分布式事务

CAP 定理

> 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
>
> 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
>
> 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

BASE 理论

> 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。

Seata 架构

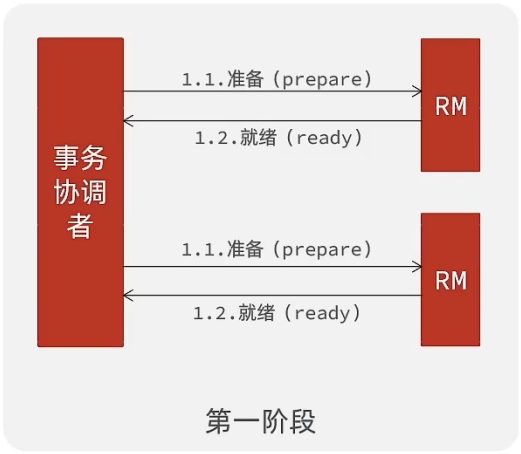
Seata是解决分布式事务问题的解决产品，在事务管理中有三个重要的角色：

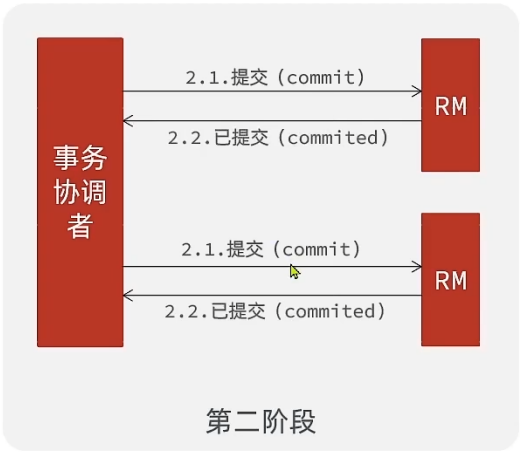
> TC （Transaction Coordinator）-事务协调者：维护全局和分支事务的状态，协调全局事务提交或回滚。
>
> TM （Transaction Manager）-事务管理器：定义全局事务的范围、开始全局事务、提交或回滚全局事务
>
> RM （Resource Manager）-资源管理器：管理分支事务处理的资源(：交谈以注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚。

Seata提供了四种不同的分布式事务解决方案

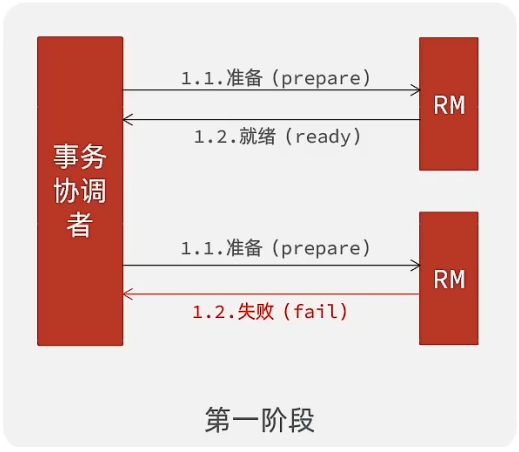
- XA模式：强一致性分阶段事务模式，牺牲了一定的可用性，无业务侵入
- TCC模式：最终一致的分阶段事务模式，有业务侵入
- AT模式：最终一致的分阶段事务模式，无业务侵入，也是Seata的默认模式
- SAGA 模式：长事务模式，有业务侵入

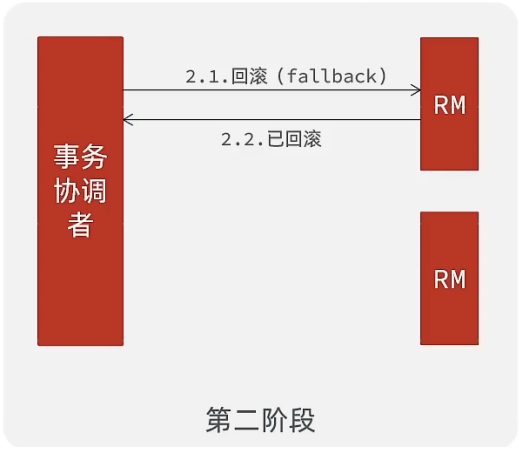
XA规范是X/Open 组织定义的分布式事务处理（DTP, Distributed Transaction Processing) 标准，XA规范描述了全局的TM与局部的RM之间的接口，几乎所有主流的数据库都对XA规范提供了支持。





如果其中有一个失败：

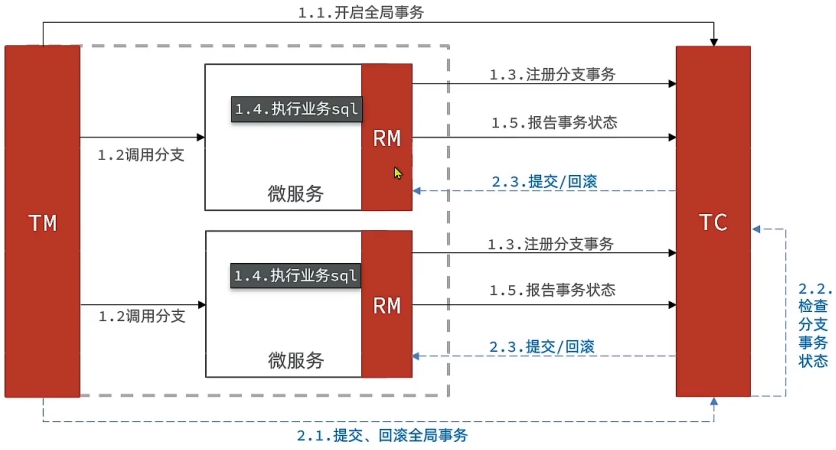




Seata-XA

seata的XA模式

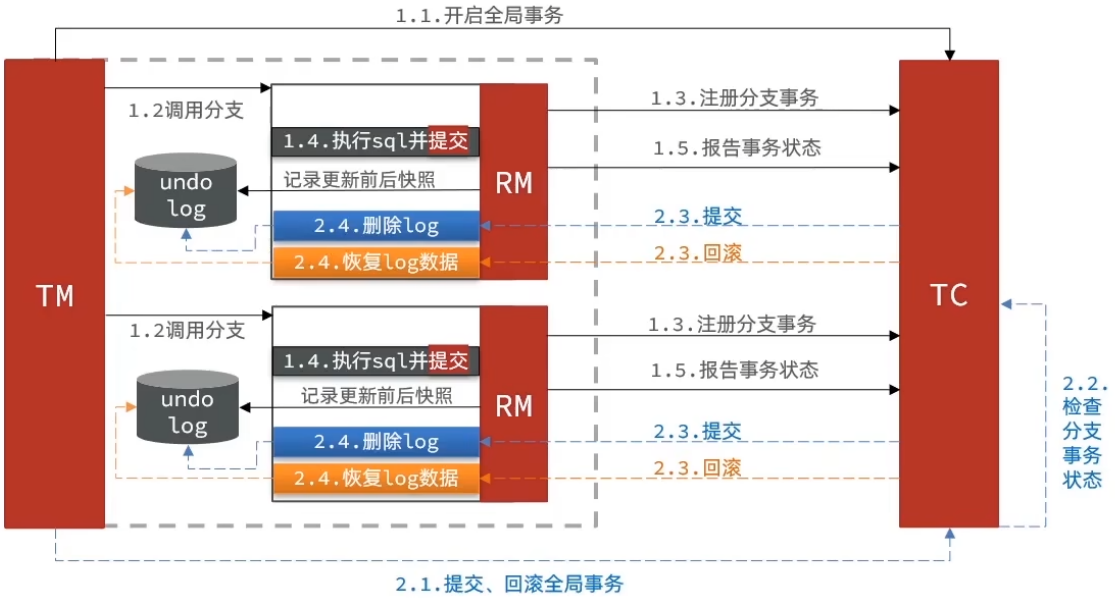
seata的XA模式做了一些调整，但大体相似：



Seata-TA

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。



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

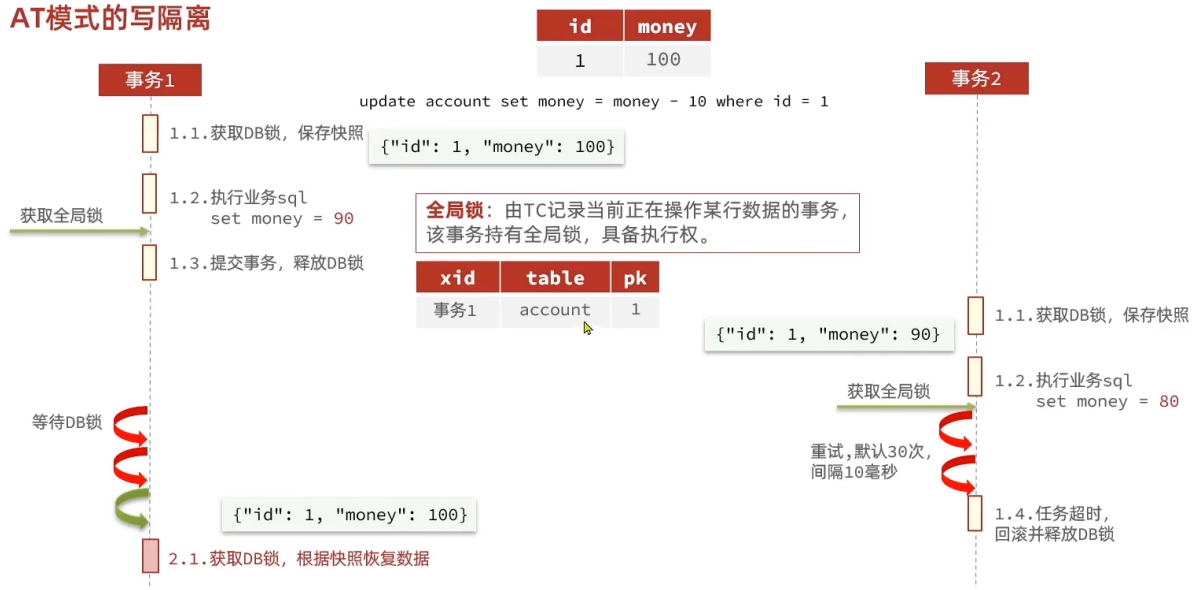
id	money
1	100

数据快照：

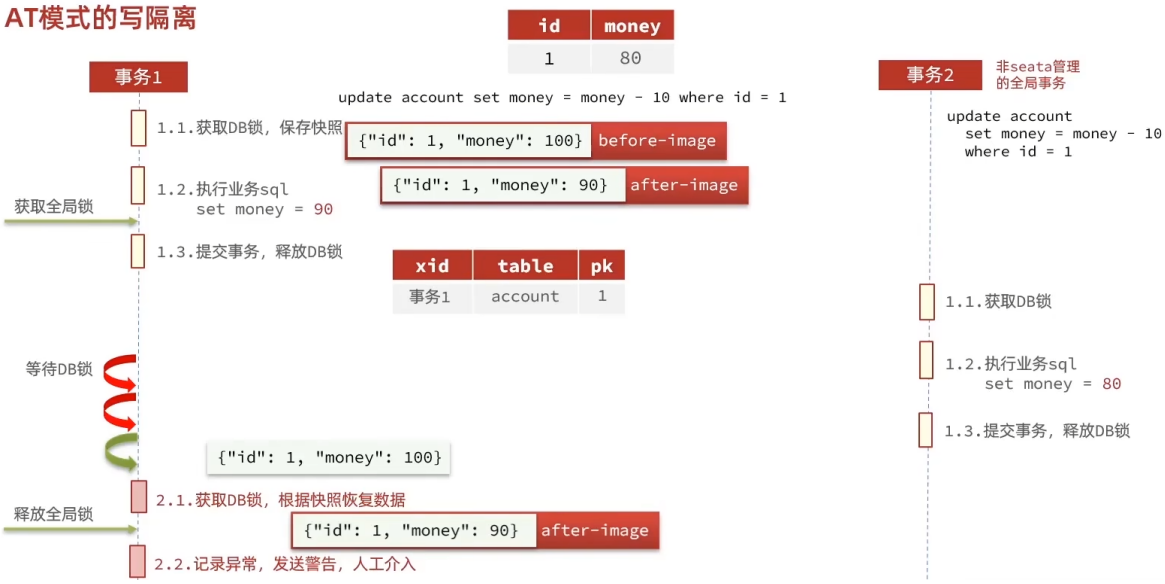
```
{
  "id": 1,
  "money": 100
}
```

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



全局锁是TC所持有的，其他非Seata事务可以直接修改，不守锁限制



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

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。

Seata-TCC

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：

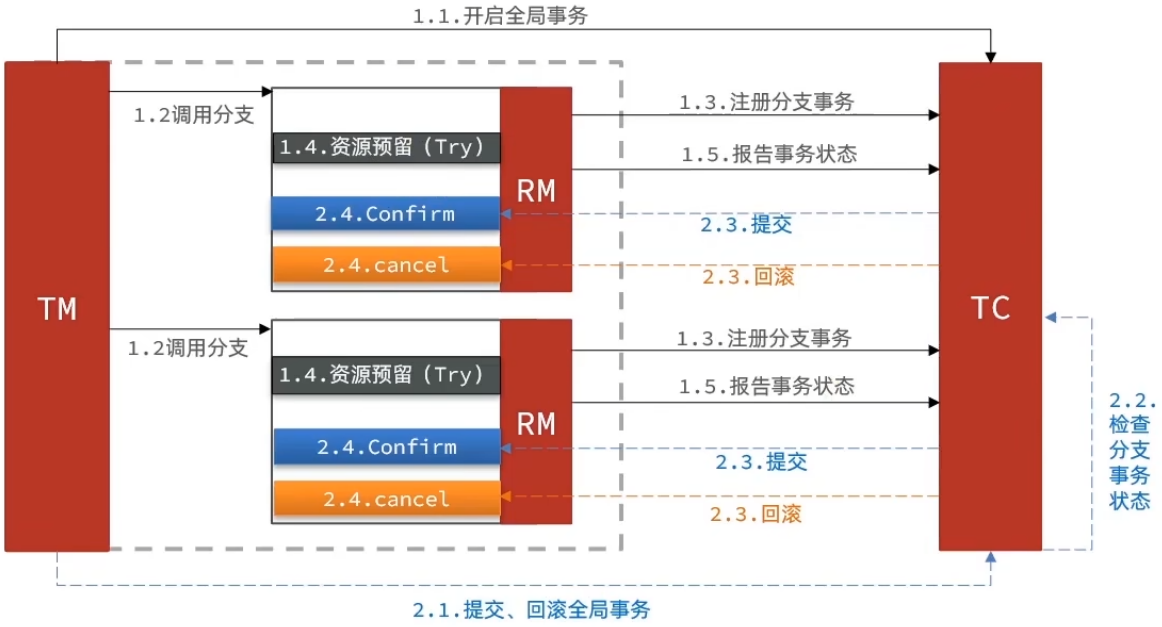
- Try：资源的检测和预留；
- Confirm：完成资源操作业务；要求 Try 成功 Confirm 一定要能成功。
- Cancel：预留资源释放，可以理解为try的反向操作。

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

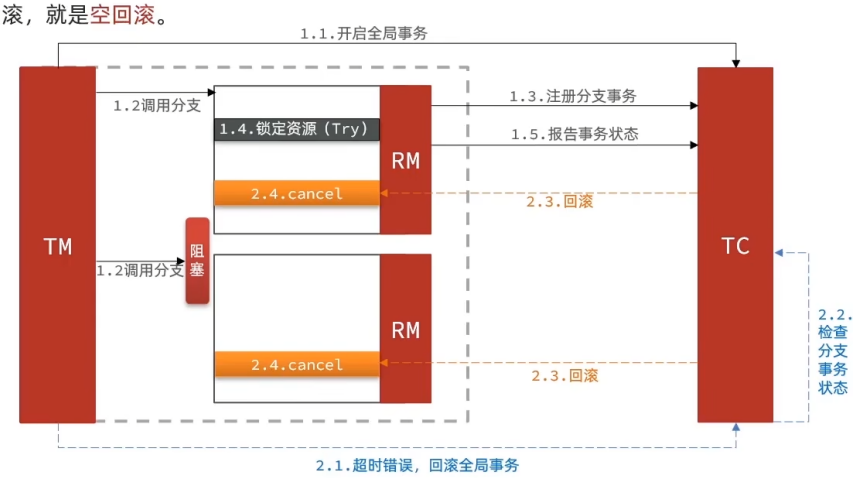
TCC的缺点是什么？

- 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

TCC的空回滚和业务悬挂

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

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。



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

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);
}
```

Seata-Saga

Saga模式是SEATA提供的长事务解决方案。也分为两个阶段：

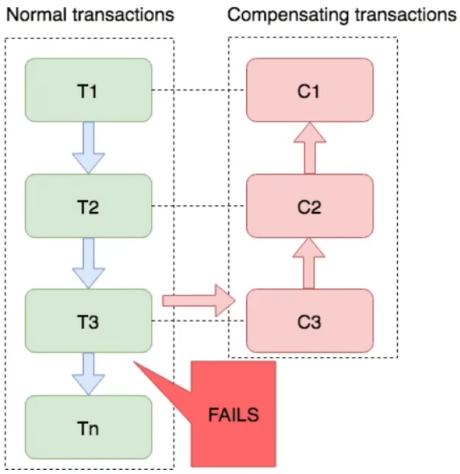
- 一阶段：直接提交本地事务
- 二阶段：成功则什么都不做；失败则通过编写补偿业务来回滚

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



总结对比

	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