基于 Web 的鱼类检索系统

项目书 v1.1

赵杰

2010/8/24

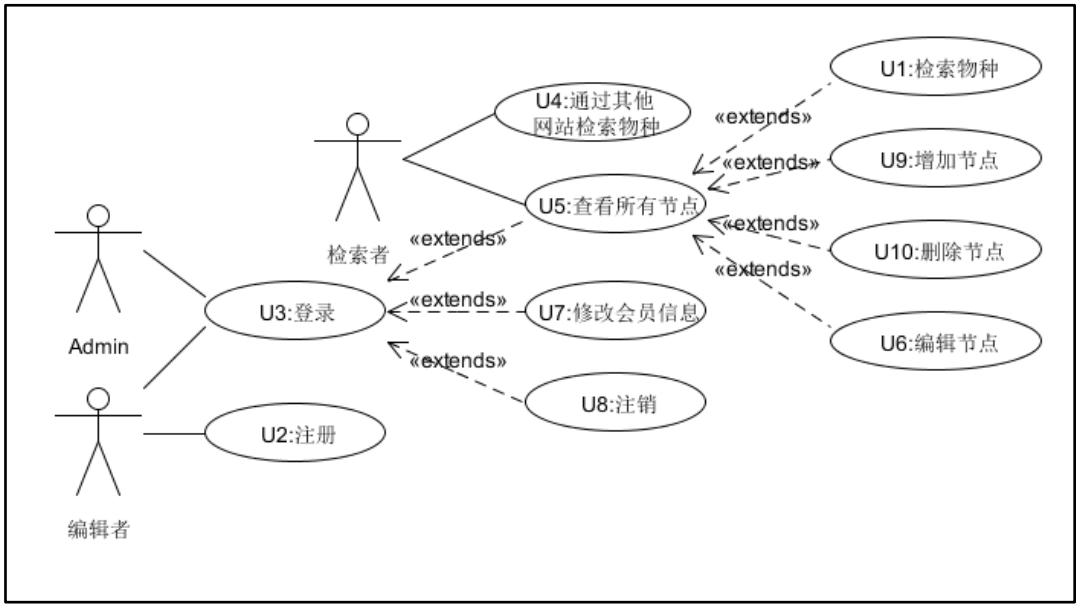
版本历史

版本/状态	作者	完成日期	备注
V1.0	赵杰	2010 年 8 月 24 日	起草
V1.1	赵杰	2010 年 10 月 6 日	添加服务调用接口，修正部分设计细节

目　录

第 1 章 需求分析

1.1 用例图



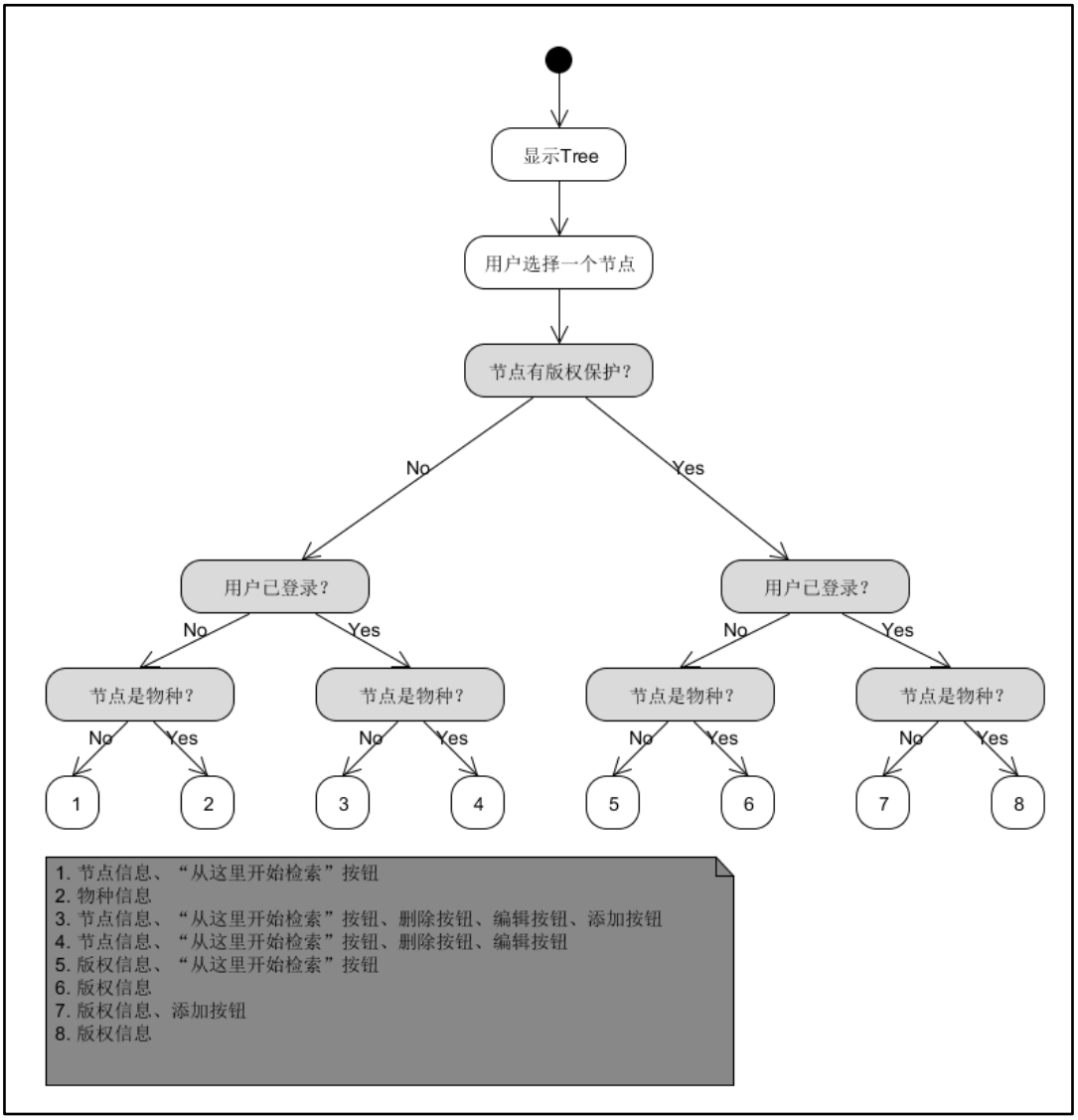
工程文件：Fish_UseCase.uxf

1.2 用例描述

U1：检索物种（扩展 U5）
前置条件：使用者已选择从某个节点开始检索
1．屏幕上出现帮助检索的一个问题
2．使用者对问题作出回答（Yes/No）
4．系统根据问答情况作出下一步动作（继续问答/给出检索结果）
5．若节点有版权保护，不予显示但给出获取查看权限的方式
后置条件：无

U5：查看所有节点（扩展 U3，被 U1、U9、U10、U6 扩展，因流程复杂，参考 U5 流程图）
前置条件：无
1．以树形结构显示数据库中的节点，一开始树是收缩的
2．用户点击一个节点后，展开该节点的下一层并显示“从这里开始检索”按钮，若用户已经登录，还应该显示“删除”、“添加”、“编辑”按钮
3．若此时用户点击“删除”按钮，用 U10 扩展，若点击“添加”按钮，用 U9 扩展，若点击“编辑”按钮，用 U6 扩展，若点击“从这里开始检索”按钮，用 U1 扩展
4．若用户点击下一层中的一个物种，显示该物种的信息，若用户已登录，显示“删除”、“添加”、“编辑”按钮

5．点击按钮的扩展方式同步骤 3
6．若用户点击下一层中的一个节点，重复步骤 2
后置条件：无



工程文件：U5.uxf

U4：通过其他网站检索物种
前置条件：无
1．用户输入一串可以唯一鉴定物种的字符串
2．选择第三方网站
3．系统给出检索结果
后置条件：无

U3：登录（被 U5、U7、U8 扩展）
前置条件：无
1．输入用户名
2．输入密码
3．点击“登录”按钮
异常路经：

a1．用户名或密码错误不能登录
后置条件：用户登录

U9：增加节点（扩展 U5）
前置条件：用户已登录并已选择一个节点
1．输入物种相关信息
2．可以选择是否启用“版权保护”
3．提交
后置条件：无

U10：删除节点（扩展 U5）
前置条件：用户已登录并已选择一个节点
1．提示用户是否真的要删除
2．根据用户回应删除或不删除
异常路经：
a1．若该节点有版权，而版权者和登录用户不是同一个，提示权限不够
后置条件：无

U6：编辑节点
前置条件：用户已登录并已选择一个节点
1．输入更新信息
2．更新
异常路经：
a1．若该节点有版权，而版权者和登录用户不是同一个，提示权限不够
后置条件：无

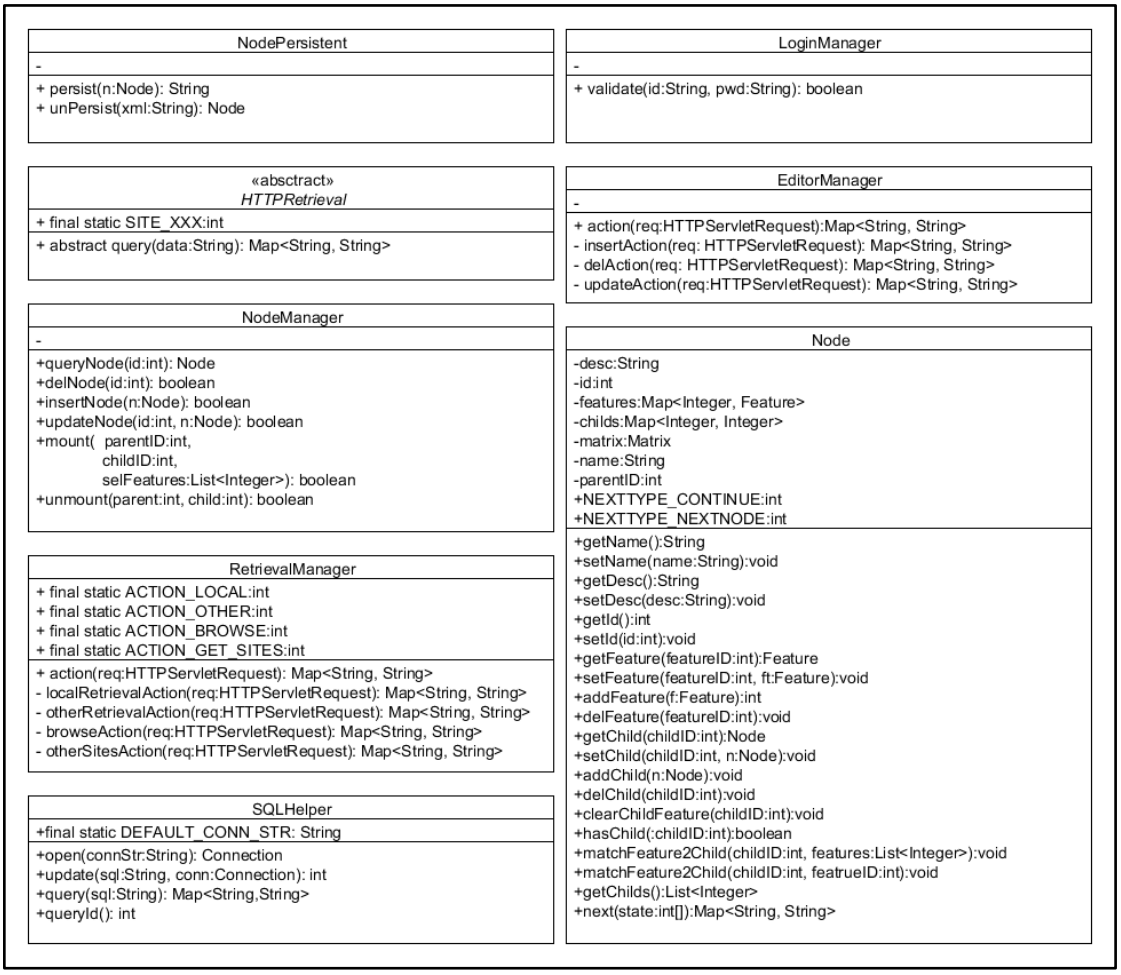
U2：注册
前置条件：无
1．输入用户信息
2．注册
后置条件：无

U7：修改会员信息（扩展 U3）
前置条件：用户已登录
1．输入新信息
2．更新
后置条件：无

U8：注销（扩展 U3）
前置条件：用户已登录
1．注销
后置条件：用户注销

第 2 章 类设计

2.1 类图



工程文件：Fish_Classes.uxf

2.2 类成员描述

2.2.1 Node 类

```
// 对节点的抽象，包含了节点的基本数据及修改这些数据的方法
class Node
```

```
- desc: String
- nodeID: int
- features: Map<Integer, Feature>
```

```
- childs: Map<Integer, Integer>
- matrix: Matrix
- name: String
- parentID: int
+ NEXTTYPE_CONTINUE:int
+ NEXTTYPE_NEXTNODE:int
```

```
// 返回节点的名字，如XX 纲
+ getName(): String
// 设置节点的名字，如XX 纲
+ setName(name:String): void
// 返回节点的描述
+ getDesc(): String
// 设置节点的描述
+ setDesc(desc:String): void
// 返回节点在数据库中的唯一标识
+ getID(): int
// 设置节点的标识，id 参数不可随意自定
// 使用此方法前必须调用NodeManager
// 的createID 方法获得一个合法标识
// 然后再调用setID 进行设置
+ setID(id:int): void
// 返回前继节点的标识
+ getParentID(): int
// 设置前继节点的标识
+ setParentID(id:int): void
// 返回指定id 的特性，如果没有id 对应的特性，返回null
+ getFeature(featureID:int): Feature
// 将id 为featureID 的特性设置为ft，
// 如果没有id 对应的特性，抛出RuntimeException
+ setFeature(featureID:int, ft:Feature): void
// 增加一个特性，并返回为此特性设定的id 号
+ addFeature(f:Feature): int
// 删除id 为featureID 的特性，如果没有id 对应的特性
// 抛出RuntimeException
+ delFeature(featureID:int): void
// 返回指定标识的子节点，如果没有id 对应的节点，返回null
+ getChild(childID:int): Node
// 将id 为childID 的节点赋予值n，如果没有id 对应的，抛出RuntimeException
+ setChild(childID:int, n:Node): void
// 增加一个子节点
+ addChild(n:Node): void
// 删除id 对应的子节点，如果没有id 对应的，抛出RuntimeException
+ delChild(childID:int): void
// 清除childID 对应的子节点所拥有的所有特性，可以理解为将特性矩阵中，
```

```
// childID 所在行的所有值赋予 0
+ clearChildFeature(childID:int): void
// 检测该节点中是否包含 childID 对应的子节点
+ hasChild(childID:int): boolean
// 为 childID 对应的节点标记拥有 features 中的值所对应的特性，
// 可以理解为在特性矩阵中，为 childID 对应的行，
// 和 features 中的值对应特性列，赋予值 2
+ matchFeature2Child(childID:int, features:List<Integer>): boolean
// 效果同上，只是一次只处理一个特性
+ matchFeature2Child(childID:int, featrue:int): boolean
// 返回所有子节点的标识号
+ getChilds(): List<Integer>
// 检索时用的方法，state 为检索状态码，
// 状态码为空数组时代表从该节点下从头开始检索，
// 不为空时，根据数组的内容给出下一步检索步骤，具体参见算法
// 返回值的格式为：
// {
//    nextType:Node.NEXTTYPE_CONTINUE | Node.NEXTTYPE_NEXTNODE
//    feature:
//    featureID:
//    curr:
//    nextID:
// }
// 如果 nextType == Node.NEXTTYPE_CONTINUE，
// 则返回 nextType、feature、featureID、curr 四个参数，
// 如果 nextType == Node.NEXTTYPE_NEXTNODE，
// 则返回 nextType、nextID 两个参数
// Node.NEXTTYPE_CONTINUE 表示在继续在该节点下检索
// Node.NEXTTYPE_NEXTNODE 表示在该节点下的检索已经完成，可以
// 从返回参数中的 nextID 获得下一个节点
+ next(selState:int[]): Map<String, String>
```

2.2.2 NodeManager 类

```
// 管理节点间的关系，持久化、反持久化节点的类
class NodeManager
// 从数据库中返回指定 id 的节点，如果没有 id 对应的，返回 null
+ queryNode(id:int): Node
// 在数据库中对应 id 的行上标识 deleteFlag 为 true，返回操作是否成功
+ delNode(id:int): boolean
// 在数据库中加入一个节点，需要检查 id 的值是否合法，返回操作是否成功
+ insertNode(n:Node): boolean
// 将数据库中 id 对应的节点更新为 n，返回操作是否成功
// 注意，如果 n 中的标识号和 id 不同，则将数据行的 id 更新为 n 中的标识号
```

```
+ updateNode(id:int, n:Node): boolean
// 将 childID 对应的节点作为 parentID 对应的节点的子节点添加，
// 并为 childID 添加 selFeatures 中的值对应的特性，
// newFeatures 中的特性作为新特性添加至 parentID 对应的节点中，
// 并同时将 newFeatures 中的特性标记为 childID 对应的节点所有
+ mount( parentID:int,
          childID:int,
          selFeatures:List<Integer>): boolean
// 将 childID 对应的节点从 parentID 对应节点的子节点中删除
// childID 在 parentID 中的特性予以保留
+ unmount(parentID:int, childID:int): boolean
```

2.2.3 EditorManager 类

```
// 增加、删除、修改节点业务的类
class EditorManager
```

```
+ final static ACTION_INSERT:int
+ final static ACTION_DELETE:int
+ final static ACTION_UPDATE:int
```

```
+ action(req:HTTPServletRequest, type:int): Map<String, String>
- insertAction(req:HTTPServletRequest): Map<String, String>
- delAction(req:HTTPServletRequest): Map<String, String>
- updateAction(req:HTTPServletRequest): Map<String, String>
```

2.2.4 RetrievalManager 类

```
// 检索节点业务的类
class RetrievalManager
```

```
+ final static ACTION_LOCAL:int
+ final static ACTION_OTHER:int
+ final static ACTION_BROWSE:int
+ final static ACTION_GET_SITES:int
```

```
+ action(req:HTTPServletRequest, type:int): Map<String, String>
- localRetrievalAction(req:HTTPServletRequest): Map<String, String>
- otherRetrievalAction(req:HTTPServletRequest): Map<String, String>
- browseAction(req:HTTPServletRequest): Map<String, String>
- otherSitesAction(req:HTTPServletRequest): Map<String, String>
```

2.2.5 NodePersistent 类

```
// 负责在节点的对象形式和 xml 形式之间转换的类
class NodePersistent
```

```
// 返回n的持久化字符串形式，即一串xml，格式参照算法一章
+ persist(n:Node): String
// 将一串xml实例化成一个Node对象，xml格式参照算法一章
+ unPersist(xml:String): Node
```

2.2.6 LoginManager 类

```
// 身份认证类
class LoginManager
```

```
// 给定一对user和pwd，判断是否是已注册用户
+ validate(user:String, pwd:String): boolean
```

2.2.7 HTTPRetrieval 类

```
// 负责向第三方网站检索的辅助类，此类为抽象类
class abstract HTTPRetrieval
```

```
// 此方法为抽象方法，子类覆盖此方法以实现不同网站的不同检索方式，
// 返回数据的格式再商议
+ abstract query(data:String): Map<String, String>
```

2.2.8 SQLHelper 类

```
// 数据持久化辅助类
class SQLHelper
```

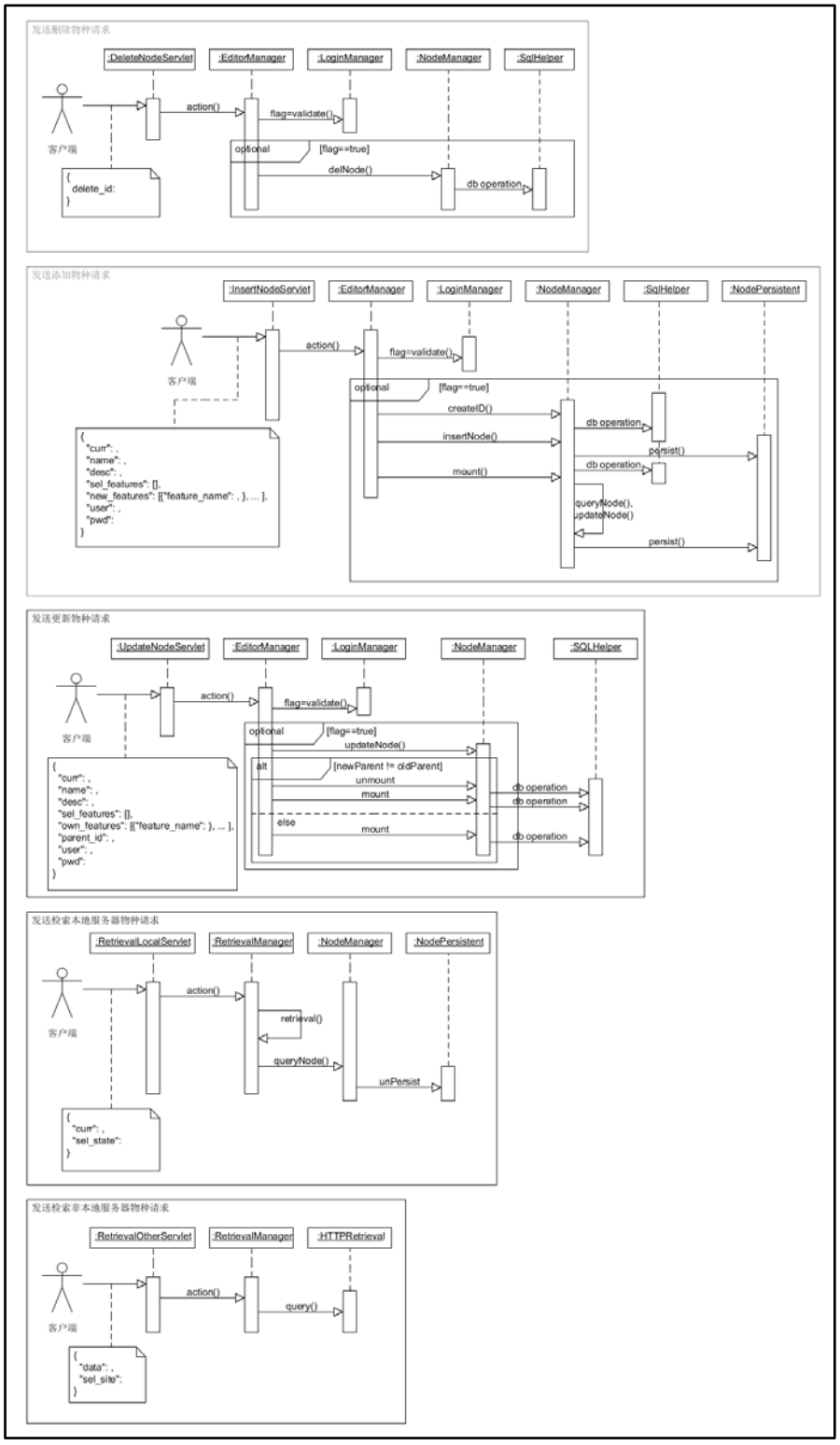
```
+ open(conStr:String): Connection
+ update(sql:String): int
// 返回的Map对象中的key为数据库中的字段名，
// value为该字段名对应值的String形式
+ query(conn:Connection, sql:String): Map<String, String>
```

2.2.9 Servlet 类型子类

```
class RetrievalLocalServlet
class RetrievalOtherServlet
class OtherSitesServlet
class BrowserServlet
class InsertNodeServlet
class UpdateNodeServlet
class DeleteNodeServlet
```

第 3 章 详细设计

3.1 序列图



工程文件：Fish_Seq.uxf

3.2 检索（Code_RetrievalLocal.txt）

```
/* 伪代码:检索
 *
 * 开启语法着色，使用等宽字体，可以更好的阅读代码
 *
 * 客户端发送请求:
 *   {
 *     curr: ,
 *     sel_state:
 *   }
 * 至~/retrievallocal
 */

public void RetrievalLocalServlet.doPost(HTTPServletRequest req, HTTPServletResponse res) {
     res.getWriter().print(parseJSON(new RetrievalManager.action(req, ACTION_LOCAL)));
}

public Map<String, String> RetrievalManager::action(HTTPServletRequest req, int type) {
     switch (type) {
          case ACTION_LOCAL:
               return localRetrievalAction(req);
          case ...
     }
}

private Map<String, String> RetrievalManager::loaclRetrievalAction(HTTPRequest req) {
     int[] selState = parseIntArray(req.getParameter("sel_state"));
     int currID = parseInt(req.getParameter("curr"));

     return retrieval(currID, selState);
}

private Map<String, String> RetrievalManager::retrieval(int currID, int[] selState) {
     Node currNode = nodeMgr.queryNode(curr);
     Map<String, String> nextStep = currNode.next(selState);

     if (result.get("nextType").equals(Node.NEXTTYPE_CONTINUE)) {
          // 说明在该 currNode 下的检索还没有完成
          // 需要返回 Feature 继续提问客户端
          return {
               curr      : result.get("curr"),
```

```
                sel_state  : result.get("selState"),
                feature    : result.get("feature"),
                feature_ID : result.get("featureID"),
                result     : null
           };
     }else if (result.get("nextType").equals(Node.NEXTTYPE_NEXTNODE)) {
           // 完成在该 currNode 下的检索
           // 转到 result 所在的 Node 继续检索
           int nextID = parseInt(result.get("nextID"));
           // 递归，传递空数组表示从 nextNode 的初始状态开始检索
           retrieval(nextID, new int[] {});
     }
}
```

3.3 通过其他站点检索（Code_RetrievalOther.txt）

```
/* 伪代码:通过其他网站检索
 *
 * 开启语法着色，使用等宽字体，可以更好的阅读代码
 *
 * 客户端发送请求:
 *   {
 *     data: ,
 *     sel_site:
 *   }
 * 至~/retrievalother
 */

public void RetrievalOtherServlet(HTTPServletRequest req, HTTPServletResponse res) {
     res.getWriter().print(parseJSON(new RetrievalManager().action(req, ACTION_OTHER)));
}

public Map<String, String> RetrievalManager::action(HTTPServletRequest req, int type) {
     switch (type) {
           case ACTION_OTHER:
                return otherRetrievalAction(req);
           case ...
     }
}

private Map<String, String> RetrievalManager::otherRetrievalAction(HTTPServletRequest req) {
     String data = req.getParameter("data");
     int site = parseInt(req.getParameter("site_id"));
```

```
     HTTPRetrieval httpRetrv;

     // 使用多态
     switch(site){
          case XX :
               httpRetrv = new XXHTTPRetrieval();
               break;
          case XXX :
               httpRetrv = new XXXHTTPRetrieval();
               break;
     }

     return httpRetrv.query(data);
}
```

3.4 查询有哪些其他站点（Code_OtherSites.txt）

```
/* 伪代码:浏览节点
 *
 * 开启语法着色，使用等宽字体，可以更好的阅读代码
 *
 * 客户端发送空请求至~/othersites
 */

public void OtherSitesServlet(HTTPServletRequest req, HTTPServletResponse res) {
     res.getWriter().print(parseJSON(new RetrievalManager().action(req, ACTION_GET_SITES)));
}

public Map<String, String> RetrievalManager::action(HTTPServletResponse req, int type) {
     switch (type) {
          case ACTION_GET_SITES:
               return otherSitesAction(req);
          case ...
     }
}

private RetrievalManager::otherSitesAction(HTTPServletRequest req) {
     result = { "other_sites": [] };

     for ( ... ) {
          result.other_sites.put(
               {"site_name": xxx, "site_id": xxx}
          );
     }
```

```
     return result;
}
```

3.5 浏览节点（Code_RetrievalBrowse.txt）

```
/* 伪代码:浏览节点
 *
 * 开启语法着色，使用等宽字体，可以更好的阅读代码
 *
 * 客户端发送请求:
 *   {
 *     curr:
 *   }
 * 至~/browser
 */

public void BrowserServlet(HTTPServletRequest req, HTTPServletResponse res) {
     res.getWriter().print(parseJSON(new RetrievalManager().action(req, ACTION_BROWSE)));
}

public Map<String, String> RetrievalManager::action(HTTPServletResponse req, int type) {
     switch (type) {
          case ACTION_BROWSE:
               return browseAction(req);
          case ...
     }
}

private Map<String, String> RetrievalManager::browseAction(HTTPServletRequest req) {
     int curr = parseInt(req.getParameter("curr"));
     NodeManager nodeMgr = new NodeManager();
     Node node = nodeMgr.queryNode(curr);
     List<Integer> childsID = node.getChilds();
     result = { curr_childs: [] };

     for (Integer id : childsID) {
          Node aChild = nodeMgr.queryNode(id.intValue());
          result.curr_childs.put({
               "node_name" : aChild.getName(),
               "node_id"   : aChild.getID(),
               "node_pic"  : ""
          });
     }
```

```
     return result;
}
```

3.6 插入节点（Code_InsertNode.txt）

```
/* 伪代码:插入节点
 *
 * 开启语法着色，使用等宽字体，可以更好的阅读代码
 *
 * 客户端发送请求:
 *   {
 *     "curr": ,
 *     "name": ,
 *     "desc": ,
 *     "sel_features": [],
 *     "new_features": [{"feature_name": , }, ... ],
 *     "user": ,
 *     "pwd":
 *   }
 * 至~/insertnode
 */

public void InsertNodeServlet.doPost(HttpServletRequest req, HttpServletResponse res) {
     res.getWriter().print(parseJSON(new EditorManager.action(req, ACTION_INSERT)));
}

public Map<String, String> EditorManager.action(HTTPServletRequest req, int type) {
     switch(type) {
          case ACTION_INSERT:
               return insertAction(req);
          case ...
     }
}

private Map<String, String> EditorManager.insertAction(HTTPServletRequest req) {
     if (!new LoginManager().validate(req.getParameter("user"), req.getParameter("pwd"))) {
          return {
               "result"  : "failed",
               "details" : "Access Denied"
          };
     }

     String desc = req.getParameter("desc");
```

```
     String name = req.getParameter("name");
     int parentId = parseInt(req.getParameter("curr"));
     List<Integer> selFeatures = parseIntList(req.getParameter("sel_features"));
     List<Feature> ownFeatures = parseFeatures(req.getParameter("own_features"));

     NodeManager nodeMgr = new NodeManager();
     int newId = nodeMgr.createID();
     Node newNode = new Node(newID, name, desc, parentID, ownFeatures);

     if (!nodeMgr.insertNode(newNode)) { // See NodeManager.insertNode() fn below
          return {
               "result"  : "failed",
               "details" : "Internal Error"
          }
     }
     if (!nodeMgr.mount(parentID, newID, selFeatures)) {
          return {
               "result"  : "failed",
               "details" : "Internal Error"
          }
     }
}

public boolean NodeManager.insertNode(Node node) {
     if (node.getID() == null || node.getID().equals("")) {
          log("loose node id.");
          return false;
     }

     String xml = NodePersistent.persist(node);
     // conn->sql->SqlHelper.update(conn, sql)
}

public boolean NodeManager.mount( int parentID,
                                  int childID,
                                  List<Integer> selFeatures) {

     Node parentNode = queryNode(parentID);
     parentNode.addChild(childID);

     // 将已有的 feature 匹配至指定的 Child
     if (!parentNode.matchFeature2Child(childID, selFeatures))
          return false;
```

```
     return updateNode(parentNode.getID(), parentNode); // See updateNode fn below.
}

public boolean NodeManager.updateNode(int id, Node node) {
     if (node.getID() == null) {
          log("loose node id.");
          return false;
     }

     String xml = NodePersistent.persist(node);
     // conn->sql->SQLHelper.update(conn, sql)
}
```

3.7 更新节点（Code_UpdateNode.txt）

```
/* 伪代码:更新节点
 *
 * 开启语法着色，使用等宽字体，可以更好的阅读代码
 *
 * 客户端发送请求:
 *   {
 *     "curr": ,
 *     "name": ,
 *     "desc": ,
 *     "sel_features": [],
 *     "own_features": [{"feature_name": }, ... ],
 *     "parent_id": ,
 *     "user": ,
 *     "pwd":
 *   }
 * 至~/updatenode
 */

public void UpdateNodeServlet.doPost(HttpServletRequest req, HttpServletResponse res) {
     Map<String, String> result = new EditorManager().action(req, ACTION_UPDATE);
     res.getWriter().print(parseJSON(result));
}

public Map<String, String> EditorManager.action(HTTPServletRequest req, int type) {
     switch(type) {
          case ACTION_UPDATE:
               return updateAction(req);
          case ...
     }
```

```
}

public Map<String, String> EditorManager.updateAction(HTTPRequest req) {
     if (!new LoginManager().validate(req.getParameter("user"), req.getParameter("pwd"))) {
          return {
               "result"  : "failed",
               "details" : "Access Denied"
          };
     }

     NodeManager nodeMgr = new NodeManager();
     String name = req.getParameter("name");
     String desc = req.getParameter("desc");
     int oldParentID = parseInt(req.getParameter("curr"));
     List<Integer> selFeatures = parseIntList(req.getParameter("sel_features"));
     List<Feature> ownFeatures = parseFeatures(req.getParameter("own_features"));
     int newParentID = parseInt(req.getParameter("parent_id"));
     int id = parseInt(req.getParameter("node_id"));
     Node node = nodeMgr.queryNode(id);

     // Update basic data.
     node.setDesc(desc);
     node.setName(name);
     node.setParentID(newParentID);
     for (Feature f : ownFeatures) {
          node.addFeature(f);
     }

     try {
          // Flash basic data.
          if (!nodeMgr.updateNode(nodeID, node))
               throw new Exception("Update Failed.");

          // If parent changed
          if (newParentID != oldParentID) {
               if (nodeMgr.unmount(oldParentID, nodeID))
                    throw new Exception("Unmount failed when change parent.");
               if (nodeMgr.mount(newParentID, nodeID, selFeatures)) {
                    throw new Exception("Mount failed when change parent.");
               }
          }else {
               if (nodeMgr.mount(newParentID, nodeID, selFeatures)) {
                    throw new Exception("Mount failed when update node.");
               }
```

```
          }
     }catch (Exception ex) {
          return {
               "result"  : "failed",
               "details" : "Internal Error"
          };
     }
}
```

3.8 删除节点（Code_DeleteNode.txt）

```
/* 伪代码:删除节点
 *
 * 开启语法着色，使用等宽字体，可以更好的阅读代码
 *
 * 客户端发送请求:
 *   {
 *     "delete_id": 需要被删除节点的 ID
 *   }
 * 至~/deletenode
 */

public void DeleteNodeServlet.doPost(HttpServletRequest req, HttpServletResponse res) {
     res.getWriter().print(parseJSON(new EditorManager.action(req)));
}

public Map<String, String> EditorManager.action(HTTPServletRequest req, int type) {
     switch(type) {
          case ACTION_DELETE:
               return deleteAction(req);
          case ...
     }
}

private Map<String, String> EditorManager.deleteAction(HTTPRequest req) {
     try {
          if (!new LoginManager().validate(req.getParameter("user"), req.getParameter("pwd")))
               throw new ValidateException("Wrong user or pwd.");

          int nodeID = parseInt(req.getParameter("del_id"));
          if (!NodeManager().delNode(nodeID))
               throw new InternalException("del failed.");

          return {
```

```
                "result"  : "ok",
                "details" : ""
          };

     }catch (ValidateException ex) {
          return {
                "result"  : "failed",
                "details" : "Access Denied"
          };
     }catch (InternalException ex) {
          log(ex);
          return {
                "result"  : "failed",
                "details" : "Internal Error."
          }
     }
}

public boolean NodeManager.delNode(int nodeID) {
     // conn->sql->SQLHelper.update(conn, sql)
     // 只是标记该条数据的 deleteFlag 字段为 true
}
```

第 4 章 算法

4.1 检索算法

参看“基于数字化的生物分类鉴定及知识集成研究”第五章的 1.1 节、1.2 节和第二章的 1 节、2 节、3 节。

4.2 数据存储结构

这里的数据存储结构指的是如何在数据库中存储物种信息，以及如何建立起他们之间的联系。

我们将具体物种（XXX 鱼）和分类（XX 纲）统一抽象为节点。

通过数据库技术为每一个节点分配一个在数据库中全局唯一的标识号，用来唯一标识一个节点。节点中的信息在数据库中的存储形式就是一串 xml 字符串，所以从数据库表的角度看，节点表结构非常简单，主要存储 2 个字段：标识号和节点信息（xml 字符串）。

节点之间的关系是指这个节点的父节点是谁，这个节点的子节点又有哪些，这很像用链表来实现树这种数据结构。由于每个节点都有了唯一的标识号，所以我们可以在节点信息中方便的加入其它节点的标识号来指出其父节点和子节点。所以若要查找一个节点 A 的子节点 B 的信息，需要两步，第一步，用 A 的标识号找到 A 节点的信息，读出 B 的标识号。第二步，根据 B 的标识号找到 B 的信息。

数据行中的 xml 格式参见农业本体一书。

4.3 检索状态码

检索过程是一个状态不断变化的过程（回答了第几个 feature？当前处于哪个节点？下一步又会处于哪个节点？），如果不对这些状态进行保存，则根本无法开展检索工作。在通常的 Web 实现技术下保存状态的实现是将状态写入客户端的 Cookies 中。但由于在服务端操作 Cookie 的具体实现依赖于具体的服务端平台（如 Java，.Net），而客户端操作 Cookie 的具体实现又依赖于浏览器，为考虑到系统的移植性，故不使用 Cookie 来保存检索状态。

我们使用状态码的概念。一个状态码就是一串数字，在客户端与服务端通信的过程中，附带上该状态码用来表示当前的检索状态，状态码的格式如下：

curr	feature_id	answer	feature_id	answer	.....

每一个格子中都是一个数字，其中，curr 是当前节点的 id，answer 有 3 种值，分别是 1、2 和 3。1 代表回答 Yes，2 代表回答 No，3 代表回答 Unknow。每一个 answer 就是对它前面一个格子中 feature 的回答，在检索过程中，服务器端通常会传送一个状态码，一个 feature，一个 featureID，将 feature 显示给检索者看，检索者作出 Yes、No、Unknow 三种回答之一，之后只需要把 featureID 和作出的回答追加在之前给的状态码的后面，发给服务端即可。注意，状态码中各数字从左到右的出现顺序代表了实际检索过程中检索的顺

序，所以该顺序不允许在客户端被随意修改。5.8 节有状态码使用的例子。

第 5 章 服务调用接口

“~”符号代表当前域，比如 http://192.168.1.1/XXX.asp 可以写成～/XXX.asp。

5.1 ~/retrievallocal 服务接口

参数传递格式	JSON
接收参数	
{"curr": , "sel_state": }	
curr	当前节点 ID
sel_state	检索状态码
返回参数	

{
"curr": ,
"sel_state": ,
"feature": ,
"feature_id": ,
"result": {"next": , "next_id": }
}	

curr	当前节点 ID
sel_state	检索状态码
feature	特征的文字描述
feature_id	特征的 ID
next	结果的文字描述
nextID	结果节点的 ID

5.2 ~/retrievalother 服务接口

参数传递格式	JSON
接收参数	
{"data": , "sel_site": }	
data	对物种的描述字符串
site_id	表示第三方网站的 ID

返回参数	
未知	

5.3 ~/browser 服务接口

参数传递格式	JSON
接收参数	
{"curr": }	
curr	当前节点 ID
返回参数	

{
"curr_childs": [
{"node_name": , "node_id": , "node_pic": },
...
]
}	

node_name	节点名字
node_id	节点 ID
node_pic	相关图片（如果有的话）路径，最多一张图片

5.4 ~/insertnode 服务接口

参数传递格式	JSON
接收参数	
{ "curr": , "name": , "desc": , "sel_features": [ ], "own_features": [{"feature_name": }, ...], "user": , "pwd":	

}	
curr	当前节点 ID
name	节点名字
desc	节点（具体）描述
sel_features	对 curr 所在节点的所有 feature 的一组选择，数组存的值是所选 feature 的 ID
own_features	节点的 feature 集合
own_features.feature_name	feature 的名字
user	用户名
pwd	密码
返回参数	
{"result": , "details:": }	
result	显示成功与否，可选值："ok";"failed"
details	操作结果的详细信息（如果有的话）

5.5 ~/updatenode 服务接口

参数传递格式	JSON
接收参数	

{
"curr": ,
"node_id": ,
"name": ,
"desc": ,
"sel_features": [],
"own_features": [{"feature_name": }, ...],
"parent_id": ,
"user": ,
"pwd"
}	

curr	当前节点 id
node_id	被编辑节点的 id
name	节点名字
desc	节点（具体）描述
sel_features	对 curr 所在节点的所有 feature 的一组选择，数组存的值是所选 feature 的 id
own_features	节点的 feature 集合
own_features.feature_name	feature 的名字
parent_id	如果需要修改节点的父节点（即需要移动位

	置 ，则 parent_id 为新父节点的 id，如果不修改，parent_id 的值等于 curr 的值
user	用户名
pwd	密码
返回参数	
{"result": , "details:": }	
result	显示成功与否，可选值："ok";"failed"
details	操作结果的详细信息（如果有的话）

5.6 ~/deletenode 服务接口

接收参数	
{"del_id": , "recursion": , "user": , "pwd": }	
del_id	被删除节点的 ID
recursion	是否递归删除其下所有节点，可选值："true";"false"
user	用户名
pwd	密码
返回参数	
{"result": , "details:": }	
result	显示成功与否，可选值："ok";"failed"
details	操作结果的详细信息（如果有的话）

5.7 ~/othersites 服务接口

接收参数	
{}	
返回参数	
{ "other_sites": [{"site_name": , "site_id": } ...] }	
site_name	站点名称

site_id	站点 id

5.8 服务调用示例

这里以~/retrievallocal 服务接口为对象做服务调用演示（服务调用示例.txt）：

客户端->服务端	{
	"curr": NULL,
	"sel_state": NULL,
	}

服务端->客户端	{
	"curr": 48,
	"sel_state": [48],
	"feature": "Is xxx?",
	"feature_id": 3,
	"result": NULL
	}

客户端->服务端	{
	"curr": 48,
	"sel_state": [48,3,1],
	}

服务端->客户端	{
	"curr": 48,
	"sel_state": [48,3,1],
	"feature": "Is xxx?",
	"feature_id": 4,
	"result": NULL
	}

客户端->服务端	{
	"curr": 48,
	"sel_state": [48,3,1,4,1],
	}

服务端->客户端	{
	"curr": 48,
	"sel_state": [48,3,1,4,1],
	"feature": NULL,
	"feature_id": NULL,
	"result": {
	"next": "XX 科",
	"nextID": 34
	}
	}

客户端->服务端（如果还想继续查询下去的话）	{
	"curr": 34,
	"sel_state": NULL,
	}

服务端->客户端	{
	"curr": 34,
	"sel_state": [34],
	"feature": "Is xxx?",
	"feature_id": 1,
	"result": NULL
	}

第 6 章 项目实施

6.1 人员分工

数据层开发小组	SQLHelper NodePersistent
算法小组	Node Matrix
Web 后端开发小组	NodeManager EditorManager RetrievalManager XXXServlet HTTPRetrieval 及其子类 LoginManager

6.2 实施步骤

先读懂项目书，理解项目需求。

想出自己分工范围内可能会涉及的技术点，比如数据层开发小组需要了解 JDBC，算法小组需要了解二维数组的操作，Web 后端开发小组需要了解 JSON 格式数据的读写、转换等等。

针对每个技术点写写 Demo 练练手，然后就可以开始实现了。