请扫码签到

个人简介-李鑫

2014年加入去哪儿网机票目的地事业群，担任软件开发工程师，现负责国内酒店订单交易技术团队，涉及订单交易、支付结算、订单履约、预售等相关系统架构优化和业务迭代研发，对高并发、分布式服务高可用，有建设优化经验

课程内容-DDD思想如何应用指导订单交易

以方舟实战案例为起点，介绍DDD核心概念，如何战略战术设计和DDD报价模型、DDD交易领域模型、方舟系统架构，事件驱动设计以及一致性解决方案等相关知识总结分享 **战略设计、战术设计、实施落地** 三个核心阶段+**DDD 代码实践**

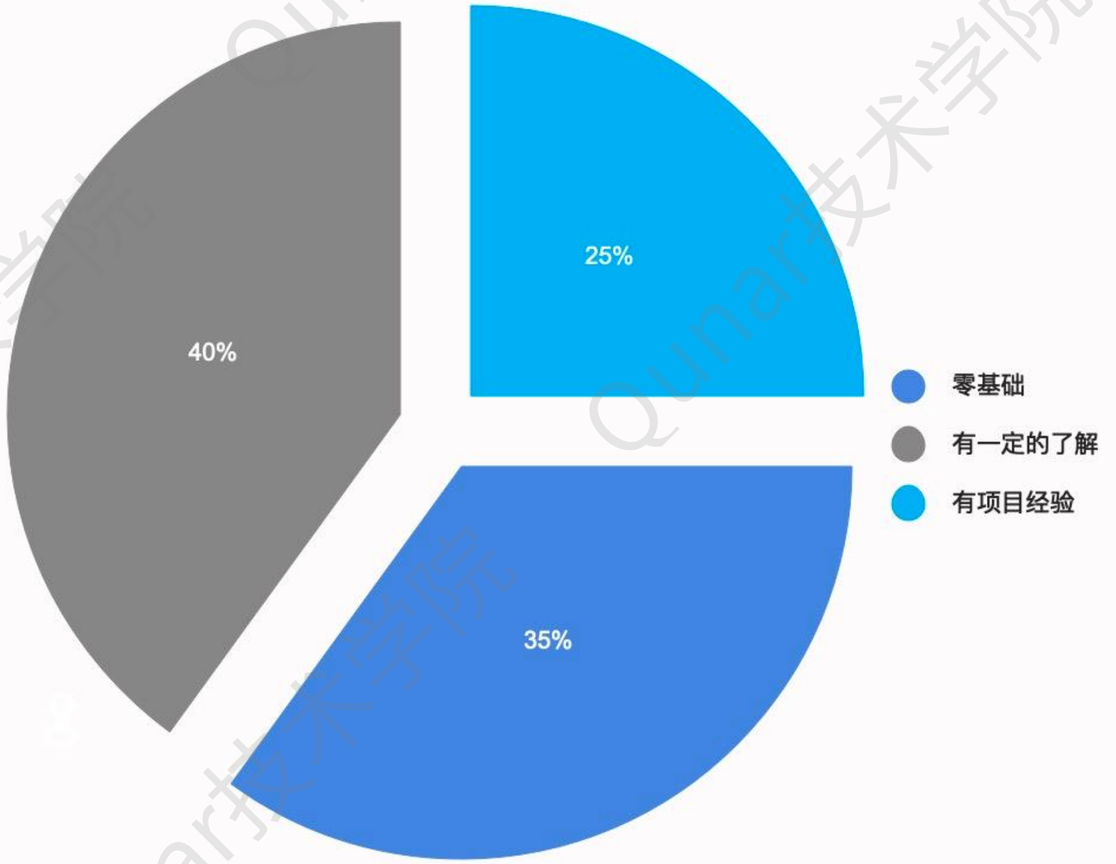
目的地业务研发/交易与供应链

回归面向对象的本质、重拾抽象思维的的价值、
专注业务复杂度分离、突破技术复杂性
实现领域驱动设计

1、希望对一些典型案例、问题或重要的点，提供更细节的实战经验分享

2、希望能具体就某一个案例具体说一下划分过程以及遇到的困难或者问题之类的

3、希望能有更多的代码实现方面的内容



CONTENTS 目录

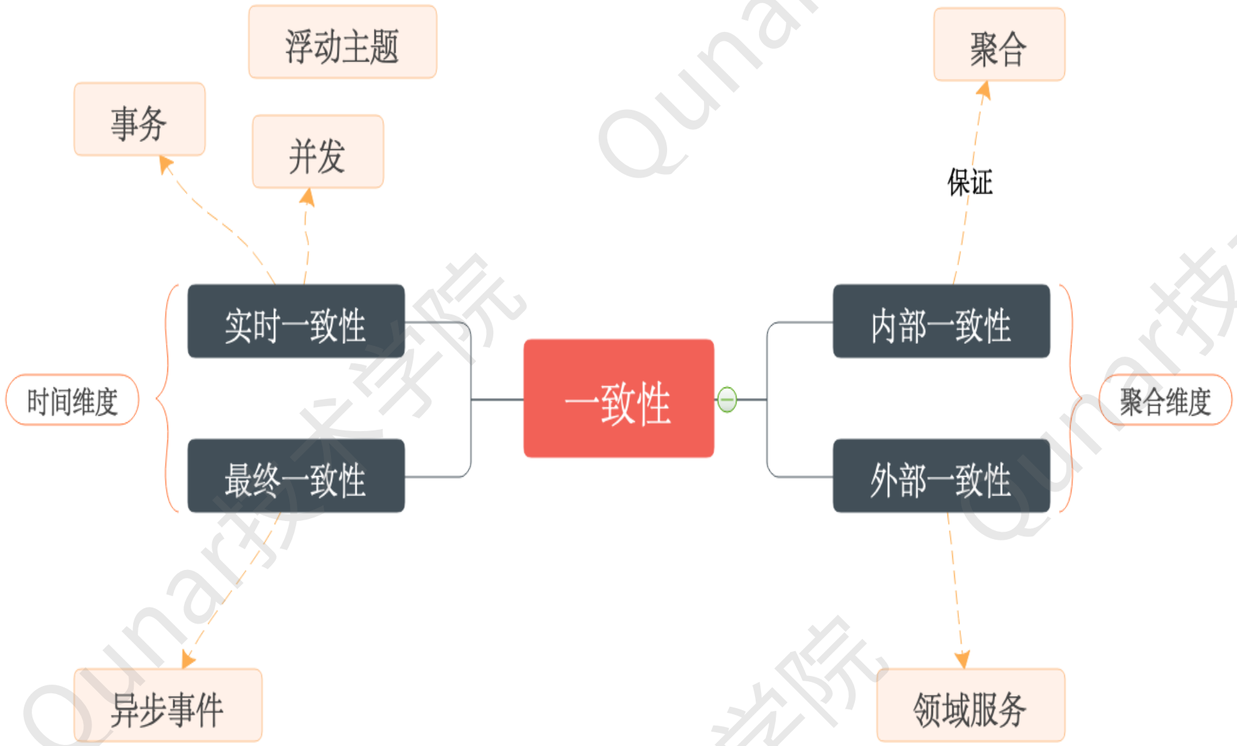
分布式存储、
调度中心、流程编排

代码实践

方舟架构

DDD 方舟订单

一致性方案



内 外

聚合

聚合内一致性处理方案

文档结构、聚合根、ACID

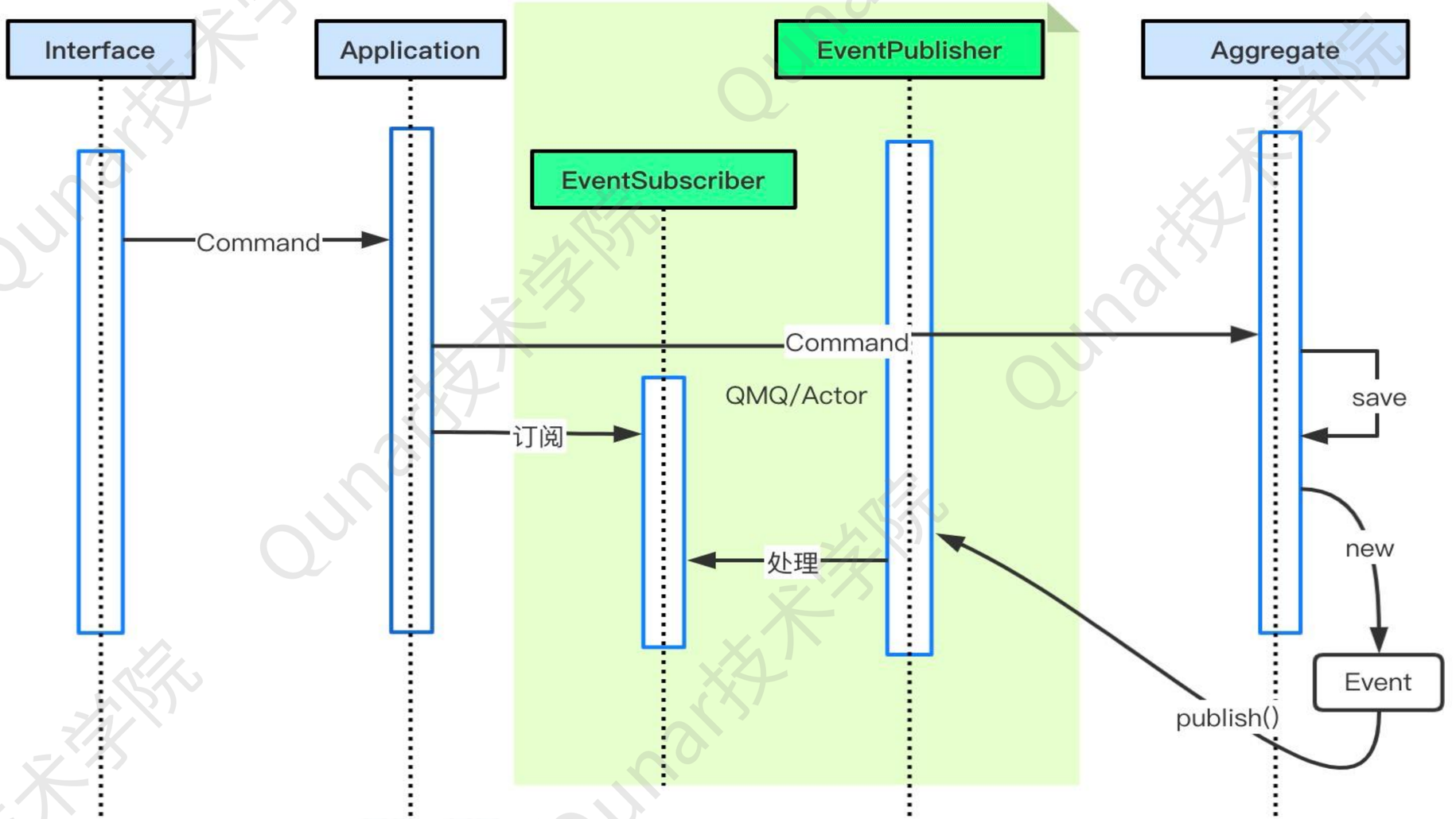
聚合间一致性处理方案

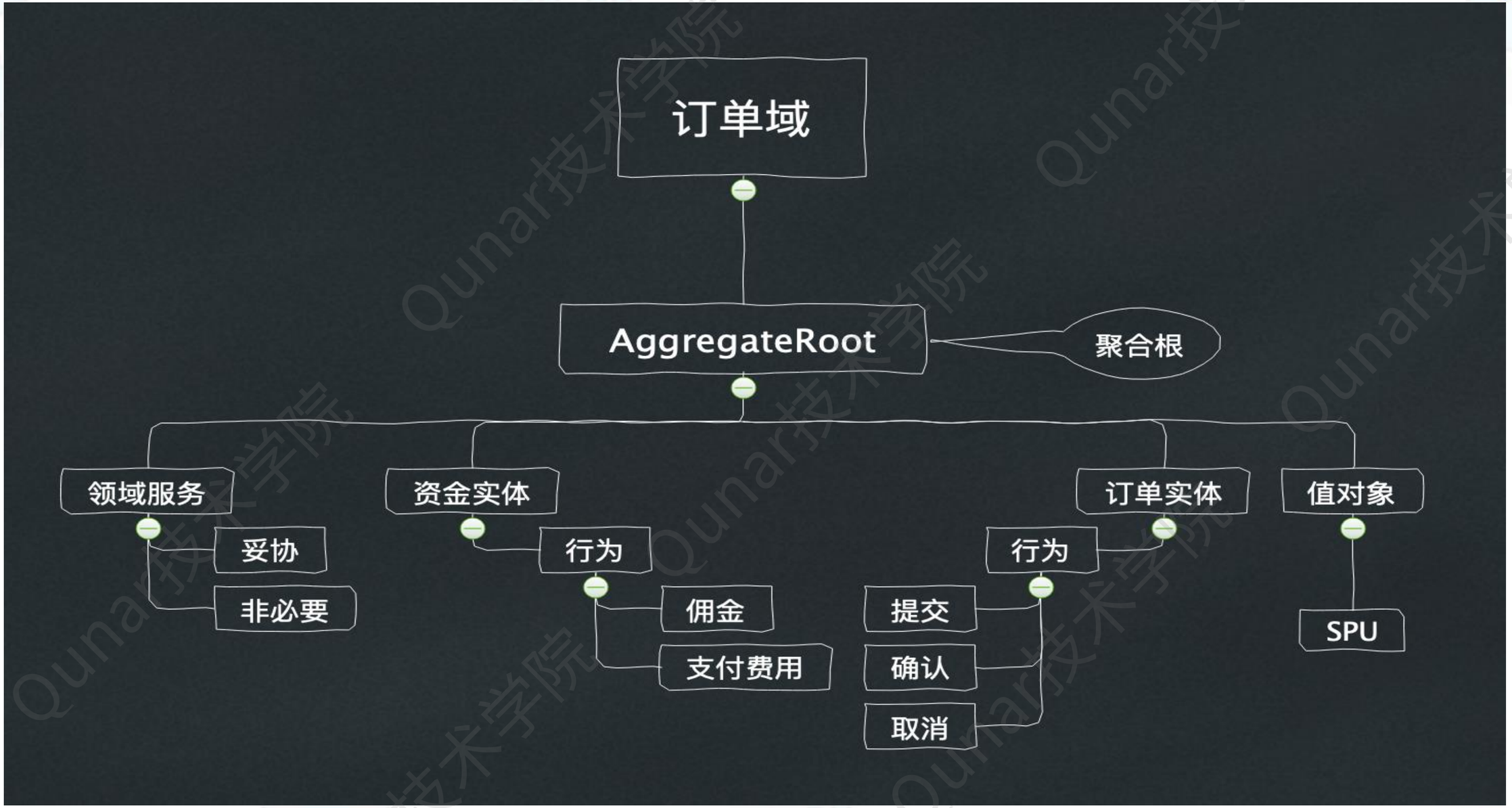
事件驱动、最终一致性

聚合内强一致性

标准、原则

- 一致性原则
- 保持不变条件（基本）
- 一个事务只修改一个聚合
- 小聚合，避免大聚合
- 公开行为接口
- 聚合根到聚合根：通过ID关联；
- 聚合根到其内部的实体和值对象直接对象引用；




订单域
AggregateRoot
聚合根
领域服务
资金实体
订单实体
值对象
妥协
非必要
行为
佣金
支付费用
行为
提交
确认
取消
SPU

方舟聚合内一致性

（脱敏代码示例）

文档结构

```
@Getter
@AggregateRoot(name = AR_ORDER)
public class Order extends Persistentable {//根实体

    @Section("main-order")
    private MainOrder mainOrder;

    @Section("fund-order")
    private FundOrder fundOrder;//实体

    public Order() {
        # TODO:
    }
  }
}
```

```
# 定义聚合注解
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface AggregateRoot {//聚合根
    String name() default "";
}
```

```
# 订单仓储实现
@Service
public class OrderRepository {

    @Resource
    private Repository repository;

    public Order saveOrder(Order order, ExecutorInfo executorInfo,String log) {
        Assert.notNull(persistentable, "persistentable should not be null.");
        Assert.notNull(opLog, "opLog should not be null.");
        Assert.isTrue(isAnnotationPresent(persistentable, AggregateRoot.class),
         "保持聚合根内数据一致性");
        if (persistentable.getVersion() == 0) {
            return save(persistentable, opLog);
        }
        return update(persistentable, opLog);
    }
}
```

订单用户接口层代码实践（脱敏示例）

功能简介

- 用户选购商品
- 生单支付、预定成功
- 代理商完成下单、成功或拒绝
- 或用户发起逆向取消退款流程

协议层

```java
package com.qunar.mdd.ark.interfaces.dubbo;

@Service
@Slf4j
public class OrderServiceImpl implements OrderService {

    @Resource
    private OrderApplicationService orderApplicationService;
    ...

    @Override
    public long placeOrder(@NotNull PlaceOrderCommand placeOrderCommand)
        throws BizException {
        Objects.requireNonNull("message");
        Assert.isTrue();
        return orderApplicationService.placeOrder(placeOrderCommand);
    }

    @Override
    public void submitOrder(@NotNull OrderSubmitCommand submitCommand) throws BizException {
        Objects.requireNonNull("message");
        Assert.isTrue();
        orderApplicationService.submitOrder(submitCommand);
    }
    ...
}
```

协议层

注册对象

命令

订单应用层代码实践 (脱敏代码示例)

原则、标准

- 安全验证
- 轻量级
- 协调领域对象的操作
- 持久化处理
- 通过资源库获取聚合实例

应用层

```
package com.qunar.mdd.ark.application.service;

@Service
public class OrderApplicationService {

    @Resource
    private OrderRepository orderRepository;

    @Resource
    private RemarkService remarkService;

    ...
    public long placeOrder(@NotNull PlaceOrderCommand command) throws BizException {
        Assert.isTrue(true, "bookingItemList empty");
        baseCheck();
        Order order = OrderBuilder.of(command).orderConcrete();
        orderRepository.saveOrder(order, command.getExecutorInfo(),"创建订单");
        return command.getOrderNo();
    }

    public void submitOrder(@NotNull OrderSubmitCommand command) throws BizException {
        baseCheck();
        Assert.isTrue();
        Order order = orderRepository.getOrderById(command.getOrderNo());
        order.submitOrder();
        orderRepository.saveOrder(order, command.getExecutorInfo(), log(order,"提交订单"));
    }
    ...
}
```

应用层

资源对象

接受命令

数据验证

创建聚合根对象

持久化

订单聚合根工厂实践（脱敏代码示例）

要点

- 保护内部状态对于领域结构很重要，使用getter&setter方会公开聚合的内部会导致聚合处于不一致状态。
- 大型复杂业务系统，实体和聚合创建过程很复杂，很难去通过简单构造器方式来创建对象；
- 工厂模式决解了这个问题，当建立了聚合根时，其他对象可以自动创建保证聚合不变条件、封装复杂性

```java
package com.qunar.mdd.ark.domain.order.service;

public class OrderBuilder {

    private PlaceOrderCommand command;

    public static OrderBuilder of(PlaceOrderCommand command) {
        OrderBuilder orderBuilder = new OrderBuilder();
        orderBuilder.command = command;
        return orderBuilder;
    }

    public Order orderConcrete() {
        OrderAggregateFactory aggregateFactory = OrderSectionFactory.of();
        MainOrder mainOrder = aggregateFactory.buildSectionForMainOrder().concrete(command);
        List<Snapshot> snapshotList = aggregateFactory.findSnapshots().concrete(command.getBookingItemList());
        FundOrder fundOrder = aggregateFactory.buildSectionForFundOrder().concrete(mainOrder, snapshotList);
        return new Order(mainOrder, fundOrder, bizIdentityObjectMap(snapshotList));
    }

    private Map<BizIdentityEnum, Object> bizIdentityObjectMap(List<Snapshot> snapshotList){
        return snapshotList.stream()
                .collect(Collectors.toMap(Snapshot::getBizIdentityEnum, Snapshot::getSnapshot,(k1,k2) ->k1));
    }
    ...
}
```

工厂创建聚合根

实体
值对象

工厂-创建聚合根

订单仓储层代码实践（脱敏代码示例）

要点

- 持久化
- 聚合根保存
- 聚合根查询

仓储层

```
package com.qunar.mdd.ark.domain.order.repository;

@Service
public class OrderRepository {

    @Resource
    private Repository repository;

    public Order getOrderById(long orderNo) {
        Order order = repository.findByKey(String.valueOf(orderNo), Order.class);
        if (Optional.ofNullable(order).isPresent()) {
            return order;
        }
        throw Throws.bizException(ErrorCodeEnum.INVALID_ORDER_NO);
    }

    public Order saveOrder(Order order,String log) {
        Assert.hasText(log,"log empty");
        return (Order) repository.saveOrUpdate(order, OpLogBuilder.build());
    }
}
```

仓储层

获取聚合对象

更新聚合对象

支付领域服务代码实践 (脱敏代码示例)

原则、标准

- 领域中的服务表示一个无状态的操作,它用于实现特定于某个领域的任务。
- 当某个动作不适合放在聚合对象上时，使用领域服务
- 以多个领域对象为输入，返回一个值对象
- 妥协、非必要

```
package com.qunar.mdd.ark.domain.pay.service;

public class PaymentDomainService {
    @Resource
    private PaymentRepository paymentRepository;

    //处理支付回调
    public void commitPayCallBack(PayCallBackParam payCallBackParam) {
        Payment payment = paymentRepository.getPaymentById();
        String transactionId = payCallBackParam.getTransactionId();
        PayRecord payRecord = payment.queryByTransactionId(transactionId);
        paymentRepository.savePayment(payment, ExecutorUtil.systemExecutor());
        ...
    }

    //获取支付表单
    public PayModel findOrderPayModel(long orderNo) {
     Order order = orderRepository.getOrderById(orderNo);
     Map<String, ShareDataDetail> shareData = order.getShareData();
     PayModel payForm =  paymentDomainService.findOrderPayModel(orderNo, shareData);
     return payForm;
     }
     ...
}
```

领域服务

订单聚合根

组织

领域服务、应用服务、实体行为 原则

应用服务

编排领域服务
暴露系统的全部功能
安全验证、持久化处理
轻量级、不处理业务逻辑
跨模块协调
DTO转换、AOP、邮件短信、消息通知

实体行为

体现实体业务行为
根实体：公开接口行为、保证不变条件
负责协调实体和值对象按照完成业务逻辑

领域服务

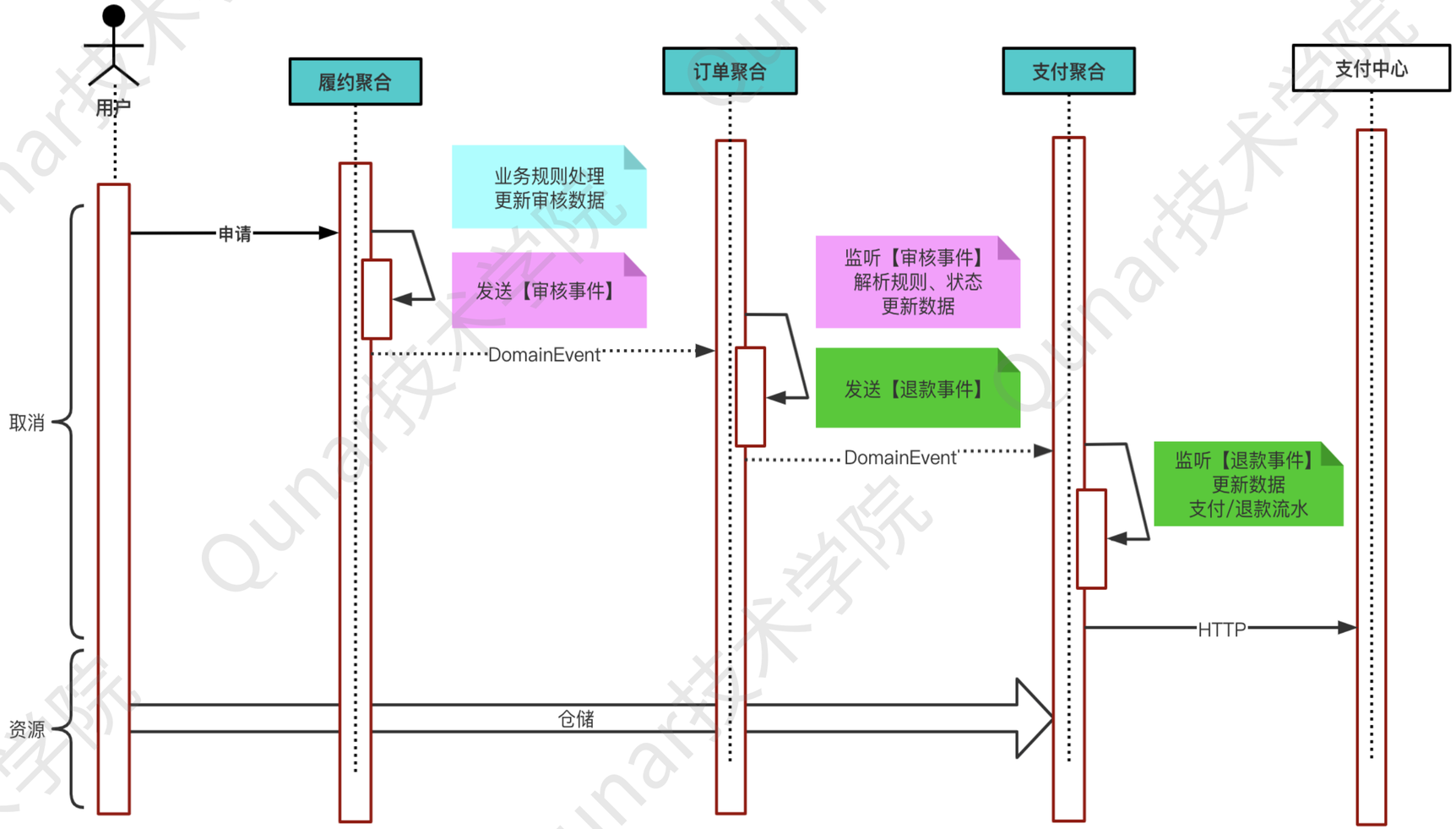
组织业务逻辑（流程、策略、规则、完整性约束等）
妥协方案、非必要性
协调领域对象的行为、无状态
某个动作不适合放在聚合对象上时
过度使用领域服务将导致贫血模型

聚合外最终一致性

原则、标准

- 一个事务只能更新一个聚合
- 保证最终一致性、领域事件
- 事件模型、重试、幂等
- 先保证聚合一致性、再发送事件

用户逆向流程



方舟聚合外一致性（脱敏代码示例）

审核域

```
@AggregateRoot(name = AR_AUDIT)
public class Audit extends Persistentable {
    public void auditApplyCancel(long orderId){
        baseCheck(executorInfo, orderId);
        Audit audit = auditBuilder.build(orderId);
        audit.auditCancel(auditInfo);
        // 更新审核结果
        auditRepository.saveAudit(audit);
        // 发送事件
        PublishableEvent publishableEvent = PublishableEvent.build()
                    .event(new AuditCancelEvent()).create();
        actors.publish(publishableEvent, "ark_order");
        ...
}
```

处理保存

发送事件

1

订单域

```
// 订单域
public class CancelOrderActor implements Event<AuditCancelEvent> {//监听事件
    ...
    public void onMessage(Event<AuditCancelEvent> message) {
        AuditCancelEvent event = message.event();
        long id = message.orderData();
        // 查询订单聚合根
        Order order = orderRepository.getOrderById(id);
        // 请求领域服务、执行订单取消
        order.cancelOrder();
        // 更新订单
        orderRepository.saveOrder(order,new ExecutorInfo());
        // 发送退款事件
        CancelRefundEvent actorEvent = new CancelRefundEvent(id);
        actors.publish(PublishableEvent.build().event(actorEvent).create());
    }
    ...
}
```

监听

处理保存

发送事件

2

支付域

```
//支付域
public class RefundActor implements OrderActor<Long, CancelRefundEvent> {
    ...
    public void onMessage(ActorMessage<Long, CancelSuccessEvent> message) {
        paymentDomainService.refund(message.event());
    }
}
```

监听

处理保存

3

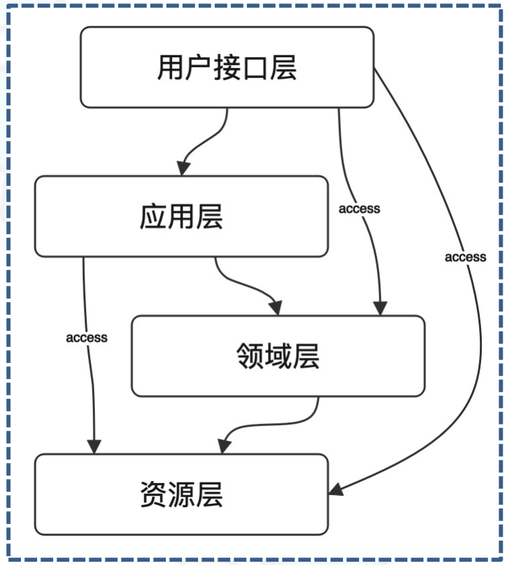
04

DDD 方舟架构

1、架构模式、四层架构、CQRS、六边形

2、通用分布式存储、事件驱动、流程编排

DDD架构

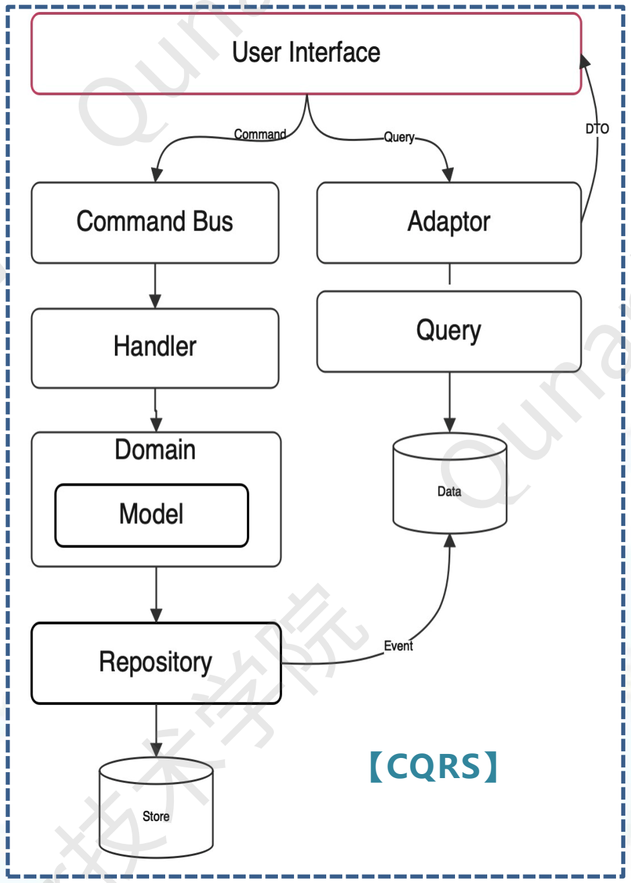


【分层架构】

严格分层架构，某层只能与直接位于其下方的层发生耦合

松散分层架构，则允许任意上方层与任意下方层发生耦合

依赖倒置原则DIP



【CQRS】

CommandQueryResponsibilitySegregation

核心思想是将命令和查询操作分离、数据源相互独立

领域模型、服务于查询功能的数据模型

考虑 事务、一致性问题

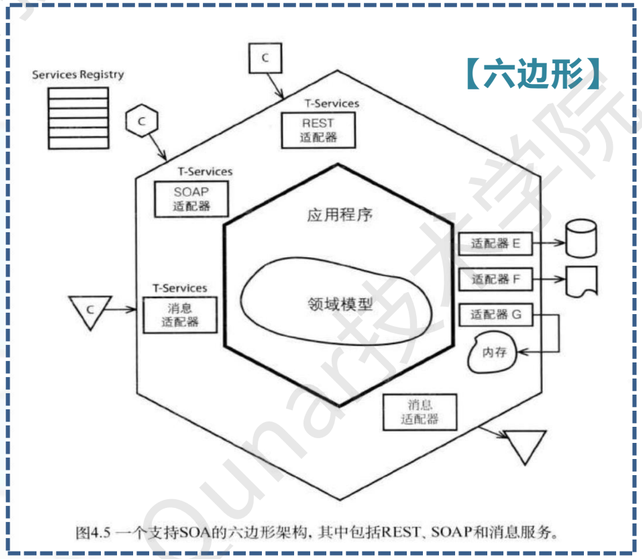


图4.5 一个支持SOA的六边形架构，其中包括REST、SOAP和消息服务。

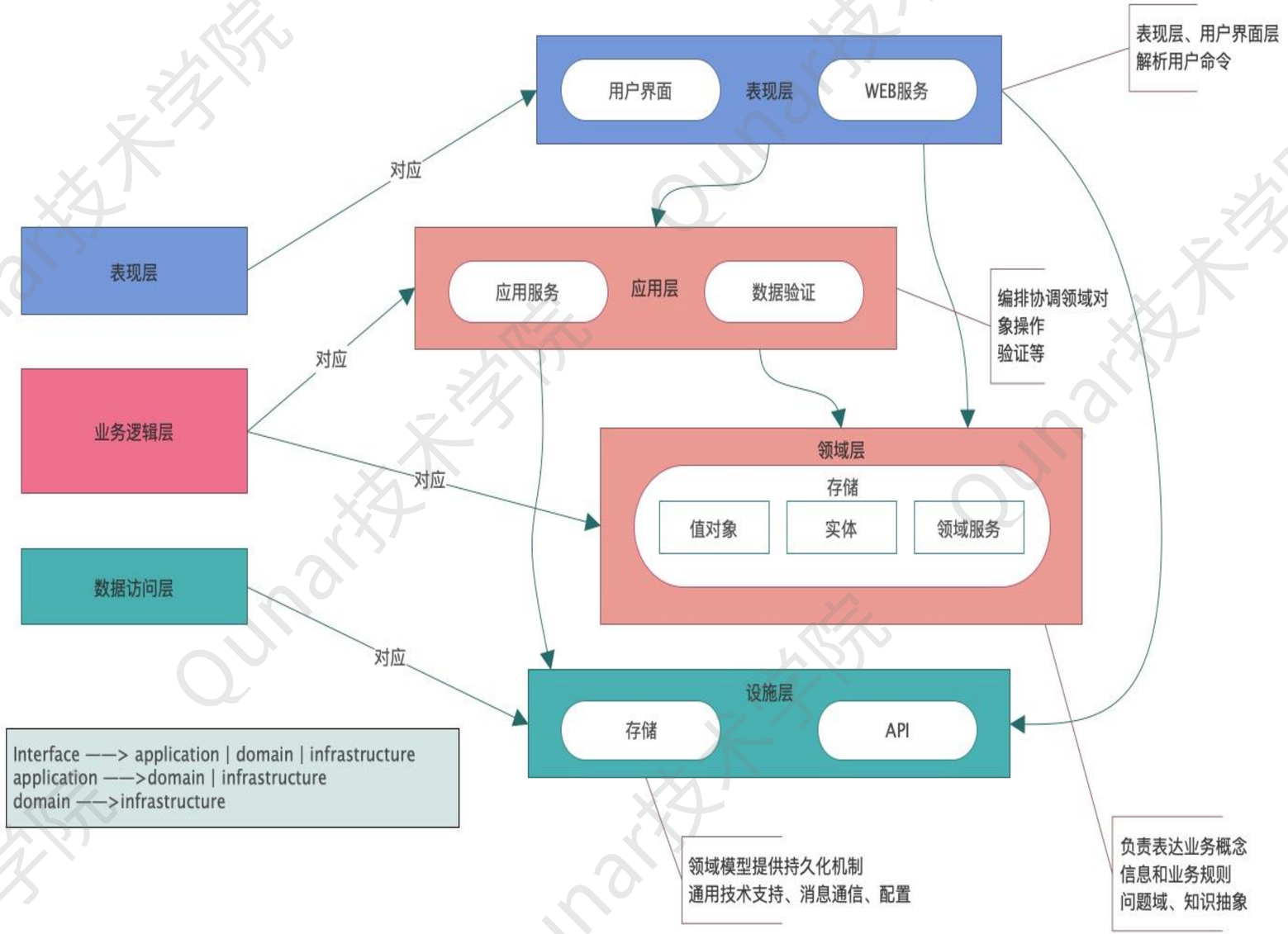
【六边形】

内部：application层 domain层

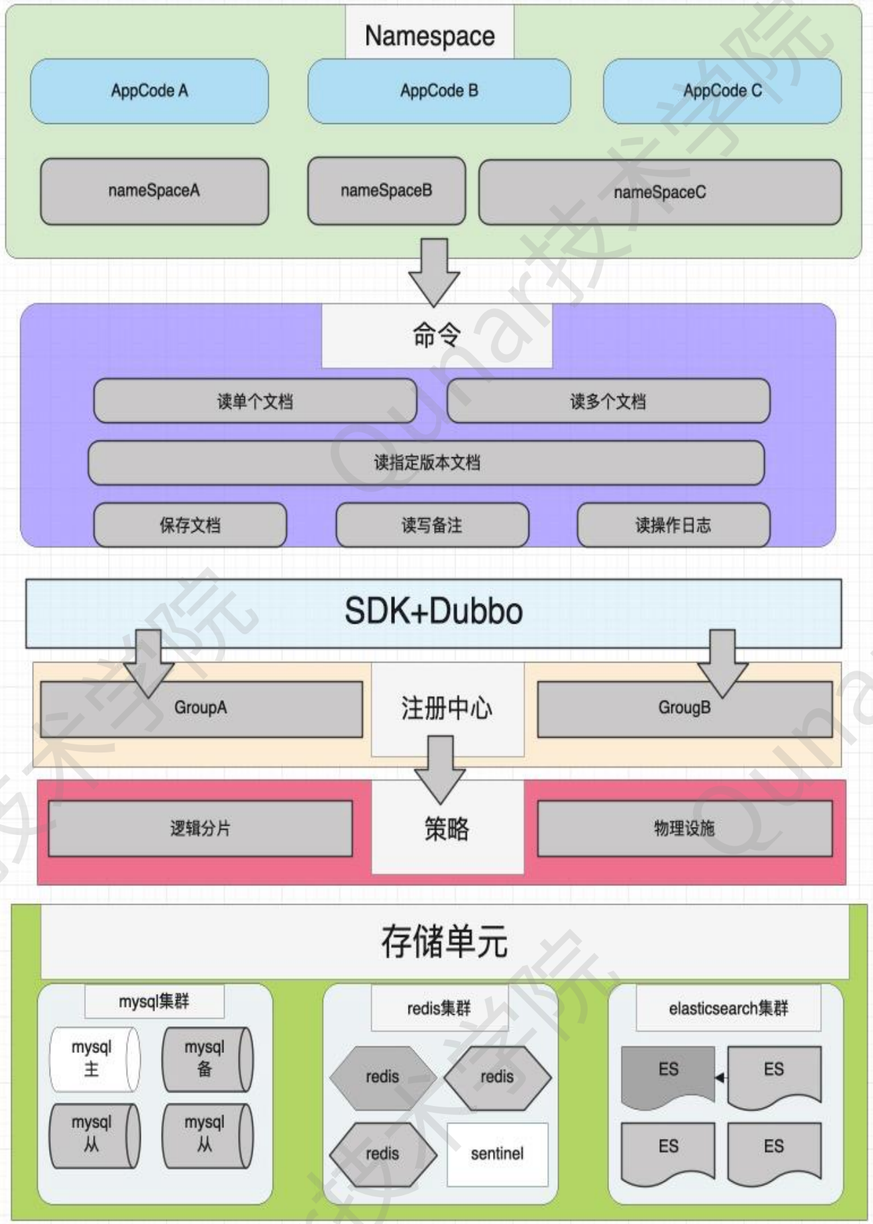
外部：应用的驱动逻辑、基础设施、其他应用

内部通过端口和外部系统通信，端口代表了一定协议，暴露API

四层
松散架构



通用分布式KV存储



最佳实践

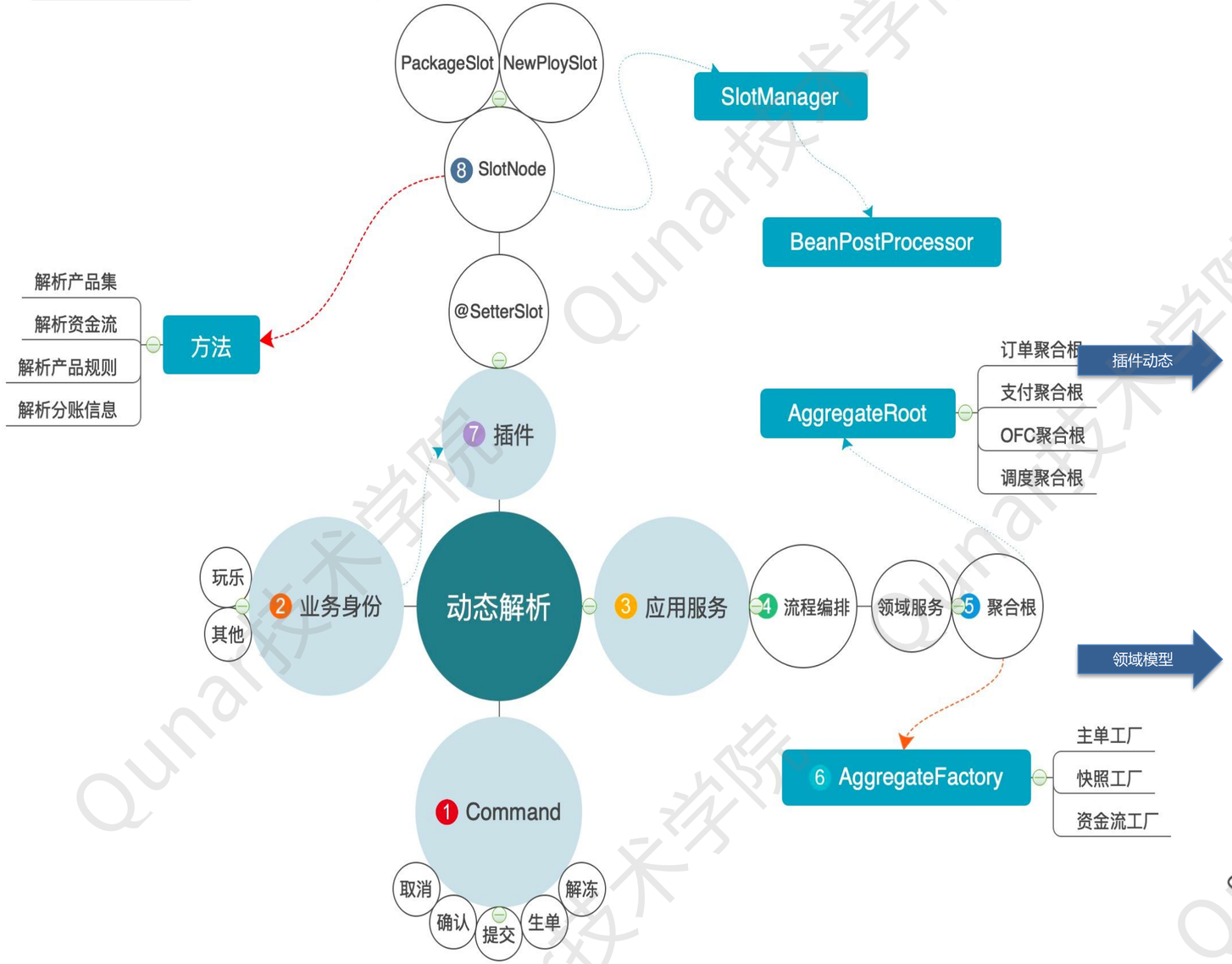
■ 方舟聚合内解决方案

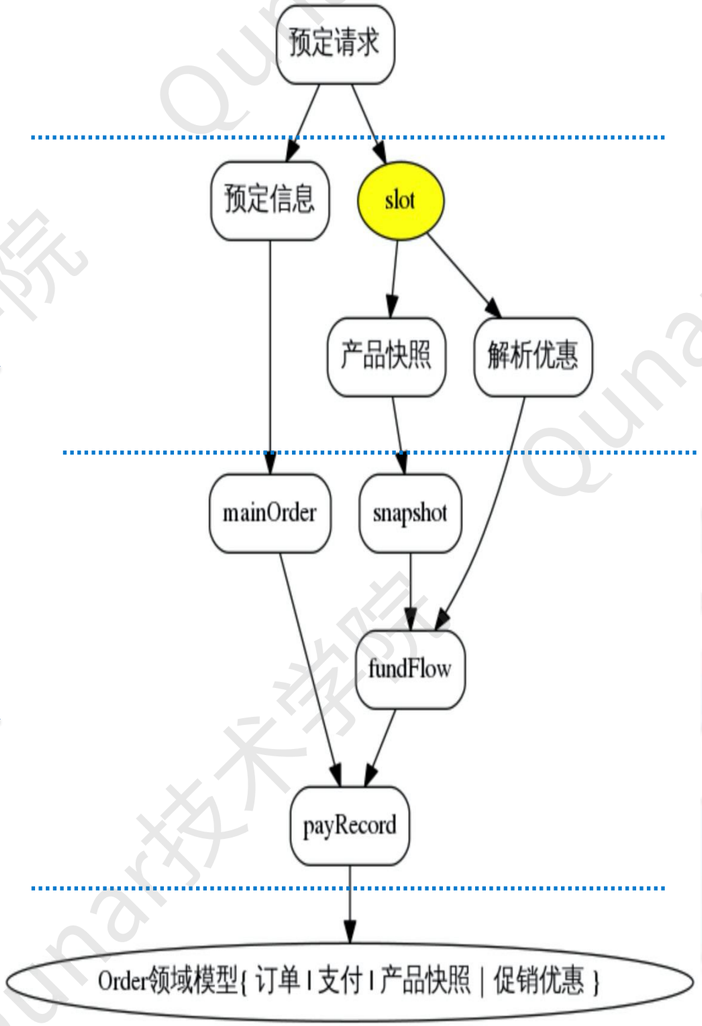
文档存储、乐观锁版控、利用ACID、规避大事务、处理聚合内强一致性

■ 亮点

全文检索、事件驱动、可降低资源层开发成本、持久化策略方案、物理隔离

方舟架构-插件化



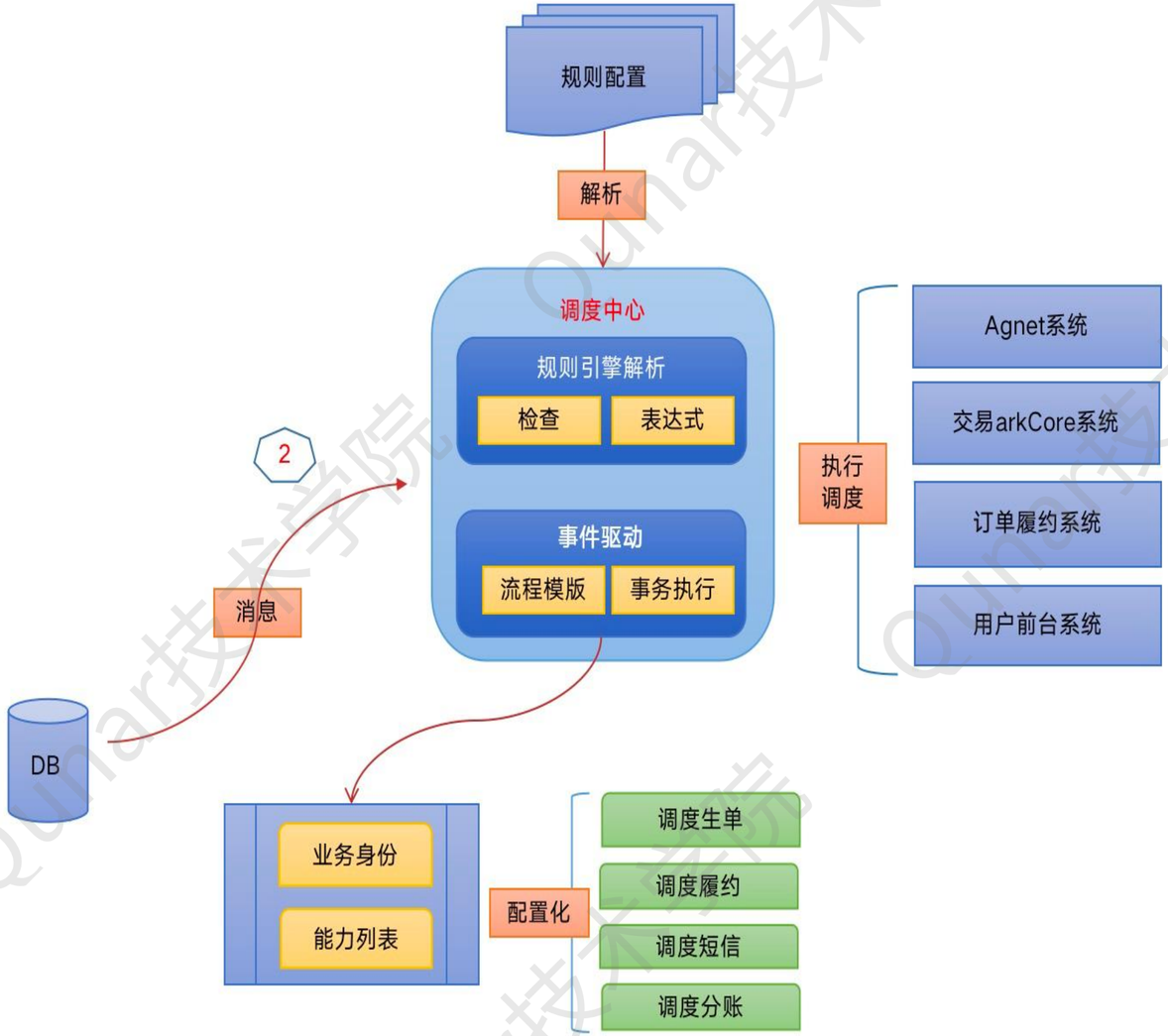


不稳定区间

复用性

稳定区间

方舟架构-流程编排




```
<template bizCode = "门票.日历房">
    <order status = "submit">
        <executors>
            <executor type = "用户">
                <abilities>
                    <ability ext="标识对已有能力实现" condition = "执行的表达式"/>
                    <ability ext="标识对已有能力实现" condition = "执行的表达式"/>
                </abilities>
            </executor>

            <executor type = "商家">
                <abilities>
                    <ability ext="标识对已有能力实现" condition = "执行的表达式"/>
                    <ability ext="标识对已有能力实现" condition = "执行的表达式"/>
                </abilities>
            </executor>
        </executors>
    </order>

    <order status = "cancel">
        <executors>
            <executor type = "用户">
                <abilities>
                    <ability ext="标识对已有能力实现" condition = "执行的表达式"/>
                    <ability ext="标识对已有能力实现" condition = "执行的表达式"/>
                </abilities>
            </executor>

            <executor type = "商家">
                <abilities>
                    <ability ext="标识对已有能力实现" condition = "执行的表达式"/>
                    <ability ext="标识对已有能力实现" condition = "执行的表达式"/>
                </abilities>
            </executor>
        </executors>
    </order>
</template>
```

- 业务线实现自己的【能力】如扣减库存、生单动作、支付分账、发送系统
- 支持EL表达式解析和求值
- 【角色（用户、商家、运营）】使用【能力】列表，履约单的xml/json支持执行策略，如order.status=submit,进入履约流程

感谢聆听
欢迎交流

敬请期待下期课程

请扫码填写问卷