支付宝并发方案之“一锁二判三更新”

每年支付宝在双11和双12的活动中，都展示了绝佳的技术能力。这个能力不但体现在处理高TPS量的访问，更体现在几乎不会出错，不会出现重复支付的情况，是怎么做到的呢？

诚然，为了实现在高并发下仍不会出错的技术目标，支付宝下了很多功夫，比如幂等性的处理，分布式事务的使用等等，但是个人觉得其中最关键的一点就是“一锁二判三更新”这句看似毫不起眼的口诀。

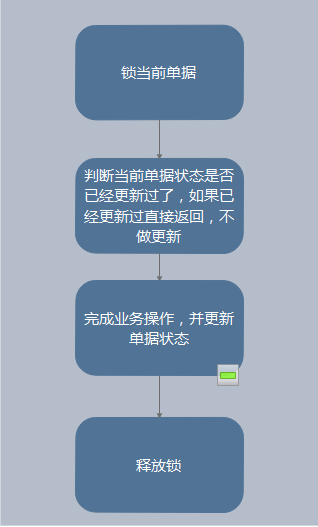
何为“一锁二判三更新”？ 简单来说就是当任何一个并发请求过来的时候

1、先锁定关联单据状态，是否之前已经更新过对应状态了。

2、然后判断关联数据状态，是否之前已经更新过对应状态了。

3、如果基于第2步判断，之前并没有请求更新过对应状态，则本次请求可以更新并完成相关业务逻辑。如果之前已经有更新过状态了，则本次不能更新，也不能完成业务逻辑。

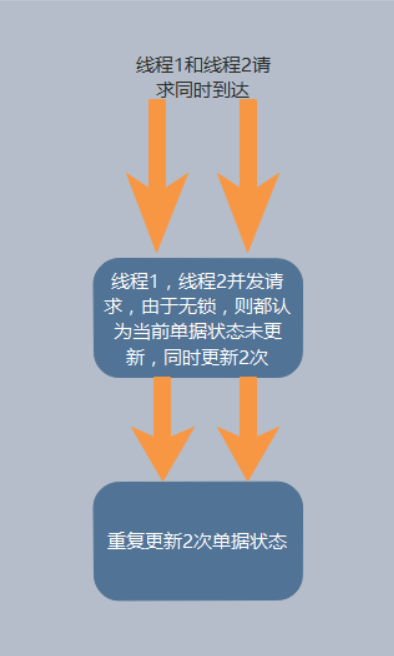
示意图：



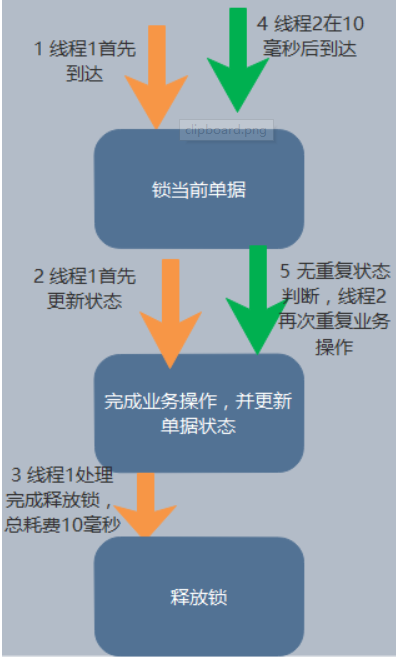
|  |
| --- |
| //第1步锁当前支付单  PaymentInfo resultPaymentInfo = commonPayCoreService.queryPaymentForUpdate(createPaymentInfo.getId());  **if** (resultPaymentInfo.isFinalStatus()) {  //第2步，判断当前支付单状态，如果是终态，则直接返回  //不做任何更新  **return** resultPaymentInfo;  }  //第3步更新当前支付单状态到终态，并完成相关业务逻辑(支付成功)  payCoreService.updateRequestResult(payChannelResult); |

基于以上方案可以100%确保在并发情况下不会出现重复更新的问题，按理论来说，就是每次状态变更锁，都要在并发安全情况下判断状态是否已经发生过变更了。

第1步缺失



第2步缺失



1.Java内存模型(堆、栈)

JVM的内存空间分为五个部分，分别是：

① 程序计数器

② Java虚拟机栈

③ 本地方法栈

④ 堆

⑤ 方法区

1.1.程序计数器

1.1.1.什么是程序计数器

程序计数器是一块较小的内存空间，可以把它看作当前线程正在执行的字节码的行号指示器。也就是说，程序计数器里面记录的是当前线程正在执行的那一条字节码指令的地址。

注：但是，如果当前线程正在执行的是Native方法，那么此时程序计数器为空。

1.1.2.程序计数器的作用

程序计数器有两个作用

① 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，

如：顺序执行、选择、循环、异常处理。

② 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

1.1.3.程序计数器的特点

① 是一块较小的存储空间

② 线程私有，每条线程都有一个程序计数器。

③ 是唯一一个不会出现OutOfMemoryError的内存区域。

④ 生命周期随着线程的创建而创建，随着线程的结束而死亡。

1.2.Java虚拟机栈

1.2.1.什么是Java虚拟机栈

① 重点是Java虚拟机栈，它是线程私有的，生命周期与线程相同。

② 每个方法执行都会创建一个栈帧，用于存放局部变量表，操作数栈，动态链接，方法出口等。每个方法从被调用，直到被执行完，对应着一个栈帧在虚拟机中从入栈到出栈的过程。

③ 通常说的栈就是指局部变量表部分，存放编译期间可知的8种基本数据类型，及对象引用和指令地址。局部变量表是在编译期间完成分配，当进入一个方法时，这个栈中的局部变量分配内存大小是确定的。

1.2.2.Java虚拟机的特点

① 局部变量表的创建是在方法被执行的时候，随着栈帧的创建而创建。

而且，局部变量表的大小在编译时期就确定下来了，在创建时只需分配实现规定好大小。

此外，在方法运行的过程局部变量表的大小是不会发生改变的。

② 会有两种异常StackOverFlowError和OutOfMemoryError。

当线程请求栈深度大于虚拟机所允许的深度就会抛出StackOverFlowError错误；

虚拟机栈动态扩展，当扩展无法申请到足够的内存空间时，抛出OutOfMemoryError。

③ Java虚拟机栈也是线程私有的，每个线程都有各自的Java虚拟机栈，

而且随着线程的创建而创建，随着线程的死亡而死亡。

1.3.本地方法栈

1.3.1.什么是本地方法栈

本地方法栈和Java虚拟机栈实现的功能类似，但本地方法区是本地方法运行的内存模型。

本地方法被执行的时候，在本地方法栈也会创建一个栈帧，

用于存放该本地方法的局部变量表、操作数栈、动态链接、出口信息。

方法执行完毕后相应的栈帧也会出栈并释放内存空间。

也会抛出StackOverFlowError和OutOfMemoryError异常。

1.4.堆

1.4.1.什么是堆

堆是用来存放对象的内存空间。

几乎所有的对象都存储在堆中。

1.4.2.堆的特点

① 线程共享

整个Java虚拟机只有一个堆，所有的线程都访问同一个堆。

而程序计数器、Java虚拟机栈、本地方法栈都是一个线程对应一个的。

② 在虚拟机启动时创建

③ 垃圾回收的主要场所

④ 可以进一步细分为：新生代、老年代。

新生代又可被分为：Eden、From Survivor、To Survivor。

不同的区域存放具有不同声明周期的对象，这样可以根据不同的区域使用不同的垃圾回收算法，从而更具有针对性，从而更高效。

⑤ 堆的大小既可以固定也可以扩展，但主流的虚拟机堆的大小是可扩展的，

因为当线程请求分配内存，但堆已满，且内存已满无法再扩展时，就抛出OutOfMemoryError

1.5.方法区

1.5.1.什么是方法区

Java虚拟机规范中定义方法区是堆的一个逻辑部分。

方法区中存放已经奔虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等。

1.5.2.方法区的特点

① 线程共享

方法区是堆的一个逻辑部分，因此都是线程共享的，整个虚拟机中只有一个方法区。

② 永久代

方法区中的信息一般需要长期存在，而且它又是堆的逻辑分区，因此用堆的划分方法，

我们把方法区称为老年代。

③ 内存回收效率低

方法区中的信息一般需要长期存在，回收一遍内存之后可能只有少量信息无效。

对方法区的内存回收的主要目标是：对常量池的回收 和 对类型的卸载。

④ Java规范对方法区的要求比较宽松

和堆一样，允许固定大小，也允许可扩展的大小，还允许不实现垃圾回收

1.5.3.什么是运行时常量池

方法区中存放三种数据：类信息、常量、静态变量、即时编译器编译后的代码。

其中常量存储在运行时常量池中。

我们一般在一个类中通过public static final来声明一个常量。

这个类被编译后便生成Class文件，这个类的所有信息都存储在这个Class文件中。

当这个类被Java虚拟机加载后，class文件中的常量存放在方法区的运行时常量池中。

而且在运行期间，可以向常量池中添加新的常量。

如：String类中的intern()方法就能在运行期间向常量池中添加字符串常量。

当运行时常量池中的某些常量没有被对象引用，同时也没有被变量引用，

那么就需要垃圾收集器回收。

1.6.直接内存

直接内存是除了Java虚拟机之外的内存，但也有可能被Java使用。

在NIO中引入了一种基于通道和缓冲的IO方式。它可以通过调用本地方法直接分配Java虚拟机之外的内存，然后通过一个存储在Java堆中的DirectByteBuffer对象直接操作该内存，而无需先将外面内存中的数据复制到堆中再操作，从而提升了数据操作的效率。

直接内存的大小不受Java虚拟机控制，

但既然是内存，当内存不足时就会抛出OOM异常。

1.7.总结

① JVM的内存模型中一共有两个“栈”，分别是：Java虚拟机栈和本地方法栈

两个“栈”的功能类似，都是方法运行过程的内存模型，

并且两个“栈”内部构造相同，都是线程私有。

② Java虚拟机的内存模型中一共有两个“堆”，一个是原本的堆，一个是方法区。

方法区本质上是属于堆的一个逻辑部分，堆中存放对象，方法区中存放类信息、常量、

静态变量、即时编译器编译的代码。

③ 堆是Java虚拟机中最大的一块内存区域，也是垃圾收集器主要的工作区域。

④ 程序计数器、Java虚拟机栈、本地方法栈是线程私有的，

即每个线程都拥有各自的程序计数器、Java虚拟机栈、本地方法区。

并且他们的生命周期和所属线程一样。

而堆、方法区是线程共享的，在Java虚拟机中只有一个堆、一个方法区。

并在JVM启动的时候就创建，JVM停止才销毁。

2.JVM类加载过程

加载->链接(验证+准备+解析)->初始化->使用->卸载

2.1.加载

在加载阶段，虚拟机需要完成以下3件事情：

① 通过一个类的全限定名来获取定义此类的二进制字节流。

② 将这个字节流所代表的静态存储结果转化为方法区的运行时数据结构。

③ 在内存中生成一个代表这个类的java.lang.Class对象，

作为方法区这个类的各种数据的访问入口。

虚拟机规范中并没有准确说明二进制字节流应该从哪里获取以及怎样获取，

这里可以通过定义自己的类加载器去控制字节流的获取方式。

2.2.验证

为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，

并且不会危害虚拟机自身的安全。

包括4个阶段的检验动作：文件格式验证、元数据验证、字节码验证、符号引用验证。

2.3.准备

正式为类变量(static修饰的字段变量)分配内存并且设置该类变量的初始值即0(static int i = 5;这里只将i初始化为0，至于5的值将在初始化赋值)，

这里不包括用final修饰的static，因为final在编译的时候就会分配了，

注意这里不会为实例变量分配初始化，类变量会分配在方法区中，

而实例变量是会随着对象一起分配到Java堆中。

2.4.解析

主要将常量池中的符号引用替换为直接引用的过程。

符号引用就是一组符号来描述目标，可以是任何字面量，

而直接引用就是指向目标的指针、相对偏移量或一个间接定位到目标的句柄。

有类或接口的解析，字段的解析，类方法解析，接口方法解析。

2.5.初始化

2.5.1.类什么时候初始化

① 创建类的实例，也就是new一个对象

② 访问某个类或接口的静态变量，或者对该静态变量赋值

③ 调用类的静态方法

④ 反射(Class.forName(“com.grace.load”))

⑤ 初始化一个类的子类(会首先初始化子类的父类)

⑥ JVM启动时标明的启动类，即文件名和类名相同的那个类

2.5.2.类的初始化顺序

① 如果这个类还没有被加载和链接，那先进行加载和链接

② 加入这个类存在直接父类，并且这个类还没有被初始化

(注意：在一个类加载器中，类只能初始化一次)，那就初始化直接的父类(不适用于接口)

③ 类中存在初始化语句(如static变量和static块)，那就依次执行这些初始化语句。

④ 总得来说，初始化顺序依次是：

(静态变量、静态初始化块) -> (变量、初始化块) -> 构造器

如果有父类，则顺序是：

父类static方法 -> 子类static方法 -> 父类构造方法 -> 子类构造方法

2.5.3.《深入JVM补充》

对于初始化阶段，虚拟机规范则是严格规定了有且只有5种情况必须立即对类进行“初始化”

① 遇到new、getstatic、putstatic或invokestatic这4条字节码指令时，

如果类没有进行过初始化，则需要先触发其初始化。

生成这4条指令的最常见的Java代码场景是：使用new关键字实例化对象的时候、

读取或设置一个类的静态字段(被final修饰、已在编译期把结果放入常量池的静态字段除外)的时候，以及调用一个类的静态方法的时候。

② 使用java.lang.reflect包的方法对类进行反射调用的时候，

如果类没有进行过初始化，则需要先触发其初始化。

③ 当初始化一个类的时候，如果发现其父类还没有进行过初始化，

则需要先触发其父类的初始化。

④ 当虚拟机启动时，用户需要指定一个要执行的主类(包含main()方法的那个类)，

虚拟机会先初始化这个主类。

⑤ 当使用JDK1.7的动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF\_getStatic、REF\_putStatic、REF\_invokeStatic的方法句柄，

并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

2.6.Java的类加载器都有哪些

2.6.1.启动类加载器(Bootstrap ClassLoader)

将存放于<JAVA\_HOME>\lib目录中，或者被-Xbootclasspath参数所指定的路径中的，

并且是虚拟机识别的(仅按照文件名识别，如rt.jar，名字不符合的类库即使放在lib目录中也不会被加载)类库加载到虚拟机内存中。启动类加载器无法被Java程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给引导类加载器，

那直接使用null代替即可。

2.6.2.扩展类加载器(Extension ClassLoader)

这个加载器由sun.misc.Launcher$ExtClassLoader实现，

它负责加载<JAVA\_HOME>\lib\ext目录中的，

或者被java.ext.dirs系统变量所指定的路径中的所有类库，

开发者可以直接使用扩展类加载器。

2.6.3.应用程序类加载器(Application ClassLoader)

这个类加载器由sun.misc.Launcher$AppClassLoader实现。

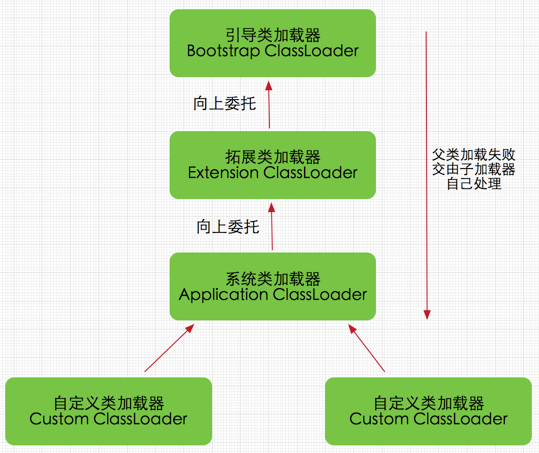
由于这个类加载器时ClassLoader中getSystemClassLoader()方法的返回值，

所以一般也称它为系统类加载器。

它负责加载用户类路径(ClassPath)上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器

在Java的日常应用程序开发中，类的加载几乎是由上述3种类加载器相互配合执行的，在必要时，我们还可以自定义类加载器，需要注意的是，Java虚拟机对class文件采用的是按需加载的方式，也就是说当需要使用该类时才会将它的class文件加载到内存生成class对象，而且加载某个类的class文件时，Java虚拟机采用的是双亲委派模式即把请求交由父类处理，它是一种任务委派模式。

2.6.4.什么是双亲委派模型



双亲委派模型要求除了顶层的启动类加载器外，其余类加载器都应当有自己的父类加载器。

这里的类加载器之间的父子关系一般不会以继承的关系来实现，

而是都使用组合关系来复用父加载器的代码。

双亲委派模型的工作过程是：

如果一个类加载器收到了类加载的请求，

它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，

每一个层次的类加载器都是如此，

因此所有的加载请求最终都应该传送到顶层的启动类加载器中，

只有当父加载器反馈自己无法完成这个加载请求(它得搜索范围中没有找到所需的类)时，子加载器才会尝试自己去加载。

2.6.5.双亲委派模式优势

采用双亲委派模式的好处是Java类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类java.lang.Object，它存放在rt.jar之中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，

因此Object类在程序的各种类加载器环境中都是同一个类。

相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，

如果用户自己编写了一个称为java.lang.Object的类，并放在程序的ClassPath中，

那系统中将会出现多个不同的Object类，Java类型体系中最基础的行为也就无法保证，

应用程序也将会变得一片混乱。

3.Java序列化和反序列化

无论是何种类型的数据，最终都需要转换成二进制流在网络上进行传输，

那么在面向对象程序设计中，如果将一个定义好的对象传输到远端呢？

数据的发送方需要将对象转换成为二进制流，才能在网络上进行传输，

而数据的接收方则需要把二进制流再恢复为对象。

对象的序列化：将对象转换为二进制流的过程。

对象的发序列化：将二进制流恢复为对象的过程。

较为常用的有Java本身内置的序列化方式、Hessian、JSON、XML。

3.1.实现Serializable接口

① 使用默认的序列化机制，即实现java.io.Serializable接口.不需要实现任何方法。

② Serializable接口没有任何方法，只是一个标记而已，告诉该类可以被序列化了。

然后利用ObjectOutputStream的writeObject方法，将Person类的实例zhansan序列化为字节数组，然后再通过OutPutInputStream的readObject方法进行反序列化。

③ 该方式下序列化机制会自动保存该对象的成员变量，

static成员变量和transient关键字修饰的成员变量不会被序列化保存。

3.2.使用Hessian进行序列化

使用Hessian进行序列化，需要引入其提供的二方包hessian-4.0.7.jar，

针对基于Java的Hessian的序列化和反序列化的实现，关键代码如下：

|  |
| --- |
| ByteArrayOutputStream os = new ByteArrayOutputStream();  // Hessian的序列化输出  HessianOutput ho = new HessianOutput(os);  ho.writeObject(zhansan);  byte[] zhansanByte = os.toByteArray();  ByteArraInputStream is= new ByteArraInputStream(zhansanByte);  // Hessian的反序列化读取对象  HessianInput hi = new HessianInput(is);  Person person = (Person)hi.readObject(); |

4.Java IO流的理解

流是一组有顺序的，有起点和终点的字节集合，即数据在两设备间的传输称为流

4.1.IO流的分类

① 根据处理数据类型的不同分为：字符流和字节流

② 根据数据流向不同分为：输入流和输出流

4.2.字符流和字节流

字节流和字符流的区别：

① 读写单位不同：字节流以字节(8bit)为单位，字符流以字符为单位，根据码表映射字符，一次可能读多个字节。

② 处理对象不同：

字节流能处理所有类型的数据(如图片、avi等)，而字符流只能处理字符类型的数据。

③ 字节流：一次读入或读出是8位二进制

④ 字符流：一次读入或读出是16位二进制

结论：只要是处理纯文本数据，就优先考虑使用字符流。除此之外都使用字节流。

4.3.输入流和输出流

输入流只能进行读操作，输出流只能进行写操作，程序中需要根据待传输数据的不同特性而使用不同的流。

4.3.1.输入字节流InputStream

① InputStream是所有的输入字节流的父类，它是一个抽象类。

② ByteArrayInputStream、StringBufferInputStream、FileInputStream

是三种基本介质流，他们分别从Byte数组、StringBuffer和本地文件中读取数据。

② PipeInputStream是从与其他线程共用的管道中读取数据。

③ ObjectInputStream和FilterInputStream的子类都是装饰流(装饰器模式的主角)

4.3.2.输出字节流(OutputStream)

① OutputStream是所有的输出字节流的父类，它是一个抽象类。

② ByteArrayOutputStream、FileOutStream是两种基本的介质流，

它们分别向Byte数组和本地文件中写入数据。

③ PipeOutputStream是向与其它线程共用的管道中写入数据。

④ ObjectOutputStream和FilterOutputStream的子类都是装饰流。

总结：

① 输入流：InputStream或者Reader：从文件中读到程序中

② 输出流：OutputStream或者Writer：从程序中输出到文件中

4.4.节点流

节点流：直接与数据源相连，读入或读出

直接使用节点流，读写不方便，为了更快的读写文件，才有了处理流

4.4.1.常用的节点流

① 父类：InputStream、OutStream、Reader、Writer等

② 文件：FileInputStream、FileOutStream、FileReader、FileWriter文件进行处理的节点流。

③ 数组：ByteArrayInputStream、ByteArrayOutputStream、CharArrayReader、CharArrayWriter对数组进行处理的节点流(对应的不再是文件，而是内存中的一个数组)

④ 字符串：StringReader、StringWriter对字符串进行处理的节点流

⑤ 管道：PipedInputStream、PipedOutputStream、PipedReader、PipedWriter对管道进行处理的节点流。

4.5.处理流

处理流和节点流一块使用，在节点流的基础上，再套接一层，套接在节点流上的就是处理流。如BufferedReader。处理流的构造方法总是要带一个其他的流对象做参数，

一个流对象经过其他流的多次包装，称为流的链接。

4.5.1.常用的处理流

① 缓冲流：BufferedInputStream、BufferedOutputStream、BufferedReader、BufferedWriter增加缓冲功能，避免频繁读写硬盘。

② 转换流：InputStreamReader、OutputStreamReader实现字节流和字符流之间的转换。

③ 数据流：DataInputStream、DataOutputStream等提供将基础数据类型写入到文件中，或者读取出来。

5.线程安全和线程同步锁的了解

线程不安全

多个线程对同一个对象中的实例变量进行并发访问时发生，产生的后果就是“脏读”，

也就是取到的数据其实是被更改过的。

线程安全

就是以获得的实例变量的值是经过同步处理的，不会出现脏读的现象。

线程同步锁

① synchronized同步方法

关键字synchronized取得的锁都是对象锁，而不是把一段代码或方法(函数)当作锁，

哪个线程先执行带synchronized关键字的方法，哪个线程就持有该方法所属对象的锁，那么其他线程只能呈等待状态，前提是多个线程访问的是同一个对象。

② synchronized同步代码块

用关键字synchronized声明方法在某些情况下是有弊端的，

比如A线程调用同步方法执行一个长时间的任务，那么B线程则必须等待比较长时间。

在这样的情况下可以使用synchronized同步代码块来解决。

synchronized方法是对当前对象进行加锁，

而synchronized代码块是对某一个对象进行加锁。

③ 静态同步synchronized方法

synchronized关键字加到static静态方法上是给Class类上锁，

而synchronized关键字加到非static静态方法上是给对象上锁。

④ 与synchronized(class)代码块

同步synchronized(class)代码块的作用其实和synchronized static方法的作用一样。

|  |
| --- |
| public static void printA() {  synchronized(Service.class) {  }  } |

6.Spring框架的IoC，AOP及底层实现机制

回答思路：

① 先用通俗易懂的话解释下何为IoC和AOP

② 各自实现原理

③ 自己的项目中如何使用

IoC：Inversion Of Control控制反转。

将我们创建的对象的方式反转，以前对象的创建是由我们开发人员自己维护，

包括依赖关系也是自己注入；

使用了Spring之后，对象的创建以及依赖关系可以由Spring完成创建以及注入。

反转控制就是反转了对象的创建方式，从我们自己创建反转给了程序。

DI：Dependency Injection依赖注入

实现IOC思想需要DI支持

依赖注入的思想是通过工厂模式+反射+配置文件

在Spring的配置文件中，经常看到如下配置：

|  |
| --- |
| <bean name=*"user"* class=*"com.grace.domain.User"* ></bean> |

通过这样的配置，Spring是通过反射帮我们实例化对象

AOP

面向切面编程，在我们的应用中，经常需要做一些事情，但是这些事情与核心业务无关，比如，要记录所有update \* 方法的执行时间，操作人等信息，记录到日志。

AOP的实现原理 - 代理

第一种：jdk动态代理(优先)

其代理对象必须是某个接口的实现，它是通过在运行期间创建一个接口的实现类来完成对目标对象的代理，其核心的两个类是InvocationHandler和Proxy。

第二种：cglib代理

第三方代理技术，cglib代理，可以对任何类生成代理，代理的原理是对目标对象进行继承代理。如果目标对象被final修饰，那么该类无法被cglib代理。

这两种代理的区别是jdk动态代理需要目标对象实现接口，而cglib的动态代理则不需要。

7.消息队列的常用几种介绍下及应用场景

8.缓存的中间件介绍一二种，及使用场景

9.数据安全用到的加解密方法介绍一二及使用场景

10.数据库悲观锁和乐观锁的原理及应用

11.考核下数据库sql的能力，函数(group by，count(\*)之类)

12.hashmap实现原理

13.有几种线程池，他们的特点

14.spring的源码了解吗，简单介绍一下

14.1.单例模式

① Spring依赖注入Bean实例默认是单例的

② Spring的依赖注入(包括lazy-init方式)都是发生在AbstractBeanFactory的getBean里，getBean的doGetBean

|  |
| --- |
| @Override  **public** Object getSingleton(String beanName) {  // 参数true设置标识允许早期依赖  **return** getSingleton(beanName, **true**);  }  **protected** Object getSingleton(String beanName, **boolean** allowEarlyReference) {  // 检查缓存中是否存在实例  Object singletonObject = **this**.singletonObjects.get(beanName);  **if** (singletonObject == **null** && isSingletonCurrentlyInCreation(beanName)) {  // 如果为空，则锁定全局变量并进行处理  **synchronized** (**this**.singletonObjects) {  // 如果此bean正在加载，则不处理  singletonObject = **this**.earlySingletonObjects.get(beanName);  **if** (singletonObject == **null** && allowEarlyReference) {  ObjectFactory<?> singletonFactory = **this**.singletonFactories.get(beanName);  **if** (singletonFactory != **null**) {  singletonObject = singletonFactory.getObject();  **this**.earlySingletonObjects.put(beanName, singletonObject);  **this**.singletonFactories.remove(beanName);  }  }  }  }  **return** (singletonObject != ***NULL\_OBJECT*** ? singletonObject : **null**);  } |

15.SpringMVC的核心类是哪几个

16.在数据库中怎么实现设计乐观锁

17.数据库的隔离级别是什么

18.数据库索引数据结构

19.sql的优化方式

20.可重入锁和synchronized的区别

21.简单介绍一下ReentrantReadWriteLock读写锁

类

22.JVM的内存区域划分

23.有几种垃圾回收器，他们的特点

24.常用的几个并发容器

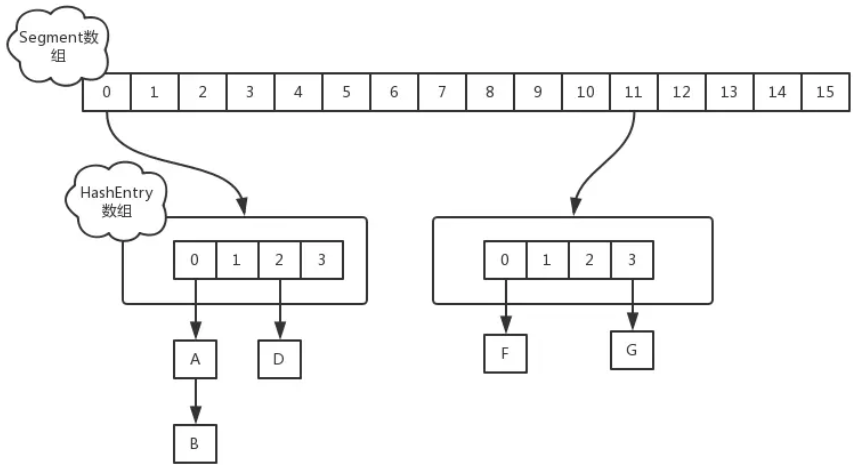
25.ConcurrentHashMap的实现原理是什么

在多线程环境下，使用HashMap进行put操作时存在丢失数据的情况，为了避免这种bug的隐患，强烈建议使用concurrenthashmap代替hashmap

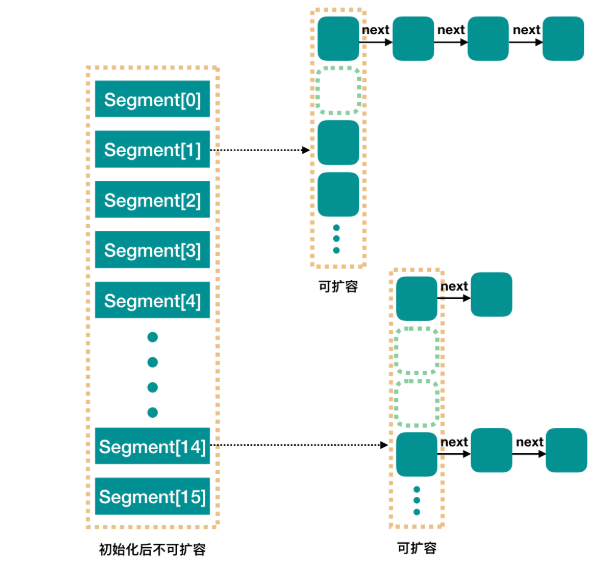
JDK1.7

数据结构

jdk1.7中采用segment+HashEntry的方式进行实现，



ConcurrentHashMap 是一个 Segment 数组，Segment 通过继承 ReentrantLock 来进行加锁，所以每次需要加锁的操作锁住的是一个 segment，这样只要保证每个 Segment 是线程安全的，也就实现了全局的线程安全。



Segment 内部是由 数组+链表 组成的。

初始化槽: ensureSegment

ConcurrentHashMap 初始化的时候会初始化第一个槽 segment[0]，对于其他槽来说，在插入第一个值的时候进行初始化。

这里需要考虑并发，因为很可能会有多个线程同时进来初始化同一个槽 segment[k]，不过只要有一个成功了就可以。

Java8

JDK1.8的实现已经摒弃了Segment的概念，而是直接用Node数组+链表+红黑树的数据结构来实现，并发控制使用synchronized和CAS来操作，整个看起来就像是优化过且线程安全的HashMap。

20180916补充

1.ArrayList、LinkedList、Vector的区别

2.volatile关键字的作用

在并发编程中，

3.ThreadLocal的设计理念与作用

4.接口和抽象类的区别



6.HashMap和HashTable的区别

① HashMap不允许key和value为null，如果value==null直接抛出空指针异常

HashMap允许key和value为null

② Hashtable是线程安全的，HashMap是线程不安全的

因为方法上加了同步synchronized。

③ Hashtable的初始容量与HashMap不同。

Hashtable的初始容量为11，HashMap的初始容量是16.

④ Hashtable计算和定位索引的方式和HashMap不同

Hashtable在定位索引的时候使用key自己的hashCode方法

得到hash值之后与int最大值(0x7FFFFFFF)，然后用结果对容量求余

HashMap在定位索引的时候使用自己实现的hash方法，先求key的hashCode然后高低位做异或扰动，

7.一步步分析SpringMVC源码

8.深入理解Java反射

9.Spring 是如何管理事务的，事务管理机制？

Spring的事务机制包括声明式事务和编程式事务。

编程式事务管理：

Spring推荐使用TransactionTemplate，实际开发中使用声明式事务较多。

声明式事务管理：

将我们从复杂的事务处理中解脱出来，获取连接，关闭连接、事务提交、回滚、异常处理等这些操作都不用我们处理了，Spring都会帮我们处理。

声明式事务管理使用了AOP面向切面编程实现的，本质就是在目标方法执行前后进行拦截。在目标方法执行前加入或创建一个事务，在执行方法执行后，根据实际情况选择提交或是回滚事务。

声明式事务的优缺点：

优点

不需要在业务逻辑代码中编写事务相关代码，只需要在配置文件配置或使用注解（@Transaction），这种方式没有侵入性。

缺点

声明式事务的最细粒度作用于方法上，如果像代码块也有事务需求，只能变通下，将代码块变为方法。

10.Spring 的不同事务传播行为有哪些，干什么用的？

spring在TractionDefinition接口中规定了7种类型的事务传播行为。

① PROPAGATION\_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。

② PROPAGATION\_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。‘

③ PROPAGATION\_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。

④ PROPAGATION\_REQUIRES\_NEW：创建新事务，无论当前存不存在事务，都创建新事务。

⑤ PROPAGATION\_NOT\_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

⑥ PROPAGATION\_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

⑦ PROPAGATION\_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。

11.synchronized和lock的区别

|  |
| --- |
|  |