

第9章 分布式事务解决方案


学习目标：

- 完成结算页商品清单的功能
- 完成收货地址选择的功能
- 完成提交订单的功能
- 能够说出CAP定理、BASE理论以及常见的分布式事务解决方案
- 完成库存扣减的分布式事务处理

1. 结算页商品清单

1.1 需求分析

在订单结算页显示送货清单、合计金额、优惠金额。

商品清单：				
	Apple iPhone 6s (A1700) 64G 玫瑰金色 移动联通电信4G手机硅胶透明防摔软壳 本色系列	¥ 5399.00	X1	有货
	7天无理由退货			
	Apple iPhone 6s (A1700) 64G 玫瑰金色 移动联通电信4G手机硅胶透明防摔软壳 本色系列	¥ 5399.00	X1	有货
	7天无理由退货			

显示购物清单应该刷新购物车，从数据库中查询最新价格计算。

下方显示合计件数与合计金额。

1.2 实现思路

后端新增方法，获取购物车列表循环更新每个商品的价格，刷新数据后更新购物车，并返回更新后的购物车数据。

前端获取列表后循环显示，显示合计件数与合计金额参考购物车页面的实现方式。

1.3 代码实现

1.3.1 后端代码

(1) CartService新增方法定义

```
/**
 * 获取最新的购物车列表
 * @param username 用户名
 * @return
 */
public List<Map<String, Object>> findNewOrderItemList(String username);
```

(2) CartServiceImpl实现该方法

```
@Override
public List<Map<String, Object>> findNewOrderItemList(String username) {
    //获取选中的购物车
    List<Map<String, Object>> cartList = findCartList(username);
    //循环购物车列表，重新读取每个商品的最新价格
    for(Map<String, Object> cart:cartList){
        OrderItem orderItem=(OrderItem)cart.get("item");
        Sku sku = skuService.findById(orderItem.getSkuId());
        orderItem.setPrice(sku.getPrice());//更新价格
        orderItem.setMoney(sku.getPrice()*orderItem.getNum());//更新金额
    }

    redisTemplate.boundHashOps(CacheKey.CART_LIST).put(username, cartList);//
    存入缓存
    return cartList;
}
```

(3) CartController新增方法

```
/**
 * 获取刷新单价后的购物车列表
 * @return
 */
@GetMapping("/findNewOrderItemList")
public List<Map<String, Object>> findNewOrderItemList(){
    String username =
SecurityContextHolder.getContext().getAuthentication().getName();
    return cartService.findNewOrderItemList(username);
}
```

1.3.2 前端代码

- (1) 将order.html拷贝至qingcheng_web_portal
- (2) 修改order.html，为div添加id

```
<div id="app">
```

添加js代码

```

<script src="js/axios.js"></script>
<script src="js/vue.js"></script>
<script>
  new Vue({
    el: '#app',
    data(){
      return {
        cartList:[],
        totalNum:0,
        totalMoney:0,
        payMoney:0,//应付金额
        preferential:0//优惠金额
      }
    },
    created(){
      this.findCartList();
    },
    methods:{
      findCartList(){
        axios.get(`/cart/findNewOrderItemList.do`).then(
response=>{
          this.cartList= response.data;
          this.count();
        });
      },
      count(){
        this.totalNum=0;
        this.totalMoney=0;
        for(let i=0;i<this.cartList.length;i++){
          if(this.cartList[i].checked){
            this.totalNum+= this.cartList[i].item.num;
            this.totalMoney+= this.cartList[i].item.money;
          }
        }

        axios.get(`/cart/preferential.do`).then( response=>{
          this.preferential= response.data.preferential; //优惠
金额

          this.payMoney =this.totalMoney- this.preferential; //

```

优惠后金额

```
        })  
      }  
    }  
  });  
</script>
```

(3) 修改页面中购物车列表部分

```
<span>商品清单: </span>  
<ul class="yui3-g" v-for="cart in cartList" v-if="cart.checked==true">  
  <li class="yui3-u-1-6">  
    <span>  
</span>  
  </li>  
  <li class="yui3-u-7-12">  
    <div class="desc">{{cart.item.name}}</div>  
    <div class="seven">7天无理由退货</div>  
  </li>  
  <li class="yui3-u-1-12">  
    <div class="price">¥{{(cart.item.price/100).toFixed(2)}}</div>  
  </li>  
  <li class="yui3-u-1-12">  
    <div class="num">X{{cart.item.num}}</div>  
  </li>  
  <li class="yui3-u-1-12">  
    <div class="exit">有货</div>  
  </li>  
</ul>
```

修改页面合计数部分

```

<div class="order-summary">
  <div class="static fr">
    <div class="list">
      <span><i class="number">{{totalNum}}</i>件商品，总商品金额
</span>
      <em class="allprice">¥{{(totalMoney/100).toFixed(2)}}</em>
    </div>
    <div class="list">
      <span>返现: </span>
      <em class="money">{{(preferential/100).toFixed(2)}}</em>
    </div>
    <div class="list">
      <span>运费: </span>
      <em class="transport">0.00</em>
    </div>
  </div>
</div>
<div class="clearfix trade">
  <div class="fc-price">应付金额: <span class="price">¥
{{(payMoney/100).toFixed(2)}}</span></div>
  <div class="fc-receiverInfo">
    寄送至:
    <span id="receive-address">北京市海淀区三环内中关村软件园9号楼
</span>
    收货人: <span id="receive-name">李西西</span>
    <span id="receive-phone">15922333201</span>
  </div>
</div>

```

2. 收货地址选择

2.1 需求分析

在结算页上列出收货地址

收件人信息

李煜	北京市海淀区三环内中关村软件园9号楼 15932223201	默认地址
李西西	北京市昌平区建材城西路金燕龙办公楼 18545692564	
王希	河北省衡水市大庆西路888号 18758946254	

点击收货人姓名后，即选中该收货人，在结算页下方显示地址、收货人姓名和电话

应付金额: **¥5399.00**

寄送至: 北京市海淀区三环内中关村软件园9号楼 收货人: 李西西 15922333201

2.2 实现思路

首先我们先看一下user数据库的收货地址表（tb_address 表）表结构

字段名称	字段含义	字段类型	字段长度	备注
id	id	INT		
username	用户名	VARCHAR		
provinceid	省	VARCHAR		
cityid	市	VARCHAR		
areaid	县/区	VARCHAR		
phone	电话	VARCHAR		
address	详细地址	VARCHAR		
contact	联系人	VARCHAR		
is_default	是否是默认 1默认 0否	VARCHAR		
alias	别名	VARCHAR		

后端实现根据当前登陆人查询收货地址列表的方法，前端使用Vue.js进行渲染。

选择收货地址通过vue.js实现。

2.3 代码实现

2.3.1 收货地址列表

（1）AddressService新增方法定义

```

/**
 * 根据用户名查询地址列表
 * @param username 用户名
 * @return 地址列表
 */
public List<Address> findByUsername(String username);

```

(2) AddressServiceImpl实现方法

```

/**
 * 根据用户名查询地址列表
 * @param username 用户名
 * @return
 */
public List<Address> findByUsername(String username) {
    Example example=new Example(Address.class);
    Example.Criteria criteria = example.createCriteria();
    criteria.andEqualTo("username",username);
    return addressMapper.selectByExample(example);
}

```

(3) CartController新增方法

```

@Reference
private AddressService addressService;

/**
 * 根据用户名查询地址列表
 * @return
 */
@RequestMapping("/findAddressList")
public List<Address> findAddressList(){
    String username =
SecurityContextHolder.getContext().getAuthentication().getName();
    return addressService.findByUsername(username);
}

```

(4) 修改 cart.html ， 增加属性， 用于存储地址列表

```
addressList:[]
```


(5) order.html新增方法，用于查询地址列表

```
findAddressList(){ //查询地址列表
    axios.get(`/cart/findAddressList.do`).then(response => {
        this.addressList=response.data;
    });
},
```

(6) 修改order.html的地址列表部分

```
<div class="choose-address" v-for="address in addressList">
    <div class="con name selected" ><a>{{address.contact}}<span title="点
击取消选择"></span></a></div>
    <div class="con address">
        <span class="place">{{address.address}}</span>
        <span class="phone">{{address.phone}}</span>
        <span class="base" v-if="address.isDefault=='1'">默认地址</span>
    </div>
    <div class="clearfix"></div>
</div>
```

2.3.2 选择收货地址

(1) order.html 新增属性，用于保存订单（订单包含了地址、手机号、联系人等信息）

```
order:{receiverAddress:'',receiverMobile:'',receiverContact:''} //当前订单
```

(2) order.html新增方法，用于选择地址

```
selectAddress(address){ //选择地址
    this.order.receiverAddress= address.address; // 地址
    this.order.receiverMobile=address.phone; //手机号
    this.order.receiverContact=address.contact; // 联系人
}
```

(3) 修改order.html地址显示区域

```
<div class="fc-receiverInfo">
  寄送至:
  <span id="receive-address">{{order.receiverAddress}}</span>
  收货人: <span id="receive-name">{{order.receiverContact}}</span>
  <span id="receive-phone">{{order.receiverMobile}}</span>
</div>
```

(4) 修改order.html的findAddressList方法，在回调后添加以下代码，实现默认地址的显示

```
for(let i=0;i<this.addressList.length;i++){
  if(this.addressList[i].isDefault=='1'){
    this.order.receiverAddress= this.addressList[i].address;// 地址
    this.order.receiverMobile= this.addressList[i].phone;//手机号
    this.order.receiverContact= this.addressList[i].contact;//联系人
  }
}
```

(5) 修改地址列表，增加样式和点击事件的调用

```
<div :class="order.receiverAddress==address.address?'con name
selected':'con name'" @click="selectAddress(address)">
```

3. 提交订单

3.1 需求分析

选择支付方式

支付方式

在线支付

货到付款

点击“提交订单”按钮，完成订单保存动作。

1件商品, 总商品金额 ¥5399.00
返现: 0.00
运费: 0.00

应付金额: **¥5399.00**

寄送至: 北京市海淀区三环内中关村软件园9号楼 收货人: 李西西 15922333201

提交订单

将购物车中选中项保存为订单。如果是在线支付则跳转到支付页面，如果是货到付款则直接跳转到完成页。

保存订单前需要进行库存进行检查和扣减，如果库存不足则不能下单。

注意：在提交保存订单时，需要再次调用刷新购物车的方法。

3.2 实现思路

首先我们先看一下相关的表结构

tb_sku 表

字段名称	字段含义	字段类型	字段长度	备注
id	商品 id	VARCHAR		
sn	商品条码	VARCHAR		
name	SKU名称	VARCHAR		
price	价格（分）	INT		
num	库存数量	INT		
alert_num	库存预警数量	INT		
image	商品图片	VARCHAR		
images	商品图片列表	VARCHAR		
weight	重量（克）	INT		
create_time	创建时间	DATETIME		
update_time	更新时间	DATETIME		
spu_id	SPUID	VARCHAR		
category_id	类目ID	INT		
category_name	类目名称	VARCHAR		
brand_name	品牌名称	VARCHAR		
spec	规格	VARCHAR		
sale_num	销量	INT		
comment_num	评论数	INT		
status	商品状态 1 -正常， 2 -下架， 3 -删除	CHAR		

tb_order 表（订单主表）

字段名称	字段含义	字段类型	字段长度	备注
id	订单id	VARCHAR		分布式id
total_num	数量合计	INT		
total_money	金额合计	INT		单位（分）
pre_money	优惠金额	INT		单位（分）
post_fee	邮费	INT		单位（分）
pay_money	实付金额	INT		单位（分）
pay_type	支付类型	VARCHAR		0、货到付款 1、微信支付， 2、支付宝 3、银联支付
create_time	订单创建时间	DATETIME		
update_time	订单更新时间	DATETIME		
pay_time	付款时间	DATETIME		
consign_time	发货时间	DATETIME		
	交易			

end_time	完成 时间	DATETIME		
close_time	交易 关闭 时间	DATETIME		
shipping_name	物流 名称	VARCHAR		
shipping_code	物流 单号	VARCHAR		
username	用户 名称	VARCHAR		
buyer_message	买家 留言	VARCHAR		
buyer_rate	是否 评价	CHAR		
receiver_contact	收货 人	VARCHAR		
receiver_mobile	收货 人手机	VARCHAR		
receiver_address	收货 人地址	VARCHAR		
source_type	订单 来源:	CHAR		1:web, 2: app, 3: 微信公众 号, 4: 微信小程序 5 H5手机 页面
transaction_id	交易 流水 号	VARCHAR		
	订单			0待付款、1待发货、2已发货、

order_status	状态	CHAR		3已完成、4已关闭
pay_status	支付状态	CHAR		0未支付、1已支付、2已退款
consign_status	发货状态	CHAR		0未发货、1已发货
is_delete	是否删除	CHAR		0：未删除 1：已删除

tb_order_item 表（订单明细表）

字段名称	字段含义	字段类型	字段长度	备注
id	ID	VARCHAR		分布式id
category_id1	1级分类	INT		
category_id2	2级分类	INT		
category_id3	3级分类	INT		
spu_id	SPU_ID	VARCHAR		
sku_id	SKU_ID	VARCHAR		
order_id	订单ID	VARCHAR		
name	商品名称	VARCHAR		
price	单价	INT		
num	数量	INT		
money	总金额	INT		
pay_money	实付金额	INT		
image	图片地址	VARCHAR		
weight	重量	INT		
post_fee	运费	INT		
is_return	是否退货	CHAR		0: 未退款 1: 已申请 2: 已退

主要实现思路:

- (1) 刷新并获取购物车
- (2) 检查库存与扣减库存, 增加销量

(3) 保存订单主表与明细表，明细表的数据来自购物车

(4) 清除选中的购物车数据

3.3 代码实现

3.3.1 库存扣减逻辑

(1) SkuMapper新增方法

```
/**
 * 扣减库存方法
 * @param id
 * @param num
 */
@Select("update tb_sku set num=num-#{num} where id=#{id}")
public void deductionStock(@Param("id") String id, @Param("num")
Integer num);

/**
 * 添加销量
 * @param id
 * @param num
 */
@Select("update tb_sku set saleNum=saleNum+#{num} where id=#{id}")
public void addSaleNum(@Param("id") String id, @Param("num") Integer
num );
```

(2) SkuService新增方法

```
/**
 * 根据购物车批量扣减库存
 * @param oderItemList
 */
public boolean deductionStock(List<OrderItem> oderItemList);
```

(2) SkuServiceImpl实现此方法

```

@Transactional
public boolean deductionStock(List<OrderItem> orderItemList) {

    //检查是否可以扣减库存
    boolean idDeduction=true;//是否可以扣减
    for( OrderItem orderItem:orderItemList){
        Sku sku = findById(orderItem.getSkuId());
        if(sku==null){
            idDeduction=false;
            break;
        }
        if(!"1".equals(sku.getStatus())){
            idDeduction=false;
            break;
        }
        if( sku.getNum().intValue()<orderItem.getNum().intValue() ){
            idDeduction=false;
            break;
        }
    }

    //执行扣减
    if(idDeduction){
        for(OrderItem orderItem:orderItemList){
            skuMapper.deductionStock(
orderItem.getSkuId(),orderItem.getNum());//扣减库存

            skuMapper.addSaleNum(orderItem.getSkuId(),orderItem.getNum());//增加销量
        }
    }
    return idDeduction;
}

```

3.3.2 保存订单逻辑

(1) 修改OrderService的add方法的定义，修改返回值，用于返回订单号和金额

```

public Map<String, Object> add(Order order);

```

(2) 修改OrderServiceImpl的add方法

```

@Autowired
private CartService cartService;

@Autowired
private IdWorker idWorker;

@Autowired
private OrderItemMapper orderItemMapper;

/**
 * 新增
 * @param order
 */
public Map<String, Object> add(Order order) {
    //获取购物车(刷新单价)
    List<Map<String, Object>> oderItemList =
cartService.findNewOrderItemList(order.getUsername());
    //获取选中的购物车
    List<OrderItem> orderItems = oderItemList.stream()
        .filter(cart -> (boolean) cart.get("checked"))
        .map(cart -> (OrderItem) cart.get("item"))
        .collect(Collectors.toList());
    //扣减库存
    if(!skuService.deductionStock(orderItems)){
        throw new RuntimeException("库存扣减失败");
    }
    //保存订单主表
    order.setId(idWorker.nextId()+"");
    //合计数计算
    IntStream numStream =
orderItems.stream().mapToInt(OrderItem::getNum);
    IntStream moneyStream =
orderItems.stream().mapToInt(OrderItem::getMoney);
    int totalNum=numStream.sum();//总数量
    int totalMoney=moneyStream.sum();//订单总金额
    int preMoney = cartService.preferential(order.getUsername()); //
计算优惠金额

    order.setTotalNum(totalNum);//总数量

    order.setTotalMoney(totalMoney);//总金额

```

```

order.setPreMoney(preMoney);//优惠金额
order.setPayMoney(totalMoney-preMoney);//支付金额=总金额+优惠金额
order.setCreateTime(new Date());//订单创建日期
order.setOrderStatus("0"); // 订单状态
order.setPayStatus("0"); // 支付状态：未支付
order.setConsignStatus("0"); //发货状态：未发货
orderMapper.insert(order);

double proportion = (double)order.getPayMoney()/totalMoney;
//保存订单明细
for(OrderItem orderItem:orderItems){
    orderItem.setOrderId(order.getId());//订单主表ID
    orderItem.setId(idWorker.nextId()+"");
    orderItem.setPayMoney( (int)(orderItem.getMoney()*
proportion) );//支付金额
    orderItemMapper.insert(orderItem);
}
//清除选中的购物车
cartService.deleteCheckedCart(order.getUsername());
//返回订单号和支付的金额
Map<String, Object> map=new HashMap();
map.put("ordersn", order.getId() );
map.put("money",order.getPayMoney());
return map;
}

```

订单服务添加 applicationContext-service.xml

```

<!--雪花分布式id生成-->
<bean id="idWorker" class="com.qingcheng.util.IdWorker">
    <constructor-arg index="0" value="1"></constructor-arg>
    <constructor-arg index="1" value="1"></constructor-arg>
</bean>

```

(2) CartController新增方法

```

@Reference
private OrderService orderService;

/**
 * 保存订单
 * @param order
 * @return
 */
@PostMapping("/saveOrder")
public Map<String, Object> saveOrder(@RequestBody Order order ){
    String
    username=SecurityContextHolder.getContext().getAuthentication().getName()
;
    order.setUsername(username);
    return orderService.add(order);
}

```

3.3.3 前端代码

(1) 修改cart.html的order属性，增加payType并设置默认值为1

```

order:
{receiverAddress:'',receiverMobile:'',receiverContact:'',payType:'1'}

```

(2) 修改cart.html的支付方式选择部分

```

<li class="selected" @click="order.payType='1'">在线支付<span title="点击取消选择"></span></li>
<li @click="order.payType='0'">货到付款<span title="点击取消选择"></span>
</li>

```

(3) 对留言文本框进行绑定

```

<textarea placeholder="建议留言前先与商家沟通确认" class="remarks-cont" v-
model="order.buyerMessage"></textarea>

```

(4) 新增方法

```

saveOrder(){ //保存订单
    axios.post(`/cart/saveOrder.do`,this.order).then(response => {
        if(response.data.ordersn!=null){//如果有订单号
            location.href=`pay.html?
ordersn=${response.data.ordersn}&money=${response.data.money}`;
        }
    });
}

```

(5) 调用方法

```

<a class="sui-btn btn-danger btn-xlarge" @click="saveOrder()" >提交订
单</a>

```

3.3.4 结果页面

从静态原型中拷贝三个页面到web_portal 分别是pay.html 、 order-success.html 、 order-fail.html

pay.html 和order-success.html 添加vue.js代码

```

<script src="/js/vue.js"></script>
<script src="/js/util.js"></script>
<script>
    new Vue({
        el:'#app',
        data(){
            return {
                ordersn:'',
                money:0
            }
        },
        created(){
            this.ordersn= getQueryString('ordersn');
            this.money= getQueryString('money');
        }
    })
</script>

```

绑定

```
<span class="f1">请您在提交订单<em class="orange time">4小时</em>之内完成支付, 超时订单会自动取消。订单号: <em>{{ordersn}}</em></span>
<span class="fr"><em class="sui-lead">应付金额: </em><em class="orange money">¥ {{(money/100).toFixed(2)}}</em></span>
```

4. 分布式事务解决方案

刚才我们编写的扣减库存与保存订单是在两个服务中存在的, 如果扣减库存后订单保存失败了是不会回滚的, 这样就会造成数据不一致的情况, 这其实就是我们所说的分布式事务的问题, 接下来我们来学习分布式事务的解决方案。

4.1 本地事务与分布式事务

4.1.1 事务

数据库事务(简称: 事务, Transaction)是指数据库执行过程中的一个逻辑单位, 由一个有限的数据库操作序列构成。

事务拥有以下四个特性, 习惯上被称为ACID特性:

原子性(Atomicity): 事务作为一个整体被执行, 包含在其中的对数据库的操作要么全部被执行, 要么都不执行。

一致性(Consistency): 事务应确保数据库的状态从一个一致状态转变为另一个一致状态。一致状态是指数据库中的数据应满足完整性约束。除此之外, 一致性还有另外一层语义, 就是事务的中间状态不能被观察到(这层语义也有说应该属于原子性)。

隔离性(Isolation): 多个事务并发执行时, 一个事务的执行不应影响其他事务的执行, 如同只有这一个操作在被数据库所执行一样。

持久性(Durability): 已被提交的事务对数据库的修改应该永久保存在数据库中。在事务结束时, 此操作将不可逆转。

4.1.2 本地事务

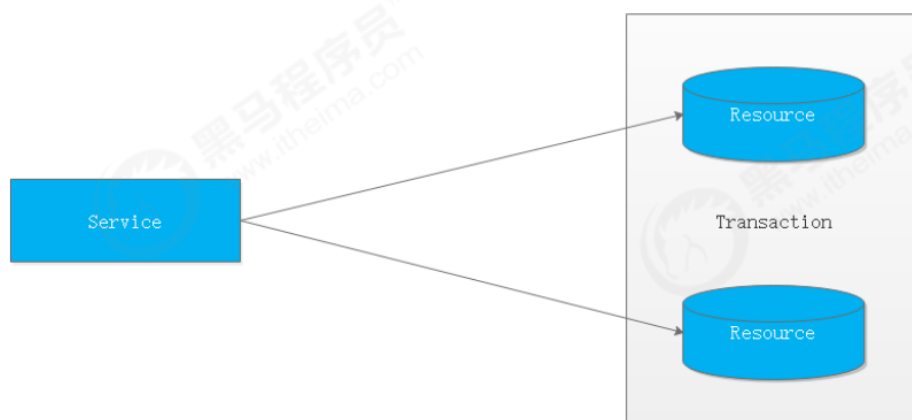
起初, 事务仅限于对单一数据库资源的访问控制, 架构服务化以后, 事务的概念延伸到了服务中。倘若将一个单一的服务操作作为一个事务, 那么整个服务操作只能涉及一个单一的数据库资源, 这类基于单个服务单一数据库资源访问的事务, 被称为本地事务(Local Transaction)。



4.1.3 分布式事务

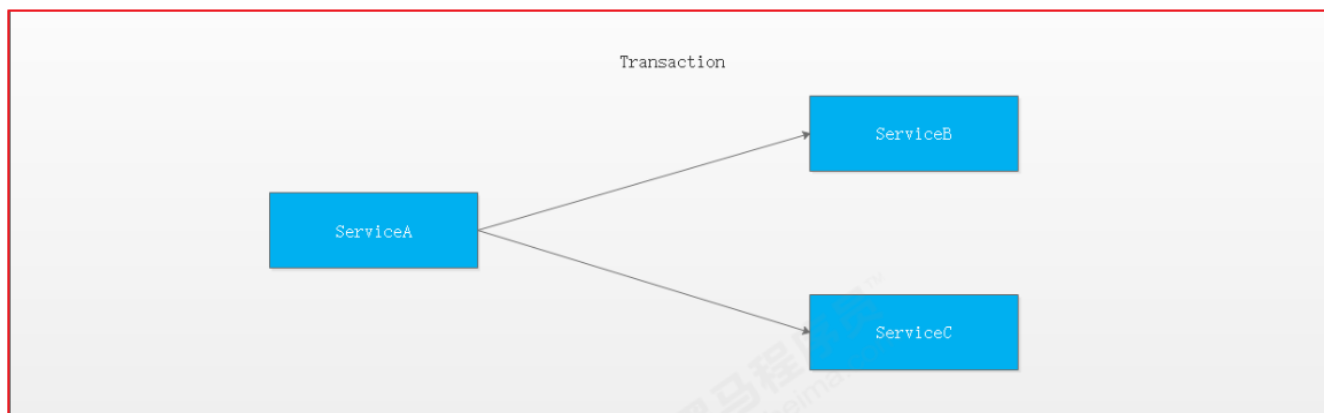
分布式事务指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上,且属于不同的应用, 分布式事务需要保证这些操作要么全部成功, 要么全部失败。本质上来说, 分布式事务就是为了保证不同数据库的数据一致性。

最早的分布式事务应用架构很简单, 不涉及服务间的访问调用, 仅仅是服务内操作涉及到对多个数据库资源的访问。

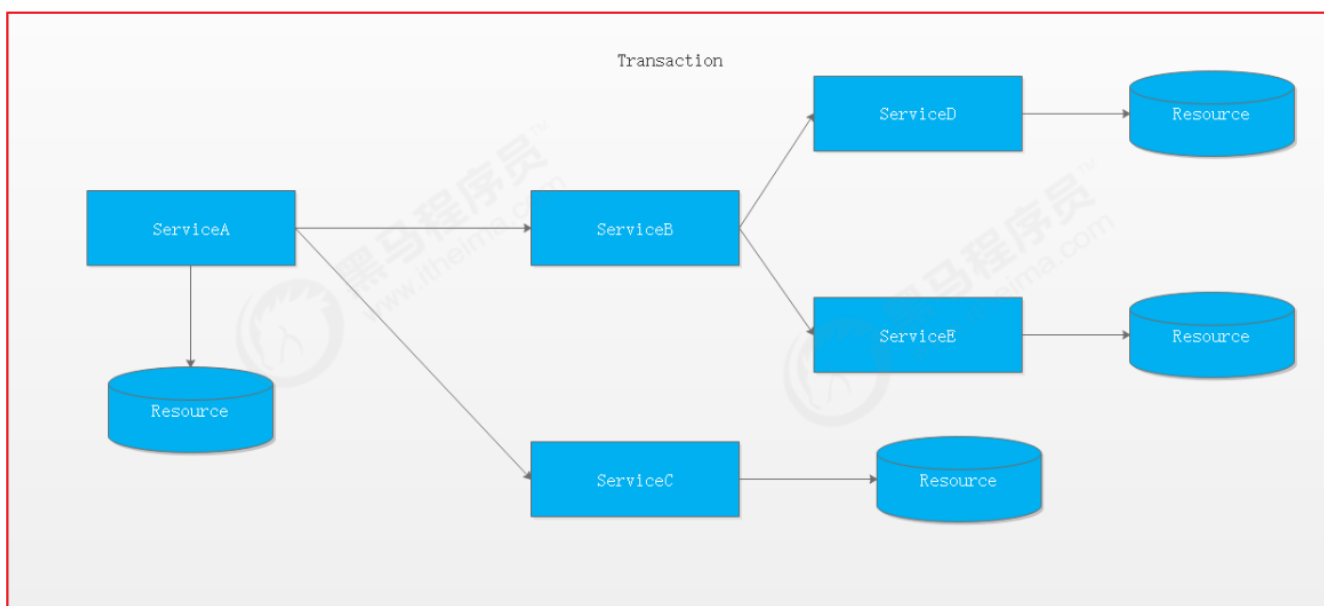


当一个服务操作访问不同的数据库资源, 又希望对它们的访问具有事务特性时, 就需要采用分布式事务来协调所有的事务参与者。

对于上面介绍的分布式事务应用架构, 尽管一个服务操作会访问多个数据库资源, 但是毕竟整个事务还是控制在单一服务的内部。如果一个服务操作需要调用另外一个服务, 这时的事务就需要跨越多个服务了。在这种情况下, 起始于某个服务的事务在调用另外一个服务的时候, 需要以某种机制流转 to 另外一个服务, 从而使被调用的服务访问的资源也自动加入到该事务当中来。下图反映了这样一个跨越多个服务的分布式事务:



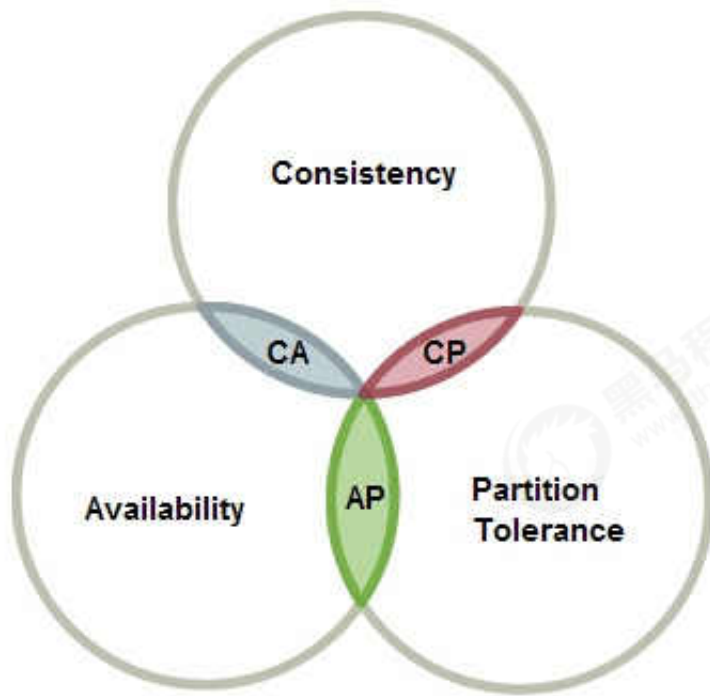
如果将上面这两种场景(一个服务可以调用多个数据库资源，也可以调用其他服务)结合在一起，对此进行延伸，整个分布式事务的参与者将会组成如下图所示的树形拓扑结构。在一个跨服务的分布式事务中，事务的发起者和提交均系同一个，它可以是整个调用的客户端，也可以是客户端最先调用的那个服务。



较之基于单一数据库资源访问的本地事务，分布式事务的应用架构更为复杂。在不同的分布式应用架构下，实现一个分布式事务要考虑的问题并不完全一样，比如对多资源的协调、事务的跨服务传播等，实现机制也是复杂多变。

4.2 分布式事务相关理论

4.2.1 CAP定理



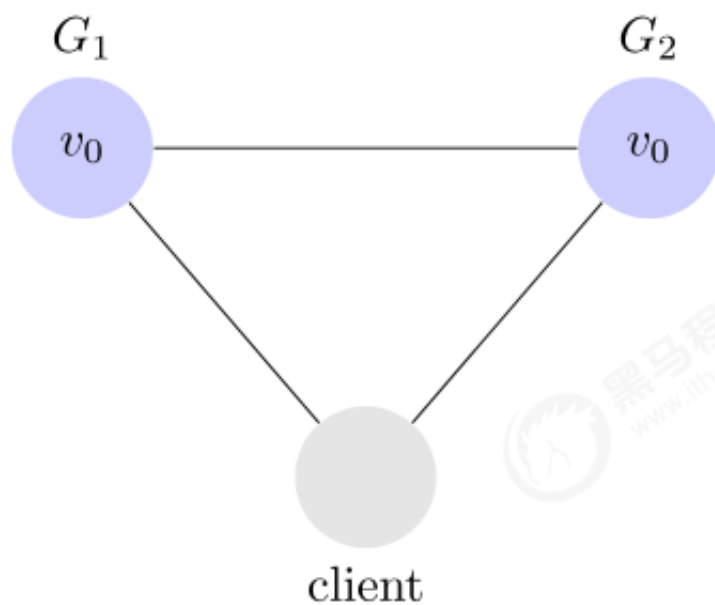
CAP定理是在 1998年加州大学的计算机科学家 Eric Brewer （埃里克.布鲁尔）提出，分布式系统有三个指标

- Consistency 一致性
- Availability 可用性
- Partition tolerance 分区容错

它们的第一个字母分别是 C、A、P。Eric Brewer 说，这三个指标不可能同时做到。这个结论就叫做 CAP 定理。

分区容错 **Partition tolerance**

大多数分布式系统都分布在多个子网络。每个子网络就叫做一个区（partition）。分区容错的意思是，区间通信可能失败。比如，一台服务器放在中国，另一台服务器放在美国，这就是两个区，它们之间可能无法通信。



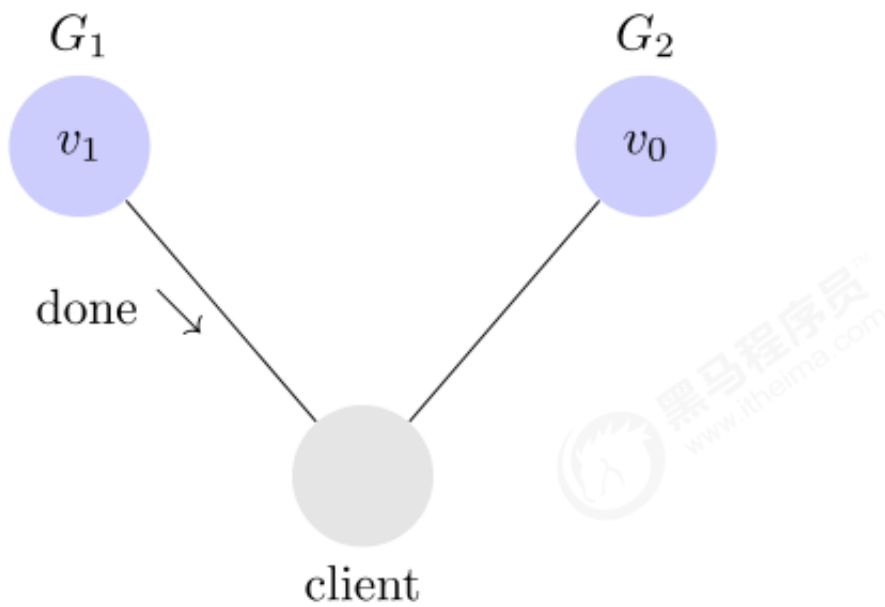
上图中，G1 和 G2 是两台跨区的服务器。G1 向 G2 发送一条消息，G2 可能无法收到。系统设计的时候，必须考虑到这种情况。

一般来说，分区容错无法避免，因此可以认为 CAP 的 P 总是成立。CAP 定理告诉我们，剩下的 C 和 A 无法同时做到。

可用性 **Availability**

Availability 中文叫做"可用性"，意思是只要收到用户的请求，服务器就必须给出回应。

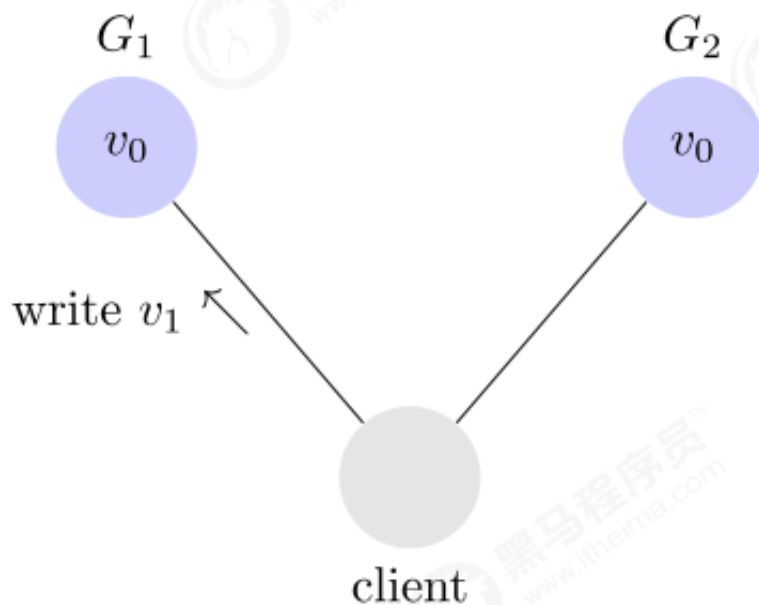
用户可以选择向 G1 或 G2 发起读操作。不管是哪台服务器，只要收到请求，就必须告诉用户，到底是 v0 还是 v1，否则就不满足可用性。



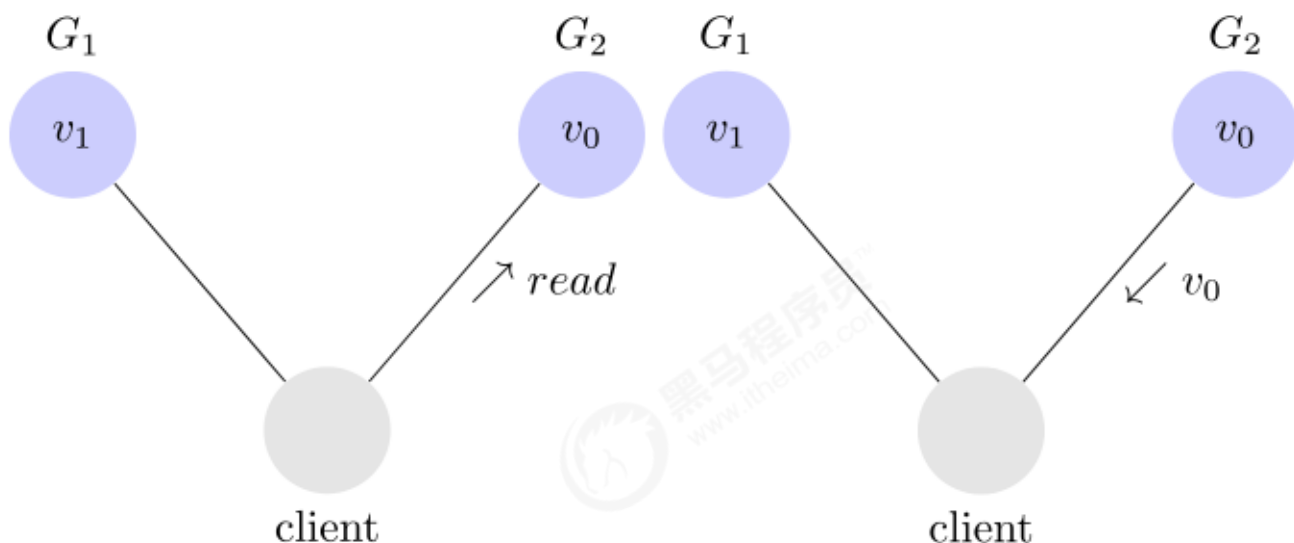
一致性 Consistency

Consistency 中文叫做"一致性"。意思是，写操作之后的读操作，必须返回该值。

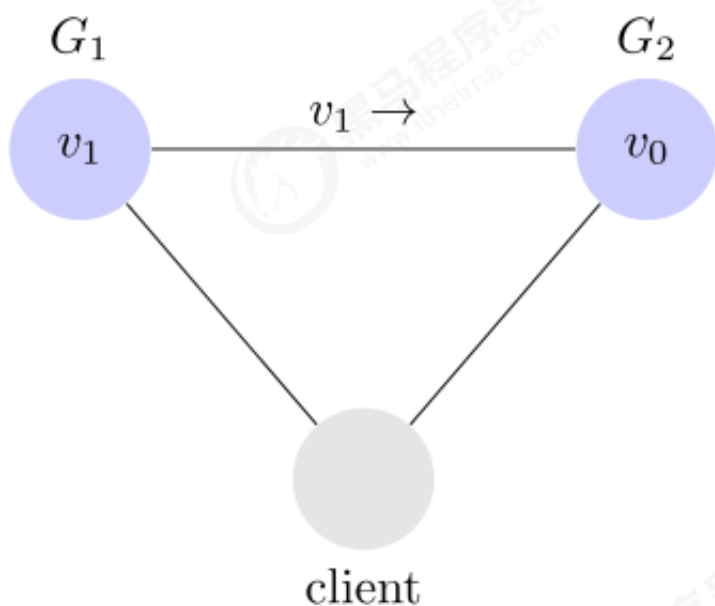
举例来说，某条记录是 v_0 ，用户向 G_1 发起一个写操作，将其改为 v_1 。



问题是，用户有可能向 G_2 发起读操作，由于 G_2 的值没有发生变化，因此返回的是 v_0 。 G_1 和 G_2 读操作的结果不一致，这就不满足一致性了。



为了让 G_2 也能变为 v_1 ，就要在 G_1 写操作的时候，让 G_1 向 G_2 发送一条消息，要求 G_2 也改成 v_1 。



一致性和可用性的矛盾

一致性和可用性，为什么不可能同时成立？答案很简单，因为可能通信失败（即出现分区容错）。

如果保证 G_2 的一致性，那么 G_1 必须在写操作时，锁定 G_2 的读操作和写操作。只有数据同步后，才能重新开放读写。锁定期间， G_2 不能读写，没有可用性不。

如果保证 G_2 的可用性，那么势必不能锁定 G_2 ，所以一致性不成立。

综上所述，G2 无法同时做到一致性和可用性。系统设计时只能选择一个目标。如果追求一致性，那么无法保证所有节点的可用性；如果追求所有节点的可用性，那就没法做到一致性。

4.2.2 BASE理论

BASE：全称：Basically Available(基本可用), Soft state（软状态）,和 Eventually consistent（最终一致性）三个短语的缩写，来自 ebay 的架构师提出。BASE 理论是对 CAP 中一致性和可用性权衡的结果，其来源于对大型互联网分布式实践的总结，是基于 CAP 定理逐步演化而来的。其核心思想是：

既是无法做到强一致性（Strong consistency），但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性（Eventual consistency）。

Basically Available(基本可用)

什么是基本可用呢？假设系统，出现了不可预知的故障，但还是能用，相比较正常的系统而言：

1. 响应时间上的损失：正常情况下的搜索引擎 0.5 秒即返回给用户结果，而基本可用的搜索引擎可以在 1 秒作用返回结果。
2. 功能上的损失：在一个电商网站上，正常情况下，用户可以顺利完成每一笔订单，但是到了大促期间，为了保护购物系统的稳定性，部分消费者可能会被引导到一个降级页面。

Soft state（软状态）

什么是软状态呢？相对于原子性而言，要求多个节点的数据副本都是一致的，这是一种“硬状态”。

软状态指的是：允许系统中的数据存在中间状态，并认为该状态不影响系统的整体可用性，即允许系统在多个不同节点的数据副本存在数据延时。

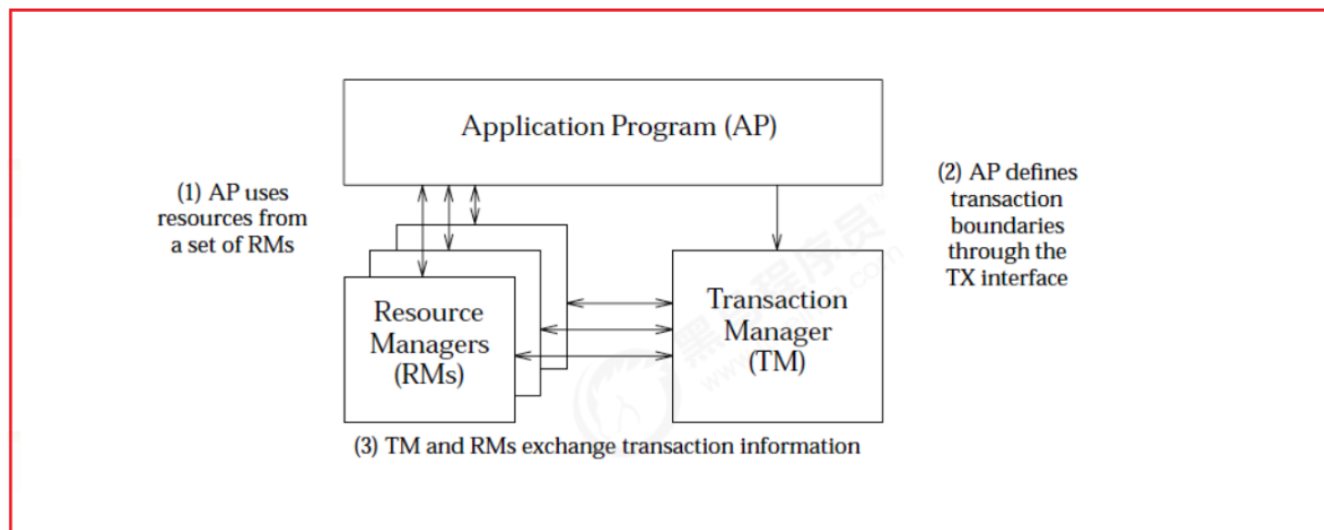
Eventually consistent（最终一致性）

系统能够保证在没有其他新的更新操作的情况下，数据最终一定能够达到一致的状态，因此所有客户端对系统的数据访问最终都能够获取到最新的值。

4.3 分布式事务解决方案

4.3.1 基于XA协议的两阶段提交

首先我们来简要看下分布式事务处理的XA规范：



可知XA规范中分布式事务有AP，RM，TM组成：

其中应用程序(Application Program，简称AP)：AP定义事务边界（定义事务开始和结束）并访问事务边界内的资源。

资源管理器(Resource Manager，简称RM)：Rm管理计算机共享的资源，许多软件都可以去访问这些资源，资源包含比如数据库、文件系统、打印机服务器等。

事务管理器(Transaction Manager，简称TM)：负责管理全局事务，分配事务唯一标识，监控事务的执行进度，并负责事务的提交、回滚、失败恢复等。

二阶段协议：

第一阶段TM要求所有的RM准备提交对应的事务分支，询问RM是否有能力保证成功的提交事务分支，RM根据自己的情况，如果判断自己进行的工作可以被提交，那就就对工作内容进行持久化，并给TM回执OK；否者给TM的回执NO。RM在发送了否定答复并回滚了已经的工作后，就可以丢弃这个事务分支信息了。

第二阶段TM根据阶段1各个RM prepare的结果，决定是提交还是回滚事务。如果所有的RM都prepare成功，那么TM通知所有的RM进行提交；如果有RM prepare回执NO的话，则TM通知所有RM回滚自己的事务分支。

也就是TM与RM之间是通过两阶段提交协议进行交互的。

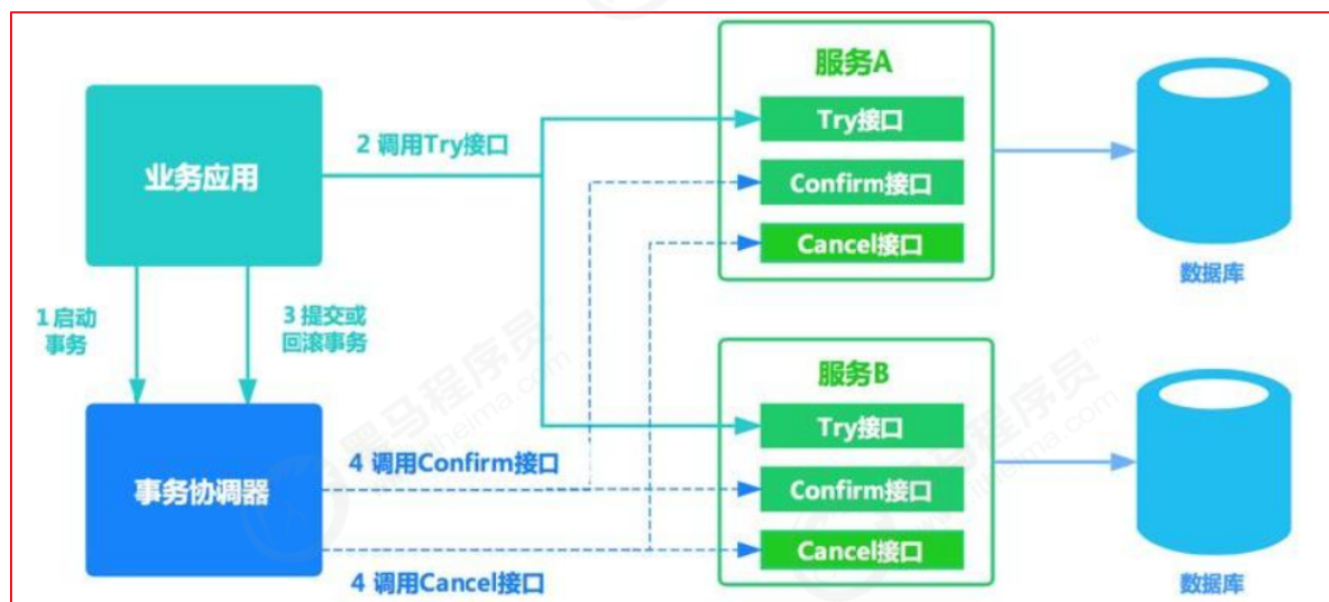
优点： 尽量保证了数据的强一致，适合对数据强一致要求很高的关键领域。（其实也不能100%保证强一致）

缺点： 实现复杂，牺牲了可用性，对性能影响较大，不适合高并发高性能场景。

4.3.2 TCC补偿机制

TCC 其实就是采用的补偿机制，其核心思想是：针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段：

- Try 阶段主要是对业务系统做检测及资源预留
- Confirm 阶段主要是对业务系统做确认提交，Try阶段执行成功并开始执行 Confirm 阶段时，默认 Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。
- Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。



例如：A要向 B 转账，思路大概是：

我们有一个本地方法，里面依次调用

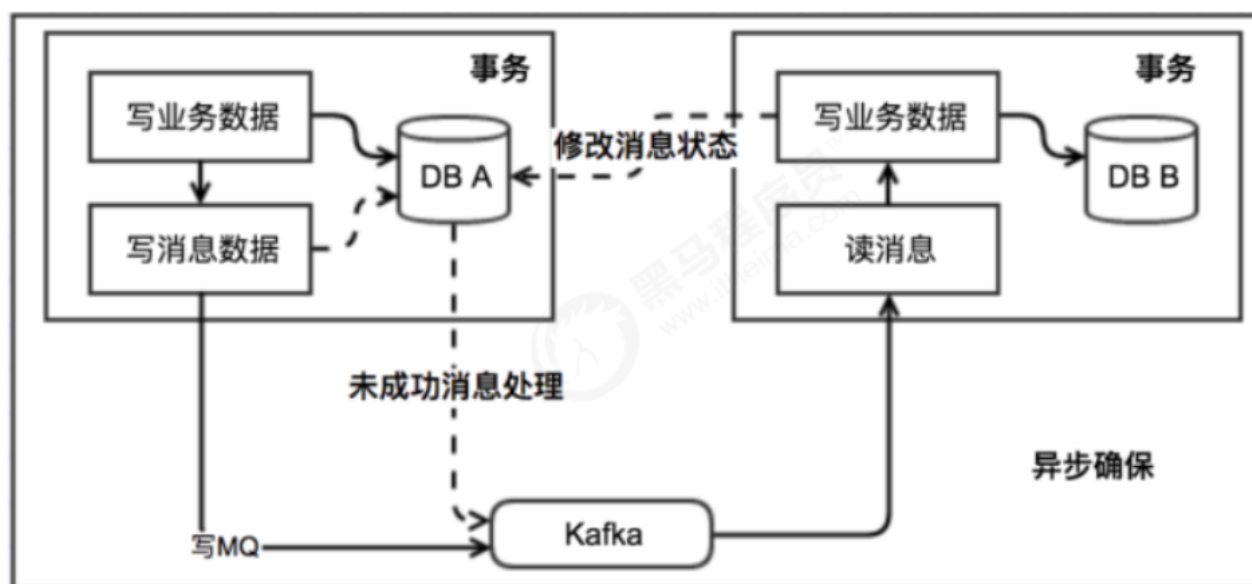
- 1、首先在 Try 阶段，要先调用远程接口把 B和 A的钱给冻结起来。
- 2、在 Confirm 阶段，执行远程调用的转账的操作，转账成功进行解冻。
- 3、如果第2步执行成功，那么转账成功，如果第二步执行失败，则调用远程冻结接口对应的解冻方法 (Cancel)。

优点：相比两阶段提交，可用性比较强

缺点：数据的一致性要差一些。TCC属于应用层的一种补偿方式，所以需要程序员在实现的时候多写很多补偿的代码，在一些场景中，一些业务流程可能用TCC不太好定义及处理。

4.3.3 消息最终一致性

消息最终一致性应该是业界使用最多的，其核心思想是将分布式事务拆成本地事务进行处理，这种思路是来源于ebay。我们可以从下面的流程图中看出其中的一些细节：



基本思路就是：

消息生产方，需要额外建一个消息表，并记录消息发送状态。消息表和业务数据要在一个事务里提交，也就是说他们要在一个数据库里面。然后消息会经过MQ发送到消息的消费方。如果消息发送失败，会进行重试发送。

消息消费方，需要处理这个消息，并完成自己的业务逻辑。此时如果本地事务处理成功，表明已经处理成功了，如果处理失败，那么就会重试执行。如果是业务上面的失败，可以给生产方发送一个业务补偿消息，通知生产方进行回滚等操作。

生产方和消费方定时扫描本地消息表，把还没处理完成的消息或者失败的消息再发送一遍。如果有靠谱的自动对账补账逻辑，这种方案还是非常实用的。

优点： 一种非常经典的实现，避免了分布式事务，实现了最终一致性。

缺点： 消息表会耦合到业务系统中，如果没有封装好的解决方案，会有很多杂活需要处理。

5. 库存扣减分布式事务的实现

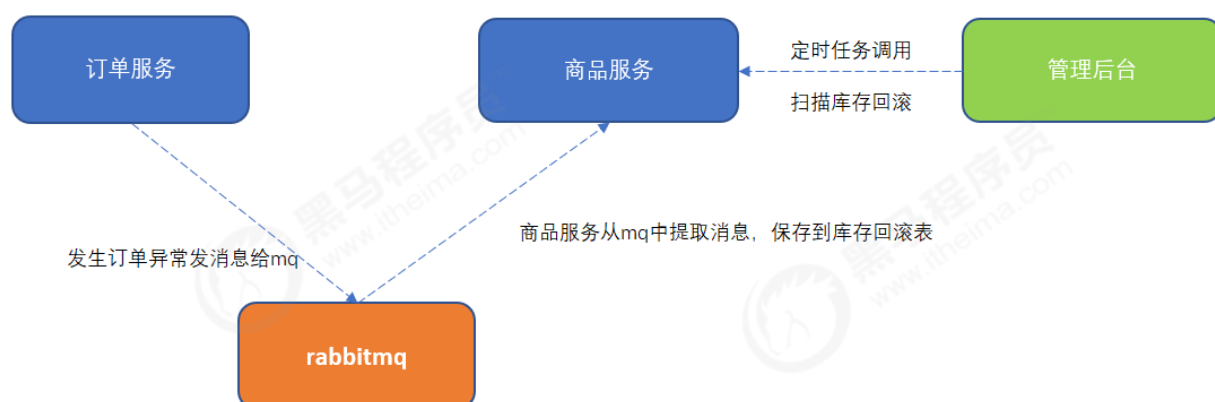
5.1 需求分析

如果订单在创建时发生了异常，此时库存已经扣减了，这样库存就比实际中的数量要少，造成数据不一致，为了避免这种情况，我们需要采用消息最终一致性的分布式事务解决方案。

5.2 实现思路

如下图：

- (1) 当订单服务发生异常时，发送消息给mq，消息内容为要购物车数据，用于恢复库存
- (2) 商品服务从mq提取消息，保存到库存回滚表中
- (3) 在管理后台开启定时任务，定时扫描库存回滚表执行库存回滚。



下面我们来看一下库存回滚表tb_stock_back的设计

字段名称	字段含义	字段类型	字段长度	备注
order_id	订单编号	VARCHAR		主键
sku_id	sku编号	VARCHAR		主键
num	回滚数量	INT		
status	状态	VARCHAR		0：未回滚 1：已回滚
create_time	创建时间	DATETIME		
back_time	回滚时间	DATETIME		

为什么要设置成联合主键的形式呢？因为消息在发送时，可能会因为失败而重复发送，而重复发送可能造成回滚数据的重复。订单编号+sku编号组合起来如果相同一定是同一笔数据，所以我们将其设置成联合主键，这样在插入重复数据时就会因为主键唯一冲突而无法插入。

5.3 代码实现

5.3.1 发送库存回滚消息

(1) qingcheng_service_order工程pom.xml引入依赖

```
<dependency>
    <groupId>org.springframework.amqp</groupId>
    <artifactId>spring-rabbit</artifactId>
    <version>2.1.4.RELEASE</version>
</dependency>
```

(2) 添加配置文件applicationContext-rabbitmq-producer.xml

```
<!--连接工厂-->
<rabbit:connection-factory id="connectionFactory" host="127.0.0.1"
port="5672" username="guest" password="guest" />
<rabbit:admin connection-factory="connectionFactory"></rabbit:admin>
<!--SKU 库存回滚队列 -->
<rabbit:queue name="queue.skuback" />
<rabbit:template id="rabbitTemplate" connection-
factory="connectionFactory" />
```

(3) 修改OrderServiceImpl的add方法

```
try{
    //保存订单主表
    //.....
    //保存订单明细
    //.....
}catch (Exception ex){
    rabbitTemplate.convertAndSend("", "queue.skuback",
JSON.toJSONString(orderItems));
    throw new RuntimeException("订单生成失败"); //抛出异常，让其回滚!
}
```

5.3.2 生成库存回滚记录

实现思路：从消息队列中取出购物车列表json，转换为列表后查询到库存回滚表。

(1) qingcheng_service_goods工程pom.xml引入依赖

```
<dependency>
    <groupId>org.springframework.amqp</groupId>
    <artifactId>spring-rabbit</artifactId>
    <version>2.1.4.RELEASE</version>
</dependency>
```

(2) 新增StockBack实体类

```
@Table(name = "tb_stock_back")
public class StockBack {

    @Id
    private String orderId;

    @Id
    private String skuId;

    private Integer num;

    private String status;

    private Date createTime;

    private Date backTime;

    // getter and setter....
}
```

(3) 新增StockBack数据访问接口

```
public interface StockBackMapper extends Mapper<StockBack> {
}
```

(4) StockBackService新增方法定义

```
public void addList(List<OrderItem> orderItems);
```

(5) StockBackServiceImpl实现方法

```

@Override
@Transactional
public void addList(List<OrderItem> orderItems) {
    for(OrderItem orderItem:orderItems){
        StockBack stockBack=new StockBack();
        stockBack.setOrderId(orderItem.getOrderId());
        stockBack.setSkuId(orderItem.getSkuId());
        stockBack.setStatus("0");
        stockBack.setNum(orderItem.getNum());
        stockBack.setCreateTime(new Date());
        stockBackMapper.insert(stockBack);
    }
}

```

(6) 编写消费端，从消息中取出消息，转换json字符串并调用服务方法

```

public class BackMessageConsumer implements MessageListener {

    @Autowired
    private StockBackService stockBackService;

    public void onMessage(Message message) {

        try {
            String jsonString = new String(message.getBody());
            List<OrderItem> orderItems = JSON.parseArray(jsonString,
OrderItem.class);
            stockBackService.addList(orderItems);
        } catch (Exception e) {
            e.printStackTrace();
            //记录日志
        }
    }
}

```

(7) qingcheng_service_goods添加配置文件applicationContext-rabbitmq-consumer.xml

```
<!--连接工厂-->
<rabbit:connection-factory id="connectionFactory" host="127.0.0.1"
port="5672" username="guest" password="guest" />
<rabbit:admin connection-factory="connectionFactory"></rabbit:admin>
<!--创建队列-->
<rabbit:queue name="queue.skuback" />
<!--消费者监听类-->
<bean id="messageConsumer"
class="com.qingcheng.consumer.BackMessageConsumer"></bean>
<!--设置监听容器-->
<rabbit:listener-container connection-factory="connectionFactory" >
    <rabbit:listener queue-names="queue.skuback" ref="messageConsumer"/>
</rabbit:listener-container>
```

5.3.3 定时执行库存回滚

实现思路：编写定时任务，间隔一小时执行库存回滚。查询库存回滚记录表中状态为0的记录，执行回滚逻辑。

(1) StockBackService新增方法定义

```
/**
 * 执行库存回滚
 */
public void doBack();
```

(2) StockBackServiceImpl实现方法

```

@Autowired
private SkuMapper skuMapper;

@Transactional
public void doBack() {
    System.out.println("库存回滚任务开始");
    //查询库存回滚表中状态为0的记录
    StockBack stockBack0=new StockBack();
    stockBack0.setStatus("0");
    List<StockBack> stockBackList =
stockBackMapper.select(stockBack0);
    for( StockBack stockBack:stockBackList){
        //添加库存
        skuMapper.deductionStock(stockBack.getSkuId(), -
stockBack.getNum() );
        //减少销量
        skuMapper.addSaleNum(stockBack.getSkuId(), -
stockBack.getNum());
        stockBack.setStatus("1");
        stockBack.setBackTime(new Date());
        stockBackMapper.updateByPrimaryKey(stockBack);
    }
    System.out.println("库存回滚任务结束");
}
}

```

(3) qingcheng_web_manager添加任务调度类

```

@Component
public class SkuTask {

    @Reference
    private StockBackService stockBackService;

    @Scheduled(cron = "0 0 0/1 * * ?")
    public void orderTimeOutLogic(){
        System.out.println("执行库存回滚");
        stockBackService.doBack();
    }
}

```