分布式事务

seata

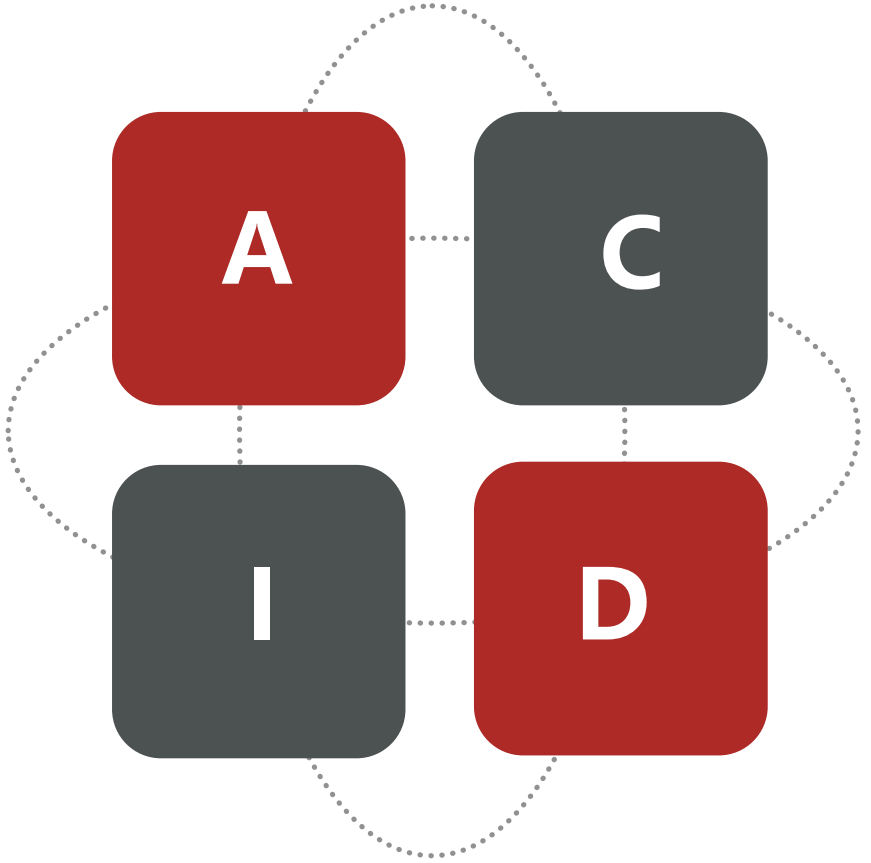
黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

事务的ACID原则

原子性

事务中的所有操作，要么全部成功，要么全部失败



一致性

要保证数据库内部完整性约束、声明性约束

隔离性

对同一资源操作的事务不能同时发生

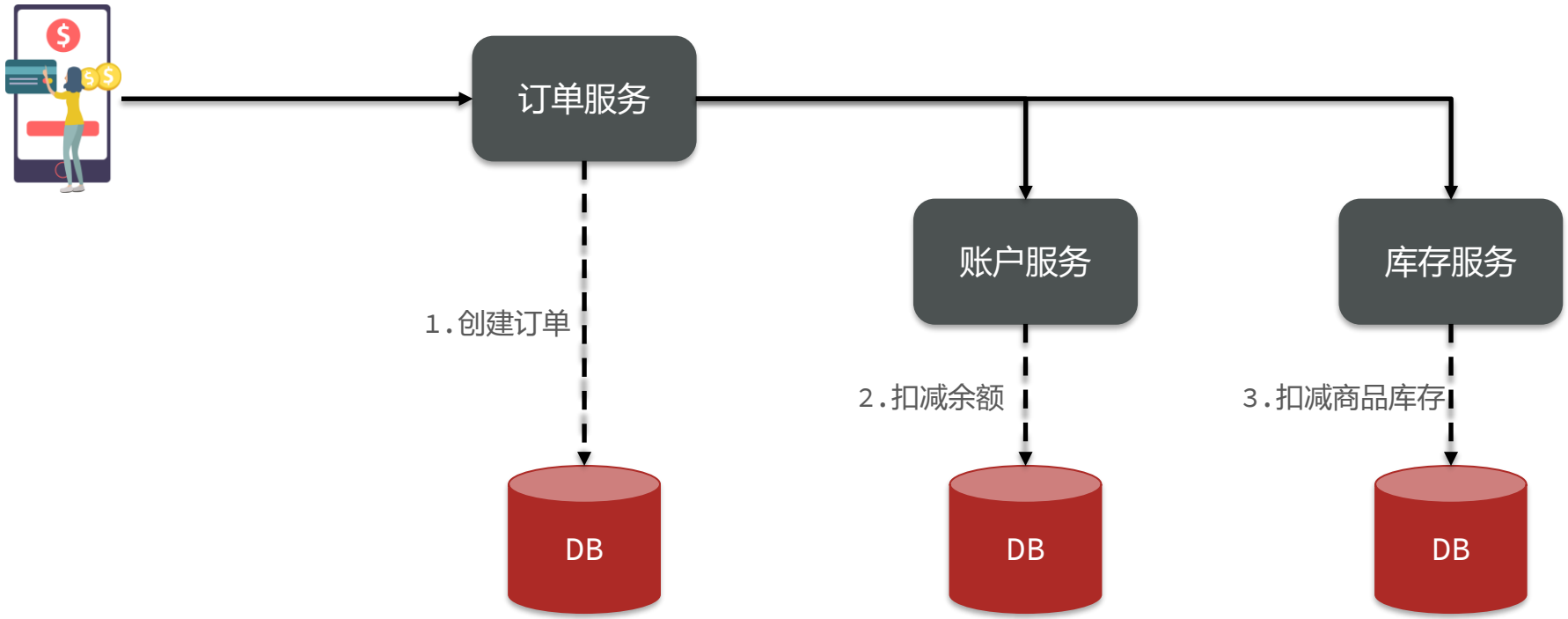
持久性

对数据库做的一切修改将永久保存，不管是否出现故障

分布式服务案例

微服务下单业务，在下单时会调用订单服务，创建订单并写入数据库。然后订单服务调用账户服务和库存服务：

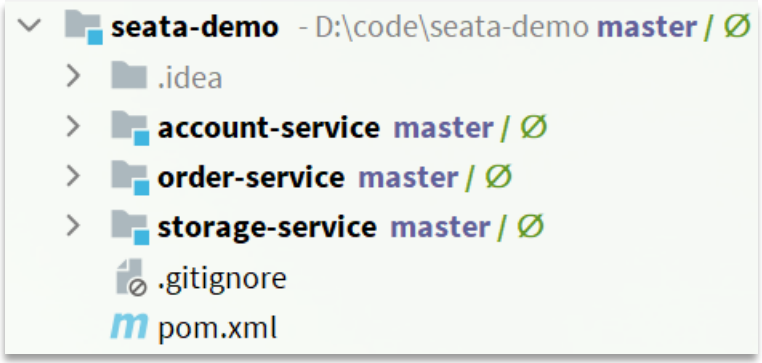
- 账户服务负责扣减用户余额
- 库存服务负责扣减商品库存



演示分布式事务问题

1. 创建数据库，名为seata_demo，然后导入课前资料提供的SQL文件：

seata-demo.sql

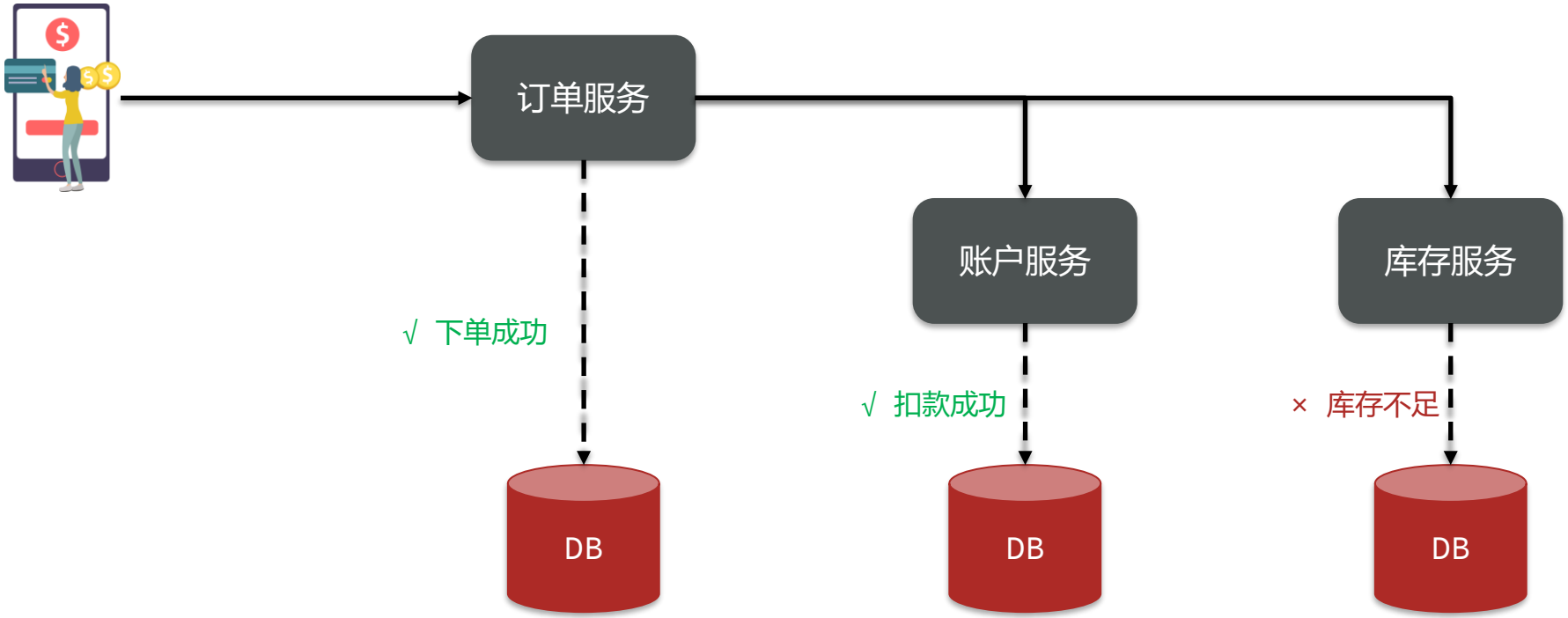


4. 测试下单功能，发出Post请求：

```
curl --location --
request POST 'http://localhost:8082/order?userId=user202103032042012&
commodityCode=100202003032041&count=2&money=200'
```

分布式服务的事务问题

在分布式系统下，一个业务跨越多个服务或数据源，每个服务都是一个分支事务，要保证所有分支事务最终状态一致，这样的事务就是分布式事务。



学习目标

分析产生原因

跨服务、跨数据源的业务

理解理论基础

思考解决分布式事务的基本思路

动手实践

利用seata解决分布式事务问题

弄清原理

了解Seata框架，学习seata原理

目录

Contents

- 理论基础
- 初识Seata
- 动手实践
- 高可用

理论基础

- CAP定理
- BASE理论

目录

Contents

- CAP定理
- BASE理论

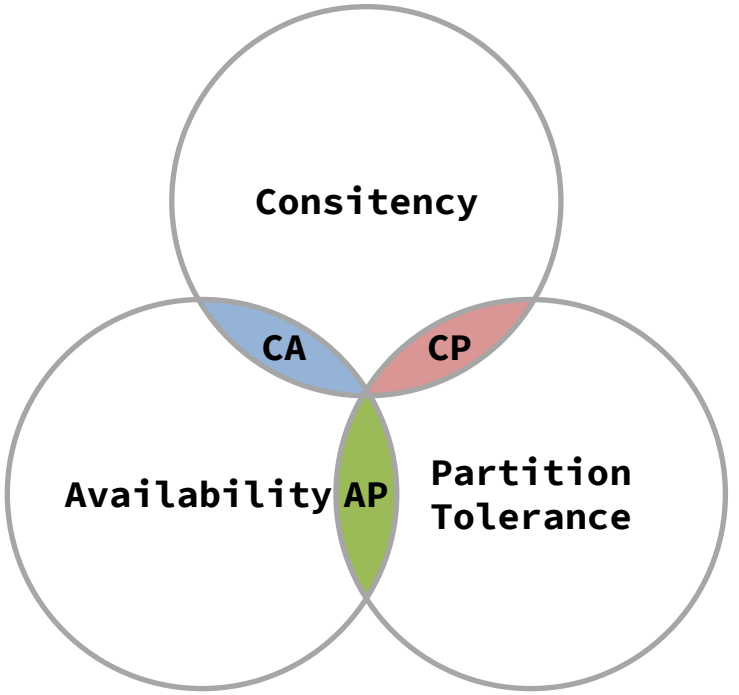
CAP定理

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）

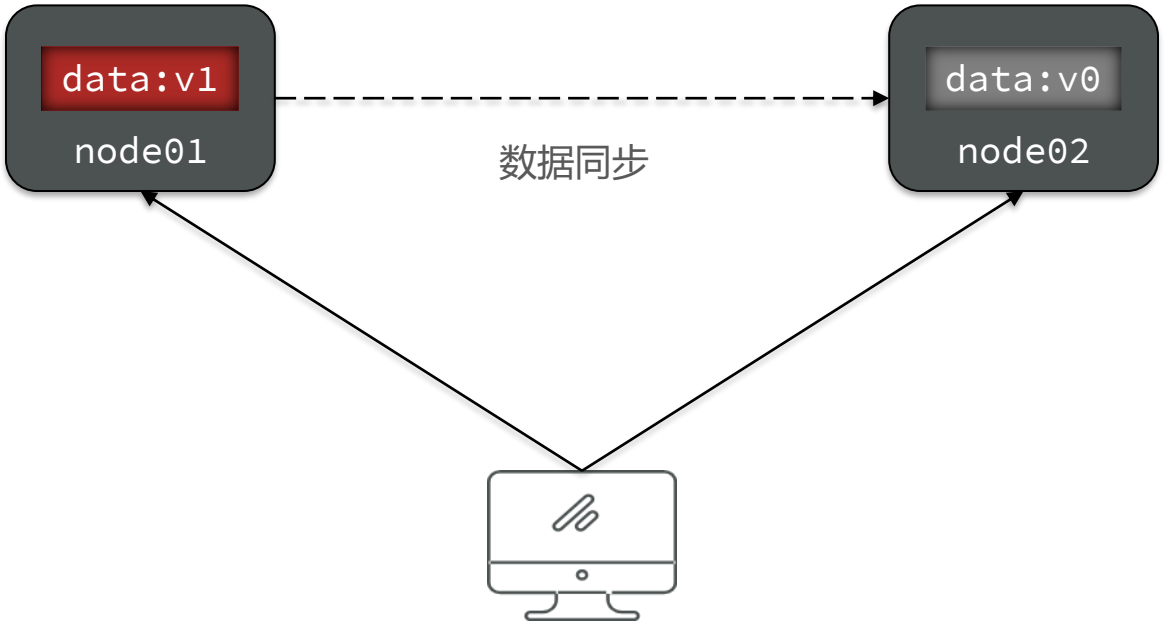
Eric Brewer 说，分布式系统无法同时满足这三个指标。

这个结论就叫做 CAP 定理。



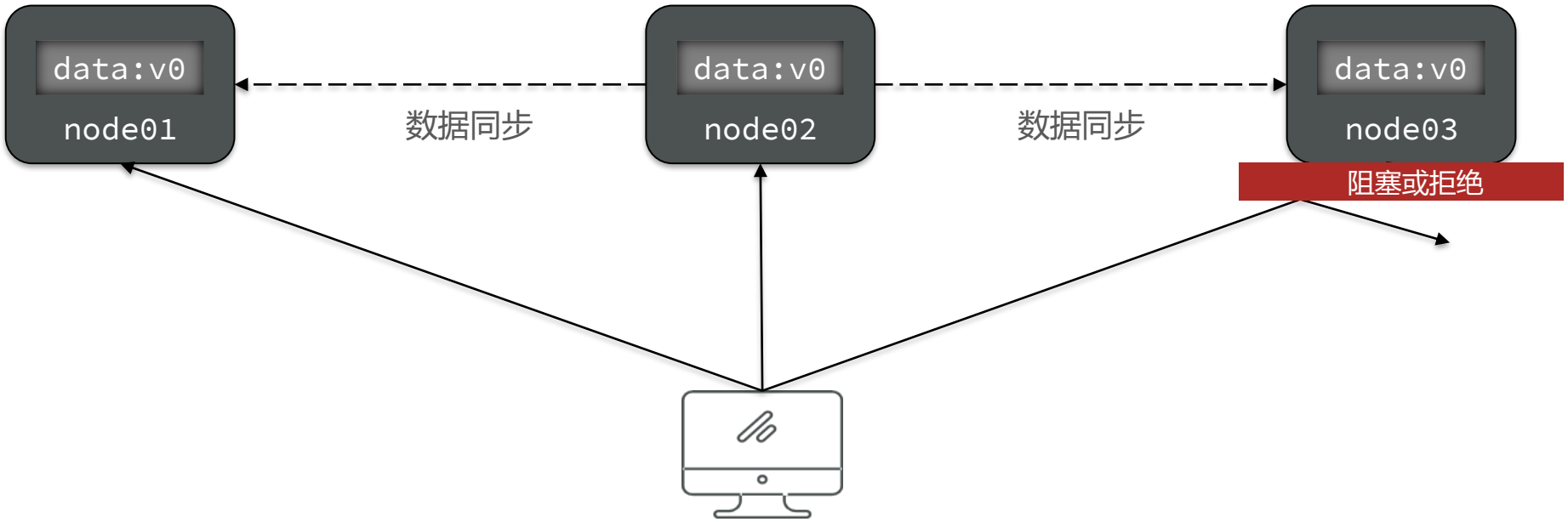
CAP定理- Consistency

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

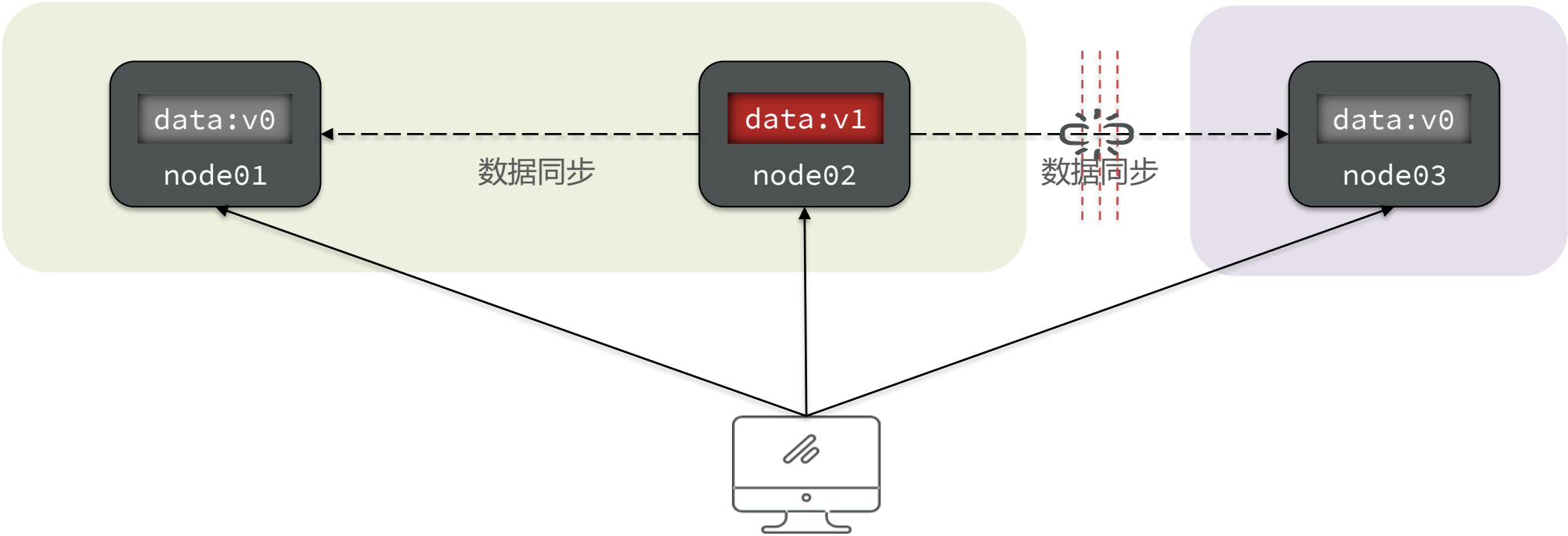
Availability （可用性）：用户访问集群中的任意健康节点，必须能得到响应，而不是超时或拒绝

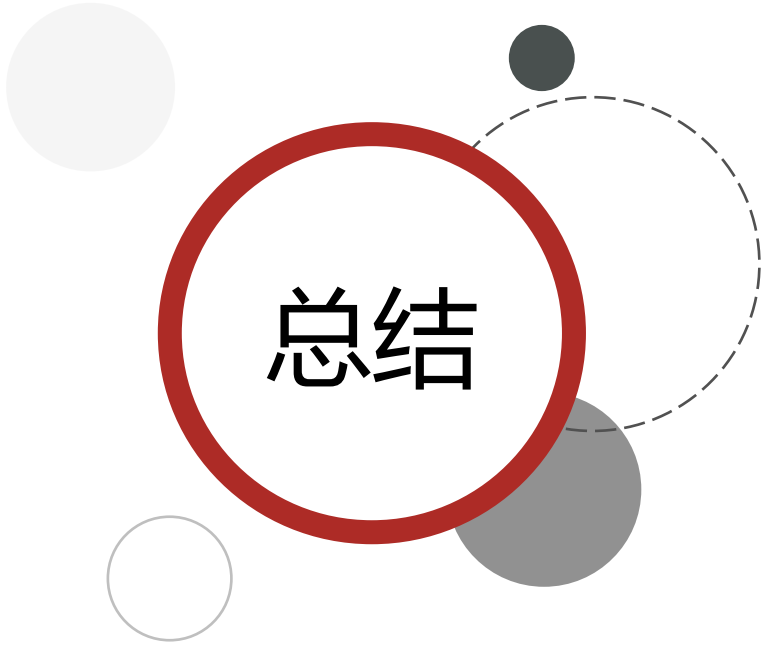


CAP定理-Partition tolerance

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





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

思考：elasticsearch集群是CP还是AP?

- ES集群出现分区时，故障节点会被剔除集群，数据分片会重新分配到其它节点，保证数据一致。因此是低可用性，高一致性，属于CP

目录

Contents

- CAP定理
- BASE理论

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。

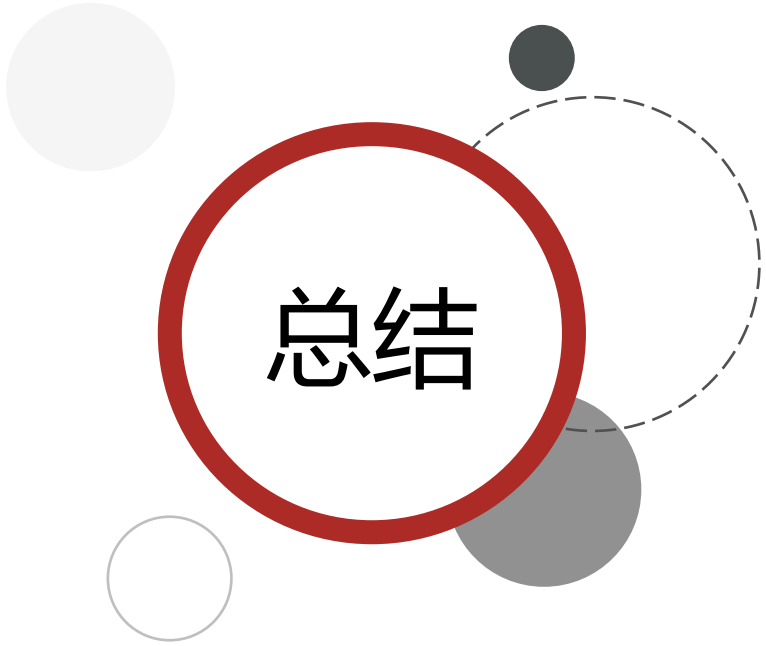
而分布式事务最大的问题是各个子事务的一致性问题，因此可以借鉴CAP定理和BASE理论：

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

分布式事务模型

解决分布式事务，各个子系统之间必须能感知到彼此的事务状态，才能保证状态一致，因此需要一个事务协调者来协调每一个事务的参与者（子系统事务）。

这里的子系统事务，称为分支事务；有关联的各个分支事务在一起称为全局事务



简述BASE理论三个思想：

- 基本可用
- 软状态
- 最终一致

解决分布式事务的思想和模型：

- 全局事务：整个分布式事务
- 分支事务：分布式事务中包含的每个子系统的事务
- 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
- 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

初识Seata

- Seata的架构
- 部署TC服务
- 微服务集成Seata

初识Seata

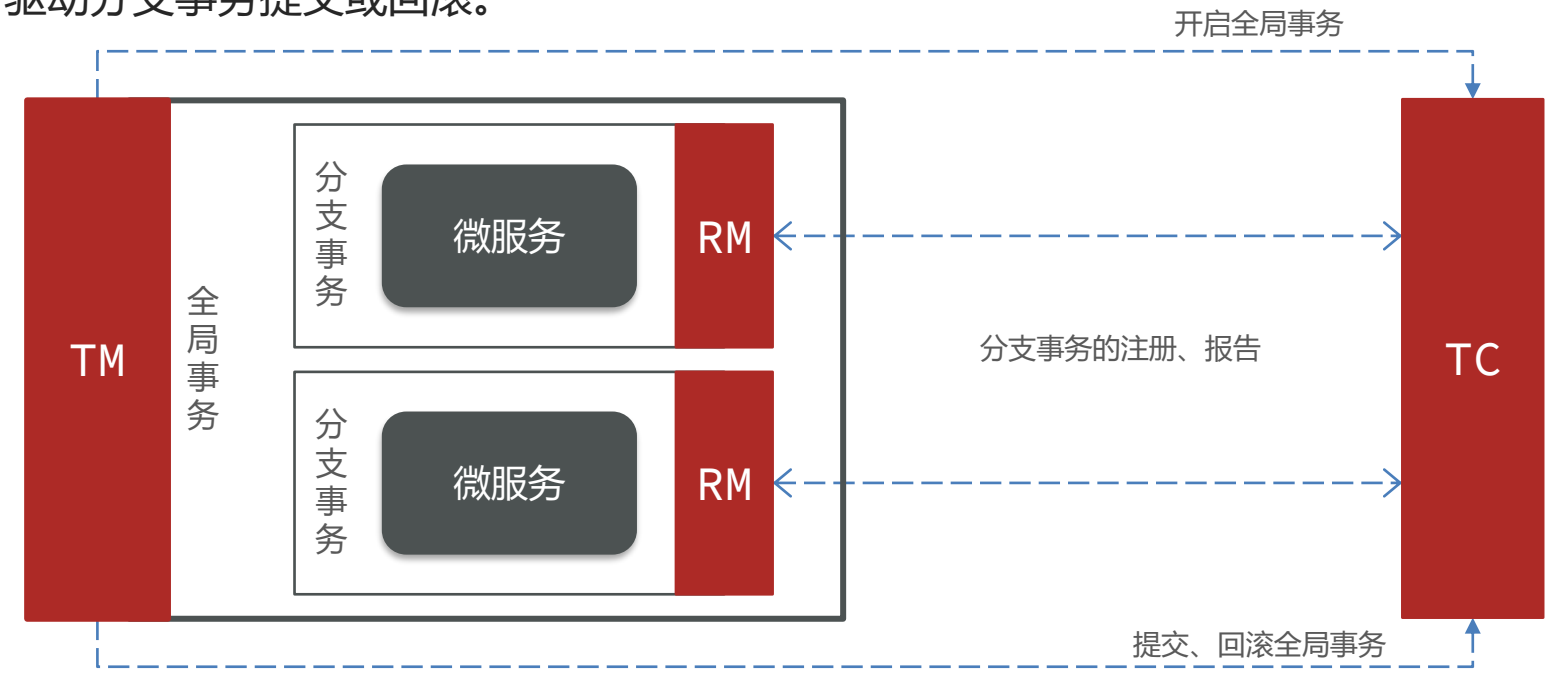
Seata是 2019 年 1 月份蚂蚁金服和阿里巴巴共同开源的分布式事务解决方案。致力于提供高性能和简单易用的分布式事务服务，为用户打造一站式的分布式解决方案。

官网地址：http://seata.io/，其中的文档、播客中提供了大量的使用说明、源码分析。

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。



初识Seata

Seata提供了四种不同的分布式事务解决方案：

- XA模式：强一致性分阶段事务模式，牺牲了一定的可用性，无业务侵入
- TCC模式：最终一致的分阶段事务模式，有业务侵入
- AT模式：最终一致的分阶段事务模式，无业务侵入，也是Seata的默认模式
- SAGA模式：长事务模式，有业务侵入

目录

Contents

- Seata的架构
- 部署TC服务
- 微服务集成Seata

部署TC服务

参考课前资料提供的文档《 seata的部署和集成.md 》：

目录

Contents

- Seata的架构
- 部署TC服务
- 微服务集成Seata

微服务集成Seata

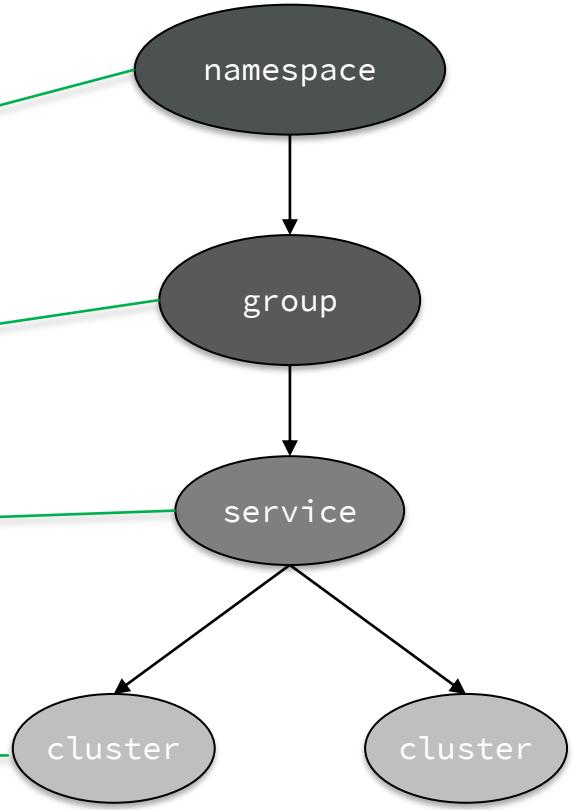
1. 首先，引入seata相关依赖：

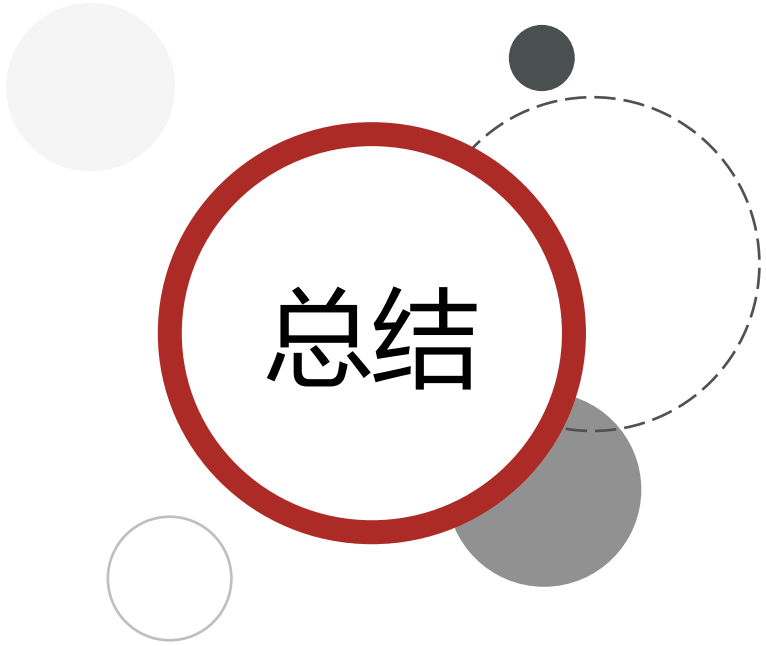
```
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
    <exclusions>
        <!--版本较低，1.3.0，因此排除-->
        <exclusion>
            <artifactId>seata-spring-boot-starter</artifactId>
            <groupId>io.seata</groupId>
        </exclusion>
    </exclusions>
</dependency>
<!--seata starter 采用1.4.2版本-->
<dependency>
    <groupId>io.seata</groupId>
    <artifactId>seata-spring-boot-starter</artifactId>
    <version>${seata.version}</version>
</dependency>
```

微服务集成Seata

2. 然后，配置application.yml，让微服务通过注册中心找到seata-tc-server：

```yaml
seata:
  registry: # TC服务注册中心的配置，微服务根据这些信息去注册中心获取tc服务地址
    # 参考tc服务自己的registry.conf中的配置，
    # 包括：地址、namespace、group、application-name 、cluster
    type: nacos
    nacos: # tc
      server-addr: 127.0.0.1:8848
      namespace: ""
      group: DEFAULT_GROUP
      application: seata-tc-server # tc服务在nacos中的服务名称
  tx-service-group: seata-demo # 事务组，根据这个获取tc服务的cluster名称
  service:
    vgroup-mapping: # 事务组与TC服务cluster的映射关系
      seata-demo: SH
```





nacos服务名称组成包括?

- namespace + group + serviceName + cluster

seata客户端获取tc的cluster名称方式?

- 以tx-group-service的值为key到vgroupMapping中查找

动手实践

- XA模式
- AT模式
- TCC模式
- SAGA模式

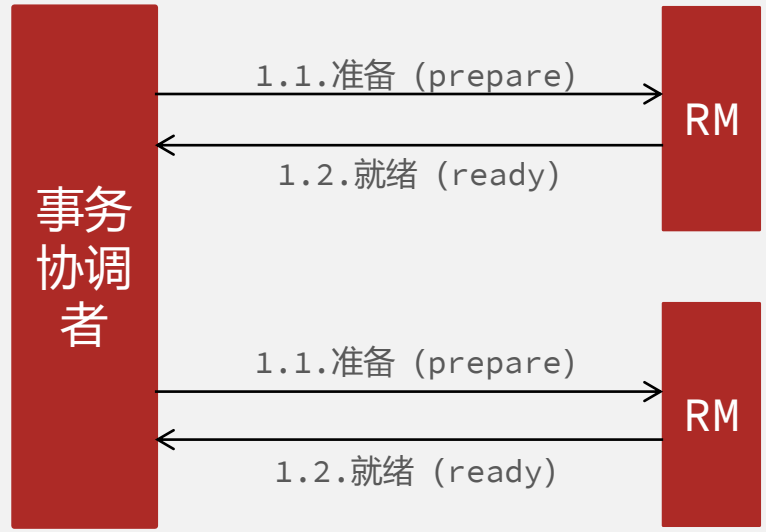
目录

Contents

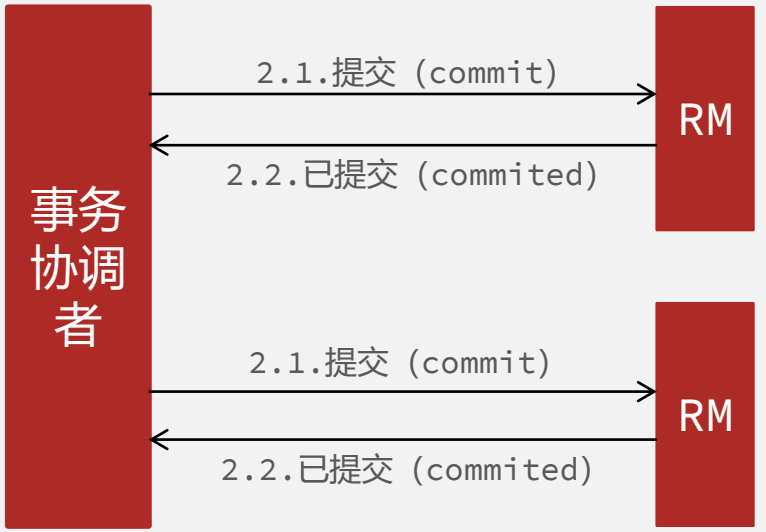
- XA模式
- AT模式
- TCC模式
- SAGA模式

XA模式原理

XA 规范 是 X/Open 组织定义的分布式事务处理（DTP，Distributed Transaction Processing）标准，XA 规范 描述了全局的TM与局部的RM之间的接口，几乎所有主流的数据库都对 XA 规范 提供了支持。



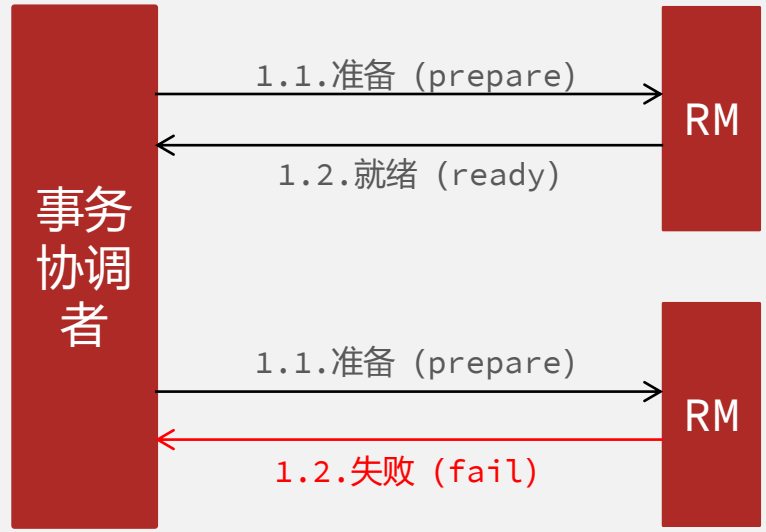
第一阶段



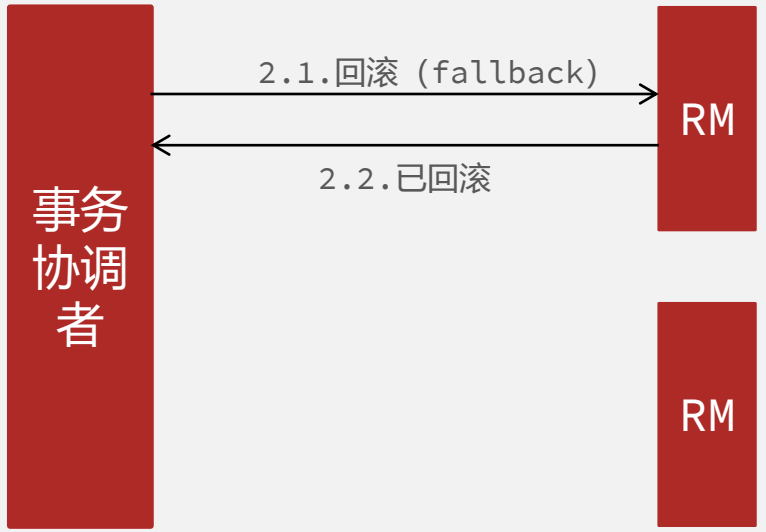
第二阶段

XA模式原理

XA 规范 是 X/Open 组织定义的分布式事务处理（DTP，Distributed Transaction Processing）标准，XA 规范 描述了全局的TM与局部的RM之间的接口，几乎所有主流的数据库都对 XA 规范 提供了支持。



第一阶段



第二阶段

seata的XA模式

seata的XA模式做了一些调整，但大体相似：

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

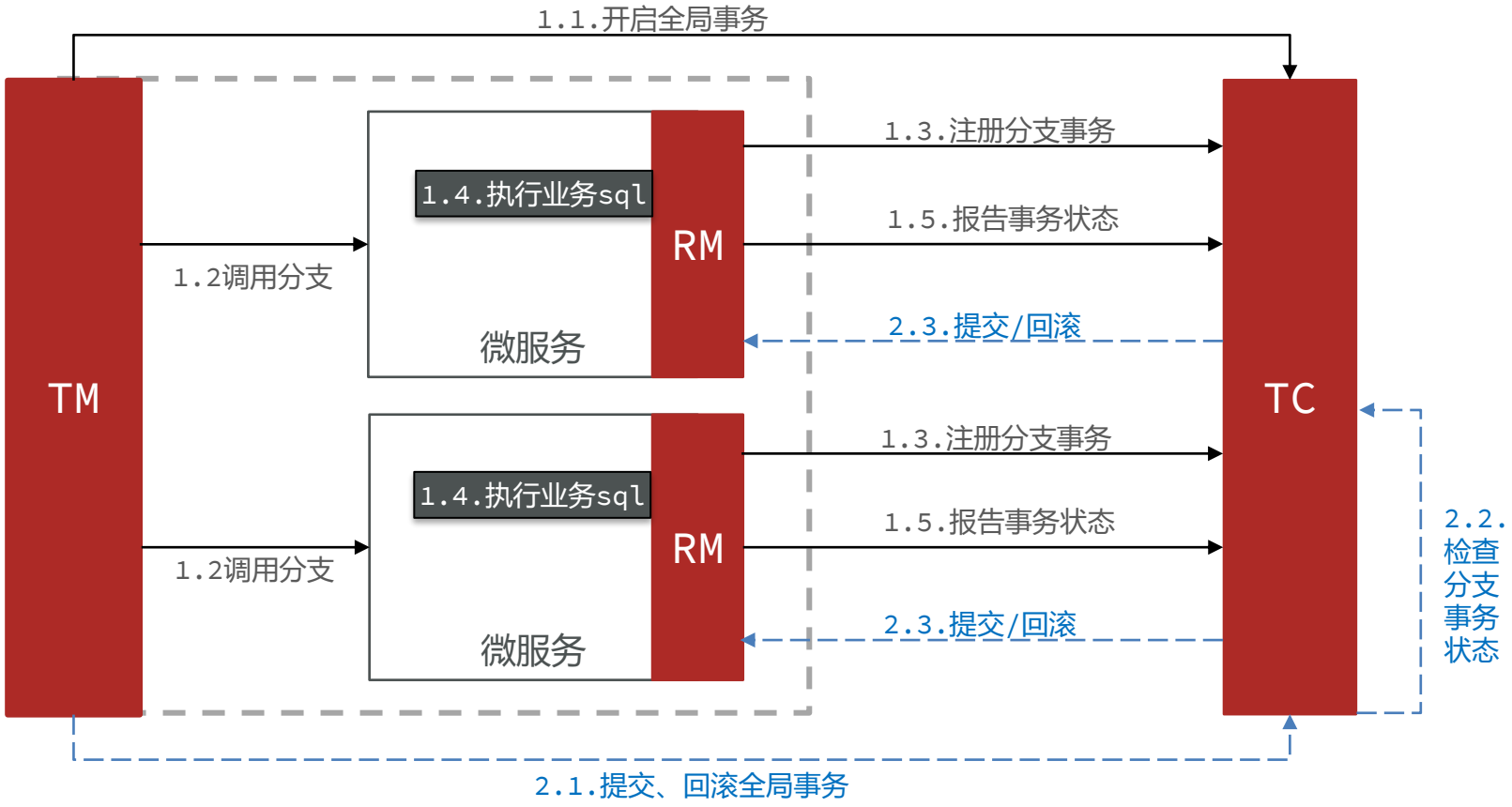
③ 报告执行状态到TC

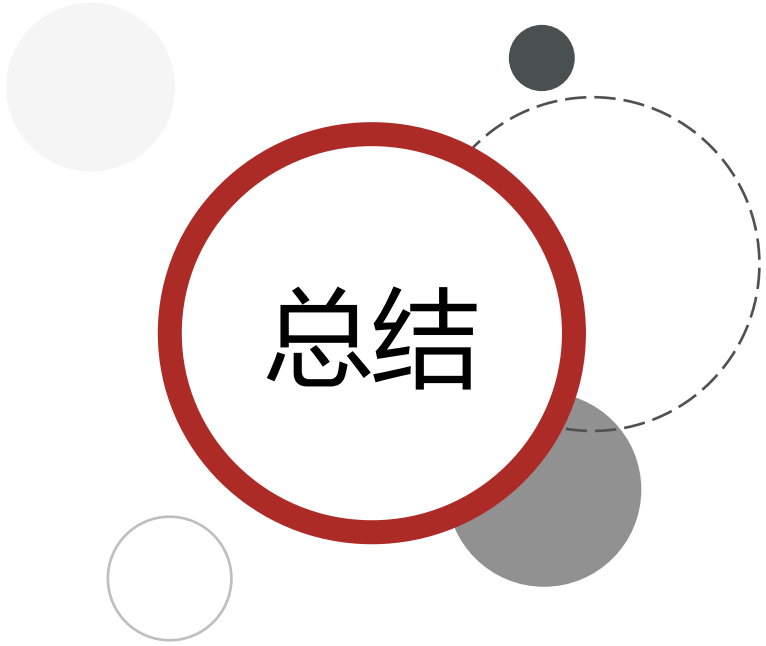
TC二阶段的工作：

- 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





XA模式的优点是什么？

- 事务的强一致性，满足ACID原则。
- 常用数据库都支持，实现简单，并且没有代码侵入

XA模式的缺点是什么？

- 因为一阶段需要锁定数据库资源，等待二阶段结束才释放，性能较差
- 依赖关系型数据库实现事务

实现XA模式

Seata的starter已经完成了XA模式的自动装配，实现非常简单，步骤如下：

1. 修改application.yml文件（每个参与事务的微服务），开启XA模式：

```yaml
seata:
  data-source-proxy-mode: XA # 开启数据源代理的XA模式
```

2. 给发起全局事务的入口方法添加@GlobalTransactional注解，本例中是OrderServiceImpl中的create方法：

```java
@Override
@GlobalTransactional
public Long create(Order order) {
    // 创建订单
    orderMapper.insert(order);
    // 扣余额 ...略
    // 扣减库存 ...略
    return order.getId();
}
```

3. 重启服务并测试

目录

Contents

- XA模式
- AT模式
- TCC模式
- SAGA模式

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：

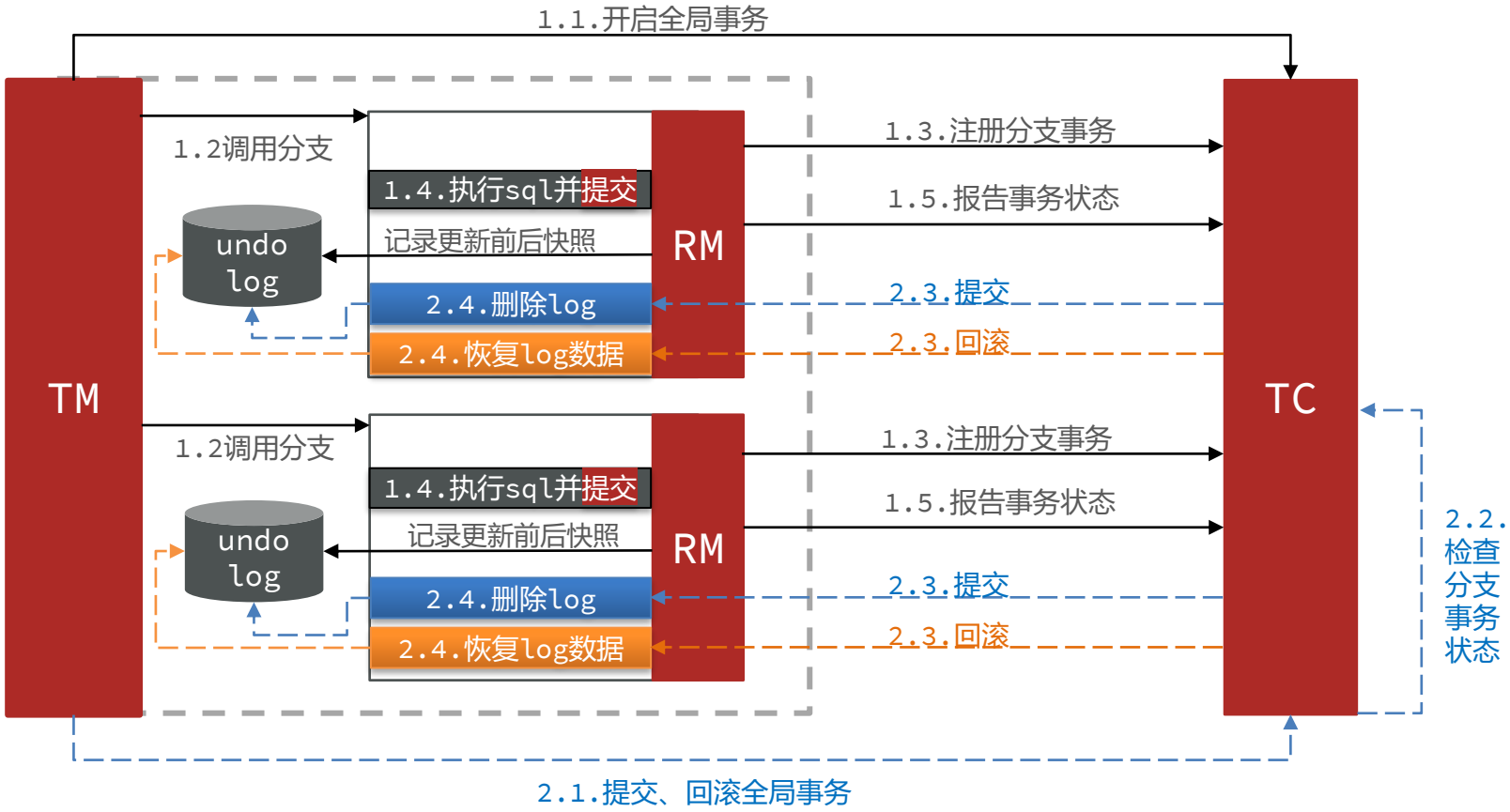
- 注册分支事务
- 记录undo-log（数据快照）
- 执行业务sql并提交
- 报告事务状态

阶段二提交时RM的工作：

- 删除undo-log即可

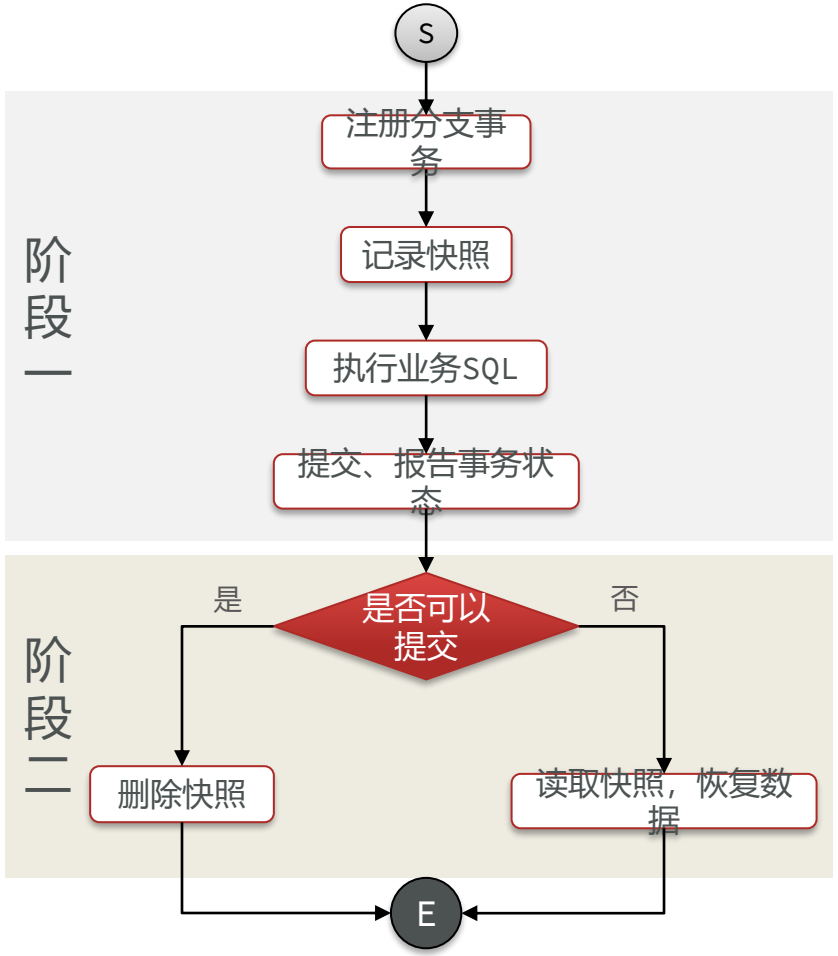
阶段二回滚时RM的工作：

- 根据undo-log恢复数据到更新前



AT模式原理

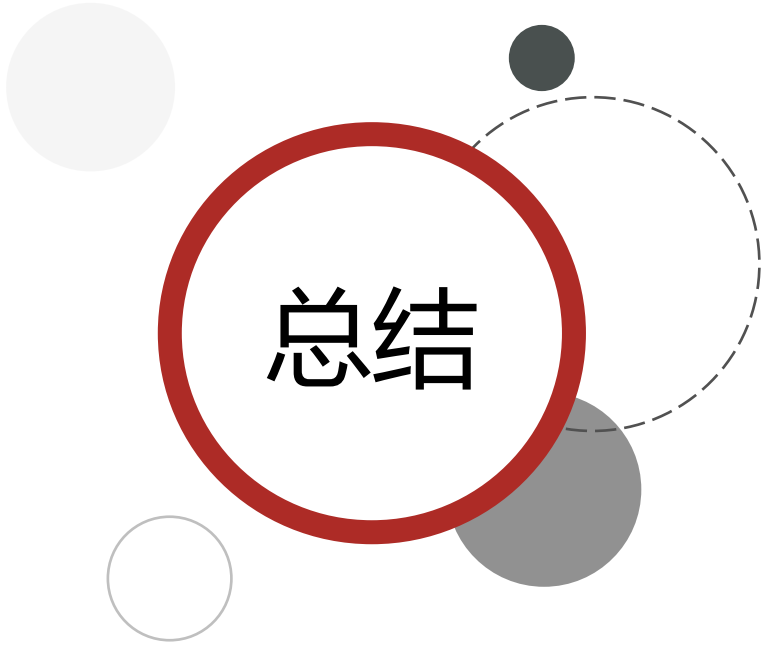
例如，一个分支业务的SQL是这样的：update tb_account set money = money - 10 where id = 1



id	money
1	1000

数据快照：

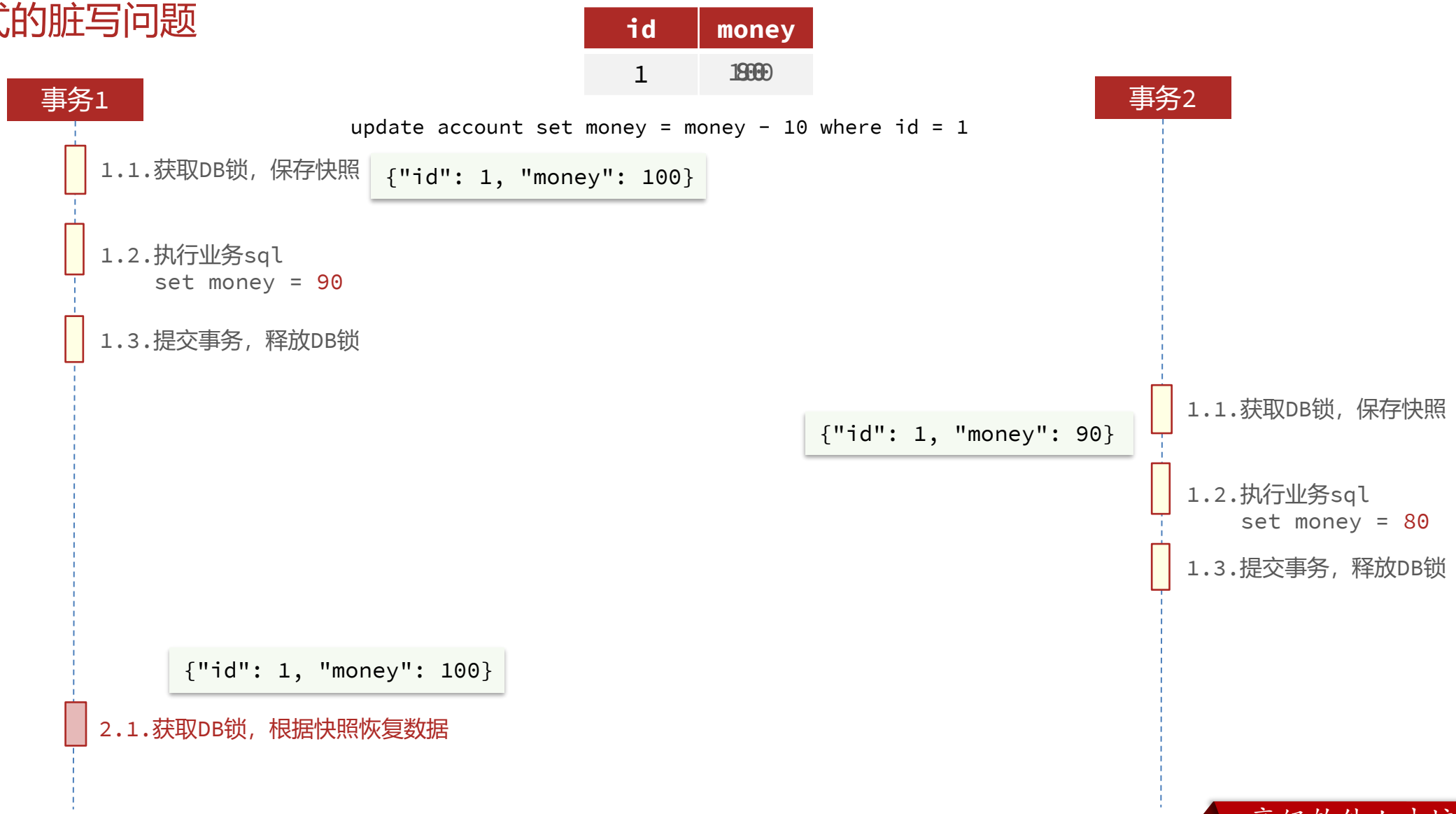
```
{
  "id": 1,
  "money": 100
}
```



简述AT模式与XA模式最大的区别是什么？

- XA模式一阶段不提交事务，锁定资源；AT模式一阶段直接提交，不锁定资源。
- XA模式依赖数据库机制实现回滚；AT模式利用数据快照实现数据回滚。
- XA模式强一致；AT模式最终一致

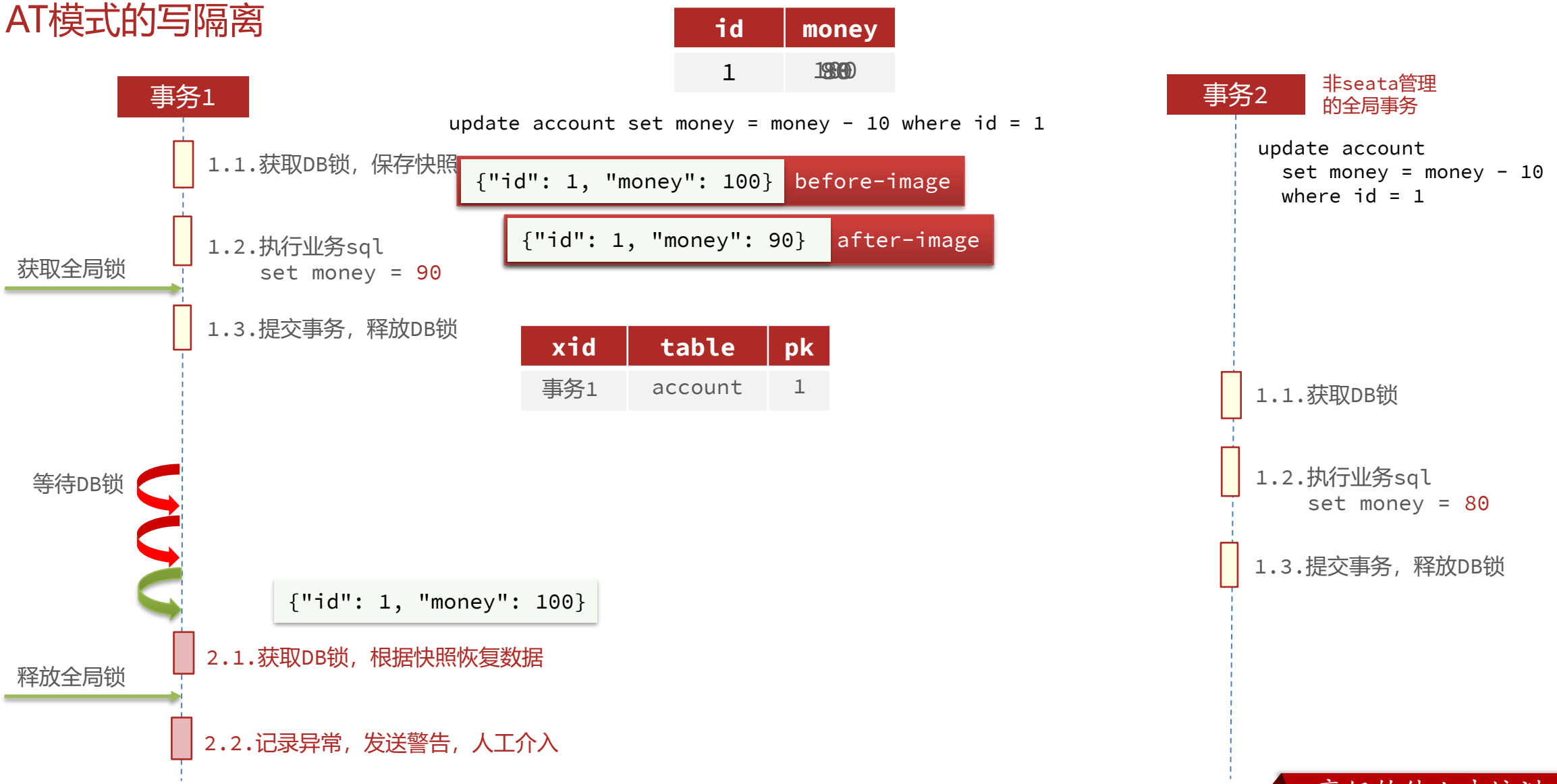
AT模式的脏写问题

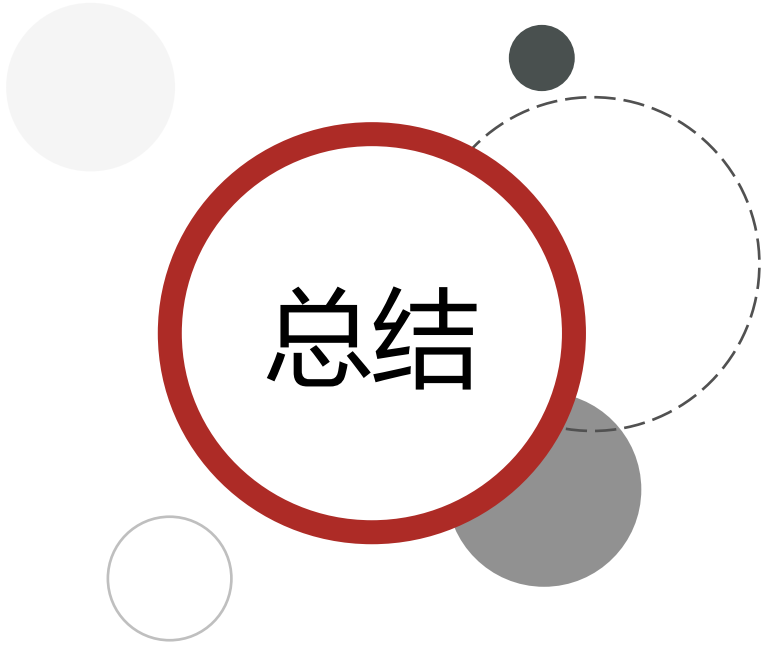

id
money
1
事务1
事务2
update account set money = money - 10 where id = 1
1.1.获取DB锁，保存快照
{"id": 1, "money": 100}
1.2.执行业务sql
set money = 90
1.3.提交事务，释放DB锁
{"id": 1, "money": 90}
1.1.获取DB锁，保存快照
1.2.执行业务sql
set money = 80
1.3.提交事务，释放DB锁
{"id": 1, "money": 100}
2.1.获取DB锁，根据快照恢复数据

AT模式的写隔离

id	money
1	1000

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
{"id": 1, "money": 90}
after-image
1.2.执行业务sql
set money = 90
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



AT模式的优点：

- 一阶段完成直接提交事务，释放数据库资源，性能比较好
- 利用全局锁实现读写隔离
- 没有代码侵入，框架自动完成回滚和提交

AT模式的缺点：

- 两阶段之间属于软状态，属于最终一致
- 框架的快照功能会影响性能，但比XA模式要好很多

实现AT模式

AT模式中的快照生成、回滚等动作都是由框架自动完成，没有任何代码侵入，因此实现非常简单。

1. 导入课前资料提供的Sql文件：seata-at.sql，其中lock_table导入到TC服务关联的数据库，undo_log表导入到微服务关联的数据库：

seata-at.sql

2. 修改application.yml文件，将事务模式修改为AT模式即可：

```yaml
seata:
  data-source-proxy-mode: AT # 开启数据源代理的AT模式
```

3. 重启服务并测试

目录

Contents

- XA模式
- AT模式
- TCC模式
- SAGA模式

TCC模式原理

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。

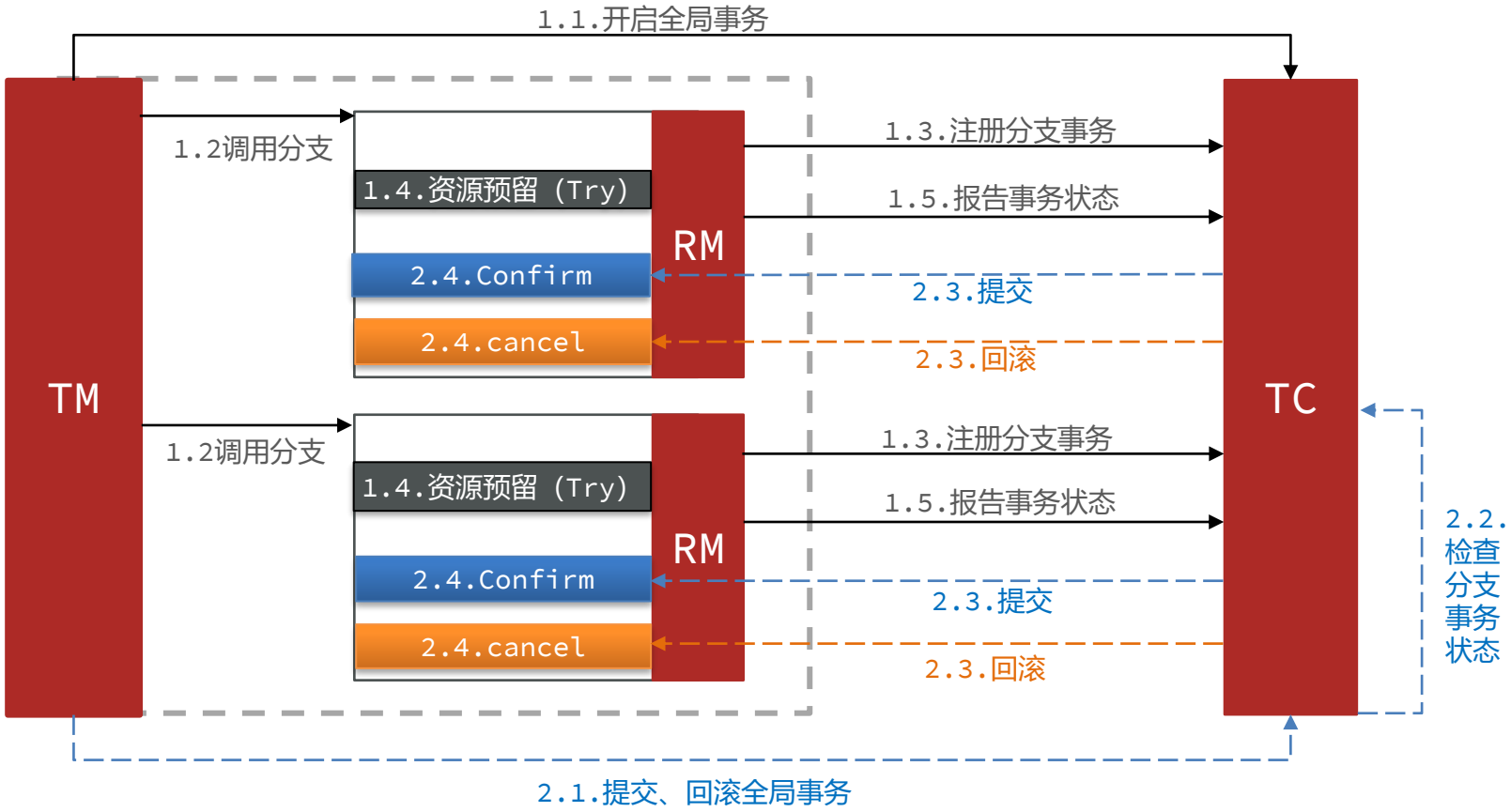
- 阶段一（Try）：检查余额是否充足，如果充足则冻结金额增加30元，可用余额扣除30

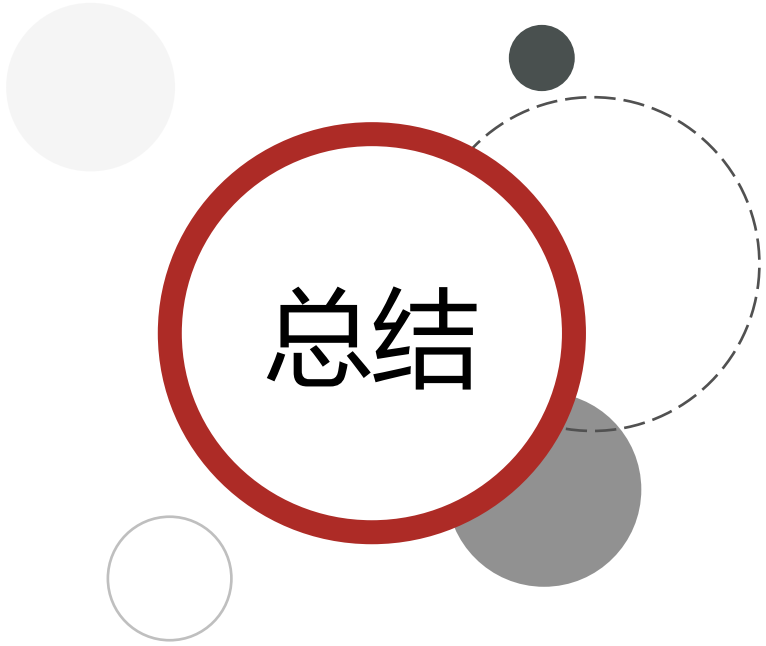
- 阶段二：假如要提交（Confirm），则冻结金额扣减30

- 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

案例 改造account-service服务，利用TCC实现分布式事务

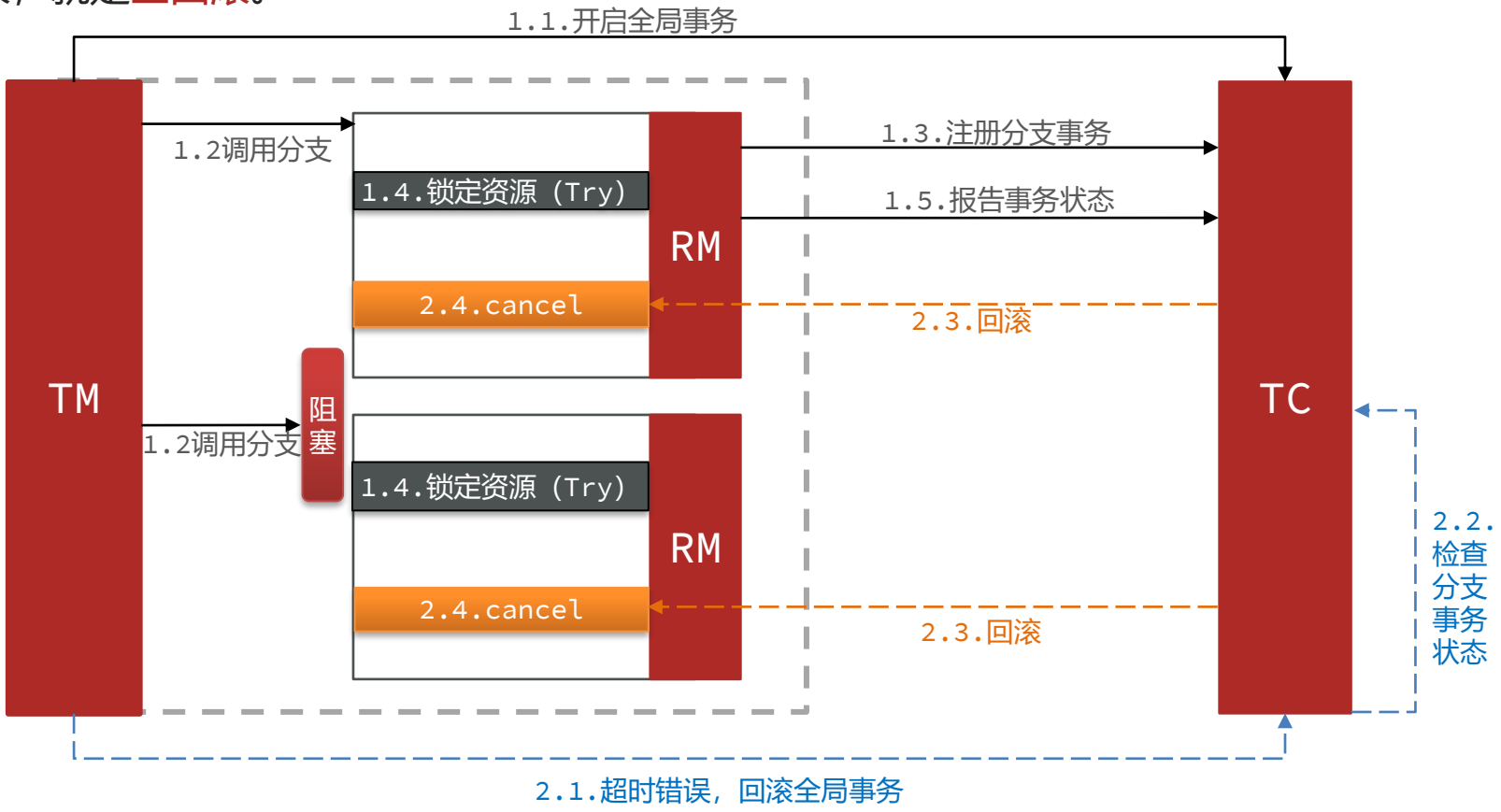
需求如下：

- 修改account-service，编写try、confirm、cancel逻辑
- try业务：添加冻结金额，扣减可用金额
- confirm业务：删除冻结金额
- cancel业务：删除冻结金额，恢复可用金额
- 保证confirm、cancel接口的幂等性
- 允许空回滚
- 拒绝业务悬挂

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



业务分析

为了实现空回滚、防止业务悬挂，以及幂等性要求。我们必须在数据库记录冻结金额的同时，记录当前事务id和执行状态，为此我们设计了一张表：

```
CREATE TABLE `account_freeze_tbl` (
  `xid` varchar(128) NOT NULL,
  `user_id` varchar(255) DEFAULT NULL COMMENT '用户id',
  `freeze_money` int(11) unsigned DEFAULT '0' COMMENT '冻结金额',
  `state` int(1) DEFAULT NULL COMMENT '事务状态，0:try，1:confirm，2:cancel',
  PRIMARY KEY (`xid`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8 ROW_FORMAT=COMPACT;
```

Try业务

- 记录冻结金额和事务状态到account_freeze表
- 扣减account表可用金额

Confirm业务

- 根据xid删除account_freeze表的冻结记录

Cancel业务

- 修改account_freeze表，冻结金额为0，state为2
- 修改account表，恢复可用金额

如何判断是否空回滚

- cancel业务中，根据xid查询account_freeze，如果为null则说明try还没做，需要空回滚

如何避免业务悬挂

- try业务中，根据xid查询account_freeze ，如果已经存在则证明Cancel已经执行，拒绝执行try业务

声明TCC接口

TCC的Try、Confirm、Cancel方法都需要在接口中基于注解来声明，语法如下：

```
@LocalTCC
public interface TCCService {
    /**
     * Try逻辑，@TwoPhaseBusinessAction中的name属性要与当前方法名一致，用于指定Try逻辑对应的方法
     */
    @TwoPhaseBusinessAction(name = "prepare", commitMethod = "confirm", rollbackMethod = "cancel")
    void prepare(@BusinessActionContextParameter(paramName = "param") String param);
    /**
     * 二阶段confirm确认方法、可以另命名，但要保证与commitMethod一致
     *
     * @param context 上下文,可以传递try方法的参数
     * @return boolean 执行是否成功
     */
    boolean confirm (BusinessActionContext context);
    /**
     * 二阶段回滚方法，要保证与rollbackMethod一致
     */
    boolean cancel (BusinessActionContext context);
}
```

目录

Contents

- XA模式
- AT模式
- TCC模式
- Saga模式

Saga模式

Saga模式是SEATA提供的长事务解决方案。也分为两个阶段：

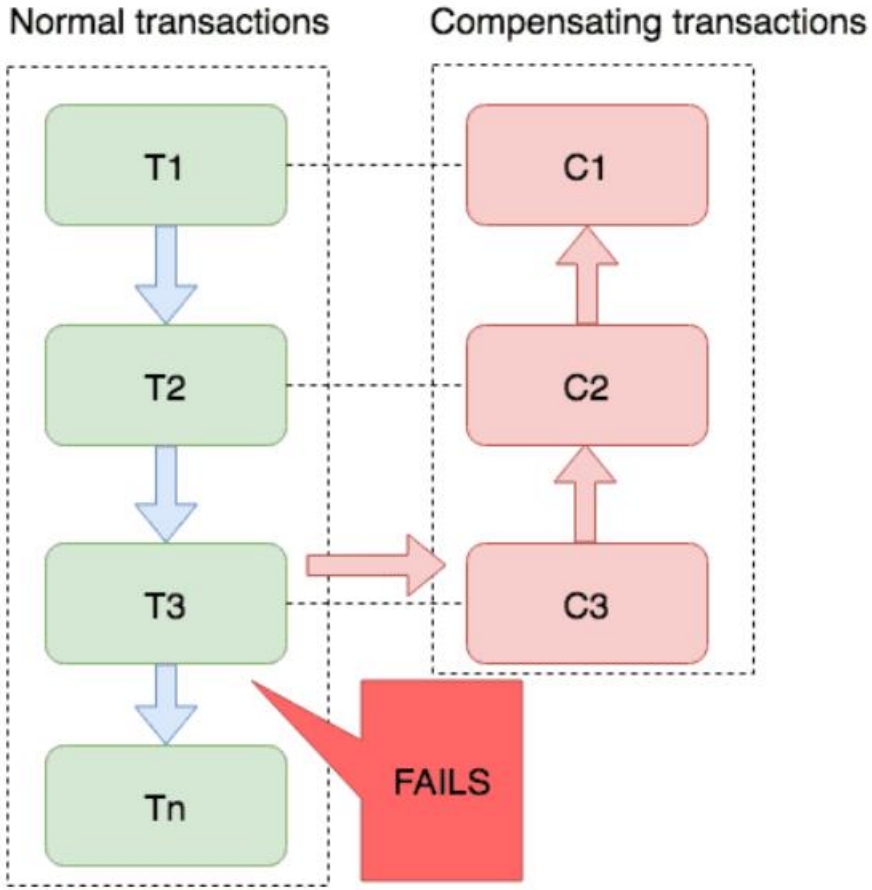
- 一阶段：直接提交本地事务
- 二阶段：成功则什么都不做；失败则通过编写补偿业务来回滚

Saga模式优点：

- 事务参与者可以基于事件驱动实现异步调用，吞吐高
- 一阶段直接提交事务，无锁，性能好
- 不用编写TCC中的三个阶段，实现简单

缺点：

- 软状态持续时间不确定，时效性差
- 没有锁，没有事务隔离，会有脏写



四种模式对比

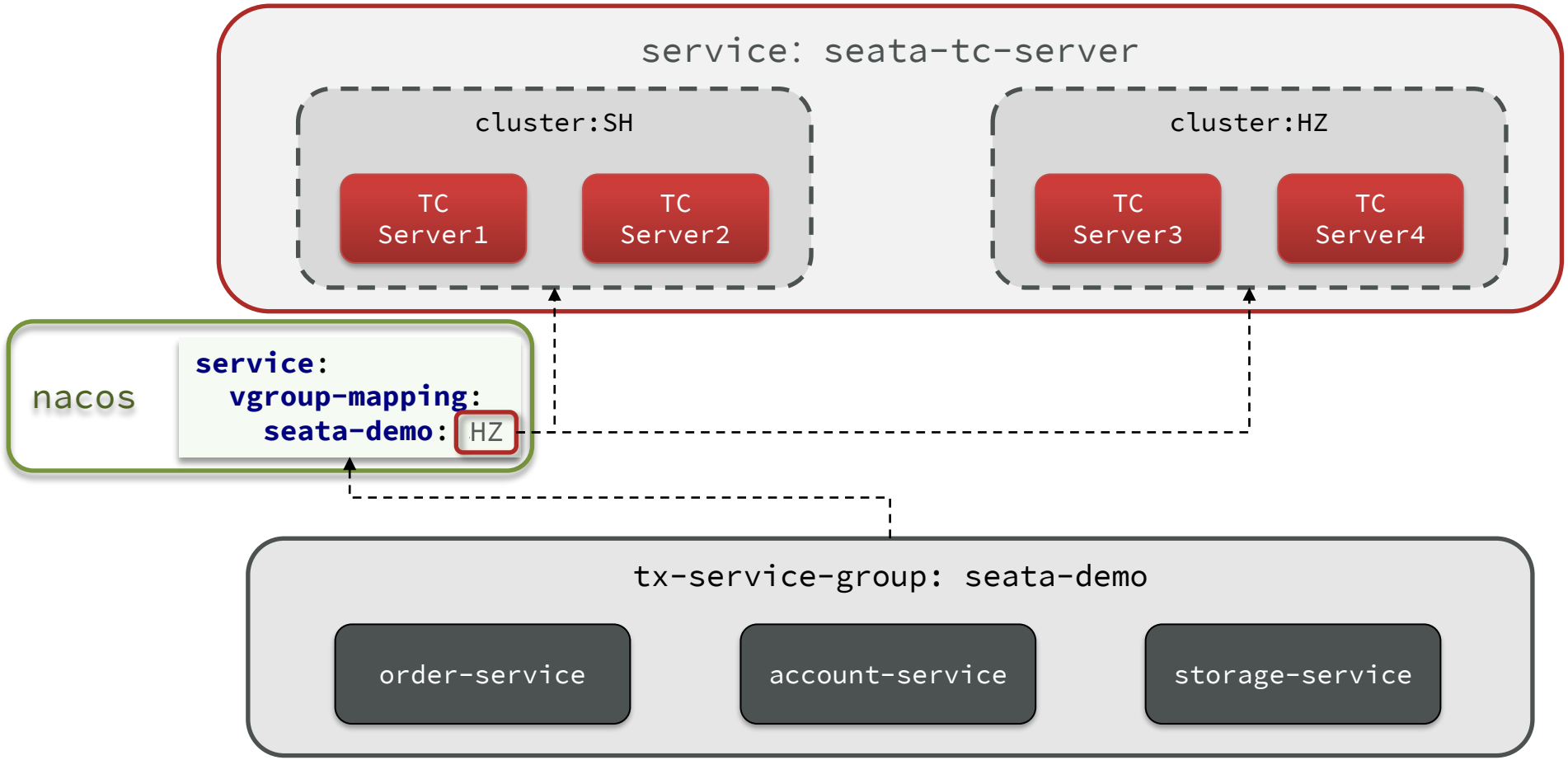
	XA	AT	TCC	SAGA
一致性	强一致	弱一致	弱一致	最终一致
隔离性	完全隔离	基于全局锁隔离	基于资源预留隔离	无隔离
代码侵入	无	无	有，要编写三个接口	有，要编写状态机和补偿业务
性能	差	好	非常好	非常好
场景	对一致性、隔离性有高要求的业务	基于关系型数据库的大多数分布式事务场景都可以	• 对性能要求较高的事务。 • 有非关系型数据库要参与的事务。	• 业务流程长、业务流程多 • 参与者包含其它公司或遗留系统服务，无法提供 TCC 模式要求的三个接口

高可用

- 高可用集群结构
- 实现高可用集群

TC的异地多机房容灾架构

TC服务作为Seata的核心服务，一定要保证高可用和异地容灾。



TC的异地多机房容灾架构

具体实现请参考课前资料提供的文档《seata的部署和集成.md》：

传智教育旗下高端IT教育品牌