第 3 章

知识表示

在人工智能发展的过程中，人们逐渐意识到知识的重要性，Feigenbaum 在 1977 年召开的第五届国际人工智能会议上提出知识工程的概念，使得知识成为人工智能研究中关键的一个环节，此后多个专家系统的成功应用验证了知识的重要作用，2012 年 Google 公司提出的知识图谱更是将知识的作用提升到一个新的高度。而如何让计算机认识知识成为人工智能中首先要解决的问题，这正是知识表示(knowledge representation)要研究的内容。

3.1 概述

知识表示研究的对象是知识，目标是使用某种形式将有关问题的知识存入计算机以便处理。由于知识的多样性与现有计算机存储模式存在的固有矛盾，知识表示一直是一个充满挑战而又较为活跃的领域。

3.1.1 知识与知识表示

什么是知识？从认识论的角度来看，知识就是人类认识自然界(包括社会和人)的精神产物，是人类进行智能活动的基础。计算机所处理的知识按其作用可大致分为三类：

(1) 描述性知识(descriptive knowledge)。表示对象及概念的特征及其相互关系的知识，以及问题求解状况的知识，也称为事实性知识。

(2) 判断性知识(judgmental knowledge)。表示与领域有关的问题求解知识(如推理规则等)，也称为启发性知识。

(3) 过程性知识(procedural knowledge)。表示问题求解的控制策略，即如何应用判断性知识进行推理的知识。

按照作用的层次，知识还可以分成以下两类：

(1) 对象级知识(object-level knowledge)。直接描述有关领域对象的知识，或称为领域

相关的知识。

(2) 元级知识(meta-level knowledge)。描述对象级知识的知识,如关于领域知识的内容、特征、应用范围、可信程度的知识以及如何运用这些知识的知识,也称为关于知识的知识。

所谓表示就是为描述世界所做的一组约定,是把知识符号化的过程。知识的表示与知识的获取、管理、处理、解释等有直接关系。对于问题能否求解以及问题求解的效率具有重要影响。一个恰当的知识表示可以使复杂的问题迎刃而解。一般而言对知识表示有如下要求:

(1) 表示能力。能够将问题求解所需的知识正确且有效表达出来。

(2) 可理解性。所表达的知识简单明了、易于理解。

(3) 可访问性。能够有效利用所表达的知识。

(4) 可扩充性。能够方便、灵活对知识进行扩充。

知识表示的方法按其表示的特征可分为两类,叙述性(declarative)表示和过程性(procedural)表示。叙述性表示将知识与控制分开,把知识的使用方法即控制部分留给计算机程序。这种方法严密性强、易于模块化,具有推理的完备性,但推理的效率比较低。过程性表示把知识与控制结合起来,因此推理效率比较高。通常认为叙述性表示方法具有固定的表示形式,包括基于逻辑的表示方法、图示法、结构化表示方法等。而过程性表示并无一定之规,所有知识都隐含在程序中。

3.1.2 知识表示的方法

知识表示方法既是一个独立的课题,又与推理的方法有着密切的关系。下面对几种主要的知识表示方法及其应用作简单介绍。

1. 逻辑(logic)

逻辑表示法采用一阶谓词逻辑表示知识,是最早的一种叙述性知识表示方法。它的推理机制可以采用归结原理,主要用于自动定理证明。

2. 语义网络(semantic network)

语义网络是 Quillian 等在 1968 年提出的,最初用于描述英语的词义。它采用结点和结点之间的弧表示对象、概念及其相互之间的关系。目前语义网络已广泛用于基于知识的系统。在专家系统中,常与产生式规则共同表示知识。

3. 产生式规则(production rule)

产生式规则把知识表示成“模式-动作”对,表示方式自然简洁。它的推理机制以演绎推理为基础,推理系统也称为产生式系统,目前已是专家系统中使用非常广泛的一种表示方法,一般将这种系统称为基于规则的系统。

4. 框架(frame)

框架理论是 Minsky 于 1974 年提出的,将知识表示成高度模块化的结构。框架是把关

于一个对象或概念的所有信息和知识都存储在一起的一种数据结构。框架的层次结构可以表示对象之间的相互关系，用框架表示知识的系统称为基于框架的系统。

5. 状态空间(state space)

状态空间表示法把求解的问题表示成问题状态、操作、约束、初始状态和目标状态。状态空间就是所有可能的状态的集合。求解一个问题就是从初始状态出发，不断应用可应用的操作，在满足约束的条件下达到目标状态。问题求解过程可以看成是问题状态在状态空间的移动。

6. 脚本(script)

脚本用于描述固定的事件序列。它的结构类似于框架，一个脚本也由一组槽组成。与框架不同的是，脚本更强调事件之间的因果关系。脚本中描述的事件形成了一个巨大的因果链，链的开始是一组进入条件，它使脚本中的第一个事件得以发生。链的末尾是一组结果，它使后继事件得以发生。框架是一种通用的结构，脚本则对某些专门知识更为有效。

7. 本体(ontology)

本体的概念最初用于哲学中，表示客观存在的一个系统的解释和说明。在人工智能学科中，本体是一个规范说明，可定义为被共享的概念化的一个形式化显式规范说明。一个本体定义了组成主题领域词汇的基本术语与关系以及这些术语和关系的规则。得益于其严格的规定和强大的表示能力，本体是近年来知识表示领域中的主要研究方向。

8. 概念从属(conceptual dependency)

概念从属是表示自然语言语义的一种理论，它的特点是便于根据语句进行推理，而且与语句本身所用的语言无关。概念从属表示的单元并不对应于语句中的单词，而是能组合成词义的概念单元。

9. Petri 网

Petri 网是由 Petri 在 20 世纪 60 年代提出的。利用 Petri 网可以模拟逻辑运算、语义网、框架、与/或图、状态空间与规则等多种功能，因此也可作为一种通用的知识表示形式。

10. 知识图谱(knowledge graph)

相对于上述知识表示方法，知识图谱可以称为“最新的”知识表示方法，是 2012 年才由 Google 公司提出的，但究其本质知识图谱关于知识表示的核心部分还是本体。

在上述知识表示方法中，逻辑与产生式规则属于基于逻辑的表示方法；语义网络、Petri 网与状态空间属于图示法；其他方法都属于结构化表示方法。在实际使用中，需要根据具体问题选择合适的知识表示方法或者它们的组合。

本章剩余部分对常用的知识表示方法如逻辑表示、产生式规则表示、语义网络、框架、脚本及本体进行介绍，最后再介绍“最新的”知识图谱。而状态空间表示法将在 4.1 节“状态空间搜索”中详细介绍。

3.2 逻辑表示法

本节所说的逻辑特指一阶谓词逻辑(first order predicate logic)。逻辑表示法是最早使用的一种知识表示方法,它具有简单、自然、精确、灵活、模块化的优点。逻辑表示法的推理系统主要采用归结原理,这种推理方法严格、完备、通用,在自动定理证明等应用中取得了成功。逻辑表示法的缺点是难于表达过程性知识和启发性知识,不易组织推理,推理方法在事实较多时易于产生组合爆炸,且不易实现非单调和不确定性的推理。本节主要介绍谓词逻辑的表示,相关推理如归结原理将在第5章"基于符号的推理"中介绍。

3.2.1 一阶谓词逻辑

谓词逻辑的合法表达式也称为合式公式(well-formed formula)或者谓词公式(也可简称为公式)。合式公式是由原子公式、连接词和量词组成的,下面分别加以介绍。

1. 原子公式

首先规定常量、变量、谓词和函数的表示方式(与Prolog语言中的规定不一致):常量是字符或数字序列,首字母为大写字母或数字;变量是小写字母与数字的序列,首字母为小写字母;谓词是大写字母序列;函数是字符或数字序列,首字母为大写字母。谓词用来表示真假,而函数用来表示一个对象与另一个对象之间的关系。

原子公式是最基本的合式公式,它由谓词、括号和括号中的项组成,其中项可以是常量、变量和函数。例如"盒子在桌子上"这一事实可以用原子公式表示为:

```
ON(Box,Table)
```

其中,Box和Table是常量,表示个体。ON是谓词,表示Box在Table上面。再如"张比王的哥哥高"这一事实可以表示为:

```
TALLER(Zhang,Brother(Wang))
```

其中,Brother是函数,Brother(Wang)代表Wang的哥哥。另外,项也可以是变量,用于表示不确定的个体。

通常,一个事实可以用多种形式的原子公式表示。例如"盒子是蓝色的"这一事实可以表示成以下三种形式:

```
BLUE(Box); COLOR(Box,Blue); VALUE(Color,Box,Blue)
```

对一个合式公式,可以规定其中的谓词、常量和函数与论域中的关系、实体和函数之间的对应关系,从而建立起一个解释(在第5章"基于符号的推理"中将详细介绍解释的定义)。一旦对一个合式公式定义了一个解释,则当该公式在论域对应的语句为真时,该公式就有值"真"(T);当该公式在论域对应的语句为假时,该公式就有值"假"(F)。例如当现实世界的盒子在桌子上时,公式ON(Box,Table)就为真,而公式ON(Room,Table)则为假。

2. 连接词

连接词用来组合原子公式以形成较复杂的合式公式。

(1) 合取连接词“∧”表示逻辑与。设 P、Q 为合式公式,则 $P \wedge Q$ 表示 P 与 Q 的合取,P 与 Q 称为合取项。当合取项 P 与 Q 均为真时,$P \wedge Q$ 取值“真”,否则取值“假”。例如“盒子在窗户旁边的桌子上”可表示为:

```
ON(Box,Table) ∧ BY(Table,Window)
```

(2) 析取连接词“∨”表示逻辑或。设 P、Q 为合式公式,则 $P \vee Q$ 表示 P 与 Q 的析取,P 与 Q 为析取项。当析取项 P 和 Q 至少有一个为真时,$P \vee Q$ 取值“真”,否则取值“假”。例如“球是红色的或蓝色的”可表示为:

```
RED(Ball) ∨ BLUE(Ball)
```

(3) 连接词“⇒”表示蕴涵。设 P、Q 为合式公式,则 $P \Rightarrow Q$ 称作蕴涵式,P 称为蕴涵式的前项,Q 称为后项。蕴涵式 $P \Rightarrow Q$ 常用于表示 if P then Q。如果后项取值为真或前项取值为假,则蕴涵式的值为真,否则为假。蕴涵式及其前项和后项的真值表如表 3.1 所示。注意,只有前项为真而后项为假时,蕴涵式才为假,其余情况蕴涵式均为真。

表 3.1 蕴涵式的真值表

P	Q	$P \Rightarrow Q$	P	Q	$P \Rightarrow Q$
T	T	T	F	T	T
T	F	F	F	F	T

(4) 符号“¬”表示否定。尽管它不是用来连接两个合式公式的,但也可以称为连接词。设 P 为合式公式,则 $\neg P$ 称为合式公式 P 的否定。当 P 为真时,$\neg P$ 取值“假”,当 P 为假时,$\neg P$ 取值“真”。一个原子公式和一个原子公式的否定都称为文字。

如果我们把合式公式限制在上面所举例子的形式,即不出现变量项,则谓词逻辑的这个子集称为命题逻辑。

3. 量化

在合式公式中出现变量的时候,前面可以加量词以说明变量的范围,这种说明称为量化。

谓词逻辑中有两个量词,全称量词和存在量词。设 $P(x)$为合式公式,如果在 $P(x)$前加以全称量词 $\forall x$(读作对所有的 x),只有在某个解释下对论域中实体 x 的所有可能值 $P(x)$都为真时,公式$(\forall x)P(x)$在该解释下才取值“真”。例如“所有的大象都是灰色的”这一事实可表示为:

```
(∀x)[ELEPHANT(x) ⇒ COLOR(x,Gray)]
```

其中 x 是被量化的变量。对被量化的公式,要注意量词的辖域,即量词的作用范围。上式中全称量词 $\forall x$ 的辖域为方括号内的范围。

设 $P(x)$为合式公式,如果在 $P(x)$前加以存在量词 $\exists x$(读作至少存在一个 x),只要在某个解释下论域中实体 x 至少有一个值使 $P(x)$为真,公式$(\exists x)P(x)$在该解释下就取值“真”。例如“有一件东西在桌子上”这一事实可表示为:

(∃x)ON(x,Table)

合式公式中经过量化的变量称为约束变量,否则称为自由变量。在一阶谓词逻辑中,只允许对变量进行量化,不允许对谓词和函数进行量化。

综上所述,合式公式可以定义为以下三类:

(1) 原子公式是合式公式;

(2) 原子公式通过连接词得到的公式称为逻辑公式,逻辑公式是合式公式;

(3) 逻辑公式中的变量被全称量词或存在量词量化后得到的公式是合式公式。

通常将不含自由变量的合式公式称为闭合式公式,而不含任何变量的合式公式称为基合式公式。

4. 合式公式实例

对于复杂的合式公式,可以用各种括号作为合式公式组的分界。例如,"张送给屋里的人每人一件礼物"可以表示为:

(∀y){[IN(y,Room)∧Human(y)]⇒(∃x)[GIVE(Zhang,x,y)∧Present(x)]}

表达方式可以是多种多样的,应根据具体问题的要求采取适当的形式。

5. 合式公式的性质

如果两个合式公式的真值表不论它们的解释如何都是相同的,则称这两个合式公式等价,用≡表示。用真值表可以很容易证明下列公式的等价性。

1) $\neg(\neg X)\equiv X$

2) $X_1\Rightarrow X_2\equiv\neg X_1\vee X_2$

3) 德·摩根定律

$\neg(X_1\wedge X_2)\equiv\neg X_1\vee\neg X_2$

$\neg(X_1\vee X_2)\equiv\neg X_1\wedge\neg X_2$

4) 分配律

$X_1\wedge(X_2\vee X_3)\equiv(X_1\wedge X_2)\vee(X_1\wedge X_3)$

$X_1\vee(X_2\wedge X_3)\equiv(X_1\vee X_2)\wedge(X_1\vee X_3)$

5) 交换律

$X_1\wedge X_2\equiv X_2\wedge X_1$

$X_1\vee X_2\equiv X_2\vee X_1$

6) 结合律

$(X_1\wedge X_2)\wedge X_3\equiv X_1\wedge(X_2\wedge X_3)$

$(X_1\vee X_2)\vee X_3\equiv X_1\vee(X_2\vee X_3)$

7) 逆否律

$X_1\Rightarrow X_2\equiv\neg X_2\Rightarrow\neg X_1$

根据量词的含义,也可建立下列公式的等价性:

(1) $\neg(\exists x)P(x)\equiv(\forall x)[\neg P(x)]$

(2) $\neg(\forall x)P(x)\equiv(\exists x)[\neg P(x)]$

(3) $(\forall x)[P(x) \wedge Q(x)] \equiv (\forall x)P(x) \wedge (\forall x)Q(x)$

(4) $(\exists x)[P(x) \vee Q(x)] \equiv (\exists x)P(x) \vee (\exists x)Q(x)$

(5) $(\forall x)P(x) \equiv (\forall y)P(y)$；$(\exists x)P(x) \equiv (\exists y)P(y)$

注意，把(3)和(4)的连接词对换之后等价并不成立。

从(5)可以看出，合式公式中的约束变量用哪一个符号是不重要的，它可以用任意一个不出现在公式中的其他变量符号代替。

3.2.2 谓词逻辑用于知识表示

一阶谓词逻辑为表示现实世界的知识提供了强有力的机制，在很多领域可以得到直接的应用。

1. 形式化的领域

在某些领域中，知识或信息是用对象和它们的属性以及它们之间的关系表示的。这些领域称为形式化的领域，如数学、数据库中的信息等。这种领域中知识和信息可直接表示成一阶谓词逻辑的形式。

假设有一个关系型数据库存储了相邻的国家，如表 3.2 所示。此时可以使用一个谓词来描述这种相邻关系，从而得到所有的关于相邻关系的原子公式：NEIGHBOR(America, Mexico)；NEIGHBOR(America, Canada)；NEIGHBOR(Mexico, Guatemala)；NEIGHBOR(China, NorthKorea)；NEIGHBOR(China, Russia)；NEIGHBOR(China, VietNam)。

表 3.2 相邻国家表示例

国 家	相 邻 国 家	国 家	相 邻 国 家
美国	墨西哥	中国	朝鲜
美国	加拿大	中国	俄罗斯
墨西哥	危地马拉	中国	越南

2. 非形式化的领域

上面所讲的关系数据库本身就是一个谓词逻辑理论的模型，因此数据库中的事实与现实世界的映射是直接的。但对大多数领域来讲，这种映射不是很显然的。例如有语句：

John gives a book to Mary(John 给 Mary 一本书)

很显然不能把这个命题作为一个整体，这句话中有意义的基本概念是 John、Mary、book 和 give。很自然就可以把 John、Mary、book 视为对象，而把 give 看作描述它们之间的关系。因此上述语句可表示为：

$(\exists x)[GIVE(John, x, Mary) \wedge BOOK(x)]$

还可以加入原子公式

HUMAN(John)
HUMAN(Mary)

表示 John 和 Mary 都是人。

再如想要表达一个定理“过任意不同两点恰有一条直线”：

$$(\forall x)(\forall y)\{[P(x)\wedge P(y)\wedge -E(x,y)]\Rightarrow [(\exists z)(L(z)\wedge F(x,y,z)\wedge(\forall w)(L(w)\wedge F(x,y,w)\Rightarrow E(z,w)))]\}$$

其中 $P(x)$表示 x 是一个点，$L(z)$表示 z 是一条直线，$F(x,y,z)$表示直线 z 过点 x 和点 y，$E(x,y)$表示 x 和 y 相同。

利用逻辑表示法将知识表示出来之后，就可以利用一些推理的方法进行推理从而得到想要的结果。

3.3 产生式规则表示法

规则的表示具有固有的模块特性，易于实现解释功能，其推理机制接近于人类的思维方式，因此获得了广泛的应用。本节介绍产生式规则的表示方法与基于规则的系统的基本工作方式，对规则推理的深入讨论在后续各章介绍。

1. 产生式系统的构成

产生式系统是人工智能中经常采用的一种计算系统，它的基本要素包括综合数据库(Global Database)、产生式规则(Production Rules)和控制系统(Control System)。

综合数据库是产生式系统所使用的主要数据结构，用来描述问题的状态。在问题求解中，它记录了已知的事实、推理的中间结果和最终结论。

产生式规则的作用是对综合数据库进行操作，使综合数据库发生变化。产生式规则的一般形式是：

if <前提> then <动作(或<结论>)

规则的前提部分是能和综合数据库匹配的任何模式，通常允许包含一些变量，这些变量在匹配过程中可能以不同的形式被约束。一旦匹配成功，则执行规则的<动作>部分。<动作>部分可以是使用约束变量的任一过程，也可以得出某一<结论>。

控制系统的功能包括：

(1) 根据综合数据库的当前状态查找可用的规则；

(2) 在可用规则集中选择一条当前应用的规则；

(3) 执行选出的规则，规则作用于综合数据库，使之发生变化。

控制系统再根据综合数据库新的状态选择一条规则作用于综合数据库，形成一个“识别-动作”循环，直至综合数据库的状态满足了结束条件或无可用规则为止。

产生式系统与一般计算系统相比具有以下特点：综合数据库是全局性的，可被所有的规则访问；规则之间不能互相调用，它们之间的联系只能通过综合数据库进行。产生式系统的特点使得对综合数据库、产生式规则和控制系统的修改可以独立进行，因此适合于人工智能的应用。

产生式系统的基本工作过程可表示为：

```
过程 PRODUCTION
    DATA = 初始数据库
    until DATA 满足结束条件 do
        在规则集合中,选择一条可应用于 DATA 的规则 R
        DATA = R 应用到 DATA 得到的结果
    end until
end 过程
```

在上述过程中,DATA 指综合数据库,选择规则 R 可以有不同的方法,由此对应不同的控制系统。

目前,产生式系统与初期的系统相比,已有了很大发展,广泛应用于基于知识的系统中,一般称为基于规则的系统。下面介绍两个用产生式系统表示问题的例子。

2. 八数码游戏

在一个有三行三列共九个格子的棋盘上,在其中八个格子上放着写有数字的方块。允许的动作是把空格旁边的一个数字方块移到空格处,最终目标是将初始的布局(状态)改变成目标布局(状态)。图 3.1 给出了一个初始布局和一个目标布局。要解决的问题就是确定一个方块移动的序列,使初始布局改变为目标布局。

要用产生式系统解决这个问题,需要规定综合数据库、一组产生式规则和控制策略。把一个问题的描述转化为产生式系统的三个部分,正是知识表示需要完成的任务。

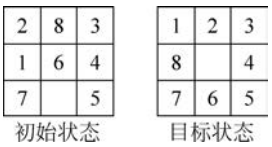

图 3.1 八数码游戏例子

首先,用一个 3×3 的矩阵来表示棋盘的布局,称为状态描述。图 3.1 中的初始布局和目标布局可表示为:

$$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 7 & 0 & 5 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

初始数据库就是问题的初始状态描述。每移动一个数字方块,状态描述就变化一次,初始数据库添加每次变化后产生的状态描述构成了综合数据库。

然后需要规定产生式规则。为简化起见,可以把移动数字方块表述为移动空格。这样,移动空格的不同方向就构成了不同的产生式规则。由于每个空格移动方向都有先决条件,例如空格上移的先决条件就是空格不在最上边的一行,因此产生式规则可表示为:

(1) if 空格不在最上一行 then 空格上移;

(2) if 空格不在最下一行 then 空格下移;

(3) if 空格不在最左一列 then 空格左移;

(4) if 空格不在最右一列 then 空格右移。

最后确定控制系统。控制系统的目标是不断选择合适的产生式规则作用于综合数据

库，改变综合数据库的状态，直至产生了目标描述为止。这时，施行过的产生式规则的序列就是问题的解。显然，就八数码游戏而言，解不是唯一的。因此我们还可以附加一些约束条件，例如求一个移动方块次数最少的解。一般来讲，可以为每个规则应用规定一个耗散值(cost)，一个解的耗散值就是解的每步规则应用的耗散值之和。问题可规定为要求耗散值最小的解。

3. 传教士和野人问题

河的左岸有 N 个传教士和 N 个野人准备渡河，河岸有一条渡船，每次最多可供 K 人渡河。为安全起见，在任何时刻河两岸以及船上的野人数目不得超过传教士的数目。求渡河的方案。

下面将 $N=3, K=2$ 的情况表示成产生式系统要求的形式。由于河的两边传教士的总数、野人的总数和船的总数是固定的，因此只需表示河某岸边的状态而非两岸的状态。为此设左岸的传教士数目为 m，左岸的野人数目为 c，左岸的渡船数目为 b(b 取 0 或 1)。渡河过程的状态可以用三元组(m，c，b)表示。问题的初始状态和目标状态可表示为(3，3，1)和(0，0，0)。

对于 $N=3$ 的情况，状态空间总数为 $4\times4\times2=32$。但是为了满足约束条件，只有满足下列条件之一时，状态才是合法的

(1) $m=0$；

(2) $m=3$；

(3) $m=c$。

系统的综合数据库记录上述状态描述，而产生式规则应对应于摆渡操作。设 P_{mc} 表示将 m 个传教士和 c 个野人从左岸送到右岸的规则，Q_{mc} 表示将 m 个传教士和 c 个野人从右岸送到左岸的规则，则规则集合为

$$R=\{P_{01},P_{10},P_{11},P_{02},P_{20},Q_{01},Q_{10},Q_{11},Q_{02},Q_{20}\}$$

考虑到约束条件，规则集应如表 3.3 所示。

表 3.3 传教士和野人问题规则表

规 则	条 件	动 作
P_{01}	$b=1,(m=0)$或$(m=3),c\geqslant1$	$b=0,c=c-1$
P_{10}	$b=1,(m=1,c=1)$或$(m=3,c=2)$	$b=0,m=m-1$
P_{11}	$b=1,m=c,c\geqslant1$	$b=0,m=m-1,c=c-1$
P_{02}	$b=1,(m=0)$或$(m=3),c\geqslant2$	$b=0,c=c-2$
P_{20}	$b=1,(m=3,c=1)$或$(m=2,c=2)$	$b=0,m=m-2$
Q_{01}	$b=0,(m=0)$或$(m=3),c\leqslant2$	$b=1,c=c+1$
Q_{10}	$b=0,(m=2,c=2)$或$(m=0,c=1)$	$b=1,m=m+1$
Q_{11}	$b=0,m=c,c\leqslant2$	$b=1,m=m+1,c=c+1$
Q_{02}	$b=0,(m=0)$或$(m=3),c\leqslant1$	$b=1,c=c+2$
Q_{20}	$b=0,(m=0,c=2)$或$(m=1,c=1)$	$b=1,m=m+2$

建立了综合数据库和产生式规则后，就可以选择合适的控制策略进行求解。

控制系统是决定产生式系统工作效率的关键问题，其中规则的选择和应用涉及搜索技术和推理技术，这将在后续章节详细讲述。

3.4 语义网络表示法

3.4.1 语义网络的结构

语义网络是一种有向图，它由结点和结点之间的弧组成。结点用于表示物理实体、概念或状态，弧表示它们之间的相互关系。由于语义网络中可能会用结点表示谓词，因此语义网络中所有常量和谓词都用大写字符序列表示。

例如，图 3.2 是描述“我的椅子”(MY-CHAIR)的一个语义网络。其中，结点 MY-CHAIR 以上的部分表示“我的椅子是一个椅子”“椅子是一种家具”“座部是椅子的一部分”。结点 MY-CHAIR 以左的部分表示“我的椅子的所有者是我”“我是一个人”。结点 MY-CHAIR 以右的部分表示“我的椅子的颜色是棕褐色”“棕褐色是一种褐色”。结点 MY-CHAIR 以下的部分表示“我的椅子的覆盖物是皮革”。

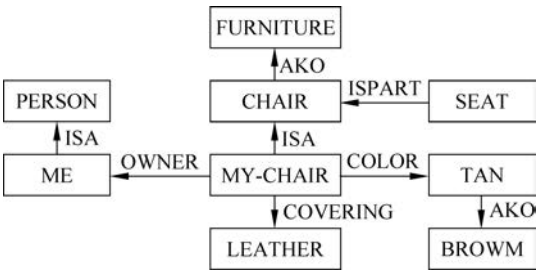

图 3.2 表示“我的椅子”的语义网络

图中 ISA 和 AKO 是语义网络中常用的关系。ISA 表示某一个体是某一集合的一个元素，读为“是……的一个实例”，如“我的椅子是一个椅子”。AKO 是 A-KIND-OF 的缩写，表示一个集合是另一个集合的子集合，如“椅子是一种家具”。有的语义网络可以用 ARE 代替 AKO。图中的关系 ISPART、OWNER、COLOR 与 COVERING 分别表示“是一部分”“所有者是”“颜色是”与“覆盖物是”。

一般可将两个对象的关系划分为四类：

(1) 从属关系，如 ISA 与 AKO 分别表示一个对象是另一个对象的实例或类型。

(2) 包含关系，如 ISPART 表示一个对象是另一个对象的一部分。

(3) 属性关系，如 OWNER 和 COLOR 分别表示一个对象的所有者和颜色属性。

(4) 位置和时间关系，如使用 BEFORE 和 LOCATE 分别表示之前以及位于的关系。

语义网络表示一元关系和二元关系都非常方便，例如“JOHN 是人”表示成一元谓词是 PERSON(JOHN)，用语义网络表示则可以表示为 JOHN $\xrightarrow{\text{ISA}}$ PERSON，构造一个 PERSON 结点表示一元谓词 PERSON。此时 ISA(JOHN,PERSON)与 PERSON(JOHN)等价。而“球的颜色是红色的”表示成二元谓词是 COLOR(BALL,RED)，用语义网络则可

表示为 BALL $\xrightarrow{\text{COLOR}}$RED。那么如何表示多元关系？语义网络本质上只能表示二元关系，因此需要对多元关系进行分解，分解成多个二元甚至一元关系，然后用语义网络表示。

回想 3.2.2 节“谓词逻辑用于知识表示”John gives a book to Mary 的谓词逻辑表示为：

$(\exists x)[GIVE(John,x,Mary) \wedge BOOK(x)]$

可以拆分为多个谓词，对应的语义网络见图 3.3。

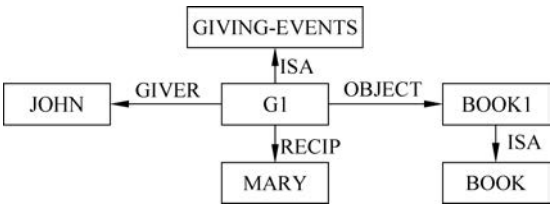

图 3.3 “John 给 Mary 一本书”的语义网络表示

3.4.2 连接词的表示

在语义网络中，如果一个结点连有多个弧，在不加说明的情况下，各个弧之间是合取关系。为了表示析取关系，可以用封闭的虚线作为析取界线并标注 DIS。例如表达式 ISA(A,B) ∨ AKO(B,C)可表示成图 3.4(a)的形式。对于嵌套的合取与析取关系，也可以用嵌套的封闭虚线表示，在内层的合取封闭虚线应注明 CONJ。例如语句 John is a programmer or Mary is a lawyer(John 是一个程序员或 Mary 是一个律师)的语义网络如图 3.4(b)所示。关系 PROFESSION 表示该职业的具体职业内容，关系 WORKER 表示该职业的工作者。

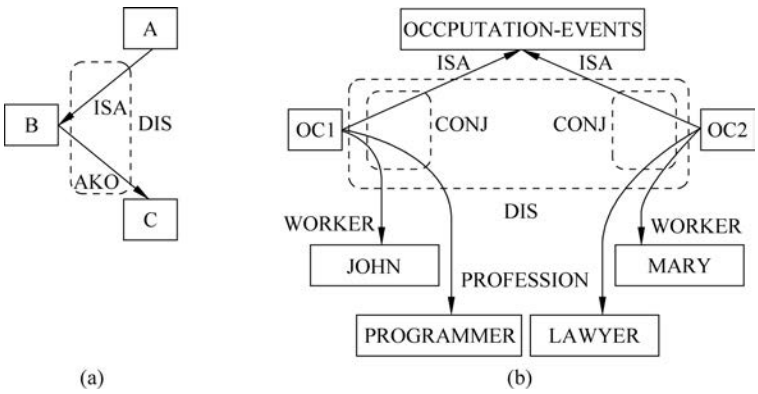

图 3.4 语义网络表示合取与析取示例

对于非的关系，可以用带有 NEG 的封闭虚线表示。例如表达式 ¬[ISA(A,B) ∧ AKO(B,C)]可以用图 3.5(a)所示的语义网络表示。类似，蕴涵关系可以用一对连接在一起的封闭虚线表示，其中前项标以 ANTE，后项标以 CONSE。例如语句 If X is a person then X has a father.(X 如果是人就必然有父亲)可以用图 3.5(b)所示的语义网络表示。规定了上述表示，就可以在语义网络中进行推理。

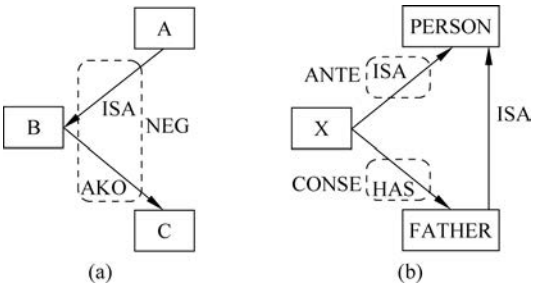

图 3.5 语义网络表示否定与蕴涵示例

3.4.3 继承性

所谓继承性是指对某一集合的特征的描述也适用于该集合的子集合或该集合的个体的描述。在语义网络中,继承性是通过表示层次关系的 ISA 和表示集合关系的 AKO 两个关系实现的。由于关系 ISA 和 AKO 具有传递性所以能够表示继承。在图 3.6 所示的关于动物分类的语义网络中,“吸入(uptake)氧气(oxygen)”是动物(animals)的特征,该特征可由动物的子类哺乳动物(mammals)及哺乳动物的子类大象(elephants)继承。由于大象 Clyde 是大象的一个个体,因此它也可以通过继承得到该特性。同样,大象 Clyde 还可以通过继承得到哺乳动物的特性“血液温度(blood-temp)暖和(warm)”。还可以通过继承得到大象的特性“颜色(color)为灰色(gray)”“纹理(texture)为多皱(wrinkled)”以及“喜欢的食物(favorite-food)是花生(peanuts)”等。

利用继承性可以大大减少存储量。虽然不使用继承性而通过规则推理也可以达到减少存储量的效果,但是利用继承性显然具有较高的效率。

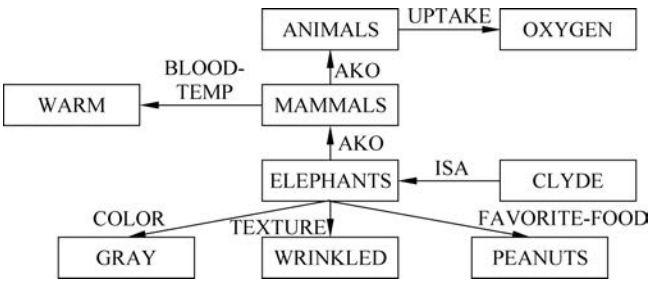

图 3.6 关于动物分类的语义网络

3.5 框架与脚本表示法

3.5.1 框架表示法

框架是一种通用的数据结构,适于表达多种类型的知识,被广泛应用于专家系统的知识表示。

框架通常用于描述具有固定形式的对象。一个框架(frame)由一组槽(slots)组成,每个

槽表示对象的一个属性，槽的值(fillers)就是对象的属性值。一个槽可以由若干个侧面(facet)组成，每个侧面可以有一个或多个值(values)。一个汽车框架如下所示：

```
name: 汽车
super-class: 运载工具
sub-class: 小轿车,SUV,客车,小货车
number-of-tyre:
    value-class: 整数
    default: 4
    value: 未知
length:
    value-class: 浮点数
    unit: 米
    value: 未知
……
```

其中，name 是框架名，槽 super-class 和 sub-class 分别表示该对象的超类和子类，各个框架之间可通过超类子类关系和成员关系形成一个层次结构。number-of-tyre 和 length 都是槽，划分为多个侧面。

框架一般表示为如下形式：

```
name: 框架名
super-class: 超类
sub-class: 子类
槽 1:
    侧面 11: 侧面值 11,侧面值 12
    侧面 12: 侧面值 13,侧面值 14,侧面值 15
    侧面 13: 侧面值 16
槽 2:
    侧面 21: 侧面值 21
    侧面 22: 侧面值 22,侧面值 23
……
```

一个框架可以有多个槽，每个槽可以有多个侧面，每个侧面又可以有一个或多个侧面值。侧面值可以是各种类型的数据，包括数值、字符串等。常用的侧面有：

```
Value: 属性的值
type(或 value-class): 属性值的类型
range: 属性值的取值范围
default: 属性值的默认值
```

框架的槽还可以附加过程，称为过程附件(procedural attachment)，包括子程序和某种推理过程，这种过程也以侧面的形式表示。附加过程根据其启动方式可分为两类。一类是自动触发的过程，称为“精灵”(demon)，这类过程一直监视系统的状态，一旦满足条件便自动开始执行。另一类是受到调用时触发的过程，称为“服务者”(servant)，用以完成特定的动作和计算。

常用的自动触发型过程有三种类型。第一种是 if-needed 侧面，在需要槽值而该槽无值且无默认值可用时，自动调用此侧面的过程，并将求值结果作为该槽的值。第二种是 if-

added 侧面，当槽增加一个值后，自动执行此侧面的过程。第三种是 if-removed 侧面，当删除一个槽值后，自动执行此侧面的过程。

框架用于知识表示有如下优点：

(1) 框架可为实体、属性、关系和默认值等提供显式的表示，其中提供默认值特别重要，它相当于人类根据以往的经验对情况的预测，非常适合于表示常识性知识(commonsense knowledge)。在推理过程中遇到不知道的情况，可用默认值代替，这样比较接近人类的推理。

(2) 容易附加过程信息。槽的过程附件不仅提供了附加的推理机制，还可进行矛盾检测，用于知识库的一致性维护。

(3) 框架的层次结构提供了继承特性。框架的属性及附加过程都可以从高层次的框架继承下来。与语义网络类似，应用继承性可以实现高效的推理。

3.5.2 脚本表示法

脚本(script)类似于框架结构，用于表示固定的事件序列，如去电影院看电影、超级市场购物、在饭馆就餐、到医院看病等。

脚本主要包括以下成分：

(1) 线索(track)，这个脚本所表示的普遍模式的某一变形，同一脚本的不同线索共享很多但不是全部的成分。

(2) 角色(roles)，脚本描述的事件中可能出现的人物。

(3) 进入条件(entry conditions)，脚本描述的事件能够发生的前提条件。

(4) 道具(props)，脚本描述的事件中可能出现的物体。

(5) 场景(scene)，描述实际发生的事件序列。

(6) 结果(results)，脚本所描述的事件发生后产生的结果。

下面是一个超级市场的脚本的例子。脚本中有 4 个场景，对应于在超级市场购物时通常发生的事件。

脚本名称：食品市场
线索：超级市场
角色：顾客，服务员，其他顾客
进入条件：顾客需要食品，食品市场开门
道具：购物车，货架，食品，收款台，钱
场景 1：进入市场
　　顾客进入市场，顾客移动购物车
场景 2：选取食品
　　顾客移动购物车通过货架，顾客注意到食品，顾客把食品放入购物车
场景 3：结账
　　顾客走到收款台，顾客排队等待，顾客准备钱，顾客把钱交给服务员，服务员把食品袋交给顾客
场景 4：离开市场
　　顾客离开市场
结果：顾客减少了钱，顾客有了食品，市场减少了食品，市场增加了钱

由于脚本具有人们通常使用的知识，因此可以为给定的情况提供期望的情景。利用脚本进行推理是从根据当前情况创建一个部分填充的脚本开始的。根据当前情况可以找到一

个匹配的脚本，与当前情况的匹配程度可根据脚本名称、进入条件及其他关键字计算。推理过程就是填充槽的过程。例如，已知 Joe 进入市场及 Joe 把钱交给服务员。根据已知的脚本，就可以推断出 Joe 需要食品，进入市场，把食品放入购物车，把钱交给服务员，离开市场等一系列动作，并推断出 Joe 有了食品，但减少了钱等结果。

脚本的形式比框架的形式应用范围窄，但适用于理解自然语言，特别是故事情节。

3.6 本体

本体(ontology)这一术语起源于哲学，1991 年美国 USC 大学的 Neches 等首次在知识表示中采用本体的概念，合作者 Gruber 于 1993 年明确定义“本体是概念化的一个显式规范说明”，Borst 于 1997 年在其博士论文中定义“本体是共享概念化的一个形式化规范说明”，而 1998 年 Studer 等则认为“本体是共享概念化的一个形式化显式规范说明”。可以看出对于本体的定义是逐渐递进的，根据 Studer 等的定义本体包含四层含义：

(1) 形式化(formal)，本体必须是计算机可读的。

(2) 显式(explicit)，涉及的概念化及其约束都有显式定义。

(3) 共享(share)，本体中的知识是相关领域共同认知的知识。

(4) 概念化(conceptualization)，概念化是对被描述世界的抽象与简化视图，是三元组 $<D,W,R>$，其中 D 是论域，W 是论域上的全关系，$<D,W>$称为领域空间，R 是领域空间上的概念关系的集合。简言之概念化是定义在领域空间上的某些概念关系的集合。

本体的目标是抽取相关领域的知识，提供对该领域知识的共同认知，确定该领域内的公共词汇(术语)，并从不同层次的形式化模式上给出这些词汇之间相互关系的明确定义。

3.6.1 本体的组成与分类

一个完整的本体通常应该包含概念(或类)、关系、函数、公理和实例五种基本元素。

(1) 概念，用于描述领域内的实际概念，既可以是具体事物也可以是抽象概念。

(2) 关系，用于描述领域中概念之间的关系，基本的关系包括 part-of、kind-of、instance-of、attribute-of，分别用于描述整体与部分、子类与超类、类与实例以及某概念是另一概念的属性。

(3) 函数，是一种特殊的关系，n 元函数表示某概念与其他 n 个概念存在的对应关系，如二元函数 line(x，y)表示经过点 x 和 y 的直线。所有函数都可以用关系替代，如 z = line(x，y)可以改写成 line_rs(x，y，z)，当 z 经过点 x 和 y 时关系 line_rs(x，y，z)成立。显然 n 元函数都可用某个 $n+1$ 元关系替代，反之则不一定。

(4) 公理，领域中任何条件下公理的取值都为真。

(5) 实例，属于某个概念(或类)的具体实体。

从语义上看实例表示的就是对象，而概念(或类)表示的是对象的集合，关系和函数则对应于对象元组的集合。概念的描述一般采用框架结构，包括概念的名称、与其他概念之间关系的集合以及自然语言对该概念的描述。

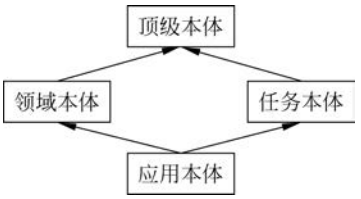

图 3.7　本体分类层次图

从不同的角度出发，存在多种对本体的分类标准。Guarino 按照本体的依赖程度将常用的本体分为下列四种类型，其层次关系如图 3.7 所示。

(1) 顶级本体，描述最普通的概念，包括知识的本质特征和基本属性。

(2) 领域本体，针对一个特定领域，描述在该领域中可重用的概念、属性、关系及其约束。

(3) 任务本体，任务本体与领域本体处于同一层次，定义通用的任务或推理活动，描述具体任务中的概念及其关系。

(4) 应用本体，描述依赖于特定领域和具体任务的概念，这些概念在该领域中执行某些任务时有重要意义。

而 Perez 和 Benjamins 在分析各种分类方法基础上，将本体归结为 10 种类型：知识表示本体、普通本体、顶级本体、元(核心)本体、领域本体、语言本体、任务本体、领域-任务本体、方法本体和应用本体，但是这些类型之间界限不够明确，10 种本体之间存有交叉。

3.6.2　本体的建模

1. 建模方法

构建一个本体可以分为两个阶段，第一阶段用自然语言和图表来表示本体的各个组成要素，这个阶段称为非形式化阶段，成果是本体原型。第二阶段需要利用某种建模语言对本体模型进行编码，形成一个形式化的、无歧义的本体。

建模方法包括 IDEF-5 法、骨架法、TOVE 法、METHONTOLOGY 法、101 法等。每种方法都有各自的适用领域，也有各自的优缺点。

(1) IDEF-5 法将本体构建划分为 5 个步骤：确定本体构建的目标并划分成若干任务；收集数据；分析数据；构建初始本体；优化本体。

(2) 骨架法的流程见图 3.8。

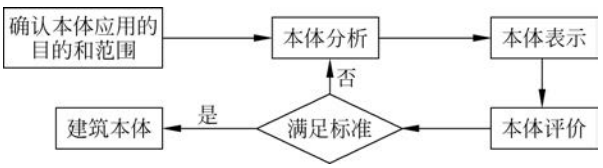

图 3.8　骨架法流程图

(3) TOVE 法的流程如图 3.9 所示，其中形式化是通过一阶谓词逻辑实现的。

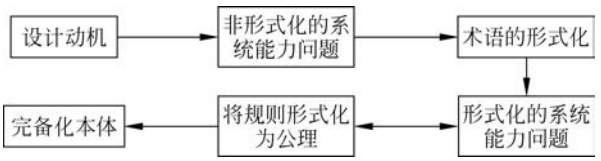

图 3.9　TOVE 法流程图

(4) METHONTOLOGY 法主要用于构建化学本体,包括规范说明、知识获取、概念化、集成、实现、评估以及文档等步骤。

(5) 101 法,主要用于领域本体的构建。包括七个步骤:①确定本体的领域和范畴;②考虑现有本体是否能够复用;③列举本体的重要术语;④定义类及其层次体系;⑤定义类的属性——槽;⑥定义槽的侧面;⑦创建实例。

2. 描述语言

本体语言允许用户为领域模型编写明确的形式化概念,需要包括:定义明确的语法;定义明确的语义;有效的推理机制;充分的表达能力以及便利的表达方式。本体语言包括 RDF(或 RDFS)、OIL、DAML、OWL、KIF、SHOE、XOL 等,目前 OWL(Web Ontology Language)使用较为广泛,下面简单介绍 OWL 的基础,包括 XML、RDF(或 RDFS),3.6.3 节"OWL"将再详细介绍 OWL 语言。

1) XML

XML(eXtensible Markup Language,可扩展标记语言)是一种机器可读文档的规范,与平台无关,广泛应用于互联网的数据存储和传输。一个 XML 文档形成一种树结构,包含根元素及其子元素,而每个子元素又可以拥有子元素,如表示一个书店的 XML 文档:

```
<?xml version = "1.0" encoding = "gb2312"?>
< bookstore >
< book category = "AI">
  < title lang = "zh - cn">人工智能与知识工程</title >
  < author >田盛丰,黄厚宽</author >
  < year > 1999 </year >
  < price > 29.40 </price >
</book >
< book category = "PROGRAMMING">
  < title lang = "en"> Practical Common Lisp </title >
  < author > Peter Seibel </author >
  < year > 2005 </year >
  < price > 58.28 </price >
</book >
</bookstore >
```

其中 book 是 bookstore 的子元素,而 title、author、year 和 price 是 book 的子元素,category 和 lang 分别是 book 和 title 元素的属性,用于表示分类和语言。

DTD(Document Type Definition)用来对 XML 文档结构进行验证,一个 DTD 文档包含元素的定义规则、元素之间的关系规则以及属性的定义规则。而 XML Schema 提供比 DTD 更强大的功能,数据类型更为完善,并且支持命名空间。上述 XML 文件的 XML Schema 如下所示:

```
< xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema" elementFormDefault = "qualified">
  < xs:element name = "bookstore">
    < xs:complexType >
      < xs:sequence >
        < xs:element maxOccurs = "unbounded" ref = "book"/>
```

```
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="book">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="title"/>
          <xs:element ref="author"/>
          <xs:element ref="year"/>
          <xs:element ref="price"/>
        </xs:sequence>
        <xs:attribute name="category" use="required" type="xs:NCName"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="title">
      <xs:complexType mixed="true">
        <xs:attribute name="lang" use="required" type="xs:NCName"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="author" type="xs:string"/>
    <xs:element name="year" type="xs:integer"/>
    <xs:element name="price" type="xs:decimal"/>
  </xs:schema>
```

2) RDF

尽管 XML 文件提供了一种良好的存储数据的方式,但是由于其不具有语义描述能力,因此 W3C 组织(World Wide Web Consortium, 万维网联盟)提出 RDF 用于克服 XML 的局限。RDF(Resource Description Framework,资源描述框架)是一种描述资源信息的框架,可用于表达关于任何可在 Web 上被标识的事物的信息,其核心是资源、属性、声明。其中资源可指任何事物,包括文档、物理对象和抽象概念等,RDF 中每个资源都拥有一个统一资源标识符(Universal Resource Identifier,URI),该标识符是一个字符串。属性用来描述某个资源的特征与性质或者资源之间的联系,属性也可以使用 URI 来标识。声明用来描述某个资源特定属性及其属性值,声明中属性的资源、属性名和值分别称为主体(Subject)、谓词(Predicate)和客体(Object)。RDF 的核心是< Subject, Predicate, Object >三元组,如表 3.4 所示就是一个三元组。

表 3.4 三元组示例

Subject	Predicate	Object
http://www.w3.org/	http://purl.org/dc/elements/1.1/title	"World Wide Web Consortium"

该三元组的含义为:

```
The title of http://www.w3.org/ is "World Wide Web Consortium"
```

其中 http://purl.org/dc/elements/1.1/title 表示都柏林核心元数据倡议(Dublin Core Metadata Initiative,一种元数据的规范)中预定义的文档 title 属性,也可以使用其他数据规范。该三元组也可以用 XML 文件表示为:

```
<rdf:RDF xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc = "http://purl.org/dc/elements/1.1/">
  <rdf:Description rdf:about = "http://www.w3.org/">
    <dc:title>World Wide Web Consortium</dc:title>
  </rdf:Description>
</rdf:RDF>
```

除了 XML 文件之外,还可以使用 N-Triples、Turtle 等方式存储 RDF 数据。

随着表示内容的变换,三元组中的三个要素可能会改变其在三元组中的作用,某个三元组的主体可能是另一个三元组的客体,某个三元组整体又可能是其他三元组的主体等。因此通过各个三元组之间的关系可以描述知识。

尽管 RDF 可以使用 XML 语言存储,但是二者存在很多区别:XML 是可扩展语言,而 RDF 只利用三元组表示二元关系;XML 固定为树状文本,而 RDF 可以采用其他灵活的方式来存储数据;XML 注重的是语法,对于语义并不关注,而 RDF 的三元组形式从本质上决定了 RDF 更适合用于描述语义。

3) RDFS

RDF 关注的是< Subject,Predicate,Object >三元组,即资源与资源之间的关系,但是并未对资源及关系进行限制或约束,这在表示知识时还有所欠缺。例如 The father of Zhang is Xiaoqiang 很容易表示成合法的三元组,但是如果 Zhang 是一个人,而 Xiaoqiang 指的是蟑螂时,这个三元组虽然从语法上成立但语义上存在矛盾。如果能够对 father 所取的值有所限制,限制为类别 person,或者更进一层为类别 man,就不会出现这种语义矛盾。RDFS 的引入正可以解决此类问题。

RDFS(RDF Schema)定义了一种模式定义语言,提供了一个定义在 RDF 上抽象的词汇集,主要包括:rdf:Class 类表示关于资源的所有类的集合;rdfs:subClassOf 类表示一个类的父类;rdf:Property 类表示 RDF 属性的类;rdfs:range 与 rdfs:domain 都是 rdf:Property 类的一个实例,前者用于表示属性的取值范围(即值域),而后者用于表示具有该属性的资源取值范围(即定义域)。对于上述例子可以限定谓词 father 的主体为 person,而客体为 man:

```
<rdfs:domain rdf:resource = "#person"/>
    <rdfs:range rdf:resource = "#man"/>
```

其中 man 定义为 person 的子类。

3.6.3 OWL

OWL 是由 W3C 组织定义的一种本体语言,用于表示语义 Web 中关于事物的知识和事物之间的关系。如前所述,RDF(或 RDFS)也能够表示 Web 中事物的知识与相互关系,但是 RDF(或 RDFS)在表示知识时存在一些不足之处,如无法确定两个属性是否等价、无法定义两个集合相交或不相交、无法对值域进行进一步限制(如限定一个人只有两个父母)等。因此 W3C 于 2004 年正式提出了 OWL 语言,并于 2009 年得到了修订和扩展。根据 W3C 的定义,OWL 是定义在 RDF(或 RDFS)之上的一种知识表示语言,包含了三个表达能力递增的子语言,分别是 OWL Lite、OWL DL 和 OWL Full,它们之间的区别如表 3.5 所示,其

中 OWL DL 中的 DL 表示描述逻辑(description logics)。

表 3.5 OWL 的三种子语言

子语言	描 述	例 子
OWL Lite	提供给那些只需要一个分类层次和简单属性约束的用户	允许支持基数(cardinality),但只允许基数为 0 或 1
OWL DL	支持那些需要在推理上拥有最大表达能力的用户,同时保留计算的完备性和可判定性	一个类可以是多重继承的子类,但不能是另外一个类的实例
OWL Full	支持需要最强的表达能力和 RDF 语法自由的用户,但没有可计算性的保证	一个类既可以是许多个体的一个集合,同时还可以是一个个体

在表达能力和推理能力上,每个子语言都是其之前的子语言的扩展,各个子语言的关系如下所示:

(1) 每个合法的 OWL Lite 本体都是一个合法的 OWL DL 本体;

(2) 每个合法的 OWL DL 本体都是一个合法的 OWL Full 本体;

(3) 每个有效的 OWL Lite 结论都是一个有效的 OWL DL 结论;

(4) 每个有效的 OWL DL 结论都是一个有效的 OWL Full 结论。

用户在选择使用哪种子语言时主要考虑:

(1) 选择 OWL Lite 还是 OWL DL 主要取决于用户需要 OWL DL 提供的表达能力的程度;

(2) 选择 OWL DL 还是 OWL Full 主要取决于用户有多需要 RDFS 的元模型机制(如定义关于类的类以及为类赋予属性);

(3) 相较于 OWL DL,OWL Full 对推理的支持并不完整。

OWL Full 子语言可以看成是对 RDF 的扩展,而另外两个子语言是对 RDF 受限版本的扩展。每个 OWL 文档都是一个 RDF 文档,而所有的 RDF 文档都是一个 OWL Full 文档,并且只有一部分 RDF 文档是合法的 OWL Lite 或 OWL DL 文档。因此将 RDF 文档迁移到 OWL 时,需要注意是否合法。

1. 命名空间

与其他语言类似,OWL 同样有命名空间的概念,可以在不同的命名空间中使用相同的名字代表不同的对象。一个典型的本体以命名空间开始,这些命名空间包含在 rdf:RDF 标签中。如:

```
<rdf:RDF
    xmlns     = "http://www.w3.org/TR/owl-guide/wine#"
    xmlns:vin = "http://www.w3.org/TR/owl-guide/wine#"
    xml:base  = "http://www.w3.org/TR/owl-guide/wine#"
    xmlns:food = "http://www.w3.org/TR/owl-guide/food#"
    xmlns:owl = "http://www.w3.org/2002/07/owl#"
    xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs = "http://www.w3.org/2000/01/rdf-schema#"
    xmlns:xsd = "http://www.w3.org/2001/XMLSchema#"     >
```

其中，xmlns 是缺省命名空间，也就是当前本体中出现没有前缀的标签时所引用的命名空间；xmlns:vin 和 xmlns:food 分别说明了当前本体中前缀为 vin 和 food 的命名空间；xml:base 说明该本体的基准 URI；xmlns:owl 说明前缀为 owl 的命名空间，是一个常规的命名空间，用于引入 OWL 词汇；xmlns:rdf 说明以 rdf 为前缀的命名空间，用于引入 RDF 定义的词汇；xmlns:rdfs 和 xmlns:xsd 分别说明 RDFS 和 XML Schema 数据类型命名空间。

可以在本体定义之前的文档类型定义中添加实体定义，用于说明命名空间，如：

```
<!DOCTYPE rdf:RDF [
    <!ENTITY vin "http://www.w3.org/TR/owl-guide/wine#" >
    <!ENTITY food "http://www.w3.org/TR/owl-guide/wine#" >]>
```

因此，上述命名空间的前四条即可改为：

```
xmlns     = "&vin;"
xmlns:vin = "&vin;"
xml:base  = "&vin;"
xmlns:food = "&food;"
```

2. 本体头部

有了命名空间之后，就可以如下断言的集合开始本体的构建：

```
<owl:Ontology rdf:about = "">
    <rdfs:comment>An example OWL ontology</rdfs:comment>
    <owl:priorVersion rdf:resource = "https://www.w3.org/TR/owl-guide/wine"/>
    <owl:imports rdf:resource = "https://www.w3.org/TR/owl-guide/food"/>
    <rdfs:label>Wine Ontology</rdfs:label>
    …… 
```

其中 owl:Ontology 用来收集关于当前文档的 OWL 元数据。rdf:about 属性为本体提供一个名称或引用，如果值为“”，则该名称默认为前述基准 URI；rdfs:comment 为本体添加注解；owl:priorVersion 是一个标准标签，用来为版本控制系统提供相关信息，指出历史版本的链接；owl:imports 提供了引入机制，只接受一个用 rdf:resource 属性标识的参数；rdfs:label 用自然语言为该本体进行标注。最后以</owl:Ontology>结束上述断言集合。

3. 类和个体

OWL 本体中的大部分元素与类(class)、属性(property)、类的实例(instance)以及这些实例间的关系有关。对于本体的运用往往与个体的推理能力有关，因此需要引入一种机制描述个体所属的类以及这些个体通过类成员关系继承到的属性。尽管本体允许为个体声明特定的属性，但是本体的推理能力往往表现为基于类层次关系的推理。因此，类及子类是 OWL 中非常重要的概念。

为了区分一个类是作为对象还是作为包含元素的集合，称后者为该类的外延(extension)。

无需显式声明，用户自定义类都是 owl:Thing 的子类。要定义特定领域的根类，只需将它们声明为一个具名类(named class)即可，OWL 也可以定义空类 owl:Nothing。例如在

制酒业的三个根类定义如下：

```
<owl:Class rdf:ID = "Winery"/>
<owl:Class rdf:ID = "Region"/>
<owl:Class rdf:ID = "ConsumableThing"/>
```

目前仅仅用了 ID 为上述根类命名，并没有指定这些类的成员等信息。rdf:ID 属性类似于 XML 中的 ID 属性，可以在文档中通过＃Region 引用 Region 类，而在其他本体中则需要通过全名 http://www.w3.org/TR/owl-guide/wine＃Region 引用。也可以采用 rdf:about ＝ "＃Region"替换上述第二行引入一个类，尤其是在分布式本体中其他本体需要引用这个类的话，采用 rdf:about＝“&ont;＃x”的方式。

OWL 中通过 rdfs:subClassOf 构造子类，子类是可传递的，如 PotableLiquid 是 ConsumableThing 的子类：

```
<owl:Class rdf:ID = "PotableLiquid">
  <rdfs:subClassOf rdf:resource = "＃ConsumableThing" />
  …… 
</owl:Class>
```

在一个类的定义中包含两个部分：命名或引用以及限制，subClassOf 就是一种限制。下面定义了类 Wine，它是 PotableLiquid 类的子类。

```
<owl:Class rdf:ID = "Wine">
  <rdfs:subClassOf rdf:resource = "&food;PotableLiquid"/>
  <rdfs:label xml:lang = "en"> wine </rdfs:label>
  <rdfs:label xml:lang = "fr"> vin </rdfs:label>
  ...
</owl:Class>
```

rdfs: label 提供了人类可读的名称，lang 属性表明支持多语言。

除了类之外，还需要描述它们的成员，即个体。个体通过如下方式声明为类中的一个成员：

```
<Region rdf:ID = "CentralCoastRegion" />
```

也可采用 rdf:about 声明，下列方式与上述语句等价：

```
<owl:Thing rdf:ID = "CentralCoastRegion" />
<owl:Thing rdf:about = "＃CentralCoastRegion">
    <rdf:type rdf:resource = "＃Region"/>
</owl:Thing>
```

其中 rdf:type 用于关联一个个体和它所属的类。

4. 属性

属性可以用来说明类的共同特征以及某些个体的专有特征，一个属性是一个二元关系。属性类型有两种：

(1) 数据类型属性(DatatypeProperty)：类实例与 RDF 文字或 XML Schema 数据类型间的关系。

(2) 对象属性(ObjectProperty),两个类的实例间的关系。

可以对属性对应的二元关系施以限制,例如 RDFS 中介绍的 domain(定义域)和 range(值域):

```
<owl:ObjectProperty rdf:ID = "madeFromGrape">
  <rdfs:domain rdf:resource = "#Wine"/>
  <rdfs:range rdf:resource = "#WineGrape"/>
</owl:ObjectProperty>
```

在 OWL 中,无显式操作符时的元素序列视为合取操作,如上述代码中属性 madeFromGrape 的 domain 和 range 分别为 Wine 和 WineGrape,该属性将 Wine 和 WineGrape 关联起来。

属性也可以像类一样按照层次结构来组织,如:

```
<owl:Class rdf:ID = "WineDescriptor" />
<owl:Class rdf:ID = "WineColor">
  <rdfs:subClassOf rdf:resource = "#WineDescriptor" />
  ...
</owl:Class>
<owl:ObjectProperty rdf:ID = "hasWineDescriptor">
  <rdfs:domain rdf:resource = "#Wine" />
  <rdfs:range rdf:resource = "#WineDescriptor" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID = "hasColor">
  <rdfs:subPropertyOf rdf:resource = "#hasWineDescriptor" />
  <rdfs:range rdf:resource = "#WineColor" />
  ...
</owl:ObjectProperty>
```

WineDescriptor 属性将 wine 与颜色(color)关联。hasColor 是 hasWineDescriptor 的子属性,hasColor 与 hasWineDescriptor 的不同在于它的值域被进一步限定为 WineColor。本例中 rdfs:subPropertyOf 关系表示:任何事物如果具有一个值为 X 的 hasColor 属性,那么它必然具有一个值为 X 的 hasWineDescriptor 属性。

还可以通过 locatedIn 属性关联所在区域,如:

```
<owl:ObjectProperty rdf:ID = "locatedIn">
  ...
  <rdfs:domain rdf:resource = "http://www.w3.org/2002/07/owl#Thing"/>
  <rdfs:range rdf:resource = "#Region" />
</owl:ObjectProperty>
```

对象属性用来关联个体与个体,而数据类型属性用来关联个体与数据类型。数据类型属性的取值范围是 RDF 文字或者是 XML Schema 数据类型中定义的那些简单类型。

大部分 XML Schema 内嵌数据类型可以在 OWL 中使用,这可以通过引用 URI http://www.w3.org/2001/XMLSchema 实现。OWL 中推荐使用表 3.6 所示的数据类型,这些数据类型与 rdfs:Literal 构成 OWL 的内置数据类型。

表 3.6 推荐使用的数据类型

xsd:string	xsd:normalizedString	xsd:boolean	
xsd:decimal	xsd:float	xsd:double	
xsd:integer	xsd:nonNegativeInteger	xsd:positiveInteger	
xsd:nonPositiveInteger	xsd:negativeInteger		
xsd:long	xsd:int	xsd:short	xsd:byte
xsd:unsignedLong	xsd:unsignedInt	xsd:unsignedShort	xsd:unsignedByte
xsd:hexBinary	xsd:base64Binary		
xsd:dateTime	xsd:time	xsd:date	xsd:gYearMonth
xsd:gYear	xsd:gMonthDay	xsd:gDay	xsd:gMonth
xsd:anyURI	xsd:token	xsd:language	
xsd:NMTOKEN	xsd:Name	xsd:NCName	

5. 属性特征

接下来介绍一些属性的具体特征，包括传递性、对称性等。

1) TransitiveProperty

对于属性 P，如果 $P(x,y)$与 $P(y,z)$成立时 $P(x,z)$也必然成立，则称属性 P 具有传递性。例如 locatedIn 是一个传递属性：

```
<owl:ObjectProperty rdf:ID = "locatedIn">
  <rdf:type rdf:resource = "&owl;TransitiveProperty" />
  <rdfs:domain rdf:resource = "&owl;Thing" />
  <rdfs:range rdf:resource = "#Region" />
</owl:ObjectProperty>
<Region rdf:ID = "SantaCruzMountainsRegion">
  <locatedIn rdf:resource = "#CaliforniaRegion" />
</Region>
<Region rdf:ID = "CaliforniaRegion">
  <locatedIn rdf:resource = "#USRegion" />
</Region>
```

根据传递性，SantaCruzMountainsRegion 位于 CaliforniaRegion，因此必然位于 USRegion。

2) SymmetricProperty

对于属性 P，如果 $P(x,y)$成立当且仅当 $P(y,x)$成立，则称属性 P 具有对称性。例如：

```
<owl:ObjectProperty rdf:ID = "adjacentRegion">
  <rdf:type rdf:resource = "&owl;SymmetricProperty" />
  <rdfs:domain rdf:resource = "#Region" />
  <rdfs:range rdf:resource = "#Region" />
</owl:ObjectProperty>
<Region rdf:ID = "MendocinoRegion">
  <locatedIn rdf:resource = "#CaliforniaRegion" />
  <adjacentRegion rdf:resource = "#SonomaRegion" />
</Region>
```

adjacentRegion 属性具有对称性，而 locatedIn 不具有对称性。

3) FunctionalProperty

对于属性 P，如果 $P(x,y)$ 和 $P(x,z)$ 成立时必然有 $y=z$，则称属性 P 具有函数性，如：

```
<owl:Class rdf:ID = "VintageYear" />
<owl:ObjectProperty rdf:ID = "hasVintageYear">
  <rdf:type rdf:resource = "&owl;FunctionalProperty" />
  <rdfs:domain rdf:resource = "#Vintage" />
  <rdfs:range rdf:resource = "#VintageYear" />
</owl:ObjectProperty>
```

hasVintageYear 属性具有函数性，因为每个 Vintage 只能关联到一个年份。

4) inverseOf

对于属性 P_1 和 P_2，如果 $P_1(x,y)$ 成立当且仅当 $P_2(y,x)$ 成立，则称属性 P_1 是属性 P_2 的逆属性，如：

```
<owl:ObjectProperty rdf:ID = "hasMaker">
  <rdf:type rdf:resource = "&owl;FunctionalProperty" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID = "producesWine">
  <owl:inverseOf rdf:resource = "#hasMaker" />
</owl:ObjectProperty>
```

hasMaker 和 producesWine 互为逆属性，因为葡萄酒厂(Winery)生产 Wine，而 Wine 又有自己的生产商。

5) InverseFunctionalProperty

对于属性 P，如果 $P(y,x)$ 与 $P(z,x)$ 成立时必然有 $y=z$ 成立，则称属性 P 具有反函数性，如前例中的 producesWine 属性。可以如下定义 hasMaker 属性和 producesWine 属性达到和前例中相同的效果：

```
<owl:ObjectProperty rdf:ID = "hasMaker" />
<owl:ObjectProperty rdf:ID = "producesWine">
  <rdf:type rdf:resource = "&owl;InverseFunctionalProperty" />
  <owl:inverseOf rdf:resource = "#hasMaker" />
</owl:ObjectProperty>
```

6. 属性限制

还可以通过进一步的约束限制属性的取值范围。

1) allValuesFrom 和 someValuesFrom

目前为止介绍的属性作用域都是全局的，而 allValuesFrom 与 someValuesFrom 提供了一种属性只在局部有作用的定义方式：

```
<owl:Class rdf:ID = "Wine">
  <rdfs:subClassOf rdf:resource = "&food;PotableLiquid" />
  ...
```

```
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource = "#hasMaker" />
        <owl:allValuesFrom rdf:resource = "#Winery" />
      </owl:Restriction>
    </rdfs:subClassOf>
    ...
  </owl:Class>
```

其中对属性的限制必须在 owl:Restriction 中通过 owl:onProperty 指出受限制的属性，allValuesFrom 限制了 Wine 的 hasMaker 属性取值如果有的话必须为 Winery，除了 Wine 以外其他食物并无此限制。另外，allValuesFrom 与 someValuesFrom 的区别在于前者要求取值必须是某个类别，而后者要求取值中至少有一个是某个类别，如果将上例中的 allValuesFrom 替换为 someValuesFrom，意味着 Wine 的 hasMaker 属性取值范围中至少有一个是 Winery，但同时还可以有其他类别。

2) Cardinality(基数)

在前文介绍 OWL 三种子语言时已经出现了基数这个概念，基数用来指定一个关系中的元素的数量。例如，指定 Vintage 只能有一个 VintageYear：

```
<owl:Class rdf:ID = "Vintage">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource = "#hasVintageYear"/>
      <owl:cardinality rdf:datatype = "&xsd;nonNegativeInteger">1
                    </owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

正如前文所说，OWL Lite 中只能指定基数为 0 或者 1，而其他两种子语言则允许基数有更多取值。此时可以使用 owl:maxCardinality 和 owl:minCardinality 限制基数的上下限。

3) hasValue

hasValue 允许基于特定的属性值来定义类(OWL Lite 子语言不支持 hasValue)，如：

```
<owl:Class rdf:ID = "Burgundy">
  ...
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource = "#hasSugar" />
      <owl:hasValue rdf:resource = "#Dry" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

其中限制所有的 Burgundy 酒都是干(dry)葡萄酒，即所有 Burgundy 的 hasSugar 属性至少有一个取值为 Dry，该限制是一个局部限制，只在 Burgundy 中成立。本例中实际上定义了一个类，该类中有一个成员是 Dry。

7. 本体的等同与区别

作为一种重要的知识表示方法，本体需要共享和重用机制。

1）类和属性的等同(equivalentClass，equivalentProperty)

equivalentClass 用于关联当前本体中某个类与其他本体中的某个类，例如在 food 本体中定义：

```
<owl:Class rdf:ID = "Wine">
  <owl:equivalentClass rdf:resource = "&vin;Wine"/>
</owl:Class>
```

上述语句使得 food 本体中的 Wine 类等同于 vin 本体中的 Wine 类。

与 equivalentClass 类似，owl:equivalentProperty 用于关联属性。

2）个体的等同(sameAs)

个体的等同与类的等同类似，例如 Mike 喜欢的酒等同于某一种酒：

```
<Wine rdf:ID = "MikesFavoriteWine">
  <owl:sameAs rdf:resource = "#StGenevieveTexasWhite" />
</Wine>
```

由于 hasMaker 属性具有函数性，因此下列语句表示 Bancroft = Beringer：

```
<owl:Thing rdf:about = "#BancroftChardonnay">
  <hasMaker rdf:resource = "#Bancroft" />
  <hasMaker rdf:resource = "#Beringer" />
</owl:Thing>
```

由于 OWL 并没有不重名假设，因此使用不同的名字描述同样的个体不会出现冲突。

3）个体的区别(differentFrom，AllDifferent)

与 sameAs 相反，differentFrom 和 AllDifferent 用于表示个体之间是不同的，例如 Dry、Sweet 以及 OffDry 都是不同的个体：

```
<WineSugar rdf:ID = "Dry" />
<WineSugar rdf:ID = "Sweet">
  <owl:differentFrom rdf:resource = "#Dry"/>
</WineSugar>
<WineSugar rdf:ID = "OffDry">
  <owl:differentFrom rdf:resource = "#Dry"/>
  <owl:differentFrom rdf:resource = "#Sweet"/>
</WineSugar>
```

differentFrom 用于表示两个个体的区别，多个个体的不同可以使用 AllDifferent 描述，如表示 Red、White 和 Rose 是不同的：

```
<owl:AllDifferent>
  <owl:distinctMembers rdf:parseType = "Collection">
    <vin:WineColor rdf:about = "#Red" />
    <vin:WineColor rdf:about = "#White" />
    <vin:WineColor rdf:about = "#Rose" />
```

```
    </owl:distinctMembers>
  </owl:AllDifferent>
```

其中 distinctMembers 必须与 AllDifferent 组合使用。

8. 复杂类

除了前文介绍的基本类之外，OWL 支持使用集合运算符构造复杂类，还可以构造枚举类和不相交类。OWL Lite 子语言不支持复杂类。

1）集合运算符（intersectionOf，unionOf，complementOf）

可以通过运算符 intersectionOf、unionOf 和 complementOf 构造集合的交、并及补。例如通过运算符 intersectionOf 构造一个 WhiteWine 类，既是 Wine 又有颜色 White：

```
<owl:Class rdf:ID = "WhiteWine">
  <owl:intersectionOf rdf:parseType = "Collection">
    <owl:Class rdf:about = "#Wine" />
    <owl:Restriction>
      <owl:onProperty rdf:resource = "#hasColor" />
      <owl:hasValue rdf:resource = "#White" />
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

unionOf 的使用与 intersectionOf 类似，如构造一种 Fruit 类，要么是 SweetFruit 要么是 NonSweetFruit：

```
<owl:Class rdf:ID = "Fruit">
  <owl:unionOf rdf:parseType = "Collection">
    <owl:Class rdf:about = "#SweetFruit" />
    <owl:Class rdf:about = "#NonSweetFruit" />
  </owl:unionOf>
</owl:Class>
```

complementOf 运算符构造一个类，该类含有不属于某个类的所有个体，如构造一个 NonConsumableThing 类，包含所有不属于 ConsumableThing 的个体：

```
<owl:Class rdf:ID = "ConsumableThing" />
<owl:Class rdf:ID = "NonConsumableThing">
  <owl:complementOf rdf:resource = "#ConsumableThing" />
</owl:Class>
```

2）枚举类（oneOf）

OWL 可以通过 oneOf 枚举一个类的所有成员定义该类，如 WineColor 类的成员是 White、Rose 以及 Red 这三个个体：

```
<owl:Class rdf:ID = "WineColor">
  <rdfs:subClassOf rdf:resource = "#WineDescriptor"/>
  <owl:oneOf rdf:parseType = "Collection">
    <owl:Thing rdf:about = "#White"/>
    <owl:Thing rdf:about = "#Rose"/>
```

```
    <owl:Thing rdf:about = "#Red"/>
  </owl:oneOf>
</owl:Class>
```

owl:Thing 也可直接替换为 WineColor。

3) 不相交类(disjointWith)

disjointWith 用于指定与某个类互不相交的一组类,如 Pasta 与 Meat、Fowl、Seafood、Dessert 以及 Fruit 都不相交:

```
<owl:Class rdf:ID = "Pasta">
  <rdfs:subClassOf rdf:resource = "#EdibleThing"/>
  <owl:disjointWith rdf:resource = "#Meat"/>
  <owl:disjointWith rdf:resource = "#Fowl"/>
  <owl:disjointWith rdf:resource = "#Seafood"/>
  <owl:disjointWith rdf:resource = "#Dessert"/>
  <owl:disjointWith rdf:resource = "#Fruit"/>
</owl:Class>
```

注意并没有指定 Meat、Fowl、Seafood、Dessert 以及 Fruit 之间是否相交。

最新使用的 OWL 版本称为 OWL 2,是 2009 年发布版本的第二版,其核心与第一版并无本质区别。OWL 2 包含了三种标准:OWL 2/EL、OWL 2/QL 以及 OWL 2/RL,各自有不同的适用场景。

3.7 知识图谱

自 2012 年美国 Google 公司提出知识图谱(knowledge graph)的概念以来,近些年知识图谱已经成为知识表示领域乃至人工智能学科中的一个重要研究方向。尽管 Google 公司提出知识图谱的初衷是为了在搜索领域中强化搜索引擎的能力,提高用户的搜索质量并改善用户体验,但目前知识图谱已经应用于如智能推荐、智能医疗等各个领域中。作为国内搜索行业的先行者,百度公司同样于 2014 年提出了百度知识图谱,截止到 2019 年底,百度知识图谱规模已经达到亿级实体和千亿级属性关系,是中文领域最大的知识图谱,同时知识图谱服务规模从 2014 年开始增长了 490 倍。

非常巧合的是,深度学习也正是从 2012 年开始得到学术界和产业界的一致认同,与深度学习提出伊始并未受到太多关注不同,知识图谱由于是由 Google 公司提出的,因此自诞生之日起就备受关注。如果认为深度学习的发展掀起了 20 世纪 80 年代之后又一次由连接主义推动的人工智能热潮,符号主义在这次人工智能热潮中也占据了相当重要的地位,主要体现在知识图谱的研究与应用中。知识图谱并非突如其来的一种新技术,而是前期多种研究成果的后继发展,如语义网络、本体、语义 Web 等。其中语义网络和本体已经在本章前文中有介绍,而语义 Web 是图灵奖获得者 Berners-Lee 于 1998 年提出的,是具有语义支持的 Web,是对现有 Web 的延伸与扩展,是知识表示与推理在 Web 中的应用,其实现的核心技术包括前述的 XML、RDF 以及本体。

知识图谱本质上是描述世界中存在的实体或概念及其关系的语义 Web,这种描述往往通过三元组完成。第 3.6 节介绍的本体也可以通过三元组的方式构造成语义 Web,但是本

体与知识图谱存在一些显著的区别。首先，知识图谱往往是比本体更大的一个概念，在知识图谱的构建中本体的搭建是其中一步重要的工作；其次，本体描述的往往是元知识，如领域本体描述的是该领域中的元知识，这些元知识相对而言较为固定，因此构建好的本体一般无须时常更新，而知识图谱则不同，知识图谱除了描述元知识之外，还需要描述时常变化的一些实体或概念及其关系，因此知识图谱往往一直处于动态更新过程中。最后，本体更多时候只是用来表示实体或概念及其相互关系，而知识图谱往往除了描述这些关系之外，还关注如何获得这些关系，又以何种方式存储，并在构建知识图谱之后如何进行推理。图 3.10 所示是一张简单的关于我国各省份的知识图谱(图中具体数据来自于百度百科)。

在图 3.10 中，圆形或椭圆形表示实体，它们之间的连线表示关系，也可以将人口、面积和 GDP 看成是每个省的属性。上述知识图谱可以通过多个三元组进行描述，如<河南，是一个(ISA)，省份>、<河南，省会，郑州>、<河南，人口，9605 万人>等。

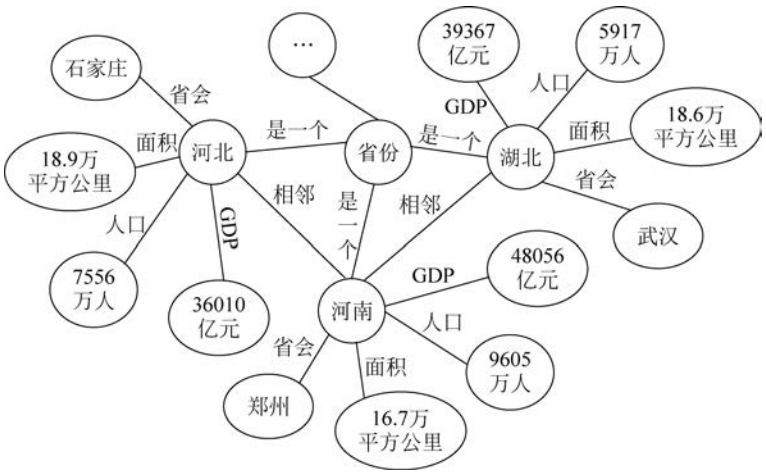

图 3.10　我国关于省份的知识图谱示例

3.7.1　构建知识图谱

一般认为知识图谱的构建与更新过程如图 3.11 所示。

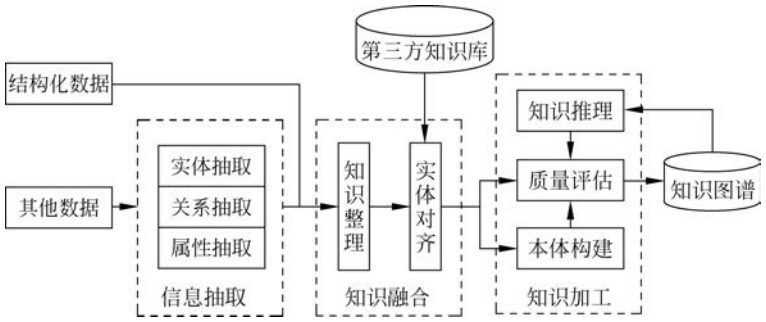

图 3.11　知识图谱的构建与更新过程

从图 3.11 可以看出，结构化的数据可以作为输入，其他数据如半结构或非结构化数据同样可以作为输入，区别就在于结构化数据中无须附加操作可以直接获得相关信息，而其他

数据则需要抽取相关内容才可为知识图谱所用。考虑到不同知识库使用的实体与名称不一样，需要进行实体对齐，主要包括实体消歧和共指消解。之后进入知识加工部分，构建本体，得到知识图谱原型，通过知识推理对知识图谱原型进行质量评估，丢弃低置信度的知识。最终得到一个较为完善的知识图谱。当构建好的知识图谱需要更新时有两种方式可以采用，第一种是从更新后的数据出发，重新构建一个知识图谱，另一种则是在原有知识图谱基础上增量更新知识图谱。

1. 信息抽取[①]

结构化数据可以通过直接读取数据库或数据文件获取其某些信息，而其他数据如文本、图像等数据却无法直接获得，需要通过信息抽取获得相关信息。信息抽取的任务主要包括实体抽取、关系抽取和属性抽取。从其他数据中抽取相关信息之后，与结构化数据一起组成知识融合阶段的输入。

2. 知识融合

知识融合阶段，首先将结构化数据与其他数据抽取的信息综合起来整理成知识，然后与第三方知识库已有知识进行融合，完成实体对齐，主要完成的工作包括实体消歧和共指消解，实体消歧用于解决同一标识符表示不同实体的问题，而共指消解用于解决识别多数据源中使用不同标识符表示同一实体的问题。

3. 知识加工

根据知识融合的结果可以构建本体，本体的构建可以采用人工用工具编辑的方式，也可以采用数据驱动的自动化方式。但由于手工方式工作量过大并且需要相关领域专家的配合，因此一般都是以现有本体为基础，通过自动构建技术构建新的本体。本体构建之后形成一个知识图谱原型，此时的知识图谱往往还不完善，存在一些缺陷，需要通过知识推理对本体进行完善，同时通过质量评估保留质量较高的知识。

3.7.2 存储知识图谱

目前主要有三种存储知识图谱的方法，分别是关系数据库、三元组数据库以及图数据库。

1. 关系数据库

在关系数据库中表示三元组有多种方案，最简单的是将所有三元组存储到一个三元组表中，但是采用这种方案时多个自连接操作导致查询性能低下。水平表将知识图谱中一个主体在三元组中对应的所有谓词及客体存储在同一行中，这种方案导致该表的列数等于知识图谱中不同的谓词数量，可能会超出关系数据库允许的列数上限，另外每一行存在大量空值。属性表是对水平表的一种细化，将同类主体存储在同一个表中，这种方案的缺点在于可能会存在大量类别的主体，需要建立数量过多的表。垂直划分按照谓词划分表，每个谓词对

① 注：有研究者将这一阶段称为知识抽取，笔者认为信息抽取可能更为恰当。

应一张表，表中仅含主体和客体，这种方案需要创建大量表，越复杂的查询效率越低。六重索引是三元组表的扩充，为所有三元组按照主体、谓词、客体的排列不同而建立六张表，可以避免三元组表中的自连接，提高查询效率，但是显然需要三元组表的六倍存储空间。

2. 三元组数据库

三元组数据库是专门为存储 RDF 数据开发的知识图谱数据库，包括 Jena、RDF4J、AllegroGraph 等多种开源和商业数据库。

3. 图数据库

Neo4j 是目前使用最为广泛的图数据库，查询效率高，尽管其不支持分布式存储，但是对于并不是非常大规模的知识图谱来说，Neo4j 是一个较好的选择。JanusGraph 可以实现分布式存储，因此更适合于超大规模的知识图谱存储。OrientDB 是一种支持多模式的数据库，也能作为图数据库存储知识图谱。

3.7.3 知识图谱推理

知识图谱推理可简单分为演绎推理和归纳推理，演绎推理主要是利用已知的一些规则通过逻辑推理得出一个原有知识库隐含的结论，例如已知“中原地带风沙较大”“河南省地处中原”可以从图 3.10 所示的知识图谱中得到“郑州风沙较大”的结论。知识图谱中的演绎推理主要包括本体推理以及规则推理，本体推理利用本体进行推理，而规则推理主要通过产生式规则完成推理，也可以通过逻辑编程语言 Prolog 及其子集 Datalog 完成。

而归纳推理则是从已有现象观察到某些规律，从而得到原有知识库并不隐含的结论，归纳推理一般分为归纳泛化和类比推理。前者是根据某些个体特征得到整体特征，然后将整体特征应用于其他个体中从而得到某些结论；而后者是根据某些个体特征直接推导出其他个体的特征，并不需要得到整体特征。即使归纳推理的步骤并无错误，但其结果并不保证一定是成立的。知识图谱中的归纳推理主要包括基于图的推理、基于规则学习的推理以及基于向量表示的推理。基于图的推理主要利用知识图谱中的图结构得到推理结论，基于规则学习的推理首先需要从数据中学习到规则，构成新知识，然后进行规则推理，而基于向量表示的推理则计算两个个体向量表示的相似度和相异度，从而将一个个体的特征赋予与其相似的其他个体。

尽管知识图谱自提出还不超过十年，但是多个通用知识图谱如 DBpedia、Wikidata、ConceptNet 以及 Yago 等已经得到了广泛的应用，在特定领域中知识图谱也得到了充足的发展，如阿里电子商务知识图谱等。知识图谱涉及的理论与算法非常广泛，而本章主要关注的是知识表示，知识图谱知识表示的核心却在于本体，因此本节仅对知识图谱做一简单介绍，供读者大致了解，更详细的内容请参考相关资料。

现阶段备受瞩目的深度学习并不太关注知识的显式表示，因为在深度学习中大部分知识都由深层网络的连接隐式表示。但是随着人工智能的螺旋式发展，符号主义所关注的知识表示依然会是人工智能研究的核心领域之一，正如近年来知识图谱的发展与本体的研究息息相关。