19. XML

结构化数据：有一个独立的模式来描述其结构。

- 优点：存储管理以及询问处理得以高效实现。

半结构化数据：自描述的，即数据本身包含着其结构模式的信息。

- 优点：增加新属性、新关系时的灵活性；模式可以是任意的（不仅可以在时间上，还可以在单个数据库中）。

半结构化数据模型：提供了灵活的描述真实世界的概念工具，可以用来整合多种多样的数据库，同样也是 Web 上分享信息的工具 XML 的底层模型。

图表示：一个半结构化数据的数据库是一系列的结点 (node)，结点在一个有根图结构中相连。

- 叶子 (leaf) 结点有相关的原子类型数据；
- 内部 (interior) 结点有边 (arc) 与其儿子结点相连；
- 根结点没有入边，表示整个数据库；且每个结点必须能从根结点经过若干边到达。
- 边上的标签指示着边的终点与起点的依赖关系。
    - 表示起点的属性，或者是起点与终点的联系。

应用：信息整合。

- 数据可能在许多地方存储，但是需要可以通过一个渠道访问，外部看上去像一个数据库。
    - 创造一个新数据库——开销巨大；整合已有的多种数据库！
- 同时，原始数据库还遗留着许多在其上开发的应用，这些原数据库不能被停止使用。
    - 使用半结构化数据，用一个统一的接口来处理新数据库上的问题；同时原有数据库仍然支持基于其上开发的应用。

XML：可扩展标记语言 (extensible markup langauge)，初始被设计用来标记文档，我们这里将其看作数据模型。

- HTML vs XML
    - HTML 用 tags 来进行展示（格式化），如 `italic`；
    - XML 用 tags 进行语义分割，如 `address`；
- XML 相当于用线性形式存储了半结构化数据的图结构，tags类似于图上的边。
- **语义标签 tags**：

    ```
    <tagname> </tagname>
    ```

    - 以 `<tagname>` 起始，以 `</tagment>` 终止。
    - **元素**：配对的 tags 以及其中的所有内容，比如：

        ```
        <FOO> Any text here. </FOO>
        ```

        - 空元素：`<FOO/>`。
    - XML 是大小写敏感的。

XML 的两种模式：

- **Well-formed XML**：没有预定义的结构，文档可以使用任何 tags，与半结构化数据相近。
    - 最低要求：文档以 XML 声明开始，并且有一个根元素 root 为整个文档的主体。

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<root-tag>
  ...
</root-tag>
```

    - `standalone="yes"` 表示没有 DTD。
    - 每个属性既可以当做子元素存在元素内部，也可以在 tags 的开始处添加并作为属性保存（即 name-value 对）。在这里参数并不是元素内容的一部分，而是标记的一部分，不过对于数据库来说没有区别。如下所示：

```
<Student sno="007", name="James Bond">
  <CNO>CS123</CNO>
  <CNO>CS456</CNO>
</Student>
```

    - 添加不在树结构中的关系：可以使用 `ID` 属性 (primary key) 与 `IDREF` 属性 (foreign key) 进行联系，比如：

```
<Student sno="007" taking="CS123 CS456">
  <Name>James Bond</Name>
</Student>
<Course cno="cs123" takenBy="007">
  <Title>Database systems</Title>
</Course>
```

- **Valid XML**：在 DTD (Document Type Definition) 中预定义允许的 tags 以及他们之间的嵌套关系。这个模式是介于半结构化数据与严格模式之间的一个模式。

XML 与 DB 的关系

- 可以用来进行数据交换；
- 也可以用来存储具有严格模式的大批量数据。
- 如何处理?
    - 开发专用 XML 的 DBMS —— 市场不认同。
    - 使用 RDB。

在 RDB 中存储 XML

- 方式 1: `Documents(docID, strXML)`；
- 方式 2: `DocRoot(docID, rootElementID)`, `SubElement(parentID, childID, position)`,

`ElementAttribute(elementID, name, value)`，`ElementValue(elementID, value)`；

- 方式 3: SQL 2003+ 提出了 xml 类型。

文档类型定义 DTD：DTD 建立了一个对于元素的直观视图。其中定义了一系列规则描述：包括什么 tags 可以在文档中出现，以及tags 可以如何嵌套。

```
<!DOCTYPE root-tag [
  <!ELEMENT name (components)>
  more elements ...
]>
```

- 使用的 `root-tag` 应该与后续的 tags 名称对应。
- 一个元素用其名称 `tag` 与括号表示。
  - 包括了其子元素的顺序以及数量要求；
    - `*`：0 个以上；
    - `+`：1 个以上；
    - `?`：0 个或 1 个。
  - 叶子结点使用 `#PCDATA` 作为成员；
  - 使用 `|` 表示 “或者”，比如：`(#PCDATA | (STREET CITY))`。
  - 特别地，`EMPTY` 表示这个元素没有任何子元素。
  - 例：

    ```
    <!ELEMENT NAME (
      (TITLE?, FIRST, LAST) | IPADDR
    )>
    ```

- 例：

```
<!DOCTYPE STUDENTS [
    <!ELEMENT STUDENTS (STUDENT+)>
    <!ELEMENT STUDENT (SNO, NAME, CNO*)>
    <!ELEMENT SNO (#PCDATA)>
    <!ELEMENT NAME (#PCDATA)>
    <!ELEMENT CNO (#PCDATA)>
]>
```

- **使用 DTD**：使用 DTD 时需要设置 `standalone="no"`，除此之外，还需要满足如下条件：
  - 要么将 DTD 作为文档的导言存储在文档开头；
  - 要么使用 `<!DOCTYPE root-type SYSTEM "xxx.dtd">` 导入外部 DTD 文件。
- **在 DTD 中声明属性**：`<!ATTLIST E A T V>` 声明了元素 `E` 的属性 `A` 是类型 `T` 的，同时有缺省值 `V`。
  - 常见类型：`CDATA`，`enumerations`，`ID`，`IDREF`，`IDREFS`；
  - 默认值可以是 `"def_value"`，`#REQUIRED`，`#IMPLIED` 或 `#FIXED "fixed_value"`。

- 可以在一个 ATTLIST 语句定义许多属性，不过这不是一个很好的代码风格。
- 例：

```
<!ELEMENT STUDENT EMPTY>
<!ATTLIST STUDENT SNO CDATA #REQUIRED>
<!ATTLIST STUDENT NAME CDATA #REQUIRED>
<!ATTLIST STUDENT AGE CDATA #IMPLIED>
<!ATTLIST STUDENT DEPT (CS | AUTO | EE) "CS">
```

使用如下：

```
<STUDENT SNO = "007"
         NAME = "James Bond"
         DEPT = "CS" />
<STUDENT SNO = "008"
         NAME = "Stephen Chow"
         AGE = "47"
         DEPT = "EE" />
```

- **ID 与 IDREF 属性**：这两个属性允许从一个对象指向另一个对象的指针，允许 XML 文档形成一个图，而不仅仅是一棵树。一个 `ID` 类型的数据可以用来给元素一个唯一的的标识；一个 `IDREF` 类型的数据可以通过 `ID` 指向某个元素。（`IDREFS` 允许指向多个元素）

  例：

```
<!DOCTYPE UNIVERSITY [
    <!ELEMENT UNIVERSITY (STUDENT*,COURSE*)>
    <!ELEMENT STUDENT (NAME)>
        <!ATTLIST STUDENT SNO ID #REQUIRED>
       <!ATTLIST STUDENT TAKES IDREFS #IMPLIED>
    <!ELEMENT NAME (#PCDATA)>
    <!ELEMENT COURSE (TITLE)>
        <!ATTLIST COURSE CNO ID #REQUIRED)>
    <!ELEMENT TITLE (#PCDATA)>
]>
```

```
<?xml version = "1.0" standalone = "no"?>
<!DOCTYPE UNIVERSITY SYSTEM "univ.dtd">
<UNIVERSITY>
    <STUDENT SNO = "007"
             TAKES = "CS123 CS456">
      <NAME>James Bond</NAME></STUDENT>
   <STUDENT SNO = "008">
      <NAME>Stephen Chow</NAME></STUDENT>
    <COURSE CNO = "CS123">
      <TITLE>DB</TITLE></COURSE>
    <COURSE CNO = "CS456">
```

```
      <TITLE>OS</TITLE></COURSE>
</UNIVERSTIY>
```

XML 模式定义 (XML Schema Definition, XSD)：描述 XML 文档模式的更有力工具；XSD 本身也是 XML 文档，通过如下引用：

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  ...
</xs:schema>
```

- **元素定义**：使用 `xs:element` 定义元素，元素含有参数：
  - `name`：tag 名称；
  - `type`：元素类型，可以是 XSD 内置类型或文档中定义的类型。

  例：

  ```
  <xs:element name="NAME" type="xs:string" />
  ```

- **复杂类型**：使用 `xs:complexType` 描述包括子元素的元素，其中 `name` 命名了这个类型。一个典型的复杂类型是 `xs:sqeuence`，其中有一系列的 `xs:element` 子元素，可以使用 `minOccurs` 与 `maxOccurs` 属性控制每个元素出现的次数的最小值与最大值。例如：

  ```
  <xs:complexType name = "studentType">
    <xs:sequence>
      <xs:element name = "SNO" type = "xs:string"
       minOccurs = "1" maxOccurs = "1" />
      <xs:element name = "NAME" type = "xs:string"
       minOccurs = "1" maxOccurs = "unbounded"/>
      <xs:element name = "AGE" type = "xs:integer"
       minOccurs = "0" maxOccurs = "1" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="STUDENT" type="studentType" />
  ```

  则在 XML 中使用如下：

```
<STUDENT>
  <SNO>007</SNO>
  <NAME>James Bond</NAME>
</STUDENT>
<STUDENT>
  <SNO>008</SNO>
  <NAME>Stephen Chow</NAME>
  <NAME>Zhou Xingxing</NAME>
  <AGE>47</AGE>
</STUDENT>
```

- **参数定义**：使用复杂类型中的 `xs:attribute` 定义该类型元素的参数。参数有如下属性：`name` 参数名称；`type` 参数类型；`default` 默认值；`use` 有 `"required"` 与 `"optional"`，分别表示必须的与可选的。例如：

```
<xs:complexType name = "studentType">
  <xs:attribute name = "SNO" type = "xs:string"
    use = "required" />
  <xs:attribute name = "NAME" type = "xs:string"
    use = "optional" />
  <xs:attribute name = "AGE" type = "xs:integer"
    default = "18" />
</xs:complexType>
<xs:element name="STUDENT" type="studentType" />
```

则可以在 XML 中使用如下：

```
<STUDENT SNO = "007" NAME = "James Bond" />
```

- **简单限制类型**：
  - **简单类型**：`xs:simpleType` 用来描述枚举类型以及区间限制类型的数据，有属性 `name`；
  - **限制类型**：`xs:restriction` 用作一个简单类型的子元素，属性 `base` 表示简单类型的 `type` 约束。
  - 此外，`xs:{min|max}{Inclusive|Exclusive}` 是四个带有 `value` 属性的元素，给了简单限制类型的上下界。
  - `xs:enumeration` 也可以用作简单类型的子元素，属性 `value` 给出了枚举的值。

例如：$[1, 180)$ 的整数。

```
<xs:simpleType name = "ageType">
  <xs:restriction base = "xs:integer" />
    <xs:minInclusive value = "1"/>
    <xs:maxExclusive value = "180"/>
  </xs:restriction>
</xs:simpleType>
```

例如：枚举类型。

```
<xs:simpleType name = "degree">
  <xs:restriction base = "xs:string">
    <xs:enumeration value = "bachelor"/>
    <xs:enumeration value = "master"/>
    <xs:enumeration value = "doctorate"/>
  </xs:restriction>
</xs:simpleType>
```

- **键值**：`xs:element` 可以有 `xs:key` 子元素表示该元素为当前元素包含范围内的键值。其含有子元素 `xs:selector` 与 `xs:field`，这两个子元素分别具有属性 `xpath`，表示其生效范围。具体来说，表示 `xs:selector` 元素中，`xs:field` 是一个键值。例如：

```
<xs:element name = "STUDENTS" … >
    . . .
  <xs:key name = "studKey">
    <xs:selector xpath = "STUDENT" />
    <xs:field xpath = "SNO" />
  </xs:key>
    . . .
</xs:element>
```

- **外键**：类似键值可以定义外键 `xs:keyref`

```
<xs:keyref name = FKname refer = Keyname>
    <xs:selector xpath = pathexp1>
    <xs:field xpath = pathexp2>
</xs:keyref>
```

  - `FKname` 为外键值的名称，`refer` 为引用的键值名称；
  - `pathexp1` 表示外键定义在哪个元素中，`pathexp2` 表示引用的键值具体元素。

  例如：

```
<xs:element name = "STUDENTS" ... >
  ...
  <xs:keyref name = "cRef" refers = "cKey"
    <xs:selector xpath = "STUDENT/TAKES" />
    <xs:field xpath = "@cno" />
  </xs:keyref>
  ...
</xs:element>
```

20. XPath 与 XQuery

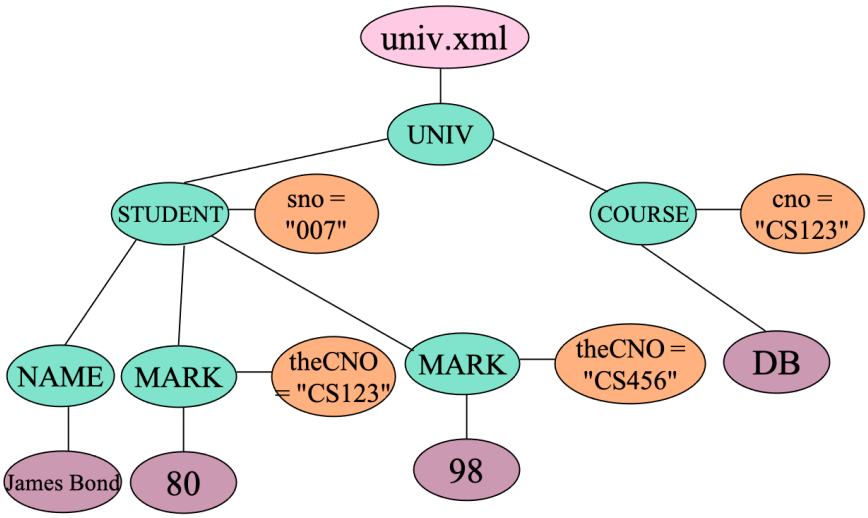
20.1 XPath

XPath 数据模型

- 将 XML 文档看作树状结构；
- XPath 是一个描述 XML 文档中路径的语言；
- XPath 的描述结果是一系列项目，每个项目要么是一个原始类型的值，要么是一个结点。

结点

- 文档首先是一个结点，表示整个 XML 文档，以 `doc(filename)` 表示，如 `doc("/mydir/univ.xml")`。
- 元素是一个结点；
- 元素的属性也是一个结点。



（绝对）路径表达式：以 `/` 开始，并在每项之间使用 `/` 分隔，如 `/UNIV/STUDENT/MARK`。

- 构造路径表达式：从上往下得到每个路径经过的节点，并且重复这个过程直到到达最终结点。
- 路径中的属性：可以在路径中出现属性，只需要在属性名之前加 `@` 标识符即可，如 `/UNIV/STUDENT/MARK/@theCNO`
  - 属性与元素的区别：路径结尾为属性则只会有属性值，为元素则会有元素的完整定义。

相对路径表达式：以当前目录开始，直接继续向下走即可。

可以以任意位置开始：在任何时候使用 `//x`，表示第一步可以在当前元素中的任意位置开始，只要下一步的 tag 为 `x` 即可。

- 找树中所有的 `NAME` 元素，不管在哪一层：`/UNIV//NAME`。

轴：在每一步中，我们可以沿着某个轴走。

- 默认的轴是 `child::`，即走向任何当前结点集合的子结点，简记为 `/`。
  - `/UNIV/STUDENT` 实际上是 `/child::UNIV/child::STUDENT/`；
- 属性轴是 `attribute::`，即走向当前结点的属性结点，简记为 `@`。
  - `/UNIV/STUDENT/@sno` 实际上是 `/child::UNIV/child::STUDENT/attribute::sno`。
- 另外一些有用的轴包括 `parent::`（`..`）、自身（`.`）、自身或子孙结点（`//`）以及祖先、自身或祖先、下一个兄弟结点（子结点有序）等等。

通配符 `*`：使用 `*` 在 tag 的位置表示“任意tag”，`@*` 表示任意属性。

条件选择：使用 `tag[condition]` 选出满足条件的 `tag` 元素。

- 仅选出满足条件 `condition` 的元素。
- 两个序列的条件比较：两个序列中各存在一个元素，这两个元素满足条件，则两个序列满足条件。
- 例：对于如下数据库可以写出如下带条件选择的路径。

```
<UNIV>
  <STUDENT sno = "007">
    <NAME>James Bond</NAME>
    <MARK theCNO = "CS123">80</MARK>
    <MARK theCNO = "CS456">98</MARK>
  </STUDENT> …
  <COURSE cno = "CS123"> DB</COURSE>…
</UNIV>
```

```
/UNIV/STUDENT/MARK[. < 90]
/UNIV/STUDENT/MARK[@theCNO = "CS123"]
```

- 其他条件：
  - `X[i]` 为真表示 `X` 是其父母的第 i 个儿子；
  - `X[T]` 为真表示 `X` 存在类型为 `T` 的子元素；
  - `X[A]` 为真表示 `X` 的 `A` 属性存在值。

20.2 XQuery

XQuery：是 XPath 的扩展，一个对 XML 数据进行查询的语言。

- 函数式语言（表达式语言）；
- 所有通过 XQuery 产生的表达式结果都是项目序列 (sequence of items)。
  - sequence 需要被抹平（消除层次结构），即 `(1 2 () (3, 4)) = (1, 2, 3, 4)`。

For-Let-Where-Return (FLWR) 表达式

- 以 0 个或多个 `for (and/or) let` 语句开始，可以嵌套；然后一个可选的 `where` 语句；最后恰好一个 `return` 语句。
  - 每个 `for` 将会创造一个循环；每个 `let` 将会定义临时变量，在每次循环时将会判断是否满足 `where` 表达式，如果满足则调用 `return` 表达式，将其值输出。
- **for 语句**：循环

```
for var in exp, ...
```

  - 变量以 `$` 开始；将会遍历 `exp` 表达式生成的结果列表的全部项，并对每项执行恰好一次 `for` 后的内容。
  - 例：返回 xml 文档的所有课号。

```
for $c in doc("univ.xml")/UNIV/COURSE/@cno
return <CNO> {$c} </CNO>
```

    - 这里的花括号表示取变量值，如果不加花括号表示取内容。
    - 而且必须返回一个 tag 的字符串，如果需要返回 untag 的内容需要加引号。

- **let 语句**：定义临时变量

```
let var1 := expression1,
    var2 := expression2, ...
```

  通常 `let` 一定可以省略（只是会复杂化语句）。

- **where 语句**：与 SQL 相似，条件将对序列中的每一个 item 进行测试，对符合条件的进行 return 输出；

```
where condition
```

- **order by 语句**：将结果排序。

```
order by expression
```

    - 可以加入可选的 `ascending` 与 `descending`。

  例：对 CS123 的所有成绩排序。

```
let $d := document("univ.xml")
for $p in $d/UNIV/STUDENT/MARK[@theCNO="CS123"]
order by $p
return $p
```

- **return 语句**

```
return expression
```

    - 输出满足条件的表达式；
    - 可以执行许多次，结果可能需要多次才能产生完整；
    - return 并不会结束查询。

基于序列的比较：只需要有一个 item 结果为真即可。例如：对如下 XML 文档撰写 XQuery 查询。

```
<T>
  <N>xxx</N>
  <A>
    <B>123</B>
    <C>abc</C>
  </A>
  <A>
    <B>456</B>
    <C>def</C>
  </A>
</T>
```

```
for $v in ../T
where $v/A/B = "123" and $v/A/C = "def"
return $v/N
```

严格比较：元素与单个 item 进行比较，可以使用 `eq, ne, lt, le, gt, ge` 表示严格比较。

- 元素与值的比较实际上比较的是其中的值；
- 元素与元素的比较比较的是他们的 ID，只有他们是同一个元素才相同！
  - 如果需要比较元素的值，使用 `data(E)` 提取元素 `E` 的值。

去重：使用 `return distinct-values(...)` 返回去重后的结果。

- 注意，这个函数将会去掉元素的 tag，比较元素的值；但是在结果中并不会重新存储 tag。

量词表达式（∀, ∃）

```
every $x in E1 satisfies E2
some $x in E1 satisfies E2
```

聚合表达式：直接取序列的聚合值即可，如

```
let $d := doc("univ.xml")
for $s in $d/UNIV/STUDENT
where count($s/MARK) > 100
return $s
```

条件表达式

```
if (E1) then E2 else E3
```

例：

```
if ($x/@sno eq "007") then $x/NAME else ()
```