# **TACoreService 高性能核心服务开发蓝图 V1.0**

### **蓝图导读**

* **致项目经理/开发团队**: 本蓝图是模组十二：TradingAgents-CN核心服务 (TACoreService)的最终、唯一的开发指南。它将指导您直接构建一个具备负载均衡和水平扩展能力的高性能服务，以支撑整个AI交易系统在高并发场景下的稳定运行。
* **核心目标**: 从一开始就构建一个生产级的、可扩展的服务集群，避免未来的架构重构。

## **一、需求书 (Requirements Document)**

* **口语化说明**: 我们要打造的TACoreService，不仅仅是一个功能的“转接器”，更是一个能同时处理多项任务、并且可以随时增派人手的“服务大厅”。

| **字段** | **值** | **说明** |
| --- | --- | --- |
| **ID** | tacore-service-module | 模块唯一标识 |
| **Version** | 1.0 | 版本号 |
| **Description** | 作为系统中TradingAgents-CN功能的唯一、统一的服务化封装，通过高性能的负载均衡架构，为其他所有模组提供稳定、可扩展的AI交易能力。 | 模块核心职责 |
| **Features** | [{"id":"feat-01", "name":"统一功能接口", "desc":"对外提供标准化的服务接口，如市场扫描、订单执行、风险评估等。"}, {"id":"feat-02", "name":"多工作进程处理", "desc":"服务能够启动多个并行的工作进程，每个进程都能独立处理一个完整的业务请求。"}, {"id":"feat-03", "name":"自动负载均衡", "desc":"所有来自客户端的请求，能被自动、公平地分发到当前空闲的工作进程上。"}, {"id":"feat-04", "name":"水平扩展能力", "desc":"支持通过简单地增加服务容器（worker）的数量，来线性地提升整个服务的处理能力。"}, {"id":"feat-05", "name":"高可用性", "desc":"单个工作进程的崩溃或重启，不应影响整个服务的可用性。"}] | 核心功能列表 (JSON格式) |

## **二、实施计划 (Implementation Plan)**

* **口语化说明**: 整个开发工程分三步走。第一步，把实际干活的“接待员”（工作进程）给培训好；第二步，搭建“服务大厅”的入口，并雇佣一个聪明的“任务分发员”（负载均衡代理）；第三步，进行一次严格的“压力测试”，看看我们的“服务大厅”究竟能同时接待多少客人。

| **阶段** | **核心任务** | **预估工时** | **关键产出物** |
| --- | --- | --- | --- |
| **1. 工作进程开发** | **将核心逻辑封装为Worker**: 编写一个worker.py脚本，使其能够连接到消息总线，接收任务，调用TradingAgents-CN的相应功能，处理完毕后返回结果。并为其编写单元测试。 | **2个工作日** | 一个功能完整、经过测试的worker.py。 |
| **2. 负载均衡代理开发** | **构建ZeroMQ代理服务**: 编写一个main.py脚本，使用ZeroMQ的ROUTER/DEALER模式创建一个负载均衡代理，负责在客户端和工作进程之间转发消息。 | **2个工作日** | 一个作为“任务分发员”的main.py。 |
| **3. 部署与压力测试** | 编写部署配置: 编写Dockerfile和docker-compose.yml，使其能够一键启动一个代理服务和多个Worker服务。  编写并执行压力测试: 编写一个脚本，模拟多个客户端（如100个）同时向TACoreService发起请求，测量其吞吐量和响应延迟。 | **1个工作日** | 1. 完整的部署配置文件。  2. 一份详细的压力测试报告，证明服务的性能达标。 |

## **三、搭建规范 (Construction Specification)**

* **口语化说明**: 这是本次开发的“施工图纸”，详细说明了我们如何用代码来实现这个强大的“服务大厅”。

| **规范项** | **具体内容** | **注释** |
| --- | --- | --- |
| **核心架构模式** | **ZeroMQ Lazy Pirate Pattern (ROUTER/DEALER)** | 这是ZeroMQ中用于构建可靠的、可扩展的请求-响应服务的经典模式。 |
| **架构图** | ```mermaid |  |
| graph TD |  |  |
| subgraph 客户端模组 (无需改动) |  |  |
| C1[扫描器] |  |  |
| C2[交易员] |  |  |
| C3[...] |  |  |
| end |  |  |

subgraph TACoreService 代理 (1个容器)  
 Proxy["负载均衡代理 (main.py)<br>ROUTER/DEALER"]  
end  
  
subgraph TACoreService Worker集群 (可多个容器)  
 W1["工作进程1 (worker.py)"]  
 W2["工作进程2 (worker.py)"]  
 W3["工作进程3 (worker.py)"]  
end  
  
C1 & C2 & C3 -- REQ请求 --> Proxy;  
Proxy -- 自动分发任务 --> W1;  
Proxy -- 自动分发任务 --> W2;  
Proxy -- 自动分发任务 --> W3;  
W1 -- REP响应 --> Proxy;  
W2 -- REP响应 --> Proxy;  
W3 -- REP响应 --> Proxy;  
Proxy -- 返回最终结果 --> C1 & C2 & C3;

``` | 客户端（扫描器、交易员等）无需任何改动，它们依然像以前一样向同一个地址发送REQ请求。所有的负载均衡和并行处理对它们是完全透明的。 |

| 代码实现示例 | `12TACoreService/main.py` (代理服务):

# main.py - Load Balancing Proxy  
import zmq  
  
def main():  
 """Main function to run the proxy."""  
 context = zmq.Context()  
   
 # Socket facing clients  
 frontend = context.socket(zmq.ROUTER)  
 frontend.bind("tcp://\*:5555")  
   
 # Socket facing workers  
 backend = context.socket(zmq.DEALER)  
 backend.bind("tcp://\*:5556")  
   
 print("Load balancing proxy started...")  
   
 # Start the built-in proxy device  
 try:  
 zmq.proxy(frontend, backend)  
 except KeyboardInterrupt:  
 print("Proxy interrupted.")  
 finally:  
 frontend.close()  
 backend.close()  
 context.term()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**12TACoreService/worker.py (工作进程)**:

# worker.py - The actual worker process  
import zmq  
import json  
import time  
# from trading\_agents\_cn import your\_functions # 导入您的核心库  
  
def process\_request(request):  
 """  
 This is where the actual business logic resides.  
 It calls the TradingAgents-CN library functions.  
 """  
 method = request.get('method')  
 params = request.get('params')  
 print(f"Worker received request: {method}")  
   
 # Example: route request to the correct function  
 # if method == 'scan.market':  
 # result = your\_functions.scan\_market(params)  
 # elif method == 'execute.order':  
 # result = your\_functions.execute\_order(params)  
 # else:  
 # result = {"error": "Unknown method"}  
   
 # Simulate processing time  
 time.sleep(1)   
 result = {"status": "success", "data": f"Processed {method}"}  
   
 return result  
  
def main():  
 """Main worker function."""  
 context = zmq.Context()  
 socket = context.socket(zmq.REP)  
 # Note: Workers connect to the proxy's backend  
 socket.connect("tcp://tacore\_service:5556")  
   
 print("Worker started, waiting for tasks...")  
   
 while True:  
 try:  
 message = socket.recv\_string()  
 request = json.loads(message)  
   
 response\_data = process\_request(request)  
 # Add request\_id back for client-side matching  
 response\_data['request\_id'] = request.get('request\_id')  
   
 socket.send\_string(json.dumps(response\_data))  
 except Exception as e:  
 print(f"Error processing request: {e}")  
 # Send an error response back  
 error\_response = {  
 "request\_id": request.get('request\_id'),  
 "status": "error",  
 "message": str(e)  
 }  
 socket.send\_string(json.dumps(error\_response))  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()  
``` | 代理服务非常简洁，核心功能由`zmq.proxy`完成。工作进程则专注于业务逻辑，并且是无状态的，便于扩展。 |  
| \*\*部署规范\*\* | \*\*`12TACoreService/docker-compose.yml` (用于独立测试)\*\*:  
```yaml  
version: '3.8'  
services:  
 tacore\_service:  
 build: .  
 container\_name: tacore\_service  
 command: python main.py  
 ports:  
 - "5555:5555" # Expose frontend port to clients  
 networks:  
 - tacore\_net  
  
 tacore\_worker:  
 build: .  
 command: python worker.py  
 depends\_on:  
 - tacore\_service  
 networks:  
 - tacore\_net  
 deploy:  
 replicas: 4 # Start with 4 workers, adjust based on CPU cores  
  
networks:  
 tacore\_net:

**在项目总docker-compose.yml中的配置**:

# In root docker-compose.yml  
services:  
 # ... other services like scanner, trader ...  
  
 tacore\_service:  
 build: ./12TACoreService  
 container\_name: tacore\_service  
 command: python main.py  
 # No need to expose port 5556 as it's for internal communication  
 # Port 5555 is accessed by other containers via the docker network  
  
 tacore\_worker:  
 build: ./12TACoreService  
 command: python worker.py  
 depends\_on:  
 - tacore\_service  
 deploy:  
 replicas: 4 # Easily scale the number of workers here  
``` | 通过`deploy.replicas`配置，我们可以非常轻松地启动任意数量的工作进程，实现弹性扩容。 |