# OWL相关概念--XML,RDF,OWL的关系(重要)

|  |
| --- |
| 请先理解RDF和RDF Schema的知识在看这篇文章。不理解的话请本站参照RDF和RDF Schema在线手册的第一篇相关概念的文章。理解概念很重要，因为对大多数人来说都有编程的基础，可以直接看别的语言的代码。但语义网相关概念是比较新 的概念。所以建议把概念一定到搞懂。通过RDF Schema，我们可以自定义词汇了。但在我们的实际生活当中，我们用的词汇直接，都是有联系的。最简单的反义词，同义词。又比如"足球队"这个词，我们 每个人脑子都有一个概念，上场比赛需要11个队员。这都是我们在生活中积累的经验。我们想让机器也理解数据的话，最起码要和人一样，也可以定义反义词，同 义词，或者词和词之间的一些关系。这些仅仅靠RDF和RDF Schema是不够的。为了达到这个要求，就有了OWL(Web Ontology Language)的出现，Ontolory本是哲学词汇--存在论的意思。大多数中文翻译为本论。其实用原本哲学的意思就很好理解。我们就是在定义词 汇，或者词汇直接的关系，或者类之间的关系等等。我们定义了它们，它们就存在于我们的网络里了。我们看看有了它对我们到底有什么好处。比如说它可以定义类 和类之间的关系，等价性，互补排斥性，限制个数，属性的对称性等等。似乎还比较模糊，那就举一个具体的例子，比如我们描述这样一个资源：从北京到上海的距 离４００公里。人们听到了这句话后就知道了，上海到北京也是４００公里。因为我们知道起点北京到终点上海的距离，和起点上海到终点北京的距离是一样的，也 就是我们懂距离，起点，终点这３个词的概念而且知道它们之间的关系，所以我们得出上面的结论。现在我们可以用RDF和RDF Schema来定义这３个词汇，然后我们需要定义的是一个关系，起点－终点的距离等于终点－起点的距离，这里运用到了等价性。这个关系不能用RDF和 RDF Schema来定义，但是可以通过OWL来定义。当然解决这个问题方法可以有许多定义，我只是在这里举个例子。大家应该大概知道OWL是做什么的吧。那么 为什么它有这个功能呢？其实它运用了人工智能中的逻辑论，逻辑论把人们的思维用式子来表达，并且可以证明这个式子的正确。其实在语义网的定义中，吸收了很 多知识的优点，面向对象，逻辑轮，树结构等等。所以大家学起来，有时会有似曾相识的感觉。下面把大家容易混淆的几个概念XML,XML Schema,RDF,RDF Schema,OWL直接的关系拿出来讲一下。 主要的理解思想就是在语义网的定义中遇到问题了，就会有新的事物被定义或者说是出现从HTML到XMLHTML的记述是我们最常用的，本来它是用来描述资源的，但是它记录的只有人可以看懂。还有它第一结构不明显，很难被应用程序解析，第二，记述的 局限性，比如某些科学符号无法表达等等。为了让应用程序好理解，就需要有良好的结构。最简单但也是功能很强大的树状结构就被采用了。我们以树状结构来记录 数据，这就是XML。XML：是一种国际通用标记语言。它为在各种应用程序间的数据和meta数据的交换，提供了一致的体系和解析器(praser)。但 是对于数据的意义没有进行定义，比如说对于某个标签< 课程>,也就是本身没有意义，除了结构的不同，它和HTML一样，数据也只有人可以看懂。想让机器理解数据，首先本身的数据类型很重要，不然自己都 不知道自己是谁，或者自己是干什么的，怎么可能让机器理解。为了更好的描述资源，也就是说为了更准确的纪录数据的类型。需要我们可以自定义类型，因为预定 义的类型只有最基本的类型，不能定义大多数现实生活中的数据类型。对于数据发布人，需要他描述清楚数据类型，也就是说数据的限制等等。所以我们需要一个可 以在预定义类型上可以自定义类型，而且它也必须能被解析器解析。所以，就出现了XML Schema。XML SchemaXML Schema：XML的定义机制。和DTD文档类型定义不同，不需要另外的解析器和编辑器。定义的是XML的构造，对已有的类型进行扩张或者进行制约，从 而定义了新的类型。XML Schema提供了在XML可以利用的基本的数据类型(日期，数值等，DTD中只提供了文字列)。但是即便是这样，它只是对XML文档的构造进行了定义， 还是不能让应用程序理解数据。所以为了实现应用程序理解数据，就出现了RDF。从XML到RDFRDF有时我们会称它为语言，其实这是不对的，RDF是一种书写规范，正确的翻译为资源描述框架。RDF的基本构造为陈述(或者叫做声明， statement)了一个资源-资源具有的属性-属性值(主体-属性-客体)的三元组。它表现的是一个数据模型，通俗的说一个陈述就是一个什么事物（资 源）具有什么属性（属性），这个属性是怎样的属性（属性值）。我们为了描述抽象的数据模型，需要具体的书写，这就用到了XML，这样RDF就继承了XML 的优点。但是不用XML的构成，利用其他的构成也可以表现RDF。所以对RDF而言，XML不是必要的(一般我们的文档格式为RDF/XML就是这个原 因，用XML表示的RDF)。我们终于可以定义一个让机器理解的词汇了，但是它也遇到了一个问题，就是预定义的词汇根本不够用。我们需要自定义一些词汇。 为了可以自定义词汇，就出现了RDF Schema。RDF SchemaRDF Schema (RDFS)翻译为资源描述框架的定义机制，大家也许会比较容易联想到XML和XML Schema的关系，认为它们是同样的。这个想法也是不对的，XML Schema是用来对XML的构造进行定义，而RDFS是用来对RDF数据模型用到的词汇进行定义。我们需要了解一个RDF的特征，它是以属性为中心的思 考方法。不是具体定义属性的值，而是定义了拥有这个属性的主体(资源)的范围(定义域)，和这个属性可以取到的值的范围(值域)。这样就我们可以比较精确 地定义词汇。为什么说比较精确，而不说精确，因为一个单独存在的词汇对我们毫无意义，即便机器可以理解它。我们语言中的词汇都是互相有关联的，正因为我们 只有知道了词汇的相互关系，我们才可以在生活中正常交流。同样我们一直说让应用程序理解词汇，其实是为了让应用程序理解词汇以及词汇之间的关系，从而自动 地，智能地提供给我们服务。为了达到这个目的，描述词汇的关系至关重要。但是RDF和RDF Schema却不能提供给我们这样一个功能，所以也就出现了OWL。OWLOWL是RDF的扩张，提供我们更广泛的由定义RDF Schema词汇。所谓更广泛就是指可以定义词汇之间的关系，类与类的关系，属性与属性之间的关系等等。具体请参照本站的在线使用手册。总结:其实每个概念都不难理解，理解的关键在于它们为了什么而出现。OWL的出现也没有能实现最终的目的。所以语义网结构上还有几层，这几层还没有推出规范，所以暂时不做它们的使用手册。不过我会在博客中陆续介绍它们的进展情况。 |

# OWL 简介

**OWL 是一门供处理 web 信息的语言。**

**在学习之前应具备的基础知识**

在您学习 OWL 之前，应当对 XML、XML 命名空间以及 RDF 有基本的了解。

如果首先学习这些项目，请访问：

W3School 的 [XML 教程](http://www.w3school.com.cn/xml/index.asp) 和 [RDF 教程](http://www.w3school.com.cn/rdf/index.asp)。

**什么是 OWL？**

* OWL 指的是 web 本体语言
* OWL 构建在 RDF 的顶端之上
* OWL 用于处理 web 上的信息
* OWL 被设计为供计算机进行解释
* OWL 不是被设计为供人类进行阅读的
* OWL 由 XML 来编写
* OWL 拥有三种子语言
* OWL 是一项 web 标准

**什么是本体？**

本体”这个术语来自于哲学，它是研究世界上的各种实体以及他们是怎么关联的科学。

对于 web，本体则关于对 web 信息及 web 信息之间的关系的精确描述。

**Why OWL?**

OWL 是“语义网远景”的组成部分 - 目标是：

* Web 信息拥有确切的含义
* Web 信息可被计算机理解并处理
* 计算机可从 Web 上整合信息

**OWL 被设计为供计算机来处理信息**

OWL 被设计为提供一种通用的处理 Web 信息的内容的方法（而不是把它显示出来）。

OWL 被设计为由计算机应用程序来读取（而不是被人类）。

**OWL 与 RDF 不同**

OWL 与 RDF 有很多相似之处，但是较之 RDF， OWL 是一门具有更强机器解释能力的更强大的语言。

与 RDF 相比，OWL 拥有更大的词汇表以及更强大的语言。

**OWL 子语言**

OWL 有三门子语言：

* OWL Lite
* OWL DL (包含 OWL Lite)
* OWL Full (包含 OWL DL)

**OWL 使用 XML 编写**

通过使用 XML，OWL 信息可在使用不同类型的操作系统和应用语言的不同类型的计算机之间进行交换。

**OWL 实例 （机场）**

OWL 资源：<http://www.daml.org/2001/10/html/airport-ont>

类：Airport

属性：

* [elevation](http://www.daml.org/cgi-bin/hyperdaml?http://www.daml.org/2001/10/html/airport-ont#elevation)
* [iataCode](http://www.daml.org/cgi-bin/hyperdaml?http://www.daml.org/2001/10/html/airport-ont#iataCode)
* [icaoCode](http://www.daml.org/cgi-bin/hyperdaml?http://www.daml.org/2001/10/html/airport-ont#icaoCode)
* [latitude](http://www.daml.org/cgi-bin/hyperdaml?http://www.daml.org/2001/10/html/airport-ont#latitude)
* [location](http://www.daml.org/cgi-bin/hyperdaml?http://www.daml.org/2001/10/html/airport-ont#location)
* [longitude](http://www.daml.org/cgi-bin/hyperdaml?http://www.daml.org/2001/10/html/airport-ont#longitude)
* [name](http://www.daml.org/cgi-bin/hyperdaml?http://www.daml.org/2001/10/html/airport-ont#name)

制作所使用：[dumpont2.java](http://www.daml.org/2003/09/dumpont/)

**OWL 是一个 Web 标准**

OWL 于 2004 年 2 月成为一项 W3C 的推荐标准。

W3C 推荐（标准）被业界以及 web 团体奉为 web 标准。W3C推荐标准 是由 W3C 工作组开发并经 W3C 成员评审的稳定规范。

在 w3c 有关 OWL 的文档：<http://www.w3.org/2004/OWL/>

///////////////////////////////////////////////////////////////////////

目前这种结构的万维网，很像一本地图做得很差的地理书，我们对于Web中可以使用的文档和服务的了解，都是基于关键字搜索的， 同时还需要灵活地使用文档的链接和使用模式。如果没有强有力的工具的支持，这么大规模的数据是很难管理的，为了能够给Web绘制出更为详实的地图，计算代理需要对于网络上可用资源的内容和能力做一个机器能够读得懂的描述。这些描述是人类能够读得懂的信息的扩展。

OWL，这种本体描述语言，可以用来描述Web文档和应用中内在的类和关系。

这篇文章解释了OWL语言的使用：

* 通过定义类以及类的属性来形式化某个领域；
* 定义个体并说明它们之间的属性；
* 在OWL语言的形式化语义允许的层次上，对类和个体进行推理。

本文的各章节间是按照类、属性、个体的集合的定义给出来的，从最简单的概念开始，逐渐过渡到更为复杂的概念。

## 本文档的状态

本文档已被W3C成员及其他相关方面审阅，并已被W3C总监（W3C Director）批准为W3C推荐标准（W3C Recommendation）。W3C制定推荐标准的任务是使之受到关注，并促使其被广泛应用。这将增强Web的功能性与互操作性。

本文档是W3C关于Web本体语言OWL的推荐标准的六个部分之一。 它已经被Web 本体工作小组(小组章程) 作为W3C语义Web行动 (行动声明) 的一部分于2004年2月10日发布。

本文档的早期版本中所描述的关于OWL的设计已被广泛评阅，并已满足工作小组的技术需求。工作小组充分考虑所有收到的意见，并做了必要的修改。本文档自从候选推荐标准版本以来的所有修改都在文后的变更日志中。

欢迎通过public-webont-comments@w3.org (历史存档)提出您的意见，也可以通过www-rdf-logic@w3.org (mailto:www-rdf-logic@w3.org)(历史存档) 参与相关技术的讨论。

可以访问到有关实现的一个列表。  
W3C维护着一个与这些工作相关的专利声明的目录。

这节描述了本文档在发布时的状态。其他文档可能替代这文档。一份当前W3C的最新出版物的目录和这个技术报告的最新版本可以在 W3C技术报告索引http://www.w3.org/TR/ 上找到。

**目录**

**1. 引言. 3**

1.1. OWL的种类. 4

1.2. 本文档的结构. 5

**2. 本体的结构. 5**

2.1. 命名空间. 5

2.2. 本体头部. 7

2.3. 数据集成与隐私. 8

**3. 基本元素（Basic Elements）. 8**

3.1. 简单的类和个体. 8

3.1.1. 简单的具名类. 9

3.1.2. 个体. 11

3.1.3. 使用方面的考虑. 11

3.2. 简单属性. 12

3.2.1. 定义属性（Defining Properties）. 13

3.2.2. 属性和数据类型. 15

3.2.3. 个体的属性. 16

3.3. 属性特性. 16

3.3.1. TransitiveProperty 17

3.3.2. SymmetricProperty 17

3.3.3. FunctionalProperty 17

3.3.4. inverseOf 18

3.3.5. InverseFunctionalProperty 18

3.4. 属性限制. 19

3.4.1. allValuesFrom, someValuesFrom 19

3.4.2. 基数限制. 20

3.4.3. hasValue [OWL DL] 20

**4. 本体映射. 21**

4.1. 类和属性之间的等价关系. 21

4.2 个体间的同一性. 22

4.3. 不同的个体. 23

**5. 复杂类 [OWL DL] 24**

5.1 集合运算符 intersectionOf,unionOf,complementOf 24

5.1.1.交运算 [some uses of OWL DL] 24

5.1.2. 并运算 [OWL DL] 25

5.1.3. 补运算 [OWL DL] 26

5.2. 枚举类 oneOf [OWL DL] 26

5.3. 不相交类 disjointWith [OWL DL] 27

**6. 本体的版本控制. 28**

**7. 使用范例. 29**

7.1.葡萄酒门户网站. 29</O:P< p>

7.2. 葡萄酒主体（agent）. 30

**致谢（略）. 32**

**OWL词汇. 32**

**术语索引及引用参照. 34**

术语索引. 34

**附录A：XML + RDF基础. 35**

**附录B: 历史. 35**

**附录C：2003年12月15日发布的提议推荐标准以来的修改日志. 36**

# 1. 引言

“告诉我我应该买什么酒提供给下列菜单的每道菜，随便说一下，我不喜欢苏特恩白葡萄酒”。

目前构造一个能够查找满足这个查询的酒的Web代理会是困难的。类似地，考虑派给软件代理一个做出合理的旅行安排的任务（更多的用例，参考OWL需求文档）。

为了支持这种计算，不仅仅用关键词而是说明Web上描述的资源的含义是必要的。这个额外的解释层表述了数据的“语义”。

Web本体语言OWL是一种定义和实例化“Web本体”的语言。“本体”这个术语来自于哲学，它是研究世界上的各种实体以及他们是怎么关联的科学。一个“Web本体”可能包含了类，属性和他们的实例的描述。给出这样的一个本体，OWL形式语义说明怎么获得它的逻辑结论，也就是说，不是逐字写在本体中的事实，而是语义蕴涵的事实。这些蕴涵可以是基于单个的文档也或利用OWL机制合并在一起的多个分布的文档。

本文档是W3CWeb本体工作组（WebOnt）制定的Web本体语言的描述的一部分。 OWL综述([Overview)的文档指南部分描述了不同部分的文档以及他们怎样结合的。

当描述另外一个XML/Web标准时，有一个问题会冒出来：这个标准给了我什么XML和XML Schema不能给的。这个问题有两个答案。

·       本体和XML Schema的区别是它是一种知识表示，而不是一种消息格式。大多数来自工业界的Web标准包含了一个消息格式和协议规范的组合。这些式已经被给予一个操作语义，例如，"一旦收到订单（PurchaseOrder）的消息，从AccountFrom账号转移Amount数量的美元到AccountTo账号，并且发货(Product)"，但是这些规范并没有设计为支持此事务上下文之外的推理。例如，一般来说，没有机制让我们推出：因为这个产品的类型是夏敦埃酒（Chardonnay，一种无甜味白葡萄酒），它必定也是一种白色酒。

·       OWL本体的一个优点是会有能够对其做推理的工具。这些工具提供了不特定于某个主题领域的通用支持，而如果要构建一个能对一个特定的工业界标准XML Schema做推理的系统，它往往是特定于一个领域的。构建一个可靠的和有用的推理系统不是一项简单的工作。而创建一个本体则更为容易处理。我们的期望就是很多团体会着手本体创建。他们会得益于基于OWL语言的形式属性的第三方工具，这些工具提供了多种多样的能力，而这些能力是大部分组织难以复制的。

## 1.1. OWL的种类

OWL提供了三种表达能力递增的子语言，以分别用于特定的实现者和用户团体。

·           OWL Lite用于提供给那些只需要一个分类层次和简单约束的用户。例如，虽然OWL Lite支持支持基数限制，但只允许基数为0或1。提供支持OWL Lite的工具应该比支持表达能力更强的其他OWL语言更简单，并且从辞典（thesauri）和分类系统（taxonomy）转换到OWL Lite更为迅速。

·           OWL DL 支持那些需要最强表达能力的推理系统的用户，且这个推理系统能够保证计算的完全性（computational completeness，即所有的结论都能够保证被计算出来）和可判定性（decidability，即所有的计算都在有限的时间内完成）。它包括了OWL语言的所有成分，但有一定的限制，如类型的分离（一个类不能同时是一个个体或属性，一个属性不能同时是一个个体或类）。OWL DL 这么命名是因为它对应于[描述逻辑]，这是一个研究一阶逻辑的一个特定可判定片断的领域。OWL DL旨在支持已有的描述逻辑商业处理（business segment）和具有良好计算性质的推理系统。

·           OWL Full 支持那些需要尽管没有可计算性保证，但有最强的表达能力和完全自由的RDF语法的用户。例如，在OWL Full中，一个类可以被同时看为许多个体的一个集合以及本身作为一个个体。另外一个和OWL DL的重要区别是owl:DatatypeProperty（数据类型属性）能作为一个owl:InverseFunctionalProperty（逆函数型属性）。OWL full允许一个本体增加预定义的（RDF、OWL）词汇的含义。这样，不太可能有推理软件能支持对OWL FULL的所有成分的完全推理

名空间。第一个声明指定了缺省命名空间，即表明所有无前缀的限定名（qualified names）都出自当前本体。第二个声明为当前本体指定了前缀 vin:。第三个声明为当前文档（参见下文）指定了基准URI（base URI）。第四个声明指出食物（food）本体将用前缀food:来标识。

第五个命名空间声明指出，在当前文档中，前缀为owl:的元素应被理解是对出自http://www.w3.org/2002/07/owl#中的事物的引用。这是引入OWL词汇表的惯例用法。

OWL要依赖RDF、RDFS以及XML Schema数据类型中的构词（constructs）。在本文档中，rdf:前缀表明事物出自命名空间 http://www.w3.org/1999/02/22-rdf-syntax-ns#。接下来的两个命名空间声明分别为RDF Schema和XML Schema数据类型指定前缀rdfs:和xsd:。

为帮助书写冗长的URLs，在本体的定义之前，在文档类型声明（DOCTYPE）中提供一些实体定义（entity definitions）常常是很有用的。这些被命名空间声明定义的名称仅当作为XML标签的一部分时才具有意义。属性值（attribute values）是不具有命名空间的。但是在OWL里，我们经常要用属性值来引用本体标识符。我们可以写出它们的完整URI形式，比如“http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine#merlot”。或者，利用实体定义来简略URI的书写，例如：

 <!DOCTYPE rdf:RDF [  
     <!ENTITY vin  "http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine#" >  
     <!ENTITY food "http://www.w3.org/TR/2004/REC-owl-guide-20040210/food#" > ]>

在声明这些实体后，我们可以将“&vin;merlot”作为“http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine#merlot”的简写。

更为重要的是，这样rdf:RDf命名空间声明可以被简化，并且只需对实体声明作修改即可在整个本体范围内应用URI的变化。

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

## 2.2. 本体头部

建立了命名空间后，接下来我们通常要在owl:Ontology标签里给出一组关于本体的声明。这些标签支持一些重要的常务工作比如注释、版本控制以及其他本体的嵌入等。

 <owl:Ontology rdf:about="">   
   <rdfs:comment>An example OWL ontology</rdfs:comment>  
   <owl:priorVersion rdf:resource="http://www.w3.org/TR/2003/PR-owl-guide-20031215/wine"/>   
   <owl:imports rdf:resource="http://www.w3.org/TR/2004/REC-owl-guide-20040210/food"/>   
   <rdfs:label>Wine Ontology</rdfs:label>   
   ...  
注意：我们使用“...”表明这里有一些文本被略去了。

owl:Ontology元素是用来收集关于当前文档的OWL元数据的。它不确保文档描述一个传统意义的本体。在某些圈子里，本体不是关于个体的，而是仅仅关于某个领域的类和属性的。在使用OWL来描述一个实例数据集合时，owl:Ontology标签也许会被需要用来记录版本信息，和导入文档所依赖的一些定义。因此，在OWL里，本体一词被放宽了，已包含实例数据（如上文）。

rdf:about属性为本体提供一个名称或引用。根据标准，当rdf:about属性的值为""时，本体的名称是owl:Ontology元素的基准URI。典型地，这是一个包含本体的文档的URI。在使用了xml:base的上下文中则是一个特殊情况，这时owl:Ontology元素的基准URI也许会被设为其他URI。

rdfs:comment提供了显然必须的为本体添加注解的能力。

owl:priorVersion是一个为用于本体的版本控制系统提供相关信息（hook）的标准标签。本体的版本控制将在后面作进一步讨论。

owl:imports提供了一种嵌入机制。owl:imports接受一个用rdf:resource属性标识的参数。

导入另一个本体将把那个本体中的全部声明引入到当前本体中。为了充分利用好这一机制，通常要与命名空间声明结合使用。请注意这两种机制的区别。命名空间声明提供的是一种方便对其他本体定义的名称进行引用的方法。概念上，owl:imports用于表明包含目标本体中的声明。在导入另一个本体02时，在02中导入的其他本体也将被导入。

注意：owl:imports并不是总能成功的。正如你所料的，在涉及语义网时，对分布在Web上的资源的访问也许是不可及的。在这种情况下，工具的响应是与具体实现相关的。

注意：不必为了使用OWL本体词汇，而导入owl.rdf本体。实际上，这样导入是不推荐的。

一个理想的可被嵌入的标签集合是部分标准的Dublin Core元数据标签。该子集包含一些值为简单类型或字符串的标签。比如：Title, Creator, Description, Publisher和Date等（参见RDF声明）。

被用作注解的属性（properties）应用owl:AnnotationProperty来声明。例如

 <owl:AnnotationProperty rdf:about="&dc;creator" />

OWL提供了若干其他的机制来将当前本体与被导入本体相关联（参见本题映射部分）。

我们也可以用rdfs:label来对本体进行自然语言标注。

本体头部定义在下列标签处结束：

 </owl:Ontology>

在这段开头之后跟随的是构成本体的实际定义，最终由

</rdf:RDF>  
终止。

## 2.3. 数据集成与隐私

OWL在表达出现在多个文档中的实例信息的能力方面，支持连接来自异源的数据。下层语义为这些数据提供推理支持，这可以产生意外的结果。特别地，owl:sameAs的表达等价的能力，可被用来表达表面上不同的个体实际上是相同的。Owl:InverseFunctionalProperty也可被用来连接个体。例如，如果一个属性，比如“SocialSecurityNumber”，是一个owl:InverseFunctionalProperty，那么两个分开的个体如果具有相同的SocialSecurityNumber属性，则可被推理出是相同的个体。当个体被这样确定为相同时，来自异源的关于这些个体的信息可以被合并。这种聚合可被用来得出不可直接从单源获得的事实。

语义网的连接来自多源的信息的能力是一个理想的、强大的特性，它可被用在许多应用中。但是合并来自多源数据的能力，加上OWL的推理能力，确实存在被误用的可能。OWL用户应对潜在的隐私问题予以警惕。具体的安全方案超出了工作组的工作范畴。一些组织正在用各种不同的安全和偏好方案来处理这些问题，比如SAML和P3P。

# 3. 基本元素（Basic Elements）

一个OWL本体中的大部分元素是与类（class）、属性（property）[译注//这里的property也可译作“特性”]、类的实例（instance）以及这些实例间的关系有关的。本节给出应用这些元素所必需的语言成分。

## 3.1. 简单的类和个体

许多情况下，使用本体是为了用它进行关于个体的推理。为了在一种有效的方式下做到这一点，我们需要一种机制来描述个体所属的类以及这些个体通过类成员关系而继承得到的属性。尽管我们总能为个体声明特定的属性，但是本体的大部分能力在于基于类的推理。

有时，我们希望强调区分一个类是作为对象还是作为包含元素的集合。我们称由属于某个类的个体所构成的集合为该类的外延（extension<EM< p>

-

>）。

### 3.1.1. 简单的具名类

**Class, rdfs:subClassOf**

一个领域中的最基本概念应分别对应于各个分类层次树的根。OWL中的所有个体都是类owl:Thing的成员。因此，各个用户自定义的类都隐含地是owl:Thing的一个子类。要定义特定领域的根类，只需将它们声明为一个具名类（named class）即可。OWL也可以定义空类，owl:Nothing。

在我们所举的葡萄酒领域的例子中，我们创建三个根类：Winery，Region和ConsumableThing。

 <owl:Class rdf:ID="Winery"/>   
 <owl:Class rdf:ID="Region"/>   
 <owl:Class rdf:ID="ConsumableThing"/>   
注意：我们只是说这里有三个具有指定名称（通过语法“rdf:ID=”）的类。形式上，即使我们使用了熟悉的英语单词作为标签,但我们除了知道这些类的存在以外，仍不了解任何其他关于它们的信息。而这些类尽管存在，但它们可能没有成员。就所有目前我们所知道的信息而言，将这些类分别命名为Thing1、Thing2和Thing3与命名为上述名称没有什么区别。

记住这一点很重要，即定义可以是增量的和分布式的。特别地，我们将在后面对Winery作更多的讨论。

语法 rdf:ID="Region" 被用于引入一个名称（作为定义的一部分）。该rdf:ID属性（attribute）（[RDF]，7.2.22）类似于XML中的ID属性（attribute）。在这一文档中，我们现在可以用#Region来引用Region类，例如 rdf:resource="#Region"。而其他的本体可以通过"http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine#Region"这一完整形式来引用该名称。

另一种引用类的形式是用语法 rdf:about="#Region" 来扩展对一个资源的定义。语法 rdf:about="&ont;#x" 的使用在分布式本体的创建中是一个关键要素。它允许导入x类的定义并对它进行扩展，而不需修改源定义文档，从而支持增量构建更大的本体。

现在，我们可以在其他OWL的构建中通过这些类的标识符来引用这些类。比如对于第一个类，同样也在该文档内的话,我们就可以使用相对标识符#Winery。由于其他文档可能也需要引用这个类，因此最合理的方式是提供命名空间和实体定义，在其中包含着这个类的定义文档作为定义源：

 ...  
 <!ENTITY vin  "http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine#" >  
 <!ENTITY food "http://www.w3.org/TR/2004/REC-owl-guide-20040210/food#" >

...

<rdf:RDF xmlns:vin ="http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine#"  
          xmlns:food="http://www.w3.org/TR/2004/REC-owl-guide-20040210/food#" ... >  
 ...

给定上述定义后，我们便可以通过XML标签vin:Winery或属性（attribute）值&vin;Winery来引用winery类。更确切地说，我们总可以使用资源的完整URL来引用它们，比如这里我们可以用http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine#Winery 来引用Winery类。

rdfs:subClassOf是用于类的基本分类构造符。它将一个较具体的类与一个较一般的类关联。如果X是Y的一个子类（subclass），那么X的每个实例同时也都是Y的实例。rdfs:subClassOf关系是可传递的，即如果X是Y的一个子类，而Y又是Z的一个子类，那么X就是Z的一个子类。  
 <owl:Class rdf:ID="PotableLiquid">   
   <rdfs:subClassOf rdf:resource="#ConsumableThing" />  
   ...  
 </owl:Class>

上面，我们把PotableLiquid（可饮用的液体）定义为ConsumableThing的子类。

在基于Web的本体世界中，这两个类可定义在一个分散的本体中，而这个本体又可以作为各种不同的食物和饮料本体的基础。我们已在食物(food)本体中定义了各种不同的食物和饮料本体，并将该食物本体导入葡萄酒（wine）本体中。食物本体中包含了许多类，如Food、EdibleThing、MealCourse和Shellfish等不属于葡萄酒的事物，但是如果要做有用的推理，则必须将它们与葡萄酒本体中的词汇相关联。为了满足我们识别葡萄酒/食物对的需求，食物和葡萄酒本体是彼此独立的。

一个类的定义由两部分组成：引入或引用一个名称，以及一个限制列表。被直接包含在类定义中的各个表达式进一步限制了该类的实例，该类的实例属于所有这些限制的交集。（这里描述的是成为某个类的必要条件，关于描述成为某个类的充分必要条件，请参见owl:equivalentClass部分。）到目前为止，我们所看到的例子均为只包含单个限制：强制被描述的新类为某个其它具名类（named class）的子类。

至此，我们可以为Wine类创建一个简单的（和不完整的）定义。Wine是一个PortableLiquid。同时，我们将Pasta定义为一个EdibleThing。

 <owl:Class rdf:ID="Wine">   
   <rdfs:subClassOf rdf:resource="&food;PotableLiquid"/>   
   <rdfs:label xml:lang="en">wine</rdfs:label>   
   <rdfs:label xml:lang="fr">vin</rdfs:label>   
   ...    
 </owl:Class>   
 <owl:Class rdf:ID="Pasta">  
   <rdfs:subClassOf rdf:resource="#EdibleThing" />  
   ...  
 </owl:Class>  
    rdfs:label是可选的，它为该类提供一个人类可读的名称。负责呈现的工具可以利用这个元素。“lang”属性为多语言提供了支持。一个label（标号）就像一个注释，不向本体的逻辑解释添加任何内容。

葡萄酒的定义仍然是不完整的。我们除了知道葡萄酒是一种事物并且适于饮用以外，对它别无所知。但我们有足够的信息来创建个体和对个体进行推理。

### 3.1.2. 个体

除了描述类，我们还希望能够描述类的成员。我们通常认为类的成员是我们所关心的范畴中的一个个体（而不是另一个类或属性）。要引入一个个体（individual），只需将它们声明为某个类的成员。

 <Region rdf:ID="CentralCoastRegion" />   
注意：下面代码的含义与上面的例子相同。

<owl:Thing rdf:ID="CentralCoastRegion" />   
 <owl:Thing rdf:about="#CentralCoastRegion">   
    <rdf:type rdf:resource="#Region"/>   
 </owl:Thing>  
    rdf:type是一个RDF属性（RDF property），用于关联一个个体和它所属的类。

这里有一些注意点。首先，我们已经决定CentralCoastRegion（一个特定的区域）是Region的成员。这里的Region类包含所有地理上的区域。其次，对于上述这一由两个元素构成的示例，并没有要求这两个陈述必须是相邻的、或必须位于同一文件中（尽管这些名字在这种情况下需要扩充一个URI）。Web本体被设计成为分布式的，我们可以通过导入和补充已有的本体来创建衍生的本体。

为了得到更多的类用于将在下一节引入的属性，我们定义了一个Grape（葡萄）的层次分类以及一个代表Cabernet Sauvignon品种的葡萄的个体。Grapes在食物本体中是这样定义的：

 <owl:Class rdf:ID="Grape">  
   ...  
 </owl:Class>  
接着，我们在葡萄酒本体中有：

 <owl:Class rdf:ID="WineGrape">  
   <rdfs:subClassOf rdf:resource="&food;Grape" />  
 </owl:Class>  
 <WineGrape rdf:ID="CabernetSauvignonGrape" />  
    正如下一节将要讨论的，CabernetSauvignonGrape是一个个体，因为它代表的是某个单个葡萄品种。

### 3.1.3. 使用方面的考虑

关于OWL中类与个体的区别，有一些重要的问题。一个类仅是一个名称和一些描述某集合内个体的属性；而个体是该集合的成员。因此，类应自然地对应于与某论域中的事物的出现集合，而个体应对应于可被归入这些类的实际的实体。

在构建本体时，这个区别常常变得模糊不清，主要有两种情况：

·          ***表示的层次：*** 在某些上下文中

-

，某些事物明显是一个类，但同时其本身又可被视为另一个事物的实例。例如：在葡萄酒本体中，我们有Grape的概念，它是代表所有葡萄品种的集合。CabernetSauvingonGrape是这个类中的一个实例，它代表Cabernet Sauvignon这一葡萄品种。但是，CabernetSauvignonGrape其自身也可被视为一个类，即代表所有实际的 Cabernet Sauvignon葡萄这一集合的类。

·           ***子类 vs. 实例：*** 实例（instance-of）关系和子类（subclass）关系很容易被混淆。例如，也许看上去可以随意地将CabernetSauvignonGrape作为Grape的一个实例，而不是作为Grape的一个子类，但实际上这个决定并不是随意的。Grape类代表的是所有葡萄品种的集合，因此任何Grape的子类应代表这些品种的一个子集。因而，CabernetSauvignonGrape应被认为是Grape的一个实例，而不是一个子类。因为CabernetSauvignonGrape是一个葡萄品种，而不是一个葡萄品种的子类。

注意：对于Wine类也同样有着上述问题。Wine类实际上代表的是所有葡萄酒种类的集合，而不是某人可以购买的瓶装葡萄酒的集合。设想在另一个本体中，各个Wine类的实例代表一个类，该类是某类葡萄酒的瓶装葡萄酒的集合。容易设想这样一个信息系统，例如一个葡萄酒商的库存系统，它需要处理各个瓶装葡萄酒。葡萄酒本体需要有把类作为实例处理的能力，以支持该解释。注意：OWL Full是允许这样表达的，这使得我们可以同时将一个葡萄酒品种的实例视为一个类，而该类的实例是瓶装葡萄酒。

同样的，葡萄酒厂在特定年份所生产的葡萄酒将被视为佳酒。为了表达佳酒这一概念，我们必须考虑将它置于当前本体中的何处。如前所述，一个Wine类的实例代表的是某葡萄酒厂所生产的某个单个葡萄酒种类，比如FormanChardonnay。

要表达“2000年生产的葡萄酒被视为佳酒”是有点复杂的，因为我们没有表达某种给定葡萄酒的个体的子集的能力。佳酒并不是一个新的葡萄酒种类，而是一个特殊的葡萄酒子集，即那些产于2000年的葡萄酒。一个方案是使用OWL Full，将Wine类的实例视为类，而后者的子类（子集）代表瓶装葡萄酒。另一个方案是使用变通手法，即将Vintage视为一个单独的类，Vintage的实例与代表它所属种类的Wine类相关联。例如，FormanChardonnay2000是一个Vintage类的个体，它通过vintageOf属性与Wine类的个体FormanChardonnay相关联。我们将在后面看到Vintage类的定义。

这里需要注意的一点是，一个本体的开发应坚定地由它的预定用途所驱动。这些问题也存在于OWL Full和OWL DL之间的一个重要区别。OWL Full允许将类（class）用作实例（instance），而OWL DL不允许。由于葡萄酒本体被预定为使用OWL DL，因此不会将个体（例如FormanChardonnay等）同时作为类来看待。

## 3.2. 简单属性

如果仅允许定义层次分类，那么这个类和个体的世界也许会变得颇为无趣。属性（propertyies）使得我们可以断言关于类成员的一般事实以及关于个体的具体事实。

### 3.2.1. 定义属性（Defining Properties）

**ObjectProperty, DatatypeProperty, rdfs:subPropertyOf, rdfs:domain, rdfs:range**

一个属性是一个二元关系。有两种类型的属性：

·         数据类型属性（datatype properties），类实例与RDF文字或XML Schema数据类型间的关系。

·         对象属性（object properties），两个类的实例间的关系。注意：对象属性这个名称并不是要反映与RDF（[RDF]，5.3.4）的联系。

在我们定义一个属性的时候，有一些对该二元关系施加限定的方法。我们可以指定定义域（domain）和值域（range）。可以将一个属性定义为某个已有属性的特殊化（子属性）。要进行更详细的限定也是可能的，我们将在后面对此作出介绍。  
 <owl:ObjectProperty rdf:ID="madeFromGrape">   
   <rdfs:domain rdf:resource="#Wine"/>  
   <rdfs:range rdf:resource="#WineGrape"/>   
 </owl:ObjectProperty>   
 <owl:ObjectProperty rdf:ID="course">  
   <rdfs:domain rdf:resource="#Meal" />  
   <rdfs:range rdf:resource="#MealCourse" />  
 </owl:ObjectProperty>  
    在OWL中，不含显式操作符的元素序列代表一个隐式的合取（conjunction）。属性madeFromGrape的定义域（domain）为Wine，且值域（range）为WineGrape。也就是说，它把Wine类的实例关联到WineGrape类的实例。为同一属性声明多个定义域表明该属性的定义域是所有这些类的交集（多个值域声明也类似这样）。

同样地，属性course将一个Meal与MealCourse相关联。

注意：OWL中值域和定义域信息的使用与程序设计语言中的类型信息有所不同。在程序设计中，类型被用来检查程序设计语言的一致性。而在OWL中，一个值域可被用来推断一个类型。比如，根据下面这段代码：

 <owl:Thing rdf:ID="LindemansBin65Chardonnay">  
   <madeFromGrape rdf:resource="#ChardonnayGrape" />  
 </owl:Thing>                                              ┐   
    我们可以推断出，LindemansBin65Chardonnay是一种葡萄酒，因为madeFromGrape的定义域为Wine。

属性也可以像类一样按照层次结构来组织。  
 <owl:Class rdf:ID="WineDescriptor" />  
 <owl:Class rdf:ID="WineColor">  
   <rdfs:subClassOf rdf:resource="#WineDescriptor" />  
   ...  
 </owl:Class>

 <owl:ObjectProperty rdf:ID="hasWineDescriptor">  
   <rdfs:domain rdf:resource="#Wine" />  
   <rdfs:range  rdf:resource="#WineDescriptor" />  
 </owl:ObjectProperty>  
 <owl:ObjectProperty rdf:ID="hasColor">  
   <rdfs:subPropertyOf rdf:resource="#hasWineDescriptor" />  
   <rdfs:range rdf:resource="#WineColor" />  
   ...  
 </owl:ObjectProperty>  
    WineDescriptor属性将葡萄酒（wine）与它们的颜色（color）和味觉成分（包括甜、浓、口味）相关联。hasColor是hasWineDescriptor的子属性，hasColor与hasWineDescriptor的不同在于它的值域被进一步限定为WineColor。rdfs:subPropertyOf关系表示：任何事物如果具有一个值为X的hasColor属性，那么它同时具有一个值为X的hasWineDescriptor属性。

下面，我们介绍locatedIn属性，它将事物和事物所在的地区相关联。

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

请注意是如何定义locateIn的定义域和值域的。该定义域定义允许任何事物被值域某个区域中，包括该区域自身。这一关系的传递的组合本质上构建了一个包含子区域和事物的地理网络。没有包含其他事物于其中的那些事物可以属于任意类,而包含其他事物或者区域的那些事物则必须是区域。

现在可以扩展Wine的定义来表达“一个葡萄酒是由至少一种WineGrape制成的”了。和属性定义一样，类定义也由多个隐含相联的部分组成。

<owl:Class rdf:ID="Wine">   
   <rdfs:subClassOf rdf:resource="&food;PotableLiquid"/>   
   <rdfs:subClassOf>  
   <owl:Restriction>   
      <owl:onProperty rdf:resource="#madeFromGrape"/>  
      <owl:minCardinality    rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>  
   </owl:Restriction>   
   </rdfs:subClassOf>  
     ...    
 </owl:Class>

上述被高亮的子类限定

     <owl:Restriction>   
       <owl:onProperty rdf:resource="#madeFromGrape"/>  
       <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>  
     </owl:Restriction>   
    定义了一个无名类（unnamed class），该无名类代表至少具有一个madeFromGrape属性的事物集合。我们称这些类为*匿名*类。在Wine类的定义中包含该限定表明属于Wine类的事物，也是该匿名类的成员。也就是说，任何葡萄酒都必须参与至少一个madeFromGrape关系。

现在，我们可以描述前面所提到的Vintage类了。  
<owl:Class rdf:ID="Vintage">   
   <rdfs:subClassOf>  
     <owl:Restriction>   
       <owl:onProperty rdf:resource="#vintageOf"/>  
       <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>  
     </owl:Restriction>  
   </rdfs:subClassOf>  
 </owl:Class>                                              ┐   
vintageOf属性将一个Vintage关联到Wine。

 <owl:ObjectProperty rdf:ID="vintageOf">  
   <rdfs:domain rdf:resource="#Vintage" />  
   <rdfs:range  rdf:resource="#Wine" />  
 </owl:ObjectProperty>                                      ┐   
我们将在下一节把Vintages关联到它们的生产年份。

### 3.2.2. 属性和数据类型

根据是将个体关联到个体、还是将个体关联到数据类型，我们可以区分两类属性：前者称为对象属性（object properties）；后者称为数据类型属性（datatype properties）。数据类型属性的值域范围是RDF文字或者是XML Schema数据类型中定义的那些简单类型（simple types）。

OWL使用XML Schema内嵌数据类型中的大部分。对这些数据类型的引用是通过对http://www.w3.org/2001/XMLSchema 这个URI引用进行的。下列数据类型是推荐在OWL中使用的：

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

上面的数据类型，连同rdfs:Literal构成了OWL的内嵌数据类型。所有的OWL推理机都应支持xsd:integer和xsd:string数据类型。

其他XML Schema内嵌数据类型可被在OWL Full中，但在OWL Semantics and Abstract Syntax中给出了一些警告。  
 <owl:Class rdf:ID="VintageYear" />  
 <owl:DatatypeProperty rdf:ID="yearValue">  
   <rdfs:domain rdf:resource="#VintageYear" />      
   <rdfs:range  rdf:resource="&xsd;positiveInteger"/>  
 </owl:DatatypeProperty>

yearValue属性将VintageYears与一个整数值相关联。我们将引入hasVintageYear属性，它将一个Vintage关联到一个VintageYear（如下）。

OWL Reference描述了owl:oneOf的使用、以及用rdf:List和rdf:rest来定义一个枚举数据类型，那里有个例子展示了如何构建一个值域为整数值列表{0, 15, 30, 40}的数据类型属性tennisGameScore。

### 3.2.3. 个体的属性

-

首先，我们描述Region和Winery个体，然后我们定义第一个葡萄酒Cabernet Sauvignon。

 <Region rdf:ID="SantaCruzMountainsRegion">  
   <locatedIn rdf:resource="#CaliforniaRegion" />  
 </Region>  
 <Winery rdf:ID="SantaCruzMountainVineyard" />  
 <CabernetSauvignon  
   rdf:ID="SantaCruzMountainVineyardCabernetSauvignon" >  
   <locatedIn   rdf:resource="#SantaCruzMountainsRegion"/>    
   <hasMaker    rdf:resource="#SantaCruzMountainVineyard" />     
 </CabernetSauvignon>    
    这仍然是不完整的。完整的本体中存在关于葡萄酒口味的其他方面的定义。但这些片断将会慢慢地揍齐为一个整体。我们不妨先开始推理葡萄酒将会与食物本体中的什么菜单项相伴。我们从上述定义得知，它将是Santa Cruz Mountain Vineyard。因为它是一种Cabernet Sauvignon（定义参见wine.rdf），我们知道它是一种干的（dry）、红色（red）葡萄酒（参见wine.rdf）。

可以用同样的方式给个体增加数据类型属性。下面，我们描述一个VintageYear的实例，并将它关联到一个特定的&xsd;positiveInteger类型的值。

 <VintageYear rdf:ID="Year1998">  
   <yearValue rdf:datatype="&xsd;positiveInteger">1998</yearValue>  
 </VintageYear>

## 3.3. 属性特性

接下来的几节描述了用以进一步说明属性的机制。我们可以对属性的特性进行详细的说明，这就提供了一种强有力的机制以增强对于一个属性的推理。

### 3.3.1. TransitiveProperty

如果一个属性P被声明为传递属性，那么对于任意的x,y和z：

P(x,y) 与 P(y,z) 蕴含 P(x,z)

属性locatedIn是传递属性。

<owl:ObjectProperty rdf:ID="locatedIn">

  <rdf:type rdf:resource="&owl;TransitiveProperty" />  
  <rdfs:domain rdf:resource="&owl;Thing" />  
  <rdfs:range rdf:resource="#Region" />  
</owl:ObjectProperty>

<Region rdf:ID="SantaCruzMountainsRegion">

  <locatedIn rdf:resource="#CaliforniaRegion" />  
</Region>

<Region rdf:ID="CaliforniaRegion">

  <locatedIn rdf:resource="#USRegion" />  
</Region>

因为圣克鲁斯山地区（SantaCruzMountainsRegion）位于（locatedIn）加利福尼亚地区（CaliforniaRegion），那么它也应该位于（locatedIn）美国地区（USRegion），因为属性locatedIn是传递属性。

### 3.3.2. SymmetricProperty

如果一个属性P被声明为对称属性，那么对于任意的x和y：

P(x,y)当且仅当P(y,x)

adjacentRegion属性是对称属性，而locatedIn属性则不是。更准确地说，locatedIn属性是没有被规定为对称属性。在当前的葡萄酒本体中没有任何限制，让它不能成为对称属性。

 <owl:ObjectProperty rdf:ID="adjacentRegion">  
   <rdf:type rdf:resource="&owl;SymmetricProperty" />  
   <rdfs:domain rdf:resource="#Region" />  
   <rdfs:range rdf:resource="#Region" />  
 </owl:ObjectProperty>  
 <Region rdf:ID="MendocinoRegion">  
   <locatedIn rdf:resource="#CaliforniaRegion" />  
   <adjacentRegion rdf:resource="#SonomaRegion" />  
 </Region>  
MendocinoRegion地区与SonomaRegion地区相邻，反过来也是这样。MendocinoRegion地区位于CaliforniaRegion地区，但是反过来并不成立。

### 3.3.3. FunctionalProperty

如果一个属性P被标记为函数型属性，那么对于所有的x, y, 和z:

P(x,y) 与P(x,z) 蕴含 y = z

在我们的葡萄酒本体中，hasVintageYear属性是函数型属性。一种葡萄酒有着一个特定的制造年份。也即，一个给定的Vintage个体只能使用hasVintageYear属性与单独一个年份相关联。owl:FunctionalProperty并不要求该属性的定义域的所有元素都有值。请参见关于Vintage的基数的讨论。

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

### 3.3.4. inverseOf

如果一个属性P1被标记为属性P2的逆（owl:inverseOf）, 那么对于所有的x 和 y:

P1(x,y) 当且仅当P2(y,x)

请注意owl:inverseOf的语法，它仅仅使用一个属性名作为参数。A 当且仅当B 意思是 (A 蕴含 B)并且(B蕴含A).

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

<owl:ObjectProperty rdf:ID="producesWine">  
   <owl:inverseOf rdf:resource="#hasMaker" />  
 </owl:ObjectProperty>

各种葡萄酒都有制造商，这些制造商在Wine类的定义中被限制为酿酒厂（Winery）。而每个酿酒厂生产的酒均以该酿酒厂为制造商。

### 3.3.5. InverseFunctionalProperty

如果一个属性P被标记为反函数型的（InverseFunctional），那么对于所有的x, y 和 z:

P(y,x) 与 P(z,x) 蕴含 y = z

我们可以注意到，在前面的章节中提到的producesWine属性是反函数型属性。因为一个函数型属性的逆必定是反函数型的。我们也能够如下定义hasMaker属性和 producesWine 属性以达到和前例中相同的效果。

<owl:ObjectProperty rdf:ID="hasMaker" />

<owl:ObjectProperty rdf:ID="producesWine">  
   <rdf:type rdf:resource="&owl;InverseFunctionalProperty" />  
   <owl:inverseOf rdf:resource="#hasMaker" />  
 </owl:ObjectProperty>  
反函数型属性的值域中的元素可以看成是在数据库意义上定义一个唯一的键值。owl:InverseFunctional意味着属性的值域中的元素为定义域中的每个元素提供了一个唯一的标识。

在OWL Full中，DatatypeProperty属性可以被声明为一个反函数型属性。这就使得我们能够使用一个字符串作为一个唯一的键值。在OWL DL中，文字（literal）与owl:Thing是不相交的，因此OWL DL不允许将InverseFunctional属性声明应用到DatatypeProperty属性。

## 3.4. 属性限制

除了能够指定属性特性，我们还能够使用多种方法进一步在一个明确的上下文中限制属性的值域。这是通过“属性限制”来完成的。下面描述的多种形式仅在owl:Restriction的上下文中才能使用。owl:onProperty元素指出了受限制的属性。

### 3.4.1. allValuesFrom, someValuesFrom

我们已经发现了一种限制组成属性的元素的类型的方法。到现在为止，我们所讲述的机制都是全局的（global），因为这些机制都会应用到属性的所有实例。而接下来要讲述的两个属性限制机制，allValuesFrom与 someValuesFrom，则是 局部的（local），它们仅仅在包含它们的类的定义中起作用。

owl:allValuesFrom属性限制要求：对于每一个有指定属性实例的类实例，该属性的值必须是由owl:allValuesFrom从句指定的类的成员。

 <owl:Class rdf:ID="Wine">  
   <rdfs:subClassOf rdf:resource="&food;PotableLiquid" />  
   ...  
   <rdfs:subClassOf>  
     <owl:Restriction>  
       <owl:onProperty rdf:resource="#hasMaker" />  
       <owl:allValuesFrom rdf:resource="#Winery" />  
     </owl:Restriction>  
   </rdfs:subClassOf>  
   ...  
 </owl:Class>  
    Wine的制造商必须是Winery。allValuesFrom限制仅仅应用在Wine的hasMaker 属性上。Cheese的制造商并不受这一局部限制的约束。

owl:someValuesFrom限制与之相似。在上个例子中，如果我们用owl:someValuesFrom替换owl:allValuesFrom，那就意味着至少有一个Wine类实例的hasMaker属性是指向一个Winery类的个体的。

<owl:Class rdf:ID="Wine">  
   <rdfs:subClassOf rdf:resource="&food;PotableLiquid" />  
   <rdfs:subClassOf>  
     <owl:Restriction>  
       <owl:onProperty rdf:resource="#hasMaker" />  
       <owl:someValuesFrom rdf:resource="#Winery" />  
     </owl:Restriction>  
   </rdfs:subClassOf>  
   ...  
 </owl:Class>   
这两种限制形式间的不同就是全称量词与存在量词间的不同。

关系

含意

allValuesFrom

对于所有的葡萄酒，如果它们有制造商，那么所有的制造商都是酿酒厂。

someValuesFrom

对于所有的葡萄酒，它们中至少有一个的制造商是酿酒厂。

前者并不要求一种葡萄酒一定要有一个制造商。如果它确实有一个或多个制造商，那么这些制造商必须全部都是酿酒厂。后者要求至少有一个制造商是酿酒厂，但是可以存在不是酿酒厂的制造商。

### 3.4.2. 基数限制

在前面我们已经看到过关于基数约束的例子了。到目前为止的例子中，这些约束都是关于最小基数的所作出的断言。更为直接的方法是使用owl:cardinality，这一约束允许对一个关系中的元素数目作出精确的限制。例如，我们可以将Vintage标识为恰好含有一个VintageYear的类。

 <owl:Class rdf:ID="Vintage">   
   <rdfs:subClassOf>  
     <owl:Restriction>  
       <owl:onProperty rdf:resource="#hasVintageYear"/>    
       <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>  
     </owl:Restriction>  
   </rdfs:subClassOf>  
 </owl:Class>  
    我们标识hasVintageYear属性为一个函数型属性，也即意味着每个Vintage有至多一个VintageYear。而如果对Vintage类的hasVintageYear属性使用基数限制则是对其作出了更强的断言，它表明了每个Vintage有恰好一个VintageYear。

值域限制在0和1的基数表达式(Cardinality expressions)是OWL Lite的一部分。这使得用户能够表示“至少一个”，“不超过一个”，和“恰好一个”这几种意思。OWL DL中还允许使用除0与1以外的正整数值。owl:maxCardinality能够用来指定一个上界。owl:minCardinality能够用来指定一个下界。使用二者的组合就能够将一个属性的基数限制为一个数值区间。

### 3.4.3. hasValue [OWL DL]

    hasValue 使得我们能够根据“特定的”属性值的存在来标识类。因此，一个个体只要至少有“一个”属性值等于hasValue的资源，这一个体就是该类的成员。

 <owl:Class rdf:ID="Burgundy">  
   ...  
   <rdfs:subClassOf>  
     <owl:Restriction>  
       <owl:onProperty rdf:resource="#hasSugar" />  
       <owl:hasValue rdf:resource="#Dry" />  
     </owl:Restriction>  
   </rdfs:subClassOf>  
 </owl:Class>  
    这里我们声明了所有的Burgundy酒都是干(dry)的酒。也即，它们的hasSugar属性必须至少有一个是值等于Dry（干的）。

    与allValuesFrom 和someValuesFrom类似，这是一个局部的限制。它仅仅对Burgundy类的hasSugar属性作出限制。

# 4. 本体映射

为了让本体发挥最大的作用，就需要让本体得到充分的共享。为了使得在开发本体时尽可能的节省人力，就需要使得开发出的本体能够被重用。更理想的情况是他们能够被组合使用。例如，你可能同时使用来自某一来源的日期本体(date ontology)和来自另一来源的物理位置本体(physical location ontology),并将位置(location)的概念加以扩展以包括这个位置所处在的时间段。

在开发一个本体的过程中，很多的精力都被投入到将类与属性联系起来以获取最大的意义的工作上去了,意识到这一点也是很重要的。我们希望对类成员作出的断言较为简单同时又要求有广泛的和有用的含意在里面。这也是在本体开发过程中最为困难的工作。如果你能够找到已经经过广泛使用和精炼的本体，那么采用它才有意义。  
多个本体的合并工作是非常具有挑战性的。为了维护其一致性,几乎必然需要工具的支持。

## 4.1. 类和属性之间的等价关系

**equivalentClass, equivalentProperty**

当我们要把一些本体组合在一起作为另一个新的本体的一部分时，能说明在一个本体中的某个类或者属性与另一个本体中的某个类或者属性是等价的，这往往很有用。在实际应用中我们这样做的时候要千万小心。因为如果要组合的那些本体是互相矛盾的（所有A的都是B的 与 A的并不全是B的），那么在组合得到的结果中就不会有满足条件的扩展（没有满足条件的个体或关系）了。

在食物本体中，我们现在想把在餐宴菜肴中对葡萄酒特点的描述与葡萄酒本体相联系起来。达到这一目的一种方法就是在食物本体中定义一个类(&food;Wine)，然后在葡萄酒本体中将一个已有的类声明为与这个类是等价的。

  <owl:Class rdf:ID="Wine">  
    <owl:equivalentClass rdf:resource="&vin;Wine"/>  
  </owl:Class>  
    属性owl:equivalentClass被用来表示两个类有着完全相同的实例。但我们要注意，在OWL DL中，类仅仅代表着个体的集合而不是个体本身。然而在OWL FULL中，我们能够使用owl:sameAs来表示两个类在各方面均完全一致。

当然了，上面我们举的例子多少有点刻意人为的意思，因为我们总是能在本来用＃Wine的地方使用&vin;Wine以取得同样的效果而不需要重新定义类。一种更可能出现的情况是：我们依赖两个独立开发的本体，并注意到他们使用了O1:foo和O2:bar这两个URI引用相同的一个类。 这时我们就能够使用owl:equivalentClass将这两个类关联起来， 使得从这两个本体中继承的限制也得到合并。

大家都知道，类名（类的表达式）既能用于<rdfs:subClassOf >设计中，又能用于<owl:equivalentClass>设计中。一个类名可多处使用，既省却了命名的麻烦，又给我们提供了基于属性要求的强大的定义能力。

 <owl:Class rdf:ID="TexasThings">   
   <owl:equivalentClass>  
     <owl:Restriction>  
       <owl:onProperty rdf:resource="#locatedIn" />  
       <owl:someValuesFrom rdf:resource="#TexasRegion" />  
     </owl:Restriction>  
   </owl:equivalentClass>  
 </owl:Class>  
    TexasThings指的是那些恰好位于Texas地区的事物。使用owl:equivalentClass 和使用rdfs:subClassOf 的不同就像必要条件和充要条件的不同一样。如果是使用subClassOf的话，位于Texas地区的事物不一定是TexasThings。但是，如果使用owl:equivalentClass，位于Texas地区的事物一定属于TexasThings类。

关系

蕴涵

subClassOf

TexasThings(x) implies?locatedIn(x,y) and TexasRegion(y)

equivalentClass

TexasThings(x) implies locatedIn(x,y) and TexasRegion(y)

    类似的，我们可以通过使用owl:equivalentProperty属性声明表达属性的等同。

## 4.2 个体间的同一性

**sameAs**

描述个体之间相同的机制与描述类之间的相同的机制类似，仅仅只要将两个个体声明成一致的就可以了。

例如这样一个例子：

 <Wine rdf:ID="MikesFavoriteWine>  
     <owl:sameAs rdf:resource="#StGenevieveTexasWhite" />  
 </Wine>  
 <Wine rdf:ID="MikesFavoriteWine">   
   <owl:sameAs rdf:resource="#StGenevieveTexasWhite" />   
 </Wine>  
    在这个例子并没有什么实际意义。所有我们从中能了解到的就是Mike喜欢一种便宜的本地酒。sameAs的一种更加典型的用法是将不同文档中定义的两个个体等同起来，作为统一两个本体的部分。

但这样做带来了一个问题。OWL中并没有名称唯一这一假定。仅仅名称不同并不意味着这两个名称引用的是不同的个体。

在上面的例子中，我们对两个截然不同的名称作出一致性断言。但是也只有在这种标示的情况下，才可能进行推理。请记住那些可能从函数型属性中得出的含意。假如hasMaker是一个函数型属性，那么下面的例子就不一定会产生冲突。

<owl:Thing rdf:about="#BancroftChardonnay">

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

除非和我们本体中的其他信息发生冲突，不然的话这样的描述是没有冲突的，他说明Bancroft和Beringer是相同的个体。

要清楚，修饰（或引用）两个类用sameAs还是用equivalentClass效果是不同的。用sameAs的时候，把一个类解释为一个个体，就像在OWL Full中一样，这有利于对本体进行分类。在OWL Full中，sameAs可以用来引用两个东西，如一个类和一个个体、一个类和一个属性等等，无论什么情况，都将被解释为个体。

## 4.3. 不同的个体

**differentFrom, AllDifferent**

这一机制提供了与sameAs相反的效果。

 <WineSugar rdf:ID="Dry" />  
 <WineSugar rdf:ID="Sweet">    
   <owl:differentFrom rdf:resource="#Dry"/>    
 </WineSugar>   
 <WineSugar rdf:ID="OffDry">  
   <owl:differentFrom rdf:resource="#Dry"/>   
   <owl:differentFrom rdf:resource="#Sweet"/>   
 </WineSugar>  
    这是一种声明这三个值相互不同的方法。但在有些时候，更重要的是利用这些定义元素能把这种不同区别开来。没有上述的定义元素，我们可能会定义一种既干又甜的葡萄酒，并且添加hasSugar属性使其取值小于等于某个值来限定该种葡萄酒的甜度。如果我们没有用 differentFrom元素来申明既干又甜的葡萄酒，这意味着“干葡萄酒”和“甜葡萄酒”是相同的。但是我们从上面申明的元素来推断，这又是矛盾的。

还有一种更便利的定义相互不同个体的机制。如下面申明红葡萄酒、白葡萄酒和玫瑰葡萄酒的例子。

 <owl:AllDifferent>  
   <owl:distinctMembers rdf:parseType="Collection">  
     <vin:WineColor rdf:about="#Red" />  
     <vin:WineColor rdf:about="#White" />  
     <vin:WineColor rdf:about="#Rose" />  
   </owl:distinctMembers>  
 </owl:AllDifferent>  
要注意，owl:distinctMembers属性声明只能和owl:AllDifferent属性声明一起结合使用。

    在葡萄酒本体中，我们为所有的WineDescriptor提供了一个owl:AllDifferent声明。我们同时还声明了所有的Winery是不同的。这时，如果我们想要在其他的某个本体中添加一个新的酿酒厂，并表明它是与其他已定义的任何酿酒厂都是不同的，我们可能需要拷贝原来的owl:AllDifferent属性声明，然后将新的制造厂添加到列表中。在OWL DL中，没有更加简单的方法以扩展一个声明为owl:AllDifferent的集合。而在OWL Full中，通过使用RDF三元组和rdf:List构造，可以实现一些其他的方法来完成这一扩展。

# 5. 复杂类 [OWL DL]

OWL另外还提供了一些用于构建类的构造子。这些构造子被用于创建所谓的类表达式。OWL支持基本的集合操作，即并，交和补运算。它们分别被命名为owl:unionOf,owl:intersectionOf,和owl:complementOf.此外，类还可以是枚举的。类的外延可以使用oneOf构造子来显示的声明。同时，我们也可以声明类的外延必须是互不相交的。

注意类表达式是可以嵌套的，它并不要求要为每一个中间类都提供一个名字。这样就允许我们通过使用集合操作来从匿名类或具有值约束的类来创建复合类。

## 5.1 集合运算符 intersectionOf,unionOf,complementOf

    记住:OWL类外延是由个体组成的集合，而这些个体都是类的成员。OWL使用基本的集合操作算子来处理类的外延。

### 5.1.1.交运算 [some uses of OWL DL]

下面的例子展示了intersectionOf构造子的使用

 <owl:Class rdf:ID="WhiteWine">  
   <owl:intersectionOf rdf:parseType="Collection">  
     <owl:Class rdf:about="#Wine" />  
     <owl:Restriction>  
       <owl:onProperty rdf:resource="#hasColor" />  
       <owl:hasValue rdf:resource="#White" />  
     </owl:Restriction>  
   </owl:intersectionOf>  
 </owl:Class>  
    使用集合操作构造的类与我们目前所看到的所有语法中的定义类似。类的成员完全是通过集合操作来说明的。上面的语

句说明WhiteWine恰好是类Wine与所有颜色是白色的事物的集合的交集。这就意味着如果某一事物是白色的并且是葡萄酒，那么它就是WhiteWine的实例。如果没有这样的定义我们只能知道白葡萄酒是葡萄酒酒并且是白色的，但是反过来就不是这样了。这是对个体进行分类的强有力工具。(请注意：''rdf:parseType="Collection"''是必需的语法元素。)

<owl:Class rdf:about="#Burgundy">  
   <owl:intersectionOf rdf:parseType="Collection">  
     <owl:Class rdf:about="#Wine" />  
     <owl:Restriction>  
       <owl:onProperty rdf:resource="#locatedIn" />  
       <owl:hasValue rdf:resource="#BourgogneRegion" />  
     </owl:Restriction>  
   </owl:intersectionOf>  
 </owl:Class>  
    在这里我们定义了Burgundy类。这个类恰好包含了那些至少有一个locatedIn关系，而同时这一关系又要联系到Bourgogne地区的葡萄酒。当然我们也可以声明一个新的类ThingsFromBourgogneRegion，并且将该类作为owl:intersectionOf结构中的类使用。但既然ThingsFromBourgogneRegion不再有其他用处，上面的声明就显得更加简短、清晰，并且这一声明不需要我们努力想一个新的名字出来。

<owl:Class rdf:ID="WhiteBurgundy">  
   <owl:intersectionOf rdf:parseType="Collection">  
     <owl:Class rdf:about="#Burgundy" />  
     <owl:Class rdf:about="#WhiteWine" />  
   </owl:intersectionOf>   
 </owl:Class>  
    最后，WhiteBurgundy类恰好是白葡萄酒和Burgundies的交集。依次，Burgundies生产在法国一个叫做Bourgogne的地方并且它是干葡萄酒（dry wine）。因此，所有满足这些标准的葡萄酒个体都是WhiteBurgundy类的外延的一部分。

### 5.1.2. 并运算 [OWL DL]

    下面的例子展示了unionOf结构的使用。它的使用方法和intersectionOf极其类似:

 <owl:Class rdf:ID="Fruit">  
   <owl:unionOf rdf:parseType="Collection">  
     <owl:Class rdf:about="#SweetFruit" />  
     <owl:Class rdf:about="#NonSweetFruit" />  
   </owl:unionOf>  
 </owl:Class>  
    Fruit类既包含了SweetFruit类的外延也包含了NonSweetFruit的外延。

请仔细观察这种并集类型的结构与下面的一个结构是多么的不同。

 <owl:Class rdf:ID="Fruit">  
   <rdfs:subClassOf rdf:resource="#SweetFruit" />  
   <rdfs:subClassOf rdf:resource="#NonSweetFruit" />  
 </owl:Class>  
    上面的例子说明Fruit的实例是SweetFruit和NonSweetFruit的交集的子集,这里我们将预计得到一个空集。

### 5.1.3. 补运算 [OWL DL]

    complementOf结构从某个论域(domain of discourse)选出不属于某个类的所有个体。通常它将指向一个非常大的个体集合：

  <owl:Class rdf:ID="ConsumableThing" />  
  <owl:Class rdf:ID="NonConsumableThing">  
    <owl:complementOf rdf:resource="#ConsumableThing" />  
  </owl:Class>  
    类NonConsumableThing包含了所有不属于ConsumableThing的外延的个体。NonConsumableThing集合包含了所有的Wines,Regions等。它实际上就是owl:Thing与ConsumableThing的这两个集合的集合差。因此，complementOf典型的用法是与其它集合运算符联合使用：

<owl:Class rdf:ID="NonFrenchWine">  
   <owl:intersectionOf rdf:parseType="Collection">  
     <owl:Class rdf:about="#Wine"/>  
     <owl:Class>  
       <owl:complementOf>  
         <owl:Restriction>  
           <owl:onProperty rdf:resource="#locatedIn" />  
           <owl:hasValue rdf:resource="#FrenchRegion" />  
         </owl:Restriction>  
       </owl:complementOf>  
     </owl:Class>  
   </owl:intersectionOf>  
 </owl:Class>  
    上面的例子定义了一个NonFrenchWine类，它是Wine类与所有不位于法国的事物的集合的交集。

5.2. 枚举类 oneOf [OWL DL]

    OWL提供了一种通过直接枚举类的成员的方法来描述类。这是通过使用oneOf结构来完成。特别地，这个定义完整地描述了类的外延，因此任何其他个体都不能被声明为属于这个类。

下面的例子定义了WineColor类，它的成员是White,Rose和Red这三个个体.

 <owl:Class rdf:ID="WineColor">  
   <rdfs:subClassOf rdf:resource="#WineDescriptor"/>  
   <owl:oneOf rdf:parseType="Collection">  
     <owl:Thing rdf:about="#White"/>  
     <owl:Thing rdf:about="#Rose"/>  
     <owl:Thing rdf:about="#Red"/>  
   </owl:oneOf>  
 </owl:Class>  
    看到上面的定义，第一件想到的事情就是由于这个类是通过枚举定义的，因此其他任何个体都不可能是一个有效的WineColor。

oneOf结构的每一个元素都必须是一个有效声明的个体。一个个体必须属于某个类。在上面的例子中，每一个个体都是通过名字来引用的。我们使用owl:Thing简单地进行引用，尽管这有点多余（因为每个个体都属于owl:Thing）。另外，我们也可以根据具体类型WineColor来引用集合中的元素：

 <owl:Class rdf:ID="WineColor">  
   <rdfs:subClassOf rdf:resource="#WineDescriptor"/>  
   <owl:oneOf rdf:parseType="Collection">  
     <WineColor rdf:about="#White" />  
     <WineColor rdf:about="#Rose" />  
     <WineColor rdf:about="#Red" />  
   </owl:oneOf>  
 </owl:Class>  
    另外，较复杂的个体描述同样也可以是oneOf结构的有效元素，例如:

<WineColor rdf:about="#White">  
   <rdfs:label>White</rdfs:label>  
 </WineColor>  
其它关于oneOf使用的例子，请参见 Reference.

## 5.3. 不相交类 disjointWith [OWL DL]

使用owl:disjointWith构造子可以表达一组类是不相交的。它保证了属于某一个类的个体不能同时又是另一个指定类的实例。

 <owl:Class rdf:ID="Pasta">  
   <rdfs:subClassOf rdf:resource="#EdibleThing"/>  
   <owl:disjointWith rdf:resource="#Meat"/>  
   <owl:disjointWith rdf:resource="#Fowl"/>  
   <owl:disjointWith rdf:resource="#Seafood"/>  
   <owl:disjointWith rdf:resource="#Dessert"/>  
   <owl:disjointWith rdf:resource="#Fruit"/>  
 </owl:Class>  
    Pasta例子声明了多个不相交类。注意它只声明了Pasta与其它所有类是不相交的。例如，它并没有保证Meat和Fruit是不相交的。为了声明一组类是互不相交的，我们必须对每两个类都使用owl:disjointWith来声明。

一个常见的需求是定义一个类为一组互不相交的子类的联合（union）。

 <owl:Class rdf:ID="SweetFruit">  
   <rdfs:subClassOf rdf:resource="#EdibleThing" />  
 </owl:Class>  
 <owl:Class rdf:ID="NonSweetFruit">  
   <rdfs:subClassOf rdf:resource="#EdibleThing" />  
   <owl:disjointWith rdf:resource="#SweetFruit" />  
 </owl:Class>  
 <owl:Class rdf:ID="Fruit">  
   <owl:unionOf rdf:parseType="Collection">  
     <owl:Class rdf:about="#SweetFruit" />  
     <owl:Class rdf:about="#NonSweetFruit" />  
   </owl:unionOf>  
 </owl:Class>  
    在上面个例子中，我们定义了Fruit是SweetFruit和NonSweetFruit的并集。而且我们知道这些子类恰好将Fruit划分成了连个截然不同的子类，因为它们是互不相交的。随着互不相交的类的增加，不相交的声明的数目也会相应的增加到n2.然而，在我们已知的用例中，n通常比较小。

当n很大时，我们可以使用另一些方法以避免声明的数目按二次方增长。其中一个方法在OWL test suite 有详细说明。

这一方法的工作原理如下。我们描述一个父类，它的元素有一个基数等于一的属性。接着，对于这个父类的每一个子类，我们都要求这个子类的实例的这一属性必须具有一个唯一的值。在这种情况下，各个不同子类就不可能有共同的成员了。

# 6. 本体的版本控制

本体和软件一样需要维护，因此它们将随着时间的推移而改变。在一个owl:Ontology元素（如上面讨论的）内，链接到一个以前定义的本体版本是可能的。属性owl:priorVersion被用来提供这种链接，并能用它跟踪一个本体的版本历史。

 <owl:Ontology rdf:about="">   
   ...  
   <owl:priorVersion rdf:resource="http://www.w3.org/TR/2003/CR-owl-guide-20030818/wine"/>   
   ...  
 </owl:Ontology>  
在上面例子中被指出的那个本体是被定义本体的一个以前版本。

本体版本可能彼此互不兼容，例如，一个本体以前的版本可能包含与现在版本中的陈述相矛盾的陈述。在一个owl:Ontology元素中，我们使用owl:backwardCompatibleWith和owl:incompatibleWith这些属性来指出本体版本是兼容还是不兼容以前的版本。如果没有进行owl:backwardCompatibleWith声明，那么我们假定就不存在兼容性。除了上面讲到的两个属性，还有一个属性owl:versionInfo适用与版本控制系统，它提供了一些相关信息（hook）。和前面三个属性相反的是，owl:versionInfo的客体是一个文字值（literal），这一属性除了可以用来注释本体之外还可以用来注释类和属性。

在许多时候，仅仅在整个本体的粒度上提供版本跟踪是不够的。维护人员可能希望能够记录类、属性、个体的版本信息——即使这些信息可能还是是不够充分。在OWL中，类表示的渐增性本质意味着了一个本体可以为一个在另一个本体中定义的（具名）类添加约束，而这些额外的约束本身可能需要版本信息。

OWL Full提供的表示能力能够对一个类进行任何类型的声明，也即可以声明一个类可以是另一个类的实例，或者一个类（不是它的实例）有一个属性和一个对应的属性值。这一框架就能被用来为版本跟踪信息建立一个由类和属性构成的本体。OWL的名称间中包括了两个预定义的类owl:DeprecatedClass和owl:DeprecatedProperty来完成这个目的。他们被用来指明某个类或属性在未来发布的版本中可能以一种不兼容的方式发生变化。

 ...  
   <owl:DeprecatedClass rdf:ID="&vin;JugWine" />  
   <owl:DeprecatedProperty rdf:ID="&vin;hasSeeds" />  
 ...  
    我们要注意到owl:DeprecatedClass及owl:DeprecatedProperty并没有附加的语义，这很重要。它们应由工具开发者和OWL使用者来确保这些属性是按其本意使用的。

# 7. 使用范例

一旦有某个初始的领域本体可以利用，人们就能够使用这个本体开发大量的应用。在本节中，我们会描述使用在葡萄酒领域的一些例子。

## 7.1.葡萄酒门户网站

现今有许多站点称自己是葡萄酒的门户网站。比如在Google上就可以找到152,000个与“wine portal”（葡萄酒门户网站）相匹配的结果。最匹配的是一个称为"Wine-Portal.com"的站点，该站点提供了问其他许多站点的途径。声称自己是葡萄酒门户网站的站点大多是信息式站点（informational sites）。例如，wine-portal.com中的第一个特色站点被称为“酒塞烹饪”（cork cuisine），这一站点就是提供关于葡萄酒和食物的搭配、关于以葡萄酒作为礼物等方面的信息的。

细读任何一个范围的主题，人们都会发现大量与该主题相关的内容，它们包含了与该主题相关的信息或者服务。像“附属品和礼物”（accessories and gifts）这个主题中就包含了关于购买特别的葡萄酒物品（wine item）时的注意事项的信息，同时也包含了众多的在线零售商的信息。另一个顶层的主题范围“购物”中包含了一个子范围“葡萄酒购买”（wine shopping），从这儿用户能够找到在线（或者“街道购物”（stree shopping））商店（按照国家分类）。这两个站点仅仅是现在许多例子中的两个，它们代表了这些门户网站的一般概念，也即葡萄酒门户为某个特定主题范围提供了大量的信息和服务。

即使在更细节的程度上考察这些站点，我们也无法得知现在它们在多大程度上依赖着本体。例如，我们从html的源代码中并无法找出它们使用本体的证据。然而，非常明显的，这些站点是有可能使用本体的并可以从中得到一些葡萄酒本体。

在门户网站中，本体的一个简单使用就是利用其进行组织和浏览。上面的类别列表就可能从葡萄酒的类别层次的最高几层生成。查询能够使用葡萄酒本体对与葡萄酒相关的信息进行检索。如果某人对本体中包含的术语进行搜索，查询就可能根据子类的信息进行扩展从而找到更多相关的答案。门户网站也可能使用（候选的）相关主题范围内的信息进行自我更新。使用强大的推理能力，它们甚至能够识别出可能的葡萄酒销售站点并通过协商将这些站点包含为门户网站的一部分。

## 7.2. 葡萄酒主体（agent）

为了说明的目的，我们启动了一个葡萄酒主体（wine agent）项目。按照我们最初的设计，葡萄酒主体的目标是为饮宴上的菜推荐合适品种的葡萄酒。该应用使用的本体就是作为本指南基础的这一葡萄酒本体。这一葡萄酒本体可以在DAML本体库中得到，名为wines.

一个个性化的葡萄酒主体能为人们够提供大量的服务。

主体能够根据给定的许多约束条件（例如要提供的餐宴）推荐合适的葡萄酒，能够找到关于某种特定的葡萄酒或者某种特定的葡萄酒的类别的信息，它还能够为一种葡萄酒找出合适的附属品（例如为某个品种的葡萄酒找到合适的特定种类的杯子等）。

接下来，我们用一个简单的原型系统来描述一个例子，该系统是作为一个学生作业项目来完成的。

考虑下面的情景：

某人正在筹划一个晚宴，其中至少有一个客人对酒了解甚深。主人希望能够使用与菜单上的菜肴最合适的酒来招待客人。主人也希望自己能够对晚宴的用酒显得学识渊博。并且，主人决定在晚宴上使用一种用番茄做成的意大利面条酱和新鲜的意大利面条作为主菜来招待客人。

为了能够提供与餐宴合适的葡萄酒，主人需要葡萄酒酒和食物搭配方面的知识。为了显得对葡萄酒有相当的了解，主人需要获得关于宴会用酒的有关信息。为了配上合适的附属品，主人还需要有关在该情况下（以及在主人认可价格范围内）的附属品的信息。

通过一个相关背景的葡萄酒本体，根据对餐宴的一定的描述，葡萄酒主体能够给出适合该餐宴的酒的类型的建议。葡萄酒主体可能会建议使用馨芳葡萄酒（Zinfandel）作为这次宴会的用酒类型。除此之外，通过相关背景的本体，葡萄酒主体可以给出建议使用某种特定的馨芳葡萄酒，比如玛丽埃塔馨芳葡萄酒（Marietta Zinfandel)。如果给定了用酒是馨芳葡萄酒的话，葡萄酒主体可能会去某个地点进行搜寻，它可能得到的是一系列的馨芳葡萄酒，也可能得到某种特定的馨芳葡萄酒，例如玛丽埃塔馨芳葡萄酒。如果本体中包含了购买葡萄酒的合适地点的信息（可能根据主人所在地以及葡萄酒销售商所在地信息进行了过滤），葡萄酒主体就可能访问一个站点如wine.com，在站点内搜索‘馨芳葡萄酒’并返回那个站点所销售的馨芳葡萄酒的列表。葡萄酒主体也可能从酿酒厂或者其他零售商处尝试寻找玛丽埃塔馨芳葡萄酒。例如，它可能发现（通过在Google进行搜索或者在某些选定站点上进行结构化的查询）在winelibrary.com上有1999年制造的玛丽埃塔馨芳葡萄酒正以13.99美元的折扣价进行销售。葡萄酒主体就可能使用附加的过滤信息进行过滤，这些过滤信息可能是由消费者提供的价格范围或者是对于不同品种的建议等。

葡萄酒主体现在就可能尝试要提供关于馨芳葡萄酒的通用信息或者玛丽埃塔馨芳葡萄酒的特定信息。它可能使用一个具有葡萄酒站点背景的本体来寻找关于特定的葡萄酒的信息。例如，它可能用到酿酒厂对它们最新的馨芳葡萄酒的描述这样的其他相关信息源的额外的评论也可能被用上。如果在某个最受欢迎的站点上没有关于玛丽埃塔馨芳葡萄酒的相关评论，在相同的地方找一些关于馨芳葡萄酒的评论可能也是有用的，比如在这里就可以找一些关于加利福尼亚Sonoma郡的馨芳葡萄酒的评论。

通用的背景信息可能也会使用到。主人可能需要做一些相关的阅读，他也可能对普通酒或者是馨芳葡萄酒的书籍感兴趣。例如，主人可能对Amazon.com上销售的馨芳葡萄酒的书籍感兴趣。主人也可能对相同地区的葡萄酒相关的信息感兴趣，这里可能就是Sonoma郡的馨芳葡萄酒。葡萄酒主体一般仅能得到它的主要知识领域相关的背景信息。比如，这个葡萄酒主体是关于食物和酒的匹配方面的，所以它可能通过免费的或者付费购买的方式得到的这一主题的信息，例如像评酒者上的关于搭配食物和酒的文章。

宴会的主人可能还要购买一些对于宴会活动来说很重要的酒的附属品。酒是使用酒杯作为容器的，而不同品种的酒最好使用不同种类的酒杯来装。例如，如果主人选择了一道菜是适合用馨芳葡萄酒来配的，主人可能就需要知道Riedel是一个知名的酒器生产商。主人可能还希望能够链接到Wine Enthusiast（一个相当受敬重的葡萄酒商品供应商）并告诉他Wine Enthusiast有一种Riedel生产的馨芳葡萄酒酒杯正在以63.95美元的价格四个一组地进行销售（如果你买两组四个的杯子的话就可以得到59.95美元一组的折扣价）。主人可能还有兴趣了解在Amazon.com上通过49.99美元(and claims a list price of $65.00)就可以购买得到的Reidel生产的Sommelier 馨芳葡萄酒单把杯（Reidel''s Sommelier Zinfandel single stem glass）。Amazon上面也有相同的6个一组的馨芳葡萄酒酒杯销售（而不像Wine Enthusiast上的4个一组），销售价是79.99美元（and claims a list price of $119.40)。葡萄酒主体能提供一个与餐宴搭配的（也即，与招待用的馨芳葡萄酒搭配的）酒器的列表给主人，然后进行价格的比较或者根据由本体中的一个属性列表中选出的其他标准来进行比较。

主人可能还要考虑其他的酒的附属品。从本体中我们可以了解到启瓶器是属于酒的附属品。背景本体可能已经对启瓶器的子类进行了编码或者这些信息可以通过相关的葡萄酒站点获得。Wine Enthusiast就有一系列它们推荐的（其中有关于启瓶器的类型和价格范围的描述）[1]。它们也根据启瓶器的不同类型（杠杆型、服务员型、固定型、旋转型、抽吸型）进行区分，而主人可能想要得到关于这些不同类型的启瓶器的信息。

根据不同领域的背景本体知识和不同的信息以及不同的服务站点，葡萄酒主体可能应用到不同复杂程度的情况中去。在本例中，我们仅仅考虑了关于酒、品种类型、食物和酒的组合、某些酒的附属品和它们相关属性的一些信息。当然，我们能够根据顾客要求将本例扩展以包含更多的信息和更多的约束。

葡萄酒主体的例子还在不断完善中，从这里我们可以得到它。