1. 系统设计

## 系统设计原则

在软件的设计过程当中，应当遵循以下几个原则：

1. 可靠性

在可靠代码设计的前提下，能够很好地提高应用的稳定性，可靠的代码越是底层与需要，防止出现系统瘫痪等应用的故障。同时，模块间的可靠性也能够在某种程度上减少调试时的工作量。

1. 健壮性

健壮的代码设计可以很好的抵抗软件的再运行过程当中出现的各种异常错误和异常现象。因为在代码最初设计的时候，总是会有各种各样实现没有想到的、或者说是没有考虑到的错误场景。如果没有一个良好的、健壮性的代码设计，应用有可能就会直接报各种各样的错误然后退出，或者在没有报错的时候继续以错误的中间过程完成某项任务，这样会造成更大的生产损失。

1. 效率性

在设计的时候应该要考虑到软件的复杂性，应尽量避免含有复杂的循环当中。

1. 可扩展性

软件通常在设计的时候，都会留有一些升级接口和升级的空间，对未来不确定的扩展开放，对修改开放。

(5）安全性

要求系统能够对用户的信息和某些操作有安全保护的机制，同时系统自身也需要能够处理很多安全的漏洞且遇到故障时能够自我修复，以提高系统的安全性能。

### 逻辑清晰原则

一个程序，它在设计的过程当中要做到逻辑清晰合理，设计具体实现之前应当能够在逻辑上实现预期的功能，不应该在具体实现的时候逐步发现问题然后再去解决。此时，可以在能够实现预期功能的基础之上，最大程度的优化精简之前设计好的逻辑功能，实现逻辑功能之前的去耦合、低耦合、高内聚，防止出现一个小问题影响整个大全局的逻辑问题。

当然在设计的过程当中需要事先规划好整盘的逻辑，在全局的角度上判断和检查逻辑是否真的可行，然后在依次细化每个之逻辑的可行性，递归地完成全部的设计流程。

### 模块化原则

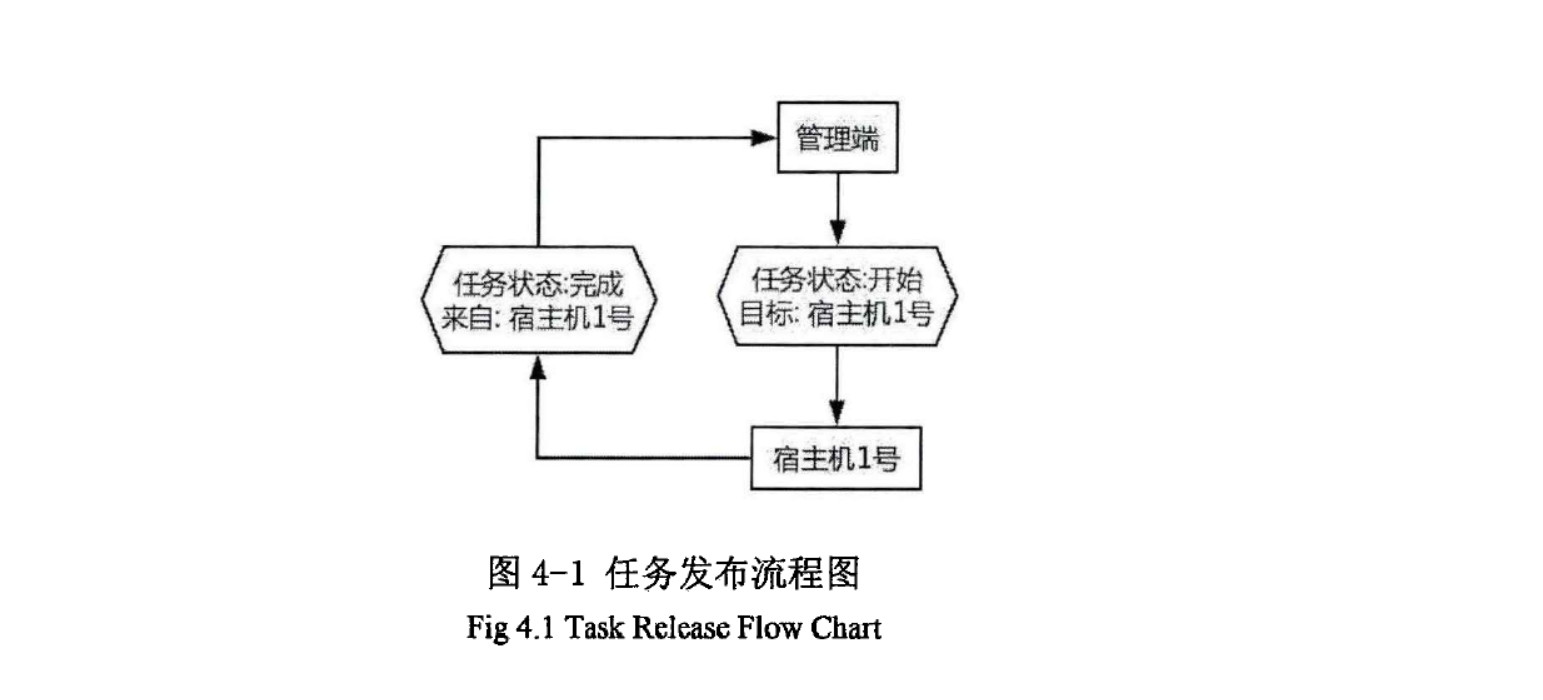
模块化能够非常好地分离每个逻辑、之逻辑之前那些耦合的地方。也就是说在最初的设计时间结构分为主程序、子程序、子过程等一系列独立的部分。在这些设计的基础之上定义好各个部分之前的输入、输出链接关系，依次逐步精简设计具体的实现，然后得到完整的功能木块。模块化的一个最关键的目的就是为如降低程序之间的复杂性，进一步简化程序的设计流程，方便程序员调试和后期的维护。

一般稍微大型点的软件，在设计的时候都要尽可能的根据更高效、更绿色的设计方法进行设计，其中模块化的设计方法就是绿色设计方法中对的一种。在当前的软件开发技术当中，模块化的设计原则，已经从最初的设计理念逐步进化到了一个成熟明确的设计方法。该设计方法的成熟有效地缩短了开发周期，让开发者们在互相不干扰的情况下并行地进行各自的开发任务。

## 系统总体设计

### 功能架构设计

从更高点的角度上来看，该软件事先的功能是虚拟机的智能调度，角色可以分为调度者和被调度者。在全局的范畴内只有一个调度者来负责全部实体的调度行为，当然该对象也被称为manager。如在manager中执行虚拟机的一个开机操作时，流程如下图4.1



如图4.1所示，在特点条件下触发新的任务时，调度者(图中manager)会重新构建一个任务数据结构(图中六边形对象),数据结构中包含任务的一些必要的信息，如任务属性、任务种类、任务接受目标等。这个数据结构会被发往被调度者(图中typervisor1)。当被调度者接受到了任务之后开始执行任务。最后将结果发送回任务调度者，此为一个完整的任务周期。对于设计中的不同功能均为该模式为样板进行拓展，保障在全局视图下任务类的行为一致性。并以此构建任务通讯模块。

程序还需要一些能够触发调度任务的性能数据，这些数据会通过两种方式收集。即调度者主动请求以及被调度者主动上传。在传递性能数据之前会由指令发出者构建同样的任务数据结构发送给对端。无论是主动还是被动模式，最后数据都会由manager的性能数据模块整理数据并入库。

有了性能数据的基础上，调度者的调度模块会实时对这些性能数据进行计算，得出当前或者未来需要被调度的实体合适执行调度命令。结构如4.2所示

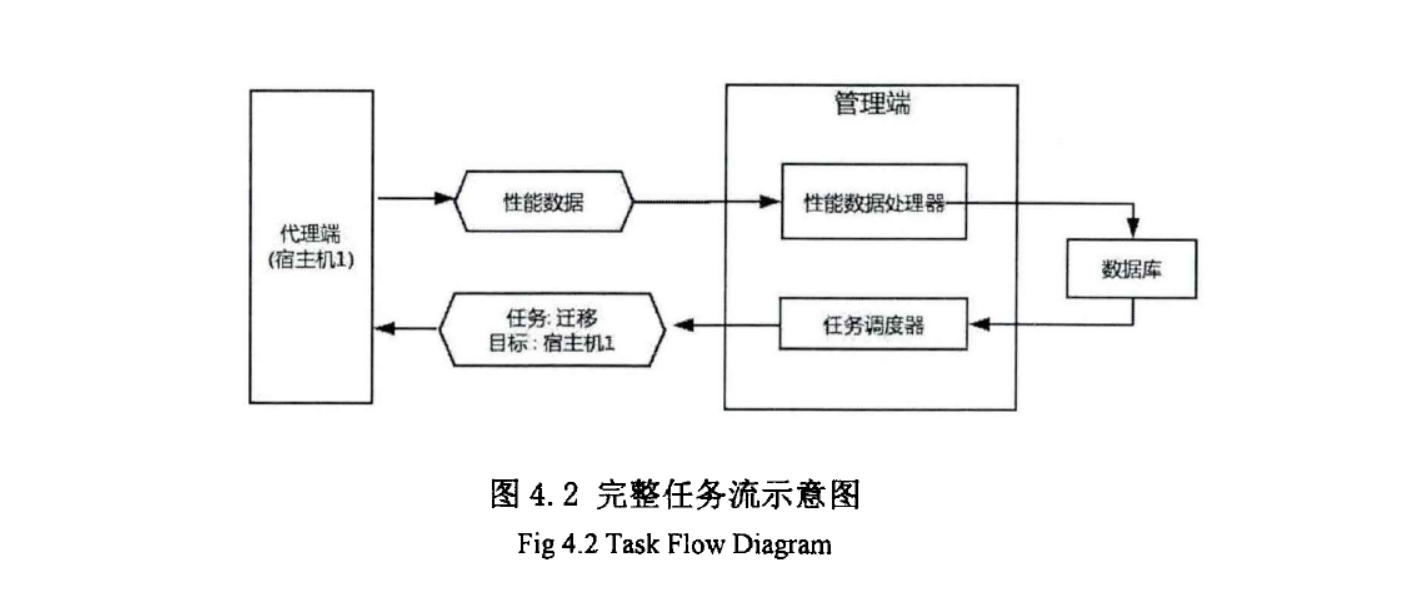
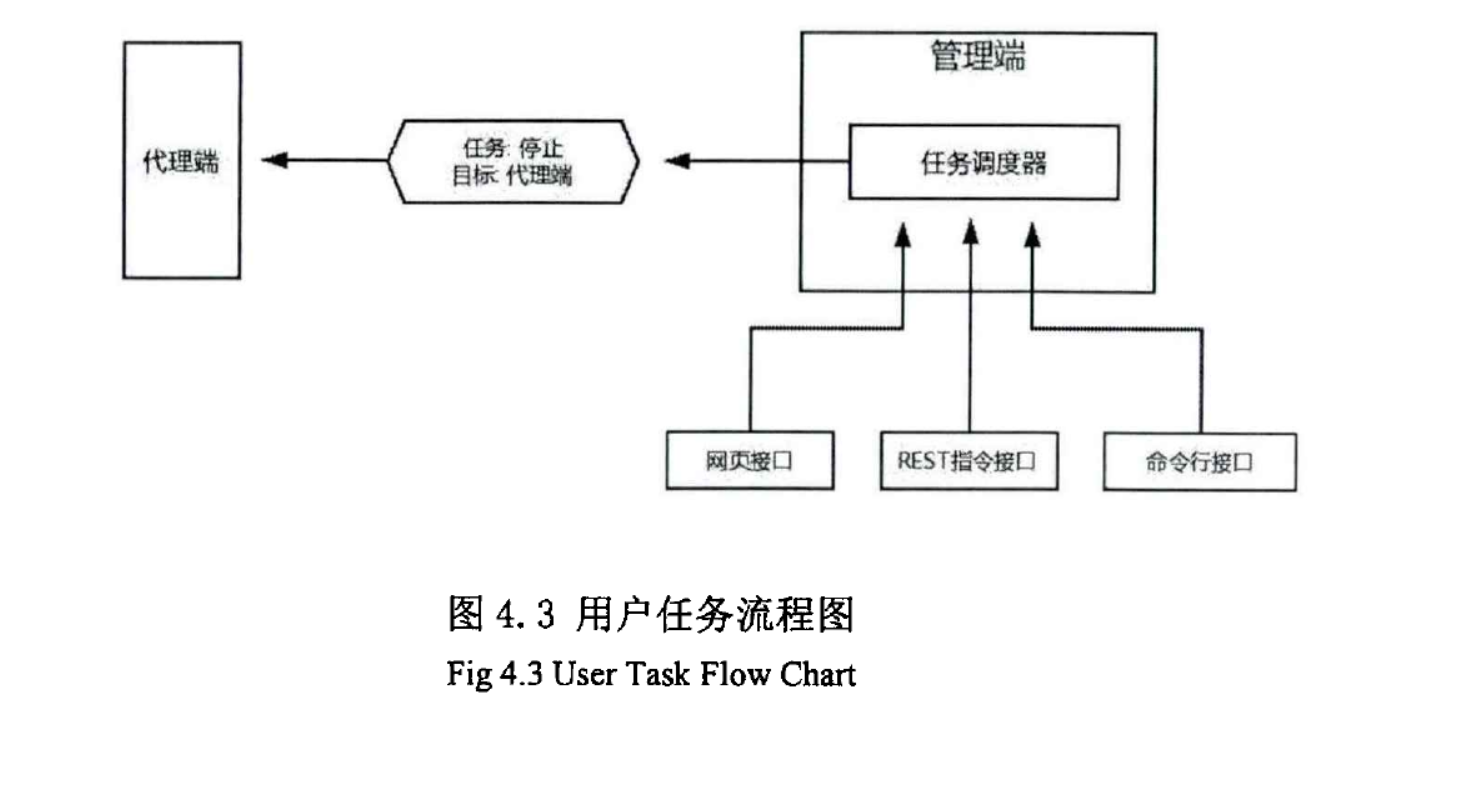
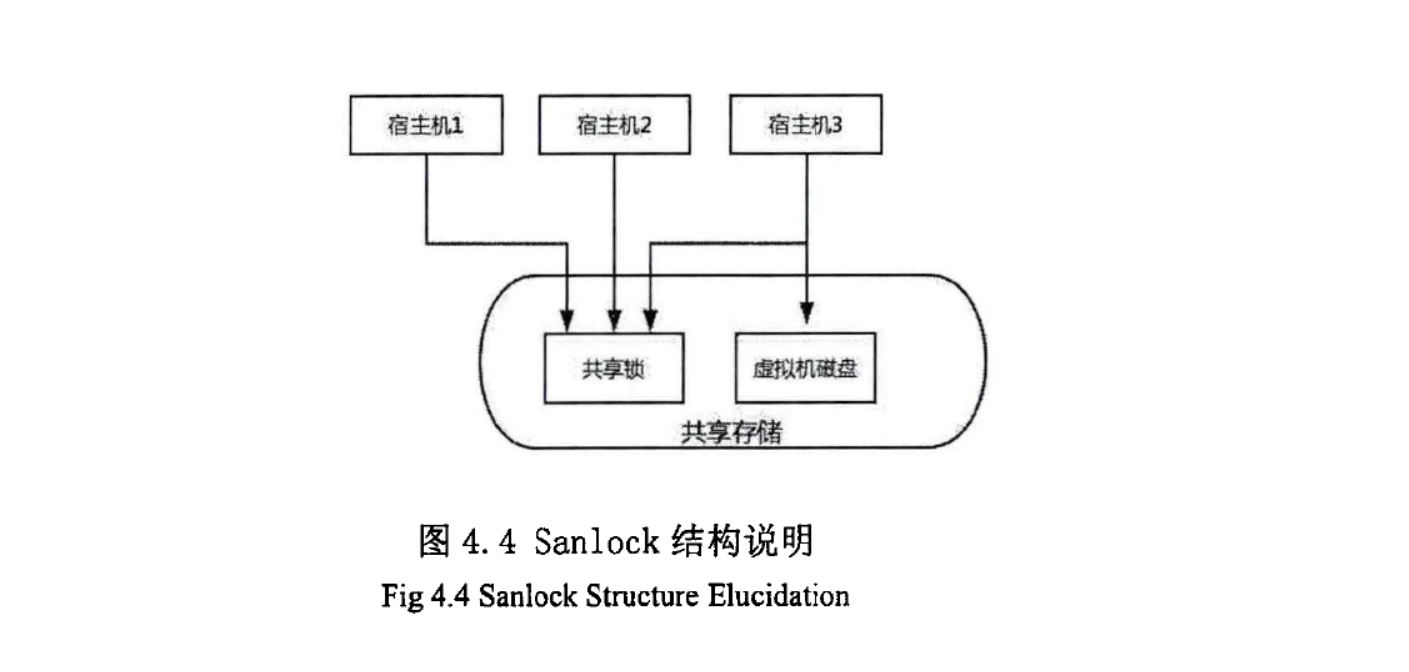


图4.2显示了自发性的任务循环流程，在该流程里无须人的手工干预可自动运行。管理员可以在限定的调度工作流下实现无线的任务调度种类，用以在某些情况下需要对特定的实例执行特定的任务。工作流如图4.3所示



当任何类型的任务通过该流程执行时，调度者都会立即处理任务。特例任务有限级远远高于自发任务，且该任务不受调度算法的控制。

除了调度者与被调度者的通讯之间的通讯外，被调度者间一般也会通过某种共享的存储来进行相互之间的通讯。该种通讯不依赖一些传统的TCP/IP协议，而是通过存储来传递状态。即Sanlock集群锁。在某些特殊的情况下回出现因为网络故障导致被调度节点丢失的问题，这时如果直接判定该节点死亡并在其他节点上启动虚拟机可能会引发数据一致性问题，导致数据磁盘损坏。Snalock能够有效防止这种事故的发生。担忧节点出现故障时，依据故障当时拥有的数据无法判定节点状态，集群中的其他节点会读取Sanlock的状态，读取到磁盘是否正在被使用，参考图4.4结构



## 系统功能详细设计

### KVM/Libvirt接口模块设计

Libvirt工具为程序提供调度，KVM为底层虚拟化的技术的实现。首先根据Libvirt官方的API文档找出所需要的基本API,包含虚拟机的基础操作列表如下：

基础操作部分：

虚拟机开机 virDomainCreate()

虚拟机关机 virDomainDestory()

虚拟机定义 virDomainDefineXML()

虚拟机销毁 virDomainUndefine()

虚拟机迁移 virDomainMigrate3()

信息收集部分：

虚拟机性能数据(处理器，内存，运行时间) virDomainGetInfo()

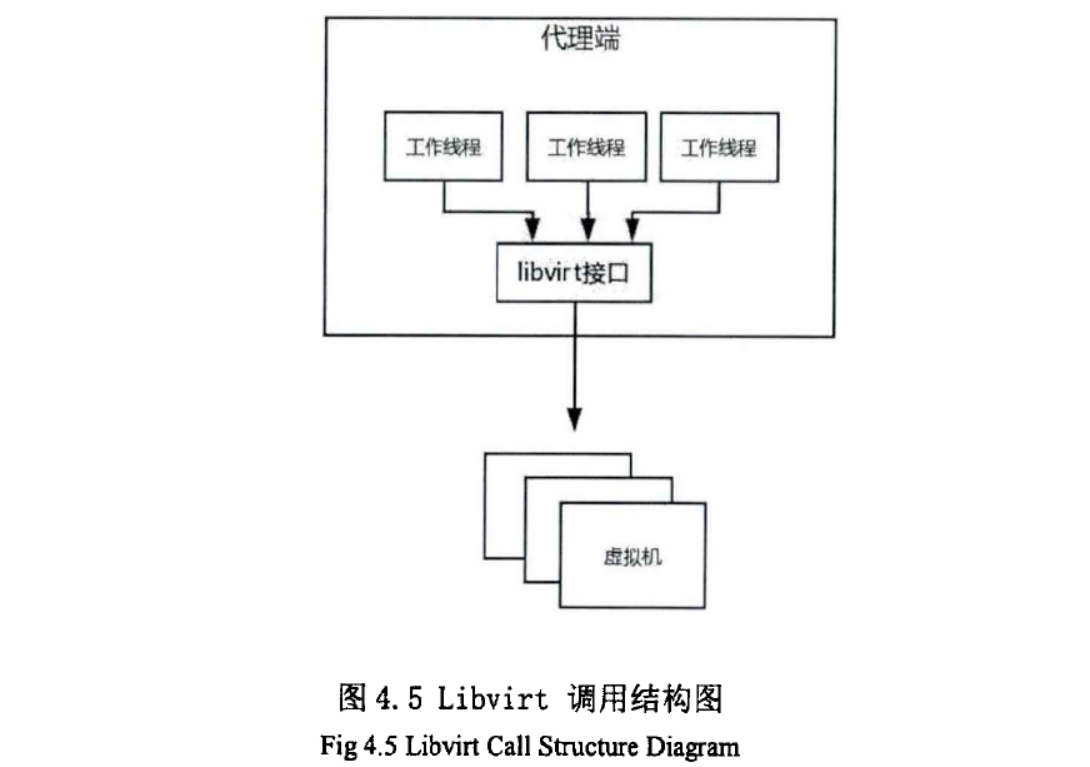
虚拟机当前任务状态 virDomainGetJobInfo() 用于获取迁移进度

虚拟机块设备任务状态 virDomainGetBlockJobInfo()

通过id查找虚拟机 virDomainLookupByID()

通过名称查找虚拟机 virDomainLookupByName()

以上的Libvirt接口作为虚拟化底层的调度接口，供上层被调度端使用。底层需要保障每个动作的行为一致性，不论一个任务是否执行成功均需要将结果正确反馈到主调函数中，防止出现调用夯死的问题。调用逻辑如果4.5所示。

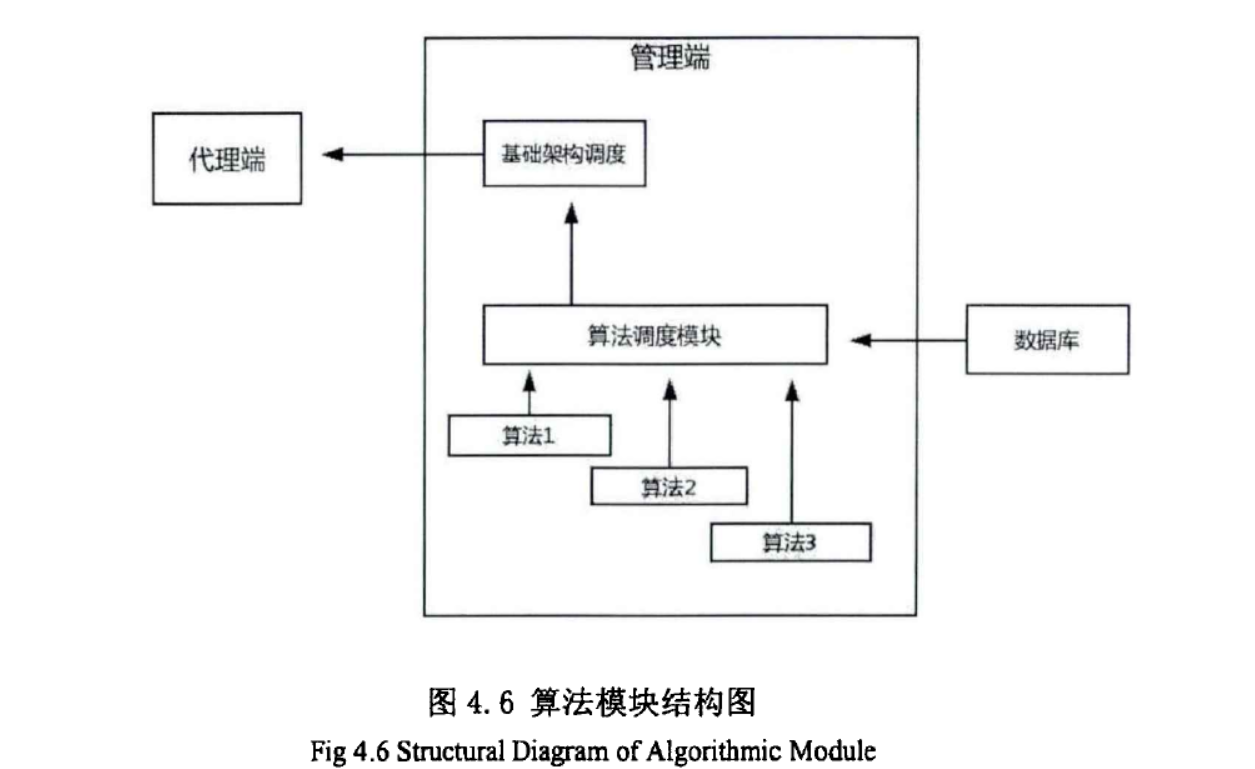


从图4.5可知，宿主机(被调度端)所有针对虚拟机的操作均通过LibvirtAPI接口，即Libvirt是程序与虚拟机通讯的唯一方式。例如调度端调度虚拟机开机，工作线程将调用virDomainLookupByID()或virDomainLookupByName()这个函数，查找虚拟机是否存在。其中前者使用id作为虚拟机唯一标识，后者使用虚拟机名称作为标识进行查找。如果能够找的到目标虚拟机则返回这个虚拟机的一个指针(C语言)或者一个Domain类型的实例(python)。但是如果查找不到则返回Libvirt事先定义好的错误代码(Libvirt.VIR\_ERR\_NO\_DOMAIN)。在能够查找到的前提下继续调用virDomainCreate()函数将这个虚拟机开机。同样如果开机成功将返回执行成功代码(0),如果开机失败则返回(-1)。以上的调用传参以及返回类型、错误代码均在Libvirt的API中定义。当出现任何问题时直接查找香瓜代码即可。

### 算法模块设计

应用会将算法和实际调度做分离处理，即算法模块仅负责计算什么时间调度、对象被执行何种调度。之后将这个结论通过构建任务的方式交给基础架构调度模块，再由调度端的调度模块来执行具体的任务。调度成功与否或者操作回滚等问题均交由调度模块，算法模块不会参与后续的操作。

算法模块会直接与数据库进行通讯，分析数据库中的性能数据。设计如图4.6所示。



算法模块可以动态的切换当前算法，例如在处理业务高峰时选择性能优先算法，将所有可用调度节点开机，并且将当前的虚拟机按照内存置备大小平均分散到所有主机中，用以应对性能需求的突发高峰问题。在夜间或者业务闲时选择资源节省优先算法，尽可能的将虚拟机调度到集中的几个节点中，空闲出其他节点，并使其他节点处于高节点甚至关机状态。另外，算法模块可以针对不同层次的抽象等级指定不同的调度策略，可以针对集群一级或者数据中心一级进行策略的指定。

除了调度模块的整体架构，该模块也包含一系列的算法，例如面向市场的虚拟资源分配调度方法或者基于遗传基因的调节算法、模拟退火算法等等。这些算法在实现的过程中唯一耦合的地方是性能信息的采集模块。只有数据库中包含了需要的性能数据信息，算法才能够正常的运行。

### 系统数据收集整理模块设计

数据收集整理模块负责全部的性能数据收集、整理、入库。如果算法模块可以类比应用的大脑，数据收集整理模块可以称之为应用的眼。任何的抽象信息都应该传递到这个模块中，并且该模块应该对数据加工处理并放置到预计的位置中。这里面抽象的信息不仅仅应该包含传统的处理器、内存、网络、吞吐额嗯性能数据，同时也指整个应用所触发的事件。例如卡机异常、关机异常、物理硬件变更、计算资源池变更事件信息。

该模块分为两部分，即信息收集子模块和性能、事件主动采集子模块。用以应对不同的响应速度需求。性能、事件主动采集子模块可以由调度端主动触发采集事件，这个事件会被立即执行到需要采集信息的被调度端，采集数据并返回。该方法有较高的时效性，可以用来作为在执行某些任务之前的预执行阶段的数据确认行为。信息收集子模块作为一种被动形式的消息接受者或者周期性数据采集者，会接受由被调度端主动上报的性能数据信息。通常情况下这个信息为实现约定好的信息格式，模块会执行处理入库等一些列操作。

### API模块设计

如总体叙述中所述，除了应用自发的对虚拟化环境进行控制，应用还提供了主动进行控制的接口，方便管理员进行实时的维护任务。

这个API是一个抽象的接口综合，包含以下几种接口:

1. 图形类接口(WEB GUI).
2. 命令行接口(CLI)
3. REST接口(REST API)

目前软件工程方面，对于直接与用户交互的界面更偏向于使用web的方式。借助于强大的html+css+js来进行开发，摒弃传统的windows应用方式。即提升了界面的开发速度也提供了良好的多平台支持性。可以做到无论用户使用何种操作系统、使用何种硬件架构平台均能够使用到一致性的前端界面和功能，大大的简化使用难度。

对于某些不具备浏览器的平台，本应用也提供了操作调度程序的接口。管理员使用命令行工具即可与调度程序进行沟通或者任务的下发。现在比较流行的云调度工具实现均提供了这种控制模式，例如微软的Azure云和阿里巴巴的阿里云。执行cli命令时接口应该提供良好的信息提示功能、出错提示以及友好的返回结果格式。

## 数据库设计

### 设计原则

数据库设计首先要满足三大范式：

第一范式：当关系模式R的所有属性都不能再分解为更基本的数据单位时，称R是满足第一范式的，简记为1NF;

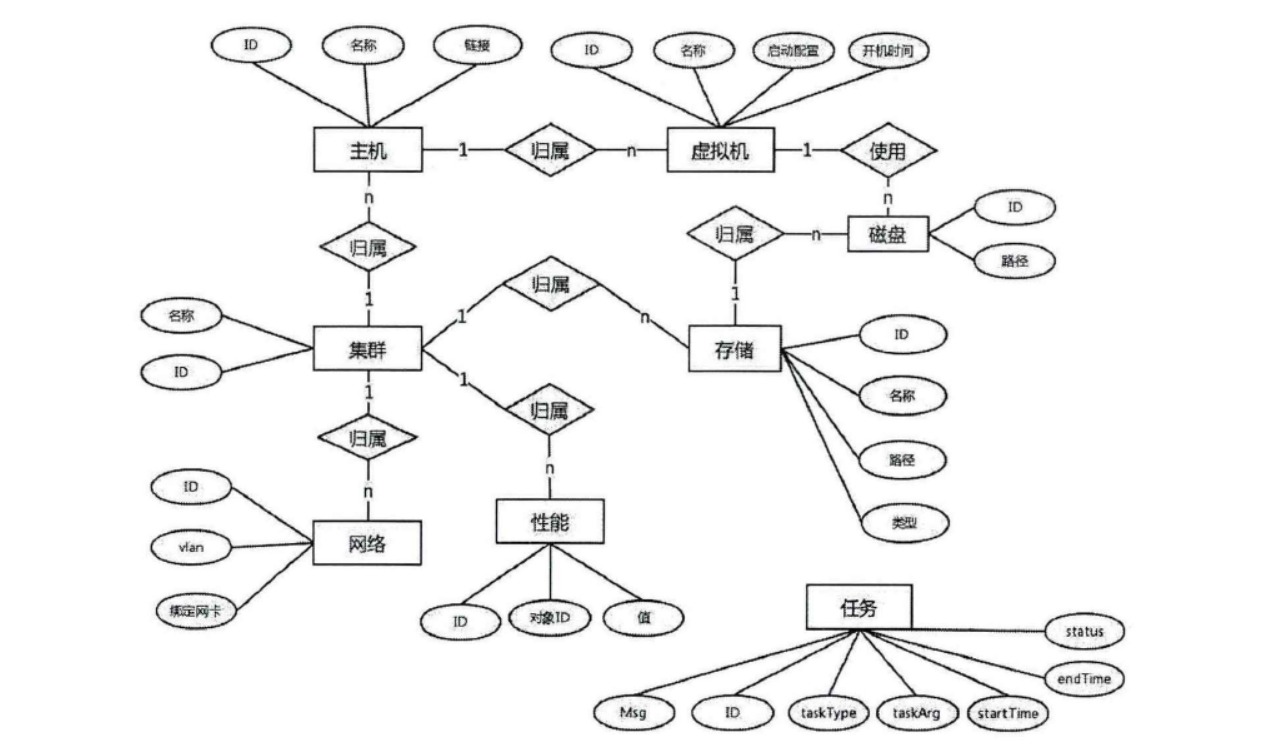
第二范式：如果关系模式R满足第一范式，并且R得所有非主属性都完全依赖于R的每一个候选关键属性，称R满足第二范式，简记为2NF;

第三范式：设R是一个满足第一范式条件的关系模式，X是R的任意属性集，如果X非传递依赖于R的任意一个候选关键字，称R满足第三范式，简记为3NF.

在此基础上进行详细的数据库设计。防止出现依赖传递，数据冗余等数据库设计不合理的问题。对于每一数据库中的每一个实体类描述则使用UUID的指定唯一性。

数据库整体使用utf8编码来应对多国语言的问题，使应用在数据持久层面上支持多国语言。

### E-R图设计

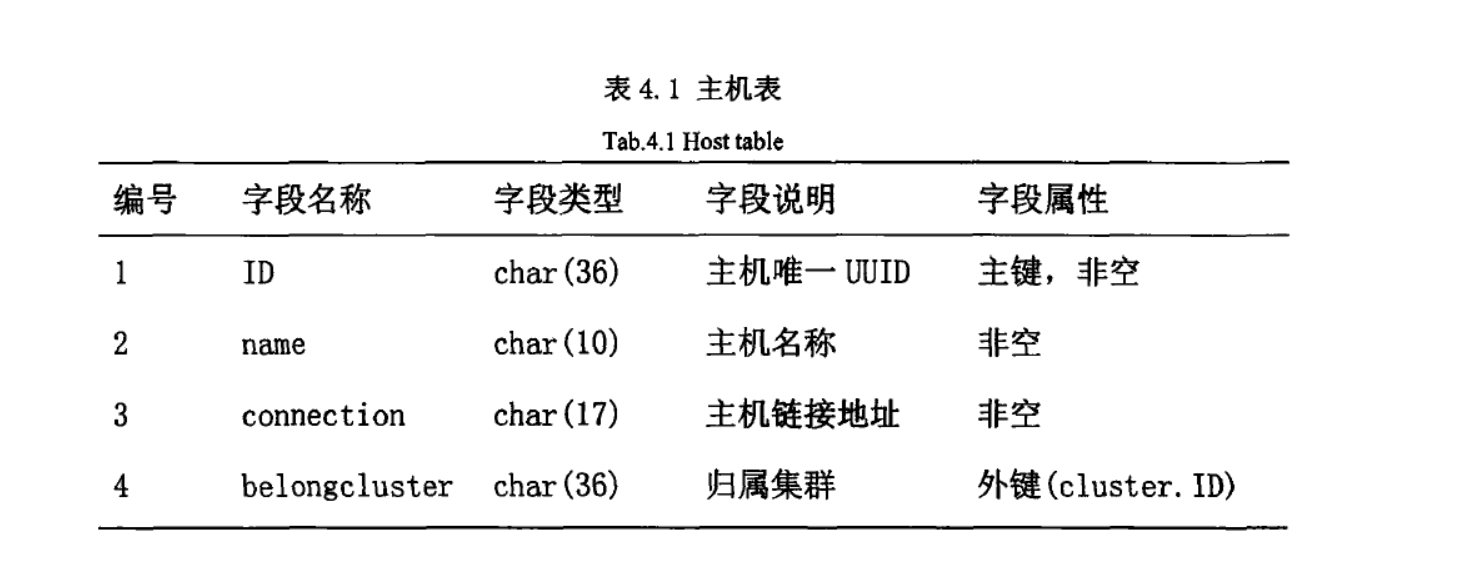


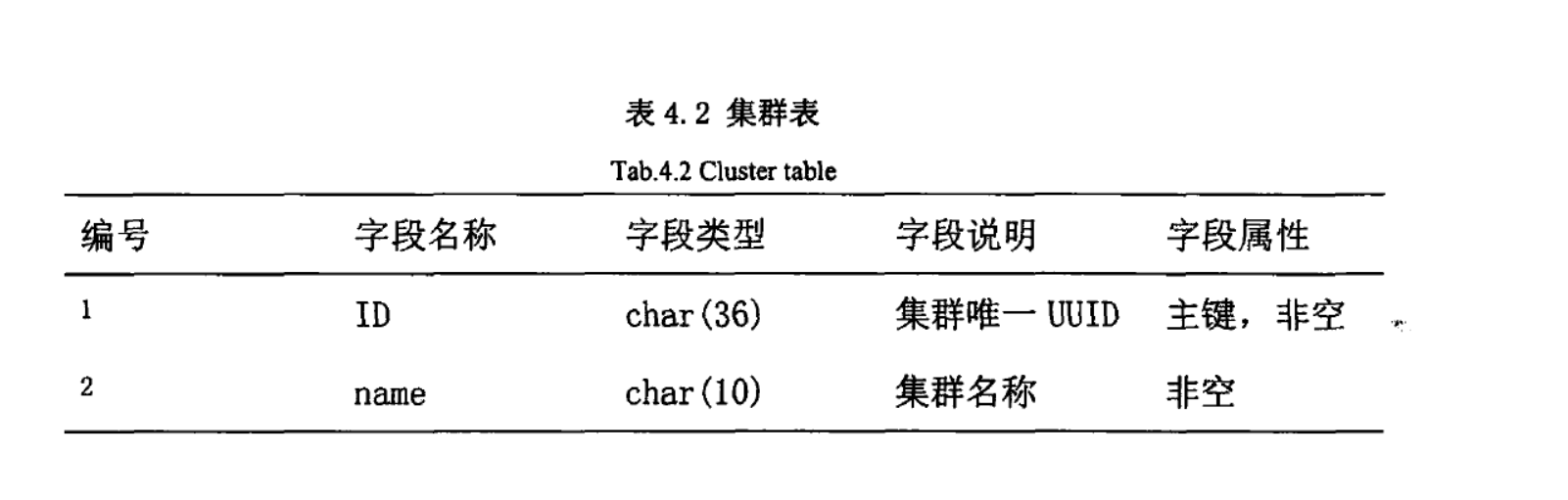
数据表说明

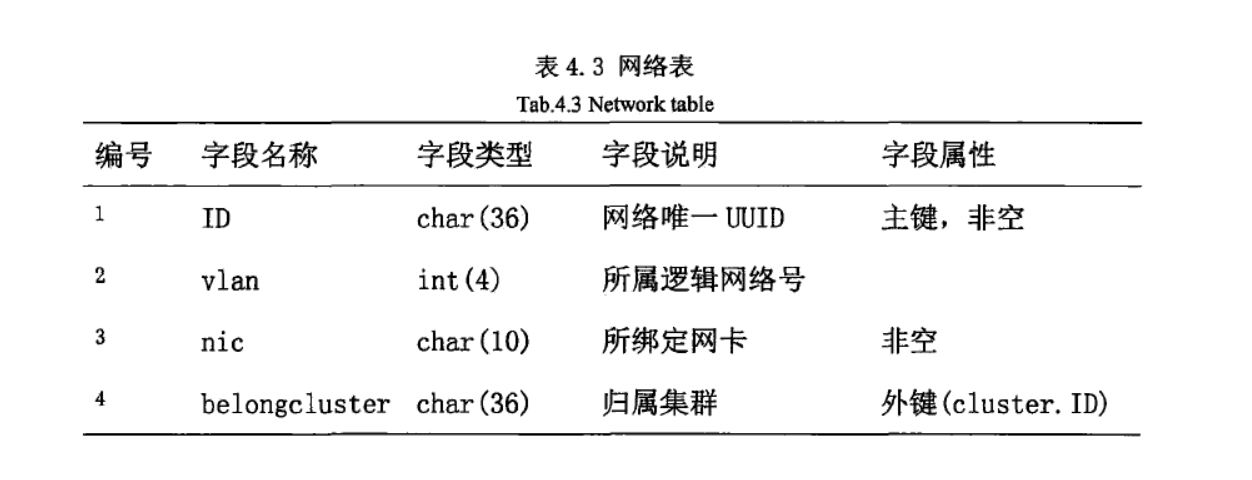
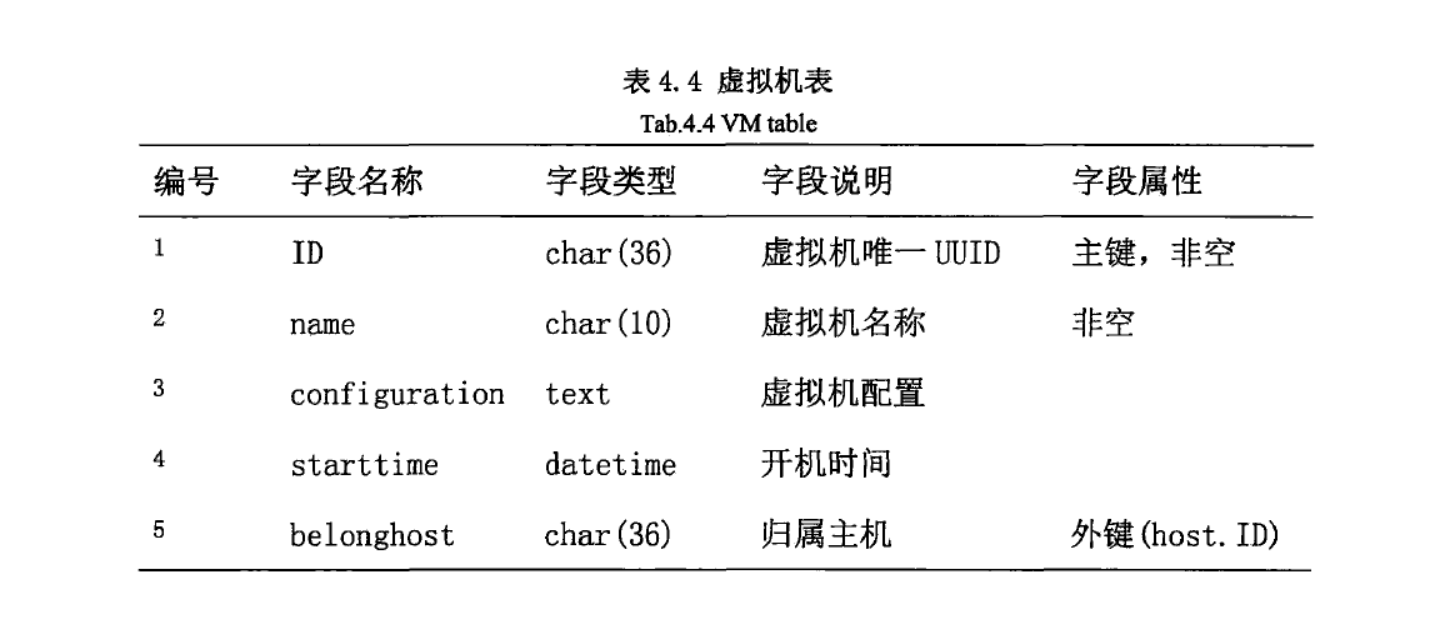
1. 主机表(host): 用来存放所有被管理主机的记录。见表4.1.
2. 集群表(cluster): 用来存放所有的集群记录。见表4.2。
3. 网络表(net): 用来存放所有的受管理网络记录。见表4.3。
4. 虚拟机表(vm): 用来存放所有的虚拟机记录。见表4.4。
5. 存储(storage): 用来存放所有的存储记录。见表4.6。
6. 磁盘(disk): 用来存放虚拟机所使用的磁盘记录。见表4.5。
7. 性能(perf): 用来存放收集的性能数据。见表4.7。
8. 任务(task): 用来存放需要处理的任务以及已经处理的任务的结果。见表4.8。

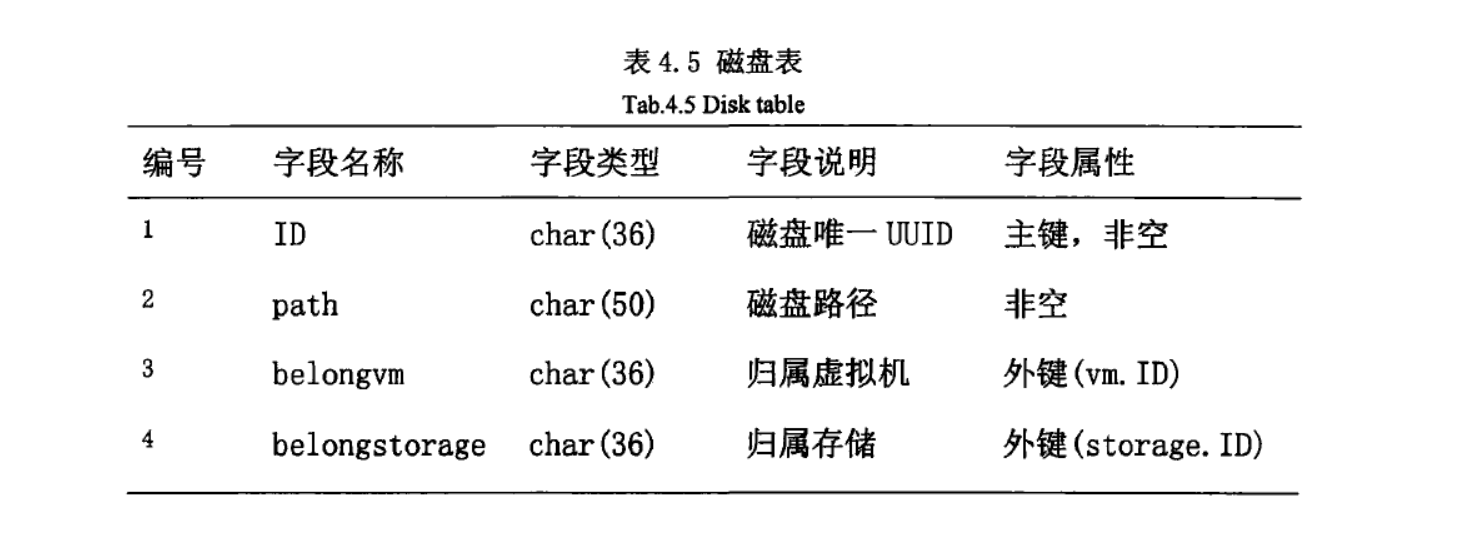
应用范围内全部的主机由主机表进行记录，其中使用36位char类型数据表示主机的唯一UUID。主机包含主机名、主机的链接地址和主机归属的集群三个属性。

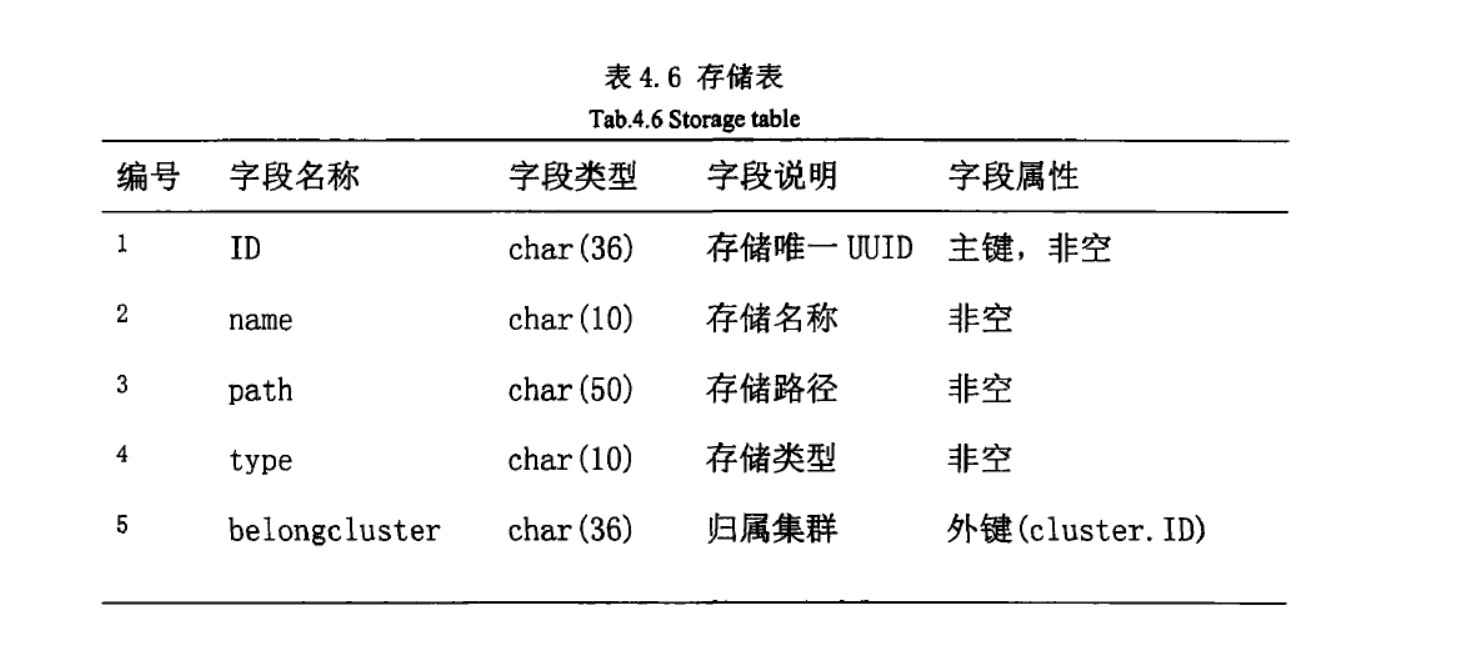
该表在新建主机时更新记录。一个主机在加入时必须指定一个已有的集群，即该主机隶属哪个集群。

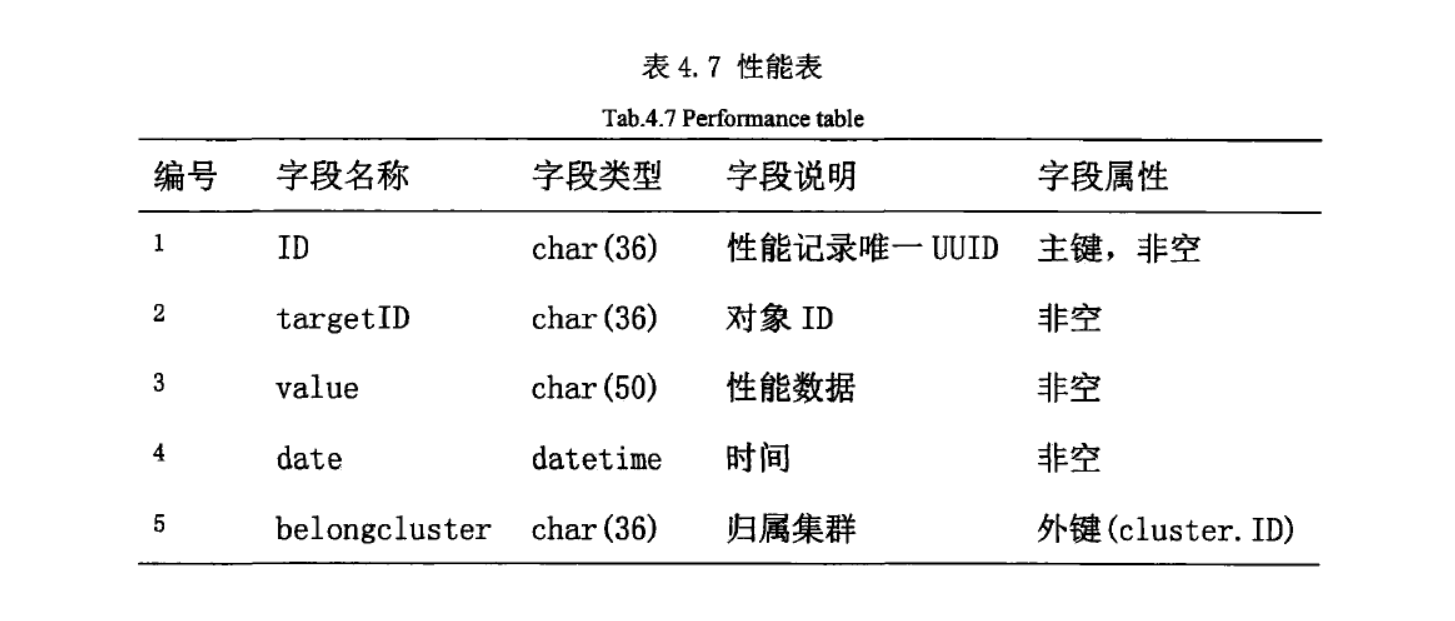
 集群表负责记录应用管理的集群。集群同样使用36位的char类型表示唯一UUID。在当前的设计中一个集群可以包含多个主机。另外集群仅仅时逻辑层面上的划分，不存在对应的实体。

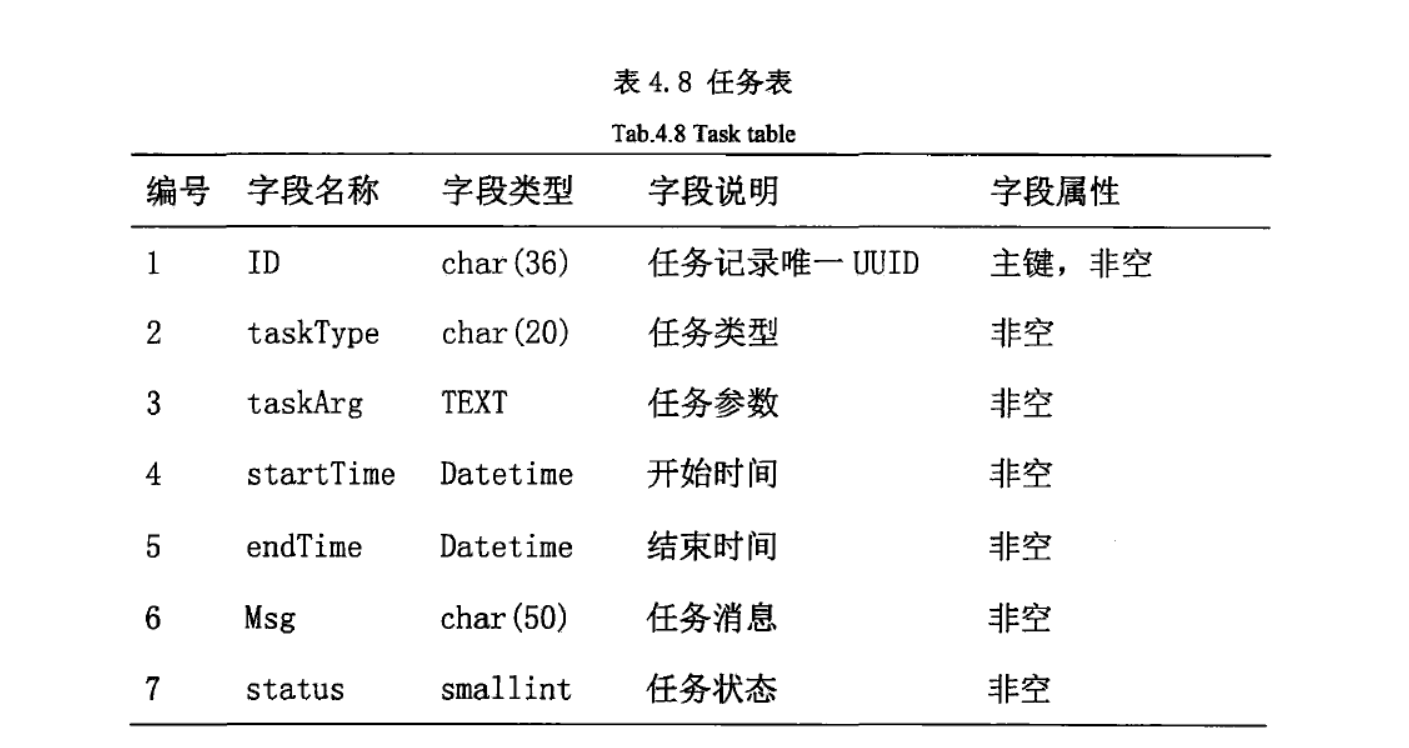
 网络表记录集群中有哪些网络，使用36位char类型表示唯一UUID。网络使用传统vlan作为一个网段的实体划分方法，一个vlan代表一个实体的隔离网络。由于要记录这个网络使用了哪个网卡所以需要nic字段来进行记录。最后使用belongcluster这个外键代表这个网络归属哪个集群。

 虚拟机负责记录应用所管理的全部虚拟机，使用36位char类型作为唯一UUID。需要记录虚拟机的属性包含虚拟机名称、虚拟机配置（内存、CPU等）、开始运行的时间和运行于哪个主机。当虚拟机运行时会根据宿主机的上报的性能数据主动更新配置和归属主机。 磁盘表记录应用所管理的全部磁盘，使用36位char类型表示唯一UUID。该表记录磁盘的实体路径、隶属哪个虚拟机和隶属于哪个存储。该表在磁盘创建或者虚拟机被删除时进行更新。

 存储表负责记录应用所管理的实体存储信息。一个实体的存储包含在这个存储所在的具体路径、这个存储的类型。逻辑层面上记录这个存储的名称和存储所在的集群。

 性能表会在应用集采运行性能数据时进行更新。tartgetID表示这条记录时记录哪个实体。由于这个实体可能是虚拟机、存储、网络所以无法使用外键进行表示。当进行记录对象查找时会使用UUID进行全表查找。数据库中仅记录性能的数据值，具体这个值如何使用以及通过什么规则使用将在应用中编写到代码中。最后为了统计集群范围内性能信息，将使用cluster外键来表示性能记录的归属集群。

 任务表时用来表示应用的任务记录，同样使用36位的char类型表示唯一UUID。每个任务无论成功或者失败，在结束时都会留有任务消息。最后任务当前状态由status字段表示。



## 本章小结

良好的数据库设计时程序健壮性基础，数据应尽可能的做到不修改或者少修改。当前设计中包含了一切程序中会设计到的实体，并针对每个实体设计了实体属性的列，设计了实体键的外键关联。

当应用程序运行时，无论管理员手工进行的变更还是由应用自动产生的变更，首先会影响到数据库中的数据。可以说对于应用来说数据库是比算法更加重要的组成部分，所有的模块均需要围绕着数据库运行。

1. 系统实现

## 系统开发策略

在开发过程中按照模块化的开发方式进行开发，根据已经设计好的接口分别进行模块内的相关设计。在大的流程方面按照数据库、被调度端、调度端的开发流程进行开发。设计过程中依据由底而上的局部开发流程依次进行编写。每完成一个部分的设计、编码后，进行局部的测试、功能验证以及debug。

## 系统硬件配置

由于应用使用分布式设计，所以硬件的配置分为管理端和受控端两部分。对于管理端主要负责接受请求、发送请求以及处理数据入库等功能，所以对实际的硬件没有太多的要求。根据等比例模型测试，规模在10台受控机的情况下使用intel core i5和4G内存即可足够应付全部的工作。对于受控端的硬件配置需要根据实际的虚拟机规模来进行确认，没有明确的配置要求。在仅运行受控端软件的前提下软件仅需要100MB的内存即可。

## 系统核心功能实现

### KVM/Libvirt调度实现

Libvirt的调度为两个部分，即调度端请求发送以及被调度端处理请求返回结果。对于调度端发送请求部分，首先调度端的调度模块会不停的轮询数据库中是否有调度任务需要处理，查询所有的任务是否有状态为未启动(此处通过ORM(Object Relational Mapping)编写时会定义多个任务状态，例如未启动的任务TASK\_STATUS\_WAITING=0)则会依次进行如下操作：

1. 信号量的获取multiprocessing.Semaphore.acquire()。
2. 改变数据库中任务状态。
3. 获取任务类型及参数。
4. 释放信号量multiprocessing.Semaphore.release()。
5. 查询对端主机状态是否可用。
6. 构建消息，发送消息。
7. 改变时数据库中对应任务的状态为(TASK\_STATUS\_SEND)。
8. 等待受控制端对任务进行确认，收到确认消息更新任务状态为（TASK\_STATUS\_ACK）。
9. 等待受控制端执行任务并返回结果，将结果中的消息更新到对应任务的msg字段中并更新任务状态为TASK\_STATUS\_DONE或者TASK\_STATUS\_FAILED。

### 锁保护功能实现

在调度的过程中或者在正常运行的过程中难免出现主机异常或者应用异常。这时需要依赖Sanlock的锁保护功能。例如当出现某个主机处于异常状态管理端与受控端甚至无法通讯时，将会有以下流程：

1. 管理端发现某一受控端异常，将这个主机状态设置为HOST\_STATUS\_PROBLEM。
2. 管理端尝试通过ssh工具重启受控端agent服务用以尝试重新获取受控机状态。
3. 如果依旧无法完成通讯，则通过IPMI接口直接与物理机控制台通讯，强行重启主机。
4. 锁管理模块构建请求寻找适宜的主机安排启动所有异常的虚拟机。
5. 受控端锁管理模块查看需要启动的虚拟机磁盘锁状态，等待原锁释放或者paxos超时。
6. 受控端获取Sanlock锁，启动虚拟机。
7. 等待管理端查询状态，将虚拟机已运行的消息发送给管理端。
8. 管理端收到消息更新对应数据库记录状态。

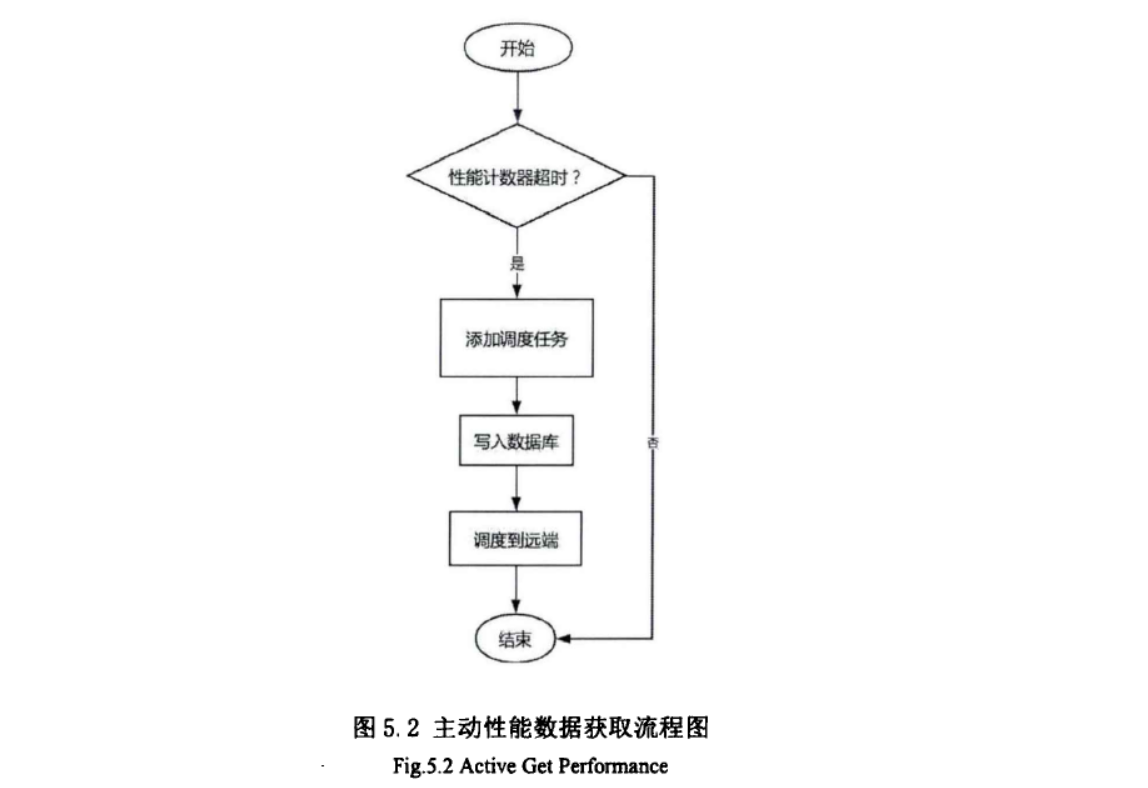
### 数据采集功能实现

数据采集分为两种方式，分别为控制端主动发送请求采集受控端状态、性能信息，另一种为受控端主动上报消息给控制端。

对于第一种主动采集方式，数据采集模块会生成定时任务轮询数据库中所有主机，生成任务类型为perf的消息，并将消息发送给受控端。经历以下流程：

1. 数据采集模块检测到某个主机的性能采集计时器超时，在数据库task表中创建一个记录。
2. 创建性能采集任务的实例并转化为JSON消息格式，设置任务状态为TASK\_STATUS\_SCHEDULING。
3. 将任务发送给目标受控端，收到受控端确认，更新数据库中的任务表项。
4. 受控端fork一个子线程来处理这个请求，收集位于本机的所有主机性能状态信息。
5. 受控端将收集到的信息同样打包为JSON格式，将消息发回控制端。
6. 控制端收到消息，整理消息入库并更新任务表项状态为TASK\_STATUS\_DONE，完成一个主动性能采集流程。

数据采集任务流程如图5.2所示。



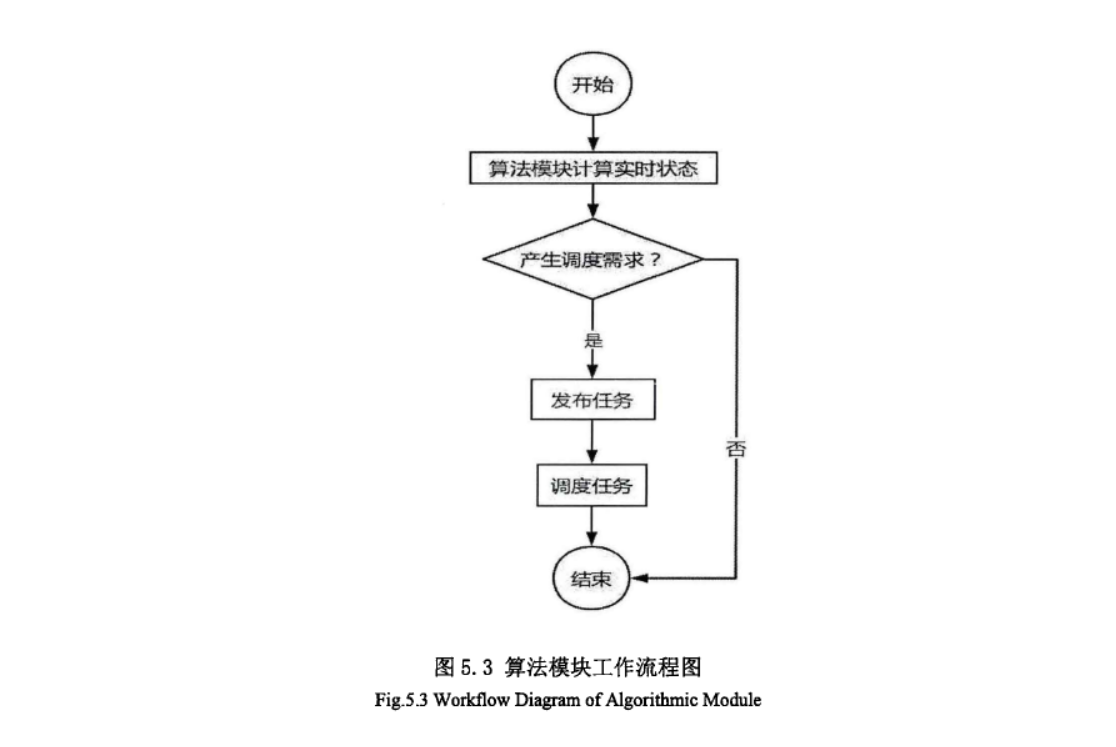
对于第二种主动上报消息的方式，一般触发于受控端某些状态改变或者某些配置改变。例如当主控端发送某个调度任务，在任务完成之后需要立即刷新数据库中的数据，这时等待第一种方式的周期轮询将等待很长的时间，可能会导致被调度模块或者算法模块拿到假数据，影响应用调度的准确性。主动发送一个性能、状态消息为解决这个问题的良好方式，流程如下。

1. 受控端完成了某个由主控端下发的任务，触发主动上传消息动作。
2. 受控端fork一个子线程收集相关性能、状态数据。
3. 受控端构建JSON消息发送给控制端。
4. 控制端收到消息，处理消息并将数据入库。

### 算法模拟实现

整体流程上，算法模块需要依赖数据采集模块的实现。在执行动作时，首先需要一些具体的算法实例，这些实例需要在运行之前编写好并在算法模块中被注册。算法不会局限与某个具体的方式，可以概括为使用现有的数据计算未来需要执行的任务，可以为FCFS、贪婪算法、轮询、遗传算法等。流程如下：

1. 算法模块根据运行配置文件选择调度算法，并实例化这个算法类。
2. 算法类开始执行分析数据库中的性能数据。
3. 算法类完成计算，产出调度对象以及调度时间和调度目标。
4. 算法模块将算法类计算出来的三个产出创建具体任务并写入数据库中等待带调度模块进行调度。
5. 调度模块开始执行调度任务，后续流程与调度模块执行一般任务完全相同。流程如5.3所示。



## 本章小结

本章介绍了全部应用的几个主要模块，各个模块之间仅通过数据库进行链接。既保证了操作的原子性也保证了唯一性。通过锁和信息量控制数据库看的读写请求解决了可能的重复操作问题。应用通过算法模块可以自动的进行调度，一定程度上实现了应用的自动化运行。同时也提供了用户的接口方便用户进行实时的控制。低耦合高内聚的编程模式为后期的功能拓展以及问题排查提供了良好的基础，可以足够使用功能的变更。