第一章 绪论

1.1课题研究背景

随着工业规模的不断扩大，对于工业厂房的控制技术也变得越来越复杂。期初，各种控制面板分布到各个工业现场，需要大量的人力资源来维护这些相互独立的面板，但是这样不能对整个工业生产过程有一个全局的概览。随后人们发现，可以手动记录每个面板相关的生产信息，然后将其传输到控制中心，这样就可以让控制中心可以很容易的了解到各个底层生产间的状况，同时可以形成一个整体了解，然后做出相应的抉择。这种控制方式一般包含一个控制中心，多个分布在各处的控制点。控制点负责采集现场的实时生产情况，控制中心负责结合各个控制点反馈的实际数据，然后做出对应的处理。这种架构是很好的，但是存在着一个问题，那就是需要大量的人力资源，每个控制点将会配置一个以上的工人，负责记录数据以及执行相关的命令，同时还要周期地向控制中心上报数据。这样会导致两个问题，一是不能及时的将现场数据反馈给控制中心，使其及时地做出决定，这样就会使得有些紧急事件无法被恰当适时地处理，可能会造成整个生产流程的中断，从而导致巨大的损失。同时由于控制中心的工作人员会由于分析的不到位，导致不能得到有效的结果，因此可能会下达错误的命令。再则由于命令是由控制中心下达的，各个控制点可能不能及时的收到命令，因此造成命令不能按时执行，也有可能中断整个生产过程；二是在整个控制过程中，需要大量的人力资源，将会使得整个生产成本增加，同时由于人工操作会带来很多不确定性，因此大大影响了整个控制系统的可靠性和稳定性。

随着通信协议的不断成熟以及图形显示技术的发展，上述问题得到了解决。人们为每个控制点设计一套独立的控制系统，让其可以很好的监视生产状况，同时记录分析整个现场数据，并且可以独立的执行相关的命令。当然整个控制系统中也存在一个控制中心，负责监视整个生产过程。但是此时控制点和控制中心的数据交换已经不再通过人力进行，而是通过一些可靠的通信协议进行传输。由于图形显示技术的发展，以及计算机计算能力的提升，设计一套复杂的算法来对整个控制流程进行控制以及显示就变得现实和正确了。根据不同的工业的控制需求设计对应的控制算法，使得整个控制过程变得简洁可靠。可以看出这一次进步使得整个工业控制消除了上面提到的两个缺陷，同时还使得系统变得更可靠更安全。因为，由于不是由操作人员来进行数据传输，这样就使得生产情况可以及时的传达到控制中心以及控制命令可以迅速的发送到各个控制点；同时由于设计了高效的控制算法，使得数据可以预先得到有效的处理，这样决策人员就可以根据预处理出的结果做出更加精准的决定。

控制技术经过不断发展，整个工业控制已经达到了高度自动化。人们也总结出了整个工业控制的层次，这样更有利于对每个层次设计出高效的算法以及解决方案，同时也使得功能独立，降低了耦合性。一般工业控制操作分为五个层次：

* 工业现场(Field Level)：包含各种现场设备，包括温度传感器，各种终端控制器等。
* 直接控制层(Direct Control)：包含各种工业化的I/O模块，一般都带有自己的处理器。
* 工厂监控层(Plant Supervisory)：包含一台或多台监控计算机，它们负责从各个控制点收集数据，然后为监控人员展示整个生产状况的画面，同时将监控人员的命令下达到相应的控制点。
* 产品控制层(Production Control)：目标是对整个产品生产的全局监控，以及做出相关上层决策。
* 生产调度层(Production Scheduling)：对多条产品线进行安排。

其中工厂监控层是比较复杂的一层，主要的软件部分和计算部分都在这一层，因此是研究探索的核心。国外将其称为SCADA(Supervisory control and data acquisition)，国内对应地称作组态软件。其实这是一种控制系统架构，它利用计算机技术，网络数据交换协议和高层的图形化界面进行监控和管理，使得下层的可编程逻辑控制器的相关的终端控制器对生产现场进行直接控制。通过SCADA监控计算机系统，操作人员可以通过图形化操作界面来监视生产情况，同时还可以通过它向下层控制器设置相应参数，达到控制的目的。因此SCADA的核心理念就是为了找到一种通用的方法，使得能够和分布在各个地方的终端控制器通信，达到控制的目的。为此，随着科技的不断的发展，SCADA的结构也在不断的变化。到目前为止总共经历了四代变化：

* 集中式时代：早期的SCADA系统的计算任务是交给小型机来完成的，由于当时的通用网络服务还没有建立起来，因此只能用一些使用环境比较受限的通信协议，从而使得SCADA系统无法与外界或远程的系统交互，只能相对独立地工作。而且此时的SCADA系统一般会配置一台主机来进行数据备份，以防发生意外导致信息丢失，因此相应的系统也产生了，如数字设备公司(Digital Equipment Corporation)生产的PDP-11系列。
* 分布式时代：SCADA系统将信息和命令分发到各个站点，这些站点通过局域网连接起来，因此可以达到近似的实时性。每个站点将会负责与各自相关的特殊任务，与第一代SCADA系统相比将节省不少开支。但是由于网络协议还没有普及和标准化，局域网所用的传输协议是私有的，除协议的开发人员之外，很少有人能够正确的使用这些协议，因此会为SCADA系统带来一些安全隐患。
* 网络时代：相对于分布式时代，网络时代提供了功能完善的网络通信协议，使得其信息交互更加高效更加安全。而且对于那些跨度很大的SCADA系统，只有网络协议才能更好的满足其需求。同时将网络加入到SCADA系统中，可以使得几个平行的SCADA系统都能够进行交互，从而让控制范围更加大。
* 物联网时代：随着云计算技术的成熟，一些企业开始提供商业的云计算服务，SCADA系统也开始采用物联网的相关技术来缩减成本，增加系统的可整合性。因此使得SCADA系统可以近似实时的报告目前的状态，同时由于云服务环境的提供了良好的水平伸缩性，在这种环境下实现较为复杂的算法也变得更加容易。早先，SCADA系统的数据收集对于终端设备是采取一对一的映射的，后来发现这种方法比较笨拙，因此提出了数据模型这种解决方案。在数据模型中，每种设备在SCADA软件中都有与之相对应的虚拟表示，这种虚拟表示不仅可以反映设备的地址映射情况，而且还可以携带其它相关信息，那将非常有利于SCADA系统对设备的有效管理。

1.2国内外研究现状

1.2.1组态软件简介

上文简要地介绍了SCADA系统，国内将其称为组态软件，两者在软件结构上是保持一致的，因此下文将统一使用组态软件来表示两者。组态软件拥有很好的伸缩性，小到只有几十个控制点的系统，大到上千控制点的系统，都可以通过组态软件进行组态，而且往往能够得到很好的成本效益。组态软件一般用在工业领域（如工业生产，过程控制，能源控制，加工等），基础设施领域（如水利，物流，灌溉系统，油气管道控制，电力传输等）和一些基于大型设施的处理过程（如楼宇，机场，船舶，太空站等）。

组态软件一般由以下几个部分组成：

1. **监控计算机：**这是整个组态软件系统的核心部分，涉及到计算机，负责与现场控制器通信的软件以及一个运行在操作台的人机交互软件。它汇集来自多个处理过程的数据，并且还要负责发送控制命令到各个现场设备。在稍小一些的组态软件运行系统中，监控计算机可能就只有一台PC机，人机交互界面是其中的一部分。而在稍大一些的组态软件运行系统中，整个监控中心可能会包含多个有显示界面的计算机，多台负责采集数据的服务器，一套分布式软件以及灾难恢复服务器。同时为了增强系统的可靠性，这些服务器还需要支持双冗余和热备份，从而可以使得在意外发生时，系统还能恢复控制与监视。
2. **远程终端单元：**也称作RTU(Remote terminal unit)，往下与各种传感器相连，往上通过网络与监控计算机相连。RTU作为一种智能I/O，往往具有独立的控制功能，如利用梯形图来完成布尔逻辑操作。
3. **可编程逻辑控制器：**即PLC(Programmable Logic Controller)，与RTU相同，往下与各种传感器相连，往上通过网络与监控计算机相连。但相比RTU，PLC具有更强的控制能力，而且支持一种或多种IEC 61131-3编程语言对其编程。由于PLC相较于RTU更加经济，更加通用，扩展性更好而且支持可配置，所以目前使用RTU的地方都被PLC替换掉了。
4. **通信设施：**即用于监控计算机系统和RTUs或PLCs之间的通讯模块，通常使用的是工业标准协议或制造商专有协议。通过监控计算机给出的最近的一条命令，RTUs或PLCs能够自主地完成控制操作，而且相当接近实时状态。同时通信设施还要保证，通信连接的部分损坏并不会影响整个工业生产过程的控制，而且一旦通信恢复，操作人员将能够继续进行监视和控制。此外一些关键的系统还会采用双冗余数据高速通路，即通过分离的路由器进行数据传输，从而保证通信的可靠性。
5. **人机界面：**即HMI(Human-Machine Interface)，监控系统的操作窗口。人机界面通过虚拟图表，警告页面，事件日志页面以及趋势图等构成。利用采集回来的数据，然后改变各种虚拟图表以及其他页面的状态，从而形成一个工业现场的概览图。一般人机界面是为操作人员准备的，其目的就是收集外部设备的数据，形成报告，发出相关警告，发送相关通知，从而达到一种集中控制。虚拟图表是由一些代表与处理过程相关的元素的线或符号组成的，往往随着数据的变化，形成一种动画的效果。对整个工业控制过程的操作也是通过人机界面完成的，一般设计鼠标操作，键盘操作或屏幕触摸。比如存在这样的一个例子，使用组态软件设计了一个抽水泵控制的系统，整个人机界面采用动画的形式展现这一过程，期间操作人员可以通过人机界面上的开关按钮来控制水泵的开关，当用鼠标点击关闭按钮，此时界面将显示出水泵关闭的效果，即水流量减少，实时地反馈到界面上。可以看出整个人机界面主要包含了两部分：现场情况反映，通过实时采集现场设备的数据，然后利用形象的动画，将其生动的展现给监控人员；控制现场，通过人机界面上的一些虚拟按钮，操作人员可以采取点击的操作，然后形成相应的命令，最后通过互联网络将其发送到工业现场，达到控制的目的。此外，人机界面还将为用户提供历史数据的浏览功能。
6. **报警处理：**是组态软件中比较重要的一部分。系统监视器将会不断地检测报警条件是否满足，而且还要决定何时可以触发一个报警事件。一旦一个报警事件被侦测到了，系统将会采取行动（如触发一个或多个报警指示器，邮件或短信通知远程管理人员等）。在多数情况下，组态软件运行系统的操作人员可以肯定某些报警事件的存在，但这也只能关闭部分报警指示器，一定要报警条件被清除，这时所有的报警指示器才可以关闭。报警条件一般分为两种：明确的报警条件，如报警条件是某个变量达到了某一确定数值，或通过一系列计算，得出了一个具体数值，从而满足了报警条件，其核心就是报警条件是非常明确的；不明确的报警条件，如某一数值达到某一范围，即会触发相应报警，此时报警条件是限制在某一范围的。报警指示器的目的是为了吸引操作人员的注意，以便使其做出恰当的操作，一般包括报警声，在屏幕上弹出一个警示框，或在屏幕相关区域进行特殊颜色的标识。
7. **PLC/RTU编程：**智能RTU或者标准的PLC都能够在没有监控计算机的干预下自主地执行一些简单的逻辑处理。这些设备都能使用标准化的控制编程语言进行编程，从而使其执行相关的处理逻辑。这类不像一些高级的编程语言，需要大量的专业知识才可以进行设计，它们往往语法简单，易于掌握。这就使得组态软件运行系统的工程师可以在RTU或者PLC上实现基础的控制算法，从而简化整个控制逻辑。

1.2.2组态软件发展趋势

如前文所述组态软件的概念源于SCADA，而SCADA主要是国外的产物，因此组态软件的发展也主要集中在国外，目前市场上比较知名的国外组态软件有Wonderware公司的InTouch，西门子公司的WinCC和Intellution公司的iFIX等；随着国内工控技术的累积与发展，同时借鉴了国外知名的组态软件，也出现了一批优秀的组态软件，如北京亚控公司的组态王，大庆三维公司的力控。国内的组态软件主要的优势在于本土化做得很好，同时积极吸取国外组态软件的优点，也使得自己的市场竞争力得到提升。

组态软件的发展主要体现在以下两个个方面：

* 运行平台：早期由于DOS系统比较盛行，组态软件主要运行在此操作系统下，由于此时的处理器大多是16位的，因此对应的组态软件也是16位的，这就大大限制了组态软件的功能，不能提供生动的图画显示，不能承载过多的计算，不能运行复杂的算法等等，同时所控制的设备数也受限制。随后随着Windows系列系统的诞生，处理器的升级，组态软件也朝着功能丰富化的方向发展，组态软件可以运行复杂的动画，可以模拟复杂的器件，可以实现复杂的控制算法，同时控制能力也得到大大提升。今后组态软件的发展是朝着多平台的方向发展的，要求不仅能够在传统的PC平台上运行，还要能够在移动设备上运行。
* 通信环境：起初组态软件的通信协议只是局限在一些商用的授权通信协议上，可互联的范围很小，随着网络技术的发展和成熟，组态软件也开始逐渐接纳TCP/IP协议，这就使得组态软件的控制范围可以跨越生产间、跨越工厂。随着云计算技术的发展，为了更好的管理数据，组态软件也开始采用云计算的概念，同时这样也为使用者节省了开支。

由于组态软件从使用一些私有的授权协议转为使用一些标准协议或一些开源的解决方案，同时大力提倡网络化，这虽然使得组态软件运行系统可以更好的互联，但同时也为网络攻击提供了方便之门。组态软件一般用在一些比较大型的设施中，如电力控制系统等，这些设施往往影响到人们生活的方方面面，如果一旦被攻击，将会造成巨大损失。

组态软件系统存在的威胁主要分为两类。一是未授权的得到控制权，通常是人为的或意外被病毒感染。二是通过网络途径攻击，通过网络将病毒植入系统或劫持替换控制命令来达到获取控制权的目的，因为这些控制协议没有使用加密技术。

组态软件的安全问题也成为组态软件研究的一个重要部分，因为组态软件一般在一些基础领域使用，如果发生意外，将会对人们的健康和安全造成影响，因此为此投入更多的研究是值得的。

1.3课题研究意义

通过使用组态软件提供的相关控件，开发人员可以通过拖拉的方式来构造人机界面，然后利用参数配置框来对系统监控流程进行配置，同时加上适当的动画效果可以达到再现整个生成过程的目的，这就是组态软件的使用方式。整个过程中开发人员不会进行大量的编程工作，只需要对生产工艺有深刻的了解就可以开发出带有人机界面的控制系统。在组态软件产生之前，开发一个控制系统，一般是先分析整个控制流程，然后用高级语言进行开发，整个开发周期很长，而且人力物力消耗很大，最关键的还是伸缩性扩展性非常不好。如果遇到需求的变动，可能将会导致整个软件架构的变动，将会带来不能承受的损失。同时，由于每个系统都是独立开发的，有些具有相同功能的模块被多次开发，这样导致费时又费力。组态软件通过将通用功能进行模块化，提供良好的扩展性，使得可以被反复利用，同时由于这些模块都是经过反复验证过的，因此可靠性更高。

通过以上简述，研究组态软件的架构与实现是非常有意义的，这将为相关领域带来更好的效益。

但目前的组态软件也存在相应的问题，这也是本文探索的根基。

1. 价格高：目前组态软件一般是以控制点的数量来进行付费的。即将一个控制系统中需要监控的项目抽象成一个个控制点，然后决定使用哪种付费方式，即使采用这种按需付费的方式也将是难以承受的，一个小型的系统可能将会为此支付数万元，大一些的系统将支付数十万元。这还不包括开发费用，只是授权使用费而已。
2. 功能繁杂：目前的组态软件讲究的是大而全。整个组态软件开发系统很大，包含各种各样的图元、图标、虚拟仪表等，提供多领域的解决方案。由于包含的内容多，因此使用流程也变得复杂，为此需要投入更多的人力进行学习。
3. 平台限制：目前组态软件一般运行在Windows系统上，由于移动设备的不断发展，以及其运算功能的不断增强，希望组态软件能够运行在移动设备或Linux平台的需求越来越强烈。
4. 应用场景单一：目前组态软件主要应用在工业控制领域，基础设施领域和一些大型控制环境中。随着各种智能设备的增加，设备之间的交互也变得很重要了，因此组态软件的控制方式很实用。
5. 可扩展性限制：由于目前的组态软件都是商用产品，对外暴露的接口很少，如果想要对其进行二次开发根本是不可能的。因此这就限制了组态软件在一些特殊领域的使用。

本文主要探讨如何利用QT这个跨平台应用程序开发框架，来开发一款轻量型组态软件。由于QT是基于C++的，并且使用其提供的API进行相关开发，可以保证在不同平台上使用不同的编译器得到相应平台的可执行程序，这样就使得该款组态软件满足了多平台支持的特性。其次，该组态软件的设计主要完成的是其整体框架的构建，其它比较特殊的模块（如与特殊领域相关的一些控制算法）不进行设计，但为其留好了接口，这样方便开发人员可以对该组态软件进行二次开发。同时该组态软件由于主要利用TCP/IP协议进行通信，因此可以将其利用到其他领域，如智能设备互联等。

1.4论文主要内容及结构

本文主要给出一种组态软件的设计方法，并在QT开发平台上将其实现。开发出的组态软件希望能够满足跨平台的工作特性，并且能够在多种应用场合中使用，同时具有良好的可扩展性，能够让使用者进行二次开发。由于时间与工作量的关系，本文将主要描述整体框架的设计与实现，一些与特殊行业相关的功能模块或解决方案并没有进行设计与实现，但是为开发者保留了接口，方便其根据现实需求来实现。

论文包括以下章节：

第一章：绪论。主要描述了组态软件产生的背景，然后介绍组态软件的发展历程，最后描述了目前组态软件存在的问题，然后围绕这些问题描述了本课题的研究意义。

第二章：组态软件的总体框架设计及相关技术研究。主要描述了组态软件的组成，然后讨论了开发实现的工具与技术，接着给出了一个总体设计框架。

第三章：组态软件开发系统的设计与实现。

第四章：组态软件工程文件的设计与实现。

第五章：组态软件运行系统的设计与实现。

第六章：组态软件的测试。

第七章：总结与展望。

第二章 组态软件的总体框架设计及相关技术研究

目前主流的组态软件主要用在一些大型的工业控制场景，但是随着互联网、物联网的发展，一些新兴的领域也开始需要组态软件这样的一类软件，然而由于监控的要求不一样，很多主流组态软件一些必要的功能并不怎么需要，如强实时性、复杂的控制算法、丰富的控制模块和丰富的驱动等。此时的组态软件需要的是轻量型，而不是大而全；需要的是可跨平台工作，而不是局限在Windows平台；需要的是可自定义、可扩展性，而不是局限的设备选择。因此设计一款可跨平台的，轻量型的并且可以高度自定义的组态软件是有必要的，本文将提出一种设计方案并且进行相应实现。

2.1 组态软件的结构

在组态软件出现以前，控制系统一般都会设计人机交互界面，这样可以对监控对象有一个全局的把控。相关的控制算法以及行业相关的工艺要求都需要非常专业的人员才可以进行操作，而对于界面的开发则对开发人员的专业限制比较小。对于一个系统的设计，一般是控制模块与界面是同时进行设计的，因此会造成软件的复杂度、耦合性大大增加，同时导致开发人员不能尽量发挥所长，这样就会使得整个项目的开发周期不断加长。而且即使完成了产品的开发工作，也不一定能够很好的满足现实的需求，因为可能由于工艺的改变，或者控制流程的变化，从而使得原有的设计存在问题，这时可能会对整个项目进行重新设计重新开发，因为整个控制模块和界面模块混在了一起，这将产生巨大的人力物力花费。

为了能够很好的适应现实需求的多样变化，组态软件产生了。组态软件通过将各个功能进行模块化，然后采用如“搭积木”的方式，将需要的各个功能模块加入到工程中，然后进行相应的配置，进行适当的个性化处理，这样就可以生成一个功能满足需求的系统。组态软件通过将各个功能进行模块封装的方式，来使得整个系统的耦合性降低的。一般组态软件会将控制部分和界面部分进行分离，这样就使得控制算法可以设计成一个一个独立的模块，相应的专业开发人员将只关注自己相关部分的设计，从而让设计更加完善以及可靠。同时对比以前，控制系统的控制部分和界面部分是作为一个整体来开发的，算法往往糅合到整个设计之中，这就使得算法的实现根本不能进行复用，这就会导致公司进行重复开发，从而增加了生产成本。而且组态软件通过不断的累积控制算法模块，本身就使得自身的价值不断提升，而且这样还降低了企业的生产成本。同时组态软件还会提供相关行业的一些解决方案，这就使得企业的设计周期大大缩短。



图2-1 组态软件的工作方式

如图2-1所示，组态软件主要分为三个部分：开发系统部分、工程文件和运行系统部分。

开发系统部分主要是系统开发工程师使用，工程师通过组态软件提供的交互式开发系统，通过拖拽、移动、配置和自定义添加功能等方式来对目标系统进行组态。整个开发系统由多个模块组成，如图元模块、变量配置模块、系统配置模块以及查询、报警模块。各个模块提供了相应的各种控件以及参数配置界面，开发工程师选定工程需要的相应模块，然后对其进行参数的配置，从而满足生产工艺的需求。

工程文件部分主要是记录工程师通过开发系统进行开发后生成的各种结果文件，这些文件记录了开发过程中的各种配置参数以及图形显示的位置参数，通过多种文档存储技术将其记录下来，然后供以后修改系统或系统运行时使用。

运行系统部分由交互界面模块、控制模块、通信模块和历史数据处理模块组成。它利用生成的工程文件，然后形成交互式界面，供监控人员使用。运行系统部分的程序逻辑都是一样的，只是通过不同的工程文件来得到差异化的展现，而工程文件又是开发工程师的设计意图，因此最终运行系统所展示的也将是开发工程师的想法。

因此，整个组态软件的工作流程也就非常清晰了。开发工程师通过开发系统，利用其提供的相关功能或者自定义一些功能来表达自己的设计意图，然后开发系统生成相应的工程文件，运行系统利用这些工程文件再现了开发工程师的设计意图，最终得到一个具有人机交互界面的监控系统。同时，如果要对监控系统进行修改，也只是通过开发系统修改了相关工程文件，这样就可以实现在监控系统运行的过程中对监控逻辑进行修改，并且不需要停止监控系统的运行。从而可以看出，整个过程只是通过对相关部件进行组态、配置就完成了（除了需要添加一些自定义的功能外），这也是组态软件存在的意义。

本文的工作也就是要完成开发系统、工程文件组织和运行系统的相关模块的设计与实现，描述如何解决其中遇到的各种难题以及对应的解决方法。

2.2 开发系统的总体框架

开发系统由多个部分组成，而且为用户提供了良好的可交互式界面，用户可以通过相应的指导完成工程设计。本文只是提出一种开发系统总体框架的设计，并且完成了基本模块的实现，因此很多与行业工艺密切相关的模块并没有进行深入探索，但是在整体框架的设计过程中考虑到了这种需求，也预留了相应的可扩展接口，这样就可以方便开发人员根据实际的需求进行相应的功能扩展，从而体现出了该款组态软件轻量型的特点。



图2-2 开发系统组成

如图2-2所示，开发系统主要由四个部分组成：图元模块、变量配置模块、系统配置模块和查询、报警模块。下面将对四个模块进行分别介绍：

（1）图元模块

图元模块主要包含三个功能部分：图形库、图元管理器和画布管理器。

图形库：包含点、线（直线、折线和曲线）、多边形（三角形、矩形）以及圆等基本图形的绘制，同时还有为用户提供文本输入的控件。此外，图形库还包含一些拟物化的图形，比如阀门、开关、电机、按钮、指示灯以及仪表等，这些图形能够生动地再现生产现场的生产过程。不管是基本图形还是拟物化的图形，用户可以根据自己的需求将其进行适当的变形，从而满足工程的视图效果。同时每个图元都可以进行参数配置，与相应的变量进行关联，这样就使得图形能够根据实时的生产数据进行相应的状态改变。而且图元之间还可以进行动画连接，因为在实际生产过程中，生产是一个动态的过程，用动画的方式将其展示出来是合理的。

图元管理器：在组态软件中存在很多图元，因此图元的管理也是一件很复杂的事情。图元管理器首先要将图元进行分类，然后使得用户能够很方便的找到自己需要的图元；其次，图元管理器还要以适当的方式，在恰当的时候为用户在画板上生成图元，并将其显示出来；再者，图元管理器还需要允许用户添加自定义的图元，因此需要一套机制来保证这个功能；最后，图元管理器还要能从动态链接库中将图元导出。

画布管理器：组态软件中开发系统是非常重要的一块，因为它是在与用户打交道，而画布又是开发系统中最重要的一块，这是用户设计思想的预览图。画布管理器需要实时记录目前用户添加的图元，以及每个图元的配置信息。同时画布还需要能够记录用户的操作，这样方便用户能够将设计工作返回到某一状态。最后，画布还需要记录屏幕的尺寸信息，方便在运行系统中能够以适当的比例重现设计画面。

（2）变量配置模块

变量配置模块由两个功能部分组成：普通变量和结构变量。

普通变量：包含整型、浮点型以及布尔型变量。这些都是一些基础型的变量，在整个工程设计中，每个变量都有唯一的标识符，通过这个标识符可以全局的引用它。每个变量都有相应的属性，这是开发人员根据实际需求来确定的。每个变量都需要确定变量名、变量值的来源、变量的类型以及变量的报警属性，为此变量配置将会有专门的对话框进行处理，同时变量配置的信息将有专门的文件进行记录。

结构变量：结构变量是由两个及以上的普通变量构成的，这样的好处在于对功能相同的变量进行统一管理，方便引用。结构变量和普通变量是共用同一套标识符系统的，因此在使用上是统一的。

（3）系统配置模块

系统配置模块包含三部分：监控站点配置、驱动配置和通信方式配置。

监控站点配置：主要对需要进行监控的站点进行相关配置。如站点需要管理的数据、命令操作以及相关的标识符等。

驱动配置：由于每个站点通信格式都不相同，因此需要为此提供相应的驱动，这样才能让监控系统采集到有效的数据。这里的驱动配置只是配置相关驱动程序存放的位置，以及相应的路径信息等。

通信方式配置：每个站点与主机的通信方式有很多（IP/TCP、串口和CAN等），因此为了能够让监控系统识别，从而使用适当的方法，开发人员需要对每个监控点与主机的通信方式进行配置。

（4）查询、报警模块

查询、报警模块主要包含两个功能部分：报警系统和历史数据查询。

报警系统：监控系统主要完成对生产过程的实时监控，因此它会分析采集回来的现场数据，然后做出相应的处理，其中比较重要的一项就是向相应的工作人员发出警报，使其能够迅速处理存在的问题。对于报警系统的设计，关键点在于对数据的处理，而现场数据在组态软件中都被抽象成了变量，因此变量中的一个重要属性就是与是否报警相关的。开发人员可以通过配置相关参数，达到设置报警界限、警报优先级以及警报显示形态等内容。

历史数据查询：在一个监控系统中，操作员往往需要了解过往的生产状况，因此需要获得以往采集的数据，历史数据查询功能部分主要为操作员提供这样一种查询接口，方便其根据相应的条件进行查询，如根据站点、变量、时间、命令发送情况等。

2.3 运行系统的总体框架

运行系统是组态软件工作后的最终成果，开发人员的意图将通过运行系统展现出来。运行系统通过读取工程文件，然后进行相应的操作，最后使得监控系统能够正常工作。运行系统包含了主要的控制流程，通过统一的参数配置来达到完成不同任务的目的。



图2-3 运行系统组成

如图2-3所示，运行系统由四个模块组成：交互界面模块、控制模块、通信模块和历史数据处理模块。下面将依次介绍这四个模块所完成的主要功能：

（1）交互界面模块

交互界面模块主要包含三部分：图元管理器、图形显示刷新和人机交互。

图元加载器：由于在开发过程中，设计者设计的界面最终存储在文件中，因此在运行系统中需要有专门的一个功能模块来加载各个图元，从而形成监视画面；而且由于图元可能比较多，用一个功能模块来管理也降低了程序的耦合度，同时还使得程序的可扩展性增强。

图形显示刷新：上面开发系统中提到过，很多图元需要根据实时数据来改变自己的显示状态，因此图形显示刷新也是一个很重要的部分，良好的设计可以使得程序结构清晰。

人机交互：在HMI(Human-Machine Interface)中，操作人员可能会根据当前的监控结果做一些命令的下发，这就涉及到两个部分的工作。一、生成命令，并且将其发送到相应的生产现场；二、命令发出或有反馈后，将要及时的在界面上形成相应的动画。同时，操作人员还会希望能够获取历史数据，因此在监控系统中会有与查询相关的交互界面。

（2）控制模块

控制模块包含变量管理器和系统信息管理器两个部分。

变量管理器：上文介绍过变量在组态软件中的重要性，变量是整个组态软件的数据表示核心，通过对它的引用，各个模块可以获得自己关心的实时数据。因此对于变量的一系列操作将会通过一个功能模块来实现，这个模块将负责变量的刷新，保持最新的数据，让各个模块能够向监控人员反馈实时数据；同时该功能模块还要负责最近变量的存储，使得可以快速地获取近期数据。

系统信息管理器：这个功能部分主要是对整个系统的全局配置信息进行管理，比如站点的相关属性，驱动的位置，变量的预处理，变量的解析等。系统信息管理器将这些信息通过适当的数据结构组织起来，提供相应的接口，方便各个模块之间的引用。

（3）通信模块

通信模块由三部分组成：驱动管理器、数据采集器和数据发送器。

驱动管理器：主要对开发者配置的驱动程序进行管理，方便使用的时候进行索引。同时由于驱动大多时候是以动态链接库的形式提供的，因此驱动管理器还要完成驱动加载的工作，利用有效的机制避免由于动态链接库的反复加载和卸载导致的系统性能下降。

数据采集器：主要完成从各个站点采集数据的工作。数据采集器是较高级的抽象层，通过这一层可以屏蔽不同通信方式的差异，从而使得开发者更容易使用，同时这样也使得程序的可扩展性得到提升。

数据发送器：主要完成向各个站点发送数据的工作。其设计思想和数据采集器一样，只不过数据发送器需要根据不同的站点将数据打包好。

（4）历史数据处理模块

历史数据处理模块包含历史数据操作和数据库管理两个部分。

历史数据操作：主要包含查询、删除历史数据。组态软件中有相应的空间可以完成数据库的查询和修改操作。

数据库管理：由于监控软件不仅需要实时了解监控现场的数据，而且有时还要查询以前的数据，因此所有的数据都应该进行存盘，为此需要有专门的数据库来管理。因为数据可能分为几种类型（采集回来的数据、发送出去的数据），为了方便管理需要配置多个数据库，因此需要进行相应的管理。

2.4 工程文件设计

组态软件只是提供给开发人员的一种设计工具，真正体现开发人员的设计成果的内容还是最终生成的工程文件。工程文件作为一座桥梁，将开发系统和运行系统联系其来，开发人员在开发系统上进行设计工作，然后在运行系统上展示设计意图。因此对于工程文件的设计是组态软件中很重要的一部分，良好的结构设计可以使得软件的设计变得简单，同时使得软件的可扩展性得到提升。

组态软件中所有生成的工程文件一般分为两个部分：文件信息和数据。

文件信息主要是对文件做一番描述，使得运行系统能够区分不同类型的文件，从而采取相应的措施。一般包含如下几项：

* 标识符：组态软件生成的文件分为几类，因此需要标识符来确定文件的类型。
* 版本号：版本号表示文件是由哪个版本的组态软件生成的，这样区分是为了防止不同版本之间的交错使用，从而产生错误。
* 幻数：用来识别版本类型，有时可以通过后缀名来识别文件类型，但是后缀名可以修改，因此通过幻数可以唯一确定文件类型，从而使得文件得到正确地处理。
* 创建者、创建时间、修改时间：这些都是工程创建和处理时的一些基本信息，这些信息使得工程能够被很好的管理。

数据部分承载着文件的主要内容，其中通过开发系统进行的各种配置信息都将记录在这里。一般一个文件会包含多个数据项，而每个数据项由两部分组成：标识符和数据。标识符作为这个数据项的唯一标识，数据则是其所带的内容。数据一般包含多个信息点，因此一个信息点一般有三部分确定：类型、长度和数据。类型表示的是有效数据的形态，长度代表的有效数据的大小，而数据则是真正的有效数据。通过这种方式一个数据项可以包含多个数据信息点，这和实际情况是相吻合的，比如在变量配置的过程中，有些变量不需要报警属性，而有些又需要，因此这样就会造成处理的不统一，如果有效数据是变长的，那么这种不统一将可以被避免，这样就可以通过实际的情况对数据进行解析。

组态软件生成的文件包含以下几种类型：

1. 工程索引文件：包含工程的一些基本信息，如项目名称、公司名称等。其次工程索引文件还将给出其它文件的路径信息，通过这些信息可以校验工程的完整性，只有在工程是完整的情况下，才会采取下一步的解析工作。同时工程索引文件还要包含监控软件的一些系统配置信息，如IP地址、端口号等信息。
2. 监控站点配置文件：描述监控软件需要监控的站点的基本信息，由于监控软件一般不止监控一个站点，因此需要对不同的站点进行特殊化的配置。监控站点配置文件还将包含驱动的相关信息，同时还需要记录通信方式。
3. 变量配置文件：组态软件中数据是通过变量来展现的，因此变量配置文件是非常重要的，它记录了每个变量处理操作。
4. 界面显示文件：开发工程师通过在画布上进行画面设计，然后开发系统将整个界面写入文件，然后运行系统通过读取界面显示文件，从而再现开发工程师设计的画面。
5. 数据库配置文件：主要记录整个系统中使用的数据库，以及每个数据库的简要描述。

2.5 相关技术研究

正如本文的题目，本文主要依托的开发工具是Qt。Qt是一个著名的跨平台的C++应用程序开发框架，利用它开发的应用程序可以在多个平台上运行，它提供了丰富的接口，并且都是用户友好型的。很多著名软件都是使用Qt进行开发的，如Autodesk Maya、Google Earth、Skype和WPS Office等。同时Qt也支持很多平台，如基于Linux/Unix(Solaris、AIX、HP-UX等)，基于Apple平台(macOS、iOS)，基于微软平台(Windows系列、Windows CE以及Window RT系列)，甚至于一些嵌入式中的实时操作系统平台也提供了支持，如VxWorks和QNX。

Qt诞生于1991年，由Trolltech（奇趣科技）发布，在2008年诺基亚斥资1.5亿美元收购了Trolltech，将Qt用于Symbian程序的开发。在2012年，诺基亚又将Qt出售给Digia公司，由其继续开发。

组态软件是一款具有人机交互界面，并且还要生成人机交互程序的软件，因此有强大的GUI接口是选择开发工具的重要参考因素。而Qt刚好能够满足这样的要求，它提供了丰富的GUI接口，并且还支持跨平台工作，能够满足这样的条件归结于它的整个库架构。



图2-4 Qt的GUI接口实现策略

如图2-4所示，Qt的GUI接口的实现一共有三种类型，正因为这样构造整个Qt的GUI接口，从而使得通过Qt开发的软件只需要通过很少的适配工作就能在多个平台上进行移植。下面将具体介绍这三种策略：

* **API映射：**所谓映射，就是将操作系统中已经存在的具有同等功能的API进行一次封装，使得在不同的平台上具有相同的接口。例如比较常用的两个操作系统平台Windows和Linux，它们都各自提供了自己平台的相关的GUI接口，比如划线的API，但是两个操作系统提供的参数，或API名称都不一样，但是功能却是一样的，因此Qt对这类API进行一次映射，这样就屏蔽了系统差异。
* **API模拟：**每个系统平台都有自己的独特性，能够提供的功能也不尽相同，因此存在这种可能，在某个平台上具有的API，在另外一个平台上没有与之功能相对应的API。这是Qt就会利用基本的API来模拟出这种功能，从而在多个平台上都提供相同的接口。虽然这样的工作量很大，但是一次开发将惠及以后的多次利用，这样的努力是值得的。
* **GUI模拟：**GUI模拟与API模拟的差别在于GUI模拟提供的是多种API共同作用导致的一种结果。也就是说GUI模拟提供的是一种比较复杂的功能，例如一些复杂的控件，对话框等。这种模拟也是相当有必要的，比如常用的文件选择框，在有些平台上就没有提供这样的功能，因此通过GUI模拟就能够达到提供相同接口的目的。

对于组态软件的开发，提供强大的GUI接口是非常必要的，但是也需要很多其他功能的支持，例如网络模块、串口通信模块、蓝牙模块等。Qt提供了很多额外功能的支持，如XML、Bluetooth、Serialport、NFC、D-Bus和OpenGL等库，这样方便了开发者，同时也为开发者省去了移植这些内容的工作。

2.6 本章小结

本章主要介绍了组态软件的整体框架，主要包含三个部分：开发系统、工程文件和运行系统。分别对三个部分进行了整体的介绍，并且设计了整体的框架，为以后的详细设计以及实现奠定了基础。最后本章还对本文的开发工具Qt进行了介绍，详细讲解了为什么选择它进行组态软件开发的理由，同时还介绍了其对跨平台开发支持所采取的一些策略。

第三章 组态软件开发系统的设计与实现

本章主要讨论组态软件开发系统的关键部分的详细设计，以及对应的实现。组态软件的开发系统关键模块包含四部分：图元模块、变量配置模块、系统配置模块和查询报警模块。下文将详细论述这四个部分的设计思想，以及对于实现过程中遇到的关键难点的解决办法。

3.1 图元模块

3.1.1 图元简介

组态软件为用户提供了一种能够快速构建具有人机交互界面的控制系统解决方案，其中人机交互界面是非常重要的一部分。为了得到最佳的用户体验，人机交互界面通常做得逼真生动，因此早期开发人员通常会为对应的项目设计图形，但是这样的图形往往与所设计的系统密切相关，并不能使用在其他相似的项目中，这样就造成了极大的浪费。后来开发工程师总结了一些领域的图形显示效果，从而将每一种显示独立出来，以空间的方式存在，这样就可以在多个项目中复用，从而图元也就诞生了。图元，意思是原子的意思，也就是最小的控制单位，用户可以通过将多个图元组织在一起，从而产生一个有意义的表达。

图元一般具有以下属性：

* 图形：即图元在界面上的显示效果，可以是一些基本的图形，如线、三角形、矩形圆等；也可以是一些拟物化的图样，如阀门、开关、电机等。
* 拖拽：由于图元是作为一个独立的控件，因此最终需要在一个画布上显示，在画布上，图元的位置可以由用户随意决定，用户可以通过拖拽图元将其放在适当的位置。
* 缩放：用户可以根据需要将相应的图元进行放大和缩小。
* 属性配置：每个图元都具有自己的属性，用户可以通过属性配置来设置图元的颜色、动画效果以及变量关联等相关信息。

对于一款组态软件，图元系统的设计非常重要，因为图元是用户对组态软件的最直观的感受，因此组态软件需要有丰富的图元，能够满足用户的各种需求。但是用户的需求是在不断变化的，因此图元系统的设计还要满足可扩展性，用户在组态软件不能满足其需要的情况下可以自己设计图元，并将其添加到组态软件的开发系统中。本文所设计的组态软件由于时间人力的限制，因此图元并不怎么丰富，还在继续完善中，但本文图元系统的设计框架非常利于用户自由地添加图元，因此这也可以弥补图元不丰富这个缺点。

3.1.2 图元的设计与实现

3.1.2.1 图元继承框架设计

上文已经介绍过，图元分为两类：基本图元和拟物化图元。基本图元包含点、线、多边形和圆等。拟物化的图元主要是通过图片来显示现实中的一些设备，如电机、开关、阀门和仪表等。在进行图元设计的时候，需要将两类图元统一起来，方便后续的管理与操作，因此基于C++语言的多态的特性，本文主要采用继承的方式来实现图元。



3-1 图元的继承框架

如图3-1所示，Primitive类继承于Qt框架的QMdiSubWindow类，从而使得Primitive类具有QMdiSubWindow类相关属性和操作，因此只需要对Primitive的相关操作进行重载，那么就能实现相关功能。同时可以看出，不管是基本图元还是拟物化图元，都将继承于Primitive，这就使得在管理这些图元的时候，利用其父类指针就可以设计出统一的接口。

用户对图元的操作主要包含两部分：变形和属性配置。变形意味着用户可以通过拖拉改变图元在画布上的位置，通过拖拽来改变图元的大小。属性配置即是用户对当前图元的个性化配置，以及显示功能上的配置（如颜色、动画等）。通过这两步操作，用户能够构建出自己想要的界面显示效果。

对于变形相关的操作，需要借助Qt的事件系统来实现。对于具有交互界面的系统来说，往往会发生鼠标移动，鼠标点击，键盘输入等操作事件，在Qt中将这些操作事件统一抽象为事件来处理。一旦用户通过鼠标或键盘进行了相关操作，系统将会产生一个事件，Qt的事件系统将会捕捉到事件，然后将其分发给具有焦点的控件，此后会有事件分发器按照事件的类型将其分配到各自的处理函数中去。因此如果需要改变这些事件的默认处理方法，用户只需要重写相关的事件处理函数即可实现。对于变形操作，涉及到的主要是鼠标的操作，因此需要对鼠标事件的处理函数进行重载，对于本组态软件来说，只需要重写Primitive类的相关函数即可，这样由于继承的关系，所有的图元都将具有相同的特性。

属性配置是用户通过图元的属性配置窗口来实现的，对于不同的图元，具有的相关属性也是不一样的，因此每个图元需要具有各自的配置属性框，为此每个图元都需要管理各自的配置信息，但是为了方便后续对图元的统一管理，因此每个图元的配置属性窗口的相关接口都是通过重载虚函数的方法实现的。

图元自身还必须具有两个关键属性：图形显示和存储。图形显示表示的是图元呈现给用户的形态，为了实现每个图元显示出不同的图像，需要Qt系统中的重绘函数。对于储存，由于需要将所有的设计操作都存储为文件，方便后续的修改和使用，因此对于每个图元都将有相应的序列化函数和反序列化函数，来实现图元的存储与再现。

3.1.2.2 图元变形

如上文所述，图元变形包括移动和缩放两部分。这两个动作都需要用户通过鼠标来完成，因此将会涉及到与鼠标事件处理相关的函数。

表3-1 鼠标事件处理函数

|  |  |
| --- | --- |
| **处理函数** | **函数说明** |
| mousePressEvent(QMouseEvent \*event) | 鼠标任意键按下时调用 |
| mouseReleaseEvent(QMouseEvent \*event) | 鼠标按键释放时调用 |
| mouseMoveEvent(QMouseEvent \*event) | 鼠标移动时调用 |

图元变形中会处理三类鼠标事件，分别是鼠标按键按下、鼠标按键释放和鼠标移动。在Qt中，相应的显示控件都有相关的处理函数，如表3-1所示。由于在Qt中大多数与界面显示相关的类都继承于QWidget类，而此表3-1的三个函数在QWidget类中是以虚函数的方式实现的，只是实现了一些默认的处理逻辑，因此如果需要对相应的事件进行特殊处理，只需继承相关的类，然后重写这三个函数即可。下文将描述如何利用这三个函数来实现图元的相应变形。

（1）Qt的坐标系统

Qt的坐标系统分为两类：全局坐标和局部坐标。全局坐标即以屏幕的左上角（Windows平台）为坐标系的原点，局部坐标即以相关控件的左上角为坐标原点，而对于鼠标操作来说，每个鼠标事件将会记录一个全局坐标和局部坐标，全局坐标是唯一的，局部坐标是相对于鼠标光标所在控件的坐标系而言的。



3-2 Qt坐标系统

本节主要讨论鼠标事件对于图元变形的作用，如图3-2所示，在一次鼠标事件中将会涉及到屏幕、画布和图元。上文已经讨论过不管是全局坐标系还是局部坐标系，坐标原点都在控件的左上角，即图3-2中的A、B和C点。如果用户用鼠标在D点进行了点击，Qt为每个鼠标事件提供了两个接口，globalPos()用于获取鼠标点击点在全局坐标系（即以A点为坐标原点的坐标系）中的坐标；pos()函数用于获取点击点在局部坐标系（即以C点为坐标原点的坐标系）中的坐标。对于图元来说，也存在全局坐标和局部坐标（相对于以B点为原点的画布坐标系），Qt也为其提供了两个接口来获得这些坐标，通过调用mapToGlobal(QPoint(0,0))可以将C点（即图元坐标系）映射为全局坐标，通过调用pos()函数可以获得C点在画布坐标系（以B点为坐标原点的坐标系）中的坐标，通过这两个坐标可以计算出该图元的父组件的全局坐标，利用这个全局坐标，能够保证图元在变形和拖拉过程中不会超出画布的范围。

接下来将继续讨论图元变形相关的实现细节。

（2）图元的拖拉和缩放

每个图元都有默认的大小，当其被拖拉到画布上时，在画布上也有默认的位置，但用户可能会根据项目的需要调整图元大小和位置，因此图元的设计中变形是相当重要的一部分。图元的变形分为两种，一是在画布上的位置发生变化，本身的大小和形状并不会发生变化；二是图元的形状发生变化，如拉长、变大等，图元的形状改变一般是通过拖拽图元的某些区域产生的，一般的图形系统中，一个图形可拖拽的区域分为九个部分，如图3-3所示。



3-3 图元拖拽区域分布

在Qt中只需要确认A点的坐标和图形的长(Height)和宽(Width)就可以表示出一个图形，在通过鼠标对图元进行操作的过程中，鼠标只是一个点，因此要确定应该进行那种操作，需要一个区域来标识。由于鼠标的作用范围应该是一个区域而不是一条线或一个点（这也是方便用户操作），一个图元的边界到鼠标作用区域的边界应该存在一定距离，这也叫做填充区(Padding)。如图3-3所示，鼠标的作用区域分为九部分，各部分的作用如表3-2所示。

表3-2 图元拖拽区域介绍

|  |  |  |  |
| --- | --- | --- | --- |
| **编号** | **区域名称** | **功能简介** | **鼠标光标** |
| 0 | TopLeft | 使得图形左上角变形 |  |
| 1 | Top | 使得图形上部变形 |  |
| 2 | TopRight | 使得图形右上角变形 |  |
| 3 | Left | 使得图形左边变形 |  |
| 4 | Inner | 使得图形位置发生变化 |  |
| 5 | Right | 使得图形右边变形 |  |
| 6 | BottomLeft | 使得图形左下角变形 |  |
| 7 | Bottom | 使得图形底部变形 |  |
| 8 | BottomRight | 使得图形右下角变形 |  |

在用户通过鼠标，对表3-2所述的区域进行操作时，各个区域对应的处理方式如下：

* TopLeft区域：如图3-3所示，此时D点将保持不动，A点将会被重新计算，图元的宽(Width)和高(Height)也将被重新计算。当鼠标移动，系统将利用A点坐标、图元宽和高，重新绘制图元，使得图元改变形状。
* Top区域：如图3-3所示，此时C点和D点将保持不动，A点坐标将会被重新计算，图元的高(Height)也将会被重新计算。当鼠标移动，系统根据新的A点坐标和宽高，重新绘制图元。
* TopRight区域：如图3-3所示，此时C点将保持不动，A点将会被重新计算，图元的宽(Width)和高(Height)也将被重新计算。当鼠标移动，系统根据新的A点坐标和宽高，重新绘制图元。
* Left区域：如图3-3所示，此时B点和D点将保持不动，A点坐标将会被重新计算，图元的宽(Width)也将会被重新计算。当鼠标移动，系统根据新的A点坐标和宽高，重新绘制图元。
* Inner区域：如图3-3所示，如果鼠标是在这个区域，代表着操作者是想改变图元在画布中的位置，而图元的形状将保持不变。
* Right区域：如图3-3所示，此时A点和C点将保持不动，图元的宽(Width)也将会被重新计算。当鼠标移动，系统根据新的宽，重新绘制图元。
* BottomLeft区域：如图3-3所示，此时B点将保持不动，A点将会被重新计算，图元的宽(Width)和高(Height)也将被重新计算。当鼠标移动，系统将利用A点坐标、图元宽和高，重新绘制图元。
* Bottom区域：如图3-3所示，此时A点和B点将保持不动，图元的高(Height)将会被重新计算。当鼠标移动，系统根据新的高，重新绘制图元。
* BottomRight区域：如图3-3所示，此时A点将保持不动，图元的宽(Width)和高(Height)将被重新计算。当鼠标移动，系统将利用图元宽和高，重新绘制图元，使得图元改变形状。

如前文所述，图元的变形主要涉及到鼠标的三个动作：鼠标按键按下、鼠标移动和鼠标按键释放。Qt对于这些鼠标事件的处理分别委托给三个函数：mousePressEvent、mouseMoveEvent和mouseReleaseEvent。通过重写这三个函数，可以实现图元的变形。图元的变形处理流程如图3-4所示，可以看出主要分为三个部分：拖拽区域判断、边界判断和位置计算。

* 拖拽区域判断：用户可以随意的移动鼠标，但只有当鼠标光标移进图元区域的时候才会触发图元相应的鼠标事件处理函数。当鼠标只是移动时，将只会调用mouseMoveEvent函数，在这个函数中将会根据图3-3所介绍的拖拽区域进行判断，记录目前鼠标正处在哪个区域，并将光标变化成表3-2中对应区域的鼠标光标形状（如鼠标在Left区域，将会显示图标），这样可以提醒用户此时正在对哪个区域进行操作。
* 边界判断：当用户在图元区域内按下鼠标左键，此时再移动鼠标，说明用户正在对图元进行变形，但由于图元是处在画布中的，而画布的大小是固定的，因此为了防止图元被移出画布边界，需要对移动的鼠标位置进行计算判断，如果在允许范围内将不做任何修改，如果超出范围将对鼠标位置进行相应修改。由于边界判断问题比较复杂，因此下部分将会进行详细讲解。
* 位置计算：一旦用户按下鼠标左键并且移动鼠标，就会在mouseMoveEvent重新计算图元的A点（如图3-3所示）和图元的宽高，并且对图元进行重绘。由于A点坐标是在画布坐标系中（如图3-2所示），而鼠标事件记录的位置在全局坐标中（即屏幕坐标，如图3-2所示），因此需要通过计算图元的A点的全局坐标与鼠标事件的位置坐标之间的相对距离，然后再作用到A点上（画布坐标系中），这样才能完成图元的变形。



图3-4 图元变形处理流程

（3）边界检查

上一部分讨论过，当用户对图元进行变形的时候，可能会将使得图元超出画布的范围，因此在进行这些操作的时候，需要对移动变形的位置进行判断，这样使得图元能够一直存在画布上。

存在两种情况可能使得图元超出画布的范围：一是移动图元；二是对图元进行变形或缩放。不管是哪种情况，对图元的改变都是通过计算鼠标位置的两次相对位置（鼠标按键按下时的位置和移动后的位置）而确定的，因此在进行边界检查的时候，也只需要对着两个鼠标位置进行处理即可。



图3-5 边界检查示意图

为了保证图元无论进行怎样的变形或移动都要处在画布里，需要弄清最终引起变形的原因。上文已经讨论过，鼠标事件是导致图元变形的原因，变形的各种参数是通过计算两次鼠标的相对位置而确定的，即鼠标按键按下时的位置和鼠标移动过程中的位置。对于鼠标按键按下时的位置是固定不变的（这取决于用户的操作，因为只要用户是在对当前画布的图元进行操作，这时的鼠标位置就是合法的），从而可知只有鼠标移动时的位置是不确定的，可能这个位置就将超出画布的范围，因此通过修正这个位置就可以保证图元不会超出画布的范围。

当需要改变图元形状的时候，也就是当用户通过拖拽图3-3中所标明的九个区域的时候，此时只会影响到图元的边界，如图3-5所示，这时鼠标的位置只是处在图元的四条边和四个点上，这时只需修正鼠标位置，保证其在画布范围内。在Qt中，通过左上角和右下角的坐标就能够确定一个图形的范围，因此如图3-5，只要使得鼠标坐标的X坐标介于A点的X坐标与D点的X坐标之间，鼠标坐标的Y坐标介于A点的Y坐标与D点的Y坐标之间，图元的变形就不会超出画布的范围。

对于图元的移动，情况就比较复杂。图元的移动表示图元的形状不会改变，只是图元在画布上的位置发生变化。以图元向左移动为例，首先用户的操作是点击需要移动的图元，这时可以得到四个参数：点击点距离图元的四边的距离。如图3-5所示，距离上边的距离为Top，距离下边的距离为Bottom，距离左边的距离为Left，距离右边的距离为Right。如果用户往左移动意味着鼠标位置不能超过图3-5所示的虚线所标识的E边，也就是说画布的AB边到E边的距离应该为Left，这样就能保证图元能够在画布内进行完全显示。以此类推，图元的其他三个方向的移动都应该满足这样的条件。当鼠标移动导致位置超出这些范围的时候，只需将鼠标位置进行修改即可，继续上面的例子，如果图元左移超过了E边，那么只需将鼠标坐标的X坐标设置为E边的X坐标，然后再进行图元位置的计算，最后重绘图元，此时的图元一定在画布内。

3.2.2.3 图元的图形显示与属性配置

上一节描述了图元的基本变化（缩放和移动），这些变化是每个图元都具有的属性，因此这些功能都是在Primitive类里实现的，一些具体的图元通过继承Primitive类从而获得了这些功能，但是对于每个具体的图元来说都有自己的独特属性，因此需要根据每个图元具有的功能来完善其设计。每个图元除了一些共有的功能外，还具有两个非常重要的属性：图形显示和属性配置。

图形显示即图元呈现给用户的外观形状，如直线图元呈现给用户的就是一条直线，矩形图元呈现给用户的就是一个矩形。

属性配置则包含两部分：图元自身相关参数的设置和图元的动画连接。自身相关参数包括颜色，是否透明和背景色等；动画连接即图元在整个画面中的运动情况，如图元根据相关的输入值，在画面中做出相应的动作。

下面将分别讨论图元的图形显示和属性配置的相关实现细节。

（1）图元的图形显示

上文讨论过，图元主要分为两个大类：基本图元和拟物化图元。其中基本图元如图3-6所示，主要包含一些常见的二维图形（三角形和多边形等）以及三维图形（正方体和圆柱体等）。这些图形通常可以有两种呈现方式：带填充色的和不带填充色的。



图3-6 基本图元

所谓的拟物化图元指的是图元所呈现出来的图像和现实生活中实际存在的事物很相像，利用这些图元可以模拟现实生活中的相关设备。具体的拟物化图元如图3-7所示，包含气泵、锅炉、房屋和传送带等。



图3-7 拟物化图元

在Qt中，如果要改变其相关类的图形显示需要重写paintEvent函数，本文采用的也是这种方法，从而实现不同图元的差异显示。在计算机图形中有两类图形表示方式：位图和矢量图。

* 位图：位图是由像素点构成的，每个像素点有匹配的存储空间，这些存储空间可以记录图像的颜色、饱和度和明度。因此对于一张图片来说，一旦图片的效果确定下来了，那么这张图片的像素点的个数以及每个像素点所带的信息也就确定下来了。这时如果要改变图形的形状，那么就会导致图形的显示失真，因为此时可能显示屏幕的多个像素点只能显示图片的一个像素点的信息，这样就会有模糊或锯齿的情况发生。
* 矢量图：矢量图是由多个基本对象组合而成的，而每个基本对象又是通过相关的绘制函数实现的，因此只要提供相应的绘制参数，就能够绘制出各个对象，从而组合出复杂的对象。由于矢量图是通过数学函数绘制的，因此在变形的过程中只是改变传入的参数信息，因此不会使得图形失真。

根据上面对位图和矢量图的分析，再结合图元的基本特性（图元可以根据用户的需要进行缩放和变形）。如果要得到良好的显示效果就应该选择使用矢量图来设计图元。SVG(Scalable Vector Graphics)是通用的矢量图格式，通过将矢量图的相关参数信息以XML(Extensible Markup Language)的格式存储起来，从而方便传输和解析，在Qt中专门的类进行处理。

图元图形显示的设计主要分为三步：

* 第一步：使正在设计的图元继承Primitive类，从而使得图元具有变形的相关特性。
* 第二步：选择相应的适量图片，如图3-6和3-7所展示的图片，利用QSvgRenderer类进行处理，其构造函数为QSvgRenderer(const QString & filename)，其中filename参数表示的是图元的存储位置。通过defaultSize()方法可以获得矢量图的默认显示尺寸，利用resize()函数可以重置图元的大小，从而得到默认的图元显示。
* 第三步：重写paintEvent函数，新建一个QPainter实例，然后设置相关属性，调用QSvgRenderer类的render()函数，并将QPainter实例以参数的形式传入，从而完成图元的图形显示。

（2）图元的属性配置

图元的属性包含两部分：自身相关参数和动画连接。自身相关参数包含颜色、是否透明、背景色和大小四部分。而动画连接则包含是否移动、输入输出值、滑动杆参数和特殊情况。其中是否移动表示的是在画面显示中，图元的位置是否会发生变化，如果需要发生变化，则需要配置相关参数；输入输出值表示图元是否会接收外部的输入值或向外部输出数值；滑动杆参数表示的是某些图元具有可移动的滑动杆，因此需要对滑动杆运动的参数进行设置；特殊情况表示的是图元可能会闪烁显示或隐藏。

由于每个图元具有自己的独特性，因此不一定所有的图元都要进行上述讨论的设置，但是考虑到代码的复用性，因此对于配置对话框的设计将采用如下策略：为属性配置对话框设计一个类AttributeSetDialog，该类中将提供所有的属性配置选项，然后通过一系列的接口将所配置的结果传给相关图元；由于该类提供的是所有的属性配置选项，而有些图元的某些选项是不能进行设置的（如有些图元不能接受输入值属性配置），因此为了能够适应不同图元的需要，AttributeSetDialog类将提供相应接口使得某些配置选项不能操作。根据上述讨论给出如下具体设计实现。

属性配置对话框如图3-8所示，可以看出主要分为三部分：对象描述、基本属性和动画连接。其中对象描述中有两个信息需要用户输入，对象名称和提示文本，对象名称指的是用户为这个图元命的名，而提示文本表示的是当鼠标光标处在图元所在区域时出现的提示信息，特别需要注意的是对象名称在整个工程项目中是唯一的。



图3-8 属性配置对话框

基本属性可以对五个参数进行设置：图形颜色、背景颜色、图元的长、图元的宽和图元背景是否透明。用户可以通过点击工具按钮，在弹出的调色板中选择相应颜色，从而设置图形颜色和背景颜色。除了通过拖拽的方式改变图元大小外，用户还可以在基本属性中直接对图元的长宽进行设置，从而改变其形状。对于透明指的是图元的背景是否透明，对比效果如图3-9所示。



图3-9 透明效果对比

动画连接包含四个部分：移动、输入输出值、滑动杆输入和特殊。每个图元可以根据自身需求对这几个部分设置使能还是不使能。其中移动这一块表示的是图元能不能够在画面中进行位置变化，当复选框选中时，这时移动参数才可以进行设置，用户需要对移动的起点和移动的终点坐标进行设置，从而限制了图元移动的范围，其次用户还需要将移动的条件与特定变量进行关联，当变量发生变化时，系统会根据变量的值来决定图元移动的距离。输入输出值部分表示的是图元能够与外界交流的数据，用户可以规定数据的最大最小值，并且用户需要将输入或输出同相应的变量进行关联。滑动杆输入时针对一些特殊的图元（如温度计等），这些图元有一根滑动杆可以指示数值，因此需要外界数据输入促使其变化，用户通过关联相应变量来达到相应的目的，同时用户还可以设置滑动杆的变化范围。特殊部分表示图元在显示过程中是否会发生变化，其中可以设置三种变化：闪烁、隐藏和流动。用户通过关联相关变量，当变量满足某个条件时触发相应效果，通过下拉框可以选择比较条件。

下面将详细介绍AttributeSetDialog类的相关属性和方法，首先需要明确的一点是，AttributeSetDialog类的主要目的是为图元服务，同时也只有图元会使用它，而且每个图元必须有一个AttributeSetDialog类的私有实例。所以为了方便数据的访问，AttributeSetDialog类的成员变量都是公有的，但由于图元中的AttributeSetDialog实例是私有的，因此其成员变量也只能由相关图元才能访问，这也达到了数据封装的效果。AttributeSetDialog类也提供了一些公共方法，使得图元类可以按照自己的需要选择能够使用的属性配置。AttributeSetDialog类详细属性和方法介绍如表3-3所示。

表3-3 AttributeSetDialog类属性和方法介绍

|  |  |  |
| --- | --- | --- |
| **AttributeSetDialog类属性** | | |
| **属性名** | **属性类型** | **描述** |
| name | QString | 对象名称 |
| tips | QString | 提示文本 |
| color | QColor | 图形颜色 |
| background\_color | QColor | 背景颜色 |
| width | int | 长 |
| height | int | 宽 |
| is\_transparent | bool | true为透明，false为不透明 |
| is\_movable | bool | true为可移动，false为不能移动 |
| move\_start | QPoint | 移动位置的起点 |
| move\_end | QPoint | 移动位置的终点 |
| move\_variant | QString | 移动关联变量 |
| input\_out\_flag | int | 0表示输入，1表示输出 |
| input\_out\_max | int | 输入输出的最大值 |
| input\_out\_min | int | 输入输出的最小值 |
| input\_out\_variant | QString | 输入输出关联变量 |
| ··· | ··· | ··· |
| **AttributeSetDialog类方法** | | |
| **方法** | **描述** | |
| void setInputOutDisabled() | 使得输入输出值部分无效 | |
| void setSliderInputDisabled() | 使得滑动杆输入部分无效 | |
| void setSpecialDisabled() | 使得特殊部分无效 | |
| void setBlinkDisabled() | 使得闪烁功能无效 | |
| void setHiddenDisabled() | 使得隐藏功能无效 | |
| void setFlowDisabled() | 使得流动功能无效 | |

3.1.2 图元管理器的设计与实现

图元管理器的功能主要分为两部分：将图元进行分类管理和方便用户添加图元到画布中。对图元分类是为了让用户能够更好的使用图元，而为了使得用户能够方便的将图元加入到画布中，则需要有相应的机制来保证，因此图元管理器的设计也是非常重要的。

为了使得图元能够与整个系统的耦合度降低，因此图元一般以动态链接库的形式存在，这样就可以保证能够独立的更新图元而不改变整个系统的框架。同时为了能够实现图元模块的独立，因此需要利用相应的机制来实现，对于图元的管理需要配置一张图元信息表，格式为序号 标识字符串，序号和标识字符串在整个图元库中都是唯一的，标识字符串是作为索引动态链接库中相关图元的字符串，格式为get+图元名。因此在设计动态链接库的时候，相应的函数设计将满足下列规则，以添加CubePrimitive图元为例，

**第一步：添加获取图元函数**

//返回图元基类的指针，方便统一操作

//函数名格式：get+图元名

//传入参数为父窗口，可以是画布或运行界面（运行系统中）

Primitive\* getCubePrimitive(QWidget \*parent)

{

return new CubePrimitive(parent);

}

**第二步：修改图元信息表**

//并且在图元信息表中添加一条信息

1 getCubePrimitive

//即满足格式：序号 标识字符串

通过这种方法，只要在图元库新添图元的时候，添加这样一个函数并且修改图元信息表，就能够让整个系统识别到添加的图元。

图元管理器的一大功能就是对图元进行分类管理，因此需要对图元进行相应的分类处理，通过下面这个枚举的定义可以看出图元的分类，

typedef enum{

BasicShape, //一些基本的图元

PushButton, //按钮形状的图元

Arrow, //箭头形状的图元

Pipe, //各类管状的图元

Blower, //鼓风机类的图元

Boiler, //锅炉类的图元

Pump, //泵形状的图元

Controller, //各类控制器形状的图元

Motor, //电机形状的图元

Tank, //各种罐子形状的图元

Other, //未分类的图元

PrimitiveTypeSize //图元类型的总数

}PrimitiveType;

每个图元的类型都是由图元设计者确定的，可以通过图元的getType()方法得到图元的类型，这样使得图元管理器就能够很方便的进行管理。在为用户展现图元的时候用到了Qt的QListWidget控件，通过这个控件能够将同一类图元以列表的形式呈现出来。而此控件是通过添加QListWidgetItem来添加数据项的，因此为了能够让每个图元控制自己的展示效果，每个图元都有getListWidgetItem方法，其函数实现细节如下：

QListWidgetItem\* getListWidgetItem()

{

QListWidgetItem \*item = new QListWidgetItem;

item->setIcon(QIcon(“图片路径”));

item->setText(QString(“图元名称”));

return item;

}

该函数主要设置了图元在列表中的显示名称以及显示的图标。

上面讨论了图元管理器的主要实现策略，下面将详述PrimitiveManager类的实现细节。

PrimitiveManager类具有两个重要属性：

1. QMap<int, Primitive \*> primitive\_set：该成员变量用来存储图元库，以图元序号作为关键字，图元实例指针作为值。
2. 各类图元的QListWidget实例：如Pump类型图元的QListWidget实例为pump\_listwidget。通过这些实例可以达到对图元进行分类的目的。

PrimitiveManager类的重要方法：

1. QListWidget\* getListWidgetByType(const PrimitiveType &type):该方法根据type值向调用者返回匹配的QListWidget实例，如type等于Pump，那么就会返回pump\_listwidget，然后调用者就可以根据返回值进行相应的图元列表显示。
2. Primitive\* getPrimitiveById(const int &id):该方法根据id值，即图元的序号，向调用者返回相应的图元指针。该函数将id作为索引查找primitive\_set结构，返回一个图元指针，然后调用图元的clone()方法得到复制图元，再将复制图元的指针返回给调用者。
3. void init(const QString &filename, QWidget \*parent):该函数主要完成加载图元库，然后完成PrimitiveManager类的相关结构的初始化。

由于图元管理器的初始化流程比较复杂，下面将进行详细介绍，如图3-10所示，初始化流程主要分为三个部分：对图元信息表的处理、动态链接库的加载以及提取图元并对其进行相应处理。其关键代码如下：

（1）对图元信息表的处理

//调用checkPrimitiveInfoTable函数，将信息存储在一个map中

//判断校验结果

QMap<int, QString> primitive\_info = checkPrimitiveInfoTable();

if (primitive\_info.isEmpty())

return;

（2）加载动态链接库

//利用Qt的QLibrary类对动态链接库进行处理

//filename为动态链接库的路径

QLibrary primitive\_lib(filename);

（3）提取图元并进行相应处理

//定义提取函数的类型

typedef Primitive\* (\*GetPrimitive) (QWidget \*);

//迭代处理所有图元

QMapIterator<int, QString> it(primitive\_info);

while (it.hasNext())

{

it.next();

int id = it.key();

QString func\_name = it.value();

//提取函数并设置primitive\_set数据结构

GetPrimitive func = primitive\_lib.resolve(func\_name);

if (!func) continue;

primitive\_set[id] = func(parent);

//根据图元类型设置对应的ListWidget

Primitive\* pri = primitive\_set[id];

PrimitiveType type = pri->getType();

QListWidgetItem \*item = pri->getListWidgetItem();

item->setWhatsThis(QString(%1).arg(id));

setListWidget(item, type);

}



图3-10 图元管理器初始化流程

3.1.4 画布管理器的设计与实现

画布管理器需要时刻记录画布的状态以及用户的操作。记录画布的状态即需要记录用户添加到画布上的所有图元，记录用户的操作是为了让用户能够撤销或重做某些步骤，防止误操作带来的影响。为此需要设计一个专门的类来进行这些管理，所以下面将描述CanvosManager类的实现。

为了能够记录用户添加的所有图元，因此CanvosManager类需要有一个数据结构来存储这些图元。QMap<QString, Primitive\*> canvos数据结构可以用来存储图元，其关键字是所添加图元的名称，即在属性配置对话框里的对象名称，可以通过图元的getObjectName()方法来获得，而对应的值则是所添加图元的指针。同时为了保证对象名称的唯一性，CanvosManager类会提供一个公有方法来判断对象名称是否已经存在，即bool isObjectNameValid(const QString &name)，图元在设置属性的时候通过调用此函数判断用户填写的对象名称是否有效，如果有效该函数将会返回true，否则返回false。

用户能够向画布中添加图元，当然也能够从画布中删除图元。为此CanvosManager类提供了deletePrimitiveFromCanvos()方法，通过此方法可以将指定图元从canvos数据结构中删除掉，然后画布再重绘图元，则被删除图元将从画布上消失。

CanvosManager类同时还需要记录用户对图元的操作，因此在需要管理两个栈：撤销栈和重做栈。其示意图如3-11所示，



图3-11 撤销重做示意图

当用户对画布中的图元进行操作的时候，此时的每一步操作都将记录在撤销栈中，如图假设用户已经做了七步操作，那么此时撤销栈中应该有七次操作，这时用户如果撤销了两步操作，这两步操作将会被放入重做栈中，图3-11描述的就是这种情况，在撤销栈中有Operation1-Operation8，重做栈中有Operation7和Operation6。下面将具体描述撤销和重做动作对两个栈的影响。

图3-11中的①路径代表的是撤销动作，假如现在用户希望撤销Operation5，当执行撤销动作时，CanvosManager类会将撤销栈中的Operation5弹出，然后将其压入重做栈中，重做动作即是其逆过程。

图3-11中的②路径代表的是用户执行了一次新操作，假如用户执行了Operation8这个新操作，一旦这个操作完成，那么就会将其压入撤销栈中，同时CanvosManager类将会清空重做栈，这就代表着用户不能再执行重做栈中的操作了。

在CanvosManager类中，是通过两个QList数据结构来实现撤销栈和重做栈的，并且提供了三个方法来管理它们：

1. void insertOperation(Primitive \*primitive)：这个函数完成新操作的插入，参数primitive即是正在操作的图元的克隆。
2. Void undoOperation()：这个函数完成撤销动作，该函数会从撤销栈中弹出一个操作项，然后将其放入canvos数据结构中，并且让画布重绘所有图元。放入canvos数据结构中时存在两种情况，一是canvos中已经有相同对象名称的图元，因此会先删除canvos中的数据，然后再将其放入canvos中；二是canvos中不存在相同对象名称的图元（即代表这一步操作是由删除动作造成的），那么就可以直接放入canvos中。
3. Void redoOperation()：这个函数完成重做动作，重做动作能够实施的前提是重做栈中有数据项存在。调用该函数将会弹出重做栈的一个数据项，然后将其放入canvos数据结构中，处理同undoOperation函数，并且还需要将数据项放入撤销栈中。

3.2 变量配置模块

在组态软件中，数据是使得其各个部分能够运转起来的关键。组态软件的最终产品是监控程序，而使得监控程序能够工作的则是不断从现场采集回来的实时数据，为了能够让组态软件能够识别各种设备产生的数据，因此需要有一定的机制来解析这些原始数据。每一种设备产生的数据是不一样的，而且只有使用者才知道数据中的每一位代表什么含义，所以组态软件不能假定存在某种固定的数据解析方式，而应该提供一种方法，让用户能通过组态软件的开发界面进行配置，系统则通过配置的结果来解析数据。这样的好处在于系统内部不需要考虑各种情况，而是用统一的解决方案来处理，从而提升了系统的可扩展性。

为了解决上述描述的问题，因此提出了变量这种概念。变量描述了一个采集数据点的相关信息，组态软件可以接收不同格式的数据，但是只有当指定了接收的数据应该用哪个变量来解析的时候，系统才可以理解接收到的数据的意义。变量是可以根据系统的需求随意添加的，而且用户还能对变量的相关属性进行设置，从而满足不同的功能需求。变量作为组态软件各个模块之间通信的介质，扮演者非常重要的角色，例如在图元设计一节也描述了图元的动画连接，其中一项属性配置就是变量关联，因此图元的变化最终也体现在数据的变化上。

下面将从普通变量和结构变量这两个方面对变量配置模块进行描述。

3.2.1 普通变量的设计与实现

普通变量即组态软件中最常见的变量，通常一个变量就代表着一个数据采集点。对于普通变量的配置主要分为两部分：基本属性和报警参数。

基本属性包含变量名、对变量的简要描述、变量的类型以及变量的最大最小值。变量名在每个工程项目中都是唯一的，是作为系统识别变量的唯一标识；对变量的简要描述，主要是用户可以通过一些简要的字段来对变量进行简单的功能说明，使得使用者可以很好该变量的设计意图；变量的类型，每一个变量都具有相应的类型（如整形、浮点型、布尔型等），这样才能够让系统理解数据的含义；变量的最大最小值，通过配置变量的最大最小值可以确定该变量的取值范围，当获取的数据值不在这个范围的时候，系统可以进行正确地处理。

报警参数包含对该变量是否需要报警以及如果需要报警，报警级别的配置。由于变量是实际生产数据的反应，因此当数据值达到某些不正常的范围的时候，需要系统告知观察员，当然用户也可以为不同的变量设置不同的报警优先级，优先级越高的报警将会最先显示在监控界面上。

图3-12展示了变量属性配置的对话框，从图中可看出属性配置分为两部分，但只有当基本属性中的报警复选框被选中时，用户才可以操作报警参数相关的参数。对于报警参数，需要配置报警名称以及报警优先级，报警优先级数值越高表示优先级越高；报警限总共分为四个层次：低低、低、高和高高。当用户激活某一层次报警限时，将同时需要设置界限值和报警文本。报警文本是当报警产生时显示给用户的文本，如发生“低低层次”的报警时，如果报警文本填写的是“低低”，则展现给用户的文本也是“低低”，这样用户就可以知道产生的是“低低层次”的报警。界限值的含义如下：

* 低低：当变量值低于此界限值时，将会产生“低低层次”报警。
* 低：当变量值低于此界限值，但高于“低低层次”界限值时，将会产生“低层次”报警。
* 高：当变量值高于此界限值，但低于“高高层次”界限值时，将会产生“高层次”报警。
* 高高：当变量值高于此界限值时，将会产生“高高层次”报警。



图3-12 变量属性配置对话框

由于变量属性配置是相对独立的一个模块，因此本文设计了一个类来专门对配置数据进行管理，该类即VariantSetDialog。对于用户添加的每一个变量都将对应一个VariantSetDialog类的实例，用户可以访问其相关的属性的到变量的配置信息。VariantSetDialog类所具有的属性如表3-4所示。

表3-4 VariantSetDialog类属性

|  |  |  |
| --- | --- | --- |
| **属性** | **类型** | **描述** |
| variant\_name | QString | 变量名称 |
| variant\_type | VariantType | 变量类型 |
| value\_max | QVariant | 变量最大值 |
| value\_min | QVariant | 变量最小值 |
| info | QString | 变量描述 |
| alarm\_flag | bool | 是否报警，true为报警，false为不需要报警 |
| alarm\_name | QString | 报警名称 |
| alarm \_priority | int | 报警优先级 |
| alarm \_level\_ll | bool | true为选中“低低”层次报警，false则不选中 |
| alarm \_level\_ll\_limit | QVariant | “低低”报警层次的界限值 |
| alarm \_level\_ll\_text | QString | “低低”报警层次的报警提示文本 |
| alarm \_level\_l | bool | true为选中“低”层次报警，false则不选中 |
| alarm \_level\_l\_limit | QVariant | “低”报警层次的界限值 |
| alarm \_level\_l\_text | QString | “低”报警层次的报警提示文本 |
| alarm \_level\_h | bool | true为选中“高”层次报警，false则不选中 |
| alarm \_level\_h\_limit | QVariant | “高”报警层次的界限值 |
| alarm \_level\_h\_text | QString | “高”报警层次的报警提示文本 |
| alarm \_level\_hh | bool | true为选中“高高”层次报警，false则不选中 |
| alarm\_level\_hh\_limit | QVariant | “高高”报警层次的界限值 |
| alarm \_level\_hh\_text | QString | “高高”报警层次的报警提示文本 |

3.2.2 结构变量

结构变量只是通过将多个普通变量组合起来，从而达到统一管理多个变量的目的。在实际生产中，可能会存在有多个数据是相互关联的，因此将这些数据用一个结构变量来管理，这样就可以使得用户的每一次操作都是针对这些数据的，这样就不用对每个数据进行单独的管理了。

结构变量是由多个普通变量组成的，因此当要创建一个结构变量的时候，需要有相应的普通变量存在。所以即使用户只想创建一个结构变量，还是需要先创建相应的普通变量，并且为其配置好相应参数。

由于组态软件中管理数据的最小单位仍然是普通变量，结构变量只是作为对普通变量的一次封装，从而方便用户使用变量而已。因此对于数据的更新、报警等操作还是变量在进行管理，而结构变量只是起到一个索引多个变量效果。

结构变量由如下成分组成：

* 结构变量名称：用户为该结构变量命的名，在整个工程中是唯一的。
* 描述：对该结构变量功能的简要描述。
* 普通变量个数：记录该结构变量包含的普通变量的个数。
* 普通变量的索引数组：该数组记录了该结构变量包含的所有普通变量的索引。

3.3 系统配置模块

组态软件作为一个构建监控系统的工具，它并不能假设用户的某些设计意图。对于一个监控系统来说，有很多参数是与实际情况有关的，因此不同的系统会有不同的参数配置，为此组态软件需要为用户提供一个简明的配置模块，使得用户可以通过使用这个配置模块来达到系统参数的配置。

组态软件的系统配置模块主要分为三部分：监控站点配置、驱动配置和通信方式配置。可以看出这三项配置都是与数据相关的，监控站点配置主要是完成下位机上传到上位机的数据的解析；驱动配置主要是为每一个设备分配相应的驱动进行数据的收发；通信方式配置可以为系统提供每个设备与监控系统的通信方法。同时这三者也是密切相关的，驱动配置只是为系统配置了收发数据的驱动，但是读取的数据对于系统来说是不可理解的，因此需要关联相应的监控站点配置，而监控站点最终又将数据引向变量，前文也讨论过变量是作为组态软件各个模块的交流介质，这样设备所产生的数据才可以被系统理解。

下面将介绍监控站点配置、驱动配置和通信配置这三个部分的详细设计以及其实现细节。

3.3.1 监控站点配置

在组态软件中设备与监控软件之间传输的数据都是具有统一格式的，只有通过这种规定，系统才可以读取配置信息来解析数据，从而影响系统中的各个模块，为此本节在开始描述监控站点配置之前需要简要描述组态软件的数据的组织结构。

监控软件所监视的设备产生的数据可以是简单，也可以是复杂的，如有些设备只会向监控软件反馈开关的开合状态，因此只需要一个数据就行了，但是有些设备可能控制的是比较复杂的生产逻辑，因此可能会产生多个有独立含义的数据。不管是简单的一个数据，还是多个有独立含义的数据，设备都会一次将数据发送给监控设备，作为对生产状态的一次报告。

如图3-13，对于某一个设备来说，它可能会监视多个工作设备的生产状态，因此对于每一个工作设备生产状态所产生的数据将会使用一个Packet将其封装起来，然后再发送到监控软件。同时由于每一个工作设备可能需要同时反映多种生产状态，因此每个Packet数据包又将包含多个Item数据项，每个Item数据项代表着一种生产状态。



图3-13 数据组织层次

每个设备产生的Packet数据包具有如下格式：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Packet ID | Item ID | Data Length | Data | ··· | ··· | Item ID | Data Length | Data |

* Packet ID：作用是对多个工作设备产生的数据进行区别，从而让系统能够正确处理。
* Item ID：由于每个Packet可以配置多个Item，因此每个Item需要一个标识符进行区别。
* Data Length：即数据的长度，方便系统提取有效数据。
* Data：即实际的生产数据。

本文通过DataDevice、DataPacket和DataItem三个类来实现监控站点的配置，下面将描述这三个类的相关接口。

（1）DataDevice类的接口

* 函数原型：DataDevice(const int &device\_id)

描述：这是DataDevice类的构造函数，device\_id作为设备的唯一标识。

* 函数原型：void setDeviceName(const QString &device\_name)

描述：为创建的设备设置一个名字，方便用户理解。

* 函数原型：void addPacket(const int &packet\_id, const QString &packet\_name)

描述：该函数是为了向Device中添加一个DataPacket实例，id即所添加的DataPacket实例的标识符，name即所添加的DataPacket的名字。

* 函数原型：void addItem(const int &packet\_id, const int &item\_id, cosnt QString &item\_name)

描述：该函数向Device中指定的Packet添加一个Item，通过packet\_id指定了向哪个Packet添加，item\_id和item\_name分别指定了Item的标识和名字。

* 函数原型：DataPacket\* findPacket(const int &packet\_id)

描述：调用该函数可以向用户返回指定packet\_id的Packet实例。

* 函数原型：DataItem\* findItem(const int &item\_id)

描述：返回指定的Item。

* 函数原型：void itemAssociatingWithVariant(const int &packet\_id, cosnt int &item\_id, const QString &variant\_name)

描述：该函数实现指定的Item与指定的变量相关联。

（2）DataPacket类接口

* 函数原型：void insertItem(DataItem \*item)

描述：将指定Item插入到Packet中。

* 函数原型：DataItem\* getItem(const int &item\_id)

描述：返回指定Item标识符的Item指针。

（3）DataItem类接口

* 函数原型：void setVariant(const QString &variant\_name)

描述：将Item与指定变量关联起来。

3.3.2 驱动配置

由于组态软件并不能假定用户在监控系统中所用的设备，因此对于用户使用的驱动程序只能由用户自己提供。用户一般以动态链接库的形式将驱动程序放在指定的目录，但为了让组态软件知道驱动程序的存在，那就需要用户对每一种设备的驱动进行配置，并且同上一小节所描述的站点配置信息关联起来。

为了达到描述驱动的目的，驱动配置至少需要有五个属性：驱动标识符、驱动路径、与之关联的DataDevice实例（即站点配置信息）、接收数据函数的字符串和发送数据函数的字符串。

驱动标识符是该驱动在系统中的唯一标识，任何模块想使用该驱动只能通过该标识符来查找；驱动路径即驱动动态链接库的文件路径，通过该路径系统可以加载动态链接库，从而使用相关函数来与设备进行数据交互；关联的站点配置，每一个驱动程序都是与某一设备相关的，因此需要为每个驱动关联相关的设备配置信息；接收数据函数的字符串和发送数据函数的字符串，通过这两个字符串，可以从动态链接库中的到相关函数指针，从而实现与设备之间的数据收发。

3.3.3 通信方式配置

一般组态软件都支持多种通信方式，如以太网、CAN和串口等。为了实现统一的接口描述，因此都将这些通信方式进行了更高层次的封装。所以当需要和一个设备进行通信的时候，需要告诉系统应该使用哪种通信方式，这样系统才能做出正确地选择。

通信方式的配置包含两部分：通信参数配置和驱动关联配置。通信参数配置即当用户选定哪种通信方式后，需要进行的相关参数的配置。如图3-14，以串口通信方式为例，当用户将通信方式选择为串口时，将会出现于串口通信相关的参数配置，正如图中所示，用户需要配置串口号、波特率、数据位、停止位和奇偶校验等信息。



图3-14 通信方式配置界面

为了能够让通信方式配置信息与驱动配置信息关联起来，因此在通信方式配置界面用户还需要选定与该通信方式相关联的驱动标识符。

3.4 查询报警模块

在3.2节中介绍了变量配置模块，其中就提到了对于每个变量用户可以对其配置是否要产生报警信息，如果要产生报警信息，用户还对产生的报警信息进行了相关参数配置，如报警优先级、报警层次以对应报警层次的显示文本信息。这一节将描述组态软件对于报警的相关处理方法，同时描述了报警模块的相关实现细节。

本小节还将介绍组态软件中与查询相关的部分，对于一个监控软件来说，数据是最重要的，因此为用户提供相应的机制来查询历史数据，从而做出正确的决策时非常重要的。

3.4.1 报警系统

在生产过程中，设备会不断地向监控系统反馈自身状态信息，设备并不会对产生的相关数据进行解析，因此用户需要对相关的数据进行处理，从而反映出系统的真实运行情况。在变量配置中，用户为变量设置的报警界限值，这就是对数据的解析工作，通过判断数据值是否超出正常范围，并且如果超出范围，根据超出范围的程度进行分类，然后产生相应的报警信息，通过报警系统将事件反馈给观察员。当观察员从界面发现相关的报警显示的时候，就会对发生异常的设备进行查看，然后进行相应的处理。如果设备恢复正常，那么就会产生正常的数据，当系统检测到数据处于正常范围内，那么就会向报警系统发送一个警报解除的指令，从而让报警系统得知该警报已经解除，并对警报的显示做出相应的处理。

上面对报警系统的基本原理进行了阐述，可以得知与报警系统相关的数据分为两类：警报数据和警报解除数据。为了让系统能够很好的解析这两类数据，因此将两者都封装成了相应的类，同时为了使得报警系统的接口设计可以统一，与警报数据相关的两个类将拥有同一个父类，这样就可以利用C++多态的性质，统一使用父类指针进行数据处理。类的继承关系及相关属性如图3-15所示，基类AlarmData具有alarm\_type属性，通过它可以区分报警数据的类型，当alarm\_type等于0的时候为AlarmInvoke类型，即警报数据；当alarm\_type等于1时，为AlarmRelease类型，即警报解除数据。



图3-15 报警数据继承关系

基类AlarmData还具有variant\_name、alarm\_priority和alarm\_time三个属性，其中variant\_name表示变量的名称，即标识出警报是由哪个变量产生的；alarm\_priority表示警报的优先级；alarm\_time表示警报产生的时间。AlarmInvoke类有三个独有属性，分别是alarm\_level、alarm\_text和data\_value，alarm\_level表示警报的层次，即“低低”、“低”、“高”和“高高”；alarm\_text记录了警报产生时显示给用户的信息；data\_value表示触发警报的变量的值。AlarmRelease类具有一个私有属性，该属性向报警系统表面是否将该条警报解除指令存入数据库，有时警报解除指令也可以反映系统的运行情况，因此合理的记录可以方便后续的系统管理。

报警系统的功能是管理系统产生的各种警报，然后向用户反映到界面上，因此报警系统对报警数据和报警解除数据的处理是其核心。报警系统有两种方式可以向观察员反映目前的警报情况：通过相关动画控件和以表格的形式显示警报信息。用户可以在报警界面添加相关的图元，使其关联相应的变量，设置相关的限制值，从而反映变量的变化情况。然而警报信息以表格的形式呈现才是报警系统的主要展现形式，报警系统通过警报的优先级对警报信息进行排序，然后使得优先级最高的显示在列表的前面，如果同一优先级有多个警报信息，那么将通过警报产生的时间先后顺序对其排序。

报警系统由AlarmHandle类实现，由于需要对警报信息进行排序，因此AlarmHandle具有两个数据结构：

* QMultiMap<QDateTime, AlarmInvoke\*> date\_sort：date\_sort是用Qt的QMultiMap实现的，该数据结构允许同一个关键字可以由多个不同的值，date\_sort数据结构的关键字是QDateTime，即警报产生的时间，其对应的值则为AlarmInvoke类实例的指针。
* QMap<int, QMultiMap<QDateTime, AlarmInvoke\*>> priority\_sort：priority\_sort数据结构的关键字是警报优先级，优先级的总数在组态软件中已经是确定好了的，因此该表的大小也是确定了的。每个关键字对应的值则为QMultiMap<QDateTime, AlarmInvoke\*>类型的数据结构，这样就可以使得具有统一优先级的警报数据被紧挨着存放，相同优先级的警报数据再根据时间进行排序，从而使得报警系统能够将高优先级的警报信息优先呈现给监控者。

报警系统对警报数据的处理统一交给void handle(AlarmData \*alarm\_data)接口处理，其处理流程如图3-16所示。



图3-16 报警系统处理流程

从图3-16可以看出报警系统对数据的处理分为两类：当数据为警报数据时，系统会调用警报数据的getAlarmPriority方法获得该警报的优先级，然后根据优先级在priority\_sort表中进行查找，如果查找到了相应的date\_sort表，就调用getAlarmTime方法获得时间，用时间作为关键字将报警信息插入date\_sort表，如果没找到对应的date\_sort表，那么就创建，然后再插入，最后将警报信息插入数据库；当数据为警报解除数据时，同样是利用优先级查找到相应的date\_sort表，然后遍历该表，删除其中变量名与警报解除数据中变量名相同的数据项，最后根据save\_flag标识符决定是否将该条警报解除数据插入数据库。

3.4.2 历史数据查询

对于一个监控系统，不仅需要实时地反映生产状况，同时还需要为使用者提供浏览历史数据的功能。在很多情况下，历史数据可以帮助用户分析设备的运转情况，同时也为用户做决策提供了实际的数据支撑，因此本文设计的组态软件也为用户提供了相关历史数据查询的模块。

历史数据分为类：变量数据、警报数据和发送数据。变量数据即监控系统从设备接收到的实时数据，这些数据反映了设备的运转情况；警报数据即系统产生的警报，对于警报数据来说又分为两类，一是警报数据，二是警报解除数据；发送数据记录的是系统往设备发送的数据。

对于上面三类数据都设计了相应的数据库进行管理，变量数据对应着variant.db；警报数据对应着alarm.db；发送数据对应着senddata.db。将各类数据分散到不同的数据库存储的好处在于，用户可以根据实际情况的需要对数据库进行裁剪。

3.5 本章小结

本章主要描述了组态软件的开发系统的设计与实现。开发系统分为四个模块：图元模块、变量配置模块、系统配置模块和查询报警模块。在图元模块的设计与实现中，主要对图元的变形、图形显示和属性配置进行了讨论，并给出了具体的实现细节代码。在变量配置模块中，重点阐述了变量在组态软件中的作用，然后介绍了其设计方法。在提供配置模块中，描述了监控站点配置、驱动配置和通信方式配置这三者的关系，并且给出了具体的设计方案。在查询报警模块中，详细介绍了报警系统的整体结构以及运作流程。

本章就组态软件运行系统的关键的四个模块进行了详细的描述，并且给出了一个基于Qt开发库的实现方法，基本满足了组态软件开发系统的功能需求。

第四章 组态软件工程文件的设计与实现

上一章介绍了组态软件的开发系统，用户可以通过开发系统对所设计的监控系统进行相关参数的设置，为了保存这些配置信息，组态软件有特定的工程文件来进行存储。

组态软件的工程文件作为开发系统和运行系统之间的桥梁，它需要将各种数据分类组织好，这样才可以方便两个系统对相关文件进行操作。同时为了简化对文件的操作，工程文件的设计需要满足以下条件：

1. 工程文件具有多种类型，这样可以达到对数据分类存储的目的。
2. 工程文件具有统一的数据组织结构，这样可以方便数据的解析与存储。
3. 工程文件提供了一种能够方便将数据写入文件和读出文件的机制。

本章将围绕工程文件需要满足的三个条件来展开对工程文件的设计与实现的详细讨论。

4.1 工程文件的分类

将同类型的数据存储在同一个文件中，这样能够很方便的对数据进行处理，因此需要对组态软件中涉及到的数据进行分类，然后将其存储在不同的工程文件中。本文描述的组态软件的工程文件分为下列几类：

（1）与工程信息相关

对于一个经过组态软件开发系统所生成的工程来说，需要有一个文件来专门记录工程的整体信息，包括工程名称、工程创建时间、工程修改时间、工程创建人、工程描述以及其他工程文件的路径信息等，这些信息都将记录在工程索引文件中，即“工程名.prj”文件（对于一个具体工程来说）。

其中其他工程文件的路径信息这一属性非常重要，系统通过解析工程索引文件，然后根据该信息判断其他文件是否存在，从而确定工程的完整性，如果发现工程缺失文件，将会告知用户。

（2）与图元相关

上一章介绍过图元库是以动态链接库的形式存在的，同时图元库还对应着一张图元信息表。因此与图元相关的文件有图元动态链接库和图元信息表，通过这两个文件可以将图元库导入系统，方便用户使用。

（3）与数据库相关

组态软件中需要记录的数据可以分为三类：变量数据、警报数据和发送数据。该三类数据将分别用三个数据库（variant.db、alarm.db和senddata.db）进行管理，数据分为三个数据库存储是因为在某些系统中有些数据是不必要的，因此用户可以将某个数据库移除掉。

（4）与变量配置相关

前面已经介绍过，变量作为组态软件中各个模块交换数据的介质是非常重要的，因此变量配置的信息将会用一个文件来记录，即variant.config文件。

（5）与系统配置相关

与系统配置相关的文件包括三个：监控站点配置文件、驱动配置文件和通信方式配置文件，分别对应文件site.config、driver.config和communication.config。

（6）与画布相关

画布作为用户图形化界面设计的载体，因此需要记录用户在上面进行过得各种操作，而画布中的元素是图元，因此对于画布的存储即是记录图元的相关属性，通过将图元信息写入文件，以后可以通过文件内容又能再现画布形态。一个工程可能包含多个画面，因此每个画面都将存储为一个文件，然后将这些文件统一放入canvos文件夹中进行管理。

4.2 工程文件的数据组织

上一小节介绍了工程中涉及到的各类文件，虽然组态软件的工程文件有多种类型，但是文件中数据的组织形式是统一的，如图4-1所示，每一个工程文件都包含两部分内容：文件信息和数据。

文件信息主要记录该文件的一些基本属性包括标识符、版本号、幻数以及创建相关的信息。标识符是作为文件类型的标识，通过该标识符能够唯一确定文件的类型，这样系统就不必通过判断文件的后缀名来判断文件类型，从而提高系统安全性。版本号和幻数都是用来确定生成工程文件的组态软件的版本的，这样就可以防止组态软件和工程文件由于版本不匹配造成的问题，因为通过这两个属性，在加载工程的时候，系统将会通知用户版本不匹配的信息。与创建相关的信息主要包括创建者和创建时间，同时还包括修改者和修改时间，通过这些信息用户可以得知对文件的操作状态，如当文件被修改了，那么可以查看是谁进行的修改以及修改的时间。

图4-1也展示了文件的数据部分的组织格式，文件的数据部分由多个数据项组成，例如在变量文件中，每一个数据项就代表着一个变量。由于每个数据项包含的内容也比较复杂（如每个变量将包含多个属性），因此每个数据项分为标识符和数据两部分。标识符是该数据项的唯一标识（如果是变量，那么就是变量名），每个数据项的标识符不能相同。数据部分则是由多个数据段组成，如图4-1中的数据1和数据2，每个数据段又通过三个属性进行描述：类型、长度和数据。类型表示的是该数据的数据类型，例如变量的值可以是整型、浮点型和布尔型，长度表示的是实际数据的长度，通过它系统能够确定读取了多少数据，数据即实际数据。



图4-1 文件数据组织格式

4.3 数据的序列化与反序列化

用户保存设计文档的动作，也就是将组态软件中各个类实例序列化到文件中的过程，因此数据的存储也就变为类的序列化问题了。

Qt为用户提供了QDataStream类，利用该类用户可以将Qt库自带的基本类型序列化到文件。利用该类来实现数据的存储有两大优势：一是该类提供了丰富的接口，利于软件的开发；二是该类对基本类型序列化后的结果是独立于操作系统和处理器的，因此不会存在字节序不一致的情况，这对于本文所设计的可移植组态软件是必需的。

由于QDataStream类只能实现基本类型的序列化，因此对于本文描述的各个类的序列化需要进行相应的处理，下面以图元的序列化为例描述实现过程。

由于每一个图元所包含的信息是不同的，因此每个图元需要存储什么信息只有自己才知道，为此图元基类Primitive将包含virtual void serialized(QDataStream &out)虚函数，而每个图元将重写该函数来保证自己能够被正确地实例化。对于每个图元序列化都需要进行两步：

1. 将图元的名称写入文件，作为数据项的标识符。
2. 依次处理图元的各个属性，例如处理图元的位置信息，首先存储数据类型即QPoint（Qt中基本类型都有相应的枚举值），利用sizeof(QPoint)得到数据的长度，然后再将实际数据写入文件。其他各个属性依次进行上述处理，即可完成该图元的序列化。

对于数据的读取（反序列化），即上述过程的逆过程。每一个需要反序列化的类都有unserialized方法，每一个类都知道自己该如何解析数据，使得自己的所有属性都具有正确的值。

4.4 工程导入流程

组态软件中，不管是开发系统还是运行系统都需要多次导入工程，开发系统导入工程是为了能够修改工程参数，运行系统导入工程是为了获得系统的相关配置信息，虽然两者的目的不同，但是其工程导入的流程大体都是一样的。

工程导入流程主要分为三个阶段：工程索引文件解析、各类文件解析和解析结果判断。如图4-2所示，



图4-2 工程导入流程图

在工程索引文件解析阶段，系统将首先确定文件的基本信息是否正确，如版本号等；其次系统将寻找索引文件中列出的各类文件的路径信息是否正确，如果不正确，将向用户做出相应的提示；最后在校验都成功的情况下，系统将各类文件的路径信息分配给相应的解析任务进行解析。

在各类文件解析阶段，对于每一类文件都将有相应的解析任务来处理，如变量，就会存在相应的变量解析任务，解析任务通过解析文件，然后将数据放入指定的数据结构中，如果解析出错，将会向系统报告错误。

在解析结果判断阶段，最后系统会汇集所有解析任务的解析结果，如果所有结果都为正确，那么工程导入结束；如果有文件解析不成功，那么系统将向用户反馈具体的错误信息，方便用户进行相应处理。

4.5 本章小结

本章描述了组态软件中工程文件的设计，工程文件作为连接开发系统和运行系统的桥梁以及用户设计思想的成果，在组态软件中具有重要地位。本章通过对工程文件的分类、工程文件的数据组织和数据的序列化和反序列化这三个方面的详细描述，展现了该组态软件工程文件的设计思想和方法。

本章最后还描述了组态软件中工程导入的具体流程的设计。

第五章 组态软件运行系统的设计与实现

上两章对组态软件的开发系统和工程文件进行了详细的设计描述，这一章将对组态软件的运行系统的相关细节进行介绍。运行系统作为呈现用户设计结果的工具，它主要由四个模块构成：交互界面模块、控制模块、通信模块和历史数据处理模块。本章将依次详细讲解这四个模块的设计与实现。

5.1 交互界面模块

通过组态软件开发系统得到的结果应该是一个监控软件，对于一个监控软件来说人机交互界面是其中非常重要的一部分，因此组态软件的运行系统利用交互界面模块对这一部分进行管理。

交互界面模块主要具有三个功能：图元加载器、图形显示刷新和人机交互。图元加载器主要负责对用户添加到画布上的图元进行管理；图形显示刷新描述的是图元如何根据变量的变化来改变自己的显示；人机交互完成了上位机和下位机的互动。

（1）图元加载器

上一章介绍过，系统首先解析工程索引文件，然后将各类文件的解析分配到相应的功能模块，因此与画布相关的文件的解析将有图元加载器完成。图元加载器的工程由CanvosManager类实现，在第三章已经介绍过该类，当时是在组态软件开发系统的设计中提到的，当用户在开发系统中在画布上对图元进行操作的时候，该类将记录所有的用户操作，方便以后的存档工作。由于开发系统和运行系统都需要进行图元加载工作，开发系统进行图元加载是为了对已有工程进行修改，所以在设计的时候由同一个类实现。

CanvosManager类有两个关键数据结构：QMap<QString, QStringList>，即图元变量关联表，该结构的关键字为变量名，值为与该变量相关联的图元的对象名称，通过该结构系统可以根据变量名索引到与之关联的所有图元，从而到达有目的的操作；另一个数据结构是QMap<QString, Primitive\*>，即图元索引表，该结构的关键字为图元的对象名称，值为相应图元的指针，通过该结构系统能够根据对象名称索引到图元实例。

整个图元加载器需要完成的功能就是对这两个数据结构进行赋值，其流程如图5-1所示。



图5-1 图元加载流程

如图5-1，图元的加载流程分为如下步骤：

* 获得相关文件，判断文件是否有效，若文件无效记录相关错误信息，文件有效则继续处理。
* 依次读取文件中的每一个数据项，直到文件读取完。对每个数据项的处理过程如下：
  + 读取数据项的标识符，该标识符是图元的名称，通过该标识符可以构造相应字符串，即get+图元名格式，然后利用该字符串从图元动态库中提取相关函数，从而获得相应图元的实例。
  + 继续读取数据项，设置图元相关属性。
  + 当图元属性设置完成后，调用图元getName方法获得图元的对象名称，然后将图元存储在图元索引表中；再调用图元的getVariantName方法得到图元所关联的变量名称，然后设置好图元变量关联表。

（2）图形显示刷新

监控系统的图形显示是一个动态变化的过程，图元作为图形显示控制的最小单元，因此图形显示变化本质是图元在发生变化。由于图元本身是静态的，因此它需要外界数据来刺激其发生变化，而外界数据在组态软件中的映射物为变量，最终图形显示的变化就在于变量的变化。因此该功能模块需要解决的问题是，当设备通过相关通信方式将数据传送到监控系统从而导致与之关联的变量的值发生变化的时候，系统如何告知图元。

在Qt中有一种对象之间的通信机制，即信号/槽。该机制可以将两个对象进行关联（对象可以存在于不同线程中），从而使得两个对象可以通信。事件发生时，对象可以发送对应的信号，对该信号关注的对象，可以将该信号与自己的相关处理函数进行关联。这样就可以使得对象能够对自己关注的事件进行处理。

在图形显示刷新中涉及到两个对象：CanvosManager和VariantManager。CanvosManager管理的是画布中的相关图元，而VariantManager（下一小节介绍）管理着系统中的所有变量。当VariantManager监测到有变量的值发生变化的时候，将所有发生变化的变量的名称放入一个QStringList中，然后产生void variantChange(QStringList)信号，所带参数即变化了的变量的名称的集合。而CanvosManager则通过connect函数将自己的处理函数与该信号进行了关联，一旦信号产生将调用处理函数进行处理。处理过程如图5-2所示。



图5-2 图形显示刷新处理流程

（3）人机交互

对于一个监控系统，监控界面是操作员能够用眼睛感知到的东西，因此当需要和下位机（即受监控的设备）进行交互的时候，往往也是通过界面操作的途径进行的。

在监控界面中，设计人员可能设计了相关按钮，当操作员对按钮进行操作的时候，这是就会触发系统产生一条命令，然后系统会将该命令交给数据发送器（后面小结会介绍）处理，数据发送器依次发送命令，最后命令将到达相关设备，设备通过解析命令做出对应的反应，然后又将在监控界面中显示出执行命令后的效果。

监控系统正在运行的过程中，操作人员可以进行相关数据的查询，这属于人机交互的一部分。前几章介绍过，用户可以通过组态软件提供的查询界面进行相关操作，可以对历史变量数据进行查询，可以对历史警报数据进行查询，也可以对发送过的命令进行查询。

5.2 控制模块

控制模块处理的内容主要包含配置文件和数据。控制模块需要将用户配置的信息导入内存，并以合理的结构进行存储，这样方便索引；同时控制模块还要负责处理下位机传来的数据。下面将依次详述相关功能部分。

5.2.1 变量管理器

上一章介绍了与变量相关的工程文件，而该工程文件则是由变量管理器来进行解析的。变量管理器主要完成如下工作：

（1）记录变量配置信息，如变量值类型，是否报警等属性。

（2）记录变量的最新值，方便其它模块使用。

（3）保留近期数据，这样可以不用访问数据库就能得到近期数据。

（4）在该变量需要报警的情况下，记录报警情况，分析是解除还是继续报警。

（5）每一次更新变量，将产生一条数据记录，发送给数据库处理部分。

（6）发送相关信号，通知各模块数据已经更新。

变量的相关信息是通过Variant类来存储的，该类具有如下属性：

* VariantAttribute attribute：记录工程文件中读取出来的变量配置信息，包括变量的名称、值得类型、是否报警以及报警相关的配置信息。
* QVariant latest\_value：记录变量的最新值。
* bool is\_alarmed：值为false时，表示不在报警状态；值为true时，表示在报警状态。
* QMap<QTime, QVariant> value\_buf：该结构关键字为时间，值为变量在该时刻的值，通过该结构能够记录一段时间内变量的值。

变量管理器的功能由VariantManager类来实现，该类存储了变量的相关属性，由于一个系统具有多个变量，因此为了很好的索引各个变量，VariantManager通过使用QMap<QString, Variant\*> variants结构进行管理，该结构可以使用变量名作为关键字查询到变量的相关信息。

除此之外，VariantManager还提供了下列几个重要方法。

（1）void init(QString &file)：该方法完成了解析变量配置信息的工程文件的功能，调用该方法主要是为了初始化Variant类实例。其处理流程分为如下步骤：

* 校验文件信息，判断是否为有效文件。
* 遍历提取数据项，获得变量名称，创建Variant实例，继续读取数据初始化Variant实例。当变量信息初始化完成后，将该变量记录到variants结构中。

（2）QVariant getLatestValue(QString &name)：通过该方法，用户可以查询到指定变量名的变量的最新值。

（3）QList<QVariant> searchVariantValue(QString &name, QTime &from, QTime &to)：用户可以使用该方法查询某段时间内某个变量的值，结果以链表的形式返回，name代表需要查询的变量名，from代表查询时间段的开始时间，to代表结束时间。

（4）void updateVariant(QMap<QStirng, QVariant> values)：当系统将取得的数据发送给变量管理器的时候，该函数被触发，values里存储了新的变量值。该函数处理流程如图5-3所示。



图5-3 变量更新处理流程

从图5-3可以看出，变量管理器接收到系统传输过来的数据时的处理步骤如下：

（1）获取一项数据，该项数据也已经在前面描述过，格式为<变量名，变量值>。通过变量名可以从variants结构中查找到相应的Variant类的实例，通过VariantAttribute类实例可以得知变量值得类型，然后将变量存储在latest\_value属性中。

（2）根据最新的变量值，读取attribute属性中的关于变量报警信息的描述，决定是否产生报警信息。如果变量需要设置报警，那么将根据变量的实际值与变量配置属性中的界限值进行比较，从而确定是否产生警报。如果变量值处在警报界限值范围内，那么发送报警数据；如果变量值处在正常范围，那么根据is\_alarmed的值进行下一步处理，当其值为false时，表示没有发生过报警，系统不做处理，当其值为true时，表示发生过报警，那么此时将发送警报解除消息并且将is\_alarmed设置为false。

（3）将变量的最新值插入到数据库并且添加到value\_buf中。变量管理器构造一条格式为变量名 变量值 日期 时间的数据，然后放入数据库模块的数据缓存中，数据库管理器将在合适的时间将数据写入数据库。

（4）将该变量的名称记录到一个链表中，这样当通知图元变量发生变化时，图元管理器可以通过这条链表得知哪些变量发生了变化，从而对相关的图元进行刷新操作。

（5）当所有的变量都更新完毕之后，变量管理器将向图元管理器发送信号，通知其变量已经更新。

5.2.2 系统信息管理器

在第三章详细介绍过与系统配置相关的功能模块，主要包含三部分：站点配置信息、驱动配置信息和通信方式配置信息。为了能够对着三类信息进行统一的管理，因此组态软件中有对应的系统信息管理功能，在本课题中被称为系统信息管理器。系统信息管理器主要的功能包括加载与系统配置相关的工程文件和管理对应的配置信息。

系统信息管理器由SystemInfoManager类实现，该类包含的主要属性和方法如表5-1所示。

表5-1 SystemInfoManager类的属性和方法

|  |  |  |  |
| --- | --- | --- | --- |
| **SystemInfoManager类的属性** | | | |
| 属性名 | 类型 | | 描述 |
| site\_info | QMap<int, DataDevice\*> | | 记录所有的站点配置信息 |
| driver\_info | QMap<int, DataDriver\*> | | 记录所有的驱动配置信息 |
| com\_info | QMap<QString, DataComMethod\*> | | 记录所有的通信方式配置信息 |
| **SystemInfoManager类的方法** | | | |
| 方法名 | | 描述 | |
| DataDevice\* getSiteInfo(int site\_id) | | 获得指定site\_id的站点配置信息 | |
| DataDriver\* getDriverInfo(int driver\_id) | | 获得指定driver\_id的驱动配置信息 | |
| DataComMethod\* getComInfo(QString &com\_id) | | 获得指定com\_id的通信方式配置信息 | |
| void initSiteInfo(QString &file) | | 根据工程文件初始化site\_info | |
| void initDriverInfo(QString &file) | | 根据工程文件初始化driver\_info | |
| void initComInfo(QString &file) | | 根据工程文件初始化com\_info | |

SystemInfoManager类的三个属性都是通过QMap来存储的，这样可以快速地找到需要的配置信息，对site\_info属性，QMap的关键字为站点的id号，值为DataDevice类型的指针；对于driver\_info属性，QMap的关键字为驱动的id号，值为DataDriver类型的指针；对于com\_info属性，QMap的关键字为一个字符串（以太网通信的时候即为设备IP地址，串口通信时则为串口号），值为DataComMethod类型的指针。关于DataDevice类的结构在3.3.1小节已经做过详细介绍，下面主要介绍DataDriver和DataComMethod两个类的详细结构。

DataDriver类主要记录从驱动配置工程文件读出的数据，由于存在多个驱动程序需要管理，因此DataDriver的实例通常也不止一个，其属性如表5-2所示。

表5-2 DataDriver类的属性

|  |  |  |
| --- | --- | --- |
| 属性名 | 类型 | 描述 |
| driver\_id | int | 驱动配置信息的标识符 |
| driver\_path | QString | 对应驱动动态库的路径 |
| site\_id | int | 与该驱动关联的站点配置信息的标识符 |
| rcv\_str | QString | 驱动动态库中接收函数的函数名 |
| send\_str | QString | 驱动动态库中发送函数的函数名 |

其中rcv\_str和send\_str属性是在系统加载驱动动态库的时候使用的，通过这两个字符串，系统能够获得动态库中相应的函数地址指针。

DataComMethod类记录了通信方式配置的相关信息，监控系统中存在多少需要监控的设备，那么就需要相等数量的DataComMethod实例来记录，其属性如表5-3所示。

表5-3 DataComMethod类的属性

|  |  |  |
| --- | --- | --- |
| 属性名 | 类型 | 描述 |
| com\_id | QString | 通信方式配置信息的标识符 |
| com\_type | CommunicationType | 通信方式类型 |
| driver\_id | int | 与该通信方式关联的驱动信息 |
| com\_attr | CommunicationAttribute | 通信方式的相关属性 |

CommunicationType为一个枚举类型，其定义如下：

typedef enum{

Ethernet, //以太网方式

SerialPort, //串口方式

Bluetooth, //蓝牙方式

TypeSize

}CommunicationType;

其中com\_attr属性为CommunicationAttribute类型的指针，由于通信方式的不同将会导致配置信息的不同，因此利用c++多态的特性，使得每一种通信方式的配置都由一个类管理，而该类继承于CommunicationAttribute类 ，属性配置相关类的继承关系如图5-4所示。



图5-4通信方式属性配置类继承关系

5.3 通信模块

5.3.1 驱动管理器

驱动管理器作为通信模块的核心，正如其名，它的主要工作就是将系统与设备建立好的连接和对应的驱动程序进行关联。驱动管理器需要完成如下工作：

* 记录当前连接设备的相关信息，关联相应的驱动。
* 解析来自设备的数据，方便变量管理器对变量进行更新。

（1）活动设备信息表

为了能够很好的管理监控软件正在监控的设备的情况，因此每一个与监控软件建立了连接的设备都将在驱动管理器中存在一条记录，通过该记录系统才能调用相应的驱动程序以及正确地解析数据。

组态软件通过DeviceConnectingInfo结构体来记录相关信息，其定义如下：

/\*驱动程序的读数据函数原型，buf表示缓存区，len记录读到的数据大小\*/

typedef void (\*ReadMethod) (char \*buf, int \*len);

/\*驱动程序的写数据函数原型，buf代表数据，len代表要写的数据长度\*/

typedef void (\*WriteMethod) (char \*buf, int len);

typedef struct{

QIODevice \*device; /\*设备建立连接后的指针\*/

int site\_id; /\*该设备关联的站点信息标识符\*/

CommunicationType com\_type; /\*通信方式\*/

ReadMethod read; /\*读数据函数指针\*/

WriteMethod write; /\*写数据函数指针\*/

}DeviceConnectingInfo;

DeviceConnectingInfo结构体中，由于在Qt中不管是以太网通信还是串口通信，其相关的类都是继承于QIODevice类，因此可以利用QIODevice类型的指针来存储各种通信方式建立的连接的实例；记录site\_id是为了能够查询到站点配置信息；通过com\_type能够获知该连接属于什么通信方式，从而可以对device的指针类型进行转换；read和write是该连接的读写函数，通过提取对应驱动中的函数来为该两个属性赋值。

在驱动管理器中，每当有一个设备请求建立连接的时候，都将初始化一个DeviceConnectingInfo实例，其处理流程如下。

* 根据连接的通信方式标识符，调用系统信息管理器的getComInfo方法得到相应的通信方式配置项。
* 根据通信配置项中的driver\_id，调用getDriverInfo方法得到对应的驱动配置项。
* 根据驱动配置项中的驱动路径信息，加载驱动，提取相关函数，为read和write赋值。
* 提取驱动配置项中的站点信息配置的标识符，赋值给site\_id。
* 将该连接信息加入活动设备信息表中，完成一条连接的信息初始化。

（2）数据解析

文章的前面章节已经介绍过站点信息配置的详细过程，可以知道来自设备的数据应该具有如下格式。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Packet ID | Item ID | Data Length | Data | ··· | ··· | Item ID | Data Length | Data |

通过从站点配置信息得到信息，可以将每一个Data与相应的变量关联起来，从而形成一个数据对<变量名，数据>，然后将该数据对放入QMap中，当数据解析完毕，驱动管理器将发送相应信号。

5.3.2 数据采集器和发送器

数据采集器和数据发送器负责接收和发送来自监控设备的数据，为了能够对多种通信方式进行统一的管理，它们需要解决如下问题：

* 组态软件支持多种通信方式（如以太网、串口和蓝牙），该如何统一地管理这些连接？
* 当得知有数据到达时，利用什么机制通知系统？
* 得知是哪个设备发送过来的数据的时候，该如何进行数据的读取？

下面将对上述三个问题给出详细的解决方案。

（1）管理连接

在Qt中，所有与外设通信相关的类都有共同的基类QIODevice，因为不管是以太网通信还是串口通信，建立连接后返回的都是对应通信类的指针，这些指针都可以通过基类的指针来存储，从而在管理上也就比较统一了，对于系统来说不需要记住每个连接的通信方式，也不需要对每个连接进行特殊的处理。

（2）数据到达通知与数据读取

由于数据采集器需要管理多个连接，因此不可能以轮询的方式来判断每个连接是否有新的数据需要处理，这样会降低处理速率。为了达到预期的处理效果，因此需要利用Qt的信号/槽机制，由于每一个连接一旦有新的数据将会发送readyRead信号，如果将该信号与相应的处理函数进行关联，那么就可以达到异步处理的效果。

但是上面也介绍了，为了使得各种通信方式能够统一的管理，所有的连接实例都是用QIODevice类型的指针来存储的，然而readyRead信号是不带参数的，因此为了能够确定是哪个连接发送的信号，需要调用QObject::sender()方法获得发送信号的实例的指针，关键代码如下。

//与系统建立连接的设备集合

QList<QIODevice\*> connections;

//建立好每个连接的信号与槽函数关系

for(auto item : connections)

{

connect(item, SIGNAL(readyRead()), this, SLOT(updateData()));

}

//信号处理函数

void updateData()

{

//获得发送信号的连接

QObject\* obj = QObject::sender();

//根据obj查找活动设备表

//利用相应的read函数进行数据读取

//将数据发送给驱动管理器

}

从上面的代码可以看出，一旦某个连接产生了新数据，那么该连接将会发送readyRead信号，触发相应的槽函数，而所有的信号都共用一个槽函数，在这个槽函数中将通过发送信号的对象的指针查找活动设备表，从而得到一个DeviceConnectingInfo实例，通过该实例中的read属性就可以读取设备发送过来的数据，数据读取成功后将通知驱动管理器进行数据解析。

数据的发送过程是数据接收过程的逆过程，首先将数据封装好，然后查找到对应的连接，利用write函数进行数据发送。

5.4 历史数据处理模块

历史数据处理模块包含两个功能部分：历史数据操作和数据库管理。历史数据操作主要实现了用户对历史数据的查询和删除；数据库管理主要实现将各种数据写入数据库中，从而可以长期保存。

Qt提供了访问数据库的相应API，因此能够方便的对数据库进行管理。当有数据被放入数据库缓存的时候，数据库线程将会被唤醒，然后判断存储的数据是否已经达到上限值，如果达到将一次将所有缓存的数据写入数据库中，如果没有就继续等待，这样可以减少IO读写时间。

5.5 模块间同步

本章前几小节已经对运行系统的界面交互模块、控制模块、通信模块和历史数据处理模块四个模块的设计与实现进行了详细描述。虽然每个模块的具有相对独立的功能，但是运行系统要完成监控任务需要个模块之间的协调工作。对于模块之间的交互主要是数据之间的同步，因此解决模块之间数据的共享问题是解决模块间同步问题的关键。图5-5展示的是模块之间的关系，该图主要以数据为中心描述了模块之间的交互。



图5-5 模块间关系

从图5-5可以看出，对于数据的处理主要由三条路径组成：接收到新数据后的处理、发送命令的处理和历史数据的处理。

（1）收到数据

通信模块中的数据采集器负责接收各设备传回来的监控数据，当数据采集器接收到完整的数据后将数据交给驱动管理器，驱动管理器需要从控制模块中的系统信息管理器获得该设备数据的解析方式，根据用户的配置，驱动管理器将数据解析成<变量名，值>的格式放入缓存中，即图5-5中所示的新接收的数据。当驱动管理器处理完数据采集器发送给它的数据的时候，驱动管理器将通知变量管理器，让其知道变量已经更新。

变量管理器接收到变量更新信号后，将更新内部的数据结构，同时完成这些工作后，它将向交互界面模块的图元管理相关的功能部分发送信号，然后图元根据变量最新的值改变自身的状态和显示。

（2）发送数据

当用户在监控界面上进行相关操作的时候，这时会向被监控的设备发送命令，命令产生后先是放入命令发送队列中，然后进过驱动管理器的相应处理后交给数据发送器，将其发送至目标设备。

（3）历史数据处理

历史数据包括变量数据、命令数据和警报数据。当变量管理器更新自身数据结构后，会产生一条数据将其放入数据库的缓存中。当交互界面产生一条命令的时候，也会将该命令放入数据库缓存中。同时若系统产生了报警，那么还将警报消息放入数据库缓存中。数据库管理部分将会在恰当的时候将缓存中的数据写入相应的数据库。

对于历史数据，用户可以通过界面进行查询，该功能由历史数据操作功能完成。

在对运行系统进行设计的过程中，界面交互模块、控制模块、通信模块和历史数据处理模块四个模块分别是由一个线程进行管理的，因此完成线程之间的同步即可达到模块间的同步。

在对线程进行同步的过程中，主要用到了Qt提供的两种技术。

* 信号/槽机制：该机制可以实现多个对象之间的通信，通过将对象之间的信号和槽函数进行连接，一旦某个对象产生某个信号，那么与之相关联的对象将被通知，然后该对象就可以进行相关的操作。例如当变量管理器更新了变量，通过发送变量更新完成信号，图元管理器将接收到该信号，从而更新图形显示。
* 互斥量和条件变量机制：对于两个线程需要共享同一块内存数据的时候，这是需要将线程的操作进行序列化处理，这样才能保证数据安全。互斥量提供了线程能够序列化访问共享数据的功能；条件变量使得线程可以等待条件发生再进行相关操作。两者通常需要同时使用。例如在历史数据处理模块中，一旦数据库缓存中插入了新数据，那么此时有数据需要插入的条件满足，那么该线程将被唤醒，从而可以进行相应的处理。

5.6 本章小结

本章详细介绍了组态软件中运行系统的详细设计与实现，分别对运行系统中的交互界面模块、控制模块、通信模块和历史数据处理模块的实现策略进行了详细阐述。在交互界面模块中主要描述了与图元相关功能的实现，在控制模块中详细展示了变量管理的细节，在通信模块中重要讲解了数据的流动与解析，在历史数据处理模块中简述了数据库的相关操作。

本章的最后还对整个运行系统的各模块的同步作了详细介绍。