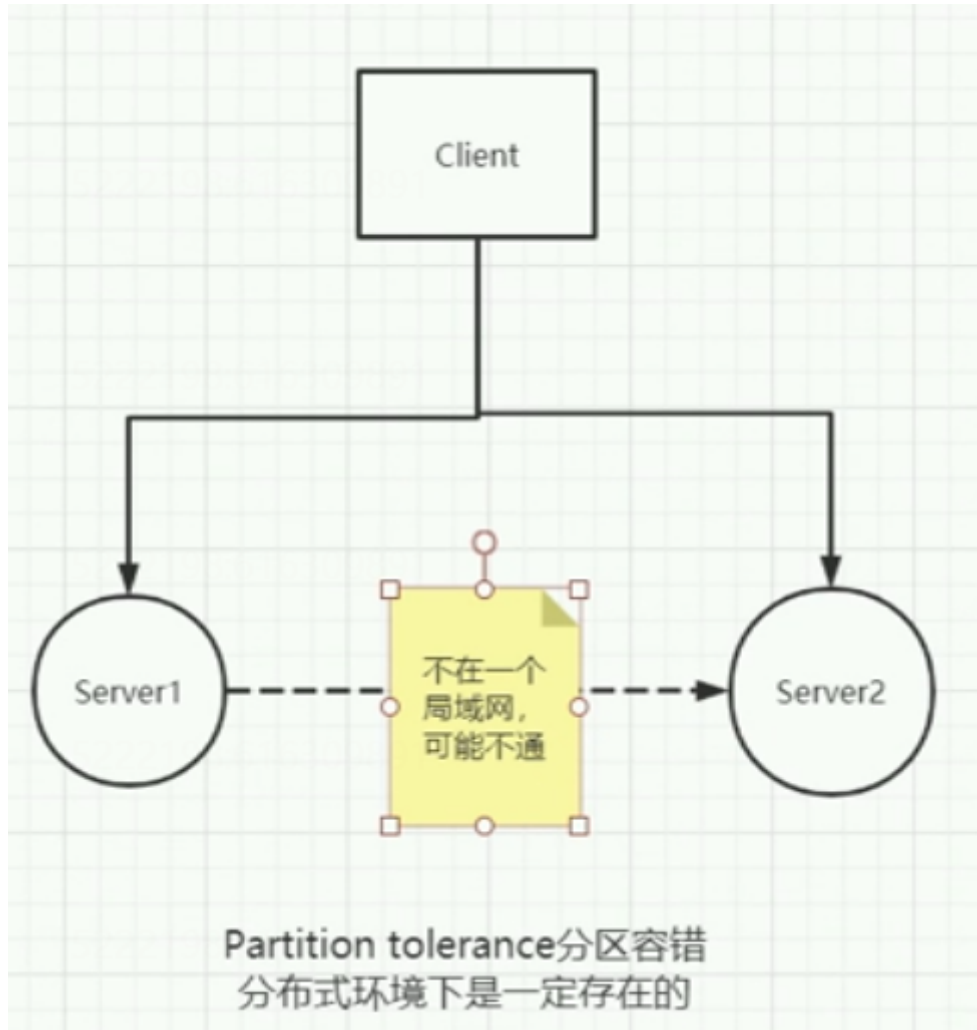


# CAP分布式事务

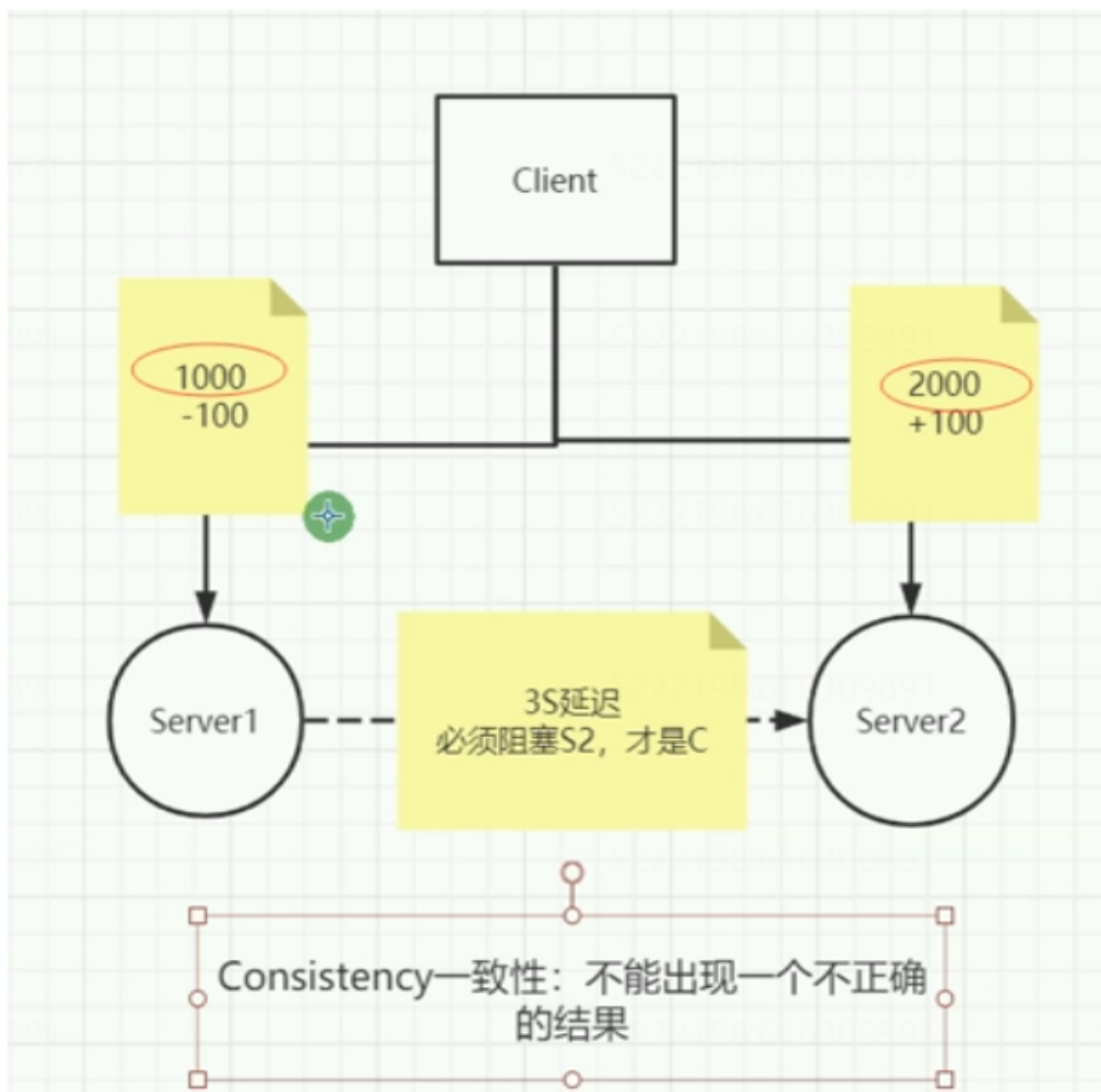
## cap定理：

## 结论：CAP是不能够被同时满足的

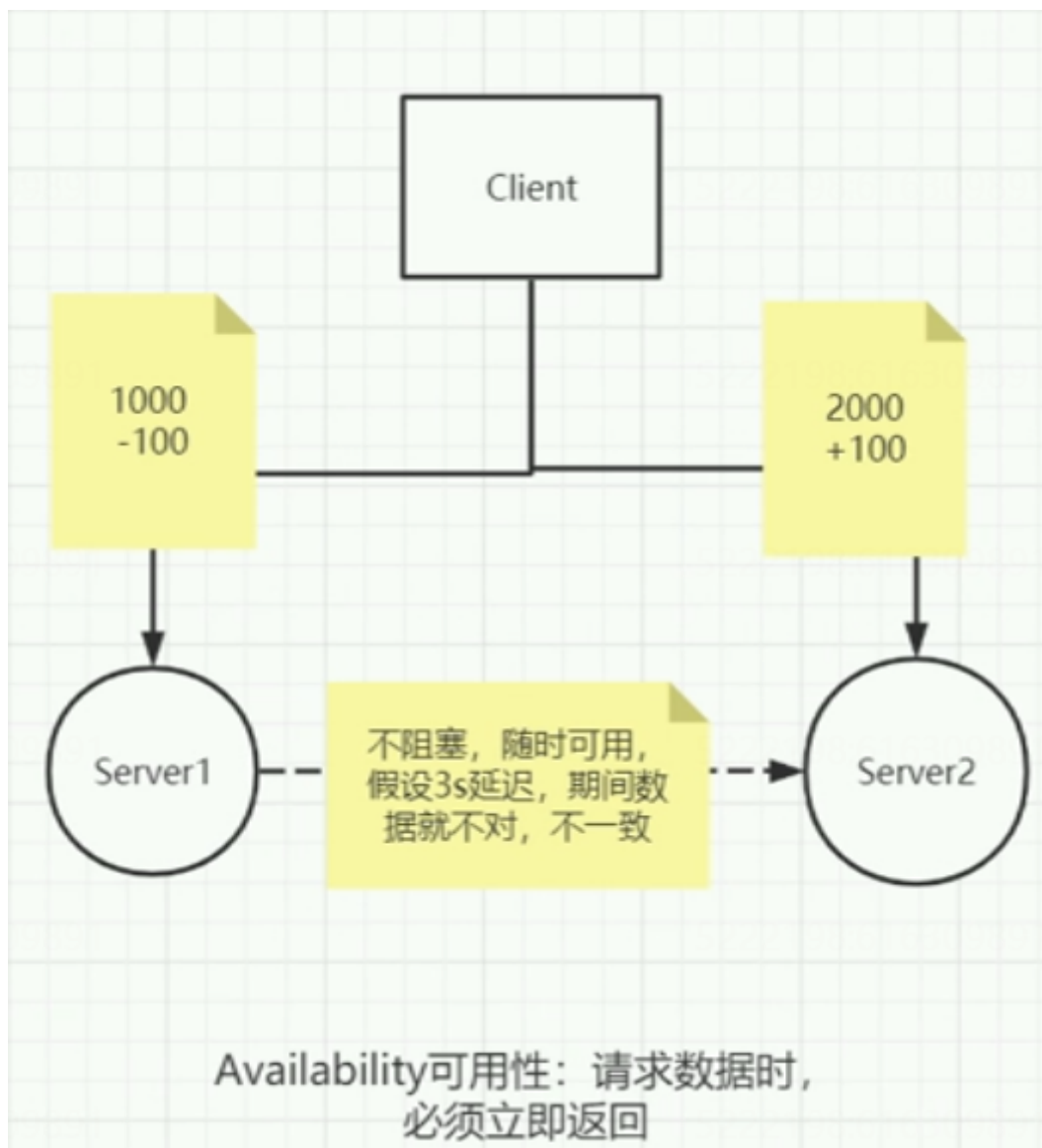
- Consistency：一致性
- Availability：可用性
- PartitionTolerance：分区容错



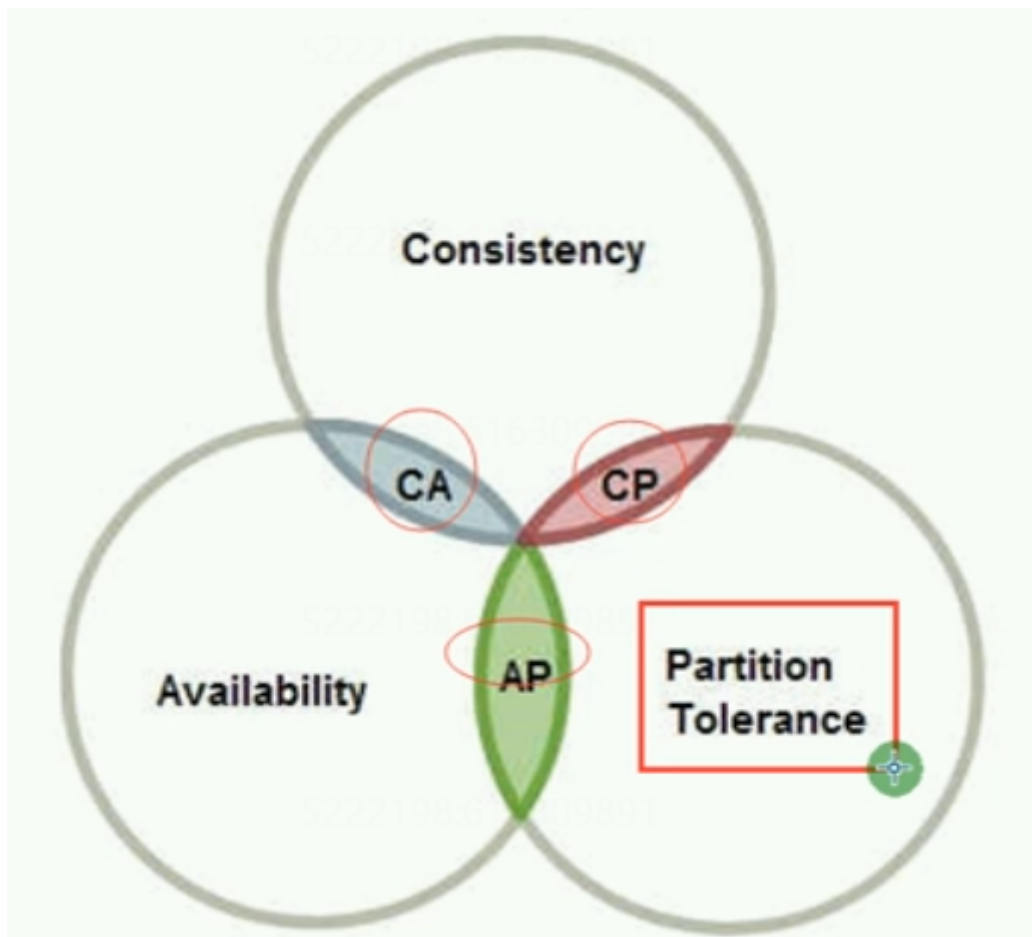
网络问题：所以会有分区容错，因为数据传输可能丢包/断网/故障，这是分布式下一定存在的。



银行转账：不能接受一个不正确的结果（距离相隔太远，需要有3秒延迟）如果阻塞，则不可用。



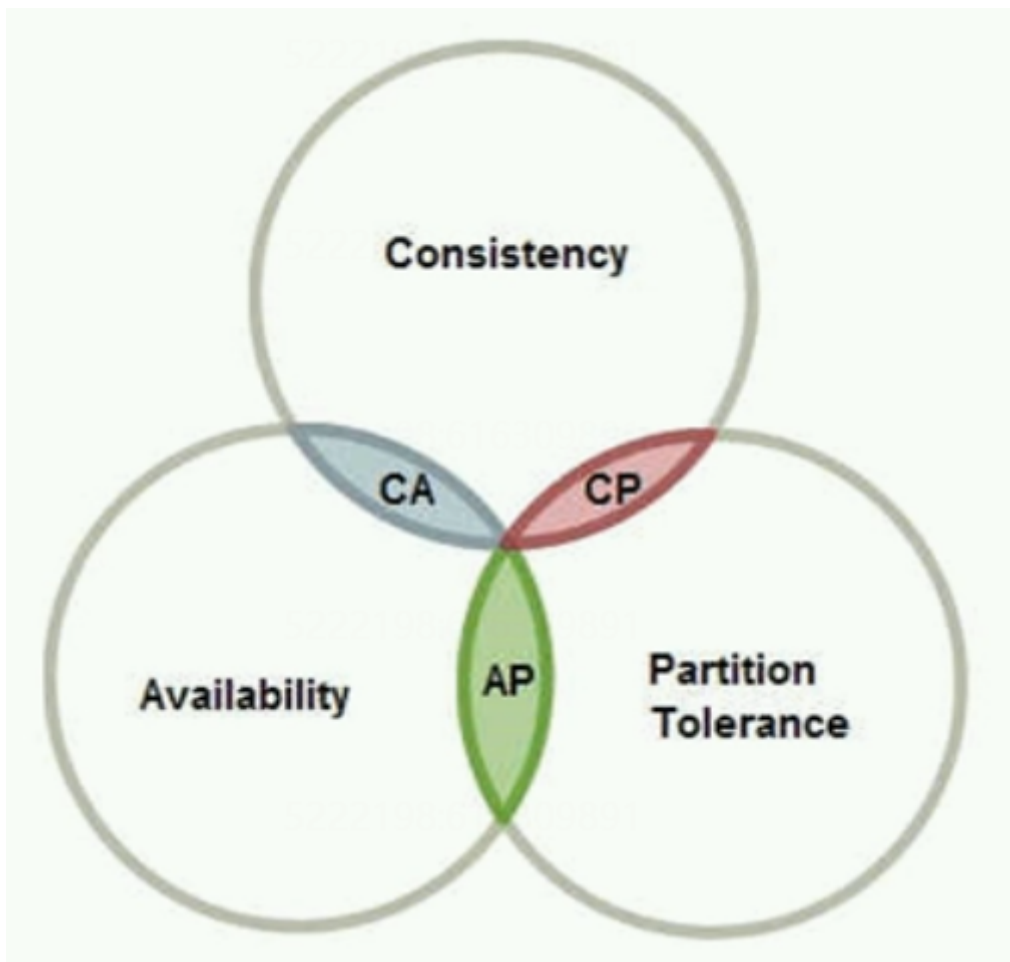
如果不强制阻塞，就会出现数据不一致的问题。



三者不能共存，但两者可以。CA、CP、AP

## 注意：CA是非分布式的，单体架构。

Consistency 和 Availability怎么选



## 前提是分布式系统---为了高并发--- 一致性和可用性，不能同时满足，要什么？

CP重要，一致性最重要了，数据不能错

银行-交易数据

AP重要，可用性最重要了，系统的可用性，

尤其是分布式----微服务，可用性尤为重要，

没有可用性是跑不起来

## 多种一致性

强一致性-任意时刻数据都是一致的 2PC 3PC

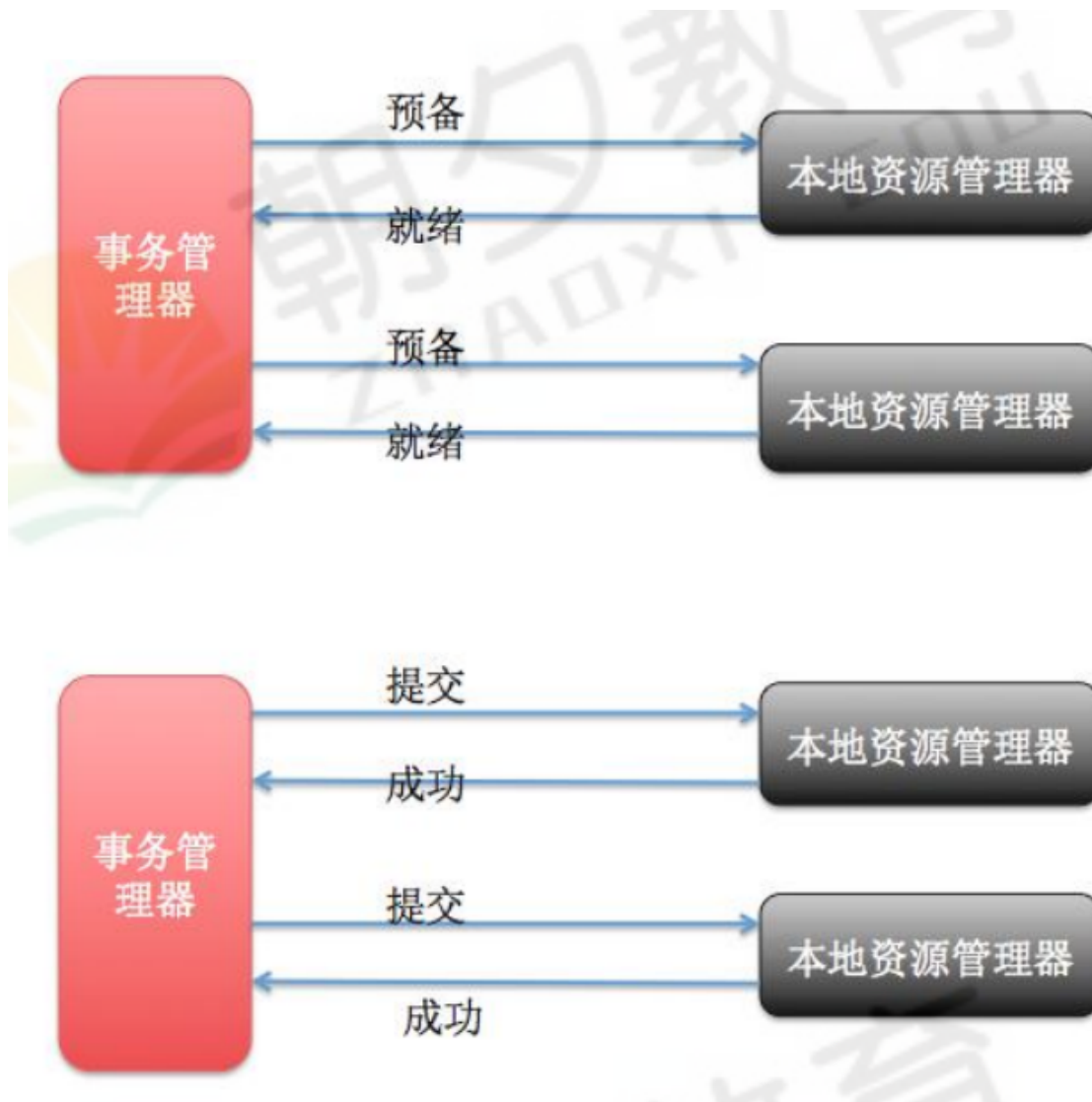
弱一致性---允许某一时刻不一致，承诺在一定时间内变成一致的

Try-Confirm-Cancel 代码层面--Saga

最终一致性-允许数据不一致，但是最终最终，数据还是得一致的 业务层面

---

## 2PC和 3PC:



## 2pc/3pc都只是一个协议，对过程管控，不关注实现

2PC/二阶段提交：

简单举例，如SQLServer、PGSql，Oracle都可以做到自己的2PC，且是ACID原子性的，但如果我们有2个以上数据库做同步怎么办？

#### ## 1. 阻塞问题

二阶段提交的第一阶段中，协调者需要等待参与者的响应，如果没有接收到任意参与者的响应，这时候进入等待状态，而其他正常发送响应的参与者，将进入阻塞状态，将无法进行其他任何操作，只有等待超时中断事务，极大的限制了系统的性能。

#### ## 2. 单点问题

协调者处于一个中心的位置，一旦出现问题，那么整个二阶段提交将无法运转，更为严重的是，如果协调者在阶段二中出现问题的话，那么其他参与者将会一直处于锁定事务资源的状态中，将无法继续完成操作

以上提到的2个问题都在3PC中得到了解决

1. 解决阻塞问题：将2PC中的第一阶段一分为二，提供了一个CanCommit阶段，此阶段并不锁定资源，这样可以大幅降低了阻塞概率

2. 解决单点问题：在参与者这边也引入了超时机制

3PC/三段式提交：

3PC是2PC的改进版本，将2PC的第一阶段：提交事务阶段一分为二，形成了CanCommit、PreCommit和doCommit三个阶段组成的事务处理协议

三段提交的心理理念是：在询问的时候并不锁定资源，除非所有人都同意了，才开始锁资源。

3PC具体的流程步骤就不在描述了，在二阶段的基础上加了事务询问的过程(CanCommit)

## 3PC虽然解决了2PC存在的2个问题，但是不管是2PC还是3PC都存在数据一致性的问题：

## 2PC：比如协调者在只给部分参与者发送了Commit请求，那就会出现部分参与者执行了Commit，部分没有提交，出现不一致问题。

## 3PC：一旦参与者无法及时收到来自协调者的信息之后，他会默认执行commit。而不会一直持有事务资源并处于阻塞状态，但是这种机制也会导致数据一致性问题。

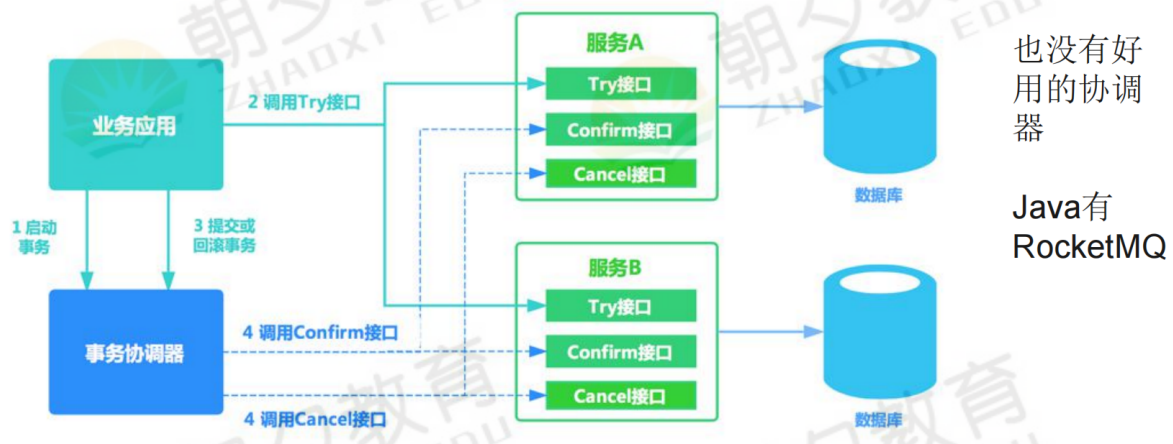
## 总结

2PC/3PC用来处理分布式事务：能够很好的提供强一致性和强事务性，但相对来说延迟比较高，比较适合传统的单体应用，在同一个方法中存在跨库操作的情况，不适合高并发和高性能要求的场景。

## 弱一致性：

## 在短时间内可以恢复一致（暂时的不一致，换来可用性）

## TCC (Try-Confirm-Cancel) ---弱一致性



### • TCC (Try-Confirm-Cancel)

会有数据库A成功，数据库B失败的可能性，且同时无法Cancel的情况发生（邮件等）

非阻塞——可以短时间内保持一致

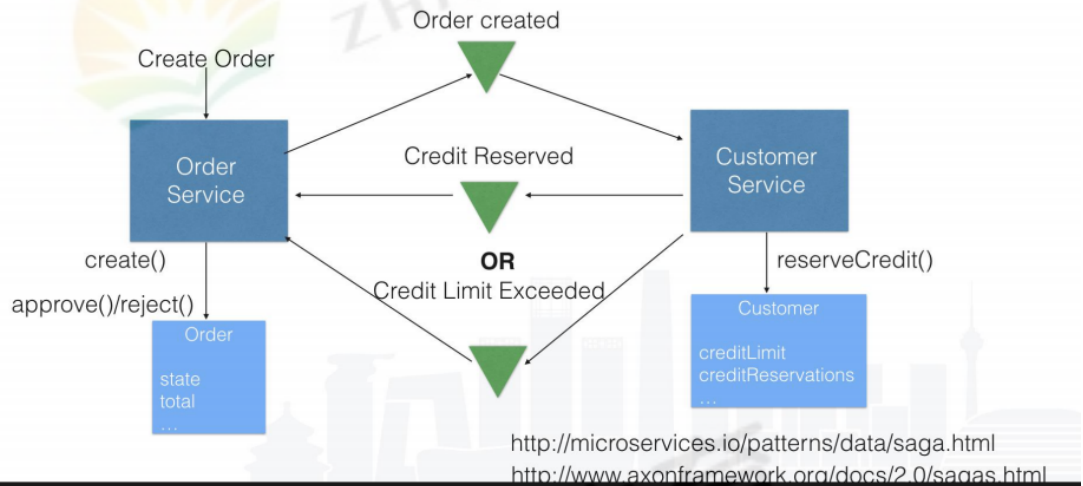
### • 幂等性

对同一个系统，使用同样的条件，一次请求和重复的多次请求对系统资源的影响是一致的。(网页提交按钮，点1次，跟10次结果一样)

场景：支付—页面修改信息---订单减库存/多次尝试不能加多次钱

### • SaGa

# 分布式Saga实现



Do和Undo，每一步都有补偿，比较极端的一种方式。

GitHub连接地址：<https://github.com/OpenSagas-csharp/servicecomb-pack-csharp>

## 最终一致性：

## 在分布式的前提下，分区容错是一定存在的，可用性是最重要的思想，但一致性也不能丢。

---眼中容沙---

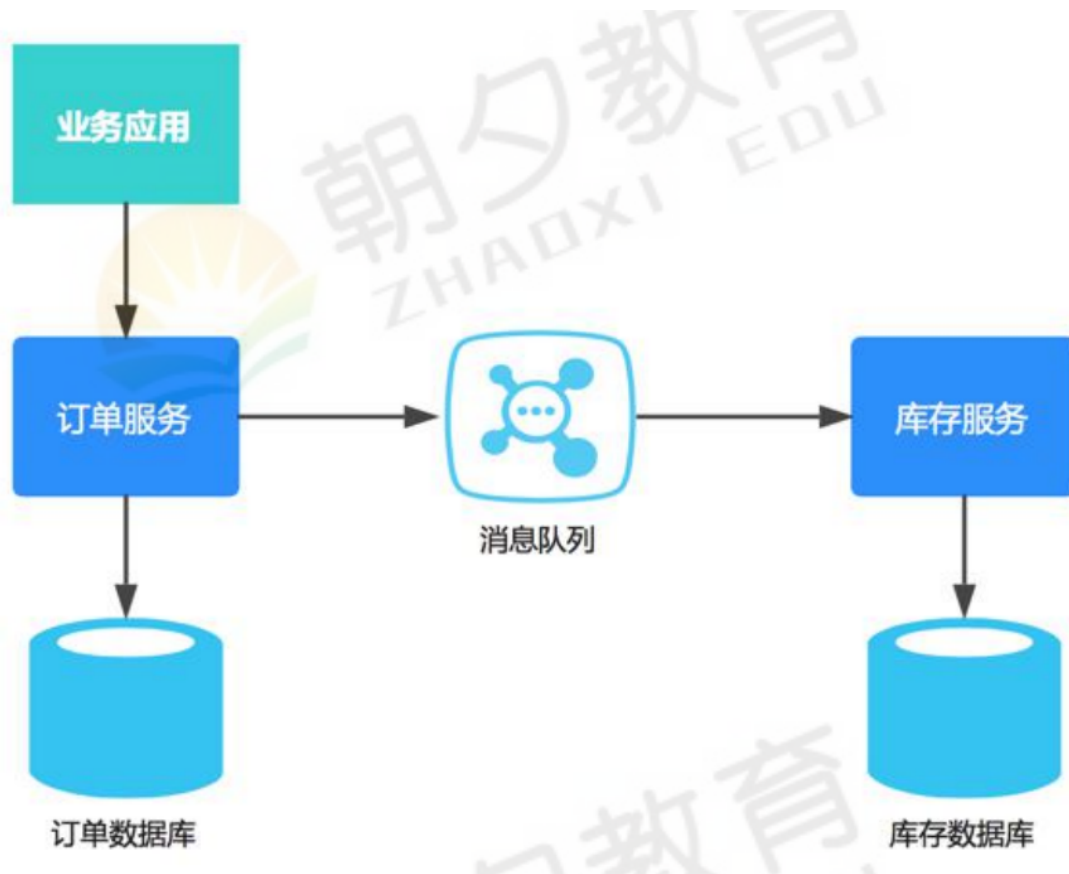
是否会有订单超卖的问题？

Base理论：

1. Basically Available(基本可用)
2. (最终一致性)
3. Soft state (软状态)
4. Eventually consistent

本地消息表分布式事务





解决了一个问题的同时，往往会带来新的问题。这时候需要慢慢的补齐问题节点即可。

上面的方式是TCC / 3PC不能接受的

想要达成上图的方式，需要保证几点：

- 上游投递消息稳定性，一定要能放入队列
- 下游获取消息，一定要拿得到，消费到
- 消息队列稳定，不会丢失

达成上面几点，即可完成本地消息表分布式事务的操作。

**难点：怎么保证RabbitMQ和数据库同时成功？**

目前没有工具可以搞定这部分，所以需要设计搞定。

1. 业务操作+本地Publish表
  2. 使用本地事务存储
  3. 读取Publish，发送MQ，更新状态，但 也没办法保证MQ发送成功，数据库一定更新状态成功
  4. 如果失败，继续重试，万一错了，无非就是写入MQ，但没有更新数据库状态
- 重放攻击，重复数据，幂等性

读取操作

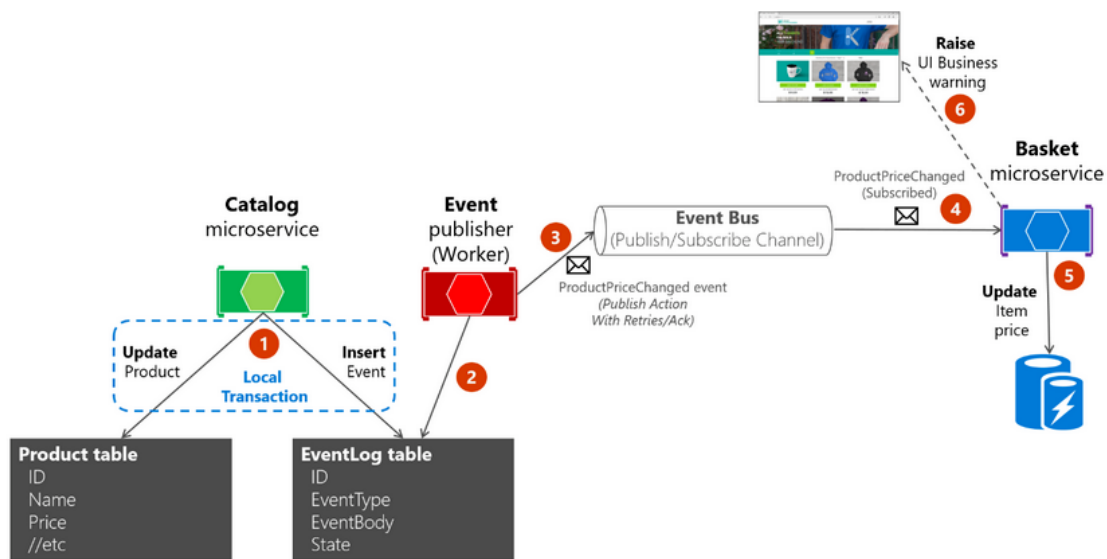
1. 全靠RabbitMQ的ACK？
2. 读取后写入Receive表，如果写入成功，回去ACK，如果写入失败，不ACK
3. 读本地数据库表，事务操作，保证一致性
4. 下一步，本地Publish表，然后MQ

**CAP落地：**

使用NCC CAP框架

## github地址: <https://github.com/dotnetcore/CAP>  
## 官网地址: <https://cap.dotnetcore.xyz/>  
## 官网文档: <https://cap.dotnetcore.xyz/user-guide/zh/cap/idempotence/>

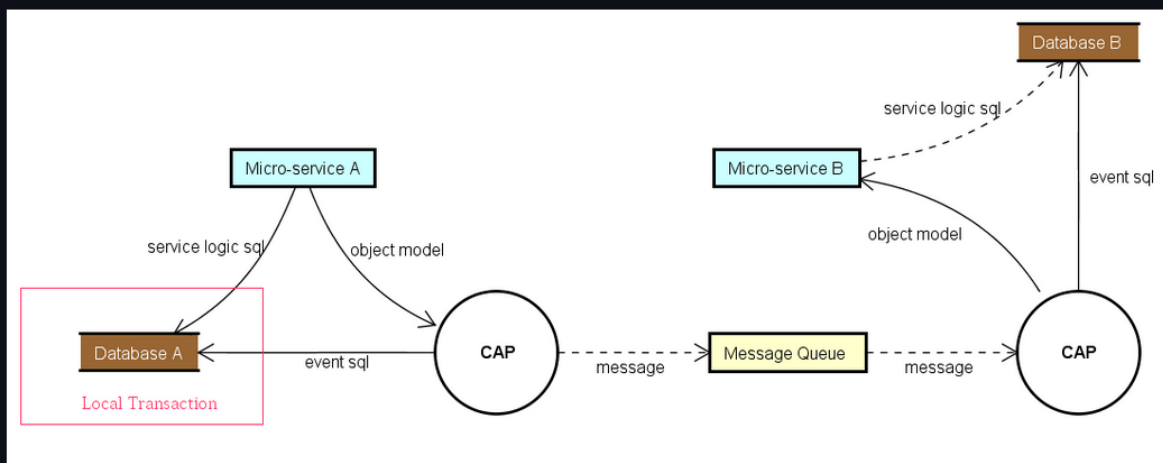
### CAP implements the Outbox Pattern described in the [eShop ebook](#)



Atomicity when publishing events to the event bus with a worker microservice

[eShop ebook](#) 已经被其他开源项目使用

### Architecture overview



CAP implements the Outbox Pattern described in the [eShop ebook](#).

这个就是它的一个基础架构图，用来参考查看

### 存储:

- 1 SQL Server
- 2 MySQL
- 3 PostgreSQL
- 4 MongoDB
- 5 In-Memory


没有Oracal, 但.....Net用的实在不多。可以自己写

咱们的目标，使用SqlServer和MongoDB来实现跨库的一个事务

传输：

- [RabbitMQ](#)
- [Kafka](#)
- [Azure Service Bus](#)
- [Amazon SQS](#)
- [NATS](#)
- [In-Memory Queue](#)
- [Redis Streams](#)
- [Apache Pulsar](#)

怎么选择运输器

	RabbitMQ	Kafka	Azure Service Bus	In-Memory
定位	可靠消息传输	实时数据处理	云	内存型，测试
分布式	✓	✓	✓	✗
持久化	✓	✓	✓	✗
性能	Medium	High	Medium	High

数据库脚本初始化：

见文件，仅供查询做基础数据支撑。。。

RabbitMQ集群：

```
docker run -d --hostname rabbit1 --name myrabbit1 -p 15672:15672 -p 5672:5672 -e RABBITMQ_ERLANG_COOKIE='rabbitcookie' rabbitmq:3.6.15-management
```

```
docker run -d --hostname rabbit2 --name myrabbit2 -p 5673:5672 --link myrabbit1:rabbit1 -e RABBITMQ_ERLANG_COOKIE='rabbitcookie' rabbitmq:3.6.15-management
```

```
docker run -d --hostname rabbit3 --name myrabbit3 -p 5674:5672 --link myrabbit1:rabbit1 --link myrabbit2:rabbit2 -e RABBITMQ_ERLANG_COOKIE='rabbitcookie' rabbitmq:3.6.15-management
```

```
docker exec -it myrabbit1 bash
rabbitmqctl stop_app
rabbitmqctl reset
rabbitmqctl start_app
exit

docker exec -it myrabbit2 bash
```

```

rabbitmqctl stop_app
rabbitmqctl reset
rabbitmqctl join_cluster --ram rabbit@rabbit1
rabbitmqctl start_app
exit

docker exec -it myrabbit3 bash
rabbitmqctl stop_app
rabbitmqctl reset
rabbitmqctl join_cluster --ram rabbit@rabbit1
rabbitmqctl start_app
exit

```

完成后访问: <http://192.168.72.164:15672>

## MongoDB集群:

```

1 拉取mongoDB镜像
docker pull mongo

2 创建本地挂在目录--建议先删除 可选挂载/不挂载
mkdir -p /Demo/mongo1 #创建挂载的db目录
mkdir -p /Demo/mongo2 #创建挂载的db目录
mkdir -p /Demo/mongo3 #创建挂载的db目录

3 启动多个MongoDB实例
#第一台:
docker run --name mongo1 -p 27017:27017 -d mongo mongod --replSet "rs0"
#第二台:
docker run --name mongo2 -p 27018:27017 -d mongo mongod --replSet "rs0"
#第三台:
docker run --name mongo3 -p 27019:27017 -d mongo mongod --replSet "rs0"

4 搭建集群
#进入mongo1容器
docker exec -ti mongo1 /bin/bash
#连接mongodb
mongo
#初始化副本集
rs.initiate({"_id": "rs0", "members": [{"_id":0, "host":"192.168.72.164:27017"}, {"_id":1, "host":"192.168.72.164:27018","arbiterOnly":true}, {"_id":2, "host":"192.168.72.164:27019"}]})
# 加这个字段,说明该节点就是仲裁不存放数据
"arbiterOnly":true
#查看副本集配置信息
rs.conf()
rs.status()
#从节点开启读数据模式

db.getMongo().setSlaveOk();
exit

5 查询使用和验证
docker exec -it mongo1 bash
mongo
use test
db.test.insert({msg: 'this is from primary', ts: new Date()})

docker exec -ti mongo3 /bin/bash
mongo

```

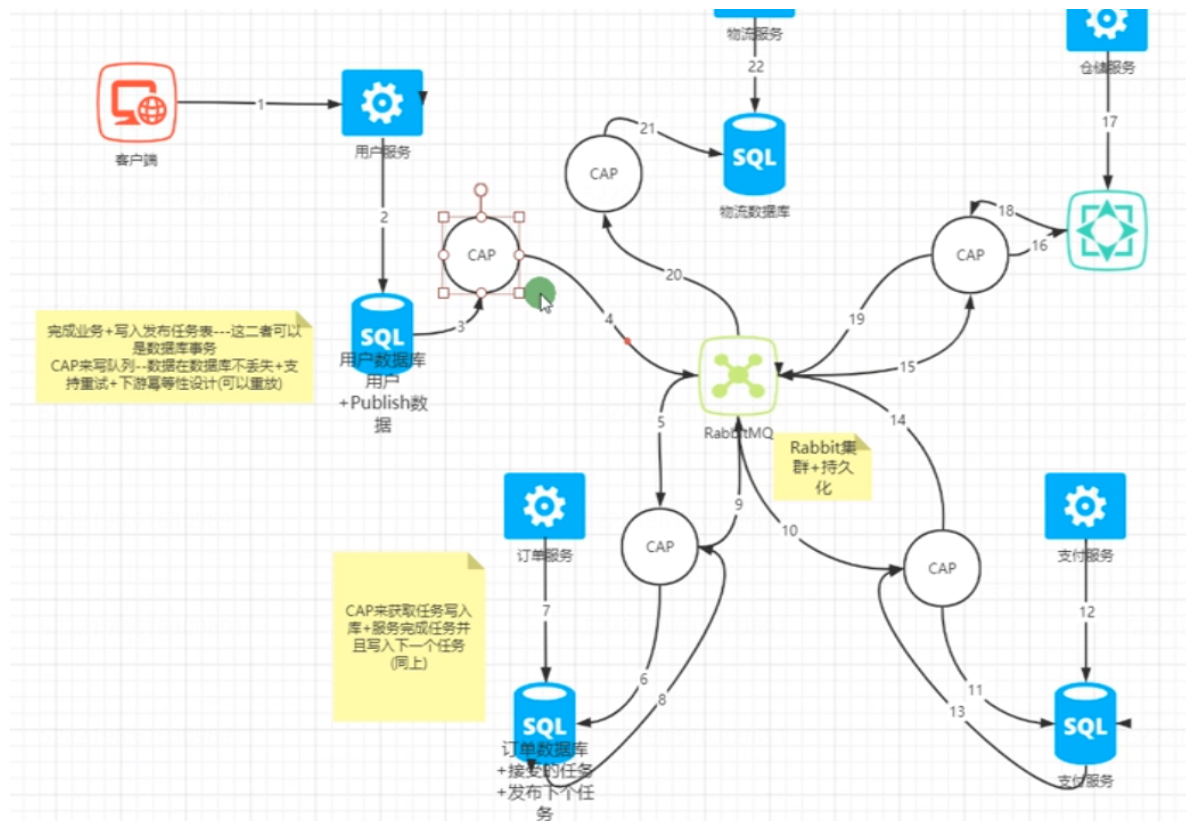
```
use test
db.test.find()
## 以前版本
db.getMongo().setSlaveOk();
## 当下版本
db.getMongo().setSecondaryOk();
```

使用工具 NoSQL Manager for MongoDB Freeware  
连接: 192.168.72.164:27017

## Consul配置:

```
consul.exe agent -dev
```

## 架构和引用包:



以该图为基础流程

配置代码：

- 纳入依赖
- 配置地址
- 配置CAP连接

```
## UserService
dotnet run --urls=http://*:11111

## 请求
http://localhost:11111/without/transaction

## OrderService
dotnet run --urls=http://*:22222
```

```
## PaymentService
dotnet run --urls=http://*:33333
```

```
## 五连贯
## UserService
dotnet run --urls=http://*:11111
## OrderService
dotnet run --urls=http://*:11112
## StorageService
dotnet run --urls=http://*:11113
## LogisticsService
dotnet run --urls=http://*:11114
## PaymentService
dotnet run --urls=http://*:11115
```

- 成功数据1天删除
- 失败数据15天——可修改Retries