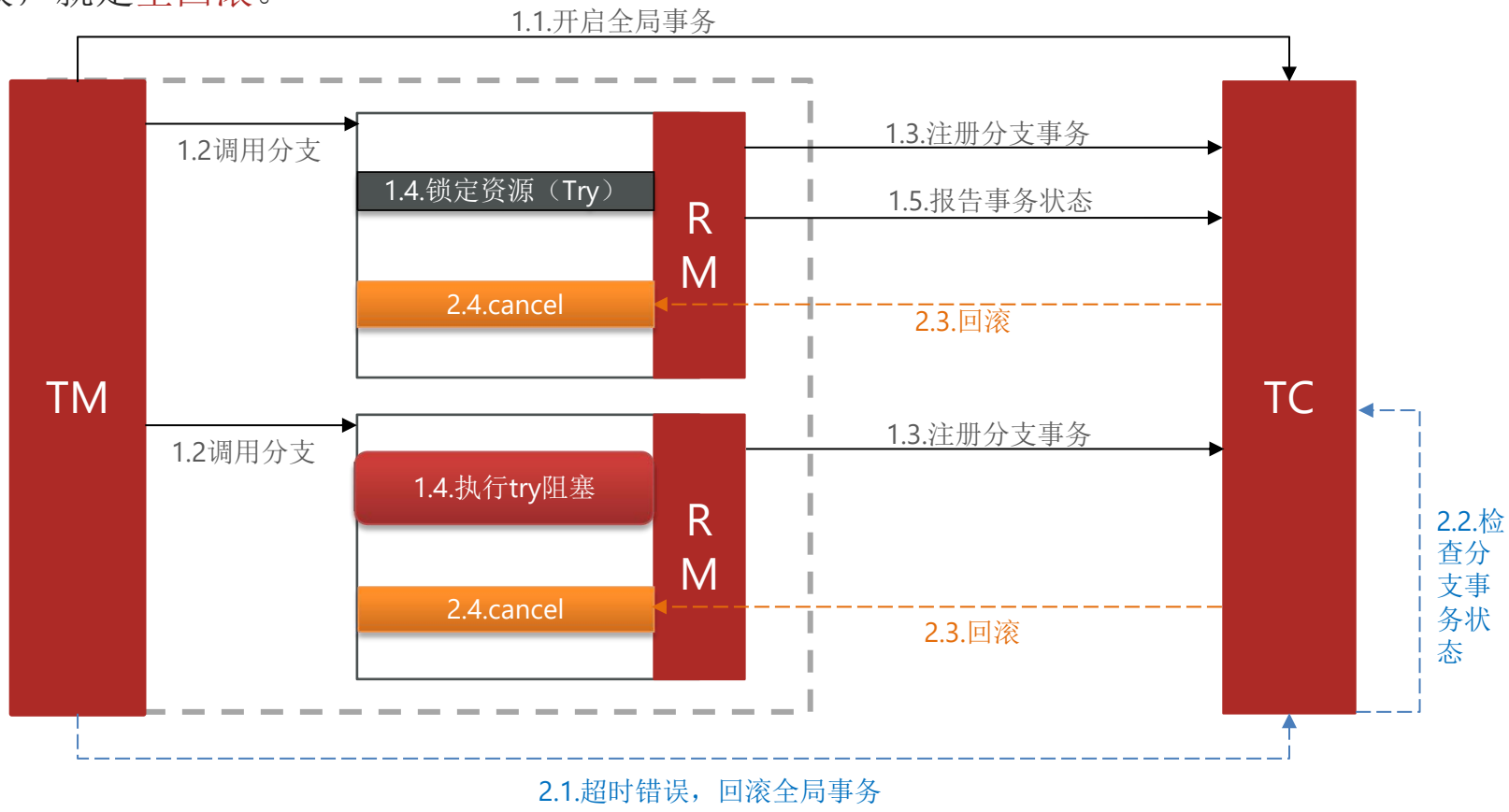
TCC的工作模型图：

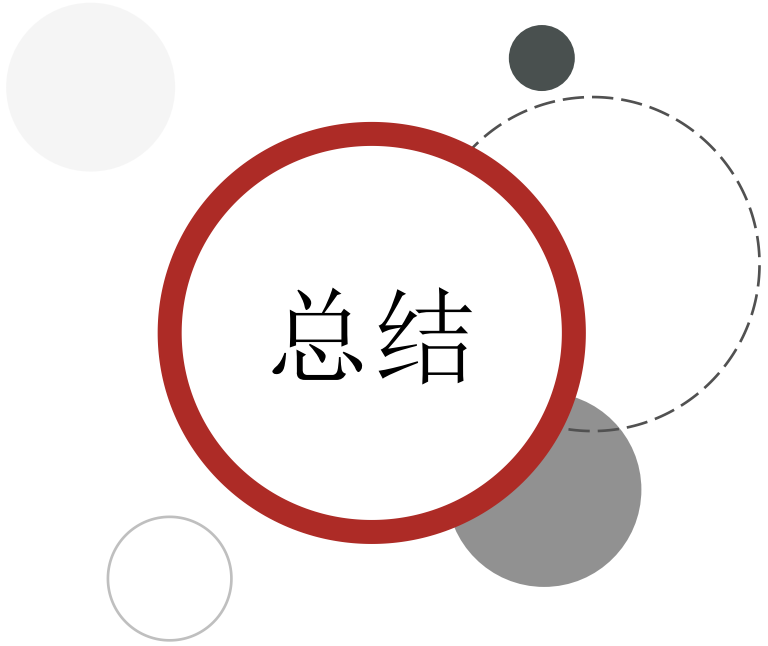


TCC的空回滚和业务悬挂

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





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
- 要编写逻辑解决空回滚和业务悬挂的问题

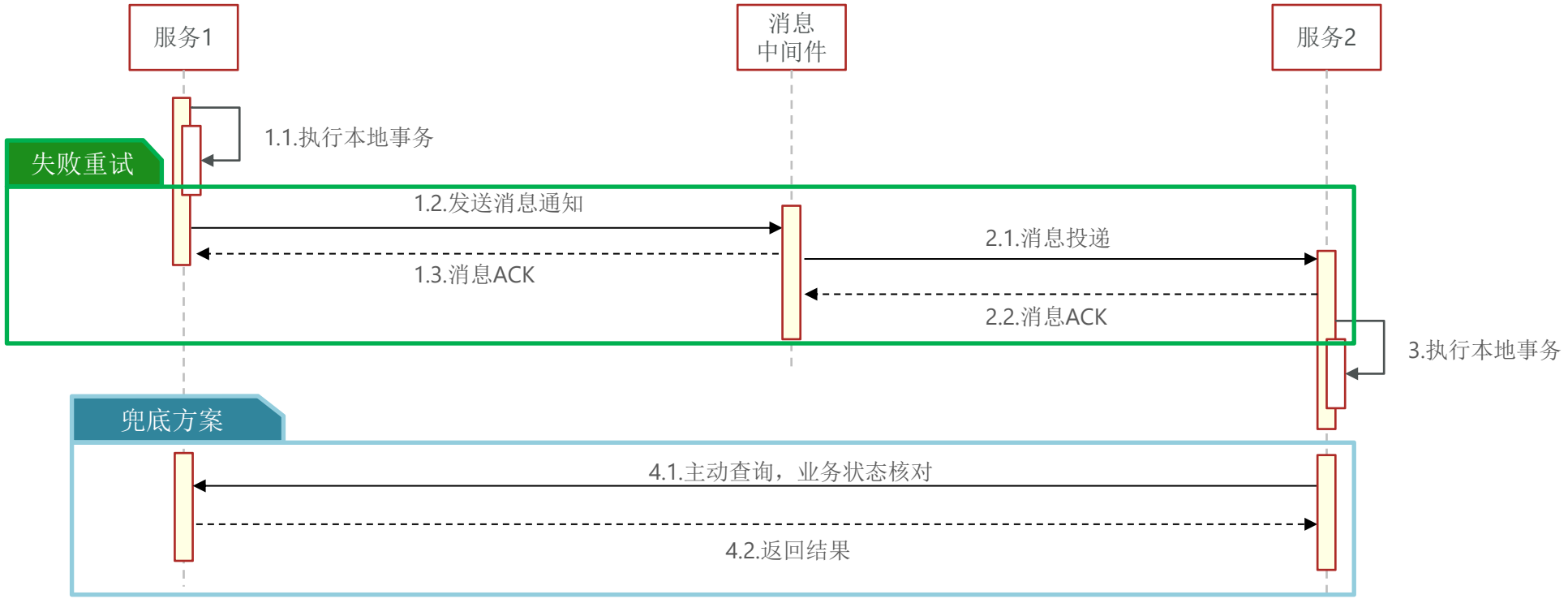
目录

Contents

- CAP和BASE
- AT模式的脏写问题
- TCC模式
- 最大努力通知

最大努力通知

最大努力通知是一种最终一致性的分布式事务解决方案。顾明思议，就是通过消息通知的方式来通知事务参与者完成业务执行，如果执行失败会多次通知。无需任何分布式事务组件介入。



注册中心

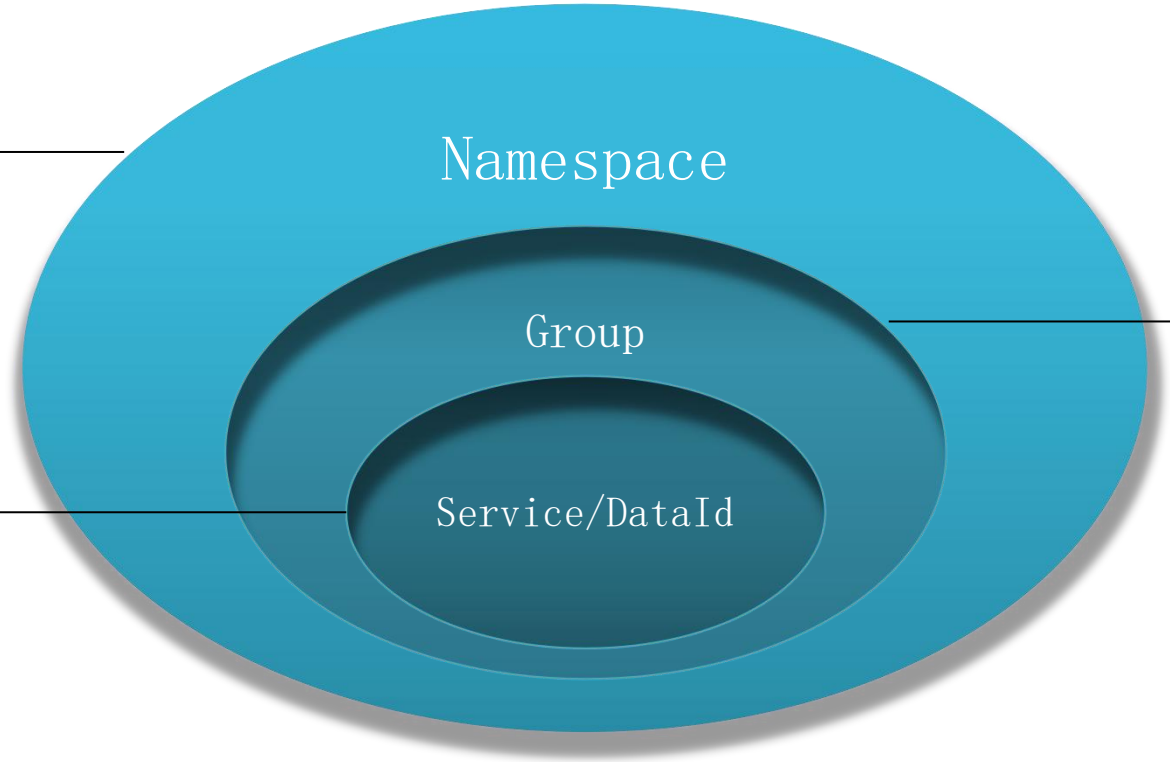
目录

Contents

- 环境隔离
- 分级模型
- Eureka与Nacos

环境隔离

企业实际开发中，往往会搭建多个运行环境，例如：开发环境、测试环境、发布环境。不同环境之间需要隔离。或者不同项目使用了一套Nacos，不同项目之间要做环境隔离。



命名空间

用于环境隔离。默认是空，表示公共环境public

分组

用于对服务或配置分组。默认是DEFAULT_GROUP

服务名/配置名

用于服务注册发现时，就是服务名。
用于配置管理时，就是配置文件名。

环境隔离

在Nacos控制台可以创建namespace，用来隔离不同环境

环境隔离

在Nacos控制台可以创建namespace，用来隔离不同环境

环境隔离

在Nacos控制台可以创建namespace，用来隔离不同环境

环境隔离

在微服务中，我们可以通过配置文件指定当前服务所属的namespace：

```yaml
spring:
  application:
    name: item-service # 服务名称
  profiles:
    active: dev
  cloud:
    nacos:
      server-addr: 192.168.150.101:8848 # nacos地址
      config:
        namespace: 8c468c63-b650-48da-a632-311c75e6d235 # 设置namespace，必须用id
        file-extension: yaml
        shared-configs:
          - dataId: shared-jdbc.yaml
          - dataId: shared-log.yaml
          - dataId: shared-swagger.yaml
          - dataId: shared-seata.yaml
      discovery: # 服务发现配置
        namespace: 8c468c63-b650-48da-a632-311c75e6d235 # 设置namespace，必须用id
```

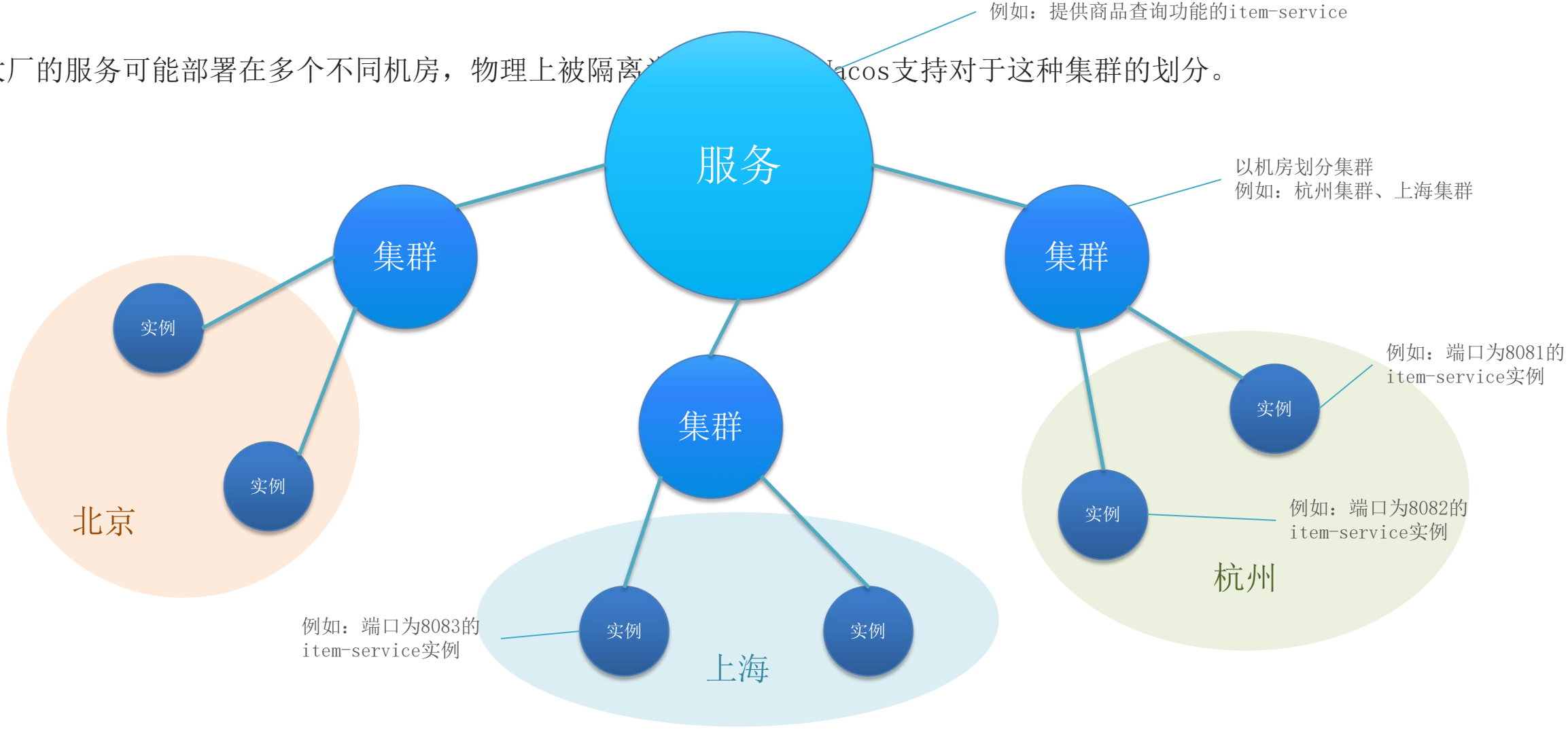
目录

Contents

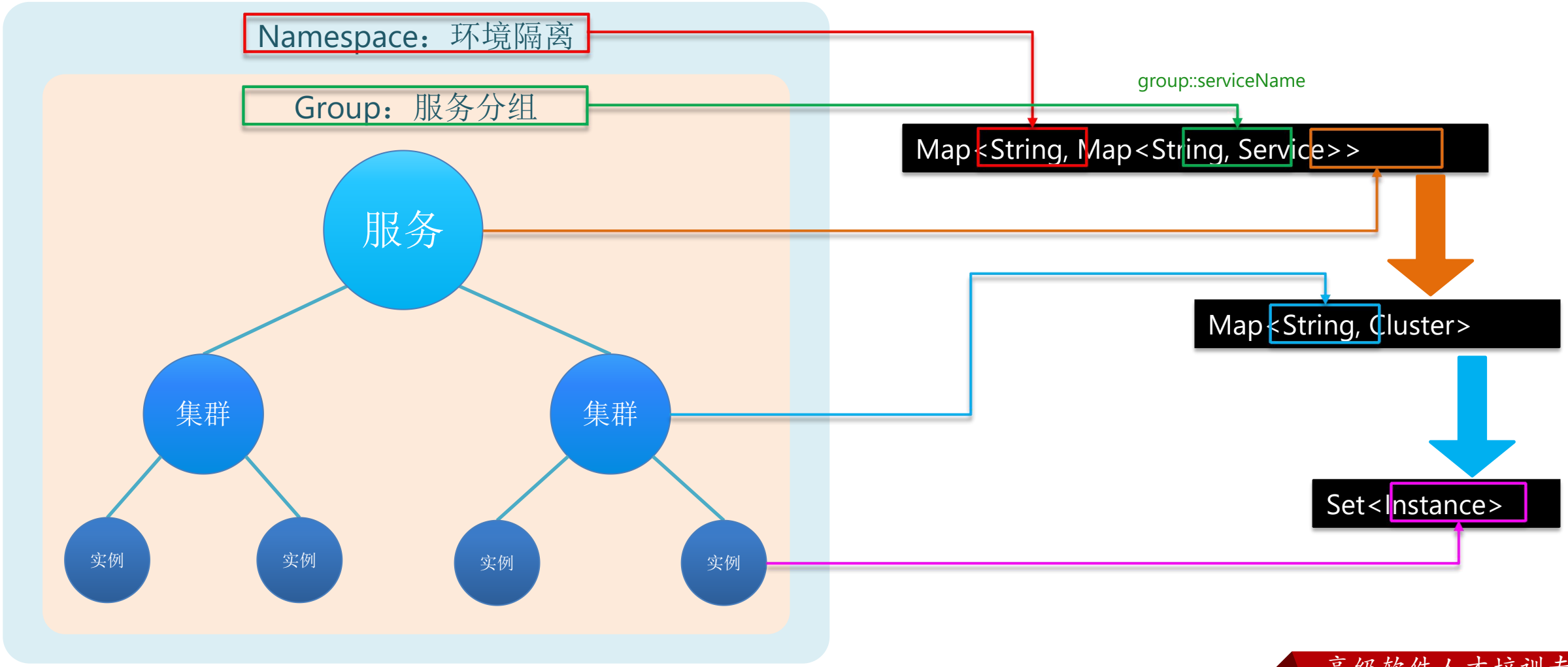
- 环境隔离
- 分级模型
- Eureka与Nacos

分级模型

大厂的服务可能部署在多个不同机房，物理上被隔离，Nacos支持对于这种集群的划分。



分级模型


Namespace：环境隔离
Group：服务分组
服务
集群
集群
实例
实例
实例
实例
group::serviceName
Map<String, Map<String, Service>>
Map<String, Cluster>
Set<Instance>

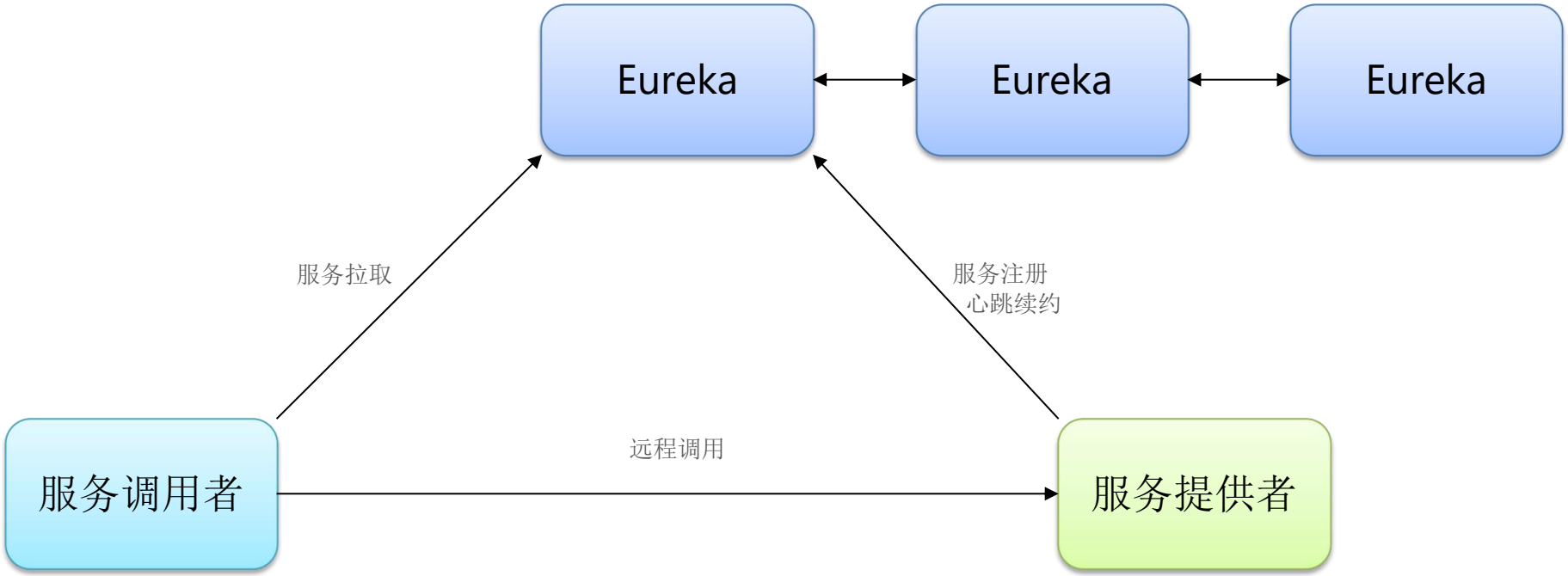
目录

Contents

- 环境隔离
- 分级模型
- Eureka与Nacos

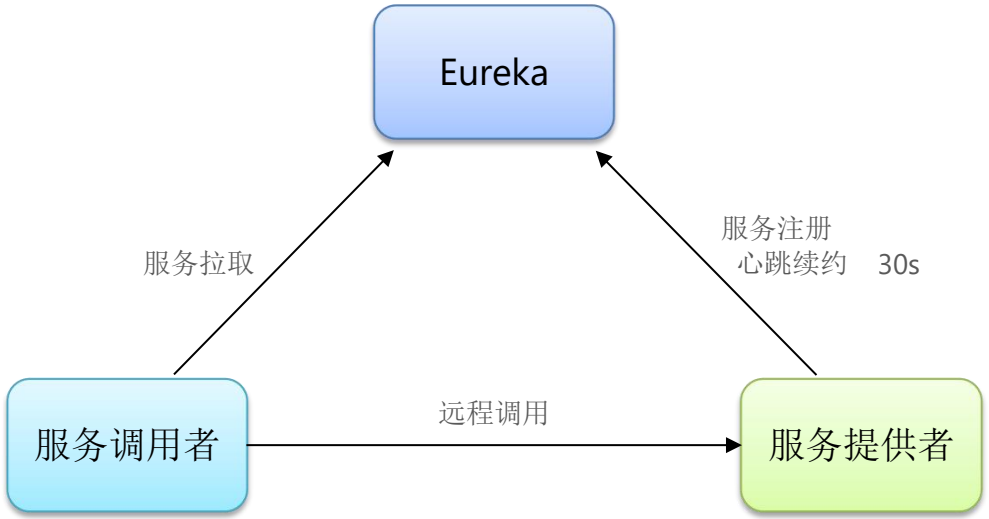
Eureka与Nacos

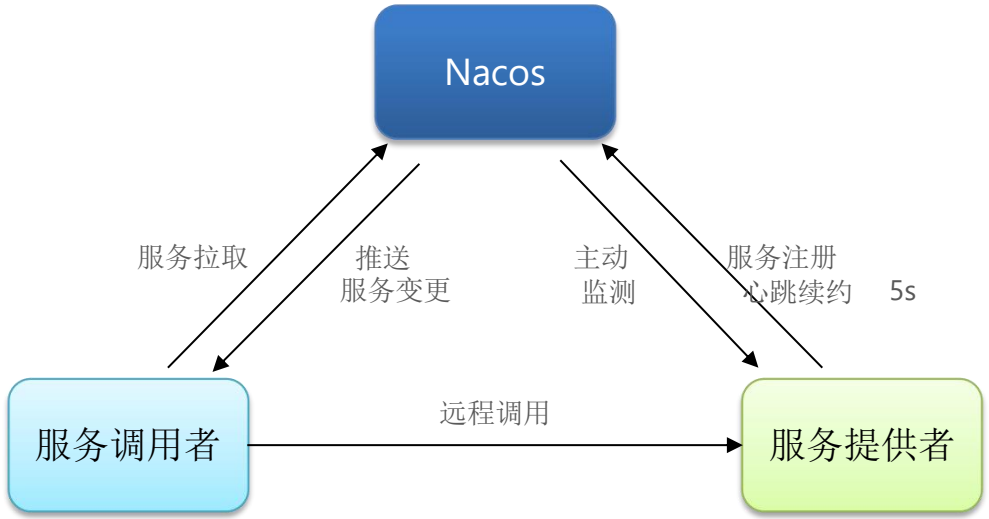
Eureka是Netflix公司开源的一个注册中心组件，目前被集成在SpringCloudNetflix这个模块下。它的工作原理与Nacos类似：



Eureka与Nacos

Eureka是Netflix公司开源的一个注册中心组件，目前被集成在SpringCloudNetflix这个模块下。它的工作原理与Nacos类似：





总结

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，但也支持CP；Eureka采用AP方式

远程调用

目录

Contents

- 负载均衡原理
- 切换负载均衡算法

负载均衡原理

自SpringCloud2020版本开始，SpringCloud弃用Ribbon，改用Spring自己开源的Spring Cloud LoadBalancer了，我们使用的OpenFeign、Gateway都已经与其整合。

OpenFeign在整合SpringCloudLoadBalancer时，与我们手动服务发现、复杂均衡的流程类似。

① 获取serviceId，也就是服务名称

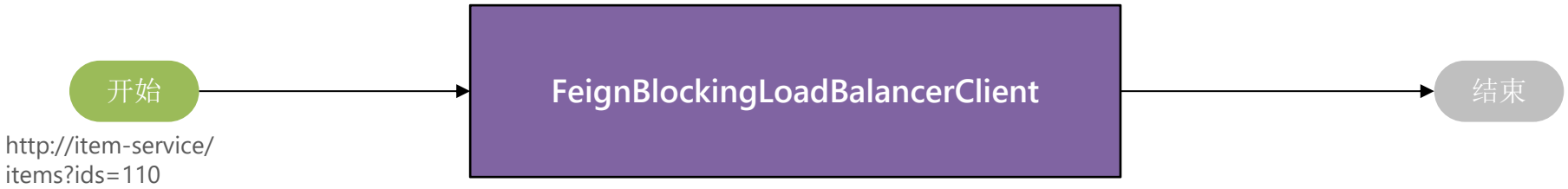
② 根据serviceId拉取服务列表

③ 利用负载均衡算法选择一个服务

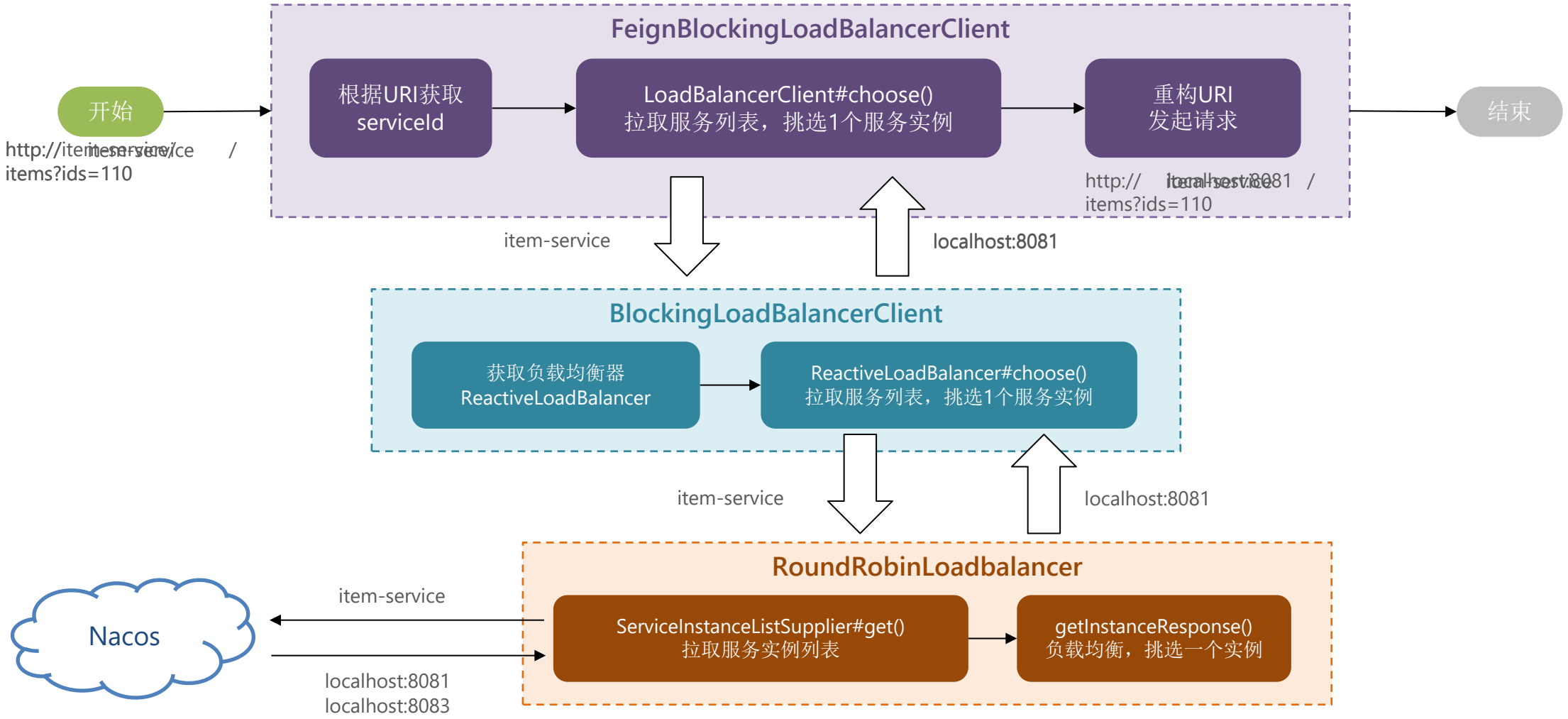
④ 重构请求的URL路径，发起远程调用

```java
// 服务拉取的客户端
private final DiscoveryClient discoveryClient;

private void handleCartItems(List<CartVO> vos) {
    // ... coding

        // 2.1.根据服务名称获取服务的实例列表
        List<ServiceInstance> instances = discoveryClient.getInstances("item-service");
    if (CollUtil.isEmpty(instances)) {
        return;
    }
    // 2.2.手写负载均衡，从实例列表中挑选一个实例
        ServiceInstance instance = instances.get(RandomUtil.randomInt(instances.size()));
    // 2.3.利用RestTemplate发起http请求，得到http的响应
        ResponseEntity<List<ItemDTO>> response = restTemplate.exchange(/* ... */);

    // ... coding
}
```

负载均衡原理


开始
http://item-service/items?ids=110
FeignBlockingLoadBalancerClient
结束

负载均衡原理


FeignBlockingLoadBalancerClient
开始
http://item-service/items?ids=110
根据URI获取serviceId
LoadBalancerClient#choose()
拉取服务列表，挑选1个服务实例
重构URI
发起请求
结束
http://localhost:8081/items?ids=110
item-service
localhost:8081
BlockingLoadBalancerClient
获取负载均衡器
ReactiveLoadBalancer
ReactiveLoadBalancer#choose()
拉取服务列表，挑选1个服务实例
item-service
localhost:8081
RoundRobinLoadbalancer
Nacos
item-service
localhost:8081
localhost:8083
ServiceInstanceListSupplier#get()
拉取服务实例列表
getInstanceResponse()
负载均衡，挑选一个实例

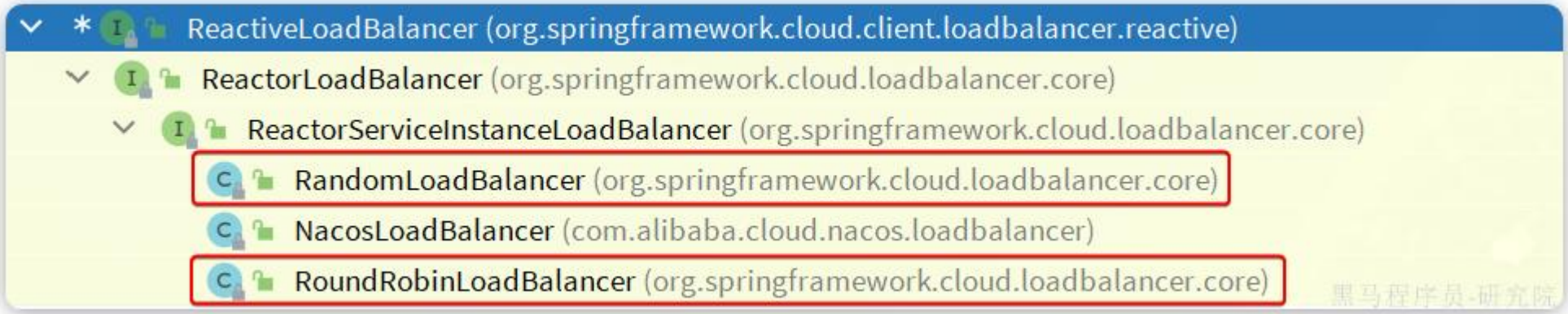
目录

Contents

- 负载均衡原理
- 切换负载均衡算法

切换负载均衡算法

分析源码的时候我们发现负载均衡的算法是有ReactiveLoadBalancer来定义的，我们发现它的实现类有三个：



其中RoundRobinLoadBalancer和RandomLoadBalancer是由Spring-Cloud-Loadbalancer模块提供的，而NacosLoadBalancer则是由Nacos-Discorvery模块提供的。

默认的策略是RoundRobinLoadBalancer，即轮询负载均衡。

切换负载均衡算法

要修改负载均衡策略则需要覆盖SpringCloudLoadBalancer中的自动装配配置：

```
public class LoadBalancerConfiguration {
    @Bean
    public ReactorLoadBalancer<ServiceInstance> reactorServiceInstanceLoadBalancer(
            Environment environment,
            NacosDiscoveryProperties properties,
            LoadBalancerClientFactory loadBalancerClientFactory) {
        String name = environment.getProperty(LoadBalancerClientFactory.PROPERTY_NAME);
        return new NacosLoadBalancer(
                loadBalancerClientFactory.getLazyProvider(name, ServiceInstanceListSupplier.class),
                name,
                properties);
    }
}
```

切换负载均衡算法

要修改负载均衡策略则需要覆盖SpringCloudLoadBalancer中的自动装配配置：

```
@LoadBalancerClients(defaultConfiguration = XxxConfiguration.class)
@EnableFeignClients(basePackages = "com.hmall.api.client")
@MapperScan("com.hmall.cart.mapper")
@SpringBootApplication
public class CartApplication {
    public static void main(String[] args) {
        SpringApplication.run(CartApplication.class, args);
    }
}
```

04 服务保护

目录

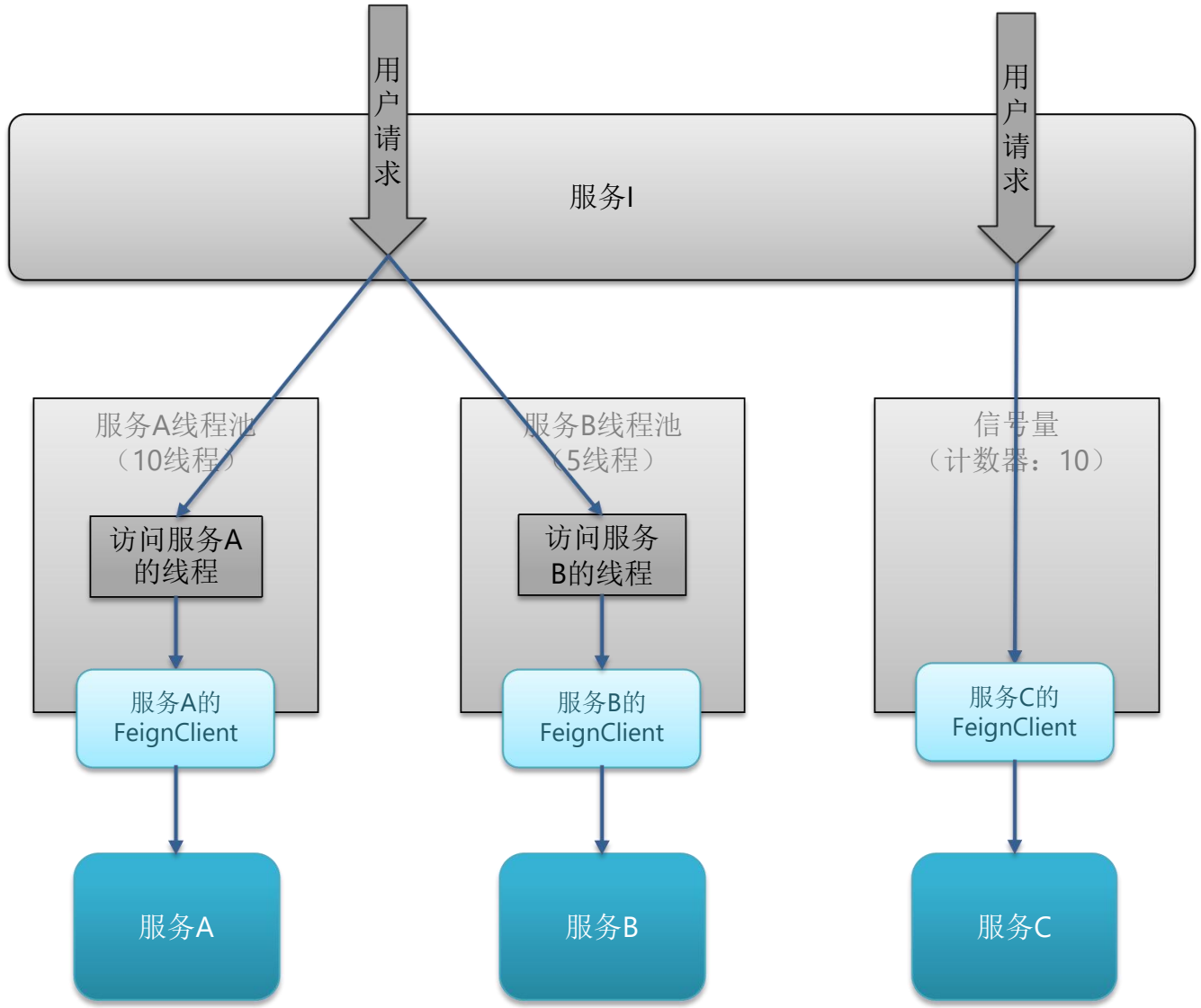
Contents

- 线程隔离
- 滑动窗口算法
- 漏桶算法
- 令牌桶算法

线程隔离

线程隔离有两种方式实现：

- 线程池隔离（Hystix默认采用）
- 信号量隔离（Sentinel默认采用）



Sentinel的线程隔离与Hystix的线程隔离有什么差别?

- **问题说明：**考察对线程隔离方案的掌握情况
- **难易程度：**一般
- **参考话术：**

答：线程隔离可以采用线程池隔离或者信号量隔离。

Hystix默认是基于线程池实现的线程隔离，每一个被隔离的业务都要创建一个独立的线程池，线程过多会带来额外的CPU开销，性能一般，但是隔离性更强。

Sentinel则是基于信号量隔离的原理，这种方式不用创建线程池，性能较好，但是隔离性一般。

目录

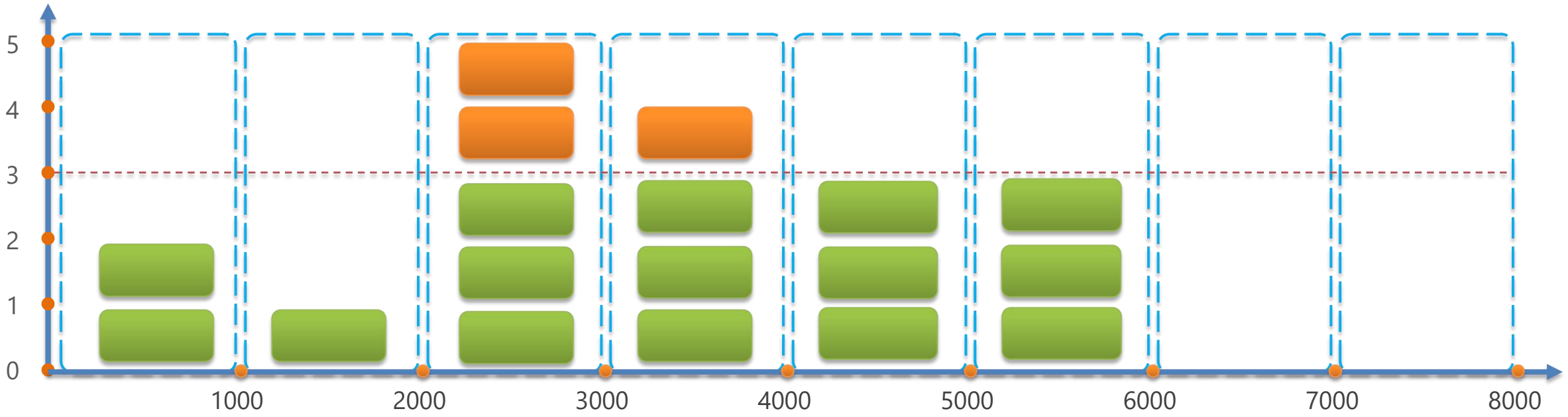
Contents

- 线程隔离
- 滑动窗口算法
- 漏桶算法
- 令牌桶算法

固定窗口计数器算法

固定窗口计数器算法概念如下：

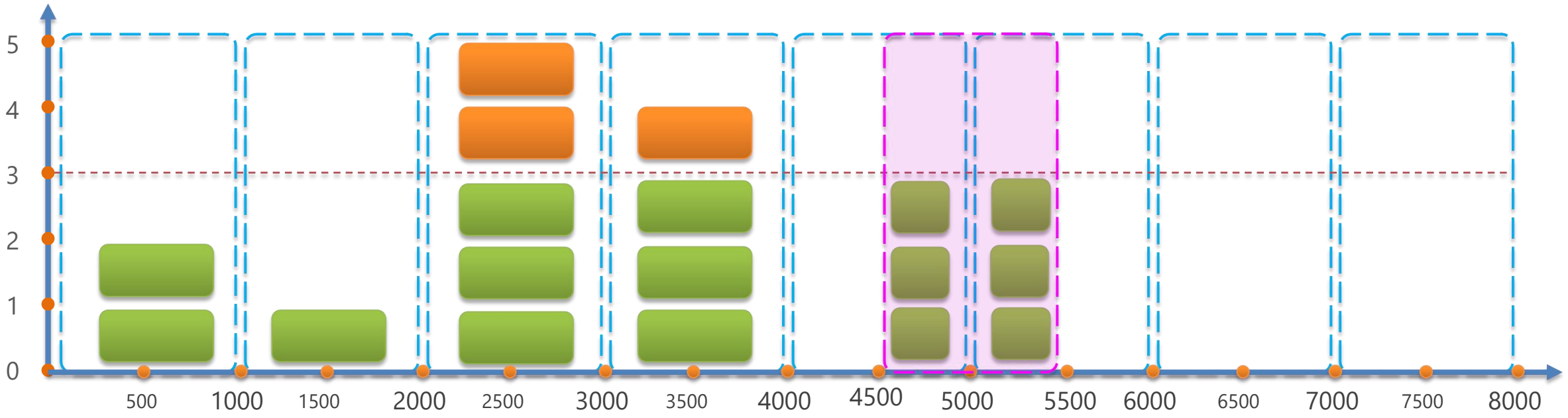
- 将时间划分为多个窗口，窗口时间跨度称为Interval，本例中为1000ms；
- 每个窗口分别计数统计，每有一次请求就将计数器加一，限流就是设置计数器阈值，本例为3
- 如果计数器超过了限流阈值，则超出阈值的请求都被丢弃。



固定窗口计数器算法

固定窗口计数器算法概念如下：

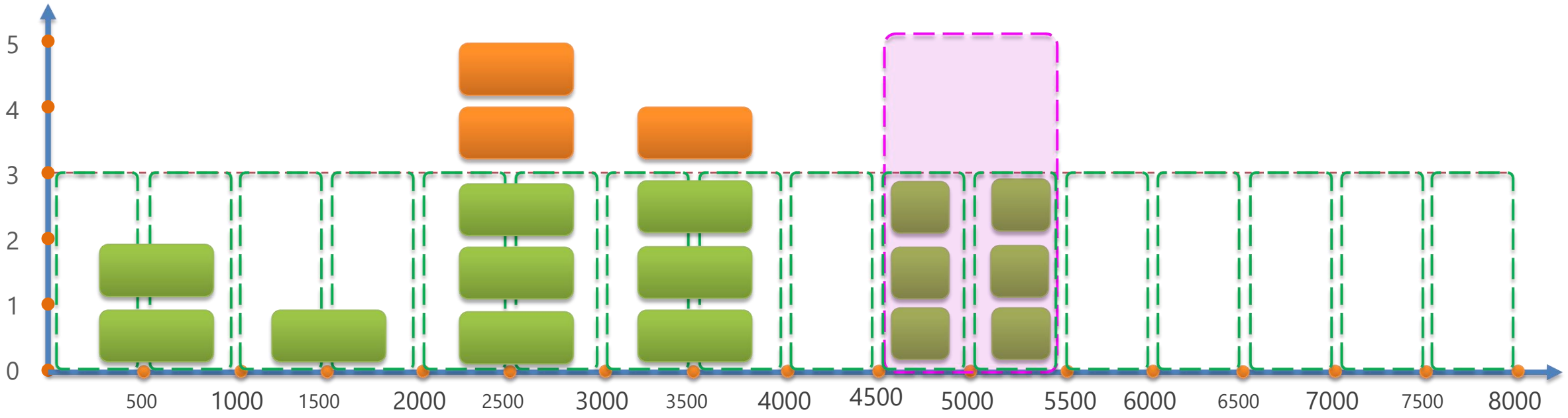
- 将时间划分为多个窗口，窗口时间跨度称为Interval，本例中为1000ms；
- 每个窗口分别计数统计，每有一次请求就将计数器加一，限流就是设置计数器阈值，本例为3
- 如果计数器超过了限流阈值，则超出阈值的请求都被丢弃。



固定窗口计数器算法

固定窗口计数器算法概念如下：

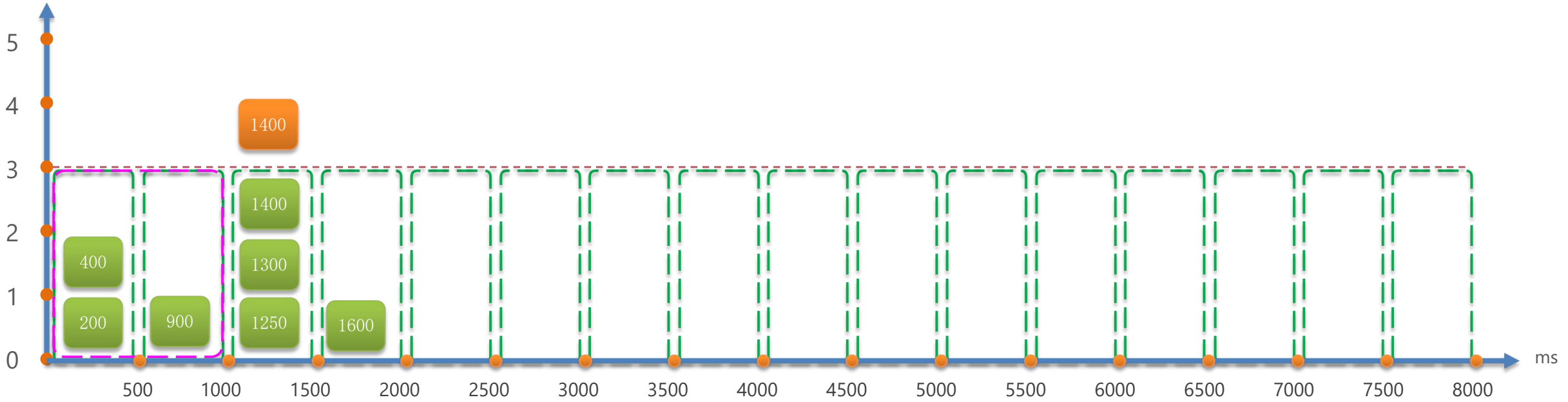
- 将时间划分为多个窗口，窗口时间跨度称为Interval，本例中为1000ms；
- 每个窗口分别计数统计，每有一次请求就将计数器加一，限流就是设置计数器阈值，本例为3
- 如果计数器超过了限流阈值，则超出阈值的请求都被丢弃。



滑动窗口计数器算法

滑动窗口计数器算法会将一个窗口划分为n个更小的区间，例如

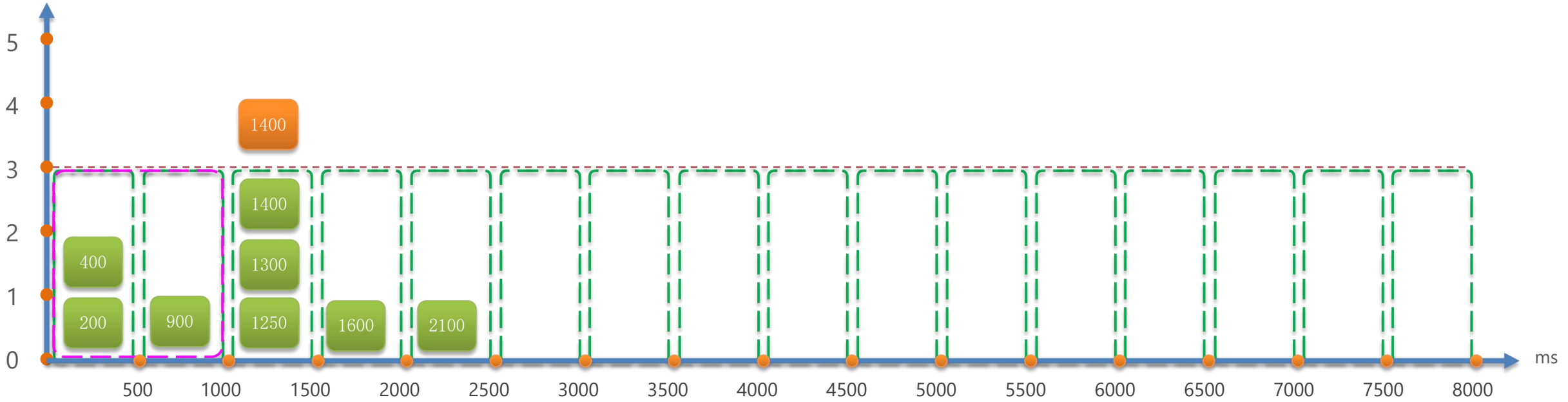
- 窗口时间跨度Interval为1秒；区间数量 n = 2 ，则每个小区间时间跨度为500ms，每个区间都有计数器
- 限流阈值依然为3，时间窗口（1秒）内请求超过阈值时，超出的请求被限流
- 窗口会根据当前请求所在时间（currentTime）移动，窗口范围是从（currentTime-Interval）之后的第一个时区开始，到currentTime所在时区结束。



滑动窗口计数器算法

滑动窗口计数器算法会将一个窗口划分为n个更小的区间，例如

- 窗口时间跨度Interval为1秒；区间数量 n = 2 ，则每个小区间时间跨度为500ms，每个区间都有计数器
- 限流阈值依然为3，时间窗口（1秒）内请求超过阈值时，超出的请求被限流
- 窗口会根据当前请求所在时间（currentTime）移动，窗口范围是从（currentTime-Interval）之后的第一个时区开始，到currentTime所在时区结束。



目录

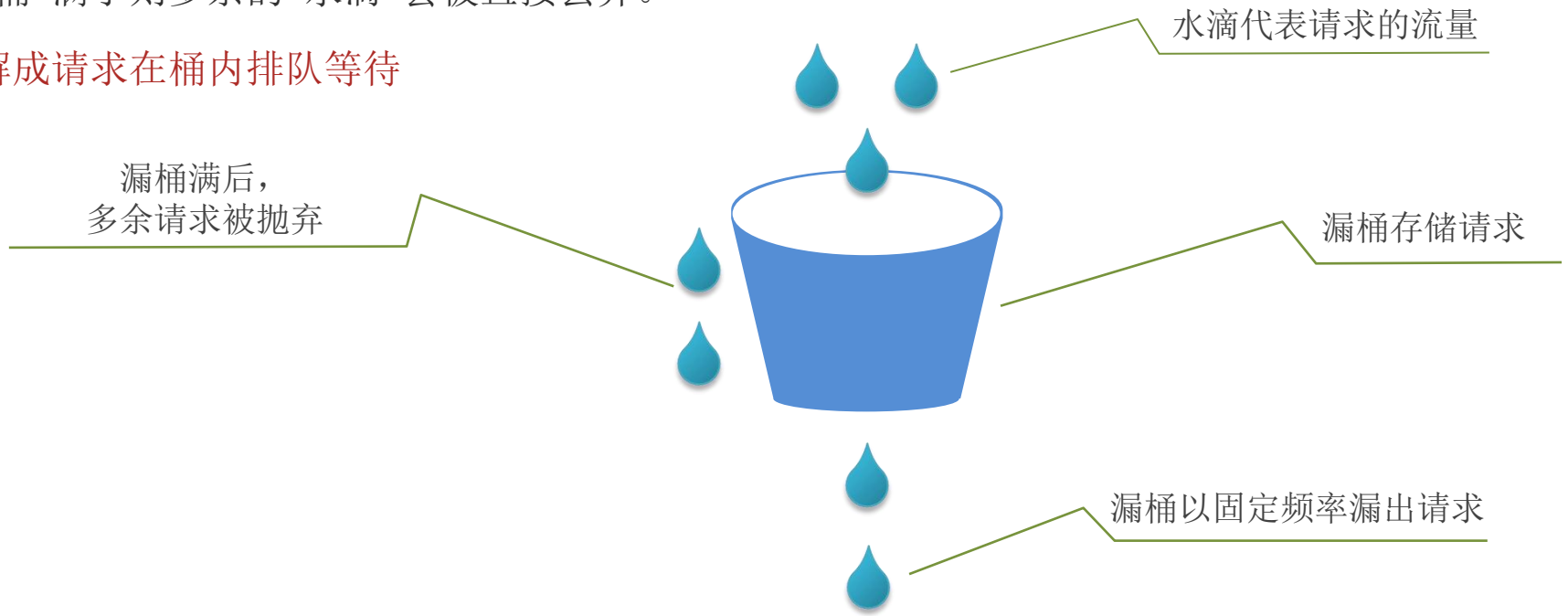
Contents

- 线程隔离
- 滑动窗口算法
- 漏桶算法
- 令牌桶算法

漏桶算法

漏桶算法说明：

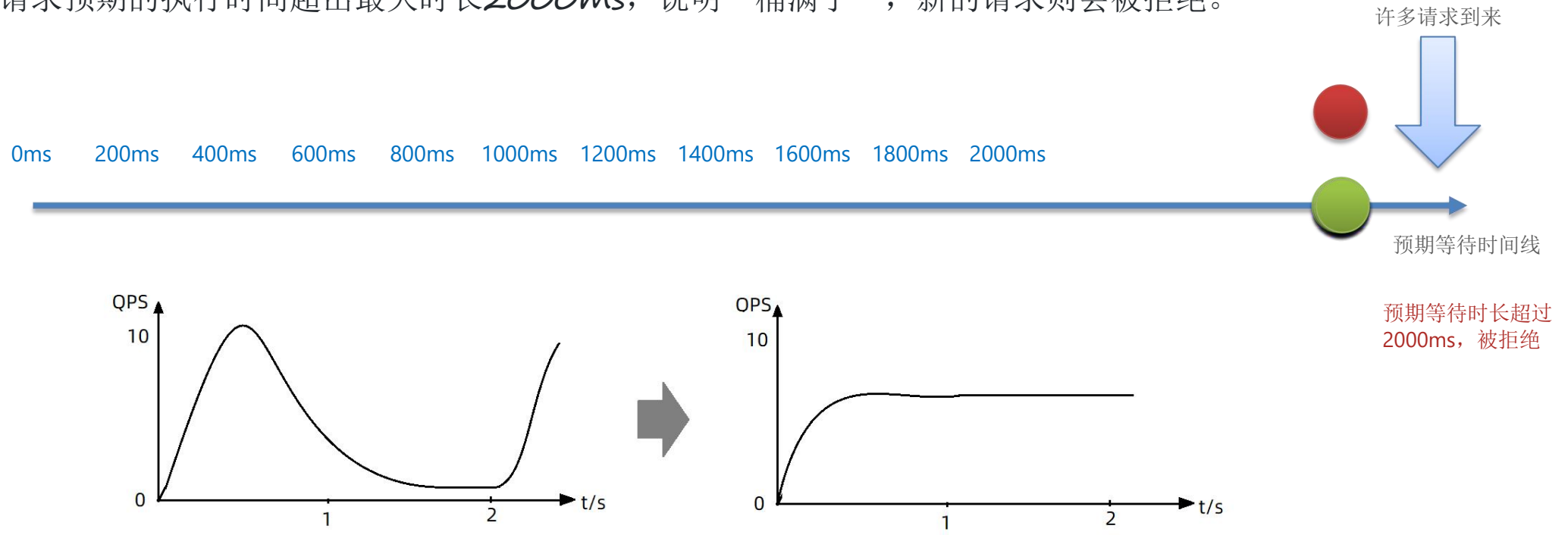
- 将每个请求视作"水滴"放入"漏桶"进行存储；
- "漏桶"以固定速率向外"漏"出请求来执行，如果"漏桶"空了则停止"漏水"；
- 如果"漏桶"满了则多余的"水滴"会被直接丢弃。
- 可以理解成请求在桶内排队等待



漏桶算法

Sentinel内部基于漏桶算法实现了排队等待效果，桶的**容量**取决限流的**QPS阈值**以及允许等待的**最大超时时间**：

例如：限流QPS=5，队列超时时间为2000ms。我们让所有请求进入一个队列中，如同进入漏桶中。由于漏桶是固定频率执行，因此QPS为5就是每200ms执行一个请求。那第N个请求的预期的执行时间 是第(N - 1) * 200ms。如果请求预期的执行时间超出最大时长2000ms，说明“桶满了”，新的请求则会被拒绝。



目录

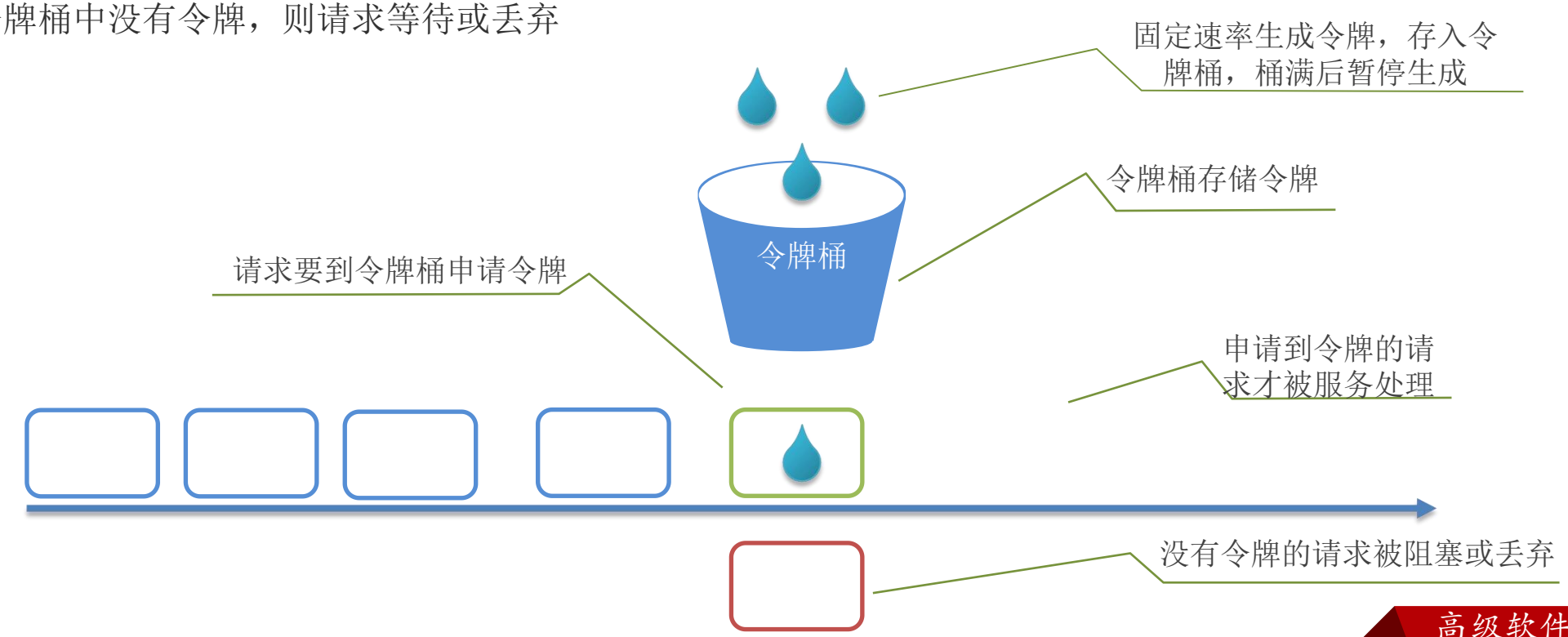
Contents

- 线程隔离
- 滑动窗口算法
- 漏桶算法
- 令牌桶算法

令牌桶算法

令牌桶算法说明：

- 以固定的速率生成令牌，存入令牌桶中，如果令牌桶满了以后，停止生成
- 请求进入后，必须先尝试从桶中获取令牌，获取到令牌后才可以被处理
- 如果令牌桶中没有令牌，则请求等待或丢弃



Sentinel的限流与Gateway的限流有什么差别？

- **问题说明：**考察对限流算法的掌握情况
- **难易程度：**难
- **参考话术：**

限流算法常见的有三种实现：滑动时间窗口、令牌桶算法、漏桶算法。Gateway则采用了基于Redis实现的令牌桶算法

而Sentinel内部却比较复杂：

✓ 默认限流模式是基于滑动时间窗口算法，另外Sentinel中断路器的计数也是基于滑动时间窗口算法

✓ 限流后可以快速失败和排队等待，其中排队等待基于漏桶算法

✓ 而热点参数限流则是基于令牌桶算法


黑马程序员
www.itheima.com
传智教育旗下高端IT教育品牌