1. **重载和重构的区别**
2. 重写：

①重写是子类对父类允许访问的方法的实现过程的重新编写，方法名，返回值，形参都不能改变，【即外壳不变，核心重写】

②重写的意义在于子类可以根据需要，定制自己的行为

③重写方法不能抛出新的检查异常或者比被重写方法申明更加宽泛的异常

④注意

· 声明为final的方法不能被重写。

· 声明为static的方法不能被重写，但是能够被再次声明。

· 子类和父类在同一个包中，那么子类可以重写父类所有方法，除了声明为private和final的方法。

· 子类和父类不在同一个包中，那么子类只能够重写父类的声明为public和protected的非final方法。

· 访问权限不能比父类中被重写的方法的访问权限更低

· 构造方法不能被重写。

· 如果不能继承一个方法，则不能重写这个方法。即重写只发生在继承的基础上

1. 重载

①重载是发生在一个类之中，方法名相同，参数不同，返回值类型可相同也可不同

②每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

③只能重载构造函数

④注意

* 被重载的方法必须改变参数列表(参数个数或类型或顺序不一样)；
* 被重载的方法可以改变返回类型；
* 被重载的方法可以改变访问修饰符；
* 被重载的方法可以声明新的或更广的检查异常；
* 方法能够在同一个类中或者在一个子类中被重载。
* 无法以返回值类型作为重载函数的区分标准

1. **String ，Stringbuffer ，Stringbuilder 区别**
2. String含义为引用数据类型，是字符串常量，是不可变对象，String是线程安全的，原因是string内部存储字符串的char数组以及和char数组相关的信息都是final的，这就保证了string对象生成的那一刻他在内存里就是不可变的。
3. StringBuffer  线程安全的可变字符序列:对StringBuffer对象本身进行操作,而不是生成新的对象.所所以在改变对象引用条件下,一般推荐使用StringBuffer.同时主要是使用append和insert方法,
4. StringBuilder 线程不安全的可变字符序列.提供一个与StringBuffer兼容的API,但不同步.设计作为StringBuffer的一个简易替换,用在字符缓冲区被单个线程使用的时候.效率比StringBuffer更快
5. 区别
   1. 执行速度：StringBuilder > StringBuffer > String
   2. String试用于少量字符串操作
      1. StringBuilder适用于单线程下字符缓冲区下进行大量操作的情况
      2. StringBuffer适用于多线程下在字符缓冲区进行大量操作的情况

**3.本类静态代码块，初始代码块和父类静态代码块，初始代码块的执行顺序**

1. 先父类再子类
2. 如果子类有静态成员变量和静态代码块，则执行完父类的静态成员变量和静态代码块后，接着执行子类的静态变量和静态代码块， 否则直接按照父类的变量->代码块->构造函数，再执行子类的变量->代码块->构造函数
3. 需要注意的是子类的静态变量和静态代码块是优先于父类的普通成员变量和代码块以及构造函数的。
4. 这也说明了先静态->再普通

**4.对象的创建过程，类加载过程 类对象的获取**

1. 对象创建的过程：
   1. 对象创建首先会在堆内存中开辟一块空间，并分配一个地址
   2. 将对象的所有非静态成员加载到开辟的空间下，并为所有非静态成员变量进行默认初始化。
   3. 调用构造函数进栈执行，在执行构造函数的时候，首先会执行构造函数的隐式三步，再执行构造函数中的代码。

隐式三步：

1. 执行super语句，调用父类的构造函数
2. 对所有非静态成员变量进行显示赋值。
3. 执行构造代码块；（非静态成员变量显示赋值和执行构造代码块无先后顺序，由代码顺序决定，谁在前先执行谁）；
4. 执行完隐式三步后，再执行构造函数中的代码，代码执行完后弹栈；
5. 将堆内存中开辟的空间地址赋值给一个引用对象，对象创建完成。
6. 类加载的过程：
   1. JVM会先去方法区中找有没有类对应的.class存在，如果有，就直接使用；如果没有，就把对应类的.class加载到方法区；
   2. 将.class加载到方法区的时候，分为两部分，首先将非静态内容加载到方法区的非静态区域内
   3. 再将静态内容加载到方法区的静态区域内，加载完成后，对所有的静态成员变量进行默认初始化
   4. 所有静态成员变量默认初始化后，进行显示赋值；再执行静态代码块；（执行静态代码块和静态成员变量显示赋值无先后顺序，与代码顺序有关，谁在前面先执行谁）
   5. 如果有继承关系，加载子类的时候会先加载父类，在加载子类。
7. java类对象的获取

Java中通过class类获取class对象的方法详解

①通过object类的getObject（）方法

Person p = new Person();

Class c = p.getObject();

②通过类名.class获取到字节码文件对象（任意数据类型都具备一个class静态属性,看上去要比第一种方式简单）

Class c2 = Person.class;

③通过class类中的方法（将类名作为字符串传递给Class类中的静态方法forName即可）

Class c3 = Class.forName(“Person”)

注意：第三种和前两种的区别

前两种你必须明确Person类型.后面是指定这种类型的字符串就行.这种扩展更强.我不需要知道类名.我只提供字符串,按照配置文件加载就可以了

**6.HashMap ,HashTable区别**

HashMap是Hashtable的轻量级实现（非线程安全的实现），他们都完成了Map接口。主要的区别有：线程安全性，同步(synchronization)，以及速度。

1.Hashtable继承自Dictionary类，而HashMap是Java1.2引进的Map interface的一个实现。

2.HashMap允许将null作为一个entry的key或者value，而Hashtable不允许。

3.HashMap是非synchronized，而Hashtable是synchronized，这意味着Hashtable是线程安全的，多个线程可以共享一个Hashtable；而如果没有正确的同步的话，多个线程是不能共享HashMap的。Java 5提供了ConcurrentHashMap，它是HashTable的替代，比HashTable的扩展性更好。（在多个线程访问Hashtable时，不需要自己为它的方法实现同步，而HashMap 就必须为之提供外同步(Collections.synchronizedMap)）

4.另一个区别是HashMap的迭代器(Iterator)是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。所以当有其它线程改变了HashMap的结构（增加或者移除元素），将会抛出ConcurrentModificationException，但迭代器本身的remove()方法移除元素则不会抛出ConcurrentModificationException异常。但这并不是一个一定发生的行为，要看JVM。这条同样也是Enumeration和Iterator的区别。

5.由于HashMap非线程安全，在只有一个线程访问的情况下，效率要高于HashTable。

6.HashMap把Hashtable的contains方法去掉了，改成containsvalue和containsKey。因为contains方法容易让人引起误解。

7.Hashtable中hash数组默认大小是11，增加的方式是 old\*2+1。HashMap中hash数组的默认大小是16，而且一定是2的指数。

8..两者通过hash值散列到hash表的算法不一样：，HashTbale是古老的除留余数法，直接使用hashcode

**7.ArrayList LinkedList区别**

ArrayList和LinkedList的大致区别如下:

1.ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。

2.对于随机访问get和set，ArrayList优于LinkedList，因为LinkedList要移动指针。

3.对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。

**8.单例，多例模式**

1.单例模式和多例模式说明：

(1)单例模式和多例模式属于对象模式。

(2)单例模式的对象在整个系统中只有一份，多例模式可以有多个实例。

(3)它们都不对外提供构造方法，即构造方法都为私有。

2.为什么用单例、多例：  
   之所以用单例，是因为没必要每个请求都新建一个对象，这样子既浪费CPU又浪费内存；  
  之所以用多例，是为了防止并发问题；即一个请求改变了对象的状态，此时对象又处理另一个请求，而之前请求对对象状态的改变导致了对象对另一个请求做了错误的处理；  
  用单例和多例的标准只有一个：  
  当对象含有可改变的状态时（更精确的说就是在实际应用中该状态会改变），则多例，否则单例；

3.何时用单例？何时用多例？  
   对于struts2来说，action必须用多例，因为action本身含有请求参数的值，即可改变的状态；  
  而对于STRUTS1来说，action则可用单例，因为请求参数的值是放在actionForm中，而非action中的；  
   另外要说一下，并不是说service或dao一定是单例，标准同第3点所讲的，就曾见过有的service中也包含了可改变的状态，同时执行方法也依赖该状态，但一样用的单例，这样就会出现隐藏的BUG,而并发的BUG通常很难重现和查找；

4.spring的Bean默认的是单例的，Bean的作用域可以通过Bean标签的scope属性进行设置，Bean的作用域包括：

默认情况下scope="singleton"，那么该Bean是单例，任何人获取该Bean实例的都为同一个实例；

scope="prototype"，任何一个实例都是新的实例；

scope="request"，在WEB应用程序中，每一个实例的作用域都为request范围；

scope="session",在WEB应用程序中，每一个实例的作用域都为session范围；

1. **懒汉式和饿汉式的区别**

**1、懒汉式**

私有化构造器

创建实例并将其设置为私有的，静态的，赋值为null

通过公共的方法(公**共的，静态的)获取，在公共方法中进行new对象，并用if进行判断**

**2、饿汉式**

私有化构造器

创建实例并将其设置为私有的、静态的，并通过new关键字创建实例对象

通过公共的方法（公共的、静态的）获取，在方法中直接将创建实例对**象的引用进行返回**

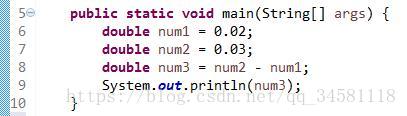
**3、懒汉式和饿汉式的区分**

饿：上来就创建好实例对象，直接返回

懒：用的时候再在方法里创建，“但懒汉式”存在线程安全问题。

所以建议使用饿汉模式设计“单例模式”的程序

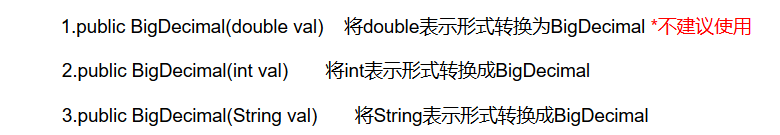
**10.BigDecimal的设计以及使用**



输出结果0.009999999999999998!

由于float和double都是浮点数, 都有取值范围, 都有精度范围. 浮点数与通常使用的小数不同, 使用中, 往往难以确定. 常见的问题是定义了一个浮点数, 经过一系列的计算, 它本来应该等于某个确定值, 但实际上并不是! 商业计算必须是完全精确的计算, 故不能使用double或者float, 而应该采用java.math.BigDecimal.

## 1.BigDecimal构造方法





**当**double**必须用作**BigDecimal**的源时，**请使用Double.toString(double)转成String，然后使用String构造方法，或使用BigDecimal的静态方法valueOf。

**2.mysql数据库设计**

BigDecimal在进行入库时, 数据库选择decimal类型, 长度可以自定义, 如18; 小数点我们项目中用的是2, 保留2位小数. 此外还要注意的就是默认值, 一定写成0.00, 不要用默认的NULL, 否则在进行加减排序等操作时, 会带来转换的麻烦!

**3.总结**

　(1)商业计算使用BigDecimal。

     (2)尽量使用参数类型为String的构造函数。

     (3) BigDecimal都是不可变的（immutable）的，在进行每一步运算时，都会产生一个新的对象，所以在做加减乘除运算时千万要保存操作后的值。

        (4)我们往往容易忽略JDK底层的一些实现细节，导致出现错误，需要多加注意。

**web部分**

1. **cookie session区别**

1、cookie数据存放在客户的浏览器上，session数据放在服务器上。

2、cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗  
   考虑到安全应当使用session。

3、session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能  
   考虑到减轻服务器性能方面，应当使用COOKIE。

4、单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。

5、所以个人建议：  
   将登陆信息等重要信息存放为SESSION  
   其他信息如果需要保留，可以放在COOKIE中

二、get和post的区别

get和post是HTTP与服务器交互的方式，

说到方式，其实总共有四种：put，delete，post，get。

他们的作用分别是对服务器资源的增，删，改，查。

所以，get是获取数据，post是修改数据。

但是，现在大家都不这么干了！

只用一个方式就可以做增删查减的操作。

。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。

区别分析

1.get把请求的数据放在url上，即HTTP协议头上，其格式为：

以?分割URL和传输数据，参数之间以&相连。

数据如果是英文字母/数字，原样发送，

如果是空格，转换为+，

如果是中文/其他字符，则直接把字符串用BASE64加密，及“%”加上“字符串的16进制ASCII码”。

post把数据放在HTTP的包体内（requrest body）。

2.get提交的数据最大是2k（原则上url长度无限制，那么get提交的数据也没有限制咯？限制实际上取决于浏览器，(大多数)浏览器通常都会限制url长度在2K个字节，即使(大多数)服务器最多处理64K大小的url。也没有卵用。）。

post理论上没有限制。实际上IIS4中最大量为80KB，IIS5中为100KB。

3.GET在浏览器回退时是无害的，POST会再次提交请求。

4.GET产生的URL地址可以被Bookmark，而POST不可以。

5.GET请求会被浏览器主动cache，而POST不会，除非手动设置。

6.GET请求只能进行url编码，而POST支持多种编码方式。

7.GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。

8.GET只接受ASCII字符的参数的数据类型，而POST没有限制

那么，post那么好为什么还用get？get效率高！。

9.GET产生一个TCP数据包，浏览器会把http header和data一并发送出去，服务器响应200(返回数据);

POST产生两个TCP数据包，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok(返回数据)。

扩展

因为POST需要两步，时间上消耗的要多一点，看起来GET比POST更有效。因此Yahoo团队有推荐用GET替换POST来优化网站性能。但这是一个坑！跳入需谨慎。为什么？

1. GET与POST都有自己的语义，不能随便混用。

2. 据研究，在网络环境好的情况下，发一次包的时间和发两次包的时间差别基本可以无视。而在网络环境差的情况下，两次包的TCP在验证数据包完整性上，有非常大的优点。

3. 并不是所有浏览器都会在POST中发送两次包，Firefox就只发送一次。

1. **http1.0和1.1区别**

**1、HTTP 1.1支持长连接（PersistentConnection）和请求的流水线（Pipelining）处理**

**HTTP 1.0规定浏览器与服务器只保持短暂的连接，浏览器的每次请求都需要与服务器建立一个TCP连接，服务器完成请求处理后立即断开TCP连接，服务器不跟踪每个客户也不记录过去的请求。**

HTTP 1.1则支持持久连接Persistent Connection, 并且默认使用persistent  connection. 在同一个tcp的连接中可以传送多个HTTP请求和响应. 多个请求和响应可以重叠，多个请求和响应可以同时进行. 更加多的请求头和响应头(比如HTTP1.0没有host的字段).

在1.0时的会话方式：  
 1. 建立连接  
 2. 发出请求信息  
 3. 回送响应信息  
 4. 关掉连接

**HTTP 1.1的持续连接，也需要增加新的请求头来帮助实现，例如，Connection请求头的值为Keep-Alive时，客户端通知服务器返回本次请求结果后保持连接；Connection请求头的值为close时，客户端通知服务器返回本次请求结果后关闭连接。HTTP 1.1还提供了与身份认证、状态管理和Cache缓存等机制相关的请求头和响应头。**

请求的流水线（Pipelining）处理，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟。例如：一个包含有许多图像的网页文件的多个请求和应答可以在一个连接中传输，但每个单独的网页文件的请求和应答仍然需要使用各自的连接。  HTTP 1.1还允许客户端不用等待上一次请求结果返回，就可以发出下一次请求，但服务器端必须按照接收到客户端请求的先后顺序依次回送响应结果，以保证客户端能够区分出每次请求的响应内容。

**2.HTTP 1.1增加host字段**

在HTTP1.0中认为每台服务器都绑定一个唯一的IP地址，因此，请求消息中的URL并没有传递主机名（hostname）。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个IP地址。

 HTTP1.1的请求消息和响应消息都应支持Host头域，且请求消息中如果没有Host头域会报告一个错误（400 Bad Request）。此外，服务器应该接受以绝对路径标记的资源请求。

**3、100(Continue) Status(节约带宽)**

HTTP/1.1加入了一个新的状态码100（Continue）。客户端事先发送一个只带头域的请求，如果服务器因为权限拒绝了请求，就回送响应码401（Unauthorized）；如果服务器接收此请求就回送响应码100，客户端就可以继续发送带实体的完整请求了。100 (Continue) 状态代码的使用，允许客户端在发request消息body之前先用request header试探一下server，看server要不要接收request body，再决定要不要发request body。

**4**、HTTP/1.1中引入了Chunked transfer-coding来解决上面这个问题，发送方将消息分割成若干个任意大小的数据块，每个数据块在发送时都会附上块的长度，最后用一个零长度的块作为消息结束的标志。这种方法允许发送方只缓冲消息的一个片段，避免缓冲整个消息带来的过载。

**5**、HTTP/1.1在1.0的基础上加入了一些cache的新特性，当缓存对象的Age超过Expire时变为stale对象，cache不需要直接抛弃stale对象，而是与源服务器进行重新激活（revalidation）。

1. **rest软件架构**

**rest**是一种软件架构风格。它是一种针对网络应用的设计和开发方式。可以降低开发的复杂性，提高系统的可伸缩性。

**实现REST将是基于HTTP应用协议，例如**put，delete，post，get：

REST能实现是一种解耦方法，让我们实现这些架构特性：性能，伸缩性，简化，可修改性，扩展性

我们可以发现REST其实是一种组织Web服务的架构，而并不是我们想象的那样是实现Web服务的一种新的技术，更没有要求一定要使用HTTP。其目标是为了创建具有良好扩展性的分布式系统。

　　反过来，作为一种架构，其提出了一系列架构级约束。这些约束有：

（1）使用客户/服务器模型。客户和服务器之间通过一个统一的接口来互相通讯。

（2）层次化的系统。在一个REST系统中，客户端并不会固定地与一个服务器打交道。

（3）无状态。在一个REST系统中，服务端并不会保存有关客户的任何状态。也就是说，客户端自身负责用户状态的维持，并在每次发送请求时都需要提供足够的信息。

（4）可缓存。REST系统需要能够恰当地缓存请求，以尽量减少服务端和客户端之间的信息传输，以提高性能。

（5）统一的接口。一个REST系统需要使用一个统一的接口来完成子系统之间以及服务与用户之间的交互。这使得REST系统中的各个子系统可以独自完成演化。

　在HTTP中，一个URL主要由以下几个部分组成：

1.协议。即HTTP以及HTTPS。

2.主机名和端口。如www.egoods.com:8421

3.资源的相对路径。如/api/categories。

4.请求参数。即由问号开始的由键值对组成的字符串：?page=1&page\_size=20

　　在为一个资源设计其所对应的URL时，我们需要着重考虑第三部分和第四部分组成

1. **网络传输七层协议**



1. **request method**



**数据库部分**

1. **事务的特性，查询可能出现的几种情况，数据库隔离级别，锁**

概念

数据库事务(Database Transaction) ，是指作为单个逻辑工作单元执行的一系列读或写操作。其目的是提供三种方法：失败恢复到正常状态的方法、保持一致性的方法、并发操作之间保持隔离的方法，通俗的说是那一系列操作要么完全地执行，要么完全地不执行。

特性

A：原子性(Actomicity)，该系列操作作为一个整体，不能被分割，要么全执行，要么都不执行；

C：一致性(Consistency)，确保数据从一种状态变为另一种状态；

I：隔离性(Isolation)，多个事务并发执行时，彼此之间互不影响；

D：持久性(Durability)，当事务提交成功后，对数据的修改要永久保存在数据库中。

并发异常

回滚丢失：由于一个事务的失败回滚导致另一个事务的更新丢失了；

覆盖丢失：更新被其他事务给覆盖了；

脏读：一个事务读取了另一个事务修改未提交的数据；

不可重复读：一个事务对同一行数据执行了两次或更多次查询，但是却得到了不同的结果；

幻读：与不可重复读相似，不过针对的是数据的数量，当一个事务查询记录两次或多次时，得到的数量不一致。

隔离级别

读未提交：一个事务的更新语句没有提交，但是别的事务可以读到这个改变；

读已提交：一个事务只能看到其他事务已经提交的更新、看不到未提交的更新，消除了脏读和回滚丢失，Oracle/Sqlserver就是此类；

可重复读：一个事务中进行两次或多次同样的对于数据内容的查询，得到的结果是一样的，但不保证对于数据条数的查询是一样的，只要存在读改行数据就禁止写，消除了不可重复读和覆盖丢失，如Mysql，需要注意的是虽然Mysql的隔离级别为可重复读，理论上是不能解决幻读异常的，但由于Next-key，Gap-lock的存在，使得Mysql可以避免幻读问题；

串行化：事务执行不允许其他事务操作，只要存在读就禁止写，但可以同时读，消除了幻读，但是效率太低。

### 锁模式

**1、共享锁**

共享锁（S 锁）允许并发事务在封闭式并发控制下读取 (SELECT)资源。有关详细信息，请参阅[并发控制的类型](http://msdn.microsoft.com/zh-cn/library/ms189132.aspx)（悲观锁和乐观锁）。资源上存在共享锁（S锁）时，任何其他事务都不能修改数据。读取操作一完成，就立即释放资源上的共享锁（S锁），除非将事务隔离级别设置为可重复读或更高级别，或者在事务持续时间内用锁定提示保留共享锁（S锁）。

**2、更新锁（U锁）**

更新锁在共享锁和排他锁的结合。更新锁意味着在做一个更新时，一个共享锁在扫描完成符合条件的数据后可能会转化成排他锁。

这里面有两个步骤：

1) 扫描获取Where条件时。这部分是一个更新查询，此时是一个更新锁。

2) 如果将执行写入更新。此时该锁升级到排他锁。否则，该锁转变成共享锁。

更新锁可以防止常见的死锁。

**3、排他锁**

排他锁（X 锁）可以防止并发事务对资源进行访问。排他锁不与其他任何锁兼容。使用排他锁（X锁）时，任何其他事务都无法修改数据；仅在使用 NOLOCK提示或未提交读隔离级别时才会进行读取操作。

**悲观锁**

悲观锁是指假设并发更新冲突会发生，所以不管冲突是否真的发生，都会使用锁机制。  
悲观锁会完成以下功能：锁住读取的记录，防止其它事务读取和更新这些记录。其它事务会一直阻塞，直到这个事务结束.  
悲观锁是在使用了数据库的事务隔离功能的基础上，独享占用的资源，以此保证读取数据一致性，避免修改丢失。

悲观锁可以使用Repeatable Read事务，它完全满足悲观锁的要求。

**乐观锁**

乐观锁不会锁住任何东西，也就是说，它不依赖数据库的事务机制，乐观锁完全是应用系统层面的东西。

**如果使用乐观锁，那么数据库就必须加版本字段，否则就只能比较所有字段，但因为浮点类型不能比较，所以实际上没有版本字段是不可行的。**

## ⑴ 原子性（Atomicity）

　　原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，这和前面两篇博客介绍事务的功能是一样的概念，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

## ⑵ 一致性（Consistency）

确保数据从一种状态变为另一种状态；

　　一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

　　拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

## ⑶ 隔离性（Isolation）

多个事务并发执行时，彼此之间互不影响；

　　隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

　　即要达到这么一种效果：对于任意两个并发的事务T1和T2，在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

## ⑷ 持久性（Durability）

当事务提交成功后，对数据的修改要永久保存在数据库中。

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

　　例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

以上介绍完事务的四大特性(简称ACID)，现在重点来说明下事务的隔离性，当多个线程都开启事务操作数据库中的数据时，数据库系统要能进行隔离操作，以保证各个线程获取数据的准确性，在介绍数据库提供的各种隔离级别之前，我们先看看如果不考虑事务的隔离性，会发生的几种问题：

## 1，脏读

　　脏读是指在一个事务处理过程里读取了另一个未提交的事务中的数据。

　　当一个事务正在多次修改某个数据，而在这个事务中这多次的修改都还未提交，这时一个并发的事务来访问该数据，就会造成两个事务得到的数据不一致。例如：用户A向用户B转账100元，对应SQL命令如下

update account set money=money+**100** where name=’B’; (此时A通知B)

update account set money=money - **100** where name=’A’;

　　当只执行第一条SQL时，A通知B查看账户，B发现确实钱已到账（此时即发生了脏读），而之后无论第二条SQL是否执行，只要该事务不提交，则所有操作都将回滚，那么当B以后再次查看账户时就会发现钱其实并没有转。

## 2，不可重复读

　　不可重复读是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。

　　例如事务T1在读取某一数据，而事务T2立马修改了这个数据并且提交事务给数据库，事务T1再次读取该数据就得到了不同的结果，发送了不可重复读。

　　不可重复读和脏读的区别是，脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

　　在某些情况下，不可重复读并不是问题，比如我们多次查询某个数据当然以最后查询得到的结果为主。但在另一些情况下就有可能发生问题，例如对于同一个数据A和B依次查询就可能不同，A和B就可能打起来了……

## 3，虚读(幻读)

　　幻读是事务非独立执行时发生的一种现象。例如事务T1对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务T2又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务T1的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务T2中添加的，就好像产生幻觉一样，这就是发生了幻读。

　　幻读和不可重复读都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

　　现在来看看MySQL数据库为我们提供的四种隔离级别：

　　① Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

　　② Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

　　③ Read committed (读已提交)：可避免脏读的发生。

　　④ Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

　　以上四种隔离级别最高的是Serializable级别，最低的是Read uncommitted级别，当然级别越高，执行效率就越低。像Serializable这样的级别，就是以锁表的方式(类似于Java多线程中的锁)使得其他的线程只能在锁外等待，所以平时选用何种隔离级别应该根据实际情况。在MySQL数据库中默认的隔离级别为Repeatable read (可重复读)。

　　在MySQL数据库中，支持上面四种隔离级别，默认的为Repeatable read (可重复读)；而在Oracle数据库中，只支持Serializable (串行化)级别和Read committed (读已提交)这两种级别，其中默认的为Read committed级别。

　　在MySQL数据库中查看当前事务的隔离级别：

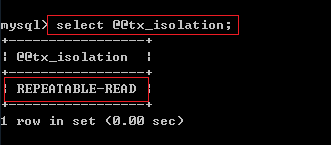
select **@@tx\_isolation**;

　　在MySQL数据库中设置事务的隔离 级别：

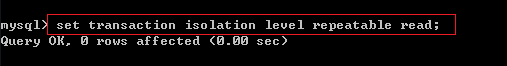
set [glogal | session] transaction isolation level 隔离级别名称;

set tx\_isolation=’隔离级别名称;’

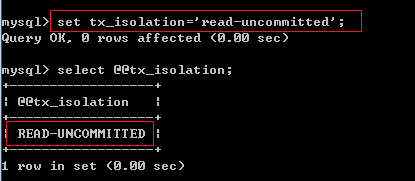
例1：查看当前事务的隔离级别：



例2：将事务的隔离级别设置为Read uncommitted级别：



或：

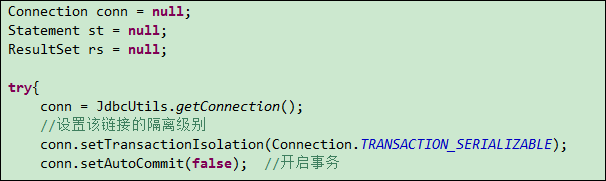


记住：设置数据库的隔离级别一定要是在开启事务之前！

如果是使用JDBC对数据库的事务设置隔离级别的话，也应该是在调用Connection对象的setAutoCommit(false)方法之前。调用Connection对象的setTransactionIsolation(level)即可设置当前链接的隔离级别，至于参数level，可以使用Connection对象的字段：



在JDBC中设置隔离级别的部分代码：

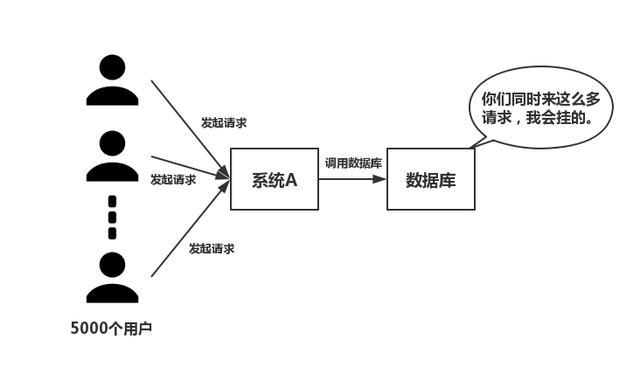


　　后记：隔离级别的设置只对当前链接有效。对于使用MySQL命令窗口而言，一个窗口就相当于一个链接，当前窗口设置的隔离级别只对当前窗口中的事务有效；对于JDBC操作数据库来说，一个Connection对象相当于一个链接，而对于Connection对象设置的隔离级别只对该Connection对象有效，与其他链接Connection对象无关。

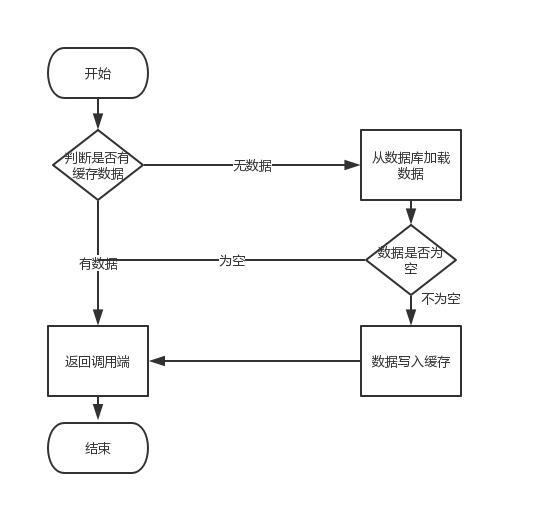
1. **redis实现缓存的时候，怎么保证mysql和redis数据一致**

**需求起因**

在[高并发](https://mb.yidianzixun.com/channel/w/%E9%AB%98%E5%B9%B6%E5%8F%91)的业务场景下，[数据库](https://mb.yidianzixun.com/channel/w/%E6%95%B0%E6%8D%AE%E5%BA%93)大多数情况都是用户并发访问最薄弱的环节。所以，就需要使用[redis](https://mb.yidianzixun.com/channel/w/redis)做一个缓冲操作，让请求先访问到redis，而不是直接访问[MySQL](https://mb.yidianzixun.com/channel/w/mysql)等数据库。



这个业务场景，主要是解决读数据从Redis缓存，一般都是按照下图的流程来进行业务操作。



读取缓存步骤一般没有什么问题，但是一旦涉及到数据更新：数据库和缓存更新，就容易出现**缓存(Redis)和数据库（MySQL）间的数据一致性问题**。

不管是先写[MySQL数据库](https://mb.yidianzixun.com/channel/w/mysql%E6%95%B0%E6%8D%AE%E5%BA%93)，再删除Redis缓存；还是先删除缓存，再写库，都有可能出现数据不一致的情况。举一个例子：

1.如果删除了缓存Redis，还没有来得及写库MySQL，另一个线程就来读取，发现缓存为空，则去数据库中读取数据写入缓存，此时缓存中为脏数据。

2.如果先写了库，在删除缓存前，写库的线程宕机了，没有删除掉缓存，则也会出现数据不一致情况。

因为写和读是并发的，没法保证顺序,就会出现缓存和数据库的数据不一致的问题。

如来解决？这里给出两个解决方案，先易后难，结合业务和技术代价选择使用。

缓存和数据库一致性解决方案

**1.第一种方案：采用延时双删策略**

在写库前后都进行redis.del(key)操作，并且设定合理的超时时间。

伪代码如下

public [void](https://mb.yidianzixun.com/channel/w/void) write(String key,[Object](https://mb.yidianzixun.com/channel/w/object) data){ redis.delKey(key); db.updateData(data); Thread.sleep(500); redis.delKey(key); }

**2.具体的步骤就是：**

1）先删除缓存

2）再写数据库

3）休眠500毫秒

4）再次删除缓存

**那么，这个500毫秒怎么确定的，具体该休眠多久呢？**

需要评估自己的项目的读数据业务逻辑的耗时。这么做的目的，就是确保读请求结束，写请求可以删除读请求造成的缓存脏数据。

当然这种策略还要考虑redis和数据库主从同步的耗时。最后的的写数据的休眠时间：则在读数据业务逻辑的耗时基础上，加几百ms即可。比如：休眠1秒。

**3.设置缓存过期时间**

从理论上来说，给缓存设置过期时间，是保证最终一致性的解决方案。所有的写操作以数据库为准，只要到达缓存过期时间，则后面的读请求自然会从数据库中读取新值然后回填缓存。

**4.该方案的弊端**

结合双删策略+缓存超时设置，这样最差的情况就是在超时时间内数据存在不一致，而且又增加了写请求的耗时。

**2、第二种方案：异步更新缓存(基于订阅binlog的同步机制)**

**1.技术整体思路：**

[MySQL binlog](https://mb.yidianzixun.com/channel/w/mysql binlog)增量订阅消费+消息队列+增量数据更新到redis

**1）读Redis**：热数据基本都在Redis

**2）写MySQL**:增删改都是操作MySQL

**3）更新Redis数据**：MySQ的数据操作binlog，来更新到Redis

**2.Redis更新**

**1）数据操作主要分为两大块：**

* 一个是全量(将全部数据一次写入到redis)
* 一个是增量（实时更新）

这里说的是增量,指的是mysql的update、insert、delate变更数据。

**2）读取binlog后分析 ，利用消息队列,推送更新各台的redis缓存数据。**

这样一旦MySQL中产生了新的写入、更新、删除等操作，就可以把binlog相关的消息推送至Redis，Redis再根据binlog中的记录，对Redis进行更新。

其实这种机制，很类似MySQL的主从备份机制，因为MySQL的主备也是通过binlog来实现的数据一致性。

这里可以结合使用canal(阿里的一款开源框架)，通过该框架可以对MySQL的binlog进行订阅，而canal正是模仿了[mysql](https://mb.yidianzixun.com/channel/w/mysql)的slave[数据库](https://mb.yidianzixun.com/channel/w/%E6%95%B0%E6%8D%AE%E5%BA%93)的备份请求，使得Redis的数据更新达到了相同的效果。

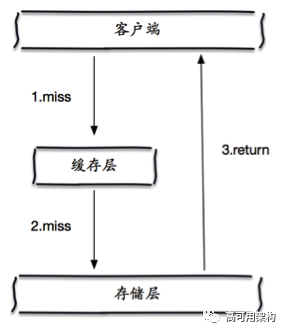
当然，这里的消息推送工具你也可以采用别的第三方：[kafka](https://mb.yidianzixun.com/channel/w/kafka)、[rabbitMQ](https://mb.yidianzixun.com/channel/w/rabbitmq)等来实现推送更新[Redis](https://mb.yidianzixun.com/channel/w/redis)

**Redis**

1. **redis穿透和血崩**

1.什么是缓存穿透

一般的缓存系统，都是按照key值去缓存查询，如果不存在对应的value，就应该去DB中查找 。这个时候，如果请求的并发量很大，就会对后端的DB系统造成很大的压力。这就叫做缓存穿透。关键词：缓存value为空；并发量很大去访问DB。



造成的原因

1.业务自身代码或数据出现问题；2.一些恶意攻击、爬虫造成大量空的命中，此时会对数据库造成很大压力。

解决方法

1.设置布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，

从避免了对底层存储系统的查询压力。

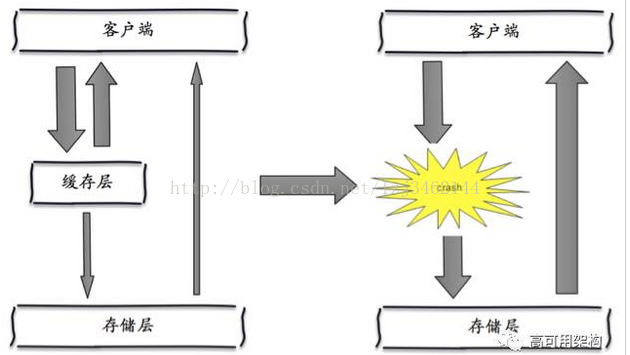
2. 如果一个查询返回的数据为空，不管是数据不存在还是系统故障，我们仍然把这个结果进行缓存，但是它的过期时间会很短

最长不超过5分钟。

二、雪崩

1.什么是雪崩

因为缓存层承载了大量的请求，有效的保护了存储 层，但是如果缓存由于某些原因，整体不能够提供服务，于是所有的请求，就会到达存储层，存储层的调用量就会暴增，造成存储层也会挂掉的情况。缓存雪崩的英文解释是奔逃的野牛，指的是缓存层当掉之后，并发流量会像奔腾的野牛一样，大量后端存储。



存在这种问题的一个场景是：当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，大量数据会去直接访问DB，此时给DB很大的压力。

2.解决方法

（1）设置redis集群和DB集群的高可用，如果redis出现宕机情况，可以立即由别的机器顶替上来。这样可以防止一部分的风险。

（2）使用互斥锁

在缓存失效后，通过加锁或者队列来控制读和写数据库的线程数量。比如：对某个key只允许一个线程查询数据和写缓存，其他线程等待。单机的话，可以使用synchronized或者lock来解决，如果是分布式环境，可以是用redis的setnx命令来解决。

（3）不同的key,可以设置不同的过期时间，让缓存失效的时间点不一致，尽量达到平均分布。

（4）永远不过期

redis中设置永久不过期，这样就保证了，不会出现热点问题，也就是物理上不过期。

（5）资源保护

使用netflix的hystrix，可以做各种资源的线程池隔离，从而保护主线程池。

3.使用

四种方案，没有最佳只有最合适， 根据自己项目情况使用不同的解决策略。

**二、SpringCloud微服务的拆分粒度**

1、微服务拆分的起点和终点

起点：既有架构的形态（将一个已有的架构转化为微服务架构）

终点：好的架构不是设计出来的，而是进化来的(一直在演进ing)

2、业务形态不适合微服务的

系统中包含很多强事务场景

业务相对稳定，迭代周期长

访问压力不大，可用性要求不高

3、康威定律

任何组织在设计一套系统（广义概念上的系统）时，所交付的设计方案在结构上都与该组织的沟通结构保持一致。（沟通的问题会影响系统的设计）

4、扩展立方模型

X轴的伸缩：由负载均衡器后运行的多个拷贝构成。如果有N份拷贝，每份拷贝处理1/N的负载。

Y轴的伸缩：Y轴伸缩将应用分成多份不同的服务，每份服务负责一个或多个紧密相关的功能。

Z轴的伸缩：使用Z轴伸缩的话，每个服务器运行一份完全相同的代码，每个服务器只负责数据的一个子集。

5、服务拆分方法

服务拆分关键地方：功能和数据

1.拆功能

单一职责（每个服务只负责业务功能的一个单独的部分），松耦合（服务之间耦合度低，修改一个服务不用导致另一个服务跟着修改），高内聚（服务内部相关的行为都聚集在一个服务内，而不是分散在不同的服务中）

关注点分离：按职责（给服务进行分类，比如订单、商品等）、按通用性（一些基础组件，与具体的业务无关的也可划分成单独的服务，比如消息服务，用户服务）、按粒度级别（微服务并不是越小越好，这个比较难把握）

2.服务和数据的关系

先考虑拆分业务功能，再考虑拆分业务功能对应的数据。

无状态服务（一个数据需要被多个服务共享，才能完成一个请求，这个数据就可以称为状态。依赖这个状态数据的服务称为有状态服务，反之无状态服务）：

3.如何拆数据

每个微服务都有单独的数据存储（一个服务的数据智能通过API来访问，服务之间数据是有隔离的）

依据服务特点，选择不同结构的数据库类型（依据服务的功能特点，选择合适的数据库）

难点在确定边界

- 针对边界设计API

- 依据边界权衡数据冗余