重庆鉴越科技有限公司-春招复习

介绍自己

各位老师好，我是赵时逸，目前在重庆理工大学攻读人工智能硕士，研究方向是跨域跨模态迁移学习，并应用于中心动脉压预测。

在学术方面，我与导师共同发表了一篇中科院二区顶刊论文，并主持了一项校级科研项目，同时申请了一项相关专利。

在实习经历上，我曾在北京高科数聚担任算法实习生，主要负责**Agent模块开发、SQLAgent优化以及LangChain工具调优**，并参与了汽车品牌舆情分析，利用PaddleNLP进行多层次文本分类。

此外，我也对时间序列预测有深入研究，曾基于Transformer改进模型完成长序列预测任务，并在多个数据集上取得了较好的实验结果。

我擅长大语言模型（LLM）、深度学习以及数据分析，希望能在贵公司贡献我的技术能力，谢谢！

模拟题

第一部分：**Python基础与高级特性**

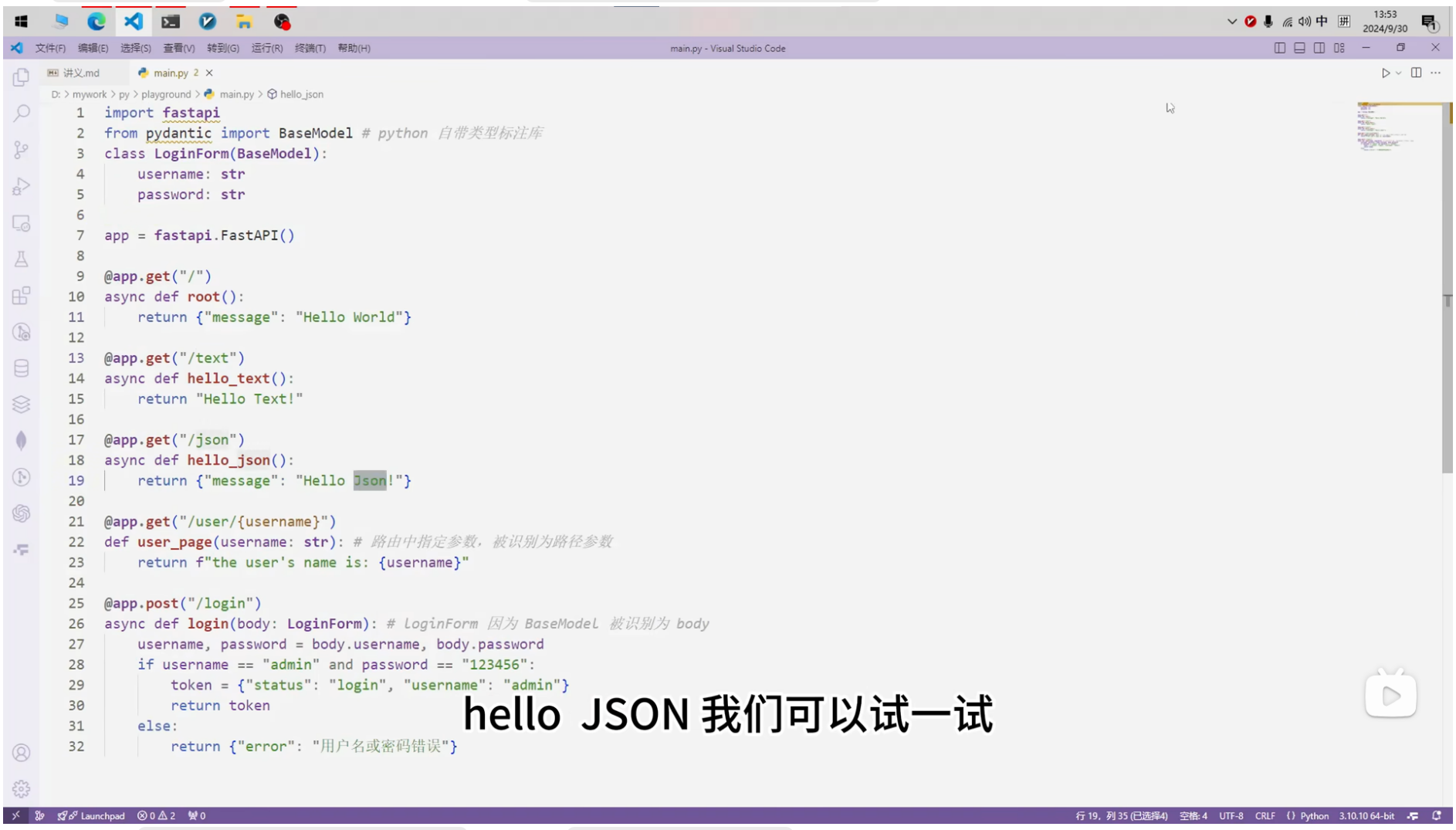
1. Python 的 `asyncio` 库用于异步编程，请实现一个异步 HTTP 请求函数，使用 `aiohttp` 发送 5 个并发请求，并收集返回的结果。
2. Python 中 `deepcopy` 和 `copy` 的区别是什么？请举例说明。

第二部分：数据库与数据处理

1. 给定一张 `user_activity` 数据表（字段：`id`, `user_id`, `action`, `timestamp`），请写一条 SQL 语句，统计每天每个 `user_id` 的 `action` 次数，并按日期排序。
2. 4. Redis 支持的数据结构有哪些？请列举 3 种并简要说明其应用场景。

第三部分：后端开发与微服务

1. 请用 `FastAPI` 编写一个 RESTful API，包含：
   - 一个 `POST /predict` 端点，接收 JSON 格式的输入 `{"text": "some input text"}`。



   - 调用一个本地 `Hugging Face Transformers` 模型（如 `bert-base-chinese`），对输入文本进行分类，并返回预测结果。
2. 在微服务架构中，如何使用 Kafka 实现异步消息队列？请简要描述 Kafka 的作用和基本工作流程。

第四部分：**AI 模型与应用**

1. 你在实习经历中提到了 `LangChain`，请问 `LangChain` 如何封装 LLM 调用？请简述 `Agent` 和 `Memory` 组件的作用。
2. 你在舆情分析项目中使用了零样本分类，请用 PaddleNLP `Taskflow` 编写一个零样本分类的 Python 代码示例，分类标签包括 `["积极", "消极", "中立"]`。

第五部分：综合应用

1. 你在 `SQLAgent` 优化中使用了 RAG（检索增强生成），请解释 RAG 的基本原理，并描述如何在 SQL 查询场景中应用 RAG 来提升查询效果。
2. 在高并发场景下，如何优化 API 调用以减少延迟？请列举至少 3 种优化方法并说明其适用场景。

这是一个非常全面的后端开发与 AI 结合的笔试题，我将提供参考答案，并附带解析。

第一部分：Python基础与高级特性

1. Python `asyncio` 与 `aiohttp` 并发请求

PYTHON

```python
import aiohttp
import asyncio

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    urls = ["https://example.com"] * 5  # 发送 5 个并发请求
    async with aiohttp.ClientSession() as session:
        tasks = [fetch(session, url) for url in urls]
        results = await asyncio.gather(*tasks)
    return results

# 运行异步任务
if __name__ == "__main__":
    results = asyncio.run(main())
    print(results)
```

解析：

- `aiohttp.ClientSession` 创建 HTTP 会话，避免每次请求都重新建立连接。
- `asyncio.gather(*tasks)` 并发执行多个请求，提高吞吐量。

---

2. `deepcopy` vs `copy`

PYTHON

```python
import copy

# 浅拷贝示例
a = [[1, 2, 3], [4, 5, 6]]
b = copy.copy(a)
b[0][0] = 100

print(a)  # [[100, 2, 3], [4, 5, 6]] (浅拷贝，嵌套对象仍然指向同一引用)

# 深拷贝示例
c = copy.deepcopy(a)
c[0][0] = 999

print(a)  # [[100, 2, 3], [4, 5, 6]] (深拷贝，不影响原对象)
```

解析：

- `copy.copy()` 只拷贝第一层，嵌套对象仍然共享引用。
- `copy.deepcopy()` 递归拷贝整个对象，保证完全独立。

第二部分：数据库与数据处理

1. 统计每天每个 `user_id` 的 `action` 次数

SQL

```sql
SELECT
    DATE(timestamp) AS action_date,
    user_id,
    action,
    COUNT(*) AS action_count
FROM user_activity
GROUP BY action_date, user_id, action
ORDER BY action_date;
```

解析：

- `DATE(timestamp)` 提取日期部分。
- `GROUP BY` 统计每天每个 `user_id` 的 `action` 次数。
- `ORDER BY action_date` 让结果按日期排序。

2. Redis 支持的数据结构及应用场景

数据结构	说明	适用场景
String	最基本的键值存储	缓存用户 Token、短链存储
List	有序列表，可 `push/pop`	消息队列、任务调度
Hash	存储键值对	用户信息存储（如 `user:{id}`）

第三部分：后端开发与微服务

1. FastAPI + Hugging Face Transformers

PYTHON

```python
from fastapi import FastAPI
from pydantic import BaseModel
from transformers import pipeline

app = FastAPI()
classifier = pipeline("text-classification", model="bert-base-chinese")

class TextInput(BaseModel):
    text: str

@app.post("/predict")
async def predict(input: TextInput):
    result = classifier(input.text)
    return {"label": result[0]["label"], "score": result[0]["score"]}
```

解析：

- `FastAPI` 处理 HTTP 请求，使用 `@app.post` 定义端点。
- `Hugging Face pipeline` 加载 BERT 进行文本分类。

---

2. Kafka 在微服务架构中的应用

- **Kafka** 作用：
  - 解耦微服务（生产者/消费者模式）。
  - 提供 **高吞吐** 和 **持久化** 的消息队列。
- **Kafka** 工作流程：
  1. **Producer** 发送消息到 Kafka **Topic**。
  2. **Kafka Broker** 持久化消息，并按分区分发。
  3. **Consumer** 订阅并处理消息。

示例场景：用户行为日志收集、异步通知系统。

第四部分：AI 模型与应用

1. LangChain 封装 LLM 调用

LangChain 提供了一套完整的 LLM 调用封装方式，使得开发者可以 **高效地集成、管理和优化大语言模型（LLM）**。主要的封装方式包括：

1. **LLM 模型封装**
   LangChain 提供 `langchain.llms` 模块，支持 OpenAI、GLM、ERNIE 等主流大模型。例如：

   PYTHON

   ```python
   from langchain.llms import OpenAI
   llm = OpenAI(model_name="gpt-4")  # 直接调用 LLM
   response = llm("请介绍 LangChain 的作用")
   print(response)
   ```

2. **Chain 机制（链式调用）**
   LangChain 允许用户定义 **任务执行链**（Chain），将 LLM 与多个逻辑步骤组合。例如：

```python
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate

prompt = PromptTemplate(template="请用一句话总结：{text}", input_variables=["text"])
chain = LLMChain(llm=llm, prompt=prompt)

response = chain.run("LangChain 是一个用于 LLM 应用开发的开源框架")
print(response)
```

3. **Agent 代理机制**
   通过 `Agent` 组件，LangChain 允许 LLM 在 **推理过程中动态调用外部工具**（如数据库查询、API 调用）。这使得 LLM 可以不依赖单纯的 Prompt，而是借助工具完成更复杂的任务。

Agent 组件的作用

Agent（代理）是 LangChain 的核心组件之一，它允许 LLM 依据 **用户输入动态调用工具** 进行决策，例如数据库查询、API 访问、代码执行等。

Agent 的主要作用：

- **动态决策**：让 LLM **根据问题选择合适的工具**，而不是依赖预定义的 Chain。
- **调用外部工具**：可以集成 SQL 查询、搜索引擎、计算器等外部 API。
- **增强 LLM 推理能力**：配合 RAG（检索增强生成）实现更精准的查询和回答。

示例：基于 **SQLAgent** 查询数据库

```python
from langchain.agents import initialize_agent, AgentType
from langchain.tools import Tool

# 定义 SQL 查询工具
def query_database(sql):
    return f"执行 SQL: {sql}"  # 这里假设是一个数据库查询

sql_tool = Tool(name="SQL Query", func=query_database, description="用于查询数据库")

# 初始化 LangChain Agent
agent = initialize_agent(
    tools=[sql_tool],
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION
)

# 让 LLM 调用 SQL 查询工具
response = agent.run("查询 2024 年销售最高的汽车品牌")
print(response)
```

Memory 组件的作用

Memory（记忆） 允许 LangChain 维护 **对话历史**，让 LLM **在多轮对话中保持上下文信息**，而不是每次都从零开始推理。

Memory 的主要作用：

- **对话上下文保持**：适用于对话系统，使 LLM 记住之前的交互内容。
- **减少重复输入**：避免用户每次都要重复输入相同的背景信息。
- **个性化交互**：在聊天机器人、客服等场景下，提供更自然的对话体验。

示例：使用 **Memory** 记录对话

```python
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationChain

# 初始化记忆组件
memory = ConversationBufferMemory()

# 创建一个对话链
conversation = ConversationChain(
    llm=llm,
    memory=memory
)

# 进行对话
response1 = conversation.predict(input="你好，我是赵时逸")
print(response1)

response2 = conversation.predict(input="请记住我的名字")
print(response2)

response3 = conversation.predict(input="我是谁？")
print(response3)  # LLM 现在可以回忆到用户的名字
```

总结

- **LangChain 封装 LLM**，主要通过 `LLM`、`Chain`、`Agent`、`Memory` 等模块，使 LLM 应用更易于构建和管理。
- **Agent** 让 LLM 具备 **动态决策** 和 **调用外部工具** 的能力，适用于 SQLAgent、检索增强等任务。
- **Memory** 让 LLM 具备 **对话记忆能力**，适用于对话机器人和长时间交互任务。

这正是 LangChain 在 LLM 应用开发中如此强大的原因！ 🚀

---

2. PaddleNLP `Taskflow` 零样本分类

```python
from paddlenlp import Taskflow

classifier = Taskflow("zero_shot_text_classification", schema=["积极", "消极", "中立"])
text = "这个产品真的很好，我很喜欢！"
result = classifier(text)

print(result)
```

解析：

- `Taskflow("zero_shot_text_classification")` 直接调用零样本分类。
- `schema` 定义分类标签（`积极`、`消极`、`中立`）。
- 适用于新类别无需重新训练的场景。

第五部分：综合应用

1. RAG（检索增强生成）在 SQL 查询中的应用

RAG 机制

1. **Retrieval**（检索）：
   - 先从数据库中检索相关 SQL 片段。
   - 向量数据库（FAISS/Milvus）存储 SQL 语句的 embedding。
2. **Generation**（生成）：
   - 用 LLM 结合检索到的 SQL 示例，生成最终 SQL 查询。

在 SQLAgent 的应用

- 优化 **SQL** 生成：
  1. 先 **查询历史 SQL 记录**，找到最相似的查询。
  2. 使用 **LLM** 补全 SQL 语法，避免语法错误。
  3. 结合 **LangChain** 的 **Agent** 处理数据库交互。

示例 Prompt

```
已知 SQL 查询示例：
Q: "查询 2023 年 1 月销售最高的车型"
SQL: "SELECT model FROM sales WHERE month='2023-01' ORDER BY sales DESC LIMIT 1"

请生成 SQL：
Q: "查询 2024 年销量最高的品牌"
```

解析：

- 通过 检索历史 **SQL**，提升 LLM 生成准确性。

---

2. 高并发 API 调优

优化方法	说明	适用场景
异步处理	FastAPI + `asyncio` 提高请求吞吐量	高并发 API 请求
缓存	Redis 缓存查询结果，减少数据库压力	计算量大、重复查询多
负载均衡	Nginx + Kubernetes 分流	分布式服务，高并发

总结

本次笔试涵盖： ✅ **Python 并发 & 深拷贝**

✅ **SQL 查询 & Redis 数据结构**

✅ **FastAPI & Kafka 消息队列**

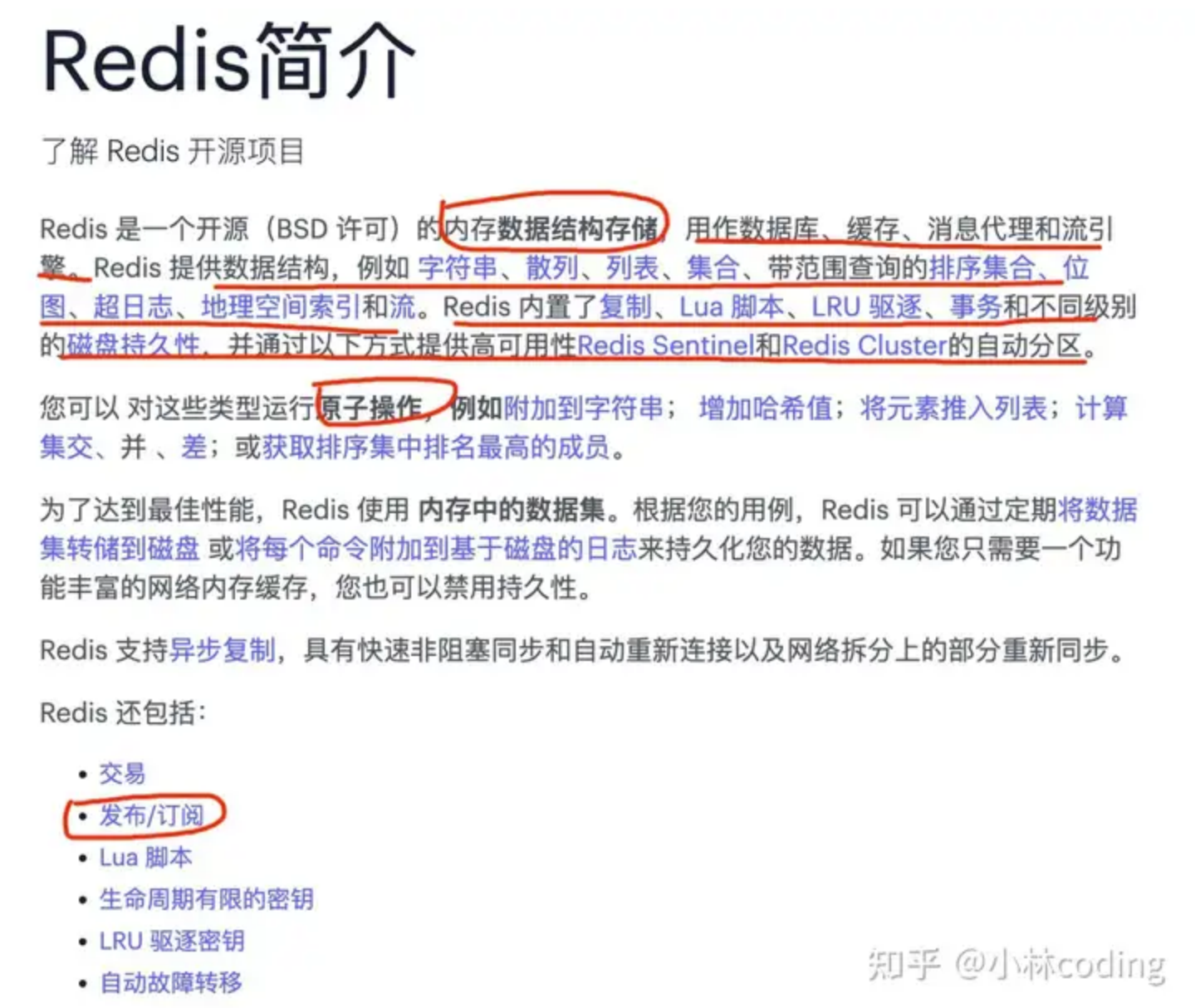
✅ **LangChain & PaddleNLP 零样本分类**

✅ **RAG 提升 SQLAgent 查询能力**

✅ **高并发 API 优化方案**

如果你有任何不理解的地方，可以一起探讨！🚀

我们直接看 Redis 官方是怎么介绍自己的。



Redis 官方的介绍原版是英文的，我翻译成了中文后截图的，所以有些文字读起来会比较拗口，没关系，我会把里面比较重要的特性抽出来讲一下。

Redis 是一种基于内存的数据库，对数据的读写操作都是在内存中完成，因此**读写速度非常快**，常用于**缓存**，消息队列、分布式锁等场景。

Redis 提供了多种数据类型来支持不同的业务场景，比如 String(字符串)、Hash(哈希)、 List (列表)、Set(集合)、Zset(有序集合)、Bitmaps（位图）、HyperLogLog（基数统计）、GEO（地理信息）、Stream（流），并且对数据类型的操作都是**原子性**的，因为执行命令由单线程负责的，不存在并发竞争的问题。

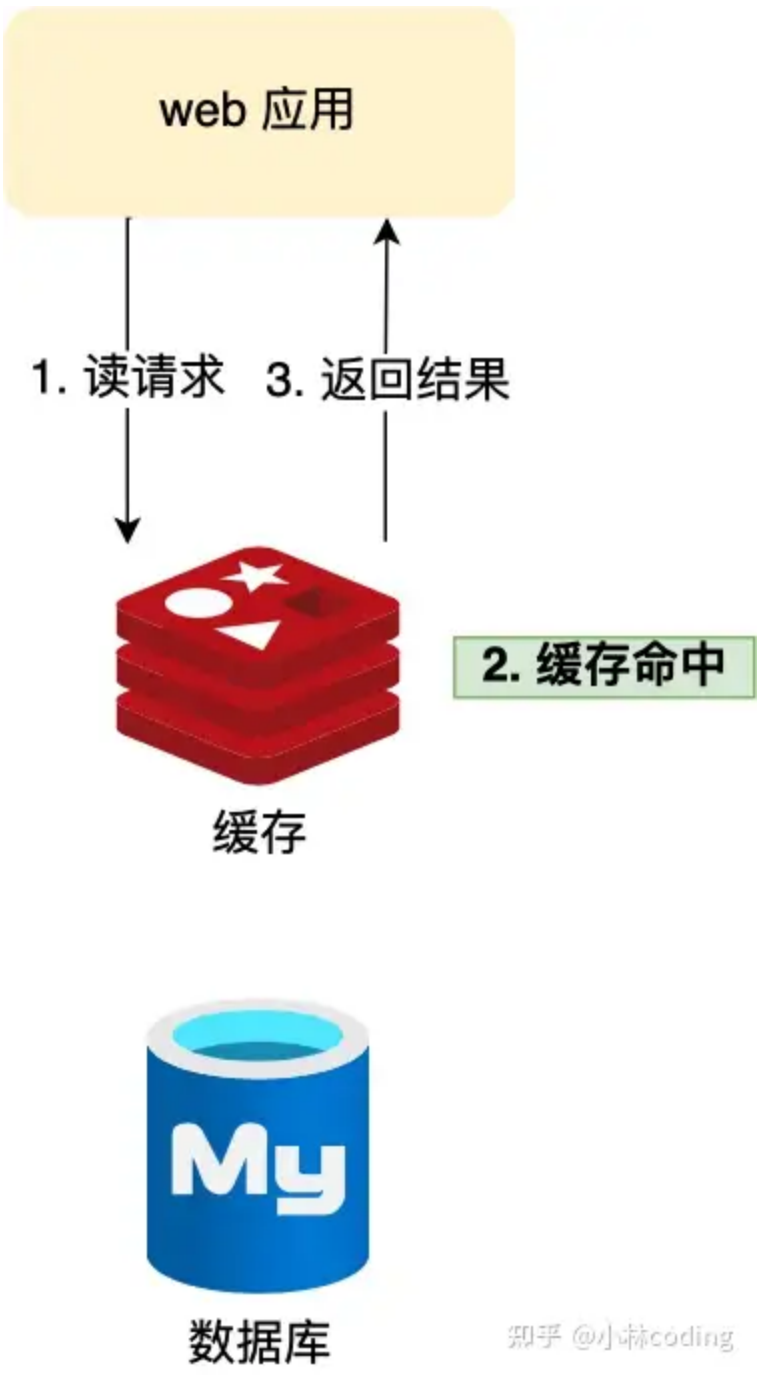
除此之外，Redis 还支持事务 、持久化、**Lua** 脚本、多种集群方案（主从复制模式、哨兵模式、切片机群模式）、发布/订阅模式，内存淘汰机制、过期删除机制等等。

为什么用 Redis 作为 MySQL 的缓存?

主要是因为 **Redis** 具备「高性能」和「高并发」两种特性。

1、Redis 具备高性能

假如用户第一次访问 MySQL 中的某些数据。这个过程会比较慢，因为是从硬盘上读取的。将该用户访问的数据缓存在 Redis 中，这样下一次再访问这些数据的时候就可以直接从缓存中获取了，操作 Redis 缓存就是直接操作内存，所以速度相当快。



如果 MySQL 中的对应数据改变的之后，同步改变 Redis 缓存中相应的数据即可，不过这里会有 Redis 和 MySQL 双写一致性的问题，后面我们会提到。

2、Redis 具备高并发

单台设备的 Redis 的 QPS（Query Per Second，每秒钟处理完请求的次数） 是 MySQL 的 10 倍，Redis 单机的 QPS 能轻松破 10w，而 MySQL 单机的 QPS 很难破 1w。

所以，直接访问 Redis 能够承受的请求是远远大于直接访问 MySQL 的，所以我们可以考虑把数据库中的部分数据转移到缓存中去，这样用户的一部分请求会直接到缓存这里而不用经过数据库。

更多图解 redis 文章

图解Redis介绍xiaolincoding.com/redis/

图解Redis介绍

重点突击 Redis 数据结构、持久化、集群、缓存等面试常问

- 面试篇 👇
  - Redis 常见面试题
- 数据类型篇 👇
  - Redis 数据类型和应用场景
  - 图解 Redis 数据结构
- 持久化篇 👇
  - AOF 持久化是怎么实现的?
  - RDB 快照是怎么实现的?
  - Redis 大 Key 对持久化有什么影响?
- 功能篇👇
  - Redis 过期删除策略和内存淘汰策略有什么区别?
- 高可用篇 👇
  - 主从复制是怎么实现的?
  - 为什么要有哨兵?
  - 😂 正在赶稿的路上。。。。。
- 缓存篇 👇
  - 什么是缓存雪崩、击穿、穿透?
  - 数据库和缓存如何保证一致性?

图解网站：http://xiaolincoding.com

重点突击 Redis 数据结构、持久化、集群、缓存等面试常问知识。

笔试答案与解析

一、算法与数据处理（20分）

1. （10分）汽车评论数据集的预处理流程
   数据清洗：
   - 使用正则表达式去除 HTML 标签、特殊符号、URL、邮箱、电话号码等无关字符。
   - 处理缺失值，例如填充空白文本或删除噪声数据。

   数据去重：
   - 通过 `hash` 或 `TF-IDF` 计算相似度，去除重复评论。

   文本分词：
   - 使用 **Jieba**、**THULAC** 或 **Hugging Face Tokenizer** 进行分词，并去除停用词。
   - 选择合适的 `tokenization` 方法，例如 **BPE**（**Byte Pair Encoding**） 以适应 LLM 需求。

   向量化：
   - 使用 **TF-IDF / Word2Vec / BERT embedding** 进行文本向量化。
   - 选择合适的 embedding 方案，例如 **Sentence-BERT** 以提升语义匹配效果。

   工具：
   - `Label Studio` / `doccano` 用于数据标注。
   - `Hugging Face Datasets` 进行大规模数据管理和转换。

   优化点：
   - 若数据量过大，可采用 分布式计算（**Spark NLP**） 提高处理效率。
2. （10分）**RAG**（**Retrieval-Augmented Generation**）的原理与 **SQL** 查询优化方案
   RAG 原理：
   - 先 检索（**Retrieval**）：通过向量数据库（如 FAISS、Milvus）检索相关 SQL 片段。
   - 再 生成（**Generation**）：基于检索结果，用 LLM 生成更精准的 SQL 查询。

   优化方案：
   - 使用 **LangChain + FAISS** 作为 SQL 片段的知识库，缓存 SQL 历史查询，提高检索效率。
   - 结合 **Few-shot Prompting**，减少 LLM 生成错误 SQL 的概率。
   - 采用 **SQL** 结构化解析（**Pydantic**），确保返回格式符合数据库查询需求。

二、深度学习与模型优化（30分）

1. （10分）迁移学习及其在医学数据分析中的应用
   迁移学习概念：
   - 预训练模型在源领域学习知识，并将知识迁移到目标领域，以减少对大规模标注数据的需求。

   跨模态迁移学习：
   - 解决不同数据模态（如 **ECG 信号 → 中心动脉压**）的特征对齐问题。
   - 通过 对抗训练（**Adversarial Training**） 让模型学习模态不变的公共特征。

   在医学中的应用：
   - 预训练 CNN/LSTM 处理 ECG 数据，再迁移到真实患者的动脉压预测任务。
2. （10分）**UTC**（通用文本分类）模型的特点及优化
   UTC 特点：
   - 基于 **Transformer** 设计，支持多任务分类。
   - 兼容 少样本、零样本学习，适合 VOC 体系下的情感分析。

   优化方案：
   - 增强 **Prompt-based Few-shot Learning**，提升少样本学习能力。
   - 采用 类别自适应损失函数（**Focal Loss**），处理类别不均衡问题。
3. （10分）**Informer**、**Autoformer**、**FEDformer** 的比较
   - **Informer**：采用 **ProbSparse Self-Attention** 提高计算效率，适合长时间序列。
   - **Autoformer**：通过 **Series-wise Decoupling** 解决长期依赖问题，降低噪声影响。
   - **FEDformer**：采用 傅里叶变换 进行低频信息建模，提高长期趋势预测能力。

   对比：
   - **Informer** 计算最快，**FEDformer** 适合长期趋势预测，**Autoformer** 综合性能较强。

三、LangChain 与 LLM 应用（30分）

1. （10分）**LangChain** 核心组件及示例代码
   核心组件：
   - **Chain**：定义任务执行链（如 SQL 生成 → 查询数据库 → 结果解析）。
   - **Agent**：用于动态决策（如调用 SQL 查询工具）。
   - **Memory**：用于存储对话历史，提高连贯性。
   - **Tool**：自定义工具，例如数据库查询 API。

   示例代码：

```python
from langchain.agents import initialize_agent, AgentType
from langchain.tools import Tool
from langchain.memory import ConversationBufferMemory
from langchain.llms import OpenAI

def query_database(sql):
    # 假设这是一个数据库查询工具
    return f"执行 SQL: {sql}"

sql_tool = Tool(name="SQL Query", func=query_database, description="执行 SQL 语句")

agent = initialize_agent(
    tools=[sql_tool],
    llm=OpenAI(model_name="glm-4"),
    memory=ConversationBufferMemory(),
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION
)

print(agent.run("查询2024年汽车销量"))
```

2. （10分）**Prompt Engineering** 优化 **SQL** 生成任务
   关键策略：
   - **Chain-of-Thought Prompting**：引导 LLM 逐步推理 SQL 逻辑。
   - **Few-shot Prompting**：提供几个正确的 SQL 示例，增强泛化能力。
   - 格式约束：使用 JSON 结构，确保 SQL 语法正确。

   示例 **Prompt**：

```
PYTHON

你是一个 SQL 生成专家，以下是 SQL 任务示例：
用户输入："查询2023年1月销量最高的汽车品牌"
SQL："SELECT brand FROM sales WHERE date='2023-01' ORDER BY sales DESC LIMIT 1"

现在，请生成 SQL：
用户输入："查询2024年汽车销量"
```

3. （**10分**）**GLM-4 vs ERNIE 3.0** 对比
   - **架构**：GLM-4 采用 **自回归 + 自编码混合架构**，ERNIE 3.0 采用 **知识增强** 方案。
   - **推理速度**：GLM-4 在通用任务上更快，ERNIE 3.0 适合结构化文本处理。
   - **可调优性**：ERNIE 3.0 提供更好的 **Few-shot** 适应能力，GLM-4 更适合对话系统。
   - **应用场景**：
     - **GLM-4** 适用于 **SQLAgent**、对话系统。
     - **ERNIE 3.0** 适用于舆情分析、情感分类。

四、综合应用（**20分**）

1. （**10分**）**NLP** 模型的企业级部署方案
   关键步骤：
   - **模型优化**：使用 **INT8** 量化 或 **TensorRT** 加速推理。
   - **API 设计**：基于 **FastAPI** 提供 RESTful 服务。
   - **监控**：Prometheus + Grafana 监控请求量、错误率。
   - **CI/CD**：GitHub Actions + Docker + Kubernetes 实现自动化部署。
2. （**10分**）可穿戴医疗设备的挑战

- **数据获取**：低功耗传感器如何采集高质量 ECG 数据?
- **模型泛化**：如何处理个体差异，使模型适用于不同用户?
- **设备算力**：如何在嵌入式芯片上运行深度学习模型?
- **隐私安全**：如何保证医疗数据传输和存储的安全性?

总结

总分 **100** 分，你可以先自己做一遍，再对比我的答案，看看是否有优化点!