**•Kubernetes 组件:**

　　Kubernetes Master控制组件，调度管理整个系统（集群），包含如下组件:

　　1.Kubernetes API Server

　　　　作为Kubernetes系统的入口，其封装了核心对象的增删改查操作，以RESTful API接口方式提供给外部客户和内部组件调用。维护的REST对象持久化到Etcd中存储。

　　2.Kubernetes Scheduler

　　　　为新建立的Pod进行节点(node)选择(即分配机器)，负责集群的资源调度。组件抽离，可以方便替换成其他调度器。

　　3.Kubernetes Controller

　　　　负责执行各种控制器，目前已经提供了很多控制器来保证Kubernetes的正常运行。

　　4. Replication Controller

　　　　管理维护Replication Controller，关联Replication Controller和Pod，保证Replication Controller定义的副本数量与实际运行Pod数量一致。

　　5. Node Controller

　　　　管理维护Node，定期检查Node的健康状态，标识出(失效|未失效)的Node节点。

　　6. Namespace Controller

　　　　管理维护Namespace，定期清理无效的Namespace，包括Namesapce下的API对象，比如Pod、Service等。

　　7. Service Controller

　　　　管理维护Service，提供负载以及服务代理。

　　8.EndPoints Controller

　　　　管理维护Endpoints，关联Service和Pod，创建Endpoints为Service的后端，当Pod发生变化时，实时更新Endpoints。

　　9. Service Account Controller

　　　　管理维护Service Account，为每个Namespace创建默认的Service Account，同时为Service Account创建Service Account Secret。

　　10. Persistent Volume Controller

　　　　管理维护Persistent Volume和Persistent Volume Claim，为新的Persistent Volume Claim分配Persistent Volume进行绑定，为释放的Persistent Volume执行清理回收。

　　11. Daemon Set Controller

　　　　管理维护Daemon Set，负责创建Daemon Pod，保证指定的Node上正常的运行Daemon Pod。

　　12. Deployment Controller

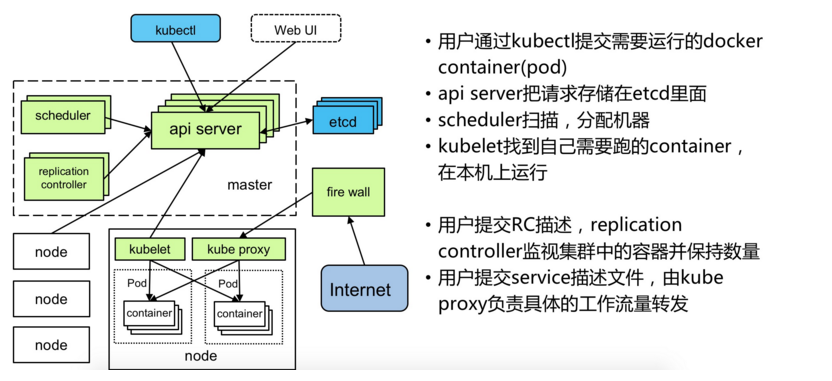
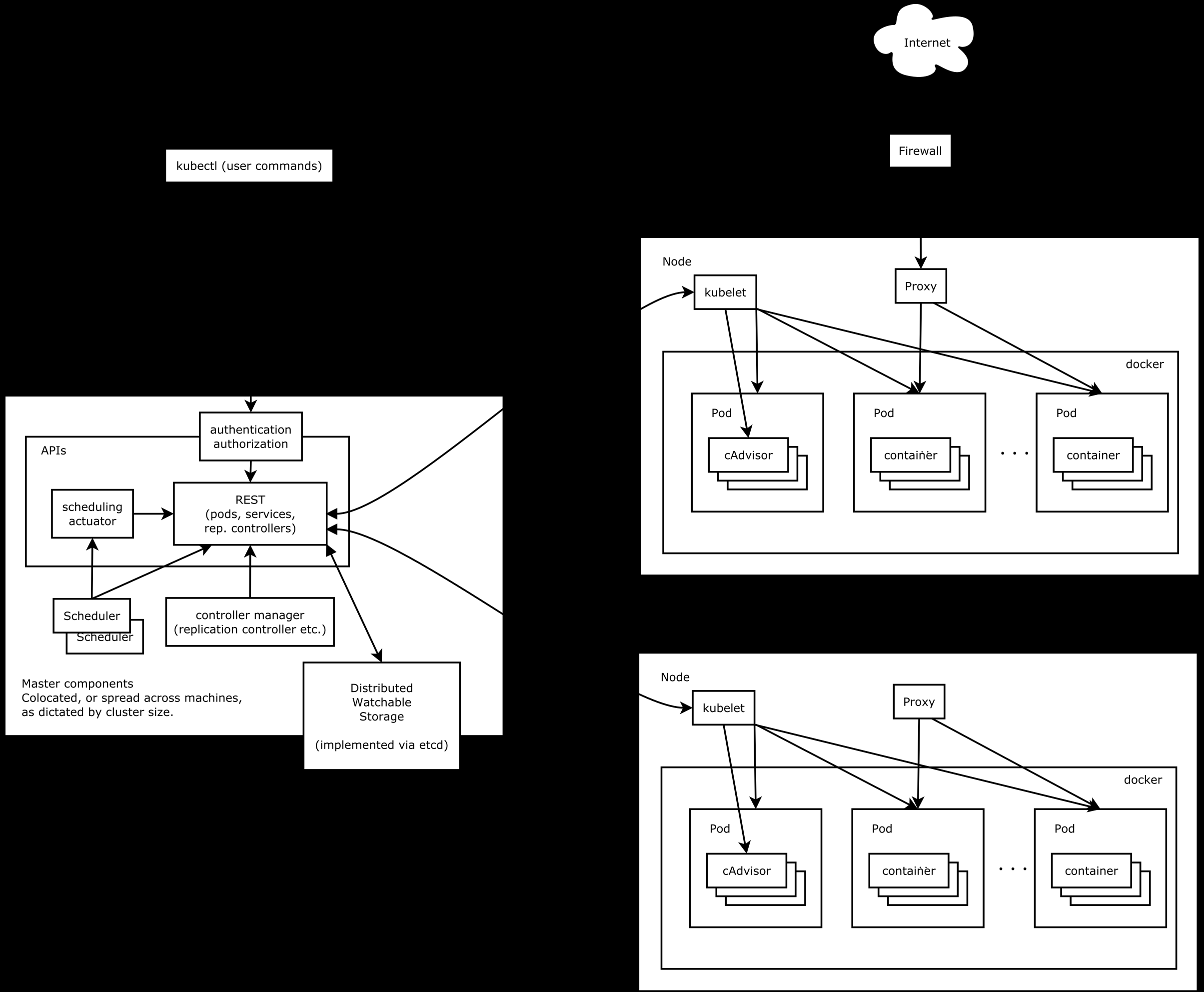
　　　　管理维护Deployment，关联Deployment和Replication Controller，保证运行指定数量的Pod。当Deployment更新时，控制实现Replication Controller和　Pod的更新。

　　13.Job Controller

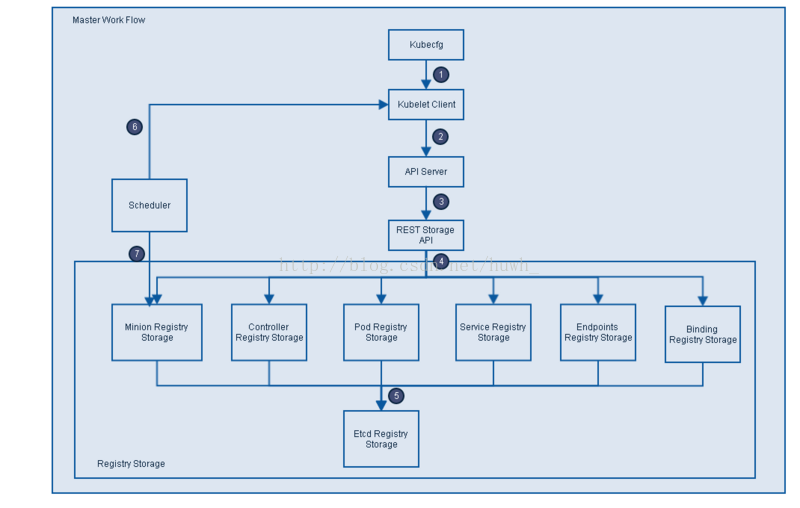
　　　　管理维护Job，为Jod创建一次性任务Pod，保证完成Job指定完成的任务数目

　　14. Pod Autoscaler Controller

　　　　实现Pod的自动伸缩，定时获取监控数据，进行策略匹配，当满足条件时执



* **master的工作流程图**

****

1. Kubecfg将特定的请求，比如创建Pod，发送给Kubernetes Client。
2. Kubernetes Client将请求发送给API server。
3. API Server根据请求的类型，比如创建Pod时storage类型是pods，然后依此选择何种REST Storage API对请求作出处理。
4. REST Storage API对的请求作相应的处理。
5. 将处理的结果存入高可用键值存储系统Etcd中。
6. 在API Server响应Kubecfg的请求后，Scheduler会根据Kubernetes Client获取集群中运行Pod及Minion/Node信息。
7. 依据从Kubernetes Client获取的信息，Scheduler将未分发的Pod分发到可用的Minion/Node节点上。

#### API Server[资源操作入口]

1. 提供了资源对象的唯一操作入口，其他所有组件都必须通过它提供的API来操作资源数据，只有API Server与存储通信，其他模块通过API Server访问集群状态。
   1. 第一，是为了保证集群状态访问的安全。
   2. 第二，是为了隔离集群状态访问的方式和后端存储实现的方式：API Server是状态访问的方式，不会因为后端存储技术etcd的改变而改变。
2. 作为kubernetes系统的入口，封装了核心对象的增删改查操作，以RESTFul接口方式提供给外部客户和内部组件调用。对相关的资源数据“全量查询”+“变化监听”，实时完成相关的业务功能。

#### 2、Controller Manager[内部管理控制中心]

1. 实现集群故障检测和恢复的自动化工作，负责执行各种控制器，主要有：
   * endpoint-controller：定期关联service和pod(关联信息由endpoint对象维护)，保证service到pod的映射总是最新的。
   * replication-controller：定期关联replicationController和pod，保证replicationController定义的复制数量与实际运行pod的数量总是一致的。

#### 3、Scheduler[集群分发调度器]

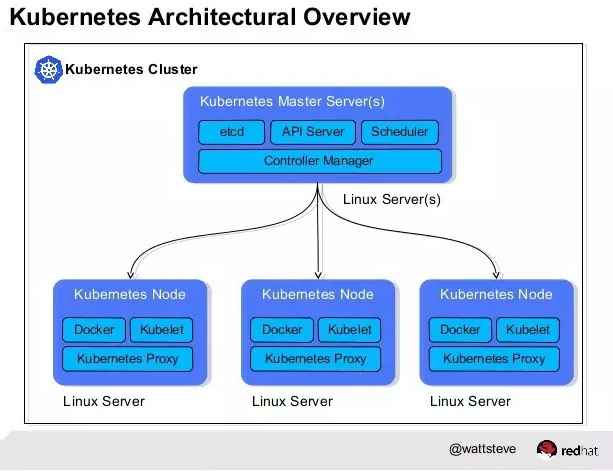
1. Scheduler收集和分析当前Kubernetes集群中所有Minion节点的资源(内存、CPU)负载情况，然后依此分发新建的Pod到Kubernetes集群中可用的节点。
2. 实时监测Kubernetes集群中未分发和已分发的所有运行的Pod。
3. Scheduler也监测Minion节点信息，由于会频繁查找Minion节点，Scheduler会缓存一份最新的信息在本地。
4. 最后，Scheduler在分发Pod到指定的Minion节点后，会把Pod相关的信息Binding写回API Server。

Master节点上面主要由四个模块组成：APIServer、scheduler、controller manager、etcd。

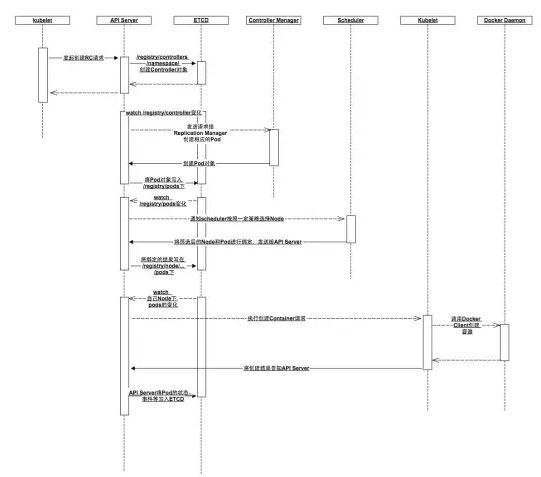
1. **APIServer**。APIServer的功能如其名，负责对外提供RESTful的Kubernetes API服务，它是系统管理指令的统一入口，任何对资源进行增删改查的操作都要交给APIServer处理后再提交给etcd。如架构图中所示，kubectl（Kubernetes提供的客户端工具，该工具内部就是对Kubernetes API的调用）是直接和APIServer交互的。
2. **schedule**。scheduler的职责很明确，就是负责调度pod（Kubernetes中最小的调度单元，后面马上就会介绍）到合适的Node上。如果把scheduler看成一个黑匣子，那么它的输入是pod和由多个Node组成的列表，输出是Pod和一个Node的绑定（bind），即将这个pod部署到这个Node上。虽然scheduler的职责很简单，但我们知道调度系统的智能程度对于整个集群是非常重要的。Kubernetes目前提供了调度算法，但是同样也保留了接口，用户可以根据自己的需求定义自己的调度算法。
3. **controller manager**。如果说APIServer做的是“前台”的工作的话，那controller manager就是负责“后台”的。后面我们马上会介绍到资源，每个资源一般都对应有一个控制器，而controller manager就是负责管理这些控制器的。还是举个例子来说明吧：比如我们通过APIServer创建一个pod，当这个pod创建成功后，APIServer的任务就算完成了。而后面保证Pod的状态始终和我们预期的一样的重任就由controller manager去保证了。
4. **etcd**。etcd是一个高可用的键值存储系统，Kubernetes使用它来存储各个资源的状态，从而实现了Restful的API。

至此，Kubernetes Master就简单介绍完了。当然，每个模块内部的实现都很复杂，而且功能也比较复杂，我现在也只是比较浅的了解了一下。如果后续了解的比较清楚了，再做总结分享。

### Node



下图描述了创建一个Replication Controller对象，各个组件之间通信的顺序图：



核心组件

下面先大概介绍一下Kubernetes的核心组件的功能：

* API Server: 提供了资源对象的唯一操作入口，其他所有的组件都必须通过它提供的API来操作资源对象。它以RESTful风格的API对外提供接口。所有Kubernetes资源对象的生命周期维护都是通过调用API Server的接口来完成，例如，用户通过kubectl创建一个Pod，即是通过调用API Server的接口创建一个Pod对象，并储存在ETCD集群中。
* Controller Manager: 集群内部的管理控制中心，主要目的是实现Kubernetes集群的故障检测和自动恢复等工作。它包含两个核心组件：Node Controller和Replication Controller。其中Node Controller负责计算节点的加入和退出，可以通过Node Controller实现计算节点的扩容和缩容。Replication Controller用于Kubernetes资源对象RC的管理，应用的扩容、缩容以及滚动升级都是有Replication Controller来实现。
* Scheduler: 集群中的调度器，负责Pod在集群的中的调度和分配。
* Kubelet: 负责本Node节点上的Pod的创建、修改、监控、删除等Pod的全生命周期管理，Kubelet实时向API Server发送所在计算节点（Node）的信息。
* Kube-Proxy: 实现Service的抽象，为一组Pod抽象的服务（Service）提供统一接口并提供负载均衡功能。

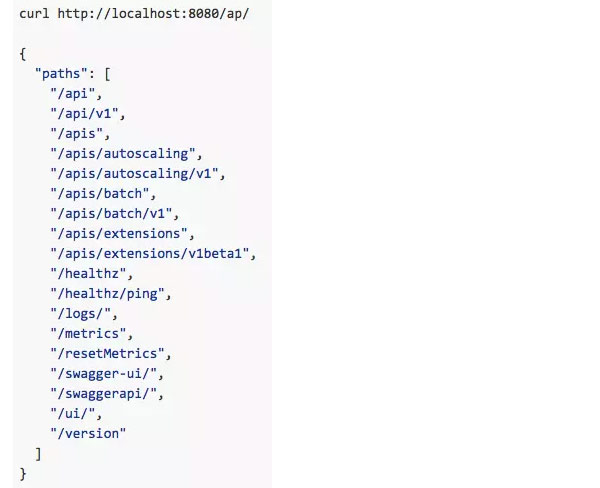
核心原理

##### API Server

如何访问Kubernetes API，Kubernetes API通过一个kube-apiserver的进程提供服务，该进程运行在Kubernetes Master节点上，默认的情况下，监听两个端口：

* 本地端口: 默认值为8080，用于接收HTTP请求，非认证授权的HTTP请求通过该端口访问API Server。
* 安全端口：默认值为6443，用于接收HTTPS请求，用于基于Token文件或者客户端证书及HTTP Base的认证，用于基于策略的授权，Kubernetes默认情况下不启动HTTPS安全访问机制。

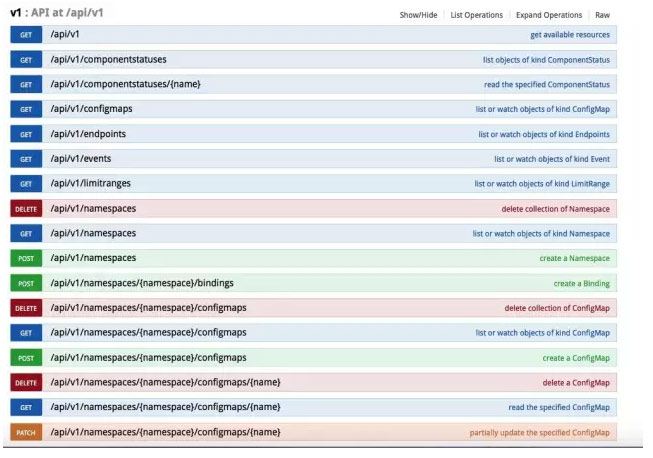
用户可以通过编程方式访问API接口，也可以通过curl命令来直接访问它，例如，我们在Master节点上访问API Server：



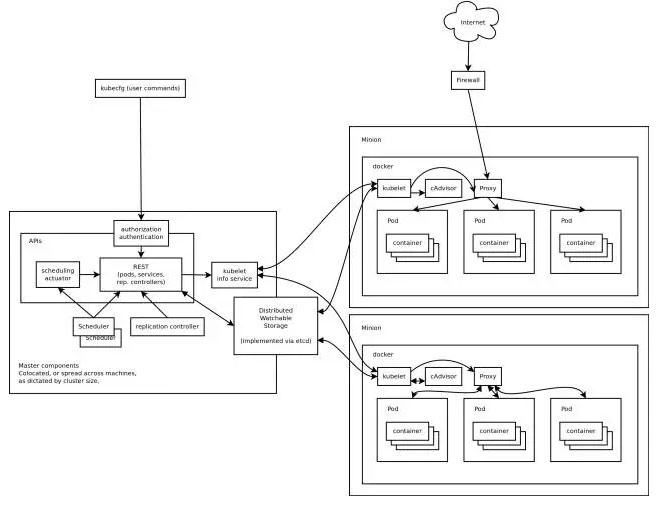
Kubernetes还提供了一个代理程序——Kubectl Proxy，它既能作为API Server的反向代理，也能作为普通客户端访问API Server，使用方法如下：



Kubernetes提供了API访问说明，您可以通过浏览器访问http://${KUBE-MASTER}:8080/swagger\_ui/来查看RESTful风格API访问规约，如图2-1所示：



##### 集群功能模块之间的通信



从图中可以看出，API Server是整个集群的核心，负责集群各个模块之间的通信。集群内部的功能模块通过API Server将信息存入ETCD，其他模块通过API Server读取这些信息，从而实现各模块之间的信息交互。比如，Node节点上的Kubelet每个一个时间周期，通过API Server报告自身状态，API Server接收这些信息后，将节点状态信息保存到ETCd中。Controller Manager中的Node Controller通过API Server定期读取这些节点状态信息，并做相应处理。Scheduler监听到某个Pod创建的信息后，检索所有符合该Pod要求的节点列表，并将Pod绑定到节点李彪中最符合要求的节点上：如果Scheduler监听到某个Pod被删除，则删除本节点上的相应Pod实例。

从上面的通信过程可以看出，API Server的访问压力很大，这也是限制（制约）Kubernetes集群规模的关键，缓解API Server的压力可以通过缓存来实现，通过watch/list操作，将资源对象的信息缓存到本地，这种方法在一定程度上缓解了API Server的压力，但是不是最好的解决办法。

##### Controller Manager

Controller Manager作为集群的内部管理控制中心，负责集群内的Node，Pod，RC，服务端点（Endpoint），命名空间（Namespace），服务账号（ServiceAccount）、资源配额（ResourceQuota）等的管理并执行自动化修复流程，确保集群出处于预期的工作状态，比如，RC实现自动控制Pod活跃副本数，如果Pod出错退出，RC自动创建一个新的Pod，来保持活跃的Pod的个数。

Controller Manager包含Replication Controller、Node Controller、ResourceQuota Controller、Namespace Controller、ServiceAccount Controller、Token Controller、Server Controller以及Endpoint Controller等多个控制器，Controller Manager是这些Controller的管理者。我们会在以后的文章中深入介绍这些Controller。

##### Scheduler

Kubernetes Scheduler负责Pod的调度管理，它负责将要创建的Pod按照一定的规则分配在某个适合的Node上。

Scheduler的默认调度流程分为以下两步：

* 预选调度过程，即遍历所有目标Node，筛选出符合要求的候选节点。为此，Kubernetes内置了多种预选策略供用户选择。
* 确定最优节点，在第一步的基础上，采用优选策略为每个候选节点打分，分值最高的胜出。

Scheduler的调度流程是通过插件方式加载“调度算法提供者”具体实现的，一个调度算法提供者其实就是包括了一组预选策略与一组有限选择策略的结构体，注册算法插件的函数如下：

20160721115241

它包含三个参数：“name string”参数为算法名，“predicateKeys”为为算法用到的预选策略集合，”priorityKeys”为算法用到的优选策略集合。

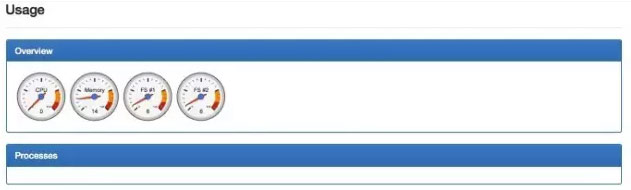
Scheduler中可用的预算策略包含：NoDiskConflict,  PodFitResources, PodSelectorMatches,  PodFitHost,  CheckNodeLabelPresence,  CheckServiceAffinity和PodFitsPorts策略等。其默认的AlgorithmProvider加载的预选策略Predicates包括：PodFitsPorts,  PodFitsResources,  NoDiskConflict,  MatchNodeSelector和HostName，即每个节点只有通过前面的五个默认预选策略后，才能初步被选中，进入下一个流程。

##### Kubelet

在Kubernetes集群中，每个计算节点（Node）上会运行一个守护进程：Kubelet。它用于处理Master节点下发到本节点的任务，管理Pod以及Pod中的容器。每个Kubelet进程会在API Server上注册自身节点的信息，定期向API Server汇报节点资源的使用情况，并通过cAdvise监控容器和节点资源。

Kubelet主要功能：

* 节点管理：kubelet可以自动向API Server注册自己，它可以采集所在计算节点的资源信息和使用情况并提交给API Server，通过启动/停止kubelet进程来实现计算节点的扩容、缩容。
* Pod管理：kubelet通过API Server监听ETCD目录，同步Pod清单，当发现有新的Pod绑定到所在的节点，则按照Pod清单的要求创建改清单。如果发现本地的Pod被删除，则kubelet通过docker client删除该容器。
* 健康检查：Pod通过两类探针来检查容器的健康状态。一个是LivenessProbe探针，用于判断容器是否健康，如果LivenessProbe探针探测到容器不健康，则kubelet将删除该容器，并根据容器的重启策略做相应的处理。另一类是ReadnessProbe探针，用于判断容器是否启动完成，且准备接受请求，如果ReadnessProbe探针检测到失败，则Pod的状态被修改。Enpoint Controller将从Service的Endpoint中删除包含该容器的IP地址的Endpoint条目。kubelet定期调用LivenessProbe探针来诊断容器的健康状况，它目前支持三种探测：HTTP的方式发送GET请求; TCP方式执行Connect目的端口; Exec的方式，执行一个脚本。
* cAdvisor资源监控: 在Kubernetes集群中，应用程序的执行情况可以在不同的级别上检测到，这些级别包含Container，Pod，Service和整个集群。作为Kubernetes集群的一部分，Kubernetes希望提供给用户各个级别的资源使用信息，这将使用户能够更加深入地了解应用的执行情况，并找到可能的瓶颈。Heapster项目为Kubernetes提供了一个基本的监控平台，他是集群级别的监控和事件数据集成器。Heapster通过收集所有节点的资源使用情况，将监控信息实时推送至一个可配置的后端，用于存储和可视化展示。 cAdvisor的截图如图2-3：





总结与展望

Kubernetes是包含容器云全要素的管理引擎，不仅仅是个容器的编排工具，以kubernetes为基础，可以实现以容器为中心的生态系统，也可以实现软件定义的计算。

Kubernetes使用简单、便捷、易学，这让很多小公司、小团队搭建私有的容器云成为了可能。

截至目前为止，Kubernetes现在已经发布1.2.4版本，1.2版本对于之前的版本增添了很多功能，并且支持1000台节点的集群规模。随着Kubernetes的快速发展和社区的日益火爆，我们相信，越来越多的人会参与到kubernetes的开发和使用中来。

以Docker为代表的容器技术会在最近的几年中得到快速的推广，它将迅速颠覆现有软件发布、运维的现状，甚至会改变互联网的技术生态格局。而且，以Swarm，Kubernetes，Mesos , Rancher 为代表的开源容器云基础组件的出现，更加速了这一过程。

容器技术的火爆预示着一个新时代的降临，您准备好了吗？

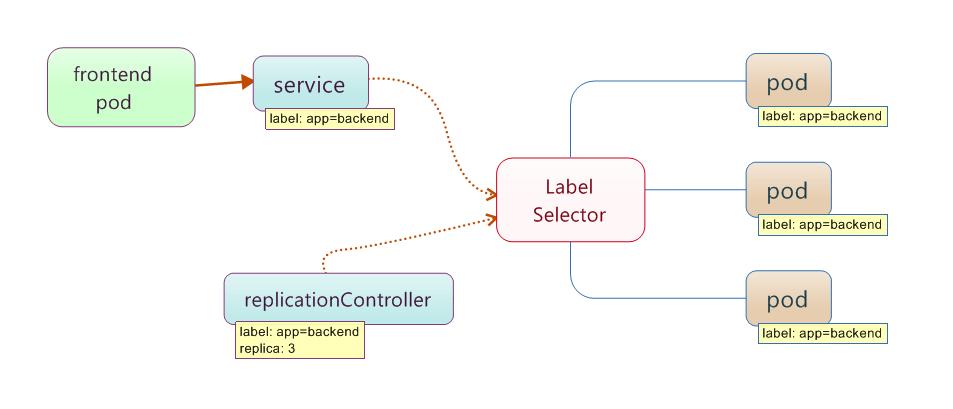
Kubernetes以RESTFul形式开放接口，用户可操作的REST对象有三个：

•pod：是Kubernetes最基本的部署调度单元，可以包含container，逻辑上表示某种应用的一个实例。比如一个web站点应用由前端、后端及[**数据库**](http://lib.csdn.net/base/mysql)构建而成，这三个组件将运行在各自的容器中，那么我们可以创建包含三个container的pod。

•service：是pod的路由代理抽象，用于解决pod之间的服务发现问题。因为pod的运行状态可动态变化(比如切换机器了、缩容过程中被终止了等)，所以访问端不能以写死IP的方式去访问该pod提供的服务。service的引入旨在保证pod的动态变化对访问端透明，访问端只需要知道service的地址，由service来提供代理。

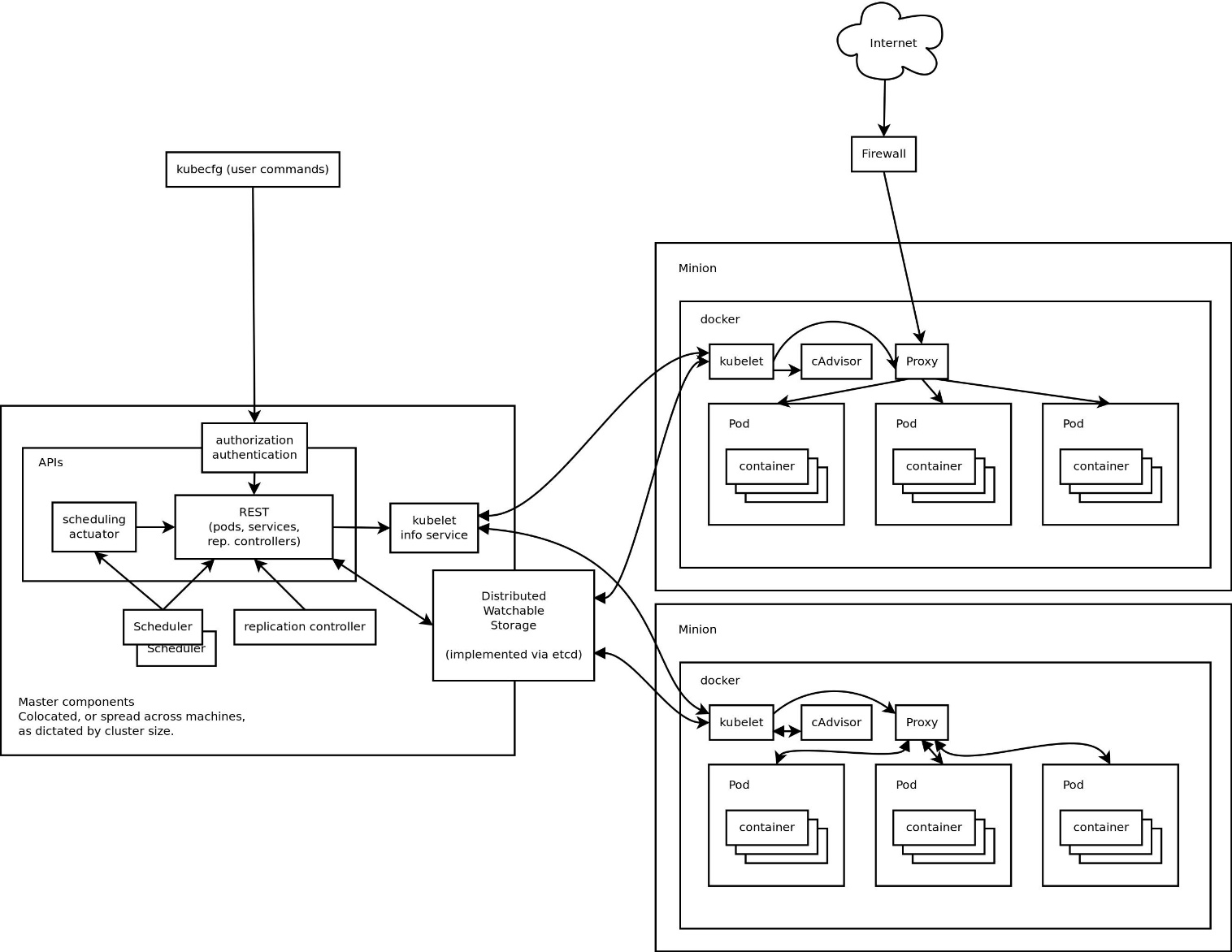
•replicationController：是pod的复制抽象，用于解决pod的扩容缩容问题。通常，分布式应用为了性能或高可用性的考虑，需要复制多份资源，并且根据负载情况动态伸缩。通过replicationController，我们可以指定一个应用需要几份复制，Kubernetes将为每份复制创建一个pod，并且保证实际运行pod数量总是与该复制数量相等(例如，当前某个pod宕机时，自动创建新的pod来替换)。

可以看到，service和replicationController只是建立在pod之上的抽象，最终是要作用于pod的，那么它们如何跟pod联系起来呢？这就要引入label的概念：label其实很好理解，就是为pod加上可用于搜索或关联的一组key/value标签，而service和replicationController正是通过label来与pod关联的。如下图所示，有三个pod都有label为"app=backend"，创建service和replicationController时可以指定同样的label:"app=backend"，再通过label selector机制，就将它们与这三个pod关联起来了。例如，当有其他frontend pod访问该service时，自动会转发到其中的一个backend pod。

[](http://s7.51cto.com/wyfs02/M00/4D/BC/wKiom1RYg2HxAthuAABgorXlnvc755.jpg)

## 功能组件

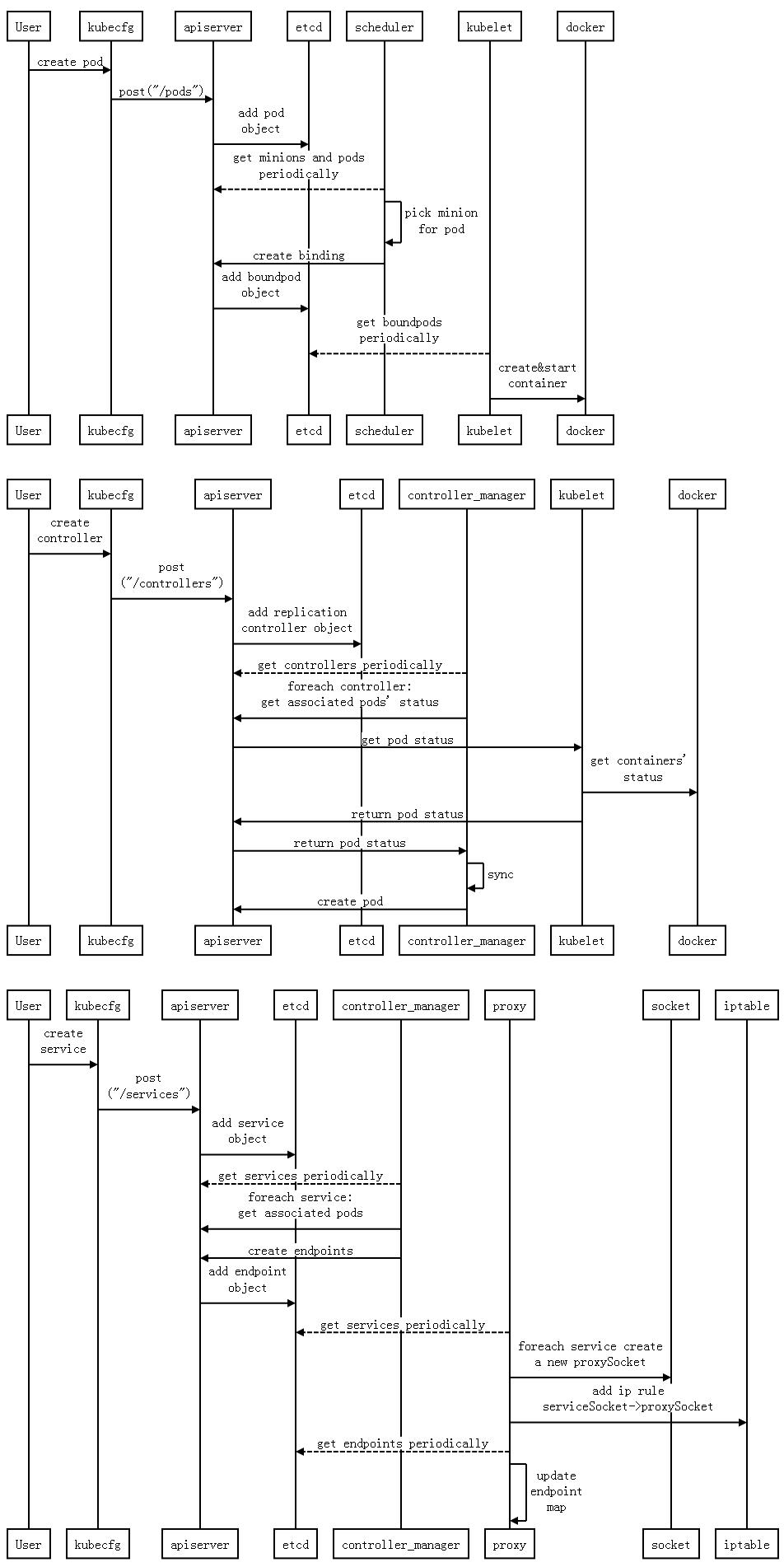
如下图所示是官方文档里的集群架构图，一个典型的master/slave模型。

[](http://s4.51cto.com/wyfs02/M00/4D/BC/wKiom1RYg2Kg-NoJAASwVW0tBsg026.jpg)

master运行三个组件：  
•apiserver：作为kubernetes系统的入口，封装了核心对象的增删改查操作，以RESTFul接口方式提供给外部客户和内部组件调用。它维护的REST对象将持久化到etcd（一个分布式强一致性的key/value存储）。  
•scheduler：负责集群的资源调度，为新建的pod分配机器。这部分工作分出来变成一个组件，意味着可以很方便地替换成其他的调度器。  
•controller-manager：负责执行各种控制器，目前有两类：  
◦endpoint-controller：定期关联service和pod(关联信息由endpoint对象维护)，保证service到pod的映射总是最新的。  
◦replication-controller：定期关联replicationController和pod，保证replicationController定义的复制数量与实际运行pod的数量总是一致的。  
  
slave(称作minion)运行两个组件：  
•kubelet：负责管控docker容器，如启动/停止、监控运行状态等。它会定期从etcd获取分配到本机的pod，并根据pod信息启动或停止相应的容器。同时，它也会接收apiserver的HTTP请求，汇报pod的运行状态。  
•proxy：负责为pod提供代理。它会定期从etcd获取所有的service，并根据service信息创建代理。当某个客户pod要访问其他pod时，访问请求会经过本机proxy做转发。

## 工作流

上文已经提到了Kubernetes中最基本的三个操作对象：pod, replicationController及service。 下面分别从它们的对象创建出发，通过时序图来描述Kubernetes各个组件之间的交互及其工作流。

[](http://s8.51cto.com/wyfs02/M01/4D/BB/wKioL1RYg8TDQpq9AAKMwsVXIak793.jpg)

### 一张图看明白Kubernetes架构4.1. Master

Master定义了Kubernetes 集群Master/API Server的主要声明，包括Pod Registry、Controller Registry、Service Registry、Endpoint Registry、Minion Registry、Binding Registry、RESTStorage以及Client, 是client(Kubecfg)调用Kubernetes API，管理Kubernetes主要构件Pods、Services、Minions、容器的入口。Master由API Server、Scheduler以及Registry等组成。从下图3-2可知Master的工作流主要分以下步骤：

1) Kubecfg将特定的请求，比如创建Pod，发送给Kubernetes Client。

2) Kubernetes Client将请求发送给API server。

3) API Server根据请求的类型，比如创建Pod时storage类型是pods，然后依此选择何种REST Storage API对请求作出处理。

4) REST Storage API对的请求作相应的处理。

5) 将处理的结果存入高可用键值存储系统Etcd中。

6) 在API Server响应Kubecfg的请求后，Scheduler会根据Kubernetes Client获取集群中运行Pod及Minion信息。

7) 依据从Kubernetes Client获取的信息，Scheduler将未分发的Pod分发到可用的Minion节点上。

下面是Master的主要构件的详细介绍：

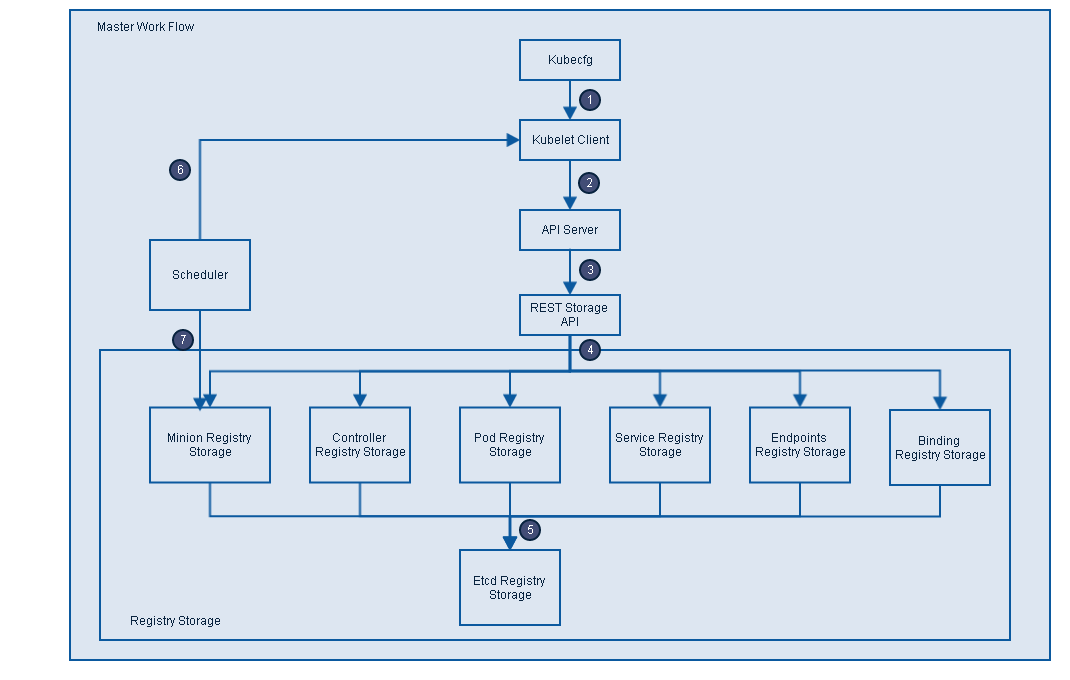


图3-2 Master主要构件及工作流

#### 3.1.1. Minion Registry

Minion Registry负责跟踪Kubernetes 集群中有多少Minion(Host)。Kubernetes封装Minion Registry成实现Kubernetes API Server的RESTful API接口REST，通过这些API，我们可以对Minion Registry做Create、Get、List、Delete操作，由于Minon只能被创建或删除，所以不支持Update操作，并把Minion的相关配置信息存储到etcd。除此之外，Scheduler算法根据Minion的资源容量来确定是否将新建Pod分发到该Minion节点。

#### 3.1.2. Pod Registry

Pod Registry负责跟踪Kubernetes集群中有多少Pod在运行，以及这些Pod跟Minion是如何的映射关系。将Pod Registry和Cloud Provider信息及其他相关信息封装成实现Kubernetes API Server的RESTful API接口REST。通过这些API，我们可以对Pod进行Create、Get、List、Update、Delete操作，并将Pod的信息存储到etcd中，而且可以通过Watch接口监视Pod的变化情况，比如一个Pod被新建、删除或者更新。

#### 3.1.3. Service Registry

Service Registry负责跟踪Kubernetes集群中运行的所有服务。根据提供的Cloud Provider及Minion Registry信息把Service Registry封装成实现Kubernetes API Server需要的RESTful API接口REST。利用这些接口，我们可以对Service进行Create、Get、List、Update、Delete操作，以及监视Service变化情况的watch操作，并把Service信息存储到etcd。

#### 3.1.4. Controller Registry

Controller Registry负责跟踪Kubernetes集群中所有的Replication Controller，Replication Controller维护着指定数量的pod 副本(replicas)拷贝，如果其中的一个容器死掉，Replication Controller会自动启动一个新的容器，如果死掉的容器恢复，其会杀死多出的容器以保证指定的拷贝不变。通过封装Controller Registry为实现Kubernetes API Server的RESTful API接口REST， 利用这些接口，我们可以对Replication Controller进行Create、Get、List、Update、Delete操作，以及监视Replication Controller变化情况的watch操作，并把Replication Controller信息存储到etcd。

#### 3.1.5. Endpoints Registry

Endpoints Registry负责收集Service的endpoint，比如Name："mysql"，Endpoints: ["10.10.1.1:1909"，"10.10.2.2:8834"]，同Pod Registry，Controller Registry也实现了Kubernetes API Server的RESTful API接口，可以做Create、Get、List、Update、Delete以及watch操作。

#### 3.1.6. Binding Registry

Binding包括一个需要绑定Pod的ID和Pod被绑定的Host，Scheduler写Binding Registry后，需绑定的Pod被绑定到一个host。Binding Registry也实现了Kubernetes API Server的RESTful API接口，但Binding Registry是一个write-only对象，所有只有Create操作可以使用， 否则会引起错误。

#### 3.1.7. Scheduler

Scheduler收集和分析当前Kubernetes集群中所有Minion节点的资源(内存、CPU)负载情况，然后依此分发新建的Pod到Kubernetes集群中可用的节点。由于一旦Minion节点的资源被分配给Pod，那这些资源就不能再分配给其他Pod， 除非这些Pod被删除或者退出， 因此，Kubernetes需要分析集群中所有Minion的资源使用情况，保证分发的工作负载不会超出当前该Minion节点的可用资源范围。具体来说，Scheduler做以下工作：

1) 实时监测Kubernetes集群中未分发的Pod。

2) 实时监测Kubernetes集群中所有运行的Pod，Scheduler需要根据这些Pod的资源状况安全地将未分发的Pod分发到指定的Minion节点上。

3) Scheduler也监测Minion节点信息，由于会频繁查找Minion节点，Scheduler会缓存一份最新的信息在本地。

4) 最后，Scheduler在分发Pod到指定的Minion节点后，会把Pod相关的信息Binding写回API Server。

### 4.2. Kubelet

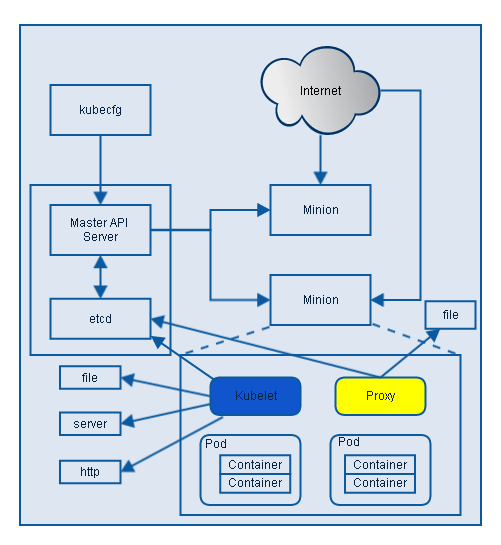


图3-3 Kubernetes详细构件

根据上图3-3可知Kubelet是Kubernetes集群中每个Minion和Master API Server的连接点，Kubelet运行在每个Minion上，是Master API Server和Minion之间的桥梁，接收Master API Server分配给它的commands和work，与持久性键值存储etcd、file、server和http进行交互，读取配置信息。Kubelet的主要工作是管理Pod和容器的生命周期，其包括Docker Client、Root Directory、Pod Workers、Etcd Client、Cadvisor Client以及Health Checker组件，具体工作如下：

1) 通过Worker给Pod异步运行特定的Action。

2) 设置容器的环境变量。

3) 给容器绑定Volume。

4) 给容器绑定Port。

5) 根据指定的Pod运行一个单一容器。

6) 杀死容器。

7) 给指定的Pod创建network 容器。

8) 删除Pod的所有容器。

9) 同步Pod的状态。

10) 从Cadvisor获取container info、 pod info、root info、machine info。

11) 检测Pod的容器健康状态信息。

12) 在容器中运行命令。

### 4.3. Proxy

Proxy是为了解决外部网络能够访问跨机器集群中容器提供的应用服务而设计的，从上图3-3可知Proxy服务也运行在每个Minion上。Proxy提供TCP/UDP sockets的proxy，每创建一种Service，Proxy主要从etcd获取Services和Endpoints的配置信息，或者也可以从file获取，然后根据配置信息在Minion上启动一个Proxy的进程并监听相应的服务端口，当外部请求发生时，Proxy会根据Load Balancer将请求分发到后端正确的容器处理。

## 5. 下篇主题

下篇讲述在CentOS7上用Kubernetes来管理容器。

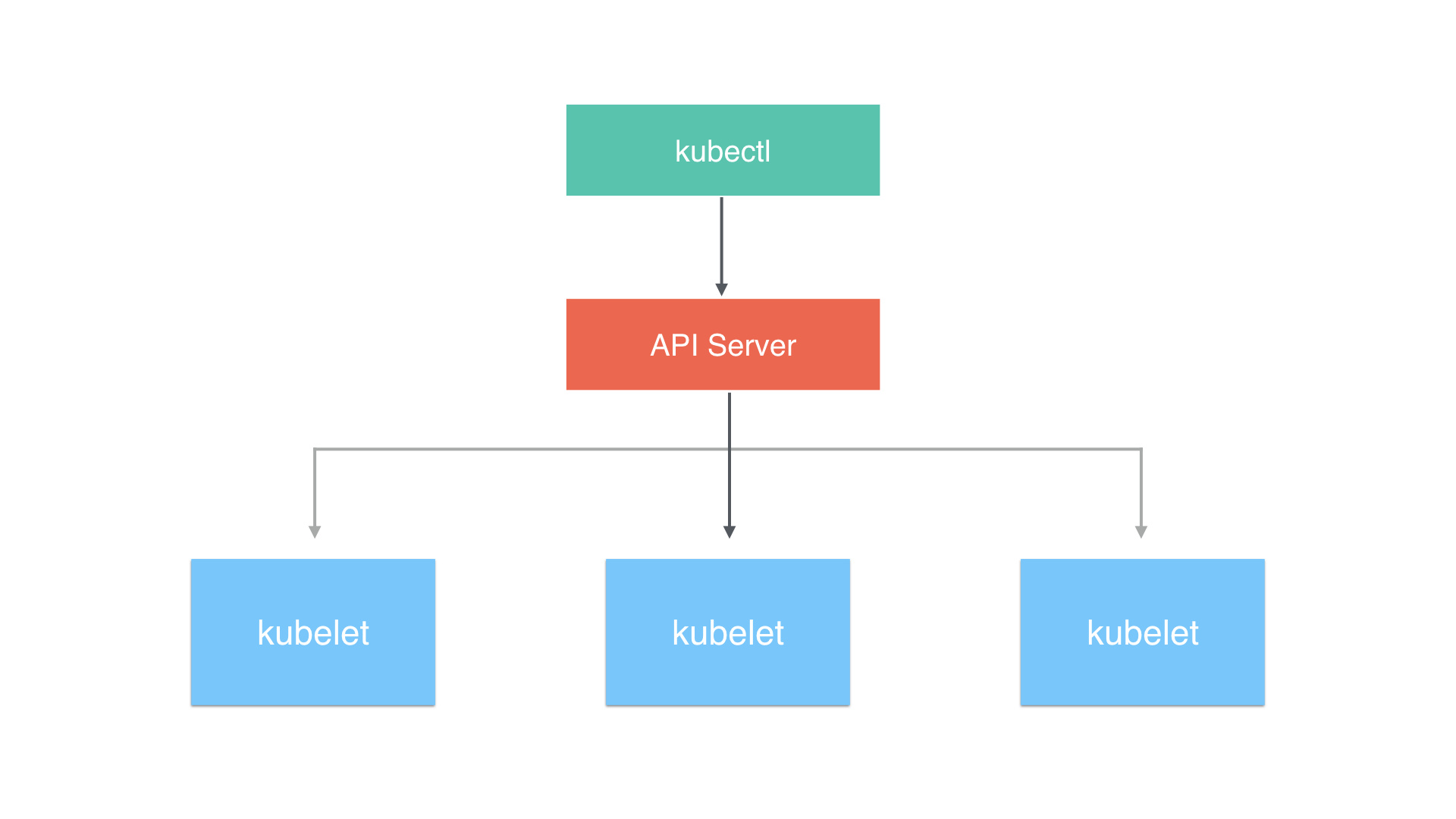
<http://blog.csdn.net/liukuan73/article/details/54971854>

http://blog.csdn.net/liukuan73/article/details/54971854

在 kubernetes 集群中，API Server 有着非常重要的角色。API Server 负责和 etcd 交互（其他组件不会直接操作 etcd，只有 API Server 这么做），是整个 kubernetes 集群的数据中心，所有的交互都是以 API Server 为核心的。简单来说，API Server 提供了一下的功能：

* 整个集群管理的 API 接口：所有对集群进行的查询和管理都要通过 API 来进行
* 集群内部各个模块之间通信的枢纽：所有模块之前并不会之间互相调用，而是通过和 API Server 打交道来完成自己那部分的工作
* 集群安全控制：API Server 提供的验证和授权保证了整个集群的安全

在这篇教程中，我们的系统架构将变成下面这个样子：



我们把要配置的 pod 通过 kubectl 发送给 API Server，里面已经手动指定了要运行的节点。API Server 解析并保存对应的资源，对应的 kubelet 定时拉取数据时候发现 pod 是分配给自己的，会下载对应的配置并执行去生成 pod。

## 参数介绍

API Server 主要是和 etcd 打交道，并且对外提供 HTTP 服务，以及进行安全控制，因此它的命令行提供的参数也主要和这几个方面有关。下面是一些比较重要的参数以及说明（不同版本参数可能会有不同）：

| **参数** | **含义** | **默认值** |
| --- | --- | --- |
| –advertise-address | 通过该 ip 地址向集群其他节点公布 api server 的信息，必须能够被其他节点访问 | nil |
| –allow-privileged | 是否允许 privileged 容器运行 | false |
| –admission-control | 准入控制 | AlwaysAdmit |
| –authorization-mode | 授权模式 ，安全接口上的授权 | AlwaysAllow |
| –bind-address | HTTPS 安全接口的监听地址 | 0.0.0.0 |
| –secure-port | HTTPS 安全接口的监听端口 | 6443 |
| –cert-dir | TLS 证书的存放目录 | /var/run/kubernetes |
| –etcd-prefix | 信息存放在 etcd 中地址的前缀 | “/registry” |
| –etcd-servers | 逗号分割的 etcd server 地址 | [] |
| –insecure-bind-address | HTTP 访问的地址 | 127.0.0.1 |
| –insecure-port | HTTP 访问的端口 | 8080 |
| –log-dir | 日志存放的目录 |  |
| –service-cluster-ip-range | service 要使用的网段，使用 CIDR 格式，参考 kubernetes 中 service 的定义 |  |

## 安装和运行

API Server 是通过提供的 kube-apiserver 二进制文件直接运行的，下面的例子指定了 service 分配的 ip 范围，etcd 的地址，和对外提供服务的 ip 地址：

/usr/bin/kube-apiserver \

--service-cluster-ip-range=10.20.0.1/24 \

--etcd-servers=http://127.0.0.1:2379 \

--advertise-address=192.168.8.100 \

--bind-address=192.168.8.100 \

--insecure-bind-address=192.168.8.100 \

--v=4

这篇教程不会提供对 API Server 进行 HTTPS 的配置，所有的操作都是直接通过 8080 非安全端口访问的。

直接访问 8080 端口，API Server 会返回它提供了哪些接口：

[root@localhost vagrant]# curl http://192.168.8.100:8080

{

"paths": [

"/api",

"/api/v1",

"/apis",

"/apis/apps",

"/apis/apps/v1alpha1",

"/apis/autoscaling",

"/apis/autoscaling/v1",

"/apis/batch",

"/apis/batch/v1",

"/apis/batch/v2alpha1",

"/apis/extensions",

"/apis/extensions/v1beta1",

"/apis/policy",

"/apis/policy/v1alpha1",

"/apis/rbac.authorization.k8s.io",

"/apis/rbac.authorization.k8s.io/v1alpha1",

"/healthz",

"/healthz/ping",

"/logs/",

"/metrics",

"/swaggerapi/",

"/ui/",

"/version"

]

}

而目前最重要的路径是 /api/v1，里面包含了 kubernetes 所有资源的操作，比如下面的 nodes：

➜ ~ http http://192.168.8.100:8080/api/v1/nodes

HTTP/1.1 200 OK

Content-Length: 112

Content-Type: application/json

Date: Thu, 08 Sep 2016 08:14:45 GMT

{

"apiVersion": "v1",

"items": [],

"kind": "NodeList",

"metadata": {

"resourceVersion": "12",

"selfLink": "/api/v1/nodes"

}

}

API 以 json 的形式返回，会通过 apiVersion 来说明 API 版本号，kind 说明请求的是什么资源。不过这里面的内容是空的，因为目前还没有任何 kubelet 节点接入到我们的 API Server。对应的，pod 也是空的：

➜ ~ http http://192.168.8.100:8080/api/v1/pods

HTTP/1.1 200 OK

Content-Length: 110

Content-Type: application/json

Date: Thu, 08 Sep 2016 08:18:53 GMT

{

"apiVersion": "v1",

"items": [],

"kind": "PodList",

"metadata": {

"resourceVersion": "12",

"selfLink": "/api/v1/pods"

}

}

## 添加节点

添加节点也非常简单，启动　kubelet　的时候使用　--api-servers　指定要接入的　API Server 就行。kubelet 启动之后，会把自己注册到指定的 API Server，然后监听 API 对应 pod 的变化，根据 API 中 pod 的实际信息来管理节点上 pod 的生命周期。

现在访问 /api/v1/nodes 就能看到已经添加进来的节点：

➜ ~ http http://192.168.8.100:8080/api/v1/nodes

HTTP/1.1 200 OK

Content-Type: application/json

Date: Thu, 08 Sep 2016 08:27:44 GMT

Transfer-Encoding: chunked

{

"apiVersion": "v1",

"items": [

{

"metadata": {

"annotations": {

"volumes.kubernetes.io/controller-managed-attach-detach": "true"

},

"creationTimestamp": "2016-09-08T08:23:01Z",

"labels": {

"beta.kubernetes.io/arch": "amd64",

"beta.kubernetes.io/os": "linux",

"kubernetes.io/hostname": "192.168.8.100"

},

"name": "192.168.8.100",

"resourceVersion": "65",

"selfLink": "/api/v1/nodes/192.168.8.100",

"uid": "74e16eba-759d-11e6-b463-080027c09e5b"

},

"spec": {

"externalID": "192.168.8.100"

},

"status": {

"addresses": [

{

"address": "192.168.8.100",

"type": "LegacyHostIP"

},

{

"address": "192.168.8.100",

"type": "InternalIP"

}

],

"allocatable": {

"alpha.kubernetes.io/nvidia-gpu": "0",

"cpu": "1",

"memory": "502164Ki",

"pods": "110"

},

"capacity": {

"alpha.kubernetes.io/nvidia-gpu": "0",

"cpu": "1",

"memory": "502164Ki",

"pods": "110"

},

"conditions": [

{

"lastHeartbeatTime": "2016-09-08T08:27:36Z",

"lastTransitionTime": "2016-09-08T08:23:01Z",

"message": "kubelet has sufficient disk space available",

"reason": "KubeletHasSufficientDisk",

"status": "False",

"type": "OutOfDisk"

},

{

"lastHeartbeatTime": "2016-09-08T08:27:36Z",

"lastTransitionTime": "2016-09-08T08:23:01Z",

"message": "kubelet has sufficient memory available",

"reason": "KubeletHasSufficientMemory",

"status": "False",

"type": "MemoryPressure"

},

{

"lastHeartbeatTime": "2016-09-08T08:27:36Z",

"lastTransitionTime": "2016-09-08T08:24:56Z",

"message": "kubelet is posting ready status",

"reason": "KubeletReady",

"status": "True",

"type": "Ready"

}

],

"daemonEndpoints": {

"kubeletEndpoint": {

"Port": 10250

}

},

"images": [

{

"names": [

"172.16.1.41:5000/nginx:latest"

],

"sizeBytes": 425626718

},

{

"names": [

"172.16.1.41:5000/hyperkube:v0.18.2"

],

"sizeBytes": 207121551

},

{

"names": [

"172.16.1.41:5000/etcd:v3.0.4"

],

"sizeBytes": 43302056

},

{

"names": [

"172.16.1.41:5000/busybox:latest"

],

"sizeBytes": 1092588

},

{

"names": [

"172.16.1.41:5000/google\_containers/pause:0.8.0"

],

"sizeBytes": 241656

}

],

"nodeInfo": {

"architecture": "amd64",

"bootID": "48955926-11dd-4ad3-8bb0-2585b1c9215d",

"containerRuntimeVersion": "docker://1.10.3",

"kernelVersion": "3.10.0-123.13.1.el7.x86\_64",

"kubeProxyVersion": "v1.3.1-beta.0.6+fbf3f3e5292fb0",

"kubeletVersion": "v1.3.1-beta.0.6+fbf3f3e5292fb0",

"machineID": "b9597c4ae5f24494833d35e806e00b29",

"operatingSystem": "linux",

"osImage": "CentOS Linux 7 (Core)",

"systemUUID": "823EB67A-057E-4EFF-AE7F-A758140CD2F7"

}

}

}

],

"kind": "NodeList",

"metadata": {

"resourceVersion": "65",

"selfLink": "/api/v1/nodes"

}

}

我们可以看到，kubelet 收集了很多关于自身节点的信息，这些信息也会不断更新。这些信息里面不仅包含节点的系统信息（系统架构，操作系统版本，内核版本等）、还有镜像信息（节点上有哪些已经下载的 docker 镜像）、资源信息（Memory 和 Disk 的总量和可用量）、以及状态信息（是否正常，可以分配 pod等）。

## 和 API Server 通信

下面我们就通过 API Server 来创建 pod，而不是像上篇文章那样用拷贝文件的方式。我们把编写的 yaml 文件转换成 json 格式，保存到文件里。主要注意的是，**我们指定了 nodeName 的名字，这个名字必须和之前通过 /api/v1/nodes 得到的结果中 metadata.labels.kubernetes.io/hostname 保持一致**：

[root@localhost vagrant]# cat nginx\_pod.yml

apiVersion: v1

kind: Pod

metadata:

name: nginx-server

spec:

nodeName: 192.168.8.100

containers:

- name: nginx-server

image: 172.16.1.41:5000/nginx

ports:

- containerPort: 80

volumeMounts:

- mountPath: /var/log/nginx

name: nginx-logs

- name: log-output

image: 172.16.1.41:5000/busybox

command:

- bin/sh

args: [-c, 'tail -f /logdir/access.log']

volumeMounts:

- mountPath: /logdir

name: nginx-logs

volumes:

- name: nginx-logs

emptyDir: {}

使用 curl 执行 POST 请求，设置头部内容为 application/json，传过去文件中的 json 值，可以看到应答(其中 status 为 pending，表示以及接收到请求，正在准备处理)：

# curl -s -X POST -H "Content-Type: application/json" http://192.168.8.100:8080/api/v1/namespaces/default/pods --data @nginx\_pod.json

{

"kind": "Pod",

"apiVersion": "v1",

"metadata": {

"name": "nginx-server",

"namespace": "default",

"selfLink": "/api/v1/namespaces/default/pods/nginx-server",

"uid": "888e95d0-75a9-11e6-b463-080027c09e5b",

"resourceVersion": "573",

"creationTimestamp": "2016-09-08T09:49:28Z"

},

"spec": {

"volumes": [

{

"name": "nginx-logs",

"emptyDir": {}

}

],

"containers": [

{

"name": "nginx-server",

"image": "172.16.1.41:5000/nginx",

"ports": [

{

"containerPort": 80,

"protocol": "TCP"

}

],

"resources": {},

"volumeMounts": [

{

"name": "nginx-logs",

"mountPath": "/var/log/nginx"

}

],

"terminationMessagePath": "/dev/termination-log",

"imagePullPolicy": "Always"

}

],

"restartPolicy": "Always",

"terminationGracePeriodSeconds": 30,

"dnsPolicy": "ClusterFirst",

"nodeName": "192.168.8.100",

"securityContext": {}

},

"status": {

"phase": "Pending"

}

}

返回中包含了我们提交 pod 的信息，并且添加了 status、metadata 等额外信息。

等一段时间去查询 pod，就可以看到 pod 的状态已经更新了：

➜ http http://192.168.8.100:8080/api/v1/namespaces/default/pods

HTTP/1.1 200 OK

Content-Type: application/json

Date: Thu, 08 Sep 2016 09:51:29 GMT

Transfer-Encoding: chunked

{

"apiVersion": "v1",

"items": [

{

"metadata": {

"creationTimestamp": "2016-09-08T09:49:28Z",

"name": "nginx-server",

"namespace": "default",

"resourceVersion": "592",

"selfLink": "/api/v1/namespaces/default/pods/nginx-server",

"uid": "888e95d0-75a9-11e6-b463-080027c09e5b"

},

"spec": {

"containers": [

{

"image": "172.16.1.41:5000/nginx",

"imagePullPolicy": "Always",

"name": "nginx-server",

"ports": [

{

"containerPort": 80,

"protocol": "TCP"

}

],

"resources": {},

"terminationMessagePath": "/dev/termination-log",

"volumeMounts": [

{

"mountPath": "/var/log/nginx",

"name": "nginx-logs"

}

]

},

{

"args": [

"-c",

"tail -f /logdir/access.log"

],

"command": [

"bin/sh"

],

"image": "172.16.1.41:5000/busybox",

"imagePullPolicy": "Always",

"name": "log-output",

"resources": {},

"terminationMessagePath": "/dev/termination-log",

"volumeMounts": [

{

"mountPath": "/logdir",

"name": "nginx-logs"

}

]

}

],

"dnsPolicy": "ClusterFirst",

"nodeName": "192.168.8.100",

"restartPolicy": "Always",

"securityContext": {},

"terminationGracePeriodSeconds": 30,

"volumes": [

{

"emptyDir": {},

"name": "nginx-logs"

}

]

},

"status": {

"conditions": [

{

"lastProbeTime": null,

"lastTransitionTime": "2016-09-08T09:49:28Z",

"status": "True",

"type": "Initialized"

},

{

"lastProbeTime": null,

"lastTransitionTime": "2016-09-08T09:49:44Z",

"status": "True",

"type": "Ready"

},

{

"lastProbeTime": null,

"lastTransitionTime": "2016-09-08T09:49:44Z",

"status": "True",

"type": "PodScheduled"

}

],

"containerStatuses": [

{

"containerID": "docker://8b79eeea60f27b6d3f0a19cbd1b3ee3f83709bcf56574a6e1124c69a6376972d",

"image": "172.16.1.41:5000/busybox",

"imageID": "docker://sha256:8c566faa3abdaebc33d40c1b5e566374c975d17754c69370f78c00c162c1e075",

"lastState": {},

"name": "log-output",

"ready": true,

"restartCount": 0,

"state": {

"running": {

"startedAt": "2016-09-08T09:49:43Z"

}

}

},

{

"containerID": "docker://96e64cdba7b05d4e30710a20e958ff5b8f1f359c8d16d32622b36f0df0cb353c",

"image": "172.16.1.41:5000/nginx",

"imageID": "docker://sha256:51d764c1fd358ce81fd0e728436bd0175ff1f3fd85fc5d1a2f9ba3e7dc6bbaf6",

"lastState": {},

"name": "nginx-server",

"ready": true,

"restartCount": 0,

"state": {

"running": {

"startedAt": "2016-09-08T09:49:36Z"

}

}

}

],

"hostIP": "192.168.8.100",

"phase": "Running",

"podIP": "172.17.0.2",

"startTime": "2016-09-08T09:49:28Z"

}

}

],

"kind": "PodList",

"metadata": {

"resourceVersion": "602",

"selfLink": "/api/v1/namespaces/default/pods"

}

}

可以看到 pod 已经在运行，并且给分配了 ip：172.17.0.2，通过 curl 也可以访问它的服务：

[root@localhost vagrant]# curl -s http://172.17.0.2 | head -n 5

***<!DOCTYPE html>***

<html>

<head>

<title>Welcome to nginx on Debian!</title>

<style>

关于 API Server 提供的所有接口，可以参考官方的 [API 文档][http://kamalmarhubi.com/blog/2015/09/06/kubernetes-from-the-ground-up-the-api-server/]。

## kubectl 命令

理论上，所有 kubernetes 提供的功能都能够直接通过 HTTP API 交互来实现，但是你也看到了，非常复杂。因此 kubernetes 提供了另外了 kubectl 命令行，它封装了 HTTP API 的交互过程，通过一系列的子命令来操作资源。比如我们上面创建 pod 的过程就可以通过

kubectl create -f nginx\_pod.yml

一行命令实现，查看 pod 的信息也很简单：

kubectl -s http://ip:8080 get pods

删除已经创建的 pod，可以使用 delete 命令：

kubectl delete pods/nginx-server

<http://www.cnblogs.com/ldaniel/p/5889931.html>

<http://blog.csdn.net/horsefoot/article/details/53812097>

<http://blog.csdn.net/qq_28703581/article/details/51260454>

<http://www.cnblogs.com/chris-cp/p/6108821.html>

<http://time-track.cn/Kubernetes-resources-summaries.html>

<http://blog.csdn.net/huwh_/article/details/71308171>

<http://time-track.cn/Kubernetes-resources-summaries.html>

<http://www.dockerinfo.net/1048.html>

<http://blog.csdn.net/jincm13/article/details/50886029>

<http://blog.csdn.net/ITBoxes/article/details/70185328>

<http://www.csdn.net/article/2015-06-11/2824933>

<http://www.infoq.com/cn/articles/Kubernetes-system-architecture-introduction>

<http://blog.csdn.net/liukuan73/article/details/54971854>

http://cizixs.com/2016/11/07/kubernetes-intro-api-server