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 工作原理 编写 第三章 安装部署 编写 第六章 应用案例

目录索引

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。

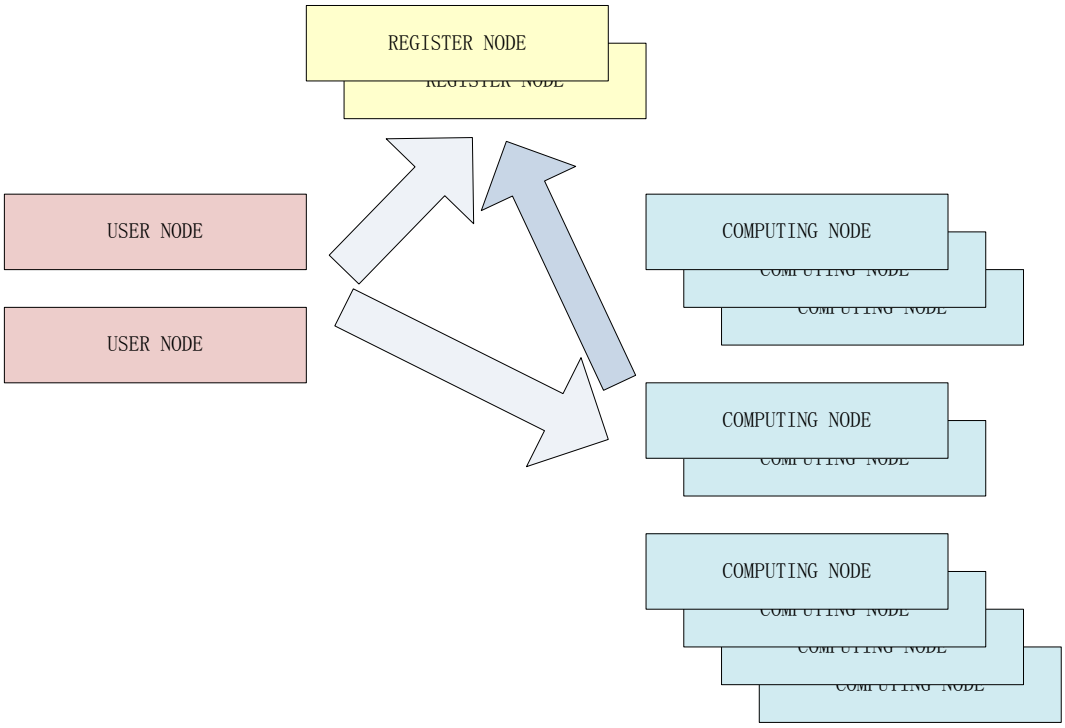
DC4C 借鉴了 Hadoop、Fourinone 等分布式产品的设计，加入自有特色，充分考虑高可靠性、高伸缩性，也是业界首先实现有向无环图任务调度引擎的分布式计算框架之一，特别适合批量任务流处理的分布式架构。

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。

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。

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。

注册节点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。

计算节点进程框架为父进程+子进程组（计算节点组）监控进程异常。

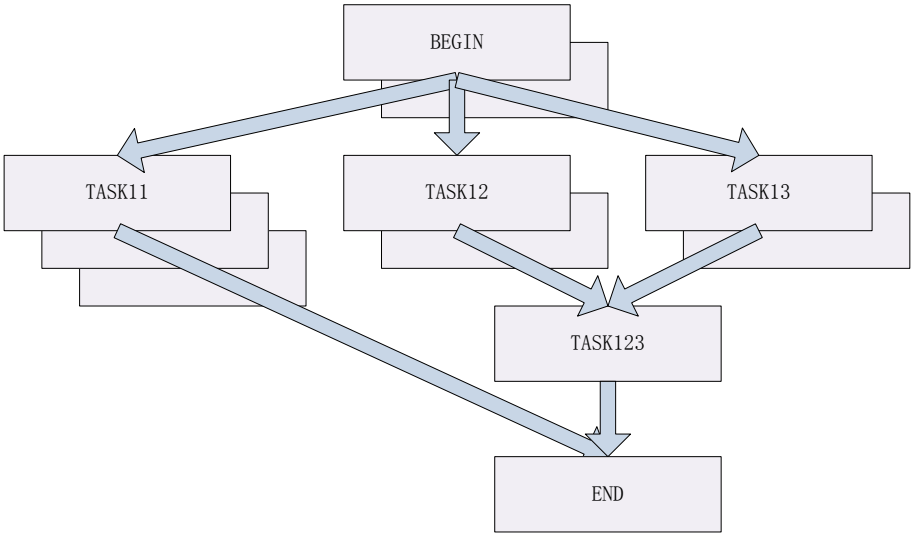
计算节点向注册节点注册后保持长连接，并互相心跳。

用户节点通过分派任务 API 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、异步任务分派，支持用户程序的多路复用结构。

* 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。

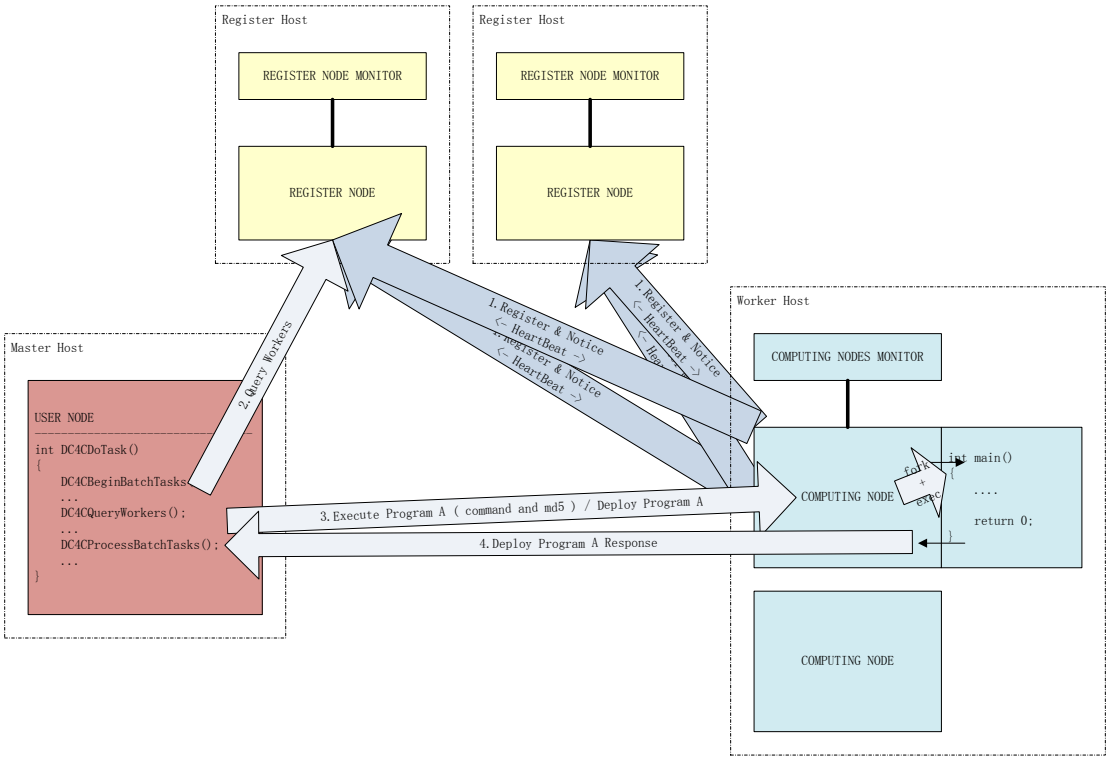
1.4 与 Hadoop 比较

	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，较少部署多个。主要多线程并发	M*N，M 台机器，每台机器最多可部署 N 个计算节点组，完全的自由部署。支持多进程或多线程并发
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；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 工作原理

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。

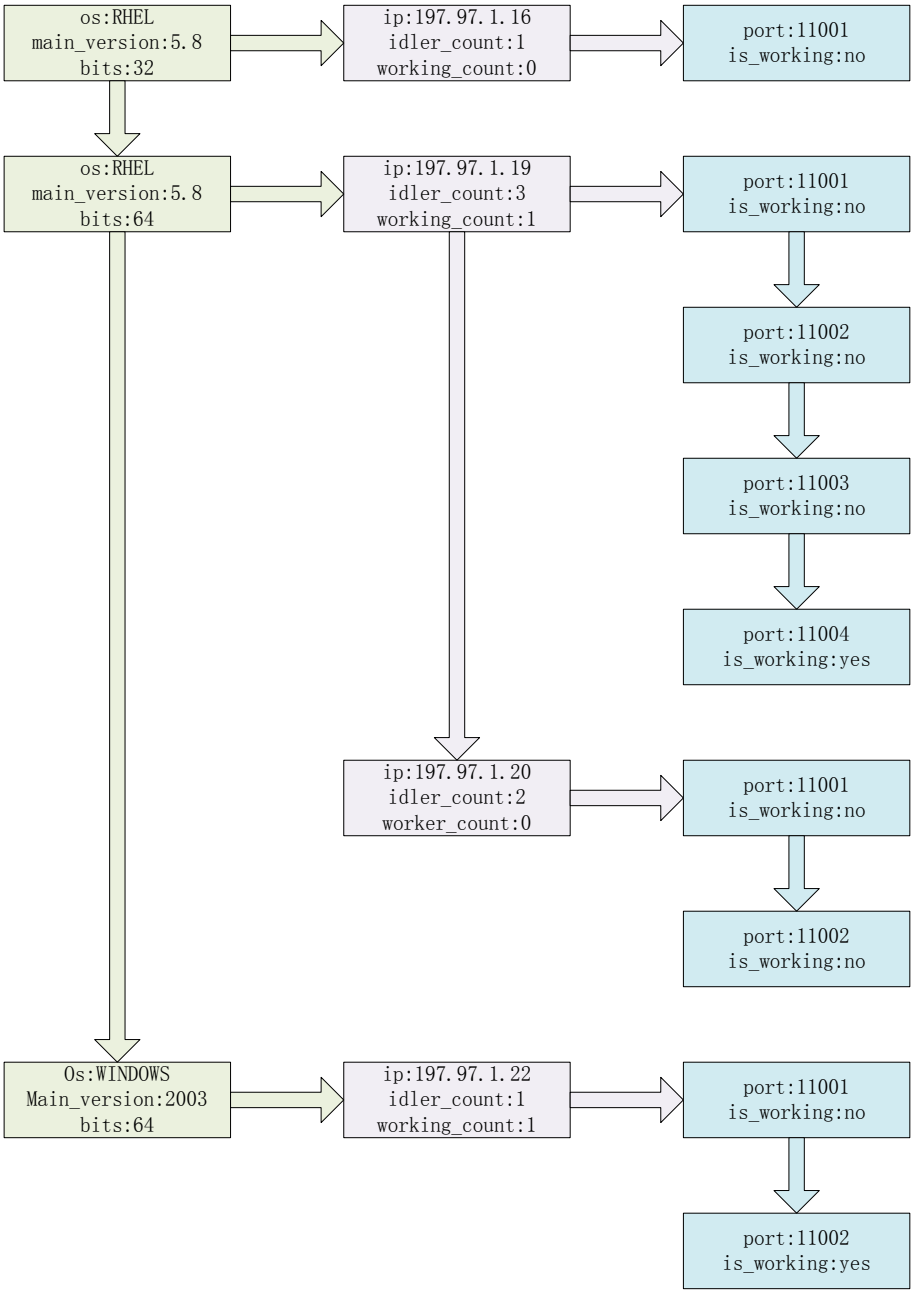
该端口还用于接受直接连接（如 telnet），接收命令行并返回命令执行响应，命令主要包含查询已注册计算节点信息。

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

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

用户节点由分布在任意主机内的用户进程组成。

用户节点进程由用户自己控制，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。

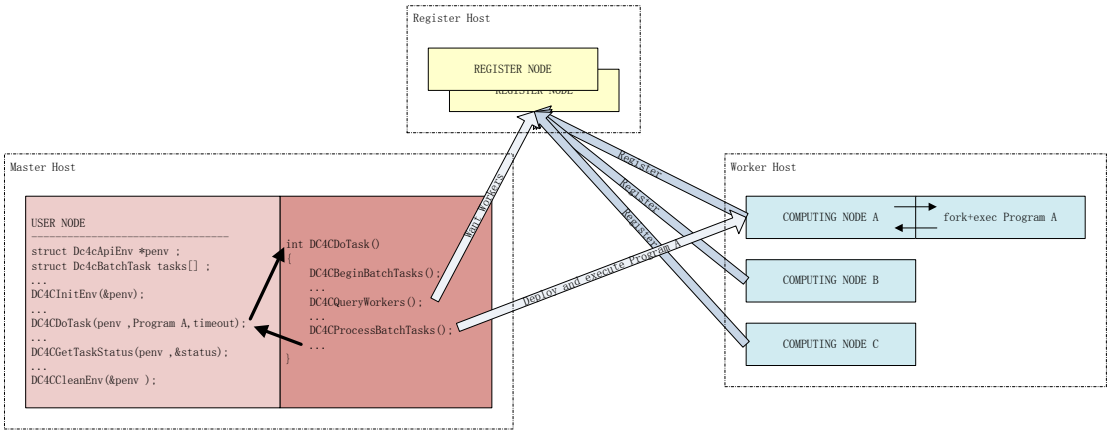
用户程序在使用用户节点 API 包的其它函数前应调用初始化任务环境函数 DC4CInitEnv 构建句柄，在结束任务处理后应调用清理任务环境函数 DC4CCleanEnv 清理和销毁句柄。

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

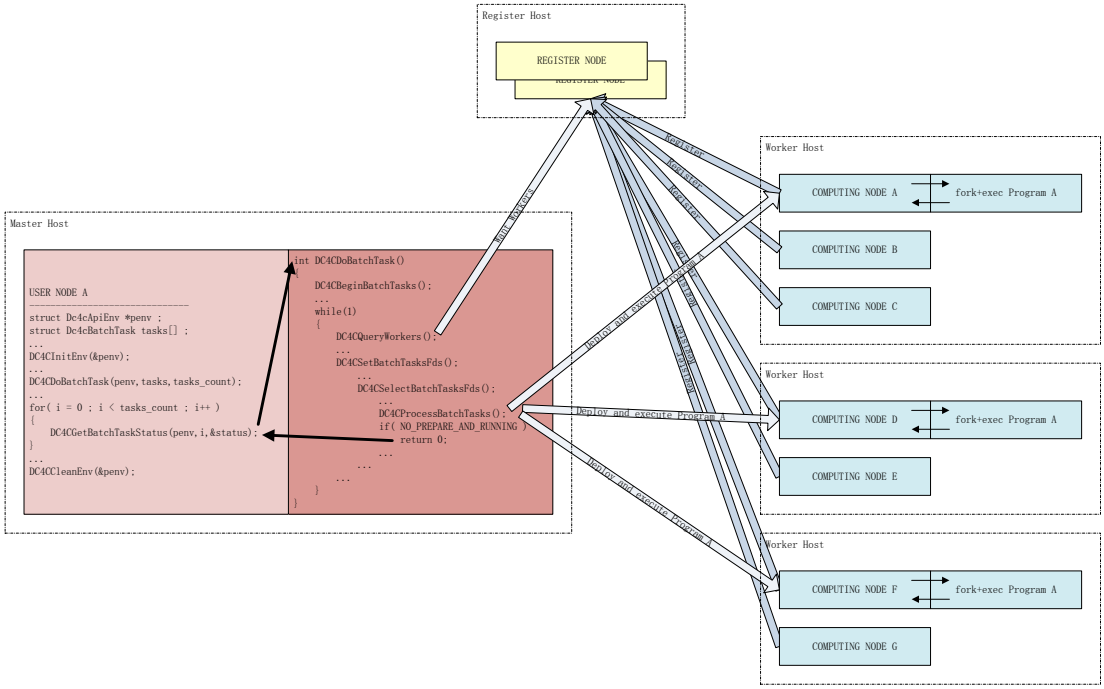
可能调用的其它 API：

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

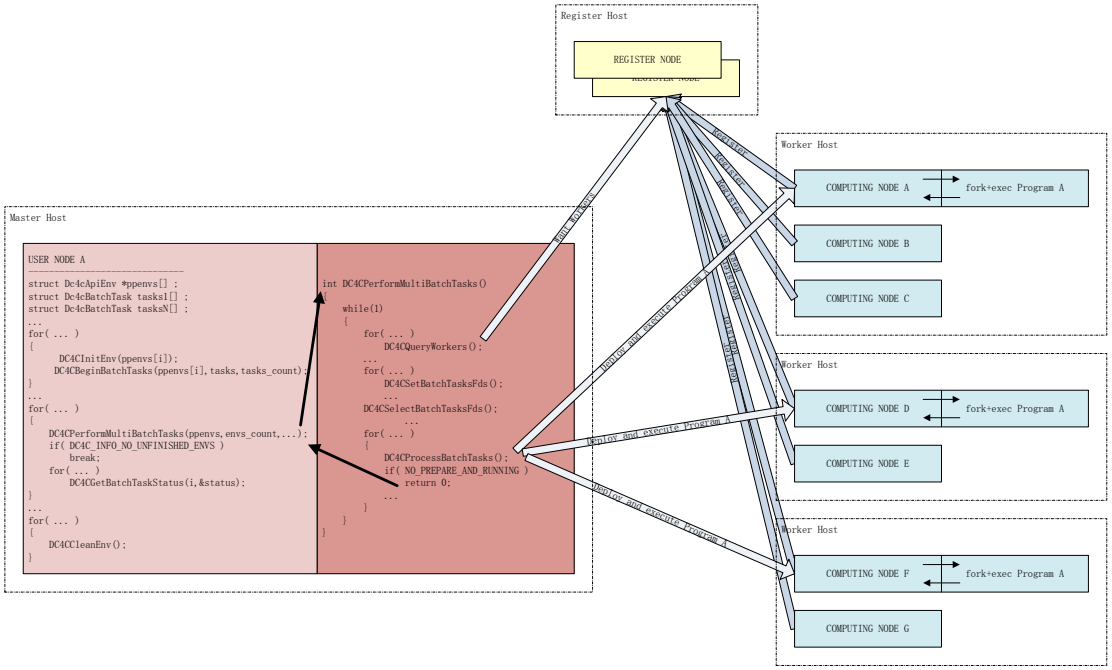
可能调用的其它 API：

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 DC4CPerformMultiBatchTasks 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

DC4CBeginBatchTasks - 开始批量任务

DC4CPerformMultiBatchTasks -当有批量任务完成时会抛出对应任务环境，直到所有批量任务都结束

DC4CCleanEnv - 清理任务环境

可能调用的其它 API：

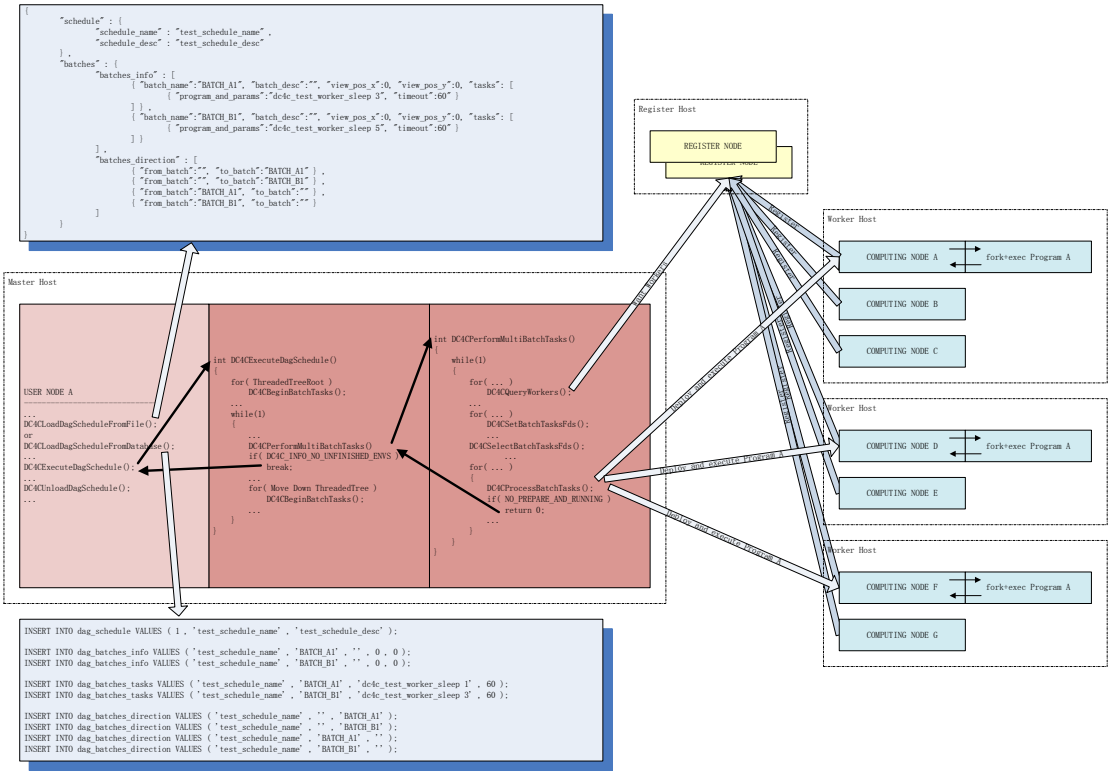
DC4CSetTimeout - 设置全局超时时间

DC4CSetOptions - 设置公共选项

DC4CGetBatchTasks* - 得到执行反馈

2.2 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，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	127.0.0.1:13001~13004
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_master 127.0.0.1:12001 "dc4c_test_worker_hello world"
DC4CInitEnv ok
DC4CDoTask ok
Task-[127.0.0.1][13002]-[1429486448000025446][dc4c_test_worker_hello world][60][0]-[0][0][hello
world]
DC4CCleanEnv ok
```

发送"world"，某个计算节点执行程序"dc4c_test_worker"命令行参数"world"，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

4.1.1.1 DC4CInitEnv

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” ) ;
```

4.1.1.2 DC4CCleanEnv

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 );
```

4.1.3 异步发起任务类

4.1.3.1DC4CBeginBatchTasks

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

4.1.3.2DC4CPerformMultiBatchTasks

函数原型： int DC4CPerformMultiBatchTasks(struct Dc4cApiEnv **ppenvs , int envs_count , struct Dc4cApiEnv **ppenv , int *p_remain_envs_count);

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_remain_envs_count 当前还剩下批量任务数量

函数返回值为 0 表示有批量任务结束了，DC4C_INFO_NO_UNFINISHED_ENVS 为所有批量任务都结束了，非 0 表示失败

代码示例：

```
struct Dc4cApiEnv      **ppenvs = NULL ;
struct Dc4cBatchTask *tasks_array = NULL ;
...
ppenvs = (struct Dc4cApiEnv**)malloc( sizeof(struct Dc4cApiEnv*) * envs_count ) ;
...
for( i= 0 ; i < envs_count ; i++ )
{
     nret = DC4CInitEnv( & (ppenvs[i]) , NULL ) ;
     ...
    tasks_array = (struct Dc4cBatchTask *)malloc( sizeof(struct Dc4cBatchTask) *
tasks_count ) ;
    for( j = 0 , p_task = tasks_array ; j < tasks_count ; j++ , p_task++ )
    {
        strcpy( p_task->program_and_params , "dc4c_test_worker_sleep -5" );
        p_task->timeout = DC4CGetTimeout(penv) ;
    }
    nret = DC4CBeginBatchTasks( ppenvs[i] , workers_count , tasks_array , tasks_count ) ;
    free( tasks_array ) ;
}
...
while(1)
{
     nret = DC4CPerformMultiBatchTasks( ppenvs , envs_count , & penv , &
remain_envs_count ) ;
     if( nret == DC4C_INFO_NO_UNFINISHED_ENVS )
          break;
```

```
    else if( nret )
        return nret;
}
...
for( i= 0 ; i < envs_count ; i++ )
{
    nret = DC4CCleanEnv( & (ppenvs[i]) ) ;
}
```

4.1.4 获取执行反馈类

4.1.4.1DC4CGetTask*

函数原型： int 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

Ip 计算节点 IP

Port 计算节点 PORT

Tid 任务 ID

ProgramAndParams 执行命令行

Timeout 配置的超时时间（单位：秒）

Elapse 实际执行时间（单位：秒）

Error 分发和反馈任务中发生的错误，0 为没有错误

Status 程序执行返回状态，即 waitpid 得到的 status

Info 应用返回信息，最长 1024 字节

函数返回值为 0 表示成功，非 0 表示失败

代码示例：

```
    char        *ip = NULL ;
    long        port ;
    char        *tid = NULL ;
    char        *program_and_params = NULL ;
    int         timeout ;
    int         elapse ;
    int         error ;
    int         status ;
    char        *info = NULL ;
```

```
...

        DC4CGetTaskIp( penv , & ip );
        DC4CGetTaskPort( penv , & port );
        DC4CGetTaskTid( penv , & tid );
        DC4CGetTaskProgramAndParams( penv , & program_and_params );
        DC4CGetTaskTimeout( penv , & timeout );
        DC4CGetTaskElapse( penv , & elapse );
        DC4CGetTaskError( penv , & error );
        DC4CGetTaskStatus( penv , & status );
        DC4CGetTaskInfo( penv , & info );
        printf( "Task-[%s][%ld]-[%s][%s][%d][%d]-[%d][%d][%s]\n" , ip , port , tid ,
program_and_params , timeout , elapse , error , WEXITSTATUS(status) , info );
```

4.1.4.2 DC4CGetBatchTasks*

函数原型： int 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

4.1.5 低层函数类

4.1.5.1 DC4CQueryWorkers

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

4.1.5.2 DC4CSetBatchTasksFds

函数原型： int DC4CSetBatchTasksFds(struct Dc4cApiEnv *penv , 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

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

4.1.5.3 DC4CSelectBatchTasksFds

函数原型： int DC4CSelectBatchTasksFds(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

4.1.5.4DC4CProcessBatchTasks

函数原型： int DC4CProcessBatchTasks(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

4.1.6 其它类

4.1.6.1DC4CResetFinishedTasksWithError

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

4.1.6.2DC4CGetUnusedWorkersCount

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）

4.2.2.3 DC4CSetReplyInfoEx

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

4.3.1.1 DC4CLoadDagScheduleFromFile

函数原型： int DC4CLoadDagScheduleFromFile(struct Dc4cDagSchedule **pp_sched , char *pathfilename , 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

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 DC4CLoadDagScheduleFromDatabase

函数原型： int DC4CLoadDagScheduleFromDatabase(struct Dc4cDagSchedule **pp_sched , char *schedule_name , int options);

函数描述： 从数据库中载入 DAG 流程模型的任务树配置

函数说明： 申请 Dc4cDagSchedule DAG 流程模型环境结构所需内存，并初始化，从数据库载入 DAG 流程模型任务数配置，让(*pp_sched)指向该结构。

char *schedule_name 计划名

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： （数据库配置表详见附件 A）

代码示例：

```
struct Dc4cDagSchedule     *p_sched = NULL ;
...
nret = DC4CLoadDagScheduleFromDatabase( & p_sched , "test_dag_schedule" ,
DC4C_OPTIONS_INTERRUPT_BY_APP ) ;
...
DC4CUnloadDagSchedule( & p_sched );
```

4.3.1.3 DC4CExecuteDagSchedule

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

函数描述： 执行 DAG 流程模型的任务树

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 );
```

4.3.1.4 DC4CUnloadDagSchedule

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

函数描述： 从数据库中载入 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

4.3.1.5 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

4.3.2 低层函数

4.3.2.1 DC4CLoadDagScheduleFromStruct

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。

4.3.2.2 DC4CInitDagSchedule

函数原型： int DC4CInitDagSchedule(struct Dc4cDagSchedule *p_sched , char *schedule_name , char *schedule_desc);

函数描述： 初始化 DAG 流程模型任务树

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

char *schedule_name 计划名

char *schedule_desc 计划描述

函数返回值为 0 表示成功，非 0 表示失败

4.3.2.3DC4CCleanDagSchedule

函数原型： void DC4CCleanDagSchedule(struct Dc4cDagSchedule *p_sched);

函数描述： 清理 DAG 流程模型任务树

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

4.3.2.4DC4CAllocDagBatch

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

4.3.2.5DC4CFreeDagBatch

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

4.3.2.6DC4CLinkDagBatch

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

4.3.2.7DC4CSetDagBatchTasks

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

4.3.2.8DC4CGetDagBatchApiEnvPPtr

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

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

4.3.2.9DC4CGetDagBatchBeginDatetimeStamp

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

4.3.2.10 DC4CGetDagBatchEndDatetimeStamp

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

4.3.2.11 DC4CGetDagBatchProgress

函数原型： void DC4CGetDagBatchProgress(struct Dc4cDagBatch *p_batch , int *p_progress);

函数描述： 返回批量节点里的进度

函数说明： struct Dc4cDagBatch *p_batch 批量节点

int *p_progress 存放返回的进度的地址

4.3.2.12 DC4CGetDagBatchResult

函数原型： void DC4CGetDagBatchResult(struct Dc4cDagBatch *p_batch , int *p_result);

函数描述： 返回批量节点里的结果

函数说明： struct Dc4cDagBatch *p_batch 批量节点

int *p_progress 存放返回的结果的地址

4.4 代码示例

4.4.1 单任务

源代码

```
#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		*ip = NULL ;
	long		port ;
	char		*tid = NULL ;
	char		*program_and_params = NULL ;
	int		timeout ;
	int		elapse ;
	int		error ;
	int		status ;
	char		*info = NULL ;

	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 , 60 );
		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" );
		}

		DC4CGetTaskIp( penv , & ip );
		DC4CGetTaskPort( penv , & port );
		DC4CGetTaskTid( penv , & tid );
		DC4CGetTaskProgramAndParams( penv , & program_and_params );
		DC4CGetTaskTimeout( penv , & timeout );
		DC4CGetTaskElapse( penv , & elapse );
		DC4CGetTaskError( penv , & error );
		DC4CGetTaskStatus( penv , & status );
		DC4CGetTaskInfo( penv , & info );
		printf( "Task-[%s][%ld]-[%s][%s][%d][%d]-[%d][%d][%s]\n" , ip , port , tid ,
program_and_params , timeout , elapse , error , WEXITSTATUS(status) , info );

		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

源代码

```
#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"
*/

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

	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 , 60 );
	DC4CSetOptions( penv , DC4C_OPTIONS_INTERRUPT_BY_APP );

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

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

	if( workers_count < 0 )
	{
		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" );
		}

		printf( "tasks_count[%d] worker_count[%d] - prepare_count[%d] running_count[%d]
finished_count[%d]\n" , DC4CGetTasksCount(penv) , DC4CGetWorkersCount(penv) ,
DC4CGetPrepareTasksCount(penv) , DC4CGetRunningTasksCount(penv) ,
DC4CGetFinishedTasksCount(penv) );

		tasks_count = DC4CGetTasksCount( penv ) ;
		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 );
			printf( "Task[%d]-[%s][%ld]-[%s][%s][%d][%d]-[%d][%d][%s]\n" , i , ip , port , tid ,
program_and_params , timeout , elapse , error , WEXITSTATUS(status) , info );
		}

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

	return 0;
}
```

4.4.3 多批量任务

源代码

```
#include "dc4c_api.h"

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

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

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

	int			nret = 0 ;

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

	if( argc >= 1 + 3 )
	{
		envs_count = 2 ;
		ppenvs = (struct Dc4cApiEnv**)malloc( sizeof(struct Dc4cApiEnv*) * envs_count ) ;
```

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

nret = DC4CInitEnv( & (ppenvs[0]) , argv[1] ) ;
if( nret )
{
	printf( "DC4CInitEnv failed[%d]\n" , nret );
	return 1;
}
else
{
	printf( "DC4CInitEnv ok\n" );
}

DC4CSetOptions( ppenvs[0] , DC4C_OPTIONS_INTERRUPT_BY_APP );

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

DC4CSetOptions( ppenvs[1] , DC4C_OPTIONS_INTERRUPT_BY_APP );

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

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

```
}

if( workers_count < 0 )
{
	workers_count = tasks_count ;
}

strcpy( task.program_and_params , "dc4c_test_worker_sleep 3" );
task.timeout = 0 ;

nret = DC4CBeginBatchTasks( ppenvs[0] , workers_count , & task , 1 ) ;
if( nret )
{
	printf( "DC4CBeginBatchTasks failed[%d] , errno[%d]\n" , nret , errno );
	return 1;
}

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 = 0 ;
}

nret = DC4CBeginBatchTasks( ppenvs[1] , 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( ppenvs , envs_count , & penv , & remain_envs_count ) ;
                if( nret == DC4C_INFO_NO_UNFINISHED_ENVS )
                {
                        printf( "DC4CPerformMultiBatchTasks ok , all is done\n" );

                        break;
                }
                else if( nret )
                {
                        printf( "DC4CPerformMultiBatchTasks failed[%d]\n" , nret );
                }
                else
                {
                        printf( "DC4CPerformMultiBatchTasks ok\n" );

                }

                tasks_count = DC4CGetTasksCount( penv ) ;
                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 );
                        printf( "Task[%d]-[%s][%ld]-[%s][%s][%d][%d]-[%d][%d][%s]\n" , i , ip , port ,
tid , program_and_params , timeout , elapse , error , WEXITSTATUS(status) , info );
                }

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

        DC4CCleanEnv( & (ppenvs[0]) );
        DC4CCleanEnv( & (ppenvs[1]) );
        free( ppenvs );
        printf( "DC4CCleanEnv ok\n" );
    }
```

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

	return 0;
}
```

4.4.4 DAG 调度多批量任务（从配置文件载入配置）

配置文件

```
/*
                                        _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": [
				{ "program_and_params":"dc4c_test_worker_sleep 4", "timeout":60" }
			] } ,
			{ "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 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"
*/

int main( int argc , char *argv[] )
{
        struct Dc4cDagSchedule      *p_sched = NULL ;

        int                     nret = 0 ;

        DC4CSetAppLogFile( "dc4c_test_tfc_dag_master" );
        SetLogLevel( LOGLEVEL_DEBUG );

        if( argc == 1 + 2 )
```

```
	{
		nret = DC4CLoadDagScheduleFromFile( & p_sched , argv[2] ,
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 , argv[1] ) ;
		if( nret )
		{
			printf( "DC4CExecuteDagSchedule failed[%d]\n" , nret );
			return 1;
		}
		else
		{
			printf( "DC4CExecuteDagSchedule ok\n" );
		}

		DC4CUnloadDagSchedule( & p_sched );
		printf( "DC4CCleanEnv ok\n" );
	}
	else
	{
		printf( "USAGE : dc4c_test_tfc_dag_master rservers_ip_port .dag_schedule\n" );
		exit(7);
	}

	return 0;
}
```

4.4.5 DAG 调度多批量任务（从数据载入配置）

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' );

INSERT INTO dag_batches_info VALUES ( 'test_schedule_name' , 'BATCH_A1' , '' , 0 , 0 );
INSERT INTO dag_batches_info VALUES ( 'test_schedule_name' , 'BATCH_B1' , '' , 0 , 0 );
INSERT INTO dag_batches_info VALUES ( 'test_schedule_name' , 'BATCH_A21' , '' , 0 , 0 );
INSERT INTO dag_batches_info VALUES ( 'test_schedule_name' , 'BATCH_A22' , '' , 0 , 0 );
INSERT INTO dag_batches_info VALUES ( 'test_schedule_name' , 'BATCH_A3' , '' , 0 , 0 );
INSERT INTO dag_batches_info VALUES ( 'test_schedule_name' , 'BATCH_B2' , '' , 0 , 0 );

INSERT INTO dag_batches_tasks VALUES ( 'test_schedule_name' , 'BATCH_A1' , 'dc4c_test_worker_sleep 1' ,
60 );
INSERT INTO dag_batches_tasks VALUES ( 'test_schedule_name' , 'BATCH_A21' , 'dc4c_test_worker_sleep 4' ,
60 );
INSERT INTO dag_batches_tasks VALUES ( 'test_schedule_name' , 'BATCH_A22' , 'dc4c_test_worker_sleep 5' ,
60 );
INSERT INTO dag_batches_tasks VALUES ( 'test_schedule_name' , 'BATCH_A3' , 'dc4c_test_worker_sleep 2' ,
60 );
INSERT INTO dag_batches_tasks VALUES ( 'test_schedule_name' , 'BATCH_A3' , 'dc4c_test_worker_sleep 1' ,
60 );
INSERT INTO dag_batches_tasks VALUES ( 'test_schedule_name' , 'BATCH_B1' , 'dc4c_test_worker_sleep 3' ,
60 );
INSERT INTO dag_batches_tasks VALUES ( 'test_schedule_name' , 'BATCH_B2' , 'dc4c_test_worker_sleep 6' ,
60 );

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

```
#include "dc4c_api.h"
#include "dc4c_tfc_dag.h"
#include "IDL_dag_schedule.dsc.ESQL.eh"

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

```
*/

int main( int argc , char *argv[] )
{
	struct Dc4cDagSchedule	*p_sched = NULL ;

	int			nret = 0 ;

	DSCDBCONN( "0.0.0.0" , 18432 , "calvin" , "calvin" , "calvin" );
	if( SQLCODE )
	{
		printf( "DSCDBCONN failed , SQLCODE[%d][%s][%s]\n" , SQLCODE , SQLSTATE , SQLDESC );
		return 1;
	}
	else
	{
		printf( "DSCDBCONN ok\n" );
	}

	DC4CSetAppLogFile( "dc4c_test_tfc_dag_master" );
	SetLogLevel( LOGLEVEL_DEBUG );

	if( argc == 1 + 2 )
	{
		nret = DC4CLoadDagScheduleFromDatabase( & p_sched , argv[2] ,
DC4C_OPTIONS_INTERRUPT_BY_APP ) ;
		if( nret )
		{
			printf( "DC4CLoadDagScheduleFromDatabase failed[%d]\n" , nret );
			return 1;
		}
		else
		{
			printf( "DC4CLoadDagScheduleFromDatabase ok\n" );
		}

		DC4CLogDagSchedule( p_sched );

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

```
            else
            {
                  printf( "DC4CExecuteDagSchedule ok\n" );
            }

            DC4CUnloadDagSchedule( & p_sched );
            printf( "DC4CCleanEnv ok\n" );
      }
      else
      {
            printf( "USAGE : dc4c_test_tfc_dag_master rservers_ip_port .dag_schedule\n" );
            exit(7);
      }

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

      return 0;
}
```

5 内部实现

（待写）

6 应用案例

6.1 计算圆周率

源代码

```
#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" );
```

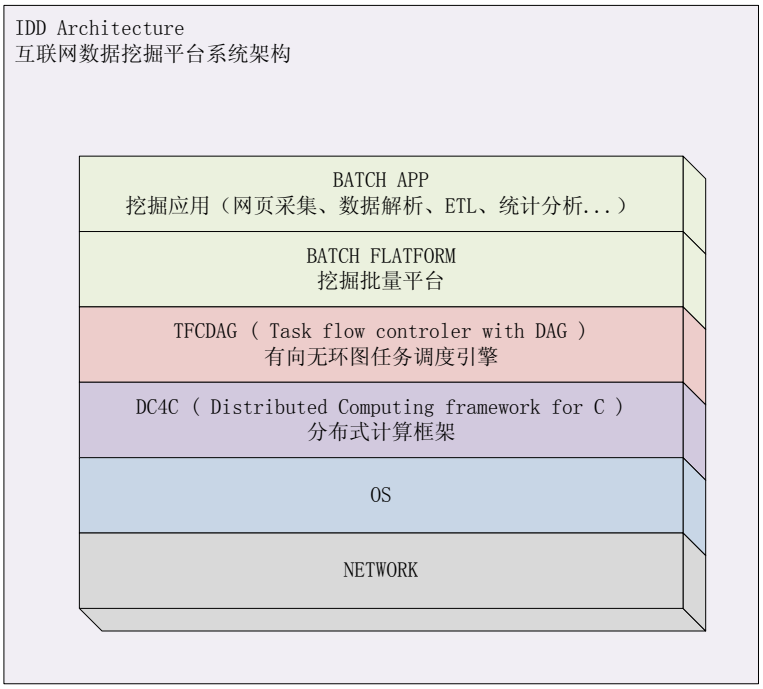
```
	nret = DC4CInitEnv( & penv , rservers_ip_port ) ;
	if( nret )
	{
		ErrorLog( __FILE__ , __LINE__ , "DC4CInitEnv failed[%d]" , nret );
		return -1;
	}
	else
	{
		InfoLog( __FILE__ , __LINE__ , "DC4CInitEnv ok" );
	}

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

	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
                                          12345678901234567890123456789012345678901
                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 max_x worker_count\n" );
                printf( "                                    start_x end_x\n" );
                exit(7);
        }
}
```

6.2 互联网数据挖掘平台

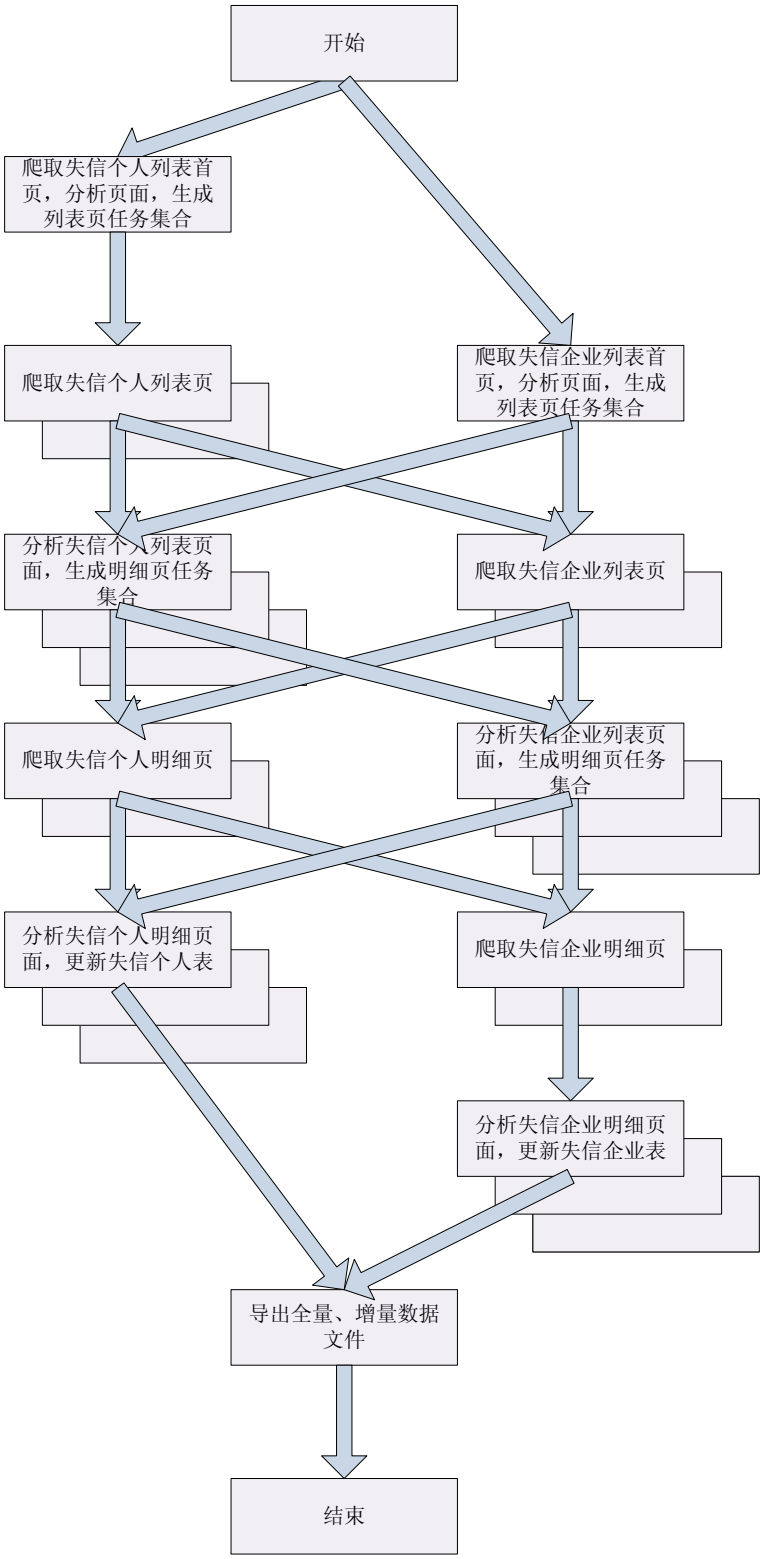


分布式计算框架 DC4C 基于网络和操作系统实例组建计算集群。

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	NOT NULL
INT	4	order_index	NOT NULL
STRING	64	schedule_name	NOT NULL
STRING	256	schedule_desc	
UNIQUE INDEX1 : order_index UNIQUE INDEX2 : schedule_name			

7.1.2 批量表

TABLE NAME		dag_batches_info	
TYPE	LENGTH	NAME	NOT NULL
STRING	64	schedule_name	NOT NULL
STRING	64	batch_name	NOT NULL
STRING	256	batch_desc	
INT	4	view_pos_x	
INT	4	view_pos_y	
UNIQUE INDEX1 : schedule_name , batch_name			

7.1.3 批量依赖关系表

TABLE NAME		dag_batches_direction	
TYPE	LENGTH	NAME	NOT NULL
STRING	64	schedule_name	NOT NULL
STRING	64	from_batch	NOT NULL
STRING	64	to_batch	
INDEX1 : schedule_name , from_batch			

7.1.4 批量任务表

TABLE NAME		dag_batches_tasks	
TYPE	LENGTH	NAME	NOT NULL
STRING	64	schedule_name	NOT NULL
STRING	64	batch_name	NOT NULL
INT	4	order_index	NOT NULL
STRING	256	program_and_params	NOT NULL
INT	4	timeout	NOT NULL
UNIQUE INDEX1 : schedule_name , batch_name , order_index			