Optaplanner 学习笔记

背景

Geoffrey De Smet 的开发日志翻译:
https://www.oschina.net/news/75942/a-decade-of-optaplanner

十年前的一个假期，我开始开发一个小项目，没有太多意义，只是按照自己的兴趣爱好来，也没有打算做得很大很专业。在漫长开发中，我慢慢试图构建一些很酷的工具。今天，Java 规划引擎：

optaplanner，应用在很多项目当中，与同领域软件相比处于领先地位。OptaPlanner 优化了商业资源调度和规划。

2003 毕业后，我曾当了 2 年的 java 顾问，负责 Multi-Agent 系统(MAS)的研究。我的同事，都是一些想改变世界的人，后来证明，他们没能做到。当时，我的一个同事正在优化护士排班问题（nurse rostering problem），对我们的研究小组提出了他的方法。他的解决方案是采用 Tabu 算法（一个优化算法在 80 年代发明的）。他的工作进展得很顺利。然而，他花了很多时间和精力去手工实现增量计算每个约束适应度函数。写代码很困难，耗时长且容易出错，但都会取得好成绩。

一年前，我曾在 JavaPolis 大会听过 Mark Proctor 的演讲（现在叫 DevoxxBE），解释 Drools 规则引擎和 RETE 算法。一个易于访问企业策略、易于调整以及易于管理的开源业务规则引擎，符合业内标准，速度快、效率高。

我提议在规则引擎中使用 Tabu 搜索算法，他们嘲笑我，那时候我没有一点算法优化经验。

不能空空其谈,所以,2006 年暑假期间,我实现了一个简单课程调度问题的概念认证。我叫它 Taseree,这是一个简化 Tabu 搜索规则引擎的规范。同年晚些时候,我添加了两个例子(N Queens and TTP) ,并且开源到 SourceForge。

研究比赛

按理说，事情到这就应该结束了，就像我的许多其他项目一样。但是我听说过一些学术研究竞赛：给出一个优化问题,5 分钟内找到最好的解决方案。所以在 2007 我参加了这个竞赛,在 200 参赛者中,我第四个完成了。

所以我确定我是在做正确的事。但是我也做错了很多事情。我们开源了各自代码，我们开始讨论和比较我们的实现，我们彼此学到了很多。其实有很多方法可以实现 Tabu 搜索。所以我开始采用替代算法实现。促使我(和 Lukaš)写的基准工具包，比较这些算法统计，评估那个算法性能更好、更适合长足发展。理解这些算法需要一个小时，但 10 000 小时掌握它们，因此我花了很多时间。

动机是关键。找到一种方法来激励自己构建下一个特性，特别是在没有工资的情况下。爱好驱动我继续前进。

另一个优势是，每场比赛都会诞生一个新的例子。我最终做了很多框架改变对我来说更容易实现这些用例。这是一个重要的、旧的项目管理原则:那些创造了痛苦,应该感到疼痛。

没有人是一座孤岛

大约在同一时间我联系到 Drools 的团队，通过加入他们的 IRC 频道。我开始发表我的看法，同时，一边学习。2007 年 9 月，马克邀请我让 Taseree 成为 Drools 的一个子项目，我们称之为 Drools Solver。所以我清理代码，删除了 spring 核心依赖和写第一章的参考手册。它被作为 Drools 5.0 的一部分释放。

到 2007 年 12 月，在 JavaPolis 大会上，我给 20 开发人员第一次讲解了 Drools Solver。

在 2008 年，我听说第一个使用 Drools Solver 的产品。

毅力

也正是 2008 年，我开始有了较多的业余时间，我可以在每周花几个小时在 Drools Solver 上。然而，我和我妻子也买了一所旧房子，我必须得花时间在房子翻修的事情上，因此，项目开始停滞，几乎一个月只能更新一次。

在这期间，我萌生了好几次放弃该项目的想法。不知何故，我觉得我不能。因此我坚持了下来。最后，一切稳步推进。

具有讽刺意味的是，项目开始变得受欢迎，下载量逐渐变多，论坛也变得活跃，我收获越来越多的建议。也正是这样，越来越多的 Bug 也随之而来。所以，我投入大量的时间进行单元测试、集成测试。到了 2009 年底，项目改名为 Drools Planner。

幸运的是，Red Hat - Drools 项目发起人已经注意到我对项目源源不断的贡献。他们很早邀请我为 Drools 工作，但在 2013 年才分配我全权负责。Red Hat 开始支付工资给项目的贡献者，并且支持该项目进行业务销售、咨询。

产品化：走向企业

到了 2012 年底，Drools Planner 已经不在是一个小项目，变得很庞大。它包括具有较高覆盖率的单元测试、集成和压力测试，完整的参考手册，完备的典例，JavaDocs 和不断壮大的社区。当一家世界 500 强的企业考虑将它应用在重要的生产环境中时，我们发现它的服务是跟不上的。技术上的卓越还是远远不够的。

因此，Red Hat 建立了一个专门的 QA 团队，支持团队，顾问团队，保障团队和产品化建设/文档团队。这使客户能够在大规模生产的情况下，自信地部署我们的开源软件。2013，我们开始这个过程，称为产品化。

通过各个服务团队的加入，产品化日趋成熟。项目成为一个顶级项目，所以我们又重命名了一次，也是最后一次，叫 OptaPlanner。同时，也创立了 optaplanner.org 。

2014 年 3 月，在红帽的支持下，我们发布了第一个技术预览版本，作为 BRMS 订阅的一部分。2015 年 3 月，我们全面升级企业支持，销售量增长。今年早些时候，我们雇佣一些核心工程师开发

OptaPlanner 工作台。与此同时，我们所有的代码仍然是开源的，在 Apache 许可下，成千上万的项目正使用它。一个双赢的局面。

我们坚信，未来是光明的。

Optaplanner 是什么？

排程工具
http://www.optaplanner.org
它本来是一个名叫 Geoffrey De Smet 的大牛自己写的，后来他就把它贡献给了 JBoss 基金会(这里省去了 N 年的曲折离奇， N >= 10)，并成为 KIE 项目组中 OptaPlanner 项目的负责人。所以 OptaPlanner 是基于 Apache2.0 开源协议的，对商业友好，就是说你想用就尽管用，有问题还可以在他们的讨论组上求助

https://github.com/ge0ffrey

OptaPlanner 其实是一个很好的**排程引擎**（更贴切地说它是一个**规划引擎**，下面就称规划引擎吧，因为它不光用于排产上），耐何在国内使用的人十分少，所以中文资料几乎没有，国内有几个比较出名的 **APS** 产品，不知道其排程核心用的是什么，不过如果是自主开发 APS 系统的话，OptaPlanner 是一个好的引擎，毕竟并不是所有企业都能找到一堆数学专家对组合优化问题进行研究的

官方描述：
OptaPlanner 是一个**约束求解器**。它优化了企业资源计划的使用情况，如车辆调度、员工排班、云优化、任务分配、任务调度、Bin Packing 等等。每个组织都面临这样的调度难题：分配一组有限的受限资源（员工、资产、时间和金钱）来提供产品或服务。OptaPlanner 提供了更有效的计划，提高服务质量并降低成本。OptaPlanner 是一个轻量级的、可嵌入的**规划引擎**。它令普通的 java 程序员有效地解决优化问题。它还与其他 JVM 语言兼容（如 Kotlin 与 Scala）。约束适用普通的域对象,可以重用现有代码。没有必要把它们作为数学方程来输入。在引擎盖之下，OptaPlanner 结合先进的优化的启发式和共通启发式演算法（如禁忌搜索、模拟退火和延迟接受）,非常高效地进行分数计算。OptaPlanner 是开放源代码的软件，Apache 软件许可下发布。它是用 **100%的纯 java**，运行在任何 JVM 在 Maven 的中央存储库也可用。

它就是一个用来解一些规划问题的引擎，而规划问题几乎都可以被视作 NPC 问题

概括来说，就是一些没有办法使用确定性算法来得到结果的问题，而对于这类问题，又分为 NP 问题和 NPC 问题，但都只能通过遍历的办法才能找到。对于 NP 问题和 NPC 问题，我有以下理解，也不知道对不对，大牛看到不对的帮忙指正一下：

NP 问题：一种无法通过确定性算法直接获得解，**但对获得的解是可验证的**，例如：结合上一篇文章提到的生产排程问题，如果老板只要求做出一个可行的生产计划，也就是只需要一个可以执行的生产排程就可以了。成本、效率什么的都不管；那么这就是一个 NP 问题。因为要做出这个计划，你也是没有直接的、确定的方法或算法来做的；更多的是靠经验、对实际情况的有限掌握、对来情况的预判和感觉。但是做出来的计划是可以验证的。也就是说车间拿着这个计划是真的可能执行的，而不会出现物料不到位、产品分配到了错误的机台上等违反硬约束问题的， 那么只要不违反这些硬性约束，就认为这是一个可行的计划。所以做一个可行计划，可以被视作是 NP 问题。

NPC 问题：**则是那种不旦无法通过确定性算法获得解，对所得的解，也没有一个确定的办法去验证的问题**。还是上面的生产排程问题，如果老板要求做一个所有情况下除了可行，还要成本最低、效率最高的计划。那么：1. 计划员也只是靠经验、预判、对数据有有限掌握做出一个计划来，计划是否可行是可能验证的(也就是 NP 问题)，但这个计划是否成本最低、效率最高，那就没办法验证了，除非你把所有可能的计划都列出来(这个就不是确定性算法了，因为并不是所有情况你都能把所有情况都列出来)。事实上，现实世界遇到的问题，光靠人类，即便通过超级计算机，也是不太可能把所有情况都遍历完的，例如一个计划有 1000 个任务，就算忽略任务的所有其它考虑因素，就是 1000 个任务无任务要求，随便自由地排列，也就是 1000 个数的排列问题了，有多少种情况？是 1000 的阶乘！(有兴趣的同学自己回顾一下高中的排列公式）再考虑每个任务的各种属性，及每个属性的可能取值范围，那么组合下来，通常是天文数字了。

所以，OptaPlanner 在排程领域的作用就是帮人们对问题的可能性进行“遍历”，为什么我把遍历引起来呢？因为如果仅仅是无序地遍历，对所有情况一个一个试，那 OptaPlanner 就没啥作用了，我们可以通过自己编写程序,就能设计出遍历所有组合情况的代码来(能不能跑完那是另外一回事)。OptaPlanner 强大之处在于，他是有方法地去遍历的，它引入了禁忌搜索，模拟退火等算法，力求在固定的时间内，找到比傻傻地遍历更好的组合方案出来。事实上也证明它这些算法是有效的。

P 问题、NP 问题和 NPC 问题

http://www.matrix67.com/blog/archives/105

时间复杂度并不是表示一个程序解决问题需要花多少时间，而是当问题规模扩大后，程序需要的时间长度增长得有多快。也就是说，对于高速处理数据的计算机来说，处理某一个特定数据的效率不能衡量一个程序的好坏，而应该看当这个数据的规模变大到数百倍后，程序运行时间是否还是一样，或者也跟着慢了数百倍，或者变慢了数万倍。不管数据有多大，程序处理花的时间始终是那么多的，我们就说这个程序很好，具有 O(1)的时间复杂度，也称常数级复杂度；数据规模变得有多大，花的时间也跟着变得有多长，这个程序的时间复杂度就是 O(n)，比如找 n 个数中的最

大值；而像冒泡排序、插入排序等，数据扩大 2 倍，时间变慢 4 倍的，属于 O(n^2)的复杂度。还有一些穷举类的算法，所需时间长度成几何阶数上涨，这就是 O(a^n)的指数级复杂度，甚至 O(n!)的阶乘级复杂度。不会存在 O(2*n^2)的复杂度，因为前面的那个“2”是系数，根本不会影响到整个程序的时间增长。同样地，O (n^3+n^2)的复杂度也就是 O(n^3)的复杂度。因此，我们会说，一个 O(0.01*n^3)的程序的效率比 O(100*n^2)的效率低，尽管在 n 很小的时候，前者优于后者，但后者时间随数据规模增长得慢，最终 O(n^3)的复杂度将远远超过 O(n^2)。我们也说，O(n^100)的复杂度小于 O(1.01^n)的复杂度。

容易看出，前面的几类复杂度被分为两种级别，其中后者的复杂度无论如何都远远大于前者：**一种是 O(1),O(log(n)),O(n^a)等，我们把它叫做*多项式级的复杂度*，因为它的*规模 n 出现在底数的位置***；**另一种是 O(a^n)和 O(n!)型复杂度，它是*非多项式级*的，其复杂度计算机往往不能承受。**当我们在解决一个问题时，我们选择的算法通常都需要是多项式级的复杂度，非多项式级的复杂度需要的时间太多，往往会超时，除非是数据规模非常小。

自然地，人们会想到一个问题：**会不会所有的问题都可以找到复杂度为多项式级的算法呢？**很遗憾，答案是否定的。有些问题甚至根本不可能找到一个正确的算法来，这称之为“不可解问题”(Undecidable Decision Problem)。

再比如，输出从 1 到 n 这 n 个数的全排列。不管你用什么方法，你的复杂度都是阶乘级，因为你总得用阶乘级的时间打印出结果来。有人说，这样的“问题”不是一个“正规”的问题，正规的问题是让程序解决一个问题，输出一个“YES”或“NO”（这被称为**判定性问题**），或者一个什么什么的最优值（这被称为*最优化问题*）。那么，根据这个定义，我也能举出一个不大可能会有多项式级算法的问题来：Hamilton 回路。问题是这样的：给你一个图，问你能否找到一条经过每个顶点一次且恰好一次（不遗漏也不重复）最后又走回来的路（满足这个条件的路径叫做 Hamilton 回路）。这个问题现在还没有找到多项式级的算法。事实上，这个问题就是我们后面要说的 NPC 问题。

下面引入 P 类问题的概念:**如果一个问题可以找到一个能在多项式的时间里解决它的算法，那么这个问题就属于 P 问题。**P 是英文单词多项式的第一个字母。哪些问题是 P 类问题呢？通常 NOI 和 NOIP 不会出不属于 P 类问题的题目。我们常见到的一些信息奥赛的题目都是 P 问题。道理很简单，一个用穷举换来的非多项式级时间的超时程序不会涵盖任何有价值的算法。

接下来引入 NP 问题的概念。这个就有点难理解了，或者说容易理解错误。在这里强调（回到我竭力想澄清的误区上），NP 问题不是非 P 类问题。NP 问题是指可以在多项式的时间里验证一个解的问题。NP 问题的另一个定义是，可以在多项式的时间里猜出一个解的问题。比方说，我 RP 很好，在程序中需要枚举时，我可以一猜一个准。现在某人拿到了一个求最短路径的问题，问从起点到终点是否有一条小于 100 个单位长度的路线。它根据数据画好了图，但怎么也算不出来，于是来问我：你看怎么选条路走得最少？我说，我 RP 很好，肯定能随便给你指条很短的路出来。然后我就胡乱画了几条线，说就这条吧。那人按我指的这条把权值加起来一看，嘿，神了，路径长度 98，比 100 小。于是答案

出来了，存在比 100 小的路径。别人会问他这题怎么做出来的，他就可以说，因为我找到了一个比 100 小的解。在这个题中，找一个解很困难，但验证一个解很容易。验证一个解只需要 O(n)的时间复杂度，也就是说我可以花 O(n)的时间把我猜的路径的长度加出来。那么，只要我 RP 好，猜得准，我一定能在多项式的时间里解决这个问题。我猜到的方案总是最优的，不满足题意的方案也不会来骗我去选它。这就是 NP 问题。当然有不是 NP 问题的问题，即你猜到了解但是没用，因为你不能在多项式的时间里去验证它。下面我要举的例子是一个经典的例子，它指出了一个目前还没有办法在多项式的时间里验证一个解的问题。很显然，前面所说的 Hamilton 回路是 NP 问题，因为验证一条路是否恰好经过了每一个顶点非常容易。但我要把问题换成这样：试问一个图中是否不存在 Hamilton 回路。这样问题就没法在多项式的时间里进行验证了，因为除非你试过所有的路，否则你不敢断定它“没有 Hamilton 回路”。

之所以要定义 NP 问题，是因为通常只有 NP 问题才可能找到多项式的算法。我们不会指望一个连多项式地验证一个解都不行的问题存在一个解决它的多项式级的算法。相信读者很快明白，信息学中的号称最困难的问题——“NP 问题”，实际上是在探讨 NP 问题与 P 类问题的关系。

很显然，所有的 P 类问题都是 NP 问题。也就是说，能多项式地解决一个问题，必然能多项式地验证一个问题的解——既然正解都出来了，验证任意给定的解也只需要比较一下就可以了。关键是，人们想知道，是否所有的 NP 问题都是 P 类问题。我们可以再用集合的观点来说明。如果把所有 P 类问题归为一个集合 P 中，把所有 NP 问题划进另一个集合 NP 中，那么，显然有 P 属于 NP。现在，所有对 NP 问题的研究都集中在一个问题上，即究竟是否有 P=NP？通常所谓的“NP 问题”，其实就一句话：证明或推翻 P=NP。

NP 问题一直都是信息学的巅峰。巅峰，意即很引人注目但难以解决。在信息学研究中，这是一个耗费了很多时间和精力也没有解决的终极问题，好比物理学中的大统一和数学中的歌德巴赫猜想等。

目前为止这个问题还“啃不动”。但是，一个总的趋势、一个大方向是有的。**人们普遍认为，P=NP 不成立，也就是说，多数人相信，存在至少一个不可能有多项式级复杂度的算法的 NP 问题**。人们如此坚信 P≠NP 是有原因的，***就是在研究 NP 问题的过程中找出了一类非常特殊的 NP 问题叫做 NP-完全问题，也即所谓的 NPC 问题***。C 是英文单词“完全”的第一个字母。正是 NPC 问题的存在，使人们相信 P≠NP。下文将花大量篇幅介绍 NPC 问题，你从中可以体会到 NPC 问题使 P=NP 变得多么不可思议。

为了说明 NPC 问题，我们先引入一个概念——约化(Reducibility，有的资料上叫“归约”)。

简单地说，一个问题 A 可以约化为问题 B 的含义即是，可以用问题 B 的解法解决问题 A，或者说，问题 A 可以“变成”问题 B。《算法导论》上举了这么一个例子。比如说，现在有两个问题：求解一个一元一次方程和求解一个一元二次方程。那么我们说，前者可以约化为后者，意即知道如何解一个一元二次方程那么一定能解出一元一次方程。我们可以写出两个程序分别对应两个问题，那么我们能找到一个“规则”，按照这个规则把解一元一次方程程序的输入数据变一下，用在解一元二次方程的程序上，两个程序总能得到一样的结果。这个规则即是：两个方程的对应项系数不变，一元二次方程的二次项系数为 0。按照这个规则把前一个问题转换成后一个问题，两个问题就等价了。同样地，我们可以说，Hamilton 回路可以约化为 TSP 问题(Travelling Salesman Problem，旅行商问题)：在 Hamilton 回路问题中，两点相连即这两点距离为 0，两点不直接相连则令其距离为 1，于是问题转化为在 TSP 问题中，是否存在一条长为 0 的路径。Hamilton 回路存在当且仅当 TSP 问题中存在长为 0 的回路。

“问题 A 可约化为问题 B”有一个重要的直观意义：B 的时间复杂度高于或者等于 A 的时间复杂度。也就是说，问题 A 不比问题 B 难。这很容易理解。既然问题 A 能用问题 B 来解决，倘若 B 的时

间复杂度比 A 的时间复杂度还低了，那 A 的算法就可以改进为 B 的算法，两者的时间复杂度还是相同。正如解一元二次方程比解一元一次方程难，因为解决前者的方法可以用来解决后者。

很显然，约化具有一项重要的性质：约化具有传递性。如果问题 A 可约化为问题 B，问题 B 可约化为问题 C，则问题 A 一定可约化为问题 C。这个道理非常简单，就不必阐述了。

现在再来说一下约化的标准概念就不难理解了：如果能找到这样一个变化法则，对任意一个程序 A 的输入，都能按这个法则变换成程序 B 的输入，使两程序的输出相同，那么我们说，问题 A 可约化为问题 B。

当然，我们所说的“可约化”是指的可“多项式地”约化(Polynomial-time Reducible)，即变换输入的方法是能在多项式的时间里完成的。约化的过程只有用多项式的时间完成才有意义。

好了，从约化的定义中我们看到，一个问题约化为另一个问题，时间复杂度增加了，问题的应用范围也增大了。通过对某些问题的不断约化，我们能够不断寻找复杂度更高，但应用范围更广的算法来代替复杂度虽然低，但只能用于很小的一类问题的算法。再回想前面讲的 P 和 NP 问题，联想起约化的传递性，自然地，我们会想问，如果不断地约化上去，不断找到能“通吃”若干小 NP 问题的一个稍复杂的大 NP 问题，那么最后是否有可能找到一个时间复杂度最高，并且能“通吃”所有的 NP 问题的这样一个超级 NP 问题？答案居然是肯定的。也就是说，存在这样一个 NP 问题，所有的 NP 问题都可以约化成它。换句话说，只要解决了这个问题，那么所有的 NP 问题都解决了。这种问题的存在难以置信，并且更加不可思议的是，这种问题不只一个，它有很多个，它是一类问题。这一类问题就是传说中的 NPC 问题，也就是 NP-完全问题。NPC 问题的出现使整个 NP 问题的研究得到了飞跃式的发展。我们有理由相信，NPC 问题是最复杂的问题。再次回到全文开头，我们可以看到，人们想表达一个问题不存在多项式的高效算法时应该说它“属于 NPC 问题”。此时，我的目的终于达到了，我已经把 NP 问题和 NPC 问题区别开了。到此为止，本文已经写了近 5000 字了，我佩服你还能看到这里来，同时也佩服一下自己能写到这里来。

NPC 问题的定义非常简单。同时满足下面两个条件的问题就是 NPC 问题。首先，它得是一个 NP 问题；然后，所有的 NP 问题都可以约化到它。证明一个问题是 NPC 问题也很简单。先证明它至少是一个 NP 问题，再证明其中一个已知的 NPC 问题能约化到它（由约化的传递性，则 NPC 问题定义的第二条也得以满足；至于第一个 NPC 问题是怎么来的，下文将介绍），这样就可以说它是 NPC 问题了。

既然所有的 NP 问题都能约化成 NPC 问题，那么只要任意一个 NPC 问题找到了一个多项式的算法，那么所有的 NP 问题都能用这个算法解决了，NP 也就等于 P 了。因此，给 NPC 找一个多项式算法太不可思议了。因此，前文才说，“正是 NPC 问题的存在，使人们相信 P≠NP”。我们可以就此直观地理解，NPC 问题目前没有多项式的有效算法，只能用指数级甚至阶乘级复杂度的搜索。

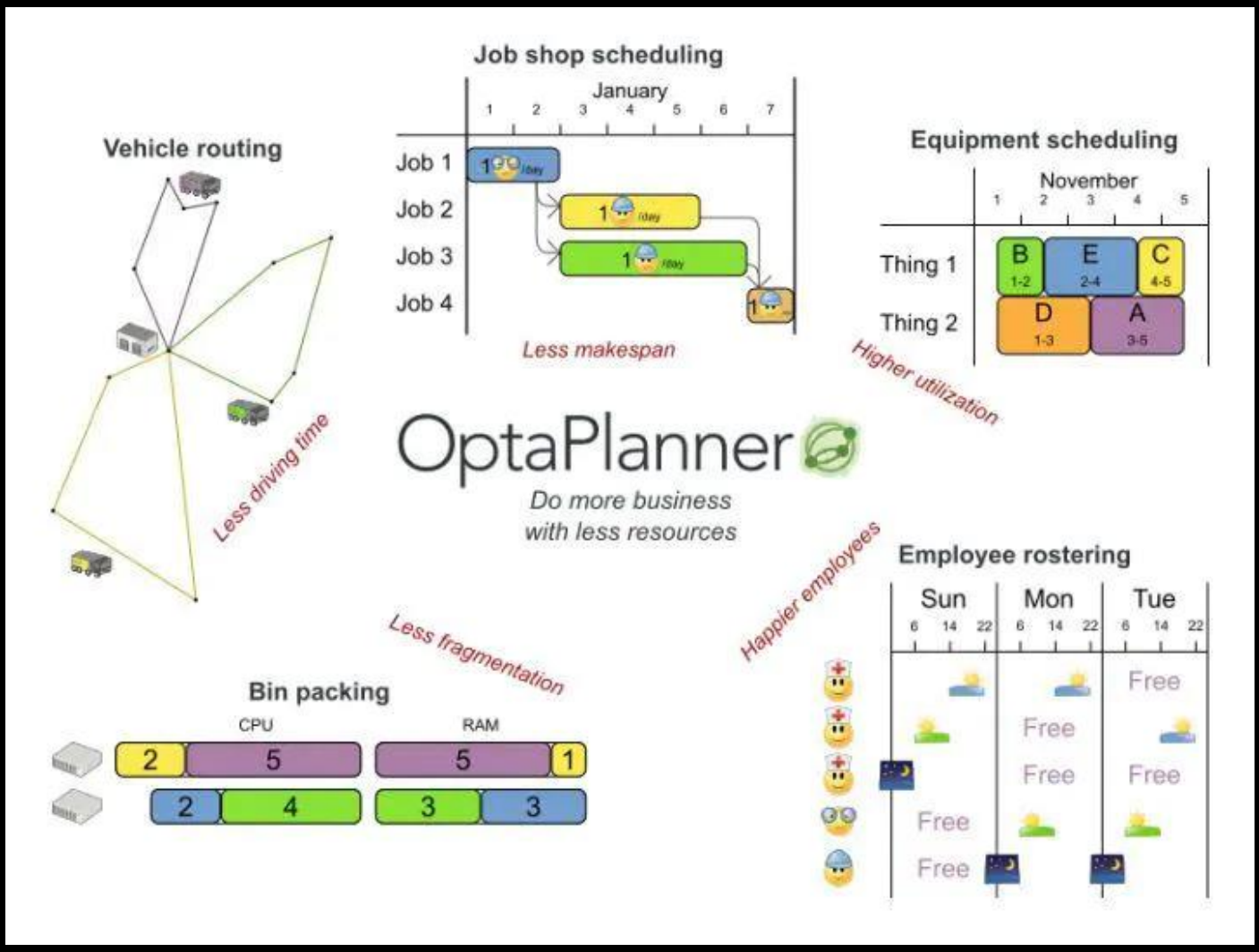
顺便讲一下 NP-Hard 问题。NP-Hard 问题是这样一种问题，它满足 NPC 问题定义的第二条但不一定要满足第一条（就是说，NP-Hard 问题要比 NPC 问题的范围广）。NP-Hard 问题同样难以找到多项式的算法，但它不列入我们的研究范围，因为它不一定是 NP 问题。即使 NPC 问题发现了多项式级的算法，NP-Hard 问题有可能仍然无法得到多项式级的算法。事实上，由于 NP-Hard 放宽了限定条件，它将有可能比所有的 NPC 问题的时间复杂度更高从而更难以解决。

组合：N 个里面选择 M 个是什么问题？ 复杂度是多少？
轻易放飞面试

作用、架构

OptaPlanner 用官方的描述就是**可以帮你规划出一个用更少的资源做更多更好事情的规划引擎。**如下图列出它可以做的工作领域（这仅仅是 OptaPalnner 的 Example 里有的示例，其实所有关于规则的问题，属于 NPC 的问题，只要你能把它抽象并建模成 OptaPlanner 可识别的模型，你就可以用 OptaPlanner 来解决）：车辆调度、工作排程、设备排程，Bin Packing(就是用袋子装石头那个问题啦)及员工排班。

构架和应用兼容性：



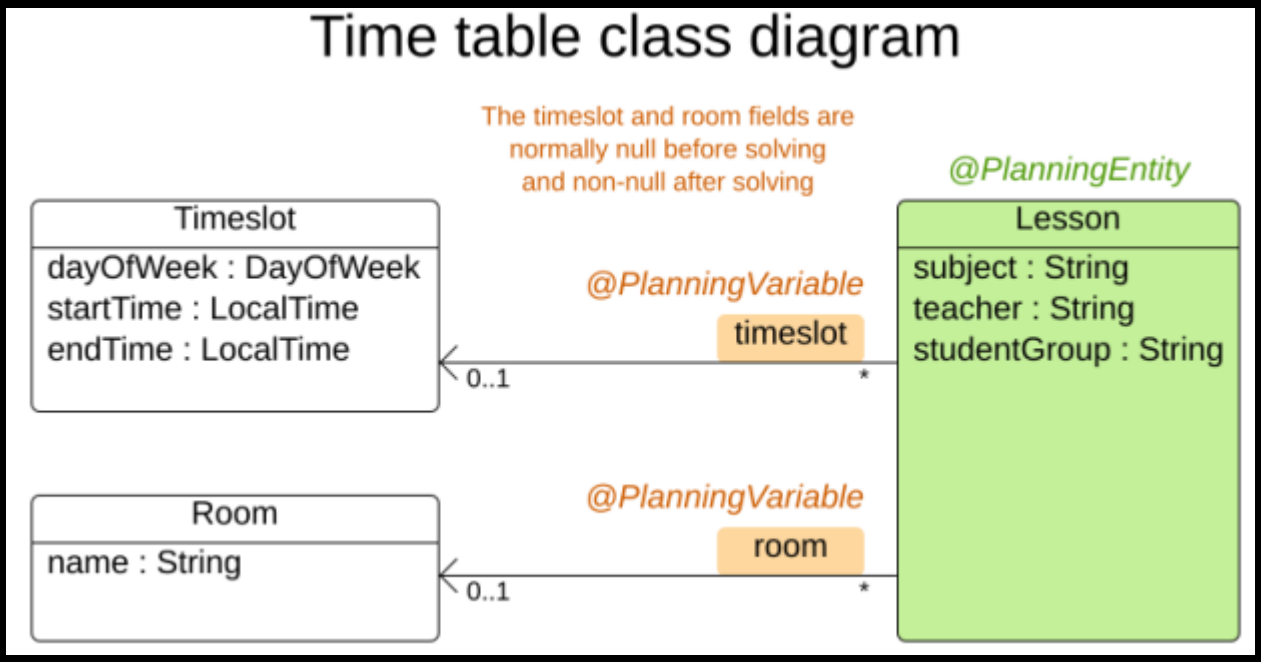
原理

那么 OptaPlanner 是通过什么方法，高效地帮我们在尽量短的时间内，找到更佳的方案呢？还记得上一任篇老农提到，我们做排程的时候，**通常有两种约束条件，分别是不可违反的硬约束**，如果一个计划违反了硬约束，那这个计划就是不可行的，例如：生产计划中，把产品固定工序的加工

次序调乱了，又或者把产品分配到错误的机台上生产(这些约束条件都是业务上你们自己定义的)，那么 OptaPlanner 就把它定义为违反了硬约束。**另一类就是软约束，就是那种可以违反，但违反得越多，就会影响越大**（影响包括成本、效率、质量等），所得结果方案的质量越差；违反这种约束，OptaPlanner 就把它定义为违反了软约束。OptaPlanner 就是对这两类约束进行打分，硬约束对应的是硬分数，软约束对应的是软分数。那么得分越高，就表示对应方案的质量越高。在计算这些约束分数的过程中，OptaPlanner 会保持优先优化硬分数、然后在硬分数最优的基础上，再去优化软分数的原则，来寻找最佳方案。例如：两个方案 A、B 对比，方案 A 的硬分数比方案 B 的硬分数高 1 分，方案 B 的软分数比方案 A 的软高出 10 万分。那么 OptaPlanner 最后还是认为方案 A 更佳。也就相当于我们写 SQL 脚本时，order by 子句中前后两个字段的关系了，靠前的字段排序比靠后的字段更优先。

硬约束（不可违反），和软约束（尽量不要违反，但将不可避免；如果违反，尽可能令违反的程度减到最小）

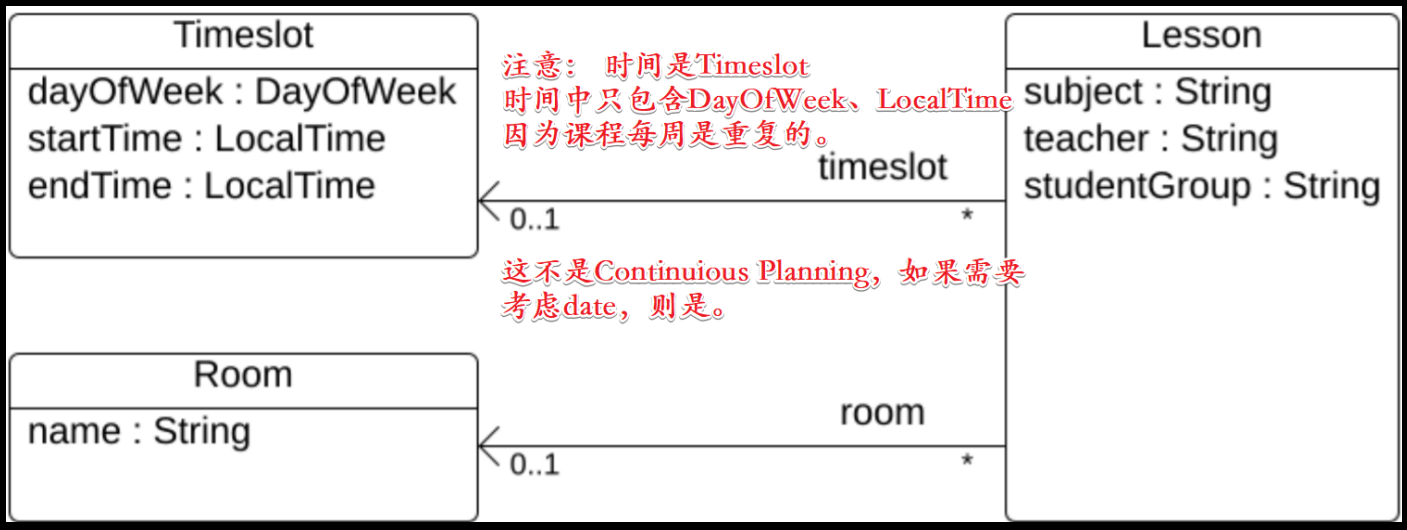
OptPlanner User Guide



@PlanningVariable

Continuious Planning

先看课程表排程：



那么如何持续排程？

持续排程应用于一个无限的场景，意思是排程的 Entity 无限增长，每天都有新的实体需要排程
持续排程实际是以一种滑动窗口的排序方式。

持续排程场景分为三部分：

1. History: 过去(已排程，已确定)，
   是 Pinned 实体
2. Published:
   Upcoming time periods that have been published. They contain only pinned and/or semimovable planning entities.
3. Draft:
   Upcoming time periods after the published time periods that can change freely. They contain movable planning entities, except for any that are pinned for other reasons
4. Unplanned(out of scope)
   不在当前的排程窗口中。

Pinned planning entities

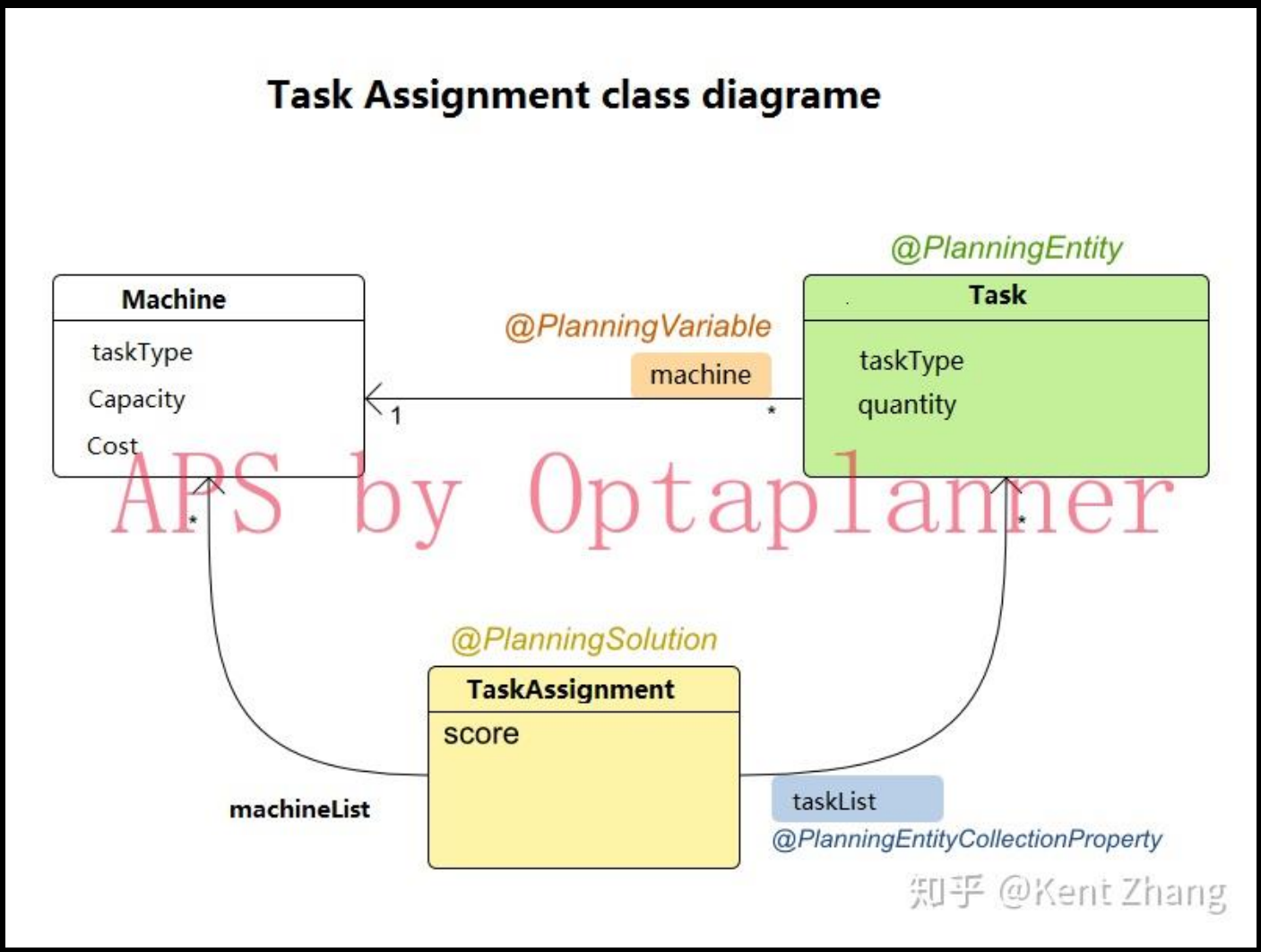
一个 pinned 的 entity 在 solve 期间不会改变，
使用注解 @PlainningPin
注解用在 boolean 的 getter 或者域上面

案例

机器-生产任务

https://zhuanlan.zhihu.com/p/69900819
类似 Cloud Balance

有机器：机器生产某种产品，同时有产能和成本
有任务：任务需要生成的产品类型，以及数量
如何调度？



1. Task，表示任务的实体类，它被注解为@PlanningEntity， 它有三个属性：taskType - 当前任务的类型； quantity - 生产量；machine - 任务将要被分配到的机台。其中 machine 属性被注解为@PlanningVariable，表示规划过程中，这个属性的值将被 plan 的，即通过调整这个属性来得到不同的方案。另外，**作为一个 Planning Entity，它必须有一个 ID 属性，用于在规划运行过程中识别不同的对象，这个 ID 属性被注解为@PlanningId**。 本例中所有实体类都继承了一个通用的类 - AbstractPersistable， 该父类负责维护此所有对象的 ID。Task 类也继承于它，因此，将该类的 ID 属性注解为@PlanningId 即可。另外，作为 Planning Entity， 它必须有一无参构造函数，若你在此类实现了有参构造的话，需要显式地实现一个无参构造函数。
2. Machine， 表示机台的实体类，它属于 ProblemFact，在其中保存了在规划过程中会用到的

属性，此类反映一个机台相关信息的属性： taskType - 可处理的任务类型； capacity - 当前机台的产能；cost -当前机台的成本。

3. TaskAssignment， 此类用来描述整个解决方案的固定类，它的结构描述了问题的各种信息，在 OptaPlanner 术语中，在执行规划前，它的对象被称作一个 Problem，完成规划并获得输出之后，输出的 TaskAssignment 对象被称作一个 Solution。它具有固定的特性要求： 必须被注解为@PlanningSolution；本例中，它至少有三个属性： machineList - 机台列表，就是可以用于分配任务的机台，本例中指的就是上述那 6 个机台；taskList - 就是需要被规划（或称分配机台）的 10 个任务，这个属性需要被注解为@PlanningEntityCollectionPropery. 还有一个是 score 属性，它用于在规划过程中对各种约束的违反情况进行打分，因为本例中存在了硬约束与软约束。因此我们使用的 Score 为 HardSoftScore.

```
public class AbstractPersistable implements Serializable,
Comparable<AbstractPersistable> {

    protected Long id;

    protected AbstractPersistable() {
    }

    protected AbstractPersistable(long id) {
        this.id = id;
    }

    @PlanningId
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Override
    public int compareTo(AbstractPersistable other) {
        return new CompareToBuilder().append(getClass().getName(),
other.getClass().getName()).append(id, other.id)
                .toComparison();
    }

    @Override
    public String toString() {
        return getClass().getName().replaceAll(".*\\.", "") + "-" + id;
    }
}
```

```
@Getter
@Setter
public class Machine extends AbstractPersistable{
    private String yarnType;
    private int capacity;
    private int cost;

    public Machine(int id, String yarnType, int capacity, int cost) {
        super(id);
        this.yarnType = yarnType;
        this.capacity = capacity;
        this.cost = cost;
    }

    public String getYarnType() {
```

```
        return yarnType;
    }

    public int getCapacity() {
        return capacity;
    }

    public int getCost() {
        return cost;
    }
}
```

```
@PlanningEntity
@Getter
@Setter
@NoArgsConstructor
public class Task  extends AbstractPersistable{
    private String requiredYarnType;
    private int amount;

    @PlanningVariable(valueRangeProviderRefs={"machineRange"})
    private Machine machine;

    public Task(int id, String requiredYarnType, int amount) {
        super(id);
        this.requiredYarnType = requiredYarnType;
        this.amount = amount;
    }

    public String getRequiredYarnType() {
        return requiredYarnType;
    }

    public int getAmount() {
        return amount;
    }

    public Machine getMachine() {
        return machine;
    }
}
```

```
@PlanningSolution
@NoArgsConstructor
@Getter
@Setter
public class TaskAssignment extends AbstractPersistable {
    @PlanningScore
    private HardSoftScore score;

    @ProblemFactCollectionProperty
    @ValueRangeProvider(id = "machineRange")
    private List<Machine> machineList;

    @PlanningEntityCollectionProperty
    @ValueRangeProvider(id = "taskRange")
    private List<Task> taskList;


    public TaskAssignment(List<Machine> machineList, List<Task> taskList) {
        //super(0);
        this.machineList = machineList;
        this.taskList = taskList;
    }
```

```
    public HardSoftScore getScore() {
        return score;
    }

    public List<Machine> getMachineList() {
        return machineList;
    }

    public List<Task> getTaskList() {
        return taskList;
    }
}
```

```
public class App {

    public static void main(String[] args) {
        startPlan();
    }

    private static void startPlan(){
        List<Machine> machines = getMachines();
        List<Task> tasks = getTasks();

        SolverFactory<TaskAssignment> solverFactory =
            SolverFactory.createFromXmlFile(new
File("./src/main/resources/DemoConfiguration.xml"));

        Solver<TaskAssignment> solver = solverFactory.buildSolver();
        TaskAssignment unassignment = new TaskAssignment(machines, tasks);

        TaskAssignment assigned = solver.solve(unassignment);//启动引擎

        List<Machine> machinesAssigned = assigned.getTaskList()
                .stream().map(Task::getMachine).distinct().collect(Collectors.to
List());
        for(Machine machine : machinesAssigned) {
            System.out.print("\n" + machine + ":");
            List<Task> tasksInMachine = assigned.getTaskList().stream().filter(x
-> x.getMachine().equals(machine)).collect(Collectors.toList());
            for(Task task : tasksInMachine) {
                System.out.print("->" + task);
            }
        }
    }


    private static List<Machine> getMachines() {
        // 六个机台
        Machine m1 = new Machine(1, "Type_A", 300, 100);
        Machine m2 = new Machine(2, "Type_A", 1000, 100);
        Machine m3 = new Machine(3, "TYPE_B", 1000, 300);
        Machine m4 = new Machine(4, "TYPE_B", 1000, 100);
        Machine m5 = new Machine(5, "Type_C", 1100, 100);
        Machine m6 = new Machine(6, "Type_D", 900, 100);

        List<Machine> machines = new ArrayList<Machine>();
        machines.add(m1);
        machines.add(m2);
        machines.add(m3);
        machines.add(m4);
        machines.add(m5);
        machines.add(m6);

        return machines;
    }
```

```
    private static List<Task> getTasks(){
        // 10个任务
        Task t1 = new Task(1, "Type_A", 100);
        Task t2 = new Task(2, "Type_A", 100);
        Task t3 = new Task(3, "Type_A", 100);
        Task t4 = new Task(4, "Type_A", 100);
        Task t5 = new Task(5, "TYPE_B", 800);
        Task t6 = new Task(6, "TYPE_B", 500);
        Task t7 = new Task(7, "Type_C", 800);
        Task t8 = new Task(8, "Type_C", 300);
        Task t9 = new Task(9, "Type_D", 400);
        Task t10 = new Task(10, "Type_D", 500);

        List<Task> tasks = new ArrayList<Task>();
        tasks.add(t1);
        tasks.add(t2);
        tasks.add(t3);
        tasks.add(t4);
        tasks.add(t5);
        tasks.add(t6);
        tasks.add(t7);
        tasks.add(t8);
        tasks.add(t9);
        tasks.add(t10);

        return tasks;
    }
}
```

配置文件DemoConfiguration.xml:

```
<solver>

    <!-- Domain model configuration -->
    <solutionClass>cn.edw.myoptaplanner.demo.TaskAssignment</solutionClass>
    <entityClass>cn.edw.myoptaplanner.demo.Task</entityClass>


    <!-- Score configuration -->
    <scoreDirectorFactory>
        <scoreDrl>DemoDools.drl</scoreDrl>
    </scoreDirectorFactory>

    <!-- Optimization algorithms configuration -->
    <termination>
        <secondsSpentLimit>10</secondsSpentLimit>
    </termination>
</solver>
```

使用drools:

```
package cn.edw.myoptaplanner.demo;

import org.optaplanner.core.api.score.buildin.hardsoft.HardSoftScoreHolder;
import cn.edw.myoptaplanner.demo.Task;
import cn.edw.myoptaplanner.demo.Machine;
import cn.edw.myoptaplanner.demo.TaskAssignment;

global HardSoftScoreHolder scoreHolder;

//类型匹配规则
rule "yarnTypeMatch"
when
    Task(machine != null, machine.yarnType != requiredYarnType)
then
    scoreHolder.addHardConstraintMatch(kcontext, -10000);
end
```

```
rule "machineCapacity"
when
    $machine : Machine($capacity : capacity)
    accumulate(
        Task(
            machine == $machine,
            $amount : amount);
        $amountTotal : sum($amount);
        $amountTotal > $capacity
        )
then
    scoreHolder.addHardConstraintMatch(kcontext, $capacity - $amountTotal);
end

rule "machineCost_used"
when
    $machine : Machine($cost : cost)
    exists Task(machine == $machine)
then
    scoreHolder.addSoftConstraintMatch(kcontext, -$cost);
end
```

Drools 规则文件

以.drl 为扩展名的文件，是 Drools 中的规则文件，规则文件的编写，遵循 Drools 规则语法。

官方文档:

https://docs.jboss.org/drools/release/7.0.0.Final/drools-docs/html_single/index.html#_droolslanguagereferencechapter

DRL 文件的整体结构如下:

```
package package-name

imports

globals

functions

queries

rules
```

对于上述元素，其顺序在 drl 文件中的顺序不重要，除了 package-name，必须是 drl 文件的第一个元素。上述所有的元素都是可选的

规则（Rule）组成

一个简单的规则结构如下所示

```
rule "name"
    attributes
    when
        LHS
    then
        RHS
```

```
end
```

```
package droolscours

//list any import classes here.
                (1)

//declare any global variables here
              (2)

rule "Your First Rule"  (3)

    when
        //conditions
    then
        //actions

end

rule "Your Second Rule"
    //include attributes such as "salience" here...
    when
        //conditions
    then
        //actions

end
```

```
rule "Your First Rule"

    when
        Account(  )
        //conditions
    then
        System.out.println("The account exits");

end
```

通常，规则文件中是不需要标点符号的。即使是规则名“name”上的双引号也是可选的。attributes 展示规则的执行方式，也是可选的。LHS 是规则的条件部分，遵循一个固定的语法规则，在下面的内容会详细介绍。RHS 通常是一个可以本地执行的代码块。

When and how is a rule fired? –rule 怎么触发?

1. drools will look at all rules that can apply and put it in its agenda.
2. drools will execute the rule that is on top of its agenda
3. Once fired, the rule will be deactivated(失效)

4. We have to tell drools of a state change in one of the facts in the when part (lhs) to make him reconsider the rule.
5. A state change can be an **insert, update, or delete** (retract).

关键字(Key Words)

从Drools 5开始，提出了“软”关键字和“硬”关键字的概念。硬关键字是保留的，不允许开发人员在编写规则文件时使用。硬关键字有如下几个：

1. true
2. false
3. null

软关键字是在文件内容中公认的一些字，开发人员可以在任何地方使用这些关键字，但是为了消除混淆，并不推荐开发人员在实际中使用这些关键字。软关键字有如下几个：

lock-on-active
date-effective
date-expires
no-loop
auto-focus
activation-group
agenda-group
ruleflow-group
entry-point
duration
package
import
dialect
salience
enabled
attributes
rule
extend
when
then
template
query
declare
function
global
eval
not
in

or
and
exists
forall
accumulate
collect
from
action
reverse
result
end
over
init

注释(Comments)

注释是规则文件中一段会被规则引擎自动忽略的一段文本。drl 文件中的注释采用类 Java 语法的方式，可以分为两类：单行注释和多行注释。

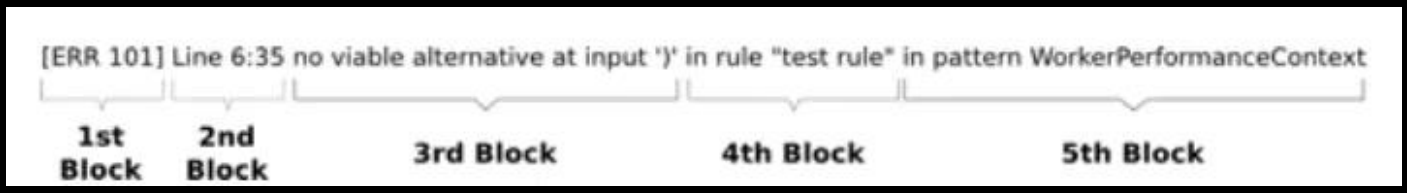
单行注释
单行注释可以简单的使用双斜杠"//"来标识。语法解析器会自动忽视其后的所有内容。
“#”开头的单行注释在 drl 文件中已经被弃用。

多行注释主要用于在代码块外对整个文件进行注释，以"/"开头和"/"结尾的之间所有的内容都会被语法解析器解释为注释。

错误信息(Error Messages)

Drools 5 开始提出了标准的错误信息。标准化的目的是为了使开发者可以更准确更简单定位错误问题所在。接下来将介绍如正确理解和识别错误信息。

标准的错误信息格式如下所示。



错误信息包含以下几个部分：
1. 1st Block：表明当前错误的错误码。
2. 2st Block：错误可能发生的行和列。
3. 3st Block：错误信息描述。
4. 4st Block：指明错误发生的规则，函数，模板，查询等。
5. 5st Block：指明错误发生于何种模式。一般不是强制性的。

错误码描述

101: No viable alternative

错误码 101 指明了最常见的错误，语法解析器无法找到替代方案。

```
1: rule one
2:   when
3:     exists Foo()
4:     exits Bar()  // "exits"
5:   then
6: end
```

上述示例会产生如下错误信息：

[ERR 101] Line 4:4 no viable alternative at input 'exits' in rule one

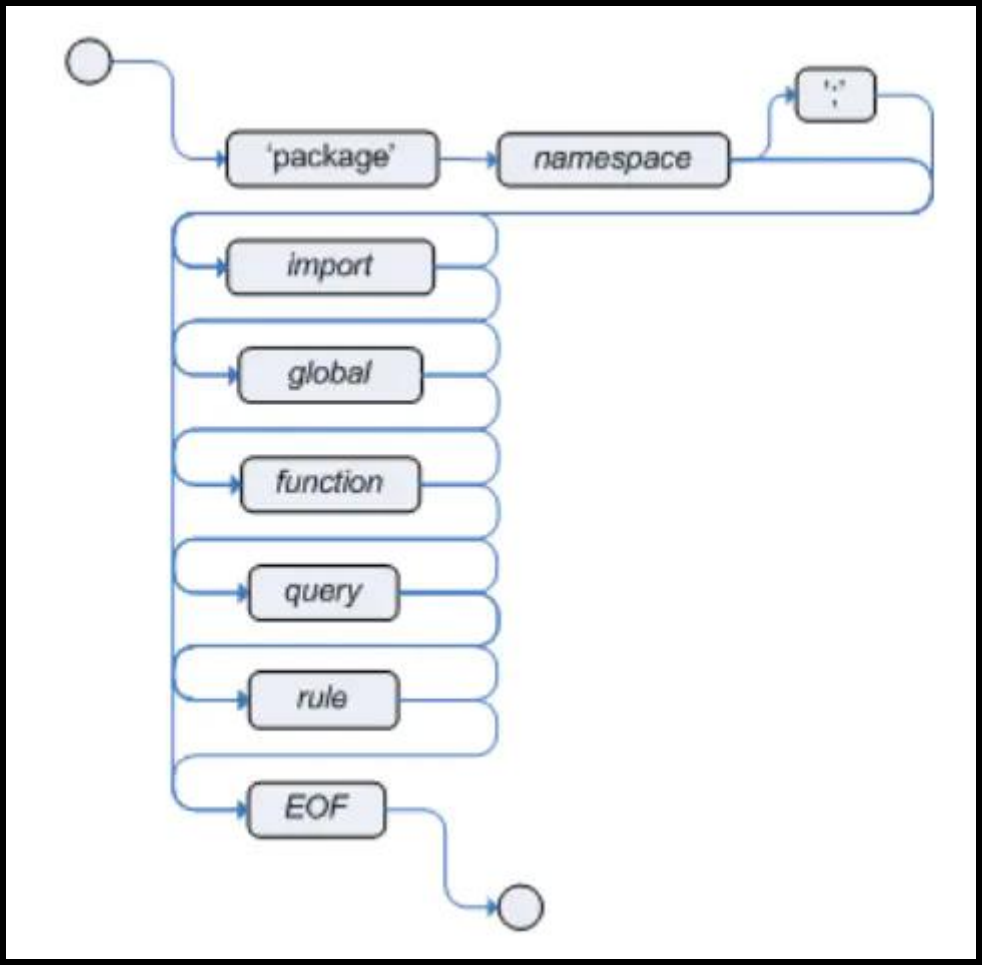
上述例子中的 exits != exists，解析器找不到 exits 的替代方案，于是报错。

102: Mismatched input

该错误表明，语法解析器在当前输入位置中未找到一个特定的符号。

包(Package)

package 是一系列 rule 或其他相关构件如 imports， globals 的容器。这个成员之间相互关联，一个 package 代表了一个命名空间，其中的每个 rule 的名字都是唯一的，package 名字本身就是命名空间，与实际的文件和文件夹没有任何关系。常见的结构是，一个文件包含多个 rule 的文件就定义成一个包。以下的线路图表明了一个包中包含的所有组成元素。

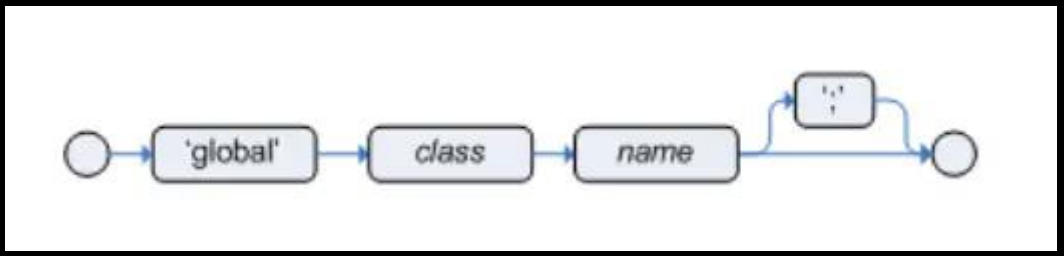


需要注意的是，一个包必须有一个命名空间，且其声明必须遵守 Java 命名规范。package 语句必须出现在包的首行，其他组成部分的出现顺序无关紧要。其中 package 语句结尾的分号;是可选的。

import

Drools 文件中的 import 语句功能与 Java 中的 import 语句功能类似。使用 import 时，必须指定对象的全称限定路径和类型名。Drools 会自动导入 Java 相同名字的包中的所有类，也会导入 java.lang.*。

global



global 用于定义全局变量。用于使应用对象对一系列规则有效。通常，用于向规则提供全局的数据和服务，特别是一些用于规则序列的应用服务，如日志、规则序列中累加的值等。全局的变量是不会插入 Woking Memory 中的，另外，全局变量不要用于建立规则的条件部分，除非它是一个不会改变的常量。全部变量的改变不会通知到规则引擎，规则引擎不跟踪全局变量的变化，因为他们并没有加

入到 Woking Memory 中。全局变量使用不当，会产生很多不可思议的结果。如果多个包中同时定义了相同标识符的全局变量，那么这些全局变量必须是相同类型，并会引用一个相同的全局值。为了更好地使用全局变量，必须遵循以下规则：

1. 在规则文件中定义常量，并使用这些常量。

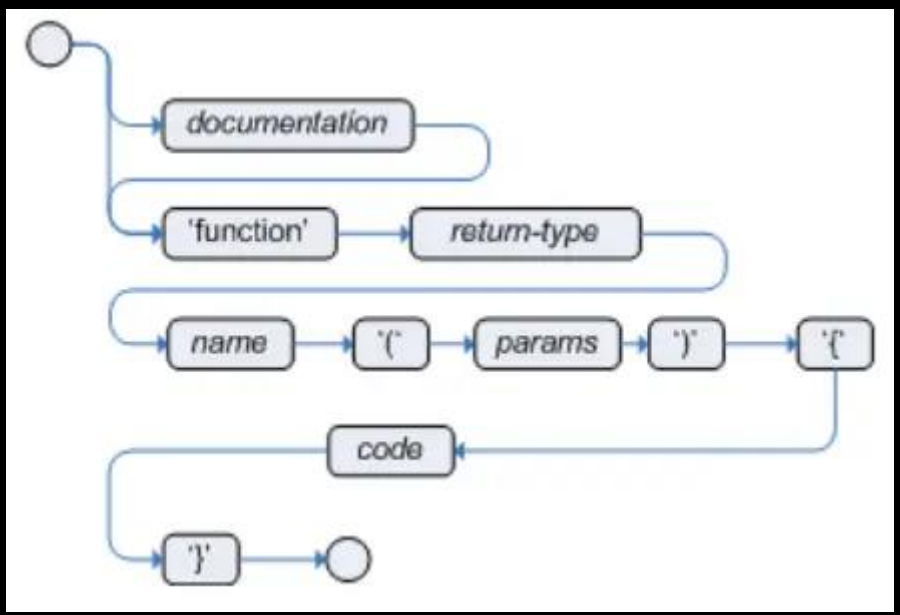
```
global java.util.List myGlobalList;
rule "Using a global"
when
    eval( true )
then
    myGlobalList.add( "Hello World" );
end
```

2. 在工作内存中，为全局变量设值。在从内存中获取所有的 fact 之前，最好将所有的全局变量设置。

```
List list = new ArrayList();
KieSession kieSession = kiebase.newKieSession();
kieSession.setGlobal( "myGlobalList", list );
```

全局变量并不是用来在规则间共享数据，而且最好不要用于在规则间共享数据。规则总是对工作内存的状态产生推理和反应，因此，如果想在规则之间传递数据，可以将这些数据作为 facts 传入工作内存。因为规则引擎并不会关心和跟踪这些全局变量的变化。

函数(Function)



function 提供了一种在规则源文件中插入语义代码的方式，与在普通 Java 类中不同。他们需要帮助类，否则不能做任何事情。(实际上，编译器会针对这些句子自动产生帮助类。)在规则中使用函数的最主要优点就是你可以把所有逻辑放在一个地方，你可以根据需要更改这些函数的逻辑。函数通常用于在规则的 then 部分调用某些动作，特别是一些经常被用到的而传入参数不一样的动作。

```
function String hello(String name) {
    return "Hello "+name+"!";
}
```

需要注意的是，这里我们使用了 function 关键字，即使它并不是 Java 语言的一部分。参数和返回值的定义和传入与 Java 语言一致。当然，参数和返回值并不是必须的。另外，我们可以使用一个帮助类中的静态方法，如：Foo.hello()。

Drools 支持函数的导入，我们所要做的就是：

```
import function my.package.Foo.hello
```

不需要考虑函数的定义和导入方式，我们可以直接在需要的地方直接使用函数名调用这个函数。

```
rule "using a static function"
when
    eval( true )
then
    System.out.println( hello( "Bob" ) );
end
```

类型声明(Type Declaration)

在规则引擎中，类型声明有两个目的：允许新的类型声明；允许元数据类型的声明。

1. 声明新类型：Drools 工作时会将外部的普通 Java 对象作为事实。但是有时候使用者想要自己定义一些规则引擎可以直接使用的模型，而不用担心用 Java 之类的底层语言来创建对象。另外有时候，当一个域模型已经建立，但是用户想或者需要用一些主要在推理过程中使用的实体来完善这个模型。
2. 声明元数据：事实往往会有一些与之相关的元数据信息。元信息的样本包含的任何种类的数据都不能代表事实的属性，且在该事实类型的所有实例中都是不变的。这些元信息在规则引擎的运行和推理古城中需要被查询。

声明新类型

为了定义新类型，我们需要使用关键字 declare，然后是一系列域，最终以 end 关键字结尾。一个新的 fact 必须有一系列域，否则规则引擎会去 classpath 中寻找一个存在的 fact```类，如果没找到，会报错。下面给出定义新类型的几个例子。

```
declare Address
   number : int
   streetName : String
   city : String
end
```

上面的例子中我们定义了一个新类型 Address，这个 **fact** 有三个属性，每个属性都具有一个 Java 中有效的数据类型。

```
import java.util.Date

declare Person
    name : String
    dateOfBirth : Date
    address : Address
end
```

当我们声明一个新的 fact 类型时，Drools 会在编译期间生成实现自一个表示该 fact 类型的 Java 类的字节码。这个生成的 Java 类：

```
public class Person implements Serializable {
    private String name;
    private java.util.Date dateOfBirth;
    private Address address;

    // empty constructor
```

```
    public Person() {...}

    // constructor with all fields
    public Person( String name, Date dateOfBirth, Address address ) {...}

    // if keys are defined, constructor with keys
    public Person( ...keys... ) {...}

    // getters and setters
    // equals/hashCode
    // toString
}
```

该类型生成的 class 是一个普通的 Java 类，可以在规则中直接使用，就向其他 fact 一样。

```
rule "Using a declared Type"
when
    $p : Person( name == "Bob" )
then
    // Insert Mark, who is Bob's mate.
    Person mark = new Person();
    mark.setName("Mark");
    insert( mark );
end
```

声明枚举类型

DRL 同时支持声明枚举类型。该类型声明需要另外一种关键字 enum，然后以都好分割可接收值的列表，最后以分号结束

```
declare enum DaysOfWeek

SUN("Sunday"),MON("Monday"),TUE("Tuesday"),WED("Wednesday"),THU("Thursday"),FRI
("Friday"),SAT("Saturday");

   fullName : String
end
```

声明完成之后，该枚举类型可以用于之后的规则中。

```
rule "Using a declared Enum"
when
   $p : Employee( dayOff == DaysOfWeek.MONDAY )
then
   ...
end
```

声明元数据

在 Drools 中元数据会被分配给一系列不同对象的构造：fact 类型，fact 属性和规则。Drools 使用@符号来引出元数据，使用使用如下格式：

```
@metadata_key( metadata_value )
```

其中 metadata_value 是可选的。

Drools 允许声明任何任意元数据属性，但是当其他属性在运行时仅仅对查询有效时，有些属性对于规则引擎来说具有不同的意义。Drools 允许为 fact 类型和 fact 属性声明元数据。所有的元数据在该属性被分配到 fact 类型前声明，而在向一个特定属性分配值之前声明。

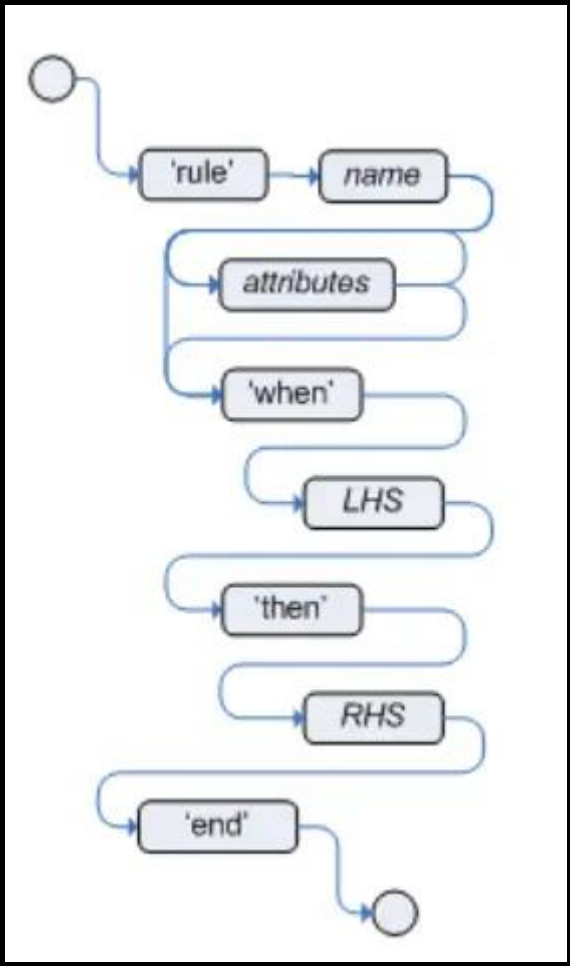
```
import java.util.Date
```

```
declare Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )

    name : String @key @maxLength( 30 )
    dateOfBirth : Date
    address : Address
end
```

上面的例子中，声明了两个 fact 类型(@author 和 @dateOfCreation)的元数据元素，另外为 name 属性声明了两个(@key 和@maxLength)元数据。其中@key 没有必须值，所有括号和值均被省略了。

规则(Rule)



一个规则，指定当（when）一系列特定的条件发生时（左手边 LHS），然后（then）做出一系列相应的动作（右手边 RHS）。一个常见的问题就是：为什么使用 when 而不是 if，因为 if 通常是执行流程中特定时间点的一部分，是一个需要检查的条件。相反的，when 指明的是一个不绑定于任何特定判断序列或时间点的条件判断，它在规则引擎的声明周期内的任何一个条件发生的情况下都可能触发，不管这个条件是否遇到，这些动作都会执行。

一个规则在一个包中必须具有独一无二的名字。如果在一个 DRL 文件中重复定义两个相同名字的规则，在加载的时候就会报错。如果向包中添加一个名字已经存在的规则，该规则会覆盖掉之前的同名规则。如果一个规则命中存在空格符，最好使用双引号将规则名包括起来。

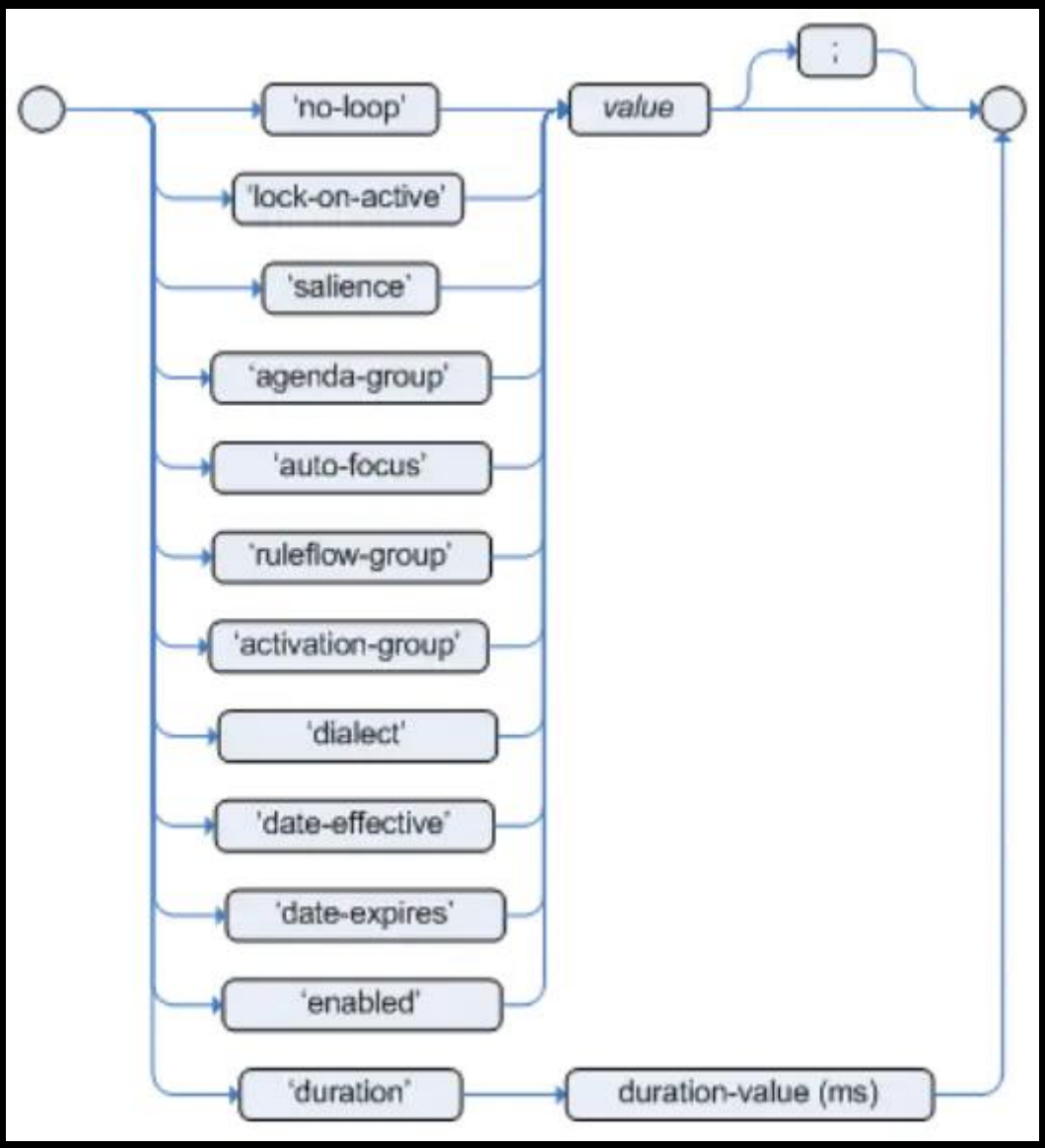
规则的属性不是必须的，且属性最好写成一行。

规则中的 LHS 在关键字 when 的后面，同样的，RHS 应该在关键字 then 的后面，规则最后以关键

字 end 结尾。另外，规则不准嵌套。

```
rule "<name>"
    <attribute>*
when
    <conditional element>*
then
    <action>*
end
```

```
rule "Approve if not rejected"
  salience -100
  agenda-group "approval"
    when
        not Rejection()
        p : Policy(approved == false, policyState:status)
        exists Driver(age > 25)
        Process(status == policyState)
    then
        log("APPROVED: due to no objections.");
        p.setApproved(true);
end
```



规则属性

规则属性显式地声明了对规则行为的影响，有些规则属性很简单，有些规则属性是复杂的子系统的一

部分，如规则流。为了从 Drools 中获得更多东西，我们需要确保对每一个规则属性均有正确的认识。常用的规则属性有如下：

常用的规则属性有如下：

1. no-loop
   默认值：false
   type：Boolean
   当规则序列更改了一个 fact，会导致该规则会被重新触发，以至于产生一个无限循环。当设置为 true 时，当前规则只会被激活一次。
2. ruleflow-group
   默认值：N/A
   type：String
   ruleflow 是 Drools 的特色之一，可以让你自己控制规则的命中。同一个 ruleflow-group 中的所有规则只有当该组激活时才能被命中。
3. lock-on-active
   默认值：false
   type：Boolean
   不管何时 ruleflow-group 和 agenda-group 被激活，只要其中的所有规则将 lock-on-active 设置为 true，那么这些规则都不会再被激活，不管一开始怎么更新，这些匹配的规则都不会被激活。这是 no-loop 属性的增强，因为这些变化现在不仅仅是规则自身的变化。
4. salience
   默认值：0
   type：Integer
   任何规则都有一个默认为 0 的 salience 属性，该属性可以为 0，正数和负数。salience 表示规则的优先级，值越大其在激活队列中的优先级越高。Drools 支持使用动态的 salience，可以使用一个包含动态约束变量的表达式来表示。

```
rule "Fire in rank order 1,2,.."
       salience( -$rank )
    when
       Element( $rank : rank,... )
    then
       ...
end
```

5. agenda-group
   默认值：MAIN
   type：String
   agenda-group 允许用户将 Agenda 分割成多个部分以提供更多的运行控制。
6. auto-focus
   默认值：false
   type：Boolean
   当一个规则被激活时 auto-focus 为 true，而且该规则的 agenda-group 还没有 focus，当该 agenda-groupfocus 时，允许该规则潜在命中。

7. activation-group
   默认值：N/A
   type：String
   属于同一个 activation-group 的规则会进行唯一命中。也就是说同一个 activation-group 中的规则，只要有一个命中，其他的规则都会被取消激活状态，这样这些规则就不会被命

中。

8. dialect
   默认值: as specified by the package
   type: String
   dialect 用于指明规则中使用的代码的语言种类，目前支持两种语言，"java"或"mvel"。
9. date-effective
   默认值: N/A
   type: String (包含日期和时间)
   当前系统时间在 date-effective 之后，该规则才会被激活。
10. date-effective
    默认值: N/A
    type: String (包含日期和时间)
    当前系统时间在 date-effective 之后，该规则不会再被激活。
11. duration
    默认值: 无
    type: long (包含日期和时间)
    duration 用于表示一个规则在一定时间之后才会被命中，如果它还是激活状态的话。

LHS 语法

LHS 是规则的条件部分的统称，由零到多条条件元素组成。如果 LHS 为空，默认为是条件部分一直为 true。当一个新的 WorkingMemory session 创建的时候，会被激活和触发。

```
rule "no CEs"
when
    // empty
then
    ... // actions (executed once)
end

// The above rule is internally rewritten as:

rule "eval(true)"
when
    eval( true )
then
    ... // actions (executed once)
end
```

LHS 中的条件元素基于一个或多个模式，最常用的条件元素是 and。当然如果 LHS 中有多个不互相连接的模式时，默认使用隐式的 and。

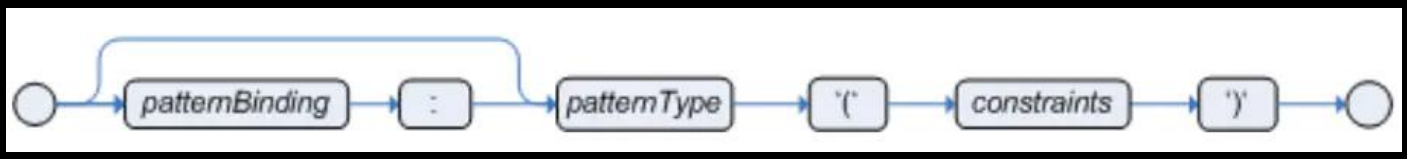
```
rule "2 unconnected patterns"
when
    Pattern1()
    Pattern2()
then
    ... // actions
end

// The above rule is internally rewritten as:
```

```
rule "2 and connected patterns"
when
    Pattern1()
    and Pattern2()
then
    ... // actions
end
```

模式

模式是最终要的条件元素，它可以隐式地匹配所有插入到 WorkingMemory 中的所有 fact。一个模式具有 0 个或多个约束条件和一个可选的模式组合。模式的结构图如下所示：



下面给出一个最简单的模式的例子

```
Person()
```

这里的类型为 Person，该模式意味着将匹配 WorkingMemory 中的所有 Person 对象。该类型不需要是一个真实 fact 对象的类。模式可以指向超类甚至是接口，这样可以匹配多个不同类的 facts。

```
Object() // matches all objects in the working memory
```

模式的括号中条件定义了模式在何种条件下满足。如下所示：

```
Person( age == 100 )
```

为了引用匹配的对象，可以使用一个模式绑定参数如：$p。

```
rule ...
when
    $p : Person()
then
    System.out.println( "Person " + $p );
end
```

$符号是非强制性的，只是用于在复杂的规则中方便标识，将其与变量及域区分开来。

约束

约束是一个返回 true 或 false 的表达式，如下例所示：

```
Person( 5 < 6 )  // just an example, as constraints like this would be useless
in a real pattern
```

约束本质上是一个与 Java 表达式稍微有点不同的表达式，例如 equals()等价于==。接下来我们深入理解一下。

Java Beans 属性获取。
任何一个 bean 的属性都可以被直接使用，bean 属性的获取也可以使用标准的 Java bean getter: getMyProperty() or isMyProperty()。例如：

```
//use directly
Person( age == 50 )

// this is the same as:
Person( getAge() == 50 )
```

Drools 还支持嵌套的属性获取方式，如：

```
//use directly
Person( address.houseNumber == 50 )

// this is the same as:
Person( getAddress().getHouseNumber() == 50 )
```

当然，约束中的条件表达式是支持 Java 表达式的，下面几个例子都是正确的：

```
Person( age == 50 )
Person( age > 100 && ( age % 10 == 0 ) )
Person( Math.round( weight / ( height * height ) ) < 25.0 )
```

逗号分隔符 AND

```
// Person is at least 50 and weighs at least 80 kg
Person( age > 50, weight > 80 )

// Person is at least 50, weighs at least 80 kg and is taller than 2 meter.
Person( age > 50, weight > 80, height > 2 )
```

逗号运算符不能出现在复合的约束表达式中，如

```
// Do NOT do this: compile error
Person( ( age > 50, weight > 80 ) || height > 2 )

// Use this instead
Person( ( age > 50 && weight > 80 ) || height > 2 )
```

绑定变量

属性值可以绑定到一个变量中：

```
// 2 persons of the same age
Person( $firstAge : age ) // binding
Person( age == $firstAge ) // constraint expression
```

分组访问嵌套对象属性

当处理嵌套对象时，往往需要将其转换成子类，可以通过使用#符号来完成。

```
Person( name == "mark", address#LongAddress.country == "uk" )
```

在该例子中将 Address 转换成 LongAddress。如果类型转换失败，该值会被认为是 false。当然，类型转换也支持全称限定名称。

```
Person( name == "mark", address#org.domain.LongAddress.country == "uk" )
```

另外，Drools 同样支持 instanceof 操作。

```
Person( name == "mark", address instanceof LongAddress, address.country == "uk" )
```

特殊文字支持

除了正常的 Java 文字，Drools 还支持以下特殊的文字：

日期文字。