

更新访问: <http://jusescn.iteye.com/>

H3A

为了简化hbm到注解的迁移过程，配置机制将自动检测在注解和hbm文件重复的映射。默认hbm文件中的声明比类中的注解元数据具有更高的优先级。这种优先级以类为单位的，你可以通过hibernate.mapping. precedence修改这种优先级。默认的值是hbm,class。

注解分为逻辑注解和物理注解：逻辑注解描述对象的模型，类之间的关系等；物理注解表述了物理的schema，表、列、索引等。

2.2. 用EJB3注解进行映射

@AccessType注释

Hibernate的范围类型分为field和property。访问类型为property时需要在get()方法上注解。访问类型为field时需要在字段上进行注解，避免两种同时存在（当某类声明访问类型为field时，其继承的类也将其访问类型）。

|  |
| --- |
| @AccessType("field")  @AccessType("property") |

2.2.1 声明实体

@Entity注解

将一个类声明为一个实体

@Id注解

声明标识属性

@Table注解

定义表（table）

@Table是类一级的注释,可以为实体bean映射指定表(table),目录(catalog)和schema的名字;@uniqueConstraints定义表的唯一性约束

|  |
| --- |
| @Entity  @Table(name = "t\_flight", catalog = "", uniqueConstraints = { @UniqueConstraint(columnNames = { "id", "name" }) })  **public** **class** Flight **implements** Serializable {} |

@Version注解

乐观锁定版本控制

在实体中用@Version注释,下例中version属性映射到OPTLOCK列,entry manager用该字段来检测更新冲突;version列可以是numberic或者timestamp。

|  |
| --- |
| @Version  @Column(name="OPTLOCK")  **public** Integer getVersion() {  **return** version;  } |

2.2.2 映射简单属性

2.2.2.1. 声明基本的属性映射

@Basic注解

声明基本的属性映射

实体bean中所有非static非transient的属性都可以被持久化。没有定义注解的属性等价于在其上面添加了@Basic注解。通过@Basic注解可以设定属性的获取策略(fetch strategy)。通常不需要对简单属性设置延迟获取(lazy simple property),不要与延迟关联(lazy association fetch)获取混淆了

|  |
| --- |
| @Basic(fetch=FetchType.*LAZY*,optional=**false**)  **public** Long getDuration() {  **return** duration;  } |

注意：为了启用属性基本的延迟获取，你的类必须经过特殊处理(instrumented)：字节码将织入原始的类中来实现延迟获取的功能，参考hibernate参考手册。

推荐的替代方案是使用Criteria查询投影（projection）功能。

@Enumerated注解

Hiberante Annotation支持将内置的枚举类型映射到一个顺序列（保存了相应的序列值）或者一个字符串类型的列（保存相应的字符串）。默认保存是序列值，可用通过@Enumerated调整。

|  |
| --- |
| @Enumerated(EnumType.*STRING*)  **public** String getNot(){**..**}; |

@Temporal注解

定义时间类型，采用@Temporal调整精度

|  |
| --- |
| @Temporal(TemporalType.*DATE*)  **public** Date getDepartureDate() {  **return** departureDate;  } |

@Lob注解

表示属性被持久化为Clob或者Blob类型，其中Clob，character[]，char[]，String将被持久化为Clob类型；Blob，byte[]，Byte[]，serializable将被持久化为Blob类型。

|  |
| --- |
| @Lob  **public** String getText(){**.. ..**};    @Lob  **public** **byte**[] getText(){**.. ..**}; |

@Transient注解

表示被忽略的字段或者属性

#### 2.2.2.2. 声明列属性

@Column注解

将属性映射到列，用来覆盖默认值。

🡺可以和@Basic一起用

🡺可以和@version一起用

🡺可以和@Temporal一起用

🡺可以和@org.hibernate.annotations.CollectionOfElements一起用

|  |
| --- |
| @Column(name="fist\_name",updatable=**false**,nullable=**false**,length=50)  **public** String getName() {  **return** name;  } |

@Column(

name="columnName"; (1)

**boolean** unique() **default** **false**; (2)

**boolean** nullable() **default** **true**; (3)

**boolean** insertable() **default** **true**; (4)

**boolean** updatable() **default** **true**; (5)

String columnDefinition() **default** ""; (6)

String secondaryTable() **default** ""; (7)

**int** length() **default** 255; (8)

**int** precision() **default** 0; // decimal precision (9)

**int** scale() **default** 0; // decimal scale

)

|  |
| --- |
| (1) name (可选): 列名 (默认值为属性名)  (2) unique (可选):是否在该列设置唯一约束 (默认值为 **false**)  (3) nullable (可选): 设置该列是否可以为空值 (默认值为 **false**)(译注: 和上面的由冲突,可能是笔误).  (4) insertable (可选): 该列是否是插入语句的一部分 (默认值为 **true**)  (5) updatable (可选): 该列是否是更新语句的一部分 (默认值为 **true**)  (6) columnDefinition (可选): 重写该列的 sql DDL 定义语句 (默认值为 non portable)  (7) secondaryTable (可选): 定义目标表( targeted table )(默认值为 primary table)  (8) length (可选): 列长度 (默认值为 255)  (8) precision (可选): 列精度 (默认值为 0)  (10) scale (可选): column decimal scale **if** useful (默认值为 0) |

#### 2.2.2.3. 嵌入式对象

@Embedded/@Embeddable注解

定义嵌入式组件（embedded component），甚至覆盖该实体中原有的列映射。组件类必须在类一级定义@Embeddable注解，在特定的实体关联实体上采用@Embedded/@AttributeOverrides注解可以覆盖该属性对应的嵌入式对象的列映射。（个人见解：调整组件的列默认映射）

|  |
| --- |
| **@Entity**  **public** **class** Person **implements** Serializable {  // Persistent component using defaults  **@Embedded**  @AttributeOverrides({  @AttributeOverride(name="city", column = @Column(name="fld\_city") ),  //hibernate支持.表达式 nationality.iso2 获得对象属性  //如nationality是homeAddress的一个Country类型属性  //nationality.iso2为homeAddress的Country类型的iso2属性,修改列指向nat\_Iso2  @AttributeOverride(name="nationality.iso2", column = @Column(name="nat\_Iso2") ),  @AttributeOverride(name="nationality.name", column = @Column(name="nat\_CountryName") )  })  Address homeAddress;  **@Embedded**  @AttributeOverrides( {  @AttributeOverride(name="iso2", column = @Column(name="bornIso2") ),  @AttributeOverride(name="name", column = @Column(name="bornCountryName") )  } )  Country bornIn;  } |

|  |
| --- |
| **@Embeddable**  **public** **class** Address **implements** Serializable {  String city;  Country nationality; //no overriding here  }  **@Embeddable**  **public** **class** Country **implements** Serializable {  **private** String iso2;  **private** String name;  **public** String getIso2() { **return** iso2; }  **public** **void** setIso2(String iso2) { **this**.iso2 = iso2; }  @Column(name="countryName")  **public** String getName() { **return** name; }  **public** **void** setName(String name) { **this**.name = name; }  } |

在上面的例子中,实体bean Person 有两个组件属性, 分别是homeAddress和bornIn. 我们可以看到homeAddress 属性并没有注解. 但是Hibernate自动检测其对应的Address类中的@Embeddable注解, 并将其看作一个持久化组件.对于Country中已映射的属性, 则使用@Embedded和@AttributeOverride 注解来覆盖原来映射的列名.

#### 2.2.2.4. 无注解之属性的默认值

🡺若属性为单一类型，默认为@Basic

🡺否则，如果属性对应的类型定义了@Embeddable注解，则映射为@Embedded

🡺否则，如果属性对应的类型实现了Serializable，则属性映射为@Basic并在一个列中保存该对象的serialized版本

🡺否则，如果属性为Clob/Blob，则作为@Lob映射到恰当的LobType

2.2.3 映射主键属性

@Id/@GeneratedValue注解

映射主键属性

使用@Id注解可以将实体bean中的某个属性定义为标识符(identifier)，使用 @GeneratedValue注解可以定义该标识符的生成策略：

🡺AUTO - 可以是identity column类型,或者sequence类型或者table类型,取决于不同的底层数据库.

🡺TABLE - 使用表保存id值

🡺IDENTITY - identity column

🡺SEQUENCE – sequence

|  |
| --- |
| @Id @GeneratedValue(strategy=GenerationType.*SEQUENCE*, generator="SEQ\_STORE")  **private** **long** id; |

|  |
| --- |
| @Id @GeneratedValue(strategy=GenerationType.*AUTO*)  **private** **long** id; |

@TableGenerator/SequenceGenerator注解

配置不同的ID生成策略

每一个identifier生成器都有自己的适用范围,可以是应用级(application level)和类一级(class level). 类一级的生成器在外部是不可见的, 而且类一级的生成器可以覆盖应用级的生成器. 应用级的生成器则定义在包一级(package level)(如package-info.java)（TableGenerator不能在应用级定义）

|  |
| --- |
| @javax.persistence.TableGenerator(  name="EMP\_GEN",  table="GENERATOR\_TABLE",  pkColumnName = "key",  valueColumnName = "hi",  pkColumnValue="EMP"（EMP指实体类的类名,如User对象）,  allocationSize=20  )  @javax.persistence.SequenceGenerator(  name="SEQ\_GEN",  sequenceName="my\_sequence"  ) |

类级别定义

|  |
| --- |
| @Entity  @javax.persistence.SequenceGenerator(  name = "SEQ\_STORE",  sequenceName = "my\_sequence"  )  **public** **class** Store **implements** Serializable {  @Id  @GeneratedValue(strategy = GenerationType.*SEQUENCE*, generator = "SEQ\_STORE")  **private** Long id;  } |

@TableGenerator说明

直接使用

|  |
| --- |
| @Id @GeneratedValue(strategy = GenerationType.*TABLE*)  **private** Long id; |

将在数据库中产生`hibernate\_sequences`表对象，其中字段为`sequence\_name`和`sequence\_next\_hi\_value`，`sequence\_name`保存的当前实体类对应的表名

定义使用，定义ID生成器名，选择定义表名，字段名等

|  |
| --- |
| @javax.persistence.TableGenerator(  name="EMP\_GEN",  table="GENERATOR\_TABLE",  pkColumnName = "key",  valueColumnName = "hi",  pkColumnValue="User",  allocationSize=20  )  @Entity  Public class User{  @Id @GeneratedValue(strategy = GenerationType.*TABLE*,generator="EMP\_GEN")  **private** Long id;  } |

@IdClass/@Id注解

定义组合主键语法

🡺将组件类注解为@Embeddable,并将组件的属性注解为@Id

🡺将组件的属性注解为@Embeddable

🡺将类注解为@IdClass,并将该实体中所有属于主键的属性都注解为@Id

组合主键类对应了一个实体类中的多个字段或属性, 而且主键类中用于定义主键的字段或属性和 实体类中对应的字段或属性在类型上必须一致

|  |
| --- |
| @Entity  @**IdClass**(Footballer.**class**)  @Table(name="t\_footballer")  **public** **class** Footballer **implements** Serializable {  @**Id**  **private** String firstname;  @**Id**  **private** String lastname;  **private** String club;  }  @**Embeddable**  **public** **class** FootballerPk **implements** Serializable {  **private** String firstname;  **private** String lastname;  } |

或者采用@Embedded@Id的方式

|  |
| --- |
| @Entity  @AssociationOverride( name="id.channel", joinColumns = @JoinColumn(name="chan\_id") )  public class TvMagazin {  @Embedded@Id  public TvMagazinPk id;  @Temporal(TemporalType.TIME) Date time;  }  @Embeddable  public class TvMagazinPk implements Serializable {  @ManyToOne  public Channel channel;  public String name;  @ManyToOne  public Presenter presenter;  } |

@compose-ID 合成ID

将多对多拆分成两个一对多

|  |
| --- |
| User对象  {  @OneToMany(mappedBy="user")  **private** Set<DeptUser> depts = **new** HashSet<DeptUser>();  }  Dept对象  {  @OneToMany(mappedBy="dept")  **private** Set<DeptUser> users = **new** HashSet<DeptUser>();  }  部门用户对象  @Entity @Table(name="T\_DEPTUSER")  **public** **class** DeptUser {    @Id @Embedded  **private** DeptUserPk deptUserPk;  **private** **boolean** isMainDept;    @SuppressWarnings("unused")  @Column(name="userID", nullable=**false**, updatable=**false**, insertable=**false**)  **private** Long user;  @SuppressWarnings("unused")  @Column(name="deptID", nullable=**false**, updatable=**false**, insertable=**false**)  **private** Long dept;    **public** DeptUser() {}    **public** DeptUser(Dept dept, User user, **boolean** isMainDept) {  **this**.deptUserPk = **new** DeptUserPk(dept,user);  **this**.isMainDept = isMainDept;  }  }  @Embeddable  **class** DeptUserPk **implements** Serializable{  **private** **static** **final** **long** *serialVersionUID* = -4196096693557203625L;  @ManyToOne @JoinColumn(name="deptID")  **private** Dept dept;    @ManyToOne @JoinColumn(name="userID")  **private** User user;    **public** DeptUserPk() {}  **public** DeptUserPk(Dept dept, User user) {  **this**.dept = dept;  **this**.user = user;  }  } |

2.2.4. 映射继承关系

@Inheritance注解

映射继承关系（暂不支持在接口上注解）

🡺每个类一张表(Table per class)策略: 在Hibernate中对应<union-class>元素:

🡺每个类层次结构一张表(Single table per class hierarchy)策略:在Hibernate中对应<subclass>元素

🡺连接的子类(Joined subclasses)策略:在Hibernate中对应 <joined-subclass>元素

这个注解需要在每个类层次结构(class hierarchy) 最顶端的实体类上使用

每个类一张表

这种策略有很多缺点(例如:多态查询和关联), Hibernate使用SQL UNION查询来实现这种策略. 通常使用场合是在一个继承层次结构的顶端，这种策略支持双向的一对多关联. 这里不支持IDENTITY生成器策略,因为id必须在多个表间共享. 当然,一旦使用这种策略就意味着你不能使用 AUTO 生成器和IDENTITY生成器.

|  |
| --- |
| @Entity  @Inheritance(strategy = InheritanceType.*TABLE\_PER\_CLASS*)  **public** **class** Flight **implements** Serializable { |

@DiscriminatorValue注解

每个类层次结构一张表

整个继承层次结构中的父类和子类的所有属性都映射到同一个表中, 他们的实例通过一个辨别符(discriminator)列来区分。

|  |
| --- |
| @Entity  @Inheritance(strategy=InheritanceType.SINGLE\_TABLE)  @DiscriminatorColumn(  name="planetype",  discriminatorType=DiscriminatorType.STRING  )  @DiscriminatorValue("Plane")  **public** **class** Plane { ... }  @Entity  @DiscriminatorValue("A320")  **public** **class** A320 **extends** Plane { ... } |

在上面这个例子中,Plane是父类,在这个类里面将继承策略定义为 InheritanceType.SINGLE\_TABLE,并通过 @DiscriminatorColumn注解定义了辨别符列(还可以定义辨别符的类型). 最后,对于继承层次结构中的每个类,@DiscriminatorValue注解指定了用来辨别该类的值. 辨别符列的名字默认为 DTYPE,其默认值为实体名(在@Entity.name中定义)，其类型 为DiscriminatorType.STRING. A320是子类,如果不想使用默认的辨别符,只需要指定相应的值即可. 其他的如继承策略,辨别标志字段的类型都是自动设定的.

@Inheritance 和 @DiscriminatorColumn 注解只能用于实体层次结构的顶端.

@PrimaryKeyJoinColumn注解

连接的子类

当每个子类映射到一个表时, @PrimaryKeyJoinColumn 和@PrimaryKeyJoinColumns 注解定义了每个子类表关联到父类表的主键:

|  |
| --- |
| @Entity  @Inheritance(strategy=InheritanceType.JOINED)  **public** **class** Boat **implements** Serializable { ... }  @Entity  **public** **class** Ferry **extends** Boat { ... }  @Entity  @PrimaryKeyJoinColumn(name="BOAT\_ID")  **public** **class** AmericaCupClass **extends** Boat { ... } |

以上所有实体都使用了JOINED策略, Ferry表和Boat表使用同名的主键. 而AmericaCupClass表和Boat表使用了条件 Boat.id = AmericaCupClass.BOAT\_ID进行关联.

@MappedSuperclass注解

从父类继承属性

有时候通过一个(技术上或业务上)父类共享一些公共属性是很有用的, 同时还不用将该父类作为映射的实体(也就是该实体没有对应的表). 这个时候你需要使用@MappedSuperclass注解来进行映射.

|  |
| --- |
| @MappedSuperclass  **Public abstract** **class** BaseEntity {  @Basic  @Temporal(TemporalType.TIMESTAMP)  **public** Date getLastUpdate() { ... }  **public** String getLastUpdater() { ... }  }  @Entity **class** Order **extends** BaseEntity {  @Id **public** Integer getId() { ... }  } |

在数据库中,上面这个例子中的继承的层次结构最终以Order表的形式出现, 该表拥有id, lastUpdate 和 lastUpdater三个列.父类中的属性映射将复制到其子类实体.

注意:

🡺这种情况下的父类不再处在继承层次结构的顶端.

🡺没有注解为@MappedSuperclass的父类中的属性将被忽略

🡺除非显式使用Hibernate annotation中的@AccessType注解, 否则将从继承层次结构的根实体中继承访问类型(包括字段或方法)

🡺这对于@Embeddable对象的父类中的属性持久化同样有效. 只需要使用@MappedSuperclass注解即可 (虽然这种方式不会纳入EJB3标准)

🡺可以将处在在映射继承层次结构的中间位置的类注解为@MappedSuperclass

🡺在继承层次结构中任何没有被注解为@MappedSuperclass 或@Entity的类都将被忽略.

@AttributeOverride/@AssociationOverride注解

覆盖实体父类中的定义的列. 可以为@Entity和@MappedSuperclass注解的类 以及那些对象为@Embeddable的属性定义 @AttributeOverride和@AssociationOverride

|  |
| --- |
| @MappedSuperclass  **public** **class** FlyingObject **implements** Serializable {  **public** **int** getAltitude() {  **return** altitude;  }  @Transient  **public** **int** getMetricAltitude() {  **return** metricAltitude;  }  @ManyToOne  **public** PropulsionType getPropulsion() {  **return** metricAltitude;  }  ...  }  @Entity  @AttributeOverride( name="altitude", column = @Column(name="fld\_altitude") )  @AssociationOverride( name="propulsion", joinColumns = @JoinColumn(name="fld\_propulsion\_fk") )  **public** **class** Plane **extends** FlyingObject {  } |

在上面这个例子中,altitude属性的值最终将持久化到Plane 表的fld\_altitude列.而名为propulsion的关联则保存在fld\_propulsion\_fk外间列.

2.2.5. 映射实体Bean的关联关系

2.2.5.1. 集合类型

你可以对 Collection ,List (指有序列表, 而不是索引列表), Map和Set这几种类型进行映射. EJB3规范定义了怎么样使用@javax.persistence.OrderBy 注解来对有序列表进行映射： 该注解接受的参数格式：用逗号隔开的(目标实体)属性名及排序指令, 如firstname asc, age desc,如果该参数为空,则默认以id对该集合进行排序. 如果某个集合在数据库中对应一个关联表(association table)的话,你不能在这个集合属性上面使用@OrderBy注解. 对于这种情况的处理方法,请参考Hibernate Annotation Extensions. EJB3 允许你利用目标实体的一个属性作为Map的key, 这个属性可以用@MapKey(name="myProperty")来声明. 如果使用@MapKey注解的时候不提供属性名, 系统默认使用目标实体的主键. map的key使用和属性相同的列：不需要为map key定义专用的列，因为map key实际上就表达了一个目标属性。 注意一旦加载,key不再和属性保持同步, 也就是说,如果你改变了该属性的值,在你的Java模型中的key不会自动更新 (请参考Hibernate Annotation Extensions). 很多人被<map>和@MapKey弄糊涂了。 其他它们有两点区别.@MapKey目前还有一些限制,详情请查看论坛或者 我们的JIRA缺陷系统。 注意一旦加载,key不再和属性保持同步, 也就是说,如果你改变了该属性的值,在你的Java模型中的key不会自动更新. (Hibernate 3中Map支持的方式在当前的发布版中还未得到支持).

集合语义

|  |  |  |
| --- | --- | --- |
| Bag语义 | java.util.List, java.util.Collection | @org.hibernate.annotations.CollectionOfElements 或 @OneToMany 或 @ManyToMany |
| List语义 | java.util.List | (@org.hibernate.annotations.CollectionOfElements 或 @OneToMany 或 @ManyToMany) 以及 @org.hibernate.annotations.IndexColumn |
| Set语义 | java.util.Set | @org.hibernate.annotations.CollectionOfElements 或 @OneToMany 或 @ManyToMany |
| Map语义 | java.util.Map | (@org.hibernate.annotations.CollectionOfElements 或 @OneToMany 或 @ManyToMany) 以及 (空 或 @org.hibernate.annotations.MapKey/MapKeyManyToMany(支持真正的map), 或 @javax.persistence.MapKey |

*从上面可以明确地看到,没有@org.hibernate.annotations.IndexColumn 注解的java.util.List集合将被看作bag类*.

EJB3规范不支持原始类型,核心类型,嵌入式对象的集合.但是Hibernate对此提供了支持

|  |
| --- |
| @Entity **public** **class** City {  @**OneToMany**(mappedBy="city")  @**OrderBy**("streetName")  **public** List<Street> getStreets() {  **return** streets;  }  }  @Entity **public** **class** Street {  **public** String getStreetName() {  **return** streetName;  }  @**ManyToOne**  **public** City getCity() {  **return** city;  }  }  @Entity  **public** **class** Software {  @**OneToMany**(mappedBy="software")  @**MapKey**(name="codeName")  **public** Map<String, Version> getVersions() {  **return** versions;  }  }  @Entity  @Table(name="tbl\_version")  **public** **class** Version {  **public** String getCodeName() {...}  @**ManyToOne**  **public** Software getSoftware() { ... }  } |

上面这个例子中,City 中包括了以streetName排序的Street的集合. 而Software中包括了以codeName作为 key和以Version作为值的Map.除非集合为generic类型,否则你需要指定targetEntity. 这个注解属性接受的参数为目标实体的class.

2.2.5.2. 一对一(One-to-one)

@OneToOne注解

建立实体bean之间的一对一的关联. 一对一关联有三种情况：

🡺关联的实体都共享同样的主键

🡺其中一个实体通过外键关联到另一个实体的主键 (注意要模拟一对一关联必须在外键列上添加唯一约束)

🡺通过关联表来保存两个实体之间的连接关系 (注意要模拟一对一关联必须在每一个外键上添加唯一约束)

@PrimaryKeyJoinColumn注解

共享主键实现One-to-one

|  |
| --- |
| @Entity  **public** **class** Owner {  @Id @GeneratedValue  **private** Integer id;    @Column(length=100)  **private** String name;    @OneToOne(cascade = CascadeType.*ALL*)  @**PrimaryKeyJoinColumn**  **private** OwnerAddress address;  } |

|  |
| --- |
| @Entity  **public** **class** OwnerAddress {  //定义ID生成策略  @GenericGenerator(strategy = "foreign", name = "fk", parameters = @Parameter(name="property", value="owner"))  @Id @GeneratedValue(generator = "fk") //id来源  **private** Integer id;  @OneToOne(mappedBy="address", optional = **false**)  **private** Owner owner;  } |

@JoinColumn注解

外键实现One-to-one

|  |
| --- |
| @Entity  **public** **class** Customer **implements** Serializable {  @Id @GeneratedValue  **private** **long** id;  @Column(length=100)  **private** String name;  @OneToOne(cascade=CascadeType.*ALL*)  @**JoinColumn**(name="pid\_fk")  **private** Passport passport;  } |

|  |
| --- |
| @Entity  **public** **class** Passport **implements** Serializable {  @Id @GeneratedValue  **private** **long** id;  //这里可以不用写关联  @OneToOne(mappedBy="passport")  **private** Customer customer;  @Column(length=100)  **private** String name;  } |

上面这个例子中,Customer 通过Customer 表中名为的passport\_fk 外键列和 Passport关联. @JoinColumn注解定义了联接列(join column). 该注解和@Column注解有点类似, 但是多了一个名为**referencedColumnName**的参数. 该参数定义了所关联目标实体中的联接列. 注意,当referencedColumnName关联到非主键列的时候, 关联的目标类必须实现Serializable, 还要注意的是所映射的属性对应单个列(否则映射无效)【意思是不通过ID关联，而通过其他的字段关联，被关联的表应实现Serializable】.

一对一关联可能是双向的.在双向关联中, 有且仅有一端是作为主体(owner)端存在的：主体端负责维护联接列(即更新). 对于不需要维护这种关系的从表则通过mappedBy属性进行声明. mappedBy的值指向主体的关联属性. 在上面这个例子中,mappedBy的值为 passport. 最后,不必也不能再在被关联端(owned side)定义联接列了,因为已经在主体端进行了声明.

如果在主体没有声明@JoinColumn,系统自动进行处理： 在主表(owner table)中将创建联接列, 列名为：主体的关联属性名＋下划线＋被关联端的主键列名. 在上面这个例子中是passport\_id, 因为Customer中关联属性名为passport, Passport的主键是id.

@JoinTable注释

通过关联表实现One-to-one

|  |
| --- |
| **public** **class** AnCustomer **implements** Serializable {  @Id @GeneratedValue(strategy=GenerationType.*AUTO*)  **private** **long** id;  @OneToOne(cascade=CascadeType.*ALL*)  @JoinTable(name="cu\_pa",joinColumns=@JoinColumn(name="cu\_pk"),inverseJoinColumns=@JoinColumn(name="pa\_fk"))  **private** AnPassport passport;  } |

#### 2.2.5.3. 多对一(Many-to-one)

@ManyToOne注释

建立实体bean之间的多对一的关联，可以通过外键关联或者关联表方式实现

@ManyToOne/@JoinColumn注释

🡺外键关联

|  |
| --- |
| @Entity()  **public** **class** Flight **implements** Serializable {  @**ManyToOne**( cascade = {CascadeType.PERSIST, CascadeType.MERGE}, targetEntity=CompanyImpl.**class** )  @**JoinColumn**(name="COMP\_ID")  **public** Company getCompany() {  **return** company;  }  ...  }  **public** **interface** Company {  ... |

其中@JoinColumn是可选的,关联字段默认值和一对一 (one to one)关联的情况相似, 列名为：主体的关联属性名＋下划线＋被关联端的主键列名. 在这个例子中是company\_id, 因为关联的属性是company, Company的主键是id.

@ManyToOne注解有一个名为targetEntity的参数, 该参数定义了目标实体名.通常不需要定义该参数, 因为在大部分情况下默认值(表示关联关系的属性类型)就可以很好的满足要求了. 不过下面这种情况下这个参数就显得有意义了：使用接口作为返回值而不是常见的实体.

@ManyToOne/@JoinTable注释

关联表实现

|  |
| --- |
| @Entity()  **public** **class** Flight **implements** Serializable {  @**ManyToOne**( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )  @**JoinTable**(name="Flight\_Company",  joinColumns = @JoinColumn(name="FLIGHT\_ID"),  inverseJoinColumns = @JoinColumn(name="COMP\_ID")  )  **public** Company getCompany() {  **return** company;  }  ...  } |

#### 2.2.5.4. 一对多(One-to-many)

@OneToMany注释

定义一对多关联.一对多关联可以是双向关联.

注意：建立一对多双向关联时，不管是主体端（多的一端）还是一对多的一端，最好在两端都加上@Cascade(CascadeType.*SAVE\_UPDATE*)

##### 2.2.5.4.1. 双向(Bidirectional)

在EJB3规范中多对一这端几乎总是双向关联中的主体(owner)端, 而一对多这端的关联注解为@OneToMany( mappedBy=... )

|  |
| --- |
| @Entity  public class Troop {  @OneToMany(mappedBy="troop")  public Set<Soldier> getSoldiers() {  ...  }  @Entity  public class Soldier {  @ManyToOne  @JoinColumn(name="troop\_fk")  public Troop getTroop() {  ...  } |

Troop 通过troop 属性和Soldier建立了一对多的双向关联. 在mappedBy端不必也不能再定义任何物理映射

对于一对多的双向映射,如果要一对多这一端维护关联关系, 你需要删除mappedBy元素并将多对一这端的 @JoinColumn的insertable和updatable设置为false. 很明显,这种方案不会得到什么明显的优化,而且还会增加一些附加的UPDATE语句.

##### 2.2.5.4.2. 单向(Unidirectional)

通过在被拥有的实体端(owned entity)增加一个外键列来实现一对多单向关联是很少见的,也是不推荐的. 我们强烈建议通过一个联接表(join table)来实现这种关联(下一节会对此进行解释). 可以通过@JoinColumn注解来描述这种单向关联关系.

|  |
| --- |
| @Entity  public class Customer implements Serializable {  @OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)  @JoinColumn(name="CUST\_ID")  public Set<Ticket> getTickets() {  ...  }  @Entity  public class Ticket implements Serializable {  ... //no bidir  } |

##### 2.2.5.4.3. 通过关联表处理单向关联

通过联接表处理单向一对多关联是首选方式.这种关联通过@JoinTable注解来进行描述.

|  |
| --- |
| @Entity  public class Trainer {  @OneToMany  @JoinTable(  name="TrainedMonkeys",  joinColumns = @JoinColumn( name="trainer\_id"),  inverseJoinColumns = @JoinColumn( name="monkey\_id")  )  public Set<Monkey> getTrainedMonkeys() {  ...  }  @Entity  public class Monkey {  ... //no bidir  } |

上面这个例子中,Trainer通过 TrainedMonkeys表和 Monkey 建立了单向关联. 其中外键trainer\_id关联到Trainer (joinColumns), 而外键monkey\_id关联到 Monkey (inversejoinColumns).

##### 2.2.5.4.4. 默认处理机制

通过联接表来建立单向一对多关联不需要描述任何物理映射. 表名由以下三个部分组成:主表(owner table)表名+下划线+从表(the other side table)表名. 指向主表的外键名：主表表名+下划线+主表主键列名 指向从表的外键名：主表所对应实体的属性名+下划线+从表主键列名 指向从表的外键定义为唯一约束,用来表示一对多的关联关系.

#### 2.2.5.5. 多对多(Many-to-many)

@ManyToMany注释

定义的多对多关联. 同时,你也需要通过注解@JoinTable描述关联表和关联条件. 如果是双向关联,其中一段必须定义为owner,另一端必须定义为inverse(在对关联表进行更新操作时这一端将被忽略):

|  |
| --- |
| @Entity  **public** **class** Employer **implements** Serializable {  @**ManyToMany**(  targetEntity=org.hibernate.test.metadata.manytomany.Employee.**class**,  cascade={CascadeType.PERSIST, CascadeType.MERGE}  )  @**JoinTable**(  name="EMPLOYER\_EMPLOYEE",  joinColumns=@JoinColumn(name="EMPER\_ID"),  inverseJoinColumns=@JoinColumn(name="EMPEE\_ID")  )  **public** Collection getEmployees() {  **return** employees;  }  ...  }    @Entity  **public** **class** Employee **implements** Serializable {  @**ManyToMany**(  cascade = {CascadeType.PERSIST, CascadeType.MERGE},  mappedBy = "employees",  targetEntity = Employer.**class**  )  **public** Collection getEmployers() {  **return** employers;  }  } |

#### 2.2.5.5. 用cascading实现传播性持久化(Transitive persistence)

@CascadeType定义

也许你已经注意到了cascade属性接受的值为CascadeType数组. 在EJB3中的cascade的概念和Hibernate中的传播性持久化以及cascade操作非常类似, 但是在语义上有细微的区别,支持的cascade类型也有点区别:

🡺CascadeType.PERSIST: 如果一个实体是受管状态, 或者当persist()函数被调用时, 触发级联创建(create)操作

🡺CascadeType.MERGE: 如果一个实体是受管状态, 或者当merge()函数被调用时, 触发级联合并(merge)操作

🡺CascadeType.REMOVE: 当delete()函数被调用时, 触发级联删除(remove)操作

🡺CascadeType.REFRESH: 当refresh()函数被调用时, 触发级联更新(refresh)操作

🡺CascadeType.ALL: 以上全部

#### 2.2.5.6. 关联关系获取

通过Hibernate你可以获得直接或者延迟获取关联实体的功能. fetch参数可以设置为FetchType.LAZY 或者 FetchType.EAGER. EAGER通过outer join select直接获取关联的对象, 而LAZY(默认值)在第一次访问关联对象的时候才会触发相应的select操作. EJBQL提供了fetch关键字,该关键字可以在进行特殊查询的时候覆盖默认值. 这对于提高性能来说非常有效,应该根据实际的用例来判断是否选择fetch关键字.

#### 2.2.5.7. 关联关系实例

|  |
| --- |
| @Entity  public class AnPerson implements Serializable {  @Id @GeneratedValue(strategy=GenerationType.*AUTO*)  private int id;    @Column(nullable=true)  private String pname;    @OneToOne  @org.hibernate.annotations.Cascade(  value={org.hibernate.annotations.CascadeType.*SAVE\_UPDATE*,  org.hibernate.annotations.CascadeType.*PERSIST*}  )  @JoinColumn(name="wife\_id")  private AnWife wife;    @ManyToMany(targetEntity=AnGirlFirend.class,fetch=FetchType.*LAZY*)  @org.hibernate.annotations.Cascade(  value={org.hibernate.annotations.CascadeType.*SAVE\_UPDATE*,  org.hibernate.annotations.CascadeType.*PERSIST*}  )  @JoinTable(  joinColumns=@JoinColumn(name="person\_id"),  name="person\_gf",  inverseJoinColumns=@JoinColumn(name="gf\_id")  )  private List<AnGirlFirend> grilFirends = new ArrayList();    @OneToMany(mappedBy="father")  @org.hibernate.annotations.Cascade(  value={org.hibernate.annotations.CascadeType.*SAVE\_UPDATE*,  org.hibernate.annotations.CascadeType.*PERSIST*}  )  private List<AnChild> children = new ArrayList();  } |

|  |
| --- |
| @Entity  public class AnWife implements Serializable {  @Id @GeneratedValue(strategy=GenerationType.*AUTO*)  private int id;    @Column(nullable=true)  private String wifeName;    @OneToOne(mappedBy="wife")  private AnPerson person;    @OneToMany(mappedBy="mother")  private List<AnChild> children = new ArrayList();  }  @Entity  public class AnGirlFirend implements Serializable {  @Id @GeneratedValue(strategy=GenerationType.*AUTO*)  private int id;    @Column(nullable=true)  private String gfName;    @ManyToMany(  mappedBy="grilFirends",  targetEntity=AnPerson.class  )  @Cascade(value={CascadeType.*SAVE\_UPDATE*,CascadeType.*PERSIST*})  private List<AnPerson> boyfirends = new ArrayList();  }  @Entity  public class AnChild implements Serializable {  @Id @GeneratedValue(strategy=GenerationType.*AUTO*)  private int id;    @Column(nullable=true)  private String cname;    @ManyToOne(targetEntity=AnPerson.class)  @Cascade(value={CascadeType.*SAVE\_UPDATE*,CascadeType.*PERSIST*})  @JoinColumn(name="father\_id")  private AnPerson father;    @ManyToOne(targetEntity=AnWife.class)  @Cascade(value={CascadeType.*SAVE\_UPDATE*,CascadeType.*PERSIST*})  @JoinColumn(name="mother\_id")  private AnWife mother;  } |

### 2.2.6. 映射复合主键与外键

@EmbeddedId注释

用一个@EmbeddedId即可代替@Id 和@Embeddable两个注解

|  |
| --- |
| @Entity  **public** **class** RegionalArticle **implements** Serializable {  @Id  **public** RegionalArticlePk getPk() { ... }  }  @Embeddable  **public** **class** RegionalArticlePk **implements** Serializable { ... }  或者  @Entity  **public** **class** RegionalArticle **implements** Serializable {  @EmbeddedId  **public** RegionalArticlePk getPk() { ... }  }  **public** **class** RegionalArticlePk **implements** Serializable { ... } |

### 2.2.7. 映射二级表(secondary tables)

@SecondaryTables注释

使用类一级的 @SecondaryTable 或 @SecondaryTables 注解可以实现单个实体到多个表的映射. 使用 @Column 或者 @JoinColumn 注解中的 table 参数可指定某个列所属的特定表.

|  |
| --- |
| @Entity  @SecondaryTables({  @SecondaryTable(name="cat1",pkJoinColumns=@PrimaryKeyJoinColumn(name="cat\_id",referencedColumnName="id")),  @SecondaryTable(name="cat2",uniqueConstraints={@UniqueConstraint(columnNames={"storyPart2"})})  })  **public** **class** Cat **implements** Serializable {  **private** **static** **final** **long** *serialVersionUID* = -144925780418904984L;  @Id @GeneratedValue(strategy=GenerationType.*AUTO*)  **private** Integer id;  **private** String name;  @Column(table="cat1")  **private** String storyPart1;  @Column(table="cat2")  **private** String storyPart2;  } |

2.3. 映射查询

### 2.3.1. 映射EJBQL/HQL查询

@NamedQuery注释

命名式查询通过它的名字和实际的查询字符串来定义

@org.hibernate.annotations.NamedQuery注释扩展

* + org.hibernate.cacheable 查询是否与二级缓存交互(默认值为false)
  + org.hibernate.cacheRegion 设置缓存区名称 (默认为otherwise)
  + org.hibernate.timeout 查询超时设定
  + org.hibernate.fetchSize 所获取的结果集(resultset)大小
  + org.hibernate.flushMode 本次查询所用的刷新模式
  + org.hibernate.cacheMode 本次查询所用的缓存模式
  + org.hibernate.readOnly 是否将本次查询所加载的实体设为只读(默认为false)
  + org.hibernate.comment 将查询注释添加入所生成的SQL

2.3.2. 映射本地化查询

@NamedNativeQuery注释

映射本地化查询(也就是普通SQL查询).

@SqlResultSetMapping注释

来描述SQL的resultset的结构 (如果你打算定义多个结果集映射，可是使用@SqlResultSetMappings). @SqlResultSetMapping和@NamedQuery, @SqlResultSetMapping一样,可以定义在类和包一级. 但是@SqlResultSetMapping的作用域为应用级下面我们会看到,@NamedNativeQuery 注解中 resultSetMapping参数值为@SqlResultSetMapping的名字. 结果集映射定义了通过本地化查询返回值和实体的映射. 该实体中的每一个字段都绑定到SQL结果集中的某个列上. 该实体的所有字段包括子类的所有字段以及 关联实体的外键列都必须在SQL查询中有对应的定义. 如果实体中的属性和SQL查询中的列名相同,这种情况下可以不进行定义字段映射.

|  |
| --- |
| @NamedNativeQuery(name="night&area", query="select night.id nid, night.night\_duration, "  + " night.night\_date, area.id aid, night.area\_id, area.name "  + "from Night night, Area area where night.area\_id = area.id", resultSetMapping="joinMapping")  @SqlResultSetMapping(name="joinMapping", entities={  @EntityResult(entityClass=org.hibernate.test.annotations.query.Night.**class**, fields = {  @FieldResult(name="id", column="nid"),  @FieldResult(name="duration", column="night\_duration"),  @FieldResult(name="date", column="night\_date"),  @FieldResult(name="area", column="area\_id"),  },discriminatorColumn="disc"),  @EntityResult(entityClass=org.hibernate.test.annotations.query.Area.**class**, fields = {  @FieldResult(name="id", column="aid"),  @FieldResult(name="name", column="name")  })  }  ) |

如果查询返回的是单个实体,或者你打算使用系统默认的映射, 这种情况下可以不使用resultSetMapping 而是使用resultClass属性:

2.4. Hibernate独有的注解扩展

2.4.1 实体

@Entity注解

* mutable: 此实体是否为可变的
* dynamicInsert: 用动态SQL新增
* dynamicUpdate: 用动态SQL更新
* selectBeforeUpdate: 指明Hibernate从不运行SQL UPDATE除非能确定对象的确已被修改
* polymorphism: (指出)实体多态是PolymorphismType.IMPLICIT(默认)还是PolymorphismType.EXPLICIT
* persister:允许对默认持久实现(persister implementation)的覆盖
* optimisticLock: 乐观锁策略(OptimisticLockType.VERSION, OptimisticLockType.NONE, OptimisticLockType.DIRTY或OptimisticLockType.ALL)

注意：@javax.persistence.Entity仍是必选的(mandatory), @org.hibernate.annotations.Entity不是取代品.

@BatchSize注解

允许你定义批量获取该实体的实例数量(如：@BatchSize(size=4)). 当加载一特定的实体时,Hibernate将加载在持久上下文中未经初始化的同类型实体,直至批量数量(上限).

@Proxy注解

定义了实体的延迟属性.Lazy(默认为true)定义了类是否为延迟(加载). proxyClassName是用来生成代理的接口(默认为该类本身).

@Where注解

定义了当获取类实例时所用的SQL WHERE子句(该SQL WHERE子句为可选).

@Check注解

定义了在DDL语句中定义的合法性检查约束(该约束为可选).

@OnDelete(action=OnDeleteAction.CASCADE)注解

定义于被连接的子类(joined subclass)：在删除时使用SQL级连删除,而非通常的Hibernate删除机制.

@Table注解

@Table(name="tableName",indexes={@Index(name="index1",columnNames={"column1", "column2"})})在tableName表的列上创建定义好的索引. 该注解可以被应用于关键表或者是其他次要的表. @Tables 注解允许你在不同的表上应用索引. 此注解预期在使用 @javax.persistence.Table或 @javax.persistence.SecondaryTable的地方中出现.

注意：@org.hibernate.annotations.Table 是对 @javax.persistence.Table的补充而不是它的替代品. 特别是当你打算改变表名的默认值的时候，你必须使用@javax.persistence.Table, 而不是@org.hibernate.annotations.Table.

|  |
| --- |
| @Entity  @BatchSize(size=5)  @org.hibernate.annotations.Entity(  selectBeforeUpdate = **true**,  dynamicInsert = **true**, dynamicUpdate = **true**,  optimisticLock = OptimisticLockType.ALL,  polymorphism = PolymorphismType.EXPLICIT)  @Where(clause="1=1")  @org.hibernate.annotations.Table(name="Forest", indexes = { @Index(name="idx", columnNames = { "name", "length" } ) } )  **public** **class** Forest { ... }  @Entity  @Inheritance(  strategy=InheritanceType.JOINED  )  **public** **class** Vegetable { ... }  @Entity  @OnDelete(action=OnDeleteAction.CASCADE)  **public** **class** Carrot **extends** Vegetable { ... } |

@Immutable注解

标示一个对象或者集合不可变。一个不变的实体不能被更新，当应用更新不变的的实体的时候，操作不会被执行，而且不会报错。该注释只能用在根实体上；当更新（增加或者减少）一个不变的集合时，会抛Hibernate错误。

2.4.2. 标识符

@GenericGenerator注释

允许你定义一个Hibernate特定的id生成器.

|  |
| --- |
| @Id @GeneratedValue(generator="system-uuid")  @GenericGenerator(name="system-uuid", strategy = "uuid")  **public** String getId() {  @Id @GeneratedValue(generator="hibseq")  @GenericGenerator(name="hibseq", strategy = "seqhilo",  parameters = {  @Parameter(name="max\_lo", value = "5"),  @Parameter(name="sequence", value="heybabyhey")  }  )  **public** Integer getId() { |

strategy可以是Hibernate3生成器策略的简称, 或者是一个IdentifierGenerator实现的(带包路径的)全限定类名. 你可以通过parameters属性增加一些参数.

2.4.3. 属性

2.4.3.1. 访问类型

@AccessType注解

在Hibernate中,你可以把访问类型覆盖成：

* 使用定制的访问类型策略
* 优化类级或属性级的访问类型

为支持这种行为,Hibernate引入了@AccessType注解.你可以对以下元素定义访问类型：

* 实体
* 父类
* 可内嵌的对象
* 属性

2.4.3.2. 公式

@Formula注解

让数据库,而非JVM,完成一些计算,也可能创建某种虚拟列. 可以使用SQL片段(亦称为公式),而不是将属性映射到(物理)列. 这种属性是只读的(属性值由公求得).SQL片段可以是任意复杂的,甚至可包含子查询.

2.4.3.3. 自定义数据类型

@org.hibernate.annotations.Type 注释

覆盖了Hibernate所用的默认类型：这通常不是必须的。

@org.hibernate.annotations.TypeDef注释

允许你来声明类型定义. 这些注解被置于类或包一级.注意,对session factory来说, 这些定义将是全局的(即使定义于类一级),并且类型定义必须先于任何使用.

|  |
| --- |
| @TypeDefs(  {  @TypeDef(  name="caster",  typeClass = CasterStringType.**class**,  parameters = {  @Parameter(name="cast", value="lower")  }  )  }  )  **package** org.hibernate.test.annotations.entity;  **public** **class** Forest {  @Type(type="caster")  **public** String getSmallText() {}  @Type(type = "caster", parameters = {@Parameter(name = "cast", value = "upper")})  **public** String getBigText() {  return bigText;  }  } |

2.4.3.4. 索引

@Index注解

通过在列属性(property)上使用@Index注解, 可以在特定列上定义索引,columnNames属性(attribute)将随之被忽略.

2.4.3.5. @Parent

@Parent 注解

在嵌入式对象内部,你可以在那些指向该嵌入式对象所属元素的属性上定义该注解.

|  |
| --- |
| @Entity  **public** **class** Person {  @Embeddable **public** Address address;  }  @Embeddable  **public** **class** Address {  @Parent **public** Person owner;  }  person == person.address.owner |

2.4.3.6. 生成的属性

@Generated注释

某些属性可以在对数据库做插入或更新操作的时候生成. Hibernate能够处理这样的属性,并触发一个后续的查询来读取这些属性(这个好像有问题,不知道什么意思).

|  |
| --- |
| @Entity  **public** **class** Antenna {  @Id **public** Integer id;  @Generated(GenerationTime.*ALWAYS*) @Column()  **public** String longitude;  @Generated(GenerationTime.*INSERT*) @Column(insertable = **false**)  **public** String latitude;  @Generated(GenerationTime.*NEVER*)  **public** Double power;  } |

你可以将属性注解为@Generated. 但是你要注意insertability和updatability不要和你选择的生成策略冲突. 如果选择了GenerationTime.INSERT,该属性不能包含insertable列, 如果选择了GenerationTime.ALWAYS,该属性不能包含insertable和updatable列.

@Version属性不可以为 @Generated(INSERT)(设计时), 只能是 NEVER或ALWAYS.

2.4.4. 继承-辨别符公式

@DiscriminatorForumla 注解

SINGLE\_TABLE 是个功能强大的策略,但有时,特别对遗留系统而言, 是无法加入一个额外的辨别符列. 由此,Hibernate引入了辨别符公式(discriminator formula)的概念： @DiscriminatorFormula是@DiscriminatorColumn的替代品, 它使用SQL片段作为辨别符解决方案的公式( 不需要有一个专门的字段).

|  |
| --- |
| @Entity  @DiscriminatorForumla("case when forest\_type is null then 0 else forest\_type end")  **public** **class** Forest { ... } |

2.4.5. 关于单个关联关系的注解

@NotFound注释

默认情况下,当预期的被关联元素不在数据库中(关乎关联列的错误id),致使Hiberante无法解决关联性问题时,Hibernate就会抛出异常. 这对遗留schema和历经拙劣维护的schema而言,这或有许多不便. 此时,你可用 @NotFound 注解让Hibernate略过这样的元素而不是抛出异常. 该注解可用于 @OneToOne (有外键)、 @ManyToOne 、 @OneToMany 或 @ManyToMany 关联.

|  |
| --- |
| @ManyToOne  @NotFound(action=NotFoundAction.IGNORE)  **public** Parent getParent() { ... } |

2.4.5.1. 延迟选项和获取模式

@LazyToOne注释

定义了 @ManyToOne 和 @OneToOne 关联的延迟选项. LazyToOneOption 可以是

* PROXY (例如:基于代理的延迟加载),
* NO\_PROXY (例如:基于字节码增强的延迟加载 - 注意需要在构建期处理字节码)
* FALSE (非延迟加载的关联)

@LazyCollection注释

定义了 @ManyToMany和 @OneToMany 关联的延迟选项. LazyCollectionOption 可以是

* TRUE (集合具有延迟性,只有在访问的时候才加载),
* EXTRA (集合具有延迟性,并且所有的操作都会尽量避免加载集合, 对于一个巨大的集合特别有用,因为这样的集合中的元素没有必要全部加载)
* FALSE (非延迟加载的关联)

@Fetch注释

定义了加载关联关系的获取策略. FetchMode 可以是

* SELECT (在需要加载关联的时候触发select操作),
* SUBSELECT (只对集合有效,使用了子查询策略,详情参考Hibernate参考文档) or
* JOIN (在加载主实体(owner entity)的时候使用SQL JOIN来加载关联关系). JOIN 将覆写任何延迟属性 (通过JOIN策略加载的关联将不再具有延迟性).

延迟和获取选项的等效注解

|  |  |  |
| --- | --- | --- |
| **Annotations** | **Lazy** | **Fetch** |
| @[One|Many]ToOne](fetch=FetchType.LAZY) | @LazyToOne(PROXY) | @Fetch(SELECT) |
| @[One|Many]ToOne](fetch=FetchType.EAGER) | @LazyToOne(FALSE) | @Fetch(JOIN) |
| @ManyTo[One|Many](fetch=FetchType.LAZY) | @LazyCollection(TRUE) | @Fetch(SELECT) |
| @ManyTo[One|Many](fetch=FetchType.EAGER) | @LazyCollection(FALSE) | @Fetch(JOIN) |

2.4.6. 关于集合类型的注解

#### 2.4.6.1. 参数注解

#### 以下是可能的设置方式

@BatchSizebatch注解

* 用@BatchSizebatch设置集合的batch大小

##### @Where注解

##### 用@Where注解设置Where子句

##### @Check注解

##### 用注解@Check来设置check子句

##### @OrderBy注解

##### 用注解@OrderBy来设置SQL的order by子句

##### @OnDelete注解

##### 利用@OnDelete(action=OnDeleteAction.CASCADE) 注解设置级连删除策略

##### @Sort注解

##### 利用@Sort注解定义一个排序比较器(sort comparator), 表明希望的比较器类型,无序、自然顺序或自定义排序,三者择一.若你想用你自己实现的comparator, 你还需要利用comparator属性(attribute)指明实现类.

|  |
| --- |
| @OneToMany(cascade=CascadeType.ALL, fetch=FetchType.*EAGER*)@JoinColumn(name="CUST\_ID")@Sort(type = SortType.*COMPARATOR*, comparator = TicketComparator.class)@Where(clause="1=1")@OnDelete(action=OnDeleteAction.*CASCADE*)public SortedSet<Ticket> getTickets() {return tickets;} |

2.4.6.2. 更多的集合类型

Hibernate Annotations支持真正的 List和Array. 映射集合的方式和以前完全一样,只不过要新增@IndexColumn注解. 该注解允许你指明存放索引值的字段.你还可以定义代表数据库中首个元素的索引值(亦称为索引基数). 常见取值为0或1.

@IndexColumn注解

|  |
| --- |
| @OneToMany(cascade = CascadeType.*ALL*)  @IndexColumn(name = "drawer\_position", base=1)  **public** List<Drawer> getDrawers() {  **return** drawers;  } |

注：假如你忘了设置@IndexColumn, Hibernate会采用包(bag)语义(译注：即允许重复元素的无序集合).

@MapKey注释

Hibernate注解支持true Map映射, 如果没有设置@javax.persistence.MapKey, hibernate将key元素或嵌入式对象直接映射到他们所属的列. 要覆写默认的列,可以使用以下注解: @org.hibernate.annotations.MapKey适用的key为基本类型或者嵌入式对象, **@org.hibernate.annotations.MapKey适用的key为实体**.

@ElementCollection注解(CollectionOfElements)

Hibernate Annotations还支持核心类型集合(Integer, String, Enums, ......)、 可内嵌对象的集合,甚至基本类型数组.这就是所谓的元素集合.

元素集合可用@ElementCollection来注解(作为@OneToMany的替代). 为了定义集合表(译注：即存放集合元素的表,与下面提到的主表对应),要在关联属性上使用@JoinTable注解, joinColumns定义了介乎实体主表与集合表之间的连接字段(inverseJoincolumn是无效的且其值应为空). 对于核心类型的集合或基本类型数组,你可以在关联属性上用@Column来覆盖存放元素值的字段的定义. 你还可以用@AttributeOverride来覆盖存放可内嵌对象的字段的定义. 要访问集合元素,需要将该注解的name属性值设置为"element"("element"用于核心类型,而"element.serial" 用于嵌入式对象的serial属性).要访问集合的index/key,则将该注解的name属性值设置为"key".

@CollectionOfElements

@JoinTable 定义关联表 (@joinColumns定义关联字段)

@Column 定义存放元素值的字段

@AttributeOverride 覆盖存放可内嵌对象的字段的定义(访问集合元素,需要将该注解的name属性值设置为"element"("element"用于核心类型,而"element.serial" 用于嵌入式对象的serial属性).要访问集合的index/key,则将该注解的name属性值设置为"key")

@indexColumn 定义排序字段 与 @CollectionId 功能一致

@org.hibernate.annotations.MapKey定义map的Key字段

|  |
| --- |
| @Entity  **public** **class** Boy {  @Id @GeneratedValue(strategy=GenerationType.*AUTO*)  **private** Integer id;    @ElementCollection  **private** Set<String> nickNames = **new** HashSet<String>();    @ElementCollection  @JoinTable(  name="BoyFavoriteNumbers",  joinColumns = @JoinColumn(name="BoyId")  )  @Column(name="favoriteNumber", nullable=**false**)  @IndexColumn(name="nbr\_index")  **private** **int**[] favoriteNumbers;    @ElementCollection// Toy嵌入对象  @AttributeOverride( name="element.serial", column=@Column(name="serial\_nbr") )  **private** Set<Toy> favoriteToys = **new** HashSet<Toy>();    @ElementCollection// Character枚举类型  **private** Set<Character> characters = **new** HashSet<Character>();  @ElementCollection// Contact嵌入对象  @CollectionId(columns=@Column(name="boy\_n\_contact"),type=@Type(type="long"),generator="increment")  @GenericGenerator(name="increment",strategy="increment")  **private** List<Contact> contacts = **new** ArrayList<Contact>();    @ElementCollection  @IndexColumn(name="games\_index")  @Column(name="game")  **private** List<String> games = **new** ArrayList<String>();    @ElementCollection//定义Map的key在数据库的字段  @org.hibernate.annotations.MapKey( columns = @Column( name = "form\_name" ) )  @Column(name="form\_url")//定义Map的value在数据库的字段  **private** Map<String,String> forms = **new** HashMap<String,String>();  @ManyToMany @Cascade(CascadeType.*SAVE\_UPDATE*) // Gril实体对象  @JoinTable(name="boy\_gril",joinColumns=@JoinColumn(name="boyID"),inverseJoinColumns=@JoinColumn(name="girlID"))  @MapKey(columns=@Column(name="gril\_name")) // 定义map的key在数据库字段  **private** Map<String,Gril> grils = **new** HashMap<String, Gril>();  } |

定义Map中的实体对象和嵌入对象的实例(Luggage是接口, LuggageImpl是实体对象,Size是嵌入对象,注意@**MapKeyManyToMany**注释)

|  |
| --- |
| **public** **class** Brand {  @Id@GeneratedValue  **private** Long id;  @ManyToMany(targetEntity = LuggageImpl.**class**)  @JoinTable(name="b\_size\_lug",joinColumns=@JoinColumn(name="b\_ID"),inverseJoinColumns=@JoinColumn(name="lug\_id"))  @MapKey(targetElement = SizeImpl.**class**)  **private** Map<Size, Luggage> luggagesBySize = **new** HashMap<Size, Luggage>();  @ElementCollection(targetElement = SizeImpl.**class**)  @JoinTable(name="b\_lug\_size",joinColumns=@JoinColumn(name="brane\_ID"))  @**MapKeyManyToMany**(joinColumns=@JoinColumn(name="lu\_id"), targetEntity = LuggageImpl.**class**)  @AttributeOverride(name="element.name",column=@Column(name="size\_name"))  **private** Map<Luggage, Size> sizePerLuggage = **new** HashMap<Luggage, Size>();  } |

2.4.7 缓存

@Cache注解

(usage = CacheConcurrencyStrategy.NONSTRICT\_READ\_WRITE)

缓存的方式有四种，分别为：

* CacheConcurrencyStrategy.NONE
* CacheConcurrencyStrategy.READ\_ONLY，只读模式，在此模式下，如果对数据进行更新操作，会有异常；
* CacheConcurrencyStrategy.READ\_WRITE，读写模式在更新缓存的时候会把缓存里面的数据换成一个锁，其它事务如果去取相应的缓存数据，发现被锁了直接就去数据库查询；
* CacheConcurrencyStrategy.NONSTRICT\_READ\_WRITE，不严格的读写模式则不会对缓存数据加锁；
* CacheConcurrencyStrategy.TRANSACTIONAL，事务模式指缓存支持事务，当事务回滚时，缓存也能回滚，只支持JTA环境。

2.4.8 接口映射

@Any注解

|  |
| --- |
| @Any( metaColumn = @Column( name = "property\_type" ) )  @Cascade( value = { CascadeType.*ALL* } )  @AnyMetaDef( idType = "integer", metaType = "string", metaValues = {  @MetaValue( value = "S", targetEntity = StringProperty.**class** ),  @MetaValue( value = "I", targetEntity = IntegerProperty.**class** )  } )  @JoinColumn( name = "property\_id" )  **public** Property getSomeProperty() {  **return** someProperty;  }  @Entity  @Table(name="string\_property")  **public** **class** StringProperty **implements** Property |

@ManyToAny注解

|  |
| --- |
| @ManyToAny(  metaColumn = @Column( name = "property\_type" ) )  @AnyMetaDef( idType = "integer", metaType = "string",  metaValues = {  @MetaValue( value = "S", targetEntity = StringProperty.**class** ),  @MetaValue( value = "I", targetEntity = IntegerProperty.**class** ) } )  @Cascade( { org.hibernate.annotations.CascadeType.*ALL* } )  @JoinTable( name = "obj\_properties", joinColumns = @JoinColumn( name = "obj\_id" ),  inverseJoinColumns = @JoinColumn( name = "property\_id" ) )  **public** List<Property> getGeneralProperties() {  **return** generalProperties;  } |

2.5 注意事项

1、@Column(name="`title`") 定义中含有’将被忽略

2、关于泛型定义(就当成普通的对象，定义为多对一或者一对一就行,不用指定ManyToOne的targetClass)

|  |
| --- |
| @MappedSuperclass  **public** **class** AItem<Type, Owner> **extends** APricedStuff {  @Id @GeneratedValue(strategy=GenerationType.*AUTO*)  **private** Integer id;  @Basic  **private** String name;  @ManyToOne  @Cascade(CascadeType.*SAVE\_UPDATE*)@JoinColumn(name="type\_ID")  **private** Type type;  @ManyToOne  @Cascade(CascadeType.*SAVE\_UPDATE*)@JoinColumn(name="guy\_ID")  **private** Owner owner;  } |

3、在一对一，一对多，以及多对多中，不定义**@Cascade**则不会在数据中产生外键关联(可以通过@NotFound(action=NotFoundAction.IGNORE)避免数据库或者程序出现的空指针)

Spring3MVC

<!-- Maps '/' requests to the 'home' view -->

<mvc:view-controller path="/" view-name="home"/>

1. 注解

@Controller

@RequestMapping

|  |
| --- |
| @RequestMapping(value="/simple/revisited",method=RequestMethod.*GET*,headers="Accept=text/plain")  @RequestMapping("/mapping/path")  **public** @ResponseBody String byPath()  实际请求路径/spring3/mapping/path/wildcard  @RequestMapping(value="/mapping/path/\*", method=RequestMethod.*GET*)  **public** @ResponseBody String byPathPattern(HttpServletRequest request)  实际请求路径含有参数foo=bar时进入方法byParameter(),不含有foo=bar时进入byParameterNegation()  @RequestMapping(value="/mapping/parameter", method=RequestMethod.*GET*, params="foo")  **public** @ResponseBody String byParameter()  @RequestMapping(value="/mapping/parameter", method=RequestMethod.*GET*, params="!foo")  **public** @ResponseBody String byParameterNegation()  实际请求: $.ajax({ url:’ /mapping/header’, dataType: "text", beforeSend: function(req) { req.setRequestHeader("FooHeader", "foo"); }});  有头信息的进入byHeader()方法，没有头信息的进入byHeaderNegation()  @RequestMapping(value="/mapping/header",method=RequestMethod.*GET*,headers="FooHeader=foo")  **public** @ResponseBody String byHeader()  @RequestMapping(value="/mapping/header", method=RequestMethod.*GET*, headers="!FooHeader")  **public** @ResponseBody String byHeaderNegation()  @RequestMapping(value="/mapping/consumes",method=RequestMethod.*POST*,consumes="application/json")  @RequestMapping(value="/mapping/produces",method=RequestMethod.*GET*,produces="application/json")  consumes/produces的意思??  @RequestMapping("/data/\*")  **public** **class** RequestDataController {  实际请求:/data/param  @RequestMapping(value="param")  **public** @ResponseBody String withParam(@RequestParam String foo)  } |

@RequestParam

|  |
| --- |
| bind request parameters to a method parameter in your controller  获得参数foo  @RequestMapping(value="param")  **public** @ResponseBody String withParam(@RequestParam String foo)  或者  **public** @ResponseBody String withParam(@RequestParam String[] foo)  实际地址: /data/group?param1=foo&param2=bar&param3=baz,组装javabean  @RequestMapping(value="group")  **public** @ResponseBody String withParamGroup(JavaBean bean) |

@ModelAttribute

|  |
| --- |
| 将属性保存在request中  两种用法：1、放在方法上，表示添加一个属性到model中，值为return的值，同时返回页面根据@RequestMapping的值返回页面  2、在参数中，表示添加一个属性到model中，值为参数的值，返回页面根据return的值进行返回  @RequestMapping(value="/test/test1")  @ModelAttribute("id")  **public** String test1(String id,String user){  **return** id+user;  }  @RequestMapping(value="/test/test12")  **public** String test12(@ModelAttribute("id")String id){  **return** "/test/test1";  }  请求路径: $.ajax({ type: "POST", url: "/messageconverters/form"  data: "foo=bar&fruit=apple", contentType: "application/x-www-form-urlencoded", dataType: "text"}),自动组装javaBean  @RequestMapping(value="/form", method=RequestMethod.*POST*)  **public** @ResponseBody String readForm(@ModelAttribute JavaBean bean) {  **return** "Read x-www-form-urlencoded: " + bean;  } |

@SessionAttributes

|  |
| --- |
| 将属性保存在session中 |

@PathVariable

|  |
| --- |
| 请求路径: /data/path/foo  @RequestMapping(value="path/{var}")注:红色保持一致  **public** @ResponseBody String withPathVariable(@PathVariable String var)  @RequestMapping(value="pathVariables/{foo}/{fruit}", method=RequestMethod.*GET*)  **public** String pathVars(@PathVariable String foo, @PathVariable String fruit) {  // No need to add @PathVariables "foo" and "fruit" to the model  // They will be merged in the model before rendering  // 请求路径:views/pathVariables/bar/apple传递值 bar和apple不需要放入model中，  //可以直接返回页面，他们自动将合并到model中进行页面渲染  **return** "views/html";  }  @RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)  public String findOwner(@PathVariable("ownerId") String theOwner, Model model) |

@RequestHeader

|  |
| --- |
| 请求路径: /data/header  @RequestMapping(value="header")  **public** @ResponseBody String withHeader(@RequestHeader String Accept) |

@RequestBody

|  |
| --- |
| 请求路径: $.ajax({ type: "POST", url: "/data/body",  data: "foo", contentType: "text/plain", dataType: "text"});  @RequestMapping(value="body", method=RequestMethod.*POST*)  **public** @ResponseBody String withBody(@RequestBody String body)  body的值为foo  请求路径: /data/entity.  @RequestMapping(value="entity", method=RequestMethod.*POST*)  **public** @ResponseBody String withEntity(HttpEntity<String> entity) {  **return** "Posted request body '" + entity.getBody() + "'; headers = " + entity.getHeaders();  } |

@ResponseBody

|  |
| --- |
| the return type is written to the response HTTP body  直接将返回信息写入到respose流中，不跳转页面  @RequestMapping(value="/response/annotation", method=RequestMethod.*GET*)  **public** @ResponseBody String responseBody() {  **return** "The String ResponseBody";  } |

@Valid

|  |
| --- |
| // enforcement of constraints on the JavaBean arg require a JSR-303 provider on the classpath  @RequestMapping("/validate")  **public** @ResponseBody String validate(@Valid JavaBean bean, BindingResult result) {  **if** (result.hasErrors()) {  **return** "Object has validation errors";  } **else** {  **return** "No errors";  }  } |

@CookieValue

|  |
| --- |
| 允许直接从cookie里读值  **public** @ResponseBody String cookie1(@CookieValue("JSESSIONID") String cookie) |

@SessionAttributes

|  |
| --- |
| Using @SessionAttributes to store model attributes in the HTTP session between requests |

@InitBinder

|  |
| --- |
| @InitBinder  **public** **void** initBinder(WebDataBinder binder) {  SimpleDateFormat dateFormat = **new** SimpleDateFormat("yyyy-MM-dd");  dateFormat.setLenient(**false**);  binder.registerCustomEditor(Date.**class**, **new** CustomDateEditor(dateFormat, **false**));  } |

类型转换

|  |
| --- |
| 数字  请求路径: /convert/primitive?value=3  @RequestMapping("primitive")  **public** @ResponseBody String primitive(@RequestParam Integer value)  日期  请求路径: /convert/date/2010-07-04  // requires Joda-Time on the classpath  @RequestMapping("date/{value}")  **public** @ResponseBody String date(@PathVariable @DateTimeFormat(iso=ISO.*DATE*) Date value)  集合  请求路径: /convert/collection?values=1&values=2&values=3&values=4&values=5  或者: /convert/collection?values=1,2,3,4,5  @RequestMapping("collection")  **public** @ResponseBody String collection(@RequestParam Collection<Integer> values)  集合<日期>  请求路径: /convert/formattedCollection?values=2010-07-04,2011-07-04  @RequestMapping("formattedCollection")  **public** @ResponseBody String formattedCollection(@RequestParam @DateTimeFormat  (iso=ISO.*DATE*) Collection<Date> values)  bean  请求路径: 属性数字/convert/bean?primitive=3  属性集合/convert/bean?list[0]=1&list[1]=2&list[2]=3  属性日期/convert/bean?date=2010-07-04  属性map/convert/bean?map[0]=apple&map[1]=pear  属性集合<日期>/convert/bean?formattedList[0]=2010-07-04&formattedList[1]=2011-07-04  属性内置对象/convert/bean?nested.foo=bar&nested.list[0].foo=baz &nested.map[key].list[0].foo=bip  @RequestMapping("bean")  **public** @ResponseBody String bean(JavaBean bean) |

页面跳转/返回值

|  |
| --- |
| 返回JSP页面  @RequestMapping(value="html", method=RequestMethod.*GET*)  **public** String prepare(Model model) {  model.addAttribute("foo", "bar");  model.addAttribute("fruit", "apple");  **return** "views/html";  }  返回viewName.jsp页面  @RequestMapping(value="/viewName", method=RequestMethod.*GET*)  **public** **void** usingRequestToViewNameTranslator(Model model) {  model.addAttribute("foo", "bar");  model.addAttribute("fruit", "apple");  }  采用redirect方式  @RequestMapping(value="/uriTemplate", method=RequestMethod.*GET*)  **public** String uriTemplate(RedirectAttributes redirectAttrs) {  redirectAttrs.addAttribute("account", "a123"); // Used as URI template variable  redirectAttrs.addAttribute("date", **new** LocalDate(2011, 12, 31)); // Appended as a query parameter  **return** "redirect:/redirect/{account}";  }  @RequestMapping(value="/uriComponentsBuilder", method=RequestMethod.*GET*)  **public** String uriComponentsBuilder() {  UriComponents redirectUri = UriComponentsBuilder.*fromPath*("/redirect/{account}").queryParam("date", date)  .build().expand("a123").encode();  **return** "redirect:" + redirectUri.toUriString();  }  采用forward方式  return "forward:/order/add";  直接返回字符串或者JSON串  @RequestMapping("/simple")  **public** @ResponseBody String simple() {  **return** "Hello world!";  }  当javaBean没有任何注解时，默认通过json解析输出  @RequestMapping(value="/mapping/produces", method=RequestMethod.*GET*)  **public** @ResponseBody JavaBean byProduces() {  **return** **new** JavaBean();  }  ModelAndView  @RequestMapping("/modelview")  public ModelAndView view(Model model) {  logger.info("view 方法调用");  ModelAndView andView = new ModelAndView();  andView.setViewName("WEB-INF/jsp/success");  return andView;  }  ResponseEntity  @RequestMapping(value="/response/entity/headers", method=RequestMethod.*GET*)  **public** ResponseEntity<String> responseEntityCustomHeaders() {  HttpHeaders headers = **new** HttpHeaders();  headers.setContentType(MediaType.*TEXT\_PLAIN*);  **return** **new** ResponseEntity<String>(" custom header Content-Type=text/plain",  headers, HttpStatus.*OK*);  }  /\*\*  \* @see 使用map作为返回值的时候 是以prefix前缀+requestMapping的value+suffix后缀组成  \* 返回一个map ,map的put方法调用相当于request.setAttribute方法，返回访问mapa.jsp  \* \*/  @RequestMapping("/mapa")  public Map<String, Object> mapa(ModelMap map1) {  Map<String, Object> map = new HashMap<String, Object>();  UserBean bean = new UserBean();  bean.setId(1);  bean.setUsername("Edward Lau");  bean.setPassword("edward");  map.put("hello", "world key");  map.put("user", bean);  return map;  }  MultiValueMap  @RequestMapping(value="/form", method=RequestMethod.*GET*)  **public** @ResponseBody MultiValueMap<String, String> writeForm(@RequestHeader String Accept) {  MultiValueMap<String, String> map = **new** LinkedMultiValueMap<String, String>();  map.add("foo", "bar");  map.add("fruit", "apple");  **return** map;  }  Jaxb2RootElementHttpMessageConverter (requires JAXB2 on the classpath)  Xml输出:javabean通过@XmlRootElement定义，自动将bean返回为xml  @RequestMapping(value="/xml", method=RequestMethod.*GET*)  **public** @ResponseBody JavaBean writeXml() {  **return** **new** JavaBean("bar", "fruit");  }  // AtomFeedHttpMessageConverter (requires Rome on the classpath)  xml输出:atom  @RequestMapping(value="/atom", method=RequestMethod.*GET*)  **public** @ResponseBody Feed writeFeed() {  Feed feed = **new** Feed();  feed.setFeedType("atom\_1.0");  feed.setTitle("My Atom feed");  **return** feed;  }  // RssChannelHttpMessageConverter (requires Rome on the classpath)  xml输出:rss  @RequestMapping(value="/rss", method=RequestMethod.*GET*)  **public** @ResponseBody Channel writeChannel() {  Channel channel = **new** Channel();  channel.setFeedType("rss\_2.0");  channel.setTitle("My RSS feed");  channel.setDescription("Description");  channel.setLink("http://localhost:8080/spring3/rss");  **return** channel;  } |

自动绑定bean

|  |
| --- |
| /data/group?param1=foo&param2=bar&param3=baz,组装javabean  @RequestMapping(value="group")  **public** @ResponseBody String withParamGroup(JavaBean bean)  通过@ModelAttribute自动绑定  @RequestMapping(value="/form", method=RequestMethod.*POST*)  **public** @ResponseBody String readForm(@ModelAttribute JavaBean bean) {  **return** "Read x-www-form-urlencoded: " + bean;  }  // Jaxb2RootElementHttpMessageConverter (requires JAXB2 on the classpath)  通过输入xml自动绑定到bean, javabean通过@XmlRootElement定义  @RequestMapping(value="/xml", method=RequestMethod.*POST*)  **public** @ResponseBody String readXml(@RequestBody JavaBean bean)  // MappingJacksonHttpMessageConverter (requires Jackson on the classpath)  通过输入json自动绑定到bean  @RequestMapping(value="/json", method=RequestMethod.*POST*)  **public** @ResponseBody String readJson(@Valid @RequestBody JavaBean bean)  通过URL参数自动绑定bean并且放入model中  @RequestMapping(value="dataBinding/{foo}/{fruit}", method=RequestMethod.*GET*)  **public** String dataBinding(@Valid JavaBean javaBean, Model model) {  // JavaBean "foo" and "fruit" properties populated from URI variables  **return** "views/dataBinding";  } |

传值

|  |
| --- |
| 通过WebRequest获得值，通过model传值  @RequestMapping(value = "/search", method = RequestMethod.GET)  public void search(WebRequest request, Model model)  通过Model进行传值  @RequestMapping(value="html", method=RequestMethod.*GET*)  **public** String prepare(Model model) {  model.addAttribute("foo", "bar");  model.addAttribute("fruit", "apple");  **return** "views/html";  }  URL直接传值渲染  @RequestMapping(value="pathVariables/{foo}/{fruit}", method=RequestMethod.*GET*)  **public** String pathVars(@PathVariable String foo, @PathVariable String fruit) {  // No need to add @PathVariables "foo" and "fruit" to the model  // They will be merged in the model before rendering  // 请求路径:views/pathVariables/bar/apple传递值 bar和apple不需要放入model中，  //可以直接返回页面，他们自动将合并到model中进行页面渲染  **return** "views/html";  } |

文件上传

|  |
| --- |
| @RequestMapping(method=RequestMethod.*POST*)  **public** **void** processUpload(@RequestParam MultipartFile file, Model model) **throws** IOException {  model.addAttribute("message", "File '" + file.getOriginalFilename() + "' uploaded successfully");  } |

处理异常

|  |
| --- |
| @ExceptionHandler  **public** @ResponseBody String handle(IllegalStateException e) {  **return** "IllegalStateException handled!";  }    @RequestMapping("/exception")  **public** @ResponseBody String exception() {  **throw** **new** IllegalStateException("Sorry!");  } |

HttpSession/tpServletRequest/HttpServletResponse

|  |
| --- |
| 请求路径: /data/standard/request  @RequestMapping(value="/data/standard/request", method=RequestMethod.*GET*)  **public** @ResponseBody String standardRequestArgs(HttpServletRequest request, Principal user, Locale locale)  请求:$.ajax({ type: "POST", url: url, data: "foo", contentType: "text/plain", dataType: "text"});  @RequestMapping(value="/data/standard/request/reader", method=RequestMethod.*POST*)  **public** @ResponseBody String requestReader(Reader requestBodyReader) **throws** IOException {  **return** "Read char request body = " + FileCopyUtils.*copyToString*(requestBodyReader);  }  @RequestMapping(value="/data/standard/request/is", method=RequestMethod.*POST*)  **public** @ResponseBody String requestReader(InputStream requestBodyIs) **throws** IOException {  **return** "Read binary request body = " + **new**String(FileCopyUtils.*copyToByteArray*(requestBodyIs  ));  }  请求路径: /data/standard/ response  @RequestMapping("/data/standard/response")  **public** @ResponseBody String response(HttpServletResponse response)  处理流  @RequestMapping("/data/standard/response/writer")  **public** **void** availableStandardResponseArguments(Writer responseWriter) **throws** IOException {  responseWriter.write("Wrote char response using Writer");  }  @RequestMapping("/data/standard/response/os")  **public** **void** availableStandardResponseArguments(OutputStream os) **throws** IOException {  os.write("Wrote binary response using OutputStream".getBytes());  }  处理session  @RequestMapping("/data/standard/session")  **public** @ResponseBody String session(HttpSession session)  RedirectAttributes  @RequestMapping(value="/uriTemplate", method=RequestMethod.*GET*)  **public** String uriTemplate(RedirectAttributes redirectAttrs)  BindingResult  **public** @ResponseBody String validate(@Valid JavaBean bean, BindingResult result) |

1. 配置
   1. 接口

* DispatcherServlet -- 前置控制器
* HandlerMapping接口 -- 处理请求的映射
  + SimpleUrlHandlerMapping 通过配置文件，把一个URL映射到Controller
  + DefaultAnnotationHandlerMapping 通过注解，把一个URL映射到Controller类上
* HandlerAdapter接口 -- 处理请求的映射
* Controller接口 -- 控制器
* HandlerInterceptor 接口--拦截器
* ViewResolver接口
* LocalResolver接口
* HandlerExceptionResolver接口 --异常处理
  1. Servlet拦截匹配

当映射为@RequestMapping("/user/add")时：

1. 拦截\*.do，例如：/user/add.do，弊端：所有的url都要以.do结尾。不会影响访问静态文件。
2. 拦截/app/\*，例如：/app/user/add，弊端：请求的url都要包含/app，@RequestMapping("/user/add")中不须要包含/app。
3. 拦截/，例如：/user/add，弊端：对jpg,js,css静态文件的访问也被拦截不能正常显示。后面有解决办法。
4. 拦截/\*，可以走到Action中，但转发到jsp时再次被拦截，不能访问到jsp。
   1. 配置讲解

|  |
| --- |
| <!-- 自动扫描的包名 -->  <context:component-scan base-package="com.app,com.core,JUnit4"/>  <!-- 默认的注解映射的支持 -->  <mvc:annotation-driven />  <!-- 视图解释类 -->  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <property name="prefix" value="/WEB-INF/jsp/" />  <property name="suffix" value=".jsp" />  <!--可为空,方便实现自已的依据扩展名来选择视图解释类的逻辑 -->  </bean>  <!-- 拦截器 -->  <mvc:interceptors>  <bean class="com.core.mvc.MyInteceptor" />  </mvc:interceptors>  <!-- 对静态资源文件的访问 方案一 （二选一） -->  <mvc:default-servlet-handler />  <!-- 对静态资源文件的访问 方案二 （二选一）-->  <mvc:resources mapping="/images/\*\*" location="/images/" cache-period="31556926" />  <mvc:resources mapping="/js/\*\*" location="/js/" cache-period="31556926" />  <mvc:resources mapping="/css/\*\*" location="/css/" cache-period="31556926" /> |

<mvc:annotation-driven /> 是一种简写形式，完全可以手动配置替代这种简写形式，简写形式可以让初学都快速应用默认配置方案。<mvc:annotation-driven /> 会自动注册DefaultAnnotationHandlerMapping与AnnotationMethodHandlerAdapter 两个bean,是spring MVC为@Controllers分发请求所必须的。并提供了：数据绑定支持，@NumberFormatannotation支持，@DateTimeFormat支持，@Valid支持，读写XML的支持（JAXB），读写JSON的支持（Jackson）。

<mvc:interceptors/> 是一种简写形式。通过看前面的大图，知道，我们可以配置多个HandlerMapping。<mvc:interceptors/>会为每一个HandlerMapping，注入一个拦截器。其实我们也可以手动配置为每个HandlerMapping注入一个拦截器。

<mvc:default-servlet-handler/> 使用默认的Servlet来响应静态文件。

<mvc:resources mapping="/images/\*\*" location="/images/" cache-period="31556926"/> 匹配URL /images/\*\* 的URL被当做静态资源，由Spring读出到内存中再响应http。

* 1. 静态文件

如何你的DispatcherServlet拦截 \*.do这样的URL，就不存在访问不到静态资源的问题。如果你的 DispatcherServlet拦截“/”，拦截了所有的请求，同时对\*.js,\*.jpg的访问也就被拦截了。

目的：可以正常访问静态文件，不要找不到静态文件报404。

* + 1. 方案一 激活Tomcat的defaultServlet来处理静态文件

|  |
| --- |
| <servlet-mapping>  <servlet-name>default</servlet-name>  <url-pattern>\*.jpg</url-pattern>  </servlet-mapping>  <servlet-mapping>  <servlet-name>default</servlet-name>  <url-pattern>\*.js</url-pattern>  </servlet-mapping>  <servlet-mapping>  <servlet-name>default</servlet-name>  <url-pattern>\*.css</url-pattern>  </servlet-mapping>  要配置多个，每种文件配置一个 |

要写在DispatcherServlet的前面， 让 defaultServlet先拦截，这个就不会进入Spring了，我想性能是最好的吧。

Tomcat, Jetty, JBoss, and GlassFish 默认 Servlet的名字 -- "default"

Google App Engine 默认 Servlet的名字 -- "\_ah\_default"

Resin 默认 Servlet的名字 -- "resin-file"

WebLogic 默认 Servlet的名字 -- "FileServlet"

WebSphere 默认 Servlet的名字 -- "SimpleFileServlet"

* + 1. 方案二 在spring3.0.4以后版本提供了mvc:resources

mvc:resources 的使用方法：

|  |
| --- |
| <!-- 对静态资源文件的访问 -->  <mvc:resources mapping="/images/\*\*" location="/images/" /> |

/images/\*\*映射到ResourceHttpRequestHandler进行处理，location指定静态资源的位置.可以是web application根目录下、jar包里面，这样可以把静态资源压缩到jar包中。cache-period 可以使得静态资源进行web cache

如果出现下面的错误，可能是没有配置<mvc:annotation-driven />的原因。

报错WARNING: No mapping found for HTTP request with URI [/mvc/user/findUser/lisi/770] in DispatcherServlet with name 'springMVC'

使用<mvc:resources/>元素,把mapping的URI注册到SimpleUrlHandlerMapping的urlMap中,

key为mapping的URI pattern值,而value为ResourceHttpRequestHandler,这样就巧妙的把对静态资源的访问由HandlerMapping转到ResourceHttpRequestHandler处理并返回,所以就支持classpath目录,jar包内静态资源的访问.另外需要注意的一点是,不要对SimpleUrlHandlerMapping设置defaultHandler.因为对static uri的defaultHandler就是ResourceHttpRequestHandler,

否则无法处理static resources request.

* + 1. 方案三 使用<mvc:default-servlet-handler/>

|  |
| --- |
| <mvc:default-servlet-handler/> |

会把"/\*\*" url,注册到SimpleUrlHandlerMapping的urlMap中,把对静态资源的访问由HandlerMapping转到org.springframework.web.servlet.resource.DefaultServletHttpRequestHandler处理并返回.

DefaultServletHttpRequestHandler使用就是各个Servlet容器自己的默认Servlet.

补充说明：多个HandlerMapping的执行顺序问题：

DefaultAnnotationHandlerMapping的order属性值是：0

<mvc:resources/ >自动注册的 SimpleUrlHandlerMapping的order属性值是： 2147483646

<mvc:default-servlet-handler/>自动注册 的SimpleUrlHandlerMapping 的order属性值是： 2147483647

spring会先执行order值比较小的。当访问一个a.jpg图片文件时，先通过 DefaultAnnotationHandlerMapping 来找处理器，一定是找不到的，我们没有叫a.jpg的Action。再 按order值升序找，由于最后一个 SimpleUrlHandlerMapping 是匹配 "/\*\*"的，所以一定会匹配上，再响应图片。

访问一个图片，还要走层层匹配。真不知性能如何？改天做一下压力测试，与Apache比一比。

最后再说明一下，如何你的DispatcherServlet拦截 \*.do这样的URL，就不存上述问题了。

* 1. 拦截器
     1. Spring中的拦截器

Spring为我们提供了：

org.springframework.web.servlet.HandlerInterceptor接口，

org.springframework.web.servlet.handler.HandlerInterceptorAdapter适配器，

实现这个接口或继承此类，可以非常方便的实现自己的拦截器。

有以下三个方法：

Action之前执行:

public boolean preHandle(HttpServletRequest request,

HttpServletResponse response, Object handler);

生成视图之前执行

public void postHandle(HttpServletRequest request,

HttpServletResponse response, Object handler,

ModelAndView modelAndView);

最后执行，可用于释放资源

public void afterCompletion(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception ex)

分别实现预处理、后处理（调用了Service并返回ModelAndView，但未进行页面渲染）、返回处理（已经渲染了页面）

在preHandle中，可以进行编码、安全控制等处理；

在postHandle中，有机会修改ModelAndView；

在afterCompletion中，可以根据ex是否为null判断是否发生了异常，进行日志记录。

参数中的Object handler是下一个拦截器。

* + 1. 使用拦截器

Spring MVC并没有总的拦截器，不能对所有的请求进行前后拦截。

Spring MVC的拦截器，是属于HandlerMapping级别的，可以有多个HandlerMapping ，每个HandlerMapping可以有自己的拦截器。

当一个请求按Order值从小到大，顺序执行HandlerMapping接口的实现类时，哪一个先有返回，那就可以结束了，后面的HandlerMapping就不走了，本道工序就完成了。就转到下一道工序了。

拦截器会在什么时候执行呢？ 一个请求交给一个HandlerMapping时，这个HandlerMapping先找有没有处理器来处理这个请求，如何找到了，就执行拦截器，执行完拦截后，交给目标处理器。

如果没有找到处理器，那么这个拦截器就不会被执行。

在spring MVC的配置文件中配置有三种方法：

方案一，（近似）总拦截器，拦截所有url

|  |
| --- |
| <mvc:interceptors>  <bean class="com.app.mvc.MyInteceptor" />  </mvc:interceptors> |

为什么叫“近似”，前面说了，Spring没有总的拦截器。

<mvc:interceptors/>会为每一 个HandlerMapping，注入一个拦截器。总有一个HandlerMapping是可以找到处理器的，最多也只找到一个处理器，所以这个拦截器总会被执行的。起到了总拦截器的作用。

方案二， （近似） 总拦截器， 拦截匹配的URL。

|  |
| --- |
| <mvc:interceptors >  <mvc:interceptor>  <mvc:mapping path="/user/\*" /> <!-- /user/\* -->  <bean class="com.mvc.MyInteceptor"></bean>  </mvc:interceptor>  </mvc:interceptors> |

方案三,HandlerMappint上的拦截器

|  |
| --- |
| <bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">  <property name="interceptors">  <list>  <bean class="com.mvc.MyInteceptor"></bean>  </list>  </property>  </bean> |

如果使用了<mvc:annotation-driven />， 它会自动注册DefaultAnnotationHandlerMapping 与AnnotationMethodHandlerAdapter 这两个bean,所以就没有机会再给它注入interceptors属性，就无法指定拦截器。

当然我们可以通过人工配置上面的两个Bean，不使用 <mvc:annotation-driven />，就可以 给interceptors属性 注入拦截器了。

其实我也不建议使用<mvc:annotation-driven />，而建议手动写配置文件，来替代 <mvc:annotation-driven />，这就控制力就强了。

* 1. 异常

|  |
| --- |
| <!-- 总错误处理-->  <bean id="exceptionResolver" class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">  <property name="defaultErrorView">  <value>/error/error</value>  </property>  <property name="defaultStatusCode">  <value>500</value>  </property>  <property name="warnLogCategory">  <value>org.springframework.web.servlet.handler.SimpleMappingExceptionResolver</value>  </property>  </bean> |

这里主要的类是SimpleMappingExceptionResolver类，和他的父类AbstractHandlerExceptionResolver类。

具体可以配置哪些属性，我是通过查看源码知道的。

你也可以实现HandlerExceptionResolver接口，写一个自己的异常处理程序。spring的扩展性是很好的。

通过SimpleMappingExceptionResolver我们可以将不同的异常映射到不同的jsp页面（通过exceptionMappings属性的配置）。

同时我们也可以为所有的异常指定一个默认的异常提示页面（通过defaultErrorView属性的配置），如果所抛出的异常在exceptionMappings中没有对应的映射，则Spring将用此默认配置显示异常信息。

注意这里配置的异常显示界面均仅包括主文件名，至于文件路径和后缀已经在viewResolver中指定。如/error/error表示/error/error.jsp

显示错误的jsp页面：

|  |
| --- |
| <%@ page language="java" contentType="text/html; charset=GBK"  pageEncoding="GBK"%>  <%@ page import="java.lang.Exception"%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv="Content-Type" content="text/html; charset=GBK">  <title>错误页面</title>  </head>  <body>  <h1>出错了</h1>  <%  Exception e = (Exception)request.getAttribute("exception");  out.print(e.getMessage());  %>  </body>  </html> |

其中一句：request.getAttribute("exception")，key是exception，也是在SimpleMappingExceptionResolver类默认指定的，是可能通过配置文件修改这个值的，大家可以去看源码。

如何把全局异常记录到日志中？

在前的配置中，其中有一个属性warnLogCategory，值是“SimpleMappingExceptionResolver类的全限定名”。我是在SimpleMappingExceptionResolver类父类AbstractHandlerExceptionResolver类中找到这个属性的。查看源码后得知：如果warnLogCategory不为空，spring就会使用apache的org.apache.commons.logging.Log日志工具，记录这个异常，级别是warn。

值：“org.springframework.web.servlet.handler.SimpleMappingExceptionResolver”，是“SimpleMappingExceptionResolver类的全限定名”。这个值不是随便写的。 因为我在log4j的配置文件中还要加入log4j.logger.org.springframework.web.servlet.handler.  
SimpleMappingExceptionResolver=WARN，保证这个级别是warn的日志一定会被记录，即使log4j的根日志级别是ERROR。

1. Media Type

* Type application:For Multipurpose files.
  + application/atom+xml: Atom feeds
  + application/ecmascript: ECMAScript/JavaScript; Defined in RFC 4329 (equivalent to application/javascript but with stricter processing rules)
  + application/EDI-X12: EDI X12 data; Defined in RFC 1767
  + application/EDIFACT: EDI EDIFACT data; Defined in RFC 1767
  + application/json: JavaScript Object Notation JSON; Defined in RFC 4627
  + application/javascript: ECMAScript/JavaScript;
  + application/octet-stream: Arbitrary binary data.
  + application/ogg: Ogg, a multimedia bitstream container format; Defined in RFC 5334
  + application/pdf: Portable Document Format, PDF has been in use for document exchange on the Internet since 1993; Defined in RFC 3778
  + application/postscript: PostScript; Defined in RFC 2046
  + application/rss+xml: RSS feeds
  + application/soap+xml: SOAP; Defined by RFC 3902
  + application/font-woff: Web Open Font Format; (candidate recommendation; use application/x-font-woff until standard is official)
  + application/xhtml+xml: XHTML; Defined by RFC 3236
  + application/xml-dtd: DTD files; Defined by RFC 3023
  + application/xop+xml:XOP
  + application/zip: ZIP archive files; Registered[7]
  + application/x-gzip: Gzip
* Type audio：For Audio.
  + audio/basic: mulaw audio at 8 kHz, 1 channel; Defined in RFC 2046
  + audio/L24: 24bit Linear PCM audio at 8-48kHz, 1-N channels; Defined in RFC 3190
  + audio/mp4: MP4 audio
  + audio/mpeg: MP3 or other MPEG audio; Defined in RFC 3003
  + audio/ogg: Ogg Vorbis, Speex, Flac and other audio; Defined in RFC 5334
  + audio/vorbis: Vorbis encoded audio; Defined in RFC 5215
  + audio/x-ms-wma: Windows Media Audio; Documented in Microsoft KB 288102
  + audio/x-ms-wax: Windows Media Audio Redirector; Documented in Microsoft help page
  + audio/vnd.rn-realaudio: RealAudio; Documented in RealPlayer Customer Support Answer 2559
  + audio/vnd.wave: WAV audio; Defined in RFC 2361
  + audio/webm: WebM open media format
* Type image
  + image/gif: GIF image; Defined in RFC 2045 and RFC 2046
  + image/jpeg: JPEG JFIF image; Defined in RFC 2045 and RFC 2046
  + image/pjpeg: JPEG JFIF image; Associated with Internet Explorer; Listed in ms775147(v=vs.85) - Progressive JPEG, initiated before global browser support for progressive JPEGs (Microsoft and Firefox).
  + image/png: Portable Network Graphics; Registered,[8] Defined in RFC 2083
  + image/svg+xml: SVG vector image; Defined in SVG Tiny 1.2 Specification Appendix M
  + image/tiff: Tag Image File Format (only for Baseline TIFF); Defined in RFC 3302
  + image/vnd.microsoft.icon: ICO image; Registered[9]
* Type message
  + message/http: Defined in RFC 2616
  + message/imdn+xml: IMDN Instant Message Disposition Notification; Defined in RFC 5438
  + message/partial: Email; Defined in RFC 2045 and RFC 2046
  + message/rfc822: Email; EML files, MIME files, MHT files, MHTML files; Defined in RFC 2045 and RFC 2046
* Type model For 3D models.
  + model/example: Defined in RFC 4735
  + model/iges: IGS files, IGES files; Defined in RFC 2077
  + model/mesh: MSH files, MESH files; Defined in RFC 2077, SILO files
  + model/vrml: WRL files, VRML files; Defined in RFC 2077
  + model/x3d+binary: X3D ISO standard for representing 3D computer graphics, X3DB binary files
  + model/x3d+vrml: X3D ISO standard for representing 3D computer graphics, X3DV VRML files
  + model/x3d+xml: X3D ISO standard for representing 3D computer graphics, X3D XML files
* Type multipart:For archives and other objects made of more than one part.
  + multipart/mixed: MIME Email; Defined in RFC 2045 and RFC 2046
  + multipart/alternative: MIME Email; Defined in RFC 2045 and RFC 2046
  + multipart/related: MIME Email; Defined in RFC 2387 and used by MHTML (HTML mail)
  + multipart/form-data: MIME Webform; Defined in RFC 2388
  + multipart/signed: Defined in RFC 1847
  + multipart/encrypted: Defined in RFC 1847
* Type text ：For human-readable text and source code.
  + text/cmd: commands; subtype resident in Gecko browsers like Firefox 3.5
  + text/css: Cascading Style Sheets; Defined in RFC 2318
  + text/csv: Comma-separated values; Defined in RFC 4180
  + text/html: HTML; Defined in RFC 2854
  + text/javascript (Obsolete): JavaScript; Defined in and obsoleted by RFC 4329 in order to discourage its usage in favor of application/javascript. However, text/javascript is allowed in HTML 4 and 5 and, unlike application/javascript, has cross-browser support. The "type" attribute of the <script> tag in HTML5 is optional and there is no need to use it at all since all browsers have always assumed the correct default (even in HTML 4 where it was required by the specification).
  + text/plain: Textual data; Defined in RFC 2046 and RFC 3676
  + text/vcard: vCard (contact information); Defined in RFC 6350
  + text/xml: Extensible Markup Language; Defined in RFC 3023
* Type video：For video.
  + video/mpeg: MPEG-1 video with multiplexed audio; Defined in RFC 2045 and RFC 2046
  + video/mp4: MP4 video; Defined in RFC 4337
  + video/ogg: Ogg Theora or other video (with audio); Defined in RFC 5334
  + video/quicktime: QuickTime video; Registered[10]
  + video/webm: WebM open media format
  + video/x-ms-wmv: Windows Media Video; Documented in Microsoft KB 288102
* Type vnd ：For vendor-specific files.
  + application/vnd.oasis.opendocument.text: OpenDocument Text; Registered[11]
  + application/vnd.oasis.opendocument.spreadsheet: OpenDocument Spreadsheet; Registered[12]
  + application/vnd.oasis.opendocument.presentation: OpenDocument Presentation; Registered[13]
  + application/vnd.oasis.opendocument.graphics: OpenDocument Graphics; Registered[14]
  + application/vnd.ms-excel: Microsoft Excel files
  + application/vnd.openxmlformats-officedocument.spreadsheetml.sheet: Microsoft Excel 2007 files
  + application/vnd.ms-powerpoint: Microsoft Powerpoint files
  + application/vnd.openxmlformats-officedocument.presentationml.presentation: Microsoft Powerpoint 2007 files
  + application/msword: Microsoft Word files
  + application/vnd.openxmlformats-officedocument.wordprocessingml.document: Microsoft Word 2007 files
  + application/vnd.mozilla.xul+xml: Mozilla XUL files
  + application/vnd.google-earth.kml+xml: KML files (e.g. for Google Earth)
* Type x ：For non-standard files.
  + application/x-www-form-urlencoded Form Encoded Data; Documented in HTML 4.01 Specification, Section 17.13.4.1
  + application/x-dvi: device-independent document in DVI format
  + application/x-latex: LaTeX files
  + application/x-font-ttf: TrueType Font No registered MIME type, but this is the most commonly used
  + application/x-shockwave-flash: Adobe Flash files for example with the extension .swf
  + application/x-stuffit: StuffIt archive files
  + application/x-rar-compressed: RAR archive files
  + application/x-tar: Tarball files
  + text/x-jquery-tmpl: jQuery template data
  + application/x-javascript:
  + application/x-deb: deb\_(file\_format), a software package format used by the Debian project
* Type x-pkcs:For PKCS standard files.
  + application/x-pkcs12: p12 files
  + application/x-pkcs12: pfx files
  + application/x-pkcs7-certificates: p7b files
  + application/x-pkcs7-certificates: spc files
  + application/x-pkcs7-certreqresp: p7r files
  + application/x-pkcs7-mime: p7c files
  + application/x-pkcs7-mime: p7m files
  + application/x-pkcs7-signature: p7s files

1. aa

iBatis

1. SqlMapConfig
   1. <setting>元素

|  |  |
| --- | --- |
| maxRequests | 同时执行 SQL 语句的最大线程数。大于这个值的线程将阻塞直到另一个线程执行完成。不同的 DBMS  有不同的限制值，但任何数据库都有这些限制。通  常这个值应该至少是 maxTransactions（参见以下）  的 10 倍，并且总 是大于 maxSessions 和  maxTranactions。减小这个参数值通常能提高性能。    例如：maxRequests=“256”  缺省值：512 |
| maxSessions | 同一时间内活动的最大 session 数。 一个 session 可以  是代码请求的显式 session，也可以是当线程使用  SqlMapClient 实例（即执行一条语句）自动获得的  session。它应该总是大于或等于 maxTransactions 并  小于 maxRequests。减小这个参数值通常能减少内存  使用。    例如：maxSessions=“64”  缺省值：128 |
| maxTransactions | 同时进入 SqlMapClient.startTransaction()的最大线程  数。大于这个值的线程将阻塞直到另一个线程退出。  不同的 DBMS 有不同的限制值，但任何数据库都有  这些限制。这个参数值应该总是小于或等于  maxSessions 并总是远远小于 maxRequests。减小这  个参数值通常能提高性能。    例如：maxTransactions=“16”  缺省值：32 |
| cacheModelsEnabled | 全局性地启用或禁用 SqlMapClient 的所有缓存  model。调试程序时使用。    例如：cacheModelsEnabled=“true”  缺省值：true（启用） |
| lazyLoadingEnabled | 全局性地启用或禁用SqlMapClient的所有延迟加载。  调试程序时使用。    例子：lazyLoadingEnabled=“true”  缺省值：true（启用） |
| enhancementEnabled | 全局性地启用或禁用运行时字节码增强，以优化访  问Java Bean属性的性能， 同时优化延迟加载的性能。    例子：enhancementEnabled=“true”  缺省值：false（禁用） |
| useStatementNamespaces | 如果启用本属性，必须使用全限定名来引用 mapped  statement。Mapped statement 的全限定名由 sql-map  的名称和 mapped-statement 的名称合成。例如：queryForObject(“sqlMapName.statementName”);    例如：useStatementNamespaces=“false”  缺省值：false（禁用） |
|  |  |
|  |  |

* 1. <typeAlias>元素

<typeAlias>元素让您为一个通常较长的、全限定类名指定一个较短的别名。

在 SQL Map配置文件预定义了几个别名

|  |  |
| --- | --- |
| **事务管理器别名** | |
| JDBC | com.ibatis.sqlmap.engine.transaction.jdbc.JdbcTransactionConfig |
| JTA | com.ibatis.sqlmap.engine.transaction.jta.JtaTransactionConfig |
| EXTERNAL | com.ibatis.sqlmap.engine.transaction.external.ExternalTransactionConfig |
| **Data Source Factory 别名** | |
| SIMPLE | com.ibatis.sqlmap.engine.datasource.SimpleDataSourceFactory |
| DBCP | com.ibatis.sqlmap.engine.datasource.DbcpDataSourceFactory |
| JNDI | com.ibatis.sqlmap.engine.datasource.JndiDataSourceFactory |

框架的三个事务管理器分别是：JDBC，JTA和 EXTERNAL

* JDBC：通过常用的 Connection commit()和 rollback()方法，让 JDBC 管理事务。
* JTA：本事务管理器使用一个 JTA 全局事务，使 SQL Map 的事务包括在更大的事务范围内，这个更大的事务范围可能包括了其他的数据库和事务资源。这个配置需要一个UserTransaction属性， 以便从JNDI获得一个UserTransaction。 参见以下JNDI数据源的例子。
* EXTERNAL：这个配置可以让您自己管理事务。您仍然可以配置一个数据源，但事务不再作为框架生命周期的一部分被提交或回退。这意味着 SQL Map 外部应用的一部分必须自己管理事务。这个配置也可以用于没有事务管理的数据库（例如只读数据库）

1. SQL Map XML映射文件

建议：

* **输入值选择parameterClass通过typeAlias标签进行实体类别名映射**
* **返回值选择resultMap通过resultMap标签进行数据库字段映射**

|  |
| --- |
| <typeAlias alias="user" type="com.alibaba.sample.petstore.dal.dataobject.User"/>  <resultMap id="userResult" class="user">  <result property="userId" column="USER\_ID"/>  <result property="role" column="ROLES"/>  </resultMap>  <select id="getUserByUserIdAndPassword" resultMap="userResult" parameterClass="user"><![CDATA[  select  USER.\*,  ACCOUNT.\*,  PROFILE.\*  from  USER  left join ACCOUNT on USER.USER\_ID = ACCOUNT.USER\_ID  left join PROFILE on USER.USER\_ID = PROFILE.USER\_ID  where  USER.USER\_ID = #userId#  and USER.PASSWORD = #password#  ]]></select> |

* 1. SQL语句

|  |
| --- |
| <![CDATA[ ]]> |

* 1. 自动生成的主键

|  |
| --- |
| <!-- Oracle SEQUENCE Example -->  <insert id="insertProduct-ORACLE" parameterClass="com.domain.Product">  <selectKey resultClass="int" keyProperty="id">  SELECT STOCKIDSEQUENCE.NEXTVAL AS ID FROM DUAL  </selectKey>  insert into PRODUCT (PRD\_ID,PRD\_DESCRIPTION) values  (#id#,#description#)  </insert>  <!--Microsoft SQL Server IDENTITY Column Example -->  <insert id="insertProduct-MS-SQL" parameterClass="com.domain.Product">  insert into PRODUCT (PRD\_DESCRIPTION) values (#description#)  <selectKey resultClass="int" keyProperty="id">  SELECT @@IDENTITY AS ID  </selectKey>  </insert> |

* 1. 存储过程

|  |
| --- |
| <parameterMap id="swapParameters" class="map">  <parameter property="email1" jdbcType="VARCHAR"  javaType="java.lang.String" mode="INOUT" />  <parameter property="email2" jdbcType="VARCHAR"  javaType="java.lang.String" mode="INOUT" />  </parameterMap>  <procedure id="swapEmailAddresses" parameterMap="swapParameters">  {call swap\_email\_address (?, ?)}  </procedure> |

* 1. parameterClass

|  |
| --- |
| <statement id="statementName" parameterClass="examples.domain.Product">  insert into PRODUCT values (#id#, #description#, #price#)  </statement> |

* 1. parameterMap

|  |
| --- |
| <parameterMap id="insert-product-param" class="com.domain.Product">  <parameter property="id" />  <parameter property="description" />  </parameterMap>  <statement id="insertProduct" parameterMap="insert-product-param">  insert into PRODUCT (PRD\_ID, PRD\_DESCRIPTION) values (?,?);  </statement> |
| <parameterMap id="insert-product-param"  class="com.domain.Product">  <parameter property="id" jdbcType="NUMERIC" javaType="int"  nullValue="-9999999" />  <parameter property="description" jdbcType="VARCHAR"  nullValue="NO\_ENTRY" />  </parameterMap>  <statement id="insertProduct" parameterMap="insert-product-param">  insert into PRODUCT (PRD\_ID, PRD\_DESCRIPTION) values (?,?);  </statement> |

* 1. Inline Parameter

|  |
| --- |
| <statement id="insertProduct" >  insert into PRODUCT (PRD\_ID, PRD\_DESCRIPTION)  values (#id#, #description#);  </statement> |
| <statement id="insertProduct" parameterClass="com.domain.Product">  insert into PRODUCT (PRD\_ID, PRD\_DESCRIPTION) values (#id#,  #description#);  </statement>  <statement id="insertProduct" parameterClass="com.domain.Product">  insert into PRODUCT (PRD\_ID, PRD\_DESCRIPTION) values  (#id:NUMERIC#, #description:VARCHAR#);  </statement>  <statement id="insertProduct" parameterClass="com.domain.Product">  insert into PRODUCT (PRD\_ID, PRD\_DESCRIPTION) values  (#id:NUMERIC:-999999#, #description:VARCHAR:NO\_ENTRY#);  </statement> |

* 1. resultClass

|  |
| --- |
| <statement id="getPerson" parameterClass="int"  resultClass="examples.domain.Person">  SELECT PER\_ID as id, PER\_FIRST\_NAME as firstName, PER\_LAST\_NAME  as lastName, PER\_BIRTH\_DATE as birthDate, PER\_WEIGHT\_KG as  weightInKilograms, PER\_HEIGHT\_M as heightInMeters FROM PERSON  WHERE PER\_ID = #value#  </statement> |

* 1. resultMap

|  |
| --- |
| <resultMap id="get-product-result"  class="com.ibatis.example.Product">  <result property="id" column="PRD\_ID" />  <result property="description" column="PRD\_DESCRIPTION" />  </resultMap>  <statement id="getProduct" resultMap="get-product-result">  select \* from PRODUCT  </statement> |

* 1. 基本类型输入参数

|  |
| --- |
| <statement id="insertProduct" parameter="java.lang.Integer">  select \* from PRODUCT where PRD\_ID = #value#  </statement>  <statement id="insertProduct" parameterClass="java.util.Map">  select \* from PRODUCT where PRD\_CAT\_ID = #catId# and PRD\_CODE =  #code#  </statement> |

* 1. Result Map

|  |
| --- |
| <resultMap id="get-product-result"  class="com.ibatis.example.Product">  <result property="id" column="PRD\_ID" />  <result property="description" column="PRD\_DESCRIPTION" />  <result property="subCode" column="PRD\_SUB\_CODE"  nullValue="-999" />  </resultMap> |

* 1. 隐式的Result Map

|  |
| --- |
| <statement id="getProduct"  resultClass="com.ibatis.example.Product">  select PRD\_ID as id, PRD\_DESCRIPTION as description from PRODUCT  where PRD\_ID = #value#  </statement> |

* 1. 基本类型的Result（即String，Integer，Boolean）

|  |
| --- |
| <resultMap id="get-product-result" class="java.lang.String">  <result property="value" column="PRD\_DESCRIPTION" />  </resultMap>  <statement id="getProductCount" resultClass="java.lang.Integer">  select count(1) as value from PRODUCT  </statement> |

* 1. Map类型的Result

|  |
| --- |
| <resultMap id="get-product-result" class="java.util.HashMap">  <result property="id" column="PRD\_ID" />  <result property="code" column="PRD\_CODE" />  <result property="description" column="PRD\_DESCRIPTION" />  <result property="suggestedPrice" column="PRD\_SUGGESTED\_PRICE" />  </resultMap>  <statement id="getProductCount" resultClass="java.util.HashMap">  select \* from PRODUCT  </statement> |

* 1. 复杂类型属性（即自定义类型的属性）

|  |
| --- |
| <resultMap id="get-product-result"  class="com.ibatis.example.Product">  <result property="id" column="PRD\_ID" />  <result property="description" column="PRD\_DESCRIPTION" />  **<result property="category" column="PRD\_CAT\_ID" select="getCategory" />**  </resultMap>  <resultMap id="get-category-result"  class="com.ibatis.example.Category">  <result property="id" column="CAT\_ID" />  <result property="description" column="CAT\_DESCRIPTION" />  </resultMap>  <statement id="getProduct" parameterClass="int"  resultMap="get-product-result">  select \* from PRODUCT where PRD\_ID = #value#  </statement>  <statement id="**getCategory**" parameterClass="int"  resultMap="get-category-result">  select \* from CATEGORY where CAT\_ID = #value#  </statement> |

* 1. 避免 N＋1 Select（1:1）

|  |
| --- |
| <resultMap id="get-product-result"  class="com.ibatis.example.Product">  <result property="id" column="PRD\_ID" />  <result property="description" column="PRD\_DESCRIPTION" />  **<result property="category.id" column="CAT\_ID" />**  **<result property="category.description" column="CAT\_DESCRIPTION" />**  </resultMap>  <statement id="getProduct" parameterClass="int"  resultMap="get-product-result">  select \* from PRODUCT, CATEGORY where PRD\_CAT\_ID=CAT\_ID and  PRD\_ID = #value#  </statement> |

* 1. 复杂类型集合的属性

|  |
| --- |
| <resultMap id="get-category-result"  class="com.ibatis.example.Category">  <result property="id" column="CAT\_ID" />  <result property="description" column="CAT\_DESCRIPTION" />  **<result property="productList" column="CAT\_ID" select=" getProductsByCatId" />**  </resultMap>  <resultMap id="get-product-result"  class="com.ibatis.example.Product">  <result property="id" column="PRD\_ID" />  <result property="description" column="PRD\_DESCRIPTION" />  </resultMap>  <statement id="getCategory" parameterClass="int"  resultMap="get-category-result">  select \* from CATEGORY where CAT\_ID = #value#  </statement>  <statement id="getProductsByCatId" parameterClass="int"  resultMap="get-product-result">  select \* from PRODUCT where PRD\_CAT\_ID = #value#  </statement> |

* 1. 组合键值或多个复杂参数属性

|  |
| --- |
| <resultMap id="get-order-result" class="com.ibatis.example.Order">  <result property="id" column="ORD\_ID" />  <result property="customerId" column="ORD\_CST\_ID" />  <result property="payments"  column="**{itemId=ORD\_ID, custId=ORD\_CST\_ID}**"  select=" getOrderPayments" />  </resultMap>  <statement id="getOrderPayments" resultMap="get-payment-result">  select \* from PAYMENT where PAY\_ORD\_ID = #**itemId**# and PAY\_CST\_ID  = #**custId**#  </statement> |

* 1. jdbcType

Array, BigInt, Binary, Bit, Blob, Boolean, Char, Clob, Datalink, Date, Decimal, Double, Float, Integer, LongVarBinary, LongVarChar, Numeric, Real, Ref, SmallInt, Struct, Time, Timestamp, TinyInt, VarBinary, VarChar.

1. 动态 Mapped Statement

|  |
| --- |
| <statement id="dynamicGetAccountList" resultMap="account-result">  select \* from ACCOUNT  <dynamic prepend="WHERE">  <isNotNull prepend="AND" property="firstName">  (ACC\_FIRST\_NAME = #firstName#  <isNotNull prepend="OR" property="lastName">  ACC\_LAST\_NAME = #lastName#  </isNotNull>  )  </isNotNull>  <isNotNull prepend="AND" property="emailAddress">  ACC\_EMAIL like #emailAddress#  </isNotNull>  <isGreaterThan prepend="AND" property="id"  compareValue="0">  ACC\_ID = #id#  </isGreaterThan>  </dynamic>  order by ACC\_LAST\_NAME  </statement> |

1. 日志

|  |
| --- |
| #log4j.logger.com.ibatis=DEBUG  #log4j.logger.com.ibatis.common.jdbc.SimpleDataSource=DEBUG  #log4j.logger.com.ibatis.common.jdbc.ScriptRunner=DEBUG  #log4j.logger.com.ibatis.sqlmap.engine.impl.SqlMapClientDelegate=DEBUG  #log4j.logger.java.sql.Connection=DEBUG  #log4j.logger.java.sql.Statement=DEBUG  #log4j.logger.java.sql.PreparedStatement=DEBUG  #log4j.logger.java.sql.ResultSet=DEBUG |

1. 常用语句
   1. 输入参数为单个值

|  |
| --- |
| <delete  id="com.fashionfree.stat.accesslog.deleteMemberAccessLogsBefore"  parameterClass="long">  delete from MemberAccessLog where accessTimestamp = #value#  </delete> |

* 1. 输入参数为一个对象

|  |
| --- |
| <insert id="com.fashionfree.stat.accesslog.MemberAccessLog.insert"  parameterClass="com.fashionfree.stat.accesslog.model.MemberAccessLog>  insert into MemberAccessLog ( accessLogId, memberId, clientIP,  httpMethod, actionId, requestURL, accessTimestamp, extend1,  extend2, extend3 ) values ( #accessLogId#, #memberId#,  #clientIP#, #httpMethod#, #actionId#, #requestURL#,  #accessTimestamp#, #extend1#, #extend2#, #extend3# )  </insert> |

* 1. 输入参数为一个java.util.HashMap

|  |
| --- |
| <select  id="com.fashionfree.stat.accesslog.selectActionIdAndActionNumber"  parameterClass="hashMap" resultMap="getActionIdAndActionNumber">  select actionId, count(\*) as count from MemberAccessLog where  memberId = #memberId# and accessTimestamp &gt; #start# and  accessTimestamp &lt;= #end# group by actionId  </select> |

* 1. 输入参数中含有数组

|  |
| --- |
| <insert id="updateStatusBatch" parameterClass="hashMap">  update Question set status = #status#  <dynamic prepend="where questionId in">  <isNotNull property="actionIds">  <iterate property="actionIds" open="(" close=")"  conjunction=",">  #actionIds[]#  </iterate>  </isNotNull>  </dynamic>  </insert>  说明：actionIds为传入的数组的名字； 使用dynamic标签避免数组为空时导致sql语句语法出错；使用isNotNull标签避免数组为null时ibatis解析出错 |

* 1. 传递参数只含有一个数组

|  |
| --- |
| <select  id="com.fashionfree.stat.accesslog.StatMemberAction.selectActionIdsOfModule"  resultClass="hashMap">  select moduleId, actionId from StatMemberAction  <dynamic prepend="where moduleId in">  <iterate open="(" close=")" conjunction=",">#[]#</iterate>  </dynamic>  order by moduleId  </select>  说明：注意select的标签中没有parameterClass一项 另：这里也可以把数组放进一个hashMap中，但增加额外开销，不建议使用 |

* 1. 让ibatis把参数直接解析成字符串

|  |
| --- |
| <select  id="com.fashionfree.stat.accesslog.selectSumDistinctCountOfAccessMemberNum"  parameterClass="hashMap" resultClass="int">  select count(distinct memberId) from MemberAccessLog where  accessTimestamp &gt;= #start# and accessTimestamp &lt; #end# and  actionId in $actionIdString$  </select> |

* 1. 分页查询 (pagedQuery)

|  |
| --- |
| <select id="com.fashionfree.stat.accesslog.selectMemberAccessLogBy"  parameterClass="hashMap" resultMap="MemberAccessLogMap">  <include refid="selectAllSql" />  <include refid="whereSql" />  <include refid="pageSql" />  </select>  <select  id="com.fashionfree.stat.accesslog.selectMemberAccessLogBy.Count"  parameterClass="hashMap" resultClass="int">  <include refid="countSql" />  <include refid="whereSql" />  </select>  <sql id="selectAllSql">  select accessLogId, memberId, clientIP, httpMethod, actionId,  requestURL, accessTimestamp, extend1, extend2, extend3 from  MemberAccessLog  </sql>  <sql id="whereSql">accessTimestamp &lt;= #accessTimestamp#</sql>  <sql id="countSql">select count(\*) from MemberAccessLog</sql>  <sql id="pageSql">  <dynamic>  <isNotNull property="startIndex">  <isNotNull property="pageSize">  limit #startIndex# , #pageSize#  </isNotNull>  </isNotNull>  </dynamic>  </sql> |

* 1. sql语句中含有大于号>、小于号<

|  |
| --- |
| <delete  id="com.fashionfree.stat.accesslog.deleteMemberAccessLogsBefore"  parameterClass="long">  delete from MemberAccessLog where accessTimestamp &lt;= #value#  </delete>  <delete  id="com.fashionfree.stat.accesslog.deleteMemberAccessLogsBefore"  parameterClass="long">  <![CDATA[  delete from  MemberAccessLog  where  accessTimestamp <= #value#  ]]>  </delete>  推荐使用第一种方式，写为&lt; 和 &gt; (XML不对CDATA里的内容进行解析，因此如果CDATA中含有dynamic标签，将不起作用) |

* 1. include和sql标签

|  |
| --- |
| <sql id="selectBasicSql">  select samplingTimestamp,onlineNum,year, month,week,day,hour  from OnlineMemberNum  </sql>  <sql id="whereSqlBefore">  where samplingTimestamp &lt;= #samplingTimestamp#  </sql>  <select  id="com.fashionfree.accesslog.selectOnlineMemberNumsBeforeSamplingTimestamp"  parameterClass="hashmap" resultClass="OnlineMemberNum">  <include refid="selectBasicSql" />  <include refid="whereSqlBefore" />  </select> |

* 1. 随机选取记录

|  |
| --- |
| <sql id="randomSql">  ORDER BY rand() LIMIT #number#  </sql>  从数据库中随机选取number条记录（只适用于MySQL） |

* 1. 将SQL GROUP BY分组中的字段拼接

|  |
| --- |
| <sql id=”selectGroupBy>  SELECT a.answererCategoryId, a.answererId, a.answererName,  a.questionCategoryId, a.score, a.answeredNum, a.correctNum,  a.answerSeconds, a.createdTimestamp,  a.lastQuestionApprovedTimestamp, a.lastModified,  GROUP\_CONCAT(q.categoryName) as categoryName FROM  AnswererCategory a, QuestionCategory q WHERE  a.questionCategoryId = q.questionCategoryId GROUP BY  a.answererId ORDER BY a.answererCategoryId  </sql>  注：SQL中使用了MySQL的GROUP\_CONCAT函数 |

* 1. 按照IN里面的顺序进行排序

|  |
| --- |
| Mysql  <sql id=”groupByInArea”>  select  moduleId, moduleName,  status, lastModifierId, lastModifiedName,  lastModified  from  StatModule  where  moduleId in (3, 5, 1)  order by  instr(',3,5,1,' , ','+ltrim(moduleId)+',')  </sql>  SQLServer  <sql id=”groupByInArea”>  select moduleId, moduleName, status, lastModifierId,  lastModifiedName, lastModified from StatModule where moduleId in  (3, 5, 1) order by charindex(','+ltrim(moduleId)+',' ,  ',3,5,1,')  </sql> |

* 1. resultMap

|  |
| --- |
| <resultMap class="java.util.HashMap"  id="getActionIdAndActionNumber">  <result column="actionId" property="actionId" jdbcType="BIGINT"  javaType="long" />  <result column="count" property="count" jdbcType="INT"  javaType="int" />  </resultMap> |

* 1. remap

|  |
| --- |
| <select id="testForRemap" parameterClass="hashMap"  resultClass="hashMap" remapResults="true">  select userId  <isEqual property="tag" compareValue="1">, userName</isEqual>  <isEqual property="tag" compareValue="2">  , userPassword  </isEqual>  from UserInfo  </select>  此例中，根据参数tag值的不同，会获得不同的结果集，如果没有remapResults="true"属性，iBatis会将第一次查询时的结果集缓存，下次再执行时（必须还是该进程中）不会再执行结果集映射，而是会使用缓存的结果集。 |

* 1. dynamic标签的prepend

|  |
| --- |
| <sql id="whereSql">  <dynamic prepend="where ">  <isNotNull property="userId" prepend="BOGUS">  userId = #userId#  </isNotNull>  <isNotEmpty property="userName" prepend="and ">  userName = #userName#  </isNotEmpty>  </dynamic>  </sql> |

* 1. 返回值为单个值

|  |
| --- |
| <select id="getMaxSaleDate" resultClass="java.util.Date">  select max(saledate) as saledate from sale  </select> |

* 1. 日期范围查询

|  |
| --- |
| <parameterMap class="java.util.HashMap" id="saleDateRangeMap">  <parameter property="beginDate"/>  <parameter property="endDate"/>  </parameterMap>  <select id="getCityTopProfitsChart" parameterMap="saleDateRangeMap" resultClass="java.util.HashMap">  select v.address,sum(profits) as profits  from (  select s.id,s.saleDate,s.userid,u.address,(s.price-p.price)\*s.number as profits  from sale s left join product p on s.productid=p.id  left join user u on s.userid = u.id  where s.saleDate &gt; ? and s.saleDate &lt; ?  ) v  group by v.address order by sum(profits) desc limit 0,10  </select> |

* 1. Todo

1. 注意事项
   1. 处理int类型为空的情况

当数据库中int类型为空时，解决：1、设置该字符不能为空，或者设置默认值；2、设置result的nullvalue的属性

* 1. 当添加字段时内部无法转换错误

数据表字段顺序与xml文件字段顺序不一致有空能出错，提示，内部无法转换，需要将表字段顺序与xml文件字段相一致

* 1. CLOB与BLOB字段顺序问题

建议将Clob与Blob字段映射在表最后一个（在多线程操作下，会出现clob后面字段字段值插入错误，比如Clob后面字段A的值插入到字段B）

* 1. ibatis模糊查询的实现

ibatis有两种方式，一种是＃，一种是$。用preparestatment来实现的时候是这样的。凡是#的，都作为参数，用setobject方式。而$方式的，则直接替换字符串。 所以。 select col1,col2 from table1 where col1=#col1# and col2 like ’$col2$’ 假如参数是col1=2 col2=2 最终的SQL就是 select col1,col2 from table1 where col1=? and col2 like ’2’ 如果需要通符，则用 like ’%$col2$%’

* 1. TODO

JVM虚拟机

1. JVM配置和优化

JVM内存的最大值跟操作系统有很大的关系。简单的说就32位处理器虽然 可控内存空间有4GB,但是具体的操作系统会给一个限制，这个限制一般是2GB-3GB（一般来说Windows系统下为1.5G-2G，Linux系统 下为2G-3G），而64bit以上的处理器就不会有限制了。

* 1. 参数说明（按官方）

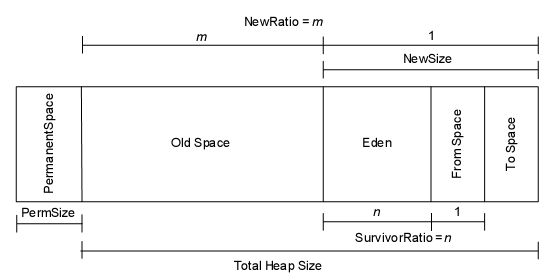
非稳态选项使用说明

-XX:+<option> 启用选项

-XX:-<option> 不启用选项

-XX:<option>=<number> 给选项设置一个数字类型值，可跟单位，例如 32k, 1024m, 2g

-XX:<option>=<string> 给选项设置一个字符串值，例如-XX:HeapDumpPath=./dump.core



* + 1. 行为参数
* -XX:-AllowUserSignalHandlers
  + 限于Linux和Solaris，默认不启用
  + 允许为java进程安装信号处理器。Java信号处理相关知识，详见 http://kenwublog.com/java-asynchronous-notify-based-on-signal
* -XX:AltStackSize=16384
* -XX:-DisableExplicitGC
  + 默认不启用
  + 禁止在运行期显式地调用 System.gc()。开启该选项后，GC的触发时机将由Garbage Collector全权掌控。注意：你熟悉的代码里没调用System.gc()，不代表你依赖的框架工具没在使用。例如RMI就在多数用户毫不知情的情况下，显示地调用GC来防止自身OOM。请仔细权衡禁用带来的影响。
* -XX:+FailOverToOldVerifier
  + Java6新引入选项，默认启用
  + 如果新的Class校验器检查失败，则使用老的校验器。为什么会失败？因为JDK6最高向下兼容到JDK1.2，而JDK1.2的class info 与JDK6的info存在较大的差异，所以新校验器可能会出现校验失败的情况。关联选项：-XX:+UseSplitVerifier
* -XX:+HandlePromotionFailure
  + java5以前是默认不启用，java6默认启用
  + 关闭新生代收集担保。

什么是新生代收集担保？

在一次理想化的minor gc中，Eden和First Survivor中的活跃对象会被复制到Second Survivor。然而，Second Survivor不一定能容纳下所有从E和F区copy过来的活跃对象。为了确保minor gc能够顺利完成，GC需要在年老代中额外保留一块足以容纳所有活跃对象的内存空间。这个预留操作，就被称之为新生代收集担保（New Generation Guarantee）。如果预留操作无法完成时，仍会触发major gc(full gc)。

为什么要关闭新生代收集担保？

因为在年老代中预留的空间大小，是无法精确计算的。为了确保极端情况的发生，GC参考了最坏情况下的新生代内存占用，即Eden+First Survivor。这种策略无疑是在浪费年老代内存，从时序角度看，还会提前触发Full GC。为了避免如上情况的发生，JVM允许开发者手动关闭新生代收集担保。

在开启本选项后，minor gc将不再提供新生代收集担保，而是在出现survior或年老代不够用时，抛出promotion failed异常。

* -XX:+MaxFDLimit
  + 限于Solaris，默认启用
  + 设置java进程可用文件描述符为操作系统允许的最大值。
* -XX:PreBlockSpin=10
  + -XX:+UseSpinning 必须先启用，对于java6来说已经默认启用了，这里默认自旋10次
  + 控制多线程自旋锁优化的自旋次数。(什么是自旋锁优化？见 -XX:+UseSpinning 处的描述)。关联选项：-XX:+UseSpinning
* -XX:-RelaxAccessControlCheck
  + 默认不启用
  + 在Class校验器中，放松对访问控制的检查。作用与reflection里的setAccessible类似。
* -XX:+ScavengeBeforeFullGC
  + 默认启用
  + 在Full GC前触发一次Minor GC。
* -XX:+UseAltSigs
  + 限于Solaris，默认启用
  + 为了防止与其他发送信号的应用程序冲突，允许使用候补信号替代 SIGUSR1和SIGUSR2。
* -XX:+UseBoundThreads
  + 限于Solaris，默认启用
  + 绑定所有的用户线程到内核线程。减少线程进入饥饿状态（得不到任何cpu time）的次数。
* -XX:-UseConcMarkSweepGC
  + 默认不启用
  + 启用CMS低停顿垃圾收集器。资料详见：http://kenwublog.com/docs/CMS\_GC.pdf
* -XX:+UseGCOverheadLimit
  + 默认启用
  + 限制GC的运行时间。如果GC耗时过长，就抛OOM。
* -XX:+UseLWPSynchronization
  + 限于Solaris，默认启用
  + 使用轻量级进程（内核线程）替换线程同步。
* -XX:-UseParallelGC
  + -server时启用,其他情况下，默认不启用
  + 策略为新生代使用并行清除，年老代使用单线程Mark-Sweep-Compact的垃圾收集器。
* -XX:-UseParallelOldGC
  + 默认不启用
  + 策略为老年代和新生代都使用并行清除的垃圾收集器。
* -XX:-UseSerialGC
  + -client时启用,其他情况下，默认不启用
  + 使用串行垃圾收集器。
* -XX:-UseSpinning
  + java1.4.2和1.5需要手动启用, java6默认已启用
  + 启用多线程自旋锁优化。

自旋锁优化原理

大家知道，Java的多线程安全是基于Lock机制实现的，而Lock的性能往往不如人意。原因是，monitorenter与monitorexit这两个控制多线程同步的bytecode原语，是JVM依赖操作系统互斥(mutex)来实现的。互斥是一种会导致线程挂起，并在较短的时间内又必须重新调度回原线程的，较为消耗资源的操作。为了避免进入OS互斥，Java6的开发者们提出了自旋锁优化。自旋锁优化的原理是在线程进入OS互斥前，通过CAS自旋一定的次数来检测锁的释放。如果在自旋次数未达到预设值前锁已被释放，则当前线程会立即持有该锁。

CAS检测锁的原理详见: http://kenwublog.com/theory-of-lightweight-locking-upon-cas

关联选项：-XX:PreBlockSpin=10

* -XX:+UseTLAB
  + 1.4.2以前和使用-client选项时，默认不启用，其余版本默认启用
  + 启用线程本地缓存区（Thread Local）。
* -XX:+UseSplitVerifier
  + java5默认不启用,java6默认启用
  + 新Class类型校验器有什么特点

新Class类型校验器，将老的校验步骤拆分成了两步：1，类型推断。2，类型校验。

新类型校验器通过在javac编译时嵌入类型信息到bytecode中，省略了类型推断这一步，从而提升了classloader的性能。

Classload顺序（供参考）load -> verify -> prepare -> resove -> init

关联选项：-XX:+FailOverToOldVerifier

* -XX:+UseThreadPriorities
  + 默认启用
  + 使用本地线程的优先级。
* -XX:+UseVMInterruptibleIO
  + 限于solaris，默认启用
  + 在solaris中，允许运行时中断线程 。
    1. 调优参数
       - -XX:+AggressiveOpts
         * JDK 5 update 6后引入，但需要手动启用。JDK6默认启用。
         * 启用JVM开发团队最新的调优成果。例如编译优化，偏向锁，并行年老代收集等。
       - -XX:CompileThreshold=10000
         * 1000
         * 通过JIT编译器，将方法编译成机器码的触发阀值，可以理解为调用方法的次数，例如调1000次，将方法编译为机器码。
       - -XX:LargePageSizeInBytes=4m
         * 默认4m，amd64位：2m
         * 设置堆内存的内存页大小。调整内存页的方法和性能提升原理，详见 http://kenwublog.com/tune-large-page-for-jvm-optimization
       - -XX:MaxHeapFreeRatio=70
         * 70
         * GC后，如果发现空闲堆内存占到整个预估上限值的70%，则收缩预估上限值。

什么是预估上限值？

JVM在启动时，会申请最大值（-Xmx指定的数值）的地址空间，但其中绝大部分空间不会被立即分配(virtual)。它们会一直保留着，直到运行过程中，JVM发现实际占用接近已分配上限值时，才从virtual里再分配掉一部分内存。这里提到的已分配上限值，也可以叫做预估上限值。引入预估上限值的好处是，可以有效地控制堆的大小。堆越小，GC效率越高嘛。注意：预估上限值的大小一定小于或等于最大值。

* + - * -XX:MaxNewSize=size
        + 1.3.1 Sparc: 32m，1.3.1 x86: 2.5m
        + 新生代占整个堆内存的最大值。
      * -XX:MaxPermSize=64m
        + 5.0以后: 64 bit VMs会增大预设值的30%，1.4 amd64: 96m，1.3.1 -client: 32m，其他默认 64m
        + Perm（俗称方法区）占整个堆内存的最大值。
      * -XX:MinHeapFreeRatio=40
        + 40
        + GC后，如果发现空闲堆内存占到整个预估上限值的40%，则增大上限值。(什么是预估上限值？见 -XX:MaxHeapFreeRatio 处的描述)关联选项：-XX:MaxHeapFreeRatio=70
      * -XX:NewRatio=2
        + Sparc -client: 8，x86 -server: 8，x86 -client: 12，-client: 4 (1.3)，8 (1.3.1+)，x86: 12，其他默认 2
        + 新生代和年老代的堆内存占用比例。例如2例如2表示新生代占年老代的1/2，占整个堆内存的1/3。
      * -XX:NewSize=2.125m
        + 5.0以后: 64 bit Vms 会增大预设值的30%，x86: 1m，x86, 5.0以后: 640k，其他默认 2.125m
        + 新生代预估上限的默认值。(什么是预估上限值？见 -XX:MaxHeapFreeRatio 处的描述)
      * -XX:ReservedCodeCacheSize=32m
        + Solaris 64-bit, amd64, -server x86: 48m, 1.5.0\_06之前, Solaris 64-bit amd64: 1024m, 其他默认 32m
        + 设置代码缓存的最大值，编译时用。
      * -XX:SurvivorRatio=8
        + Solaris amd64: 6，Sparc in 1.3.1: 25，Solaris platforms 5.0以前: 32，其他默认 8
        + Eden与Survivor的占用比例。例如8表示，一个survivor区占用 1/8 的Eden内存，即1/10的新生代内存，为什么不是1/9？因为我们的新生代有2个survivor，即S1和S22。所以survivor总共是占用新生代内存的 2/10，Eden与新生代的占比则为 8/10。
      * -XX:TargetSurvivorRatio=50
        + 50
        + 实际使用的survivor空间大小占比。默认是50%，最高90%。该参数设置较大的话可提高对survivor空间的使用率。当较大的堆栈使用较低的SurvivorRatio时，应增加该值到80至90，以更好利用救助空间。
      * -XX:ThreadStackSize=512
        + Sparc: 512，Solaris x86: 320 (5.0以前 256)，Sparc 64 bit: 1024，Linux amd64: 1024 (5.0 以前 0)，其他默认 512.
        + 线程堆栈大小
      * -XX:+UseBiasedLocking
        + JDK 5 update 6后引入，但需要手动启用。JDK6默认启用。
        + 启用偏向锁。偏向锁原理详见 http://kenwublog.com/theory-of-java-biased-locking
      * -XX:+UseFastAccessorMethods
        + 默认启用
        + 优化原始类型的getter方法性能。
      * -XX:-UseISM
        + 默认启用
        + 启用solaris的ISM。详见Intimate Shared Memory.
      * -XX:+UseLargePages
        + JDK 5 update 5后引入，但需要手动启用。JDK6默认启用。
        + 启用大内存分页。调整内存页的方法和性能提升原理，详见http://kenwublog.com/tune-large-page-for-jvm-optimization。关联选项，-XX:LargePageSizeInBytes=4m
      * -XX:+UseMPSS
        + 1.4.1 之前: 不启用。其余版本默认启用
        + 启用solaris的MPSS，不能与ISM同时使用。
      * -XX:+UseStringCache
        + 默认启用
        + 启用字符串缓存。
      * -XX:AllocatePrefetchLines=1
        + 1
        + 与机器码指令预读相关的一个选项，资料比较少，本文档不做解释。有兴趣的朋友请自行阅读官方doc。
      * -XX:AllocatePrefetchStyle=1
        + 1
        + 与机器码指令预读相关的一个选项，资料比较少，本文档不做解释。有兴趣的朋友请自行阅读官方doc。
      * -XX:+UseCompressedStrings
        + 采用字节数组表示字符串(JDK6Update21引入)
        + Use a byte[] for Strings which can be represented as pure ASCII. (Introduced in Java 6 Update 21 Performance Release)
      * -XX:+OptimizeStringConcat
        + 在可能的情况下优化字符串连接操作（JDK6Update20引入）
        + Optimize String concatenation operations where possible. (Introduced in Java 6 Update 20)
      * -XX:ParallelGCThreads=
        + 设置Parallel垃圾收集器并行收集线程的个数，默认为CPU的个数
        + Sets the number of garbage collection threads in the young and old parallel garbage collectors. The default value varies with the platform on which the JVM is running.
      * -XX:+UseCompressedOops
        + 开启64位机器Java堆大小小于32GB的指针压缩
        + Enables the use of compressed pointers (object references represented as 32 bit offsets instead of 64-bit pointers) for optimized 64-bit performance with Java heap sizes less than 32gb.
      * -XX:+AlwaysPreTouch
        + Pre-touch the Java heap during JVM initialization. Every page of the heap is thus demand-zeroed during initialization rather than incrementally during application execution.
      * -XX:AllocatePrefetchDistance=
        + Sets the prefetch distance for object allocation. Memory about to be written with the value of new objects is prefetched into cache at this distance (in bytes) beyond the address of the last allocated object. Each Java thread has its own allocation point. The default value varies with the platform on which the JVM is running.
      * -XX:InlineSmallCode=
        + 代码内联的判断依据，调整编译后代码的字节大小是判断条件
        + Inline a previously compiled method only if its generated native code size is less than this. The default value varies with the platform on which the JVM is running.
      * -XX:MaxInlineSize=35
        + 限制动态编译的内联函数的虚拟机指令的最大数量
        + Maximum bytecode size of a method to be inlined.
      * -XX:FreqInlineSize=
        + 限制经常使用的动态编译的函数的虚拟机指令的最大数量
        + Maximum bytecode size of a frequently executed method to be inlined. The default value varies with the platform on which the JVM is running.
      * -XX:LoopUnrollLimit=
        + 编译优化时将循环展开的上限值
        + Unroll loop bodies with server compiler intermediate representation node count less than this value. The limit used by the server compiler is a function of this value, not the actual value. The default value varies with the platform on which the JVM is running.
      * -XX:InitialTenuringThreshold=7
        + 新生代中，parallel垃圾收集器初始化的年龄阀值为7.年龄阀值，值得是对象在Survivor区内经过了多少次MirrorGC后，晋升到老年代。
        + Sets the initial tenuring threshold for use in adaptive GC sizing in the parallel young collector. The tenuring threshold is the number of times an object survives a young collection before being promoted to the old, or tenured, generation.
      * -XX:MaxTenuringThreshold=
        + Survivor区晋升到老年代，对象的年龄阀值。当前最大值是15.Parallel垃圾收集器默认值是15，CMS垃圾收集器默认值是4
        + Sets the maximum tenuring threshold for use in adaptive GC sizing. The current largest value is 15. The default value is 15 for the parallel collector and is 4 for CMS.
    1. 调试参数
       - -XX:-CITime
         * 1.4引入。默认启用
         * 打印JIT编译器编译耗时。
       - -XX:ErrorFile=./hs\_err\_pid<pid>.log
         * Java 6引入。
         * 如果JVM crashed，将错误日志输出到指定文件路径。
       - -XX:-ExtendedDTraceProbes
         * Java6引入，限于solaris。默认不启用
         * 启用dtrace诊断。
       - -XX:HeapDumpPath=./java\_pid<pid>.hprof
         * 默认是java进程启动位置，即user.dir
         * 堆内存快照的存储文件路径。

什么是堆内存快照？

当java进程因OOM或crash被OS强制终止后，会生成一个hprof（Heap PROFling）格式的堆内存快照文件。该文件用于线下调试，诊断，查找问题。文件名一般为java\_<pid>\_<date>\_<time>\_heapDump.hprof解析快照文件，可以使用 jhat, eclipse MAT，gdb等工具。

* + - * -XX:-HeapDumpOnOutOfMemoryError
        + 1.4.2 update12 和 5.0 update 7 引入。默认不启用
        + 在OOM时，输出一个dump.core文件，记录当时的堆内存快照（什么是堆内存快照? 见 -XX:HeapDumpPath 处的描述）。
      * -XX:OnError="<cmd args>;<cmd args>"
        + 1.4.2 update 9引入
        + 当java每抛出一个ERROR时，运行指定命令行指令集。指令集是与OS环境相关的，在linux下多数是bash脚本，windows下是dos批处理。
      * -XX:OnOutOfMemoryError="<cmd args>; <cmd args>"
        + 1.4.2 update 12和java6时引入
        + 当第一次发生OOM时，运行指定命令行指令集。指令集是与OS环境相关的，在linux下多数是bash脚本，windows下是dos批处理。
      * -XX:-PrintClassHistogram
        + 默认不启用
        + 在Windows下, 按ctrl-break或Linux下是执行kill -3（发送SIGQUIT信号）时，打印class柱状图。Jmap –histo pid也实现了相同的功能。
      * -XX:-PrintConcurrentLocks
        + 默认不启用
        + 在thread dump的同时，打印java.util.concurrent的锁状态。Jstack –l pid 也同样实现了同样的功能。
      * -XX:-PrintCommandLineFlags
        + 5.0 引入，默认不启用
        + Java启动时，往stdout打印当前启用的非稳态jvm options。例如：-XX:+UseConcMarkSweepGC -XX:+HeapDumpOnOutOfMemoryError -XX:+DoEscapeAnalysis
      * -XX:-PrintCompilation
        + 默认不启用
        + 往stdout打印方法被JIT编译时的信息。
      * -XX:-PrintGC
        + 默认不启用
        + 开启GC日志打印。打印格式例如：[Full GC 131115K->7482K(1015808K), 0.1633180 secs]。该选项可通过 com.sun.management.HotSpotDiagnosticMXBean API 和 Jconsole 动态启用。
      * -XX:-PrintGCDetails
        + 1.4.0引入，默认不启用
        + 打印GC回收的细节。打印格式例如：[Full GC (System) [Tenured: 0K->2394K(466048K), 0.0624140 secs] 30822K->2394K(518464K), [Perm : 10443K->10443K(16384K)], 0.0625410 secs] [Times: user=0.05 sys=0.01, real=0.06 secs]。该选项可通过 com.sun.management.HotSpotDiagnosticMXBean API 和 Jconsole 动态启用。
      * -XX:-PrintGCTimeStamps
        + 默认不启用
        + 打印GC停顿耗时。打印格式例如：2.744: [Full GC (System) 2.744: [Tenured: 0K->2441K(466048K), 0.0598400 secs] 31754K->2441K(518464K), [Perm : 10717K->10717K(16384K)], 0.0599570 secs] [Times: user=0.06 sys=0.00, real=0.06secs]。该选项可通过 com.sun.management.HotSpotDiagnosticMXBean API 和 Jconsole 动态启用。
      * -XX:-PrintTenuringDistribution
        + 默认不启用
        + 打印对象的存活期限信息。 打印格式例如：[GC Desired survivor size 4653056 bytes, new threshold 32 (max 32) - age 1: 2330640 bytes, 2330640 total - age 2: 9520 bytes, 2340160 total 204009K->21850K(515200K), 0.1563482 secs]。Age1 2表示在第1和2次GC后存活的对象大小。
      * -XX:-TraceClassLoading
        + 默认不启用
        + 打印class装载信息到stdout。记Loaded状态。
      * -XX:-TraceClassLoadingPreorder
        + 1.4.2引入，默认不启用
        + 按class的引用/依赖顺序打印类装载信息到stdout。不同于 TraceClassLoading，本选项只记 Loading状态。
      * -XX:-TraceClassResolution
        + 1.4.2引入，默认不启用
        + 打印所有静态类，常量的代码引用位置。用于debug。
      * -XX:-TraceClassUnloading
        + 默认不启用
        + 打印class的卸载信息到stdout。记Unloaded状态。
      * -XX:-TraceLoaderConstraints
        + Java6 引入，默认不启用
        + 打印class的装载策略变化信息到stdout。

例如：

[Adding new constraint for name: java/lang/String, loader[0]: sun/misc/Launcher$ExtClassLoader, loader[1]: <bootloader> ]

[Setting class object in existing constraint for name: [Ljava/lang/Object; and loader sun/misc/Launcher$ExtClassLoader ]

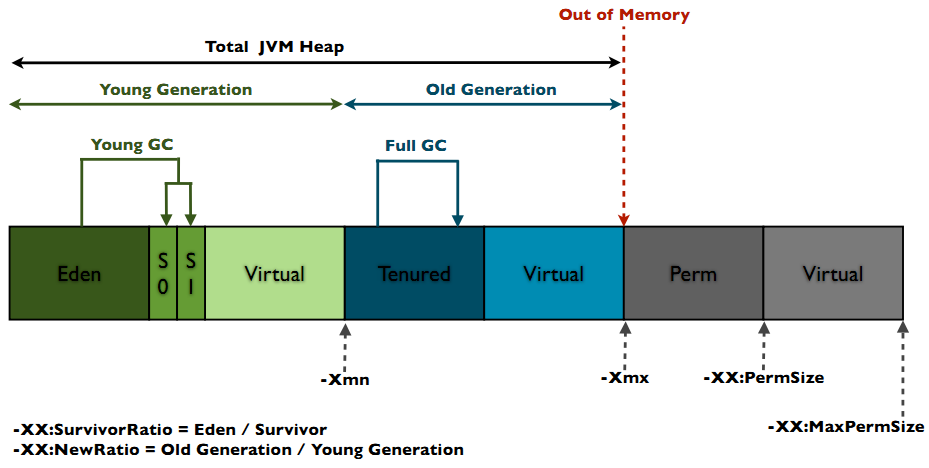
[Updating constraint for name org/xml/sax/InputSource, loader <bootloader>, by setting class object ]

[Extending constraint for name java/lang/Object by adding loader[15]: sun/reflect/DelegatingClassLoader ]

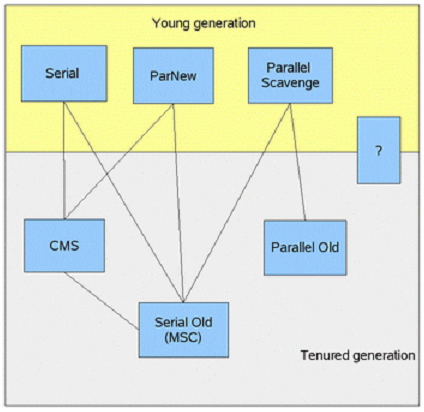
装载策略变化是实现classloader隔离/名称空间一致性的关键技术。

对此感兴趣的朋友，详见 http://kenwublog.com/docs/Dynamic+Class+Loading+in+the+Java+Virtual+Machine.pdf 中的 contraint rules一章。

* + - * -XX:+PerfSaveDataToFile
        + 默认启用
* 当java进程因OOM或crashed被强制终止后，生成一个堆快照文件（什么是堆内存快照? 见 -XX:HeapDumpPath 处的描述）。
  1. 参数说明（按常用参数）



* + 1. 通用参数
       - -server
         * 如果不配置该参数，JVM会根据应用服务器硬件配置自动选择不同模式，server模式启动比较慢，但是运行期速度得到了优化，适合于服务器端运行的JVM。
       - -client
         * 启动比较快，但是运行期响应没有server模式的优化，适合于个人PC的服务开发和测试。
       - -Xmx
         * 设置java heap的最大值，默认是机器物理内存的1/4。这个值决定了最多可用的Java堆内存：分配过少就会在应用中需要大量内存作缓存或者临时对象时出现OOM（Out Of Memory）的问题；如果分配过大，那么就会因PermSize过小而引起的另外一种Out Of Memory。所以如何配置还是根据运行过程中的分析和计算来确定，如果不能确定还是采用默认的配置。
       - -Xms
         * 设置Java堆初始化时的大小，默认情况是机器物理内存的1/64。这个主要是根据应用启动时消耗的资源决定，分配少了申请起来会降低运行速度，分配多了也浪费。
       - -Xmn
         * 直接设置青年代大小(1.4or lator)。整个JVM可用内存大小=青年代大小 + 老年代大小 + 持久代大小 。持久代一般固定大小为64m，所以增大年轻代后，将会减小老年代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。按照Sun的官方设置比例，则上面的例子中年轻代的大小应该为2048\*3/8=768M。**注意**：-Xmn、-XX:NewSize/MaxNewSize和-XX:NewRatio，三个参数都可以控制新生代的大小，优先级为-Xmn与-XX:NewSize/MaxNewSize持平，大于-XX:NewRatio。也就是说-Xmn或-XX:NewSize/MaxNewSize任一存在时，-XX:NewRatio失效。-Xmn和-XX:NewSize/MaxNewSize同时存在，且设置不同值时，两个参数谁排在启动参数后面，新生代的值采用谁。
       - -Xss
         * 设置每个线程的堆栈大小，根据应用的线程所需内存大小进行调整，在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。当这个选项被设置的较大（>2MB）时将会在很大程度上降低系统的性能。因此在设置这个值时应该格外小心，调整后要注意观察系统的性能，不断调整以期达到最优。JDK5.0以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。
       - -Xnoclassgc
         * 这个选项用来取消系统对特定类的垃圾回收。它可以防止当这个类的所有引用丢失之后，这个类仍被引用时不会再一次被重新装载，因此这个选项将增大系统堆内存的空间。禁用类垃圾回收，性能会高一点；
       - -XX:PermSize
         * 初始化永久内存区域大小。永久内存区域全称是Permanent Generation space，是指内存的永久保存区域，程序运行期不对PermGen space进行清理，所以如果你的APP会LOAD很多CLASS的话,就很可能出现PermGen space错误。这种错误常见在web服务器对JSP进行pre compile的时候。 如果你的WEB APP下用了大量的第三方jar，其大小超过了jvm默认的PermSize大小(4M)那么就会产生此错误信息了。
       - -XX:MaxPermSize
         * 设置永久内存区域最大大小。
       - -XX:NewSize
         * (for 1.3/1.4)中设置新生代的大小。**注意**：-Xmn、-XX:NewSize/MaxNewSize和-XX:NewRatio，三个参数都可以控制新生代的大小，优先级为-Xmn与-XX:NewSize/MaxNewSize持平，大于-XX:NewRatio。也就是说-Xmn或-XX:NewSize/MaxNewSize任一存在时，-XX:NewRatio失效。-Xmn和-XX:NewSize/MaxNewSize同时存在，且设置不同值时，两个参数谁排在启动参数后面，新生代的值采用谁。
       - -XX:MaxNewSize
         * (for 1.3/1.4)中设置新生代的大小。
       - -XX:NewRatio
         * 控制默认的Young代的大小，例如，设置-XX:NewRatio=3意味着Young代和老年代的比率是1:3。换句话说，Eden和Survivor空间总和是整个堆大小的1/4。**注意**：-Xmn、-XX:NewSize/MaxNewSize和-XX:NewRatio，三个参数都可以控制新生代的大小，优先级为-Xmn与-XX:NewSize/MaxNewSize持平，大于-XX:NewRatio。也就是说-Xmn或-XX:NewSize/MaxNewSize任一存在时，-XX:NewRatio失效。-Xmn和-XX:NewSize/MaxNewSize同时存在，且设置不同值时，两个参数谁排在启动参数后面，新生代的值采用谁。
       - -XX:SurvivorRatio
* 新生代这的Eden区域与Survivor区域的容量比值，默认是8，代表Eden:Survivor=8:1。比如设置为4，则两个Survivor区与一个Eden区的比值为2:4，一个Survivor区占整个年轻代的1/6。越大的survivor空间可以允许短期对象尽量在年青代消亡；如果Survivor空间太小，Copying收集将直接将其转移到老年代中，这将加快老年代的空间使用速度，引发频繁的完全垃圾回收。
* -XX:PretenureSizeThreshold
* 直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配(只对Serial和PraNew收集器有效)
  + - * -XX:MaxTenuringThreshold
* 晋升到老年代的对象年龄，每个对象在坚持过一次的MinorGC之后，年龄就加1，当超过这个参数值时就进入老年代。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入老年代。对于老年代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象在年轻代的存活时间，增加在年轻代即被回收的概率。
  + - * -XX:SoftRefLRUPolicyMSPerMB
* 相对于客户端模式的虚拟机（-client选项），当使用服务器模式的虚拟机时（-server选项），对于软引用（soft reference）的清理力度要稍微差一些。可以通过增大-XX:SoftRefLRUPolicyMSPerMB来降低收集频率。默认值是 1000，也就是说每秒一兆字节。Soft reference在虚拟机中比在客户集中存活的更长一些。其清除频率可以用命令行参数 -XX:SoftRefLRUPolicyMSPerMB=<N> 来控制，这可以指定每兆堆空闲空间的 soft reference 保持存活（一旦它不强可达了）的毫秒数，这意味着每兆堆中的空闲空间中的 soft reference 会（在最后一个强引用被回收之后）存活1秒钟。注意，这是一个近似的值，因为 soft reference 只会在垃圾回收时才会被清除，而垃圾回收并不总在发生。每兆堆空闲空间中SoftReference的存活时间
  + - * -XX:LargePageSizeInBytes
        + 内存页的大小， 不可设置过大，会影响Perm的大小。一般情况，不建议将页size调得太大，4-64M，是可以接受的（默认是4M）。为了合理设置这个值，你应该对你的系统做一下benchmark。说实话，网上我见过调得最猛的，有调到256M，从benchmark报表上看，性能不是太坏。如果你有64位的大内存机器，不妨尝试一下。另外，网上有很多GC调优的文章内容中都有提到 LargePageSizeInBytes，但未提任何OS限制。在OS不支持的情况下，设置这个参数，这个参数将仅仅是个摆设。
      * -XX:+UseFastAccessorMethods
        + 原始类型的快速优化，get,set 方法转成本地代码。
      * -XX:+DisableExplicitGC
        + 禁止 java 程序中的 full gc, 如 System.gc() 的调用。 最好加上防止程序在代码里误用了，对性能造成冲击。
      * -XX:+AggressiveHeap
        + 特别说明下：(我感觉对于做java cache应用有帮助) ，试图是使用大量的物理内存 ，长时间大内存使用的优化，能检查计算资源（内存， 处理器数量）， 至少需要256MB内存，大量的CPU／内存， （在1.4.1在4CPU的机器上已经显示有提升）
      * -XX:+AggressiveOpts
        + 加快编译
      * -XX:+UseBiasedLocking
        + 锁机制的性能改善。
      * -XX:MaxDirectMemorySize
* 指定DirectMemory的容量，如果不指定，则默认与Java堆的最大值(-Xmx)一样（NIO可以使用native函数库直接分配堆外内存，可能造成该区内存溢出）。



图：JVM1.6垃圾收集器组合

* + 1. 新生代
       1. Serial
* -XX:+UseSerialGC
  + 虚拟机运行在Client模式下的默认值，打开此开关后，使用Serial+Serial Old的收集器组合进行内存回收
    - 1. ParNew
* -XX:+UseParNewGC
  + 打开此开关后，使用ParNew+Serial Old收集器组合进行内存回收，是多线程的并行收集器
* -XX:ParallelGCThreads
  + 设置并行GC是进行内存回收的线程数，默认为CPU的个数
    - 1. Parallel Scavenge
* -XX:+UseParallelGC
  + 虚拟机运行在Server模式下的默认值，打开此开关后，使用Parallel Scavenge+Serial Old的收集器进行内存回收，即年轻代使用并行收集，而老年代仍旧使用串行收集。采用了多线程并行管理和回收垃圾对象，提高了回收效率，提高了服务器的吞吐量，适合于多处理器的服务器。启动的多线程数，当CPU核数<=8时，为CPU核数；当多余8核时为3+(核数 \*5)/8，如16核为13线程。
* -XX:MaxGCPauseMillis

设置GC的最大停顿时间。仅在使用Parallel收集器时生效。停顿时间下载，吞吐量也将下降。

* -XX:GCTimeRatio

GC时间占总时间的比率，默认值是99，即允许1%的GC时间，仅在使用Parallel Scavenge收集器时有效。该参数设置为n的话，则垃圾回收时间占程序运行时间百分比的公式为1/(1+n) ，如果n=19表示java可以用5%的时间来做垃圾回收，1/(1+19)=1/20=5%。

* -XX:+UseAdaptiveSizePolicy

动态调整Java堆中各个区域的大小以及进入老年代的年龄，以达到目标系统规定的最低响应时间或者收集频率等。

* -XX:ParallelGCThreads
  + 设置并行GC是进行内存回收的线程数，<=8核，或者3+(核数\*5)/8
    1. 老生代
       1. Serial Old(MSC)
       2. CMS
* -XX:+UseConcMarkSweepGC
  + 打开此开关后，使用ParNew+CMS+Serial Old的收集器组合进行内存回收。SerialOld收集器将作为CMS收集器出现Concurrent Mode Failure失败后的后备收集器使用。默认回收线程数: (幵行GC线程数+3）/4，也可用-XX:ParallelCMSThreads=10指定
* -XX:CMSInitiatingOccupancyFraction

设置CMS收集器在老年代空间被使用多少后触发垃圾收集器，默认值68%，仅在使用CMS收集器时生效。

这个参数设置有很大技巧，基本上满足公式：

(Xmx-Xmn)\*(100-CMSInitiatingOccupancyFraction)/100>=Xmn

时就不会出现promotion failed。在我的应用中Xmx是6000，Xmn是500，那么Xmx-Xmn是5500兆，也就是老年代有5500兆，CMSInitiatingOccupancyFraction=90说明老年代到90%满的时候开始执行对老年代的并发垃圾回收（CMS），这时还剩10%的空间是5500\*10%=550兆，所以即使Xmn（也就是年轻代共500兆）里所有对象都搬到老年代里，550兆的空间也足够了，所以只要满足上面的公式，就不会出现垃圾回收时的promotion failed；如果按照Xmx=2048,Xmn=768的比例计算，则CMSInitiatingOccupancyFraction的值不能超过40，否则就容易出现垃圾回收时的promotion failed。

* -XX:+CMSCompactAtFullCollection

仅在使用CMS收集器时生效,设置CMS收集器在完成垃圾收集后是否需要进行一次内存碎片整理。CMS是不会移动内存的， 因此， 这个非常容易产生碎片， 导致内存不够用， 因此， 内存的压缩这个时候就会被启用。 增加这个参数是个好习惯。可能会影响性能,但是可以消除碎片

* -XX:CMSFullGCsBeforeCompaction

仅在使用CMS收集器时生效，设置CMS收集器在执行多少次不压缩的Full GC后，跟着来一次带压缩的。由于并发收集器不对内存空间进行压缩,整理,所以运行一段时间以后会产生"碎片",使得运行效率降低.此值设置运行多少次GC以后对内存空间进行压缩,整理.

* -XX:+CMSClassUnloadingEnabled
  + 使CMS收集持久代的类，而不是fullgc，默认是92%时开始收集
* -XX:+CMSPermGenSweepingEnabled

使CMS收集持久代的类，而不是fullgc

* -XX:CMSInitiatingPermOccupancyFraction
  + 设置CMS收集器在持久代空间被使用多少后触发垃圾收集器，默认92%
* -XX:-CMSParallelRemarkEnabled

在使用 UseParNewGC 的情况下 , 尽量减少 mark 的时间。降低标记停顿。

* -XX:+UseCMSInitiatingOccupancyOnly

指示只有在老年代在使用了初始化的比例后 concurrent collector 启动收集。使用手动定义初始化定义开始CMS收集。禁止hostspot自行触发CMS GC。

* + - 1. Parallel Old
* -XX:+UseParallelOldGC
  + 打开此开关后，使用Parrallel Scavenge + Parallel Old的收集器组合进行内存回收
  1. 调试参数

|  |
| --- |
| -verbose:gc 输出每次GC的相关情况。  -Xloggc:gc.log 把相关日志信息记录到文件以便分析  -XX:ErrorFile=./hs\_err\_pid<pid>.log 如果JVM crashed，将错误日志输出到指定文件路径。  -XX:+HeapDumpOnOutOfMemoryError 在OOM时，输出dump.core文件，记录当时的堆内存快照  -XX:HeapDumpPath=./java\_pid<pid>.hprof 堆内存快照的存储文件路径  -XX:+PrintCommandLineFlags Java启动时，打印当前启用的非稳态jvm options  -XX:+PrintGC 开启GC日志打印  -XX:+PrintGCDetails 打印GC回收的细节  -XX:+PrintGCTimeStamps 打印GC停顿耗时  -XX:+PrintGCApplicationStoppedTime 打印垃圾回收期间程序暂停的时间  -XX:+PrintGCApplicationConcurrentTime 打印每次垃圾回收前,程序未中断的执行时间  -XX:+PrintHeapAtGC 打印GC前后的详细堆栈信息  -XX:+PrintTenuringDistribution 查看每次minor GC后新的存活周期的阈值  -XX:+PrintClassHistogram 打印class柱状图 |

* 1. 配置策略
     1. 策略一

一般建议young区为整个堆的1/4，如堆为2g，则young区分配500M。sun推荐的配置，survivor区一般设置为young区的1/8，如果young区为500M，则survivor可以设置为60M。如果一个线程占用的内存为2M，则50M的survivor 支持25个并发线程是肯定OK的

（实际上jvm调优应该是以减少GC的时间和系统停顿时间为目的而进行堆内存的各个区段的分配，以及GC策略的调整）

* + 1. 策略二

年轻代设置策略：对于响应时间优先的应用需尽可能设大，直到接近系统的最低响应时间限制（根据实际情况选择）。在此种情况下，年轻代收集发生的频率也是最小的。同时，减少到达老年代的对象。对于吞吐量优先的应用则尽可能的设置大，可达到Gbit的程度。因为对响应时间没有要求，垃圾收集可以并行进行，一般适合8CPU以上的应用。

老年代设置策略：对于响应时间优先的应用，老年代使用并发收集器，所以其大小需要小心设置，一般要考虑**并发会话率**和**会话持续时间**等一些参数。如果堆设置小了，可能会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。

最优化的方案，一般需要参考以下数据获得：

* + 并发垃圾收集信息
  + 持久代并发收集次数
  + 传统GC信息
  + 花在年轻代和老年代回收上的时间比例
  + 减少年轻代和老年代花费的时间，一般会提高应用的效率

对于吞吐量优先的应用，一般吞吐量优先的应用都有一个很大的年轻代和一个较小的老年代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而老年代尽存放长期存活对象。

较小堆引起的碎片问题 ：因为老年代的并发收集器使用标记、清除算法，所以不会对堆进行压缩。当收集器回收时，他会把相邻的空间进行合并，这样可以分配给较大的对象。但是，当堆空间较小时，运行一段时间以后，就会出现“碎片”，如果并发收集器找不到足够的空间，那么并发收集器将会停止，然后使用传统的标记、清除方式进行回收。如果出现“碎片”，可能需要进行如下配置： -XX:+UseCMSCompactAtFullCollection 使用并发收集器时，开启对老年代的压缩；-XX:CMSFullGCsBeforeCompaction=0上面配置开启的情况下，这里设置多少次Full GC后，对老年代进行压缩。

* + 1. 策略三

**promotion failed:**

垃圾回收时promotion failed是个很头痛的问题，一般可能是两种原因产生，第一个原因是救助空间不够，救助空间里的对象还不应该被移动到年老代，但年轻代又有  
很多对象需要放入救助空间；第二个原因是年老代没有足够的空间接纳来自年轻代的对象；这两种情况都会转向Full GC，网站停顿时间较长。

解决方方案一：

*第一个原因我的最终解决办法是去掉救助空间，设置-XX:SurvivorRatio=65536 -XX:MaxTenuringThreshold=0即可，第二个原因我的解决办法是设置  
CMSInitiatingOccupancyFraction为某个值（假设70），这样年老代空间到70%时就开始执行CMS，年老代有足够的空间接纳来自年轻代的对象。*

解决方案一的改进方案：

*又有改进了，上面方法不太好，因为没有用到救助空间，所以年老代容易满，CMS执行会比较频繁。我改善了一下，还是用救助空间，但是把救助空间加大，这样也不  
会有promotion failed。具体操作上，32位Linux和64位Linux好像不一样，64位系统似乎只要配置MaxTenuringThreshold参数，CMS还是有暂停。为了解  
决暂停问题和promotion failed问题，最后我设置-XX:SurvivorRatio=1 ，并把MaxTenuringThreshold去掉，这样即没有暂停又不会有promotoin failed，而  
且更重要的是，年老代和永久代上升非常慢（因为好多对象到不了年老代就被回收了），所以CMS执行频率非常低，好几个小时才执行一次，这样，服务器都不用重启了。*

-Xmx4000M -Xms4000M -Xmn600M -XX:PermSize=500M -XX:MaxPermSize=500M -Xss256K -XX:+DisableExplicitGC -XX:SurvivorRatio=1   
-XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection   
-XX:CMSFullGCsBeforeCompaction=0 -XX:+CMSClassUnloadingEnabled -XX:LargePageSizeInBytes=128M -XX:+UseFastAccessorMethods -XX:+UseCMSInitiatingOccupancyOnly -XX:CMSInitiatingOccupancyFraction=80 -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+PrintClassHistogram   
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC -Xloggc:log/gc.log

* + 1. 策略四

CMSInitiatingOccupancyFraction值与Xmn的关系公式

上面介绍了promontion faild产生的原因是EDEN空间不足的情况下将EDEN与From survivor中的存活对象存入To survivor区时,To survivor区的空间不足，  
再次晋升到old gen区，而old gen区内存也不够的情况下产生了promontion faild从而导致full gc.那可以推断出：eden+from survivor < old gen区剩余内  
存时，不会出现promontion faild的情况，即：  
(Xmx-Xmn)\*(1-CMSInitiatingOccupancyFraction/100)>=(Xmn-Xmn/(SurvivorRatior+2))  进而推断出：

CMSInitiatingOccupancyFraction <=((Xmx-Xmn)-(Xmn-Xmn/(SurvivorRatior+2)))/(Xmx-Xmn)\*100

例如：

当xmx=128 xmn=36 SurvivorRatior=1时 CMSInitiatingOccupancyFraction<=((128.0-36)-(36-36/(1+2)))/(128-36)\*100 =73.913

当xmx=128 xmn=24 SurvivorRatior=1时 CMSInitiatingOccupancyFraction<=((128.0-24)-(24-24/(1+2)))/(128-24)\*100=84.615…

当xmx=3000 xmn=600 SurvivorRatior=1时  CMSInitiatingOccupancyFraction<=((3000.0-600)-(600-600/(1+2)))/(3000-600)\*100=83.33

CMSInitiatingOccupancyFraction低于70% 需要调整xmn或SurvivorRatior值。

令：

[网上一童鞋](http://bbs.weblogicfans.net/archiver/tid-2835.html)推断出的公式是：:(Xmx-Xmn)\*(100-CMSInitiatingOccupancyFraction)/100>=Xmn 这个公式个人认为不是很严谨，在内存小的时候会影响xmn的计算。

* 1. 典型配置
     1. 实例一

|  |
| --- |
| 根据反复的测试并结合被测系统业务特点，最终敲定了使用以下最优方案进行8小时压力测试：  JAVA\_OPTS=-server  -Xms1024M  -Xmx1024M  -Xmn128M  -XX:NewSize=128M  -XX:MaxNewSize=128M  -XX:SurvivorRatio=20  -XX:MaxTenuringThreshold=10  -XX:GCTimeRatio=19  -XX:+UseParNewGC  -XX:+UseConcMarkSweepGC  -XX:+CMSClassUnloadingEnabled  -XX:+UseCMSCompactAtFullCollection  -XX:CMSFullGCsBeforeCompaction=0  -XX:-CMSParallelRemarkEnabled  -XX:CMSInitiatingOccupancyFraction=70  -XX:SoftRefLRUPolicyMSPerMB=0  -XX:PermSize=256m  -XX:MaxPermSize=256m  -Djava.awt.headless=true  对于XX银行采购系统的登录操作来说，将jvm的NewRatio 和SurviorRatio设置成4时，性能表现最好！在此基础上在设置-XX:TargetSurvivorRatio=90和-Xmx1024M后性能也有一定程度的提升。 |

* + 1. 实例二

|  |
| --- |
| 舍得网的典型配置  $JAVA\_ARGS .= " -Dresin.home=$SERVER\_ROOT  -server  -Xms6000M  -Xmx6000M  -Xmn500M  -XX:PermSize=500M  -XX:MaxPermSize=500M  -XX:SurvivorRatio=65536  -XX:MaxTenuringThreshold=0  -Xnoclassgc  -XX:+DisableExplicitGC  -XX:+UseParNewGC  -XX:+UseConcMarkSweepGC  -XX:+UseCMSCompactAtFullCollection  -XX:CMSFullGCsBeforeCompaction=0  -XX:+CMSClassUnloadingEnabled  -XX:-CMSParallelRemarkEnabled  -XX:CMSInitiatingOccupancyFraction=90  -XX:SoftRefLRUPolicyMSPerMB=0  -XX:+PrintClassHistogram  -XX:+PrintGCDetails  -XX:+PrintGCTimeStamps  -XX:+PrintHeapAtGC  -Xloggc:log/gc.log ";  -Dcom.sun.management.jmxremote  -Dcom.sun.management.jmxremote.port=9004  -Dcom.sun.management.jmxremote.authenticate=false  -Dcom.sun.management.jmxremote.ssl=false  说明：  1、 -XX:SurvivorRatio=65536 -XX:MaxTenuringThreshold=0就是去掉了救助空间；  2、-Xnoclassgc禁用类垃圾回收，性能会高一点；  3、-XX:+DisableExplicitGC禁止System.gc()，免得程序员误调用gc方法影响性能；  4、-XX:+UseParNewGC，对年轻代采用多线程并行回收，这样收得快；  带CMS参数的都是和并发回收相关的，不明白的可以上网搜索；  CMSInitiatingOccupancyFraction，这个参数设置有很大技巧，基本上满足(Xmx-Xmn)\*(100-CMSInitiatingOccupancyFraction)/100>=Xmn就不会 出现promotion failed。在我的应用中Xmx是6000，Xmn是500，那么Xmx-Xmn是5500兆，也就是年老代有5500兆，CMSInitiatingOccupancyFraction=90 说明年老代到90%满的时候开始执行对年老代的并发垃圾回收（CMS），这时还剩10%的空间是5500\*10%=550兆，所以即使Xmn（也就是年轻代共500兆）里所 有对象都搬到年老代里，550兆的空间也足够了，所以只要满足上面的公式，就不会出现垃圾回收时的promotion failed；  SoftRefLRUPolicyMSPerMB这个参数我认为可能有点用，官方解释是softly reachable objects will remain alive for some amount of time after the last time  they were referenced. The default value is one second of lifetime per free megabyte in the heap，我觉得没必要等1秒；  -Xmx4000M -Xms4000M -Xmn600M -XX:PermSize=500M -XX:MaxPermSize=500M -Xss256K -XX:+DisableExplicitGC -XX:SurvivorRatio=1 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=0 -XX:+CMSClassUnloadingEnabled -XX:LargePageSizeInBytes=128M -XX:+UseFastAccessorMethods -XX:+UseCMSInitiatingOccupancyOnly -XX:CMSInitiatingOccupancyFraction=80 -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+PrintClassHistogram -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC -Xloggc:log/gc.log  改进方案：  上面方法不太好，因为没有用到救助空间，所以年老代容易满，CMS执行会比较频繁。我改善了一下，还是用救助空间，但是把救助空间加大，这样也不会有promotion failed。 具体操作上，32位Linux和64位Linux好像不一样，64位系统似乎只要配置MaxTenuringThreshold参数，CMS还是有暂停。为了解决暂停问题和promotion failed问题，最后我设置-XX:SurvivorRatio=1 ，并把MaxTenuringThreshold去掉，这样即没有暂停又不会有promotoin failed，而且更重要的是，年老代和永久代上升非常慢（因为好多对象到不了年老代就被回收了），所以CMS执行频率非常低，好几个小时才执行一次，这样，服务器都不用重启了。 |

* + 1. 实例三

|  |
| --- |
| 本文提供的设置仅仅是在高压力， 多CPU， 高内存环境下设置  -server -Xmx3g -Xms3g -XX:MaxPermSize=128m  -XX:NewRatio=1 eden/old 的比例  -XX:SurvivorRatio=8 s/e的比例  -XX:+UseParallelGC  -XX:ParallelGCThreads=8  -XX:+UseParallelOldGC 这个是JAVA 6出现的参数选项  -XX:LargePageSizeInBytes=128m 内存页的大小， 不可设置过大， 会影响Perm的大小。  -XX:+UseFastAccessorMethods 原始类型的快速优化  -XX:+DisableExplicitGC 关闭System.gc()  另外 -Xss 是线程栈的大小， 这个参数需要严格的测试， 一般小的应用， 如果栈不是很深， 应该是128k够用的， 不过，我们的应用调用深度比较大， 还需要做详细的测试。 这个选项对性能的影响比较大。 建议使用256K的大小.  压力测试下合适结果：  -server -XX:+PrintGCTimeStamps -XX:+PrintGCDetails -Xmx2g -Xms2g -Xmn256m -  XX:PermSize=128m -Xss256k -XX:MaxTenuringThreshold=31 -XX:+DisableExplicitGC -  XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -  XX:+UseCMSCompactAtFullCollection -XX:LargePageSizeInBytes=128m -  XX:+UseFastAccessorMethods  GC参数：  JAVA\_OPTS=" -server -XX:+PrintGCApplicationStoppedTime -XX:+PrintGCTimeStamps -  XX:+PrintGCDetails -Xmx2g -Xms2g -Xmn256m -XX:PermSize=128m -Xss256k -  XX:MaxTenuringThreshold=31 -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC -  XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -  XX:LargePageSizeInBytes=128m -XX:+UseFastAccessorMethods -  XX:+UseCMSInitiatingOccupancyOnly -XX:CMSInitiatingOccupancyFraction=70 "  实际上我们可以看到并行young gc执行时间是： 324.198s／15211＝20ms, cms的执行时间是  1.668／66＝25ms. 当然严格来说， 这么算是不对的， 世界停顿的时间要比这是数据稍微大5-10ms. 对我们来说如果不输出日志， 对我们是有参考意义的。 |

* + 1. 实例四

某网友:

$JAVA\_ARGS  
.=  
"  
-Dresin.home=$SERVER\_ROOT  
-server  
-Xmx3000M  
-Xms3000M  
-Xmn600M  
-XX:PermSize=500M  
-XX:MaxPermSize=500M  
-Xss256K  
-XX:+DisableExplicitGC  
-XX:SurvivorRatio=1  
-XX:+UseConcMarkSweepGC  
-XX:+UseParNewGC  
-XX:+CMSParallelRemarkEnabled  
-XX:+UseCMSCompactAtFullCollection  
-XX:CMSFullGCsBeforeCompaction=0  
-XX:+CMSClassUnloadingEnabled  
-XX:LargePageSizeInBytes=128M  
-XX:+UseFastAccessorMethods  
-XX:+UseCMSInitiatingOccupancyOnly  
-XX:CMSInitiatingOccupancyFraction=70  
-XX:SoftRefLRUPolicyMSPerMB=0  
-XX:+PrintClassHistogram  
-XX:+PrintGCDetails  
-XX:+PrintGCTimeStamps  
-XX:+PrintHeapAtGC  
-Xloggc:log/gc.log  
";

64位jdk参考设置，年老代涨得很慢，CMS执行频率变小，CMS没有停滞，也不会有promotion failed问题，内存回收得很干净

* 1. 实例说明

说明：

|  |
| --- |
| -server  -Xmx1024M  -Xms1024M  -Xss256K  -XX:PermSize=128M  -XX:MaxPermSize=128M  -Xmn256M  -XX:SurvivorRatio=8  -XX:CMSInitiatingOccupancyFraction=69  -XX:MaxTenuringThreshold=15  -XX:+DisableExplicitGC  -XX:+UseConcMarkSweepGC  -XX:+UseParNewGC  -XX:+CMSParallelRemarkEnabled  -XX:+UseCMSCompactAtFullCollection  -XX:CMSFullGCsBeforeCompaction=0  -XX:+CMSClassUnloadingEnabled  -XX:LargePageSizeInBytes=128M  -XX:+UseFastAccessorMethods  -XX:+UseCMSInitiatingOccupancyOnly  -XX:SoftRefLRUPolicyMSPerMB=0  -XX:+UseCompressedOops  -Dcom.sun.management.jmxremote  -Dcom.sun.management.jmxremote.port=9004  -Dcom.sun.management.jmxremote.authenticate=false  -Dcom.sun.management.jmxremote.ssl=false |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 128（85） | 192（76） | 256（66） | 320（54） | 384（40） | 448（22） |
| 1 | 10431  136.203s  6  256.189ms | 4643  105.739s  17  3.111s | 7893  133.238s  6  388.682ms | 4163  113.714 s  10  502.330ms | 3512  100.891s  14  708.299 ms | 3043  87.567s  40  2.243s |
| 2 | 6839  156.028  4  264.366ms | 4692  126.217s  6  302.589ms | 3430  101.918  4  253.345ms | 2323  83.415  8  562.976ms | 2406  79.278  8  514.307 | 2026  67.972  22  1.283s |
| 3 | 5686  148.006  4  384.861 | 3763  120.055  4  274.276 | 2925  92.324  4  257.069 | 2365  77.683  4  197.239 | 1936  65.411  6  367.790 | 1756  59.017  18  1.321s |
| 4 | 4962  131.984  4  1.204 |  |  |  |  |  |
| 5 |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  |
| 7 |  | 3083  82.288s  6  378.982ms |  |  |  |  |
| 8 |  | 3066  75.763s  6  375.754ms | 2130  69.206s  8  640.647ms | 1643  61.894  8  610.666 | 1450  55.892  12  915.672 |  |
| 12 |  |  |  |  |  |  |
| 16 |  |  |  |  |  |  |
| 20 |  |  |  |  |  |  |

* 1. JVM零散记录
  + JDK1.6Update14后，提供了普通对象指针压缩功能（**-XX:+UseCompressedOops**），在解释器解释字节码时，植入压缩指令以节省内存消耗
  + **-Xoss**设置本地方法栈大小，该参数实际无效，可以通过-Xss设定
  + **-XX:+HandlePromotionFailure**:JDK1.6Update24后，废除是否允许担保失败
  + **-XX:-DontCompileHugeMethods**：解除大方法的编译限制（Hotspot对字节码超过8000字节的大方法的限制，当方法即使满足JIT编译阈值CompileThreshold=1500次的要求，也只进行解释方式执行，而不动态优化编译成机器码执行，这样照成代码执行速度下降）。解除限制后，一个比较明显的缺点是JVM会尝试编译所遇到的所有大方法，者会使JIT编译任务负担更重，而且需要占用更多的Code Cache区域去保存编译后的代码。但是优点是编译后可以让大方法的执行速度变快，且可能提高GC速度。一旦Code Cache满了，HotSpot会停止所有后续的编译任务，虽然已编译的代码不受影响，但是后面的所有方法都会强制停留在纯解释模式。因此，如非必要，应该尽量避免生成大方法；如果解除了大方法的编译限制，则要留意配置Code Cache区的大小，准备更多空间存放编译后的代码（运行时Code Cache的使用量可以通过JMX或者JConsole获得）。

1. 性能监控
   1. JMX

|  |
| --- |
| -Dcom.sun.management.jmxremote  -Dcom.sun.management.jmxremote.port=9004  -Dcom.sun.management.jmxremote.authenticate=false  -Dcom.sun.management.jmxremote.ssl=false |

* 1. jps

虚拟机进程状况工具

jps 命令格式:

jps [options] [hostid]

jps的主要选项：

* -q 只输出LVMID，省略主类的名称
* -m 输出虚拟机进程启动时传递给主类main()函数的参数
* -l 输出主类的全名，如果进程执行的是jar包，输出jar包的路径。
* -v 输出虚拟机进程启动JVM参数

jps可以通过RMI协议查询开启RMI服务的远程虚拟机进程的状态，hostid为RMI注册表中注册的主机名。

* 1. jstat

虚拟机统计信息监视工具

jstat 命令格式:

jstat [option [-t] [-h<lines>] vmid [interval [s|ms] [count]]]

如果是远程虚拟机进行那么vmid的格式是:

[protocol:][//]lvmid[@hostname[:port]/servername]

实例： jstat –gc 2765 250 20 表示对2764进行每250毫秒查询一次垃圾收集情况，共计20次。

jstat 的主要参数

* -h n 每隔几行输出标题
* -t 在第一列显示自JVM启动以来的时间戳
* -J 修改java进程的参数。类似jinfo -flag <name>=<value>。例如-J-Xms48m 设置初始堆为48M。

jstat的主要选项

* -class 监视类装载、卸载数量、总空间及类装载所消耗的时间
* -gc 监视java堆状况、包含Eden区、2个survivor区，老年代、永久带的容量、已用空间、GC时间合计等
* -gccapacity 监视内容与-gc基本相同，但是输出主要关注java堆各个区域使用到的最大和最小空间
* -gcutil监视内容与-gc基本相同，但是输出主要关注已使用空间占总空间的百分比
* -gccause监视内容与-gcutil基本相同，但是会额外输出导致上一次GC产生的原因
* -gcnew 监视新生代GC的状况
* -gcnewcapacity监视内容与-gcnew基本相同，输出主要关注使用的最大和最小空间
* -gcold 监视老生代GC的状况
* -gcoldcapacity监视内容与-gcold基本相同, 输出主要关注使用的最大和最小空间
* -gcpermcapacity输出永久代使用到的最大和最小空间
* -compiler 输出JIT编译器编译过的方法、耗时等信息
* -printcompilation 输出已经被JIT编译的方法

|  |
| --- |
| -gc Option:Garbage-collected heap statistics  Column Description  S0C Current survivor space 0 capacity (KB).  S1C Current survivor space 1 capacity (KB).  S0U Survivor space 0 utilization (KB).  S1U Survivor space 1 utilization (KB).  EC Current eden space capacity (KB).  EU Eden space utilization (KB).  OC Current old space capacity (KB).  OU Old space utilization (KB).  PC Current permanent space capacity (KB).  PU Permanent space utilization (KB).  YGC Number of young generation GC Events.  YGCT Young generation garbage collection time.  FGC Number of full GC events.  FGCT Full garbage collection time.  GCT Total garbage collection time. |
| -gccapacity Option:Memory Pool Generation and Space Capacities  Column Description  NGCMN Minimum new generation capacity (KB).  NGCMX Maximum new generation capacity (KB).  NGC Current new generation capacity (KB).  S0C Current survivor space 0 capacity (KB).  S1C Current survivor space 1 capacity (KB).  EC Current eden space capacity (KB).  OGCMN Minimum old generation capacity (KB).  OGCMX Maximum old generation capacity (KB).  OGC Current old generation capacity (KB).  OC Current old space capacity (KB).  PGCMN Minimum permanent generation capacity (KB).  PGCMX Maximum Permanent generation capacity (KB).  PGC Current Permanent generation capacity (KB).  PC Current Permanent space capacity (KB).  YGC Number of Young generation GC Events.  FGC Number of Full GC Events. |
| -gccause Option:Garbage Collection Statistics, Including GC Events  Column Description  LGCC Cause of last Garbage Collection.  GCC Cause of current Garbage Collection. |
| -gcnew Option:New Generation Statistics  Column Description  S0C Current survivor space 0 capacity (KB).  S1C Current survivor space 1 capacity (KB).  S0U Survivor space 0 utilization (KB).  S1U Survivor space 1 utilization (KB).  TT Tenuring threshold.  MTT Maximum tenuring threshold.  DSS Desired survivor size (KB).  EC Current eden space capacity (KB).  EU Eden space utilization (KB).  YGC Number of young generation GC events.  YGCT Young generation garbage collection time. |
| -gcnewcapacity Option:New Generation Space Size Statistics  Column Description  NGCMN Minimum new generation capacity (KB).  NGCMX Maximum new generation capacity (KB).  NGC Current new generation capacity (KB).  S0CMX Maximum survivor space 0 capacity (KB).  S0C Current survivor space 0 capacity (KB).  S1CMX Maximum survivor space 1 capacity (KB).  S1C Current survivor space 1 capacity (KB).  ECMX Maximum eden space capacity (KB).  EC Current eden space capacity (KB).  YGC Number of young generation GC events.  FGC Number of Full GC Events. |
| -gcold Option:Old and Permanent Generation Statistics  Column Description  PC Current permanent space capacity (KB).  PU Permanent space utilization (KB).  OC Current old space capacity (KB).  OU old space utilization (KB).  YGC Number of young generation GC events.  FGC Number of full GC events.  FGCT Full garbage collection time.  GCT Total garbage collection time. |
| -gcoldcapacity Option:Old Generation Statistics  Column Description  OGCMN Minimum old generation capacity (KB).  OGCMX Maximum old generation capacity (KB).  OGC Current old generation capacity (KB).  OC Current old space capacity (KB).  YGC Number of young generation GC events.  FGC Number of full GC events.  FGCT Full garbage collection time.  GCT Total garbage collection time. |
| -gcpermcapacity Option: Permanent Generation Statistics  Column Description  PGCMN Minimum permanent generation capacity (KB).  PGCMX Maximum permanent generation capacity (KB).  PGC Current permanent generation capacity (KB).  PC Current permanent space capacity (KB).  YGC Number of young generation GC events.  FGC Number of full GC events.  FGCT Full garbage collection time.  GCT Total garbage collection time. |
| -gcutil Option:Summary of Garbage Collection Statistics  Column Description  S0 - Heap上的 Survivor space 0 区已使用空间的百分比  S1 - Heap上的 Survivor space 1 区已使用空间的百分比  E - Heap上的 Eden space 区已使用空间的百分比  O - Heap上的 Old space 区已使用空间的百分比  P - Perm space 区已使用空间的百分比  YGC - 从应用程序启动到采样时发生 Young GC 的次数  YGCT- 从应用程序启动到采样时 Young GC 所用的时间(单位秒)  FGC - 从应用程序启动到采样时发生 Full GC 的次数  FGCT- 从应用程序启动到采样时 Full GC 所用的时间(单位秒)  GCT - 从应用程序启动到采样时用于垃圾回收的总时间(单位秒) |

* 1. jinfo

实时的查看和调整虚拟机的各项参数

jinfo 命令格式：

jinfo [option] vmid

jinfo 主要的选项

* -flag <name> 显示JVM该属性的值
* -flag [+|-]<name> 给该JVM增加获取去除某属性参数
* -flag <name>=<value> 设置JVM某参数的值
* -flags 显示所有的参数
* -sysprops 显示系统参数
  1. jmap

生成堆转存快照文件。

jmap 命令格式：

jmap [option] vmid

jmap 主要选项

* -dump 生成Java堆转存快照，格式为:-dump:[live,]format=b,file=<filename>，其中live子参数说明是否只dump出存活的对象。
* -finalizerinfo 显示在F-Queue中等待finalizer线程执行finalize方法的对象。只在Linux/Solaris平台下有效。
* -heap 显示Java堆详细信息，如使用哪种回收器，参数配置、分代状况等。
* -histo 显示堆中对象的统计信息，包含类、实例数量和合计容量
* -permstat 以Classloader为统计口径显示永久代内存状态。
* -F 当虚拟机进程对-dump没有响应时，可以使用该选项强制生成dump快照

jmap实例

jmap –dump:format=b,file=c:/a.bin 3456

jmap –heap 3456

* 1. jhat

分析jmap生成的堆转储快照

实例 jhat c:/test.bin (当系统显示Server is ready，可以通过<http://localhost:7000>访问)

* 1. jstack

生成当前时刻线程快照

jstack 命令格式

jstack [option] vmid

jstack 主要选项

* -F 当输出的请求不被响应时，强制输出线程堆栈
* -l 除堆栈外，显示关于锁的附件信息
* -m 如果调用到本地方法的话，显示C/C==的堆栈
  1. jstatd

此命令是一个RMI Server应用程序，提供了对JVM的创建和结束监视，也为远程监视工具提供了一个可以attach的接口

jstatd 命令格式

jstatd [option] -J-Djava.security.policy=c:/jstatd.all.policy

jstat 主要选项

* -nr 当一个存在的RMI Registry没有找到时，不尝试创建一个内部的RMI Registry
* -p port 端口号，默认为1099
* -n rminame 默认为JStatRemoteHost；如果多个jstatd服务开始在同一台主机上，rminame唯一确定一个jstatd服务
* -J jvm选项

创建安全策略文件，命名为jstatd.all.policy

grant codebase "file:${java.home}/../lib/tools.jar" {

permission java.security.AllPermission;

};

* 1. jvisual/jconsole

直接运行

* 1. MemoryAnalyzer

eclipse网站下载

* 1. Btrace
     1. 实例讲解

原文: <http://rdc.taobao.com/team/jm/archives/509>

|  |
| --- |
| **import** java.util.Random;  **public** **class** Case1{    **public** **static** **void** main(String[] args) **throws** Exception{  Random random=**new** Random();  CaseObject object=**new** CaseObject();  **boolean** result=**true**;  **while**(result){  result=object.execute(random.nextInt(1000));  Thread.*sleep*(1000);  }  }    }  **public** **class** CaseObject{    **private** **static** **int** *sleepTotalTime*=0;    **public** **boolean** execute(**int** sleepTime) **throws** Exception{  System.*out*.println("sleep: "+sleepTime);  *sleepTotalTime*+=sleepTime;  Thread.*sleep*(sleepTime);  **return** **true**;  }    } |

1、调用此方法时传入的是什么参数，返回的是什么值，当时sleepTotalTime是多少？

BTrace脚本如下：

|  |
| --- |
| **import** **static** com.sun.btrace.BTraceUtils.\*;  **import** com.sun.btrace.annotations.\*;    @BTrace **public** **class** TraceMethodArgsAndReturn{  @OnMethod(  clazz="CaseObject",  method="execute",  location=@Location(Kind.RETURN)  )  **public** **static** **void** traceExecute(@Self CaseObject instance,**int** sleepTime,@Return **boolean** result){  println("call CaseObject.execute");  println(strcat("sleepTime is:",str(sleepTime)));  println(strcat("sleepTotalTime is:",str(get(field("CaseObject","sleepTotalTime"),instance))));  println(strcat("return value is:",str(result)));  }  } |

2、execute方法执行耗时是多久？

BTrace脚本如下：

|  |
| --- |
| **import** **static** com.sun.btrace.BTraceUtils.\*;  **import** com.sun.btrace.annotations.\*;    @BTrace **public** **class** TraceMethodExecuteTime{    @TLS **static** **long** *beginTime*;    @OnMethod(  clazz="CaseObject",  method="execute"  )  **public** **static** **void** traceExecuteBegin(){  *beginTime*=timeMillis();  }    @OnMethod(  clazz="CaseObject",  method="execute",  location=@Location(Kind.RETURN)  )  **public** **static** **void** traceExecute(**int** sleepTime,@Return **boolean** result){  println(strcat(strcat("CaseObject.execute time is:",str(timeMillis()-*beginTime*)),"ms"));  }  } |

3、谁调用了execute方法？

BTrace脚本如下：

|  |
| --- |
| **import** **static** com.sun.btrace.BTraceUtils.\*;  **import** com.sun.btrace.annotations.\*;    @BTrace **public** **class** TraceMethodCallee{  @OnMethod(  clazz="CaseObject",  method="execute"  )  **public** **static** **void** traceExecute(){  println("who call CaseObject.execute :");  jstack();  }  } |

4、有没有人调用CaseObject中的哪一行代码？

BTrace脚本如下：

|  |
| --- |
| **import** **static** com.sun.btrace.BTraceUtils.\*;  **import** com.sun.btrace.annotations.\*;    @BTrace **public** **class** TraceMethodLine{  @OnMethod(  clazz="CaseObject",  location=@Location(value=Kind.LINE,line=5)  )  **public** **static** **void** traceExecute(@ProbeClassName String pcn,@ProbeMethodName String pmn,**int** line){  println(strcat(strcat(strcat("call ",pcn),"."),pmn));  }  } |

从上面可看出，在有了BTrace后，要动态的跟踪代码的运行细节还是非常爽的，更多的细节的操作请大家查看BTrace的User Guide。http://kenai.com/projects/btrace/pages/UserGuide

1. TODO

Linux

1. 命令
   1. 文件处理命令

* ls

英文原意：list

所在路径：/bin/list

执行权限：所有用户

功能描述：显示目录文件

语法：ls 选项[-ald] [文件或目录]

-a (all)显示所有文件，包含隐藏文件

-l (long)详细信息显示

-d (directory)查看目录属性

|  |  |  |  |
| --- | --- | --- | --- |
| drwxr-xr-x 2 root root 4096 12-01 20:21  2 – 表示硬链接数 所有者 所属组 文件大小 创建时间或者最后修改时间  r – read w – write x – execute | | | |
| d | rwx | r-x | r-x |
| 文件类型 | 所有者u | 所属组g | 其他人o |
| d —directory目录  - -binary二进制  l –link软链接文件 | user  owner | group | others |
|  |  |  |  |

* cd

英文原意：chang directory

所在路径：shell内置命令

执行权限：所有用户

功能描述： 切换目录

语法：cd [目录]

范例： $cd / 返回根目录

$cd .. 返回上级目录

* pwd

英文原意:print work directory

所在路径:/bin/pwd

执行权限:所有用户

功能描述: 显示当前所在的工作目录

语法:pwd

范例:$pwd

* touch

英文原意:touch

所在路径:/bin/touch

执行权限:所有用户

功能描述: 创建空文件

语法:$touch [文件名]

范例:$touch newfile

* mkdir

英文原意:make directories

所在路径:/bin/mkdir

执行权限:所有用户

功能描述: 创建新目录

语法:mkdir [目录名]

范例:$mkdir newdirs

* cp

英文原意:copy

所在路径:/bin/cp

执行权限:所有用户

功能描述: 复制文件或者目录

语法:cp [-R] [源文件或者目录] [目标文件夹]

-R 复制目录

范例:$cp file1 file2 dir1 将文件file1,file2复制到dir1中

$cp –R dir1 dir2 将目录dir2复制到目录dir2下面

* mv

英文原意:move

所在路径:/bin/mv

执行权限:所有用户

功能描述: 移动文件、改名

语法:mv [源文件或者目录] [目标目录]

范例:$mv file1 file3 将当前目录下的file1改名file2

$mv file2 dir2 将file2移动到dir2目录下

* rm

英文原意:

所在路径:

执行权限:所有用户

功能描述:

语法:

范例:

* cc
* cc

英文原意:

所在路径:

执行权限:所有用户

功能描述:

语法:

范例:

* 1. 权限管理命令
  2. 文件搜索命令
  3. 帮助命令
  4. 压缩解压命令
  5. 网络通信命令
  6. 系统关机命令
  7. Shell应用技巧

1. TODO

guava

1. base
   1. Function

主要用于**转换**

* Function can be used to execute defined action (function) on many objects
  + Function用来在许多对象上执行已定义的功能。
* It is used to change collections into collections of different type
  + Function用来改变集合内对象的类型以形成一个不同的新集合
* Main usage with transform() method from classes Colllections2, Lists, Iterables, Iterators, etc.
  + 主要用于Collections\Lists\Iterables\Iterators等类的transform()方法
* Functions class provides a few pre-defined functions and allows to compose them into more complicated ones
  + Functions提供已定义的方法用于构成复杂的function
* samples

|  |
| --- |
| //Function  Function<Country,String> cityFunction = **new** Function<Country,String>(){  @Override  **public** String apply(@Nullable Country input) {  **if** ( input == **null** ) **return** "";  **return** input.city;  }};  List<Country> counties = Lists.*newArrayList*(**new** Country("北京"),**new** Country("东京"),**null**);  Collection<String> cities = Collections2.*transform*(counties, cityFunction);  System.*out*.println("cities:"+cities);    //Functions-复合function  Function<String,String> reversefunction = **new** Function<String, String>(){  **public** String apply(String input) {  **if** ( input == **null** )**return** **null**;  **return** **new** StringBuilder(input).reverse().toString();  }};    Function<Country, String> composefunction = Functions.*compose*(reversefunction, cityFunction);  Collection<String> reCities = Collections2.*transform*(counties, composefunction);  System.*out*.println("reCities:"+reCities);    //预定义function  Map<String,String> map = Maps.*newHashMap*();  map.put("中国", "北京");  map.put("日本", "东京");  map.put("韩国", "首尔");  map.put("美国", "华盛顿");  Function<String, String> preDefinedfunction = Functions.*forMap*(map);  List<String> list = Lists.*newArrayList*("中国","日本");  Collection<String> result = Collections2.*transform*(list, preDefinedfunction);  System.*out*.println("result:"+result);    //预定义function-默认值  Function<String, String> preDefinedfunction2 = Functions.*forMap*(map,"unkown");  List<String> list2 = Lists.*newArrayList*("中国","俄罗斯");  Collection<String> result2 = Collections2.*transform*(list2, preDefinedfunction2);  System.*out*.println("result2:"+result2); |

* 1. Functions
  2. Predicate

主要用于**检查、过滤**

* Predicate checks if given condition is met for passed object.
  + Predicate检查传递的对象是否满足给定的条件
* It is mainly used to filter collections using filter() method from Iterables, Iterators, Collections2, etc.
  + Predicate主要用于Iterables/Iterators/Collections等类的filter()方法用于过滤集合
* I can be also used to check if all elements from collections are satisfying defined rule - method Iterables.all()
  + Predicate也用于检查集合中的所有对象是否满足既定的规则-见Iterables.all()方法
* Predicates class provides a few utility methods to work use and allows to compose many Predicates in one
  + Predicates类提供一些工具方法使用以及构成复杂的Predicate
* Samples

|  |
| --- |
| //Predicate-检查  Predicate<Country> cityPredicate = **new** Predicate<Country>(){  **public** **boolean** apply(@Nullable Country country) {  **return** !Strings.*isNullOrEmpty*(country.city);  }};  **boolean** predicateCheck1 = Iterables.*all*(  Lists.*newArrayList*(**new** Country("北京"),**new** Country("东京"),**new** Country()), cityPredicate);  System.*out*.println("predicateCheck1:"+predicateCheck1);  //Predicates-过滤  Predicate<Country> sizePredicate = **new** Predicate<Country>(){  **public** **boolean** apply(@Nullable Country country) {  **return** country.city.length()<5;  }};  Predicate<Country> composetPredicate = Predicates.*and*(cityPredicate, sizePredicate);  Iterable<Country> iterable = Iterables.*filter*(Lists.*newArrayList*(**new** Country("北京"),**new** Country("东京东京东京"),**new** Country()), composetPredicate);  System.*out*.println("iterable:"+iterable);    Predicate<CharSequence> charPredicate = Predicates.*containsPattern*("\\d\\d");  System.*out*.println(charPredicate.apply("hello world 40")); |

* 1. Predicates
  2. CharMatcher
* It is a class similar to Predicate, but working with chars
  + CharMatcher与Predicate相似，但是它操作字符
* It allows to check if a sequence of characters satisfies given condition
  + 它检查字符串是否满足给定的条件
* We can also use it to filter chars
  + 也用于过滤字符
* Additionaly it provides methods to modify char sequences: removeFrom(), replaceFrom(), trimFrom(), collapseFrom(), retainFrom()
  + 额外的提供修改字符串的方法
* CharMatcher.retainFrom() – allows to keep only those chars that are satisfying defined matching rule
  + CharMatcher.retainFrom()-过滤不满足条件的字符
* CharMatcher.collapseFrom() – replaces group of chars satysfying matching rule with defined sequence of characters
  + CharMatcher.collapseFrom()-将满足条件的一组字符替换为定义字符
* CharMatcher.replaceFrom() - replaces every char satysfying matching rule with defined sequence of characters
  + CharMatcher.replaceFrom()-将满足条件的的每一个字符替换为定义字符
* samples
  1. Optional

判断引用对象是否为空，其实将应用的对象的null指针错误进行定位转换。

|  |
| --- |
| Optional.*fromNullable*("a").get();  Optional.*of*("bb").get(); |

* 1. aa

1. cache
   1. aa
   2. bb
2. collect
   1. Tools
      1. Lists
      2. Sets
      3. Maps
      4. Iterables
      5. Iterators
      6. Multimaps
      7. Multisets
      8. ObjectArrays
      9. Queues
      10. Ranges
      11. Tables
   2. Immutable

真正的不可修改的集合

|  |
| --- |
| ImmutableSet<String> immutableSet1 = ImmutableSet.*of*("RED", "GREEN");  ImmutableSet<Object> immutableSet2 = ImmutableSet.*copyOf*(Sets.*newHashSet*());  Builder<String> builder = ImmutableSet.*builder*();  ImmutableSet<String> immutableSet3 = builder.add("RED").addAll(Sets.*newHashSet*("GREEN")).build();  ImmutableSet.*of*();  ImmutableList.*of*();  ImmutableMap.*of*();  ImmutableBiMap.*of*();  ImmutableMultimap.*of*();  ImmutableMultiset.*of*();  ImmutableSortedSet.*of*();  ImmutableSortedMap.*of*();  ImmutableSortedMultiset.*of*();  ImmutableTable.*of*(); |

* + 1. ImmutableSet
    2. ImmutableList
    3. ImmutableMap
    4. ImmutableBiMap
    5. ImmutableMultimap
    6. ImmutableMultiset
    7. ImmutableSortedSet
    8. ImmutableSortedMap
    9. ImmutableSortedMultiset
    10. ImmutableTable
  1. Multiset

把重复的元素放入集合

|  |
| --- |
| Multiset<String> multiset = HashMultiset.*create*();  multiset.add("one");  multiset.add("one");  multiset.count("one"); |

* + 1. HashMultiset
    2. LinkedHashMultiset
    3. TreeMultiset
    4. EnumMultiset
    5. ImmutableMultiset
  1. Multimap

在 Map 的 value 里面放多个元素

* + 1. HashMultimap
    2. ArrayListMultimap
    3. LinkedHashMultimap
    4. TreeMultimap
    5. ImmutableMultimap
  1. BiMap

双向 Map,特点是它的 value 和它 key 一样也是不可重复的,inverse 方法就是把 BiMap 的 key 集合 value 集合对调，因此 biMap == biMap.inverse().inverse()。

* + 1. HashBiMap
    2. EnumBiMap
    3. ImmutableBiMap
  1. ComparisonChain

实例

|  |
| --- |
| Collections.*sort*(**null**, **new** Comparator<Foo>(){  **public** **int** compare(Foo f1, Foo f2) {  **return** ComparisonChain.*start*()  .compare(f1.a, f2.a)  .compare(f1.b, f2.b)  .compare(f1.c, f2.c).result();  }}); |

* 1. Ordering
  2. Collections2
  3. aa
  4. aaa
  5. aaa

1. eventbus
   1. a
2. hash
   1. a
3. io
   1. a
4. math
   1. a
5. net
   1. a
6. primitives
   1. a
7. reflect
   1. a
8. concurrent
   1. a
9. TODO

Maven

1. 注意事项

* **MAVEN\_OPTS**:-Xms128m –Xmx512m
* **GAV**: groupId、artifactId、version
* **dependency-scope**:依赖范围 **compile**(spring-core),**test**(junit),**provided**(servlet-api),  
  **runtime**(jdbc12.jar),**system**(rt.jar),import
* dependency-classifier:附属构建,比如jdk1.5,javadoc,source
* mvn **clear compile test package install deploy site**
* mvn clear install-U(强制检查更新,比如用户A更新快照, 用户B依赖快照,可以强制更新获得最新快照)
* mvn  dependency:sources ***source:jar*** install –Dmaven.test.skip=true
* mvn clean package -Dmaven.test.skip=false **-U –e –Ptest** 采用testProfile配置，强制更新
* mvn jetty:run –Djetty.port=9999
* mvn dependency:list dependency:tree dependency:analyze
* 归类依赖: <springframework.version>3.2.3.RELEASE</springframework.version>
* default生命周期: validate initialize generate-sources process-sources generate-resources process-reources compile process-classes generate-test-sources process-test-sources generate-test-reources process-test-reources test-compile process-test-classes test prepare-package package pre-integration-test integration-test post-integration-test verity intall deploy
* 继承与聚合 pom.xml 的 <packaging>pom</packaging>
* 构建指定模块 mvn clean install –pl account-email,account-persist
* 构建指定模块及其依赖模块 mvn clean install –pl account-email –am
* 构建指定模块及其父模块与依赖父模块的所有模块 mvn clean install –pl account-email –amd
* 指定优先构建 mvn clean install –rf account-emaill
* 总和 mvn clean install –pl account-parent –amd –rf account-email
* 版本说明:1.3.4-beta-2 <主版本>.<次版本>.<增量版本>-<里程碑版本>
* MAVEN 与 DLL处理
  + 方法一：上传DLL到私服，然后添加到依赖,package时会自动将dll下载到lib目录,但是无法eclipse实时调试

|  |
| --- |
| <dependency>  <groupId>org.hyperic</groupId>  <artifactId>sigar-amd64-winnt</artifactId>  <version>1.6.4</version>  <type>dll</type>  <scope>runtime</scope>  </dependency> |

* + 方法二:将dll放置某dlls目录,eclipse调试时 启动方法JVM参数添加-Djava.library.path=${workspace\_loc:drm-agent}/src/main/resources/dll,参考<http://chemingliang.iteye.com/blog/1708645>
  + 方法二:MAVEN测试时，修改test插件

|  |
| --- |
| <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <configuration>  <forkMode>once</forkMode>  <workingDirectory>target</workingDirectory>  <argLine>-Djava.library.path=${basedir}/src/main/resources/lib</argLine>  </configuration>  </plugin> |

* MAVEN与WTP集成，**强烈建议不使用maven自带的WTP集成插件**
  + eclipse-jee-indigo-SR2-win32原版安装m2e1.4 缺少plugins/org.slf4j.api\_1.6.4.v20120130-2120.jar
  + 将项目修改为Project Facets，Properties->Project Facets,Convert to faceted form…选Dynamic Web Module,Vesion选择2.5
  + 修改.classpath文件, **<classpathentry** kind="con" path="org.maven.ide.eclipse.MAVEN2\_CLASSPATH\_CONTAINER"**/>**添加属性

|  |
| --- |
| **<attributes>**  **<attribute name="org.eclipse.jst.component.dependency" value="/WEB-INF/lib"/>**  **</attributes>** |

* 打war包(先打jar包,然后打war包)

|  |
| --- |
| <!-- jar包拷贝 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-jar-plugin</artifactId>  <version>2.3.2</version>  <configuration>  <encoding>UTF-8</encoding>  </configuration>  <executions>  <execution>  <phase>**prepare-package**</phase>  <goals>  <goal>jar</goal>  </goals>  <configuration>  <classesDirectory>target/classes</classesDirectory>  <finalName>${project.artifactId}-${project.version}</finalName> <outputDirectory>target/${project.artifactId}-${project.version}/WEB-INF/lib</outputDirectory>  <excludes>  <exclude>config/\*\*</exclude>  </excludes>  </configuration>  </execution>  </executions>  </plugin>  <!-- 打war包 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-war-plugin</artifactId>  <version>2.4</version>  <configuration>  <packagingExcludes>  WEB-INF/classes/com/\*\*,  WEB-INF/classes/remote/\*\*,  WEB-INF/classes/spring/\*\*,  WEB-INF/classes/sql/\*\*  </packagingExcludes>  </configuration>  </plugin> |

1. POM属性

* ${basedir}：项目根目录
* ${project.groupId}:项目的groupId
* ${project.artifactId}:项目的artifactId
* ${project.version}:项目的version
* ${project.build.sourceDirectory}:源码目录,默认src/main/java
* ${project.build.testSourceDirectory}:测试源码目录,默认src/test/java
* ${project.build.directory}:构建输出目录,默认target/
* ${project.outputDirectory}:主代码编译输出目录,默认target/classes
* ${project.testOutputDirectory}:测试代码编译输出目录,默认target/test-classes/
* ${project.build.finalName}:打包输出文件默认名,默认${project.artifactId}-${project.version}

1. 常用插件
   1. compiler

|  |
| --- |
| <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <configuration>  <source>1.6</source>  <target>1.6</target>  </configuration>  </plugin> |

* 1. shade

自定义jar启动类

|  |
| --- |
| <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-shade-plugin</artifactId>  <version>2.1</version>  <executions>  <execution>  <phase>package</phase>  <goals>  <goal>shade</goal>  </goals>  <configuration>  <transformers>  <transformer implementation=*"org.apache.maven.plugins.shade.resource.ManifestResourceTransformer"*>  <mainClass>com.juvenxu.mvnbook.helloworld.HelloWorld</mainClass>  </transformer>  </transformers>  </configuration>  </execution>  </executions>  </plugin> |

* 1. jetty

|  |
| --- |
| <plugin>  <groupId>org.mortbay.jetty</groupId>  <artifactId>jetty-maven-plugin</artifactId>  <version>8.1.11.v20130520</version>  <configuration>  <scanIntervalSeconds>10</scanIntervalSeconds>  <webAppConfig>  <contextPath>/mvntestweb</contextPath>  </webAppConfig>  </configuration>  </plugin> |

mvn jetty:run

* 1. TODO

1. 配置说明
   1. repository

|  |
| --- |
| <!-- 私服Nexus -->  <repositories>  <repository>  <id>localhost-repository</id>  <url>http://localhost:8081/nexus/content/groups/public/</url>  <releases>  <enabled>true</enabled>  </releases>  <snapshots>  <enabled>true</enabled>  </snapshots>  </repository>  </repositories>    <pluginRepositories>  <pluginRepository>  <id>localhost-pluginRepository</id>  <name>localhost-pluginRepository</name>  <url>http://localhost:8081/nexus/content/groups/public/</url>  <releases><enabled>true</enabled></releases>  <snapshots><enabled>true</enabled></snapshots>  </pluginRepository>  </pluginRepositories> |

* 1. distributionManagement

|  |
| --- |
| <!-- 远程部署 mvn deploy -->  <repository>  <id>local-releases</id>  <name>local release</name>  <url>http://localhost:8081/nexus/content/repositories/releases/</url>  </repository>  <snapshotRepository>  <id>local-snapshots</id>  <name>local snapshot</name>  <url>http://localhost:8081/nexus/content/repositories/snapshots/</url>  </snapshotRepository> |

修改setttings.xml

|  |
| --- |
| <server>  <id>local-releases</id>  <username>admin</username>  <password>admin123</password>  </server>  <server>  <id>local-snapshots</id>  <username>admin</username>  <password>admin123</password>  </server> |

* 1. dependencyManagement

pom.xml(parent)

|  |
| --- |
| <dependencyManagement>  <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.7</version>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>${springframework.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aop</artifactId>  <version>${springframework.version}</version>  </dependency>  </dependencies>  </dependencyManagement> |

pom.xml(child)

|  |
| --- |
| <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  </dependency>  </dependencies> |

* 1. TODO

1. 实例
   1. 打war包前先jar处理

|  |
| --- |
| <!-- jar包拷贝 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-jar-plugin</artifactId>  <version>2.3.2</version>  <configuration>  <encoding>UTF-8</encoding>  </configuration>  <executions>  <execution>  <phase>**prepare-package**</phase>  <goals>  <goal>jar</goal>  </goals>  <configuration>  <classesDirectory>target/classes</classesDirectory>  <finalName>${project.artifactId}-${project.version}</finalName>  <outputDirectory>target/drm/WEB-INF/lib</outputDirectory>  <excludes>  <exclude>config/\*\*</exclude>  </excludes>  </configuration>  </execution>  </executions>  </plugin>  <!-- 打war包 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-war-plugin</artifactId>  <version>2.4</version>  <configuration>  <packagingExcludes>  WEB-INF/classes/com/\*\*,  WEB-INF/classes/remote/\*\*,  WEB-INF/classes/spring/\*\*,  WEB-INF/classes/sql/\*\*  </packagingExcludes>  </configuration>  </plugin> |

* 1. 编译时拷贝ibatis的xml文件

|  |
| --- |
| <resources>  <resource>  <directory>src/main/resources</directory>  </resource>  <resource>  <directory>src/main/java</directory>  <includes>  <include>\*\*/\*.xml</include>  </includes>  </resource>  </resources> |

* 1. assembly打包

java的CS应用，先打jar包,拷贝DLL文件,然后打tar包

|  |
| --- |
| <!-- 打jar包 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-shade-plugin</artifactId>  <version>2.1</version>  <executions>  <execution>  <phase>package</phase>  <goals>  <goal>shade</goal>  </goals>  <configuration>  <artifactSet>  <includes>  <include>${project.groupId}:${project.artifactId}</include>  </includes>  </artifactSet>  <filters>  <filter>  <artifact>${project.groupId}:${project.artifactId}</artifact>  <excludes>  <exclude>dll/\*\*</exclude>  <exclude>config/\*\*</exclude>  </excludes>  </filter>  </filters>  </configuration>  </execution>  </executions>  </plugin>  <!-- 打tar包 -->  <plugin>  <artifactId>maven-assembly-plugin</artifactId>  <configuration>  <descriptor>src/main/assembly/assembly.xml</descriptor>  </configuration>  <executions>  <execution>  <id>make-assembly</id>  <phase>package</phase>  <goals>  <goal>single</goal>  </goals>  </execution>  </executions>  </plugin> |
| assembly.xml文件 |
| <assembly>  <id>assembly</id>  <formats>  <format>tar.gz</format>  </formats>  <includeBaseDirectory>true</includeBaseDirectory>  <fileSets>  <fileSet>  <directory>src/main/scripts</directory>  <outputDirectory>bin</outputDirectory>  <fileMode>0755</fileMode>  </fileSet>  <fileSet>  <directory>src/main/resources/config</directory>  <outputDirectory>conf/config</outputDirectory>  <fileMode>0644</fileMode>  </fileSet>  <fileSet>  <directory>src/main/resources/dll</directory>  <outputDirectory>lib</outputDirectory>  <fileMode>0755</fileMode>  </fileSet>  </fileSets>  <dependencySets>  <dependencySet>  <outputDirectory>lib</outputDirectory>  </dependencySet>  </dependencySets>  </assembly> |

* 1. TODO

Redis

1. 配置说明

* daemonize : 默认情况下，redis不是在后台运行的，如果需要在后台运行，把该项的值更改为yes
* pidfile: 当Redis在后台运行的时候，Redis默认会把pid文件放在/var/run/redis.pid，你可以配置到其他地址。当运行多个redis服务时，需要指定不同的pid文件和端口
* bind: 指定Redis只接收来自于该IP地址的请求，如果不进行设置，那么将处理所有请求，在生产环境中最好设置该项
* port: 监听端口，默认为6379
* timeout: 设置客户端连接时的超时时间，单位为秒。当客户端在这段时间内没有发出任何指令，那么关闭该连接
* loglevel: log 等级分为4级，debug, verbose, notice, 和warning。生产环境下一般开启notice
* logfile: 配置log 文件地址，默认使用标准输出，即打印在命令行终端的窗口上
* databases: 设置数据库的个数，可以使用SELECT <dbid>命令来切换数据库。默认使用的数据库是0
* save: 设置Redis进行数据库镜像的频率。if(在60秒之内有10000个keys发生变化时){ 进行镜像备份}else if(在300秒之内有10个keys发生了变化){ 进行镜像备份}else if(在900秒之内有1个keys发生了变化){ 进行镜像备份}
* rdbcompression: 在进行镜像备份时，是否进行压缩
* dbfilename: 镜像备份文件的文件名
* dir: 数据库镜像备份的文件放置的路径。这里的路径跟文件名要分开配置是因为Redis在进行备份时，先会将当前数据库的状态写入到一个临时文件中，等备份完成时，再把该该临时文件替换为上面所指定的文件，而这里的临时文件和上面所配置的备份文件都会放在这个指定的路径当中
* slaveof: 设置该数据库为其他数据库的从数据库
* masterauth: 当主数据库连接需要密码验证时，在这里指定
* requirepass: 设置客户端连接后进行任何其他指定前需要使用的密码。警告：因为redis速度相当快，所以在一台比较好的服务器下，一个外部的用户可以在一秒钟进行150K次的密码尝试，这意味着你需要指定非常非常强大的密码来防止暴力破解。
* maxclients: 限制同时连接的客户数量。当连接数超过这个值时，redis 将不再接收其他连接请求，客户端尝试连接时将收到error信息。
* maxmemory: 设置redis 能够使用的最大内存。当内存满了的时候，如果还接收到set命令，redis 将先尝试剔除设置过expire信息的key，而不管该key的过期时间还没有到达。在删除时，将按照过期时间进行删除，最早将要被过期的key将最先被删除。如果带有expire信息的key都删光了，那么将返回错误。这样，redis将不再接收写请求，只接收get请求。maxmemory的设置比较适合于把redis当作于类似memcached的缓存来使用。
* appendonly：默认情况下，redis 会在后台异步的把数据库镜像备份到磁盘，但是该备份是非常耗时的，而且备份也不能很频繁，如果发生诸如拉闸限电、拔插头等状况，那么将造成比较大范围的数据丢失。所以redis提供了另外一种更加高效的数据库备份及灾难恢复方式。开启append only 模式之后，redis 会把所接收到的每一次写操作请求都追加到appendonly.aof文件中，当redis重新启动时，会从该文件恢复出之前的状态。但是这样会造成appendonly.aof 文件过大，所以redis 还支持了BGREWRITEAOF 指令，对appendonly.aof 进行重新整理。所以我认为推荐生产环境下的做法为关闭镜像，开启appendonly.aof，同时可以选择在访问较少的时间每天对appendonly.aof进行重写一次。
* appendfsync：设置对appendonly.aof 文件进行同步的频率。always 表示每次有写操作都进行同步，everysec表示对写操作进行累积，每秒同步一次。这个需要根据实际业务场景进行配置
* vm-enabled：是否开启虚拟内存支持。因为redis 是一个内存数据库，而且当内存满的时候，无法接收新的写请求，所以在redis 2.0中，提供了虚拟内存的支持。但是需要注意的是，redis中，所有的key都会放在内存中，在内存不够时，只会把value值放入交换区。这样保证了虽然使用虚拟内存，但性能基本不受影响，同时，你需要注意的是你要把vm-max-memory设置到足够来放下你的所有的key
* vm-swap-file：设置虚拟内存的交换文件路径
* vm-max-memory：这里设置开启虚拟内存之后，redis 将使用的最大物理内存的大小。默认为0，redis 将把他所有的能放到交换文件的都放到交换文件中，以尽量少的使用物理内存。在生产环境下，需要根据实际情况设置该值，最好不要使用默认的0
* vm-page-size：设置虚拟内存的页大小，如果你的value值比较大，比如说你要在value中放置博客、新闻之类的所有文章内容，就设大一点，如果要放置的都是很小的内容，那就设小一点。
* vm-pages：设置交换文件的总的page数量，需要注意的是，page table信息会放在物理内存中，每8个page就会占据RAM中的1个byte。总的虚拟内存大小 ＝ vm-page-size \* vm-pages
* vm-max-threads：设置VM IO同时使用的线程数量。因为在进行内存交换时，对数据有编码和解码的过程，所以尽管IO 设备在硬件上本上不能支持很多的并发读写，但是还是如果你所保存的vlaue值比较大，将该值设大一些，还是能够提升性能的
* glueoutputbuf：把小的输出缓存放在一起，以便能够在一个TCP packet中为客户端发送多个响应，具体原理和真实效果我不是很清楚。所以根据注释，你不是很确定的时候就设置成yes
* hash-max-zipmap-entries：在redis 2.0中引入了hash数据结构。当hash中包含超过指定元素个数并且最大的元素没有超过临界时，hash将以一种特殊的编码方式（大大减少内存使用）来存储，这里可以设置这两个临界值
* activerehashing：开启之后，redis将在每100毫秒时使用1毫秒的CPU时间来对redis的hash表进行重新hash，可以降低内存的使用。当你的使用场景中，有非常严格的实时性需要，不能够接受Redis时不时的对请求有2毫秒的延迟的话，把这项配置为no。如果没有这么严格的实时性要求，可以设置为yes，以便能够尽可能快的释放内存
* hash-max-zipmap-entries 64 ：#配置字段最多64个(如果field 或者value的大小超出一定限制后，Redis会在内部自动将zipmap替换成正常的hash实现.)
* hash-max-zipmap-value 512 : #配置value最大为512字节

1. 类型
   1. strings

string类型是二进制安全的。意思是redis的string可以包含任何数据，比如jpg 图片或者序列化的对象。从内部实现来看其实string可以看作byte数组，最大上限是1G字节

* 1. hashes

Redis hash是一个string类型的field 和value的映射表.它的添加、删除操作都是O(1)（平均）。hash特别适合用于存储对象。相较于将对象的每个字段存成单个string类型。将一个对象存储在hash类型中会占用更少的内存，并且可以更方便的存取整个对象。省内存的原因是新建一个hash对象时开始是用zipmap（又称为small hash）来存储的。这个zipmap其实并不是hash table，但是zipmap相比正常的hash实现可以节省不少hash本身需要的一些元数据存储开销。尽管zipmap的添加，删除，查找都是O(n)，但是由于一般对象的field 数量都不太多。所以使用zipmap也是很快的,也就是说添加删除平均还是O(1)。如果field 或者value的大小超出一定限制后，Redis会在内部自动将zipmap替换成正常的hash实现. 这个限制可以在配置文件中指定 hash-max-zipmap-entries 64 #配置字段最多64个 hash-max-zipmap-value 512 #配置value最大为512字节

* 1. lists

Redis的list 类型其实就是一个每个子元素都是string类型的双向链表。链表的最大长度是(2的32次方)。我们可以通过push,pop操作从链表的头部或者尾部添加删除元素。这使得list既可以用作栈，也可以用作队列。

* 1. sets

set的是通过hash table实现的，所以添加、删除和查找的复杂度都是O(1)。hash table会随着添加或者删除自动的调整大小。需要注意的是调整hash table大小时候需要同步（获取写锁）会阻塞其他读写操作，可能不久后就会改用跳表（skip list）来实现，跳表已经在sorted set 中使用了。关于set 集合类型除了基本的添加删除操作，其他有用的操作还包含集合的取并集(union)，交集(intersection)，差集(difference)。通过这些操作可以很容易的实现sns中的好友推荐和blog的tag功能。

* 1. sort sets

sorted set是set的一个升级版本，它在set的基础上增加了一个顺序属性，这一属性在添加修改元素的时候可以指定，每次指定后，zset会自动重新按新的值调整顺序。可以理解为有两列的mysql表，一列存value，一列存顺序。操作中key理解为zset的名字。和set一样sorted set也是string类型元素的集合，不同的是每个元素都会关联一个double类型的score。sorted set的实现是skip list和hash table的混合体。当元素被添加到集合中时，一个元素到score的映射被添加到hash table中，所以给定一个元素获取score的开销是O(1),另一个score到元素的映射被添加到skip list，并按照score排序，所以就可以有序的获取集合中的元素。添加，删除操作开销都是O(log(N))和skip list的开销一致,redis 的skip list实现用的是双向链表,这样就可以逆序从尾部取元素。sorted set最经常的使用方式应该是作为索引来使用.我们可以把要排序的字段作为score存储，对象的id当元素存储。

* 1. todo

1. 高级特性
   1. 主从复制

通过主从复制可以允许多个slave server 拥有和master server相同的数据库副本。

Redis的复制功能是完全建立在基于内存快照的持久化策略基础上的，也就是说无论你的持久化策略选择的是什么，只要用到了Redis的复制功能，就一定会有内存快照发生，**那么首先要注意你的系统内存容量规划——物理内存使用量不要超过3/5。**

* + 1. 复制特点
* master可以拥有多个slave
* 多个slave可以连接同一个master外，还可以连接到其他slave
* 主从复制不会阻塞master，在同步数据时，master可以继续处理client请求
* 提高系统的伸缩性
  + 1. 复制过程

当配置好slave后，slave与master建立连接，然后发送sync命令。无论是第一次连接还是重新连接，master都会启动一个后台进程，将数据库快照保存到文件中，同时master主进程会开始收集新的写命令并缓存。后台进程完成写文件后，master就发送文件给slave，slave将文件保存到硬盘上，再加载到内存中，接着master就会把缓存的命令转发给slave，后续master将收到的写命令发送给slave。如果master同时收到多个slave发来的同步连接命令，master只会启动一个进程来写数据库镜像，然后发送给所有的slave。

* 1. 事务控制

redis 对事务的支持目前还比较简单。redis 只能保证一个client发起的事务中的命令可以连续的执行，而中间不会插入其他client的命令。 由于redis是单线程来处理所有client的请求的所以做到这点是很容易的。一般情况下redis 在接受到一个client发来的命令后会立即处理并 返回处理结果，但是当一个client在一个连接中发出multi命令有，这个连接会进入一个事务上下文，该连接后续的命令并不是立即执行，而是先放到一个队列中。当从此连接受到exec命令后，redis会顺序的执行队列中的所有命令。并将所有命令的运行结果打包到一起返回给client.然后此连接就 结束事务上下文。

* 1. 持久化机制

redis支持两种持久化方式，一种是Snapshotting（快照）也是默认方式，另一种是Append-only file（缩写aof）的方式。

redis的持久化使用了buffer io

Redis对持久化文件的写入和读取操作都会使用物理内存的Page Cache,而大多数数据库系统会使用Direct IO来绕过这层Page Cache并自行维护一个数据的Cache，而当Redis的持久化文件过大(尤其是快照文件)，并对其进行读写时，磁盘文件中的数据都会被加载到物理内存中作为操作系统对该文件的一层Cache,而这层Cache的数据与Redis内存中管理的数据实际是重复存储的，虽然内核在物理内存紧张时会做Page Cache的剔除工作，但内核很可能认为某块Page Cache更重要，而让你的进程开始Swap ,这时你的系统就会开始出现不稳定或者崩溃了

经验：**当Redis物理内存使用超过内存总容量的3/5时就会开始比较危险了**

由于fork调用的内存会采用操作系统的COW机制，所以内存不会马上加倍，只有在最坏的情况下，也就是在dump的时候主进程中的数据完全被修改了才COPY出两倍的内存来



当Jedis服务器挂掉时,重启时将按以下优先级恢复数据到内存中:

1. 如果只配置了AOF,重启时加载AOF文件恢复数据.

2.如果同时配置了RBD和AOF,启动时只加载AOF文件恢复数据.

3.如果只配置了RBD,启动时将加载dump文件恢复数据.

* + 1. snapshotting方式

快照是默认的持久化方式。这种方式是就是将内存中数据以快照的方式写入到二进制文件中,默认的文件名为dump.rdb。可以通过配置设置自动做快照持久化的方式。我们可以配置redis在n秒内如果超过m个key被修改就自动做快照

快照保存过程：

redis调用fork,现在有了子进程和父进程。

父进程继续处理client请求，子进程负责将内存内容写入到临时文件。由于os的实时复制机制（copy on write)父子进程会共享相同的物理页面，当父进程处理写请求时os会为父进程要修改的页面创建副本，而不是写共享的页面。所以子进程地址空间内的数据是fork时刻整个数据库的一个快照。

当子进程将快照写入临时文件完毕后，用临时文件替换原来的快照文件，然后子进程退出。

client 也可以使用save或者bgsave命令通知redis做一次快照持久化。save操作是在主线程中保存快照的，由于redis 是用一个主线程来处理所有client的请求，这种方式会阻塞所有client请求。所以不推荐使用。另一点需要注意的是，每次快照持久化都是将内存数据完整写入到磁盘一次，并不是增量的只同步变更数据。如果数据量大的话，而且写操作比较多，必然会引起大量的磁盘io 操作，可能会严重影响性能。

* + 1. aop方式

由于快照方式是在一定间隔时间做一次的，所以如果redis 意外down掉的话，就会丢失最后一次快照后的所有修改。如果应用要求不能丢失任何修改的话，可以采用aof持久化方式。

aof 比快照方式有更好的持久化性，是由于在使用aof持久化方式时,redis 会将每一个收到的写命令都通过write函数追加到文件中(默认是appendonly.aof)。当redis 重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。当然由于os会在内核中缓存 write 做的修改，所以可能不是立即写到磁盘上。这样aof方式的持久化也还是有可能会丢失部分修改。不过我们可以通过配置文件告诉redis我们想要通过fsync 函数强制os写入到磁盘的时机。有三种方式如下（默认是：每秒fsync 一次）

appendonly yes //启用aof持久化方式

# appendfsync always //收到写命令就立即写入磁盘，最慢，但是保证完全的持久化

appendfsync everysec //每秒钟写入磁盘一次，在性能和持久化方面做了很好的折中

# appendfsync no //完全依赖os，性能最好,持久化没保证

为了压缩aof的持久化文件。redis提供了bgrewriteaof命令。收到此命令redis 将使用与快照类似的方式将内存中的数据以命令的方式保存到临时文件中，最后替换原来的文件。具体过程如下：

1、redis调用fork ，现在有父子两个进程

2、子进程根据内存中的数据库快照，往临时文件中写入重建数据库状态的命令

3、父进程继续处理client请求，除了把写命令写入到原来的aof文件中。同时把收到的写命令缓存起来。这样就能保证如果子进程重写失败的话并不会出问题。

4、当子进程把快照内容写入已命令方式写到临时文件中后，子进程发信号通知父进程。然后父进程把缓存的写命令也写入到临时文件。

5、现在父进程可以使用临时文件替换老的aof文件，并重命名，后面收到的写命令也开始往新的aof文件中追加。

需要注意到是重写aof文件的操作，并没有读取旧的aof文件，而是将整个内存中的数据库内容用命令的方式重写了一个新的aof文件,这点和快照有点类似。

* 1. 虚拟内存

redis 没有使用操作系统提供的虚拟内存机制而是自己在实现了自己的虚拟内存机制，主要的理由有两点:

1、操作系统的虚拟内存是已4k页面为最小单位进行交换的。而redis的大多数对象都远小于4k，所以一个操作系统页面上可能有多个redis对象。另外redis的集合对象类型如list,set可能存在与多个操作系统页面上。最终可能造成只有10%key被经常访问，但是所有操作系统页面都会被操作系统认为是活跃的，这样只有内存真正耗尽时操作系统才会交换页面。

2、相比于操作系统的交换方式，redis可以将被交换到磁盘的对象进行压缩,保存到磁盘的对象可以去除指针和对象元数据信息，一般压缩后的对象会比内存中的对象小10倍，这样redis的虚拟内存会比操作系统虚拟内存能少做很多io 操作。

下面是vm相关配置

vm-enabled yes #开启vm功能

vm-swap-file /tmp/redis.swap #交换出来的value保存的文件路径

vm-max-memory 1000000 #redis使用的最大内存上限

vm-page-size 32 #每个页面的大小32个字节

vm-pages 134217728 #最多使用多少页面

vm-max-threads 4 #用于执行value对象换入换出的工作线程数量

jedis的虚拟内存在设计上为了保证key的查找速度，只会将value交换到swap文件中。所以如果是内存问题是由于太多value很小的key造成的，那么虚拟内存并不能解决，和操作系统一样redis也是按页面来交换对象的。redis规定同一个页面只能保存一个对象。但是一个对象可以保存在多个页面中。在redis使用的内存没超过vm-max-memory之前是不会交换任何value的。当超过最大内存限制后，redis会选择较过期的对象。如果两个对象一样过期会优先交换比较大的对象，精确的公式swappability = age\*log(size\_in\_memory)。对于vm-page-size的设置应该根据自己的应用将页面的大小设置为可以容纳大多数对象的大小，太大了会浪费磁盘空间，太小了会造成交换文件出现碎片。对于交换文件中的每个页面，redis会在内存中对应一个1bit值来记录页面的空闲状态。所以像上面配置中页面数量(vm-pages 134217728 )会占用16M内存用来记录页面空闲状态。

1. 相关资料
   1. 内部实现
   2. 应用优化与运维
      1. Redis 常见的性能问题和解决方法

* Master写内存快照

save命令调度rdbSave函数，会阻塞主线程的工作，当快照比较大时对性能影响是非常大的，会间断性暂停服务，所以Master最好不要写内存快照。

* Master AOF持久化

如果不重写AOF文件，这个持久化方式对性能的影响是最小的，但是AOF文件会不断增大，AOF文件过大会影响Master重启的恢复速度。

* Master调用BGREWRITEAOF

Master调用BGREWRITEAOF重写AOF文件，AOF在重写的时候会占大量的CPU和内存资源，导致服务load过高，出现短暂服务暂停现象。

下面是我的一个实际项目的情况，大概情况是这样的：一个Master，4个Slave，没有Sharding机制，仅是读写分离，Master负责写入操作和AOF日志备份，AOF文件大概5G，Slave负责读操作，当Master调用BGREWRITEAOF时，Master和Slave负载会突然陡增，Master的写入请求基本上都不响应了，持续了大概5分钟，Slave的读请求过也半无法及时响应，上面的情况本来不会也不应该发生的，是因为以前Master的这个机器是Slave，在上面有一个shell定时任务在每天的上午10点调用BGREWRITEAOF重写AOF文件，后来由于Master机器down了，就把备份的这个Slave切成Master了，但是这个定时任务忘记删除了，就导致了上面悲剧情况的发生，原因还是找了几天才找到的。

将no-appendfsync-on-rewrite的配置设为yes可以缓解这个问题，设置为yes表示rewrite期间对新写操作不fsync，暂时存在内存中，等rewrite完成后再写入。最好是不开启Master的AOF备份功能。

* Redis主从复制的性能问题

第一次Slave向Master同步的实现是：Slave向Master发出同步请求，Master先dump出rdb文件，然后将rdb文件全量传输给slave，然后Master把缓存的命令转发给Slave，初次同步完成。第二次以及以后的同步实现是：Master将变量的快照直接实时依次发送给各个Slave。不管什么原因导致Slave和Master断开重连都会重复以上过程。Redis的主从复制是建立在内存快照的持久化基础上，只要有Slave就一定会有内存快照发生。虽然Redis宣称主从复制无阻塞，但由于Redis使用单线程服务，如果Master快照文件比较大，那么第一次全量传输会耗费比较长时间，且文件传输过程中Master可能无法提供服务，也就是说服务会中断，对于关键服务，这个后果也是很可怕的。

以上1.2.3.4根本问题的原因都离不开系统io瓶颈问题，也就是硬盘读写速度不够快，主进程 fsync()/write() 操作被阻塞。

* 单点故障问题

由于目前Redis的主从复制还不够成熟，所以存在明显的单点故障问题，这个目前只能自己做方案解决，如：主动复制，Proxy实现Slave对Master的替换等，这个也是Redis作者目前比较优先的任务之一，作者的解决方案思路简单优雅，详情可见 Redis Sentinel design draft <http://redis.io/topics/sentinel-spec>。

* 总结
  + Master最好不要做任何持久化工作，包括内存快照和AOF日志文件，特别是不要启用内存快照做持久化。
  + 如果数据比较关键，某个Slave开启AOF备份数据，策略为每秒同步一次。
  + 为了主从复制的速度和连接的稳定性，Slave和Master最好在同一个局域网内。
  + 尽量避免在压力较大的主库上增加从库
  + 为了Master的稳定性，主从复制不要用图状结构，用单向链表结构更稳定，即主从关系为：Master<–Slave1<–Slave2<–Slave3…….，这样的结构也方便解决单点故障问题，实现Slave对Master的替换，也即，如果Master挂了，可以立马启用Slave1做Master，其他不变。
    1. 分布式存储管理

Redis分布式存储管理软件有，代理实现的Twemproxy，客户端实现的Jedis，还有官方正在实现的Redis Cluster。

Twemproxy是目前较为成熟的分布式管理的成品，但是由于它本身是将Redis定位于缓存管理而不是持久存储，所以只能成为分布式缓存管理软件，因为它缺乏必要的高可用性和一致性保证。而Jedis由于客户端所限，无法具备易伸缩的扩展能力。

由于Twitter开源的Twemproxy(https://github.com/twitter/twemproxy)直接使用epoll驱动，导致其他不支持epoll的系统无法使用，因此我fork了一个版本，加入了kqueue支持，让FreeBSD和Mac os x能够成功编译运行，并且因为Twemproxy不合理的将epoll嵌入到业务逻辑中，因此，只能大改事件驱动的相关代码，将复用IO的结构抽象出来，并且增加了部分结构。

我的Twemproxy Fork版本: <https://github.com/yuyuyu101/twemproxy>

* + 1. 运维上的想法

其实快照和aof一样，都使用了Copy-on-write技术。多次试验发现每次做数据dump的时候，内存都会扩大一倍(关于这个问题可以参考我去年写的redis的内存陷阱，很多人用redis做为缓存，数据量小，dump耗时非常短暂，所以不太容易发现)，这个时候会有三种情况：

一：物理内存足以满足，这个时候dump非常快，性能最好

二：物理内存+虚拟内存可以满足，这个时候dump速度会比较慢，磁盘swap繁忙，服务性能也会下降。所幸的是经过一段比较长的时候数据dump完成了，然后内存恢复正常。这个情况系统稳定性差。

三： 物理内存+虚拟内存不能满足，这个时候dump一直死着，时间久了机器挂掉。这个情况就是灾难！

如果数据要做持久化又想保证稳定性，建议留空一半的物理内存。如果觉得无法接受还是有办法，下面讲：

快照和aof虽然都使用Copy-on-write，但有个不同点，快照你无法预测redis什么时候做dump，aof可以通过bgrewriteaof命令控制dump的时机。

根据这点我可以在一个服务器上开启多个redis节点(利用多CPU)，使用aof的持久化方式。

**例如在24G内存的服务器上开启3个节点，每天用bgrewriteaof定期重新整理数据，每个节点dump的时间都不一样，这样理论上每个节点可以消耗6G内存，一共使用18G内存，另外6G内存在单个节点dump时用到，内存一下多利用了6G！ 当然节点开的越多内存的利用率也越高。如果带宽不是问题，节点数建议 = CPU数。**

我的应用里为了保证高性能，数据没有做dump，也没有用aof。因为不做dump发生的故障远远低于做dump的时候，即使数据丢失了，自动修复脚本可以马上数据恢复。毕竟对海量数据redis只能做数据分片，那么落到每个节点上的数据量也不会很多。

redis的虚拟内存建议也不要用，用redis本来就是为了达到变态的性能，虚拟内存、aof看起来都有些鸡肋。

现在还离不开redis，因为它的mget是现在所有db里性能最好的，以前也考虑过用tokyocabinet hash方式做mget，性能不给力。直接用redis，基本上单个redis节点mget可以达到10W/s

纠错

之前说过redis做数据dump的时候内容会扩大一倍，后来我又做了些测试，发现有些地方说的不对。

top命令并不是反映真实的内存占用情况，在top里尽管fork出来的子进程占了和父进程一样的内存，但是当做dump的时候没有写操作，实际使用的是同一份内存的数据。当有写操作的时候内存才会真实的扩大（具体是不是真实的扩大一倍不确定，可能数据是按照页分片的），这才是真正的Copy-on-write。

基于这点在做数据持久化会更加灵活。

参考：http://www.yiihsia.com/2011/04/%E5%AF%B9redis%E6%95%B0%E6%8D%AE%E6%8C%81%E4%B9%85%E5%8C%96%E7%9A%84%E4%B8%80%E4%BA%9B%E6%83%B3%E6%B3%95/

* + 1. Redis与Mysql的同步

<http://blog.csdn.net/haier_jiang/article/details/8194670>

<http://blog.csdn.net/haier_jiang/article/details/8101624>

<http://blog.csdn.net/haier_jiang/article/details/8082734>

<http://blog.csdn.net/haier_jiang/article/details/8223010>

* + - 1. 采用binlog架构刷新缓存可行性分析
  + 采用binlog架构刷新缓存可行性分析
* Mysql日志格式介绍可参考我以前的的介绍。
* 对于使用MIXED日志格式，此日志格式，记录的是对应数据库操作的SQL语句，采用此日志方式存在的问题：
  + - * 对于一些未任何更新操作的SQl语句，像条件不满足，对应的sql也会记录到binlog日志中。
      * SQL语句记录的未必包括所有的更新操作。
      * 对于一些分布式数据库，对于SQL中的where条件指定的是非均衡字段，也许会存在多条SQL,跟设计有关！
      * 基于以上考虑，采用MIXED的日志格式进行binlog解析是行不通的。（官网给出的指示是failed statementsare not logged ，但不包括语法没错误，更新条件不符合对应的SQL）
  + 采用ROW日志格式
* 对于此日志格式，每行变化都有对应的记录，此日志格式，对于解析及采集数据都是非常方便的，也只有采用此日志格式，才能基于binlog修改，做刷新缓存相关方案的设计。但是基于此日志格式也存在一些问题：
* 需要考虑项目中是否有大量的批量的update操作，如果采用此日志格式，批量操作每一行修改都会记录一条日志，大量的批量操作所产生的日志量，以及所带来的IO开销是否可以接受。
* 通过以上分析，最终项目中还是考虑基于ROW日志格式进行缓存刷新，还有一个问题需要考虑，在应用层DB进行了相应的update操作后，所产生的Binlog是会带来一定的延迟，如果Binlog处理模块正常运行，数据是的延迟会非常少，MS级别以内，对用户体验是没有感知的，但是Binlog模块是多点，异常，以及相应的延迟肯定会是存在的，这样，缓存数据肯定会存在脏数据。
* 不过通过以上方案，数据能达到最终一致性，因此how to权衡，需要考虑。
* 通过以上分析，是否采用Binlog来做缓存数据刷新相信大家有一个基本概念了
  + 基于binlog刷新缓存的实现时注意的地方
* 如果是采用java做相关开发，可以使用开源的tungstenAPI
* Binlog日志解析是按照mysql 的master/slave同步流程来实现，即一个线程同步，一个线程解析。
* 设计是可分Binlog处理模块以及缓存处理SqlEvent两部分，其中Binlog处理解析好对应的SqlEvent，然后对应的缓存刷新处理SqlEvent,一个简单的生产者-消费者模式。
* 对于多个Binlog处理模块可以是单点，也可以是通过一些协同工具来管理，看需求。可以使用ZooKeeper等。
* 对于分布式缓存中的数据，对于Binlog来刷新的缓存数据会存在load数据的问题，为了减轻DB的额外压力，flush操作可在get缓存数据处完成。看需求，如果读写完全分享的话此DB的额外压力可以接收的话也可行。
* 对于缓存数据性一致性要求比较高的，可以通过版本号来控制，即在应用层引入一定的耦合，在DB操作时带mark ，缓存刷新是也mark，另外get操作时比较双版本号来达到数据的一致性。（此跟5谈论的一定的联系，读写是否完全分离，以及相应一致性实现的一些方法）
  + - 1. Mysql Binlog三种格式介绍及分析

一．Mysql Binlog格式介绍

Mysql binlog日志有三种格式，分别为Statement,MiXED,以及ROW！

1.Statement：每一条会修改数据的sql都会记录在binlog中。

优点：不需要记录每一行的变化，减少了binlog日志量，节约了IO，提高性能。(相比row能节约多少性能与日志量，这个取决于应用的SQL情况，正常同一条记录修改或者插入row格式所产生的日志量还小于Statement产生的日志量，但是考虑到如果带条件的update操作，以及整表删除，alter表等操作，ROW格式会产生大量日志，因此在考虑是否使用ROW格式日志时应该跟据应用的实际情况，其所产生的日志量会增加多少，以及带来的IO性能问题。)

缺点：由于记录的只是执行语句，为了这些语句能在slave上正确运行，因此还必须记录每条语句在执行的时候的一些相关信息，以保证所有语句能在slave得到和在master端执行时候相同 的结果。另外mysql 的复制,像一些特定函数功能，slave可与master上要保持一致会有很多相关问题(如sleep()函数， last\_insert\_id()，以及user-defined functions(udf)会出现问题).

使用以下函数的语句也无法被复制：

\* LOAD\_FILE()

\* UUID()

\* USER()

\* FOUND\_ROWS()

\* SYSDATE() (除非启动时启用了 --sysdate-is-now 选项)

同时在INSERT ...SELECT 会产生比 RBR 更多的行级锁

2.Row:不记录sql语句上下文相关信息，仅保存哪条记录被修改。

优点： binlog中可以不记录执行的sql语句的上下文相关的信息，仅需要记录那一条记录被修改成什么了。所以rowlevel的日志内容会非常清楚的记录下每一行数据修改的细节。而且不会出现某些特定情况下的存储过程，或function，以及trigger的调用和触发无法被正确复制的问题

缺点:所有的执行的语句当记录到日志中的时候，都将以每行记录的修改来记录，这样可能会产生大量的日志内容,比如一条update语句，修改多条记录，则binlog中每一条修改都会有记录，这样造成binlog日志量会很大，特别是当执行alter table之类的语句的时候，由于表结构修改，每条记录都发生改变，那么该表每一条记录都会记录到日志中。

3.Mixedlevel: 是以上两种level的混合使用，一般的语句修改使用statment格式保存binlog，如一些函数，statement无法完成主从复制的操作，则采用row格式保存binlog,MySQL会根据执行的每一条具体的sql语句来区分对待记录的日志形式，也就是在Statement和Row之间选择一种.新版本的MySQL中队row level模式也被做了优化，并不是所有的修改都会以row level来记录，像遇到表结构变更的时候就会以statement模式来记录。至于update或者delete等修改数据的语句，还是会记录所有行的变更。

二.Binlog基本配制与格式设定

1.基本配制

Mysql BInlog日志格式可以通过mysql的my.cnf文件的属性binlog\_format指定。如以下：

binlog\_format = MIXED //binlog日志格式

log\_bin =目录/mysql-bin.log //binlog日志名

expire\_logs\_days = 7 //binlog过期清理时间

max\_binlog\_size 100m //binlog每个日志文件大小

2.Binlog日志格式选择

Mysql默认是使用Statement日志格式，推荐使用MIXED.

由于一些特殊使用，可以考虑使用ROWED，如自己通过binlog日志来同步数据的修改，这样会节省很多相关操作。对于binlog数据处理会变得非常轻松,相对mixed，解析也会很轻松(当然前提是增加的日志量所带来的IO开销在容忍的范围内即可)。

3.mysqlbinlog格式选择

mysql对于日志格式的选定原则:如果是采用 INSERT，UPDATE，DELETE 等直接操作表的情况，则日志格式根据 binlog\_format 的设定而记录,如果是采用 GRANT，REVOKE，SET PASSWORD 等管理语句来做的话，那么无论如何 都采用 SBR 模式记录

三．Mysql Binlog日志分析

通过MysqlBinlog指令查看具体的mysql日志，如下:

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

SET TIMESTAMP=1350355892/\*!\*/;

BEGIN

/\*!\*/;

# at 1643330

#121016 10:51:32 server id 1 end\_log\_pos 1643885 Query thread\_id=272571 exec\_time=0 error\_code=0

SET TIMESTAMP=1350355892/\*!\*/;

Insert into T\_test….)

/\*!\*/;

# at 1643885

#121016 10:51:32 server id 1 end\_log\_pos 1643912 Xid = 0

COMMIT/\*!\*/;

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

1.开始事物的时间:

SET TIMESTAMP=1350355892/\*!\*/;

BEGIN

2.sqlevent起点

#at 1643330 :为事件的起点，是以1643330字节开始。

3.sqlevent 发生的时间点

#121016 10:51:32:是事件发生的时间，

4.serverId

server id 1 :为master 的serverId

5.sqlevent终点及花费时间，错误码

end\_log\_pos 1643885:为事件的终点，是以1643885 字节结束。

execTime 0: 花费的时间

error\_code=0:错误码

Xid:事件指示提交的XA事务

Mixed日志说明：

在slave日志同步过程中，对于使用now这样的时间函数，MIXED日志格式，会在日志中产生对应的unix\_timestamp()\*1000的时间字符串，slave在完成同步时，取用的是sqlEvent发生的时间来保证数据的准确性。另外对于一些功能性函数slave能完成相应的数据同步，而对于上面指定的一些类似于UDF函数，导致Slave无法知晓的情况，则会采用ROW格式存储这些Binlog，以保证产生的Binlog可以供Slave完成数据同步。

* + - 1. 基于tungsten监测mysql数据修改系统介绍

一.项目背景

考虑到我们新项目中使用到的是mysql 数据库，同时通过Binlog来监测数据的更新情况在业界也有相关的应用，比如通过Binlog来更新缓存（这也是我现在手上正在做的一个项目，正在开发过程中），对新浪更新redis缓存也是基于此，相关很多公司都有应用，以上为基于Mysql Binlog来监测数据库数据更新情况的背景说明。

二.开源项目选择

Tungsten是一个开源的数据库同步工具，详细可参考官网（https://code.google.com/p/tungsten-replicator/）其提供的tungsten API可供下载，此API实现了mysql slave同步流程的相关功能。

三.基本流程

基于Mysql Binlog完成数据监控主要有三个流程：（对于tungsten使用的API包主要是：com.continuent.tungsten.replicator.extractor.mysql）

1.把Binlog从指定的mysql master日志目录同步过来，通过以上包中的RelayClient类即可，可指定mysql主机地址，相关用户名密码，即JDBC连接的方式参数指定，即可完成Binlog日志的同步。

2.把同步过来的Binlog日志解析成我们能看的懂的SQL语句。对于此解析，由于日志格式的不同，解析也有不同，是通过MysqlExtractor类，指定Binlog的本地目录，即可完成解析。（对于mysql binlog的日志说明可以参考：Mysql Binlog三种格式介绍及分析）

3.在以上解析后会得到对应的Sql语句，对于如果选用row日志格式，会知道具体的数据变化情况,MysqlExtractor类数据可转换化RowChangeData对象，即可完成修改数据的读取;如果是基于Statement的日志解析，得到的对应的SQL语句，如果需要提取对应数据修改，需要引入SQL解析，我们项目中采用了MIXED的日志格式，SQL解析采用了JSqlParser,此开源项目对SQL的解析有限，它是通过javacc来完成SQL语义的解析，我们项目中对javacc定义文件的做了修改，使其支撑更复杂的SQL解析。

通过以上三个流程，即可完成数据库对应修改数据的提取。对应的mysql产生的binlog会有很多，在实际应用中解析时可以跟据自己监测的特定SQL，这样很大提高解析速度。（PS:对于增加特定SQL解析所用的时间是几百甚至几十纳表即可完成，但是对于一次解析则需要MS级别的解析。因此如何选择供借鉴）以上是整体流程，一般是多线程异步执行（mysql slave同步流程也是如此）。一个线程执行RelayClient完成日志的同步流程，一个线程完成MysqlExtractor的日志解析流程。一般来说，日志同步的速度远远大于日志解析的速度，因此对于流程的控制是个需要考虑的问题。（我们是修改tungsten API源码，通过java Semaphore信号量来控制日志的同步与解析速度）日志同步如果没有新的新的日志产生会出现等待新日志更新，如果没有新的日志更新，日志解析线程也会等待，此流程跟Mysql的slave同步线程的流程是一样的。实际线上使用时，mysql日志的产生速度，以及mysql的解析速度可测试，实时性是否满足需求，我们的流程在测试阶段还算比较不错，实时性在MS级别上。

四:说明

上面是基于tungsten API完成mysql数据更新监控项目的简单介绍。在此需要注意是不同的日志格式，解析相关比较大，因此在考虑自己应用的基础上，通过部分的IO开销，来使解析变得容易是可取的。而且statement日志格式对应的解析有时是做不了的。（个人推荐mysql 使用row日志格式）

五:基于tungsten EOF packet received的问题解决

查看mysql主从同步时，不同slave相关配制时 会配制不同的serverId，立即查看tungsten相关源码，其中有此项配制，由于测试环境与开发环境使用的是同一mysql结点，并且使用了同一serverId，从而引起了此问题的产生，经测试，使用不同serverId，此问题不再重现。

* + 1. 一步完成 MySQL 向 Redis 迁移

<http://www.oschina.net/translate/mysql-to-redis-in-one-step>

Mysql到Redis的数据协议

redis-cli命令行工具有一个批量插入模式，是专门为批量执行命令设计的。这第一步就是把Mysql查询的内容格式化成redis-cli可用的数据格式。here we go!

我的统计表：

|  |
| --- |
| CREATE TABLE events\_all\_time (  id int(11) unsigned NOT NULL AUTO\_INCREMENT,  action varchar(255) NOT NULL,  count int(11) NOT NULL DEFAULT 0,  PRIMARY KEY (id),  UNIQUE KEY uniq\_action (action)  ); |

准备在每行数据中执行的redis命令如下：

|  |
| --- |
| HSET events\_all\_time [action] [count] |

按照以上redis命令规则，创建一个events\_to\_redis.sql文件，内容是用来生成redis数据协议格式的SQL：

|  |
| --- |
| -- events\_to\_redis.sql  SELECT CONCAT(  "\*4\r\n",  '$', LENGTH(redis\_cmd), '\r\n',  redis\_cmd, '\r\n',  '$', LENGTH(redis\_key), '\r\n',  redis\_key, '\r\n',  '$', LENGTH(hkey), '\r\n',  hkey, '\r\n',  '$', LENGTH(hval), '\r\n',  hval, '\r'  )  FROM (  SELECT  'HSET' as redis\_cmd,  'events\_all\_time' AS redis\_key,  action AS hkey,  count AS hval  FROM events\_all\_time  ) AS t |

ok, 用下面的命令执行：

|  |
| --- |
| mysql stats\_db --skip-column-names --raw < events\_to\_redis.sql | redis-cli --pipe |

很重要的mysql参数说明：

--raw: 使mysql不转换字段值中的换行符。

--skip-column-names: 使mysql输出的每行中不包含列名。

* + 1. 从Redis的数据丢失说起

<http://www.litrin.net/2011/12/22/%E4%BB%8Eredis%E7%9A%84%E6%95%B0%E6%8D%AE%E4%B8%A2%E5%A4%B1%E8%AF%B4%E8%B5%B7/>

碰到一个悲催的事情：一台Redis服务器，4核，16G内存且没有任何硬件上的问题。持续高压运行了大约3个月，保存了大约14G的数据，设置了比较完备的Save参数。而就是这台主机，在一次重起之后，丢失了大量的数据，14G的数据最终只恢复了几百兆而已。

重启动时的日志：

|  |
| --- |
| [26641] 21 Dec 09:46:34 \* Starting BGSAVE for SYNC  [26641] 21 Dec 09:46:34 # Can’t save in background: fork: Cannot allocate memory  [26641] 21 Dec 09:46:34 \* Replication failed, can’t BGSAVE |

在小内存的进程上做一个fork,不需要太多资源，但当这个进程的内存空间以Ｇ为单位时，fork就成为一件很恐怖的操作。何况在16G内存的主机上fork 14G内存的进程呢？肯定会报内存无法分配的。更可气的是，越是改动频繁的主机上fork也越频繁，fork操作本身的代价恐怕也不会比假死好多少。

找到原因之后，直接修改内核参数vm.overcommit\_memory = 1

Linux内核会根据参数vm.overcommit\_memory参数的设置决定是否放行。

vm.overcommit\_memory = 1，直接放行

vm.overcommit\_memory = 0：则比较 此次请求分配的虚拟内存大小和系统当前空闲的物理内存加上swap，决定是否放行。

vm.overcommit\_memory = 2：则会比较 进程所有已分配的虚拟内存加上此次请求分配的虚拟内存和系统当前的空闲物理内存加上swap，决定是否放行。

可以参考3.3上说明，需要对内存进行预留。

* + 1. Redis持久化实践及灾难恢复模拟

http://heylinux.com/archives/1932.html

既然持久化的数据的作用是用于重启后的数据恢复，那么我们就非常有必要进行一次这样的灾难恢复模拟了。

据称如果数据要做持久化又想保证稳定性，则建议留空一半的物理内存。因为在进行快照的时候，fork出来进行dump操作的子进程会占用与父进程一样的内存，真正的copy-on-write，对性能的影响和内存的耗用都是比较大的。

目前，通常的设计思路是利用Replication机制来弥补aof、snapshot性能上的不足，达到了数据可持久化。

即Master上Snapshot和AOF都不做，来保证Master的读写性能，而Slave上则同时开启Snapshot和AOF来进行持久化，保证数据的安全性。

首先，修改Master上的如下配置：

$ sudo vim /opt/redis/etc/redis\_6379.conf

|  |
| --- |
| #save 900 1 #禁用Snapshot  #save 300 10  #save 60 10000  appendonly no #禁用AOF |

接着，修改Slave上的如下配置：

$ sudo vim /opt/redis/etc/redis\_6379.conf

|  |
| --- |
| save 900 1 #启用Snapshot  save 300 10  save 60 10000  appendonly yes #启用AOF  appendfilename appendonly.aof #AOF文件的名称  # appendfsync always  appendfsync everysec #每秒钟强制写入磁盘一次  # appendfsync no    no-appendfsync-on-rewrite **yes** #在日志重写时，不进行命令追加操作  auto-aof-rewrite-percentage 100 #自动启动新的日志重写过程  auto-aof-rewrite-min-size 64mb #启动新的日志重写过程的最小值 |

分别启动Master与Slave

$ /etc/init.d/redis start

启动完成后在Master中确认未启动Snapshot参数

redis 127.0.0.1:6379> CONFIG GET save

1) "save"

2) ""

然后通过以下脚本在Master中生成25万条数据

在数据的生成过程中，可以很清楚的看到Master上仅在第一次做Slave同步时创建了dump.rdb文件，之后就通过增量传输命令的方式给Slave了。

dump.rdb文件没有再增大。

dongguo@redis:/opt/redis/data/6379$ ls -lh

total 4.0K

-rw-r--r-- 1 root root 10 Sep 27 00:40 dump.rdb

而Slave上则可以看到dump.rdb文件和AOF文件在不断的增大，并且AOF文件的增长速度明显大于dump.rdb文件。

dongguo@redis-slave:/opt/redis/data/6379$ ls -lh

total 24M

-rw-r--r-- 1 root root 15M Sep 27 12:06 appendonly.aof

-rw-r--r-- 1 root root 9.2M Sep 27 12:06 dump.rdb

等待数据插入完成以后，首先确认当前的数据量。

当前的数据量为25万条key，占用内存31.52M。

然后我们直接Kill掉Master的Redis进程，模拟灾难。

dongguo@redis:/opt/redis/data/6379$ sudo killall -9 redis-server

我们到Slave中查看状态：

可以看到master\_link\_status的状态已经是down了，Master已经不可访问了。

而此时，Slave依然运行良好，并且保留有AOF与RDB文件。

下面我们将通过Slave上保存好的AOF与RDB文件来恢复Master上的数据。

首先，将Slave上的同步状态取消，避免主库在未完成数据恢复前就重启，进而直接覆盖掉从库上的数据，导致所有的数据丢失。

redis 127.0.0.1:6379> SLAVEOF NO ONE

OK

确认一下已经没有了master相关的配置信息：

在Slave上复制数据文件：

dongguo@redis-slave:/opt/redis/data/6379$ tar cvf /home/dongguo/data.tar \*

appendonly.aof

dump.rdb

将data.tar上传到Master上，尝试恢复数据:

可以看到Master目录下有一个初始化Slave的数据文件，很小，将其删除。

dongguo@redis:/opt/redis/data/6379$ ls -l

total 4

-rw-r--r-- 1 root root 10 Sep 27 00:40 dump.rdb

dongguo@redis:/opt/redis/data/6379$ sudo rm -f dump.rdb

然后解压缩数据文件：

dongguo@redis:/opt/redis/data/6379$ sudo tar xf /home/dongguo/data.tar

dongguo@redis:/opt/redis/data/6379$ ls -lh

total 29M

-rw-r--r-- 1 root root 18M Sep 27 01:22 appendonly.aof

-rw-r--r-- 1 root root 12M Sep 27 01:22 dump.rdb

启动Master上的Redis；

dongguo@redis:/opt/redis/data/6379$ sudo /etc/init.d/redis start

Starting Redis server...

查看数据是否恢复：

redis 127.0.0.1:6379> INFO

可以看到25万条数据已经完整恢复到了Master上。

此时，可以放心的恢复Slave的同步设置了。

redis 127.0.0.1:6379> SLAVEOF 10.6.1.143 6379

OK

查看同步状态：

redis 127.0.0.1:6379> INFO

master\_link\_status显示为up，同步状态正常。

在此次恢复的过程中，我们同时复制了AOF与RDB文件，那么到底是哪一个文件完成了数据的恢复呢？

实际上，当Redis服务器挂掉时，重启时将按照以下优先级恢复数据到内存：

1. 如果只配置AOF,重启时加载AOF文件恢复数据；

2. 如果同时 配置了RDB和AOF,启动是只加载AOF文件恢复数据;

3. 如果只配置RDB,启动是将加载dump文件恢复数据。

也就是说，AOF的优先级要高于RDB，这也很好理解，因为AOF本身对数据的完整性保障要高于RDB。

在此次的案例中，我们通过在Slave上启用了AOF与RDB来保障了数据，并恢复了Master。

但在我们目前的线上环境中，由于数据都设置有过期时间，采用AOF的方式会不太实用，过于频繁的写操作会使AOF文件增长到异常的庞大，大大超过了我们实际的数据量，这也会导致在进行数据恢复时耗用大量的时间。

因此，可以在Slave上仅开启Snapshot来进行本地化，同时可以考虑将save中的频率调高一些或者调用一个计划任务来进行定期bgsave的快照存储，来尽可能的保障本地化数据的完整性。

在这样的架构下，如果仅仅是Master挂掉，Slave完整，数据恢复可达到100%。

如果Master与Slave同时挂掉的话，数据的恢复也可以达到一个可接受的程度。

* + 1. redis运维之道

<http://v.youku.com/v_show/id_XMjgxMTg2NTg4.html>

* + - * 增量同步:Postion\_rdb+AOF
      * 加载速度:
        + sortSet限制:score为timestamp member为整形
        + 修改源码：将内存中各种类型的数值直接保存到rdb文件中，加载rdb文件时直接还原各种数值类型，无需进行转换。
        + 修改源码 key值为hash处理,value为8为int
      * redis绑定cpu
      * 双写—>容灾
    1. Redis复制与可扩展集群搭建

注：**本文与4.2.8是对新浪redis的全面解析，具体实现，可参考4.2.4与4.2.11，基本上所有功能都能实现**。

<http://blog.nosqlfan.com/html/3153.html?ref=rediszt>

Redis的作者提出了一种叫做presharding的方案来解决动态扩容和数据分区的问题，实际就是在同一台机器上部署多个Redis实例的方式，当容量不够时将多个实例拆分到不同的机器上，这样实际就达到了扩容的效果。

拆分过程如下：

1.在新机器上启动好对应端口的Redis实例。

2.配置新端口为待迁移端口的从库。

3.待复制完成，与主库完成同步后，切换所有客户端配置到新的从库的端口。

4.配置从库为新的主库。

5.移除老的端口实例。

6.重复上述过程迁移好所有的端口到指定服务器上。

以上拆分流程是Redis作者提出的一个平滑迁移的过程，不过该拆分方法还是很依赖Redis本身的复制功能的，如果主库快照数据文件过大，这个复制的过程也会很久，同时会给主库带来压力。所以做这个拆分的过程最好选择为业务访问低峰时段进行。

##### Redis复制的改进思路

##### 我们线上的系统使用了我们自己改进版的Redis,主要解决了Redis没有增量复制的缺陷，能够完成类似Mysql Binlog那样可以通过从库请求日志位置进行增量复制。

##### 我们的持久化方案是首先写Redis的AOF文件，并对这个AOF文件按文件大小进行自动分割滚动，同时关闭Redis的Rewrite命令，然后会在业务低峰时间进行内存快照存储，并把当前的AOF文件位置一起写入到快照文件中，这样我们可以使快照文件与AOF文件的位置保持一致性，这样我们得到了系统某一时刻的内存快照，并且同时也能知道这一时刻对应的AOF文件的位置，那么当从库发送同步命令时，我们首先会把快照文件发送给从库，然后从库会取出该快照文件中存储的AOF文件位置，并将该位置发给主库，主库会随后发送该位置之后的所有命令，以后的复制就都是这个位置之后的增量信息了。

##### medium

##### Redis与MySQL的结合

##### 目前大部分互联网公司使用MySQL作为数据的主要持久化存储，那么如何让Redis与MySQL很好的结合在一起呢？我们主要使用了一种基于MySQL作为主库，Redis作为高速数据查询从库的异构读写分离的方案。

##### 为此我们专门开发了自己的MySQL复制工具，可以方便的实时同步MySQL中的数据到Redis上。

##### medium2

##### （MySQL-Redis 异构读写分离）

##### 总结：

##### 1.Redis的复制功能没有增量复制，每次重连都会把主库整个内存快照发给从库，所以需要避免向在线服务的压力较大的主库上增加从库。

##### 2.Redis的复制由于会使用快照持久化方式，所以如果你的Redis持久化方式选择的是日志追加方式(aof),那么系统有可能在同一时刻既做aof日志文件的同步刷写磁盘，又做快照写磁盘操作，这个时候Redis的响应能力会受到影响。所以如果选用aof持久化，则加从库需要更加谨慎。

##### 3.可以使用主动复制和presharding方法进行Redis集群搭建与在线扩容。

* + 1. Redis监控技巧

http://blog.nosqlfan.com/html/4166.html

内存使用

如果 Redis 使用的内存超出了可用的物理内存大小，那么 Redis 很可能系统会被 OOM Killer 杀掉。针对这一点，你可以通过 info 命令对 used\_memory 和 used\_memory\_peak 进行监控，为使用内存量设定阈值，并设定相应的报警机制。当然，报警只是手段，重要的是你得预先计划好，当内存使用量过大后，你应该做些什么，是清除一些没用的冷数据，还是把 Redis 迁移到更强大的机器上去。

持久化

如果因为你的机器或 Redis 本身的问题导致 Redis 崩溃了，那么你唯一的救命稻草可能就是 dump 出来的 rdb文件了，所以，对 Redis dump 文件进行监控也是很重要的。你可以通过对 rdb\_last\_save\_time 进行监控，了解你最近一次 dump 数据操作的时间，还可以通过对 rdb\_changes\_since\_last\_save 进行监控来知道如果这时候出现故障，你会丢失多少数据。

主从复制

如果你设置了主从复制模式，那么你最好对复制的情况是否正常做一些监控，主要是对 info 输出中的 master\_link\_status 进行监控，如果这个值是 up，那么说明同步正常，如果是 down，那么你就要注意一下输出的其它一些诊断信息了

Fork 性能

当 Redis 持久化数据到磁盘上时，它会进行一次 fork 操作，通过 fork 对内存的 copy on write 机制最廉价的实现内存镜像。但是虽然内存是 copy on write 的，但是虚拟内存表是在 fork 的瞬间就需要分配，所以 fork 会造成主线程短时间的卡顿（停止所有读写操作），这个卡顿时间和当前 Redis 的内存使用量有关。通常 GB 量级的 Redis 进行 fork 操作的时间在毫秒级。你可以通过对 info 输出的 latest\_fork\_usec 进行监控来了解最近一次 fork 操作导致了多少时间的卡顿。

配置一致

Redis 支持使用 CONFIG SET 操作来实现运行实的配置修改，这很方便，但同时也会导致一个问题。就是通过这个命令动态修改的配置，是不会同步到你的配置文件中去的。所以当你因为某些原因重启 Redis 时，你使用 CONFIG SET 做的配置修改就会丢失掉，所以我们最好保证在每次使用 CONFIG SET 修改配置时，也把配置文件一起相应地改掉。为了防止人为的失误，所以我们最好对配置进行监控，使用 CONFIG GET 命令来获取当前运行时的配置，并与 redis.conf 中的配置值进行对比，如果发现两边对不上，就启动报警。

慢日志

Redis 提供了 SLOWLOG 指令来获取最近的慢日志，Redis 的慢日志是直接存在内存中的，所以它的慢日志开销并不大，在实际应用中，我们通过 crontab 任务执行 SLOWLOG 命令来获取慢日志，然后将慢日志存到文件中，并用 Kibana 生成实时的性能图表来实现性能监控。

值得一提的是，Redis 的慢日志记录的时间，仅仅包括 Redis 自身对一条命令的执行时间，不包括 IO 的时间，比如接收客户端数据和发送客户端数据这些时间。另外，Redis 的慢日志和其它数据库的慢日志有一点不同，其它数据库偶尔出现 100ms 的慢日志可能都比较正常，因为一般数据库都是多线程并发执行，某个线程执行某个命令的性能可能并不能代表整体性能，但是对 Redis 来说，它是单线程的，一旦出现慢日志，可能就需要马上得到重视，最好去查一下具体是什么原因了。

监控服务

-Sentinel

Sentinel 是 Redis 自带的工具，它可以对 Redis 主从复制进行监控，并实现主挂掉之后的自动故障转移。在转移的过程中，它还可以被配置去执行一个用户自定义的脚本，在脚本中我们就能够实现报警通知等功能。

-Redis Live

Redis Live 是一个更通用的 Redis 监控方案，它的原理是定时在 Redis 上执行 MONITOR 命令，来获取当前 Redis 当前正在执行的命令，并通过统计分析，生成web页面的可视化分析报表。

-Redis Faina

Redis Faina 是由著名的图片分享应用 instagram 开发的 Redis 监控服务，其原理和 Redis Live 类似，都是对通过 MONITOR 来做的。

数据分布

弄清 Redis 中数据存储分布是一件很难的是，比如你想知道哪类型的 key 值占用内存最多。下面是一些工具，可以帮助你对 Redis 的数据集进行分析。

-Redis-sampler

Redis-sampler 是 Redis 作者开发的工具，它通过采样的方法，能够让你了解到当前 Redis 中的数据的大致类型，数据及分布状况。

-Redis-audit

Redis-audit 是一个脚本，通过它，我们可以知道每一类 key 对内存的使用量。它可以提供的数据有：某一类 key 值的访问频率如何，有多少值设置了过期时间，某一类 key 值使用内存的大小，这很方便让我们能排查哪些 key 不常用或者压根不用。

-Redis-rdb-tools

Redis-rdb-tools 跟 Redis-audit 功能类似，不同的是它是通过对 rdb 文件进行分析来取得统计数据的。

* + 1. Redis数据分片以及扩容

**注意：该文章能扩展实现增量RDB+AOF，也就是4.2.8运维之道中提到的增量保存。**

<http://blog.nosqlfan.com/html/4092.html>

**分片方法**

我们单服务器的内存是64G，我们估计64\* 16G 在很长一段时间内是满足需求的，但是这个数据积累的过程可能比较缓慢，很长一段时间不会超过64G，因此一台Redis 服务器就足够，但是为了考虑到以后的扩展，一开始将数据sharding 到16个db中，也就是说在只有一个Redis时，client每次写数据会先计算key的hash,模16，得到db num，select db，然后写入，也可以为每个db 保持一个client，这样就可以避免每次select db 了。 当需要变更为两个Redis时，为了不丢失数据，需要将原来Redis的数据分为2份，一份是db 0-7,第二份是db 8-15, 用这两个数据启动Redis，就可以实现扩容了，因此必须要要有脚本能够切分Redis dump 出来的dump.rdb， 下面介绍我们的切分脚本：

* + dump.rdb 结构： head + db 0 + db 1 + … + db n + eof
  + 修改Redis，在启动过程中打印出每部分的offset
  + 提供一个c 程序，可以将一个大文件按照指定的offset 进行切分
  + 将切分出的各个部分进行重新组装

例如我们的例子

* + 初始时dump.rdb 的结构： head + db0 + … + db15 + eof
  + 得到head 以及每个db的offset
  + 切分出head， db0 + … + db7 , db8 + … + db15 三个部分
  + 将head ，db0 + … + db7 cat 在一个文件中，并在结尾加上eof，同样，将head , db7 + … + db15 cat在一个文件中，加上eof
  + 用上面的两个文件启动Redis，完成数据切分

**脚本**

切分脚本如下：

|  |
| --- |
| echo "Usage start-end db"  startdb=$1 #上面的例子start 0  enddb=$2 # end 是 7  outdb="$1-$2.rdb" #输出文件的名字  if [[ ! -f "dump.rdb" ]]; then #使用当前目录下dump.rdb 作为源文件  echo "no dump.rdb,must have"  exit -1  fi  #使用修改过的redis，打印offset，然后退出，使用awk 得到head 的offset  headstart=`/global/share/bin/chenjp/redis-db-offset >& tmp.log ; cat tmp.log | grep offset | grep -v dbid | awk -F '=' '{print $NF}'`  headfile="split-0-$headstart" #head所在文件  /global/share/bin/chenjp/vsplit dump.rdb 0 $headstart #根据offset 切分文件，0- headoffset 为head  dbstartoffset=`cat tmp.log | grep "dbid=$startdb" | awk -F '=|,' '{print $(NF-2)}'` #找到db的offset  dbendoffset=`cat tmp.log | grep "dbid=$enddb" | awk -F '=|,' '{print $(NF-2)}'`  if [[ $dbendoffset -eq "" ]];then  dbendoffset=`ls -l dump.rdb | awk -F ' ' '{print $5}'`  fi  echo "start:"$dbstartoffset":"$dbendoffset  dbfile="split-$dbstartoffset-$dbendoffset"  /global/share/bin/chenjp/vsplit dump.rdb $dbstartoffset $dbendoffset #得到db 文件  cat $headfile $dbfile > $outdb #拼接  printf "\xff" >> $outdb #eof  #rm -rf tmp.log  #rm -rf split-\*  echo "file $dbfile ok, containts db $startdb to $enddb, pls mv to dump.rdb to start redis server" |

**redis-db-offset原理**

而上面的redis-db-offset实现也并不困难，只需要在load的时候将各个db开始的offset值打印也来就行了。diff如下：

|  |
| --- |
| [chenjp@nb290 redis-2.4.10]$ diff src/rdb.c ../../redis-2.4.10/src/rdb.c  959c959  <  ---  > fprintf(stderr,"redis\_db head finished,offset=%lld\n",ftell(fp));  982a983,984  >  > long db\_start = ftell(fp) - 1;  988a991  > fprintf(stderr,"redis\_db select,offset=%lld,dbid=%d\n",db\_start,dbid);  [chenjp@nb290 redis-2.4.10]$ diff src/redis.c ../../redis-2.4.10/src/redis.c  1790a1791,1792  >  > //exit(1); |

* + 1. 节约内存
       1. Redis内存容量的预估和优化

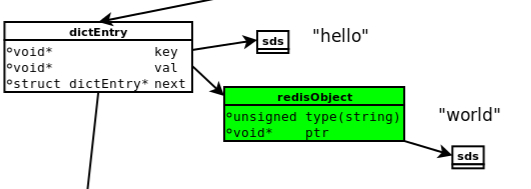
<http://blog.nosqlfan.com/html/3430.html?ref=rediszt>

Redis是个内存全集的kv数据库,不存在部分数据在磁盘部分数据在内存里的情况,所以提前预估和节约内存非常重要.本文将以最常用的string和zipmap两类数据结构在jemalloc内存分配器下的内存容量预估和节约内存的方法.

先说说jemalloc,传说中解决firefox内存问题freebsd的默认malloc分配器,area,thread-cache功能和tmalloc非常的相识.在2.4版本被Redis引入,在antirez的博文中提到内节约30%的内存使用.相比glibc的malloc需要在每个内存外附加一个额外的4字节内存块,jemalloc可以通过je\_malloc\_usable\_size函数获得指针实际指向的内存大小,这样Redis里的每个key或者value都可以节约4个字节,不少阿.

**string**

string类型看似简单,但还是有几个可优化的点.先来看一个简单的set命令所添加的数据结构.



一个set hello world命令最终(中间会malloc,free的我们不考虑)会产生4个对象,一个dictEntry(12字节),一个sds用于存储key,还有一个redisObject(12字节),还有一个存储string的sds.sds对象除了包含字符串本生之外,还有一个sds header和额外的一个字节作为字符串结尾共9个字节.

|  |
| --- |
| sds.c  ========  51 sds sdsnewlen(const void \*init, size\_t initlen) {  52 struct sdshdr \*sh;  53  54 sh = zmalloc(sizeof(struct sdshdr)+initlen+1);  sds.h  =======  39 struct sdshdr {  40 int len;  41 int free;  42 char buf[];  43  }; |

根据jemalloc size class那张表,这个命令最终申请的内存为16(dictEtnry) + 16 (redisObject) + 16(“hello”) + 16(“world”),一共64字节.注意如果key或者value的字符串长度+9字节超过16字节,则实际申请的内存大小32字节.

提一下string常见的优化方法

**尽量使value为纯数字**

这样字符串会转化成int类型减少内存的使用.

|  |
| --- |
| redis.c  =========  37 void setCommand(redisClient \*c) {  38 c->argv[2] = tryObjectEncoding(c->argv[2]);  39 setGenericCommand(c,0,c->argv[1],c->argv[2],NULL);  40 }  object.c =======  275 o->encoding = REDIS\_ENCODING\_INT;  276 sdsfree(o->ptr);  277 o->ptr = (void\*) value; |

这样一个set hello 111就只需要32字节,连redisObject也省了.所以对于value都是小数字的应用,适当调大REDIS\_SHARED\_INTEGERS这个宏可以很好的节约内存.

出去kv之外,dict的bucket逐渐变大也需要消耗内存,bucket的元素是个指针(dictEntry\*\*), 而bucket的大小是超过key个数向上求整的2的n次方,对于1w个key如果rehash过后就需要16384个bucket.（注:2#14=16384个key值正好>1w）

开始string类型的容量预估测试, 脚本如下

|  |
| --- |
| #! /bin/bash  redis-cli info|grep used\_memory:  for (( start = 10000; start < 30000; start++ ))  do  redis-cli set a$start baaaaaaaa$start > /dev/null  done  redis-cli info|grep used\_memory: |

根据上面的总结我们得出string公式

**string类型的内存大小 = 键值个数 \* (dictEntry大小 + redisObject大小 + 包含key的sds大小 + 包含value的sds大小) + bucket个数 \* 4**

下面是我们的预估值

|  |
| --- |
| >>> 20000 \* (16 + 16 + 16 + 32) + 32768 \* 4  1731072 |

运行一下测试脚本

|  |
| --- |
| hoterran@~/Projects/redis-2.4.1$ bash redis-mem-test.sh  used\_memory:564352  used\_memory:2295424 |

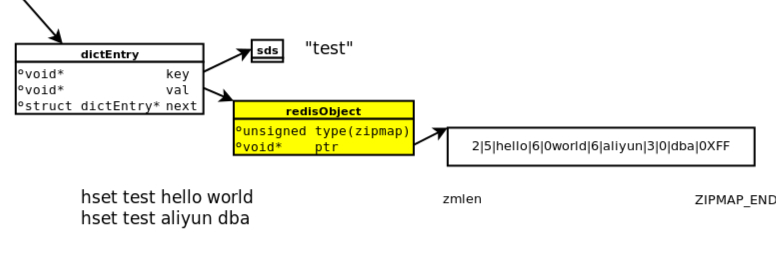
计算一下差值

|  |
| --- |
| >>> 2295424 - 564352  1731072 |

都是1731072,说明预估非常的准确, ^\_^

**zipmap**

这篇文章已经解释zipmap的效果,可以大量的节约内存的使用.对于一个普通的subkey和value,只需要额外的3个字节(keylen,valuelen,freelen)来存储,另外的hash key也只需要额外的2个字节(zm头尾)来存储subkey的个数和结束符.



**zipmap类型的内存大小 = hashkey个数 \* (dictEntry大小 + redisObject大小 + 包含key的sds大小 + subkey的总大小) + bucket个数 \* 4**

开始容量预估测试,100个hashkey,其中每个hashkey里包含300个subkey, 这里key+value的长度为5字节

|  |
| --- |
| #! /bin/bash  redis-cli info|grep used\_memory:  for (( start = 100; start < 200; start++ ))  do  for (( start2 = 100; start2 < 400; start2++ ))  do  redis-cli hset test$start a$start2 "1" > /dev/null  done  done  redis-cli info|grep used\_memory: |

这里subkey是同时申请的的,大小是300 \* (5 + 3) + 2 =2402字节,根据上面jemalloc size class可以看出实际申请的内存为2560.另外100hashkey的bucket是128.所以总的预估大小为

|  |
| --- |
| >>> 100 \* (16 + 16 + 16 + 2560) + 128 \* 4  261312 |

运行一下上面的脚本

|  |
| --- |
| hoterran@~/Projects/redis-2.4.1$ bash redis-mem-test-zipmap.sh  used\_memory:555916  used\_memory:817228 |

计算一下差值

|  |
| --- |
| >>> 817228 - 555916  261312 |

是的完全一样,预估很准确.

另外扯扯zipmap的一个缺陷,zipmap用于记录subkey个数的zmlen只有一个字节,超过254个subkey后则无法记录,需要遍历整个zipmap才能获得subkey的个数.而我们现在常把hash\_max\_zipmap\_entries设置为1000,这样超过254个subkey之后每次hset效率都很差.

|  |
| --- |
| 354 if (zm[0] < ZIPMAP\_BIGLEN) {  355 len = zm[0]; //小于254,直接返回结果  356 } else {  357 unsigned char \*p = zipmapRewind(zm); //遍历zipmap  358 while((p = zipmapNext(p,NULL,NULL,NULL,NULL)) != NULL) len++;  359  360 /\* Re-store length if small enough \*/  361 if (len < ZIPMAP\_BIGLEN) zm[0] = len;  362 } |

简单把zmlen设置为2个字节(可以存储65534个subkey)可以解决这个问题,今天和antirez聊了一下,这会破坏rdb的兼容性,这个功能改进推迟到3.0版本,另外这个缺陷可能是weibo的Redis机器cpu消耗过高的原因之一.

* + - 1. 节约内存：Instagram的Redis实践

http://blog.nosqlfan.com/html/3379.html

Instagram的开发者首先否定了数据库存储的方案，他们保持了KISS原则（Keep It Simple and Stupid），因为这个应用根本用不到数据库的update功能，事务功能和关联查询等等牛X功能，所以不必为这些用不到的功能去选择维护一个数据库。

于是他们选择了Redis，Redis是一个支持持久化的内存数据库，所有的数据都被存储在内存中（忘掉VM吧），而最简单的实现就是使用Redis的String结构来做一个key-value存储就行了。像这样：

|  |
| --- |
| SET media:1155315 939  GET media:1155315  > 939 |

其中1155315是图片ID，939是用户ID，我们将每一张图片ID为作key，用户uid作为value来存成key-value对。然后他们进行了测试，将数据按上面的方法存储，1,000,000数据会用掉70MB内存，300,000,000张照片就会用掉21GB的内存。对比预算的17GB还是超支了。

（**NoSQLFan：其实这里我们可以看到一个优化点，我们可以将key值前面相同的media去掉，只存数字，这样key的长度就减少了，减少key值对内存的开销【注：Redis的key值不会做字符串到数字的转换，所以这里节省的，仅仅是media:这6个字节的开销】。经过实验，内存占用会降到50MB，总的内存占用是15GB，是满足需求的，但是Instagram后面的改进任然有必要**）

于是Instagram的开发者向Redis的开发者之一Pieter Noordhuis询问优化方案，得到的回复是使用Hash结构。具体的做法就是将数据分段，每一段使用一个Hash结构存储，由于Hash结构会在单个Hash元素在不足一定数量时进行压缩存储，所以可以大量节约内存。这一点在上面的String结构里是不存在的。而这个一定数量是由配置文件中的hash-zipmap-max-entries参数来控制的。经过开发者们的实验，将hash-zipmap-max-entries设置为1000时，性能比较好，超过1000后HSET命令就会导致CPU消耗变得非常大。

于是他们改变了方案，将数据存成如下结构：

|  |
| --- |
| HSET "mediabucket:1155" "1155315" "939"  HGET "mediabucket:1155" "1155315"  > "939" |

通过取7位的图片ID的前四位为Hash结构的key值，保证了每个Hash内部只包含3位的key，也就是1000个。

再做一次实验，结果是每1,000,000个key只消耗了16MB的内存。总内存使用也降到了5GB，满足了应用需求。

（**NoSQLFan：同样的，这里我们还是可以再进行优化，首先是将Hash结构的key值变成纯数字，这样key长度减少了12个字节，其次是将Hash结构中的subkey值变成三位数，这又减少了4个字节的开销，如下所示。经过实验，内存占用量会降到10MB，总内存占用为3GB**）

|  |
| --- |
| HSET "1155" "315" "939"  HGET "1155" "315"  > "939" |

优化无止境，只要肯琢磨。希望你在使用存储产品时也能如此爱惜内存。

* + 1. aliredis



* + 1. TODO

1. KEY设计

设计key 为RemoteJob:id:taskId:bizKey:status:completeCode:nodeName value为id

的值(中间为各种值),可以进行简单的条件查询,排序什么的还是要靠sortSET（key\* 不支持高并发）

下面是思路，具体实现可参考gorm

@redisIndex 可以对字符串，数字，日期

对每个字段缓存，处理为 Object:Property Sort set zadd propertyName propertyValue id 通过ZREMRANGEBYSCORE可以获得关联的id

可以进行的操作有 = > < 或者区间

多属性 进行 想交获得最后的id值即可。

save

1、保存object到 hash；

2、将各个redisIndex的属性 保存到 sortSet

update

1、更新对象:更新hash；更新各个属性对应的sortSet

2、更新单个值：更新hash；更新属性对应的sortSet

delete

1、删除hash;

2、删除各个属性对应的sortSet

select

单表

getById:

getByProperty:

getByMutiProperties:

findByProperties;

多表关联

MySQL

1. MySQL基础
   1. rpm安装

rpm –ivh MySQL-Server-xxx.rpm

rpm –ivh MySQL-Client-xxx.rpm

默认安装目录：/usr/share/mysql

默认数据路径：/var/lib/mysql/

查看mysql状态: netstat –nlp 或者 ps -ef | grep mysql

查看数据表大小（进入数据目录）:du –sh test1.\*

启动与关闭：

* 命令行模式
  + cd /usr/bin
  + ./mysqld\_safe
  + mysqladmin –uroot shutdown
* 服务模式（RPM安装方式）
  + service mysql start
  + service mysql restart
  + service mysql stop

链接mysql: $mysql –uroot –p (-u跟数据库用户名 –p表示需要输入密码)

授予远程用户链接：

grant select,insert,update,delete on \*.\* to root@192.168.20.72 Identified by "root";

1）察看mysql是否在自动启动列表中

[root@test1 local]#　/sbin/chkconfig --list

2）把MySQL添加到你系统的启动服务组里面去

[root@test1 local]#　/sbin/chkconfig　–- add　mysql

3）把MySQL从启动服务组里面删除。

[root@test1 local]#　/sbin/chkconfig　–-del　mysql

* 1. 常用函数

concat(s1,s2,..sn):连接s1,s2,,sn为一个字符串

insert(str,x,y,instr):将字符串str从第x位置开始，y个字符长的子串替换为字符串instr

lower(str):小写 upper(str):大写

left(str,x):返回字符串str最左边的x个字符

right(str,x):返回字符串str最右边的x个字符

lpad(str,n,pad):用字符串pad对str最左边进行填充，直到长度为n个字符长度

rpad(str,n,pad):用字符串pad对str最右边进行填充，直到长度为n个字符长度

ltrim(str):去掉str左侧空格 rtrim(str):去掉str右侧字符串

repeat(str,x):返回str重复x次的结果；

replace(str,a,b):用字符串b替换字符串str中所有出现的字符串a

strcmp(s1,s2):比较字符串s1和s2,返回-1/0/1

trim(str)：去掉字符串行尾和行头的空格

substring(str,x,y):返回字符串str x位置起y个字符长度的字串

日期函数：

curdate()：返回当前日期 curtime():返回当前时间 now():返回当前日期和时间 unix\_timestampe(date): 返回日期date的unix时间戳 from\_unixtime 返回unix时间戳的日期值

week(date)：返回日期date为一年中的第几周 year(date): 返回日期date的年份 hour(time): 返回日期time的小时数 minute(time):返回time的分钟数 monthname(date): 返回date的月份名

date\_format(date,fmt): 返回按字符串fmt格式化日期date值

date\_add(date,interval expr type): 返回一个日期或者时间值加上一个时间间隔的时间值

datediff(expr,expr2): 返回起始时间expr和结束时间expr2之间的天数

if(value,t,f)

ifnull(value1,value2)

case when [value1] then [result1]..else [default]end

case [expr] when [value1] then [result1]..else [default]end

database() 当前数据库

now() 当前日期

version() 当前版本

MD5() SHA1() CRC32()

* 1. 存储引擎
* MyISAM
  + 默认引擎，不支持事务、也不支持外键；优势是访问的速度快、对事务完整性没有要求或者以SELECT、INSERT为主的应用基本上都可以使用该引擎
  + 存储成3个文件、其文件名与表名相同，扩展名：.frm(存储表定义)、MYD(MYData,存储数据)、MYI（MYIndex，存储索引），数据文件和索引文件可以放置在不同的目录。要指定索引文件和数据文件的路径，需要在创建表的时候通过DATA DIRECTORY和INDEX DIRECTORY语句指定，文件路径使用绝度路径。
  + 支持3种不同的存储格式：静态（固定长度）表；动态表；压缩表。
  + 静态表时默认的存储格式，静态表中的字段都是非变长字段，每个记录是固定长度。优点：存储非常迅速，容器缓存，出现故障容易恢复；缺点是占用的空间比动态表多。静态表在存储时候按照列宽度定义补足空格，但是应用访问时不会得到这些空格，但是如果保存的内容本来就有空格，在返回的时候也会被去掉，需要注意。
  + 动态表包含变长字段，记录不是固定长度，优点占用空间相对少，会产生碎片，需要定期执行OPTIMIZE TABLE语句或者myisamchk –r来改善性能，出现故障恢复比较困难
  + 压缩表有myisampack工具创建，占用非常小的磁盘空间，只有非常小的访问开支。
* InnoDB
  + 存储方式
    - 使用共享表空间存储，这种方式创建的表的表结构保存在.frm文件中，数据和索引保存在innodb\_data\_home\_dir和innodb\_data\_file\_path定义的表空间中，可以是多个文件。
    - 使用多表空间存储，这种方式创建的表的表结构仍然保存在.frm文件中，但是每个表的数据和索引单独保存在.ibd中，如果是个分区表，则每个分区对应单独的.ibd文件，文件名是“表名+分区名”，可以在创建分区的时候指定每个分区的数据文件的位置，依次来将表的IO均匀分布在多个磁盘上。
  + 使用多表空间的存储方式，需要设置参数innodb\_file\_per\_table,并重启服务才能生效，对新建表有效，修改回去后，已有的多表空间的表仍然保存原来的访问方式。
* 使用场景
  + MyISAM:默认的Mysql插件式存储引擎。如果应用是以读操作和写操作为主，只有很少的更新和删除操作，并且对事务的完整性、并发性要求不是很高，那么选择这个存储引擎是非常合适的。MyISAM是在Web、数据仓库和其他应用环境下最常使用的存储引擎之一。
  + InnoDB:用于事务处理应用程序，支持外键。如果应用程序对事务的完整性有比较高的要求，在并发条件下要求数据的一致性，数据操作除了插入和查询外，还包括很多的更新、删除操作，那么InnoDB存储引擎应该是比较合适的选择。InnoDB存储引擎除了有效地减低由于删除和更新导致的锁定，还可以确保事务的完整提交和回滚，对于类似计费系统或者财务系统等对数据准确性要求比较高的系统，InnoDB都是合适的选择。
  + MEMORY:将所有数据保存在RAM中，在需要快速定位记录和其他类似数据和环境下，可提供极快的访问。MEMORY的缺陷是对表的大小有限制，太大的表无法CACHE在内存中，其次是要确保表的数据可以恢复，数据库异常终止后表中的数据时可以恢复的。MEMORY表通常用于更新不太频繁的小表，用以快速得到访问结果。
  + MERGE:用于将一系列等同的MyISAM表以逻辑方式组合在一起，并作为一个对象引用它们。MERGE表的优点在于可以突破对单个MyISAM表大小的限制，并且通过将不同的表分布式在多个磁盘上，可以有效地改善MERGE表的访问效率。
  1. 数据类型
* 常用类型
  + 整数类型：tinyint、smallint、mediumint、int、integer、bigint
  + 浮点数类型：float、double
  + 定点数类型：dec(m,d)、decimal(m,d) (m表示总共多少位,d表示小数点后多少位)
  + 位类型：bit(m)(m:1~64) select bin(column)(显示二进制格式),hex(column)( 显示为十六进制格式) 读取bit的值
  + 字符串类型：
    - char(m):M为0~255
    - varchar(m):m为0-65535
    - tinyblob:允许长度0-255（字符串最长255个字符）
    - blog：允许长度0~65535
    - mediumblog:允许长度0~16772150
    - longblog:允许长度0~4294967295
    - tinytext:允许长度0~255
    - text：允许长度0~65535
    - mediumtext: 允许长度0~16772150
    - longtext: 允许长度0~4294967295
    - varbinary(m):允许长度0~m个字节的变长字节字符串
    - binary(m): 允许长度0~m个字节的定长字节字符串
* CHAR与VARCHAR
  + CHAR属于固定长度的字符类型，varchar属于可变长度的字符类型
  + 检索时char删除了尾部的空格
  + char是固定长度，处理速度比varchar快得多，缺点是浪费存储空间，程序需要对尾行空格进行处理，所以对于那些长度变化不大并且对查询速度有较高要求的数据可以考虑使用char类型来存储。
  + 不同的存储引擎对于char和varchar使用原则有不同
    - MyISAM引擎:建议使用固定长度的数据列代替可变长度的数据列
    - MEMORY引擎：目前都是使用固定长度的数据行存储，因此无论使用char或者varchar都没有关系。
    - InnoDB引擎：建议使用varchar类型。对于InnoDB数据表，内部的行存储格式没有区分固定长度和可变长度列，因此，使用char列不一定比使用varchar列的性能好。
* Text与BLOB
  + 删除大量BLOB数据后，采用optimzie table <表名>功能进行碎片整理
  + 采用合成索引(md5(blob)生成新字段)调高blob的查询效率
* 浮点数和定点数
  + float、double(real)用来表示浮点数，当插入数据的精度超过定义的实际精度，则插入值被四舍五入到实际定义的精度值，过程不会抛错
  + decimal(numberic)用来表示定点数，在默认SQLMode模式下，如果实际插入的数据精度大于实际定义的精度，MYSQL会进行警告并四舍五入插入，在TRADITIONAL（传统模式），系统直接抛错，无法插入
    - 浮点数存在误差问题
    - 对货币等对精度敏感的数据，应该用定点数表示存储；
    - 在编程中，如果用到浮点数，要特别注意误差问题，并尽量避免做浮点数比较；
  1. 字符集
* 设定字符集
  + my.conf中设置 [mysqld] default-character-set=gbk
  + 在启动项中指定：mysqld –default-character-set=gbk
  + 在编译的时候指定： ./configure –with-charset=gbk
  + 查看字符集：show variables like 'character\_set\_server';
  + 查看校队规则：show variables like 'collation\_server';
  1. TODO

1. TODO

RxJava

1. API说明
   1. Creating Observables
      1. Create

— create an Observable from scratch by calling observer methods programmatically

* 通过编程调用观察者方法从头开始创建一个可观察
  + 1. Defer

— do not create the Observable until the observer subscribes, and create a fresh Observable for each observer

— 直到有观察者订阅才创建可观察，同时为每个订阅者创建一个新的可观察

* + 1. Empty/Never/Throw

— create Observables that have very precise and limited behavior

— 创建非常精准和有限行为的观察者

* + 1. From

— convert some other object or data structure into an Observable

* + 1. Interval

— create an Observable that emits a sequence of integers spaced by a particular time interval

* + 1. Just

— convert an object or a set of objects into an Observable that emits that or those objects

* + 1. Range

— create an Observable that emits a range of sequential integers

* + 1. Repeat

— create an Observable that emits a particular item or sequence of items repeatedly

* + 1. Start

— create an Observable that emits the return value of a function

* + 1. Timer

— create an Observable that emits a single item after a given delay

bb

* 1. Transforming Observables
     1. Buffer — periodically gather items from an Observable into bundles and emit these bundles rather than emitting the items one at a time
     2. FlatMap — transform the items emitted by an Observable into Observables, then flatten the emissions from those into a single Observable
     3. GroupBy — divide an Observable into a set of Observables that each emit a different group of items from the original Observable, organized by key
     4. Map — transform the items emitted by an Observable by applying a function to each item
     5. Scan — apply a function to each item emitted by an Observable, sequentially, and emit each successive value
     6. Window — periodically subdivide items from an Observable into Observable windows and emit these windows rather than emitting the items one at a time
  2. Filtering Observables
     1. Debounce — only emit an item from an Observable if a particular timespan has passed without it emitting another item
     2. Distinct — suppress duplicate items emitted by an Observable
     3. ElementAt — emit only item n emitted by an Observable
     4. Filter — emit only those items from an Observable that pass a predicate test
     5. First — emit only the first item, or the first item that meets a condition, from an Observable
     6. IgnoreElements — do not emit any items from an Observable but mirror its termination notification
     7. Last — emit only the last item emitted by an Observable
     8. Sample — emit the most recent item emitted by an Observable within periodic time intervals
     9. Skip — suppress the first n items emitted by an Observable
     10. SkipLast — suppress the last n items emitted by an Observable
     11. Take — emit only the first n items emitted by an Observable
     12. TakeLast — emit only the last n items emitted by an Observable
  3. Combining Observables
     1. And/Then/When — combine sets of items emitted by two or more Observables by means of Pattern and Plan intermediaries
     2. CombineLatest — when an item is emitted by either of two Observables, combine the latest item emitted by each Observable via a specified function and emit items based on the results of this function
     3. Join — combine items emitted by two Observables whenever an item from one Observable is emitted during a time window defined according to an item emitted by the other Observable
     4. Merge — combine multiple Observables into one by merging their emissions
     5. StartWith — emit a specified sequence of items before beginning to emit the items from the source Observable
     6. Switch — convert an Observable that emits Observables into a single Observable that emits the items emitted by the most-recently-emitted of those Observables
     7. Zip — combine the emissions of multiple Observables together via a specified function and emit single items for each combination based on the results of this function
  4. Error Handling Operators
     1. Catch — recover from an onError notification by continuing the sequence without error
     2. Retry — if a source Observable sends an onError notification, resubscribe to it in the hopes that it will complete without error
  5. Observable Utility Operators
     1. Delay — shift the emissions from an Observable forward in time by a particular amount
     2. Do — register an action to take upon a variety of Observable lifecycle events
     3. Materialize/Dematerialize — represent both the items emitted and the notifications sent as emitted items, or reverse this process
     4. ObserveOn — specify the scheduler on which an observer will observe this Observable
     5. Serialize — force an Observable to make serialized calls and to be well-behaved
     6. Subscribe — operate upon the emissions and notifications from an Observable
     7. SubscribeOn — specify the scheduler an Observable should use when it is subscribed to
     8. TimeInterval — convert an Observable that emits items into one that emits indications of the amount of time elapsed between those emissions
     9. Timeout — mirror the source Observable, but issue an error notification if a particular period of time elapses without any emitted items
     10. Timestamp — attach a timestamp to each item emitted by an Observable
     11. Using — create a disposable resource that has the same lifespan as the Observable
  6. Conditional and Boolean Operators
     1. All — determine whether all items emitted by an Observable meet some criteria
     2. Amb — given two or more source Observables, emit all of the items from only the first of these Observables to emit an item
     3. Contains — determine whether an Observable emits a particular item or not
     4. DefaultIfEmpty — emit items from the source Observable, or a default item if the source Observable emits nothing
     5. SequenceEqual — determine whether two Observables emit the same sequence of items
     6. SkipUntil — discard items emitted by an Observable until a second Observable emits an item
     7. SkipWhile — discard items emitted by an Observable until a specified condition becomes false
     8. TakeUntil — discard items emitted by an Observable after a second Observable emits an item or terminates
     9. TakeWhile — discard items emitted by an Observable after a specified condition becomes false
  7. Mathematical and Aggregate Operators
     1. Average — calculates the average of numbers emitted by an Observable and emits this average
     2. Concat — emit the emissions from two or more Observables without interleaving them
     3. Count — count the number of items emitted by the source Observable and emit only this value
     4. Max — determine, and emit, the maximum-valued item emitted by an Observable
     5. Min — determine, and emit, the minimum-valued item emitted by an Observable
     6. Reduce — apply a function to each item emitted by an Observable, sequentially, and emit the final value
     7. Sum — calculate the sum of numbers emitted by an Observable and emit this sum
  8. Backpressure Operators
     1. backpressure operators — strategies for coping with Observables that produce items more rapidly than their observers consume them
  9. Connectable Observable Operators
     1. Connect — instruct a connectable Observable to begin emitting items to its subscribers
     2. Publish — convert an ordinary Observable into a connectable Observable
     3. RefCount — make a Connectable Observable behave like an ordinary Observable
     4. Replay — ensure that all observers see the same sequence of emitted items, even if they subscribe after the Observable has begun emitting items
  10. Operators to Convert Observables
      1. To — convert an Observable into another object or data structure

1. API 实例
2. ccc
3. ddd
4. ff