程序员面试制胜宝典

图灵学院撰写整理
Tuling Clooege Write

如您对本书有任何疑惑
可添加客服微信：tulingtiku

图灵学院

独家内部教材

目录

一、基础篇

1.1 基本功

1.1.1 面向对象特征

封装，继承，多态和抽象

1. 封装

封装给对象提供了隐藏内部特性和行为的能力。对象提供一些能被其他对象访问的方法来改变它内部的数据。在 Java 当中，有 3 种修饰符：public，private 和 protected。每一种修饰符给其他的位于同一个包或者不同包下面对象赋予了不同的访问权限。

下面列出了使用封装的一些好处：

- 通过隐藏对象的属性来保护对象内部的状态
- 提高了代码的可用性和可维护性，因为对象的行为可以被单独的改变或者是扩展
- 禁止对象之间的不良交互提高模块化

2. 继承

继承给对象提供了从基类获取字段和方法的能力。继承提供了代码的重用行，也可以在不修改类的情况下给现存的类添加新特性。

3. 多态

多态是编程语言给不同的底层数据类型做相同的接口展示的一种能力。一个多态类型上的操作可以应用到其他类型的值上面。

4. 抽象

抽象是把想法从具体的实例中分离出来的步骤，因此，要根据他们的功能而不是实现细节来创建类。 Java 支持创建只暴漏接口而不包含方法实现的抽象的类。这种抽象技术的主要目的是把类的行为和实现细节分离开。

1.1.2 final, finally, finalize 的区别

1. final

修饰符（关键字）如果一个类被声明为 final，意味着它不能再派生出新的子类，不能作为父类被继承。因此一个类不能既被声明为 abstract 的，又被声明为 final 的。将变量或方法声明为 final，可以保证它们在使用中不被改变。被声明为 final 的变量必须在声明时给定初值，而在以后的引用中只能读取，不可修改。被声明为 final 的方法也同样只能使用，不能重写。

2. finally
在异常处理时提供 finally 块来执行任何清除操作。如果抛出一个异常，那么相匹配的 catch 子句就会执行，然后控制就会进入 finally 块（如果有的话）。

3. finalize
方法名。Java 技术允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的。它是在 Object 类中定义的，因此所有的类都继承了它。子类覆盖 finalize() 方法以整理系统资源或者执行其他清理工作。finalize() 方法是在垃圾收集器删除对象之前对这个对象调用的。

1.1.3 int 和 Integer 有什么区别

int 是基本数据类型；Integer 是其包装类，注意是一个类。

为什么要提供包装类呢？

- 一是为了在各种类型间转化，通过各种方法的调用。否则 你无法直接通过变量转化。比如，现在 int 要转为 String：

```
int a=0;
String result=Integer.toString(a);
```

在 java 中包装类，比较多的用途是用在于各种数据类型的转化中。

我写几个 demo

```
//通过包装类来实现转化的
int num=Integer.valueOf("12");
int num2=Integer.parseInt("12");
double num3=Double.valueOf("12.2");
double num4=Double.parseDouble("12.2");
//其他的类似。通过基本数据类型的包装来的 valueOf 和 parseXX 来实现 String 转为 XX
String a=String.valueOf("1234");//这里括号中几乎可以是任何类型
String b=String.valueOf(true);
String c=new Integer(12).toString();//通过包装类的 toString()也可以
String d=new Double(2.3).toString();
```

再举例下。比如我现在要用泛型

```
List<Integer> nums;
```

这里<>需要类。如果你用 int。它会报错的。

1.1.4 重载和重写的区别

- **override（重写）**

1. 方法名、参数、返回值相同。
2. 子类方法不能缩小父类方法的访问权限。
3. 子类方法不能抛出比父类方法更多的异常(但子类方法可以不抛出异常)。
4. 存在于父类和子类之间。
5. 方法被定义为 final 不能被重写。

- **overload（重载）**

1. 参数类型、个数、顺序至少有一个不相同。
2. 不能重载只有返回值不同的方法名。
3. 存在于父类和子类、同类中。

区别点	重载	重写（覆写）
英文	Overloading	Overiding
定义	方法名称相同，参数的类型或个数不同	方法名称、参数类型、返回值类型全部相同
权限	对权限没要求	被重写的方法不能拥有更严格的权限
范围	发生在一个类中	发生在继承类中

1.1.5 抽象类和接口有什么区别

- 接口是公开的，里面不能有私有的方法或变量，是用于让别人使用的，而抽象类是可以有私有方法或私有变量的；
- 另外，实现接口的一定要实现接口里定义的所有方法，而实现抽象类可以有选择地重写需要用到的方法，一般的应用里，最顶级的是接口，然后是抽象类实现接口，最后才到具体类实现；
- 还有，接口可以实现多重继承，而一个类只能继承一个超类，但可以通过继承多个接口实现多重继承；
- 接口还有标识（里面没有任何方法，如 Remote 接口）和数据共享（里面的变量全是常量）的作用。

1.1.6 说说反射的用途及实现

Java 反射机制主要提供了以下功能：**在运行时构造一个类的对象；判断一个类所具有的成员变量和方法；调用一个对象的方法；生成动态代理。反射最大的应用就是框架**。
Java 反射的主要功能：

- 确定一个对象的类
- 取出类的 modifiers,数据成员,方法,构造器,和超类
- 找出某个接口里定义的常量和方法说明

- 创建一个类实例,这个实例在运行时刻才有名字(运行时间才生成的对象)
- 取得和设定对象数据成员的值,如果数据成员名是运行时刻确定的也能做到
- 在运行时刻调用动态对象的方法
- 创建数组,数组大小和类型在运行时刻才确定,也能更改数组成员的值

反射的应用很多，很多框架都有用到：

- spring 的 ioc/di 也是反射…
- javaBean 和 jsp 之间调用也是反射…
- struts 的 FormBean 和页面之间…也是通过反射调用…
- JDBC 的 classForName()也是反射…
- hibernate 的 find(Class clazz) 也是反射…

反射还有一个不得不说的问题，就是性能问题，大量使用反射系统性能大打折扣。怎么使用使你的系统达到最优就看你系统架构和综合使用问题啦，这里就不多说了。

来源：http://uule.iteye.com/blog/1423512

1.1.7 说说自定义注解的场景及实现

（此题自由发挥，就看你对注解的理解了）
登陆、权限拦截、日志处理，以及各种 Java 框架，如 Spring，Hibernate，JUnit。
提到注解就不能不说反射，Java 自定义注解是通过运行时靠反射获取注解。实际开发中，例如我们要获取某个方法的调用日志，可以通过 AOP（动态代理机制）给方法添加切面，通过反射来获取方法包含的注解，如果包含日志注解，就进行日志记录。

1.1.8 HTTP 请求的 GET 与 POST 方式的区别

GET 方法会把名值对追加在请求的 URL 后面。因为 URL 对字符数目有限制，进而限制了用在客户端请求的参数值的数目。并且请求中的参数值是可见的，因此，敏感信息不能用这种方式传递。
POST 方法通过把请求参数值放在请求体中来克服 GET 方法的限制，因此，可以发送的参数的数目是没有限制的。最后，通过 POST 请求传递的敏感信息对外部客户端是不可见的。

1.1.9 session 与 cookie 区别

cookie 是 Web 服务器发送给浏览器的一块信息。浏览器会在本地文件中给每一个 Web 服务器存储 cookie。以后浏览器在给特定的 Web 服务器发请求的时候，同时会发送所有为该服务器存储的 cookie。下面列出了 session 和 cookie 的区别：
无论客户端浏览器做怎么样的设置，session 都应该能正常工作。客户端可以选择禁用 cookie，但是， session 仍然是能够工作的，因为客户端无法禁用服务端的 session。

1.1.10 JDBC 流程

1. 加载 JDBC 驱动程序：
在连接数据库之前，首先要加载想要连接的数据库的驱动到 JVM（Java 虚拟机），这通过 java.lang.Class 类的静态方法 forName(String className)实现。
例如：

```
try{
    //加载 MySql 的驱动类
    Class.forName("com.mysql.jdbc.Driver") ;
}catch(ClassNotFoundException e){
    System.out.println("找不到驱动程序类 ，加载驱动失败！");
    e.printStackTrace() ;
}
```

成功加载后，会将 Driver 类的实例注册到 DriverManager 类中。

2. 提供 JDBC 连接的 URL

- 连接 URL 定义了连接数据库时的协议、子协议、数据源标识。
- 书写形式：协议：子协议：数据源标识

协议：在 JDBC 中总是以 jdbc 开始 子协议：是桥连接的驱动程序或是数据库管理系统名称。
数据源标识：标记找到数据库来源的地址与连接端口。

例如：
jdbc:mysql://localhost:3306/test?
useUnicode=true&characterEncoding=gbk;useUnicode=true;（MySql 的连接 URL）
表示使用 Unicode 字符集。如果 characterEncoding 设置为 gb2312 或 GBK，本参数必须设置为 true 。characterEncoding=gbk：字符编码方式。

3. 创建数据库的连接

- 要连接数据库，需要向 java.sql.DriverManager 请求并获得 Connection 对象，该对象就代表一个数据库的连接。
- 使用 DriverManager 的 getConnectin(String url , String username , String password)方法传入指定的欲连接的数据库的路径、数据库的用户名和 密码来获得。

```
例如： //连接 MySql 数据库，用户名和密码都是 root
String url = "jdbc:mysql://localhost:3306/test" ;
String username = "root" ;
String password = "root" ;
try{
Connection con = DriverManager.getConnection(url , username , password) ;
}catch(SQLException se){
```

```
System.out.println("数据库连接失败！ ");
se.printStackTrace() ;
```

4. 创建一个 Statement

要执行 SQL 语句，必须获得 java.sql.Statement 实例，Statement 实例分为以下 3 种类型：

1) 执行静态 SQL 语句。通常通过 Statement 实例实现。
2) 执行动态 SQL 语句。通常通过 PreparedStatement 实例实现。
3) 执行数据库存储过程。通常通过 CallableStatement 实例实现。

具体的实现方式：

```
Statement stmt = con.createStatement() ;
PreparedStatement pstmt = con.prepareStatement(sql) ;
CallableStatement cstmt = con.prepareCall("{CALL demoSp(? , ?)}") ;
```

5. 执行 SQL 语句

Statement 接口提供了三种执行 SQL 语句的方法：executeQuery、executeUpdate 和 execute。

1) ResultSet executeQuery(String sqlString)：执行查询数据库的 SQL 语句 ，返回一个结果集（ResultSet）对象。
2) int executeUpdate(String sqlString)：用于执行 INSERT、UPDATE 或 DELETE 语句以及 SQL DDL 语句，如：CREATE TABLE 和 DROP TABLE 等。
3) execute(sqlString):用于执行返回多个结果集、多个更新计数或二者组合的 语句。

具体实现的代码：

```
ResultSet rs = stmt.executeQuery(“SELECT * FROM …”) ;
int rows = stmt.executeUpdate(“INSERT INTO …”) ;
boolean flag = stmt.execute(String sql) ;
```

6. 处理结果

两种情况：

1) 执行更新返回的是本次操作影响到的记录数。
2) 执行查询返回的结果是一个 ResultSet 对象。

- ResultSet 包含符合 SQL 语句中条件的所有行，并且它通过一套 get 方法提供了对这些行中数据的访问。
- 使用结果集（ResultSet）对象的访问方法获取数据：

```
while(rs.next()){
    String name = rs.getString(“name”) ;
    String pass = rs.getString(1) ; //  此方法比较高效
}
```

（列是从左到右编号的，并且从列 1 开始）

7. 关闭 JDBC 对象
操作完成以后要把所有使用的 JDBC 对象全都关闭，以释放 JDBC 资源，关闭顺序和声明顺序相反：
1) 关闭记录集
2) 关闭声明
3) 关闭连接对象

```
if(rs != null){ // 关闭记录集
    try{
        rs.close() ;
    }catch(SQLException e){
        e.printStackTrace() ;
    }
}
if(stmt != null){ // 关闭声明
    try{
        stmt.close() ;
    }catch(SQLException e){
        e.printStackTrace() ;
    }
}
if(conn != null){ // 关闭连接对象
    try{
        conn.close() ;
    }catch(SQLException e){
        e.printStackTrace() ;
    }
}
```

1.1.11 MVC 设计思想

MVC 就是
M:Model 模型
V:View 视图
C:Controller 控制器

模型就是封装业务逻辑和数据的一个一个的模块；控制器就是调用这些模块的(java 中通常是用 Servlet 来实现,框架的话很多是用 Struts2 来实现这一层)；视图就主要是你看到的，比如 JSP 等。
当用户发出请求的时候，控制器根据请求来选择要处理的业务逻辑和要选择的数据，再返回去把结果输出到视图层，这里可能是进行重定向或转发等

1.1.12 equals 与 == 的区别

值类型（int,char,long,boolean 等）都是用==判断相等性。对象引用的话，判断引用所指的对象是否是同一个。equals 是 Object 的成员函数，有些类会覆盖（override）这个方法，用于判断对象的等价性。例如 String 类，两个引用所指向的 String 都是"abc"，但可能出现他们实际对应的对象并不是同一个（和 jvm 实现方式有关），因此用判断他们可能不相等，但用 equals 判断一定是相等的。

1.2 集合

1.2.1 List 和 Set 区别

List,Set 都是继承自 Collection 接口
List 特点：元素有放入顺序，元素可重复
Set 特点：元素无放入顺序，元素不可重复，重复元素会覆盖掉
（注意：元素虽然无放入顺序，但是元素在 set 中的位置是有该元素的 HashCode 决定的，其位置其实是固定的，加入 Set 的 Object 必须定义 equals()方法 ，另外 list 支持 for 循环，也就是通过下标来遍历，也可以用迭代器，但是 set 只能用迭代，因为他无序，无法用下标来取得 想要的值。）

Set 和 List 对比：
Set：检索元素效率低下，删除和插入效率高，插入和删除不会引起元素位置改变。
List：和数组类似，List 可以动态增长，查找元素效率高，插入删除元素效率低，因为会引起其他元素位置改变。

1.2.2 List 和 Map 区别

List 是对象集合，允许对象重复。
Map 是键值对的集合，不允许 key 重复。

1.2.3 Arraylist 与 LinkedList 区别

Arraylist：
优点：ArrayList 是实现了基于动态数组的数据结构,因为地址连续，一旦数据存储好了，查询操作效率会比较高（在内存里是连着放的）。
缺点：因为地址连续， ArrayList 要移动数据,所以插入和删除操作效率比较低。

LinkedList：
优点：LinkedList 基于链表的数据结构,地址是任意的，所以在开辟内存空间的时候不需要等一个连续的地址，对于新增和删除操作 add 和 remove，LinedList 比较占优势。LinkedList 适用于要头尾操作或插入指定位置的场景
缺点：因为 LinkedList 要移动指针,所以查询操作性能比较低。

适用场景分析：
当需要对数据进行对此访问的情况下选用 ArrayList，当需要对数据进行多次增加删除修改时采用 LinkedList。

1.2.4 ArrayList 与 Vector 区别

```
public ArrayList(int initialCapacity)//构造一个具有指定初始容量的空列表。
public ArrayList()//构造一个初始容量为 10 的空列表。
public ArrayList(Collection<? extends E> c)//构造一个包含指定 collection  的元素的列表
```

Vector 有四个构造方法：

```
public Vector()//使用指定的初始容量和等于零的容量增量构造一个空向量。
public Vector(int initialCapacity)//构造一个空向量，使其内部数据数组的大小，其标准容量增量为零。
public Vector(Collection<? extends E> c)//构造一个包含指定 collection 中  的元素的向量
public Vector(int initialCapacity,int capacityIncrement)//使用指定的初始容量和容量增量构造一个空的向量
```

ArrayList 和 Vector 都是用数组实现的，主要有这么四个区别：
(1) Vector 是多线程安全的，线程安全就是说多线程访问同一代码，不会产生不确定的结果。而 ArrayList 不是，这个可以从源码中看出，Vector 类中的方法很多有 synchronized 进行修饰，这样就导致了 Vector 在效率上无法与 ArrayList 相比；
(2) 两个都是采用的线性连续空间存储元素，但是当空间不足的时候，两个类的增加方式不同。
(3) Vector 可以设置增长因子，而 ArrayList 不可以。
(4) Vector 是一种老的动态数组，是线程同步的，效率很低，一般不赞成使用。

适用场景分析：
1) Vector 是线程同步的，所以它也是线程安全的，而 ArrayList 是线程异步的，是不安全的。如果不考虑到线程的安全因素，一般用 ArrayList 效率比较高。
2) 如果集合中的元素的数目大于目前集合数组的长度时，在集合中使用数据量比较大的数据，用 Vector 有一定的优势。

1.2.5 HashMap 和 Hashtable 的区别

1. hashMap 去掉了 HashTable 的 contains 方法，但是加上了 containsValue()和 containsKey()方法。
2. hashTable 同步的，而 HashMap 是非同步的，效率上逼 hashTable 要高。
3. hashMap 允许空键值，而 hashTable 不允许。

注意：
TreeMap：非线程安全基于红黑树实现。
TreeMap 没有调优选项，因为该树总处于平衡状态。
Treemap：适用于按自然顺序或自定义顺序遍历键(key)。
参考：http://blog.csdn.net/qq_22118507/article/details/51576319

1.2.6 HashSet 和 HashMap 区别

1) set 是线性结构，set 中的值不能重复，hashset 是 set 的 hash 实现，hashset 中值不能重复是用 hashmap 的 key 来实现的。
2) map 是键值对映射，可以空键空值。HashMap 是 Map 接口的 hash 实现，key 的唯一性是通过 key 值 hash 值的唯一来确定，value 值是则是链表结构。
3) 他们的共同点都是 hash 算法实现的唯一性，他们都不能持有基本类型，只能持有对象

1.2.7 HashMap 和 ConcurrentHashMap 的区别

ConcurrentHashMap 是线程安全的 HashMap 的实现。
(1) ConcurrentHashMap 对整个桶数组进行了分割分段(Segment)，然后在每一个分段上都用 lock 锁进行保护，相对于 HashTable 的 syn 关键字锁的粒度更精细了一些，并发性能更好，而 HashMap 没有锁机制，不是线程安全的。
(2) HashMap 的键值对允许有 null，但是 ConCurrentHashMap 都不允许。

HashMap 的工作原理及代码实现
参考：https://tracylihui.github.io/2015/07/01/Java 集合学习 1：HashMap 的实现原理/

ConcurrentHashMap 的工作原理及代码实现

HashTable 里使用的是 synchronized 关键字，这其实是对对象加锁，锁住的都是对象整体，当 Hashtable 的大小增加到一定的时候，性能会急剧下降，因为迭代时需要被锁定很长的时间。
ConcurrentHashMap 算是对上述问题的优化，其构造函数如下，默认传入的是 16，0.75，16。

```
public ConcurrentHashMap(int paramInt1, float paramFloat, int paramInt2)     {
    //…
    int i = 0;
    int j = 1;
    while (j < paramInt2) {
        ++i;
        j <<= 1;
    }
    this.segmentShift = (32 - i);
    this.segmentMask = (j - 1);
    this.segments = Segment.newArray(j);
    //…
    int k = paramInt1 / j;
    if (k * j < paramInt1)
        ++k;
    int l = 1;
    while (l < k)
        l <<= 1;
    for (int i1 = 0; i1 < this.segments.length; ++i1)
        this.segments[i1] = new Segment(l, paramFloat);
}

public V put(K paramK, V paramV)       {
    if (paramV == null)
        throw new NullPointerException();
    int i = hash(paramK.hashCode()); //这里的 hash 函数和 HashMap 中的不一样
    return this.segments[(i >>> this.segmentShift & this.segmentMask)].put(paramK, i, paramV,
false);
}
```

ConcurrentHashMap 引入了分割(Segment)，上面代码中的最后一行其实就可以理解为把一个大的 Map 拆分成 N 个小的 HashTable，在 put 方法中，会根据 hash(paramK.hashCode())来决定具体存放进哪个 Segment，如果查看 Segment 的 put 操作，我们会发现内部使用的同步机制是基于 lock 操作的，这样就可以对 Map 的一部分（Segment）进行上锁，这样影响的只是将要放入同一个 Segment 的元素的 put 操作，保证同步的时候，锁住的不是整个 Map（HashTable 就是这么做的），相对于 HashTable 提高了多线程环境下的性能，因此 HashTable 已经被淘汰了。

1.3 线程

1.3.1 创建线程的方式及实现

Java 中创建线程主要有三种方式：

1. 继承 Thread 类创建线程类

（1）定义 Thread 类的子类，并重写该类的 run 方法，该 run 方法的方法体就代表了线程要完成的任务。因此把 run()方法称为执行体。

（2）创建 Thread 子类的实例，即创建了线程对象。

（3）调用线程对象的 start()方法来启动该线程。

```
package com.thread;

public class FirstThreadTest extends Thread{
    int i = 0; 5
    //重写 run 方法，run 方法的方法体就是现场执行体
    public void run() {
        for(;i<100;i++){
            System.out.println(getName()+" "+i);
        }
    }
    public static void main(String[] args) {
        for(int i = 0;i< 100;i++)
        {
            System.out.println(Thread.currentThread().getName()+" : "+i);
            if(i==20)
            {
                new FirstThreadTest().start();
                new FirstThreadTest().start();
            }
        }
    }
}
```

上述代码中 Thread.currentThread()方法返回当前正在执行的线程对象。getName()方法返回调用该方法的线程的名字。

2. 通过 Runnable 接口创建线程类

（1）定义 runnable 接口的实现类，并重写该接口的 run()方法，该 run()方法的方法体同样是该线程的线程执行体。

（2）创建 Runnable 实现类的实例，并依此实例作为 Thread 的 target 来创建 Thread 对象，该 Thread 对象才是真正的线程对象。

（3）调用线程对象的 start()方法来启动该线程。

```
package com.thread;

public class RunnableThreadTest implements Runnable {
    private int i;

    public void run()
    {
        for(i = 0;i <100;i++)
        {
            System.out.println(Thread.currentThread().getName()+" "+i);
        }
    }
    public static void main(String[] args)
    {
        for(int i = 0;i < 100;i++)
        {
            System.out.println(Thread.currentThread().getName()+" "+i);
            if(i==20)
            {
                RunnableThreadTest rtt = new RunnableThreadTest();
                new Thread(rtt,"新线程 1").start();
                new Thread(rtt,"新线程 2").start();
            }
        }
    }
}
```

3. 通过 Callable 和 Future 创建线程

（1）创建 Callable 接口的实现类，并实现 call()方法，该 call()方法将作为线程执行体，并且有返回值。

（2）创建 Callable 实现类的实例，使用 FutureTask 类来包装 Callable 对象，该 FutureTask 对象封装了该 Callable 对象的 call()方法的返回值。

（3）使用 FutureTask 对象作为 Thread 对象的 target 创建并启动新线程。

（4）调用 FutureTask 对象的 get()方法来获得子线程执行结束后的返回值

```
package com.thread;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

public class CallableThreadTest implements Callable<Integer>{
    public static void main(String[] args)
    {
```

```
        CallableThreadTest ctt = new CallableThreadTest();
        FutureTask<Integer> ft = new FutureTask<>(ctt);
        for(int i = 0;i < 100;i++) {
            System.out.println(Thread.currentThread().getName()+" 的循 环变量 i 的值"+i);
            if(i==20) {
                new Thread(ft,"有返回值的线程").start();
            }
        }
        try
        {
            System.out.println("子线程的返回值： "+ft.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }

    @Override
    public Integer call() throws Exception{
        int i = 0;
        for(;i<100;i++) {
            System.out.println(Thread.currentThread().getName()+" "+i);
        }
        return i;
    }
}
```

创建线程的三种方式的对比：

采用实现 Runnable、Callable 接口的方式创见多线程时，优势是：

线程类只是实现了 Runnable 接口或 Callable 接口，还可以继承其他类。

在这种方式下，多个线程可以共享同一个 target 对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将 CPU、代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。

劣势是：

编程稍微复杂，如果要访问当前线程，则必须使用 Thread.currentThread()方法。

使用继承 Thread 类的方式创建多线程时优势是：

编写简单，如果需要访问当前线程，则无需使用 Thread.currentThread()方法，直接使用 this 即可获得当前线程。

劣势是：

线程类已经继承了 Thread 类，所以不能再继承其他父类。

1.3.2 sleep() 、join()、yield()有什么区别

1. sleep()方法
在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。 让其他线程有机会继续执行，但它并不释放对象锁。也就是如果有 Synchronized 同步块，其他线程仍然不能访问共享数据。注意该方法要捕获异常比如有两个线程同时执行(没有 Synchronized)，一个线程优先级为 MAX_PRIORITY，另一个为 MIN_PRIORITY，如果没有 Sleep()方法，只有高优先级的线程执行完成后，低优先级的线程才能执行；但当高优先级的线程 sleep(5000)后，低优先级就有机会执行了。
总之，sleep()可以使低优先级的线程得到执行的机会，当然也可以让同优先级、高优先级的线程有执行的机会。

2. yield()方法
yield()方法和 sleep()方法类似，也不会释放“锁标志”，区别在于，它没有参数，即 yield()方法只是使当前线程重新回到可执行状态，所以执行 yield()的线程有可能在进入到可执行状态后马上又被执行，另外 yield()方法只能使同优先级或者高优先级的线程得到执行机会，这也和 sleep()方法不同。

3. join()方法
Thread 的非静态方法 join()让一个线程 B“加入”到另外一个线程 A 的尾部。在 A 执行完毕之前，B 不能工作。

```
Thread t = new MyThread();
t.start();
t.join();
```

保证当前线程停止执行，直到该线程所加入的线程完成为止。然而，如果它加入的线程没有存活，则当前线程不需要停止。

1.3.3 说说 CountDownLatch 原理

参考：
分析 CountDownLatch 的实现原理
什么时候使用 CountDownLatch
Java 并发编程：CountDownLatch、CyclicBarrier 和 Semaphore

1.3.4 说说 CyclicBarrier 原理

参考：
JUC 回顾之 CyclicBarrier 底层实现和原理

1.3.5 说说 Semaphore 原理

JAVA 多线程 - 信号量(Semaphore)
JUC 回顾之 Semaphore 底层实现和原理

1.3.6 说说 Exchanger 原理

java.util.concurrent.Exchanger 应用范例与原理浅析

1.3.7 说说 CountDownLatch 与 CyclicBarrier 区别

CountDownLatch	CyclicBarrier
减计数方式	加计数方式
计算为 0 时释放所有等待的线程	计数达到指定值时释放所有等待线程
计数为 0 时，无法重置	计数达到指定值时，计数置为 0 重新开始
调用 countDown()方法计数减一，调用 await()方法只进行阻塞，对计数没任何影响	调用 await()方法计数加 1，若加 1 后的值不等于构造方法的值，则线程阻塞
不可重复利用	可重复利用

尽量把 CyclicBarrier 和 CountDownLatch 的区别说通俗点

1.3.8 ThreadLocal 原理分析

Java 并发编程：深入剖析 ThreadLocal

1.3.9 讲讲线程池的实现原理

主要是 ThreadPoolExecutor 的实现原理
Java 并发编程：线程池的使用

1.3.10 线程池的几种方式

1. newFixedThreadPool(int nThreads)
创建一个固定长度的线程池，每当提交一个任务就创建一个线程，直到达到线程池的最大数量，这时线程规模将不再变化，当线程发生未预期的错误而结束时，线程池会补充一个新的线程。

2. newCachedThreadPool()
创建一个可缓存的线程池，如果线程池的规模超过了处理需求，将自动回收空闲线程，而当需求增加时，则可以自动添加新线程，线程池的规模不存在任何限制

3. newSingleThreadExecutor()
这是一个单线程的 Executor，它创建单个工作线程来执行任务，如果这个线程异常结束，会创建一个新的来替代它；它的特点是能确保依照任务在队列中的顺序来串行执行

4. newScheduledThreadPool(int corePoolSize)
创建了一个固定长度的线程池，而且以延迟或定时的方式来执行任务，类似于 Timer。

举个栗子

```
private static final Executor exec=Executors.newFixedThreadPool(50);
Runnable runnable=new Runnable(){
    public void run(){
        ...
    }
}
exec.execute(runnable);

Callable<Object> callable=new Callable<Object>() {
    public Object call() throws Exception {
        return null;
    }
};

Future future=executorService.submit(callable);
future.get(); // 等待计算完成后，获取结果
future.isDone(); // 如果任务已完成，则返回 true
future.isCancelled(); // 如果在任务正常完成前将其取消，则返回 true
future.cancel(true); // 试图取消对此任务的执行，true 中断运行的任务，false 允许正在运行的任务运行完成
```

参考：
[创建线程池的几种方式]

1.3.11 线程的生命周期

新建(New)、就绪（Runnable）、运行（Running）、阻塞(Blocked)和死亡(Dead)5 种状态。
生命周期的五种状态：
新建（new Thread）
当创建 Thread 类的一个实例（对象）时，此线程进入新建状态（未被启动）。
例如：Thread t1=new Thread();

就绪（runnable）
线程已经被启动，正在等待被分配给 CPU 时间片，也就是说此时线程正在就绪队列中排队等候得到 CPU 资源。例如：t1.start();

运行（running）
线程获得 CPU 资源正在执行任务（run()方法），此时除非此线程自动放弃 CPU 资源或者有优先级更高的线程进入，线程将一直运行到结束。

死亡（dead）
当线程执行完毕或被其它线程杀死，线程就进入死亡状态，这时线程不可能再进入就绪状态等待执行。
自然终止：正常运行 run()方法后终止。
异常终止：调用**stop()**方法让一个线程终止运行。

堵塞（blocked）
由于某种原因导致正在运行的线程让出 CPU 并暂停自己的执行，即进入堵塞状态。
正在睡眠：用 sleep(long t) 方法可使线程进入睡眠方式。一个睡眠着的线程在指定的时间过去可进入就绪状态。
正在等待：调用 wait()方法。（调用 motify()方法回到就绪状态）
被另一个线程所阻塞：调用 suspend()方法。（调用 resume()方法恢复）

参考：
线程的生命周期

1.4 锁机制

1.4.1 说说线程安全问题

线程安全是指要控制多个线程对某个资源的有序访问或修改，而在这些线程之间没有产生冲突。在 Java 里，线程安全一般体现在两个方面：
1. 多个 thread 对同一个 java 实例的访问（read 和 modify）不会相互干扰，它主要体现在关键字 synchronized。如 ArrayList 和 Vector，HashMap 和 Hashtable（后者每个方法前都有 synchronized 关键字）。如果你在 interator 一个 List 对象时，其它线程 remove 一个 element，问题就出现了。

2. 每个线程都有自己的字段，而不会在多个线程之间共享。它主要体现在 java.lang.ThreadLocal 类，而没有 Java 关键字支持，如像 static、transient 那样。

1.4.2 volatile 实现原理

聊聊并发（一）——深入分析 Volatile 的实现原理

1.4.3 悲观锁 乐观锁

乐观锁 悲观锁是一种思想。可以用在很多方面。
比如数据库方面：
悲观锁就是 for update（锁定查询的行）；
乐观锁就是 version 字段（比较跟上一次的版本号，如果一样则更新，如果失败则要重复读 - 比较 - 写的操作。）

JDK 方面：
悲观锁就是 sync
乐观锁就是原子类（内部使用 CAS 实现）

本质来说，就是悲观锁认为总会有人抢我的。
乐观锁就认为，基本没人抢。

1.4.4 CAS 乐观锁

乐观锁是一种思想，即认为读多写少，遇到并发写的可能性比较低，所以采取在写时先读出当前版本号，然后加锁操作（比较跟上一次的版本号，如果一样则更新），如果失败则要重复读比较写的操作。
CAS 是一种更新的原子操作，比较当前值跟传入值是否一样，一样则更新，否则失败。
CAS 顶多算是乐观锁写那一步操作的一种实现方式罢了，不用 CAS 自己加锁也是可以的。

ABA 问题
ABA：如果另一个线程修改 V 值假设原来是 A，先修改成 B，再修改回成 A，当前线程的 CAS 操作无法分辨当前 V 值是否发生过变化。
参考：
Java CAS 和 ABA 问题

1.4.5 乐观锁的业务场景及实现方式

乐观锁（Optimistic Lock）：

每次获取数据的时候，都不会担心数据被修改，所以每次获取数据的时候都不会进行加锁，但是在更新数据的时候需要判断该数据是否被别人修改过。如果数据被其他线程修改，则不进行数据更新，如果数据没有被其他线程修改，则进行数据更新。由于数据没有进行加锁，期间该数据可以被其他线程进行读写操作。

乐观锁：比较适合读取操作比较频繁的场景，如果出现大量的写入操作，数据发生冲突的可能性就会增大，为了保证数据的一致性，应用层需要不断的重新获取数据，这样会增加大量的查询操作，降低了系统的吞吐量。

二、核心篇

2.1 数据存储

2.1.1 MySQL 索引使用的注意事项

参考：
mysql 索引使用技巧及注意事项

2.1.2 说说反模式设计

参考：
每个程序员要注意的 9 种反模式

2.1.3 说说分库与分表设计

分表与分库使用场景以及设计方式

2.1.4 分库与分表带来的分布式困境与应对之策

服务端指南 数据存储篇 | MySQL（09） 分库与分表带来的分布式困境与应对之策

2.1.5 说说 SQL 优化之道

sql 优化的几种方法

2.1.6 MySQL 遇到的死锁问题

参考：
Mysql 并发时经典常见的死锁原因及解决方法

2.1.7 存储引擎的 InnoDB 与 MyISAM

1）InnoDB 支持事务，MyISAM 不支持，这一点是非常之重要。事务是一种高级的处理方式，如在一些列增删改中只要哪个出错还可以回滚还原，而 MyISAM 就不可以了。
2）MyISAM 适合查询以及插入为主的应用，InnoDB 适合频繁修改以及涉及到安全性较高的应用
3）InnoDB 支持外键，MyISAM 不支持
4）从 MySQL5.5.5 以后，InnoDB 是默认引擎
5）InnoDB 不支持 FULLTEXT 类型的索引
6）InnoDB 中不保存表的行数，如 select count() from table 时，InnoDB 需要扫描一遍整个表来计算有多少行，但是 MyISAM 只要简单的读出保存好的行数即可。注意的是，当 count() 语句包含 where 条件时 MyISAM 也需要扫描整个表
7）对于自增长的字段，InnoDB 中必须包含只有该字段的索引，但是在 MyISAM 表中可以和其他字段一起建立联合索引
8）清空整个表时，InnoDB 是一行一行的删除，效率非常慢。MyISAM 则会重建表
9）InnoDB 支持行锁（某些情况下还是锁整表，如 update table set a=1 where user like ‘%lee%’
参考：
MySQL 存储引擎之 MyIsam 和 Innodb 总结性梳理

2.1.8 数据库索引的原理

参考：
http://blog.csdn.net/suifeng3051/article/details/52669644

2.1.9 为什么要用 Btree

鉴于 Btree 具有良好的定位特性，其常被用于对检索时间要求苛刻的场合，例如：
1. Btree 索引是数据库中存取和查找文件(称为记录或键值)的一种方法。
2. 硬盘中的结点也是 Btree 结构的。与内存相比，硬盘必须花成倍的时间来存取一个数据元素，这是因为硬盘的机械部件读写数据的速度远远赶不上纯电子媒体的内存。与一个结点两个分支的二元树相比，Btree 利用多个分支（称为子树）的结点，减少获取记录时所经历的结点数，从而达到节省存取时间的目的。

2.1.10 聚集索引与非聚集索引的区别

参考：
快速理解聚集索引和非聚集索引

2.1.11 limit 20000 加载很慢怎么解决

LIMIT n 等价于 LIMIT 0,n
此题总结一下就是让 limit 走索引去查询，例如：order by 索引字段，或者 limit 前面根 where 条件走索引字段等等。
参考：
MYSQL 分页 limit 速度太慢优化方法

2.1.12 选择合适的分布式主键方案

参考：
分布式系统唯一 ID 生成方案汇总

2.1.13 选择合适的数据存储方案

- 关系型数据库 MySQL

  MySQL 是一个最流行的关系型数据库，在互联网产品中应用比较广泛。一般情况下，MySQL 数据库是选择的第一方案，基本上有 80%～90% 的场景都是基于 MySQL 数据库的。因为，需要关系型数据库进行管理，此外，业务存在许多事务性的操作，需要保证事务的强一致性。同时，可能还存在一些复杂的 SQL 的查询。值得注意的是，前期尽量减少表的联合查询，便于后期数据量增大的情况下，做数据库的分库分表。

- 内存数据库 Redis

  随着数据量的增长，MySQL 已经满足不了大型互联网类应用的需求。因此，Redis 基于内存存储数据，可以极大的提高查询性能，对产品在架构上很好的补充。例如，为了提高服务端接口的访问速度，尽可能将读频率高的热点数据存放在 Redis 中。这个是非常典型的以空间换时间的策略，使用更多的内存换取 CPU 资源，通过增加系统的内存消耗，来加快程序的运行速度。

  在某些场景下，可以充分的利用 Redis 的特性，大大提高效率。这些场景包括缓存，会话缓存，时效性，访问频率，计数器，社交列表，记录用户判定信息，交集、并集和差集，热门列表与排行榜，最新动态等。

  使用 Redis 做缓存的时候，需要考虑数据不一致与脏读、缓存更新机制、缓存可用性、缓存服务降级、缓存穿透、缓存预热等缓存使用问题。

- 文档数据库 MongoDB

  MongoDB 是对传统关系型数据库的补充，它非常适合高伸缩性的场景，它是可扩展性

的表结构。基于这点，可以将预期范围内，表结构可能会不断扩展的 MySQL 表结构，通过 MongoDB 来存储，这就可以保证表结构的扩展性。

此外，日志系统数据量特别大，如果用 MongoDB 数据库存储这些数据，利用分片集群支持海量数据，同时使用聚集分析和 MapReduce 的能力，是个很好的选择。

MongoDB 还适合存储大尺寸的数据，GridFS 存储方案就是基于 MongoDB 的分布式文件存储系统。

- 列族数据库 HBase

HBase 适合海量数据的存储与高性能实时查询，它是运行于 HDFS 文件系统之上，并且作为 MapReduce 分布式处理的目标数据库，以支撑离线分析型应用。在数据仓库、数据集市、商业智能等领域发挥了越来越多的作用，在数以千计的企业中支撑着大量的大数据分析场景的应用。

- 全文搜索引擎 ElasticSearch

在一般情况下，关系型数据库的模糊查询，都是通过 like 的方式进行查询。其中，like “value%” 可以使用索引，但是对于 like “%value%” 这样的方式，执行全表查询，这在数据量小的表，不存在性能问题，但是对于海量数据，全表扫描是非常可怕的事情。

ElasticSearch 作为一个建立在全文搜索引擎 Apache Lucene 基础上的实时的分布式搜索和分析引擎，适用于处理实时搜索应用场景。此外，使用 ElasticSearch 全文搜索引擎，还可以支持多词条查询、匹配度与权重、自动联想、拼写纠错等高级功能。因此，可以使用 ElasticSearch 作为关系型数据库全文搜索的功能补充，将要进行全文搜索的数据缓存一份到 ElasticSearch 上，达到处理复杂的业务与提高查询速度的目的。

ElasticSearch 不仅仅适用于搜索场景，还非常适合日志处理与分析的场景。著名的 ELK 日志处理方案，由 ElasticSearch、Logstash 和 Kibana 三个组件组成，包括了日志收集、聚合、多维度查询、可视化显示等。

2.1.14 ObjectId 规则

参考：

MongoDB 学习笔记~ObjectId 主键的设计

mongodb 中的 _id 的 ObjectId 的生成规则

2.1.15 聊聊 MongoDB 使用场景

参考：

什么场景应该用 MongoDB ？

2.1.16 倒排索引

参考：

什么是倒排索引？

2.1.17 聊聊 ElasticSearch 使用场景

在一般情况下，关系型数据库的模糊查询，都是通过 like 的方式进行查询。其中，like “value%” 可以使用索引，但是对于 like “%value%” 这样的方式，执行全表查询，这在数据量小的表，不存在性能问题，但是对于海量数据，全表扫描是非常可怕的事情。

ElasticSearch 作为一个建立在全文搜索引擎 Apache Lucene 基础上的实时的分布式搜索和分析引擎，适用于处理实时搜索应用场景。此外，使用 ElasticSearch 全文搜索引擎，还可以支持多词条查询、匹配度与权重、自动联想、拼写纠错等高级功能。因此，可以使用 ElasticSearch 作为关系型数据库全文搜索的功能补充，将要进行全文搜索的数据缓存一份到 ElasticSearch 上，达到处理复杂的业务与提高查询速度的目的。

2.2 缓存使用

2.2.1 Redis 有哪些类型

Redis 支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及 zset(sorted set：有序集合)。

参考：

Redis 数据类型

2.2.2 Redis 内部结构

参考：

redis 内部数据结构深入浅出

2.2.3 聊聊 Redis 使用场景

随着数据量的增长，MySQL 已经满足不了大型互联网类应用的需求。因此，Redis 基于内存存储数据，可以极大的提高查询性能，对产品在架构上很好的补充。例如，为了提高服务端接口的访问速度，尽可能将读频率高的热点数据存放在 Redis 中。这个是非常典型的以空间换时间的策略，使用更多的内存换取 CPU 资源，通过增加系统的内存消耗，来加快程序的运行速度。

在某些场景下，可以充分的利用 Redis 的特性，大大提高效率。这些场景包括缓存，会话缓存，时效性，访问频率，计数器，社交列表，记录用户判定信息，交集、并集和差集，热门列表与排行榜，最新动态等。

使用 Redis 做缓存的时候，需要考虑数据不一致与脏读、缓存更新机制、缓存可用性、缓 存服务降级、缓存穿透、缓存预热等缓存使用问题。

2.2.4 Redis 持久化机制

参考：
redis 的持久化和缓存机制

2.2.5 Redis 如何实现持久化

参考：
Redis 如何实现持久化

2.2.6 Redis 集群方案与实现

参考：
redis 集群主流架构方案分析

2.2.7 Redis 为什么是单线程的

单纯的网络 IO 来说，量大到一定程度之后，多线程的确有优势——但并不是单纯的多线程， 而是每个线程自己有自己的 epoll 这样的模型，也就是多线程和 multiplexing 混合。一般这个开头我们都会跟一个“但是”。
但是。
还要考虑 Redis 操作的对象。它操作的对象是内存中的数据结构。如果在多线程中操作，那就需要为这些对象加锁。最终来说，多线程性能有提高，但是每个线程的效率严重下降了。而且程序的逻辑严重复杂化。

要知道 Redis 的数据结构并不全是简单的 KeyValue，还有列表，hash，map 等等复杂的结构，这些结构有可能会进行很细粒度的操作，比如在很长的列表后面添加一个元素，在 hash 当中添加或者删除一个对象，等等。这些操作还可以合成 MULTI/EXEC 的组。这样一个操作中可能就需要加非常多的锁，导致的结果是同步开销大大增加。这还带来一个恶果就是吞吐量虽然增大，但是响应延迟可能会增加。

Redis 在权衡之后的选择是用单线程，突出自己功能的灵活性。在单线程基础上任何原子操作都可以几乎无代价地实现，多么复杂的数据结构都可以轻松运用，甚至可以使用 Lua 脚本这样的功能。对于多线程来说这需要高得多的代价。

并不是所有的 KV 数据库或者内存数据库都应该用单线程，比如 ZooKeeper 就是多线程的，最终还是看作者自己的意愿和取舍。单线程的威力实际上非常强大，每核心效率也非常高，在今天的虚拟化环境当中可以充分利用云化环境来提高资源利用率。多线程自然是可以比单线程有更高的性能上限，但是在今天的计算环境中，即使是单机多线程的上限也往往不能满足需要了，需要进一步摸索的是多服务器集群化的方案，这些方案中多线程的技术照样是用 不上的，所以单线程、多进程的集群不失为一个时髦的解决方案。

链接：https://www.zhihu.com/question/23162208/answer/142424042

2.2.8 缓存奔溃

参考：
Redis 持久化

2.2.9 缓存降级

服务降级的目的，是为了防止 Redis 服务故障，导致数据库跟着一起发生雪崩问题。因此，对于不重要的缓存数据，可以采取服务降级策略，例如一个比较常见的做法就是，Redis 出现问题，不去数据库查询，而是直接返回默认值给用户。

2.2.10 使用缓存的合理性问题

参考：
Redis 实战（一） 使用缓存合理性

2.3 消息队列

2.3.1 消息队列的使用场景

主要解决应用耦合，异步消息，流量削锋等问题
消息队列使用的四种场景介绍

2.3.2 消息的重发补偿解决思路

参考：
JMS 消息传送机制

2.3.3 消息的幂等性解决思路

参考：
MQ 之如何做到消息幂等

2.3.4 消息的堆积解决思路

参考：
Sun Java System Message Queue 3.7 UR1 管理指南

2.3.5 自己如何实现消息队列

参考：
自己动手实现消息队列之 JMS

2.3.6 如何保证消息的有序性

参考：
消息队列的 exclusive consumer 功能是如何保证消息有序和防止脑裂的

三、框架篇

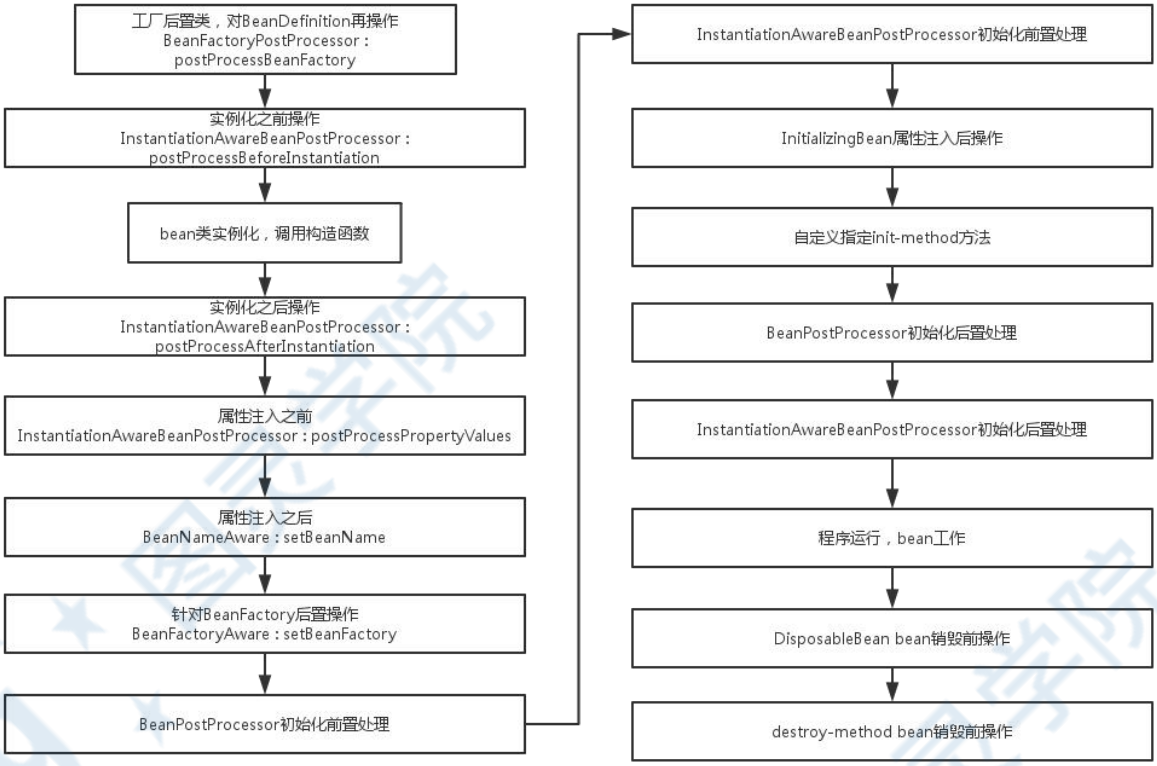
3.1 Spring

3.1.1 BeanFactory 和 ApplicationContext 有什么区别

beanfactory 顾名思义，它的核心概念就是 bean 工厂，用作于 bean 生命周期的管理，而 applicationcontext 这个概念就比较丰富了，单看名字（应用上下文）就能看出它包含的范围更广，它继承自 bean factory 但不仅仅是继承自这一个接口，还有继承了其他的接口，所以它不仅仅有 bean factory 相关概念，更是一个应用系统的上下文，其设计初衷应该是一个包罗万象的对外暴露的一个综合的 API。

3.1.2 Spring Bean 的生命周期

参考：
Spring Bean 生命周期详解



3.1.3 Spring IOC 如何实现

参考：
Spring：源码解读 Spring IOC 原理

3.1.4 说说 Spring AOP

参考：
Spring AOP 详解

3.1.5 Spring AOP 实现原理

参考：Spring AOP 实现原理

3.1.6 动态代理（cglib 与 JDK）

java 动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用 InvokeHandler 来处理。
而 cglib 动态代理是利用 asm 开源包，对代理对象类的 class 文件加载进来，通过修改其字节码生成子类来处理。

1．如果目标对象实现了接口，默认情况下会采用 JDK 的动态代理实现 AOP
2．如果目标对象实现了接口，可以强制使用 CGLIB 实现 AOP
3．如果目标对象没有实现了接口，必须采用 CGLIB 库，spring 会自动在 JDK 动态代理和 CGLIB 之间转换

如何强制使用 CGLIB 实现 AOP？
（1）添加 CGLIB 库，SPRING_HOME/cglib/*.jar
（2）在 spring 配置文件中加入<aop:aspectjautoproxy proxytargetclass=“true”/>

JDK 动态代理和 CGLIB 字节码生成的区别？
（1）JDK 动态代理只能对实现了接口的类生成代理，而不能针对类
（2）CGLIB 是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法，因为是继承，所以该类或方法最好不要声明成 final

参考：
Spring 的两种代理 JDK 和 CGLIB 的区别浅谈

3.1.7 Spring 事务实现方式

参考：
Spring 事务管理实现方式之编程式事务与声明式事务详解

3.1.8 Spring 事务底层原理

参考：
深入理解 Spring 事务原理

3.1.9 如何自定义注解实现功能

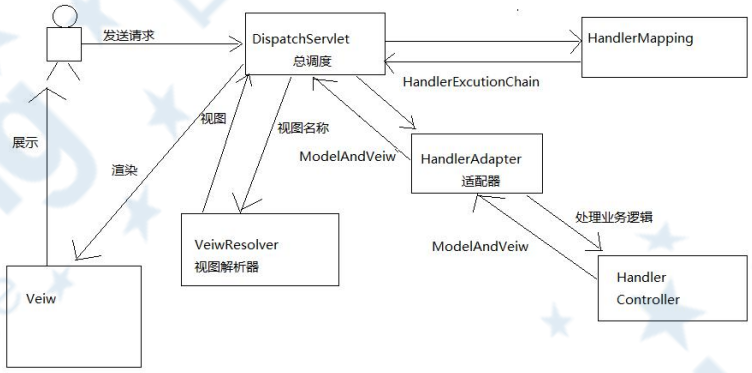
可以结合 spring 的 AOP，对注解进行拦截，提取注解。
大致流程为：
1. 新建一个注解@MyLog，加在需要注解申明的方法上面
2. 新建一个类 MyLogAspect，通过@Aspect 注解使该类成为切面类。
3. 通过@Pointcut 指定切入点 ，这里指定的切入点为 MyLog 注解类型，也就是被@MyLog 注解修饰的方法，进入该切入点。
4. MyLogAspect 中的方法通过加通知注解（@Before、@Around、@AfterReturning、@AfterThrowing、@After 等各种通知）指定要做的业务操作。

3.1.10 Spring MVC 运行流程

1. 先用文字描述
1) 用户发送请求到 DispatchServlet

2) DispatchServlet 根据请求路径查询具体的 Handler
3) HandlerMapping 返回一个 HandlerExcutionChain 给 DispatchServlet HandlerExcutionChain：Handler 和 Interceptor 集合
4) DispatchServlet 调用 HandlerAdapter 适配器
5) HandlerAdapter 调用具体的 Handler 处理业务
6) Handler 处理结束返回一个具体的 ModelAndView 给适配器，ModelAndView:model->数据模型，view->视图名称
7) 适配器将 ModelAndView 给 DispatchServlet
8) DispatchServlet 把视图名称给 ViewResolver 视图解析器
9) ViewResolver 返回一个具体的视图给 DispatchServlet
10) 渲染视图
11) 展示给用户

2. 画图解析



3.1.11 SpringMvc 的配置

```
<!-- 注解驱动 -->
<mvc:annotation-driven />
<!-- 扫描controller -->
<context:component-scan base-package="cn.itcast.usermanager.controller" />
<!-- 视图解析器-->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views"></property>
    <property name="suffix" value=".jsp" />
</bean>
<mvc:default-servlet-handler />
```

3.1.12 Spring MVC 启动流程

参考：

SpringMVC 启动过程详解（li）

3.1.13 Spring 的单例实现原理

参考：
Spring 的单例模式底层实现

3.1.14 Spring 框架中用到了哪些设计模式

Spring 框架中使用到了大量的设计模式，下面列举了比较有代表性的：

1. 代理模式—在 AOP 和 remoting 中被用的比较多。
2. 单例模式—在 spring 配置文件中定义的 bean 默认为单例模式。
3. 模板方法—用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。
4. 工厂模式—BeanFactory 用来创建对象的实例。
5. 适配器 - spring aop 装饰器 - spring data hashmapper
6. 观察者 spring 时间驱动模型
7. 回调 - Spring ResourceLoaderAware 回调接口
8. 前端控制器 - spring 用前端控制器 DispatcherServlet 对请求进行分发

3.1.15 Spring 其他产品（Srping Boot、Spring Cloud、Spring Secuirity、Spring Data、Spring AMQP 等）

参考：
说一说 Spring 家族

3.2 Netty

3.2.1 为什么选择 Netty

Netty 是业界最流行的 NIO 框架之一，它的健壮性、功能、性能、可定制性和可扩展性在同类框架中都是首屈一指的，它已经得到成百上千的商用项目验证，例如 Hadoop 的 RPC 框架 Avro 使用 Netty 作为通信框架。很多其它业界主流的 RPC 和分布式服务框架，也使用 Netty 来构建高性能的异步通信能力。

Netty 的优点总结如下：

- API 使用简单，开发门槛低；
- 功能强大，预置了多种编解码功能，支持多种主流协议；
- 定制能力强，可以通过 ChannelHandler 对通信框架进行灵活的扩展；
- 性能高，通过与其它业界主流的 NIO 框架对比，Netty 的综合性能最优；
- 社区活跃，版本迭代周期短，发现的 BUG 可以被及时修复，同时，更多的新功能会被

加入；

- 经历了大规模的商业应用考验，质量得到验证。在互联网、大数据、网络游戏、企业应用、电信软件等众多行业得到成功商用，证明了它完全满足不同行业的商用标准。

正是因为这些优点，Netty 逐渐成为 Java NIO 编程的首选框架。

3.2.2 说说业务中，Netty 的使用场景

有关“为何选择 Netty”的 11 个疑问及解答

3.2.3 原生的 NIO 在 JDK 1.7 版本存在 epoll bug

它会导致 Selector 空轮询，最终导致 CPU 100%。官方声称在 JDK1.6 版本的 update18 修复了该问题，但是直到 JDK1.7 版本该问题仍旧存在，只不过该 BUG 发生概率降低了一些而已，它并没有被根本解决。该 BUG 以及与该 BUG 相关的问题单可以参见以下链接内容。
http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6403933
http://bugs.java.com/bugdatabase/view_bug.do?bug_id=2147719

异常堆栈如下：

```
java.lang.Thread.State: RUNNABLE
    at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
    at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:210)
    at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:65)
    at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:69)
    - locked <0x0000000750928190> (a sun.nio.ch.Util$2)
    - locked <0x00000007509281a8> (a java.util.Collections$ UnmodifiableSet)
    - locked <0x0000000750946098> (a sun.nio.ch.EPollSelectorImpl)
    at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:80)
    at net.spy.memcached.MemcachedConnection.handleIO(Memcached Connection.java:217)
    at net.spy.memcached.MemcachedConnection.run(MemcachedConnection. java:836)
```

3.2.4 什么是 TCP 粘包/拆包

参考：
TCP 粘包，拆包及解决方法

3.2.5TCP 粘包/拆包的解决办法

参考：

TCP 粘包，拆包及解决方法

3.2.6 Netty 线程模型

参考：
Netty4 实战第十五章：选择正确的线程模型

3.2.7 说说 Netty 的零拷贝

参考：
理解 Netty 中的零拷贝（ZeroCopy）机制

3.2.8 Netty 内部执行流程

参考：
Netty：数据处理流程

3.2.9 Netty 重连实现

参考：
Netty Client 重连实现

四、微服务篇

4.1 微服务

4.1.1 前后端分离是如何做的

参考：
实现前后端分离的心得

4.1.2 微服务哪些框架

Spring Cloud、Dubbo、Hsf 等。

4.1.3 你怎么理解 RPC 框架

RPC 的目的是让你在本地调用远程的方法，而对你来说这个调用是透明的，你并不知道这个调用的方法是部署哪里。通过 RPC 能解耦服务，这才是使用 RPC 的真正目的。

4.1.4 说说 RPC 的实现原理

参考：
你应该知道的 RPC 原理
从零开始实现 RPC 框架 RPC 原理及实现

4.1.5 说说 Dubbo 的实现原理

dubbo 提供功能来讲，提供基础功能 RPC 调用，提供增值功能 SOA 服务治理。
dubbo 启动时查找可用的远程服务提供者，调用接口时不是最终调用本地实现，而是通过拦截调用（又用上 JDK 动态代理功能）过程经过一系列的的序列化、远程通信、协议解析最终调用到远程服务提供者
参考：
Dubbo 解析及原理浅析

4.1.6 你怎么理解 RESTful

REST 是 一种软件架构风格、设计风格，它是一种面向资源的网络化超媒体应用的架构风格。它主要是用于构建轻量级的、可维护的、可伸缩的 Web 服务。基于 REST 的服务被称为 RESTful 服务。REST 不依赖于任何协议，但是几乎每个 RESTful 服务使用 HTTP 作为底层协议，RESTful 使用 http method 标识操作，例如：
http://127.0.0.1/user/1 GET 根据用户 id 查询用户数据
http://127.0.0.1/user POST 新增用户
http://127.0.0.1/user PUT 修改用户信息
http://127.0.0.1/user DELETE 删除用户信息

http方法	资源操作	幂等	安全
GET	SELECT	是	是
POST	INSERT	否	否
PUT	UPDATE	是	否
DELETE	DELETE	是	否

幂等性：对同一REST接口的多次访问，得到的资源状态是相同的。

安全性：对该REST接口访问，不会使服务器端资源的状态发生改变。

4.1.7 说说如何设计一个良好的 API

参考：
如何设计一个良好的 API?

4.1.8 如何理解 RESTful API 的幂等性

参考：
如何理解 RESTful 的幂等性

4.1.9 如何保证接口的幂等性

参考：
后端接口的幂等性

4.1.10 说说 CAP 定理、BASE 理论

参考：
CAP 原理和 BASE 思想

4.1.11 怎么考虑数据一致性问题

参考：
分布式系统事务一致性解决方案

4.1.12 说说最终一致性的实现方案

可以结合 MQ 实现最终一致性，例如电商系统，把生成订单数据的写操作逻辑通过事务控制，一些无关紧要的业务例如日志处理，通知，通过异步消息处理，最终到请求落地。
参考：
系统分布式情况下最终一致性方案梳理

4.1.13 你怎么看待微服务

- 小：微服务体积小
- 独：能够独立的部署和运行。
- 轻：使用轻量级的通信机制和架构。
- 松：为服务之间是松耦合的。

4.1.14 微服务与 SOA 的区别

可以把微服务当做去除了 ESB 的 SOA。ESB 是 SOA 架构中的中心总线，设计图形应该是星形的，而微服务是去中心化的分布式软件架构。
参考：
SOA 与 微服务的区别

4.1.15 如何拆分服务

参考：
微服务架构(二): 如何把应用分解成多个服务

4.1.16 微服务如何进行数据库管理

参考：
在微服务中如何管理数据

4.1.17 如何应对微服务的链式调用异常

参考：
踢开绊脚石：微服务难点之服务调用的解决方案

4.1.18 对于快速追踪与定位问题

参考：

微服务架构下，如何实现分布式跟踪？

4.1.19 微服务的安全

参考：
论微服务安全

4.2 分布式

4.2.1 谈谈业务中使用分布式的场景

1. 解决 java 集群的 session 共享的解决方案：
   1) 客户端 cookie 加密。(一般用于内网中企业级的系统中，要求用户浏览器端的 cookie 不能禁用，禁用的话，该方案会失效）。
   2) 集群中，各个应用服务器提供了 session 复制的功能，tomcat 和 jboss 都实现了这样的功能。特点：性能随着服务器增加急剧下降，容易引起广播风暴；session 数据需要序列化，影响性能。
   3) session 的持久化，使用数据库来保存 session。就算服务器宕机也没事儿，数据库中的 session 照样存在。特点：每次请求 session 都要读写数据库，会带来性能开销。使用内存数据库，会提高性能，但是宕机会丢失数据(像支付宝的宕机，有同城灾备、异地灾备)。
   4) 使用共享存储来保存 session。和数据库类似，就算宕机了也没有事儿。其实就是专门搞一台服务器，全部对 session 落地。特点：频繁的进行序列化和反序列化会影响性能。
   5) 使用 memcached 来保存 session。本质上是内存数据库的解决方案。特点：存入 memcached 的数据需要序列化，效率极低。
2. 分布式事务的解决方案:
   1) TCC 解决方案：try confirm cancel。
   参考：
   为什么说传统分布式事务不再适用于微服务架构？

4.2.2 Session 分布式方案

1．客户端 cookie 加密。(一般用于内网中企业级的系统中，要求用户浏览器端的 cookie 不能 禁用，禁用的话，该方案会失效）。

2．集群中，各个应用服务器提供了 session 复制的功能，tomcat 和 jboss 都实现了这样的功能。特点：性能随着服务器增加急剧下降，容易引起广播风暴；session 数据需要序列化，影响性能。

3．session 的持久化，使用数据库来保存 session。就算服务器宕机也没事儿，数据库中的 session 照样存在。特点：每次请求 session 都要读写数据库，会带来性能开销。使用内存数据库，会提高性能，但是宕机会丢失数据(像支付宝的宕机，有同城灾备、异地灾备)。

4．使用共享存储来保存 session。和数据库类似，就算宕机了也没有事儿。其实就是专门搞一台服务器，全部对 session 落地。特点：频繁的进行序列化和反序列化会影响性能。

5．使用 memcached 来保存 session。本质上是内存数据库的解决方案。特点：存入 memcached 的数据需要序列化，效率极低。

4.2.3 分布式锁的场景

比如交易系统的金额修改，同一时间只能又一个人操作，比如秒杀场景，同一时间只能一个用户抢到，比如火车站抢票等等。

4.2.4 分布式锁的实现方案

1. 基于数据库实现分布式锁
2. 基于缓存实现分布式锁
3. 基于 Zookeeper 实现分布式锁

参考：

分布式锁的多种实现方式

4.2.5 分布式事务

参考：

深入理解分布式事务,高并发下分布式事务的解决方案

4.2.6 集群与负载均衡的算法与实现

参考：

负载均衡算法及手段

4.2.7 说说分库与分表设计

参考：

分表与分库使用场景以及设计方式

4.2.8 分库与分表带来的分布式困境与应对之策

参考：

服务端指南 数据存储篇 | MySQL（09） 分库与分表带来的分布式困境与应对之策

五、安全&性能

5.1 安全问题

5.1.1 安全要素与 STRIDE 威胁

参考：
http://blog.720ui.com/2017/security_stride/

5.1.2 防范常见的 Web 攻击

XSS 攻击

- 跨站脚本攻击;
- 是什么：攻击者向有 XSS 漏洞的网站中输入恶意的 HTML 代码，当其浏览器浏览该网站时，这段 HTML 代码会自 动执行。（理论上所有可以输入的地方没有对输入的数据进行处理，都会存在 XSS 攻击）;
- 危害： 盗取用户 cookie，破坏页面结构，重定向到其他网站;
- 防御：对用户输入的信息进行处理，只允许合法的值;

CSRF 攻击

- 跨站请求伪造
- 是什么：攻击者盗用了你的身份，以你的名义发送恶意请求;
- 危害：以你的名义发送邮件，盗取帐号，购买东西等;
- 原理： 首先个登录某网站，并在本地生成 cookie;然后在不登出的情况下，访问危害网站。
- 防御： 可以从服务端和客户端两方面进行考虑。但是在服务端的效果好。
    a. 随机的 cookie
    b. 添加验证码
    c. 不同的表单包含一个不同的伪随机值
- 注意：如果用户在一个站点上同时打开了两个不同的表单。CSRF 保护措施不应该影响到他对任何表单的提交

SQL 注入

- 是什么：通过 sql 命令伪装成正常的 http 请求参数，传递到服务端，服务器执行 sql 命令造成对数据库进行攻击
- 原理：sql 语句伪造参数，然后在对参数机型拼接后形成破坏性的 sql 语句，最后导致数据库收到攻击
- 防御：
    a. 对参数进行转义
    b. 数据库中的密码不应明文存储，可以对密码使用 md5 进行加密。

DDOS 攻击（分布式拒绝服务攻击）

- 是什么：简单来说就是 ifasong 大量的请求使服务器瘫痪。
- 被攻击的原因：服务器带宽不足，不能挡住攻击者的攻击流量。
- 防御：
    a. 最直接的方法就是增加带宽;
    b. 使用硬件防火墙;
    c. 优化资源使用提高 web server 的负载能力

来源：
https://blog.csdn.net/wen_special/article/details/80461394

5.1.3 服务端通信安全攻防

HTTPS 原理剖析
参考：
https://www.cnblogs.com/zery/p/5164795.html
HTTPS 降级攻击
参考：
http://blog.720ui.com/2016/security_https_tls/

5.1.4 授权与认证

基于角色的访问控制
参考：
https://blog.csdn.net/yin767833376/article/details/64907383
基于数据的访问控制

基于角色的访问控制，只验证访问数据的角色，但是没有对角色内的用户做细分。举个例子，用户甲与用户乙都具有用一个角色，但是如果只建立基于角色的访问控制，那么用户甲可以对用户乙的数据进行任意操作，从而发生了越权访问。因此，在业务场景中仅仅使用基于角色的访问控制是不够的，还需要引入基于数据的访问控制。如果将基于角色的访问控制视为一种垂直权限控制，那么，基于数据的访问控制就是一种水平权限控制。在业务场景中，往往对基于数据的访问控制不够重视，举个例子，评论功能是一个非常常见的功能，用户可以在客户端发起评论，回复评论，查看评论，删除评论等操作。一般情况下，只有本人才可以删除自己的评论，如果此时，业务层面没有建立数据的访问控制，那么用户甲可以试图绕过客户端，通过调用服务端 RESTful API 接口，猜测评论 ID 并修改评论 ID 就可以删除别人的评论。事实上，这是非常严重的越权操作。除此之外，用户之间往往也存在一些私有的数据，而这些私有的数据在正常情况下，只有用户自己才能访问。

基于数据的访问控制，需要业务层面去处理，但是这个也是最为经常遗落的安全点，需要引 起重视。这里，再次使用删除评论的案例，通过 Java 语言进行介绍。在这个案例中，核心 的代码片段在于，判断当前用户是否是评论的创建者，如果是则通过，不是则报出没有权限的错误码。那么，这样就可以很好地防止数据的越权操作。

```
@RestController
@RequestMapping(value = {"/v1/c/apps"})
public class AppCommentController{
    @Autowired
    private AppCommentService appCommentService;

    @RequestMapping(value = "/{appId:\\d+}/comments/{commentId:\\d+}", method = RequestMethod.DELETE)
    public void deleteAppCommentInfo(@PathVariable Long appId, @PathVariable Long commentId,
        @AuthenticationPrincipal UserInfo userInfo) {

        AppComment appComment = this.appCommentService.checkCommentInfo(commentId);

        // 判断当前用户是否是评论的创建者，如果是则通过，不是则报出没有权限的错误码。
        if(!appComment.getUserId().equals(Long.valueOf(userInfo.getUserId())) ){
            throw new BusinessException(ErrorCode.ACCESS_DENIED);
        }
        this.appCommentService.delete(commentId);

    }
}
```

总结下，基于角色的访问控制是一种垂直权限控制，通过建立用户与角色的对应关系，使得不同角色之间具有高低之分。用户根据拥有的角色进行操作与资源访问。基于数据的访问控制是一种水平权限控制，它对角色内的用户做细分，确保用户的数据不能越权操作。基于数据的访问控制，需要业务层面去处理，但是这个也是最为经常遗落的安全点，需要引起重视。

5.2 性能优化

性能指标有哪些
如何发现性能瓶颈
性能调优的常见手段
说说你在项目中如何进行性能调优

六、工程篇

6.1 需求分析

你如何对需求原型进行理解和拆分
说说你对功能性需求的理解
说说你对非功能性需求的理解
你针对产品提出哪些交互和改进意见
你如何理解用户痛点

6.2 设计能力

说说你在项目中使用过的 UML 图
你如何考虑组件化
你如何考虑服务化
你如何进行领域建模
你如何划分领域边界
说说你项目中的领域建模
说说概要设计

6.3 设计模式

你项目中有使用哪些设计模式
说说常用开源框架中设计模式使用分析
说说你对设计原则的理解 23 种设计模式的设计理念
设计模式之间的异同，例如策略模式与状态模式的区别
设计模式之间的结合，例如策略模式+简单工厂模式的实践
设计模式的性能，例如单例模式哪种性能更好。

6.4 业务工程

你系统中的前后端分离是如何做的
说说你的开发流程
你和团队是如何沟通的
你如何进行代码评审
说说你对技术与业务的理解
说说你在项目中经常遇到的 Exception

说说你在项目中遇到感觉最难 Bug，怎么解决的
说说你在项目中遇到印象最深困难，怎么解决的
你觉得你们项目还有哪些不足的地方
你是否遇到过 CPU 100% ，如何排查与解决
你是否遇到过 内存 OOM ，如何排查与解决
说说你对敏捷开发的实践
说说你对开发运维的实践
介绍下工作中的一个对自己最有价值的项目，以及在这个过程中的角色

6.5 软实力

说说你的亮点
说说你最近在看什么书
说说你觉得最有意义的技术书籍
工作之余做什么事情
说说个人发展方向方面的思考
说说你认为的服务端开发工程师应该具备哪些能力
说说你认为的架构师是什么样的，架构师主要做什么
说说你所理解的技术专家