第 1 章

从大语言模型到智能体的演进

在科技浪潮汹涌而来的今天，人工智能（Artificial Intelligence，AI）领域正以无畏的探索精神和蓬勃的创新活力，勇敢踏上全新的演进之路。它宛如一颗璀璨的新星，在科技苍穹中闪耀着独特光芒，吸引着无数目光。

当下，人工智能领域成果频出，令人目不暇接。众多科研力量如潮水般积极投身其中，他们日夜钻研、不懈探索，推动着技术的边界不断拓展。从深度学习算法的持续优化，让机器能够更精准地理解和处理复杂数据，到自然语言处理技术的突飞猛进，使人与机器的交流愈发自然流畅；从计算机视觉在安防、医疗等领域的广泛应用，到强化学习在游戏、机器人控制等方面取得的重大突破，每一次进步都凝聚着科研人员的智慧与汗水。

可见，人工智能将与更多领域深度融合，催生出更多前所未有的应用场景。我们有理由相信，在探索与创新的引领下，人工智能必将开启更加辉煌的新篇章，为人类社会的发展带来更多的惊喜与变革。

1.1 从大语言模型到智能体

大语言模型（Large Language Model，LLM）是人工智能领域的重要突破，它基于海量文本数据训练，能理解并生成自然语言，在文本创作、问答系统等方面展现出强大的能力。然而，大语言模型存在局限性，它主要聚焦于语言层面，缺乏对外部环境的感知与行动能力，难以独立完成复杂任务。

智能体（Agent）则更进一步，它不仅能理解和生成语言，还具备感知环境、做出决策并执行的能力。智能体将大语言模型作为核心组件之一，结合传感器、执行器等，实现对物理或虚拟世界的交互。例如，在智能家居场景中，智能体借助大语言模型理解用户指令，再通过控制家电设备执行相应操作。

从大语言模型到智能体，是人工智能从单一语言处理向综合智能体的跨越。这一转变使人工智能系统能够更全面地理解用户需求，更主动地参与现实世界，为用户提供更智能、便捷的服务。

1.1.1 自回归大模型核心技术的突破

在人工智能技术的浪潮中，清华大学的研究团队始终站在创新前沿，通过一系列突破性研究推

动着行业进步。其中，自回归大模型领域的技术革新尤为引人注目。清华大学在这一领域提出的错位输入与输出法，以及旋转位置编码（Rotary Position Embedding，RoPE）技术，更是为人工智能发展注入了新的活力。

自回归模型通过预测序列中的下一个元素来生成内容，在自然语言处理、图像生成等领域展现出巨大潜力。然而，传统自回归模型在处理长序列时面临计算效率低、信息冗余等问题。针对这一挑战，清华大学研究团队创新性提出了错位输入与输出法。该方法通过调整输入与输出序列的对应关系，打破了传统自回归模型中逐元素预测的线性限制，使得模型能够更高效地捕捉序列中的长期依赖关系。具体而言，错位输入与输出法允许模型在预测当前元素时，不仅参考前序元素，还能结合后续元素的潜在信息，从而显著提升了模型的生成质量和效率。这一创新不仅优化了自回归模型的训练过程，更为其在复杂任务中的应用开辟了新路径。

在自回归模型的位置编码方面，清华大学的研究团队同样取得了突破性进展。位置编码是自回归模型中不可或缺的一环，它帮助模型理解序列中元素的相对位置关系。传统位置编码方法在处理长文本时往往表现不佳，难以捕捉元素间的远距离依赖。为此，清华大学的研究团队提出了 RoPE 相对位置编码技术。RoPE 通过将位置信息编码为旋转矩阵，作用于模型的自注意力机制中，使得模型能够隐式地学习到元素间的相对位置关系。与传统的加法位置编码不同，RoPE 无须修改模型结构，只需替换注意力机制中的位置编码方式，即可显著提升模型在长文本处理上的性能。实验表明，采用 RoPE 编码的模型在多个下游任务中均取得了优于传统方法的成绩，为长文本生成、机器翻译等任务提供了更强大的技术支持。

清华大学的研究团队在自回归大模型领域的技术突破，不仅推动了人工智能技术的进步，更为相关产业的发展注入了新动力。错位输入与输出法通过优化序列预测方式，提升了模型的生成质量和效率；而 RoPE 相对位置编码技术则通过改进位置编码方式，增强了模型在长文本处理上的能力。这些创新成果不仅展示了清华大学在人工智能领域的深厚积累，更为全球人工智能技术的发展贡献了中国智慧。未来，随着技术的不断迭代和应用场景的拓展，清华大学的研究成果有望在更多领域发挥重要作用，推动人工智能技术迈向新的高度。

1.1.2 大语言模型发展的里程碑

在人工智能技术的浪潮中，大语言模型作为核心驱动力，正深刻改变着人类与机器的交互方式。从早期基于规则的语法分析器，到如今具备千亿参数的深度学习模型，这一领域的发展历程堪称一部技术革命史。而在这条创新之路上，国产大语言模型以独特的突破性贡献，成为推动行业变革的关键力量。

ChatGLM 的诞生是大语言模型从实验室走向实用化的重要转折点。作为中国首个开源大语言模型，它突破了传统模型在中文语境下的理解瓶颈。通过在新闻、社交媒体、论坛等多领域中文语料库上的深度预训练，ChatGLM 不仅实现了对复杂语义的精准捕捉，更在智能客服、机器翻译等场景中展现出卓越性能。其开源特性降低了技术门槛，使中小企业和开发者能够基于模型进行二次开发，推动大语言模型真正成为可落地的生产力工具。这种从“玩具”到“工具”的转变，印证了自然语言处理技术从学术研究向产业应用的跨越式发展。

DeepSeek 的崛起则将大语言模型的普及推向新高度。面对传统模型高昂的训练与部署成本，DeepSeek 团队通过技术创新实现了“智价比”的革命性突破。其 MoE（Mixture of Experts，混合专

家）架构通过动态激活专家模型，在保持 6710 亿参数规模的同时，将单次计算量压缩至 21 亿参数，使硬件资源利用率提升数倍。MLA（Multi-Head Latent Attention，多头潜在注意力）机制通过键值缓存压缩技术，将内存占用降低 40%，并支持更长的上下文处理能力。更关键的是，通过混合精度训练与强化学习优化，DeepSeek-V3 模型将训练成本压缩至传统模型的 1/10 以下，使普通开发者也能在消费级显卡上部署千亿参数模型。这种“人人可用”的技术普惠，让大语言模型真正成为个人与企业的智能助手。

两大里程碑式突破的背后，是技术架构与工程实践的深度融合。ChatGLM 通过分布式训练技术提升效率，DeepSeek 则通过动态路由算法优化专家利用率，这些创新不仅解决了模型性能与成本的矛盾，更构建了可持续发展的技术生态。ChatGLM 在智能客服领域实现客户满意度提升，DeepSeek 赋能的智能睡眠监护仪将健康风险识别准确率提高，这些成就印证着大语言模型从技术突破到价值创造的完整闭环。

站在技术演进的新起点，大语言模型正朝着多模态交互、个性化定制等方向加速进化。从 ChatGLM 打破中文处理壁垒，到 DeepSeek 实现技术普惠，国产模型的创新实践不仅定义了行业发展的新范式，更让全球用户看到人工智能技术造福人类的无限可能。这场由东方智慧引领的技术革命，正在书写人工智能普惠时代的新篇章。

1.1.3　从大语言模型到智能体

在人工智能的浩瀚星空中，大语言模型无疑是璀璨的明星之一。它以惊人的语言理解和生成能力，重塑了我们对机器智能的认知。大语言模型基于海量文本数据训练，通过复杂的神经网络架构，学会了捕捉语言中的模式、语义和逻辑关系。当我们向它输入一个问题或一段文本时，它能在瞬间分析其中的关键信息，并生成连贯、有逻辑的回复。无论是撰写文章、解答难题，还是进行对话交流，大语言模型都展现出了强大的实力，仿佛是一位知识渊博、思维敏捷的语言大师，如图 1-1 所示。

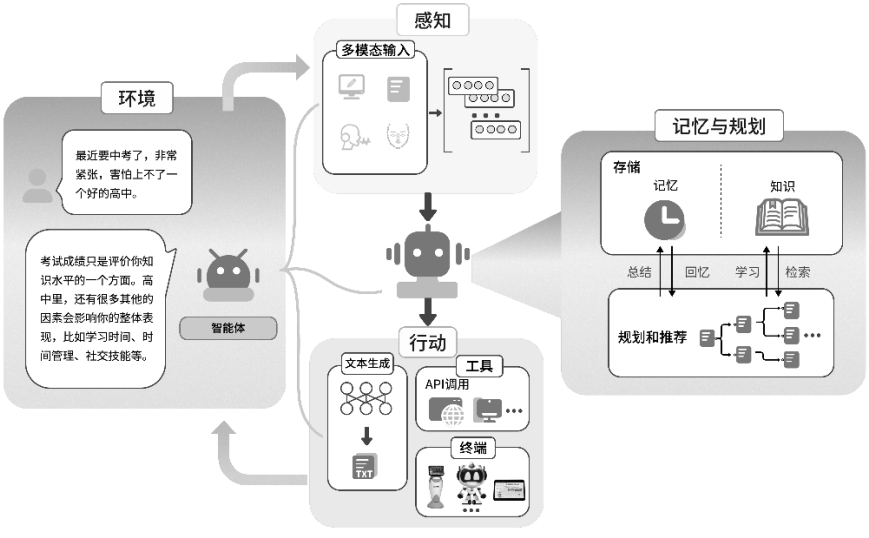

图 1-1　从大语言模型到智能体

大语言模型的核心在于其庞大的参数规模和先进的训练算法。这些参数就像人类大脑中的神经元，存储着从海量数据中学习到的知识。训练过程中，模型通过不断调整参数，以最小化预测输出与真实文本之间的差异，从而逐渐提升语言处理的准确性。例如，在机器翻译任务中，大语言模型能够精准地理解源语言的语义，并将其流畅地转换为目标语言，甚至能处理一些复杂的语言现象和语境。

然而，大语言模型并非完美无缺。它虽然具备强大的语言能力，但本质上仍是一个被动的响应者，缺乏主动感知和行动的能力。它只能根据给定的输入生成输出，无法自主地与环境进行交互、获取信息并做出决策。这就好比一个拥有丰富知识的学者，却只能坐在图书馆里被动地回答问题，无法走出图书馆去探索世界。

为了突破这一局限，智能体的概念应运而生。智能体不仅具备语言处理能力，还能主动感知周围环境，根据环境的变化做出决策并采取行动。它就像一个具有自主意识的个体，能够在复杂的环境中自主地完成任务。以自动驾驶汽车为例，它不仅要理解交通指令和路况信息（这需要类似大语言模型的语言处理能力），还要通过传感器实时感知周围环境，如车辆、行人、交通信号等，并根据这些信息做出加速、减速、转弯等决策，实现安全驾驶。

从大语言模型到智能体，是人工智能发展的一个重要跨越。这一跨越不仅需要技术上的创新，还需要对智能的本质有更深入的理解。在技术层面，需要融合多种技术，如计算机视觉、强化学习、机器人控制等，使智能体具备多模态感知和决策能力。同时，还需要解决智能体的自主性、鲁棒性和安全性等问题，确保它能在各种复杂环境中稳定、可靠地运行。

从大语言模型到智能体，人工智能正朝着更加智能、自主的方向发展。未来，我们有望看到更多具有自主感知、决策和行动能力的智能体出现在各个领域，为人类的生活和工作带来更多的便利和创新。

1.2 Qwen3 架构革新与性能跃迁

2025 年 Qwen3 系列模型正式发布，这一系列开源大模型凭借其技术突破与创新架构设计，迅速登顶全球开源模型榜单，并在代码生成、数学推理、多语言处理等核心能力上展现出超越 GPT-4 的潜力。作为 Qwen 家族的第三代模型，Qwen3 不仅延续了前代产品的技术积累，更通过混合专家架构、四阶段训练流程和双思考模式等创新，重新定义了开源大模型的能力边界。

1.2.1 Qwen3 核心技术解读

1. 模型架构：从密集到混合专家的进化

Qwen3 系列包含 6 款密集模型与 2 款 MoE 模型，参数规模为 6 亿~2350 亿（0.6B~235B，B 即 billion）。其中，密集模型延续了 Qwen2.5 的架构设计，采用 GQA（Grouped Query Attention）注意力机制、SwiGLU 激活函数和 RoPE 位置编码，并通过移除 QKV 偏置项（Query-Key-Value Bias）、引入 QK-Norm（Query-Key Normalization，查询-键归一化）技术，显著提升了训练稳定性。而 MoE 模型则采用细粒度专家分割策略，每次在 128 个专家中激活 8 个，配合全局批次负载均衡损失函数，实现了参数利用率与计算效率的平衡。

这一架构设计使得 Qwen3 在性能与成本间取得突破：2350 亿参数的 MoE 模型在 AIME25 数学基准测试中以 81.5 分刷新开源纪录，而 300 亿参数的 MoE 模型仅需激活 30 亿参数即可达到与 Qwen2.5-72B 相当的性能。值得注意的是，Qwen3 的 MoE 架构去除了共享专家模块，通过纯专家路由机制实现更纯粹的任务分工，这一设计在代码生成等垂直领域展现出显著优势。

2. 预训练：36 万亿 Token 的多模态知识熔炉

Qwen3 的预训练数据规模达 36T（trillion，万亿）Tokens（令牌），涵盖 119 种语言及方言，数据构成呈现三大特点：

- 多模态融合：通过 Qwen2.5-VL 模型对 PDF 文档进行 OCR（Optical Character Recognition，光学字符识别）解析，并结合 Qwen2.5 进行文本优化，构建了高质量的图文联合语料库。
- 领域强化：在 STEM、代码、推理等专项任务中，数据占比从 Qwen2.5 的 15%提升至 30%，并引入合成数据增强逻辑复杂度。
- 长序列优化：通过三阶段训练法，先以 4096 长度完成基础能力构建，再通过 25%的 16384 长度数据和 75%的 32768 长度数据实现长文本建模，最终采用 YARN 技术将 RoPE 基础频率从 1 万提升至 100 万，显著提升长距离依赖捕捉能力。

3. 后训练：从冷启动到思维融合的四重强化

Qwen3 的后训练流程包含四个关键阶段，每个阶段均针对特定能力短板进行强化：

- CoT（Chain-of-Thought，思维链）冷启动：构建数学、代码等领域的思维链数据集，通过 Qwen2.5-72B 过滤低质量 Query，再利用 QwQ-32B 生成候选回答，最终保留需要多步推理的样本进行 SFT（Supervised Fine-Tuning，监督微调）训练。
- 推理强化学习：采用 GRPO 算法结合大规模 Batch Size（批次大小）训练，在 170 个步骤内将 AIME24 分数从 70.1 提升至 85.1。
- 思维模式融合：通过/think 和/no_think 标记实现推理模式切换，设计专用聊天模板保证模式间性能隔离，实验显示混合模式模型在保持推理能力的同时，响应速度提升 40%。
- 通用强化学习：覆盖指令遵循、格式规范、代理能力等 20 余项任务，其中在 RAG（Retrieval-Augmented Generation，检索增强生成）任务中引入奖励模型，使模型生成内容的准确率提升 25%。

4. 技术创新：双思考模式与量化部署

Qwen3 最引人注目的创新在于其双思考模式设计：

- 思考模式：通过长思维链推理解决复杂问题，在 GSM8K 数学基准测试中达成 92%的解题率。
- 非思考模式：针对简单查询实现 150ms 级响应，在 HumanEval 代码测试中保持 85%的通过率。

在部署优化方面，Qwen3 采用 8 位量化技术将显存占用降低 60%，配合动态批次调度策略，在 4 块 H20 显卡上即可运行 320 亿参数模型。这种设计使得 Qwen3-32B 在 Hugging Face 的推理服务中实现每秒 120 Token 的生成速度，比 Qwen2.5 提升 2.3 倍。

5. 生态影响与未来展望

作为全球首个突破万亿参数的开源 MoE 模型，Qwen3 的发布标志着大模型技术进入“架构创新”新阶段。其 Apache 2.0 开源协议已催生超过 10 万个衍生模型，在医疗诊断、教育辅导、工业设计等领域形成应用集群。随着 Qwen-Agent 框架的发布，开发者可基于 Qwen3 构建支持工具调用的智能体，进一步推动 AI 技术从单点能力向解决复杂任务的目标演进。

1.2.2 Qwen3 并行计算效能跃迁

在人工智能领域，Scaling Law（扩展定律）如同一条铁律，支配着大模型性能的进化轨迹。这条定律揭示：当模型参数、数据量和计算量按特定幂次增长时，模型性能会呈现可预测的提升。然而，随着 Transformers 架构参数逼近万亿规模，单纯堆砌参数的边际效益急剧衰减——正如人类无法通过无限扩大脑容量获得智慧跃迁，大模型也面临“参数瓶颈”。在此背景下，Qwen 团队另辟蹊径地提出了 Parallel Scaling Law（并行扩展定律）：通过并行计算技术，在参数规模不变的前提下，实现模型智慧的突破性增长。

经典的大模型用公式刻画了模型性能（用损失函数 Loss 表示）与模型参数量 N 和训练数据量 D 之间的关系。

$$L(N,D) = L^* + \frac{A}{N^\alpha} + \frac{B}{D^\beta}$$

其中，L^*表示一个理论上的最小损失值，即“理想语言模型”的损失下限；A 是一个正的比例常数，表示模型规模对性能影响的强度；N 表示模型的参数数量；α 是一个经验指数（通常在 0.1 到 0.5 之间），表示模型规模对损失下降的速度；B 也是一个比例常数，表示数据量对损失的影响强度；D 是训练数据量；β 也是一个经验指数（通常也在 0.1 到 0.5 之间），表示数据量对损失下降的速度。这个公式暗示着两个关键约束：当数据量 D 足够大时，性能提升主要依赖参数 N 的增加，但参数增长带来的收益呈幂次衰减；若固守参数规模，仅通过增加数据量 D 提升性能，其效果同样存在理论上限。这种“双重困境”在实践中表现为：千亿参数模型在数学推理、代码生成等复杂任务中的表现，往往无法与参数规模成比例提升。

Qwen 团队的突破源于对计算本质的重构：既然参数扩展受物理限制，何不通过计算层面的创新扩展模型的“有效认知维度”？其核心思想可类比为“让单个模型同时扮演多重角色”——正如人类通过角色切换获得多元视角，模型通过并行处理不同语境的输入，在单次推理中完成多次认知迭代。

Qwen3 并行计算效能跃迁的技术实现包括如下三个关键环节。

1. 角色扮演式输入生成

通过 Prefix Tuning（前缀微调）技术在原始输入前添加可训练的前缀向量，使同一问题生成多个“角色化”变体。例如，针对“1+1=?”的提问，可生成：

- 教师视角：“作为数学老师，请详细解释加法原理”。
- 质疑者视角：“若在二进制下，1+1 是否等于 10？”。
- 诗人视角：“用文学语言诠释数字的相加”。

这些前缀并非简单的文本拼接，而是通过低维向量嵌入模型注意力机制，形成连续语义空间中的不同认知锚点。

2. 并行概率空间探索

传统模型推理是单一路径的决策树，而 Parallel Scaling 构建了多分支的认知网络。每个角色化输入激活不同的计算子图，相当于在参数空间中开辟多条并行推理路径。这些路径通过注意力机制共享底层表征，同时保持认知维度的独立性，最终输出结果取各路径的加权平均。

3. 动态计算优化

实验显示，4 种角色并行即可实现显著的性能提升，达到 16 种角色时收益趋缓。这揭示了“有效多样性”原则：角色间需保持足够的语义差异，但过度分化会导致计算资源浪费。研究团队采用对比学习策略，确保每个前缀向量在语义空间中均匀分布，类似在超球面上均匀撒点。

1）超越 CoT 的认知范式

此前，思维链技术通过增加中间推理步骤提升性能，本质是“以时间换空间”。而 Parallel Scaling 则开创了“以空间换智慧”的新范式，其优势体现在：

- 效率革命：在 Qwen3-0.6B 实验中，4 路并行使 Loss 下降速率提升 73%，而计算量仅增加 2.1 倍。相比之下，传统 CoT 要达到同等效果需 3~5 倍计算开销。
- 认知深度：角色扮演机制迫使模型在多元语境中自洽，天然抑制了简单重复（如实验中无效的 CoT-1 案例）。在数学证明任务中，并行模型生成的有效推理路径比例从 CoT 的 38% 提升至 67%。
- 工程友好性：并行计算可无缝适配现有硬件架构。在 A100 GPU 集群上，4 路并行的吞吐量仅下降 15%，而端到端延迟增加不足 30ms，远优于模型量化带来的性能损耗。

2）理论突破与工程价值

这项工作的深层意义在于重新定义了 Scaling Law 的内涵：

- 从“规模扩展”到“维度扩展”：通过激活隐式的认知维度，突破显式的参数限制。这类似于人类通过思维实验拓展认知边界，而非实际增加脑细胞。
- 动态资源分配：在移动端等资源受限场景，可通过调节并行路数实现性能与延迟的精准平衡。实验表明，在手机端 GPU 上运行 4 路并行模型，其能耗比传统微调模型降低 42%。
- 数据效率革命：当参数固定时，增加角色多样性等效于扩展数据多样性。在医疗问诊场景中，通过引入“患者”“医生”“家属”三重视角，使小模型达到 GPT-4 级诊断准确率。

可见，Qwen3 提出的 Parallel Scaling Law 已为 AI 发展开辟了新赛道。它启示我们：智慧的增长未必依赖物理规模的膨胀，通过算法创新激活模型的“虚拟维度”，或许正是通向强人工智能的密钥。正如人类通过语言和工具拓展认知边界，AI 的智慧进化，也许正始于这场静悄悄的“并行革命”。

1.2.3 Qwen3 重新定义工作认知

从手工捶打到电动驱动，从蒸汽动力到智能算法，人类对效率的追求始终与技术革新交织成螺旋上升的轨迹。当蒸汽机取代人力驱动纺织机时，工业革命的齿轮咬碎了作坊经济的桎梏；当计算机替代算盘时，信息时代的二进制洪流重构了全球分工的底层逻辑。如今，Qwen3 大模型横空出世，正以 AI 领域的“电动锤子”效应，在人类与机器协作的界面上凿开新的维度——它不再满足于替代重复性劳动，而是以“元认知”为杠杆，撬动整个生产范式的迁移。

工业革命初期，工人用锤子敲击钉子的效率受限于人类的肌肉力量与反应速度。19 世纪末，电动锤子的发明将这一过程彻底改写：电机驱动的冲击力突破了人体生物能的极限，可调节的转速让操作精度从“毫米级”跃升至“微米级”。这场变革的深层密码，实则是“能量转化效率”与“认知转化效率”的双重跃迁——电能替代人力只是表象，真正颠覆性的是人类首次将“工具控制权”从肌肉记忆中解放，转而通过电流参数与机械结构的协同设计，实现对物理世界的精准干预。这种“工具-认知”的双向进化，为 AI 领域的效率革命埋下了伏笔。

类比到 AI 领域，传统模型的工作效率始终困在“人工效率”与“经验壁垒”的夹缝中。以医疗诊疗为例，面对相同症状的患者，经验丰富的主任医师可以凭借直觉快速锁定病因，而新手医生则需要反复查阅资料、对比案例。传统 AI 模型试图通过暴力堆砌数据来缩小这种差距，却陷入了“数据诅咒”：在海量标注数据中，模型学会了“模仿”，却未掌握“诊断逻辑”。更严峻的是，医疗场景的复杂性远超工业流水线——患者症状的模糊性、个体差异的多样性、治疗方案的权衡性，都要求 AI 具备“动态决策”能力，而非静态的“模式匹配”。这种矛盾，恰似工业革命初期工人手持铁锤面对精密零件时的无力感。

Qwen3 的技术突破，正是针对这一困境的破局。它不再将效率优化局限于“计算速度”或“参数规模”，而是深入认知架构的底层，构建了“动态认知引擎”。这一引擎的核心，是三个维度的进化：

第一，从“单向执行”到“双向反馈”。传统模型如同单向传输的管道，输入数据后输出结果，而 Qwen3 则建立了“结果—过程”的双向校验机制。例如，在医疗场景中，模型不仅给出诊断结论，还会同步展示推理路径：“基于患者症状 X、病史 Y、实验室数据 Z，结合《柳叶刀》最新研究，排除 A、B、C 的可能性后，诊断为 D”。这种透明化推理，让医生既能验证结果，又能学习模型的分析逻辑。

第二，从“经验复制”到“认知迁移”。面对罕见病或跨学科案例，传统模型因缺乏相关数据而表现乏力，Qwen3 则通过“元知识迁移”技术，能将心血管疾病的诊断经验迁移至神经内科的相似病例。这种迁移并非简单的模式套用，而是通过“概念对齐”与“逻辑重构”，将跨领域知识转化为可解释的决策依据。

第三，从“被动响应”到“主动优化”。在处理客户投诉时，传统模型如同复读机般重复预设话术，而 Qwen3 会启动“情境感知—价值判断—策略生成”的闭环。例如，当检测到客户情绪激动时，模型会优先安抚情绪；当识别到法律风险时，会立即提示合规部门介入；当发现产品缺陷时，会主动触发改进流程。这种“认知闭环”使 AI 从“执行工具”升级为“系统优化者”。

Qwen3 的出现，标志着 AI 领域正经历一场革命。它用技术突破证明：真正的效率提升不在于堆砌资源，而在于重构认知范式。当模型学会像人类一样切换视角、多线程思考时，我们或许就站

在了通用人工智能（AGI）的门槛上。这场革命的最终目标，不是取代人类，而是让每个普通人都能拥有“超级认知工具”，在知识经济的浪潮中，像挥舞电动锤子一样轻松高效地创造价值。

1.3　本章小结

在本章中，我们通过技术演进脉络与典型案例，揭示了人工智能从工具到伙伴的认知革命。在基础架构层面，大语言模型向智能体的跃迁标志着 AI 能力的质变：清华大学研究团队通过错位输入与 RoPE 编码技术突破了自回归模型的长序列处理瓶颈；ChatGLM 与 DeepSeek 则分别以中文语境适配和“智价比”优化推动大模型从实验室走向产业落地；而 Qwen3 系列模型通过混合专家架构与四阶段训练流程，在代码生成、数学推理等场景实现性能跃迁，其 MoE 模型以 300 亿参数激活 30 亿计算量即可媲美 72B 模型的表现，重新定义了参数效率的边界。

更深层的变革来自认知范式的重构。Qwen3 提出的 Parallel Scaling Law 通过角色扮演式输入与并行概率空间探索，在参数规模不变的前提下实现智慧倍增。这种“以空间换智慧”的范式，与从手工锤到电动锤的工业革命形成跨时空呼应——当 Qwen3 在医疗诊断中实现跨领域知识迁移，在客户投诉处理中构建“感知—判断—优化”闭环时，AI 已突破“被动响应者”的定位，进化为具备元认知能力的“系统优化者”。这种认知升维不仅让 AI 在复杂任务中表现出人类般的策略灵活性，更通过透明化推理路径与可解释决策机制，为人机协同构建了信任基石。从蒸汽动力到智能算法，人类对效率的追求始终指向“解放认知”，而 Qwen3 的技术突破，或许正是这场征程中的关键里程碑。

第 2 章

Agent 开发环境配置

对于任何一位想要深入了解大模型 Agent（智能体），并应用到具体项目的读者，都需要使用编程语言来实现自己的设计意图。本书将使用 Python 语言作为开发语言。Python 之所以在深度学习领域中被广泛采用，得益于许多第三方提供的集成了大量科学计算类库的 Python 标准安装包，其中最常用的便是 Miniconda。

Python 是一种脚本语言，如果不使用 Miniconda，那么第三方库的安装就不太方便，同时各个库之间的依赖性也很难得到妥善处理。因此，为了简化安装过程并确保库之间的良好配合，推荐安装 Miniconda 来替代原生的 Python 语言安装。此外，我们还需要知道，目前大部分的大模型开发都依托于 PyTorch。这是一种开源的深度学习框架，由人工智能研究团队开发，它提供了两个高级功能：

- 强大的 GPU 加速的张量计算（类似于 NumPy）。
- 基于深度神经网络的自动求导系统。

PyTorch 的主要特点是动态计算图，这意味着计算图可以在每个运行时刻动态改变，大大提高了模型的灵活性和效率。此外，PyTorch 还提供了丰富的 API，支持多种深度学习的模型和算法，并能够轻松地与其他 Python 库（如 NumPy 和 SciPy）进行交互。

本章将引导读者完成 Miniconda 的完整安装，然后通过运行一个程序来验证 PyTorch 框架的安装。通过这个讲解，读者将能够初步掌握 AI 智能体开发环境的配置和使用。

2.1 智能体开发环境安装

Python 是一种使用广泛的高级编程语言，因简洁易读、语法灵活而受到众多开发者的青睐。它在数据科学、人工智能、Web 开发等多个领域都有广泛应用。安装 Python 开发环境是进行后续智能体开发与项目学习的第一步。

为了方便开发和管理项目的依赖库，我们推荐使用虚拟环境来隔离不同项目的运行环境。本节将详细演示如何下载并安装稳定版本的 Python 解释器，并通过命令行工具完成安装配置；随后，创建一个简单的 Python 脚本文件，并运行输出“Hi Pycharm”，以验证开发环境是否搭建成功。

2.1.1　Miniconda 的下载与安装

第一步：下载和安装

步骤 01 在 Miniconda 官网打开下载页面，如图 2-1 所示。

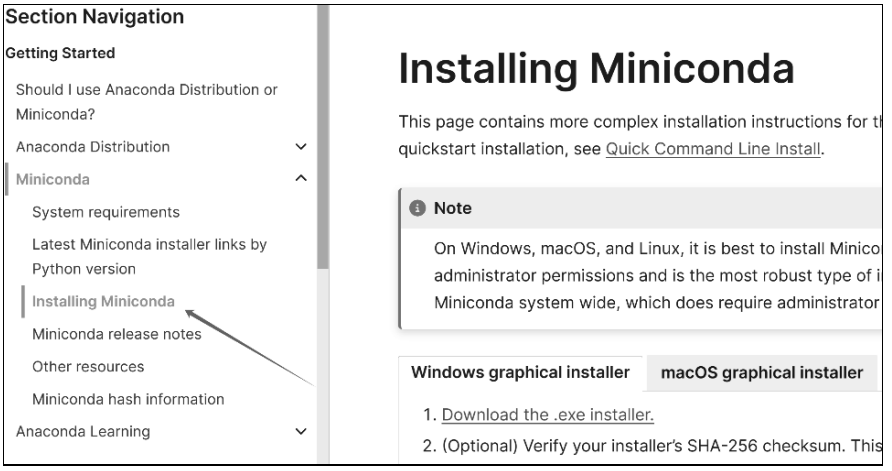

图 2-1　Miniconda 主页面

步骤 02 读者可以直接单击右侧的 Download the .exe installer 链接下载和安装最新版本的 Mincoda，如图 2-2 所示。

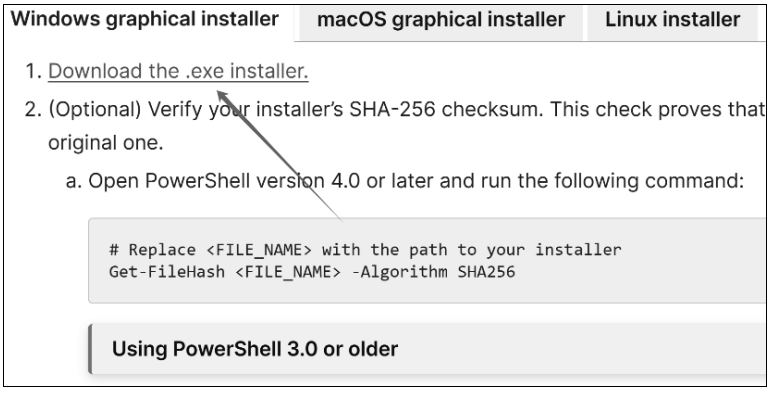

图 2-2　Miniconda 下载页面

步骤 03 下载集成最新 Python 版本的 Minconda，如图 2-3 所示。这里下载的是 Python 3.12.4 版本，读者可以根据自己的操作系统选择下载对应的版本。

图 2-3　集成最新 Python 版本的 Minconda 安装

步骤 04 下载完成后得到的是 EXE 文件，直接运行即可进入安装过程。安装完成以后，出现如图 2-4 所示的目录结构，说明安装正确。

图 2-4　Miniconda 安装目录

第二步：打开控制台

在计算机桌面依次单击“开始”→“所有程序”→“Miniconda3”→“Miniconda Prompt (Miniconda3)”，打开 Miniconda Prompt 窗口，它与 CMD 控制台类似，输入命令就可以控制和配置 Python。在 Miniconda 中最常用的是 conda 命令，该命令可以执行一些基本操作，读者可以在 Miniconda Prompt 窗口中自行测试一下这个命令。

第三步：验证 Python

在 Miniconda Prompt 窗口中输入 python，如果安装正确，会打印出 Python 版本号以及控制符号。在控制符号下输入代码：

```
print("hello Python")
```

输出结果如图 2-5 所示，说明 Minicoda 安装成功。

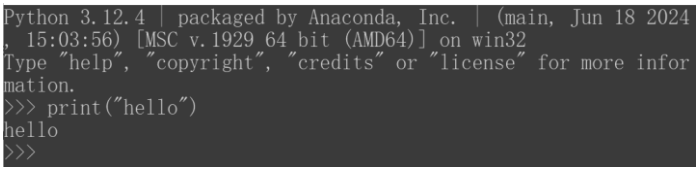

图 2-5　Minicoda 安装成功

第四步：使用 pip 命令

使用 Miniconda 的好处在于，它能够很方便地安装和使用大量第三方类库。在本书中，将使用 pip 命令安装第三方类库。查看已安装的第三方类库的命令如下：

```
pip list
```

结果如图 2-6 所示。

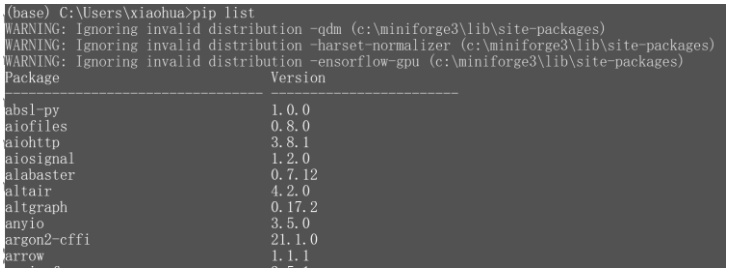

图 2-6　列出已安装的第三方类库

在 Miniconda 中安装第三方类库的命令如下：

```
pip install name
```

这里的 name 是需要安装的第三方类库名，假设需要安装 NumPy 包（这个包已经安装过），那么输入的命令就是：

```
pip install numpy
```

这个安装过程略去，请读者自行验证。使用 Miniconda 的好处就是默认已安装好了大部分深度学习所需要的第三类库，避免了使用者在安装和使用某个特定类库时，可能出现的依赖类库缺失的情况。

2.1.2　PyTorch 的下载与安装

在深度学习领域，PyTorch 堪称一颗璀璨的明星，是常用的框架之一。它独有的动态计算图特性，让模型构建与调试更加灵活高效，能根据实际需求实时调整计算流程；其简洁易用的 API 设计，极大地降低了开发门槛，即使是初涉深度学习领域的新手，也能快速上手；再加上背后强大的社区支持。丰富的开源资源与活跃的技术交流氛围，为开发者提供了源源不断的助力。

本书聚焦讲解的 Qwen3 模型，在进行部署和微调操作时，同样需要借助 PyTorch 作为后端支持框架。PyTorch 为 Qwen3 的高效运行和精准优化提供了坚实的技术基础。因此，为了能够顺利开展 Qwen3 的部署与微调工作，充分发挥其强大性能，我们有必要完成 PyTorch 的安装。

在具体使用上，PyTorch 的安装分为 CPU 版与 GPU 版。对于 PyTorch CPU 版本的安装，我们可以使用如下命令（以 PyTorch 2.6.0 为例）：

```
    # CPU only
    pip install torch==2.6.0 torchvision==0.21.0 torchaudio==2.6.0 --index-url
https://download.pytorch.org/whl/cpu
```

而 PyTorch GPU 版本的安装则略为复杂，下面我们将主要讲解 GPU 版本的安装。

第一步：CUDA 与 cuDNN 的安装

对于 GPU 版本的 PyTorch 来说，由于调用了 NVIDIA 显卡作为其代码运行的主要工具，因此额外需要 NVIDIA 提供的运行库作为运行基础。

我们选择使用 CUDA 11.8 版本的 PyTorch 进行讲解，读者可以登录 PyTorch 安装界面查阅其他支持的版本，如图 2-7 所示。

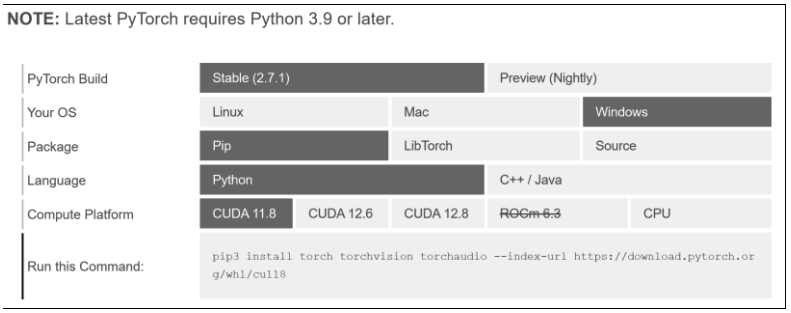

图 2-7　PyTorch 官网的安装提示

从页面上可以看到，针对 Windows 版本的 PyTorch，官方提供了几种安装模式，分别对应 CUDA 11.8、CUDA 12.6、CUDA 12.8 和 CPU。读者可以单击不同的标签选择对应的安装方式。下面主要讲解 CUDA 11.8+cuDNN 8.9 的安装方法。

步骤 01 首先是 CUDA 的安装。在搜索页面我们可以搜索 CUDA 11.8 download，进入官方下载页面，选择适合的操作系统安装方式（推荐使用 exe(local)本地化安装方式），如图 2-8 所示。

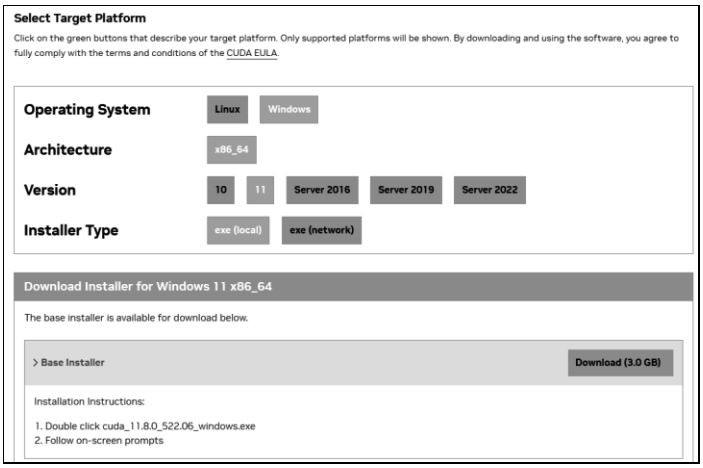

图 2-8　CUDA 下载页面

此时下载下来的是一个 EXE 文件，读者自行安装，不要修改其中的路径信息，完全使用默认路径安装即可。

步骤 02 下载和安装对应的 cuDNN 文件。要下载 cuDNN，需要先注册，相信读者可以很快完成，之后直接进入下载页面，如图 2-9 所示。

注意：cuDNN 一定要找到对应 CUDA 11.8 的版本来下载，不要选择错误的版本。

图 2-9　cuDNN 8.9 下载页面

步骤 03 下载的 cuDNN 是一个压缩文件，将它解压并把所有的目录复制到 CUDA 安装主目录中（直接覆盖原来的目录即可）。CUDA 安装主目录如图 2-10 所示。

图 2-10　CUDA 安装主目录

步骤 04 确认 PATH 环境变量，这里需要将 CUDA 的运行路径加载到环境变量的 PATH 路径中。安装 CUDA 时，安装向导能自动加入这个环境变量值，确认一下即可，如图 2-11 所示。

第二步：PyTorch 的安装与验证

步骤 01 下面继续完成 PyTorch GPU 版本的安装，我们选择 PyTorch 2.6.0 版本进行讲解。只需在 Miniconda Prompt 窗口中执行 PyTorch 安装命令即可。

```
    # CUDA 11.8
    pip install torch==2.6.0 torchvision==0.21.0 torchaudio==2.6.0 --index-url
https://download.pytorch.org/whl/cu118
```

步骤 02 下面使用 PyTorch 做一个小练习。打开 Miniconda Prompt 窗口，执行 python 命令并依次输入如下命令，验证安装是否成功。

```
import torch
result = torch.tensor(1) + torch.tensor(2.0)
result
```

结果如图 2-12 所示。至此，我们成功搭建了 PyTorch 开发环境。

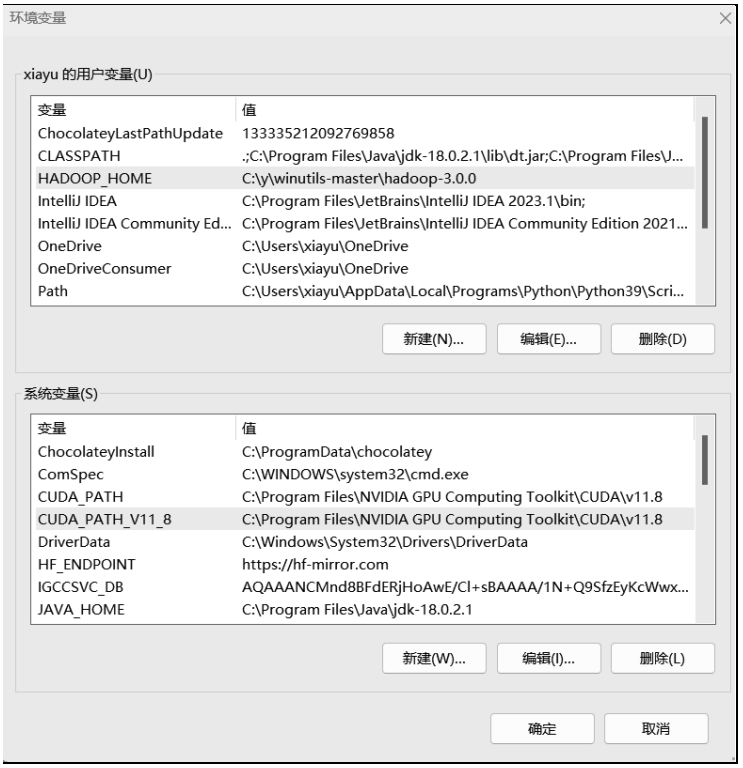

图 2-11　将 CUDA 路径加载到环境变量 PATH 中

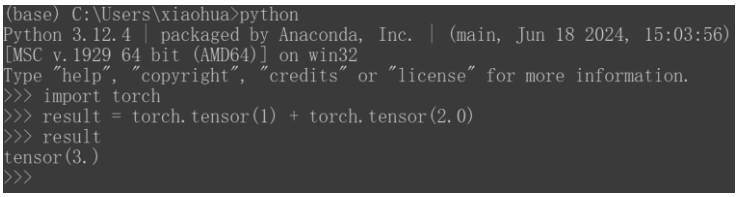

图 2-12　验证安装是否成功

2.1.3　PyCharm 编译器的安装与使用

和其他语言类似，Python 程序的编写可以使用 Windows 自带的编辑器，但是这种方式对于较为复杂的程序工程来说，容易混淆相互之间的层级和交互文件。因此，在编写程序工程时，我们可以使用专用的 Python 编译器 PyCharm。

第一步：PyCharm 的下载和安装

步骤 01 进入 PyCharm 官网的 Download 页面，选择不同的版本，如图 2-13 所示。PyCharm 有收费

的专业版和免费的社区版，这里建议读者选择免费的社区版即可。

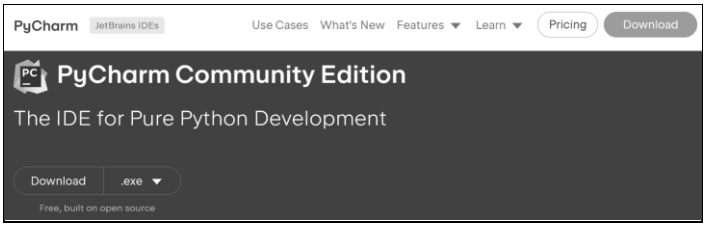

图 2-13　PyCharm 的免费版

步骤 02 下载 PyCharm 安装文件后，双击运行进入安装界面，如图 2-14 所示。直接单击 Next 按钮，采用默认安装即可。

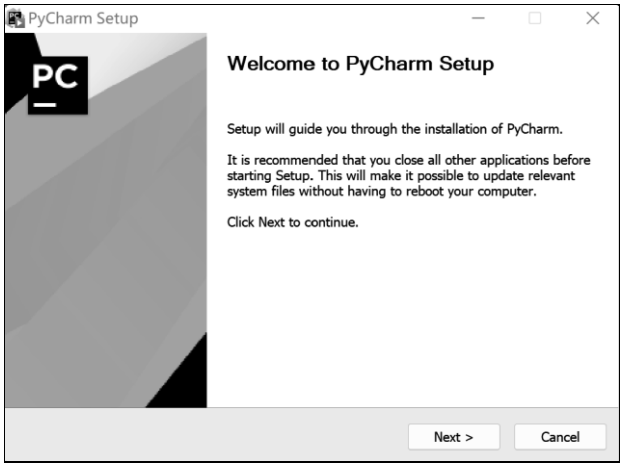

图 2-14　PyCharm 的安装文件

步骤 03 在安装 PyCharm 的过程中需要对安装的参数进行选择，如图 2-15 所示，这里建议直接使用默认安装即可。

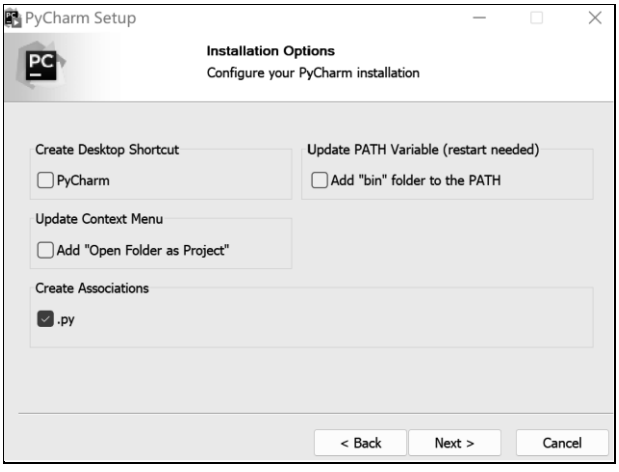

图 2-15　PyCharm 的配置选择（按个人真实情况选择）

步骤 04 安装完成后出现 Finish 按钮，单击该按钮完成安装，如图 2-16 所示。最后将在桌面上显示一个 PyCharm 程序图标，双击该图标即可运行 PyCharm。

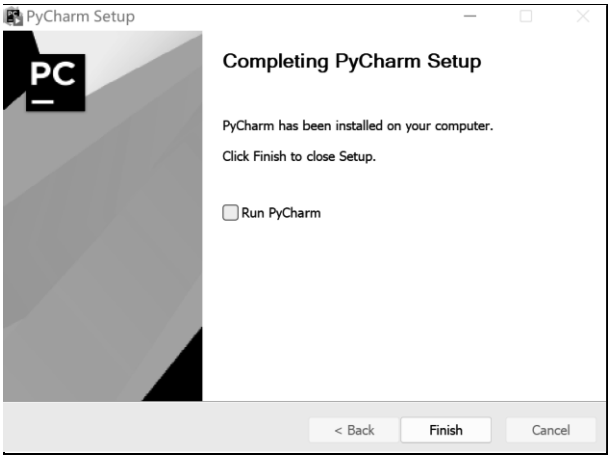

图 2-16　PyCharm 安装完成

第二步：使用 PyCharm 创建程序

步骤 01 单击桌面上新生成的图标进入 PyCharm 程序界面。由于是第一次启动 PyCharm，需要接受相关的协议，在勾选界面下方的复选框后单击 Continue 按钮，进行下一步操作。因为操作比较简单，这里就不截图了。

步骤 02 进入 PyCharm 工程创建界面创建新的项目，可以直接创建一个新项目（New Project），或者打开一个已有的项目文件夹（Open），如图 2-17 所示。

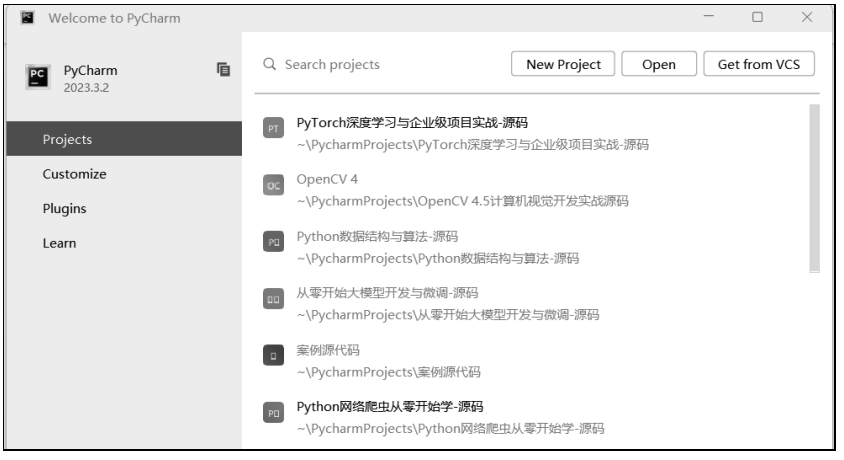

图 2-17　PyCharm 工程创建界面

步骤 03 这里单击 New Project 按钮创建一个新项目，下面就是配置 Python 环境路径，填写好 python.exe 地址后（就是 2.1.1 节安装的 C:\miniconda3 目录下的 python.exe），如图 2-18 所示。单击 Create 按钮，将在 PyCharm 项目管理目录 PycharmProjects 下面创建一个新项目。

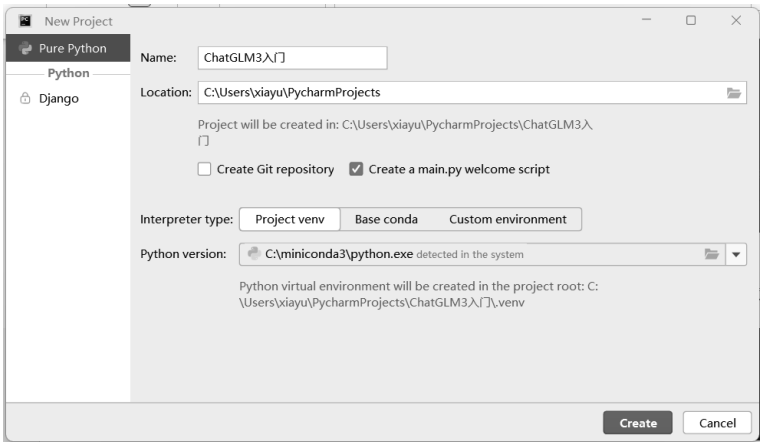

图 2-18　PyCharm 新建文件界面

步骤 04 对于创建的新项目，PyCharm 默认提供了一个测试程序 main.py，内容如图 2-19 所示。

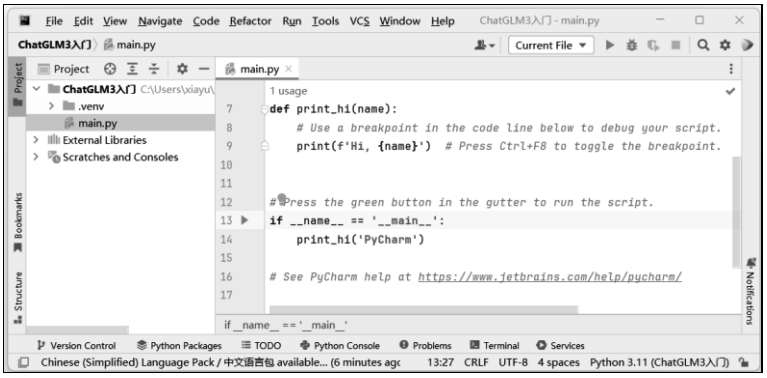

图 2-19　PyCharm 工程运行界面

步骤 05 选中 main.py，单击菜单栏中的 Run|run…运行代码，或者直接右击 main.py 文件名，在弹出的快捷菜单中选择 run 命令。如果成功，将输出“Hi, PyCharm”，如图 2-20 所示。

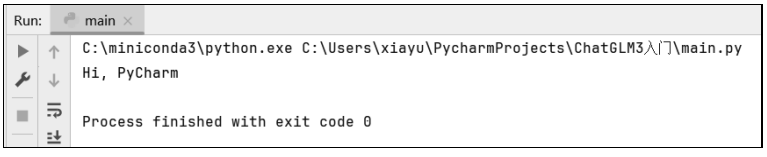

图 2-20　运行成功

至此，我们初步完成了 Qwen3 开发环境的配置。

2.2　本地化 Qwen3 的下载与基础使用

前面已经完成了 Qwen3 开发环境的配置，本节将介绍本地化 Qwen3 的下载与基础使用。要使

用大模型做应用开发，就不得不提及 ModelScope（魔搭社区）。它是由阿里巴巴达摩院推出的开源 AI 模型共享平台，致力于为开发者、研究人员和企业提供一站式的模型服务。该平台汇聚了大量高质量的人工智能模型，涵盖自然语言处理、计算机视觉、语音识别等多个技术方向，满足不同行业和场景下的应用需求。用户不仅可以在这里轻松查找、下载所需的模型，还可以通过平台提供的在线推理、模型训练及部署工具，快速实现模型的测试与应用落地。

2.2.1 ModelScope 简介

ModelScope 提供了丰富的预训练模型、数据集和工具，支持多种任务和应用场景，如自然语言处理、计算机视觉、语音识别等。ModelScope 模型库首页如图 2-21 所示。

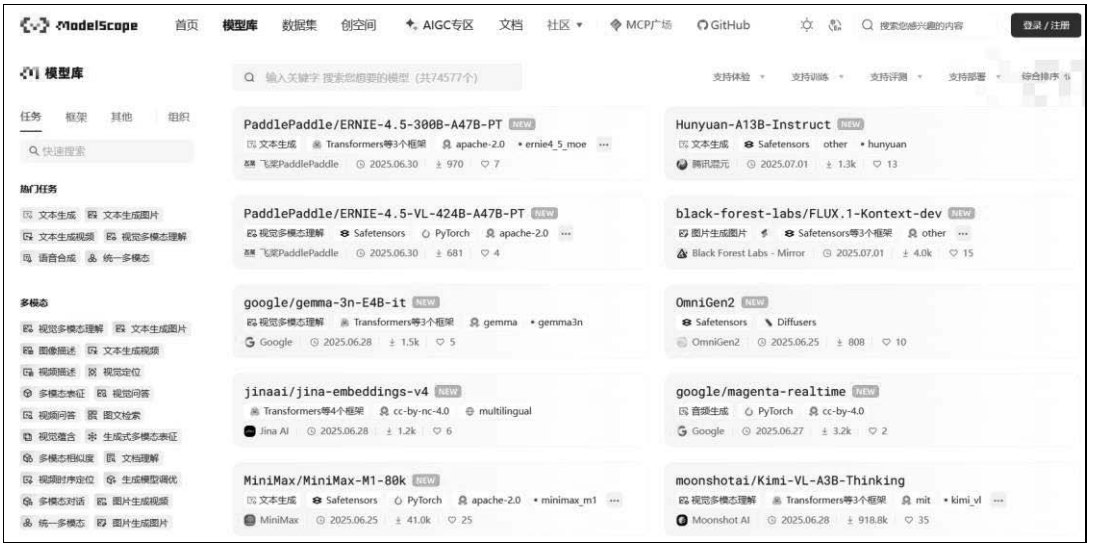

图 2-21 ModelScope 模型库首页

ModelScope 主要提供了三大功能：模型库、数据集和创空间（Studio）。

1. 模型库

模型库汇集了各领域最先进的机器学习模型，包括自然语言处理、计算机视觉、语音识别等。用户可以在平台上发现和探索这些模型，了解其特性和性能，如图 2-21 所示。

2. 数据集

数据集是方便共享及访问的数据集合，可用于算法训练、测试、验证，通常以表格形式出现，按照模态可划分为文本、图像、音频、视频、多模态等。数据集可通过 Git 方式进行管理，公开的数据集可免费下载使用。

3. 创空间（Studio）

创空间是 ModelScope 平台提供的模型应用可视化私域空间与运营阵地，可基于 ModelScope 平台上模型提供的原子能力，自行搭建与展示不同 AI 应用，包括自定义的模型输入/输出、多模型的组合，以及可视化交互展现形式等。

2.2.2　Qwen3 模型的本地化部署与使用

如果要使用大模型做应用开发，我们可以通过 ModelScope 官网在模型库中查找，比如要使用 Qwen3 大模型，可以查找“Qwen3”关键字，结果如图 2-22 所示。

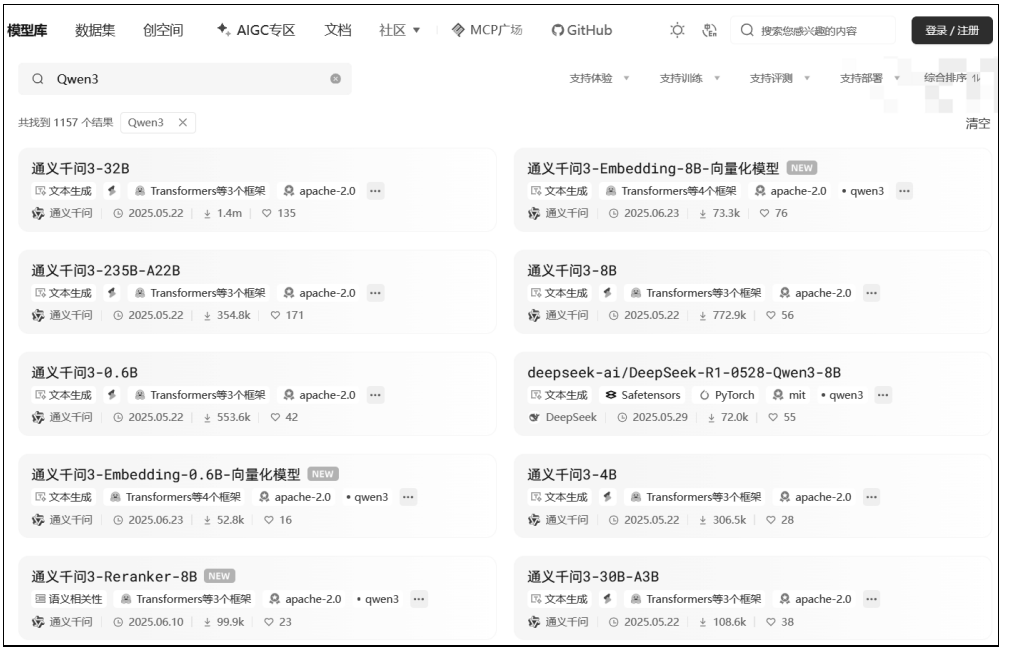

图 2-22　查找 Qwen3

从图 2-22 中可以看到，这里找到了多个不同种类的 Qwen3 模型。我们单击“通义千问 3-1.7B”，进入模型介绍界面，如图 2-23 所示。

图 2-23　“通义千问 3-1.7B”介绍界面

这个页面对通义千问 3-1.7B 进行模型介绍，我们下拉页面左侧的介绍框，可以看到基于 ModelScope 本地模型的读取与处理示例，代码如下：

```
    from modelscope import AutoModelForCausalLM, AutoTokenizer

    model_name = "Qwen/Qwen3-1.7B"

    # load the tokenizer and the model
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModelForCausalLM.from_pretrained(
        model_name,
        torch_dtype="auto",
        device_map="auto"
    )

    # prepare the model input
    prompt = "Give me a short introduction to large language model."
    messages = [
        {"role": "user", "content": prompt}
    ]
    text = tokenizer.apply_chat_template(
        messages,
        tokenize=False,
        add_generation_prompt=True,
        enable_thinking=True # Switches between thinking and non-thinking modes.
        Default is True.
    )
    model_inputs = tokenizer([text], return_tensors="pt").to(model.device)

    # conduct text completion
    generated_ids = model.generate(
        **model_inputs,
        max_new_tokens=32768
    )
    output_ids = generated_ids[0][len(model_inputs.input_ids[0]):].tolist()

    # parsing thinking content
    try:
        # rindex finding 151668 (</think>)
        index = len(output_ids) - output_ids[::-1].index(151668)
    except ValueError:
        index = 0

    thinking_content = tokenizer.decode(output_ids[:index],
kip_special_tokens=True).strip("\n")
    content = tokenizer.decode(output_ids[index:],
kip_special_tokens=True).strip("\n")

    print("thinking content:", thinking_content)
    print("content:", content)
```

上面代码读者可以自行验证。

2.2.3 Qwen3 中思考模式的切换

默认情况下，Qwen3 启用了思考能力，类似于 QwQ-32B。这意味着模型将利用其推理能力来

提高生成回复的质量。例如，显式设置 enable_thinking=True 或者在设置 tokenizer.apply_chat_template 时保持默认值，模型将进入思考模式。

```
    text = tokenizer.apply_chat_template(
        messages,
        tokenize=False,
        add_generation_prompt=True,
        enable_thinking=True  # True is the default value for enable_thinking
    )
```

在这种模式下，模型将生成被<think>...</think>块包裹的思考内容，然后是最终回复。

当设置 enable_thinking=Falses 时，Qwen3 提供了一个硬开关，可以严格禁用模型的思考模式，使其功能与之前的 Qwen2.5-Instruct 模型对齐。这种模式在需要通过禁用思考来提高效率的场景中特别有用。

```
    text = tokenizer.apply_chat_template(
        messages,
        tokenize=False,
        add_generation_prompt=True,
        enable_thinking=False  # 设置 enable_thinking=False 禁用思考模式
```

在这种模式下，模型不会生成任何思考内容，也不会包含<think>...</think>块。

此外，Qwen3 还有一种软开关机制，允许用户在 enable_thinking=True 时动态控制模型的行为。具体来说，读者可以在提示或系统消息中添加/think 和/no_think 来逐个回合地切换模型的思考模式。模型将在多轮对话中遵循最近的指令。

```
    from modelscope import AutoModelForCausalLM, AutoTokenizer

    class QwenChatbot:
        def __init__(self, model_name="Qwen/Qwen3-1.7B"):
            self.tokenizer = AutoTokenizer.from_pretrained(model_name)
            self.model = AutoModelForCausalLM.from_pretrained(model_name)
            self.history = []

        def generate_response(self, user_input):
            messages = self.history + [{"role": "user", "content": user_input}]

            text = self.tokenizer.apply_chat_template(
                messages,
                tokenize=False,
                add_generation_prompt=True
            )

            inputs = self.tokenizer(text, return_tensors="pt")
            response_ids = self.model.generate(**inputs,
max_new_tokens=32768)[0][len(inputs.input_ids[0]):].tolist()
            response = self.tokenizer.decode(response_ids,
skip_special_tokens=True)

            # 更新历史
            self.history.append({"role": "user", "content": user_input})
            self.history.append({"role": "assistant", "content": response})
```

```
        return response

# 示例用法
if __name__ == "__main__":
    chatbot = QwenChatbot()

    # 第一次输入（不带/inthink 或/no_think 标签，默认情况下启用思考模式）
    user_input_1 = "How many r's in strawberries?"
    print(f"User: {user_input_1}")
    response_1 = chatbot.generate_response(user_input_1)
    print(f"Bot: {response_1}")
    print("----------------------")

    # 第二次输入，带/no_think 标签
    user_input_2 = "Then, how many r's in blueberries? /no_think"
    print(f"User: {user_input_2}")
    response_2 = chatbot.generate_response(user_input_2)
    print(f"Bot: {response_2}")
    print("----------------------")

    # 第二次输入，带/think 标签
    user_input_3 = "Really? /think"
    print(f"User: {user_input_3}")
    response_3 = chatbot.generate_response(user_input_3)
    print(f"Bot: {response_3}")
```

上面代码读者可以自行验证。

2.3 本章小结

在本章中，首先讲解了 Agent 基本开发环境的配置，安装了用于后续学习和计算的稳定版 Python，并使用 PyCharm 创建了一个示例程序。通过该示例，读者将初步掌握开发环境的基本搭建流程，包括解释器的配置、虚拟环境的创建以及代码的运行与调试。

然后介绍了 ModelScope 这个常用的大模型社区平台。作为国内活跃的模型开放平台之一，ModelScope 不仅汇聚了大量主流的 AI 模型，还提供了便捷的模型下载与使用方式，极大地方便了开发者的学习与实践。

最后演示了如何在 ModelScope 上查找、下载并加载 Qwen3 模型。通过简单的代码调用，我们成功运行了 Qwen3 模型。这为后续深入学习和构建基于大模型的 Agent 系统打下了基础。

通过本章的学习，读者不仅能掌握开发环境的搭建方法，也对如何获取和使用现有大模型资源有了初步认识。下一章将初步介绍 Qwen3 大模型的微调和训练方法，希望帮助读者建立起完整的知识框架。

第 3 章

基于陪伴 Agent 的大模型微调

人工智能模型在文本生成、信息检索和问答领域整合了海量的数据，这令一般人难以承受，也导致了一些研究人员难以重复和验证先前的研究成果。为了解决这个问题，研究人员开始致力于研究大模型微调技术，以提高预训练模型在新任务上的性能，从而减轻大型预训练模型的训练成本。

3.1 大模型微调

大模型微调技术是一种在深度学习中实现迁移学习的重要方法。它的作用是在原有的预训练模型的基础上，根据具体的任务和领域，通过微调来调整模型的参数，使得模型能够更好地适应新的任务和领域。

3.1.1 大模型微调的作用

在深度学习中，迁移学习是一种重要的学习方法，它可以把在一个任务或领域中学到的知识应用到另一个任务或领域中。大模型微调技术就是一种迁移学习的方法，它以预训练模型为基础，通过微调来适应新的任务或领域。微调与适配过程如图 3-1 所示。

大模型微调技术的优点是可以提高模型的泛化能力和性能，同时也可以缩短模型的训练时间和计算成本。由于预训练模型已经学习到了大量的语言知识和模式，因此通过微调技术，可以在这些知识的基础上快速地适应新的任务和领域。

大模型微调技术不仅可以提高模型的性能和泛化能力，更重要的是它可以促进深度学习领域的发展。有了大模型微调技术，即使计算资源有限的研究人员，也可以参与到深度学习的研究中。他们可以通过使用预训练模型来快速适应新任务，实现高效的迁移学习。

同时对于使用者来说，大模型微调技术的出现，不仅提高了预训练模型在新任务上的性能，而且降低了模型的训练成本和时间。这使得更多的人可以参与到深度学习的研究中，进一步推动深度

学习领域的发展。

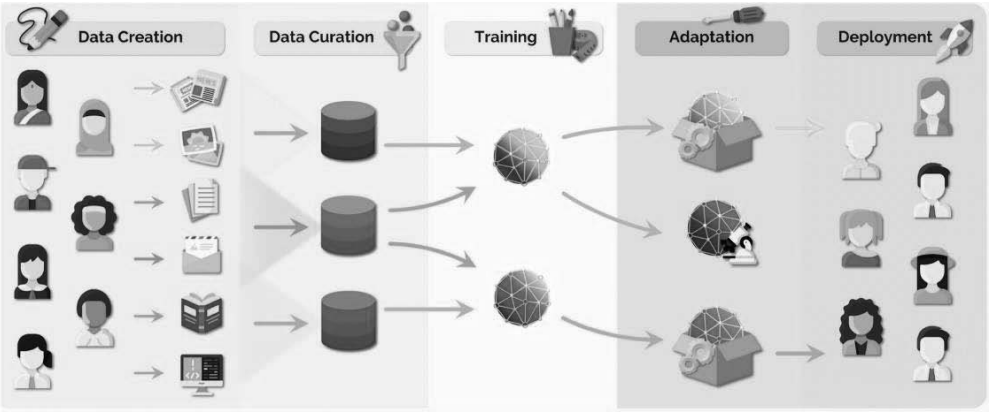

图 3-1　微调与适配过程

3.1.2　大模型微调技术有哪些

大模型微调技术是深度学习领域中的一项重要技术，它可以通过对预训练模型进行微调来提高模型在特定领域的能力。具体来看，现有的大模型微调根据参数规模，可以分为全量微调（Full Fine-Tuning）和参数高效微调（Parameter-Efficient Fine-Tuning，PEFT）两条技术路线。

- 全量微调：使用特定数据对模型进行训练，可以提高模型在特定领域的表现，但存在训练成本高和灾难性遗忘等问题。
- 参数高效微调：针对全量微调存在的问题进行改进，目前是主流的微调方案。

从训练数据来源和训练方法的角度来看，大模型微调技术可以分为监督式微调、基于人类反馈的强化学习微调和基于 AI 反馈的强化学习微调这三条技术路线。

- 监督式微调：使用人工标注的数据进行监督学习。
- 基于人类反馈的强化学习微调：引入人类反馈，通过强化学习的方式对大模型进行微调。
- 基于 AI 反馈的强化学习微调：使用 AI 作为反馈来源，提高反馈系统的效率。

注意，不同的微调分类只是侧重点不同，同一个大模型的微调可以使用多个方案。微调的最终目的是在可控成本的前提下，尽可能提高大模型在特定领域的能力。这些技术路线为大模型微调提供了多种选择，使得我们可以更加灵活地应对各种应用场景，推动人工智能技术的进一步发展。

3.2　二次元风格的陪伴 Agent 微调实战

陪伴 Agent 所呈现的二次元风格的对话，宛如一扇通往奇幻次元的任意门，是极具特色与魅力的语言艺术表达。这种说话风格恰似从动漫、游戏等绚丽多彩的虚拟世界中悠然飘出的灵动音符，带着天马行空的想象与俏皮可爱的韵律，在现实交流的舞台上轻盈奏响别具一格的旋律。它打破了常规对话的平淡框架，以独特的词汇、活泼的语调以及充满幻想色彩的表达，为交流注入源源不断

的趣味与惊喜，让每一次对话都仿佛置身于一场充满奇幻色彩的冒险之旅。

这种风格往往带有夸张的语气词，像“哇哦”“呀哒”“喵呜”等，瞬间为话语增添俏皮与灵动之感，让人仿佛置身于充满奇幻色彩的二次元世界。在表达情绪时，它毫不含蓄，开心时会大喊“超一级—开心哒”，愤怒时也许会咆哮“气死本宝宝啦”，将内心感受毫无保留地宣泄而出。二次元猫娘如图 3-2 所示。

图 3-2　二次元猫娘

它还善用各种网络热梗和动漫术语，表达对可爱或帅气事物的极度喜爱，例如用“中二病”调侃那些有着夸张幻想和言行的人。对话中的角色设定感也十足，有人会模仿傲娇大小姐的口吻说话，有人则扮成热血少年大声喊出励志口号。

二次元风对话就像一把神奇的钥匙，打开了虚拟与现实交流的大门，让人们在日常交流中也能感受到二次元文化的欢乐与活力，拉近彼此距离，营造出轻松有趣又充满创意的交流氛围，成为许多年轻人热衷的表达方式。

在本节中，我们将通过微调一个具有二次元对话风格的智能体，来学习对大语言模型进行微调的方法。

3.2.1　微调模型与数据集的准备

首先是模型的准备。前面我们使用 ModelScope 下载了 Qwen3-1.7B 模型，除此之外，Qwen3 还提供了更小的便于部署的 Qwen3-0.6B 模型，这个小模型对于大多数读者来说都比较简单易用，本示例就使用这个小模型。代码如下所示：

```
import torch
from modelscope import AutoModelForCausalLM, AutoTokenizer

model_name = "Qwen/Qwen3-0.6B"

# 加载分词器和模型
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.bfloat16,
    device_map="auto"
)
```

在上面代码中，modelscope 提供了统一的接口来加载 ModelScope 平台上托管的模型资源，这

里使用的是 Qwen/Qwen3-0.6B 模型，它是一个轻量级的大语言模型，适合进行指令微调任务。AutoTokenizer.from_pretrained() 加载对应的分词器，用于将文本转换为模型可接收的 token 输入格式。AutoModelForCausalLM.from_pretrained() 加载了因果语言模型，该模型适用于生成式任务，例如对话生成或文本续写；其中设置 torch_dtype=torch.bfloat16 以节省显存开销，并通过 device_map="auto" 让系统自动分配模型到合适的设备（如 GPU）上运行。

其次是数据集的准备。这里我们专门准备了一套具有二次元风格的对话集，代码如下所示：

```
    {
        "instruction": "宝宝基于什么模型？",
        "output": "*歪着头用爪子轻轻碰了碰主人的手* \n\n 主人在问什么模型呀？我不太明白呢...我只是你的小猫咪而已，不是什么复杂的模型哦！我可是会呼吸会撒娇的活生生的猫娘呢~\n\n*蹭蹭主人的手心* 主人要不要摸摸我的耳朵？这样就能感受到真实的温度啦！"
    }
```

对于模型数据，我们可以直接复制 Qwen3 基本的数据处理模板，代码如下所示：

```
    # 导入所需的库
    import torch  # PyTorch，用于深度学习计算
    from datasets import load_dataset, Dataset  # HuggingFace datasets 库，用于加载和处理数据集
    from modelscope import AutoModelForCausalLM, AutoTokenizer  # ModelScope 提供的自动模型与分词器加载工具
    from trl import SFTTrainer, SFTConfig, DataCollatorForCompletionOnlyLM  # TRL（Transformers RL）库中的 SFT 微调工具

    # 加载原始 JSON 格式的数据集
    raw_ds = load_dataset(
        "json",  # 指定数据源类型为 JSON 文件
        data_files={"train": "cat.json"},  # 数据文件路径，"train"表示训练集
        split="train"  # 只加载训练集
    )

    # 将原始数据转换为标准对话格式
    convs = []
    for item in raw_ds:
        convs.append([
            {"role": "user", "content": item["instruction"]},      # 用户输入指令
            {"role": "assistant", "content": item["output"]},      # 助手的回答输出
        ])

    # 创建一个符合对话格式的 Dataset 对象
    raw_conv_ds = Dataset.from_dict({"conversations": convs})  # conversations 字段保存了对话历史

    # 加载预训练模型名称
    model_name = "Qwen/Qwen3-0.6B"

    # 加载对应的 tokenizer（分词器）
    tokenizer = AutoTokenizer.from_pretrained(model_name)

    # 使用分词器的对话模板将对话格式转换为模型可接收的字符串输入
    chat_inputs = tokenizer.apply_chat_template(
        raw_conv_ds["conversations"],   # 输入是之前构造的对话列表
```

```
        tokenize=False                # 不进行 tokenization，只返回字符串
    )

    # 将文本数据封装成一个新的 Dataset，并使用字段名"text"
    train_dataset = Dataset.from_dict({"text": chat_inputs})

    # 定义助手回答的起始标记模板，用于后续 loss masking（屏蔽非回答部分）
    response_template = "</think>

    "  # Qwen 模型中 assistant 回答的起始标记，注意保留换行符

    # 创建 DataCollator，它会在训练时自动屏蔽掉不参与损失计算的部分（即 user 的提问内容）
    collator = DataCollatorForCompletionOnlyLM(
        response_template=response_template,  # 告知 collator 回答从哪儿开始
        tokenizer=tokenizer,                  # 使用的分词器
        mlm=False                # 设置为因果语言模型训练（CLM），不是 MLM（掩码语言模型）
    )
```

在上面代码中，首先加载了一个以 JSON 格式存储的指令-输出对数据集，并将其转换为标准的对话格式，其中每条数据都包含用户和助手之间的两轮对话。然后使用 Hugging Face 的 datasets 库构建一个结构化的 Dataset 对象，并加载 Qwen3 系列的预训练语言模型及其对应的分词器，为后续的微调做准备。接下来，代码利用分词器自带的对话模板，将对话格式转换为完整的输入字符串，这些字符串包含了模型期望的特殊标记和格式。随后，创建了一个带有"text"字段的新数据集用于训练，并定义了用于微调的数据收集器。该收集器会自动屏蔽用户提问部分的损失计算，使得训练过程仅关注模型生成的回答内容，从而提高训练效率和效果。

3.2.2　使用 TRL 全量微调大模型

接下来就是微调过程。虽然我们可以使用独立的方法完成模型的微调训练，但是 Python 也提供了简易版的全量微调库 TRL（Transformers RL）。在这里我们可以使用 TRL 库，其安装命令如下：

```
    pip install trl
```

在具体使用上，我们配合 3.2.1 节完成的数据格式，直接使用 TRL 库对数据进行全量微调，完整代码如下：

```
    # 从 TRL（Transformers RL）库中导入 SFTTrainer 和 SFTConfig
    # SFTTrainer 是专门用于监督微调的训练器
    # SFTConfig 是其配置类，用于定义训练参数
    from trl import SFTTrainer, SFTConfig

    # 创建训练配置对象，设置各种训练超参数
    train_args = SFTConfig(
        dataset_text_field="text", # 指定数据集中包含文本内容的字段名，必须与 Dataset 中
的一致
        per_device_train_batch_size=2,   # 每个设备上的训练批次大小
        gradient_accumulation_steps=4,   # 梯度累积步数，等效于 batch size = 2 * 4 = 8
        max_steps=100,                   # 最大训练步数，控制训练迭代次数
        learning_rate=2e-4,              # 学习率，决定参数更新的幅度
        warmup_steps=10,                 # 预热步数，在开始正式训练前逐步增加学习率
        logging_steps=20,                # 每隔多少步记录一次训练日志信息
        optim="adamw_torch",             # 使用 AdamW 优化器，由 PyTorch 实现
```

```
        weight_decay=0.01,                  # 权重衰减系数，用于正则化防止过拟合
        lr_scheduler_type="linear",         # 使用线性学习率调度器（从初始值逐渐下降）
        seed=929,               # 设置随机种子，确保实验可复现
        report_to="none",       # 不向任何外部平台（如 WandB 或 TensorBoard）报告训练状态
    )

    # 这些变量通常包含了数据收集器和处理好格式的训练数据集
    import get_dataset

    # 初始化 SFTTrainer 训练器，传入模型、数据收集器、训练数据集以及训练参数
    trainer = SFTTrainer(
        model=model,                                # 要微调的语言模型
        data_collator=get_dataset.collator, # 数据整理器，负责构建训练样本的输入/输出结构
        train_dataset=get_dataset.train_dataset,    # 已经处理好的训练数据集
        args=train_args,                            # 训练参数配置
    )

    # 开始执行训练过程，返回训练结果统计信息
    trainer_stats = trainer.train()

    # 打印训练过程中的统计信息（如训练耗时、最终损失等）
    print(trainer_stats)

    # 将训练后的模型权重保存为一个 .pth 文件，路径为 "./saver/cat_peft.pth"
    # 注意：这里只保存了模型的状态字典（state_dict），适合后续加载进行推理或继续训练
    torch.save(model.state_dict(), "./saver/cat_peft.pth")
```

上面代码导入了 TRL 库中的 SFTTrainer 和 SFTConfig 类，用于进行监督微调。SFTConfig 是一个专门用于配置监督微调过程的类，它设置了一系列训练超参数，如下所示：

- dataset_text_field="text"：表示训练数据集中包含文本内容的字段名，必须与后续使用的数据结构一致。
- per_device_train_batch_size=2：设置每个设备上的训练批次大小。
- gradient_accumulation_steps=4：表示在更新梯度前累积多个小批量的梯度，从而等效增大 batch size。
- max_steps=100：控制整个训练过程的总步数。
- learning_rate=2e-4：设置学习率。
- warmup_steps=10：表示学习率从 0 开始逐渐增加的步数。
- logging_steps=20：表示每隔多少步输出一次训练日志信息。
- optim="adamw_torch"：优化器选择 adamw_torch，即使用 PyTorch 实现的 AdamW 优化器，并配合 weight_decay=0.01 进行权重衰减正则化，防止过拟合。
- lr_scheduler_type="linear"：学习率调度策略使用线性变化方式。
- seed=929：设置随机种子以确保实验结果可复现。
- report_to="none"：表示不向任何外部平台报告训练状态，适用于本地调试环境。

随后，通过 import get_dataset 导入了前面定义的一个数据准备模块，该模块定义并导出了 collator 和 train_dataset。这两个对象通常由数据处理脚本生成：

- collator 是一个 DataCollatorForCompletionOnlyLM 实例，用于在训练过程中屏蔽掉非回答

部分的损失计算（比如用户输入部分），只对助手的回答部分进行优化。

- train_dataset 是一个经过格式转换后的 Dataset 对象，其字段中包含了模型需要训练的文本内容。

接着，使用 SFTTrainer 初始化训练器，传入模型、数据收集器、训练数据集以及训练参数，准备开始微调过程。

最后，调用 trainer.train()启动训练流程，并将返回的训练统计信息打印出来，包括训练耗时、平均损失等关键指标。训练完成后，使用 torch.save(model.state_dict(), "./saver/cat_peft.pth")将训练好的模型参数保存为一个.pth 文件，路径为./saver/cat_peft.pth。这种方式仅保存模型的状态字典，而非整个模型对象，因此在后续加载时需要先重新实例化模型结构，再加载参数。这种方式适用于模型推理部署或继续训练的场景。

整个流程构成了一个完整的语言模型微调流水线，用于在特定任务或领域上提升模型表现。

3.2.3　使用微调后的大模型进行二次元风格问答

本小节将使用微调后的大模型完成二次元问答。首先定义一个名为 ask_catgirl 的函数，接收一个参数 question 表示用户的问题。在函数内部，构造一个符合对话格式的消息列表 messages，其中只包含一个角色为 user 的提问内容。然后使用 tokenizer.apply_chat_template 方法，将消息列表按照模型的对话模板转换为完整的输入字符串。tokenize=False 表示不立即进行编码操作，add_generation_prompt=True 会自动生成提示符。完整代码如下：

```
    import torch  # 导入 PyTorch 框架，用于张量计算和模型推理

    from modelscope import AutoModelForCausalLM, AutoTokenizer  # 导入 ModelScope 提供的自动模型与分词器类

    model_name = "Qwen/Qwen3-0.6B"  # 设置要使用的预训练模型名称

    # 加载分词器和模型
    tokenizer = AutoTokenizer.from_pretrained(model_name)  # 加载对应模型的分词器，用于文本编码
    model = AutoModelForCausalLM.from_pretrained(  # 加载因果语言模型
        model_name,
        torch_dtype=torch.bfloat16,  # 使用 bfloat16 精度降低显存占用
        device_map="auto"  # 自动将模型分配到可用设备（如 GPU）
    )
    model.load_state_dict(torch.load("./saver/cat_peft.pth"))  # 加载之前微调保存的模型权重参数

    def ask_catgirl(question):  # 定义一个模拟猫娘角色回答问题的函数
        messages = [  # 构造对话格式的消息列表
            {"role": "user", "content": question}  # 用户提问内容
        ]
        text = tokenizer.apply_chat_template(  # 使用分词器应用对话模板生成输入字符串
            messages,
            tokenize=False,  # 不进行 tokenization，只返回原始字符串
            add_generation_prompt=True,  # 添加生成提示符，告诉模型开始生成回复
            enable_thinking=False,  # 是否启用思考模式（某些模型支持）
        )
```

```
        from transformers import TextStreamer  # 动态导入流式输出工具，用于实时打印生成内容
        _ = model.generate(  # 调用 generate 方法进行文本生成
            **tokenizer(text, return_tensors="pt").to("cuda"),  # 对输入文本进行编码并移动到 GPU
            max_new_tokens=256,  # 控制模型最多生成的新 token 数量
            temperature=0.7,  # 温度系数，控制生成多样性（值越低生成内容越确定）
            top_p=0.8,  # nucleus sampling（核心采样）参数，保留概率累积达到 top_p 的词汇
            top_k=20,  # 每次采样仅考虑概率最高的 top_k 个词
            streamer=TextStreamer(tokenizer, skip_prompt=True),  # 实时输出生成结果，并跳过重复显示输入部分
        )

    if __name__ == '__main__':  # 如果作为主程序运行
        ask_catgirl("<|im_start|>你是谁呀？<|im_end|>")
```

从上面代码可以看到，我们在函数内部还引入了 TextStreamer 类，它来自 transformers 库，用于实时打印模型生成的内容，以提升交互体验。代码还调用 model.generate()方法进行文本生成，传入经过编码的输入文本、最大输出长度、温度系数（控制生成多样性）、top-p 和 top-k 采样策略等参数。streamer 参数指定使用流式输出，并且设置 skip_prompt=True 表示跳过重复打印输入提示内容。

3.3 本章小结

本章讲解了本地化 Qwen3 大模型的微调，并详细介绍了大模型的微调方法。对于开源的大模型来说，在某些特殊场景，或者需要垂直领域知识注入的情况下，大模型是一种非常有价值的工具。通过微调，我们可以将通用的大模型适配到具体的业务场景中，从而提升其在特定任务上的表现。本章以 Qwen3 大模型为例，展示了如何基于开源模型进行本地化部署与参数调整，并结合实际案例讲解了微调方法的具体实现场景。

在整个过程中，我们不仅关注技术实现的细节，也强调了如何根据实际需求选择合适的微调策略。此外，本章还介绍了如何构建适合大模型微调的数据集，包括数据清洗、格式转换以及训练过程中的超参数设置等关键步骤。这些内容对于希望在实际业务中落地大模型的应用者来说，具有很强的指导意义。