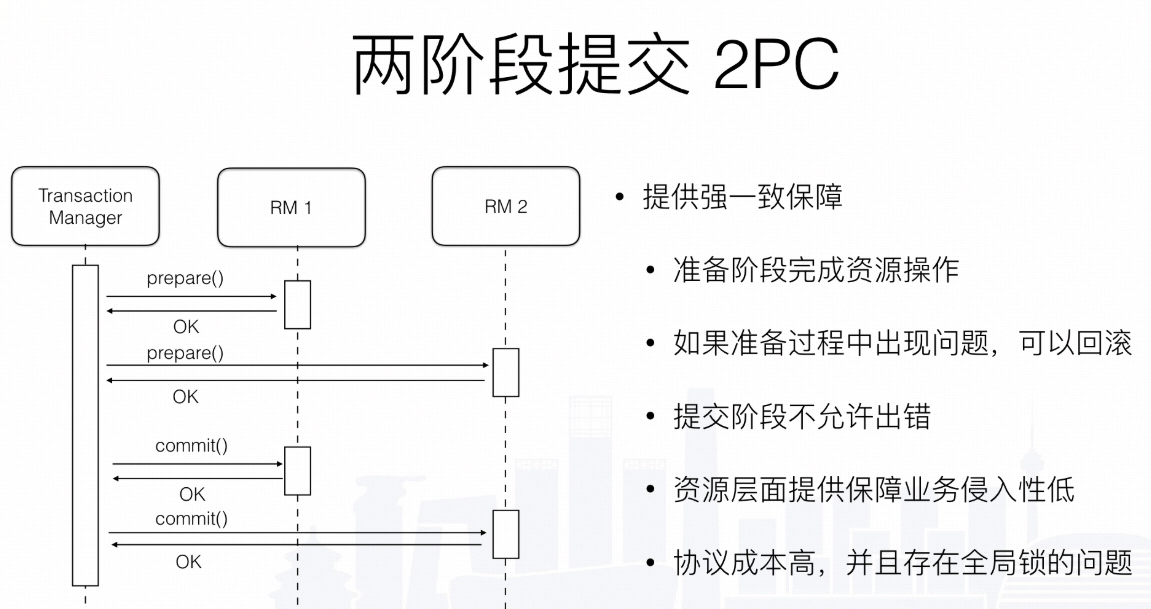
# 1、分布式事物的解决方案

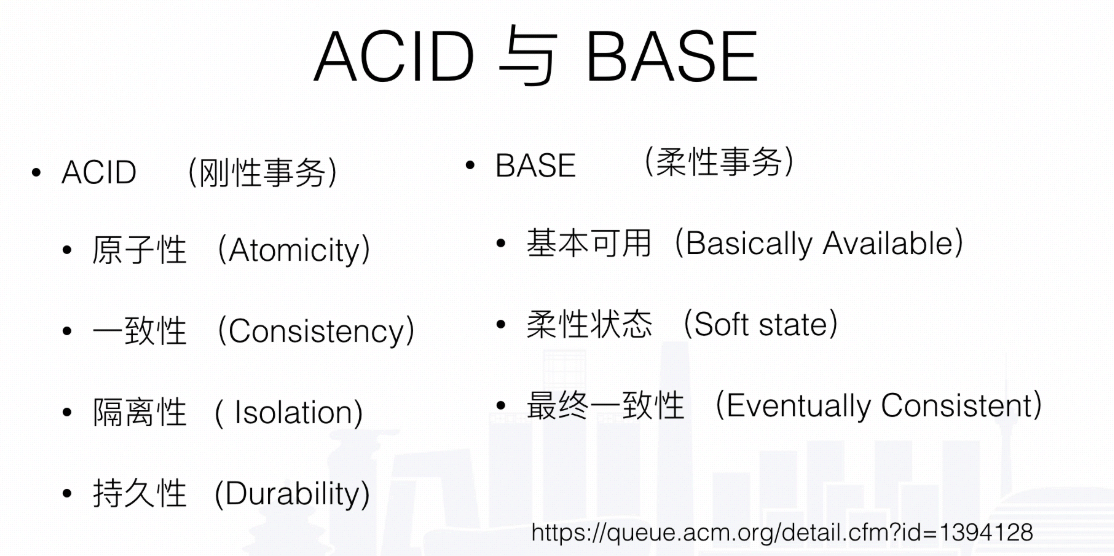
## 1.1 XA刚性事物

两阶段事务提交采用的是X/OPEN组织所定义的[DTP模型](http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf)，通过抽象出来的AP, TM, RM的概念可以保证事务的强一致性。 其中TM和RM间采用XA的协议进行双向通信。 与传统的本地事务相比，XA事务增加了prepare阶段，数据库除了被动接受提交指令外，还可以反向通知调用方事务是否可以被提交。 因此TM可以收集所有分支事务的prepare结果，最后进行原子的提交，保证事务的强一致性。事务管理器国内使用Atomikos较多，其它主流事务管理器 Narayana和Bitronix

缺点：由于隔离性互斥的要求，在事务执行过程中，所有的资源都是被锁定的，这种情况只适合执行时间确定的短事务。 但是为了保证分布式事务的一致性，大都是采用串行化的隔离级别来保证事务一致性，这样会降低系统的吞吐，同时两阶段提交在提交阶段不允许出错，如果这个阶段发生异常无法回滚，会造成数据异常。

但因为2PC的协议成本比较高，又有全局锁的问题，性能会比较差。现在大家基本上不会采用这种强一致解决方案

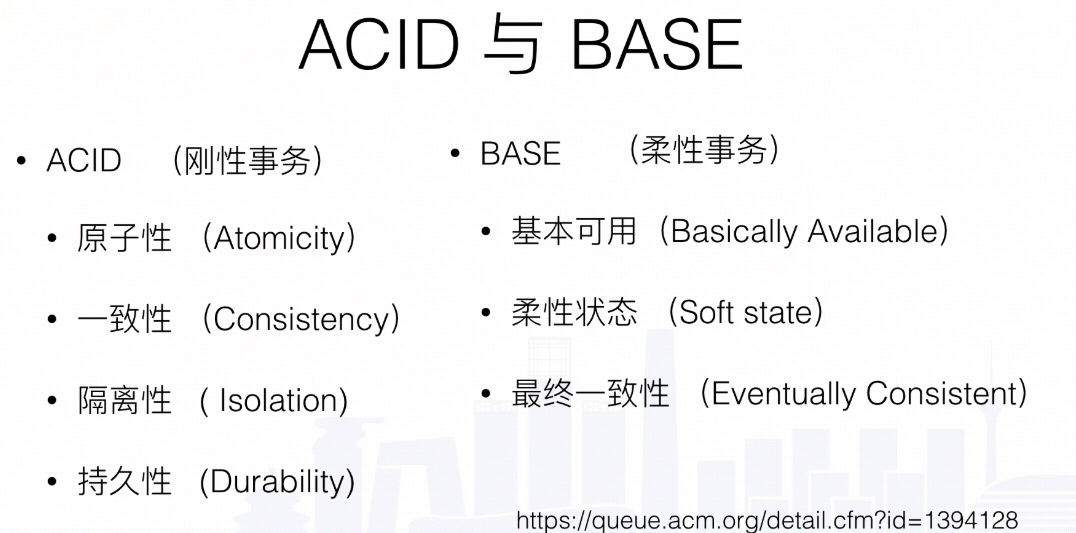




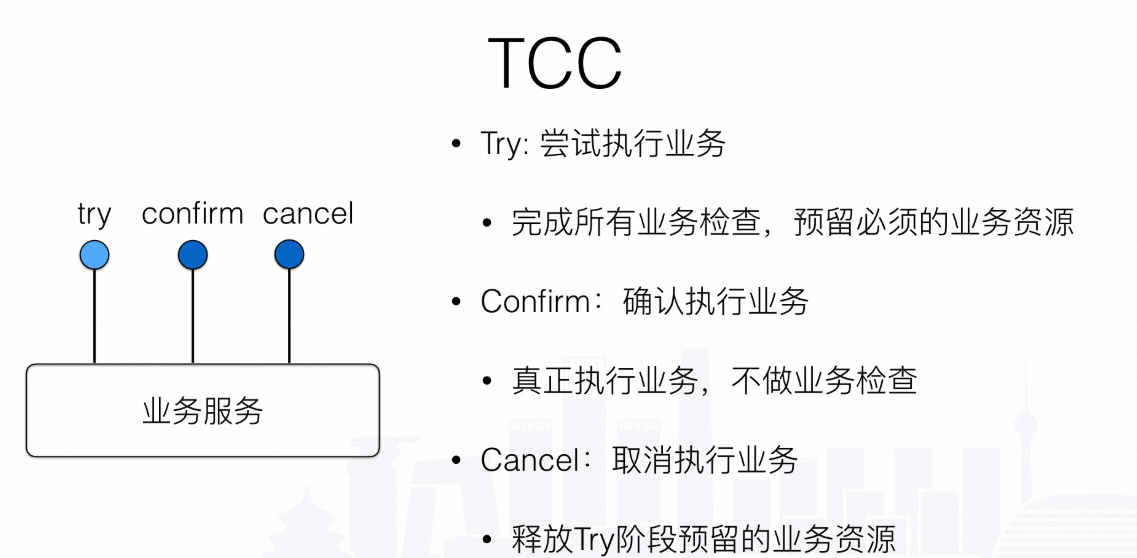
## 1.2 柔性事物

BASE基本可用 可以保证分布式事务参与方不一定同时在线。柔性状态 允许系统状态更新有一定的延时，这个延时对客户来说不一定能察觉。最终一致性 通常是通过消息可达的方式保证系统的最终一致性。

通过业务逻辑将互斥锁操作从资源层面上移到业务层面，这并不是完全放弃了ACID，而是通过放宽一致性要求，借助本地事务来实现最终分布式事务一致性的同时也保证系统的吞吐。



### 2.1 tcc实现方式



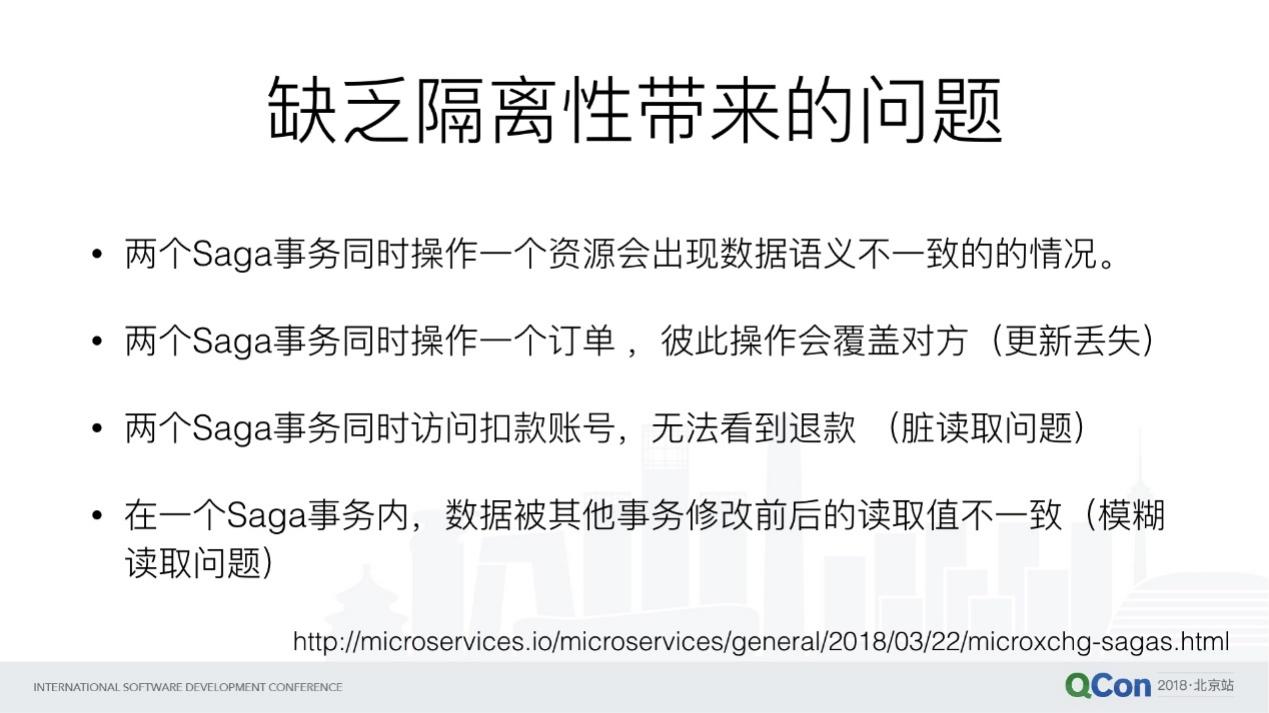
TCC名字的由来是其中包含了 try, confirm, cancel三个操作。

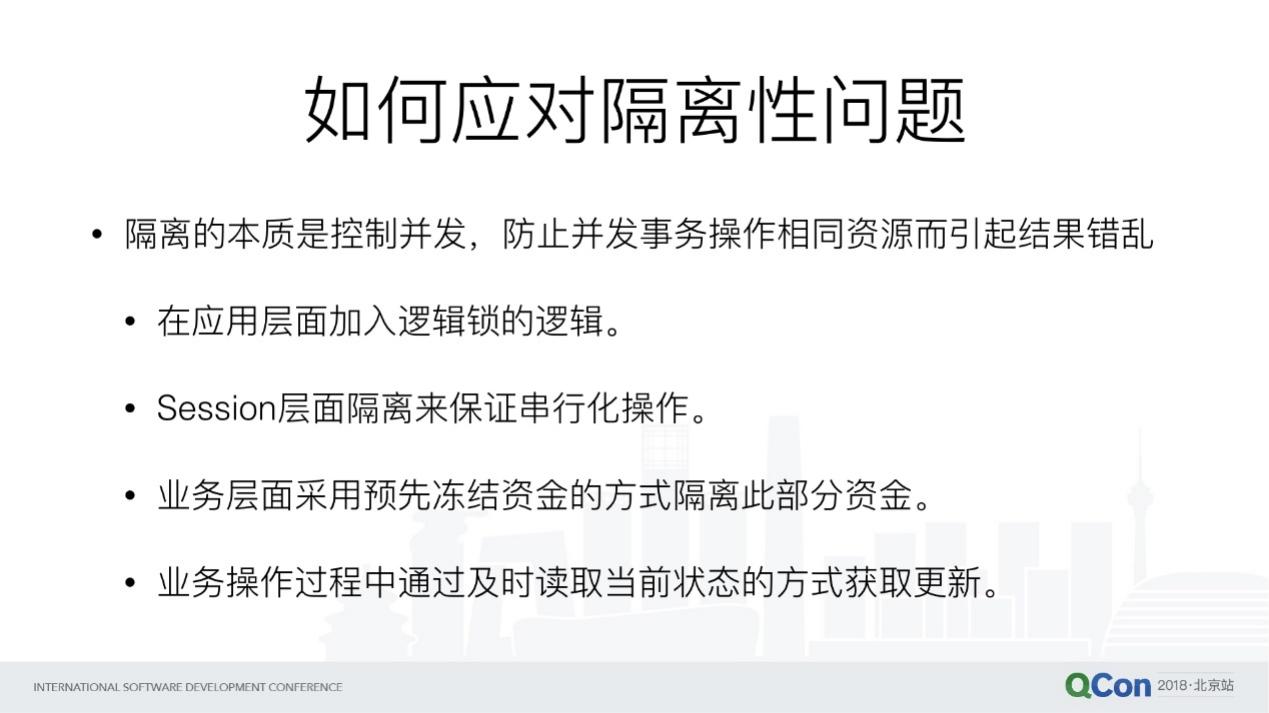
与两阶段提交相比，TCC位于业务服务层， 没有单独的准备阶段，Try操作可以灵活选择业务资源锁的粒度。TCC是通过最终一致性来解决系统性能问题的这个设计，对我们设计抉择有很大的启发。有些时候系统的技术问题是可以通过业务建模的方式来解决的。

Tcc使用很复杂，需要提供try预留资源逻辑，还需要confirm真正改变数据的逻辑，与cancel取消资源修改的逻辑，相当于每个分布式调用都需要手动实现功能的补偿，代码量较大，但不会锁表，性能消耗主要体现在各个事务的执行，以及状态的同步上时间的消耗。

### 2.2 saga的实现方式

[](https://servicecomb.apache.org/assets/images/saga/Saga.019.jpeg#)

[](https://servicecomb.apache.org/assets/images/saga/Saga.020.jpeg#)



#### 2.1 基于saga的华为ServiceComb

未解决隔离性问题

<https://servicecomb.apache.org/cn/docs/distributed-transactions-saga-implementation/>

#### 2.2 基于saga升级的seata

解决了隔离性问题