|  |
| --- |
| ACME3.0测井采集控制管理软件基础平台  系统详细设计报告  digia_logo |
| 机密 |

| 版本 | 作者 – 日期 | 修订人 – 日期 | 批准人 – 日期 |
| --- | --- | --- | --- |
| 1.0 | 迪智软件(成都)有限公司–2012/11 |  |  |

| 当前状态 |  |
| --- | --- |
| 发布记录 |  |
| 储存位置 |  |

目录

[1 文档概述 7](#_Toc341951426)

[2 系统模块设计 7](#_Toc341951427)

[3 系统基础设计 7](#_Toc341951428)

[3.1 工程文件详细设计 7](#_Toc341951429)

[3.1.1 工程文件基础介绍 7](#_Toc341951430)

[3.1.2 工程文件内部结构 8](#_Toc341951431)

[3.2 微内核详细设计 9](#_Toc341951432)

[3.2.1 微内核基础介绍 9](#_Toc341951433)

[3.2.2 系统启动流程图 9](#_Toc341951434)

[3.2.3 设置最大线程数 10](#_Toc341951435)

[3.2.4 分析程序启动参数 11](#_Toc341951436)

[3.2.5 程序配置 11](#_Toc341951437)

[3.2.6 配置参数 11](#_Toc341951438)

[3.2.7 配置文件 12](#_Toc341951439)

[3.2.8 插件管理器 12](#_Toc341951440)

[3.3 数据处理层详细设计 14](#_Toc341951441)

[3.3.1 基本设计 14](#_Toc341951442)

[3.3.2 UML图 15](#_Toc341951443)

[基础数据结构 15](#_Toc341951444)

[上层数据接口层 16](#_Toc341951445)

[数据访问/缓存层 16](#_Toc341951446)

[3.3.3 设计模式 17](#_Toc341951447)

[3.3.4 数据缓存 17](#_Toc341951448)

[3.3.5 数据层工作流程 18](#_Toc341951449)

[数据读取流程 18](#_Toc341951450)

[数据访问者注册/反注册流程 19](#_Toc341951451)

[数据修改流程 20](#_Toc341951452)

[4 插件系统(动态模块体系)详细设计 20](#_Toc341951453)

[4.1 扩展点基础介绍 21](#_Toc341951454)

[4.2 扩展点服务概述 21](#_Toc341951455)

[4.3 搜索解析 24](#_Toc341951456)

[4.4 数据存储 25](#_Toc341951457)

[4.4.1 插件信息表 25](#_Toc341951458)

[4.4.2 扩展点定义信息表 25](#_Toc341951459)

[4.4.3 扩展点功能实现信息表 26](#_Toc341951460)

[4.5 数据层访问接口 26](#_Toc341951461)

[4.6 扩展点管理器 28](#_Toc341951462)

[5 集成界面 28](#_Toc341951463)

[5.1 主窗口布局设计 28](#_Toc341951464)

[5.2 主界面区域功能 29](#_Toc341951465)

[5.2.1 菜单栏区域功能 29](#_Toc341951466)

[5.2.2 工具栏区域功能 29](#_Toc341951467)

[5.2.3 停泊框区域功能 29](#_Toc341951468)

[5.2.4 状态栏区域功能 30](#_Toc341951469)

[5.2.5 中心窗口区域功能 30](#_Toc341951470)

[5.3 子窗口管理功能 30](#_Toc341951471)

[5.3.1 停泊框管理 30](#_Toc341951472)

[5.3.2 快捷键管理 30](#_Toc341951473)

[5.3.3 窗口拖拽 30](#_Toc341951474)

[5.4 扩展点接口定义 31](#_Toc341951475)

[5.4.1 主窗口菜单Id定义 31](#_Toc341951476)

[5.4.2 多语言定义 31](#_Toc341951477)

[5.4.3 菜单扩展点接口属性定义 31](#_Toc341951478)

[5.4.4 工具栏扩展点接口属性定义 32](#_Toc341951479)

[5.5 XML文件样例 32](#_Toc341951480)

[5.6 主界面实现原理 35](#_Toc341951481)

[5.6.1 主界面工作流程 35](#_Toc341951482)

[5.6.2 初始化窗口UI 35](#_Toc341951483)

[5.6.3 界面消息循环 36](#_Toc341951484)

[5.6.4 扩展点功能处理 36](#_Toc341951485)

[5.6.5 子窗口管理 36](#_Toc341951486)

[5.6.6 解析扩展点插件推送消息，添加功能点 36](#_Toc341951487)

[5.7 窗口分割实现原理 36](#_Toc341951488)

[5.7.1 基本原理 36](#_Toc341951489)

[5.7.2 事件处理 36](#_Toc341951490)

[5.8 扩展点菜单实现原理 37](#_Toc341951491)

[5.8.1 菜单实现原理 37](#_Toc341951492)

[菜单管理 37](#_Toc341951493)

[5.9 UML图 38](#_Toc341951494)

[5.9.1 XML解析器类图 38](#_Toc341951495)

[5.9.2 MainWindow类图 39](#_Toc341951496)

[5.9.3 CentralWidget类图 40](#_Toc341951497)

[5.9.4 快捷键管理对话框类图 40](#_Toc341951498)

[5.10 Undo/Redo框架设计 41](#_Toc341951499)

[5.10.1 功能描述 41](#_Toc341951500)

[5.10.2 典型功能用例 41](#_Toc341951501)

[5.10.3 性能描述 41](#_Toc341951502)

[5.10.4 输入项目 41](#_Toc341951503)

[5.10.5 输出项目 41](#_Toc341951504)

[5.10.6 系统架构 41](#_Toc341951505)

[5.10.7 流程逻辑 43](#_Toc341951506)

[5.11 属性框 44](#_Toc341951507)

[5.11.1 概述 44](#_Toc341951508)

[5.11.2 属性类型 44](#_Toc341951509)

[5.11.3 事件结构 47](#_Toc341951510)

[5.11.4 示例 48](#_Toc341951511)

[其它插件发送需要显示的属性信息给属性框插件 48](#_Toc341951512)

[其它插件接收来自属性框插件的属性变更信息 49](#_Toc341951513)

[6 图形绘制与交互 49](#_Toc341951514)

[6.1 概述 49](#_Toc341951515)

[6.2 画布 50](#_Toc341951516)

[6.2.1 基础逻辑和流程 50](#_Toc341951517)

[6.2.2 UML图 51](#_Toc341951518)

[6.2.3 接口设计 51](#_Toc341951519)

[6.3 图件 51](#_Toc341951520)

[6.3.1 道 52](#_Toc341951521)

[定义 52](#_Toc341951522)

[属性 52](#_Toc341951523)

[6.3.2 曲线 52](#_Toc341951524)

[定义 52](#_Toc341951525)

[属性 52](#_Toc341951526)

[6.3.3 UML图 53](#_Toc341951527)

[6.4 拖拽 53](#_Toc341951528)

[6.4.1 需求 53](#_Toc341951529)

[6.4.2 基础设计 54](#_Toc341951530)

[Qt拖拽技术概述 54](#_Toc341951531)

[拖拽的信息传递 54](#_Toc341951532)

[拖拽的流程及设计 55](#_Toc341951533)

[6.5 曲线填充 57](#_Toc341951534)

[6.5.1 需求 57](#_Toc341951535)

[6.5.2 基础设计 57](#_Toc341951536)

[两条曲线之间的填充算法： 57](#_Toc341951537)

[一条曲线和一条基线之间的填充： 58](#_Toc341951538)

[6.6 波列曲线 59](#_Toc341951539)

[6.6.1 需求 59](#_Toc341951540)

[6.6.2 基础设计 59](#_Toc341951541)

[绘制区域坐标系转换 59](#_Toc341951542)

[子坐标系绘制方式 60](#_Toc341951543)

[曲线绘制和填充 60](#_Toc341951544)

[渐变色绘制 60](#_Toc341951545)

[6.7 实时显示 61](#_Toc341951546)

[6.7.1 需求说明 61](#_Toc341951547)

[6.7.2 典型功能用例 61](#_Toc341951548)

[6.7.3 性能描述 62](#_Toc341951549)

[6.7.4 输入项 62](#_Toc341951550)

[6.7.5 输出项 62](#_Toc341951551)

[6.7.6 系统架构 62](#_Toc341951552)

[6.7.7 UML图 63](#_Toc341951553)

[6.7.8 逻辑流程 63](#_Toc341951554)

[6.7.9 数据输入输出模块设计 64](#_Toc341951555)

[6.7.10 通讯子模块的设计 66](#_Toc341951556)

[6.7.11 系统实时性设计与实现 68](#_Toc341951557)

[数据驱动 68](#_Toc341951558)

[显示驱动 68](#_Toc341951559)

[多点同步驱动 69](#_Toc341951560)

[追及方案 69](#_Toc341951561)

[数据传输中的追及方案 70](#_Toc341951562)

[显示与数据传输的追及 70](#_Toc341951563)

[6.7.12 接口 71](#_Toc341951564)

[6.7.13 限制条件 73](#_Toc341951565)

[7 打印 73](#_Toc341951566)

[7.1 打印 73](#_Toc341951567)

[7.1.1 功能描述 73](#_Toc341951568)

[7.1.2 典型功能用例 74](#_Toc341951569)

[7.1.3 输入项 74](#_Toc341951570)

[7.1.4 输出项 74](#_Toc341951571)

[7.1.5 系统架构 74](#_Toc341951572)

[7.1.6 接口 75](#_Toc341951573)

[7.2 实时打印 76](#_Toc341951574)

[7.2.1 功能描述 76](#_Toc341951575)

[7.2.2 性能描述 76](#_Toc341951576)

[7.2.3 输入项 76](#_Toc341951577)

[7.2.4 输出项 76](#_Toc341951578)

[7.2.5 系统架构 77](#_Toc341951579)

[7.2.6 流程逻辑 78](#_Toc341951580)

[7.2.7 关键技术设计 78](#_Toc341951581)

[7.2.8 接口 79](#_Toc341951582)

[7.2.9 限制条件 80](#_Toc341951583)

[8 二次开发 80](#_Toc341951584)

[8.1 开发向导 80](#_Toc341951585)

[8.1.1 功能描述 80](#_Toc341951586)

[8.1.2 界面UI 81](#_Toc341951587)

[程序主界面如下图所示： 81](#_Toc341951588)

[扩展点实现界面： 82](#_Toc341951589)

[8.1.3 典型功能用例 83](#_Toc341951590)

[8.1.4 输入项 83](#_Toc341951591)

[8.1.5 输出项 83](#_Toc341951592)

[8.1.6 基础架构 83](#_Toc341951593)

[8.1.7 扩展点定义功能 84](#_Toc341951594)

[8.1.8 接口 85](#_Toc341951595)

[8.2 插件授权管理工具 87](#_Toc341951596)

[8.2.1 概述 87](#_Toc341951597)

[8.2.2 基础设计 87](#_Toc341951598)

[8.2.3 界面UI 88](#_Toc341951599)

# 文档概述

本文档适用于有一定软件开发经验的工程师或者系统维护人员，用于了解系统的详细设计思路和实现方法，从而指导代码开发和维护工作.

# 系统模块设计

系统详细设计，主要分为以下几个模块

* 系统基础设计
* 插件系统（动态模块体系）设计
* 集成主界面
* 图形绘制与交互
* 图表编辑与显示
* 打印
* 二次开发

# 系统基础设计

本章节主要介绍了系统的基础设计，包括工程与工程文件的设计，系统内核的设计以及系统中数据层的基本设计思路以及其实现原理.

## 工程文件详细设计

### 工程文件基础介绍

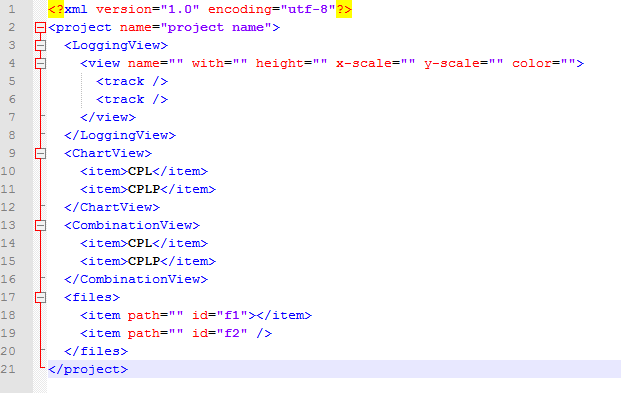
工程文件是ACME 3.0实时绘图显示与交互平台的核心文件，它能帮助用户根据自身需要，对各种数据和视图进行便捷的操作，管理和归档，方便日后进行查阅和修正.

工程文件的主要作用有：

1. 记录工程使用的测井视图模板信息。
2. 记录当前工程使用的图表视图模板信息。
3. 记录当前工程使用的组合视图模板信息。
4. 记录当前模板包含的数据文件（\*.ldf/\*.ldfx）。

工程文件后缀名为prjx，其内部采用xml格式存储工程相关信息.

### 工程文件内部结构



上图所示是一个工程文件所包含的基本内容，图中各个标签具体意义如下：

**Project**

标签代表一个工程，其中name表示工程名字.

**LoggingView** 表示测井视图，测井视图中的每个子项对应着一个预先定义好的测井视图模板文件，这些模板文件都放在程序安装目录下的template/LoggingView文件夹中.

**ChartView** 表示图表视图，图表视图中的每个子项对应着一个预先定义好的图表视图模板文件.这些模板文件都放在程序安装目录下的template/ChartView文件夹中.

**CombinationView** 表示组合视图，组合视图中的每个子项对应着一个预先定义好的组合视图模板文件，这些文件都放在程序安装目录下的template/CombinationView文件件中.

**Files** 表示工程包含的数据文件，files的每个子项都对应着一个测井数据文件（ldf/ldfx）在磁盘上的的绝对路径，以及该文件在工程中的id.

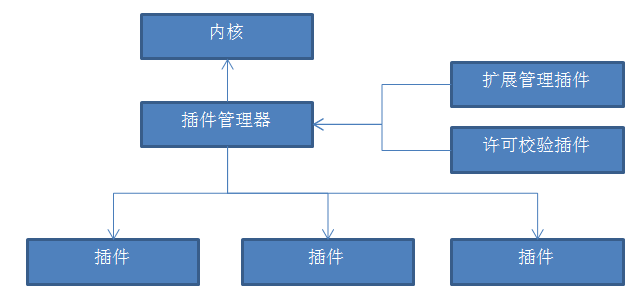
## 微内核详细设计

### 微内核基础介绍

微内核是实现工程插件化的基础，它主要负责启动插件管理器.

而插件管理器启动后，会自动进行搜索、加载、启动/关闭插件.

插件管理器会优先启动许可校验插件和扩展管理插件.其框架结构示意图如下：



图框架结构示意图

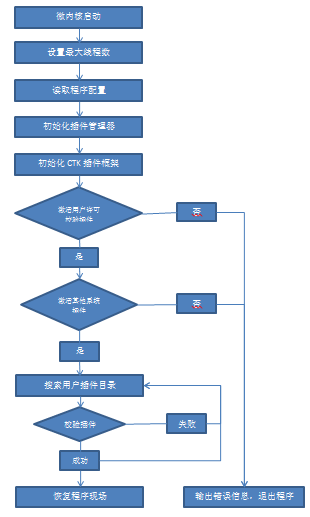
**许可校验插件：**

校验程序中存在的插件是否是用户许可允许的插件.

**扩展管理插件：**

搜索插件的扩展点配置文件（plug.xml），并且解析出扩展点和扩展数据.等插件启动时需要向有扩展点的插件推送其他插件实现的扩展点数据。

### 系统启动流程图



### 设置最大线程数

在微内核中，系统通过Qt的全局线程池来得到CPU最大支持的线程数目.

### 分析程序启动参数

程序启动时，用户也可以输入启动参数.通过这些参数可以显示程序的基本信息和设置程序的基本的运行模式、参数等等.

|  |  |
| --- | --- |
| **参数** | **含义** |
| -help | 显示程序所有的启动参数和帮助说明 |
| -version | 显示主程序的版本号 |
|  |  |
|  |  |
|  |  |

### 程序配置

程序的全局配置需要保存在应用程序路径下面.通过QCoreApplication::applicationDirPath() 得到程序当前的运行目录.

### 配置参数

|  |  |
| --- | --- |
| **配置** | **参数** |
| General/PluginPath | 插件目录的相对主程序的路径 |
| SystemPlugin/LicenceCheckingPlugin | 指定用户许可校验插件的路径 |
| SystemPlugin/ExtensionManagerPlugin | 指定扩展点管理插件的路径 |
| SystemPlugin/EventBusPlugin | 指定事件管理插件的路径 |
| SystemPlugin/ConfigurationManagerPlugin | 指定配置管理插件的路径 |
| SystemPlugin/LogManagerPlugin | 指定日志管理插件的路径 |

### 配置文件

配置文件为ini格式，文本编码格式为UTF-8.存放于主程序运行目录，与主程序文件名保持一致.

配置文件示例：

[General]

PluginPath=。。/Plugins

[SystemPlugin]

LicenceCheckingPlugin=。。/Service/org。acme。servce。licenceChecking。dll

ExtensionManagerPlugin=。。/Service/org。acme。service。extensionManager。dll

EventBusPlugin=。。/Service/org。acme。service。eventbus。dll

ConfigurationManagerPlugin=。。/Service/org。acme。service。configurationManager。dll

LogManagerPlugin=。。/Service/org。acme。service。logManager。dll

### 插件管理器

插件管理器负责查找、加载（卸载）、激活（反激活）系统插件和用户插件.

插件管理器会根据配置文件（1.3章节）获取系统插件路径和用户插件的目录，然后首先加载、激活用户许可校验插件，然后加载其他系统插件，并进行用户许可校验，校验通过后激活该插件.

加载完所有系统插件后开始从配置文件中获得用户插件的路径，然后搜索所有用户插件，加载并校验用户插件，通过许可校验的插件会调用扩展点管理插件读取并解析该插件的扩展点定义和扩展点数据.

所有的插件加载完成后，插件管理器会调用配置管理服务读取用户设置恢复程序现场（用户上次退出程序时的程序的状态）.

插件被激活时，插件管理器会根据插件间的依赖关系，顺序激活依赖的插件.扩展点管理服务会搜索与此插件相关的扩展点数据，并推送给此插件.

插件被激活后，会通过调用配置管理服务来读取其现场设置，并恢复上次用户状态.



图：插件管理器恢复现场流程



图：插件管理器类图

## 数据处理层详细设计

数据层的设计，基于常规软件MVC的设计思路，让数据的处理与应用层与逻辑层进行分离，便于维护扩展与二次开发.

本系统的数据处理层，主要针对用户对图件图表等业务数据处理操作.

### 基本设计

数据处理中心主要分为三层：底层数据访问层、数据缓存层和上层数据接口层.

底层数据访问接口主要针对数据文件和网络数据进行封装的访问接口的封装层.

数据缓存层主要是对数据修改后对修改的数据进行缓存处理，对数据访问者进行修改后的数据同步.

上层数据接口层主要是用于上层数据访问者向底层请求数据处理、获取数据更新通知等.

数据文件

网络数据

数据文件访问接口

网络数据访问接口

数据缓存服务

数据处理服务

上层数据访问接口层

### UML图

#### 基础数据结构



#### 上层数据接口层



#### 数据访问/缓存层



### 设计模式

上层数据访问服务采用观察者模式.

使用数据接口层访问所关注的数据后，数据服务会把此访问者加入订阅列表，当访问者所关注的数据发生变化时，数据服务会直接推送数据到数据访问者.

数据访问者

数据访问中间层

注册

数据更新

每个需要数据并且需要数据更新获得主动通知的地方，都需要通过数据访问中间层创建一个数据访问者.创建访问者的同时，会自动注册到数据访问中间层，当数据访问者访问的数据发生变化时，数据访问中间层会主动推送更新的数据给访问者.

### 数据缓存

当数据访问者修改数据后，并不会立刻写入数据文件，因此需要把数据保存在内存中.并通知其他的数据访问者更新，重新读取数据.

数据缓存管理

数据缓存

数据缓存

数据访问管理

底层数据接口

外部通过数据访问管理读取数据时，数据访问管理会先去查找有无数据缓存，如果有数据缓存则读取相应的数据并返回.如果没有数据缓存则直接读取底层数据并返回.

外部调用保存接口时，数据访问管理会查找出相应的数据缓存，并通过底层数据接口写入文件.

### 数据层工作流程

#### 数据读取流程



#### 数据访问者注册/反注册流程



#### 数据修改流程

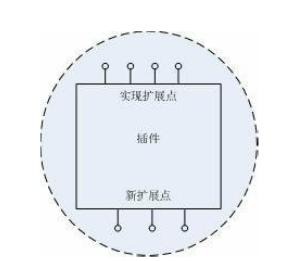


# 插件系统(动态模块体系)详细设计

为了便于项目的维护和二次开发，根据客户需求，系统采用了动态模块及插件化工程的设计思路，基于扩展点进行系统的整体框架设计.

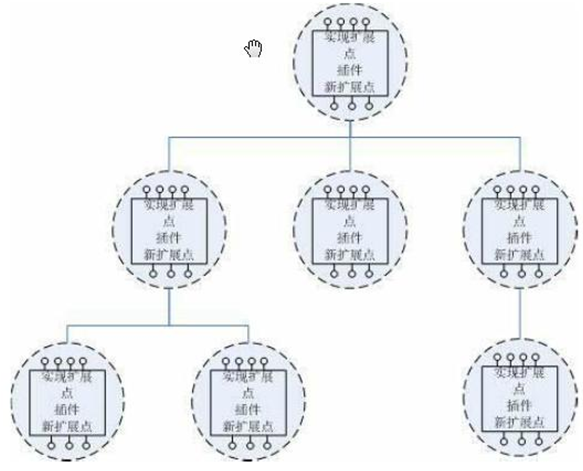
## 扩展点基础介绍

所谓扩展点，就是扩展点是一个端口的定义，即为其他插件提供服务的入口



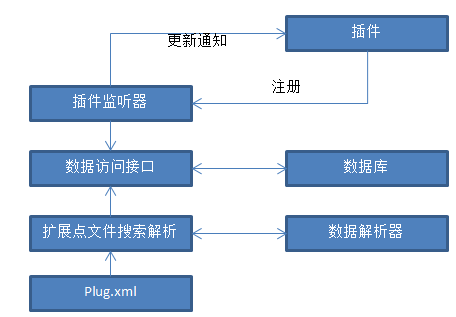
在单个插件的基础上，插件之间不断进行扩展，就会形成如下的插件网络.

宿主插件定义扩展点，并且暴露出来，新插件根据扩展点的规范创建和扩展，即实现扩展点规范，接着插件将其注册到一个特殊的插件注册表中，这样一来宿主插件就可以通过这个特殊的插件注册表找到它的扩展实现.



## 扩展点服务概述

扩展点服务主要提供了插件扩展点功能定义的搜索、扩展点依赖关系的管理和扩展点定义的推送等功能.



插件管理器在搜索到plug.xml后会从数据库查询是否存在该插件的扩展点数据文件，如果存在会查询出其MD5值与现在的文件的MD5值进行比对，如果MD5值不一致说明配置文件已经被修改，此时需要清除之前数据库中的数据，重新解析该配置文件，并保存数据到数据库中.

扩展点数据操作流程：



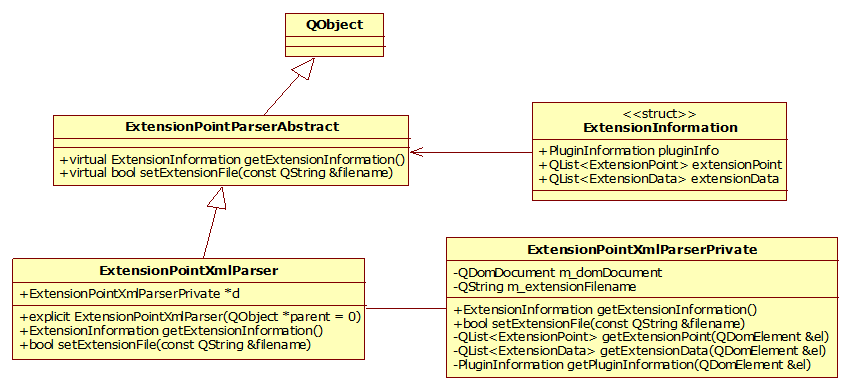


## 搜索解析

每个plug.xml 文件都放在插件目录下面，在搜索插件的时候首先搜索plug.xml 文件.

该文件需要解析出Plugin、ExtensionPointSet和ExtensionDataSet三部分.Plugin和ExtensionPointSet需要解析出详细的标签.ExtensionDataSet只需要获取ExtensionData标签下的所有文本作为数据，并以id作为索引即可.如果ExtensionData标签中包含有需要翻译的多语言字符串，则需要添加TranslationDataSet标签.

解析出数据后通过数据层进行保存.



## 数据存储

系统的数据存储使用SQLite数据库保存扩展点定义和实现的数据及对应关系.

数据库大概需要三张表分别来保存插件信息、扩展点定义信息和扩展点实现信息，它们是：

* 插件信息表
* 扩展点定义信息表
* 扩展点功能实现信息表

### 插件信息表

PluginInfoTable

|  |  |  |
| --- | --- | --- |
| **字段名** | **类型** | **备注** |
| name | String | 插件名字，主索引 |
| filename | string | 插件文件名 |
| modificationCheck | string | plug.xml 的MD5串，用来判断文件是否被修改 |

### 扩展点定义信息表

ExtensionPointTable

|  |  |  |
| --- | --- | --- |
| **字段名** | **类型** | **备注** |
| id | int | 自增长，主索引 |
| epid | string | 扩展点定义ID |
| plugin | string | 插件的名字 |
| label | string | 扩展点名字 |
| description | string | 扩展点描述 |

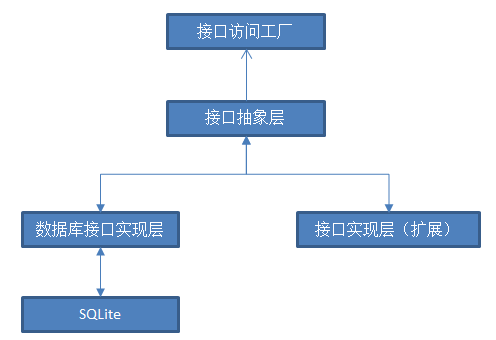
### 扩展点功能实现信息表

ExtensionPointDataTable

|  |  |  |
| --- | --- | --- |
| **字段名** | **类型** | **备注** |
| id | int | 自增长，主索引 |
| epid | string | 实现的扩展点ID |
| plugin | string | 插件名字 |
| epPlugin | string | 被扩展的插件 |
| extensionData | string | 扩展点数据 |

## 数据层访问接口

数据访问接口分为三层，接口访问工厂、接口抽象层和接口实现层.



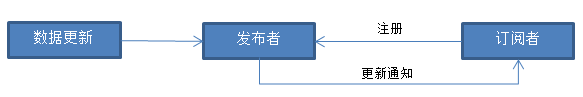
图：数据访问层接口框架图



图：数据访问层类图

## 扩展点管理器

插件启动时，需要向扩展点服务注册，从而扩展点数据更新时，扩展点服务会自动推送数据给插件.



图：扩展点服务观察者模式示意图



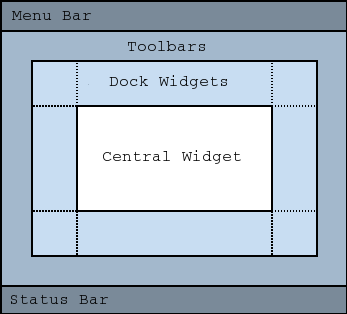
图：扩展点服务观察者模式类图

插件如果含有扩展点需要其他插件实现扩展点功能，即插件需要扩展点管理插件推送扩展点数据给此插件，则此插件需要实现PluginExtensionListener这个监听类，以便向扩展点管理插件注册并及时获取扩展点的数据.

# 集成界面

## 主窗口布局设计

主窗口布局采用QMainWindow提供的布局方式，如下图所示.



## 主界面区域功能

### 菜单栏区域功能

插件根据需要向这些菜单项中加入子菜单或操作项，也可以根据需要在菜单栏中添加新的菜单项.

插件之间的菜单项归属关系由插件自己管理.

菜单栏默认提供”File”， “Edit”， “Process”， “System”， “Windows”， ”Help”， 6个菜单项，菜单项通过主界面的扩展点方式加入，写入代码.

### 工具栏区域功能

工具栏默认没有提供任何子工具栏供插件加入操作项.操作项与子工具栏的归属关系由提供操作项的插件自己管理，主窗口插件不参与工具栏与操作项的归属管理.

### 停泊框区域功能

插件若提供停泊框相关功能，提供的停泊框必须继承自QWidget，否则无法加入到停泊框区域.

停泊框区域的窗口为唯一窗口，即由插件管理窗口的内存释放，并且插件在全局范围内只能创建一个该窗口，mainwindow主要控制该窗口的显示/隐藏.

### 状态栏区域功能

状态栏可以捕获CtkEventBus上订阅的信息，插件可以通过CtkEventBus向状态栏发送文字信息.

插件也可以向状态栏加入一些功能性窗口，窗口必须继承自QWidget.

状态栏区域的窗口为唯一窗口，即由插件管理窗口的内存释放，并且插件在全局范围内只能创建一个该窗口，mainwindow主要控制该窗口的显示/隐藏.

### 中心窗口区域功能

中心窗口由多个选项卡窗口组成，插件可以向选项卡中添加功能窗口.

选项卡窗口可以通过拖拽选项卡的方式分割成多个组合的选项卡窗口.

中心窗口为可以多次添加窗口，插件响应操作后产生新的窗口，并将窗口传给mainwindow，该窗口的父对象即为mainwindow，该窗口关闭后内存清理由中心窗口管理，插件不需要再管理.

## 子窗口管理功能

### 停泊框管理

“Windows”菜单项包含”停泊框管理(停泊框显示/隐藏)”，该项提供停泊框的显示/隐藏设置.

### 快捷键管理

“Windows”菜单项包含”快捷键管理”，该项提供菜单与工具栏操作项的快捷键设置与管理.

### 窗口拖拽

工具栏，停泊框的拖拽方式与QMainWindow提供的方式相同，以下主要定义中心窗口的选项卡拖拽操作.

拖拽中心窗口上的选项卡，可以将选项卡中的窗口分离出来单独作为窗口放置，也可以拖拽将多个窗口合并在同一个选项卡窗口中放置.

中心窗口根据当前中心窗口的布局，分为垂直方向分割与水平方向分割.

垂直方向上分割，分割的窗口可以添加到中心窗口的上下端.

水平方向上分割，分割的窗口可以添加到中心窗口的左右端.

## 扩展点接口定义

主界面扩展点包括”菜单栏扩展”，”工具栏扩展”，”功能点扩展”.

### 主窗口菜单Id定义

* File: org.acme.mainwindow.desktop.filemenu
* Control: org.acme.mainwindow.desktop.controlmenu
* Drawing: org.acme.mainwindow.desktop.paintmenu
* Data: org.acme.mainwindow.desktop.datamenu
* Processing: org.acme.mainwindow.desktop.processmenu
* Windows: org.acme.mainwindow.desktop.windowsmenu
* Help: org.acme.mainwindow.desktop.helpmenu

### 多语言定义

扩展点有字符内容的数据无法使用Qt自带的多国语言方案解决，因为获取插件的扩展点信息与插件启动没有直接关系。

所以我们在xml扩展点文件中加入多国语言的信息，在插件解析时，获取当前位置信息，并解析出xml中当前位置的扩展点信息。

### 菜单扩展点接口属性定义

* Id:菜单Id
* Label:菜单名（多语言）
* ParentId:父菜单Id，为空则直接加入菜单栏
* Index:菜单插入顺序索引（索引由0开始）
* Icon:菜单图标路径
* parentGroup:菜单父动作组名
* groupList:菜单子动作组链表

### 工具栏扩展点接口属性定义

* Id:工具栏Id
* Label:工具栏标题（多语言）
* Orientation:工具栏默认区域（Top，Left，Right，Bottom）
* groupList:工具条子动作组链表
* 功能点扩展接口属性定义
* Id:功能点Id
* Label:功能点名（多语言）
* Icon:功能点图标路径
* Shortcut:功能点动作使用的动作快捷键
* toolTips:功能点动作使用的提醒文字信息（多语言）
* Checkable:功能扩展点是否为on/off开关框
* serviceClass:提供服务接口类名
* eventbustopic:服务订阅事件标题
* command:动作事件命令
* defaultChecked:功能扩展点默认开关值
* defaultEnable:功能扩展点默认是否可用
* parentMenu:父菜单Id
* menuGroup:所属父菜单动作组名
* parentToolBar:父工具条Id
* toolBarGroup:所属父工具条动作组名
* menuGroupIndex:功能点在菜单动作群组中的索引值
* toolBarGroupIndex:功能点在工具条动作群组中的索引值

## XML文件样例

<ExtensionData plugin="plugin name" extendPlugin="extend plugin name" id="org.acme.MainWindow.Desktop.ACTION">

<MenuSet>

<Menu id="org.acme.MainWindow.Desktop.FileMenu"

label="MENUITEM\_FILEMENU"

icon="images/icon.png"

parentId=""

parentGroup="group1"

parentIndex="0">

<Group id="group1" />

<Group id="group2" />

</Menu>

</MenuSet>

<ToolbarSet>

<Toolbar id="org.acme.MainWindow.Desktop.MainToolbar"

label="MAINWINDOW\_MAINTOOLBAR"

position="top"

>

<Group id="group1" />

</Toolbar>

</ToolbarSet>

<ActionSet>

<Action id="a"

label="ACTIONITEM\_XXX"

icon="images/icon.png"

checkable="false"

defalutChecked="false"

defaultEnable="true"

shortCuts="Ctrl+F"

toolTips="File Menu"

parentMenu="menubarID"

menuGroup="group1"

menuGroupIndex="1"

parentToolbar="toolbarId"

toolbarGroup="group1"

toolbarGroupIndex="1"

serviceClass="MouseDrawService"

eventbustopic=””

command=""

/>

</ActionSet>

<DockSet>

<Dock

id=”DockID”

title=”Log View”

icon="images/icon.png"

shortCuts="Ctrl+F"

toolTips="File Menu"

parentMenu="menubarID"

menuGroup="group1"

menuGroupIndex="1"

parentToolbar="toolbarId"

toolbarGroup="group1"

toolbarGroupIndex="1"

serviceClass="MouseDrawService"

eventbustopic=””

command=""

activationPolicy=”lazy” //eager， lazy

/>

</DockSet>

<TranslationDataSet>

<TranslationData location="en\_US">

<Translation source="MENUITEM\_FILEMENU">File</Translation>

<Translation source="MENUITEM\_CONTROLMENU">Control</Translation>

</TranslationData>

<TranslationData location="zh\_CN">

<Translation source="MENUITEM\_FILEMENU">(F)</Translation>

<Translation source="MENUITEM\_CONTROLMENU">(C)</Translation>

</TranslationData>

</TranslationDataSet>

</ExtensionData>

在XML中有五种预定义非法字符需要替换为实体引用：

|  |  |  |
| --- | --- | --- |
| 字符 | 实体引用 | 意义 |
| &lt; | < | 小于 |
| &gt; | > | 大于 |
| &amp; | & | 和号 |
| &apos; | ' | 省略号 |
| &quot; | " | 引号 |

严格地讲，在 XML 中仅有字符 "<"和"&" 是非法的.省略号、引号和大于号是合法的，建议替换为实体引用.

## 主界面实现原理

### 主界面工作流程

主界面工作流程图如下：

插件启动

初始化主窗口UI

界面消息循环

扩展点功能处理

子窗口管理操作

是否退出

卸载插件

解析扩展点管理插件推送消息，添加功能点

### 初始化窗口UI

初始化主窗口UI，初始化成员变量.

### 界面消息循环

界面事件循环捕获，连接扩展点功能相应动作，子窗口管理操作，扩展点功能点添加.

### 扩展点功能处理

根据扩展点功能展示方式，对扩展点响应后功能进行处理.

### 子窗口管理

子窗口下的管理操作，停泊框管理，快捷按键管理，中央窗口分页.

### 解析扩展点插件推送消息，添加功能点

解析扩展点插件推送消息，添加功能动作到菜单栏或工具栏.

## 窗口分割实现原理

### 基本原理

* 通过拖放Tab页的项目，将选中的Tab页指针放入继承自QMimeData的拖拽数据项中；
* 截取拖放Tab页Widget截图，设置截图到拖拽数据上；
* 根据中央窗口布局，分为水平布局与垂直布局，如果是水平布局，则拖拽在中央窗口左右两边时，显示预览窗口，如果是垂直布局，则拖拽在中央窗口上下两边时，显示预览窗口；
* 拖拽时，判定不在中央窗口预览窗口出现判定区域，则可以向其它Tab窗口拖放当前Tab页；

### 事件处理

需要对Tab页，中央窗口与预览窗口的拖拽，放下事件进行重载或过滤处理.

* Tab页：过滤拖拽进入，拖拽移动，放下，拖拽移出事件；
  + 拖拽进入：记录防抖动记录点；
  + 拖拽移动：根据区域范围判断是否添加预览Tab页；
  + 放下：移除中央预览窗口，移除预览Tab页，加入拖拽Tab页指针到当前窗口；
  + 拖拽移出：移除预览Tab页；
* 中央窗口：重载拖拽进入，拖拽移动，放下，拖拽移出事件；
  + 拖拽进入：记录防抖动记录点；
  + 拖拽移动：根据区域范围判断是否添加预览Tab页.水平布局，判断区域为中央窗口左右两边横向200像素范围.垂直布局，判断区域为中央窗口上下两边垂直200像素范围；
  + 放下：移除中央预览窗口，移除预览Tab页，加入新的Tab窗口，Tab页加入到新的Tab窗口中；
  + 拖拽移出：移除预览Tab页；
* 预览窗口：过滤操作与中央窗口类似，不再一一阐述；

## 扩展点菜单实现原理

### 菜单实现原理

扩展点菜单根据需要，需要能动态的加载到菜单栏中，并且由于插件机制的特殊性，无法判定插件加载的先后顺序.所以需要在扩展点xml文件中指定动作/菜单所在位置.

为方便菜单的添加与删除等管理操作，我们采用二叉树的形式来实现菜单的组织.

#### 菜单管理

* 构建菜单二叉树，二叉树右边为同级菜单/动作，左边为子菜单/动作；
* 根据扩展点xml中推送的消息，向二叉树中加入菜单项；
* 根据动作群组的分组，向每个菜单中添加动作时，都需要先删除该菜单中的动作，然后从二叉树中提取出菜单中的动作，并按需求排序后加入菜单；

## UML图

### XML解析器类图



### MainWindow类图



### CentralWidget类图



### 快捷键管理对话框类图



## Undo/Redo框架设计

### 功能描述

在使用本测井软件的过程中，会经常遇到这样的情况: 进行了某一个操作然后在对这个操作的结果进行保存前，可以进行Undo/Redo操作，以达到取消上次操作的目的.

### 典型功能用例

用例1:在对绘制在道上的曲线进行鼠标描绘操作，在保存这根曲线前，在工具条上按undo按钮，取消这次的鼠标描绘操作，曲线应该恢复到原来的状态，相应的曲线数据也应该恢复到原来的状态.

### 性能描述

Undo 或者 Redo操作应该立即生效，界面上能有任何停顿，同时数据也要完成类似的操作.

### 输入项目

需要进行Undo和Redo操作的对象.

### 输出项目

实现Undo和Redo操作.

### 系统架构

整个系统将被分成两部分:操作部分和操作队列管理.

在操作部分，所有的操作都将通过使用Command模式包装成单个的command.在操作队列的管理部分，前述的一个个的command将被放进一个队列里采取observer模式来进行管理，每一个操作队列对应着一个相应需要进行undoredo操作的对象.

同时，为了方便地管理多个对象的undo/redo操作，又将前述的操作队列放进一个新的队列里进行管理，这个新的队列被称为group，他们之间的关系大致如下图:

Undo-RedoStack

Cmd1

Cmd2

Cmd3

Cmd4

Cmd5

被操作对象

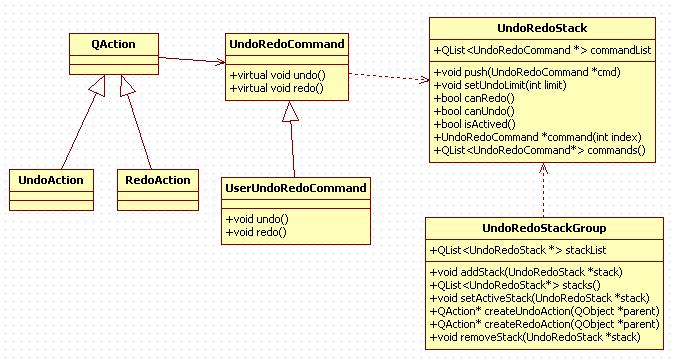
Undo-RedoStack-Group

Undo-RedoStack01

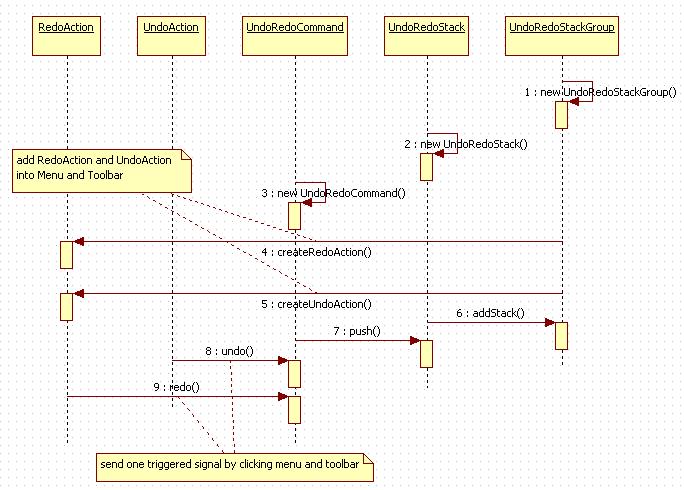
Undo-RedoStack02

Undo-RedoStack03

主要类关系图如下:



### 流程逻辑



流程描述:

第一步:生成一个UndoRedoStackGroup对象.

第二步:对应一个需要进行 Undo-Redo操作的对象，生成一个UndoRedoStack对象.

第三步:将需要进行Undo，Redo操作的步骤封装成一个UndoRedoCommand对象，实现UndoRedoCommand里的undo()，redo()函数，第一次的操作和和向前的操作都封装在redo()函数里，需要回滚的操作都封装在undo()函数里.

第四步:调用UndoRedoStackGroup的createRedoAction()函数生成一个RedoAction对象，并把它加到菜单和工具条中.

第五步: 调用UndoRedoStackGroup的createUndoAction()函数生成一个UndoAction对象，并把它加到菜单和工具条中.

第六步:将生成的UndoRedoStack对象加到UndoRedoStackGroup对象中.

第七步:将生成的UndoRedoCommand对象加到UndoRedoStack对象中.

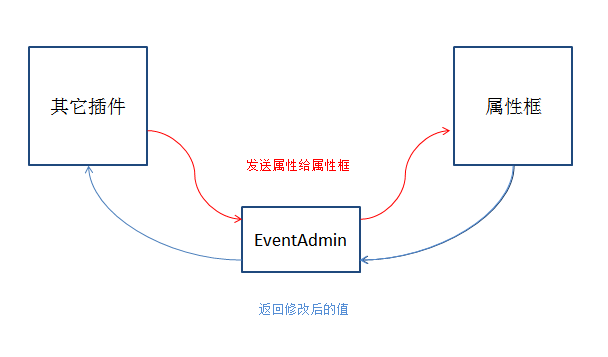
第八步:通过点击菜单和工具条上的Undo图标来触发UndoRedoCommand中的Undo()函数，实现回滚操作.

第九步: 通过点击菜单和工具条上的Redo图标来触发UndoRedoCommand中的Redo()函数，实现向前操作.

## 属性框

### 概述

属性框在ACME 3.0中作为一个插件而独立存在，属性框插件使用ctkEventAdmin提供的事件机制与其它插件进行交互.当其它插件需要显示属性信息时，通过ctkEventAdmin发送事件消息给属性框插件；当属性被改变后，属性框插件会通过ctkEventAdmin将改变后的属性值反馈给原插件.



### 属性类型

enum PropertyType

{

PROPERTY\_TYPE\_INVALID = 0， ///< invalid property type

PROPERTY\_TYPE\_GROUP， ///< group property type

PROPERTY\_TYPE\_BOOL， ///< bool property type

PROPERTY\_TYPE\_INT， ///< int property type

PROPERTY\_TYPE\_DOUBLE， ///< double property type

PROPERTY\_TYPE\_STRING， ///< string property type

PROPERTY\_TYPE\_ENUM， ///< enum property type

PROPERTY\_TYPE\_COLOR， ///< color property type

PROPERTY\_TYPE\_DATE， ///< date property type

PROPERTY\_TYPE\_POINT， ///< point property type

PROPERTY\_TYPE\_SIZE， ///< size property type

PROPERTY\_TYPE\_FONT， ///< font property type

PROPERTY\_TYPE\_LINE\_STYLE， ///< line style type

PROPERTY\_TYPE\_LINE\_WIDTH， ///< line width type

PROPERTY\_TYPE\_FILL\_STYLE， ///< fill style type

PROPERTY\_TYPE\_MUTEX ///< mutex property type

};

PROPERTY\_TYPE\_INVALID: 无效属性

PROPERTY\_TYPE\_GROUP：Group属性，表示某一类属性信息

PROPERTY\_TYPE\_BOOL：bool属性，表示是否选中

PROPERTY\_TYPE\_INT：int属性，表示具体值

PROPERTY\_TYPE\_DOUBLE：double属性，表示具体数字

PROPERTY\_TYPE\_STRING：string属性，表示具体文字信息

PROPERTY\_TYPE\_ENUM：enum属性，表示可选择的信息

PROPERTY\_TYPE\_COLOR：颜色属性，表示颜色信息

PROPERTY\_TYPE\_FONT：字体属性，表示字体信息

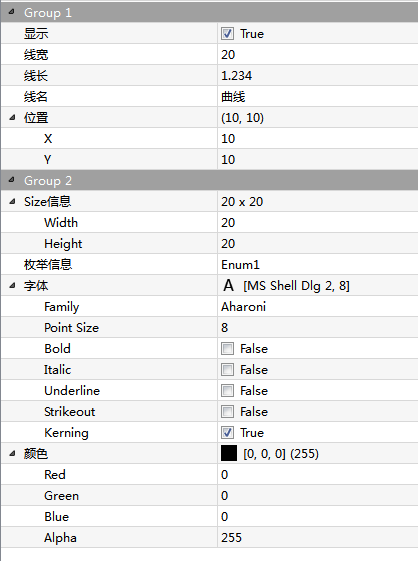
PROPERTY\_TYPE\_LINE\_STYLE：曲线类型，表示曲线类型信息（实质为enum信息）

PROPERTY\_TYPE\_LINE\_WIDTH：曲线宽度，表示曲线宽度信息（实质为enum信息）

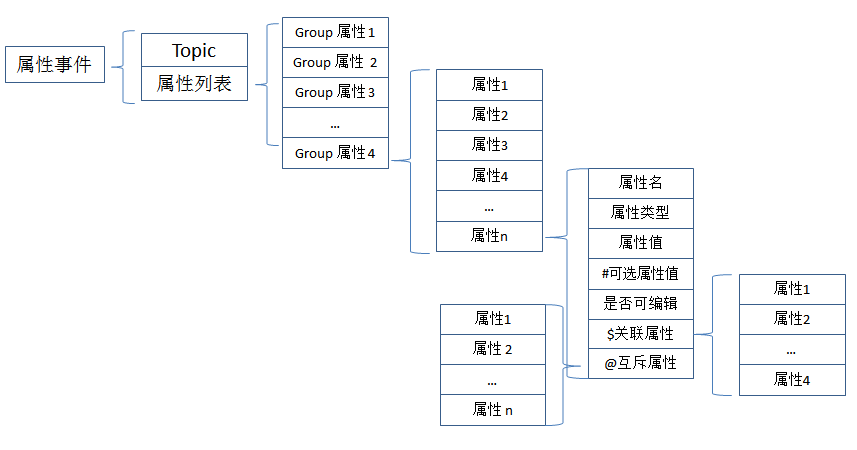
PROPERTY\_TYPE\_FILL\_STYLE：填充类型，表示填充类型信息（实质为enum信息）

PROPERTY\_TYPE\_MUTEX: 互斥类型，表示几个不能同时为真的属性

具体显示如下图：



### 事件结构



当一个插件（假设叫做PluginA）需要显示属性时，就构造一个类似上图的事件，其中各个字段的意义如下：

Topic：

表示PluginA订阅的关于属性变更的事件主题，当用户在属性框中将属性改变后，属性框插件就会以该topic给PluginA发送更新后的属性值.

属性列表：

表示插件PluginA需要显示的所有属性

属性名：

属性名，同时也会在属性框中显示

属性类型：

属性的类型，如颜色属性，文字属性等

属性值：

当前属性值

#可选属性值：

当前属性能够选择的所有值，只有enum类属性才需要可选属性值

是否可编辑：

当前属性能否编辑

$关联属性：

表示依赖于当前属性的属性，只有bool类属性才具有关联属性信息；当前属性为true时，关联属性可编辑，反之关联属性不可以编辑

@互斥属性：

表示不能与其他属性同时生效的属性，只有互斥类属性具有互斥属性信息

### 示例

#### 其它插件发送需要显示的属性信息给属性框插件

QList<QVariant> propertites;

QHash<QString， QVariant> width;

width[PropertyConstants::PROPERTY\_NAME] = QLatin1String("width");

width[PropertyConstants::PROPERTY\_VALUE\_TYPE] = PROPERTY\_TYPE\_INT;

width[PropertyConstants::PROPERTY\_VALUE] = 10;

width[PropertyConstants::PROPERTY\_VALUE\_MAX] = 100;

width[PropertyConstants::PROPERTY\_VALUE\_MIN] = 0;

width[PropertyConstants::PROPERTY\_EDITABLE] = true;

QHash<QString， QVariant> height(width);

height[PropertyConstants::PROPERTY\_NAME] = QLatin1String("height");

height[PropertyConstants::PROPERTY\_EDITABLE] = false;

propertites.append(width);

propertites.append(height);

ctkDictionary properties;

properties[PropertyConstants::PROPERTY\_CLIENT\_TOPIC]

= "org/acme/ui/propertybrowser/topic";

properties[PropertyConstants::PROPERTY\_LIST] = propertites;

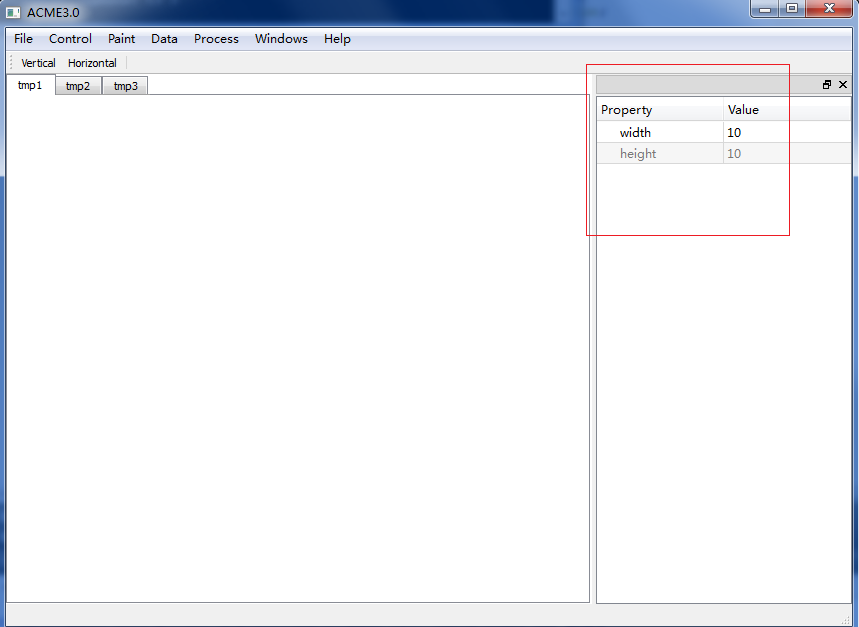
ctkEvent event(PropertyConstants::PROPERTY\_SERVICE\_TOPIC， properties);

m\_eventAdmin->*sendEvent*(event);

**说明：**

1. 示例中属性列表中有两个INT类型的属性，Width和Height
2. 两个属性的当前值都是10，取值范围为0~100
3. 两个属性的显示名分别为width和height
4. 属性width可编辑，height不能编辑
5. 接收属性变更事件的topic为"org/acem/ui/propertybrowser/topic"
6. 例子中m\_eventAdmin 是event admin service的指针

**上面例子的显示效果如下图：**



#### 其它插件接收来自属性框插件的属性变更信息

ctkDictionary props;

props.insert(ctkEventConstants::EVENT\_TOPIC，

QLatin1String("org/acem/ui/propertybrowser/topic "));

m\_eventAdmin->subscribeSlot(this， SLOT(handleEvent(const ctkEvent&))，

props);

其中，handleEvent是需要接收属性变更的类的槽函数，而topic表示要接收事件的topic，该topic和上面发送属性时的topic一致.

# 图形绘制与交互

## 概述

图形的绘制，主要包括：

* 画布绘制和显示
* 图件绘制和显示

## 画布

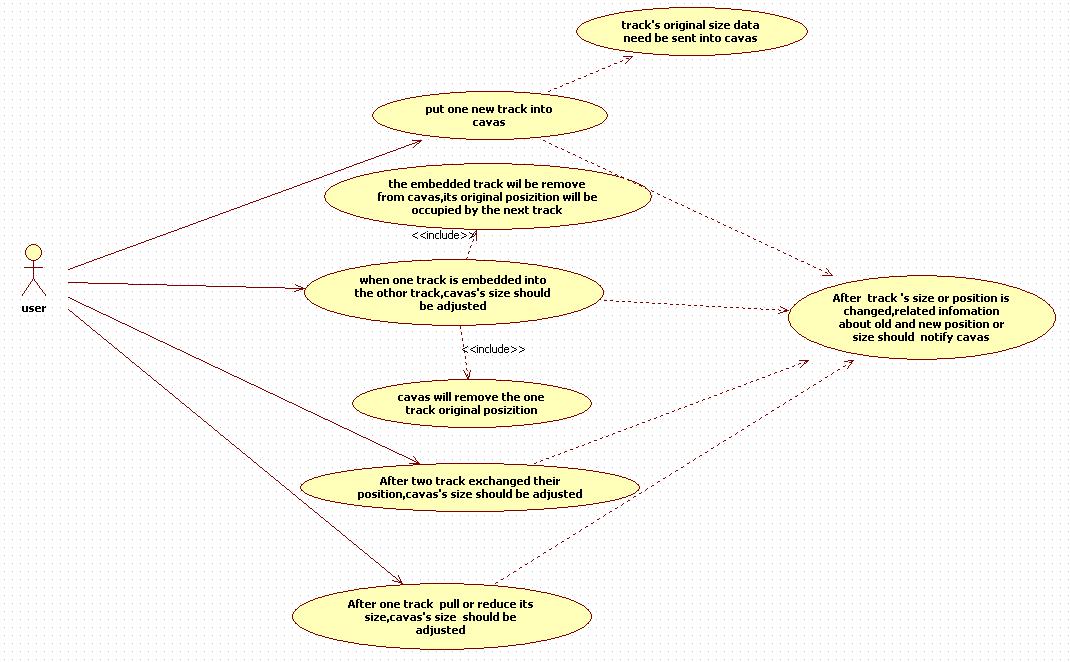
画布（canvas）是整个绘图框架用来放置图件的容器，是整个绘图的最底的一层.

画布能够对内部图件管理与布局.

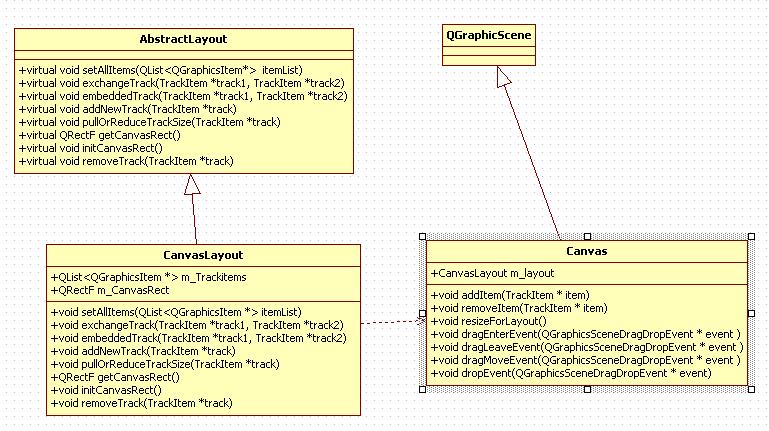
画布能够将内部图件整体进行缩放与旋转.

### 基础逻辑和流程

在画布上放置道，道可以在画布上通过拖放来改变位置，道可以进行嵌套，嵌套后，画布上的相关道的位置需要进行相应调整.同时，当单个道自己的尺寸通过拉伸改变时，画布上其他道的尺寸要相应改变.在画布上进行相关操作的用例图如下:



### UML图



### 接口设计

当道的尺寸，位置发生改变的时候需要发消息通知画布，画布会根据这些消息来调整画布各个道的位置.由于道和画布最终都会被包装成CTK插件形式，因此，会使用eventbus机制在不同的插件之间传递消息.对应的在画布实现类Canvas中要实现下面的响应接口:

dragEnterEvent，dragLeaveEvent，dragMoveEvent，dropEvent.，resizeForLayout

## 图件

图件（Item）是系统绘制的基本单元.所以画布上进行绘制和显示的元素，都可以理解为图件.

图件能够绘制基于各种数据源和数据类型的多种数据.

图件能够实现基于具体功能应用的交互.

图件能够基于数据分析的应用解析.

图件之间能够根据具体约束条件相互嵌套.

图件采用双缓冲绘制方式绘制.

图件主要包括：

* 道
* 曲线
* 注释图件
* 等

### 道

#### 定义

道是图件的容器对象，道本身也是一个图件，根据一定的约束条件，能够内嵌多个图件，也能够被嵌入到其它图件中.

道由道头与道身两部分组成，道头一般用来放置内部图件的数据相关绘制信息，道身一般用来放置内部图件的数据绘制对象.

#### 属性

深度：每个道都有自己的深度值范围（minDepth， maxDepth）

深度比例：显示深度与实际深度之间的比例

### 曲线

#### 定义

曲线是一个图件对象，必须内嵌在道中使用.

每个曲线对象包含两个部分，曲线绘制对象与曲线幅值标尺，曲线绘制对象放在道身中，曲线幅值标尺放在道头中.

曲线绘制根据道提供的深度标尺与曲线幅值标尺建立的坐标系来绘制曲线.

#### 属性

曲线数据：一组坐标数据，每个数据点包含深度值（Y值），幅值（X值）

曲线类型名：曲线类型名称

曲线数据名：曲线数据名称

### UML图



## 拖拽

### 需求

* 用户能够从曲线列表拖拽某曲线到测井视图的道头，向此道增加对应的曲线图形显示（包括道头，道身）
* 用户能够在同一测井视图的道之间通过拖拽曲线头来移动曲线
* 通过拖拽的落点不同，拖拽的曲线头能够插入在道头内不同的位置上进行显示
* 当用户从道头拖拽曲线到道外放下，弹出对话框询问是否删除曲线，若用户选择是，则删除对应的曲线显示
* 用户可以通过拖拽道头的方式来改变道在画布上排列的顺序

### 基础设计

#### Qt拖拽技术概述

出于跨平台的考虑，直接使用Qt的拖拽机制，通过改写以下三个对应的拖拽函数来完成拖拽：

* void dragEnterEvent()
* void dragMoveEvent()
* void dropEvent()

对每一个需要接收拖拽事件的图件，在图件初始化的时候设置拖拽Enable标志：setAcceptDrops(true)，处理完拖拽事件后，调用event>accept()来表明此拖拽事件已被处理，如果该图件对某拖拽事件不处理，则可简单调用基本的处理函数，并将拖拽事件发送给下一个想要处理的图件( 在同一个拖拽落点上可能有多个图件 ).

#### 拖拽的信息传递

为了在拖拽中方便的加入我们自定义的数据格式，我们从QMimeData派生一个自定义类：TrackContentMimeData，声明如下：

class GRAPHICSLIBRARYSHARED\_EXPORT TrackContentMimeData : public QMimeData

{

Q\_OBJECT

public:

virtual void setItemList(const QList<TrackContentItem \*> &itemList);

virtual QList<TrackContentItem \*> getItemList() const;

virtual void setFormats(const QStringList &formats);

virtual QStringList formats() const;

private:

QStringList m\_formats;

QList<TrackContentItem \*> m\_itemList;

};

其中， m\_itemList用来保存拖拽的源图件指针，支持多选择对像同时拖拽.

m\_formats用来标识拖拽源的一些数据信息，用文本来进行描述.

由于继承自QMimeData， 因此也支持Qt默认的拖拽数据格式传递.

#### 拖拽的流程及设计

1. 曲线列表向测井视图道头拖拽曲线

曲线发起拖拽的时候，只需要向拖拽对像传送曲线名和对应的文件名，并标识拖拽类型为：“DragCurveFromCurveList”，伪代码如下;

mimeData->setText(QLatin1String("DragCurveFromCurveList"));

mimeData->setData(QLatin1String("filename")， m\_curveModel->getFileName().toUtf8());

mimeData->setData(QLatin1String("curvename")， m\_curveModel->getCurveName(index.row()).toUtf8());

QDrag drag(this);

drag.setMimeData(mimeData);

drag.exec( Qt::CopyAction );

接收此类事件的图件对像为TrackHeadItem， 因此在TrackHeadItem初始化时，加上拖拽Enable的标识：setAcceptDrops(true)， 接收到拖拽信息后，通过获取的文件名和曲线名，通过工厂生成对应的CurveItem， 并加入到目标道中进行显示.

1. 道间的曲线拖拽

道间的曲线拖拽为了实现在落点处插入拖拽的曲线头，在道头的曲线头区域增加一个垂直布局，所有曲线头都在这个布局中自动排列.

曲线头图件作为此类拖拽的起点，拖拽发生时将父指针( 譬如CurveItem)加入到拖拽数据，这样可以方便接收方进行数据处理，并标明此类拖拽类型为：”TrackContentItemDrag”.为了让用户能方便的看到当前拖拽的曲线，因此在拖拽过程中加入拖拽的道头的显示.

接收此类拖放的图件也在道头，即TrackHeadItem， 当判断到拖拽类型为”TrackContentItemDrag”时，解析出拖拽源的指针，然后通过TrackItem的removeContentItem()方法来去掉源图件于源道的父子关系，然将目标曲线添加到当前道中显示即可.

在添加的过程中，需要通过判断拖拽的落点，通过落点获取应该将曲线头加到道头内的指定位置，最终使用TrackHeadItem的insertContentItem()方法来进行添加.

1. 曲线的拖拽删除

与道间曲线拖拽相同，曲线头图件作为此类拖拽的起点.

在画布上道外区域响应此拖拽消息，因为QGraphicsScene本身不处理拖拽消息，因此我们需要在对应的QGraphicsView里来处理.从QGraphicsView派生一个类：class CanvasView : public QGraphicsView，主要实现以下两个接口：

void dragMoveEvent(QDragMoveEvent \*event)

void dropEvent(QDropEvent \*event)

在dragMoveEvent()中主要是对拖拽事件进行过滤，仅对拖拽类型为”TrackContentItemDrag”并且落点不在道内区域的事件进行accept， 其它情况都调用基类函数：QGraphicsView::dragMoveEvent( event )，由系统交由下一个想要处理此拖拽事件的图件.

在dropEvent（）中，判断拖拽类型后，弹出消息对话框让用户确认，若确认删除，则通过调用目标所在道的delTrackContent()函数来完成曲线的删除.

1. 道的拖拽

TrackTitleItem作为此类拖拽发生的起点，发生拖拽时，将当前所属的道的指针（TrackItem\*）传到拖拽数据中，同时设置拖拽类型为“TrackDrag”.

拖拽过程中为了标明拖拽对像，向拖拽增加道头Title的重绘bitmap.

画布的整个道身以上区域作为此类拖拽的落点.在得到指定类型的拖拽事件后，通过落点判断道拖拽到指定位置，调用ICanvasScene的changeTrackIndex接口来实现道的位置的切换.

1. 道的删除

同上，TrackTitleItem作为此类拖拽发生的起点.

在测井视图画布以外的区域作为此类拖拽事件的接收区域.在得到指定类型的拖拽事件后，弹出对话框，让用户确认是否从测井视图上删除目标道，确认后调用ICanvasScene的delTrack接口来删除该道在视图上的显示，同时删除道内所有的对像，并重新排列及重绘剩下的道.

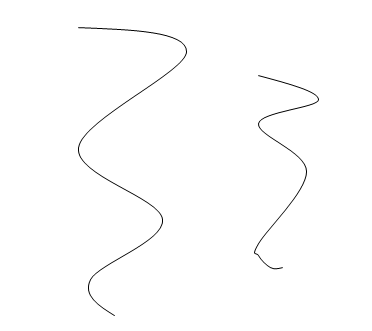
## 曲线填充

### 需求

* 要求在任意选定的两条曲线之间使用指定的样式和颜色进行填充
* 要求在任意选定的一条曲线和一条基准直线之间使用指定的样式和颜色进行填充

### 基础设计

#### 两条曲线之间的填充算法：

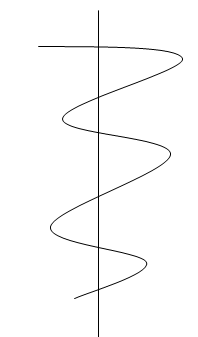


两掉曲线之间存在很多位置关系，上图只是展示了其中的一种，下面的算法就是基于上图，其他的位置关系也采用内似的方法进行填充。

1. 选定右边曲线，找到它的Y值最大和最小的点
2. 从Y值最小的点开始，沿着Y值增大的方向遍历整个曲线，直到最后一个Y值最大的点，在遍历的过程中，不断把这些遍历到的点加入到一个多边形Polygon中
3. 当遍历到最后一个点后，从这个点开始做一根和Y轴平行的直线和左边的曲线相交，由此产生一个交点，如果这个交点没有在这根曲线上，就要在该曲线上找出这个交点两边的点，利用找出的这两个点构成一条直线，然后把这个交点的Y值带入到这个直线中，利用直线方程得到这个交点的X值，最后把这个新产生的交点加入到第根曲线中.如果这个交点本身就在第二根曲线上，则不做任何处理，只是记录下这个交点
4. 同样的道理，可以从被选定的右边曲线的第一点开始，做一根和Y轴平行的直线和左边的曲线相交，采取和步骤3同样的方法，找到或者产生一个交点
5. 上面两步做完后，从步骤3或4中产生的交点开始向着另一个交点的方向遍历第二根曲线，直到到达另一个交点为止，在这中间遍历经过的所有的点都被加入到和执行步骤2时相同的多边形Polygon中

利用QT的填充功能填充上述步骤产生的多边形.

#### 一条曲线和一条基线之间的填充：



一根曲线和一根直线之间的位置关系也存在很多种，上图仅仅是其中的一种，下面的算法就是基于上图，其他的位置关系也可以采用类似的方法进行填充.填充的时候以基线为准，可以将填充分为左填充和右填充两部分，下面的算法以右填充为例，左填充采取类似的算法.

1. 从曲线的起点开始想着Y值增大的方向循环，每次取两个点P1和P2，然后将这两个点的X值和基线的X值进行比较，如果这两个点的X值都比基线的X值大，就把这两个点加入到一个多边形Polygon中.如果P1的X值比基线的X值小，P2的X值比基线的X值大，说明P1在基线的左侧，而P2在基线的右侧，这时候需要在P1，P2之间找到一个和基线的交点，并把这个交点交入到多边形Polygon中，如果这个交点本身就在曲线上，就直接把这个交点加入到多边形Polygon中，如果这个交点本身没在曲线上，就需要利用P1，P2构造直线方程，然后把基线的X值带入这个直线方程求出该交点的Y值，接着把这个新产生的交点加入到多边形Polygon中

如果P1的X值比基线的X值大，P2的X值比基线的X值小，说明P1在基线的右侧，而P2在基线的左侧，同时说明曲线已经进入到基线的左边，这时需要把右边的这个多边形进行填充.此时同样需要进行判断P1和P2之间是否和基线存在交点，如果这个交点本身就在曲线上，就直接把这个交点加入到多边形Polygon中，完成这个多边形的填充，如果这个交点没在这个曲线上，就需要使用同样的方法制造一个新的交点然后加入到多边形Polygon中

1. 上边的多边形加入到一个List中
2. 继续沿着曲线进行遍历，重复步骤1和2的过程
3. 最后利用QT的填充技术把List中的所有的Polygon进行一次填充，完成整个过程

## 波列曲线

### 需求

* 绘制阵列数据深度点上的横向连线.
* 定制起始点和终止点.
* 定制线条颜色和填充颜色.
* 多种绘制方式：正半轴填充、负半轴填充.

### 基础设计

#### 绘制区域坐标系转换

波列数据包含两个维度，每个深度点都包含一组数据，横向波形是把每个深度点的数据集合以点连线方式显示出波形.由于深度点是连续变化的间隔点，即纵向值一直会变化增加，而深度点上的数据集合变化范围在某个固定值范围内，需要在深度的坐标系中创建一个小的笛卡尔坐标系来展示数据集合.

y

x

深度坐标轴

(0， 0)

0。3

0。2

0。1

横向坐标轴

程序只绘制屏幕可视区域内的数据，通过屏幕的可视区域映射到深度坐标轴，得到需要绘制的深度数据，并计算得到第一个可视区域内的小坐标系的深度坐标点.绘制该深度坐标点的数据集合.绘制完该深度点后，移动该笛卡尔坐标系到下一个需要绘制的深度点继续绘制，可设置跳过的深度点个数.

#### 子坐标系绘制方式

小笛卡尔坐标系中的Y坐标轴为数据值坐标，X坐标轴按照深度点数据集合的数据个数均分，即该深度点的数据均匀绘制在X坐标轴.用户可以选择绘制该深度点数据的起始和结束的数据点，X坐标轴根据用户选择的数据点个数重新调整数据点间隔，并重新绘制.

Y坐标轴可以设置绘制的物理像素点个数（通过深度值映射像素值）和真实数据的取值范围（如图中数据点数值的最大值和最小值）.基准线作为填充上下半轴的分界线，用户可以自由设置基准线在Y坐标轴上的位置.

y

x

用户选择数据起始点

用户选择数据结束点

按照选择的数据点个数均分X坐标轴

数据点数值最大值

基准线

数据点数值最小值

#### 曲线绘制和填充

曲线绘制前，将需要绘制的数据点映射为子坐标系中的坐标点.其中Y坐标轴为数据点数值，保持不变，X坐标轴会根据该深度点的用户所选择的数据点个数对X轴均分，并根据每个数据点的顺序位置确定X坐标值.

转换完成后，使用多边形连线绘制方式对该身段点上的所有数据点坐标进行连点完成曲线绘制.在坐标转换时，使用基准线的基准值完成曲线的上下半轴的数据取值操作，即如果填充上半轴则超过基线数值的数据点使用基线值，如果填充下半轴则低于基线数值的数据点数值使用基线值.数据转换完成后使用多边形填充绘制方式完成曲线填充.

#### 渐变色绘制

使用QLinearGradient生成渐变色调色板，通过QPen::setBursh方法设置给painter，并把调色板设置到Y坐标轴的实际深度值范围，曲线和填充绘制会自动套用画笔的调色板设置进行绘制.

## 实时显示

### 需求说明

测井主控程序可在实时测井或者重测井时，将测井数据通过网络发送出来，ACME3.0显示客户端在连接上主控程序的情况下，建立一个新工程，工程里的数据对象来自网络数据，同时创建一个新视图，视图使用默认显示模板，用户可在此模板里添加曲线对像，然后在视图里就能看到对应数据对像的实时图形.伴随着图形显示的深度变化，显示区域也会相应滚动，并尽可能保证当前实时图形在可视区域.

网络连接：采用Tcp/Ip协 议， 在ACME3.0程序里需要增加相关的网络连接设置.

实时显示时，用户可拖动滚动条，观察其它区域的图形，并通过快键操作（按钮，快捷键等）回到实时数据区域.用户将滚动条拖离实时区域后，视图也会保持滚动，表明当前实时测井正在进行中

### 典型功能用例

* 用例1：打开ACME3.0程序，点击菜单项：网络连接配置，用户可设置并保存网络连接服务器的IP地址和端口号.
* 用例2：打开ACME3.0程序，点击工具栏按钮：“网络连接”，若连接服务器成功，则“网络断开”按钮高亮，表示网络连接成功，此时点击“网络断开”，则程序断开网络连接，“网络断开”按钮变灰，“网络连接”按钮高亮.若连接失败，则弹出消息框，提示“连接服务器失败”.
* 用例3：当网络连接成功时，若收到服务器端发出的实时数据，则在工程区显示一个新工程，同时在曲线列表中列出当前实时数据所包含的曲线，在此工程新建一个视图，视图使用一个空模板加载显示.
* 用例4：在用例3的情况下，用户可拖动曲线列表中的曲线到显示区，向显示区增加新的曲线，同时对应的曲线将在显示区实时显示，根据测井方向的不同，显示区域也相应滚动以保证曲线实时深度在可见区域.
* 用例5：在用例4的场景下，用户想要回看一下之前显示的数据，可手动拖动滚动条到指定深度（或时间，或帧数），即用户可以手动调整当前的显示区域，并且整个显示区域继续保持实时滚动.
* 用例6： 在用例5的场景下，用户可点击右键菜单：跳转到实时深度（时间，帧数），则显示区域将滚动到当前的实时数据位置.

### 性能描述

* 网络传输

目前的数据传输速率一般在512 kbps以下，设计为 <= 4M bps，以满足未来的扩展需求.

* 实时显示

显示滚屏速度由数据速率驱动，设计为 <= 5cm / s

### 输入项

来自于网络的实时测井数据(或重测井数据)

### 输出项

ACME3.0程序：屏幕上的实时数据显示，根据配置的不同，显示不同的曲线实时滚动波形，反映当前测井的实时数据情况

### 系统架构

从功能上，可分解为以下六大模块：

* 实时数据收发

负责连接服务器，发送命令和接收实时数据，接收到的数据放到实时数据缓冲中， 每一个实时数据收发实例管理一个实时数据缓冲.

数据的收发单独放到一个线程里处理.

* 实时数据缓冲

用来存储实时数据，并向实时显示绘制提供所需要的显示数据，需要管理缓冲大小，读写指针的循环移动，并保证读指针不会超过写指针.允许多线程同时访问缓冲，保证线程安全.

* 实时显示管理

管理实时数据收发模块，设置数据收发模板的参数，控制数据收发的开始和结束.

* 实时数据解析及处理

接收到实时模块的数据后，解析数据，并存储到实时数据缓冲中. 数据解析、存储在一个单独的线程中运行.

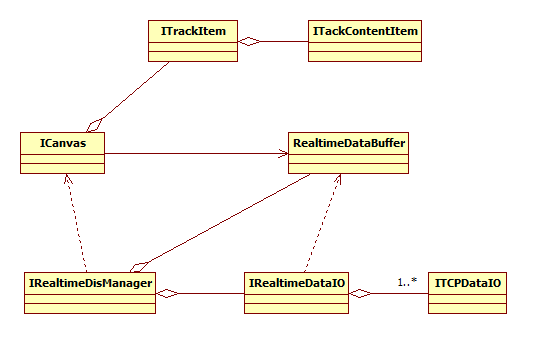
* 实时显示绘制

从实时数据缓冲中获得新的显示数据，更新到绘图区显示的曲线中去，控制显示区的自动滚动.能响应用户对显示曲线的新增、删除，移动，显示参数修改等操作.

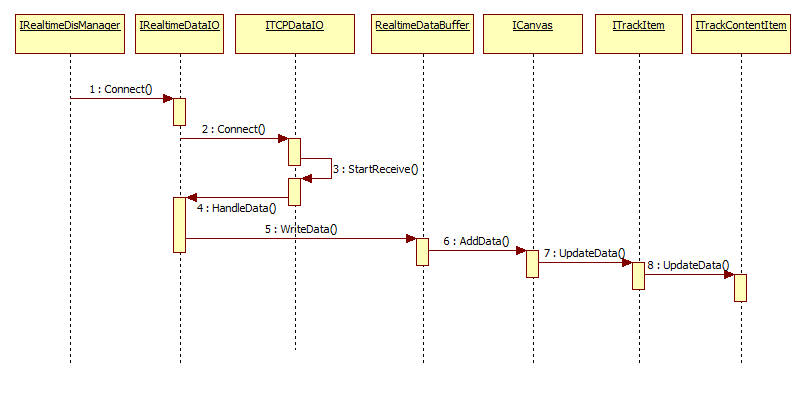
* 主界面设置及控制

在ACME程序里能设置并保存网络连接参数，在工具栏和菜单上增加“网络连接”，“网络断开”的操作.当收到实时数据时，新建一个工程，并显示对应的曲线列表（曲线列表），用户可将曲线列表里的曲线拖到绘图区.

### UML图



### 逻辑流程



* 每一个ITCPDataIO负责一个Tcp Socket连接，IRealtimeDataIO聚合一个或多个 ITCPDataIO.
* IRealtimeDataIO需要向ITCPDataIO设置每收到多少个字节的数据后进行一次回调.
* ITCPDataIO在单独的线程里负责接收实时数据，收到指定大小的实时数据后（或超过），调用IRealtimeDataIO设置的回调函数(基于效率的考虑，这里不使用信号/槽连接方式)来处理收到的实时数据.IRealtimeDataIO在回调里对原始数据进行解析，并根据需要，进行下面三种处理: 1 实时显示; 2 实时打印 3实时存储.
* (实时打印、实时存储请参考相关文档) 实时显示时，获取RealtimeDataBuffer的写指针，当判定读写指针的距离满足要求时，从写指针的位置向数据缓冲写入解析后的数据，并移动写指针的位置.
* IRealtimeDisManager从IRealtimeDataIO得到反馈，发现有实时数据接收到时，向系统框架发起请求，新建一个工程，打开曲线列表，并按默认模板新增一个显示视图，得到对应的ICanvas接口.同时IRealtimeDisManager新起一个线程来进行实时数据缓冲数据的读取，当发现读写指针间距满足条件时，从读指针读取指定的数据缓冲大小，二次处理后，发给ICanvas， ICanvas再发送给对应的道及道内的曲线对像，曲线对像得到新增的对据后，更新自身的显示数据缓存，并刷新对应的显示区域.同时ICanvas根据新增的数据量，进行显示区域的滚动.

### 数据输入输出模块设计

由于扩展性的考虑，为了适应未来可能的Tcp/IP v6协议，数据采集卡，串口，USB口， CAN卡中的一种或多种并存，因此IReatimeDataIO被设计为与硬件无关的一个中间层.

在IReatimeDataIO内部以聚合的方式将具体硬件的操作转发给对应的IO类，如ITcpIO， IUsbIO、IComIO、IDataCardIO，根据init传入参数type的不同，IReatimeDataIO将打开对应的外设，启动相应的IO子模块来与硬件交换数据，同时设置一个对应的回调函数来解析收到的数据，然后设置好回调响应的时机（即收到多少数据时响应一次回调函数），具体与硬件打交通的IO子模块在接收到指定数据量的数据后，把数据交由对应的回调函数处理，以此实现数据收发与数据分析的解耦.在这些回调函数中解析数据，然后转存到RealtimeDataBuffer维护的数据缓冲里.

以Tcp Socket数据传输为例，典型的初始化方法如下：

bool IReatimeDataIO::Init(int type， …)

{

……

if(type = = TCPIP\_PORT)

{

if (!OpenTcp())

return false;

//设置接收缓冲区大小

if (!SetDataBuffer(iBufferSize))

return false;

//注册回调函数

if (!RegisterCallBack(HandTcpData， this))

return false;

//开始采集

if (!StartCollect())

{

Sleep(500);

if(!StartCollect())

{

CloseDevice();

return false;

}

}

}

……

}

其中的SetDataBuffer就是用来设置回调时新收到的数据缓冲的大小，当子模块收到指定大小的新数据时，调用回调函数，回调函数的参数如下：

bool CALLBACK IReatimeDataIO::HandTcpData(void\* buffer， void\* context)

{

char\* buf = (char\*)buffer;

IReatimeDataIO \* realtimeIO = (IReatimeDataIO \*)context;

//数据解析

……

}

回调函数使用两个参数，第一个参数为回调函数的设置者，第二个参数为此次接收到的新数据的缓冲.有了第一个参数，回调函数便拥有了回调设置者的丰富的信息，即可以对回调设置者的身份进行充分的判断.

当第一次连接失败时，程序稍作停顿（Seeep(500)），再尝试第二次启动，第二次确定无法进行正常的数据传输，才宣告连接失败，关闭设备后，返回false.

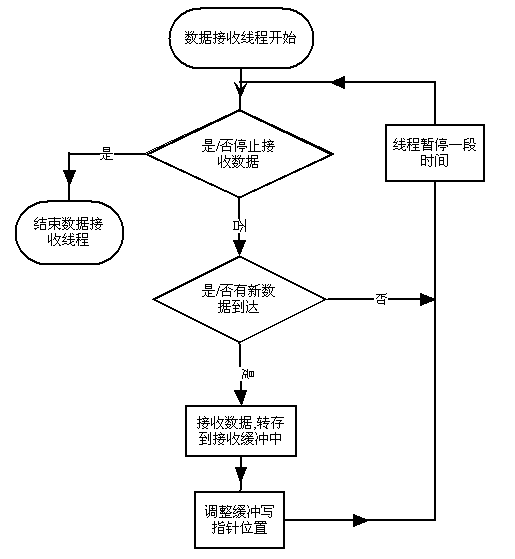
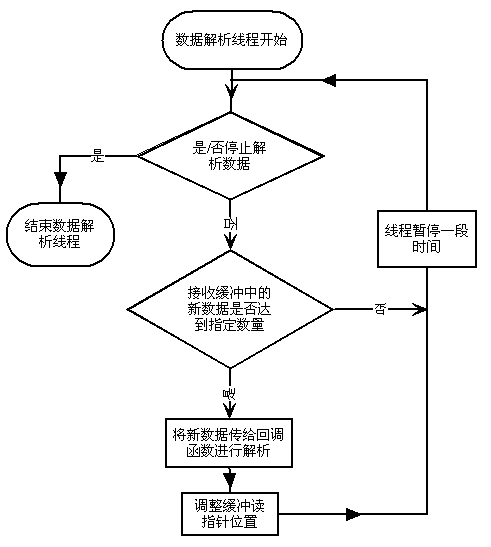
### 通讯子模块的设计

每一个与硬件通讯的子模块都将具体实现屏蔽在模块内部，并且在逻辑上与外部模块独立，实现松耦合的关系，且对外采用同样的公有接口，以方便IRealtimeDataIO的统一化管理，以实现良好的模块独立性和可重用性.

其主要功能函数设计如下：

* open（……）接口用来连接服务器
* SetBuffer（UINT size）接口用来设置每当收到size个字节新数据的时候，将这些数据作为参数转发给回调函数，这样的设计可以使得数据接收与数据分析解耦，让本串口模块可以在各种情况下被重用.
* PointerMove（int pos）接口用来调整回调时传送数据的位置，以保证回调时传送的指定数量的数据为完整的数据帧，便于上层模块使用.上层模块每次回调收到数据时，首先根据通信协议对其帧头进行判断，当判断数据为完整帧时，开始解析，若数据不是完整的数据帧，则在该缓冲中找到帧头相对缓冲的偏移pos，并以此偏移作为参数调用PointerMove(pos)，这样，串口类的数据指针将向指定方向偏移pos个字节，其结果是串口模块下一次回调需要收到pos + 指定回调数据长度才开始调用回调函数，并把pos以后的数据传给回调函数，在正常情况下，此时传送的数据将是完整的数据帧.
* 成员变量m\_bSendCmd是对发送命令时的一个保护.因为发送命令需要一定的延时，且每一个命令都需要等到硬件返回确认信息后才可进行下一次的命令发送，因此，在开始发送命令时，m\_bSendCmd置true，直到收到硬件的确认命令发送成功的信息后，方把m\_bSendCmd置false.在每一次发送命令时，都应该对此变量作检测，若m\_bSendCmd为true，表示有命令在发送中，当前发送应延迟发送.

数据接收和数据回调解析同步流程如下：



### 系统实时性设计与实现

出发扩展性的考虑，系统可能从多个端口同时输入实时数据，每一个端口的数据接收与发送都需要一个线程，同时，为了避免主界面UI的影响，显示端每一个显示模块也是一个独立的线程在运转，因此，数据在这些线程之间流转，既要保证其正确性，又要有良好的实时稳定性，在设计手法上，就出现了多个选择.为了让整个系统能稳定良好地运转，需要至少一个驱动源，来对系统的各部分进行时间分配.

#### 数据驱动

这个方法是实时系统中最常用的解法.它从数据输入端得到数据开始，来驱动整个系统，每接收到希望的数据后，依次进行所需要的数据解析，数据存储或传输，数据波形显示等，然后等待接收下一次的数据.这个方案的好处是整体驱动框架简单，容易实现，因此在数据传输率较小且其它模块相对简单的领域，特别是普通工业上的实时数据检测，有着广泛的应用.而它最大的两个缺点则是：

1. 因为在数据接收端，往往对其它模块的细节了解不够，便难以对循环中的子模块的运转进行精确的时间控制，特别是子模块也对时间分配有着复杂需求的时候.譬如显示如果性能要求较高，数据接收端不便于对其进行优化控制.
2. 由于硬件传输数据的不稳定性，可能会影响其它模块的时间分配，譬如因为某种原因，网络接收数据出现了短暂的停留，数据驱动的方案会使得整个系统停滞下来，而它里面的某个子模块可能此时并不依赖于这个数据，这时相关功能模块会遭受无妄之灾，会发生停顿.这是用户所不希望看到的.

#### 显示驱动

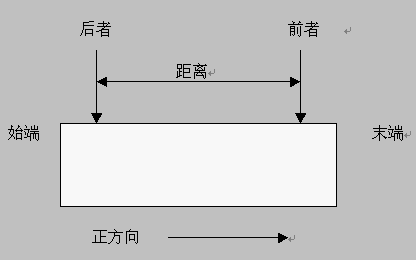
这是传统实时显示设备常用的方案，它一般是根据每秒所需要的显示帧数来设定一个定时器（根据精度需求不同，也可选用普通系统定时器或多媒体定时器），在这个定时器的回调函数中，来依次处理数据接收及解析，数据存储及传输，数据波形显示等内容.这样处理最大的好处是可以做到显示优先，且其中的显示可以不依赖于输入的数据，.其次，它对显示可以有比较精确的控制，如果需要的话，在这里实现垂直空白同步也比较容易.但它也有比较大的缺点，和数据驱动类似，它对数据接收模块会比较陌生，当数据传输为高速传输和较复杂的多端口传输时，不容易精确控制数据接收/解析/存储/传送的各个细节，导致的直接后果就是显示模块与数据模块的耦合程度加大，后期难以变更和维护.由于松耦合是整个系统设计的出发点，因此本系统并不采用这个方案.

#### 多点同步驱动

经过仔细研究需求以后，本系统决定采用一种新的多点驱动同步方案.在允许的范围内，牺牲掉一部分数据实时反应性(正负0.5s)，来获取各个模块对时间分配上的松耦合关系.各个模块由自身按照预先的设定来进行驱动，整个系统按照一个“追及问题”的思路，利用循环缓冲来实现多点同步驱动.

#### 追及方案

在中小学的数学应用题中，有一个常见的模型，那就是让甲和乙往同一方向运动，来求解其中的路程、时间和速度的关系.当甲和乙分别代表两个模块当前处理的数据时，这就变成了模块之间的追及问题.很显然，当它们之间的距离保持恒定时，两个模块对数据的处理在速度上是相同的，如果它们之间的距离为零，则两个模块是完全同步的.在两个模块分别对自己进行驱动的情况下，它们之间的距离是不可能保持为零的，任何一个模块在某个小的时间段内都可能出现“速度”不稳定的状况.比较可行的办法是让它们保持在一定范围的“距离”上，当落后的一方发现距离开始拉大时，便加快步伐，若发现距离开始拉近，便降低自己的速度，当即将追上前者时，则停下自己的脚步，直到两者之间的距离恢复到预定水平.这个“距离”，使得两者可在一定范围内保持良好的速度同步性，并且两个模块彼此之间无需知道对方的任何细节，可以最大限度地降低模块的耦合性，但两者之间较长的距离也降低了两者的实时同步性，是以牺牲实时性为代价的.

具体实现时，由于不可能设置一个无限长的跑道来使用，因此将直线运动改进为单向循环缓冲的模型，详见下图：

在初始时刻，前者和后者均指向缓冲始端，系统运作时，由前者开始按正方向运动，运动中系统不断进行检测，当发现后者落后于前者指定距离时，后者开始同向运动，两者运动到末端时均返回到始端重新开始.

#### 数据传输中的追及方案

这里的数据传输，主要指通讯子模块对硬件数据的连续接收，为了将数据接收与数据解析解耦（由上层模块回调实现），这两部分分别由一个线程来进行控制.接收线程只负责接收硬件数据，并写入到一个单向循环缓冲当中.而数据处理线程则不停地对已收到的未处理过的数据进行检测，当检测到其数量大于指定数值时，开始将这部分新数据通过回调函数传给上层模块进行数据解析和其它处理，然后移动后者的指针.很显然，这里的数据处理线程是完全依赖于数据接收线程的，可以将距离设置为零，获得最大的实时性，即可满足一般需要.由于硬件传输数据在较长时间段内是稳定的，所以较长时间间隔才调用一次的数据处理回调函数，可以吸收掉大部分数据接收线程细小的不稳定状态，为后续的同步处理提供一个可靠的基础.

#### 显示与数据传输的追及

显示与接收到的数据的同步和稳定性是整个系统中最核心的同步部分，在这里，显示的稳定性和稳定的延时性比实时同步更加重要.即，因为某个原因使两者的速度出现短暂的不匹配了，也要保证显示方面是有数据可用，能保持稳定和平滑的显现.所以在这里，数据接收为前者，显示为后者，且后者应该在需求允许的情况下，落后前者相对长一点的距离，根据实际情况，这个距离被设计为0.5秒的数据量.由于两者速度不可能精确匹配，因此需要实时地不断调整显示的速度（这里指显示对数据的取用数量），当显示与数据输入在速度上完全匹配时，可计算出每一次显示所取用的“标准量”的数据，基本算法如下：

while(1)

{

if(距离大于设定最大距离)

当前帧的显示取用比标准多的数据；

else if(距离小于设定最小距离)

当前帧的显示取用比标准少的数据；

else if(距离小于最小临界距离)

暂停显示；

else

当前帧的显示取用标准量的数据；

……

}

其中 最大距离 > 最小距离 > 最小临界距离

### 接口

主要模块接口设计如下：

class IRealtimeDisManager

{

public:

void init( ICanvas\* canvas );

virtual void connect();

void disConnect();

};

class IReatimeDataIO

{

public：

bool init(int type， …)；

bool sendTCPCmd(…);

bool sendUsbCmd(…)；

bool sendComCmd(…)；

bool sendCardCmd(…)；

static bool CALLBACK dataUsbAnalyse(…);

static bool CALLBACK dataComAnalyse(…);

static bool CALLBACK dataCardAnalyse(…);

bool stopCom(…)；

bool stopUsb(…)；

bool stopCard(…)；

};

class RealtimeDataBuffer

{

public:

bool init( int bufferSize );

void readerMove( int step );

void writerMove( int step );

dataPointer\* getRPointer();

dataPointer\* getWPointer();

int getPointerDistance();

};

class IRealtimeDataIO

{

public:

//连接服务器

bool open(……);

//设置一次交换缓冲区数据的大小

bool setBuffer(UINT iSize);

//启动监控线程

bool startMonitoring();

//挂起监控线程

bool stopMonitoring();

//注册回调函数

bool registerCallBack(PDATA\_CALLBACK pFunc，PVOID pRef);

//取消回调函数

bool unRegisterCallBack();

//移动交换缓冲区指针位置

void pointerMove(int iPos);

//返回每个帧的数据长度

UINT getBufferSize();

//关闭

void close(void);

//发送命令

bool SendCmd(char\* string， int iSize);

//读指针设置

void ReaderMove(int iPos);

};

### 限制条件

* 硬件需确保网络传输的稳定性.
* 对测井的网络传输格式有依赖，若传输格式升级则需要进行对应的数据解析升级.
* 在屏幕上滚动的速度不能太快，譬如400mm / s， 既有性能问题，也给最终用户带来不便

# 打印

## 打印

### 功能描述

1. 打印设置：提供打印设置界面，并在菜单项中添加打印设置菜单；保存/读取用户关于打印的设置参数.
2. 打印当前页面信息：如果当前页面为组合视图或图表视图，直接打印当前页面内容；如果当前页面为测井视图的话，需要额外指定打印深度，并且打印图头.
3. 实时打印：实时打印来自网络的测井视图信息.

### 典型功能用例

1. 用户点击打印设置菜单，程序弹出打印设置对话框；当用户修改打印设置选项后，保存最新的打印设置.
2. 用户打击打印菜单，如果当前显示页面为测井视图，则弹出打印深度范围对话框，当用户点击确定后，开始打印；如果当前为图表视图或者组合视图，直接开始打印.

### 输入项

1. 来自数据文件或实时测井数据
2. 组合视图内容
3. 图表视图内容

### 输出项

1. pdf文档
2. png或svg格式图片
3. 打印机输出图纸

### 系统架构

从功能上分，打印模块可以分为以下几大模块：

* 打印设置：

提供打印设置界面，以及保存和读取用户设置；提供获取打印设置参数的接口，以便其它模块获取当前打印设置参数.

* 打印管理模块：

主要负责管理、协调各个模块，如获取打印设置参数，调用打印模块的接口打印图标、打印测井视图、打印折纸线等.

* 打印模块：

主要负责实际打印功能，包括打印输出为pdf文档，png或svg等格式图片，驱动打印机输出图纸.

另外，打印功能还需要已存在的canvas模块的功能，关于canvas模块，请参阅相关设计文档.

程序模块结构图如下：



### 接口

enum PrintMode

{

Print\_Mode\_Normal = 0，

Print\_Mode\_Realtime，

Print\_Mode\_Invalide = 100

};

class IPrinterManager

{

virtual void *begin*(PrintMode mode) = 0;

virtual void *end*() = 0;

virtual void *print*(const QGraphicsScene \*scene，

const QRect &rect = QRect()) = 0;

};

## 实时打印

### 功能描述

在实时打印的场景下，当ACME3.0程序连接主控程序成功并接收到实时数据后，用户可连接滚筒打印机，并将实时数据在打印机上实时打印出来，打印内容与绘图区打印输出相同，带图头图尾，打印尺寸按照比例尺保持物理值不变（即与屏幕显示的尺寸大小一致）.

典型用例（ 以下用例皆为ACME3.0接收到实时显示数据并开始实时显示的前提下 ）：

* 用例1：用户点击菜单项：“打印设置”，弹出对话框可设置以下参数：是否双边打印道头，打印曲线时自动打印图头，打印曲线颜色，是否打印速度缺口等.
* 用例2：用户点击工具栏按钮：“实时打印”，打印机实时滚动打印开始，打印图头，道头，实时曲线.
* 用例3：在用例2的情况下，用户点击工具栏按钮：“停止实时打印”，打印机停止打印实时曲线，根据设置，打印道头，图尾

### 性能描述

* 网络传输

目前的数据传输速率一般在512 kbps以下，设计为 <= 4M bps，以满足未来的扩展需求.

* 实时打印速度

打印实时曲线速度由实时数据驱动， 打印的实时数据延迟一般不超过30秒， 最大不超过60秒.

### 输入项

来自于网络的实时测井数据(或重测井数据)

### 输出项

ACME3.0程序：实时数据显示，根据配置的不同，显示不同的曲线实时滚动波形，反映当前测井的实时数据情况.

### 系统架构

从功能上，可分解为以下5大模块：

1. 实时数据收发

负责连接服务器，发送命令和接收实时数据，接收到的数据放到实时数据缓冲中， 每一个实时数据收发实例管理一个实时数据缓冲.

数据的收发单独放到一个线程里处理.

1. 实时数据缓冲

用来存储实时数据，并向实时显示绘制提供所需要的显示数据，需要管理缓冲大小，读写指针的循环移动，并保证读指针不会超过写指针.允许多线程同时访问缓冲，保证线程安全.

1. 实时数据解析及处理

接收到实时模块的数据后，解析数据，并存储到实时数据缓冲中.

数据解析、存储在一个单独的线程中运行.

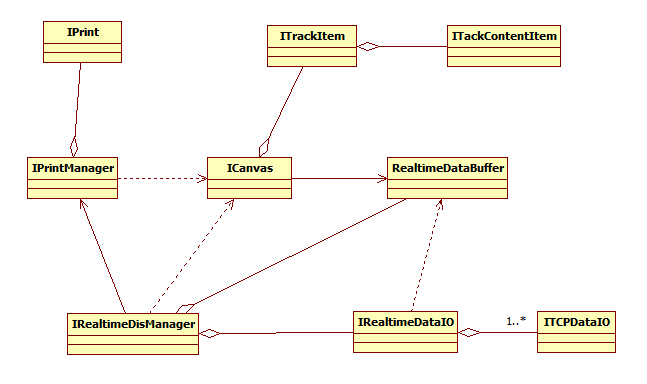
1. 实时打印控制模块

设置、保存打印参数，控制打印模式，范围，根据不同的设置进行不同的内容打印，并控制实时和非实时打印

1. 打印模块

与系统打印模块交互，负责打印机的设置，纸张选择，页边距设置等，开放打印接口供打印控制模块使用.

模块关系图如下：



### 流程逻辑

* 实时数据收发及解析处理流程请参阅文档《详细设计\_实时打印》.
* IRealtimeDisManager在调用ICanvas完成实时显示后，判断当前数据是否需要实时打印，如果需要的话，计算出需要实时打印的显示区域，将此区域传递给IPrintManager，调用IPrintManager的实时打印接口.
* IPrintManager根据打印设置，打开所需要的打印设备，然后构造对应的QPainter， 将QPainter作为参数，调用ICanvas的render函数.
* ICanvas在render函数里，会调用各个图元Item进行重绘 ，各个Item在重绘的时候，需要判断当前Paint device是否是打印机，如果是打印机，缩放比例需要按照打印机页面分辨率进行调整.
* IPrintManager除了打印实时曲线以外，还要按照打印设置，打印图头，图尾等图表元素，IPrintManager需要自己构造相应的ICanvas以及图元进行相应的打印

### 关键技术设计

技术难点：

实时打印

系统所使用的打印机都是滚筒式打印机，所打印的图形都是连续走纸打印输出.而默认的打印方式是设置好纸张大小（譬如 A0 – A5 ）， 然后一次性向打印机输出打印图形，如果超出一页，则控制打印机进行分页处理即可.

因为实时打印需要在现在打印的机制上增加一些额外的控制手段来实现.

1. 控制自定义页大小，让一小段数据作为一页数据向打印机输出
2. 通过Printer.end()函数结束当前页数据的打印并让打印机走纸
3. 每次输出的时候，在调用scene.render(QPainter\*， QRectF targetRect， QRectFsourceRect )函数时，需要根据当前的实时输出进度，调整targetRect和sourceRect的位置.sourceRect来源于当前实时显示刷新的矩形，targetRect则是根据打印页的大小来确定，不需要作页数的累计，对打印机来说，每次总是只输出一页数据，并滚动对应的纸张长度.

### 接口

主要模块相关接口设计如下：

Class IRealtimeDisManager

{

public:

void enableRealtimePrint( bool enable );

int getRealtimePrintStatus();

};

class IPrintManager

{

public:

void setRealtimePageSize( QSize size );

void startRealtimePrint();

void endRealtimePrint();

void realtimePrint();

void setHeadItem();

void setTailItem();

void configure();

};

class IPrint

{

public：

void initial();

void preparePrint();

QPrinter\* getPrinter();

void setPrinter( QPrinter\* printer );

void printeOut( Painter\* );

void endPrint();

};

### 限制条件

打印机属于慢速设备，实时数据速率不能远远大于数据打印速度，否则会造成打印缓冲溢出.

# 二次开发

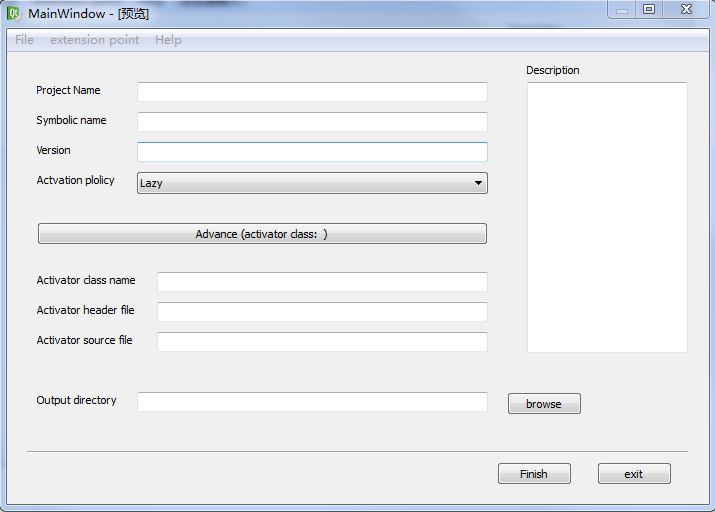
## 开发向导

### 功能描述

1. 输入信息：提供简单的界面给用户设置插件相关信息，如project name， symbolic name， version， activation policy等.
2. 插件工程生成：根据用户输入信息（project name， symbolic name， version， activation policy…）创建基于Qt Creator的插件工程文件，包括.pro 文件， MANIFIEST.MF文件，.qrc 文件.
3. 简化实现扩展点程序： 列出当前系统所有插件扩展点供用户选择，界面会显示该扩展点的简单描述，当用户选中某个扩展点后，界面会列出实现该扩展点需要的所有设置，包括实现指定接口、接收事件信息、发送事件信息等.
4. 代码生成： 生成基于ctk 插件所必须的基础代码，包括Activator class的定义和简单实现；如果需要实现其它插件提供的扩展点接口，生成继承该接口的类的简单实现代码.
5. plug.xml文件生成：根据用户设置信息，为当前插件生成plug.xml文件.
6. 代码license信息设置： 用户可以设置代码文件中默认license信息，从而在开发向导中使用.

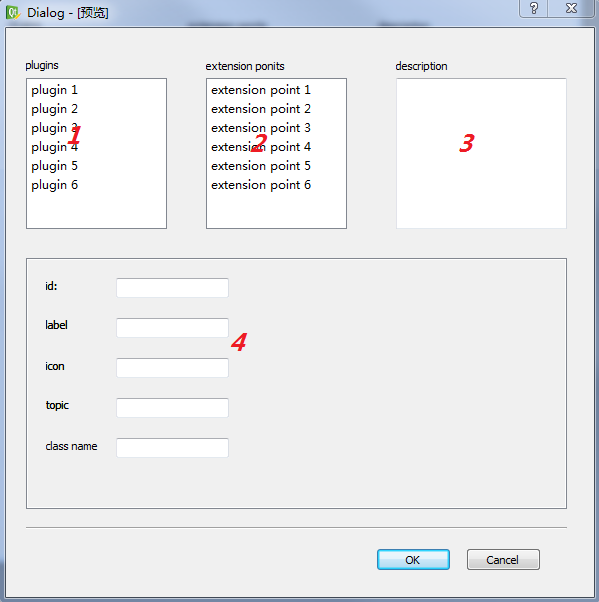
### 界面UI

#### 程序主界面如下图所示：



其中，description是对当前输入项的简单描述，activation policy是插件激活策略，activator class name是生成代码中，activator class 的名字， 而activator header file和activator source file分别为activator class 的头文件和实现文件，output directory是指生成的工程存放的路径.

#### 扩展点实现界面：



1. 区域1是一个list，列出当前系统中所有提供扩展的插件。
2. 区域2列出当前选中插件提供的所有扩展点（一个插件完全可以有不止一个扩展点）。
3. 区域3中会显示当前选中扩展点的简单描述，如这个扩展点用来干什么么，实现这个扩展点需要做哪些事情等，区域3中还会显示当前正在输入项的简单描述，从而让用户知道这个输入项是干什么用的。
4. 区域4中显示的内容可能会根据扩展点的不同而不同，4中目前的内容只是一个示例，表示需要用户输入扩展点的id，label，icon等信息，topic表示实现这个扩展点需要接收ctkEvent，用户需要接收这个event的topic， class name表示要实现这个扩展点，用户需要继承接口，class name就是派生类的名字。

### 典型功能用例

1. 用户输入插件信息完成后，点击finish，开发向导生成基于ctk的插件工程.
2. 用户点击菜单中license项，弹出对话框，用户可以设置代码中使用的license声明

### 输入项

1. 用户输入
2. 当前系统中的插件信息

### 输出项

1. 工程文件(\*.pro)
2. MANIFEST.MF文件
3. 资源文件(\*.qrc)
4. plug.xml
5. 简单的程序代码

### 基础架构

从功能上分，打印模块可以分为以下几大模块：

* 界面显示：

提供简单的界面，从而方便用户设置插件参数.

* 代码生成(CodeGenerator)：

根据用户设置，生成插件必须要的代码，其中代码生成模块又分为Activator生成器和扩展点代码生成器.

* 扩展点发现模块(ExtensionPointFinder)：

主要负责查询当前系统中存在的插件，以及插件的扩展点信息.

* xml文件生成器(XmlGenerator)：

主要负责生成plug.xml文件.

* ProGenerator/ResourceGenerator:

ResourceGenerator主要负责资源文件的生成，包括MANIFEST.MF文件，ProGenerator主要负责工程文件的生成和相关设置.

程序模块结构图如下：

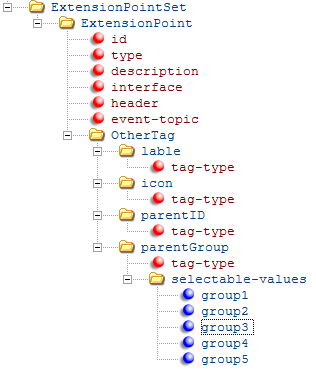


### 扩展点定义功能

如果插件定义一个扩展点，需要通过扩展点定义文件（xml格式文件），指定这个扩展点必要信息，在程序运行的时候，这个文件不是必须的，这些信息主要是给插件开发者使用的，其它插件如果要实现这个扩展点，就需要根据扩展点定义文件中的内容来实现一些指定的接口/事件接收/事件发送等操作.对于扩展点定义来说，有些信息是每一个扩展点都必须有的，比如扩展点id，而有些则是特定扩展点根据自身需要而定制的，比如label/icon等.

开发向导会读取这个文件的内容，然后根据内容在开发向导界面中显示需要用户输入的项，让用户以更直观的方式输入扩展点实现信息，并生成plug.xml文件和简单的扩展点实现代码.

扩展点定义文件具体定义如下图：



扩展点定义文件是一个xml格式多的文件，其中各个标签标示的含义如下：

EextensionPointSet: 扩展点定义集合，它的每一个子标签表示一个扩展点定义.

type: 表示扩展点的类型，目前系统中主要有两种扩展点，一是像MainWindow提供的Menu扩展点那样的，MainWindow自身会添加菜单，并在菜单项被点击时发送信号给扩展点实现插件，所以只需要扩展点实现插件指定接收的事件主题，并接收事件就可以了；另外一种是需要扩展点实现插件实现一个指定的接口，即扩展点定义插件定义一个纯虚接口，扩展点实现插件实现这个接口，并以注册为service的方式让扩展点定义插件使用，这种扩展点定义参看道/线性道/对数道/深度道的实现方式.

description: 对于扩展点的简单描述.

interface: 对于需要实现指定接口的扩展点，interface表示需要实现的接口名.

header: interface 定义所在的头文件.

event-topic: 如果扩展点定义插件需要接收来自扩展点实现插件的事件，event-topic是接收事件的主题，扩展点实现插件可以使用这个主题发布事件.

OtherTag: 表示一些插件所特有的需要扩展点实现插件注意的信息，并不是每个插件都会有这些项.

label: 像MainWindow定义的Menu扩展点，就需要扩展实现插件提供一个label，表示在菜单中显示的名字.

tag-type: 这个标签对扩展点实现插件和扩展点定义插件都没有任何作用，这个标签是给开发向导使用的，开发向导可以根据这个标签的内容决定其对应项在开发向导界面中显示成什么样子，是显示成输入框，还是显示成组合框等.

icon: 图标，MainWindow定义的Menu扩展点中，菜单项可以显示图标，icon就是指这个图标.

parentId/parentGrope: 和前面的label/icon一样，只是举的例子而已，表示扩展点实现需要的一些信息.

selectable-values: 如果需要在开发向导界面显示为combox，需要题库可以被选择的值，selectable-values的子项就表示可以被选择的值.

### 接口

enum Extension\_Point\_Item\_Type

{

INVALIDE\_TYPE，

INPUT\_TYPE，

SELECTABLE\_TYPE

};

struct ExtensionPointItem

{

QString tagName;

QString desciption;

QStringList selectableValues;

int type;

};

struct ExtensionPoint

{

QString id;

QHash<QString， ExtensionPointItem> contents;

};

class XmlGenerator : public QObject

{

Q\_OBJECT

public:

explicit XmlGenerator(QObject \*parent = 0);

void setSymbolicName(const QString &sybolic);

void addExtentsionPoint(ExtensionPoint \* expt);

void addTranslations(const QStringList &trans);

void execute();

};

class ExtensionPointFinder : public QObject

{

Q\_OBJECT

public:

explicit ExtensionPointFinder(const QString &pluginPath， QObject \*parent = 0);

QVector<ExtensionPoint> getAllExtensionPoints();

QVector<ExtensionPoint> getExtensionPoints(const QString &symbolic);

void initialize();

};

class CodeGenerator : public QObject

{

Q\_OBJECT

public:

explicit CodeGenerator(QObject \*parent = 0);

void setClassName(const QString &name);

void setHeaderName(const QString &name);

void setSourceName(const QString &name);

void setInterfaceName(const QString &name);

void setInterfaceFile(const QString &name);

void setLicence(const QString &name);

void execute();

};

class ProGenerator : public QObject

{

Q\_OBJECT

public:

explicit ProGenerator(QObject \*parent = 0);

void setHeaderFiles(const QStringList &headers);

void setSourceFiles(const QStringList &sources);

void setResourceFile(const QString &res);

void setSymbolicName(const QString &symbolic);

void execute();

};

## 插件授权管理工具

### 概述

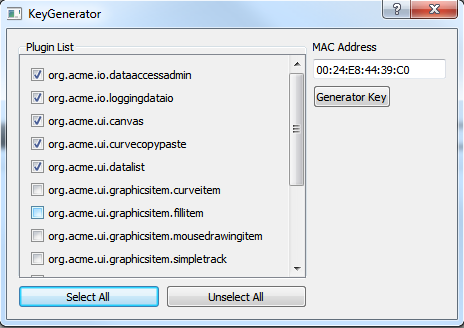
插件授权管理工具，是一个独立开发的工具项目，其目的是便于用户对二次开发插件，新开发插件在商业性上进行便捷的快速管理，以适应于不同用户对于不同功能插件的合理购买和管理.

### 基础设计

根据用于希望授权的插件和要运行的目标机的MAC地址.点击界面上的“Generator Key”， 程序会使用Sha-1 算法生成一个名称是license.key的文件，服务商可以把这个文件发送给最终用户， 最终用户把该文件放在和acmed.exe同一级别的目录下，以保证启动系统时能够找到该文件.

当用户启动ACME3.0，系统内的授权检查机制就会根据本地运行环境与授权license key文件进行比对，如果完全一致，正常启动，否则，未授权插件不能成功启动（授权插件还是正常启动）

### 界面UI



图：UI示意图