分布式计算框架(DC4C)

厉华

版本修订

版本	日期	修订人	内容
0.1.0	2015-05-28	厉华	创建文档 编写 第四章 开发接口 编写 附录 A.任务调度引擎 数据库表结构
0.2.0	2015-05-30	厉华	编写 第二章 架构与模块 编写 第三章 安装部署 编写 第六章 应用案例
0.3.0	2015-05-31	厉华	编写 附件 B.自带测试程序
0.4.0	2015-06-02	厉华	修改了表结构 调整了一些函数接口原型
0.5.0	2015-06-04	厉华	编写 2.1.4 高可靠性 2.1.5 高伸缩性
0.6.0	2015-06-07	厉华	重构了发起任务的高层和低层 API
0.7.0	2015-06-11	厉华	编写 5 内部实现
0.8.0	2015-06-24	厉华	调整了一些函数原型

目录索引

1 概述 6

1.1 简介 6

1.2 体系结构 6

1.3 功能和优势 8

1.4 与 Hadoop 比较 9

2 架构与模块 11

2.1 基础平台架构 11

2.1.1 注册节点 11

2.1.2 计算节点和 API 包 13

2.1.3 用户节点和 API 包 14

2.1.4 高可靠性 18

2.1.5 高伸缩性 20

2.2 任务调度引擎 21

2.2.1 DAG 任务调度引擎 21

3 安装部署 22

3.1 单机部署 22

3.1.1 安装 22

3.1.2 部署 23

3.1.3 测试 24

3.2 集群部署 25

3.2.1 部署 25

3.2.2 扩大集群 27

3.2.3 缩小集群 28

4 开发接口 29

4.1 用户节点接口 29

4.1.1 环境类 29

4.1.2 同步发起任务类 31

4.1.3 获取执行反馈类 33

4.1.4 低层函数类 34
4.1.5 其它类 37
4.2 计算节点接口 37
4.2.1 日志操作类 38
4.2.2 反馈信息类 38
4.3 任务调度引擎接口 39
4.3.1 高层函数 39
4.3.2 低层函数 42
4.4 代码示例 45
4.4.1 单任务分派 45
4.4.2 批量任务分派（高层函数） 47
4.4.3 批量任务分派（低层函数） 50
4.4.4 多批量任务分派（高层函数） 54
4.4.5 多批量任务分派（低层函数） 58
4.4.6 DAG 调度多批量任务分派（从配置文件载入配置）（高层函数） 63
4.4.7 DAG 调度多批量任务分派（从配置文件载入配置）（低层函数） 66
4.4.8 DAG 调度多批量任务分派（从数据载入配置）（低层函数） ... 68
5 内部实现 73
5.1 基础平台架构 73
5.1.1 接口协议 73
5.1.2 架构与模块 77
5.2 任务调度引擎 83
5.2.1 DAG 任务调度引擎 83
6 应用案例 85
6.1 计算圆周率 85
6.2 互联网数据挖掘平台 91
7 附件 94

7.1 附件 A.任务调度引擎 数据库表结构 94
7.1.1 计划表 94
7.1.2 批量表 94
7.1.3 批量依赖关系表 95
7.1.4 批量任务表 95
7.2 附件 B.自带测试程序用法 96
7.2.1 单任务分派 96
7.2.2 批量任务分派 96
7.2.3 多批量任务分派 97

1 概述

1.1 简介

DC4C 是一个通用的分布式计算框架，研发初衷来自于 2015 年初我开发互联网数据挖掘平台的任务调度的技术需求。经过 2015 年 4 月一个月的研发，发布第一版原型，而后不断优化完善，扩展功能，目前最新版本为 v1.1.4。

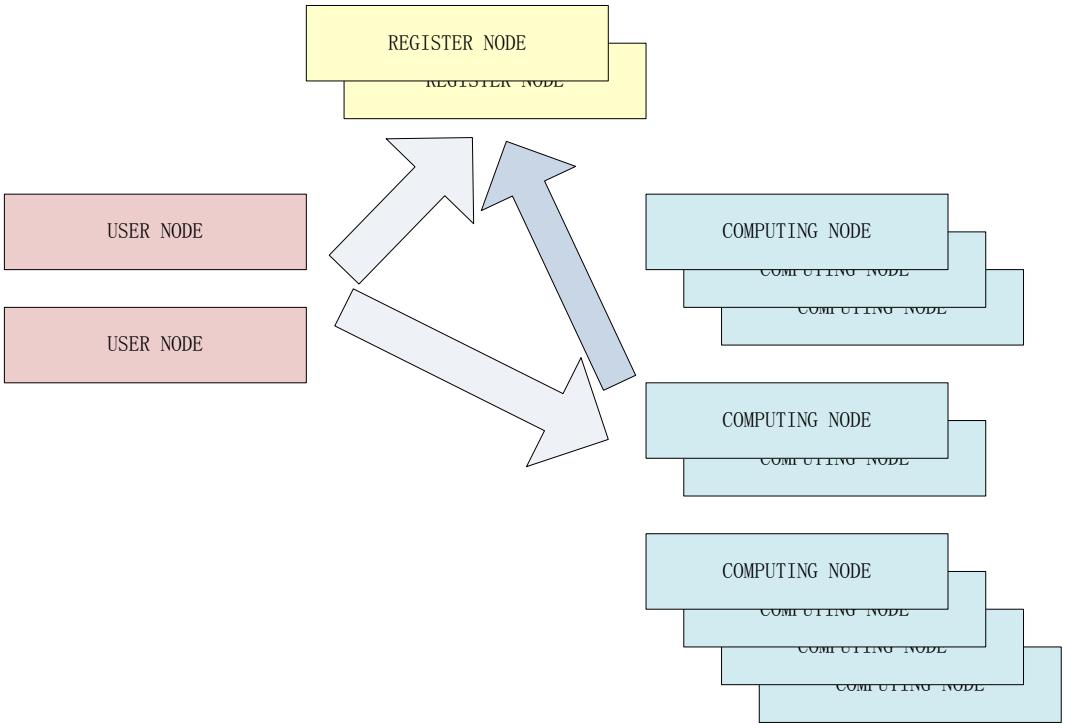
DC4C 借鉴了 Hadoop、Fourinone 等分布式产品的设计，加入自有特色，充分考虑高可靠性、高伸缩性，也是业界首先实现有向无环图任务调度引擎的分布式计算框架之一，特别适合批量任务流处理的分布式架构。

DC4C 核心完全用 C 编写，手工代码约 1 万行。此外大量使用代码自动化生成技术（如 json 报文的打包解包），大幅减小了开发量、提高了开发效率、减轻了底层细节编码压力。用户 API 包和计算节点也可以用其它语言实现以支持不同语言开发的应用。

1.2 体系结构

DC4C 体系结构包含**基础平台架构**、**用户 API 包**和**基于有向无环图（下面简称 DAG）的任务调度引擎**。

基础平台架构包含三类节点：**注册节点**、**计算节点**和**用户节点**。



注册节点（守护进程）：负责接受计算节点注册、状态变更、注销；接受用户节点查询空闲计算节点；接受 telnet 连接在线查询和管理。进程框架为父子进程监控进程异常，可以同时起多对保持计算节点注册信息冗余，提高可靠性。

计算节点（守护进程）：负责向注册节点注册；接受用户节点分派任务并反馈执行结果；随时向注册节点报告状态。进程框架为父进程+子进程组（计算节点组）监控进程异常。

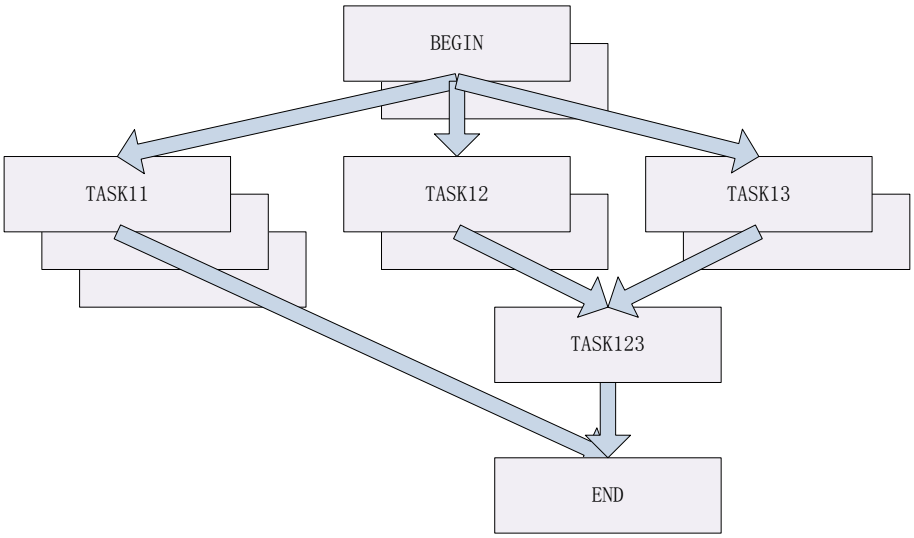
用户节点（可以是守护进程、命令行进程或其它任何类型的用户自己控制的进程）：用户程序调用用户 API，向注册节点查询当前空闲的计算节点，分派任务给计算节点并监督执行。

计算节点启动后向**注册节点**注册后保持长连接，并互相心跳。

用户节点用户应用通过分派任务 API 向**注册节点**查询当前空闲的**计算节点**集合，然后发送执行命令行和执行程序 MD5 给**计算节点**，**计算节点**校验执行程序 MD5，如果不匹配，联动请求**用户节点**分发新版程序，然后再执行命令。

用户 API 包供用户节点和计算节点应用调用，实现**用户节点**的任务分派、计算节点应用控制等功能。

有向无环图(DAG)任务调度引擎封装了基础平台架构提供的基本任务分派功能，实现了有向无环图数据结构的任务流的执行控制，便于用户直接搭建复杂任务依赖关系调度平台。



1.3 功能和优势

* 对于应用开发人员，无需编写任何并发控制细节代码（如 fork、wait）就可轻松实现本地或集群的并发管理，以及本地风格（wait 子进程退出值 status）的执行反馈。计算节点用户应用本身就是可执行程序，便于本地调试。

* 对于系统运维人员，随时根据当前系统负载随时伸缩（扩大或减小）集群规模，而不影响系统的功能性，更无需应用开发人员参与。集群伸缩无需重启等影响当前正在处理的动作，没有配置文件，大大减少运维复杂度，实现高伸缩性。

* 通过网络连接心跳、父子进程监控、数据冗余等实现高可用性。

* 用户 API 包提供了单任务分派、批量任务分派、多批量任务分派等高层封装 API，也提供了低层 API 供用户自己封装适应自己应用场景的任务分派器。用户 API 包也提供了同步、多路复用等分派模式，支持各种用户程序结构。

* 实现了一个 DAG 任务调度引擎，是业界最早实现该数据结构任务调度引擎的分布式计算框架之一。

* 用户应用程序自动检查应用版本和自动分发部署。

* 支持 telnet 直接查询和管理集群状态。

* 所有类型节点可自由部署在任意台机器内；支持最多 8 个注册节点和（理论上）10 万个计算节点。

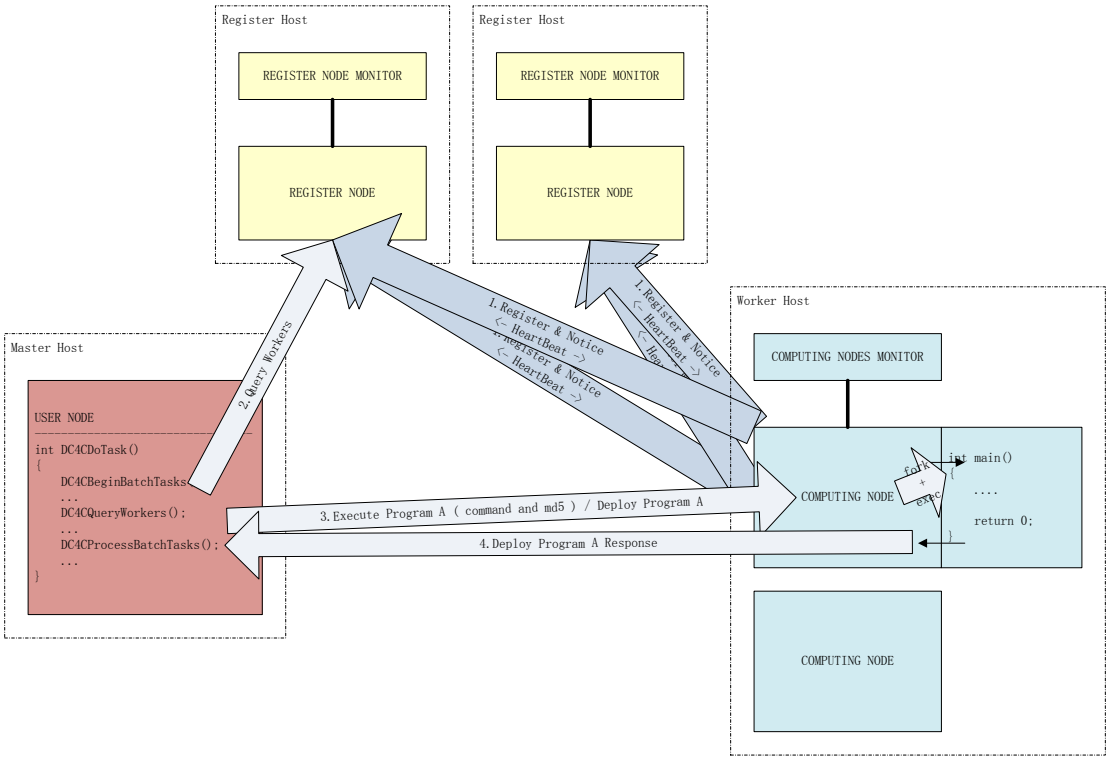
1.4 与 Hadoop 比较

	Hadoop	DC4C
开发语言	Java	C
体积	71MB	676KB
软件包依赖	约 12 个 jar 包	仅依赖开源库 fasterjson
概念	JobTraker NameNode TaskTraker/DataNode	UserNode（用户节点） RegisterNode（注册节点） ComputingNode（计算节点）
分布式部件	分布式计算、分布式存储、分布式协调	目前仅实现了分布式计算
中心节点	单点	最多 8 个多活、信息冗余
基础平台配置	较多配置文件和复杂配置	无配置文件
集群搭建	需要创建专用用户组、用户；需要配置 ssh 等复杂环境；需要安装 JVM 和设置 JVM 环境	任意系统用户中启动节点守护进程即可
分布式模式	基于分布式文件系统的分布式计算	通用的分布式计算
并发模式	M*(1...N)，M 台机器，每台机器部署 1 个 JVM 实例，较	M*N，M 台机器，每台机器最多可部署 N 个计算节点组，完全的自由

	少部署多个。主要多线程并发	部署。支持多进程或多线程（用户控制）并发
内存资源耗用	单 JVM 实例模式/进程最多只能使用 2GB 内存；Java 耗内存	计算节点组/进程组可完全利用所有内存；C 自主可控内存
计算资源耗用	多线程充分利用多核 CPU	多进程或多线程充分利用多核 CPU；还支持计算节点绑定独占 CPU 核，提高计算性能
进程监控	无	注册节点父子进程监控重启、计算节点父子进程组监控重启
自动部署	通过分布式缓存统一分发	任务分发时智能自动部署
任务调度模型	基于 MapReduce 的手工代码控制	基于有向无环图(DAG)的任务调度引擎+配置（配置文件或数据库）
操作系统支持	JVM 支持的所有操作系统；官方不建议把 WINDOWS 用作生产环境	目前注册节点和计算节点只能运行在 Linux 上，用户节点可运行在 Linux/UNIX、WINDOWS

2 架构与模块

2.1 基础平台架构



2.1.1 注册节点

注册节点由分布在一台或若干台主机内的父子进程对（守护进程）组成。

注册节点父进程启动后转换为守护进程，然后创建子进程对外提供服务，父进程则监控子进程异常（如崩溃后自动重启）。

注册节点子进程通过对外 TCP 端口接受计算节点和用户节点的连接，接收请求并返回响应。接收用户节点空闲计算节点查询并响应。接收计算节点心跳并响应。

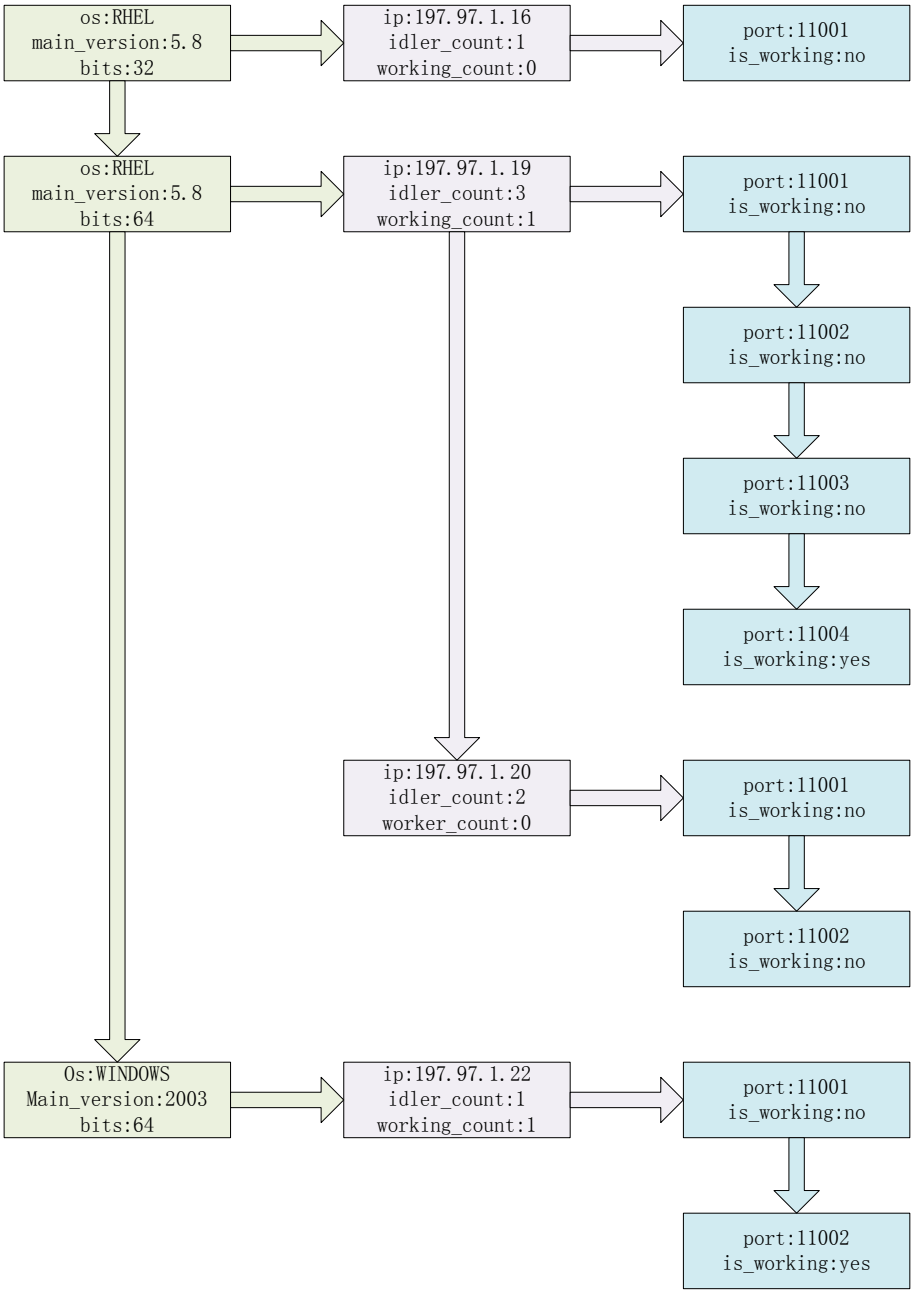
该端口还用于接受直接连接（如 telnet），接收命令行并返回命令执行响应，命令主要包含查询已注册计算节点信息。

注册节点与计算节点之间保持长连接，并定时互相心跳以监测长连接健康

性。

注册节点内部通过多路复用实现同时为所有连接方服务。

2.1.1.1注册节点维护计算节点信息策略



用户节点向注册节点查询空闲计算节点时，分配策略目标：

保证分配的计算节点尽量均分到所有主机上。

计算节点新注册策略：

* 新注册 PORT 块上浮到顶。

* IP 块上浮到合适的位置，IP 块按空闲计算节点数量、工作计算节点数量从上（最大）到下（最小）冒泡。

用户节点请求分配 worker 策略:

* 操作系统、主版本号、位数必须匹配。

* 优先分配空闲计算节点数量多的 IP 块。

* 每个 IP,PORT 分配完后，IP 块下降到合适的位置，按空闲计算节点数量、工作计算节点数量从上（最大）到下（最小）冒泡。

计算节点状态通知调整策略：

* 设置完状态后，IP 块上浮到合适的位置，按空闲计算节点数量、工作计算节点数量从上（最大）到下（最小）冒泡。

计算节点注销调整策略：

* 直接删除

* 如果下层链表为空，则递归向上删除

2.1.2 计算节点和 API 包

计算节点由分布在若干台主机内的父进程-子进程组（守护进程）组成。

计算节点父进程启动后转换为守护进程，创建子进程组对外提供服务，父进程则监控子进程组异常（如崩溃后自动重启）。

计算节点子进程组中的进程通过父进程传递的 TCP 基端口和进程序号（从 0 开始）偏移计算出独有端口号，向注册节点注册自己，唯一标识为"IP:PORT"，然后互作心跳以监测长连接健康性。

计算节点端口还接受用户节点连接，接收执行程序请求，对比程序 MD5 是

否与本地程序 MD5 一致，如不一致则请求用户节点发送新版程序。计算节点子进程创建孙子进程并覆盖映像为执行程序，然后子进程监控程序执行，直到执行结束或中断，最后把错误码（子进程处理过程中的错误信息）和孙子进程返回码响应回用户节点。

计算节点用户 API 包提供了计算节点用户程序的附加功能，如计算节点用户进程需要返回用户节点用户进程一块任意格式的文本信息。

计算节点内部通过多路复用实现同时为所有连接方服务。

2.1.3 用户节点和 API 包

用户节点由分布在任意主机内的用户编写的程序进程组成。

用户节点进程框架由用户自己控制，可以是守护进程，也可以是命令行进程或其它任何类型的用户自己控制的进程，DC4C 不假设其生命周期和进程模型。

用户程序调用用户节点 API 包实现用户节点角色，主要提供任务分派功能。

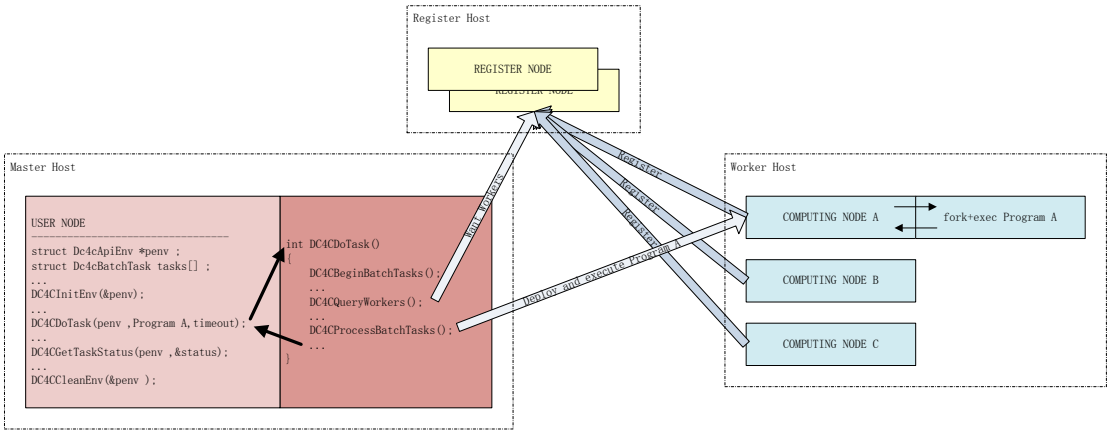
用户程序在使用用户节点 API 包的其它函数前应调用初始化任务环境函数 DC4CInitEnv 构建句柄，在结束任务处理后应调用清理任务环境函数 DC4CCleanEnv 清理和销毁句柄。

用户节点 API 包主要提供了单任务分派、批量任务分派、多批量任务分派等任务分派功能，获取任务执行结束反馈信息，也提供了同步分派、异步分派等任务控制功能。

用户节点 API 包还提供了高层函数和低层函数，高层函数封装了低层函数集合，方便用户简单、直接使用，低层函数提供给用户自己封装。

用户节点对外连接用完后立即断开，即短连接。

2.1.3.1单任务分派



用户节点里的用户程序在初始化任务环境后，调用 DC4CDoTask 向注册节点询问当前空闲计算节点信息、分派任务给该计算节点、等待计算节点执行完成、或超时，如果要查询执行反馈可调用 DC4CGetTask*

如果计算节点发现任务包里的执行程序 MD5 与本地的不一致，请求用户节点的用户程序发送新版执行程序过来，然后再执行。

用户程序里涉及调用 API 及流程：

DC4CInitEnv - 初始化任务环境

DC4CDoTask - 分派单任务，并等待结束

DC4CCleanEnv - 清理任务环境

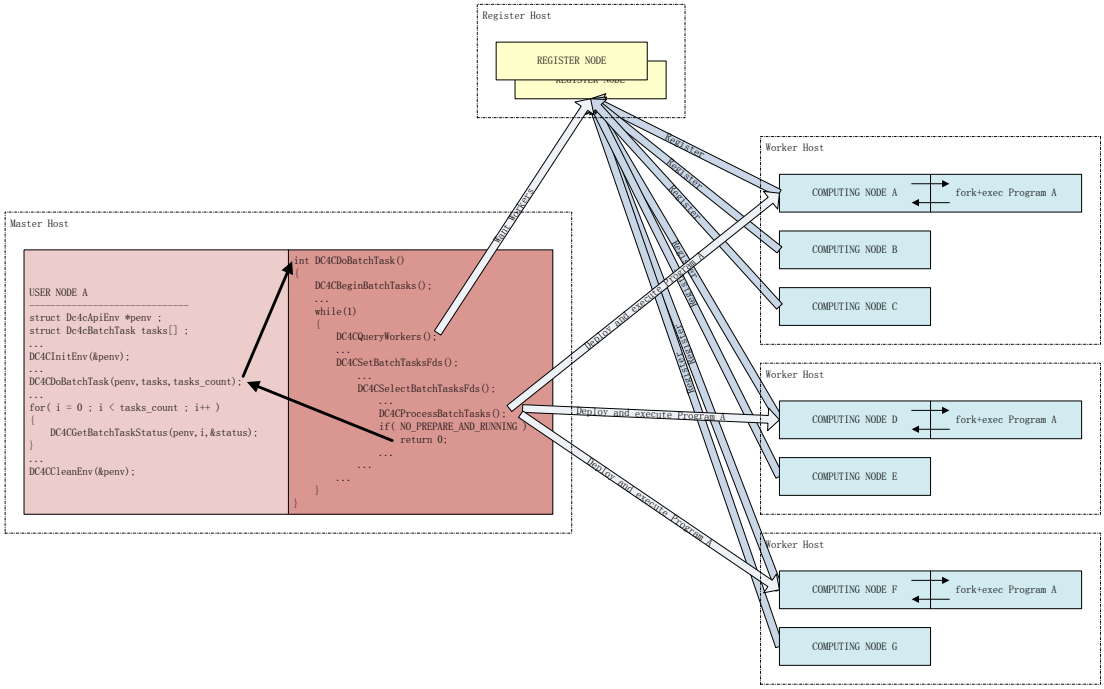
可能调用的其它 API：

DC4CSetTimeout - 设置全局超时时间

DC4CSetOptions - 设置公共选项

DC4CGetTask* - 得到执行反馈

2.1.3.2批量任务分派



用户节点里的用户程序在初始化任务环境后，构造 struct Dc4cBatchTask tasks_array[]数组，填充任务集合，接着调用 DC4CDoBatchTask 向注册节点询问当前空闲计算节点信息集合、分派任务集合给计算节点集合（当空闲节点少于任务集合时分批次分派任务）、等待计算节点执行完成、或超时，如果要查询执行反馈可调用 DC4CGetBatchTasks*。

用户程序里涉及调用 API 及流程：

DC4CInitEnv - 初始化任务环境

DC4CDoBatchTasks - 分派批量任务，并等待结束

DC4CCleanEnv - 清理任务环境

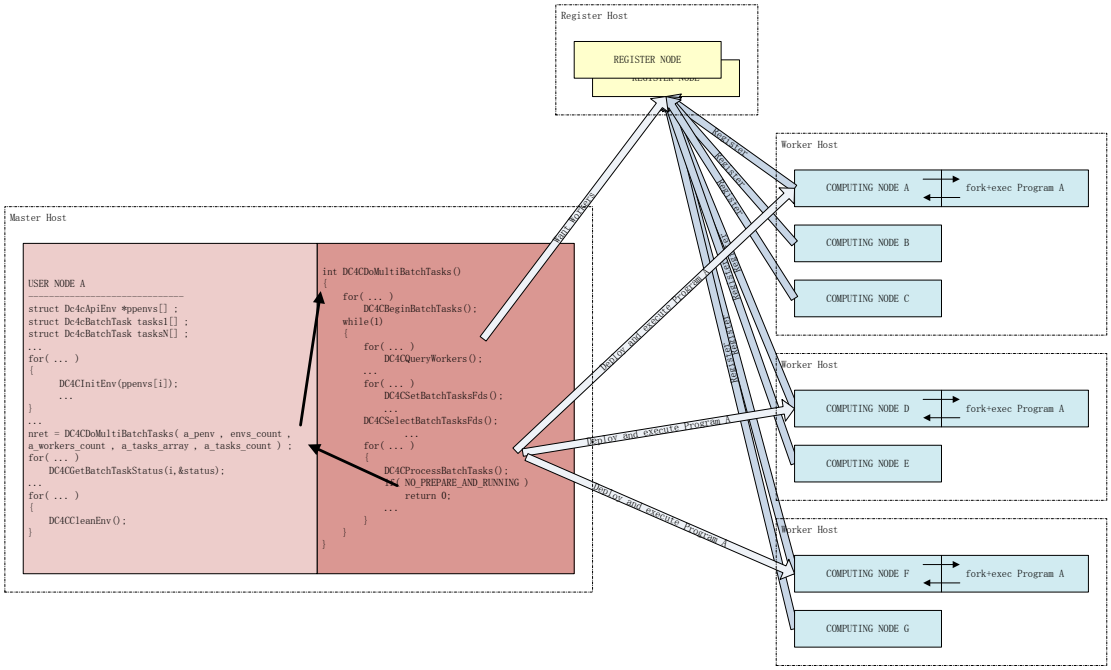
可能调用的其它 API：

DC4CSetTimeout - 设置全局超时时间

DC4CSetOptions - 设置公共选项

DC4CGetBatchTasks* - 得到执行反馈

2.1.3.3多批量任务分派



用户节点里的用户程序在初始化任务环境集合后，构造每个任务环境中的struct Dc4cBatchTask tasks_array[]数组，填充任务集合，并开始分派批量任务，接着调用 DC4CDoMultiBatchTasks 分派各个任务环境中的任务集合给计算节点集合（当空闲节点少于任务集合时分批次分派任务）、等待计算节点执行完成、或超时，当有批量任务完成时会抛出对应任务环境（如果要查询执行反馈可调用DC4CGetBatchTasks*），直到所有批量任务都结束。

用户程序里涉及调用 API 及流程：

DC4CInitEnv - 初始化任务环境

DC4CDoMultiBatchTasks- 开始多批量任务

DC4CCleanEnv - 清理任务环境

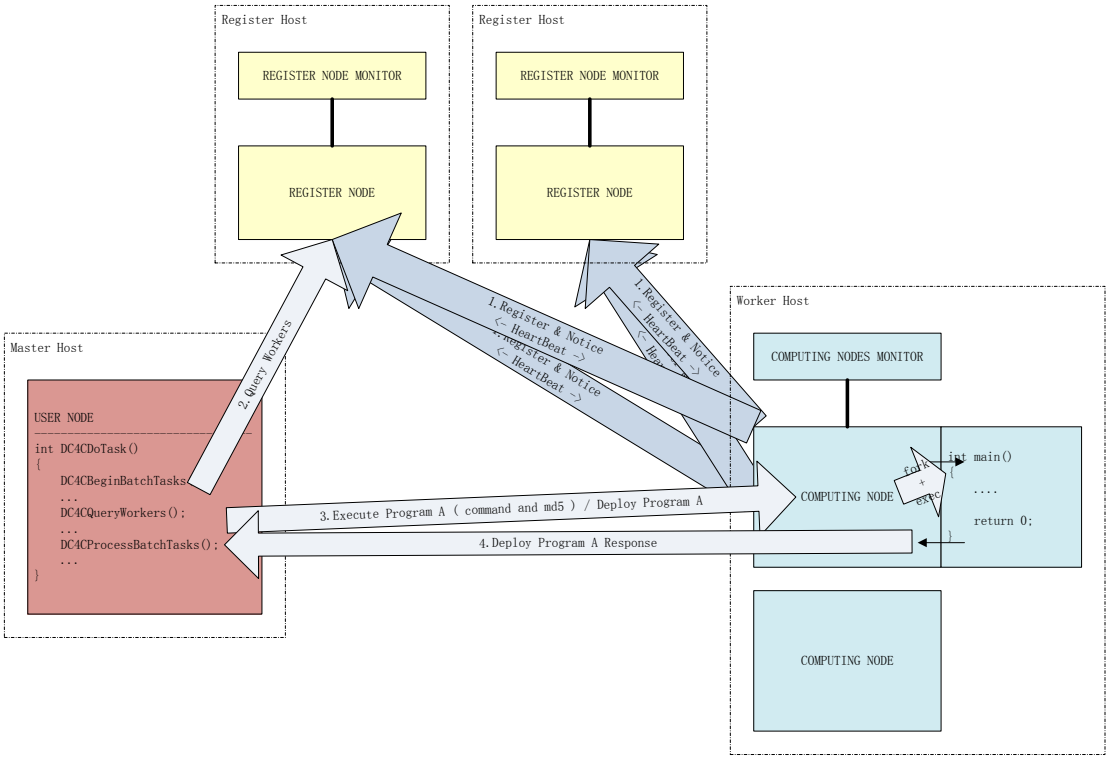
可能调用的其它 API：

DC4CSetTimeout - 设置全局超时时间

DC4CSetOptions - 设置公共选项

DC4CGetBatchTasks* - 得到执行反馈

2.1.4 高可靠性



当注册节点崩溃时，计算节点立即侦测到与之连接异常，然后尝试重新连接，用户节点立即侦测到与之连接异常，转而连接其它注册节点，因其它注册节点拥有相同的冗余计算节点注册信息，不影响用户节点工作。注册节点父进程立即侦测到子进程崩溃，重启该进程。

当注册节点僵死时，计算节点通过心跳发现异常，强制断开并尝试重新连接，用户节点与之通讯超时，转而连接其它注册节点。注册节点父进程不会侦测到子进程异常，但该节点被孤立，所有错误日志指向该节点，便于手工介入处理。

当计算节点崩溃时，注册节点立即侦测到与之连接异常，断开等待其重启后重连，用户节点立即侦测与之连接异常，如果设置了忽略错误选项，重新向注册节点查询其它空闲计算节点，分派任务，否则报错，人工介入处理。计算节点父进程立即侦测到子进程崩溃，重启该进程。

当计算节点僵死时，注册节点通过心跳发现异常，强制断开并等待重连，用户节点与之通讯超时，如果设置了忽略错误选项，重新向注册节点查询其它空闲计算节点，分派任务，否则报错，人工介入。计算节点父进程不会侦测到子进程

异常，但该节点被孤立，所有错误日志指向该节点，便于手工介入处理。

当计算节点用户程序进程崩溃时，计算节点立即侦测到用户程序进程非正常结束，提前发送执行完成响应给用户节点，然后等待其他用户节点任务。

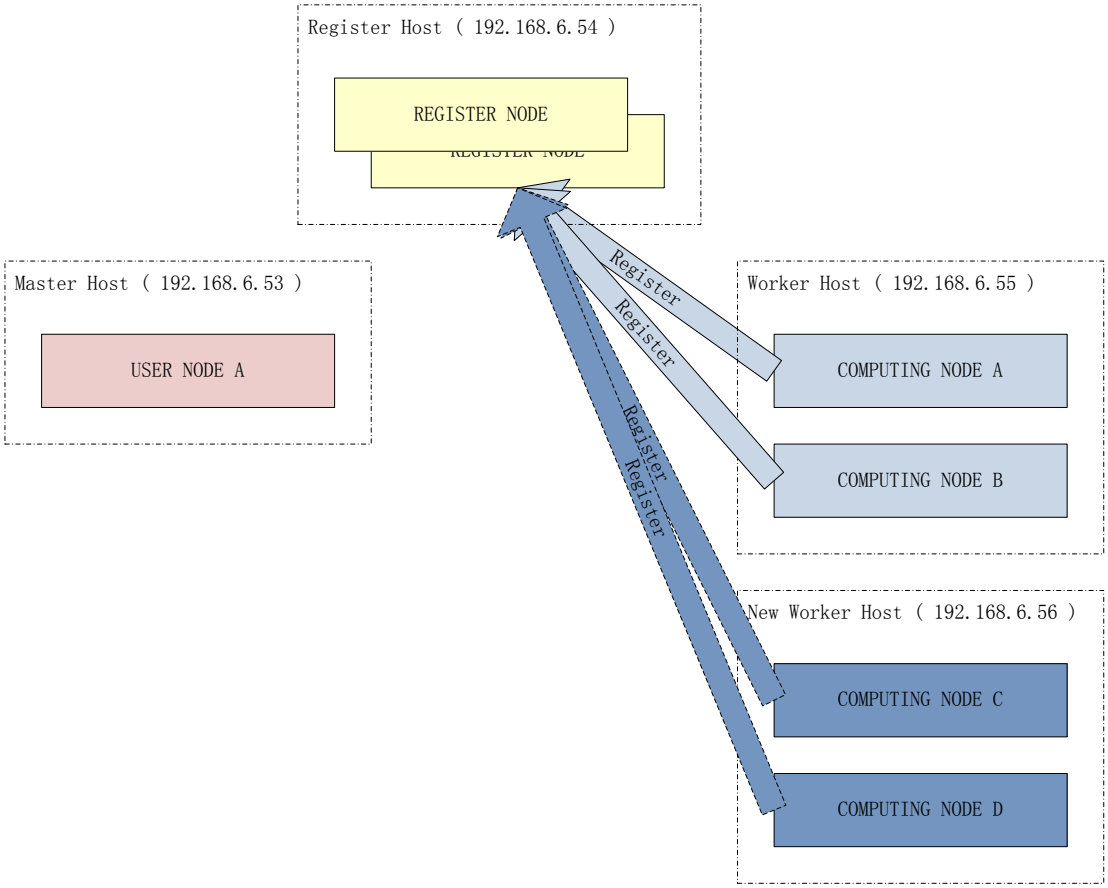
当计算节点用户程序进程僵死时，直到用户节点等待执行完成响应超时，强制断开连接，计算节点立即侦测到当前执行任务的用户节点断开连接，也强制结束计算节点用户程序进程，然后等待其他用户节点任务。

当用户节点用户程序进程崩溃时，计算节点立即侦测到当前执行任务的用户节点断开连接，也强制结束计算节点用户程序进程。

当用户节点用户程序进程僵死时，计算节点等待本节点用户程序进程结束，发送执行完成响应给用户节点，然后等待其他用户节点任务。

当用户节点等待计算节点用户程序进程超时时，强制断开连接，报错返回，计算节点立即侦测到当前执行任务的用户节点断开连接，也强制结束计算节点用户程序进程，然后等待其他用户节点任务

2.1.5 高伸缩性



当三类节点组成分布式集群后，需要新增计算节点，只需在新加入主机或已有主机的某用户内，启动新计算节点组，查询日志确认加入成功后。把该命令并入本机统一管理脚本中便于以后直接使用。

```
$ dc4c_wserver -r 192.168.6.54:12001,192.168.6.54:12002 -w 192.168.6.56:13001 -c 2
```

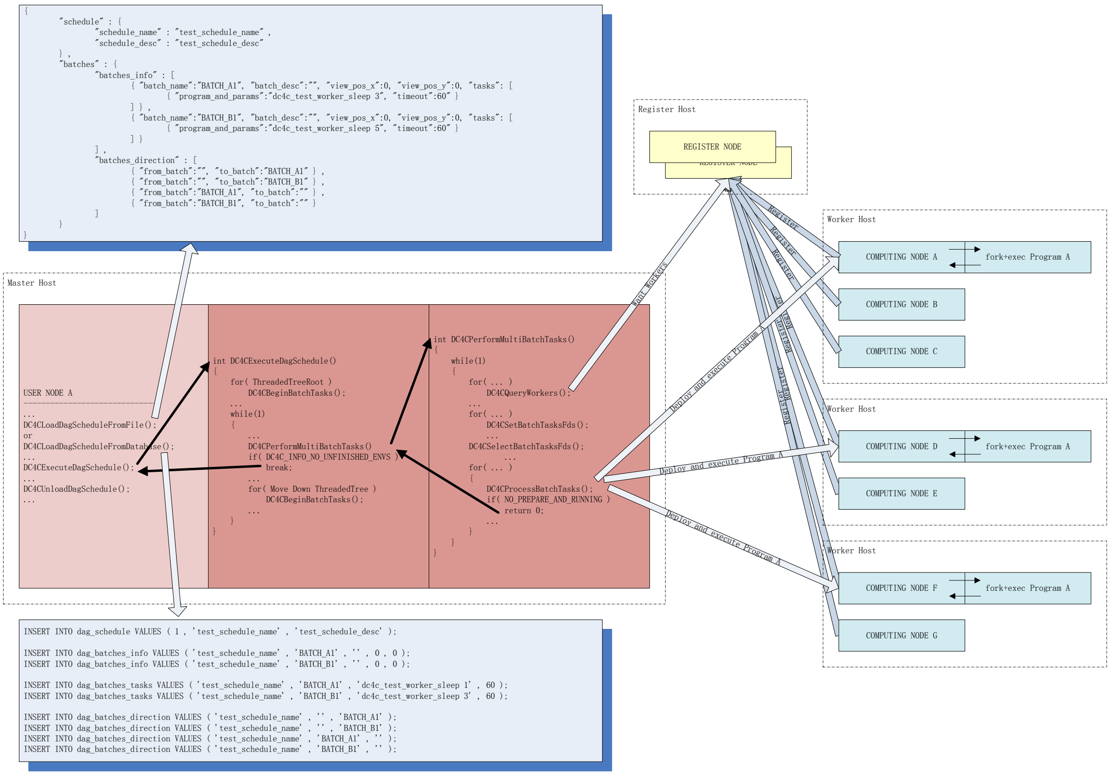
整个加入过程不影响正在处理的业务。

当需要删除计算节点时，只需向目标计算节点组的父进程发送中止信号(TERM)，父进程转发信号给子进程组，没有任务的计算节点立即结束，有任务执行的计算节点关闭其它服务，等待任务完成后再结束。在本机统一管理脚本中删除该计算节点相关命令。

无需编写任何配置文件！无需重启！

2.2 任务调度引擎

2.2.1 DAG 任务调度引擎



任务调度引擎是用户节点用户程序的任务流控制的函数接口，替代手工编写代码控制，实现根据任务依赖关系，自动、快速调度任务的功能。

任务调度引擎封装了原生的同步发起多批量任务函数接口，提供了一批 API 函数，用于读入外部配置文件或数据库中的配置，按序执行线性、树形等多批量任务。

目前实现有向无环图(DAG)数据结构的任务流模型，几乎可以配置出任意流程的任务树。

3 安装部署

3.1 单机部署

3.1.1 安装

新建系统用户 dc4c 或在已有用户里，展开二进制安装包

```
[dc4c@rhel54 /home/dc4c] tar xvzf dc4c-Linux-bin-v1.1.4.tar.gz
bin/dc4c_rserver
bin/dc4c_wserver
bin/dc4c_test_master
bin/dc4c_test_batch_master
bin/dc4c_test_multi_batch_master
bin/dc4c_test_tfc_dag_master
bin/dc4c_test_worker_pi
bin/dc4c_test_worker_sleep
lib/libdc4c_util.so
lib/libdc4c_proto.so
lib/libdc4c_api.so
lib/libdc4c_tfc_dag.so
lib/libfasterjson.so
shbin/dc4c.do
```

建立日志目录

```
[dc4c@rhel54 /home/dc4c] mkdir log
```

如果作为用户节点，推荐设置环境变量 DC4C_RSERVERS_IP_PORT，调用函数 DC4CInitEnv 时 rservers_ip_port（置为 NULL）取环境变量。

```
[dc4c@rhel54 /home/dc4c] vi .profile
...
################################################
# for dc4c
export DC4C_RSERVER_IP_PORT=0:12001
...
```

DC4C 安装完成

3.1.2 部署

我们这样规划单机架构：

注册节点	1*1 对	127.0.0.1:12001
计算节点	1*5 个	127.0.0.1:13001~13005
用户节点	-	127.0.0.1

按照此规划，分别启动注册节点和计算节点

```
[dc4c@rhel54 /home/dc4c] dc4c_rserver -r 127.0.0.1:12001
[dc4c@rhel54 /home/dc4c] dc4c_wserver -r 127.0.0.1:12001 -w 127.0.0.1:13001 -c 5
```

查看进程

```
[dc4c@rhel54 /home/dc4c] ps -ef | grep dc4c
dc4c      31213 31212  0 07:08 pts/2    00:00:00 -bash
dc4c      31331 31330  0 07:18 pts/3    00:00:00 -bash
dc4c      31368     1  0 07:19 ?        00:00:00 dc4c_rserver -r 127.0.0.1:12001
dc4c      31369 31368  0 07:19 ?        00:00:00 dc4c_rserver -r 127.0.0.1:12001
dc4c      31373     1  0 07:19 ?        00:00:00 dc4c_wserver -r 127.0.0.1:12001 -w 127.0.0.1:13001 -c 5
dc4c      31374 31373  0 07:19 ?        00:00:00 dc4c_wserver -r 127.0.0.1:12001 -w 127.0.0.1:13001 -c 5
dc4c      31375 31373  0 07:19 ?        00:00:00 dc4c_wserver -r 127.0.0.1:12001 -w 127.0.0.1:13001 -c 5
dc4c      31376 31373  0 07:19 ?        00:00:00 dc4c_wserver -r 127.0.0.1:12001 -w 127.0.0.1:13001 -c 5
dc4c      31377 31373  0 07:19 ?        00:00:00 dc4c_wserver -r 127.0.0.1:12001 -w 127.0.0.1:13001 -c 5
dc4c      31378 31373  0 07:19 ?        00:00:00 dc4c_wserver -r 127.0.0.1:12001 -w 127.0.0.1:13001 -c 5
dc4c      31379 31213  0 07:20 pts/2    00:00:00 ps -ef
dc4c      31380 31213  0 07:20 pts/2    00:00:00 grep dc4c
```

检查是否有错误等级的日志

```
[dc4c@rhel54 /home/dc4c] cd log
[dc4c@rhel54 /home/dc4c/log] ls -l
total 32
-rwxrwxrwx 1 dc4c dc4c 2881 Apr 20 07:20 dc4c_rserver_1_127.0.0.1:12001.log
-rwxrwxr-x 1 dc4c dc4c  174 Apr 20 07:19 dc4c_rserver_m_127.0.0.1:12001.log
-rwxrwxrwx 1 dc4c dc4c 1032 Apr 20 07:20 dc4c_wserver_1_127.0.0.1:13001.log
-rwxrwxrwx 1 dc4c dc4c 1032 Apr 20 07:20 dc4c_wserver_2_127.0.0.1:13002.log
-rwxrwxrwx 1 dc4c dc4c 1032 Apr 20 07:20 dc4c_wserver_3_127.0.0.1:13003.log
-rwxrwxrwx 1 dc4c dc4c 1032 Apr 20 07:20 dc4c_wserver_4_127.0.0.1:13004.log
-rwxrwxrwx 1 dc4c dc4c 1032 Apr 20 07:20 dc4c_wserver_5_127.0.0.1:13005.log
-rwxrwxr-x 1 dc4c dc4c  601 Apr 20 07:19 dc4c_wserver_m_127.0.0.1:13001.log
[dc4c@rhel54 /home/dc4c/log] grep ERROR *.log
[dc4c@rhel54 /home/dc4c/log]
```

连接注册节点端口，直接用命令查询已注册的计算节点信息

```
[dc4c@rhel54 /home/dc4c/log] telnet 127.0.0.1 12001
```

```
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
list workers
Linux    2.6.18-164.el5   32       127.0.0.1        13005    0
Linux    2.6.18-164.el5   32       127.0.0.1        13003    0
Linux    2.6.18-164.el5   32       127.0.0.1        13004    0
Linux    2.6.18-164.el5   32       127.0.0.1        13002    0
Linux    2.6.18-164.el5   32       127.0.0.1        13001    0
list hosts
Linux    2.6.18-164.el5   32       127.0.0.1        5        0
list os
Linux    2.6.18-164.el5   32
quit
Connection closed by foreign host.
```

这里的命令可以采用缩写，如"list"可以缩写成"lis"或"li"甚至"l"，只要左面一段匹配上就可以了，上面输入的 4 条命令可以缩写为：

```
l w
l h
l o
q
```

"list worker"输出最后一列为该计算节点是否空闲，"0"表示空闲状态，"1"表示工作状态。

"list hosts"输出最后两列为该主机空闲计算节点数量和工作计算节点数量。

这里显示的可以看出，集群启动成功，可以在用户节点测试了。

3.1.3 测试

用安装包自带的"hello world"测试

```
[dc4c@rhel54 /home/dc4c/log] ./dc4c_test_batch_master 127.0.0.1:12001 -2 -2 "dc4c_test_worker_hello world"
DC4CInitEnv ok
DC4CDoTask ok
Task[0]-[127.0.0.1][13004]-[1429489271000028148][dc4c_test_worker_hello world][60][0]-[0][0][hello world]
Task[1]-[127.0.0.1][13005]-[1429489271000028295][dc4c_test_worker_hello world][60][0]-[0][0][hello world]
Task[2]-[127.0.0.1][13003]-[1429489271000028713][dc4c_test_worker_hello world][60][0]-[0][0][hello world]
```

```
Task[3]-[127.0.0.1][13002]-[14294892710000028798][dc4c_test_worker_hello world][60][0]-[0][0][hello world]
Task[4]-[127.0.0.1][13001]-[14294892710000029201][dc4c_test_worker_hello world][60][0]-[0][0][hello world]
DC4CCleanEnv ok
```

向所有计算节点发送执行程序"dc4c_test_worker_hello world"请求，都反馈"hello world"回用户节点。

测试成功！

3.2 集群部署

3.2.1 部署

我们这样规划集群架构：

注册节点	2*1 对	192.168.6.91:12001,192.168.6.92:12001
计算节点	5*10 个	192.168.6.101:13001~13009 192.168.6.102:13001~13009 192.168.6.103:13001~13009 192.168.6.104:13001~13009 192.168.6.105:13001~13009
用户节点	-	192.168.6.81

在所有注册节点和计算节点主机里展开 DC4C 安装包，然后依次启动注册节点和计算节点

```
[dc4c@rhel91 /home/dc4c] dc4c_rserver -r 192.168.6.9112001
```

```
[dc4c@rhel92 /home/dc4c] dc4c_rserver -r 192.168.6.92:12001
```

```
[dc4c@rhel101 /home/dc4c] dc4c_wserver -r 192.168.6.90:12001,192.168.6.91:12001 -w 192.168.6.101:13001 -c 10
```

```
[dc4c@rhel102 /home/dc4c] dc4c_wserver -r 192.168.6.90:12001,192.168.6.91:12001 -w 192.168.6.102:13001 -c 10
```

```
[dc4c@rhel103 /home/dc4c] dc4c_wserver -r 192.168.6.90:12001,192.168.6.91:12001 -w
192.168.6.103:13001 -c 10
```

```
[dc4c@rhel104 /home/dc4c] dc4c_wserver -r 192.168.6.90:12001,192.168.6.91:12001 -w
192.168.6.104:13001 -c 10
```

```
[dc4c@rhel105 /home/dc4c] dc4c_wserver -r 192.168.6.90:12001,192.168.6.91:12001 -w
192.168.6.105:13001 -c 10
```

连接注册节点端口，直接用命令查询已注册的计算节点信息

```
[dc4c@rhel54 /home/dc4c/log] telnet 192.168.6.91 12001
Trying 192.168.6.91...
Connected to localhost.localdomain (192.168.6.91).
Escape character is '^]'.
l w
Linux   2.6.18-164.el5  32      192.168.6.101       13005   0
Linux   2.6.18-164.el5  32      192.168.6.101       13003   0
Linux   2.6.18-164.el5  32      192.168.6.101       13004   0
Linux   2.6.18-164.el5  32      192.168.6.101       13002   0
Linux   2.6.18-164.el5  32      192.168.6.101       13001   0
Linux   2.6.18-164.el5  32      192.168.6.102       13005   0
Linux   2.6.18-164.el5  32      192.168.6.102       13003   0
Linux   2.6.18-164.el5  32      192.168.6.102       13004   0
Linux   2.6.18-164.el5  32      192.168.6.102       13002   0
Linux   2.6.18-164.el5  32      192.168.6.102       13001   0
Linux   2.6.18-164.el5  32      192.168.6.103       13005   0
Linux   2.6.18-164.el5  32      192.168.6.103       13003   0
Linux   2.6.18-164.el5  32      192.168.6.103       13004   0
Linux   2.6.18-164.el5  32      192.168.6.103       13002   0
Linux   2.6.18-164.el5  32      192.168.6.103       13001   0
Linux   2.6.18-164.el5  32      192.168.6.104       13005   0
Linux   2.6.18-164.el5  32      192.168.6.104       13003   0
Linux   2.6.18-164.el5  32      192.168.6.104       13004   0
Linux   2.6.18-164.el5  32      192.168.6.104       13002   0
Linux   2.6.18-164.el5  32      192.168.6.104       13001   0
Linux   2.6.18-164.el5  32      192.168.6.105       13005   0
Linux   2.6.18-164.el5  32      192.168.6.105       13003   0
Linux   2.6.18-164.el5  32      192.168.6.105       13004   0
Linux   2.6.18-164.el5  32      192.168.6.105       13002   0
Linux   2.6.18-164.el5  32      192.168.6.105       13001   0
l h
Linux   2.6.18-164.el5  32      192.168.6.101       5       0
```

```
Linux    2.6.18-164.el5   32       192.168.6.102       5       0
Linux    2.6.18-164.el5   32       192.168.6.103       5       0
Linux    2.6.18-164.el5   32       192.168.6.104       5       0
Linux    2.6.18-164.el5   32       192.168.6.105       5       0
l o
Linux    2.6.18-164.el5   32
q
Connection closed by foreign host.
```

集群可以工作了！

3.2.2 扩大集群

当集群负载过大时，可考虑新增计算节点

新增计算节点	1*10 个	192.168.6.106:13001~13009

在新增计算节点主机里展开 DC4C 安装包，然后启动计算节点

```
 [dc4c@rhel106 /home/dc4c] dc4c_wserver -r 192.168.6.90:12001,192.168.6.91:12001 -w
192.168.6.106:13001 -c 10
```

连接注册节点端口，直接用命令查询到新增已注册的计算节点信息

```
[dc4c@rhel54 /home/dc4c/log] telnet 192.168.6.91 12001
Trying 192.168.6.91...
Connected to localhost.localdomain (192.168.6.91).
Escape character is '^]'.
l w
...
Linux    2.6.18-164.el5   32       192.168.6.106       13005   0
Linux    2.6.18-164.el5   32       192.168.6.106       13003   0
Linux    2.6.18-164.el5   32       192.168.6.106       13004   0
Linux    2.6.18-164.el5   32       192.168.6.106       13002   0
Linux    2.6.18-164.el5   32       192.168.6.106       13001   0
...
l h
...
Linux    2.6.18-164.el5   32       192.168.6.106       5       0
...
l o
Linux    2.6.18-164.el5   32
q
Connection closed by foreign host.
```

注册节点接受了新增计算节点组注册，下次用户节点再来查询空闲计算节点

信息时就会被优先查询到。

新增计算节点组可以工作了！

3.2.3 缩小集群

当集群负载过小时，可考虑减少计算节点

直接在要移除的计算节点主机里杀死计算节点进程组即可。

目标计算节点组脱离集群了！

4 开发接口

4.1 用户节点接口

4.1.1 环境类

4.1.1.1DC4CInitEnv

函数原型： int DC4CInitEnv(struct Dc4cApiEnv **ppenv , char *rservers_ip_port);

函数描述： 初始化批量任务环境

函数说明： 申请 Dc4cApiEnv 任务环境结构所需内存，并初始化，让(*ppenv)指向该结构。

解析注册节点地址信息，保存到环境结构中。注册节点地址信息可以是”ip:port”格式，也可以带多个地址”ip1:port1,ip2:port2,…”。如果该参数为 NULL，尝试取系统环境变量”DC4C_RSERVERS_IP_PORT”。

函数返回值为 0 表示成功，非 0 表示失败。

代码示例：

```
struct Dc4cApiEnv *penv = NULL ;
...
nret = DC4CInitEnv( & penv , “127.0.0.1:12001” ) ;
```

4.1.1.2DC4CCleanEnv

函数原型： void DC4CCleanEnv(struct Dc4cApiEnv **ppenv);

函数描述： 清理任务环境

函数说明： 清理并释放 Dc4cApiEnv 环境结构所需内存。

代码示例：

```
struct Dc4cApiEnv *penv = NULL ;
...
nret = DC4CInitEnv( & penv , “127.0.0.1:12001” ) ;
...
DC4CCleanEnv( & penv );
```

4.1.1.3DC4CSetTimeout

函数原型： void DC4CSetTimeout(struct Dc4cApiEnv *penv , int timeout);

函数描述： 设置公共超时时间

函数说明： struct Dc4cApiEnv *penv 任务环境结构指针

int timeout 公共超时时间（单位：秒）。实际任务超时时间取公共超时时间与配置任务时间较大值

4.1.1.4DC4CGetTimeout

函数原型： int DC4CGetTimeout(struct Dc4cApiEnv *penv);

函数描述： 得到公共超时时间

函数说明： struct Dc4cApiEnv *penv 批量任务环境结构指针

返回公共超时时间

4.1.1.5DC4CSetOptions

函数原型： void DC4CSetOptions(struct Dc4cApiEnv *penv , unsigned long options);

函数描述： 设置公共选项

函数说明： struct Dc4cApiEnv *penv 任务环境结构指针

unsigned long options 公共选项

备　　注： 选项由如下宏组合而成

DC4C_OPTIONS_INTERRUPT_BY_APP 允许应用返回 status 非 0 时中断后续任务执行，缺省为不中断

DC4C_OPTIONS_BIND_CPU 尝试在任务程序开始执行前绑定 CPU

代码示例：

```
DC4CSetOptions( penv , DC4C_OPTIONS_INTERRUPT_BY_APP |
DC4C_OPTIONS_BIND_CPU );
```

4.1.1.6DC4CGetOptions

函数原型： unsigned long DC4CGetOptions(struct Dc4cApiEnv *penv);

函数描述： 得到公共选项

函数说明： struct Dc4cApiEnv *penv 任务环境结构指针

返回公共选项

4.1.2 同步发起任务类

4.1.2.1DC4CDoTask

函数原型： int DC4CDoTask(struct Dc4cApiEnv *penv , char *program_and_params , int timeout);

函数描述： 分发单任务，同步等待结束

函数说明： struct Dc4cApiEnv *penv 任务环境结构指针

char *program_and_params 执行命令行

int timeout 配置超时时间（单位：秒）

函数返回值为 0 表示成功，非 0 表示失败

代码示例：

```
struct Dc4cApiEnv      *penv = NULL ;
...
nret = DC4CInitEnv( & penv , "127.0.0.1:12001" ) ;
...
nret = DC4CDoTask( penv , "dc4c_test_worker_sleep 5", 60 ) ;
...
DC4CCleanEnv( & penv );
```

4.1.2.2DC4CDoBatchTasks

函数原型： int DC4CDoBatchTasks(struct Dc4cApiEnv *penv , int workers_count , struct Dc4cBatchTask *a_tasks , int tasks_count);

函数描述： 分发批量任务，同步等待全部结束

函数说明： struct Dc4cApiEnv *penv 任务环境结构指针

int workers_count 并发数量。当为-1 时尽可能大并发

struct Dc4cBatchTask *a_tasks 任务集合，每个任务包含执行命令行和配置超时时间

int tasks_count 任务数量

函数返回值为 0 表示成功，非 0 表示失败

代码示例：

```
struct Dc4cApiEnv      *penv = NULL ;
struct Dc4cBatchTask *tasks_array = NULL ;
...
nret = DC4CInitEnv( & penv , "127.0.0.1:12001" ) ;
...
tasks_array = (struct Dc4cBatchTask *)malloc( sizeof(struct Dc4cBatchTask) *
tasks_count ) ;
for( i = 0 , p_task = tasks_array ; i < tasks_count ; i++ , p_task++ )
{
      strcpy( p_task->program_and_params , "dc4c_test_worker_sleep -5" );
      p_task->timeout = DC4CGetTimeout(penv) ;
}
...
nret = DC4CDoBatchTasks( penv , workers_count , tasks_array , tasks_count ) ;
...
free( tasks_array );
...
DC4CCleanEnv( & penv );
```

4.1.2.3 DC4CDoMultiBatchTasks

函数原型： int DC4CDoMultiBatchTasks(struct Dc4cApiEnv **a_penv , int envs_count , int *a_workers_count , struct Dc4cBatchTask **aa_tasks , int *a_tasks_count);

函数描述： 分发多批量任务，同步等待全部结束

函数说明： struct Dc4cApiEnv **a_penv 任务环境结构数组指针

envs_count 任务环境结构数组大小

int *aworkers_count 并发数量数组

struct Dc4cBatchTask **aa_tasks 任务集合数组

int *a_tasks_count 任务数量数组

函数返回值为 0 表示成功，非 0 表示失败

代码示例： （详见 test/dc4c_test_multi_batch_master.c）

4.1.3 获取执行反馈类

4.1.3.1DC4CGetTask*

函数原型： * DC4CGetTask*(struct Dc4cApiEnv *penv , …);

函数描述： 得到单任务反馈信息

函数说明： struct Dc4cApiEnv *penv 任务环境结构指针

函数返回值为 0 表示成功，非 0 表示失败

代码示例：

```
nret = DC4CDoTask( penv , argv[2] , DC4CGetTimeout(penv) ) ;
...
printf( "[%s][%ld]-[%s][%s][%d][%s][%s][%d]-[%d][%d][%s]\n"
      , DC4CGetTaskIp(penv) , DC4CGetTaskPort(penv)
      , DC4CGetTaskTid(penv) , DC4CGetTaskProgramAndParams(penv) ,
DC4CGetTaskTimeout(penv) ,
ConvertTimeString(DC4CGetTaskBeginTimestamp(penv),begin_timebuf,sizeof(begin_time
buf))+11 ,
ConvertTimeString(DC4CGetTaskEndTimestamp(penv),end_timebuf,sizeof(end_timebuf))+
11 , DC4CGetTaskElapse(penv)
      , DC4CGetTaskError(penv) , WEXITSTATUS(DC4CGetTaskStatus(penv)) ,
DC4CGetTaskInfo(penv) );
```

4.1.3.2DC4CGetBatchTasks*

函数原型： * DC4CGetBatchTasks*(struct Dc4cApiEnv *penv , int index , …);

函数描述： 得到批量任务反馈信息

函数说明： struct Dc4cApiEnv *penv 任务环境结构指针集合

int index 任务索引，从 0 开始

（其它同 DC4CGetTask*）

函数返回值为 0 表示成功，非 0 表示失败

4.1.4 低层函数类

4.1.4.1DC4CBeginBatchTasks

函数原型： int DC4CBeginBatchTasks(struct Dc4cApiEnv *penv , int workers_count , struct Dc4cBatchTask *a_tasks , int tasks_count);

函数描述： 开始批量任务

函数说明： （同同步发起批量任务 DC4CDoBatchTasks）

函数返回值为 0 表示成功，非 0 表示失败

4.1.4.2DC4CPerformBatchTasks

函数原型： int DC4CPerformBatchTasks(struct Dc4cApiEnv *penv , int *p_task_index);

函数描述： 处理多批量任务

函数说明： struct Dc4cApiEnv *penvs 批量任务环境结构指针

int *p_task_index 当前完成任务索引

函数返回值为 DC4C_INFO_TASK_FINISHED 表示有任务结束了，DC4C_INFO_BATCH_TASKS_FINISHED 为批量任务都结束了，DC4C_ERROR_TIMEOUT 为有任务超时了，DC4C_ERROR_APP 为有任务程序返回非 0，其它表示失败

代码示例： （详见 test/dc4c_test_batch_master_stepbystep.c）

4.1.4.3DC4CPerformMultiBatchTasks

函数原型： int DC4CPerformMultiBatchTasks(struct Dc4cApiEnv **ppenvs , int

envs_count , struct Dc4cApiEnv **ppenv , int *p_task_index);

函数描述： 多路复用处理多批量任务

函数说明： struct Dc4cApiEnv **ppenvs 批量任务环境结构指针集合

int envs_count 批量任务环境数量

struct Dc4cApiEnv **ppenv 输出当前完成或失败批量任务

int *p_task_index 当前完成任务索引

函数返回值为 DC4C_INFO_TASK_FINISHED 表示有任务结束了，DC4C_INFO_BATCH_TASKS_FINISHED 为批量任务都结束了，DC4C_INFO_ALL_ENVS_FINISHED 为所有批量都结束了，DC4C_ERROR_TIMEOUT 为有任务超时了，DC4C_ERROR_APP 为有任务程序返回非 0，其它表示失败

代码示例： （详见 test/dc4c_test_multi_batch_master_stepbystep.c）

4.1.4.4DC4CQueryWorkers

函数原型： int DC4CQueryWorkers(struct Dc4cApiEnv *penv);

函数描述： 向注册节点查询空闲计算节点信息

函数说明： struct Dc4cApiEnv *penv 任务环境结构指针

函数返回值为 0 表示成功，非 0 表示失败

4.1.4.5DC4CSetTasksFds

函数原型： int DC4CSetTasksFds(struct Dc4cApiEnv *penv , struct Dc4cApiEnv *penv_QueryWorkers , fd_set *read_fds , fd_set *write_fds , fd_set *expect_fds , int *p_max_fd);

函数描述： 从已连接计算节点会话和连接空闲计算节点中会话（发送执行命令请求），设置描述字集合

函数说明： struct Dc4cApiEnv *penv 任务环境结构指针

struct Dc4cApiEnv *penv_QueryWorkers 用于连接注册节点的环境结构指针，如为 NULL，则使用 penv 中的查询计算节点信息

fd_set *read_fds 可读事件描述字集合

fd_set *write_fds （目前未用）

fd_set *expect_fds （目前未用）

int *p_max_fd 可读事件描述字集合中的最大描述字值

函数返回值为 0 表示成功，非 0 表示失败

4.1.4.6DC4CSelectTasksFds

函数原型： int DC4CSelectTasksFds(fd_set *p_read_fds , fd_set *write_fds , fd_set *expect_fds , int *p_max_fd , int select_timeout);

函数描述： 多路复用等待可读描述字集合事件

函数说明： struct Dc4cApiEnv *penv 任务环境结构指针

fd_set *read_fds 可读事件描述字集合

fd_set *write_fds （目前未用）

fd_set *expect_fds （目前未用）

int *p_max_fd 可读事件描述字集合中的最大描述字值

int select_timeout 等待事件最长时间

函数返回值为 0 表示成功，非 0 表示失败

4.1.4.7DC4CProcessTasks

函数原型： int DC4CProcessTasks(struct Dc4cApiEnv *penv , fd_set *p_read_fds , fd_set *write_fds , fd_set *expect_fds);

函数描述： 处理可读描述字集合事件，接收执行命令响应、接收分发程序请求和发送响应

函数说明： struct Dc4cApiEnv *penv 任务环境结构指针

fd_set *read_fds 可读事件描述字集合

fd_set *write_fds （目前未用）

fd_set *expect_fds （目前未用）

函数返回值为 0 表示成功，非 0 表示失败

4.1.5 其它类

4.1.5.1DC4CResetFinishedTasksWithError

函数原型： void DC4CResetFinishedTasksWithError(struct Dc4cApiEnv *penv);

函数描述： 重置批量任务中程序执行返回失败的任务为待处理

函数说明： struct Dc4cApiEnv *penv 任务环境结构指针

4.1.5.2DC4CGetUnusedWorkersCount

函数原型： int DC4CGetUnusedWorkersCount(struct Dc4cApiEnv *penv);

函数描述： 得到上次向注册节点查询空闲计算节点还未使用信息

函数说明： struct Dc4cApiEnv *penv 批量任务环境

4.2 计算节点接口

用户节点分发过来到计算节点的应用调用的函数接口，主要信息处理完后的设置反馈信息等功能。

4.2.1 日志操作类

4.2.1.1 DC4CSetAppLogFile

函数原型： void DC4CSetAppLogFile(char *program);

函数描述： 如果计算节点应用使用 DC4C 日志函数库，可以调用该函数重定向日志文件

函数说明： char *program 应用名

备　　注： 重定向后的日志文件名格式为”dc4c_wserver_(序号)_(IP:PORT).(应用名).log”

4.2.2 反馈信息类

4.2.2.1 DC4CFormatReplyInfo

函数原型： int DC4CFormatReplyInfo(char *format , ...);

函数描述： 用于计算节点应用反馈信息给用户节点，类型为字符串，最长 1024 字符

函数说明： char *format 格式化串

备　　注： （同 snprintf）

4.2.2.2 DC4CSetReplyInfo

函数原型： int DC4CSetReplyInfo(char *str);

函数描述： 用于计算节点应用反馈信息给用户节点，类型为字符串，最长 1024 字符

函数说明： （同上，非格式化串直接复制版本）

4.2.2.3DC4CSetReplyInfoEx

函数原型： int DC4CSetReplyInfoEx(char *buf , int len);

函数描述： 用于计算节点应用反馈信息给用户节点，类型为字符串，最长 1024 字符

函数说明： （同上，非格式化串带长度版本）

4.3 任务调度引擎接口

4.3.1 高层函数

4.3.1.1DC4CLoadDagScheduleFromFile

函数原型： int DC4CLoadDagScheduleFromFile(struct Dc4cDagSchedule **pp_sched , char *pathfilename , char *rservers_ip_port , int options);

函数描述： 从外部配置文件中载入 DAG 流程模型的任务树配置

函数说明： 申请 Dc4cDagScheduleDAG 流程模型环境结构所需内存，并初始化，从配置文件载入 DAG 流程模型任务数配置，让(*pp_sched)指向该结构。

char *pathfilename 带路径的外部配置文件名

char *rservers_ip_port 注册节点地址

int options 传递给用户节点接口的 DC4CSetOptions 用于设置全局选项

函数返回值为 0 表示成功，非 0 表示失败

备　　注： 最多配置 1000 个批量任务，1000 个批量任务关系，每个批量任务最多配置 1000 个任务

代码示例：

```
struct Dc4cDagSchedule     *p_sched = NULL ;
...
nret = DC4CLoadDagScheduleFromFile( & p_sched ,
```

```
"/home/calvin/etc/test.dag_schedule" , DC4C_OPTIONS_INTERRUPT_BY_APP ) ;
...
DC4CUnloadDagSchedule( & p_sched );
```

4.3.1.2 DC4CUnloadDagSchedule

函数原型： void DC4CUnloadDagSchedule(struct Dc4cDagSchedule **pp_sched);

函数描述： 卸载 DAG 流程模型的任务树配置

函数说明： struct Dc4cDagSchedule **pp_sched　DAG 流程模型环境结构指针的地址

4.3.1.3 DC4CLogDagSchedule

函数原型： void DC4CLogDagSchedule(struct Dc4cDagSchedule *p_sched);

函数描述： 把 DAG 流程模型的任务树配置输出到日志

函数说明： struct Dc4cDagSchedule *p_sched　DAG 流程模型环境结构指针

4.3.1.4 DC4CExecuteDagSchedule

函数原型： int DC4CExecuteDagSchedule(struct Dc4cDagSchedule *p_sched , char *rservers_ip_port);

函数描述： 执行 DAG 模型的任务流

函数说明： struct Dc4cDagSchedule *p_sched　DAG 流程模型环境结构指针

char *rservers_ip_port 注册服务器地址

函数返回值为 0 表示成功，非 0 表示失败

代码示例：

```
struct Dc4cDagSchedule    *p_sched = NULL ;
...
nret = DC4CLoadDagScheduleFromDatabase( & p_sched , "test_dag_schedule" ,
DC4C_OPTIONS_INTERRUPT_BY_APP ) ;
...
nret = DC4CExecuteDagSchedule( p_sched , NULL ) ;
```

```
...
DC4CUnloadDagSchedule( & p_sched );
```

4.3.1.5DC4CBeginDagSchedule

函数原型： int DC4CBeginDagSchedule(struct Dc4cDagSchedule *p_sched);

函数描述： 开始 DAG 模型的任务流

函数说明： struct Dc4cDagSchedule *p_sched DAG 流程模型环境结构指针

函数返回值为 0 表示成功，非 0 表示失败

备　　注： DC4CBeginDagSchedule 和 DC4CPerformDagSchedule 简单组合成 DC4CExecuteDagSchedule。当应用想在每个批量任务处理完后做一些自己的事情，可调用分拆版本，否则一次性调用组合版本即可。

4.3.1.6DC4CPerformDagSchedule

函数原型： int DC4CPerformDagSchedule(struct Dc4cDagSchedule *p_sched , struct Dc4cDagBatch **pp_batch);

函数描述： 处理 DAG 模型的任务流

函数说明： struct Dc4cDagSchedule *p_sched DAG 流程模型环境结构指针

struct Dc4cDagBatch **pp_batch 当一个批量任务结束后抛出批量节点给应用

函数返回值为 DC4C_INFO_NO_UNFINISHED_ENVS 时表示所有批量都结束了，为 0 表示单个批量任务结束了，非 0 表示失败

代码示例：

```
struct Dc4cDagSchedule     *p_sched = NULL ;
struct Dc4cApiEnv      *penv = NULL ;
...
nret = DC4CLoadDagScheduleFromDatabase( & p_sched , "test_dag_schedule" ,
"127.0.0.1:12001" , DC4C_OPTIONS_INTERRUPT_BY_APP ) ;
...
nret = DC4CBeginDagSchedule( p_sched ) ;
...
```

```
while(1)
{
    nret = DC4CPerformDagSchedule( p_sched , & p_batch ) ;
    if( nret == DC4C_INFO_NO_UNFINISHED_ENVS )
        break;
    DC4CGetDagBatchApiEnvPPtr( p_batch , & penv );
    ...
}
...
DC4CUnloadDagSchedule( & p_sched );
```

4.3.2 低层函数

4.3.2.1DC4CLoadDagScheduleFromStruct

函数原型： int DC4CLoadDagScheduleFromStruct(struct Dc4cDagSchedule **pp_sched , dag_schedule_configfile *p_config , int options);

函数描述： 从大配置结构体载入 DAG 流程模型的任务树配置

函数说明： 申请 Dc4cDagScheduleDAG 流程模型环境结构所需内存，并初始化，从大配置结构体载入 DAG 流程模型任务数配置，让(*pp_sched)指向该结构。

int options 传递给用户节点接口的 DC4CSetOptions 用于设置全局选项

函数返回值为 0 表示成功，非 0 表示失败

备　　注： 此函数被 DC4CLoadDagScheduleFromFile 、DC4CLoadDagScheduleFromDatabase 调用，实现载入配置。

4.3.2.2DC4CAllocDagBatch

函数原型： struct Dc4cDagBatch *DC4CAllocDagBatch(struct Dc4cDagSchedule *p_sched , char *batch_name , char *batch_desc , int view_pos_x , int view_pos_y);

函数描述： 创建一个 DAG 批量树节点，并初始化

函数说明： struct Dc4cDagSchedule *p_sched DAG 流程模型结构指针

char *batch_name 批量名

char *batch_desc 批量描述

int view_pos_x 批量节点图形配置坐标 X

int view_pos_y 批量节点图形配置坐标 Y

4.3.2.3DC4CFreeDagBatch

函数原型： BOOL DC4CFreeDagBatch(void *pv);

函数描述： 清理一个 DAG 批量树节点，并释放之

函数说明： void *pv DAG 流程模型结构指针

4.3.2.4DC4CLinkDagBatch

函数原型： int DC4CLinkDagBatch(struct Dc4cDagSchedule *p_sched , struct Dc4cDagBatch *p_parent_batch , struct Dc4cDagBatch *p_batch);

函数描述： 挂接一个批量节点到 DAG 流程模型树上

函数说明： struct Dc4cDagSchedule *p_sched DAG 流程模型结构指针

struct Dc4cDagBatch *p_parent_batch 父级批量节点

struct Dc4cDagBatch *p_batch 要挂接的批量节点

4.3.2.5DC4CGetDagBatchName

函数原型： void DC4CGetDagBatchName(struct Dc4cDagBatch *p_batch , char **pp_name);

函数描述： 得到批量任务的名字

函数说明： struct Dc4cDagBatch *p_batch 要挂接的批量节点

char **pp_name 批量节点的名字

4.3.2.6DC4CSetDagBatchTasks

函数原型： void DC4CSetDagBatchTasks(struct Dc4cDagBatch *p_batch , struct Dc4cBatchTask *a_tasks , int tasks_count);

函数描述： 设置批量任务集合到批量节点上

函数说明： struct Dc4cDagBatch *p_batch 要挂接的批量节点

struct Dc4cBatchTask *a_tasks 批量任务集合

int tasks_count 批量任务数量

4.3.2.7DC4CGetDagBatchApiEnvPPtr

函数原型： struct Dc4cApiEnv **DC4CGetDagBatchApiEnvPPtr(struct Dc4cDagBatch *p_batch , struct Dc4cApiEnv **pp_env);

函数描述： 返回批量节点里的批量任务环境结构指针的地址

函数说明： struct Dc4cDagBatch *p_batch 要挂接的批量节点

struct Dc4cApiEnv **pp_env 批量节点里的批量任务环境结构指针的地址

函数返回值也是批量节点里的批量任务环境结构指针的地址

4.3.2.8DC4CGetDagBatchBeginDatetimeStamp

函数原型： void DC4CGetDagBatchBeginDatetimeStamp (struct Dc4cDagBatch *p_batch , char begin_datetime[19+1] , long *p_begin_datetime_stamp);

函数描述： 返回批量节点里的批量开始日期时间戳

函数说明： struct Dc4cDagBatch *p_batch 批量节点

char begin_datetime[19+1] 存放返回的日期时间字符串

long *p_begin_datetime_stamp 存放返回的日期时间戳的地址

4.3.2.9DC4CGetDagBatchEndDatetimeStamp

函数原型： void DC4CGetDagBatchEndDatetimeStamp(struct Dc4cDagBatch *p_batch , char end_datetime[19+1] , long *p_end_datetime_stamp);

函数描述： 返回批量节点里的批量结束日期时间戳

函数说明： struct Dc4cDagBatch *p_batch 批量节点

char end_datetime[19+1] 存放返回的日期时间字符串

long *p_begin_datetime_stamp 存放返回的日期时间戳的地址

4.4 代码示例

4.4.1 单任务分派

源代码 test/dc4c_test_master.c

```
#include "dc4c_api.h"

/* for testing
time ./dc4c_test_master 192.168.6.54:12001,192.168.6.54:12002 "dc4c_test_worker_sleep 3"
*/

int main( int argc , char *argv[] )
{
	struct Dc4cApiEnv	*penv = NULL ;

	char		begin_timebuf[ 256 + 1 ] ;
	char		end_timebuf[ 256 + 1 ] ;

	int		nret = 0 ;

	DC4CSetAppLogFile( "dc4c_test_master" );
	SetLogLevel( LOGLEVEL_DEBUG );
```

```
	if( argc == 1 + 2 )
	{
		nret = DC4CInitEnv( & penv , argv[1] ) ;
		if( nret )
		{
			printf( "DC4CInitEnv failed[%d]\n" , nret );
			return 1;
		}
		else
		{
			printf( "DC4CInitEnv ok\n" );
		}

		DC4CSetTimeout( penv , 15 );
		DC4CSetOptions( penv , DC4C_OPTIONS_INTERRUPT_BY_APP );

		nret = DC4CDoTask( penv , argv[2] , DC4CGetTimeout(penv) ) ;
		if( nret )
		{
			printf( "DC4CDoTask failed[%d]\n" , nret );
		}
		else
		{
			printf( "DC4CDoTask ok\n" );
		}

		printf( "[%s][%ld]-[%s][%s][%d][%s][%s][%d]-[%d][%d][%s]\n"
			, DC4CGetTaskIp(penv) , DC4CGetTaskPort(penv)
			, DC4CGetTaskTid(penv) , DC4CGetTaskProgramAndParams(penv) ,
DC4CGetTaskTimeout(penv) ,
ConvertTimeString(DC4CGetTaskBeginTimestamp(penv),begin_timebuf,sizeof(begin_timebuf))+11 ,
ConvertTimeString(DC4CGetTaskEndTimestamp(penv),end_timebuf,sizeof(end_timebuf))+11 ,
DC4CGetTaskElapse(penv)
			, DC4CGetTaskError(penv) , WEXITSTATUS(DC4CGetTaskStatus(penv)) ,
DC4CGetTaskInfo(penv) );

		DC4CCleanEnv( & penv );
		printf( "DC4CCleanEnv ok\n" );
	}
	else
	{
		printf( "USAGE : dc4c_test_master rserver_ip:rserver_port program_and_params\n" );
		exit(7);
	}
```

```
	return 0;
}
```

4.4.2 批量任务分派（高层函数）

源代码 test/dc4c_test_batch_master.c

```
#include "dc4c_api.h"

/* for testing
time ./dc4c_test_batch_master 192.168.6.54:12001,192.168.6.54:12002 2 3 "dc4c_test_worker_sleep 1"
"dc4c_test_worker_sleep 2" "dc4c_test_worker_sleep 3"
time ./dc4c_test_batch_master 192.168.6.54:12001,192.168.6.54:12002 4 -10 "dc4c_test_worker_sleep 10"
time ./dc4c_test_batch_master 192.168.6.54:12001,192.168.6.54:12002 -1 -100 "dc4c_test_worker_sleep
-10"
time ./dc4c_test_batch_master 192.168.6.54:12001,192.168.6.54:12002 -2 -100 "dc4c_test_worker_sleep
-10"
*/

int main( int argc , char *argv[] )
{
	struct Dc4cApiEnv	*penv = NULL ;
	struct Dc4cBatchTask *tasks_array = NULL ;
	struct Dc4cBatchTask *p_task = NULL ;
	int			tasks_count ;
	int			workers_count ;
	int			repeat_task_flag ;
	int			i ;

	char			begin_timebuf[ 256 + 1 ] ;
	char			end_timebuf[ 256 + 1 ] ;

	int			nret = 0 ;

	DC4CSetAppLogFile( "dc4c_test_batch_master" );
	SetLogLevel( LOGLEVEL_DEBUG );

	if( argc >= 1 + 3 )
	{
		nret = DC4CInitEnv( & penv , argv[1] ) ;
		if( nret )
		{
```

```
        printf( "DC4CInitEnv failed[%d]\n" , nret );
        return 1;
}
else
{
        printf( "DC4CInitEnv ok\n" );
}

DC4CSetTimeout( penv , 15 );
DC4CSetOptions( penv , DC4C_OPTIONS_INTERRUPT_BY_APP );

workers_count = atoi(argv[2]) ;
tasks_count = atoi(argv[3]) ;

if( tasks_count < 0 )
{
        tasks_count = -tasks_count ;
        repeat_task_flag = 1 ;
}
else
{
        repeat_task_flag = 0 ;
}

if( workers_count < 0 )
{
        if( workers_count == -2 )
        {
                nret = DC4CQueryWorkers( penv ) ;
                if( nret )
                {
                        printf( "DC4CQueryWorkers failed[%d]\n" , nret );
                        return 1;
                }

                workers_count = DC4CGetUnusedWorkersCount( penv ) ;
                if( workers_count <= 0 )
                {
                        printf( "workers_count[%d] invalid\n" , workers_count );
                        return 1;
                }

                tasks_count = workers_count ;
        }
```

```
		else
		{
			workers_count = tasks_count ;
		}
	}

	tasks_array = (struct Dc4cBatchTask *)malloc( sizeof(struct Dc4cBatchTask) * tasks_count ) ;
	if( tasks_array == NULL )
	{
		printf( "alloc failed , errno[%d]\n" , errno );
		return 1;
	}
	memset( tasks_array , 0x00 , sizeof(struct Dc4cBatchTask) * tasks_count );

	for( i = 0 , p_task = tasks_array ; i < tasks_count ; i++ , p_task++ )
	{
		if( repeat_task_flag == 1 )
			strcpy( p_task->program_and_params , argv[4] );
		else
			strcpy( p_task->program_and_params , argv[4+i] );
		p_task->timeout = DC4CGetTimeout(penv) ;
	}

	nret = DC4CDoBatchTasks( penv , workers_count , tasks_array , tasks_count ) ;
	free( tasks_array );
	if( nret )
	{
		printf( "DC4CDoBatchTasks failed[%d]\n" , nret );
	}
	else
	{
		printf( "DC4CDoBatchTasks ok\n" );
	}

	tasks_count = DC4CGetTasksCount( penv ) ;
	for( i = 0 ; i < tasks_count ; i++ )
	{
		printf( "[%d]-[%s][%ld]-[%s][%s][%d][%s][%s][%d]-[%d][%d][%s]\n"
			, i , DC4CGetBatchTasksIp(penv,i) , DC4CGetBatchTasksPort(penv,i)
			, DC4CGetBatchTasksTid(penv,i) , DC4CGetBatchTasksProgramAndParams(penv,i) ,
DC4CGetBatchTasksTimeout(penv,i) ,
ConvertTimeString(DC4CGetBatchTasksBeginTimestamp(penv,i),begin_timebuf,sizeof(begin_timebuf))+11 ,
ConvertTimeString(DC4CGetBatchTasksEndTimestamp(penv,i),end_timebuf,sizeof(end_timebuf))+11 ,
DC4CGetBatchTasksElapse(penv,i)
```

```
                        , DC4CGetBatchTasksError(penv,i) , WEXITSTATUS(DC4CGetBatchTasksStatus(penv,i)) ,
DC4CGetBatchTasksInfo(penv,i) );
            }

            DC4CCleanEnv( & penv );
            printf( "DC4CCleanEnv ok\n" );
        }
        else
        {
            printf( "USAGE : dc4c_test_batch_master rserver_ip:rserver_port,... workers_count tasks_count
program_and_params_1 ...\n" );
            exit(7);
        }

        return 0;
}
```

4.4.3 批量任务分派（低层函数）

源代码 test/dc4c_test_batch_master_stepbystep.c

```
#include "dc4c_api.h"

/* for testing
time ./dc4c_test_batch_master 192.168.6.54:12001,192.168.6.54:12002 2 3 "dc4c_test_worker_sleep 1"
"dc4c_test_worker_sleep 2" "dc4c_test_worker_sleep 3"
time ./dc4c_test_batch_master 192.168.6.54:12001,192.168.6.54:12002 4 -10 "dc4c_test_worker_sleep 10"
time ./dc4c_test_batch_master 192.168.6.54:12001,192.168.6.54:12002 -1 -100 "dc4c_test_worker_sleep
-10"
time ./dc4c_test_batch_master 192.168.6.54:12001,192.168.6.54:12002 -2 -100 "dc4c_test_worker_sleep
-10"
*/

int main( int argc , char *argv[] )
{
        struct Dc4cApiEnv      *penv = NULL ;
        struct Dc4cBatchTask *tasks_array = NULL ;
        struct Dc4cBatchTask *p_task = NULL ;
        int                    tasks_count ;
        int                    task_index ;
        int                    workers_count ;
        int                    repeat_task_flag ;
        int                    i ;
```

```
char            begin_timebuf[ 256 + 1 ] ;
char            end_timebuf[ 256 + 1 ] ;

int             nret = 0 ;

DC4CSetAppLogFile( "dc4c_test_batch_master_stepbystep" );
SetLogLevel( LOGLEVEL_DEBUG );

if( argc >= 1 + 3 )
{
        nret = DC4CInitEnv( & penv , argv[1] ) ;
        if( nret )
        {
                printf( "DC4CInitEnv failed[%d]\n" , nret );
                return 1;
        }
        else
        {
                printf( "DC4CInitEnv ok\n" );
        }

        DC4CSetTimeout( penv , 15 );
        DC4CSetOptions( penv , DC4C_OPTIONS_INTERRUPT_BY_APP );

        workers_count = atoi(argv[2]) ;
        tasks_count = atoi(argv[3]) ;

        if( tasks_count < 0 )
        {
                tasks_count = -tasks_count ;
                repeat_task_flag = 1 ;
        }
        else
        {
                repeat_task_flag = 0 ;
        }

        if( workers_count < 0 )
        {
                if( workers_count == -2 )
                {
                        nret = DC4CQueryWorkers( penv ) ;
                        if( nret )
```

```
                {
                        printf( "DC4CQueryWorkers failed[%d]\n" , nret );
                        return 1;
                }

                workers_count = DC4CGetUnusedWorkersCount( penv ) ;
                if( workers_count <= 0 )
                {
                        printf( "workers_count[%d] invalid\n" , workers_count );
                        return 1;
                }

                tasks_count = workers_count ;
        }
        else
        {
                workers_count = tasks_count ;
        }
}

tasks_array = (struct Dc4cBatchTask *)malloc( sizeof(struct Dc4cBatchTask) * tasks_count ) ;
if( tasks_array == NULL )
{
        printf( "alloc failed , errno[%d]\n" , errno );
        return 1;
}
memset( tasks_array , 0x00 , sizeof(struct Dc4cBatchTask) * tasks_count );

for( i = 0 , p_task = tasks_array ; i < tasks_count ; i++ , p_task++ )
{
        if( repeat_task_flag == 1 )
                strcpy( p_task->program_and_params , argv[4] );
        else
                strcpy( p_task->program_and_params , argv[4+i] );
        p_task->timeout = DC4CGetTimeout(penv) ;
}

nret = DC4CBeginBatchTasks( penv , workers_count , tasks_array , tasks_count ) ;
free( tasks_array );
if( nret )
{
        printf( "DC4CBeginBatchTasks failed[%d]\n" , nret );
}
else
```

```
	{
		printf( "DC4CBeginBatchTasks ok\n" );
	}

	while(1)
	{
		nret = DC4CPerformBatchTasks( penv , & task_index ) ;
		if( nret == DC4C_INFO_TASK_FINISHED )
		{
			printf( "DC4CPerformBatchTasks return DC4C_INFO_TASK_FINISHED\n" );
			printf( "[%d]-[%s][%ld]-[%s][%s][%d][%s][%s][%d]-[%d][%d][%s]\n"
				, task_index , DC4CGetBatchTasksIp(penv,task_index) ,
DC4CGetBatchTasksPort(penv,task_index)
				, DC4CGetBatchTasksTid(penv,task_index) ,
DC4CGetBatchTasksProgramAndParams(penv,task_index) , DC4CGetBatchTasksTimeout(penv,task_index) ,
ConvertTimeString(DC4CGetBatchTasksBeginTimestamp(penv,task_index),begin_timebuf,sizeof(begin_time
buf))+11 ,
ConvertTimeString(DC4CGetBatchTasksEndTimestamp(penv,task_index),end_timebuf,sizeof(end_timebuf))+
11 , DC4CGetBatchTasksElapse(penv,task_index)
				, DC4CGetBatchTasksError(penv,task_index) ,
WEXITSTATUS(DC4CGetBatchTasksStatus(penv,task_index)) , DC4CGetBatchTasksInfo(penv,task_index) );
		}
		else if( nret == DC4C_INFO_BATCH_TASKS_FINISHED )
		{
			printf( "DC4CPerformBatchTasks return DC4C_INFO_BATCH_TASKS_FINISHED\n" );
			break;
		}
		else if( nret == DC4C_ERROR_TIMEOUT )
		{
			printf( "DC4CPerformBatchTasks return DC4C_ERROR_TIMEOUT\n" );
			break;
		}
		else if( nret == DC4C_ERROR_APP )
		{
			printf( "DC4CPerformBatchTasks return DC4C_ERROR_APP\n" );
			break;
		}
		else
		{
			printf( "DC4CPerformBatchTasks failed[%d]\n" , nret );
			break;
		}
	}
```

```
		DC4CCleanEnv( & penv );
		printf( "DC4CCleanEnv ok\n" );
	}
	else
	{
		printf( "USAGE : dc4c_test_batch_master rserver_ip:rserver_port,... workers_count tasks_count
program_and_params_1 ...\n" );
		exit(7);
	}

	return 0;
}
```

4.4.4 多批量任务分派（高层函数）

源代码 test/dc4c_test_multi_batch_master.c

```
#include "dc4c_api.h"

/* for testing
time ./dc4c_test_multi_batch_master 192.168.6.54:12001,192.168.6.54:12002 3 -1 -100
"dc4c_test_worker_sleep -10"
*/

int main( int argc , char *argv[] )
{
	struct Dc4cApiEnv	**a_penv = NULL ;
	int			*a_tasks_count ;
	int			*a_workers_count ;
	struct Dc4cBatchTask **a_tasks_array = NULL ;
	int			envs_count ;
	int			envs_index ;
	struct Dc4cApiEnv	*penv = NULL ;
	int			tasks_count ;
	int			task_index ;
	struct Dc4cBatchTask *p_task = NULL ;
	int			repeat_task_flag ;

	char			begin_timebuf[ 256 + 1 ] ;
	char			end_timebuf[ 256 + 1 ] ;

	int			nret = 0 ;
```

```
DC4CSetAppLogFile( "dc4c_test_multi_batch_master" );
SetLogLevel( LOGLEVEL_DEBUG );

if( argc >= 1 + 4 )
{
	envs_count = atoi(argv[2]) ;

	a_penv = (struct Dc4cApiEnv**)malloc( sizeof(struct Dc4cApiEnv*) * envs_count ) ;
	if( a_penv == NULL )
	{
		printf( "malloc failed[%d] , errno[%d]\n" , nret , errno );
		return 1;
	}
	memset( a_penv , 0x00 , sizeof(struct Dc4cApiEnv*) * envs_count );

	a_tasks_count = (int*)malloc( sizeof(int) * envs_count ) ;
	if( a_tasks_count == NULL )
	{
		printf( "malloc failed[%d] , errno[%d]\n" , nret , errno );
		return 1;
	}
	memset( a_tasks_count , 0x00 , sizeof(int) * envs_count );

	a_workers_count = (int*)malloc( sizeof(int) * envs_count ) ;
	if( a_workers_count == NULL )
	{
		printf( "malloc failed[%d] , errno[%d]\n" , nret , errno );
		return 1;
	}
	memset( a_workers_count , 0x00 , sizeof(int) * envs_count );


	if( a_tasks_array == NULL )
	{
		printf( "malloc failed[%d] , errno[%d]\n" , nret , errno );
		return 1;
	}
	memset( a_tasks_array , 0x00 , sizeof(struct Dc4cBatchTask*) * envs_count );

	for( envs_index = 0 ; envs_index < envs_count ; envs_index++ )
	{
		nret = DC4CInitEnv( & (a_penv[envs_index]) , argv[1] ) ;
		if( nret )
		{
```

```
        printf( "DC4CInitEnv failed[%d]\n" , nret );
        return 1;
}
else
{
        printf( "DC4CInitEnv ok , penv[%p]\n" , a_penv[envs_index] );
}

DC4CSetTimeout( a_penv[envs_index] , 15 );
DC4CSetOptions( a_penv[envs_index] , DC4C_OPTIONS_INTERRUPT_BY_APP );

a_workers_count[envs_index] = atoi(argv[3]) ;
a_tasks_count[envs_index] = atoi(argv[4]) ;

if( a_tasks_count[envs_index] < 0 )
{
        a_tasks_count[envs_index] = -a_tasks_count[envs_index] ;
        repeat_task_flag = 1 ;
}
else
{
        repeat_task_flag = 0 ;
}

if( a_workers_count[envs_index] < 0 )
{
        if( a_workers_count[envs_index] == -2 )
        {
                nret = DC4CQueryWorkers( penv ) ;
                if( nret )
                {
                        printf( "DC4CQueryWorkers failed[%d]\n" , nret );
                        return 1;
                }

                a_workers_count[envs_index] = DC4CGetUnusedWorkersCount( penv ) ;
                if( a_workers_count[envs_index] <= 0 )
                {
                        printf( "workers_count[%d] invalid\n" , a_workers_count[envs_index] );
                        return 1;
                }

                a_tasks_count[envs_index] = a_workers_count[envs_index] ;
        }
```

```
                        else
                        {
                                a_workers_count[envs_index] = a_tasks_count[envs_index] ;
                        }
                }

                a_tasks_array[envs_index] = (struct Dc4cBatchTask *)malloc( sizeof(struct Dc4cBatchTask)
* a_tasks_count[envs_index] ) ;
                if( a_tasks_array[envs_index] == NULL )
                {
                        printf( "alloc failed , errno[%d]\n" , errno );
                        return 1;
                }
                memset( a_tasks_array[envs_index] , 0x00 , sizeof(struct Dc4cBatchTask) *
a_tasks_count[envs_index] );

                for( task_index = 0 ; task_index < a_tasks_count[envs_index] ; task_index++ )
                {
                        p_task = & (a_tasks_array[envs_index][task_index]) ;

                        p_task->order_index = task_index ;
                        if( repeat_task_flag == 1 )
                                strcpy( p_task->program_and_params , argv[5] );
                        else
                                strcpy( p_task->program_and_params , argv[5+task_index] );
                        p_task->timeout = DC4CGetTimeout(a_penv[envs_index]) ;
                }
        }

        nret = DC4CDoMultiBatchTasks( a_penv , envs_count , a_workers_count , a_tasks_array ,
a_tasks_count ) ;
        if( nret )
        {
                printf( "DC4CDoMultiBatchTasks failed[%d] , penv[%p]\n" , nret , penv );
        }
        else
        {
                printf( "DC4CDoMultiBatchTasks ok\n" );

        }

        for( envs_index = 0 ; envs_index < envs_count ; envs_index++ )
        {
                penv = a_penv[envs_index] ;
```

```
                printf( "penv[%p]\n" , penv );
                tasks_count = DC4CGetTasksCount( penv ) ;
                for( task_index = 0 ; task_index < tasks_count ; task_index++ )
                {
                        printf( "[%d]-[%s][%ld]-[%s][%s][%d][%s][%s][%d]-[%d][%d][%s]\n"
                                , task_index , DC4CGetBatchTasksIp(penv,task_index) ,
DC4CGetBatchTasksPort(penv,task_index)
                                , DC4CGetBatchTasksTid(penv,task_index) ,
DC4CGetBatchTasksProgramAndParams(penv,task_index) , DC4CGetBatchTasksTimeout(penv,task_index) ,
ConvertTimeString(DC4CGetBatchTasksBeginTimestamp(penv,task_index),begin_timebuf,sizeof(begin_time
buf))+11 ,
ConvertTimeString(DC4CGetBatchTasksEndTimestamp(penv,task_index),end_timebuf,sizeof(end_timebuf))+
11 , DC4CGetBatchTasksElapse(penv,task_index)
                                , DC4CGetBatchTasksError(penv,task_index) ,
WEXITSTATUS(DC4CGetBatchTasksStatus(penv,task_index)) , DC4CGetBatchTasksInfo(penv,task_index) );
                }
        }

        for( envs_index = 0 ; envs_index < envs_count ; envs_index++ )
        {
                free( a_tasks_array[envs_index] );
                DC4CCleanEnv( & (a_penv[envs_index]) );
        }
        free( a_tasks_count );
        free( a_workers_count );
        free( a_penv );
        free( a_tasks_array );
        printf( "DC4CCleanEnv ok\n" );
    }
    else
    {
        printf( "USAGE : dc4c_test_multi_batch_master rserver_ip:rserver_port,... envs_count
workers_count task_count program_and_params_1 ...\n" );
        exit(7);
    }

    return 0;
}
```

4.4.5 多批量任务分派（低层函数）

源代码 test/dc4c_test_multi_batch_master.c

```
#include "dc4c_api.h"

/* for testing
time ./dc4c_test_multi_batch_master 192.168.6.54:12001,192.168.6.54:12002 3 -1 -100
"dc4c_test_worker_sleep -10"
*/

int main( int argc , char *argv[] )
{
	struct Dc4cApiEnv	**a_penv = NULL ;
	int			envs_count ;
	int			envs_index ;
	struct Dc4cApiEnv	*penv = NULL ;
	struct Dc4cBatchTask *tasks_array = NULL ;
	struct Dc4cBatchTask *p_task = NULL ;
	int			tasks_count ;
	int			workers_count ;
	int			task_index ;
	int			repeat_task_flag ;

	char			begin_timebuf[ 256 + 1 ] ;
	char			end_timebuf[ 256 + 1 ] ;

	int			nret = 0 ;

	DC4CSetAppLogFile( "dc4c_test_multi_batch_master_stepbystep" );
	SetLogLevel( LOGLEVEL_DEBUG );

	if( argc >= 1 + 4 )
	{
		envs_count = atoi(argv[2]) ;
		a_penv = (struct Dc4cApiEnv**)malloc( sizeof(struct Dc4cApiEnv*) * envs_count ) ;
		if( a_penv == NULL )
		{
			printf( "malloc failed[%d] , errno[%d]\n" , nret , errno );
			return 1;
		}
		memset( a_penv , 0x00 , sizeof(struct Dc4cApiEnv*) * envs_count );

		for( envs_index = 0 ; envs_index < envs_count ; envs_index++ )
		{
			nret = DC4CInitEnv( & (a_penv[envs_index]) , argv[1] ) ;
			if( nret )
			{
```

```
        printf( "DC4CInitEnv failed[%d]\n" , nret );
        return 1;
}
else
{
        printf( "DC4CInitEnv ok , penv[%p]\n" , a_penv[envs_index] );
}

DC4CSetTimeout( a_penv[envs_index] , 15 );
DC4CSetOptions( a_penv[envs_index] , DC4C_OPTIONS_INTERRUPT_BY_APP );

workers_count = atoi(argv[3]) ;
tasks_count = atoi(argv[4]) ;

if( tasks_count < 0 )
{
        tasks_count = -tasks_count ;
        repeat_task_flag = 1 ;
}
else
{
        repeat_task_flag = 0 ;
}

if( workers_count < 0 )
{
        if( workers_count == -2 )
        {
                nret = DC4CQueryWorkers( penv ) ;
                if( nret )
                {
                        printf( "DC4CQueryWorkers failed[%d]\n" , nret );
                        return 1;
                }

                workers_count = DC4CGetUnusedWorkersCount( penv ) ;
                if( workers_count <= 0 )
                {
                        printf( "workers_count[%d] invalid\n" , workers_count );
                        return 1;
                }

                tasks_count = workers_count ;
        }
```

```
                        else
                        {
                                workers_count = tasks_count ;
                        }
                }

                tasks_array = (struct Dc4cBatchTask *)malloc( sizeof(struct Dc4cBatchTask) *
tasks_count ) ;
                if( tasks_array == NULL )
                {
                        printf( "alloc failed , errno[%d]\n" , errno );
                        return 1;
                }
                memset( tasks_array , 0x00 , sizeof(struct Dc4cBatchTask) * tasks_count );

                for( task_index = 0 ; task_index < tasks_count ; task_index++ )
                {
                        p_task = & (tasks_array[task_index]) ;

                        p_task->order_index = task_index ;
                        if( repeat_task_flag == 1 )
                                strcpy( p_task->program_and_params , argv[5] );
                        else
                                strcpy( p_task->program_and_params , argv[5+task_index] );
                        p_task->timeout = DC4CGetTimeout(a_penv[envs_index]) ;
                }

                nret = DC4CBeginBatchTasks( a_penv[envs_index] , workers_count , tasks_array ,
tasks_count ) ;
                free( tasks_array );
                if( nret )
                {
                        printf( "DC4CBeginBatchTasks failed[%d] , errno[%d]\n" , nret , errno );
                        return 1;
                }
        }

        while(1)
        {
                nret = DC4CPerformMultiBatchTasks( a_penv , envs_count , & penv , & task_index ) ;
                if( nret == DC4C_INFO_TASK_FINISHED )
                {
                        printf( "DC4CPerformMultiBatchTasks return DC4C_INFO_TASK_FINISHED ,
penv[%p]\n" , penv );
```

```
                        printf( "[%p][%d]-[%s][%ld]-[%s][%s][%d][%s][%s][%d]-[%d][%d][%s]\n"
                                , penv , task_index , DC4CGetBatchTasksIp(penv,task_index) ,
DC4CGetBatchTasksPort(penv,task_index)
                                , DC4CGetBatchTasksTid(penv,task_index) ,
DC4CGetBatchTasksProgramAndParams(penv,task_index) , DC4CGetBatchTasksTimeout(penv,task_index) ,
ConvertTimeString(DC4CGetBatchTasksBeginTimestamp(penv,task_index),begin_timebuf,sizeof(begin_time
buf))+11 ,
ConvertTimeString(DC4CGetBatchTasksEndTimestamp(penv,task_index),end_timebuf,sizeof(end_timebuf))+
11 , DC4CGetBatchTasksElapse(penv,task_index)
                                , DC4CGetBatchTasksError(penv,task_index) ,
WEXITSTATUS(DC4CGetBatchTasksStatus(penv,task_index)) , DC4CGetBatchTasksInfo(penv,task_index) );
                }
                else if( nret == DC4C_INFO_BATCH_TASKS_FINISHED )
                {
                        printf( "DC4CPerformMultiBatchTasks return DC4C_INFO_BATCH_TASKS_FINISHED ,
penv[%p]\n" , penv );
                }
                else if( nret == DC4C_INFO_ALL_ENVS_FINISHED )
                {
                        printf( "DC4CPerformMultiBatchTasks return DC4C_INFO_ALL_ENVS_FINISHED ,
penv[%p]\n" , penv );
                        break;
                }
                else if( nret == DC4C_ERROR_TIMEOUT )
                {
                        printf( "DC4CPerformMultiBatchTasks return DC4C_ERROR_TIMEOUT\n" );
                }
                else if( nret == DC4C_ERROR_APP )
                {
                        printf( "DC4CPerformMultiBatchTasks return DC4C_ERROR_APP\n" );
                }
                else
                {
                        printf( "DC4CPerformMultiBatchTasks failed[%d] , penv[%p]\n" , nret , penv );
                        break;
                }

                if( nret == DC4C_ERROR_APP )
                        DC4CResetFinishedTasksWithError( penv );
        }

        for( envs_index = 0 ; envs_index < envs_count ; envs_index++ )
        {
                DC4CCleanEnv( & (a_penv[envs_index]) );
```

```
        }
        free( a_penv );
        printf( "DC4CCleanEnv ok\n" );
    }
    else
    {
        printf( "USAGE : dc4c_test_multi_batch_master rserver_ip:rserver_port,... envs_count
workers_count task_count program_and_params_1 ...\n" );
        exit(7);
    }

    return 0;
}
```

4.4.6 DAG 调度多批量任务分派（从配置文件载入配置）（高层函数）

配置文件 test/dc4c_test_tfc_dag_master.c

```
/*
                                    _BEGIN_
                                /                       \
                        BATCH_A1                            BATCH_B1
                "dc4c_test_worker_sleep 1"          "dc4c_test_worker_sleep 3"
                    /               \                       |
            BATCH_A21               BATCH_A22                   BATCH_B2
  "dc4c_test_worker_sleep 4" "dc4c_test_worker_sleep 5"   "dc4c_test_worker_sleep 6"
                    \               /                       |
                        BATCH_A3                            /
                "dc4c_test_worker_sleep 2"                /
                "dc4c_test_worker_sleep 1"              /
                                    \               /
                                    _END_
*/
{
    "schedule" : {
            "schedule_name" : "test_schedule_name" ,
            "schedule_desc" : "test_schedule_desc"
    } ,
    "batches" : {
            "batches_info" : [
                    { "batch_name":"BATCH_A1", "batch_desc":"", "view_pos_x":0, "view_pos_y":0,
```

```
"tasks": [
                        { "program_and_params":"dc4c_test_worker_sleep 1", "timeout":60" }
                ] },
                { "batch_name":"BATCH_A21", "batch_desc":"", "view_pos_x":0, "view_pos_y":0,
"tasks": [
                ] },
                { "batch_name":"BATCH_A22", "batch_desc":"", "view_pos_x":0, "view_pos_y":0,
"tasks": [
                        { "program_and_params":"dc4c_test_worker_sleep 5", "timeout":60" }
                ] },
                { "batch_name":"BATCH_A3", "batch_desc":"", "view_pos_x":0, "view_pos_y":0,
"tasks": [
                        { "program_and_params":"dc4c_test_worker_sleep 3", "timeout":60" } ,
                        { "program_and_params":"dc4c_test_worker_sleep 2", "timeout":60" } ,
                        { "program_and_params":"dc4c_test_worker_sleep 1", "timeout":60" }
                ] },
                { "batch_name":"BATCH_B1", "batch_desc":"", "view_pos_x":0, "view_pos_y":0,
"tasks": [
                        { "program_and_params":"dc4c_test_worker_sleep 3", "timeout":60" }
                ] },
                { "batch_name":"BATCH_B2", "batch_desc":"", "view_pos_x":0, "view_pos_y":0,
"tasks": [
                        { "program_and_params":"dc4c_test_worker_sleep 6", "timeout":60" }
                ] }
        ],
        "batches_direction" : [
                { "from_batch":"", "to_batch":"BATCH_A1" } ,
                { "from_batch":"", "to_batch":"BATCH_B1" } ,
                { "from_batch":"BATCH_A1", "to_batch":"BATCH_A21" } ,
                { "from_batch":"BATCH_A1", "to_batch":"BATCH_A22" } ,
                { "from_batch":"BATCH_A21", "to_batch":"BATCH_A3" } ,
                { "from_batch":"BATCH_A22", "to_batch":"BATCH_A3" } ,
                { "from_batch":"BATCH_B1", "to_batch":"BATCH_B2" } ,
                { "from_batch":"BATCH_A3", "to_batch":"" } ,
                { "from_batch":"BATCH_B2", "to_batch":"" }
        ]
    }
}
```

源代码

```
#include "dc4c_api.h"
#include "dc4c_tfc_dag.h"

/* for testing
time ./dc4c_test_tfc_dag_master rservers_ip_port *.dag_schedule"
```

```
*/

static int TestDagSchedule( char *dag_schedule_pathfilename , char *rservers_ip_port )
{
	struct Dc4cDagSchedule	*p_sched = NULL ;

	int			nret = 0 ;

	nret = DC4CLoadDagScheduleFromFile( & p_sched , dag_schedule_pathfilename , rservers_ip_port ,
DC4C_OPTIONS_INTERRUPT_BY_APP ) ;
	if( nret )
	{
		printf( "DC4CLoadDagScheduleFromFile failed[%d]\n" , nret );
		return -1;
	}
	else
	{
		printf( "DC4CLoadDagScheduleFromFile ok\n" );
	}

	DC4CLogDagSchedule( p_sched );

	nret = DC4CExecuteDagSchedule( p_sched ) ;
	if( nret )
	{
		printf( "DC4CExecuteDagSchedule failed[%d]\n" , nret );
		return -1;
	}
	else
	{
		printf( "DC4CExecuteDagSchedule ok\n" );
	}

	DC4CUnloadDagSchedule( & p_sched );
	printf( "DC4CUnloadDagSchedule ok\n" );

	return 0;
}

int main( int argc , char *argv[] )
{
	DC4CSetAppLogFile( "dc4c_test_tfc_dag_master" );
	SetLogLevel( LOGLEVEL_DEBUG );
```

```
	if( argc == 1 + 2 )
	{
		return TestDagSchedule( argv[2] , argv[1] );
	}
	else
	{
		printf( "USAGE : dc4c_test_tfc_dag_master rservers_ip_port .dag_schedule\n" );
		exit(7);
	}
}
```

4.4.7 DAG 调度多批量任务分派（从配置文件载入配置）（低层函数）

源代码

```
#include "dc4c_api.h"
#include "dc4c_tfc_dag.h"

/* for testing
time ./dc4c_test_tfc_dag_master rservers_ip_port *.dag_schedule"
*/

static int TestDagSchedule( char *dag_schedule_pathfilename , char *rservers_ip_port )
{
	struct Dc4cDagSchedule	*p_sched = NULL ;
	struct Dc4cDagBatch	*p_batch = NULL ;
	struct Dc4cApiEnv	*penv = NULL ;
	int			task_index ;

	int			perform_return = 0 ;
	int			nret = 0 ;

	nret = DC4CLoadDagScheduleFromFile( & p_sched , dag_schedule_pathfilename , rservers_ip_port ,
DC4C_OPTIONS_INTERRUPT_BY_APP ) ;
	if( nret )
	{
		printf( "DC4CLoadDagScheduleFromFile failed[%d]\n" , nret );
		return -1;
	}
	else
	{
```

```
		printf( "DC4CLoadDagScheduleFromFile ok\n" );
	}

	DC4CLogDagSchedule( p_sched );

	nret = DC4CBeginDagSchedule( p_sched ) ;
	if( nret )
		return nret;

	while(1)
	{
		perform_return = DC4CPerformDagSchedule( p_sched , & p_batch , & penv , & task_index ) ;
		if( perform_return == DC4C_INFO_TASK_FINISHED )
		{
			printf( "DC4CPerformDagSchedule return DC4C_INFO_TASK_FINISHED , batch_name[%s]
task_index[%d]\n" , DC4CGetDagBatchName(p_batch) , task_index );
		}
		else if( perform_return == DC4C_INFO_BATCH_TASKS_FINISHED )
		{
			printf( "DC4CPerformDagSchedule return DC4C_INFO_BATCH_TASKS_FINISHED ,
batch_name[%s]\n" , DC4CGetDagBatchName(p_batch) );
		}
		else if( perform_return == DC4C_INFO_ALL_ENVS_FINISHED )
		{
			printf( "DC4CPerformDagSchedule return DC4C_INFO_ALL_ENVS_FINISHED\n" );
			break;
		}
		else if( perform_return == DC4C_ERROR_TIMEOUT )
		{
			printf( "DC4CPerformDagSchedule return DC4C_ERROR_TIMEOUT\n" );
			break;
		}
		else if( perform_return == DC4C_ERROR_APP )
		{
			printf( "DC4CPerformDagSchedule failed[%d] , batch_name[%s]\n" , perform_return ,
DC4CGetDagBatchName(p_batch) );
			break;
		}
		else
		{
			printf( "DC4CPerformDagSchedule failed[%d]\n" , nret );
			break;
		}
	}
```

```
	DC4CUnloadDagSchedule( & p_sched );
	printf( "DC4CUnloadDagSchedule ok\n" );

	return 0;
}

int main( int argc , char *argv[] )
{
	DC4CSetAppLogFile( "dc4c_test_tfc_dag_master" );
	SetLogLevel( LOGLEVEL_DEBUG );

	if( argc == 1 + 2 )
	{
		return TestDagSchedule( argv[2] , argv[1] );
	}
	else
	{
		printf( "USAGE : dc4c_test_tfc_dag_master rservers_ip_port .dag_schedule\n" );
		exit(7);
	}
}
```

4.4.8 DAG 调度多批量任务分派（从数据载入配置）（低层函数）

数据库配置

```
truncate table dag_schedule ;
truncate table dag_batches_info ;
truncate table dag_batches_tasks ;
truncate table dag_batches_direction ;

INSERT INTO dag_schedule VALUES ( 1 , 'TEST_SCHEDULE_NAME' , 'TEST_SCHEDULE_DESC' , '' , '' , 0 );

INSERT INTO dag_batches_info VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_A1' , 'BATCH_A1_DESC' , 0 , 0 , '' ,
'' , 0 );
INSERT INTO dag_batches_info VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_B1' , 'BATCH_B1_DESC' , 0 , 0 , '' ,
'' , 0 );
INSERT INTO dag_batches_info VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_A21' , 'BATCH_A21_DESC' , 0 , 0 ,
'' , '' , 0 );
INSERT INTO dag_batches_info VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_A22' , 'BATCH_A22_DESC' , 0 , 0 ,
```

```
'' , '' , 0 );
INSERT INTO dag_batches_info VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_A3' , 'BATCH_A3_DESC' , 0 , 0 , '' ,
'' , 0 );
INSERT INTO dag_batches_info VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_B2' , 'BATCH_B2_DESC' , 0 , 0 , '' ,
'' , 0 );

INSERT INTO dag_batches_tasks VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_A1' , 1 ,
'dc4c_test_worker_sleep 1' , 60 , '' , '' , 0 , 0 , 0 );
INSERT INTO dag_batches_tasks VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_A22' , 1 ,
'dc4c_test_worker_sleep 5' , 60 , '' , '' , 0 , 0 , 0 );
INSERT INTO dag_batches_tasks VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_A3' , 1 ,
'dc4c_test_worker_sleep 1' , 60 , '' , '' , 0 , 0 , 0 );
INSERT INTO dag_batches_tasks VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_A3' , 2 ,
'dc4c_test_worker_sleep 1' , 60 , '' , '' , 0 , 0 , 0 );
INSERT INTO dag_batches_tasks VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_A3' , 3 ,
'dc4c_test_worker_sleep 1' , 60 , '' , '' , 0 , 0 , 0 );
INSERT INTO dag_batches_tasks VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_B1' , 1 ,
'dc4c_test_worker_sleep 3' , 60 , '' , '' , 0 , 0 , 0 );
INSERT INTO dag_batches_tasks VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_B2' , 1 ,
'dc4c_test_worker_sleep 6' , 60 , '' , '' , 0 , 0 , 0 );

INSERT INTO dag_batches_direction VALUES ( 'TEST_SCHEDULE_NAME' , '' , 'BATCH_A1' );
INSERT INTO dag_batches_direction VALUES ( 'TEST_SCHEDULE_NAME' , '' , 'BATCH_B1' );
INSERT INTO dag_batches_direction VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_A1' , 'BATCH_A21' );
INSERT INTO dag_batches_direction VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_A1' , 'BATCH_A22' );
INSERT INTO dag_batches_direction VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_A21' , 'BATCH_A3' );
INSERT INTO dag_batches_direction VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_A22' , 'BATCH_A3' );
INSERT INTO dag_batches_direction VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_B1' , 'BATCH_B2' );
INSERT INTO dag_batches_direction VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_A3' , '' );
INSERT INTO dag_batches_direction VALUES ( 'TEST_SCHEDULE_NAME' , 'BATCH_B2' , '' );
```

源代码 test/dc4c_test_tfc_dag_master_pgsql.c

```
static int TestDagSchedule( char *schedule_name , char *rservers_ip_port )
{
	struct Dc4cDagSchedule	*p_sched = NULL ;
	struct Dc4cDagBatch	*p_batch = NULL ;
	struct Dc4cApiEnv	*penv	= NULL ;
	int			task_index ;

	int			perform_return = 0 ;
	int			nret = 0 ;

	nret = DC4CLoadDagScheduleFromDatabase( & p_sched , schedule_name , rservers_ip_port ,
DC4C_OPTIONS_INTERRUPT_BY_APP ) ;
	if( nret )
```

```
	{
		printf( "DC4CLoadDagScheduleFromDatabase failed[%d]\n" , nret );
		return -1;
	}
	else
	{
		printf( "DC4CLoadDagScheduleFromDatabase ok\n" );
	}

	if( p_sched == NULL )
	{
		printf( "All is done\n" );
		return 0;
	}

	DC4CLogDagSchedule( p_sched );

	nret = DC4CBeginDagSchedule( p_sched ) ;
	if( nret )
	{
		printf( "DC4CBeginDagSchedule failed[%d]\n" , nret );
		return -1;
	}
	else
	{
		printf( "DC4CBeginDagSchedule ok\n" );
	}

	while(1)
	{
		perform_return = DC4CPerformDagSchedule( p_sched , & p_batch , & penv , & task_index ) ;
		if( perform_return == DC4C_INFO_TASK_FINISHED )
		{
			printf( "DC4CPerformDagSchedule return DC4C_INFO_TASK_FINISHED , batch_name[%s]
task_index[%d]\n" , DC4CGetDagBatchName(p_batch) , task_index );
			continue;
		}
		else if( perform_return == DC4C_INFO_BATCH_TASKS_FINISHED )
		{
			printf( "DC4CPerformDagSchedule return DC4C_INFO_BATCH_TASKS_FINISHED ,
batch_name[%s]\n" , DC4CGetDagBatchName(p_batch) );
		}
		else if( perform_return == DC4C_INFO_ALL_ENVS_FINISHED )
		{
```

```
                printf( "DC4CPerformDagSchedule return DC4C_INFO_ALL_ENVS_FINISHED\n" );
                break;
        }
        else if( perform_return == DC4C_ERROR_TIMEOUT )
        {
                printf( "DC4CPerformDagSchedule return DC4C_ERROR_TIMEOUT\n" );
                break;
        }
        else if( perform_return == DC4C_ERROR_APP )
        {
                printf( "DC4CPerformDagSchedule return DC4C_ERROR_APP\n" );
                break;
        }
        else
        {
                printf( "DC4CPerformDagSchedule failed[%d] , batch_name[%s]\n" , perform_return ,
DC4CGetDagBatchName(p_batch) );
                break;
        }

        nret = DC4CUpdateBatchTasks( p_sched , p_batch , penv );
        if( nret )
        {
                printf( "DC4CUpdateBatchTasks failed[%d]\n" , nret );
                break;
        }
        else
        {
                printf( "DC4CUpdateBatchTasks ok\n" );
        }
    }

    nret = DC4CUnloadDagScheduleToDatabase( & p_sched ) ;
    if( nret )
    {
        printf( "DC4CUnloadDagScheduleToDatabase failed[%d]\n" , nret );
        return -1;
    }
    else
    {
        printf( "DC4CUnloadDagScheduleToDatabase ok\n" );
    }

    return 0;
```

```
}

int main( int argc , char *argv[] )
{
	int		nret = 0 ;

	DC4CSetAppLogFile( "dc4c_test_tfc_dag_master_pgsql" );
	SetLogLevel( LOGLEVEL_DEBUG );

	if( argc == 1 + 2 )
	{
		DSCDBCONN( getenv("DBHOST") , atoi(getenv("DBPORT")) , getenv("DBNAME") ,
getenv("DBUSER") , getenv("DBPASS") );
		if( SQLCODE )
		{
			printf( "DSCDBCONN failed , SQLCODE[%d][%s][%s]\n" , SQLCODE , SQLSTATE , SQLDESC );
			return 1;
		}
		else
		{
			printf( "DSCDBCONN ok\n" );
		}

		nret = TestDagSchedule( argv[2] , argv[1] ) ;

		DSCDBDISCONN();
		printf( "DSCDBDISCONN ok\n" );

		return -nret;
	}
	else
	{
		printf( "USAGE : dc4c_test_tfc_dag_master rservers_ip_port .dag_schedule\n" );
		exit(7);
	}
}
```

5 内部实现

5.1 基础平台架构

5.1.1 接口协议

5.1.1.1通讯协议

通讯方式采用原生 TCP。

|通讯头(8bytes)|通讯体(nbytes)|

通讯协议分通讯头和通讯体两段。

通讯头长 8 个字符，为通讯体长度值的数字字符。

5.1.1.2报文协议

通讯体内为完整报文。

报文分报文头和报文体两段。

| 通讯体(nbytes) |

|报文头(8bytes)|报文体(mbytes)|

报文头长 8 个字符，前三个字符为报文类型，后五个字符备用，如报文类型“TTQ”，报文体“{“json_key”:”json_value”}”，完整通讯数据为

“00000036TTQ {“json_key”:”json_value”}”

报文协议定义文件在 src/proto/*.dsc

注册请求

计算节点启动后向注册节点注册自己的请求

IDL_worker_register_request.dsc

```
STRUCT  worker_register_request
{
        STRING  32      sysname
        STRING  128     release
        INT     4       bits

        STRING  40      ip
        INT     4       port
}
```

注册响应

计算节点启动后向注册节点注册自己的响应

IDL_worker_register_response.dsc

```
STRUCT  worker_register_response
{
        INT     4       error
}
```

状态变更通知

计算节点随时发送自己的状态同步给注册节点

IDL_worker_notice_request.dsc

```
STRUCT  worker_notice_request
{
        STRING  64      sysname
        STRING  64      release
        INT     4       bits

        STRING  40      ip
        INT     4       port

        INT     4       is_working
}
```

查询计算节点请求

用户节点向注册节点查询当前空闲的计算节点信息的请求

IDL_query_workers_request.dsc

```
STRUCT  query_workers_request
{
      STRING  64      sysname
      STRING  64      release
      INT     4       bits

      INT     4       count
}
```

查询计算节点响应

用户节点向注册节点查询当前空闲的计算节点信息的响应

IDL_query_workers_response.dsc

```
STRUCT  query_workers_response
{
      INT     4       error
      STRUCT  nodes   ARRAY   1000
      {
            STRUCT  node
            {
                  STRING  40      ip
                  INT     4       port
            }
      }
}
```

执行任务请求

用户节点向计算节点发起执行任务的请求

IDL_execute_program_request.dsc

```
STRUCT  execute_program_request
{
```

```
        STRING  40      ip
        INT     4       port
        STRING  20      tid
        INT     4       order_index
        STRING  256     program_and_params
        STRING  32      program_md5_exp
        INT     4       timeout
        INT     4       begin_datetime_stamp
        INT     1       bind_cpu_flag
}
```

执行任务响应

用户节点向计算节点发起执行任务的响应

IDL_execute_program_response.dsc

```
STRUCT  execute_program_response
{
        STRING  20      tid
        INT     4       end_datetime_stamp
        INT     4       elapse
        INT     4       error
        INT     4       status
        STRING  1024    info
}
```

部署程序请求

当计算节点收到用户节点的执行任务请求后，校验程序 MD5 不一致时，计算节点向用户节点发起部署程序请求

IDL_deploy_program_request.dsc

```
STRUCT  deploy_program_request
{
        STRING  64      program
}
```

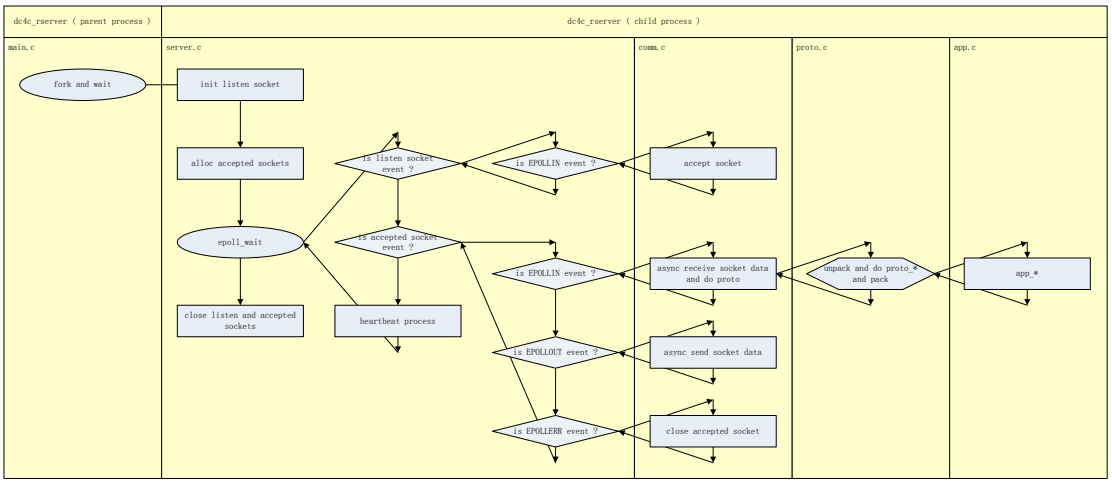
（部署程序响应报文用可执行程序裸数据覆盖通讯体）

5.1.1.3其它协议

注册节点与计算节点和用户节点之间都是用以上接口协议交换数据，除此以外，直接 socket 连接注册节点（如 telnet）查询、管理注册节点内部信息则使用行协议。

5.1.2 架构与模块

5.1.2.1注册节点



注册节点程序 dc4c_rserver 启动后(main.c)创建子进程，并监控子进程异常。

通讯层管理层(server.c)初始化侦听端口，进入 epoll_wait 事件循环，每个事件处理函数由通讯层(comm.c)负责。

如果是侦听事件

如果是输入事件

接受新连接

如果是已接受连接事件

如果是输入事件

非堵塞接收数据，如果收完整报文，交由报文转换层处理

如果是输出事件

非堵塞发送数据，如果发送完报文，转而等待输入事件

如果是错误事件

关闭通讯连接

周期性发送心跳报文，检查心跳超时连接

报文转换层(proto.c)入口函数根据报文类型分支拆解报文，调用应用层(app.c)处理应用逻辑，然后返回上层。

主要应用层函数：

app_WorkerRegisterRequest 处理从计算节点发来的注册请求，并填充注册响应

app_WorkerNoticeRequest 处理从计算节点发来的状态变更通知

app_WorkerUnregister 当侦测到计算节点断开连接时，同步删除内部维护的计算节点信息列表

app_QueryWorkersRequest 处理从用户节点发来的查询空闲计算节点请求，内含算法使得返回的计算节点信息尽量集群均分

app_QueryAllOsTypes 处理以行协议接收到的查询所有主机类型列表请求

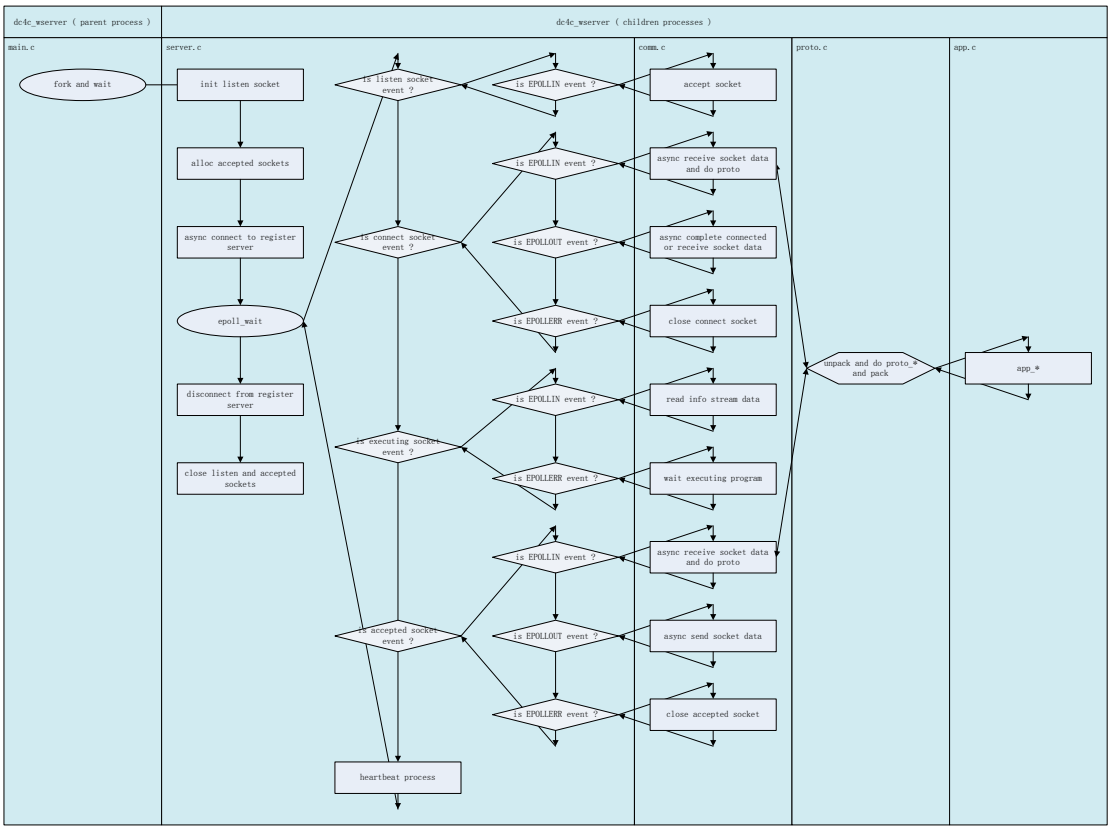
app_QueryAllHosts 处理以行协议接收到的查询所有主机列表请求

app_QueryAllWorkers 处理以行协议接收到的查询所有计算节点列表请求

app_HeartBeatRequest 定时向计算节点发送心跳报文

app_HeartBeatResponse 处理从计算节点发来的心跳报文

5.1.2.2计算节点



计算节点程序 dc4c_wserver 启动后(main.c)创建子进程组（计算节点组），并监控子进程异常。

通讯层管理层(server.c)初始化侦听端口，非堵塞连接注册节点，进入 epoll_wait 事件循环，每个事件处理函数由通讯层(comm.c)负责。

如果是侦听事件

如果是输入事件

接受新连接

如果是连接注册节点会话事件

如果是输入事件

非堵塞接收数据，如果收完整报文，交由报文转换层处理

如果是输出事件

如果未连接

测试非堵塞连接注册节点是否成功

否则

非堵塞发送数据，如果发送完报文，转而等待输入事件

如果是错误事件

关闭通讯连接

如果是孙子进程之间的管道事件

如果是输入事件

非堵塞接收管道数据，保存到执行任务响应报文中

如果是用户节点连接会话事件

如果是输入事件

非堵塞接收数据，如果收完整报文，交由报文转换层处理

如果是输出事件

非堵塞发送数据，如果发送完报文，转而等待输入事件

如果是错误事件

关闭通讯连接

周期性发送心跳报文，检查心跳超时连接

报文转换层(proto.c)入口函数根据报文类型分支拆解报文，调用应用层(app.c)处理应用逻辑，然后返回上层。

主要应用层函数：

app_WorkerRegisterResponse 处理从注册节点接收到的注册响应

app_SendWorkerNotice 组织状态变更请求报文

app_ExecuteProgram 创建孙子进程，执行任务命令行

app_WaitProgramExiting 回收孙子进程

app_ExecuteProgramRequest 处理从用户节点发来的的执行任务请求

app_DeployProgramResponse 处理从用户节点发来的部署程序响应

app_HeartBeatRequest 定时向注册节点发送心跳报文

app_HeartBeatResponse 处理从注册节点发来的心跳报文

安装包内自带了测试程序也用于演示计算节点用户程序构成和 API 用法：

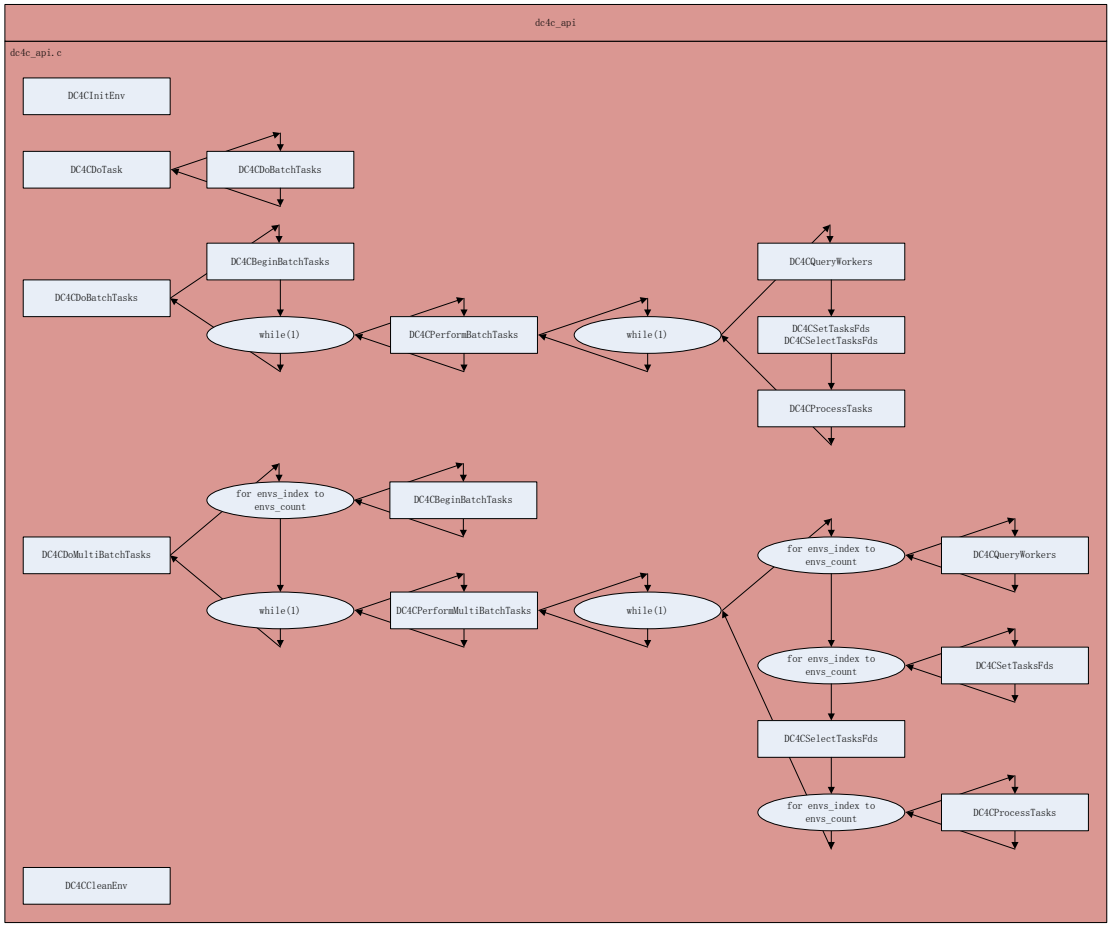
dc4c_test_worker_hello.c 反射 hello

dc4c_test_worker_sleep.c 沉睡一段时间，用于模拟任务耗时

dc4c_test_worker_sleep_or_error.c 当沉睡秒数为 0 时，报错返回 1，用于模拟任务出错

dc4c_test_worker_pi.c 用于计算圆周率，内含用户节点内容和计算节点内容

5.1.2.3用户节点



用户程序调用用户节点 API 实现（批量）任务分派和反馈。

结构 struct Dc4cApiEnv 为（批量）任务环境，在分派（批量）任务前必须构造，在结束使用前必须清理。

单任务分派函数 DC4CDoTask 调用**批量任务分派函数 DC4CDoBatchTasks** 来实现。

批量任务分派函数 DC4CDoBatchTasks 内部先调用函数 DC4CBeginBatchTasks 初始化任务环境、导入任务到执行任务请求报文、预连接注册节点，然后循环调用函数 DC4CPerformBatchTasks 直至所有任务结束。

函数 DC4CPerformBatchTasks 内部循环调用函数 DC4CQueryWorkers 向注册节点查询空闲计算节点、函数 DC4CSetTasksFds 记录已连接会话描述字到描述字集合、函数 DC4CSelectTasksFds 侦测所有描述字事件、函数 DC4CProcessTasks 处理就绪描述字事件。如果出现超时或计算节点用户程序返回错误（允许错误中断选项打开）设置中断标志，等待正在执行的任务都结束后报错返回。

函数 DC4CSetTasksFds 还会根据未完成任务和未取用空闲计算节点信息，分派新的任务，并记录描述字到描述字集合。

函数 DC4CProcessTasks 接收处理执行任务响应、部署程序请求以及错误，此外还处理超时连接。

多批量分派函数 DC4CDoMultiBatchTasks 内部先调用函数 DC4CBeginBatchTasks 初始化任务环境集合，然后循环调用函数 DC4CPerformMultiBatchTasks 直至所有批量任务结束。

函数 DC4CPerformMultiBatchTasks 内部循环调用函数 DC4CQueryWorkers 取得所有（批量）任务环境的查询空闲计算节点信息、函数 DC4CSetTasksFds 记录所有（批量）任务环境的已连接会话描述字到描述字集合、函数 DC4CSelectTasksFds 侦测所有描述字事件、函数 DC4CProcessTasks 处理所有（批量）任务环境的就绪描述字事件。如果出现超时或计算节点用户程序返回错误（允许错误中断选项打开）设置中断标志，等待正在执行的（批量）任务都结束后报错返回。

安装包内自带了测试程序也用于演示用户节点用户程序构成和 API 用法：

dc4c_test_master.c 单任务分派 API 调用

dc4c_test_batch_master.c 批量任务分派高层 API 调用

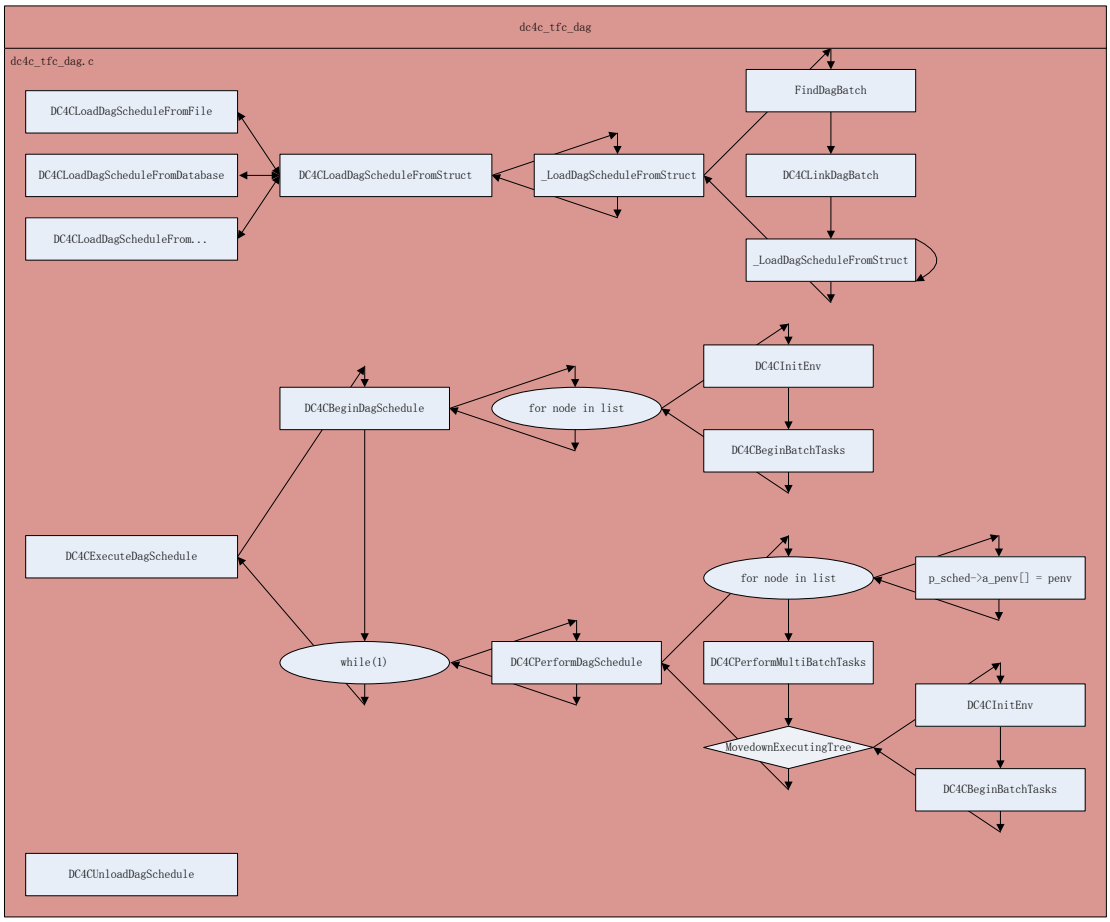
dc4c_test_batch_master_stepbystep.c 批量任务分派低层 API 调用

dc4c_test_multi_batch_master.c 多批量任务分派高层 API 调用

dc4c_test_multi_batch_master_stepbystep.c 多批量任务分派低层 API 调用

5.2 任务调度引擎

5.2.1 DAG 任务调度引擎



dc4c_tfc_dag 调度环境结构为 struct Dc4cDagSchedule，每个调度块结构为 struct Dc4cDagBatch。

调度环境结构从外部载入计划配置，如外部配置文件、数据库等。函数 DC4CLoadDagScheduleFromFile 提供了从外部 json 配置文件载入计划配置，test/dc4c_test_tfc_dag_master_pgsql_stepbystep.ec 演示了通过编写自定义函数 DC4CLoadDagScheduleFromDatabase 从数据库载入计划配置。载入函数内部主要是根据外部数据填充结构 dag_schedule_configfile，然后调用低层函数 DC4CLoadDagScheduleFromStruct 构造调度环境结构。构造调度环境结构主要是

创建了基于有向无环图数据结构的调度块树形引用链表，但是每个调度块实体信息结构被串连在一个线性链表中，实体信息结构主要包含了一个批量任务环境、一个前置调度块集合线性链表、一个后续调度块集合线性链表。

执行任务调度引擎函数 DC4CExecuteDagSchedule 首先调用函数 DC4CBeginDagSchedule 开始初始调度块。调度环境结构内部维护着一个线性的执行调度块集合链表，用于跟踪当前正在处理的调度块集合，如果一个调度块执行结束则判断它的每一个后续调度块的所有前置调度块集合是否都已经结束，如果都结束，该后续调度块则串连进执行调度块集合链表，当前结束调度块从执行调度块集合链表中删除，整个判断过程被封装成内部函数 MovedownExecutingTree。初始时执行调度块集合链表只串连了开始调度虚块”[BEGIN]”，当最后只处于结束调度虚块”[END]”时，整个任务调度过程结束。如果中间出现错误，且设置了允许中断选项，调度引擎会等待执行调度块集合链表中的所有正在执行的任务都完成后再报错返回。

也可以自行调用低层函数 DC4CBeginDagSchedule 和 DC4CPerformDagSchedule 来实现随时获取调度引擎内部事件，如某一任务结束、某一批量任务结束等。

调度结束后可以马上调用函数 DC4CGetDagSchedule*和 DC4CGetDagBatch*查询内部状态，如中断后查询哪个任务失败了。

全部完成后必须调用函数 DC4CUnloadDagSchedule 清理释放调度环境结构。

安装包内自带了测试程序也用于演示用户节点用户程序构成和 API 用法：

dc4c_test_tfc_dag_master.c 从外部 json 读入计划配置的 DAG 任务调度引擎高层 API 调用

dc4c_test_tfc_dag_master_stepbystep.c 从外部 json 文件（测试用 dc4c_test*.dag_schedule）读入计划配置的 DAG 任务调度引擎低层 API 调用

dc4c_test_tfc_dag_master_pgsql_stepbystep.ec 从数据库（表 dag_schedule、dag_batches_info、dag_batches_direction、dag_batches_tasks）读入计划配置的 DAG 任务调度引擎低层 API 调用

6 应用案例

6.1 计算圆周率

圆周率可利用泰勒公式展开计算:

$$\pi = 4 * (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots)$$

= **3.14159265**358979323846264338327 95...

编写计算节点用户程序，选定展开式分母最大值，分割区间并发计算，最后再合并结果

```
[idd@WebCrawler /home/idd/exsrc/dc4c/test] time ./dc4c_test_worker_pi 1 1000000000

real    0m55.590s
user    0m55.593s
sys     0m0.003s
[idd@WebCrawler /home/idd/log] cat dc4c_test_worker_pi.log
...
2015-05-20 18:32:39.950828 | INFO   | 17086:0:dc4c_test_worker_pi.c:172 | pi_worker() - start_x[1]
end_x[1000000000] - PI[3.141592651576240040 34]
...

[idd@WebCrawler /home/idd/exsrc/dc4c/test] time ./dc4c_test_master 0:12001 "dc4c_test_worker_pi
0:12001 1000000000 4"

real    0m14.323s
user    0m0.002s
sys     0m0.001s
[dc4c@WebCrawler /home/dc4c/log] cat dc4c_wserver_2_0.0.0.0:13002.dc4c_test_worker_pi.log
...
2015-05-20 18:28:46.172764 | INFO   | 17036:0:dc4c_test_worker_pi.c:75 | DC4CDoBatchTasks ok
2015-05-20 18:28:46.172788 | INFO   | 17036:0:dc4c_test_worker_pi.c:92 |
Task[1]-[0.0.0.0][13004]-[1432117711000085654][dc4c_test_worker_pi 1
250000000][600][15]-[0][0][3.141592661586404707 48]
2015-05-20 18:28:46.172818 | INFO   | 17036:0:dc4c_test_worker_pi.c:92 |
Task[2]-[0.0.0.0][13003]-[1432117711000085694][dc4c_test_worker_pi 250000002
```

```
500000001][600][14]-[0][0][-0.000000003999999952000000336]
2015-05-20 18:28:46.172833 | INFO   | 17036:0:dc4c_test_worker_pi.c:92 |
Task[3]-[0.0.0.0][13001]-[143211771000085851][dc4c_test_worker_pi 500000003
750000002][600][15]-[0][0][-0.000000006666666636444444456356]
2015-05-20 18:28:46.172844 | INFO   | 17036:0:dc4c_test_worker_pi.c:92 |
Task[4]-[0.0.0.0][13005]-[143211771000085997][dc4c_test_worker_pi 750000004
1000000003][600][15]-[0][0][0.000000000666666666044444448555
6]
2015-05-20 18:28:46.172868 | INFO   | 17036:0:dc4c_test_worker_pi.c:100 | pi_master() -
max_x[1000000003] tasks_count[4] - PI[3.141592651586404779 43]
...
```

源代码 test/dc4c_test_worker_pi.c

```
#include "dc4c_api.h"

#include "gmp.h"

int pi_master( char *rservers_ip_port , unsigned long max_x , int tasks_count )
{
	struct Dc4cApiEnv	*penv = NULL ;
	struct Dc4cBatchTask *tasks_array = NULL ;
	int			workers_count ;
	unsigned long		dd_x ;
	unsigned long		start_x , end_x ;

	int			i ;
	struct Dc4cBatchTask *p_task = NULL ;
	char			*ip = NULL ;
	long			port ;
	char			*tid = NULL ;
	char			*program_and_params = NULL ;
	int			timeout ;
	int			elapse ;
	int			error ;
	int			status ;
	char			*info = NULL ;

	mpf_t			pi_incr ;
```

```
mpf_t                   pi ;

char                output[ 1024 + 1 ] ;

int                 nret = 0 ;

DC4CSetAppLogFile( "pi_master" );
SetLogLevel( LOGLEVEL_DEBUG );

InfoLog( __FILE__ , __LINE__ , "pi_master" );

nret = DC4CInitEnv( & penv , rservers_ip_port ) ;
if( nret )
{
        ErrorLog( __FILE__ , __LINE__ , "DC4CInitEnv failed[%d]" , nret );
        return -1;
}
else
{
        InfoLog( __FILE__ , __LINE__ , "DC4CInitEnv ok" );
}

DC4CSetTimeout( penv , 600 );
DC4CSetOptions( penv , DC4C_OPTIONS_BIND_CPU );

if( tasks_count == -2 )
{
        nret = DC4CQueryWorkers( penv ) ;
        if( nret )
        {
                ErrorLog( __FILE__ , __LINE__ , "DC4CQueryWorkers failed[%d]" , nret );
                return -1;
        }

        tasks_count = DC4CGetUnusedWorkersCount( penv ) ;
        if( tasks_count <= 0 )
        {
                ErrorLog( __FILE__ , __LINE__ , "tasks_count[%d] invalid" , tasks_count );
                return -1;
        }
}

tasks_array = (struct Dc4cBatchTask *)malloc( sizeof(struct Dc4cBatchTask) * tasks_count ) ;
if( tasks_array == NULL )
```

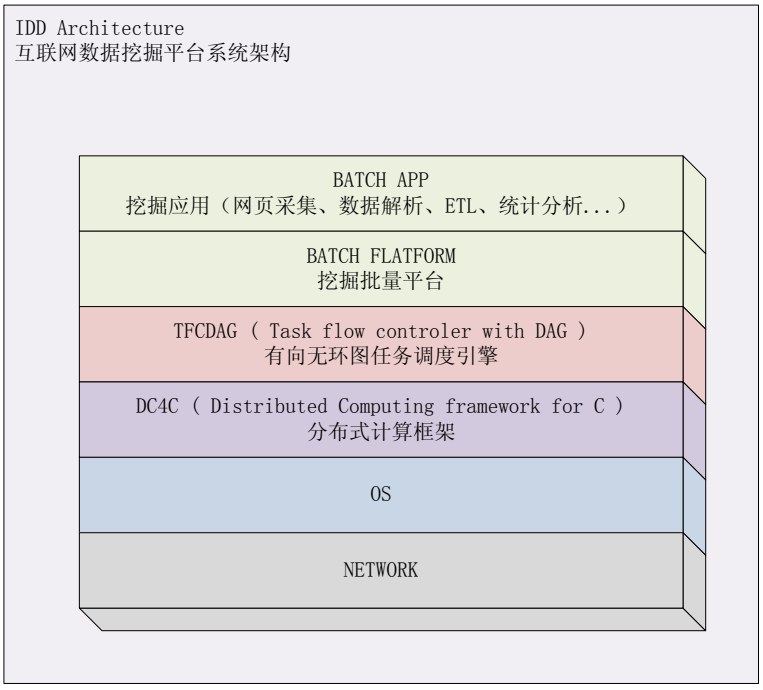
```
	{
		ErrorLog( __FILE__ , __LINE__ , "alloc failed , errno[%d]" , errno );
		return -1;
	}
	memset( tasks_array , 0x00 , sizeof(struct Dc4cBatchTask) * tasks_count );

	dd_x = ( max_x - tasks_count ) / tasks_count ;
	start_x = 1 ;
	for( i = 0 , p_task = tasks_array ; i < tasks_count ; i++ , p_task++ )
	{
		end_x = start_x + dd_x ;
		snprintf( p_task->program_and_params , sizeof(p_task->program_and_params) ,
"dc4c_test_worker_pi %lu %lu" , start_x , end_x );
		p_task->timeout = DC4CGetTimeout(penv) ;
		start_x = end_x + 2 ;
	}

	workers_count = tasks_count ;
	nret = DC4CDoBatchTasks( penv , workers_count , tasks_array , tasks_count ) ;
	free( tasks_array );
	if( nret )
	{
		ErrorLog( __FILE__ , __LINE__ , "DC4CDoBatchTasks failed[%d]" , nret );
		return -1;
	}
	else
	{
		InfoLog( __FILE__ , __LINE__ , "DC4CDoBatchTasks ok" );
	}

	mpf_init( pi_incr );
	mpf_init_set_d( pi , 0.00 );

	for( i = 0 ; i < tasks_count ; i++ )
	{
		DC4CGetBatchTasksIp( penv , i , & ip );
		DC4CGetBatchTasksPort( penv , i , & port );
		DC4CGetBatchTasksTid( penv , i , & tid );
		DC4CGetBatchTasksProgramAndParams( penv , i , & program_and_params );
		DC4CGetBatchTasksTimeout( penv , i , & timeout );
		DC4CGetBatchTasksElapse( penv , i , & elapse );
		DC4CGetBatchTasksError( penv , i , & error );
		DC4CGetBatchTasksStatus( penv , i , & status );
		DC4CGetBatchTasksInfo( penv , i , & info );
```

```
            InfoLog( __FILE__ , __LINE__ ,   "Task[%d]-[%s][%ld]-[%s][%s][%d][%d]-[%d][%d][%s]" ,
i , ip , port , tid , program_and_params , timeout , elapse , error , WEXITSTATUS(status) , info );

            mpf_set_str( pi_incr , info , 10 );
            mpf_add( pi , pi , pi_incr );
        }

        memset( output , 0x00 , sizeof(output) );
        gmp_snprintf( output , sizeof(output)-1 , "%.Ff" , pi );
        InfoLog( __FILE__ , __LINE__ , "pi_master() - max_x[%u] tasks_count[%d] - PI[%s]" , max_x ,
tasks_count , output );

        mpf_clear( pi_incr );
        mpf_clear( pi );

        DC4CCleanEnv( & penv );
        InfoLog( __FILE__ , __LINE__ , "DC4CCleanEnv ok" );

        return 0;
}

int pi_worker( unsigned long start_x , unsigned long end_x )
{
        mpf_t               _4 ;
        unsigned long    x ;
        int           flag ;
        mpf_t               pi_incr ;
        mpf_t               pi ;

        char         output[ 1024 + 1 ] ;

        DC4CSetAppLogFile( "pi_worker" );
        SetLogLevel( LOGLEVEL_DEBUG );

        InfoLog( __FILE__ , __LINE__ , "pi_worker" );

        if( start_x % 2 == 0 )
                start_x++;
        if( end_x % 2 == 0 )
                end_x++;

        if( start_x < 1 )
                start_x = 1 ;
        if( end_x < start_x )
```

```
        end_x = start_x ;

    mpf_init_set_d( _4 , 4.00 );
    mpf_init( pi_incr );
    mpf_init( pi );

    /*
                                                0          1          2          3          4
                                                12345678901234567890123456789012345678901234567890
                    1    1    1    1    1
        PI = 4 * ( _ - _ + _ - _ + _ ... ) = 3.14159265358979323846264338327950
                    1    3    5    7    9
                                    4 1000000000 3.14159265158640477943 14.962s
                                      1000000000 3.14159265557624002834 56.315s
                                     4 100000000 3.14159263358945602263   1.460s
                                       100000000 3.14159267358843756024   5.888s
                                        10000000 3.14159285358961767296   0.621s
                                         1000000 3.14159465358577968703   0.091s
                                          100000 3.14161265318979787768   0.011s
                                           10000 3.14179261359579270235   0.003s
    */
    mpf_set_d( pi , 0.00 );
    flag = ((start_x/2)%2)?'-':' ' ;
    for( x = start_x ; x <= end_x ; x += 2 )
    {
        mpf_div_ui( pi_incr , _4 , x );
        if( flag == '-' )
            mpf_neg( pi_incr , pi_incr );
        mpf_add( pi , pi , pi_incr );

        flag = '-' + ' ' - flag ;
    }

    memset( output , 0x00 , sizeof(output) );
    gmp_snprintf( output , sizeof(output)-1 , "%.Ff" , pi );
    InfoLog( __FILE__ , __LINE__ , "pi_worker() - start_x[%lu] end_x[%lu] - PI[%s]" , start_x , end_x ,
output );

    DC4CSetReplyInfo( output );

    mpf_clear( _4 );
    mpf_clear( pi_incr );
    mpf_clear( pi );
```

```
	return 0;
}

int main( int argc , char *argv[] )
{
	if( argc == 1 + 3 )
	{
		return pi_master( argv[1] , atol(argv[2]) , atoi(argv[3]) );
	}
	else if( argc == 1 + 2 )
	{
		return pi_worker( (unsigned long)atol(argv[1]) , (unsigned long)atol(argv[2]) );
	}
	else
	{
		printf( "USAGE : dc4c_test_worker_pi rservers_ip_port max_x worker_count\n" );
		printf( "                                start_x end_x\n" );
		exit(7);
	}
}
```

6.2 互联网数据挖掘平台

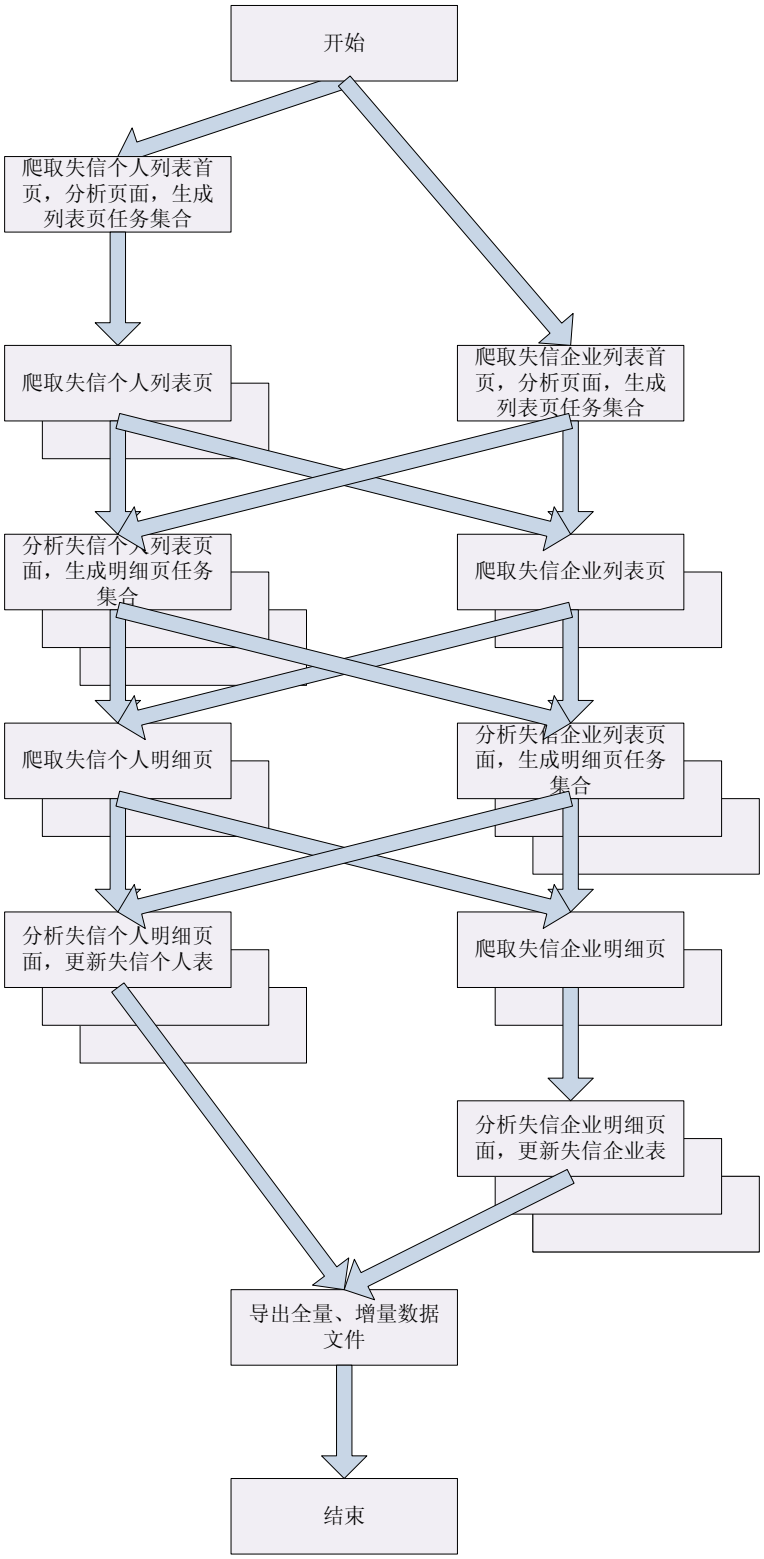


分布式计算框架 DC4C 基于网络和操作系统实例组建计算集群。

DAG 任务调度引擎封装了原生的 DC4C 同步任务控制接口，实现了树状任务流调度。

互联网数据挖掘平台批量平台负责向下对接任务调度引擎，向上作为批量任务用户程序容器。

开发各种各样的批量应用，部署在批量平台容器内供执行调度。



高耗网络带宽 IO 批次任务和高耗存储 IO 批次任务并行执行，充分利用系统资源，提高整体性能。

7 附件

7.1 附件 A.任务调度引擎 数据库表结构

7.1.1 计划表

TABLE NAME		dag_schedule		
TYPE	LENGTH	NAME	DESC	REMARK
INT	4	order_index	顺序索引	NOT NULL
STRING	64	schedule_name	计划名称	NOT NULL
STRING	256	schedule_desc	计划描述	
STRING	19	begin_datetime	开始执行日期时间	
STRING	19	end_datetime	结束执行日期时间	
INT	4	progress	进度	DC4C_DAGSCHEDULE_PROGRESS_INIT 0 DC4C_DAGSCHEDULE_PROGRESS_EXECUTING 1 DC4C_DAGSCHEDULE_PROGRESS_FINISHED 2 DC4C_DAGSCHEDULE_PROGRESS_FINISHED_WITH_ERROR 4
UNIQUE INDEX1 : order_index UNIQUE INDEX2 : schedule_name				

7.1.2 批量表

TABLE NAME		dag_batches_info		
TYPE	LENGTH	NAME	DESC	REMARK
STRING	64	schedule_name	计划名称	NOT NULL
STRING	64	batch_name	批量名称	NOT NULL
STRING	256	batch_desc	批量描述	
INT	4	view_pos_x	图形编辑坐标 X	
INT	4	view_pos_y	图形编辑坐标 Y	
INT	4	interrupt_by_app	是否允许应用中断	1:interrupt 0:not interrupt
STRING	19	begin_datetime	开始执行日期时间	
STRING	19	end_datetime	结束执行日期时间	
INT	4	progress	进度	DC4C_DAGBATCH_PROGRESS_INIT 0 DC4C_DAGBATCH_PROGRESS_EXECUTING 1 DC4C_DAGBATCH_PROGRESS_FINISHED 2 DC4C_DAGBATCH_PROGRESS_FINISHED_WITH_ERROR 4
UNIQUE INDEX1 : schedule_name , batch_name				

7.1.3 批量依赖关系表

TABLE NAME		dag_batches_direction		
TYPE	LENGTH	NAME	DESC	REMARK
STRING	64	schedule_name	计划名称	NOT NULL
STRING	64	from_batch	前依赖批量名称	NOT NULL
STRING	64	to_batch	后依赖批量名称	NOT NULL 如： 批量 A -> 批量 B -> 批量 C 配置如下： 前依赖批量：批量 A 后依赖批量：批量 B 前依赖批量：批量 B 后依赖批量：批量 C
INDEX1 : schedule_name , from_batch				

7.1.4 批量任务表

TABLE NAME		dag_batches_tasks		
TYPE	LENGTH	NAME	DESC	REMARK
STRING	64	schedule_name	计划名称	NOT NULL
STRING	64	batch_name	批量名称	NOT NULL
INT	4	order_index	顺序索引	NOT NULL
STRING	256	program_and_params	执行命令行	NOT NULL
INT	4	timeout	执行程序 MD5	
STRING	19	begin_datetime	开始执行日期时间	
STRING	19	end_datetime	结束执行日期时间	
INT	4	progress	进度	DC4C_DAGTASK_PROGRESS_INIT 0 DC4C_DAGTASK_PROGRESS_EXECUTING 1 DC4C_DAGTASK_PROGRESS_FINISHED 2 DC4C_DAGTASK_PROGRESS_FINISHED_WITH_ERROR 4
INT	4	error	执行系统响应码	
INT	4	status	执行应用响应码	
UNIQUE INDEX1 : schedule_name , batch_name , order_index				

7.2 附件 B.自带测试程序用法

安装包自带测试程序用于功能测试和展示 API 使用，源代码目录在 test/dc4c_test_*

7.2.1 单任务分派

程序名： dc4c_test_master

执行语法：

```
dc4c_test_master rserver_ip:rserver_port program_and_params
# rserver_ip:rserver_port 注册节点地址，多个注册节点用','分割
# program_and_params 分派的命令行，用双引号括起来
```

执行示例：

```
# 向注册节点 192.168.6.54:12001,192.168.6.54:12002 查询一个空闲的计算节点，分派命令"
dc4c_test_worker_sleep 3"（等待 3 秒后返回）
time ./dc4c_test_master 192.168.6.54:12001,192.168.6.54:12002
"dc4c_test_worker_sleep 3"
```

7.2.2 批量任务分派

程序名： dc4c_test_batch_master

执行语法：

```
dc4c_test_batch_master rserver_ip:rserver_port,... workers_count tasks_count
program_and_params_1 ...
# rserver_ip:rserver_port 注册节点地址，多个注册节点用','分割
# worker_count 并发数量，当-1 时和任务数量一样，当-2 时当前所有空闲计算节点数量
# tasks_count 任务数量，当-1 时实际数量为-tasks_count 且分派的命令行相同，当-2 时当前所有空闲计算节点数量
# program_and_params_1 ... 分派的命令行，用双引号括起来，当 tasks_count>1 时为多个且必须写满
```

执行示例：

```
#分派命令" dc4c_test_worker_sleep 1"、" dc4c_test_worker_sleep 2"和"
dc4c_test_worker_sleep 3"，并发数量为 2 个
time ./dc4c_test_batch_master 192.168.6.54:12001,192.168.6.54:12002 2 3
"dc4c_test_worker_sleep 1" "dc4c_test_worker_sleep 2" "dc4c_test_worker_sleep 3"

#分派命令" dc4c_test_worker_sleep 10"，并发数量为 4 个，任务数量为 10 个（每个任务的命令都一样）
time ./dc4c_test_batch_master 192.168.6.54:12001,192.168.6.54:12002 4 -10
"dc4c_test_worker_sleep 10"

#分派命令" dc4c_test_worker_sleep -10"，并发数量为任务数量，任务数量为 100 个（每个任务的命令都一样）
```

```
time ./dc4c_test_batch_master 192.168.6.54:12001,192.168.6.54:12002 -1 -100
"dc4c_test_worker_sleep -10"

#分派命令" dc4c_test_worker_sleep -10"，并发数量为当前所有空闲计算节点数量，任务数量为 100
个（每个任务的命令都一样）
time ./dc4c_test_batch_master 192.168.6.54:12001,192.168.6.54:12002 -2 -100
"dc4c_test_worker_sleep -10"
```

7.2.3 多批量任务分派

程序名： dc4c_test_multi_batch_master

执行语法：

```
dc4c_test_multi_batch_master rserver_ip:rserver_port,... envs_count workers_count
task_count program_and_params_1 ...
# envs_count 批量数量
# （其它见批量任务分派）
```

执行示例：

```
#分派命令" dc4c_test_worker_sleep -10"，批量数量为 3 批，并发数量为任务数量，任务数量为 100
个（每个任务的命令都一样）
time ./dc4c_test_multi_batch_master 192.168.6.54:12001,192.168.6.54:12002 3 -1 -100
"dc4c_test_worker_sleep -10"
```