金蝶软件（中国）有限公司

插件开发指南

金蝶云苍穹

修订版历史

日期	版本	说明	作者	审阅
2017-12-06	1.0	初稿	丁振华	
2019-8-22	1.0.1	增加打印插件的章节	于成刚	
2020-12-30			金蝶云苍穹伙伴产品赋能部	

编写指南

- ［版本］刚开始时为1.0版本，［说明］统一为‘初稿’
- 版本号逐步增加，不要跳跃
- 经过一轮审阅（或检查）后进行的一次修订，才认为是一个新的revision（小版本增加）
- Version（大版本号）的递增，通常是以一个软件项目周期为单位
- ［说明］最好逐项分列，并指明‘新增’‘修订’‘删除’

目录

1. 概述

1.1. 为什么要开发插件

使用应用开发平台的设计器开发业务对象，全程配置，简单易学，但不够灵活。

配置业务对象时使用的业务语意，需要预先定义，遇到未考虑的业务场景，则无法处理。

业务对象设计器，目标是实现常见的 80%业务语意，利用开放的插件开发，实现剩余 20%的功能。

1.2. 插件如何工作？

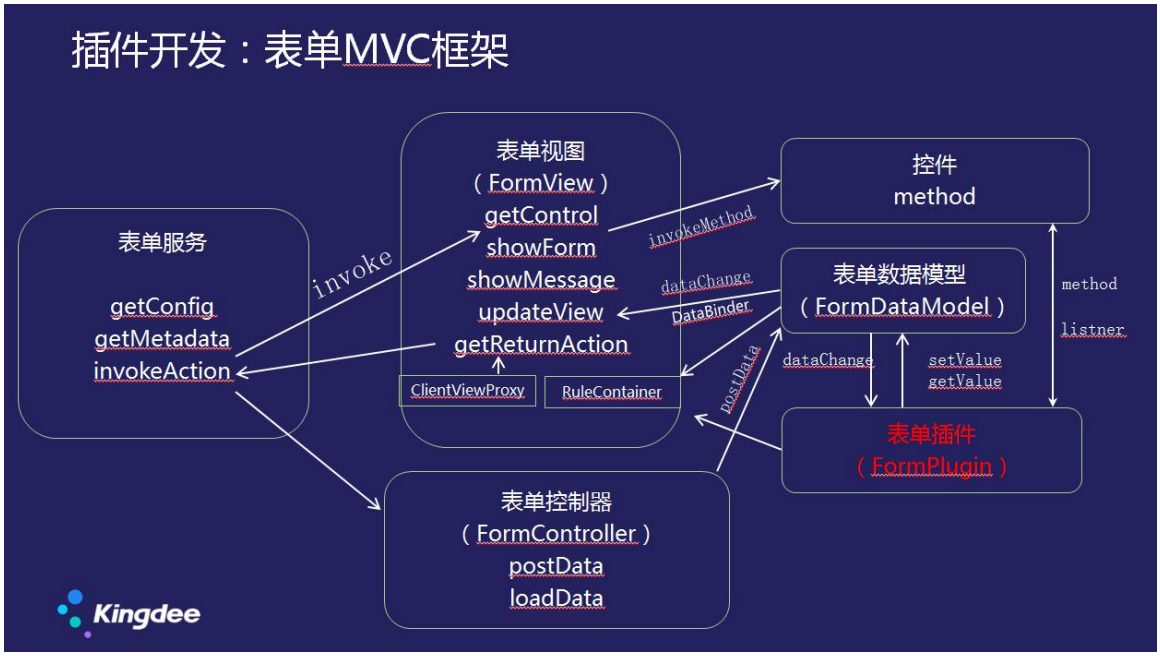
插件可以在适当的时机，根据接收到的上下文信息，对系统功能进行控制。

适当的时机，是指插件只会在系统功能运行到了特定时刻，才能收到系统通知，进行功能处理；并不能对系统功能的全过程进行干预。

插件拿到的上下文信息，能够调用的控制方法，也是经过系统封装，有所限制；

简单的说，插件可以在系统约束的框架之内，对系统进行适度的干预，在灵活性与安全性之间有个平衡。

如下图，表单界面在加载、展示期间，关键功能执行时，调用插件接口方法，触发插件事件，通知插件工作；
而插件能够在被调用的事件方法中，调用表单视图、控件模型、表单数据模型，对表单界面进行控制。



1.3. 如何开发插件

1.3.1. 步骤一：确定应用场景，选择插件基类

不同的业务对象类型，提供特定的业务功能，有适用的应用场景。

系统为各种业务对象类型、各种应用场景，封装了相应的插件接口、事件方法，定义了抽象插件基类，实现最基本的插件接口。

进行业务功能设计时，首先需要根据业务需求特点，分析业务应用场景，选择业务对象类型；

如果需要进行插件开发，则根据前面确定好的业务对象类型及应用场景，从下表中选择对应插件基类进行扩展：

业务对象类型	应用场景	预置的 Java 插件基类
动态表单	PC 端界面	AbstractFormPlugin
	移动端界面	AbstractMobFormPlugin
单据	PC 端界面	AbstractBillPlugIn
基础资料	移动端界面	AbstractMobBillPlugIn
	单据列表	AbstractListPlugin
	左树右表单据列表	AbstractTreeListPlugin
	树形基础资料列表	StandardTreeListPlugin
	移动端单据列表	AbstractMobListPlugin

	业务操作	AbstractOperationServicePlugIn
	单据转换	AbstractConvertPlugIn
	关联反写	AbstractWriteBackPlugIn
	生成凭证	AbstractBuildVchPlugin
报表	界面	暂缺
	取数	暂缺

1.3.2. 步骤二：确定事件源与控件

有交互界面的应用场景（如表单、单据列表等），在界面加载、关闭时，会触发相应的插件事件；

另外，用户与界面，以及界面上的控件交互时，也会触发插件事件。

各种控件有自己的功能特点，适用不同的业务需求，提供相应的插件事件；

在进行功能设计时，需要根据业务需求，选用合适的控件，确定需要重写的插件接口方法，响应插件事件。

后面的控件与字段章节，将详细列出各种控件的功能特定及支持的插件事件，供您参考。

➢ 特别说明：
没有交互界面的应用场景（如单据操作、单据转换、关联反写、生产凭证等），不会与用户发生交互，其插件事件是由服务引擎按顺序触发的。

这些应用场景，不需要关注事件源、控件；只需要根据业务需求，捕获合适的插件事件即可。

1.3.3. 步骤三：响应插件事件

系统封装了各种插件事件接口；
表单、控件、字段等事件源，各自有选择的支持了部分插件事件接口。

系统会在适当的时机，调用插件事件接口的方法，传入上下文参数，触发插件事件。
比如用户与前端界面、控件交互时，系统会把交互请求传递给表单、控件；表单、控件调用其下的插件事件接口实例（即插件）的方法，触发插件事件。

不同的插件事件，触发的时机、传入的上下文参数、可以控制的功能，差异非常大。

因此，必须根据业务需求，选择合适的插件事件响应。

后文会详细介绍各种应用场景、字段、控件，提供的插件事件，及其触发时机、事件参数。

本节介绍业务插件如何响应（捕捉）插件事件：

无交互界面的场景

没有交互界面的应用场景，插件基类已经实现了必要的插件接口，插件只需要扩展插件基类，重写事件方法即可完成事件的捕捉：

Java

```
package kd.bos.metadata.botp;

import kd.bos.entity.plugin.AbstractOperationServicePlugIn;
import kd.bos.entity.plugin.args.BeforeOperationArgs;

public class WriteBackRuleOpPlug extends AbstractOperationServicePlugIn {
    @Override
    public void beforeExecuteOperationTransaction(BeforeOperationArgs e) {
        // TODO : 在此添加业务逻辑
    }
}
```

说明：插件基类 AbstractOperationServicePlugIn 已经实现业务操作插件接口，插件直接重写需要响应的事件方法即可。

有交互界面的场景

有交互界面的应用场景，插件基类实现了表单界面支持的插件接口，但没有实现各种控件的插件接口。

如果只是要捕获表单界面事件，则扩展插件基类，重写表单事件方法即可。

如果要捕获各种控件事件，则需要：

- 在插件类定义，实现控件支持的插件事件接口，例如树形控件支持的节点勾选插件事件接口 TreeNodeCheckListener；
- 实现控件的插件事件接口中的方法，完成事件的捕捉；
- 在表单界面插件 registerListener 事件，向控件实例注册本插件实例，完成控件与插件实例的绑定：控件事件发生时，即触发其所绑定的插件事件；

如下例，插件需要响应树形控件的节点勾选 treeNodeCheck 事件：

Java

```
package kd.bos.plugin.sample.dynamicform.pcform.control.template;

import java.util.EventObject;
```

```
import java.util.List;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.control.TreeView;
import kd.bos.form.control.events.TreeNodeCheckEvent;
import kd.bos.form.control.events.TreeNodeCheckListener;
import kd.bos.form.plugin.AbstractFormPlugin;

public class TreeViewTreeNodeCheck extends AbstractFormPlugin implements TreeNodeCheckListener
{

    private final static String KEY_TREEVIEW1 = "treeviewap1";

    @Override
    public void registerListener(EventObject e) {
        super.registerListener(e);

        // 侦听树节点勾选事件
        TreeView treeView = this.getView().getControl(KEY_TREEVIEW1);
        treeView.addTreeNodeCheckListener(this);
    }

    @Override
    public void beforeBindData(EventObject e) {
        super.beforeBindData(e);
        TreeView treeView = this.getView().getControl(KEY_TREEVIEW1);
        treeView.setMulti(true);  // 支持多选
    }

    @Override
    public void treeNodeCheck(TreeNodeCheckEvent arg0) {
        TreeView treeView = (TreeView) arg0.getSource();
        if (StringUtils.equals(treeView.getKey(), KEY_TREEVIEW1)){
            List<String> selectNodeIds = treeView.getTreeState().getCheckedNodeIds();
            // TODO 在此添加业务逻辑
        }
    }
}
```

说明：插件基类 AbstractFormPlugin 没有实现树形控件的节点点击事件接口 TreeNodeCheckListener，插件需要自行实现。

1.4.什么是表单视图模型、控件编程模型？

下一代金蝶云，是 B/S 结构的，使用不同的客户端（PC 端、移动端），通过网络连接到统一的服务端。

用户看到的交互界面，是运行在客户端的，而业务逻辑和业务插件，则运行在服务端；

业务插件运行在服务端，没办法直接获取到客户端界面上控件句柄，不能直接控制前端控件。

但插件可以通过系统封装的视图模型接口 IFormView，间接的访问、控制前端界面；
通过系统封装的各种控件代理对象，间接的访问、控制前端界面上的控件；

这些运行在服务端的控件代理对象，称为控件编程模型，或简称为控件、编程模型等。

插件可以通过 this.getView()方法，获取表单的视图模型接口实例；
可以通过 this.getView().getControl(String key)方法，获取到控件编程模型实例；

1.5.什么是表单数据模型

表单的界面与数据，是分离：

- 界面显示在客户端浏览器或者移动端，在服务端，系统封装了视图模型及控件编程模型来间接控制前端界面及前端控件；
- 数据存储在服务端，在服务端，系统封装了数据模型，来控制界面上的数据；

表单的数据模型，提供各种方法访问界面数据，并且持有：

- 主实体模型：MainEntityType，表单运行时元数据对象，包含：
  - 子实体：EntityType，对应单据体、子单据体等；
  - 属性：DynamicProperty，对应字段；
- 界面数据包：DynamicObject，基于主实体模型构建的一个数据字典，存储单据体、字段值；

1.6. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=01 进行反馈。

2. 动态表单插件

动态表单是最基本的交互界面，移动端表单、单据、基础资料、报表，都是基于动态表单，表单

的视图模型、数据模型、插件模型等，会被这些业务对象继承扩展。

动态表单不与物理表格直接关联，只显示内存中的数据，可以用于：
1. 显示交互信息；
2. 显示其他界面加工后传入的数据；
3. 显示来自于数据库，但需要经过加工的数据；
4. 纯交互界面；
5. 其他；

动态表单相对于单据，数据来源更加的灵活，用途更加广泛。

2.1.插件基类

动态表单插件，必须从插件基类 AbstractFormPlugin 中派生。

2.1.1. 插件基类定义

动态表单插件基类 AbstractFormPlugin，已经实现了动态表单界面插件事件接口 IFormPlugin，以及表单数据事件接口 IDataModelListener，自定义的动态表单界面插件，必须派生自此类。

插件基类定义：

Java

```
package kd.bos.form.plugin;
public class AbstractFormPlugin extends AbstractDataModelPlugin implements IFormPlugin {
```

动态表单插件基类，特别封装了如下方法，辅助插件完成业务功能：

方法	说明
getView	获取动态表单界面视图模型接口 IFormView 的实例
getModel	获取动态表单界面数据模型接口 IDataModel 的实例
getPageCache	获取页面缓存管理器，存取数据到缓存中
getControl	获取界面上的控件
addClickListeners	注册插件，监听按钮点击事件
addItemClickListeners	注册插件，监听子菜单项点击事件

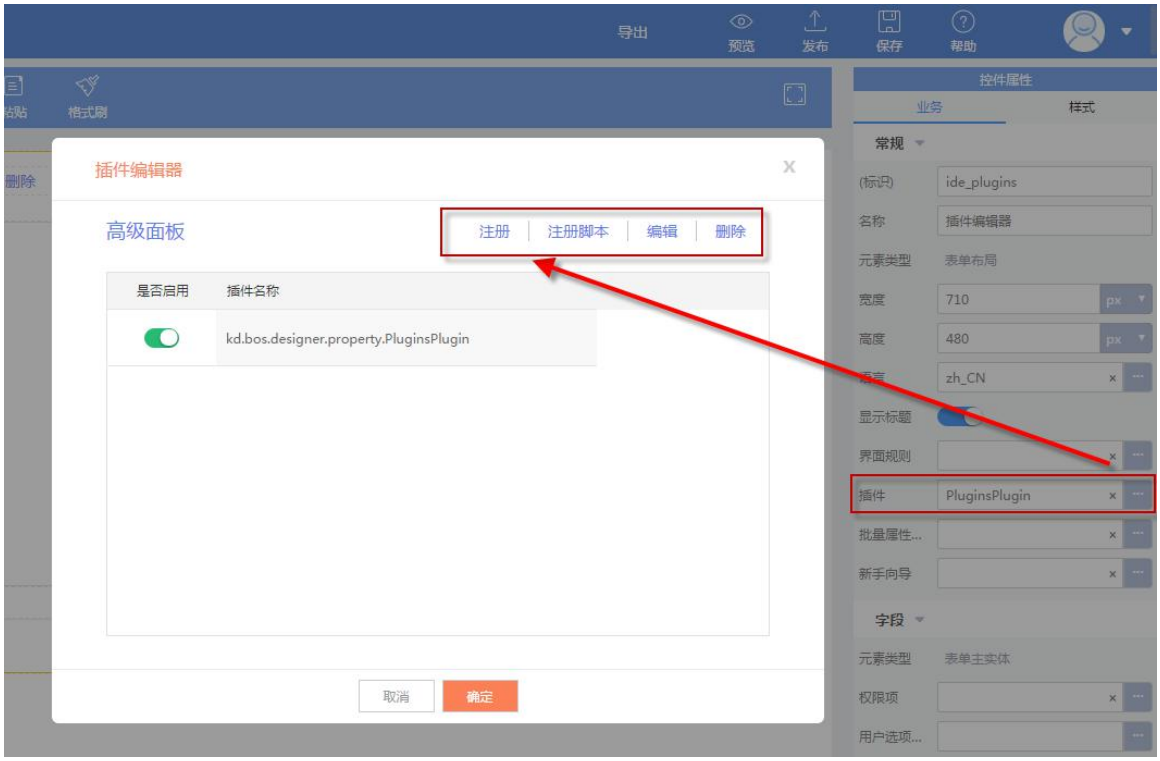
2.1.2. 创建并注册插件

自定义动态表单界面插件，必须派生自 AbstractFormPlugin，重写事件处理方法。
如下例，定义了一个动态表单界面插件（未捕获事件）：

Java

```
package kd.bos.plugin.sample.dynamicform.pcform.form.bizcase;

import kd.bos.entity.datamodel.IDataModel;
import kd.bos.form.IFormView;
import kd.bos.form.plugin.AbstractFormPlugin;

public class FormViewSample extends AbstractFormPlugin {

    private void getFormViewInstance(){
        IFormView view = this.getView();
        IDataModel model = this.getModel();
    }
}
```

开发好的表单插件，通过表单设计器、插件注册，与动态表单绑定：



2.2. 视图模型

动态表单界面插件，可以通过系统封装的表单视图模型，访问界面信息，对界面进行控制。

2.2.1. 接口与实现类

动态表单的视图模型接口为 IFormView，定义了各种界面控制方法：

```Java
package kd.bos.form;
public interface IFormView {
```

动态表单的视图模型实现类为 FormView，实现了接口 IFormView：

```Java
package kd.bos.mvc.form;
public class FormView extends AbstractFormView {
```

```Java
package kd.bos.form;
public abstract class AbstractFormView implements IFormView {
```

插件可以使用如下代码，获取动态表单界面视图模型实例：

```Java
IFormView view = this.getView();
```

2.2.2. 功能方法及使用

动态表单界面视图模型 IFormView 接口，定义了很多方法，下表列出插件需要用到的部分方法：

方法	说明
getPageId	表单被加载时，会随机生成一个界面 PageId； 一个表单被两个用户同时打开时，生成的界面 PageId 不同； 可以根据返回的 pageId，获取到表单视图模型实例；
getView	指定 PageId，获取对应的表单视图模型； 可以据此对目标表单进行控制；
getEntityId	获取表单对应的主实体标识
getModel	获取表单数据模型实例
sendFormAction	把目标表单的控制指令发送给前端； 插件调用了其他表单的控制方法后，必须调用本方法，把控制指令，发送给前端；
getParentView	获取父表单视图模型实例；

getMainView	获取主界面视图模型实例；
updateView	把数据模型中的数据，发送到前端界面； 定义了多个重载函数，可以指定只刷新单个单据体、单个控件
getControl	获取表单的控件实例
getRootControl	获取表单实例
getService	获取服务实例
invokeOperation	执行操作
activate	激活表单
close	关闭表单
setEnable	设置控件可用性
setVisible	设置控件可见性
showForm	传入表单显示参数，打开一个新的表单，作为本表单的子表单； 该表单关闭时，会触发本表单插件 closedCallBack 事件； 请参阅表单 closedCallBack 事件说明及示例；
getFormShowParameter	获取表单显示参数
cacheFormShowParameter	修改表单显示参数对象属性值之后，调用本方法把参数更新到缓存
returnDataToParent	设置返回到父表单的返回值
openUrl	打开一个新窗口链接到指定的 URL
showUpload	显示一个文件上传界面； 文件上传完毕，确认返回时，会触发插件 afterUpload 事件； 请参阅按钮 afterUpload 事件说明及示例；
showMessage	单据内悬浮消息框，默认没有按钮，自动消失
showErrMessage	显示错误消息
showOperationResult	显示操作结果
showConfirm	显示确认消息； 用户确认完毕，会触发 confirmCallBack 事件； 请参阅 confirmCallBack 事件说明及实例
showSuccessNotification	单据内成功悬浮消息框，默认 2 秒自动消失 消息内容,不能超过 50 字,超过部分用三个点代替
showErrorNotification	单据内失败悬浮消息框，需要手动关闭 消息内容,不能超过 50 字,超过部分用三个点代替
showTipNotification	单据内提示类别悬浮消息框，提示类会显示按钮，需要手动关闭 消息内容,不能超过 50 字,超过部分用三个点代替
showRobotMessage	发送消息给机器人助手
closeRobotMessage	关闭消息给机器人助手
showFieldTip	字段上显示提示信息
showFieldTips	字段上显示提示信息（批量）

2.3.数据模型

动态表单界面插件可以通过系统封装的表单数据模型，访问界面数据。

2.3.1. 接口与实现类

动态表单界面数据模型接口为 IDataModel，提供了各种控制表单数据的方法：

```Java
package kd.bos.entity.datamodel;
public interface IDataModel extends ISupportInitialize, IEntryOperate, IDataProvider {
```

动态表单的数据模型实现类为 FormDataModel，实现了接口 IDataModel 的方法：

```Java
package kd.bos.mvc.form;
public class FormDataModel extends AbstractFormDataModel{
```

```Java
package kd.bos.entity.datamodel;
public abstract class AbstractFormDataModel implements IDataModel, IRefrencedataProvider {
```

插件，可以通过如下代码，获取到动态表单界面的数据模型实例：

```Java
IDataModel model = this.getModel();
```

2.3.2. 功能方法及使用

动态表单数据模型接口 IDataModel，定义了很多方法，下表列出插件需要用到的部分方法：

方法	说明
getDataEntityType	获取运行时表单实体元数据对象，又称为主实体模型； 通过表单主实体模型，可以或者界面上包含了那些单据体、字段
getProperty	获取运行时字段元数据对象，又称为实体的属性对象
createNewData	根据表单主实体模型，创建表单新的数据包，字段填写好默认值
getDataEntity	获取表单数据包
updateCache	提交当前表单数据包到缓存
getValue	获取字段值
setValue	设置字段值
setItemValueByNumber	根据基础资料的编码，设置基础资料字段值

setItemValueByID	根据基础资料的内码，设置基础资料字段值
getContextVariable	获取上下文变量
putContextVariable	添加上下文变量
removeContextVariable	删除上下文变量
addDataModelListener	订阅模型相关事件
addDataModelChangeListener	订阅模型改变事件

2.3.3. 主实体模型

动态表单设计完毕，系统会根据表单字段结构，为表单创建一个运行时元数据对象，又称为表单主实体模型 MainEntityType：

- 表单上的单据体、字段，都会转为属性对象 DynamicProperty，分为如下三类：
  - ✓ 简单值属性：SimpleProperty，对应普通字段；
  - ✓ 复杂值属性：ComplexProperty，对应基础资料字段，关联基础资料主实体 RefEntityType，嵌套包含基础资料属性；
  - ✓ 集合值属性：CollectionProperty，对应单据体等分录，关联到分录子实体 EntryType，嵌套包含分录属性；
- 各属性对象之间，会按照表单设计，具有层级从属关系；
  - ✓ 复杂值、集合值属性对象，会关联子实体，继续包含下层属性对象（字段、子单据体等）

主实体模型会被缓存，请勿在插件中直接修改主实体模型内容，以免串账；业务需求必须动态修改主实体模型时，只能修改复制品；

主实体模型 MainEntityType 部分常用方法如下：

方法	说明
getAllEntities	全部子实体
getAllFields	全部字段，不包括系统自动注册的属性对象，如主键
getMainOrg	主业务组织标识，可能为 null
getAppId	业务应用 Id
getPermissionControlType	功能权限控制配置

MainEntityType 的超类 EntityType 提供的常用方法如下：

方法	说明
findProperty	查找属性，会遍历本实体以及子实体

EntityType 的超类 DynamicType 提供的常用方法如下：

方法	说明
getName	实体的标识

getDisplayName	实体的标题
getDBRouteKey	分库标识
getPrimaryKey	主键 动态表单，与物理表格无关，没有主键
getProperties	本实体的属性对象集合，不包括子实体的属性
getAlias	物理表格名称 动态表单不与物理表格关联，此属性为空； 单据、基础资料的主实体，此属性存储单据表格名
isDbIgnore	是否关联物理表格
registerSimpleProperty	注册新的简单值属性，如文本、数值、日期等
registerComplexProperty	注册新的复杂值属性，如基础资料
registerCollectionProperty	注册新的集合值属性，如单据体
createInstance	基于本实体模型，创建空白的数据包(DynamicObject)

下例简单的演示了如何使用 MainEntityType 的方法（演示代码没有实际的业务意义，仅供参考）：

Java

```
package kd.bos.plugin.sample.dynamicform.pcform.form.bizcase;

import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.dataentity.entity.LocaleString;
import kd.bos.dataentity.metadata.IDataEntityProperty;
import kd.bos.entity.MainEntityType;
import kd.bos.form.plugin.AbstractFormPlugin;

public class MainEntityTypeSample extends AbstractFormPlugin {

    private void useMainEntityType(){

        // 获取当前表单的主实体模型
        MainEntityType mainEntityType = this.getModel().getDataEntityType();

        // 基于表单主实体模型，创建空的数据包（多种方法）
        DynamicObject dataEntity1 = new DynamicObject(mainEntityType);
        DynamicObject dataEntity2 = (DynamicObject)mainEntityType.createInstance();

        // 获取主实体部分属性值
        String entityNumber = mainEntityType.getName();         // 实体标识
        LocaleString entityCaption = mainEntityType.getDisplayName();   // 标题，支持多语言
        String tableName = mainEntityType.getAlias();  // 物理表格
        boolean isDbIgnore = mainEntityType.isDbIgnore();  // 有没有关联物理表格
        String dbRouteKey = mainEntityType.getDBRouteKey();     // 分库标识
        String bizAppId = mainEntityType.getAppId();   // 业务应用标识
```

```
        // 获取单据头上的字段属性（多种方法）
        IDataEntityProperty billNoProp1 = mainEntityType.getProperties().get("billno");
        IDataEntityProperty billNoProp2 = mainEntityType.getProperty("billno");
        IDataEntityProperty billNOProp3 = mainEntityType.findProperty("billno");

        // 获取单据体上的字段属性（多种方法）
        IDataEntityProperty bdProp1 =
mainEntityType.getAllEntities().get("entryentity").getProperty("basedatafield1");
        IDataEntityProperty bdProp2 = mainEntityType.findProperty("basedatafield1");
    }
}
```

2.3.4. 数据包

表单数据包，DynamicObject 类型，实际是个数据字典：严格按照表单主实体模型的结构，构建数据字典，存储各个属性的值。

表单主实体模型有三种属性类型，在数据包中，有对应的属性值类型：

- 简单值：存储文本、数值、日期、布尔值；
- 复杂值：存储引用的基础资料数据包，也是 DynamicObject 类型；
- 集合值：存储数据包集合，DynamicObjectCollection 类型，即 DynamicObject 类型集合；

下例演示逐层递归读取 DynamicObject 对象中存储的全部属性的值：

Java

```
package kd.bos.plugin.sample.dynamicform.pcform.form.bizcase;

import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.dataentity.entity.DynamicObjectCollection;
import kd.bos.dataentity.metadata.IDataEntityProperty;
import kd.bos.dataentity.metadata.IDataEntityType;
import kd.bos.dataentity.metadata.clr.CollectionProperty;
import kd.bos.dataentity.metadata.clr.ComplexProperty;
import kd.bos.form.plugin.AbstractFormPlugin;

public class DynamicObjectSample extends AbstractFormPlugin {

    private void useDynamicObject(){

        // 当前表单数据包
```

```
        DynamicObject dataEntity = this.getModel().getDataEntity(true);

        // 数据包是否从数据库加载，还是全新创建的？
        boolean isFromDB = dataEntity.getDataEntityState().getFromDatabase();

        // 读取表单数据包中全部的字段值
        this.readPropValue(dataEntity);
    }

    /**
     * 递归读取数据包中全部属性的值
     *
     * @param dataEntity
     */
    private void readPropValue(DynamicObject dataEntity){

        // 获取数据包对应的实体模型
        IDataEntityType dType = dataEntity.getDataEntityType();

        for(IDataEntityProperty property : dType.getProperties()){
            if (property instanceof CollectionProperty){
                // 集合属性，关联子实体，值是数据包集合
                DynamicObjectCollection rows =
dataEntity.getDynamicObjectCollection(property);
                // 递归读取子实体各行的属性值
                for (DynamicObject row : rows){
                    this.readPropValue(row);
                }
            }
            else if (property instanceof ComplexProperty){
                // 复杂属性，关联引用的基础资料，值是另外一个数据包
                DynamicObject refDataEntity = dataEntity.getDynamicObject(property);
                // 递归读取引用的基础资料属性值
                if (refDataEntity != null){
                    this.readPropValue(refDataEntity);
                }
            }
            else {
                // 简单属性，对应普通的字段，值是文本、数值、日期、布尔等
                Object propValue = dataEntity.get(property);

                // 输出" 属性名 = 属性值 "
                String msg = String.format("%s = %s", property.getName(), propValue);
                System.out.println(msg);
```

```
            }
        }
    }
}
```

2.4.表单插件事件

动态表单界面事件如下（按触发先后顺序列出）：

分类	事件	触发时机
界面显示前	setPluginName	显示界面，准备构建界面显示配置 formConfig 前，构建插件时触发此事件，传入脚本名称；
	preOpenForm	显示界面前，准备构建界面显示参数时，触发此事件；
	loadCustomControlMetas	显示界面前，构建界面显示参数时，触发此事件；
界面初始化	setView	表单视图模型初始化，创建插件时，调用此方法，向插件传入表单视图模型 IFormView 实例；
	initialize	表单视图模型初始化，创建插件后，触发此事件；
	registerListener	用户与界面上的控件交互时，触发此事件；
	getEntityType	表单基于实体模型，创建数据包之前，触发此事件；
	createNewData	界面初始化或刷新，开始新建数据包时触发此事件；
	afterCreateNewData	界面初始化或刷新，新建数据包完毕后，触发此事件
	beforeBindData	界面数据包构建完毕，开始生成指令，刷新前端字段值、控件状态之前，触发此事件；
	afterBindData	界面数据包构建完毕，生成指令，刷新前端字段值、控件状态之后，触发此事件；
用户交互事件	beforeItemClick	用户点击界面菜单按钮时，执行绑定的操作前，触发此事件；
	itemClick	用户点击界面菜单按钮时触发此事件；
	beforeDoOperation	用户点击按钮、菜单，执行绑定的操作前，触发此事件；
	afterDoOperation	用户点击按钮、菜单，执行完绑定的操作后，不论成功与否，均会触发此事件；
	confirmCallBack	前端交互提示确认后，通知插件进行后续处理；
	closedCallBack	子界面关闭时，如果回调函数由父界面处理，则会触发父界面的此事件；
	flexBeforeClosed	弹性域维护界面关闭时，触发父界面此事件；
	onGetControl	在有代码尝试获取控件的编程模型时，触发此事件；

	customEvent	触发自定义控件的定制事件；
	TimerElapsed	定时触发此事件；
界面关闭	beforeClosed	界面关闭之前触发此事件；
	destory	界面关闭后，释放资源时，触发此事件
	pageRelease	界面关闭后，释放资源时，触发此事件；
	onCreateDynamicUIMetas	动态创建 UI 元数据
	contextMenuClick	菜单点击事件

2.4.1. setPluginName 事件

2.4.1.1. 事件触发时机

界面显示前，准备构建界面显示参数时，系统会先创建界面插件实例。

在构建 JS 插件时，触发此事件，传入脚本名称；
这个事件在插件刚刚构建之后即会触发，是插件能接受到的第一个事件。

只有 JS 代码插件，会触发这个事件；Java 代码插件，不触发。

2.4.1.2. 代码模板

Java

```java
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import kd.bos.form.plugin.AbstractFormPlugin;

public class SetPluginName extends AbstractFormPlugin {
    String plugName;

    @Override
    public void setPluginName(String name) {
        this.plugName = name;
    }

    @Override
    public String getPluginName() {
        return this.plugName;
```

```
    }
}
```

2.4.1.3. 参数说明

- String name：脚本插件标识

2.4.1.4. 应用示例

这个插件事件触发的时机太早，没有什么实际用处，无需捕获。

示例略。

2.4.2. preOpenForm 事件

2.4.2.1. 事件触发时机

系统收到 showForm 指令，显示界面前，准备构建界面显示参数时，触发此事件。

此事件触发时，新界面还没有显示出来，可以在此事件，取消界面的显示，或者修改显示参数。

2.4.2.2. 代码模板

Java

```java
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import kd.bos.form.events.PreOpenFormEventArgs;
import kd.bos.form.plugin.AbstractFormPlugin;

public class PreOpenForm extends AbstractFormPlugin {

    @Override
    public void preOpenForm(PreOpenFormEventArgs e) {
```

```
        super.preOpenForm(e);
        // TODO : 可以在此取消界面显示
    }
}
```

2.4.2.3. 参数说明

- PreOpenFosrmEventArgs e：事件参数对象，定义如下：
  - ✓ Object getSource()：界面显示参数 FormShowParameter 对象，可以通过调整此对象属性值，控制界面显示
  - ✓ void setCancel(boolean cancel) ：取消界面显示
  - ✓ void setCancelMessage(String cancelMessage)：cancelMessage 取消原因，提示用户

2.4.2.4. 应用示例

- 案例说明

1. 自定义界面标题
2. 自行校验界面查看权，如果未授权，不允许显示

- 实现方案

1. 捕获界面显示 preOpenForm 事件
   a. 修改显示参数，调整界面标题
   b. 校验权限，如果无权，取消界面显示
2. 上述场景仅用于演示，实际验权由系统自动处理，不需要插件验权。

- 运行效果

图一：显示效果，界面标题被修改

图二：如果没有权限，则界面不会打开，直接提示无权

➢ 实例代码

Java

```
package kd.bos.plugin.sample.dynamicform.pcform.form.bizcase;

import kd.bos.context.RequestContext;
import kd.bos.form.FormShowParameter;
import kd.bos.form.events.PreOpenFormEventArgs;
import kd.bos.form.plugin.AbstractFormPlugin;
import kd.bos.servicehelp.permission.PermissionServiceHelper;
import kd.bos.servicehelper.model.PermissionStatus;

public class PreOpenFormSample extends AbstractFormPlugin {

    @Override
```

```
    public void preOpenForm(PreOpenFormEventArgs e) {

        // 设置显示参数 - 是否触发TimerElapsed事件
        FormShowParameter showParameter = (FormShowParameter)e.getSource();
        showParameter.setListentimerElapsed(true);

        // 设置显示参数 - 界面标题
        showParameter.setCaption("hello world");

        // 检查用户是否获得授权，如未授权，撤销界面显示
        if (!this.checkFunctionPermission(showParameter.getFormId(), PermissionStatus.View)){
            e.setCancel(true);
            e.setCancelMessage("对不起，您无权访问此页面！");
        }
    }

    private boolean checkFunctionPermission(String formId, String permissionItemId){
        long userId = Long.parseLong(RequestContext.get().getUserId());
        int result = PermissionServiceHelper.checkFunctionPermission(
                userId,
                RequestContext.get().getOrgId(),
                formId,
                permissionItemId);

        return result == 1;
    }
}
```

2.4.3. loadCustomControlMetas 事件

2.4.3.1. 事件触发时机

显示界面前，构建界面显示参数时，触发此事件；

插件可以在此事件修改显示参数，向前端动态增加控件。

特别说明：
动态添加控件时，还需要同步处理 onGetControl 事件，向表单添加控件编程模型实例，并侦听控

件的插件事件；

动态增加字段，则还需要同步处理 getEntityType, createNewData, beforeBindData 事件，调整后台数据模型及数据包。

需同步处理的事件：

- `public void loadCustomControlMetas(LoadCustomControlMetasArgs e)` ：向前端浏览器界面输出动态控件的元数据；
- `public void onGetControl(OnGetControlArgs e)`：向后台视图模型，输出动态控件的编程模型实例，并侦听控件的事件
- `public void getEntityType(GetEntityTypeEventArgs e)`：向界面主实体模型，动态注册新的属性对象，存储动态添加的字段值；
- `public void createNewData(BizDataEventArgs e)`：基于修改后的实体模型，创建界面数据包，包含动态添加的字段值；
- `public void beforeBindData(EventObject e)`：向后台视图模型，添加动态字段的控件编辑模型；随后，系统将基于这些新添加的字段编辑模型(FieldEdit)，进行数据绑定。如果不处理这个事件，动态添加的字段值，显示不出来。

2.4.3.2. 代码模板

Java

```java
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import java.util.EventObject;

import kd.bos.entity.datamodel.events.BizDataEventArgs;
import kd.bos.entity.datamodel.events.GetEntityTypeEventArgs;
import kd.bos.form.events.LoadCustomControlMetasArgs;
import kd.bos.form.plugin.AbstractFormPlugin;

public class LoadCustomControlMetas extends AbstractFormPlugin {

    @Override
    public void loadCustomControlMetas(LoadCustomControlMetasArgs e) {
        super.loadCustomControlMetas(e);
        // TODO 在此修改显示参数，向前端动态添加控件配置
    }

    @Override
    public void getEntityType(GetEntityTypeEventArgs e) {
        // TODO 在此修改实体模型，增加属性
    }
```

```
    @Override
    public void createNewData(BizDataEventArgs e) {
        // TODO 在此基于调整后的实体模型，创建界面数据包
    }

    @Override
    public void beforeBindData(EventObject e) {
        // TODO 在此向界面编程模型，动态添加字段编程模型(FieldEdit)
    }

    @Override
    public void onGetControl(OnGetControlArgs e) {
        // TODO 在此创建动态控件的实例，并捕获控件事件
    }
}
```

2.4.3.3. 参数说明

- LoadCustomControlMetasArgs e
  - ✓ Object getSource()：FormShowParameter，显示参数对象，包含了待显示界面的标识
  - ✓ List<Map<String, Object>> getItems()：动态添加到前端的控件配置集合；

2.4.3.4. 应用示例

- 案例说明

1. 在界面加载时，向界面添加字段与控件：
   a. 向单据头添加一个新的文本字段 autotext1，字段值由插件填写；
   b. 向单据体添加一个新的文本字段 autotext2，字段值由插件填写；
   c. 添加一个新的按钮 autobutton1；
2. 用户点击动态添加的按钮时，显示提示信息

- 实现方案

1. 在界面上，增加一个容器面板：myfieldcontainer，作为容器，包含自动添加的字段、控件
2. 捕获 loadCustomControlMetas 事件，向容器控件添加新的字段、控件
3. 捕获 getEntityType 事件，向主实体，注册添加的文本属性
4. 捕获 createNewData 事件，自行构建界面数据包
5. 捕获 beforeBindData 事件，输出自定义字段值到前端界面
6. 捕获 onGetControl 事件，输出动态添加的字段、按钮的控件编程模型

7. 捕获 click 事件，响应动态按钮的点击事件

- 运行效果

图一：界面加载完的效果

图二：点击动态添加的按钮，弹出的提示信息

➢ 实例代码

Java

```
package kd.bos.plugin.sample.dynamicform.pcform.form.bizcase;

import java.util.EventObject;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.dataentity.entity.DynamicObjectCollection;
import kd.bos.dataentity.entity.LocaleString;
import kd.bos.dataentity.utils.StringUtils;
import kd.bos.entity.EntryType;
import kd.bos.entity.MainEntityType;
```

```
import kd.bos.entity.datamodel.events.BizDataEventArgs;
import kd.bos.entity.datamodel.events.GetEntityTypeEventArgs;
import kd.bos.entity.property.TextProp;
import kd.bos.exception.ErrorCode;
import kd.bos.exception.KDException;
import kd.bos.form.ClientProperties;
import kd.bos.form.container.Container;
import kd.bos.form.control.Button;
import kd.bos.form.control.Control;
import kd.bos.form.control.EntryGrid;
import kd.bos.form.control.events.ClickListener;
import kd.bos.form.events.LoadCustomControlMetasArgs;
import kd.bos.form.events.OnGetControlArgs;
import kd.bos.form.field.TextEdit;
import kd.bos.form.plugin.AbstractFormPlugin;
import kd.bos.metadata.entity.commonfield.TextField;
import kd.bos.metadata.form.container.FlexPanelAp;
import kd.bos.metadata.form.control.ButtonAp;
import kd.bos.metadata.form.control.EntryAp;
import kd.bos.metadata.form.control.EntryFieldAp;
import kd.bos.metadata.form.control.FieldAp;

public class LoadCustomControlMetasSample extends AbstractFormPlugin implements ClickListener {

    private final static String KEY_ENTRYENTITY = "entryentity";
    private final static String KEY_MYFIELDCONTAINER = "myfieldcontainer";
    private final static String KEY_AUTOTEXT1 = "autotext1";
    private final static String KEY_AUTOTEXT2 = "autotext2";
    private final static String KEY_AUTOBUTTON1 = "autobutton1";

    /**
     * 界面显示前，触发此事件：向前端界面输出动态添加的字段、控件
     * @remark
     * 这个事件只能向前端界面添加字段、控件；后台的视图模型、数据模型，均没有改变
     */
    @Override
    public void loadCustomControlMetas(LoadCustomControlMetasArgs e) {
        super.loadCustomControlMetas(e);

        // 动态添加单据头字段、按钮
        FlexPanelAp headAp = this.createDynamicPanel();
        Map<String, Object> mapHead = new HashMap<>();
        mapHead.put(ClientProperties.Id, KEY_MYFIELDCONTAINER);
        mapHead.put(ClientProperties.Items,
```

```
headAp.createControl().get(ClientProperties.Items));
        e.getItems().add(mapHead);

        // 动态添加单据体字段
        EntryAp entryAp = this.createDynamicEntryAp();
        Map<String, Object> mapEntry = new HashMap<>();
        mapEntry.put(ClientProperties.Id, KEY_ENTRYENTITY);
        mapEntry.put(ClientProperties.Columns,
entryAp.createControl().get(ClientProperties.Columns));
        e.getItems().add(mapEntry);
    }

    /**
     * 此事件在系统要用到表单主实体模型时触发
     * @param e
     * @remark
     * 插件修改原始主实体，注册自定义属性，返回新的主实体给系统
     */
    @Override
    public void getEntityType(GetEntityTypeEventArgs e) {

        // 取原始的主实体
        MainEntityType oldMainType = e.getOriginalEntityType();
        // 复制主实体
        MainEntityType newMainType = null;
        try{
            newMainType = (MainEntityType)oldMainType.clone();
        }
        catch (CloneNotSupportedException exp){
            throw new KDException(exp, new ErrorCode("LoadCustomControlMetasSample",
exp.getMessage()));
        }

        // 为自定义的文本字段，向主实体注册文本属性
        this.registDynamicProps(newMainType);

        // 回传主实体给系统
        e.setNewEntityType(newMainType);
    }

    /**
     * 此事件在表单创建界面数据包时触发
     * @remark
     * 由插件自行创建界面数据包，包含自定义字段
```

```
     */
    @Override
    public void createNewData(BizDataEventArgs e) {

        DynamicObject dataEntity = new DynamicObject(this.getModel().getDataEntityType());
        dataEntity.set(KEY_AUTOTEXT1, "动态文本字段1的值");

        DynamicObjectCollection rows = dataEntity.getDynamicObjectCollection(KEY_ENTRYENTITY);
        DynamicObject newRow = new DynamicObject(rows.getDynamicObjectType());
        newRow.set(KEY_AUTOTEXT2, "动态文本字段2的值");

        rows.add(newRow);

        e.setDataEntity(dataEntity);
    }

    /**
     * 此事件在把数据绑定到界面之前触发：
     * 系统会调用FormDataBinder对象，把字段值输出给前端字段编辑控件；
     * @param e
     * @remark
     * 动态添加的字段，在FormDataBinder中并没有记录，因此，默认不会绑定动态添加的字段值；
     * 必须在此事件，向FormDataBinder中注册动态添加的字段
     */
    @Override
    public void beforeBindData(EventObject e) {

        // 单据头添加的字段、控件，注入到容器面板的控件编程模型中
        FlexPanelAp dynamicPanel = this.createDynamicPanel();
        Container myFldPanel = this.getView().getControl(KEY_MYFIELDCONTAINER);
        myFldPanel.getItems().addAll(dynamicPanel.buildRuntimeControl().getItems());
        this.getView().createControlIndex(myFldPanel.getItems());

        // 单据体添加的字段，注入到单据体表格的控件编程模型中
        EntryAp dynamicEntryAp = this.createDynamicEntryAp();
        EntryGrid entryGrid = this.getView().getControl(KEY_ENTRYENTITY);
        List<Control> fieldEdits = dynamicEntryAp.buildRuntimeControl().getItems();
        for(Control fieldEdit : fieldEdits){
            fieldEdit.setView(this.getView());
            entryGrid.getItems().add(fieldEdit);
        }
    }

    /**
```

```
     * 用户与自定义的控件进行交互时，会触发此事件
     * @remark
     * 插件在此事件中，创建自定义控件的编程模型，并侦听其事件
     */
    @Override
    public void onGetControl(OnGetControlArgs e) {
        if (StringUtils.equals(KEY_AUTOBUTTON1, e.getKey())){
            // 用户点击按钮时，会触发此事件：创建按钮的控件编程模型Button实例返回

            Button button = new Button();
            button.setKey(KEY_AUTOBUTTON1);   // 必须
            button.setView(this.getView()); // 必须

            button.addClickListener(this);
            e.setControl(button);
        }
        else if (StringUtils.equals(KEY_AUTOTEXT1, e.getKey())){
            // 用户修改了文本1字段值，前端上传字段值时，会先取字段的控件编程模型；
            // 如果没有本段代码，字段值上传失败，不会修改到数据模型中
            TextEdit textEdit = new TextEdit();
            textEdit.setKey(KEY_AUTOTEXT1);
            textEdit.setView(this.getView());
            e.setControl(textEdit);
        }
        else if (StringUtils.equals(KEY_AUTOTEXT2, e.getKey())){
            TextEdit textEdit = new TextEdit();
            textEdit.setKey(KEY_AUTOTEXT2);
            textEdit.setEntryKey(KEY_ENTRYENTITY);
            textEdit.setView(this.getView());
            e.setControl(textEdit);
        }
    }

    @Override
    public void click(EventObject evt) {
        super.click(evt);
        Control source = (Control)evt.getSource();
        if (StringUtils.equals(source.getKey(), KEY_AUTOBUTTON1)){
            String text1 = (String)this.getModel().getValue(KEY_AUTOTEXT1);
            this.getView().showMessage("您点击了动态添加的按钮1，此时动态文本字段1的值是：" +
text1);
        }
    }
```

```
/**
 * 创建一个面板，并向其中动态添加字段、控件
 */
private FlexPanelAp createDynamicPanel(){

    FlexPanelAp headPanelAp = new FlexPanelAp();
    headPanelAp.setKey("headAp");

    // 动态添加一个文本字段
    FieldAp fieldAp = new FieldAp();
    fieldAp.setId(KEY_AUTOTEXT1);
    fieldAp.setKey(KEY_AUTOTEXT1);
    fieldAp.setName(new LocaleString("自动文本1"));
    fieldAp.setBackColor("#FFFFFF");
    fieldAp.setFireUpdEvt(true);  // 即时触发值更新事件

    TextField field = new TextField();
    field.setId(KEY_AUTOTEXT1);
    field.setKey(KEY_AUTOTEXT1);
    fieldAp.setField(field);

    headPanelAp.getItems().add(fieldAp);

    // 动态添加一个按钮
    ButtonAp buttonAp = new ButtonAp();
    buttonAp.setId(KEY_AUTOBUTTON1);
    buttonAp.setKey(KEY_AUTOBUTTON1);
    buttonAp.setName(new LocaleString("自动添加的按钮"));

    headPanelAp.getItems().add(buttonAp);

    return headPanelAp;
}

/**
 * 创建一个单据体表格，并向其中动态添加字段
 */
private EntryAp createDynamicEntryAp(){

    EntryAp entryAp = new EntryAp();
    entryAp.setKey("entryap");

    // 动态添加一个文本字段
    EntryFieldAp fieldAp = new EntryFieldAp();
```

```
        fieldAp.setId(KEY_AUTOTEXT2);
        fieldAp.setKey(KEY_AUTOTEXT2);
        fieldAp.setName(new LocaleString("自动文本2"));
        fieldAp.setFireUpdEvt(true);  // 即时触发值更新事件

        TextField field = new TextField();
        field.setId(KEY_AUTOTEXT2);
        field.setKey(KEY_AUTOTEXT2);
        fieldAp.setField(field);

        entryAp.getItems().add(fieldAp);

        return entryAp;
    }

    /**
     * 向主实体注册动态添加的属性
     */
    private void registDynamicProps(MainEntityType newMainType){
        // 向单据头动态注册一个新的文本属性
        TextProp textProp1 = new TextProp();

        textProp1.setName(KEY_AUTOTEXT1); // 标识
        textProp1.setDisplayName(new LocaleString("自动文本1"));     // 标题

        textProp1.setDbIgnore(true);  // 此字段不需到物理表格取数
        textProp1.setAlias("");           // 物理字段名

        // 把新字段，注册到单据头
        newMainType.registerSimpleProperty(textProp1);

        // 向单据体动态注册一个新的文本属性
        EntryType entryType = (EntryType)newMainType.getAllEntities().get(KEY_ENTRYENTITY);

        TextProp textProp2 = new TextProp();

        textProp2.setName(KEY_AUTOTEXT2); // 标识
        textProp2.setDisplayName(new LocaleString("自动文本2"));     // 标题

        textProp2.setDbIgnore(true);  // 此字段不需到物理表格取数
        textProp2.setAlias("");           // 物理字段名

        // 把新字段，注册到单据体
        entryType.registerSimpleProperty(textProp2);
```

```
    }
}
```

2.4.4. setView 事件

2.4.4.1. 事件触发时机

表单视图模型初始化，创建插件时，调用此方法，向插件传入表单视图模型 IFormView 实例；

插件基类 AbstractFormPlugin，已经处理了此事件，接收了传入的表单视图模型，并存储在本地变量中，因此，业务插件可以通过插件基类提供的 getView()方法获取表单视图模型实例；

业务插件，通常不需要捕获此事件。
特别说明：

严格来说，setView 方法，并不是插件事件：
插件初始化时，系统通过此方法把界面编程模型传递给插件，为插件准备运行上下文环境。

插件不能覆盖基类的实现方法，否则插件的 getView()方法，获取不到表单视图模型实例，无法对界面进行控制。

插件基类 setView 方法的实现代码如下：

Java

```java
public class AbstractFormPlugin extends AbstractDataModelPlugin implements IFormPlugin {

    private IFormView formView;

    public IFormView getView() {
        return this.formView;
    }

    @Override
    public void setView(IFormView formView) {
        this.formView = formView;
    }
}
```

2.4.4.2. 代码模板

不需要业务插件重写，略过。

2.4.4.3. 参数说明

- `IFormView formView` ：表单编程模型，插件可以据此访问、控制表单。详细方法说明，请参见 IFormView 介绍章节

2.4.4.4. 应用示例

业务插件，通常不需要重写此事件方法，示例略。

2.4.5. initialize 事件

2.4.5.1. 事件触发时机

表单视图模型初始化，创建插件后，触发此事件；

下一代金蝶云，表单界面在服务端是无状态的：
即界面加载完毕之后，服务端的表单视图模型实例、数据模型实例，会被即时销毁，界面数据，自动存储到缓存中；

用户与前端界面发生交互，前端向服务器发送请求，服务端会重新构建表单的视图模型实例、数据模型实例，并从缓存中恢复界面数据。

因此，在表单显示期间，后台的表单视图模型，会被频繁的构建、初始化、销毁，业务插件也会被频繁的构建、初始化、销毁；

插件可以在此事件中，初始化必要的变量；
不可以在此事件设置字段值、设置控件状态、侦听控件事件；
因为这个事件执行的非常频繁，尽可能不要重写，执行复杂逻辑，消耗性能；

2.4.5.2. 代码模板

```Java
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import kd.bos.form.plugin.AbstractFormPlugin;

public class Initialize extends AbstractFormPlugin {

    @Override
    public void initialize() {
        super.initialize();
        // TODO 在此添加业务逻辑
    }
}
```

2.4.5.3. 参数说明

本事件无参数。
但可以通过 this.getView()方法，获取表单编程模型 IFormView 对象，据此访问各种界面信息。

2.4.5.4. 应用示例

此事件执行的非常频繁，插件尽量不要捕捉此事件，消耗性能。

示例略。

2.4.6. registerListener 事件

2.4.6.1. 事件触发时机

用户与界面上的控件进行交互时，即会触发此事件。

建议在此事件，侦听各个控件的插件事件。

2.4.6.2. 代码模板

```Java
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import java.util.EventObject;

import kd.bos.form.plugin.AbstractFormPlugin;

public class RegisterListener extends AbstractFormPlugin {

    @Override
    public void registerListener(EventObject e) {
        // TODO 侦听控件的插件事件
    }
}
```

2.4.6.3. 参数说明

- `EventObject e`：事件参数对象，含有事件源
  - ✓ `Object getSource()`：事件源，表单编程模型 IFormView 对象

2.4.6.4. 应用示例

- 案例说明：

1. 响应主菜单上自定义的菜单点击事件；
2. 响应自定义按钮点击事件；
3. 响应单据体行点击事件；
4. 响应树形控件节点点击事件；

- 实现方案：

1. 插件实现控件的插件事件接口，实现事件的处理方法
2. 捕获表单 refisterListener 事件，侦听控件的插件事件，传入实现了事件接口的插件实例
3. 略过事件处理的详细逻辑

- 实例代码：

```Java
package kd.bos.plugin.sample.dynamicform.pcform.form.bizcase;
```

```
import java.util.EventObject;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.control.Button;
import kd.bos.form.control.Control;
import kd.bos.form.control.EntryGrid;
import kd.bos.form.control.Toolbar;
import kd.bos.form.control.TreeView;
import kd.bos.form.control.events.ClickListener;
import kd.bos.form.control.events.ItemClickEvent;
import kd.bos.form.control.events.ItemClickListener;
import kd.bos.form.control.events.RowClickEvent;
import kd.bos.form.control.events.RowClickEventListener;
import kd.bos.form.control.events.TreeNodeClickListener;
import kd.bos.form.control.events.TreeNodeEvent;
import kd.bos.form.plugin.AbstractFormPlugin;

public class RegisterListenerSample extends AbstractFormPlugin implements ItemClickListener,
ClickListener, RowClickEventListener, TreeNodeClickListener {

    private final static String KEY_MBAR = "tbmain";
    private final static String KEY_BARITEM1 = "baritem1";
    private final static String KEY_BUTTON1 = "buttonap1";
    private final static String KEY_ENTRYENTITY = "entryentity";
    private final static String KEY_TREEVIEW1 = "treeviewap1";

    @Override
    public void registerListener(EventObject e) {
        super.registerListener(e);

        // 侦听各控件的插件事件，传入实现了事件接口的插件实例

        // 主菜单按钮点击
        Toolbar mbar = this.getView().getControl(KEY_MBAR);
        mbar.addItemClickListener(this);

        // 按钮点击
        Button button = this.getView().getControl(KEY_BUTTON1);
        button.addClickListener(this);

        // 单据体行点击
        EntryGrid entryGrid = this.getView().getControl(KEY_ENTRYENTITY);
        entryGrid.addRowClickListener(this);
```

```
        // 树型控件点击
        TreeView treeView = this.getView().getControl(KEY_TREEVIEW1);
        treeView.addTreeNodeClickListener(this);
    }

    @Override
    public void itemClick(ItemClickEvent evt) {
        super.itemClick(evt);
        if (StringUtils.equals(KEY_BARITEM1, evt.getItemKey())){
            // 事件处理代码略过
        }
    }

    @Override
    public void click(EventObject evt) {
        super.click(evt);
        Control source = (Control)evt.getSource();
        if (StringUtils.equals(KEY_BUTTON1, source.getKey())){
            // 事件处理代码略过
        }
    }

    @Override
    public void entryRowClick(RowClickEvent evt) {
        Control source = (Control) evt.getSource();
        if (StringUtils.equals(KEY_ENTRYENTITY, source.getKey())){
            // 事件处理代码略过
        }
    }

    @Override
    public void treeNodeClick(TreeNodeEvent evt) {
        // 事件处理代码略过
        TreeView treeView = (TreeView)evt.getSource();
        if (StringUtils.equals(KEY_TREEVIEW1, treeView.getKey())){
            // 事件处理代码略过
        }
    }
}
```

2.4.7. getEntityType 事件

2.4.7.1. 事件触发时机

表单基于实体模型，创建数据包之前，触发此事件，传入系统自动读取到的表单主实体模型；

插件可以在此事件，修改表单原始实体模型，动态注册新的属性，从而实现向界面动态添加字段。

特别说明：
向界面动态添加字段，还需要和另外几个事件一起配合，详见 loadCustomControlMetas 事件说明；

2.4.7.2. 代码模板

本事件需要与其他事件一起配合使用，代码模板放在一起，详见 loadCustomControlMetas 事件。

2.4.7.3. 参数说明

- `GetEntityTypeEventArgs e`:
  - ✓ `MainEntityType getOriginalEntityType()`: 界面原始的实体模型
  - ✓ `setNewEntityType(MainEntityType newEntityType)`: 设置界面最新的实体模型；通常是以原始界面实体模型为模板，克隆后，动态添加一些新的属性对象，传回界面；

2.4.7.4. 应用示例

参见 loadCustomControlMetas 事件的应用示例，动态添加字段后，同步处理此事件，向主实体注册新字段对应的属性。

2.4.8. createNewData 事件

2.4.8.1. 事件触发时机

界面初始化或刷新，开始新建数据包时触发此事件；

插件可以在此事件，自行创建界面数据包传回给系统，跳过系统内置的数据包创建过程。

典型的应用场景：
1. 需要由外部传入完整地界面数据包，显示在界面上：就可以在此事件，先读取传入的数据包，然后传回表单，替代系统创建数据包的过程；
2. 动态添加字段时，系统内置创建的数据包，没有新字段的值，需要自行创建界面数据包；

2.4.8.2. 代码模板

Java

```java
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import kd.bos.entity.datamodel.events.BizDataEventArgs;
import kd.bos.form.plugin.AbstractFormPlugin;

public class CreateNewData extends AbstractFormPlugin {

    @Override
    public void createNewData(BizDataEventArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

2.4.8.3. 参数说明

- BizDataEventArgs e:
  - ✓ void setDataEntity(Object dataEntity): 设置界面数据包(DynamicObject 实例)。
  - ✓ void setIsExecuteRule(boolean value): 此参数暂时未启用。

2.4.8.4. 应用示例

参阅 loadCustomControlMetas 事件实例，动态添加字段后，由插件捕获本事件，自行创建界面数据包。

2.4.9. afterCreateNewData 事件

2.4.9.1. 事件触发时机

界面初始化或刷新，新建表单数据包成功，并给字段填写了默认值之后，触发此事件；

插件可以在此事件，重设字段的默认值。

部分字段的默认值难以通过设计器配置出来，如需要计算的值、根据系统参数选项决定的值，必须写插件实现。

2.4.9.2. 代码模板

Java

```java
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import java.util.EventObject;

import kd.bos.form.plugin.AbstractFormPlugin;

public class AfterCreateNewData extends AbstractFormPlugin {

    @Override
    public void afterCreateNewData(EventObject e) {
        // TODO 在此添加业务逻辑
    }
}
```

2.4.9.3. 参数说明

- `EventObject e`：事件参数对象，含有事件源

- ✓ `Object getSource()`：表单数据模型 IDataModel 对象，可以据此对表单数据进行控制

2.4.9.4. 应用示例

➢ 案例说明

1. 表单加载时，给单据体默认创建 10 行，并设置整数字段(integerfieldap1)值为与行号相同

➢ 实现方案

1. 处理 afterCreateNewData 事件
   a. 给单据体添加 10 行；
   b. 逐行填写整数字段值；

➢ 实例代码

Java

```
package kd.bos.plugin.sample.dynamicform.pcform.form.bizcase;

import java.util.EventObject;

import kd.bos.form.plugin.AbstractFormPlugin;

public class AfterCreateNewDataSample extends AbstractFormPlugin {

    private final static String KEY_ENTRYENTITY = "entryentity";
    private final static String KEY_INTEGERFIELD1 = "integerfieldap1";


    @Override
    public void afterCreateNewData(EventObject e) {
        int rowCount = this.getModel().getEntryRowCount(KEY_ENTRYENTITY);
        if (rowCount < 10){
            // 给单据体补足10行
            this.getModel().batchCreateNewEntryRow(KEY_ENTRYENTITY, 10 - rowCount);
            rowCount = 10;
        }

        // 逐行给整数字段设置默认值
        for(int row = 0; row < rowCount ; row++){
            int fldValue = row + 1;
            this.getModel().setValue(KEY_INTEGERFIELD1, fldValue, row);
        }
    }
```

```
}
```

2.4.10. beforeBindData 事件

2.4.10.1. 事件触发时机

界面数据包构建完毕，开始生成指令，刷新前端字段值、控件状态之前，触发此事件；

插件可以在此事件中，调整后台视图模型(IFormView)中的字段、控件属性，间接控制前端界面字段值、控件状态；

特别说明：

- 本事件与 afterCreateNewData 事件的区别：
  - ✓ 本事件比 afterCreateNewData 事件晚触发；
  - ✓ 适合在 afterCreateNewData 事件中，修改数据模型中的字段值：
    - 在 afterCreateNewData 改变字段值，数据修改标志为 false；退出时，不会提示数据被修改；而在此事件中修改字段值，数据修改标志为 true，退出时系统可能会提示数据被修改。
  - ✓ 适合在 beforeBindData 中，调整视图模型中的控件属性；
  - ✓ 单据界面插件，afterCreateNewData 不是必然会被触发（与 afterLoadData 互斥），而 beforeBindData 必然触发

- 本事件与 afterBindData 事件的差别：
  - ✓ 本事件比 afterBindData 早触发：在本事件之后，系统会调用内置的字段值绑定过程，随后才会触发 afterBindData 事件；
  - ✓ beforeBindData 事件，适合设置字段、控件的属性，以间接的控制前端字段值、控件状态的刷新过程；
    - 在 beforeBindData 事件中设置控件状态会没有效果，因为系统随后会清空所有控件的状态；
  - ✓ afterBindData 事件，适合直接设置控件在前端表现的内容、状态；

2.4.10.2. 代码模板

Java

```
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import java.util.EventObject;
```

```
import kd.bos.form.plugin.AbstractFormPlugin;

public class BeforeBindData extends AbstractFormPlugin {

    @Override
    public void beforeBindData(EventObject e) {
        super.beforeBindData(e);

        // TODO 在此添加业务逻辑
    }
}
```

2.4.10.3. 参数说明

- EventObject e：事件参数对象，含有事件源
  - ✓ Object getSource()：表单视图模型 IFormView 对象

2.4.10.4. 应用示例

参阅 loadCustomControlMetas 事件实例，向前端动态添加字段控件元数据后，在本事件中，向后台视图模型，添加字段的控件编程模型实例，以确保动态添加的字段值被输出到前端。

2.4.11. afterBindData 事件

2.4.11.1. 事件触发时机

界面数据包构建完毕，生成指令，刷新前端字段值、控件状态之后，触发此事件；

插件可以在此事件，根据各字段值数据，重新设置控件、字段的可用、可见性等。

不要在此事件，修改字段值。

请参阅 beforeBindData 事件说明，了解本事件与 beforeBindData 事件的区别。

2.4.11.2. 代码模板

```Java
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import java.util.EventObject;

import kd.bos.form.plugin.AbstractFormPlugin;

public class AfterBindData extends AbstractFormPlugin {

    @Override
    public void afterBindData(EventObject e) {
        super.afterBindData(e);
        // TODO 在此添加业务逻辑
    }
}
```

2.4.11.3. 参数说明

- `EventObject e`：事件参数对象，含有事件源
  - ✓ `Object getSource()`：表单视图模型 IFormView 对象

2.4.11.4. 应用示例

请参阅过滤条件表格控件(FilterGrid)的应用示例：
1. 在 beforeBindData 事件中，设置过滤条件表格控件绑定的业务单据，以便系统在刷新过滤条件表格时，读取业务单据的字段，绑定到过滤条件表格；
2. 在 afterBindData 事件中，把已经配置好的条件内容，绑定到过滤条件表格；

2.4.12. beforeItemClick 事件

2.4.12.1. 事件触发时机

用户点击菜单按钮后，在执行按钮绑定的操作前，触发此事件；

插件可以在此事件，取消菜单绑定的操作；

2.4.12.2. 代码模板

Java

```
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import java.util.EventObject;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.control.events.BeforeItemClickEvent;
import kd.bos.form.plugin.AbstractFormPlugin;

public class BeforeItemClick extends AbstractFormPlugin {

    private final static String KEY_MAINBAR = "tbar_main";
    private final static String KEY_BARITEM_NEW = "baritem_new";

    @Override
    public void registerListener(EventObject e) {
        super.registerListener(e);
        // 侦听主菜单按钮点击事件
        this.addItemClickListeners(KEY_MAINBAR);
    }

    @Override
    public void beforeItemClick(BeforeItemClickEvent evt) {
        if (StringUtils.equals(KEY_BARITEM_NEW, evt.getItemKey())){
            // TODO 在此添加业务逻辑
        }
    }
}
```

说明：
常量 *KEY_MAINBAR* 是示例菜单标识；
常量 *KEY_BARITEM_NEW* 是示例按钮标识；

2.4.12.3. 事件参数

- `public class BeforeItemClickEvent extends ItemClickEvent`
  - ✓ `public String getItemKey()`：菜单项标识

- ✓ `public String getOperationKey()`：菜单项绑定的操作
- ✓ `public void setCancel(boolean cancel)`：取消操作

2.4.12.4. 示例

参见 itemClick 事件示例，在处理自定义菜单点击事件前，检查数据录入是否合理，决定能否执行菜单点击逻辑；

2.4.13. itemClick 事件

2.4.13.1. 事件触发时机

用户点击菜单项时，触发此事件；

插件可以在此响应自定义菜单项的点击处理。

2.4.13.2. 代码模板

Java

```java
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import java.util.EventObject;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.control.events.ItemClickEvent;
import kd.bos.form.plugin.AbstractFormPlugin;

public class ItemClick extends AbstractFormPlugin {

    private final static String KEY_MAINBAR = "tbar_main";
    private final static String KEY_BARITEM_NEW = "baritem_new";

    @Override
    public void registerListener(EventObject e) {
        super.registerListener(e);
```

```
        // 侦听主菜单按钮点击事件
        this.addItemClickListeners(KEY_MAINBAR);
    }

    @Override
    public void itemClick(ItemClickEvent evt) {
        super.itemClick(evt);
        if (StringUtils.equals(KEY_BARITEM_NEW, evt.getItemKey())){
            // TODO 在此添加业务逻辑
        }
    }
}
```

说明：
常量 ***KEY_MAINBAR*** 是菜单栏标识；
常量 ***KEY_BARITEM_NEW*** 是按钮标识；

2.4.13.3. 事件参数

- `public class ItemClickEvent extends EventObject`
  - ✓ `public String getItemKey()`：菜单项标识
  - ✓ `public String getOperationKey()`：菜单项绑定的操作

2.4.13.4. 示例

- 案例说明

1. 主菜单栏，增加一个自定义菜单；
2. 用户点击自定义菜单时，检查名字字段有没有填写；
   a. 已经填写了名字时，提示"hello , xxx!"
   b. 没有填写名字时，提示 "hello, who are you?"

- 实现方案

1. 在 beforeItemClick 事件，检查用户有没有填写名字
   a.如果没有填写，取消后续操作，显示”hello, who are you?”
2. 在 itemClick 事件，不需再检查名字有没有填写，直接显示”hello , xxx!”

- 实例代码

Java

```java
package kd.bos.plugin.sample.dynamicform.pcform.form.bizcase;

import java.util.EventObject;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.control.events.BeforeItemClickEvent;
import kd.bos.form.control.events.ItemClickEvent;
import kd.bos.form.plugin.AbstractFormPlugin;

public class ItemClickSample extends AbstractFormPlugin {

    private final static String KEY_MAINBAR = "tbar_main";
    private final static String KEY_BARITEM_HELLO = "baritem_hello";
    private final static String KEY_NAME = "name";

    @Override
    public void registerListener(EventObject e) {
        super.registerListener(e);
        // 侦听主菜单按钮点击事件
        this.addItemClickListeners(KEY_MAINBAR);
    }

    @Override
    public void beforeItemClick(BeforeItemClickEvent evt) {
        if (StringUtils.equals(KEY_BARITEM_HELLO, evt.getItemKey())){
            String youName = (String)this.getModel().getValue(KEY_NAME);
            if (StringUtils.isBlank(youName)){
                this.getView().showMessage("hello, who are you?");
                evt.setCancel(true); // 取消后续操作
            }
        }
    }

    @Override
    public void itemClick(ItemClickEvent evt) {
        super.itemClick(evt);
        if (StringUtils.equals(KEY_BARITEM_HELLO, evt.getItemKey())){
            String youName = (String)this.getModel().getValue(KEY_NAME);
            this.getView().showMessage("hello, " + youName + "!");
        }
    }
}
```

2.4.14. beforeDoOperation 事件

事件触发时机

用户点击按钮、菜单，执行绑定的操作逻辑前，触发此事件；

插件可以在此事件：
1. 提示确认消息；
2. 校验数据，取消操作的执行；
3. 传递给自定义操作参数给操作服务、操作插件；

表单 beforeDoOperation 事件与操作校验器的区别：
- 运行时机不同：
  - ✓ 表单 beforeDoOperation 事件，是由表单触发的，只有在表单上执行操作时，才会被触发；后台直接操作服务，不会触发此事件；数据校验逻辑放在这个事件，有可能漏过；
  - ✓ 操作校验器，是由微服务层操作引擎执行的，不管是在表单上执行操作，还是后台调用操作服务，都会执行微服务层操作引擎，都会执行操作校验器；
- 适用的操作类型不同：
  - ✓ 操作分为两种大类：
    - 表单操作：对界面进行处理，如关闭界面；
    - 实体操作：更新数据库，如保存；
  - ✓ 只有实体操作，才允许配置操作校验；
  - ✓ 对普通的表单操作进行数据校验，只能使用表单插件 beforeDoOperation 事件；
- 控制颗粒度不同：
  - ✓ 批量操作时，操作校验器，对批量数据进行逐个校验，略过校验失败的数据，继续执行校验成功数据；
  - ✓ 表单 beforeDoOperation 事件，只能整体取消操作，不能对批量数据进行区分；

因此，对操作进行数据校验，尽可能配置操作校验器、或使用操作插件，而不是使用表单 beforeDoOperation 事件。

代码模板

Java

```
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.events.BeforeDoOperationEventArgs;
import kd.bos.form.operate.FormOperate;
```

```
import kd.bos.form.plugin.AbstractFormPlugin;

public class BeforeDoOperation extends AbstractFormPlugin {

    final static String KEY_OPKEY = "myoperation";

    @Override
    public void beforeDoOperation(BeforeDoOperationEventArgs args) {
        super.beforeDoOperation(args);

        FormOperate formOperate = (FormOperate)args.getSource();
        if ( StringUtils.equals(KEY_OPKEY, formOperate.getOperateKey())){
            // TODO 在此添加业务逻辑
        }
    }
}
```

说明：
常量 ***KEY_OPKEY*** 是操作标识，要根据实际场景进行替换。

参数说明

- `BeforeDoOperationEventArgs args`：
  - ✓ `Object getSource()`：FormOperate 类型，操作执行类，包含了操作的配置信息
  - ✓ `void setCancel(boolean cancel)`：可以取消操作

应用示例

- 案例说明

1. 用户点击"付款"菜单按钮时，提示用户确认
   a. 用户确认后，才继续执行付款操作
   b. 用户取消，则不执行付款操作

- 实现方案

1. 捕获表单 beforeDoOperation 事件：
   a. 开始执行付款前，显示交互提示，取消本次操作；
   b. 用户确认后再次执行付款操作时，不再重复显示交互提示；
   c. 通过自定义操作参数，标志是否为确认后再次执行操作；
2. 捕获表单 confirmCallBack 事件：
   a. 获取用户确认结果；

b. 用户确认付款时，重新调用付款操作；
c. 设置自定义操作参数值，标志为确认后再次执行操作，避免重复显示交互提示；

➢ 实例代码

Java

```
package kd.bos.plugin.sample.dynamicform.pcform.form.bizcase;

import kd.bos.dataentity.OperateOption;
import kd.bos.dataentity.RefObject;
import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.ConfirmCallBackListener;
import kd.bos.form.ConfirmTypes;
import kd.bos.form.MessageBoxOptions;
import kd.bos.form.MessageBoxResult;
import kd.bos.form.events.BeforeDoOperationEventArgs;
import kd.bos.form.events.MessageBoxClosedEvent;
import kd.bos.form.operate.FormOperate;
import kd.bos.form.plugin.AbstractFormPlugin;

public class BeforeDoOperationSample extends AbstractFormPlugin {

    private final static String KEY_PAY = "pay";
    private final static String OPPARAM_AFTERCONFIRM = "afterconfirm";

    /**
     * 执行操作前，触发此事件
     */
    @Override
    public void beforeDoOperation(BeforeDoOperationEventArgs args) {
        super.beforeDoOperation(args);

        FormOperate operate = (FormOperate)args.getSource();
        if (StringUtils.equals(operate.getOperateKey(), KEY_PAY)){
            // 付款操作

            // 尝试读取操作自定义参数：判断是否确认后再次执行付款操作
            RefObject<String> afterConfirm = new RefObject<>();
            if (!operate.getOption().tryGetVariableValue(OPPARAM_AFTERCONFIRM,
afterConfirm)){
                // 自定义操作参数中，没有afterconfirm参数：说明是首次执行付款操作，需要提示用户确认

                // 显示确认消息
                ConfirmCallBackListener confirmCallBacks = new
ConfirmCallBackListener(KEY_PAY, this);
```

```
                String confirmTip = "您确认要付款给xxx?";
                this.getView().showConfirm(confirmTip, MessageBoxOptions.YesNo,
ConfirmTypes.Default, confirmCallBacks);

                // 在没有确认之前，先取消本次操作
                args.setCancel(true);
            }

            // 如下代码，演示如何增加自定义操作参数，由系统传递给操作服务及操作插件
            // （仅供演示，与本案例需求无关）
            operate.getOption().setVariableValue(OPPARAM_AFTERCONFIRM, "true");
        }
    }

    /**
     * 用户确认了交互信息后，触发此事件
     */
    @Override
    public void confirmCallBack(MessageBoxClosedEvent messageBoxClosedEvent) {
        super.confirmCallBack(messageBoxClosedEvent);

        if (StringUtils.equals(KEY_PAY, messageBoxClosedEvent.getCallBackId())){
            // 付款确认

            if (messageBoxClosedEvent.getResult() == MessageBoxResult.Yes){
                // 确认执行付款操作

                // 构建操作自定义参数，标志为确认后再次执行操作，避免重复显示交互提示
                OperateOption operateOption = OperateOption.create();
                operateOption.setVariableValue(OPPARAM_AFTERCONFIRM, "true");

                // 执行付款操作，并传入自定义操作参数
                this.getView().invokeOperation(KEY_PAY, operateOption);
            }
        }
    }
}
```

2.4.15. afterDoOperation 事件

事件触发时机

用户点击按钮、菜单，执行完绑定的操作后，不论成功与否，均会触发此事件；

插件可以在此事件，根据操作结果控制界面。

特别说明：
这个事件，是在表单界面层执行的，没有事务保护。
不允许在此事件同步修改数据库数据，以免同步失败导致数据不一致。

代码模板

Java

```java
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.events.AfterDoOperationEventArgs;
import kd.bos.form.operate.FormOperate;
import kd.bos.form.plugin.AbstractFormPlugin;

public class AfterDoOperation extends AbstractFormPlugin {

    final static String KEY_OPKEY = "myoperation";

    @Override
    public void afterDoOperation(AfterDoOperationEventArgs args) {
        super.afterDoOperation(args);

        FormOperate formOperate = (FormOperate)args.getSource();
        if (StringUtils.equals(KEY_OPKEY, formOperate.getOperateKey())
                && args.getOperationResult() != null
                && args.getOperationResult().isSuccess()){
            // TODO 在此添加业务逻辑
        }
    }
}
```

说明：
常量 `KEY_OPKEY` 是操作标识，要根据实际场景进行替换。

参数说明

- `AfterDoOperationEventArgs args`:
  - ✓ `Object getSource()`: FormOperate 类型，操作执行类，包含了操作的配置信息
  - ✓ `String getOperateKey()`: 当前执行的操作标识
  - ✓ `OperationResult getOperationResult()`: 操作结果对象，包含操作成功标志，提示信息等;

应用示例

- 案例说明

1. 界面上有个"连续新增"选项;
2. 如果勾选了连续新增，则保存成功后，把界面切换为新增状态;
3. 如果未勾选连续新增，则保存成功后，自动关闭界面

- 实现方案

1. 捕获 afterDoOperation 事件，获取操作结果，如果操作成功:
   a. 未勾选连续新增选项，则关闭界面
   b. 勾选了连续新增选项，则执行新增操作

- 实例代码

Java

```java
package kd.bos.plugin.sample.dynamicform.pcform.form.bizcase;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.entity.operate.result.OperationResult;
import kd.bos.form.events.AfterDoOperationEventArgs;
import kd.bos.form.plugin.AbstractFormPlugin;

public class AfterDoOperationSample extends AbstractFormPlugin {

    private final static String OPERATEKEY_NEW = "new";
    private final static String OPERATEKEY_SAVE = "save";
    private final static String KEY_NEWCONTINUOUS = "newcontinuous";

    @Override
    public void afterDoOperation(AfterDoOperationEventArgs afterDoOperationEventArgs) {
```

```
        super.afterDoOperation(afterDoOperationEventArgs);

        if (StringUtils.equals(OPERATEKEY_SAVE, afterDoOperationEventArgs.getOperateKey())){

            OperationResult opResult = afterDoOperationEventArgs.getOperationResult();
            if (opResult != null && opResult.isSuccess()){

                // 读取界面上"连续新增"字段值
                boolean newContinuous = (boolean)this.getModel().getValue(KEY_NEWCONTINUOUS);
                if (newContinuous){
                    // 连续新增：执行新增操作
                    this.getView().invokeOperation(OPERATEKEY_NEW);
                    // 界面刷新后，重设"连续新增"选项值，确保此选项值的延续
                    this.getModel().setValue(KEY_NEWCONTINUOUS, true);
                }
                else {
                    // 不连续新增：关闭界面
                    this.getView().close();
                }
            }
        }
    }
}
```

2.4.16. confirmCallBack 事件

事件触发时机

用户确认了交互提示信息后，触发此事件，通知插件进行后续处理；

插件可以在此事件，了解用户的态度，决定后续业务逻辑。

特别说明：
1. 必须在显示交互提示时，设置回调参数，才会触发此事件；
2. 要注意避免重复显示相同交互信息，进入死循环；

示例：
下面的代码显示一条交互提示，需要用户确认；用户确认后，即会触发 confirCallBack 事件。

Java

```
ConfirmCallBackListener confirmCallBacks = new ConfirmCallBackListener("contentChange", this);
String confirmTip = sConfirmMsg + "";
this.getView().showConfirm(confirmTip, MessageBoxOptions.OKCancel, ConfirmTypes.Default,
confirmCallBacks);
```

代码模板

Java

```
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.MessageBoxResult;
import kd.bos.form.events.MessageBoxClosedEvent;
import kd.bos.form.plugin.AbstractFormPlugin;

public class ConfirmCallBack extends AbstractFormPlugin {

    private final static String CALLBACKID_MYMESSAGE = "mymessage";

    @Override
    public void confirmCallBack(MessageBoxClosedEvent messageBoxClosedEvent) {
        super.confirmCallBack(messageBoxClosedEvent);

        if (StringUtils.equals(CALLBACKID_MYMESSAGE, messageBoxClosedEvent.getCallBackId())
                && messageBoxClosedEvent.getResult() == MessageBoxResult.Yes){
            // TODO 在此添加业务逻辑
        }
    }
}
```

说明：
常量 ***CALLBACKID_MYMESSAGE*** 需要根据实际场景进行替换。

参数说明

- `MessageBoxClosedEvent messageBoxClosedEvent`：
  - ✓ `MessageBoxResult getResult()`：用户选择的确认结果，如 Yes，No 等；
  - ✓ `String getCallBackId()`：多处代码显示交互提示时，以此区分；

应用示例

请参阅 beforeDoOperation 事件的应用示例：
1. 执行付款操作前，显示确认提示消息；
2. 用户确认后，在 confirmCallBack 事件，重新执行付款操作；

2.4.17. closedCallBack 事件

事件触发时机

子界面关闭时，触发父界面的 closedCallBack 事件；

父界面的插件，可以在此事件，接收子界面返回的数据。

特别说明：
需要在显示子界面时，调用 FormShowParameter 参数的 setCloseCallBack 方法，设置回调属性，才会在子界面关闭时触发此事件：

Java

```
FormShowParameter showParameter = new FormShowParameter();
showParameter.setFormId(GetValByConditionEdit.FormId_ValByCondition);
showParameter.setCloseCallBack(new CloseCallBack(plugin, CALLBACKID_MYCALLBACK));
showParameter.getOpenStyle().setShowType(ShowType.Modal);
this.getView().showForm(showParameter);
```

代码模板

Java

```
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.events.ClosedCallBackEvent;
import kd.bos.form.plugin.AbstractFormPlugin;

public class ClosedCallBack extends AbstractFormPlugin {

    private final static String CALLBACKID_MYCALLBACK = "mycallback";
```

```
    @Override
    public void closedCallBack(ClosedCallBackEvent closedCallBackEvent) {
        super.closedCallBack(closedCallBackEvent);

        if (StringUtils.equals(closedCallBackEvent.getActionId(), CALLBACKID_MYCALLBACK)
                && closedCallBackEvent.getReturnData() != null){
            // TODO 在此添加业务逻辑
        }
    }
}
```

说明：
常量 ***CALLBACKID_MYCALLBACK*** 替代回调标识。

参数说明

- `ClosedCallBackEvent closedCallBackEvent`：
  - ✓ `public Object getSource()`：实现了 ICloseCallBack 接口的插件，后续将由此插件响应回调事件；
  - ✓ `public String getActionId()`：多处代码显示子界面时，通过此标识区分来源；
  - ✓ `public Object getReturnData()`：子界面返回数据

应用示例

- 案例说明

1. 表单上有个文本摘要字段 largertext1，编辑风格(按钮+文本)，可手工输入文本的摘要；
2. 表单上有个文本详情字段 largertext1_tag，隐藏不可见，存储完整的文本内容
3. 用户点击文本字段按钮时，打开大文本编辑界面，编辑文本详情
4. 大文本编辑界面关闭时，把文本内容，填写在本界面的文本详情字段中

- 实现方案

1. 捕获文本按钮 click 事件，打开大文本编辑界面，并传入文本字段名
2. 捕获 closedCallBack 事件，接受大文本编辑界面上的文本内容，填写在文本字段中

- 特别说明

1. 大文本编辑界面插件，会自动根据文本摘要字段名，加上_tag 后缀，作为文本详情字段名，然后自动到父界面上取此字段值；
2. 因此，本界面上文本详情标识、是文本字段 largertext1，加上_tag 后缀：largertext1_tag；

➢ 实例代码

Java

```
package kd.bos.plugin.sample.dynamicform.pcform.form.bizcase;

import java.util.EventObject;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.CloseCallBack;
import kd.bos.form.FormShowParameter;
import kd.bos.form.ShowType;
import kd.bos.form.control.Control;
import kd.bos.form.control.events.ClickListener;
import kd.bos.form.events.ClosedCallBackEvent;
import kd.bos.form.field.TextEdit;
import kd.bos.form.plugin.AbstractFormPlugin;

public class ClosedCallBackSample extends AbstractFormPlugin implements ClickListener {

    private final static String KEY_LARGETEXT1 = "largetext1";
    private final static String KEY_LARGETEXT1_TAG = "largetext1_tag";
    private final static String ENTITYNUMBER_LARGETEXTEDIT = "ide_largertextedit";

    /**
     * 注册控件事件
     */
    @Override
    public void registerListener(EventObject e) {
        super.registerListener(e);

        // 侦听文本字段的按钮点击事件
        TextEdit textEdit = this.getView().getControl(KEY_LARGETEXT1);
        textEdit.addClickListener(this);
    }

    /**
     * 控件点击事件
     */
    @Override
    public void click(EventObject evt) {
        super.click(evt);
        Control source = (Control)evt.getSource();
        if (StringUtils.equals(KEY_LARGETEXT1, source.getKey())){
```

```
            this.showLargerTextForm();
        }
    }

    /**
     * 子界面关闭事件
     */
    @Override
    public void closedCallBack(ClosedCallBackEvent closedCallBackEvent) {
        super.closedCallBack(closedCallBackEvent);
        if (StringUtils.equals(closedCallBackEvent.getActionId(), KEY_LARGETEXT1)){
            this.receiveLargerText(closedCallBackEvent);
        }
    }

    /**
     * 显示大文本编辑界面
     */
    private void showLargerTextForm(){

        FormShowParameter showParameter = new FormShowParameter();

        showParameter.setFormId(ENTITYNUMBER_LARGETEXTEDIT);

        // 传入自定义参数：文本摘要字段标识 largertext1
        // 大文本编辑界面插件会据此生成文本详情字段标识largertext1_tag，并到父界面取详情字段值
        showParameter.setCustomParam("fieldKey", KEY_LARGETEXT1);
        showParameter.setCustomParam("entryKey", "");

        showParameter.getOpenStyle().setShowType(ShowType.Modal);

        CloseCallBack callBack = new CloseCallBack(this, KEY_LARGETEXT1);   // 由本插件处理子界面的回调
        showParameter.setCloseCallBack(callBack);

        this.getView().showForm(showParameter);
    }

    /**
     * 接收大文本编辑界面返回的文本详情，填写在本界面的文本详情字段上
     * @param args
     */
    private void receiveLargerText(ClosedCallBackEvent args){
        if (args.getReturnData() == null){
```

```
            // 大文本编辑界面，点击了取消按钮，不做处理
            return;
        }

        this.getModel().setValue(KEY_LARGETEXT1_TAG, args.getReturnData());
    }
}
```

2.4.18. flexBeforeClosed 事件

事件触发时机

弹性域维护界面关闭时，触发父界面此事件；

插件可以对弹性域录入值进行校验，取消弹性域录入界面的关闭。

代码模板

Java

```
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.events.FlexBeforeClosedEvent;
import kd.bos.form.plugin.AbstractFormPlugin;

public class FlexBeforeClosed extends AbstractFormPlugin {

    private final static String KEY_BASEDATA1 = "basedata1";

    @Override
    public void flexBeforeClosed(FlexBeforeClosedEvent e) {
        super.flexBeforeClosed(e);
        if (StringUtils.equals(KEY_BASEDATA1, e.getBasedataKey())){
            // TODO 在此添加业务逻辑
        }
    }
}
```

说明：*KEY_BASEDATA1* 是弹性域父基础资料字段 Key，需要根据实际业务替换。

参数说明

- `FlexBeforeClosedEvent e`:
  - ✓ `public Object getSource()`: 弹性域维护界面的 IFormView，可以据此获取用户录入的弹性域各维度值
  - ✓ `public String getBasedataKey()`: 弹性域父基础资料字段 key
  - ✓ `public String getFlexKey()`: 弹性域字段 key
  - ✓ `public void setCancel(boolean cancel)`: 取消弹性域界面关闭

应用示例

暂缺示例。

2.4.19. onGetControl 事件

事件触发时机

在有代码尝试获取控件的编程模型时，触发此事件；
插件可以在此事件输出自定义控件的编程模型，侦听控件事件。

此事件，有两种典型的应用场景：

- 动态添加控件：向界面动态添加控件，需要由插件构建控件编程模型实例，并侦听其事件。
- 自定义控件：由插件构建自定义控件的编程模型实例；

代码模板

Java

```java
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.events.OnGetControlArgs;
import kd.bos.form.plugin.AbstractFormPlugin;
```

```
public class OnGetControl extends AbstractFormPlugin {

    private final static String KEY_CONTROL1 = "control1";

    @Override
    public void onGetControl(OnGetControlArgs e) {
        super.onGetControl(e);

        if (StringUtils.equals(KEY_CONTROL1, e.getKey())){
            // TODO 在此添加业务逻辑
        }
    }
}
```

说明：*KEY_CONTROL1* 是自定义控件的标识，需要根据实际业务进行替换

参数说明

- OnGetControlArgs e:
  - ✓ public Object getSource(): 表单 IFormView
  - ✓ public String getKey(): 自定义控件标识
  - ✓ public void setControl(Control control): 设置自定义控件的编程模型

应用示例

参见 loadCustomControlMetas 一节，演示如何动态添加按钮并侦听按钮点击事件。

2.4.20. customEvent 事件

事件触发时机

本事件，用于触发自定义控件的定制事件。

前端自定义控件，在与用户发生交互后，可以包装一个参数包，传入事件源，事件名，事件参数，发送一个事件触发请求到下一代 web 服务器，由系统转发给表单。

表单接收到此事件后，继续转发给业务插件，通知业务插件处理自定义控件的定制事件。

整个定制事件的触发过程，系统只作为传递通道。

代码模板

Java

```java
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.events.CustomEventArgs;
import kd.bos.form.plugin.AbstractFormPlugin;

public class CustomEvent extends AbstractFormPlugin {

    private final static String KEY_CONTROL1 = "control1";
    private final static String EVENT_CUSTOM = "customevent";

    @Override
    public void customEvent(CustomEventArgs e) {
        if (StringUtils.equals(e.getKey(), KEY_CONTROL1)
                && StringUtils.equals(e.getEventName(), EVENT_CUSTOM)){
            // TODO 在此添加业务逻辑
        }
    }
}
```

说明：
常量 ***KEY_CONTROL1*** 替代自定义控件标识；
常量 ***EVENT_CUSTOM*** 替代自定义控件的定制事件名；

参数说明

- CustomEventArgs e:
  - ✓ public Object getSource(): 表单 IFormView
  - ✓ public String getKey(): 自定义控件
  - ✓ public String getEventName(): 事件名
  - ✓ public String getEventArgs(): 事件参数，Json 字符串

应用示例

暂缺示例。

2.4.21. TimerElapsed 事件

事件触发时机

前端定时发送请求，触发此界面事件；

插件可以在此事件，定时向前端回馈信息；

特别说明：
需要在 preOpenForm 事件中，设置本界面显示参数 FormShowParameter 的 ListentimerElapsed 属性为 true，开启 TimerElapsed 事件。
否则，不会触发此事件。

代码模板

Java

```java
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import kd.bos.form.FormShowParameter;
import kd.bos.form.events.PreOpenFormEventArgs;
import kd.bos.form.events.TimerElapsedArgs;
import kd.bos.form.plugin.AbstractFormPlugin;

public class TimerElapsedEvent extends AbstractFormPlugin {

    @Override
    public void preOpenForm(PreOpenFormEventArgs e) {
        super.preOpenForm(e);
        // 要求触发TimerElapsed事件
        ((FormShowParameter)e.getSource()).setListentimerElapsed(true);
    }

    @Override
    public void TimerElapsed(TimerElapsedArgs e) {
```

```
        // TODO 在此添加业务逻辑
    }
}
```

说明：
必须同时重写 preOpForm 事件，开启 TimerElapsed 事件开关。

参数说明

- `public class TimerElapsedArgs extends EventObject:`
  - ✓ `public Object getSource():` 表单 IFormView

应用示例

暂缺示例。

2.4.22. beforeClosed 事件

事件触发时机

界面关闭之前触发此事件；

插件可以在此事件，取消界面关闭。

代码模板

Java

```
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import kd.bos.form.events.BeforeClosedEvent;
import kd.bos.form.plugin.AbstractFormPlugin;

public class BeforeClosed extends AbstractFormPlugin {

    @Override
```

```
    public void beforeClosed(BeforeClosedEvent e) {
        super.beforeClosed(e);
        // TODO 在此添加业务逻辑
    }
}
```

参数说明

- `public class BeforeClosedEvent extends EventObject:`
  - ✓ `public Object getSource()`: 表单 IFormView
  - ✓ `public void setCancel(boolean cancel)`: 设置 true，取消界面关闭
  - ✓ `public void setCheckDataChange(boolean checkDataChange)`: 设置 true，退出时，不提示数据改变（此属性只在单据表单插件上有效）

应用示例

暂缺示例。

2.4.23. destory 事件

事件触发时机

界面关闭后，系统开始释放表单所占的资源时，触发此事件；
插件可以在此事件，释放插件创建的资源。

此事件比 pageRelease 事件早触发，效果一样；

与 beforeClosed 事件的区别：
1. destory 事件，比 beforeClosed 事件晚触发；
2. beforeClosed 是界面准备关闭前触发，各种界面资源还存在；
3. destory 事件触发时，表单上下文环境可能已经被销毁，因此，不要在此事件中试图访问表单信息；

代码模板

```Java
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import kd.bos.form.plugin.AbstractFormPlugin;

public class DestoryEvent extends AbstractFormPlugin {

    @Override
    public void destory() {
        super.destory();
        // TODO 在此添加业务逻辑
    }
}
```

参数说明

无参数

应用示例

暂缺示例。

2.4.24. pageRelease 事件

事件触发时机

界面关闭后，系统开始释放表单所占的资源时，触发此事件；
插件可以在此事件，释放插件创建的资源。

本事件触发比 destory 稍晚，效果一样

代码模板

```Java
package kd.bos.plugin.sample.dynamicform.pcform.form.template;

import java.util.EventObject;

import kd.bos.form.plugin.AbstractFormPlugin;

public class PageRelease extends AbstractFormPlugin {

    @Override
    public void pageRelease(EventObject e) {
        // TODO 在此添加业务逻辑
    }
}
```

参数说明

- `public class EventObject implements java.io.Serializable:`
  - ✓ `public Object getSource():` 表单 IFormView

应用示例

暂缺示例。

2.4.25. onCreateDynamicUIMetas 事件

代码模板

```Java
package kd.form.demo.plugin;

import kd.bos.form.events.OnCreateDynamicUIMetasArgs;
import kd.bos.form.plugin.AbstractFormPlugin;
```

```
public class FormDemoPlugin extends AbstractFormPlugin{

    @Override
    public void onCreateDynamicUIMetas(OnCreateDynamicUIMetasArgs e) {
        // TODO Auto-generated method stub
        super.onCreateDynamicUIMetas(e);
    }
}
```

示例

待更新

2.4.26. contextMenuClick 事件

代码模板

Java

```
package kd.form.demo.plugin;

import kd.bos.form.events.ContextMenuClickEvent;
import kd.bos.form.plugin.AbstractFormPlugin;

public class FormDemoPlugin extends AbstractFormPlugin{

    @Override
    public void contextMenuClick(ContextMenuClickEvent e) {
        // TODO Auto-generated method stub
        super.contextMenuClick(e);
    }
}
```

示例

待更新

2.5. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问

https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=02 进行反馈。

3. 移动端表单插件

手机等移动端连接到下一代金蝶云，打开的表单，需要特别设计，以适应移动端特性：
1. 显示区域小
2. 触控操作，使用区块设计更合适

移动端表单界面插件开发，与动态表单一致，但增加了一些新特性与事件，以及一些特别用于移动端的控件。

本章介绍移动端表单插件开发，移动端控件请参阅控件与字段相关章节；

3.1. 插件基类

3.1.1. 插件接口及基类

移动端动态表单界面插件接口为 IMobFormPlugin，定义了移动端界面特有的事件方法（见后文）：

Java

```
package kd.bos.form.plugin;
public interface IMobFormPlugin {
```

移动端动态表单界面插件为基类 AbstractMobFormPlugin，派生自动态表单插件基类 AbstractFormPlugin，并实现了移动端界面插件接口 IMobFormPlugin：

Java

```
package kd.bos.form.plugin;
public class AbstractMobFormPlugin extends AbstractFormPlugin implements IMobFormPlugin {
```

3.1.2. 创建并注册插件

请参阅动态表单创建与注册插件的介绍。

3.2. 视图模型

3.2.1. 接口与实现类

移动表单的视图模型接口为 IMobileView，派生自 IFormView：

Java

```
package kd.bos.form;
public interface IMobileView extends IFormView {
```

移动表单的视图模型实现类为 MobileFormView，派生自动态表单视图模型实现类 FormView，并实现了接口 IMobileView：

Java

```
package kd.bos.mvc.form;
public class MobileFormView extends FormView implements IMobileView {
```

3.2.2. 功能方法及使用

IMobileView 增加如下公共方法：

方法	说明
getCurrentCityId	获取用户所在的城市； 使用手机获取用户所在的地图坐标，定位城市
setLocation	设置地图坐标
setMenuItemVisible	设置菜单可见性
upload	上传文件

3.3. 数据模型

移动表单的数据模型，与 PC 端动态表单数据模型完全一致，不重复描述。

3.4. 插件事件

移动表单，除了动态表单界面插件事件，还支持如下插件事件：

事件方法	事件所在接口	触发时机
uploadFile	IMobFormPlugin	暂未发现触发时机
locate	IMobFormPlugin	接收当前位置信息

3.4.1. uploadFile 事件

暂未发现触发时机。

3.5. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=03 进行反馈。

4. 单据界面插件

单据派生自动态表单，与物理表格直接关联，记录、显示流水数据。
系统预置了保存、删除等单据操作，实现单据数据存取。

单据界面，继承了动态表单界面全部功能，包括视图模型、数据模型以及插件事件，并为这些模型及插件增加了新特性，并封装了一批用于单据的字段；

本章仅介绍单据界面模型、插件的新特性。
在阅读本章之前，建议先阅读动态表单章节，了解单据可以从动态表单继承的功能；
请参阅控件与字段相关章节，了解适用于单据的字段；

单据业务对象，还有单据列表、单据操作、单据转换、反写等应用场景，也支持插件开发，这些应用场景的插件开发，在后续章节分别介绍。

4.1. 插件基类

4.1.1. 插件接口及基类

系统为单据界面，新增了单据界面插件事件接口 IBillPlugin，派生自动态表单界面插件事件接口，增加了只适用于单据的插件事件（具体增加的事件，请参阅后文）：

Java

```
package kd.bos.bill;
public interface IBillPlugin extends IFormPlugin {
```

单据界面插件基类为 AbstractBillPlugIn，继承自动态表单界面插件基类 AbstractFormPlugin，实现了新增加的单据界面插件接口 IBillPlugin：

Java

```
package kd.bos.bill;
public class AbstractBillPlugIn extends AbstractFormPlugin implements IBillPlugin  {
```

4.1.2. 创建并注册插件

注册单据插件方式，与注册动态界面插件一致，只是插件基类必须改为 AbstractBillPlugIn。

4.2. 视图模型

4.2.1. 接口与实现类

单据视图模型接口为 IBillView，派生自动态表单视图模型接口 IFormView，增加了少量仅用于单据界面的控制方法（见后文介绍）：

Java

```
package kd.bos.bill;
public interface IBillView extends IFormView {
```

单据视图模型实现类为 BillView，派生自动态表单视图模型实现类 FormView，实现了单据视图模型接口 IBillView，并重写了部分动态表单视图模型的方法：

Java

```
package kd.bos.mvc.bill;
```

```
public class BillView extends FormView implements IBillView, IConfirmCallBack {
```

单据插件，可以直接把插件 getView()方法返回的实例，转为 IBillView 的实例：

```
Java
IBillView billView = (IBillView)this.getView();
```

4.2.2. 功能方法及使用

单据视图模型接口 IBillView，派生自动态表单视图模型接口 IFormView，建议先阅读动态表单章节，了解 IFormView 接口提供的方法。

IBillView 新增加了如下方法：

方法	说明
setBillStatus	设置单据界面编辑状态：新增、修改、查看、提交、审核等； 动态表单界面，没有这些状态的划分；
updateViewStatus	根据界面编辑状态，刷新控件、字段的可见性、锁定性

4.3. 数据模型

4.3.1. 接口与实现类

单据界面数据模型接口为 IBillModel，派生自动态表单界面数据模型接口 IDataModel，增加了单据特用的数据模型控制方法：

```
Java
package kd.bos.entity.datamodel;
public interface IBillModel extends IDataModel{
```

单据数据模型实现类为 BillModel，派生自动态表单数据模型实现类 FormDataModel，新增了单据模型接口 IBillModel：

```
Java
package kd.bos.mvc.bill;
public class BillModel extends FormDataModel implements IBillModel {
```

单据插件，可以直接把插件 getModel()方法返回的实例，转为 IBillModel 的实例：

```
Java
```

```
IBillModel billModel = (IBillModel)this.getModel();
```

4.3.2. 功能方法及使用

单据数据模型接口 IBillModel，派生自动态表单数据模型接口 IDataModel，新增加单据数据控制方法：

方法	说明
getPKValue	获取当前单据主键
setPKValue	设置当前单据的主键
load	指定主键，加载单据
push	传入下推生成的单据

4.3.3. 主实体模型 BillEntityType

单据的主实体模型是 BillEntityType，派生自动态表单主实体模型 MainEntityType，增加了单据特别属性：

Java

```
package kd.bos.entity;
public class BillEntityType extends MainEntityType {
```

新增加了如下方法：

方法	说明
getBillNo	单据编号字段标识
getMainOrgProperty	主业务组织
getBillStatus	数据状态字段标识
getBillType	单据类型字段标识
getForbidStatus	禁用状态字段标识（专用于基础资料）
getMobFormId	绑定的移动单据界面标识
getBillParameter	自定义的单据参数界面标识

4.4. 插件事件

单据界面插件，支持动态表单界面插件全部事件，并实现了单据插件事件接口 IBillPlugin。

新增加的插件事件，触发时机及顺序说明如下：

分类	事件	触发时机
界面初始化	afterLoadData	加载单据数据后，触发此事件

4.4.1. afterLoadData 事件

事件触发时机

单据界面加载完毕，会根据传入的单据主键值，到数据库加载单据数据包。

本事件在单据数据包加载完毕后触发。

插件可以在此事件，根据单据的数据决定业务逻辑的执行。

afterLoadData 事件与 afterCreateNewData 的差别：

- 这两个事件的触发时机一样，都是在界面数据包准备完毕之后触发，但数据包来源不同：
  - ✓ afterCreateNewData 事件的数据包，是全新创建的数据包，适合在此事件调整默认值；
  - ✓ afterLoadData 事件的数据包，来自于数据库，不建议在此事件修改字段值；
- 这两个事件，在单据界面加载、刷新过程中，只会触发其中一个（互斥）：
  - ✓ 单据界面以新增模式打开，不需要到数据库加载单据，触发 afterCreateNewData 事件；
  - ✓ 单据界面以修改、查看模式打开，需到数据库加载单据，触发 afterLoadData 事件；

代码模板

Java

```java
package kd.bos.plugin.sample.bill.pcform.template;

import java.util.EventObject;

import kd.bos.bill.AbstractBillPlugIn;

public class AfterLoadData extends AbstractBillPlugIn {

    @Override
    public void afterLoadData(EventObject e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `EventObject e`：通用的事件参数对象，含有事件源
  - ✓ `Object getSource()`：事件源，单据表单视图模型 IBillView 实例

示例

暂缺示例。

4.5. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=04 进行反馈。

5. 基础资料界面插件

基础资料派生自单据，用于存储会被重复引用的数据。基础资料继承了单据的功能，插件开发与单据一致。

5.1. 插件基类

5.1.1. 插件接口及基类

基础资料插件基类为 AbstractBasePlugIn，继承自单据界面插件基类 AbstractBillPlugIn

Java

```
package kd.bos.bill;
public class AbstractBasePlugIn extends AbstractBillPlugIn {
}
```

5.1.2. 创建并注册插件

注册基础资料插件方式，与注册单据界面插件一致。

5.2. 数据模型

5.2.1. 接口与实现类

基础资料数据模型接口为 IBaseModel，派生自动态表单界面数据模型接口 IBillModel。

Java

```
package kd.bos.entity.datamodel;
public interface IBaseModel extends IBillModel {
}
```

5.3. 插件事件

基础资料继承了单据的功能，插件开发与单据一致。详见单据开发。

5.4. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=05 进行反馈。

6. 移动端基础资料插件

移动端基础资料继承了移动端单据的功能，插件开发与移动端单据一致。

6.1. 插件基类

6.1.1. 插件接口及基类

移动端基础资料插件基类为 AbstractMobBasePlugIn，继承自移动端单据界面插件基类 AbstractMobBillPlugIn

Java

```
package kd.bos.base;
import kd.bos.bill.AbstractMobBillPlugIn;
public class AbstractMobBasePlugIn extends AbstractMobBillPlugIn {
```

6.1.2. 创建并注册插件

注册移动端基础资料插件方式，与注册移动端单据界面插件一致。

6.2. 数据模型

6.2.1. 接口与实现类

待更新

6.3. 插件事件

移动端基础资料继承了移动端单据的功能，插件开发与单据一致。详见单据开发。

6.4. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=06 进行反馈。

7. 标准单据列表插件

单据列表界面，是个动态表单，包含过滤面板、列表控件，在列表控件中，拉平显示单据数据。

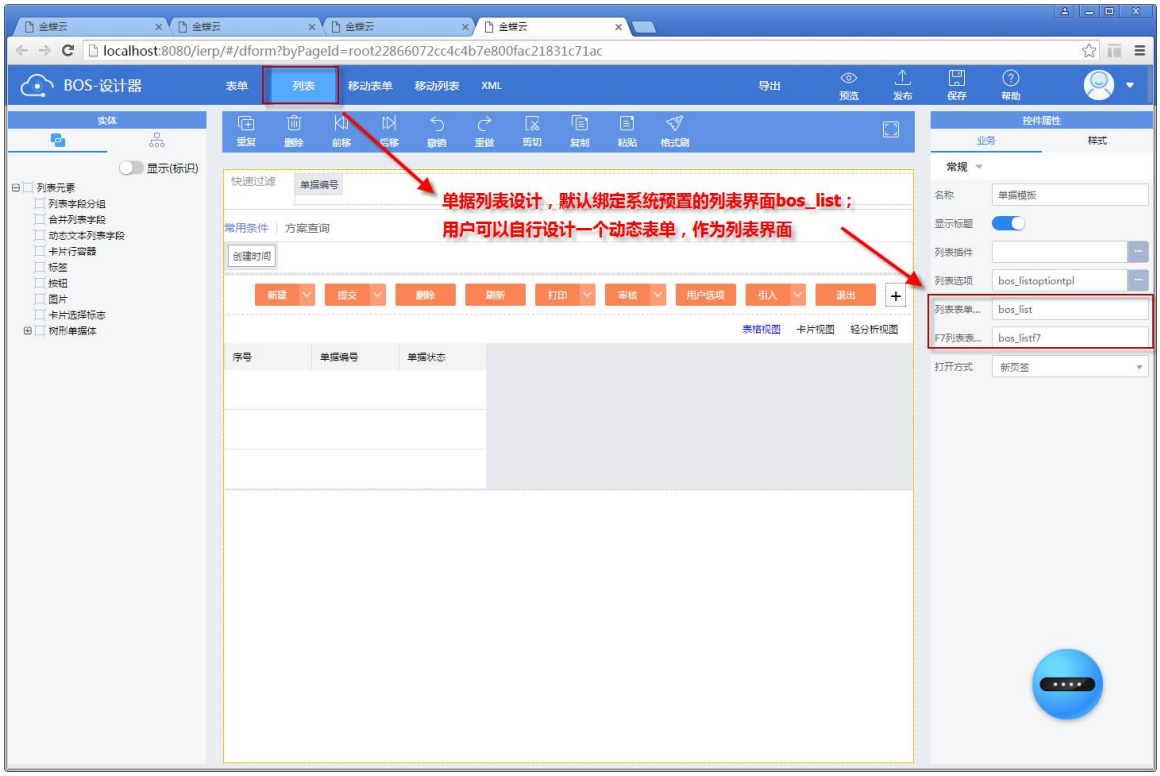
在列表界面加载、过滤取数、显示的过程中，除了会触发动态表单的界面事件之外，还会触发特别定义的单据列表插件事件。

系统预置了两种单据列表界面：

- 标准单据列表，默认绑定到了单据上
  - ✓ 普通列表界面：bos_list
  - ✓ F7 选择列表界面：bos_listf7
- 左树右表单据列表，默认绑定到有分组的基础资料上
  - ✓ 普通列表界面：bos_treelist
  - ✓ F7 选择列表界面：bos_treelistf7

如有必要，用户完全可以自定义一个含有单据列表控件的动态表单，绑定到单据上，作为单据的列表界面。（参阅单据列表控件、过滤控件两种控件的使用介绍）

图一：单据默认绑定预置的列表界面 bos_list



系统预置的标准单据列表界面 bos_list，分为上下两个部分，上部分为过滤控件，下部分为列表控件：

7.1. 插件基类

7.1.1. 插件接口及基类

标准单据列表界面插件基类为 AbstractListPlugin，派生自动态表单界面插件基类，能够支持表单的各种事件。
另外还实现了 ListRowClickListener, IListPlugin 接口，即表格行点击插件事件、单据列表插件事件。

单据列表界面插件基类定义：

Java

```
package kd.bos.list.plugin;
public class AbstractListPlugin extends AbstractFormPlugin implements ListRowClickListener,
IListPlugin {
```

单据列表界面插件事件接口为 IListPlugin，派生自动态表单界面插件接口 IFormPlugin，增加列表行交互等事件：

Java

```
package kd.bos.list.plugin;
public interface IListPlugin extends IFormPlugin {
```

7.1.2. 创建并注册插件

自定义单据列表界面插件，需要扩展 AbstractListPlugin，如下例：

Java

```
package kd.bos.plugin.sample.bill.list.bizcase;

import kd.bos.list.plugin.AbstractListPlugin;

public class ListEventSample extends AbstractListPlugin {
}
```

打开单据设计器，切换到列表设计页签，注册单据列表插件。

7.2. 视图模型

7.2.1. 接口与实现类

插件可以通过列表视图模型控制列表界面。

单据列表界面，提供了两个层次的视图模型：

- 表单视图模型 IFormView：据此访问表单各种信息；
- 列表视图模型 IListView：据此访问单据列表控件的各种信息；

列表视图模型 IListView，派生自表单视图模型 IFormView，增加了列表访问方法：

Java

```
package kd.bos.list;
public interface IListView extends IFormView {
```

在列表界面插件(AbstractListPlugin)中，通过 this.getView()，返回的是 ListView 实例，该实例即可以转为 IFormView 接口，也可以转为 IListView 接口：

Java

```
package kd.bos.plugin.sample.bill.list.bizcase;

import kd.bos.form.IFormView;
import kd.bos.list.IListView;
import kd.bos.list.plugin.AbstractListPlugin;

public class ListViewSample extends AbstractListPlugin {

    private void getListView(){
        IFormView formview = this.getView();
        IListView listview = (IListView)this.getView();
    }
}
```

7.2.2. 功能方法及使用

列表视图模型接口 IListView，在动态表单视图模型接口 IFormView 基础上，新增加如下方法：

方法	说明
getBillFormId	获取列表界面绑定的单据
setBillFormId	设置列表界面绑定的单据 内部方法，插件请勿调用

getListModel	获取列表数据模型
getFocusRow	获取当前焦点行号，整数
getFocusRowPkId	获取当前焦点行显示的单据数据内码
getCurrentSelectedRowInfo	获取当前焦点行数据详情，明细到单据体行、子单据体行
getSelectedRows	获取当前勾选了的全部行数据详情
getCurrentListAllRowCollection	获取列表全部行数据详情
clearSelection	清除所选的行
returnLookupData	把当前所选行，返回给父界面
refresh	刷新列表，重新取数
getSelectedMainOrgIds	获取选中的主业务组织
setSelectedMainOrgIds	设置选中的主业务组织
export	引出
importData	引入
getTreeListView	获取树形列表视图模型 ITreeListView 仅适用于左树右表列表模式
focusRootNode	树形列表，切换到根节点

7.3. 数据模型

7.3.1. 接口与实现类

单据列表界面，包含了两个层次的数据模型：

- 表单数据模型 IDataModel：据此访问列表控件所在的表单数据；
  - ✓ 列表界面，除了单据列表控件，还可以增加各种自定义字段，需要通过表单数据模型 IDataModel，访问这些自定义字段的值；
- 列表数据模型 IListModel：据此访问单据列表控件中的数据；

表单数据模型 IDataModel，在动态表单章节介绍。

列表数据模型 IListModel，定义如下：

Java

```
package kd.bos.entity.datamodel;
public interface IListModel {
```

表单数据模型 IDataModel 与列表数据模型 IListModel，分别有各自的实现类及实例，没有派生关系，不能互相转换。

单据列表插件(AbstractListPlugin)，可以如下代码，分别获取表单数据模型 IDataModel、列表数据模型 IListModel 的实例：

Java

```
package kd.bos.plugin.sample.bill.list.bizcase;

import kd.bos.entity.datamodel.IDataModel;
import kd.bos.entity.datamodel.IListModel;
import kd.bos.list.IListView;
import kd.bos.list.plugin.AbstractListPlugin;

public class ListModelSample extends AbstractListPlugin {

    private void getListModel(){
        IDataModel formmodel = this.getModel();
        IListModel listmodel = ((IListView)this.getView()).getListModel();
    }
}
```

7.3.2. 功能方法及使用

IListModel 提供的方法如下：

方法	说明
getDataEntityType	
setDataEntityType	
setPageId	
getData	
getQingData	
setEntityId	
getEntityId	
setLookup	
isLookup	
getDataCount	
setFilterParameter	
setListFields	
getProvider	
setProvider	
getPkFields	
setPkFields	
setFieldCotnrolRules	
getQueryResult	
getRegisterPropertyListeners	
setRegisterPropertyListeners	
getSelectFields	

setSelectFields

7.4. 插件事件

单据列表界面插件，支持动态表单界面插件全部事件，并新增了 IListPlugin、ListRowClickListener、IRegisterPropertyListener 插件接口。

新增加的插件事件，触发时机及顺序说明如下：

分类	事件	触发时机
列表初始化	createTreeListView	不触发 仅树形列表界面，会触发此事件；
	filterContainerInit	初始化过滤控件时，触发此事件 把配置的过滤字段传递给插件
	beforeCreateListColumns	在构建列表列之前触发，传入当前待创建的列分组、列
	beforeCreateListDataProvider	列表初始化，构建列表取数器前触发此事件
	setFilter	列表在生成了过滤条件之后，准备查询数据之前，触发此事件
用户交互	filterContainerSearchClick	用户修改了快捷过滤、常用过滤条件，或者点击过滤面板确定按钮，触发此事件
	beforeItemClick	用户点击列表主菜单按钮时，触发此事件
	itemClick	用户点击列表主菜单按钮时，触发此事件
	billListHyperLinkClick	点击超链接单元格时，触发此事件
	beforeShowBill	执行新建、修改、查看等操作，打开单据维护界面之前，触发此事件
	billClosedCallBack	列表打开的单据维护界面关闭后返回时，触发此事件
	listRowClick	列表行点击时，触发此事件
	listRowDoubleClick	列表行双击时，触发此事件
	filterContainerAfterSearchClick	过滤容器搜索点击后的处理方法,此事件发生在过滤条件解析后，主要用于点击过滤条件时联动修改其他过滤字段控件，修改前已先在 filterContainerInit 事件将要修改的字段用全局变量进行缓存
	filterContainerBeforeF7Select	过滤容器内 F7 弹出前的处理方法
	filterColumnSetFilter	过滤字段上的基础资料字段过滤条件调整事件
	baseDataColumnDependFieldSet	设置常用过滤的基础资料依赖字段
	setCellFieldValue	设置单元格指令

7.4.1. filterContainerInit 事件

事件触发时机

本事件，有两次触发时机：
1. 列表界面，初始化过滤面板时，触发此事件；
2. 用户在过滤面板点击搜索时，也会重新初始化过滤面板，触发此事件；

插件可以在此事件，获取到列表过滤面板中，快捷过滤、常用过滤、方案过滤所包含的过滤字段信息，并对这些过滤字段进行调整；

特别注意：
本事件会多次触发，如果逻辑只能在界面初始化时执行一次，需要增加标志变量，进行区分；

代码模板

Java

```
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.form.events.FilterContainerInitArgs;
import kd.bos.list.plugin.AbstractListPlugin;

public class FilterContainerInit extends AbstractListPlugin {

    @Override
    public void filterContainerInit(FilterContainerInitArgs args) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class FilterContainerInitArgs`
  - ✓ `public FilterContainerInitEvent getFilterContainerInitEvent()`
    - ◆ `public Object getSource()`：事件源，过滤控件 FilterContainer 实例
    - ◆ `public List<FilterColumn> getFastFilterColumns()`：快捷过滤字段
    - ◆ `public List<FilterColumn> getCommonFilterColumns()`：常用过滤字段

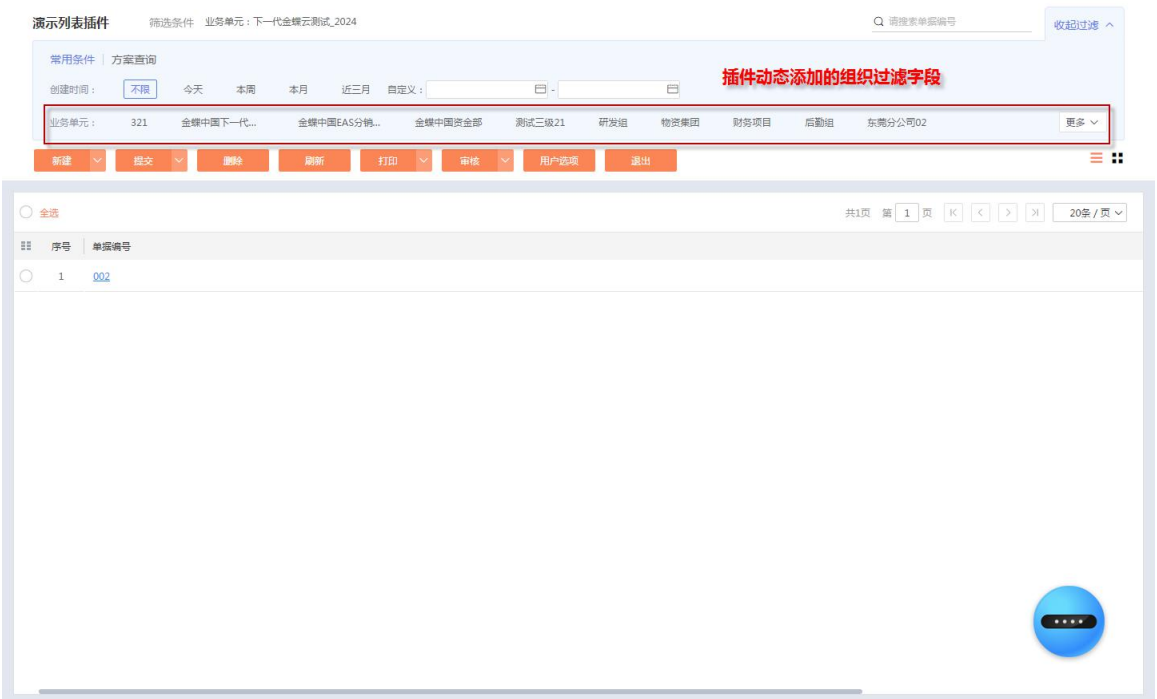
- ◆ `public List<FilterColumn> getSchemeFilterColumns()`：方案过滤字段

示例

- ➢ 案例说明：
  - ✓ 在常用过滤面板，增加组织过滤字段
  - ✓ 默认取当前组织，对单据进行过滤；
  - ✓ 如果用户在过滤面板中，设置了组织值，则按照用户设置的组织过滤单据

- ➢ 实现方案：
  - ✓ 捕获列表 filterContainerInit 事件，过滤面板初始化事件
    - ◆ 确保过滤面板在设计时，添加了组织过滤字段；如果设计时没有添加，则自动创建
    - ◆ 取当前用户已授权组织，设置到组织过滤字段上，作为可选组织
    - ◆ 取当前登录组织值，放在本地变量中，后续默认按此组织过滤
  - ✓ 捕获 filterContainerSearchClick 事件，搜索按钮点击事件
    - ◆ 取用户设置的过滤组织，放在本地变量中，后续按此组织过滤；
    - ◆ 从条件配置中，移除过滤组织，避免系统自动生成了组织过滤条件；
  - ✓ 捕获 setFilter 事件，列表过滤条件生成后触发；
    - ◆ 取前述事件中，放在本地变量中的组织值，生成过滤条件，传递给列表
  - ✓ 捕获 beforeF7Select 事件，设置自定义组织过滤字段，F7 显示的基础资料

- ➢ 运行效果



➢ 实例代码：

Java

```
package kd.bos.plugin.sample.bill.list.bizcase;

import java.util.ArrayList;
import java.util.EventObject;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

import kd.bos.context.RequestContext;
import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.dataentity.entity.DynamicObjectCollection;
import kd.bos.dataentity.entity.LocaleString;
import kd.bos.dataentity.serialization.SerializationUtils;
import kd.bos.dataentity.utils.StringUtils;
import kd.bos.entity.BillEntityType;
import kd.bos.filter.CommonFilterColumn;
import kd.bos.filter.FilterColumn;
import kd.bos.filter.FilterContainer;
import kd.bos.form.events.FilterContainerInitArgs;
import kd.bos.form.events.FilterContainerSearchClickArgs;
import kd.bos.form.events.SetFilterEvent;
import kd.bos.form.field.ComboItem;
import kd.bos.form.field.events.BeforeFilterF7SelectEvent;
import kd.bos.form.field.events.BeforeFilterF7SelectListener;
import kd.bos.list.IListView;
import kd.bos.list.ListShowParameter;
import kd.bos.list.plugin.AbstractListPlugin;
import kd.bos.orm.ORM;
import kd.bos.orm.query.QFilter;
import kd.bos.servicehelp.permission.PermissionServiceHelper;
import kd.bos.servicehelper.basedata.BaseDataServiceHelper;
import kd.bos.servicehelper.model.PermissionStatus;

public class FilterContainerInitSample extends AbstractListPlugin implements
BeforeFilterF7SelectListener {

    /** 缓存标识 */
    private final static String CACHEKEY_ORGCOMBOITEMS = "orgcomboitems";
    /** 组织主实体标识 */
```

```
    private final static String ORG_ENTITY = "bos_org";

    /** 用户是否点击了过滤面板搜索按钮 */
    private boolean isClickSearch = false;
    /** 筛选组织：优先按用户在过滤面板中，设置的组织筛选数据；列表初始化时，按当前组织筛选数据 */
    private long orgId = 0;

    @Override
    public void registerListener(EventObject e) {
        super.registerListener(e);

        // 侦听过滤面板上，过滤字段F7点击事件：设置自定义组织过滤字段，F7打开的基础资料列表
        FilterContainer filterContainer = this.getView().getControl(FILTERCONTAINERID);
        filterContainer.addBeforeF7SelectListener(this);
    }

    /**
     * 1. 界面初始化时，触发此事件；
     * 2. 用户在过滤面板上，点击搜索时，也触发此事件
     * @param args
     * @remark
     * 1. 给常用过滤面板，增加组织过滤字段
     * 2. 取默认过滤组织
     */
    @Override
    public void filterContainerInit(FilterContainerInitArgs args) {

        // F7列表，不显示常用过滤，无需修改过滤字段信息
        ListShowParameter listShowParameter =
(ListShowParameter)this.getView().getFormShowParameter();
        if(listShowParameter.isLookUp()){
            return;
        }

        if (!this.isBillHasMainOrg()){
            // 列表绑定的单据，没有主业务单元字段：无需增加组织过滤字段
            return;
        }

        // 创建一个业务单元过滤字段，参与常用过滤
        CommonFilterColumn orgColumn = this.buildOrgFilterColumn();

        // 取到全部常用过滤字段
        List<FilterColumn>  filterColumnList =
```

```
args.getFilterContainerInitEvent().getCommonFilterColumns();

        // 如果常用过滤面板中，已经存在业务单元字段，不再重复添加（按FieldName匹配）
        if(!filterColumnList.contains(orgColumn)) {
            filterColumnList.add(orgColumn);

            // 界面初始化时，取登录组织，放在本地变量中（后续在setFilter事件据此构建过滤条件）
            if (!this.isClickSearch){
                this.orgId = RequestContext.get().getOrgId();

                // 判断用户有没有获得当前组织的授权
                Set<Long> hasPermOrgs = new HashSet<>();
                if (orgColumn.getComboItems() != null){
                    for(ComboItem item : orgColumn.getComboItems()){
                        hasPermOrgs.add(Long.valueOf(item.getId()));
                    }
                }
                if (!hasPermOrgs.contains(RequestContext.get().getOrgId())){
                    this.orgId = hasPermOrgs.iterator().next();
                }
            }
            // 设置组织过滤字段的选中值
            orgColumn.setDefaultValue(String.valueOf(this.orgId));
        }
    }

    /**
     * 用户在列表界面，点击过滤面板搜索按钮时，触发此事件
     * @remark
     * 1. 取用户选择的组织值，放在本地变量
     */
    @Override
    public void filterContainerSearchClick(FilterContainerSearchClickArgs args) {

        this.isClickSearch = true;

        // 列表绑定的单据，没有主业务单元字段：无需按组织过滤
        if (!this.isBillHasMainOrg()){
            return;
        }
        String orgFldName = this.getBillEntityType().getMainOrg() + ".id";

        // 获取用户在过滤面板中设置的过滤条件：从中搜索出过滤组织值
        Map<String, List<Map<String, List<Object>>>> filterValues =
```

```
args.getSearchClickEvent().getFilterValues();
        List<Map<String, List<Object>>> customFiterList = filterValues.get("customfilter");
        if(customFiterList == null) return ;

        // 先清除本地变量值
        this.orgId = 0;

        // 逐项条件匹配，找出自定义的组织过滤字段
        for(int i = customFiterList.size()-1; i >= 0 ; i--){

            Map<String, List<Object>> customFiter = customFiterList.get(i);
            List<Object> fieldNames = customFiter.get("FieldName");
            if(fieldNames == null || fieldNames.isEmpty()) continue;

            if(StringUtils.equals(orgFldName, (String)fieldNames.get(0))){
                // 找到了自定义的组织过滤字段

                List<Object> orgIds = customFiter.get("Value");
                if(orgIds == null || orgIds.isEmpty()) continue;

                // 取用户在过滤面板上，选择的组织，放在本地变量中（后续在setFilter事件中要用到）
                this.orgId = Long.parseLong((String)orgIds.get(0));

                // 从条件集合中，移除组织值：
                // 避免由系统自动拼接组织过滤条件，改由本插件生成组织过滤条件
                customFiterList.remove(customFiter);

                break;
            }
        }
        super.filterContainerSearchClick(args);
    }



    /**
     * 在开始对列表数据进行过滤取数前，触发此事件
     * @remark
     * 1. 使用本地组织值，生成列表过滤条件，添加到列表过滤条件中
     */
    @Override
    public void setFilter(SetFilterEvent e) {

        if (this.orgId == 0){
```

```
            return;
        }

        // 调用基础封装的帮助类，生成组织筛选条件：
        // 返回的条件，可能包含了数据授权信息
        BaseDataServiceHelper helper = new BaseDataServiceHelper();
        QFilter qfilter = helper.getBaseDataFilter(this.getBillEntityId(), this.orgId);

        if(qfilter != null){
            List<QFilter> filters = e.getQFilters();
            filters.add(qfilter);
            e.setQFilters(filters);
        }
        super.setFilter(e);
    }

    /**
     * 过滤面板，基础资料字段，点击F7时触发此事件
     * @remark
     * 1. 如果点击的是自定义的组织过滤字段，则自行构建组织F7列表显示参数
     */
    @Override
    public void beforeF7Select(BeforeFilterF7SelectEvent arg0) {

        // 列表绑定的单据，没有主业务单元字段：无需按组织过滤
        if (!this.isBillHasMainOrg()){
            return;
        }
        String orgFldName = this.getBillEntityType().getMainOrg() + ".id";
        if (StringUtils.equals(orgFldName, arg0.getFieldName())){
            arg0.setRefEntityId(ORG_ENTITY);
            arg0.setRefPropKey("id");
        }
    }

    /**
     * 获取列表绑定的单据类型
     * @return
     */
    private String getBillEntityId(){
        return  ((IListView)this.getView()).getEntityId();
    }

    /**
```

```
     * 获取列表绑定的单据主实体
     * @return
     */
    private BillEntityType getBillEntityType(){
        return
(BillEntityType)((IListView)this.getView()).getListModel().getDataEntityType();
    }

    /**
     * 获取列表绑定的单据是否有主业务单元字段
     * @return
     */
    private boolean isBillHasMainOrg(){
        return StringUtils.isNotBlank(this.getBillEntityType().getMainOrg());
    }

    /**
     * 构建组织过滤字段
     * @return
     */
    private CommonFilterColumn buildOrgFilterColumn(){

        String mainOrgKey = this.getBillEntityType().getMainOrg();  // 主业务字段
        CommonFilterColumn orgColumn = new CommonFilterColumn();
        orgColumn.setKey(mainOrgKey + ".id");
        orgColumn.setCaption(new LocaleString("业务单元"));
        orgColumn.setFieldName(mainOrgKey + ".id");
        orgColumn.setMustInput(true);

        // 获取登录用户，有查询权的组织
        List<ComboItem> combos = this.buildOrgComboItems();

        // 设置业务单元过滤字段的可选项
        orgColumn.setComboItems(combos);
        orgColumn.setType("enum");

        return orgColumn;
    }

    /**
     * 构建可选的组织选项
     * @return
     */
    @SuppressWarnings("unchecked")
```

```
    private List<ComboItem> buildOrgComboItems(){
        // 尝试取缓存的过滤字段
        String cacheString = this.getPageCache().get(CACHEKEY_ORGCOMBOITEMS);
        if (StringUtils.isNotBlank(cacheString)){
            // 已缓存：把缓存的字符串，反序列化为过滤字段
            return (List<ComboItem>)SerializationUtils.fromJsonStringToList(cacheString,
ComboItem.class);
        }

        // 获取登录用户，有查询权的组织
        List<ComboItem> combos = new ArrayList<>();
        long userId = Long.parseLong(RequestContext.get().getUserId());
        List<Long> hasPermOrgs = PermissionServiceHelper.getAllPermissionOrgs(userId,
this.getBillEntityId(), PermissionStatus.View);

        // 取这些组织的内码、名称
        if (!hasPermOrgs.isEmpty()){
            ORM orm =ORM.create();
            DynamicObjectCollection col = orm.query(ORG_ENTITY, "id,name", new QFilter[]{new
QFilter("id", "in", hasPermOrgs)});
            for(DynamicObject org : col){
                if(org == null){
                    continue;
                }
                ComboItem item = new ComboItem();
                item.setId(String.valueOf(org.getPkValue()));
                item.setCaption(new LocaleString(org.getString("name")));
                item.setValue(String.valueOf(org.getPkValue()));
                combos.add(item);
            }
        }

        // 压入缓存
        cacheString = SerializationUtils.toJsonString(combos);
        this.getPageCache().put(CACHEKEY_ORGCOMBOITEMS, cacheString);

        return combos;
    }
}
```

7.4.2. beforeCreateListColumns 事件

事件触发时机

刷新单据列表，构建单据列表显示的列时，触发此事件，传入在设计器中已配置的列集合；

插件可以在此事件中，根据页面参数、过滤条件，动态添加、删除单据列表的显示列；

代码模板

Java

```java
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.form.events.BeforeCreateListColumnsArgs;
import kd.bos.list.plugin.AbstractListPlugin;

public class BeforeCreateListColumns extends AbstractListPlugin {

    @Override
    public void beforeCreateListColumns(BeforeCreateListColumnsArgs args) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class BeforeCreateListColumnsArgs extends EventObject`
  - ✓ `public Object getSource()`：事件源，单据列表控件 BillList
  - ✓ `public List<IListColumn> getListColumns()`：列集合
  - ✓ `public List<ListColumnGroup> getListGroupColumns()`：列分组集合

示例

- 案例说明

1. 根据列表显示自定义参数 state 的值，给列表动态添加列
   a. 如果参数 state =1，动态添加列"文本 1"

b. 如果参数 state =2，动态添加列"文本 2"

c. 文本 1、文本 2，是单据上已有的字段

2. 根据过滤条件中，用户选择的数据状态值，动态添加列：

a. 如果用户选择了已审核状态，则动态添加"审核人.名字"

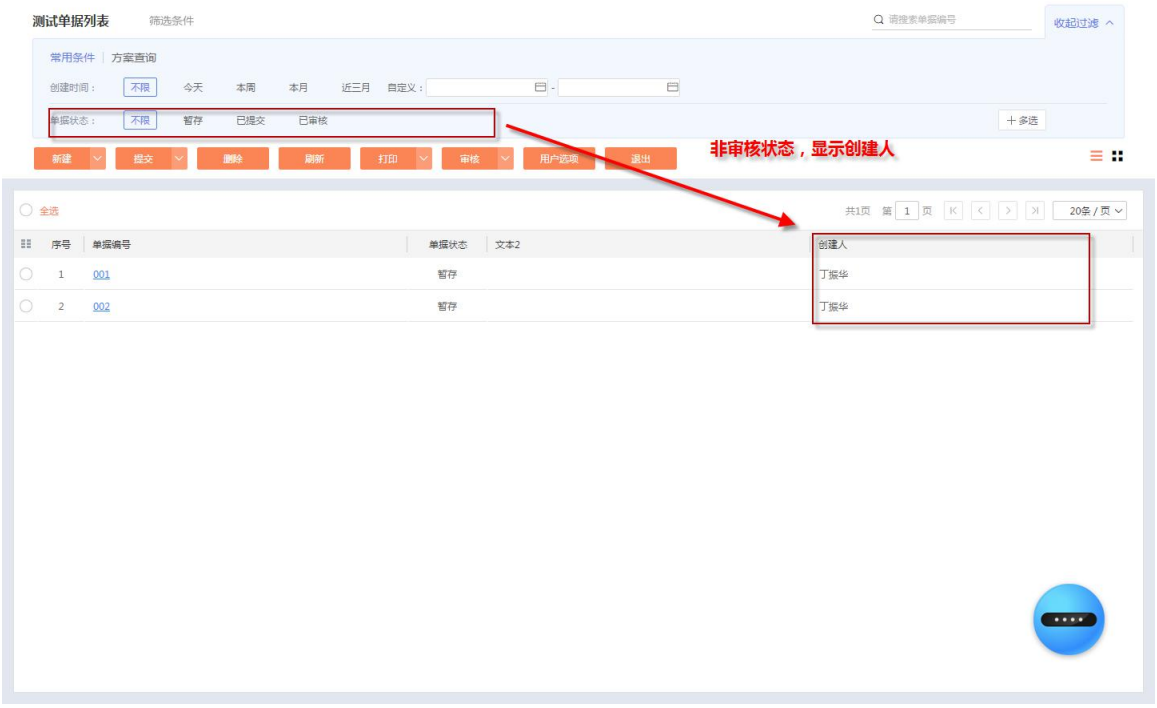
b. 如果用户选择了未审核状态，则动态添加"创建人.名字"

➢ 实现方案

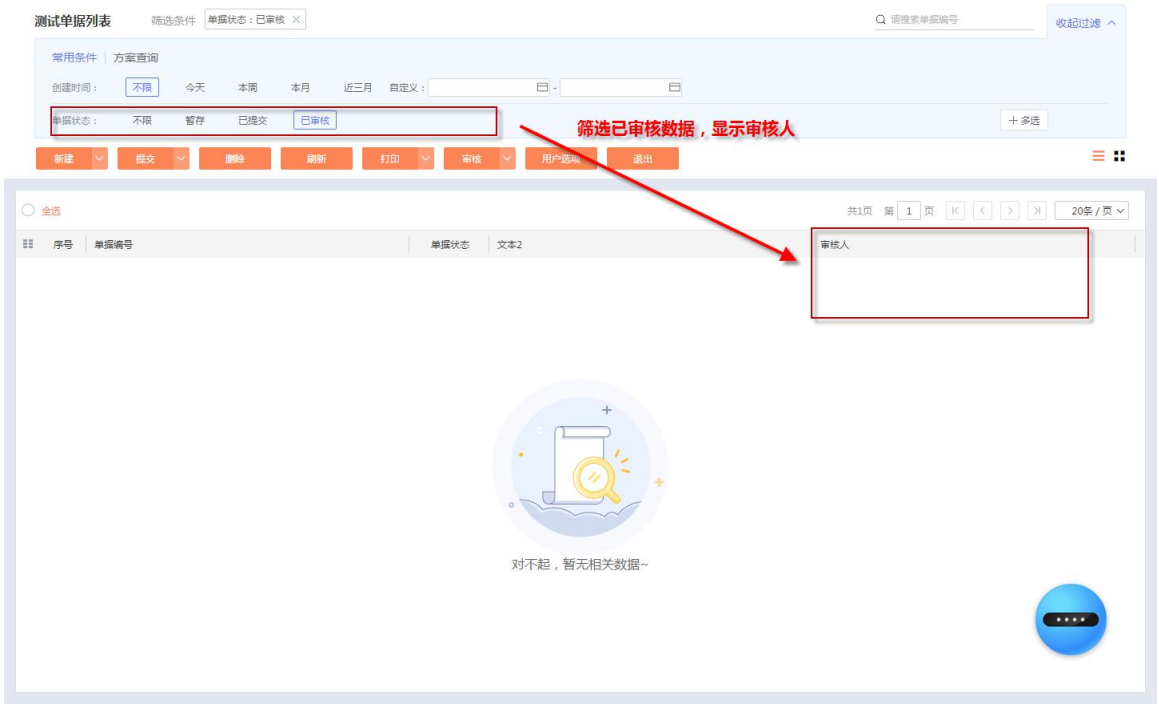
1. 重写 filterContainerSearchClick 方法，获取用户选择的状态值
2. 重写 beforeCreateListColumns 方法，根据自定义参数 state 的值，动态添加列

➢ 运行效果

图一：筛选非审核状态的单据，显示创建人列



图二：筛选审核状态的单据，显示审核人列



➢ 实际代码

Java

```
package kd.bos.plugin.sample.bill.list.bizcase;

import java.util.List;
import java.util.Map;

import kd.bos.dataentity.entity.LocaleString;
import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.events.BeforeCreateListColumnsArgs;
import kd.bos.form.events.FilterContainerSearchClickArgs;
import kd.bos.list.IListColumn;
import kd.bos.list.ListColumn;
import kd.bos.list.plugin.AbstractListPlugin;

public class BeforeCreateListColumnsSample extends AbstractListPlugin {

    /** 用户选择的数据状态过滤值 */
    private String billStateFilterValue;

    /**
     * 用户在过滤条件面板，修改了过滤条件之后，触发此事件
     * @remark
     * 在此事件，获取用户设置的数据状态过滤值
     */
```

```
    @Override
    public void filterContainerSearchClick(FilterContainerSearchClickArgs args) {

        billStateFilterValue = "A";

        // 获取用户在过滤面板中设置的过滤条件
        Map<String, List<Map<String, List<Object>>>> filterValues =
args.getSearchClickEvent().getFilterValues();
        List<Map<String, List<Object>>> customFiterList = filterValues.get("customfilter");
        if(customFiterList == null) return ;

        // 逐项条件匹配，找出数据状态过滤条件
        for(int i = customFiterList.size()-1; i >= 0 ; i--){

            Map<String, List<Object>> customFiter = customFiterList.get(i);

            List<Object> fieldNames = customFiter.get("FieldName");
            if(fieldNames == null || fieldNames.isEmpty()) continue;

            if(StringUtils.equals("billstatus", (String)fieldNames.get(0))){
                // 找到了数据状态过滤条件

                List<Object> values = customFiter.get("Value");
                if(values != null && !values.isEmpty()) {
                    billStateFilterValue = (String) values.get(0);
                }

                break;
            }
        }
    }

    /**
     * 在构建列表显示的列时触发，传入设计时预置的列集合
     * @remark
     * 在此事件，根据自定义参数值，动态添加列
     */
    @Override
    public void beforeCreateListColumns(BeforeCreateListColumnsArgs args) {

        // 设计器预置的列集合
        List<IListColumn> columns = args.getListColumns();

        // 根据自定义参数state的值，动态添加列
```

```
        //int state = 1;
        int state = 2;
        String stateParamValue =
this.getView().getFormShowParameter().getCustomParam("state");
        if (StringUtils.isNotBlank(stateParamValue)){
            state = Integer.valueOf(stateParamValue);
        }

        switch (state) {
        case 1:
            // 动态添加新列：文本1

            ListColumn colText1 = this.createListColumn("textfield1", "文本1", 1);
            columns.add(colText1);

            break;
        case 2:
        default:
            // 动态添加新列：文本2

            ListColumn colText2 = this.createListColumn("textfield2", "文本2", 1);
            columns.add(colText2);

            break;
        }

        // 根据数据状态过滤条件值，动态添加列
        if (StringUtils.equals(billStateFilterValue, "C")){
            ListColumn colUser = this.createListColumn("auditor.name", "审核人", 2);
            columns.add(colUser);
        }
        else {
            ListColumn colUser = this.createListColumn("creator.name", "创建人", 1);
            columns.add(colUser);
        }
    }

    /**
     * 创建列对象返回
     *
     * @param key 列标识，需要显示的字段，如"textfield"、 "basedatafield.name"
     * @param caption 列标题
     * @param colIndex 列顺序
     * @return
```

```
     */
    private ListColumn createListColumn(String key, String caption, int colIndex){

        ListColumn col = new ListColumn();

        col.setCaption(new LocaleString(caption));
        col.setKey(key);
        col.setListFieldKey(key);
        col.setFieldName(key);
        col.setSeq(colIndex);
        col.setVisible(11);

        return col;
    }
}
```

7.4.3. beforeCreateListDataProvider 事件

事件触发时机

构建列表取数器之前，触发此事件；

插件可以在此事件，构建自定义的列表取数器，实现自主取数；

代码模板

Java

```
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.form.events.BeforeCreateListDataProviderArgs;
import kd.bos.list.plugin.AbstractListPlugin;

public class BeforeCreateListDataProvider extends AbstractListPlugin {

    @Override
    public void beforeCreateListDataProvider(BeforeCreateListDataProviderArgs args) {
        // TODO 在此添加业务逻辑
```

```
    }
}
```

事件参数

- `public class BeforeCreateListDataProviderArgs extends EventObject`
  - ✓ `public Object getSource()`：事件源，单据列表控件 BillList
  - ✓ `public void setListDataProvider(IListDataProvider listDataProvider)`：设置列表取数器，IListDataProvider 接口实现类

示例

- 案例说明

1. 由插件自行填写 textfield2 列的值

- 实现方案

1. 扩展 ListDataProvider，自定义列表取数器 MyListDataProvider
2. 捕获 beforeCreateListDataProvider 事件，传入自定义列表取数器
3. 重写列表取数 getData()方法，对系统取出的列表数据，进行调整

- 运行效果

- 实例代码

```
Java
```

```java
package kd.bos.plugin.sample.bill.list.bizcase;

import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.dataentity.entity.DynamicObjectCollection;
import kd.bos.form.events.BeforeCreateListDataProviderArgs;
import kd.bos.list.plugin.AbstractListPlugin;
import kd.bos.mvc.list.ListDataProvider;

public class BeforeCreateListDataProviderSample extends AbstractListPlugin {

    /**
     * 构建列表取数器时，触发此事件；
     * @remark
     * 插件可以在此事件，构建自定义的列表取数器，实现自主取数
     */
    @Override
    public void beforeCreateListDataProvider(BeforeCreateListDataProviderArgs args) {
        args.setListDataProvider(new MyListDataProvider());
    }
}

class MyListDataProvider extends ListDataProvider{

    private final static String KEY_TEXTFIELD2 = "textfield2";

    /**
     * 加载列表数据
     * @remark
     * 获取系统自动加载的列表数据，然后对内容进行修正
     */
    @Override
    public DynamicObjectCollection getData(int arg0, int arg1) {
        DynamicObjectCollection rows = super.getData(arg0, arg1);
        if (rows.isEmpty()){
            return rows;
        }

        if (!rows.get(0).getDataEntityType().getProperties().containsKey(KEY_TEXTFIELD2)){
            // 没有textfield2列，无需处理
            return rows;
        }

        // 自主设置textfield2列的内容
        for(DynamicObject row : rows){
```

```
            row.set(KEY_TEXTFIELD2, "这是插件填写的值");
        }

        return rows;
    }
}
```

7.4.4. setFilter 事件

事件触发时机

单据列表控件，在构建好取数条件，准备重新取数之前，触发此事件；

插件可以调整条件内容、追加条件，从而影响列表取数；

代码模板

Java

```
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.form.events.SetFilterEvent;
import kd.bos.list.plugin.AbstractListPlugin;

public class SetFilter extends AbstractListPlugin {

    @Override
    public void setFilter(SetFilterEvent e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class SetFilterEvent extends EventObject`
  - ✓ `public Object getSource()`：事件源，过滤控件 FilterContainer 实例

- ✓ `public List<QFilter> getQFilters()`：系统根据过滤面板中的条件内容，生成的列表过滤条件集合；可以从此集合移除条件、添加条件；

示例

参阅 filterContainerInit 事件示例。

7.4.5. filterContainerSearchClick 事件

事件触发时机

用户在过滤条件面版上改变过滤条件，触发此事件；

插件可以在此事件，调整用户在过滤面板上的条件配置，后续系统会根据调整后的条件配置，生成取数条件；

代码模板

Java

```java
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.form.events.FilterContainerSearchClickArgs;
import kd.bos.list.plugin.AbstractListPlugin;

public class FilterContainerSearchClick extends AbstractListPlugin {

    @Override
    public void filterContainerSearchClick(FilterContainerSearchClickArgs args) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class FilterContainerSearchClickArgs`
  - ✓ `public SearchClickEvent getSearchClickEvent()`
    - ◆ `public Object getSource()`：事件源，过滤控件 FilterContainer 实例
    - ◆ `public Map<String, List<Object>> getCurrentCommonFilter()`：常用过滤条件
    - ◆ `public Map<String, List<Map<String, List<Object>>>> getFilterValues()`：全部过滤条件，按快捷过滤、常用过滤分开
    - ◆ `public String getEntryEntity()`：实体名
    - ◆ `public IFilterModel getFilterModel()`：过滤面板的数据模型 IFilterModel
    - ◆ `public FilterParameter getFilterParameter()`：基于常用过滤条件，生成的过滤参数对象；

示例

参阅 filterContainerInit 事件示例。

7.4.6. beforeItemClick 事件

事件触发时机

用户点击单据列表主菜单工具栏上的按钮时，触发此事件；

插件可以在此事件中，对列表选中数据进行检查，取消系统预置的按钮操作执行。

代码模板

```
Java
```

```java
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.control.events.BeforeItemClickEvent;
import kd.bos.list.plugin.AbstractListPlugin;

public class BeforeItemClick extends AbstractListPlugin {

    private final static String KEY_BARITEM1 = "baritemap1";

    @Override
    public void beforeItemClick(BeforeItemClickEvent evt) {
        if (StringUtils.equals(KEY_BARITEM1, evt.getItemKey())){
            // TODO 在此添加业务逻辑
        }
    }
}
```

事件参数

- `public class BeforeItemClickEvent extends ItemClickEvent`
  - ✓ `public String getItemKey()`: 按钮标识
  - ✓ `public String getOperationKey()`: 按钮绑定的操作
  - ✓ `public void setCancel(boolean cancel)`: 取消操作

示例

参阅 itemClick 事件示例。

7.4.7. itemClick 事件

事件触发时机

用户点击单据列表主菜单工具栏上的按钮时，触发此事件；

插件可以在此事件中，实现自定义按钮的业务逻辑。

代码模板

Java

```java
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.control.events.ItemClickEvent;
import kd.bos.list.plugin.AbstractListPlugin;

public class ItemClick extends AbstractListPlugin {

    private final static String KEY_BARITEM1 = "baritemap1";

    @Override
    public void itemClick(ItemClickEvent evt) {
        super.itemClick(evt);
        if (StringUtils.equals(KEY_BARITEM1, evt.getItemKey())){
            // TODO 在此添加业务逻辑
        }
    }
}
```

事件参数

- `public class ItemClickEvent extends EventObject`
  - ✓ `public String getItemKey()`: 按钮标识
  - ✓ `public String getOperationKey()`: 按钮绑定的操作

示例

➢ 案例说明

1. 取消修改菜单的功能
2. 用户点击自定义菜单时，显示一个提示消息

➢ 实现方案

1. 捕获 beforeItemClick 事件，如果点击的是修改菜单项，取消后续操作；
2. 捕获 itemClick 事件，如果点击的是自定义菜单项，提示用户

➢ 实例代码

Java

```java
package kd.bos.plugin.sample.bill.list.bizcase;

import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.control.events.BeforeItemClickEvent;
import kd.bos.form.control.events.ItemClickEvent;
import kd.bos.list.plugin.AbstractListPlugin;

public class ItemClickSample extends AbstractListPlugin {

    private final static String KEY_BARITEM_MODIFY = "btnmodify";
    private final static String KEY_BARITEM1 = "baritemap1";

    /**
     * 用户点击主菜单按钮时，触发此事件
     * @remark
     * 插件可以在此事件，检查选中的列表数据，取消按钮绑定操作执行
     */
    @Override
    public void beforeItemClick(BeforeItemClickEvent evt) {
        if (StringUtils.equals(KEY_BARITEM_MODIFY, evt.getItemKey())){
            // 取消修改操作的执行
            this.getView().showMessage("修改菜单的功能，被插件取消了！");
            evt.setCancel(true);
        }
    }

    /**
     * 用户点击主菜单按钮时，触发此事件
     * @remark
     * 插件可以在此事件中，实现自定义按钮的逻辑处理
     */
```

```
    @Override
    public void itemClick(ItemClickEvent evt) {
        if (StringUtils.equals(KEY_BARITEM1, evt.getItemKey())){
            this.getView().showMessage(String.format("您点击了菜单项 %s", evt.getItemKey()));
        }
    }
}
```

7.4.8. billListHyperLinkClick 事件

事件触发时机

单据列表上显示为超链接的单元格，用户点击时，系统默认会打开单据维护界面。
在打开单据界面之前，触发此事件；

插件可以在此事件，判断点击的列，取消单据界面打开，自行显示其它界面；

代码模板

Java

```
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.form.events.HyperLinkClickArgs;
import kd.bos.list.plugin.AbstractListPlugin;

public class BillListHyperLinkClick extends AbstractListPlugin {
    @Override
    public void billListHyperLinkClick(HyperLinkClickArgs args) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class HyperLinkClickArgs`

- ✓ `public void setCancel(boolean isCancel)`：取消后续处理
- ✓ `public HyperLinkClickEvent getHyperLinkClickEvent()`
  - ◆ `public Object getSource()`：事件源，单据列表控件 BillList
  - ◆ `public String getFieldName()`：列名
  - ◆ `public int getPageIndex()`：页码
  - ◆ `public int getRowIndex()`：行号
  - ◆ `public DynamicObject getRowData()`：行数据。可以据此获取到当前单元格的内容，从而决定自行打开的子界面参数

示例

➢ 案例说明

1. 列表上，单据编号、文本 1 两列，均显示为超链接
2. 点击单据编号超链接，打开单据本身的编辑界面；
3. 点击文本 1，打开自定义的表单(如物料新增界面)

➢ 实现方案

1. 捕获 billListHyperLinkClick
   a. 如果当前点击的不是 textfield1，不做干预，由系统自动打开单据本身的界面
   b. 如果当前点击的是 textfield1，取消系统处理，由插件自行打开界面

➢ 实例代码

Java

```java
package kd.bos.plugin.sample.bill.list.bizcase;

import kd.bos.bill.BillShowParameter;
import kd.bos.bill.OperationStatus;
import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.ShowType;
import kd.bos.form.events.HyperLinkClickArgs;
import kd.bos.list.plugin.AbstractListPlugin;

public class BillListHyperLinkClickSample extends AbstractListPlugin {

    private final static String KEY_TEXTFIELD1 = "textfield1";

    /**
     * 用户点击超链接单元格时，触发此事件
     */
    @Override
    public void billListHyperLinkClick(HyperLinkClickArgs args) {
```

```
        if (StringUtils.equals(KEY_TEXTFIELD1,
args.getHyperLinkClickEvent().getFieldName())){
            // 当前点击的是文本1

            // 取消系统自动打开本单的处理
            args.setCancel(true);

            // 打开物料新增界面
            BillShowParameter showParameter = new BillShowParameter();
            showParameter.setFormId("bd_material");
            showParameter.getOpenStyle().setShowType(ShowType.Modal);
            showParameter.setStatus(OperationStatus.ADDNEW);

            this.getView().showForm(showParameter);
        }
    }
}
```

7.4.9. beforeShowBill 事件

事件触发时机

在执行操作时，打开子界面之前，触发此事件；

插件可以修改子界面显示参数，或取消显示；

代码模板

Java

```
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.list.events.BeforeShowBillFormEvent;
import kd.bos.list.plugin.AbstractListPlugin;

public class BeforeShowBill extends AbstractListPlugin {

    @Override
    public void beforeShowBill(BeforeShowBillFormEvent e) {
```

```
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class BeforeShowBillFormEvent extends EventObject`
  - ✓ `public Object getSource()`：事件源，单据列表控件 BillList
  - ✓ `public void setCancel(boolean isCancel)`：取消后续处理
  - ✓ `public BillShowParameter getParameter()`：单据界面显示参数

示例

- 案例说明

1. 在单据列表上，新建、修改、查看单据时，弹出模态窗口（默认是在新页签显示）

- 实现方案

1. 捕获 beforeShowBill 事件，修改界面显示参数中的界面显示风格

- 实例代码

```
Java
```

```
package kd.bos.plugin.sample.bill.list.bizcase;

import kd.bos.form.ShowType;
import kd.bos.list.events.BeforeShowBillFormEvent;
import kd.bos.list.plugin.AbstractListPlugin;

public class BeforeShowBillSample extends AbstractListPlugin {

    /**
     * 在列表上打开单据界面之前，触发此事件
     * @remark
     * 插件可以在此事件，取消界面显示，或者修改界面显示参数
     */
    @Override
    public void beforeShowBill(BeforeShowBillFormEvent e) {
        e.getParameter().getOpenStyle().setShowType(ShowType.Modal);
    }
}
```

7.4.10. billClosedCallBack 事件

事件触发时机

列表打开的单据界面关闭并返回到列表时，触发此事件；

插件可以在此事件，接收单据界面返回的值，进行后续处理；
也可以与双击、超链接等事件处理配合，打开自定义界面，接收自定义界面的返回值

特别说明：

列表界面的 billClosedCallBack 事件，与 closedCallBack 事件（已在表单事件章节介绍）有什么区别？

列表打开的单据界面关闭时，如果设置了回调属性，会先后触发列表插件的 billClosedCallBack、closeCallBack 两个事件；

在 billClosedCallBack 事件，系统会自动把单据界面上的单据内码，传递给事件；
而在 closeCallBack 事件，只能收到单据界面主动返回给列表界面的任意数据；

代码模板

Java

```
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.list.events.BillClosedCallBackEvent;
import kd.bos.list.plugin.AbstractListPlugin;

public class BillClosedCallBack extends AbstractListPlugin {

    @Override
    public void billClosedCallBack(BillClosedCallBackEvent e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class BillClosedCallBackEvent extends EventObject`
  - ✓ `public Object getSource()`：事件源，单据列表控件 BillList
  - ✓ `public Object getPkId()`：获取单据内码
  - ✓ `public CloseCallBack getCloseCallBack()`：回调参数，据此了解回调源头

示例

- 案例说明

1. 单据列表上，单据编号、文本 1 两列，均显示为超链接
2. 点击文本 1，打开物料新增界面
3. 比较 billClosedCallBack、closedCallBack 两个事件的事件参数值

- 实现方案

1. 捕获 billListHyperLinkClick 事件，显示物料新增界面，并指定回调参数
2. 捕获 billClosedCallBack，接收事件参数 pkid，提示出来
3. 捕获 closedCallBack，接收事件参数 returndata，提示出来

- 实例代码

Java

```
package kd.bos.plugin.sample.bill.list.bizcase;

import kd.bos.bill.BillShowParameter;
import kd.bos.bill.OperationStatus;
import kd.bos.dataentity.utils.StringUtils;
import kd.bos.form.CloseCallBack;
import kd.bos.form.ShowType;
import kd.bos.form.events.ClosedCallBackEvent;
import kd.bos.form.events.HyperLinkClickArgs;
import kd.bos.list.events.BillClosedCallBackEvent;
import kd.bos.list.plugin.AbstractListPlugin;

public class BillClosedCallBackSample extends AbstractListPlugin {

    private final static String KEY_TEXTFIELD1 = "textfield1";

    /**
     * 用户点击超链接单元格时，触发此事件
     */
    @Override
    public void billListHyperLinkClick(HyperLinkClickArgs args) {
        if (StringUtils.equals(KEY_TEXTFIELD1,
args.getHyperLinkClickEvent().getFieldName())){
            // 当前点击的是文本1

            // 取消系统自动打开本单的处理
            args.setCancel(true);

            // 打开物料新增界面
            BillShowParameter showParameter = new BillShowParameter();
            showParameter.setFormId("bd_material");
            showParameter.getOpenStyle().setShowType(ShowType.Modal);
            showParameter.setStatus(OperationStatus.ADDNEW);

            CloseCallBack closeCallBack = new CloseCallBack(this, KEY_TEXTFIELD1);
            showParameter.setCloseCallBack(closeCallBack);

            this.getView().showForm(showParameter);
        }
    }

    /**
     * 单据界面关闭时，触发本事件，传入单据内码
     */
```

```
    @Override
    public void billClosedCallBack(BillClosedCallBackEvent e) {

        if (StringUtils.equals(KEY_TEXTFIELD1, e.getCloseCallBack().getActionId())){
            // 自定义的物料新增界面返回
            long materialId = (long)e.getPkId();
            this.getView().showMessage(String.format("事件 billClosedCallBack，可以收到系统自动打包的子界面内码%d", materialId));
        }
    }

    /**
     * 单据界面关闭时，也会触发本事件；但是默认不带数据返回
     */
    @Override
    public void closedCallBack(ClosedCallBackEvent closedCallBackEvent) {
        if (StringUtils.equals(KEY_TEXTFIELD1, closedCallBackEvent.getActionId())){
            // 自定义的物料新增界面返回
            Object returnData = closedCallBackEvent.getReturnData();
            //this.getView().showMessage(String.format("事件 closedCallBack，只能收到子界面主动返回给列表界面的数据%s", returnData));
        }
    }

}
```

7.4.11. listRowClick 事件

事件触发时机

用户点击单据列表行时，触发此事件。

在移动端的单据列表上，用户点击行时，系统会自动打开单据的详情界面。插件可以在此事件，取消上述系统内置的逻辑；

在 PC 端的单据列表上，这个事件也会被触发，但捕捉此事件的意义不大。

代码模板

```Java
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.list.events.ListRowClickEvent;
import kd.bos.list.plugin.AbstractListPlugin;

public class ListRowClick extends AbstractListPlugin {

    @Override
    public void listRowClick(ListRowClickEvent evt) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class ListRowClickEvent extends RowClickEvent`
- `public class RowClickEvent extends EventObject`
  - ✓ `public Object getSource()`：事件源，单据列表控件BillList
  - ✓ `public void setCancel(boolean cancel)`：取消后续逻辑
  - ✓ `public int getPageIndex()`：获取当前行所在的页码
  - ✓ `public int getRow()`：获取当前行号
  - ✓ `public ListSelectedRow getCurrentListSelectedRow()`：获取当前行信息对象
  - ✓ `public ListSelectedRowCollection getListSelectedRowCollection()`：获取当前列表，勾选的全部行信息

示例

参阅 listRowDoubleClick 事件示例。

7.4.12. listRowDoubleClick 事件

事件触发时机

用户双击单据列表行时，触发此事件。

普通单据列表双击行，会自动打开单据修改界面；而单据 F7 列表双击行，会把选中的行，返回给调用页面；

插件可以在此事件中，取消上述系统内置的逻辑；

代码模板

Java

```
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.list.events.ListRowClickEvent;
import kd.bos.list.plugin.AbstractListPlugin;

public class ListRowDoubleClick extends AbstractListPlugin {

    @Override
    public void listRowDoubleClick(ListRowClickEvent evt) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

参阅 listRowClick 事件参数。

示例

➢ 案例说明

1. 用户单击列表行时，不做任何处理；
2. 用户双击列表行时，不弹出默认的单据界面，而是打开当前行使用的物料界面

➢ 实现方案

1. 捕获 listRowClick 事件，取消后续操作
2. 捕获 listRowDoubleClick 事件
   a. 取消系统预置的后续操作
   b. 获取当前行上的物料内码，自行打开物料查看界面

➢ 实例代码

Java

```
package kd.bos.plugin.sample.bill.list.bizcase;

import kd.bos.bill.BillShowParameter;
import kd.bos.bill.OperationStatus;
import kd.bos.form.ShowType;
import kd.bos.list.ListShowParameter;
import kd.bos.list.events.ListRowClickEvent;
import kd.bos.list.plugin.AbstractListPlugin;

public class ListRowDoubleClickSample extends AbstractListPlugin {

    private final static String ENTITYID_MATERIAL = "bd_material";
    private final static String KEY_MATERIALID = "material";

    /**
     * 用户单击行时触发此事件
     * @remark
     * 在移动端单据列表上，用户单击行打开单据界面；
     * 可以在此事件，取消上述逻辑
     */
    @Override
    public void listRowClick(ListRowClickEvent evt) {
        // 取消后续处理
        evt.setCancel(true);
    }

    /**
     * 用户双击行时触发此事件
     * @remark
     * 普通单据列表，双击行，会自动打开单据修改界面；
     * 而单据F7列表双击行行，会把选中的行，返回给调用页面；
     * 可以在插件中，取消上述逻辑
     */
    @Override
    public void listRowDoubleClick(ListRowClickEvent evt) {
```

```
        if (!isLookup()){
            // 取消系统内置的逻辑处理
            evt.setCancel(true);

            // 自行打开物料查看界面
            if (!evt.getCurrentListSelectedRow().getDataMap().containsKey(KEY_MATERIALID)){
                return;
            }
            long materialId =
(long)evt.getCurrentListSelectedRow().getDataMap().get(KEY_MATERIALID);
            if (materialId == 0){
                return;
            }

            BillShowParameter showParameter = new BillShowParameter();
            showParameter.setFormId(ENTITYID_MATERIAL);
            showParameter.setPkId(materialId);

            showParameter.getOpenStyle().setShowType(ShowType.Modal);
            showParameter.setStatus(OperationStatus.VIEW);

            this.getView().showForm(showParameter);
        }
    }

    /**
     * 是否F7列表
     *
     * @return true是，false不是
     */
    private boolean isLookup() {
        boolean isLookup = false;
        if (this.getView().getFormShowParameter() instanceof ListShowParameter) {
            ListShowParameter listShowParameter = (ListShowParameter)
this.getView().getFormShowParameter();
            isLookup = listShowParameter.isLookUp();
        }
        return isLookup;
    }
}
```

7.4.13. filterContainerBeforeF7Select 事件

过滤容器内 F7 弹出前的处理方法

代码模板

Java

```
package kd.billlist.demo.plugin;

import kd.bos.form.field.events.BeforeFilterF7SelectEvent;
import kd.bos.list.plugin.AbstractListPlugin;

public class BillListDemoPlugin extends AbstractListPlugin{

    @Override
    public void filterContainerBeforeF7Select(BeforeFilterF7SelectEvent args) {
        // TODO Auto-generated method stub
        super.filterContainerBeforeF7Select(args);
    }
}
```

示例

➢ 案例说明

-- 需要在常用过滤基础资料字段点击“更多”或者方案过滤基础资料字段输入框的弹窗增加过滤条件

➢ 实现方案

--

➢ 实例代码

Java

```
package kd.billlist.demo.plugin;

import kd.bos.form.field.events.BeforeFilterF7SelectEvent;
import kd.bos.list.plugin.AbstractListPlugin;
import kd.bos.orm.query.QFilter;

public class BillListDemoPlugin extends AbstractListPlugin{

    @Override
    public void filterContainerBeforeF7Select(BeforeFilterF7SelectEvent args) {
        // TODO Auto-generated method stub
        if ("basedatafield2.name".equals(args.getFieldName()))
```

```
            args.addCustomQFilter(new QFilter("name", QFilter.equals, "test123"));
        super.filterContainerBeforeF7Select(args);
    }
}
```

7.4.14. filterColumnSetFilter 事件

过滤字段上的基础资料字段过滤条件调整事件

代码模板

Java

```
package kd.billlist.demo.plugin;

import kd.bos.form.events.SetFilterEvent;
import kd.bos.list.plugin.AbstractListPlugin;

public class BillListDemoPlugin extends AbstractListPlugin{

    @Override
    public void filterColumnSetFilter(SetFilterEvent args) {
        // TODO Auto-generated method stub
        super.filterColumnSetFilter(args);
    }
}
```

示例

- 案例说明
  -- 需要在常用过滤基础资料字段初始化时增加过滤条件

- 实现方案
  --

- 实例代码

Java

```
package kd.billlist.demo.plugin;

import kd.bos.form.events.FilterColumnSetFilterEvent;
```

```
import kd.bos.form.events.SetFilterEvent;
import kd.bos.list.plugin.AbstractListPlugin;
import kd.bos.orm.query.QFilter;

public class BillListDemoPlugin extends AbstractListPlugin{

    @Override
    public void filterColumnSetFilter(SetFilterEvent args) {
        // TODO Auto-generated method stub
        if ("basedatafield2.name".equals(args.getFieldName()))
            args.addCustomQFilter(new QFilter("name", QFilter.equals, "test123"));
        // 基础资料依赖于上面的选择字段orgfield1,将orgfield1的已选中值添加到basedatafield2
        if ("basedatafield2.name".equals(args.getFieldName())) {
            FilterColumnSetFilterEvent args2 = (FilterColumnSetFilterEvent) args;
            args.addCustomQFilter(new QFilter("name", QFilter.equals,
args2.getCommonFilterValue("orgfield1.number")));
        }
        super.filterColumnSetFilter(args);
    }
}
```

7.4.15. filterContainerAfterSearchClick 事件

过滤容器搜索点击后的处理方法,此事件发生在过滤条件解析后，主要用于点击过滤条件时联动修改其他过滤字段控件。

代码模板

Java

```
package kd.billlist.demo.plugin;

import kd.bos.form.events.FilterContainerSearchClickArgs;
import kd.bos.list.plugin.AbstractListPlugin;

public class BillListDemoPlugin extends AbstractListPlugin{

    @Override
    public void filterContainerAfterSearchClick(FilterContainerSearchClickArgs args) {
        // TODO Auto-generated method stub
        super.filterContainerAfterSearchClick(args);
    }
}
```

示例

- 案例说明
  -- 需要在快速过滤中，点击搜索，清空搜索框

- 实现方案
  -- 在 filterContainerAfterSearchClick 事件中，对搜索框的条件清除。

- 实例代码

Java

```
package kd.billlist.demo.plugin;

import java.util.List;

import kd.bos.form.events.FilterContainerSearchClickArgs;
import kd.bos.list.plugin.AbstractListPlugin;
import kd.bos.orm.query.QFilter;

public class BillListDemoPlugin extends AbstractListPlugin{

    @Override
    public void filterContainerAfterSearchClick(FilterContainerSearchClickArgs args) {
        // TODO Auto-generated method stub
        super.filterContainerAfterSearchClick(args);
        List<QFilter> qfilters = args.getFastQFilters();
        if (qfilters.size() != 0) {
            qfilters.remove(0);
            this.getView().updateView("filtercontainerap");
        }
    }
}
```

7.4.16. baseDataColumnDependFieldSet 事件

设置常用过滤的基础资料依赖字段

代码模板

Java

```
package kd.billlist.demo.plugin;
```

```
import kd.bos.form.control.events.BaseDataColumnDependFieldSetEvent;
import kd.bos.list.plugin.AbstractListPlugin;

public class BillListDemoPlugin extends AbstractListPlugin{

    @Override
    public void baseDataColumnDependFieldSet(BaseDataColumnDependFieldSetEvent args) {
        // TODO Auto-generated method stub
        super.baseDataColumnDependFieldSet(args);
    }
}
```

示例

待更新

7.4.17. setCellFieldValue 事件

设置单元格指令

代码模板

Java

```
package kd.billlist.demo.plugin;

import kd.bos.list.events.SetCellFieldValueArgs;
import kd.bos.list.plugin.AbstractListPlugin;

public class BillListDemoPlugin extends AbstractListPlugin{

    @Override
    public void setCellFieldValue(SetCellFieldValueArgs args) {
        // TODO Auto-generated method stub
        super.setCellFieldValue(args);
    }
}
```

示例

待更新

7.5. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=07 进行反馈。

8. 左树右表单据列表插件

左树右表单据列表，是高级的单据列表界面，在左边显示分组树、右边显示数据列表：
1. 默认会自动加载分组基础资料数据，填充在分组树；
2. 用户点击分组树节点，自动对数据列表进行过滤；
3. 选择数据比较方便，通常用于显示基础资料。

左树右表继承了标准列表的功能，增加了对分组树的管理，插件事件也做了相应的补充。

8.1. 插件基类

8.1.1. 插件接口及基类

左树右表单据列表界面插件基类为 AbstractTreeListPlugin，派生自标准单据列表界面插件基类，支持表单、标准单据列表界面的各种插件事件，另外还新新增了 ITreeListPlugin, SearchEnterListener 事件接口。

左树右表单据列表界面插件基类定义：

Java

```
package kd.bos.list.plugin;
public class AbstractTreeListPlugin extends AbstractListPlugin implements ITreeListPlugin,
SearchEnterListener {
```

8.1.2. 创建并注册插件

如下示例代码，自定义了一个树形单据列表界面业务插件：

Java

```
package kd.bos.plugin.sample.bill.list.bizcase;

import kd.bos.list.plugin.AbstractTreeListPlugin;

public class TreeListEventSample extends AbstractTreeListPlugin {

}
```

8.2. 视图模型

8.2.1. 接口与实现类

左树右表单据列表界面，提供了三个层次的视图模型：

- 表单视图模型 IFormView：据此访问表单的各种信息；
- 列表视图模型 IListView：据此访问单据列表的各种信息；
- 树列表视图模型 ITreeListView：据此访问树列表的各种信息；

表单视图模型 IFormView、列表视图模型 IListView，使用一个统一的实现类和实例（ListView），可以互相转换；

但是树列表视图模型 ITreeListView，采用了一个独立的实现类 TreeListView，与 ListView 没有派生关系，不能互相转换。

左树右表列表插件(AbstractTreeListPlugin)，可以使用如下代码，分别获取 IFormView、IListView、ITreeListView 接口实例：

Java

```
IFormView formView = this.getView();
IListView listView = (IListView)this.getView();
ITreeListView treeListView = listView.getTreeListView();
```

ITreeListView 接口定义如下：

Java

```
package kd.bos.list;

import kd.bos.entity.datamodel.ITreeModel;
import kd.bos.form.control.TreeView;

public interface ITreeListView {
```

```
    ITreeModel getTreeModel();
    void initialize(IListView view, TreeView tv);
    void refresh();
    void refreshTreeView();
    void refreshTreeNode(String nodeId);
    void focusRootNode();
    TreeView getTreeView();
}
```

8.2.2. 功能方法及使用

ITreeListView 接口提供如下方法：

方法	说明
initialize	初始化
getTreeModel	获取树形列表视图模型
refresh	刷新列表，重新取数
refreshTreeView	刷新树列表，重新取数
focusRootNode	回到树根节点
getTreeView	获取界面上的树形控件实例

8.3. 数据模型

8.3.1. 接口与实现

左树右表列表界面，包含了三个数据模型：

- 表单数据模型 IDataModel：据此访问列表控件所在的表单数据：
  - ✓ 列表界面，除了单据列表控件，还可以增加各种自定义字段，需要通过表单数据模型 IDataModel，访问这些自定义字段的值；
- 列表数据模型 IListModel：据此访问单据列表控件中的数据；
- 树数据模型 ITreeModel：据此访问分组树控件中的数据；

这三个数据模型之间，没有继承关系，需要分别获取，不能互相转换。

表单数据模型 IDataModel，在动态表单章节已有介绍，列表数据模型 IListModel 请参阅标准单据列表章节。

左树右表列表数据模型 ITreeModel，定义如下：

```Java
package kd.bos.entity.datamodel;
public interface ITreeModel {
```

左树右表单据列表插件(AbstractTreeListPlugin)，可以如下代码，分别获取表单数据模型 IDataModel、列表数据模型 IListModel、树列表数据模型 ITreeModel 的实例：

```Java
IDataModel formModel = this.getView().getModel();
IListModel listModel = ((IListView)this.getView()).getListModel();
//ITreeModel treeModel = ((IListView)this.getView()).getTreeListView().getTreeModel();
ITreeModel treeModel = this.getTreeModel();
```

8.3.2. 功能方法及使用

ITreeModel 接口提供如下方法：

方法	说明
createRootNode	创建一个根节点对象实例返回
addNode	给指定节点下，添加子节点，放在缓存中； 本方法本身不会同步给前端树控件添加子节点； 因此，需要在合适的事件中调用，或者自行向树控件下达指令
deleteNode	从缓存中删除子节点 本方法本身不会同步删除前端树控件的子节点； 因此，需要在合适的事件中调用，或者自行向树控件下达指令
deleteGroup	从数据库，删除分组数据
getRoot	获取缓存的根节点及其下全部子节点；
setRoot	重置缓存中的根节点及其子节点； 本方法本身不会同步刷新前端树控件； 因此，需要在合适的事件中调用，或者自行向树控件下达指令
getCurrentNodeId	获取当前节点标识
setCurrentNodeId	设置当前节点标识； 本方法没有同步更新前端树控件的焦点； 因此，需要自行向树控件下达指令
isNodeClickExpand	?
isRootVisable	根节点是否可见
setRootVisable	设置根节点是否可见
getGroupProp	获取分组字段属性对象
getTreeFilter	树节点取数条件 系统自动取分组数据时，会用此条件对分组数据进行过滤
getDefaultQueryLevel	分组节点搜索最深级次；

	暂未发现在何处使用到这个属性
setDefaultQueryLevel	设置分组节点搜索最深级次
setTextFormat	设置树节点显示格式，{name}表示名称，{code}表示编码： 格式样式如："{name} - {code}" 或者 "名称：{name} - 编码：{code}" 默认为"{name}"
getListFilter	获取根据节点生成的列表过滤条件 内部方法，插件勿调用
refreshNode	前端界面通过此方法，通知插件刷新节点； 内部方法，插件勿调用

8.4. 插件事件

左树右表单据列表界面插件基类为 AbstractTreeListPlugin，派生自单据列表界面插件基类，能够支持表单、单据列表界面的各种插件事件，另外还新新增了 ITreeListPlugin, SearchEnterListener 事件接口。

新增加的插件事件，触发时机及顺序说明如下：

分类	事件	触发时机
界面初始化	setView	系统构建出插件实例时，传入当前表单视图实例
	createTreeListView	在系统准备创建 ITreeListView 接口实例时，触发此事件；
	setTreeListView	系统在触发树形列表插件事件之前，都会调用此方法，传入当前树形列表视图模型
	initializeTree	树形视图模型 ITreeListView 实例初始化时，触发此事件；
树分组	initTreeToolbar	分组树工具栏初始化后，触发此事件
	treeToolbarClick	用户点击了分组树上的工具栏按钮时，触发此事件；
	beforeBuildTreeNode	未发现触发时机
	refreshNode	重新刷新树形控件节点时触发；
	expendTreeNode	用户点击树节点前的"+"标识时，需要懒加载子节点时，触发此事件
	beforeTreeNodeClick	未发现触发时机
	treeNodeClick	用户点击分组树上的节点时，触发此事件；
	buildTreeListFilter	基于当前所选分组节点，生成列表过滤条件时触发
树节点搜索	search	用户在树形控件上的搜索框，输入文本后回车，即触发此事件；

nodeClickFilter	设置点击节点时 list 的过滤条件

特别说明：

左树右表单据列表，使用了树形控件来展示分组数据，树形控件本身的功能特点及插件事件，也可以应用在树形单据列表界面插件中。因此，可以结合树形控件的插件事件，实现更加灵活的分组树功能，如懒加载树节点；

树形控件的使用，请阅读树形控件章节。

8.4.1. setView 事件

事件触发时机

系统初始化表单视图模型时，会同步创建表单插件，并调用表单插件的 setView 方法，传入当前表单视图实例；

插件可以在此获取表单视图模型 IFormView 实例，把 IFormView 实例放在本地，以备后续事件中使用。

表单插件基类 AbstractFormPlugin 已经实现了本方法，把收到的 IFormView 实例，放在了本地变量，其他插件事件中，可以通过 this.getView()获取到本事件传入的 IFormView 实例。

本事件在表单事件章节已有介绍，此处不重复。

8.4.2. createTreeListView 事件

事件触发时机

当系统或者插件，尝试获取树列表视图模型实例时，即触发此事件。

插件可以在此事件中，构建自定义的树列表视图模型实例，代替系统预置的树列表视图模型实例 TreeListView。

此事件属于高级编程，业务插件自行实现列表视图模型 ITreeListView 接口，完成更加灵活的业务需求。

代码模板

```Java
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.list.events.CreateTreeListViewEvent;
import kd.bos.list.plugin.AbstractTreeListPlugin;

public class CreateTreeListView extends AbstractTreeListPlugin {

    @Override
    public void createTreeListView(CreateTreeListViewEvent e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class CreateTreeListViewEvent extends EventObject`
  - ✓ `public Object getSource()`：事件源，列表视图模型 IListView 接口实例
  - ✓ `public void setView(AbstractTreeListView view)`：设置自定义的树形列表视图模型实例，扩展实现了 ITreeListView 接口的抽象基类 AbstractTreeListView

示例

通常情况下，业务插件不需要关注此事件，示例略。

8.4.3. setTreeListView 事件

事件触发时机

系统在触发树列表插件事件之前，都会调用此方法，传入当前树列表视图模型

左树右表列表界面插件基类 AbstractTreeListPlugin 已经实现了本方法，把传入的树列表视图模型实例，放在本地，后续使用 this.getTreeListView()即可获得实例。

业务插件无需重写此方法。

代码模板

Java

```
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.list.ITreeListView;
import kd.bos.list.plugin.AbstractTreeListPlugin;

public class SetTreeListView extends AbstractTreeListPlugin {

    @Override
    public void setTreeListView(ITreeListView treeListView) {
        super.setTreeListView(treeListView);
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `ITreeListView treeListView`：树形列表视图模型 ITreeListView 接口实例

示例

通常情况下，业务插件不需要关注此事件，示例略。

8.4.4. initializeTree

事件触发时机

系统在构建树列表视图模型 ITreeListView 实例，以及树列表数据模型 ITreeModel 实例成功后，即触发此事件；

在界面初始化时，以及树形列表前端界面每次与服务端交互时，均会创建 ITreeListView、ITreeModel 的全新实例，因此 initializeTree 事件会被频繁的触发。

插件可以在此事件，设置分组数据取数条件，或初始化一些本地变量。

特别说明：
不建议在此事件中加载分组树节点；
如果在此事件中给分组树添加节点，需要同时处理 refreshNode 事件，避免节点被覆盖，导致添加的节点没有显示出来；

initializeTree 事件与其他几个初始化事件的差异：

- setView：在此事件中，还没有构建 ITreeListView 接口实例；
- createTreeListView：在此事件中，还没有构建 ITreeListView 接口实例；
- setTreeListView：树形列表每次事件触发时，均先触发此事件，触发时机不够明确，不适合业务插件捕获；
- initialize：此事件属于表单的界面初始化事件，单据界面、列表界面、树形列表界面，都会触发 initialize 事件，比 initializeTree 更加通用，触发时机也早些；
- initializeTree：在初始化 ITreeListView 接口实例时触发，此是已经有了 ITreeListView 接口实例；

代码模板

Java

```java
package kd.bos.plugin.sample.bill.list.template;

import java.util.EventObject;

import kd.bos.list.plugin.AbstractTreeListPlugin;

public class InitializeTree extends AbstractTreeListPlugin {

    @Override
    public void initializeTree(EventObject e) {
        super.initializeTree(e);
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class EventObject implements java.io.Serializable`
  - ✓ `public Object getSource()`：事件源，ListViewPluginProxy 实例

示例

- 案例说明：

1. 强制设置分组节点过滤条件 number like '%abc%'
2. 设置分组节点内容格式化为 名称 {名称}(编码{编码})

- 实现方案：

1. 捕获 initializeTree 事件，设置 ITreeModel 属性

- 实例代码：

Java

```java
package kd.bos.plugin.sample.bill.list.bizcase;

import java.util.EventObject;

import kd.bos.list.plugin.AbstractTreeListPlugin;
import kd.bos.orm.query.QFilter;

public class InitializeTreeSample extends AbstractTreeListPlugin {

    @Override
    public void initializeTree(EventObject e) {
        super.initializeTree(e);

        // 根节点是否显示
        this.getTreeModel().setRootVisable(true);

        if (this.getTreeModel().getGroupProp() != null){

            // 分组节点取数条件：
            // 只有单据有分组字段，分组节点由系统自动读取、构建时，才会用到这个条件
            this.getTreeModel().getTreeFilter().add(new QFilter("number", "like", "%abc%"));

            // 分组节点内容格式化
            this.getTreeModel().setTextFormat("名称{name}(编码{code})");
```

```
            // 分组节点取数级次 (暂未发现在何处使用到这个属性)
            this.getTreeModel().setDefaultQueryLevel(5);
        }
    }
}
```

8.4.5. initTreeToolbar

事件触发时机

左树右表列表界面初始化，分组树控件的工具栏面板初始化结束后，触发此事件。

工具栏面板中，包含分组节点搜索框、新增、修改、删除节点按钮；
插件可以在此事件，设置工具栏面板的可见性；

代码模板

Java

```java
package kd.bos.plugin.sample.bill.list.template;

import java.util.EventObject;

import kd.bos.list.plugin.AbstractTreeListPlugin;

public class InitTreeToolbar extends AbstractTreeListPlugin {

    @Override
    public void initTreeToolbar(EventObject e) {
        super.initTreeToolbar(e);
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class EventObject implements java.io.Serializable`
  - ✓ `public Object getSource()`：事件源

示例

示例代码节选（完整实例请参阅 refreshNode 事件示例）：

Java

```
/**
 * 初始化树分组控件上的工具面板时，触发此事件
 * @remark
 * 插件在此事件，隐藏树工具面板
 */
@Override
public void initTreeToolbar(EventObject e) {
    super.initTreeToolbar(e);
    this.getView().setVisible(false, KEY_TREEBUTTONPANEL);
    // 如下代码演示单独隐藏树分组面板-新增按钮
    //this.getView().setVisible(false, "btnnew");
}
```

8.4.6. treeToolbarClick

事件触发事件

用户点击了分组树上的工具栏按钮时，触发此事件；

插件可以在此事件中，自行处理分组新增(btnnew)、修改(btnedit)、删除(btndel)功能。

特别说明：
分组基础资料列表上默认配置的通用插件 TemplateGroupBaseDataPlugin(派生自 StandardTreeListPlugin)，已经对分组树工具栏按钮的点击进行了处理，业务插件不需要重写。

如果需要重写，请停用系统默认配置的通用插件 TemplateGroupBaseDataPlugin，然后扩展该类，

重写 treeToolbarClick 事件。

代码模板

Java

```
package kd.bos.plugin.sample.bill.list.template;

import java.util.EventObject;

import kd.bos.list.plugin.AbstractTreeListPlugin;

public class TreeToolbarClick extends AbstractTreeListPlugin {

    @Override
    public void treeToolbarClick(EventObject e) {
        super.treeToolbarClick(e);
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class EventObject implements java.io.Serializable`
  - ✓ `public Object getSource()`: 事件源

示例

业务插件通常不需要捕捉此事件，示例略过。

8.4.7. beforeBuildTreeNode

暂未触发。

8.4.8. refreshNode

事件触发时机

系统开始加载、刷新分组树子节点时，触发此事件；

单据存在分组字段时，系统默认会自动加载分组基础资料的数据作为分组树节点；

如果分组树节点的数据，不是来自于分组基础资料，则需要业务插件捕获此事件，自行构建分组树上的节点；

特别注意：
在展开节点、或者列表刷新时，均会重复触发本事件，需要避免重复添加子节点；

代码模板

Java

```java
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.form.control.events.RefreshNodeEvent;
import kd.bos.list.plugin.AbstractTreeListPlugin;

public class RefreshNode extends AbstractTreeListPlugin {

    @Override
    public void refreshNode(RefreshNodeEvent e) {
        super.refreshNode(e);
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class RefreshNodeEvent extends EventObject`
  - ✓ `public Object getSource()`：事件源
  - ✓ `public Object getNodeId()`：当前节点标识，需要加载此节点的子节点
  - ✓ `public void setChildNodes(List<TreeNode> childNodes)`：设置子节点

示例

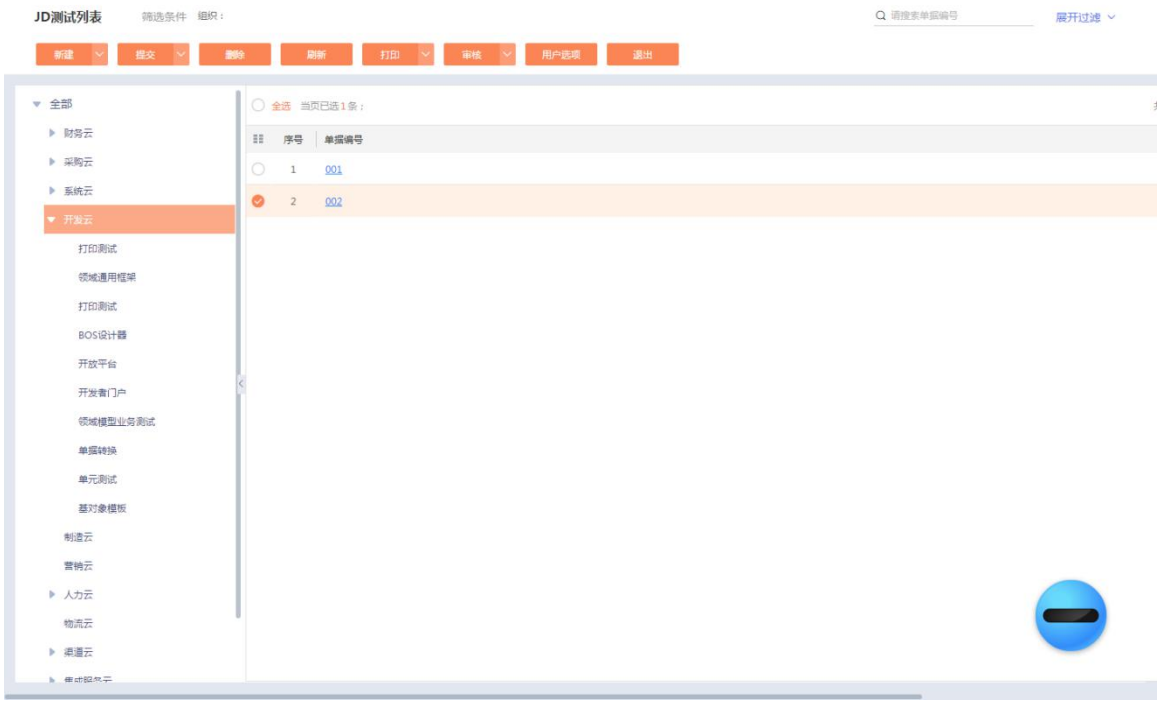
➢ 案例说明

1. 单据上有个字段"bizappid"，存储了业务应用基础资料；
2. 单据列表上，需要以业务云、业务应用作为分组节点；
3. 用户点击业务应用时，按业务应用过滤列表；点击业务云时，需按此业务云下的全部业务应用过滤列表；

➢ 实现方案

1. 捕获 refreshNode 事件，刷新加载业务云、业务应用节点；
2. 因为业务云、业务应用数量有限，不使用节点懒加载；
3. 捕获 buildTreeListFilter 事件，根据点击的业务云、业务应用，生成列表过滤条件

➢ 运行效果



➢ 实例代码

Java

```
package kd.bos.plugin.sample.bill.list.bizcase;

import java.util.ArrayList;
import java.util.EventObject;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
```

```
import kd.bos.algo.DataSet;
import kd.bos.algo.Row;
import kd.bos.dataentity.utils.StringUtils;
import kd.bos.entity.tree.TreeNode;
import kd.bos.form.control.events.RefreshNodeEvent;
import kd.bos.list.events.BuildTreeListFilterEvent;
import kd.bos.list.plugin.AbstractTreeListPlugin;
import kd.bos.orm.ORM;
import kd.bos.orm.query.QCP;
import kd.bos.orm.query.QFilter;

public class RefreshNodeSample extends AbstractTreeListPlugin {

    /** 树形控件上方的按钮的主面板 */
    private static final String KEY_TREEBUTTONPANEL  = "flexpanel_treebtn";
    /** 业务应用字段标识 */
    private static final String KEY_BIZAPPID  = "bizappid";

    /**
     * 初始化树分组控件上的工具面板时，触发此事件
     * @remark
     * 插件在此事件，隐藏树工具面板
     */
    @Override
    public void initTreeToolbar(EventObject e) {
        super.initTreeToolbar(e);
        this.getView().setVisible(false, KEY_TREEBUTTONPANEL);
    }

    /**
     * 加载、刷新子节点事件
     * @remark
     * 默认情况下，系统会自动加载基础资料分组表格中的数据作为分组树节点；
     * 插件可以在此事件中，自行构建分组树上的节点；
     * 另外，在展开任意节点、或者是列表刷新时，均会触发本事件，需要避免重复添加子节点
     */
    @Override
    public void refreshNode(RefreshNodeEvent e) {
        super.refreshNode(e);

        // 取缓存的根节点
        TreeNode root = this.getTreeModel().getRoot();
```

```
        // 当前节点
        String currNodeId = (String)e.getNodeId();
        TreeNode currNode = root.getTreeNode(currNodeId, 10);

        // 判断当前节点下，是否已经缓存了子节点：如果有，无需重复加载
        if (currNode.getChildren() != null && !currNode.getChildren().isEmpty()){
            // 直接返回缓存的子节点
            e.setChildNodes(currNode.getChildren());
        }
        else {
            // 没有缓存子节点
            if (StringUtils.equals(currNodeId, root.getId())){
                // 当前要刷新的是根节点：读取全部业务云、业务应用，生成节点返回
                e.setChildNodes(this.loadAppNodes());
            }
            else {
                // 当前要刷新非根节点
                // 业务应用节点是一次性加载完毕的，没有采用懒加载；如果在缓存中没有找到其子节点，则说明其没有子节点
                // TODO 如果采用了懒加载子节点方案，则需要在这里尝试加载子节点
                e.setChildNodes(null);
            }
        }
    }

    /**
     * 基于当前选择的分组节点，生成单据列表过滤条件时触发
     * @remark
     * 插件可以在此插件中，略过系统内置的分组条件，自行生成分组过滤条件
     */
    @Override
    public void buildTreeListFilter(BuildTreeListFilterEvent e) {
        super.buildTreeListFilter(e);

        // 生成过滤条件
        QFilter filter = this.buildAppQFilter(KEY_BIZAPPID, (String)e.getNodeId());
        e.addQFilter(filter);

        e.setCancel(true);    // 略过系统内置的分组过滤条件
    }

    /************** 如下代码，用来加载、搜索业务应用清单  *****************/

    /**
```

```
     * 构造业务应用树节点返回
     */
    private List<TreeNode> loadAppNodes(){

        // 读取全部业务云
        Map<String, TreeNode> allNodes = new HashMap<>();
        List<TreeNode> cloudNode4 = getCloudData();
        for(TreeNode node: cloudNode4){
            allNodes.put(node.getId(), node);
        }

        // 读取全部应用
        List<TreeNode> appNode4 = getAppData();
        for(TreeNode node: appNode4){
            allNodes.put(node.getId(), node);
        }

        // 构建节点的父子关系
        for(TreeNode node : allNodes.values()){
            TreeNode parentNode = allNodes.get(node.getParentid());
            if (parentNode != null){
                parentNode.addChild(node);
            }
        }

        return cloudNode4;
    }

    /**
     * 查询数据库，获取业务云数据，构造为树节点集合返回
     * @return
     */
    private List<TreeNode> getCloudData() {
        List<TreeNode> cloudNode = new ArrayList<TreeNode>();

        ORM orm = ORM.create();
        String fields = "id, number, name";
        String orderby = "seq asc";
        QFilter[] filters = new QFilter[] {};

        try(DataSet ds = orm.queryDataSet("bos_devportal_bizcloud", "bos_devportal_bizcloud",
fields, filters, orderby)) {
            Iterator<Row> rows = ds.iterator();
            while (rows.hasNext()) {
```

```
                Row row = rows.next();
                TreeNode node = new TreeNode();
                String nodeID = "cloud/" + row.getString("number");
                node.setText(row.getString("name"));
                node.setParentid("0");
                node.setId(nodeID);
                cloudNode.add(node);
            }
        }
        return cloudNode;
    }

    /**
     * 查询数据库，获取业务应用数据，构造为业务应用树节点集合返回
     * @return
     */
    private static List<TreeNode> getAppData() {
        // 可见性与发布状态
        List<TreeNode> appNode = new ArrayList<TreeNode>();

        ORM orm = ORM.create();
        String fields = "id,number, name, bizcloud.id, bizcloud.number";
        String orderby = "seq asc";

        QFilter[] filters = new QFilter[] {};

        try (DataSet ds = orm.queryDataSet("bos_devportal_bizapp", "bos_devportal_bizapp",
fields, filters, orderby)){
            Iterator<Row> rows = ds.iterator();
            while (rows.hasNext()) {
                Row row = rows.next();
                TreeNode node = new TreeNode();
                String parentId = "cloud/" + row.getString("bizcloud.number");
                String nodeID = row.getString("id");
                node.setId(nodeID);
                node.setParentid(parentId);
                node.setText(row.getString("name"));
                appNode.add(node);
            }
        }
        return appNode;
    }

    /**
```

```
     * 根据当前选择节点，生成过滤业务应用范围的条件：
     *
     * 如果当前点击的节点是业务应用，则直接按此业务应用过滤；
     * 如果点钱点击的节点是业务云，则取此业务云下全部业务应用进行过滤；
     *
     * @param appIdFldKey 单据上，业务应用字段名
     * @param nodeId 当前点击的节点
     * @return
     */
    private QFilter buildAppQFilter(String appIdFldKey, String nodeId){

        List<String> appIds = new ArrayList<>();
        TreeNode root = this.getTreeModel().getRoot();
        List<TreeNode> bizCloudNodes = root.getChildren();

        if (bizCloudNodes != null){
            for(TreeNode bizCloudNode : bizCloudNodes){
                if (matchCloudNode(nodeId, bizCloudNode, appIds)){
                    break;
                }
            }
        }

        if (appIds.isEmpty()){
            // 所选节点，未命中任何业务应用：应该选择的是根节点，不限条件
            return null;
        }
        else if (appIds.size() == 0) {
            return new QFilter(appIdFldKey, QCP.equals, appIds.get(0));
        }
        else {
            return new QFilter(appIdFldKey, QCP.in, appIds.toArray(new String[appIds.size()]));
        }
    }

    /**
     * 判断当前节点，是否与指定的业务云匹配；如果是，返回true，并把本业务云下的全部应用，输出到outAppIds参数中
     *
     * @param nodeId 当前节点
     * @param bizCloudNode 待匹配的业务云节点
     * @param outAppIds 输出
     * @return
     */
```

```
    private boolean matchCloudNode(String nodeId, TreeNode bizCloudNode, List<String>
outAppIds){

        if (StringUtils.equals(nodeId, bizCloudNode.getId())){
            // 当前节点，就是本次比较的业务云节点：则输出全部业务应用进行过滤

            if (bizCloudNode.getChildren() != null){
                for(TreeNode appNode : bizCloudNode.getChildren()){
                    outAppIds.add(appNode.getId());
                }
            }
            if (outAppIds.isEmpty()){
                // 如果本业务云下没有任何业务应用：
                // 输出一个不存在的业务应用标识，生成一个永不成立的条件，使单据列表不显示任何数据
                outAppIds.add(nodeId);
            }
            return true;
        }
        else {
            // 当前节点，不是本次比较的业务云节点：则往下匹配其包含的业务应用节点

            if (matchAppNode(nodeId, bizCloudNode)){
                outAppIds.add(nodeId);
                return true;
            }
        }

        return false;
    }

    /**
     * 判断当前节点，是否属于所传业务云下的业务应用节点，如果是，返回true
     *
     * @param nodeId 当前节点
     * @param bizCloudNode 待匹配的业务云节点
     * @return
     */
    private boolean matchAppNode(String nodeId, TreeNode bizCloudNode){
        if (bizCloudNode.getChildren() != null){
            for(TreeNode appNode : bizCloudNode.getChildren()){
                if (StringUtils.equals(appNode.getId(), nodeId)){
                    return true;
                }
            }
```

```
        }
        return false;
    }
}
```

8.4.9. expendTreeNode

事件触发时机

用户点击树节点前的"+"标识时，触发此事件，向系统请求懒加载子节点。

插件可以在此方法，构建子节点返回；

特别说明：
1. 节点需要 setChildren(new ArrayList<TreeNode>())，相当于打上了懒加载标志，才会在用户点击+时，触发此事件；
2. 树形列表界面插件基类 AbstractTreeListPlugin 已经捕获此事件，并即时触发 refreshNode 事件。因此，自定义插件不需要捕获 expendTreeNode 事件，统一在 refreshNode 事件加载分组树节点。

代码模板

Java

```java
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.form.control.events.TreeNodeEvent;
import kd.bos.list.plugin.AbstractTreeListPlugin;

public class ExpendTreeNode extends AbstractTreeListPlugin {

    @Override
    public void expendTreeNode(TreeNodeEvent e) {
        super.expendTreeNode(e);
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class TreeNodeEvent extends EventObject`
  - ✓ `public Object getSource()`：事件源
  - ✓ `public Object getNodeId()`：当前待展开的节点标识，需要懒加载此节点的子节点
  - ✓ `public Object getParentNodeId()`：上一级节点标识
  - ✓ `public boolean isPropagation()`：不适用
  - ✓ `public void setExpandedNode(TreeNode expandedNode)`：不适用

示例

通常不需要捕获本事件，示例略过。

8.4.10. beforeTreeNodeClick

暂未触发。

8.4.11. treeNodeClick

事件触发时机

用户点击分组树上的节点时，触发此事件；

插件可以重写此方法，自行构建过滤条件，调用单据列表控件的方法，刷新数据；

但是，这种方式，需要自行刷新单据列表，调用的代码行比较多，过程比较复杂，不建议采用；建议业务插件，捕捉 buildTreeListFilter 事件，构建好分组过滤条件返回，然后由系统去刷新单据列表；

代码模板

Java

```
package kd.bos.plugin.sample.bill.list.template;
```

```
import kd.bos.form.control.events.TreeNodeEvent;
import kd.bos.list.plugin.AbstractTreeListPlugin;

public class TreeNodeClick extends AbstractTreeListPlugin {

    @Override
    public void treeNodeClick(TreeNodeEvent e) {
        super.treeNodeClick(e);
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- public class TreeNodeEvent extends EventObject
  - ✓ public Object getSource(): 事件源
  - ✓ public Object getNodeId(): 当前点击的节点标识
  - ✓ public Object getParentNodeId(): 上一级节点标识
  - ✓ public boolean isPropagation(): 不适用
  - ✓ public void setExpandedNode(TreeNode expandedNode): 不适用

示例

不建议捕获此事件，示例略过；

8.4.12. buildTreeListFilter

事件触发时机

基于当前选择的分组节点，生成单据列表过滤条件时触发此事件。

插件可以在此插件中，自行生成分组过滤条件，替换掉系统内部生成的分组条件。

代码模板

Java

```java
package kd.bos.plugin.sample.bill.list.template;

import kd.bos.list.events.BuildTreeListFilterEvent;
import kd.bos.list.plugin.AbstractTreeListPlugin;

public class BuildTreeListFilter extends AbstractTreeListPlugin {

    @Override
    public void buildTreeListFilter(BuildTreeListFilterEvent e) {
        super.buildTreeListFilter(e);
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class BuildTreeListFilterEvent extends EventObject`
  - ✓ `public Object getSource()`：事件源
  - ✓ `public Object getNodeId()`：当前点击的节点标识，需据此过滤列表数据
  - ✓ `public void setCancel(boolean cancel)`：取消系统内置的列表分组过滤条件
  - ✓ `public void addQFilter(QFilter filter)`：添加自定义的列表过滤条件

示例

参见 refreshNode 事件示例：
插件自行构建分组树节点，并根据当前点击的节点，生成列表过滤条件；

8.4.13. search

事件触发时机

用户在树形控件上的搜索框，输入文本后回车，即触发此事件；

插件可以重写此方法，在树形控件的子节点中进行搜索，根据搜索结果设置焦点节点。

插件自行加载分组数据时，需要同步实现搜索事件；
另外，懒加载子分组时，如果搜索到的子分组还没有显示在分组树上，需要同步向分组树，添加子分组；

这个事件与搜索控件的 search 事件用法相同，详情请参阅 search 控件。

8.4.14. nodeClickFilter 事件

设置点击节点时 list 的过滤条件

代码模板

Java

```java
package kd.treelist.demo.plugin;

import kd.bos.list.plugin.AbstractTreeListPlugin;
import kd.bos.orm.query.QFilter;

public class TreeListDemoPlugin extends AbstractTreeListPlugin{

    @Override
    protected QFilter nodeClickFilter() {
        // TODO Auto-generated method stub
        return super.nodeClickFilter();
    }
}
```

示例

待更新

8.5. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=08 进行反馈。

9. 树形基础资料列表插件

系统预置了一种特殊的基础资料模板：树形基础资料模板；

派生自这种模板的基础资料，称为树形基础资料，存储的资料数据之间，具有从属关系。

典型的树形基础资料，如部门、科目、资料类别：
一级部门下有二级部门，再下有三级部门、四级部门等；
科目下面还有分科目，如资产科目下有固定资产、流动资产等；
资产类别也是需要逐级细分，电子资产下分服务器、PC 机、平板电脑等等，还可以继续下分；

这种基础资料，默认采用左树右表列表界面，并由系统预置了一个标准树基础资料业务插件 StandardTreeListPlugin，派生自单据列表界面插件基类 AbstractTreeListPlugin，在这个插件中实现了构建分组树节点、筛选列表数据的功能。

继承树形基础资料模板建立的基础资料，已经把插件 StandardTreeListPlugin 绑定到了列表界面。

可以扩展 StandardTreeListPlugin 插件，改写系统内置的业务逻辑。

StandardTreeListPlugin 插件可重写的方法如下：

方法	说明
treeToolbarClick	树形控件上的按钮点击处理； 系统已经内置了新增、删除分组的处理，插件可以重写
search	用户在树形控件上的搜索框输入内容，搜索树节点； 如果树节点，是业务插件自行构建的，可能需要自行重写这个方法
beforeItemClick	列表主菜单点击处理； 系统预置了对删除、禁用操作的检查，有下级分组时，不允许删除，插件可以重写；

9.1. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=09 进行反馈。

10. 移动端单据插件

10.1. 插件基类

10.1.1. 插件接口及基类

移动端单据界面插件基类为 AbstractMobBillPlugIn，派生自单据插件基类 AbstractBillPlugIn，并实现了移动端界面插件接口 IMobFormPlugin：

Java

```
package kd.bos.bill;
public class AbstractMobBillPlugIn extends AbstractBillPlugIn implements IMobFormPlugin {
```

10.1.2. 创建并注册插件

请参阅单据创建与注册插件的介绍。

10.2. 视图模型

10.2.1. 接口与实现类

移动端单据的视图模型接口为 IMobileBillView，派生自 IBillView，IMobileView：

Java

```
package kd.bos.bill;
public interface IMobileBillView extends IBillView, IMobileView {
```

移动端单据的视图模型实现类为 MobileBillView，派生自单据视图模型实现类 BillView，并实现了接口 IMobileView：

Java

```
package kd.bos.mvc.bill;
public class MobileBillView extends BillView implements IMobileView {
```

10.2.2. 功能方法及使用

IMobileBillView 继承 IBillView，IMobileView。没有新增方法。

10.3. 数据模型

移动端单据的数据模型，与 PC 端单据数据模型完全一致，不重复描述。

10.4. 插件事件

移动端单据，除了 PC 端单据界面插件事件，还支持如下插件事件：

方法	说明
uploadFile	暂未发现触发时机
locate	接收当前位置信息

10.4.1. uploadFile 事件

暂未发现触发时机。

10.4.2. locate 事件

这个获取定位功能需要苍穹与承载应用的客户端有协议获取定位功能，现在只与云之家打通，暂时只在云之家可以触发

代码模板

Java

```
@Override
public void locate(LocateEvent e) {
    // TODO Auto-generated method stub
    super.locate(e);
}
```

示例

- 案例说明
  -- 接收当前位置信息

- 实现方案
  -- 在 locate 方法中接收位置信息

➢ 实例代码

Java

```
package kd.mobil.demo.plugin;

import java.util.EventObject;

import kd.bos.bill.AbstractMobBillPlugIn;
import kd.bos.bill.events.LocateEvent;
import kd.bos.entity.MobLocation;

public class MobileDemoPlugin extends AbstractMobBillPlugIn{

    @Override
    public void locate(LocateEvent e) {
        // TODO Auto-generated method stub
        super.locate(e);
        MobLocation location = e.getMobLocation(); //MobLocation对象包含的就是定位信息
    }
}
```

10.5. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=10 进行反馈。

11. 移动端单据列表插件

11.1. 插件基类

11.1.1. 插件接口及基类

移动端单据列表界面插件基类为 AbstractMobListPlugin，派生自插件基类 AbstractMobFormPlugin，并实现了插件接口 ListRowClickListener，IMobListPlugin：

Java

```
package kd.bos. list.plugin;
public class AbstractMobListPlugin extends AbstractMobFormPlugin implements
ListRowClickListener, IMobListPlugin {
```

11.1.2. 创建并注册插件

请参阅单据创建与注册插件的介绍。

11.2. 视图模型

11.2.1. 接口与实现类

移动端单据列表的视图模型接口为 IMobileListView，派生自 IListView，IMobileView：

Java

```
package kd.bos. list;
public interface IMobileListView extends IMobileView, IListView {
```

11.2.2. 功能方法及使用

IMobileListView 继承 IListView，IMobileView。没有新增方法。

11.3. 数据模型

待更新

11.4. 插件事件

移动端单据，除了 PC 端移动端单据列表，除了 PC 端单据列表插件事件，还支持如下插件事件：

方法	说明
mobileSearchInit	
mobileSearchFocus	
afterPushDownRefresh	
mobFilterSortSearchClick	
mobileSearchTextChange	

11.5. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=11 进行反馈。

12. 单据操作插件

系统预置了一批操作，可以绑定在界面菜单、按钮上；
用户点击按钮时，即自动执行这些操作，完成特定的操作功能。

这些预置的操作，可以根据是否写数据入库，分为两大类：
1. 表单操作：仅对表单界面及界面上的数据进行处理，不会写数据入库，如”新建”、”关闭”操作；
2. 实体操作：会写数据入库，只能配置在单据、基础资料上，如”保存”、”提交”、”审核”操作；

可以给单据上绑定的实体操作，单独开发操作插件，对操作写数据入库的过程，进行干预。

本节介绍单据操作插件及其事件。

表单操作，不会写数据入库，只能针对表单界面及界面数据进行处理，不支持操作插件，但可以开发表单插件，对表单操作进行适度的干预；

与表单操作相关的表单插件事件，包括：
1. 菜单、按钮点击事件(beforeitemClick, itemClick, beforeClick, click)，可以取消操作的执行；
2. 操作执行前后事件(beforeDoOperation, afterDoOperation)，准备操作参数、接收操作执行结果刷新界面；

➢ 附：实体操作清单

下表列出了系统预置的实体操作，只能对这些操作，开发操作插件（其他未列出的操作属于界面操作，不能开发操作插件）：

操作	功能说明
保存 (save)	把单据数据包中的数据，存入单据表格
保存并新增 (saveandnew)	把单据数据包中的数据，存入单据表格； 保存成功后，清空界面数据，进入新增状态
状态转换 (statusconvert)	切换单据状态字段值，并更新单据表格
提交 (submit)	切换单据状态字段值为”审核中”，并更新单据表格
提交并新增 (submitandnew)	切换单据状态字段值为”审核中”， 并更新单据表格 提交成功后，清空界面数据，进入新增状态
撤销 (unsubmit)	切换单据状态值为”暂存”， 并更新单据表格
审核 (audit)	切换单据状态值为”已审核”，并记录审核人，更新单据表格
反审核 (unaudit)	切换单据状态值为”暂存”，并清除审核人，更新单据表格
禁用 (disable)	切换使用状态为”停用”，并更新单据表格
启用 (enable)	切换使用状态为”启用”，并更新单据表格
作废 (invalid)	切换作废状态为”作废”，并更新单据表格
生效 (valid)	切换作废状态为”正常”，并更新单据表格
删除 (delete)	从单据表格中，删除单据数据
空操作 (donothing)	特殊的实体操作，本身并不写单据入库； 但会执行操作的全部过程，包括权限校验、写日志，支持操作插件开发，触发全部操作事件；

12.1. 插件基类

12.1.1. 插件接口及基类

单据操作插件接口为 IOperationServicePlugIn、IOperationService；
系统预置了一个操作插件基类 AbstractOperationServicePlugIn，实现了上述操作插件接口；

自定义的操作插件，扩展预置的操作插件基类 AbstractOperationServicePlugIn 即可。

插件基类 AbstractOperationServicePlugIn 内置了如下属性方法和本地变量，供插件访问，用以获取操作执行上下文：

方法	说明
billEntityType	单据主实体
operateMeta	操作配置，据此获知当前执行的操作信息
operationResult	操作结果，可以向其中添加操作提示信息
getOption()	自定义操作参数字典，可以包含各种自定义操作参数

12.1.2. 创建并注册插件

自定义的单据操作插件，需要扩展单据操作插件基类 AbstractOperationServicePlugIn。

如下例：

Java

```
package kd.bos.plugin.sample.bill.bizoperation.bizcase;

import kd.bos.entity.plugin.AbstractOperationServicePlugIn;
import kd.bos.entity.plugin.AddValidatorsEventArgs;
import kd.bos.entity.plugin.PreparePropertysEventArgs;
import kd.bos.entity.plugin.args.AfterOperationArgs;
import kd.bos.entity.plugin.args.BeforeOperationArgs;
import kd.bos.entity.plugin.args.BeginOperationTransactionArgs;
import kd.bos.entity.plugin.args.EndOperationTransactionArgs;
import kd.bos.entity.plugin.args.RollbackOperationArgs;

/**
 * 演示操作插件全部事件的捕获及触发时机
```

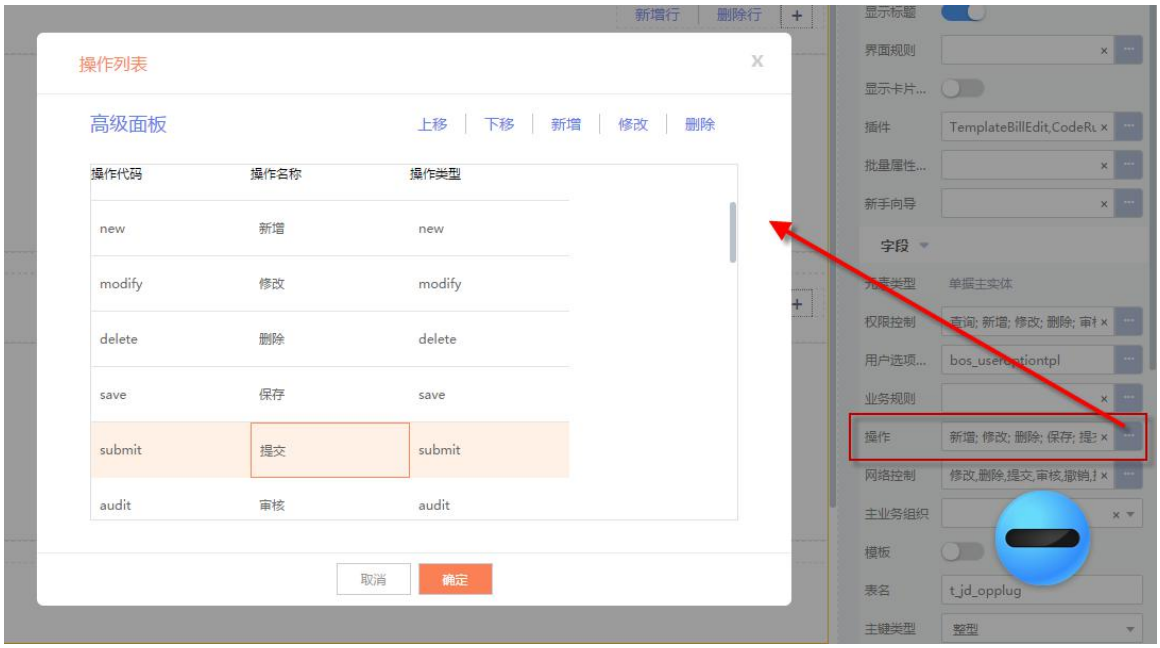
```
 *
 * @author rd_johnnyding
 * @remark
 * 本示例代码，捕捉的事件，由前往后触发
 *
 */
public class OperationEventSample extends AbstractOperationServicePlugIn {

    /**
     * 操作执行，加载单据数据包之前，触发此事件；
     *
     * @remark
     * 在单据列表上执行单据操作，传入的是单据内码；
     * 系统需要先根据传入的单据内码，加载单据数据包，其中只包含操作要用到的字段，然后再执行操作；
     * 在加载单据数据包之前，操作引擎触发此事件；
     *
     * 插件需要在此事件，添加需要用到的字段；
     * 否则，系统加载的单据数据包，可能没有插件要用到的字段值，从而引发中断
     */
    @Override
    public void onPreparePropertys(PreparePropertysEventArgs e) {
        this.printEventInfo("", "");
    }

    /**
     * 构建好操作校验器之后，执行校验之前，触发此事件；
     *
     * @remark
     * 插件可以在此事件，增加自定义操作校验器，或者去掉内置的校验器
     */
    @Override
    public void onAddValidators(AddValidatorsEventArgs e) {
        this.printEventInfo("", "");
    }

    /**
     * 操作校验通过之后，开启事务之前，触发此事件；
     *
     * @remark
     * 插件可以在此事件，对通过校验的数据，进行整理
     */
    @Override
    public void beforeExecuteOperationTransaction(BeforeOperationArgs e) {
        this.printEventInfo("", "");
```

```
}

/**
 * 操作校验通过，开启了事务，准备把数据提交到数据库之前触发此事件；
 *
 * @remark
 * 可以在此事件，进行数据同步处理
 */
@Override
public void beginOperationTransaction(BeginOperationTransactionArgs e) {
    this.printEventInfo("", "");
}

/**
 * 单据数据已经提交到数据库之后，事务未提交之前，触发此事件；
 *
 * @remark
 * 可以在此事件，进行数据同步处理；
 */
@Override
public void endOperationTransaction(EndOperationTransactionArgs e) {
    this.printEventInfo("", "");
}

/**
 * 操作事务提交失败，事务回滚之后触发此事件；
 *
 * @remark
 * 该方法在事务异常后执行，插件可以在此事件，对没有事务保护的数据更新进行补偿
 */
@Override
public void rollbackOperation(RollbackOperationArgs e) {
    this.printEventInfo("", "");
}

/**
 * 操作执行完毕，事务提交之后，触发此事件；
 *
 * @remark
 * 插件可以在此事件，处理操作后续事情，与操作事务无关
 */
@Override
public void afterExecuteOperationTransaction(AfterOperationArgs e) {
    this.printEventInfo("", "");
```

```
    }

    private void printEventInfo(String eventName, String argString){
        String msg = String.format("%s : %s", eventName, argString);
        System.out.println(msg);
    }
}
```

已经开发好的单据操作插件，通过如下步骤，绑定到操作上：

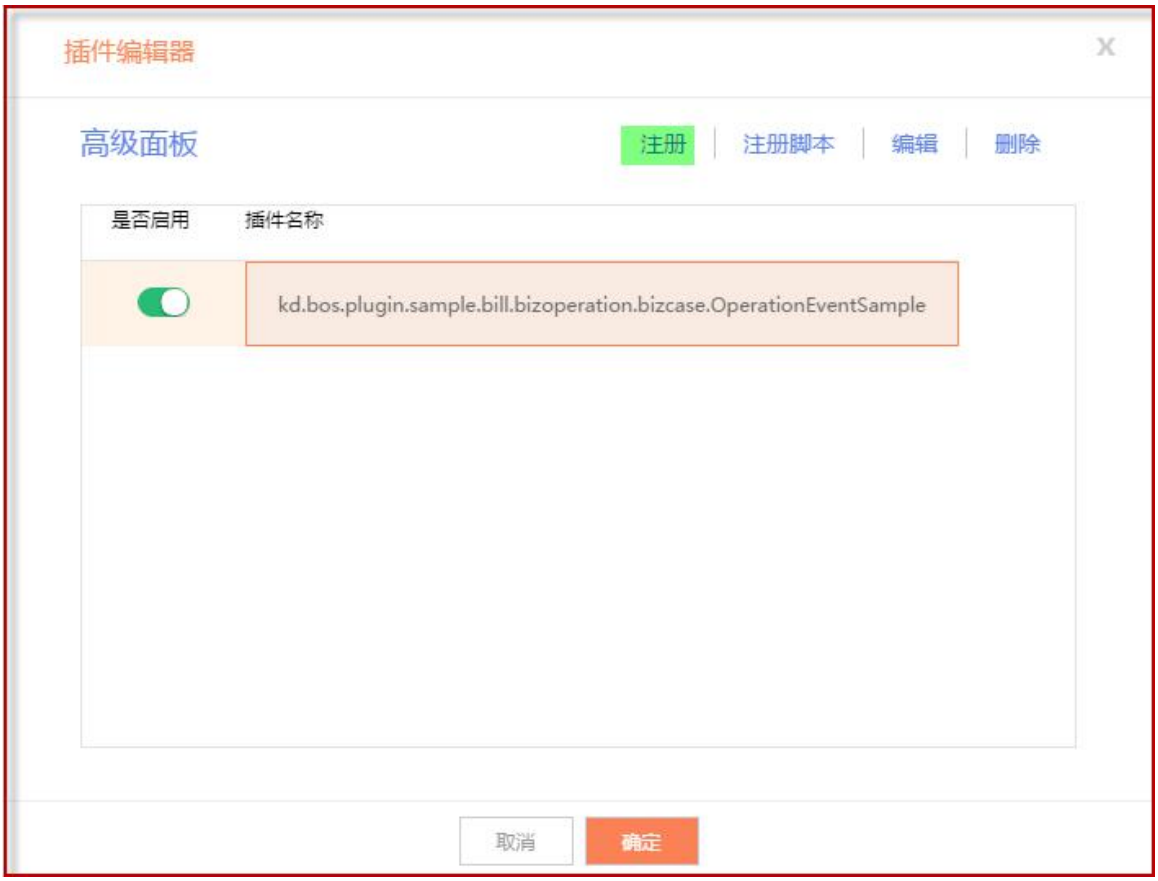
- 修改单据操作列表



- 修改操作的配置


操作编辑
操作类型
submit
操作编码
submit
操作名称
提交
参数设置
其他控制
操作后刷新字段
校验规则描述
合法性校验, 组合字段唯一性校验, 必录校验
服务端服务
权限项描述
提交
服务插件
[{"_Type_":"Plugin","MetadataId":"ZPYG29333==","ClassName":"kd.bos.plugin.sampl
操作前确认提示
取消
确定

- 绑定自定义操作插件



12.2. 插件事件

单据操作插件，提供如下插件事件：

事件	触发时机
onPreparePropertys	在单据列表上执行单据操作，系统需要先根据传入的单据内码，加载单据数据包； 在加载单据数据包之前，触发此事件； 插件需要在此事件，添加需要用到的字段；
onAddValidators	系统预置的操作校验器加载完毕，执行校验之前，触发此事件；
beforeExecuteOperationTransaction	操作校验通过之后，开启事务之前，触发此事件；
beginOperationTransaction	操作校验通过，开启了事务，准备把数据提交到数据库之前触发此事件；
endOperationTransaction	数据已经提交到数据库之后，事务未提交之前，触发此事件；
rollbackOperation	操作事务提交失败，事务回滚之后触发此事件；
afterExecuteOperationTransaction	操作执行完毕，事务提交之后，触发此事件；
setContext	设置上下文
initializeOperationResult	初始化操作结果集

12.2.1. onPreparePropertys 事件

事件触发时机

操作执行，加载单据数据包之前，触发此事件；

在单据列表上执行单据操作，传入的是单据内码；
操作引擎需要先根据传入的单据内码，加载单据数据包，其中只包含操作要用到的字段，然后再执行操作；在加载单据数据包之前，触发此事件；

在单据维护界面，执行单据操作时，传入的是单据数据包，不需要操作引擎自行加载单据，不会触发此事件；

插件需要在此事件，添加需要用到的字段；否则，操作引擎加载出的单据数据包，可能没有插件要用到的字段值，从而引发中断

代码模板

Java

```
package kd.bos.plugin.sample.bill.bizoperation.template;

import kd.bos.entity.plugin.AbstractOperationServicePlugIn;
import kd.bos.entity.plugin.PreparePropertysEventArgs;

public class OnPreparePropertys extends AbstractOperationServicePlugIn {

    @Override
    public void onPreparePropertys(PreparePropertysEventArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class PreparePropertysEventArgs`

- ✓ `public List<String> getFieldKeys()`：需加载的字段标识

示例

参阅 onAddValidators 事件示例：
在 onPreparePropertys 事件，添加自定义校验器要用到的字段；

12.2.2. onAddValidators 事件

事件触发时机

构建好操作校验器之后，执行操作校验之前，触发此事件；

插件可以在此事件，增加自定义操作校验器，或者去掉内置的校验器。

代码模板

Java

```java
package kd.bos.plugin.sample.bill.bizoperation.template;

import kd.bos.entity.plugin.AbstractOperationServicePlugIn;
import kd.bos.entity.plugin.AddValidatorsEventArgs;

public class OnAddValidators extends AbstractOperationServicePlugIn {

    @Override
    public void onAddValidators(AddValidatorsEventArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class AddValidatorsEventArgs`
  - ✓ `public List<AbstractValidator> getValidators()`：预置的校验器

- ✓ `public void addValidator(AbstractValidator validator)`：添加自定义校验器
- ✓ `public DynamicObject[] getDataEntities()`：本次操作、待校验的数据包

示例

➢ 案例说明

1. 单据体中有"最迟送货日期"字段，子单据体中有"预计送货日期"字段；
2. 预计送货日期，不能晚于最迟送货日期；
3. 如果不满足此条件，该单不允许继续操作
4. 批量操作单据时，把不满足条件的单据剔除出去；其他满足条件的单据，可以继续操作；

➢ 实现方案

1. 捕获 onPreparePropertys 事件，要求加载预计送货日期、最迟送货日期字段
2. 捕获 onAddValidators 事件，添加自定义校验器
3. 实现自定义校验器，对预计送货日期、最迟送货日期，进行比较

➢ 实例代码

Java

```java
package kd.bos.plugin.sample.bill.bizoperation.bizcase;

import java.text.SimpleDateFormat;
import java.util.Date;

import kd.bos.entity.ExtendedDataEntity;
import kd.bos.entity.formula.RowDataModel;
import kd.bos.entity.plugin.AbstractOperationServicePlugIn;
import kd.bos.entity.plugin.AddValidatorsEventArgs;
import kd.bos.entity.plugin.PreparePropertysEventArgs;
import kd.bos.entity.validate.AbstractValidator;

public class OnAddValidatorsSample extends AbstractOperationServicePlugIn {

    /**
     * 操作执行前，准备加载单据数据之前，触发此事件
     * @remark
     * 插件可以在此事件中，指定需要加载的字段
     */
    @Override
    public void onPreparePropertys(PreparePropertysEventArgs e) {
        // 要求加载预计送货日期、最迟送货日期字段
        e.getFieldKeys().add(DelivaryDateValidator.KEY_DELIVERYDATE);
        e.getFieldKeys().add(DelivaryDateValidator.KEY_LASTDATE);
```

```
    }

    /**
     * 执行操作校验前，触发此事件
     * @remark
     * 插件可以在此事件，调整预置的操作校验器；或者增加自定义操作校验器
     */
    @Override
    public void onAddValidators(AddValidatorsEventArgs e) {
        // 添加自定义的校验器：送货日期校验器
        e.addValidator(new DelivaryDateValidator());
    }
}

/**
 * 自定义操作校验器：校验送货日期
 *
 * @author rd_JohnnyDing
 */
class DelivaryDateValidator extends AbstractValidator {

    /** 预计送货日期字段标识 */
    public final static String KEY_DELIVERYDATE = "deliverydate";
    /** 最迟送货日期字段标识 */
    public final static String KEY_LASTDATE = "lastdate";

    /**
     * 返回校验器的主实体：系统将自动对此实体数据，逐行进行校验
     */
    @Override
    public String getEntityKey() {
        return this.entityKey;
    }

    /**
     * 给校验器传入上下文环境及单据数据包之后，调用此方法；
     * @remark
     * 自定义校验器，可以在此事件进行本地变量初始化：如确认需要校验的主实体
     */
    @Override
    public void initializeConfiguration() {
        super.initializeConfiguration();
        // 在此方法中，确认校验器检验的主实体：送货子单据体
        // 需要对送货子单据体行，逐行判断预计送货日期
```

```
        this.entityKey = "subentryentity";
    }

    /**
     * 校验器初始化完毕，从单据数据包中，提取出了主实体数据行，开始校验前，调用此方法；
     * @remark
     * 此方法，比initializeConfiguration更晚调用；
     * 在此方法调用this.getDataEntities()，可以获取到需校验的主实体数据行
     * 不能在本方法中，确认需要校验的主实体
     */
    @Override
    public void initialize() {
        super.initialize();
    }


    /**
     * 执行自定义校验
     */
    @Override
    public void validate() {

        // 定义一个行数据存取模型：用于方便的读取本实体、及父实体、单据头上的字段
        RowDataModel rowDataModel = new RowDataModel(this.entityKey,
this.getValidateContext().getSubEntityType());
        SimpleDateFormat timesdf = new SimpleDateFormat("yyyy-MM-dd");

        // 逐行校验预计送货
        for(ExtendedDataEntity rowDataEntity : this.getDataEntities()){
            rowDataModel.setRowContext(rowDataEntity.getDataEntity());
            Date deliveryDate = (Date)rowDataModel.getValue(KEY_DELIVERYDATE);
            Date lastDate = (Date)rowDataModel.getValue(KEY_LASTDATE);

            if (deliveryDate.compareTo(lastDate) > 0 ){
                // 校验不通过，输出一条错误提示
                this.addErrorMessage(rowDataEntity,
                        String.format("预计送货日期(%s)，不能晚于最迟送货日期(%s)！",
                                timesdf.format(deliveryDate), timesdf.format(lastDate)));
            }
        }
    }
}
```

12.2.3. beforeExecuteOperationTransaction 事件

事件触发时机

操作校验通过之后，开启事务存储数据之前，触发此事件；

插件可以在此事件，对已经通过校验的数据，进行整理，或者取消操作的执行。

这个事件触发时，还没有启动事务保护，请勿在此修改数据库数据。

代码模板

Java

```java
package kd.bos.plugin.sample.bill.bizoperation.template;

import kd.bos.entity.plugin.AbstractOperationServicePlugIn;
import kd.bos.entity.plugin.args.BeforeOperationArgs;

public class BeforeExecuteOperationTransaction extends AbstractOperationServicePlugIn {

    @Override
    public void beforeExecuteOperationTransaction(BeforeOperationArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class BeforeOperationArgs extends OperationArgs`
  - ✓ `public List<ExtendedDataEntity> getSelectedRows()`：已经通过校验的单据
  - ✓ `public DynamicObject[] getDataEntities()`：本次操作的全部单据，含校验失败的单据
  - ✓ cancel：是否取消后续操作
  - ✓ `public void setCancelMessage(String cancelMessage)`：操作取消原因

示例

➢ 案例说明

1. 子单据体中，有预计送货日期字段，父单据体中有最迟送货日期字段
2. 预计送货日期，不能迟于最迟送货日期；
3. 如果不满足此条件，该单不允许继续操作（批量操作单据时，把这些不满足条件的单据剔除出去）；
4. 其他满足条件的单据，可以继续操作；

➢ 实现方案

1. 本案例的场景，最优方案是采用自定义校验器，见实例 OnAddValidatorsSample ；
2. 本实例演示如何在 beforeExecuteOperationTransaction 事件达成同样的效果；

➢ 实例代码

Java

```java
package kd.bos.plugin.sample.bill.bizoperation.bizcase;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.dataentity.entity.DynamicObjectCollection;
import kd.bos.entity.ExtendedDataEntity;
import kd.bos.entity.MainEntityType;
import kd.bos.entity.formula.RowDataModel;
import kd.bos.entity.plugin.AbstractOperationServicePlugIn;
import kd.bos.entity.plugin.PreparePropertysEventArgs;
import kd.bos.entity.plugin.args.BeforeOperationArgs;
import kd.bos.entity.validate.ErrorLevel;
import kd.bos.entity.validate.ValidationErrorInfo;

public class BeforeExecuteOperationTransactionSample extends AbstractOperationServicePlugIn {

    /** 单据体 */
    private final static String KEY_ENTRYENTITY = "entryentity";
    /** 子单据体 */
    private final static String KEY_SUBENTRYENTITY = "subentryentity";

    /** 预计送货日期字段标识 */
```

```
private final static String KEY_DELIVERYDATE = "deliverydate";
/** 最迟送货日期字段标识 */
private final static String KEY_LASTDATE = "lastdate";

private SimpleDateFormat timesdf = new SimpleDateFormat("yyyy-MM-dd");

/**
 * 操作执行前，准备加载单据数据之前，触发此事件
 * @remark
 * 插件可以在此事件中，指定需要加载的字段
 */
@Override
public void onPreparePropertys(PreparePropertysEventArgs e) {
    // 要求加载预计送货日期、最迟送货日期字段
    e.getFieldKeys().add(KEY_DELIVERYDATE);
    e.getFieldKeys().add(KEY_LASTDATE);
}

/**
 * 操作校验执行完毕，开启事务保存单据之前，触发此事件
 * @remark
 * 可以在此事件，对单据数据包进行整理、取消操作
 */
@Override
public void beforeExecuteOperationTransaction(BeforeOperationArgs e) {

    List<ExtendedDataEntity> passDataEntitys = new ArrayList<>();

    // 逐单校验送货日期
    for(ExtendedDataEntity dataEntity : e.getSelectedRows()){
        if (delivaryDateValidate(dataEntity)){
            passDataEntitys.add(dataEntity);
        }
    }

    // 向系统传回校验通过的单据
    e.getSelectedRows().clear();
    if (passDataEntitys.isEmpty()){
        e.cancel = true;      // 没有单据通过了校验，取消后续操作
    }
    else {
        e.getSelectedRows().addAll(passDataEntitys);
    }
}
```

```
    /**
     * 校验单据的送货日期是否晚于最迟送货日期
     *
     * @param dataEntity
     * @return
     */
    private boolean delivaryDateValidate(ExtendedDataEntity dataEntity){

        // 构建子单据体行数据模型：用于快速访问子单据体行、单据体行、单据头上的字段值
        MainEntityType mainType =
(MainEntityType)dataEntity.getDataEntity().getDataEntityType();
        RowDataModel rowDataModel = new RowDataModel(KEY_SUBENTRYENTITY, mainType);

        // 取全部单据体行
        DynamicObjectCollection entryRows =
dataEntity.getDataEntity().getDynamicObjectCollection(KEY_ENTRYENTITY);
        // 对单据体行循环
        for (DynamicObject entryRow : entryRows){
            // 取子单据体行
            DynamicObjectCollection subEntryRows =
entryRow.getDynamicObjectCollection(KEY_SUBENTRYENTITY);
            // 对子单据体行进行循环
            for (DynamicObject subEntryRow : subEntryRows){
                rowDataModel.setRowContext(subEntryRow);
                Date deliveryDate = (Date)rowDataModel.getValue(KEY_DELIVERYDATE);
                Date lastDate = (Date)rowDataModel.getValue(KEY_LASTDATE);
                if (deliveryDate.compareTo(lastDate) > 0 ){
                    // 校验不通过，输出一条错误提示，并略过后续行的检查
                    this.addErrMessage(dataEntity,
                            String.format("预计送货日期(%s)，不能晚于最迟送货日期(%s)！",
                                    timesdf.format(deliveryDate),
timesdf.format(lastDate)));
                    return false;    // 校验不通过
                }
            }
        }

        return true;
    }

    /**
     * 向操作结果，添加一条错误提示
     *
```

```
     * @param dataEntity
     * @param errMsg
     * @return
     */
    private void addErrMessage(ExtendedDataEntity dataEntity, String errMsg){

        Object pkId = dataEntity.getDataEntity().getPkValue();
        int dataIndex = dataEntity.getDataEntityIndex();
        int rowIndex = 0;
        ErrorLevel errorLevel = ErrorLevel.Error;

        ValidationErrorInfo errInfo = new ValidationErrorInfo("",
                pkId, dataIndex, rowIndex,
                "BeforeExecuteOperationTransactionSample",
                "送货日期检查",
                errMsg,
                errorLevel);

        this.operationResult.addErrorInfo(errInfo);
    }
}
```

12.2.4. beginOperationTransaction 事件

事件触发时机

操作校验通过，开启了事务之后，还没有把数据提交到数据库之前触发此事件；

可以在此事件，进行数据同步处理。

这个事件触发时，系统还没有调用 ORM 引擎存储单据：
此时访问数据库，拿到的单据数据是旧的；
单据数据包中的脏标志也没有被复位；

可能会执行失败的同步处理，如和第三方系统的对接，放在这个事件比较合适。

代码模板

Java

```java
package kd.bos.plugin.sample.bill.bizoperation.template;

import kd.bos.entity.plugin.AbstractOperationServicePlugIn;
import kd.bos.entity.plugin.args.BeginOperationTransactionArgs;

public class BeginOperationTransaction extends AbstractOperationServicePlugIn {

    @Override
    public void beginOperationTransaction(BeginOperationTransactionArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class BeginOperationTransactionArgs extends OperationArgs`
  - ✓ `public DynamicObject[] getDataEntities()`: 已经通过校验的单据
  - ✓ `public void setDataEntities(DynamicObject[] dataEntities)`: 设置允许继续操作的数据
  - ✓ `public void setCancelFormService(boolean cancelFormService)`: 暂未使用
  - ✓ `public void setCancelOperation(boolean cancelOperation)`: 是否取消操作

示例

- 案例说明

1. 银企互联，需要逐条单据提交给银行；
2. 提交数据到银行时，需要捕获错误并提示出来，但不影响其他单据的执行；

- 实现方案

1. 这种场景，没有办法在校验器中提前检查出错误，只有实际执行时，才知道会不会成功；
2. 捕获 beginOperationTransaction 事件，在系统存储单据之前，调用银行接口；
3. 调用银行接口时，捕获错误；如果出错，把数据从待保存数据包中，排除出去；
4. 随后系统保存单据时，就只会存储银联成功的单据；

➢ 实例代码

Java

```java
package kd.bos.plugin.sample.bill.bizoperation.bizcase;

import java.util.ArrayList;
import java.util.List;

import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.entity.plugin.AbstractOperationServicePlugIn;
import kd.bos.entity.plugin.args.BeginOperationTransactionArgs;
import kd.bos.entity.validate.ErrorLevel;
import kd.bos.entity.validate.ValidationErrorInfo;
import kd.bos.exception.ErrorCode;
import kd.bos.exception.KDBizException;

public class BeginOperationTransactionSample extends AbstractOperationServicePlugIn {

    /**
     * 事务开始后，数据还没有提交到数据库时，触发此事件
     */
    @Override
    public void beginOperationTransaction(BeginOperationTransactionArgs e) {

        // 记录执行成功的单据
        List<DynamicObject> successObjs = new ArrayList<>();

        for(DynamicObject obj : e.getDataEntities()){
            try{
                // 逐条处理数据，并捕获预知的业务错误，不影响下张单据
                this.doBizLogic(obj);
                successObjs.add(obj);
            }
            catch (KDBizException exp){
                // 如果本条数据遇到了预知的错误：输出错误原因
                this.operationResult.addErrorInfo(this.buildErrMessage(obj, exp));
            }
        }

        // 向系统回传执行成功的单据
        e.setDataEntities(successObjs.toArray(new DynamicObject[successObjs.size()]));
    }
```

```
    /**
     * 执行银企互联逻辑
     *
     * @param obj
     * @remark
     * 实际银企互联代码略过，直接抛出一个错误，演示对错误的捕捉处理
     */
    private void doBizLogic(DynamicObject obj){
        throw new KDBizException(new
ErrorCode("Sample_BeginOperationTransaction_AddErrorInfo", "银企互联，抛出的随机错误"));
    }

    /**
     * 生成一条错误提示
     *
     * @param obj
     * @param exp
     * @return
     */
    private ValidationErrorInfo buildErrMessage(DynamicObject obj, KDBizException exp){

        Object pkId = obj.getPkValue();
        int dataIndex = 0;
        int rowIndex = 0;
        ErrorLevel errorLevel = ErrorLevel.Error;
        String msg = exp.getMessage();

        ValidationErrorInfo info = new ValidationErrorInfo("",
                pkId, dataIndex, rowIndex,
                exp.getErrorCode().getCode(),
                "银企互联",
                msg,
                errorLevel);

        return info;
    }
}
```

12.2.5. endOperationTransaction 事件

事件触发时机

单据数据已经提交到数据库之后，事务未提交之前，触发此事件；

可以在此事件，进行数据同步处理；

这个事件触发，系统已经调用 ORM 引擎存储了单据，数据已经入库，单据数据包中的脏标志已经复位。

可能执行失败的同步处理，不适合放在这个事件执行，事务回滚比较麻烦。

特别说明：
系统会根据脏标志，判断出数据是否来自数据库，有没有更改：脏标志被复位后，调用 ORM 数据存储引擎，存储数据包，会被 ORM 引擎直接略过，不再更新入库。

代码模板

Java

```java
package kd.bos.plugin.sample.bill.bizoperation.template;

import kd.bos.entity.plugin.AbstractOperationServicePlugIn;
import kd.bos.entity.plugin.args.EndOperationTransactionArgs;

public class EndOperationTransaction extends AbstractOperationServicePlugIn {

    @Override
    public void endOperationTransaction(EndOperationTransactionArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class EndOperationTransactionArgs extends OperationArgs`
  - ✓ `public DynamicObject[] getDataEntities()`：已经保存到数据库的单据

示例

暂缺示例。

12.2.6. rollbackOperation 事件

事件触发时机

操作事务提交失败，事务回滚之后触发此事件；

该方法在事务异常后执行，插件可以在此事件，对没有事务保护的数据更新进行补偿。

代码模板

Java

```java
package kd.bos.plugin.sample.bill.bizoperation.template;

import kd.bos.entity.plugin.AbstractOperationServicePlugIn;
import kd.bos.entity.plugin.args.RollbackOperationArgs;

public class RollbackOperation extends AbstractOperationServicePlugIn {

    @Override
    public void rollbackOperation(RollbackOperationArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class RollbackOperationArgs`
  - ✓ `public DynamicObject[] getDataEntitys()`: 保存失败，被回滚的单据

示例

示例暂缺。

12.2.7. afterExecuteOperationTransaction 事件

事件触发时机

操作执行完毕，事务提交之后，触发此事件；

插件可以在此事件，对操作结果进行整理，或者执行其他无需事务保护的逻辑。

这个事件触发时，事务已经完成并提交，没有了事务保护，请勿在此事件更新数据库。

代码模板

Java

```
package kd.bos.plugin.sample.bill.bizoperation.template;

import kd.bos.entity.plugin.AbstractOperationServicePlugIn;
import kd.bos.entity.plugin.args.AfterOperationArgs;

public class AfterExecuteOperationTransaction extends AbstractOperationServicePlugIn {

    @Override
    public void afterExecuteOperationTransaction(AfterOperationArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

```
//取消操作成功提示
this.operationResult.setShowMessage(false)
//默认为true
```

事件参数

- `public class AfterOperationArgs extends OperationArgs`
  - ✓ `public DynamicObject[] getDataEntities ()`: 操作成功的单据

示例

暂缺示例。

12.2.8. setContext 事件

设置上下文

代码模板

Java

```
package kd.operation.demo.plugin;

import java.util.Map;

import kd.bos.dataentity.OperateOption;
import kd.bos.entity.MainEntityType;
import kd.bos.entity.plugin.AbstractOperationServicePlugIn;

public class OperationDemoPlugin extends AbstractOperationServicePlugIn {

    @Override
    public void setContext(MainEntityType billEntityType, Map<String, Object> operateMeta,
OperateOption option) {
        // TODO Auto-generated method stub
        super.setContext(billEntityType, operateMeta, option);
    }
}
```

示例

待更新

12.2.9. initializeOperationResult 事件

初始化操作结果集

代码模板

```Java
package kd.operation.demo.plugin;

import kd.bos.entity.operate.result.OperationResult;
import kd.bos.entity.plugin.AbstractOperationServicePlugIn;

public class OperationDemoPlugin extends AbstractOperationServicePlugIn {

    @Override
    public void initializeOperationResult(OperationResult result) {
        // TODO Auto-generated method stub
        super.initializeOperationResult(result);
    }
}
```

示例

待更新

12.3. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=12 进行反馈。

13. 单据转换插件

单据转换，能够把 A 单据的数据，根据转换规则，转换生成 B 单据。

单据转换过程，大概分为如下几个步骤：

- 读取源单到目标单之间的全部转换规则；
- 匹配转换规则的业务范围，确定适用于当前所选源单的转换规则；
- 提取转换规则上，字段映射页签及其他页签，使用到的源单字段；
- 根据源单内码，生成源单取数条件；
- 读取源单数据行：
  - ✓ 只读取转换规则用到的源单字段；
  - ✓ 如果源单是整单下推，则读取源单全部行；否则取源单所选行数据；
  - ✓ 把单据头、单据体字段组合在一起，生成拉平后的源单数据行；

- 根据转换规则，数据范围的配置，对源单行数据，进行筛选，剔除不符合条件的行；
- 根据分单、分录行合并策略，对源单数据行进行分组；
- 根据分组后的源单数据行，生成目标单、目标单分录行；
- 逐行填写目标分录行、单据头字段值；
- 在目标单的关联子实体中，记录来源单据信息；
  - ✓ 后续保存目标单时，会根据关联子实体中的来源单据信息，记录单据关联关系及反写；
  - ✓ 如果没有在关联子实体中记录来源单据，则不记录单据关联关系，不能联查，也不能反写；
- 输出生成好的目标单数据包，完成单据转换
  - ✓ 特别说明：单据转换，只是生成目标单数据包，并没有保存入库；

如上单据转换过程，会触发单据转换插件事件，允许插件干预。

本节介绍单据转换插件各事件的触发时机及用途。

13.1. 插件基类

13.1.1. 插件接口及基类

系统预置了单据转换插件基类 AbstractConvertPlugIn，实现了转换插件接口 IConvertPlugin：

Java

```
package kd.bos.entity.botp.plugin;
public class AbstractConvertPlugIn implements IConvertPlugIn {
```

单据转换插件接口是 IConvertPlugin：

Java

```
package kd.bos.entity.botp.plugin;
public interface IConvertPlugIn {
```

13.1.2. 创建并注册插件

自定义单据转换插件，必须扩展插件基类 AbstractConvertPlugIn，绑定到单据转换规则上：

➢ 附：自定义单据转换插件示例

Java

```
package kd.bos.plugin.sample.bill.billconvert.bizcase;

import kd.bos.entity.BillEntityType;
import kd.bos.entity.botp.ConvertOpType;
import kd.bos.entity.botp.ConvertRuleElement;
import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
import kd.bos.entity.botp.plugin.args.AfterBuildQueryParemeterEventArgs;
import kd.bos.entity.botp.plugin.args.AfterConvertEventArgs;
import kd.bos.entity.botp.plugin.args.AfterCreateLinkEventArgs;
import kd.bos.entity.botp.plugin.args.AfterCreateTargetEventArgs;
import kd.bos.entity.botp.plugin.args.AfterFieldMappingEventArgs;
import kd.bos.entity.botp.plugin.args.AfterGetSourceDataEventArgs;
import kd.bos.entity.botp.plugin.args.BeforeBuildGroupModeEventArgs;
import kd.bos.entity.botp.plugin.args.BeforeBuildRowConditionEventArgs;
import kd.bos.entity.botp.plugin.args.BeforeCreateLinkEventArgs;
import kd.bos.entity.botp.plugin.args.BeforeCreateTargetEventArgs;
import kd.bos.entity.botp.plugin.args.BeforeGetSourceDataEventArgs;
import kd.bos.entity.botp.plugin.args.InitVariableEventArgs;

/**
 * 演示单据转换插件事件的触发时机
 *
```

```
 * @author rd_JohnnyDing
 *
 */
public class BillConvertEventSample extends AbstractConvertPlugIn {

    /**
     * 演示如何获取上下文信息
     */
    private void getContext(){
        // 源单主实体
        BillEntityType srcMainType = this.getSrcMainType();
        // 目标单主实体
        BillEntityType tgtMainType = this.getTgtMainType();
        // 转换规则
        ConvertRuleElement rule = this.getRule();
        // 转换方式：下推、选单
        ConvertOpType opType = this.getOpType();
    }

    /**
     * 初始化变量事件
     *
     * @param e
     * @remark
     * 获取上下文信息，构建一些必须的变量
     */
    @Override
    public void initVariable(InitVariableEventArgs e) {
        this.printEventInfo("initVariable", "");
    }

    /**
     * 构建取数参数后事件
     *
     * @param e
     * @remark
     * 添加额外的字段、过滤条件
     */
    @Override
    public void afterBuildQueryParemeter(AfterBuildQueryParemeterEventArgs e) {
        this.printEventInfo("afterBuildQueryParemeter", "");
    }

    /**
```

```
     * 编译数据筛选条件前事件
     *
     * @param e
     * @remark
     * 设置忽略规则原生的条件，改用插件定制条件，或者在规则条件基础上，追加定制条件
     *
     */
    @Override
    public void beforeBuildRowCondition(BeforeBuildRowConditionEventArgs e) {
        this.printEventInfo("beforeBuildRowCondition", "");
    }

    /**
     * 取源单数据前事件
     *
     * @param e
     * @remark
     * 修改取数语句、取数条件
     */
    @Override
    public void beforeGetSourceData(BeforeGetSourceDataEventArgs e) {
        this.printEventInfo("beforeGetSourceData", "");
    }

    /**
     * 取源单数据后事件
     *
     * @param e
     * @remark
     * 根据源单数据，获取其他定制的引用数据；也可以替换系统自动获取到的数据
     */
    @Override
    public void afterGetSourceData(AfterGetSourceDataEventArgs e) {
        this.printEventInfo("afterGetSourceData", "");
    }


    /**
     * 构建分单、行合并模式之前事件
     *
     * @param e
     * @remark
     * 调整分单、合并策略及依赖的字段
     */
```

```
@Override
public void beforeBuildGroupMode(BeforeBuildGroupModeEventArgs e) {
    this.printEventInfo("beforeBuildGroupMode", "");
}


/**
 * 初始化创建目标单据数据包前事件
 *
 * @param e
 * @remark
 * 这个事件，只在选单时触发：
 * 选单时，需要基于现有的目标单数据包，进行追加处理；
 * 插件可以在此事件，获取到现有的目标单数据包，提前进行定制处理
 */
@Override
public void beforeCreateTarget(BeforeCreateTargetEventArgs e) {
    this.printEventInfo("beforeCreateTarget", "");
}

/**
 * 创建目标单据数据包后事件
 *
 * @param e
 * @remark
 * 这个事件，只在下推时触发，把根据分单规则创建好的目标单，传递给插件
 */
@Override
public void afterCreateTarget(AfterCreateTargetEventArgs e) {
    this.printEventInfo("afterCreateTarget", "");
}

/**
 * 目标字段赋值完毕后事件
 *
 * @param e
 * @remark
 * 插件可以在此基础上，继续填写目标字段值
 */
@Override
public void afterFieldMapping(AfterFieldMappingEventArgs e) {
    this.printEventInfo("afterFieldMapping", "");
}
```

```
    /**
     * 记录关联关系前事件
     *
     * @param e
     * @remark
     * 取消记录关联关系
     */
    @Override
    public void beforeCreateLink(BeforeCreateLinkEventArgs e) {
        this.printEventInfo("beforeCreateLink", "");
    }

    /**
     * 记录关联关系后事件
     *
     * @param e
     * @remark
     * 根据系统自动记录的关联关系，进行相关数据的同步携带，如携带其他子单据体数据
     */
    @Override
    public void afterCreateLink(AfterCreateLinkEventArgs e) {
        this.printEventInfo("afterCreateLink", "");
    }

    /**
     * 单据转换后事件，最后执行
     *
     * @param e
     * @remark
     * 插件可以在这个事件中，对生成的目标单数据，进行最后的修改
     */
    @Override
    public void afterConvert(AfterConvertEventArgs e) {
        this.printEventInfo("afterConvert", "");
    }

    private void printEventInfo(String eventName, String argString){
        String msg = String.format("%s : %s", eventName, argString);
        System.out.println(msg);
    }
}
```

13.2. 插件事件

单据转换插件，提供如下插件事件：

事件	触发时机
initVariable	初始化变量事件
afterBuildQueryParemeter	构建取数参数后事件
beforeBuildRowCondition	编译数据筛选条件前事件
beforeGetSourceData	取源单数据前事件
afterGetSourceData	取源单数据后事件
beforeBuildGroupMode	构建分单、行合并模式之前事件
beforeCreateTarget	暂未触发
afterCreateTarget	创建目标单据数据包后事件
afterFieldMapping	目标字段赋值完毕后事件
beforeCreateLink	记录关联关系前事件
afterCreateLink	记录关联关系后事件
afterConvert	单据转换完毕事件，最后执行

13.2.1. initVariable 事件

事件触发时机

开始运行转换规则，创建好了转换规则上绑定的单据转换插件之后，即触发此事件。

插件可以在此事件中，对本地变量进行初始化。

此事件发生时，源单主实体、目标单主实体、转换规则都已经确定，可以基于这些上下文信息，初始化本地变量。

一些通用的单据转换业务插件，需要自动适应各种单据，这个事件就显得比较重要：
可以在转换规则执行前，让通用插件了解到当前的上下文，初始化一些变量，决定后续业务逻辑。

插件可以利用如下方法，获取到源单、目标单主实体、反写规则：

Java

```
    private void getContext(){
        // 源单主实体
        BillEntityType srcMainType = this.getSrcMainType();
        // 目标单主实体
        BillEntityType tgtMainType = this.getTgtMainType();
        // 转换规则
```

```
        ConvertRuleElement rule = this.getRule();
        // 转换方式：下推、选单
        ConvertOpType opType = this.getOpType();
    }
```

代码模板

Java

```
package kd.bos.plugin.sample.bill.billconvert.template;

import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
import kd.bos.entity.botp.plugin.args.InitVariableEventArgs;

public class InitVariable extends AbstractConvertPlugIn {

    @Override
    public void initVariable(InitVariableEventArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class InitVariableEventArgs extends ConvertPluginEventArgs`
  - ✓ 暂无属性

示例

- 案例说明

1. 需要开发一个通用单据转换插件，把目标单单据体上的金额字段值，合计到单据头上；
2. 不同的单据上，单据体、单据头上的金额字段标识不同；
3. 暂时只支持采购费用发票、销售费用发票；

- 实现方案

1. 捕获 initVariable 事件，确认源单、目标单，单据体、单据头各金额字段标识；

2. 在 afterCreateLink 事件，根据单据体金额字段，合计单据头金额字段，并换算本位币金额；

➢ 实例代码

Java

```
package kd.bos.plugin.sample.bill.billconvert.bizcase;

import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;
import java.util.Date;
import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.dataentity.entity.DynamicObjectCollection;
import kd.bos.dataentity.utils.StringUtils;
import kd.bos.entity.ExtendedDataEntity;
import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
import kd.bos.entity.botp.plugin.args.AfterCreateLinkEventArgs;
import kd.bos.entity.botp.plugin.args.InitVariableEventArgs;

public class InitVariableSample extends AbstractConvertPlugIn {

    /** 目标单主实体标识 */
    private String targetEntityNumber;
    /** 目标单单据体标识 */
    private String key_entryentity;

    private boolean isOtherBill = false;

    /** 字段标识，单据头.价税合计_原币*/
    private String key_AllAmountOri;
    /** 字段标识，单据体.价税合计_原币*/
    private String key_AllAmountOri_D;
    /** 字段标识，单据头.价税合计_本位币*/
    private String key_AllAmount;

    /** 字段标识，单据头.不含税金额_原币*/
    private String key_AmountOri;
    /** 字段标识，单据体.不含税金额_原币*/
    private String key_AmountOri_D;
    /** 字段标识，单据头.不含税金额_本位币*/
    private String key_Amount;

    /** 字段标识，单据头.税额_原币*/
    private String key_TaxAmountOri;
```

```
    /** 字段标识，单据体.税额_原币*/
    private String key_TaxAmountOri_D;
    /** 字段标识，单据头.税额_本位币*/
    private String key_TaxAmount;

    /**
     * 初始化事件：根据目标单，确定价税合计金额等字段标识
     */
    @Override
    public void initVariable(InitVariableEventArgs e) {
        this.targetEntityNumber = this.getTgtMainType().getName();
        this.setAmountKey();
    }

    /**
     * 关联关系已经记录，并重算了反写控制字段值之后，触发此事件
     *
     * @remark
     * 金额字段，是反写控制字段：
     * 反写控制字段，需要记录从每条源单行携带过来的值，然后自动合计到单据体字段上；
     * 用户手工修改单据体上的反写控制字段时，会根据原始携带量分配、反写到源单
     *
     * 因此，如果要依赖于反写控制字段值进行运算，必须在afterCreateLink事件之后
     *
     */
   @Override
    public void afterCreateLink(AfterCreateLinkEventArgs e) {

    if (this.isOtherBill){
        // 其他未知的单据，不合计金额
        return;
    }

    ExtendedDataEntity[] billDataEntitys =
e.getTargetExtDataEntitySet().FindByEntityKey(this.targetEntityNumber);
    // 逐单合计单据头金额字段值
    for(ExtendedDataEntity billDataEntity : billDataEntitys){
        this.calcAmount(billDataEntity);
    }
    }

    /**
    * 根据目标单，设置含税价格等字段标识
    */
```

```
private void setAmountKey() {

    if (StringUtils.equals(this.targetEntityNumber, "iv_purexpinv")){
     // 采购费用发票

     this.key_entryentity = "entryentity";

        //价税合计
     this.key_AllAmountOri = "amountori";
     this.key_AllAmountOri_D = "amountori";
     this.key_AllAmount = "amount";

        //不含税金额
     this.key_AmountOri = "headauxpriceori";
     this.key_AmountOri_D = "headauxpriceori";
     this.key_Amount = "headauxprice";

        //税额
     this.key_TaxAmountOri = "deductibletaxori";
     this.key_TaxAmountOri_D = "deductibletaxori";
     this.key_TaxAmount = "deductibletax";
    }
    else if (StringUtils.equals(this.targetEntityNumber, "iv_saleexpinv")){
     // 销售费用发票

     this.key_entryentity = "entryentity";

        //价税合计
     this.key_AllAmountOri = "aftertotaltaxfor";
     this.key_AllAmountOri_D = "allamountori";
     this.key_AllAmount = "aftertotaltax";

        //不含税金额
     this.key_AmountOri = "headauxpriceori";
     this.key_AmountOri_D = "amountori";
     this.key_Amount = "headauxprice";

        //税额
     this.key_TaxAmountOri = "taxamount";
     this.key_TaxAmountOri_D = "detailtaxamountori";
     this.key_TaxAmount = "taxamountori";
    }
    else {
     // 其他单据：不在此插件中计算价税合计
```

```
            isOtherBill = true;
        }
    }

    /**
     * 计算价税合计
     *
     * @param billDataEntity
     */
    private void calcAmount(ExtendedDataEntity billDataEntity){

        long currencyID = (long) billDataEntity.getValue("currencyid_id");                //
原币
        long mainCurrencyID =(long) billDataEntity.getValue("mainbookstdcurrid_id");  // 本币
        long exchangeTypeID = (long) billDataEntity.getValue("exchangetype_id");      // 换算
类型
        Date date = (Date) billDataEntity.getValue("date");      // 业务日期

        // 取当日汇率
        BigDecimal rate = new BigDecimal("0");
        if (currencyID > 0 && mainCurrencyID > 0) {
            rate = this.getExchangeBusRate(currencyID, mainCurrencyID, exchangeTypeID, date);
        }

        // 把汇率回填到单据上
        billDataEntity.setValue("exchangerate", rate);

        // 开始合计金额：逐行循环，汇总到变量上
        BigDecimal taxAmountOriH = new BigDecimal("0");   // 单据头.税额
        BigDecimal noTaxAmountOriH = new BigDecimal("0");   // 单据头.不含税金额
        BigDecimal totalTaxAmountOriH = new BigDecimal("0");   //单据头.价税合计

        DynamicObjectCollection entryRows =
(DynamicObjectCollection)billDataEntity.getValue(this.key_entryentity);
        for (DynamicObject entryRow : entryRows){
            taxAmountOriH = taxAmountOriH.add(entryRow.getBigDecimal(key_TaxAmountOri_D));   //
原币 单据体.税额
            noTaxAmountOriH = noTaxAmountOriH.add(entryRow.getBigDecimal(key_AmountOri_D));
    // 原币 单据体.不含税金额
            totalTaxAmountOriH =
totalTaxAmountOriH.add(entryRow.getBigDecimal(key_AllAmountOri_D)); // 原币 单据体.价税合计
        }

        // 填写单据头.原币各金额
```

```
        billDataEntity.setValue(key_TaxAmountOri, taxAmountOriH);
        billDataEntity.setValue(key_AmountOri, noTaxAmountOriH);
        billDataEntity.setValue(key_AllAmountOri, totalTaxAmountOriH);

        // 本位币各金额 ：原币金额 * 汇率 (四舍五入，10位小数)
        MathContext mc = new MathContext(10, RoundingMode.HALF_UP);
        billDataEntity.setValue(key_TaxAmount, taxAmountOriH.multiply(rate, mc));
        billDataEntity.setValue(key_Amount, noTaxAmountOriH.multiply(rate, mc));
        billDataEntity.setValue(key_AllAmount, totalTaxAmountOriH.multiply(rate, mc));
    }

    /**
     * 取当日汇率
     * @return
     */
    private BigDecimal getExchangeBusRate(long currencyId1, long currencyId2, long exchangeType,
Date date){
    // 略过取汇率的逻辑，直接返回 1
    return new BigDecimal("1");
    }
}
```

13.2.2. afterBuildQueryParemeter 事件

事件触发时机

系统根据转换规则上的字段映射关系，确认好了需要加载的源单字段之后，触发此事件，并传入需转换的源单行过滤条件(FID in [1,2,3,4])。

插件可以在此事件中，增加需要加载的源单字段，调整源单行取数条件。

代码模板

Java

```
package kd.bos.plugin.sample.bill.billconvert.template;

import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
```

```java
import kd.bos.entity.botp.plugin.args.AfterBuildQueryParemeterEventArgs;

public class AfterBuildQueryParemeter extends AbstractConvertPlugIn {

    @Override
    public void afterBuildQueryParemeter(AfterBuildQueryParemeterEventArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class AfterBuildQueryParemeterEventArgs extends ConvertPluginEventArgs`
  - ✓ `public Map<String, String> getSrcFldAlias()`：预计会加载的源单字段及其别名；
  - ✓ `public void addSrcField(String fullPropName)` ：
    - 添加插件需要用到的源单字段；
    - 传入字段标识，如 textfield；
    - 如果要取源单基础资料字段的引用属性，要传入基础资料字段标识及其引用属性，如 basedatafield.name；
    - 不需要在前面带单据体标识；
  - ✓ `public List<QFilter> getQFilters()`：系统根据传入的源单内码，生成的源单取数条件，插件可以调整此集合中的条件对象；

示例

- 案例说明

1. 源单有单据编号、业务日期、金额字段
2. 需要把这三个字段值，拼成一个字符串，填写到目标单内容字段上；
3. 业务日期格式化为 yyyy-MM-dd，金额带币别

- 实现方案

1. 捕获 afterBuildQueryParemeter 事件，要求加载源单单据编号、业务日期、金额、币别字段
2. 捕获 afterFieldMapping 事件，取源单字段值，格式化后填写在目标单上

- 实例代码

Java

```java
package kd.bos.plugin.sample.bill.billconvert.bizcase;

import java.math.BigDecimal;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.List;

import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.entity.ExtendedDataEntity;
import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
import kd.bos.entity.botp.plugin.args.AfterBuildQueryParemeterEventArgs;
import kd.bos.entity.botp.plugin.args.AfterFieldMappingEventArgs;
import kd.bos.entity.botp.runtime.ConvertConst;

public class AfterBuildQueryParemeterSample extends AbstractConvertPlugIn {

    private final static String KEY_BILLNO = "billno";
    private final static String KEY_DATE = "date";
    private final static String KEY_AMOUNT = "amount";
    private final static String KEY_CURRENCYNAME = "currency.name";
    private final static String KEY_CONTENT = "content";

    /**
     * 在开始读取源单数据前，触发此事件
     * @remark
     * 在此事件中，要求加载插件要用到的源单字段
     */
    @Override
    public void afterBuildQueryParemeter(AfterBuildQueryParemeterEventArgs e) {

        e.addSrcField(KEY_BILLNO);        // 单据编号
        e.addSrcField(KEY_DATE);      // 业务日期
        e.addSrcField(KEY_AMOUNT);        // 金额
        e.addSrcField(KEY_CURRENCYNAME);  // 币别.名称 currency.name
    }

    /**
     * 目标单字段值，携带完毕后，触发此事件
     * @remark
     * 在此事件中，自行取源单字段值，格式化后填写到目标单
     */
    @SuppressWarnings("unchecked")
    @Override
    public void afterFieldMapping(AfterFieldMappingEventArgs e) {
```

```
        // 取目标单，单据头数据包 （可能会生成多张单，是个数组）
        String targetEntityNumber = this.getTgtMainType().getName();
        ExtendedDataEntity[] billDataEntitys =
e.getTargetExtDataEntitySet().FindByEntityKey(targetEntityNumber);

        SimpleDateFormat timesdf = new SimpleDateFormat("yyyy-MM-dd");

        // 逐单处理
        for(ExtendedDataEntity billDataEntity : billDataEntitys){

            // 取当前目标单，对应的源单行
            List<DynamicObject> srcRows =
(List<DynamicObject>)billDataEntity.getValue(ConvertConst.ConvExtDataKey_SourceRows);
            // 取源单第一行上的字段值，忽略其他行
            DynamicObject srcRow = srcRows.get(0);

            // 开始格式化字段值
            StringBuilder sBuilder = new StringBuilder();
            sBuilder.append(this.getSrcMainType().getDisplayName().toString());

            // 单据编号字段值
            String billno = (String)e.getFldProperties().get(KEY_BILLNO).getValue(srcRow);
            sBuilder.append(billno).append("; ");

            // 日期
            Object date = e.getFldProperties().get(KEY_DATE).getValue(srcRow);
            if (date != null){
                sBuilder.append("日期: ").append(timesdf.format((Date)date)).append("; ");
            }

            // 金额
            BigDecimal amount =
(BigDecimal)e.getFldProperties().get(KEY_AMOUNT).getValue(srcRow);
            sBuilder.append("金额: ").append(amount.toString());

            // 币别
            String currency =
(String)e.getFldProperties().get(KEY_CURRENCYNAME).getValue(srcRow);
            sBuilder.append(currency);

            // 给目标单字段赋值
            billDataEntity.setValue(KEY_CONTENT, sBuilder.toString());
        }
```

```
    }
}
```

13.2.3. beforeBuildRowCondition 事件

事件触发时机

编译转换规则 - 数据范围，配置的源单数据筛选条件之前，触发此事件。

插件可以在此事件，忽略转换规则上配置的条件，改用插件定制条件，或者追加插件定制条件；

代码模板

Java

```java
package kd.bos.plugin.sample.bill.billconvert.template;

import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
import kd.bos.entity.botp.plugin.args.BeforeBuildRowConditionEventArgs;

public class BeforeBuildRowCondition extends AbstractConvertPlugIn {

    @Override
    public void beforeBuildRowCondition(BeforeBuildRowConditionEventArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class BeforeBuildRowConditionEventArgs extends ConvertPluginEventArgs`
  - ✓ `public void setIgnoreRuleFilterPolicy(boolean ignoreRuleFilterPolicy)`：完全忽略转换规则上的数据范围；
  - ✓ `public List<QFilter> getCustQFilters()`：插件定制条件，用于数据库取数；
  - ✓ `public void setCustFilterExpression(String custFilterExpression)`：插件定制条件表达式，用于内存运算；条件含义需要与 CustQFilters 一致，系统会分别用于不同的时机点；

- ✓ `public void setCustFilterDesc(String custFilterDesc)`：插件定制条件描述，当源单数据行，不符合此条件时，系统会把这段条件描述，提示给用户，告诉用户数据行不允许下推的原因；

示例

➢ 案例说明

1. 单据下推，需要根据系统选项，动态增加数据筛选条件：
   a. 如果勾选了"锁定的订单不允许下推"选项，则不能下推锁定状态为已锁定(B)的单据
   b. 如果没有勾选此选项，则不做此限制

➢ 实现方案

1. 捕捉 beforeBuildRowCondition 事件，读取选项值
2. 如果勾选了选项，则追加数据行筛选条件，只允许下推未锁定的单据

➢ 实例代码

Java

```
package kd.bos.plugin.sample.bill.billconvert.bizcase;

import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
import kd.bos.entity.botp.plugin.args.BeforeBuildRowConditionEventArgs;
import kd.bos.orm.query.QCP;
import kd.bos.orm.query.QFilter;

public class BeforeBuildRowConditionSample extends AbstractConvertPlugIn {

    @Override
    public void beforeBuildRowCondition(BeforeBuildRowConditionEventArgs e) {

        if (!cannotPushLockBill()){
            e.setCustFilterDesc("不允许下推已锁定的单据");        // 给出不允许下推的原因

            // 设置条件表达式，用于脚本执行 （必选）
            e.setCustFilterExpression(" lockstatus = 'A' ");

            // 同时设置具有相同含义的QFilter条件，用于选单数据查询 （必选）
            QFilter qFilter = new QFilter("lockstatus", QCP.equals, "A");
            e.getCustQFilters().add(qFilter);
        }
    }
```

```
    /**
     * 读取业务应用的系统参数值
     * @return
     */
    private boolean cannotPushLockBill(){
        // 实际业务系统并没有这个选项，本演示代码，直接返回false，演示添加条件
        return false;
        //return (boolean)SystemParamterServiceHelper.getParameter(0, 0, "sal",
"cannotpushlockbill");
    }
}
```

13.2.4. beforeGetSourceData 事件

事件触发时机

系统读取源单数据前，触发此事件。

插件可以对取数 SELECT 子句、取数条件，做最后的修改。

代码模板

Java

```
package kd.bos.plugin.sample.bill.billconvert.template;

import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
import kd.bos.entity.botp.plugin.args.BeforeGetSourceDataEventArgs;

public class BeforeGetSourceData extends AbstractConvertPlugIn {

    @Override
    public void beforeGetSourceData(BeforeGetSourceDataEventArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class BeforeGetSourceDataEventArgs extends ConvertPluginEventArgs`
  - ✓ `public String getSelectSQL()`：取数 SELECT 子句，包含了全部源单字段及别名；
  - ✓ `public List<QFilter> getQFilters()`：源单行内码筛选条件；

示例

不建议在这个事件调整取数 SELECT 子句、取数条件；
类似需求，放在 afterBuildQueryParemeter 事件比较合适。

示例略。

13.2.5. afterGetSourceData 事件

事件触发时机

源单行数据读取完毕，并按照转换规则上的数据范围，对源单数据行进行筛选之后，触发此事件。

插件可以根据通过条件的源单行数据，获取其他定制的引用数据；
也可以直接替换掉系统自动获取到的源单行数据，改用插件自定读取的源单行数据；

代码模板

Java

```
package kd.bos.plugin.sample.bill.billconvert.template;

import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
import kd.bos.entity.botp.plugin.args.AfterGetSourceDataEventArgs;

public class AfterGetSourceData extends AbstractConvertPlugIn {

    @Override
    public void afterGetSourceData(AfterGetSourceDataEventArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class AfterGetSourceDataEventArgs extends ConvertPluginEventArgs`
  - ✓ `public Map<String, DynamicProperty> getFldProperties()`：源单字段与字段别名映射字典，可以通过字段标识，取到字段属性对象，然后使用属性对象到源单数据行中取字段值；
  - ✓ `public List<DynamicObject> getSourceRows()`：通过了数据筛选条件的全部源单行；

示例

暂无示例。

13.2.6. beforeBuildGroupMode 事件

事件触发时机

系统按照单据转换规则 – 分单策略，对读取到的源单行，进行分组前，触发此事件。

插件可以在此事件，调整分单依赖的字段，影响后续的分单。

单据转换规则的分单策略，把源单行进行分组，不同的组生成不同的目标单：

- 一对一：一张源单，生成一张目标单，即按照源单单据内码分组；
- 多对一：本次选择的全部源单，下推到一张目标单中，即按照常量值分组，全部单，都会分到一个组；
- 按规则分单：本次选择的全部源单行，按照分单字段值进行分组；

单据转换规则支持的分录行合并策略，把源单行进行分组，不同的组合并为一条目标单分录行：

- 一对一：一条源单分录行，生成一条目标单分录行，即按照源单分录行内码分组；
- 多对一：本次选择的全部源单行，合并为一条目标单分录行，即按常量分组，全部源单行，都回分到一个组；
- 按规则合并：本次选择的全部源单行，按照字段值进行分组；相同的字段值分为一组，合并为一条分录行；

正常情况下，可以配置转换规则 – 分单策略，控制分单、合并。

插件可以根据实际业务数据，在此事件中，动态调整分单、合并依赖的字段，从而影响后续的分单。

代码模板

Java

```
package kd.bos.plugin.sample.bill.billconvert.template;

import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
import kd.bos.entity.botp.plugin.args.BeforeBuildGroupModeEventArgs;

public class BeforeBuildGroupMode extends AbstractConvertPlugIn {

    @Override
    public void beforeBuildGroupMode(BeforeBuildGroupModeEventArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- public class BeforeBuildGroupModeEventArgs extends ConvertPluginEventArgs
  - ✓ public String getHeadGroupKey(): 现有的分单依据字段；
  - ✓ public void setHeadGroupKey(String headGroupKey) : 设置新的分单依据字段
  - ✓ public String getEntryGroupKey(): 现有的单据体行合并依据字段；
  - ✓ public void setEntryGroupKey(String entryGroupKey): 设置单据体行合并依据字段；

- ✓ `public String getSubEntryGroupKey()`：子单据体行合并字段；
- ✓ `public void setSubEntryGroupKey(String subEntryGroupKey)`：设置子单据体行合并字段

示例

暂无示例。

13.2.7. beforeCreateTarget 事件

暂未触发。

13.2.8. afterCreateTarget 事件

事件触发时机

根据分单策略，创建好目标单据、单据体数据行之后，触发此事件；

此事件发生时，目标单各行数据包字段仅填写了默认值。后续代码会按照字段映射关系，携带源单字段值，覆盖默认值。

插件可以在此事件，对目标单字段设置默认值，或者根据源单字段值，动态生成目标单字段值。

代码模板

Java

```java
package kd.bos.plugin.sample.bill.billconvert.template;

import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
import kd.bos.entity.botp.plugin.args.AfterCreateTargetEventArgs;
```

```
public class AfterCreateTarget extends AbstractConvertPlugIn {

    @Override
    public void afterCreateTarget(AfterCreateTargetEventArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class AfterCreateTargetEventArgs extends ConvertPluginEventArgs`
  - ✓ `public ExtendedDataEntitySet getTargetExtDataEntitySet()`：生成的目标单扩展数据包；
  - ✓ `public Map<String, DynamicProperty> getFldProperties()` ：源单字段与源单行数据包属性对象映射字典，需要据此到源单行中取需要的字段值；

示例

暂缺示例。

13.2.9. afterFieldMapping 事件

事件触发时机

按照字段映射配置，把源单字段值，填写到了目标单字段上之后，触发此事件。

插件可以在此事件中，对目标单字段值，进行修订、计算、汇总等，或者根据生成的目标单字段值，进行拆单、拆行。

本事件与 afterCreateTarget 事件的差别：
1. 本事件在 afterCreateTarget 事件之后触发；
2. afterCreateTarget 事件发生时，目标单字段还没有赋值，适合赋默认值；
3. 本事件发生时，目标单字段已经根据字段映射关系，赋值完毕，适合字段值计算；

代码模板

Java

```
package kd.bos.plugin.sample.bill.billconvert.template;

import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
import kd.bos.entity.botp.plugin.args.AfterFieldMappingEventArgs;

public class AfterFieldMapping extends AbstractConvertPlugIn {

    @Override
    public void afterFieldMapping(AfterFieldMappingEventArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class AfterFieldMappingEventArgs extends ConvertPluginEventArgs`
  - ✓ `public ExtendedDataEntitySet getTargetExtDataEntitySet()`：生成的目标单扩展数据包；
  - ✓ `public Map<String, DynamicProperty> getFldProperties()` ：源单字段与源单行数据包属性对象映射字典，需要据此到源单行中取需要的字段值；

示例

参阅 afterBuildQueryParemeter 事件示例：
在 afterFieldMapping 事件中，组合、格式化多个源单字段值，填写到目标单字段上。

13.2.10. beforeCreateLink 事件

事件触发时机

目标单字段值填写完毕，开始在目标单关联子实体中，记录源单信息之前，触发此事件。

插件可以在此事件，撤销记录源单信息。

代码模板

Java

```java
package kd.bos.plugin.sample.bill.billconvert.template;

import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
import kd.bos.entity.botp.plugin.args.BeforeCreateLinkEventArgs;

public class BeforeCreateLink extends AbstractConvertPlugIn {

    @Override
    public void beforeCreateLink(BeforeCreateLinkEventArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class BeforeCreateLinkEventArgs extends ConvertPluginEventArgs`
  - ✓ `public void setCancel(boolean cancel)`：取消记录关联关系；
  - ✓ `public ExtendedDataEntitySet getTargetExtDataEntitySet()`：生成的目标单扩展数据包；
  - ✓ `public Map<String, DynamicProperty> getFldProperties()`：源单字段与源单行数据包属性对象映射字典，需要据此到源单行中取需要的字段值；

示例

➢ 案例说明：

1. 根据动态条件，决定是否记录与源单的关联关系

➢ 实现方案：

1. 捕获 beforeCreateLink 事件
2. 判断条件，如果条件满足，则取消记录关联关系

➢ 实例代码：

Java

```
package kd.bos.plugin.sample.bill.billconvert.bizcase;

import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
import kd.bos.entity.botp.plugin.args.BeforeCreateLinkEventArgs;

public class BeforeCreateLinkSample extends AbstractConvertPlugIn {

    @Override
    public void beforeCreateLink(BeforeCreateLinkEventArgs e) {
        e.setCancel(this.isCancelLink());
    }

    private boolean isCancelLink(){
        return true;
    }
}
```

13.2.11. afterCreateLink 事件

事件触发时机

系统在目标单关联子表，记录好源单信息、重算了反写控制字段值之后，触发此事件。

插件可以在此事件，根据目标单记录的源单信息，携带其他数据到目标单。

代码模板

```Java
package kd.bos.plugin.sample.bill.billconvert.template;

import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
import kd.bos.entity.botp.plugin.args.AfterCreateLinkEventArgs;

public class AfterCreateLink extends AbstractConvertPlugIn {

    @Override
    public void afterCreateLink(AfterCreateLinkEventArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class AfterCreateLinkEventArgs extends ConvertPluginEventArgs`
  - ✓ `public ExtendedDataEntitySet getTargetExtDataEntitySet()`：生成的目标单扩展数据包；
  - ✓ `public Map<String, DynamicProperty> getFldProperties()`：源单字段与源单行数据包属性对象映射字典，需要据此到源单行中取需要的字段值；

示例

参阅 initVariable 事件示例，在 afterCreateLink 事件，根据来源单据取值计算单据头金额。

13.2.12. afterConvert 事件

事件触发时机

目标单据生成完毕，触发此事件。

这个事件，是最后触发的，至此，全部业务逻辑已经执行完毕。

插件可以在这个事件，对生成的目标单数据，进行最后的调整。

代码模板

Java

```java
package kd.bos.plugin.sample.bill.billconvert.template;

import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
import kd.bos.entity.botp.plugin.args.AfterConvertEventArgs;

public class AfterConvert extends AbstractConvertPlugIn {

    @Override
    public void afterConvert(AfterConvertEventArgs e) {
        // TODO 在此添加业务逻辑
    }
}
```

事件参数

- `public class AfterConvertEventArgs extends ConvertPluginEventArgs`
  - ✓ `public ExtendedDataEntitySet getTargetExtDataEntitySet()`：生成的目标单扩展数据包；
  - ✓ `public Map<String, DynamicProperty> getFldProperties()` ：源单字段与源单行数据包属性对象映射字典，需要据此到源单行中取需要的字段值；

示例

- 案例说明

1. 采购单据，转固定资产卡片时，需每个物品生成一张卡片，即按数量分单
2. 当前转换规则的分单策略，无法配置出此需求，只能插件开发

- 实现方案

1. 捕获 afterConvert 事件，复制新单：根据资产数量字段值，确认新单的复制次数；

- 实例代码

Java

```
package kd.bos.plugin.sample.bill.billconvert.bizcase;

import java.util.ArrayList;
import java.util.List;

import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.dataentity.utils.OrmUtils;
import kd.bos.entity.ExtendedDataEntity;
import kd.bos.entity.botp.plugin.AbstractConvertPlugIn;
import kd.bos.entity.botp.plugin.args.AfterConvertEventArgs;

public class AfterConvertSample extends AbstractConvertPlugIn {

    private final static String FAREALCARD_ENTITYNAME = "fa_card_real";

    @Override
    public void afterConvert(AfterConvertEventArgs e) {

        // 获取已生成的资产卡片
        ExtendedDataEntity[] billDataEntitys =
e.getTargetExtDataEntitySet().FindByEntityKey(FAREALCARD_ENTITYNAME);

        // 构造 ExtendedDataEntity 时需要的索引值
        int dataIndex = billDataEntitys.length;

        List<ExtendedDataEntity> copyDataEntitys = new ArrayList<>();
        for(ExtendedDataEntity billDataEntity : billDataEntitys){

            // 原始的资产数量
            int qty = (int) billDataEntity.getValue("assetamount");

            // 将资产数量改为1
            billDataEntity.setValue("assetamount", 1);

            // 来源分录拆分序号 从1开始
            int splitSeq = 1;
            billDataEntity.setValue("sourceentrysplitseq", splitSeq++);

            // 复制 （原始的资产数量 - 1）个卡片对象
            for(int i = 1; i < qty; i++){
                DynamicObject copyObj = (DynamicObject)
OrmUtils.clone(billDataEntity.getDataEntity(), false, true);
                copyObj.set("sourceentrysplitseq", splitSeq++);
```

```
                copyDataEntitys.add(new ExtendedDataEntity(copyObj, dataIndex++, 0));
            }
        }

        // 将复制出的单据，放入 targetExtDataEntitySet ， 最终就会生成那么多的卡片
        e.getTargetExtDataEntitySet().AddExtendedDataEntities(FAREALCARD_ENTITYNAME,
copyDataEntitys);
    }
}
```

13.3. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=13 进行反馈。

14. 打印插件

14.1 插件基类

系统预置了打印插件基类 `AbstractPrintServicePlugin`，实现了打印插件接口 `IPrintServicePlugin`：

```
package kd.bos.entity.plugin;

import kd.bos.entity.MainEntityType;

public class AbstractPrintServicePlugin implements IPrintServicePlugin {
```

打印插件接口是 `IPrintServicePlugin`：

```
package kd.bos.entity.plugin;

import kd.bos.entity.MainEntityType;

public interface IPrintServicePlugin {
```

14.2 插件事件

14.2.1 beforeLoadData 事件

事件触发时机

打印引擎运行期间，读取数据前，触发此事件。可通过此事件取消系统默认的读取数据的动作。但必须通过 customPrintDataEntities 事件来重新提供数据，否则将打印出空的内容.

代码模板

```
@Override
public void beforeLoadData(BeforeLoadDataArgs e) {
    e.setCancel(true);
}
```

事件参数

- `public class BeforeLoadDataArgs extends EventObject`
  - ✓ `public String getDataSourceName()`：获取数据源标识名称；
  - ✓ `public void setCancel(boolean cancel)`：设置是否取消；

示例

```
package kd.bos.plugin.sample.print;

import kd.bos.entity.plugin.AbstractPrintServicePlugin;
import kd.bos.entity.plugin.args.BeforeLoadDataArgs;
import kd.bos.entity.plugin.args.CustomPrintDataEntitiesArgs;

public class MyPrintPlugin extends AbstractPrintServicePlugin {
    @Override
    public void beforeLoadData(BeforeLoadDataArgs e) {
        String ds = e.getDataSourceName();
        if ("dsname".equals(ds)) {
            e.setCancel(true);
        }
    }

    @Override
    public void customPrintDataEntities(CustomPrintDataEntitiesArgs e) {
        // 提供自定义数据包
    }

}
```

14.2.2 customPrintDataEntities 事件

事件触发时机

打印引擎运行期间，系统默认读取数据后，触发此事件。可通过此事件对默认读取的数据进行加工。

代码模板

```
@Override
public void customPrintDataEntities(CustomPrintDataEntitiesArgs e) {
    // 提供自定义数据包
    // 新的数据包
    List<DynamicObject> newDataEntities = new ArrayList<>();
    String ds = e.getDataSourceName();
    if ("customdsname".equals(ds)) {
        //构造dynamicobject 放入newDataEntities
        e.setDataEntities(newDataEntities);
    }
}
```

事件参数

- `public class CustomPrintDataEntitiesArgs extends EventObject`
  - ✓ `public String getDataSourceName()`：获取数据源标识名称；
  - ✓ `public Set<String> getCustomFields()`：获取自定义字段的集合；
  - ✓ `public String getPageId() {`：获取 PageId；
  - ✓ `public List<DynamicObject> getDataEntities()`：获取系统默认读取的数据集合；
  - ✓ `public void setDataEntities(List<DynamicObject> dataEntities)`：设置数据包；
  - ✓ `public DynamicObjectType getDynamicObjectType()`： 获取数据包类型；
  - ✓ `public QFilter getFilter()`： 获取过滤条件

示例

示例 1：

场景：需要动态打印一个名称，但是源单的单据头上并没有这样的字段。

模板设置：打印模板中为一个文本控件设置如下属性： 绑定类型属性为”自定义”，数据源属性为标识为”dsname”的数据源，自定义属性为 customfield

插件代码：

```
@Override
public void customPrintDataEntities(CustomPrintDataEntitiesArgs e) {
    // 根据自定义字段对数据包进行修改
    List<DynamicObject> newDataEntities = new ArrayList<>();
    List<DynamicObject> sysDataEntities = e.getDataEntities();
    Set<String> customFields = e.getCustomFields();
    String ds = e.getDataSourceName();
    if ("dsname".equals(ds)) {
        for (DynamicObject dataEntity : sysDataEntities) {
            DynamicObjectType dataEntityType = (DynamicObjectType) dataEntity.getDataEntityType();
            DynamicObjectType cloneType = null;
            try {
                cloneType = (DynamicObjectType) dataEntityType.clone();
                for (String customField : customFields) {
                    DynamicSimpleProperty demoProp = new DynamicSimpleProperty(customField, String.class,
                            "自定义字段默认值");
                    cloneType.registerSimpleProperty(demoProp);
                }
                DynamicObject newObj = (DynamicObject) (new CloneUtils(false, false)).clone(cloneType, dataEntity);
                for (String customField : customFields) {
                    newObj.set(customField, "自定义字段value");
                }
                newDataEntities.add(newObj);
            } catch (CloneNotSupportedException e1) {
            }
        }
    }
}
```

示例 1：

场景：需要在 A 单的模板上，打印 B 单的单据体数据。

模板设置：打印模板的整体属性中，点击”自定义数据源”属性的编辑按钮。设置自定义数据源。

插件代码：

```
public void customPrintDataEntities(CustomPrintDataEntitiesArgs e) {
    List<DynamicObject> newDataEntities = new ArrayList<>();
    DynamicObjectType dt = e.getDynamicObjectType();
    //默认的过滤条件未A单的pk
    QFilter filter = e.getFilter();
    String ds = e.getDataSourceName();
    if ("dsname".equals(ds)) {
        //TODO: 根据filter读取B单数据
        DynamicObject newObj = new DynamicObject(dt);
        //TODO: 将B单数据设置到newObj中;
        newDataEntities.add(newObj);
    }
}
```

示例 1：

场景：报表打印。

说明：报表的默认主数据源标识为本报表的标识，一般用它来做表头。因此我们需要添加自定义数据源用于承载表格显示内容。

模板设置：在打印模板的整体属性中，点击”自定义数据源”属性的编辑按钮，然后设置自定义数据源。

插件代码：

```
@Override
public void customPrintDataEntities(CustomPrintDataEntitiesArgs e) {
    List<DynamicObject> objectList = new ArrayList<>();
    if (e.isMainDs()) {
        // 主数据源
        DynamicObjectType dt = new DynamicObjectType("mainds");
        dt.registerProperty("id", String.class, "1", false);
        dt.registerProperty("main1", String.class, "main1", false);
        for (int i = 1; i < 3; i++) {
            DynamicObject obj = new DynamicObject(dt);
            obj.set("id", i);
            obj.set("main1", "我是主数据源" + i);
            objectList.add(obj);
        }
    } else {
        QFilter filter = e.getFilter();
        // 从数据源，按过滤条件分批次构造，主数据如果有有10条，那么这里就会进入10次。每次的过滤条件都是不一样的，每次都是为自己的从数据源获取对应数据包。
        // 此例进入两次，过滤条件分别为: id = 1, id = 2
    }
    e.setDataEntities(objectList);
}
```

14.2.3 beforeOuputElement 事件

事件触发时机

打印引擎对设计的控件进行解析并生成绘制对象前触发

代码模板

```
@Override
public void beforeOuputElement(OutputElementArgs e) {
    if("text1".equals(e.getKey())){
        //Do something
    }
}
```

事件参数

- `public class OutputElementArgs extends EventObject`
  - ✓ `public String getKey()`：获取控件标识；
  - ✓ `public String getCurrentDataSource()`：获取控件绑定数据源；
  - ✓ `public IPrintScriptable getOutput()`：获取控件输出对象；

示例

```
@Override
public void beforeOuputElement(OutputElementArgs e) {
    if("text1".equals(e.getKey())){
        e.getOutput().setHide(true);
    }
}
```

14.2.4 afterOutputElement 事件

事件触发时机

打印引擎对设计的控件进行解析并生成绘制对象后触发

代码模板

```
@Override
public void afterOutputElement(OutputElementArgs e) {
    if("text1".equals(e.getKey())){
        //Do something
    }
}
```

事件参数

- `public class OutputElementArgs extends EventObject`
  - ✓ `public String getKey()`：获取控件标识；
  - ✓ `public String getCurrentDataSource()`：获取控件绑定数据源；
  - ✓ `public IPrintScriptable getOutput()`：获取控件输出对象；

示例

场景：

1、标识为”text”的控件绑定了一个布尔的字段，需要展示位是/否

2、标识为”text1” 的控件需要输出为页码

3、标识为”text2” 的控件设计时绑定为 fielda，现需要输出为 fieldb 的值。

插件代码：

```
@Override
public void afterOutputElement(OutputElementArgs e) {
    if (e.getKey().equals("text")) {
        IPrintScriptable apw = e.getOutput();
        Object value = apw.getFieldValue("fieldb");
        if ("true".equals(value)) {
            apw.setValue("是");
        } else{
            apw.setValue("否");
        }
    } else if (e.getKey().equals("text1")) {
        /*获取页码并赋值给text1控件*/
        IPrintScriptable  apw =  e.getOutput();
        int pageNumber = apw.getPageNumber();
        apw.setValue(pageNumber);
    } else if (e.getKey().equals("text2")) {
        /*获取同一数据源下绑定的字段为fieldb的值并赋值给text2控件*/
        IPrintScriptable  apw =  e.getOutput();
        Object newValue = apw.getFieldValue("fieldb");
        apw.setValue(newValue);
    }
}
```

14.3 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=14 进行反馈。

15. 单据反写

通过单据转换生成的下游单据，保存、审核时，会执行反写规则，反写源单。

单据反写源单的过程，支持插件开发。

本节介绍单据反写插件各事件的触发时机及用途。

15.1.1. 插件接口及基类

系统预置了单据反写插件基类 AbstractWriteBackPlugIn，实现了插件接口 IWriteBackPlugIn：

Java

```
package kd.bos.entity.botp.plugin;
```

```
public class AbstractWriteBackPlugIn implements IWriteBackPlugIn {
```

单据反写插件接口是 IConvertPlugin：

Java

```
package kd.bos.entity.botp.plugin;
public interface IWriteBackPlugIn{
```

15.1.2. 创建并注册插件

15.2. 插件事件

单据反写插件，提供如下反写事件：

事件	触发时机
getTargetSubMainType	下游目标单主实体模型
getCurrLinkSetItem	当前处理的关联主实体
prepareProperty s	在读取下游目标单数据之前，触发此事件：用于添加需要加载的目标单字段
beforeTrack	构建本关联主实体全部关联记录前，触发此事件：用于取消关联、反写
beforeCreateArticulationRow	构建本关联主实体，单行数据与源单的关联记录前，触发此事件：用于取消本行的关联、反写
beforeExecWriteBackRule	开始分析反写规则，计算反写量前触发此事件：用于取消当前反写规则的执行
afterCalcWriteValue	基于下游单据当前行，反写值计算完毕后，触发此事件：用于修正反写量，调整对各源单行的分配量
beforeReadSourceBill	读取源单数据之前，触发此事件：用于添加需要加载的源单字段
afterReadSourceBill	读取源单数据之后，触发此事件：供插件读取相关的第三方单据
afterCommitAmount	执行反写规则，把当前反写量，写到源单行之后，触发此事件：用于对源单行，进行连锁更新
beforeExcessCheck	对源单行反写执行完毕，超额检查前，触发此事件：用于取消超额检查
afterExcessCheck	对源单行反写执行完毕，超额检查完毕后，触发此事件：用于控制是否中止反写、提示超额，修正提示内容
beforeCloseRow	关闭上游行前调用
afterCloseRow	对上游行进行了关闭状态填写后调用
beforeSaveSourceBill	反写规则执行完毕后，源单数据保存到数据库之前调用

afterSaveSourceBill	源单数据保存到数据库后调用
rollbackSave	保存失败，触发此事件，通知插件回滚数据
finishWriteBack	反写所有逻辑已经执行完毕后触发，用于通知插件释放资源，如插件申请的网控

15.2.1. getTargetSubMainType 事件

代码模板

Java

```java
package kd.writeback.demo.plugin;

import kd.bos.entity.BillEntityType;
import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public BillEntityType getTargetSubMainType() {
        // TODO Auto-generated method stub
        return super.getTargetSubMainType();
    }
}
```

示例

待更新

15.2.2. getCurrLinkSetItem 事件

代码模板

Java

```java
package kd.writeback.demo.plugin;

import kd.bos.entity.LinkSetItemElement;
import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
```

```
public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public LinkSetItemElement getCurrLinkSetItem() {
        // TODO Auto-generated method stub
        return super.getCurrLinkSetItem();
    }
}
```

示例

待更新

15.2.3. prepareProperty s 事件

代码模板

Java

```
package kd.writeback.demo.plugin;

import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
import kd.bos.entity.botp.plugin.args.PreparePropertysEventArgs;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public void preparePropertys(PreparePropertysEventArgs e) {
        // TODO Auto-generated method stub
        super.preparePropertys(e);
    }
}
```

示例

待更新

15.2.4. beforeTrack 事件

代码模板

Java

```
package kd.writeback.demo.plugin;

import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
import kd.bos.entity.botp.plugin.args.BeforeTrackEventArgs;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public void beforeTrack(BeforeTrackEventArgs e) {
        // TODO Auto-generated method stub
        super.beforeTrack(e);
    }
}
```

示例

待更新

15.2.5. beforeCreateArticulationRow 事件

代码模板

Java

```
package kd.writeback.demo.plugin;

import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
import kd.bos.entity.botp.plugin.args.BeforeCreateArticulationRowEventArgs;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public void beforeCreateArticulationRow(BeforeCreateArticulationRowEventArgs e) {
```

```
        // TODO Auto-generated method stub
        super.beforeCreateArticulationRow(e);
    }
}
```

示例

待更新

15.2.6. beforeExecWriteBackRule 事件

代码模板

Java

```
package kd.writeback.demo.plugin;

import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
import kd.bos.entity.botp.plugin.args.BeforeExecWriteBackRuleEventArgs;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public void beforeExecWriteBackRule(BeforeExecWriteBackRuleEventArgs e) {
        // TODO Auto-generated method stub
        super.beforeExecWriteBackRule(e);
    }
}
```

示例

待更新

15.2.7. afterCalcWriteValue 事件

代码模板

Java

```java
package kd.writeback.demo.plugin;

import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
import kd.bos.entity.botp.plugin.args.AfterCalcWriteValueEventArgs;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public void afterCalcWriteValue(AfterCalcWriteValueEventArgs e) {
        // TODO Auto-generated method stub
        super.afterCalcWriteValue(e);
    }
}
```

示例

待更新

15.2.8. beforeReadSourceBill 事件

代码模板

Java

```java
package kd.writeback.demo.plugin;

import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
import kd.bos.entity.botp.plugin.args.BeforeReadSourceBillEventArgs;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public void beforeReadSourceBill(BeforeReadSourceBillEventArgs e) {
```

```
        // TODO Auto-generated method stub
        super.beforeReadSourceBill(e);
    }
}
```

示例

待更新

15.2.9. afterReadSourceBill 事件

代码模板

Java

```
package kd.writeback.demo.plugin;

import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
import kd.bos.entity.botp.plugin.args.AfterReadSourceBillEventArgs;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public void afterReadSourceBill(AfterReadSourceBillEventArgs e) {
        // TODO Auto-generated method stub
        super.afterReadSourceBill(e);
    }
}
```

示例

待更新

15.2.10. afterCommitAmount 事件

代码模板

Java

```java
package kd.writeback.demo.plugin;

import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
import kd.bos.entity.botp.plugin.args.AfterCommitAmountEventArgs;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public void afterCommitAmount(AfterCommitAmountEventArgs e) {
        // TODO Auto-generated method stub
        super.afterCommitAmount(e);
    }
}
```

示例

待更新

15.2.11. beforeExcessCheck 事件

代码模板

Java

```java
package kd.writeback.demo.plugin;

import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
import kd.bos.entity.botp.plugin.args.BeforeExcessCheckEventArgs;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public void beforeExcessCheck(BeforeExcessCheckEventArgs e) {
```

```
        // TODO Auto-generated method stub
        super.beforeExcessCheck(e);
    }
}
```

示例

待更新

15.2.12. afterExcessCheck 事件

代码模板

Java

```
package kd.writeback.demo.plugin;

import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
import kd.bos.entity.botp.plugin.args.AfterExcessCheckEventArgs;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public void afterExcessCheck(AfterExcessCheckEventArgs e) {
        // TODO Auto-generated method stub
        super.afterExcessCheck(e);
    }
}
```

示例

待更新

15.2.13. beforeCloseRow 事件

代码模板

Java

```java
package kd.writeback.demo.plugin;

import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
import kd.bos.entity.botp.plugin.args.BeforeCloseRowEventArgs;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public void beforeCloseRow(BeforeCloseRowEventArgs e) {
        // TODO Auto-generated method stub
        super.beforeCloseRow(e);
    }
}
```

示例

待更新

15.2.14. afterCloseRow 事件

代码模板

Java

```java
package kd.writeback.demo.plugin;

import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
import kd.bos.entity.botp.plugin.args.AfterCloseRowEventArgs;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public void afterCloseRow(AfterCloseRowEventArgs e) {
```

```
        // TODO Auto-generated method stub
        super.afterCloseRow(e);
    }
}
```

示例

待更新

15.2.15. beforeSaveSourceBill 事件

代码模板

Java

```java
package kd.writeback.demo.plugin;

import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
import kd.bos.entity.botp.plugin.args.BeforeSaveSourceBillEventArgs;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public void beforeSaveSourceBill(BeforeSaveSourceBillEventArgs e) {
        // TODO Auto-generated method stub
        super.beforeSaveSourceBill(e);
    }
}
```

示例

待更新

15.2.16. afterSaveSourceBill 事件

代码模板

Java

```java
package kd.writeback.demo.plugin;

import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
import kd.bos.entity.botp.plugin.args.AfterSaveSourceBillEventArgs;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public void afterSaveSourceBill(AfterSaveSourceBillEventArgs e) {
        // TODO Auto-generated method stub
        super.afterSaveSourceBill(e);
    }
}
```

示例

待更新

15.2.17. rollbackSave 事件

代码模板

Java

```java
package kd.writeback.demo.plugin;

import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
import kd.bos.entity.botp.plugin.args.RollbackSaveEventArgs;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public void rollbackSave(RollbackSaveEventArgs e) {
```

```
        // TODO Auto-generated method stub
        super.rollbackSave(e);
    }
}
```

示例

待更新

15.2.18. finishWriteBack 事件

代码模板

Java

```
package kd.writeback.demo.plugin;

import kd.bos.entity.botp.plugin.AbstractWriteBackPlugIn;
import kd.bos.entity.botp.plugin.args.FinishWriteBackEventArgs;

public class WriteBackDemoPlugin extends AbstractWriteBackPlugIn{

    @Override
    public void finishWriteBack(FinishWriteBackEventArgs e) {
        // TODO Auto-generated method stub
        super.finishWriteBack(e);
    }
}
```

示例

待更新

15.3. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId

=kdec_quesfb_plugin&chapter=15 进行反馈。

16. 报表取数插件

16.1. 插件基类

16.1.1. 插件接口及基类

报表取数插件，必须从插件基类 AbstractReportListDataPlugin 中派生。插件基类 AbstractReportListDataPlugin 内置了如下方法，供插件访问，用以获取操作执行上下文：

事件	说明
query	报表查询方法 说明：参数 queryParam 为查询参数，selectedObj 为左树（表）右表时选中左树（表）对象，左树时为节点 ID，左表时为选中行数据
getColumns	改变列信息 说明：参数 columns 为报表设计列，返回值为页面显示列
setProgress	设置进度，用于异步查询时界面显示进度
getQueryParam	获取查询条件
getSelectedObj	获取左表、左树选择行或节点

16.1.2. 创建并注册插件

自定义的报表取数插件，需要继承插件基类 AbstractReportListDataPlugin。

如下例：

Java

```
package kd.bos.demo.plugin;

import java.util.List;

import kd.bos.algo.DataSet;
import kd.bos.entity.report.AbstractReportColumn;
import kd.bos.entity.report.AbstractReportListDataPlugin;
import kd.bos.entity.report.ReportColumn;
import kd.bos.entity.report.ReportQueryParam;
import kd.bos.servicehelper.QueryServiceHelper;
```

```
public class ReportDemo1Plugin extends AbstractReportListDataPlugin{

    @Override
    public DataSet query(ReportQueryParam arg0, Object arg1) throws Throwable {
        // TODO Auto-generated method stub
        return dataSet;
    }

    @Override
    public List<AbstractReportColumn> getColumns(List<AbstractReportColumn> columns) throws
Throwable {
        // TODO Auto-generated method stub
        return columns;
    }
}
```

16.2. 插件事件

16.2.1. query 事件

代码模板

Java

```
package kd.bos.demo.plugin;

import java.util.List;

import kd.bos.algo.DataSet;
import kd.bos.db.DBRoute;
import kd.bos.entity.report.AbstractReportColumn;
import kd.bos.entity.report.AbstractReportListDataPlugin;
import kd.bos.entity.report.ReportColumn;
import kd.bos.entity.report.ReportQueryParam;
import kd.bos.servicehelper.QueryServiceHelper;

public class ReportDemo1Plugin extends AbstractReportListDataPlugin{

    @Override
    public DataSet query(ReportQueryParam arg0, Object arg1) throws Throwable {
        // TODO Auto-generated method stub
        return dataSet;
```

```
    }
}
```

事件参数

示例

➢ 案例说明

-- 报表需要展示某单据的部分字段，并且对其中一列整数字段进行合计展示。

➢ 实现方案

-- 先查询指定单据的部分字段的数据集 dataset，然后再创建一个一行数据的 dataset 用来做合计行，将两个 dataset 进行 union，其中合计行可以用 sum 函数完成。

➢ 实例代码

Java

```java
package kd.bos.demo.plugin;

import java.util.List;

import kd.bos.algo.Algo;
import kd.bos.algo.DataSet;
import kd.bos.entity.report.AbstractReportColumn;
import kd.bos.entity.report.AbstractReportListDataPlugin;
import kd.bos.entity.report.ReportColumn;
import kd.bos.entity.report.ReportQueryParam;
import kd.bos.orm.query.QFilter;
import kd.bos.servicehelper.QueryServiceHelper;

public class ReportDemo1Plugin extends AbstractReportListDataPlugin{

    @Override
    public DataSet query(ReportQueryParam arg0, Object arg1) throws Throwable {
        DataSet dataSet = QueryServiceHelper.queryDataSet(this.getClass().getName(),
"kdec_billdemo4", "integerfield1 as integerfield, billno as textfield", null, null);
        DataSet dataSet2 = dataSet.groupBy(null).sum("integerfield").finish();
        DataSet ds = dataSet.addField("'合计'","integerfield");
        return dataSet.union(ds);
    }
}
```

16.2.2. getColumns 事件

代码模板

Java

```
package kd.bos.demo.plugin;

import java.util.List;

import kd.bos.algo.DataSet;
import kd.bos.db.DBRoute;
import kd.bos.entity.report.AbstractReportColumn;
import kd.bos.entity.report.AbstractReportListDataPlugin;
import kd.bos.entity.report.ReportColumn;
import kd.bos.entity.report.ReportQueryParam;
import kd.bos.servicehelper.QueryServiceHelper;

public class ReportDemo1Plugin extends AbstractReportListDataPlugin{

    @Override
    public List<AbstractReportColumn> getColumns(List<AbstractReportColumn> columns) throws
Throwable {
        // TODO Auto-generated method stub
        return columns;
    }
}
```

事件参数

示例

- 案例说明
  -- 报表需要动态的对某一列进行列冻结

- 实现方案
  --在 getColumns 事件里对指定的列的属性进行修改，设置其 Freeze 属性为 ture。

- 实例代码

Java

```
package kd.bos.demo.plugin;
```

```
import java.util.List;

import kd.bos.algo.Algo;
import kd.bos.algo.DataSet;
import kd.bos.entity.report.AbstractReportColumn;
import kd.bos.entity.report.AbstractReportListDataPlugin;
import kd.bos.entity.report.ReportColumn;
import kd.bos.entity.report.ReportQueryParam;
import kd.bos.orm.query.QFilter;
import kd.bos.servicehelper.QueryServiceHelper;

public class ReportDemoPlugin extends AbstractReportListDataPlugin{

    @Override
    public List<AbstractReportColumn> getColumns(List<AbstractReportColumn> columns) throws
Throwable {
        // TODO Auto-generated method stub
        for(int i = 0; i < columns.size(); i ++) {
            ReportColumn rColumn = (ReportColumn) columns.get(i);
            String key = rColumn.getFieldKey();
            if(key.equals("textfield")) {
                rColumn.setFreeze(true);
                columns.set(i, rColumn);
            }
        }
        return columns;
    }
}
```

16.3. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=16 进行反馈。

17. 报表界面插件

17.1. 插件基类

17.1.1. 插件接口及基类

报表界面插件基类为 AbstractReportFormPlugin，继承自动态表单界面插件基类 AbstractFormPlugin：

Java

```
package kd.bos.bill;
public class AbstractReportFormPlugin extends AbstractFormPlugin  {
```

17.1.2. 创建并注册插件

注册报表插件方式，与注册动态界面插件一致，只是插件基类必须改为 AbstractReportFormPlugin。

17.2. 视图模型

17.2.1. 接口与实现类

报表视图模型接口为 IReportView，派生自动态表单视图模型接口 IFormView，增加了少量仅用于报表界面的控制方法（见后文介绍）：

Java

```
package kd.bos.report;
public interface IReportView extends IFormView {
```

报表视图模型实现类为 ReportView，派生自动态表单视图模型实现类 FormView，实现了报表视图模型接口 IReportView，并重写了部分动态表单视图模型的方法：

Java

```
package kd.bos.mvc.report;
public class ReportView extends FormView implements
IReportView,IExportExcelOperate,TreeNodeClickListener,RowClickEventListener  {
```

报表插件，可以直接把插件 getView()方法返回的实例，转为 IReportView 的实例：

Java

```
IReportView reportView= (IReportView)this.getView();
```

17.2.2. 功能方法及使用

报表视图模型接口 IReportView，派生自动态表单视图模型接口 IFormView，建议先阅读动态表单章节，了解 IFormView 接口提供的方法。

IReportView 新增加了如下方法：

方法	说明
refresh	刷新数据
showData	展示数据
isAsynQuery	是否异步查询
getProgress	当前进度
getQueryParam	获取查询参数

17.3. 数据模型

17.3.1. 接口与实现类

报表数据模型 IReportListModel

17.3.2. 功能方法及使用

17.4. 插件事件

方法	说明
filterContainerInit	初始化过滤容器触发方法
filterContainerBeforeF7Select	过滤容器内 F7 弹出前的处理方法
initDefaultQueryParam	初始化默认查询参数
processRowData	行数据处理 说明：参数 gridPK 为表格控件标识，参数 rowData 为一页数据，参数 queryParam 为查询参数
packageData	打包数据给前端时触发
formatDisplayFilterField	格式化主界面显示的筛选过滤字段信息
verifyQuery	查询前条件验证
beforeQuery	查询前置处理
afterQuery	查询后置处理
afterCreateColumn	表格列创建完成后置事件

17.4.1. filterContainerInit 事件

初始化过滤容器触发该事件，可在事件中对过滤容器的数据进行修改。

代码模板

Java

```
package kd.report.demo.plugin;

import kd.bos.entity.report.ReportQueryParam;
import kd.bos.form.control.events.FilterContainerInitEvent;
import kd.bos.report.plugin.AbstractReportFormPlugin;

public class DemoReportFormPlugin extends AbstractReportFormPlugin{

    @Override
    protected void filterContainerInit(FilterContainerInitEvent contInitEvent, ReportQueryParam queryParam) {
        // TODO Auto-generated method stub
        super.filterContainerInit(contInitEvent, queryParam);
    }
}
```

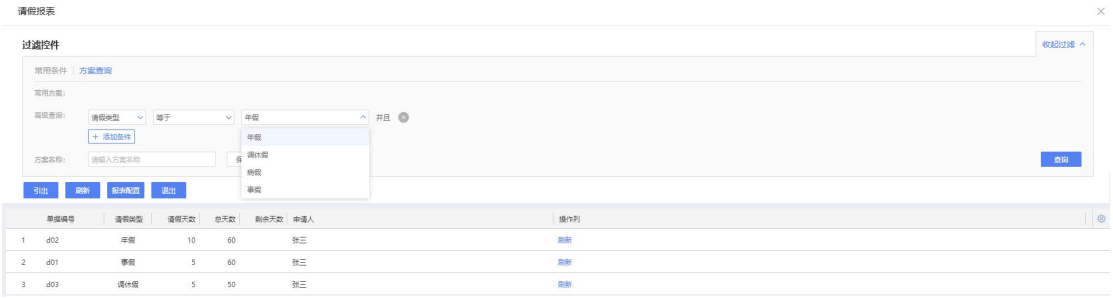
事件参数

示例

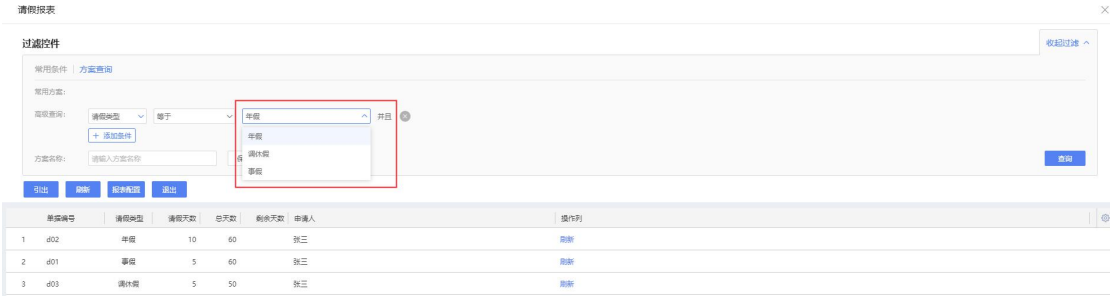
- 案例说明
  -- 对报表过滤条件中，方案查询的请假类型的枚举值进行部分删除。使得过滤中下拉列表的值中的事假不显示。

- 实现方案
  -- 在 filterContainerInit 方法中对请假类型下拉列表中的值删除。

- 案例演示
  实现前：



实现后：



➢ 实例代码

Java

```
package kd.report.demo.plugin;

import java.util.List;

import kd.bos.entity.report.ReportQueryParam;
import kd.bos.filter.FilterColumn;
import kd.bos.filter.SchemeFilterColumn;
import kd.bos.form.control.events.FilterContainerInitEvent;
import kd.bos.form.field.ComboItem;
import kd.bos.report.plugin.AbstractReportFormPlugin;

public class DemoReportFormPlugin extends AbstractReportFormPlugin{

    @Override
    protected void filterContainerInit(FilterContainerInitEvent contInitEvent, ReportQueryParam
queryParam) {
        super.filterContainerInit(contInitEvent, queryParam);
        List<FilterColumn> sColumns = contInitEvent.getSchemeFilterColumns();
        for (FilterColumn filterColumn : sColumns) {
            String filedName = filterColumn.getFieldName();
            if(filedName.equals("kdec_applytype")) {
                SchemeFilterColumn sFilter = (SchemeFilterColumn)filterColumn;
                List<ComboItem> cItems = sFilter.getComboItems();
                for (ComboItem comboItem : cItems) {
```

```
                    if(comboItem.getValue().equals("C")) {
                        cItems.remove(comboItem);
                    }
                }
                sFilter.setComboItems(cItems);
            }
        }
    }
}
```

17.4.2. filterContainerBeforeF7Select 事件

过滤容器内 F7 弹出前触发该事件，可进行 f7 过滤。

代码模板

Java

```
package kd.report.demo.plugin;

import kd.bos.form.field.events.BeforeFilterF7SelectEvent;
import kd.bos.report.plugin.AbstractReportFormPlugin;

public class DemoReportFormPlugin extends AbstractReportFormPlugin{

    @Override
    public void filterContainerBeforeF7Select(BeforeFilterF7SelectEvent args) {
        // TODO Auto-generated method stub
        super.filterContainerBeforeF7Select(args);
    }
}
```

事件参数

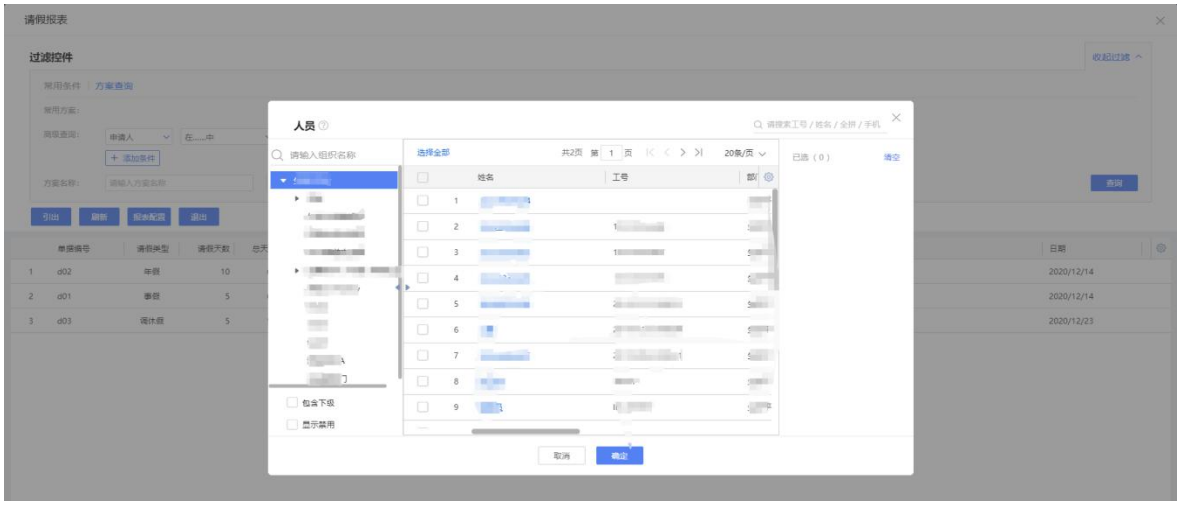
示例

- 案例说明
  -- 对报表过滤条件中，方案查询的申请人进行过滤，使得 f7 页面只有当前用户的选项

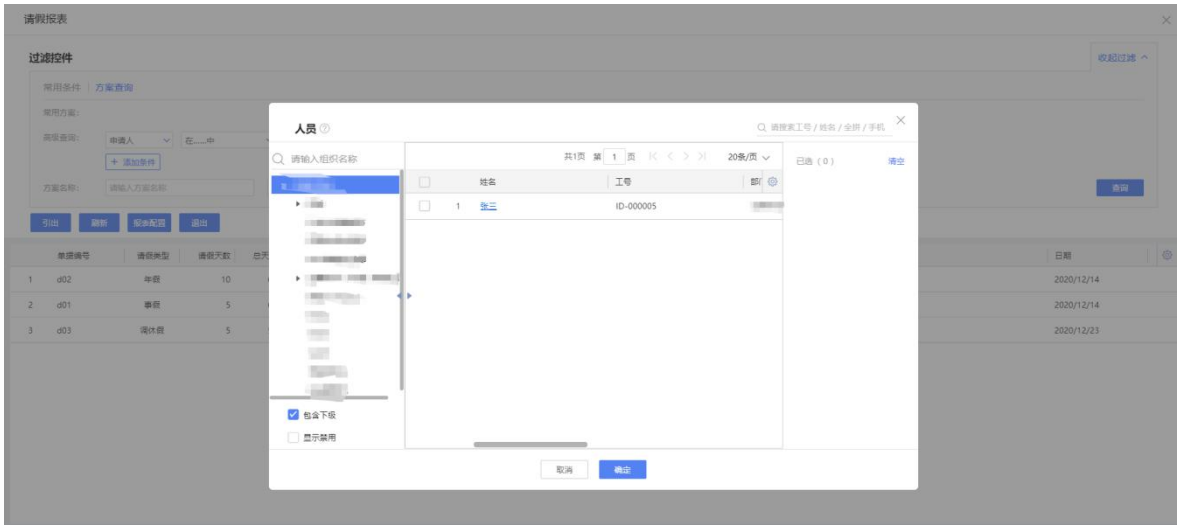
- 实现方案
  -- 在 filterContainerBeforeF7Selec 方法中对申请人 f7 事件进行过滤。

➢ 案例演示

实现前：



实现后：



➢ 实例代码

Java

```
package kd.report.demo.plugin;

import java.util.ArrayList;
import java.util.List;

import kd.bos.context.RequestContext;
import kd.bos.form.field.events.BeforeFilterF7SelectEvent;
import kd.bos.orm.query.QCP;
import kd.bos.orm.query.QFilter;
import kd.bos.report.plugin.AbstractReportFormPlugin;

public class DemoReportFormPlugin extends AbstractReportFormPlugin{
```

```
    @Override
    public void filterContainerBeforeF7Select(BeforeFilterF7SelectEvent args) {
        // TODO Auto-generated method stub
        super.filterContainerBeforeF7Select(args);
        String userId = RequestContext.get().getUserId();
        QFilter qFilter = new QFilter("id", QCP.equals, userId);
        List<QFilter> qFilters = new ArrayList<>();
        qFilters.add(qFilter);
        // 设置列表过滤条件
        args.setQfilters(qFilters);
    }
}
```

17.4.3. processRowData 事件

行数据处理，可以对查询出的数据进行计算等。

代码模板

Java

```
package kd.report.demo.plugin;

import kd.bos.dataentity.entity.DynamicObjectCollection;
import kd.bos.entity.report.ReportQueryParam;
import kd.bos.report.plugin.AbstractReportFormPlugin;

public class DemoReportFormPlugin extends AbstractReportFormPlugin{

    @Override
    public void processRowData(String gridPK, DynamicObjectCollection rowData, ReportQueryParam
queryParam) {
        // TODO Auto-generated method stub
        super.processRowData(gridPK, rowData, queryParam);
    }
}
```

事件参数

示例

- 案例说明
  -- 报表取数中有请假天数和总天数，剩余天数需要自己计算显示

- 实现方案
  -- 在 processRowData 方法中通过总天数和请假天数计算剩余天数，并赋值返回。

- 案例演示
  实现前：

  实现后：

- 实例代码

Java

```
package kd.report.demo.plugin;

import java.util.Iterator;

import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.dataentity.entity.DynamicObjectCollection;
import kd.bos.entity.report.ReportQueryParam;
import kd.bos.report.plugin.AbstractReportFormPlugin;

public class DemoReportFormPlugin extends AbstractReportFormPlugin{

    @Override
    public void processRowData(String gridPK, DynamicObjectCollection rowData, ReportQueryParam
queryParam) {
        // TODO Auto-generated method stub
        super.processRowData(gridPK, rowData, queryParam);
```

```
        Iterator<DynamicObject> iterator = rowData.iterator();
        while (iterator.hasNext()) {
            DynamicObject row = iterator.next();
            int residueday = row.getInt("kdec_totalday") - row.getInt("kdec_applyday");
            row.set("kdec_residueday", residueday);
        }
    }
}
```

17.4.4. packageData 事件

把数据打包给前端时触发，可以对指定单元格赋值。

代码模板

Java

```
package kd.report.demo.plugin;

import kd.bos.entity.datamodel.events.PackageDataEvent;
import kd.bos.report.plugin.AbstractReportFormPlugin;

public class DemoReportFormPlugin extends AbstractReportFormPlugin{

    @Override
    public void packageData(PackageDataEvent packageDataEvent) {
        // TODO Auto-generated method stub
        super.packageData(packageDataEvent);
    }
}
```

事件参数

示例

- 案例说明
  -- 将调休假显示动态修改为 100 天

- 实现方案
  -- 在 packageData 方法中对指定的单元格赋值

➢ 案例演示

实现前：

实现后：

➢ 实例代码

Java

```java
package kd.report.demo.plugin;

import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.entity.datamodel.events.PackageDataEvent;
import kd.bos.entity.report.ReportColumn;
import kd.bos.list.column.ListOperationColumnDesc;
import kd.bos.report.plugin.AbstractReportFormPlugin;

public class DemoReportFormPlugin extends AbstractReportFormPlugin{

    @Override
    public void packageData(PackageDataEvent packageDataEvent) {
        // TODO Auto-generated method stub
        super.packageData(packageDataEvent);
        if (packageDataEvent.getSource() instanceof ListOperationColumnDesc) {
        } else if (packageDataEvent.getSource() instanceof ReportColumn) {
            DynamicObject dObject = packageDataEvent.getRowData();
            String applyType = dObject.getString("kdec_applytype");
            if(applyType.equals("A")) {
               ReportColumn column = (ReportColumn) packageDataEvent.getSource();
               System.out.println(column.getFieldKey());
               if ("kdec_totalday".equals(column.getFieldKey())) {
                   packageDataEvent.setFormatValue(100);
               }
            }
        }
        super.packageData(packageDataEvent);
    }
```

```
}
```

17.4.5. verifyQuery 事件

在查询之前进行条件验证，可以根据自己的业务校验是否符合。

代码模板

Java

```java
package kd.report.demo.plugin;

import kd.bos.entity.report.ReportQueryParam;
import kd.bos.report.plugin.AbstractReportFormPlugin;

public class DemoReportFormPlugin extends AbstractReportFormPlugin{

    @Override
    public boolean verifyQuery(ReportQueryParam queryParam) {
        // TODO Auto-generated method stub
        return super.verifyQuery(queryParam);
    }
}
```

事件参数

示例

➢ 案例说明

-- 对报表进行查询前条件验证，如果当前用户是请假单申请人，则可以正常显示数据，如果非申请人，则提示，并且不显示报表数据。

➢ 实现方案

-- 在 verifyQuery 方法中对当前用户和请假单申请人进行对比，若符合怎么返回 true，否则返回 false，并提示

➢ 案例演示

当前用户是申请人：

当前用户非申请人：

➢ 实例代码

Java

```
package kd.report.demo.plugin;

import kd.bos.algo.DataSet;
import kd.bos.algo.Row;
import kd.bos.context.RequestContext;
import kd.bos.entity.report.ReportQueryParam;
import kd.bos.report.plugin.AbstractReportFormPlugin;
import kd.bos.servicehelper.QueryServiceHelper;

public class DemoReportFormPlugin extends AbstractReportFormPlugin{

    @Override
    public boolean verifyQuery(ReportQueryParam queryParam) {
        // TODO Auto-generated method stub
        String userId = RequestContext.get().getUserId();
        DataSet dataSet = QueryServiceHelper.queryDataSet(this.getClass().getName(),
"kdec_applybill", "billno as kdec_billno,kdec_applytype,kdec_applyday,kdec_totalday,creator as
kdec_creator", null, null);
        String applyUserId = "";
        for (Row row : dataSet) {
         long appUserId = (long) row.get("kdec_creator");
         applyUserId = String.valueOf(appUserId);
        }
        if(userId.equals(applyUserId)) {
            return true;
```

```
        }else {
            this.getView().showMessage("当前用户不是申请人！");
            return false;
        }
    }
}
```

17.4.6. beforeQuery 事件

查询取数之前触发，可以处理一些查询前置条件。

代码模板

Java

```
package kd.report.demo.plugin;

import kd.bos.entity.report.ReportQueryParam;
import kd.bos.report.plugin.AbstractReportFormPlugin;

public class DemoReportFormPlugin extends AbstractReportFormPlugin{

    @Override
    public void beforeQuery(ReportQueryParam queryParam) {
        // TODO Auto-generated method stub
        super.beforeQuery(queryParam);
    }
}
```

事件参数

示例

- 案例说明
  -- 对每张请假单的总天数加 10 天

- 实现方案
  -- 在 beforeQuery 对每张请假单总天数加 10 天，然后保存

- 案例演示
  总天数加之前：

总天数加之后：

➢ 实例代码

Java

```
package kd.report.demo.plugin;

import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.entity.report.ReportQueryParam;
import kd.bos.report.plugin.AbstractReportFormPlugin;
import kd.bos.servicehelper.BusinessDataServiceHelper;
import kd.bos.servicehelper.operation.SaveServiceHelper;

public class DemoReportFormPlugin extends AbstractReportFormPlugin{

    @Override
    public void beforeQuery(ReportQueryParam queryParam) {
        // TODO Auto-generated method stub
        DynamicObject[] dCollection = BusinessDataServiceHelper.load("kdec_applybill", "billno
as kdec_billno,kdec_applytype,kdec_applyday,kdec_totalday,creator as kdec_creator", null);
        DynamicObject[] dObjects = dCollection;
        for(int i = 0; i < dCollection.length; i++) {
            int totalDay = dCollection[i].getInt("kdec_totalday") + 10;
            dObjects[i].set("kdec_totalday", totalDay);
        }
        SaveServiceHelper.save(dObjects);
    }
}
```

17.4.7. afterQuery 事件

查询取数之后触发，可以在查询数据之后进行一些处理。

代码模板

Java

```java
package kd.report.demo.plugin;

import kd.bos.entity.report.ReportQueryParam;
import kd.bos.report.plugin.AbstractReportFormPlugin;

public class DemoReportFormPlugin extends AbstractReportFormPlugin{

    @Override
    public void afterQuery(ReportQueryParam queryParam) {
        // TODO Auto-generated method stub
        super.afterQuery(queryParam);
    }
}
```

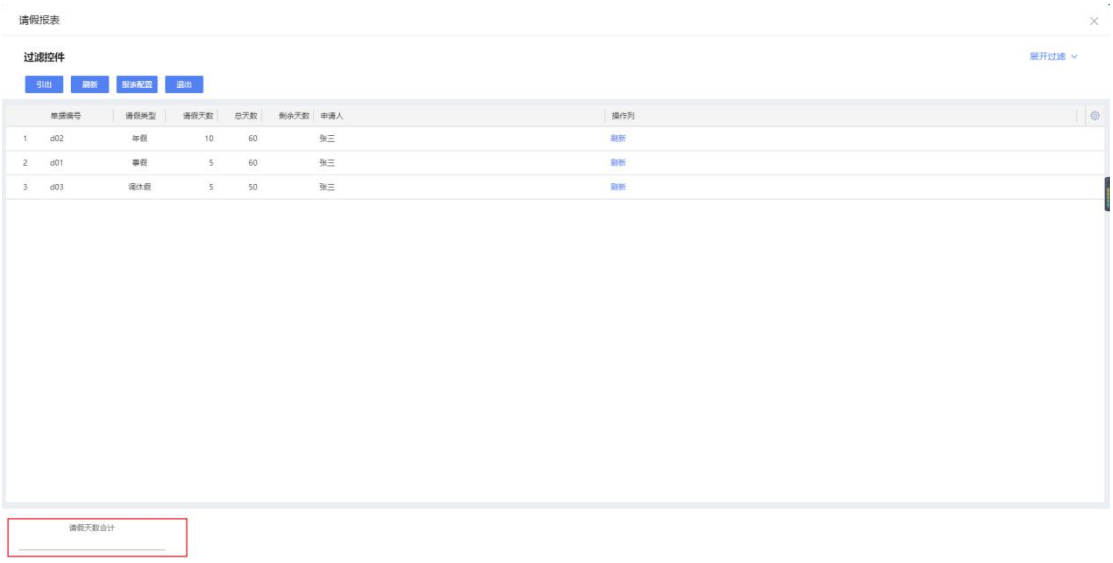
事件参数

示例

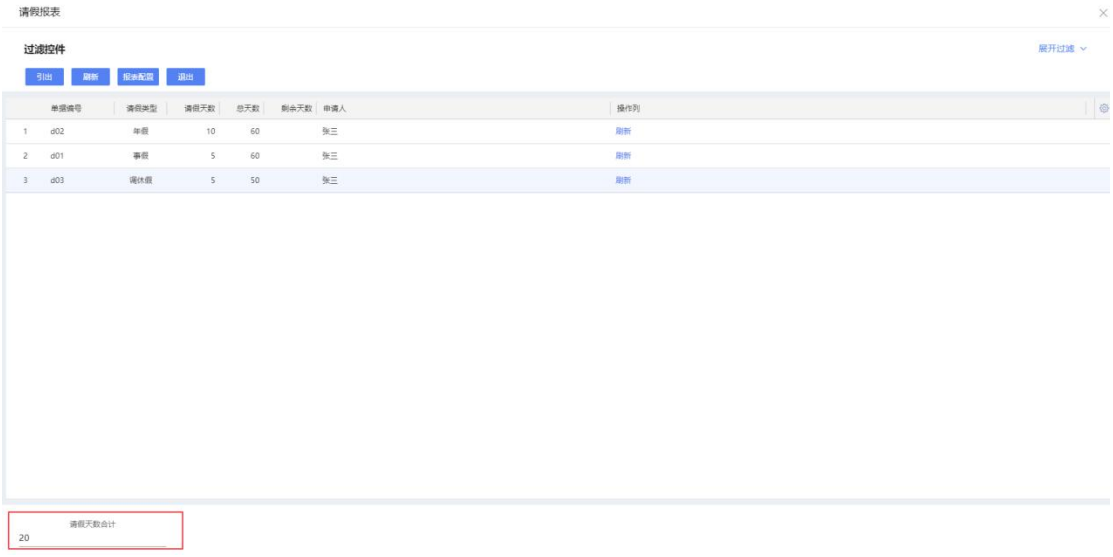
- 案例说明
  -- 在报表界面加一个请假天数合计字段，用来统计请假天数

- 实现方案
  -- 在查询之后 afterQuery 中，对请假天数合计。

- 案例演示
  请假天数合计之前：



请假天数合计之后：



➢ 实例代码

Java

```
package kd.report.demo.plugin;

import kd.bos.algo.DataSet;
import kd.bos.algo.Row;
import kd.bos.entity.report.ReportQueryParam;
import kd.bos.report.plugin.AbstractReportFormPlugin;
import kd.bos.servicehelper.QueryServiceHelper;

public class DemoReportFormPlugin extends AbstractReportFormPlugin{

    @Override
    public void afterQuery(ReportQueryParam queryParam) {
```

```
        // TODO Auto-generated method stub
        DataSet dataSet = QueryServiceHelper.queryDataSet(this.getClass().getName(),
"kdec_applybill", "billno as kdec_billno,kdec_applytype,kdec_applyday,kdec_totalday,creator as
kdec_creator", null, null);
        int sumApplyDay = 0;
        for (Row row : dataSet) {
            int applyday = (int) row.get("kdec_applyday");
            sumApplyDay = sumApplyDay + applyday;
        }
        this.getModel().setValue("kdec_sumapplyday", sumApplyDay);
        super.afterQuery(queryParam);
    }
}
```

17.4.8. initDefaultQueryParam 事件

初始化默认查询参数。

代码模板

Java

```
package kd.report.demo.plugin;

import kd.bos.entity.report.ReportQueryParam;
import kd.bos.report.plugin.AbstractReportFormPlugin;

public class DemoReportFormPlugin extends AbstractReportFormPlugin{

    @Override
    public void initDefaultQueryParam(ReportQueryParam queryParam) {
        // TODO Auto-generated method stub
        super.initDefaultQueryParam(queryParam);
    }
}
```

事件参数

示例

待更新

17.4.9. formatDisplayFilterField 事件

格式化主界面显示的筛选过滤字段信息。

代码模板

Java

```
package kd.report.demo.plugin;

import kd.bos.report.events.FormatShowFilterEvent;
import kd.bos.report.plugin.AbstractReportFormPlugin;

public class DemoReportFormPlugin extends AbstractReportFormPlugin{

    @Override
    public void formatDisplayFilterField(FormatShowFilterEvent evt) {
        // TODO Auto-generated method stub
        super.formatDisplayFilterField(evt);
    }
}
```

事件参数

示例

待更新

17.4.10. afterCreateColumn 事件

表格列创建完成后置事件。

代码模板

Java

```
package kd.report.demo.plugin;

import kd.bos.report.events.CreateColumnEvent;
import kd.bos.report.plugin.AbstractReportFormPlugin;
```

```
public class DemoReportFormPlugin extends AbstractReportFormPlugin{

    @Override
    public void afterCreateColumn(CreateColumnEvent event) {
        // TODO Auto-generated method stub
        super.afterCreateColumn(event);
    }
}
```

事件参数

示例

待更新

17.5. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问进行 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=17 反馈。

18. 工作流插件

18.1. 插件基类

18.1.1. 插件接口及基类

系统预置了工作流接口 IWorkflowPlugin，工作流插件需要实现该接口。

Java

```
package kd.bos.workflow.engine.extitf;
public interface IWorkflowPlugin extends ITaskPlugin {
```

18.1.2. 功能方法及使用

工作流接口提供了以下方法：

方法	说明

calcUserIds	根据特定的业务逻辑返回参与人 id 列表
hasTrueCondition	按自己的业务逻辑返回 true 或 false，返回 true 表示条件为真，返回 false 表示条件为假。
formatFlowRecord	用于单据内嵌的审批记录内容的格式化或者查看流程图右侧审批记录的格式化，如果提供的标准显示不符合要求，可以扩展此接口来修改审批记录的显示。
notify	节点进入，正向执行时调用 notify 方法
notifyByWithdraw	节点离开，撤回时调用 notifyByWithdraw 方法。

18.2. 插件事件

18.2.1. calcUserIds 事件

当用户创建好流程后，针对每个审批节点都要设置对应的参与人（审批人）。

代码模板

Java

```java
package kd.workflow.demo.plugin;

import java.util.List;

import kd.bos.workflow.api.AgentExecution;
import kd.bos.workflow.engine.extitf.IWorkflowPlugin;

public class WorkflowDemoPlugin implements IWorkflowPlugin{

    @Override
    public List<Long> calcUserIds(AgentExecution execution) {
        // TODO Auto-generated method stub
        return IWorkflowPlugin.super.calcUserIds(execution);
    }
}
```

示例

- 案例说明
  -- 用户提交单据后，需要指定人员审核，例如工号为 ID-000027 的人员审核

- 实现方案

  -- 在 calcUserIds 事件中，查询工号为 ID-000027 的人员的 id，并返回。

- 实例代码

Java

```
package kd.workflow.demo.plugin;

import java.util.ArrayList;
import java.util.List;

import kd.bos.workflow.api.AgentExecution;
import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.orm.query.QCP;
import kd.bos.orm.query.QFilter;
import kd.bos.servicehelper.QueryServiceHelper;
public class WorkflowDemoPlugin implements IWorkflowPlugin{

    @Override
    public List<Long> calcUserIds(AgentExecution execution) {
       List<Long> userIds = new ArrayList<Long>();
       QFilter filter = new QFilter("number", QCP.equals, "ID-000027");
       DynamicObject dObject = QueryServiceHelper.queryOne("bos_user", "id,name,number", new
QFilter[] {filter});
       Long userId = dObject.getLong("id");
       userIds.add(userId);
       return userIds ;//返回值为 List<Long> 类型的数据。返回内容为参与人的 id
    }
}
```

18.2.2. hasTrueCondition 事件

平台考虑到用户在使用流程中可能会设置条件，所以在对应位置开放该权限。设置条件的方法有两种：规则设置和外部接口。当规则设置不能满足用户要求时，用户可以使用外部接口，自己定义插件，完成对应的功能。

代码模板

Java

```
package kd.workflow.demo.plugin;

import kd.bos.workflow.api.AgentExecution;
import kd.bos.workflow.engine.extitf.IWorkflowPlugin;
```

```
public class WorkflowDemoPlugin implements IWorkflowPlugin{

    @Override
    public boolean hasTrueCondition(AgentExecution execution) {
        // TODO Auto-generated method stub
        return IWorkflowPlugin.super.hasTrueCondition(execution);
    }
}
```

示例

- 案例说明
  -- 用户提交请假申请，如果请假天数大于 3 天，则需要领导审批，反之则不需要

- 实现方案
  -- 在参与人那里设置条件，如果满足则指定分配给对应的参与人

- 实例代码

Java

```
package kd.workflow.demo.plugin;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import kd.bos.context.RequestContext;
import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.orm.query.QCP;
import kd.bos.orm.query.QFilter;
import kd.bos.servicehelper.BusinessDataServiceHelper;
import kd.bos.servicehelper.QueryServiceHelper;
import kd.bos.servicehelper.operation.SaveServiceHelper;
import kd.bos.servicehelper.user.UserServiceHelper;
import kd.bos.workflow.api.AgentExecution;
import kd.bos.workflow.api.WorkflowElement;
import kd.bos.workflow.component.approvalrecord.IApprovalRecordItem;
import kd.bos.workflow.engine.extitf.IWorkflowPlugin;

public class WorkflowDemoPlugin implements IWorkflowPlugin{

    @Override
```

```
    public boolean hasTrueCondition(AgentExecution execution) {
        String businessKey = execution.getBusinessKey();//单据的 BusinessKey(业务 ID)
        if(businessKey!=null) {
            DynamicObject dynamicObject = BusinessDataServiceHelper.loadSingle(businessKey,
"kdec_applybill");
            int applyday = dynamicObject.getInt("kdec_applyday");
            if(applyday>3) {
                return true;
            }else {
                return false;
            }
        }
        return false;
    }
}
```

18.2.3. formatFlowRecord 事件

用户查看审批详情时，想修改显示值可以通过插件来实现。可以在“节点记录格式化插件”中，放入自己的插件实现自己想要的逻辑。

代码模板

Java

```
package kd.workflow.demo.plugin;

import kd.bos.workflow.component.approvalrecord.IApprovalRecordItem;
import kd.bos.workflow.engine.extitf.IWorkflowPlugin;

public class WorkflowDemoPlugin implements IWorkflowPlugin{

    @Override
    public IApprovalRecordItem formatFlowRecord(IApprovalRecordItem item) {
        // TODO Auto-generated method stub
        return IWorkflowPlugin.super.formatFlowRecord(item);
    }
}
```

示例

- 案例说明

-- 用户查看审批详情时，希望能看到前流程节点和当前审批人文字说明

➢ 实现方案

-- 在 formatFlowRecord 获取当前流程节点和当前审批人，为其添加文字说明并进入 item 返回。

➢ 实例代码

Java

```java
package kd.workflow.demo.plugin;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import kd.bos.context.RequestContext;
import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.orm.query.QCP;
import kd.bos.orm.query.QFilter;
import kd.bos.servicehelper.BusinessDataServiceHelper;
import kd.bos.servicehelper.QueryServiceHelper;
import kd.bos.servicehelper.operation.SaveServiceHelper;
import kd.bos.servicehelper.user.UserServiceHelper;
import kd.bos.workflow.api.AgentExecution;
import kd.bos.workflow.api.WorkflowElement;
import kd.bos.workflow.component.approvalrecord.IApprovalRecordItem;
import kd.bos.workflow.engine.extitf.IWorkflowPlugin;

public class WorkflowDemoPlugin implements IWorkflowPlugin{
  @Override
    public IApprovalRecordItem formatFlowRecord(IApprovalRecordItem item) {
        String activityName = item.getActivityName();//获取当前流程节点名称
        String assignee = item.getAssignee();//获取审批人姓名
        item.setActivityName("当前流程节点：" + activityName);
        item.setAssignee("当前审批人：" + assignee);
        return item;
    }
}
```

18.2.4. notify 事件

用户创建流程时，针对每个节点不同时机可以有不同的操作。当平台提供的操作不能满足用户需要时，用户可以通过插件来实现自己需要的功能。正向执行时调用 notify 方法。

代码模板

Java

```
package kd.workflow.demo.plugin;

import kd.bos.workflow.api.AgentExecution;
import kd.bos.workflow.engine.extitf.IWorkflowPlugin;

public class WorkflowDemoPlugin implements IWorkflowPlugin{

    @Override
    public void notify(AgentExecution execution) {
        IWorkflowPlugin.super.notify(execution);
    }
}
```

示例

- 案例说明
  -- 用户提交请假申请，当请假被驳回时，当前请假单请假天数清零，剩余请假天数复原。

- 实现方案
  -- 在驳回时注册插件，在 notify 方法中，对请假单进行处理，并保存。

- 实例代码

Java

```
package kd.workflow.demo.plugin;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import kd.bos.context.RequestContext;
import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.orm.query.QCP;
import kd.bos.orm.query.QFilter;
import kd.bos.servicehelper.BusinessDataServiceHelper;
import kd.bos.servicehelper.QueryServiceHelper;
import kd.bos.servicehelper.operation.SaveServiceHelper;
import kd.bos.servicehelper.user.UserServiceHelper;
import kd.bos.workflow.api.AgentExecution;
import kd.bos.workflow.api.WorkflowElement;
import kd.bos.workflow.component.approvalrecord.IApprovalRecordItem;
import kd.bos.workflow.engine.extitf.IWorkflowPlugin;

public class WorkflowDemoPlugin implements IWorkflowPlugin{
    @Override
    public void notify(AgentExecution execution) {
        String businessKey = execution.getBusinessKey();//单据的 BusinessKey(业务 ID)
        DynamicObject dynamicObject = BusinessDataServiceHelper.loadSingle(businessKey,
"kdec_applybill");
        //请假天数
        int applyDay = dynamicObject.getInt("kdec_applyday");
        //剩余天数
        int residueDay = dynamicObject.getInt("kdec_residueday");
        //请假不通过，剩余天数复原
        applyDay = 0;
        residueDay = residueDay + applyDay;
        dynamicObject.set("kdec_applyday", applyDay);
        dynamicObject.set("kdec_residueday", residueDay);
        SaveServiceHelper.save(new DynamicObject[] {dynamicObject});
        IWorkflowPlugin.super.notify(execution);
    }
}
```

18.2.5. notifyByWithdraw 事件

节点离开，撤回时调用 notifyByWithdraw 方法。

代码模板

```Java
package kd.workflow.demo.plugin;

import kd.bos.workflow.api.AgentExecution;
import kd.bos.workflow.engine.extitf.IWorkflowPlugin;

public class WorkflowDemoPlugin implements IWorkflowPlugin{

    @Override
    public void notifyByWithdraw(AgentExecution execution) {
        IWorkflowPlugin.super.notifyByWithdraw(execution);
    }
}
```

示例

待更新

18.3. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=18 进行反馈。

19. 引入引出插件

用户打开引入起始页，准备引入参数（引入类型、匹配字段、数据文件），点击开始引入后，系统打开引入进度界面并启动两个线程分别执行“引入插件”的解析和保存方法，两个线程并发执行，通过缓存池共享单据数据。

19.1. 插件基类

19.1.1. 插件接口及基类

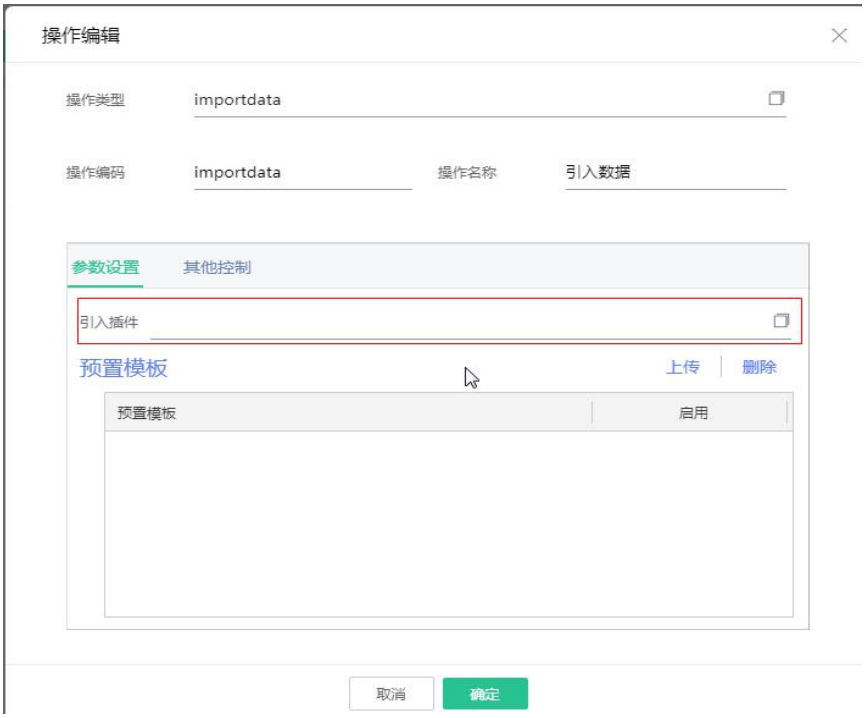
系统预置了引入插件基类 BatchImportPlugin，实现了接口 IImportDataPlugin：

```Java
```

```
package kd.bos.form.plugin.impt;
public class BatchImportPlugin implements Callable<Object>, IImportDataPlugin{
```

19.1.2. 创建并注册插件

首先，创建一个插件类，继承 kd.bos.form.plugin.impt.BatchImportPlugin 插件类；然后根据需要重写上面提到的方法（具体详情参看插件类的方法注释）；最后将引入插件挂载到引入操作的操作参数里面。



19.1.3. 功能方法及使用

引入插件基类会提供如下方法：

事件	说明
getBillFormId	获取待引入的实体 key
getOverrideFieldsConfig	覆盖引入的匹配字段备选项
getDefaultKeyFields	覆盖引入的匹配字段的缺省值
getDefaultImportType	缺省引入模式，新增、覆盖、覆盖并新增
getDefaultLockUIs	锁定的控件列表
save	缺省的保存逻辑
getBatchImportSize	引入的批量数据大小
isForceBatch	是否强制批处理
getSaveWebApi	构建 Web API 保存操作实例，用于转换 Json 并保存单
buildMainEntityType	根据数据动态构造实体类型

resolveExcel	解析 excel 数据后压入缓存队列
importData	从缓存队列分批取数，调用 Save()方法保存，如果有数据失败会生成错误数据文件
convertApiResult	帮助方法：将操作结果转成引入结果
getExportMainEntityType	根据表单标识和模板数据构造引出的实体类型 ，下载引出

19.2. 插件事件

19.2.1. save 事件

用户在引入 excel 的时候，可以根据需求在 save 事件中，重写保存逻辑，比如对数据进行筛选再保存等。

代码模板

Java

```java
package kd.batchimport.demo.plugin;

import java.util.List;

import kd.bos.entity.api.ApiResult;
import kd.bos.entity.plugin.ImportLogger;
import kd.bos.form.plugin.impt.BatchImportPlugin;
import kd.bos.form.plugin.impt.ImportBillData;

public class BathImportDemoPlugin extends BatchImportPlugin{

    @Override
    protected ApiResult save(List<ImportBillData> rowdatas, ImportLogger logger) {
        // TODO Auto-generated method stub
        return super.save(rowdatas, logger);
    }
}
```

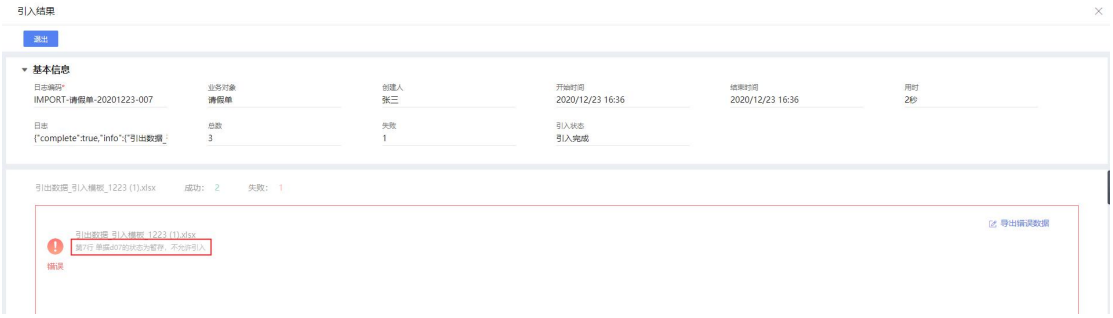
示例

- 案例说明
  -- 对引入的数据进行处理，只引入非暂存的数据，并给出引入失败提示

- 实现方案

  -- 在 save 方法中，对数据进行判断，对于非暂存的时候进行删除再保存

- 案例演示



- 实例代码

Java

```java
package kd.batchimport.demo.plugin;

import java.util.Iterator;
import java.util.List;
import java.util.Map;

import kd.bos.entity.api.ApiResult;
import kd.bos.entity.plugin.ImportLogger;
import kd.bos.form.plugin.impt.BatchImportPlugin;
import kd.bos.form.plugin.impt.ImportBillData;

public class BathImportDemoPlugin extends BatchImportPlugin{

    @Override
    protected ApiResult save(List<ImportBillData> rowdatas, ImportLogger logger) {
        Iterator<ImportBillData> it = rowdatas.iterator();
        while(it.hasNext()){
            ImportBillData data = it.next();
            Map<String , Object> billData = data.getData();
            String billstatus = (String) billData.get("billstatus");
            String billno = (String) billData.get("billno");
            if(billstatus.equals("A")) {
                // 返回校验信息
                String validMsg = "单据" + billno + "的状态为暂存，不允许引入";
                // 有校验提示，校验不通过，记录日志，移除数据
                logger.log(data.getStartIndex(), validMsg).fail();
                it.remove();
            }
        }
```

```
        // 调用缺省方法保存合法的数据
          return super.save(rowdatas, logger);
    }
}
```

19.3. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=19 进行反馈。

20. 开放 API 插件

20.1. 插件基类

20.1.1. 插件接口及基类

系统预置了开放 api 接口 IBillWebApiPlugin，api 插件需要实现该接口。

Java

```
package kd.bos.bill;
public interface IBillWebApiPlugin {
```

20.1.2. 创建并注册插件

开放 Api 插件必须实现 IBillWebApiPlugin 接口

Java

```
package kd.bos.bill;

import java.util.Map;
import kd.bos.bill.events.AICommandEvent;
import kd.bos.bill.events.ConvertPkEvent;
import kd.bos.entity.api.ApiResult;
import kd.bos.entity.api.WebApiContext;

public interface IBillWebApiPlugin {

    default ApiResult doCustomService(Map<String, Object> params) {
        return null;
```

```
    }
    default ApiResult doCustomService(WebApiContext ctx) {
        return null;
    }
}
```

20.2. 插件事件

开放 api 接口提供了以下方法：

方法	说明
setFormId	设置表单 Id
convertPk	通过 Api 参数自己实现 PK 值获取，而不是基于编码自动获得
doAICommand	执行 AICommand
doCustomService(Map<String, Object> params)	执行自定义服务方法
doCustomService(WebApiContext ctx)	执行自定义服务方法

20.2.1. doCustomService(Map<String, Object> params)事件

该方法可以接收第三方系统传递过来的参数，格式为 **Map<String, Object> params**。开发者可以自定义参数类型和格式传递过来，然后对参数进行解析操作。

代码模板

Java

```
package kd.bos.api.plugin;

import java.util.Map;
import kd.bos.bill.IBillWebApiPlugin;
import kd.bos.entity.api.ApiResult;

public class ApiDemo implements IBillWebApiPlugin {

    @Override
    public ApiResult doCustomService(Map<String, Object> params) {
        // TODO Auto-generated method stub
        return IBillWebApiPlugin.super.doCustomService(params);
    }
```

```
}
```

示例

➢ 案例说明

-- 第三方系统通过自定义服务实现人员新增。

➢ 实现方案

-- 通过传递进来的人员相关信息的参数，调用人员接口 UserServiceHelper 去实现人员新增

➢ 实例代码

Java

```java
package kd.bos.api.plugin;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

import kd.bos.bill.IBillWebApiPlugin;
import kd.bos.entity.api.ApiResult;
import kd.bos.permission.model.UserParam;
import kd.bos.servicehelper.user.UserServiceHelper;

public class AddUserApiPlugin implements IBillWebApiPlugin {

    @Override
    public ApiResult doCustomService(Map<String, Object> params) {
        // 获取参数，执行自定义服务逻辑......
        String phone = (String) params.get("phone");
        String name = (String) params.get("name");
        String companyname = (String) params.get("companyname");
        String uid = (String) params.get("uid");
        Map<String, Object> userdata = new HashMap<String, Object>();
        userdata.put("name", name);
        userdata.put("number", phone);
        userdata.put("usertype", 1);
        userdata.put("phone", phone);
        userdata.put("fuid", uid);
        ArrayList<LinkedHashMap<String, Object>> entryentity = new ArrayList<>();
```

```
        LinkedHashMap<String, Object> entryentitydata = new LinkedHashMap<>();
        entryentitydata.put("dpt", 100000);
        entryentitydata.put("position", companyname);
        entryentity.add(entryentitydata);
        userdata.put("entryentity", entryentity);
        // 创建一个人员
        List<UserParam> userList = new ArrayList<>();
        UserParam uParam = new UserParam();
        uParam.setDataMap(userdata);
        userList.add(uParam);
        // 新增人员
        UserServiceHelper.addOrUpdate(userList);
        String msg = uParam.getMsg();
        UserServiceHelper.enable(userList);
        ApiResult data = new ApiResult();
        // 如果人员创建成功
        if (uParam.isSuccess()) {
            data = ApiResult.success("人员新增成功");
            return data;
        } else {
            data = ApiResult.success(msg);
            return data;
        }
    }
}
```

20.3. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=20 进行反馈。

21. 后台任务插件

21.1. 插件基类

21.1.1. 插件接口及基类

系统预置了后台任务插件基类 AbstractTask，实现了接口 Task：

Java

```
package kd.bos.schedule.executor;
public abstract class AbstractTask implements Task {
```

接口 Task 的定义如下：

Java

```
package kd.bos.schedule.api;
public interface Task {
```

21.1.2. 创建并注册插件

后台任务插件需要继承 AbstractTask 类

Java

```
package kd.task.demo.plugin;
public class TaskDemoPlugin extends AbstractTask{
```

21.1.3. 功能方法及使用

后台任务基类会提供如下方法：

方法	说明
execute	执行任务
feedbackProgress	反馈进度，需要业务方法主动调用
feedbackCustomdata	反馈自定义数据，需要业务方法主动调用
stop	中止任务

21.2. 插件事件

21.2.1. execute 事件

代码模板

Java

```
package kd.task.demo.plugin;

import java.util.Map;

import kd.bos.context.RequestContext;
```

```
import kd.bos.exception.KDException;
import kd.bos.schedule.executor.AbstractTask;

public class TaskDemoPlugin extends AbstractTask{

    @Override
    public void execute(RequestContext arg0, Map<String, Object> arg1) throws KDException {
        // TODO Auto-generated method stub
    }
}
```

示例

- 案例说明
  -- 每隔一个月给用户增加一天年假

- 实现方案
  -- 配置后台任务，在 execute 事件中给用户增加一天年假。

- 实例代码

Java

```
package kd.task.demo.plugin;

import java.util.Map;

import kd.bos.context.RequestContext;
import kd.bos.dataentity.entity.DynamicObject;
import kd.bos.exception.KDException;
import kd.bos.orm.query.QCP;
import kd.bos.orm.query.QFilter;
import kd.bos.schedule.executor.AbstractTask;
import kd.bos.servicehelper.BusinessDataServiceHelper;
import kd.bos.servicehelper.operation.SaveServiceHelper;

public class TaskDemoPlugin extends AbstractTask{

    @Override
    public void execute(RequestContext arg0, Map<String, Object> arg1) throws KDException {
        QFilter qFilter = new QFilter("kdec_applytype", QCP.equals, "A");
        DynamicObject dObject = BusinessDataServiceHelper.loadSingle("kdec_applybill",
"kdec_billno,kdec_applytype,kdec_totalday", new QFilter[] {qFilter});
        int totalDay = dObject.getInt("kdec_totalday");
```

```
        totalDay = totalDay + 1;
        dObject.set("kdec_totalday", totalDay);
        SaveServiceHelper.save(new DynamicObject[] {dObject});
    }
}
```

21.3. 建议/问题反馈

有关本章节的任何问题，或建设性的意见，请通过在 PC 端访问 https://ecodevops.kingdee.com/index.html?userId=Guest&accountId=1078088639713379328&formId=kdec_quesfb_plugin&chapter=21 进行反馈。