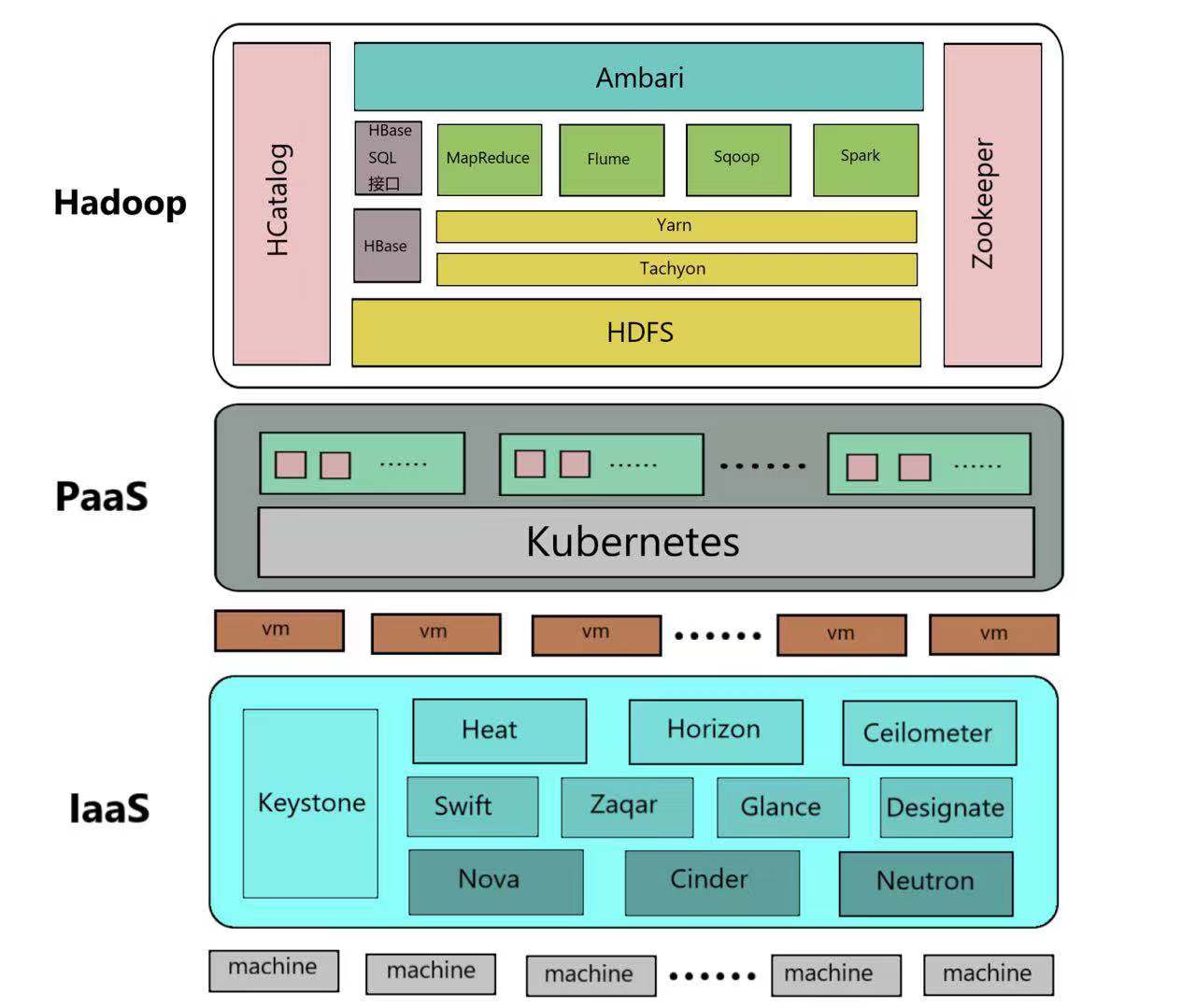
“大数据分析平台设”架构设计

# 设计方案

整体架构图如下：



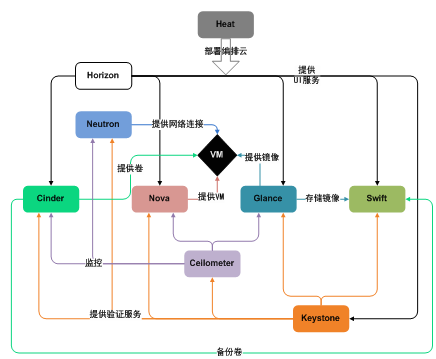
IaaS层采用OpenStack将物理机虚拟为云平台，并为上层提供虚拟机（vm），在Kubernetes调度下，将资源快速容器化，为上层Hadoop数据处理提供必要的运行环境，

Hadoop以HDFS作为存储系统，利用其合适的各种组件对现有数据进行分析。

## 底层云平台：OpenStack（IaaS） + Kubernetes + Docker（PaaS）

搭建OpenStack所需要的组件：Horizon（提供GUI）、Keystone（提供鉴权功能）、Nova（负责响应虚拟机创建请求、调度、销毁云主机）、Heat（Sahara可以配置成使用Heat来协调Hadoop集群所需要的服务）、Glance（存放Hadoop虚拟机镜像）、Swift（可以用于存放Hadoop任务处理的数据）、Cinder（用于提供块存储）、Neutron（提供网络服务）、Ceilometer（用于收集集群的信息来达到计量和监控的目的）、Sohara（负责Hadoop集群的快速创建和后期管理工作）

OpenStack各个组件之间的逻辑关系如图：



Keystone：实际上OpenStack所有的组件都依赖keystone，他主要有两个功能：

（1）用户管理：验证用户身份信息合法性

（2）服务目录管理：提供各个服务目录的（Service Catalog:包括service和endpoint）服务，无论任何服务或者客户访问openstack都要访问keystone获取服务列表，以及每个服务的endpoint

Keystone内的基本概念：

user：User即用户，指的是使用openstack service的用户，可以是人，服务，系统，就是说只要是访问openstack service 的对象都可以称为User。

Credentials：是用于确认用户身份的凭证。（用户名和密码、用户名和密钥、一个keyStone分配了身份的token）

Authentication：

（1）：用户身份验证的过程。keystone服务通过检查用户的Credentials来确定用户的身份

　　（2）：第一次验证身份是使用用户名与密码或者用户名与API Key的形式。当用户的Credentials被验证后，keystone会给用户分配一个Authentication token 供该用户的后续请求操作（返回的token中就包含User的Role列表）。

Token：

（1）是一串数字字符串，当用户访问资源时需要使用的东西，在keystone中主要是引入令牌机制来保护用户对资源的访问，同时引入PKI、PKIZ、fernet、UUID其中一个随机加密产生一串数字，对令牌加以保护

（2）token并不是长久有效的，是有时效性的，在有效的时间内可以访问资源。

Role：

（1）本身是一堆ACL集合，主要用于权限的划分。

（2）可以给User指定Role，是user获得role对应的操作权限。

（3）系统默认使用管理Role的角色 管理员用户：admin 普通用户：member（老版本） user（新版本）

（4）user验证的时候必须带有Project。老版本叫（Tenant）

Policy：

对于keystone service 来说，Policy就是一个JSON文件，rpm安装默认是在/etc/keyston/policy.json。Policy就是用来控制User对Project（tenant）中资源的操作权限。

Service：

　　　　即服务，如Nova，Glace，等各个组件

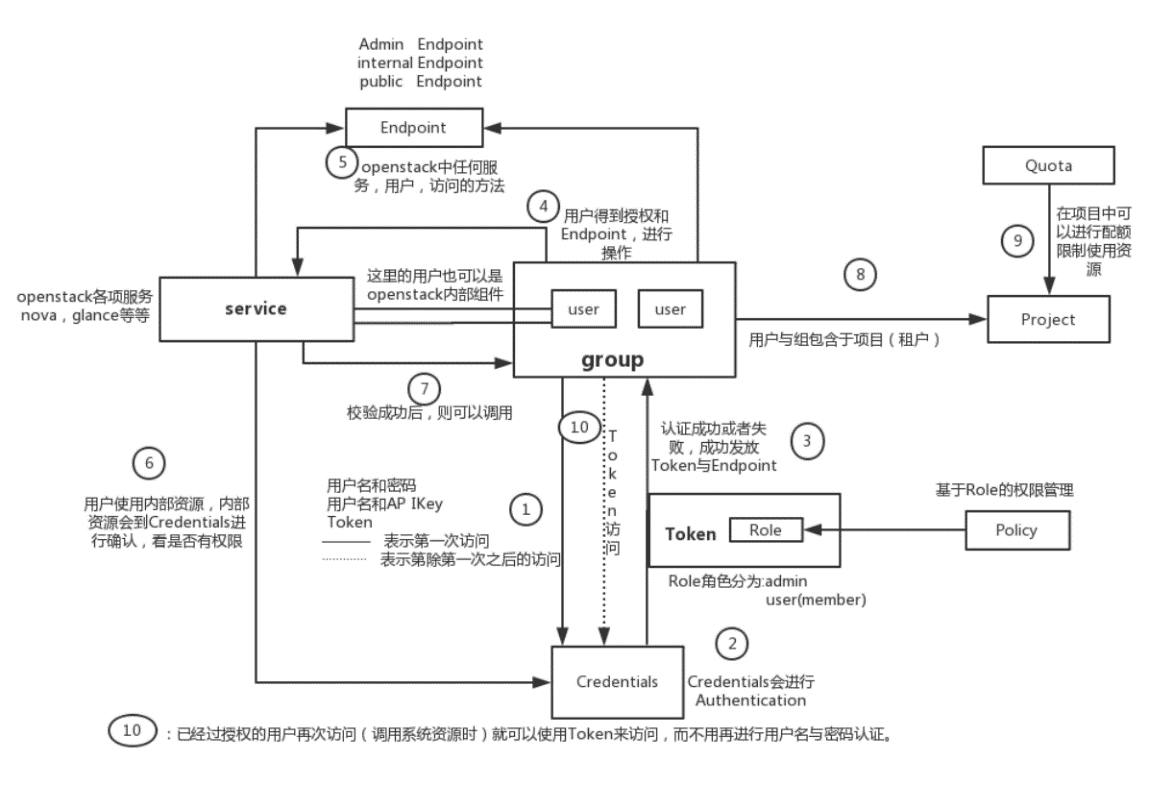
Endpoint：

（1）用来通过访问和定位某个openstack service的地址，通常是一个URL

（2）不同的region有不同的Endpoint（region使用与跨地域的云服务，比如像阿里云有华北，华东等等）。

（3）任何服务都访问openstack service中的资源时，都要访问keystone。

Keystone工作原理如图：



1.用户/API 想创建一个实例，首先会将自己的credentials发给keystone。认证成功后，keystone会颁给用户/API一个临时的令牌(Token)和一个访问服务的Endpoint。 PS:Token没有永久的

2.用户/API 把临时Token提交给keystone，keystone并返回一个Tenant(Project)

　　3.用户/API 向keystone发送带有特定租户的凭证，告诉keystone用户/API在哪个项目中，keystone收到请求后，会发送一个项目的token到用户/API  PS：第一个Token是来验证用户/API是否有权限与keystone通信，第二个Token是来验证用户/API是否有权限访问我keystone的其它服务。用户/API 拿着token和Endpoint找到可访问服务

　　4.服务向keystone进行认证，Token是否合法，它允许访问使用该服务（判断用户/API中role权限）？

　　5.keystone向服务提供额外的信息。用户/API是允许方法服务，这个Token匹配请求，这个Token是用户/API的

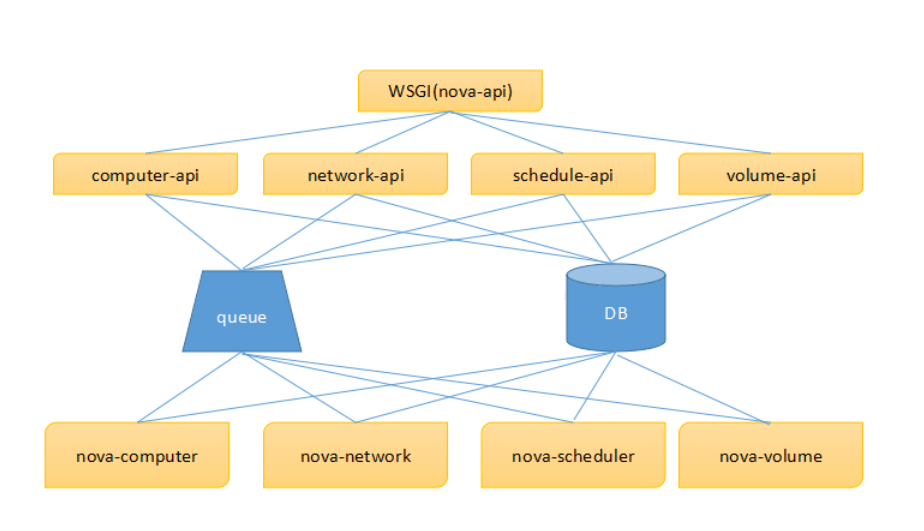
　　6.服务执行用户/API发起的请求，创建实例

　　7.服务会将状态报告给用户/API。最后返回结果，实例已经创建

##### Nova：

Nova是OpenStack云中的计算组织控制器。支持OpenStack云中实例（instances）生命周期的所有活动都由Nova处理。这样使得Nova成为一个负责管理计算资源、网络、认证、所需可扩展性的平台。

Nova的架构如下：



nova-api对外统一提供标准化接口，各子模块，如计算资源（computer），存储资源（volume）、网络资源（network）模块通过相应的api接口服务对外提供服务。

API接口操作DB实现资源数据模型的维护。

通过消息中间件，通知相应的守护进程如nova-compute实现服务接口，api与守护进程共享DB数据库，但守护进程侧重维护状态信息，网络资源状态。

守护进程之间不能直接调用，需要通过api调用，如nova-compute为虚拟机分配网络，需要调用network-api，而不能直接调用nova-network，这样易于解耦合。

Nova的调度方式：

ChanceScheduler(随机调度器)：从所有正常运行nova-compute服务的HostNode中随机选取来创建Instance。

FilterScheduler(过滤调度器)：根据指定的过滤条件以及权重来挑选最佳Instance的Host Node。

Caching(缓存调度器)：是过滤调度器的一种，在其基础上将Host资源信息缓存到本地的内存中，然后通过后台的定时任务从数据库中获取最新的Host资源信息。

上述三种调度器均继承了 /nova/scheduler/driver.py中的Scheduler类 这就是所有调度器实现都要继承的基类。因此我们可以通过继承该类并实现其中的接口，定义自己的调度器。

##### Horizon：

Horizon是openstack整个应用的一个入口，提供一个web UI 的方式来访问、控制他们的计算、存储、和网络资源：如创建和启动实例、分配IP地址等。

Horizon为两种用户提供了两种不同的功能界面：

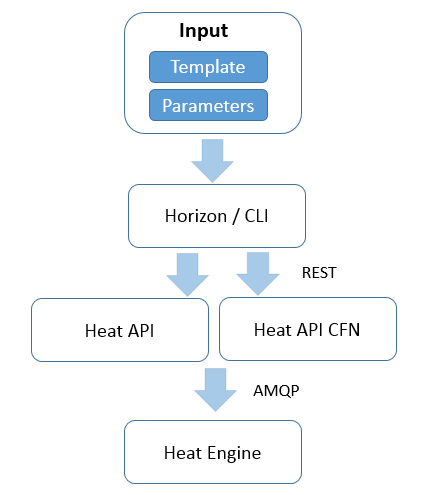
1. 云管理员：提供了一个整体的视图可以总览整个云的资源大小及运行状况，可以创建终端用户和项目，向终端用户分配项目并进行项目的资源配额管理；

2. 终端用户：提供了一个自主服务的门户，可以在管理员分配的项目中，在不超过额定配额的限制内，自由操作、使用和 存储网络资源；

##### Heat：

Heat是一套业务流程平台，旨在帮助用户更轻松地配置以OpenStack为基础的云体系。利用Heat应用程序，开发人员能够在程序中使用模板以实现资源的自动化部署。Heat能够启动应用、创建虚拟机并自动处理整个流程。它还拥有出色的[跨平台](https://www.baidu.com/s?wd=%E8%B7%A8%E5%B9%B3%E5%8F%B0&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "_blank)兼容性，能够与Amazon Web Services业务流程平台CloudFormation相对接——这意味着用户完全可以将AWS模板引入OpenStack环境当中。

Heat架构：



用户在 Horizon 中或者命令行中提交包含模板和参数输入的请求，Horizon 或者命令行工具会把请求转化为 REST 格式的 API 调用，然后调用 Heat-api 或者是 Heat-api-cfn。Heat-api 和 Heat-api-cfn 会验证模板的正确性，然后通过 AMQP 异步传递给 Heat Engine 来处理请求。

当 Heat Engine 拿到请求后，会把请求解析为各种类型的资源，每种资源都对应 OpenStack 其它的服务客户端，然后通过发送 REST 的请求给其它服务。通过如此的解析和协作，最终完成请求的处理。

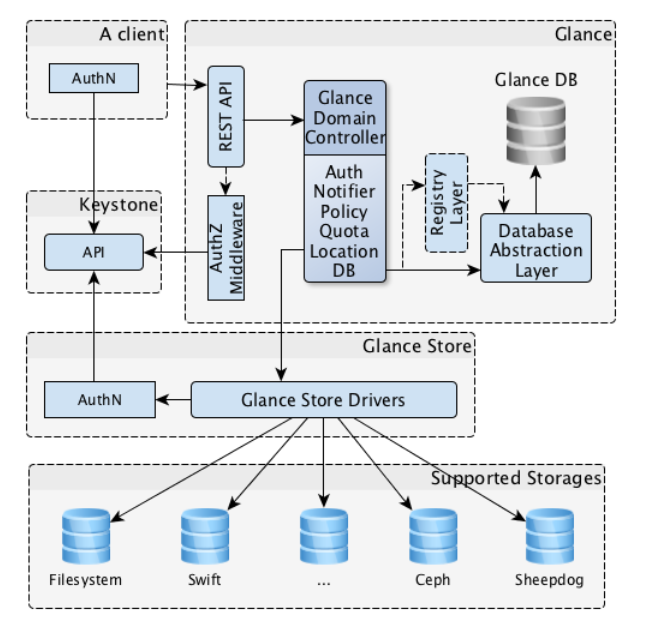
Heat Engine 在这里的作用分为三层： 第一层处理 Heat 层面的请求，就是根据模板和输入参数来创建 Stack，这里的 Stack 是由各种资源组合而成。 第二层解析 Stack 里各种资源的依赖关系，Stack 和嵌套 Stack 的关系。第三层就是根据解析出来的关系，依次调用各种服务客户段来创建各种资源。

Heat 提供了多种资源类型来支持对于软件配置和部署的编排。Heat 采用了业界流行使用的模板方式来设计或者定义编排。Heat 从四个方面来支持编排。首先是 OpenStack 自己提供的基础架构资源，包括计算，网络和存储等资源。通过编排这些资源，用户就可以得到最基本的 VM。值得提及的是，在编排 VM 的过程中，用户可以提供一些简单的脚本，以便对 VM 做一些简单的配置。然后用户可以通过 Heat 提供的 Software Configuration 和 Software Deployment 等对 VM 进行复杂的配置，比如安装软件、配置软件。接着如果用户有一些高级的功能需求，比如需要一组能够根据负荷自动伸缩的 VM 组，或者需要一组负载均衡的 VM，Heat 提供了 AutoScaling 和 Load Balance 等进行支持。如果要用户自己单独编程来完成这些功能，所花费的时间和编写的代码都是不菲的。现在通过 Heat，只需要一段长度的 Template，就可以实现这些复杂的应用。Heat 对诸如 AutoScaling 和 Load Blance 等复杂应用的支持已经非常成熟，有各种各样的模板供参考。

##### Glance

Glance主要提供了一个虚拟机镜像文件的存储、查询和检索服务，通过提供一个虚拟磁盘映像目录和存储库，为Nova的虚拟机提供镜像服务。

Glance架构如下：



Glance负责镜像文件的注册、查询和存储管理。

glance-api 负责接受Image API请求，处理image查询和存储等。

glance-registry 负责存储，处理和检索image的元数据(大小，类型等)。

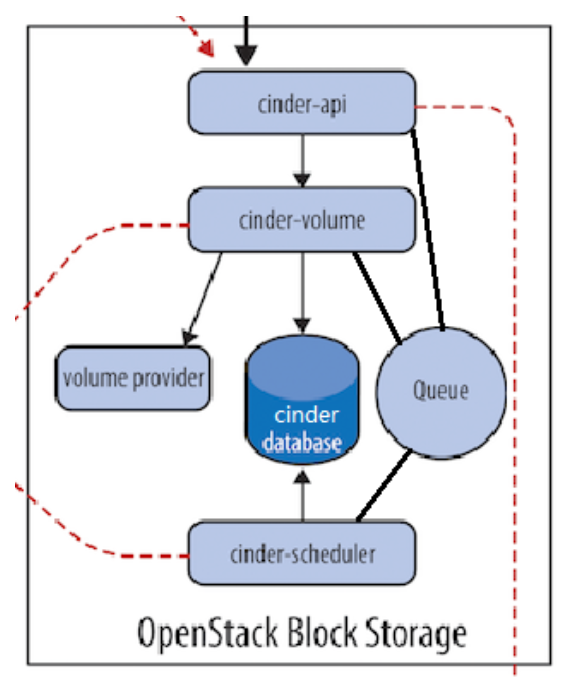
使用数据库来存储image文件的元数据。

支持不同的存储仓库来存储image文件，包括swift,本地磁盘，RADOS块设备，Amazon S3,HTTP。

##### Cinder

Cinder是一个资源管理系统，负责向虚拟机提供持久块存储资源。他能把不同的后端存储进行封装，向外提供统一的API。Cinder采用插件的方式，结合不同后端存储的驱动提供块存储服务。

Cinder架构如图：



Cinder 包含如下几个组件：

cinder-api：接收 API 请求，调用 cinder-volume 执行操作。cinder-api的作用主要是为用户提供Restful风格的接口，接收client的请求，在该服务中可以对用户的权限和传入的参数进行提前的检查，无误后方才将请求信息交给消息队列，由后续的其它服务根据消息队列信息进行处理。

cinder-volume：管理 volume 的服务，与 volume provider 协调工作，管理 volume 的生命周期。运行 cinder-volume 服务的节点被称作为存储节点。cinder-volume是部署在存储节点上的服务，cinder-volume的主要功能是对后端存储进行一层抽象封装，为用户提供统一的接口，cinder-volume通过调用后端存储驱动API来进行存储相关的操作。

cinder-scheduler

scheduler 通过调度算法选择最合适的存储节点创建 volume。

该服务中包含过滤器算法和权重计算算法，Cinder默认的过滤算法有三个：

（1）AvailabilityZoneFilter过滤算法：判断cinder host的availability zone是否与目标zone一致，否则过滤掉该节点；

（2）CapacityFilter过滤算法：判断host的可用存储空间是否不小于要分配卷的大小，否则过滤掉该节点；

（3）CapabilitiesFilter过滤算法：检查host的属性否和volume type中的extra specs相同，不相同则过滤掉该节点。通过指定的过滤算法可能会得到一系列的host，这时还需使用权重计算算法来计算各节点的权重值，权重值最大的会认为是最优节点，cinder-scheduler会基于消息队列服务的rpc调用来让最优节点对请求进行处理，以下列出几个计算权重的算法：

（1）AllocatedCapacityWeigher算法：存储空间使用最小的节点为最优节点；

（2）CapacityWeigher算法：可用存储空间最大的节点成为最优节点；

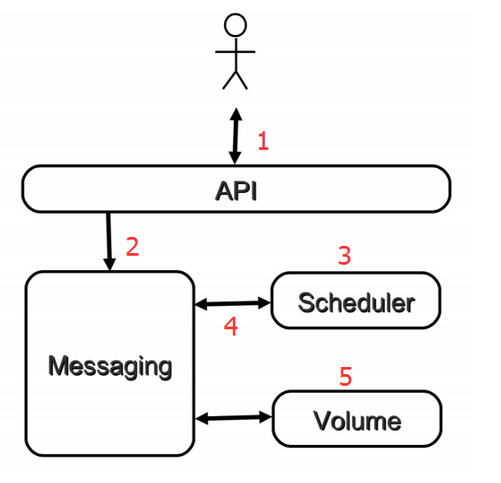
（3）ChanceWeigher算法：随机选择一个节点作为最优节点。

volume provider：数据的存储设备，为 volume 提供物理存储空间。 cinder-volume 支持多种 volume provider，每种 volume provider 通过自己的 driver 与cinder-volume 协调工作。

Message Queue：Cinder 各个子服务通过消息队列实现进程间通信和相互协作。因为有了消息队列，子服务之间实现了解耦，这种松散的结构也是分布式系统的重要特征。

Database Cinder：有一些数据需要存放到数据库中，一般使用 MySQL。数据库是安装在控制节点上的，比如在我们的实验环境中，可以访问名称为“cinder”的数据库。

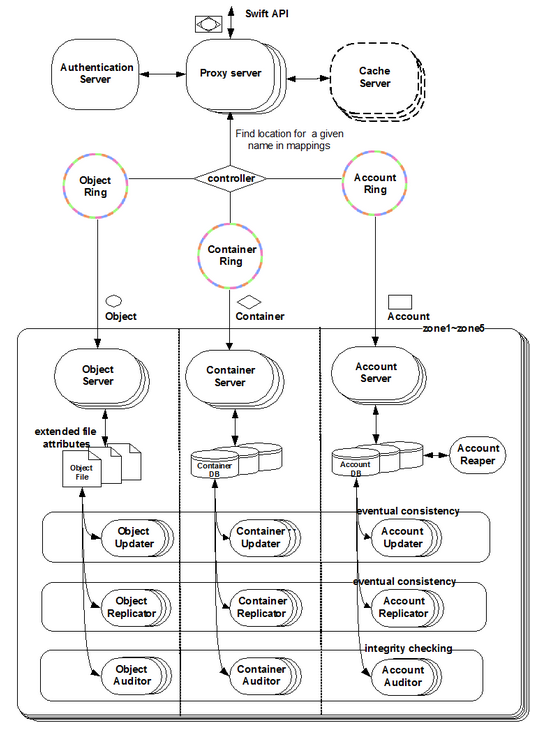
Cinder组件协同流程如图：



##### Swift

Swift 构筑在比较便宜的标准硬件存储基础设施之上，无需采用 RAID（磁盘冗余阵列），通过在软件层面引入一致性散列技术和数据冗余性，牺牲一定程度的数据一致性来达到高可用性和可伸缩性，支持多租户模式、容器和对象读写操作，适合解决互联网的应用场景下非结构化数据存储问题。

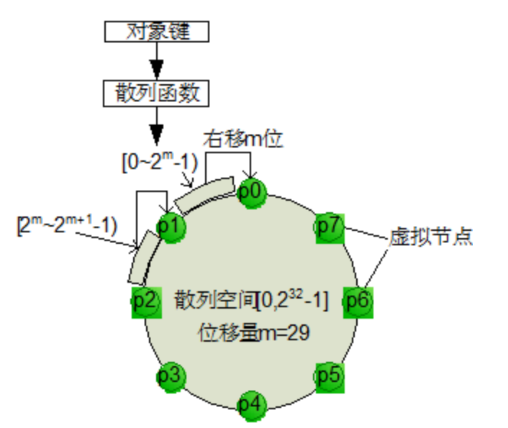
Swift系统架构如图：



Swift涉及到的技术：数据一致性模型、一致性散列、环的数据结构。

一致性散列

Swift 是基于一致性散列技术，通过计算可将对象均匀分布到虚拟空间的虚拟节点上，在增加或删除节点时可大大减少需移动的数据量；虚拟空间大小通常采用 2 的 n 次幂，便于进行高效的移位操作；然后通过独特的数据结构 Ring（环）再将虚拟节点映射到实际的物理存储设备上，完成寻址过程。



如图中所示，以逆时针方向递增的散列空间有 4 个字节长共 32 位，整数范围是[0~232-1]；将散列结果右移 m 位，可产生 232-m个虚拟节点，例如 m=29 时可产生 8 个虚拟节点。在实际部署的时候需要经过仔细计算得到合适的虚拟节点数，以达到存储空间和工作负载之间的平衡。

数据一致性模型

按照 Eric Brewer 的 CAP（Consistency，Availability，Partition Tolerance）理论，无法同时满足 3 个方面，Swift 放弃严格一致性（满足 ACID 事务级别），而采用最终一致性模型（Eventual Consistency），来达到高可用性和无限水平扩展能力。为了实现这一目标，Swift 采用 Quorum 仲裁协议(Quorum 有法定投票人数的含义)：

**（1）定义：**N：数据的副本总数；W：写操作被确认接受的副本数量；R：读操作的副本数量

**（2）强一致性：**R+W>N，以保证对副本的读写操作会产生交集，从而保证可以读取到最新版本；如果 W=N，R=1，则需要全部更新，适合大量读少量写操作场景下的强一致性；如果 R=N，W=1，则只更新一个副本，通过读取全部副本来得到最新版本，适合大量写少量读场景下的强一致性。

（3）**弱一致性**：R+W<=N，如果读写操作的副本集合不产生交集，就可能会读到脏数据；适合对一致性要求比较低的场景。

Swift 针对的是读写都比较频繁的场景，所以采用了比较折中的策略，即写操作需要满足至少一半以上成功 W >N/2，再保证读操作与写操作的副本集合至少产生一个交集，即 R+W>N。Swift 默认配置是 N=3，W=2>N/2，R=1 或 2，即每个对象会存在 3 个副本，这些副本会尽量被存储在不同区域的节点上；W=2 表示至少需要更新 2 个副本才算写成功；当 R=1 时意味着某一个读操作成功便立刻返回，此种情况下可能会读取到旧版本（弱一致性模型）；当 R=2 时，需要通过在读操作请求头中增加 x-newest=true 参数来同时读取 2 个副本的元数据信息，然后比较时间戳来确定哪个是最新版本（强一致性模型）；如果数据出现了不一致，后台服务进程会在一定时间窗口内通过检测和复制协议来完成数据同步，从而保证达到最终一致性。

环的数据结构

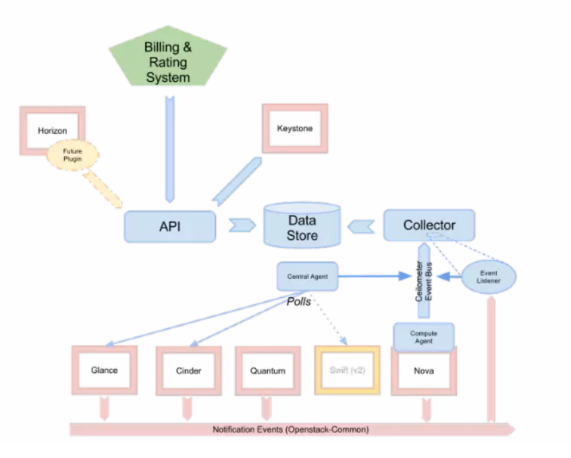
环是为了将虚拟节点（分区）映射到一组物理存储设备上，并提供一定的冗余度而设计的，其数据结构由以下信息组成：

存储设备列表、设备信息包括唯一标识号（id）、区域号（zone）、权重（weight）、IP 地址（ip）、端口（port）、设备名称（device）、元数据（meta）、分区到设备映射关系、计算分区号的位移。

##### Ceilometer

ceilometer项目是OpenStack中用来做计量计费功能的一个组件，后来又逐步发展增加了部分监控采集、告警的功能。

Ceilometer架构如图：



ceilometer 主要有下面几个概念:

meter：是ceilometer定义的监控项，诸如内存占用，网络IO，磁盘IO等等。

sample：是每个采集时间点上meter对应的值。

statistics：一般是统计学上某个周期内，meter对应的值(平均值之类)。

resource：是被监控的资源对象，这个可以是一台虚拟机，一台物理机或者一块云硬盘。

alarm：是ceilometer的告警机制，你可以通过阈值或者组合条件告警，并设置告警时触发的action。

采集机制

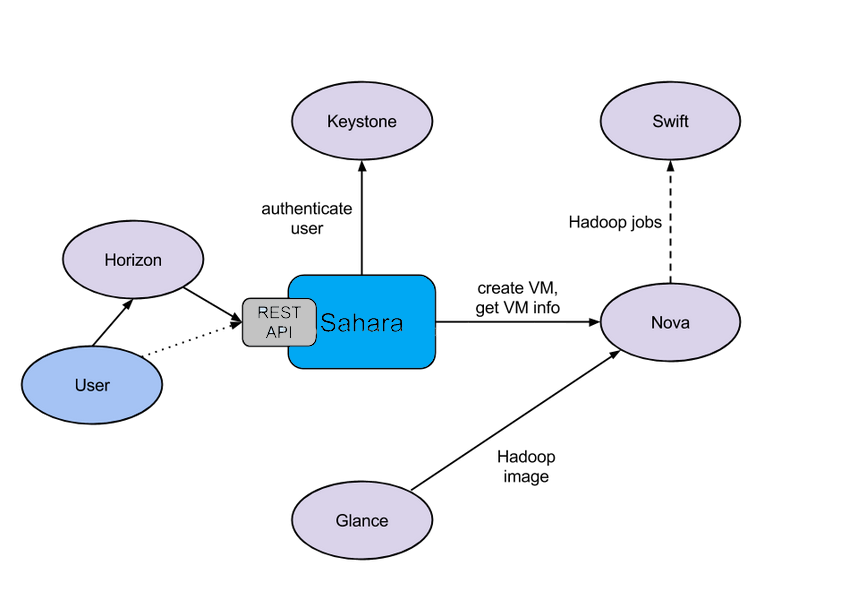
ceilometer的各个服务中，与采集相关的服务是ceilometer-collector、ceilometer-agent-central、ceilometer-agent-compute、ceilometer-agent-notification。

gent-\*服务负责采集信息，采集的信息可以通过三种方式publish出来，包括RPC、UDP、File。RPC是将采集的信息以payload方式发布到消息队列，collector服务通过监听对应的queue来收集这些信息，并保存到存储介质中；UDP通过socket创建一个UDP数据通道，然后collector通过bind这个socket来接收数据，并保存到存储介质中；File方式比较直接，就是将采集的数据以filelog的方式写入log文件中。agent-\*三个采集组件分别负责采集不同类型的信息，agent-notification负责收集各个组件推送到oslo-messaging的消息，oslo-messaging是openstack整体的消息队列框架，所有组件的消息队列都使用这个组件；agent-compute只负责收集虚拟机的CPU内存IO等信息，所以他需要安装在Hypervisor机器上；agent-central是通过各个组件API方式收集有用的信息；agent-notification只需监听AMQP中的queue即可收到信息，而agent-compute和agent-central都需要定期Poll轮询收集信息。信息通过agent-\*采集并由collector汇总处理，最终需要持久化到存储介质中。

##### Sahara：

Sahara旨在为用户提供简单部署Hadoop集群的能力，比如通过简单的配置：Hadoop版本、集群结构、节点硬件信息等。在用户提供了这些参数后，Sahara迅速把Hadoop集群部署起来。同时也支持集群的扩容和减容。

Sahara的架构如图:



Sahara架构包含一下几个模块：

鉴权模块：负责鉴权和授权，和keystone进行交换。

DAL（Data Access Layer）：和数据库访问相关。

供应引擎（Provisioning Engine）：用于和组件Nova, Heat, Cinder、Glance交换

供应商插件：插件形式提供，用于在虚拟机上配置和启动Hadoop服务。已有的解决方案包括：Apache的Ambari和Cloudera。

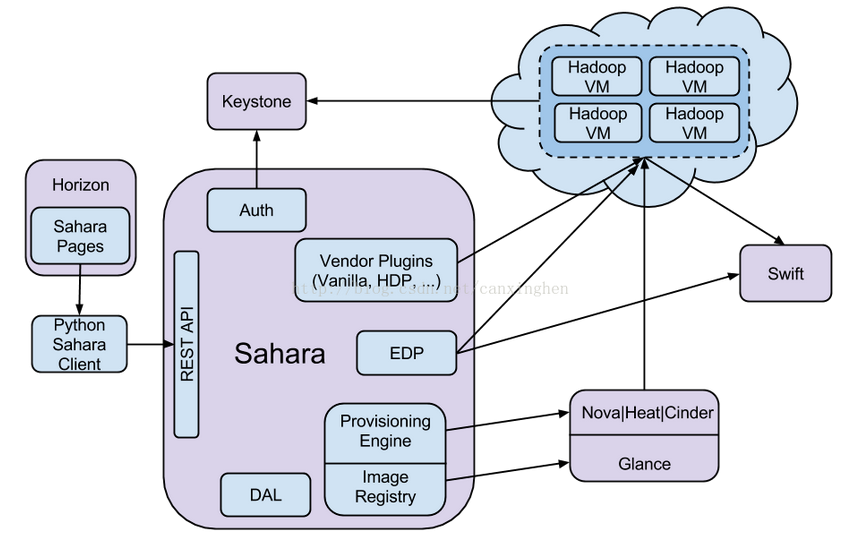
EDP（Elastic Data Processing）：负责调度和管理Sahara提供的Hadoop集群上的计算任务。

REST API：提供REST使用Sahara功能。

Sahara的Python客户端：和OpenStack其他组件的CLI一样。

Sahara的GUI页面：Horizon上提供Sahara相关的GUI。

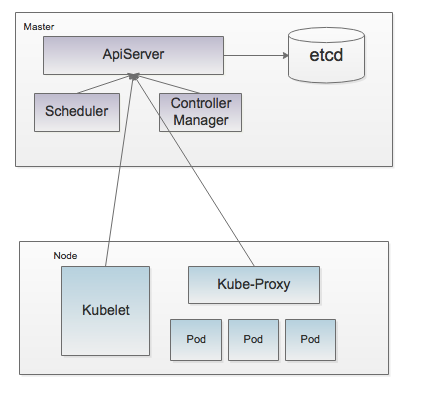
Sahara通过API快速部署Hadoop架构如图：



##### Kubernetes

Kubernetes（k8s）是自动化容器操作的开源平台，这些操作包括部署，调度和节点集群间扩展。K8s 是一个完备的分布式系统支撑平台，具有完备的集群管理能力，多层次的安全防护和准入机制、多租户应用支撑能力、透明的服务注册和发现机制、內建负载均衡器、强大的故障发现和自我修复能力、服务滚动升级和在线扩容能力、可扩展的资源自动调度机制以及多粒度的资源配额管理能力。同时 K8s 提供完善的管理工具，涵盖了包括开发、部署测试、运维监控在内的各个环节。

Kubernetes的总体架构如图：



有关Kubernetes的设计理念：API对象是K8s集群中的管理操作单元。K8s集群系统每支持一项新功能，引入一项新技术，一定会新引入对应的API对象，支持对该功能的管理操作。例如副本集Replica Set对应的API对象是RS。

每个API对象都有3大类属性：元数据metadata、规范spec和状态status。元数据是用来标识API对象的，每个对象都至少有3个元数据：namespace，name和uid；除此以外还有各种各样的标签labels用来标识和匹配不同的对象，例如用户可以用标签env来标识区分不同的服务部署环境，分别用env=dev、env=testing、env=production来标识开发、测试、生产的不同服务。规范描述了用户期望K8s集群中的分布式系统达到的理想状态（Desired State），例如用户可以通过复制控制器Replication Controller设置期望的Pod副本数为3；status描述了系统实际当前达到的状态（Status），例如系统当前实际的Pod副本数为2；那么复制控制器当前的程序逻辑就是自动启动新的Pod，争取达到副本数为3。

K8s中所有的配置都是通过API对象的spec去设置的，也就是用户通过配置系统的理想状态来改变系统，这是k8s重要设计理念之一，即所有的操作都是声明式（Declarative）的而不是命令式（Imperative）的。声明式操作在分布式系统中的好处是稳定，不怕丢操作或运行多次，例如设置副本数为3的操作运行多次也还是一个结果，而给副本数加1的操作就不是声明式的，运行多次结果就错了。

Pod

Pod是在K8s集群中运行部署应用或服务的最小单元，它是可以支持多容器的。Pod的设计理念是支持多个容器在一个Pod中共享网络地址和文件系统，可以通过进程间通信和文件共享这种简单高效的方式组合完成服务。Pod对多容器的支持是K8s最基础的设计理念。Pod是K8s集群中所有业务类型的基础，可以看作运行在K8s集群中的小机器人，不同类型的业务就需要不同类型的小机器人去执行。目前K8s中的业务主要可以分为长期伺服型（long-running）、批处理型（batch）、节点后台支撑型（node-daemon）和有状态应用型（stateful application）；分别对应的小机器人控制器为Deployment、Job、DaemonSet和PetSet。

Replication Controller

RC是K8s集群中最早的保证Pod高可用的API对象。通过监控运行中的Pod来保证集群中运行指定数目的Pod副本。指定的数目可以是多个也可以是1个；少于指定数目，RC就会启动运行新的Pod副本；多于指定数目，RC就会杀死多余的Pod副本。即使在指定数目为1的情况下，通过RC运行Pod也比直接运行Pod更明智，因为RC也可以发挥它高可用的能力，保证永远有1个Pod在运行。RC是K8s较早期的技术概念，只适用于长期伺服型的业务类型。

Replica Set

RS是新一代RC，提供同样的高可用能力，区别主要在于RS后来居上，能支持更多种类的匹配模式。副本集对象一般不单独使用，而是作为Deployment的理想状态参数使用。

Deployment

部署表示用户对K8s集群的一次更新操作。部署是一个比RS应用模式更广的API对象，可以是创建一个新的服务，更新一个新的服务，也可以是滚动升级一个服务。滚动升级一个服务，实际是创建一个新的RS，然后逐渐将新RS中副本数增加到理想状态，将旧RS中的副本数减小到0的复合操作；这样一个复合操作用一个RS是不太好描述的，所以用一个更通用的Deployment来描述。以K8s的发展方向，未来对所有长期伺服型的的业务的管理，都会通过Deployment来管理。

Service

RC、RS和Deployment只是保证了支撑服务的微服务Pod的数量，但是没有解决如何访问这些服务的问题。一个Pod只是一个运行服务的实例，随时可能在一个节点上停止，在另一个节点以一个新的IP启动一个新的Pod，因此不能以确定的IP和端口号提供服务。要稳定地提供服务需要服务发现和负载均衡能力。服务发现完成的工作，是针对客户端访问的服务，找到对应的的后端服务实例。在K8s集群中，客户端需要访问的服务就是Service对象。每个Service会对应一个集群内部有效的虚拟IP，集群内部通过虚拟IP访问一个服务。在K8s集群中微服务的负载均衡是由Kube-proxy实现的。Kube-proxy是K8s集群内部的负载均衡器。它是一个分布式代理服务器，在K8s的每个节点上都有一个；这一设计体现了它的伸缩性优势，需要访问服务的节点越多，提供负载均衡能力的Kube-proxy就越多，高可用节点也随之增多。与之相比，我们平时在服务器端做个反向代理做负载均衡，还要进一步解决反向代理的负载均衡和高可用问题。

Job

Job是K8s用来控制批处理型任务的API对象。批处理业务与长期伺服业务的主要区别是批处理业务的运行有头有尾，而长期伺服业务在用户不停止的情况下永远运行。Job管理的Pod根据用户的设置把任务成功完成就自动退出了。成功完成的标志根据不同的spec.completions策略而不同：单Pod型任务有一个Pod成功就标志完成；定数成功型任务保证有N个任务全部成功；工作队列型任务根据应用确认的全局成功而标志成功。

DaemonSet

长期伺服型和批处理型服务的核心在业务应用，可能有些节点运行多个同类业务的Pod，有些节点上又没有这类Pod运行；而后台支撑型服务的核心关注点在K8s集群中的节点（物理机或虚拟机），要保证每个节点上都有一个此类Pod运行。节点可能是所有集群节点也可能是通过nodeSelector选定的一些特定节点。典型的后台支撑型服务包括，存储，日志和监控等在每个节点上支持K8s集群运行的服务。

PetSet

PetSet是用来控制有状态服务，PetSet中的每个Pod的名字都是事先确定的，不能更改。PetSet中Pod的名字的作用，并不是《千与千寻》的人性原因，而是关联与该Pod对应的状态。

对于RC和RS中的Pod，一般不挂载存储或者挂载共享存储，保存的是所有Pod共享的状态，Pod像牲畜一样没有分别（这似乎也确实意味着失去了人性特征）；对于PetSet中的Pod，每个Pod挂载自己独立的存储，如果一个Pod出现故障，从其他节点启动一个同样名字的Pod，要挂载上原来Pod的存储继续以它的状态提供服务。

适合于PetSet的业务包括数据库服务MySQL和PostgreSQL，集群化管理服务Zookeeper、etcd等有状态服务。PetSet的另一种典型应用场景是作为一种比普通容器更稳定可靠的模拟虚拟机的机制。传统的虚拟机正是一种有状态的宠物，运维人员需要不断地维护它，容器刚开始流行时，用容器来模拟虚拟机使用，所有状态都保存在容器里，而这已被证明是非常不安全、不可靠的。使用PetSet，Pod仍然可以通过漂移到不同节点提供高可用，而存储也可以通过外挂的存储来提供高可靠性，PetSet做的只是将确定的Pod与确定的存储关联起来保证状态的连续性。

Federation

K8s的设计定位是单一集群在同一个地域内，因为同一个地区的网络性能才能满足K8s的调度和计算存储连接要求。而联合集群服务就是为提供跨Region跨服务商K8s集群服务而设计的。

每个K8s Federation有自己的分布式存储、API Server和Controller Manager。用户可以通过Federation的API Server注册该Federation的成员K8s Cluster。当用户通过Federation的API Server创建、更改API对象时，Federation API Server会在自己所有注册的子K8s Cluster都创建一份对应的API对象。在提供业务请求服务时，K8s Federation会先在自己的各个子Cluster之间做负载均衡，而对于发送到某个具体K8s Cluster的业务请求，会依照这个K8s Cluster独立提供服务时一样的调度模式去做K8s Cluster内部的负载均衡。而Cluster之间的负载均衡是通过域名服务的负载均衡来实现的。

Volume

K8s集群中的存储卷跟Docker的存储卷有些类似，只不过Docker的存储卷作用范围为一个容器，而K8s的存储卷的生命周期和作用范围是一个Pod。每个Pod中声明的存储卷由Pod中的所有容器共享。K8s支持非常多的存储卷类型，特别的，支持多种公有云平台的存储，包括AWS，Google和Azure云；支持多种分布式存储包括GlusterFS和Ceph；也支持较容易使用的主机本地目录hostPath和NFS。

Persistent Volume

存储的PV和PVC的这种关系，跟计算的Node和Pod的关系是非常类似的；PV和Node是资源的提供者，根据集群的基础设施变化而变化，由K8s集群管理员配置；而PVC和Pod是资源的使用者，根据业务服务的需求变化而变化，有K8s集群的使用者即服务的管理员来配置。

Node

K8s集群中的计算能力由Node提供，最初Node称为服务节点Minion，后来改名为Node。K8s集群中的Node也就等同于Mesos集群中的Slave节点，是所有Pod运行所在的工作主机，可以是物理机也可以是虚拟机。不论是物理机还是虚拟机，工作主机的统一特征是上面要运行kubelet管理节点上运行的容器。

Secret

Secret是用来保存和传递密码、密钥、认证凭证这些敏感信息的对象。使用Secret的好处是可以避免把敏感信息明文写在配置文件里。在K8s集群中配置和使用服务不可避免的要用到各种敏感信息实现登录、认证等功能，例如访问AWS存储的用户名密码。为了避免将类似的敏感信息明文写在所有需要使用的配置文件中，可以将这些信息存入一个Secret对象，而在配置文件中通过Secret对象引用这些敏感信息。这种方式的好处包括：意图明确，避免重复，减少暴漏机会。

User Account

用户帐户为人提供账户标识，而服务账户为计算机进程和K8s集群中运行的Pod提供账户标识。用户帐户和服务帐户的一个区别是作用范围；用户帐户对应的是人的身份，人的身份与服务的namespace无关，所以用户账户是跨namespace的；而服务帐户对应的是一个运行中程序的身份，与特定namespace是相关的。

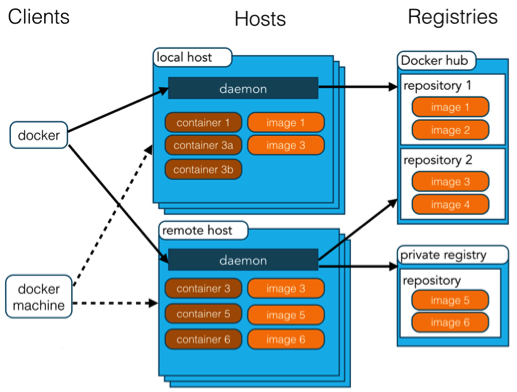
Namespace

名字空间为K8s集群提供虚拟的隔离作用，K8s集群初始有两个名字空间，分别是默认名字空间default和系统名字空间kube-system，除此以外，管理员可以可以创建新的名字空间满足需要。

##### Docker：

Docker 是一个[开源](https://baike.baidu.com/item/%E5%BC%80%E6%BA%90/246339)的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 [Linux](https://baike.baidu.com/item/Linux) 机器上，也可以实现[虚拟化](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E5%8C%96/547949)。容器是完全使用[沙箱](https://baike.baidu.com/item/%E6%B2%99%E7%AE%B1/393318)机制，相互之间不会有任何接口。

Docker架构



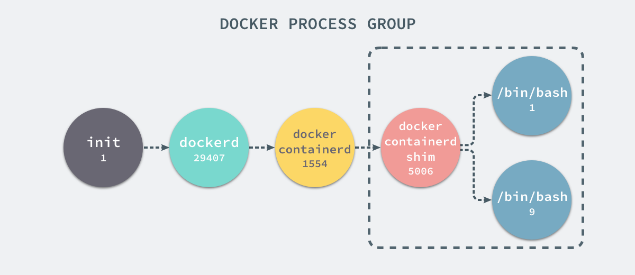
一个完整的Docker有客户端、守护进程、镜像、容器构成。Docker 使用客户端-服务器 (C/S) 架构模式，使用远程API来管理和创建Docker容器。Docker 容器通过 Docker 镜像来创建。容器与镜像的关系类似于面向对象编程中的对象与类。

Docker采用 C/S架构 Docker daemon 作为服务端接受来自客户的请求，并处理这些请求（创建、运行、分发容器）。 客户端和服务端既可以运行在一个机器上，也可通过 socket 或者RESTful API 来进行通信。

Docker daemon 一般在宿主主机后台运行，等待接收来自客户端的消息。 Docker 客户端则为用户提供一系列可执行命令，用户用这些命令实现跟 Docker daemon 交互。

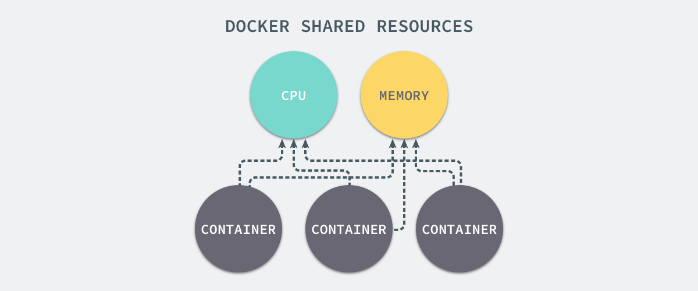
Docker所需的核心技术：NameSpace、Control Group、Advanced UnionFS

Namespace



进程是 Linux 以及现在操作系统中非常重要的概念，它表示一个正在执行的程序，也是在现代分时系统中的一个任务单元。命名空间（namespaces）是 Linux 为我们提供的用于分离进程树、网络接口、挂载点以及进程间通信等资源的方法。Docker 其实就通过 Linux 的 Namespaces 对不同的容器实现了隔离。

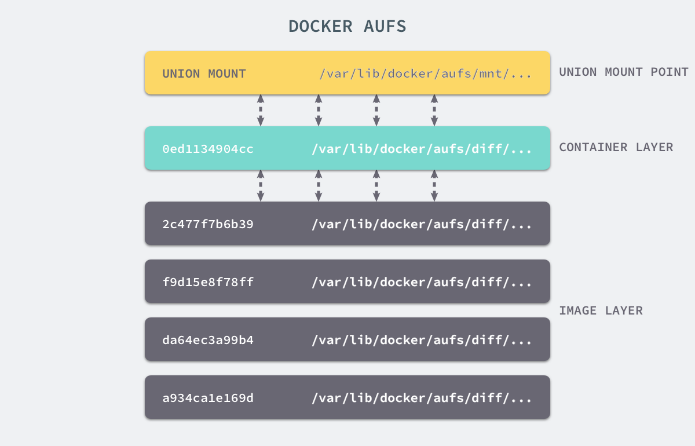
Control Group

****

通过 Linux 的命名空间为新创建的进程隔离了文件系统、网络并与宿主机器之间的进程相互隔离，但是命名空间并不能够为我们提供物理资源上的隔离，比如 CPU 或者内存，如果在同一台机器上运行了多个对彼此以及宿主机器一无所知的容器，这些容器却共同占用了宿主机器的物理资源。如果其中的某一个容器正在执行 CPU 密集型的任务，那么就会影响其他容器中任务的性能与执行效率，导致多个容器相互影响并且抢占资源。如何对多个容器的资源使用进行限制就成了解决进程虚拟资源隔离之后的主要问题，而 Control Groups（简称 CGroups）就是能够隔离宿主机器上的物理资源，例如 CPU、内存、磁盘 I/O 和网络带宽。每一个 CGroup 都是一组被相同的标准和参数限制的进程，不同的 CGroup 之间是有层级关系的，它们之间可以从父类继承一些用于限制资源使用的标准和参数。

Advanced UnionFS

解决了镜像存储隔离问题。Docker中的每一层镜像都是由一系列只读层组成，Dockerfile中的每一个命令都会在已有的只读层上创建一个新的层。当镜像被 docker run 命令创建时就会在镜像的最上层添加一个可写的层，也就是容器层，所有对于运行时容器的修改其实是对这个容器读写层的修改。



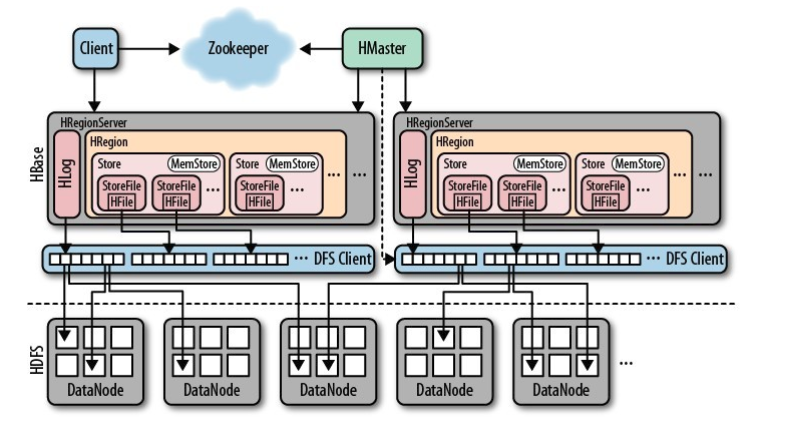
UnionFS 其实是一种为 Linux 操作系统设计的用于把多个文件系统联合到同一个挂载点的文件系统服务。每一个镜像层或者容器层都是docker 目录下的一个子文件夹；在 Docker 中，所有镜像层和容器层的内容都存储在 docker的二级子目录下（./aufs/diff/） 目录中。而 /var/lib/docker/aufs/layers/ 中存储着镜像层的元数据，每一个文件都保存着镜像层的元数据，最后的 /var/lib/docker/aufs/mnt/ 包含镜像或者容器层的挂载点，最终会被 Docker 通过联合的方式进行组装。每一个镜像层都是建立在另一个镜像层之上的，同时所有的镜像层都是只读的，只有每个容器最顶层的容器层才可以被用户直接读写，所有的容器都建立在一些底层服务（Kernel）上，包括命名空间、控制组、rootfs 等等，这种容器的组装方式提供了非常大的灵活性，只读的镜像层通过共享也能够减少磁盘的占用。

**ZooKeeper：**

ZooKeeper用于集群的管理，包括统一配置管理，选举Leader等。（后有介绍）

## 二、数据存储机制：HDFS + HBase + zookeeper

整体架构如下：

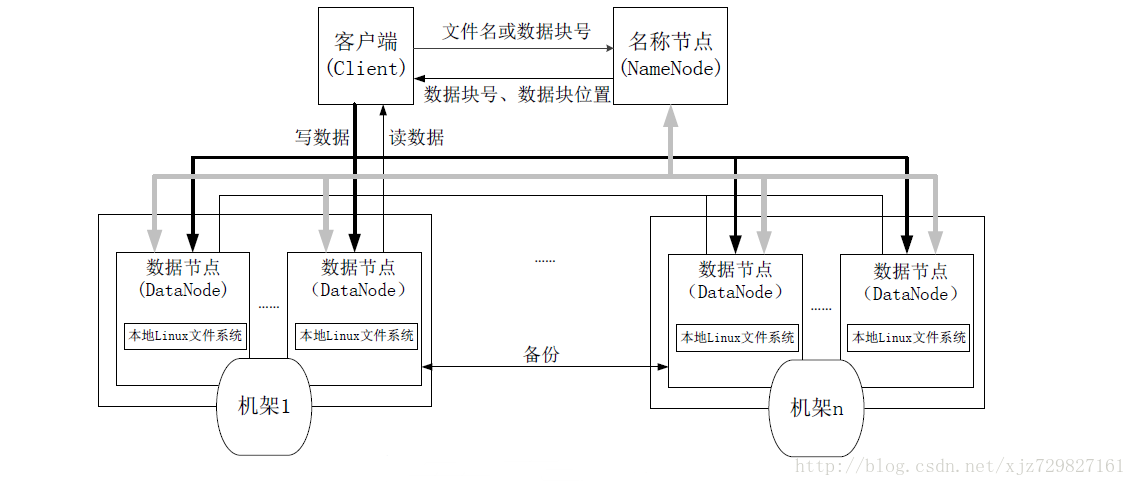


HDFS+HBase与单机中的数据库+文件系统类似，但又不完全一样。首先，HBase不实现复制机制，而是交给底下的HDFS系统，HDFS则会把一个块写入到多个数据节点中。因此，HBase更像维护表格与文件之间映射关系的一个计算引擎。

HDFS是GFS的开源实现版本，解决大规模分布式数据存储的问题。然而，HBase则是基于HDFS之上的一个分布式的、面向列的开源NoSQL数据库，解决大规模非结构化和半结构化数据存储的问题。

##### HDFS工作机制

HDFS架构图如下：



HDFS 采用Master/Slave的架构来存储数据，这种架构主要由四个部分组成，分别为HDFS Client、NameNode、DataNode和Secondary NameNode。

HDFS的文件在物理上是分块存储，默认大小为128M（hadoop2.x）。HDFS文件系统会给客户端提供一个统一的抽象目录树，客户端通过路径进行访问文件。HDFS目录结构及文件分块位置（元数据）管理由namenode节点承担，namenode是HDFS集群主节点，负责维护整个hdfs文件系统的目录树，以及每一个路径（文件）所对应的数据块信息。文件的各个block的存储管理由datanode节点承担，datanode是HDFS集群从节点，每一个block都可以在多个datanode上存储多个副本（第二份副本放在同一机架不同主机，第三份分本放在不同机架的主机上）。HDFS通过这种多副本的方式来保证数据的可靠性（未来也可以通过集成Erasure code算法来降低存储成本）。

Client：

1、文件切分。文件上传 HDFS 的时候，Client 将文件切分成 一个一个的Block，然后进行存储。

2、与 NameNode 交互，获取文件的位置信息。

3、与 DataNode 交互，读取或者写入数据。

4、Client 提供一些命令来管理 HDFS，比如启动或者关闭HDFS。

5、Client 可以通过一些命令来访问 HDFS。

NameNode：

1、管理 HDFS 的名称空间。

2、管理数据块（Block）映射信息

3、配置副本策略

4、处理客户端读写请求。

DataNode：NameNode 下达命令，DataNode 执行实际的操作。

1、存储实际的数据块。

2、执行数据块的读/写操作。

Secondary NameNode：并非 NameNode 的热备。当NameNode 挂掉的时候，它并不能马上替换 NameNode 并提供服务。

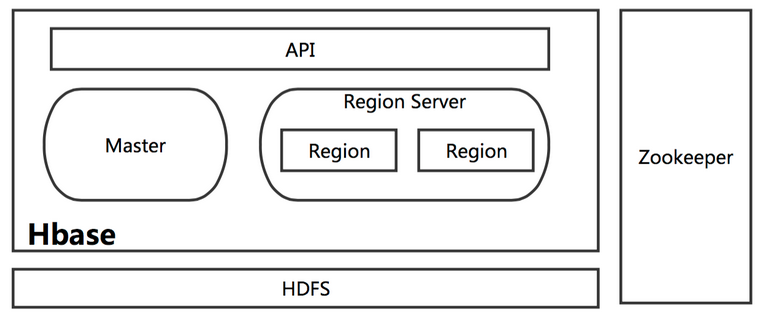
1、辅助 NameNode，分担其工作量。

2、定期合并 fsimage和fsedits，并推送给NameNode。

3、在紧急情况下，可辅助恢复 NameNode。

##### HBase工作机制

Hbase架构如图：



HBase是一个分布式的、面向列的开源数据库，它不同于一般的关系数据库,是一个适合于非结构化数据存储的数据库。另一个不同的是HBase基于列的而不是基于行的模式。HBase使用和 BigTable非常相同的数据模型。用户存储数据行在一个表里。一个数据行拥有一个可选择的键和任意数量的列，一个或多个列组成一个ColumnFamily，一个Fmaily下的列位于一个HFile中，易于缓存数据。表是疏松的存储的，因此用户可以给行定义各种不同的列。在HBase中数据按主键排序，同时表按主键划分为多个Region。

在分布式的生产环境中，HBase 需要运行在 HDFS 之上，以 HDFS 作为其基础的存储设施。HBase 上层提供了访问的数据的 Java API 层，供应用访问存储在 HBase 的数据。在 HBase 的集群中主要由 Master 和 Region Server 组成，以及 Zookeeper。

**Master：**

HBase Master用于协调多个Region Server，侦测各个RegionServer之间的状态，并平衡RegionServer之间的负载。HBaseMaster还有一个职责就是负责分配Region给RegionServer。HBase允许多个Master节点共存，但是这需要Zookeeper的帮助。不过当多个Master节点共存时，只有一个Master是提供服务的，其他的Master节点处于待命的状态。当正在工作的Master节点宕机时，其他的Master则会接管HBase的集群。

**Region Server：**

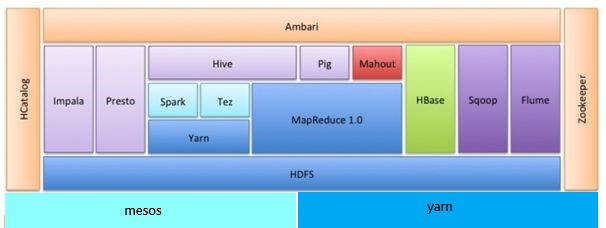
对于一个RegionServer而言，其包括了多个Region。RegionServer的作用只是管理表格，以及实现读写操作。Client直接连接RegionServer，并通信获取HBase中的数据。对于Region而言，则是真实存放HBase数据的地方，也就说Region是HBase可用性和分布式的基本单位。如果当一个表格很大，并由多个CF组成时，那么表的数据将存放在多个Region之间，并且在每个Region中会关联多个存储的单元（Store）。

**Zookeeper：**

对于 HBase 而言，Zookeeper的作用是至关重要的。首先Zookeeper是作为HBase Master的HA解决方案。也就是说，是Zookeeper保证了至少有一个HBase Master 处于运行状态。并且Zookeeper负责Region和Region Server的注册。其实Zookeeper发展到目前为止，已经成为了分布式大数据框架中容错性的标准框架。不光是HBase，几乎所有的分布式大数据相关的开源框架，都依赖于Zookeeper实现HA。

三、数据处理框架

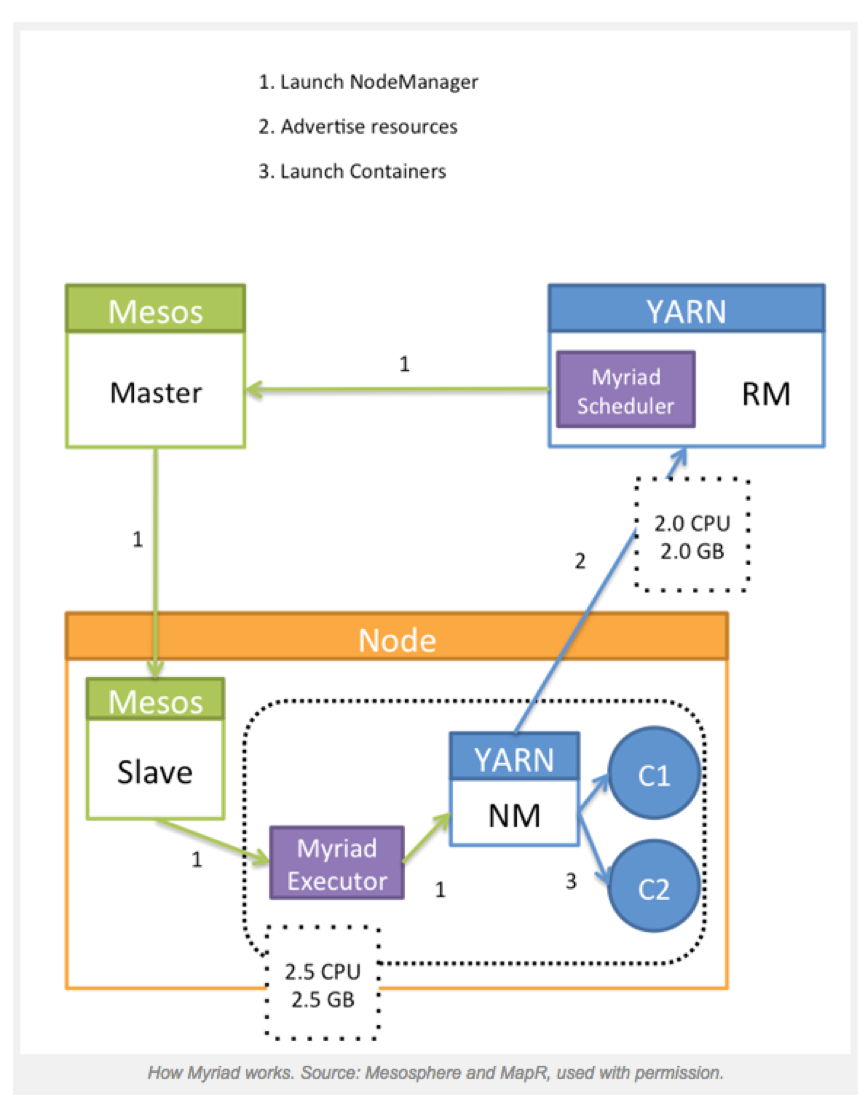
整个生态组件如图：



数据处理框架所需要的组件：Hadoop HDFS（分布式文件系统），Hadoop Yarn（Hadoop资源管理器），Spark（），Mesos（可拓展全局资源管理器），Myriad（协同Mesos和YARN），**flume（日志收集工具），pig（基于Hadoop的数据流系统），hive（基于Hadoop的数据仓库），Hbase（分布式列存数据库）**

注：使用Mesos管理平台的所有资源，同时使用YARN安全管理Hadoop任务，将集群划分为Hadoop集群和非Hadoop集群两个区域。

Mesos + Yarn+ Myriad协同管理分布式集群的架构如下：



Mesos：（1）决定云平台中哪些资源可用，将分配请求返回给一个具体的应用调度框架（Yarn）；（2）扮演仲裁者，可以在多个调度器上公平的调度资源。（3）支持Spark运行，并且比当Spark运行在YARN上时更灵活，支持两种调度方式：a.粗粒度模式，b.细粒度模式。（Spark部署在YARN上的方式，目前仅仅支持粗粒度模式，且YRAN上Container资源不可以动态伸缩，一旦启动，可使用的资源不能再发生变化。Standalone模式资源利用率低）

Hadoop Yarn：

Hadoop集群通过Yarn管理资源，提高Hadoop集群的资源利用率、方便资源统一管理和共享。

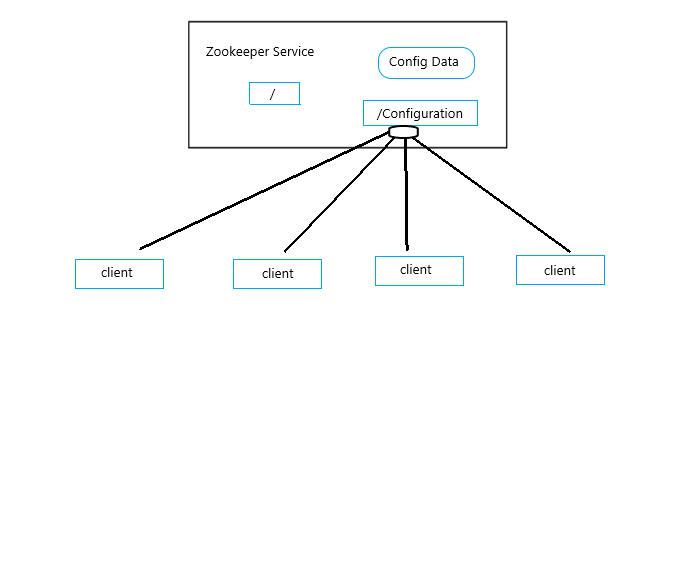
Myriad:

在同时使用Mesos和YRAN调度是，需要创建两个静态分区，当资源被Hadoop的YARN管理时，Mesos就无法起作用。Myriad可以协调Mesos和YARN。使得Mesos能够管理YARN的资源请求。当一个任务到达YARN时，它会通过Myriad调度器调度它，使请求与Mesos提供的资源匹配。相应的，Mesos也会将它传递给Mesos工作节点。之后，这个Mesos节点会把这个请求与一个正在执行YARN节点的管理器的Myriad执行器关联。Myriad在Mesos资源启动YARN节点管理器，启动之后，Mesos资源会告诉YARN资源管理器哪些资源可用。这时YARN就可以随意地使用这些资源。Myriad为Mesos的可用资源池和YARN的任务（需要用到Mesos中资源）之间架起了一座无缝连接的桥梁。数据分析可以在和运行生产服务的相同硬件上执行。你不再需要面临由静态分区引起的资源限制（和低利用率）。资源可以根据业务的需求弹性的伸缩。

**Zookeeper:**

解决分布式环境下的数据管理问题：统一命名，状态同步，集群管理，配置同步,提供集群的高可用、冷热备份等。

**Zookeeper架构配置：**



Hadoop集群将程序分布在多台机器上，为了保证集群数据的可靠性，和程序之间的协同，将所有的配置放置在Zookeeper上，保存在 Zookeeper 的某个目录节点中，然后所有相关应用程序对这个目录节点进行监听，一旦配置信息发生变化，每个应用程序就会收到 Zookeeper 的通知，然后从 Zookeeper 获取新的配置信息应用到系统中。

Zookeeper主要提供文件系统和通知机制。

**Zookeeper的工作机制**

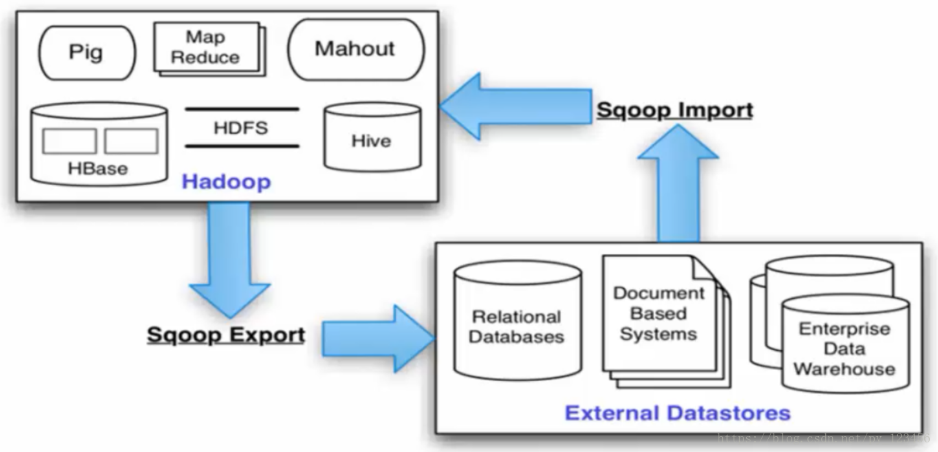
其核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协议。Zab协议有两种模式，它们分别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数Server完成了和 leader的状态同步以后，恢复模式就结束了。状态同步保证了leader和Server具有相同的系统状态。

为了保证事务的顺序一致性，zookeeper采用了递增的事务id号（zxid）来标识事务。所有的提议（proposal）都在被提出的时候加上了zxid。实现中zxid是一个64位的数字，它高32位是epoch用来标识leader关系是否改变，每次一个leader被选出来，它都会有一个新的epoch，标识当前属于那个leader的统治时期。低32位用于递增计数。

**HDFS:** HDFS是GFS的开源实现版本，解决大规模非结构化数据存储的问题。

**Hbase：HBase是Bigtable的开源实现，利用HDFS作为其文件存储系统，面向大数据场景的高可靠、高性能、高伸缩的分布式结构化存储系统。**

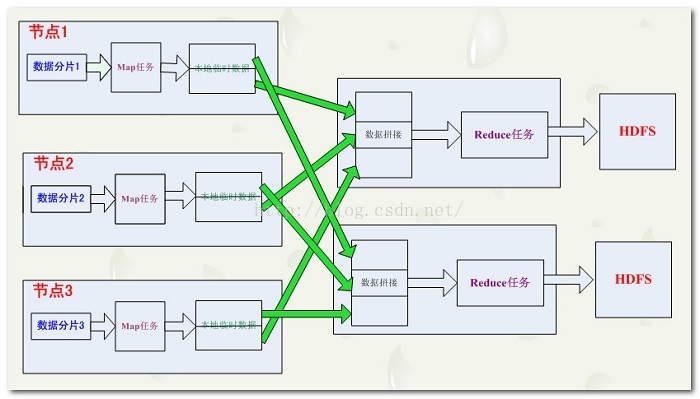
**Sqoop：主要用于传统数据库和Hadoop之间传输数据，为HBase提供了方便的RDBMS数据导入功能，方便了传统数据向HBase中迁移。并且数据的导入导出本质上时MapReduce程序，充分利用了MapReduce的并行化和容错性。**Sqoop 的核心设计思想是利用 MapReduce 加快数据传输速度。也就是说 Sqoop 的导入和导出功能是通过 MapReduce 作业实现的。所以它是一种批处理方式进行数据传输，难以实现实时的数据进行导入和导出。



MapReduce:

**Hadoop MapReduce采用了多进程模型。多进程模型便于细粒度控制每个任务占用的资源，但会消耗较多的启动时间，不适合运行低延迟类型的作业**。MapReduce是一种计算模型，用于处理大数据量的计算。其中map对应数据集上的独立元素进行指定的操作，生成键-值对形式中间，reduce则对中间结果中相同的键的所有值进行规约，以得到最终结果。但是MapReduce的计算模型框架，导致在复杂的多重处理，低延迟的交互式查询需求方面表现性不好。但是Spark弥补了MapReduce的不足。（但是pig,hive，sqoop依赖于MapReduce）。

MapReduce工作机制如图：



Spark: **Spark采用了多线程模型，**很适合运行低延迟类型的作业，利用HDFS文件系统存储数据，他可以用于存储任何兼容于Hadoop的数据源，如HDFS，HBase等。

相比较于MapReduce，Spark存在以下优点：

1.每一个作业独立调度，可以把所有的作业做一个图进行调度，各个作业之间相互依赖，在调度过程中一起调度，速度快。

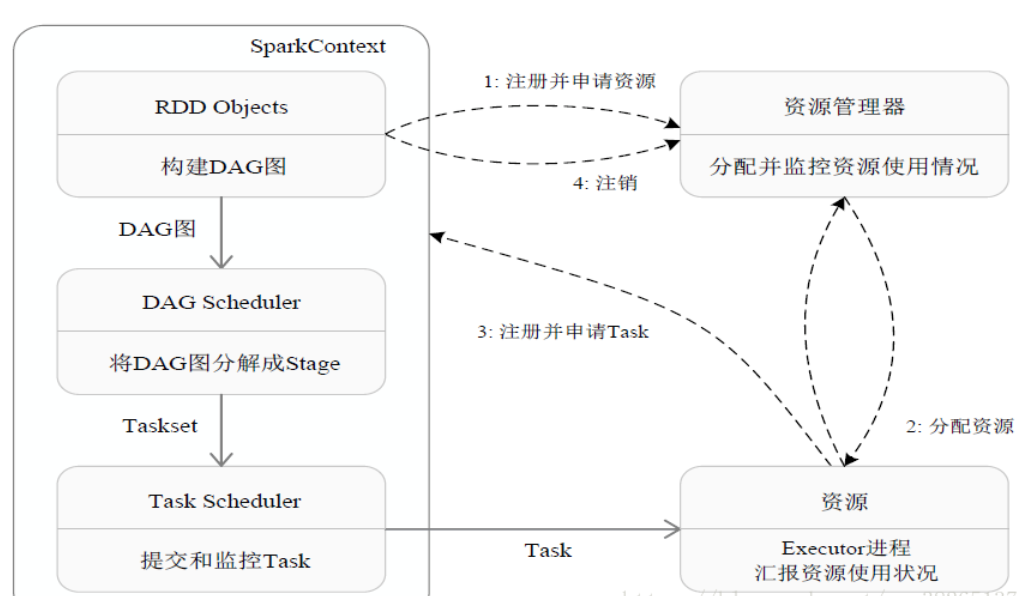
2.所有过程都基于内存，所以通常也将Spark称作是基于内存的迭代式运算框架，便于内存共享。

3.spark提供了更丰富的算子，让操作更方便。

4.更容易的API：支持Python，Scala和Java

但是Spark相比MapReduce还是有一个明显的缺点，就是内存的消耗是比较大的。在超大规模数据集离线计算并且时效性要求不高的情况下可以考虑优先使用MapReduce。

Spark运行的基本流程如下：



1、首先为应用构建起基本的运行环境，即由Driver创建一个SparkContext，进行资源的申请、任务的分配和监控。

2、资源管理器为Executor分配资源，并启动Executor进程。

3、SparkContext根据RDD的依赖关系构建DAG图，DAG图提交给DAGScheduler解析成Stage（开销最小的调度方法以及运行task的理想位置），然后把一个个TaskSet提交给底层调度器TaskScheduler处理；Executor向SparkContext申请Task，Task Scheduler将Task发放给Executor运行，并提供应用程序代码。

4、Task在Executor上运行，把执行结果反馈给TaskScheduler，然后反馈给DAGScheduler，运行完毕后写入数据并释放所有资源。

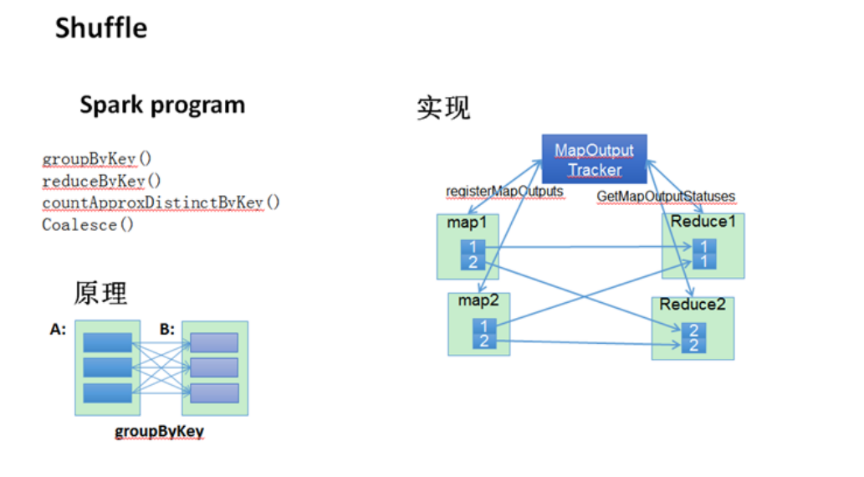
此外，DAGScheduler还处理由于shuffle数据丢失导致的失败，这有可能需要重新提交运行之前的stage（非shuffle数据丢失导致的task失败由TaskScheduler处理）。TaskScheduler维护所有TaskSet，当Executor向Driver发送心跳时，TaskScheduler会根据其资源剩余情况分配相应的Task。另外TaskScheduler还维护着所有Task的运行状态，重试失败的Task。

RDD：一个RDD就是一个分布式对象集合，本质上是一个只读的分区记录集合，其实际数据分布存储于一批机器中（内存或磁盘中）。每个RDD可分成多个分区，每个分区就是一个数据集片段，并且一个RDD的不同分区可以被保存到集群中不同的节点上，从而可以在集群中的不同节点上进行并行计算。RDD提供的转换接口都非常简单，都是类似map、filter、groupBy、join等粗粒度的数据转换操作，而不是针对某个数据项的细粒度修改。它的功能还包括容错、集合内的数据可以并行处理等。

Shuffle原理简介

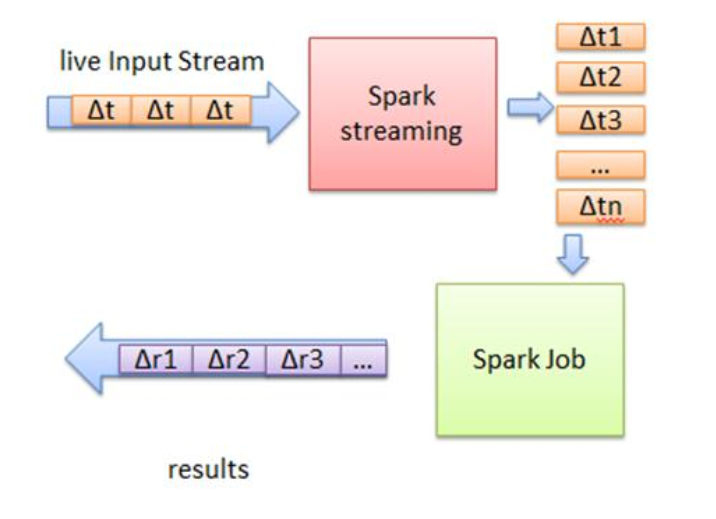
在Spark编程时，不仅仅只有reduce才会产生shuffle过程，RDD提供的groupByKey，countApproxDistinctByKey等操作都会生成shuffle。Spark中shuffle的实现与MapReduce的shuffle有比较大的差别，首先是map阶段，map的输出不再需要排序，直接写到文件中，一个map会把属于不同reduce的数据分别输出到不同的文体中，而reduce则通过aggregator处理所有shuffle fetch获取的partition。

从流程上看，MapTask结束后，Driver的MapOutPutTracker会注册MapOutPuts，ReduceTask启动后向Driver获取MapOutPutStatuses，然后fetch相应的MapOutPuts。



Spark Streaming：Spark Streaming是建立在Spark上的实时计算框架，通过它提供丰富的API、基于内存的高速执行引擎，用户可以结合流式、批处理和交互试查询应用。

Spark Streaming的基本原理是将输入数据流以时间片（秒级）为单位进行拆分，然后以类似批处理的方式处理每个时间片数据，其基本原理如下图所示。



Spark Streaming把实时输入数据流以时间片Δt （如1秒）为单位切分成块。Spark Streaming会把每块数据作为一个RDD，并使用RDD操作处理每一小块数据。每个块都会生成一个Spark Job处理，最终结果也返回多块。

**Pig：Pig是面向数据流的编程方式。（可舍弃）**

**Hive：Hive是建立在Hadoop上的数据仓库基础框架，**是一种可以存储、查询和分析存储在 Hadoop 中的大规模数据的机制。**定义了类SQL的查询语言。**适合于Hadoop数据存储的数据库。但是Hive 并不能够在大规模数据集上实现低延迟快速的查询。