**基于多Agent的动态调度系统的设计与研究**

**1 研究背景和意义**

生产调度系统是当今制造系统的核心部分，调度的合理性和高效性直接影响整个系统的性能和质量，因此一直是研究人员关注的热点。在传统调度理论中，调度问题往往被认为是组合优化问题，但是随着问题规模的扩展，如生产设备和数控设备数量的急剧增长以及生产任务的频繁更改，导致组合结果呈指数级增长，即出现非常棘手的组合爆炸问题。随着对模拟退火算法、进化算法、禁忌搜索算法、神经网络算法等开展的广泛研究，企业在生产调度上取得了丰硕的成果。按理说经典调度理论已经比较成熟了，但是在实际应用中效果并不理想，原因之一是这些调度方法模型较为简化、方法单一，因此适用范围很窄，而更为关键的原因在于制造业环境比起以往任何时期都要显得动态多变。制造业环境发生的变化简单陈列如下表。

|  |  |  |
| --- | --- | --- |
|  | 传统制造业 | 现代制造业 |
| 消费者需求 | 满足基本需求、物美价廉 | 个性化、多样化、自我定制 |
| 生产需求 | 低成本、高质量 | 快速及时交货、满足用户多样化需求 |
| 市场环境 | 供求环境相对稳定 | 需求快速多变、难以预测 |
| 资源分布 | 集中 | 全球分布 |
| 控制和管理方式 | 集中控制式 | 分布式 |

由于科学技术的深刻发展和人们生活水平的极大提高，导致制造业必须迫使自身在生产和管理上作出深刻的变化。消费者对产品个性化、多样化的需求使得以往相对稳定的传统市场更加动态多变，以往的大批量生产逐渐向大批量用户定制以及多品种、小批量生产模式转变。同时，目前基于订单的生产成为主流，如航天、汽车、电子电器和机械制造业，产品种类繁多，每种产品的批量较小，多的会有十几件到上千件，少的产品订单可能只有几件甚至一件。形势更为严峻的是企业面临订单更改频繁的局面，例如从确定调度方案并下发到车间进行生产时，经常会发生新增紧急订单、撤销订单、订单优先级改变等问题，这给企业在业务管理和调度规划上造成了更大的压力。

此外，很多企业拥有大量机床和数控设备，这些设备在型号、品牌和种类上纷繁复杂且新老交替，因此在性能和效率上显得参差不齐，在高强度的生产流程中难免会发生各式各样的机械故障，加上同样难以避免的的原料短缺和质量问题，往往需要进行重调度优化，在这个过程中如果企业对诸如此类的突发状况反应不及时，会对生产流程的效率和质量造成非常大的影响。

还需要详细考虑的环境变化因素还包括当今飞速发展的互联网络，高速的带宽和良好的通信质量保证使得现代制造业逐渐抛弃传统的集中式控制结构，而是越来越倾向于使用分布式制造系统对生产流程进行管理和调度，导致技术人员、设备、工艺和文档资源等离散地分布在企业的各个生产场所，系统各部分需要高效互联，相互协调才能有效整合资源，提高生产效率。

面对生产过程中多样化和个性化的用户需求、灵活多变的订单生产、随时可能发生的重调度优化以及分布式的企业制造环境，基于高级的数学和算法理论的经典调度方法以及不足以胜任当今灵活多变的市场需求和充满随机性、不可预知的制造环境。经典调度理论倾向于使用严格的数学表达和以及对工艺知识进行形式化、量化地表达，针对单个或多个目标约束，基于知识库和推理规则进行优化计算，具有一定的自适应性、容错能力和并行处理能力，但是当前的制造业的系统平台正朝着服务化、网络化和分布式的方向发展，因此经典调度理论方法模型在这种情况下显得过于简单，方法应用也过于单一，因此这些理论的应用范围很窄，目前更多地只是停留在建模和试验阶段。

生产调度是企业制造的核心难题，随着各研究人员对该问题源源不断的研究，先后提出了准时生产、敏捷生产等概念，为的就是能够快速响应市场和制造环境的多变性，满足客户定制化、小批量、多品种、快速交货的生产要求。因此要解决现代制造中复杂的调度问题，仅仅依靠研究调度算法是不足够的，还需要从系统的角度出发，研究理想的调度系统模型以及合理的调度系统结构，这对系统的鲁棒性、柔性和快速响应能力提出了新要求。随着人工智能的发展，多Agent系统被引入到了调度领域，该系统在不同任务的协同处理上具有强大的并行能力，同时具有良好的分布性、适应性和鲁棒性，目前已经引起了学术界广大研究人员的关注热情，常常与经典调度理论相结合，对那些短期的、对敏捷性要求较高的动态调度问题具有高效的求解能力。尽管多Agent系统的研究开展时长已久，但是目前在制造系统上仍然还是停留在试验阶段，真正投入企业使用的产品还很少，因此我们的项目主要针对离散制造业中小批量多品种的客户定制生产，集成多Agent系统和调度策略等先进的技术和理论，根据客户的订单进行设计、生产和装配。

**2 多Agent动态调度系统模型的分析和设计**

现代制造业中每个工件由于拥有不同的加工特征，而且往往能在不同的机床设备上进行加工，加上受工艺约束的工序排序，使得各工件具有多条可选的工艺路线，我们研究的问题可以描述为：将一组任务安排到一组不同的设备资源上，每个任务包含一系列工序，这些工序的执行顺序有着严格的约束条件，不可以随意被打乱，每种工序都对应一组资源，每种资源在任一时刻只能对一道工序进行加工，而我们的目的就在于为该组任务匹配一组设备资源从而最小化任务的最大完成时间，当完成时间能满足任务的交货期时，说明这次调度是满足要求的。

在调度问题上，传统的方法往往采用集中控制方式，由单个计算单元负责绝大部分的计算任务，并把机床设备、订单任务看做简单的符号表示。这种方式通常要求计算单元有着优越的计算性能和负载能力，价格昂贵，而且适应性差，一旦计算单元故障将导致制造系统的全面瘫痪。同时，系统简单的模型构建难以及时对各种动态变化如设备故障灯中断作出快速反应。我们在调度系统中引入多Agent技术，将任务和资源映射为具有主动性、智能性的实体，具有更好的分布性、敏捷性和鲁棒性。此外，对于生产过程中出现的以外情况，Agent可以动态的进行反应和处理。

多Agent系统通过若干自主的Agent的协作来共同完成特定的任务，它可以将较复杂的问题分解成简单的任务。Agent可以接受动态和不确定的知识，具有一定的智能性，能够克服单个Agent知识不完全的局限性。多个Agent之间的相互协作、异步和并行处理处理机制非常适用于分布式架构。

**2.1 系统架构的选择**



图2.1

目前广泛使用的多Agent系统的体系架构主要分为层次型、异构型和混合型。层次型为Agent设立了多个控制等级和主从关系，高级控制实体具有全局观念，对低级控制实体具有管理和监控功能，这种架构能有效减少Agent之间的通信量，但是由于信号需要层层传递和反馈，对外部的干扰信息的反应比较迟钝。异构型认为Agent之间是平等的关系，他们通过相互协作作出自己的决策，这种架构具有更好的分布性和鲁棒性，但是由于Agent之间的自主行为会导致协商更加困难。而混合型是综合以上两种架构的优点，在子系统中使用层次型提高全局控制性能，同时整体使用异构型来鼓励各个模块之间的协商。如图2.1所示，我们的系统架构正是使用了混合型，资源子系统中使用了层次型，重点提高动态调度过程中的资源Agent的相互联系和调度效率，整体使用异构型加强系统对制造环境和市场变化因素的反应速度，提高系统柔性和抗干扰能力。

**2.2 Agent的分类**

如图2.1所示，根据现代制造系统的问题规模以及任务特点，我们定义了五类Agent作为基于多Agent的动态调度系统的基本组成构件，包括管理Agent、资源Agent、工艺Agent、算法Agent和监控Agent。每个Agent都可以表示具有一定智能和自治性的物理实体或者逻辑实体，例如资源Agent表示生产设备，工艺Agent表示产品和工件信息，算法Agent表示调度逻辑等等。下面我们分别对五类Agent进行介绍。

**2.2.1 管理Agent**

管理Agent相当于一个虚拟的车间管理员。主要负责生产订单任务的输入和分配以及对整个多Agent系统运行状况的监控和日志文件的记录。当系统启动后，会自动创建并启动管理Agent的主线程，主线程的工作主要包括：

·开启Agent的注册/注销监听端口，其他Agent只有将自身的信息登记在管理Agent中才能真正算是系统的正式一员，登记的信息包括标识符（IP地址，Socket监听端口号）、类别（资源Agent、工艺Agent等）、名称等；

·初始化Socket通信线程池，每当有其他Agent发起连接请求，管理Agent会从线程池中取出一个空闲线程处理请求。线程池中线程数目可根据计算单元的计算能力和性能进行设定，避免超出计算负载。

·初始化任务队列，任务队列负责接收客户的订单任务，客户可以在网页版或者桌面客户端选择需要的产品类型，订单任务也可以由系统管理员直接指派，之后会生成订单的XML表达形式，XML中<Product>标签中包含产品的的总体信息，如产品的名称、交货日期、工件组成（使用<Component>标签表示）；工件（Component）标签包含几何形状、制造信息等属性，如几何形状包括各种类型的平面、孔、槽等，制作信息包括材料信息、尺寸精度、表面粗糙度等。任务队列每接收到一个XML文件后都会对其进行解析，生成对应的Component类对象，最终封装为Product类对象。产品的XML表示大致如下：

<?xml version="1.0" encoding="utf-8"?>

<procuct

schedule:id=... <!--产品ID-->

schedule:name=... <!--产品名称-->

schedule:deadline=... <!--产品交货期-->

...>

<component

schedule:name=...

schedule:shape=... <!--工件形状-->

schedule:material=... <!--材料-->

schedule:heatTreatment=... <!--热处理方式-->

/>

<component

...

/>

...

<product/>

·管理Agent的工作还包括与监控系统交互获取生产线上各设备的工作状态，从而将信息返回到资源Agent进行模拟仿真；与工艺Agent交互更新自身数据库中与产品工艺相关的信息等。

**2.2.2 资源Agent**

资源Agent表征制造系统的生产设备。多个资源Agent组成了调度系统的资源子系统。资源Agent分三类，分别是系统Agent，车间Agent和设备Agent。系统Agent代表整个调度系统的资源集合，负责统筹兼顾其下的各个车间Agent；车间Agent表示企业的一个车间，负责管理车间内的若干设备Agent；设备Agent表示一台机床设备。资源Agent采用层次型结构，上层Agent对下层Agent具有控制和管理的权限，因此需要保存下层Agent的通信标识符。当系统引入新设备或者旧设备淘汰时，资源Agent也会相应进行的扩展或者删除。设备Agent中需要对相应的设备工作状况进行模拟，记录分配到该设备的工序集合，按照工序的开始时间进行排列。

资源子系统的目的是从管理Agent的任务队列中获取优先级最高的任务，然后搜索一组资源设备能够最大程度地满足产品订单的交货期要求。我们为资源子系统提出一种基于合同网的迭代竞价内嵌于整体竞价的协商策略，该策略将在下一章节进行描述。在资源子系统中的基于合同网的协商过程可以简单描述如下：

1．订单任务通知处理：一旦接收到上层资源Agent传达的任务通知，资源Agent通过检查任务的详细说明来判断是否能满足要求。订单任务可能是Product对象或者是Component对象，后者可看作为是零件级任务。Product对象中封装了产品的详细信息，包括产品名称、产品ID、交货期限以及产品的工件组成；而Component表示一个工件对象，里面封装了工件的相关属性，同样包括工件的名称、ID、完工期限等信息，以及尺寸精度、表面粗糙度、材料、加工方式等要求。资源Agent根据订单任务的信息以及自身的能力进行匹配，如果成功匹配，那么资源Agent将该任务信息进行保存，接着进行任务的投标，表述中会注明对订单任务的安排方案，同时给出自身对完成该订单任务的代价，即虚拟价格。虚拟价格和完成时间高度相关。

2. 标书处理：每次竞标都有时间限制，时间截止后，上层Agent把从下层Agent中的投标者投递的标书进行收集，根据虚拟价格从中选择出价最低的资源Agent并把任务授予该Agent。

3、授予处理：投标失败的Agent简单的删除与投标任务相关的信息即可，而投标成功的Agent，即任务承包者必须按照标书承诺的时间、精度要求等完成任务。任务完成时，承包者需要将任务的执行结果返回给上级Agent。

以上是每一轮迭代过程中基本的协商机制，经过多轮的迭代，每一轮都保存当前虚拟价格最低的方案，迭代结束后即可得到最优的方案。基于合同网的协商逻辑简单高效，已经成为分布式问题求解中实现最广泛的框架之一。

**2.2.3 工艺Agent**

工艺Agent表征制造系统所能生产的产品。一个产品由若干个工件组成，一个工件的成型需要经过若干道工序的加工。工艺Agent就负责记录各产品信息以及相应的工件加工流程图，同时提供数据咨询接口。

工艺Agent的数据库使用可扩展标记语言（XML）的形式来记录产品、工件和工序的参数，标签分别为<product>、<component>和<Operation>。相比上面提及的<product>和<component>的参数，工艺数据库除了不存储交货日期属性外基本相同，只是每个属性可能会存在多个可选值，例如尺寸精度、表面粗糙度、加工方式等属性会列出系统中所有设备所能满足的条件。为了减少数据的冗余，XML文件中使用<include>标签引用其他标签，如下XML文件所示。除了能够减少数据的重复外，还能够降低XML文件在大小上显得更为轻量级，提升传输速度。更重要的一点是能够满足不同Agent的数据需求。例如管理Agent更多地需要<product>标签的属性和信息，而资源Agent的需求更多体现在工件的属性和参数上。

<procuct

schedule:id=... <!--产品ID-->

schedule:name=... <!--产品名称-->

schedule:deadline=... <!--产品交货期-->

...>

<include

schedule:compinent=... <!—引用的Component的ID-->

/>

<component

...

/>

一个工序代表工件加工过程中的一个操作，例如孔加工、面加工、槽加工等。每种工序根据各自的定义拥有不同的描述信息，以孔为例，可分为圆柱孔、阶梯孔、螺纹孔等，在加工过程中有着自身独特的参数，例如阶梯孔的加工需要指定指定两孔直径，螺旋孔需要指定螺纹深度、螺距、钻孔深度等。由于各个工序在工件特征上差异很大，因此使用XML方式进行表达就不合适了，原因在于XML适合描述具有更多共性的对象。最终我们使用Java序列化和反序列的方式对工序信息进行存储。Java序列化是指把Java对象转换为字节序列的过程；而Java反序列化是指把字节序列恢复为Java对象的过程。当我们需要在两个进程进行远程通信时，可以相互发送各种类型的数据，包括文本、图片、音频、视频等，而这些数据都会以二进制序列的形式在网络上传送。使用Java序列化和反序列化方法我们同样可以在网络中传输Java对象，只需要我们在传输前使用Java序列化技术把对象转换为字节序列，而接收方使用Java反序列化技术把字节序列还原为Java对象

我们对每一种工序都使用对应Java类来表示，类中设置尺寸精度、表面粗糙度、材料等属性以及对应属性的操作方法（setter/getter）。当我们需要使用到某个工件的描述信息时，可以生成对应类的对象并在网络传输。使用Java序列化和反序列化技术，我们不仅能充分利用面向对象技术的继承、多态和封装的好处，这可以使我们在定义类属性的过程中就完成了消息格式的定义工作，而且在通信传输过程中避免对数据的解析，只需要我们使用setter/getter方法就能完成数据的读写，能有效提高解析效率。

2.2.4 算法Agent

算法Agent根据设备集合以及生产任务进行问题求解。资源Agent负责对生产任务进行分解以得出若干可行解，每个可行解对应一个设备集合，算法Agent根据集合内设备的工作状况以及生产任务得出最优的调度结果。算法Agent封装了陈昕叶师姐的基于改进蜂群算法的资源分配策略，在此不赘述。算法Agent除了需要与资源Agent交互外，还需要从工艺Agent初获取特定工件的工艺流程图，基于一组设备资源和若干工件的工艺流程图生成最优的调度结果。

**2.2.5 监控Agent**

监控Agent表征制造系统的监管人员，负责监控生产线信息的采集以及故障分析。监控Agent采集生产线各个工位的工作状态信息，根据信息分析工位是处于安全状态还是故障状态，并把相关信息反馈到管理Agent，由管理Agent判断调度方案的重构。

当调度系统启动后，初始化监控Agent的创建和启动线程。监控Agent成功在管理Agent初注册后，加载数据库的信息，获取电科院工作站服务进程的IP和端口号，发起Socket连接，采集服务器上的相关数据，数据的编码和解码使用Java的序列化和反序列技术，采集的数据形式包括以下几种：

1. 调度系统中的资源子系统结构，即系统中包含多少个车间、车间内有多少设备以及设备的工作状态。资源子系统使用SystemAgent类表示，其中的JobShops链表表示了系统中的车间集合；车间使用JobShopAgent类表示，其中封装了车间名称、地理位置等属性，machines链表代表了车间内有多少设备；设备资源使用MachineAgent类表示，其中封装了设备的名称、ID、能提供的功能（加工方式的种类、精度范围等）以及设备的工作状态chips。MachineAgent中的chips链表表示了当前安排在该设备上的工序集合，链表中每一个元素都表示一个工序，其中记录了工序的开始时间、结束时间等信息。服务器根据资源架构和生产线上各设备的工作状态生成对应的SystemAgent对象，使用Java序列化技术在网络上传输此对象。当监控Agent获取到Socket的输入字节流InputStream后，使用反序列技术还原出SystemAgent，调度系统根据该对象创建并初始化各个资源Agent，并形成完整的资源子系统结构。

2. 故障信息。监控Agent需要采集生产线各个工位和设备上的运作信息，从中分析出设备故障的类型和原因。在空调装配示范线包含安检工位、安装压缩机工位、抽真空工位、打包工位等。在工作站的数据库保存各个工位的即时数据和历史数据。例如抽真空工位在某个时间段的历史数据如下表。随着机床设备资源的数量增长会产生大量的历史数据，为加快数据传输速度、降低对带宽的压力，我们对数据库的局部数据提取出来形成若干个数据库.db文件，同时采用哈夫曼压缩编码，减小文件的大小。监控Agent获取到db文件后，将其填充到自身的数据库中，同时对数据进行分析，检测示范线上是否存在工位发生故障以及给出故障原因，最终把结果返回到管理Agent中。

|  |
| --- |
| 工作状态:0(0：停止中，1 工作中) |
| 故障:0(0：无故障，1 故障) |
| 抽空结果:0(0：合格，1 不合格) |
| 卡片写入信息:kdkdkdkd0021 |
| 时间:2016-07-15 09::05:22 |
| 条码:12345678912311 |
| 型号:PA1021 |
| 结果:0(0：合格，1 不合格) |

3. 定时获取各设备资源的工作状态。监控Agent在系统初始启动时需要连接工作站获取各设备的工作状态，为的是创建对应的资源子系统结构，对各设备的工作状态进行模拟仿真。随着时间的推进，模拟结果可能与实际产生误差，且时间越长，误差积累就会越严重，甚至可能发生模拟结果的加工速度远远超过或落后于实际的加工进度，这时候后续的调度和重调度结果就不再有意义，情况严重时，可能会导致下达的调度任务与现有调度进度产生冲突，导致生产线的混乱甚至瘫痪，因此非常有必要同步两者之间的时间差。监控Agent中开启了一个专门的计时器Timer，Timer中内嵌一个线程，线程根据设立的时钟间隔，每隔一定时间就会向工作站的服务进程发起Socket连接，从中获取SystemAgent对象，然后传输到系统的SystemAgent对象，对两个对象中的MachineAgent的工作状态进行对比，调整误差。

**3 调度系统的实现**

目前面向多Aent的系统开发尚处于初步阶段，开发方法并不成熟，这主要是由于Agent的开发语言ACL跟当前主流的程序设计语言不同，它还处于开发和推广的初级阶段，对API的设计还不够完善，因此开发人员在面向Agent开发中面临许多问题和困难，己出现的一些开发方法和平台缺乏系统性和标准化，面向Agent的开发工作存在一些随意性。从这一点来讲,多Agent更多的知识提供一个面向制造系统应用的功能模型，而如何构造基于多Agent的执行系统则是我们需要解决的一个关键问题。

对于Agent的开发语言通常有如下要求：（1）面向对象：Agent是一种对象，具有数据和方法。Agent之间的通信通常是通过对公共方法的调用。显然具有比对象更高的自治性；（2）平台独立性：便于主体在网络异构计算机环境下工作；（3）通信能力；（4）安全性：作为一种著名的适合于分布式环境的面向对象的编程语言，Java语言无疑可以满足以上要求。Java具有平台无关性，可以保证不加修改的运行在不同的软硬件平台上。网络编程是其最强大的功能之一，使用Java的序列化和反序列技术可以方便的实现对象之间的传输，免除消息格式的定制、数据的解析等工作。此外，Java的多线程机制允许我们以更小的系统运行代价实现多个Agent之间的并行性。最后，Java的安全机制可以为系统提供较高的可靠性。因此在我们的工作中，为了解决多Agent系统的实现问题，我们使用面向对象技术、Java语言和Socket通信机制来研究基于多Agent的动态调度原型系统的设计与实现问题。

**3.1 原型系统的架构**

基于多Agent的动态调度原型系统可以分为三大子系统以及两个模块，分别是管理子系统、资源子系统、监控子系统以及算法调度模块和工艺管理模块。系统各部分之间使用Socket通信、多线程的传输流的方法来解决Agent通讯和互操作的问题。系统架构如下图所示：



我们的原型系统是基于分布式的多Agent系统，目的是能够将该系统应用在分布式环境中，例如多机系统。在分布式环境中，每个Agent可以表征一个物理实体或者软件逻辑实体，例如可以表示一台设备，这就是我们系统中的资源Agent；也可以把Agent对应为一个过程，例如我们将调度算法封装到Agent中，形成我们的算法Agent。调度系统中包含多个Agent，它们在同一时间内完成不同的任务。在Java中，主要通过多线程机制来实现并行性。线程是一种轻量级的进程，它和进程都由操作系统负责调度。多个线程的执行的并发性是逻辑上的并发,而不是物理上的“同时”,但是用户并不能感觉到其中的区别。

下面说明一下系统中进程和线程的关系和特点。首先，进程是操作系统内资源分配的基本单位,它拥有独立的虚拟地址空间，而线程属于某一进程，它与进程内的其它线程一起共享进程的资源，包括虚拟的存储空间。其次，与进程相比线程更加轻量级，可以减少CPU的调度开销，便于系统管理。最后,显而易见，不同的进程可以具有分布性，而如果我们把全部Agnet对应为同一进程下的多个线程，那么系统将不再具有分布性。

整个系统中的计算量主要集中在资源Agent之间的协商过程、算法Agent的调度过程以及管理Agent与其它Agent的通信过程。虽然工艺Agent与其它类型的Agent之间都有直接交互，但是它在功能上更多地体现为一个数据接口，除此之外并无计算量。当前企业的资源倾向于分布在不同的地理位置，不同的部门或者车间可能相距甚远，因此资源Agent的分布也是具有全球地理性质。因此我们最终决定将SystmAgent、管理Agent、监控Agent作为三个独立进程，这三个进程倾向于共处一个操作系统中，减少通信开销。而每一个JobShopAgent是必须作为一个独立的进程，它根据对应车间的地理位置而分布在各处，JobShopAgent之下的各个MachineAgent作为进程中的多个线程，这是由于MachineAgent对应的设备必定是与所属车间处于同一地理位置的。这样，使用多进程配合多线程的机制能够合理分配系统资源的同时，也能实现系统的分布性。

**3.2 管理Agent和资源Agent的通信实现**

管理Agent是整个原型系统的核心，负责管理、监控和协调其他Agent的工作。管理Agent还负责接收外界的订单任务，订单任务可根据用户的需求生成特定格式的消息或者数据包，经HTTP/POST方式传送到管理Agent，也可以由管理员直接下达订单任务。管理Agent中有特定的线程专门用于接收外界的订单任务，同时还专门开辟一块特定大小的缓冲区用于接收订单任务的数据包，这块缓冲区就是任务队列。根据管理Agent所在的计算机的请求并发处理能力的大小，我们可以为缓冲区设置不同的大小，这是为了避免当用户订单过多造成管理Agent负载过重，严重影响原型系统整体的运行效率。如计算机的处理速度较快和内存容量较大，即能快速处理多个请求，我们可以将缓冲区设置稍得大些。当任务队列不为空，资源Agent从中逐个取出任务并进行处理；当任务队列为空时，资源Agent的线程处于休眠状态直至有新的任务到达任务队列，如下图所示：



我们为任务队列设置两个条件变量Condition，分别为notFull和notEmpty。当任务队列为空时，资源Agent的任务获取线程使用notEmpty.await阻塞从而陷入睡眠。当管理Agent的任务接收线程获取到任务订单并投放到任务队列时，使用notEmpty.signal唤醒沉睡的资源Agent线程，通知该线程获取并处理任务；同理，当任务队列满时，管理Agent的任务接收线程使用notFull.await阻塞从而陷入睡眠。当资源Agent从任务队列取出任务后，使用notFull.signal唤醒任务接收线程，通知该线程把新任务投放到任务队列中。使用线程间的同步机制能保证资源Agent和管理Agent之间的通信的有序和高效，当缓冲区为空或或满时，对应的线程陷入睡眠释放处理器，减少系统资源的负载。

正如前几章所提及的，管理Agent主要接收三种类型的数据：

· 订单任务，该任务以XML文件的形式传送到管理Agent，该订单任务可由客户在客户端或者Web端输入，之后管理Agent根据XML中的产品信息向工艺Agent获取各工件的描述信息，最终生成Product对象放在任务队列中。

· 监控Agent从生产线工作站数据库获取的SystemAgent对象，该对象与调度系统中的SystemAgent进程不属于同一概念，前者是数据结构，后者为进程，具有多个线程以及自己的数据信息。SystemAgent对象包含了构建资源子系统所需要的信息，例如有多少车间，车间内有多少设备以及设备的工作状态等等必要信息。

监控Agent从生产线工作站数据库获取的.db文件以上数据传输都基于Socket通信机制，数据使用Java的序列化和反序列化技术进行字节流传输。

**3.3 资源Agent间的通信实现**

多个资源Agent组成原型系统的资源子系统，该子系统使用层次型架构，上层主Agent对下层从Agent具有管理、指导和协调作用。在分布式环境中，上层Agent中记录有下层Agent的IP地址和端口号，同时每个Agent内部都开辟一个线程池，用于处理来自上层和下层Agent的Socket请求。线程池我们选用定长线程池，即限定线程的最大并发数，超出的线程会在队列中等待，这样可以避免当Socket请求过多而导致系统资源的使用超过负载，造成系统运行性能的急剧降低，如下图所示：



上层Agent把任务分解后，将子任务合同封装成Socket消息传递到下层Agent，下层Agent同样把任务进行分解后再发布到再下层子Agent，这样任务的分解和分配就层层向下传递，由下层Agent进行竞标和投标。传递的每一个消息都需要注明是合同亦或是标书，以辅助Agent进行辨明消息时来源于上层Agent还是下层Agent。为了提高并发程度，合理利用系统资源，对于每一个Socket请求，资源Agent都会从线程池中获取一个线程处理请求，这样上层Agent能够从容灵活地处理来自下层Agent的消息。

**3.4 管理Agent与监控Agent之间的通信实现**

管理Agent与监控Agent之间的通信同样是基于Socket的。监控Agent与电科院的工作站建立长久连接，持续地从工作站的数据库获取生产线各个工位的实时数据，把数据存进数据库，进行生产日志的记录，此外还需要对数据进行故障分析。若数据分析过程中发现设备故障，将把故障信息使用Socket通信的方式发送到管理Agent。监控Agent除了进行故障分析外，还需要将生产线各个设备的运行状况生成特定格式的数据，这是为了方便资源Agent模拟的设备运行状况与设备实际的运行状况进行同步，控制时间误差。因此我们在管理Agent中设置专门的定时器用于向监控Agent周期性地获取数据，并把返回数据与各资源Agent的数据进行对比，当两者的数据误差过大，需要重新设置时钟和工作进度。

其次，各Agent与工艺Agent和算法Agent的通信方式较为相似，在此不再赘述。

**3.5 与现有系统的衔接**

为了降低开发费用，节省开发成本，使实现的调度系统尽快投入使用，因此将桂刘冰师兄的生产线监控系统作为独立的监控Agent。监控系统原有的数据采集和故障分析功能保持不变，增加Socket通信与管理Agent相连接，同时将生产线的实时数据封装成特定的格式，以便于与资源Agent模拟的设备工作状态进行对比，控制两者之间的时间误差。

**3.5.1 MFC系统与Java系统的集成**

由于师兄的监控系统是使用MFC实现，而目前的多Agent调度原型系统是基于Java实现的，为了降低系统集成的难度，我们在监控系统中集成了多Agent系统，实现方法是Sun公司的JNI技术。

JNI即Java Native Interface 。从Java 1.1开始，Java Native Interface (JNI)标准成为java平台的一部分，它允许Java代码和其他语言写的代码进行交互。因此我们可以使用JNI在C/C++代码内调用Java类方法，要做到这一点**必须使用 Invocation API 在本机代码内创建和初始化一个 JVM。具体步骤通常分为以下几步：编写、编译Java代码，加载JVM动态库，创建嵌入式JVM，使用特定的API调用Java类的静态方法，或者生成Java的类对象，从而调用类方法。**

**实现过程需要添加JVM动态库的链接，**调用JNI\_CreateJavaVM函数创建JVM。该函数会传出一个JNI上下文环境对象（JNIEnv），利用该对象的相关函数就可以调用JAVA类的属性和方法了。

**3.5.2 监控Agent与工作站的通信**

监控Agent与工作站的服务进程均是进程，因此双方的通信本质是跨进程通信。这里我们使用Socket作为两者之间的通信方式。监控Agent在本地保存工作站的IP以及服务进程的端口号。服务进程指定自身的IP和端口号，且Socket套接字一直处于阻塞状态等待客户进程的连接。客户进程，即监控Agent打开自身的Socket套接字，根据服务进程的IP和端口号试图连接服务进程的Socket,服务进程接收到监控Agent的连接请求，于是被动打开自身的Socket，开始接收监控Agent的连接，双方连接成功，于是监控Agent开始向Socket写入信息，服务进程从Socket获取信息。

上面提到监控Agent主要从服务进程采集两类数据：设备的工作状态以及设备的即时数据，前者用于生产线的模拟仿真，后者用于故障分析。监控Agent向Socket中写入的数据是名为Request的类对象，使用Java序列化和反序列化进行数据的编码和解码。Request类有两个子类，分别是MachineRequest和FailureRequest，两类的基本属性如下图。监控Agent把Request对象传送到服务进程后，服务进程通过反序列化技术获取到Request对象，并解析对象，根据对象的性质从数据库中返回对应的数据给监控Agent。



**3.6 各类Agent的程序设计**

在第二章中，我们阐述了各个Agent的内部模型，其中的各个部分代表了特定Agent所具有的功能。例如管理Agent具有动态数据库、静态数据库、自学习模块、注册/注销模块、推理机、调度执行模块和通信管理模块。这里，我们主要给出管理Agent和资源Agent的程序设计，如其中的属性。方法、继承的基类和实现的接口等。

管理Agent的类设计如下所示：



资源Agent主要分为三类Agent：SystemAgent、JobShopAgent和MachineAgent，分别代表制造系统、车间以及设备资源。三类Agent的继承关系如下图所示。



下面主要列出SystemAgent的类设计。其中主要涉及任务的分解，即投标—竞标—中标三个过程。



这一章主要阐述如何使用面向对象技术实现基于多Agent的动态调度原型系统的系统架构。在面向对象的Java语言的支持下,定义了Agent的功能接口以及对系统内公共数据库等关键组件进行了对象化的封装,通过对象之间的消息传递,体现系统的社会性、智能型与合作能力。在这种框架下,使得多Agent车间调度系统的开放性、模块化、稳定性和易维护性等特性得以实现。