计算机科学丛书

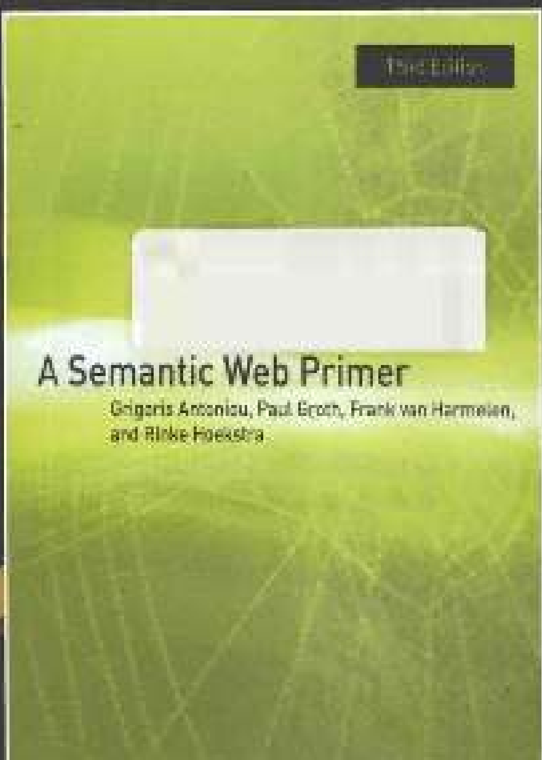
原书第3版

语义网基础教程

（希）Grigoris Antoniou （荷）Paul Groth
（荷）Frank van Harmelen （荷）Rinke Hoekstra 著

胡伟 程淼 黄智生 译

A Semantic Web Primer
Third Edition



机械工业出版社
China Machine Press

语义网基础教程 原书第3版

A Semantic Web Primer Third Edition

"'数据万维网'发展迅速，但是如果你真的想要理解正在发生什么，对语义理解是必要的。本书深入浅出地展示了'语义'系统如何应对规模的变化，以及是什么限制了它们。本书是面向学生的工程知识的原汁原味、最好的教材。"

——Dave Robertson，爱丁堡大学信息学院院长

"本书第3版紧跟快速发展的语义网技术的步伐，带领读者从入门开始掌握语义网应用开发的细节。本版更好地调整了内容，强调了SPARQL等关键技术并探讨了链接数据等新趋势。"

——Jerome Euzenat，法国国家信息与自动化研究院（INRIA）高级研究科学家

随着网络内容的机器解读，语义网的发展孕育着万维网及其应用的一场革命。本书为这个持续发展的领域提供了一个导引，描述了其核心思想、语言和技术。本书可用作教材或专业人员的自学读本，主要介绍适合本科程度的基础概念和技术，并提供了练习和项目、建议阅读材料等，旨在帮助读者通过学习得以自行开发有关的应用。

第3版彻底更新了这本使用广泛的教材，提供了大量反映这个快速发展领域的新内容。

主要内容包括

- 对多种语言（OWL2、规则）的讲解扩展了RDF和OWL，引入了Turtle和RDFa的介绍。
- 第4章专门致力于介绍OWL2这一新的W3C标准。
- 新增了对查询语言SPARQL、规则语言RIF，以及规则和本体语言及应用的交互可能性的介绍。
- 第6章关于语义网的应用体现了过去几年语义网的快速发展。
- 第7章提供了构建工程的思路。

作者简介

Grigoris Antoniou 希腊赫拉克利翁研究和技术基金会（Foundation for Research and Technology Hellas，FORTH）计算机科学研究所教授。他是语义网OWL语言创始人之一，主要研究兴趣包括近似推理、语义技术等。

Paul Groth、Frank van Harmelen和Rinke Hoekstra 分别是荷兰阿姆斯特丹自由大学（VU University Amsterdam）计算机科学系知识表示与推理研究组的副教授、教授和博士后研究员。

上架指导：计算机\人工智能\语义网

ISBN 978-7-111-47544-6

9 787111 475446

定价：49.00元

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

计算机科学丛书

原书第3版

语义网基础教程

(希) Grigoris Antoniou　(荷) Paul Groth
(荷) Frank van Harmelen　(荷) Rinke Hoekstra　著

胡伟 程龚 黄智生 译

A Semantic Web Primer

Third Edition

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

语义网基础教程（原书第3版）/（希）安东尼乌（Antoniou, G.）等著；胡伟，程龚，黄智生译. —北京：机械工业出版社，2014.9
（计算机科学丛书）
书名原文：A Semantic Web Primer, Third Edition

ISBN 978-7-111-47544-6

Ⅰ. 语… Ⅱ. ①安… ②胡… ③程… ④黄… Ⅲ. 语义网络–教材 Ⅳ. TP18

中国版本图书馆CIP数据核字（2014）第170001号

本书版权登记号：图字：01-2014-2861

本书主要介绍语义网的核心思想、语言和技术，为这个日新月异的领域提供指南。全书共8章，主要内容包括语义网的愿景、技术标准（RDF、SPARQL、OWL2、RIF等）、以及最新的语义网典型应用和本体工程。第3版增加了大量有关语义网最新进展的内容。本书可作为语义网领域的入门教材和参考书，适合所有对语义网感兴趣的读者。

出版发行：机械工业出版社（北京市西城区百万庄大街22号 邮政编码：100037）

责任编辑：朱秀英	责任校对：殷 虹
印 刷：北京诚信伟业印刷有限公司	版 次：2014年9月第1版第1次印刷
开 本：185mm×260mm 1/16	印 张：10.75
书 号：ISBN 978-7-111-47544-6	定 价：49.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991 88361066　　投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259　　读者信箱：hzjsj@hzbook.com

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线。由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

华章教育

华章科技图书出版中心

It is with great pride that I wrote the preface to this Chinese translation of the 3rd and revised edition of *A Semantic Web Primer*.

Much has changed in the Semantic Web community since the publication of the Chinese translation of the first edition, in 2007. Semantic Web technology has matured, and has been adopted in many different sectors of industry and government across the world. In Chinese in particular, the Semantic Web community has strengthened substantially in the past decade. Chinese researchers are now contributing world-class research results on Semantic Web technologies, and are well regarded internationally.

The Chinese universities and research foundations have invested substantially in research in this field, allowing a new generation of young researchers to develop.

Also content-wise, the Semantic Web field has developed substantially since the first edition of this book. Developments such as OWL2, SPARQL, N3 syntax and many new applications made it necessary to update the first edition. We are therefore extremely grateful to Dr. Wei Hu and Dr. Gong Cheng of the State Key Laboratory for Novel Software Technology, Dept. of Computer Science and Technology of Nanjing University for taking the initiative in the translation of the new edition of our book, and for their excellent work on completing this translation in such a short time. We also thank Prof. Zhisheng Huang of the VU University for his tireless assistance in this process. We could not have wished for a better translation team.

We sincerely hope that this translated third edition of *A Semantic Web Primer* will contribute to further strengthen the interests in Semantic Web research and development, and that it will be a useful aid to students and teachers alike.

（我怀着非常自豪的心情为本书的中文翻译版撰写序言。

自从2007年本书第1版的中文翻译版出版以后，语义网社区发生了许多变化。语义网技术已经成熟，并且在世界范围内被工业界和政府部门等诸多领域所采用。尤其在中国，过去10年里语义网社区已经取得了实质性的进展。中国的研究人员正在为语义网技术贡献世界级的研究成果，并且在国际上也备受关注。

中国的大学和科研基金对这个领域投入颇多，使得新一代的青年研究学者得以发展。

就内容而言，自从本书第1版出版以后，语义网领域取得了诸多实质性的进展。诸如OWL2、SPARQL、N3语法等的发展以及许多新的应用使得很有必要对本书第1版进行修订。在此，我们非常感谢南京大学计算机软件新技术国家重点实验室和计算机科学与技术系的胡伟博士以及程龚博士主动承担了本书新版的翻译工作，并对他们在这么短的时间里出色地完成这项翻译工作表示谢意。我们也感谢荷兰阿姆斯特丹自由大学的黄智生教授在此过程中不知疲倦的支持。我们很难期待有比这更好的翻译团队。

我们真诚地希望本书能够进一步增强大家对语义网的研究和开发兴趣，为老师和同学等提供有益帮助。）

Frank van Harmelen

Grigoris Antoniou

Paul Groth

Rinke Hoekstra

译者序

高速发展的万维网已成为人类历史上影响最深远、最广泛的信息传播媒介，同时也推动着下一代万维网技术的发展。1998 年，万维网的发明人蒂姆·伯纳斯-李（Tim Berners-Lee）提出了语义网（Semantic Web，也译作语义万维网）的设想。2001 年，《科学美国人》杂志刊登题为“The Semantic Web”的科普文章，宣告了语义网的诞生。同年，万维网联盟（W3C）成立了一系列工作组，致力于制定语义网技术规范。随着资源描述框架（RDF）、万维网本体语言（OWL2）、RDF 查询语言（SPARQL）及规则交换格式（RIF）等一批技术规范被确立为推荐标准，语义网为万维网上的知识表示、推理、交换和复用奠定了基础。随之而来的是语义网技术在众多领域的蓬勃发展和广泛应用。

本书是语义网的入门性教科书，作者在语义网研究和应用方面有着广博的见识和丰富的经验。自 2004 年出版第 1 版以来，本书已被许多大学和研究机构选为语义网课程教材并被翻译为多国语言，其中文翻译版由中国科学技术大学的陈小平教授等译。伴随语义网技术的快速发展，本书两度大幅更新。在最新的第 3 版中，作者加入了新近成为 W3C 推荐标准的 OWL2 和规则语言的内容，同时补充了 N3/Turtle 和 RDFa 等流行的 RDF 语法，另外还介绍了最新的语义网典型应用。总之，这是一本理论与实践相结合、通俗易懂的好书。

本书的翻译工作由胡伟、程龚和黄智生 3 人共同完成。胡伟负责翻译了第 1 ~ 2 章、第 4 章和第 7 ~ 8 章，并统稿全书。程龚负责翻译了第 3 章、第 5 ~ 6 章和附录部分。黄智生负责翻译了序言，并审校全书。另外，特别感谢本书作者 Frank van Harmelen 教授就书中多处细节为译者进行了解答和确认，并为翻译提供了一切便利。

限于水平，译文中难免有错误与不足之处，欢迎读者朋友批评指正。

译者

2014 年 6 月

与传统观念不同的是，信息系统作为量身定做的、成本密集型的数据库应用的时光一去不复返了。这种变化一部分是受到逐渐成熟的软件产业的推动（软件产业大量使用了现成的通用组件和标准的软件解决方案），而另一部分则是由于信息革命的冲击。反过来，这种改变导致了对信息服务的一系列全新需求，即要求其表示模式与交互模式的统一性、软件体系结构的开放性以及使用范围的全局性。这些需求主要来自诸如电子商务、银行业、制造业（包括软件产业本身）、培训、教育和环境管理等领域，恕不一一列举。

未来的信息系统必须支持与各种运行在异构平台和分布式信息网络上的独立多厂商数据源和遗留应用的平滑交互。元数据在描述这类数据源并使之融合为一体时将起到至关重要的作用。

同时，面向社区的更多样化交互模式将必须由下一代信息系统来提供支持。这些交互可能涉及导航、查询和检索，并且必须与个性化通知、标注和分析机制相结合。这些交互也将必须提供与应用软件的智能化接口，并且需要动态地集成到定制的、高度连接的协同环境中。此外，政府和企业等在信息资源上大量投资，需要特定的措施来确保其内容的安全性、保密性和准确性。

所有这些都是下一代信息系统所面临的挑战，我们将这样的系统称为协同信息系统（cooperative information system）。

专业地讲，协同信息系统将服务于以内容－社区－商务（content-community-commerce）为特征的多样化需求。这些需求源自现有的软件解决方案（例如企业资源规划系统和电子商务系统）的当前发展趋势。

建立协同信息系统的一个主要挑战在于开发新技术以满足目前大量投资的信息资源和系统不断增强和演化的需要。这些技术必须提供一个恰当的基础设施以支持软件开发和演化。

协同信息系统的早期研究成果正在成为面向社区的信息门户或网关的核心技术。如一个提供“一站式购物”的网关提供了广泛的信息资源和服务，从而获得了一个忠实的用户群。

面向协同信息系统的研究进展并非来自信息技术中的单一研究领域。数据库和基于知识的系统、分布式系统、群件以及图形用户界面都已经是成熟的技术。随着个别技术的逐步完善，技术进步的最大杠杆将来自建立和管理一个无缝连接的协同信息系统的进展。

数据源中的数据之所以有用，是因为它们建立了现实世界及其主题（或应用程序、论域）的部分模型。数据语义（data semantic）的问题在于建立和维护数据源，即模型（model）和它所涉及主题间的对应关系。这些模型可以是存储某公司员工资料的数据库，描述零件、项目与供应商的数据库模式，提供大学信息的网站，或描述滑铁卢战役的一个纯文本文件。这样的问题自从第一个数据库出现之后就一直与我们相伴。然而，只要数据库的运行环境仍然是封闭和相对稳定的，该问题就仍然可控。在这样的环境下，数据的含义可以从数据库中正确地提取出来，并委托给一小组常规用户和应用程序。

万维网的出现改变了这一切。今天的数据库已经以某种方式在万维网上可用，其中用户、

应用程序及其使用都是开放的，而且处于不断变化之中。在这样的环境下，数据的语义必须随数据一起创建。对于人类用户，通过选择适当的表示形式来实现。但是对于应用程序，这种语义则必须以一种形式化的、机器可处理的形式提供。因此需要语义网[1]。

毫不奇怪，蒂姆·伯纳斯-李的倡议受到了研究人员和开发人员等的巨大关注。现在已有国际语义网系列会议[2]、爱思唯尔出版的《Semantic Web Journal》[3]，以及产业委员会正在关注第一代语义网技术标准。

本书介绍了语义网概念，包括XML、DTD和XML模式，RDF和RDFS，OWL，逻辑和推理等。本书的另一个特色是通篇使用例子和应用来展现概念。希望读者会觉得本书有趣、精辟，并从中受益。

---

[1] Tim Berners-Lee and Mark Fischetti, *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor.* San Francisco: HarperCollins, 1999.

[2] http://iswc.semanticweb.org.

[3] 期刊名称应为《Journal of Web Semantics》，主编的网址是：http://www.websemanticsjournal.org/，另有一个非爱思唯尔出版的期刊叫做《Semantic Web Journal》。——译者注

第 1 章
A Semantic Web Primer, Third Edition

语义网的愿景

1.1 引言

1.1.1 语义网的动机

"语义网"的主要愿景可以概括为一句话：*使计算机更能解读万维网*（to make the web more accessible to computers）。当前万维网是一个文字和图片网络，这些媒体对人而言很有用，但是计算机在目前的万维网上只发挥了非常有限的作用：它们索引关键词，将信息从服务器端传输到客户端，仅此而已。所有的智能工作（选择、组合、聚集等）必须通过人类读者来完成。如果我们能够使得万维网更适合机器处理，使得万维网上充满机器可读取、"可理解"的数据（data）将会如何？这样的一个万维网将有助于完成许多在当前万维网上不可行的事情：搜索（search）将不再局限于简单地查找关键词，而将变得更加语义化，包括查询同义词，识别同音异义词，并且考虑搜索查询的情境和意图。如果个人浏览 agent 能够理解一个网页的内容并将其裁剪为个人感兴趣的概述，网站将变得更加个性化（personalized）。通过当前用户的活动来动态确定那些网页会是有用的目的地，而非为所有用户预先硬编码相同的链接，链接（linking）将变得更加语义化。跨网站集成（integrate）信息也将成为可能，而不像目前用户在某个网站发现了一些信息，只能"精神上复制－粘贴"到他们想要组合信息的另一个网站。

1.1.2 语义网的设计方案

着手构建一个更"语义的"万维网有多种方式。一种方式可以是构建一个"巨型 Google"，依赖"数据不可思议的效力"[⊖]来发现诸如词语之间、术语和情境之间的正确关联。我们在过去几年中已经见证了搜索引擎性能的停滞，这似乎暗示了此种方法存在缺陷：没有一个搜索巨头能够超越仅返回分散页面的简单扁平列表的情况。

语义网（或近年来被逐渐熟知的数据万维网[⊜]）则遵循了不同的设计原则，可以概括如下：

1）使得结构化和半结构化的数据以标准化的格式在万维网上可用；

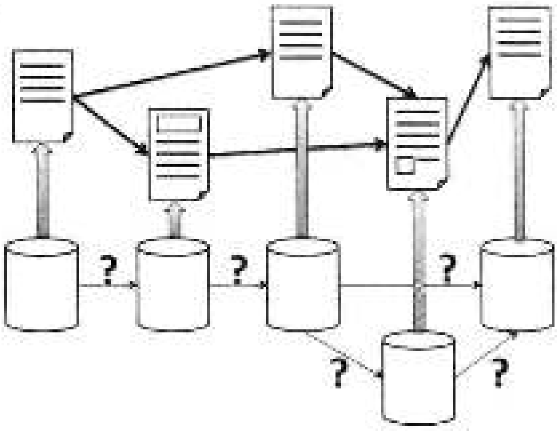
2）不仅制造数据集，还创建万维网上可解读的个体数据元素及其关系；

⊖ The Unreasonable Effectiveness of Data, Alon Halevy, Peter Norvig, and Fernando Pereira, IEEE Intelligence Systems, March/April 2009, pgs. 8-12, http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//pubs/archive/35179.pdf。

⊜ http://www.readwriteweb.com/archives/web_of_data_machine_accessible_information.php。

3）使用形式化模型来描述这些数据的隐含语义，使得这些隐含语义能够被机器处理。

2 决定利用结构化和半结构化数据基于一个关键的观察结论，即在当前无结构的"文本和图片万维网"之下实际上存在着大量结构化和半结构化数据。万维网的绝大部分内容正是从数据库和包含仔细结构化了的数据集的内容管理系统中产生的。然而，这些数据集中可用的富结构在结构化数据发布为人们可读的超文本标记语言（Hypertext Markup Language，HTML）页面的过程中几乎完全丢失了（参见图1-1）。一个关键认识在于，如果我们能发布和互联（interlink）底层的结构化数据集（而不仅是在底层结构丢失后发布和互联HTML页面），我们已经朝构建一个更加语义的万维网愿景迈进了一大步。



3 图1-1　万维网上的结构化和半结构化数据

1.1.3　语义网的基础技术

之前提到的3个设计原则已经被转化为实际的技术，而本书的大部分内容将致力于介绍这些技术。

1）使用带标签的图（labeled graph）作为对象及其关系的数据模型，图中将对象作为节点，对象间的关系表示为边。使用被草草命名为"资源描述框架"（Resource Description Framework，RDF）㊀的形式化模型来表示这种图结构。

2）使用万维网标识符（统一资源标识符（Uniform Resource Identifier，URI））来标识出现在数据集中的单个数据项以及它们之间的关系。这同样反映在RDF的设计中。

3）使用本体（ontology，简言之：类型和关系的层次化词汇表）作为数据模型来形式化地表达数据的隐含语义。诸如RDF模式（RDF schema）和万维网本体语言（Web Ontology Language，OWL）的形式化模型被用于该目的，同样也使用URI来表示类型和它们的属性。

1.1.4　从数据到知识

为了真正捕获数据的隐含语义，诸如RDF模式和OWL的形式化模型不仅是数据描述语

㊀ 可能"富数据格式"（Rich Data Format）是一个更好的名字。

言，实际上还是轻量级的知识表示（knowledge representation）语言。认识到这点很重要。它们是允许从显式声明的信息中推理出额外信息的"逻辑"。RDF 模式是一种表达能力很弱的逻辑，它允许一些非常简单的推理，例如在一个类型层次上的属性继承、定义域/值域的类型推理。类似地，OWL 是一种表达能力颇强（但依然相对轻量级）的逻辑，它允许更多的推理，例如等价和不等价、数量限制、对象的存在和其他。RDF 模式和 OWL 中的这些推理为 4
信息发布者提供了创建一个事实的最小下界的可能性，读者必须相信这些被发布的数据。此外，OWL 为信息发布者提供了禁止信息阅读者相信被发布数据的某些事情的可能性（至少只要每个人打算与被发布的本体保持一致性）。

综上所述，在这些逻辑上执行推理相当于对发布数据的隐含语义同时施加了下界和上界。通过逐步精炼这些本体，这些下界和上界能够任意地靠近，因此为了始终精确地确认数据的隐含语义，在一定程度上需要直接提供用例。

1.1.5　语义网的万维网体系结构

传统万维网的一个重要方面在于它的内容是分布式的。不仅位置上是，所有权上也是：相互链接的网页经常存在于不同的万维网服务器，这些服务器位于不同的物理位置并由不同的组织所有。对万维网发展起到关键作用的是"任何人可以说关于任何事的任何话"㊀，或更准确地说：任何人可以参考其他任何人的网页，而无须先协商允许，或征求合适的地址或标识符来使用。语义网也采用了类似的机制（参见图 1-2）：第一个组织可以在万维网上发布一个数据集（图 1-2 的左侧），第二个组织可以独立发布一个术语表（图 1-2 的右侧），而第三个组织可能会决定使用第二个组织发布的术语来标注第一个组织的对象，而无需经过他们中任何一方的允许，并且事实上这两个组织甚至都不知道这件事情。这种松耦合是语义网的万维网式特征的本质。 5

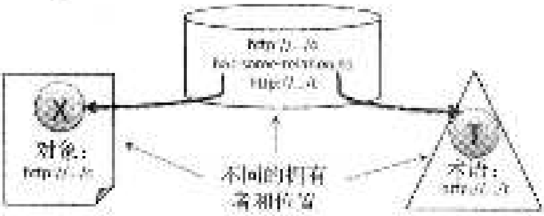


图 1-2　链接数据的万维网体系结构

1.1.6　如何由此及彼

当然，需要一些重要步骤来实现上述愿景以及将上述体系结构原则变为可行的现实。

1）我们必须同意使用标准的语法来表示数据和元数据。

2）我们必须对元数据词汇表取得足够的共识，使得可以分享数据的隐含语义。

3）我们必须使用第 1）步的格式和第 2）步的词汇表来发布大量的数据。

㊀ http://www.w3.org/DesignIssues/RDFnot.html。

20 世纪（最早的语义网项目始于 20 世纪的最后几年），以上 3 个步骤都取得了实质性进展：RDF、RDF 模式和 OWL（以及它们的变种，例如 RDFa、OWL2 等）已经获得了万维网联盟（World Wide Web Consortium，W3C）的正式支持，将它们提升为万维网上的事实标准。
6 数以千计的词汇表使用这些格式发布[⊖]，并且这些词汇表间的汇聚已经开始发生，这既是自动化本体映射技术，也是社会和经济需求施压的结果（例如，schema.org 词汇表的开发）[⊜]。此外，链接数据云（Linked Data Cloud）[⊛]的发展也使得数以十亿计使用共享的语法和词汇表的对象以及它们间的关系在线可用。

1.1.7 我们的现状

对比本书 2003 年出版第 1 版时的情况，许多基础构件已经就绪。许多快速成熟的技术支撑了语义网技术的所有部署阶段，商业领域和开放组织的真实案例的数目也在快速增长。但是，主要的挑战依然存在，例如应对持续增长的规模，降低使用的门槛，当然还有与信息系统中无所不在的"毒药"：语义异构性的斗争。

1.2 语义网技术

1.2.1 显式元数据

当前，万维网上内容的格式更适合人类读者而非计算机程序。HTML 是（直接或借助工具）撰写网页的主流语言。一个理疗师的典型网页的一部分可能如下所示：

```
<h1>Agilitas Physiotherapy Centre</h1>
Welcome to the Agilitas Physiotherapy Centre home page
Do you feel pain? Have you had an injury? Let our staff
Lisa Davenport, Kelly Townsend (our lovely secretary)
and Steve Matthews take care of your body and soul.
<h2>Consultation hours</h2>
Mon 11am - 7pm<br>
Tue 11am - 7pm<br>
Wed 3pm - 7pm<br>
Thu 11am - 7pm<br>
Fri 11am - 3pm<p>
But note that we do not offer consultation
during the weeks of the
<a href=". . .">State of Origin</a> games.
```

7

对人们而言，这些信息以一个令人满意的方式表达，但是对机器而言则存在问题。基于关键

⊖ http://swoogle.umbc.edu。

⊜ http://schema.org。

⊛ http://linkeddata.org。

词的搜索会识别 physiotherapy 和 consultation hours 等词。并且一个智能 agent 甚至能够识别该中心的全体人员。但是，区分治疗师和秘书则会存在困难，查找准确的咨询时间还会碰到更大的困难（因为不得不沿着 State of Origin 游戏的链接来查找它们发生的时间）。

解决这些问题的语义网方法不是开发一种超级智能的 agent，而是尝试从网页端入手。如果 HTML 能够被更适合的语言取代，网页就可以携带它们的内容。在包含为人类读者生成的文档格式信息的同时，网页还可以包含有关它们内容的信息。

在这个方向上的第一步是可扩展标记语言（eXtensible Markup Language，XML），它允许在网页上定义信息的结构。在该例子中，可能存在以下信息：

```
<company>
  <treatmentOffered>Physiotherapy</treatmentOffered>
  <companyName>Agilitas Physiotherapy Centre</companyName>        8
  <staff>
    <therapist>Lisa Davenport</therapist>
    <therapist>Steve Matthews</therapist>
    <secretary>Kelly Townsend</secretary>
  </staff>
</company>
```

这种表达形式非常容易被机器处理。特别是，它适用于万维网上的信息交换，这是 XML 技术最重要的一个应用领域。

但是，XML 依然处于语法层次，因为它描述的是信息的结构（structure），而非信息的含义（meaning）。语义网的基础语言是 RDF，它是一种生成有关信息片段声明的语言。在该例子中，这样的声明包括：

Company A offers physiotherapy.

The name of A is "Agitilitas Physiotherapy".

Lisa Davenport is a therapist.

Lisa Davenport works for A.

...

对于人类读者，XML 表示形式和一个 RDF 声明列表的差异微不足道，但是它们本质上非常不同：XML 描述的是结构而 RDF 生成的是信息片段的声明㊀。

元数据（metadata）一词的含义是：关于数据的数据。元数据捕获数据的含义（meaning） 9
部分，也就是语义网中所说的语义（semantic）一词。

1.2.2 本体

本体（ontology）一词源于哲学。在哲学中，它被用作一个哲学子领域的名称，即对存在

㊀ 人类读者通过选择标签名称将含义赋予 XML 表示形式，但是这对机器处理者而言还不够。

的本质的研究（希腊单词的 *Ontologia*），它属于形而上学的一个分支。以最通俗的术语而言，它关注于识别真实存在的事物的类别，以及如何描述它们。例如，世界是由具体事物构成的，这些事物根据共享的属性可以被归类到抽象的类别，这种观测是一种典型的本体化任务。

但是，近些年来，本体成为被计算机科学领域引进的许多单词之一，并且被赋予了一个与原始含义迥然不同的具体技术含义。我们现在使用“一个本体”来代替“本体”。针对我们的目标，我们将使用 T. R. Gruber 的定义，这个定义后来由 R. Studer 修订：*一个本体是一个概念体系的一种显式的、形式化的归约*（An ontology is an explicit and formal specification of a conceptualization）。

通常，一个本体形式化地描述了一个论域。典型地，一个本体由一个包含术语以及术语间的联系的有限列表组成。术语（term）指称领域中的重要概念（concept，对象的类（class））。例如，对于一个大学领域，教职工、学生、课程、阶梯教室和学科是一些重要概念。

一般地讲，联系（relationship）刻画了类的层次性。一个层次描述了一个类 *C* 是另一个类 *C'* 的子类，仅当 *C* 中的每个对象也属于 *C'*。例如，所有教师都是教职工。图 1-3 描述了大学领域的这样一个层次结构。

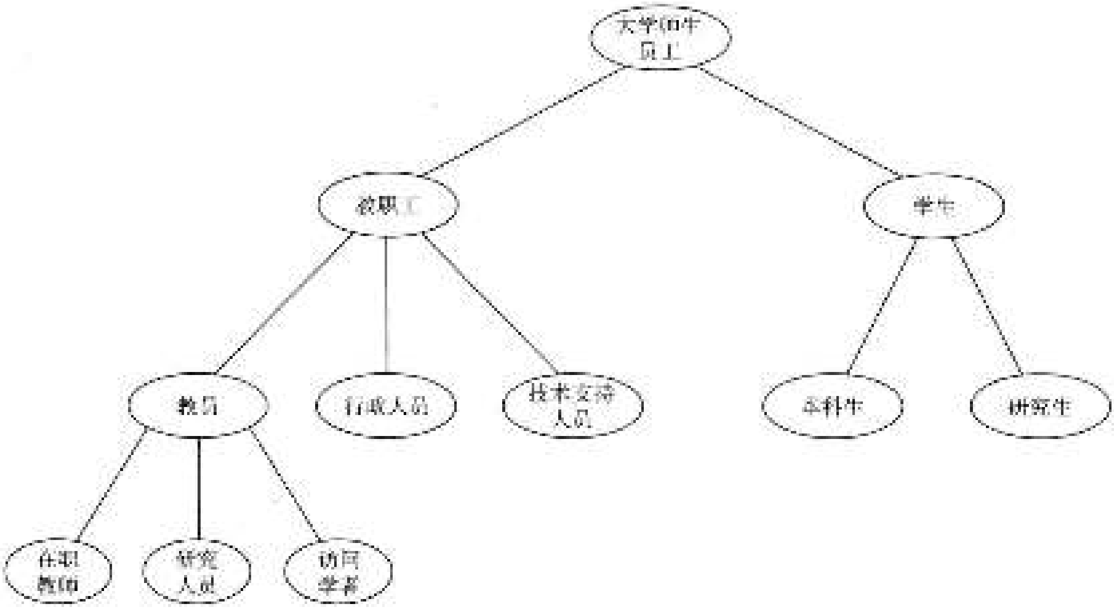


图 1-3 一个层次结构

除了子类联系之外，本体还可能包含以下信息：

- 属性（X teaches Y）；
- 值限制（只有教师才能授课）；
- 10 不相交声明（教师和普通教工是不相交的）；
- 对象间逻辑关系的说明（每个院系必须至少有 10 位教师）。

万维网环境下，本体提供了某个领域的一个共享的理解。该共享的理解对于克服术语上

的差异必不可少。例如，一个应用中的邮编（zip code）可能与另一个应用中的邮编（postcode）一样。另一个问题是，两个应用可能使用相同的术语来表示不同的含义。在大学 A 中，course 可能指的是一门学位（如计算机科学），而在大学 B 中则可能是一个科目（如 CS 101）。这种差异可以通过将特殊的术语映射到一个共享的本体或定义本体间的直接映射来克服。无论是何种情况，都可以明显看出本体提供了对语义互操作性的支持。

本体有助于网站的组织和导航。现在的许多网站在页面的左边呈现了一个术语的概念层次的顶层目录，用户可以点击它们来展开子目录。

同时，本体有助于提高万维网搜索的准确性。搜索引擎可以寻找指向一个本体中一个精确概念的页面，而不是通过某些含糊的关键词收集到的所有页面。通过这种方式，网页和查询之间术语上的差异可以克服。 11

此外，万维网搜索可以利用一般化 / 特殊化信息。如果一个查询无法找到任何相关文档，那么搜索引擎可以建议用户使用一个更一般的查询。甚至搜索引擎可以预先执行这样的查询来降低用户采用这一建议的响应时间。或者在获得太多的搜索结果时，搜索引擎可以建议用户使用更特殊的查询。

人工智能（Artificial Intelligence，AI）领域中，开发和使用本体语言已经有很长的历史。这是语义网研究可以依赖的基础。目前，万维网上最重要的本体语言包括：

- RDF 模式是一种词汇表描述语言，用来描述 RDF 资源的属性和类，以及这些属性和类的泛化层次的语义。此外，属性的定义域和值域也可以定义。
- OWL 是一种描述属性和类的更丰富的词汇表描述语言，例如类之间的关系（比如不相交）、基数（比如“恰好等于 1”）、相等、更加丰富的属性类型定义、属性的性质（比如对称性），以及枚举类等。

1.2.3 逻辑

逻辑是一门研究推理的原理的学科，它可以追溯到亚里士多德（Aristotle）。一般而言，逻辑首先提供*形式语言*（formal language）来表达知识。其次，逻辑为我们提供广泛理解的*形式语义*（well-understood formal semantics）：在绝大多数逻辑中，句子的含义不需要通过对知识的操作来定义。我们常提到描述性知识：我们描述什么是成立的，而不需要关心它如何被推断出来。 12

再者，自动化的推理机能够从给定知识中推断（推导）出结论，因而使得隐式的知识显式化。这些推理机已经在人工智能领域中广泛研究。这里有一个推导的例子。假设我们知道所有的教授都是教师，所有的教师都是教职工，而 Michael 是一名教授。在谓词逻辑中，这些信息可以表述如下：

$prof(X) \rightarrow faculty(X)$

$faculty(X) \rightarrow staff(X)$

$prof(michael)$

接下来，我们可以推断出下列知识：

$faculty(michael)$

$staff(michael)$

$prof(X) \rightarrow staff(X)$

注意，这个例子中包含了本体中常见的知识。因此，逻辑能够用于揭露隐式给定的本体知识。通过逻辑推理，也可以发现未知的联系和不一致性。

但是逻辑比本体更加宽泛。逻辑还可以被智能 agent 用于决策和选择动作路线。例如，一个购物 agent 可能决定基于下面的规则给某位顾客一个折扣：

$loyalCustomer(X) \rightarrow discount(X, 5\%)$

其中，顾客的忠诚度由存储在企业数据库中的数据确定。

通常，在表达能力和计算高效性之间存在一个权衡。一个逻辑的表达能力越强，它生成结论的计算代价就变得越昂贵。甚至当遇到不可计算性障碍时，生成某些结论将变得不可能。
13 幸运的是，和语义网相关的大量知识看起来是一种相对受限的形式。例如，之前的例子包含了规则的形式“如果前提，那么结论”，其中前提和结论是简单的声明，并且只有有限个对象需要被考察。这个逻辑子集称为 Horn 逻辑，是易处理的并且由高效的推理工具支持。

逻辑的一个重要的优点是它能够提供结论的解释（explanation）：推导步骤的序列是可回溯的。而且人工智能领域的研究人员已经开发出对人类友好的方式来展现解释。通过将一个证明组织成一个自然的推导过程，并将一组底层的推导步骤归类成人们通常认为的单个证明步骤的元步骤。最终解释将从答案回溯到一个给定的事实和使用的推理规则的集合。

解释对于语义网很重要，因为它们增加了用户对语义网 agent 的信心（参见之前的理疗师例子）。Tim Berners-Lee 提到过一个“Oh yeah?”按钮来请求解释。

解释对于 agent 之间的活动也必不可少。当一些 agent 有能力生成逻辑结论时，另一些则有能力验证证明（validate proof），即检查一个由其他 agent 生成的断言是否可以被证明。这里有一个简单的例子。假设 agent 1，代表一个在线商店，发送一条消息“你欠我 80 美元”（当然，不是以自然语言的形式，而是用一种形式化的、机器可处理的语言）给 agent 2（代表一个人）。然后，agent 2 可能会要求提供解释，而 agent 1 可能响应一个如下形式的序列：

购买超过 80 美元商品的万维网日志
商品交付证明（例如，UPS 的追踪号）
使用商店术语和条件的规则：

$$purchase(X, Item) \wedge price(Item, Price) \wedge delivered(Item, X) \rightarrow owes(X, Price)$$

14 这些事实通常将被追溯到某些网址（它们的可信性可以被 agent 验证），而这些规则可能是一

个共享的商业本体或者在线商店政策的一部分。

对于万维网上有用的逻辑，它必须能和其他数据一同使用，并且还必须可以被机器处理。因此使用万维网语言来表达逻辑知识和证明还在进行中。早期的方法工作在 XML 层次，但是在未来，规则和证明将需要表示在 RDF 和本体语言层次。

1.2.4 语义网与人工智能

正如我们所说的，实现语义网所需的大部分技术建立在人工智能领域工作的基础上。尽管人工智能拥有很长的历史，但它在商业上并不是一直成功的，人们或许会担心，最坏情况下，语义网将重蹈人工智能的覆辙：大的承诺导致过高的期待，最终发现无法实现（至少不是在许诺的时间范围内实现）。

这个担心是不公平的。实现语义网愿景并不依赖于人类层次的智能；事实上，正如我们尝试解释的那样，语义网是以一种不同的方式来接近挑战。人工智能的完整问题是一个深层次的科学问题，可能与物理学（解释物理世界）或者生物学（解释生物世界）的核心问题相当。正如所见，没有实现过去某些时候曾许诺的在 10~20 年里实现人类层次的人工智能，并没有令人感到吃惊。

但是在语义网上，部分解决方案是可行的。即使一个智能 agent 不能生成一个人类用户可以生成的所有结论，相比现在，这个 agent 仍然对万维网贡献良多。这为我们带来了另一个区别。如果人工智能的最终目标是建立一个表达人类层次智力（或更高层次）的智能 agent，语 15
义网的目标则是辅助人类用户参与日常在线活动。

显而易见，语义网频繁使用当前的人工智能技术，而人工智能技术的发展将带来一个更好的语义网。但是不必等到人工智能取得更高层次的成就，当前的人工智能技术已经足够在很长一段时间内实现语义网的愿景。

1.3 一种分层方法

语义网的发展按步骤进行，每步都在一个层次之上建立另一个层次。这个方法的实用性论证是它更容易在小的步骤上达成一致，也就是说，如果有太多的目标，则很难让参与的每个人都同意。通常许多研究组研究不同的方向，思想的竞争是科学进步的主要驱动力。但是，从工程的角度则需要标准化。因此，如果绝大多数研究人员同意某些事情而反对另一些，稳固这些共识则是有意义的。这样，即使一些更有野心的研究成果会失败，但是至少有部分正面的结果。

一旦一个标准被确立，许多组织和公司将采用它而不是等待着观察最终是否有可替代的研究路线会成功。语义网的本质在于公司和单个用户必须构建工具、添加内容并使用这些内容。我们无法等到完整的语义网愿景变成熟——这可能还要 10 年时间来完全实现它（当然，是以今天的想象）。

在一个语义网层次上构建另一个层次的过程中，必须遵循两个原则。

- 向下兼容性。完全理解某一层次的 agent 应该能够解释和使用更低层次的信息。例如，

16 理解 OWL 语义的 agent 能够充分利用 RDF 和 RDF 模式中的信息。

- 向上部分理解。设计应该使得完全理解某一层次的 agent 应该能够至少部分利用更高层次的信息。例如，一个仅理解 RDF 和 RDF 模式语义的 agent 可能能够通过忽略超出 RDF 和 RDF 模式的元素，部分地解释 OWL 中的知识。当然，并不是要求所有工具都提供这种功能性，关键在于允许这种选择。

虽然这些想法理论上很吸引人并且被作为指导原则来指导语义网的发展，实践上却存在困难，并且不得不采取某些妥协。这在第 4 章讨论 RDF 和 OWL 分层时会进一步说明。

17 图 1-4 展示了语义网的“分层蛋糕”，它描述了语义网设计和愿景的主要层次。在最底层我们可以发现 XML，一种允许用户使用用户定义的词汇表来撰写结构化万维网文档的语言。XML 特别适合于在万维网上发送文档。此外，用于 XML 中的 URI 可以按照它们的命名空间（namespace）聚类，在图中表示为 NS。

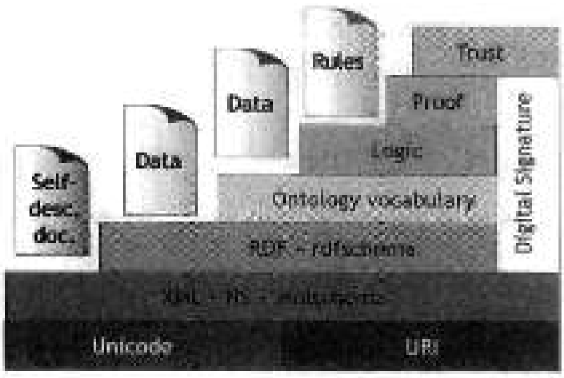


图 1-4 语义网的一个分层方法

RDF 是一个基本数据模型，正如实体－联系模型一样，它表达了万维网对象（资源）的声明。RDF 数据模型不依赖于 XML，但是 RDF 有一个基于 XML 的语法。因此，在图 1-4 中，RDF 位于 XML 层之上。

RDF 模式提供了将万维网对象组织为层次结构的建模原语。重要的原语包括类和属性，子类和子属性联系，以及定义域和值域约束。RDF 模式是基于 RDF 的。

RDF 模式可以被认为是一种表达本体的基本语言。但是还需要一些更加强大的本体语言（ontology language），它们扩展 RDF 模式并且允许表达万维网对象间更加复杂的联系。逻辑层用于进一步增强本体语言并允许描述面向特定应用的声明式知识。

证明层（proof layer）包括了实际的演绎过程，以及使用（更低层次的）万维网语言来表达证明和验证证明。

最后，信任层（trust layer）将伴随使用数字签名（digital signature）和知识的其他类别出现，基于可信 agent 的推荐或者基于评分和证书中介及消费体。有时，“信任网”用于表示信任将被组织成与万维网本身一样的分布式的、混乱的方式。位于金字塔的最顶层，信任是一个高层且重要的概念：当用户对其操作（安全性）和提供的信息的质量信任时，万维网才能实

现其所有潜力。

经典的分层蛋糕是语义网日程表上的主要驱动力，但是现在已经相当过时了。特别是本体词汇表层的许多替代者已经出现。另外规则语言已被定义在 RDF 之上，绕开了本体词汇表层，这特别适用于最近从丰富的语义结构向大规模（语义）数据处理的转变。因此这里给出的分层蛋糕主要是示例的目的，作为展现语义网历史观点的一种方式。 18

1.4 本书内容安排

在这本书中，我们关注的是已经达到一个合理成熟度的语义网技术。

第 2 章讨论 RDF 和 RDF 模式。RDF 是一种用于表达有关对象（资源）的声明的语言；它是一个标准的数据模型以提供机器可处理的语义。RDF 模式提供了一组用于将 RDF 词汇表组织成带类型的层次结构的建模原语。

第 3 章致力于介绍 SPARQL 查询语言，它在 RDF 领域中起到的作用与 SQL 在关系领域中的作用相同。

第 4 章讨论 OWL2，它是万维网本体语言 OWL 的最新修订版。OWL2 提供了比 RDF 模式更多的建模原语，并且它拥有清晰的、形式化的语义。

第 5 章致力于介绍语义网框架中的规则。虽然语义网上的规则还尚未达到和 RDF、SPARQL 或 OWL 相同级别的一致意见，但其被采用的原理却非常清楚，因此在此介绍规则也很有意义。

第 6 章介绍一些应用领域并解释语义网愿景的成熟所带来的好处。

第 7 章介绍有关为万维网开发基于本体的系统的各种关键问题。

第 8 章简要探讨目前语义网社区中正在争论的一些话题。 19

1.5 小结

- 语义网是一场旨在改进当前万维网状况的运动。
- 其核心思想是机器可处理的万维网信息的使用。
- 其核心技术包括发布带有显式元数据的数据、本体、逻辑和推理。
- 语义网的开发是分层次进行的。

建议阅读

一篇有关语义网愿景的优秀介绍性论文是：

- T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American* 284 (May 2001): 34–43.

一本有关万维网历史（和未来）的启发性图书是：

- T. Berners-Lee, with M. Fischetti. *Weaving the Web*. San Francisco: Harper, 1999.

一些维护有关语义网及相关主题最新信息的网站包括：

- www.semanticweb.org/.

- www.w3.org/2001/sw/.

20 一篇精选的研究性论文，它提供了语义网相关问题的技术信息：

- J. Domingue, D. Fensel, and J. Hendler. *Handbook of Semantic Web Technologies*. Springer, 2011.

语义网上最重要的想法、技术和应用最新进展的主要发表之处是：

- 系列会议 International Semantic Web Conference。参见 http://www.semanticweb.org/。
- 系列会议 Extended (European) Semantic Web Conference。参见 http://www.eswc.org/。
- 语义网杂志。参见 http://www.journals.elsevier.com/journal-of-web-semantics/。

一些专注于语义网的图书近些年来也已出现。其中

- D. Allemang and J. Hendler. *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*. New York, NY.: Morgan Kaufmann, 2008.

21–22 侧重于本体建模问题，而

- P. Hitzler, M. Kroetzsch, and S. Rudolph. *Foundations of Semantic Web Technologies*. Boca Raton, FL.: Chapman and Hall, 2009.

侧重于逻辑基础。

描述万维网资源：RDF

2.1 引言

万维网的成功展现了使用标准化的信息交换和通信机制的力量。HTML 是编写网页的标准语言。它允许任何人发布一个文档并且相信该文档可以被任何万维网浏览器正确呈现。

HTML 和其他交换语言都拥有以下 3 个组成元素：语法、数据模型和语义。语法告诉我们如何撰写数据。数据模型告诉我们数据的结构或组织形式。语义告诉我们如何解释数据。我们可以通过下面的 HTML 片段展现上述每个组成元素：

```
<html>
    <head>
        <title>Apartments for Rent</title>
        </head>
    <body>
        <ol>
                <li> Studio apartment on Florida Ave.
                <li> 3 bedroom Apartment on Baron Way
        </ol>
    </body>
</html>
```

23

HTML 的语法是用尖括号撰写的带标签文本（例如 `<title>`）。HTML 的数据模型称为文档对象模型（Document Object Model），将由标签定义的元素组织成一个层次树状结构。例如，`<head>` 应该在 `<body>` 之前，而 `<li>` 元素应该包含在 `<ol>` 元素内。最后，HTML 的语义告诉我们浏览器应该如何解释网页。例如，浏览器应该将网页体的内容呈现在浏览器窗口中，同时 `<ol>` 元素应该呈现为一个有序列表。语法、数据模型和语义都在 HTML 标准中定义。

HTML 用于传递有关面向人类的文档结构的信息。而对于语义网，我们的需求更加丰富。我们需要一个能够被各种应用使用的数据模型，不仅为人类描述文档，而且为特定应用描述信息。这个数据模型需要是领域无关（domain independent）的，因此从房地产到社交网络的应用都可以使用它。除了一个灵活的数据模型之外，我们还需要一种机制来将语义赋予使用这个数据模型表达的信息。它应该允许用户描述一个应用该如何在一个社交网络描述中解释

"朋友"，以及如何在一个地理信息描述中解释"城市"。最终，和HTML类似，我们需要将所有这些信息写下来，即语法。

24 RDF（资源描述框架）恰好提供了这样一个灵活并且领域无关的数据模型。它的基础构件是一个实体–属性（attribute[⊖]）–取值的三元组，称为声明（statement）。例如，我们可以使用这个模型来表达"The Baron Way Apartment is an Apartment"、"The Baron Way Apartment is part of The Baron Way Building"以及"The Baron Way Building is located in Amsterdam"。因为RDF不针对任何领域及使用，对用户而言必须定义他们在这些声明中使用的术语。为此，需要利用RDF模式（RDFS）。RDFS允许用户精确地定义它们的词汇表（vocabulary，即它们的术语）应该如何解释。

综合起来，这些技术定义了在不同机器间交换任意数据的一种标准化语言的组成部分：

- RDF——数据模型
- RDFS——语义
- Turtle / RDFa / RDF-XML——语法

尽管RDF主要是指数据模型，它也经常被用来作为上述所有的总称（本书中也会这样使用）。

本章概述

- 2.2节介绍RDF。
- 2.3节介绍RDF使用的各种语法。
- 2.4节介绍RDF模式的基础知识，完整的语言将在2.5节和2.6节中介绍。
- 2.7节和2.8节将从两个方面介绍RDFS的形式化含义。

2.2 RDF：数据模型

25 RDF中的基本概念包括资源、属性、声明和图。

2.2.1 资源

我们可以认为一个资源是一个对象，我们希望谈论的一个"事物"。资源可以是作者、公寓、网球选手、地点、人、旅馆、查询，等等。每个资源都有一个URI。一个URI可以是一个URL（（Uniform Resource Locator，统一资源定位符），或网址）或者另一种唯一的标识符。URI机制不仅被定义为万维网上的位置，还可以是电话号码、ISBN号和地理位置。URI提供了一种机制来无歧义地标识我们想要谈论的一个"事物"。因此，如果想指称一个游泳池（pool），我们可以为其分配一个URI，使它不会和撞球（billiard，pool的另一个词义）或者一群程序员（the pool of programmers）混淆。这称为一词多义问题。

使用URI不必能访问（access）到一个资源。但是使用可以解引用的（dereferenceable）URL作为资源标识符被认为是一种好的做法。它使得用户既可以获取资源本身（比如一张图

⊖ 本书中，attribute和property都翻译为"属性"，通常可根据上下文判断。在可能引发误解的地方，"属性"默认指的是property。当指的是attribute时，会在括号中附加英文说明。——译者注

片），也可以获取资源的进一步描述（比如一个人）。这种做法将贯穿本书。使用 URI 是 RDF 背后的一个关键设计方案。它允许全球唯一的命名方案的存在。使用这种机制能够大幅缓解迄今为止困扰分布式数据表示的一词多义问题。

2.2.2 属性

属性是一类特殊的资源，它们描述了资源之间的关系。例如，“friend of”、“written by”和“located in”。和其他资源一样，属性也由 URI 标识。我们也可以解引用属性的 URL 来找到它们的描述。

2.2.3 声明

声明断言了资源的属性。一个声明是一个实体－属性（attribute）－取值的三元组，由一个资源、一个属性和一个属性值组成。属性值要么是一个资源，要么是一个文字（literal）。文字是原子值，例如，数字、字符串或日期。我们经常使用主语一词来指称三元组里的实体，而使用宾语来指称其取值。 26

例如，对于声明“Baron Way Building is located in Amsterdam”，我们可以这么写：

```
<http://www.semanticwebprimer.org/ontology/apartments.ttl#BaronWayBuilding>
<http://dbpedia.org/ontology/location>
<http://dbpedia.org/resource/Amsterdam>.
```

注意，我们在这个声明中如何使用 URL 来标识我们指称的事物。

2.2.4 图

我们也可以使用图形化的方式来书写相同的声明。注意，为了增加可读性，我们在图中不使用 URI。

如图 2-1 所示，带标签的节点通过带标签的边连接。边是有向的，从声明的主语到声明的宾语，声明的属性被标记在边上。节点上的标签是主语和宾语的标识符。一个声明的宾语可以是另一个声明的主语。例如，我们可以说“Amsterdam is a city”。我们可以在图 2-2 中看见这个图形化结果。 27

图 2-1 一个图形化表示的 RDF 声明

这种图形化表示强调了 RDF 是一个以图为中心的数据模型这一概念。事实上，RDF 和人工智能领域中的语义网络（semantic net）类似。我们能够继续扩展有关 Baron Way Building 信息的图。图 2-3 展示了 RDF 图的一个扩展版本。

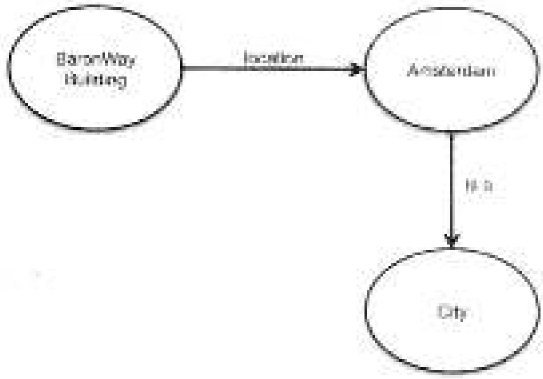


图 2-2　一个 RDF 图

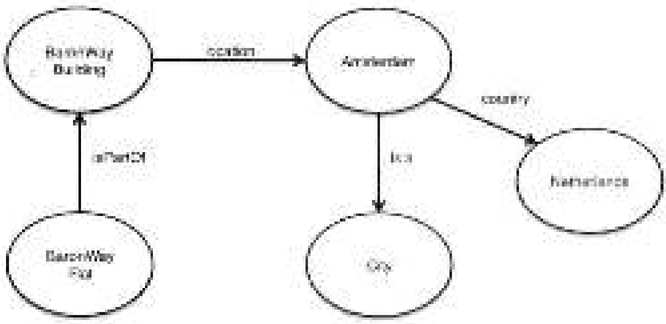


图 2-3　一个扩展的 RDF 图

重要的是，这个图可以以一种分布式的方式，由多个不同参与人使用相同的 URL 来创
28 建。这使得我们可以创建一个允许知识被重用的数据万维网（Web of Data）。例如，如果我们在万维网上发现描述 Amsterdam 的 RDF，我们可以仅通过使用该 URL 来重用这些信息。事实上，有一组称为链接数据原则（Linked Data principle⊖）的最佳实践，鼓励我们重用和使信息可用来帮助创建一个全局的图。

1）使用 URI 作为事物的名称。

2）使用 HTTP URI，以便人们可以查询到这些名称。

3）当某人查询一个 URI 时，使用标准（RDF）来提供有用的信息。

4）包含到其他 URI 的链接，以便可以发现更多事物。

虽然 RDF 数据模型不要求我们必须遵循这些原则，但是通过这样做可以使我们从他人贡献的知识中获益。注意，在该例子中，我们已经重用了 DBpedia.org 提供的信息。你可以沿着这些 URL 来发现更多关于所指称的概念的信息。

2.2.5　指向声明和图

有时能够指向特定的声明或图的某些部分是很有用的，例如当赋予一个声明一个信念度

⊖ http://www.w3.org/DesginIssues/LinkedData.html。

时，或者标识一个声明从哪里来时。例如，我们可能想要描述一个关于 Baron Way Building 的位置的声明是由一个叫 Frank 的人创建的。RDF 提供了两种实现机制。

一种称为具体化（reification）。具体化背后的关键思想是引入一个额外的对象，例如 *LocationStatement*，并将它和原来声明中的三个部分通过属性 *subject*、*predicate* 和 *object* 关 29
联。在之前的例子中，*LocationStatement* 的主语是 *BaronWayBuilding*，谓语是 *location*，而宾语是 *Amsterdam*。随后我们可以将这个声明作为另一个三元组的主语来定义创建者。图 2-4 描述了生成的结果图。同样出于展示的原因，完整的 URI 没有在图 2-4 中给出。

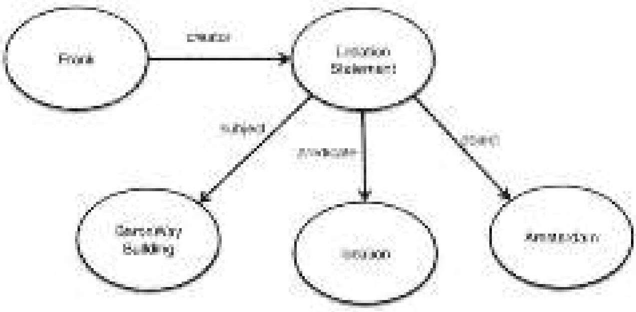


图 2-4　一个具体化的例子

这种相对笨重的方式是必要的，因为 RDF 中只存在三元组，因此我们不能直接为三元组添加一个标识符（否则就变成了四元组）。正是由于具体化的昂贵代价，在较新版本的 RDF 标准中引入了命名图的概念。这时，一个显式的标识符（同样是 URL）被赋予一个声明或声明集合。然后这个标识符就可以在普通的三元组中引用。这是一种更加直接地定义声明和图的机制。简言之，一个命名图允许圈出一个 RDF 声明的集合并为这些声明提供一个标识符。2.3.1 节第一部分内容提供了一个使用命名图来表示上述具体化声明的例子。 30

2.2.6　处理更丰富的谓语

我们可以将一个三元组 (x, P, y) 想成一个逻辑公式 $P(x, y)$，其中二元谓词 P 关联对象 x 到对象 y。事实上 RDF 只提供了二元谓词（属性）。但是在一些情况下，我们可能需要谓语包含超过两个参数。幸运的是，这样的谓语可以通过一组二元谓词来模拟。我们通过一个包含 3 个参数的谓语 *broker* 来展现这个技术。*broker*(X, Y, Z) 的直观含义是

X 是房屋出售人 Y 和购买人 Z 之间的中介。

现在介绍一个新的额外的资源 *home-sale* 以及二元谓词 *broker*、*seller* 和 *purchaser*。接下来，可以将 *broker*(X, Y, Z) 表达为如下形式：

broker(home-sale, X)

seller(home-sale, Y)

purchaser(home-sale, Z)

虽然拥有 3 个参数的谓语书写更简洁，使用二元谓词确实简化了整个数据模型。

2.3 RDF 语法

我们已经看到了一种 RDF 语法，即图形化的语法。但是这种语法既不是机器可解释的，也不是标准化的。本节介绍一种标准的机器可解释的语法，称为 *Turtle*，另外还将简要介绍一些其他语法。

2.3.1 Turtle

Turtle（Terse RDF Triple Language）是一种基于文本的 RDF 语法。Turtle 文本文件使用的
31 后缀名是“.ttl”。我们之前已经见到如何使用 Turtle 写一个声明。下面是一个例子：

```
<http://www.semanticwebprimer.org/ontology/apartments.ttl#BaronWayBuilding>
<http://dbpedia.org/ontology/location>
<http://dbpedia.org/resource/Amsterdam>.
```

URL 包含在尖括号中。一个声明的主语、属性和宾语依次出现，由句号结尾。事实上，我们能够仅使用这种方法来编写整个 RDF 图。

```
<http://www.semanticwebprimer.org/ontology/apartments.ttl#>
    <http://www.semanticwebprimer.org/ontology/apartments.ttl#isPartOf>
    <http://www.semanticwebprimer.org/ontology/apartments.ttl#BaronWayBuilding>.
<http://www.semanticwebprimer.org/ontology/apartments.ttl#BaronWayBuilding>
    <http://dbpedia.org/ontology/location>
    <http://dbpedia.org/resource/Amsterdam>.
```

1. 文字

到目前为止，我们已经定义了将资源链接在一起的声明。正如之前讨论的那样，我们也能在 RDF 中引入文字，即原子值。在 Turtle 中，我们简单地将值写在引号中，并附上值的数据类型。数据类型告诉我们是否应该将一个值解释为字符串、日期、整型数，还是其他类型。数据类型也使用 URL 表达。实践中建议使用 XML 模式定义的数据类型。当使用这些数据类型时，值必须服从 XML 模式定义。如果一个文字之后没有指定数据类型，则假设数据类型是字符型。下面是一些常见的数据类型以及它们在 Turtle 中的形式：

32

```
string - "Baron Way"
integers - "1"^^<http://www.w3.org/2001/XMLSchema#integer>
decimals - "1.23"^^<http://www.w3.org/2001/XMLSchema#decimal>
dates - "1982-08-30"^^<http://www.w3.org/2001/XMLSchema#date>
time - "11:24:00"^^<http://www.w3.org/2001/XMLSchema#time>
date with a time -
    "1982-08-30T11:24:00"^^<http://www.w3.org/2001/XMLSchema#dateTime>
```

假设我们想要向图中添加 Baron Way Apartment 有 3 个房间，可以将下面 Turtle 中的声明

添加到图中。

```
<http://www.semanticwebprimer.org/ontology/apartments.ttl#BaronWayApartment>
<http://www.semanticwebprimer.org/ontology/apartments.ttl#hasNumberOfBedrooms>
"3"^^<http://www.w3.org/2001/XMLSchema#integer>

<http://www.semanticwebprimer.org/ontology/apartments.ttl#BaronWayApartment>
<http://www.semanticwebprimer.org/ontology/apartments.ttl#isPartOf>
<http://www.semanticwebprimer.org/ontology/apartments.ttl#BaronWayBuilding>.

<http://www.semanticwebprimer.org/ontology/apartments.ttl#BaronWayBuilding>
<http://dbpedia.org/ontology/location>
<http://dbpedia.org/resource/Amsterdam>
```

上述例子相对而言不容易使用。为使其更加清晰，Turtle提供了一些构造子来使书写变得更加容易。

2. 缩写

当我们定义词汇表时，我们经常在相同的URI上定义。在例子中，资源Baron Way Apartment和Baron Way Building都定义在了http://www.semanticwebprimer.org/ontology/apartments.ttl这个URL下。这个URL定义了这些资源的*命名空间*（namespace）。Turtle使用了这个惯例，允许URL被缩写。它引入了`@prefix`语法来定义特定命名空间的替代形式。例如，可以用`swp`作为http://www.semanticwebprimer.org/ontology/apartment.ttl的替代形式。这种替代称为*限定名*（qualified name）。以下使用前缀重写了该例子。 33

```
@prefix swp:       <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.
@prefix xsd:      <http://www.w3.org/2001/XMLSchema#>.

swp:BaronWayApartment swp:hasNumberOfBedrooms "3"^^<xsd:integer>.
swp:BaronWayApartment swp:isPartOf swp:BaronWayBuilding.
swp:BaronWayBuilding dbpedia-owl:location dbpedia:Amsterdam.
```

注意，使用限定名来指称资源时，资源两侧的尖括号被去掉了。其次，我们能够混用这些限定名与常规URL并匹配它们。

Turtle还允许在我们重复使用某些主语的时候不需要再重复书写。在上面的例子中，`swp:BaronWayApartment`被用作两个三元组的主语。这可以通过在一个声明的结尾处使用一个分号来使书写更加紧凑。例如：

```
@prefix swp:        <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.
@prefix xsd:      <http://www.w3.org/2001/XMLSchema#>.

swp:BaronWayApartment  swp:hasNumberOfBedrooms "3"^^<xsd:integer>;
                                       swp:isPartOf swp:BaronWayBuilding.
swp:BaronWayBuilding dbpedia-owl:location dbpedia:Amsterdam.
```

如果主语和谓语都被重复使用，我们可以在声明的结尾处使用一个逗号。例如，如果希望扩展该例子，说明 Baron Way Building 不仅位于 Amsterdam 还位于 Netherlands，可以在
34 Turtle 中这样写：

```
@prefix swp:        <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.
@prefix xsd:      <http://www.w3.org/2001/XMLSchema#>.

swp:BaronWayApartment swp:hasNumberOfBedrooms "3"^^<xsd:integer>;
swp:isPartOf swp:BaronWayBuilding.
swp:BaronWayBuilding dbpedia-owl:location dbpedia:Amsterdam,
                                                             dbpedia:Netherlands.
```

最后，Turtle 允许我们简写常见的数据类型。例如，数字可以不使用引号来写。如果数字包含一个小数点（例如 14.3），那么它们就被解释为小数。如果它们不包含一个小数点（例如 1），那么它们就被解释成整数。这更加简化了该例子：

```
@prefix swp: <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.

swp:BaronWayApartment swp:hasNumberOfBedrooms 3;
                                    swp:isPartOf swp:BaronWayBuilding.
swp:BaronWayBuilding dbpedia-owl:location dbpedia:Amsterdam,
                                 dbpedia:Netherlands.
```

3. 命名图

我们之前讨论了指向一组声明的能力。Trig 是 Turtle 的一个扩展，它允许我们表达这个
35 概念。例如，我们可能想说，关于 Baron Way Apartment 的声明是由一个叫 Frank 的人创建的，他通过 http://www.cs.vu.nl/~frankh 来标识。为了实现它，我们将一组想要的声明用花括号括起来并赋予这组声明一个 URL。让我们看下面的例子：

```
@prefix swp: <http://www.semanticwebprimer.org/ontology/apartments.ttl#>
@prefix dbpedia: <http://dbpedia.org/resource/>
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>
@prefix dc: <http://purl.org/dc/terms/>

{
    <http://www.semanticwebprimer.org/ontology/apartments.ttl#>
    dc:creator <http://www.cs.vu.nl/~frankh>
}

<http://www.semanticwebprimer.org/ontology/apartments.ttl#>
{
    swp:BaronWayApartment swp:hasNumberOfBedrooms 3;
      swp:isPartOf swp:BaronWayBuilding.
    swp:BaronWayBuilding dbpedia-owl:location dbpedia:Amsterdam,
      dbpedia:Netherlands.
}
```

在这个方法中，位于花括号中但是之前没有 URL 的声明不是一个特定图的一部分，它称为默认图。

2.3.2 其他语法

除了 Turtle 之外，还存在其他一些可用于编写 RDF 的语法。其中有两个标准的语法：RDF/XML 和 RDFa。 36

1. RDF/XML

RDF/XML 是 RDF 在 XML 语言中的编码。它允许 RDF 被已有 XML 处理工具使用。起初，RDF/XML 是 RDF 的唯一语法。但是，由于 Turtle 通常更容易阅读，所以作为一种额外标准被采纳。下面展示了一个 RDF/XML。主语在一个 `rdf:Description` 元素中通过 `rdf:about` 定义（包含在尖括号内）。与主语关联的谓语和宾语也包含在 `rdf:Description` 元素中。命名空间可以通过 XML 命名空间结构（`xmlns:`）被使用。所有的 RDF/XML 必须被包含在一个 `rdf:RDF` 元素中。

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF xmlns:dbpedia-owl="http://dbpedia.org/ontology/"
  xmlns:dbpedia="http://dbpedia.org/resource/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:swp="http://www.semanticwebprimer.org/ontology/apartments.ttl#">
 <rdf:Description
  rdf:about="http://www.semanticwebprimer.org/ontology/apartments.ttl#BaronWayApartment">
  <swp:hasNumberOfBedrooms
```

```
    rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
        3
    </swp:hasNumberOfBedrooms>
  </rdf:Description>
  <rdf:Description
   rdf:about="http://www.semanticwebprimer.org/ontology/apartments.ttl#BaronWayApartment">
    <swp:isPartOf
     rdf:resource="http://www.semanticwebprimer.org/ontology/apartments.ttl#BaronWayBuilding"/>
  </rdf:Description>
  <rdf:Description
   rdf:about="http://www.semanticwebprimer.org/ontology/apartments.ttl#BaronWayBuilding">
37    <dbpedia-owl:location
        rdf:resource="http://dbpedia.org/resource/Amsterdam"/>
  </rdf:Description>
  <rdf:Description
   rdf:about="http://www.semanticwebprimer.org/ontology/apartments.ttl#BaronWayBuilding"
    <dbpedia-owl:location
      rdf:resource="http://dbpedia.org/resource/Netherlands"/>
  </rdf:Description>
</rdf:RDF>
```

2. RDFa

RDF 的一个用例是描述或标注 HTML 网页的内容。为了使其更加简单，引入 RDFa 语法来帮助实现这个用例。RDFa 在 HTML 标签的属性（attribute）中嵌入 RDF。我们使用 Baron Way Apartment 的一个广告作为例子。

```
<html>
<body>
<H1> Baron Way Apartment for Sale</H1>
The Baron Way Apartment has three bedrooms and is located in the fam
ily friendly Baron Way Building. The Apartment is located in the north of Amsterdam.
</body>
</html>
```

这个页面并不包含任何机器可读的描述。可以使用如下 RDFa 来标注这个页面：

```
<html xmlns:dbpedia="http://dbpedia.org/resource/"
    xmlns:dbpediaowl="http://dbpedia.org/ontology/"
    xmlns:swp="http://www.semanticwebprimer.org/ontology/apartments.ttl#"
    xmlns:geo="http://www.geonames.org/ontology#">
38 <body>
<H1> Baron Way Flat for Sale</H1>
```

```
<div about="[swp:BaronWayFlat]">
The Baron Way Flat has <span property="swp:hasNumberOfBedrooms">3</span> bed-
rooms and is located in the family friendly <span rel="swp:isPartOf" re-
source="[swp:BaronWayBuilding]">Baron Way Building</span>

<div about="[swp:BaronWayBuilding]">
The building is located in the north of Amsterdam.
  <span rel="dbpediaowl:location" resource="[dbpedia:Amsterdam]"></span>
  <span rel="dbpediaowl:location" resource="[dbpedia:Netherlands]"></span>
</div>

</div>
</body>
</html>
```

这个标注将生成和之前 Turtle 表示的完全一样的 RDF。因为 RDF 被编码在诸如 span、paragraph 和 link 之类的标签中，所以在显示 HTML 页面时不会被浏览器解析。和 RDF/XML 类似，命名空间使用 `xmlns` 声明来编码。一些情况下，必须使用方括号来告诉解析器我们正在使用前缀。主语通过 `about` 属性（attribute）来标识。属性通过 `rel` 或 `property` 属性（attribute）来标识。当一个声明的宾语是一个资源时使用 `rel`，而当一个声明的宾语是文字时使用 `property`。谓语和主语通过使用 HTML 层次结构来关联。

上述每种 RDF 语法适用于不同的场景。然而必须意识到尽管可能会使用不同的语法，但它们的底层数据模型和语义是相同的。迄今为止我们已经讨论了如何通过 URL 标识来编写事物的声明。但是这些声明的含义是什么？一个计算机应该如何解释这些生成的声明？这些问题将在下一节介绍 RDF 的模式语言时讨论。 39

2.4 RDFS：添加语义

RDF 是一种通用语言，它允许用户使用他们自己的词汇表来描述资源。RDF 既不假设与任何特定应用领域有关，也不定义任何领域的语义。为了指明语义，一个开发者或用户需要通过 RDF 模式中定义的一组基本的领域无关的结构来定义其词汇表的含义。

2.4.1 类和属性

如何描述一个特定领域？让我们考虑一下公寓租赁领域。首先我们必须指定想描述的"事物"。这里首先做一个基本区分。一方面，我们希望描述特定的公寓，例如 Baron Way Apartment，以及特定的地点，例如 Amsterdam，这些之前已经在 RDF 中完成了。

另一方面，我们还想描述公寓、建筑、国家、城市，等等。区别是什么？第一种情况中我们描述的是个体对象（individual object，资源），而第二种情况中我们描述的是类（class），它们定义了对象的类型。

一个类可以被理解为一个元素集合。属于一个类的个体对象称为该类的实例（instance）。RDF 给我们提供了一种通过使用一个特殊属性 rdf:type 来定义实例和类之间联系的方式。

类的一个重要用法是，施加限制来约束在一个 RDF 文档中使用模式可以声明什么。在编
40 程语言中，类型定义（typing）用来阻止无意义的编程（例如 $A+1$，其中 A 是一个数组，而我们要求"+"的参数必须是数字）。RDF 中也需要相同的功能。之后，我们可能会不允许如下声明：

Baron Way Apartment rents Jeff Meyer

Amsterdam has number of bedrooms 3

第一个声明是无意义的，因为建筑不能出租人。这在属性"rents"的取值上施加了限制。用数学语言来说，我们限制了属性的值域（range）。

第二个声明也是无意义的，因为城市不能拥有房间。这在可以应用属性的对象上施加了限制。用数学语言来说，我们限制了属性的定义域（domain）。

2.4.2 类层次和继承

一旦有了类，我们就希望建立它们之间的联系。例如，假设已经有了如下类：

unit

residential unit　　commercial unit

house& apartment　office

这些类不是互不关联的。例如，每个 residential unit（居住单元）都是一个 unit。我们称"residential unit"是"unit"的一个子类（subclass），或等价地说"unit"是"residential unit"的一个超类（superclass）。子类联系定义了类的一个层次，如图 2-5 所示。通常，如果 A 的每个实例也是 B 的实例，则 A 是 B 的子类。RDF 模式不要求所有的类形成一个严格的层次结构。换句话说，图 2-5 所示的一个子类图不必是一棵树。一个类可能有多个超类。如果一个类 A
41 同时是 B_1 和 B_2 的子类，这简单意味着 A 的每个实例同时是 B_1 的实例和 B_2 的实例。

类的一个层次化组织拥有非常重要的现实意义，我们在这里阐述。考虑值域限制

People can only rent residential units.

假设 Baron Way Apartment 被定义为一个 apartment。接下来，根据前面的限制，它并没有被限定为是一个 residential unit，因为不存在一个声明来指定 Baron Way Apartment 也是一个 residential unit。通过为描述添加上述声明来解决这个问题不符合直觉想象。相反，我们希望 Baron Way Apartment 继承（inherit）residential unit 允许被出租的能力。这正是 RDF 模式所做的。

通过这样做，RDF 模式修复了"is a subclass of"的语义。现在不是根据应用来解释"is a subclass of"，而是它的特定含义必须被所有 RDF 处理软件所使用。通过创建这样的语义定义，RDFS 是一种（能力依然受限的）定义特定领域语义的语言。换句话说，RDF 模式是一种

基本的本体语言。

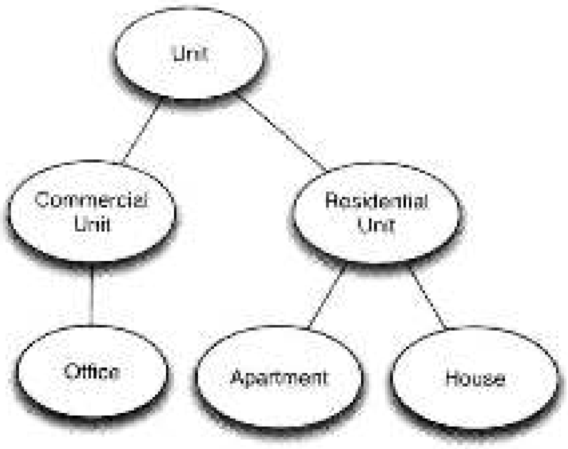


图 2-5　一个类的层次

42

当然，类、继承和属性也常见于计算的其他领域中，例如在面向对象编程中。但是尽管有很多相似性，也存在许多不同之处。在面向对象编程中，一个对象类定义了可以应用于它的属性。为一个类添加新的属性意味着修改这个类。

但是在 RDFS 中，属性是全局定义的，也就是说，它们没有作为属性（attribute）被封装在类定义中。可以在一个已有的类上定义应用的新属性，而不需要修改这个类。

一方面，这是一个后果深远的有力机制：我们可以使用他人定义的类，并通过新的属性来调整它们以适合我们的需求。另一方面，属性的处理方式脱离了源自建模和面向对象编程领域中的标准做法。它是 RDF/RDFS 中的一个特有特征。 43

2.4.3 属性层次

我们已经看见了，可以定义类之间的层次联系。属性间也可以采用相同的方式。例如，"rents" 是 "resident at" 的子属性（subproperty）。如果一个人 p 租了一个居住单元 r，则 p 也居住在 r。相反的关系则不成立。例如，p 可能是一个家庭中的儿童，p 不支付租金，或者 p 可能只是一个访客。

总之，某个属性 P 是 Q 的子属性，仅当 $P(x, y)$ 成立时 $Q(x, y)$ 总成立。

2.4.4 RDF 和 RDFS 的分层对比

最后，我们使用一个简单的例子来说明 RDF 和 RDFS 涉及的不同分层。考虑下面的 RDF 声明

Jeff Meyer rents the Baron Way Apartment.

这个声明的模式可能包含人、公寓、房屋、单元等类，以及出租、居住在、地址等属性。图 2-6 展现了这个例子的 RDF 层次和 RDF 模式层次。其中，方块是属性，虚线以上的圆圈是类，而虚线以下的圆圈是实例。

图2-6中的模式本身也使用一种形式语言（RDF模式）来编写，可以表达的组成元素有：subClassOf、Class、Property、subPropertyOf、Resource等。接下来我们将更详细地介绍RDF模式。

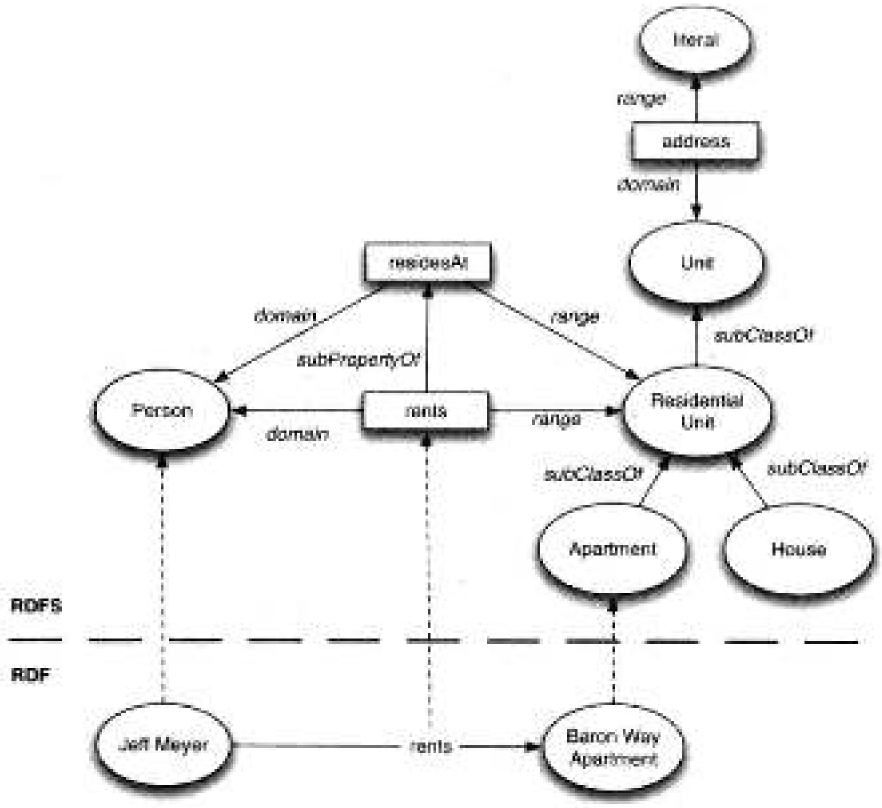


图2-6 RDF层和RDFS层

2.5 RDF模式：语言

44
:
45 RDF模式提供建模原语来表达2.5节中的信息。一个必须做的决定是使用什么样的形式语言。使用RDF本身并不让人吃惊：RDF模式的建模原语使用资源和属性定义。通过观察图2-6可以验证这个决定。我们将这个图表达为一个类/属性的层次以及实例，但是它本身当然也仅仅是一个带标签的图，可以用RDF编码。记住RDF允许表达有关任何资源的声明，并且任何拥有URI的事物都可以作为资源。因此，如果想让类"apartment"是"residential unit"的一个子类，我们可能会

1）为apartment、residential unit和subClassOf定义所需的资源；

2）定义subClassOf为一个属性；

3）编写三元组（apartment subClassOf residential unit）。

所有这些步骤都包含在RDF的表达能力中。因此，一个RDFS文档仅仅是一个RDF文

档，而我们可以使用某种 RDF 标准语法来编写。

现在，我们定义 RDF 模式的建模原语。

2.5.1 核心类

核心类包括：

rdfs:Resource，所有资源的类。

rdfs:Class，所有类的类。

rdfs:Literal，所有文字（字符串）的类。

rdf:Property，所有属性的类。

rdf:Statement，所有具体化声明的类。

2.5.2 定义联系的核心属性

用来定义联系的核心属性包括：

rdf:type，将一个资源关联到它的类（参见 2.4.1 节）。该资源被声明为该类的一个实例。 46

rdfs:subClassOf，将一个类关联到它的超类。一个类的所有实例都是它的超类的实例。注意，一个类可能是多个类的子类。例如，类 femaleProfessor 可能同时是 female 和 professor 的子类。

rdfs:subPropertyOf，将一个属性关联到它超属性中的一个。

这里有一个例子，表达了所有的公寓都是居住单元：

```
swp:apartment rdfs:subClassOf swp:ResidentialUnit
```

注意，rdfs:subClassOf 和 rdfs:subPropertyOf 被定义为传递的。并且很有趣的是，rdfs:Class 是 rdfs:Resource 的一个子类（所有的类都是资源），同时 rdfs:Resource 是 rdfs:Class 的一个实例（rdfs:Resource 是所有资源的类，因此它是一个类！）出于同样的原因，每个类都是 rdfs:Class 的实例。

2.5.3 限制属性的核心属性

用来限制属性的核心属性包括：

rdfs:domain，指定一个属性 P 的定义域，声明任何拥有某个给定属性的资源是定义域类的一个实例。

rdfs:range，指定一个属性 P 的值域，声明一个属性的取值是值域类的实例。

这里有一个例子声明当任何资源有一个地址时，它（通过推理）是一个单元并且取值是一个文字：

```
swp:address rdfs:domain swp:Unit
swp:address rdfs:range rdf:Literal
```
47

2.5.4 对具体化有用的属性

下面是一些对具体化有用的属性：

rdf:subject，将一个具体化声明关联到它的主语。

rdf:predicate，将一个具体化声明关联到它的谓语。

rdf:object，将一个具体化属性关联到它的宾语。

2.5.5 容器类

RDF 还允许用一个标准的方式表达容器。可以表达包、序列或选择。

rdf:Bag，包的类。

rdf:Seq，序列的类。

rdf:Alt，选择的类。

rdfs:Container，所有容器类的超类，包括前面提到的 3 种。

2.5.6 效用属性

一个资源可以在万维网上的许多地方被定义和描述。下列属性允许我们定义链接到这些地址：

rdfs:seeAlso，将一个资源关联到另一个解释它的资源。

rdfs:isDefinedBy，它是 rdfs:seeAlso 的一个子属性，将一个资源关联到它的定义之处，一般是一个 RDF 模式。

48 为人类读者提供更多的信息常常很有用。这可以通过以下属性来实现：

rdfs:comment，注释，一般是长的文本，可以与一个资源关联。

rdfs:label，将一个人类友好的标签（名字）与一个资源关联。其中的一个目的是用作 RDF 文档的图形化表示中节点的名称。

2.5.7 示例：住房供给

我们展示一个住房供给的例子，并提供该领域的概念模型，即一个本体。

```
@prefix swp: <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.

swp:Person rdf:type rdfs:Class.
swp:Person rdfs:comment "The class of people".

swp:Unit rdf:type rdfs:Class.
swp:Unit rdfs:comment "A self-contained section of accommodations
in a larger building or group of buildings.".

swp:ResidentialUnit rdf:type rdfs:Class.
swp:ResidentialUnit rdfs:subClassOf swp:Unit.
swp:ResidentialUnit
```

```
    rdfs:comment "The class of all units or places where people live.".

swp:Apartment rdf:type rdfs:Class.
swp:Apartment rdfs:subClassOf swp:ResidentialUnit.
swp:Apartment rdfs:comments "The class of apartments".
swp:House rdf:type rdfs:Class.
swp:House rdfs:subClassOf swp:ResidentialUnit.
swp:House rdfs:comment "The class of houses".

swp:residesAt rdf:type rdfs:Property.
swp:residesAt rdfs:comment "Relates persons to their residence".
swp:residesAt rdfs:domain swp:Person.
swp:residesAt rdfs:range swp:ResidentialUnit.

swp:rents rdf:type rdfs:Property.
swp:rents rdfs:comment "It inherits its domain (swp:Person)
  and range (swp:ResidentialUnit) from its superproperty (swp:residesAt)".
swp:rents rdfs:subPropertyOf swp:residesAt.

swp:address rdf:type rdfs:Property.
swp:address rdfs:comment "Is a property of units and takes literals as its value".
swp:address rdfs:domain swp:Unit.
swp:address rdfs:range rdf:Literal.
```

49

2.5.8 示例：汽车

这里，我们介绍一个简单的汽车本体。图 2-7 展示了它的类层次。

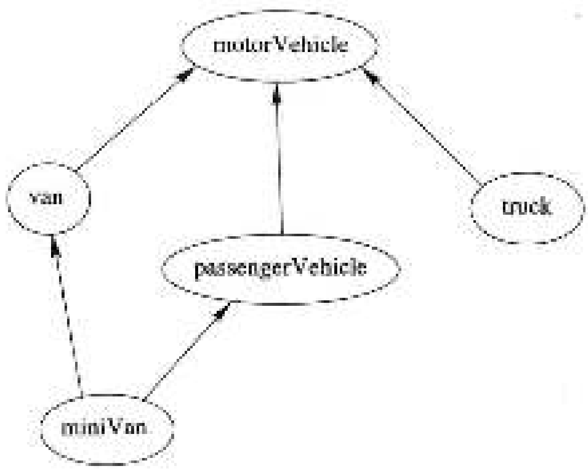


图 2-7 汽车示例的类层次

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

<#miniVan> a rdfs:Class ;
   rdfs:subClassOf <#passengerVehicle>, <#van> .

<#motorVehicle> a rdfs:Class .

<#passengerVehicle> a rdfs:Class ;
   rdfs:subClassOf <#motorVehicle> .

<#truck> a rdfs:Class ;
   rdfs:subClassOf <#motorVehicle> .

<#van> a rdfs:Class ;
   rdfs:subClassOf <#motorVehicle> .
```

50 / 51

2.6 RDF 和 RDF 模式的定义

现在我们已经了解了 RDF 和 RDFS 语言的主要构成，看看 RDF 和 RDFS 的定义可能很有意义。这些定义使用 RDF 模式语言来表达。一个任务是在每个元素的含义都已经清晰的基础上，观察它们现在如何能够被简单地理解。

下面的定义仅仅是完整语言规范的一部分，剩余部分可以在 `rdf:RDF` 命名空间里看到。我们使用原始的 XML 语法来给出它们。

2.6.1 RDF

```
<?xml version="1.0" encoding="UTF-16"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

 <rdfs:Class rdf:ID="Statement"
  rdfs:comment="The class of triples consisting of a
              predicate, a subject and an object
              (that is, a reified statement)" />

 <rdfs:Class rdf:ID="Property"
  rdfs:comment="The class of properties"/>
```

```
<rdfs:Class rdf:ID="Bag"
 rdfs:comment="The class of unordered collections"/>

<rdfs:Class rdf:ID="Seq"
 rdfs:comment="The class of ordered collections"/>

<rdfs:Class rdf:ID="Alt"
 rdfs:comment="The class of collections of alternatives"/>
<rdf:Property rdf:ID="predicate"
 rdfs:comment="Identifies the property used in a statement
              when representing the statement
              in reified form">
 <rdfs:domain rdf:resource="#Statement"/>
 <rdfs:range rdf:resource="#Property"/>
</rdf:Property>

<rdf:Property rdf:ID="subject"
 rdfs:comment="Identifies the resource that a statement is
              describing when representing the statement
              in reified form">
 <rdfs:domain rdf:resource="#Statement"/>
</rdf:Property>

<rdf:Property rdf:ID="object"
 rdfs:comment="Identifies the object of a statement
              when representing the statement
              in reified form"/>

<rdf:Property rdf:ID="type"
 rdfs:comment="Identifies the class of a resource.
              The resource is an instance
              of that class."/>

</rdf:RDF>
```

2.6.2 RDF 模式

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
```

```
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<rdfs:Class rdf:ID="Resource"
rdfs:comment="The most general class"/>

<rdfs:Class rdf:ID="comment"
 rdfs:comment="Use this for descriptions">
 <rdfs:domain rdf:resource="#Resource"/>
 <rdfs:range rdf:resource="#Literal"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Class"
 rdfs:comment="The concept of classes.
              All classes are resources.">
 <rdfs:subClassOf rdf:resource="#Resource"/>
</rdfs:Class>

<rdf:Property rdf:ID="subClassOf">
 <rdfs:domain rdf:resource="#Class"/>
 <rdfs:range rdf:resource="#Class"/>
</rdf:Property>

<rdf:Property rdf:ID="subPropertyOf">
 <rdfs:domain rdf:resource="&rdf;Property"/>
 <rdfs:range rdf:resource="&rdf;Property"/>
</rdf:Property>

</rdf:RDF>
```

52 ? 54

这些命名空间并没有提供 RDF 和 RDF 模式的完整定义。例如考虑 `rdfs:subClassOf`。命名空间只是指定它被用于类，且取值是一个类。作为一个子类的含义，即一个类的所有实例也是它的超类的实例，没有在任何地方体现。事实上，在一个 RDF 文档中这是无法表达的。如果可以表达，就没有必要定义 RDF 模式了。

我们下一节介绍一个形式语义。显然，RDF 解析器和其他 RDF 软件工具（包括查询处理器）必须理解完整的语义。

2.7 RDF 和 RDF 模式的公理化语义

本节中，我们形式化定义 RDF 和 RDF 模式的建模原语的含义。因而，可以获得 RDF 和 RDFS 的语义。

我们使用的形式语言是谓词逻辑（predicate logic），它作为所有（符号）知识表示的基础被广泛接受。在形式化过程中使用的公式被称为公理（axiom）。

通过使用形式语言（如逻辑形式语言）来描述 RDF 和 RDFS 的语义，使得语义没有二义性并且机器可存取。另外，我们通过自动推理机操纵逻辑公式的能力提供了对推理的支持。

2.7.1 方法

RDF 和 RDF 模式中的所有语言原语被表示成常量：*Resource*、*Class*、*Property*、*subClassOf*，等等。一些预先定义的谓词被用作表达这些常量之间联系的基础。

使用链表的一个辅助理论，它包括下面的函数符号

nil（空链表）

cons(x, l)（添加一个元素到链表的头部） 55

first(l)（返回第一个元素）

rest(l)（返回链表的剩余部分）

和谓词符号

item(x, l)（取值为真，当且仅当一个元素出现在链表中）

list(l)（取值为真，当且仅当 *l* 是一个链表）

链表在 RDF 中被用于表达容器。它也在更丰富的本体语言中用于表达某些结构的含义（例如基数约束）。

大多数公理提供了类型信息。例如

$Type(subClassOf, Property)$

表示 *subClassOf* 是一个属性。我们使用带等式的谓词逻辑（predicate logic with equality）。变量名以 ? 开头。所有公理都隐式地全称量化。

下面我们展示 RDF 和 RDF 模式中大多数元素的定义。完整语言的公理化语义可以在一个在线文档中查看，参见建议阅读中的文档 Fikes 和 McGuinness (2001)。

2.7.2 基本谓词

基本谓词包括：

PropVal(P, R, V)，一个包含 3 个参数的谓词，它被用于表达一个包含资源 *R*、属性 *P* 和取值 *V* 的 RDF 声明。

Type(R, T)，是 *PropVal(Type, R, T)* 的缩写，它指出一个资源 *R* 的类型是 *T*。

$$Type(?r, ?t) \longleftrightarrow PropVal(type, ?r, ?t)$$ 56

2.7.3 RDF

一个 RDF 声明（三元组）(*R*, *P*, *V*) 表示为 *PropVal(P, R, V)*。

1. 类

在我们的语言中，我们有常量 *Class*、*Resource*、*Property* 和 *Literal*。所有的类都是 *Class*

的实例，即它们的类型是 *Class*：

$$Type(Class, Class)$$

$$Type(Resource, Class)$$

$$Type(Property, Class)$$

$$Type(Literal, Class)$$

Resource 是最一般的类：每个对象都是一个资源。因此，每个类和每个属性都是一个资源：

$$Type(?p, Property) \longrightarrow Type(?p, Resource)$$

$$Type(?c, Class) \longrightarrow Type(?c, Resource)$$

最后，一个 RDF 声明中的谓语必须是一个属性：

$$PropVal(?p, ?r, ?v) \longrightarrow Type(?p, Property)$$

2. 类型属性

type 是一个属性：

$$Type(type, Property)$$

注意，它和 *PropVal(type, type, Property)* 等价：*type* 的类型是 *Property*。*type* 可以用于资源，并且取值是一个类：

57 $$Type(?r, ?c) \longrightarrow (Type(?r, Resource) \wedge Type(?c, Class))$$

3. 辅助的 FuncProp 属性

一个函数型属性是一个函数：它将一个资源关联到最多一个值。函数型属性不是一个 RDF 的概念，但是它用于其他原语的公理化。

常量 *FuncProp* 表示所有函数型属性的类。*P* 是一个函数型属性，当且仅当它是一个属性，并且不存在 x、y_1 和 y_2，满足 $P(x, y_1)$、$P(x, y_2)$，且 $y_1 \neq y_2$。

$$Type(?p, FuncProp) \longleftrightarrow$$

$$(Type(?p, Property) \wedge \forall ?r \forall ?v1 \forall ?v2$$

$$(PropVal(?p, ?r, ?v1) \wedge PropVal(?p, ?r, ?v2) \longrightarrow ?v1 = ?v2))$$

4. 具体化声明

常量 *Statement* 表示所有具体化声明的类。所有具体化声明都是资源，同时 *Statement* 是 *Class* 的一个实例：

$Type(?s, Statement) \longrightarrow Type(?s, Resource)$

$Type(Statement, Class)$

一个具体化声明可以分解成一个 RDF 三元组的 3 个部分：

$Type(?st, Statement) \longrightarrow$

$\exists ?p \exists ?r \exists ?v (PropVal(Predicate, ?st, ?p) \wedge$

$PropVal(Subject, ?st, ?r) \wedge PropVal(Object, ?st, ?v))$

Subject、*Predicate* 和 *Object* 是函数型属性。也就是说，每个声明只能有一个主语、一个谓语和一个宾语：

$Type(Subject, FuncProp)$

$Type(Predicate, FuncProp)$

$Type(Object, FuncProp)$ 58

它们的类型信息是

$PropVal(Subject, ?st, ?r) \longrightarrow$

$(Type(?st, Statement) \wedge Type(?r, Resource))$

$PropVal(Predicate, ?st, ?p) \longrightarrow$

$(Type(?st, Statement) \wedge Type(?p, Property))$

$PropVal(Object, ?st, ?v) \longrightarrow$

$(Type(?st, Statement) \wedge (Type(?v, Resource) \vee Type(?v, Literal)))$

最后一个公理的含义是，如果 *Object* 作为属性出现在一个 RDF 声明中，那么它必须应用到一个具体化声明上，并且取值是一个资源或者一个文字。

5. 容器

所有的容器都是资源：

$Type(?c, Container) \longrightarrow Type(?c, Resource)$

容器是链表：

$Type(?c, Container) \longrightarrow list(?c)$

容器可以是包、序列或者选择：

$Type(?c, Container) \longleftrightarrow$

$(Type(?c, Bag) \vee Type(?c, Seq) \vee Type(?c, Alt))$

包和序列是不相交的：

$$\neg(Type(?x, Bag) \wedge Type(?x, Seq))$$

对于每个自然数 $n>0$，存在一个选择器 $\_n$，它选择一个容器中的第 n 个元素。它是一个
59 函数型属性

$$Type(\_n, FuncProp)$$

并且只用于容器：

$$PropVal(\_n, ?c, ?o) \longrightarrow Type(?c, Container)$$

2.7.4 RDF 模式

1. 子类和子属性

`subClassOf` 是一个属性：

$$Type(subClassOf, Property)$$

如果一个类 C 是另一个类 C' 的子类，那么 C 的所有实例也都是 C' 的实例：

$$PropVal(subClassOf, ?c, ?c') \longleftrightarrow$$
$$(Type(?c, Class) \wedge Type(?c', Class) \wedge$$
$$\forall ?x(Type(?x, ?c) \longrightarrow Type(?x, ?c')))$$

对于 `subPropertyOf` 也类似。P 是 P' 的一个子属性，仅当有 $P(x, y)$ 时必有 $P'(x, y)$：

$$Type(subPropertyOf, Property)$$

$$PropVal(subPropertyOf, ?p, ?p') \longleftrightarrow$$
$$(Type(?p, Property) \wedge Type(?p', Property) \wedge$$
$$\forall ?r \forall ?v(PropVal(?p, ?r, ?v) \longrightarrow PropVal(?p', ?r, ?v)))$$

2. 约束

每个约束资源是一个资源：

$$PropVal(subClassOf, ConstraintResource, Resource)$$

60 约束属性是属性，也是约束资源：

$$Type(?cp, ConstraintProperty) \longleftrightarrow$$
$$(Type(?cp, ConstraintResource) \wedge Type(?cp, Property))$$

`domain` 和 `range` 是约束属性：

$Type(domain, ConstraintProperty)$

$Type(range, ConstraintProperty)$

domain 和 range 分别定义了一个属性的定义域和值域。回想一个属性 P 的定义域是 P 可以应用在的一个对象集合。如果 P 的定义域是 D，那么对于任意 $P(x, y)$，有 $x \in D$。

$PropVal(domain, ?p, ?d) \longrightarrow$

$\forall ?x \forall ?y (PropVal(?p, ?x, ?y) \longrightarrow Type(?x, ?d))$

一个属性 P 的值域是 P 可以取的值的集合。如果 P 的值域是 R，那么对于任意 $P(x, y)$，有 $y \in R$。

$PropVal(range, ?p, ?r) \longrightarrow$

$\forall ?x \forall ?y (PropVal(?p, ?x, ?y) \longrightarrow Type(?y, ?r))$

从上述公式可以推导出：

$PropVal(domain, range, Property)$

$PropVal(range, range, Class)$

$PropVal(domain, domain, Property)$

$PropVal(range, domain, Class)$

因此，我们已经形式化定义了 RDF 和 RDFS 的语义。装配这种知识的软件能够产生有趣的结论。例如，给定 *rents* 的值域是 *ResidentialUnit*，而 *ResidentialUnit* 是 *Unit* 的一个子类，有 *rents*(*JeffMeyer*, *BaronWayApartment*)，那么 agent 可以使用谓词逻辑语义或任何一个谓词逻辑证明系统来自动地演绎出 *Unit*(*BaronWayApartment*)。 61

2.8 RDF 和 RDFS 的一个直接推理系统

如前所述，在 2.7 节中详细介绍的公理化语义可以用作 RDF 和 RDF 模式的自动化推理。但是，它需要一阶逻辑证明系统来实现。这是一个很重的需求，并且当涉及数以百万计（亿计）的声明时很难扩展（例如，数以百万计的 *Type*(?*r*, ?*c*) 形式的声明）。

出于这个原因，RDF 也被赋予了一个基于 RDF 三元组的语义（以及针对这种语义的完备的推导系统）。取代基于一阶逻辑来重述 RDF，正如在 2.7 节中介绍过的公理化语义。

推导系统包含如下形式的规则：

IF　E 包含某些三元组
THEN 添加一些额外的三元组到 E

（其中 E 是任意一个 RDF 三元组的集合）。

这里不再重复所有的推导规则（这些规则可以在官方的 RDF 文档中找到），在这里给出几

个基本的例子：

IF　*E* 包含三元组 (?*x*, ?*p*, ?*y*)
THEN *E* 也包含三元组 (?*p*, `rdf:type`, `rdf:Property`)

这表明任何一个用于一个三元组的属性位置的资源 ?*p* 可以推导出它是 `rdf:Property` 类的一个成员。

一个或许更加有趣的例子是下面的规则：

IF　*E* 包含三元组 (?*u*, `rdfs:subClassOf`, ?*v*) 和 (?*v*, `rdfs:subClassOf`, ?*w*)
THEN *E* 也包含三元组 (?*u*, `rdfs:subClassOf`, ?*w*)

这体现了子类关系的传递性。

62 相关的规则包括：

IF　*E* 包含三元组 (?*x*, `rdf:type`, ?*u*) 和 (?*u*, `rdfs:subClassOf`, ?*v*)
THEN *E* 也包含三元组 (?*x*, `rdf:type`, ?*v*)

这定义了 `rdfs:subClassOf` 含义的本质。

最后一个例子常令第一次了解 RDF 模式的人感到惊讶：

IF　*E* 包括三元组 (?*x*, ?*p*, ?*y*) 和 (?*p*, `rdfs:range`, ?*u*)
THEN *E* 也包含三元组 (?*y*, `rdf:type`, ?*u*)

这个例子说明任何一个资源 ?*y* 作为一个属性 ?*p* 的值出现时，它可以被推导出是 ?*p* 的值域的一个成员。这说明 RDF 模式中的值域定义不是用来约束一个属性的值域，而是推导值域的成员关系。

规则闭集的总数不超过几十个，并且无需经验丰富的定理证明技术来高效实现。

2.9 小结

- RDF 为表示和处理机器可理解的数据提供了基础。
- RDF 使用基于图的数据模型。它的核心概念包括资源、属性、声明和图。一个声明是一个资源－属性－值的三元组。
- RDF 拥有三种标准语法（Turtle、RDF/XML 和 RDFa）来支持语法层的互操作性。
- 63 RDF 使用分布式思想，允许递增式的构建知识，以及知识的共享和复用。
- RDF 是领域无关的。RDF 模式提供了一种描述特定领域的机制。
- RDF 模式是一种基本的本体语言。它提供一组具有固定含义的建模原语。RDF 模式的核心概念有类、子类关系、属性、子属性关系，以及定义域和值域限制。

建议阅读

下面列出了一些官方的在线文档：

- B. Adida and M. Birbeck. RDFa Primer: Bridging the Human and Data Webs.W3C Working Group Note. October 4, 2008.
  http://www.w3.org/TR/xhtml-rdfa-primer/.
- D. Beckett, ed. RDF/XML Syntax Specification (Revised). W3C Recommendation.February 10, 2004.
  http://www.w3.org/TR/rdf-syntax-grammar/.
- D. Beckett, T. Berners-Lee, and E. Prud'hommeaux, eds. Turtle: Terse RDF Triple Language. W3C Working Draft. August 9, 2011.
  http://www.w3.org/TR/turtle/.
- D. Brickley and R.V. Guha, eds. RDF Vocabulary Description Language 1.0:RDF Schema. W3C Recommendation. February 10, 2004.
  http://www.w3.org/TR/rdf-schema/.
- R. Fikes and D. McGuinness. An Axiomatic Semantics for RDF, RDF Schema and DAML+OIL. March 2001.
  http://www.daml.org/2001/03/axiomatic-semantics.html.
- P. Hayes, ed. RDF Semantics. W3C Recommendation. February 10, 2004.
  http://www.w3.org/TR/rdf-mt/. 64
- G. Klyne and J. Carroll, eds. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation. February 10, 2004.
  http://www.w3.org/TR/rdf-concepts/.
- F. Manola and E. Miller, eds. RDF Primer. W3C Recommendation. February 10, 2004.
  http://www.w3.org/TR/rdf-primer/.
- E. Prud'hommeaux and A. Seaborne, eds. SPARQL Query Language for RDF. W3C Candidate Recommendation. June 14, 2007.
  http://www.w3.org/TR/rdf-sparql-query/.

下面是一些有用的进阶阅读：

- C. Bizer and Richard Cyganiak. SPARQL The Trig Syntax.
  http://www4.wiwiss.fu-berlin.de/bizer/trig/.

练习和项目

1. 列出 10 个标识你所处环境中的事物的 URI。

2. 给出如何扩展 RDF 中文字定义的例子。为什么这样做很有用？

3. 阅读 RDFS 命名空间，尝试理解本章中没有介绍的元素。

4. RDFS 允许为一个属性定义超过一个定义域，并使用这些定义域的交集。讨论采用定义域的并集和交集的优缺点。

5. 在早期的 RDFS 规范中，`rdfs:subClassOf` 不允许有环。试想一下，什么情况下，

65 一个类联系的循环是有益的（提示：考虑类的等价关系）。

6. 讨论下面声明的区别，并用图的形式画出这些区别：

X 支持某个提案；Y 支持这个提案；Z 支持这个提案。

X、Y 和 Z 的全体支持这个提案。

7. 使用前面的公理来推导 2.7 节末的公式。

8. 讨论为什么 RDF 和 RDFS 不允许逻辑矛盾。任何 RDF/S 文档是一致的，因此它拥有至少一个模型。

9. 尝试将关系数据库模型映射到 RDF。

10. 比较实体－联系模型和 RDF。

11. 用 RDF 模式来描述图书馆的一部分：图书、作者、出版商、年代、册数、日期，等等。接下来用 RDF 写出一些声明。使用 Turtle 语法描述并使用一个检验器来确保你的 RDF 语法上有效。参见 http://librdf.org/parse 或 http://www.rdfabout.com/demo/validator/。

12. 写一个有关地理位置的本体：城市、国家、首都、边界、州，等等。

13. 写一个关于你自己的简单网页。标识网页中的概念和关系，并建立一个小的本体来表达它们。如果可能，使用一个已经存在的本体。用这个本体标注你的网页。

接下来，你要考虑 RDFS 的限制。具体地，什么应该在 RDF 模式中被真正表达，并且是否能够被表达？这些限制将与第 4 章相关，在第 4 章中我们将介绍一种更加丰富的建模语言。

66 1）考虑男人和女人类。命名一个应该被包含在本体中的它们之间的联系。

2）考虑人、男人和女人这 3 个类。命名一个应该被包含在本体中的它们三者之间的联系。这个联系的哪些部分可以在 RDF 模式中表达？

3）假设我们宣称 Bob 和 Peter 都是 Mary 的父亲。显然这里存在一个语义错误。语义模型应该如何做才能消除这个错误？

4）“is child of” 和 “is parent of” 之间的联系是什么？

5）考虑属性 *eats*，它的定义域是 *animal*，值域是 *animal* 或 *plant*。假设我们定义一个
67:68 新类 *vegetarian*（素食者）。为这个类在属性 *eats* 上命名一个合适的约束。你认为通过使用 `rdfs:range` 可以在 RDF 模式中表达这种约束吗？

查询语义网

在上一章中，我们学会了如何使用 RDF 来表示知识。当信息被表示为 RDF 后，出于推理和应用开发的需要，我们需要能够存取其中相关的部分。在本章中，我们将把注意力集中于一个叫做 SPARQL 的查询语言，它能够让我们通过选择、抽取等方式很容易地从被表示为 RDF 的知识中获得特定的部分。SPARQL 是专为 RDF 设计的，适合并依赖于万维网上的各种技术。如果你熟悉诸如 SQL 等数据库查询语言，你会发现 SPARQL 和它们有很多相似之处。即使你不熟悉——本章也不假设你已熟悉，本章将为你从头开始学习提供一切所需。

本章概述

在本章的开头，先讨论使 SPARQL 查询得以执行的基础设施（即软件）。接下来，讨论 SPARQL 的基础知识，并逐步介绍其更复杂的部分。最后，我们集中讨论从语义网上收集 RDF 数据的各种方式。

3.1 SPARQL 基础设施

想要执行一条 SPARQL 查询，就需要一个能执行查询的软件。能做到这一点的最常用的软件叫做*三元组存储库*（triple store）。本质上，一个三元组存储库就是一个 RDF 的数据库。在网上可以下载到很多三元组存储库。在 SPARQL 的相关规范中，三元组存储库也称为图存储库。

在查询一个三元组存储库之前，需要先向其中填充 RDF 数据。大部分三元组存储库都提供批量上传的选项。也有一种称为 SPARQL 更新的机制，提供了一系列向三元组存储库中插入、加载及删除 RDF 的选项。本章中将稍后讨论 SPARQL 更新。

当数据被加载进三元组存储库之后，就可以使用 SPARQL 协议来发送 SPARQL 查询去查询了。每个三元组存储库都提供一个*端点*（endpoint），在此提交 SPARQL 查询。重要的一点是，客户端使用 HTTP 协议向端点发送查询。事实上，要将一条 SPARQL 查询提交给一个端点，其实可以将它输入浏览器地址栏中！当然，我们还是建议使用一个专门为 SPARQL 设计的客户端。网上也能找到很多。

因为 SPARQL 使用标准的万维网技术，你在网上将会找到大量的 SPARQL 端点。这些端点可以让你存取大量的数据。例如，dbpedia.org/sparql 提供了一个查询端点来查询一份维基百科的 RDF 表示。在 CKAN.org 中可以找到一份完整的 SPARQL 端点清单。

一旦我们有了这个基础设施，就可以开始写 SPARQL 查询了。

3.2 基础知识：匹配模式

70 回顾一下上一章中描述 Baron Way 公寓及其位置的 RDF 数据：

```
@prefix swp: <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
@prefix dbpedia: <http://dbpedia.org/resource/>
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.

swp:BaronWayApartment swp:hasNumberOfBedrooms 3;
                      swp:isPartOf swp:BaronWayBuilding.
swp:BaronWayBuilding dbpedia-owl:location dbpedia:Amsterdam,
                     dbpedia:Netherlands.
```

我们可能想要在这段数据上做一个查询。例如，找到这幢建筑的位置。怎样用 SPARQL 来表述呢？可以如下构建这个查询。我们想要匹配的是下面这个三元组：

```
swp:BaronWayBuilding dbpedia-owl:location dbpedia:Amsterdam.
```

在 SPARQL 中，我们可以将三元组中的任何一个元素替换为一个变量。变量的首字符是一个？（问号）。要引入一个变量表示位置，我们可以这样写：

```
swp:BaronWayBuilding dbpedia-owl:location ?location.
```

三元组存储库将接收这个图模式（graph pattern）[1]并尝试去找到能够匹配这个模式的那些三元组集合。因此，在之前的 RDF 数据上运行这个模式，一个三元组存储库将会返回 `dbpedia:Amsterdam` 和 `dbpedia:Netherlands`。本质上，它找到了所有以 `swp:BaronWayBuilding` 作为主语、`dbpedia-owl:location` 作为谓语的三元组。

要构建一个完整的 SPARQL 查询，还需要增加一些内容。首先，需要定义所有的前缀。还需要告诉三元组存储库我们对一个特定变量的结果感兴趣。因此，上述查询对应的完整

71 SPARQL 查询如下：

```
PREFIX swp:    <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
PREFIX dbpedia: <http://dbpedia.org/resource/>.
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>.
SELECT ?location
WHERE {
      swp:BaronWayBuilding dbpedia-owl:location ?location.
   }
```

与 Turtle 类似，PREFIX 关键词指明各种 URL 的缩写。SELECT 关键词表明了哪些变量是感兴趣的。需要被匹配的图模式出现在 WHERE 关键词之后的括号中。返回的查询结果是

[1] 本章中，pattern 和 schema 都被翻译为"模式"，通常可根据上下文判断。在可能引发误解的地方，"模式"默认指的是 pattern。当指的是 schema 时，会在括号中附加英文说明。——译者注

一组称作绑定（binding）的映射，表示了哪些元素对应到一个给定的变量。表格中的每一行是一条结果或一个绑定。因此，这条查询的结果如下：

?location
http://dbpedia.org/resource/Amsterdam.
http://dbpedia.org/resource/Netherlands.

SPARQL 的全部基础就是这个简单的概念：尝试去找到能够匹配一个给定图模式的那些三元组集合。SPARQL 提供了更多的功能，用来指定更加复杂的模式并以不同的方式提供结果；但是无论模式多么复杂，运用的过程都是一样的。再举一个例子，查找到 BaronWayApartment 的位置。对应的 SPARQL 查询是：

```
PREFIX swp:     <http://www.semanticwebprimer.org/ontology/apartments.ttl#>
PREFIX dbpedia: <http://dbpedia.org/resource/>.
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>.
SELECT ?location
WHERE {
  swp:BaronWayApartment swp:isPartOf ?building.   72
  ?building dbpedia-owl:location ?location.
}
```

我们已经扩展了图模式。在这个查询中有一些注意点：首先，变量也作为主语出现。在 SPARQL 查询中，变量可以出现在任何位置。其次，查询重用了变量名 ?building。这样，三元组存储库就知道它应该找那些第一条的宾语与第二条的主语相同的三元组。我们请读者来确定这条查询的答案。

我们并不局限于只匹配一个变量。我们可能想要找到三元组存储库中关于 Baron Way Apartment 的所有信息。可以使用这条 SPARQL 查询：

```
PREFIX swp:     <http://www.semanticwebprimer.org/ontology/apartments.ttl#>
PREFIX dbpedia: <http://dbpedia.org/resource/>.
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>.
SELECT ?p ?o
WHERE {
    swp:BaronWayApartment ?p ?o
}
```

它将返回如下结果：

?p	?o
swp:hasNumberOfBedrooms	3
swp:isPartOf	swp:BaronWayBuilding

同样，表格中的每一行是一条单独的能匹配图模式的结果。对于我们这个很小的数据集，
73 所有可能的答案可以很容易地返回。然而，在更大的数据集上，我们可能不知道有多少条结果，或者我们的查询会不会返回整个数据集。事实上，写一些能返回几百万个三元组的查询还是很容易的。因此，一个好的做法是限制一条查询能返回的答案的数量，特别是在使用公共端点时。这很容易通过使用如图 3-1 所示的 LIMIT 关键词来实现。在这个例子中，我们将返回的结果条数限制为 10。

```
PREFIX swp:     <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
PREFIX dbpedia: <http://dbpedia.org/resource/>.
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>.
SELECT ?p ?o
WHERE {
        swp:BaronWayApartment ?p ?o.
}
LIMIT 10
```

图 3-1　一个带有 LIMIT 的 SPARQL 查询

我们已经学会了如何去匹配单个模式或者由多个三元组模式构成的链。SPARQL 提供了一种精确表述属性链的方式。这一机制称为*属性路径*（property path）。例如下面这个例子，找到所有作为一幢位于 Amsterdam 的建筑的一部分的那些公寓。

```
PREFIX swp:     <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
PREFIX dbpedia: <http://dbpedia.org/resource/>.
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>.
SELECT ?apartment
WHERE {
        ?apartment swp:isPartOf ?building.
        ?building dbpedia-owl:location dbpedia:Amsterdam.
    }
```

我们可以这样来表达同样的意思：

74
```
PREFIX ex:      <http://www.example.org/>
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX geo:     <http://www.geonames.org/ontology#>.
SELECT ?tournament
WHERE {
        ?apartment swp:isPartOf/dbpedia-owl:location dbpedia:Amsterdam..
    }
```

有许多其他属性路径可以用来帮助在查询中表述很长的、各式各样的路径。在本章中，强调了更多这样的构造了。事实上，当读者写更复杂的 SPARQL 查询时，这些属性路径的快捷表示可能会变得更有用。

我们仅仅通过匹配图模式就可以完成很多事情。然而，有时候我们想要对查询结果施加更复杂的约束。在下一节中，我们讨论如何使用过滤器来表述这些约束。

3.3 过滤器

继续看公寓这个例子，让我们来找到所有拥有 3 间卧室的公寓。目前我们已经看到的例子中，在查询时只在图模式中使用过资源，还没使用过文字。事实上，文字可以直接包含在图模式中。这样的 SPARQL 查询如下：

```
PREFIX swp:     <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
PREFIX dbpedia: <http://dbpedia.org/resource/>.
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>.
SELECT ?apartment
WHERE {
        ?apartment  swp:hasNumberOfBedrooms 3.
}
```

75

注意，与 Turtle 类似，SPARQL 也允许常见文字的缩写形式。在这个例子中，3 是 `"3"^^xsd:integer` 的缩写表示。SPARQL 和 Turtle 的各种语法缩写表示是一样的。

然而，这个查询有一点不自然。绝大多数情况下，我们想要找的是拥有大于或小于一个特定数量的卧室的那些公寓。我们可以使用 SPARQL 来问这个问题，并使用 FILTER 关键词：

```
PREFIX swp:     <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
PREFIX dbpedia: <http://dbpedia.org/resource/>.
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>.
SELECT ?apartment
WHERE {
        ?apartment  swp:hasNumberOfBedrooms ?bedrooms.
         FILTER (?bedrooms > 2).
}
```

结果是：

?apartment
swp:BaronWayApartment

数值型数据类型（即整型数、小数）和日期/时间都支持小于、大于和等于运算。SPARQL 也支持字符串的过滤。例如，假设我们的数据集包含如下三元组：

```
swp:BaronWayApartment swp:address "4 Baron Way Circle".
```

我们可能想要找所有在地址中包含 "4 Baron Way" 的资源。这可以使用 SPARQL 内置支

持的正则表达式来实现。正则表达式是一种表达字符串搜索的有力方式。详细介绍正则表达
76 式超出了本书的范畴，但作者鼓励读者去探索。用于找到字符串 "4 Baron Way" 出现在另一个字符串开头的正则表达式是 "^4 Baron Way"。这可以如下表述：

```
PREFIX swp:    <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
PREFIX dbpedia: <http://dbpedia.org/resource/>.
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>.
SELECT ?apartment
WHERE {
        ?apartment  swp:address ?address.
        FILTER regex(?address, "^4 Baron Way").
}
```

这里，在 FILTER 关键词之后，引入了一个特殊的过滤函数名：regex。这个函数的参数在随后的括号中给出。SPARQL 还包含一些其他类型的过滤器，在一些特定的场合中可能有用。然而，数值和字符串过滤器是最常使用的。最后一个常用的函数是 str，它将资源和文字转换为可以在正则表达式中使用的字符串表示。例如，我们可以在资源的 URL 而不是标签中搜索 Baron，如下所示：

```
PREFIX swp:    <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>.
SELECT ?apartment ?address
WHERE {
        ?apartment  swp:address ?address.
        FILTER regex(str(?apartment), "Baron").
```
77
```
    }
```

过滤器为我们提供了一种获得灵活性的机制。SPARQL 提供了更多的构造子来处理语义网上发现的往往是不一致的和不断变化的信息。

3.4 处理一个开放世界的构造子

与传统数据库不同，不是每个语义网上的资源都会以同样的模式（schema）描述，或者都拥有同样的属性。这叫做开放世界假设。例如，一些公寓可能比其他的拥有更多的描述。进一步地，它们可能以一种不同的词汇表来描述。例如以下这个 RDF 例子：

```
@prefix swp:    <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.
@prefix xsd:    <http://www.w3.org/2001/XMLSchema#>.
```

```
swp:BaronWayApartment swp:hasNumberOfBedrooms 3.
swp:BaronWayApartment dbpedia-owl:location dbpedia:Amsterdam.
swp:BaronWayApartment rdfs:label "Baron Way Apartment for Rent".

swp:FloridaAveStudio swp:hasNumberOfBedrooms 1.
swp:FloridaAveStudio dbpedia-owl:locationCity dbpedia:Amsterdam.
```

在这个例子中，Florida Ave 单间公寓并没有一个对人类友好的标签，并且它的位置以
`dbpedia-owl:locationCity` 而非 `dbpedia-owl:location` 为谓语来描述。即使有这
种不一致，我们仍然想要在数据上查询并找到位于 Amsterdam 的公寓并返回它们对人类友好 78
的标签——如果有。SPARQL 为表述这个查询提供了两种构造子。让我们来看一个样例查询：

```
PREFIX swp:    <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
PREFIX geo:    <http://www.geonames.org/ontology#>.
PREFIX dbpedia: <http://dbpedia.org/resource/>.
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>.

SELECT ?apartment ?label
WHERE {
        {?apartment  dbpedia-owl:location dbpedia:Amsterdam.}
        UNION
        {?apartment  dbpedia-owl:locationCity dbpedia:Amsterdam.}
        OPTIONAL
        {?apartment rdfs:label ?label.}
}
```

这个查询的结果是：

?apartment	?label
swp:BaronWayApartment	Baron Way Apartment for Rent
swp:FloridaAveStudio	

UNION 关键词告诉三元组存储库返回那些仅匹配一个图模式或两个都匹配的结果。OPTIONAL 关键词告诉三元组存储库为特定的图模式返回结果——如果能找到。即对于待返回的查询而言，这个图模式未必要被满足。因此，在这个例子中，如果没有这个可选项，这间单间公寓就不会在查询结果中返回。

类似地，属性路径也可被用来创建一个更简洁的 SPARQL 查询。使用 | 运算符，我们可 79
以表述一个或更多的可能性。因此，上述 SPARQL 可以被重写成如下形式：

```
PREFIX swp:    <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
PREFIX dbpedia: <http://dbpedia.org/resource/>.
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>.

SELECT ?apartment ?label
WHERE {
    {?apartment dbpedia-owl:location|dbpedia-owl:locationCity dbpedia:Amsterdam.}
    OPTIONAL
    {?apartment rdfs:label ?label.}
}
```

这些仅仅是一些例子，来说明 SPARQL 是如何被设计为可以容易地查询来自不同来源的知识。

3.5 组织结果集

一种常见的情况是，我们想要查询结果以一种特定的方式返回：分组的、计数的或排序的。SPARQL 支持一些函数来帮助我们组织结果集。我们已经知道了如何使用 LIMIT 关键词来限制结果的数量。我们也可以使用 DISTINCT 关键词，把它放在选择关键词之后（例如 SELECT DISTINCT ?name WHERE），来消除结果集中的重复结果。这将确保返回互不相同的变量绑定。

SPARQL 也允许使用 ORDER BY 关键词来对返回的结果集排序。例如，我们可以要求公寓按卧室数量排序。

80

```
PREFIX swp:    <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
PREFIX dbpedia: <http://dbpedia.org/resource/>.
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>.

SELECT ?apartment ?bedrooms
WHERE {
        ?apartment  swp:hasNumberOfBedrooms ?bedrooms.
}
ORDER BY DESC(?bedrooms)
```

其返回的是：

?apartment	?bedrooms
swp:BaronWayApartment	3
swp:FloridaAveStudio	1

DESC 关键词指明了按降序排列。类似地，ASC 指的是升序。此外，注意字符串或 URL 的排序是根据字典序。

我们也可以使用聚集（aggregate）函数来汇总结果集。特别地，我们可以计数结果的数量（COUNT）、求和（SUM）以及计算最小值、最大值和平均值（MIN、MAX、AVG）。这是一个计算我们的数据集中平均卧室数量的例子。

```
PREFIX swp:     <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
PREFIX dbpedia: <http://dbpedia.org/resource/>.
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>.


SELECT  (AVG(?bedrooms) AS ?avgNumRooms)
WHERE {
        ?apartment  swp:hasNumberOfBedrooms ?bedrooms.
}
```

它将返回： 81

?avgNumRooms
2

这个聚集函数与 AS 关键词组合使用，来指明结果集中的变量。并没有限制我们在整个结果集上运用这些聚集。我们也可以使用 GROUP BY 关键词来聚集特定的分组。

因此，SPARQL 为以最适合具体应用的方式组织结果提供了有力机制。

3.6 其他形式的 SPARQL 查询

到目前为止，我们已经关注了从一个 RDF 集合中选择特定的值。SPARQL 也支持一些其他形式的查询。除了 SELECT 以外，两种常用的查询是 ASK 和 CONSTRUCT。

ASK 形式的查询简单地检查一个数据集中是否存在一个图模式，而不是去返回结果。例如，下面这个查询将返回真。

```
PREFIX swp:     <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
PREFIX dbpedia: <http://dbpedia.org/resource/>.
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>.
ASK ?apartment
WHERE {
        ?apartment  swp:hasNumberOfBedrooms 3.
}
```

使用 ASK 查询的原因是，比起检索一个完整的结果集，ASK 查询计算得更快。

CONSTRUCT 形式的查询用来从一个更大的 RDF 集中检索出一个 RDF 图。因此，可以

82 查询一个三元组存储库并检索一个 RDF 图而非一组变量绑定。例如，我们可以创建一个新图，将那些拥有超过 2 间卧室的公寓标记为大公寓。

```
PREFIX ex:      <http://www.example.org/>
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX geo:     <http://www.geonames.org/ontology#>
CONSTRUCT {?apartment swp:hasNumberOfBedrooms ?bedrooms. ?apartment swp:isBigApartment true.}
WHERE{
        ?apartment  swp:hasNumberOfBedrooms ?bedrooms.
}
FILTER (?bedrooms > 2)
```

这将返回如下的图。

```
@prefix swp:     <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#>.

swp:BaronWayApartment swp:hasNumberOfBedrooms 3.
swp:BaronWayApartment swp:isBigApartment true.
```

CONSTRUCT 查询经常用来在模式（schema）之间转换——通过查询特定的模式，并用目标模式中的属性替换。

3.7 查询模式

重要的是，因为模式（schema）信息是用 RDF 表示的，SPARQL 可以用来查询关于模式
83 本身的信息。例如，以下是上一章中住房本体的一部分。

```
@prefix swp: <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.

swp:Unit rdf:type rdfs:Class.

swp:ResidentialUnit rdf:type rdfs:Class.
swp:ResidentialUnit rdfs:subClassOf swp:Unit.

swp:Apartment rdf:type rdfs:Class.
swp:Apartment rdfs:subClassOf swp:ResidentialUnit.
```

使用SPARQL，通过同时查询实例数据和模式（schema），我们可以确定数据集中的Residential Units：

```
PREFIX swp: <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .


SELECT ?athlete
WHERE{
  ?apartment a ?unitType.
  ?unitType rdfs:subClassOf swp:ResidentialUnit.
}
```

注意，我们使用了与Turtle相同的缩写法：用a来表示rdf:type。对模式的查询能力是SPARQL和RDF的一项重要能力，因为它不仅允许检索信息，还可以查询信息的语义。 84

3.8 通过 SPARQL 更新来增加信息

正如3.1节所述，SPARQL也定义了一个用来更新三元组存储库内容的协议。这就是SPARQL更新协议。本质上，它在SPARQL中引入了一系列新的关键词来支持三元组的插入、加载和删除。以下我们展现每种请求的例子。

插入和加载三元组 以下插入一个三元组，阐述Luxury Apartment是Apartment的一个子类。它将这个三元组加入三元组存储库的已有内容之中。

```
PREFIX swp: <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#>


INSERT DATA
{
    swp:LuxuryApartment rdfs:subClassOf swp:Apartment.
}
```

注意，数据本身仍是我们在第2章中就熟悉的Turtle语法。

如果你在万维网上有一个大的包含RDF的文件，你可以使用以下命令将它加载进一个三元组存储库：

```
LOAD <http://example.com/apartment.rdf>
```

删除三元组 从一个三元组存储库中删除三元组有几种方式。一种是使用DELETE DATA关键词准确指定哪些三元组是你想要删除的。将之前插入的三元组删除可以这样： 85

```
PREFIX swp: <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
```

```
DELETE DATA
{
    swp:LuxuryApartment rdfs:subClassOf swp:Apartment.
}
```

注意，在这种形式下是不允许变量的，所有三元组都必须被完整指定。

一种更加灵活的方式是使用 DELETE WHERE 构造子。它可以删除匹配指定图模式的那些三元组。以下将要删除包含关于拥有超过 2 间卧室的公寓的信息的所有三元组。

```
PREFIX swp: <http://www.semanticwebprimer.org/ontology/apartments.ttl#>.
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .

WHERE{
        ?apartment  swp:hasNumberOfBedrooms ?bedrooms.
        FILTER (?bedrooms > 2)
}
```

在这两种形式中，如果模式未能匹配或者三元组不在三元组存储库中，那么什么都不会发生。

最后，要删除一个三元组存储库中的所有内容，可以如下使用 CLEAR 构造子：

86
```
CLEAR ALL
```

SPARQL 更新提供了更多的构造子来管理部分三元组存储库。如果向一个三元组存储库中逐步添加数据，那么更新操作就特别有用了。在下一节中，我们讨论一种特殊情况，其中这些更新操作很有用。

3.9 “跟着感觉走”原则

SPARQL 提供了机制来查询和更新三元组存储库。但是，如何填充这些三元组存储库呢？如前所述，许多数据提供者通过 SPARQL 端点发布他们的数据。然而，其他数据提供者仅仅通过链接数据的形式，即 RDF 数据或者以万维网文件的形式发布，或者内嵌在网页中。我们可以将这些三元组插入本地的三元组存储库中。然而，语义网允许任何提供者使用万维网上的其他资源和信息来描述他们的信息。在这些情况下，可以运用*跟着感觉走原则*（follow your nose principle）：给定一个指向某些 RDF 的 URL，通过解引用这个 URL 来加载相应的数据。可以持续这样做，直到有足够的三元组来回答给定的查询。

一些查询引擎，如 SQUIN，实现了这一功能（参见 http://squin.sourceforge.net/）。此外，新版本的 SPARQL 也包含了这些联邦查询命令。然而，这种联邦查询往往很耗时，因为数据必须在查询时才能收集。

跟着感觉走是一种在使用语义网来处理查询时收集所需信息的方式。

3.10 小结

在本章中，我们介绍了 SPARQL 查询和更新，以及支撑 SPARQL 的基础设施。 87

- SPARQL 通过匹配图模式来选择信息，并提供基于数值和字符串比较的过滤机制。
- SPARQL 查询采用类似 Turtle 的语法。
- 数据和模式（schema）都可以使用 SPARQL 来查询。
- UNION 和 OPTIONAL 构造了允许 SPARQL 更容易地处理开放世界数据。
- SPARQL 更新提供了从三元组存储库中更新和删除信息的机制。

建议阅读

- ARQ SPARQL Tutorial. http://jena.sourceforge.net/ARQ/Tutorial/.
- S. Harris and A. Seaborne, eds. SPARQL 1.1 Query Language. W3C Working Draft (work in progress), 12 May 2011.
  http://www.w3.org/TR/sparql11-query/.
- A. Polleres, P. Gearon, and A. Passant, eds. SPARQL 1.1 Update. W3C Working Draft (work in progress), 12 May 2011. www.w3.org/TR/sparql11-update/.
- E. Torres, L. Feigenbaum, and K. Clark, eds. SPARQL Protocol for RDF. W3C Recommendation, 15 January 2008. www.w3.org/TR/rdf-sparql-protocol/.

练习和项目

1. 从本章给出的查询中选择一个。将其背后的数据画成一张图。再画一个图，标记选择的查询中的那些变量。你能将查询也表达为一个图吗？ 88

2. 思考什么会让一条 SPARQL 查询难以被一个三元组存储库回答。讨论哪些因素对回答查询造成了困难。

3. 比较 SPARQL 和 SQL。语言上的不同和相似之处分别是什么？

4. 在 http://www.dbpedia.org 上执行几条查询。使用其提供的执行查询的几个万维网界面中的一个：http://dbpedia.org/snorql 或者 http://dbpedia.org/sparql。讨论构建查询的困难是什么。

5. 下载并安装一个三元组存储库，例如 4store、Virtuoso、Sesame 和 OWLIM。加载本书网站（http://www.semanticwebprimer.org）上的 RDF 数据，看看你能否回答本章中的所有查询。如果三元组存储库支持推理，查询答案会随着结果变化吗？

6. 讨论 SPARQL 如何使用其他万维网标准（例如 HTTP）。

7. 在构建 SPARQL 查询时，本体的好处是什么？

8. 作为一项更大的任务，每 2 ~ 4 人为一组，开发一个万维网应用。使用上一章中的一个本体作为基础。例如，可以开发一个公寓或者图书的查找应用。不采用标准的数据库作为数据存储方式，而是采用三元组存储库。我们建议使用 rdfquery javascript 库（http://code.google.com/p/rdfquery/）在网页中与三元组存储库交互。此外，一个 PHP（超文本预处理器）引擎，诸如 RAP（http://www4.wiwiss.fu-berlin.de/bizer/rdfapi/），是另一个选择。应用构建完成以后，写一个报告阐述使用本体和三元组存储库来开发万维网应用的优点和缺点。 89/90

万维网本体语言：OWL2

4.1 引言

使用前几章中介绍的 RDF 和 RDF 模式来描述各种事物是非常受限的。RDF（笼统地说）被限定为二元闭谓词，而 RDF 模式（笼统地说）被限定为子类层次和属性层次，以及属性的定义域和值域定义。这两种语言的设计都考虑灵活性。

但是，在许多情况下，我们需要表达更加先进的、更具“表达能力的”的知识——例如，每个人只有一个精确的出生日期，或者没有人可以同时是男人和女人。

后续的 W3C 工作组[⊖]、万维网本体工作组和 OWL 工作组，标识了语义网中的一些典型
91 用例。它们需要 RDF 和 RDFS 已经提供的语言特征以外的更多特征。结果产生了 OWL2 语言，代表万维网本体语言，它和逻辑家族中的一类关联紧密，该类特别适合表达术语知识。描述逻辑（Description Logic，DL）有很长的历史，并且它的特性被大家广泛理解。OWL2 是 OWL 语言的第 2 版。

本章概述

在本章中，首先根据需求描述 OWL2 的动机（4.2 节）以及它与 RDF 和 RDFS 的关系（4.3 节）。接下来，我们在 4.4 节中详细介绍 OWL2 的各种语言元素，之后在 4.5 节讨论 3 个 OWL2 概要（profile）。

4.2 本体语言的需求

我们已经在前面的章节中看到了 RDF 和 RDFS 允许我们描述存在于某个领域中的类，或者“概念”，并允许我们在万维网上共享这些描述。一个领域中概念的显式的形式化规约称为本体（ontology）。因此允许我们表达本体的语言称为本体语言（ontology language）。对这些语言的主要需求包括：一个良定的语法，一个形式语义，足够的表达能力，方便的表达方式和高效的推理支持。

4.2.1 语法

一个良定语法（well-defined syntax）的重要性在于清晰并且被编程语言领域熟知，这是机器处理信息的必要条件。一个语法是良定的，仅当你能够使用它以一种无二义的方式写出某

⊖ 参见 http://www.w3.org/2001/sw/WebOnt/ 和 http://www.w3.org/2007/OWL/wiki/OWL_Working_Group。

种语言允许你表达的所有事物。我们已经介绍过的所有语言都有一个良定的语法。正如我们将看到的，OWL2 建立在 RDF 和 RDFS 之上，并且使用了它们语法的一种扩展。 92

一个良定的语法不必是非常友好的。例如，众所周知 RDF/XML 语法对人而言难以阅读。但是这个缺点并不是非常关键，因为绝大多数本体工程师会使用专门的本体开发工具，而不是文本编辑器，来构建本体。

4.2.2 形式语义

形式语义（formal semantic）精确地描述了一种语言的含义。精确（precisely）意味着语义不涉及主观想象，不同人（或机器）也不会存在不同的解释。例如，形式语义的重要性在数理逻辑领域中被很好地建立起来。

将形式语义和一个良定的语法组合起来允许我们解释（interpret）用某种语法表达的句子：我们现在理解（know）了句子的含义。形式语义也使得我们对表达在句子中的知识进行推理。例如，RDFS 的形式语义允许我们推理类成员关系（class membership）。给定：

```
:x rdf:type        :C .
:C rdfs:subClassOf :D .
```

我们可以推导出 `:x` 是 `:D` 的一个实例。`rdfs:domain` 和 `rdfs:range` 属性允许类似的推理：

```
:p rdfs:range     :D .
:x :p             :y .
```

允许我们推导出 `:y rdf:type :D`。 93

4.2.3 表达能力

不幸的是，RDF 和 RDFS 的表达能力（expressive power）在某些地方非常有限。我们有时需要提供超出 RDF 和 RDFS 允许描述的更精确的定义。如果构建本体，可能希望能够进行如下推理：

类成员关系 我们已经看到 RDFS 拥有一些简单的机制，使用子类和定义域/值域来检测个体实例的类成员关系。但是，对于一个实例在什么情况下可以被认为属于一个类的更精确的描述将允许更加精细的推理。例如，如果我们已经声明某些属性－值对是一个类 `:A` 的成员关系的充分条件，则如果某个实例 `:x` 满足这些条件，我们可以得出 `:x` 必然是 `:A` 的实例的结论：某个事物是网球比赛，那么它至少包含球员、球拍等。

分类 类似地，我们希望使用类成员关系的条件来推导类自身之间的关系。例如，一个网球比赛的简单定义可以重用来定义羽毛球比赛。

等价关系和相同性 表达类之间的等价关系（equivalence）很有用。例如，`:Tortoise` 类和 `:Land_Turtle` 类共享所有的成员；因此它们是等价的。类似地，我们希望能够表达两个实例何时相同：`:morning_star` 和 `:evening_star` 是同个星球 `:venus` 的不同名字，因此这些实例是相同的。能够直接表达这些很好，但是也应该可以通过在类的描述上应用形

式语义来确定等价关系和相同性。

94 **不相交关系和不同性** 同样，有时我们知道两个类不共享任何实例（它们是不相交的（disjoint））或者两个实例是明确不同的事物。例如，:Winner 和 :Loser 是不相交的，而 :roger_federer 和 :rafael_nadal 是不同的个体。

类的二元组合 有时类需要以超出子类关系的方式组合。例如，我们可能希望定义 :Person 类是两个不相交类 :Female 和 :Male 的并集。

属性的局部作用域 rdfs:range 声明一个属性的值域中的实例，例如 :plays 的值域都属于某个类。因此，在 RDFS 中我们不能区别不同情境中的值域限制。例如，我们不能说网球运动员只打网球，而其他人可能打羽毛球。

属性的特性 有时声明一个属性是传递的（transitive），例如 :greater_than；唯一的（unique），例如 :is_mother_of；或者互逆的（inverse），例如 :eats 和 :is_eaten_by，都很有用。

基数限制 有时，我们需要对一个属性可能或者必须拥有的不同取值的数目施加限制。例如，每个人有且只有双亲，一门课至少需要一个讲师授课。

一致性 一旦我们能够确定类之间的关系，我们可能也希望确定它们定义之间的冲突。假设我们声明 :Fish 和 :Mammal 是不相交的类，则断言 :dolphin 是两者共同的实例就是个错误。一个具有足够表达能力的本体语言应该允许我们检测这些不一致性的情况。

最后，一个本体语言必须使得利用它的表达能力来构建句子越方便（convenient）越好。
95 例如，一个语言如果在我们希望声明两个类等价时需要不断重复完整的定义，那么它可能就不是很方便。

4.2.4 推理支持

形式语义是推理支持（reasoning support）的前提。推导之前的例子可以通过机制而非手工完成。自动推理很重要，因为它允许我们检查本体的正确性。例如：

- 检查本体的一致性。
- 检查类之间的意外关系。
- 检查实例的意外分类。

自动的推理支持允许检查比手工检查更多的情况。诸如在前述例子的检查对于设计大型本体非常有价值，例如当涉及多个作者时，又如当从不同数据源集成和共享本体时。

我们能够通过将一个本体语言映射到一个已知的逻辑系统，并且通过使用针对该系统已有的自动化推理机，为该本体语言提供形式语义和推理支持。

很明显，我们需要比 RDF 模式更加丰富的本体语言，一种能够提供上述特征甚至更多特征的语言。在设计这种语言的过程中，应该意识到表达能力和高效推理支持之间的权衡。通俗地讲，一个逻辑系统越丰富，推理支持越低效，时常越过了可判定性的界限；也就是说，在这种逻辑上的推理不能保证终止。因此我们需要一个妥协，一种可以被相对高效的推理机
96 支持的语言，同时能够具有足够的表达能力来表达一大部分知识。

4.3 OWL2 和 RDF/RDFS 的兼容性

理想情况是 OWL2 作为 RDF 模式的扩展，也就是说，OWL2 采用 RDFS 含义的类和属性（`rdfs:Class`、`rdfs:subClassOf` 等），并添加语言原语来提供所需的更丰富的表达能力。这种方法将和语义网的分层体系结构一致（参见图 1-4）。

不幸的是，简单地扩展 RDF 模式将和获得更强、更高效的推理相背离。RDF 模式拥有一些非常强大的建模原语。例如 `rdfs:Class`（所有类的类）和 `rdf:Property`（所有属性的类）的结构具有很强的表达能力，如果 OWL2 的底层逻辑以一般形式包含这些原语，将会导致不可控的计算性质。

两种语义

本体语言的完整需求集合看起来无法获得：高效的推理支持能够支持的语言的表达能力无法达到将 RDF 模式和一种完整逻辑组合的程度。事实上，这些需求导致后继 W3C 工作组将 OWL2 分为两个不同的子语言，各自拥有不同的底层语义，面向满足需求全集的不同方面。⊖

1. OWL2 Full：基于 RDF 的语义

完整的语言称为 OWL2 Full，并且使用所有的 OWL2 语言原语。它也允许以任意方式将这些原语与 RDF 和 RDF 模式组合。这包括了通过将语言原语用于其他原语来改变预定义的（RDF 或 OWL2）原语的含义的能力。例如，在 OWL2 Full 中，我们可以引入一个基数约束到所有类的类，这本质上限制了任意一个本体中可以描述的类的数目。 97

OWL2 Full 的优点在于它被映射到一个基于 RDF 的语义（RDF-based semantic）。因此它在结构上和语义上完全向上兼容 RDF：任何合法的 RDF 文档也是一个合法的 OWL2 Full 文档，并且任何有效的 RDF 模式推理也是一个有效的 OWL2 Full 结论。OWL2 Full 的缺点是这个语言已经变得太强以至于是不可判定的，使得任何完备（或高效）推理支持的希望都破灭了。

2. OWL2 DL：直接语义

为了重新获得计算效率，第二个子语言 OWL2 DL 被映射到描述逻辑（DL）上。描述逻辑是谓词逻辑的一个子集，它使得高效的推理支持成为可能。OWL2 DL 限制了 OWL2、RDF 和 RDFS 的原语的使用方式。一些限制包括：

- OWL2 DL 不允许 OWL2 的原语应用于其他原语。
- 其次，OWL2 DL 只能定义非文字资源的类。所有 OWL2 DL 类是 `owl:Class` 的实例，而不是 `rdfs:Class` 的。
- 再次，OWL2 DL 严格区分了值域包含非文字资源的属性和关联文字值的属性。所有 OWL2 DL 属性或是 `owl:ObjectProperty` 或是 `owl:DatatypeProperty` 的实例，而不能同时是两者的实例。

⊖ OWL 的第一个版本包括第三个子语言，称为 "OWL Lite"。但是，这个语言被 4.5 节中介绍的 "概要" 所取代。

- 最后，在 OWL2 DL 中，一个资源不能同时是类、属性和实例。它们可能拥有相同的名字（被称为“双关语”(punning)），但是总能根据底层逻辑区分为不同的事物。

98 上述这些限制保证该语言维护了与一个广泛理解的描述逻辑之间的直接对应。图 4-1 展示了 OWL2 和 RDF/RDFS 的一些建模原语之间的子类联系。

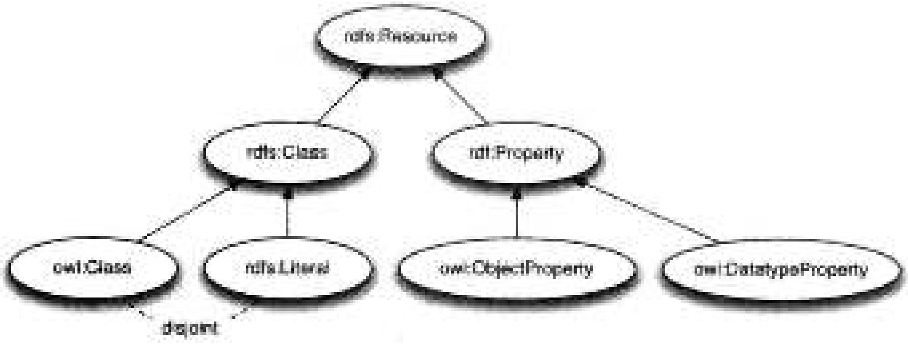


图 4-1 OWL2 和 RDF/RDFS 之间的子类联系

这种受限的表达能力的优势在于它允许高效的推理支持。OWL2 DL 可以利用大量现有的推理机，例如 Pellet、FaCT、RACER 和 HermiT。缺点在于我们失去了和 RDF 间完整的兼容性。一个 RDF 文档在变成一个合法的 OWL2 DL 文档之前，通常将不得不以某种方式扩展或以另一些方式加以限制。但是每个合法的 OWL2 DL 文档是一个合法的 RDF 文档。

语义网的分层体系结构（参见图 1-4）背后一个主要的动机是希望在各种层次之间向下兼容对应软件的复用。但是，对于 OWL2 的完整向下兼容性（任何感知 OWL2 的处理机也将提供任意 RDF 模式文档的正确解释）的优势只能通过 OWL2 Full 实现，代价是不易计算性。

在 4.5 节中，我们将继续讨论关于 OWL2 的 3 个额外的概要，每个概要将寻求表达能力
99 和高效推理之间的不同权衡。

4.4 OWL 语言

这节介绍 OWL2 的语言原语。由于它与形式逻辑的紧密联系，采用一些相关的词汇表很方便：

- 在 OWL2 中，类的成员通常被称为个体（individual）而非实例（instance），但是我们将互换地使用这两个术语。
- 当我们声明某个资源是一个特定类型时，称这是一个断言（assertion）。例如：

```
:roger_federer rdf:type :Person .
```

是一个类断言（class assertion），将一个个体 :roger_federer 关联到它的类。

- 当我们组合类、属性和实例时，它们形成了表达式（expression）。例如：

```
_:x rdf:type owl:Class ;
  owl:unionOf ( :Man :Woman ) .
```

是一个类表达式（class expression），它指出了一个（匿名的）类是 `:Man` 和 `:Woman` 类的并集。

- 如果我们随后将这个定义关联到我们的某个类，我们就创建了公理（axiom）。例如：

```
:Person owl:equivalentClass _:x .
_:x    rdf:type         owl:Class ;
       owl:unionOf      ( :Man :Woman ) .
```

是一个等价类公理，它声明了类 `:Person` 等价于我们直接引入的并集。类公理有时也称为限制（restriction），因为它们对可以作为类的成员的一组个体施加了约束。 100

记住 OWL2 本质上是一个描述事物集合的语言这点很有用。这些集合被称为"类"。我们为一个 OWL2 类产生的任何声明意味着将这个类和所有事物的集合区别开。

4.4.1 语法

OWL2 建立在 RDF 和 RDF 模式之上，并且可以使用所有合法的 RDF 语法来表达。但是，存在许多 OWL2 语法，每个都有优缺点。

函数式语法 这种语法和本体的形式化结构关系紧密。它用于 OWL2 的语言规范文档，OWL2 本体的语义定义，与 RDF 语法的相互映射，以及 OWL2 的不同概要中。它比其他语法更加紧凑和可读。例如，上面的类约束可以写成下面的语法：

```
EquivalentClasses( :Person ObjectUnionOf( :Man :Woman ) )
```

OWL/XML 这是一种 OWL2 的 XML 语法，但是它不遵从 RDF 习惯，而是紧密映射到函数式语法。[⊖] 这种语法的主要优点在于它允许我们使用标准的现成的 XML 编辑工具来与本体交互。例如，上面类公理的等价 OWL/XML 语法如下：

```
<EquivalentClasses>
   <Class abbreviatedIRI=":Person"/>
   <ObjectUnionOf>
      <Class IRI="#Man"/>
      <Class IRI="#Woman"/>
   </ObjectUnionOf>
</EquivalentClasses>
```

101

曼彻斯特语法 这种语法起初由曼彻斯特大学（University of Manchester）开发，设计得越适合人类阅读越好。它是诸如 Protégé 在内的当前绝大多数本体编辑器在用户界面中使用的语法。

```
Class: Person
  EquivalentTo: Man or Woman
```

⊖ OWL/XML 序列化被定义在 http://www.w3.org/TR/owl-xml-serialization/。不要和早期的 OWL/XML 表示语法混淆（定义在 http://www.w3.org/TR/owl-xmlsyntax），那是基于 OWL 第 1 版的抽象语法。

除了这些语法之外，所有RDF语法都可以用于OWL。因此，在本节中，我们依旧使用在先前章节中介绍过的Turtle语法。

4.4.2 本体文档

当使用Turtle语法时，OWL2本体文档，或简单地称为本体，仅仅是另一个RDF文档。OWL2本体最少包含下面几个命名空间：

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
```

一个针对家政目的的OWL2本体起始于一组断言。这些断言介绍了一个基命名空间，本体本身、它的名字、可能的注释、版本控制以及包含的其他本体。例如：

102

```
@prefix : <http://www.semanticwebprimer.org/ontologies/apartments.ttl#> .
@prefix dbpedia-owl: <http://dbpedia.org/ontology/> .
@prefix dbpedia: <http://dbpedia.org/resource/> .
@base <http://www.semanticwebprimer.org/ontologies/apartments.ttl> .

<http://www.semanticwebprimer.org/ontologies/apartments.ttl>
   rdf:type owl:Ontology ;
   rdfs:label "Apartments Ontology"^^xsd:string ;
   rdfs:comment "An example OWL2 ontology"^^xsd:string ;
   owl:versionIRI <http://www.semanticwebprimer.org/ontologies/apartments.ttl#1.0> ;
   owl:imports <http://dbpedia.org/ontology/> ;
   owl:imports <http://dbpedia.org/resource/> .
```

Imports 只有一个断言会影响本体的逻辑含义：`owl:imports`，它指向其他本体，这些本体的公理也是当前本体的一部分。我们的公寓本体导入了被定义在DBpedia本体中的所有公理，以及DBpedia本身的所有事物。这立刻就引发了`owl:imports`的一个问题：为了能够使用DBpedia中的部分信息，我们不得不导入它其中描述的所有6.72亿个三元组。[⊖]

命名空间仅被用于消除歧义，而被导入的本体提供了可以被使用的定义。通常一个本体对每个它所使用的命名空间包含一个导入声明，但是也可以导入额外的本体——例如，提供定义而不引入新名称的本体。`owl:imports`属性是传递的，也就是说，如果本体O_i引入本体O_j，而本体O_j又引入本体O_k，那么本体O_i也引入本体O_k。

4.4.3 属性类型

在4.3.1节中，我们讨论了OWL2区别两种类型的属性：对象（object）属性和数据类型

⊖ 现状参见 http://dbpedia.org/About。

（datatype）属性。事实上，针对一些属性的特征，OWL2 还提供了额外的类型。在本节中，我们将概述这些类型。 103

对象属性 这些属性将个体与个体关联。例如，:rents 和 :livesIn：

```
:rents rdf:type           owl:ObjectProperty ;
       rdfs:domain        :Person ;
       rdfs:range         :Apartment ;
       rdfs:subPropertyOf :livesIn .
```

数据类型属性 这些属性将个体与一种数据类型的文字值关联。例如，:name 和 :age：

```
:age rdf:type   owl:DatatypeProperty ;
     rdfs:range xsd:nonNegativeInteger .
```

正如像 RDF 中的那样，OWL2 允许使用 XML 模式中的数据类型来声明一个文字的类型，或者指定一个数据类型属性的值域。[⊖] 用户定义的数据类型可以在 XML 模式中指定，并用于 OWL2 本体中（参见 4.4.6 节）。

由于直接语义的约束，在 OWL2 DL 中，下面的属性类型中只有函数型属性（functional property）类型可以被赋予数据类型属性。

标注属性 标注属性是一些不携带任何 OWL2 DL 直接语义含义的属性。也就是说，它们被一个描述逻辑推理机忽略。但是，会在 RDF 形式和 OWL2 Full 推理机中考虑它。

标注属性特别用于为 OWL2 本体、类、属性和个体添加可读的标签、注释或解释。 104

```
:label  rdf:type           owl:AnnotationProperty ;
        rdfs:range         rdf:PlainLiteral ;
        rdfs:subPropertyOf rdfs:label .

:Apartment  :label  "Apartment"@en ,
                    "Appartement"@nl ,
                    "Διαμέρισμα"@el .
```

上面的例子说明了几件事情。我们首先定义 :label 属性的类型是 owl:Annotation-Property，值域是 rdf:PlainLiteral。这是一个针对自然语言文本的特殊 RDF 数据类型，即无类型文字可以拥有一个语言标签。我们接下来定义 :label 属性是 rdfs:label 的一个子属性，并且给 :Apartment 类英语、荷兰语和希腊语三种标签。

通常情况下，标注属性将拥有文字值，但是它们也可能被用于关联非文字资源。

顶层和底层属性 OWL2 中的所有对象属性是 owl:topObjectProperty 的子属性。这个属性被定义为关联本体中所有个体的属性。相反地，owl:bottomObjectProperty 不关联任何个体。类似地，owl:topDataProperty 关联所有个体到任何可能的文字值，而 owl:bottomDataProperty 不关联任何个体到任何值。

⊖ OWL2 引入两个额外的数据类型，owl:real 和 owl:rational，作为 xsd:decimal 的超类。

传递属性　从`rdfs:subClassOf`的讨论可知，这个关系是传递的：每个类是它直接超类的所有超类的子类。显然，存在其他关系也是传递的，例如`:isPartOf`或`:isCheaperThan`。我们可以用下面的方式定义一个属性为传递的：

```
:isPartOf rdf:type owl:ObjectProperty ;
          rdf:type owl:TransitiveProperty .
```

105

传递属性是所谓的复合属性（composite property）：它们可以通过多步被声明。例如，给定：

```
:BaronWayApartment :isPartOf :BaronWayBuilding .
:BaronWayKitchen   :isPartOf :BaronWayApartment .
```

一个推理机将推导：

```
:BaronWayKitchen   :isPartOf :BaronWayBuilding .
```

最后一个`:isPartOf`关系是由前面两个属性断言复合而成。因为这种复合关系，传递属性服从一些约束，在表4-1中列出。

表 4-1　复合属性的约束

何时属性是复合的?
• *top* 和 *bottom* 属性都是复合的
• 任意属性自身是传递的，或者有一个逆属性是传递的 • 任意属性拥有一个传递的子属性，或者子属性的逆属性是传递的 • 任意属性是一个属性链（property chain）的超属性，或者是一个属性链的超属性的逆属性 • 任意属性是以上某种属性的等价属性，或者是以上某种属性的等价属性的超属性 复合属性有时也称为复杂角色（complex roles）或非简单（non-simple）属性
限制
复合属性不能出现在以下公理中： • 类上的限定和非限定基数限制 • 类上的自限制 • 不相交的属性公理 而且，它们不能被赋予以下属性类型： • 函数型或反函数型 • 反自反的 • 非对称的

对称和非对称属性　某些属性，例如`:isAdjacentTo`，是对称的；也就是说，如果

106 *a*`:isAdjacentTo` *b*，则反过来也成立。换句话说，对称属性和它们的逆等价（参见4.4.4节）。对于其他属性，我们知道这可能不成立。例如，`:isCheaperThan`关系是非对称的，因为没有人能够被他们击败的人所击败[⊖]。

```
:isAdjacentTo rdf:type owl:ObjectProperty ;
              rdf:type owl:SymmetricProperty .
```

⊖ 当然，这只在每对人只允许进行一场比赛的情况下成立。

```
:isCheaperThan rdf:type owl:ObjectProperty ;
        rdf:type owl:AsymmetricProperty ;
        rdf:type owl:TransitiveProperty .
```

函数型和反函数型属性 对于某些属性，我们知道每个个体通过该属性总是最多存在另一个相关个体。例如，`:hasNumberOfRooms` 是一个函数型属性，而 `:hasRoom` 属性是反函数型的：

```
:hasNumberOfRooms rdf:type owl:DatatypeProperty ;
        rdf:type owl:FunctionalProperty .

:hasRoom      rdf:type owl:ObjectProperty ;
        rdf:type owl:InverseFunctionalProperty .
```

注意，如果两个公寓 a_1 和 a_2 通过 `:hasRoom` 关联到相同的房间 r，这个不一定是不一致的：这些个体可以被推断为相同的。

自反和反自反属性 一个属性的自反性意味着每个个体通过该属性关联到自身。例如，每个事物 `:isPartOf` 其自身。另一方面，反自反性意味着没有个体通过该属性关联到自身。大多数定义域和值域不相交的属性事实上是反自反的。一个例子是 `:rents` 属性： 107

```
:isPartOf     rdf:type owl:ObjectProperty ;
        rdf:type owl:ReflexiveProperty .

:rents        rdf:type owl:ObjectProperty ;
        rdf:type owl:IrreflexiveProperty .
```

4.4.4 属性公理

除了之前章节中介绍的属性类型外，我们能够根据属性与类及其他属性如何关联来指定属性的额外特性。一些和 RDF 模式中的相类似；而另一些则完全是新的。

定义域和值域 在 4.4.3 节中我们已经看到，OWL2 处理属性的定义域和值域的方式与 RDF 模式一样。如果一个属性断言了超过一个 `rdfs:range` 或 `rdfs:domain`，实际的值域或定义域是在属性公理中指定的类的交集。

一个常见的误解是将定义域和值域用作约束，约束了可能通过一个属性关联的个体类型。事实上，定义域和值域只能用于确定这些个体的类成员关系。考虑上面的 `:rents` 定义，任意两个个体 p 和 a 满足 p`:rents`a，将被各自分类为 `:Person` 和 `:Apartment` 的成员。

逆属性 OWL2 允许我们定义属性的逆。一个常见的例子是 `:rents` 和 `:isRentedBy` 属性对。例如：

```
:isRentedBy rdf:type    owl:ObjectProperty ;
        owl:inverseOf :rents .
```

这意味着一个推理机将确定两个个体 p 和 m，它们除了拥有 p`:rents` m，还拥有 108

m:isRentedBy。定义域和值域从逆属性继承下来：:isRentedBy 有 :Person 作为值域，而 :Apartment 作为定义域。在 OWL2 DL 中，只有对象属性才能拥有逆属性。

等价属性 属性也可以被定义为等价的。也就是说，每两个通过某属性关联的个体总是通过它的等价属性关联，反之亦然。等价性是一种方便地将不同本体中的元素相互映射的机制。例如：

```
:isPartOf rdf:type              owl:ObjectProperty ;
        owl:equivalentProperty dbpedia:partOf .
```

不相交属性 对于一些属性，我们知道两个个体可以通过一个属性相关联，则不可能通过另一个：拥有属性的个体对的集合是不相交的。例如 :rents 和 :owns 属性：

```
:rents rdf:type             owl:ObjectProperty ;
    rdfs:domain             :Person ;
    rdfs:range              :Apartment ;
    owl:disjointProperty :owns .
```

显然，你不能租某些你拥有的东西。注意，在 OWL2 DL 的直接语义下，owl:ObjectProperty 和 owl:DatatypeProperty 也是不相交的。

属性链 OWL2 的一个更加复杂的特性是定义属性的链（chain）的能力。有时指定关联各种个体的属性的图的路径是有用的。例如，如果我们知道 :Paul :rents :BaronWayApartment，同时 :BaronWayApartment :isPartOf :BaronWayBuilding，它的 dbpedia:location 是 dbpedia:Amsterdam，则我们知道 :Paul 必然有一个 :livesIn 关系到 :Amsterdam。在 OWL2 中，我们可以通过属性链公理来指定这种

109 关系：

```
:livesIn rdf:type           owl:ObjectProperty ;
      owl:propertyChainAxiom ( :rents :isPartOf :location ) .
```

图 4-2 展示了 :livesIn 关系如何通过公寓例子推导得到。注意，属性链公理不能使命名属性（:livesIn）等价于属性链；它更像是链的一个子属性。在 OWL2 DL 中，属性链值只允许涉及对象属性，虽然绝大多数推理机能够在最后处理拥有数据类型属性的链。

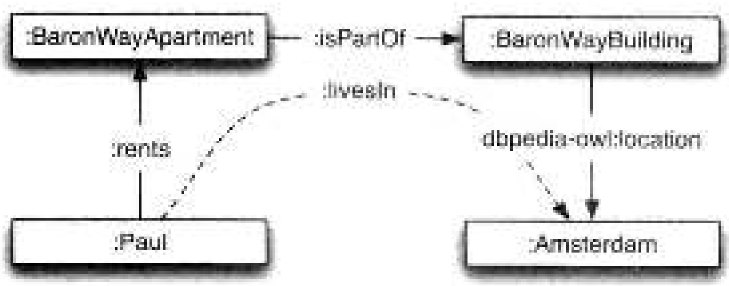


图 4-2 属性链（虚线由推理机推导得到）

出于表达能力，属性链需要服从一些约束。首先，和传递属性类似，属性链的超属性是复合的。这意味着它们不能使用一些公理（参见表 4-1）。其次，属性链不能是递归的：链的超属性、它的逆或它的一个子属性（或子属性的逆）不能出现在属性链公理中。例如，OWL2 DL 不允许我们以下面的方式扩展 :livesIn 属性：

```
:livesIn rdf:type              owl:ObjectProperty ;
        owl:propertyChainAxiom ( :rents :isPartOf dbpedia-owl:location ) ;
        owl:propertyChainAxiom ( :livesIn dbpedia-owl:country ) .
```

110

尽管它允许我们推导出，因为 :Paul 居住在 dbpedia:Amsterdam，他也必须居住在 dbpedia:Netherland。

4.4.5 类公理

类通过断言某个资源的类型是 owl:Class 来定义。存在两个预定义的类在推理中起重要作用：owl:Thing 和 owl:Nothing。前者是最一般的类；每个可能的 OWL2 个体都是这个类的成员，而 owl:Class 的每个实例都是 owl:Thing 的子类。owl:Nothing 类是空类，它没有成员，并且 owl:Class 的每个实例都是该类的超类。不一致的类不能拥有任何成员，因而等价于 owl:Nothing。注意，owl:Thing 的约束拥有非常深远的影响：它们对于本体的每个类和个体都成立。

子类关系 子类关系的定义和RDF模式中类似。例如，我们可以定义一个类 :LuxuryApartment 如下：

```
:LuxuryApartment rdf:type        owl:Class ;
                 rdfs:subClassOf :Apartment .
```

类等价 类的等价意味着一个类的每个成员也必须是等价类的成员，反之亦然。换句话说，两个类涵盖了完全一样的个体集合。类等价可以通过使用 owl:equivalentClass 属性定义：

```
:Apartment       owl:equivalentClass dbpedia:Apartment .
```

这声明了在我们的公寓本体中的 :Apartment 类和从 DBpedia 导入的 dbpedia: Apartment 类等价。断言类之间的等价关系等价于断言双向的子类关系： 111

```
:Apartment        rdfs:subClassOf dbpedia:Apartment .
dbpedia:Apartment rdfs:subClassOf :Apartment .
```

小插曲：双关语 你可能注意到 DBpedia apartment 的定义来源于 dbpedia 命名空间，而不是源自 dbpedia-owl。真正的它不是一个类，而是一个个体。

对比我们的本体，DBpedia 在更高的抽象层次上描述了公寓。DBpedia 本体中的类不试图分类个体实体（例如 Amsterdam 的公寓），而是分类个体主题。将个体作为类称为元建模（meta-modeling）。

虽然 OWL2 的直接语义不允许元建模，OWL2 DL 通过一个称为双关语（或"文字游戏"）的语法技巧绕过这个限制。这意味着，当 URI `dbpedia:Apartment` 出现在类公理定义中时，它就被当做类，而当它出现在个体断言中时，它就被当做个体。

双关语允许用在下面的情况中：类的名字、个体的名字和属性的名字可以自由地互换。但是对象属性名和数据类型属性名不可以混用。

枚举 最直接（尽管无意义且计算昂贵）的方式来定义类，可以通过显式的枚举它包含的所有个体：

```
:BaronWayRooms rdf:type       owl:Class;
               owl:oneOf    ( :BaronWayKitchen
                              :BaronWayBedroom1
                              :BaronWayBedroom2
                              :BaronWayBedroom3
                              :BaronWayLivingroom
                              :BaronWayBathroom
112                           ... ) .
```

这定义了 Amsterdam 所有公寓的类。或许，如果已知的成员列表很长，这种类定义方式是非常繁重的，而且如果我们当前不能知道所有的个体，这种方式甚至是不可行的。例如，我们可能决定把 `:BaronWayBedroom1` 和 `:BaronWayBedroom2` 之间的墙敲掉，来创建一个新的房间。

不相交类 类的不相交性意味着一个类的成员不可能也是另一个类的成员。通过两个类描述的个体集合不含交集。我们可以通过使用 `owl:disjointWith` 属性来声明 `:LuxuryApartment` 类和 `:ColdWaterFlat`[⊖]不相交。

```
:LuxuryApartment owl:disjointWith :ColdWaterFlat .
```

这意味着没有 `:LuxuryApartment` 可以同时是 `:ColdWaterFlat`。

补 一个类 A 的补 C 是所有不属于 A 的事物的类。换句话说，A 和 C 的并集等价于 `owl:Thing`。注意，这意味着补总是 A 的不相交类的超集。记住互补性是一个很强大的建模结构，必须小心使用这点很重要。例如：

```
:FurnishedApartment rdfs:subClassOf  :Apartment .
:UnFurnishedApartment  rdfs:subClassOf :Apartment ;
                       owl:complementOf :FurnishedApartment .
```

这声明了有家具的公寓类是没有家具的公寓类的补。如果我们的本体包含除了公寓之外的其

113 他类，那么就有问题了。例如，如果我们额外声明：

```
:SemiDetached owl:disjointWith :Apartment .
```

⊖ 一个冷水公寓是没有热水供应的公寓。

:SemiDetached 类将会为空。为什么？如果类 :Apartment 同时涵盖 :Funished-Apartment 和它的补，那么 :Apartment 将等价于 owl:Thing；不可能存在一个个体既不属于某个类也不属于它的补。如果我们另外引入一个和 :Apartment 不相交的类，这个类在效果上和 owl:Thing 不相交。:SemiDetached 类不能包含任何个体，因此等价于 owl:Nothing。

并集和不相交并集 我们经常知道某个类等价于两个或两个以上类：该类的每个成员都至少是并集中一个类的成员。这可以使用 owl:unionOf 结构来指定。例如：

```
:Apartment    rdf:type        owl:Class ;
          owl:unionOf    ( :ColdWaterFlat
                           :LuxuryApartment
                           :PenthouseApartment
                           :StudioApartment
                           :BasementApartment
                           :FurnishedApartment
                           :UnFurnishedApartment
                         ) .
```

在许多情况下，并集的成员类是互不相交的。当然，我们可以显式地断言类之间的 owl:disjointWith 关系，但是直接声明则更加方便：

```
:Apartment    rdf:type            owl:Class;
          owl:disjointUnionOf (
                              :FurnishedApartment
                              :UnFurnishedApartment ) .
```

114

交集 类似地，我们可以声明一个类是两个或者多个类的交集：这个类的每个成员是交集中每个类的成员。例如：

```
:LuxuryApartment
          rdf:type          owl:Class ;
          owl:intersectionOf ( :GoodLocationApartment
                              :LargeApartment
                              :NiceViewApartment
                              :LuxuryBathroomApartment ) .
```

这声明了 :LuxuryApartment 类是由拥有好的地点、大空间、优美的风景以及豪华浴室的个体公寓组成。

4.4.6 属性上的类公理

OWL2 允许比我们在前面章节中看到的更加精细的控制类的定义。我们可以指定额外的类公理来约束个体集合，这些个体通过观察它们的属性被认为是某个类的成员。例如，

这使得我们自动地推导类成员关系。类限制公理通过关联到一个特别类型的匿名类（在 Turtle 中，一个 owl:Restriction）被连接到一个 owl:Class，它将所有满足限制的个体聚集在一起。

全称限制 在一个类 C 和属性 p 上的全称限制声明对于 C 的每个成员，p 的所有值都属于某个类。换句话说，全称限制可以用于指定一个属性的值域，这个值域对于受约束的类而言是局部的。这个类型的约束使用 owl:allValuesFrom 结构来建立。例如：
115

```
:LuxuryBathroomApartment
        rdf:type          owl:Class;
        rdfs:subClassOf  [ rdf:type          owl:Restriction;
                           owl:onProperty    :hasBathroom ;
                           owl:allValuesFrom :LuxuryBathroom
                         ].
```

这定义了 :LuxuryBathroomApartment 类作为个体集合的一个子类，对于 :hasBathroom 属性的取值只能是 :LuxuryBathroom 的实例。注意，一个 owl:allValuesFrom 限制仅仅声明了，如果受限类的某个成员拥有某个属性的值，那么这个值必须是特定类的成员。这个限制不要求属性必须含有任何值：在这种情况下，这个限制被无意义地满足。在我们的公寓例子中，上面的定义不要求一个豪华浴室公寓必定拥有一个浴室！

全称限制也可以用于数据类型属性——例如，声明一个属性的值必须是某种类型，或者落入某种数据值域中（见后）。

存在限制 一个类 C 和属性 p 上的存在限制声明对于 C 的每个成员，至少存在 p 的某个值属于某个特定类。这种类型的限制使用 owl:someValuesFrom 关键词来指明：

```
:LuxuryBathroomApartment
        rdf:type          owl:Class;
        rdfs:subClassOf  [ rdf:type           owl:Restriction;
                           owl:onProperty     :hasBathroom ;
                           owl:someValuesFrom :LuxuryBathroom
                         ].
```

116 **必要和充分条件** 如果不使用 rdfs:subClassOf 属性来将我们的类关联到限制，我们也可以使用 owl:equivalentClass 属性来声明这个受限类恰好是限制描述的类。rdfs:subClassOf 限制声明了类成员的必要条件，而 owl:equivalentClass 则声明了必要和充分条件。

一般而言，一个推理机只基于充分必要条件来直接推导个体的类成员关系。例如，上面的存在限制不会使得推理机得出下面的结论：每个拥有 :hasBathroom 关系到 :LuxuryBathroom 类型个体的实例必须是 :LuxuryBathroomApartment 的实例。这个公寓只是这个限制的子类，而我们没有足够的信息来确定个体也是类本身的成员。如果我

们使得这个类与限制指定的类等价，很明显任何满足限制的个体也必是类的成员。

但是，在这两种情况下，如果我们显式地声明一个个体是 :LuxuryBathroomApartment 类的一个实例，推理机将会推导出至少存在某个（未知的）:LuxuryBathroom 类型的个体作为 :hasBathroom 属性的值。

值限制 值限制用于当我们想基于与已知个体的关系或与特定数据类型属性取值的关系来定义类时。例如，我们可以定义 Amsterdam 的所有公寓的类：

```
:AmsterdamApartment
    rdf:type            owl:Class;
    owl:equivalentClass [ rdf:type        owl:Restriction;
                          owl:onProperty  dbpedia-owl:location ;
                          owl:hasValue    dbpedia:Amsterdam
                        ].
```

[117]

基数限制 一个基数限制约束了对于一个类而言某个属性可以拥有的值的数目。如果我们额外指定这些值必须属于的类，那么这个约束就被称为是限定的（qualified）。例如：

```
:StudioApartment
    rdf:type          owl:Class;
    rdfs:subClassOf [ rdf:type        owl:Restriction;
                      owl:onProperty  :hasRoom ;
                      owl:cardinality "1"^^xsd:integer
                    ].
```

这指定了一个单间公寓只能有一个 :hasRoom 属性的值。我们可以将其改为限定的基数限制，通过声明这个基数只对 :LivingRoom、:Kitchen 和 :Bedroom 类的成员成立（单间通常确实包含独立的卫生间）：

```
:StudioApartment
    rdf:type          owl:Class;
    rdfs:subClassOf [ rdf:type                 owl:Restriction;
                      owl:onProperty           :isPlayedBy ;
                      owl:qualifiedCardinality "1"^^xsd:integer ;
                      owl:onClass [ owl:unionOf (:LivingRoom :Kitchen :Bedroom) ]
                    ].
```

注意，限定限制依然允许受限类的成员拥有属性的额外值，只要这些属于受限类的补。一个在 owl:Thing 上的受限的基数限制等价于一个没有限定的约束。表 4-2 罗列了 OWL2 中允许的不同基数限制。

表 4-2 OWL2 中的基数限制

限制的类型	受限的	不受限的
Exact cardinality	owl:qualifiedCardinality	owl:cardinality
Minimum cardinality	owl:qualifiedMinCardinality	owl:minCardinality
Maximum cardinality	owl:qualifiedMaxCardinality	owl:maxCardinality

118 **数据值域限制和数据类型** 针对数据类型属性的全称和存在限制允许一个类的成员指定数据类型中的任何值作为属性的值。有时，我们需要更精确的定义，例如，允许租公寓的成人类，或者公寓的最小面积。在 OWL2 中，我们可以指定一个属性所允许的值域的限制：

```
:Adult   rdfs:subClassOf   dbpedia:Person ;
         rdfs:subClassOf [ rdf:type       owl:Restriction ;
                           owl:onProperty :hasAge ;
                           owl:someValuesFrom
                            [ rdf:type       rdfs:Datatype ;
                              owl:onDatatype xsd:integer ;
                              owl:withRestrictions (
                               [ xsd:minInclusive "18"^^xsd:integer ]
                              )
                            ]
                         ] .
```

这定义了 `:Adult` 作为人的子类，拥有 `:hasAge` 的值并属于大于等于 18 的整数值域。你可以看到这个数据值域被定义为 `rdfs:Datatype` 类型的一个匿名类。我们也可以引入一个新的命名数据类型，使得我们在本体中可以重用：

```
:AdultAge rdf:type          rdfs:Datatype ;
          owl:onDatatype    xsd:integer ;
          owl:withRestrictions (
              [ xsd:minInclusive "18"^^xsd:integer ]
                    ) .
```

119

```
:Adult   rdf:type        owl:Class ;
         rdfs:subClassOf   dbpedia:Person ;
         rdfs:subClassOf [ rdf:type         owl:Restriction ;
                           owl:onProperty     :hasAge ;
                           owl:someValuesFrom :AdultAge
                         ] .
```

OWL2 允许使用 XML 模式来定义数据类型。但是，只有使用 XML 模式刻面（facets）定义的数据类型才能用于限制中。参见建议阅读来获得更多信息。

自限制 我们都知道好的公寓可以自卖：如果它拥有好的位置，有好的风景，并且拥有合适的大小，不需要花太多时间来重新装修就可以卖得很好。在 OWL2 中，我们可以使用自

限制来表达这点。例如：

```
ex:GoodApartment
        rdf:type        owl:Class ;
        rdfs:subClassOf [ rdf:type      owl:Restriction ;
                          owl:onProperty ex:sells ;
                          owl:hasSelf    "true"^^xsd:boolean ;
                        ] .
```

它声明了 ex:GoodApartment 的每个实例都通过 ex:sells 属性关联到它自身。显然，OWL2 不允许数据类型属性的自约束。

键 数据库经常使用键来标识表中的记录。这些键不必是 URI，并且很难使用一个优雅的转换策略生成。OWL2 允许我们声明，对于某个类（看做是表），一个特定的数据类型属性（或者属性组合）的值应该被作为一个唯一标识符。例如，邮编和街道门牌号码的组合为荷兰的所有住所提供了一个唯一的标识符： 120

```
:postcode      rdf:type     owl:DatatypeProperty .
:addressNumber rdf:type     owl:DatatypeProperty .

:Dwelling
        rdf:type     owl:Class ;
        owl:hasKey ( :postcode :addressNumber ) .
```

注意，这种键的机制允许我们定义局部作用于某个类的*反函数型数据类型属性*。任意两个类型为 ex:Dwelling 的个体，如果拥有相同的 :postcode 和 :addressNumber 值，必定被认为是相同的个体。不幸的是，OWL2 DL 由于计算代价的原因，不允许我们指定全局的反函数型数据类型属性。

4.4.7 个体事实

现在我们已经有了在 OWL2 中如何定义属性和类的一般方法，现在将注意力转移到模型中的个体实体。在许多情况下，我们已经拥有了许多关于这些实体的知识，并且只需要类公理来推导额外信息。关于个体的声明通常称为断言。

类和属性断言 OWL2 中的类成员关系和属性断言与 RDF 模式中的声明方式一样：

```
:Apartment          rdf:type     owl:Class .

:BaronWayApartment rdf:type         :Apartment ;
                   :hasNumberOfRooms "4"^^xsd:integer ;
                   :isRentedBy       :Paul .
```
121

这引入了 :BaronWayApartment，它是类 :Apartment 的一个实例。它拥有 4 个房间，出租给 :Paul。记住在 OWL2 DL 的直接语义下，rdf:type 关系只能在两个严格区分

的层次下成立：即类之间和个体之间⊖。

同一性（Identity）断言 因为OWL2采用开放世界假设，我们永远不能假设拥有不同URI的两个个体是不同的实体。我们可能是处理一个拥有多个名字的单一个体。虽然我们已经在一些情况中看见了可以自动地推导出同一性关系，显式地声明它有时更加方便：

```
:BaronWayApartment owl:sameAs        :PaulsApartment ;
                   owl:differentFrom :FranksApartment .
```

不同个体的列表很容易变得很长。例如，一个小型城市已经包含了数百个公寓，使得我们需要用 `owl:differentFrom` 关系来成对地断言它们。幸运的是，我们可以使用 `owl:AllDifferent` 结构稍微优雅点地声明它们：

```
_:x  rdf:type    owl:AllDifferent ;
     owl:members ( :FranksApartment :PaulsApartment ) .
```

负断言 有时我们知道某些事物不成立。在开放世界中，将这些知识显式化很有价值：排除可能性常允许我们推导出新的知识。例如，`:BaronWayApartment` 没有出租给 `:Frank` 的知识可能允许我们推断出它不是 `:FranksApartment`：

122
```
_:x  rdf:type              owl:NegativePropertyAssertion ;
     owl:sourceIndividual  :BaronWayApartment ;
     owl:assertionProperty :isRentedBy ;
     owl:targetIndividual  :Frank .
```

如果 `owl:assertionProperty` 指向的是数据类型属性，要用 `owl:targetValue` 来代替 `owl:targetIndividual`。

注意，如果我们知道一个个体不是某个类的成员，也可以通过断言它是该类的补的成员来显式地断言这点：

```
:BaronWayApartment  rdf:type  [ owl:complementOf :LuxuryApartment ] .
```

4.5 OWL2 概要

OWL2规范包括一组称为概要的内容：它们中的一些是OWL2 DL规范中著名的子集，而另一些更具表达能力，但是不含OWL2 Full的完整语义。提供这些概要的动机是许多已有本体倾向于只使用DL中可用的语言结构的一个特定部分。基于一个较弱表达能力语言的推理能够实现推理机性能的显著提高。逻辑概要的标准库以及特别受欢迎的表达能力和计算复杂性之间的权衡在现实中很有用。

特别地，这些概要包括：

1）通过语法来限制。一个概要语法的语义由OWL2 DL规范提供。

2）通过逻辑来定义，使其能够在多项式时间内处理一些有趣的推理服务，关乎于：

⊖ 参见4.4.5节中有关对关系的讨论。

- 本体中事实的数目。
- 本体的整体大小。 [123]

本节将概述 OWL2 中定义的概要以及它们的典型应用领域。

OWL2 EL　*EL* 概要是 *EL* 描述逻辑的扩展。它的主要优势在于可以用多项式时间推理拥有大量类公理的本体，它被设计用于涵盖卫生保健和生命科学领域中多个已有的大型本体的表达能力（例如 SNOMED-CT、Gene 本体和 GALEN）。⊖

OWL2 EL 的主要优势是处理合取和存在限制。它是轻量级的语言，并且以多项式时间支持可靠且完备的推理。和 OWL2 DL 最主要的区别是它取消了 `owl:allValuesFrom` 限制，然而它还是支持属性上的 `rdfs:range` 限制，这能取得相似的效果。

OWL2 QL　为 OWL2 DL 和 OWL2 EL 开发的推理机为类公理的推理而优化，当处理拥有相对不复杂的类定义但是包含大量个体断言的本体时则相对低效。OWL2 的 QL 概要被开发用于高效地处理对这些本体的查询应答，并采用了源自关系数据库管理的技术。它基于 DL-Lite 描述逻辑并扩展了更具表达能力的特性，例如属性包含公理（owl:subPropertyOf）和函数型／反函数型对象属性。

OWL2 RL　OWL2 RL 概要基于称为描述逻辑编程（Description Logic Programming）的语言，并使得描述逻辑和规则能够交互；它是使用规则能够实现的 OWL2 DL 的最大语法部分。这是一个非常重要的特性，因为规则可以高效地并行执行，允许实现大规模推理。 [124]

OWL2 RL 区别于 QL 和 EL 概要的地方在于，它提供了 DL 视角和 OWL Full 视角之间的桥梁：规则推理机能够很容易地丢弃 OWL DL 的限制（例如类和个体之间的区别）。这意味着，OWL2 RL 的规则实现可以实现 OWL Full 的子集。面向语义网语言的大多数可伸缩的推理机中的大部分实现了 OWL2 RL 或者一个相似的称为 pD* 或 OWL-Horst 的语言。必须被实现的规则集合作为 OWL2 RL 规范的一部分发布。

4.6　小结

- OWL2 扩展了 RDF 和 RDF 模式，引入了一组表达能力更强的语言特征，例如基数约束、类的等价性、交集和并集。
- 通过 OWL 与逻辑之间的对应关系提供了形式语义和推理支持。
- OWL2 分为两类。OWL2 DL 在 OWL2 和 RDFS 语言元素的组合上施加某些限制来保留可判定性。OWL2 Full 则是 RDF 模式的完全兼容扩展，拥有所有 OWL2 语言特征，但是它被证明是不可判定的。
- 3 个概要，OWL2 EL、OWL2 QL 和 OWL2 RL，是满足计算性质要求的语法子集。特别地，OWL2 RL 可以使用基于规则的技术实现，并且已经成为在语义网上表示推理的事实标准。
- OWL2 有 4 种标准语法，RDF/XML、曼彻斯特语法、OWL/XML 和函数式语法。 [125]

⊖ 参见 http://www.snomed.org，http://www.geneontology.org 和 http://www.openclinical.org/prj_galen.html。

建议阅读

下面是学习更多有关 OWL2 语言的主要入口点：

- Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz, eds. OWL 2 Web Ontology Language Profiles.
www.w3.org/TR/owl2-profiles/.
- Boris Motik, Peter F. Patel-Schneider, and Bernardo Cuenca Grau, eds. OWL 2 Web Ontology Language Direct Semantics.
www.w3.org/TR/owl2-direct-semantics/.
- Boris Motik, Peter F. Patel-Schneider, and Bijan Parsia, eds. OWL 2 Structural Specification and Functional-Style Syntax.
www.w3.org/TR/owl2-syntax/.
- Michael Schneider, ed. OWL 2 Web Ontology Language RDF-Based Semantics.
www.w3.org/TR/owl2-rdf-based-semantics/.
- W3C OWL Working Group, eds. OWL 2 Web Ontology Language Document Overview.
www.w3.org/TR/owl2-overview/.

有关 OWL2 的基础教程和导读包括：

- Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. OWL 2 Web Ontology Language Primer.
www.w3.org/TR/owl2-primer/.
- Matthew Horridge. The Manchester Pizza Tutorial.
owl.cs.manchester.ac.uk/tutorials/protegeowltutorial/.

126 有关 OWL2 的逻辑基础的重要文献有：

- Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the EL Envelope. Proceedings of IJCAI 2005: 364–369.
- I. Horrocks, O. Kutz, and U. Sattler. The Even More Irresistible SROIQ. InProceedings of the 10th International Conference of Knowledge Representation and Reasoning (KR-2006, Lake District UK), 2006.
- Herman J. ter Horst. Combining RDF and Part of OWL with Rules: Semantics, Decidability, Complexity. International Semantic Web Conference 2005, 668–684.

一个（非常不完整的）针对 OWL 开发的软件和推理机列表（在本书写作时）：

- CEL，一个针对 OWL2 EL 概要优化的 OWL 推理机，由德累斯顿大学开发。参见 http://lat.inf.tu-dresden.de/systems/cel/。
- HermiT，一个面向复杂本体的快速推理机，由牛津大学开发。参见 http://www.hermit-reasoner.com。
- OWLIM，一个 OWL2 RL 概要的快速 OWL 推理机，由 Ontotext 开发。参见 http://www.ontotext.com/owlim。

- Pellet，特性最丰富的 OWL 推理机之一，由 Clark 和 Parsia 开发。参见 http://pellet.owldl.com。
- Protégé，OWL 本体的事实标准编辑环境，由斯坦福大学开发。它拥有多个内置推理机。参见 http://protege.stanford.edu。
- TopBraid Composer，一个 OWL 本体的基于 RDF 的编辑环境，由 TopQuadrant 开发。它支持 SPARQL，连接三元组存储库，以及使用 OWLIM 推理。参见 http://www.topquadrant.com/products/TB_Composer.html。 127
- WebPIE，一个针对 OWL 的 ter Horse 部分（笼统说 OWL2 RL）的大规模可伸缩的推理引擎，由阿姆斯特丹自由大学开发。参见 http://www.few.vu.nl/~jui200/webpie.html。

练习和项目

1. 阅读 OWL 的在线规范。

2. 用 3 种不同的方法声明两个类不相交。

3. 表达以下事实：所有数学课程只由 David Billington 讲授（没有其他讲师参与）。同时，表达以下事实：David Billington 只讲授数学课程。这两个事实的区别清楚吗?

4. 严格地说，`owl:SymmetricProperty` 定义在 OWL 中并不需要，因为它可以根据其他语言原语表达。解释如何实现（提示：也考虑 inverse）。

5. 类似的问题也适合 `owl:FunctionalProperty` 和 `owl:NegativePropertyAssertion`。展示如何使用其他语言结构来表达它们。

6. 判断通常 OWL 中哪些特征是必须的，而哪些只是为了方便但是可以通过其他建模原语来模拟。

7. 解释 `FunctionalProperty`、`InverseFunctionalProperty` 和 `inverseOf` 三个概念之间的联系。

8. 解释为什么需要声明 `owl:Class` 是 `rdfs:Class` 的一个子类。

9. 在 2.7 节，我们介绍了 RDF 的一个公理化语义。也可以为 OWL 开发一个相似的公理化语义。定义 `owl:intersectionOf` 的公理化语义。 128

10. 定义 `owl:inverseOf` 的公理化语义。

11. 在这个练习中，请你开发基数限制的一个公理化语义：

（a）定义 `noRepeatsList`。L 是一个"无重复的列表"，仅当 L 中不存在一个元素出现超过一次。这个概念不是 OWL 语言的一部分，但是它将被用于计数基数限制中的元素个数。

（b）定义 `owl:minCardinality` 和 `owl:maxCardinality` 为定义域是 `owl:Restriction` 而值域是 `:NonNegativeInteger` 的属性。

（c）给出描述 `minCardinality` 含义的公理：

如果有 *onProperty*(R, P) 和 *minCardinality*(R, n)，那么 x 是 R 的一个实例，当且仅当存在一个"无重复的列表" L，其长度 $\geq n$，满足对于所有 $y \in L$ 有 $P(x, y)$。

（d）以相似的方式表达 `owl:maxCardinality` 的含义。

12. 观察 http://www.co-ode.org/ontologies/ 中的一些本体。

13. 使用 OWL2 写一个你自己的本体。

14. OWL2 是 OWL 语言的最新版本。阅读之前版本的网页（参见 http://www.w3.org/2004/OWL/）以及一些示例本体。比较旧的 OWL 语言和 OWL2，特别注意它们的共同点和区别
129/130 之处。

15. 比较 OWL2 和第 1 版 OWL 的相关在线文档。

逻辑与推理：规则

5.1 引言

5.1.1 逻辑与规则

从一个抽象的角度来说，第 2 章和第 4 章的主题都与知识的表示（representation of knowledge）相关：关于万维网资源内容的知识，关于一个论域中的概念及其相互联系（本体）的知识。

早在万维网出现之前，知识表示就已经在人工智能领域和更早的哲学领域中被研究了很长时间。事实上，可以追溯到古希腊。亚里士多德被认为是逻辑之父。逻辑仍然是知识表示的基石，特别是谓词逻辑（predicate logic），也称为一阶逻辑（first-order logic）。下面是逻辑之所以流行和重要的原因：

- 它提供了一种高级语言，以一种透明的方式来描述知识。并且，它有很强的描述能力。
- 它有一种很好理解的形式语义，将无歧义的含义赋予逻辑语句。 131
- 对逻辑结论有精确的定义，以确定一个语句在语义上是否可以从另一组语句（前提）推出。事实上，逻辑的首要原始动机就是研究逻辑结论的客观规律。
- 已经有了一些证明系统，可以在语法上自动地从一组前提推导出语句。
- 已经有了一些证明系统，在其内部，语义逻辑推理和语法推导是一致的。证明系统应当是正确的（所有推导的语句都能在语义上从前提推出）和完备的（前提的所有逻辑结论都能在证明系统中推导）。
- 谓词逻辑很特殊，因为正确和完备的证明系统是存在的。描述能力更强的逻辑（高阶逻辑）没有这样的证明系统。
- 由于证明系统的存在，有可能跟踪通向逻辑结论的证明过程。在这个意义上，逻辑可以提供答案的解释。

RDF 和 OWL2 语言（除了 OWL2 Full）可以视作谓词逻辑的专门化。

这种专门化的语言存在的一个缘由是它们提供了适合于特殊用途（在我们的场景中，即基于标签的万维网语言）的语法。另一个主要缘由是它们定义了合理的逻辑子集。如前所述，在逻辑的描述能力和计算复杂性之间有一个权衡问题：语言的描述能力越强，相应的证明系统就越低效。我们已经提到过，大多数 OWL 的变种对应于一个描述逻辑，是谓词逻辑的一个存在高效证明系统的子集。 132

谓词逻辑的另一个存在高效证明系统的子集由规则系统（rule system）组成，也叫做 *Horn 逻辑*（Horn logic）或*有定逻辑程序*（definite logic program）。规则的形式如下：

$$A_1, \ldots, A_n \rightarrow B$$

其中，A_i 和 B 是原子公式。事实上，有两种直观理解这条规则的方式：

1）如果已知 $A_1, \cdots, A_n$ 为真，那么 B 也为真。以这种方式解释的规则称为*演绎式规则*（deductive rule）[㊀]。

2）如果条件 $A_1, \cdots, A_n$ 为真，那么执行动作 B。以这种方式解释的规则称为*反应式规则*（reactive rule）。

两种视角都有重要应用。然而，在本章中，我们采用演绎式方法。我们研究的是语言、可以问的可能查询以及合适的答案。此外，我们概述能返回这些答案的一套证明机制的运作。

有趣的是，描述逻辑和 Horn 逻辑在某种意义上是正交的，即它们都不是对方的子集。例如，在描述逻辑中，不可能定义出一个表示快乐配偶的类，即那些与自己最好朋友结婚的人。但这一知识可以容易地使用规则来表示：

$$married(X, Y), bestFriend(X, Y) \rightarrow happySpouse(X)$$

另一方面，规则（通常）不能断言：（a）类的否定 / 补集；（b）析取 / 并集的信息（例如，
133 一个人或者是男人或者是女人）；（c）存在量词（例如，所有人都有父亲）。相反，OWL 能够描述类的补集和并集，以及某些形式的存在量词。

接下来，我们将注意力转移到另一种规则。给出一个简单的例子。假设一个在线供应商想在顾客生日当天提供一个特殊折扣。一种简单的使用规则来表示这一业务策略的方式如下：

*R*1：如果是生日，则有特殊折扣。

*R*2：如果不是生日，则没有特殊折扣。

当生日已知时，这一解决方案可以正常工作。但想象一下，一个顾客出于隐私考虑拒绝提供他的生日。在这种情况下，之前的规则就没法运用了，因为它们的前提未知。为了应对这种情况，我们需要这样写：

*R*1：如果是生日，则有特殊折扣。

*R*2′：如果生日未知，则没有特殊折扣。

然而，规则 *R*2′ 的前提超出了谓词逻辑的描述能力。因此，我们需要一种新型的规则系统。我们注意到，基于规则 *R*1 和 *R*2 的解决方案在我们拥有完整信息时（例如，或者是生日或者不是生日）是可以工作的。这种新型的规则系统将在可获得的信息不完整时找到应用场景。

谓词逻辑及其特例是*单调的*（monotonic），其含义如下。只要能得出一个结论，即便加入了新的知识，它仍然有效。但如果规则 *R*2′ 用来推导“没有特殊折扣”这一结论，那么之后当顾客的生日变为未知且恰与购买日期相同时，这一结论可能就变为无效了。这一行为称为*非单调*（nonmonotonic），因为新信息的加入导致了结论的丢失。因此，我们对非单调规则和

㊀ 原则上讲，有两种方式运用演绎规则：从体（$A_1, \cdots, A_n$）到结论（B）（*前向链*，forward chaining），以及从结论（目标）到体（*后向链*，backward chaining）。

单调规则（是谓词逻辑的一种特例）加以区分。在本章中，我们同时讨论单调和非单调规则。 134

5.1.2 语义网上的规则

规则技术已经存在几十年了，已被广泛应用于实践中，并且已经较为成熟。这样的部署已经催生了各种各样的方法。结果是，在（语义）网的情境中标准化这一领域越发困难。W3C 的一个工作组已经开发出了规则交换格式（Rule Interchange Format，RIF）标准。了解它和 RDF 与 OWL 的区别是很重要的：后两者作为直接表达知识的语言，而 RIF 被设计为主要用来在不同应用之间交换规则。例如，一家在线商店可能希望一些智能系统可以存取它以规则描述的定价、退款和隐私策略。语义网方法是用一种我们已经讨论过的万维网语言来以一种机器可存取的方式描述知识。

鉴于潜在目标是提供服务于不同规则系统之间的交换格式，RIF 组合了很多特征，所以相当复杂。因此有一些质疑其是否真的会作为描述知识的主要语言被广泛使用。事实上，那些希望为语义网开发规则系统的人们有各种选择：

- RDF 上的规则可以使用 SPARQL 构件来优雅地描述；在这个方向上，一项近期的提案是 SPIN[⊖]。
- 如果希望在遇到丰富语义结构时使用规则，可以使用 SWRL，它结合了 OWL DL 的功能和某些规则。
- 如果希望用 OWL 来建模，但又想使用规则技术来实现，可以使用 OWL2 RL。

方法上的多元性正是本章看起来和之前几章非常不同的原因，之前几章是基于一个或一簇非常稳定并广泛使用的标准。RIF 正朝着这个方向发展，但尚未同等地得到社区的一致认可和采纳。因此，本章在一个宽泛的层面陈述观点，并陈述一些具体方法。 135

本章概述

建议那些对谓词逻辑的记号和基本概念感到不适的读者，去读一些逻辑书籍的导论章节，例如本章最后罗列的那些。或者，可以跳过 5.3 节和 5.4 节的一些部分。

- 5.2 节提供了一个使用单调规则（即称为 Horn 逻辑的谓词逻辑的子集）的例子。
- 5.3 节和 5.4 节描述了 Horn 逻辑的语法和语义。
- 5.5 节讨论了 OWL2 RL 和规则之间的联系。
- 5.6 节陈述了 RIF 方言簇，并特别关注基于逻辑的语言。
- 5.7 节描述了 SWRL，其作为一种结合规则和描述逻辑的方式。
- 5.8 节简要描述了怎样使用 SPARQL 构造子来建模规则。
- 5.9 节描述了非单调规则的语法，5.10 节陈述了非单调规则的一个例子。
- 5.11 节简要描述了 RuleML，它是一项正在进行的为了在万维网上标记规则的活动，它有一个开放的和实验性的日程表，可能会在未来形成新的标准。 136

⊖ 然而，SPARQL 并不是一种规则语言，因为它本质上执行的是规则的一个应用。规则系统必须建立在例如 SPIN 的上层。

5.2 单调规则的例子：家庭关系

想象一个数据库包含关于一些家庭关系的事实。假设数据库包含关于如下基础谓词（base predicate）的事实：

mother(X,Y) X 是 Y 的母亲

father(X,Y) X 是 Y 的父亲

male(X) X 是男的

female(X) X 是女的

接下来，我们可以使用合适的规则来推出更多的联系。首先，我们可以定义一个 *parent* 谓词：一位 *parent* 或者是一位父亲或者是一位母亲。

$$mother(X,Y) \to parent(X,Y)$$

$$father(X,Y) \to parent(X,Y)$$

接下来，我们可以将拥有共同父亲（或母亲）的男人定义为兄弟：

$$male(X), parent(P,X), parent(P,Y), notSame(X,Y) \to$$
$$brother(X,Y)$$

notSame 谓词指不相同；我们假设这些事实被保存在一个数据库中。当然，每个实际的逻辑系统都提供了方便地描述相同和不相同的方式，但我们选择这个抽象的解决方案以使得讨论更具一般性。

类似地，*sister* 如下定义：

$$female(X), parent(P,X), parent(P,Y), notSame(X,Y) \to$$
137 $$sister(X,Y)$$

叔叔是父母亲的兄弟：

$$brother(X,P), parent(P,Y) \to uncle(X,Y)$$

祖母是父母亲的妈妈：

$$mother(X,P), parent(P,Y) \to grandmother(X,Y)$$

祖先或者是父母亲，或者是父母亲的祖先：

$$parent(X,Y) \to ancestor(X,Y)$$

$$ancestor(X,P), parent(P,Y) \to ancestor(X,Y)$$

5.3 单调规则：语法

我们来考虑一条简单的规则，说明所有超过 60 岁的忠实顾客都有权享受一项特殊折扣：

$$loyalCustomer(X), age(X) > 60 \rightarrow discount(X)$$

我们区分一些规则成分：

- 变量（variable）是值的占位符：X。
- 常量（constant）指代固定的值：60。
- 谓词（predicate）将对象关联起来：*loyalCustomer*，>。
- 函数符号（function symbol）在作用于特定参数时指称一个值：*age*。

如果不使用任何函数符号，我们讨论的就是无函数（Horn）逻辑。 138

5.3.1 规则

规则的形式是：

$$B_1, \ldots, B_n \rightarrow A$$

其中，A、$B_1,\cdots,B_n$ 是原子公式。A 是规则的头（head），$B_1,\cdots,B_n$ 是规则的前提（premise）。集合 $\{B_1,\cdots,B_n\}$ 称为规则的体（body）。

规则体中的逗号理解为合取：如果 B_1 and B_2 and $\cdots$ and B_n 为真，那么 A 也为真（或者等价地说，要证明 A，只要证明所有 $B_1,\cdots,B_n$ 就足够了）。

注意，变量可能出现在 A、$B_1,\cdots,B_n$ 中。例如，

$$loyalCustomer(X), age(X) > 60 \rightarrow discount(X)$$

这条规则用于任何顾客：如果一个顾客碰巧是忠实的并且超过 60 岁，那么她就会得到折扣。换句话说，变量 X 是隐式全称量化的（使用 $\forall X$）。一般而言，出现在规则中的所有变量都是隐式全称量化的。

总之，一条规则 r

$$B_1, \ldots, B_n \rightarrow A$$

被解释为下述公式，记作 $pl(r)$：

$$\forall X_1 \ldots \forall X_k((B_1 \wedge \ldots \wedge B_n) \rightarrow A)$$

或者等价地，

$$\forall X_1 \ldots \forall X_k(A \vee \neg B_1 \vee \ldots \vee \neg B_n)$$

其中，$X_1,\cdots,X_k$ 是所有出现在 A、$B_1,\cdots,B_n$ 中的变量。 139

5.3.2 事实

事实是一个原子公式，例如 *loyalCustomer*(a345678) 阐述了 ID 为 a345678 的顾客是忠实的。一条事实中的变量是隐式全称量化的。

5.3.3 逻辑程序

逻辑程序 P 是一个由事实和规则组成的有限集合。它的谓词逻辑转换 $pl(P)$ 是 P 中所有规则和事实的谓词逻辑解释的集合。

5.3.4 目标

目标指的是向一段逻辑程序提出的一个查询 G。它的形式是

$$B_1, \ldots, B_n \rightarrow$$

如果 $n=0$，就称为空目标（empty goal）。

我们的下一个任务是在谓词逻辑中解释目标。借助我们目前已经知道的手法（将逗号解释为合取，隐式全称量化），我们得到如下解释：

$$\forall X_1 \ldots \forall X_k(\neg B_1 \vee \ldots \vee \neg B_n)$$

这个公式与 $pl(r)$ 几乎相同，唯一的不同之处是规则头 A 被省略了[⊖]。

谓词逻辑中的一种等价表示是

$$\neg\exists X_1 \ldots \exists X_k(B_1 \wedge \ldots \wedge B_n)$$

其中，$X_1,\cdots,X_k$ 是出现在 $B_1,\cdots,B_n$ 中的所有变量。我们来简要解释这个公式。假设我们
140 知道

$$p(a)$$

并且，我们的目标是

$$p(X) \rightarrow$$

事实上，我们想要知道是否存在一个让 p 为真的值。我们预期会得到一个肯定的答案，因为有事实 $p(a)$。因此，$p(X)$ 是存在量化的。但接下来，为什么我们将公式取否定了呢？原因是使用了数学中的一种叫做反证法（proof by contradiction）的证明技术。这项技术证明语句 A 可以从语句 B 推出的方法是假设 A 为假，并结合 B 推导出一个矛盾。那么，A 就必然可以从语句 B 推出。

在逻辑编程中，我们要证明一个目标可以被肯定回答的方式是，否定这个目标，再证明可以通过使用逻辑程序得到一个矛盾。例如，给定逻辑程序

$$p(a)$$

和目标

$$\neg\exists X p(X)$$

我们得到一个逻辑矛盾：第二个公式说没有元素拥有属性 p，但第一个公式说值 a 拥有属性 p。

⊖ 注意这个公式等价于 $\forall X_1,\cdots,X_k(false \vee \neg B_1 \vee \cdots \vee \neg B_n)$，因此，失去的规则头可以被认为是矛盾项。

因此，$\exists X p(X)$ 可以从 $p(a)$ 推出。

5.4 单词规则：语义

5.4.1 谓词逻辑语义

一种回答查询的方式是使用规则、事实和查询的谓词逻辑解释，并利用众所周知的谓词逻辑的语义。具体而言，给定一段逻辑程序 P 和一个查询 141

$B_1, \ldots, B_n \to$

及其包含的变量 $X_1, \cdots, X_k$，我们对这个查询的回答是肯定的，当且仅当

$$pl(P) \models \exists X_1 \ldots \exists X_k (B_1 \wedge \ldots \wedge B_n) \quad (1)$$

或者等价地，当

$$pl(P) \cup \{\neg \exists X_1 \ldots \exists X_k (B_1 \wedge \ldots \wedge B_n)\} \quad (2)$$

不可满足时。换句话说，当程序 P 的谓词逻辑解释和查询的谓词逻辑解释不可满足（存在矛盾）时，我们给出一个肯定的答案。

谓词逻辑的语义概念的形式化定义在文献中可以找到。这里我们仅给出一个非形式化的陈述。逻辑语言（签名）的组件可能具有任何我们希望的含义。每个谓词逻辑模型 A 都赋予一种特定的含义。特别地，它的组成包括

- 一个域（domain）$dom(A)$，是一个非空的对象集合，公式构成了关于它们的语句。
- 对应域中每个常量的元素。
- 对应每个函数符号的 $dom(A)$ 上的具体函数。
- 对应每个谓词的 $dom(A)$ 上的具体关系。

当 = 符号用来指示相等时（即它的解释是固定的），我们讨论的就是带等式的 Horn 逻辑（Horn logic with equality）。逻辑连接符 ¬、∨、∧、→、∀、∃ 根据它们的直观含义定义为：非、或、与、蕴涵、任取、存在。这样我们定义，当一个公式在模型 A 中为真时，记作 $A \models \varphi$。

公式 φ 可以从公式集合 M 中推出，当 φ 在所有 M 为真的模型 A 中都为真时（即 M 中的所有公式在 A 中都为真）。 142

现在，我们能够解释（1）和（2）了。无论我们怎样解释 P 中出现的常量、谓词和函数符号以及查询，只要 P 的谓词逻辑解释为真，$\exists X_1 \cdots \exists X_k (B_1 \wedge \cdots \wedge B_n)$ 也一定为真，即变量 $X_1, \cdots, X_k$ 有值使得所有原子公式 B_i 都为真。

例如，假设 P 是逻辑程序

$p(a)$

$p(X) \to q(X)$

考虑查询

$q(X) \to$

显然，$q(a)$ 可以从 $pl(P)$ 推出。因此，$\exists X q(X)$ 可以从 $pl(P)$ 推出，所以 $pl(P) \cup \{\neg \exists X q(X)\}$ 是不可满足的，于是我们给出一个肯定的答案。但如果考虑查询

$q(b) \to$

那么必须给出一个否定的答案，因为 $q(b)$ 不能从 $pl(P)$ 推出。

5.4.2 最小 Herbrand 模型语义

另一种逻辑程序的语义——最小 Herbrand 模型语义，要求更多的技术讨论，可以在标准的逻辑教科书（参见建议阅读）中找到描述。

5.4.3 闭证据和参数化证据

到目前为止，我们已经关注了查询的是 / 否答案。然而，这种查询未必是最优的。假设
143 我们有如下事实

$p(a)$

以及查询

$p(X) \to$

肯定的答案是正确的但并不令人满意。它很像一个笑话中提到的，你被问到"你知道现在几点吗？"，你看了看你的手表回答"是的"。在我们的例子中，合适的答案是一个替代

$\{X/a\}$

其给出了 X 的一个实例化，并同时给出了肯定的答案。常量 a 叫做闭证据（ground witness）。给定以下事实

$p(a)$

$p(b)$

对同样的查询有两个闭证据：a 和 b。或者等价地，我们应该返回替代

$\{X/a\}$

$\{X/b\}$

尽管是有价值的，闭证据却并不总是最优答案。考虑逻辑程序

$add(X, 0, X)$

$add(X, Y, Z) \to add(X, s(Y), s(Z))$

如果我们将 s 理解为"后继函数"，其返回值是其参数值加 1，那么这个程序做的是加法计算。

add 的第三个参数计算前两个参数的和。考虑查询 [144]

$add(X, s^8(0), Z) \rightarrow$

可能的闭证据根据以下替代来确定

$\{X/0, Z/s^8(0)\}$

$\{X/s(0), Z/s^9(0)\}$

$\{X/s(s(0)), Z/s^{10}(0)\}$

$\ldots$

然而，参数化证据（parameterized witness）$Z=s^8(X)$ 是以下存在查询的最概括的方式

$\exists X \exists Z\ add(X, s^8(0), Z)$

因为它表示的事实是，只要 Z 的值等于 X 的值加 8，$add(X, s^8(0), Z)$ 就为真。

最具概括性的证据的计算是一个叫做 SLD 归结的证明系统的首要目标，其陈述超出了本书的范围。

5.5 OWL2 RL：当描述逻辑遇见规则

如本章开始时所述，Horn 逻辑和描述逻辑是正交的。当尝试将它们集成进一个框架中时，最简单的方法是考虑两个逻辑的交集，即一种语言中可以用一种保持语义的方式转换到另一种语言的那个部分，且反之亦也成立。本质上，OWL2 RL 旨在捕获 OWL 的这个片段。这种方法的优势包括：

- 从建模者的角度来说，可以为了建模的目的，根据建模者的经验和偏好，来自由地使用 OWL 或者规则（以及相关的工具和方法）。 [145]
- 从实现的角度来说，描述逻辑推理机和演绎规则系统都可以使用。因此，有可能使用一个框架来建模，比如 OWL，然后使用另一个框架的推理机，比如规则。这一特性给各种工具提供了额外的灵活性并保证了互操作性。

在本节的剩余部分中，我们展现许多 RDF 模式和 OWL2 RL 的构件是如何使用 Horn 逻辑来描述的，并讨论一些通常不能描述的构件。这一讨论关注的是那些能凸显规则和描述逻辑之间联系与区别的构件。想要了解更多的关于 OWL2 RL 构件及其与逻辑的关系的信息，请查阅 5.6.4 节以及本章最后的建议阅读。

我们以 RDF 和 RDF 模式开头。一个形如 (a, P, b) 的 RDF 三元组可以描述为一个事实

$P(a, b)$

类似地，一个形如 $type(a,C)$ 的实例声明——阐述了 a 是类 C 的实例，可以描述为

$C(a)$

C是D的子类这一事实可以容易地描述为

$$C(X) \rightarrow D(X)$$

对于子属性也是类似的。最后，定义域和值域约束也可以用Horn逻辑描述。例如，以下规则阐述了C是属性P的定义域：

$$P(X,Y) \rightarrow C(X)$$

146 现在我们来看OWL。*equivalentClass*(C, D)可以用一对规则来描述

$$C(X) \rightarrow D(X)$$

$$D(X) \rightarrow C(X)$$

对*equivalentProperty*也是类似的。一个属性P的传递性可以容易地描述为

$$P(X,Y), P(Y,Z) \rightarrow P(X,Z)$$

现在我们来看布尔运算。我们可以如下阐述类C_1和C_2的交集是D的子类：

$$C_1(X), C_2(X) \rightarrow D(X)$$

从另一个方向，我们可以阐述C是D_1和D_2的交集的子类：

$$C(X) \rightarrow D_1(X)$$

$$C(X) \rightarrow D_2(X)$$

对于并集，我们可以用以下这对规则来描述C_1和C_2的并集是D的子类：

$$C_1(X) \rightarrow D(X)$$

$$C_2(X) \rightarrow D(X)$$

然而，其反方向超出了Horn逻辑的描述能力。为了描述C是D_1和D_2的并集的子类这一事实，需要在相应的规则头中出现析取，在Horn逻辑中这是不允许的。注意，有一些情况下转换是可能的。例如，当D_1是D_2的子类时，那么规则$C(X) \rightarrow D_2(X)$足以描述C是D_1和D_2的并集的子类。但问题在于没有一种适合于所有情况的转换方式。

147 最后，我们需要讨论OWL中的一些约束形式。OWL语句

```
:C rdfs:subClassOf [ rdf:type owl:Restriction ;
                     owl:onProperty :P ;
                     owl:allValuesFrom :D ] .
```

可以用Horn逻辑如下描述：

$$C(X), P(X,Y) \rightarrow D(Y)$$

然而，其反方向一般而言没法描述。此外，OWL语句

```
[ rdf:type           owl:Restriction ;
  owl:onProperty     :P ;
  owl:someValuesFrom :D ] rdfs:subClassOf :C .
```

可以用 Horn 逻辑如下描述：

$$P(X,Y), D(Y) \rightarrow C(X)$$

其反方向一般而言无法描述。

并且，基数约束和类的补集一般而言也不能用 Horn 逻辑描述。

5.6 规则交换格式：RIF

5.6.1 概述

规则技术至今已经存在几十年了，并日趋多样化（例如，动作规则、一阶规则、逻辑编程）。因此，W3C 规则交换格式工作组的目标并非去开发一个新的能适合各种目的的规则语言，而是致力于万维网上多样的（已有的或未来的）规则系统之间的交换。采用的方法是开发一簇语言，称为*方言*（dialect）。RIF 定义了两种方言： 148

1）*基于逻辑的方言*（logic-based dialect）。这些用来覆盖那些基于某种逻辑的规则语言，例如，一阶逻辑和各种有着对否定的不同解释的逻辑编程方法（回答集编程、良基语义等）。这条分支上目前已经开发出的具体的方言是：

- RIF *核心*（Core），本质上对应于无函数的 Horn 逻辑。
- RIF *基本逻辑方言*（Basic Logic Dialect，BLD），本质上对应于带等式的 Horn 逻辑。

2）*带动作的规则*（rule with action）。这些用来覆盖那些产生式系统和反应式规则。这条分支上目前已经开发出的具体的方言是：*产生式规则方言*（Production Rule Dialect，PRD），又称 RIF-PRD。

RIF 簇被设计为具有*通用性*和*可扩展性*。通用性的实现是通过所有的 RIF 方言的语法和语义都共享一些基本原则。可扩展性指的是未来的方言可以被开发并添加到 RIF 簇中的可能性。对于基于逻辑的方言，RIF 工作组通过开发*逻辑方言框架*（Framework for Logic Dialect，FLD）来支持通用性和可扩展性，允许通过实例化方法的各种参数来指定各种规则语言。这一框架是一项重要成果，但超出了本书的范围。接下来，我们将要陈述 RIF-BLD 的基本思想。

在这之前，需要说明的是，RIF 工作组的大量（如果不是绝大部分的话）工作都致力于语义层面。当然，在语法层面（例如，使用 XML）通过使用一个逻辑系统的各种语法特征和 RIF 之间的映射，也能做到规则的交换。但主要目的是以一种*语义保持的方式*（semantic preserving way）来交换规则。 149

5.6.2 RIF-BLD

RIF 基本逻辑方言基本上对应于带等式的 Horn 逻辑并附加上：

- 数据类型和一些内置。

• 框架。

RIF-BLD 和其他 RIF 的变种一样，希望支持一个通用的常用数据类型、谓词和函数的集合。这个集合包括数据类型（例如，整数、布尔、字符串、日期）、“内置”谓词（例如，数值大于、前缀、日期小于）和取值为这些数据类型的函数（例如，数值减法、替换、时间中的小时）。

举例来说，假设我们希望描述一条规则，来陈述如果一个演员在至少 5 年的跨度内拍摄了超过 3 部的成功电影，那么他就是一个电影明星。并且，一部成功的电影是指获得了好评（例如，在 10 分制评级中超过 8 分）或者经济上很成功（票房超过了 1 亿美元）。这些规则应该在 DBpedia 数据集上执行。

这些规则可以用 RIF-BLD 描述如下：

```
Document(
    Prefix(func <http://www.w3.org/2007/rif-builtin-function#>
    Prefix(pred <http://www.w3.org/2007/rif-builtin-predicate#>
    Prefix(rdfs <http://www.w3.org/2000/01/rdf-schema#>
    Prefix(imdbrel <http://example.com/imdbrelation#>
    Prefix(dbpedia <http://dbpedia.org/ontology/>
    Prefix(ibdbrel <http://example.com/ibdbrelation#>
Group(
    Forall ?Actor ?Film ?Year (
        If And(   dbpedia:starring(?Film ?Actor)
                  dbpedia:dateOfFilm(?Film ?Year)
     Then dbpedia:starredInYear(?Film ?Actor ?Year)
)
Forall ?Actor (
    If ( Exists ?Film1 ?Film2 ?Film3 ?Year1 ?Year2 ?Year3
            And (   dbpedia:starredInYear(?Film1 ?Actor ?Year1)
                    dbpedia:starredInYear(?Film2 ?Actor ?Year2)
                    dbpedia:starredInYear(?Film3 ?Actor ?Year3)
                    External ( pred:numeric-greater-than(
                        External(func:numeric-subtract ?Year1 ?Year3) 5)))
            dbpedia:successful(?Film1)
            dbpedia:successful(?Film2)
            dbpedia:successful(?Film3)
            External (pred:literal-not-identical(?Film1 ?Film2))
            External (pred:literal-not-identical(?Film1 ?Film3))
            External (pred:literal-not-identical(?Film2 ?Film3))
                    )
            Then dbpedia:movieStar(?Actor)
```

```
        )
        Forall ?Film (
              If Or (
                    External(pred:numeric-greater-than(
                         dbpedia:criticalRating(?Film 8))
                    External(pred:numeric-greater-than(
                         dbpedia:boxOfficeGross(?Film) 100000000)))
              Then dbpedia:successful(?Film)
        )
    )
)
```

150 / 151

这个例子演示了数据类型和内置的使用。注意运用内置谓词时使用的 *External*，以及将一些规则放在一起的 *Group*。

RIF 的语法非常简单，尽管相当冗长（当然，也有一种基于 XML 的语法来支持规则系统之间的交换）。变量名以问号开头。符号 =、# 和 ## 分别用来描述相等、类属和子类联系。

框架（frame）的使用在面向对象语言和知识表示中有很悠久的历史，在规则语言（例如 F-Logic）领域也很突出。其基本思想是将对象表示为框架，将它们的属性表示为槽（slot）。例如，我们可能有一个教授类，它的槽如姓名、办公室、电话、院系等。这些信息在 RIF-BLD 中的描述如下：

```
oid[slot1 -> value1 ... slotn -> valuen]
```

5.6.3 与 RDF 和 OWL 的兼容性

RIF 的一个主要特征是其与 RDF 和 OWL 标准兼容，即可以在 RIF、RDF 和 OWL 文档的一个组合上进行推理。因此，RIF 促进交换的不仅仅是规则，也是 RDF 图和 OWL 公理。

结合 RIF 和 RDF 的基本思想是使用 RIF 框架公式来表示 RDF 三元组：三元组 *s p o* 被表示为 s[p->o]。其语义定义是该三元组得到满足，当且仅当相应的 RIF 框架公式也得到满足。例如，如果以下 RDF 三元组

```
ex:GoneWithTheWind ex:FilmYear ex:1939
```

为真，那么以下这条 RIF 事实也为真

```
ex:GoneWithTheWind[ex:FilmYear -> ex:1939]
```

152

给定如下的 RIF 规则（阐述的是 1930 ~ 1968 年间使用的好莱坞产品代码）：

```
Group(
   Forall ?Film (
      If And( ?Film[ex:Year -> ?Year]
            External(pred:dateGreaterThan(?Year 1930))
            External(pred:dateGreaterThan(1968 ?Year)))
```

```
        Then ?Film[ex:HollywoodProductionCode -> ex:True]
    )
)
```

可以得出的结论是

```
ex:GoneWithTheWind[ex:HollywoodProductionCode -> ex:True]
```

以及相应的 RDF 三元组。

类似技术被用来实现 OWL 和 RIF 之间的兼容性。主要特征是：

- OWL 和 RIF 的语义是兼容的。
- 可以从 OWL 公理和 RIF 知识的某些结合来推出结论。
- OWL2 RL 可以用 RIF 实现（参见下一节）。

5.6.4 用 RIF 描述 OWL2 RL

一个一阶规则的集合部分地描述 OWL2 RL，它们为使用规则技术的一个实现建立了基础。为了建立规则系统和 OWL2 RL 本体之间的互操作性，这一公理化可以使用 RIF（BLD，
153 实际上即使是更简单的核心）规则来描述。

OWL2 RL 规则可以分为 4 个（不相交的）类别：三元组模式规则、不一致性规则、列表规则和数据类型规则。

三元组模式规则 这些规则从 RDF 三元组模式的合取中推导某些 RDF 三元组。这些规则到 RIF 的转换（使用框架公式）很简单，使用如下形式的规则⊖：

```
Group(
    Forall ?V1 ... ?Vn(
        s[p->o] :- And(s1[p1->o1]... sn[pn->on]))
)
```

不一致性规则 这些规则指明了原始 RDF 图（当然，是对于现有的 OWL 知识而言）中的不一致。这样的规则可以容易地使用 RIF 来表示为带有 *rif:error* 结论的规则，*rif:error* 是 RIF 命名空间中的一个谓词符号，用来描述不一致性。例如，当两个谓词已经被声明为不相交，但又连接到同样的实体时，一条不一致就发生了。这可以使用 RIF 如下描述：

```
Group(
    Forall ?P1 ?P2 ?X ?Y(
        rif:error :- And(
            ?P1[owl:propertyDisjointWith ?P2] ?X[?P1->?Y] ?X[?P2->?Y]))
)
```

列表规则 一些 OWL2 RL 规则涉及包括 RDF 列表（例如 `owl:AllDifferent`）的 OWL 描述的处理。有两种方法可以使用 RIF 来描述这些规则。一种可以在运行时使用递归规

⊖ 为提升可读性，这些规则以 Prolog（反向）表示法给出，而不是之前一直使用的 If-Then（正向）表示法。

则来遍历 RDF 图，产生一个通用的表示。或者，可以采用一种预处理方法，规则对于实际出 154
现在输入 RDF 图中的列表直接实例化，可能在实际中表现更好。请读者阅读转换文档（参见建议阅读）来获取细节。

数据类型规则 这些规则提供了类型检测和所支持的数据类型中的带类型文字的值相等 / 不相等检测。例如，这样的规则对于数据类型中的相同的值（例如 1 和 1.0）可以推导出 `owl:sameAs` 三元组，或者当一个文字被指定为一个数据类型的实例，但它的取值超出了那个数据类型的值空间时，可以推导出一条不一致。到 RIF 规则的转换是非常简单的。

5.7 SWRL

SWRL（语义网规则语言）是一种设计出的结合了 OWL DL 和无函数 Horn 逻辑的语义网语言，是用一元 / 二元 Datalog RuleML（参见 5.11 节）写的。因此，它允许类似 Horn 的规则和 OWL DL 本体的结合。

SWRL 规则的形式如下：

$$B_1, \ldots, B_n \to A_1, \ldots, A_m$$

其中箭头两侧的逗号表示合取，$A_1,\cdots,A_m$ 和 $B_1,\cdots,B_n$ 可以形如 $C(x)$、$P(x,y)$、$sameAs(x,y)$ 或者 $differentFrom(x,y)$，其中 C 是一个 OWL 描述，P 是一个 OWL 属性，x 和 y 是 Datalog 变量、OWL 实例或者 OWL 数据值。

如果一个规则头拥有超过一个原子（当它是没有共享变量的若干原子的合取时），这条规则可以容易地被转换成一个等价的规则集合，其中每个规则头含有一个原子。

SWRL 语言的主要复杂性源于任意的 OWL 描述——例如限制，都可以出现在一条规则 155
的头或体中。这一特征为 OWL 添加了很强的描述能力，但也付出了不可判定性的代价，即没有推理引擎能推出和 SWRL 语义一样的结论。

与 OWL2 RL 相比，SWRL 在集成描述逻辑和无函数规则上位于其对立面。OWL2 RL 采用一种非常保守的方式，试图去结合两种语言的共同子语言的优点。SWRL 采用一种更加激进的方法，合并了它们的描述能力。从实际的角度出发，挑战在于识别出 SWRL 的子语言，以能够在描述能力和计算易处理性之间找到正确的平衡。这样的一个候选子语言是 OWL DL 的带 DL 安全规则（DL-safe rule）的扩展，其中每个变量必须出现在规则体中的一个非描述逻辑原子中。参见建议阅读中的关于集成规则与描述逻辑的论文。

注意，一个与 SWRL 相似的结果可以通过结合 RIF-BLD 和 OWL2 RL 来获得（参见建议阅读）。

5.8 用 SPARQL 描述规则：SPIN

规则可以使用 SPARQL 来描述，使用其 CONSTRUCT 特征。例如，规则

$$grandparent(X, Z) \leftarrow parent(Y, Z), parent(X, Y)$$

可以被描述为：

```
CONSTRUCT {
    ?X grandParent ?Z.
}   WHERE {
    ?Y parent ?Z.
    ?X parent ?Y.
```

156
```
}
```

新近提出的 SPIN 以此为起点提出了一个基于 SPARQL 的建模语言。它的主要思想和特征包括：

- 使用面向对象建模的思想将规则关联到类。因此，规则可能表示那个类的行为，并且它们自身可能并不独立存在（尽管全局规则可以被定义）。
- 使用 SPARQL CONSTRUCT、DELETE、INSERT 以及使用 SPARQL ASK 构造子的约束来描述规则。
- 使用模板为规则提供抽象机制，其本质上封装了参数化的 SPARQL 查询；用户定义的 SPIN 函数作为一种机制基于更简单的组件来构建更高层的规则（复杂的 SPARQL 查询）。

作为示例，OWL2 RL 规则已经使用 SPIN 来描述了。例如，规则

$C_2(X) \leftarrow C_1(X), equivalentClass(C_1, C_2)$

可以用 SPARQL 表示为：

```
CONSTRUCT {
    ?X a ?C2.
}
WHERE {
    ?X a ?C1.
    ?C1 equivalentClass ?C2.
```

157
```
}
```

并被实例化为类 `owl:Thing` 上的一条 `spin:rule`，这将允许这条规则被运用于所有可能的实例。

应该注意的是，SPARQL 及 SPIN 是一个规则语言，不是一个实现的规则系统；例如，CONSTRUCT 只能描述一个推理步骤（从一个 RDF 图到另一个）。一个基于 SPARQL 的规则系统（例如一个 SPIN 推理引擎）则至少需要迭代地执行 CONSTRUCT，并且当存在递归规则时控制递归。

5.9 非单调规则：动机和语法

5.9.1 漫谈

现在我们将注意力集中到非单调规则系统上。到目前为止，只要一条规则的前提得以证明，这条规则就可以被运用，它的头会作为一条结论推导出来。在非单调规则系统中，即使一条规则的所有前提都已知，它也未必会被运用，因为我们不得不考虑对立的推理链。一般而言，从现在起，我们所考虑的规则称为*可废止的*（defeasible），因为它们可以被其他规则击败。为了支持规则间的冲突，*否定的原子公式*（negated atomic formula）可以出现在规则的头和体中。例如，我们可以写成

$p(X) \to q(X)$

$r(X) \to \neg q(X)$

为了区分可废止规则和标准的单调规则，我们用一种不同的箭头：

$p(X) \Rightarrow q(X)$

$r(X) \Rightarrow \neg q(X)$

在这个例子中，也给定如下事实 158

$p(a)$

$r(a)$

我们既推不出 $q(a)$ 也推不出 $\neg q(a)$。这是一个典型的两条规则互相阻止对方的例子。这一冲突可以通过使用*规则间的优先级*（priorities among rules）来消解。假设我们通过某种方式知道第一条规则比第二条规则要强，那么我们就可以推导出 $q(a)$。

优先级在实际中自然形成，可能基于各种各样的原则：

- 一条规则的来源可能比另一条规则的来源更可靠，或者它可能具有更高的权威性。例如，联邦法律优于州立法律；在商业管理中，高层管理比中层管理具有更高的权威。
- 一条规则可能因为它更新而优于另一条。
- 一条规则可能因为它更加具体而优于另一条。一个典型的例子是带有若干例外情况的一条通用的规则，在这样的情况下，例外就比通用规则更强。

具体性经常基于给定的规则来计算，但另两条原则不能从逻辑形式化来确定。因此，我们从特定的优先级原则中抽象出来，假设在规则集上存在一个*外部的优先级关系*（external priority relation）。为了从语法上描述这个关系，我们扩展规则语法来加入一个独特的标记。例如，

$r_1 : p(X) \Rightarrow q(X)$

$r_2 : r(X) \Rightarrow \neg q(X)$

可以写成

159 $r_1 > r_2$

来指明 r_1 比 r_2 强。

我们并不在“>”上施加太多条件，甚至不要求规则形成一个完整的序。我们只要求优先级关系是无环的，即不可能有如下形式的圈：

$r_1 > r_2 > \ldots > r_n > r_1$

注意，优先级是用来消解竞争规则（competing rule）之间的冲突的。在简单情况下，两条规则是竞争的，仅当一条规则头是另一条规则头的否定。但在应用中，经常出现的情况是只要一个谓词 p 推导出来了，一些其他的谓词就要被排除出去。例如，一个投资顾问可能根据投资人愿意承受的风险级别的不同将他的推荐分为三层：低、中和高。每个投资人在任何时刻只允许接受一个风险级别。技术上来说，这些情况是通过为每个文字 L 维护一个冲突集 $C(L)$ 来建模的。$C(L)$ 总是包含 L 的否定，但可以包含更多的文字。

5.9.2 语法定义

可废止规则（defeasible rule）的形式如下：

$r: L_1, \ldots, L_n \Rightarrow L$

其中 r 是标记（label），$\{L_1,\cdots,L_n\}$ 是体（body）或前提（premise），L 是规则头（head）。$L, L_1,\cdots,L_n$ 是肯定或否定的文字（一个文字是一个原子公式 $p(t_1,\cdots,t_m)$ 或其否定 $\neg p(t_1,\cdots,t_m)$）。函数符号不能出现在规则中[①]。有时候我们将一条规则的头记为 $head(r)$，体记为 $body(r)$。稍微滥用一下，有时候也用标记 r 来指代整条规则。

可废止逻辑程序（defeasible logic program）是一个三元组 $(F, R, >)$，由一个事实集 F、一个有限的可废止规则集 R 和一个 R 上的无环二元关系“>”（准确地说，一个 $r>r'$ 对构成的集合，
160 其中 r 和 r' 是 R 中规则的标记）组成。

5.10 非单调规则的例子：交易中介

这个例子展现了规则怎样应用于电子商务应用（其理想地运行于语义网上）中。交易中介通过一个独立的第三方——中间人来进行。中间人匹配买方的需求和卖方的能力，提出一笔双方都满意的交易。

作为一个具体应用，我们将讨论公寓租赁[②]——一种往往是乏味和费时的常见活动。合适的万维网服务可以很大程度上减少工作。我们首先呈现潜在租户的需求。

> Carlos 正寻找一间至少 45 平方米、至少带有两间卧室的公寓。如果位于三楼或者更高，这幢楼就必须有电梯。并且，必须允许饲养宠物。

① 施加这一约束是因为技术上的原因，其讨论超出了本章的范围。

② 在这个例子中，房东扮演了抽象卖方的角色。

Carlos希望为位于市中心的45平方米的公寓支付300美元，而对于郊区的类似公寓则支付250美元。此外，他愿意为更大的公寓额外支付每平方米5美元，为一个花园额外支付每平方米2美元。

他总共不能支付超过400美元。如果有几个选择，他将挑最便宜的那个选项；他第二优先考虑的是花园；最后考虑的是额外的空间。

5.10.1 Carlos的需求的形式化

我们使用以下谓词来描述公寓的属性：

apartment(x)	表明 x 是一间公寓 161
size(x,y)	y 是公寓 x 的面积（单位：平方米）
bedroom(x,y)	x 有 y 间卧室
price(x,y)	y 是 x 的价格
floor(x,y)	x 在第 y 层
garden(x,y)	x 有一个面积为 y 的花园
elevator(x)	x 的楼里有一部电梯
pets(x)	x 允许饲养宠物
central(x)	x 位于市中心

我们也使用以下谓词：

acceptable(x)	公寓 x 满足Carlos的需求
offer(x,y)	Carlos愿意为公寓 x 支付 y 美元

现在我们陈述Carlos的硬性需求。任何公寓预先都被认为是可接受的。

$$r_1: apartment(X) \Rightarrow acceptable(X)$$

然而，只要Carlos的一个需求无法满足，X 就是不可接受的。

$$r_2: bedrooms(X,Y), Y < 2 \Rightarrow \neg acceptable(X)$$

$$r_3: size(X,Y), Y < 45 \Rightarrow \neg acceptable(X)$$

$$r_4: \neg pets(X) \Rightarrow \neg acceptable(X)$$

$$r_5: floor(X,Y), Y > 2, \neg lift(X) \Rightarrow \neg acceptable(X)$$

$$r_6: price(X,Y), Y > 400 \Rightarrow \neg acceptable(X)$$

规则 $r_2 \sim r_6$ 是规则 r_1 的例外，所以增加

$$r_2 > r_1, r_3 > r_1, r_4 > r_1, r_5 > r_1, r_6 > r_1$$

接下来，计算Carlos愿意为一间公寓支付的价格。 162

$$r_7: size(X,Y), Y \geq 45, garden(X,Z), central(X) \Rightarrow$$

$$offer(X, 300 + 2Z + 5(Y - 45))$$

$$r_8:\ size(X,Y), Y \geq 45, garden(X,Z), \neg central(X) \Rightarrow$$
$$offer(X, 250 + 2Z + 5(Y - 45))$$

一间公寓是可以接受的仅当 Carlos 愿意支付的钱数不少于房东指定的价格（我们假设不能讨价还价）。

$$r_9:\ offer(X,Y), price(X,Z), Y < Z \Rightarrow \neg acceptable(X)$$

$$r_9 > r_1$$

5.10.2 可获得的公寓的表示

每间可获得的公寓都被赋予一个唯一的名字，它的属性被表示为事实。例如，公寓 a_1 可能如下描述：

$bedrooms(a_1, 1)$

$size(a_1, 50)$

$central(a_1)$

$floor(a_1, 1)$

$\neg elevator(a_1)$

$pets(a_1)$

$garden(a_1, 0)$

$price(a_1, 300)$

可获得的公寓的描述汇总在表 5-1 中。在实际中，可提供的公寓可以存储在一个关系数
163 据库中，或者在语义网环境下，存储在一个 RDF 存储系统中。

表 5-1 可获得的公寓

公寓	卧室	面积	市中心	楼层	电梯	宠物	花园	价格
a_1	1	50	yes	1	no	yes	0	300
a_2	2	45	yes	0	no	yes	0	335
a_3	2	65	no	2	no	yes	0	350
a_4	2	55	no	1	yes	no	15	330
a_5	3	55	yes	0	no	yes	15	350
a_6	2	60	yes	3	no	no	0	370
a_7	3	65	yes	1	no	yes	12	375

如果匹配 Carlos 的需求和可获得的公寓，我们发现

- 公寓 a_1 不可接受，因为它只有一间卧室（规则 r_2）；
- 公寓 a_4 和 a_6 不可接受，因为不允许饲养宠物（规则 r_4）；
- 对于 a_2，Carlos 愿意支付的是 300 美元，但它的价格超出了（规则 r_7 和 r_9）；
- 公寓 a_3、a_5 和 a_7 是可接受的（规则 r_1）。

5.10.3 选择一间公寓

到目前为止，我们已经识别出了 Carlos 可以接受的公寓。这个选择本身是有价值的，因为它将注意力集中于相关的公寓，从而有可能接下来进行手工检查。但通过考虑更多的偏好，也有可能进一步降低数量，甚至降到只剩一间公寓。Carlos 的偏好按序分别是价格、花园面积、面积。我们如下表示它们：

$$r_{10}: acceptable(X) \Rightarrow cheapest(X)$$

$$r_{11}: acceptable(X), price(X,Z), acceptable(Y), price(Y,W),$$
$$W < Z \Rightarrow \neg cheapest(X)$$

$$r_{12}: cheapest(X) \Rightarrow largestGarden(X)$$ 164

$$r_{13}: cheapest(X), gardenSize(X,Z), cheapest(Y),$$
$$gardenSize(Y,W), W > Z \Rightarrow \neg largestGarden(X)$$

$$r_{14}: largestGarden(X) \Rightarrow rent(X)$$

$$r_{15}: largestGarden(X), size(X,Z), largestGarden(Y),$$
$$size(Y,W), W > Z \Rightarrow \neg rent(X)$$

$$r_{11} > r_{10},\ r_{13} > r_{12},\ r_{15} > r_{14}$$

规则 r_{10} 表明每间可接受的公寓默认来说都是最便宜的。然而，如果有一间可接受的公寓比 X 更便宜，规则 r_{11}（比 r_{10} 更强）将起作用并推出 X 不是最便宜的。

类似地，规则 r_{12} 和 r_{13} 从最便宜的公寓中选择拥有最大花园的那些公寓。在它们之中，规则 r_{14} 和 r_{15} 再基于公寓面积选择了推荐的公寓来租赁。

在我们的例子中，公寓 a_3 和 a_5 是最便宜的。它们之中，a_5 拥有最大的花园。注意在这个例子中，公寓面积标准并不起作用：r_{14} 仅对 a_5 有用，而规则 r_{15} 没有运用。因此，就做出了一个选择，Carlos 很快会搬进去。

5.11 RuleML

RuleML（规则标记语言）是一项致力于开发万维网上规则标记的长期努力。它实际上并

不是一门语言，而是一簇规则标记语言，对应于不同种类的规则语言：推导规则、完整性约束、反应规则，等等。RuleML 簇的内核是 Datalog，即无函数的 Horn 逻辑。

RuleML 实验性地研究规则语言的各种特征，还远没有被标准化（例如非单调规则）。其想法是这些努力可以进入未来的标准中，就如同 RuleML 的结果曾是 RIF 开发中的一个重要
165 组件那样。

RuleML 簇提供了 XML 方式的规则标记语言的描述，以 RELAX NG 或 XML 模式的方式（或者对于更早的版本是文档类型定义）。规则成分的表达是简单的。图 5-1 描述了 Datalog RuleML 的关键词汇表。

规则成分	RuleML
fact	Asserted Atom
rule	Asserted Implies
head	then
body	if
atom	Atom
conjunction	And
predicate	Rel
constant	Ind
variable	Var

图 5-1 RuleML 词汇表

使用 RuleML 词汇来描述规则是简单的。例如，规则"如果一个优质顾客购买了一件奢侈产品，那么这个顾客得到的折扣是 7.5%"用 RuleML 1.0 如下表示。

```
<Implies>
  <then>
    <Atom>
      <Rel>discount</Rel>
      <Var>customer</Var>
      <Var>product</Var>
      <Ind>7.5 percent</Ind>
    </Atom>
  </then>
  <if>
```

166

```
    <And>
      <Atom>
        <Rel>premium</Rel>
        <Var>customer</Var>
      </Atom>
      <Atom>
        <Rel>luxury</Rel>
```

```
                <Var>product</Var>
            </Atom>
        </And>
    </if>
</Implies>
```

5.7 节中介绍的 SWRL 语言是 RuleML 的一个扩展，并且它的使用是直接的。举例而言，对于这条规则

$brother(X,Y), childOf(Z,Y) \rightarrow uncle(X,Z)$

用基于 RuleML 1.0 的 SWRL 的 XML 语法表示如下：

```
<ruleml:Implies>
    <ruleml:then>
        <swrlx:individualPropertyAtom swrlx:property="uncle">
            <ruleml:Var>X</ruleml:Var>
            <ruleml:Var>Z</ruleml:Var>
        </swrlx:individualPropertyAtom>
    </ruleml:then>
    <ruleml:if>
        <ruleml:And>
            <swrlx:individualPropertyAtom swrlx:property="brother">
                <ruleml:Var>X</ruleml:Var>
                <ruleml:Var>Y</ruleml:Var>
            </swrlx:individualPropertyAtom>                                   167
            <swrlx:individualPropertyAtom swrlx:property="childOf">
                <ruleml:Var>Z</ruleml:Var>
                <ruleml:Var>Y</ruleml:Var>
            </swrlx:individualPropertyAtom>
        </ruleml:And>
    </ruleml:if>
</ruleml:Implies>
```

5.12 小结

- （语义）网上的规则形成了一个非常丰富且异构的景象。
- Horn 逻辑是谓词逻辑的一个支持高效推理的子集。它形成了一个正交于描述逻辑的子集。Horn 逻辑是单调规则的基础。
- RIF 是一个万维网上规则的新标准。它的逻辑方言 BLD 基于 Horn 逻辑。
- OWL2 RL 本质上是描述逻辑和 Horn 逻辑的交集，可以嵌入 RIF 中。
- SWRL 是一种更加丰富的规则语言，结合了描述逻辑的特征和规则的受限类型。

- 当可获得的信息不完整时，非单调规则是有用的。它们是那些可以被相反证据（其他规则）覆盖的规则。
- 优先级用来消解非单调规则之间的一些冲突。

168 • 用类似 XML 的语言来表达规则，例如 RIF 和 RuleML 提供的那样，是直接的。

建议阅读

Horn 逻辑在逻辑中是一个标准的主题。更多的信息可以在相关教科书中找到，例如：

- E. Burke and E. Foxley. *Logic and Its Applications*. Upper Saddle River, N.J.: Prentice Hall, 1996.
- M. A. Covington, D. Nute, and A. Vellino. *Prolog Programming in Depth*. 2nd ed. Upper Saddle River, N.J.: Prentice Hall, 1997.
- A. Nerode and R. A. Shore. *Logic for Applications*. New York: Springer, 1997.
- U. Nilsson and J. Maluszynski. *Logic, Programming and Prolog*, 2nd ed. New York: Wiley, 1995.
- N. Nissanke. *Introductory Logic and Sets for Computer Scientists*. Boston: Addison-Wesley, 1998.

非单调规则是一个相当新的主题。信息可以在上述第二本教科书以及以下论文中找到：

- G. Antoniou, D. Billington, G. Governatori, and M. J. Maher. Representation Results for Defeasible Logic. *ACM Transactions on Computational Logic* 2 (April 2001): 255–287.
- N. Bassiliades, G. Antoniou, and I. Vlahavas. A Defeasible Logic Reasoner for the Semantic Web. *International Journal on Semantic Web and Information Systems* 2,1 (2006): 1–41.
- T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining Answer Set Programming with Description Logics for the SemanticWeb. In *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning*
169 (KR '04), AAAI Press 2004, 141–151.
- D. Nute. Defeasible Logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming* Vol. 3, ed. D. M. Gabbay, C. J. Hogger, and J. A. Robinson. New York: Oxford University Press, 1994.

关于 RIF 及其与 RDF 和 OWL 的兼容性的信息可以在以下网址找到：

- http://www.w3.org/2005/rules/wiki/Primer.
- http://www.w3.org/TR/rif-overview/.
- http://www.w3.org/TR/rif-bld/.
- http://www.w3.org/TR/rif-rdf-owl/.

对 RIF-BLD 的形式化基础的介绍可以在以下文献中找到：

M. Kifer. Knowledge Representation and Reasoning on the Semantic Web: RIF. In *Handbook of Semantic Web Technologies*, eds. J. Domingue, D. Fensel, and J. Hendler. Springer 2011.

关于 OWL2 RL 及其嵌入 RIF 的信息可以在以下网址找到：

- http://www.w3.org/TR/owl2-profiles/.

- http://www.w3.org/TR/rif-owl-rl/.

关于 SPIN 及其对部分 OWL2 RL 的编码的信息可以在以下网址找到：

- http://www.w3.org/Submission/spin-overview/.
- http://topbraid.org/spin/owlrl-all.html.

关于 SWRL 的信息可以在以下网址找到：

- http://www.w3.org/Submission/SWRL/.

其他在集成规则与描述逻辑方面的重要工作包括： 170

- I. Horrocks, P. Patel-Schneider, S. Bechhofer, and D. Tsarkov. OWL Rules: A Proposal and Prototype Implementation. *Journal of Web Semantics* 3,1 (2005): 23–40. www.websemanticsjournal.org/ps/pub/2005-2.
- B. Motik, U. Sattler, and R. Studer. Query Answering for OWL-DL with Rules. *Journal of Web Semantics* 3,1 (2005): 41–60. www.websemanticsjournal.org/ps/pub/2005-3.
- R. Rosati. On the Decidability and Complexity of Integrating Ontologies and Rules. *Journal of Web Semantics* 3,1 (2005): 61–73. www.websemanticsjournal.org/ps/pub/2005-4.

关于规则标记语言及它们在语义网中的应用的一般信息可以在 RuleML 网站找到：

- www.ruleml.org/.

练习和项目

以下项目使用规则技术。假设读者已经具备了基本的语义网技术的知识和经验。如果没有，最好先做一些基本的项目（参见第 7 章）。

语义中介

1. 这个基本的项目可以由两个人在 2 或 3 周内完成。目的是实现一个类似于 5.10 节中公寓租赁例子的应用。完成以下任务：

（a）选择一个中介活动将要进行的主题。这里中介指的是匹配供给和需求。 171

（b）为这个领域构建一个 RDFS 本体。

（c）用一些供给来实例化本体，用 RDF 描述。

（d）使用非单调规则来描述选择标准。

（e）使用一个诸如 DR-DEVICE [1]或者 DR-Prolog [2]的引擎来运行你的规则及 RDF/RDFS 信息。为此，你将需要用这些系统指定的格式来描述规则。

2. 这个高级的项目可以由 2 或 3 个人在一个学期内完成，目的是在一个多智能体环境中实现一个中介场景。除了执行项目 5.4 中描述的步骤以外，项目参与者至少需要：

[1] http://lpis.csd.auth.gr/systems/dr-device.html.

[2] http://www.csd.uoc.gr/~bikakis/DR-Prolog/.

(a) 通过学习一些相关文献来建立对多智能体环境中的中介的基本理解：

K. Sycara, S. Widoff, M. Klusch, and J. Lu. Larks: Dynamic Matchmaking among Heterogeneous Software Agents in Cyberspace. *Autonomous Agents and Multi-Agent Systems* 5, 2 (2002): 173–203.

G. Antoniou, T. Skylogiannis, A.Bikakis, and N. Bassiliades. A Deductive Semantic Brokering System.In *Proceedings of the 9th International Conference on Knowledge-Based Intelligent Information and Engineering Systems*. LNCS 3682, Springer 2005,746–752.

172 (b) 自己去选择并熟悉一个多智能体系统。我们已经在 JADE[㊀]上有了很好的经验。

(c) 决定智能体之间交换的准确消息。

(d) 了解如何去远程调用要使用的推理引擎。

证明层

这些项目的目标是实现一个证明层，其概况已经在第 1 章中简要概述了。注意，并不存在单一的证明层，而是每个选择的语义网推理系统（逻辑）都有一个证明层。然而，一些考虑对于所有这样的系统都是通用的。可以为两种可能的逻辑系统实现一个证明层。

- 一种简单的单调规则语言，例如 Datalog（没有函数符号的 Horn 逻辑），其中可以使用推理工具 Mandarax[㊁]；
- 一种非单调规则语言，像本章中讨论的那样，其中可以使用 DR-DEVICE[㊂]或者 DR-Prolog[㊃]。

3. 这个基本的项目可以由 2 或 3 个人在大约两个月内完成。目的是开发一个交互式系统向用户提供解释。需要涉及的重要方面包括以下几点。

(a) 决定如何从整体证明轨迹中抽取相关的信息。你可以查阅自动推理和逻辑编程的文献来获取想法。

(b) 为表示证明过程定义颗粒度级别。应该展现整个证明还是仅仅一些元步骤？当用户问到某一个步骤时，这些可以重定义。

173 (c) 最终，一个证明的“叶子”将会是 RDF 事实、规则或者用到的推理条件。

4. 4 或 5 个人可以在一个学期内完成这个项目。目标是在一个多智能体环境中实现一个证明层——即请求和证明部分将在智能体之间交换。额外需要考虑的包括以下方面。

(a) 自己去选择并熟悉一个多智能体系统。我们已经在 JADE[㊄]上有了很好的经验。

(b) 用一种 XML 语言表示证明，最好是通过扩展 RuleML。

174 (c) 决定用于在智能体之间交换的精确消息。

㊀ http://jade.tilab.com/。

㊁ http://mandarax.sourceforge.net/。

㊂ http://lpis.csd.auth.gr/systems/dr-device.htm。

㊃ http://www.csd.uoc.gr/~bikakis/DR-Prolog/。

㊄ http://jade.tilab.com/。

第 6 章

A Semantic Web Primer, Third Edition

应　用

最近几年，语义网技术的运用已经显著加速，而语义网应用刚萌芽的那几年（2000 – 2007 年）则主要是被未全面投产的工业和学术原型系统主导。近年来（从 2008 年起），在商业和贸易的很多领域，已经出现了全面投产的、核心基于语义网技术的系统。本章将简要描述一些这样的应用。

本章概述

我们介绍 GoodRelations 本体的部署是如何开始改变在线零售领域的功能的（6.1 节），BBC 是如何利用语义网技术来维护和发布他们的艺术家和音乐档案的（6.2 节）以及服务于他们高级别体育赛事的报道的（6.3 节），政府是如何使用语义网技术来发布他们的数据的（6.4 节），以及以《纽约时报》为例的出版业是如何使用语义网技术的（6.5 节）。最后，我们通过 175
介绍 Sig.ma（6.6 节）、OpenCalais（6.7 节）和 Schema.org（6.8 节）来讨论万维网搜索的未来。

6.1 GoodRelations

6.1.1 背景

电子商务，特别是企业对消费者（Business-to-Consumer，B2C）的电子商务，已经成为万维网快速融入日常生活的主要驱动力之一。如今，在店面和货车上看见 URL 已经不稀奇了。以美国为例，B2C 市场已经从 2000 年 4 月的 8700 万英镑增长到了 2009 年年底的 684 亿英镑，在 10 年中增长了 1000 倍。

这一巨大的电子商务市场受到传统万维网所含缺陷的影响：电子商务网站通常从结构化的信息系统中生成，罗列价格、可获得性、产品类型、交付选项等。但当这些信息抵达公司的网页中时，已经被转换为 HTML，所有机器可理解的结构已经消失了。其结果是机器没法再区分一个价格和一个产品代码。搜索引擎在解释它们试图去爬取和索引的电子商务网页时受此影响，不能正确地区分产品类型或者形成有意义的产品分组。

GoodRelations[1]是与 OWL 相符的本体，描述了电子商务领域。它可以用来描述产品的供给、指明价格、描述业务，等等。GoodRelations 的 RDFa 语法允许这一信息被嵌入现有的网页中，使得它们可以被其他计算机处理。GoodRelations 的主要好处及其日益普及的主要驱动力是它对搜索的改进。将 GoodRelations 添加到网页中，可以改进供给在现代搜索引擎和推荐 176

[1] http://www.heppnetz.de/projects/goodrelations/。

系统中的能见度。GoodRelations 支持万维网上产品和服务的标注，其采用的方式可以被搜索引擎用来向它们的用户传递一种更好的搜索体验。它支持非常具体的搜索查询并给出非常精确的答案。

除了关于产品和提供者的信息以外，GoodRelations 本体也允许描述商业和电子商务交易的功能细节，例如符合条件的国家、支付和交付选项、数量折扣、营业时间等。

GoodRelations 本体[㊀]包括了诸如 `gr:ProductOrServiceModel`、`gr:PriceSpecification`、`gr:OpeningHoursSpecification` 和 `gr:DeliveryChargeSpecification` 等类，以及 `gr:typeOfGood`、`gr:acceptedPaymentMethods`、`gr:hasCurrency` 和 `gr:eligibleRegions` 等属性。

6.1.2　样例

我们展现一个简单但真实的例子，来自于德国销售聚会服饰的 Karneval Alarm 商店的网页。一个特定的网页[㊁]描述了一件超人服饰，尺寸是 48/50，售价为 59.90 欧元。这个产品（在 Karneval Alarm 的目录中编号为 935）用 RDF 实体 `offering_935` 来表示。这个产品的网页使用了 RDFa 语法，包含以下 RDF 语句：

```
offering_935 gr:name "Superman Kostum 48/50" ;
   gr:availableAtOrFrom http://www.karneval-alarm.de/#shop ;
   gr:hasPriceSpecification UnitPriceSpecification_935 .
UnitPriceSpecification_935 gr:hasCurrency "EUR" ;
                           gr:hasCurrencyValue "59.9" ;
177                        gr:valueAddedTaxIncluded "true" .
```

这些描述也可以使用 Sindice 的 RDFa 检查器[㊂]在线浏览[㊃]。图 6-1 呈现了完整的图，或者使用一个在线可缩放的视图[㊄]。

当然，这样的 GoodRelations 标注需要提及产品类型和种类。为此，Product 本体[㊅]描述了来自维基百科网页的产品。

举个例子

```
pto:Party_costume a owl:Class;
      rdfs:subClassOf gr:ProductOrService;
      rdfs:label "Party Costume"@en;
      rdfs:comment """A party costume is clothing..."""@en
```

其允许我们叙述

```
offering_935 gr:typeOfGood pto:Party_Costume
```

㊀ http://www.heppnetz.de/ontologies/goodrelations/v1。

㊁ http://www.karneval-alarm.de/superman-m.html。

㊂ http://inspector.sindice.com/。

㊃ http://www.karneval-alarm.de/superman-m.html。

㊄ http://inspector.sindice.com/inspect?url=http%3A//www.karneval-alarm.de/superman-m.html#GRAPH。

㊅ http://www.productontology.org/。

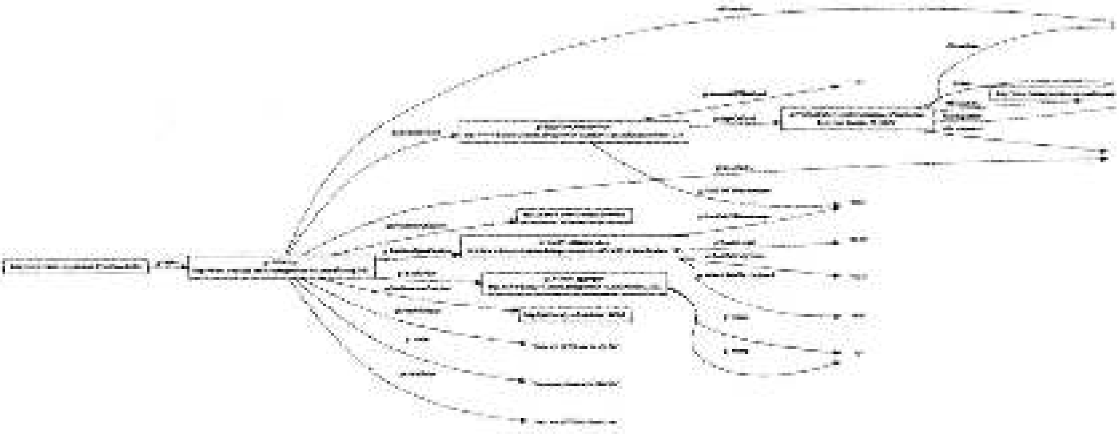


图 6-1　从一件超人服饰的产品网页中提取出的 RDFa 标记

Product 本体包括了几十万个产品的 OWL DL 类定义。这些类定义与维基百科紧耦合；维基百科中的编辑会反映在 Product 本体中。这意味着，如果一个供应商销售一件没有列在 Product 本体中的产品，他们可以为它在维基百科上创建一个网页，其产品类型将在 24 小时内出现在 Product 本体中。 178

6.1.3　运用

第一家大规模采用 GoodRelations 本体的公司是百思买（BestBuy）——一家美国大型消费电器零售商。百思买报告了其采用了 GoodRelations 的网页的搜索流量增长了 30%，点击率也大幅提升。Google 正推荐使用 GoodRelations 来语义标注电子商务网页，Yahoo 和 Google 正在爬取 RDFa 语句并用它们来增强其搜索结果的呈现。在写作本书时，其他采用者还包括 Overstock.com 零售商、O'Reilly Media 出版社、Peek & Cloppenburg 服装连锁店以及一些小型企业，诸如 Robinson Outdoors 和之前提到的 Karneval Alarm。在写作本书时，Sindice 语义搜索引擎列出了 27.3 万个带 GoodRelations 词汇标注的网页。

6.1.4　著作

Martin Hepp. GoodRelations: An Ontology for Describing Products and Services Offerson the Web. In Proceedings of the 16th International Conference on KnowledgeEngineering and Knowledge Management (EKAW2008). Acitrezza, Italy. September29 – October 3, 2008. Springer LNCS, Vol 5268, 332–347. 179

6.2　BBC 艺术家

6.2.1　背景

BBC Music Beta 项目[①]是 BBC 的一项努力，构建关于有歌曲在 BBC 电台播放的艺术家

① http://www.bbc.co.uk/music/artists.

和歌唱家的语义链接和标注网页。在这些网页中，数据集增强了，并与语义元数据互联起来，让音乐爱好者可以探索他们可能不曾知道的音乐家之间的联系。

此前，BBC 的编辑们不得不在它们发布的每个艺术家的网页上写上（并保持更新）有趣而相关的内容。取而代之，在 Music Beta 项目中，BBC 正从外部网站（诸如 MusicBrainz 和维基百科）引入信息，并将这些信息集成进他们的网页中。

为此，BBC 已经采用了 RDF 标准，并已经将它自己的数据模式与 MusicBrainz 发布的数据模式建立了映射，以利用 MusicBrainz 提供的唯一标识符。这将允许 BBC 网站利用公开领域的内容，扩充在那找到的档案网页。MusicBrainz 是一个开放内容音乐“元数据库”，列出了超过 40 万个艺术家的信息；维基百科网页上的信息盒（info-box）中包含的信息被 DBpedia 所捕获。

6.2.2 样例

作为一个例子，我们看一看 John Lennon 的 BBC Artist 网页⊖。初看起来，这个网页像一个普通的带有诸如传记信息、专业信息的艺术家信息的网页，例如 John Lennon 合作过的艺术家，他加入过的乐队，专辑的评论，等等。但这个网页也是一个丰富的 RDF 三元组集，使用
180 John Lennon 在 MusicBrainz 的标识符 `4d5447d7` 作为一个资源来表述这样的信息：

```
5c014631#artist foaf:name "John Lennon"


4d5447d7#artist bio:event _:node15vknin3hx2
_:node15vknin3hx2 rdf:type bio:Death
_:node15vknin3hx2 bio:date "1980-12-08"


4d5447d7#artist foaf:made _:node15vknin3hx7
_:node15vknin3hx7 dc:title "John Lennon/Plastic Ono Band"
4d5447d7#artist owl:sameAs dbpedia:John_Lennon
```

其表述了一个叫做 John Lennon 的艺术家死于 1980 年 12 月 12 日，他录制了一张叫做“John Lennon/Plastic Ono Band”的唱片，并且他也被 `dbpedia:John_Lennon` 这个 URI 标识。

BBC Artist 关于 John Lennon 的网页的完整内容包含 60 个三元组，基于此并使用描述家庭关系、音乐领域、时间和日期、地理信息、社交关系等 40 个不同的本体可以推理出另外 300 个三元组。

完整的网站包含大约 40 万个艺术家网页、16 万个外部链接和 10 万条艺术家到艺术家的关系。

语义网技术的使用，例如使用 URI 作为标识符并将其与外部语义数据提供者配准，意味着网页的创建和维护所需要的人力只是过去的一小部分。有趣的是，BBC 不仅消费了这样的

⊖ http://www.bbc.co.uk/music/artists/4d5447d7-c61c-4120-ba1b-d7f471d385b9。

信息资源，也将它们回馈给全世界。简单地将.rdf添加到任何一个BBC Artist网页的URI，实际上将会提供这个网页所基于的RDF数据。通过以这种方式发布RDF，BBC正在将他们的数据提供给想要使用它的第三方。

当然，当使用公开信息作为输入时，总是面临信息包含错误这样的风险。在这种情况下，BBC没有内部修复那些错误。相反地，他们会在诸如MusicBrainz和DBpedia等外部来源上修复错误。这不仅同时修复了BBC网站上的错误，也为这些数据源的任何其他用户修复了错误，因此为提升可公开获得的数据源的质量做出了贡献。 181

6.2.3 运用

BBC Artist项目是"BBC推进的，从在多种遗留内容生产系统中构建的网页向事实上发布那些我们可以在整个万维网上用一种更动态的方式使用的数据进行迁移的一个普遍运动中的一部分"。BBC正将他们网站的其他部分构建了包含节目信息㊀的链接数据标准和资源，包括节目的URI和使用BBC的节目本体㊁以及来源于BBC的使用了其自己的WildLife本体㊂的广泛的高质量的自然节目的野生动物信息。

6.3 BBC世界杯2010网站

6.3.1 背景

除了广播电台和电视节目以外，BBC将大量的努力用于构建网站来提供新闻和媒体信息。在他们的2010世界杯足球赛网站上，BBC部署了语义技术来获得更加自动的内容发布、更多可以通过更少人力来管理的网页、语义导航和个性化。 182

6.3.2 样例

一个典型的运动员的网页类似于图6-2㊃。类似的网页还有几百个运动员、几十支队伍、所有的分组、所有的比赛，等等。当然所有这些都是高度相连的。

BBC已经开发了一些小本体来捕获足球领域，包括特定领域的关于足球队和锦标赛的概念，以及使用诸如FOAF㊄和GeoNames㊅等知名本体的关于事件和地理位置的宽泛概念。

图6-3展现了一个小的BBC世界杯本体的例子。这些本体被用于推出额外的信息以显示给用户，例如一个运动员参与的国内比赛以及运动员所属的球队。

㊀ http://www.bbc.co.uk/programmes。
㊁ http://www.bbc.co.uk/ontologies/programmes/2009-09-07.shtml。
㊂ http://www.bbc.co.uk/ontologies/wildlife/2010-02-22.shtml。
㊃ http://news.bbc.co.uk/sport/football/world_cup_2010/groups_and_teams/team/england/wayne_rooney。
㊄ http://xmlns.com/foaf/spec/。
㊅ http://www.geonames.org/。

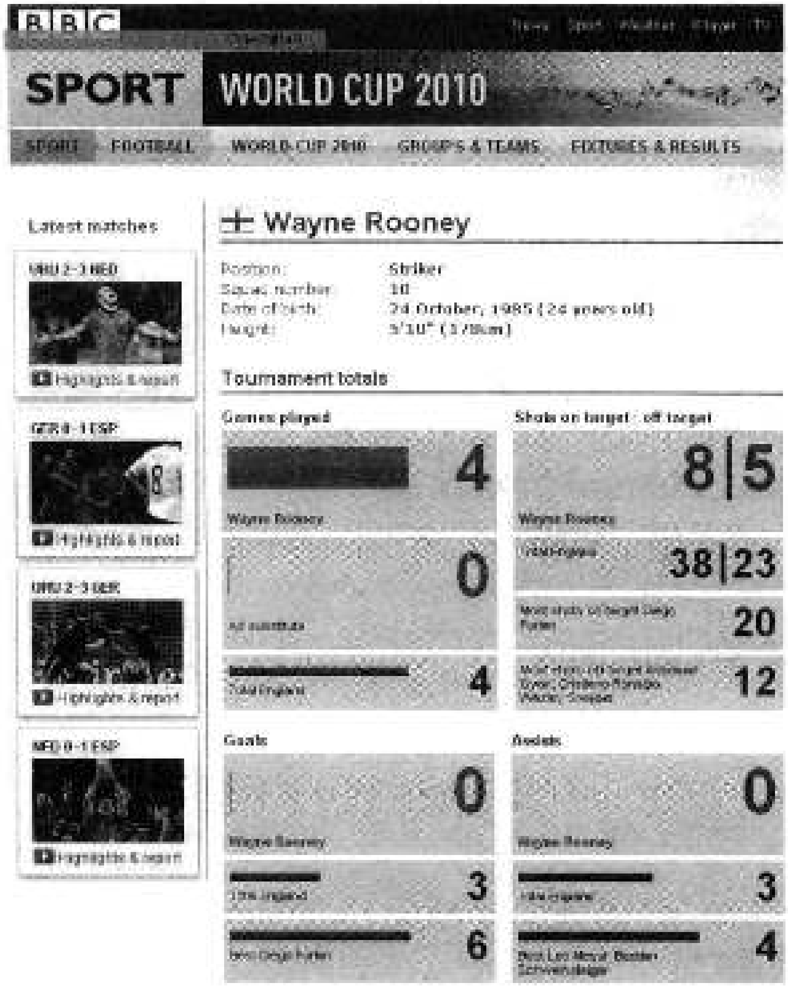


图 6-2 BBC 世界杯网站的典型网页

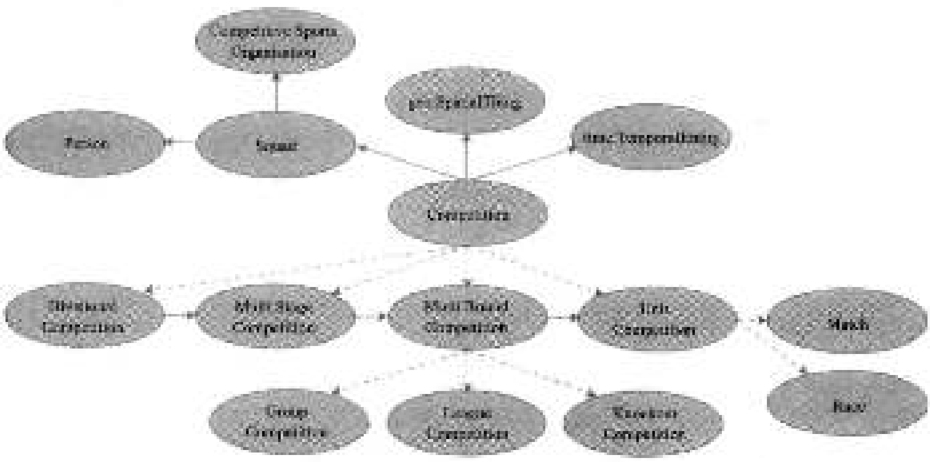


图 6-3 BBC 体育竞技赛本体的一部分

6.3.3 运用

运转BBC世界杯网站的系统采用一个经典的三层体系结构，其中（i）所有信息存储在一个RDF三元组存储库中，（ii）这个三元组存储库被一些本体来组织，支持三元组存储库的查询，以及（iii）一个用户界面层使用基于本体的三元组存储库查询来获得信息并呈现给用户。

有了这个发布模型，BBC声称已经极大地增加了他们的内容重用及赋予新用途的机会，降低了需要维护网站的记者人数，并通过语义驱动的网页布局和多维入口（运动员、比赛、分组等）提升了用户体验。

在其高峰期，系统由一家商业三元组存储库供应商运行，其每分钟处理一千条SPARQL 183
查询（相当于每天一百万条查询），每分钟几百条RDF语句的插入或更新以及使用完备的 184
OWL推理。

BBC正计划将相同的方法运用于他们的奥林匹克2012网站，其将会覆盖来自超过200个国家的超过10 000个运动员，对每个项目和每个事件都有网页，呈现几乎实时的统计数据并提供58 000小时的视频内容。图6-4展现了BBC正在开发的通用体育本体的一小部分。

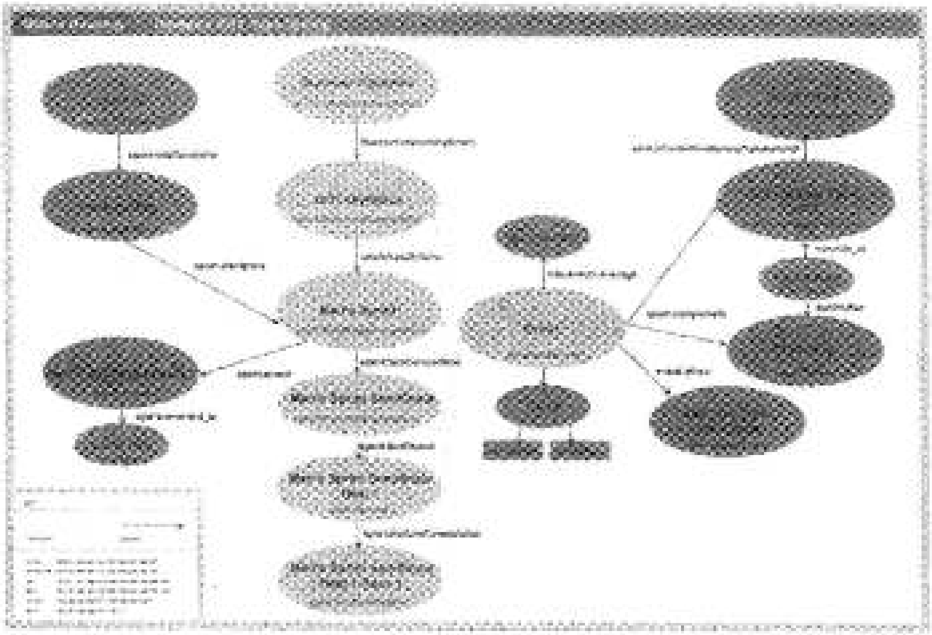


图6-4　BBC奥林匹克本体的一部分

6.4 政府数据

6.4.1 背景

语义网技术早期在世界上的一些大规模运用是奥巴马政府推动的政府透明化的成果。将过去传统上封锁在政府机构内部而不对公众公开的数据源发布出来，这被视为面向更透明政府的一项关键步骤，语义网技术被认为是发布这些数据的一项关键技术。如http://data.gov等

185 网站目前已经发布了来自250个不同的政府机构的250 000个数据集，涉及经济指标、医疗服务统计数据，甚至环境数据。并非所有（或者甚至不是太多的）数据都完全是链接数据的格式，都采用RDF、URI、本体，或都链接到其他数据集。在这种情况下，就可以考虑W3C引入的刻画了从很简单的技术直到发布链接数据的五星（5-star）标准。简言之：

★以任何格式、在任何地方都将数据发布在网上。例如，仅仅将带有表格（甚至是扫描的文档）的PDF文档放在某个网站上就可以在发布数据方面评为1颗星。尽管技术上很平凡，在实际中这已经意味着克服了社会、组织和官僚式的阻碍。

186 ★★使用一种机器可读的格式。这意味着避免诸如PDF文档等格式。发布Excel电子表格是最典型的数据发布为2颗星的例子。

★★★使用一种开放的格式。这意味着避免Excel等专有格式，而是使用诸如.csv文件或Open Office格式等开放的格式。典型的简单格式转换就足以使评级从2颗星变为3颗星。

★★★★为每个数据项分配一个URI。这是万维网语义崭露头角的第一步：为每个数据类型、数据项和数据属性都给予一个URI来指称。这允许其他人链接到这个发布的数据集。

★★★★★最后，链到共享的词汇表。例如，当讨论所有城市的类时使用DBpedia的城市词汇，以及类似的关于实体名称和属性的词汇。这样使用外部词汇表真正实现了跨越不同网站的数据集的互链。

6.4.2 运用

尽管http://data.gov上的大量数据已经（并且仍然）仅仅是2颗星格式，事实上提升这些数据在上述评级系统中的位置并不困难。在8个月时间内，伦斯勒理工学院（Rensselaer Polytechnic Institute）的学生们设法让http://data.gov的64亿条数据项"万维网化"了，更重要的是，他们为这些多样的项目创建了应用，例如绘制邮政服务开支和表现的关系图，绘制荒野火灾和机构预算的关系图（以度量其效率），用税务表格上的信息来覆盖州际迁移，绘制家庭收入和医疗索赔的关系图，比较3个公开预算数据集中的机构预算，以及绘制参观了白宫的人们的社交网络。尽管不可能去建立因果关系，但据报道，公民根据信息自由法案的申诉数量从http://data.gov运行起已经大幅下降。

187 受这一成功的鼓舞，其他政府已经开始效仿这个样例，包括了美国许多州和大城市的政府，以及加拿大、爱尔兰、挪威、澳大利亚和新西兰等国家，而在法国、意大利、丹麦、奥地利、德国和许多其他国家也已经发起了公民倡议。也许最重要的举措已经在英国出现。http://data.gov.uk已经受到了来自政府的强烈支持，其不仅被视为迈向更透明政府的一个步骤，也是一个降低成本的机制。不再去构建和维护昂贵的带有政府信息的网站，政府简单地发布底层的数据源并鼓励第三方（或者是公民或者是商业机构）基于这些发布的数据源开发服务。据报道，仅仅在英国，第三方已经构建了超过200个应用。世界范围内这样的显著应用包括一个自行车路线的安全地图、购房者关于他们新邻居的信息、一个学校查找器、一个托儿所查找器、污染警告、一张区域性支出地图和WhereDidMyTaxGo[⊖]。

⊖ http://www.wheredidmytaxgo.co.uk/.

最后，政府间组织正沿着同样的道路前进，例如世界银行（http://data.worldbank.org）、欧盟的欧洲招标程序网站（http://ted.europa.eu）和欧洲统计办公室 Eurostat（http://ec.europa.eu/eurostat/）。

6.5 《纽约时报》

从 1913 年起，《纽约时报》(New York Times) 已经维护了一个它所出版的所有主题的索引。这已经增长成为一个包含大约 30 000 个“主题标题”的集合，描述了位置、人物、组织和事件。《纽约时报》从 1851 年第一次问世起的每篇文章都已经被标注了这些主题标题，这样的标签被用来提供新闻提醒服务、以自动化编辑过程，并构建“主题网页”来收集关于一个给定主题的所有《纽约时报》的信息。 188

2009 年，《纽约时报》开始将其整个主题标题索引转换为语义网格式。不仅整个标题列表以 RDF 格式发布，其词汇也已经被链到了链接数据万维网上的中心节点，例如 DBpedia、Freebase 和 GeoNames。截至 2010 年，大约 30% 的主题标题已经完成这个过程。

《纽约时报》已经报告说他们的主题标题与链接数据万维网的链接帮助他们提供了地理信息（通过与 GeoNames 的链接），帮助他们将文章与其他那些诸如美国国会图书馆等经常被报纸使用的数据集配准，并使得快速构建他们存档材料的非标准融合成为可能，例如找到关于一所给定大学校友录的所有条目。

6.6 Sig.ma 和 Sindice

尽管诸如 GoodRelations 等应用帮助传统搜索引擎改进了他们的结果而又不曾让终端用户面对底层的 RDF 图，搜索引擎 Sindice[⊖]和它的前端 Sig.ma[⊜]与之不同：Sindice 直接操作构成整个语义网的 RDF 图，而 Sig.ma 向用户呈现以这种方式发现的三元关系。

Sindice 有一个经典的搜索引擎体系结构：一个爬虫、一个非常大的索引的存储库和一个检索界面。但区别于一个经典搜索引擎的是，Sindice 并不检索和索引词和词组，而是检索和索引 RDF 图。在本书写作时，Sindice 索引了 4 亿个 RDF“文档”，得到了 120 亿条 RDF 语句。这些语句被索引并可以通过一个 SPARQL 端点查询。这形成了一种类似数据库的数据之网的存取，可以被开发者和商家使用以增强其产品和服务。有趣的是，Sindice 爬虫并不限于爬取 RDF 图，也爬取其他形式的结构化数据，例如使用 HTML5 微数据格式、Schema.org 词汇表或其他微格式。 189

Sig.ma 是一个用户界面，用来演示数据万维网上的在线查询。始于关键词搜索，Sig.ma 识别出数据万维网上的与一个关键词相关的那些 URL，并找到所有描述这些 URL 的三元组。通过一个交互式用户界面，用户可以辨析哪些三元组源自哪些数据源，选择或删除特定的数据源，通过点击给出的与搜索词相关的那些 URL“在图中漫步”，并打开与它们相关的 URL，等等。

⊖ http://sindice.com。

⊜ http://sig.ma。

通过使用 Sindice 搜索引擎，Sig.ma 向用户提供了一个关于“数据之网”上任意给定主题的在线视图。因为其巧妙的用户界面和 Sindice 非常完备的数据库，Sig.ma 很可能是普通用户体验当前数据万维网的结构和规模及其已经捕获的纯量知识的最佳场所。

6.7 OpenCalais

通讯社汤普森路透社（Thompson Reuters）每天生产几千条新闻，广泛涉及科学、商业、政治和体育。虽然新闻条目刚开始是为了给人阅读的，但是路透社的一个重要发现是他们新闻条目的大量消费者不再是人，而是计算机。计算机正在阅读从路透社流出的新闻，分析它并产生摘要、读取列表、金融投资建议等。

这导致一个自相矛盾的情况，计算机为计算机生产信息，却使用无结构的自然语言作为
190 一种相当不合适的通信媒介。路透社的 OpenCalais 服务用来帮助计算机处理以自然语言文本形式捕获的数据。它分析一段文本，识别出诸如人、位置、公司等命名实体，并用 RDF 标注文本来标识这些命名实体。

作为一个简单的例子，以下文本产生了如图 6-5 所示的实体和关系。

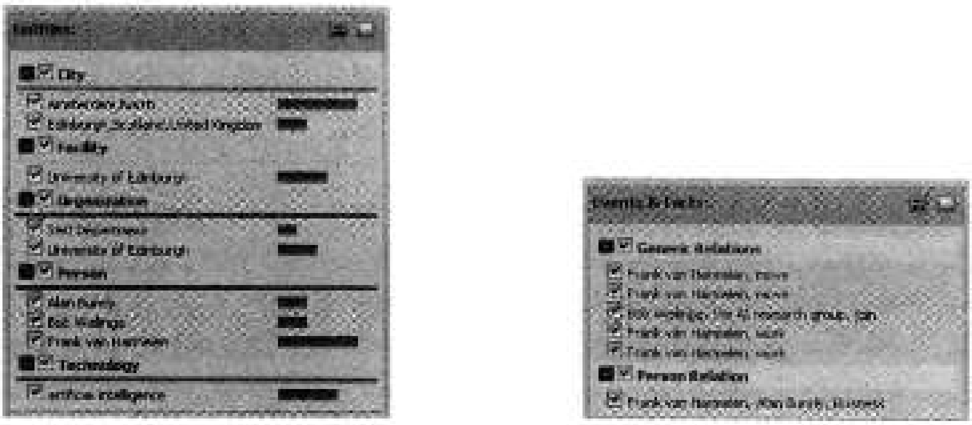


图 6-5 OpenCalais 识别出的对象和关系

After studying mathematics and computer science in Amsterdam, Frank van Harmelen moved to the Department of AI of the University of Edinburgh. While in Edinburgh, he worked with Prof. Alan Bundy on proof planning for inductive theorem proving. After his PhD research, he moved back to Amsterdam where he worked from 1990 to 1995 in the SWI Department under Prof. Bob Wielinga. In 1995 he joined the AI research group at the VrijeUniversiteit Amsterdam.

对于这些对象，路透社使用它自己的 URI 来标识实体，但大量这些私有的 URI 通过 `owl:sameAs` 声明链到相同实体的那些取自 GeoNames、DBpedia、Freebase、互联网电影数据库等的更为广泛使用的 URI。这允许我们进一步检索关于这些来自路透社数据库的实体
191 的信息，例如财务信息、董事会成员的姓名等。因此，通过 OpenCalais，有可能从一个提及词汇的自然语言文本迁移到一个结构化信息项的万维网。据报道，2011 年 Calais 每天能

处理5百万个文档。

6.8 Schema.org

Schema.org是一项搜索引擎提供商（Google、Yahoo、Microsoft、Yandex）鼓励语义标记网页的努力。它的鼓励方式是定义一个在Schema.org可以获得的公共模式或词汇表。这个词汇表涵盖了人们搜索的常见事物，如产品、工作和事件。通过使用一个公共词汇表，搜索引擎能够更好地索引网页，并在搜索结果中展现更丰富的信息。

Schema.org同时支持微数据和RDFa 1.1 Lite形式的标记。因此，可以将Schema.org词汇表与万维网上的其他词汇表混合起来，例如之前讨论过的GoodRelations本体。Schema.org展现了应用如何得以推动机器可理解的标记的运用。

6.9 小结

语义网技术正被各式各样的应用所使用，从支撑网站到让搜索引擎更容易地理解网页的内容。所有这些应用中的一个关键思想是，公共的本体为集成和理解来自多个来源的知识提供了一个基础。我们仅仅强调了一些主要的语义网应用，每天还有无数更多的应用正在被使用。我们鼓励读者去探索和理解其他正在被实现的应用。 192

第 7 章
A Semantic Web Primer, Third Edition

本体工程

7.1 引言

到目前为止，我们主要关注对于语义网至关重要的技术：表示语言、查询语言、转换和推理技术，以及工具。显然，介绍如此多的新工具和技术也引发了方法论问题：这些工具和技术如何才能被最好地应用？在什么场景下应该使用何种语言和工具，以什么顺序使用？质量控制和资源管理的问题如何解决？

本章概述

语义网中的许多问题已经在其他环境中被研究过——例如，在软件工程、面向对象设计和知识工程中。全面介绍所有这些内容已经超出了本书的范围。然而在本章中，我们主要探
193 讨在构建本体过程中的一些方法论问题，特别是手工构建本体（7.2 节）、本体复用（7.3 节）和使用半自动化方法（7.4 节）。7.5 节介绍本体映射。7.6 节介绍如何从关系数据库中发布本体实例。最后，7.7 节解释语义网技术如何能被集成到单个体系结构中来构建应用。

7.2 手工构建本体

对于我们探讨的本体的手工构建，我们主要参考 Noy 和 McGuinness 的《Ontology Development 101: A Guide to Creating Your First Ontology》教程。更多的参考文献请见建议阅读。

我们在本体开发过程中区分以下主要阶段。

1）确定范围。
2）考虑复用。
3）枚举术语。
4）定义分类。
5）定义属性。
6）定义刻面。
7）定义实例。
8）检测异常。

和任何开发过程类似，在实践中这个过程并不是线性的。这些步骤将不得不迭代，在开发过程中的任何时间点上都可能需要回溯到更早的阶段。我们将不讨论复杂的过程管理，而是介绍单个步骤。

7.2.1 确定范围

开发一个领域本体本身不是目的。开发一个本体类似于定义一组被其他程序所使用的数

据和结构。换句话说，一个本体是一个特定领域的模型（model），为特殊目的而构建。作为结
果，对一个具体领域而言不存在正确的本体。一个本体必然是一个特定领域的抽象，因此总
是存在可行的代替品。什么被包含在抽象中应该由将会使用的本体和已经期望的未来扩展所 194
决定。这个阶段将要回答的基本问题是：这个本体将涵盖什么领域？我们出于什么原因使用
这个本体？这个本体应该回答什么样的问题？谁将使用和维护这个本体？

7.2.2 考虑复用

伴随着语义网被广泛部署，许多本体，尤其在公共领域（社交网络、医学、地理），已经可用。因此，我们很少需要自己从头定义本体。几乎总是存在一个可用的第三方本体，至少可以为我们自己的本体提供一个有用的起点（见 7.3 节）。

7.2.3 枚举术语

在实际定义一个本体的过程中，第一步是写下一个所有相关术语的非结构化列表，期望这些术语在本体中出现。通常，名词构成类名的基础，而动词（或动词短语）构成属性名的基础（例如，*is part of*、*has component*）。

传统的知识工程工具，例如梯形和格分析，可以被有效地用于本步骤，用来获得术语集合及这些术语的初始结构。

7.2.4 定义分类

在识别相关术语后，这些术语必须被组织成一个分类层次。对于采用自顶向下还是自底向上的方法哪种更有效 / 更可靠会存在分歧。

当然，很重要的一点是确保层次确实是一个分类（子类）层次。换句话说，如果 A 是 B 195
的一个子类，那么 A 中的每个实例都必须是 B 的实例。只有这样才能确保我们遵循了诸如
`rdfs:subClassOf` 的原语的内置语义。

7.2.5 定义属性

这一步骤通常和之前的步骤交错：在组织类层次的过程中，组织链到类的属性很自然。

记得 `subClassOf` 关系的语义要求当 A 是 B 的一个子类时，每个 B 的实例所拥有的属性声明也必须适用于 A 的实例。由于继承，将属性关联到在它可用的层次中最高层次的类上很有意义。

在将属性关联到类时，立即为这些属性提供定义域和值域声明也很有意义。在一般化和特殊化之间存在一个方法论的折中。一方面，给予属性尽可能通用的定义域和值域很具有吸引力，这使得属性可以被子类（通过继承）使用。另一方面，将定义域和值域定义的越窄越有用，使得我们可以通过发现定义域和值域的违约情况来检测本体中潜在的不一致性和错误概念。

7.2.6 定义刻面

在所有这些步骤之后，该本体只需要 RDF 模式所提供的表达能力，而没有用到 OWL 中任何额外的原语。注意到这点很有趣。这将在本阶段中改变，本阶段将用刻面来丰富之前定

义的属性。

- 基数。为越多越好的属性指出它们是否允许或需要拥有特定数目的不同取值。经常出
196 现的例子是“至少1个值”(即必要属性)和“最多1个值”(即单值属性)。
- 需要的取值。经常，借助某个属性含有特定的值来定义类，而这些所需取值可以在OWL中使用 `owl:hasValue` 指定。有时要求没有这么严格：一个属性被要求有某些取值来源于一个给定的类（而不必是一个特定的值，即 `owl:someValuesFrom`）。
- 关系特征。刻面的最后一类关注属性的关系特征：对称性、传递性、互逆属性和函数型取值。

在本体构建过程中完成这一步之后，可以检查本体内部是否有不一致性。(在这步之前是不可以的，仅仅是因为RDF模式还没有丰富到能够表达不一致性。)经常发生不一致性的例子包括对于传递的、对称的或互逆的属性的定义域和值域定义不兼容。类似地，基数属性经常是不一致性的来源。最后，对于属性取值的要求可能会与定义域和值域限制冲突，这给出了另一种可能的不一致性的来源。

7.2.7 定义实例

当然，我们很少为本体自身而定义本体。相反地，我们使用本体来组织一组实例，而为本体添加这些实例是一个独立的步骤。通常，实例的数目可能超越本体中类的数目几个数量级。本体中类的数目从几百个到数万个，而实例的数目从几百个到数十万个，甚至更多。

因为如此巨大的数目，为本体发布实例通常不是通过手工完成。实例经常从遗留数据源中获取，例如7.6节中将介绍的数据库。另一种经常使用的技术是从文本语料库中自动抽取
197 实例。

7.2.8 检测异常

使用OWL而非RDF模式的一个重要优点在于检测本体自身存在不一致性的可能性，或者检测本体发布的一组实例的不一致性的可能性。经常发生不一致性的例子包括对于传递的、对称的或互逆的属性的定义域和值域定义不兼容。类似地，基数属性经常是不一致性的来源。最后，对于属性取值的要求可能会与定义域和值域限制冲突，这给出了另一种可能的不一致性的来源。

7.3 复用已有本体

如果可能，应该从一个已有的本体开始。现已存在各式各样的本体了。

7.3.1 专家知识的编纂

一些本体由一大组专家耗费许多年时间仔细构建而成。医学领域的一个例子是美国国家癌症中心（National Cancer Institute）的癌症本体[1]。文化领域中的例子包括Art and

[1] http://www.mindswap.org/2003/CancerOntology/。

Architecture Thesaurus（ATT）[21]，它包含12.5万个术语，以及Union List of Artist Names（ULAN）[22]，它包含22万个艺术家的条目。另一个例子是描述文化艺术品的Iconclass词汇表[23]，它包含2.8万个术语。地理领域的一个例子是Getty Thesaurus of Geographic Names（TGN）[24]，它包含了超过1百万个条目。 198

7.3.2 集成的词汇表

有时，尝试将一些独立开发的词汇表融合成单个大的资源。最好的一个例子是Unified Medical Language System[25]，它集成了100个生物医学词汇表和分类。UMLS元词典自身包含75万个概念，以及这些概念之间超过1千万条链接。并不让人感到吃惊的是，这样集成了许多独立开发的词汇表的资源的语义相对较弱，但是不可否认它对于许多应用而言是有用的，至少可以作为一个起点。

7.3.3 上层本体

之前介绍的本体都是高度领域相关的，有一些尝试试图定义非常普遍适用的本体（有时称为上层本体）。两个很好的例子是Cyc[26]，它包含了6千个概念和6万个断言，以及Standard Upperlevel Ontology（SUO）[27]。

7.3.4 主题层次

严格意义上说，这些"本体"不能称为本体：它们是术语的简单集合，被松散地组织成一个特殊化层次。这个层次通常不是一个严格的分类，而是混合了不同的特殊化关系，例如*is-a*、*part-of*或*contained-in*。然而，这些资源经常可以用作一个起点。一个大规模的例子是Open Directory层次[28]，它包含超过40万个层次化组织的类别并且以RDF格式发布。 199

7.3.5 语言学资源

有些资源起初并不是为抽象一个领域而构建的，而是作为该领域的语言学资源。同样地，它们也有助于作为本体开发的起点。这个类别中的一个好的例子是WordNet[29]，它包含超过9万个词语含义的定义。

7.3.6 百科知识

维基百科，依靠大众生成的百科知识，提供了有关一系列主题的丰富信息。DBpedia项

[21] http://www.getty.edu/research/tools/vocabulary/aat。
[22] http://www.getty.edu/research/conducting_research/vocabularies/ulan。
[23] http://www.iconclass.nl/。
[24] http://www.getty.edu/research/conducting_research/vocabularies/tgn。
[25] http://umlsinfo.nlm.nih.gov/。
[26] http://www.opencyc.org/。
[27] http://suo.ieee.org/。
[28] http://dmoz.org/。
[29] http://wordnet.princeton.edu/，RDF格式位于http://semanticweb.cs.vu.nl/lod/wn30/。

目[㊀]从维基百科中抽取知识，并使用 RDF 和 OWL 将其发布为链接数据。维基百科宽泛的知识使得 DBpedia 网站成为构建本体时的首要参考点。Yago[㊁]是利用维基百科的另一个知识库，但是它还包含了源自 WordNet 和诸如 GeoNames[㊂]的地理资源的信息。

7.3.7 本体库

存在一些在线的本体库。例子包括 TONES 本体库[㊃]、BioPortal[㊄]以及由 Protégé 本体编辑器[㊅]提供的。或许目前最好的在线本体库是 Swoogle[㊆]，它登记了超过 1 万个语义网文档并索引它们的类、属性和实例，以及它们之间的联系。Swoogle 也定义了一个语义网文档的排序
200 属性并用它来帮助排序搜索结果。类似地，Sindice[㊇]搜索索引维护了一个几乎包含所有语义网上 RDF 数据的索引。Prefix.cc[㊈]列举了语义网上最常用的命名空间前缀。这些前缀链到它们所代表的对应本体。

在很少一些情况下，现有本体可以不加修改地被复用。通常，现有概念和属性必须被精化（使用 `owl:subClassOf` 和 `owl:subPropertyOf`）。另外，别名也被引入以更好地适合特定领域（例如，使用 `owl:equivalentClass` 和 `owl:equivalentProperty`）。此外，RDF 和 OWL 提供了进一步开发本体的机会，允许私自修改被其他本体定义的类。

引入本体和建立它们之间映射的主要问题还待研究，这被认为是语义网研究中最难的问题之一。

7.4 半自动化的本体获取

将语义网的愿景变为现实存在两个核心挑战。

首先，为构建元数据之网，必须支持语义充实的再工程（reengineering）任务。语义网的成功主要依靠本体和关系型元数据的增多。这要求这些元数据能够被高速且低成本地生产。为实现这个目标，通过本体融合和配准来建立语义互操作性的任务可能需要机器学习技术的支持。

其次，必须提供一种维护和使用机器可存取数据的方式，这是语义网的基础。因此，我们需要能够支持万维网动态性的机制。

201 虽然本体工程工具已经在过去的 10 年成熟了，手工获取本体依然是一个耗时、昂贵、高技术、甚至有时笨重的任务，很容易导致知识获取的瓶颈。

这些问题类似过去的 20 年知识工程师遇到的，他们研究知识获取的方法学或定义知识库

㊀ http://dbpedia.org。

㊁ http://www.mpi-inf.mpg.de/yago-naga/yago/。

㊂ http://www.geonames.org/。

㊃ http://owl.cs.manchester.ac.uk/repository/browser。

㊄ 参见 http://bioportal.bioontology.org/。

㊅ 通过 http://protege.stanford.edu/download/ontologies.html 访问。

㊆ http://swoogle.umbc.edu/。

㊇ http://sindice.com。

㊈ http://prefix.cc。

的工作台。将知识获取和机器学习方法集成已被证明对知识获取有益。

机器学习在知识获取或抽取、知识修订或维护研究领域有很长的历史了，它提供了一大群可能有助于解决问题的技术。下面这些任务可以通过机器学习技术支持。

- 从万维网上已有的数据中抽取本体。
- 从万维网上已有的数据中抽取关系数据和元数据。
- 通过分析概念的外延来融合和映射本体。
- 通过分析实例数据来维护本体。
- 通过观察用户来改进语义网应用。

机器学习提供了一组技术以支持下面的任务。

- 聚类。
- 增量式本体更新。
- 对知识工程师的支持。
- 改进大型的自然语言本体。
- 纯（领域）本体学习。

Omelayenko（参见建议阅读）标识了下面3种类型的本体，它们能够使用机器学习技术来支持。 202

7.4.1 自然语言本体

自然语言本体（Natural Language Ontologies，NLO）包含语言概念之间的词汇关系，它们规模很大并且不需要经常更新。通常它们代表系统的后台知识，被用于扩展用户查询。NLO学习的最新进展看上去非常乐观：不仅存在一个稳定的通用NLO，而且还有自动化的或半自动化的领域相关NLO的构建和充实技术。

7.4.2 领域本体

领域本体捕获一个特定领域的知识，例如药理学或打印机知识。这些本体在一个受限领域中提供了对领域概念的详细描述。通常，它们是手工构建，但是不同的学习技术可以辅助（尤其是缺乏经验的）知识工程师。学习领域本体的进展没有NLO快。领域本体的获取依然需要人类知识工程师的指导，而自动化的学习技术在知识获取过程中只起到很小的作用。它们只能在领域文本中发现统计意义上可验证的依赖关系，并将它们建议给知识工程师。

7.4.3 本体实例

本体实例可以被自动化地生成并经常更新（例如，黄页中某个公司的信息会常更新），而本体保持不变。本体实例的学习任务非常适合机器学习的框架，并且已经有一些机器学习算法的成功应用。但是这些应用要不就是严格依赖于领域本体，要不就是发布标记而不关联到任何领域理论。对于给定一个领域本体作为输入，一种从文本中抽取本体实例的通用技术还没有被开发出来。 203

除了能够被支持的本体的不同类型，本体学习中也存在不同的用法。在下面列表中，前

3个任务（同样源自Omelayenko的研究）涉及知识工程中的本体获取任务，而后3个涉及本体维护任务。

- 由知识工程师开始创建本体。在这个任务中，机器学习通过建议领域中最重要的关系或者验证构建的知识库来辅助知识工程师。
- 从万维网文档中抽取本体模式。在这个任务中，机器学习系统在知识工程师可能的帮助下，以数据和元知识（例如元本体）作为输入，并生成即刻使用的本体作为输出。
- 本体实例抽取为本体模式填充数据，并抽取表示在万维网文档中的本体实例。这个任务和信息抽取及页面标注类似，可以应用这些领域开发的技术。
- 本体集成和浏览处理重构与浏览大型的、可能机器可学习的知识库。例如，这个任务可能将机器学习器的命题层次知识库改变为一阶逻辑知识库。
- 本体维护任务更新本体的某些部分，这些部分被设计为能更新的（例如格式化那些只能在页面布局中跟踪修改的标签）。
- 本体扩充（或本体调谐）将微小的关系的自动修改加入已有本体。这不改变本体中的
204 主要概念和结构，但是会使得本体更加精确。

机器学习领域已有各种各样的技术、算法和工具。但是，本体表示的一个重要需求是本体必须符号化、人类可读、可理解。这使得我们只涉及产生泛化的符号学习算法，而跳过诸如神经网络、遗传算法之类的其他方法。下面是一些潜在适用的算法：

- 命题规则学习算法学习关联规则或其他形式的属性－值规则。
- 贝叶斯学习的最佳代表是朴素贝叶斯分类器。它基于贝叶斯定理，基于训练实例间的属性是条件独立的这一假设，生成概率性的属性－值规则。
- 一阶逻辑规则学习归纳出包含变量的规则，称为一阶Horn子句。
- 聚类算法使用基于实例的属性值定义的相似度或距离度量将实例聚类在一起。

总之，我们可以说，虽然可以被部署到语义网工程中的潜在的机器学习技术很多，但是这个领域距离完全掌握还为时尚早。

7.5 本体映射

当复用而非从头开发变成本体部署的范式，本体集成将成为一个逐渐棘手的任务。很少
205 有可能单个本体可以完全满足一个特定领域的需求，更多情况是多个本体需要被组合。这导致了本体集成问题（也称为本体配准或本体映射）。由于这个严峻的事实，这个问题近年来受到研究领域的广泛关注。

当前本体映射的方法包括各个层次的不同方法，来源于许多不同领域。我们将它们分为语言学的、统计的、结构的和逻辑的方法。

7.5.1 语言学方法

最基本的方法试图利用附着于源本体和目标本体中概念上的语言学标签来发现潜在的匹配。简单的方法可以是基本的提词干技术或计算汉明距离，或者也可以使用特殊的领域知识。

后者的一个例子是 *Diabetes Melitus type I* 和 *Diabetes Melitus type II* 不是一个可以忽略的小差异，但是它却具有很小的汉明距离。

7.5.2 统计方法

一些方法不使用概念的语言学标签，而是使用实例数据来确定概念间的对应。如果源本体和目标本体中的概念的实例存在一个显著的统计对应关系，则有理由相信这些概念是（通过一个包含关系或者甚至可能是等价关系）强关联的。显然，这些方法依赖于存在一个足够大的被源本体和目标本体分类的实例库。 206

7.5.3 结构方法

因为本体含有内部结构，利用源本体和目标本体的图结构并试图确定这些结构间的相似度是可行的，这种方法常与其他方法一起使用。如果一个源概念和一个目标概念有相似的语言学标签，则它们图上邻居节点的不相似性可以被用于检测同形异义问题，而纯语言学方法可能会错误地宣称发现了一个潜在的映射。

7.5.4 逻辑方法

逻辑方法或许是最针对本体映射的方法。毕竟，由 R. Studer 定义的，本体是"一个共享概念体系的形式化规约"，利用源本体和目标本体的形式化表达是可行的。这种方法的一个严重缺陷是许多实际使用的本体的语义是轻量级的，因此没有携带很多逻辑公式。

7.5.5 映射实现

已经有一些本体映射框架，例如 R2R 框架[1]和 LIMES[2]。sameas.org 服务从不同数据源采集和发布 owl:sameAs 映射。尽管这些项目已经在创建映射方面大幅前进，但这还是一个非常有挑战的领域。研究社区已经在过去 9 年中举办本体配准评测活动（Ontology Alignment Evaluation Initiative[3]），以鼓励构建更加精确和全面的映射。

7.6 发布关系数据库

大多数当今的网站不是存储在万维网服务器上的一系列静态页面，而是从存储在关系数据库（relational database）中的数据动态生成的。例如，一个房地产网站维护各种房屋和公寓的数据库，包括价格、位置和便利设施。这个数据库将被用来发布网页。由于关系数据库中存在大多的数据，它可以提供一个方便的实例数据源。

下面，我们将概述发布关系数据库为本体的过程。

7.6.1 映射术语

首先，我们回顾数据库的术语，以及这些术语如何映射到 RDFS/OWL 术语。下面是一个

[1] http://www4.wiwiss.fu-berlin.de/bizer/r2r/。
[2] http://aksw.org/Projects/LIMES?v=z1l。
[3] 参见 http://oaei.ontologymatching.org/。

房产数据库中的表（table）的例子。一个表也称为一个关系（relation）。表由一系列包含表头的列组成，例如 HomeId、City 和 Price。这些列称为属性（attribute）。表的每行称为一个元组（tuple）。

Homes

HomeId	City	Price (Euros)
1	Amsterdam	100 000
2	Utrecht	200 000

基于这些术语，可以用一个简单的方法来将关系数据库模式映射到 RDFS 或 OWL。数据库中的每个表被认为是一个类。每个属性（attribute）可以被认为是一个属性，而每个元组可以被认为是一个实例。这样，在我们的例子中，我们有 Home 类的两个实例。每个实例有一个 city 属性和一个 price 属性，以及它们对应的值（例如，对于 HomeId 为 1 的实例，它的
208 city 属性的值是 Amsterdam）。

当进行映射时，还必须为每个实体创建 URI。这个常通过在属性（attribute）或表名前面添加一个命名空间。类似地，经常可以使用主键（primary key）作为每个实例的 URI。要重点注意的是，关系数据库和 RDF 之间的主要区别在于 RDF 使用 URI 来标识实体，意味着任何事物都有一个全球唯一的标志符。而关系数据库只在给定数据库的局部范围内拥有标志符。

7.6.2 转换工具

由于关系数据库映射到本体存在系统化的机制，自动化转换过程是可能的。存在一些已有的工具，由 W3C 关系数据库到 RDF 孵化组列出。你可以在建议阅读里发现这个报告的链接。这些工具中的大多数通过分析关系数据库的结构，然后生成几乎完整的 RDF。接下来，用户被要求修改配置文件来指定更合适的 URI 以及到已有本体的链接。例如，在上面的例子中，对 Amsterdam 可以使用 DBpedia 的 URL 来替代自动生成的 URI。

转换工具经常提供两种功能。一种直接将关系数据库发布为 SPARQL 端点。第二种是将数据库的大部分转换为 RDF，使得之后可以被载入三元组存储中。后一种功能常被用于将实例数据与本体集成并需要推理时。一个推荐的起始工具是 D2R Server[⊖]，因为它以一种简单
209 的安装包的形式提供了上述两种功能。

7.7 语义网应用体系结构

构建语义网不仅涉及使用本书中介绍的新语言，还涉及一个相对不同的工程风格和一个相对不同的应用集成方法。为说明这点，我们概述一组语义网相关的工具应该如何通过语义网标准被集成到单个轻量级应用体系结构中，以实现独立的工程工具间的互操作性（参见图 7-1）。

⊖ http://www4.wiwiss.fu-berlin.de/bizer/d2r-server/。

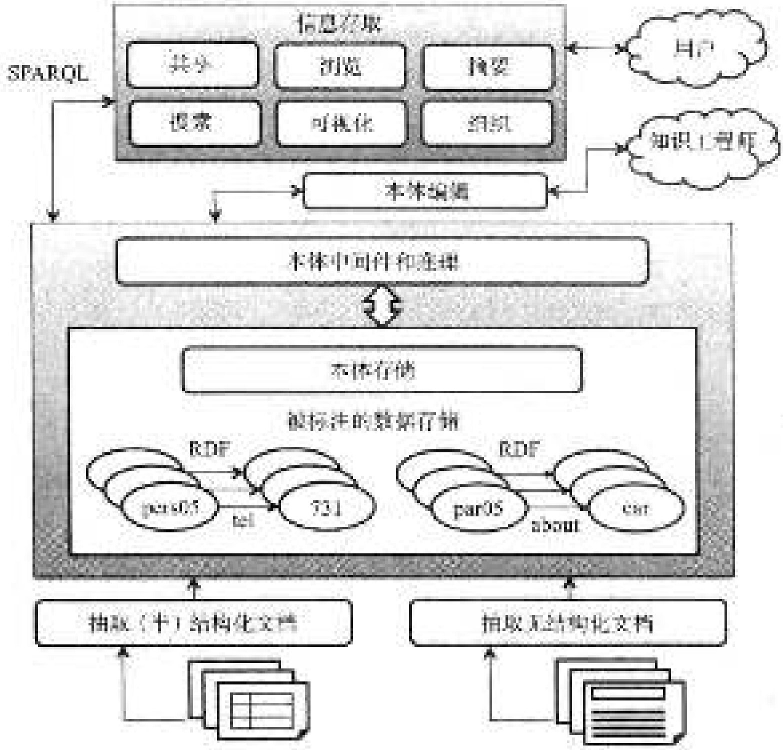


图 7-1　语义网知识管理体系结构　210

7.7.1　知识获取

图 7-1 下部，我们发现了使用表层分析技术从文档中获取内容的工具。这些文档可以是无结构的自然语言文档，也可以是结构化或半结构化的文档（例如，数据库、HTML 表格和电子表单）。

对于无结构的文档，这些工具通常联合使用统计技术和浅自然语言技术来从文档中抽取核心概念。

对于更结构化的文档，可以使用之前介绍的数据库转换工作。归纳和模式识别技术被用于从更加弱结构化的文档中抽取内容。

7.7.2　知识存储

分析工具的输出是一组概念，被组织成一个浅的概念层次，最多有一些跨分类的联系，使用 RDF 和 RDF 模式就足以表达这些被抽取的信息。这些信息还包括实例数据。

除了简单地存储由抽取工具产生的知识外，显然存储库也必须提供检索这些知识的能力，偏向于使用一个结构化查询语言，例如 SPARQL。任何合理的 RDF 模式库也将支持 RDF 模型论，包括基于定义域和值域定义的类成员关系的演绎，以及 subClassOf 联系的传递闭包的生成等。

注意，存储库将存储本体（类层次、属性定义）和本体的实例（属于类的具体个体，拥有

211 特定属性的个体对）。

7.7.3 知识维护

除了基本的存储和检索功能，一个实用的语义网存储库还提供管理和维护本体的功能：变更管理、访问权限和所有权、事务管理。

除了从无结构或半结构化数据中自动生成的轻量级本体，还必须为人类工程师提供对知识密集型本体的支持。精心设计的编辑环境可以被用来从库中检索本体，允许知识工程师操纵它们，并将它们放回到存储库中。

7.7.4 知识使用

存储库中的本体和数据将被应用程序使用来服务终端用户。我们已经介绍了一些这样的应用。此外，外部应用可能通过以下的一种或多种方法从暴露的数据中存取知识：SPARQL端点、链接数据或RDFa。

7.7.5 应用体系结构

在On-To-Knowledge项目[1]中，图7-1的体系结构通过组件之间轻量级的关联来实现。语法的互操作性被实现，因为所有组件使用RDF来通信。语义互操作性被实现，因为所有语义使用RDF模式表达。物理互操作性被实现，因为组件之间所有的通信都建立在简单的HTTP连接上，除了本体编辑器之外的所有组件都被实现为远程服务。当从Amsterdam操作On-To-
212 Knowledge系统时，运行在Norway（挪威）的本体抽取工具被给予一个位于伦敦的文档URL来分析，生成的RDF和RDF模式被上传到运行在Amersfoort（荷兰）的存储服务器上。这些数据被上传到一个本地安装的本体编辑器中，编辑之后，下载回Amersfoort的服务器。这些数据随后被用于驱动一个瑞典的基于本体的网站生成器，以及一个位于英国的搜索引擎，它们都在Amsterdam的浏览器屏幕上展现结果。

总之，所有这些工具都是远程运行的，都是独立于工程实现的，而且仅依赖HTTP和RDF来获得高度的互操作性。

7.7.6 框架

存在许多框架实现了上述体系结构。例如，Drupal内容管理系统[2]为一个广泛使用的内容管理系统添加语义支持。存在两个被广泛使用且支持良好的开源框架：Jena[3]和Sesame[4]。
213 最后，诸如Kasabi和Dydra之类的公司提供了构建语义网应用的上框架。

建议阅读

一些被用作本章基础的重要论文包括以下内容。

---

[1] http://www.ontoknowledge.org/。
[2] http://semantic-drupal.com/。
[3] http://jena.sourceforge.net/。
[4] http://openrdf.org/。

- N. Noy and D. McGuinness. Ontology Development 101: A Guide to CreatingYour First Ontology.
  www.ksl.stanford.edu/people/dlm/papers/ontology101/ontology101-noy-mcguinness.html.
- B. Omelayenko. Learning of Ontologies for the Web: The Analysis of Existing Approaches. In *Proceedings of the International Workshop on Web Dynamics, 8th International Conference on Database Theory* (ICDT 2001). 2001.
  www.cs.vu.nl/~borys/papers/WebDyn01.pdf.
- Satya S. Sahoo, WolfgangHalb, Sebastian Hellmann, Kingsley Idehen, Ted Thibodeau Jr, Sören Auer, Juan Sequeda, and Ahmed Ezzat. A Survey of Current Approaches for Mapping of Relational Databases to RDF (PDF). W3CRDB2RDF Incubator Group 2009-01-31.
  http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport.pdf.
- M. Uschold and M. Gruninger. Ontologies: Principles, Methods and Applications. *Knowledge Engineering Review*, Volume 11, 2 (1996): 93–155.

两本被经常引用的书是：

- J. Davies, D. Fensel, and F. van Harmelen. *Towards the Semantic Web: Ontology-Driven Knowledge Management*. New York: Wiley, 2003.
- A. Maedche. *Ontology Learning for the Semantic Web*. New York: Springer 2002. 214

练习和项目

这个中等难度的项目可以由2或3个人在2～3个星期内完成。所有需要的软件都可以免费获得。我们提供一些已经成功使用过的软件的链接，但是考虑到这个领域的发展非常活跃，可用的软件很可能已经快速更新了。另外，如果某个软件没有被提到，这不意味着我们不同意使用它。

作业由三部分组成。

1）在第一部分中，你将创建一个描述领域的本体，包含你自己的应用所需的信息。你将使用在该本体中定义的术语来描述具体的数据。在这步中，你将应用这节开始部分介绍的本体构建方法学，并使用OWL作为你的本体的表示语言（参见第4章）。

2）在第二部分中，你将使用你的本体来为你的数据创建不同的视图，并且你将查询本体和数据来抽取每个视图所需的信息。在这部分，你将应用RDF存储和查询软件（参见第2章）。

3）在第三部分中，你将使用基于万维网的技术来创建被抽取数据的不同的图形化表示。

第一部分　创建一个本体

第一步，确定一个应用领域，建议是你拥有充足知识的或者你很容易从某个专家那里获取这些知识的领域。

在项目的描述中，我们使用广播和电视节目领域，包括节目、播出计划、频道、风格、名人。当然，你可以用你选择的领域来替换这个领域。在我们自己的课程中，我们每年使用 215

不同的领域，从大学排名到电影，到财富 500 强。

第二步，你将使用 OWL 建立本体来描述该领域（例如，你的教员）。这个本体不必覆盖完整的领域，但是应该至少包含几十个类。特别注意本体的质量（宽度、深度），并有针对性地使用越多的 OWL 的表达能力越好。在这步中存在许多可用的工具。当前最好的工具或许是 Protégé[1]，但是我们觉得 TopBraid Composer[2]也不错。

如果你有野心，甚至可以使用从文本中抽取本体的工具，或试验允许你导入诸如 Excel 表单、tab 界定的文件之类的半结构化数据源的工具（如 Excel2RDF、ConvertToRDF、Any23 或 XLWrap）来开始你的本体开发。当然，你可能会选择从本领域的某些已有的本体开始。

同时建议使用一个推理引擎来验证你的本体和检查不一致性。如果你使用 Protégé，你可能希望利用一些已有的编辑器插件，例如对你的本体或使用 Pellet 或 HermiT 推理的多种可视化。

第三步，将你的本体和具体的实例及属性一起发布。依赖于编辑工具的选择，这步可以通过使用相同的工具完成（Protégé），或者给定 RDF 中实例的简单语法结构，你甚至可能决定手工写下这些实例或者编写一些简单的脚本来从可用的数据源中抽取实例信息。例如，你可以将一个关系数据库及给定数据转换为 RDF。或者你可能希望从许多包含广播和电视播出计划、节目、风格和名人的网站上搜刮一些信息。BBC 甚至提供了一个方便的应用编程接口

216 来直接查询它们的播出计划[3]。你可能希望使用 W3C 提供的语法验证服务[4]——这个服务不仅验证你的文件的语法正确性，也提供了对已有三元组的可视化。同时，在这步中你或许能够试验一些允许从半结构数据源中导入数据的工具。

在这步结束时，你应该能够提供下面的东西。

- 完整的 OWL 本体。
- 本体的实例，使用 RDF 描述。
- 一个描述本体范围以及在建模过程中你做出的主要设计决定的报告。

第二部分　使用 SPARQL 查询建立概况

接下来，你将使用查询工具来抽取你的本体和数据中的相关部分。为实现这点，你需要在某个库中存储你的本体的方法来支持查询和推理工具。你可以使用 Sesame RDF 存储和查询工具[5]，它绑定了一个 OWLIM 推理机。我们也发现 Joseki Sparql Server 是一个好的起始点，因为它提供了一个内建的万维网服务器。

第一步是上载你的本体（以 RDF/XML 或 Turtle 的格式）并将实例关联到库中。这可能需要一些安装步骤。

接下来，使用 SPARQL 查询语言来定义不同的用户概况，并使用查询为每个概况抽取相关数据。

---

[1] http://protege.stanford.edu/。

[2] 参见 http://topquadrant.com/products/TB_Composer.html。

[3] 参见 http://www.bbc.co.uk/programmes/developers。

[4] http://www.w3.org/RDF/Validator/。

[5] http://www.openrdf.org/。

以建模电视节目为例，你可能选择为有特别偏好（体育、时事）的人们或特殊年龄组的观众（例如未成年人）定义观看指南，从多个（甚至跨国的）电视台收集数据，为宽带或移动连接访问生成表现形式，等等。 217

定义一个概况的查询的输出通常使用 XML 或 JSON（JavaScript Object Notation）的形式。

第三部分 表示基于概况的信息

使用第二部分的查询输出，为不同概况生成一个人类可读的表示形式。根据你最喜欢的编程语言，有一些方便的库用来查询 SPARQL 端点：Python 拥有 SPARQLWrapper[㉛]、PHP 拥有 ARC2 库[㉜]，而 Java 用户会喜欢 ARQ[㉝]，它是流行的 Jena 库的一部分。甚至有一些从 Javascript 中可视化 SPARQL 结果的库，例如 sgvizler[㉞]。

这部分的挑战在于从第一部分和第二部分生成和选择的数据中，定义数据的可浏览的、高度相互链接的表示形式。

领域的其他选择

除了使用描述广播节目领域的半结构化数据集，还可能建模一个大学的教职工领域，包括教师、课程和院系。在这个例子中，你可以使用在线数据源，例如从教职工电话册中的信息、经历描述、教学计划等中获得本体和实例数据。这个领域中的示例概况可能有不同年份的学生概况、外国学生的概况、学生和教师的概况，等等。

总结

在完成项目的所有部分之后，你已经能有效地实现图 7-1 中展现的体系结构的大部分。你已经使用了本书介绍的大部分语言（RDF、RDF 模式、SPARQL、OWL2），并且已经构建了一个真正的语义网应用：用一个本体来建模部分世界，用查询来定义特定用户的视图，以及用万维网技术来定义特定用户视图的可浏览的表示形式。 218/219

㉛ http://sparql-wrapper.sourceforge.net/。

㉜ http://incubator.apache.org/jena/documentation/query/index.html。

㉝ http://incubator.apache.org/jena/。

㉞ http://code.google.com/p/sgvizler/。

第8章
A Semantic Web Primer, Third Edition

总　　结

语义网的愿景已经成为现实。正如在第6章中讨论的，越来越多的公司、政府和用户正在采用本书中介绍的标准、技术和方法。我们看到了如何将万维网的概念分解为一个灵活的数据模型，RDF，使得应用之间可以交换和复用信息。基于这个数据模型，我们看到了如何逐步添加更丰富的语义，允许产生逐步增强的推理（第2章）。OWL2（第4章）允许构建丰富的知识表示，而规则语言（第5章）允许系统性地编入特定应用的推理。所有信息通过一个集成于万维网的查询语言SPARQL来使用（第3章）。最后，我们看到了本体工程方法如何用于创建更智能的、更先进的应用（第6章）。

贯穿本书，我们看到了这些技术是如何植根于计算机学科宽广的知识当中，包括人工智
221 能、数据库、分布式系统等，并且我们还指出了进一步学习所涉及内容的地方。我们鼓励读者更深入地学习这些领域。

8.1 原理

学习语义网不仅是学习其中的应用或技术，它还提供了一组有助于设计系统的通用原理，而这些系统甚至不使用语义网的方法。我们在这里重新回顾一下。

8.1.1 提供一个从轻量级到重量级技术的路线

开发者和用户需要简单的技术入口，同时也需要在他们的需求增长时能够增加复杂性。过渡路线这一概念已被成功用于语义网。例如，可以从RDF中的一个简单数据模型开始，逐渐过渡到更加丰富的、更强的OWL语言。在OWL中甚至也有多种选择，从简单的支持快速推理的规则版本到OWL2 Full的复杂描述。

8.1.2 标准节约时间

万维网和类似的语义网受益于标准。这些标准意味着信息的消费者不必担心如何适应每个新的信息生产者。同样地，生产者知道他们正在给消费者提供他们想要的。没有必要重新发明语法或模型。在语义网中，标准使得信息的复用成为可能。如果我在我的应用中想要关于Amsterdam的信息，我不再需要自己去收集，而是可以将我的应用建立在可用的维基百科（以它的RDF表示形式，DBpedia）之上。这种复用能力节约了应用开发和集成的时间。语义网社区一直学习从标准中获益。不仅是标准化格式和技术，而且还标准化本体和其他内容，
222 这些都已经逐渐开展。

8.1.3 链接是关键

和万维网类似，语义网的能力在于它的链接。链接使得知识可以分布在不同的组织、人和系统中。这让知识的最想获得者能得到它，也依然让他人能访问它、发现它。足够轻量级的链接使其很容易被创建，同时又足够强大到允许知识的集成。我们在链接数据云中看到了数以百计的特定数据集可以依赖于其他特定数据集，它们（以一种简单的方式）提供了有关某个主题或概念的更多、更细节的知识。可以这样说，在任何一个大的信息系统中，链接的能力可以预测其成功或失败。

8.1.4 一点语义可以影响深远

语义网的一位先驱 Jim Hendler 创造了术语"A little semantics goes a long way"。这句话强调自动理解术语含义的能力的重要性。共享数据是不够的——还需要共享含义。简单地知道 bank 指的是一张长椅而非一个金融机构对构建精准的系统作用巨大。在本书中，我们已经介绍了为数据添加语义及在应用中使用它们的技术。是否在系统中采用这些特别的技术是一回事，但关键原则是认识到语义可以是一种差异化要求。

上述原理已经帮助语义网取得了成功，但更重要的是，它们适用于其他任何大规模软件系统或技术集的设计。 223

8.2 展望

语义网依然在发展和变化。现有技术正在被改良而新技术正在研发之中。概括地说，存在以下正被活跃研究的主要研究方向。

- 语境中的推理。这使得推理具有不仅依赖于给定的知识，还依赖于应用所处的环境和知识来源（它的出处）的能力。例如，当你刚开始寻找公寓时，你可能会相信某个有关公寓的数据源，但是在做最终决定时可能就不再相信了。
- 大规模推理。由于语义网上的数据量不断增长，使得推理这些数据变得非常困难。诸如大型知识对撞机（Large Knowledge Collidor，http://www.larkc.eu）的项目已经展示了在数以亿计的三元组上开展简单的规则式推理的能力。但是，伴随语义的复杂性增长以及数据和知识量持续爆炸，需要有更好、更快的推理机。
- 分布式查询。当前在语义网上使用分布式知识源的标准做法是从各种数据源中获取并集中化知识。这么做的缘由是效率。但是，至于如何使用仍然是一个开放问题，并且对于特殊查询如何使用这些分布式信息源而不必集中化它们也是一个问题。
- 流知识。语义网技术主要处理静态的或缓慢变化的信息。这一现状正在快速改变，传感器、微博和其他来源的信息正加速产生。开放问题包括如何从这些流数据源中抽取、推理和使用语义。 224

这些只是一些当前语义网相关的热点研究方向。语义网领域本身和它的技术正在快速发展。本书只是介绍了核心基础。给语义网下结论还言之尚早。想要了解目前的进展，请访问在线补充材料：http://www.semanticwebprimer.org。 225

XML 基础

A.1 XML 语言

一个 XML 文档（XML document）由一段序言、一些元素和一段可选的尾声组成（不在这里讨论）。

A.1.1 序言

序言由一个 XML 声明和一个可选的外部结构文档的引用组成。这是一个 XML 声明（XML declaration）的例子：

```
<?xml version="1.0" encoding="UTF-16"?>
```

它指明了当前文档是一个 XML 文档，并定义了其版本和用于特定系统的字符编码（例如 UTF-8、UTF-16 和 ISO 8859-1）。字符编码不是强制性的，但是指明它被认为是好的做法。有时我们也指明文档是否是自包含的——即它是否不涉及外部结构文档：

227

```
<?xml version="1.0" encoding="UTF-16" standalone="no"?>
```

外部结构文档的一个引用看起来像这样：

```
<!DOCTYPE book SYSTEM "book.dtd">
```

这里，结构信息可以在一个称为 book.dtd 的本地文件中找到。引用也可以是一个 URL。如果仅仅使用一个本地识别的名称或者一个 URL，那么就使用标记 SYSTEM。然而，如果希望同时给出一个局部名称和一个 URL，那么则应该使用标记 PUBLIC。

A.1.2 元素

XML 元素表示这个 XML 文档谈论的“事物”，例如图书、作者和出版社。它们构成了 XML 文档的主要概念。一个元素由一个开始标签（opening tag）、它的内容（content）和一个结束标签（closing tag）组成。例如，

```
<lecturer>David Billington</lecturer>
```

标签名称几乎可以自由选择，只有非常少的一些限制。最重要的限制是首字符必须是字母、下划线或冒号，并且名称不能以字符串“xml”的任何大小写形式开头（例如“Xml”和“xML”）。

内容可以是文本、其他元素或空缺。例如，

```
<lecturer>
  <name>David Billington</name>
  <phone>+61-7-3875 507</phone>
</lecturer>
```

如果没有内容，那么元素叫做空（empty）。一个形如以下的空元素

```
<lecturer></lecturer>
```

可以被简写为

```
<lecturer/>
```

228

A.1.3 属性

一个空元素未必没有意义，因为它可能有一些属性（attribute）。一个属性是一个元素的开始标签内的一个名–值对：

```
<lecturer name="David Billington" phone="+61-7-3875 507"/>
```

这是一个非空元素的属性的例子：

```
<order orderNo="23456" customer="John Smith"
          date="October 15, 2002">
  <item itemNo="a528" quantity="1"/>
  <item itemNo="c817" quantity="3"/>
</order>
```

同样的信息可以写成如下形式，用嵌套元素来替换属性：

```
<order>
  <orderNo>23456</orderNo>
  <customer>John Smith</customer>
  <date>October 15, 2002</date>
  <item>
    <itemNo>a528</itemNo>
    <quantity>1</quantity>
  </item>
  <item>
    <itemNo>c817</itemNo>
    <quantity>3</quantity>
  </item>
</order>
```

229

何时使用元素和何时使用属性经常是看个人喜好，但是要注意属性不能被嵌套。

A.1.4 注释

一条注释是一段会被解析器忽略的文本。它具有以下形式：

```
<!-- This is a comment -->
```

A.1.5 处理指令

处理指令（Processing Instruction，PI）提供了一种向一个应用程序传递关于如何处理元素的信息的机制。一般形式是：

```
<?target instruction?>
```

例如，

```
<?stylesheet type="text/css" href="mystyle.css"?>
```

PI 在一个声明式的环境中提供了过程式的可能性。

A.1.6 良构的 XML 文档

一个语法上正确的 XML 文档称为是良构的。以下是一些语法规则。

- 文档中只有一个最外层元素（称为根元素（root element））。
- 每个元素包含一个开始和一个对应的结束标签。
- 标签不能交叠，例如

```
<author><name>Lee Hong</author></name>.
```

- 一个元素内的属性有唯一的名称。

230 - 元素和标签名称必须是合法的。

A.1.7 XML 文档的树模型

可以将良构的 XML 文档表示为树，因此，树为 XML 提供了一个形式化的数据模型。这种表示往往是有益的。作为一个例子，考虑以下文档：

```
<?xml version="1.0" encoding="UTF-16"?>
<!DOCTYPE email SYSTEM "email.dtd">
<email>
  <head>
    <from   name="Michael Maher"
            address="michaelmaher@cs.gu.edu.au"/>
    <to     name="Grigoris Antoniou"
            address="grigoris@cs.unibremen.de"/>
    <subject>Where is your draft?</subject>
  </head>
  <body>
```

```
    Grigoris, where is the draft of the paper
    you promised me last week?
    </body>
</email>
```

图 A-1 展现了这个 XML 文档的树表示。它是一棵有序的、有标记的树。

- 恰有一个根。
- 没有圈。
- 每个节点，除了根以外，恰有一个父节点。
- 每个节点有一个标记。
- 元素的顺序是重要的。 231

然而，尽管元素的顺序是重要的，属性的顺序却并非如此。因此，以下两个元素是等价的：

```
<person lastname="Woo" firstname="Jason"/>
<person firstname="Jason" lastname="Woo"/>
```

这方面没有在树中恰当地表示出来。一般来说，我们要求一个更加细致的树的概念，例如，我们也应该区分不同类型的节点（元素节点，属性节点等）。然而，这里我们使用图作为示例，因此我们不再深入到细节中。

图 A-1 也体现出根（root）（表示这个 XML 文档）有别于根元素（root element）——即例子中的 email 元素。这一区别在讨论寻址和查询 XML 文档时有所体现。

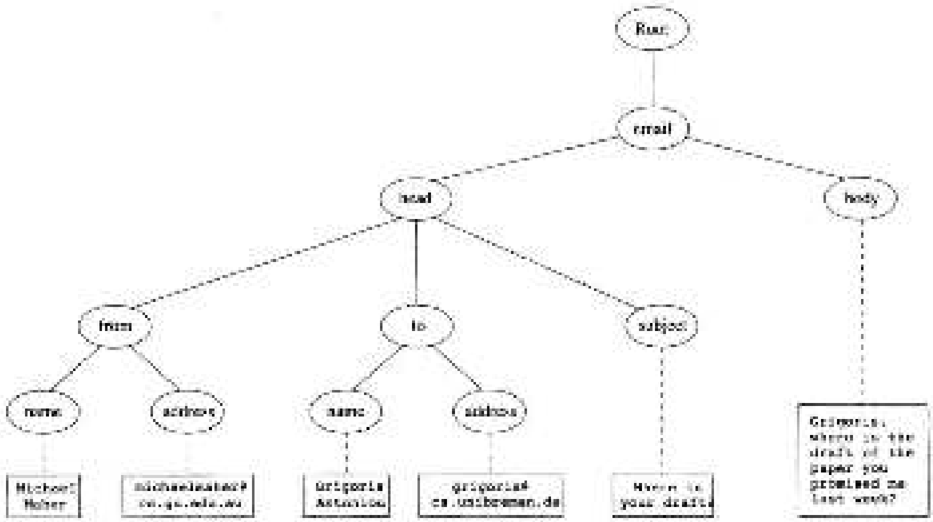


图 A-1 一个 XML 文档的树表示 232

A.2 结构化

一个遵循特定语法规则的 XML 文档是良构的。然而，那些规则并没有特别规定文档的结构。现在，想象两个试图通信并希望使用相同词汇表的应用。为此，就需要定义所有可能用到的元素和属性名称。进一步地，结构也应当被定义：一个属性可以取哪些值，哪些元素可能或者必须出现在其他元素中，等等。

有了这样的结构化信息，我们就有了验证文档的更大可能性。我们说一个 XML 文档是有效的（valid），如果它是良构的，使用了结构化信息并且遵循这一信息。

有两种定义 XML 文档结构的方法：DTD——古老且更受限的方法；XML 模式——其主要为数据类型的定义提供了扩展的可能性。

A.2.1 DTD

1. 外部和内部 DTD

一个 DTD 的组件可以定义在一个单独的文件中（外部 DTD（external DTD）），或者定义在 XML 文档本身的内部（内部 DTD（internal DTD））。通常，最好使用外部 DTD，因为它们的定义可以跨越多个文档使用；否则冗余不可避免，并且长期维护一致性会变得困难。

2. 元素

考虑上一节中的元素：

```
<lecturer>
  <name>David Billington</name>
  <phone>+61-7-3875 507</phone>
</lecturer>
```
233

这个元素类型⊖的一个 DTD 类似这样：

```
<!ELEMENT lecturer (name,phone)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
```

这段 DTD 的含义如下。

- 元素类型 `lecturer`、`name` 和 `phone` 可以在文档中使用。
- 一个 `lecturer` 元素按序包含一个 `name` 元素和一个 `phone` 元素。
- `name` 元素和 `phone` 元素可以有任何内容。在 DTD 中，`#PCDATA` 是元素唯一的原子类型。

如下描述一个 `lecturer` 元素包含一个 `name` 元素或者一个 `phone` 元素：

```
<!ELEMENT lecturer (name|phone)>
```

当我们想要以任何顺序指明一个 `lecturer` 元素包含一个 `name` 元素和一个 `phone` 元

⊖ 元素类型 `lecturer` 和该类型的一个特定元素，例如 David Billington，之间的区分应该是清晰的。类型 `lecturer` 的所有特定元素（称为 `lecturer` 元素）共享这里定义的相同结构。

表时，会更困难一些。我们只能使用这个技巧

```
<!ELEMENT lecturer ((name,phone)|(phone,name))>
```

然而，这种方法在实际使用中受到局限（试想一下以任何顺序时的 10 个元素）。

3. 属性

考虑上一节中的元素： 234

```
<order orderNo="23456" customer="John Smith"
           date="October 15, 2002">
  <item itemNo="a528" quantity="1"/>
  <item itemNo="c817" quantity="3"/>
</order>
```

它的一个 DTD 类似这样：

```
<!ELEMENT order (item+)>
<!ATTLIST order
  orderNo      ID       #REQUIRED
  customer     CDATA    #REQUIRED
  date         CDATA    #REQUIRED>
<!ELEMENT item EMPTY>
<!ATTLIST item
  itemNo       ID       #REQUIRED
  quantity     CDATA    #REQUIRED
  comments     CDATA    #IMPLIED>
```

与上一个例子相比，一个新的地方是 `item` 元素类型定义为空。另一个新的地方是在 `order` 元素类型的定义中，在 `item` 之后出现的 +。它是势运算符（cardinality operator）之一：

?：出现零次或一次。

*：出现零次或多次。

+：出现一次或多次。

没有势运算符就表示恰好一次。

除了定义元素以外，还要定义属性。这通过一个属性列表（attribute list）来实现。第一个组件是列表运用到的元素类型的名称，接下来是一些属性名称、属性类型和取值类型构成的三元组。 235 一个属性（attribute）名称是一个可以在一个使用 DTD 的 XML 文档中使用的名称。

4. 属性类型

属性类型与预定义数据类型类似，但其选择非常有限。最重要的属性类型是：

• `CDATA`，一个字符串（字符序列）。

- ID，整个 XML 文档范围内唯一的一个名称。
- IDREF，对另一个元素的一个引用，该元素 ID 属性与此 IDREF 属性具有相同的取值。
- IDREFS，一系列 IDREF。
- $(v_1|\cdots|v_n)$，所有可能取值的枚举。

这样的选择并不令人满意。例如，日期和数字不能被指明，它们只能被解释为字符串(CDATA)，因此，它们的特定结构不能被强制。

5. 取值类型

有 4 种取值类型。

- #REQUIRED。这个元素类型每次在 XML 文档中出现时，这个属性也必须出现。在上一个例子中，itemNo 和 quantity 必须总是出现在 item 元素内。
236 - #IMPLIED。这个属性的出现是可选的。在这个例子中，注释是可选的。
- #FIXED "取值"。每个元素必须有这个属性，其总是取 DTD 中 #FIXED 之后给定的取值。在 XML 文档中给定某个取值是无意义的，因为它被固定值覆盖了。
- "取值"。这指定了这个属性的默认值。如果一个具体的取值出现在 XML 文档中，它会覆盖默认值。例如，电子邮件系统的默认编码为 "mime"，但也可以用 "binhex"，如果用户显式指定。

6. 引用

这是一个使用 IDREF 和 IDREFS 的例子。首先我们给出一个 DTD：

```
<!ELEMENT family (person*)>
<!ELEMENT person (name)>
<!ELEMENT name (#PCDATA)>
<!ATTLIST person
  id        ID      #REQUIRED
  mother    IDREF   #IMPLIED
  father    IDREF   #IMPLIED
  children  IDREFS  #IMPLIED>
```

遵循这个 DTD 的一个 XML 元素如下：

```
<family>

  <person id="bob" mother="mary" father="peter">
    <name>Bob Marley</name>
  </person>

  <person id="bridget" mother="mary">
```
238
```
    <name>Bridget Jones</name>
```

```
  </person>

  <person id="mary" children="bob bridget">
    <name>Mary Poppins</name>
  </person>

  <person id="peter" children="bob">
    <name>Peter Marley</name>
  </person>

</family>
```

读者应该研究 person 之间的那些引用关系。

7. XML 实体

一个 XML 实体可以扮演多种角色，例如重复字符的占位符（作为一种快捷记法），一段外部数据（例如 XML 或其他）或者作为元素的声明的一部分。例如，假设一个文档有几个版权声明都提到了当前的年份，那么声明一个实体就有意义了。

```
<!ENTITY thisyear "2007">
```

于是，在需要包含当前年份的每个地方，我们都可以使用实体引用 &thisyear; 来代替。这样，要想将整个文档的年份值更新为"2008"意味着只要改变这个实体声明就行了。

8. 一个总结性的例子

作为一个最终的例子，我们为之前呈现的 email 元素给出一个 DTD：

```
<!ELEMENT email (head,body)>
<!ELEMENT head (from,to+,cc*,subject)>          238
<!ELEMENT from EMPTY>
<!ATTLIST from
  name      CDATA     #IMPLIED
  address   CDATA     #REQUIRED>
<!ELEMENT to EMPTY>
<!ATTLIST to
  name      CDATA     #IMPLIED
  address   CDATA     #REQUIRED>
<!ELEMENT cc EMPTY>
<!ATTLIST cc
  name      CDATA     #IMPLIED
  address   CDATA     #REQUIRED>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT body (text,attachment*)>
```

```
<!ELEMENT text (#PCDATA)>
<!ELEMENT attachment EMPTY>
<!ATTLIST attachment
  encoding    (mime|binhex) 'mime'
  file        CDATA       #REQUIRED>
```

我们浏览这个 DTD 中一些有趣的部分。

- 一个 `head` 元素按序包含了一个 `from` 元素、至少一个 `to` 元素、零个或更多的 `cc` 元素以及一个 `subject` 元素。
- 在 `from`、`to` 和 `cc` 元素中，`name` 属性并不是必需的；另一方面，`address` 属性总是必需的。
- 一个 `body` 元素包含一个 `text` 元素，可能还会跟着一些 `attachment` 元素。
- 239 一个 `attachment` 元素的 `encoding` 属性的取值必须是 "`mime`" 或者是 "`binhex`"，前者是默认值。

我们以关于 DTD 的两个额外讨论作为结束。首先，一个 DTD 可以被解释为一个扩展的巴科斯范式（Extended Backus-Naur Form，EBNF）。例如，以下声明

```
<!ELEMENT email (head,body)>
```

等价于规则

```
email ::= head body
```

其意味着一封电子邮件先后包含一个头和一个体。其次，DTD 中允许递归定义。例如，

```
<!ELEMENT bintree ((bintree root bintree)|emptytree)>
```

定义了二叉树：一棵二叉树是一棵空树，或者由一棵左子树、一个根和一棵右子树组成。

A.2.2 XML 模式

XML 模式为定义 XML 文档的结构提供了一种特别丰富的语言。它的一个特征是其语法就基于 XML 本身。这一设计决策极大地提升了可读性，但更重要的是，它也允许技术的大幅复用。不再像 DTD 要求的那样需要去写单独的解析器、编辑器、优美的输出器等来谋求一种单独的语法；任何 XML 都可以做到。一个甚至更重要的改进是复用和改善模式的可能性。XML 模式支持通过扩展或限制现有类型来定义新的类型。与基于 XML 的语法结合起来，这一特征支持从其他模式来构建模式，因而减少了工作负担。最后，XML 模式提供了一个数据类型的精良集合，可以在 XML 文档中使用（DTD 仅局限于字符串）。

240 一个 XML 模式是一个带有开始标签的元素，形如

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
  version="1.0">
```

这个元素使用了可以在 W3C 网站找到的 XML 模式的模式。它可以说是构建新模式的基础。前缀 `xsd` 指称那个模式的命名空间（下一节详细解释命名空间）。如果前缀在 `xmlns` 属性中被省略了，那么我们默认使用来自这个命名空间中的元素：

```
<schema
  xmlns="http://www.w3.org/2000/10/XMLSchema"
  version="1.0">
```

以下我们省略 `xsd` 前缀。

现在，我们来看 `schema` 元素。它们最重要的内容是元素和属性类型的定义，使用数据类型来定义。

1. 元素类型

元素类型的语法是：

```
<element name="..."/>
```

并且它们可以有一些可选属性，例如类型，

```
type="..."
```

或者势约束

- `minOccurs="x"`，其中 x 可以是任何自然数（包括零）。
- `maxOccurs="x"`，其中 x 可以是任何自然数（包括零）或者无限制。 241

`minOccurs` 和 `maxOccurs` 是 DTD 提供的势运算符 ?、* 和 + 的泛化。当势约束没有显式提供时，`minOccurs` 和 `maxOccurs` 默认值为 1。

下面是一些例子。

```
<element name="email"/>

<element name="head" minOccurs="1" maxOccurs="1"/>

<element name="to" minOccurs="1"/>
```

2. 属性类型

属性类型的语法是：

```
<attribute name="..."/>
```

并且它们可以有一些可选属性，例如类型，

```
type="..."
```

或者存在（对应于 DTD 中的 `#OPTIONAL` 和 `#IMPLIED`），

`use="x"`，其中 x 可以是 `optional` 或 `required` 或 `prohibited`，

或者一个默认值（对应于 DTD 中的 `#FIXED` 和默认值）。

下面是一些例子：

```
<attribute name="id" type="ID" use="required"/>

<attribute name="speaks" type="Language" use="optional"
242    default="en"/>
```

3. 数据类型

我们已经认识到了数据类型在选择上的高度局限性是 DTD 的一个关键弱点。XML 模式为定义数据类型提供了强大的能力。首先有各种内置数据类型（built-in data type），这里只罗列了一些。

- 数值数据类型，包括 integer、short、Byte、long、float、decimal。
- 字符串数据类型，包括 string、ID、IDREF、CDATA、language。
- 日期和时间数据类型，包括 time、date、gMonth、gYear。

也有用户定义数据类型（user-defined data type），包括不能使用元素或属性的简单数据类型（simple data type），和可以使用元素和属性的复杂数据类型（complex data type）。我们首先讨论复杂类型，简单数据类型的讨论推迟到我们讨论限制的时候。复杂类型从现有的数据类型定义，通过定义一些属性（如果有）并使用

- sequence，现有数据类型元素的一个序列，重要之处在于按一种预定义顺序出现。
- all，一组必须出现但其顺序并不重要的元素。
- choice，一组元素，其中之一将被选中。

这是一个例子：

```
<complexType name="lecturerType">
  <sequence>
    <element name="firstname" type="string"
      minOccurs="0" maxOccurs="unbounded"/>
    <element name="lastname" type="string"/>
243  </sequence>
  <attribute name="title" type="string" use="optional"/>
</complexType>
```

其含义是一个 XML 文档中的一个被声明为 lecturerType 类型的元素可以有一个 title 属性；它也可以包含任意数量的 firstname 元素，并且必须仅含一个 lastname 元素。

4. 数据类型扩展

现有数据类型可以被新的元素或属性扩展。作为一个例子，我们扩展 lecturer 数据类型。

```
<complexType name="extendedLecturerType">
  <complexContent>
```

```
        <extension base="lecturerType">
          <sequence>
            <element name="email" type="string"
              minOccurs="0" maxOccurs="1"/>
          </sequence>
          <attribute name="rank" type="string" use="required"/>
        </extension>
      </complexContent>
    </complexType>
```

在这个例子中，`lecturerType` 被一个 `email` 元素和一个 `rank` 属性扩展。形成的数据类型看起来像这样：

```
    <complexType name="extendedLecturerType">
      <sequence>
        <element name="firstname" type="string"
          minOccurs="0" maxOccurs="unbounded"/>
        <element name="lastname" type="string"/>
        <element name="email" type="string"
          minOccurs="0" maxOccurs="1"/>
      </sequence>
      <attribute name="title" type="string" use="optional"/>
      <attribute name="rank" type="string" use="required"/>
    </complexType>
```

244

在原始和扩展类型之间存在一个层次关系。*扩展类型的实例也是原始类型的实例。*它们可以包含额外的信息，但不能包含更少的信息或者错误类型的信息。

5. 数据类型限制

一个现有的数据类型也可以通过在某些取值上增加约束来限制。例如，可以添加新的 `type` 和 `use` 属性，或者收紧 `minOccurs` 和 `maxOccurs` 的数值约束。

更重要的是，理解限制不是扩展的对立过程。限制并不通过删除元素或属性来实现。因此，以下层次关系仍然成立：*限制类型的实例也是原始类型的实例。*它们至少要满足原始类型的约束和一些新的约束。

作为一个例子，我们限制 `lecturer` 数据类型如下：

```
    <complexType name="restrictedLecturerType">
      <complexContent>
        <restriction base="lecturerType">
          <sequence>
            <element name="firstname" type="string"
              minOccurs="1" maxOccurs="2"/>
```

```
        </sequence>
245     <attribute name="title" type="string" use="required"/>
      </restriction>
    </complexContent>
  </complexType>
```

收紧的约束用粗体显示。读者应当将它们与原始值进行比较。

简单数据类型也可以通过限制现有数据类型来定义。例如，我们可以如下定义一个取值为 1 ～ 31 的 `dayOfMonth` 类型：

```
<simpleType name="dayOfMonth">
  <restriction base="integer">
    <minInclusive value="1"/>
    <maxInclusive value="31"/>
  </restriction>
</simpleType>
```

也可以通过罗列所有可能的取值来定义一个数据类型。例如，我们可以如下定义一个 `dayOfWeek` 数据类型：

```
<simpleType name="dayOfWeek">
  <restriction base="string">
    <enumeration value="Mon"/>
    <enumeration value="Tue"/>
    <enumeration value="Wed"/>
    <enumeration value="Thu"/>
    <enumeration value="Fri"/>
    <enumeration value="Sat"/>
    <enumeration value="Sun"/>
  </restriction>
246 </simpleType>
```

6. 一个总结性的例子

这里我们为 `email` 定义一个 XML 模式，使得它可以和早前提供的 DTD 相比较。

```
<element name="email" type="emailType"/>
<complexType name="emailType">
  <sequence>
    <element name="head" type="headType"/>
    <element name="body" type="bodyType"/>
  </sequence>
</complexType>
```

```
<complexType name="headType">
  <sequence>
    <element name="from" type="nameAddress"/>
    <element name="to" type="nameAddress"
      minOccurs="1" maxOccurs="unbounded"/>
    <element name="cc" type="nameAddress"
      minOccurs="0" maxOccurs="unbounded"/>
    <element name="subject" type="string"/>
  </sequence>
</complexType>

<complexType name="nameAddress">
  <attribute name="name" type="string" use="optional"/>
  <attribute name="address" type="string" use="required"/>
</complexType>

<complexType name="bodyType">
  <sequence>
```
247
```
    <element name="text" type="string"/>
    <element name="attachment" minOccurs="0"
        maxOccurs="unbounded">
      <complexType>
        <attribute name="encoding" use="optional"
            default="mime">
          <simpleType>
            <restriction base="string">
              <enumeration value="mime"/>
              <enumeration value="binhex"/>
            </restriction>
          </simpleType>
        </attribute>
        <attribute name="file" type="string"
            use="required"/>
      </complexType>
    </element>
  </sequence>
</complexType>
```

注意，一些数据类型是单独定义并且是命了名的，而剩下的是在其他类型内定义并且是匿名定义的（attachment 元素和 encoding 属性的类型）。一般而言，如果一个类型只使

用一次，为了局部使用而匿名定义它就有意义了。然而，当嵌套很深时，这个方法很快就会到达它的极限。

A.3 命名空间

248 使用 XML 作为一种通用（元）标记语言的一项主要优势是可以存取各种来源的信息：用技术术语来说，一个 XML 文档可以使用超过一种 DTD 或模式。但因为每个结构化文档是独立开发的，命名冲突（name clash）不可避免地会出现。如果 DTD A 和 DTD B 用不同的方式定义了一个元素类型 e，那么一个试图去验证一个出现了 e 元素的 XML 文档的解析器必须被告知去用哪个 DTD 来验证。

技术解决方案很简单：通过为每个 DTD 或模式使用一个不同的前缀来消解歧义。前缀和局部名称之间用冒号分隔：

```
prefix:name
```

作为一个例子，考虑一所（假想的）澳大利亚大学——例如格里菲斯大学，和一所美国大学——例如肯塔基大学，合作为在线学生提供一个统一的视图。每所学校用它自己的术语，因而存在一些差异。例如，美国的讲师不被视为常规教员，而在澳大利亚却是肯定的（事实上，他们对应于美国的助理教授）。下面这个例子展现了如何消解歧义。

```
<?xml version="1.0" encoding="UTF-16"?>
<vu:instructors
      xmlns:vu="http://www.vu.com/empDTD"
      xmlns:gu="http://www.gu.au/empDTD"
      xmlns:uky="http://www.uky.edu/empDTD">
   <uky:faculty
      uky:title="assistant professor"
      uky:name="John Smith"
      uky:department="Computer Science"/>
   <gu:academicStaff
      gu:title="lecturer"
      gu:name="Mate Jones"
      gu:school="Information Technology"/>
</vu:instructors>
```

249

因此，命名空间在一个元素内声明，并且可以用于该元素及其孩子（元素和属性）。一个命名空间声明具有以下形式：

```
xmlns:prefix="location"
```

其中 `location` 可以是 DTD 或模式的地址。如果未指定前缀，例如，

```
xmlns="location"
```

那么 location 就作为其默认值。例如，上一个例子等价于以下文档：

```
<?xml version="1.0" encoding="UTF-16"?>
<vu:instructors
    xmlns:vu="http://www.vu.com/empDTD"
    xmlns="http://www.gu.au/empDTD"
    xmlns:uky="http://www.uky.edu/empDTD">
  <uky:faculty
    uky:title="assistant professor"
    uky:name="John Smith"
    uky:department="Computer Science"/>
  <academicStaff
    title="lecturer"
    name="Mate Jones"
    school="Information Technology"/>
</vu:instructors>
```
[250]

A.4 寻址和查询 XML 文档

在关系数据库中，一个数据库的一部分可以通过诸如 SQL 查询语言来选择和检索。对 XML 文档也是如此。存在一些查询语言的提案，例如 XQL、XML-QL 和 XQuery。

XML 查询语言的核心概念是路径描述（path expression），指明了如何抵达 XML 文档的树表示中的一个节点或一个节点集。我们用 XPath 的形式介绍路径描述，因为它们除了查询以外还有别的用处——转换 XML 文档。

XPath 是一种寻址 XML 文档中特定部分的语言。它操作 XML 的树模型，并具有非 XML 语法。关键概念是路径描述。它们可以是：

- 绝对的（从树根开始）；在语法上，它们以符号 / 开头，其指明了文档的根，在文档根元素的上一层；或者：
- 相对于一个上下文节点。

考虑以下 XML 文档：

```
<?xml version="1.0" encoding="UTF-16"?>
<!DOCTYPE library PUBLIC "library.dtd">
<library location="Bremen">
  <author name="Henry Wise">
    <book title="Artificial Intelligence"/>
    <book title="Modern Web Services"/>
    <book title="Theory of Computation"/>
  </author>
  <author name="William Smart">
```

251
```
      <book title="Artificial Intelligence"/>
    </author>
    <author name="Cynthia Singleton">
      <book title="The Semantic Web"/>
      <book title="Browser Technology Revised"/>
    </author>
</library>
```

它的树表示如图 A-2 所示。

以下我们用一些路径描述的例子来示例 XPath 的能力。

1）寻址所有 author 元素。

```
/library/author
```

这一路径描述寻址到紧接在根下的 library 元素节点的孩子中的所有 author 元素。使用一个序列 $n_1,\cdots,n_k$，其中每个 t_{i+1} 是 t_i 的子节点，我们通过树表示定义了一条路径。

2）对于一个例子的另一种解决方案是：

```
//author
```

这里 // 表明我们应该考虑文档中的所有元素，并检查它们是否具有类型 author。换句话说，这一路径描述寻址文档中任何位置的所有 author 元素。由于我们 XML 文档的特殊结构，这一描述和前一个会产生相同的结果，但一般来说它们可能导致不同的结果。

252-253 3）在 library 元素节点内寻址 location 属性节点。

```
/library/@location
```

符号 @ 用来指代属性节点。

4）在文档各处的 book 元素内寻址所有值为“Artificial Intelligence”的 title 属性节点（参见图 A-3）。

```
//book/@title=[.="Artificial Intelligence"]
```

5）寻址所有题为“Artificial Intelligence”的图书（参见图 A-4）。

```
//book[@title="Artificial Intelligence"]
```

我们将一条方括号内的测试称为一个过滤描述（filter expression）。它限制了被寻址的节点集。注意这个描述和查询 4 中的描述的区别。这里，我们寻址那些标题满足一个特定条件的 book 元素。在查询 4 中，我们收集 book 元素的 title 属性节点。比较图 A-3 和 A-4 将看出区别来。

6）寻址 XML 文档中的第一个 author 元素节点。

```
//author[1]
```

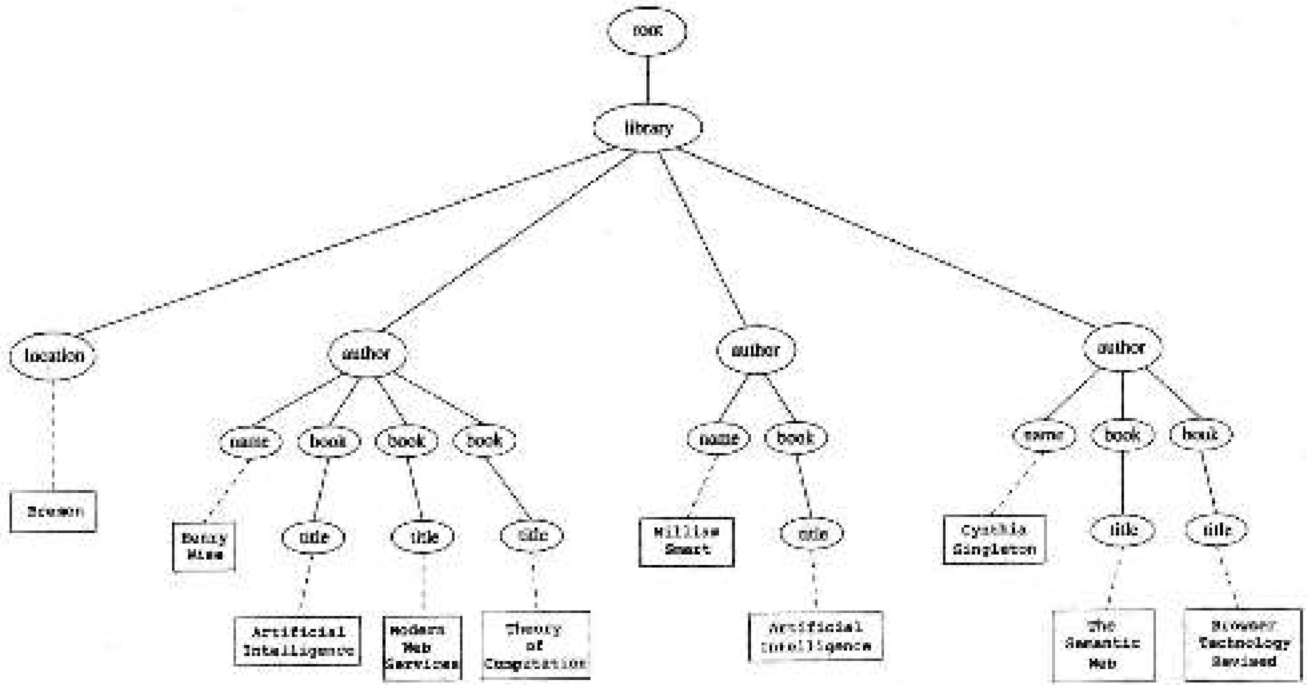


图 A-2 一个图书馆文档的树表示

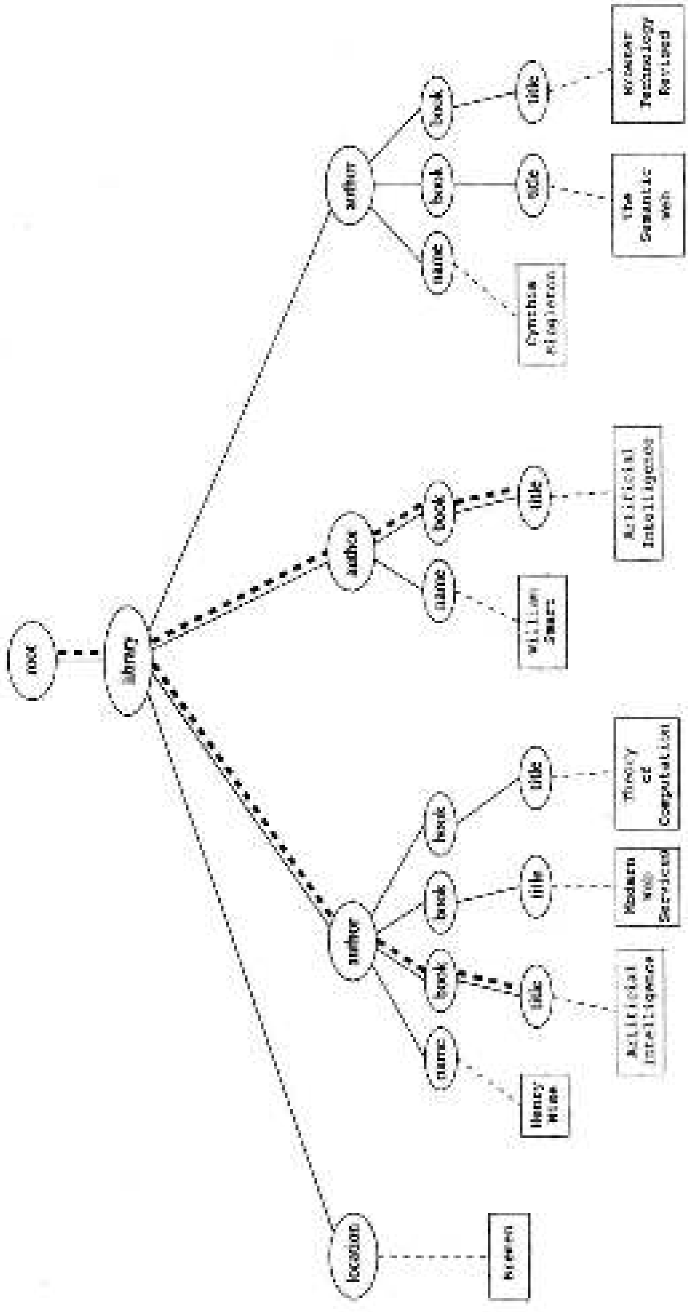

root
library
location
author
name
book
title
Artificial Intelligence
Modern Web Services
Theory of Computation
William Smart
The Semantic Web

图 A-3 查询 4 的树表示

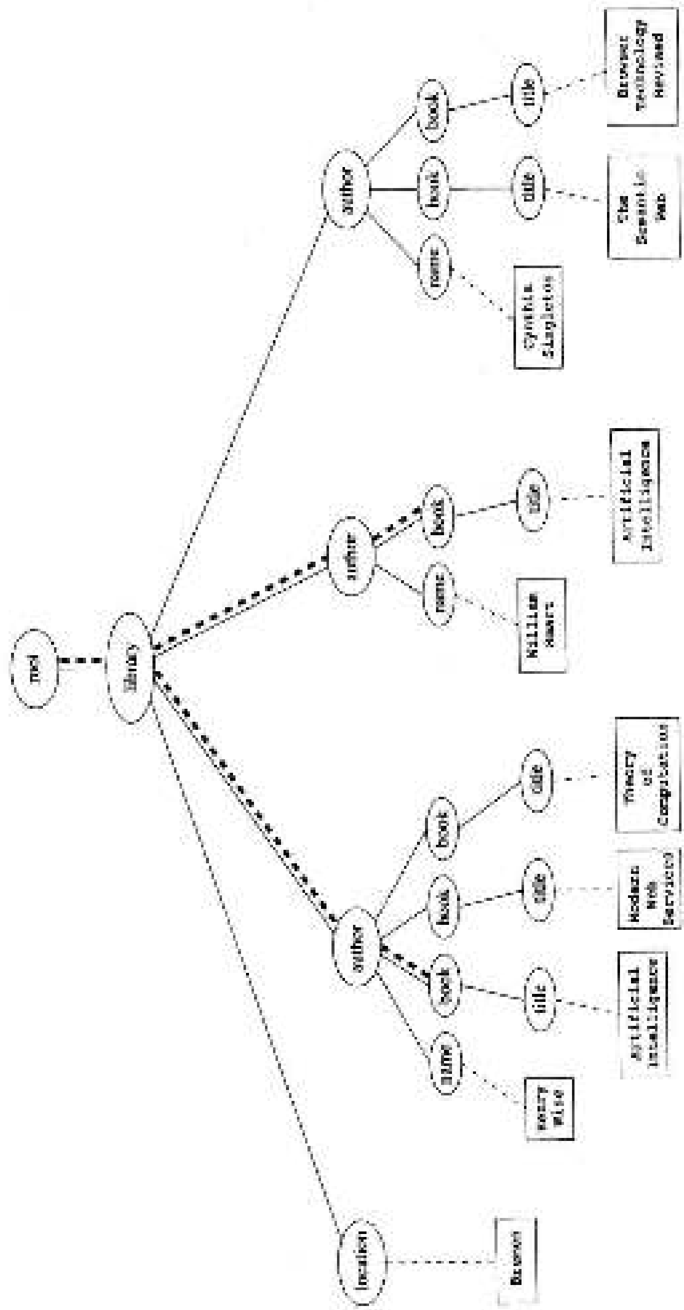

root
library
location
Boston
author
name
Henry Wise
book
title
Artificial Intelligence
Modern Web Services
Theory of Computation
William Smart
Cynthia Singleton
The Semantic Web
Browser Technology Revised

图 A-4 查询 5 的树表示

7）寻址文档中的第一个 author 元素节点中的最后一个 book 元素。

```
//author[1]/book[last()]
```

8）寻址所有没有 title 属性的 book 元素节点。

```
//book[not (@title)]
```

这些例子体现了路径描述的描述能力。一般而言，一个路径描述由一系列由斜线分隔的
254 步骤组成。一个步骤（step）由一个轴选择符、一条节点测试和一个可选的谓词组成。

- 一个轴选择符（axis specifier）决定了被寻址的节点和上下文节点之间的树关系。例如父母、祖先、孩子（默认）、兄弟和属性节点。// 就是这样的一个轴选择符，它指代后裔或者自身。
- 一条节点测试（node test）指明了要寻址哪些节点。最常见的节点测试是元素名称（可以使用命名空间信息），但也有其他的。例如，* 寻址所有元素节点，comment() 指所有注释节点，等等。
- 谓词（predicate）或称过滤描述（filter expression），是可选的，用来精化被寻址的节点集。例如，描述 [1] 选择第一个节点，[position()=last()] 选择最后一个节点，[position() mod 2 = 0] 选择偶数节点，等等。

我们已经呈现的是缩写的语法，但 XPath 实际上有一个更复杂的完整语法。参考文献在这个附录的最后可以找到。

A.5 处理

到目前为止，我们还没有提供任何关于如何显示 XML 文档的信息。这样的信息是必要的，因为不同于 HTML 文档，XML 文档并不包含格式信息。其好处是当运用不同的样式表（style sheet）时，一个给定的 XML 文档可以用各种方式呈现。例如，考虑以下 XML 元素：

255
257

```
<author>
  <name>Grigoris Antoniou</name>
  <affiliation>University of Bremen</affiliation>
  <email>ga@tzi.de</email>
</author>
```

如果用了一个样式表，那么输出可能像如下这样：

Grigoris Antoniou

University of Bremen

ga@tzi.de

或者，如果用了一个不同的样式表，可能如下显现：

Grigoris Antoniou

University of Bremen

ga@tzi.de

样式表可以用多种语言来写，例如 CSS2（层叠样式表第 2 级）。另一种可能是 XSL（可扩展样式表语言）。

XSL 包括一种转换语言（XSLT）和一种格式化语言。显然，它们中的每一种都是一个 XML 应用。XSLT 指明了一个输入 XML 文档被转换为另一个 XML 文档（一个 HTML 文档或纯文本）的规则。输出文档可以使用与输入文档相同的 DTD 或模式，或者可以使用一个完全不同的词汇表。

XSLT（XSL 转换）可以独立于格式化语言而使用。它能够将数据和元数据从一种 XML 表示迁移为另一种，这使得它成为基于 XML 的应用的一个最有价值的工具。通常，当使用不同 DTD 或模式的应用需要通信时，会选择 XSLT。XSLT 是一个用于机器对内容的处理而与为人的阅读显示信息毫不相关的工具。尽管如此，接下来我们仅使用 XSLT 来显示 XML 文档。

定义一个 XML 文档呈现的一种方式是将其转换为一个 HTML 文档。这里是一个例子。我们定义一个用于上面作者例子的 XSLT 文档。 258

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/author">
    <html>
      <head><title>An author</title></head>
      <body bgcolor="white">
        <b><xsl:value-of select="name"/></b><br></br>
        <xsl:value-of select="affiliation"/><br></br>
        <i><xsl:value-of select="email"/></i>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

这个样式表作用于之前的 XML 文档后的输出产生以下 HTML 文档（现在定义了呈现）：

```
<html>
  <head><title>An author</title></head>
  <body bgcolor="white">
    <b>Grigoris Antoniou</b><br>
    University of Bremen<br>
    <i>ga@tzi.de</i>
```

```
  </body>
</html>
```

让我们来观察一下。XSLT 文档是 XML 文档，因此 XSLT 建立在 XML 之上（即它是一个 XML 应用）。XSLT 文档定义了一个模板（template），这种情况下，就是一个 HTML 文档，包
259 含一些待插入内容的占位符（参见图 A-5）。

```
<html>
<head><title>An author</title></head>
<body bgcolor="white">
  <b>...</b><br>
  ...<br>
  <i>...</i>
</body>
</html>
```

图 A-5 一个模板

在之前的 XSLT 文档中，`xsl:value-of` 检索一个元素的取值并将它复制到输出文档中。即它将一些内容放入模板中。

现在假设我们有一个包含几个作者细节的 XML 文档。显然单独处理每个 `author` 元素是一种精力的浪费。在这样的情况下，为 `author` 元素定义一个特殊的模板，被主模板调用。我们在下面这个输入文档中示例这个方法：

```
<authors>
  <author>
    <name>Grigoris Antoniou</name>
    <affiliation>University of Bremen</affiliation>
    <email>ga@tzi.de</email>
  </author>
  <author>
    <name>David Billington</name>
    <affiliation>Griffith University</affiliation>
    <email>david@gu.edu.net</email>
  </author>
</authors>
```
260

我们定义以下 XSLT 文档：

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
```

```
    <html>
    <head><title>Authors</title></head>
    <body bgcolor="white">
      <xsl:apply-templates select="authors"/>
      <!-- Apply templates for AUTHORS children -->
    </body>
    </html>
  </xsl:template>

  <xsl:template match="authors">
    <xsl:apply-templates select="author"/>
  </xsl:template>

  <xsl:template match="author">
    <h2><xsl:value-of select="name"/></h2>
    Affiliation:<xsl:value-of select="affiliation"/><br>
    Email: <xsl:value-of select="email"/>
    <p>
  </xsl:template>
</xsl:stylesheet>
```

产生的输出是：

```
<html>
<head><title>Authors</title></head>                261
<body bgcolor="white">
  <h2>Grigoris Antoniou</h2>
  Affiliation: University of Bremen<br>
  Email: ga@tzi.de
  <p>
  <h2>David Billington</h2>
  Affiliation: Griffith University<br>
  Email: david@gu.edu.net
  <p>
</body>
</html>
```

xsl:apply-templates 元素让上下文节点的所有孩子与选择的路径描述匹配。例如，如果当前模板作用于 /（即如果当前上下文节点是根），那么元素 xsl:apply-templates 作用于根元素——在这个情况下就是 authors 元素（记住 / 在根元素之上）。并且如果当前节点是 authors 元素，那么 xsl:apply-templates select="author" 元素让 author

元素的模板作用于 authors 元素的所有 author 孩子。

好的做法是为文档中的每种元素类型定义一个模板。即使没有特定的处理要作用于某些元素——比如我们的例子中，xsl:apply-templates 元素也应该被使用。这样，从树的根到叶子，所有模板都被运用了。

现在我们将注意力转移到属性上。假设我们希望用 XSLT 处理元素：

```
<person firstname="John" lastname="Woo"/>
```

让我们尝试可以想象的最简单的任务——元素到自身的转换。你一定会写成：

262

```
<xsl:template match="person">
  <person
    firstname="<xsl:value-of select="@firstname">"
    lastname="<xsl:value-of select="@lastname">"/>
</xsl:template>
```

然而，这并不是一个格式良好的 XML 文档，因为标签不允许出现在属性值中。但其意图是清楚的，我们希望将属性值添加到模板中。在 XSLT 中，花括号内的数据代替了 xsl:value-of 元素。为这个例子定义一个模板的正确方式如下：

```
<xsl:template match="person">
  <person
    firstname="{@firstname}"
    lastname="{@lastname}"/>
</xsl:template>
```

最后，我们给出一个从一个 XML 文档到另一个的转换样例，其并未指明显示部分。我们再一次使用 authors 文档作为输入，并如下定义一个 XSLT 文档：

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <authors>
      <xsl:apply-templates select="authors"/>
    </authors>
  </xsl:template>
  <xsl:template match="authors">
    <xsl:apply-templates select="author"/>
  </xsl:template>

  <xsl:template match="author">
```

```
    <author>
      <name><xsl:value-of select="name"/></name>
      <contact>
        <institute>
          <xsl:value-of select="affiliation"/>
        </institute>
        <email><xsl:value-of select="email"/></email>
      </contact>
    </author>
  </xsl:template>

</xsl:stylesheet>
```

输出文档的树表示如图 A-6 所示，展现了 XSLT 的树转换特性。 263 / 264

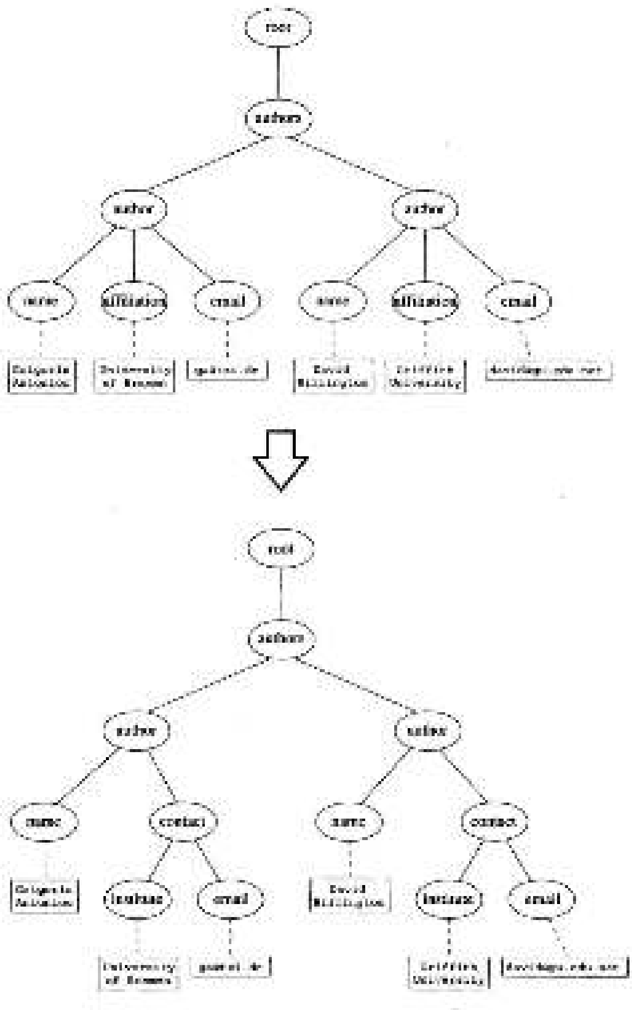


图 A-6 XSLT 作为树转换

索　　引

索引中的页码为英文原书页码，与书中页边标注的页码一致。