JAVA基础进阶

[1. 面向对象应用进阶 1-3](#_Toc17995394)

[1.1. JVM概述及内存结构 1-3](#_Toc17995395)

[1.2. JVM内存中的两大对象 1-3](#_Toc17995396)

[1.2.1. 字节码对象 1-4](#_Toc17995397)

[1.2.2. 类的实例对象 1-5](#_Toc17995398)

[1.3. JAVA对象的四大特性 1-6](#_Toc17995399)

[1.3.1. 核心特性应用 1-7](#_Toc17995400)

[1.3.2. 扩展特性应用 1-9](#_Toc17995401)

[1.4. JAVA中的两大抽象类型 1-11](#_Toc17995402)

[1.4.1. 如何理解接口? 1-11](#_Toc17995403)

[1.4.2. 如何理解抽象类? 1-12](#_Toc17995404)

[2. 新特性应用加强 2-14](#_Toc17995405)

[2.1. 泛型基础加强 2-14](#_Toc17995406)

[2.1.1. 如何理解泛型? 2-14](#_Toc17995407)

[2.1.2. 为何使用泛型? 2-14](#_Toc17995408)

[2.1.3. 泛型的应用类型？(重点) 2-14](#_Toc17995409)

[2.1.4. 泛型的通配符"？" 2-18](#_Toc17995410)

[2.1.5. 泛型的上下界问题？(了解) 2-18](#_Toc17995411)

[2.1.6. 泛型类型擦除?(了解) 2-19](#_Toc17995412)

[2.2. 序列化基础加强 2-20](#_Toc17995413)

[2.2.1. 何为对象序列化&反序列化？ 2-20](#_Toc17995414)

[2.2.2. 序列化的应用场景？ 2-20](#_Toc17995415)

[2.2.3. 对象的序列化与反序列化实现(重点)？ 2-21](#_Toc17995416)

[2.2.4. 序列化存在安全问题如何解决(了解)？ 2-23](#_Toc17995417)

[2.2.5. 序列化的粒度如何控制? 2-26](#_Toc17995418)

[2.2.6. 序列化的性能问题及如何优化(重点)？ 2-28](#_Toc17995419)

[2.3. 枚举应用基础加强 2-33](#_Toc17995420)

[2.3.1. 如何理解枚举 2-33](#_Toc17995421)

[2.3.2. 枚举应用场景 2-33](#_Toc17995422)

[2.3.3. 枚举的定义 2-34](#_Toc17995423)

[2.3.4. 枚举的应用？ 2-34](#_Toc17995424)

[2.4. 注解应用基础加强 2-36](#_Toc17995425)

[2.4.1. 如何理解注解(Annotation)？ 2-36](#_Toc17995426)

[2.4.2. 注解(Annotation)应用场景？ 2-36](#_Toc17995427)

[2.4.3. 注解(Annotation)的定义及应用？ 2-36](#_Toc17995428)

[2.4.4. 注解应用案例分析实现？ 2-38](#_Toc17995429)

[3. 核心API应用进阶 3-40](#_Toc17995430)

[3.1. 反射应用基础加强 3-40](#_Toc17995431)

[3.1.1. 如何理解反射？ 3-40](#_Toc17995432)

[3.1.2. 反射的应用场景？ 3-40](#_Toc17995433)

[3.1.3. 反射核心API及应用加强 3-41](#_Toc17995434)

[4. 综合案例进阶（扩展） 4-42](#_Toc17995435)

[4.1. 手写Spring简易工厂 4-42](#_Toc17995436)

[4.1.1. 需求分析 4-42](#_Toc17995437)

[4.1.2. 业务设计 4-43](#_Toc17995438)

[4.1.3. 代码实现 4-43](#_Toc17995439)

[4.2. 重构Spring简易工厂 4-43](#_Toc17995440)

[5. 总结 5-44](#_Toc17995441)

[5.1. 重难点分析 5-44](#_Toc17995442)

[5.2. 常见FAQ 5-44](#_Toc17995443)

[5.2.1. 理论分析 5-44](#_Toc17995444)

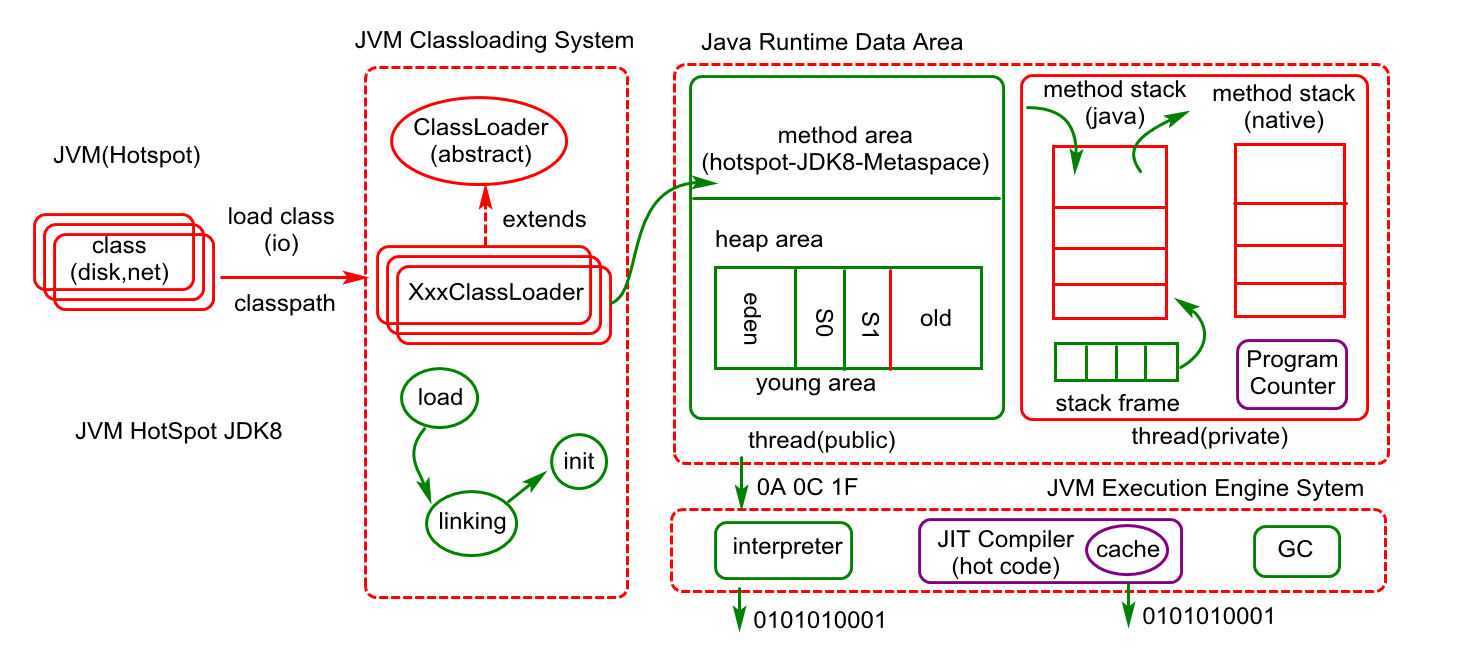
[5.2.2. 实践分析 5-49](#_Toc17995445)

[5.3. 作业 5-52](#_Toc17995446)

# 面向对象应用进阶

## JVM概述及内存结构

JVM (JAVA 虚拟机),定义了一套编译,加载,解释执行JAVA代码的规范,基于这套规范市场上不同产品实现,例如Hotspot,JRockit,J9等.其简易内存体系结构如下:



FAQ?

1. ClassLoader作用是什么?(负责将类从磁盘或网络加载内存)
2. ClassLoader 可以自己定义吗?(可以,参考Tomcat,MyBatis,Spring..)
3. JVM 中的方法区如何理解?(逻辑上的一种定义,不同JVM有不同实现)
4. HotSpot JDK8虚拟机在创建对象时,所有的对象都会分配在堆中吗?
5. JVM 常见运行参数配置(-XX:+TraceClassLoading,…)

## JVM内存中的两大对象

在java中可以将对象分为两大体系:字节码对象和实例对象

说明:字节码对象是一种特殊的实例对象.

### 字节码对象

每个类在加载(将类读到内存)时都会创建一个字节码对象,其类型为Class类型,且这个对象在一个JVM内存中是唯一的.此对象中存储的是类的结构信息(元数据信息).

字节码对象的获取方式?(常用方式有三种)

1. 类名.class
2. Class.forName(“包名.类名”)
3. 类的实例对象.getClass();

说明:字节码对象是获取类结构信息的入口，同时也是应用反射技术的起点。

代码分析:演示类的字节码对象获取

**package** com.java.oop;

//呈现类加载过程(通过配置JVM参数实现)

//-XX:+TraceClassLoading

**public** **class** TestClassObject01 {

**static** **public** **void** main(String[] args)**throws** Exception {

Class<Object> c1=Object.**class**;

Class<?> c2=Class.*forName*("java.lang.Object");

System.***out***.println(c1==c2);

Class<?> c3=**new** Object().getClass();

System.***out***.println(c2==c3);

}

}

代码分析:演示JAVA中的类加载器对象

**package** com.java.oop;

**public** **class** TestClassObject02 {

**public** **static** **void** main(String[] args) {

//AppClassLoader (负责加载我们自己写的类)

ClassLoader loader01=

ClassLoader.*getSystemClassLoader*();

System.***out***.println(loader01);

//ExtClassLoader(负责加载jdk/lib/ext/\*.jar)

ClassLoader loader02=loader01.getParent();

System.***out***.println(loader02);

//BootstrapClassLoader(负责加载jdk/lib/rt.jar)

ClassLoader loader03=loader02.getParent();

System.***out***.println(loader03);

}

}

课堂练习:

1. 编写类,然后验证类的字节码对象的唯一性.
2. 编写类,并验证类加载时是否执行静态代码块（是否执行取决于加载方式）
3. 编写对象工厂,通过工厂基于类的字节码对象创建类的实例对象.

### 类的实例对象

如何理解类的实例对象(类的对象)

1. 客观事物在内存中的呈现(堆或栈)
2. 类的实例对象在同一个JVM内存中可以有多份.

Java中对象的创建方式:

1. 通过new关键字创建(前提条件是编译时即可确定此类的存在)
2. 通过反射创建(首先要先获取字节码对象:Class.forName(pkgClass))

Java中对象的引用方式?

1. 强引用(通过对象类型的变量直接引用,例如Object o1=new Object())
2. 弱引用(WeakReference<T>) 只要有 GC,弱引用引用的对象就会被回收。
3. 软引用(SoftReference<T>)一般是FULL GC,软引用引用的对象会被回收。
4. 虚引用(PhantomReference<T>) 当对象被回收时,可以得到通知。

案例：参考课堂案例。

Java中对象的作用？

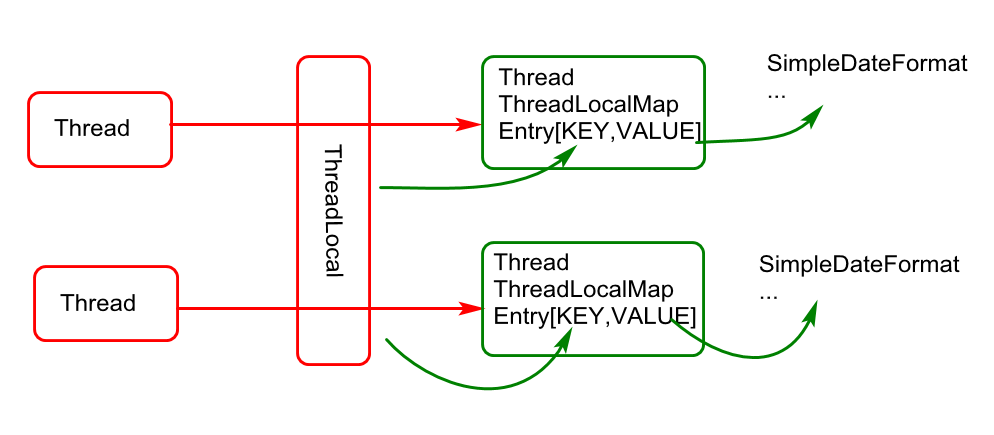
1. 存储数据(变量:类变量,实例变量,参数变量,局部变量)
2. Pojo (普通的java对象:属性,set/get)
3. Vo (值对象(Value Object):一种特殊的pojo)
4. 执行业务逻辑(方法):各司其职,各尽所能.
5. Controller
6. Service
7. Dao

建议:面向对象设计时不要设计一个大而全的对象.

FAQ? 如何理解延迟加载?(懒加载,按需加载)

课堂练习:如何对类进行单例设计?(至少先写5种)

课后扩展:了解ThreadLocal对象的应用,并写一个日期工具类,应用SimpleDataFormat对象的同时保证其线程安全.ThreadLocal的简易应用原理分析如下:



总结：

1. ThreadLocal的作用？ 线程绑定
2. ThreadLocal应用场景？取消线程共享，实现线程内部单例

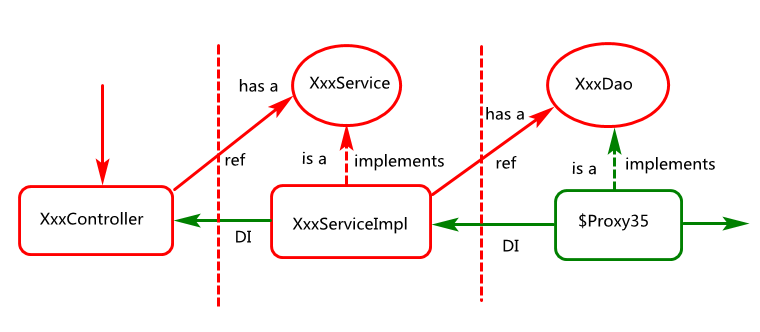
3) ThreadLocal常用方法？get()/set(obj)/..

4) ThreadLocal 应用原理？（基于ThreadLocalMap实现对象的存储和获取）

## JAVA对象的四大特性

Java中对象特性可从如下两点进行理解:

1. 核心特性:封装,继承,多态
2. 扩展特性:组合,例如圆(Circle)中有圆心(Point)



### 核心特性应用

1.封装特性

1. 广义封装:一个项目有哪些系统构成,一个系统由哪些模块构成,...
2. 狭义封装:对象属性私有化,方法能公开则公开.

封装案例应用分享：

生活中的封装:

1. 广义:大到国家有多个省份,小到家庭有多少个成员.
2. 狭义:每个人都有特征(个头高,帅气,漂亮),都有行为(说话,跳舞,唱歌).

框架中的封装:

1. mybatis:(封装JDBC操作,对JDBC参数处理,结果处理做了减法设计)
2. 广义:(会话管理,连接池模块,缓存处理模块,日志模块处理,...)
3. 狭义:SqlSession对象中应该有什么(,哪些设计为私有,哪些设计为公开)
4. Spring:(封装了对象的创建,对象的管理)
5. 广义:(IOC,MVC,AOP,…)
6. 狭义:(ClassPathXmlApplicationContext对象的构成)

思考:

1. 如何实现一个权限管理系统?(对用户进行权限控制)
2. 如何实现一个商品评价系统?(商品模块,评价模块,日志模块)

2.继承特性

1. 优势:实现代码的复用,提高程序的扩展性.(案例分析:自定义ClassLoader,参考框架mybatis,spring或tomcat中间件中的实现)
2. 劣势:大范围扩展可能会导致类爆炸,会降低代码的可维护性.

继承应用案例分享：

1）生活中的继承:

1. 子承父业
2. 文化传承

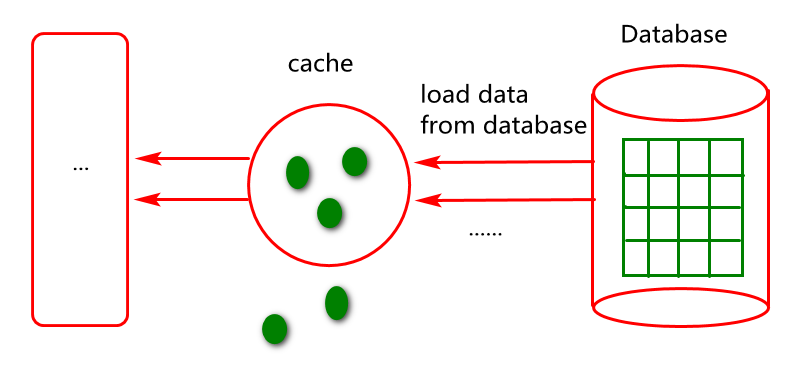
2）框架中的继承:

* 1. Mybatis(ContextMap,PersistenceException...)
  2. Spring (ManagedList,FlashMap,AnnotationAttributes....)

课堂练习:(经常在笔试中出现)

如何基于LinkedHashMap实现一个LRU算法(缓存淘汰算法)的Cahce对象?

提示:(方法:通过继承)



3.多态特性

a)编译时多态:方法的重载（名字相同，参数列表不同）

b)运行时多态:同一个行为(方法),因对象不同表现结果可能不同.

说明:此特性基于继承特性,父类引用可以指向子类对象,基于此特性可以更好实现程序之间的解耦合,提高程序可扩展性.

多态案例应用分享:

1)生活中多态:

a) 睡觉:有的人磨牙,有的人说梦话,有的人打呼噜,有的人梦游,...

b) 吃饭:有的人细嚼慢咽,有的人狼吞虎咽,...

2)框架中多态:

a)Mybatis (Executor,SimpleExecutor,CachingExecutor)

b)Spring (BeanbFactory,ClassPathXmlApplicationContext,...)

多态应用场景:

1. 方法的返回值类型能用抽象则用抽象.
2. 方法的参数类型能用抽象则用抽象.

### 扩展特性应用

组合特性可以理解为面向对象中的一个扩展特性，即可多个对象通过相互关联(协同),共同完成一个业务模块功能.

为什么要组合(相互关联)呢?

1)类设计时要遵循单一职责原则，即类设计时不要设计大而全的对象,对象职

责越多引起类变化的原因就会更多。

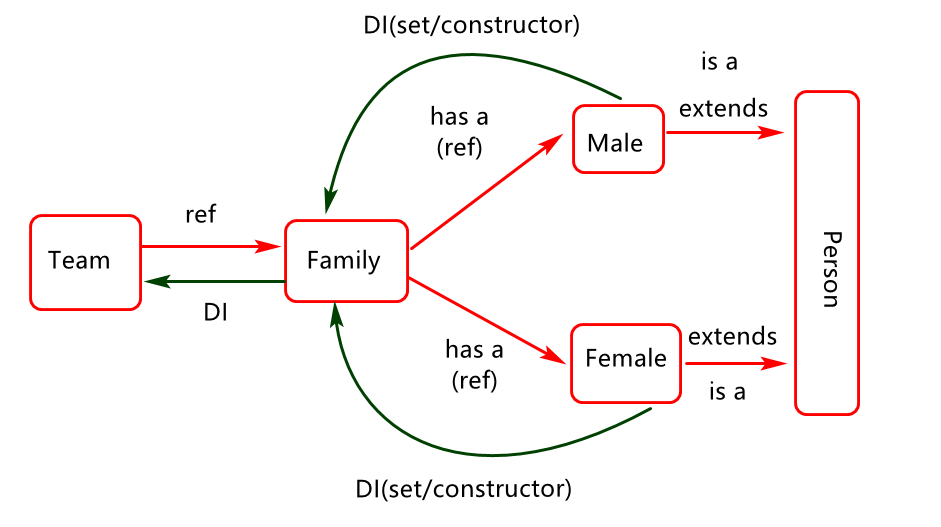
2)类设计要各司其职,各尽所能,这样的可扩展性和维护性都会比较好。

组合案例应用分享：

1)生活中的组合:

1. 陆海空,公检法
2. 游戏中CS战队

例如:



2）程序或框架中的组合:

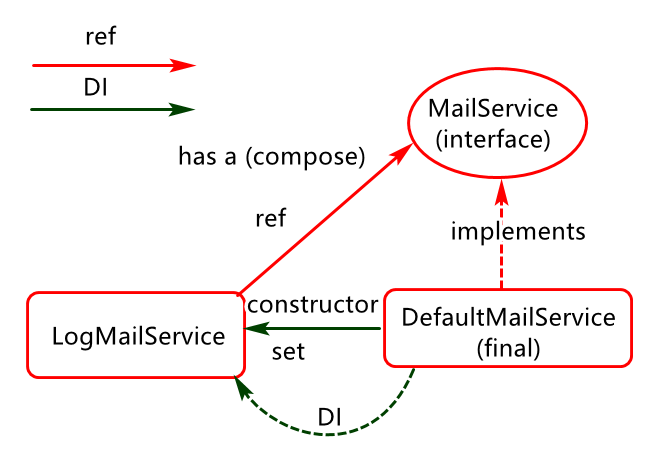
1. 如何执行一个JDBC操作?(Connection,Statement,ResultSet相互组合)
2. 如何完成一个MyBatis操作?

(SqlSessionFactory,SqlSession,Executor,.)

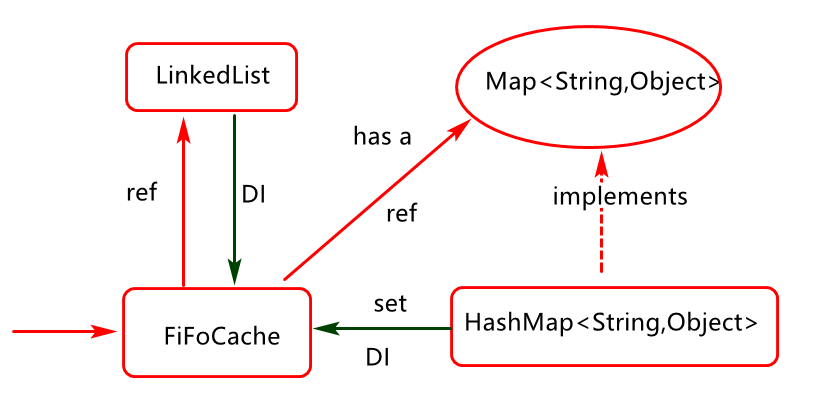
1. Spring框架可以理解为一个资源整合框架,通过资源整合(组合)完成具体的业务功能.

课堂案例:

1. 编写一个邮件发送业务,通过组合方式实现日志功能扩展.



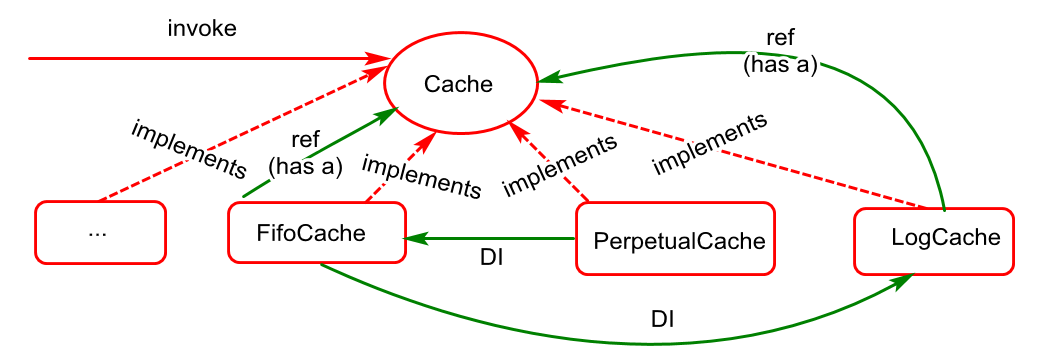
1. 编写一个基于组合方式实现的FifoCache对象(先进先出).



1. 编写一个基于组合方式实现的LruCache对象(作业).

课后作业:定义系列Cache类型对象

1. 定义Cache接口
2. 定义基于Cache接口的实现类,例如FIFOCache, PerpetualCache等
3. 基于组合方式实现对象功能增强,参考如下图的设计



## JAVA中的两大抽象类型

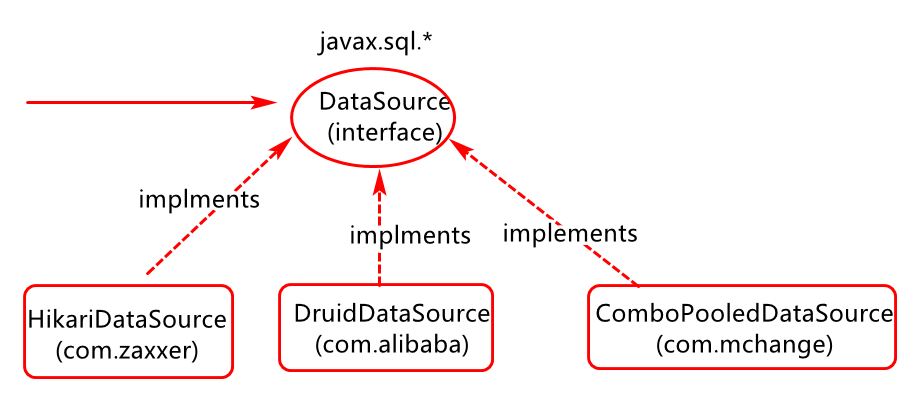
Java中提供了两大抽象类型

1. 接口(interface)
2. 抽象类(abstract)

### 如何理解接口?

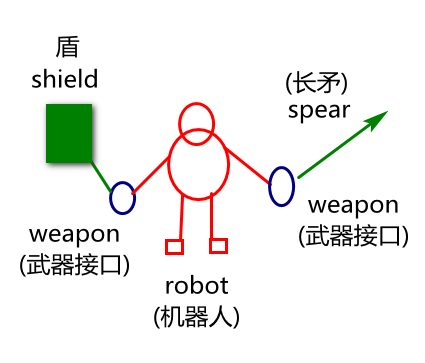
接口:定义规范,标准.(例如javax.sql.DataSource)

1. 解耦(对象之间存在耦合时尽量耦合于接口):解耦并不是没有耦合
2. 扩展(一个接口可以有很多不同实现类,例如List)



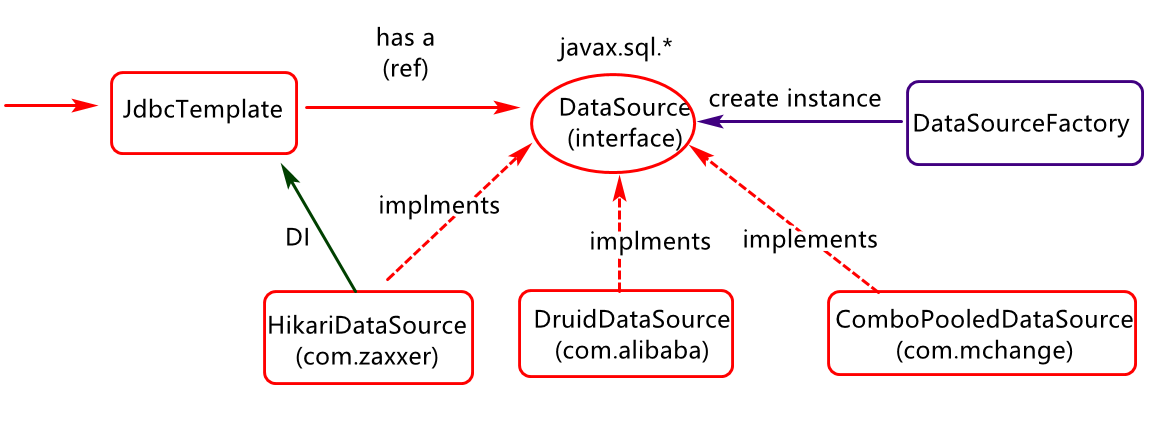
接口案例应用分享：

1. 生活中的接口:usb接口,汽车引擎接口,...



1. 程序中的接口:
2. JDK:javax.sql.DataSource
3. MyBatis:SqlSessionFactory,SqlSession,Executor,...
4. Spring:BeanFactory,ApplicationContext

案例实现:基于JdbcTemplate实现数据访问操作



其中:

1. DataSourceFactory 用于创建连接池(DataSource)对象
2. JdbcTemplate 耦合于DataSouce接口

扩展:JDK8接口中可以定义静态方法以及默认(default)方法.

### 如何理解抽象类?

抽象类:(对标准的部分共性做实现,特性交给子类实现)

1. 定义模板方法(封装某个操作的具体步骤-业务共性),特性交给子类实现.
2. 不允许外界直接构建对象(当然此类中可以不存在抽象方法).

抽象类案例应用分享

1. 生活中的抽象类:例如一些拼图



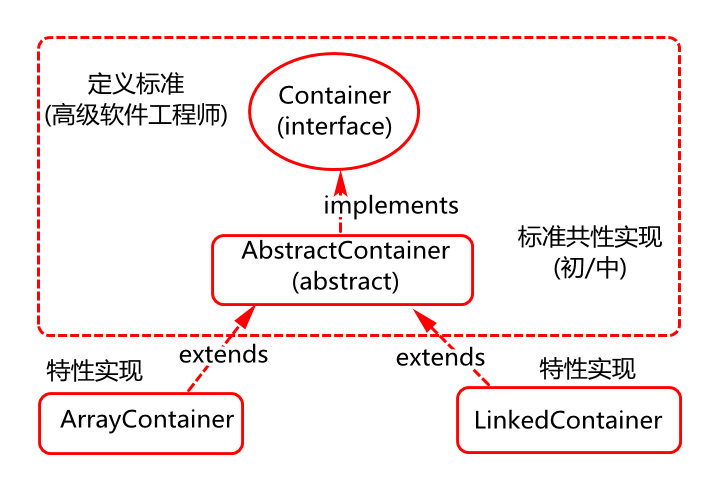
2)程序中的抽象类:

a)JDK (AbstractList,ClassLoader...)

b)MyBatis(BaseExecutor,...)

c)Spring(AbstractApplicationContext,....)

案例分析:



# 新特性应用加强

## 泛型基础加强

### 如何理解泛型?

1. 参数化的类型,数是JDK1.5的新特性。（定义泛型时使用参数可以简单理解为形参）,例如JDK中的List<E>,Map<K,V>接口中泛型应用.
2. 编译时的一种类型，此类型仅仅在编译阶段有效，运行时无效.例如List<String>在运行时String会被擦除,最终系统会认为都是Object.

### 为何使用泛型?

泛型是进行类型设计或方法定义时的一种约束规范,基于此规范可以:

1.提高编程时灵活性(有点抽象,后续结合实例理解)。

2.提高程序运行时的性能。（在编译阶段解决一些运行时需要关注的问题，例如强转）

说明:泛型应用相对比较简单,难点在泛型类或方法的设计上,通过这样的设计如何对现有类进行”减法设计”,提高类或方法的通用性.

### 泛型的应用类型？(重点)

泛型定义时常用方式有三种(可参考List<E>,Map<K,V)等接口定义):

1.泛型类： class 类名<泛型,…>{}

2.泛型接口: interface 接口名<泛型,…>{}

3.泛型方法: 访问修饰符 <泛型> 方法返回值类型 方法名(形参){}

案例分析及实现:

案例1:单泛型参数应用

接口泛型定义:(用于约束类中方法参数和方法返回值类型)

**package** com.cy.java.generic;

**interface** Container<T>{//泛型接口

**void** add(T t);

T get(**int** i);

**int** size();

}

类泛型定义:

定义一个抽象泛型类:

**abstract** **class** AbsContainer<T> **implements** Container<T>{//泛型类

**protected** **int** size;

**public** **int** size() {

**return** size;

}

}

定义一具体泛型类:

**class** ArrayContainer **extends** AbsContainer<Integer>{

**private** Object[] array;

**public** ArrayContainer() {

**this**.array=**new** Object[16];

}

@Override

**public** **void** add(Integer t) {}

@Override

**Public** Integer get(**int** i) {

**return** **null**;

}

}

泛型应用测试类:

**public** **class** TestGeneric01 {

**public** **static** **void** main(String[] args) {

//泛型只能应用对象类型

ArrayContainer c=

**new** ArrayContainer();

c.add(100);

c.size();

}

}

说明：泛型应用时相当于实参传给形参，但是实参必须为对象类型。

案例2:多泛型参数应用

接口泛型定义:

定义接口时指定泛型,用于约束接口方法参数类型以及方法返回值类型,这里无须关心此类要做什么,重点先了解语法.

**interface** Task<Param,Result>{//思考map中的泛型Map<K,V>

/\*\*

\* 此方法用于执行任务

\* **@param** arg 其类型由泛型参数Param决定

\* **@return** 其类型由泛型参数result决定

\*/

Result execute(Param arg1);

}

接口泛型应用

**class** IntegerConvertTask **implements** Task<String,Integer>{

@Override

**public** Integer execute(String arg) {

**return** Integer.*parseInt*(arg);

}

}

泛型应用测试

**public** **class** TestGeneric02 {

**public** **static** **void** main(String[] args) {

IntegerConvertTask task=**new** IntegerConvertTask();

Integer result=task.execute("100");

System.***out***.println(result);

}

}

课堂练习:

1. 自己写一个线程安全的ArrayList(至少保证add和get方法的线程安全)
2. 自己写一个线程安全的HashMap<至少保证put和get方法的线程安全)

方法泛型应用:

实例泛型方法应用分析:

定义类实现如上接口:关注泛型方法的语法

**class** ObjectFactory{

//泛型方法

**public** <T>T newInstance(Class<T> cls)

**throws** Exception{

**return** cls.newInstance();

}

/\*\*判定一个对象是否是Collection类型\*/

**public** <T>**boolean** isCollection(Class<T> cls) {

**return** Collection.**class**.isAssignableFrom(cls);

}

}

静态泛型方法应用分析:

**class** ContainerUtils{

//泛型方法

//1)静态方法假如有泛型肯定是泛型方法

//2)泛型类和泛型接口不作用于静态方法

//3)泛型方法一定是静态方法吗?不是

**public** **static** <T>**void** sort(List<T> list) {}

}

说明:框架中泛型方法参数应用:

1. MyBatis中的SqlSession接口对象
2. Spring 中的BeanFactory接口对象

总结:

1. 泛型类和泛型接口用于约束类或接口中实例方法参数类型,返回值类型.
2. 泛型类或泛型接口中实际泛型类型可以在定义子类或构建其对象时传入.
3. 泛型方法用于约束本方法(实例方法或静态方法)的参数类型或返回值类型.
4. 泛型类上的泛型不能约束类中静态方法的泛型类型.

课堂练习:定义一个对象工厂,负责基于class类型构建类的实例对象

a)类名为ObjectFactory

b)方法设计static <T>T newInstance (Class<T> cls)

### 泛型的通配符"？"

通配符一般可以理解为一种通用的类型,在这里的通配符泛指一种不确定性类型.

1.泛型应用时有一个特殊符号”?”,可以代表一种任意参数类型(实参类型)。

2.通配符泛型只能应用于变量的定义。例如:Class<?> c1;

例如:

/\*\*

\* 泛型通配符"?"的应用

\* 说明:"?"代表一种不确定的类型,

\* 当使用一个泛型类时假如其类型不确定可以使用"?"替代

\*/

**public** **class** TestGeneric05 {

**public** **static** **void** main(String[] args)**throws** Exception {

Class<Object> c1=Object.**class**; //类对象

//System.out.println(c1.toString());

//"?"为泛型应用的一个通配符

//当泛型应用时,无法判定具体类型时,使用"?"替代

//此案例在编译阶段根本无法确定字符串中代理的类型具体为何种类型.

Class<?> c2=Class.*forName*("java.lang.Object");

//Class<Log> c3=Class.forName("java.lang.Object"); 错误

System.***out***.println(c1==c2);

}

}

说明:"?"只能应用在泛型变量类型的定义

### 泛型的上下界问题？(了解)

泛型在应用时通常要指定对象的上届和下届,我们称这种形式叫限定通配符，其实现方式如下:

1.指定泛型下界：<? super 类型>

2.指定泛型上界：<? extends 类型>

例如：分析对错

List<Object> list1=new ArrayList<String>(); 错误

List<String> list2=new ArrayList<Object>(); 错误

List<? extends CharSequence> list1=new ArrayList<String>(); 正确

List<? super Integer> list2=new ArrayList<Number>();正确

说明：这种上下界一般会用于方法参数变量定义，方法返回值类型定义。

案例实现:类中方法定义

**class** PrintUtil{

**static** **void** doPrint(List<? **extends** CharSequence> list){//上界

System.***out***.println(list);

}

**static** **void** doPrint(Set<? **super** Integer> set){//下届

System.***out***.println(list);

}

}

### 泛型类型擦除?(了解)

泛型是编译时的一种类型，在运行时无效，运行时候都会变成Object类型。

作业:尝试基于反射向List<String> list=new ArrayList<String>()

集合中添加整数，关键代码如下：

List<String> list=**new** ArrayList<>();

list.add("A");

list.add("B");

//list.add(100);

//在运行时将100写入到list集合

//1.获取list对象的字节码对象

Class<?> cls=list.getClass();

//2.获取list字节码对象中的add方法对象

Method method=

//cls.getDeclaredMethod("add",Object.class);

cls.getDeclaredMethod("add",**int**.**class**,Object.**class**);

//3.通过反射执行方法对象将100写入集合。

//执行list对象的method方法

//method.invoke(list, 100);

method.invoke(list, 0,100);

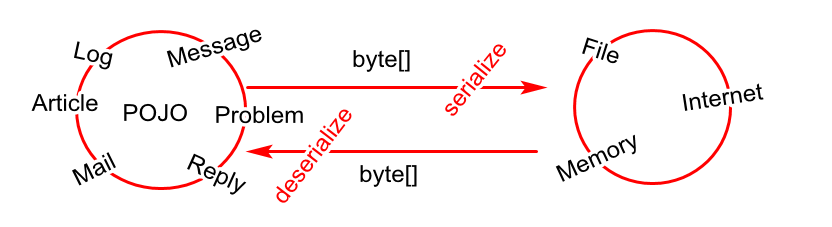
System.***out***.println(list);

## 序列化基础加强

### 何为对象序列化&反序列化？

序列化和反序列化是java中进行数据存储和数据传输的一种方式.

1. 对象序列化：将对象转换为字节的过程。
2. 对象反序列化：将字节转换为对象的过程。

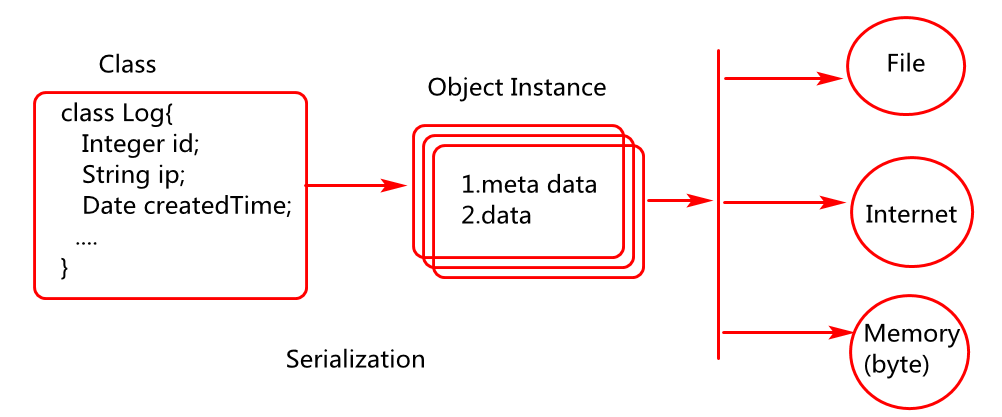


说明:在当前软件行业中有时也会将对象转换为字符串的过程理解为序列化,例如将对象转换为json格式的字符串。

### 序列化的应用场景？

序列化和反序列化通常应用在:

1. 网络通讯(C/S):以字节方式在网络中传输数据
2. 数据存储(例如文件，缓存)



说明:项目一般用于"存储数据的对象"通常会实现序列化接口.便于基于java中的序列化机制对对象进行序列化操作.

例如:

1. tomcat 中session对象的序列化(钝化)和反序列化(活化)
2. mybatis中缓存对象的序列化(对象存储到缓存)和反序列化.
3. …………

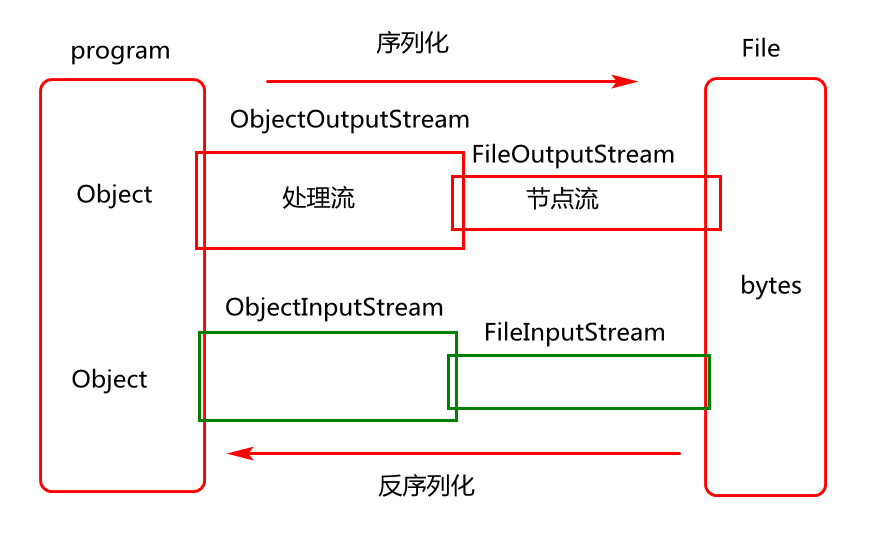
### 对象的序列化与反序列化实现(重点)？

java中如何实现对象的序列化和反序列化呢?一般要遵循如下几个步骤?

1. 对象要直接或间接实现Serializable接口。
2. 添加序列化id（为反序列化提供保障）
3. 借助对象流对象实现序列化和反序列化？

代码实现:

业务描述



定义一需要序列化的java类对象(用户行为日志对象)

**class** Message **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 7770654353954140625L;

**private** **int** id;

**private** String content;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getContent() {

**return** content;

}

**public** **void** setContent(String content) {

**this**.content = content;

}

@Override

**public** String toString() {

**return** "Message [id=" + id + ", content=" + content + "]";

}

}

定义测试类:

**public** **class** TestSerializable01 {

**public** **static** **void** main(String[] args)

**throws** Exception{

Message msg=**new** Message();

msg.setId(100);

msg.setContent("hello cgb1904");

//将如上对象序列化到文件

//1.构建流对象

ObjectOutputStream out=

**new** ObjectOutputStream(

**new** FileOutputStream("f1.dat"));

//2.将对象序列化

out.writeObject(msg);

//3.释放资源

out.close();

System.***out***.println("序列化ok");

//将文件中的内容反序列化

//1.构建流对象

ObjectInputStream in=

**new** ObjectInputStream(

**new** FileInputStream("f1.dat"));

Object obj=in.readObject();

in.close();

System.***out***.println(msg==obj);//false

System.***out***.println(msg.equals(obj));//false

System.***out***.println(obj);

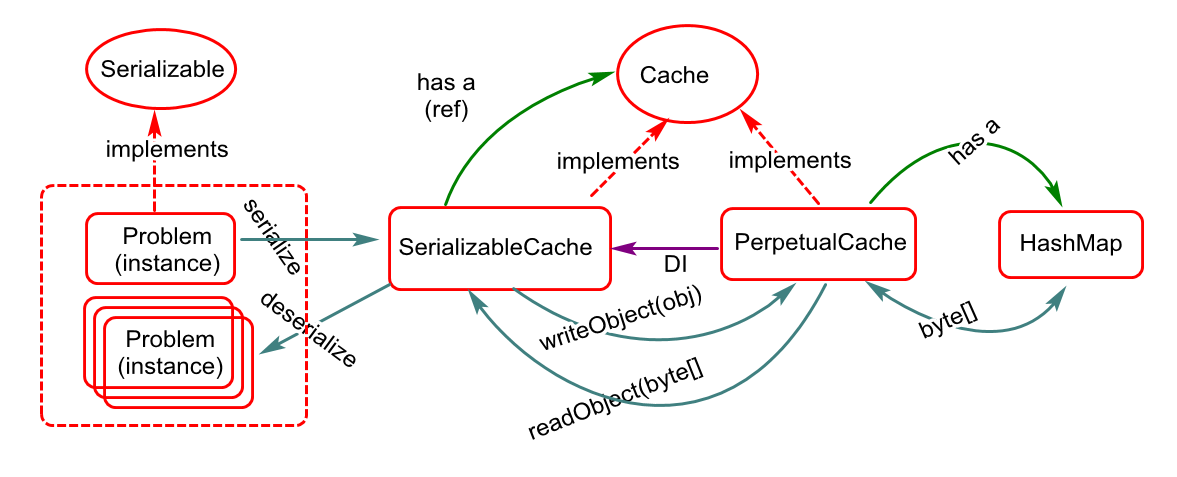
}

}

说明:

1. Serializable接口只起一个标识性的作用.
2. 建议实现序列化接口的类自动生成一个序列化id.假如没有在类中显式添加此id,不会影响对象的序列化,但可能会对反序列化有影响(对象结构发生变化).
3. 系统底层会基于类的结构信息自动生成序列化id.
4. 序列化和反序列化的顺序应该是一致的(先序列化谁,就先反序列化谁).

课堂练习:构建一个序列化cache,可以基于此cache,将对象序列化到缓存.



实现步骤分析：

1. 构建SerializableCache 类,负责实现序列化存储和获取。
2. 构建Problem类，负责对问题对象进行描述
3. 构建测试类实现对Problem对象的序列化和反序列化。

### 序列化存在安全问题如何解决(了解)？

java中的默认序列化是存在一些安全问题的,例如对象序列化以后的字节通过网络传输,有可能在网络中被截取。那如何保证数据安全呢?通常可以在对象序列化时对对象内容进行加密,对象反序列化时对内容进行解密。

具体实现过程分析：

1. 在序列化对象中添加writeObject(ObjectOutpuStream out)方法

对内容进行加密再执行序列化。

1. 在序列化对象中添加readObject(ObjectInputStream in)方法对

内容先进行反序列化然后在执行解密操作

代码实现：

问题答复对象设计:

**class** Replay **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = -5004931868757494485L;

**int** id;

String content;

//此方法会在执行ObjectOutputStream对象的writeObject时自动执行

//此方法的写法,必须这么写.

**private** **void** writeObject(ObjectOutputStream out)

**throws** IOException{

//获取加密对象(借助JDK中的Base64这个类)

Encoder encoder=Base64.*getEncoder*();

//对内容进行加密

**byte**[] contentArray=encoder.encode(content.getBytes());

content=**new** String(contentArray);

//进行默认序列化

out.defaultWriteObject();

}

//此方法会在执行ObjectInputStream的readObject方法时自动执行

//此方法可以在反序列化时执行一些解密等操作

**private** **void** readObject(ObjectInputStream in)

**throws** IOException,ClassNotFoundException{

//1.先默认反序列化(将字节读出自动赋值给属性)

in.defaultReadObject();

//2.获取解密对象

Decoder d=Base64.*getDecoder*();

//3.执行解密操作

**byte**[] contentArray=d.decode(content.getBytes());

content=**new** String(contentArray);

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getContent() {

**return** content;

}

**public** **void** setContent(String content) {

**this**.content = content;

}

@Override

**public** String toString() {

**return** "Replay [id=" + id + ", content=" + content + "]";

}

}

说明: writeObject/readObject方法:

1. 访问修饰符,返回值类型,方法名,参数应与如上代码相同(java规范中定义)
2. 两个方法会在序列化和反序列化时由系统底层通过反射调用.

测试代码:

**public** **class** TestSerializable03 {

//客户端

**static** **void** client()**throws** Exception {

Socket socket=**new** Socket("127.0.0.1", 9999);

ObjectOutputStream oos=

**new** ObjectOutputStream(socket.getOutputStream());

Replay a=**new** Replay();

a.setId(1);

a.setContent("java se");

//将对象序列化到网络

oos.writeObject(a);

oos.close();

socket.close();

}

//服务端

**static** **void** server()**throws** Exception {

//构建ServerSocket对象

ServerSocket server=**new** ServerSocket(9999);

System.***out***.println("server start ok");

//等待客户端的连接

Socket socket=server.accept();

System.***out***.println("client connection");

//获取网络中的流对象并对其进行封装(装饰)

ObjectInputStream ois=

**new** ObjectInputStream(socket.getInputStream());

//对网络中的数据进行反序列化

Object obj=ois.readObject();

System.***out***.println(obj);

//释放资源

socket.close();

server.close();

}

**public** **static** **void** main(String[] args)**throws** Exception {

**new** Thread() {

**public** **void** run() {

**try**{*server*();}**catch**(Exception e) {}

};

}.start();

**new** Thread() {

**public** **void** run() {

**try**{*client*();}**catch**(Exception e) {}

};

}.start();

}

}

### 序列化的粒度如何控制?

所谓序列化粒度一般指对象序列化时,如何控制对象属性的序列化。例如哪些序列化，哪些属性不序列化。java中的具体方案一般有两种：

方案1:不需要序列化的属性使用Transient修饰.

当少量属性不需要序列化时，使用此关键字修饰比较方便.例如 private transient Integer id;(例如arraylist,hashmap中的属性)

方案2:让序列化对象实现Externalizable接口,自己指定属性的序列化和反序列化过程, 但是要序列化的对象对应的类必须使用public修饰.

代码实现:

**public class** Article **implements** Externalizable{

**private** Integer id;//10

**private** String title;

**private** String content;

**private** String createdTime;

//序列化时调用

@Override

**public** **void** writeExternal(ObjectOutput out) **throws** IOException {

out.writeUTF(title);

}

//反序列化调用

@Override

**public** **void** readExternal(ObjectInput in) **throws** IOException, ClassNotFoundException {

title=in.readUTF();

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** **void** setTitle(String title) {

**this**.title = title;

}

**public** **void** setContent(String content) {

**this**.content = content;

}

**public** **void** setCreatedTime(String createdTime) {

**this**.createdTime = createdTime;

}

@Override

**public** String toString() {

**return** "Message [id=" + id + ", title=" + title + ", content=" + content + ", createdTime=" + createdTime

+ "]";

}

}

其中:序列化和反序列化需要在Externalizable接口方法中进行实现.

测试实现:

**public** **class** TestSerializable02 {

**public** **static** **void** main(String[] args)**throws** Exception {

//1.构建日志对象

Article a=**new** Article ();

a.setId(100);

a.setTitle("cgb");

a.setContent("cgb1903");

a.setCreatedTime("2019-05-28");

//2.将对象序列化到文件

ObjectOutputStream out=

**new** ObjectOutputStream( **new**

FileOutputStream("f2.dat"));

out.writeObject(msg);

out.close();

System.***out***.println("序列化ok");

//3.将对象从文件反序列化.

ObjectInputStream in=**new** ObjectInputStream(

**new** FileInputStream("f2.dat"));

Object msg2=in.readObject();

System.***out***.println(msg2);

in.close();

}

}

### 序列化的性能问题及如何优化(重点)？

Java中默认的序列化机制,其性能相对较差。此性能的提升，目前会借助一些第三方的框架进行实现,例如kryo ,其官方API应用网址.

(<https://github.com/EsotericSoftware/kryo>)。

* **案例一：基于Kryo实现对象对象序列化和反序列化**

第一步：添加依赖：

<dependency>

<groupId>com.esotericsoftware</groupId>

<artifactId>kryo</artifactId>

<version>5.0.0-RC4</version>

</dependency>

第二步：定义对象(频道对象)

**class** Channel **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = -5996516514113734906L;

**private** **int** id;

**private** String name;

**private** **int** state;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getState() {

**return** state;

}

**public** **void** setState(**int** state) {

**this**.state = state;

}

@Override

**public** String toString() {

**return** "Channel [id=" + id + ", name=" + name + ", state=" + state + "]";

}

}

第三步：实现对象序列化和反序列化，关键代码分析

//构建频道对象

Channel channel=**new** Channel();

channel.setId(100);

channel.setName("教育");

channel.setState(1);

//对象序列化

Kryo kryo = **new** Kryo();

kryo.register(Channel.**class**);

Output output = **new** Output(

**new** FileOutputStream("file.bin"));

kryo.writeObject(output,channel);

output.close();

//对象的反序列化

Input input = **new** Input(

**new** FileInputStream("file.bin"));

Channel object2 =

kryo.readObject(input, Channel.**class**);

System.***out***.println(object2);

input.close();

第四步：基于Kryo实现序列化和反序列化工具类

**package** com.company.java.serializable;

**import** java.io.ByteArrayInputStream;

**import** java.io.ByteArrayOutputStream;

**import** java.io.IOException;

**import** com.esotericsoftware.kryo.Kryo;

**import** com.esotericsoftware.kryo.io.Input;

**import** com.esotericsoftware.kryo.io.Output;

**public** **final** **class** KryoUtil {

**private** KryoUtil() {}

//static Kryo kryo=new Kryo();线程不安全

**private** **static** ThreadLocal<Kryo> *tl*=

**new** ThreadLocal<Kryo>() {

**protected** Kryo initialValue() {

System.***out***.println("initialValue()");

Kryo kryo =**new** Kryo();

kryo.setRegistrationRequired(**false**);

**return** kryo;

};

};

/\*\*完成对象序列化\*/

**public** **static** **byte**[] serialize(Object obj)**throws** IOException {

//1.构建Kryo对象

Kryo kryo=*tl*.get();

kryo.setRegistrationRequired(**false**);

//2.构建流对象

ByteArrayOutputStream bos=

**new** ByteArrayOutputStream();

Output output=**new** Output(bos);

//3.对象序列化

kryo.writeClassAndObject(output, obj);

//4.释放资源

output.close();

bos.close();

**return** bos.toByteArray();

}

/\*\*完成对象的反序列化\*/

**public** **static** Object deserialize(**byte**[] array) {

//1.构建Kryo对象

Kryo kryo=*tl*.get();

//2.构建输入流对象

Input input=**new** Input(

**new** ByteArrayInputStream(array));

//3.对象的反序列化

Object obj=kryo.readClassAndObject(input);

//4.释放资源

input.close();

**return** obj;

}

}

第五步：基于Kryo实现序列化Cache对象(扩展实现)：

**package** com.company.java.cache;

**import** com.company.java.exception.CacheException;

**import** com.company.java.serializable.KryoUtil;

**public** **class** KryoSerializedCache **implements** Cache{

**private** Cache cache;

**public** KryoSerializedCache(Cache cache) {

**this**.cache=cache;

}

@Override

**public** **void** putObject(Object key, Object value) {

**try** {

//1.序列化

**byte** array[]=KryoUtil.*serialize*(value);

//2.写缓存

cache.putObject(key, array);

}**catch**(Exception e) {

**throw** **new** CacheException("序列化异常");

}

}

@Override

**public** Object getObject(Object key) {

//1.读缓存

**byte**[] array=(**byte**[])cache.getObject(key);

//2.反序列化

**return** KryoUtil.*deserialize*(array);

}

@Override

**public** Object removeObject(Object key) {

**return** cache.removeObject(key);

}

**public** **static** **void** main(String[] args) {

KryoSerializedCache cache=

**new** KryoSerializedCache(**new** PerpetualCache());

Integer a=1000;

Integer b=2000;

cache.putObject("A",a);

cache.putObject("B",b);

Object o1=cache.getObject("A");

Object o2=cache.getObject("B");

System.***out***.println(a==o1);

System.***out***.println(b==o2);

PerpetualCache pCache=

**new** PerpetualCache();

pCache.putObject("C", a);

Object o3=pCache.getObject("C");

System.***out***.println(o3==a);

}

}

* **扩展案例:hessian 框架的应用(官网<http://hessian.caucho.com>)**

业务描述:基于hession完成mail对象的序列化和反序列化.

第一步：添加依赖

<dependency>

<groupId>com.caucho</groupId>

<artifactId>hessian</artifactId>

<version>4.0.60</version>

</dependency>

第二步:定义序列化和反序列化方法,可参考官方:

<http://hessian.caucho.com/doc/hessian-overview.xtp>

添加对象序列化方法定义

**private** **static** **void** serialize(Object obj) **throws** FileNotFoundException, IOException {

OutputStream os = **new** FileOutputStream("test.xml");

Hessian2Output out = **new** Hessian2Output(os);

out.writeObject(obj);

out.flush();

os.close();

}

添加对象反序列化方法定义

**private** **static** Object deserialize() **throws** FileNotFoundException, IOException {

InputStream is = **new** FileInputStream("test.xml");

Hessian2Input in = **new** Hessian2Input(is);

Object obj2 = in.readObject();

is.close();

**return** obj2;

}

第三步：定义测试方法:

**public** **static** **void** main(String[] args)**throws** Exception {

Channel obj = **new** Channel();

obj.setId(200);

obj.setName("edu");

obj.setState(**1**);

//序列化

*serialize* (obj);

//反序列化

Channel obj2 =(Channel) deserialize();

System.***out***.println(obj2);

}

## 枚举应用基础加强

### 如何理解枚举

枚举是JDK1.5以后推出的一种新的类型（特殊的类）,主要用于更加严格的约束变量类型,例如现有一个产品对象,此对象有一个性别属性,请问此属性的类型如何定义?

### 枚举应用场景

基于枚举更好限定变量的取值,例如一个变量要求有A,B,C三个值,该如何定义?此时就可以使用枚举,枚举类中一般定义的是几个固定的常量值.

案例分析:

1. JDK

**public** **enum** RetentionPolicy {

***SOURCE***,

***CLASS***,

***RUNTIME***

}

1. mybatis 中Executor对象的类型

**public** **enum** ExecutorType {

***SIMPLE***, ***REUSE***, ***BATCH***

}

1. spring

…..

### 枚举的定义

枚举定义时需要借助enum关键字进行定义，例如:

**enum** Week {

***MONDAY***, ***TUESDAY***, ***WEDNESDAY***,

***THURSDAY***, ***FRIDAY***, ***SATURDAY***, ***SUNDAY***

}

其中:Week中MONDAY, TUESDAY等都属于枚举的实例,这些实例都是在类加载时创建,可通过枚举类名直接访问,例如Week.MONDAY

说明:

1. 每个枚举类中都默认有一个无参构造函数,默认使用private修饰.
2. 枚举每个实例的创建都会初始化类中属性(可以添加),调用类中构造方法(默认无参,当然也可以添加带参数的构造函数)

### 枚举的应用？

案例1:定义一枚举类型,基于此类型约束产品Product类中的变量.

枚举类定义(性别类)

**enum** Gender{//Gender.class

***MALE***,***FEMALE***,***NONE***;

}

产品类定义

**class** Product{

/\*\*性别要求\*/

**private** Gender gender=Gender.***NONE***;

**public** **void** setGender(Gender gender) {

**this**.gender = gender;

}

}

测试类定义

**public** **class** TestEnum01 {

**public** **static** **void** main(String[] args) {

Product pro=**new** Product();

pro.setGender(Gender.***MALE***);

}

}

案例2:定义性别枚举,用于约束用户类中的性别属性

枚举类定义:类中可以定义带参数的构造方法,构建实例时为属性赋值.

**enum** Sex{

//枚举类型的对象是在类加载时创建

***MALE***("男"),***FEMALE***("女");//执行带参构造函数

**private** String name;

**private** Sex(String name){

**this**.name=name;

}

**public** String getName() {

**return** name;

}

}

会员类中枚举类应用

**class** Member{

**private** Sex sex=Sex.***MALE***;

**public** **void** setSex(Sex sex) {

**this**.sex = sex;

}

}

测试类定义

**public** **class** TestEnum02 {

**public** **static** **void** main(String[] args) {

Member m=**new** Member();

String sexStr="MALE";

//将字符串转换为枚举类型时,字符串的值需要

//与枚举类中的实例名相同(区分大小写)

//Sex sex=Sex.valueOf(sexStr);

Sex sex=Enum.*valueOf*(Sex.**class**, sexStr);

System.***out***.println(sex.getName());

m.setSex(sex);

}

}

说明:枚举类型与字符串类型之间进行转换.

## 注解应用基础加强

### 如何理解注解(Annotation)？

注解是:

1. JDK1.5推出的一种新的应用类型（特殊的class）
2. 元数据(Meta Data):一种描述性类型，[用于描述对象.例如@Override](mailto:用于描述对象.例如@Override)

个人认为:注解可以理解为一个为生活中的标签.

### 注解(Annotation)应用场景？

1. 描述类及其成员(属性，方法)：例如@Override,@Controller
2. 替换项目中xml方式对相关对象的描述,例如<bean id=”” class=””/>

注解的本质:描述元素(接口,类,注解,属性,方法,参数)

### 注解(Annotation)的定义及应用？

在实际项目注解可能由第三方定义,也可能会由我们自己定义.自己如何定义呢?

可以借助@interface关键字进行定义，例如Override注解的应用

@Target(value=METHOD)

@Retention(value=SOURCE)

public @interface Override{}

其中：

1)@Target 用于描述定义的注解能够修饰的对象。

2)@Retention 用于描述定义的注解何时有效。

案例实现:

定义注解Entity

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***TYPE***)//表示只能描述类

**@interface** Entity{}

定义注解 ID

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***FIELD***)//表示只能描述属性

**@interface** ID{}

使用注解描述类及成员

@Entity

**class** Column{

@ID

**private** Integer id;

}

说明：我们自己定义的注解，包括框架中的很多注解基本都是运行时有效。

测试代码实现:

**public** **class** TestAnnotation01 {

**public** **static** **void** main(String[] args)

**throws** Exception{

//1.判定SysLog类上是否有Entity注解

Class<?> c1=Column.**class**;

**boolean** flag = c1.isAnnotationPresent(Entity.**class**);

System.***out***.println(flag);

//2.判定SysLog对象中id属性上是否有ID注解

//2.1获取属性id

Field f1 = c1.getDeclaredField("id");

//2.2判定属性上是否有ID注解

flag=f1.isAnnotationPresent(ID.**class**);

System.***out***.println(flag);

}

}

### 注解应用案例分析实现？

实际项目中的注解:

1. 与编译器结合实用(@Override),无须我们自己实现
2. 与反射API结合使用(@RequiredLog,@Transaction,…)

案例分享-01:通过反射获取类上的注解以及注解对应的信息

定义注解:

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***TYPE***)

**@interface** Component{

/\*\*注解中的属性定义\*/

String value() **default** "";

**boolean** lazy() **default** **true**;

}

使用注解描述指定的类

@Service(value="lruCache",lazy=**false**)

**class** HashLruCache{}

编写业务测试类:获取DefaultSearchService中的注解信息

**public** **class** TestAnnotation01 {

**public** **static** **void** main(String[] args) {

//如何获取类或属性等上面注解呢?

//1.获取字节码对象(入口)

Class<?> cls=HashLruCache.**class**;

//2.获取类上的注解

Component c=

cls.getDeclaredAnnotation(Component.**class**);

//3.获取Entity注解上的内容

String value=c.value();

**boolean** lazy=c.lazy();

System.***out***.println(value);

System.***out***.println(lazy);

}

}

案例分享-02: 基于注解中定义的包名获取指定路径下的类.

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***TYPE***)

**@interface** ComponentScan{

String value();

}

定义配置类

@ComponentScan("com.company.java.annotation")

**class** AppConfig{}

测试业务实现:

**public** **class** TestAnnotation03 {

**public** **static** **void** main(String[] args) {

//1.获取AppConfig类上的@ComponentScan注解

Class<?> c= AppConfig.**class**;

ComponentScan cs=

c.getDeclaredAnnotation(ComponentScan.**class**);

//2.获取 @ComponentScan注解中value属性的值

String pkg=cs.value();

System.***out***.println(pkg);

//3.获取value属性对应包下的所有类

String pkgDir=pkg.replace(".", "/");

System.***out***.println(pkgDir);

//获取对应目录的的资源路径

URL url=

ClassLoader.*getSystemResource*(pkgDir);

System.***out***.println(url.getPath());

File file=**new** File(url.getPath());

File[] fs=file.listFiles();

**for**(File f:fs) {

System.***out***.println(f.getName());

}

}

}

# 核心API应用进阶

## 反射应用基础加强

### 如何理解反射？

反射是Java中特有的一种技术,是JAVA中自省特性的一种实现(对象运行时动态发现对象成员),可以基于此特性实现java的动态编程(例如对象创建,成员调用等).

### 反射的应用场景？

反射通常用于平台或框架编程,来实现更加通用的编程过程.例如:

1. 框架中对象的构建.
2. 框架中方法的调用.

框架中反射应用案例分享

1)对象创建

1. mybatis映射元素的resultType,resultMap等.
2. spring中的bean对象(<bean class="">,@Service,..)

2)方法调用

1. 对象set方法，get方法,...
2. spring mvc 控制层方法，..

总之："反射不能预知未来，但可驾驭未来"，通过反射可以更好构建一些编程框架，以实现通用性编程，从而达到简化代码编写。

FAQ:反射有什么缺陷?会存在一定的"性能"问题.

### 反射核心API及应用加强

反射应用的入口为"字节码对象",任意的一个类在同一个JVM内部,字节码对象是唯一的,此字节码对象会在第一次类加载时创建,用于存储类的结构信息(元数据信息).

基于字节码对象,我们可以获取如下对象:

1. Constructor (构造方法对象类型,基于此对象构建类的实例对象)
2. Field (属性对象类型)
3. Method (方法对象类型)
4. Annotation(注解对象类型)
5. …

反射案例应用分享:

基于字节码对象获取无参构造方法对象,并构建类的实例对象.

//基于类的字节码对象创建类的实例对象

**private** **static** <T>T doCreateInstance(Class<T> cls) **throws** Exception{

//1.获取类中的无参构造方法对象

Constructor<T> con=cls.getDeclaredConstructor();

//2.基于构造方法对象构建类的实例对象

con.setAccessible(**true**);//设置构造方法可访问

**return** con.newInstance();//默认调用无参构造函数

}

基于字节码对象获取带参构造方法对象,并构建类的实例对象.

@SuppressWarnings("unused")

**private** **static** <T>T doCreateInstance(

Class<T> cls,//字节码对象

Object[] args,//创建实例对象需要的实际参数

Class<?>[] paramTypes)**throws** Exception {

//1.获取构造方法对象

Constructor<T> con=

cls.getDeclaredConstructor(paramTypes);

//2.基于构造方法对象构建类的实例对象

**return** con.newInstance(args);

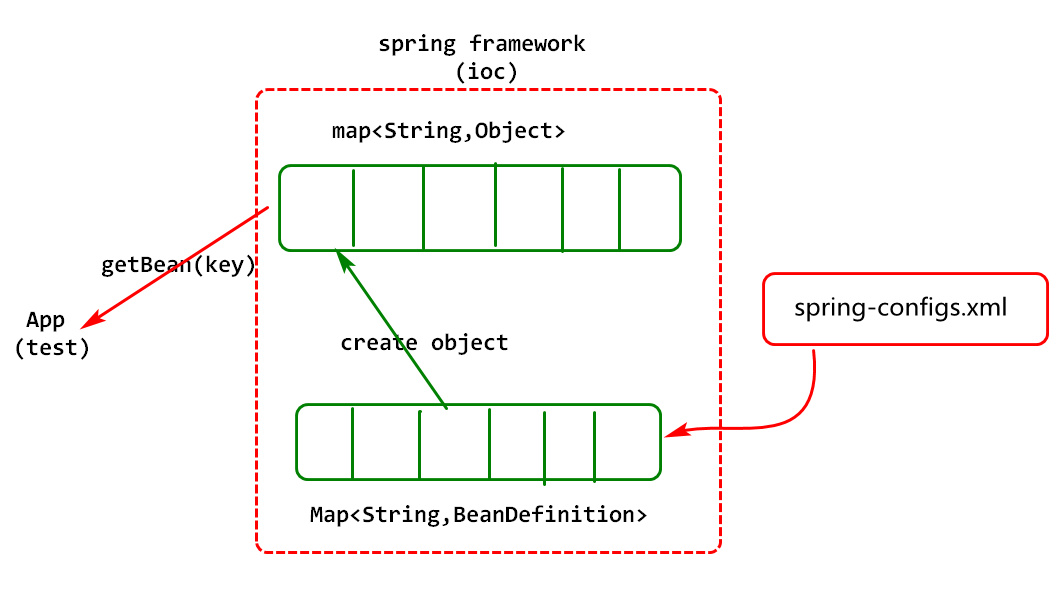
}

# 综合案例进阶（扩展）

## 手写Spring简易工厂

### 需求分析

核心业务:基于配置文件中的描述,通过一个工厂构建对象实例,并存储实例,然后外界需要时,可以让外界通过特定方法获取需要的实例对象.业务架构如下:

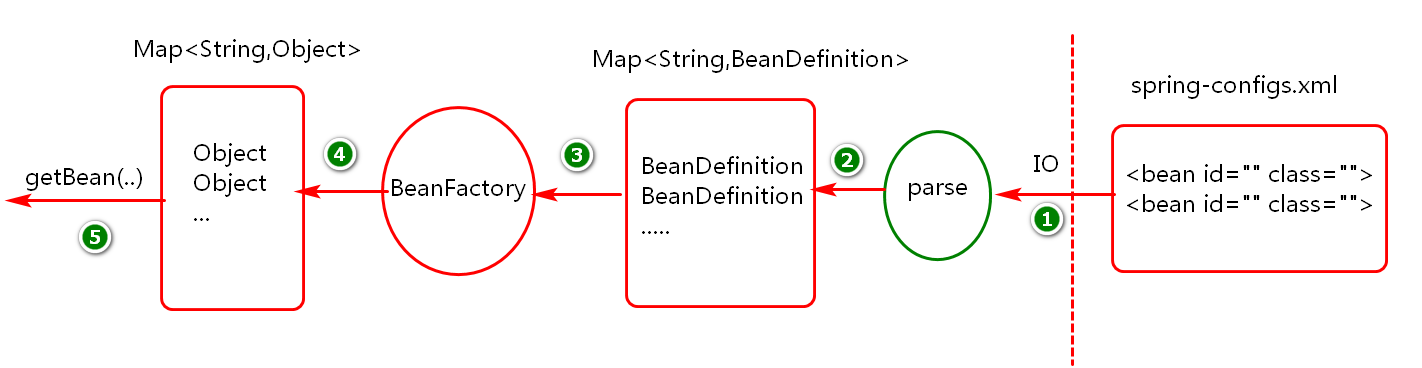


其中:

1. spring-configs.xml为配置文件
2. Map<String,BeanDefinition>用于存储配置文件信息
3. Map<String,Object>用于存储基于配置信息创建的实例.

### 业务设计

* + - 1. 业务实现过程设计



2.业务实现步骤设计

1. 创建一个配置文件spring-configs.xml,用于描述一些bean对象
2. 创建一个BeanDefinition对象,用于封装从配置文件读取的信息
3. 创建一个工厂DefaultBeanFactory对象,负责基于配置信息创建对象,存储对象,并对外提供访问对象的方式.

说明:

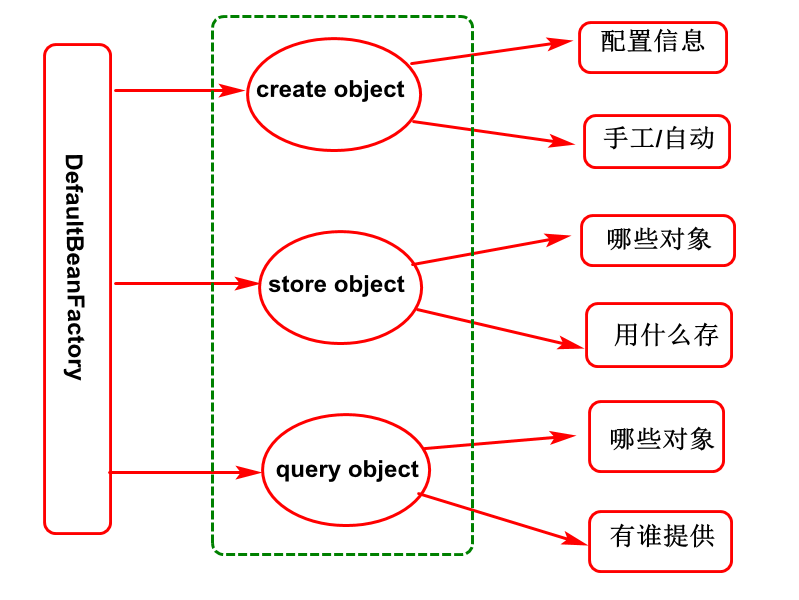
1. 工厂中提供两大map,一个存储配置信息,一个存储实例信息
2. 工厂中基于DOM解析,从xml文件中读取信息并进行封装.

### 代码实现

尝试自己实现:参考课堂案例.

## 重构Spring简易工厂

1.工厂增强分析



# 总结

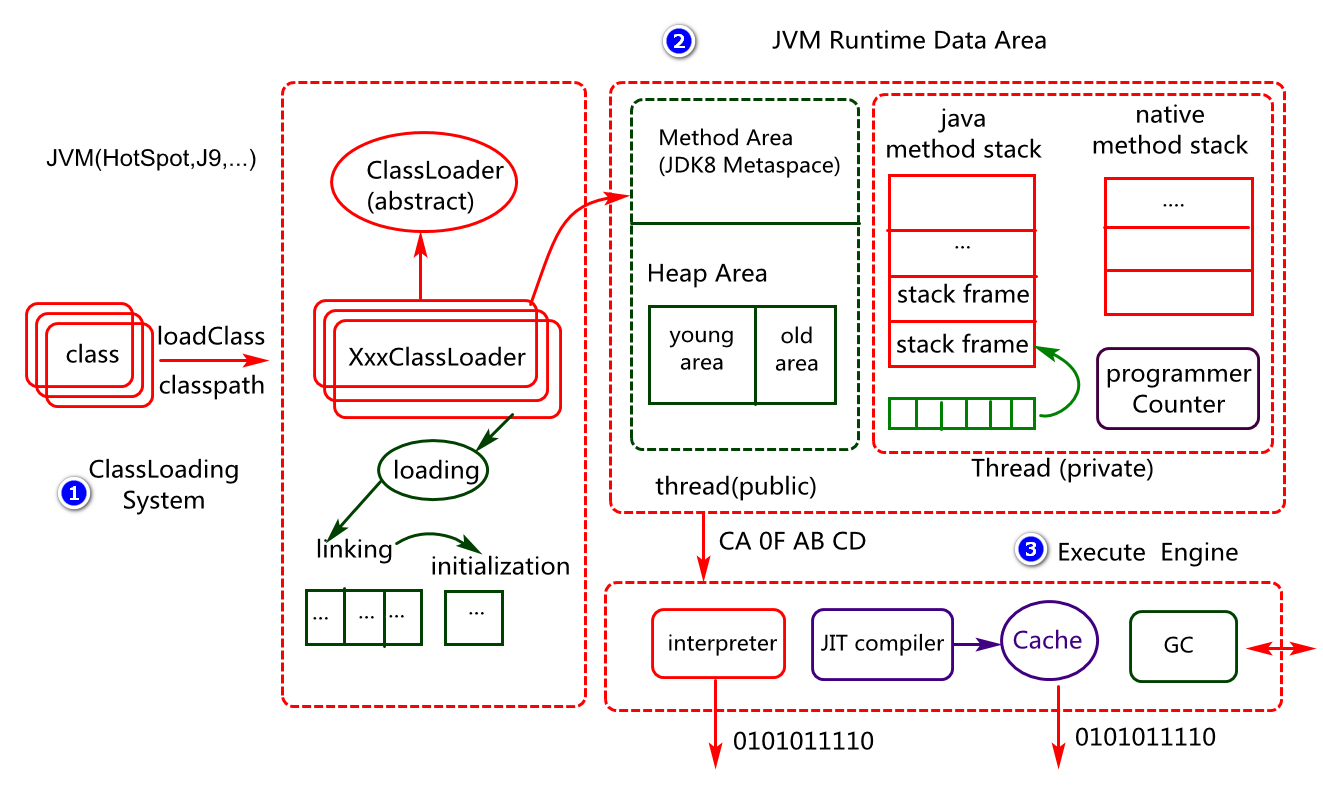
## 重难点分析

1. 内存中的两大对象:字节码对象与实例对象
2. JAVA对象四大特性(封装,继承,多态,组合):重点是宏观视角.
3. JAVA中的两大抽象:重点加强对标准的理解和认识.
4. 泛型的定义及在框架中的应用?
5. 序列化定义及应用实现分析
6. 注解的定义及应用
7. 反射定义及应用

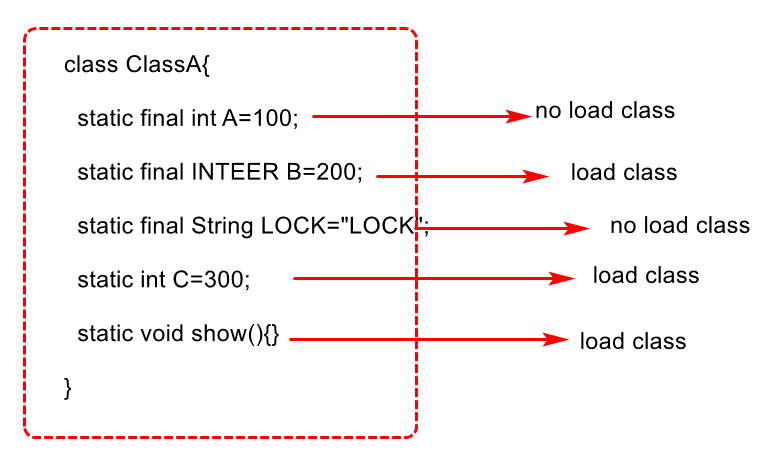
## 常见FAQ

### 理论分析

1. 说说JVM体系结构



1. 面向对象相关
   * + 1. 如何理解JVM?(一种JAVA虚拟机规范,基于此规范有不同JVM产品)
       2. 你了解有哪些JVM产品?(HotSpot,J9,JRockit)
       3. JVM 规范中有哪几部分的构成?(类加载子系统,运行时数据区,执行引擎)
       4. 说一说你了解的类加载器?(负责将类读到内存的一个对象,在Java中默认类型为ClassLoader类型,基于此类型可以有很多实现类,可以基于类特点的不同可以选择不同的类加载器,也可以自己定义类加载器)
       5. 说说JVM中默认的类加载机制?(双亲委派机制)
       6. 类加载的方式有哪些?(显式加载,隐式加载)
       7. 何时会触发隐式类加载?(构建对象,访问类成员-分情况),如下图所示



* + - 1. 类的字节码对象相关?
* 如何理解类加载?将类(字节码文件)从磁盘,从网络读到内存的过程.
* 类加载时会创建字节码对象,请问此对象中存储的是什么?(类结构信息)
* 类加载时一定会执行静态代码块吗?不一定,要看加载方式
* 如何跟踪,查看类的加载过程?(通过jvm参数-XX:+TraceClassLoading)\
* 我们可以自己写类加载器吗?(可以,直接或间接继承ClassLoader)
* 为什么我们要自己写类加载器?(默认的类加载器不能满足业务需求)
  + - 1. 类的实例对象相关?
* 类实例对象的创建方式?(new,反射)
* 类实例对象在内存中可以有多份吗?可以
* 假如不断增加内存中实例对象的个数会有什么问题?内存溢出
* 构建类的实例对象时会加载类吗?(假如已经加载则不再加载)
* 说说类实例对象的构建过程是怎样的?
* 你觉得java中的对象有什么作用?
  + - 1. 面向对象中的核心特性?
* 你理解的面向对象有哪些核心特性? 封装,继承,多态,组合.
* 你是如何理解面向对象中的封装特性的? 宏观视角和微观视角
* 你是如何理解面向对象中的继承特性的? 复用(共享)和扩展(特性)
* 你是如何理解面向对象中的组合特性的? 协作

1. 泛型应用相关?

* 什么是泛型?(参数化类型,编译时有效,运行时无效)
* 为什么要使用泛型?(编程更加灵活,通用性更强,可以提高代码的执行效率)
* 泛型有哪些应用类型?(泛型接口,泛型类,泛型方法)
* 泛型的通配符与限定符如何理解?(?,<? extends 类型>,<? super 类型>)
* 如何理解泛型应用中的类型擦除?(泛型运行时失效)

1. 序列化相关?

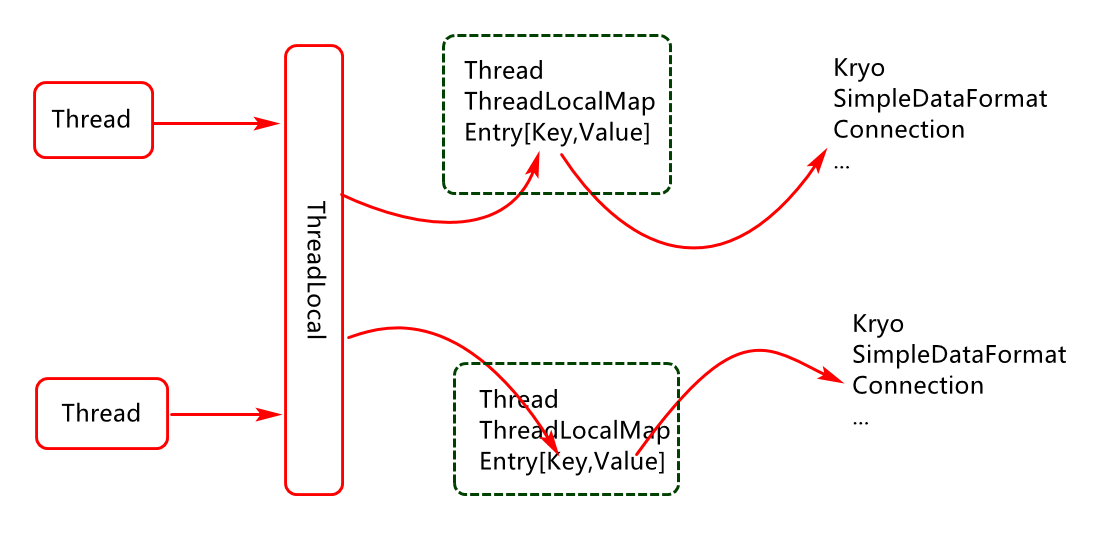
* 广义的序列化和反序列化.
* Java中序列化的实现?
* JAVA中序列化ID的作用?
* JAVA中序列化过程的加密和解密?(保证安全)
* JAVA中序列户的粒度控制?( transient, Externalizable)
* Java中序列化操作的性能完善?(Kryo,Hessian,…)
* Java中对象转换为JSON格式的字符串?(jackson,fastjson)

说说kryo框架的应用场景?(netty+kryo,spark+kryo,…)

1. ThreadLocal 的认识?

ThreadLocal 提供一种线程绑定机制,可以基于此对象将某个对象绑定当前线程中,也可以从当前线程获取某个对象.常用方法:

1. initialValue()创建要绑定的对象
2. set() 绑定对象到当前线程
3. get() 从当前线程获取对象
4. remove() 从当前线程移出对象(可有效防止内存泄漏)



记住:

1. 一个线程一个ThreadLocalMap对象
2. 一个ThreadLocalMap对象可以有多个Entry对象
3. Entry中的key为ThreadLocal类型的对象
4. Entry中的Value就是要绑定到当前线程中的对象.
5. 枚举类型的应用?

* 特殊类,主要用于更加严格的规范(约束)变量的类型
* 枚举类型的定义?
* 枚举实例的定义及理解?(只能在类的内部,类加载时创建)
* 枚举类型与字符串类型之间的转换?(两种方案)

参考:mybatis 中的ExecutorType,JdbcType

1. 注解的应用?

* 特殊类,相当于生活中的标签,主要用于描述类及成员.
* 注解的定义及描述?(借助@interface关键字)默认都实现Annotation接口
* 注解的应用?(描述类,描述属性,描述方法,…)
* 基于注解的描述执行具体业务?(借助反射,此时主要需要在运行时有效)

案例:参考课堂案例.

1. JAVA中四大引用类型及应用?

### 实践分析

案例分析1: 如下代码中d1,d2,d3是否指向的是内存中的同一个对象.

//1.获取类的字节码对象方式1

Class<Date> d1=Date.**class**;

//2.获取类的字节码对象方式2

Class<?> d2=Class.*forName*("java.util.Date");

//3.获取类的字节码对象方式3

Class<?> d3=**new** Date().getClass();

案例分析2: 获取类的加载器对象

//获取AppClassLoader

ClassLoader loader01=ClassLoader.*getSystemClassLoader*();

//获取 ExtClassLoader

ClassLoader loader02=loader01.getParent();

//根类加载器 BootstrapClassLoader (底层使用c完成)

ClassLoader loader03=loader02.getParent();

案例分析3: 采用不同方式加载类分析执行过程

**class** ClassC{

**static** **int** *c*=100;

**static** {

System.***out***.println("ClassC.static");

}

}

**class** ClassD **extends** ClassC{

**static** **int** *d*=200;

**static** {

System.***out***.println("ClassD.static");

}

}

测试代码

ClassLoader loader=ClassLoader.*getSystemClassLoader*();

loader.loadClass("com.java.oop.ClassC");

//System.out.println(ClassD.c);

案例4:分析如下代码是否有问题,该如何修改?

**class** ClassE{

//static Map<String,Object> pool=new HashMap<>();

**static** ClassE *instance*=**new** ClassE();

**static** Map<String,Object> *pool*=**new** HashMap<>();

**public** ClassE() {

*pool*.put("A", "100");

*pool*.put("B", "200");

}

}

案例5:分析如下代码,请问访问ClassAA类的Count1是否会初始化,InnerAA是否会被加载?

**class** ClassAA{

**static** Integer *count1*=100;

**static** Integer[] *count2*=**new** Integer[200];//延迟加载

**static** **class** InnerAA{

**static** {

System.***out***.println("InnerAA");

}

}

}

案例6:请问访问ClassCC中的instance对象时执行结果是怎样的?

**class** ClassCC{

//静态代码块在类加载时可以初始化

//但假如有多个静态成员,则会按编写顺序执行.

**static** {

System.***out***.println("static{}");

}

**static** ClassCC *instance*=**new** ClassCC();

**int** a;//0

{//实例代码块(每次构建对象都会执行)

a=100;

System.***out***.println("{}");

}

**public** ClassCC() {

a=200;

System.***out***.println("ClassCC()");

}

}

案例7:请问访问ClassDD中的count属性时,ClassDD类是否会被加载.

**class** ClassDD{

**public** **static** **final** **int** ***count***=100;

**public** **static** **final** String ***LOCK***="LOCK";

**public** **static** **final** Date ***date***=**new** Date();

**static** {

System.***out***.println("static{}");

}

}

案例8:泛型案例应用分析

interface Tast<Param,Args,Result>{

Result execute(Param p1,Args p2);

}

class ConvertTask implements Tast <String,String,Object>{

Object execute(String p1,String p2);

}

class SortTask<Result> implements Tast<String,String,Result>{

Result execute(String p1,String p2){…}

}

class DaoFactory{

<T>T getInstance(Class<T> cls){…}

}

class ReflectUtil{

static <T>T getInstance(Class<T> cls){…}

}

class ContainerUtil{

static <T>void sort(List<T> list){}

static void swap(List<? extends CharSequence> list){}

static <T extends Number>void convert(List<T> lsit){}

}

案例9:编写基于Hession完成序列化操作的工具类.

## 作业

1. 现有一个DefaultSearchService类,代码如下:

**class** DefaultSearchService{

**public** Object search(String key) {

//System.out.println("start:"+System.currentTimeMillis());

System.***out***.println("search by key ...");

//System.out.println("end:"+System.currentTimeMillis());

**return** "search result";

}

}

请问: 如何在不修改此类的情况下添加简易的日志输出(功能输出)

1. 现有一MailService接口和一个实现类DefaultMailService,代码如下:

**interface** MailService{

**void** send(String content);

}

**final** **class** DefaultMailService **implements** MailService{

@Override

**public** **void** send(String content) {

System.***out***.println("send "+content);

}

}

请问: 如何在不修改如上类的情况下添加简易的日志输出(功能输出)

1. 基于LinkedList编写一个Stack容器(算法先进后出)?

**class** Stack<E>{

**private** LinkedList<E> list=**new** LinkedList<>();

/\*\*

\* 入栈

\* **@param** e

\*/

**public** **void** push(E e) {

list.addFirst(e);

}

/\*\*

\* 出栈

\* **@return**

\*/

**public** E pop() {

**return** list.pop();

}

}

1. 基于LinkedHashMap编写一个LruCache容器?

**class** LRUCache<K,V> **extends** LinkedHashMap<K, V> {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**protected** **int** maxElements;

**public** LRUCache(**int** maxSize) {

**super**(maxSize,0.75f, **true**);//true表示记录元素访问顺序

**this**.maxElements = maxSize;

}

/\*\*

\* 此方法的返回值决定容器中是否要移除元素.

\* 1)true表示移除元素

\* 2)false表示不移除

\*/

@Override

**protected** **boolean** removeEldestEntry(java.util.Map.Entry<K, V> eldest) {

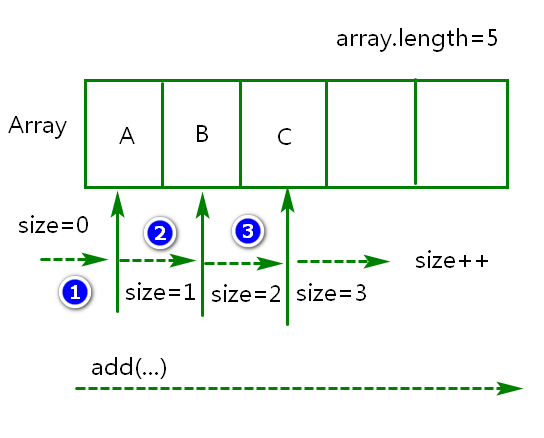
**return** size()>**this**.maxElements;

}

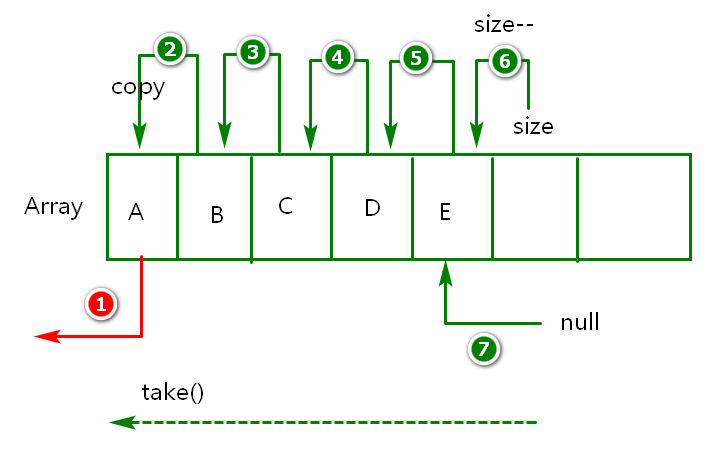
}

1. 基于”数组”结构写一个”FIFO”算法的队列容器.(尝试去做)
2. 类名:Container<T>
3. 方法:add(T t),take()

实现分析:添加操作



实现分析:取操作实现



代码分享:

/\*\*

\* 创建一个容器(队列容器)

\* 1)数据结构:数组(线性结构)

\* 2)算法:FIFO

\*/

**class** ArrayContainer<T>{

/\*\*通过数组存储元素\*/

**private** Object[] array;

/\*\*通过size属性记录有效元素个数(

实际存储的我们放入的元素)\*/

**private** **int** size;

**public** ArrayContainer() {

**this**(16);

}

**public** ArrayContainer(**int** cap) {

**if**(cap==0)

**throw** **new** IllegalArgumentException("你传入的参数无效");

array=**new** Object[cap];

}

/\*\*向容器添加数据:永远添加在size位置\*/

**public** **void** add(T t) {

//1.判定容器是否已满,满了扩容

**if**(size==array.length) {

//创建新数组,其长度为原数组长度两倍

//将原数组数据复制到新数组

//让原数组原先的引用指向新的数组

array=Arrays.*copyOf*(array,2\*array.length);

}

//2.将数据放在size位置

array[size]=t;

//3.有效元素个数加1

size++;

}

/\*\*从容器获取数据:永远从第0个位置开始\*/

@SuppressWarnings("unchecked")

**public** T take() {

//1.判定容器是否为空

**if**(size==0)

**throw** **new** NoSuchElementException("没有元素");

//2.取数据

Object obj=array[0];

//3.移动元素

System.*arraycopy*(array,//从哪个数组拷贝

1,//从哪个位置复制

array,//放到哪个数组

0,//从哪开始放

size-1);//放多少

//4.有效元素个数减1

size--;

//5.size位置元素设置为null(可选)

array[size]=**null**;

//6.返回元素

**return** (T)obj;

}

**public** **int** size() {

**return** size;

}

@Override

**public** String toString() {//不要求写

**if** (size==0)**return** "[]";

StringBuilder sb = **new** StringBuilder();

sb.append('[');

**for** (**int** i=0;i<size;i++) {

sb.append(array[i]).append(",");

}

sb.deleteCharAt(sb.lastIndexOf(","));

**return** sb.append("]").toString();

}

}

1. 基于jackson库将一个java对象转换为json串.(必须要做)
2. 添加jackson依赖:

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.9.8</version>

</dependency>

2)将SysLog对象转换为json格式字符串

...

SysLog log1=**new** SysLog();

log1.setId(100);

log1.setUsername("tedu");

//构建jackson中的对象转换器

ObjectMapper om=**new** ObjectMapper();

//将对象转换为json格式的字符串(序列化)

String jsonLog=om.writeValueAsString(log1);

System.***out***.println(jsonLog);

//将串转换为json格式的对象(反序列化)

SysLog log2=om.readValue(jsonLog,SysLog.**class**);

System.***out***.println(log2);

1. 基于fastjson库将一个java对象转换为json串.(必须要做)
2. 添加fastjson依赖

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>1.2.56</version>

</dependency>

1. 将SysLog对象转换为json格式字符串

...

SysLog log1=**new** SysLog();

log1.setId(100);

log1.setUsername("tedu");

//将对象转换为json格式的字符串(序列化)

String jsonStr=

JSON.*toJSONString*(log1);

System.***out***.println(jsonStr);

//将串转换为json格式的对象(反序列化)

SysLog log2=

JSON.*parseObject*(jsonStr,SysLog.**class**);

System.***out***.println(log2);

1. 如何对一个类进行单例设计?(多写几种)