**Java学习总结**

# Java语法

## Instanceof 关键字

[**Java**](http://lib.csdn.net/base/java) 中的instanceof 运算符是用来在运行时指出对象是否是特定类的一个实例。instanceof通过返回一个布尔值来指出，这个对象是否是这个特定类或者是它的子类的一个实例。  
 用法：  
result = object instanceof class  
参数：  
Result：布尔类型。  
Object：必选项。任意对象表达式。  
Class：必选项。任意已定义的对象类。  
说明：  
如果 object 是 class 的一个实例，则 instanceof 运算符返回 true。如果 object 不是指定类的一个实例，或者 object 是 null，则返回 false。

## Java的native关键字

一. 什么是Native Method  
   简单地讲，一个Native Method就是一个java调用非java代码的接口。一个Native Method是这样一个java的方法：该方法的实现由非java语言实现，比如C。这个特征并非java所特有，很多其它的编程语言都有这一机制，比如在C＋＋中，你可以用extern "C"告知C＋＋编译器去调用一个C的函数。  
   "A native method is a Java method whose implementation is provided by non-java code."  
   在定义一个native method时，并不提供实现体（有些像定义一个java interface），因为其实现体是由非java语言在外面实现的。，下面给了一个示例：

1. package java.lang;
2. **public** **class** Object {
3. ......
4. **public** final native Class<?> getClass();
5. **public** native **int** hashCode();
6. **protected** native Object clone() throws CloneNotSupportedException;
7. **public** final native **void** notify();
8. **public** final native **void** notifyAll();
9. **public** final native **void** wait(**long** timeout) throws InterruptedException;
10. ......
11. }

        标识符native可以与所有其它的java标识符连用，但是abstract除外。这是合理的，因为native暗示这些方法是有实现体的，只不过这些实现体是非java的，但是abstract却显然的指明这些方法无实现体。native与其它java标识符连用时，其意义同非Native Method并无差别。

    一个native method方法可以返回任何java类型，包括非基本类型，而且同样可以进行异常控制。这些方法的实现体可以自制一个异常并且将其抛出，这一点与java的方法非常相似。  
    native method的存在并不会对其他类调用这些本地方法产生任何影响，实际上调用这些方法的其他类甚至不知道它所调用的是一个本地方法。JVM将控制调用本地方法的所有细节。

    如果一个含有本地方法的类被继承，子类会继承这个本地方法并且可以用java语言重写这个方法（这个似乎看起来有些奇怪），同样的如果一个本地方法被fianl标识，它被继承后不能被重写。  
   本地方法非常有用，因为它有效地扩充了jvm.事实上，我们所写的java代码已经用到了本地方法，在sun的java的并发（多线程）的机制实现中，许多与操作系统的接触点都用到了本地方法，这使得java程序能够超越java运行时的界限。有了本地方法，java程序可以做任何应用层次的任务。  
二.为什么要使用Native Method  
   java使用起来非常方便，然而有些层次的任务用java实现起来不容易，或者我们对程序的效率很在意时，问题就来了。  
   与java环境外交互：  
   有时java应用需要与java外面的环境交互。这是本地方法存在的主要原因，你可以想想java需要与一些底层系统如操作系统或某些硬件交换信息时的情况。本地方法正是这样一种交流机制：它为我们提供了一个非常简洁的接口，而且我们无需去了解java应用之外的繁琐的细节。  
   与操作系统交互：  
   JVM支持着java语言本身和运行时库，它是java程序赖以生存的平台，它由一个解释器（解释字节码）和一些连接到本地代码的库组成。然而不管怎样，它毕竟不是一个完整的系统，它经常依赖于一些底层（underneath在下面的）系统的支持。这些底层系统常常是强大的操作系统。通过使用本地方法，我们得以用java实现了jre的与底层系统的交互，甚至JVM的一些部分就是用C写的，还有，如果我们要使用一些java语言本身没有提供封装的操作系统的特性时，我们也需要使用本地方法。  
    Sun's Java  
    Sun的解释器是用C实现的，这使得它能像一些普通的C一样与外部交互。jre大部分是用java实现的，它也通过一些本地方法与外界交互。例如：类java.lang.Thread 的 setPriority()方法是用java实现的，但是它实现调用的是该类里的本地方法setPriority0()。这个本地方法是用C实现的，并被植入JVM内部，在Windows 95的平台上，这个本地方法最终将调用Win32 SetPriority() API。这是一个本地方法的具体实现由JVM直接提供，更多的情况是本地方法由外部的动态链接库（external dynamic link library）提供，然后被JVM调用。  
三.JVM怎样使Native Method跑起来：  
    我们知道，当一个类第一次被使用到时，这个类的字节码会被加载到内存，并且只会回载一次。在这个被加载的字节码的入口维持着一个该类所有方法描述符的list，这些方法描述符包含这样一些信息：方法代码存于何处，它有哪些参数，方法的描述符（public之类）等等。  
    如果一个方法描述符内有native，这个描述符块将有一个指向该方法的实现的指针。这些实现在一些DLL文件内，但是它们会被操作系统加载到java程序的地址空间。当一个带有本地方法的类被加载时，其相关的DLL并未被加载，因此指向方法实现的指针并不会被设置。当本地方法被调用之前，这些DLL才会被加载，这是通过调用java.system.loadLibrary()实现的。  
   最后需要提示的是，使用本地方法是有开销的，它丧失了java的很多好处。如果别无选择，我们可以选择使用本地方法。

**可以将native方法比作Java程序同Ｃ程序的接口，其实现步骤：  
１、在Java中声明native()方法，然后编译；  
２、用javah产生一个.h文件；  
３、写一个.cpp文件实现native导出方法，其中需要包含第二步产生的.h文件（注意其中又包含了JDK带的jni.h文件）；  
４、将第三步的.cpp文件编译成动态链接库文件；  
５、在Java中用System.loadLibrary()方法加载第四步产生的动态链接库文件，这个native()方法就可以在Java中被访问了。**

自己写了一个调用VB的DLL的例子，不过还没写完，调用函数部分不会写，只写了加载DLL的过程

JAVA部分：

1. **public** **class** TestNativeMothod {
2. **public** **static** **void** main(String[] args) {
4. //PROCESS LOGIC
6. }
8. **public** native String callNativeMothod();
10. **static** {
11. System.loadLibrary("TestNativeMethodProj");
12. }
13. }

VB部分：

1. Option Explicit
2. Public Function callNativeMothod() As String
3. callNativeMothod = "This is a method implemented by VB"
4. End Function

# 一、学习Spring框架的学习

## Spring框架

Spring是一个开源框架，Spring是于2003 年兴起的一个轻量级的Java 开发框架，由Rod Johnson创建。简单来说，Spring是一个分层的JavaSE/EEfull-stack(一站式) 轻量级开源框架。

### 1.1 Spring框架

Spring框架主要负责技术整合（可以整合很多技术），该框架提供IoC和AOP机制，基于这些特性整合，可以降低系统组件之间的耦合度，便于系统组件的维护、扩展和替换。

### 1.2 Spring框架的模块组成

1）Spring Core：核心容器，BeanFactory提供了组件生命周期的管理，组件的创建，装配，销毁等功能

2）SpringContext：ApplicationContext，扩展核心容器，提供事件处理、国际化等功能。它提供了一些企业级服务的功能，提供了JNDI，EJB，RMI的支持。

3）Spring AOP：提供切面支持

4）Spring DAO：提供事务支持，JDBC，DAO支持

5）Spring ORM：对流行的O/R Mapping封装或支持

6）Spring Web：提供Web应用上下文，对Web开发提供功能上的支持，如请求，表单，异常等。

7）Spring Web MVC：全功能MVC框架，作用等同于Struts。

**Spring框架的两大核心：IOC容器（也叫依赖注入DI）和AOP。**

### 1.3 IOC容器

Spring容器负责bean生成、组装、销毁组件，并负责事件处理、国际化等功能。

实现方式有：

(1) BeanFactory<interface>

① 核心容器，负责组件生成和装配

② 实现主要包括Xml BeanFactory

(2) ApplicationContext（常用）

(3) WebApplicationContext

**IOC的使用**

Resource：interface，用来包装资源

xmlBeanFactory：BeanFactory的一个实现，使用Resource对象来查找配置文件

BeanFactory.gerBean（“BeanId”）：取得以参数命名，或者Id等于参数值的一个Bean实例。

BeanFactory（容器）在默认情况下，会采用singleton方式（单例方式）返回对象。容器只到调用getBean方法时，才会实例化某个对象。

1）Spring可以采用XML或者在XML中引用.properties文件作配置

2）配置文件（XML）主要信息

a、导入需要的命名空间

b、声明bean

c、装配bean

d、配置aop切面

3）简单注入bean属性

a、<constructor-arg value="15"/> 通过带参数的构造方法注入

b、<property name="song" value="Juggle Bells"/> 通过setter方法注入

如果Bean属性不多，并且属性值必须要注入才能使用，则应该采用constructor注入，其他情况就要set方法注入。

4）使用Spring的命名空间P装配属性

需要在配置前声明p命名空间，p命名空间是另一种Bean的装配方式，配置时不需要过多的尖括号。

5） 注入集合类型

① Set

Set中存放字符串，对象，集合，不允许重复

② List

List中可以放入字符串，对象，允许重复

③ Map

Map 有<entry>子元素来存取key和value，其中key和value可以是任意类型

④ Properties

Properties有<props>子元素，其中key和value必须都是string类型

6）自动装配（Autowring Properties）

装配方式：byName，byType，autodetect，constructor

注：自动装配与手动装配可以结合使用，手动装配会覆盖自动装配。

a、autowire=“byName”：按照名称匹配，按照Bean的Id与属性的名称进行匹配

b、autowire=“byType”：按照类型匹配

注：要注入的属性类型与配置文件中的Bean类型匹配的个数超过一个，会出错。

c、autowire=“constructor”：先去匹配构造器中参数类型，后与配置文件中的Bean类型匹配。

d、autowire=“antodetect”：先按照construct，后按照byType。

7）比较两种装配方式（简单注入和自动装配）

关于自动匹配：

优点：快速开发

缺点：依赖关系不清楚，易出错，不易维护。

自动匹配的应用场合：

a、构建系统原型

b、与依赖关系检查（Dependency-check）结合使用

c、自动装配与手动装配结合

8）练习xml配置代码

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd">  
 <bean id="duke" class="com.spring.action.idol.Juggler">  
 <constructor-arg value="15"/>  
 </bean>  
 <bean id="sonnet29" class="com.spring.action.idol.Sonnet29"/>  
 *<!-- 装配poemJuggler -->* <bean id="poemticduke" class="com.spring.action.idol.PoemJuggler">  
 <constructor-arg ref="sonnet29"></constructor-arg>  
 <constructor-arg value="10"></constructor-arg>  
 </bean>  
 *<!--使用静态工厂方法getInstance()返回舞台实例 -->* <bean id="theStage" class="com.spring.action.idol.Stage"  
 factory-method="getInstance"/>  
 *<!-- init-method 和 detroy-method方法 -->* <bean id="auditorium"  
 class="com.spring.action.idol.Auditorium"  
 init-method="turnOnLights"  
 destroy-method="turnOffLights"/>  
 *<!-- setter and getter()方法注入bean，构造器注入bean测试 -->* <bean id="kenny" class="com.spring.action.idol.Instrumentlist">  
 <property name="song" value="Juggle Bells"/>  
 *<!-- ref 引用其他bean -->* <property name="instrument" ref="piano"/>  
 </bean>  
 <bean id="guitar" class="com.spring.action.idol.Guitar"/>  
 <bean id="piano" class="com.spring.action.idol.Piano"></bean>  
</beans>

9）添加依赖关系

<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>4.3.8.RELEASE</version>  
</dependency>

### 1.4 Spring中数据源配置

无论选择Spring的哪种数据访问方式，你都需要配置一个数据源的引用。Spring提供了在Spring上下文中配置数据源bean的多种方式，包括：

1）通过JDBC驱动程序定义的数据源；

2）通过JNDI查找的数据源；

3）连接池的数据源（建议使用）

#### 1.4.1 使用JDDI数据源

说明:JndiObjectFactoryBean 能够通过JNDI获取DataSource   
*<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">   
     <property name="jndiName">*

*<value>*[***Java***](http://lib.csdn.net/base/17)*:comp/env/jdbc/roseindiaDB\_local</value>*

*</property>   
</bean>*

#### 1.4.2 使用连接池的数据源（推荐）

尽管Spring并没有提供数据源连接池实现，但是我们有多项可用的方案，包括如下开源的实现：  
 Apache Commons DBCP (http://jakarta.apache.org/commons/dbcp)；  
 c3p0 (http://sourceforge.net/projects/c3p0/) ；  
 BoneCP (http://jolbox.com/) 。

如配置DBCP BasicDataSource的方式：

*<!--使用连接池配置数据源-->*<bean id="dataSource" destroy-method="close" class="org.apache.commons.dbcp.BasicDataSource">  
 <property name="driverClassName" value="${jdbc.driverClassName}"/>  
 <property name="url" value="${jdbc.url}"/>  
 <property name="username" value="${jdbc.user}"/>  
 <property name="password" value="${jdbc.password}"/>  
</bean>

#### 1.4.3 使用JDBC配置数据源

在Spring中，通过JDBC驱动定义数据源是最简单的配置方式。 Spring提供了三个这样的数据源类（均位于org.springframework.jdbc.datasource包中）供选择：  
 DriverManagerDataSource：

在每个连接请求时都会返回一个新建的连接。与DBCP的BasicDataSource不同，由DriverManagerDataSource提供的连接并没有进行池化管理；  
 SimpleDriverDataSource：  
 与DriverManagerDataSource的工作方式类似，但是它直接使用JDBC驱动，来解决在特定环境下的类加载问题，这样的环境包括OSGi容器；  
 SingleConnectionDataSource：

在每个连接请求时都会返回同一个的连接。尽管SingleConnectionDataSource不是严格意义上的连接池数据源，但是你可以将其视为只有一个连接的池。

例如使用org.springframework.jdbc.datasource.DriverManagerDataSource 配置方式：  
 说明：DriverManagerDataSource建立连接是只要有连接就新建一个connection,根本没有连接池的作用。   
<bean

id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName">

<value>${jdbc.driverClassName}</value>

</property>   
    <property name="url"><value>${jdbc.url}</value></property>   
    <property name="username"><value>${jdbc.username}</value></property>   
    <property name="password"><value>${jdbc.password}</value></property>   
</bean>

尽管这些数据源对于小应用或开发环境来说是不错的，但是要将其用于生产环境，你还是需要慎重考虑。因为SingleConnectionDataSource有且只有一个数据库连接，所以不适合用于多线程的应用程序，最好只在测试的时候使用。而DriverManagerDataSource和SimpleDriverDataSource尽管支持多线程，但是在每次请求连接的时候都会创建新连接，这是以性能为代价的。鉴于以上的这些限制，我强烈建议应该使用数据源连接池。

## 2. Spring MVC + hibernate

在JSP中定义变量并进行引用

如下：

<%--脚本程序的语法格式：<% java代码片段 %> --%>  
**<%** int count = 10;  
 String s = "Your IP address is " + request.getRemoteAddr();  
**%>**<%-- 引用上面定义的变量: <%=变量名%> --%>  
<p>count: **<%=**count**%>**</p>  
<p>**<%=**s**%>**</p>

# 二、Protobuf的学习

下载地址：<http://repo1.maven.org/maven2/com/google/protobuf/protoc/>

ProtocolBuffer是用于结构化数据串行化的灵活、高效、自动的方法，类似于XML，不过它更小、更快、也更简单。可以定义自己的数据结构，然后使用代码生成器生成的代码来读写这个数据结构。无需重新部署程序的情况下更新数据结构。

使用方法：

1）准备好两个文件： protoc-3.2.exe和protobuf-Java-2.4.1.jar

在依赖中添加依赖包：

<dependency>  
 <groupId>com.google.protobuf</groupId>  
 <artifactId>protobuf-java</artifactId>  
 <version>3.3.0</version>  
</dependency>

2）创建一个proto文件，如Person.proto

如account.proto文件内容：

syntax ="proto3";  
package com.jump.account.base.protobuf;  
option java\_outer\_classname ="AccountOuterClass";  
message AccountMessage {  
 int32 id = 1;  
 string name = 2;  
 string message = 3;  
 string url = 4;  
 string user\_name = 5;  
 bytes password = 6;  
}

### 将proto文件生成java文件

将Person.proto放在与刚解压的protoc.exe同级目录中，如D:\Users下

在cmd窗口中，到D:\Users文件夹下，执行protoc.exe --java\_out=./ Person.proto 则可以找到的一个生成的MsgInfo.java文件。

如生成c++文件使用命令：protoc --cpp\_out=d:\ person.proto

1. protobuf消息定义

字段格式：

**限定修饰符 数据类型 字段名称 = 字段编码值 [字段默认值]**

**限定修饰符** required , optional, repeated

Required: 表示是一个**必须字段**，必须相对于发送方，在发送消息之前设置该字段的值，对于接收方，必须能够识别该字段的意思。发送之前没有设置required字段或者无法识别required字段都会引发编解码异常，导致消息被丢弃。

Optional：表示是一个**可选字段**，可选对于发送方，在发送消息时，可以有选择性的设置或者不设置该字段的值。对于接收方，如果能够识别可选字段就进行相应的处理，如果无法识别，则忽略该字段，消息中的其它字段正常处理。---因为optional字段的特性，很多接口在升级版本中都把后来添加的字段都统一的设置为optional字段，这样老的版本无需升级程序也可以正常的与新的软件进行通信，只不过新的字段无法识别而已，因为并不是每个节点都需要新的功能，因此可以做到按需升级和平滑过渡。

Repeated：表示该字段**可以包含0~N个元素**。其特性和optional一样，但是每一次可以包含多个值。可以看作是在传递一个数组的值。

**数据类型**

|  |  |  |  |
| --- | --- | --- | --- |
| 数据类型 | 描述 | 打包 | C++语言映射 |
| bool | 布尔类型 | 1字节 | bool |
| double | 64位浮点数 | N | double |
| float | 32为浮点数 | N | float |
| int32 | 32位整数、 | N | int |
| uin32 | 无符号32位整数 | N | unsigned int |
| int64 | 64位整数 | N | \_int64 |
| uint64 | 64为无符号整 | N | unsigned \_int64 |
| sint32 | 32位整数，处理负数效率更高 | N | int32 |
| sing64 | 64位整数 处理负数效率更高 | N | \_int64 |
| fixed32 | 32位无符号整数 | 4 | unsigned int32 |
| fixed64 | 64位无符号整数 | 8 | unsigned \_int64 |
| sfixed32 | 32位整数、能以更高的效率处理负数 | 4 | unsigned int32 |
| sfixed64 | 64为整数 | 8 | unsigned \_int64 |
| string | 只能处理 ASCII字符 | N | std::string |
| bytes | 用于处理多字节的语言字符、如中文 | N | std::string |
| enum | 可以包含一个用户自定义的枚举类型uint32 | N(uint32) | enum |
| message | 可以包含一个用户自定义的消息类型 | N | object of class |

注意：N 表示打包的字节并不是固定。而是根据数据的大小或者长度。

**字段名称：**

protobuf建议字段的命名采用以下划线分割的驼峰式。例如 first\_name 而不是firstName

**字段编码值**

有了该值，通信双方才能互相识别对方的字段。当然相同的编码值，其限定修饰符和数据类型必须相同。

编码值的取值范围为 1~2^32（4294967296）。

其中 1~15的编码时间和空间效率都是最高的，编码值越大，其编码的时间和空间效率就越低（相对于1-15），当然一般情况下相邻的2个值编码效率的是相同的，除非2个值恰好实在4字节，12字节，20字节等的临界区。比如15和16.

1900~2000编码值为Google protobuf 系统内部保留值，建议不要在自己的项目中使用。

protobuf 还建议把经常要传递的值把其字段编码设置为1-15之间的值。

注意：消息中的字段编码值无需连续，但不能在同一个消息中有字段包含相同的编码值。

建议：项目投入运营以后涉及到版本升级时的新增消息字段全部使用optional或者repeated，尽量不实用required。如果使用了required，需要全网统一升级，如果使用optional或者repeated可以平滑升级。

**默认值**

当在传递数据时，如果用户没有设置值，则使用默认值传递到对端。

1. **protobuf中的关键字**

**关于import**

protobuf 接口文件可以像C语言的h文件一个，分离为多个，在需要的时候通过 import导入需要对文件。其行为和C语言的#include或者java的import的行为大致相同。

**关于package**

避免名称冲突，可以给每个文件指定一个package名称，对于java解析为java中的包。对于C++则解析为名称空间。

**关于message**

支持嵌套消息，消息可以包含另一个消息作为其字段。也可以在消息内定义一个新的消息。

**关于enum**

枚举值必须是大于等于0的整数，使用分号(;)分隔枚举变量。

# 三、MongoDB数据库的学习

### 1、介绍

MongoDB 是由C++语言编写的，是一个基于分布式文件存储的开源nosql型数据库系统。

MongoDB 旨在为WEB应用提供可扩展的高性能数据存储解决方案。

MongoDB 将数据存储为一个文档，数据结构由键值(key=>value)对组成。MongoDB 文档格式为BSON （二进制格式）对象。字段值可以包含其他文档，数组及文档数组。

MongoDB官网：​<https://www.mongodb.org/>

### 2、与关系型数据库（sql）比较

|  |  |  |
| --- | --- | --- |
| **SQL术语/概念** | **MongoDB术语/概念** | **解释/说明** |
| database | database | 数据库 |
| table 表格 | collection 集合 | 数据库表/集合 |
| row 行 | document 文档 | 数据记录行/文档 |
| column 列 | field 域 | 数据字段/域 |
| index | index | 索引 |
| table joins |  | 表连接,MongoDB不支持 |
| primary key | primary key | 主键,MongoDB自动将\_id字段设置为主键 |

|  |  |
| --- | --- |
| **数据库服务和客户端** | |
| Mysqld/Oracle | mongod |
| mysql/sqlplus | mongo |

**文档**

文档是一组键值(key-value)对(即BSON)。MongoDB 的文档不需要设置相同的字段，并且相同的字段不需要相同的数据类型，这与关系型数据库有很大的区别，也是 MongoDB 非常突出的特点。

**需要注意的地方**：

文档中的键/值对是有序的。

文档中的值不仅可以是在双引号里面的字符串，还可以是其他几种数据类型（甚至可以是整个嵌入的文档)。

MongoDB区分类型和大小写。

MongoDB的文档不能有重复的键。

文档的键是字符串。除了少数例外情况，键可以使用任意UTF-8字符。

**文档键命名规范**：

键不能含有\0 (空字符)。这个字符用来表示键的结尾。

.和$有特别的意义，只有在特定环境下才能使用。

以下划线"\_"开头的键是保留的(不是严格要求的)。

**2）集合**

集合就是 MongoDB 文档组，类似于 RDBMS （关系数据库管理系统：Relational Database Management System)中的表格。

注意集合存在于数据库中，集合没有固定的结构

**集合的命名要求：**

集合名不能是空字符串""；

集合名不能含有\0字符（空字符)，这个字符表示集合名的结尾；

集合名不能以"system."开头，这是为系统集合保留的前缀；

用户创建的集合名字不能含有保留字符。有些驱动程序的确支持在集合名里面包含$，这是因为某些系统生成的集合中包含该字符$。除非你要访问这种系统创建的集合，否则千万不要在名字里出现$。

**元数据**

数据库的信息是存储在集合中。它们使用了系统的命名空间：

dbname.system.\*

在MongoDB数据库中名字空间 <dbname>.system.\* 是包含多种系统信息的特殊集合(Collection)，如下:

|  |  |
| --- | --- |
| **集合命名空间** | **描述** |
| dbname.system.namespaces | 列出所有名字空间。 |
| dbname.system.indexes | 列出所有索引。 |
| dbname.system.profile | 包含数据库概要(profile)信息。 |
| dbname.system.users | 列出所有可访问数据库的用户。 |
| dbname.local.sources | 包含复制对端（slave）的服务器信息和状态。 |

### 3、MongoDB 的数据类型

|  |  |
| --- | --- |
| **数据类型** | **描述** |
| String | 字符串。存储数据常用的数据类型。在 MongoDB 中，UTF-8 编码的字符串也是合法的。 |
| Integer | 整型数值。用于存储数值。根据采用的服务器，可分为 32 位或 64 位。 |
| Boolean | 布尔值。用于存储布尔值（真/假）。 |
| Double | 双精度浮点值。用于存储浮点值。 |
| Min/Max keys | 将一个值与 BSON（二进制的 JSON）元素的最低值和最高值相对比。 |
| Arrays | 用于将数组或列表或多个值存储为一个键。 |
| Timestamp | 时间戳。记录文档修改或添加的具体时间。 |
| Object | 用于内嵌文档。 |
| Null | 用于创建空值。 |
| Symbol | 符号。该数据类型基本上等同于字符串类型，但不同的是，它一般用于采用特殊符号类型的语言。 |
| Date | 日期时间。用 UNIX 时间格式来存储当前日期或时间。你可以指定自己的日期时间：创建 Date 对象，传入年月日信息。 |
| Object ID | 对象 ID。用于创建文档的 ID。 |
| Binary Data | 二进制数据。用于存储二进制数据。 |
| Code | 代码类型。用于在文档中存储 JavaScript 代码。 |
| Regular expression | 正则表达式类型。用于存储正则表达式。 |

### 4、MongoDB中常用命令

1、Help查看命令提示 help

2、切换/创建数据库

use yourDB; 当创建一个集合(table)的时候会自动创建当前数据库

3、查询所有数据库

show dbs;

4、删除当前使用数据库

db.dropDatabase();

5、从指定主机上克隆数据库

db.cloneDatabase(“127.0.0.1”); 将指定机器上的数据库的数据克隆到当前数据库

6、从指定的机器上复制指定数据库数据到某个数据库

db.copyDatabase("mydb", "temp", "127.0.0.1");将本机的mydb的数据复制到temp数据库中

7、修复当前数据库

db.repairDatabase();

8、查看当前使用的数据库

db.getName();

db; db和getName方法是一样的效果，都可以查询当前使用的数据库

9、显示当前db状态

db.stats();

10、当前db版本

db.version();

11、查看当前db的链接机器地址

db.getMongo();

**Collection聚集集合**

创建一个聚集集合（table）

db.createCollection(“collName”, {size: 20, capped: 5, max: 100});

2、得到指定名称的聚集集合（table） db.getCollection("account");

3、得到当前db的所有聚集集合 db.getCollectionNames();

4、显示当前db所有聚集索引的状态 db.printCollectionStats();

5、查询所有记录db.userInfo.find();

**索引**

1、创建索引db.userInfo.ensureIndex();

2、查询当前聚集集合所有索引db.userInfo.getIndexes();

3、查看总索引记录大小db.userInfo.totalIndexSize();

4、读取当前集合的所有index信息db.users.reIndex();

5、删除指定索引db.users.dropIndex("name");

6、删除所有索引索引db.users.dropIndexes();

**修改、添加、删除集合数据**

1、添加

db.users.save();添加的数据的数据列，没有固定，根据添加的数据为准

 2、修改

db.collection.update(criteria, objNew, upsert, multi )

criteria:update的查询条件，类似sql update查询内where后面的

objNew:update的对象和一些更新的操作符（如$,$inc...）等，也可以理解为sql update查询内set后面的。

upsert : 如果不存在update的记录，是否插入objNew,true为插入，默认是false，不插入。

multi : mongodb默认是false,只更新找到的第一条记录，如果这个参数为true,就把按条件查出来多条记录全部更新。

如：db.users.update({name: 'Lisi'}, {$inc: {age: 50}}, false, true);

相当于：update users set age = age + 50 where name = ‘Lisi’;

再如：db.users.update({name: 'Lisi'}, {$inc: {age: 50}, $set: {name: 'hoho'}}, false, true);

相当于：update users set age = age + 50, name = ‘hoho’ where name = ‘Lisi’;

3、删除db.users.remove();

4、查询修改db.users.findAndModify();

#### 比较/$in/$nin/$or/$not

mongo通find来查找文档。可以执行精确匹配和模糊匹配。

比较：> $gt , >= $gte, < $lt, <= $lte, != $ne

> db.tianyc02.find()

{ "\_id" : ObjectId("50ea6eba12729d90ce6e3423"), "name" : "xttt", "age" : 111 }

{ "\_id" : ObjectId("50ea6eba12729d90ce6e3424"), "name" : "xttt", "age" : 222 }

{ "\_id" : ObjectId("50ea6b6f12729d90ce6e341b"), "name" : "xtt", "age" : 11 }

{ "\_id" : ObjectId("50ea6b7312729d90ce6e341c"), "name" : "xtt", "age" : 22 }

> db.tianyc02.find({age:{$lt:100}})

{ "\_id" : ObjectId("50ea6b6f12729d90ce6e341b"), "name" : "xtt", "age" : 11 }

{ "\_id" : ObjectId("50ea6b7312729d90ce6e341c"), "name" : "xtt", "age" : 22 }

> db.tianyc02.find({age:{$lt:100,$gt:20}})

{ "\_id" : ObjectId("50ea6b7312729d90ce6e341c"), "name" : "xtt", "age" : 22 }

> db.tianyc02.find({age:{$ne:11}})

{ "\_id" : ObjectId("50ea6eba12729d90ce6e3423"), "name" : "xttt", "age" : 111 }

{ "\_id" : ObjectId("50ea6eba12729d90ce6e3424"), "name" : "xttt", "age" : 222 }

{ "\_id" : ObjectId("50ea6b7312729d90ce6e341c"), "name" : "xtt", "age" : 22 }

$in & $nin

> db.tianyc02.find()

{ "\_id" : ObjectId("50ea6eba12729d90ce6e3423"), "name" : "xttt", "age" : 111 }

{ "\_id" : ObjectId("50ea6eba12729d90ce6e3424"), "name" : "xttt", "age" : 222 }

{ "\_id" : ObjectId("50ea6b6f12729d90ce6e341b"), "name" : "xtt", "age" : 11 }

{ "\_id" : ObjectId("50ea6b7312729d90ce6e341c"), "name" : "xtt", "age" : 22 }

> db.tianyc02.find({age:{$in:[11,22]}})

{ "\_id" : ObjectId("50ea6b6f12729d90ce6e341b"), "name" : "xtt", "age" : 11 }

{ "\_id" : ObjectId("50ea6b7312729d90ce6e341c"), "name" : "xtt", "age" : 22 }

> db.tianyc02.find({age:{$nin:[11,22]}})

{ "\_id" : ObjectId("50ea6eba12729d90ce6e3423"), "name" : "xttt", "age" : 111 }

{ "\_id" : ObjectId("50ea6eba12729d90ce6e3424"), "name" : "xttt", "age" : 222 }

$or

> db.tianyc02.find({$or:[{age:11},{age:22}]})

{ "\_id" : ObjectId("50ea6b6f12729d90ce6e341b"), "name" : "xtt", "age" : 11 }

{ "\_id" : ObjectId("50ea6b7312729d90ce6e341c"), "name" : "xtt", "age" : 22 }

> db.tianyc02.find({$or:[{age:11},{name:'xttt'}]})

{ "\_id" : ObjectId("50ea6eba12729d90ce6e3423"), "name" : "xttt", "age" : 111 }

{ "\_id" : ObjectId("50ea6eba12729d90ce6e3424"), "name" : "xttt", "age" : 222 }

{ "\_id" : ObjectId("50ea6b6f12729d90ce6e341b"), "name" : "xtt", "age" : 11 }

$not

> db.tianyc02.find({age:{$mod:[11,0]}})

{ "\_id" : ObjectId("50ea6b6f12729d90ce6e341b"), "name" : "xtt", "age" : 11 }

{ "\_id" : ObjectId("50ea6b7312729d90ce6e341c"), "name" : "xtt", "age" : 22 }

> db.tianyc02.find({age:{$not:{$mod:[11,0]}}})

{ "\_id" : ObjectId("50ea6eba12729d90ce6e3423"), "name" : "xttt", "age" : 111 }

{ "\_id" : ObjectId("50ea6eba12729d90ce6e3424"), "name" : "xttt", "age" : 222 }

$mod会将查询的值除以第一个给定的值，若余数等于第二个给定的值，则返回该结果。

$not与正则表达式联合使用时极为有效，用来查找那些与特定模式不匹配的文档。

### 5、在Centos7上安装MongoDB数据库

注意：推荐使用

1、下载

wget https://fastdl.mongodb.org/linux/mongodb-linux-x86\_64-rhel70-3.2.8.tgz

2、解压

tar zxvf MongoDB-Linux-x86\_64-rhel70-3.2.8.tgz

mv mongodb-linux-x86\_64-rhel70-3.2.8 /home/ mongodb

cd mongodb/

mkdir db

mkdir logs

cd bin/

3、编辑配置文件：

vim mongodb.conf

输入以下内容：

*dbpath=/home/mongodb/db*

*logpath=/home/mongodb/logs/mongodb.log*

*#是否追加日志*

*logappend = true*

*bind\_ip=127.0.0.1*

*port=27017*

*#是否以后台进程启动*

*fork=true*

*#是否支持HTTP形式访问*

*nohttpinterface = true*

*#是否启动授权认证*

*auth = true*

4、创建新的账号：

groupadd mongodb

useradd mongodb -g mongodb

cd /

chown -R mongodb:mongodb mongodb

5、启动：

**/home/mongodb/bin/mongod -f /home/mongodb/bin/mongodb.conf**

6、设置开机自动启动mongodb

vi /etc/rc.d/rc.local

/home/mongodb/bin/mongod -f /home/mongodb/bin/mongodb.conf

赋予可执行权限

chmod +x /etc/rc.d/rc.local

7、将mongodb/bin中的命令加入系统环境变量中

1）临时有效：

[root@centos7]# export PATH=$PATH:/home/mongodb/bin/

2）永久有效，写入/etc/profile配置文件中，在最后加入下面一行

PATH=$PATH:/home/mongodb/bin/

使配置生效

[root@centos7 init.d]# source /etc/profile

查看

[root@centos7 ~]# echo $PATH

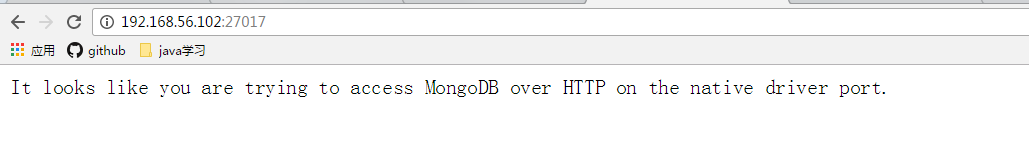
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/home/mongodb/bin/:/root/bin

8、进入mongodb的shell模式：

/home/mongodb/bin/mongo 或mongo

9、如何证明数据库已经启动？

可以打开浏览器，输入http://192.168.56.102:27017/，显示如下：证明数据库已经启动。



### 6、windows上安装

Windows上cmd查看端口占用情况：

netstat |findstr "27017"

安装过程参考博客：

<http://www.cnblogs.com/lzrabbit/p/3682510.html>

cmd启动

D:\Users\MongoDB1\bin>mongod --dbpath D:\Users\MongoDB1\data

### 7、复制集模式 replica Sets

参考博客

<http://blog.csdn.net/yyywyr/article/details/27235857>

1）介绍

复制集是一个带有故障转移的主从集群。是从现有的主从模式演变而来，增加了自动故障转移和节点成员自动恢复。

复制集模式中没有固定的主结点，在启动后，多个服务节点间将自动选举产生一个主结点。该主结点被称为primary，一个或多个从结点被称为secondaries。primary结点基本上就是master结点，不同之处在于primary结点在不同时间可能是不同的服务器。如果当前的主结点失效了，复制集中的其余结点将会试图选出一个新的主结点。

在一个复制集集群中，各个服务器有以下几种状态：

Primary 主节点，一个复制集有且仅有一台服务器处于Primary状态，只有主节点才对外提供读写服务。如果主节点挂掉，复制集将会投票选出一个备用节点成为新的主节点。

Secondary 备用节点，复制集允许有多台Secondary，每个备用节点的数据与主节点的数据是完全同步的。

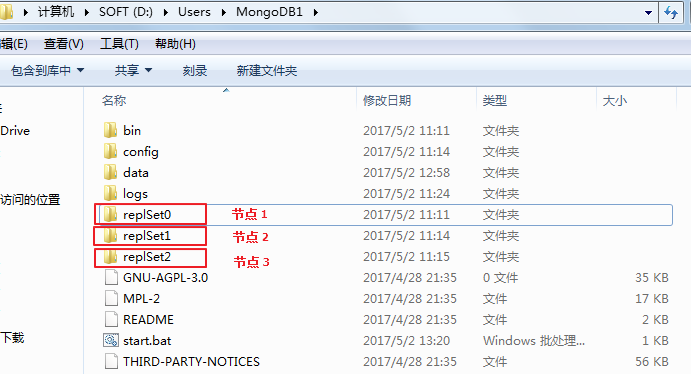
Recovering 恢复中，当复制集中某台服务器挂掉或者掉线后数据无法同步，重新恢复服务后从其他成员复制数据，这时就处于恢复过程，数据同步后，该节点又回到备用状态。

Arbiter 仲裁节点，该类节点可以不用单独存在，如果配置为仲裁节点，就主要负责在复本集中监控其他节点状态，投票选出主节点。该节点将不会用于存放数据。如果没有仲裁节点，那么投票工作将由所有节点共同进行。

Down 无效节点，当服务器挂掉或掉线时就会处于该状态。

2）部署

在windows上通过使用不同目录模拟了3个节点，端口分别是20000,20001,20002。



在cmd窗口中通过如下命令分别启动3个节点：

mongod --port "20000" --dbpath "D:\Users\MongoDB1\replSet0\data" --replSet "rs0"

打开任意节点，如节点1，执行如下内容

> config = {\_id:"rs0", members:[

... {\_id:0,host:"127.0.0.1:20000"},

... {\_id:1,host:"127.0.0.1:20001"},

... {\_id:2,host:"127.0.0.1:20002",arbiterOnly:true}

... ]

... }

> rs.initiate(config)

{ "ok" : 1 }

调用rs.conf()可查看配置。

SECONDARY节点默认是不可读的。需要通过在连接时指定或者在主库指定slaveOk，从而使SECONDARY分担读的压力，PRIMARY只承担写操作，可通过下面两个方法：

第一种：在从节点设置slaveOk（）：

rs.slave()

第二种：在主节点设置：

http://img.blog.csdn.net/20140527220330375?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQveXl5d3ly/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

### 8、分片

1）创建目录

mkdir -p /home/mongodb/mongos/shard1/data

mkdir -p /home/mongodb/mongos/shard2/data

mkdir -p /home/mongodb/mongos/shard3/data

mkdir -p /home/mongodb/mongos/config/data

mkdir -p /home/mongodb/logs

2）启动shard数据节点

mongod --dbpath /home/mongodb/mongos/shard1/data --logpath /home/mongodb/logs/shard1.log --shardsvr --port 20001 --fork

mongod --dbpath /home/mongodb/mongos/shard2/data --logpath /home/mongodb/logs/shard2.log --shardsvr --port 20002 --fork

mongod --dbpath /home/mongodb/mongos/shard3/data --logpath /home/mongodb/logs/shard3.log --shardsvr --port 20003 --fork

3）准备config节点（复制集形式，复制集名称为rs0）

mongod --dbpath /home/mongodb/mongos/config/data/ --logpath /home/mongodb/logs/config.log --port 20004 --configsvr --replSet rs0 --fork

初始化复制集

mongo localhost:20004

> use admin

switched to db admin

> cfg = {

... \_id:'rs0',

... configsvr:true,

... members:[

... {\_id:0,host:'localhost:20004'}

... ]

... };

> rs.initiate(cfg);

4）准备router节点（注意configdb采用复制集形式）

mongos --configdb rs0/localhost:20004 --logpath /home/mongodb/logs/route.log --port 20000 --fork

连接mongos节点

mongo localhost:20000

增加分片

mongos> sh.addShard('localhost:20001');

{ "shardAdded" : "shard0000", "ok" : 1 }

mongos> sh.addShard('localhost:20002');

{ "shardAdded" : "shard0001", "ok" : 1 }

mongos> sh.addShard('localhost:20003');

{ "shardAdded" : "shard0002", "ok" : 1 }

mongos>

查看分片信息

mongos> sh.status();

--- Sharding Status ---

sharding version: {

"\_id" : 1,

"minCompatibleVersion" : 5,

"currentVersion" : 6,

"clusterId" : ObjectId("59277fab02d770918e77900d")

}

shards:

{ "\_id" : "shard0000", "host" : "localhost:20001", "state" : 1 }

{ "\_id" : "shard0001", "host" : "localhost:20002", "state" : 1 }

{ "\_id" : "shard0002", "host" : "localhost:20003", "state" : 1 }

active mongoses:

"3.4.3" : 1

autosplit:

Currently enabled: yes

balancer:

Currently enabled: yes

Currently running: no

Balancer lock taken at Fri May 26 2017 09:06:51 GMT+0800 (CST) by ConfigServer:Balancer

Failed balancer rounds in last 5 attempts: 0

Migration Results for the last 24 hours:

No recent migrations

databases:

mongos> exit

5）测试

连接route节点，并创建开启分片的数据库test，和分片集合user

mongo localhost:20000

mongos> use test

switched to db test

mongos> sh.enableSharding('test');

{ "ok" : 1 }

mongos> sh.shardCollection('test.user',{uid:1})

mongos> for(i=-100;i<1000;i++){db.user.insert({uid:i,username:'test-'+i});}

WriteResult({ "nInserted" : 1 })

mongos> sh.status()

--- Sharding Status ---

sharding version: {

"\_id" : 1,

"minCompatibleVersion" : 5,

"currentVersion" : 6,

"clusterId" : ObjectId("59277fab02d770918e77900d")

}

shards:

{ "\_id" : "shard0000", "host" : "localhost:20001", "state" : 1 }

{ "\_id" : "shard0001", "host" : "localhost:20002", "state" : 1 }

{ "\_id" : "shard0002", "host" : "localhost:20003", "state" : 1 }

active mongoses:

"3.4.3" : 1

autosplit:

Currently enabled: yes

balancer:

Currently enabled: yes

Currently running: no

Balancer lock taken at Fri May 26 2017 09:06:51 GMT+0800 (CST) by ConfigServer:Balancer

Failed balancer rounds in last 5 attempts: 0

Migration Results for the last 24 hours:

1 : Success

databases:

{ "\_id" : "test", "primary" : "shard0000", "partitioned" : true }

test.user

shard key: { "uid" : 1 }

unique: false

balancing: true

chunks:

shard0000 2

shard0001 1

{ "uid" : { "$minKey" : 1 } } -->> { "uid" : -99 } on : shard0001 Timestamp(2, 0)

{ "uid" : -99 } -->> { "uid" : -83 } on : shard0000 Timestamp(2, 1)

{ "uid" : -83 } -->> { "uid" : { "$maxKey" : 1 } } on : shard0000 Timestamp(1, 3)

mongos>

### 9、实战：

<dependency>  
 <groupId>org.mongodb</groupId>  
 <artifactId>mongo-java-driver</artifactId>  
 <version>3.4.2</version>  
</dependency>

#### Update类的方法

Update类可以使用一些“语法糖”，因为它的方法意在链接在一起，您可以通过静态方法public static Update update(String key, Object value)和静态导入来启动创建新的Update实例。

以下是Update类上的方法列表

Update addToSet (String key, Object value)更新使用$addToSet更新修饰符

Update currentDate (String key)更新使用$currentDate更新修饰符

Update currentTimestamp (String key)更新使用$currentDate更新修饰符与$type timestamp

Update inc (String key, Number inc)使用$inc更新修饰符更新

Update max (String key, Object max)更新使用$max更新修饰符

Update min (String key, Object min)更新使用$min更新修饰符

Update 乘以 (String key, Number multiplier)更新使用$mul更新修饰符

Update pop (String key, Update.Position pos)更新使用$pop更新修饰符

Update (String key, Object value)使用$pull更新修饰符拉取更新

Update pullAll (String key, Object[] values) Update使用$pullAll更新修饰符

Update (String key, Object value)使用$push更新修饰符推送更新

Update (String key, Object[] values)使用$pushAll更新修饰符的pushAll更新

Update (String oldName, String newName)使用$rename更新修饰符重命名更新

Update (String key, Object value)使用$set更新修饰符设置更新

Update setOnInsert (String key, Object value) Update使用$setOnInsert更新修饰符

Update (String key)使用$unset更新修饰符取消设置更新

#### 标准类的方法 Criteria

Criteria all (Object o)使用$all运算符创建标准

Criteria and (String key)增加了一个链接Criteria与指定key为当前Criteria和返回新创建的一个

Criteria andOperator (Criteria…​ criteria)使用$and运算符创建并查询所有提供的条件（需要MongoDB 2.0或更高版本）

Criteria elemMatch (Criteria c)使用$elemMatch运算符创建标准

Criteria exists (boolean b)使用$exists运算符创建条件

Criteria gt (Object o)使用$gt运算符创建标准

Criteria gte (Object o)使用$gte运算符创建标准

Criteria in (Object…​ o)使用$in运算符为varargs参数创建标准。

Criteria in (Collection<?> collection)使用$in运算符使用集合创建标准

Criteria is (Object o)使用$is运算符创建标准

Criteria lt (Object o)使用$lt操作符创建标准

Criteria lte (Object o)使用$lte操作符创建标准

Criteria mod (Number value, Number remainder)使用$mod运算符创建标准

Criteria ne (Object o)使用$ne运算符创建标准

Criteria nin (Object…​ o)使用$nin操作符创建标准

Criteria norOperator (Criteria…​ criteria)使用$nor运算符为所有提供的条件创建或查询

Criteria not ()使用$not影响直接跟随的子句的元运算符创建标准

Criteria orOperator (Criteria…​ criteria)使用$or运算符创建或查询所有提供的条件

Criteria 正则表达式 (String re)使用a创建标准$regex

Criteria size (int s)使用$size运算符创建标准

Criteria type (int t)使用$type运算符创建条件

#### Query类的方法

Query addCriteria (Criteria criteria)用于向查询添加附加条件

Field 字段 ()用来定义字段被包括在查询结果中

Query 限制 (int limit)用于将返回结果的大小限制为提供的限制（用于分页）

Query 跳过 (int skip)用于跳过结果中提供的文档数量（用于分页）

Query 与 (Sort sort)用于为结果提供定义排序

#### 集合分片生成命令：

分片时需要指定分片的key， 语法为

sh.shardCollection("<database>.<collection>", shard-key-pattern) :

实例1：生成采用分片的node\_status集合

sh.shardCollection("key\_store.node\_status",{"\_id":"hashed"})

#### 8.5 mongo排序和查询

mongodb.core.query.Criteria 查询非空(ne)和排序(sort)

实例1：

1. query.addCriteria(Criteria.where("modelId").ne("").ne(**null**));
2. query.with(**new** Sort(**new** Order(Direction.DESC, "ct")));

若多个字段排序：第一排序按照sort降序，第二排序按照ct降序

1. query.with(**new** Sort(**new** Order(Direction.DESC, "sort")).and(**new** Sort(Direction.DESC,"ct")));

实例2：按照createTime降序排序  
 query.with(new Sort(new Order(Sort.Direction.*DESC*, "createTime")));

//findOne返回第一个匹配的对象实例

mongoTemplate.findOne(query, NodeEntity.class)

mongo shell 中的分析explain()命令：

V3.0版本以上，需要增加参数才会显示执行时间等。命令参考如下：

>db.getCollection('order').find({}).explain("executionStats")

{

"queryPlanner" : {

"plannerVersion" : 1,

"namespace" : "OrdersDB.order",

"indexFilterSet" : false,

"parsedQuery" : {},

"winningPlan" : {

"stage" : "COLLSCAN",

"direction" : "forward"

},

"rejectedPlans" : []

},

"executionStats" : {

"executionSuccess" : true,

"nReturned" : 4,

"executionTimeMillis" : 0,

"totalKeysExamined" : 0,

"totalDocsExamined" : 4,

"executionStages" : {

"stage" : "COLLSCAN",

"nReturned" : 4,

"executionTimeMillisEstimate" : 0,

"works" : 6,

"advanced" : 4,

"needTime" : 1,

"needYield" : 0,

"saveState" : 0,

"restoreState" : 0,

"isEOF" : 1,

"invalidates" : 0,

"direction" : "forward",

"docsExamined" : 4

}

},

"serverInfo" : {

"host" : "centos7",

"port" : 27017,

"version" : "3.4.3",

"gitVersion" : "f07437fb5a6cca07c10bafa78365456eb1d6d5e1"

},

"ok" : 1.0

}

# 四、Idea开发工具学习

IntelliJ IDEA 2017破解方法：

**在激活的时候选择 license server;**

**输入如下地址激活：**

**http://idea.imsxm.com**

### 1、Git使用

介绍

Git是一个开源的分布式版本控制系统，可以有效、高速的处理从很小到非常大的项目版本管理。[

Git的作用可以这样理解：我们在开发一个项目时，通过将项目所在的根目录创建成一个Git仓库（也就是版本库），以后只要我们更新项目信息（需要是文本类型），比如更新或添加代码，Git仓库会帮助我们记录所有更改的文本文件及其更改内容，每一次的改动都相当于项目的一个版本，我们可以备注每个版本，方便我们通过Git日志查看每个版本信息，当然最大的好处是可以通过Git恢复到任意的版本。有时候编写代码时会忘记改了哪些内容，令人很烦恼，有了Git之后，就可以随时查看修改的内容了，非常方便和实用。

Git常用命令

git add file 增加文件到git仓库

git commit –m “添加修改信息提示” 讲文件提交到仓库并进行说明

git status 查看当前git仓库的状态，如是否修改了某个文件

git diff file 查看file文件修改前后发生变化的部分

git log 查看git提交记录

git checkout branch 切换分支

git reset –hard HEAD^ 回退到上一个版本，（注：HEAD^^时代表上一个版本的上一个版本，或者HEAD^ 用git log中查到的版本号代替，恢复到对应版本）

#### 配置用户名和邮箱的命令

1 git config --global user.name "youname"

2 git config --global user.email "youeamil@email.com"

最后可以通过 **git config -l** 这个命令查看已配置的用户名和邮箱信息

#### 1.1 centos7上安装git私有仓库过程

在服务器上创建一个新用户

**adduser**git

创建一个Git仓库

**cd** /data/git //切换到希望创建工作区的目录

git init --bare **sample**.git //创建一个Git仓库,此时Git会创建一个唯一分支master

chown -R git:git **sample**.git//将sample.git的所有权从root修改到新创建的git

注意：若不使用RSA认证，配置完此处便初步搭建完成，可进行远程clone。

在服务器端打开RSA认证，在文件/etc/ssh/sshd\_config中添加下列三行内容:

RSAAuthentication yes

PubkeyAuthentication yes

AuthorizedKeysFile .ssh/authorized\_keys

创建用于存放客户端公钥的文件

cd /home/git //切换到刚刚创建的git用户目录下

mkdir .ssh

vim authorized\_keys //创建用于存放客户端公钥的文件

在客户端创建SSH key

**ssh-keygen** **-t** **rsa** **-C** "**sample**@**geekzhou**.**com**"

该命令会在用户主目录的.ssh文件夹下生成两个文件:用于保存私钥的id\_rsa和用于保存公钥的id\_rsa.pub

把步骤4生成的公钥导入服务器，将生成的id\_rsa.pub中的内容拷贝到服务器文件/home/git/.ssh/authorized\_keys

在客户端clone远程git仓库

git clone git@115.28.222.133:/data/git/sample.git

其中115.28.222.133是服务器端的IP地址

(由于进行了步骤4和步骤5的设置,所以在clone仓储的时候就无需再输入密码了)

禁止用户git通过shell进行登录

由于进行了步骤4和步骤5的设置,此时客户端可以通过ssh git@115.28.222.133 直接登录到服务器,如需禁止,修改文件/etc/passwd:

将git:x:1000:1000::/home/git:/bin/bash

修改为:git:x:1000:1000::/home/git:/usr/bin/git-shell

初始化master分支

在进行了一个commit操作后,master分支就会出现,此时执行git push就可以将本地的修改推送到远程的master分支了.

至此远程git服务器搭建完毕!

### 2、JUnit4测试工具

参考博客：<http://blog.csdn.net/feihong247/article/details/7828143>

JUnit4通过注解的方式来识别测试方法。目前支持的主要注解有：

@BeforeClass 全局只会执行一次，而且是第一个运行

@Before 在测试方法运行之前运行

@Test 测试方法

@After 在测试方法运行之后允许

@AfterClass 全局只会执行一次，而且是最后一个运行

@Ignore 忽略此方法

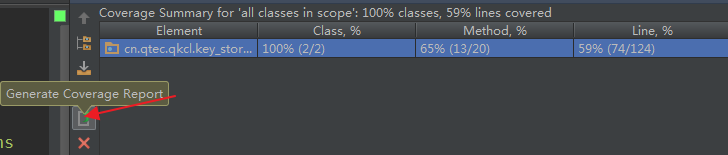
**覆盖率在编辑器中的颜色显示：**

没有被测试覆盖到的代码都将高亮显示为红色。覆盖的代码颜色则是绿色。如果一些代码是只覆盖部分,那没将显示为黄色。

#### 2.1 添加依赖关系：

<dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.12</version>  
 <scope>test</scope>  
</dependency>

#### 2.2覆盖率结果生成报告



#### 2.3单元测试：

原则：

1）代码运行前后数据库内容保持一致；

2）保证每次运行都是成功的。

单元测试类类名前需要进行如下注解：

@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration("classpath\*:\*.xml")  
public class TestBlogDao {

//code  
 }

#### 2.4 获取每个测试方法名称

@Rule  
public TestName testName = new TestName();

@Before  
public void setUp() throws Exception {  
 System.*out*.println(testName.getMethodName());  
}

### 3、Assert断言

Junit4都提供了一个Assert类（虽然package不同，但是大致差不多）。Assert类中定义了很多静态方法来进行断言。列表如下：

assertTrue(String message, boolean condition) 要求condition == true

assertFalse(String message, boolean condition) 要求condition == false

fail(String message) 必然失败，同样要求代码不可达

assertEquals(String message, XXX expected,XXX actual) 要求expected.equals(actual)

assertArrayEquals(String message, XXX[] expecteds,XXX [] actuals) 要求expected.equalsArray(actual)

assertNotNull(String message, Object object) 要求object!=null

assertNull(String message, Object object) 要求object==null

assertSame(String message, Object expected, Object actual) 要求expected == actual

assertNotSame(String message, Object unexpected,Object actual) 要求expected != actual

assertThat(String reason, T actual, Matcher matcher) 要求matcher.matches(actual) == true

### 4、Idea中年常用的快捷键

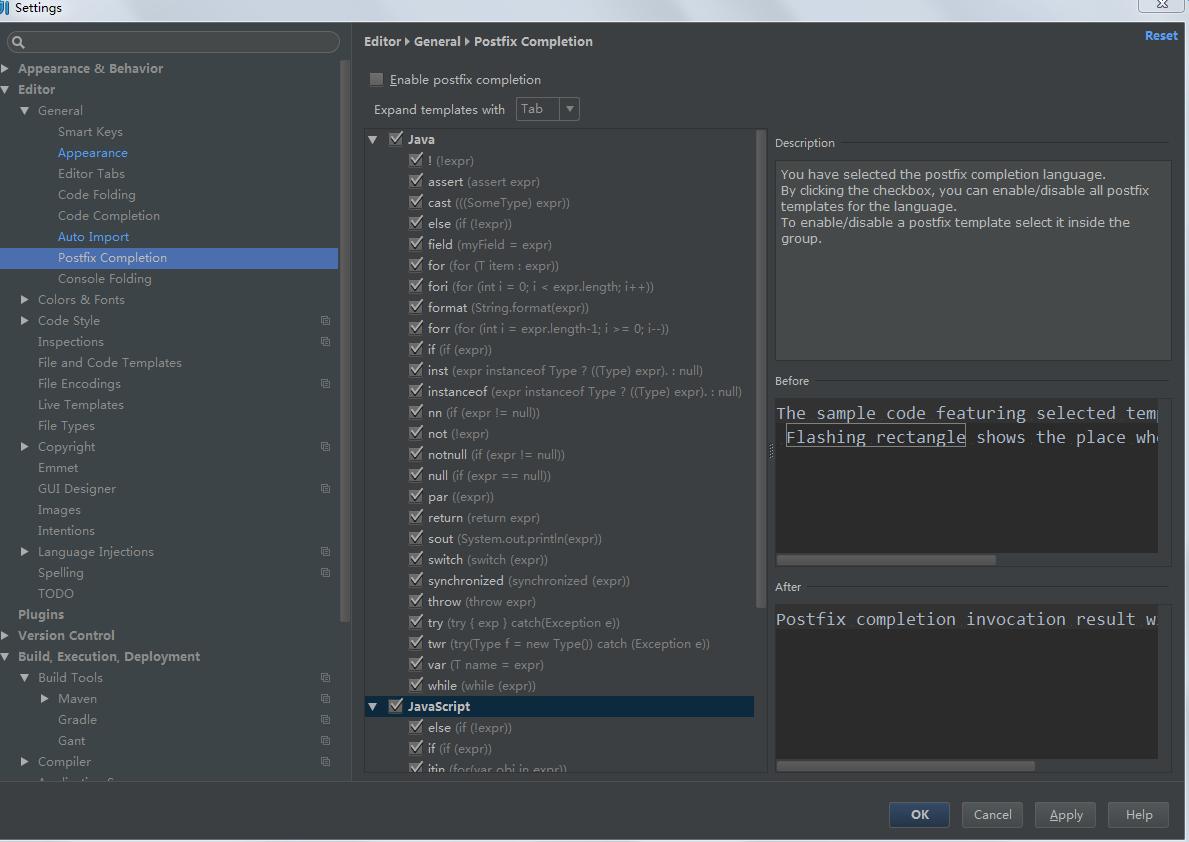
1）快捷键或输入 结果

sout 自动System.out.println()

psvm 自动生成public static void main

fori 自动for (int i = 0; i < ; i++) {}

/\*\* 自动生成方法的注释



2）在maven项目pom.xml中添加依赖关系

<dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.12</version>  
 <scope>test</scope>  
</dependency>

### 5、maven项目

内置变量

1. ${basedir} 项目根目录
2. ${project.build.directory} 构建目录，缺省为target
3. ${project.build.outputDirectory} 构建过程输出目录，缺省为target/classes
4. ${project.build.finalName} 产出物名称，缺省为${project.artifactId}-${project.version}
5. ${project.packaging} 打包类型，缺省为jar
6. ${project.xxx} 当前pom文件的任意节点的内容

#### 5.1 Maven项目常用plugin配置

1）自定义变量和设置JDK版本

<properties>  
 <!-- 设置jdk版本 项目编码-->  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <maven.compiler.source>1.8</maven.compiler.source>  
 <maven.compiler.target>1.8</maven.compiler.target>  
 <netty.version>4.1.15.Final</netty.version>  
 <spring.version>4.3.8.RELEASE</spring.version>  
 <slf4j.version>1.7.25</slf4j.version>  
 <protobuf.version>3.2.0</protobuf.version>  
 <common-codec.version>1.9</common-codec.version>  
</properties>

2）plugins常用配置

<build>  
 <plugins>  
 <!-- 定义项目的 jdk 版本-->  
 <plugin>  
 <artifactId>maven-compiler-plugin</artifactId>  
 <version>3.7.0</version>  
 <configuration>  
 <source>1.8</source>  
 <target>1.8</target>  
 </configuration>  
 </plugin>  
  
 <!-- 复制依赖的jar包到指定的文件夹里 -->  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-dependency-plugin</artifactId>  
 <version>3.0.1</version>  
 <executions>  
 <execution>  
 <id>copy-dependencies</id>  
 <phase>package</phase>  
 <goals>  
 <goal>copy-dependencies</goal>  
 </goals>  
 <configuration>  
 <outputDirectory>${project.build.directory}/lib</outputDirectory>  
 </configuration>  
 </execution>  
 </executions>  
 </plugin>  
  
 <!-- 打成 jar 包时，设定manifest 的参数，如指定运行的Main Class，依赖的jar包，加入classpath中 -->  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-jar-plugin</artifactId>  
 <version>2.4</version>  
 <configuration>  
 <archive>  
 <manifest>  
 <addClasspath>true</addClasspath>  
 <classpathPrefix>/data/lib</classpathPrefix>  
 <mainClass>cn.zhp.netty.custom.server.CustomServer</mainClass>  
 </manifest>  
 </archive>  
 </configuration>  
 </plugin>  
  
 <!-- 在maven中运行Ant任务，比如在打包阶段，对文件进行复制 -->  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-antrun-plugin</artifactId>  
 <version>1.7</version>  
 <executions>  
 <execution>  
 <phase>package</phase>  
 <goals>  
 <goal>run</goal>  
 </goals>  
 <configuration>  
 <target name="copy">  
 <delete>  
 <fileset dir="target" includes="\*.properties"></fileset>  
 </delete>  
 <copy todir="target">  
 <fileset dir="files"></fileset>  
 </copy>  
 </target>  
 </configuration>  
 </execution>  
 </executions>  
 </plugin>  
  
 <!-- 用于一键部署，把本地打包的jar文件，上传到远程服务器，并执行服务器上的shell命令 -->  
 <plugin>  
 <groupId>org.codehaus.mojo</groupId>  
 <artifactId>wagon-maven-plugin</artifactId>  
 <version>1.0</version>  
 <configuration>  
 <serverId>crawler</serverId>  
 <fromDir>target</fromDir>  
 <includes>\*.jar,\*.properties,\*.sh</includes>  
 <!-- 修改远程IP -->  
 <url>sftp://59.110.162.178/home/zhangxianhe</url>  
 <!-- 修改命令 -->  
 <commands>  
 <command>chmod 755 /home/zhangxianhe/update.sh</command>  
 <command>/home/zhangxianhe/update.sh</command>  
 </commands>  
 <displayCommandOutputs>true</displayCommandOutputs>  
 </configuration>  
 </plugin>  
  
 <!-- 用于远程部署 Java Web 项目 -->  
 <plugin>  
 <groupId>org.apache.tomcat.maven</groupId>  
 <artifactId>tomcat7-maven-plugin</artifactId>  
 <version>2.2</version>  
 <configuration>  
 <url>http://59.110.162.178:8080/manager/text</url>  
 <username>linjinbin</username>  
 <password>linjinbin</password>  
 </configuration>  
 </plugin>  
  
 <!-- 用于把多个jar包，打成1个jar包,  
 一般Java项目都会依赖其他第三方jar包，最终打包时，希望把其他jar包包含在一个jar包里 -->  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-shade-plugin</artifactId>  
 <version>2.4.3</version>  
 <executions>  
 <execution>  
 <phase>package</phase>  
 <goals>  
 <goal>shade</goal>  
 </goals>  
 <configuration>  
 <transformers>  
 <transformer  
 implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">  
 <manifestEntries>  
 <Main-Class>com.meiyou.topword.App</Main-Class>  
 <X-Compile-Source-JDK>${maven.compile.source}</X-Compile-Source-JDK>  
 <X-Compile-Target-JDK>${maven.compile.target}</X-Compile-Target-JDK>  
 </manifestEntries>  
 </transformer>  
 </transformers>  
 </configuration>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
</build>

### 6、mvn命令：

mvn clean compile 编译

mvn clean test 测试

mvn clean package 打包

mvn clean install 把生成的jar/war包复制到本地repository(就是~/.m2/repository下面)

mvn clean deploy 把生成的jar/war包发送到远程repository（建议配置了私服，那就是往私服发送了）

mvn cargo:run 通过cargo插件，把生成的war包部署到本地服务器，并启动。（注意要先运行 mvn clean package 打包）

mvn cargo:redeploy 通过cargo插件，把生成的war包部署到远程服务器：如果已经有了，就先undeploy再deploy，如果没有直接deploy

### 7、maven项目中的pom.xml使用点滴

#### 7.1定义jdk版本

<build>  
 <plugins>  
 <!-- 定义项目的 jdk 版本-->  
 <plugin>  
 <artifactId>maven-compiler-plugin</artifactId>  
 <configuration>  
 <source>1.8</source>  
 <target>1.8</target>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

### 8.Linux 运行jar包命令如下：

#### 方式一：

1. java -jar XXX.jar

特点：当前ssh窗口被锁定，可按CTRL + C打断程序运行，或直接关闭窗口，程序退出

那如何让窗口不锁定？

#### 方式二：

1. java -jar XXX.jar &

&代表在后台运行。

特定：当前ssh窗口不被锁定，但是当窗口关闭时，程序中止运行。

继续改进，如何让窗口关闭时，程序仍然运行？

#### 方式三：

nohup [**Java**](http://lib.csdn.net/base/javase) -jar XXX.jar &

nohup 意思是不挂断运行命令,当账户退出或终端关闭时,程序仍然运行

当用 nohup 命令执行作业时，缺省情况下该作业的所有输出被重定向到nohup.out的文件中，除非另外指定了输出文件。

#### 方式四：

1. nohup java -jar XXX.jar >temp.txt &

解释下 >temp.txt

command >out.file

command >out.file是将command的输出重定向到out.file文件，即输出内容不打印到屏幕上，而是输出到out.file文件中。

可通过jobs命令查看后台运行任务

1. jobs

那么就会列出所有后台执行的作业，并且每个作业前面都有个编号。  
如果想将某个作业调回前台控制，只需要 fg + 编号即可。

1. fg 23

查看某端口占用的线程的pid

netstat -nlp |grep :9181

#### 8.1查看服务器的连接数/有效连接数

$ netstat -na|wc -l #连接数

$ netstat -nat|grep ESTABLISHED|wc -l #有效连接数,中间的参数 ESTABLISHED表示有效的连接数！

$ netstat -n | awk '/^tcp/ {++S[$NF]} END {for(a in S) print a, S[a]}' #对各种状态的连接数分组统计结果

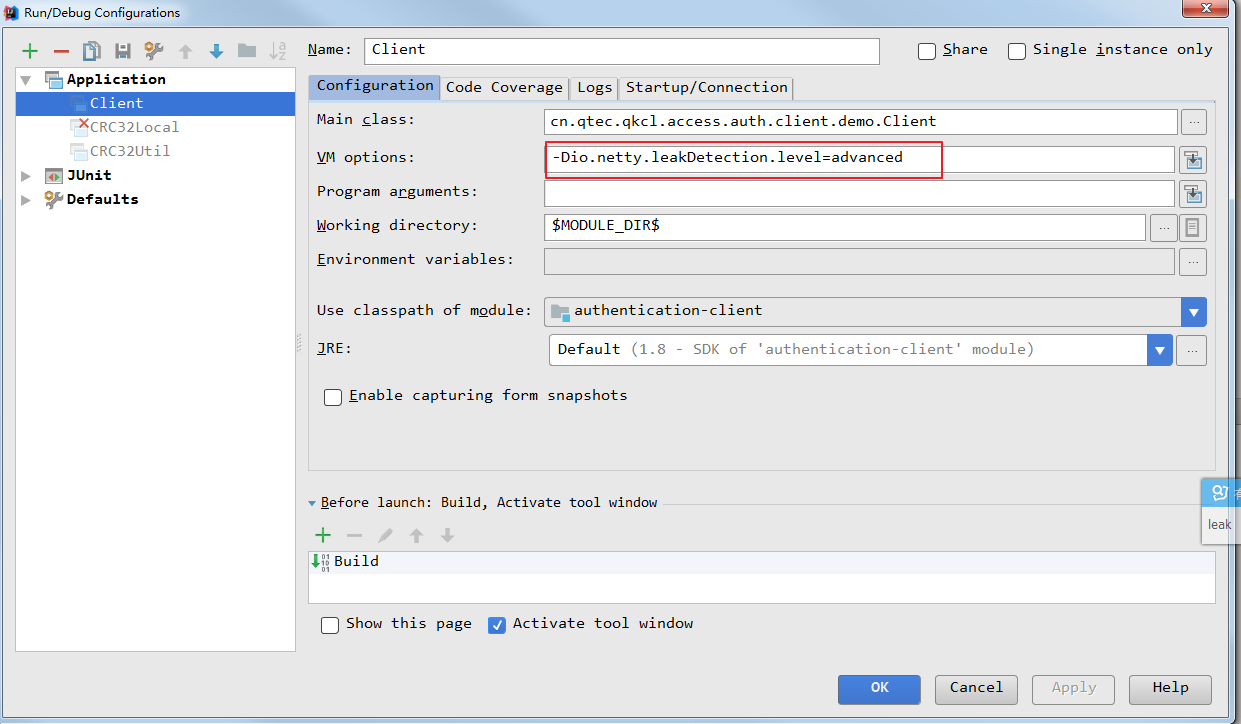
### 9. java的System.getProperty()方法可以获取的值

|  |  |
| --- | --- |
| java.version | Java 运行时环境版本 |
| java.vendor | Java 运行时环境供应商 |
| java.vendor.url | Java 供应商的 URL |
| java.home | Java 安装目录 |
| java.vm.specification.version | Java 虚拟机规范版本 |
| java.vm.specification.vendor | Java 虚拟机规范供应商 |
| java.vm.specification.name | Java 虚拟机规范名称 |
| java.vm.version | Java 虚拟机实现版本 |
| java.vm.vendor | Java 虚拟机实现供应商 |
| java.vm.name | Java 虚拟机实现名称 |
| java.specification.version | Java 运行时环境规范版本 |
| java.specification.vendor | Java 运行时环境规范供应商 |
| java.specification.name | Java 运行时环境规范名称 |
| java.class.version | Java 类格式版本号 |
| java.class.path | Java 类路径 |
| java.library.path | 加载库时搜索的路径列表 |
| java.io.tmpdir | 默认的临时文件路径 |
| java.compiler | 要使用的 JIT 编译器的名称 |
| java.ext.dirs | 一个或多个扩展目录的路径 |
| os.name | 操作系统的名称 |
| os.arch | 操作系统的架构 |
| os.version | 操作系统的版本 |
| file.separator | 文件分隔符（在 UNIX 系统中是“/”） |
| path.separator | 路径分隔符（在 UNIX 系统中是“:”） |
| line.separator | 行分隔符（在 UNIX 系统中是“/n”） |
| user.name | 用户的账户名称 |
| user.home | 用户的主目录 |
| user.dir | 用户的当前工作目录 |

### 10. 启动程序时添加JVM参数观察是否存在内存泄漏

VM参数如下： -Dio.netty.leakDetection.level=advanced

Idea中配置如下：



配置完成后，启动程序，若存在内存泄漏，会打印出如下日志：

2017/12/27 16:55:27,890 ERROR [nioEventLoopGroup-68-1] (Slf4JLogger.java::error:171) - LEAK: ByteBuf.release() was not called before it's garbage-collected. See http://netty.io/wiki/reference-counted-objects.html for more information.

WARNING: 10 leak records were discarded because the leak record count is limited to 4. Use system property io.netty.leakDetection.maxRecords to increase the limit.

Recent access records: 4

#4:

io.netty.buffer.AdvancedLeakAwareByteBuf.readBytes(AdvancedLeakAwareByteBuf.java:492)

cn.qtec.qkcl.kmip.transport.util.WrapTransportMessageUtil.processBinaryArray(WrapTransportMessageUtil.java:50)

cn.qtec.qkcl.kmip.transport.handler.server.TcpServerHandler.channelRead(TcpServerHandler.java:48)

io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:362)

io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:348)

io.netty.channel.AbstractChannelHandlerContext.fireChannelRead(AbstractChannelHandlerContext.java:340)

io.netty.handler.codec.ByteToMessageDecoder.fireChannelRead(ByteToMessageDecoder.java:310)

io.netty.handler.codec.ByteToMessageDecoder.channelRead(ByteToMessageDecoder.java:284)

io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:362)

io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:348)

io.netty.channel.AbstractChannelHandlerContext.fireChannelRead(AbstractChannelHandlerContext.java:340)

io.netty.channel.DefaultChannelPipeline$HeadContext.channelRead(DefaultChannelPipeline.java:1359)

io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:362)

io.netty.channel.AbstractChannelHandlerContext.invokeChannelRead(AbstractChannelHandlerContext.java:348)

io.netty.channel.DefaultChannelPipeline.fireChannelRead(DefaultChannelPipeline.java:935)

io.netty.channel.nio.AbstractNioByteChannel$NioByteUnsafe.read(AbstractNioByteChannel.java:134)

io.netty.channel.nio.NioEventLoop.processSelectedKey(NioEventLoop.java:645)

io.netty.channel.nio.NioEventLoop.processSelectedKeysOptimized(NioEventLoop.java:580)

io.netty.channel.nio.NioEventLoop.processSelectedKeys(NioEventLoop.java:497)

io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:459)

io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:858)

io.netty.util.concurrent.DefaultThreadFactory$DefaultRunnableDecorator.run(DefaultThreadFactory.java:138)

java.lang.Thread.run(Thread.java:745)

……

出现此日志的代码位置如下：

**byte**[] body = **new byte**[bodyLength];  
 byteBuf.readBytes(body);

byteBuf读取完数据后，未显示调用release()方法，导致多线程运行时出现内存泄漏，解决方法为读取数据结束后，显示调用release()方法，添加如下红色字体代码：  
 **byte**[] body = **new byte**[bodyLength];  
 byteBuf.readBytes(body);  
 *//防止 LEAK: ByteBuf.release() was not called before it's garbage-collected.* byteBuf.release();

### 11. linux：nohup 不生成 nohup.out的方法

*nohup java -jar /xxx/xxx/xxx.jar >/dev/null 2>&1 &*

关键在于最后的 >/dev/null 2>&1 部分，/dev/null是一个虚拟的空设备（类似物理中的黑洞），任何输出信息被重定向到该设备后，将会石沉大海

>/dev/null 表示将标准输出信息重定向到"黑洞"

2>&1 表示将标准错误重定向到标准输出(由于标准输出已经定向到“黑洞”了，即：标准输出此时也是"黑洞"，再将标准错误输出定向到标准输出，相当于错误输出也被定向至“黑洞”)

# 五、netty框架的学习

## 1、介绍

Netty 是一个基于NIO的客户、服务器端编程框架，使用Netty 可以确保你快速和简单的开发出一个网络应用，例如实现了某种协议的客户，[服务端](http://baike.baidu.com/item/%E6%9C%8D%E5%8A%A1%E7%AB%AF)应用。Netty相当简化和流线化了网络应用的编程开发过程，例如，TCP和UDP的socket服务开发。

## 2、Netty包括的组件：

* 1. Bootstrap or ServerBootstrap
  2. EventLoop
  3. EventLoopGroup
  4. ChannelPipeline
  5. Channel
  6. Future or ChannelFuture
  7. ChannelInitializer
  8. ChannelHandler

Bootstrap，一个Netty应用通常由一个Bootstrap开始，它主要作用是配置整个Netty程序，串联起各个组件。

EventLoops的目的是为Channel处理IO操作，一个EventLoop可以为多个Channel服务。

EventLoopGroup会包含多个EventLoop。

ChannelPipeline，一个Netty应用基于ChannelPipeline机制，这种机制需要依赖于EventLoop和EventLoopGroup，因为它们三个都和事件或者事件处理相关。

Channel代表了一个Socket链接，或者其它和IO操作相关的组件，它和EventLoop一起用来参与IO处理。

Future，在Netty中所有的IO操作都是异步的，因此，你不能立刻得知消息是否被正确处理，但是我们可以过一会等它执行完成或者直接注册一个监听，具体的实现就是通过Future和ChannelFutures,他们可以注册一个监听，当操作执行成功或失败时监听会自动触发。总之，所有的操作都会返回一个ChannelFuture。

ChannelInitializer，当一个链接建立时，我们需要知道怎么来接收或者发送数据，当然，我们有各种各样的Handler实现来处理它，那么ChannelInitializer便是用来配置这些Handler，它会提供一个ChannelPipeline，并把Handler加入到ChannelPipeline。

Handler，为了支持各种协议和处理数据的方式，便诞生了Handler组件。Handler主要用来处理各种事件，这里的事件很广泛，比如可以是连接、数据接收、异常、数据转换等。

ChannelInboundHandler，一个最常用的Handler。这个Handler的作用就是处理接收到数据时的事件，也就是说，我们的业务逻辑一般就是写在这个Handler里面的，ChannelInboundHandler就是用来处理我们的核心业务逻辑。

## 3、ServerBootstrap的启动过程

1）创建两个EventLoopGroup事件循环对象，一个用于处理accept请求事件，另一个用于处理accept到的处理事件；

EventLoopGroup bossGroup = new NioEventLoopGroup();

EventLoopGroup workGroup = new NioEventLoopGroup

2）创建netty服务器的辅助启动类对象

ServerBootstrap bootstrap = new ServerBootstrap();

3）设置事件循环对象，前者用于处理accept请求事件，后者处理accept到的事件

bootstrap.group(bossGroup, workGroup);

4）使用指定类建立新的accept事件的channel，用于构造serversocketchannel的工厂类

bootstrap.channel(NioServerSocketChannel.class);

5）标识当服务器请求处理线程全满时，设置临时存放请求的队列的最大长度

bootstrap.option(ChannelOption.*SO\_BACKLOG*, 100);

6）设置ChannelHandler，对处理accept到的事件的Channel进行初始化

bootstrap.childHandler(new ChildChannelHandler());

7）创建ChannelFuture对象，绑定端口并开始监听

ChannelFuture future = bootstrap.bind(port).sync();

8）等待服务端监听端口关闭

future.channel().close().sync();

## 4、Pipeline的生命周期

**Server端：**

1）ServerSocketChannel：

ChannelRegistered（注册到EventLoop）-->bind（绑定端口）-->ChannelActive（激活）-->read（注册OP\_ACCEPT到SelectKey）-->ChannelRead（接收客户端连接，此时会将客户端连接注册到workerGroup）-->ChannelReadComplete（读取客户连接完成）-->接收下一个连接-->直到最终关闭触发ChannelUnregistered（从EventLoop中取消注册）

2）SocketChannel：

ServerSocketChannel接收到客户端请求后，将其注册到workGroup中-->ChannelRegistered（注册到EventLoop中）--> ChannelActive（激活）-->read（注册OP\_ACCEPT到SelectKey）-->ChannelRead（读取到数据）-->业务数据处理—>write（写数据到buffer）-->flush（数据最终发送）-->writeAndFlush（前两个操作的组合）-->ChannelReadComplete（读取过程结束）-->ChannelInactive（连接关闭）--> channelUnregistered（从EventLoop中取消注册）

**Client端**

SocketChannel：

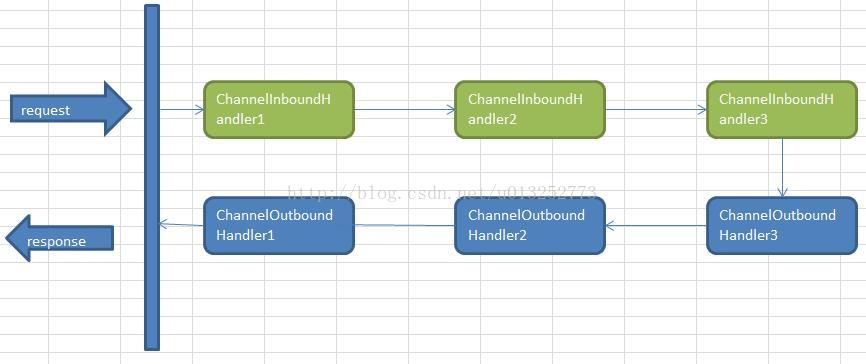
ChannelRegistered（注册到EventLoop中）--> connect（连接server）-->ChannelActive（激活）-->read（注册OP\_READ）--> write（写数据到buffer）-->flush（数据最终发送）-->writeAndFlush（前两个操作的组合）-->ChannelRead（读取到server传回的数据）-->ChannelReadComplete（读取过程结束）-->ChannelInactive（连接关闭）--> channelUnregistered（从EventLoop中取消注册）。

## 5、Handler的执行顺序

Handler在netty中，无疑占据着非常重要的地位。Handler与Servlet中的filter很像，通过Handler可以完成通讯报文的解码编码、拦截指定的报文、统一对日志错误进行处理、统一对请求进行计数、控制Handler执行与否。一句话，没有它做不到的只有你想不到的。

Netty中的所有handler都实现自ChannelHandler接口。按照输出输出来分，分为ChannelInboundHandler、ChannelOutboundHandler两大类。ChannelInboundHandler对从客户端发往服务器的报文进行处理，一般用来执行解码、读取客户端数据、进行业务处理等；ChannelOutboundHandler对从服务器发往客户端的报文进行处理，一般用来进行编码、发送报文到客户端。

Netty中，可以注册多个handler。ChannelInboundHandler按照注册的先后顺序执行；ChannelOutboundHandler按照注册的先后顺序逆序执行，如下图所示，按照注册的先后顺序对Handler进行排序，request进入Netty后的执行顺序为：



## 6、Netty单元测试

正如前面所学的，Netty提供了一个简单的方法在ChannelPipeline上“堆叠”不同的ChannelHandler实现。所有的ChannelHandler都会参与处理事件，这个设计允许独立出可重用的小逻辑块，它只处理一个任务。这不仅使代码更清晰，也更容易测试。

    测试ChannelHandler可以通过使用“嵌入式”传输很容易的传递事件槽管道以测试你的实现。对于这个嵌入式传输，Netty提供了一个特定的Channel实现：EmbeddedChannel。但是它是如何工作的呢？EmbeddedChannel的工作非常简单，它允许写入入站或出站数据，然后检查ChannelPipeline的结束。这允许你检查消息编码/解码或触发ChannelHandler任何行为。

编写入站和出站的却别是什么？入站数据是通过ChannelInboundHandler处理，代表从远程对等通道读取数据；出站数据是通过ChannelOutboundHandler处理，代表写入数据到远程对等通道。因此测试ChannelHandler就会选择writeInbound(...)或writeOutbound()(或者都选择)。

EmbeddedChannel提供了下面一些方法：

writeInbound(Object...)，写一个消息到入站通道

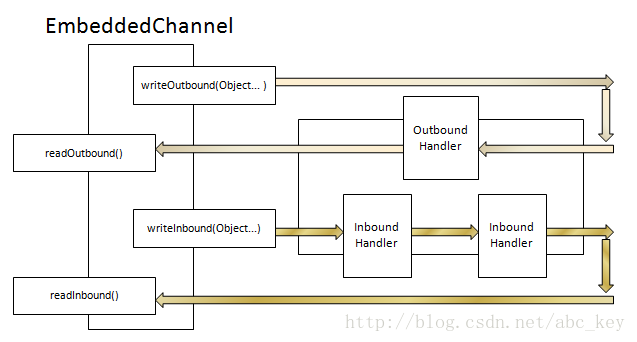
writeOutbound(Object...)，写消息到出站通道

readInbound()，从EmbeddedChannel读取入站消息，可能返回null

readOutbound()，从EmbeddedChannel读取出站消息，可能返回null

finish()，标示EmbeddedChannel已结束，任何写数据都会失败

为了更清楚的了解其处理过程，看下图：



如上图所示，使用writeOutbound(...)写消息到通道，消息在出站方法通过ChannelPipeline，之后就可以使用readOutbound()读取消息。着同样使用与入站，使用writeInbound(...)和readInbound()。处理入站和出站是相似的，它总是遍历整个ChannelPipeline直到ChannelPipeline结束，并将处理过的消息存储在EmbeddedChannel中。下面来看看如何测试你的逻辑。

### 6.1 测试ChannelHandler

测试ChannelHandler最好的选择是使用EmbeddedChannel。

Spring配置文件

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/context   
 http://www.springframework.org/schema/context/spring-context.xsd">  
  
 <context:component-scan base-package="cn.zhp.netty"/>  
  
 <bean name="kmipEncoder" class="cn.qtec.key.kmip.process.encoder.KMIPEncoder"/>  
 <bean name="kmipDecoder" class="cn.qtec.key.kmip.process.decoder.KMIPDecoder"/>  
 <context:property-placeholder location="classpath:config.properties"/>  
   
 <!-- for Netty Handler junit test -->  
 <bean name="" class="io.netty.channel.embedded.EmbeddedChannel">  
 <constructor-arg name="handlers" ref="clientLoginAuthReqHandler"/>  
 </bean>  
</beans>

#### 6.1.1 客户端ClientLoginAuthReqHandler

package cn.zhp.netty.custom.handler;  
import cn.zhp.netty.custom.transport.entity.NettyMessageHeader;  
import cn.zhp.netty.custom.transport.entity.NettyMessage;  
import io.netty.channel.\*;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.stereotype.Component;

*/\*\*  
 \* Description:  
 \* 进行登录认证  
 \** ***@author*** *Created by zhangp on 2017/9/13.  
 \** ***@version*** *v1.0.0  
 \*/*  
@Component  
@Sharable  
public class ClientLoginAuthReqHandler extends ChannelInboundHandlerAdapter {  
 private static final Logger *logger* = LoggerFactory.*getLogger*(ClientLoginAuthReqHandler.class);  
 @Override  
 public void channelActive(ChannelHandlerContext ctx) throws Exception {  
 *logger*.debug("------------ 进行登录认证");  
 //进行单元测试时不能使用writeAndFlush, 无法触发ChannelRead()，导致返回null  
 //ctx.writeAndFlush(buildLoginReq());  
 //fireChannelRead()触发下一个inHandler的channelRead()方法  
 ctx.fireChannelRead(buildLoginReq());

}  
 @Override  
 public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  
 *logger*.debug("------------ 接收服务端认证响应结果");  
 NettyMessage message = (NettyMessage) msg;  
 if (message.getHeader() != null && message.getHeader().getMessageType() == (byte) 2) {  
 byte loginResult = (byte) message.getBody();  
 if (loginResult != 0) {  
 *logger*.error("=========================>>>>>>>>> 认证失败！");  
 ctx.close();  
 }  
 *logger*.debug("Login is OK, message: {}", message);  
 *logger*.debug("=========================>>>>>>>>> 登录认证成功！");  
 }

}  
 @Override  
 public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {  
 *logger*.error("------------ LoginAuthReqHandler error: {}", cause.getMessage());  
 ctx.close();  
 }  
 private NettyMessage buildLoginReq() {  
 NettyMessage message = new NettyMessage();  
 NettyMessageHeader header = new NettyMessageHeader();  
 header.setMessageType((byte) 1);  
 message.setHeader(header);  
 message.setBody("loginReq");  
 return message;  
 }  
}

注意：必须是ctx.fireChannelRead(buildLoginReq());

##### 6.1.1.1 ClientLoginAuthReqHandler单元测试

package test.handler;  
  
import cn.zhp.netty.custom.transport.entity.NettyMessage;  
import cn.zhp.netty.custom.transport.entity.NettyMessageHeader;  
import cn.zhp.netty.custom.transport.enums.MessageTypeEnum;  
import cn.zhp.netty.custom.transport.enums.ResultStatusEnum;  
import io.netty.channel.embedded.EmbeddedChannel;  
import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.test.context.ContextConfiguration;  
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;  
import test.BaseJunit4Test;  
import java.nio.ByteBuffer;  
import static org.junit.Assert.\*;  
@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration(locations = "classpath:/spring-context.xml")  
public class ClientLoginAuthReqHandlerTest extends BaseJunit4Test {  
 @Autowired  
 private EmbeddedChannel channel;  
  
 @Test  
 public void testChannelRead\_\_\_InBound() {  
 byte type = MessageTypeEnum.*LOGIN\_RESP*;  
 byte status = ResultStatusEnum.*SUCCESS*;  
 NettyMessage message = new NettyMessage();  
 NettyMessageHeader header = new NettyMessageHeader();  
 header.setMessageType(type);  
 message.setHeader(header);  
 //body: 0  
 message.setBody(status);  
 */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \*\*\*\* write bytes to inBound \*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 assertTrue*(channel.writeInbound(message));  
 *assertTrue*(channel.finish());  
 */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \*\*\*\* read message inBound \*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/* NettyMessage result = channel.readInbound();  
 *assertNotNull*(result);  
 *assertEquals*("loginReq", result.getBody());  
 NettyMessageHeader resultHeader = result.getHeader();  
 *assertEquals*(MessageTypeEnum.*LOGIN\_REQ*, resultHeader.getMessageType());  
 }  
}

#### 6.1.2 客户端代码

package cn.zhp.netty.base.client;  
import cn.zhp.netty.custom.handler.codec.NettyMessageDecoder;  
import cn.zhp.netty.custom.handler.codec.NettyMessageEncoder;  
import io.netty.bootstrap.Bootstrap;  
import io.netty.channel.\*;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import java.util.ArrayList;  
import java.util.concurrent.Executors;  
import java.util.concurrent.ScheduledExecutorService;  
public class NettyClient {  
 private final static Logger *logger* = LoggerFactory.*getLogger*(NettyClient.class);  
 private String host;  
 private int port;  
 private ArrayList<ChannelHandler> childChannelHandlers = new ArrayList<>();  
 //客户端重连使用  
 private ScheduledExecutorService executor = Executors.*newScheduledThreadPool*(1);  
 public NettyClient() {  
 this.host = "127.0.0.1";  
 this.port = 6666;  
 }  
 public NettyClient(String host, int port) {  
 this.host = host;  
 this.port = port;  
 }  
 public void start() throws Exception {  
 EventLoopGroup workerGroup = new NioEventLoopGroup();  
 //配置客户端NIO线程组  
 try {  
 Bootstrap bootstrap = new Bootstrap();  
 bootstrap.group(workerGroup)  
 .channel(NioSocketChannel.class)  
 .option(ChannelOption.*TCP\_NODELAY*, false)  
 .handler(new ChildChannelHandlers())  
 //异步连接操作  
 ChannelFuture future = bootstrap.connect(host, port).sync();  
 *logger*.debug("Netty Client start on {}", port);  
 *logger*.debug("Waiting for connect...");  
 // Wait until the connection is closed.  
 future.channel().closeFuture().sync();  
 } finally {  
 workerGroup.shutdownGracefully();  
 */\*\*\* 以下内容为客户端进行重连代码\*\*/* //释放所有资源后，清空资源，再次发起重连操作  
 /\* executor.execute(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 //睡眠 5s  
 TimeUnit.SECONDS.sleep(5);  
 //发起重连操作  
 start();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 });\*/  
 }  
 }  
 public void addChildChannelHandler(ChannelHandler childChannelHandler) {  
 childChannelHandlers.add(childChannelHandler);  
 }  
 private class ChildChannelHandlers extends ChannelInitializer<SocketChannel> {  
 @Override  
 protected void initChannel(SocketChannel socketChannel) throws Exception {  
 for (ChannelHandler childChannelHandler : childChannelHandlers) {  
 socketChannel.pipeline().addLast(childChannelHandler);  
 }  
 }  
 }  
}

@Component  
public class CustomClient extends NettyClient{  
 public static void main(String[] args) throws Exception {  
 ApplicationContext context = new ClassPathXmlApplicationContext("classpath:spring-context.xml");  
 CustomClient customClient = (CustomClient) context.getBean("customClient");  
 ClientLoginAuthReqHandler clientLoginAuthReqHandler = (ClientLoginAuthReqHandler) context.getBean("clientLoginAuthReqHandler");  
 customClient.addChildChannelHandler(clientLoginAuthReqHandler);  
 customClient.start();  
 }  
}

#### 6.1.3 服务端handler

package cn.zhp.netty.custom.handler;  
import cn.zhp.netty.custom.transport.entity.NettyMessageHeader;  
import cn.zhp.netty.custom.transport.entity.NettyMessage;  
import cn.zhp.netty.custom.transport.enums.MessageTypeEnum;  
import cn.zhp.netty.custom.transport.enums.ResultStatusEnum;  
import io.netty.channel.\*;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.stereotype.Component;  
import java.net.InetSocketAddress;  
import java.util.\*;  
*/\*\*  
 \* 登录认证处理器  
 \** ***@author*** *Created by zhangp .  
 \** ***@version*** *v1.0.0  
 \*/*@Component  
@Sharable  
public class ServerLoginAuthRespHandler extends ChannelInboundHandlerAdapter {  
 private static final Logger *logger* = LoggerFactory.*getLogger*(ServerLoginAuthRespHandler.class);  
  
 private Map<String, Boolean> nodeCheck = new ConcurrentHashMap<String,Boolean>();  
 //IP白名单  
 private String[] whitekList = {"127.0.0.1"};  
   
 @Override  
 public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  
 *logger*.debug("=================== 进行登录认证! ");  
   
 NettyMessage message = (NettyMessage) msg;  
 //如果是握手请求消息，处理， 其他消息透传  
 if (message.getHeader() != null && message.getHeader().getMessageType() == MessageTypeEnum.*LOGIN\_REQ*) {  
 String currentNodeAddress = ctx.channel().remoteAddress().toString();  
 NettyMessage loginResp = null;  
  
 //重复登录，拒绝  
 if (nodeCheck.containsKey(currentNodeAddress)) {  
 *logger*.warn("远程客户端 {} 重复登录，已拒绝！", currentNodeAddress);  
 loginResp = buildLoginResponse(ResultStatusEnum.*FAIL*);  
 ctx.writeAndFlush(loginResp);  
 } else {  
 //IP认证白名单校验  
 InetSocketAddress address = (InetSocketAddress) ctx.channel().remoteAddress();  
 String ip = address.getAddress().getHostAddress();  
 boolean isOK = false;  
 for (String WIP : whitekList) {  
 if (WIP.equals(ip)) {  
 isOK = true;  
 break;  
 }  
 }  
  
 if (isOK) {  
 loginResp = buildLoginResponse(ResultStatusEnum.*SUCCESS*);  
 nodeCheck.put(currentNodeAddress, true);  
 } else {  
 loginResp = buildLoginResponse(ResultStatusEnum.*FAIL*);  
 }  
  
 *logger*.debug(" The login response is : {} ", loginResp);  
 //fireChannelRead()触发下一个inHandler的channelRead()方法  
 ctx.fireChannelRead(loginResp);  
 //进行单元测试时不能使用, 无法触发ChannelRead()，导致返回null  
 //ctx.writeAndFlush(loginResp);  
 }  
 } else {  
 //ctx.writeAndFlush(msg);  
 ctx.fireChannelRead(msg);  
 }  
 }  
  
 @Override  
 public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {  
 *logger*.debug("------------ server channelReadComplete...");  
 ctx.flush();  
 }  
  
 @Override  
 public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {  
 *logger*.error("------------ LoginAuthRespHandler error: {}", cause.getMessage());  
 cause.printStackTrace();  
 ctx.close();  
 }  
  
 private NettyMessage buildLoginResponse(byte result) {  
 NettyMessage message = new NettyMessage();  
 NettyMessageHeader header = new NettyMessageHeader();  
 //2 bytes  
 header.setMessageType((byte)2);  
 message.setHeader(header);  
 message.setBody(result);  
 return message;  
 }  
}

##### 6.1.3.1 ServerLoginAuthRespHandler单元测试

package test.handler;  
import cn.zhp.netty.custom.transport.\*;  
import io.netty.channel.embedded.EmbeddedChannel;  
import org.junit.Test;  
import org.springframework.beans.factory.annotation.Autowired;  
import test.BaseJunit4Test;  
import static org.junit.Assert.\*;  
public class ServerLoginAuthRespHandlerTest extends BaseJunit4Test {  
 @Autowired  
 private EmbeddedChannel channel;  
 @Test  
 public void channelRead() throws Exception {  
 byte type = MessageTypeEnum.*LOGIN\_REQ*;  
 NettyMessage message = new NettyMessage();  
 NettyMessageHeader header = new NettyMessageHeader();  
 header.setMessageType(type);  
 message.setHeader(header); */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \*\*\*\* write bytes to inBound \*\*\*  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 assertTrue*(channel.writeInbound(message));  
 *assertTrue*(channel.finish()); */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \*\*\*\* read message inBound \*\*\*  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/* NettyMessage result = channel.readInbound();  
 *assertNotNull*(result);  
 *assertEquals*(ResultStatusEnum.*SUCCESS*, result.getBody());  
 NettyMessageHeader resultHeader = result.getHeader();  
 *assertEquals*(MessageTypeEnum.*LOGIN\_RESP*, resultHeader.getMessageType());  
 }  
}

#### 6.1.4 服务端代码

package cn.zhp.netty.base.server;  
  
import cn.zhp.netty.custom.handler.codec.NettyMessageDecoder;  
import cn.zhp.netty.custom.handler.codec.NettyMessageEncoder;  
import io.netty.bootstrap.ServerBootstrap;  
import io.netty.channel.ChannelFuture;  
import io.netty.channel.ChannelHandler;  
import io.netty.channel.ChannelInitializer;  
import io.netty.channel.ChannelOption;  
import io.netty.channel.EventLoopGroup;  
import io.netty.channel.nio.NioEventLoopGroup;  
import io.netty.channel.socket.SocketChannel;  
import io.netty.channel.socket.nio.NioServerSocketChannel;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.stereotype.Component;  
import java.util.ArrayList;  
@Component  
public class NettyServer {  
 private final static Logger *logger* = LoggerFactory.*getLogger*(NettyServer.class);  
  
 private int port;  
 private ArrayList<ChannelHandler> childChannels;  
  
 public NettyServer() {  
 port = 6666;  
 childChannels = new ArrayList<>();  
 }  
  
 public NettyServer(int port) {  
 this.port = port;  
 }  
  
 public void start() throws Exception{  
 EventLoopGroup bossGroup = new NioEventLoopGroup();  
 EventLoopGroup workerGroup = new NioEventLoopGroup();  
  
 try {  
 ServerBootstrap b = new ServerBootstrap();  
 b.group(bossGroup, workerGroup)  
 .channel(NioServerSocketChannel.class)  
 .option(ChannelOption.*SO\_BACKLOG*, 1024)  
 .childOption(ChannelOption.*SO\_KEEPALIVE*, true)  
 .childHandler(new ChildChannelHandler());  
  
 //绑定端口，同步等待成功  
 ChannelFuture f = b.bind(port).sync();  
  
 *logger*.debug("Netty服务器启动, 端口: {}", port);  
  
 //等待服务端监听端口关闭  
 f.channel().closeFuture().sync();  
 }finally {  
 //优雅退出  
 bossGroup.shutdownGracefully();  
 workerGroup.shutdownGracefully();  
 }  
 }  
  
 public ArrayList<ChannelHandler> getChildChannels() {  
 return childChannels;  
 }  
  
 public void setChildChannels(ArrayList<ChannelHandler> childChannels) {  
 this.childChannels = childChannels;  
 }  
  
 public void addChildChannel(ChannelHandler childChannel) {  
 childChannels.add(childChannel);  
 }  
  
 private class ChildChannelHandler extends ChannelInitializer<SocketChannel> {  
 @Override  
 protected void initChannel(SocketChannel socketChannel) throws Exception {  
 for (ChannelHandler childChannel : childChannels) {  
 socketChannel.pipeline().addLast(childChannel);  
 }  
 }  
 }  
}

package cn.zhp.netty.custom.server;  
  
import cn.zhp.netty.base.server.NettyServer;  
import cn.zhp.netty.custom.handler.CustomServerHandler;  
import cn.zhp.netty.custom.handler.ServerHeartBeatRespHandler;  
import cn.zhp.netty.custom.handler.ServerLoginAuthRespHandler;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
import org.springframework.stereotype.Component;  
@Component  
public class CustomServer extends NettyServer{  
 public static void main(String[] args) throws Exception {  
 ApplicationContext context = new ClassPathXmlApplicationContext("classpath:spring-context.xml");  
 CustomServer customServer = (CustomServer) context.getBean("customServer");  
 ServerLoginAuthRespHandler serverLoginAuthRespHandler = (ServerLoginAuthRespHandler) context.getBean("serverLoginAuthRespHandler");  
 customServer.addChildChannel(serverLoginAuthRespHandler);  
 customServer.start();  
 }  
}

#### 6.1.5 消息结构

public final class NettyMessage {  
 //消息头  
 private NettyMessageHeader header;  
 //消息体  
 private Object body;  
 public final NettyMessageHeader getHeader() {  
 return header;  
 }  
 public final void setHeader(NettyMessageHeader header) {  
 this.header = header;  
 }  
 public final Object getBody() {  
 return body;  
 }  
 public final void setBody(Object body) {  
 this.body = body;  
 }  
 public boolean hasBody() {  
 return body != null;  
 }  
 @Override  
 public String toString() {  
 StringBuilder sb = new StringBuilder();  
// sb.append("NettyMessage:");  
 sb.append("\n\t消息头:\t").append(header.toString());  
 if (hasBody()) {  
 sb.append("\n\t消息体:\t").append(body.toString());  
 }  
 return sb.toString();  
 }  
}

package cn.zhp.netty.custom.transport.entity;  
  
import java.util.HashMap;  
import java.util.Map;  
*/\*\*  
 \* 消息头  
 \** ***@author*** *Created by zhangp  
 \** ***@version*** *v1.0.0  
 \*/*public final class NettyMessageHeader {  
 //消息校验码  
 private int crcCode = 0xABEF01;  
 //版本  
 private byte version;  
 //会话ID  
 private long sessionID;  
 //消息类型  
 private byte messageType;  
 //消息优先级  
 private byte priority;  
 //消息长度  
 private short messageLength;  
 //附件  
 private Map<String, Object> attachment = new HashMap<>();  
 public final int getCrcCode() {  
 return crcCode;  
 }  
 public final void setCrcCode(int crcCode) {  
 this.crcCode = crcCode;  
 }  
 public final byte getVersion() {  
 return version;  
 }  
 public final void setVersion(byte version) {  
 this.version = version;  
 }  
 public final long getSessionID() {  
 return sessionID;  
 }  
 public final void setSessionID(long sessionID) {  
 this.sessionID = sessionID;  
 }  
 public final byte getMessageType() {  
 return messageType;  
 }  
 public final void setMessageType(byte messageType) {  
 this.messageType = messageType;  
 }  
 public final byte getPriority() {  
 return priority;  
 }  
 public final void setPriority(byte priority) {  
 this.priority = priority;  
 }  
 public final short getMessageLength() {  
 return messageLength;  
 }  
 public final void setMessageLength(short messageLength) {  
 this.messageLength = messageLength;  
 }  
 @Override  
 public String toString() {  
 return "MessageHeader{" +  
 "crcCode=" + crcCode +  
 ", version=" + version +  
 ", sessionID=" + sessionID +  
 ", messageType=" + messageType +  
 ", priority=" + priority +  
 ", messageLength=" + messageLength +  
// ", attachment=" + attachment +  
 '}';  
 }  
 public Map<String, Object> getAttachment() {  
 return attachment;  
 }  
 public void setAttachment(Map<String, Object> attachment) {  
 this.attachment = attachment;  
 }  
}

# 六、thrift框架的学习

### 1、介绍

Apache Thrift 是 Facebook 实现的一种高效的、支持多种编程语言的远程服务调用的框架。它采用**接口描述语言**定义并创建服务，支持可扩展的跨语言服务开发，所包含的代码生成引擎可以在多种语言中，如 C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk 等创建高效的、无缝的服务，其传输数据采用二进制格式，相对 XML 和 JSON 体积更小，对于高并发、大数据量和多语言的环境更有优势。

在多种不同的语言之间通信thrift可以作为二进制的高性能的通讯中间件，支持数据(对象)序列化和多种类型的RPC服务。

Thrift适用于程序对程 序静态的数据交换，需要先确定好他的数据结构，他是完全静态化的，当数据结构发生变化时，必须重新编辑**IDL文件**，代码生成，再编译载入的流程，跟其他IDL工具相比较可以视为是Thrift的弱项，Thrift适用于搭建大型数据交换及存储的通用工具，对于大型系统中的内部数据传输相对于JSON和xml无论在性能、传输大小上有明显的优势。

什么是RPC框架？

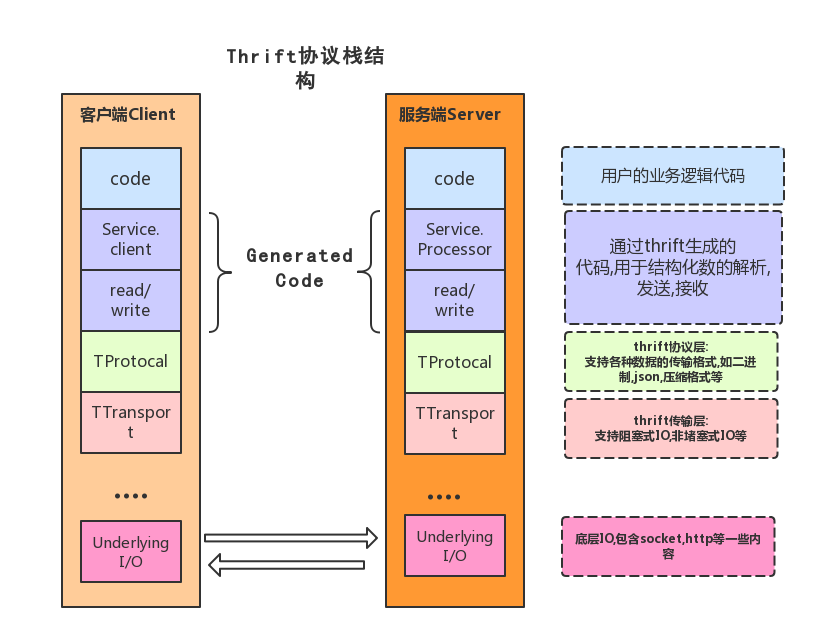
RPC全称为Remote Procedure Call,意为远程过程调用。

假设有两台服务器A,B.A服务器上部署着一个应用a,B服务器上部署着一个应用b,现在a希望能够调用b应用的某个函数(方法),但是二者不在同一个进程内,不能直接调用,就需要通过网络传输,在AB服务器之间建一条网络传输通道,a把参数传过去,b接收到参数调用自己的方法,得到结果,再通过网络传回给a,简单讲就是A通过网络来调用B的过程.这个过程要涉及的东西很多,比如多线程,Socket,序列化反序列化,网络I/O,很复杂,于是牛掰的程序员把这些封装起来做成一套框架,供大家使用,就是RPC框架。

thrift使用的IDL语言

thrift通过一个中间语言IDL(接口定义语言)来定义RPC的数据类型和接口,这些内容写在以.thrift结尾的文件中,然后通过特殊的编译器来生成不同语言的代码,以满足不同需要的开发者,比如java开发者,就可以生成java代码,c++开发者可以生成c++代码,生成的代码中不但包含目标语言的接口定义,方法,数据类型,还包含有RPC协议层和传输层的实现代码。

### 2、thrift使用的协议栈结构



thrift是一种c/s的架构体系.在最上层是用户自行实现的业务逻辑代码.第二层是由thrift编译器自动生成的代码，主要用于结构化数据的解析，发送和接收。TServer主要任务是高效的接受客户端请求，并将请求转发给Processor处理。Processor负责对客户端的请求做出响应，包括RPC请求转发，调用参数解析和用户逻辑调用，返回值写回等处理。从TProtocol以下部分是thirft的传输协议和底层I/O通信。TProtocol是用于数据类型解析的，将结构化数据转化为字节流给TTransport进行传输。TTransport是与底层数据传输密切相关的传输层，负责以字节流方式接收和发送消息体，不关注是什么数据类型。底层IO负责实际的数据传输，包括socket、文件和压缩数据流等。

### 3、Thrift使用的基本流程

准备：到官网下载需要的jar包和thrift编译工具：最新版本为thrift-0.10.0

<http://thrift.apache.org/download>

* [thrift-0.10.0.tar.gz](http://www.apache.org/dyn/closer.cgi?path=/thrift/0.10.0/thrift-0.10.0.tar.gz) [[PGP](https://www.apache.org/dist/thrift/0.10.0/thrift-0.10.0.tar.gz.asc)] [[MD5](https://www.apache.org/dist/thrift/0.10.0/thrift-0.10.0.tar.gz.md5)]
* [Thrift compiler for Windows (thrift-0.10.0.exe)](http://www.apache.org/dyn/closer.cgi?path=/thrift/0.10.0/thrift-0.10.0.exe) [[PGP](https://www.apache.org/dist/thrift/0.10.0/thrift-0.10.0.exe.asc)] [[MD5](https://www.apache.org/dist/thrift/0.10.0/thrift-0.10.0.exe.md5)]

1. 创建thrift的IDL文件：

**namespace java** com.demo.thrift  
**service** Hello{  
 **string** helloString(1:**string** para)  
 **i32** helloInt(1:**i32** para)  
 **bool** booleanHello(1:**bool** para)  
 **void** helloVoid()  
 **string** helloNull()  
}

2）编译IDL文件，生成对应的语言

生成php语言：

thrift-0.10.0.exe -r --gen php hello.thrift

生成java语言（这里使用java）：

thrift-0.10.0.exe -r --gen java hello.thrift

生成C++语言

thrift-0.10.0.exe -r --gen cpp hello.thrift

注：在cmd中需要切换到hello.thrift所在目录下执行。

3）在maven项目的pom.xml文件中添加如下内容

<dependency>  
 <groupId>org.apache.thrift</groupId>  
 <artifactId>libthrift</artifactId>  
 <version>0.10.0</version>  
</dependency>

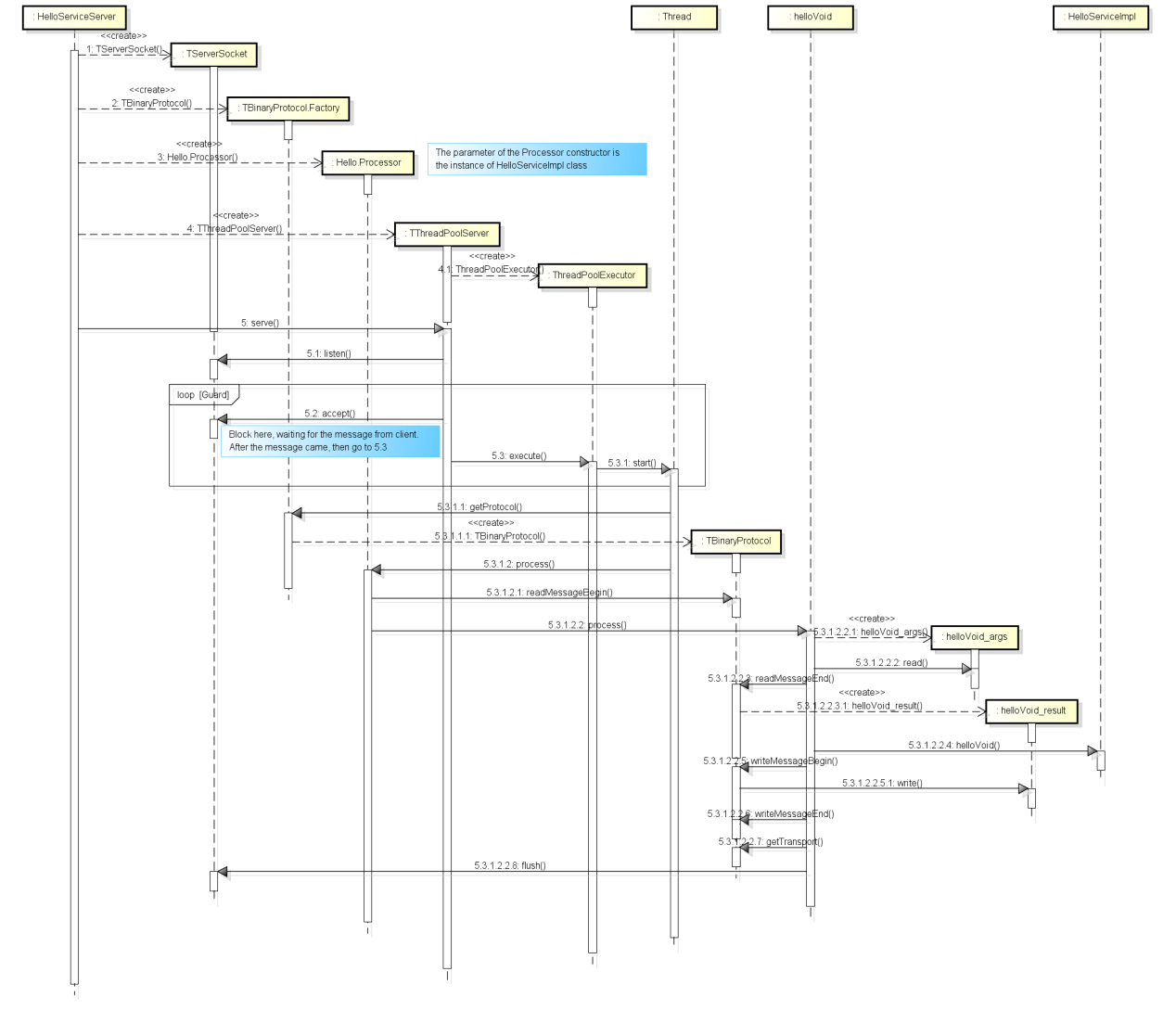
4）创建HelloServic.Iface的实现类

5）创建HelloServiceServer类

6）创建HelloServiceClient类

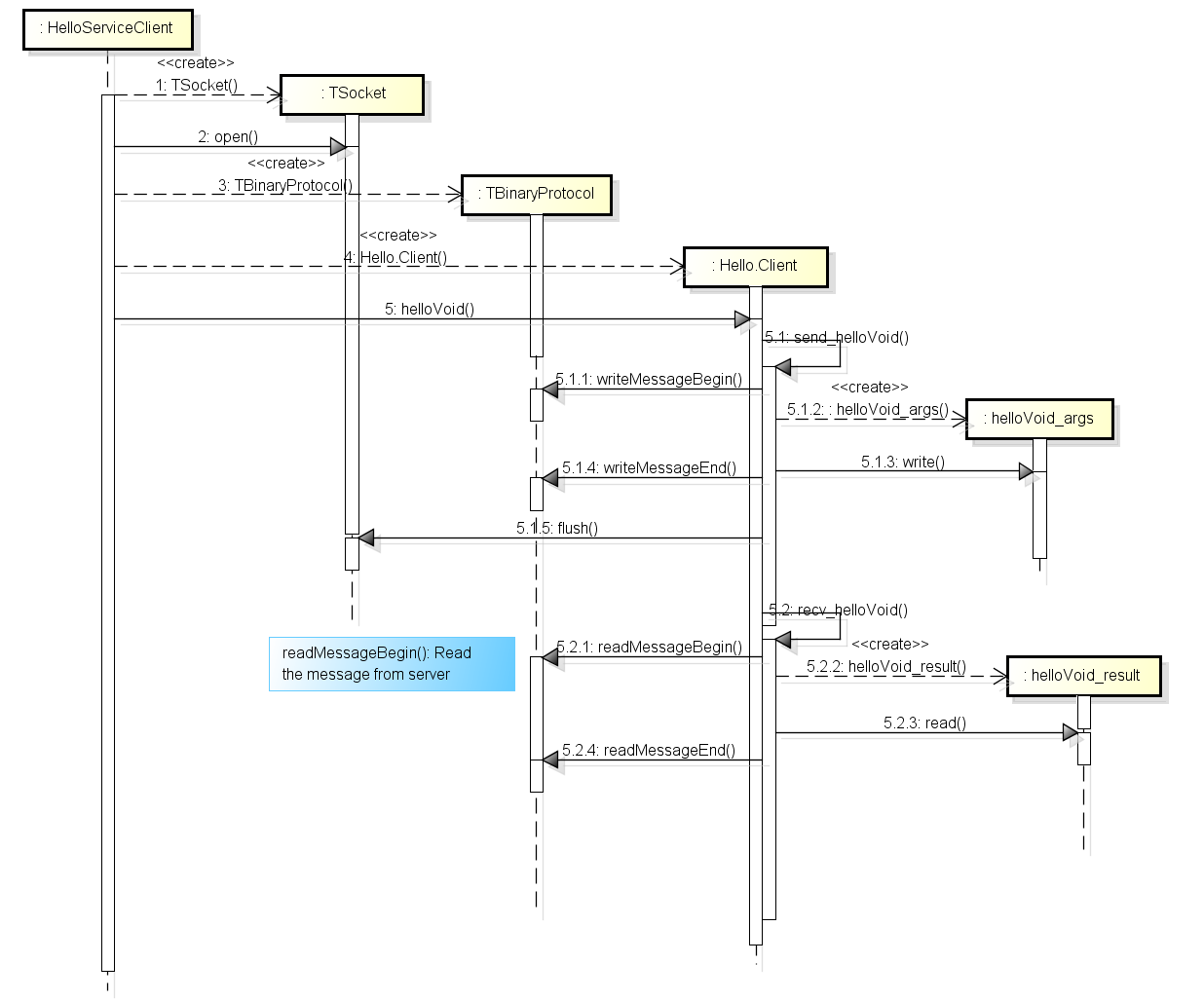
7）先运行HelloServiceServer，再运行HelloServiceClient

**Server端启动、服务时序图**



该图所示是 HelloServiceServer 启动的过程以及服务被客户端调用时，服务器的响应过程。从图中我们可以看到，程序调用了 TThreadPoolServer 的 serve 方法后，server 进入阻塞监听状态，其阻塞在 TServerSocket 的 accept 方法上。当接收到来自客户端的消息后，服务器发起一个新线程处理这个消息请求，原线程再次进入阻塞状态。在新线程中，服务器通过 TBinaryProtocol 协议读取消息内容，调用 HelloServiceImpl 的 helloVoid 方法，并将结果写入 helloVoid\_result 中传回客户端。

**Client调用服务时序图**



该图所示是 HelloServiceClient 调用服务的过程以及接收到服务器端的返回值后处理结果的过程。从图中我们可以看到，程序调用了 Hello.Client 的 helloVoid 方法，在 helloVoid 方法中，通过 send\_helloVoid 方法发送对服务的调用请求，通过 recv\_helloVoid 方法接收服务处理请求后返回的结果。

### 4、数据类型

Thrift 脚本可定义的数据类型包括以下几种类型：

**基本类型：**

bool： 布尔值，true 或 false，对应 Java 的 boolean

byte： 8 位有符号整数，对应 Java 的 byte

i16： 16 位有符号整数，对应 Java 的 short

i32： 32 位有符号整数，对应 Java 的 int

i64： 64 位有符号整数，对应 Java 的 long

double： 64 位浮点数，对应 Java 的 double

string： 未知编码文本或二进制字符串，对应 Java 的 String

**结构体类型：**

struct：定义公共的对象，类似于 C 语言中的结构体定义，在 Java 中是一个 JavaBean

**容器类型：**

list：对应 Java 的 ArrayList

set：对应 Java 的 HashSet

map：对应 Java 的 HashMap

**异常类型：**

exception：对应 Java 的 Exception

**服务类型：**

service：对应服务的类

### 5、协议

Thrift 可以让用户选择客户端与服务端之间传输通信协议的类别，在传输协议上总体划分为文本 (text) 和二进制 (binary) 传输协议，为节约带宽，提高传输效率，一般情况下使用二进制类型的传输协议为多数，有时还会使用基于文本类型的协议，这需要根据项目 / 产品中的实际需求。

常用协议有以下几种：

TBinaryProtocol 二进制编码格式进行数据传输

TCompactProtocol 高效率的、密集的二进制编码格式进行数据传输

TJSONProtocol 使用 JSON 的数据编码协议进行数据传输

TSimpleJSONProtocol 只提供 JSON 只写的协议，适用于通过脚本语言解析

### 6、传输层

常用的传输层有以下几种：

TSocket —— 使用阻塞式 I/O 进行传输，是最常见的模式

TFramedTransport —— 使用非阻塞方式，按块的大小进行传输，类似于 Java 中的 NIO

注：若使用 TFramedTransport 传输层，其服务器必须修改为非阻塞的服务类型。

TNonblockingTransport —— 使用非阻塞方式，用于构建异步客户端

### 7、服务端类型

常见的服务端类型有以下几种：

TSimpleServer 单线程服务器端使用标准的阻塞式 I/O

TThreadPoolServer 多线程服务器端使用标准的阻塞式 I/O

TNonblockingServer 多线程服务器端使用非阻塞式 I/O

### 8、异步客户端构建练习

# 七、日志组件SLF4J

### 1、官网介绍 <https://www.slf4j.org/>

The Simple Logging Facade for Java (SLF4J) serves as a simple facade or abstraction for various logging frameworks (e.g. java.util.logging, logback, log4j) allowing the end user to plug in the desired logging framework at deployment time.

官网上最新的稳定版本

* [**slf4j-1.7.25.tar.gz**](https://www.slf4j.org/dist/slf4j-1.7.25.tar.gz) Linux下使用
* [**slf4j-1.7.25.zip**](https://www.slf4j.org/dist/slf4j-1.7.25.zip) Windows下使用

注：版本中带alpha的为模块化实验版本；

解压后slf4j-api-xxx.jar是核心包，而slf4j-api-xxx-source.jar是源码包

### 2、使用配置

#### 2.1 在maven项目中添加依赖关系

<dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-log4j12</artifactId>  
 <version>1.7.25</version>  
 <scope>test</scope>  
</dependency>

<dependency>  
 <groupId>log4j</groupId>  
 <artifactId>log4j</artifactId>  
 <version>1.2.17</version>  
</dependency>

#### 2.2 日志系统配置

可以使用以下两种方式对日志系统的输出格式、记录级别、输出方式等进行配置，且两者只能选其一。

##### 2.2.1 properties文件方式

在resources目录下创建log4j.properties文件，内容如下所示：

#全局设置  
#为true时将显示logger的debug信息，默认为false  
log4j.debug = false  
log4j.rootLogger = info, stdout, errorlog

#输出信息到控制台的设置  
log4j.appender.stdout = org.apache.log4j.ConsoleAppender  
#自定义输出格式  
log4j.appender.stdout.layout = org.apache.log4j.PatternLayout  
log4j.appender.stdout.Target = System.out  
log4j.appender.stdout.layout.ConversionPattern = %d{yyyy-MM-dd HH:mm:ss,SSS} [%-5p] method:%l%n%m%n

#ERROR以上级别日志文件输出到logs/error.log  
log4j.appender.errorlog = org.apache.log4j.RollingFileAppender  
log4j.appender.errorlog.file = logs/error.log  
log4j.appender.errorlog.Append = true  
log4j.appender.errorlog.Threshold = ERROR  
log4j.appender.errorlog.MaxBackupIndex = 5  
log4j.appender.errorlog.layout = org.apache.log4j.PatternLayout  
log4j.appender.errorlog.layout.ConversionPattern = %d{yyyy-MM-dd HH:mm:ss,SSS} [%-5p] method:%l%n%m%n

##### 2.2.2 xml文件方式

首先在pom中添加如下依赖

*<!--添加logback依赖关系-->*<dependency>  
 <groupId>ch.qos.logback</groupId>  
 <artifactId>logback-core</artifactId>  
 <version>1.2.3</version>  
</dependency>

然后创建loopback.xml配置文件。

#### 2.3日志系统配置说明

##### 输出级别的种类

ERROR 为严重错误 主要是程序的错误

WARN 为一般警告，比如session丢失

INFO 为一般要显示的信息，比如登录登出

DEBUG 为程序的调试信息

**配置日志信息输出目的地**

**log4j.appender.appenderName=??**

org.apache.log4j.ConsoleAppender（控制台）

org.apache.log4j.FileAppender（文件）

org.apache.log4j.DailyRollingFileAppender（每天产生一个日志文件）

org.apache.log4j.RollingFileAppender（文件大小到达指定尺寸的时候产生一个新的文件）

org.apache.log4j.WriterAppender（将日志信息以流格式发送到任意指定的地方）

##### 配置日志信息的格式

**log4j.appender.appenderName.layout = ??**

org.apache.log4j.HTMLLayout（以HTML表格形式布局）

org.apache.log4j.PatternLayout（可以灵活地指定布局模式）

org.apache.log4j.SimpleLayout（包含日志信息的级别和信息字符串）

org.apache.log4j.TTCCLayout（包含日志产生的时间、线程、类别等等信息）

**ConsoleAppender选项**

Threshold=DEBUG:指定日志消息的输出最低层次。

ImmediateFlush=true:默认值是true，意味着所有的消息都会被立即输出。

Target=System.err:默认情况下是System.out,指定输出控制台

**FileAppender选项**

Threshold=DEBUG:指定日志消息的输出最低层次。

ImmediateFlush=true:默认值是true,意谓着所有的消息都会被立即输出。

File=mylog.txt:指定消息输出到mylog.txt文件。

Append=false:默认值是true,即将消息增加到指定文件中，false指将消息覆盖指定的文件内容。

**RollingFileAppender选项**

Threshold=DEBUG:指定日志消息的输出最低层次。

ImmediateFlush=true:默认值是true,意谓着所有的消息都会被立即输出。

File=mylog.txt:指定消息输出到mylog.txt文件。

Append=false:默认值是true,即将消息增加到指定文件中，false指将消息覆盖指定的文件内容。

MaxFileSize=100KB: 后缀可以是KB, MB 或者是 GB. 在日志文件到达该大小时，将会自动滚动，即将原来的内容移到mylog.log.1文件。

MaxBackupIndex=2:指定可以产生的滚动文件的最大数。

##### 日志信息格式中几个符号所代表的含义

其中%m%n等符号所代表的含义如下：    
　　                    －X号: X信息输出时左对齐；    
                        %p: 输出日志信息优先级，即DEBUG，INFO，WARN，ERROR，FATAL,    
                        %d: 输出日志时间点的日期或时间，默认格式为ISO8601，也可以在其后指定格式，比如：%d{yyy MMM dd HH:mm:ss,SSS}，输出类似：2002年10月18日 22：10：28，921    
                        %r: 输出自应用启动到输出该log信息耗费的毫秒数    
                        %c: 输出日志信息所属的类目，通常就是所在类的全名    
                        %t: 输出产生该日志事件的线程名    
                        %l: 输出日志事件的发生位置，相当于%C.%M(%F:%L)的组合,包括类目名、发生的线程，以及在代码中的行数。举例：Testlog4.main(TestLog4.java:10)    
                        %x: 输出和当前线程相关联的NDC(嵌套诊断环境),尤其用到像java servlets这样的多客户多线程的应用中。    
                        %%: 输出一个"%"字符    
                        %F: 输出日志消息产生时所在的文件名称    
                        %L: 输出代码中的行号    
                        %m: 输出代码中指定的消息,产生的日志具体信息    
                        %n: 输出一个回车换行符，Windows平台为"\r\n"，Unix平台为"\n"输出日志信息换行    
                        可以在%与模式字符之间加上修饰符来控制其最小宽度、最大宽度、和文本的对齐方式。如：    
                        %20c：指定输出category的名称，最小的宽度是20，如果category的名称小于20的话，默认的情况下右对齐。    
                        %-20c:指定输出category的名称，最小的宽度是20，如果category的名称小于20的话，"-"号指定左对齐。    
                        %.30c:指定输出category的名称，最大的宽度是30，如果category的名称大于30的话，就会将左边多出的字符截掉，但小于30的话也不会有空格。    
                        %20.30c:如果category的名称小于20就补空格，并且右对齐，如果其名称长于30字符，就从左边交远销出的字符截掉。

# 八、Hibernate持久化框架

### 1、介绍：

Hibernate是数据持久层的一个开源轻量级框架。是一个开源的ORM（object relations mapping）框架，提供了查询获取数据的方法，用面向对象的思想来操作数据库。

ORM（Object Relational Mapping）是对象关系映射，是一种解决实体对象与关系型数据库相互匹配的技术，其实现思想是：将数据库宏总的数据表映射为对象，对关系型数据以对象的形式进行操作。实质上，ORM在业务逻辑层与数据库层之间充当桥梁，它对对象（Object）到关系数据（Relational）进行映射（Mapping）。

在Hibernate框架中，主要使用了ORM的思想。Hibernate主要通过持久化类（\*.java）、Hibernate映射文件（\*.hbm.xml）或annotation方式、以及Hibernate配置文件（\*.cfg.xml）与数据库进行交互。其中，持久化类是操作对象，用于描述数据表的结构；映射文件指定持久化类与数据表之间的映射关系；配置文件用于指定Hibernate的属性信息等，如数据库的连接信息等。

**Hibernate框架特点：**

1）将对数据库的操作转换为对Java对象的操作，从而简化开发。通过修改一个“持久化”对象的属性从而修改数据库表中对应的记录数据。

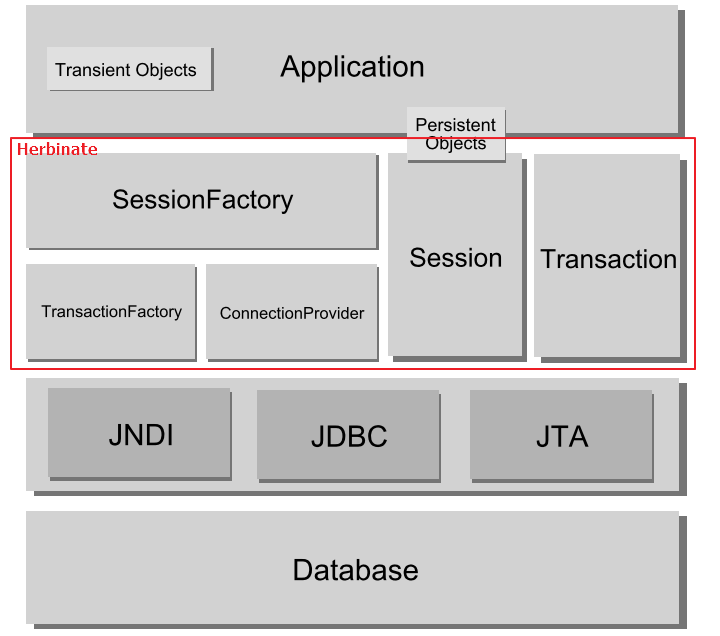
2）提供线程和进程两个级别的缓存提升应用程序性能。

3）有丰富的映射方式将Java对象之间的关系转换为数据库表之间的关系。

4）屏蔽不同数据库实现之间的差异。在Hibernate中只需要通过“方言”的形式指定当前使用的数据库，就可以根据底层数据库的实际情况生成适合的SQL语句。

5）非侵入式：Hibernate不要求持久化类实现任何接口或继承任何类，POJO即可。

### 2、体系架构：



Hibernate框架使用许多对象，如会话工厂、会话、事务等以及现有的Java API，如JDBC(Java数据库连接)，JTA(Java事务API)和JNDI(Java命名目录接口)。

#### 2.1中可以看出整个架构的核心接口：

1）SessionFactory (org.hibernate.SessionFactory)

针对单个数据库映射关系经过编译后的内存镜像，是线程安全的（不可变）。 它是生成Session的工厂，本身要用到ConnectionProvider。 该对象可以在进程或集群的级别上，为那些事务之间可以重用的数据提供可选的二级缓存。

2）Session (org.hibernate.Session)

表示应用程序与持久储存层之间交互操作的一个单线程对象，此对象生存期很短。 其隐藏了JDBC连接，也是Transaction的工厂。 其会持有一个针对持久化对象的必选（第一级）缓存，在遍历对象图或者根据持久化标识查找对象时会用到。

3）事务Transaction (org.hibernate.Transaction)

（可选的）应用程序用来指定原子操作单元范围的对象，它是单线程的，生命周期很短。 它通过抽象将应用从底层具体的JDBC、JTA以及CORBA事务隔离开。 某些情况下，一个Session之内可能包含多个Transaction对象。 尽管是否使用该对象是可选的，但无论是使用底层的API还是使用Transaction对象，事务边界的开启与关闭是必不可少的。

4）ConnectionProvider (org.hibernate.connection.ConnectionProvider)

（可选的）生成JDBC连接的工厂（同时也起到连接池的作用）。 它通过抽象将应用从底层的Datasource或DriverManager隔离开。 仅供开发者扩展/实现用，并不暴露给应用程序使用。

5）Query

Query接口让你方便地对数据库及持久对象进行查询，它可以有两种表达方式：HQL语言或本地数据库的SQL语句。Query经常被用来绑定查询参数、限制查询记录数量，并最终执行查询操作。

6）Configuration

Configuration 类的作用是对Hibernate 进行配置，以及对它进行启动。在Hibernate 的启动过程中，Configuration 类的实例首先定位映射文档的位置，读取这些配置，然后创建一个SessionFactory对象。虽然Configuration 类在整个Hibernate 项目中只扮演着一个很小的角色，但它是启动hibernate 时所遇到的第一个对象。

#### 2.2 实例状态

一个持久化类的实例可能处于三种不同状态中的某一种。这三种状态的定义则与所谓的持久化上下文(persistence context)有关。Hibernate的Session对象就是这个所谓的持久化上下文：

**瞬态（transient）**

该实例从未与任何持久化上下文关联过。它没有持久化标识（相当于主键值）。

**持久化(persistent)**

实例目前与某个持久化上下文有关联。它拥有持久化标识（相当于主键值），并且可能在数据库中有一个对应的行。 对于某一个特定的持久化上下文，Hibernate保证持久化标识与Java标识（其值代表对象在内存中的位置）等价。

**脱管(detached)**

实例曾经与某个持久化上下文发生过关联，不过那个上下文被关闭了，或者这个实例是被序列化(serialize)到另外的进程。它拥有持久化标识，并且在数据库中可能存在一个对应的行。对于脱管状态的实例，Hibernate不保证任何持久化标识和Java标识的关系。

*注：持久化是session打开后且未关闭期间的实例状态；态是在被应用程序实例化后，尚未进行持久化的对象；托管是因为实例化他们的Session已经被关闭而脱离持久化的对象。*

### 3、配置：

#### 3.1 hibernate.cfg.xml配置文件

<?xml version='1.0' encoding='utf-8'?>  
<!DOCTYPE hibernate-configuration PUBLIC  
 "-//Hibernate/Hibernate Configuration DTD//EN"  
 "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  
<hibernate-configuration>  
 <session-factory>  
 *<!--属性-->* <property name="hibernate.connection.url">

jdbc:mysql://192.168.56.102:3306/hibernate</property>  
 <property name="hibernate.connection.driver\_class">

com.mysql.cj.jdbc.Driver</property>  
 <property name="hibernate.connection.username">root</property>  
 <property name="hibernate.connection.password">mysql</property>  
 *<!--使在控制台显示Hibernate用到的SQL语句-->* <property name="hibernate.show\_sql">true</property>  
 <property name="hibernate.format\_sql">true</property>  
 <property name="connection.pool\_size">4</property>  
 <property name="hibernate.connection.release\_mode">auto</property>  
 *<!--自动在数据库中创建表-->* <property name="hbm2ddl.auto">update</property>  
 *<!--注意选择MySQLDialect时无法自动创建表-->* <property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>  
 *<!--映射定义文件-->*  
 <mapping resource="part2/hibernate/NewsEntity.hbm.xml"/>  
 <mapping resource="part2/hibernate/employee.hbm.xml"/>  
 *<!--添加使用注解的持久化类的映射-->* <mapping class="com.spring.action.idol.part2.hibernate.entity.Employee"/>  
 </session-factory>  
</hibernate-configuration>

配置属性说明

|  |  |
| --- | --- |
| **属性名** | **用途** |
| hibernate.dialect | 一个Hibernate Dialect类名允许Hibernate针对特定的关系数据库生成优化的SQL.  **取值** full.classname.of.Dialect |
| hibernate.show\_sql | 输出所有SQL语句到控制台. 有一个另外的选择是把org.hibernate.SQL这个log category设为debug。  **eg.** true | false |
| hibernate.format\_sql | 在log和console中打印出更漂亮的SQL。  **取值** true | false |
| hibernate.default\_schema | 在生成的SQL中, 将给定的schema/tablespace附加于非全限定名的表名上.  **取值** SCHEMA\_NAME |
| hibernate.default\_catalog | 在生成的SQL中, 将给定的catalog附加于非全限定名的表名上.  **取值** CATALOG\_NAME |
| hibernate.connection.<propertyName> | 将JDBC属性propertyName传递到DriverManager.getConnection()中去. |
| hibernate.hbm2ddl.auto | 在SessionFactory创建时，自动检查数据库结构，或者将数据库schema的DDL导出到数据库. 使用 create-drop时,在显式关闭SessionFactory时，将drop掉数据库schema.  取值 validate | update | create | create-drop |
| hibernate.jdbc.fetch\_size | 非零值，指定JDBC抓取数量的大小 (调用Statement.setFetchSize()). |
| hibernate.jdbc.batch\_size | 非零值，允许Hibernate使用JDBC2的批量更新.  取值 建议取5到30之间的值 |
| hibernate.jdbc.batch\_versioned\_data | 如果你想让你的JDBC驱动从executeBatch()返回正确的行计数 , 那么将此属性设为true(开启这个选项通常是安全的). 同时，Hibernate将为自动版本化的数据使用批量DML. 默认值为false.  eg. true | false |
| hibernate.jdbc.factory\_class | 选择一个自定义的Batcher. 多数应用程序不需要这个配置属性.  eg. classname.of.Batcher |
| hibernate.jdbc.use\_scrollable\_resultset | 允许Hibernate使用JDBC2的可滚动结果集. 只有在使用用户提供的JDBC连接时，这个选项才是必要的, 否则Hibernate会使用连接的元数据.  取值 true | false |
| hibernate.jdbc.use\_streams\_for\_binary | 在JDBC读写binary (二进制)或serializable (可序列化) 的类型时使用流(stream)(系统级属性).  取值 true | false |
| hibernate.jdbc.use\_get\_generated\_keys | 在数据插入数据库之后，允许使用JDBC3 PreparedStatement.getGeneratedKeys() 来获取数据库生成的key(键)。需要JDBC3+驱动和JRE1.4+, 如果你的数据库驱动在使用Hibernate的标 识生成器时遇到问题，请将此值设为false. 默认情况下将使用连接的元数据来判定驱动的能力.  取值 true|false |
| hibernate.connection.provider\_class | 自定义ConnectionProvider的类名, 此类用来向Hibernate提供JDBC连接.  取值 classname.of.ConnectionProvider |
| hibernate.connection.isolation | 设置JDBC事务隔离级别. 查看java.sql.Connection来了解各个值的具体意义, 但请注意多数数据库都不支持所有的隔离级别.  取值 1, 2, 4, 8 |
| hibernate.connection.autocommit | 允许被缓存的JDBC连接开启自动提交(autocommit) (不建议).  取值 true | false |

#### 3.2 映射配置文件（\*.hbm.xml）

两种方式：xml配置文件和annotation注解。

Xml配置文件，实例如下：

<?xml version='1.0' encoding='utf-8'?>  
<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">  
<hibernate-mapping>  
 <class name="com.spring.action.idol.part2.hibernate.entity.NewsEntity" table="News" schema="" catalog="hibernate">  
 <id name="id">  
 <generator class="increment"/>  
 </id>  
 <property name="title">  
 <column name="title" sql-type="char" length="16"/>  
 </property>  
 <property name="content">  
 <column name="content" sql-type="char" length="16"/>  
 </property>  
 <property name="fullContent">  
 <column name="fullContent" sql-type="char" length="16"/>  
 </property>  
 </class>  
</hibernate-mapping>

采用xml配置方式时需要在hibernate.cfg.xml文件中添加如下映射关系：

<mapping resource="part2/hibernate/NewsEntity.hbm.xml"/>

映射文件配置说明：

class元素

class元素用来定义一个持久化类。class元素的主要属性有：

①name：持久化类的完整名字。

②table：与持久化类对应的表名字。

③discriminator-value：置顶区分支区分不同的子类。

④polymorphism：多态性，默认值为implicit，可以指定为explicit。

⑤lazy：延迟加载，可以指定为true或false。

⑥abstract：抽象类，指定该类是否为抽象类。

id元素

①name：持久化类中的标识属性名字。

②type：标识属性的Hibernate类型。

③column：表中的主键字段。

generator元素

id元素下必须存在generator元素，用来指定标识属性的生成类，这些生成类都实现了IdentifierGenerator接口。

①increment：自动增加，用于为long/short/int类型生成唯一标识。

②identity：对DB2、MySQL等数据库的内置标识字段进行支持。

③native：根据数据库底层能力，悬着适合的生成方式。

④assigned：通过应用程序指定标识属性，是默认的生成策略。

⑤foreign：使用另外一个关联对象的标识作为标识属性。

property元素

class元素下往往存在大量的property元素。property元素映射了持久化类非标识属性与表字段的映射关系。

①name：持久化类中的属性名。

②type：属性的Hibernate类型。

③column：属性对应的表字段。

④update/insert：默认为true。定义当执行update以及insert操作时是否包含该属性。

⑤lazy：该属性是否延迟抓取。。

component元素

如果某个持久类的属性非常多，可以将某些属性封装到新的类中。

在映射文件中，即可使用component元素映射新类的属性和表字段。

subclass元素

如果持久化类存在子类，但是子类并没有映射具体的表，那么可以使用subclass元素定义子类属性与表字段的映射关系。

joined-subclass元素

如果持久化类对应一张表，同时持久化类存在子类，而且该子类对应具体的表，子类与父类对应的表存在关联关系，则使用joined-subclass元素定义子类属性与表字段的映射关系。

union-subclass元素

如果持久化类没有对应的表，而持久化类存在子类，子类都对应具体的表，那么可以使用union-subclass元素定义自雷属性与表字段的映射关系。

**采用注解的方式**：

|  |  |
| --- | --- |
| **@Entity** | 注释声明该类为持久类。 |
| **@Table**(name="tableName") | 持久性映射的表(表名="tableName)。@Table是类一级的注解,定义在@Entity下,为实体bean映射表,目录和schema的名字,默认为实体bean的类名。 |
| **@Id** | 注释可以表明哪种属性是该类中的独特标识符(即相当于数据表的主键)。 |
| **@GeneratedValue** | 定义自动增长的主键的生成策略. |
| **@Transient** | 将忽略这些字段和属性,不用持久化到数据库. |
| **@Temporal**(TemporalType.TIMESTAMP) | 声明时间格式 |
| **@Enumerated** | 声明枚举 |
| **@Version** | 声明添加对乐观锁定的支持 |
| **@OneToOne** | 可以建立实体bean之间的一对一的关联 |
| **@OneToMany** | 可以建立实体bean之间的一对多的关联 |
| **@ManyToOne** | 可以建立实体bean之间的多对一的关联 |
| **@ManyToMany** | 可以建立实体bean之间的多对多的关联 |
| **@Formula** | 一个SQL表达式，这种属性是只读的,不在数据库生成属性 |
| **@OrderBy** | Many端某个字段排序(List) |

采用注解方式时需要在hibernate.cfg.xml配置文件中添加如下映射语句:

<mapping class="com.spring.action.idol.part2.hibernate.entity.NewsEntity"/>

*注：在Idea中可以通过选择数据库中表自动生成映射配置文件和持久类*

#### 3.3 Generator类

<id name="id">  
 <generator class="native"/>  
</id>

生成器参数说明

|  |  |
| --- | --- |
| increment | 用于为long, short或者int类型生成唯一标识。只有在没有其他进程往同一张表中插入数据时才能使用。在集群下不要使用。 |
| identity | 对DB2,MySQL, MS SQL Server, Sybase和HypersonicSQL的内置标识字段提供支持。返回的标识符是long, short 或者int类型的。 |
| sequence | 在DB2,PostgreSQL, Oracle, SAP DB, McKoi中使用序列（sequence)， 而在Interbase中使用生成器(generator)。返回的标识符是long, short或者 int类型的。 |
| hilo | 使用一个高/低位算法高效的生成long, short 或者 int类型的标识符。高/低位算法生成的标识符只在一个特定的数据库中是唯一的。 |
| seqhilo | 使用一个高/低位算法来高效的生成long, short 或者 int类型的标识符，给定一个数据库序列（sequence)的名字。 |
| uuid | 用一个128-bit的UUID算法生成字符串类型的标识符， 这在一个网络中是唯一的（使用了IP地址）。UUID被编码为一个32位16进制数字的字符串。 |
| guid | 在MS SQL Server 和 MySQL 中使用数据库生成的GUID字符串。 |
| native | 根据底层数据库的能力选择identity, sequence 或者hilo中的一个。 |
| assigned | 让应用程序在save()之前为对象分配一个标示符，默认生成策略。 |
| select | 通过数据库触发器选择一些唯一主键的行并返回主键值来分配一个主键。 |
| foreign | 使用另外一个相关联的对象的标识符。通常和<one-to-one>联合起来使用。 |
| sequence  -identity | 一种特别的序列生成策略,使用数据库序列来生成实际值,但将它和JDBC3的getGeneratedKeys结合在一起,使得在插入语句执行的时候就返回生成的值。 |

#### 3.3 Spring 整合Hibernate后的配置过程：

##### 3.3.1 config.properties文件：

### JDBC configure ###  
#注意：在xml配置文件中配置数据库url时添加 ?useUnicode=true&characterEncoding=UTF-8  
#解决java中的中文存入mysql中乱码的问题  
jdbc.url = jdbc:mysql://192.168.56.102:3306/001\_Jump?useUnicode=true&characterEncoding=UTF-8  
jdbc.driverClassName = com.mysql.cj.jdbc.Driver  
jdbc.username = root  
jdbc.password = 123456  
jdbc.maxAge = 10000  
  
### Hibernate Configure ###  
hibernate.show\_sql = true  
hibernate.format\_sql = true  
hibernate.hbm2ddl.auto = update  
hibernate.dialect = org.hibernate.dialect.MySQL57Dialect  
hibernate.current\_session\_context\_class = thread  
hibernate.connection.pool\_size = 1  
### Cache Configure ###

##### 3.3.2 hibernate.xml配置实例（使用AOP切面处理声明式事务）：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xmlns:tx="http://www.springframework.org/schema/tx"  
 xmlns:aop="http://www.springframework.org/schema/aop"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.0.xsd  
 http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd  
 http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd ">  
  
 <!-- 加载 properties 属性文件-->  
 <context:property-placeholder location="classpath:config/config.properties"/>  
  
 <!-- 数据源配置 -->  
 <bean id="dataSource" class="org.apache.tomcat.jdbc.pool.DataSource" destroy-method="close">  
 <property name="url" value="${jdbc.url}"/>  
 <property name="driverClassName" value="${jdbc.driverClassName}"/>  
 <property name="username" value="${jdbc.username}"/>  
 <property name="password" value="${jdbc.password}"/>  
 <property name="maxWait" value="${jdbc.maxAge}"/>  
 </bean>  
  
 <!-- sessionFactory 配置 -->  
 <bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">  
 <property name="dataSource" ref="dataSource"/>  
 <property name="packagesToScan"  
 value="com.jump.account.base.entity"/>  
 <property name="hibernateProperties">  
 <props>  
 <prop key="hibernate.show\_sql">${hibernate.show\_sql}</prop>  
 <prop key="hibernate.format\_sql">${hibernate.format\_sql}</prop>  
 <prop key="hibernate.dialect">${hibernate.dialect}</prop>  
 <!-- update the database schema on startup -->  
 <prop key="hibernate.hbm2ddl.auto">${hibernate.hbm2ddl.auto}</prop>  
 <!-- JDBC connection pool (use the built-in) -->  
 <prop key="hibernate.connection.pool\_size">${hibernate.connection.pool\_size}</prop>  
 </props>  
 </property>  
 </bean>  
  
 <!-- 配置事务管理器 -->  
 <bean id="transactionManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">  
 <property name="sessionFactory" ref="sessionFactory"/>  
 </bean>  
  
 <!-- <tx:annotation-driven/> 和下面的tx:advice,aop 属于两种处理事务方式-->  
 <!-- 第一种：通过使用注解 @transactional 进行事务驱动-->  
 <!--<tx:annotation-driven/>-->  
  
 <!-- 第二种 使用aop切面处理事务 -->  
 <!-- 定义事务通知 -->  
 <tx:advice id="txAdvice">  
 <tx:attributes>  
 <tx:method name="insert" propagation="REQUIRED" read-only="false"/>  
 <tx:method name="\*" propagation="REQUIRED" read-only="false"/>  
 </tx:attributes>  
 </tx:advice>  
  
 <!-- 定义事务切面 并应用事务通知 -->  
 <aop:config proxy-target-class="true">  
 <aop:pointcut id="pointcut" expression="execution(\* com.jump.account.base.dao.\*.\*(..))"/>  
 <aop:advisor advice-ref="txAdvice" pointcut-ref="pointcut"/>  
 </aop:config>  
</beans>

在Dao层使用注解声明为bean即可使用。

@Repository  
public class AccountDaoImpl extends BaseDaoImpl<Account> implements IAccountDao<Account> { }

### 4、日志

Hibernate使用log4j或者slf4j来为各种事件记录日志。

表 4.1  Hibernate日志类别

|  |  |
| --- | --- |
| **类别** | **功能** |
| org.hibernate.SQL | 在所有SQL DML语句被执行时为它们记录日志 |
| org.hibernate.type | 为所有JDBC参数记录日志 |
| org.hibernate.tool.hbm2ddl | 在所有SQL DDL语句执行时为它们记录日志 |
| org.hibernate.pretty | 在session清洗(flush)时，为所有与其关联的实体(最多20个)的状态记录日志 |
| org.hibernate.cache | 为所有二级缓存的活动记录日志 |
| org.hibernate.transaction | 为事务相关的活动记录日志 |
| org.hibernate.jdbc | 为所有JDBC资源的获取记录日志 |
| org.hibernate.hql.AST | 在解析查询的时候,记录HQL和SQL的AST分析日志 |
| org.hibernate.secure | 为JAAS认证请求做日志 |
| org.hibernate | 为任何Hibernate相关信息做日志 (信息量较大, 但对查错非常有帮助) |

### 5、持久类

在应用程序中，用来实现业务的实体类就是持久类。

持久化类使用POJO（plain ordinary java object  简单java对象）类。如：

package com.spring.action.idol.part2.hibernate.entity;  
public class Employee {  
 private int id;  
 private String firstName;  
 private String lastName;  
 public int getId() {  
 return id;  
 }  
 public void setId(int id) {  
 this.id = id;  
 }  
 public String getFirstName() {  
 return firstName;  
 }  
 public void setFirstName(String firstName) {  
 this.firstName = firstName;  
 }  
 public String getLastName() {  
 return lastName;  
 }  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 }  
}

### 6、关联关系映射

#### 6.1 单向 one-to-one映射

两个对象之间一对的关系，例如：Person（人）-IdCard（身份证）。

两种策略：

1）主键关联：即让两个对象具有相同的主键值，以表明它们之间的一一对应的关系；数据库表不会有额外的字段来维护它们之间的关系，仅通过表的主键来关联。

2）唯一外键关联（常用）：外键关联，本来是用于多对一的配置，但是加上唯一的限制之后（采用<many-to-one>标签来映射，指定多的一端unique为true，这样就限制了多的一端的多重性为一），也可以用来表示一对一关联关系，其实它就是多对一的特殊情况。

<?xml version='1.0' encoding='utf-8'?>  
<!DOCTYPE hibernate-mapping PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">  
<hibernate-mapping>  
 <class name="com.spring.action.idol.part2.hibernate.hibernate2.entity.Person" table="z\_person1\_1">  
 <id name="id">  
 <generator class="native">  
 *<!-- property只关联对象 -->* <param name="property">idCard</param>  
 </generator>  
 </id>  
 <property name="name"/>  
 *<!--单向one-to-one映射配置-->  
 <!--&lt;!&ndash; 主键关联：one-to-one映射&ndash;&gt;  
 <one-to-one name="idCard" constrained="true"/>-->  
 <!--唯一外键关联:-->* <many-to-one name="address" unique="true" cascade="save-update"/>  
 </class>  
  
 <class name="com.spring.action.idol.part2.hibernate.hibernate2.entity.Address" table="z\_address1\_1">  
 <id name="id">  
 <generator class="native"/>  
 </id>  
 <property name="address"/>  
 </class>  
</hibernate-mapping>

6.2 单向many-to-one映射

多对一关联映射原理：在多的一端加入一个外键，指向一的一端。

关键映射代码：在多的一端加入如下标签映射：

<?xml version='1.0' encoding='utf-8'?>  
<!DOCTYPE hibernate-mapping PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">  
<hibernate-mapping>  
 <class name="com.spring.action.idol.part2.hibernate.hibernate2.entity.Person" table="z\_person1\_2">  
 <id name="id" column="person\_id">  
 <generator class="native">  
 </generator>  
 </id>  
 <property name="name"/>  
 *<!--单向many-to-one映射配置-->* <many-to-one name="address" column="address\_id" cascade="save-update"/>  
 </class>  
  
 <class name="com.spring.action.idol.part2.hibernate.hibernate2.entity.Address" table="z\_address1\_2">  
 <id name="id" column="address\_id">  
 <generator class="native"/>  
 </id>  
 <property name="address"/>  
 </class>  
</hibernate-mapping>

#### 6.3 单向one-to-many映射

一对多关联映射原理：在一的一端加入一个外键，指向多的一端。

关键映射代码：在多的一端加入如下标签映射：

<set name="addresses" cascade="all">  
 <key column="person\_id"></key>  
 <one-to-many class="com.spring.action.idol.part2.hibernate2.entity.Address"/>  
</set>

#### 6.4 单向 many-to-many映射

多对多关联映射时需新增加一张表才完成基本映射。

关键映射代码：只需在Person的一端加入如下标签映射address侧不需添加：

<set name="addresses" cascade="all" table="z\_person\_address">  
 <key column="person\_id"/>  
 <many-to-many  
 column="address\_id"  
 class="com.spring.action.idol.part2.hibernate2.entity.Address">  
 </many-to-many>  
</set>

#### 6.5 双向one-to-one映射

映射时需要在两个对象Person和IdCard中都加入<one-to-one>标签。

同样也有两种方式：基于外键关联的双向一对一关联和基于主键关联的一对一关联。

注意：采用基于主键关联的一对一关联需要使用特定的id生成器（即foreigh）。

<?xml version='1.0' encoding='utf-8'?>  
<!DOCTYPE hibernate-mapping PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">  
<hibernate-mapping>  
 <class name="com.spring.action.idol.part2.hibernate2.entity.Person" table="z\_person">  
 <id name="id">  
 <generator class="native"/>  
 </id>  
 <property name="name"/>  
 *<!--双向 one-to-one 映射-->  
 <!--基于外键关联-->* <many-to-one name="idCard" unique="true" cascade="save-update"/>  
 </class>  
 <class name="com.spring.action.idol.part2.hibernate2.entity.IDCard" table="z\_idCard">  
 <id name="id" column="idCard\_id">  
 <generator class="native"/>  
 </id>  
 <property name="cardNo" column="cardNo"/>  
 <one-to-one name="person" constrained="true"/>  
 </class>  
</hibernate-mapping>

与单向one-to-one对比：

1）主键关联时关联主键的值相同。

2）生成的表结构相同，双向时只是改变了一对一关联映射的加载。

#### 6.6 双向one-to-many映射（重点）

双向多对一关联 是最常见的关联关系。（这也是标准的父/子关联关系。）

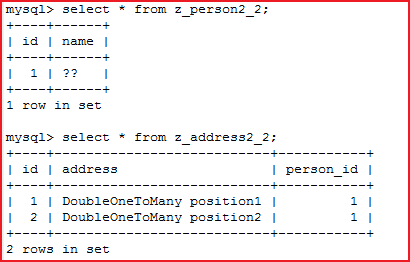
从配置文件上看，一对多双向关联的配置文件中在多的一端的配置文件上存在<many-to-one />的相关配置，即保证多对一的映射，这就保证了双向。

<?xml version='1.0' encoding='utf-8'?>  
<!DOCTYPE hibernate-mapping PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">  
<hibernate-mapping>  
 <class name="com.spring.action.idol.part2.hibernate2.entity.Person" table="z\_person2\_2">  
 <id name="id">  
 <generator class="native"/>  
 </id>  
 <property name="name"/>  
 *<!--双向 one-to-many/many-to-one 映射-->* <set name="addresses" inverse="true">  
 <key column="person\_id"/>  
 <one-to-many class="com.spring.action.idol.part2.hibernate2.entity.Address"/>  
 </set>  
 </class>  
 <class name="com.spring.action.idol.part2.hibernate2.entity.Address" table="z\_address2\_2">  
 <id name="id">  
 <generator class="native"/>  
 </id>  
 <property name="address"/>  
 <many-to-one name="person" column="person\_id"/>  
 </class>  
</hibernate-mapping>

*注：\* inverse属性可以用在一对多和多对多双向关联上，inverse属性默认为false，为false表示本端可以维护关系，如果inverse为true，则本端不能维护关系，会交给另一端维护关系，本端失效。所以一对多关联映射我们通常在多的一端维护关系，让一的一端失效。*

*\* inverse是控制方向上的反转，只影响存储。*

写入到数据库中的信息如下：



单向和双向性其实是通过使用<one-to-many>和<many-to-one>来实现的，<one-to-many>重在设置关联关系，并不会生成新列，<many-to-one>在生成关联关系的同时会生成新列。

#### 6.7 双向many-to-many映射

双向的目的就是为了两端都能将对方加载上来，和单向多对多的区别就是双向需要在两端都加入标签映射，需要注意的是：

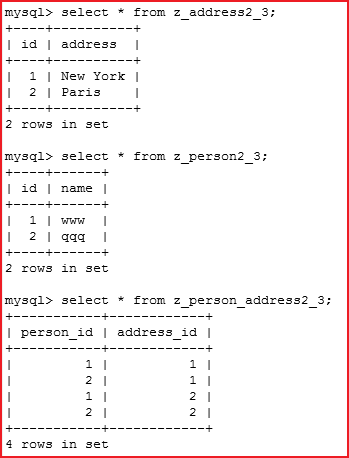
1）生成的中间表名称必须一样;

2）生成的中间表中的字段必须一样;

映射文件配置：

<?xml version='1.0' encoding='utf-8'?>  
<!DOCTYPE hibernate-mapping PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">  
<hibernate-mapping>  
 <class name="com.spring.action.idol.part2.hibernate2.entity.Person" table="z\_person2\_3">  
 <id name="id">  
 <generator class="native"/>  
 </id>  
 <property name="name"/>  
 *<!--双向 many-to-many 映射-->* <set name="addresses" table="z\_person\_address2\_3" cascade="save-update">  
 <key column="person\_id"/>  
 <many-to-many column="address\_id" class="com.spring.action.idol.part2.hibernate2.entity.Address"/>  
 </set>  
 </class>  
 <class name="com.spring.action.idol.part2.hibernate2.entity.Address" table="z\_address2\_3">  
 <id name="id">  
 <generator class="native"/>  
 </id>  
 <property name="address"/>  
 <set name="persons" table="z\_person\_address2\_3" cascade="save-update">  
 <key column="address\_id"/>  
 <many-to-many column="person\_id"  
 class="com.spring.action.idol.part2.hibernate2.entity.Person"/>  
 </set>  
 </class>  
</hibernate-mapping>

写入到数据库中的信息如下：



7、Spring集合hibernate框架

增加依赖包：

### 8、HQL语言（面向对象语言）

1）.使用别名时，select 后面必须接别名，否则报错！

String hql = "select a from AuthenticationInfo as a " +  
 "where a.userInfo.userName=:userName and a.userInfo.userName=:mobilephone;

2）模糊查询

public List<T> queryByKeyword(String keyword) {  
 *LOGGER*.debug("receive keyword:{}", keyword);  
 //方式一 使用变量 :keyword  
 /\*String hql = "from " + clazz.getSimpleName() + " as a where a.message like :keyword";  
 Query query = getCurrentSession().createQuery(hql);  
 query.setParameter("keyword", "%" + keyword + "%");\*/  
   
 //方式二 使用占位符  
 String hql = "from Account where url like ?";  
 Query query = getCurrentSession().createQuery(hql);  
 query.setParameter(0, "%" + keyword + "%");  
 return query.list();  
}

Druid 参数

Druid 参数

|  |  |  |  |
| --- | --- | --- | --- |
| 配置参数 | 缺省值 | 游戏服设置的值 | 参数说明 |
| initialSize | 0 | 4 | 初始化连接数量 |
| minIdle | 0 | 4 | 最小空闲连接数 |
| maxActive | 8 | 8 | 最大并发连接数 |
| maxWait | -1L | 60000 | 获取连接时最大等待时间，单位毫秒。配置了maxWait之后， 缺省启用公平锁，并发效率会有所下降， 如果需要可以通过配置useUnfairLock属性为true使用非公平锁。 |
| timeBetweenEvictionRunsMillis | 60000 | 60000 | 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒 Destroy线程会检测连接的间隔时间 |
| minEvictableIdleTimeMillis | 1800000 | 1800000 | 配置一个连接在池中最小生存的时间，单位是毫秒 |
| validationQuery | null | select 1 | 用来检测连接是否有效的sql，要求是一个查询语句 |
| testOnBorrow | FALSE | FALSE | 申请连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能。 |
| testOnReturn | FALSE | FALSE | 归还连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能 |
| testWhileIdle | TRUE | TRUE | 建议配置为true，不影响性能，并且保证安全性。 申请连接的时候检测，如果 空闲时间大于 timeBetweenEvictionRunsMillis， 执行validationQuery检测连接是否有效。 |
| poolPreparedStatements | FALSE | TRUE | false 是否缓存preparedStatement，也就是PSCache。 PSCache对支持游标的数据库性能提升巨大，比如说oracle。 在mysql5.5以下的版本中没有PSCache功能，建议关闭掉。 5.5及以上版本有PSCache，建议开启。 |
| maxPoolPreparedStatementPerConnectionSize | 10 | 100 | 要启用PSCache，必须配置大于0，当大于0时， poolPreparedStatements自动触发修改为true。 单个connnection独享一个statement cache，也就是说maxOpenPreparedStatements是针对单个connection链接的 |

### 9、项目实例：

#### 9.1 idea配置

1、安装前准备：

在centos上安装mysql数据库，作为Hibernate连接使用的数据库。

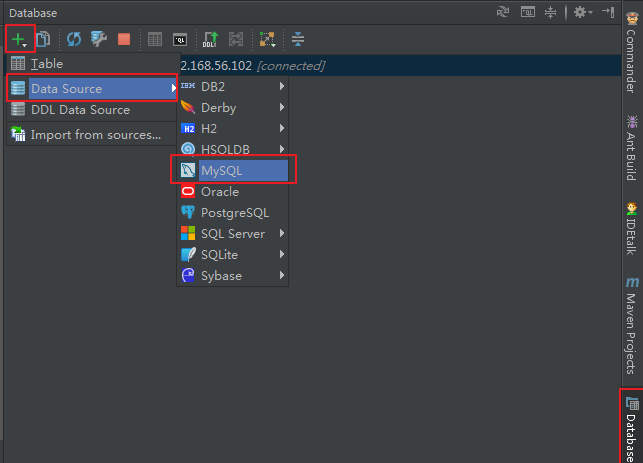
2、建立maven项目，添加Hibernate依赖关系

<dependency>  
 <groupId>org.hibernate</groupId>  
 <artifactId>hibernate-core</artifactId>  
 <version>5.2.10.Final</version>  
</dependency>  
<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>6.0.6</version>  
</dependency>

3、配置hibernate.cfg.xml配置文件

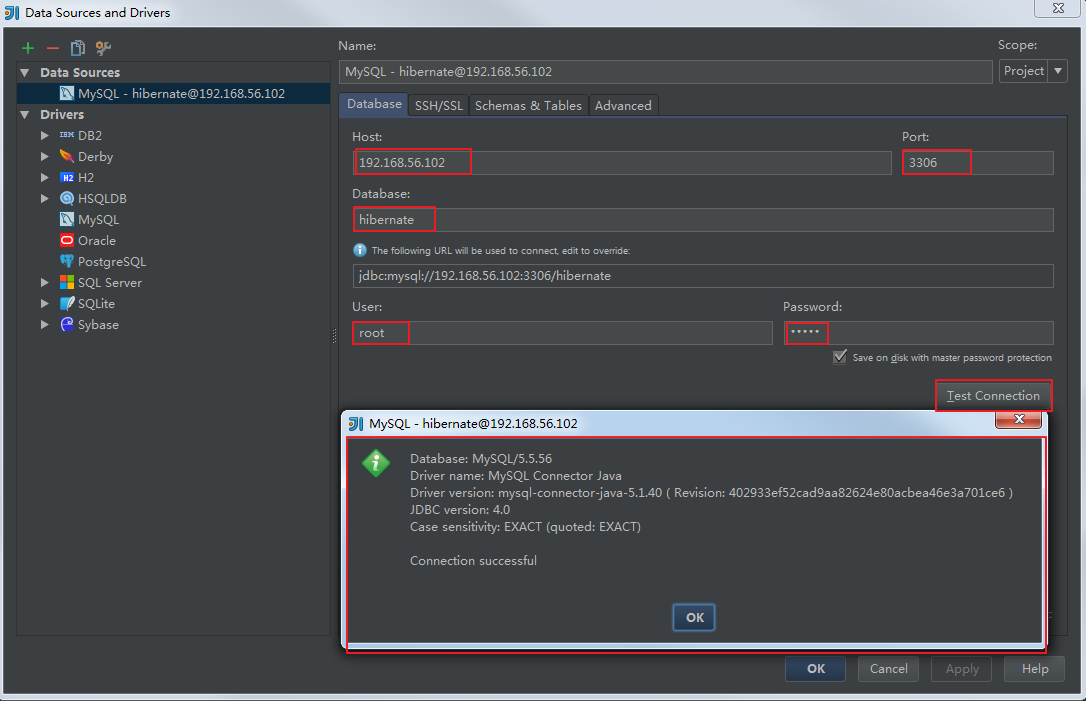
<?xml version='1.0' encoding='utf-8'?>  
<!DOCTYPE hibernate-configuration PUBLIC  
 "-//Hibernate/Hibernate Configuration DTD//EN"  
 "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  
<hibernate-configuration>  
 <session-factory>  
 *<!--设置连接的数据库-->* <property name="hibernate.connection.url">jdbc:mysql://192.168.56.102:3306/hibernate</property>  
 *<!--数据库驱动-->* <property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>  
 <property name="hibernate.connection.username">root</property>  
 <property name="hibernate.connection.password">mysql</property>  
 <property name="hibernate.show\_sql">true</property>  
 <property name="connection.pool\_size">4</property>  
 <mapping resource="part2/hibernate/StudentsEntity.hbm.xml"/>  
 <mapping resource="part2/hibernate/NewsEntity.hbm.xml"/>  
 </session-factory>  
</hibernate-configuration>

4、在idea中连接mysql数据库

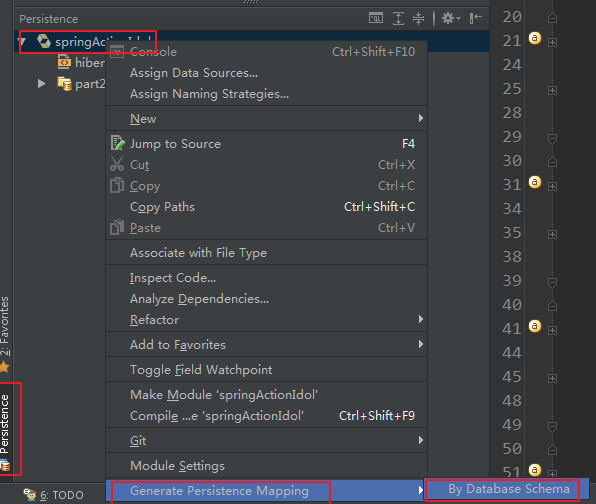


注：如果找不到view->database，则需要添加Database Davigator的插件。

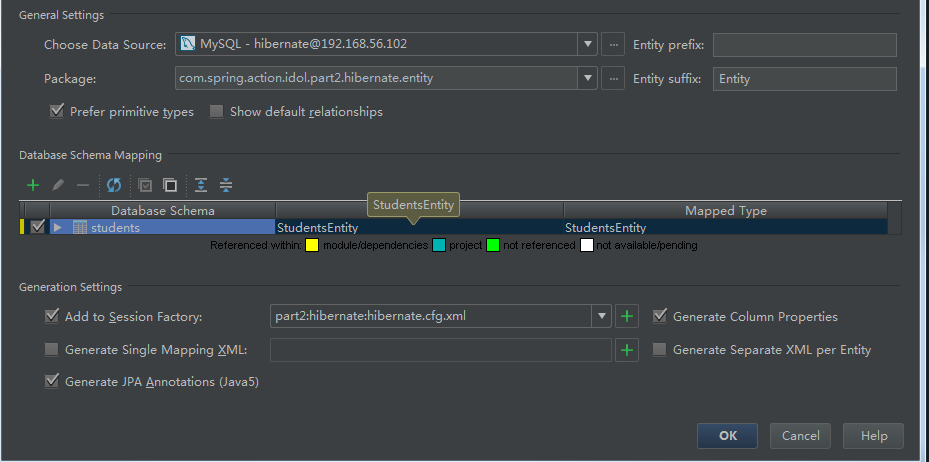
设置mysql服务器，点击Test Connection ，如下显示连接成功。



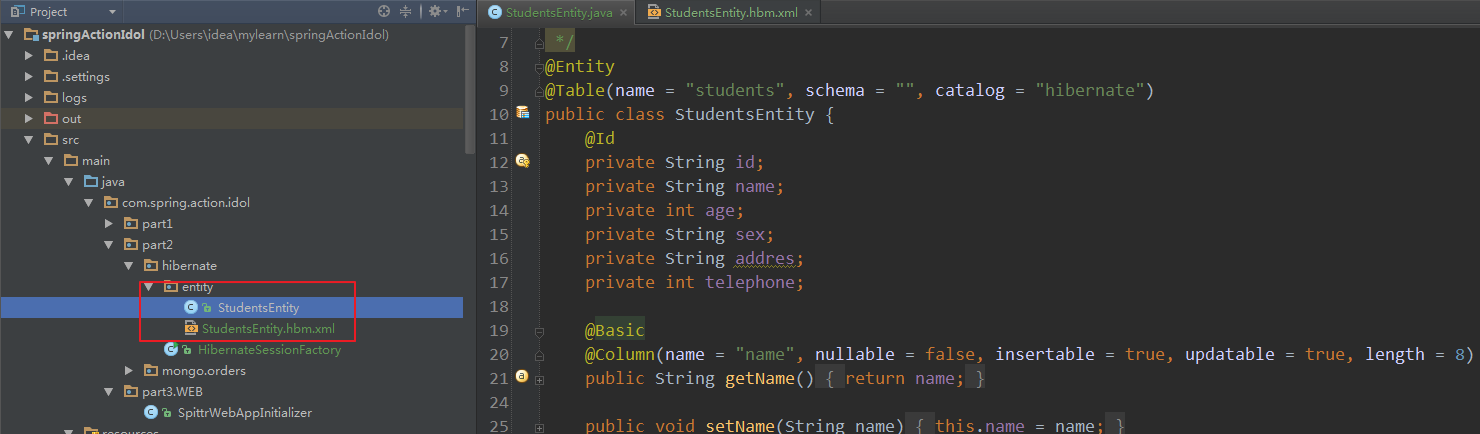
4、到IDE左下角找到 Persistence 标签页，并点开它，右键点击项目名称spring\*



点击 Generate Persistence Mapping > By Database Schema



生成成功后会生成students的实体类。



import org.hibernate.boot.registry.StandardServiceRegistryBuilder;  
  
import java.util.Map;  
  
/\*\*  
 \* Created by zhangp on 2017/5/5.  
 \*/  
public class Main {  
 private static final SessionFactory *ourSessionFactory*;  
 private static final ServiceRegistry *serviceRegistry*;  
  
 static {  
 try {  
 Configuration configuration = new Configuration();  
 configuration.configure();  
  
 *serviceRegistry* = new StandardServiceRegistryBuilder().applySettings(configuration.getProperties()).build();

#### 9.2 spring boot集成ehcache 2x 用于hibernate二级缓存

# 九、正则表达式

1、 表示空格

" \\s"， "[ ]"， "[\\s]"

表示多个空格

"\\s+"， "[ ]+"， "[\\s]+"

2、 表示数字

 "\\d"， "[\\d]"， "[0-9]"

表示多个数字，同理，在后面加上"+"

# 十、VS2017开发JNI接口

创建空项目

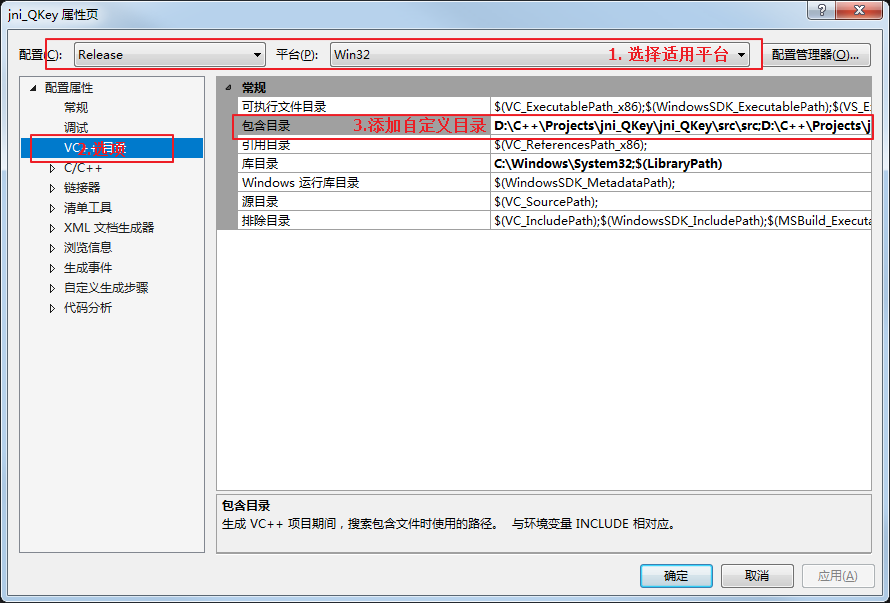
### 添加头文件目录

如需要添加jni.h的头文件，目录为

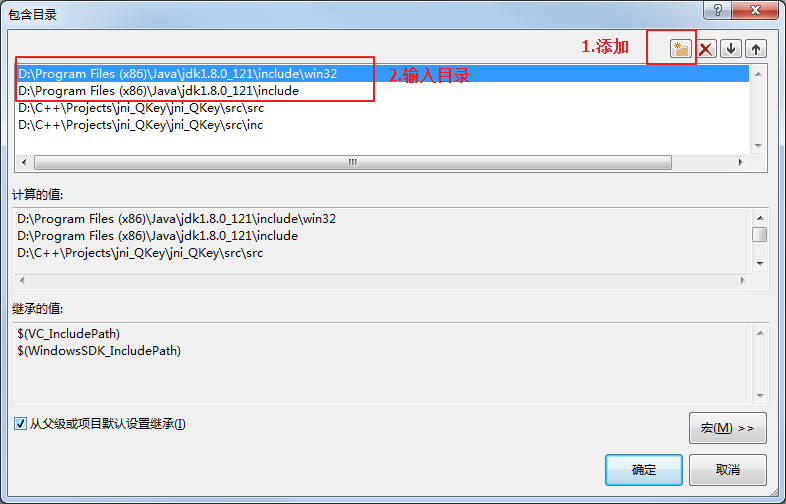
D:\Program Files (x86)\Java\jdk1.8.0\_121\include

D:\Program Files (x86)\Java\jdk1.8.0\_121\include\win32

,右键项目名称，选择项目属性，在VC++-->包含目录 中添加自定义路径

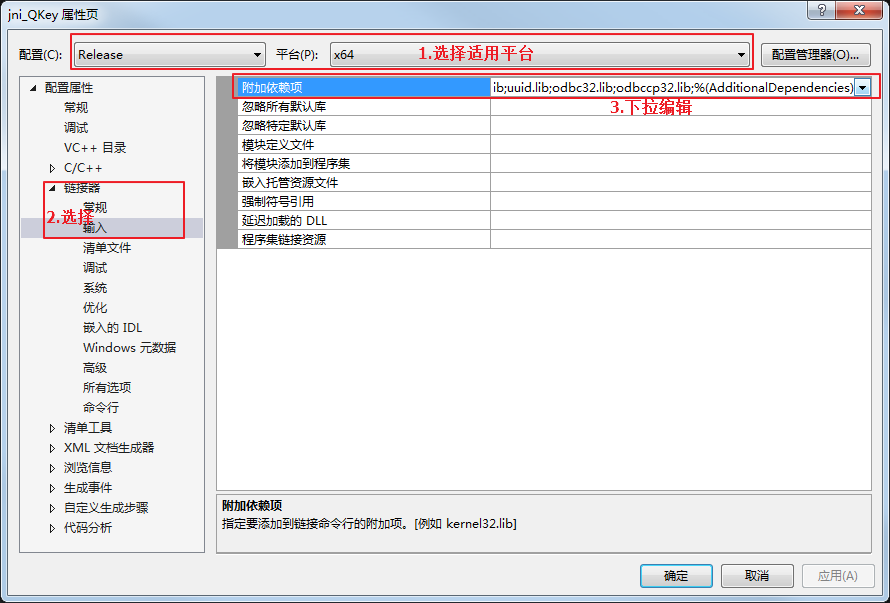


在弹出的窗口中添加java中的jni.h头文件所在目录，如下

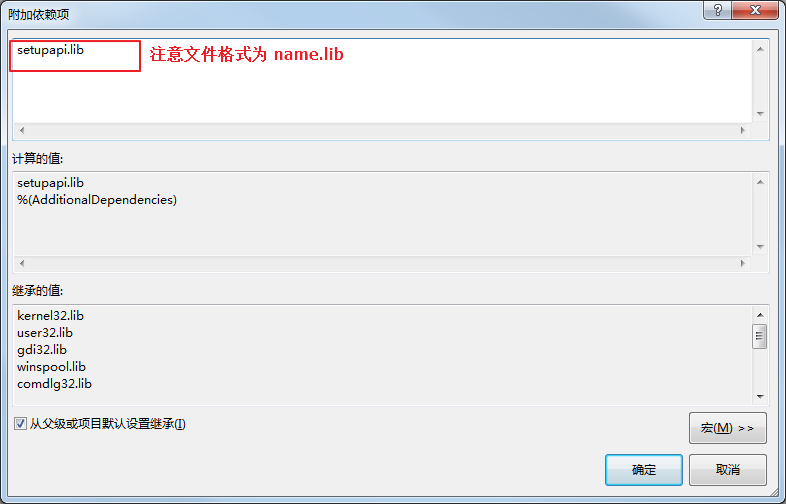


### 添加Windows下的setupapi.dll动态库

项目属性中，找到连接器->输入->添加依赖项新增，如下图



注意附加的依赖项格式，如这里需要添加setupapi.dll，填写的内容setupapi.lib。要以.lib结尾。



# 性能优化

1.并发时发现加解密初始化耗时较多，加解密使用并发队列ConcurrentLinkedQueue，一次性完成初始化，后续通过ThreadLocal获取加解密对象，提高性能。

# Linux脚本

## pgrep查询mongo集群进程并杀掉所有mongo进程

[root@Centos6 roc]# pgrep mongo

31286

31323

31360

31392

[root@Centos6 roc]# pgrep mongo|xargs kill -9

[root@Centos6 roc]# pgrep mongo

[root@Centos6 roc]#

# 问题记录：

### 1、Junit测试时spring自动注入失败---已解决

问题代码：

public class TestMongoTemplate {

//此处注入失败，运行test程序时抛出null异常  
 @Autowired  
 private MongoTemplate mongoTemplate;  
 @Test  
 public void testFindAndModify() {  
 Person p1 = new Person("p1", 21);  
 mongoTemplate.insert(p1);  
 Query query = new Query(Criteria.*where*("name").is("p1"));  
 Update update = new Update().inc("age", 31);  
 Person p11 = mongoTemplate.findAndModify(query, update, Person.class);  
 System.*out*.println(p1.toString());  
 }  
}

xml文件（未发现异常）：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context" xmlns:mongo="http://www.springframework.org/schema/data/mongo" xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  
http://www.springframework.org/schema/context  
http://www.springframework.org/schema/context/spring-context.xsd  
http://www.springframework.org/schema/data/mongo http://www.springframework.org/schema/data/mongo/spring-mongo.xsd">  
 *<!--自动注解配置开启-->*  
 <context:component-scan base-package="com.mongo.idol"/>  
 <context:annotation-config></context:annotation-config>  
 *<!-- Default bean name is 'mongo' -->* <mongo:mongo-client host="localhost" port="27017">  
 <mongo:client-options connections-per-host="8"   
 connect-timeout="1000"/>  
 </mongo:mongo-client>  
 <mongo:db-factory id="mongoDbFactory"  
 dbname="database"  
 mongo-ref="mongo"/>

<!—需要注入的bean-->  
 <bean

id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">  
 <constructor-arg ref="mongoDbFactory"/>  
 </bean>  
</beans>

原因：

使用Junit4.4测试时，需要在类上配置好红色的Annotation注解，由于没有配置，故注入失败：

@RunWith(SpringJUnit4ClassRunner.class) 用于配置spring中测试的环境

@ContextConfiguration(Locations="../applicationContext.xml") 用于指定配置文件所在的位置

@Test标注在方法前，表示其是一个测试的方法 无需在其配置文件中额外设置属性.

解决方案：

添加需要的注解，成功注入的代码如下：

/\*\*

\*测试前需要配置@RunWith和@ContextConfiguration,谨记！否则注入会抛出NullPointerException\*/  
@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration(locations = "classpath\*:/spring-mongo.xml")  
public class TestMongoTemplate {  
 @Autowired  
 private MongoTemplate mongoTemplate;  
 @Test  
 public void testFindAndModify() {…}

#### 1.1 Spring定时任务 Could not find default TaskScheduler bean异常处理

2017-05-15 16:05:27,558 DEBUG [main] (ScheduledAnnotationBeanPostProcessor.java->finishRegistration:238) - Could not find default TaskScheduler bean

org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type 'org.springframework.scheduling.TaskScheduler' available

...

2017-05-15 16:05:27,568 DEBUG [main] (ScheduledAnnotationBeanPostProcessor.java->finishRegistration:258) - Could not find default ScheduledExecutorService bean

org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type 'java.util.concurrent.ScheduledExecutorService' available

个人使用中出现了2个问题：

　　解决：这个错误问题其实是debug级别输出，就是不影响定时任务的使用，在网上看到别人有这样解释

　　Spring的定时任务调度器会尝试获取一个注册过的 task scheduler来做任务调度，它会尝试通过BeanFactory.getBean的方法来获取一个注册过的scheduler bean，获取的步骤如下：

　　a.尝试从配置中找到一个TaskScheduler Bean

　　b.寻找ScheduledExecutorService Bean

　　c.使用默认的scheduler

　　前两步，如果找不到的话，就会以debug的方式抛出异常，

　　分别是： logger.debug("Could not find default TaskScheduler bean", ex);

　　logger.debug("Could not find default ScheduledExecutorService bean", ex);

　　 所以，日志中打印出来的两个异常，根本不是什么错误信息，也不会影响定时器的使用，只不过是spring的自己打印的一些信息罢了

　　如果想去掉这个输出，可以在log4j配置文件中增加一行即可：

*log4j.logger.org.springframework.scheduling = INFO*

　　这个问题后来发现是由于Spring的配置文件被加载了两次造成的，listener和DispatcherServlet都会初始化spring配置文件，所有注释掉listener即可

#### 1.2 jar运行时提示“没有主清单属性”

报错：*D:\Users\idea\mylearn\test2\target\AAAA-1.0-SNAPSHOT.jar中没有主清单属性*

解决：用解压工具打开AAAA-1.0-SNAPSHOT.jar包，找到 MANIFEST.MF文件，由于idea插件生成的jar包时MF文件中缺少Main-Class信息，添加下面一行更新jar包即可。

Main-Class: com.test.Main

注意：

1）com.test.Main后面必须敲下回车，使光标在下一行，否则仍报“*没有主清单属性*”的错误。

2）冒号后面需要有一个空格，否则运行jar包报错“*Error: Invalid or corrupt jarfile D:\Users\idea\mylearn\test2\target\AAAA-1.0-SNAPSHOT.jar*”

### 2、HibernateSessionFactory.java 运行时提示Mapping（RESOURSE）not found。

报错如下：

Exception in thread "main" java.lang.ExceptionInInitializerError

at com.spring.action.idol.part2.hibernate.HibernateSessionFactory.<clinit>(HibernateSessionFactory.java:41)

at java.lang.Class.forName0(Native Method)

at java.lang.Class.forName(Class.java:264)

at com.intellij.rt.execution.application.AppMain.main(AppMain.java:116)

Caused by: org.hibernate.boot.MappingNotFoundException: Mapping (RESOURCE) not found : com/spring/action/idol/part2/hibernate/entity/StudentsEntity.hbm.xml : origin(com/spring/action/idol/part2/hibernate/entity/StudentsEntity.hbm.xml)

at org.hibernate.boot.spi.XmlMappingBinderAccess.bind(XmlMappingBinderAccess.java:56)

at org.hibernate.boot.MetadataSources.addResource(MetadataSources.java:274)

at org.hibernate.boot.MetadataSources.addClass(MetadataSources.java:262)

at org.hibernate.cfg.Configuration.addClass(Configuration.java:513)

at com.spring.action.idol.part2.hibernate.HibernateSessionFactory.<clinit>(HibernateSessionFactory.java:33)

... 3 more

说明：

Configuration cfg = new Configuration()  
 .configure("part2/hibernate/hibernate2/hibernate2.cfg.xml")  
 .addResource("part2/hibernate/hibernate2/single/one2one.hbm.xml");

### 3、Hibernate执行时报错ERROR：object references an unsaved transient instance

13:41:24,082 ERROR ExceptionMapperStandardImpl:39-HHH000346: Error during managed flush [org.hibernate.TransientObjectException: object references an unsaved transient instance - save the transient instance beforeQuery flushing: com.spring.action.idol.part2.hibernate1.hibernate2.entity.IDCard]

解决方法：

添加cascade="save-update"属性。

<?xml version='1.0' encoding='utf-8'?>  
<!DOCTYPE hibernate-mapping PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">  
  
<hibernate-mapping>  
 <class name="com.spring.action.idol.part2.hibernate.hibernate2.entity.Person" table="z\_person1\_1">  
 <id name="id">  
 <generator class="native">  
 *<!-- property只关联对象 -->* <param name="property">idCard</param>  
 </generator>  
 </id>  
 <property name="name"/>  
<many-to-one name="address" unique="true" cascade="save-update"/>  
 </class>  
  
 <class name="com.spring.action.idol.part2.hibernate.hibernate2.entity.Address" table="z\_address1\_1">  
 <id name="id">  
 <generator class="native"/>  
 </id>  
 <property name="address"/>  
 </class>  
</hibernate-mapping>

#### 3.1 Spring-hibernate：org.springframework.beans.factory.UnsatisfiedDependencyException?已解决。

2017-05-11 08:34:27,864 [ERROR] org.springframework.test.context.TestContextManager.prepareTestInstance(TestContextManager.java:234) Caught exception while allowing TestExecutionListener [org.springframework.test.context.support.DependencyInjectionTestExecutionListener@3f49dace] to prepare test instance [com.spring.hibernate3.TestBlogDao@2d7275fc]  
java.lang.IllegalStateException: Failed to load ApplicationContext  
 …

Caused by: org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'blogDao': Unsatisfied dependency expressed through field 'transactionManager'; nested exception is org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'transactionManager' defined in class path resource [part2/hibernate/hibernate3/spring-hibernate.xml]: Cannot resolve reference to bean 'factory' while setting bean property 'sessionFactory'; nested exception is org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'factory' defined in class path resource [part2/hibernate/hibernate3/spring-hibernate.xml]: Invocation of init method failed; nested exception is org.hibernate.MappingException: Repeated column in mapping for entity: com.spring.action.idol.part2.hibernate.hibernate3.entity.BlogItem column: TITLE (should be mapped with insert="false" update="false")…

原因blogItem表中两个列的名字重复，修改列名称后解决:

*<property name="title" column="TITLE" not-null="true"/>  
<property name="text" column="TITLE" not-null="true"/>*

#### 3.2 spring-hibernate：Java.sql.SQLSyntaxErrorException: 异常。已解决

Java.sql.SQLSyntaxErrorException:You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '(6), BLOG\_ID bigint not null, primary key (BLOG\_ITEM\_ID) ' at line 5…

原因：hibernate方言版本配置错误，

<prop key="hibernate.dialect">org.hibernate.dialect.MySQL57Dialect</prop>

变更为：

<prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>

### 4 Java实现AES加密,异常java.security.InvalidKeyException: Illegal key size 的解决

Java实现AES加密，抛出异常如下：java.security.InvalidKeyException: Illegal key size

**原因：**

Illegal key size or default parameters 是指密钥长度受限制，

Java运行时环境读到的是受限的policy文件。

policy文件位于${java\_home}/jre/lib/security 目录下。

这种限制是因为美国对软件出口的控制。

**解决办法：**

去除该限制只需下载 Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files，

覆盖上述目录下的对应jar文件(local\_policy.jar, US\_export\_policy.jar)即可。

下载地址：

JDK8 http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html

下载包的readme.txt 有安装说明，即替换

${jdk\_home}/jre/lib/security

${jre\_home}/lib/security

目录下的 local\_policy.jar 和 US\_export\_policy.jar 文件。

### 5.Spring整合Hibernate时，通过aop配置事务异常：UnsatisfiedDependencyException。

在spring3以上版本中使用spring的依赖注入(注解或者xml方式)和aop功能时，发现了一个问题，如果不设置<aop:aspectj-autoproxy proxy-target-class="true"/>那么在获取bean时一直报：(无论通过name还是type都获取不到bean)

org.springframework.beans.factory.NoSuchBeanDefinitionException: No unique bean of type

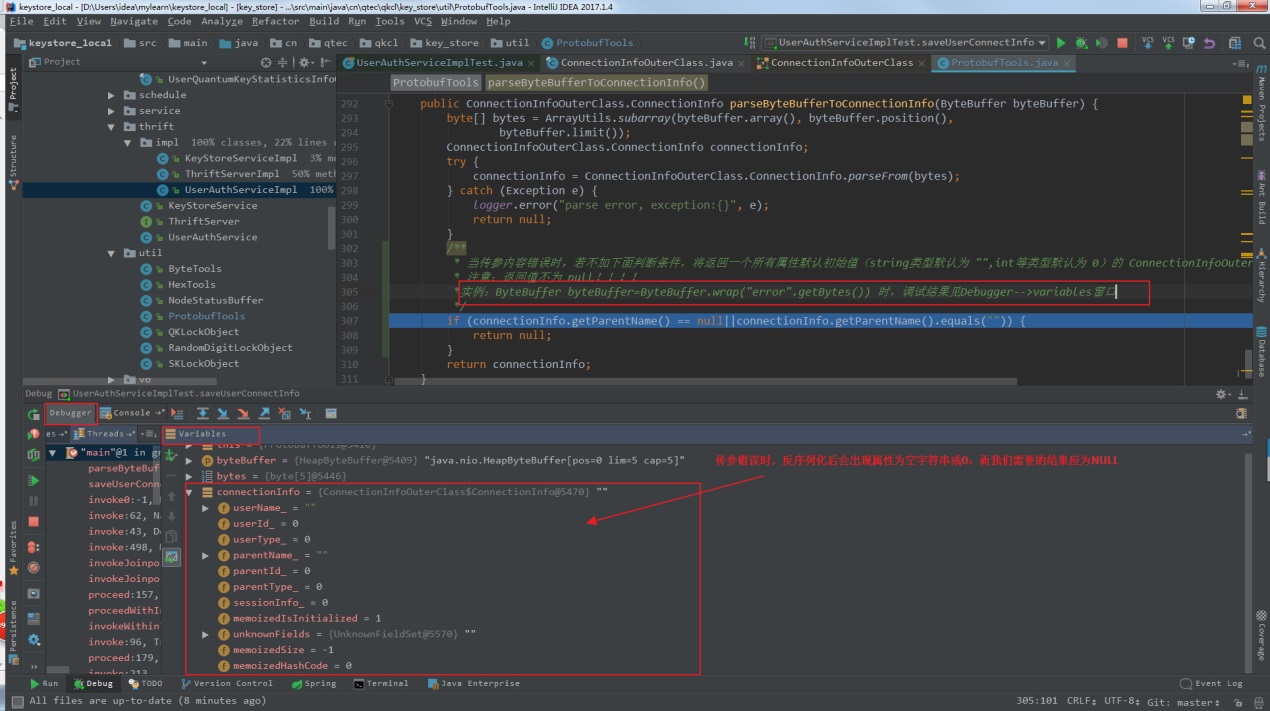
或者

org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'test.dao.BaseDaoImplTest': Unsatisfied dependency expressed through field 'accountDao'; nested exception is org.springframework.beans.factory.BeanNotOfRequiredTypeException: Bean named 'accountDaoImpl' is expected to be of type 'com.jump.account.base.dao. impl.AccountDaoImpl' but was actually of type 'com.sun.proxy.$Proxy24'

proxy-target-class="true" 与proxy-target-class="false"的区别：

proxy-target-class属性值决定是基于接口的还是基于类的代理被创建。如果proxy-target-class 属性值被设置为true，那么基于类的代理将起作用（这时需要cglib库）。如果proxy-target-class属值被设置为false或者这个属性被省略，那么标准的JDK 基于接口的代理

6、protobuf反序列化返回的对象属性为空字符串（””）或0的处理：



6．Log4j日志打印时报错

原因是因为添加了<scope>test</scope>导致。

<dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-log4j12</artifactId>  
 <version>1.7.25</version>  
 <!--<scope>test</scope>-->  
</dependency>

标签<scope>详解：

|  |  |
| --- | --- |
| **Compile** | 默认的scope，表示dependency 都可以在生命周期中使用。而且，这些dependencies 会传递到依赖的项目中。适用于所有阶段，会随着项目一起发布 |
| **Provided** | 跟compile相似，但是表明了dependency由JDK或者容器提供，例如Servlet AP和一些Java EE APIs。这个scope只能作用在编译和测试时，同时没有传递性。 |
| **Runtime** | 表示dependency不作用在编译时，但会作用在运行和测试时，如JDBC驱动，适用运行和测试阶段。 |
| **Test** | 表示dependency作用在测试时，不作用在运行时。 只在测试时使用，用于编译和运行测试代码。不会随项目发布。 |
| **System** | 跟provided 相似，但是在系统中要以外部JAR包的形式提供，maven不会在repository查找它。 |

### 6.mongo数据库sort查询错误

Error: error: {

"ok" : 0,

"errmsg" : "Executor error during find command: OperationFailed: Sort operation used more than the maximum 33554432 bytes of RAM. Add an index, or specify a smaller limit.",

"code" : 96,

"codeName" : "OperationFailed"

}、

KMIP

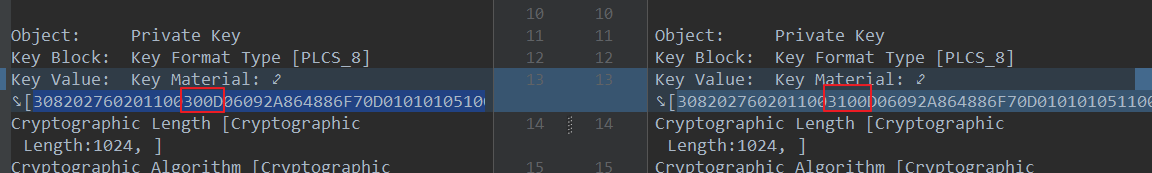
### 7、org.junit.ComparisonFailure 解码KeyMaterial时设置keyMaterialByteString属性时数据不一致？

修改前，红色字体行：

private void decodeKeyMaterial(List<Byte> list, KeyBlock keyBlock) throws KMIPUnexpectedTypeException, KMIPUnexpectedTagException, UnsupportedEncodingException, KMIPPaddingExpectedException {  
 checkTag(EnumTag.*KeyMaterial*, list);  
 int type = decodeType(list.get(3));  
 KeyMaterial keyMaterial;  
 int length = decodeLength(list);  
 switch (type) {  
 case EnumType.*ByteString*:  
 keyMaterial= new KeyMaterial(decodeKMIPByteString(list).getValueString());  
 break;  
 case EnumType.*Structure*:  
 keyMaterial = new KeyMaterial(decodeKeyMaterialStructure(list.subList(TTL\_SIZE, TTL\_SIZE + length), keyBlock.getKeyFormatType()));  
 break;  
 default:  
 throw new KMIPUnexpectedTypeException(new EnumType(type).getKey());  
 }  
 keyBlock.getKeyValue().setKeyMaterial(keyMaterial);  
}

通过改行方式设置值时多处位置出现多余的数字1.

如下：



解决：

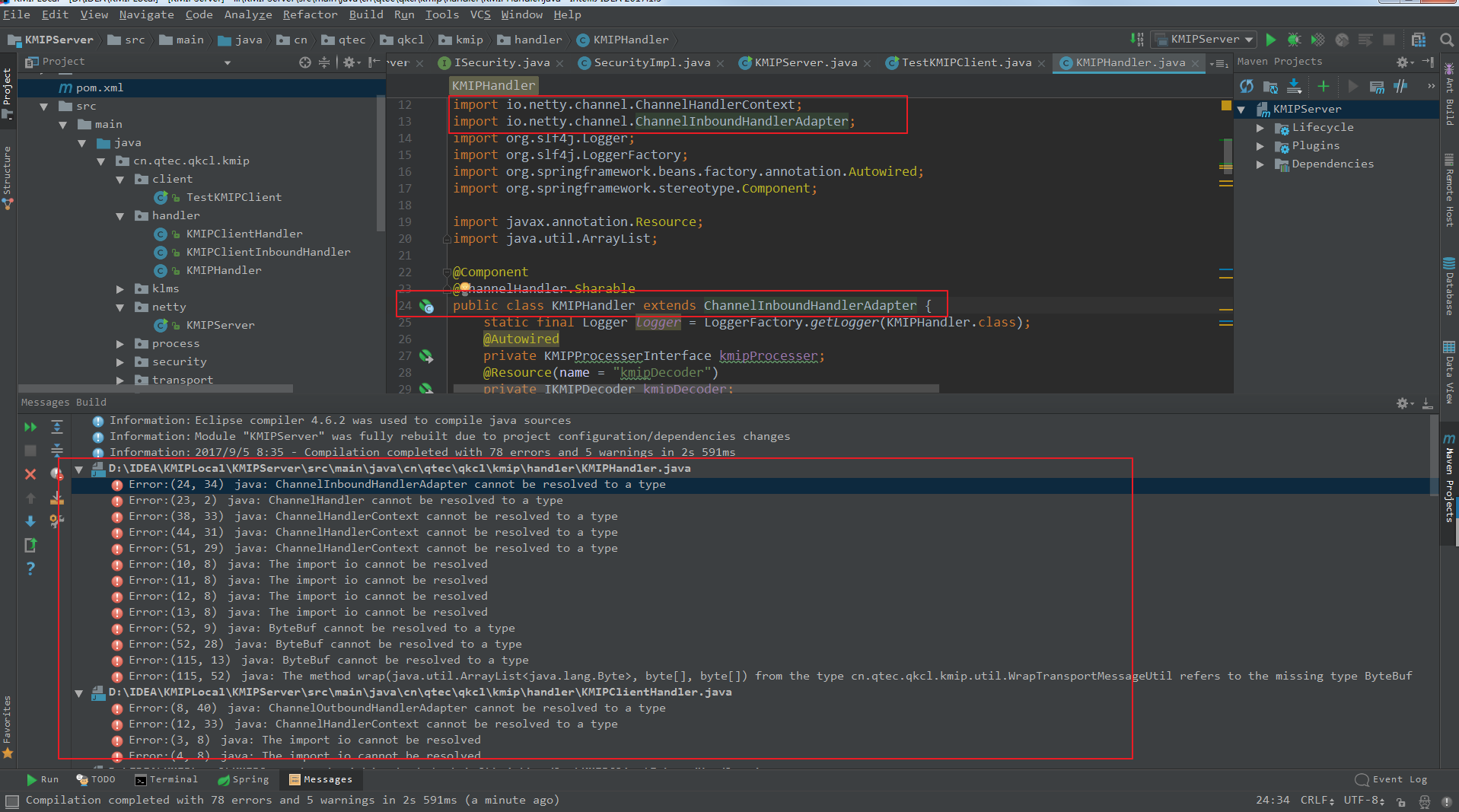
上述方式传值是通过先将KMIPByteString值转为String类型，在构造函数里使用new KMIPByteString（String str）赋值；debug发现在调用下面setValue时数据开始发生改变。

private byte[] value;

public KMIPByteString(String value) {  
 super();  
 setValue(value);  
}  
public void setValue(String value) {  
 this.value = KMIPUtils.*convertHexStringToByteArray*(value);  
}

同时发现此方式饶了一圈，后改为直接使用传参类型为KMIPByteString的构造函数，不进行string的转换后，结果恢复正常。

### 8、编译时报错：The import io cannot be resolved 或 ChannelHandler cannot be resolved to a type



Solve:

1)网上查找很多说是JDK版本问题，检查idea的JDK版本设置，均为1.8，排除此原因；

2）检查import 进来的netty jar包，pom.xml中添加的版本如下：

<dependency>  
 <groupId>io.netty</groupId>  
 <artifactId>netty-all</artifactId>  
 <version>4.1.14.Final</version>  
</dependency>

怀疑此版本问题，于是更换版本，

<dependency>  
 <groupId>io.netty</groupId>  
 <artifactId>netty-all</artifactId>  
 <version>5.0.0.Alpha1</version>  
</dependency>

再次运行KMIPServer main方法，问题解决。

### 9、Mysql死锁问题

参考博客：<http://blog.csdn.net/github_26672553/article/details/52931263>

<http://blog.csdn.net/zq602316498/article/details/49428825>

mysql> show engine innodb status;

//略

------------------------

LATEST DETECTED DEADLOCK

------------------------

-- 发生死锁的时间--

180209 13:17:32

-- 事务1的状态

\*\*\* (1) TRANSACTION:

TRANSACTION 5B3E30, ACTIVE 0 sec inserting

mysql tables in use 1, locked 1

LOCK WAIT 3 lock struct(s), heap size 1248, 2 row lock(s), undo log entries 1

MySQL thread id 7882, OS thread handle 0x2b39ac6da700, query id 21595271 192.168.20.142 root update

*insert into connection\_info\_addr (local\_id, parent\_id, parent\_name, parent\_type, remote\_ip, remote\_port, user\_id, user\_name, user\_type) values (701, 701, 'node\_ak', 0, 1751034048, 39507, 901, 'client\_1', 1)*

\*\*\* (1) WAITING FOR THIS LOCK TO BE GRANTED:

-- 事务1正在等待的锁 page no 1188 n bits 160 index `user\_name\_unique`

-- 隐藏的事务1在进行delete by`user\_name\_unique`时同样持有锁，恰好是事务2等待的锁

RECORD LOCKS space id 0 page no 1188 n bits 160 index `user\_name\_unique` of table `QkPool`.`connection\_info\_addr` trx id 5B3E30 lock\_mode X locks gap before rec insert intention waiting

-- 事务2的状态

\*\*\* (2) TRANSACTION:

TRANSACTION 5B3E33, ACTIVE 0 sec inserting

mysql tables in use 1, locked 1

3 lock struct(s), heap size 1248, 2 row lock(s), undo log entries 1

MySQL thread id 7877, OS thread handle 0x2b39ac248700, query id 21595275 192.168.20.142 root update

*insert into connection\_info\_addr (local\_id, parent\_id, parent\_name, parent\_type, remote\_ip, remote\_port, user\_id, user\_name, user\_type) values (701, 701, 'node\_ak', 0, 1751034048, 39509, 902, 'client\_2', 1)*

\*\*\* (2) HOLDS THE LOCK(S):

-- 事务2持有的锁与事务1相同 page no 1188 n bits 160 index `user\_name\_unique`

RECORD LOCKS space id 0 page no 1188 n bits 160 index `user\_name\_unique` of table `QkPool`.`connection\_info\_addr` trx id 5B3E33 lock\_mode X locks gap before rec

\*\*\* (2) WAITING FOR THIS LOCK TO BE GRANTED:

--事务2正在等待的锁，等待的是**同一个表，同一个索引，同一个page上的record lock X锁**，但是内存地址不同，即不同的行上的锁：

RECORD LOCKS **space id 0 page no 1188** n bits 160 index `user\_name\_unique` of table `QkPool`.`connection\_info\_addr` trx id 5B3E33 lock\_mode X locks gap before rec insert intention waiting

-- 事务回滚--

\*\*\* WE ROLL BACK TRANSACTION (2)

--------------------------------------------

当一个事务持有了其他事务需要的锁，同时又想获得其他事务持有的锁时，等待关系图上就会产生循环，Innodb不会显示所有持有和等待的锁，但是，它显示了足够的信息来帮你确定，查询操作正在使用哪些索引，这对于你确定能否避免死锁有极大的价值。

-----------------------------------------------

解决方法：

尽可能让所有的数据检索都通过索引来完成，从而避免Innodb因为无法通过索引键加锁而升级为表级锁定；

此处将原来通过`user\_name\_unique`进行的delete操作改为通过索引`id`进行delete，问题成功解决；

代码参考如下：

*//优化后的代码*ConnectionInfo dbConnection = **connectionInfoDao**.getByUserNameAndParentName(connectionInfo.getUserName(), connectionInfo.getParentName());  
**if** (dbConnection != **null**) {  
 **boolean** isDelete = **connectionInfoDao**.deleteById(dbConnection.getId());  
 ***logger***.info(**"Delete the old connectionInfo: {}"**, isDelete);  
}  
*//出现mysql死锁时的delete代码  
//connectionInfoDao.deleteByUserNameAndParentName(connectionInfo.getUserName(), connectionInfo.getParentName());***return connectionInfoDao**.insert(conn);