Assessment 2 202283910012 张景橦

**1. Orchestration tools, such as Kubernetes, play a key role in the server infrastructure for the modern applications.**

**(a) Explain how these tools help manage and scale application servers.**

Unified resource management: Integrate scattered servers (nodes) into a logical cluster to centrally manage hardware resources such as CPU, memory, and storage, avoiding resource waste or overload on single nodes.

Container lifecycle management: Automatically monitor the running status of application containers (e.g., health checks). If a container crashes or a node fails, the tool recreates the container on other normal nodes to ensure continuous application availability.

Elastic scaling: Support automatic increase/decrease in the number of application instances based on resource utilization (e.g., CPU utilization exceeding 80%) or custom metrics (e.g., request volume), without manual server operations.

Load balancing: Built-in load distribution capability to evenly distribute user requests to multiple application instances, preventing a single server from crashing due to excessive traffic.

**(b) Describe how orchestration tools facilitate automated deployment, scaling, and management of application servers.**

Orchestration tools enable automated deployment, scaling, and management of application servers through a "declarative configuration + automated execution" mechanism:

Automated deployment: Users only need to define the target state of the application (e.g., image used, number of instances) through a configuration file (such as Kubernetes YAML). The tool automatically compares the current state with the target state and executes the entire process of "pulling the image → creating the container → starting the service" without manual login to the server.

Automated scaling: Support configuration of "Horizontal Pod Autoscaler (HPA) rules". For example, when CPU utilization exceeds 70% for 5 consecutive minutes, instances are automatically increased; when utilization is below 30%, instances are automatically decreased, with no manual intervention throughout the process.

Automated management: Includes three core capabilities: ① Health checks (regularly detect whether containers respond normally, and restart if abnormal); ② Rolling updates (replace old instances in batches when updating applications to ensure services are not interrupted during updates); ③ Fault self-healing (automatically migrate application instances on a failed node to other healthy nodes when the node fails).

**2. Explain the difference between a Pod, Deployment, and Service.**

Pod, Deployment, and Service are three core resources of Kubernetes, with distinct positioning and functions. Their relationship can be summarized as "Deployment manages Pods, and Service provides stable access to Pods":

| **Resource Type** | **Core Positioning** | **Key Functions** | **Lifecycle Relationship** |
| --- | --- | --- | --- |
| Pod | Minimum deployment unit | Contains 1 or more tightly coupled containers (e.g., application container + log collection container), sharing network and storage | Ephemeral and dynamic; IP changes with reconstruction, not directly exposed externally |
| Deployment | Controller for Pods | Manages Pod creation, update, and scaling, ensuring the number of Pods always matches the "desired replica count", supporting rolling updates and rollbacks | Long-term stable; maintains the desired state of Pods through a "control loop" |
| Service | Network access abstraction | Assigns a fixed virtual IP (ClusterIP) to a group of Pods with the same function (matched by labels), realizing "dynamic Pod changes but unchanged access address" | Decoupled from Pods; as long as Pod labels match, Service can route traffic |

**3. What is a Namespace in Kubernetes? Please list one example.**

Definition of Namespace: A Namespace in Kubernetes is a mechanism for logically isolating cluster resources. It does not isolate networks (Pods in different Namespaces can communicate by default), but isolates "resource naming" and "resource quotas"—that is, Pods/Deployments with the same name can exist in different Namespaces, but their resources (CPU, memory) are restricted by the quotas of their respective Namespaces.

Core Role: Solves the problem of multiple teams/environments sharing a cluster. For example, placing the "development environment", "test environment", and "production environment" in different Namespaces to avoid resource naming conflicts and misoperations (e.g., deleting Pods in the production environment).

Example: The default Namespace that comes with Kubernetes by default—if a user creates a resource without specifying a Namespace, the resource is automatically assigned to the default Namespace (e.g., a Pod created by executing kubectl run nginx --image=nginx will be in default).

**4. Explain the role of the Kubelet. How do you check the nodes in a Kubernetes cluster? (kubectl command expected)**

(1) Role of the Kubelet

The Kubelet is a core agent running on each node in the cluster, acting as a "bridge between the Kubernetes control plane and the node". Its core role is to ensure that containers on the node "run according to the Pod definition":

Receive and execute Pod instructions: Obtain the Pod configuration information on the node from the Kubernetes control plane (API Server), and perform operations such as "pulling container images → creating containers → mounting storage volumes → configuring networks".

Continuous health monitoring: Regularly execute the "liveness probe" and "readiness probe" of the Pod. If the liveness probe fails, the container is restarted; if the readiness probe fails, the Pod is marked as "not ready" to avoid traffic routing to the Pod.

Report node status: Real-time report the resource usage (CPU, memory) and health status (e.g., whether the node is online) of the node to the control plane, providing a basis for the scheduler to assign Pods.

(2) kubectl command to check nodes in a Kubernetes cluster

Use the kubectl get nodes command, which lists the name, status (Ready/NotReady), role (Control Plane/Worker), Kubernetes version, and other information of all nodes in the cluster.

To view detailed node information, add the -o wide parameter: kubectl get nodes -o wide (displays the node's internal IP, operating system, container runtime, etc.).

**5. What is the difference between ClusterIP, NodePort, and LoadBalancer services?**

ClusterIP, NodePort, and LoadBalancer are three main types of Services in Kubernetes. Their core differences lie in network access scope and application senarios, with specific comparisons as follows:

| **Service Type** | **Access Scope** | **Core Features** | **Application Scenarios** |
| --- | --- | --- | --- |
| ClusterIP | Cluster-internal only | Automatically assigns a unique virtual IP (ClusterIP) within the cluster; only Pods/services in the cluster can access it | Communication between services within the cluster (e.g., frontend Pods accessing backend API Pods) |
| NodePort | Both internal and external to the cluster | Based on ClusterIP, opens a fixed port (30000-32767) on each node; external access is via "node IP:NodePort" | Temporary external access in development/test environments (e.g., local debugging of in-cluster applications) |
| LoadBalancer | External to the cluster (requires cloud provider support) | Based on NodePort, automatically associates with a cloud provider's load balancer (e.g., AWS ELB, Alibaba Cloud SLB); external access is via the load balancer IP | Formal external access in production environments (requires high availability and stable entry) |

**6. How do you scale a Deployment to 5 replicas using kubectl?**

Use the kubectl scale command, specifying the Deployment name and the target number of replicas (5). The core command format is:

kubectl scale deployment <Deployment Name> --replicas=5

**7. How would you update the image of a Deployment without downtime?**

Use rolling update:

kubectl set image deployment <name> <container>=<new-image>

**8. How do you expose a Deployment to external traffic?**

1.Create NodePort/LoadBalancer Service:

kubectl expose deployment <name> --type=NodePort/LoadBalancer --port=<port>

2.For production, use Ingress: configure domain/path routing to Service (requires Ingress Controller).

**9. How does Kubernetes scheduling decide which node a Pod runs on?**

Two-phase process:

Filtering: Exclude nodes that don't meet resource requirements, affinity rules, etc.

Scoring: Rate eligible nodes based on resource balance, affinity, etc., and select the highest-scoring node.

**10. What is the role of Ingress and how does it differ from a Service?**

Ingress：Manages external HTTP/HTTPS access with domain/path routing and HTTPS (L7 layer, needs controller);

Service：Provides fixed access to Pods (L4 layer, supports TCP/UDP) without complex routing.

Core Differences Between Ingress and Service

Both Ingress and Service are used to expose in-cluster services, but their positioning, functions, and working layers are completely different. The specific differences are shown in the table below:

| **Comparison Dimension** | **Ingress** | **Service** |
| --- | --- | --- |
| **Core Positioning** | "Route manager" for external HTTP/HTTPS traffic to the cluster | "Network access abstraction" for Pods (internal/external to the cluster) |
| **Working Network Layer** | Application layer (L7): Only handles HTTP/HTTPS protocols | Transport layer (L4): Handles TCP/UDP protocols (protocol-agnostic) |
| **Supported Access Rules** | Supports domain/path routing; configurable HTTPS and session persistence | Only supports "IP+port" forwarding; no advanced routing capabilities |
| **Number of Associated Resources** | Can associate multiple Services (multiple services share one entry) | Usually associates with Pods of one Deployment (one-to-one) |
| **Dependent Components** | Requires an Ingress Controller to work | No additional dependencies; natively supported by Kubernetes |
| **Number of External Access Entries** | Single entry (e.g., one Ingress Controller IP) | Independent entry per Service (NodePort/LoadBalancer IP) |
| **Applicable Scenarios** | Production environments with unified exposure of multiple services; scenarios requiring domains/HTTPS | Single-service exposure; in-cluster service communication; simple testing scenarios |