# GORM

domain是应用的核心组件，domain类代码业务处理的状态。domain之间是相互关联的，有一对一、一对多和多对多的关系。

GORM是grails的ORMapping实现。GORM基于hibernate 3，通过groovy动态脚本语言和grails的约定高于配置的特性，GORM使用起来要比hibernate简单的多。下面就是一个在action中使用GORM的例子：

|  |
| --- |
| def book = Book.findByTitle("Groovy in Action")  book  .addToAuthors(name:"Dierk Koenig")  .addToAuthors(name:"Guillaume LaForge")  .save() |

GORM解决了如何将domain对象持久化到数据库中的问题。

## 快速上手

grails命令：create-domain-class用来创建domain类。例如：

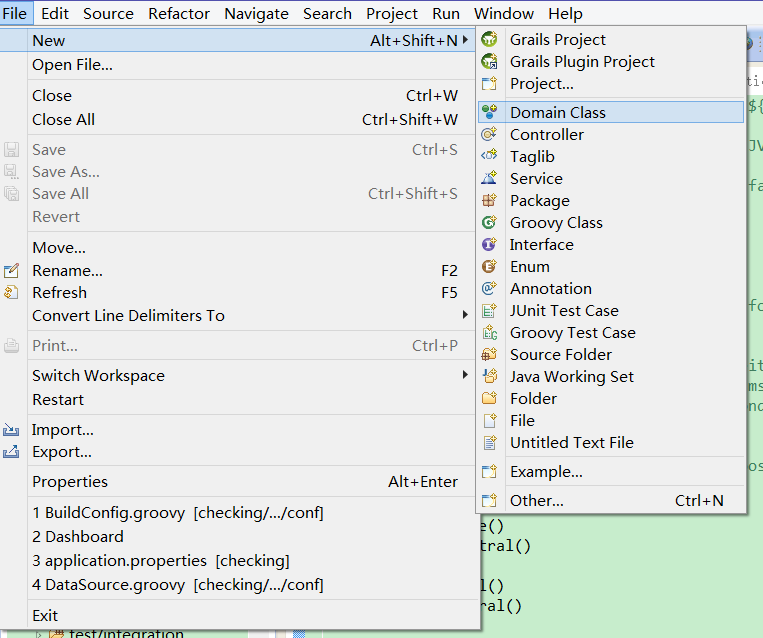
|  |
| --- |
| grails create-domain-class helloworld.Person |

如果class的包名没有设置，则默认使用application名作为包名。

上面这个命令就创建一个domain类：grails-app/domain/helloworld/Persion.groovy。

如果在DataSource.groovy的配置中设置dbCreate为update、cteate或者create-drop，那么grails就会自动在数据库中创建或者修改domain class对应的表。

同样适用ggts可以创建domain class：



Persion.groovy的代码如下：

|  |
| --- |
| package helloworld  class Person {  } |

这时，就可以在domain class中增加属性

|  |
| --- |
| class Person {  String name  Integer age  Date lastVisit  } |

### 基本的GRUD操作

#### Create

创建一个domain class可以使用map作为构造函数的参数，同时调用save方法。save方法会将domain class的实例持久化到数据库中。

|  |
| --- |
| def p = new Person(name: "Fred", age: 40, lastVisit: new Date())  p.save() |

#### Read

Grails自动会给domain class增加一个隐含的id属性，可以使用这个id属性从数据库表中读取出domain class实例。

|  |
| --- |
| def p = Person.get(1) //使用get方法//使用get方法  def list = Person.getAll(1，2，3) //使用getAll获取一组数据  def p = Person.read(1)//使用read方法获取一个只读的版本  def p = Person.load(1)/\*获得一个domain class的proxy，只有当除了getId以外的任何一个方法被调用的时候，grails才会真正从数据库中读取这个domain class（lazy read）\*/  assert 1 == p.id |

#### Update

更新domain class实例的属性值，然后调用save

|  |
| --- |
| def p = Person.get(1)  p.name = "Bob"  p.save() |

#### Delete

直接在domain class实例上调用delete方法

|  |
| --- |
| def p = Person.get(1)  p.delete() |

## 建模

使用grails创建应用重点考虑的是对业务领域模型进行建模。比如要创建一个在线书店的应用，设计到的领域模型有book、author、customer和publisher。每个领域模型都有属性。领域模型对应数据库表，而属性对应表的字段。

建模完成，就可以创建对应的domain class了。例如：

|  |
| --- |
| grails create-domain-class org.bookstore.Book |

创建源文件为：grails-app/domain/org/bookstore/Book.groovy domain class，代码如下：

|  |
| --- |
| package org.bookstore  class Book {  } |

这个class自动对应数据库表book。

现在可以给class增加properties，即对应表的字段。

|  |
| --- |
| package org.bookstore  class Book {  String title  Date releaseDate  String ISBN  } |

三个属性分别对应字段titile、release\_date和isbn。这个属性和字段的转换是grails的默认行为，可以在Constraints中进行定制。

### 关系

关系定义了domain class之间如何关联。除非在关系两头的domain class都显示的定义，否则关系只存在定义它的domain class上。

#### 多对一和一对一

多对一是最简单的关系类型，通过一个其他domain class类型的属性来定义。例如

|  |
| --- |
| class Face {  Nose nose  } |
| class Nose {  } |

这个例子定义了一个单向的从Face到Nose的多对一的关系。

双向的多对一关系

|  |
| --- |
| class Face {  Nose nose  } |
| class Nose {  static belongsTo = [face:Face]  } |

这个例子使用belongsTo来说明Nose属于Face，在数据库中会在表face中创建一个对表nose的外键关联nose\_id。

GROM会给Nose类增加一个名为face的Face类型的属性。

如果不希望增加这个属性，可以使用：static belongsTo = Face。

这样当我们有了一个Face的实例并调用save或者delete方法时，grails也会save或者delete这个face所拥有的nose。也就是说更新是级联的。

|  |
| --- |
| new Face(nose:new Nose()).save()//同时在face和nose表中新增一条记录  new Nose(face:new Face()).save()//反过来执行会报错  def f = Face.get(1)  f.delete()//同时删除face和nose中的记录 |

使用hasOne可以使关系变成一对一的。

|  |
| --- |
| class Face {  static hasOne = [nose:Nose]  } |
| class Nose {  Face face  } |

hasOne是一种双向的关系，在数据库中表nose会创建一个对表face的外键关联face\_id。

最后，可以增加一个唯一约束。

|  |
| --- |
| class Face {  static hasOne = [nose:Nose]  static constraints = {  nose unique: true  }  } |
| class Nose {  Face face  } |

**控制关联的两端**

有时候关系建立在相同类型的domain class上，例如：

|  |
| --- |
| class Person {  String name  Person parent  static belongsTo = [ supervisor: Person ]  static constraints = { supervisor nullable: true }  } |

#### 一对多

一对多是指一个class实例拥有另一个class的多个实例。grails使用hasMany来定义这种关系。

|  |
| --- |
| class Author {  static hasMany = [books: Book]  String name  } |
| class Book {  String title  } |

这就定义了一个单向的一对多的关系。grails使用关系表来映射这种关系。

grails会自动给定义hasMany的domain class注入一个类型为java.util.Set的属性，所以可以使用如下的代码：

|  |
| --- |
| def a = Author.get(1)  for (book in a.books) {  println book.title  } |

注意grails默认使用lazy加载，也就说只有真正使用到book时才从数据库中读取记录。这会导致N+1问题。

默认的级联行为是save和update。如果想要使delete行为也能级联，那么就需要在关系的另一头使用belongsTo

|  |
| --- |
| class Author {  static hasMany = [books: Book]  String name  } |
| class Book {  static belongsTo = [author: Author]  String title  } |

如果在一对多的多这一头的domain class中有多个相同类型的属性，那么就要使用mappedBy来明确那个collection会被映射到。例如：

|  |
| --- |
| class Airport {  static hasMany = [flights: Flight]  static mappedBy = [flights: "departureAirport"]  } |
| class Flight {  Airport departureAirport  Airport destinationAirport  } |

同样，也可以用mappedBy定义多个一对多关系

|  |
| --- |
| class Airport {  static hasMany = [outboundFlights: Flight, inboundFlights: Flight]  static mappedBy = [outboundFlights: "departureAirport",  inboundFlights: "destinationAirport"]  } |
| class Flight {  Airport departureAirport  Airport destinationAirport  } |

#### 多对多

在关系的两头都使用hasMany并且在拥有这个关系的一头定义了belongsTo那么就定义了一个多对多的关系。

|  |
| --- |
| class Book {  static belongsTo = Author  static hasMany = [authors:Author]  String title  } |
| class Author {  static hasMany = [books:Book]  String name  } |

grails使关系表来映射多对多的关系。拥有关系的头端，也就是定义了belongsTo的那头，负责将这个关系映射到数据库中。而且只有发生在拥有关系的头端的行为才会级联。例如下面的代码会级联保存

|  |
| --- |
| new Author(name:"Stephen King")  .addToBooks(new Book(title:"The Stand"))  .addToBooks(new Book(title:"The Shining"))  .save() |

但是这个就不会级联保存

|  |
| --- |
| new Book(name:"Groovy in Action")  .addToAuthors(new Author(name:"Dierk Koenig"))  .addToAuthors(new Author(name:"Guillaume Laforge"))  .save() |

#### 基本集合类型

一般来讲关联都是表示两个不同的domain class之间的关系，但是grails也可以使用基本的java类型集合来表示关系。例如：

|  |
| --- |
| class Person {  static hasMany = [nicknames: String]  } |

grails使用关系表来映射这种关系。可以使用joinTable来定义关系表。

|  |
| --- |
| class Person {  static hasMany = [nicknames: String]  static mapping = {  hasMany joinTable: [name: 'bunch\_o\_nicknames',  key: 'person\_id',  column: 'nickname',  type: "text"]  }  } |

这样就会在数据库中创建一个代表这种关系的关系表，主键是persion\_id，字段是nickname，字段类型是字符串。

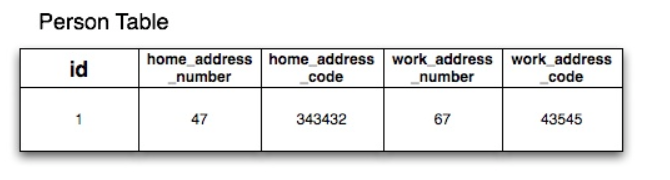
|  |
| --- |
| ---------------------------------------------  | person\_id | nickname |  ---------------------------------------------  | 1 | Fred |  --------------------------------------------- |

### GORM中的合成（composition）

不同于关联（association）将关系对象映射成不同的表，grails也支持合成（composition）即将关系对象映射到一个表中。例如：

|  |
| --- |
| class Person {  Address homeAddress  Address workAddress  static embedded = ['homeAddress', 'workAddress']  }  class Address {  String number  String code  } |

会被映射成如下的数据库表



注意使用合成时，domain class要定义在一个groovy源文件内。

### GORM中的继承

domain class也支持继承。grails默认使用table-per-hierarchy，就是说一个继承层级中的所有class都映射成一个表。

table-per-hierarchy有个不足之处就是对于子类字段不能定义成non-nullable。grails的ORM DSL也提供了table-per-subclass的方式，即父类和子类都各自映射一个数据库表。

|  |
| --- |
| class Payment {  Integer amount  static mapping = {  tablePerHierarchy false  }  }  class CreditCardPayment extends Payment {  String cardNumber  } |

多态（Polymorphic）查询

domain class支持继承关系又一个好处就是查询也能够支持多态。例如，domain class Content有Blog、Book和Podcast三个子类:

|  |
| --- |
| def content = Content.list() // list all blog entries, books and podcasts  content = Content.findAllByAuthor('Joe Bloggs') // find all by author  def podCasts = PodCast.list() // list only podcasts |

### Sets，Lists和Maps

**对象集合**

grails默认使用java.util.Set表示一对多或者多对多关系，集合是没有顺序且不能重复的一组对象。例如：

|  |
| --- |
| class Author {  static hasMany = [books: Book]  } |

Author就有一个类型是java.util.Set<Book>类型的名为books的属性。

Set的特性是不能重复且没有顺序，如果想使用有顺序的Set，那么可以定义如下：

|  |
| --- |
| class Author {  SortedSet books  static hasMany = [books: Book]  } |

同时Book类也需要支持排序

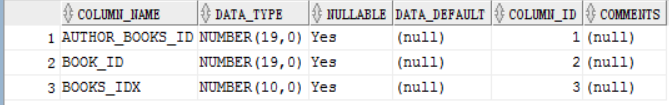
|  |
| --- |
| class Book implements Comparable {  String title  Date releaseDate = new Date()  int compareTo(obj) {  releaseDate.compareTo(obj.releaseDate)  }  } |

**对象列表**

使用List可以是一组对象集合按照加入集合的先后顺序排序，而且可以根据对象的index来访问对象。

|  |
| --- |
| class Author {  List books  static hasMany = [books: Book]  } |

GORM会在关系表author\_book中创建一个books\_idx字段用来记录books的index。



注意当使用List的时候，在save之前list不能为空。

|  |
| --- |
| // This won't work!抛出异常org.hibernate.HibernateException:null index column for collection  def book = new Book(title: 'The Shining')  book.save()  author.addToBooks(book)  // Do it this way instead.  def book = new Book(title: 'Misery')  author.addToBooks(book)  author.save() |

**Bags**

hibernate维持集合对象的唯一性和排序是要付出性能代价的。如果对象集合的排序和唯一性都不需要，那么可以使用hibernate的Bag类型。

|  |
| --- |
| class Author {  Collection books  static hasMany = [books: Book]  } |

**Maps**

使用Map的例子如下：

|  |
| --- |
| class Author {  Map books // map of ISBN:book names  }  def a = new Author()  a.books = ["1590597583":"Grails Book"]  a.save() |

注意：map的key和value必须是String类型。

如果想要value类型是Object，则可以如下：

|  |
| --- |
| class Book {  Map authors  static hasMany = [authors: Author]  }  def a = new Author(name:"Stephen King")  def book = new Book()  book.authors = [stephen:a]  book.save() |

**集合类型和性能**

由于Set中的元素是唯一性的，而List中的元素是有序的，hibernate为了保证唯一性和有序，需要把整个集合的所有元素都从数据库中读取出来。这样当这个集合非常大的时候，就会引起性能问题。

所以，如果你使用的集合中有大量的元素，建议是关系的关联变成双向的，这样就可以从关系的另一头来创建对象。例如下面的代码：

|  |
| --- |
| def book = new Book(title:"New Grails Book")  def author = Author.get(1)  book.author = author  book.save() |

性能要优于

|  |
| --- |
| def book = new Book(title:"New Grails Book")  def author = Author.get(1)  author.addToBooks(book)  author.save() |

## 持久化的基本概念

grails使用hibernate来保存数据，所以理解hibernate的session的概念在grails中也同样重要，没有很好的理解hibernate的基本概念，往往会导致应用程序产生严重的性能问题。

### hibernate session（持久化上下文）

Session，可以看作是一个容器，拥有设计到的domain对象的引用。session的职责就是使这些对象的状态保持和数据库中的数据同步一致。

session也可以看做是一个缓存，事实上session就是hibernate的一级缓存。如果session中的对象和数据库中的一致，那么session直接返回缓存中的对象给请求方，而不会再一次从数据库中去读取数据。

同步的过程由session的flush方法触发。

grails为每一个http request都绑定一个session，所以可以直接在代码中使用save或者delete等GORM的方法。

事务性的延迟写

hibernate有一区别于JDBC和其他ORM框架的特性，就是当你调用save或者delete等方法时比没有立刻执行任何sql语句。事实是，hibernate将这些sql语句打包在一起，并且尽可能的延迟执行。grails也同样具有这个特性。

也就说，hibernate将数据库更新都缓存起来，只在认为需要session.flush的时候才将这些更新一次性的更新到数据库中。发生session.flush的通常情况是：

* 在controller的action执行完毕，页面返回之后
* 当某次查询其查询结果的数据已经有版本在session缓存中的时候。对于数据库读写频繁的应用，这种特性能大大提高性能。
* 在事务提交之前。

注意，session的flush同事务提交并不是一回事。当数据库操作包含在事务的上下文中的时候，session.flush会立刻执行sql语句，但是数据库会将这些更新缓存在事务队列中，只有当事务提交的时候才真正的将事务队列中的更新提交到数据库中。

#### 直接使用session

session是一个缓存，如果需要持久化的对象数据非常庞大，而session又没有及时清空的话，会导致应用产生性能问题，甚至直接OOM。这时就需要在使用的过程中手工清除session。

首先需要等到一个当前session的引用。GORM通过sessionFactory来获取session。利用grails的依赖注入，只需要在controller或者service中声明一个sessionFactory的成员变量就可以得到sessionFactory的实例，然后就可以获取当前session。

|  |
| --- |
| def sessionFactory  ...  def index() {  def session = sessionFactory.currentSession()  } |

使用domain对象的withSession方法也可以获取当前的session实例，然后对session进行操作。

|  |
| --- |
| def index() {  Album.withSession { session ->  def allAlbums = Album.list()  for(album in allAlbums) {  def songs = Song.findAllByAlbum(album)  // do something with the songs  ...  session.clear()  }  }  } |

### 理解GORM中的事务

不管什么情况GORM和数据库之间的交互都是在一个事务之内。grails在controller收到一个http请求之后就会创建并打开一个session，当session第一次和数据库进行交互的时候，grails就会打开一个事务。

一个session对应一个jdbcConnection，所以默认情况下session和一个jdbc事务相关联，当jdbc连接关闭则事务提交、session关闭，或者中间发生一场，则事务回滚。

当手工调用了session.flush方法后，则情况就变得不同了。假如如下代码：

|  |
| --- |
| def save() {  def album = Album.get(params.id)  album.title = "Changed Title"  album.save(flush:true)  // something goes wrong  throw new Exception("Ruh Roh!")  } |

即使异常抛出事务回滚，但是由于调用save(flush.true)更新还是已经同步到了数据库中，并没有被回滚掉。

解决这个问题的办法之一就是使用withTransaction明确的声明事务边界

|  |
| --- |
| def save() {  Album.withTransaction {  def album = Album.get(params.id)  album.title = "Changed Title"  album.save(flush:true)  ...  // something goes wrong  throw new Exception("Ruh Roh!")  }  } |

另外也可以将代码移动到支持事务的service中。

这种情况下，Grails底层使用spring的PlatformTransactionManager来管理事务。

### 保存和更新

一个保存的操作可以是如下代码：

|  |
| --- |
| def p = Person.get(1)  p.save() |

保存操作并不会立即被推送到数据库中，只有当session.flush调用的时候保存才会被推送到数据库。但是可以通过代码来控制session的flush。

|  |
| --- |
| def p = Person.get(1)  p.save(flush: true) |

这样sql语句会被立即执行，更新会被立即推送到数据库中。

当使用hibernate的乐观锁机制的时候，flush：true还可以用来获取异常信息。

|  |
| --- |
| def p = Person.get(1)  try {  p.save(flush: true)  }  catch (org.springframework.dao.DataIntegrityViolationException e) {  // deal with exception  } |

保存和更新时，grails还会对domain对象进行数据校验，可以使用failOnError：true来捕获校验出错的信息，并返回更加友好的信息给用户。

|  |
| --- |
| def p = Person.get(1)  try {  p.save(failOnError: true)  }  catch (ValidationException e) {  // deal with exception  } |

### 删除

删除的例子如下：

|  |
| --- |
| def p = Person.get(1)  p.delete() |

同样使用flush：true可以立即从数据库中删除记录。如果删除违反了外键关联等约束，可以通过捕获异常来获取信息。

|  |
| --- |
| def p = Person.get(1)  try {  p.delete(flush: true)  }  catch (org.springframework.dao.DataIntegrityViolationException e) {  flash.message = "Could not delete person ${p.name}"  redirect(action: "show", id: p.id)  } |

注意，grails比没有提供deleteAll方法来批量删除数据。这是因为grails更加推荐置标志位的方式逻辑删除数据。如果确实需要批量删除数据，可以直接使用DML语句。

|  |
| --- |
| Customer.executeUpdate("delete Customer c where c.name = :oldName",  [oldName: "Fred"]) |

### 理解更新和删除中的级联行为

理解级联更新和级联删除行为对使用GORM至关重要。GORM的belongsTo控制了哪个class拥有关系。

对于一对一、一对多和多对多的关系，belongsTo可以让更新操作从拥有关系的class级联到关系另一头的class上，对于多对一和一对多还可以级联删除。

如果没有定义belongsTo，那么就不会有级联行为发生（有一个例外，就是在一对多的关系中，如果新建的对象有一个hasMany集合，那么会发生级联更新）。

下面的例子使用了hasMany和belongsTo：

|  |
| --- |
| class Airport {  String name  static hasMany = [flights: Flight]  } |
| class Flight {  String number  static belongsTo = [airport: Airport]  } |

则如下的代码会同时保持所有的Airport和Flight对象到数据库中

|  |
| --- |
| new Airport(name: "Gatwick")  .addToFlights(new Flight(number: "BA3430"))  .addToFlights(new Flight(number: "EZ0938"))  .save() |

同样，如果删除这个Airport对象，那么所有关联到的Flight对象所映射的记录也会从数据库中删除。

|  |
| --- |
| def airport = Airport.findByName("Gatwick")  airport.delete() |

#### 带有belongsTo、双向的一对多关系

|  |
| --- |
| class A { static hasMany = [bees: B] } |
| class B { static belongsTo = [a: A] } |

一到多：save-update-delete级联，多到一：没有级联

#### 没有belongsTo，单向的一对多关系

|  |
| --- |
| class A { static hasMany = [bees: B] } |
| class B { } |

一到多：save-update级联

#### 没有belongsTo，双向的一对多关系

|  |
| --- |
| class A { static hasMany = [bees: B] } |
| class B { A a } |

一到多：save-update级联，多到一：没有级联

#### 带有belongsTo，单向的一对多关系

|  |
| --- |
| class A { } |
| class B { static belongsTo = [a: A] } |

一到多：save-update-delete级联

### 立即取（Eager Fetching）和延迟取（Lazy Fetching）

下面的例子解释了什么是延迟取

|  |
| --- |
| class Airport {  String name  static hasMany = [flights: Flight]  } |
| class Flight {  String number  Location destination  static belongsTo = [airport: Airport]  } |
| class Location {  String city  String country  } |
| def airport = Airport.findByName("Gatwick")  for (flight in airport.flights) {  println flight.destination.city  } |

grails使用一句sql获取Airport实例，一句sql获取跟这个airport关联的所有的Flight实例集合，然后在遍历flights的每个循环内都执行一条sql获取真正的flight实例的Location属性。这就是N+1查询。

#### 配置立即取（Eager Fetching）模式

有时N+1查询会带来性能问题。所以grails提供了配置成立即查询的途径。上面的例子中的Airport可以写成这样：

|  |
| --- |
| class Airport {  String name  static hasMany = [flights: Flight]  static mapping = {  flights lazy: false  }  } |

这样第一条sql获取airport，第二条sql获取相关联的所有的flight实例。也可以使用fetch:'join'，通过关联表获取记录。这样只需要执行一条sql就可以了。但是需要注意GROM使用的left out join，所以大数量情况下可能会有性能问题。

一个比较好的建议是：fetch: 'join' for single-ended association ;lazy: false for one-to-manys

|  |
| --- |
| class Airport {  String name  static hasMany = [flights: Flight]  static mapping = {  flights fetch: 'join'  }  } |

#### 使用批量获取（Batch fetching）

尽管立即取能获得不错的性能，但是一不小心，就有可能把整个数据库的数据都读到内存中。所以为了及考虑系能，又能避免内存问题，还可以使用批量获取（Batch fetching）的方式。

|  |
| --- |
| class Airport {  String name  static hasMany = [flights: Flight]  static mapping = {  flights batchSize: 10  }  } |

这样，GORM一次获取10个flight的实例。加入某个airport实例关联30个flight实例，那么如果没有配置batchSize:10，那么GORM使用一条sql获取airport实例，然后在每一次循环内都使用一条sql获取一个flight实例，总共需要执行31条sql。如果配置了batchSize:10，那么GORM会使用一条sql获取airport实例，接下来一条sql获取10个flight实例，遍历完后再获取10个flight实例。这样总共需要执行的sql只有四条。

批量获取可以看作是延迟取得优化方式。批量获取也可以配置到class级别。

|  |
| --- |
| class Flight {  …  static mapping = {  batchSize 10  }  } |

### 悲观锁和乐观锁

#### 乐观锁

乐观锁是hibernate的一种机制。hibernate在表中增加一个名为version的字段，这个字段在记录每次更新后加1。

GORM默认使用乐观锁机制。

domain对象都有一个掩藏的version属性，可以通过它来获取当前记录的版本信息。

|  |
| --- |
| def airport = Airport.get(10)  println airport.version |

GORM更新时发现version值和表中的version字段值不一致时，就会抛出异常一个运行时异常，并导致事务回滚。

在代码中可以通过flush：true来捕获这个异常。

|  |
| --- |
| def airport = Airport.get(10)  try {  airport.name = "Heathrow"  airport.save(flush: true)  }  catch (org.springframework.dao.OptimisticLockingFailureException e) {  // deal with exception  // merge或者通知用户冲突信息  } |

#### 悲观锁

悲观锁等同于"select \* for update"，行锁，会造成其他线程等待解锁。

GORM使用lock方法进行悲观锁。

|  |
| --- |
| def airport = Airport.get(10)  airport.lock() // lock for update  airport.name = "Heathrow"  airport.save() |

事务提交则自动解锁。

注意，在执行get方法和执行lock方法之间，其他线程还是有机会更新记录的。可以使用static lock方法来规避这个问题。

|  |
| --- |
| def airport = Airport.lock(10) // lock for update  airport.name = "Heathrow"  airport.save() |

也可以有其他方式来获取一个悲观锁。

|  |
| --- |
| def airport = Airport.findByName("Heathrow", [lock: true]) |
| def airport = Airport.createCriteria().get {  eq('name', 'Heathrow')  lock true  } |

#### 比较

乐观锁性能较好，但是需要新增一个version字段，需要自行处理异常。

悲观锁能保证数据一致性，但是造成其它线程等待，性能较差。

### 检查是否有改动

当执行get方法获取一个 domain对象时，并不是一定是从数据库中载入这个对象。hibernate首先会检查当前session的缓存中是否有该对象，如果有则直接返回，没有才从数据库中载入。

当执行其他query方法时，而该query方法返回的对象已经存在session的缓存中的时候，hibernate则会flush当前的session，在执行从数据库中载入数据的操作。而这个时候有可能并不希望更新被flush到数据中。

#### isDirty

isDirty方法用来检测对象是否已经被修改过。

|  |
| --- |
| def airport = Airport.get(10)  assert !airport.isDirty()  airport.properties = params  if (airport.isDirty()) {  // do something based on changed state  } |

也可以判断某个属性是否已被更新。

|  |
| --- |
| def airport = Airport.get(10)  assert !airport.isDirty()  airport.properties = params  if (airport.isDirty('name')) {  // do something based on changed name  } |

可以使用getDirtyPropertiyNames来获取被更新的属性。

|  |
| --- |
| def airport = Airport.get(10)  assert !airport.isDirty()  airport.properties = params  def modifiedFieldNames = airport.getDirtyPropertyNames()  for (fieldName in modifiedFieldNames) {  // do something based on changed value  } |

可以使用getPersistentValue来获取被更新字段的原有值。

|  |
| --- |
| def airport = Airport.get(10)  assert !airport.isDirty()  airport.properties = params  def modifiedFieldNames = airport.getDirtyPropertyNames()  for (fieldName in modifiedFieldNames) {  def currentValue = airport."$fieldName"  def originalValue = airport.getPersistentValue(fieldName)  if (currentValue != originalValue) {  // do something based on changed value  }  } |

## 查询

GORM提供了四种方式来查询数据，包括：

* Dynamic Finders
* Where Queries
* Criteria Queries
* Hibernate Query Language (HQL)

### 基础方法

首先是一些查询数据的基础方法。

#### List

|  |
| --- |
| def books = Book.list()//查出所有的book数据 |
| def books = Book.list(offset:10, max:20)//查出第11-20条的数据 |
| def books = Book.list(sort:"title", order:"asc")//查出数据并排序 |

list方法可以接受分页所需的参数:max、offset、sort、order，参数以Map的形式传入

#### Get

get方法根据ID获取数据

|  |
| --- |
| def book = Book.get(23)//获取id为23的记录 |
| def books = Book.getAll(23, 93, 81)//获取id分别为23、93、81的记录 |

### Dynamic finder

dynamic finder的方法执行起来有点像执行静态方法，但是dynamic finder方法在代码层面是不存在的。这些方法在运行时动态生成。

dynamic finder的方法名具有查询语意，通常以findBy或者findAllBy开头，后面跟方法表达式（method expressions）。

例如，有如下两个domain class：

|  |
| --- |
| class Book {  String title  Date releaseDate  Author author  } |
| class Author {  String name  } |

则可以使用下面的dynamic finder方法：

|  |
| --- |
| def book = Book.findByTitle("The Stand")  book = Book.findByTitleLike("Harry Pot%")  book = Book.findByReleaseDateBetween(firstDate, secondDate)  book = Book.findByReleaseDateGreaterThan(someDate)  book = Book.findByTitleLikeOrReleaseDateLessThan("%Something%", someDate) |

#### 方法表达式（method expressions）

方法表达式的基本格式是：

|  |
| --- |
| Book.findBy([Property][Comparator][Boolean Operator])?[Property][Comparator] |

可用的comparator有：

* InList - In the list of given values
* LessThan - less than a given value
* LessThanEquals - less than or equal a give value
* GreaterThan - greater than a given value
* GreaterThanEquals - greater than or equal a given value
* Like - Equivalent to a SQL like expression
* Ilike - Similar to a Like, except case insensitive
* NotEqual - Negates equality
* InRange - Between the from and to values of a Groovy Range
* Rlike - Performs a Regexp LIKE in MySQL or Oracle otherwise falls back to Like
* Between - Between two values (requires two arguments)
* IsNotNull - Not a null value (doesn't take an argument)
* IsNull - Is a null value (doesn't take an argument)

注意，最后三个comparator的参数个数比较特殊。例如：

|  |
| --- |
| def now = new Date()  def lastWeek = now - 7  def book = Book.findByReleaseDateBetween(lastWeek, now)  books = Book.findAllByReleaseDateIsNull()  books = Book.findAllByReleaseDateIsNotNull() |

可用的boolean operator有：AND/OR。

例如：

|  |
| --- |
| def books = Book.findAllByTitleLikeAndReleaseDateGreaterThan(  "%Java%", new Date() - 30) |
| def books = Book.findAllByTitleLikeOrReleaseDateGreaterThan(  "%Java%", new Date() - 30) |

注意：AND和OR不能同时使用。

dynamic query也支持联合查询，例如：

|  |
| --- |
| def author = Author.findByName("Stephen King")  def books = author ? Book.findAllByAuthor(author) : [] |

用dynamic query进行分页查询和排序也很简单:

|  |
| --- |
| def books = Book.findAllByTitleLike("Harry Pot%",  [max: 10, offset: 10, sort: "title", order: "desc"]) |

### Where Queries

domain的where方法使用Detached criteria执行查询。where方法提供了比dynamic query更大的灵活性，同时又比criteria查询简单，并且能够很好的支持组合查询。

#### 基本查询

where方法接受一个closure作为参数，在closure内定义查询所需的条件。

|  |
| --- |
| def query = Person.where {  firstName == "Bart"  }  Person bart = query.find() |

where方法返回一个DetachedCriteria对象，返回对象并不依存于任何数据库连接或者hibernate session，所以可以用来作为类的成员变量。

|  |
| --- |
| class Person {  static simpsons = where {  lastName == "Simpson"  }  …  }  …  Person.simpsons.each {  println it.firstname  } |

每一个groovy操作符都对应一个criteria查询的方法：

|  |  |  |
| --- | --- | --- |
| **Operator** | **Criteria Method** | **Description** |
| **==** | eq | Equal to |
| **!=** | ne | Not equal to |
| **>** | gt | Greater than |
| **<** | lt | Less than |
| **>=** | ge | Greater than or equal to |
| **<=** | le | Less than or equal to |
| **in** | inList | Contained within the given list |
| **==~** | like | Like a given string |
| **=~** | ilike | Case insensitive like |

使用where可以创建很复杂的查询：

|  |
| --- |
| def query = Person.where {  (lastName != "Simpson" && firstName != "Fred") || (firstName == "Bart" && age > 9)  }  def results = query.list(sort:"firstName") |

where的like查询也支持正则表达式（==~ Match符，~ Pattern符）

|  |
| --- |
| def query = Person.where {  firstName ==~ ~/B.+/  } |

其他

|  |
| --- |
| def query = Person.where {  age in 18..65 //between查询  } |
| def query = Person.where {  middleName == null //is null 或者is not null  } |

#### 组合查询

Detached criteria可以组合使用。

|  |
| --- |
| def query = Person.where {  lastName == "Simpson"  }  def bartQuery = query.where {  firstName == "Bart"  }  Person p = bartQuery.find() |

不能直接将一个closure作为参数传递给where方法，下面的用法是错误的：

|  |
| --- |
| def callable = {//错误的用法  lastName == "Simpson"  }  def query = Person.where(callable) |

显式的将closure声明成一个DetachedCriteria对象后，就可以作为参数传递给where方法。

|  |
| --- |
| import grails.gorm.DetachedCriteria  def callable = {  lastName == "Simpson"  } as DetachedCriteria<Person>  def query = Person.where(callable) |

#### 合取、析取和否定（Conjunction, Disjunction and Negation）

|  |
| --- |
| def query = Person.where {  (lastName != "Simpson" && firstName != "Fred") || (firstName == "Bart" && age > 9)  } |
| def query = Person.where {  firstName == "Fred" && !(lastName == 'Simpson')  } |

常用的属性比较方法对应的运算符如下：

|  |  |  |
| --- | --- | --- |
| **Operator** | **Criteria Method** | **Description** |
| **==** | eqProperty | Equal to |
| **!=** | neProperty | Not equal to |
| **>** | gtProperty | Greater than |
| **<** | ltProperty | Less than |
| **>=** | geProperty | Greater than or equal to |
| **<=** | leProperty | Less than or equal to |

#### 关联查询

可以使用associateObject.propertyName进行关联查询，例如：

|  |
| --- |
| def query = Pet.where {  owner.firstName == "Joe" || owner.firstName == "Fred"  } |
| def query = Person.where {  pets { name == "Jack" || name == "Joe" }  } |
| def query = Person.where {  pets { name == "Jack" } || firstName == "Ed"  } |
| def query = Person.where {  pets.size() == 2  } |

对于size方法支持的操作符有：

|  |  |  |
| --- | --- | --- |
| **Operator** | **Criteria Method** | **Description** |
| **==** | sizeEq | The collection size is equal to |
| **!=** | sizeNe | The collection size is not equal to |
| **>** | sizeGt | The collection size is greater than |
| **<** | sizeLt | The collection size is less than |
| **>=** | sizeGe | The collection size is greater than or equal to |
| **<=** | sizeLe | The collection size is less than or equal to |

#### 子查询

在where方法内还可以进行子查询

|  |
| --- |
| final query = Person.where {  age > avg(age)  } |

支持的子查询有：

|  |  |
| --- | --- |
| **Method** | **Description** |
| **avg** | The average of all values |
| **sum** | The sum of all values |
| **max** | The maximum value |
| **min** | The minimum value |
| **count** | The count of all values |
| **property** | Retrieves a property of the resulting entities |

可以使用of进行复杂的子查询

|  |
| --- |
| def query = Person.where {  age > avg(age).of { lastName == "Simpson" } && firstName == "Homer"  } |
| Person.where {  age < property(age).of { lastName == "Simpson" }  } |

#### 其他函数

where还支持使用的其它函数有：

|  |  |
| --- | --- |
| **Method** | **Description** |
| **second** | The second of a date property |
| **minute** | The minute of a date property |
| **hour** | The hour of a date property |
| **day** | The day of the month of a date property |
| **month** | The month of a date property |
| **year** | The year of a date property |
| **lower** | Converts a string property to upper case |
| **upper** | Converts a string property to lower case |
| **length** | The length of a string property |
| **trim** | Trims a string property |

举例：

|  |
| --- |
| def query = Pet.where {  year(birthDate) == 2011  } |
| def query = Person.where {  year(pets.birthDate) == 2009  } |

#### 批量更新和删除

因为where方法返回的是一个DetachedCriteria对象，所以可以用来进行批次更新和删除。例如：

|  |
| --- |
| def query = Person.where {//批量更新  lastName == 'Simpson'  }  int total = query.updateAll(lastName:"Bloggs") |
| def query = Person.where {//批量删除  lastName == 'Simpson'  }  int total = query.deleteAll() |

### Criteria

Criteria其实就是hibernate的criteriaAPI进行查询。它可以进行很复杂的查询，相比使用StringBuffer拼凑查询语句，criteria是一种更好的方式。

Criteria可以使用createCriteria或者withCriteria方法查询，底层是使用hibernate criteria API中的Restrictions的静态方法执行查询。例如：

|  |
| --- |
| def c = Account.createCriteria()  def results = c {  between("balance", 500, 1000)  eq("branch", "London")  or {  like("holderFirstName", "Fred%")  like("holderFirstName", "Barney%")  }  maxResults(10)  order("holderLastName", "desc")  } |

注意：如果查询结果为空，results就是一个空list。

#### 合取、析取和否定（Conjunction, Disjunction and Negation）

|  |
| --- |
| or {  between("balance", 500, 1000)  eq("branch", "London")  } |
| and {  between("balance", 500, 1000)  eq("branch", "Shanghai")  } |
| not {  between("balance", 500, 1000)  eq("branch", "London")  } |

上面上个条件的sql含义：

(t.balance between 500 and 1000 or t.branch = 'London') and (t.balance between 500 and 1000 and t.branch = 'Shanghai') and (t.balance not between 500 and 1000 and t.branch <> 'London')

#### 关联查询

同样criteria也可以使用属性进行关联查询，例如：

|  |
| --- |
| class Account {//定义一个Account domain类，其中有一个属性名是transactions的关联  …  static hasMany = [transactions: Transaction]  …  } |
| def c = Account.createCriteria()////查出所有在最近十天内有过交易的账户  def now = new Date()  def results = c.list {  transactions {  between('date', now - 10, now)  }  } |
| def c = Account.createCriteria()////查出所有在最近十天内有过交易的或者在十天内开户的账户  def now = new Date()  def results = c.list {  or {  between('created', now - 10, now)  transactions {  between('date', now - 10, now)  }  }  } |

#### 使用投影（Projections）进行查询

本质就是使用hibernate的[Projections](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/criterion/Projections.html) 类进行查询。

|  |
| --- |
| def c = Account.createCriteria()  def numberOfBranches = c.get {  projections {  countDistinct('branch')  }  } |

#### 使用SQL投影（Projections）进行查询

本质就是使用hibernate的SQL projection API。

|  |
| --- |
| // Box is a domain class…  class Box {  int width  int height  } |
| // 得到所有box的周长和面积  def c = Box.createCriteria()  def results = c.list {  projections {  sqlProjection '(2 \* (width + height)) as perimeter, (width \* height) as area', ['perimeter', 'area'], [INTEGER, INTEGER]  }  } |

假设有如下的box数据：

|  |  |
| --- | --- |
| **width** | **height** |
| 2 | 7 |
| 2 | 8 |
| 2 | 9 |
| 4 | 9 |

那么返回的数据就是：

|  |
| --- |
| [[18, 14], [20, 16], [22, 18], [26, 36]] |

GROM的DSL也支持GroupProjection：

|  |
| --- |
| def results = c.list {  projections {  sqlGroupProjection 'width, sum(height) as combinedHeightsForThisWidth', 'width', ['width', 'combinedHeightsForThisWidth'], [INTEGER, INTEGER]  }  } |

第二个参数就表示group by width。这个查询返回的结果是：

|  |
| --- |
| [[2, 24], [4, 9]] |

#### 使用SQL Restrictions进行查询

本质上就是使用hibernate的SQL Restrictions API。

|  |
| --- |
| def c = Person.createCriteria()  def peopleWithShortFirstNames = c.list {  sqlRestriction "char\_length(first\_name) <= 4"  } |
| def c = Person.createCriteria()  def peopleWithShortFirstNames = c.list {  sqlRestriction "char\_length(first\_name) < ? AND char\_length(first\_name) > ?", [maxValue, minValue]  } |

#### 使用Scrollable结果集

GROM的Criteria可以使用hibernate的ScrollableResults类在返回结果中滚动。

|  |
| --- |
| def results = crit.scroll {  maxResults(10)  }  def f = results.first()  def l = results.last()  def n = results.next()  def p = results.previous()  def future = results.scroll(10)  def accountNumber = results.getLong('number') |

#### 立即取（Eager Fetching）

criteria也可以使用立即取避免N+1问题。

使用join：

|  |
| --- |
| def criteria = Task.createCriteria()  def tasks = criteria.list{  eq "assignee.id", task.assignee.id  join 'assignee'  join 'project'  order 'priority', 'asc'  } |

使用select

|  |
| --- |
| import org.hibernate.FetchMode as FM  …  def results = Airport.withCriteria {  eq "region", "EMEA"  fetchMode "flights", FM.SELECT  } |

如果查询本身就使用了关联属性，那么就是自动会使用立即取查询数据

|  |
| --- |
| def results = Airport.withCriteria {  eq "region", "EMEA"  flights {  like "number", "BA%"  }  } |

#### 可使用的方法

criteria默认使用list方法，所以下面两个是等效的。

|  |
| --- |
| c { … } |
| c.list { … } |

除了list其它的方法有：

|  |  |
| --- | --- |
| **Method** | **Description** |
| **list** | This is the default method. It returns all matching rows. |
| **get** | Returns a unique result set, i.e. just one row. The criteria has to be formed that way, that it only queries one row. This method is not to be confused with a limit to just the first row. |
| **scroll** | Returns a scrollable result set. |
| **listDistinct** | If subqueries or associations are used, one may end up with the same row multiple times in the result set, this allows listing only distinct entities and is equivalent toDISTINCT\_ROOT\_ENTITY of the [CriteriaSpecification](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/criterion/CriteriaSpecification.html) class. |
| **count** | Returns the number of matching rows. |

#### 组合使用criteria

criteria可以组合使用，例如：

|  |
| --- |
| def emeaCriteria = {  eq "region", "EMEA"  }  def results = Airport.withCriteria {  emeaCriteria.delegate = delegate  emeaCriteria()  flights {  like "number", "BA%"  }  } |

### detached criteria

detached criteria不依赖于某个数据库连接/hibernate session，可以用来创建可重复使用的公共查询，用来执行批量更新和删除。

前面讲到的where查询其实就是使用detached criteria。

### HQL

GROM也支持直接使用HQL进行查询。可以使用的方法有find、findAll和executeQuery。例如：

|  |
| --- |
| def results =  Book.findAll("from Book as b where b.title like 'Lord of the%'") |

#### Positional and Named Parameters

|  |
| --- |
| def results =  Book.findAll("from Book as b where b.title like ?", ["The Shi%"])//通过位置 |
| def author = Author.findByName("Stephen King")  def books = Book.findAll("from Book as book where book.author = ?",  [author])//通过位置，直接传对象 |
| def results =  Book.findAll("from Book as b " +  "where b.title like :search or b.author like :search",  [search: "The Shi%"])//通过名字 |
| def author = Author.findByName("Stephen King")  def books = Book.findAll("from Book as book where book.author = :author",  [author: author])//通过名字，直接传对象 |

#### 拼写多行的hql

多行sql不能使用groovy的'''xxxx'''，但是可以可以使用\表示换行

|  |
| --- |
| def results = Book.findAll("\  from Book as b, \  Author as a \  where b.author = a and a.surname = ?", ['Smith']) |

#### 分页和排序

用于分页和排序的查询条件可以直接作为参数给HQL查询使用。

|  |
| --- |
| def results =  Book.findAll("from Book as b where " +  "b.title like 'Lord of the%' " +  "order by b.title asc",  [max: 10, offset: 20]) |

#### executeQuery

需要注意，HQL的语法和SQL很类似，但是HQL使用的是类名和类的属性名进行查询。

推荐使用executeQuery这个方法来执行HQL查询。不管查询结果是一条还是多条记录，executeQuery都返回一个java.util.ArrayList对象。grails会给所有的domain对象都自动加上这个方法。

一个简单的没有参数的HQL查询可以如下：

|  |
| --- |
| def roles = Role.executeQuery('select r from Role r order by r.name') |

如果查询有参数，可以使用？和一个参数列：

|  |
| --- |
| String loginName = ...  def users = User.executeQuery(  'from User u where u.username=? or u.email=?',  [loginName, loginName]) |

或者使用named parameter加上一个Map：

|  |
| --- |
| String loginName = ...  def users = User.executeQuery(  'from User u where u.username=:login or u.email=:login',  [login: loginName]) |

上面的例子中因为named parameter都是:login，所以只需要在map中放一个参数就行了。

在参数Map中还可以添加如下参数，用于分页等查询：

* max

Limits the maximum number of records to return (typically for pagination)

* offset

Specifies the offset position into the results (typically for pagination)

* readOnly

If true, will return results that are not dirty-checked and whose snapshots of persistent

state are not maintained

* fetchSize

Specifies the fetch size for the underlying JDBC query

* timeout

Specifies the timeout for the underlying JDBC query

* flushMode

Overrides the current session flush mode

* cache

If true, will use the query cache

#### 查询语法

HQL的语法和SQL类似，主要的不同是HQL使用类名和类的属性名，而SQL使用表名和字段名。而且，在HQL中很少直接使用join语法，因为hibernate知道对象之间的关系，它会自动创建使用join的sql。

当查询所有属性的时候，可以省略掉select关键词，也就是说select u from User u和from User u是一样的。当在关联查询而只需要返回某个类的属性的时候，需要使用select，例如：

|  |
| --- |
| select u from User u，Role r where ... |

在HQL中并不需要指定class的包名，因为GORM会自动import所有的domain类。当在不同的包下面有两个相同类名的domain类的时候，这种机制就会出错，这个时候就需要禁用自动import，在domain的mapping块中增加如下代码：

|  |
| --- |
| static mapping = {  autoImport false  } |

然后在HQL中对该domain类使用包名：

|  |
| --- |
| from com.xxx.User u ... |

或者在GString中直接使用：

|  |
| --- |
| From $User.name u ... |

因为User.name就是调用User.class.getName()。

#### Report Query

大多数的HQL查询都返回的是domain对象，但是HQL也可以直接返回属性值。这种查询大多数都用在报表功能中，所以也叫作Report Query。

##### 单个属性

如果只返回单个属性，HQL就返回一个该值的list：

|  |
| --- |
| def firstNames = Author.executeQuery('select a.firstName from Author a') |

上面的代码返回的就是一个String类型的List。

##### 多个属性值

如果返回的是多个属性，则返回类型是一个Object[]的List：

|  |
| --- |
| def names = Author.executeQuery('select a.name, a.age from Author a') |

返回的List中某个Object[]数组的第一个元素是name的值，第二个元素是age的值。

也可以返回一个Map对象的List：

|  |
| --- |
| def names = Author.executeQuery(  'select new map(a.name as fullName, a.age as age) from Author a where ...') |

#### 集合函数

HQL还支持很多集合函数，主要有：

• avg(...)

• sum(...)

• min(...)

• max(...)

• count(\*)

• count(...)

• count(distinct ...)

• count(all ...)

#### 表达式

HQL还支持各种表达式，主要有：

• 数学运算操作符: +,-, \*,/

• 二元比较符: =, >=, <=, <>, !=, like

• and, or, not

• in, not in, between, is null, is not null, is empty, is not empty, member

of, not member of

• 日期和时间函数: current\_date(), current\_time(), current\_time

stamp(), second(...), minute(...), hour(...), day(...), month(...),

year(...)

• EJB-QL 3.0 functions: substring(), trim(), lower(), upper(), length(), lo

cate(), abs(), sqrt(), bit\_length(), mod()

• 集合函数: size(), minelement(), maxelement(), minindex(), maxin

dex(), elements()

#### 集合

在HQL中可以直接使用集合类型，例如：

|  |
| --- |
| def newAuthors = Author.executeQuery(  "select a from Author a where a.books is empty")  def prolificAuthors = Author.executeQuery(  "select a from Author a where size(a.books) > 10")  def grailsAuthors = Author.executeQuery(  "select a from Author a join a.books as book " +  "where lower(book.title) like '%grails%'")  Book book = ...  def author = Author.executeQuery(  "select a from Author a where :book in elements(a.books)", [book: book]) |

## GROM高级功能

### 事件和自动时间戳

在domain class中可以加入下面的方法，这些方法会在对应的事件发生的时候调用。

* beforeInsert - Executed before an object is initially persisted to the database
* beforeUpdate - Executed before an object is updated
* beforeDelete - Executed before an object is deleted
* beforeValidate - Executed before an object is validated
* afterInsert - Executed after an object is persisted to the database
* afterUpdate - Executed after an object has been updated
* afterDelete - Executed after an object has been deleted
* onLoad - Executed when an object is loaded from the database

例如：

在insert之前触发调用

|  |
| --- |
| class Person {  private static final Date NULL\_DATE = new Date(0)  String firstName  String lastName  Date signupDate = NULL\_DATE  def beforeInsert() {  if (signupDate == NULL\_DATE) {  signupDate = new Date()  }  }  } |

在update之前触发调用

|  |
| --- |
| class Person {  def securityService  String firstName  String lastName  String lastUpdatedBy  static constraints = {  lastUpdatedBy nullable: true  }  def beforeUpdate() {  lastUpdatedBy = securityService.currentAuthenticatedUsername()  }  } |

#### 自定义事件监听器

grails2.0之后的版本提供了自定义持久化事件监听器的API。自定义监听器只需要实现AbstractPersistentEventListener接口，并实现onPersistentEvent和supportEventType两个方法：

|  |
| --- |
| public MyPersistenceListener(final Datastore datastore) {  super(datastore)  }  @Override  protected void onPersistenceEvent(final AbstractPersistenceEvent event) {  switch(event.eventType) {  case PreInsert:  println "PRE INSERT ${event.entityObject}"  break  case PostInsert:  println "POST INSERT ${event.entityObject}"  break  case PreUpdate:  println "PRE UPDATE ${event.entityObject}"  break;  case PostUpdate:  println "POST UPDATE ${event.entityObject}"  break;  case PreDelete:  println "PRE DELETE ${event.entityObject}"  break;  case PostDelete:  println "POST DELETE ${event.entityObject}"  break;  case PreLoad:  println "PRE LOAD ${event.entityObject}"  break;  case PostLoad:  println "POST LOAD ${event.entityObject}"  break;  }  }  @Override  public boolean supportsEventType(Class<? extends ApplicationEvent> eventType) {  return true  } |

创建完监听器之后，需要把它注册到applicationContext中。可以在BootStrap.groovy中增加如下的代码：

|  |
| --- |
| def init = {  application.mainContext.eventTriggeringInterceptor.datastores.each { k, datastore ->  applicationContext.addApplicationListener new MyPersistenceListener(datastore)  }  } |

#### Hibernate事件

一般来讲并不建议直接使用Hibernate的事件监听，但是当有更加明细化的监听需求的时候，grails也提供了直接使用Hibernate事件监听器的机制。可以在grails-app/conf/spring/resource.groovy中注册spring bean，在插件中则在doWithSpring方法中注册spring bean。

hibernate事件监听器的bean name是hibernateEventListeners，该bean有一个名为listenerMap的属性用来定义那些事件监听器需要被注册。

可用的监听器有：

|  |  |
| --- | --- |
| **Name** | **Interface** |
| auto-flush | [AutoFlushEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/AutoFlushEventListener.html) |
| merge | [MergeEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/MergeEventListener.html) |
| create | [PersistEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PersistEventListener.html) |
| create-onflush | [PersistEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PersistEventListener.html) |
| delete | [DeleteEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/DeleteEventListener.html) |
| dirty-check | [DirtyCheckEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/DirtyCheckEventListener.html) |
| evict | [EvictEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/EvictEventListener.html) |
| flush | [FlushEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/FlushEventListener.html) |
| flush-entity | [FlushEntityEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/FlushEntityEventListener.html) |
| load | [LoadEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/LoadEventListener.html) |
| load-collection | [InitializeCollectionEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/InitializeCollectionEventListener.html) |
| lock | [LockEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/LockEventListener.html) |
| refresh | [RefreshEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/RefreshEventListener.html) |
| replicate | [ReplicateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/ReplicateEventListener.html) |
| save-update | [SaveOrUpdateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/SaveOrUpdateEventListener.html) |
| save | [SaveOrUpdateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/SaveOrUpdateEventListener.html) |
| update | [SaveOrUpdateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/SaveOrUpdateEventListener.html) |
| pre-load | [PreLoadEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PreLoadEventListener.html) |
| pre-update | [PreUpdateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PreUpdateEventListener.html) |
| pre-delete | [PreDeleteEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PreDeleteEventListener.html) |
| pre-insert | [PreInsertEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PreInsertEventListener.html) |
| pre-collection-recreate | [PreCollectionRecreateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PreCollectionRecreateEventListener.html) |
| pre-collection-remove | [PreCollectionRemoveEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PreCollectionRemoveEventListener.html) |
| pre-collection-update | [PreCollectionUpdateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PreCollectionUpdateEventListener.html) |
| post-load | [PostLoadEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostLoadEventListener.html) |
| post-update | [PostUpdateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostUpdateEventListener.html) |
| post-delete | [PostDeleteEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostDeleteEventListener.html) |
| post-insert | [PostInsertEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostInsertEventListener.html) |
| post-commit-update | [PostUpdateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostUpdateEventListener.html) |
| post-commit-delete | [PostDeleteEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostDeleteEventListener.html) |
| post-commit-insert | [PostInsertEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostInsertEventListener.html) |
| post-collection-recreate | [PostCollectionRecreateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostCollectionRecreateEventListener.html) |
| post-collection-remove | [PostCollectionRemoveEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostCollectionRemoveEventListener.html) |
| post-collection-update | [PostCollectionUpdateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostCollectionUpdateEventListener.html) |

例如：

|  |
| --- |
| beans = {  auditListener(AuditEventListener)// AuditEventListener实现了PostInsertEventListener、PostUpdateEventListener和PostDeleteEventListener接口  hibernateEventListeners(HibernateEventListeners) {  listenerMap = ['post-insert': auditListener,  'post-update': auditListener,  'post-delete': auditListener]  }  } |

#### 自动时间戳

只要在domain class中定义了dateCreated或者lastUpdated属性，grails就会自动在持久化的时候给这两个属性赋值当前日期。也可以通过autoTimestamp false关闭自动时间戳。

|  |
| --- |
| class Person {  Date dateCreated  Date lastUpdated  static mapping = {  autoTimestamp false  }  } |

### 自定义ORM mapping

通过grails的ORM DSL（Object Relational Mapping Domain Specific Language ）也可以将domain class映射到已有的数据库表上。往往只有想GORM 映射到历史遗留schema或者想要配置domain的缓存的时候才会用到自定义ORM mapping。

通过在domian class中加入mapping块来完成自定义mapping。

|  |
| --- |
| class Person {  …  static mapping = {  version false  autoTimestamp false  }  } |

也可以在Config.xml中进行全局的映射配置：

|  |
| --- |
| grails.gorm.default.mapping = {  version false  autoTimestamp false  } |

#### 表名和字段名

使用table映射表名

|  |
| --- |
| class Person {  …  static mapping = {  table 'people'  }  } |

使用column映射字段名

|  |
| --- |
| class Person {  String firstName  static mapping = {  table 'people'  firstName column: 'First\_Name'  }  } |

字段类型

GROM使用hibernate types来映射字段的类型。例如：

|  |
| --- |
| class Address {  String number  String postCode  static mapping = {  postCode type: PostCodeType  }  } |
| class Address {  String number  String postCode  static mapping = {  postCode type: 'text'  }  } |

[hibernate Basic Type](http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/mapping.html#mapping-types-basictypes)

### 缓存策略

#### 配置缓存

GORM使用hibernate的二级缓存配置，其配置如下：

|  |
| --- |
| hibernate {  cache.use\_second\_level\_cache=true  cache.use\_query\_cache=true  cache.provider\_class='org.hibernate.cache.EhCacheProvider'//使用EHCache作为缓存实现  } |

#### 缓存对象实例

|  |
| --- |
| class Person {  …  static mapping = {  table 'people'  cache true  }  } |
| class Person {  …  static mapping = {  table 'people'  cache usage: 'read-only', include: 'non-lazy'  }  } |

#### 缓存关联

不但可以缓存对象实例，还可以缓存对象关联（集合）

|  |
| --- |
| class Person {  String firstName  static hasMany = [addresses: Address]  static mapping = {  table 'people'  version false  addresses column: 'Address', cache: true  }  } |
| class Address {  String number  String postCode  } |

同样，也可以配置缓存特性

|  |
| --- |
| cache: 'read-write' // or 'read-only' or 'transactional' |

#### 缓存查询

对于dynamic finder和criteria，也可以缓存。

|  |
| --- |
| def person = Person.findByFirstName("Fred", [cache: true])//dynamic finder |
| def people = Person.withCriteria {//criteria  like('firstName', 'Fr%')  cache true  } |

#### 缓存使用类型

* read-only - 只读，持久化类没有修改的情况下
* read-write - 持久化类会被更新的情况下
* nonstrict-read-write - 更新不是很频繁，不同事务更新相同数据的情况很少发生，事务没有严格隔离的情况下
* transactional - 之适用于支持事务缓存的缓存实现如：Jboss treeCache，只能在JTA环境下使用。

### 自定义主键

GROM默认使用字段id作为主键。默认情况下GROM依赖底层数据库机制产生主键值。也可以通过配置hibernate id generator自定义如何产生主键。

|  |
| --- |
| class Person {  …  static mapping = {  table 'people'  version false  id generator: 'hilo',//使用hibernate内置的hilo generator  params: [table: 'hi\_value',  column: 'next\_value',  max\_lo: 100]  }  } |

[hibernate id generator](http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/mapping.html#mapping-declaration-id-generator)

### 组合主键

GORM也支持组合主键，但并不推荐使用组合主键。

### 数据库索引

为了提高查询的性能，可以在domain的properties上使用index定义映射索引

|  |
| --- |
| class Person {  String firstName  String address  static mapping = {  table 'people'  version false  id column: 'person\_id'  firstName column: 'First\_Name', index: 'Name\_Idx'  address column: 'Address', index: 'Name\_Idx,Address\_Index'  }  } |

### 乐观锁和Versioning

GORM默认使用乐观锁，并且会为domain class自动生成一个version属性。version属性映射成version字段。

如果GORM映射的是一个历史遗留schema，表中并没有version这个字段，那也可以关闭乐观锁。

|  |
| --- |
| class Person {  …  static mapping = {  table 'people'  version false  }  } |

默认情况下version字段的类型是Long，每一次记录更新后，字段值都加一。也可以使用Timestamp作为version的字段的类型。

|  |
| --- |
| import java.sql.Timestamp  class Person {  …  Timestamp version  static mapping = {  table 'people'  }  } |

### Derived Properties

Derived Properties是指该属性是由其它属性推导而来。例如：

|  |
| --- |
| class Product {  Float price  Float taxRate  Float tax  } |
| class Product {  Float price  Float taxRate  Float tax  static mapping = {  tax formula: 'PRICE \* TAX\_RATE'  }  } |

当执行Product.get(42)时，对应的SQL如下：

|  |
| --- |
| select  product0\_.id as id1\_0\_,  product0\_.version as version1\_0\_,  product0\_.price as price1\_0\_,  product0\_.tax\_rate as tax4\_1\_0\_,  product0\_.PRICE \* product0\_.TAX\_RATE as formula1\_0\_  from  product product0\_  where  product0\_.id=? |

Derived Properties同样适用于dynamic finder和criteria。

|  |
| --- |
| Product.findAllByTaxGreaterThan(21.12) |
| Product.withCriteria {  gt 'tax', 21.12f  } |

上面的例子会产生如下的SQL：

|  |
| --- |
| select  this\_.id as id1\_0\_,  this\_.version as version1\_0\_,  this\_.price as price1\_0\_,  this\_.tax\_rate as tax4\_1\_0\_,  this\_.PRICE \* this\_.TAX\_RATE as formula1\_0\_  from  product this\_  where  this\_.PRICE \* this\_.TAX\_RATE>? |

### 默认排序

一般情况下可以使用如下方式对查询结果进行排序：

|  |
| --- |
| def airports = Airport.list(sort:'name') |

但是也可以在domain的mapping中定义sort来配置默认排序：

|  |
| --- |
| class Airport {  …  static mapping = {  sort name: "desc"//默认是asc  }  } |
| class Airport {//也可以对关联的对象集合进行排序  …  static hasMany = [flights: Flight]  static mapping = {  flights sort: 'number', order: 'desc'  }  } |

## 编程式事务处理

grails的编程式事务处理基于spring的编程式事务处理，然后GORM通过withTransaction方法大大简化了spring的编程式事务处理。withTransaction方法接收一个closure作为参数，该closure有一个类型是TransactionStatus的参数。

|  |
| --- |
| def transferFunds() {  Account.withTransaction { status ->  def source = Account.get(params.from)  def dest = Account.get(params.to)  def amount = params.amount.toInteger()  if (source.active) {  if (dest.active) {  source.balance -= amount  dest.amount += amount  }  else {  status.setRollbackOnly()  }  }  }  } |

## GORM和约束条件

关于domain的约束条件将在validation小节详细描述。这里，有必要知道的是有些约束会影响到最终生成的数据库schema。

对String类型的属性有影响的约束

* [inList](http://grails.org/doc/latest/ref/Constraints/inList.html)
* [maxSize](http://grails.org/doc/latest/ref/Constraints/maxSize.html)
* [size](http://grails.org/doc/latest/ref/Constraints/size.html)

对数字类型的属性有影响的约束

* [min](http://grails.org/doc/latest/ref/Constraints/min.html)
* [max](http://grails.org/doc/latest/ref/Constraints/max.html)
* [range](http://grails.org/doc/latest/ref/Constraints/range.html)