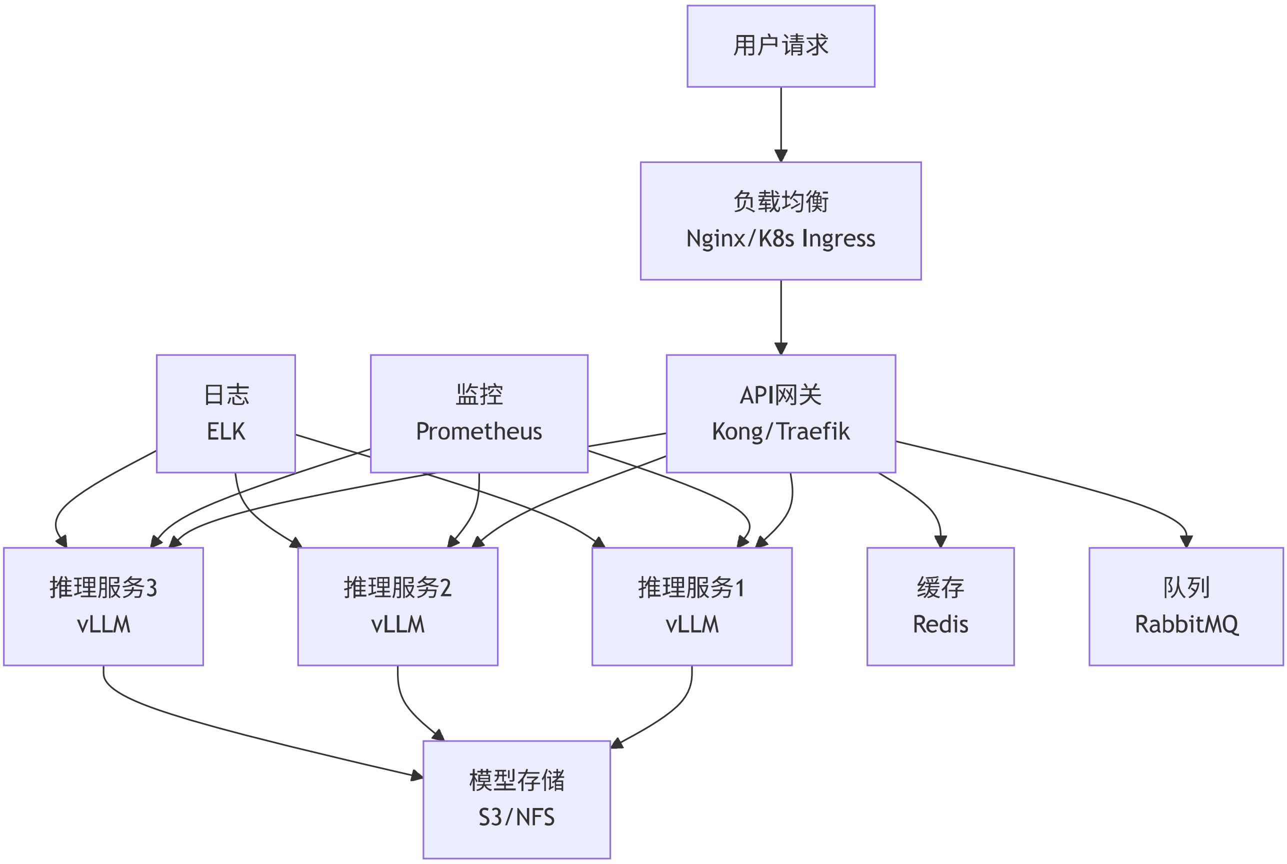
第10章 部署与服务化

> 将大模型从实验室搬到生产环境

10.1 部署架构概览

10.1.1 典型部署架构



10.1.2 部署方式对比

方式	优点	缺点	适用场景
本地部署	完全控制、低延迟	需要GPU资源	开发测试
云服务API	简单、按需付费	成本高、依赖外部	快速原型
自建GPU集群	灵活、成本可控	运维复杂	**生产环境**
Serverless	自动扩展、按用量	冷启动慢	间歇性负载
边缘部署	低延迟、隐私	资源受限	IoT设备

10.2 使用vLLM部署

10.2.1 基础部署

```python
# 1. 安装
pip install vllm

# 2. Python API
from vllm import LLM, SamplingParams

# 初始化模型
llm = LLM(
    model="meta-llama/Llama-2-7b-chat-hf",
    tensor_parallel_size=2,  # 2张GPU并行
    dtype="float16",
    max_model_len=4096
)

# 生成
prompts = [
    "The capital of France is",
    "The largest planet is"
]

sampling_params = SamplingParams(
    temperature=0.8,
    top_p=0.95,
    max_tokens=256
)

outputs = llm.generate(prompts, sampling_params)

for output in outputs:
    print(output.outputs[0].text)
```

10.2.2 OpenAI兼容服务器

```python
# 启动服务器
python -m vllm.entrypoints.openai.api_server \
    --model meta-llama/Llama-2-7b-chat-hf \
    --tensor-parallel-size 2 \
    --port 8000

# 客户端调用（兼容OpenAI API）
from openai import OpenAI

client = OpenAI(
    base_url="http://localhost:8000/v1",
    api_key="fake-key"  # vLLM不需要真实key
```

```python
)

response = client.chat.completions.create(
    model="meta-llama/Llama-2-7b-chat-hf",
    messages=[
        {"role": "user", "content": "你好，介绍一下自己"}
    ]
)

print(response.choices[0].message.content)
```

10.2.3 Docker部署

```dockerfile
# Dockerfile
FROM vllm/vllm-openai:latest

# 设置环境变量
ENV MODEL_NAME=meta-llama/Llama-2-7b-chat-hf
ENV TENSOR_PARALLEL_SIZE=2
ENV GPU_MEMORY_UTILIZATION=0.9

# 启动命令
CMD python -m vllm.entrypoints.openai.api_server \
    --model $MODEL_NAME \
    --tensor-parallel-size $TENSOR_PARALLEL_SIZE \
    --gpu-memory-utilization $GPU_MEMORY_UTILIZATION \
    --host 0.0.0.0 \
    --port 8000
```

```bash
# 构建镜像
docker build -t my-llm-service .

# 运行容器
docker run --gpus all \
    -p 8000:8000 \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    my-llm-service
```

10.3 API服务设计

10.3.1 FastAPI实现

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import List, Optional
import asyncio

app = FastAPI(title="LLM API Service")

# 请求模型
class CompletionRequest(BaseModel):
    prompt: str
    max_tokens: int = 256
    temperature: float = 0.8
    top_p: float = 0.95
    n: int = 1
    stream: bool = False

class ChatMessage(BaseModel):
    role: str
    content: str

class ChatRequest(BaseModel):
    messages: List[ChatMessage]
    max_tokens: int = 256
    temperature: float = 0.8
    top_p: float = 0.95
    stream: bool = False

# 初始化模型（全局单例）
from vllm import LLM, SamplingParams

llm = LLM(model="meta-llama/Llama-2-7b-chat-hf")

@app.post("/v1/completions")
async def create_completion(request: CompletionRequest):
    """
    文本补全接口
    """
    try:
        sampling_params = SamplingParams(
            temperature=request.temperature,
            top_p=request.top_p,
            max_tokens=request.max_tokens,
            n=request.n
        )

        outputs = llm.generate([request.prompt], sampling_params)
```

```python
        return {
            "choices": [
                {
                    "text": output.outputs[0].text,
                    "index": 0,
                    "finish_reason": "stop"
                }
                for output in outputs
            ]
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))


@app.post("/v1/chat/completions")
async def create_chat_completion(request: ChatRequest):
    """
    对话接口
    """
    try:
        # 构造prompt（根据模型格式）
        prompt = format_chat_prompt(request.messages)

        sampling_params = SamplingParams(
            temperature=request.temperature,
            top_p=request.top_p,
            max_tokens=request.max_tokens
        )

        if request.stream:
            # 流式输出
            return StreamingResponse(
                stream_generator(prompt, sampling_params),
                media_type="text/event-stream"
            )
        else:
            # 普通输出
            outputs = llm.generate([prompt], sampling_params)

            return {
                "choices": [{
                    "message": {
                        "role": "assistant",
                        "content": outputs[0].outputs[0].text
                    },
```

```python
                    "finish_reason": "stop"
                }]
            }
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))


def format_chat_prompt(messages: List[ChatMessage]) -> str:
    """
    格式化对话为prompt（LLaMA-2格式）
    """
    prompt = "<s>"
    for msg in messages:
        if msg.role == "system":
            prompt += f"[INST] <<SYS>>\n{msg.content}\n<</SYS>>\n\n"
        elif msg.role == "user":
            prompt += f"[INST] {msg.content} [/INST] "
        elif msg.role == "assistant":
            prompt += f"{msg.content} </s><s>"

    return prompt


async def stream_generator(prompt, sampling_params):
    """
    流式生成器
    """
    # 使用异步生成
    async for text in llm.generate_async(prompt, sampling_params):
        yield f"data: {json.dumps({'text': text})}\n\n"

    yield "data: [DONE]\n\n"


@app.get("/health")
async def health_check():
    """
    健康检查接口
    """
    return {"status": "ok"}


# 启动服务
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000, workers=1)
```

10.3.2 限流和缓存

```python
from fastapi import FastAPI, Request
from slowapi import Limiter, _rate_limit_exceeded_handler
from slowapi.util import get_remote_address
from slowapi.errors import RateLimitExceeded
import redis
import hashlib

app = FastAPI()

# 限流器
limiter = Limiter(key_func=get_remote_address)
app.state.limiter = limiter
app.add_exception_handler(RateLimitExceeded, _rate_limit_exceeded_handler)

# Redis缓存
redis_client = redis.Redis(host='localhost', port=6379, db=0)

def get_cache_key(prompt: str, params: dict) -> str:
    """
    生成缓存key
    """
    content = f"{prompt}_{json.dumps(params, sort_keys=True)}"
    return hashlib.md5(content.encode()).hexdigest()


@app.post("/v1/completions")
@limiter.limit("100/minute")  # 每分钟最多100次请求
async def create_completion(request: Request, req: CompletionRequest):
    """
    带缓存的补全接口
    """
    # 1. 检查缓存
    cache_key = get_cache_key(req.prompt, {
        "temperature": req.temperature,
        "max_tokens": req.max_tokens
    })

    cached = redis_client.get(cache_key)
    if cached:
        return json.loads(cached)

    # 2. 生成结果
    result = await generate_completion(req)
```

```
        # 3. 存入缓存（过期时间1小时）
        redis_client.setex(cache_key, 3600, json.dumps(result))

        return result
```

10.4 批处理和队列

10.4.1 异步批处理

```
import asyncio
from collections import deque

class BatchProcessor:
    """
    批处理器：收集请求，批量推理
    """
    def __init__(self, model, max_batch_size=32, max_wait_time=0.1):
        self.model = model
        self.max_batch_size = max_batch_size
        self.max_wait_time = max_wait_time

        self.queue = deque()
        self.processing = False

    async def add_request(self, prompt, params):
        """
        添加请求到队列
        """
        future = asyncio.Future()
        self.queue.append((prompt, params, future))

        # 触发处理
        if not self.processing:
            asyncio.create_task(self.process_batch())

        # 等待结果
        return await future

    async def process_batch(self):
        """
        处理一批请求
        """
        self.processing = True

        # 等待accumulate
```

```
        await asyncio.sleep(self.max_wait_time)

        # 收集batch
        batch = []
        futures = []

        while self.queue and len(batch) < self.max_batch_size:
            prompt, params, future = self.queue.popleft()
            batch.append((prompt, params))
            futures.append(future)

        if batch:
            # 批量推理
            prompts = [item[0] for item in batch]
            results = self.model.generate(prompts)

            # 返回结果给各个请求
            for future, result in zip(futures, results):
                future.set_result(result)

        self.processing = False

        # 如果还有请求，继续处理
        if self.queue:
            asyncio.create_task(self.process_batch())


# 在FastAPI中使用
batch_processor = BatchProcessor(llm)

@app.post("/v1/completions")
async def create_completion(request: CompletionRequest):
    result = await batch_processor.add_request(
        request.prompt,
        {"temperature": request.temperature}
    )
    return result
```

10.4.2 任务队列（Celery）

```
from celery import Celery
import redis

# 初始化Celery
celery_app = Celery(
```

```python
    'llm_tasks',
    broker='redis://localhost:6379/0',
    backend='redis://localhost:6379/1'
)

@celery_app.task(bind=True)
def generate_text_task(self, prompt, params):
    """
    异步生成任务
    """
    try:
        # 更新任务状态
        self.update_state(state='PROCESSING')

        # 生成
        result = llm.generate([prompt], SamplingParams(**params))

        return {
            "status": "success",
            "text": result[0].outputs[0].text
        }
    except Exception as e:
        return {
            "status": "error",
            "error": str(e)
        }


# FastAPI接口
@app.post("/v1/async/completions")
async def create_async_completion(request: CompletionRequest):
    """
    异步任务接口
    """
    task = generate_text_task.delay(
        request.prompt,
        {
            "temperature": request.temperature,
            "max_tokens": request.max_tokens
        }
    )

    return {
        "task_id": task.id,
        "status": "queued"
    }
```

```python
@app.get("/v1/async/completions/{task_id}")
async def get_task_result(task_id: str):
    """
    查询任务结果
    """
    task = generate_text_task.AsyncResult(task_id)

    if task.ready():
        return {
            "status": "completed",
            "result": task.result
        }
    else:
        return {
            "status": "processing"
        }
```

10.5 监控和日志

10.5.1 Prometheus监控

```python
from prometheus_client import Counter, Histogram, Gauge, generate_latest
from fastapi import Response
import time

# 定义指标
request_count = Counter(
    'llm_requests_total',
    'Total number of requests',
    ['endpoint', 'status']
)

request_duration = Histogram(
    'llm_request_duration_seconds',
    'Request duration in seconds',
    ['endpoint']
)

model_gpu_memory = Gauge(
    'llm_gpu_memory_used_bytes',
    'GPU memory used in bytes',
    ['gpu_id']
)
```

```python
active_requests = Gauge(
    'llm_active_requests',
    'Number of active requests'
)


# 中间件
@app.middleware("http")
async def monitor_middleware(request: Request, call_next):
    """
    监控中间件
    """
    active_requests.inc()
    start_time = time.time()

    try:
        response = await call_next(request)

        # 记录指标
        duration = time.time() - start_time
        request_duration.labels(endpoint=request.url.path).observe(duration)
        request_count.labels(
            endpoint=request.url.path,
            status=response.status_code
        ).inc()

        return response
    finally:
        active_requests.dec()


# 暴露metrics
@app.get("/metrics")
async def metrics():
    """
    Prometheus metrics endpoint
    """
    return Response(
        content=generate_latest(),
        media_type="text/plain"
    )


# 更新GPU内存指标（定时任务）
import torch
```

```python
def update_gpu_metrics():
    """
    更新GPU metrics
    """
    for i in range(torch.cuda.device_count()):
        memory = torch.cuda.memory_allocated(i)
        model_gpu_memory.labels(gpu_id=str(i)).set(memory)
```

10.5.2 结构化日志

```python
import logging
import json
from pythonjsonlogger import jsonlogger

# 配置JSON日志
logHandler = logging.StreamHandler()
formatter = jsonlogger.JsonFormatter(
    '%(asctime)s %(name)s %(levelname)s %(message)s'
)
logHandler.setFormatter(formatter)

logger = logging.getLogger()
logger.addHandler(logHandler)
logger.setLevel(logging.INFO)


# 在API中使用
@app.post("/v1/completions")
async def create_completion(request: CompletionRequest):
    request_id = str(uuid.uuid4())

    logger.info(
        "Request received",
        extra={
            "request_id": request_id,
            "prompt_length": len(request.prompt),
            "temperature": request.temperature,
            "max_tokens": request.max_tokens
        }
    )

    start_time = time.time()

    try:
```

```python
            result = await generate_completion(request)

            logger.info(
                "Request completed",
                extra={
                    "request_id": request_id,
                    "duration": time.time() - start_time,
                    "tokens_generated": len(result["choices"][0]["text"].split())
                }
            )

            return result
        except Exception as e:
            logger.error(
                "Request failed",
                extra={
                    "request_id": request_id,
                    "error": str(e),
                    "duration": time.time() - start_time
                },
                exc_info=True
            )
            raise
```

10.6 Kubernetes部署

10.6.1 Deployment配置

```yaml
# llm-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: llm-service
  labels:
    app: llm-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: llm-service
  template:
    metadata:
      labels:
        app: llm-service
    spec:
```

```yaml
      containers:
      - name: llm-service
        image: my-llm-service:latest
        ports:
        - containerPort: 8000
        resources:
          requests:
            nvidia.com/gpu: 1
            memory: "16Gi"
            cpu: "4"
          limits:
            nvidia.com/gpu: 1
            memory: "32Gi"
            cpu: "8"
        env:
        - name: MODEL_NAME
          value: "meta-llama/Llama-2-7b-chat-hf"
        - name: TENSOR_PARALLEL_SIZE
          value: "1"
        volumeMounts:
        - name: model-cache
          mountPath: /root/.cache/huggingface
      volumes:
      - name: model-cache
        persistentVolumeClaim:
          claimName: model-cache-pvc
```

10.6.2 Service和Ingress

```yaml
# llm-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: llm-service
spec:
  selector:
    app: llm-service
  ports:
  - port: 8000
    targetPort: 8000
  type: ClusterIP

---
# llm-ingress.yaml
apiVersion: networking.k8s.io/v1
```

```yaml
kind: Ingress
metadata:
  name: llm-ingress
  annotations:
    nginx.ingress.kubernetes.io/proxy-body-size: "10m"
    nginx.ingress.kubernetes.io/proxy-read-timeout: "300"
spec:
  rules:
  - host: llm.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: llm-service
            port:
              number: 8000
```

10.6.3 HPA（水平自动扩缩容）

```yaml
# llm-hpa.yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: llm-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: llm-service
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Pods
    pods:
      metric:
        name: llm_active_requests
      target:
```

```yaml
        type: AverageValue
        averageValue: "50"
```

10.7 性能优化

10.7.1 性能指标

指标	定义	目标
延迟（Latency）	首Token时间 + 生成时间	<500ms (首Token) <50ms/token (生成)
吞吐量（Throughput）	每秒处理的tokens数	>1000 tokens/s
并发数（Concurrency）	同时处理的请求数	>100
GPU利用率	GPU计算资源使用率	>80%

10.7.2 优化策略

```python
class OptimizedLLMService:
    """
    优化的LLM服务
    """
    def __init__(self, model_path):
        # 1. 使用vLLM（最优推理引擎）
        self.llm = LLM(
            model=model_path,
            tensor_parallel_size=2,
            dtype="float16",
            gpu_memory_utilization=0.95,  # 最大化GPU利用率
            max_num_seqs=256,  # 最大并发序列数
            max_num_batched_tokens=8192,  # 批处理token上限
        )

        # 2. 预热模型
        self._warmup()

        # 3. 启用缓存
        self.cache = {}

    def _warmup(self):
        """
        预热模型（避免冷启动）
        """
        dummy_prompts = ["Hello"] * 10
        self.llm.generate(dummy_prompts, SamplingParams(max_tokens=10))
```

```python
    async def generate(self, prompt, params):
        """
        生成（带优化）
        """
        # 检查缓存
        cache_key = self._get_cache_key(prompt, params)
        if cache_key in self.cache:
            return self.cache[cache_key]

        # 生成
        result = await self._generate_with_retry(prompt, params)

        # 更新缓存
        self.cache[cache_key] = result

        return result

    async def _generate_with_retry(self, prompt, params, max_retries=3):
        """
        带重试的生成
        """
        for attempt in range(max_retries):
            try:
                return self.llm.generate([prompt], SamplingParams(**params))
            except Exception as e:
                if attempt == max_retries - 1:
                    raise
                await asyncio.sleep(0.1 * (2 ** attempt))  # 指数退避
```

10.8 成本优化

10.8.1 成本分析

```python
class CostAnalyzer:
    """
    成本分析器
    """
    def __init__(self):
        self.gpu_cost_per_hour = {
            "A100-80GB": 3.0,  # $/hour
            "A100-40GB": 2.5,
            "A10": 1.0,
            "T4": 0.5
        }
```

```python
def calculate_cost(self, model_size, requests_per_day, avg_tokens_per_request):
    """
    计算日成本

    Args:
        model_size: 模型大小（如"7B", "13B", "70B"）
        requests_per_day: 每日请求数
        avg_tokens_per_request: 平均每请求生成tokens
    """
    # 1. 估算所需GPU
    gpu_type, num_gpus = self._estimate_gpu_requirement(model_size)

    # 2. 估算每请求时间
    time_per_request = avg_tokens_per_request * 0.05  # 假设50ms/token

    # 3. 计算每日GPU时间
    total_time_hours = (requests_per_day * time_per_request) / 3600

    # 4. 考虑利用率（通常只能达到50-70%）
    utilization = 0.6
    required_gpu_hours = (total_time_hours / utilization) * num_gpus

    # 5. 计算成本
    daily_cost = required_gpu_hours * self.gpu_cost_per_hour[gpu_type]

    return {
        "gpu_type": gpu_type,
        "num_gpus": num_gpus,
        "daily_cost": daily_cost,
        "monthly_cost": daily_cost * 30,
        "cost_per_1k_requests": daily_cost / (requests_per_day / 1000)
    }

def _estimate_gpu_requirement(self, model_size):
    """
    估算GPU需求
    """
    requirements = {
        "7B": ("A10", 1),
        "13B": ("A100-40GB", 1),
        "70B": ("A100-80GB", 4)
    }
    return requirements.get(model_size, ("A100-80GB", 1))
```

```python
# 使用
analyzer = CostAnalyzer()
cost = analyzer.calculate_cost(
    model_size="7B",
    requests_per_day=100000,
    avg_tokens_per_request=200
)
print(f"Monthly cost: ${cost['monthly_cost']:.2f}")
```

10.8.2 成本优化策略

策略	节省	实现难度
量化（INT8/INT4）	50-75%	⭐⭐
使用更小模型	70-90%	⭐
Spot实例	60-70%	⭐⭐
批处理	30-50%	⭐⭐⭐
缓存	20-40%	⭐⭐
请求去重	10-20%	⭐

10.9 面试高频问题

Q1: 如何选择部署方案?

决策树:

```
请求量?
├─ <1000/day → API服务（OpenAI等）
├─ 1000-10000/day → 云GPU实例 + vLLM
└─ >10000/day → 自建集群 + K8s

数据敏感性?
├─ 高 → 自建/私有云
└─ 低 → 公有云

延迟要求?
├─ <100ms → 边缘部署
└─ <1s → 标准云部署
```

Q2: vLLM为什么快?

核心技术:

1. **PagedAttention**：KV Cache分页管理
2. **Continuous Batching**：动态组batch
3. **优化的CUDA kernels**：高效的底层实现

性能对比：

```
HuggingFace: 100 tokens/s
vLLM: 800+ tokens/s (8x提升)
```

Q3: 如何处理突发流量？

策略：

1. **自动扩容**：K8s HPA
2. **限流**：保护后端
3. **队列**：削峰填谷
4. **降级**：返回cached或简化回复

```python
# 限流+队列组合
@app.post("/v1/completions")
@limiter.limit("100/minute")
async def create_completion(request: CompletionRequest):
    if active_requests.get() > MAX_CONCURRENT:
        # 超过并发上限，加入队列
        return await enqueue_request(request)
    else:
        return await process_immediately(request)
```

Q4: 如何降低推理成本？

优先级排序：

1. **量化**（最高ROI）：成本降50%+，精度损失<5%
2. **批处理**：吞吐量提升3-5x
3. **缓存**：重复请求免费
4. **更小模型**：7B vs 70B，成本差10倍

Q5: 生产环境必备的监控指标？

三大类：

1. 业务指标：

- QPS（每秒请求数）
- 延迟（P50, P95, P99）
- 错误率

2. 系统指标：

- GPU利用率
- GPU显存
- CPU/内存

3. 成本指标：

- 每1K请求成本
- GPU空闲时间
- 缓存命中率

10.10 本章小结

本章全面介绍了大模型的部署与服务化：

✅ **部署方案**：本地、云服务、自建集群 ✅ **推理引擎**：vLLM是首选 ✅ **API设计**：FastAPI + 异步 + 批处理 ✅ **K8s部署**：容器化 + 自动扩缩容 ✅ **监控日志**：Prometheus + 结构化日志 ✅ **成本优化**：量化、批处理、缓存

关键要点：

- vLLM是生产环境首选推理引擎
- 批处理和缓存能显著提升吞吐量
- 监控和日志对生产至关重要
- 量化是成本优化的最佳手段

下一章预告： 第11章将讲解模型评估方法和基准测试。