

# 使用Python编写 Kubernetes集群控制器 的实践

高朋  
和鲸科技 架构师  
2024.11.23



**PYCHINA**  
Python 中国社区



# 个人介绍：高朋



负责基于Kubernetes的  
数据科学协同平台ModelWhale的建设

和鲸社区

数据科学实践社区

 ModelWhale

数据科学协同平台

承接多起全国性数据竞赛的支持工作，  
以及多所高校的教学平台的交付任务。



**PYCHINA**

Python 中国社区

# 用云原生的方式做资源管理

在 Kubernetes 中进行资源管理时，通常需要使用控制器来管理相应的对象。在某些依赖数据工具的场景中，Python 常常成为更为合适的选择。

希望能为关注 Kubernetes、云资源调度以及 Python 应用的开发者提供一些实用的参考。

# 使用Python编写Kubernetes集群控制器的实践

一 Kubernetes API和Controller介绍

二 Python SDK 的使用

三 一个云节点的资源控制器的应用场景



kubernetes

# Kubernetes和Controller

Kubernetes的基本概念：

**集群：** Node、Pod、Service等组成部分。

**声明式API：** 用户定义期望状态，Kubernetes负责实现。

Resource: Metadata + Spec + Status

Controller:

持续监控(Watch)资源的当前状态并将其与期望状态同步(Reconcile)。

内置控制器： Deployment Controller、Node Controller、Horizontal Pod Autoscaler

# Kubernetes和Controller

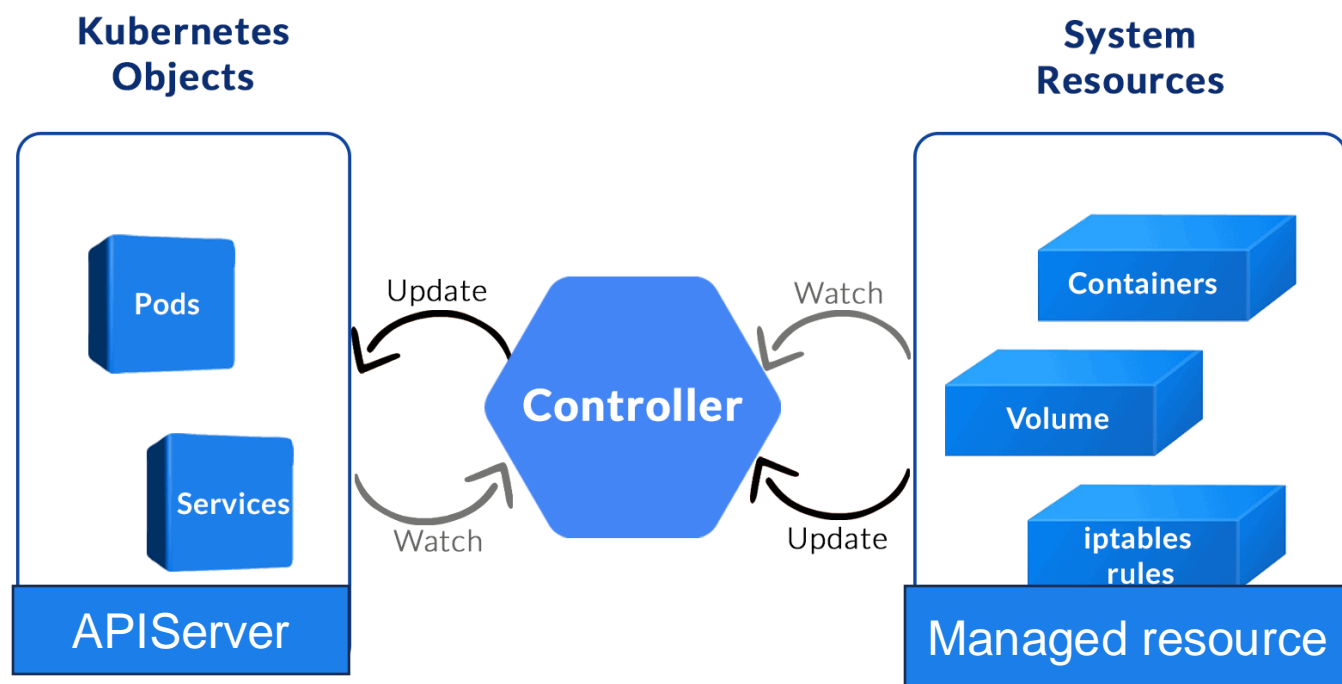
控制器的实现

控制器和K8S的API的设计理念是一致的，都是声明式（有点像最终一致性）。

事件监听

期望状态与当前状态的对比

向声明的状态迁移



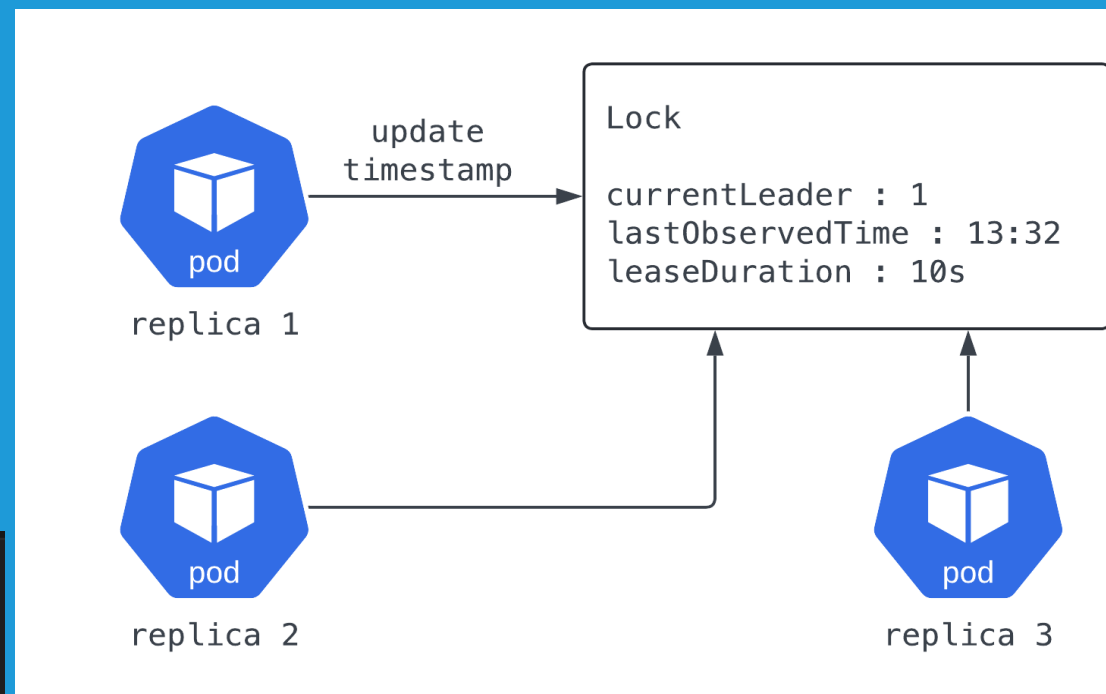
# Kubernetes和Controller

高可用的控制器

通过Lease对象  
获取锁的所有权  
从而成为主

其他副本 Stand By

```
{
  "apiVersion": "coordination.k8s.io/v1",
  "kind": "Lease",
  "metadata": {
    "name": "example-lease",
    "namespace": "example-lease"
  },
  "spec": {
    "holderIdentity": "replica-1",
    "leaseDurationSeconds": 15
  }
}
```



# Kubernetes和Controller



Go作为Kubernetes的“第一公民”：

- 官方支持、广泛的库和文档、性能优化
- 更高的性能，原生支持Kubernetes的高级功能
- 完整的Informer缓存系统。
- 高级库：operator-sdk、kubebuilder等等

Python作为数据科学的“第一公民”：

- 在机器学习、统计学方面的广泛应用，Python库丰富
- API操作简单易用，但缺少Informer中的一些高级功能
- 适合数据科学工作流中的快速集成，如加载模型、统计分析
- 高级库：kopf

Tips To Learn Important Python Libraries





# Kubernetes和Controller

Kubernetes API 常用操作 Create Read Update Patch  
Delete List Watch

GET PUT POST PATCH 都是标准的 HTTP 方法  
Watch 是一种特殊的 GET (query中包含?watch)

LIST 是对对象集合的GET

管理数据库使用的是Etcd，所以很多方法跟Etcd的使用方法也比较像。

# Kubernetes和Controller



Note that since Job is non cluster wide object, it's located within a namespace as contrasted with Node

API Path 规则

CoreAPI `/v1/api`

版本规则

`v1alpha1`

`V2beta3`

`v1`

# Kubernetes和Controller

Watch使用方法，帮助我们基于resourceVersion获取事件。

```
GET /api/v1/namespaces/test/pods?watch=1&sendInitialEvents=true
&allowWatchBookmarks=true&resourceVersion=&resourceVersionMatch=NotOlderThan
---
200 OK
Transfer-Encoding: chunked
Content-Type: application/json

{
  "type": "ADDED",
  "object": {"kind": "Pod", "apiVersion": "v1", "metadata":
    {"resourceVersion": "8467", "name": "foo"}, ...}
}
{
  "type": "ADDED",
  "object": {"kind": "Pod", "apiVersion": "v1", "metadata":
    {"resourceVersion": "5726", "name": "bar"}, ...}
}
{
  "type": "BOOKMARK",
  "object": {"kind": "Pod", "apiVersion": "v1", "metadata":
    {"resourceVersion": "10245"} }
}
...
<followed by regular watch stream starting from resourceVersion="10245">
```

```
GET /api/v1/namespaces/test/pods
---
200 OK
Content-Type: application/json

{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {"resourceVersion": "10245"},
  "items": [...]
}
```

# Python SDK 的使用

## 初始化

```
from kubernetes import client, config

# 加载本地 kubeconfig 配置
config.load_kube_config()
# 加载集群内配置 config.load_incluster_config()
# 创建 API 客户端, 对应的 API Group
core_v1_api = client.CoreV1Api()
apps_v1_api = client.AppsV1Api()
```

# Python SDK 的使用

常用操作 Create Read Update Patch Delete List Watch  
CoreV1Api delete\_namespaced\_pod  
API Group Version action 有无namespace kind

List接口可以不区分namespace就有以下形式  
list\_pod\_for\_all\_namespaces

kubernetes-client/python 同步版本  
tomplus/kubernetes\_asyncio

```
from kubernetes import client, config
```

```
# 加载本地 kubeconfig
```

```
config.load_kube_config()
```

```
# 定义 Deployment
```

```
deployment = client.V1Deployment(  
    metadata=client.V1ObjectMeta(name="nginx-deployment"),  
    spec=client.V1DeploymentSpec(  
        replicas=3,  
        selector=client.V1LabelSelector(match_labels={"app": "nginx"}),  
        template=client.V1PodTemplateSpec(  
            metadata=client.V1ObjectMeta(labels={"app": "nginx"}),  
            spec=client.V1PodSpec(  
                containers=[  
                    client.V1Container(  
                        name="nginx",  
                        image="nginx:1.21",  
                        ports=[client.V1ContainerPort(container_port=80)],  
                    )  
                ],  
            )  
        ),  
    ),  
)
```

```
# 创建 Deployment
```

```
apps_v1 = client.AppsV1Api()  
response = apps_v1.create_namespaced_deployment(namespace="default", body=deployment)
```

```
# 删除 Deployment
```

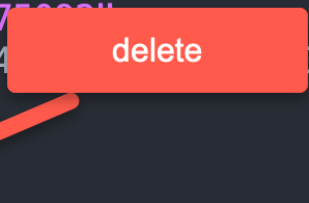
```
apps_v1.delete_namespaced_deployment(  
    name="nginx-deployment",  
    namespace="default"  
)
```

```
print(f"Deployment created: {response.metadata.name}")
```

# Python SDK 的使用

## 创建和删除资源

```
"  
apiVersion: v1  
kind: Namespace  
metadata:  
  annotations:  
    kubectl.kubernetes.io/last-applied-configuration:  
      {"apiVersion":"v1","kind":"Namespace","met  
  creationTimestamp: "2024-09-13T17:07:39Z"  
  deletionTimestamp: "2024-09-19T11:43:13Z"  
  labels:  
    kubernetes.io/metadata.name: payment  
  name: payment  
  resourceVersion: "175000"  
  uid: 379764a6-18df-4b8f-8b3f-35c6  
spec:  
  finalizers:  
    - kubernetes  
status:  
  conditions:  
    - lastTransitionTime: "2024-09-19T11:43:18Z"
```



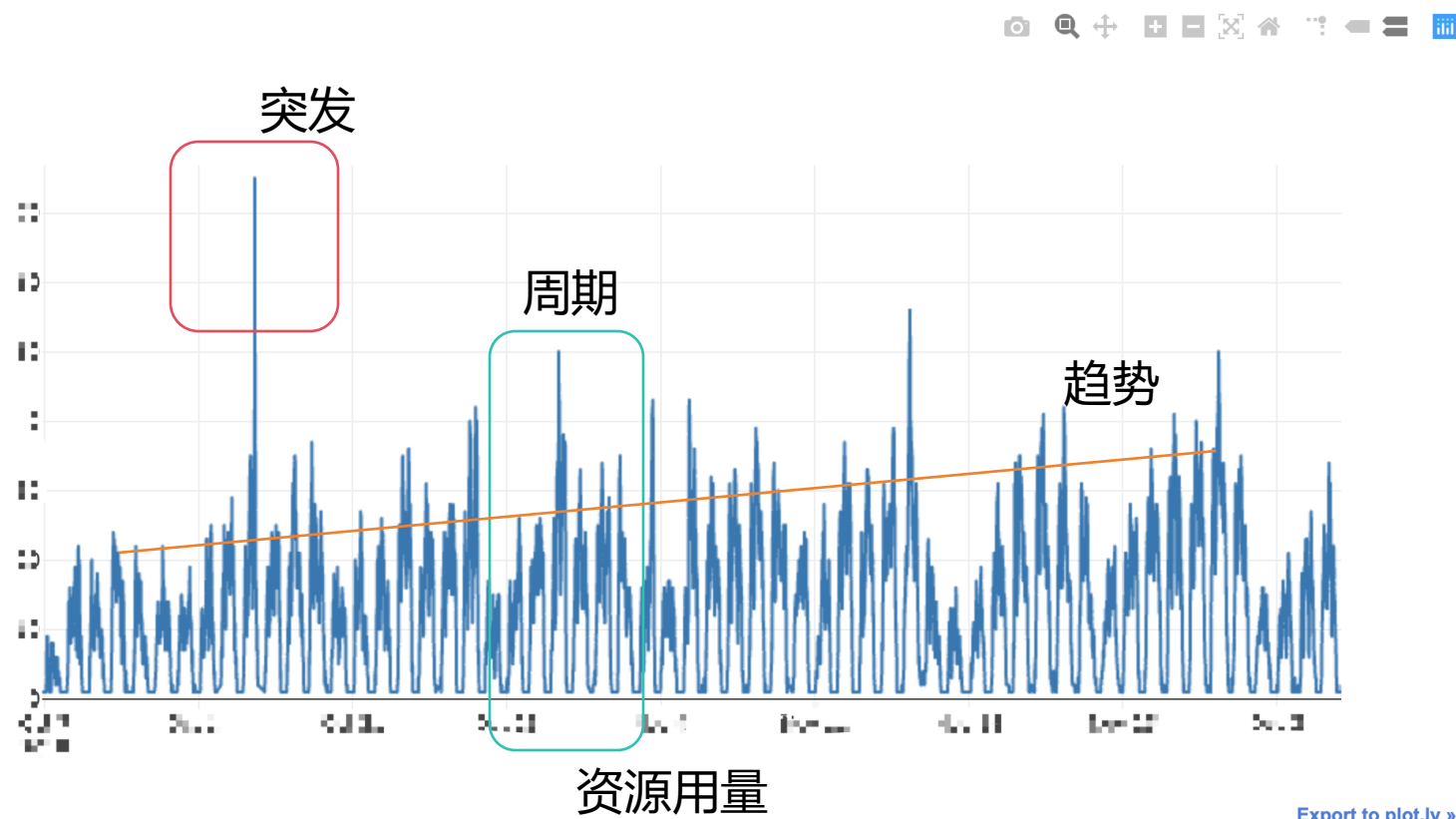
# Python SDK 的使用

Watch使用方法，stream是一个函数wrapper，通过设置list函数中的watch=True，同时帮助我们基于resourceVersion处理事件对象。

```
w = watch.Watch()  
for event in w.stream(core_v1.list_pod_for_all_namespaces, timeout_seconds=60):  
    pod = event['object']  
    print(f"Event: {event['type']}, Pod: {pod.metadata.name}, Status: {pod.status.phase}")
```

# 一个云节点的资源控制器

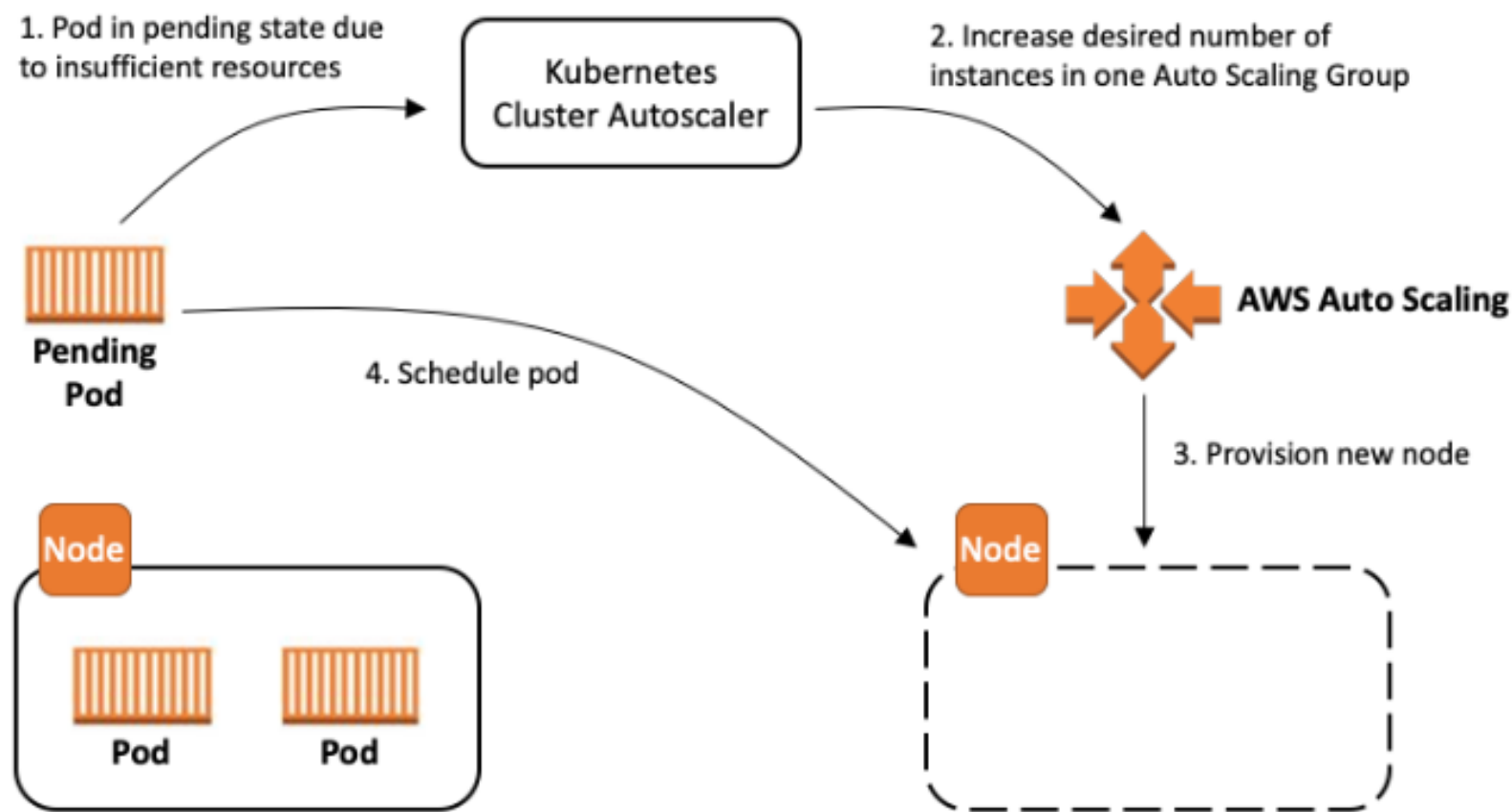
## 一个常规的使用场景



- 资源用量有明显周期性
- 整体用量存在长期的增长趋势
- 用量存在激增的突发情况
- 理想的资源池应该和这条曲线完全重合，资源不存在浪费



# 一个云节点的资源控制器



Cluster-Autoscaler: 通过绑定公有云的自动扩展组对节点进行扩容，在空闲时进行缩容。

Pod Pending -> Scale Node -> Node Idle -> Scale Down Node

使用CA的好处：公有云都有支持，有统一的抽象层，不需要额外实现。对于多云的场景友好。

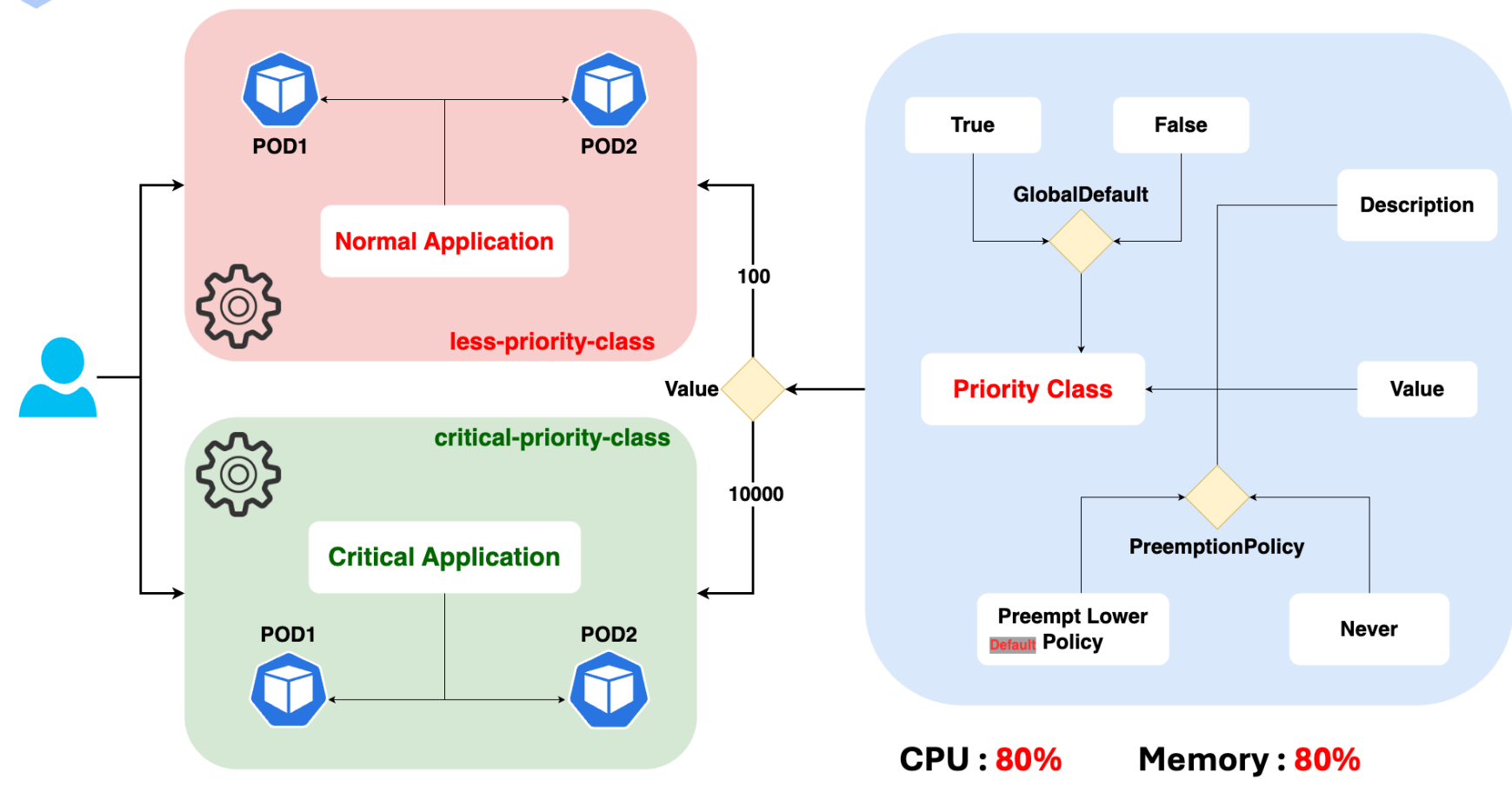
# 一个云节点的资源控制器

通过低权重的方式触发资源的预留，当有实际负载时被高权重的实际负载抢占。  
不用关心节点的调度，只需要做到Pod层面的控制。节点的扩展由Cluster-Autoscaler负责。



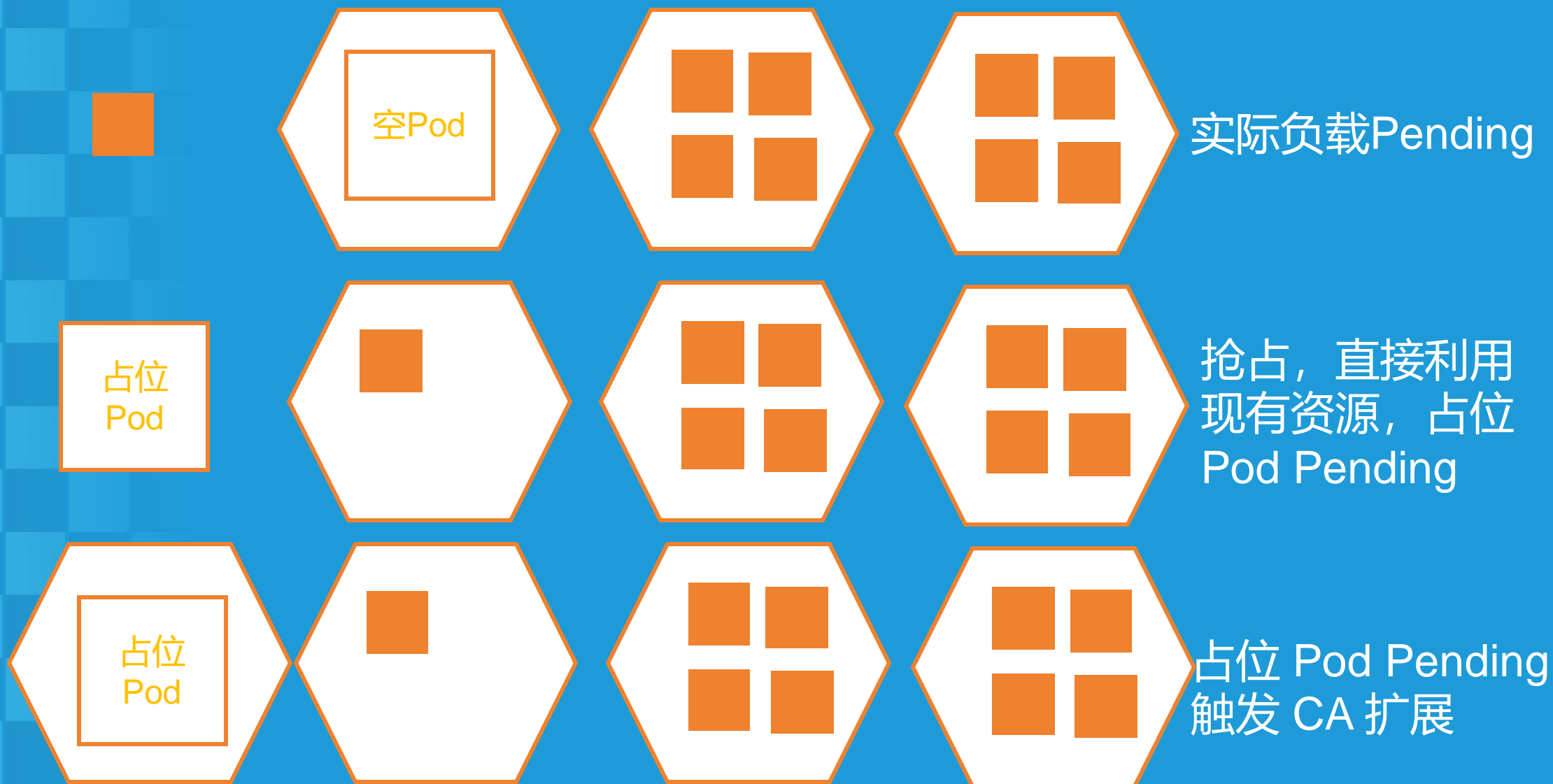
## Priority Classes in Kubernetes

Anvesh Muppada



**apiVersion:** scheduling.k8s.io/v1  
**kind:** PriorityClass  
**metadata:** name: overprovisioning  
**value:** -1

# 一个云节点的资源控制器



# 一个云节点的资源控制器

## 主体循环

`calculate_target_replicas`

例如：期望200核 – 当前运行的正常负载的核数100=100调整值

`scale_deployment`

Deployment 是一个16核的低优先级pod的集合。

例子中的  $100/16 = 6$ ，把deployment副本数调整为6，从而触发CA下层的云资源的扩展。

# 一个云节点的资源控制器

## Watch pod 对象


```
def setup_pod_watch():  
    ...  
    for event in watcher.stream(  
        core_v1_api.list_namespaced_pod,  
        namespace="namespace",  
        label_selector="label=label", # 逗分隔的与选择器  
        timeout_seconds=60  
    ):  
        pod = event['object']  
        if event['type'] in ['ADDED', 'MODIFIED']:  
            pod_cache[pod.metadata.name] = pod  
        elif event['type'] == 'DELETED':  
            pod_cache.pop(pod.metadata.name, None)  
  
import threading  
watch_thread = threading.Thread(target=watch_pods, daemon=True)  
watch_thread.start()
```

# 一个云节点的资源控制器

## 计算CPU核数


```
def get_running_pod_cpu_usage():  
    total_cpu_requested = 0  
    for pod in pod_cache.values():  
        for container in pod.spec.containers:  
            cpu_request = (container.resources.requests or  
                            {}).get("cpu", "0")  
            total_cpu_requested += int(cpu_request.rstrip("m")) /  
            1000 if cpu_request else 0  
    return total_cpu_requested
```

```
resources:  
  limits:  
    cpu: 1500m  
    memory: 7680Mi  
  requests:  
    cpu: 700m  
    memory: 2469606195200m  
securityContext:
```



# 一个云节点的资源控制器

```
day_of week,start,end,cpu_count
Monda 180
Monday,260
Monday,260
Monday,100
Tuesday,260
```



## 计算需要扩展数量

```
def calculate_target_replicas(cpu_needed):
    current_cpu = get_running_pod_cpu_usage()
    print(f"Current CPU usage: {current_cpu}", flush=True)
    cpu_diff = max(0, cpu_needed - current_cpu)
    print(f"CPU difference: {cpu_diff}", flush=True)

    return cpu_diff // INSTANCE_CPU_CORES
```

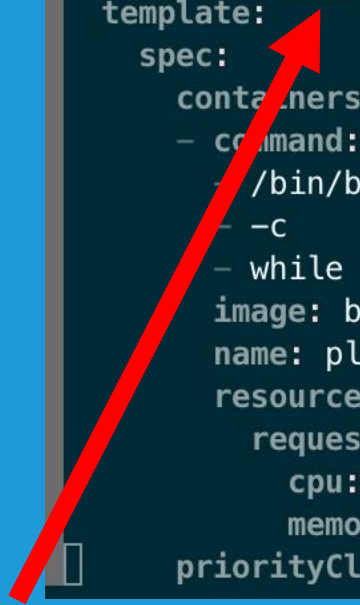
*INSTANCE\_CPU\_CORES* = 16

# 一个云节点的资源控制器

## 扩展占位实例

```
def scale_deployment(target_replicas):  
    body = {  
        "spec": {  
            "replicas": target_replicas  
        }  
    }  
  
    api_response = apps_v1_api.patch_namespaced_deployment_scale(  
        name=DEPLOYMENT_NAME,  
        namespace=NAMESPACE,  
        body=body  
    )
```

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: placeholder-c16-m64  
spec:  
  replicas: 12  
  template:  
    spec:  
      containers:  
      - command:  
        - /bin/bash  
        - -c  
        - while true; do sleep 30; done;  
        image: bash  
        name: placeholder-c16-m64  
        resources:  
          requests:  
            cpu: 1600m  
            memory: 64Gi  
        priorityClassName: overprovisioning
```





# 一个云节点的资源控制器

## 延伸话题：期望的预留CPU核数

- 一 基于统计方法比如：滑动平均值
- 二 基于一些时序预测的机器学习方法：比如最近的Transformer
- 三 基于一些静态的规则，适用于一些无法被学习的突发场景

在这些场景中Python实现的controller会比较方便，因为目前比较流行的统计库或者机器学习库一般是用Python先实现的。

# 谢谢聆听！

