**任务1.1：基于运行时行为的微服务智能拆分工具**

传统单体软件系统各功能模块高度耦合，依赖程度高，系统规模化也导致了代码库庞大、扩展和维护成本不断增加。此外，微服务拆分过程中边界划分和服务拆分工作难度大，需要丰富经验和对业务流程和软件架构的深入理解，人工成本极高。针对这些问题，本课题将解决微服务拆分过程中的微服务合理划分和系统正确拆分问题。具体而言，如何合理地进行微服务拆分，在降低耦合度的同时不影响系统整体的性能和功能，是单体架构的云原生应用进行微服务架构迁移的关键问题。本课题将设计实现一种基于运行时行为分析的微服务智能拆分工具，自动化识别、拆分、合并系统中的功能模块，提供更合适的微服务架构重构方法，从而帮助开发人员合理划分服务边界、优化业务流程，降低手动拆分难度和错误风险，显著提升中大型系统向微服务架构迁移的效率。具体方案如下：

**（1）运行时行为的监控和收集**

传统的静态分析方法在微服务的划分任务主要通过对代码中的调用关系进行收集和分析，排除一些冗余代码。然而，在实际的微服务划分任务中，各模块之间的调用频率和数据交互量等特征在简单的静态分析中无法得出，并且在实际使用场景中，用户的访问量、系统硬件的负载等特征也远超静态分析的范畴，这些都使得静态分析难以准确把握微服务化分过程中各模块之间的实际依赖关系而存在潜在的性能问题。因此，本课题拟采用动态分析手段指导重构与拆分，从系统运行时行为入手，对系统的实际运行场景进行监控和分析，得到真实准确的系统运行时信息，如系统资源使用、响应时间、用户访问量等，并将获得的运行时信息根据实际情况转变为模块间的关联度和耦合度，构建完整的模块依赖网络，为后续微服务架构的重构提供坚实的数据支撑。

为了全面收集系统运行时行为数据，如图 4所示，我们将监控调用链与请求、数据库操作、服务间通信等多种运行时状态，使用日志收集工具进行收集。为更好地模拟用户实际操作和真实用户访问量，以生成具有代表性的系统运行时行为数据，拟使用软件测试工具，如Postman或Apifox等工具，并编写完整的集成测试用例，依据不同的业务场景，精确模拟用户对系统的访问行为。通过控制不同测试用例的调用频率和数据量来模拟用户对于不同功能的实际需求。例如在电商应用场景中，显著增加订单创建、支付等核心功能的调用频率，同时增加商品查询功能的数据量，以获得系统在高并发业务场景情况下的实际运行状态数据，为后续系统的重构和对重构架构的评估提供依据。

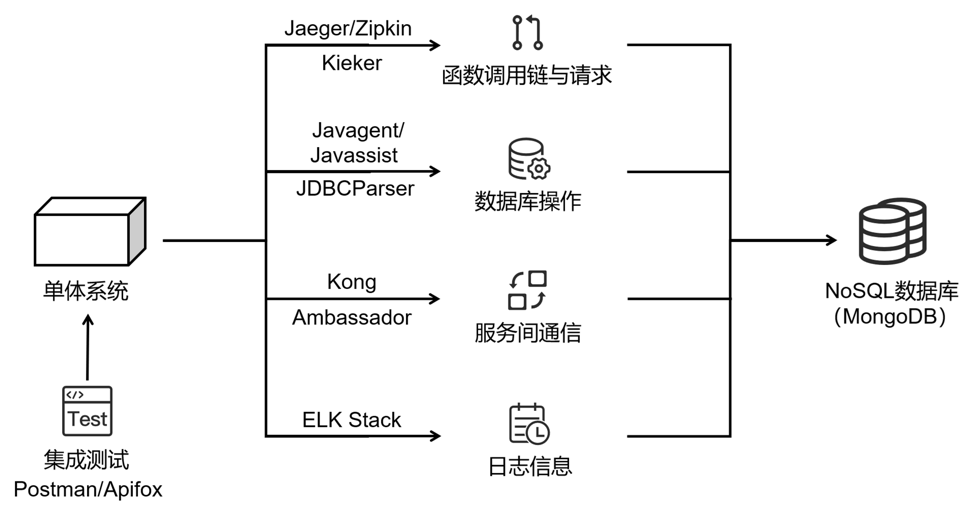


图 4运行时行为数据收集

**调用链与请求追踪：**为更好地收集所有模块和服务之间的调用关系，拟使用Kieker开源工具对系统进行字节码插桩以获取请求的执行时间、资源消耗、调用链信息等数据。字节码插桩技术允许在程序运行时动态修改或增强程序的字节码，使得Kieker能在应用程序的不同地方插入监控点，从而监控并获取详细的运行时数据。在进行集成测试的同时，监控并记录系统各函数、模块的详细运行时数据，获取函数间调用关系以及执行时间等，并且以日志的形式进行保存，为后续依赖关系网络的构建提供依据。此外，Kieker可支持对用户操作加权，因而能区分不同测试样例之间的重要程度和参考价值。

**数据库操作追踪：**为收集数据库表间的关系以及系统对于数据库的访问频率、访问效率等数据，拟利用SQL操作解析工具JDBCParser对应用程序中的数据库操作进行全面的监控与记录。JDBCParser通过解析和分析通过JDBC接口执行的SQL查询语句，将其解析为抽象语法树（AST），从而详细地展示查询的各个组成部分。通过监控和深入分析数据库操作，可精确获取表单间的依赖关系。针对不同情况，我们可以有针对性的进行如下处理。 例如，当同一个SQL查询语句中同时出现的两个表单，或当多个业务场景下的多个数据库表单同时出现时，通常意味着这些表单间具有一定程度的的依赖关系；再例如，当多个微服务同时操作一个数据库表单时，可以对相应的语句和功能进行合并。我们也将数据库表单和服务的调用关系也进行保存，为后续微服务的划分提供帮助。

**服务间通信监控：**为监控服务间的通信请求量、延迟以及错误等信息，拟使用Kong和Ambassador等API网关工具对HTTP请求进行监控。Kong集成了Prometheus插件，可导出HTTP请求的详细指标，包括延迟、错误率、吞吐量等，同时支持将请求和相应日志输出，以便于收集相关数据；Ambassador则在Kubernetes环境中提供了强大的服务间通信监控功能，依赖Envoy Proxy生成服务间通信的请求计数、延迟分布、错误率和服务间通信的流量等指标，并使用Prometheus进行收集。通过对服务间通信数据的分析，可掌握拆分后微服务间的依赖关系和耦合程度，可将通信请求量较高的微服务进行合并，也能对最终得到的微服务架构的评估提供帮助。

**日志收集：**利用ELK Stack强大的日志收集、处理和分析功能，对监控工具所产生的数据和系统本身产生的日志信息进行收集和处理。ELK Stack包括Elasticsearch、Logstash和Kibana，其中Logstash支持输入、处理和输出的插件集成，能够从不同的来源收集数据，使用自定义插件对数据进行转换、过滤和格式化，并将处理后的数据传送到Elasticsearch等目标中；Elasticsearch则提供强大的搜索和分析功能，并且由于其分布式架构，能够处理大规模的数据并提供快速的数据查询，可以为后续数据的读取提供帮助；Kibana是一个可视化工具，提供数据的实时可视化和分析，便于查看。通过对系统运行时产生的日志和监控工具得到的数据的收集和处理，使得后续操作能够方便地从ELK Stack中提取所需数据，方便下一步对其进一步的分析和建模。

此外，在工业界环境下我们将研发微服务或函数代码的自动化提取工具，集成上述关键技术与方法，在此基础上基于日志流量回放、Trace染色技术，识别需要迁移的代码和可不迁移的代码。

**（2）依赖网络图的构建**

我们将进一步对收集到的运行时行为数据进行分析，从而获取结构化和指标化的依赖关系。对调用链与请求、数据库操作、服务间通信等数据和日志数据进行详细处理，提取出函数间、表单间和模块间的依赖关系以及后续作为划分依据的指标，在图数据库中进行保存。

**图数据库结构设计**：图数据库专门用于存储和处理图结构数据，以节点和边的形式来表示数据之间的关系，构建相应的图结构，方便后续进行关系分析和数据查询，还可使用图遍历、最短路径等算法对图数据库进行数据挖掘和分析。图数据库中的节点是指图结构中的实体元素，一般使用有唯一标示的标识符来代表，用于准确区分不同的实体对象。在微服务拆分场景下，实体对象可以是系统中的各种元素，因此我们将函数、数据库表单等作为图数据库中的节点，以便清晰地表示系统中的各个组成部分，为后续分析它们之间的关系奠定基础。图数据库中的边是指连接节点的关系纽带，具有描述节点间关系特征的各种数据信息，即边的属性，这些属性能够量化节点间关系的强度、方向和其他特性，帮助我们深入理解节点之间的关联情况。

我们将函数和表单间的关系建模为图数据库的边，使用有向边来表示依赖关系的方向，并使用收集到的调用频率、调用时间、不同业务场景下的权重等指标作为相应的属性，如图5所示，依赖网络的构建和指标的处理包括以下几部分：

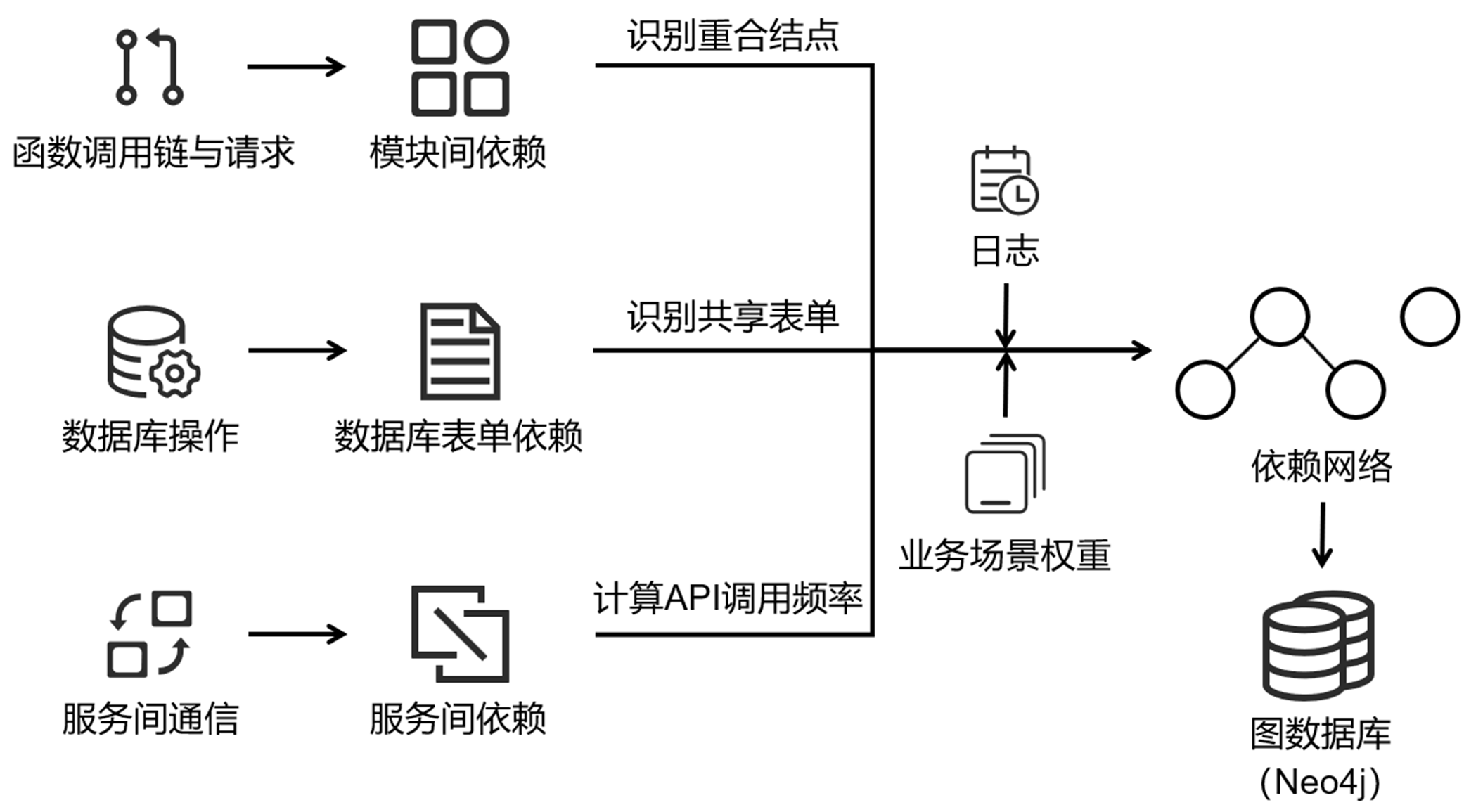


图5 依赖网络构建

**调用链与请求数据处理：**在使用Kieker工具监控不同测试样例下的函数调用链后，通过分析输出的日志文件，可以得到函数间的调用关系以及函数的执行时间，将其中的函数建模为图网络中的节点，使用函数的全限定名作为函数节点的唯一标识，包含其包路径、类名、方法名等，从而区分不同函数，并且能够在后续的微服务拆分和函数提取中通过全限定名直接定位函数，将函数的单次平均执行时间作为函数节点的属性。对于函数间的依赖关系，建立对应节点之间带有权重的有向边，边的方向表示调用方向，同时为对应的边添加权重等属性，包括调用频率、调用时间等，在后续的数据处理中补充。将监控时对不同测试样例添加的权重也作为属性添加到边上。

**数据库操作数据处理：**在利用JDBCParser工具对应用程序中的数据库操作进行监控并记录相关数据后，通过深入分析其解析得到的抽象语法树和数据库操作信息，可以得到数据库表单间的依赖关系以及数据库操作的相关指标。将数据库表单建模为图网络中的节点，使用表单的全限定名作为表单节点的唯一标识，包括其数据库名、表名等，从而区分不同的表单，并在后续微服务拆分和数据库表单的拆分和合并中通过全限定名直接定位表单。根据函数对数据库表单的操作关系，建立对应节点之间的有向边，边的方向表示依赖关系的方向，与此同时，为了区别函数对数据库表单的增删改查操作，我们将相应的操作种类和次数作为属性添加到对应边上；根据数据库表单间的依赖关系建立对应节点之间的有向边或无向边，其中无向边表示表单间的联合关系，从表单A到表单B的有向边表示表单A对表单B的查询等关系。对于所有指向数据库表单的边，根据所得数据添加操作频率、获取数据量等属性，并记录不同测试样例下相应边所对应的权重。

**服务间通信数据处理：**在运用Kong和Ambassador等API网关工具对服务间的HTTP和Kubernetes请求实施监控，并成功收集到服务间通信的请求量、延迟、错误率、吞吐量等详细指标数据之后，通过分析实际系统的服务间通信表现，可以将相关指标根据服务间的函数调用关系作为属性绑定到对应函数节点之间的边上，以添加服务层面的参考数据，为后续分析环节中对于现有微服务拆分效果和系统表现的评价提供支持。此外，对于在同一个服务中的函数，在相应的服务间延迟等属性处赋予零值，以表示其在目前的微服务划分中的位置关系。对于服务间的通信关系，根据函数调用关系挖掘深层次的调用关系，并将相应的数据赋予对应的边。例如服务A通过函数a调用了服务B中的函数b，则建立由函数a节点指向函数b节点的边，并将服务A和服务B之间通过函数a调用函数b的调用时间、数据量、错误率等数据作为边的属性。对于其他具有调用关系的函数节点，将其调用时间和错误率属性定义为0，来表示其存在于同一个微服务当中。

综上所述，所构建的依赖关系图在图数据库中保存和维护，函数和数据库表单作为图节点，分别以其全限定名作为唯一标识，函数具有其执行时间属性；图的边为函数及表单之间的依赖关系，其中使用有向图的方向来表示依赖的方向、无向边表示表单的关联关系；边的属性为运行时行为监控得到的数据，包括调用频率、数据量、调用时间、业务场景权重等。在后续操作中，只需要对所得出的图网络进行分析和拆分即可。

图6给出了上述实施方案的一个具体例子。

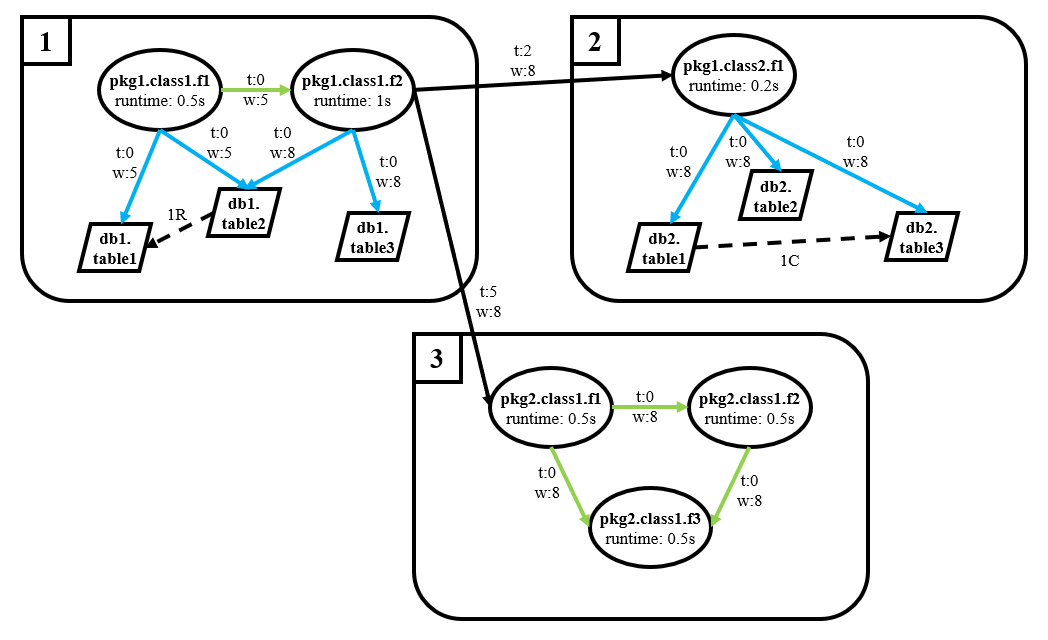


图6 依赖关系网络图示例

**（3）基于强化学习的微服务拆分策略选择**

为了更好地分析系统模块间的依赖关系，得到合适的微服务划分策略，在通过对系统运行时行为进行建模得到依赖网络图后，拟使用强化学习方法对依赖网络图进行图划分，实现自动化的微服务划分。

强化学习是一种让智能体通过与环境的互动，不断调整行为策略，以最大化累积奖励的学习方法。在强化学习中，智能体在特定的环境中进行决策，选择动作并获得反馈。智能体的每个动作会影响环境的状态，并带来一个奖励或惩罚，以指导智能体调整未来的行为。强化学习的核心思想是通过试错法进行学习，在不断探索和试验中发现哪些动作能够带来更高的回报，进而形成一个能最大化长期奖励的策略。强化学习的优势在于其自适应性和灵活性，尤其是在面对复杂的系统环境时。与传统的枚举法或图划分算法不同，强化学习能够根据实时的反馈信息自主选择最适合当前环境的划分策略，而不依赖于固定的规则或假设。通过不断探索和试错，强化学习能发现不同场景下的最优划分策略，避免了传统方法的高计算成本和对场景的局限性。因此，强化学习更适合微服务划分这一任务，尤其是在依赖关系复杂、需求多样化的实际系统中，能够提供更加灵活和高效的自动化划分方案。

特别地，强化学习主要包括智能体、环境、状态空间、动作空间和奖励函数等关键部分。其中，智能体负责与环境交互并做出决策，通过选择动作来影响环境的状态，通过环境根据其动作做出的反馈来调整行为策略，其选择和调整算法是智能体的核心；环境是智能体所在的外部系统，接收智能体的动作，并根据这些动作更新状态，同时返回奖励信号，以影响智能体后续的决策；状态空间是系统中所有可能状态的集合，表示系统在某一时刻的所有可能配置；动作空间定义了智能体在每个状态下可以选择的所有可能动作；而奖励函数是评估智能体选择的动作质量的标准，反映了智能体在执行某个动作后所带来的效益。如图 7所示，基于强化学习的微服务拆分策略设计如下：

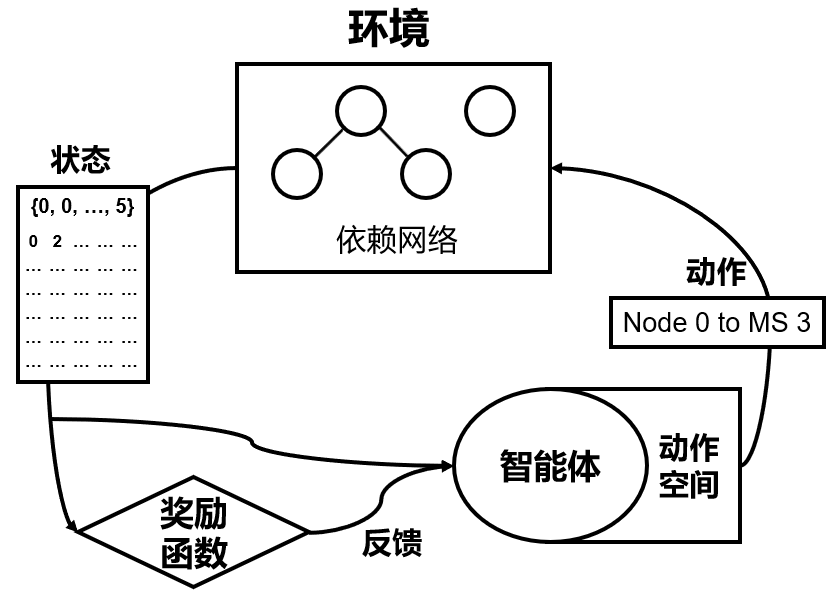


图 7基于强化学习的划分

**环境：**在微服务划分任务中，环境就是整个系统以及建模出的依赖网络图，在智能体对依赖网络图执行划分动作后，环境会做出相应的改变，并返回给智能体一定的评估。例如，当智能体将某个节点从一个微服务划分到另一个微服务时，环境会更新依赖网络图的结构，包括节点的归属、节点间的连接关系等。同时，环境会根据这种变化评估新的状态，依据奖励函数向智能体返回一个奖励信号，以此引导智能体后续的决策。

**状态空间：**当前任务的状态空间表示依赖网络图在不同时刻的各种可能配置，包括节点的划分状态、节点间的依赖关系和微服务相关的指标。其中，节点的划分状态即每个节点所属的微服务标识，是状态空间的核心部分，可以使用一个长度等于节点个数的向量来表示，其中每个元素表示对应节点所属的微服务编号。节点间的依赖关系是依赖网络图中的重要信息，可以使用邻接矩阵等方式来表示节点间的依赖关系以及依赖的强度，邻接矩阵的每个元素可以表示两个节点之间的调用频率等依赖信息。微服务相关的指标，如当前每个微服务的资源使用情况或功能复杂度等，通过对当前微服务划分策略进行模拟或计算得出，可以帮助智能体更全面地了解当前划分状态下各微服务的特性，从而做出更合理的决策。

**动作空间：**当前任务下，由于最终目标是对系统进行微服务的划分，而采取的粒度是函数和表单级别的，因此主要操作即是将函数和表单向某个微服务添加、从某个微服务中删除、或从某个微服务移动到另一个微服务。考虑到实际场景中，从单一系统划分出的微服务一般涉及系统中的多个函数或表单，拟将初始状态空间定义为所有函数和表单节点分别属于一个不同的微服务，从而使得智能体所需要做的动作只有将某一个节点从某一个微服务中移动到另一个微服务中，如果移动后微服务中不再包括节点，则将微服务销毁，同时保留一个空的微服务以使得节点可以被移动到空的微服务中形成独立的微服务。

**奖励函数：**奖励函数是评估智能体选择的动作质量的关键，也是评估微服务划分策略好坏的关键，应当紧密围绕着微服务划分的目标来设计，以反映智能体在执行某个动作后所带来的收益。在当前任务中，将某个函数或表单从某个微服务移动到另一个微服务，对于整体系统可能的影响包括跨服务的通信成本、微服务的内聚性、微服务的功能复杂度、弹性伸缩时资源的浪费、访问数据库表单时的数据一致性等，根据已经收集到的系统静态和运行时数据，设计奖励函数对所产生的影响进行建模。具体包括：

1. **通信时间**：考虑到跨服务函数调用所带来的通信时长增加问题，以跨服务的函数调用和数据访问频率估算系统总通信时间*T*，*T*可通过加和跨服务函数调用频率和跨服务的数据访问频率计算，即，其中*fc*是跨服务函数调用频率，*fa*是跨服务数据访问频率，*c­1*和*c2*是参数。
2. **内聚度：**考虑将单一系统带来的负载升高时的资源占用问题，以及微服务的高内聚性等特征，将每个微服务所包含的测试用例数作为微服务独立性的考量，总样例数记为*N*，即，其中*Ni*是第i个微服务的测试用例数。
3. **一致性：**考虑到跨服务访问数据库时可能带来的一致性问题，对多个微服务访问同一个数据库的情况加以惩罚，记为*P*，，其中*ai*是访问第i个表单的微服务数，*c3*是参数。

综上，总的奖励函数对上述三个参数进行加权计算，并且减去上一状态的计算结果，第x个状态的得分记为*Sx*，，其中*c­4*、*c5*和*c­6*是参数。奖励记为*Rx*，。

**强化学习算法设计：**智能体的策略选择和根据奖励函数调整策略的方法是强化学习的核心，强化学习算法一般按照基于价值或基于策略进行分类，基于策略的算法主要适用于在连续动作空间中进行策略优化，而在微服务划分任务中，动作空间只包括将节点在微服务之间移动，是离散的，因此采用基于价值的强化学习算法算法。基于价值的算法以Q-learning为典型代表，它旨在估计每个状态-动作对的Q值，即智能体在某个状态下采取某个动作后能获得的未来积累奖励的期望。当状态空间或动作空间规模变大时，使用表格来存储和更新Q值变得不切实际，因此拟选择结合了深度学习与Q-learning的深度Q算法（DQN）作为本任务的算法之一。DQN使用深度神经网络来近似Q值函数，即使用神经网络来估计每个状态-动作对的Q值。网络的输入时环境的状态，输出是该状态下每个可能动作对应的Q值。通过对DQN输出的Q值进行选择，大幅降低使用Q-learning时带来的时间和空间损耗。

通过使用强化学习对依赖网络图进行微服务划分，得到实际应用场景下综合考虑通信、弹性等系统特征时最合适的微服务划分策略。此时的划分策略以独立标号的节点集合的方式呈现，而微服务内和微服务间的函数与表单之间关系仍使用图的形式呈现。在划分策略可视化的基础上，实际划分出可运行的微服务系统是本项目的核心目标，因此还需要进一步根据微服务划分的结果对源代码进行处理，得到可运行的微服务系统。

**（4）代码大模型辅助的微服务代码拆分和部署**

为了将原有系统自动化地拆分成可以运行的微服务系统，在得到合适的微服务划分策略后，拟使用代码大模型根据划分策略对云原生应用的源代码进行拆分、重构和部署，以实现自动化的微服务化架构重构。人为的源代码拆分往往费时费力，在修改函数调用方式、删除冗余代码等任务时需要进行大量的重复工作，即使使用脚本进行也会有大量的特殊情况需要考虑，而代码大模型是专门为变成任务设计的深度学习模型，旨在理解和生成源代码，帮助开发者进行代码生成、代码补全、错误修复、代码重构等任务，针对微服务拆分任务，在进行冗余代码的删除、函数调用的重写、部署脚本的编写等任务时，可以利用已有内容快速生成和修改代码，大大减少人力投入，减少了错误的可能。

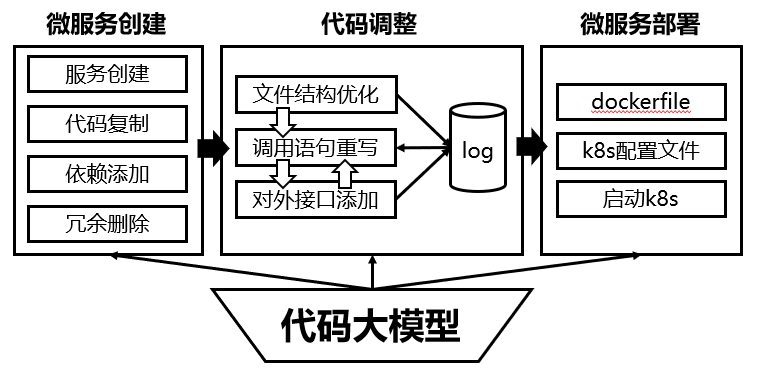


图8 自动代码拆分和部署流程

如图8所示，具体设计方案思路如下：

**微服务创建：**根据微服务拆分的结果将相应的函数、表单和配置文件等内容添加到对应的微服务当中。首先，使用大模型分析函数名和微服务涉及到的测试用例名，或根据人为设定的微服务名创建新的微服务项目，并按照原有文件结构将源代码中的对应函数和表单添加到微服务项目目录中。其次，分析微服务中代码的依赖关系，将所需但当前不在微服务中的配置文件等内容也添加到微服务项目中，并利用代码大模型对现有代码中的冗余代码进行删除。此时，得到目标微服务代码所需的所有函数、表单和配置文件，原系统中同一微服务中的函数若此时还在同一微服务中，则其调用方式应当是正确的，这一步是为了保留原有的文件结构，方便后续调整配置等内容。

**代码调整：**借助代码大模型，对微服务中目录结构和调用代码等进行调整。首先，利用代码大模型对微服务项目的文件结构进行优化和重构，并将重构后函数和表单的位置保存到日志中，与依赖网络图中节点的全限定名相对应，以便后续重构后找到相应函数或表单。其次，对于微服务内函数间和对表单的调用关系，借助代码大模型修改函数中相应的调用语句，根据节点在微服务当中的新位置进行重写，以确保微服务内的正确通信，同时修改依赖和导入包相关语句，使得微服务内的依赖逻辑合理。再次，根据依赖网络图，对于需要向外部开放接口的函数，借助代码大模型自动添加RESTful API接口代码以及相应url配置文件，向其他微服务开放接口，并将相应端口保存到日志文件中。最后，根据微服务间的函数调用关系，将微服务中调用外部服务的调用语句调整为通过HTTP访问其他微服务的开放接口，以确保微服务间正确通信。

**微服务部署：**为实现重构后微服务的自动部署，拟使用Docker和Kubernetes对微服务进行容器化并实现自动化的容器部署和服务发现。首先，借助代码大模型编写Dockerfile，使得重构后的单个微服务项目能够成功运行，此时由于代码大模型输出可能存在的不稳定性，可以选择使用固定的Dockerfile对项目进行镜像构建，在保证依赖关系和配置文件正确的前提下，微服务项目应当能够自动构建镜像。其次，编写Kubernetes配置文件：Deployment配置文件定义微服务的部署、更新、扩容等操作，指定微服务镜像、开放端口、配置环境变量，并根据依赖网络图中业务用例的权重配置初始副本数量，同时配置一定的弹性，使得微服务在负载变化的情况下自动进行扩缩容；Service配置文件暴露微服务的网络访问点，使得集群内部其他服务能够访问；Ingress配置文件配置外部访问微服务的入口，通过定义路由规则可以将请求转发到集群内的不同微服务。最后，启动Kubernetes，完成微服务的自动部署。