TASKCTL

敏捷批量调度开拓者，开启批量调度工具化时代

敏捷调度技术平台 v7

产品核心介绍

成都塔斯克信息技术有限公司

产品网站：www.taskctl.com

1 前言

1.1 文档目的

产品核心指 TASKCTL 调度平台的服务后台，它是整个产品最核心部分。本文旨在描述产品核心架构、主要组件以及相关重要的工作原理与机制。虽然产品核心主要以后台服务程序的方式呈现，并且用户具体应用是通过客户端工具操作完成，但是读者通过本文，可以对 TASKCTL 有更深入的了解，特别在系统维护时，有极其重要的帮助作用。

1.2 读者对象

《TASKCTL 7.0 产品核心》主要适合以下读者对象：

- ✓ 系统维护任务
- ✓ 技术开发人员

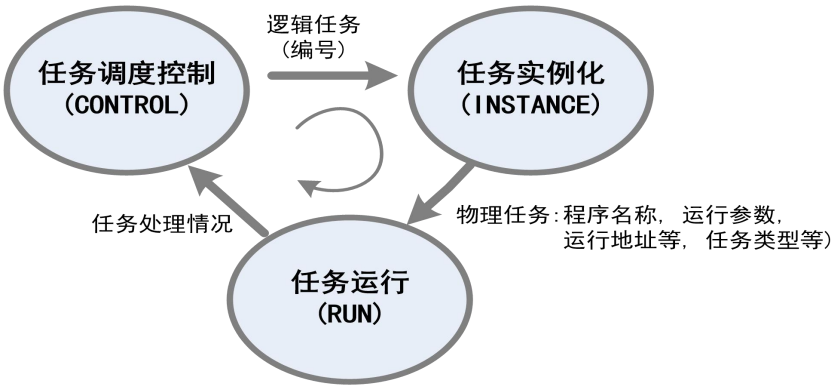
2 总述

产品核心是建立在 CIR 概念基础上的核心控制体系，它主要由一系列服务组件部署在不同节点上，并通过不同服务组件间协调处理，从而完成任务调度以及与客户端通信处理等工作。

2.1 核心概念 – CIR

TASKCTL 产品核心，主要工作原理是建立在 CIR 概念基础之上。

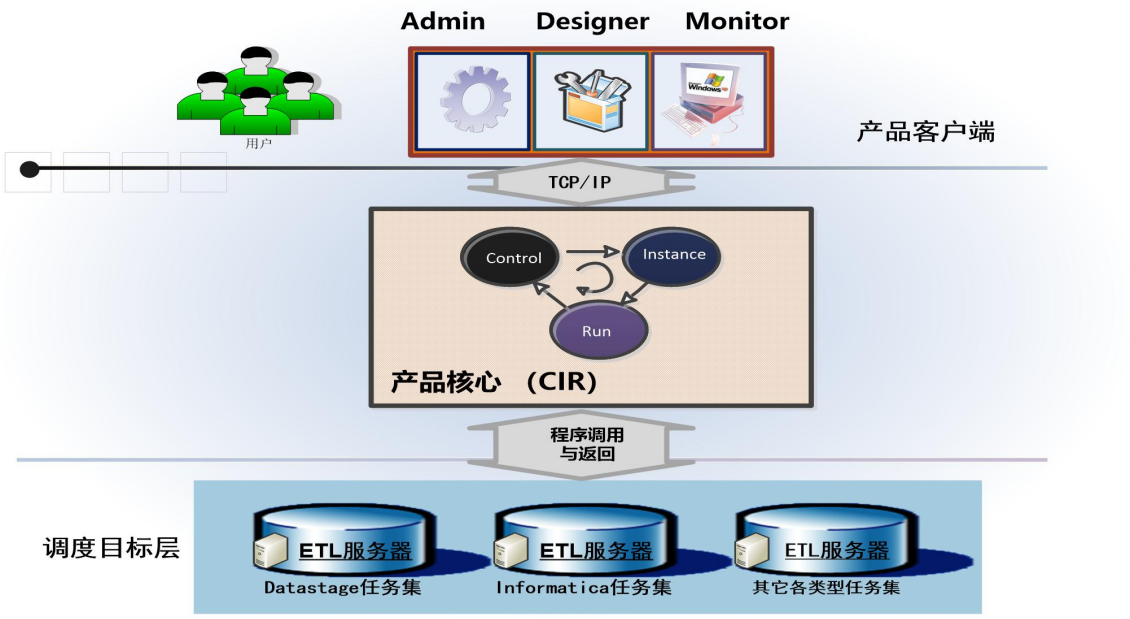
CIR (即 Control Instance Run 的缩写)，它主要指调度核心运行机制概念。Control：指任务调度；Instance：指对逻辑任务的实例化，确定逻辑的任务的程序名称、参数、运行环境等相关信息；Run：执行实例化后的任务。CIR 概念提出主要是为了对任务调度这一领域的核心内容进行统一定义。CIR 充分体现了调度控制的核心处理过程，并确定了每一过程的职责范围，C、I、R 三个步骤既独立又统一。它们之间的关系如下：



CIR概念模型

2.2 架构体系

2.2.1 总体架构



TASKCTL-CIR 2.1 整体架构

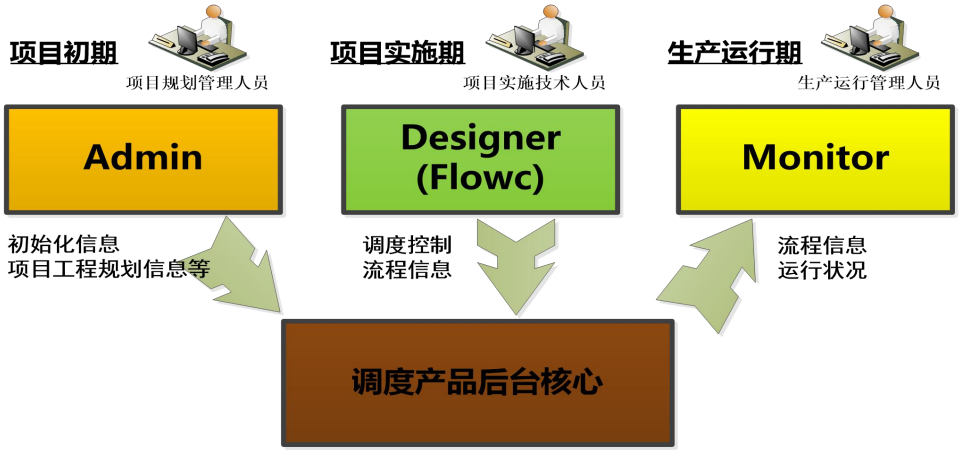
由上图可知，TASKCTL-CIR 2.1 产品主要由客户端工具软件与产品核心构成。对于调度目标 ETL 服务器部署的诸如 Datastage、Informatica 等各种不同类型的任务程序，主要通过产品核心进行调度与控制。

（一）客户端软件

TASKCTL-CIR 客户端分别由桌面图形客户端与后台字符界面客户端两套独立应用体系构成，每个应用体系均包含三套软件，分别为：

- ✓ Admin：调度平台管理软件
- ✓ Designer(后台称 Flowc)：流程开发设计软件
- ✓ Monitor：流程监控维护管理软件

以上不同渠道的三套软件关系为：



另外，虽然前后台均具有以上三套相互对应的客户端软件，但前后台软件还是存在一定的区别，主要表现为：

（1）前台桌面软件比后台软件更直观：由于不同展示技术的原因，前台采用图形方式，而后台采用字符方式，因此在一些展示功能方面以及操作便捷性方面，前台比后台更具优势。

（2）后台软件比前台桌面软件功能更全面：但如果只从功能角度，后台比前台更为全面，有些深层次的应用，前台可能不具有，但后台有相关的应用。比如流程 Debug 功能，前台不具备有后台 Flowc 软件具备该功能。

（二）产品核心

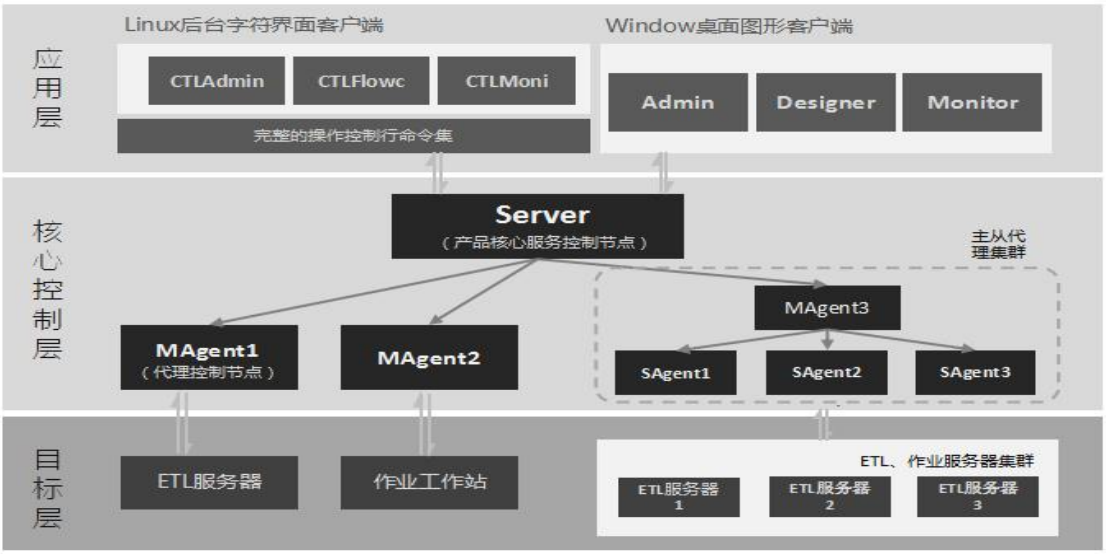
产品核心是建立在 CIR 概念基础上的核心控制体系，它主要由一系列服务组件部署在不同节点上，并通过不同服务组件间协调工作而完成任务调度以及与客户端通信处理等工作。

另外，产品核心目前只能运行于 AIX 以及 unix 环境。

（三）调度目标层

调度目标层主要指 ETL 服务器上部署的各种各样的任务程序，比如 Datastage 开发的 Job、Informatica 开发的 Session 或者 Shell、存储过程、可执行性程序等。调度目标层主要通过产品核心相关节点进行执行控制。

2.2.2 核心架构



TASKCTL 自动化技术标准产品采用典型的 CS 模式，应用层为客户端，控制层为服务端。同时，服务端完成对目标层的调度控制。

- ✓ 应用层应用层从功能的角度，主要分 admin，designer,monitor。从应用渠道的角度，又分桌面客户端渠道与后台字符界面客户端渠道。同时，为了进一步方便用户，系统服务端还提供了丰富的控制操作行命令。
- ✓ 控制层是多级金字塔架构，顶层为服务控制节点，完成各种调度服务控制以及为客户端提供各种操作应用服务。而代理层代理层完成与目标服务器（ETL 等）的控制交互。另，代理层通过主从代理级联方式，可实现对集群部署的服务器进行调度控制，实现负载均衡等。

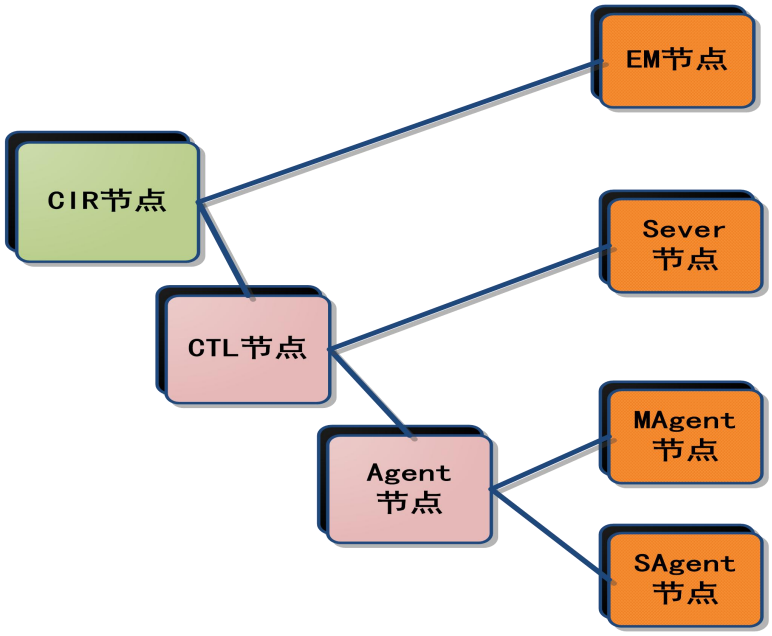
目标层，是整个产品所控制的标的，比如我们的 ETL 服务器，作业工作站等。

3 核心节点

3.1 节点分类

由产品核心架构得知，TASKCTL-CIR 调度平台核心体系是一个金字塔体系，并由多种不同类型的节点构成，这些节点类型包括：EM、Server、MAgent 以及 SAgent，其中 EM 为金字塔尖根节点，其下分别级联 Server、MAgent、SAgent。

在 TASKCTL 概念体系中，这四类不同节点，虽有不同作用与功能，但概念上又有一定的统一。以下我们通过节点分类图进行说明：



核心节点分类结构图

由上图可知，核心所有节点都可归类为 CIR 节点，CIR 节点包括 EM 节点与 CTL 节点；而 CTL 节点又分 Server（控制节点）与 Agent 节点；Agent（代理）节点又分 MAgent（主代理）节点与 SAgent（从代理）节点。

3.2 节点间通信

产品核心是一个无数据技术平台，平台不同层节点之间不存在类似数据库一样的信息共享区，所有信息均通过 Socket 通信进行交互。具体实现时，节点之间通

信按传统交易方式进行，并通过交易码来区分每次通信的类型。

整体上说，产品核心节点之间 Socket 通信具有以下两个特征：

（一）短连接方式

目前整个平台每笔交易均采用短连接方式，即每笔交易都产生一次 connect 动作，这不仅体现在客户端与核心 EM 节点之间的通信，同时，也体现在产品核心所有节点之间的通信。连接方式实质上只是一个技术问题，短连接虽然使系统构建简单且稳定，但同时会牺牲一定的通信效率。

（二）通信内容透明性

为了增加系统的可维护性以及信息的可跟踪性，产品核心的交易通信信息是透明的，用户可以跟踪该信息。如下图所示：

实际应用中，用户可以通过跟踪交易通信信息，不仅可以了解平台更多的处理过程，而且还可以了解平台更多的关键信息内容。

另外，我们需要明确一点：交易是用于客户端与核心以及核心不同节点之间的通信，针对每笔交易通信来说，存在客户端与服务端的区分，但从整个产品核心节点角度来说，核心节点之间是对等的，每个节点都可以向其它节点发起服务请求，每个节点既是客户端，也是服务端。从这一点，也可以说明，我们为什么将核心所有节点作统一的分类。

3.3 节点灵活部署

产品核心是一个多层逻辑体系，具体通过在不同逻辑层部署不同节点来展示。但实际应用中，这种多层结构不是固定不变的，用户可根据项目的规模与需求对产品核心进行灵活部署，整个体系即可简单，也可复杂。另外，产品核心节点既是逻辑的，也是物理的，不同节点即可以在相同主机上部署，也可以在不同主机上部署。

为了对产品核心的灵活部署特征有更深入的了解，本节分别对单服务、多服务以及负载均衡等不同部署模式进行详细描述。

3.3.1 部署基本原则

在讲解不同部署之前，我们首先需要明白以下两点部署基本原则：

（一）环境独立原则

环境独立原则指相同类型的节点只能部署于独立的用户环境，也就是说，两个 CTL 节点不能同时部署在同一主机同一用户环境下，但不同类型的节点可以部署于相同环境，比如 EM 节点与 CTL 节点，可以同时部署在同一主机同一用户环境之下。

（二）节点内置原则

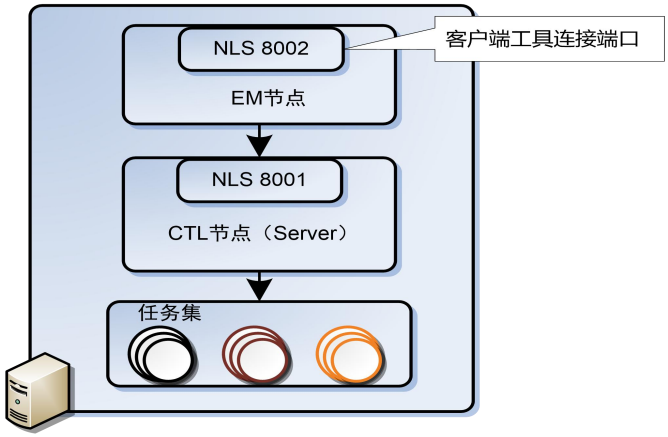
由前面相关章节我们得知，核心节点分 EM、Server、MAgent 以及 SAgent 四类节点，而 MAgent 与 SAgent 统称为 Agent 节点；Agent 与 Server 统称为 CTL 节点。在 TASKCTL 设计中，一个 Server 节点内置一个 MAgent 节点；而一个 MAgent 又内置一个 SAgent 节点。因此， 对于部署了 Server 节点的用户环境，就不能再部署 Agent 节点，并且，Server 节点同样可以完成任务的执行代理功能。

3.3.2 单服务部署

单服务部署指整个平台只部署一个调度服务节点。部署结构相对比较简单。单服务我们又分单机部署和网络部署，以下我们分别对这两种部署进行描述。

（一）单机部署

以下是单机部署示意图：



由上图可知，这种部署实质就是将调度服务节点与任务程序集部署于同一主机，即将调度平台所有节点部署在同一 ETL 服务器之上。部署说明如下：

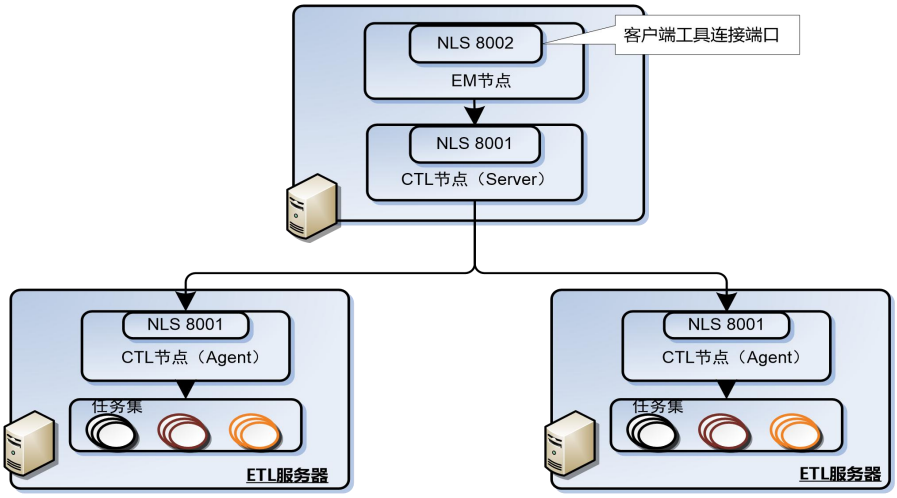
（1）EM 节点与 Server 节点既可以部署在同一用户环境，也可以部署在不同用户环境

（2）由于 Server 节点直接部署在 ETL 服务器之上，利用节点内置原则，我们就不用部署代理节点即可完成对任务实际控制。

在实际应用中，单机部署既是最简单的部署模式，也是应用最为广泛的部署模式。目前，国内 ETL 应用中，调度多由不同项目独立控制，且只有一个 ETL 服务器。因此，单机部署成为首选部署模式，不仅部署简单，也便于维护。

（二）网络部署

以下是单服务网络署示意图：



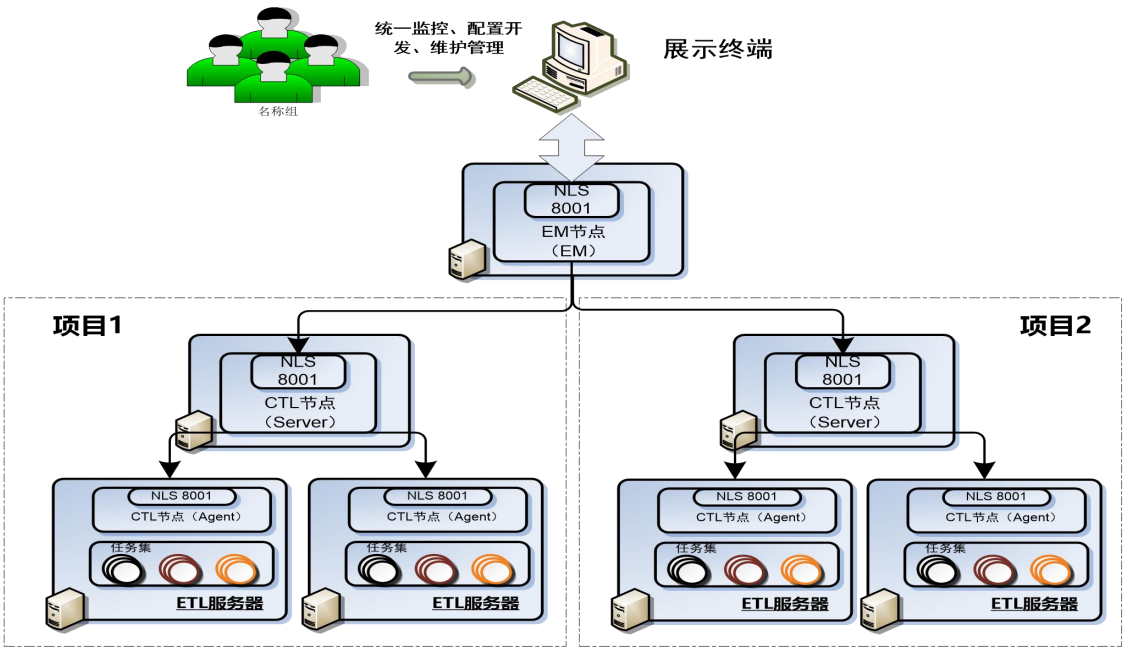
由上图可知，单服务网络部署实质就是将调度服务节点与 ETL 服务器分开部署，并通过在 ETL 服务器上独立部署代理节点来完成任务的执行控制。这种调度方式，业界一般称为远程调度。

远程调度也是调度比较普遍的调度方式。这种方式主要应用于规模较大的项目，整个 ETL 项目需要多个 ETL 服务器且各个 ETL 服务器的任务存在一定的运行关系，对于这种需求，TASKCTL 采用网络部署模式，调度服务器只负责 ETL 流程的逻辑调度，并通过部署在远程 ETL 主机上的代理完成任务的最终执行控制。

3.3.3 多服务部署

多服务部署指整个平台部署多个调度服务器，并通过一个统一的应用平台进行应用管理。该部署模式节点较多，相对比较复杂。但是，对于一家企业来说，该部署模式才真正实现企业统一调度控制，将企业各个项目的 ETL 调度统一构建在一个 TASKCTL 调度平台之上，实现统一调度、管理以及一些与 ETL 控制相关的统计分析等，使企业整个 ETL 体系结构更清晰、更透明、更易控制与管理。

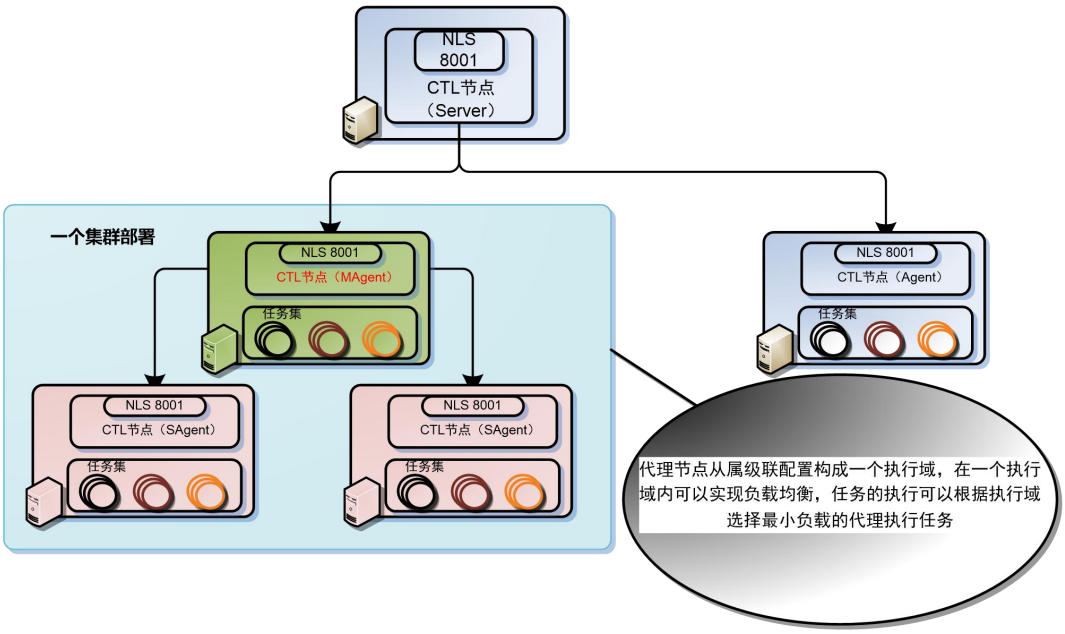
该部署模式如下所示：



企业统一调度平台

3.3.4 负载均衡部署

负载均衡部署是为了有效利用物理资源，并提高 ETL 处理效率。它主要通过代理的级联方式实现。如下图：



由上图得知：负载均衡是相对一个集群而言，即在一个执行代理级联构成的执行域内实现负载均衡。

对于一个集群内，要求每个 ETL 服务器上的任务部署一样。调度产品核心根据集群内 ETL 服务器资源使用情况，将任务自动分配到相对空闲的 ETL 主机并执行任务程序。

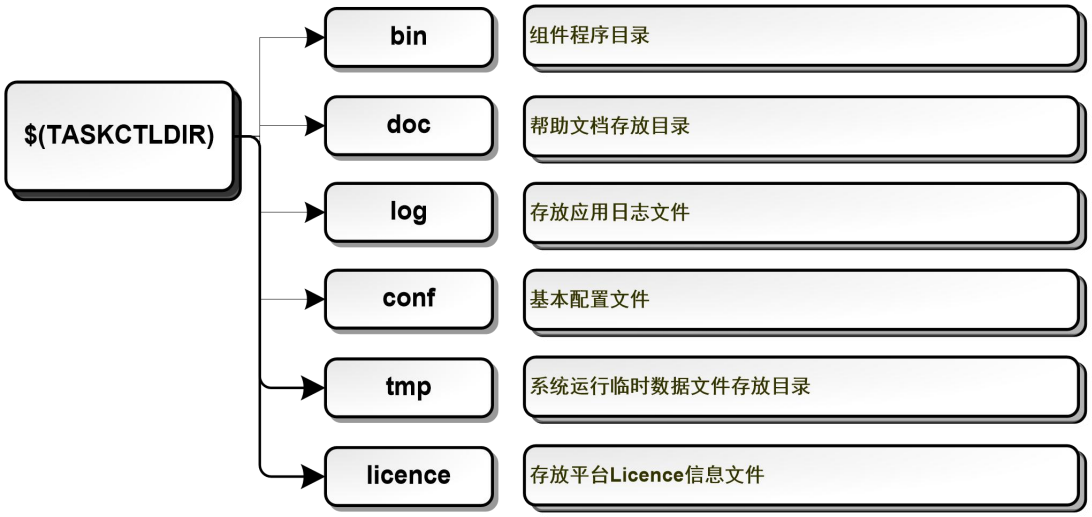
4 EM 节点

EM 节点是产品核心的必须节点，它对整个平台主要起以下几点作用：

- ✓ **对外唯一访问接口：**EM 是客户端以及外围访问调度平台信息的唯一入口节点，它起到客户端与产品核心的通信桥梁作用。
- ✓ **访问权限控制：**对于不同版本或用户，客户端对核心信息访问是有一定权限控制，该控制主要由 EM 节点完成。
- ✓ **核心管理与控制：**产品核心不仅有多个节点，而且每个节点还有多个组件，EM 管理并控制这些节点组件的有效运行。另外 EM 还对整个平台做一些日常管理工作，比如日志清理等。

4.1 目录结构

EM 节点物理目录结构如下：



- ✓ bin：EM 节点组件服务程序以及相关可执行程序存放目录。主要组件程序包括：emnls（EM 节点监听程序）、emkmi（核心管理组件）等。
- ✓ doc：帮助文档存放目录。该目录目前不仅存放与 EM 节点相关的帮助文档，同时还包括整个后台帮助文档。
- ✓ log：日志目录。当相关组件运行异常时，会生成相关组件的日志信息。对于 EM 节点，只生成 EM 节点相关组件日志。
- ✓ conf：存放平台总控信息文件 ctlconf.xml，该文件主要包括平台节点信息、

节点关系信息、任务类型信息以及全局变量信息等。该信息已可以说是平台初始化信息并由用户配置生成。

- ✓ tmp：存放一些在运行过程中产生的临时文件。该目录下的文件用户不能轻易删除。一般情况下，这些临时文件平台会自动清理，但我们不排除在系统异常时，该目录下会累积一些永久性文件，此时，需要系统管理人员对该目录做手动清理。
- ✓ licence：该目录存放平台的 licence 文件，文件名称为 taskctl.lic。平台 EM 节点主要根据该文件对用户做一些访问限制管理以及核心相关权限限制等。

4.2 节点组件

EM 节点目前主要包括 emnls、与 emkim 两个服务组件，它们分别起不同的作用。

（一）emnls 服务组件作用

emnls 服务组件作用包括：

- ✓ 监听作用：监听包括客户端监听，以及核心其它节点的访问监听。
- ✓ 通信转发：比如一些调度控制操作处理，具体处理需要在 server 节点完成。此时 EM 节点就起到客户端与调度服务器信息桥梁作用，EM 自身不完成相关服务处理。
- ✓ 基本数据服务：为客户端提供一些与调度无关的数据服务，比如核心节点信息等。

（二）emkim 服务组件作用

emkim 服务组件作用包括：

- ✓ 平台状态管理：EM 节点对平台其它节点的有效性管理主要通过该组件完成。
- ✓ 日志清理：按用户要求定期清理一些平台运行日志。

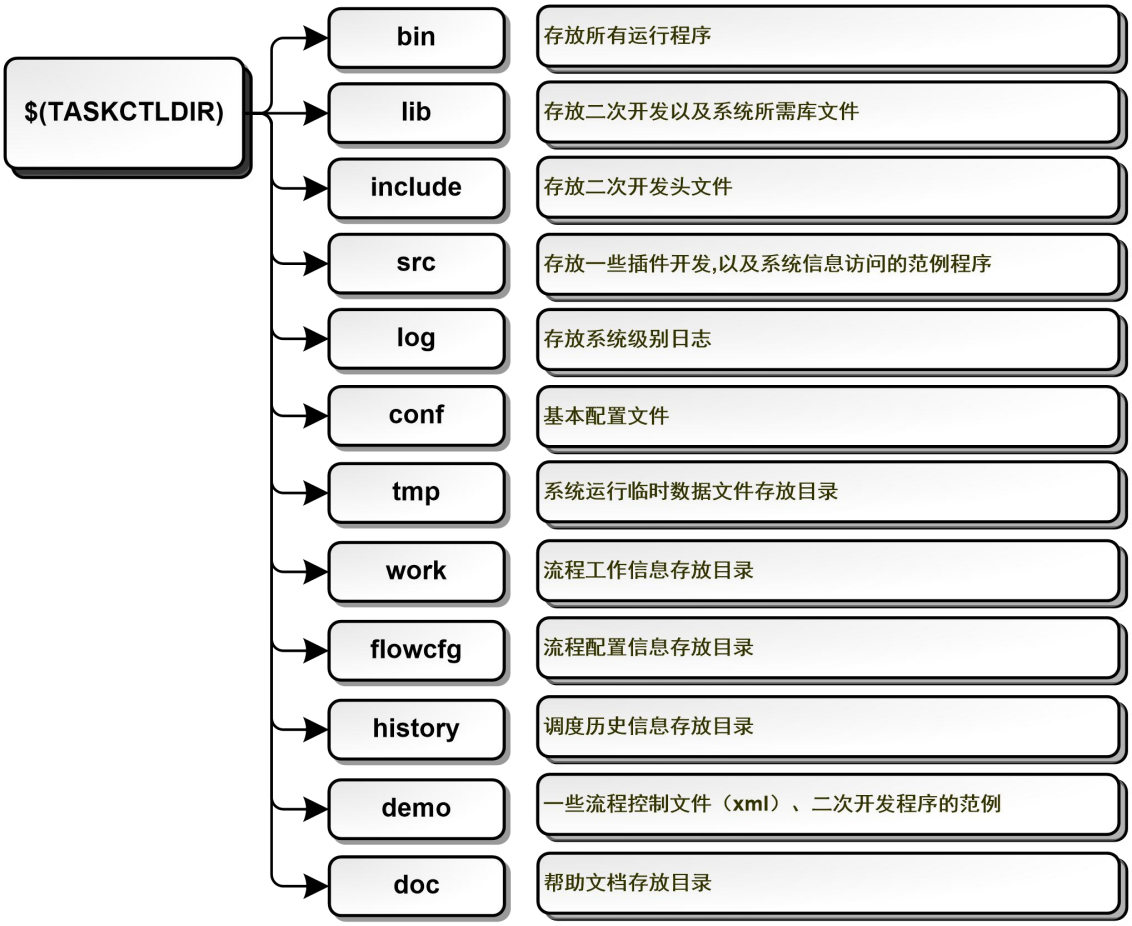
5 CTL 节点

CTL 节点是核心调度工作的主要实现节点，它分调度服务节点(Server)与代理节点(Agent)，Server 主要完成调度控制，而 Agent 主要完成任务的执行。

在 TASKCTL 设计中，不同类型的 CTL 节点，都是同样的安装与同样的程序，类型区别主要通过用户初始化配置决定。在初始化配置中，用户需要设置每个 CTL 节点的类型属性，这些属性包括：Server(调度服务节点)、MAgent(主代理节点)以及 SAgent(从代理节点)。也正因为此，TASKCTL 概念体系中将 Server 节点与 Agent 节点统一称为 CTL 节点。

5.1 目录结构

CTL 节点物理目录结构如下：



- ✓ bin：CTL 节点组件服务程序以及相关可执行程序存放目录。具体组件程

序在下一章节会详细讲解。另外，该目录对于代理节点，还存放各种任务类型的插件驱动程序。

- ✓ doc：帮助文档存放目录。与 EM 节点一样，该目录目前不仅存放与 CTL 节点相关的帮助文档，同时还包括整个后台帮助文档。
- ✓ log：日志目录。与 EM 节点一样。
- ✓ conf：与 EM 节点一样，存放平台总控信息文件 ctlconf.xml。但该目录下 ctlconf.xml 文件由 EM 节点统一控制管理管理，包括修改同步等。
- ✓ tmp：与 EM 节点一样。

以下目录只针对服务节点有效：

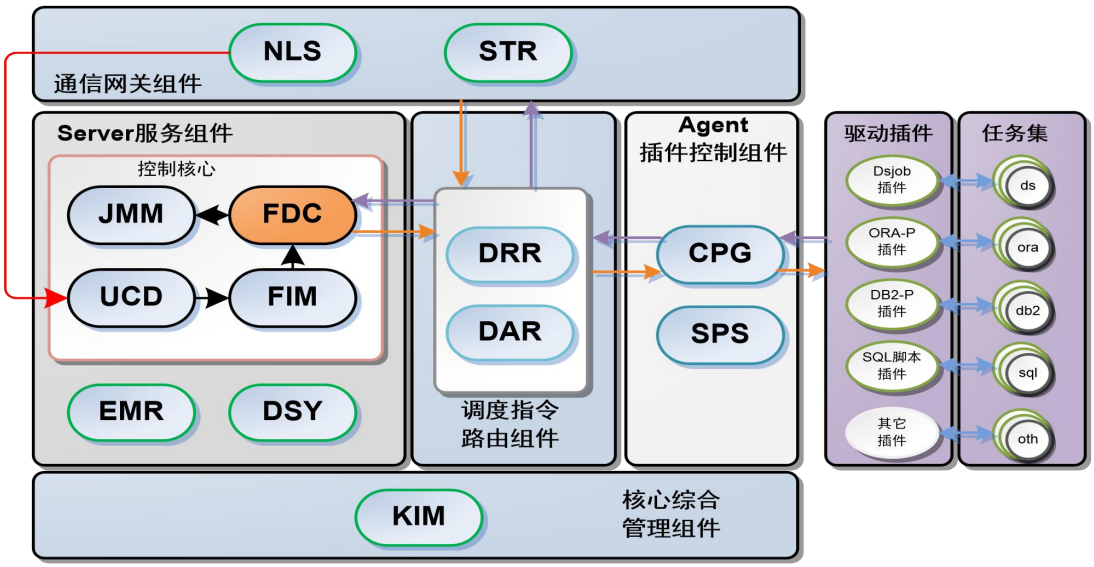
- ✓ lib：存放二次开发相关静态库
- ✓ include：存放二次开发相关头文件
- ✓ src：主要用于存放二次开发的源代码（但实际由用户决定代码的存储位置）
- ✓ demo：存放平台二次开发的一些范例程序，用于指导用户二次开发。
- ✓ flowcfg：该目录称为流程开发区，存放用户开发流程核心信息。
- ✓ work：该目录称为流程工作区，存放活动流程信息。
- ✓ history：存放流程调度历史信息。

提示： 开发区与工作区是 TASKCTL 产品核心的重要概念，该概念在“ 3.5.3 调度服务（Server）”章节会作相关讲解

5.2 CTL 节点组件

5.2.1 组件逻辑架构

CTL 节点由多个具有不同功能的组件构成，主要可以分为三类：公共组件、Server 服务组件与 Agent 代理组件。而公共组件又分为通信网关组件、路由组件与管理组件。不同组件逻辑关系如下：



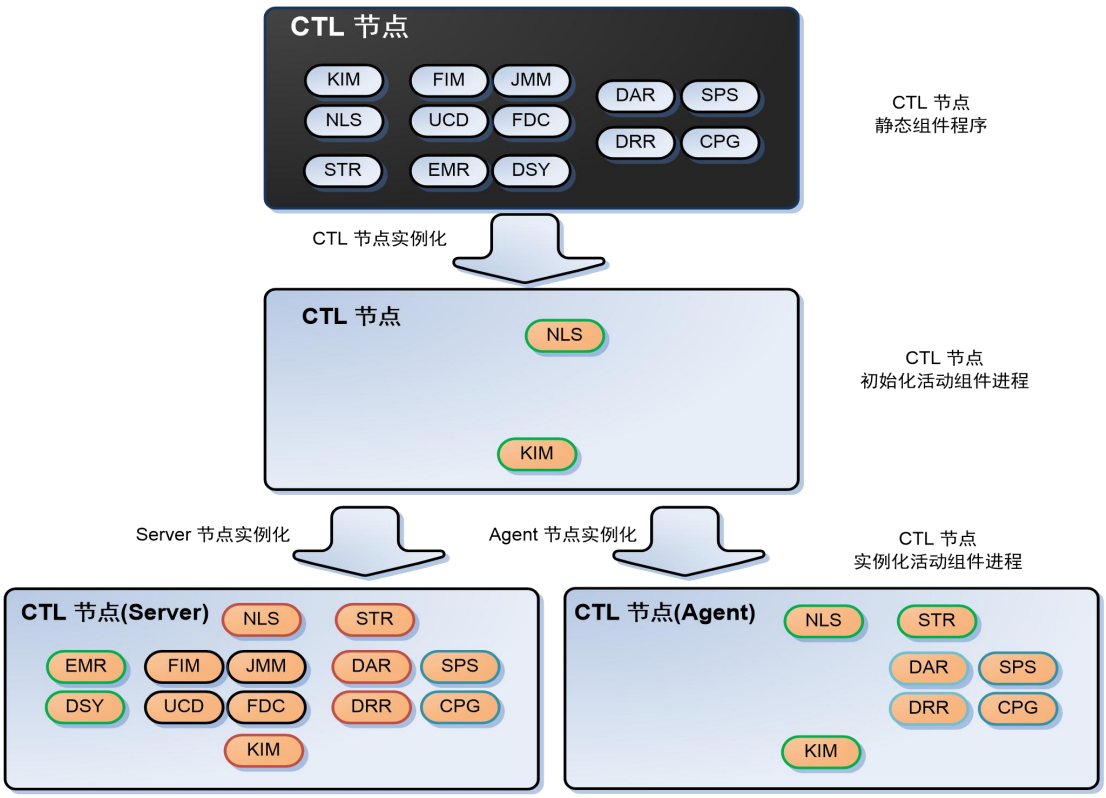
CTL 节点各组件说明如下列表：

分类	名称	名称说明	功能说明
公共	KIM	Kernel Integrate Manange	核心综合管理组件
	NLS	Net Listten	监听组件(接收器)
	STR	Send Message To Remoto	远程信息发送组件（发送器）
	DRR	Dispatch Request Router	请求指令路由器
	DAR	Dispatch Answer Router	响应指令路由器
Server	FDC	Flow Dispatch Core	流程调度控制核心
	FIM	Flow Instance Manange	流程实例化管理器
	JMM	Job Mutex Manange	互斥任务管理器
	UCD	User Command Deal	客户端用户请求命令处理器
	EMR	Kernel Event Manange And Release	核心事件管理与发布
	DSY	Data Synchronous	数据同步处理器
Agent	SPS	Search Plugin State	插件状态查询管理器
	CPG	Call Plugin	插件执行调用管理器

5.2.2 节点实例化

前面我们已经提到，CTL 节点分 Server 节点与 Agent 节点，所有 CTL 节点的物理结构一样，只有当服务节点在启动时才确定是 Server 节点或 Agent 节点，也就是说，是 Server 节点还是 Agent 节点是节点启动实例化所决定的。

CTL 节点启动实例化过程如下图所示：



由上图可知，一个 CTL 节点在实例化过程中有三种状态变迁，这三种状态分别为：静止状态、初始化状态、实例化完成状态。

（一）静止状态

静止状态指 CTL 节点无任何服务组件启动，此时不能与任何其它节点通信。

（二）初始化状态

初始化状态指节点的服务监听已经启动，表示可以与 EM 节点通信。在该状态下，CTL 节点实际已经启动两个组件，即 ctlnls 与 ctlkim，此时不仅可以通信，已通过 ctlkim 组件完成基本的节点管理工作。

对于 CTL 节点的初始化，需要用户到节点所安装的指定环境通过 ctlinit 命令人工操作完成。

（三）实例化完成状态

在 ctlconf.xml 配置中，已经配置了每个节点的节点属性（即是什么类型的节

点），节点在实例化时，首先启动公共组件，其次根据配置中的节点属性启动相应的组件，如果是 Server 节点，启动 Server 与 Agent 相关组件，如果是 Agent 节点，启动与 Agent 相关组件。

对于节点实例化，不需要用户到指定环境操作，因为节点初始化后，监听组件已经启动，并建立了对外的通信渠道，用户可以通过客户端软件 Admin 相关命令完成节点实例化工作。

提示：

Server 节点实例化除了启动公共组件与 Server 相关组件之外，还启动 Agent 相关组件。正是这种机制体现了节点内置原则，即一个服务节点内置一个代理节点

5.2.3 组件间通信

如前面相关章节所言，产品核心不仅由不同功能的节点构成，而每个节点又由功能不同的多个组件组成，节点之间通过 Socket 完成通信，而每个节点组件之间又通过消息队列方式完成通信。

为了使组件之间的通信机制更完善，我们在消息队列的基础上封装设计了同步通信与异步通信。对该机制的了解，首先便于我们对核心工作机制的进一步了解，其次便于我们在系统维护中对一些重要通信信息的跟踪与维护。

5.2.3.1 请求队列与响应队列

为了有效实现组件进程间同步通信与异步通信，我们在物理消息队列的基础上，为每个组件进程分别逻辑分配了请求队列与响应队列。

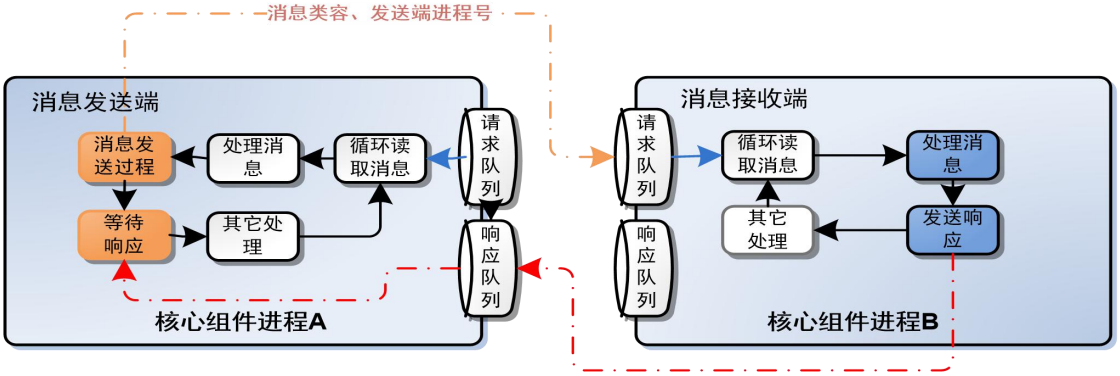
- ✓ **请求队列：**接收其它组件进程请求消息的队列，此时，当前进程为提供服务的服务端。
- ✓ **响应队列：**接收其它服务进程的响应信息的消息队列，此时，当前进程为请求服务的客户端。

由于每个进程都有请求队列与响应队列，说明每个进程即可提供服务，也可请求服务，当提供服务时，组件是服务端；当请求服务时，组件是客户端。这种

特征与产品核心节点间通信机制类似，核心节点之间通信是对等的，同样，节点内组件之间通信也是对等的。

5.2.3.2 同步通信

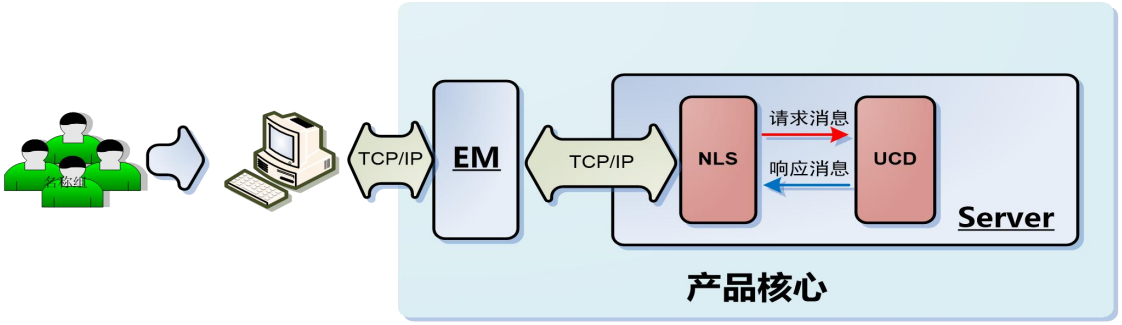
以下是组件间同步通信示意图：



进程间同步通信

由上图可知，同步通信指请求进程发送服务请求后，需要等待服务进程的响应。

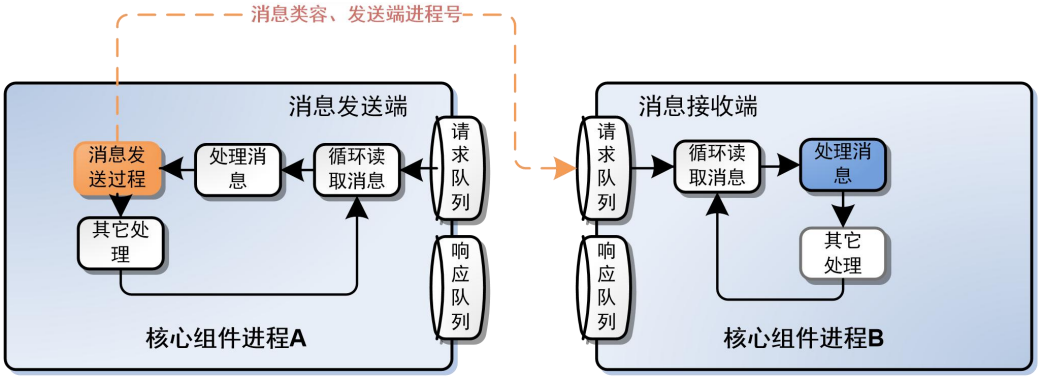
同步通信在核心主要用于 UCD（用户命令处理器）接收用户指令处理，下图展示了一个用户流程启动的处理过程：



上图中 UCD 得到来自 NLS 的流程启动指令后，执行启动处理并将处理情况返回。在此处理过程中，NLS 与 UCD 之间就采用消息同步通信。

5.2.3.3 异步通信

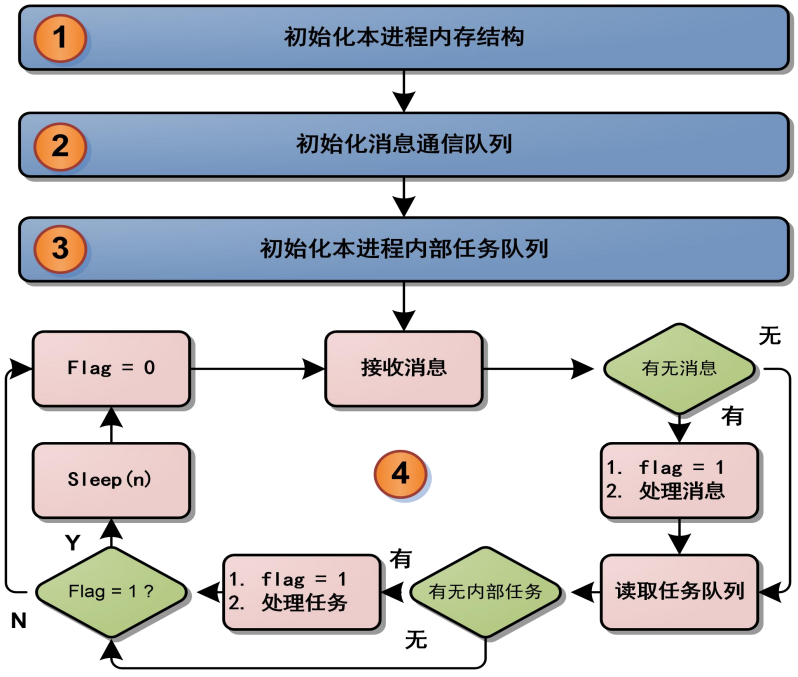
以下是组件间异步通信示意图：



进程间异步通信

由上图可知，异步通信指请求进程发送服务请求后，无需等待服务进程的响应。异步通信是产品核心节点组件间主要通信手段。实际上，核心组件均采用消息驱动模式，使原本需要同步通信解决的问题均可采用异步通信的方式。比如我们核心调度指令，FDC 调度核心进程只管发起执行指令而无需阻塞等待结果（关于调度指令我们在下文有专门论述）。

以下是我们组件进程的基本结构：



通过组件进程基本结构，我们可以看出，进程主要是由消息驱动。对于进程间消息互动，不论请求还是响应，我们均视为组件的外围信息，均可以异步方式进行处理。

5.3 调度服务（Server）

调度服务节点是 CTL 节点实例化后具有实际功能的节点，它是产品核心中最重要的处理节点，负责核心最关键的任务调度工作。

整个调度服务节点涉及组件较多，功能也比较丰富。本文主要通过以下几个角度对调度服务节点进行描述：

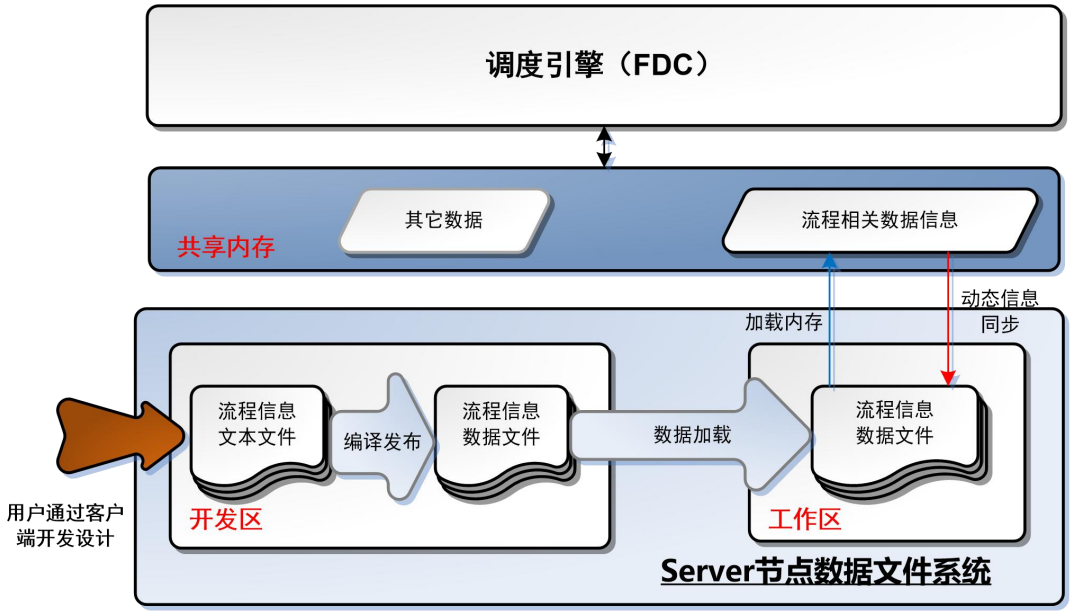
- ✓ **核心数据：**核心数据指我们用户设计的流程信息，它是产品核心调度处理的主要信息来源与处理依据。该数据主要由调度服务节点进行管理。
- ✓ **数据同步：**TASKCTL 是无数据库设计，为了保证平台信息的连续性，对核心调度过程中产生的动态信息，主要通过数据同步机制将动态信息以文件的方式永久固化。比如任务的执行状态信息，数据同步机制保证了系统异常退出重启后，还能有效根据相关状态值实现流程的断点续传功能。
- ✓ **调度引擎：**调度引擎是整个产品核心最关键的进程，该进程主要功能是：判断流程任务是否可以执行。同时，该进程也体现了产品核心技术概念 CIR 中的 Control 概念。
- ✓ **调度指令：**调度指令指与调度相关的一组指令，该组指令是核心产品不同组件间交互的核心信息，通过了解该指令，不仅可以了解调度引擎与其它组件的关系，同时也可进一步了解产品核心调度的主要工作机制。
- ✓ **核心事件管理与发布：**核心事件与调度本身无关，但与我们二次特色开发息息相关，该机制主要展示了核心的动态变化如何适时与外围交互。

5.3.1 核心数据

核心数据即流程信息，是调度的主要信息来源，其内容主要包括：流程总控信息与流程核心信息：

- ✓ 流程总控信息：主要包括流程基本信息、流程变量、模块信息等
- ✓ 流程核心信息：主要包括任务基本定义信息与流程调度控制策略信息等

调度核心为了对该信息有效管理以及使用，主要通过开发区、工作区、以及共享内存对其分别管理，其关系如下：



由图可知以下重要信息：

（一）数据以文件方式存储

调度平台整个核心数据以文件方式存储，并分别存储在开发区与工作区。

- ✓ **开发区：**主要面向用户开发使用。开发区在文件系统中指向目录为：$TASKCTLDIR/flowcfg
- ✓ **工作区：**主要面向调度核心使用。工作区在文件系统中指向目录为：$TASKCTLDIR/work

（二）数据以两种方式体现

数据两种体现方式指：文本文件方式与数据二进制文件方式。文本文件是可读的，用于用户开发编写；而数据文件是用户不可读的，用于核心调度。

数据文件是经过文本文件编译得到。

（三）整体数据流

从图中可以看出，整体数据流为：

第一步：用户通过工具操作开发区文本数据文件，即流程设计与开发。

第二步：通过用户编译，如果成功，将相应流程文本文件转换成数据文件并存储于开发区。

第三步：调度核心在需要新流程信息时自动（也可由用户加载）从开发区加载数据文件到工作区，并同步到共享内存。

第四步：调度引擎从共享内存使用相关流程的数据信息。

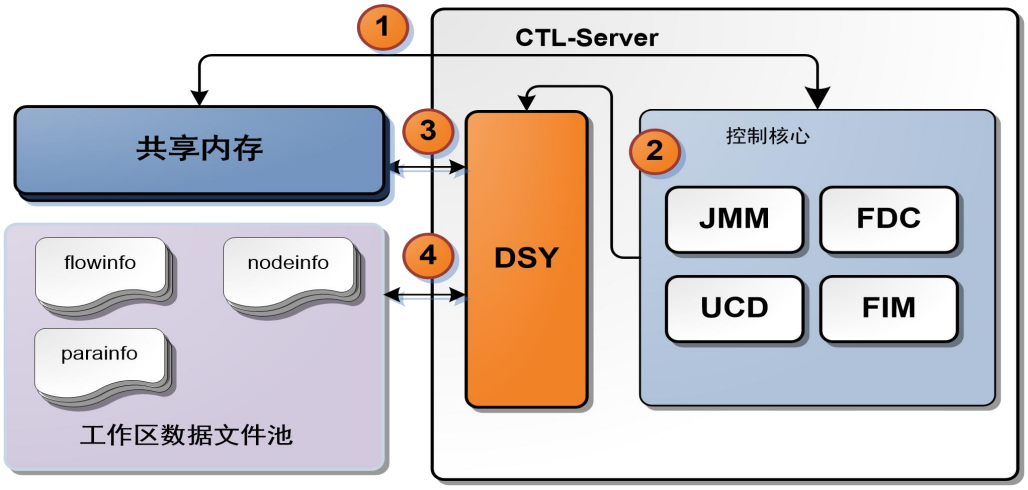
（四）调度引擎直接操作共享内存数据

由图可知，调度引擎使用流程信息时，并不直接读取工作区数据，而是从共享内存读取。共享内存技术在很多工具平台都有使用，在此不做过多论述。

提示： 关于流程信息具体内容详细信息及其内容组织规则请参考《TASKCTL-CIR 2.1 流程与模块代码》

5.3.2 数据同步处理

数据同步机制是产品核心重要机制之一。上一节讲到流程信息的存储与主要流向，但主要针对静态数据信息，一个完整的调度数据体系管理还包括运行动态数据存储。为了保证调度动态数据能适时保存，同时又不影响相关功能组件的运行效率，TASKCTL 核心采用数据同步处理机制，主要通过专门的同步进程组件 DSY 实现，其工作示意图如下：



由上图得知，数据同步主要通过以下几个步骤完成：

第一步： 控制核心直接读写共享内存

第二步： 控制核心在写共享内存时，发送同步消息给 DSY 进程

第三步： 同步进程收到消息后，根据消息从共享内存读取变更数据。

第四步： 同步进程将变更数据写到工作区相关数据文件。

提示： 流程信息数据文件主要由基本信息 flowinfo、任务节点及其控制策略信息 nodeinfo 与流程私有参数信息 parainfo 三种文件构成

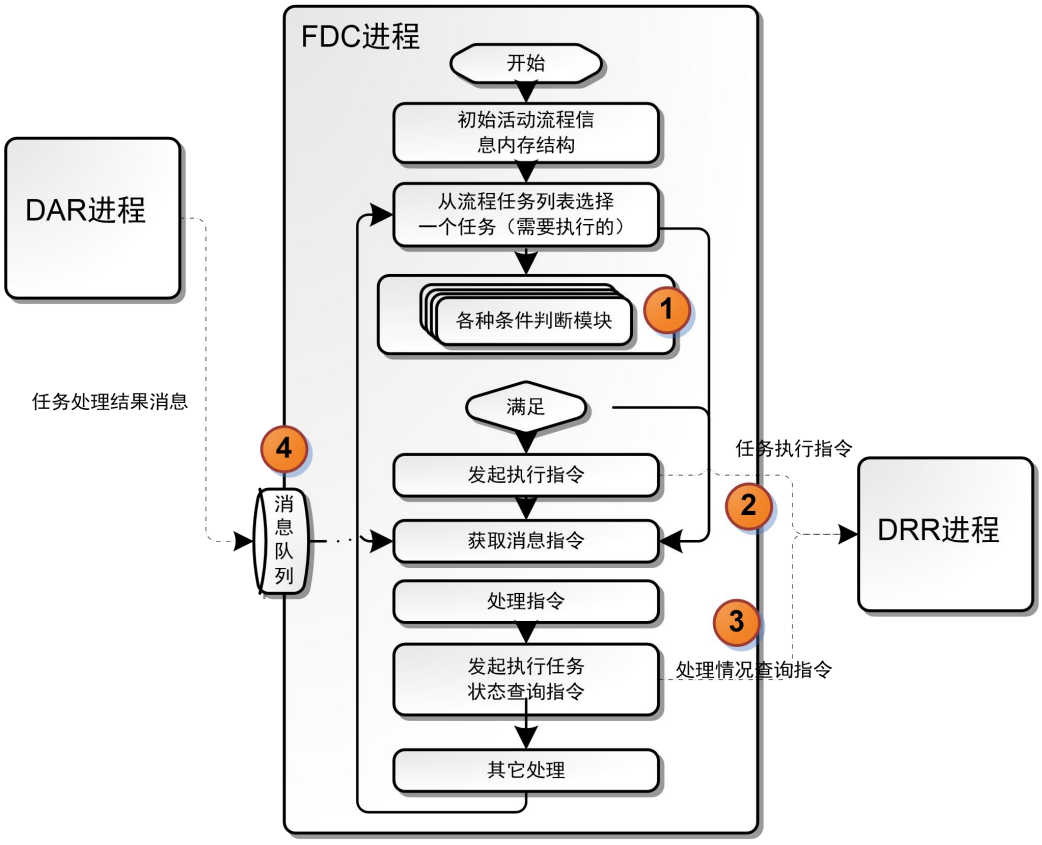
5.3.3 调度引擎-FDC

调度引擎即组件进程 FDC，是产品核心最重要的进程。该进程主要判断流程任务是否可以执行。同时，该进程也体现了我们核心技术概念 CIR 中的 Control 概念。

另外，FDC 进程在与其它组件进程相比，是一个有特殊处理机制的进程，该特殊性表现为：其它进程在核心平台启动时就启动，而且在一个 CIR 节点中，每个组件进程是惟一的，而 FDC 进程的启动与退出，是由用户单独通过客户端相关命令实现，同时，该进程在一个 Serever 节点内并不唯一，每启动一个流程就会启动一个与之对应的 FDC 进程。

一旦一个流程的 FDC 调度引擎启动，如果没特殊需要，FDC 进程不用退出。它会持续不断控制着相关流程每一批次的调度处理。

调度引擎 FDC 主要处理流程如下：



以上是调度引擎主要处理流程，整个流程相对复杂，但只要关注图中标识的四个重要处理步骤，即可了解调度引擎的处理过程。以下分别对这四个重要处理

步骤进行说明：

（一）任务各种执行条件判断处理

任务各种执行条件判断处理是整个调度引擎最核心处理步骤，一个流程的任务是否可以执行就通过该步骤确定。比如依赖、互斥、执行计划等各种各样的控制策略。

（二）调度指令发起

调度引擎核心功能是寻找一个可以执行的任务，它不担当具体任务的执行工作，只需在找到需要执行的任务后，将相应执行命令对外发起即可。具体在什么地方执行，调度引擎并不关心。

（三）状态查询指令发起

状态查询指令是相对执行指令而言的，当一个执行指令发出后，调度核心会按一定时间间隔跟踪该任务的执行情况。跟踪主要通过对外发起相关任务执行情况查询指令的方式完成。

（四）接收处理响应消息

执行指令发出，经过一定的渠道，最终由相关代理节点接收该指令，代理接收指令后会执行相关任务，并将处理结果通过一定的线路返回，并以消息的方式通知调度引擎，引擎会自动处理该消息，并最终决定任务的处理情况。

5.3.4 调度指令

调度指令与调度引擎有紧密的联系，而且在上一节中我们已经提到了相关调度指令，比如执行指令、查询指令等。但是，上一节我们只讲到了引擎的指令发起，并未提到这些指令如何到达目的地以及到达后相关的后续动作等，本节着重讲该方面的相关内容。

调度指令主要包括：执行指令、查询指令、停止指令、响应指令，其中执行、查询、指令由调度引擎发起，停止指令主要由用户通过客户端发起，而响应指令是相对前三种指令的响应并最终由调度引擎接收并处理。

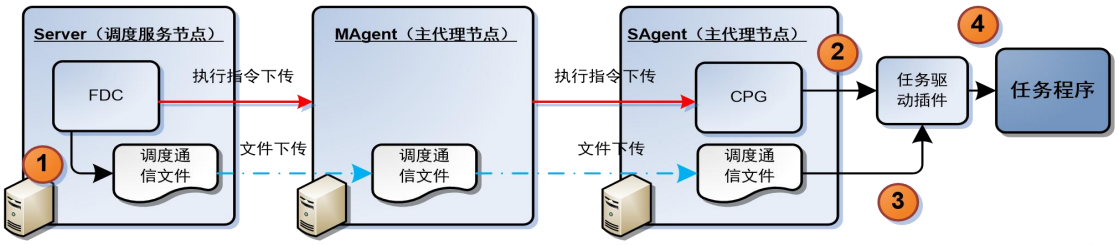
5.3.4.1 关于调度通信文件

在介绍调度指令之前，我们先了解调度通信文件。

（一）通信文件作用

调度通信文件是调度处理机制中的一个重要组成部分，文件中记录一次调度的完成信息：包括任务的完整信息以及调度过程的状态变迁信息。调度通信文件是一个临时文件，是在调度引擎发出指令时生成，并随着一个任务的调度的结束而消亡。

调度通信文件生成后，它随着执行指令下发而下发，即执行指令下传到哪个代理节点，该文件就下传到那个代理节点。从这个角度，我们可以理解通信文件是一个执行指令的真实内容附件 ，如下图所示：



示图说明：

1. 调度引擎发出指令生成相应任务的通信文件
2. 指令指令达到目的后CPG调用相应驱动插件
3. 驱动插件从通信文件读取具体任务信息
4. 驱动插件执行具体任务程序

上图简单说明了一个由 Server 发起的执行命令以及相应通信文件的在不同节点之间的传动过程。

在一个任务的调度生命周期内，其它相关指令在不同节点会与该通信文件产生信息交互行为。

（二）命名规则

该通信文件主要存放在 tmp 目录，命名规则为：

规则：【调度服务节点编号】【流程号】【任务号】【运行模式】.rc

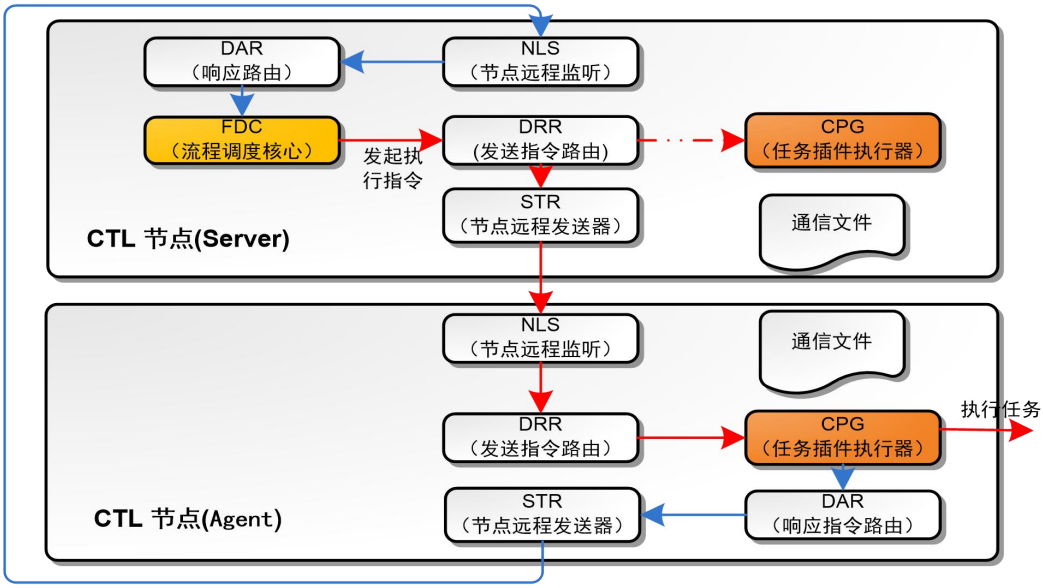
举例： 00200030002_01.rc

- ✓ **调度服务节点编号：**即调度指令发起地 Server 节点的编号，定长三位
- ✓ **流程号：**调度任务所属流程的编号，定长四位
- ✓ **任务号：**当前执行任务编号，定长四位
- ✓ **运行模式：**1-正常模式；2-调试模式

说明： 流程由核心调度时，任务的运行模式为正常模式；当任务通过调试运行时为调试模式。关于流程调试请参考《TASKCTL-CIR 2.1 后台客户端系统》相关章节

5.3.4.2 执行指令

执行指令由调度引擎（FDC）发起，该指令通过每个节点的路由器（DRR）不断传递到最终目的地。如下图所示：



上图表达了一个执行指令的传递过程，其传递过程由路由器根据通信文件中任务的代理属性（执行目的地）进行路由。在命令传递过程中，通信文件中任务执行状态均记录为正在执行。当任务执行完后，执行结果以响应指令的方式，并且根据通信文件记录的路由信息原路返回，每返回一个节点，将任务状态修改为实际执行结果状态。当顺利从一个节点返到下一节点。将本节点通信文件删除。

说明： 路由针对简单的代理配置没多大意义，当针对集群负载均为配置时就充分体现出优势。

5.3.4.3 状态查询指令

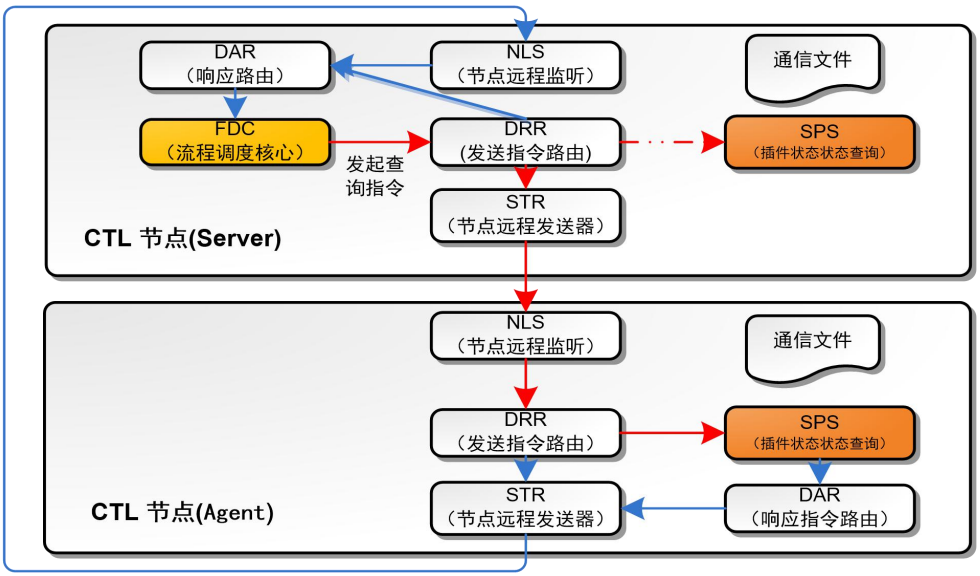
状态查询指令同样由调度引擎（FDC）发起。在讲状态查询指令之前，我们

先明白为什么需要查询状态以及查询状态的意义。

首先，我们的执行指令通信是异步的，FDC 只需要将指令发出即可。在正常情况下，FDC 最终会收到执行结果的消息，但是，如果系统出现某种异常，比如网络问题，响应指令在返回过程中，可能因为系统异常而丢失，从而造成调度核心永远收不到相关执行结果。为了解决该问题，核心采用定期主动查询的方式以解决该问题。

其次，ETL 任务一般执行时间相对较长。任务在目的地长时运行时，作为控制中心的调度引擎主动去”关心”任务的执行情况是一种比较稳定安全的做法。实际上，由于核心采用主动查询的方式，使其变得更为稳定，不会出现一些调度系统’死锁’现象，即使调度过程出现网络异常，也不影响我们的顺利调度。

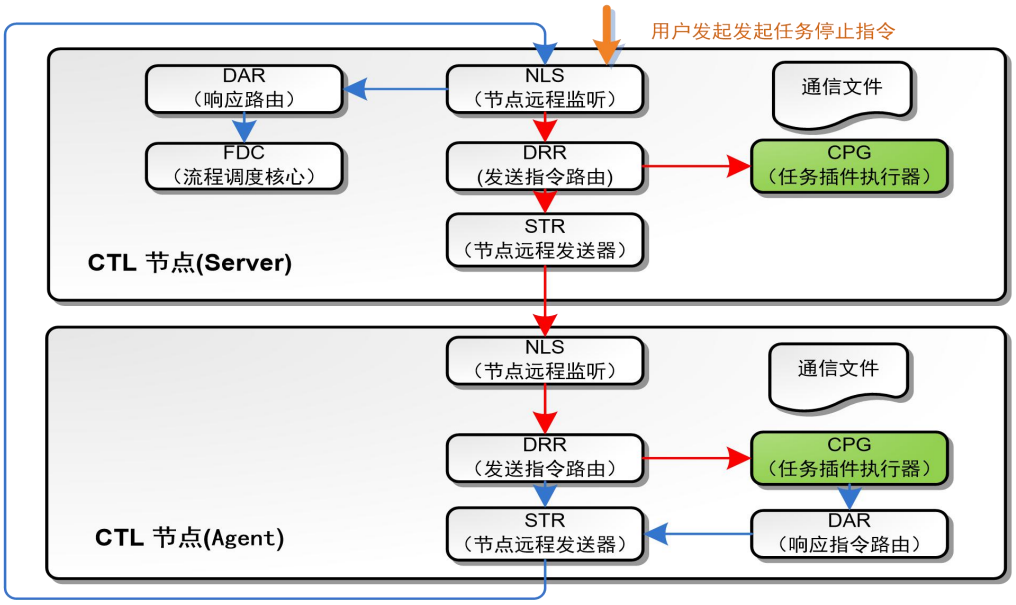
以下是状态查询指令示意图：



状态指令是沿着执行指令线路进行查询，每到一个节点，先查看通信文件中任务的状态，如果是正在执行状态，继续下行，反之，发起响应指令，响应指令从当前节点沿着执行指令线路返回。如果查询指令一直查到任务执行目的地，通信文件中状态依然是正在执行状态，则通过插件状态检测组件 SPS 确定插件的状态，如果插件状态正常，表明任务仍在执行，否则，表明异常，同时发起异常响应指令。

5.3.4.4 停止指令

一般情况下，停止指令由用户发起，该指令用于强行终止任务运行。如下图所示：



停止指令首先由用户通过客户端相关软件发起，经过 EM 将命令传到 Server，并由 Server 相关组件将指令传递到指令路由器，指令路由器一旦收到该指令后，就按查询指令方式下传，直到目的地，并通过停止插件终止相关任务程序。

说明： 调度平台要实现任务终止功能，必须开发并配置相应任务的停止插件。关于停止插件的开发与配置请参考《TASKCTL-CIR 2.1 任务扩展应用》相关章节

5.3.5 核心事件管理与发布

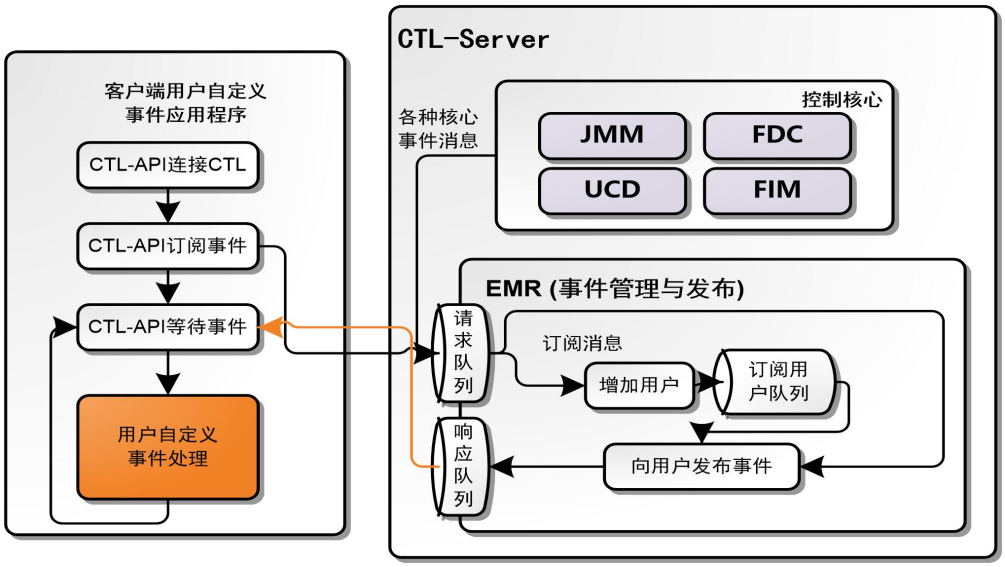
首先，我们需明白什么是核心事件？

核心事件指一切调度核心以及流程各种对象的状态变化，比如系统一些异常事件、调度引擎的退出以及流程的状态变迁与各个任务的状态变化等。

显然，核心事件与调度本身无关，但是，核心事件的管理与发布与我们二次

特色开发息息相关，该机制主要展示了核心的动态变化如何适时与外围交互。比如，核心流程的状态变化，核心以事件的方式对外发布，用户在二次开发程序中，只要通过相应的接口，即可适时捕获该状态的变化以及相关信息。

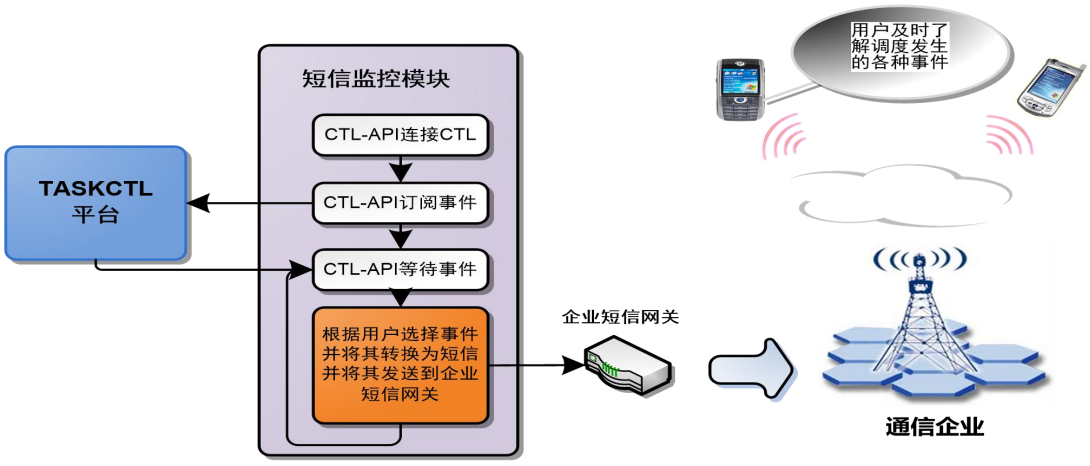
核心事件管理发布机制如下图所示：



由上图可知，事件主要有调度核心产生，而事件的管理与发布主要通过 EMR 组件完成，该组件组要起到核心事件与外围程序的对接作用。它一方面接收核心的各种事件并进行管理，同时接收外围程序的订阅事件，凡是有订阅的用户程序，EMR 就适时向其发布来自核心的相应事件信息。

说明： 关于核心事件的具体应用，请参阅《TASKCTL-CIR 2.1 二次开发》相关章节

核心事件的应用很广泛，我们可以利用该机制轻松开发一些特色应用。以下是我们利用该机制开发短信监控的应用示意图：



由上图得知：我们通过开发一个程序去订阅核心事件，并将这些事件通过短信的方式发送给相关监控人员，使监控维护人员对系统的监控更轻松方便。

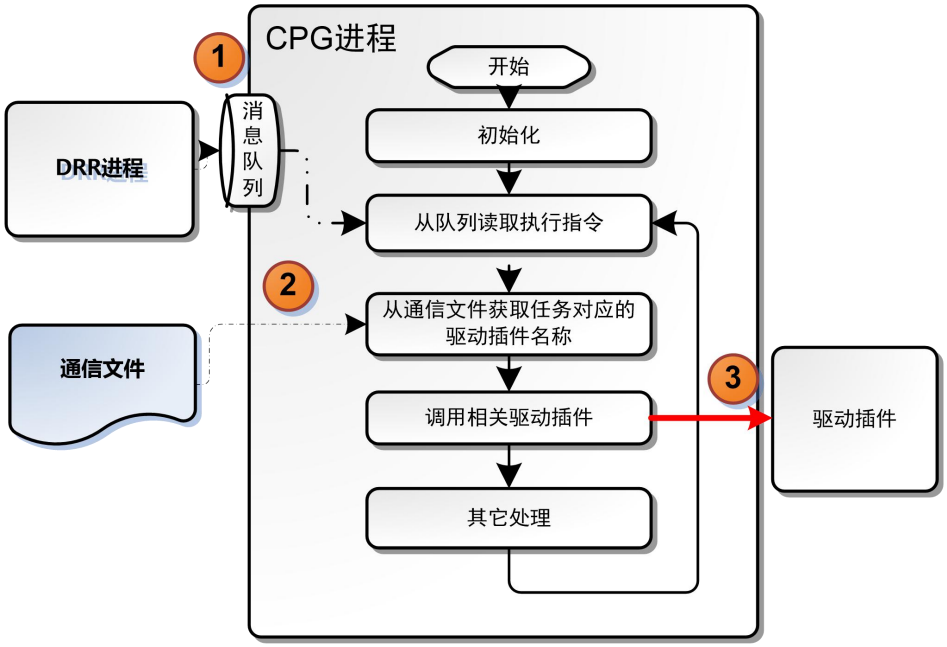
5.4 执行代理（Agent）与任务驱动

执行代理是 CTL 节点实例化后具有实际功能的节点，该节点主要起到对任务的执行与控制。同时执行代理在有集群部署时，主代理还起到负载均衡的控制作用。

5.4.1 执行代理

执行代理主要通过代理节点 CPG 组件完成。该组件不真正执行任务，而只是接收执行指令后，通过统一接口调用相应驱动插件完成对任务的具体执行。

以下是 CPG 任务执行主要处理流程：



由上图可知：CPG 接收执行指令后，只调用相关插件。图中标明的重要三个地方说明如下：

- ✓ **执行指令获取：**CPG 通过消息队列读取执行指令，该信息由路由 DRR 发送过来。
- ✓ **任务对应的插件名称：**该信息由 CPG 通过通信文件获取
- ✓ **插件调用：**CPG 通过非阻塞方式调用插件程序，即 CPG 调用插件后，不必等待插件对任务的执行完毕，而是调用插件成功后立即继续往下处理。

5.4.2 任务驱动

由前面所知，任务的具体执行由插件完成。插件机制是调度平台最重要机制之一，它的引入主要目的是：

（一）任务的灵活扩展

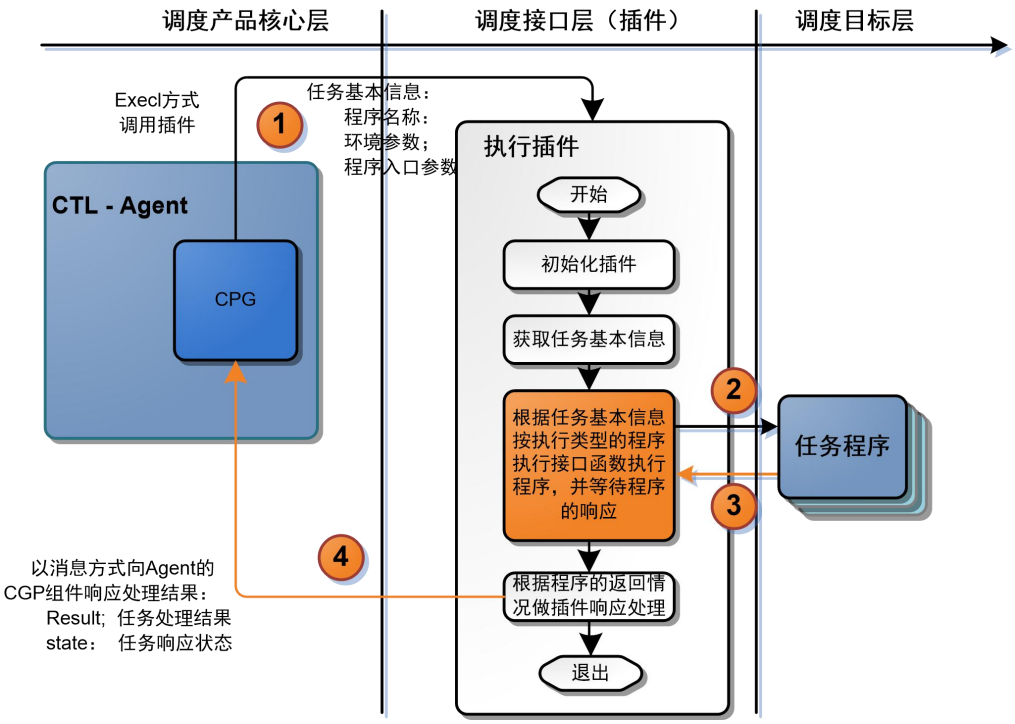
产品核心面对诸如 Datastage、Informatica 以及其它不同类型的任务时，为了保证核心的独立，特意引入插件隔离不同任务的异构性，同时，通过不同任务类型相对应的不同插件实现各种任务类型的灵活扩展。

（二）统一不同任务类型的应用接口

对于 ETL 应用来说，不同任务类型的使用目的都是一样，都是为了完成 ETL 处理工作，只是因人为以及一些其它因素采用了不同技术而已。通过插件，我们可以屏蔽不同任务的应用差异，使不同类型的任务在调度应用时，具有统一的应用接口。这不仅统一了应用，同时也统一了对任务定义，使整个调度应用变得更简单、更清晰。

5.4.2.1 执行驱动

以下是驱动插件处理流程示意图：

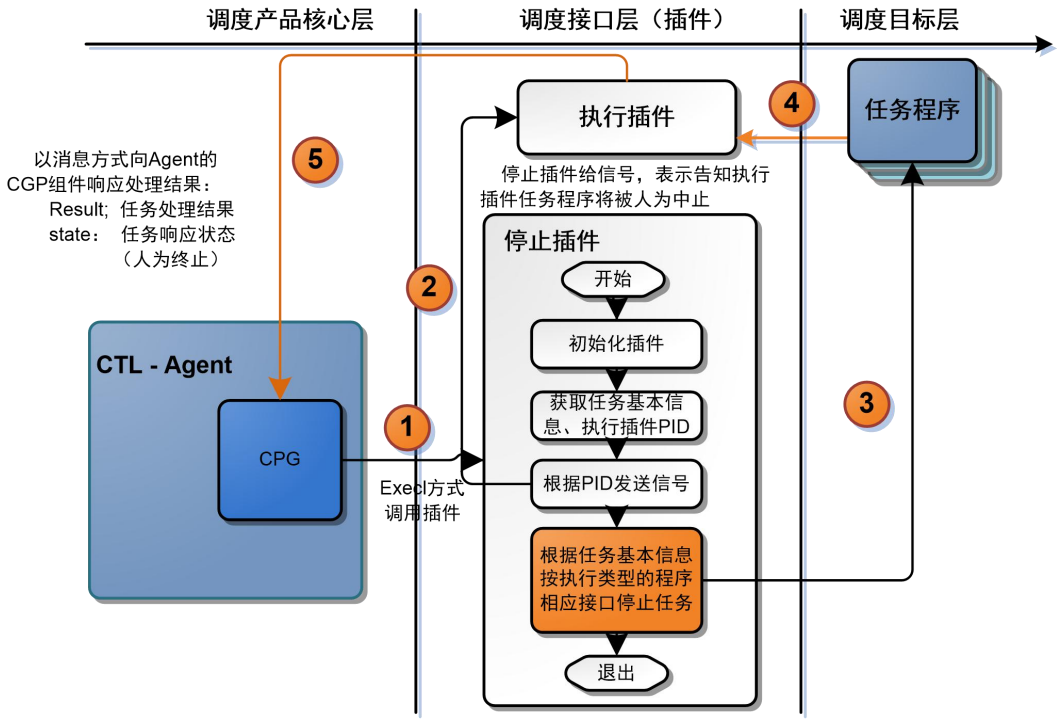


由图可知，插件是一个独立程序，它由服务进程 CPG 调起，在任务执行完后插件程序也相应退出，插件程序的生命周期与实际程序是同生同灭的。

说明： 关于执行插件的开发以及更多信息请参阅《TASKCTL-CIR 2.1 任务类型扩展》相关章节。

5.4.2.2 停止驱动

以下是停止驱动插件处理流程示意图：



停止插件与执行插件一样，是一个独立程序，并由服务进程 CPG 调起。该插件处理机制要比执行插件相对复杂。由图可知，它不仅要与 CPG 互动、同时还需要与相关程序以及对应的执行插件互动。

停止插件主要完成两件事：

1. 在实现任务程序停止之前，停止插件先通知对应执行插件。
2. 通过相应任务类型 API 强行终止任务。任务程序被终止后，执行插件就会自动收到任务被终止的异常信息，但由于停止插件的预先通知，执行插件就不会把该异常信息向核心返回，而是以一种调度平台定义的人为终止信息向核心返回。

说明： 关于停止插件的开发以及更多信息请参阅《TASKCTL-CIR 2.1 任务类型扩展》相关章节。

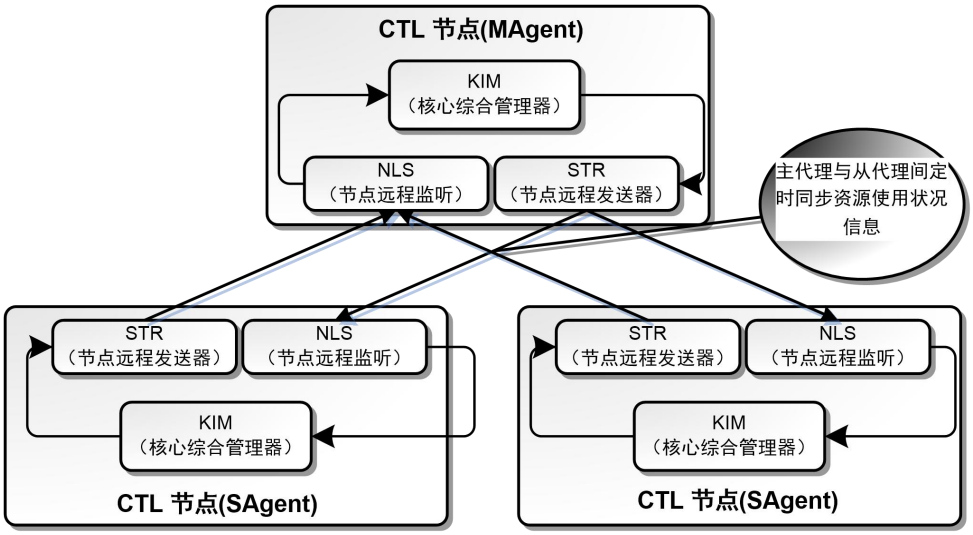
5.4.3 负载均衡处理机制

负载均衡主要通过代理级联方式完成。同时负载均衡也是相对一个执行域而言，相关信息在本文“3.3.4 负载均衡部署“已作过描述，本节着重描述负载均衡

的实现过程。

负载均衡最重要一点就是资源信息的收集，在 TASKCTL 核心体系中，资源收集以一个代理执行域（也可称为一个集群域）为单位，并通过相应主代理收集完成。

主代理收集资源如下图所示：



由上图可知，代理节点收集资源主要通过 KIM 组件完成，该组件会定期收集本节点的资源情况，并向上级主代理汇报。主代理通过收集下级从代理的资源，当有执行指令时，主代理就可根据本执行域各个代理节点的资源情况，将执行指令转到相对空闲的代理节点，从而完成负载均衡处理。