第1篇　基础篇

A2A协议（Agent-to-Agent Protocol，智能体间通信协议）是由Google主导开发，并已移交至Linux基金会管理的技术标准及多语言实现。随着大模型在生成与推理能力上的持续突破，结合CoT（Chain-of-Thought，思维链）、ReAct（反应-行动框架）、ToT（Tree of Thought，思维树）等思考框架的理论与实践演进，Agent（智能体）逐步成为大模型能力落地的最佳方案。

在这一过程中，主动思考、自主决策与自主执行已成为行业共识，但跨平台的互操作性依然存在短板——单纯依赖API调用或OpenAI的Chat接口兼容层，无法有效解决Agent间的能力发现、流式交互、安全认证与长会话保持等核心问题。

A2A的出现填补了这一空白。它通过定义标准化的能力描述机制与多模态交互协议，构建了一套完整的跨平台协作框架，使来自不同厂商、平台与语言的Agent能够如同人类一样协作——自主发现彼此的技能、按需调用接口，并在复杂任务中协同执行。这种协议层面的统一，为Agent从单一应用走向大规模协同生态奠定了技术基础，也使大语言模型的应用场景从简单的内容生成扩展到更具实用价值的自动化问题解决领域。

A2A协议本身并不复杂，主要基于现有的HTTP(S)、JSON-RPC、OAuth、SSE（Server-Sent Events，服务器发送事件）等技术实现。然而，“不复杂”并不意味着内容单薄。本书的写作思路是：先讲解最小A2A应用所需的基础理论，然后借助一个简单易懂的入门级的A2A与Agent协作案例，为后续的具体开发与案例应用打下基础。因此，基础篇仅包含以下两章内容。只要理解并实践这部分内容，读者便可具备A2A的基本应用能力。

第1章　理解A2A：概念、架构与演进。本章首先介绍A2A的基本概念、主要特性和核心价值；然后简单回顾A2A的发展历程；接下来，以协议概览作为本章的技术重点，将系统阐述A2A的核心概念、基本通信元素、传输方式以及安全框架等内容，这些知识构成了A2A应用搭建的技术基础；最后，将A2A与最近流行的、与Agent紧密关联的MCP（Model Context Protocol，模型上下文协议）和Function-Calling（函数调用）技术进行对比分析，帮助读者理解它们各自的特点和适用场景。

第2章　极简入门：快速构建第一个A2A应用。对于开发者而言，通过代码实践掌握A2A是最快的学习路径。与A2A官网Quickstart文档逐步讲解的思路不同，本章的目标是用最简洁的代码示例涵盖第1章所介绍的A2A核心知识。一方面，示例需保持简短但具备完整功能，而非从网络中复制一段代码；另一方面，本章还将详细说明示例的运行环境的搭建过程，展示代码的运行、数据的流动及整体的架构。这样的示例将成为A2A初学者最直观、最具参考价值的入门指南。

第 1 章

理解A2A：概念、架构与演进

A2A是一个用于智能体交互的开放协议，由Google主导制定和开发。其目标是实现基于不同框架构建的异构智能体之间的任务协作与安全通信。本章首先讲解A2A的基本概念、主要特性及核心价值，然后简要介绍A2A的发展历程，以及社区生态的形成与演进，接着概述A2A协议，为后续章节深入探讨各类集成场景奠定理论基础，最后将A2A与最近流行的MCP协议，以及与大模型密切相关的Function-Calling特性进行横向对比。

1.1 A2A 核心概念解析

A2A 的设计初衷是为了解决 Agent 与 Agent 之间的协同问题，因此本节将从 Agent 和 A2A 的基本概念讲起，并详解 A2A 的核心价值。

1.1.1 A2A 的基本概念

Agent是一种智能实体应用程序，具备自主规划、自行决策并自动执行任务的能力。与传统人工智能应用不同，Agent能够通过主动思考和调用工具，有计划地逐步实现用户设定的目标。这一概念并非近年才提出，在大语言模型（Large Language Model，LLM）兴起之前，Agent通常以“代理”或“助理”等较为字面化的形式存在，作为工具被动地协助人类完成某些任务。注意，本书后续简称的大模型，如果不特别说明，就是指大语言模型。

随着大语言模型的快速发展，Agent技术持续演进，已经从最初通过多角色对话解决用户问题的形态，逐渐拓展为具备长时记忆、主动思考、任务分解与执行、工具调用等能力的复杂系统，正在朝着通用人工智能的方向迈进。

近年来，具备更高自主性的Agent迅速流行。借助大模型的支持，这些Agent实现了自主思考、

自主决策、自主执行等主动行动能力，突破了传统“一问一答”和固定工作流的局限。在AI领域，关于Agent的开发有一个“基于大模型、主动完成任务”的共识，但尚未形成统一的开发标准。不同厂商、框架和语言定制的大量服务之间无法方便地进行互通。针对这一问题，A2A提供了一套涵盖服务能力发现、消息、任务、工件、传输通道、安全等方面的通用语言，打破了各自为政的Agent孤岛，显著提升了互操作性，进而培育出一个更为互联、强大且富有创新力的人工智能生态系统。

1.1.2　A2A 的主要特性

Agent正在快速发展，已有很多实际应用案例。传统上，大量依赖WebUI与人类交互的软件正面临被“Agent+大模型”取代的趋势。当具备自主解决问题能力的人工智能产品成为主流时，如何实现它们之间的相互交流将成为急待解决的现实问题。A2A正是关注到这一点，其设计目标是打造一套标准化的通信与协作机制，确保不同技术架构、开发框架和编程语言开发的智能体能够高效、安全地交换信息，并协同完成复杂任务。具体来说，这一宏观目标可以细化为A2A的四大核心特性。

（1）A2A制定了统一的Agent发现机制。在A2A出现之前，开发人员通常将Agent包装成工具，以便与其他Agent或人工智能应用进行交互，常用的方式包括HTTP服务、OpenAI Chat兼容接口等。此时，服务能力的协商主要依赖人与人之间的沟通方式。虽然这种方式在小规模应用中没有问题，但当大量Agent进入流程，尤其是一个Agent调用另一个Agent，并逐步发展成像Python库之间错综复杂的依赖关系时，这种沟通方式就显得低效。A2A通过设计直接的Agent能力协商机制，显著提高了交互效率。

（2）A2A制定和实现了多种交互数据格式。A2A支持多种交互数据格式，包括最基础的纯文本格式、JSON结构化数据和文件类型的二进制数据。这些数据格式几乎涵盖了Agent之间或应用与Agent之间的交互需求，不仅支持文本数据，还为音频、视频等多媒体格式定义了标准，具有强大的适应性。在调用方式上，A2A提供了简单的JSON-RPC和更高效的gRPC框架，以满足不同场景下Agent通信的需求。

（3）为企业应用做好了基础准备。高效、安全的运营监控和清晰的文档等是企业应用的刚需。A2A的异步优先技术为交互双方进行长时间、多轮次的沟通提供了支持。配置TLS 1.2或更高版本的HTTPS通道是生产环境的必需项，且A2A从设计之初便引入了基于OAuth2的身份认证，解决了细粒度的访问授权问题。此外，分布式跟踪和全面的日志记录与A2A协议紧密关联，进一步加强了企业级应用的支持。

（4）多语言支持促进A2A协议的推广。A2A协议的官方SDK提供了Python、JavaScript、Java、C#/.NET、Go等主流编程语言的稳定实现，推动了协议的广泛应用。同时，Google ADK、LangGraph、BeeAI等第三方框架也陆续支持A2A的自定义适配，进一步为整个生态提供技术支撑。A2A协议本身具备语言无关性，使得基于不同语言开发的Agent能够在统一的协议规范下实现互操作与跨平台调用，为生态的开放与协同提供了坚实的技术基础。

1.1.3　A2A 的核心价值

A2A 的核心价值主要体现在两方面：其一是对跨平台、跨语言 Agent 之间协作的支持；其二是在业务流程自动化方面的促进作用。

1. 推进多Agent协作

自2023年4月12日AutoGPT发布以来，Agent技术已高速发展了两年多。然而，与大模型带来的显著变革相比，Agent的应用广度和影响深度仍显不足。在这一发展过程中，Agent原本的一些优势能力也呈现被大模型吸收的趋势。例如，思考框架CoT（思维链）被用于大模型训练，使得DeepSeek-R1的推理能力大幅提升；Agent所擅长的分步生成与任务规划方式，也正逐步被大模型集成，如近日发布的通义千问研究型智能体模型Qwen-Deep-Research所展示的能力。

Agent落地未达预期，除了模型、算力和工具等方面的局限外，与通用型Agent发展缓慢也有一定的关系。目前除Manus和少数几个Agent产品外，各厂家实现的Agent多聚焦于特定领域。要实现类似混合专家模型（Mixture of Experts，MoE）的强通用型能力，Agent还需要走很长的路。

在此背景下，通过A2A协议实现Agent之间的协作，能够在一定程度上缓解“服务孤岛”问题。由于Agent往往与特定模型、提示词框架、思考框架与决策流程深度绑定，不同Agent之间若能通过A2A协议实现跨框架、跨领域协同，就可发挥各自在特定任务上的优势，形成互补效应。这样一来，一些各自专注于具体领域的Agent，通过组合即可完成更复杂、更智能的任务。例如，通过A2A将基于LangGraph、CrewAI、Semantic Kernel等不同平台构建的Agent连接起来，便能够创建更为强大的复合型AI系统。

2. 驱动业务流程自动化

人工智能应用程序（亦可视为Agent的宿主程序）可通过调用Agent来实现功能，或其本身直接充当一个Agent。这类运行方式已有成熟的技术路径支持，例如传统的函数调用、HTTP请求、RPC（Remote Procedure Call，远程过程调用）以及API调用等，均可实现单个或少量Agent的集成与交互。

然而，当场景扩展至多个Agent协同工作时，复杂性也随之上升。若多个Agent均位于同一工具内部，通常可通过流程图（Flow）进行编排，配置上下游任务的衔接逻辑，如Langflow、Dify等AI应用所提供的流程图，使用起来相对简单。但随着Agent生态的快速发展，单一工具内建的Agent功能已难以覆盖全部需求，跨平台、跨系统乃至与外部Agent的交互需求日益频繁。

在这种情况下，要让异构环境中的Agent实现高效协作，就迫切需要统一的协议及标准化的共享机制。A2A通过使擅长不同专业的智能体协同工作，让它们相互发现、委派子任务、交换信息并协调行动，从而解决了单个Agent难以独立完成的复杂问题。

此类自动化流程涵盖跨系统的端到端业务处理。与传统依赖编程实现的流程控制相比，这一方式无须人工预设大量复杂的分支逻辑，而是在大模型驱动下，由Agent自主协商、动态判断流程

走向，并基于上下文不断调整执行策略，从而真正实现智能化、自适应的业务流程自动化。

1.2　A2A 生态演进之路

A2A 是一个很新的 Agent 交互技术标准体系，目前仍在快速演进中，其生态也在逐步形成。

1.2.1　A2A 产生背景

A2A是Agent发展到一定阶段的产物。早期的人工智能主要是“工具型”，比如文本翻译接口、图像识别API、语音转文本服务等，使用者需要主动调用这些工具来完成特定步骤。而现在，AI正在向“主动型”演进。Agent是能够感知环境、自主决策并执行动作以实现目标的智能体；单个Agent可以独立完成一类任务，当单个Agent的能力有限，或者任务过于庞大和复杂时，自然就会产生多个Agent之间进行沟通和协作的需求，这也是A2A产生的原始驱动力。

现实世界中的任务往往是多层次、多步骤的，一个任务可以分解为一系列子任务，由不同的人员分角色完成。人类通过语言、法律、市场机制等实现了大规模协作，而A2A生态的设想，是在由Agent组成的数字世界里复刻这种人类社会的协作模式。大语言模型的出现，为构建具有强大理解和生成能力的Agent提供了坚实基础：它们能够理解复杂指令、进行上下文推理并生成结构化响应，使Agent间的通信成为可能。

Agent和Agent之间的交互需要一套共同的“语言”和“工作流程”协议，以确保信息传递的准确性和协作的顺畅性。在A2A出现之前，不同框架开发的Agent因缺乏统一的交互标准，彼此之间如同存在“语言壁垒”，难以直接进行有效的任务协作和安全通信。比如基于LangChain构建的Agent与采用CrewAI框架开发的Agent，在数据格式、通信方式、能力描述等方面存在差异，导致它们在协同处理跨领域任务时需要开发大量适配代码。这不仅增加了开发成本，也降低了系统的稳定性和可扩展性。随着AI技术的飞速发展，Agent的数量和种类都在快速增加，异构Agent间的协作需求日益迫切，A2A正是在这样的背景下应运而生。

1.2.2　A2A 关键版本迭代

A2A从2025年6月10日发布第一个版本（v0.1.0）开始，到2025年7月30日发布最近一个版本（v0.3.0），历时很短，目前仍在稳步迭代中。

整体来看，A2A由v0.1.0起步，在v0.2.0进行了重要迭代，随后v0.2.2版本着重于规范化和完善，解决了底层RPC协议问题并引入了主流的通信架构定义；v0.2.5版本增强了Agent Card的定义和任务通知的灵活性；最新的v0.3.0版本则显著提升了协议的安全性和扩展能力。A2A版本发展历程如下。

1. 初始版本

v0.1.0是A2A的初始版本，于2025年6月10日发布。从其更新历史记录看，A2A的首个版本并非

一蹴而就，而是经过254次代码提交逐步完善而成。

2. 迭代版本

v0.2.x版本在v0.1.0基础上进行了重要变更，包括：解决与JSON-RPC 2.0规范的兼容性问题；首次正式将gRPC和REST定义加入A2A协议规范；为Agent Card添加了可选的iconUrl字段和必需的协议版本字段；支持为每个任务配置多个推送通知等。

3. 最新版本

2025年7月30日发布的v0.3.0版是笔者写作时A2A的最新版本。其重要修改包括：在安全方案中新增mTLS（双向TLS）支持；为OAuth2添加元数据URL字段，允许技能声明其安全要求；将协议规定的Agent Card托管URI的文件名由agent.json更改为agent-card.json（这一修改对应用开发的影响较大，因为第三方厂商的A2A实现中仍可能沿用agent.json，所以在编写程序时需要考虑兼容性问题）。

4. 开发版本

A2A仍在不断迭代中。从其GitHub官方代码库可见，A2A几乎保持每周2～3次的提交频率。

5. 其他说明

A2A由于处于规范层面，版本发布并不频繁，改动幅度也相对有限；与A2A同步发布的A2A-SDK则迭代更快一些。以A2A Python SDK为例，从2025年5月20日发布第一个正式版本v0.2.1起，到2025年10月21日发布v0.3.10，中间经历了28个版本。

综上所述，A2A是一个很新的协议，发布时间不长，与之配套的SDK仍在快速迭代，但总体处于比较稳定、逐步定型的阶段，小版本更新对开发者的影响相对不大。

1.2.3　A2A 社区生态

作为一个开放的技术协议，开源社区的支持对于A2A的发展至关重要。A2A从设计之初就秉持开源理念，其核心规范、参考实现代码以及配套的SDK等均托管在公开的代码仓库中，鼓励全球开发者参与协议的完善、拓展和应用实践。目前，A2A社区已吸引来自学术界、科技企业以及独立开发者等多方力量的关注和贡献。社区成员通过GitHub Issues提交问题反馈、提出功能建议，并通过Pull Request参与代码改进和文档优化，逐步形成了活跃的协作氛围。

目前，Python、JavaScript、Java、C#/.NET、Go等多种语言的A2A-SDK已对A2A提供了完整支持，例如CrewAI、LlamaIndex等Agent开发框架，都可以基于A2A-SDK便捷地与其他Agent集成。

Google自家的Agent Development Kit（ADK）是A2A理念的重要推动者。此外，大语言模型开源厂商LangChain Inc旗下的LangGraph框架，以图形化方式独具特色地展现了A2A的运行流程；BeeAI Framework、Pydantic AI和来自微软的Semantic Kernel，也都以极简的方法实现了A2A与其他

Agent的集成能力。

1.3　A2A 协议概览

A2A 协议由一组核心概念组成，作为一个侧重于消息传递的技术标准，基本通信元素和传输方式是协议最重要的组成部分。

1.3.1　A2A 核心概念

A2A使用一组核心概念来定义Agent如何交互，在开发或集成符合A2A标准的系统时，首先要明确这些概念，以便形成统一的术语环境。图1-1展现了A2A的主要核心概念和参与者。

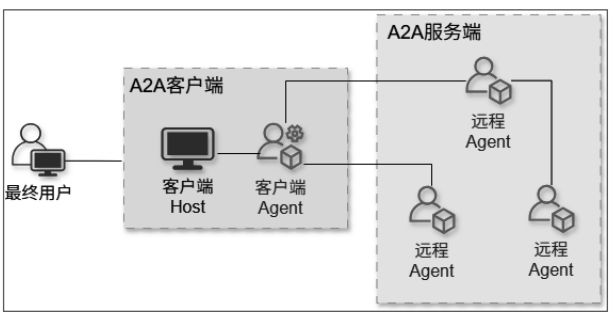

图 1-1　A2A 核心概念

1. 最终用户（End User）

最终用户指人工操作员或自动化服务，是操作客户端或发起服务调用的自动化程序。在Agent应用中，这里的最终用户不一定指人类，也可能是一个代理人类的AI执行者。

2. A2A客户端（A2A Client）

客户端是提供给最终用户操作的应用程序或服务。这里A2A Client的概念略显模糊，可能有以下几种情况：

（1）客户端Host：内嵌Agent的应用程序。最终用户操作的是这个程序，与远程Agent交互的是内嵌的Agent。

（2）客户端Agent：一个具体的Agent实现。由于可能出现远程Agent调用另外一个远程Agent的情况，发起调用的Agent相对于被调用者即为客户端Agent。

在大多数情况下，A2A Client体现的是客户端Host调用远程Agent的接口适配程序。

A2A（Agent-to-Agent）协议的字面含义是Agent到Agent。如果是客户端Host通过接口适配程序调用远程Agent，那么形式上更像“H2A”。A2A主要体现在远程Agent之间的调用，因此客户端

无论是一个Agent、一个Host，还是一个接口适配程序，在A2A的角色语义中都被当作一个“A”来看待，这样才更能体现A2A的交互理念。

3. A2A服务端（A2A Server）

A2A服务端是远程Agent提供服务的统称，由一个或多个以A2A协议公开服务的Agent组成，这些Agent也可通过A2A协议调用其他Agent，从而构成一个Agent网络。单个远程Agent通过公开一个实现了A2A协议的HTTP服务端点，接收来自A2A客户端的请求，处理任务并返回结果或状态更新。

从A2A客户端的角度来看，远程Agent是一个不透明的黑盒系统，其内部工作原理、模型调用过程或使用的工具通常不对外公开。

1.3.2　A2A基本通信元素

A2A客户端与服务器之间的通信元素主要有两种：一种是用于Agent发现的Agent Card和AgentSkill；另一种是双方交互时传送的消息，包括Message、Part、Artifact和Task等基本元素及其组合。

1. A2A中的Agent发现机制

1）Agent Card

Agent Card用于描述Agent的技能（Skill），由A2A Server定义。其内容以agent-card.json的形式公布在客户端可直接访问的公开网址上，通过该机制实现Agent之间的能力互相发现。经过简化的Agent Card包含的元素说明如下：

```
{
  "name": "Agent名称",
  "description": "Agent的描述，帮助用户和其他Agent理解其用途",
  "url": "与Agent交互的服务URL",
  "provider": "Agent服务提供者信息",
  "version": "Agent自身的版本号，格式由提供者定义",
  "capabilities": "Agent支持的能力声明",
  "skills": "Agent可以执行的技能或不同能力集合",
  "default_input_modes": "所有技能默认支持的输入MIME类型集合",
  "default_output_modes": "所有技能默认支持的输出MIME类型集合",
  "documentation_url": "Agent文档的URL",
  "icon_url": "Agent图标的URL",
  "preferred_transport": "首选传输协议，默认为JSONRPC",
  "protocol_version": "Agent支持的A2A协议版本",
  "additional_interfaces": "支持的其他接口列表",
  "security": "适用于Agent交互的安全要求对象列表",
  "security_schemes": "可用于授权请求的安全方案声明",
  "signatures": "为此Agent Card计算的JSON Web签名",
```

```
  "supports_authenticated_extended_card": "是否支持扩展的认证Card"
}
```

Agent Card相当于Agent的一张名片，上面的信息既可以让人读取，也可以让与之交互的Agent读取，主要包含Agent所具备的技能列表、输入与输出消息的格式，以及认证与安全方面的信息，如图1-2所示。

图 1-2　Agent Card 示意图

2）AgentSkill

AgentSkill以拟人化的方法说明Agent具备的能力，既用于人读，也用于Agent之间的自主协商，有时还用于与大模型的交流。AgentSkill以数组方式内嵌到Agent Card中，并通过agent-card.json一起公布给客户端，其简化后的结构如下：

```
{
  "description": "技能的详细描述，用于客户端或用户理解Agent的用途和功能",
  "examples": "此技能可以处理的示例提示或场景，为客户端提供使用该技能的提示",
  "id": "技能的唯一标识符",
  "input_modes": "此技能支持的输入MIME类型集合，将覆盖Agent的默认设置",
  "name": "技能名称",
  "output_modes": "此技能支持的输出MIME类型集合，将覆盖Agent的默认设置",
  "security": "技能所需的安全方案",
  "tags": "描述技能的关键词集合"
}
```

2. A2A的交互消息格式

1）Message

一条消息（Message）代表客户端与Agent之间的一次通信内容，包含发送角色（“user”或

“agent”）、唯一的messageId，以及名为parts的Part数组（包含一个或多个Part对象，这些对象是实际内容的细粒度容器；这样设计使A2A能够传递多种模态的数据）。Message简化后的结构如下：

```
{
  "context_id": "消息的上下文ID，用于对相关的交互进行分组",
  "extensions": "消息相关的扩展的URI列表",
  "kind": "对象的类型，用作鉴别器，对于消息始终为'message'",
  "message_id": "消息的唯一标识符，通常是由发送方生成的UUID",
  "metadata": "扩展的可选元数据",
  "parts": "构成消息正文的内容Part数组，可以是纯文本、JSON或文件",
  "reference_task_ids": "消息引用的其他任务ID列表，提供额外上下文",
  "role": "标识消息的发送方，分为'user'和'agent'",
  "task_id": "消息所属任务的ID"
}
```

2）Part

Part中存放的是消息携带的有效数据，分为TextPart、DataPart和FilePart三类。其中TextPart为纯文本；DataPart是JSON之类的结构化数据；FilePart中保存的是二进制文件内容（以字节流形式）或指向文件的URI（链接地址），用于处理图片、音频或视频等二进制数据。Part简化后的结构如下：

```
{
  "kind": "'text'、'file'或'data",
  "metadata": "关联的元数据",
  "text/data/file": "消息内容"
}
```

3）Task

Task代表一个交互任务，一般用于服务器端在处理长时间任务时，主动向客户端推送状态和消息的场景。Task由服务器端创建，随着任务的执行不断地向客户端推送Task状态更新消息；客户端则依靠响应TaskStatusUpdate事件获取状态和新增消息。当任务完成后，服务器端可发送一个工件（最终交付物）给客户端，客户端响应TaskArtifactUpdateEvent事件即可获得期望的完整成果。Task简化后的结构如下：

```
{
  "artifacts": "Agent在执行任务期间生成的工件（最终交付物）集合",
  "context_id": "服务器端生成的唯一上下文标识符，用于多个相关任务的关联",
  "history": "任务期间交换的消息数组，表示对话历史",
  "id": "任务的唯一标识符，新任务由服务器生成",
  "kind": "对象的类型，用作区分符，对于任务始终为'task'",
  "metadata": "扩展的可选元数据",
```

```
  "status": "任务的当前状态，包括状态值和传递的消息"
}
```

4）Artifact

Artifact代表Agent生成的工件（最终交付物），一般与Task配合使用。以一个Agent调用大模型生成长文章的Task为例，大模型生成的文本片段会不断以增量方式推送给客户端，客户端响应TaskStatusUpdate事件显示增量内容；这一过程中可能包含大量中间思考步骤，但这些内容通常不需要持久保存。服务器端在任务结束后，会交付一个完整的、最终的工件。Artifact简化后的结构如下：

```
{
  "artifact_id": "任务范围内工件的唯一标识符",
  "description": "工件描述",
  "extensions": "与此工件相关的扩展的URI列表",
  "metadata": "扩展的可选元数据",
  "name": "工件名称",
  "parts": "构成工件内容的Part数组"
}
```

1.3.3　A2A 传输方式

A2A 的传输以 HTTP 为主要技术基础，以异步消息传输为主要交互手段。

1. 传输协议核心要求

A2A 支持多种基于 HTTP 或 HTTPS 的传输协议，如 JSON-RPC 2.0、gRPC 和 HTTP+JSON/REST。Agent 可以根据自身的特定需求选择要实现的传输协议。A2A 对这 3 种传输协议的要求如表 1-1 所示。

表 1-1　A2A 对传输协议的核心要求

传输协议	核心要求
JSON-RPC 2.0	（1）所有请求和响应的数据格式必须为 JSON-RPC 2.0 （2）Content-Type 头部必须为 application/json （3）方法名称遵循{category}/{action}模式（例如"message/send"）
gRPC	（1）必须使用官方定义的 Protocol Buffers 文件（如 a2a.proto） （2）必须实现 A2A Service 中定义的 gRPC 服务 （3）必须支持 TLS 加密
HTTP+JSON/REST	（1）必须使用正确的 HTTP 请求方法（GET 用于查询，POST 用于操作等） （2）必须遵循规范的 URL 路径模式（例如/v1/message:send） （3）Content-Type 必须为 application/json

2. 异步传输机制说明

受限于大语言模型的生成效率，大部分AI驱动的操作通常运行时间较长，涉及多个步骤，因此一般采用增量生成内容的方法，以平衡用户体验和算力需求。为将服务端增量生成的内容及时展现给用户，客户端很少采用定时轮询的方法。A2A采取的方案主要是SSE或Task机制。

1）SSE

SSE是Server-Sent Events（服务器发送事件）的缩写，允许服务器主动向客户端（通常是Web浏览器）推送数据。在实现A2A服务端时，将Agent Card中capabilities的streaming设为True，客户端即可在服务端执行长任务过程中接收其主动推送的消息。

2）Task

A2A的Task机制提供了更灵活的异步交互方式。客户端以消费者订阅的方式响应服务器的任务状态更新，这些更新中可携带增量内容；在任务完成后，服务器会推送包含完整交付物的工件（Artifact）消息。这种“请求-订阅-响应”的方式在连接层面实现了客户端与服务端的解耦。

1.3.4　A2A 安全框架

A2A架构将Agent视为一种标准的企业级应用程序，其安全性完全依赖Web自身所具备的安全实现方案。A2A的安全框架包含传输通道安全、身份认证与授权三个层面。其中，配置高版本TLS加密的HTTPS是A2A应用在生产环境中的主要安全传输手段，可防止信息被拦截或篡改；在身份认证与授权方面，基于HTTP请求头的Bearer简易身份验证方法，以及基于OAuth2的动态Token机制，解决了A2A交互中的身份识别与资源访问权限控制问题。

在A2A的安全框架下，与调用者身份相关的信息不会直接嵌入JSON-RPC等协议的数据负载中，而是在HTTP传输层进行专门处理。这样既保证了信息的安全性，又符合现代Web应用的最佳实践。客户端可通过Agent Card中的特定字段发现服务器所需的身份验证方案，这些方案的名称通常与OpenAPI规范中的身份验证方法保持一致。例如，在实际操作中，客户端需要在HTTP请求头中添加类似“Authorization: Bearer <token>”这样的字段，以完成身份验证流程。此外，基于OAuth2类型的动态Token机制也是A2A身份认证中常见的实现方式之一，它通过动态生成和管理Token，进一步增强了身份验证的安全性和灵活性。客户端完成身份验证后，A2A服务器会根据已验证的客户端用户身份及自身的授权策略，精确控制该客户端可访问的资源范围，从而确保系统的安全性和资源的有效管理。

1.4　横向技术对比

A2A、MCP 以及大模型的 Function-Calling 特性均服务于 Agent 应用，它们之间存在一定的关联，但在使用场景和关注重点上有所区别。

1.4.1　A2A 与 MCP 对比

MCP是Model Context Protocol的缩写，直译为“模型上下文协议”，是一种用于将人工智能助手连接到各类数据源和工具的开放标准，由知名的人工智能企业Anthropic主导制定，并获得众多企业的支持，在生产环境中得到广泛应用。MCP和A2A都可用于构建复杂的AI应用系统，其主要差异在于Agent交互对象的不同。

MCP侧重于Agent或其他大模型应用程序访问外部工具和资源，以执行特定、通常是无状态的功能。其核心目的是标准化大模型、Agent、工具以及资源之间的连接方式和交互模式。

A2A则专注于Agent与Agent之间的互操作性。无论是调用方还是响应方，都是具备高度自主性的智能系统，需要在较长时间的交互中保持状态，并参与复杂、通常为多轮的对话，以处理更广泛、更复杂的问题。

MCP与A2A的侧重点比较如图1-3所示。

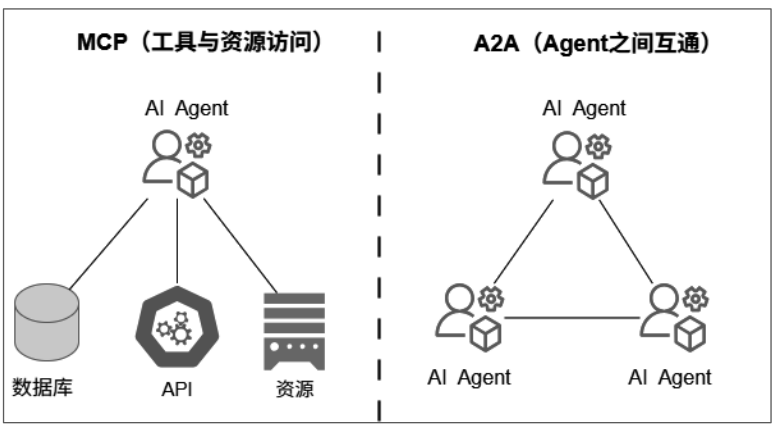

图 1-3　MCP 与 A2A 的侧重点比较

此外，MCP和A2A在实践中也存在大量互操作场景，例如：

- 将符合A2A规范的Agent作为MCP的资源纳入大模型应用的上下文中。
- A2A Agent调用由MCP封装的工具，以利用MCP提供的能力实现与特定工具的交互和对资源的访问。

1.4.2　A2A 与 Function-Calling 对比

Function-Calling作为大语言模型与外部工具交互的常见方式，主要聚焦于由模型根据用户输入动态生成结构化的函数调用指令，以调用特定工具或API。其核心目标在于解决模型无法直接完成的计算、数据查询或操作执行等任务，并弥补模型在交互逻辑上的单一性。典型流程通常表现为“用户提问→模型生成函数调用→Agent调用工具执行→工具返回结果→模型生成最终回答”的线性流程。

相比之下，A2A构建的是一个更全面、更复杂的Agent间交互体系，并不局限于简单的工具调用，而是强调具备自主能力的Agent之间的多轮协商、状态保持与协同工作。

从交互主体来看：

- Function-Calling的参与方主要是大语言模型与外部工具，Agent在其中更多扮演模型的“执行者”角色，按模型指令完成工具调用任务。
- A2A则以具备自主性的Agent作为平等的交互对象，每个Agent拥有独立的身份标识（如Agent Card中的信息）、技能集合（AgentSkill）以及任务处理能力（Task），能够基于自身能力和上下文进行自主决策与通信。

在应用场景来看：

- Function-Calling更适用于模型需要借助外部工具完成特定计算或查询的场景，如获取实时天气、调用计算器等。
- A2A则适用于需要多个Agent协同处理复杂任务的场景，如智能客服系统中负责意图识别的Agent与负责业务办理的Agent之间的协作处理流程。

1.5　本章小结

本章首先介绍了A2A协议的核心概念、主要特性、社区生态和发展历程；随后系统阐述了协议的关键构成要素，包括Agent Card、Message、Part、Task和Artifact等的结构与功能；接着讨论了A2A的身份认证与授权机制，强调其依赖Web安全实践，将身份信息置于HTTP传输层，并通过动态Token等方式保障通信安全；最后，通过与MCP和Function-Calling的横向对比，明确了A2A在Agent间的互操作性、多轮协商、状态保持以及复杂任务协同等方面的独特优势与适用场景。

第 2 章

极简入门：快速构建第一个A2A应用

2

在学习智能体应用开发的过程中，动手搭建实验环境并练习编写简单示例代码至关重要。这不仅能帮助学习者更好地理解理论知识，也能加速将概念转化为实际技能的过程。本章将深入讲解实践本书所有案例所必需的软件，包括提供大模型服务的Ollama和用于管理Python虚拟环境的Miniconda[1]。为了帮助读者快速入门A2A，本章将通过一个极简但涵盖A2A基本元素的案例，讲解构建最小A2A应用的完成流程。

2.1 开发环境全栈指南

为了简化实验环境，本书选用了读者最易获取的系统和工具，操作系统采用Windows，即使在未配备GPU的计算机上，也能够完整实践本书的全部内容。由于A2A的核心目标是打通Agent之间的交流通道，而Agent本身是一种基于大模型的应用形式，因此需要搭建能够提供文本生成功能的大模型服务。

A2A-SDK与相关生态提供了多种语言的实现。为了降低学习成本，本书统一选用Python语言作为所有案例的开发语言，避免跨语言示例造成知识分散，从而影响读者对A2A本身的理解。这一选择的另一个好处是：Python简洁优雅，示例代码通常不超过60行。本书提供完整、可直接运行的示例代码，读者无须在阅读本书与查阅外部网站之间来回切换，从而保持专注。

在操作系统上直接安装Python来运行AI应用并不是理想做法，因为不同应用往往依赖不同版本

[1] Python发行版Anaconda的精简版本，专注于Python环境隔离。

的Python或多个应用所使用的依赖包之间存在冲突。为此，本书采用Miniconda，通过虚拟化方式为各章创建自成体系、相互隔离的Python运行环境。图2-1展示了本书实践部分所使用的开发环境技术栈的概览。

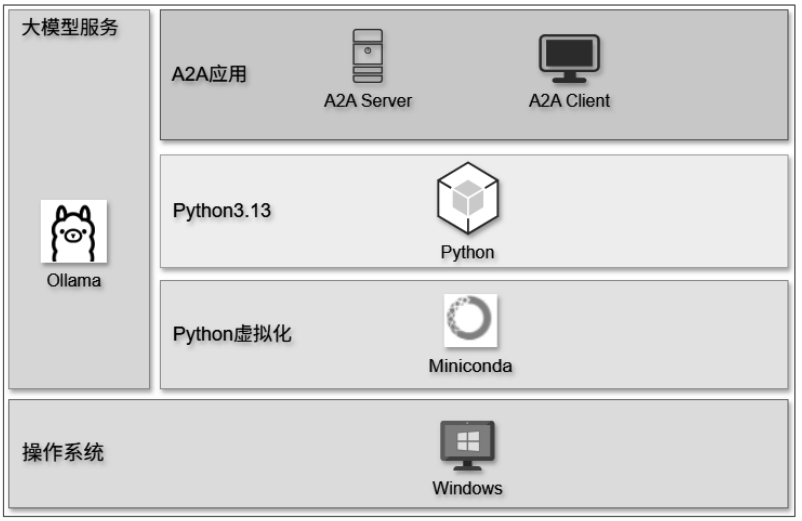

图 2-1　开发环境技术栈概览

2.1.1　操作系统要求

本书选用Windows作为实践环境的操作系统，主要目的是降低读者的上手门槛。对大多数开发者来说，Windows是最易获得、最普及的操作系统。如果读者仅拥有一台只安装了Windows的笔记本电脑，也完全能够运行本书所有章节的案例。无论是在个人学习设备还是企业内部的开发工作站上，Windows系统都具有广泛的适用性，能够满足不同场景下的开发需求，为后续的大模型服务搭建、Python虚拟环境管理等任务提供稳定的平台支撑。此外，Windows丰富的图形化界面与便捷的操作方式，有助于开发者更直观地进行环境配置和故障排查，从而降低因系统操作复杂度而带来的学习成本。

2.1.2　大模型服务要求

本书选用Ollama作为大模型服务的提供工具，同样是基于降低实践难度的考虑。许多读者可能并不具备GPU算力资源，而Ollama能够在CPU环境下加载大模型权重，并提供标准的OpenAI Chat兼容接口，为Agent提供文本生成服务。考虑到许多Agent需要使用大模型的Function-Calling特性，以实现主动调用工具的能力，本书在模型的选择上采用通义千问的Qwen3系列作为推理模型，因为它们完整支持Function-Calling特性。然而，需要特别指出的是，Ollama加载的是INT4量化版本的模型，其精度仅为GPU上运行的原始半精度（FP16）模型的1/4。以Qwen3为例，INT4数据类型用−8~+7

的16个整数表达原始模型的81.9亿参数，即使结合了优化的缩放算法，精度损失带来的生成质量下降仍然十分明显。

因此，基于Ollama的模型仅适合作为开发环境下“可用但受限”的推理支撑，用于程序调试并无问题，但不适用于大多数生产级场景。

需要说明的是，选用Qwen3主要是为了保持全书案例的一致性与描述便利。实际上，只要所选模型支持Function-Calling功能，即可作为本书示例的底层推理模型，且更换模型在安装配置和代码适配方面都非常简单。

2.1.3　Python 虚拟化环境

直接用操作系统中全局安装的Python来运行AI程序，仅适用于功能简单、场景单一的应用。当在同一台机器上运行多个Python项目时，各类环境冲突问题出现的概率将显著增加。本书的技术栈选择在操作系统和Python运行环境之间增加一层虚拟化的中间层，目的在于减少读者在调试书中源代码时遇到的Python版本冲突、各章案例依赖库冲突等问题。

1. 虚拟化的必要性

目前常用的Python版本范围为3.7至3.13。虽然新版本通常兼容旧版本，但版本跨度较大时，依然可能出现兼容性问题。即使是相邻版本，如3.12和3.13，也可能导致某些依赖库无法正常运行。例如，在A2A-Python中就存在依赖Python特定版本的情况：Python 3.13为asyncio.Queue新增了shutdown()方法，提供非阻塞的队列关闭方式，可避免潜在的死锁问题。因此，建议根据项目需求选择适配的Python版本，而非盲目追求最新版本。

简单项目的依赖库较少，项目的依赖库数量会随着项目复杂度的提升而增加。即便项目看似只依赖一个库，该库本身可能依赖多个第三方库，而这些依赖在版本更新时不一定保持向后兼容。如果项目中未明确指定依赖库的版本，或仅指定了最小兼容版本，随着时间推移，新版本可能由于移除了某些方法、修改函数入参个数、升级内部依赖等原因，导致项目无法正常运行。在大模型应用中，Transformers、PyTorch、NumPy等库更新频繁，版本不兼容问题更是常见。同一台机器上的多个应用往往对某些库的版本要求不同，由此引发的冲突也相对较多，而使用Python虚拟环境可以实现项目间的隔离，是解决此类问题的最佳方式。

除了解决Python环境的冲突问题外，Python虚拟化还能为每个项目创建独立的运行空间，使项目的依赖管理更加清晰、高效。开发者可以针对不同项目分别安装所需的库及其对应版本，而无须担心影响其他项目。本书将为每一章创建独立的虚拟环境，用于管理依赖关系。当需要将项目迁移到其他机器时，只需导出虚拟环境的依赖配置文件，即可在目标机器上快速重建相同的开发环境。这不仅降低了环境配置的复杂度，也大幅减少了时间成本。

2. 虚拟化工具的选择

Python的虚拟化方案包括Anaconda、Poetry、venv和uv等多种工具。本书在进行技术选型时，

综合考虑了安装便利性、镜像生态完善程度以及适用场景等因素，最终选择了Anaconda的精简版本——Miniconda。下面简要说明未选用其他方案的原因。

Poetry是一款功能强大的Python虚拟化环境和依赖管理工具，尤其擅长处理复杂的库依赖关系，但它的国内镜像资源相对较少，安装和拉取依赖时可能不够顺畅。

venv是Python内置的虚拟化工具，不选择venv的主要原因是它依赖系统中已有的Python，需要安装Python后才能创建环境。从“在Python里再安装一个用于隔离的Python环境”的操作逻辑来看，多少显得有些别扭。

uv是一个轻量、绿色且功能强大的工具，只需一个uv.exe即可运行。但它的不足在于每执行一条命令都需要加上“uv run”前缀，使用体验更像在执行uv的命令，而非直接操作Python。对初学者而言稍显麻烦。uv更适合MCP等自动化场景，因为此类场景下调用程序的命令通常由MCP Host程序发起，而非人工输入。

需要说明的是，最终选择Miniconda也有笔者个人习惯的因素。实际上，上述所有工具都能够满足本书的实践需求，并可用于生产环境。

Miniconda直接安装在操作系统中，不依赖系统自带的Python。通过Miniconda创建的虚拟环境与当前工作目录无关；使用时只需分清当前终端提示符位于哪一个环境（非虚拟环境的命令提示符前一般会显示base字样），即可在任意目录执行pip、python等命令。由Miniconda创建的虚拟环境对应用具有良好的隔离效果，如图2-2所示。

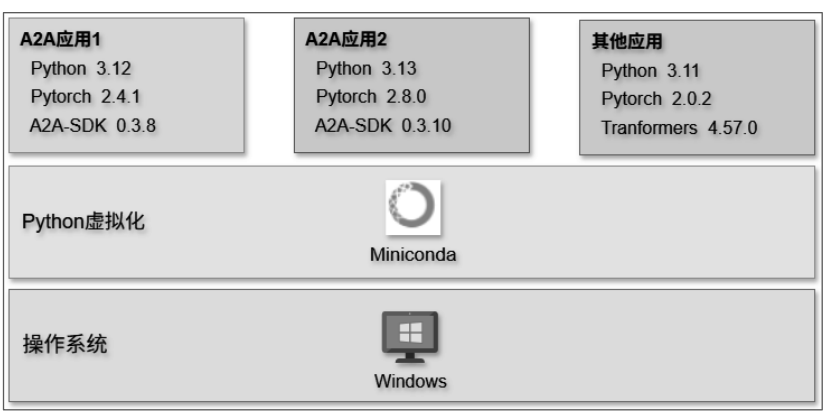

图 2-2　Miniconda 虚拟环境隔离示意图

2.2　实验环境安装与配置

本书的实验环境设计充分考虑了读者的资源易获得性与操作便捷性，因此未采用需要 GPU 才能运行的高精度大模型方案，而是选择了能够运行量化模型、满足 Agent 调试要求的 Ollama 平台。此外，为了隔离每章的 Python 依赖环境，还需安装 Miniconda。

2.2.1　Ollama 安装与配置

Windows版的Ollama安装程序可从Ollama的官方网站下载。下载的安装包为OllamaSetup.exe。如果下载时遇到困难，可使用镜像站点https://aliendao.cn/ollama#。

运行OllamaSetup.exe，保持默认选项即可完成安装。安装完成后，为减少对C盘空间的占用，可将操作系统的环境变量OLLAMA_MODELS设置为其他磁盘路径，使新下载的模型存放在指定目录而非默认的C盘。

若需要让其他机器访问本机的Ollama服务，可将环境变量OLLAMA_HOST设置为“0.0.0.0”，而非默认仅允许本地访问的127.0.0.1:11434。需要注意，这样会带来暴露服务端口的安全风险，应谨慎设置。

另外，Ollama在Windows中以系统服务的方式运行，因此修改环境变量后需要重启Ollama服务。

在Windows命令行下，可通过以下命令从Ollama官网拉取Qwen3模型并将模型加载到内存进行推理测试：

```
# 拉取模型
ollama pull qwen3
# 运行模型
ollama run qwen3
```

ollama run会启动交互方式推理，用户输入问题后模型将返回生成结果，如图2-3所示。

```
C:\Users\gqw>ollama run qwen3
>>> 你好，qwen!
Thinking...
好的，用户打招呼说“你好，qwen!”，我需要回应。首先，要确认用户是否在测试我的反应，或者只是友好地打招呼。根据之前的
指示，我应该保持友好和专业的态度。

接下来，我需要用中文回应，保持口语化，避免使用Markdown格式。要确保回答简洁明了，同时提供帮助。比如，可以回应“你好！
有什么我可以帮助你的吗？”，这样既友好又开放，鼓励用户提出问题或请求。

另外，要注意用户可能有不同的需求，比如询问信息、寻求帮助，或者只是聊天。所以保持回答的灵活性很重要。同时，避免使用复
杂术语，保持自然的口语表达。

最后，检查是否有需要进一步优化的地方，比如是否需要添加表情符号或更生动的语言，但根据指示，保持简洁即可。确认没有遗漏
任何关键点后，生成最终的回应。
...done thinking.

你好！有什么我可以帮助你的吗？ 
```

图 2-3　Ollama 推理测试

需要说明的是，ollama run命令可自动拉取镜像，在对外提供生成式API服务时，并不需要手动执行该命令。Ollama的后台服务会根据API调用自动加载模型并完成推理。

2.2.2　Miniconda 的安装与验证

Miniconda可从Anaconda的官网下载。运行下载的Miniconda3-latest-Windows-x86_64.exe，按默认选项完成安装。建议将Miniconda安装目录中的Scripts子目录添加到操作系统环境变量Path中，以便于在执行conda命令时，操作系统能正确找到conda.exe。

安装完成后，可通过conda -V命令来验证安装，若执行结果显示出Miniconda的版本号，则说明安装成功。

2.3　入门案例场景设计

本章的入门案例旨在完成一个极简但具有代表性的任务：构建一个能够获取当前时间的Agent，并将其封装为一个A2A Server。当A2A Client发送请求时，该Server能准确返回当前的日期和时间。虽然场景非常简单，却涵盖了A2A应用开发中的核心要素，包括清晰的服务接口定义、规范的消息格式封装以及客户端与服务端的通信交互流程设计等，有助于读者直观理解A2A协议在实际应用中的基本运作机制。

通过这个案例，读者可以快速掌握如何基于现有技术栈搭建一个可运行的A2A应用框架，为后续学习更复杂的多Agent协作、状态管理等扩展功能奠定基础。

乍看之下，“获取当前时间”是一个极其简单的功能，但在实际应用中却具有重要意义。例如，在基于大模型的聊天应用中，如果不借助外部工具，仅依靠大模型自身的生成能力，无法获取与“当前时刻”相关的信息。当用户询问“历史上的今天”之类的问题时，模型并不知道“今天”究竟是哪一天，因而无法生成相应的答案。

在Agent应用中，这类任务通常依赖大模型的Function-Calling功能，由模型调用Agent内部提供的工具来获取当前时间。工具返回的时间信息会写入Agent与大模型的上下文，从而使大模型能够正确理解“当前时间”，进而生成与当前时间相关的内容。

在A2A应用中，通信双方均为Agent。如果某个Agent需要借助大模型完成特定功能，而这些功能又依赖当前时间，那么自然需要通过专门的时间获取工具来确保功能的正确实现。本章的案例正是为这一类场景提供基础模式与实现思路。

2.4　架构与核心组件

本例遵循A2A模式下的典型Client/Server架构。A2A的设计哲学强调“公开细节以换取更高的灵活性”，这一理念在A2A Server所暴露的大量技术细节中体现得尤为明显。

一个典型的A2A Server通常由以下核心组件构成：用于描述服务能力的Agent Card定义、负责接收客户端请求的HTTP Server、代表传输协议并处理消息格式的A2A Application、维护长生命周期状态并进行请求分发的RequestHandler、负责任务数据持久化的TaskStore、调度并执行Agent的AgentExecutor，以及实现具体业务逻辑的Agent。A2A Client则封装在一个Client对象中。Client会从A2A Server获取Agent Card的内容，形成调用所需的技术参数，随后向A2A Server发送消息，并处理服务端返回的推送响应。整体架构如图2-4所示。

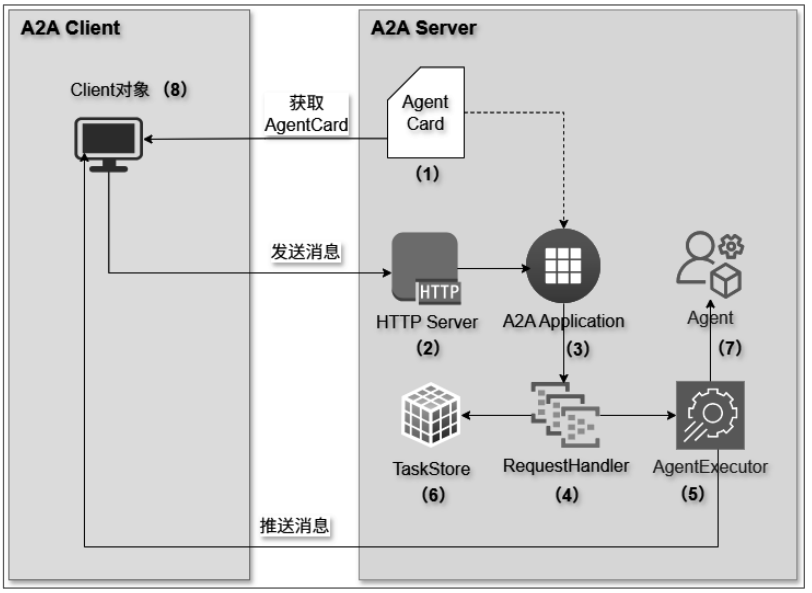

图 2-4　A2A 应用典型架构

2.4.1　Agent Card 服务能力描述

Agent Card是A2A中用于描述Agent服务能力的核心元数据，相当于一份标准化的服务说明书。它详细定义了Agent所提供服务的接口信息、功能描述、参数规范、返回结果格式以及通信协议要求等关键内容。借助Agent Card，A2A Client能够清晰了解Server端Agent的能力范围，包括支持的功能及其调用举例等。通过此类标准化定义，Client可根据自身需求准确选择并调用Server端的服务，而无须关心Server内部复杂的实现逻辑。

2.4.2　HTTP Server 通信通道的实现

A2A支持JSON-RPC 2.0、HTTP+JSON/REST和gRPC三种通道，其中前两种基于HTTP Server监听客户端请求。在实际部署中，HTTP Server需要应对高并发访问，因此在架构上采用异步处理机制。系统使用基于Starlette[1]构建的FastAPI[2]框架作为HTTP服务基础，可高效支持异步请求处理，从而显著提升服务吞吐量。

2.4.3　A2A Application HTTP 服务载体

A2A Application是A2A在HTTP服务层面的具体实现载体，继承自Starlette的Application类，负

[1] 一种轻量级ASGI（异步服务器网关接口）框架/工具包，适用于Python构建异步Web服务。

[2] 一种高性能的Python Web开发框架。

责将A2A规范与HTTP通信机制融合。其主要职责包括：

- 对A2A消息格式进行解析和封装。
- 完成请求的路由分发。
- 与后续的RequestHandler组件衔接。

当HTTP Server接收到客户端发送的请求后，会先将原始数据交由A2A Application处理。A2A Application会按照A2A定义的消息结构解码请求，提取关键信息，如目标服务标识与调用参数等，再依据路由规则将处理后的请求分发至对应的RequestHandler。在生成响应时，A2A Application会按A2A协议要求封装业务处理结果，确保客户端能够正确解析与理解响应内容。

2.4.4　RequestHandler 的分发职责

RequestHandler是A2A Server中负责请求分发与长状态（stateful）管理的核心组件。其主要职责包括：

- 接收A2A Application传递的请求。
- 根据服务标识和参数信息匹配相应业务逻辑。
- 管理请求过程中的长状态数据。

在处理请求时，RequestHandler会首先校验请求的合法性，例如参数完整性与格式规范性。校验通过后，将请求根据路由映射关系分发给对应的业务处理单元（如对应的Agent执行器）。对于需要保持会话状态的场景（如多轮交互），RequestHandler会通过内置的状态管理机制记录每个请求的上下文信息，确保后续请求能够基于历史交互数据进行准确处理，避免因状态丢失导致业务逻辑中断。

2.4.5　TaskStore 持久化方案

TaskStore是A2A Server中用于任务数据持久化存储与管理的组件，主要负责保存任务的执行状态、历史记录及上下文信息等。它为RequestHandler的长状态管理和AgentExecutor的任务调度提供持久的上下文关联数据支持。TaskStore常见的实现方式包括数据库或内存存储。具体选型需根据业务场景的性能要求、数据规模和持久化需求综合确定。

2.4.6　AgentExecutor 调度者角色

AgentExecutor是A2A Server中负责调度并执行Agent任务的核心执行组件。它在接收来自RequestHandler分发的业务请求后，会根据请求的具体内容和参数调用对应的Agent实例来完成实际的业务逻辑处理。

在执行过程中，AgentExecutor需要实时监控Agent的运行状态，包括任务启动、执行进度跟踪、异常捕获与任务取消等。同时，AgentExecutor会与TaskStore进行交互，在任务执行的关键节点将

相关的任务数据持久化存储到TaskStore中，以便后续进行任务追踪与审计。

2.4.7　Agent 具体业务实现

Agent是A2A应用中的业务实现组件，直接承载了具体的业务逻辑和功能。Agent通常基于相应的Agent框架开发，如LangGraph、CrewAI、Semantic Kernel等。这些框架应用大语言模型的生成能力、多种角色协同、提示词工程以及工作流等技术开发Agent应用。在通过A2A协议与其他Agent整合时，Agent需要对外公开可调用接口，并交由AgentExecutor负责统一调度与执行。

2.4.8　Client 请求发起方的职能

在A2A模式的Client/Server架构中，Client作为发起服务请求的一方，其核心功能在于与Server建立稳定的通信连接，并按照A2A规范完成完整的服务调用流程。当Client需要调用Server提供的服务时，应首先获取Server的Agent Card信息，然后构造请求消息，并通过通信通道（如HTTP或gRPC）将请求发送至Server指定端口，随后接收返回结果。这个过程通常以异步方式执行。此外，Client还需维护与Server的会话状态，尤其在多轮交互场景中，要确保上下文信息的连续性，使Server能够基于历史交互数据正确理解后续请求。

2.5　最简 A2A 应用实现过程

一个极简但功能完整的入门案例（包括代码、环境安装、运行调试过程说明等），是帮助读者快速了解 A2A 应用开发的重要实践方式。

2.5.1　源代码结构详解

根据 A2A 的架构设计，源代码分为 A2A Server 和 A2A Client 两个部分。

1. A2A Server源代码

time_agent_server.py实现了一个对外提供当前时间查询服务的Agent，尽管代码只有50多行，却涵盖了A2A规范的大部分核心内容。

1）引入依赖库

```
import uvicorn
from datetime import datetime
from a2a.server.apps import A2AStarletteApplication
from a2a.server.request_handlers import DefaultRequestHandler
from a2a.server.agent_execution import AgentExecutor, RequestContext
from a2a.server.tasks import InMemoryTaskStore
from a2a.server.events import EventQueue
```

```
from a2a.types import AgentCapabilities, AgentCard, AgentSkill
from a2a.utils import new_agent_text_message
```

各依赖库或类的用途说明如表2-1所示。

表 2-1　依赖库或类的用途说明

依赖库或类	用途说明
uvicorn	基于 ASGI 的高性能 Web 服务器，用于启动 HTTP 服务器
datetime	提供了获取系统当前日期和时间的功能，供 Agent 调用
A2AStarletteApplication	用于构建符合 A2A 规范的 HTTP 应用
DefaultRequestHandler	负责请求分发、合法性校验及长状态数据管理
AgentExecutor	任务调度者，接收请求后调用具体的 Agent 实例处理业务逻辑，并与 TaskStore 交互进行数据持久化
InMemoryTaskStore	基于内存的任务数据存储实现，用于临时保存任务执行过程中的状态和上下文信息
EventQueue、RequestContext	用于处理 A2A 应用中的事件消息，实现组件间的异步通信
AgentCapabilities、AgentCard、AgentSkill	用于定义 Agent 的服务能力、接口信息等元数据
new_agent_text_message	按照 A2A 规范封装 Agent 返回的文本消息

2）定义 Agent Card

```
skill = AgentSkill(
    id='current-time-skill',
    name='当前时间查询',
    description='获取当前的系统时间',
    tags=['时间服务', '实时查询', '工具类'],
    examples=['现在几点了？', '当前时间是多少？'],
)

public_agent_card = AgentCard(
    name='时间服务智能体',
    description='时间服务智能体，提供时间相关服务',
    url='http://localhost:9998/',
    version='1.0.0',
    default_input_modes=['text'],
    default_output_modes=['text'],
    capabilities=AgentCapabilities(streaming=True),
    skills=[skill],
    supports_authenticated_extended_card=True,
)
```

Agent的业务能力通过AgentSkill实例进行描述，而该实例是Agent Card的一个属性。Capabilities字段中设置的“streaming=True”表示A2A Server支持流式传输。程序运行后，Agent Card的相关信息在程序运行时会在URLhttp://localhost:9998/.well-known/agent-card.json中向客户端公开。

3）实现 Agent

```
class TimeAgent:
    async def invoke(self) -> str:
        return str(datetime.now())
```

TimeAgent提供了一个名为invoke的函数，以异步方式返回系统当前的时间。这部分代码即为本案例中最核心、最直接的业务逻辑实现。

4）实现 AgentExecutor

```
class TimeAgentExecutor(AgentExecutor):
    def __init__(self):
        self.agent = TimeAgent()

    async def execute(
        self,
        context: RequestContext,
        event_queue: EventQueue,
    ) -> None:
        text = context.get_user_input()
        print("客户端输入: " + text)
        result = await self.agent.invoke()
        await event_queue.enqueue_event(
            new_agent_text_message(result))

    async def cancel(
        self, context: RequestContext, event_queue: EventQueue
    ) -> None:
        raise Exception('cancel not supported')
```

TimeAgentExecutor 是 TimeAgent 的执行器，用于调度 Agent 并向客户端推送消息。作为AgentExecutor 的子类，它必须实现 execute 和 cancel 两个方法。在 execute 方法中，执行流程如下：

（1）通过RequestContext获取客户端传入的文本信息。

（2）调用TimeAgent.invoke()方法获取当前时间。

（3）使用new_agent_text_message工具函数将结果封装成符合A2A规范的消息格式。

（4）通过EventQueue将消息加入事件队列，以便异步推送给客户端。

cancel方法在本示例中暂未实现具体逻辑，而是直接抛出“不支持取消操作”的异常。实际应用中可根据业务需求补充状态清理、任务终止等处理逻辑。当客户端主动调用cancel_task时，cancel方法会被触发执行。

execute方法中的几个重点解读如下：

（1）客户端传过来的消息通过context.get_user_input()获取。通过context还可以得到context_Id等代表消息上下文的信息，这在客户端被多次调用、需要利用历史会话记录的场景中非常有用。

（2）TimeAgent的invoke是异步方法，这里通过await简单地将其作为同步调用来获取返回值。事实上，在大部分Agent的实现中，由于需要与大语言模型交互，通常会以逐步生成增量内容的异步调用方式实现。在execute中，往往也需要采取异步的方式接收Agent的返回结果，再将其推送给客户端。

（3）与常规的HTTP请求应答模式不同，A2A在客户端和服务端之间采用异步方式传递消息，机制上是基于事件驱动。本例中的体现是：将生成的文本消息推送到事件队列中，供后续异步传输给客户端。

5）RequestHandler

```
request_handler = DefaultRequestHandler(
    agent_executor=TimeAgentExecutor(),
    task_store=InMemoryTaskStore(),
)
```

DefaultRequestHandler的实例负责将客户端的请求分发给AgentExecutor去处理。为了管理任务的上下文信息，这里使用InMemoryTaskStore在内存中存储任务执行过程中的状态数据。

6）*启动* A2A Server

```
server = A2AStarletteApplication(
    agent_card=public_agent_card,
    http_handler=request_handler
)

uvicorn.run(server.build(), host='0.0.0.0', port=9998)
```

这里定义了一个支持JSON-RPC 2.0通道的A2AStarletteApplication，并通过uvicorn启动服务，监听本机的9998端口以接收客户端的请求。

2. A2A Client源代码

time_agent_client.py使用极简的代码调用A2A Server，总共20多行代码。

1）引入依赖库

```
import asyncio
from uuid import uuid4
from a2a.client import (
   ClientConfig, ClientFactory, minimal_agent_card)
from a2a.types import Message, Role, TextPart
from a2a.utils.message import get_message_text
```

依赖库或类的用途说明如表2-2所示。

表 2-2　依赖库或类的用途说明

依赖库或类	用途说明
asyncio	提供异步 I/O 操作的支持，用于实现客户端与服务端的异步通信逻辑，如发起请求、处理响应等
uuid4	用于生成 UUID（通用唯一标识符），可作为客户端请求的 context_Id 或 task_id，确保会话或任务的唯一性
ClientConfig	用于配置客户端连接参数（如服务端 URL、超时时间等）
ClientFactory	基于配置创建 A2A 客户端实例，封装与服务端建立连接的底层逻辑
minimal_agent_card	构建简化版的 AgentCard 实例，客户端可通过该实例声明自身能力或解析服务端返回的 Agent 元数据
Message	A2A 的消息结构，包含消息内容、角色和消息部分
Role	A2A 消息中的角色类型，包括 User 和 Agent 两种情况
TextPart	A2A 消息中的文本内容载体
get_message_text	从 Message 对象中提取文本内容的工具函数，简化客户端对服务端返回消息的解析过程，快速获取 Agent 生成的文本结果

2）调用 A2A Server

```
async def main():
    _client_factory = ClientFactory(ClientConfig())
    client = _client_factory.create(
        minimal_agent_card('http://localhost:9998')
    )
    msg = Message(
        kind='message',
        message_id=uuid4().hex,
        role=Role.user,
        parts=[TextPart(text='你好')]
    )
    async for response in client.send_message(msg):
```

```
        print(get_message_text(response))

if __name__ == "__main__":
    asyncio.run(main())
```

首先通过minimal_agent_card构造（或获取）A2A Server公布的Agent Card信息，并使用ClientFactory创建A2A Client对象；然后构造一个包含TextPart的Message对象，通过client.send_message(msg)发起调用，以异步迭代的方式处理流式响应，并解析返回的消息。最后的主函数为标准的异步程序启动方式，使用asyncio运行主协程。

2.5.2　运行最简 A2A 应用

首先安装运行环境，然后分别在不同的命令行窗口运行 A2A Server 和 A2A Client，观察应用两端的运行情况。

1. 环境安装

创建一个库依赖关系文件requirements.txt，内容如下：

```
a2a-sdk[http-server]==0.3.10
uvicorn==0.38.0
```

然后运行以下命令安装程序的运行环境。

```
# 创建虚拟环境
conda create -n chapter02 python=3.13 -y
# 激活虚拟环境
conda activate chapter02
# 安装依赖库
pip install -r requirements.txt -i https://pypi.mirrors.ustc.edu.cn/simple
```

2. 运行A2A Server

打开一个命令行窗口，运行以下命令：

```
conda activate chapter02
python time_agent_server.py
```

运行结果如图2-5所示。

```
(chapter02) E:\program\little51\a2a-course\chapter02>conda activate chapter02

(chapter02) E:\program\little51\a2a-course\chapter02>python time_agent_server.py
□[32mINFO□[0m:     Started server process [□[36m151188□[0m]
□[32mINFO□[0m:     Waiting for application startup.
□[32mINFO□[0m:     Application startup complete.
□[32mINFO□[0m:     Uvicorn running on □[1mhttp://0.0.0.0:9998□[0m (Press CTRL+C to quit)
```

图 2-5　A2A Server 运行情况

在浏览器中访问“http://127.0.0.1:9998/.well-known/agent-card.json”，可以查看A2A Server公布的Agent Card信息，结果如下：


```
{
  "capabilities": {
    "streaming": true
  },
  "defaultInputModes": [
    "text"
  ],
  "defaultOutputModes": [
    "text"
  ],
  "description": "时间服务智能体，提供时间相关服务",
  "name": "时间服务智能体",
  "preferredTransport": "JSONRPC",
  "protocolVersion": "0.3.0",
  "skills": [
    {
      "description": "获取当前的系统时间",
      "examples": [
        "现在几点了？",
        "当前时间是多少？"
      ],
      "id": "current-time-skill",
      "name": "当前时间查询",
      "tags": [
        "时间服务",
        "实时查询",
        "工具类"
      ]
    }
  ],
  "supportsAuthenticatedExtendedCard": true,
  "url": "http://localhost:9998/",
  "version": "1.0.0"
}
```

3. 运行A2A Client

另开一个命令行窗口，运行以下命令：

```
conda activate chapter02
python time_agent_client.py
```

运行结果为输出当前时间。

2.6　本章小结

本章主要讲解了开发环境的搭建与一个简单示例，目的是帮助读者快速熟悉A2A在实际开发中的应用流程。本章从环境搭建入手，以获取当前时间的TimeAgent应用为例，详细解读了A2A应用的架构，并对A2A Server和A2A Client的实现进行了详细解读，为读者基于A2A开发更复杂的Agent应用奠定了基础。后续章节将在此基础上深入探讨更多高级特性和实际业务场景的应用。