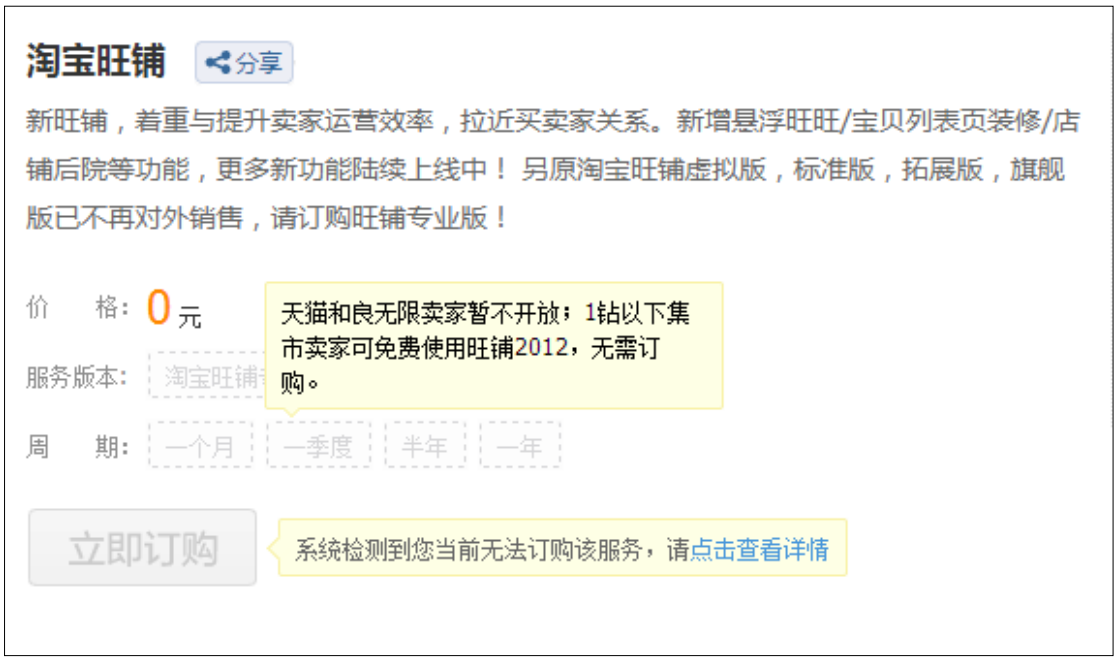
QlExpress 实战宝典

场景一：（淘宝服务平台）

http://fuwu.taobao.com/ser/detail.htm?service_code=App_Service_W_Shop
当你登陆淘宝之后，你会发现并不是所有的卖家都可以或者需要订购淘宝旺铺。
而类似的购买条件非常多，依赖方很多。产品层面，运营希望尽量少的了解业务，而且可以灵活配置条件的组合。例如：
!（天猫卖家 or 良无限卖家）and !（一钻以下卖家）



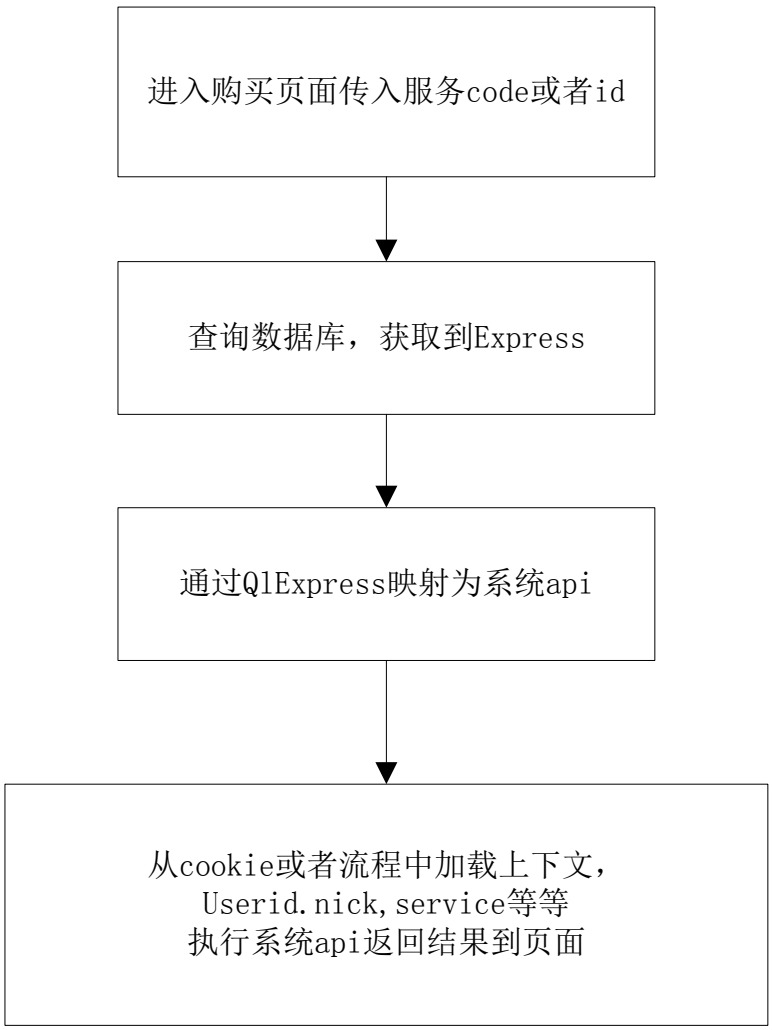
实战方案（qlExpress+ spring 框架）

数据定义以及流程设计

数据表定义：taobao_service

Id	Code	Name	Express
1	App_Service_W_Shop	淘宝旺铺	!（天猫卖家 or 良无限卖家）and !（一钻以下卖家）
2	……	……	……

系统流程设计：



业务层 spring 的实现：【示意代码】

```
<bean id="tmallService" class="com.taobao.hsf.app.spring.util.HSFSpringConsumerBean" init-method="init">
```

```
        <property name="interfaceName" value="com.taobao. tmall.TmallService " />
        <property name="version" value="${******.version}" />
    </bean>
    <bean id="liangwuxianService" class="com.taobao.hsf.app.spring.util.HSFSpringConsumerBean" init-method="init">
        <property name="interfaceName" value="com.taobao. tmall. LiangwuxianService " />
        <property name="version" value="${******.version}" />
    </bean>
    <bean id="userManager" class="com.taobao. UserManager "/>
```

词组映射框架

业务层底层映射 xml 配置：

```
    <atomExpress>
    <name>天猫卖家</name>
    <expression> tmallService.isTmall(userId ,bizType)</expression>
    </atomExpress>
    <atomExpress>
    <name>良无限卖家</name>
    <expression>liangwuxian.userExits (userId,nick)</expression>
    </atomExpress>
    <atomExpress>
    <name>一钻以下卖家</name>
    <expression>userManager.leverJudge (userId,1,"diamond",true)</expression>
    </atomExpress>
```

解析 XML 文件,扩展 QlExpress 的 ExpressRunner：

```
public void initialExpressFromLocalXml(ExpressRunner runner){
    List<AtomExpressDTO> list = loadAtomExpressLocal
            .getAllLocalAtomRule();//解析 XML 文件，在此不累赘实现
    for (AtomExpressDTO atom : list) {
        runner.addMacro(atom.getName(),atom.getExpress());
    }
}
```

业务层调用

扩展的执行器 MyExpressExecutor

```java
public class MyExpressExecutor extends ApplicationContextAware,
        InitializingBean
{
    ExpressRunner expressRunner = new ExpressRunner();

    @Override
    public void afterPropertiesSet() throws Exception {
        //从XML解析
        initialExpressFromLocalXml(this.expressRunner);
    }

    @Override
    public void setApplicationContext(ApplicationContext aContext)
    throws BeansException {
        applicationContext = aContext;
    }

    /**
     * 表达式执行
     * @param expressContext
     * @param express
     * @return
     */
    public Object coreExecute(ExpressContext expressContext, String express){
        try {
            expressContext.setApplicationContext(this.applicationContext);
            Object value = this.expressRunner.execute(express,
                    expressContext, null, true, false);
            return value;
        } catch (Exception e) {
            // 异常处理
            String errorMsg = handleExceptionResult(express, expressContext);
            logger.error(errorMsg, e);
        }
    }
}

```

扩展的上下文 ExpressContext

```
public class ExpressContext
        extends HashMap<String,Object>
        implements IExpressContext<String,Object> {

    private ApplicationContext applicationContext;

    Public ExpressContext(Map<String,Object> aProperties)
    {
        super(aProperties);
    }

    /**
     * 根据 key 从容器里面获取对象
     *
     * @param key
     * @return
     */
    public Object get(Object key) {
        Object object = super.get(key);
        try {
            if (object == null && this.applicationContext != null
            && this.applicationContext.containsBean((String)key))
            {
                object = this.applicationContext.getBean((String)key);
            }
        } catch (Exception e) {
            throw new RuntimeException("表达式容器获取对象失败", e);
        }
        return object;
    }

    /**
     * 把 key-value 放到容器里面去
     *
     * @param key
     * @param value
     */
    public Object put(String key, Object value) {
        return super.put(key, value);
    }

}
```

流程实现：

```
@Resource
Private MyExpressExecutor myExpressExecutor;
@Resource
Private ServiceDAO serviceDAO;

Public bool canBuyTheService(Map<String,Object>context,String serviceCode){
    String express = serviceDAO.getService(serviceCode).getExpress();
    ExpressContext expressContext = new ExpressContext(context);
    myExpressExecutor.coreExecute(expressContext,express);
}
```

注：@Resource 为 spring 的 bean 注解

不得不说的几个 nice

1、运营和开发人员的职责分离

a) 运营人员：只关注底层提供的 API 映射够不够
b) 技术人员：只要适时增加新的 API 映射，随便他们怎么玩

2、类似这个 bool 表达式，还有智能的逻辑短路功能哦：

!（天猫卖家 or 良无限卖家）and !（一钻以下卖家）

如果是天猫卖家=true，马上可以判断整个语句为 false，不会浪费不必要的系统远程调用。

性能评估

在此感谢我们的淘宝内部业务方 @逊尧 提供的资料。【所在系统同时支持两种 groovy 和 QlExpress 脚本语言，上线前做了性能测试】

Groovy 版本：

```
import trade.model.OrderModel

OrderModel orderModel = binding.getProperty("orderModel");

商品包含标 = { int a ->
    return orderModel.getItemModel().itemTags != null && orderModel.getItemModel().itemTags.contains(a)
}

商品特征包含={String key,Object value->
    return  orderModel.itemModel.features!=null && orderModel.itemModel.features.get(key).equals(value);
}

if (orderModel.bizType == 2100)
    return 5;

if (商品包含标(3) || 商品包含标(5))
    return 6;

if (商品包含标(4) && orderModel.itemModel.features != null && 商品特征包含("a",4))
    return 8;

if (商品包含标(4) && orderModel.itemModel.features != null && 商品特征包含("b",5))
    return 8.5;

if (orderModel.attributes.t == "4" && 商品包含标(4))
    return 9l;

return 10;
```

QlExpress 版本

```
alias 商品 订单.itemModel;

function 商品包含标(int a){
  return 商品.itemTags.contains(a);
};

function 商品特征包含(String key,Object value){
  return  商品.features!=null and 商品.features.get(key)==value;
}

if(订单.bizType==2100)
   return 5;

if(商品包含标(3) || 商品包含标(5))
   return 6;

if(商品包含标(4) and 商品特征包含("a",4) )
   return 8;

if(商品包含标(4) && 商品特征包含("b",5) )
   return 8.5;

if(订单.attributes.t=="4" && 商品包含标(4))
   return 9l;

   return 10;
```

分别运行以上 100w 次：

QLExpress: 18931ms，groovy: 10421 ms。

性能上，在这个涉及到复杂的方法定义，类型判断，逻辑条件判断，类路径设置的场景中：

groovy 是 QlExpress 的 1.8 倍左右。Groovy 的缓存是设置在内存的方法区，QlExpress 缓存在普通堆区中。

QlExpress 目前还没有发现成为业务系统瓶颈的案例。