后台开发手册

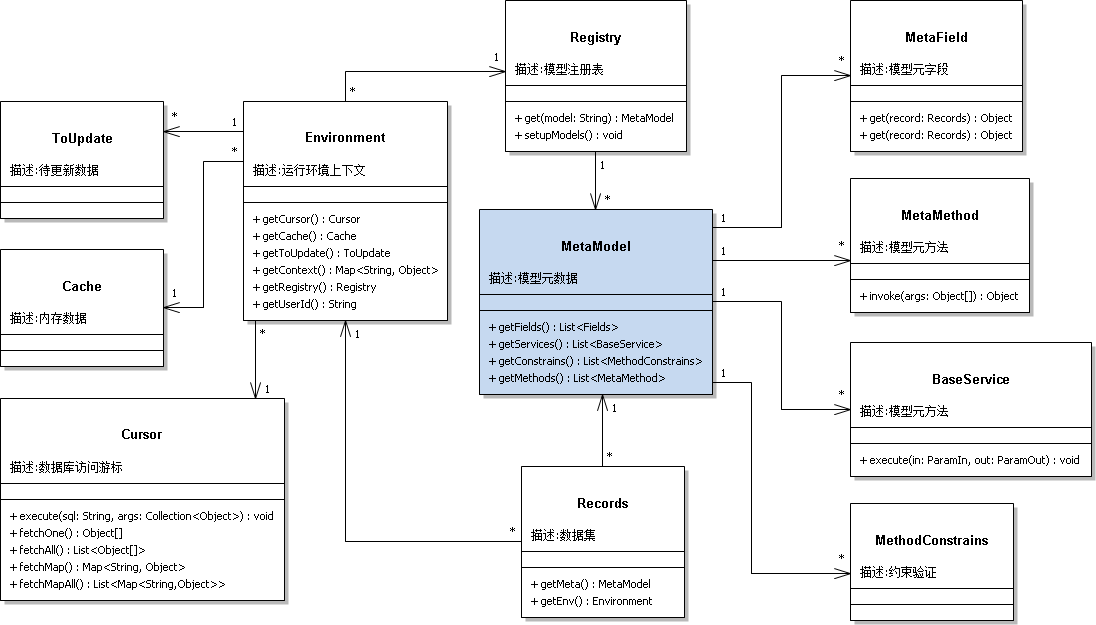
# 概要

## 目的

企业数字化转型，需要部署大量企业级应用，随着业务的发展，很多应用不能及时响应，大大增加了数字化转型的成本。本项目的目标是开发一个能快速响应业务变化的平台。本文档 介绍元模型的相关概念和开发示例。

# 核心概念

模型驱动架构(Model Driven Architecture, MDA) ，以模型为中心，使用元数据定义一个全新的模型体系，让所有模型皆可扩展，让应用能快速响应业务变化。



元模型结构图

## 元模型：

元模型(MetaModel)用于定义模型结构，使用模型(Model)+记录集(Records)的方式定义全新的模型系统。元模型包含模型的描述元数据、元字段、元方法、方法约束、服务等。与常规java类定义的实体不一样，模型用于定义结构(MetaField)和行为(MetaMethod)，记录集用于数据访问。模型的定义突破java语法的限制，可以在多个java类里声明模型的扩展，通过重新构建模型继承链，实现模型纵向多继承扩展+横向组合式扩展。参考Python的MRO(Method resolve order)算法，实现继承链的解析。

## 模型注册表

模型注册表(Registry)保存所有加载的元数据信息。不同租户或环境(生产、测试等)可以加载不同的元模型。

## 环境

环境(Environment)提供运行所需的相关上下文。

## 元字段

元字段(MetaField)用于定义模型的字段，描述字段的类型、标题、帮助等元数据。与java类实现的数据字段不一样，元字段不保存模型实例的数据，字段的数据缓存于内存数据缓存(Cache)中。

## 元方法

元方法(MetaMethod)是基于java的Method进行封装，实现了没继承关系的java类之间的方法重写。

## 记录集

记录集(Records)是同一模型的数据的有序集合。传统的java对象，通过反射获取描述类的元数据，通过字段存储数据。在全新的模型体系中，使用Model和Records分别进行定义。Model定义模型的元数据，模型中的字段不存储数据，用来定义字段的元数据。通过Records对数据执行操作。

## 服务

服务(Service)是模型提供的API，可通过jsonrpc远程调用。

## 约束

约束(Constrains)定义模型的验证，分为唯一约束(UniqueConstraint)和方法约束(MethodConstrains)。

唯一约束：使用@Model.UniqueConstraint声明，生成数据库唯一索引

方法约束：在元方法上声明@Model.MethodConstrains，在执行create\update操作时调用验证

# 模型开发说明

## 模型

使用注解描述模型的信息，在模型中定义字段Field:

|  |
| --- |
| @Model.Meta(name = "res.user", description = "用户")  public class ResUser extends Model {      static Field name = Field.Char().label("名称").help("姓名");  } |

### BaseModel类

业务模型的基类。业务模型继承以下类进行定义：

Model：常规数据库持久化的模型

TransientModel：临时数据，保存到数据库中，会被定时自动删除

AbstractModel：抽象模型，为子类提供共性字段和方法

系统会为每个已安装的持久化模型创建数据库表。模型映射了数据库表的数据结构，数据库表里的数据对应一个"记录集"，Records是有序的数据集合。可以通过browse()方法创建记录集，也可以调用search()从数据库查询记录集。

BaseModel的字段说明

|  |  |  |  |
| --- | --- | --- | --- |
| 字段 | 说明 | 类型 | 默认值 |
| isAuto | 是否应创建数据库表，如果为false，就重写init方法自定义数据库表 | boolean | Model和TransentModel默认为true，AbstractModel默认为false |
| isAbstract | 模型是否为抽象 | boolean | 见AbstractModel |
| isTransient | 是否瞬态 | boolean | 见TransientModel |

除了以上字段，模型的其它描述信息使用@Model.Meta进行声明。

@Model.Meta参数说明

|  |  |  |  |
| --- | --- | --- | --- |
| 参数 | 说明 | 类型 | 默认值 |
| name | 模型的名称，与inherit应至少指定一个 | String |  |
| label | 模型的标题 | String |  |
| description | 模型的说明 | String |  |
| inherit | 继承的模型，与name应至少指定一个   * 如果name已设置，则表示要继承的模型的名称 * 如果name未设置，则表示扩展的单个模型的名称 | String[] |  |
| table | 模型的表名，isAuto为true时才有效 |  |  |
| order | 默认排序SQL，如：name asc |  |  |
| displayField | 记录显示的字段，通常使用name字段 |  |  |
| logAccess | 是否自动生成访问日志字段，默认不指定，使用BaseModel的isAuto字段的值 | BoolState | None |

### AbstractModel

抽象模型，类似java的抽象类，为子类提供共性的字段和方法

### Model

常规数据库持久化的业务模型超类。

业务模型通过继承定义

|  |
| --- |
| public class ResUser extends Model {  } |

系统为每个已安装的模型(安装应用时安装相应的模型)创建数据库表。

isAuto为true时，自动创建表，如果为false，就重写init()方法，自定义创建表

|  |
| --- |
| public class Foo extends Model {      public Foo(){          isAuto = false;      }      @Override      public void init(Records rec){          Cursor cr = rec.getEnv().getCursor();          cr.execute("CREATE TABLE ...");          ...      }  } |

### TransientModel

临时数据模型的超类，表示暂时持久化，并定期进行删除清理。

用户只能访问他们自己创建的记录。

### 继承和扩展

系统提供了以模块化的方式来继承或扩展模型：

* 继承现有模型创建新模型，将新信息添加到新模型，不影响原来的模型
* 就地扩展其他模块中定义的模型，替换以前的版本

#### 继承

指定inherit和name创建新的模型，inherit是被继承的模型名称，name是新模型的名称。新模型将继承原模型的所有字段、方法和服务。

|  |
| --- |
| @Model.Meta(name = "inheritance.0", description = "Inheritance Zero")  public class Inheritance0 extends Model {      static Field name = Field.Char().label("名称");      public void run(Records rec){          check(rec, "model 0");      }      public void check(Records rec, String s){          System.out.println(String.format("This is %s record %s", s, rec.get("name")));      }  }  @Model.Meta(name = "inheritance.1", inherit = "inheritance.0", description = "Inheritance One")  public class Inheritance1 extends Model {      static Field name = Field.Char().label("名称");      public void run(Records rec){          rec.call("check", "model 1");      }  } |

调用示例：

|  |
| --- |
| Records a = env.get("inheritance.0").create(new HashMap<String, Object>() {      {        put("name", "A");      }  });  Records b = env.get("inheritance.1").create(new HashMap<String, Object>() {      {        put("name", "B");      }  });  a.call("run");  b.call("run"); |

输出结果

|  |
| --- |
| This is model 0 record A  This is model 1 record B |

#### 扩展

指定inherit，但省略name时，新模型会替换现有模型，实质上就是扩展了原来的模型。这对于向现有模型（在其他模块中创建）添加新字段或方法很有用。

|  |
| --- |
| @Model.Meta(name = "inheritance.0", description = "Inheritance Zero")  public class Inheritance0 extends Model {      static Field name = Field.Char().label("名称").defaultValue(Default.value("A"));  }  @Model.Meta(inherit = "inheritance.0")  public class Inheritance1 extends Model {      static Field description = Field.Char().label("名称").defaultValue(Default.value("Extended"));  } |

调用示例：

|  |
| --- |
| Records r = env.get("inheritance.0").create(Collections.emptyMap());  Map<String, Object> values = r.read(Arrays.asList("name", "description")).get(0);  System.out.println(values); |

输出结果

|  |
| --- |
| {name=A, description=Extended} |

## 字段

### Field

字段Field用于配置元数据，并不保存记录的值。

字段通用参数：

|  |  |  |  |
| --- | --- | --- | --- |
| 参数 | 说明 | 类型 | 默认值 |
| label | 用户看到的字段的标签,如果未设置，默认显示字段名 | String |  |
| help | 用户看到的字段的帮助提示 | String |  |
| readonly | 该字段是否为只读。这只会对 UI 产生影响。代码中的任何字段设置值都将起作用 | boolean | false |
| required | 字段的值是否是必需的 | boolean | false |
| store | 字段是否存储在数据库中，对计算字段不生效 | boolean | true |
| compute | 计算字段，通过计算获取值，不保存于数据库。可以由元方法计算，也可以是groovy脚本计算 | Callable |  |
| copy | 复制记录时是否应复制字段值 | boolean | 对于普通字段：true  对于one2many，related字段：false |
| index | 该字段是否在数据库中被索引，对于非存储字段不生效 | boolean | false |
| depends |  |  |  |
| auth | 是否需要权限控制 | boolean | false |
| prefetch | 是否预加载，设置false表示懒加载 | boolean | binary字段：false  其它字段：true |
| defaultValue | 字段的默认值。可以是常量值，可是通过元方法提供值，也可以通过groovy脚本提供值 | Default |  |

### 基础字段

#### Boolean

TODO

#### Char

TODO

#### Float

TODO

#### Integer

TODO

#### Binary

TODO

#### Html

TODO

#### Image

TODO

#### Selection

TODO

#### Text

TODO

#### Date

TODO

#### DateTime

TODO

### 高级字段

#### 关联字段

##### Many2one

TODO

##### One2many

TODO

##### Many2many

TODO

##### Reference

TODO

##### Many2oneReference

TODO

#### 计算字段

TODO

#### 自动字段

TODO

## 记录集

与模型和数据的交互是通过记录集执行的，记录集是同一模型的数据的有序集合。

在模型上定义的元方法，第一个参数必须是记录集对象：

|  |
| --- |
| @Model.Meta(name = "a.foo")  public class Foo extends Model {      public void myMethod(Records rec){          //rec 可以是空记录，与可以是数据库中所有记录          doOperation(rec);      }  } |

对记录集进行迭代将产生单个记录的新集 (单行记录)

|  |
| --- |
| public void doOperation(Records rec){          System.out.print(rec); // -> a.foo[id1,id2,id3]          for(Records r : rec){              System.out.print(r); // -> a.foo[id1]a.foo[id2]a.foo[id3]          }      } |

### 字段数据访问

记录集提供数据访问接口，字段值可以像字典项一样被读取和写入。

|  |
| --- |
| rec.set("name", "myName");  System.out.print(rec.get("name")); // -> myName |

把记录集转换成模型类，可以使用get/set方法更优雅地访问字段，使用of()转换多行对象，使用as()转换单行对象：

|  |
| --- |
| @Model.Meta(name = "a.foo")  public class Foo extends Model {      static Field name = Field.Char().label("名称")      /\*\* get名称 \*/      public String getName() {          return (String) get(name);      }      /\*\* set名称 \*/      public void setName(String value) {          set(name, value);      }      public void myMethod(Records rec){          for(Foo f : rec.of(Foo.class)){              f.setName("myName");              System.out.print(f.getName()); // -> myName          }          for(Records r : rec){              ResUser f = r.as(ResUser.class);              f.setName("myName");              System.out.print(f.getName()); // -> myName          }      }  } |

访问关系字段 ( Many2one, One2many, Many2many) 总是返回一个记录集，如果该字段未设置，则为空。

### 记录缓存和预加载

系统为字段的数据维护一个缓存(Cache)，因此并非每个字段访问都会发出数据库请求，这对性能来说是很糟糕的。以下示例仅针对第一条语句查询数据库：

|  |
| --- |
| rec.get("name"); // 第一次访问从数据库读取  rec.get("name"); // 第二次访问从缓存读取 |

为了避免一次读取一行记录上的一个字段，系统会预读取更多的记录和字段数据以获得良好的性能，预读取的数据会存储在缓存中以供以后使用。

预读取字段：所有prefetch=true的字段。

预读取记录：通过迭代循环访问单行记录时，会预加载被迭代的记录集的数据。

观察以下示例，假设 Records 有1000行数据，如果没有预加载，循环访问会产生2000次数据库请求，有预加载，只会请求一次：

|  |
| --- |
| for(Records r : rec){ // rec -> a.foo[id1,id2,...id1000]    r.get("name"); // 第一次访问字段时，预加载 name 和 qty (还有其它定义为预加载的字段)  // 并且读取了rec的所有行数据(id1,id2,...id1000)    r.get("qty");  } |

预加载对于关联字段同样有效。访问关联字段时，他们的记录集也会预加载。以下示例只产生两次请求，一次请求foo，一次请求bar：

|  |
| --- |
| for(Records f : foo){    Records bar = (Records)f.get("bar\_id"); // 第一次访问foo字段时，预加载所有foo  bar.get("name"); // 第一次访问bar字段时，预加载所有bar  } |

注：如果记录集超过1000行，受限于数据库in最大1000个参数，系统会分批次，每次1000行进行读取。

## 方法

模型的方法在定义模型的java类中定义，但并不是类中所有的java方法都会成为模型的方法。满足以下规则的java方法才会被解析为模型的方法：

* 同一模型中方法名必须唯一。为了避免冲突，模型方法不支持重载，类似于Python的方法。
* 必须声明为public。模型类中定义的非public方法会被认为是辅助方法，不会解析为模型方法。
* 第一个参数必须是数据集(Records)。

模型的方法可以定义在多个扩展或多个被继承的模型中，它们可能在不同的java类中，并且这些java类没有继承关系。模型的方法可以在扩展或继承模型中重写。方法是定义在Model类中，记录集Records通过call()方法调用模型的方法。

记录集还提供了基础方法(create, update, read, search, delete, exists等)，可以直接调用。同一java类中的方法也可以直接调用：

|  |
| --- |
| @Model.Meta(name = "a.foo")  public class Foo extends Model {      public void myMethod1(Records rec, Records arg){ ... }      public void myMethod2(Records rec){  rec.call("myMethod1", "arg"); // 使用call调用指定名称的方法  // 在其它类中定义的方法只能使用call调用          myMethod1(rec, "arg"); // 同一java类中的方法直接调用  rec.read(Arrays.asList("field1", "field2")); // Records提供的基础方法可直接调用      }  } |

## 服务

模型的方法只能在代码中调用，服务可以提供远程调用。

服务有两种方式声明：

* 在模型方法上使用@Model.ServiceMethod声明
* 在模型上使用@Model.Service声明一个BaseService的子类提供服务方法

|  |
| --- |
| @Model.Service(name = "test1", type = FooTestService.class) // 在模型上声明服务  public class Foo extends Model {  @Model.ServiceMethod // 在模型方法上声明服务      public void test2(Records rec, String arg) {          //服务逻辑      }  }  public class FooTestService extends Service {      @Override      protected void execute(ParamIn in, ParamOut out) {          //服务逻辑      }  } |

服务远程遵循 jsonrpc 2.0 协议，调用的地址：http(s)://ip:port/rpc/service，示例：

清求：

|  |
| --- |
| {      "jsonrpc": "2.0",      "method": "read",      "id": "99BA5433-DF5F-A898-C8E0-78B8BA55F251",      "params": {          "args": {              "ids": ["01ng9if6fve2o"],              "fields": ["name", "login"]          },          "model": "res.user",          "context": {              "tz": "",              "uid": "01ng9if6fve2o",              "lang": "zh\_CN",              "ticket": "ticket",              "tenant": "tenant"          }      }  } |

响应：

|  |
| --- |
| {      "id": "99BA5433-DF5F-A898-C8E0-78B8BA55F251",      "result": {          "data": [              {                  "name": "张三",                  "id": "01no6a1ocg9hc",                  "login": "zhangsan"              }          ],          "context": {              "token": "token"          }      },      "jsonrpc": "2.0"  } |

### 通用模型服务

#### create

为模型创建新记录。

请求：

|  |
| --- |
| {      "jsonrpc": "2.0",      "id": "guid",      "method": "create",      "params": {          "args": {              "field1": "value1",              "field2": "value2"          },          "model": "model name",          "context": {          }      }  } |

响应：

|  |
| --- |
| {      "id": "guid",      "result": {          "data": "01no7um0jilfk",          "context": {        }      },      "jsonrpc": "2.0"  } |

#### create\_batch

批量创建新记录。

请求：

|  |
| --- |
| {      "jsonrpc": "2.0",      "method": "create\_batch",      "id": "guid",      "params": {          "args": {              "values": [                  {                      "field1": "value11",                      "field2": "value12"                  },                  {                      "field1": "value21",                      "field2": "value22"                  }              ]          },          "model": "model name",          "context": {          }      }  } |

响应：

|  |
| --- |
| {      "id": "guid",      "result": {          "data": [              "01no6a1ocg9hc",              "01no6a1otxjwg"          ],          "context": {          }      },      "jsonrpc": "2.0"  } |

#### find

根据参数搜索记录。

请求：

|  |
| --- |
| {      "jsonrpc": "2.0",      "method": "find",      "id": "guid",      "params": {          "args": {              "criteria": [],              "offset":0,              "limit":10,              "order":"create\_date desc",              "next\_test":**false**          },          "model": "model name",          "context": {          }      }  } |

响应：

|  |
| --- |
| {      "id": "guid",      "result": {          "data": {              "ids": [                  "01no6a1ocg9hc",                  "01no6a1otxjwg"              ]          },          "context": {          }      },      "jsonrpc": "2.0"  } |

#### read

读取记录集指定字段的值。

请求：

|  |
| --- |
| {      "jsonrpc": "2.0",      "method": "read",      "id": "guid",      "params": {          "args": {              "ids": ["01no6a1ocg9hc"],              "fields": ["field1", "field2"]          },          "model": "model name",          "context": {          }      }  } |

响应：

|  |
| --- |
| {      "id": "guid",      "result": {          "data": [              {                  "field1": "value1",                  "id": "01no6a1ocg9hc",                  "field2": "value2"              }          ],          "context": {         }      },      "jsonrpc": "2.0"  } |

#### search

搜索并读取记录集指定字段的值。

请求：

|  |
| --- |
| {      "jsonrpc": "2.0",      "method": "search",      "id": "guid",      "params": {          "args": {              "criteria": [],              "fields": ["field1", "field2"],              "limit": 10,              "next\_test" : **true**          },          "model": "model name",          "context": {          }      }  } |

响应：

|  |
| --- |
| {      "id": "guid",      "result": {          "data": {              "hasNext": **false**,              "values": [                  {                      "field1": "value1",                      "field2": "value2"                  },                  {                      "field1": "value3",                      "field2": "value4"                  }              ]          },          "context": {          }      },      "jsonrpc": "2.0"  } |

#### count

统计匹配条件的记录数。

请求：

|  |
| --- |
| {      "jsonrpc": "2.0",      "method": "count",      "id": "guid",      "params": {          "args": {              "criteria": []          },          "model": "model name",          "context": {          }      }  } |

响应：

|  |
| --- |
| {      "id": "guid",      "result": {          "data": 0,          "context": {         }      },      "jsonrpc": "2.0"  } |

#### delete

删除当前集合的记录。

请求：

|  |
| --- |
| {      "jsonrpc": "2.0",      "method": "delete",      "id": "guid",      "params": {          "args": {              "ids": ["01m3p9jeo3lkw"]          },          "model": "model name",          "context": {          }      }  } |

响应：

|  |
| --- |
| {      "id": "guid",      "result": {          "data": **true**,          "context": {          }      },      "jsonrpc": "2.0"  } |

#### update

使用提供的值更新当前集中的所有记录。

请求：

|  |
| --- |
| {      "jsonrpc": "2.0",      "method": "update",      "id": "guid",      "params": {          "args": {              "ids": [                  "01m3p9jeo3lkw"              ],              "values": {                  "field1": "value1",                  "field2": "value2"              }          },          "model": "model name",          "context": {          }      }  } |

响应：

|  |
| --- |
| {      "id": "guid",      "result": {          "data": **true**,          "context": {          }      },      "jsonrpc": "2.0"  } |

## 环境

所有记录集都有一个环境(Environment)，该环境是不可变的。环境提供模型使用的各种上下文数据：

* 当前请求的环境参数(用户、言语、时区、租户等)
* 模型注册表(Registry)
* 数据库访问游标(Cursor)
* 内存数据缓存(Cache)
* 数据待更新列表(ToUpdate)

从其他记录集创建记录集时，将继承环境。该环境可用于获取其他模型中的空记录集，并查询该模型：

|  |
| --- |
| rec.getEnv().get("foo"); // a.foo[]  rec.getEnv().get("foo").search(rec.filter("name","=","foo1")); // a.foo[01no6a1otxjwg] |

### 修变环境

Records.withContext(Map<String,Object> ctx)

返回附加到扩展上下文的此记录集的新版本

Records.withNewContext(Map<String,Object> ctx)

返回附加到全新上下文的此记录集的新版本

Records.withUser(String uid)

返回附加到给定用户的此记录集的新版本

Records.withEnv(Environment env)

返回附加到提供的环境的此记录集的新版本

### 执行SQL

遇到模型无法实现的复杂查询或者基于性能原因，可以从环境getCursor()方法获取当前数据库事务的游标Cursor，允许直接执行SQL。

|  |
| --- |
| rec.getEnv().getCursor().execute("some\_sql", params); |

执行自定义的INSERT\UPDATE\DELETE SQL后，需要调用Cache的invalidate()方法清除缓存以避免模型的数据不一致。