## 步骤：

什么是BigTable？

BigTable的特性有哪些？

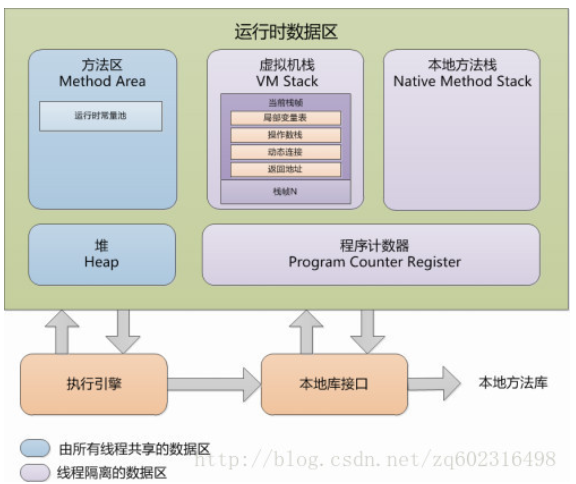
为什么要使用BigTable？

如何使用BigTable？

BigTable的实现原理是什么？

BigTable的优缺点是什么？  
– 参考资料

## JVM



### 1.程序计数器：

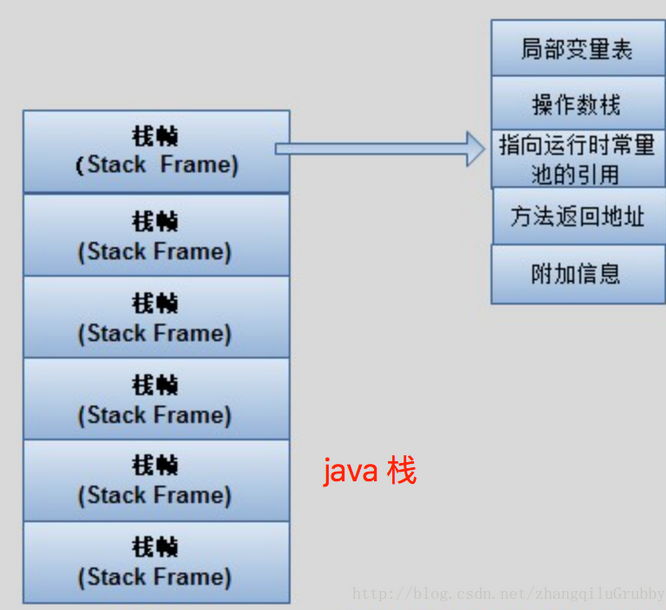
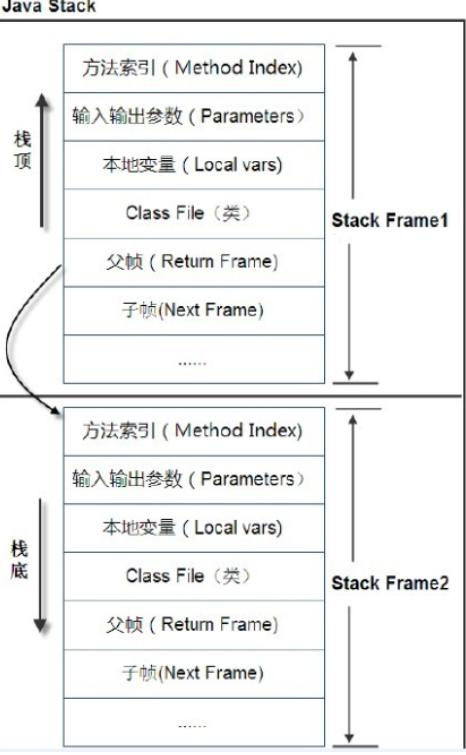
1.如果线程正在执行的是一个java方法，那么计数器记录的当前代码的行号，确定执行哪一条字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都会依赖这个计数器完成

2.如果执行的native方法，计数器当中的内容应当是空   
3.此内存区域在java的虚拟机规范当中是唯一一个没有规定OutOfMemoryError的区域

### 2.JVM栈：

栈帧：一个栈帧随着一个方法的调用开始而创建，这个方法调用完成而销毁。局部变量表存放了编译器可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference类型，它不等同于对象本身，可能是指向一个代表对象的句柄或者其他与此对象相关的变量）和retureAddress类型。其中64位长度的long和double类型的数据会占2个局部变量空间，其余的数据类型只占1个。

Java栈也称作虚拟机栈（Java Vitual Machine Stack），JVM栈只对栈帧进行存储，压栈和出栈操作。Java栈是Java方法执行的内存模型。下面我们来看一个Java栈图。

对于所有的程序设计语言来说，栈这部分空间对程序员来说是不透明的。

栈内存的大小可以有两种设置，固定值和根据线程需要动态增长。  
在JVM栈这个数据区可能会发生抛出两种错误。   
1. StackOverflowError 出现在栈内存设置成固定值的时候，当程序执行需要的栈内存超过设定的固定值会抛出这个错误。   
2. OutOfMemoryError 出现在栈内存设置成动态增长的时候，当JVM尝试申请的内存大小超过了其可用内存时会抛出这个错误。

### 3.本地方法栈

这块内存区域和虚拟机栈非常相似，他们的区别从名字就可以看出来：Java虚拟机栈是用来执行Java方法的，而本地方法栈是用来执行Native方法的。

### 4.堆内存：

Java堆应该是虚拟机管辖范围内最大的一块内存区域。这块区域是被所有线程共享，在虚拟机一启动的时候就创建的，它的唯一目的就是存放对象实例，几乎所有对象的实例都需要到这里申请内存。

Java堆是垃圾回收期（GC）管理的主要区域，从内存回收的角度来看，现代的收集器都采用分代收集算法，所有这块区域又会细分为好几个区域：新生代、老年代；再细节的又会分为 Eden、From Survivor、To Survivor，下图展示了详细的区域：



**新生代（Young Generation）**：对象被创建时，内存的分配首先发生在年轻代（大多数对象可以直接被创建在年老代），大部分的对象在创建后很快就不再使用，因此很快变得不可用，于是被年轻代的GC机制清理掉（IBM的研究表明，98%的对象都是很快消亡的），这个GC机制被称为Minor GC或叫Young GC。注意，Minor GC并不代表年轻代内存不足，它事实上只表示在Eden区上的GC。

年轻代上的内存分配是这样的，年轻代可以分为3个区域：Eden区（伊甸园，亚当和夏娃偷吃禁果生娃娃的地方，用来表示内存首次分配的区域，再贴切不过）和两个存活区（Survivor 0 、Survivor 1）

1. 绝大多数刚创建的对象会被分配在Eden区，其中的大多数对象很快就会消亡。Eden区是连续的内存空间，因此在其上分配内存极快；
2. 最初一次，当Eden区满的时候，执行Minor GC，将消亡的对象清理掉，并将剩余的对象复制到一个存活区Survivor0（此时，Survivor1是空白的，两个Survivor总有一个是空白的）；
3. 下次Eden区满了，再执行一次Minor GC，将消亡的对象清理掉，将存活的对象复制到Survivor1中，然后清空Eden区；
4. 将Survivor0中消亡的对象清理掉，将其中可以晋级的对象晋级到Old区，将存活的对象也复制到Survivor1区，然后清空Survivor0区；
5. 当两个存活区切换了几次（HotSpot虚拟机默认15次，用XX:MaxTenuringThreshold控制，大于该值进入老年代，但这只是个最大值，并不代表一定是这个值）之后，仍然存活的对象（其实只有一小部分，比如，我们自己定义的对象），将被复制到老年代。

　从上面的过程可以看出，Eden区是连续的空间，且Survivor总有一个为空。经过一次GC和复制，一个Survivor中保存着当前还活着的对象，而Eden区和另一个Survivor区的内容都不再需要了，可以直接清空，到下一次GC时，两个Survivor的角色再互换。因此，这种方式分配内存和清理内存的效率都极高，这种垃圾回收的方式就是著名的**“停止复制（Stopandcopy）”清理法（将Eden区和一个Survivor中仍然存活的对象拷贝到另一个Survivor中）**

**年老代（Old Generation）**：对象如果在年轻代存活了足够长的时间而没有被清理掉（即在几次Young GC后存活了下来），则会被复制到年老代，年老代的空间一般比年轻代大，能存放更多的对象，在年老代上发生的GC次数也比年轻代少。当年老代内存不足时，将执行Major GC，也叫 Full GC。

**Minor GC ，Full GC 触发条件**

Minor GC触发条件：当Eden区满时，触发Minor GC。

Full GC触发条件：

（1）调用System.gc时，系统建议执行Full GC，但是不必然执行

（2）老年代空间不足

（3）方法区空间不足。

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存

（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

GC机制的基本算法是：分代收集

在JVM调优，可以通过配置以下参数改变整个JVM堆的配置比例

1.Java heap的大小（新生代+老年代）

　　Xms堆的最小值

　　Xmx堆空间的最大值

2.新生代堆空间大小调整

　　XX:NewSize新生代的最小值

　　XX:MaxNewSize新生代的最大值

　　XX:NewRatio设置新生代与老年代在堆空间的大小

　　XX:SurvivorRatio新生代中Eden所占区域的大小

3.永久代大小调整

　　XX:MaxPermSize

4.其他

　 XX:MaxTenuringThreshold,设置将新生代对象转到老年代时需要经过多少次垃圾回收，但是仍然没有被回收

### 5.方法区(非堆内存)：

**【名词解析】  
        >和java堆一样，方法区是一块所有线程共享的内存区域。  
        >保存系统的类信息，比如，类的字段，方法，常量池等。  
        >方法区的大小决定了系统可以保存多少个类，如果系统定义了太多的类，导致方法区溢出，虚拟机同样会抛出内存溢出的错误  
        >jdk1.6和jdk1.7方法区可以理解为永久区。  
        >jdk1.8已经将方法区取消，替代的是元数据区。  
        >jdk1.8的元数据区可以使用参数XX:MaxMetaspaceSzie设定大小，这是一块堆外的直接内存，与永久区不同，如果不指定大小，默认情况下，虚拟机会耗尽可用系统内存  
  
【参数设定】  
        >jdk1.6和jdk1.7的永久区可以使用参数XX:PermSize 和XX:MaxPermSize指定  
        >XX:PermSize =5m 默认启动大小为5M  
        >XX:MaxPermSize=64m 最大大小为64M  
【异常】  
        >jdk1.8元数据区内存溢出：java.lang.OutOfMemoryError:Metaspace**

**（类加载时候执行）**

静态代码块>静态属性>静态方法>普通属性>普通代码块>构造方法>普通方法

(静态代码块，静态属性代码顺序有关，属性为默认值) –> (普通属性,普通代码块，代码顺序有关，属性为默认值)>构造方法

### 6.JVM类加载，链接，初始化。

程序运行时，加载类主要经过3个阶段分别是类的加载，连接和初始化。分别介绍一下这三个过程。

加载 –> 验证 > 准备 > 解析 > 初始化 > 使用 > 卸载

#### 6.1 加载

**类的加载指的是将类的.class文件中二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象用来封装类在方法区内的数据结构。【】**

加载.class文件的方式

1、从本地系统中直接加载

2、通过网络下载.class文件

3、从zip,jar等归档文件中加载.class文件

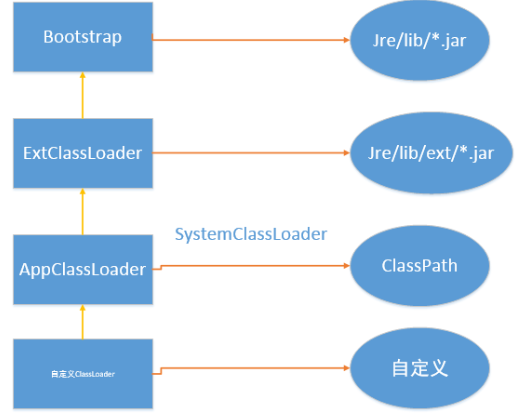
4、从专有数据库中提取.class文件

5、将Java源文件动态编译为.class文件

类加载的最终产品是位于堆区中的class对象，Class对象封装了类在方法区内的数据结构，并向Java程序员提供了访问方法区内的数据机构的接口.   
我们可以通过类名.class来获取一个类的类型的引用，通过new 类名().getClass()来获取一个实例变量的类的引用

类的加载机制

从JDK1.2开始类加载采用父亲委托机制。除了Java虚拟机自带的根类加载器以外，其余的类加载器都有且只有一个父加载器。当Java程序启动加载器加载某个类时，加载器会首先委托自己的父加载器去加载该类，若父加载器能加载，则由父加载器完成加载任务，否则才由自加载器去加载。



同时，所有能成功返回Class对象的引用的类加载器(包括定义类加载器，即包括定义类加载器和它下面的所有子加载器)都被称为初始类加载器。   
假设loader1实际加载了Sample类，则loader1为Sample类的定义类加载器

#### 6.2，链接

类加载完成后就进入了类的连接阶段，连接阶段主要分为三个过程分别是：验证，准备和解析。在连接阶段，主要是将已经读到内存的类的二进制数据合并到虚拟机的运行时环境中去。

验证

**这个阶段主要目的是保证Class流的格式是正确的。主要验证的内容包括：**

1、文件格式的验证

是否以0xCAFEBABE开头

版本号是否合理

2、元数据的验证

是否有父类

是否继承了final类

非抽象类实现了所有抽象方法

3、字节码验证

运行检查

栈数据类型和操作码数据参数吻合

跳转指令指定到合理的位置

4、符号引用验证

常量池中描述类是否存在

访问的方法或字段是否存在且有足够的权限

准备

这个阶段主要是为对象和变量分配内存，并为类设置初始值（方法区中）

给基本类型赋值0，false，引用类型为null。

重点：对于static类型变量在这个阶段会为其赋值为默认值，比如public static int v=5,在这个阶段会为其赋值为v=0，而对于static final类型的变量，在准备阶段就会被赋值为正确的值

PS：

解析

在这个阶段会将符号引用转换成直接引用。   
原来的符号引用仅仅是一个字符串，而引用的对象不一定被加载，直接引用只的是将引用对象的指针或者地址偏移量指向真正的对象，将字符串所指向的对象加载到内存中。

#### 6.3，初始化

**在这个阶段主要执行类的构造方法。并且先为静态变量赋值为初始值，执行静态块代码。**

**(静态代码块，静态属性初始化真正值，代码顺序有关，属性为默认值) –> (普通属性,普通代码块，代码顺序有关，属性为默认值)>构造方法**

类的初始化步骤   
1、假如这个类还没有被加载和连接，那就先进行加载和连接   
2、假如类存在直接的父类，并且这个父类还没有被初始化，那就先初始化它的父类   
3、假如类中存在初始化语句时，那就依次执行这些初始化语句。

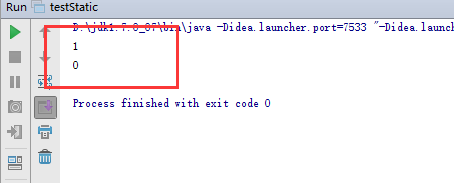
类的初始化时机   
所有的Java类只有在对类的首次主动使用时才会被初始化。主动使用的情况有六中，其他情况都属于被动使用：   
1、 创建类的实例   
2、访问某个类或接口的静态变量，或者对该静态变量赋值   
3、调用类的静态方法   
4、反射（Class.fotName）   
5、初始化一个类的子类   
6、Java虚拟机启动时被标明为启动类的类（main方法所在的类）

注意：1、当Java虚拟机初始化一个类时，要求他的所有父类都已经被初始化，但是这条规则并不适合接口。在初始化一个类或接口时，并不会先初始化它所实现的接口。   
2、只有当程序访问的静态变量或静态方法确实在当前类或当前接口中定义时，才可以认为是对类或接口的主动使用。如果静态方法或变量在parent中定义，从子类进行调用，则不会初始化子类。

实例1：

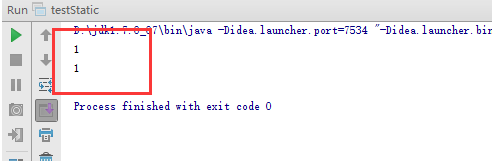
|  |
| --- |
| public class testStatic {  public static void main(String args[]){  Singleton singleton=Singleton.getInstance();  System.out.println(singleton.count1);  System.out.println(singleton.count2);  }  }  class Singleton{  private static Singleton singleton=new Singleton();  public static int count1;  public static int count2=0;  private Singleton(){  count1++;  count2++;  }  public static Singleton getInstance(){  return singleton;  }  } |

输出的结果：



|  |
| --- |
| class Singleton{  public static int count1;  public static int count2=0;  private static Singleton singleton=new Singleton();//调换了位置  private Singleton(){  count1++;  count2++;  }  public static Singleton getInstance(){  return singleton;  }  } |

将位置调换，结果为



分析原因：

**PS：按自上向下顺序来：**

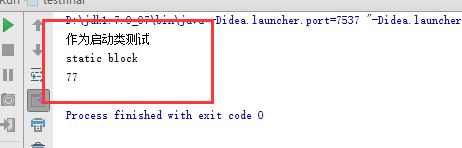
主要演示静态变量的加载。

1、在main方法中调用静态方法单利模式创建类的实例时，是对类的主动使用，同时在类的初始化时，会执行类的构造方法，在第一种情况下，执行完构造方法时，   
Java虚拟机会对类中的静态变量进行复制，所以按顺序执行count1没有被赋值，还是count1++=1，而count2=0所以又重新被赋值为0了。   
2、而第二种情况，是先赋值，再执行构造方法，所以结果为1,1.这个小例子说明类在初始化时，类里面的静态变量赋值语句和构造方法执行时是有先后顺序的。

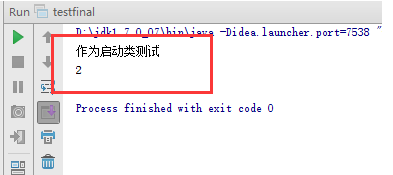
**实例二：**

|  |
| --- |
| class FinalTest {  // public static final int x=new Random().nextInt(100); //需要在运行时赋值，所以需要进行初始化 第一种场景  public static final int x=2; //在编译时已经确定，不需要进行初始化 第二种场景  static  {  System.out.println("static block");  }  }  public class testfinal{  static{  System.out.println("作为启动类测试");  }  public static void main(String args[]){  System.out.println(FinalTest.x);  }  } |

第一种场景：



第二种场景：



分析原因：

主要考虑，静态变量为final类型的情形。  
1、两种情况都输出了”作为启动类测试”，这是因为这个静态块放在了main函数所在的类作为了启动类，所以这属于对类的主动使用，所以每次都会执行这个静态块。   
2、第一种情况输出了”static block”是因为在声明x时是new Random.nextInt(100)，而这句声明在编译时是不能确定x的具体的值的。所以在运行时，需要去计算x的值，这就相当于调用了类中的静态变量。所以会去初始化该类。   
3、第二种情况没有输出”static block”是因为x=2，而这条语句编译器在编译时就能确定x的值，所以在运行时直接调用它的值就可以了，不需要去初始化这个类。所以不会去执行静态块。   
以上就是Java类从加载，连接到初始化的整个过程。

**Synchronized关键字及用法**

synchronized是Java中的关键字，是一种同步锁。它修饰的对象有以下几种：   
1. 修饰一个代码块，被修饰的代码块称同步语句块，其作用的范围是大括号{}括起来的代码，作用的对象是调用这个代码块的对象；   
2. 修饰一个方法，被修饰的方法称为同步方法，其作用的范围是整个方法，作用的对象是调用这个方法的对象；   
3. 修改一个静态的方法，其作用的范围是整个静态方法，作用的对象是这个类的所有对象；   
4. 修改一个类，其作用的范围是synchronized后面括号括起来的部分，作用主的对象是这个类的所有对象。

## 线程安全解释：

我们之前给出了同步基本思想的三个点，我们也说了jvm的三种锁都是以基本思想为基础的，而这三种锁在第1、2点的实现上本质上是一样的：

1. 在共享数据里保存一个锁 //java同步是通过synchronized关键字实现的，synchronized有三种用法：一种是同步块，这种用法需要指明一个锁定对象；一种是修饰静态方法，这种用法相当于锁定Class对象；一种是修饰普通方法，这种用法相当于锁定方法所在的实例对象。因此，在java里能够被synchronized关键字锁定的一定是对象，因此就要在对象里保存一个锁，而对象内存结构里的MarkWord就可以认为是这个锁。三种锁虽然实现细节不同，但是都是使用MarkWord保存锁的。
2. 在锁里保存这个线程的标识 //偏向锁是在MarkWord里保存线程id，轻量级锁是在MarkWord里保存指向拥有锁的线程栈中锁记录的指针，重量级锁是在MarkWord中保存指向互斥量的指针(互斥量只向一个线程授予对共享资源的独占访问权，可以认为是记录了线程的标识)而区分这三种锁的关键，就是同步基本思想的第三点：
3. 其他线程访问已加锁共享数据要等待锁释放 //这里的等待锁释放是一个抽象的说法，并没有严格要求怎么等待。而重量级锁因为使用了互斥量，这里的等待就是线程阻塞。使用互斥量可以保证所有情况下的并发安全，但是使用互斥量会带来较大的性能消耗。而且在实际的项目代码中，很可能一段本来不会有并发情况的代码被加了锁，这样每次使用互斥量就白白消耗了性能。能不能先假设被加锁的代码不会有并发的情况，等到发现有并发的时候再使用互斥量呢？答案是可以的，轻量级锁和偏向锁都是基于这种假设来实现的。
4. 其实抛开实现的细节，java的多线程很简单：

java多线程主要面临的问题就是线程安全问题.  
 线程安全问题是由线程间的通信造成的，

多个线程间不通信就没有线程安全问题    
java中线程通信只能通过类变量和实例变量，因此解决线程安全问题就是解决对变量的安全访问问题    
java中解决变量的安全访问采用的是同步的手段，同步是通过锁实现的  
有三种锁能保证变量只有一个线程访问，偏向锁最快但是只能用于从始至终只有一个线程获得锁，轻量级锁较快但是只能用于线程串行获得锁，重量级锁最慢但是可以用于线程并发获得锁，先用最快的偏向锁，每次假设不成立就升级一个重量。

\*

当 Java 类文件编译时，所有变量和方法的引用都被当做符号引用存储在这个类的常量池中。符号引用是一个逻辑引用，实际上并不指向物理内存地址。JVM 可以选择符号引用解析的时机，一种是当类文件加载并校验通过后，这种解析方式被称为饥饿方式。另外一种是符号引用在第一次使用的时候被解析，这种解析方式称为惰性方式。无论如何 ，JVM 必须要在第一次使用符号引用时完成解析并抛出可能发生的解析错误。绑定是将对象域、方法、类的符号引用替换为直接引用的过程。绑定只会发生一次。一旦绑定，符号引用会被完全替换。如果一个类的符号引用还没有被解析，那么就会载入这个类。每个直接引用都被存储为相对于存储结构（与运行时变量或方法的位置相关联的）偏移量

## Java8 新特性：

### 一、Lambda表达式

Lambda表达式可以说是Java 8最大的卖点，她将函数式编程引入了Java。Lambda允许把函数作为一个方法的参数，或者把代码看成数据。

### 二、接口的默认方法与静态方法

我们可以在接口中定义默认方法，使用default关键字，并提供默认的实现。所有实现这个接口的类都会接受默认方法的实现，除非子类提供的自己的实现

### 三、方法引用

通常与Lambda表达式联合使用，可以直接引用已有Java类或对象的方法。一般有四种不同的方法引用：

1. 构造器引用。语法是Class::new，或者更一般的Class< T >::new，要求构造器方法是没有参数；
2. 静态方法引用。语法是Class::static\_method，要求接受一个Class类型的参数；
3. 特定类的任意对象方法引用。它的语法是Class::method。要求方法是没有参数的；
4. 特定对象的方法引用，它的语法是instance::method。要求方法接受一个参数，与3不同的地方在于，3是在列表元素上分别调用方法，而4是在某个对象上调用方法，将列表元素作为参数传入；

### 四、Stream

Stream API是把真正的函数式编程风格引入到Java中。其实简单来说可以把Stream理解为MapReduce，当然Google的MapReduce的灵感也是来自函数式编程。她其实是一连串支持连续、并行聚集操作的元素。从语法上看，也很像linux的管道、或者链式编程，代码写起来简洁明了，非常酷帅！

### 五、Optional

Java 8引入Optional类来防止空指针异常，Optional类最先是由Google的Guava项目引入的。Optional类实际上是个容器：它可以保存类型T的值，或者保存null。使用Optional类我们就不用显式进行空指针检查了。

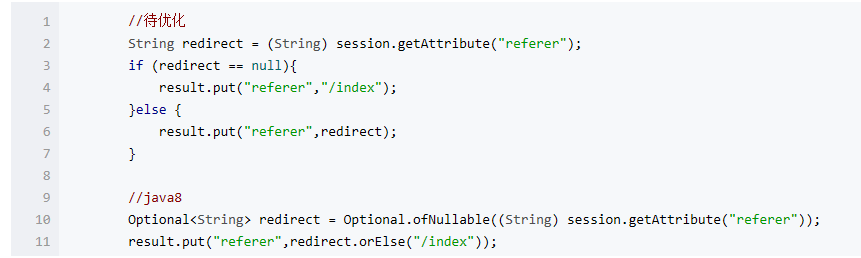
1. 尽量避免使用get()方法

下面是get方法的源码,当值为null的时候,会抛出异常,这和我们使用该类方法的目的冲突,我们 的目的就是尽可能消灭异常.



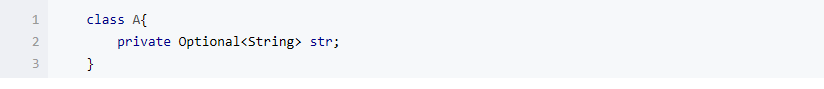
2.尽量避免使用isPresent()方法

无参的该方法,看源码,该方法判断该对象是否存在,对于代码来说和直接写if(value != null)区别并不大. 这里更应该把isPresent()视为一个private方法,因为Optional内部其他实现是大多使用了该方法.



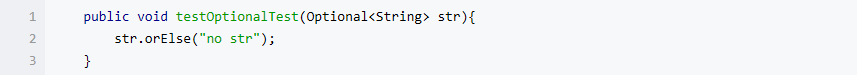
3.不要作为类的实例属性

如下做法是大错特错,Optional是容器,如下是把容器当成实例对象.



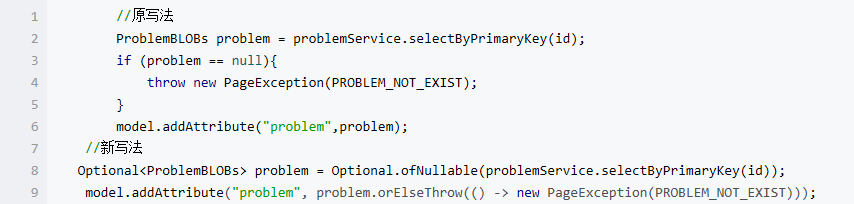
4.不要作为方法参数

Optional 只设计为类库方法的, 可明确表示可能无值情况下的返回类型. Optional 类型不可被序列化,用作字段类型会出问题的.



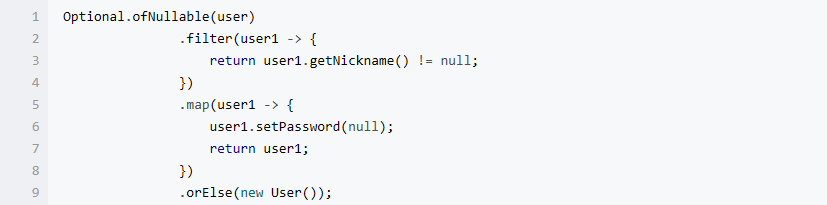
5.抛出异常

这个就看个人喜欢了,只是一种代替方案,相比下原写法更加清晰明了.



6.流式调用

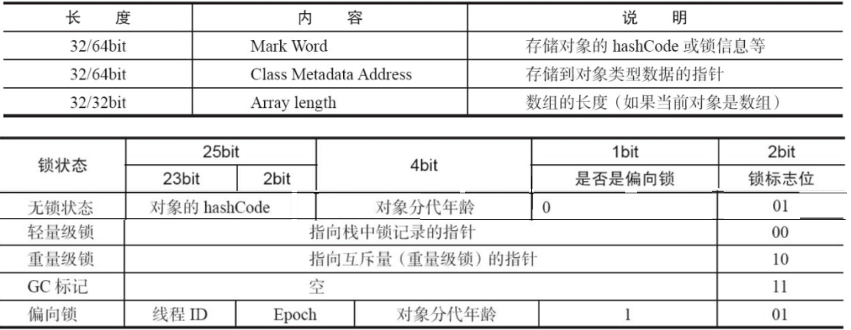
使用 filter和map进行过滤和转换,使用orElse进行收尾,这是一般常用做法.



`

## 锁机制。

对象头：三部分组成，存放在虚拟机栈



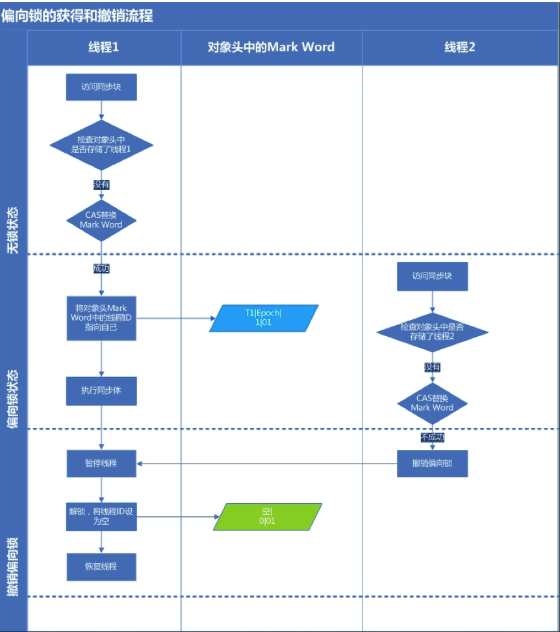
### 1.偏向锁：

加锁

顾名思义，它会偏向于第一个访问锁的线程，如果在运行过程中，同步锁只有一个线程访问，不存在多线程争用的情况，则线程是不需要触发同步的，这种情况下，就会给线程加一个偏向锁。

偏向锁加锁发生在偏向线程第一次进入同步块时，CAS原子操作尝试更新对象的Mark Word（偏向锁标志位为"1"，记录偏向线程的ID）。

释放锁  
如果在运行过程中，遇到了线程2抢占锁，比较锁线程1 的ID，如果一致就不再加锁，提高了执行效率，如果不一致，则线程1（持有偏向锁的线程）会被挂起，JVM会消除它身上的偏向锁，然后线程1和线程2 膨胀为轻量级锁。



如果锁一直处于竞争状态，就没必要使用偏向锁。

代码演示：

### 2.轻量级锁。

加锁：

 加锁过程和偏向锁加锁差不多，也是CAS修改对象头，只是修改的内容不同。

1. 在MarkWord中保存当前线程的指针
2. 修改锁标识位为“00”

**采用CAS操作的原因是，不想在加锁解锁上再加同步**

如果对象处于无锁状态（偏向锁标志位为"0"，锁标志位为"01"），会在线程的栈中开辟个锁记录空间（Lock Record），将Mark Word拷贝一份到Lock Record中，称为Displaced Mark Word，在Lock Record中保存对象头的指针（owner）。  
接下来CAS更新MarkWord，将MarkWord指向当前线程，owner指向MarkWord，如果失败了，则意味着出现了另一个线程竞争锁，此时需要锁膨胀为轻量级锁

释放锁：

用CAS操作锁置为无锁状态（偏向锁位为"0"，锁标识位为"01"），若CAS操作失败则是出现了竞争，锁已膨胀为重量级锁了，此时需要释放锁（持有重量级锁线程的指针位为"0"，锁标识位为"10"）并唤醒重量锁的线程。

膨胀为重量级锁的时机：

1. 自选的次数超过一定次数，自动升级为重量级锁。
2. 在自旋过程中，发现有新的线程发生竞争，要获得相同锁资源，自动升级为重量级锁

CAS：利用

**自旋锁**：

自旋锁原理非常简单，如果持有锁的线程能在很短时间内释放锁资源，那么那些等待竞争锁的线程就不需要做内核态和用户态之间的切换进入阻塞挂起状态，它们只需要等一等（自旋），等持有锁的线程释放锁后即可立即获取锁，这样**就避免用户线程和内核的切换的消耗**

但是线程自旋是需要消耗cpu的，说白了就是让cup在做无用功，线程不能一直占用cup自旋做无用功，所以需要设定一个自旋等待的最大时间。

### 3.重量级锁。

重量级锁的加锁、解锁过程和轻量级锁差不多，区别是：竞争失败后，线程阻塞。

释放锁后，唤醒阻塞的线程，不使用自旋锁，不会那么消耗CPU，所以重量级锁适合用在同步块执行时间长的情况下。

偏向锁，轻量锁，重量锁的优缺点。



同步：在发出一个同步调用时，在没有得到结果之前，该调用就不返回。  
异步：在发出一个异步调用后，调用者不会立刻得到结果，该调用就返回了。

**阻塞：**是指调用结果返回之前，调用者会进入阻塞状态等待。只有在得到结果之后才会返回。  
**非阻塞：**是指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。

## Enum：

### 什么是Enum？

1.创建枚举类型要使用 enum 关键字，隐含了所创建的类型都是 java.lang.Enum 类的子类（java.lang.Enum 是一个抽象类）。

2.枚举类型符合通用模式 Class Enum<E extends Enum<E>>，而 E 表示枚举类型的名称。

3.枚举类型的每一个值都将映射到 protected Enum(String name, int ordinal) 构造函数中，在这里，每个值的名称都被转换成一个字符串，并且序数设置表示了此设置被创建的顺序。

定义格式：

[修饰符] enum 枚举类名

{

常量A,常量B,常量C;

}

### Enum的特性有哪些？

1、枚举的直接父类是java.lang.Enum，但是不能显示的继承Enum   
2、枚举就相当于一个类，可以定义构造方法、成员变量、普通方法和抽象方法   
3、默认私有的构造方法，即使不写访问权限也是**private。（**假构造器，底层没有无参数的构造器）   
4、每个实例分别用于一个全局常量表示，枚举类型的对象是固定的，实例个数有限，不能使用new关键字。   
5、枚举实例必须位于枚举中最开始部分，枚举实例列表的后面要有分号月其他成员相分隔   
6、枚举实例后有花括号时，该实例是枚举的匿名内部类对象

### 为什么要使用Enum？

1. enum的存在本身就是为了解决常量的可读性的问题，在没有enum之前，标记一种状态，常使用int类型，其1，2之类的可读性不是很好。

2. 而且新增一个常量要写一个长长语句，为了解决这一系列问题在jdk5就加上了这个功能。其本质就是一个类，只不过在创建enum的时候，会自动生成values()、ordinal()等方法，并且支持了switch语句，覆盖了常量会出现的场景，比直接定义常量要方便使用。

### 如何使用Enum？

**用法一：常量**

在JDK1.5 之前，我们定义常量都是： public static fianl.... 。现在好了，有了枚举，可以把相关的常量分组到一个枚举类型里，而且枚举提供了比常量更多的方法。

1. **public** **enum** Color {
2. RED, GREEN, BLANK, YELLOW
3. }

**用法二：switch**

JDK1.6之前的switch语句只支持int,char,enum类型，使用枚举，能让我们的代码可读性更强。

1. **enum** Signal {
2. GREEN, YELLOW, RED
3. }
4. **public** **class** TrafficLight {
5. Signal color = Signal.RED;
6. **public** **void** change() {
7. **switch** (color) {
8. **case** RED:
9. color = Signal.GREEN;
10. **break**;
11. **case** YELLOW:
12. color = Signal.RED;
13. **break**;
14. **case** GREEN:
15. color = Signal.YELLOW;
16. **break**;
17. }
18. }
19. }

**用法三：向枚举中添加新方法**

如果打算自定义自己的方法，那么必须在enum实例序列的最后添加一个分号。而且 Java 要求必须先定义 enum

1. **public** **enum** Color {
2. RED("红色", 1), GREEN("绿色", 2), BLANK("白色", 3), YELLO("黄色", 4);
3. // 成员变量
4. **private** String name;
5. **private** **int** index;
6. // 构造方法
7. **private** Color(String name, **int** index) {
8. **this**.name = name;
9. **this**.index = index;
10. }
11. // 普通方法
12. **public** **static** String getName(**int** index) {
13. **for** (Color c : Color.values()) {
14. **if** (c.getIndex() == index) {
15. **return** c.name;
16. }
17. }
18. **return** **null**;
19. }
20. // get set 方法
21. **public** String getName() {
22. **return** name;
23. }
24. **public** **void** setName(String name) {
25. **this**.name = name;
26. }
27. **public** **int** getIndex() {
28. **return** index;
29. }
30. **public** **void** setIndex(**int** index) {
31. **this**.index = index;
32. }
33. }

用法四：覆盖枚举的方法

下面给出一个toString()方法覆盖的例子

1. **public** **enum** Color {
2. RED("红色", 1), GREEN("绿色", 2), BLANK("白色", 3), YELLO("黄色", 4);
3. // 成员变量
4. **private** String name;
5. **private** **int** index;
6. // 构造方法
7. **private** Color(String name, **int** index) {
8. **this**.name = name;
9. **this**.index = index;
10. }
11. //覆盖方法
12. @Override
13. **public** String toString() {
14. **return** **this**.index+"\_"+**this**.name;
15. }
16. }

用法五：实现接口

所有的枚举都继承自java.lang.Enum类。由于Java 不支持多继承，所以枚举对象不能再继承其他类。

1. **public** **interface** Behaviour {
2. **void** print();
3. String getInfo();
4. }
5. **public** **enum** Color **implements** Behaviour{
6. RED("红色", 1), GREEN("绿色", 2), BLANK("白色", 3), YELLO("黄色", 4);
7. // 成员变量
8. **private** String name;
9. **private** **int** index;
10. // 构造方法
11. **private** Color(String name, **int** index) {
12. **this**.name = name;
13. **this**.index = index;
14. }
15. //接口方法
16. @Override
17. **public** String getInfo() {
18. **return** **this**.name;
19. }
20. //接口方法
21. @Override
22. **public** **void** print() {
23. System.out.println(**this**.index+":"+**this**.name);
24. }
25. }

用法六：使用接口组织枚举

1. **public** **interface** Food {
2. **enum** Coffee **implements** Food{
3. BLACK\_COFFEE,DECAF\_COFFEE,LATTE,CAPPUCCINO
4. }
5. **enum** Dessert **implements** Food{
6. FRUIT, CAKE, GELATO
7. }
8. }

用法七：关于枚举集合的使用

java.util.EnumSet和java.util.EnumMap是两个枚举集合。EnumSet保证集合中的元素不重复；EnumMap中的 key是enum类型，而value则可以是任意类型。关于这个两个集合的使用就不在这里赘述，可以参考JDK文档。

### Enum的实现原理是什么？

注意enum不是Enum，有Java基础的同学们应该都不会把二者混淆了。简单来说，enum只是jdk1.5引入的语法糖，它不是java中的新增类型，编译器在编译阶段会自动将它转换成一个继承于Enum的子类，例如如下的代码

public enum GenderEnum {

MALE,

FEMALE

}

编译成class文件后，通过javap GenderEnum.class得到的简单的编译后的结构为：

public final class com.yuanxz.example.GenderEnum extends java.lang.Enum<com.yuanxz.example.GenderEnum> {

public static final com.yuanxz.example.GenderEnum MALE;

public static final com.yuanxz.example.GenderEnum FEMALE;

static {};

public static com.yuanxz.example.GenderEnum[] values();

public static com.yuanxz.example.GenderEnum valueOf(java.lang.String);

}

翻译成java类如下：

由此可以看出，enum中的每个元素都相当于enum这个类的一个实例，如果这个类有公共方法，可以通过这个实例直接调用。

public final class com.yuanxz.example.GenderEnum extends java.lang.Enum<com.yuanxz.example.GenderEnum> {

public static final com.yuanxz.example.GenderEnum MALE;

public static final com.yuanxz.example.GenderEnum FEMALE;

private static final com.yuanxz.example.GenderEnum[] VALUES;

static {

MALE = new com.yuanxz.example.GenderEnum("MALE", 0);

FEMALE = new com.yuanxz.example.GenderEnum("FEMALE", 1);

VALUES = new GenderEnum[2];

VALUES[0] = MALE;

VALUES[1] = FEMALE;

}

public static com.yuanxz.example.GenderEnum[] values() {

com.yuanxz.example.GenderEnum[] values = new com.yuanxz.example.GenderEnum[VALUES.length];

System.arraycopy(VALUES, 0, values, 0, VALUES.length);

return values;

}

public static com.yuanxz.example.GenderEnum valueOf(java.lang.String name) {

return Enum.valueOf(com.yuanxz.example.GenderEnum, name);

}

}

### Enum的优缺点是什么？

## Volatitle：

### .volatile关键字的两层语义

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

2）禁止进行指令重排序。

先看一段代码，假如线程1先执行，线程2后执行：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | //线程1  boolean stop = false;  while(!stop){      doSomething();  }    //线程2  stop = true; |

这段代码是很典型的一段代码，很多人在中断线程时可能都会采用这种标记办法。但是事实上，这段代码会完全运行正确么？即一定会将线程中断么？不一定，也许在大多数时候，这个代码能够把线程中断，但是也有可能会导致无法中断线程（虽然这个可能性很小，但是只要一旦发生这种情况就会造成死循环了）。

下面解释一下这段代码为何有可能导致无法中断线程。在前面已经解释过，每个线程在运行过程中都有自己的工作内存，那么线程1在运行的时候，会将stop变量的值拷贝一份放在自己的工作内存当中。

那么当线程2更改了stop变量的值之后，但是还没来得及写入主存当中，线程2转去做其他事情了，那么线程1由于不知道线程2对stop变量的更改，因此还会一直循环下去。

但是用volatile修饰之后就变得不一样了：

第一：使用volatile关键字会强制将修改的值立即写入主存；

第二：使用volatile关键字的话，当线程2进行修改时，会导致线程1的工作内存中缓存变量stop的缓存行无效（反映到硬件层的话，就是CPU的L1或者L2缓存中对应的缓存行无效）；

第三：由于线程1的工作内存中缓存变量stop的缓存行无效，所以线程1再次读取变量stop的值时会去主存读取。

那么在线程2修改stop值时（当然这里包括2个操作，修改线程2工作内存中的值，然后将修改后的值写入内存），会使得线程1的工作内存中缓存变量stop的缓存行无效，然后线程1读取时，发现自己的缓存行无效，它会等待缓存行对应的主存地址被更新之后，然后去对应的主存读取最新的值。

那么线程1读取到的就是最新的正确的值。

### volatile保证原子性吗？

从上面知道volatile关键字保证了操作的可见性，但是volatile能保证对变量的操作是原子性吗？

下面看一个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class Test {      public volatile int inc = 0;        public void increase() {          inc++;      }        public static void main(String[] args) {          final Test test = new Test();          for(int i=0;i<10;i++){              new Thread(){                  public void run() {                      for(int j=0;j<1000;j++)                          test.increase();                  };              }.start();          }            while(Thread.activeCount()>1)  //保证前面的线程都执行完              Thread.yield();          System.out.println(test.inc);      }  } |

大家想一下这段程序的输出结果是多少？也许有些朋友认为是10000。但是事实上运行它会发现每次运行结果都不一致，都是一个小于10000的数字。

可能有的朋友就会有疑问，不对啊，上面是对变量inc进行自增操作，由于volatile保证了可见性，那么在每个线程中对inc自增完之后，在其他线程中都能看到修改后的值啊，所以有10个线程分别进行了1000次操作，那么最终inc的值应该是1000\*10=10000。

这里面就有一个误区了，volatile关键字能保证可见性没有错，但是上面的程序错在没能保证原子性。可见性只能保证每次读取的是最新的值，但是volatile没办法保证对变量的操作的原子性。

在前面已经提到过，自增操作是不具备原子性的，它包括读取变量的原始值、进行加1操作、写入工作内存。那么就是说自增操作的三个子操作可能会分割开执行，就有可能导致下面这种情况出现：

假如某个时刻变量inc的值为10，

线程1对变量进行自增操作，线程1先读取了变量inc的原始值，然后线程1被阻塞了；

然后线程2对变量进行自增操作，线程2也去读取变量inc的原始值，由于线程1只是对变量inc进行读取操作，而没有对变量进行修改操作，所以不会导致线程2的工作内存中缓存变量inc的缓存行无效，所以线程2会直接去主存读取inc的值，发现inc的值时10，然后进行加1操作，并把11写入工作内存，最后写入主存。

然后线程1接着进行加1操作，由于已经读取了inc的值，注意此时在线程1的工作内存中inc的值仍然为10，所以线程1对inc进行加1操作后inc的值为11，然后将11写入工作内存，最后写入主存。

那么两个线程分别进行了一次自增操作后，inc只增加了1。

解释到这里，可能有朋友会有疑问，不对啊，前面不是保证一个变量在修改volatile变量时，会让缓存行无效吗？然后其他线程去读就会读到新的值，对，这个没错。这个就是上面的happensbefore规则中的volatile变量规则，但是要注意，线程1对变量进行读取操作之后，被阻塞了的话，并没有对inc值进行修改。然后虽然volatile能保证线程2对变量inc的值读取是从内存中读取的，但是线程1没有进行修改，所以线程2根本就不会看到修改的值。

根源就在这里，自增操作不是原子性操作，而且volatile也无法保证对变量的任何操作都是原子性的。

把上面的代码改成以下任何一种都可以达到效果：

采用synchronized：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class Test {      public  int inc = 0;        public synchronized void increase() {          inc++;      }        public static void main(String[] args) {          final Test test = new Test();          for(int i=0;i<10;i++){              new Thread(){                  public void run() {                      for(int j=0;j<1000;j++)                          test.increase();                  };              }.start();          }            while(Thread.activeCount()>1)  //保证前面的线程都执行完              Thread.yield();          System.out.println(test.inc);      }  } |

采用Lock：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class Test {      public  int inc = 0;      Lock lock = new ReentrantLock();        public  void increase() {          lock.lock();          try {              inc++;          } finally{              lock.unlock();          }      }        public static void main(String[] args) {          final Test test = new Test();          for(int i=0;i<10;i++){              new Thread(){                  public void run() {                      for(int j=0;j<1000;j++)                          test.increase();                  };              }.start();          }            while(Thread.activeCount()>1)  //保证前面的线程都执行完              Thread.yield();          System.out.println(test.inc);      }  } |

采用AtomicInteger：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class Test {      public  AtomicInteger inc = new AtomicInteger();        public  void increase() {          inc.getAndIncrement();      }        public static void main(String[] args) {          final Test test = new Test();          for(int i=0;i<10;i++){              new Thread(){                  public void run() {                      for(int j=0;j<1000;j++)                          test.increase();                  };              }.start();          }            while(Thread.activeCount()>1)  //保证前面的线程都执行完              Thread.yield();          System.out.println(test.inc);      }  } |

在java 1.5的java.util.concurrent.atomic包下提供了一些原子操作类，即对基本数据类型的 自增（加1操作），自减（减1操作）、以及加法操作（加一个数），减法操作（减一个数）进行了封装，保证这些操作是原子性操作。atomic是利用CAS来实现原子性操作的（Compare And Swap），CAS实际上是利用处理器提供的CMPXCHG指令实现的，而处理器执行CMPXCHG指令是一个原子性操作。

### volatile能保证有序性吗？

在前面提到volatile关键字能禁止指令重排序，所以volatile能在一定程度上保证有序性。

volatile关键字禁止指令重排序有两层意思：

1）当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；

2）在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。

可能上面说的比较绕，举个简单的例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | //x、y为非volatile变量  //flag为volatile变量    x = 2;        //语句1  y = 0;        //语句2  flag = true;  //语句3  x = 4;         //语句4  y = 1;       //语句5 |

由于flag变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句3放到语句1、语句2前面，也不会讲语句3放到语句4、语句5后面。但是要注意语句1和语句2的顺序、语句4和语句5的顺序是不作任何保证的。

并且volatile关键字能保证，执行到语句3时，语句1和语句2必定是执行完毕了的，且语句1和语句2的执行结果对语句3、语句4、语句5是可见的。

那么我们回到前面举的一个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | //线程1:  context = loadContext();   //语句1  inited = true;             //语句2    //线程2:  while(!inited ){    sleep()  }  doSomethingwithconfig(context); |

前面举这个例子的时候，提到有可能语句2会在语句1之前执行，那么久可能导致context还没被初始化，而线程2中就使用未初始化的context去进行操作，导致程序出错。

这里如果用volatile关键字对inited变量进行修饰，就不会出现这种问题了，因为当执行到语句2时，必定能保证context已经初始化完毕。

### volatile的原理和实现机制

前面讲述了源于volatile关键字的一些使用，下面我们来探讨一下volatile到底如何保证可见性和禁止指令重排序的。

下面这段话摘自《深入理解Java虚拟机》：

“观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，加入volatile关键字时，会多出一个lock前缀指令”

lock前缀指令实际上相当于一个内存屏障（也成内存栅栏），内存屏障会提供3个功能：

1）它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；

2）它会强制将对缓存的修改操作立即写入主存；

3）如果是写操作，它会导致其他CPU中对应的缓存行无效。

### 使用volatile关键字的场景

synchronized关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而volatile关键字在某些情况下性能要优于synchronized，但是要注意volatile关键字是无法替代synchronized关键字的，因为volatile关键字无法保证操作的原子性。通常来说，使用volatile必须具备以下2个条件：

1）对变量的写操作不依赖于当前值

2）该变量没有包含在具有其他变量的不变式中

## 反射：

### – 什么是反射？

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

要想解剖一个类,必须先要获取到该类的字节码文件对象。而解剖使用的就是Class类中的方法.所以先要获取到每一个字节码文件对应的Class类型的对象.

– 反射的特性有哪些？  
– 为什么要使用反射？  
– 如何使用反射？  
– 反射的实现原理是什么？  
– 反射的优缺点是什么？

### **ECLISPE可以直接创建Servlet类。**

– Servlet的实现原理是什么？  
– Servlet的优缺点是什么？  
– 参考资料

## Git:

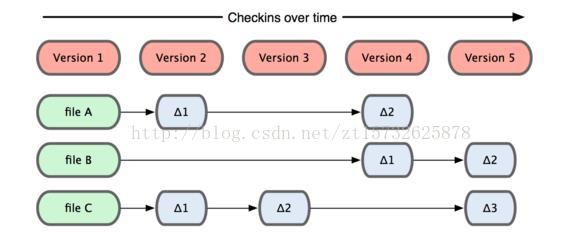
### – 什么是Git？

Git是一个开源的分布式版本控制系统，用于敏捷高效地处理任何或小或大的项目。

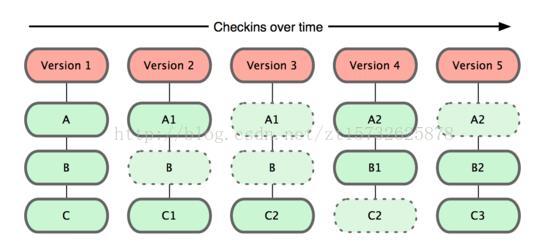
### – Git的特性有哪些？

#### 1. 直接记录快照，而非差异比较

            git 关心的只是文件数据的整体是否发生变化，而大多数其他系统（比如svn）则只关心文件内容的具体差异，每次记录都有哪些文件做了更新，以及都更新了哪些行的什么内容。工作方式参见下图 。



      git  并不保存这些前后变化的差异数据，保存的是文件的快照。实际上，git更像是把变化的文件做快照后，记录在一个微型的文件系统中。每次提交更新时，它会纵览一遍所有文件的指纹信息并对文件作一快照，然后保存一个指向这次快照的索引，为提高性能，若文件没有变化，git不会再次保存，而只对上次保存的快照作一链接，git的工作方式如下图所示：



#### 2.近乎所有操作都是本地执行

            在git中的绝大多数操作都只需要访问本地文件和资源，不用联网，因为它有本地数据库的镜像。但如果用集中式版本控制系统的话，差不多所有操作都需要联网，因为git 在本地磁盘上就保存着所有当前项目的历史更新，所以处理起来速度飞快。

           当你使用git的时候，你可以在飞机上或者火车上，频繁地提交更新，等到有网络了再上传到远程仓库，同样，在家里，你也可以继续工作，等到回到公司有网的时候，再推送到远程仓库中；当你使用svn的时候就没有这么爽快了， 虽然可以编辑文件，但是在进行文件提交和更新的时候就需要网络的支持，在实际体验之后，你会很享受git带来的不同。

#### 3. 时刻保持数据完整性

          在保存到git 之前，所有数据都要进行内容的校验和计算，并将此结果作为数据的唯一标识和索引。换句话说，不可能在你修改了文件或目录之后，git 一无所知。这项特性作为git 的设计哲学，建在整体架构的最底层。所以如果文件在传输时变得不完整，或者磁盘损坏导致文件数据缺失，git都能立即察觉。

         git  使用SHA1 算法计算数据的校验和的机制叫做SHA1 散列。这时一个由40 个十六进制字符（09和 af ）组成字符串， 基于git中文件的内容或目录结构计算出来。SHA1哈希看起来是这样：

http://img.blog.csdn.net/20170323193247705?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvenQxNTczMjYyNTg3OA==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

            Git中使用这种哈希值的情况很多，你将经常看到这种哈希值。实际上，Git数据库中保存的信息都是以文件内容的哈希值来索引，而不是文件名。

【总结】

          作为新时代的产品，git 还有许多先进的特点，最好的理解就是去了解它，使用它，你会慢慢喜欢上它带来的体验的！！！

– 为什么要使用Git？

### – 如何使用Git？

|  |  |
| --- | --- |
| 新建代码库 | git init |
| 克隆项目 | git clone |
| 显示当前git配置信息 | git config list |
| 配置 | git config global user.name "\*\*\*" |
| 配置 | git config global user.email "\*\*\*" |
| 添加当前目录all文件 | git add . |
| 添加制定file 2 暂存区 | git add file |
| 删除工作区文件 | git rm file |
| 提交代码 | git commit m 'message' |
| 强制提交 | git commit am 'message' |
| 查看文件变化 | git diff file |
| 显示修改过的文件 | git status |
| get rmt仓库与loc合并 | git pull |
| 推送代码 | git push |
| 列出本地所有分支 | git branch |
| 远程分支查看 | git branch r |
| show rmt and loc 分支 | git branch a |
| 新建分支 | git branch name |
| 切换分支 | git checkout branchname |
| 创建并切换分支 | git checkout b branchname |
| mge指定brch 2 cubrch | git merge branch |
| 删除分支 | git branch d branchname |
| 删除远程分支 | git push origin :branchname |

### – Git的实现原理是什么？

### – Git的优缺点是什么？

优点：

* 比svn更快速的分支切换，因为分支都在本地。这使得git特别适合处理临时需求，或者小实验。
* 可以在星巴克写代码的时候随时提交，因为各版本快照（commit）都在本地
* 有一个全世界最大的开源社区——github，上面可以看到各种优秀的各语言代码，去官网语言列表看看才知道有那么多语言，就跟奥运会开幕式才知道世界上有那么多国家一样

缺点：

* 因为是分布式的，每台机器上都commit的时候，有的时候忘记了push到远程服务器（github），那就不同步了。不过常常status一下看看状态，问题不大
* 要使用命令行（我在windows下装的git bash），其实用gui也是可以的，但基本上还是命令行更灵活，而且有一本专业开源教程[《Pro Git（中文）》](http://git-scm.com/2010/06/09/pro-git-zh.html)，而且如果遇到问题可以在stackoverflow上去搜解决方案，答案都是命令行

– 参考资料

### Spring:

1.什么是Spring？

Spring是一个开源的轻量级，企业级应用程序开发框架。

2.什么是控制反转（IOC）。

3.什么是AOP

3.什么是spring 事务。

## Redis:

### – 什么是Redis？

redis是一个开源的、使用C语言编写的、支持网络交互的、可基于内存也可持久化的KeyValue数据库。

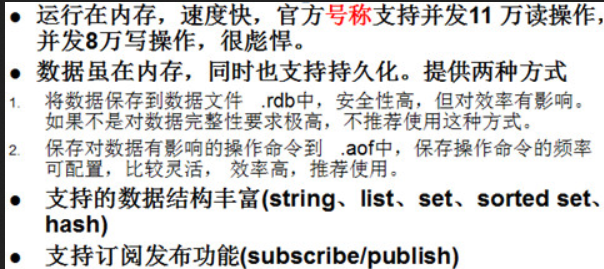
### –Redis的特性有哪些？

1、速度快   
Redis是用C语言实现的；   
Redis的所有数据存储在内存中。   
2、持久化   
Redis的所有数据存储在内存中，对数据的更新将异步地保存到磁盘上。   
3、支持多种数据结构   
Redis支持五种数据结构：String、List、Set、Hash、Zset   
4、支持多种编程语言   
Java、php、Python、Ruby、Lua、Node.js   
5、功能丰富   
除了支持五种数据结构之外，还支持事务、流水线、发布/订阅、消息队列等功能。   
6、源码简单   
约23000行C语言源代码。   
7、主从复制   
主服务器（master）执行添加、修改、删除，从服务器执行查询。   
8、高可用及分布式   
RedisSentinel（v2.8）支持高可用   
RedisCluster（v3.0）支持分布式

什么是高可用？   
一台服务器挂掉会有另一台服务器立马顶上，RedisSentinel监听所有的监听所有的Redis主从服务器，一旦发现主服务器挂掉了，他会从若干服务器中选择一台新的机器作为master，其他从服务器会成为这台新master的从服务器。

什么是分布式？   
在一台物理计算机上，把数据A放到数据库D1，把数据B放到数据库D2，把数据C放到数据库D3属于分布式，但由于数据库的访问连接池是有限的，所以这种分布式解决方案存在性能瓶颈；   
把数据A放到计算机C1的数据库D1，把数据B放到计算机C1的数据库D2，把数据C放到计算机C3的数据库D3也属于分布式，这种分布式解决方案扩展性好，如果系统增加了新业务X，X产生的数据Y可以放到计算机C4的数据库D4中，理论上来说这种方式可以无限扩展，从而保证系统的性能最大化。   
存在瓶颈且价格较高的高性能单台服务器>可无限扩展的分布式集群

### – 为什么要使用Redis？



– 如何使用Redis？

Redis 数据类型

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)。

String（字符串）

string是redis最基本的类型，你可以理解成与Memcached一模一样的类型，一个key对应一个value。

string类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象 。

string类型是Redis最基本的数据类型，一个键最大能存储512MB。

实例

redis 127.0.0.1:6379> SET name "runoob"

OK

redis 127.0.0.1:6379> GET name

"runoob"

在以上实例中我们使用了 Redis 的 **SET** 和 **GET** 命令。键为 name，对应的值为 **runoob**。

**注意：**一个键最大能存储512MB。

Hash（哈希）

Redis hash 是一个键值(key=>value)对集合。

Redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象。

实例

redis> HMSET myhash field1 "Hello" field2 "World"

"OK"

redis> HGET myhash field1

"Hello"

redis> HGET myhash field2

"World"

以上实例中 hash 数据类型存储了包含用户脚本信息的用户对象。 实例中我们使用了 Redis **HMSET, HGETALL** 命令，**user:1** 为键值。

每个 hash 可以存储 232 1 键值对（40多亿）。

List（列表）

Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）。

实例

redis 127.0.0.1:6379> lpush runoob redis

(integer) 1

redis 127.0.0.1:6379> lpush runoob mongodb

(integer) 2

redis 127.0.0.1:6379> lpush runoob rabitmq

(integer) 3

redis 127.0.0.1:6379> lrange runoob 0 10

1) "rabitmq"

2) "mongodb"

3) "redis"

redis 127.0.0.1:6379>

列表最多可存储 232 1 元素 (4294967295, 每个列表可存储40多亿)。

Set（集合）

Redis的Set是string类型的无序集合。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

sadd 命令

添加一个string元素到,key对应的set集合中，成功返回1,如果元素已经在集合中返回0,key对应的set不存在返回错误。

sadd key member

实例

redis 127.0.0.1:6379> sadd runoob redis

(integer) 1

redis 127.0.0.1:6379> sadd runoob mongodb

(integer) 1

redis 127.0.0.1:6379> sadd runoob rabitmq

(integer) 1

redis 127.0.0.1:6379> sadd runoob rabitmq

(integer) 0

redis 127.0.0.1:6379> smembers runoob

1) "rabitmq"

2) "mongodb"

3) "redis"

**注意：**以上实例中 rabitmq 添加了两次，但根据集合内元素的唯一性，第二次插入的元素将被忽略。

集合中最大的成员数为 232 1(4294967295, 每个集合可存储40多亿个成员)。

zset(sorted set：有序集合)

Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

zset的成员是唯一的,但分数(score)却可以重复。

zadd 命令

添加元素到集合，元素在集合中存在则更新对应score

zadd key score member

实例

redis 127.0.0.1:6379> zadd runoob 0 redis

(integer) 1

redis 127.0.0.1:6379> zadd runoob 0 mongodb

(integer) 1

redis 127.0.0.1:6379> zadd runoob 0 rabitmq

(integer) 1

redis 127.0.0.1:6379> zadd runoob 0 rabitmq

(integer) 0

redis 127.0.0.1:6379> ZRANGEBYSCORE runoob 0 1000

1) "redis"

2) "mongodb"

3) "rabitmq"

### –Redis的实现原理是什么？

### –Redis的优缺点是什么？

**1. 使用redis有哪些好处？**

(1) 速度快，因为数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)

(2) 支持丰富数据类型，支持string，list，set，sorted set，hash

(3) 支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行

(4) 丰富的特性：可用于缓存，消息，按key设置过期时间，过期后将会自动删除

**2. redis相比memcached有哪些优势？**

(1) memcached所有的值均是简单的字符串，redis作为其替代者，支持更为丰富的数据类型

(2) redis的速度比memcached快很多

(3) redis可以持久化其数据

**3. redis常见性能问题和解决方案：**

(1) Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件

(2) 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次

(3) 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内

(4) 尽量避免在压力很大的主库上增加从库

(5) 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master < Slave1 < Slave2 < Slave3...

这样的结构方便解决单点故障问题，实现Slave对Master的替换。如果Master挂了，可以立刻启用Slave1做Master，其他不变。

– 参考资料

## [Java并发编程：Callable、Future和FutureTask](http://www.cnblogs.com/dolphin0520/p/3949310.html)

### 一.Callable与Runnable

先说一下java.lang.Runnable吧，它是一个接口，在它里面只声明了一个run()方法：

|  |  |
| --- | --- |
| 1  2  3 | public interface Runnable {  public abstract void run();  } |

 　　由于run()方法返回值为void类型，所以在执行完任务之后无法返回任何结果。

　　Callable位于java.util.concurrent包下，它也是一个接口，在它里面也只声明了一个方法，只不过这个方法叫做call()：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public interface Callable<V> {      /\*\*       \* Computes a result, or throws an exception if unable to do so.       \*       \* @return computed result       \* @throws Exception if unable to compute a result       \*/      V call() throws Exception;  } |

 　　可以看到，这是一个泛型接口，call()函数返回的类型就是传递进来的V类型。

　　那么怎么使用Callable呢？一般情况下是配合ExecutorService来使用的，在ExecutorService接口中声明了若干个submit方法的重载版本：

|  |  |
| --- | --- |
| 1  2  3 | <T> Future<T> submit(Callable<T> task);  <T> Future<T> submit(Runnable task, T result);  Future<?> submit(Runnable task); |

　　第一个submit方法里面的参数类型就是Callable。

　　暂时只需要知道Callable一般是和ExecutorService配合来使用的，具体的使用方法讲在后面讲述。

　　一般情况下我们使用第一个submit方法和第三个submit方法，第二个submit方法很少使用。

### 二.Future

　　Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过get方法获取执行结果，该方法会阻塞直到任务返回结果。

　　Future类位于java.util.concurrent包下，它是一个接口：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public interface Future<V> {      boolean cancel(boolean mayInterruptIfRunning);      boolean isCancelled();      boolean isDone();      V get() throws InterruptedException, ExecutionException;      V get(long timeout, TimeUnit unit)          throws InterruptedException, ExecutionException, TimeoutException;  } |

 　　在Future接口中声明了5个方法，下面依次解释每个方法的作用：

* cancel方法用来取消任务，如果取消任务成功则返回true，如果取消任务失败则返回false。参数mayInterruptIfRunning表示是否允许取消正在执行却没有执行完毕的任务，如果设置true，则表示可以取消正在执行过程中的任务。如果任务已经完成，则无论mayInterruptIfRunning为true还是false，此方法肯定返回false，即如果取消已经完成的任务会返回false；如果任务正在执行，若mayInterruptIfRunning设置为true，则返回true，若mayInterruptIfRunning设置为false，则返回false；如果任务还没有执行，则无论mayInterruptIfRunning为true还是false，肯定返回true。
* isCancelled方法表示任务是否被取消成功，如果在任务正常完成前被取消成功，则返回 true。
* isDone方法表示任务是否已经完成，若任务完成，则返回true；
* get()方法用来获取执行结果，这个方法会产生阻塞，会一直等到任务执行完毕才返回；
* get(long timeout, TimeUnit unit)用来获取执行结果，如果在指定时间内，还没获取到结果，就直接返回null。

　　也就是说Future提供了三种功能：

　　1）判断任务是否完成；

　　2）能够中断任务；

　　3）能够获取任务执行结果。

　　因为Future只是一个接口，所以是无法直接用来创建对象使用的，因此就有了下面的FutureTask。

### 三.FutureTask

　　我们先来看一下FutureTask的实现：

|  |  |
| --- | --- |
| 1 | public class FutureTask<V> implements RunnableFuture<V> |

 　　FutureTask类实现了RunnableFuture接口，我们看一下RunnableFuture接口的实现：

|  |  |
| --- | --- |
| 1  2  3 | public interface RunnableFuture<V> extends Runnable, Future<V> {      void run();  } |

 　　可以看出RunnableFuture继承了Runnable接口和Future接口，而FutureTask实现了RunnableFuture接口。所以它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值。

　　FutureTask提供了2个构造器：

|  |  |
| --- | --- |
| 1  2  3  4 | public FutureTask(Callable<V> callable) {  }  public FutureTask(Runnable runnable, V result) {  } |

　　事实上，FutureTask是Future接口的一个唯一实现类。

### 四.使用示例

　　1.使用Callable+Future获取执行结果

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | public class Test {      public static void main(String[] args) {          ExecutorService executor = Executors.newCachedThreadPool();          Task task = new Task();          Future<Integer> result = executor.submit(task);          executor.shutdown();            try {              Thread.sleep(1000);          } catch (InterruptedException e1) {              e1.printStackTrace();          }            System.out.println("主线程在执行任务");            try {              System.out.println("task运行结果"+result.get());          } catch (InterruptedException e) {              e.printStackTrace();          } catch (ExecutionException e) {              e.printStackTrace();          }            System.out.println("所有任务执行完毕");      }  }  class Task implements Callable<Integer>{      @Override      public Integer call() throws Exception {          System.out.println("子线程在进行计算");          Thread.sleep(3000);          int sum = 0;          for(int i=0;i<100;i++)              sum += i;          return sum;      }  } |

 　　执行结果：

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifView Code

　　2.使用Callable+FutureTask获取执行结果

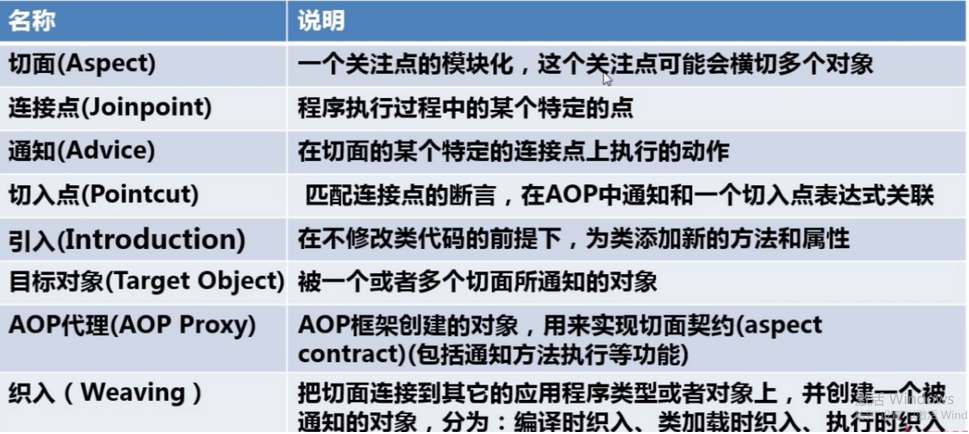
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | public class Test {      public static void main(String[] args) {          //第一种方式          ExecutorService executor = Executors.newCachedThreadPool();          Task task = new Task();          FutureTask<Integer> futureTask = new FutureTask<Integer>(task);          executor.submit(futureTask);          executor.shutdown();            //第二种方式，注意这种方式和第一种方式效果是类似的，只不过一个使用的是ExecutorService，一个使用的是Thread          /\*Task task = new Task();          FutureTask<Integer> futureTask = new FutureTask<Integer>(task);          Thread thread = new Thread(futureTask);          thread.start();\*/            try {              Thread.sleep(1000);          } catch (InterruptedException e1) {              e1.printStackTrace();          }            System.out.println("主线程在执行任务");            try {              System.out.println("task运行结果"+futureTask.get());          } catch (InterruptedException e) {              e.printStackTrace();          } catch (ExecutionException e) {              e.printStackTrace();          }            System.out.println("所有任务执行完毕");      }  }  class Task implements Callable<Integer>{      @Override      public Integer call() throws Exception {          System.out.println("子线程在进行计算");          Thread.sleep(3000);          int sum = 0;          for(int i=0;i<100;i++)              sum += i;          return sum;      }  } |

## SpringAOP；

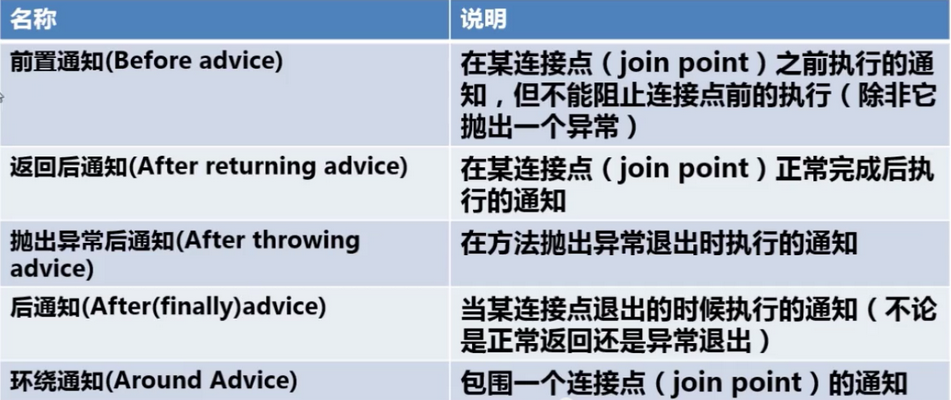
### 什么是springaop？

AOP（Aspect Oriented Programming），即面向切面编程，可以说是OOP（Object Oriented Programming，面向对象编程）的补充和完善。OOP引入封装、继承、多态等概念来建立一种对象层次结构，用于模拟公共行为的一个集合。不过OOP允许开发者定义纵向的关系，但并不适合定义横向的关系，例如日志功能。日志代码往往横向地散布在所有对象层次中，而与它对应的对象的核心功能毫无关系对于其他类型的代码，如安全性、异常处理和透明的持续性也都是如此，这种散布在各处的无关的代码被称为横切（cross cutting），在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。  
–springaop的特性有哪些？

AOP的几个相关概念：



Advice的类型：



– 为什么要使用springaop？

### 如何使用springaop？

第一步：

定义一个目标类

第二步，

定义一个切面类，

第三步配置AOP设置

1）定义一个切面

2）定义一个切点方法

3）在切面的切点方法上执行对应的通知。

|  |
| --- |
| <!-- 目标类 -->  <bean id=*"userManager"* class=*"com.learn.service.UserManagerServiceImpl"* />  <!-- 切面类 -->  <bean id=*"aspetBean"* class=*"com.learn.aop.InterceptorAop"* />  <aop:config>  <aop:aspect id=*"aspet"* ref=*"aspetBean"*>  <aop:pointcut expression=*"execution (\* com.learn.service..\*.\*(..))"*  id=*"cut"* />  <!-- Before:在目标方法被调用之前做增强处理,@Before只需要指定切入点表达式即可 -->  <aop:before pointcut-ref=*"cut"* method=*"doBefore"* />  <!-- AfterReturning:在目标方法正常完成后做增强,@AfterReturning除了指定切入点表达式后，  还可以指定一个返回值形参名returning,代表目标方法的返回值 -->  <aop:after-returning pointcut-ref=*"cut"*  method=*"doAferReturning"* />  <!-- After:在目标方法完成之后做增强，无论目标方法是否成功完成。@After可以指定一个切入点表达式 -->  <aop:after pointcut-ref=*"cut"* method=*"doAfter"* />  <!-- AfterThrowing:主要用来处理程序中未处理的异常,@AfterThrowing除了指定切入点表达式后，  还可以指定一个throwing的返回值形参名,可以通过该形参名来访问目标方法中所抛出的异常对象 -->  <aop:after-throwing pointcut-ref=*"cut"* method=*"doAferThrowing"* />  <!-- Around:环绕通知,在目标方法完成前后做增强处理,环绕通知是最重要的通知类型,  像事务,日志等都是环绕通知,注意编程中核心是一个ProceedingJoinPoint -->  <aop:around pointcut-ref=*"cut"* method=*"doAround"* />  </aop:aspect>  </aop:config> |

AOP编程其实是很简单的事情，纵观AOP编程，程序员只需要参与三个部分：

1、定义普通业务组件

2、定义切入点，一个切入点可能横切多个业务组件

3、定义增强处理，增强处理就是在AOP框架为普通业务组件织入的处理动作

所以进行AOP编程的关键就是定义切入点和定义增强处理，一旦定义了合适的切入点和增强处理，AOP框架将自动生成AOP代理，即：**代理对象的方法=增强处理+被代理对象**的方法。

### springaop的实现原理是什么？

说AOP原理三句话就能概括：

1. 对类生成代理使用CGLIB
2. 对接口生成代理使用JDK原生的Proxy
3. 可以通过配置文件指定对接口使用CGLIB生成代理

与AspectJ的静态代理不同，Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

Spring AOP中的动态代理主要有两种方式，

1.JDK动态代理：JDK动态代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK动态代理的核心是InvocationHandler接口和Proxy类。

2. CGLIB动态代理：如果目标类没有实现接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成某个类的子类，注意，CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

### springaop的优缺点是什么？

### 参考资料

## Spring事务；

### 什么是Spring事务？

Spring有自己的事务管理机制，一般是使用TransactionMananger进行管理，可以通过Spring的注入来完成此功能。

Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。对于纯JDBC操作数据库，想要用到事务，可以按照以下步骤进行：

1. 获取连接 Connection con = DriverManager.getConnection()
2. 开启事务con.setAutoCommit(true/false);
3. 执行CRUD
4. 提交事务/回滚事务 con.commit() / con.rollback();
5. 关闭连接 conn.close();

使用Spring的事务管理功能后，我们可以不再写步骤 2 和 4 的代码，而是由Spirng 自动完成

### Spring事务的特性有哪些？

事务必须服从ACID原则。ACID指的是原子性（atomicity）、一致性（consistency）、隔离性（isolation）和持久性（durability）。

通俗理解，事务其实就是一系列指令的集合。

事务的四大特征

1.原子性：一个事务中所有对数据库的操作是一个不可分割的操作序列，要么全做要么全不做

2.一致性：数据不会因为事务的执行而遭到破坏

3.隔离性：一个事物的执行，不受其他事务的干扰，即并发执行的事物之间互不干扰

4.持久性：一个事物一旦提交，它对数据库的改变就是永久的

### 为什么要使用Spring事务？

### 如何使用Spring事务？

#### 一、Spring 事务的传播属性

所谓spring事务的传播属性，就是定义在存在多个事务同时存在的时候，spring应该如何处理这些事务的行为。这些属性在TransactionDefinition中定义，具体常量的解释见下表：



**1. Propagation.REQUIRED：**如果当前存在事务，使用当期事务，如果不存在，创建一个事务运行。

**2. Propagation.REQUIRES\_NEW：**如果当前存在事务，**当前事务挂起，创建一个新事务运行**。

**3. Propagation.SUPPORTS**：如果当前存在事务，使用当前事务，如果不存在，以非事务运行。

**4. Propagation.NOT\_SUPPORTED：**如果当前存在事务，**将当前事务挂起，以非事务运行。总是以非事务运行。**

**5. Propagation.MANDATORY：**当前存在事务，使用当前事务，**如果不存在事务，则抛异常**，总是以事务运行。

**6. Propagation.NEVER：如果当前存在事务，则抛异常**，总是以非事务运行

**7、Propagation.NESTED**如果当前存在事务，则运行在一个嵌套的事务中. 如果没有事务, 则按 TransactionDefinition.PROPAGATION\_REQUIRED 属性执行。需要JDBC3.0以上支持。

#### 二、数据库隔离级别



**脏读：**事务1对数据进行了增删改，但未提交，事务2可以读取到未提交的数据。如果事务1这时候回滚了，那么第二个事务就读到了脏数据。**强调：事务1对数据进行增删改操作**

事务T1：更新一条数据  
          -->事务T2：读取事务T1更新的记录  
 事务T1：调用rollback数据

**不可重复读：**事务1中发生了两次读操作，第一次读操作和第二次操作之间，事务2对数据进行了修改，这时候两次读取的数据是不一致的。**强调：事务1对数据的查询的内容**

事务T1：查询一条记录  
         -->事务T2：更新事务T1查询的记录  
         -->事务T2：调用commit进行提交  
事务T1：再次查询上次的记录

**幻读：**第一个事务对一定范围的数据进行批量修改，第二个事务在这个范围增加一条数据，这时候第一个事务就会丢失对新增数据的修改，数目不一样。**强调：事务1对数据的查询的条数**

事务T1：查询表中所有记录  
          -->事务T2：插入一条记录  
          -->事务T2：调用commit进行提交  
事务T1：再次查询表中所有记录

**总结：**

隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大。

大多数的数据库默认隔离级别为 Read Commited，比如 SqlServer、Oracle

少数数据库默认隔离级别为：Repeatable Read 比如： MySQL InnoDB

#### 三、Spring中的隔离级别



|  |  |  |  |
| --- | --- | --- | --- |
|  | **脏读** (Dirty reads） | **不可重复读** (non-repeatable reads ) | **幻读** (phantom reads) |
| **read uncommitted** | 会 | 会 | 会 |
| **read committed** | 不会 | 会 | 会 |
| **repeatable read** | 不会 | 不会 | 会 |
| **serializable** | 不会 | 不会 | 不会 |

#### 四、事务API

最重要的有三个API：TransactionDefinition、PlatformTransactionManager、TransactionStatus。所谓事务管理，

TransactionDefinition 表示：其实就是“按照给定的事务规则来执行提交或者回滚操作”。“给定的事务规则”就是用的，

PlatformTransactionManager 表示：“按照……来执行提交或者回滚操作”，

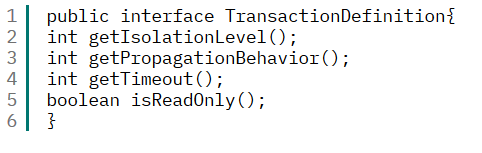
TransactionStatus 表示：一个运行着的事务的状态。

打一个不恰当的比喻，TransactionDefinition 与 TransactionStatus 的关系就像程序和进程的关系。

##### TransactionDefinition

是通过接口来定义的。该接口包含与事务属性有关的方法。具体如：

清单1. TransactionDefinition 接口中定义的主要方法

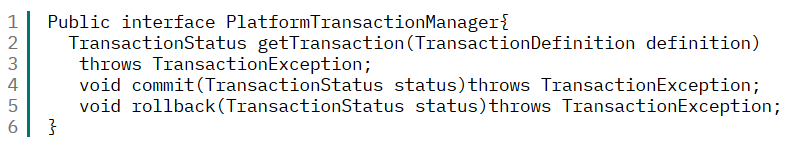


也许你会奇怪，为什么接口只提供了获取属性的方法，而没有提供相关设置属性的方法。其实道理很简单，事务属性的设置完全是程序员控制的，因此程序员可以自定义任何设置属性的方法，而且保存属性的字段也没有任何要求。唯一的要求的是，Spring 进行事务操作的时候，通过调用以上接口提供的方法必须能够返回事务相关的属性取值。

##### PlatformTransactionManager

用于执行具体的事务操作。接口定义如清单2所示：

清单2. PlatformTransactionManager 接口中定义的主要方法



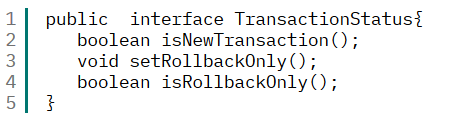
根据底层使用的不同持久化 API 或框架，PlatformTransactionManager 的主要实现类大致如下：

|  |
| --- |
|  DataSourceTransactionManager：适用于使用JDBC和iBatis进行数据持久化操作的情况。   HibernateTransactionManager：适用于使用Hibernate进行数据持久化操作的情况。   JpaTransactionManager：适用于使用JPA进行数据持久化操作的情况。   另外还有JtaTransactionManager 、JdoTransactionManager、JmsTransactionManager等等 |

##### TransactionStatus

PlatformTransactionManager.getTransaction(…) 方法返回一个 TransactionStatus 对象。返回的TransactionStatus 对象可能代表一个新的或已经存在的事务（如果在当前调用堆栈有一个符合条件的事务）。TransactionStatus 接口提供了一个简单的控制事务执行和查询事务状态的方法。该接口定义如清单3所示

清单3. TransactionStatus 接口中定义的主要方法



#### 五．编程式事务：

Spring实现编程式事务，依赖于2大类：

1）PlatformTransactionManager

根据PlatformTransactionManager、TransactionDefinition 和 TransactionStatus 三个核心接口，我们完全可以通过编程的方式来进行事务管理。

2）TransactionTemplate（推荐使用）

TransactionTemplate对事务的使用

●transactionTemplate.execute里会会通过PlatformTransactionManager.getTransaction获取一个TransactionStatus实例，在获取该实例方法中，会调用DataSourceTransactionManager的doBegin方法，该方法会执行connection.setAutoCommit(false);

●执行一组sql的操作

●执行PlatformTransactionManager.commit提交事务

●如果有异常transactionStatus.rollbackOnException回滚事务

#### 六．声明式事务

声明式事务基于AOP动态代理，不会破坏原有代码逻辑

##### 1）基于基于TransactionProxyFactoryBean的方式

|  |
| --- |
| <!-- 配置业务层代理 -->  <bean id=*"accountServiceProxy"* class=*"org.springframework.transaction.interceptor.TransactionProxyFactoryBean"*>  <!-- 配置目标对象 -->  <property name=*"target"* ref=*"accountService"*/>  <!-- 注入事务管理器 -->  <property name=*"transactionManager"* ref=*"transactionManager"*/>  <!-- 注入事务的属性 -->  <property name=*"transactionAttributes"*>  <props>  <prop key=*"transfer"*>PROPAGATION\_REQUIRED, readOnly</prop>  </props>  </property>  </bean> |

##### 2）基于AspectJ的XML方式

|  |
| --- |
| <!--  <tx:advice>定义事务通知，用于指定事务属性，其中“transaction-manager”属性指定事务管理器，相当于指定切面，  并通过<tx:attributes>指定具体需要拦截的方法  <tx:method>拦截方法，其中参数有：  name:方法名称，将匹配的方法注入事务管理，可用通配符  propagation：事务传播行为，  isolation：事务隔离级别定义；默认为“DEFAULT”  timeout：事务超时时间设置，单位为秒，默认-1，表示事务超时将依赖于底层事务系统；  read-only：事务只读设置，默认为false，表示不是只读；  rollback-for：需要触发回滚的异常定义，可定义多个，以“，”分割，默认任何RuntimeException都将导致事务回滚，而任何Checked Exception将不导致事务回滚；  no-rollback-for：不被触发进行回滚的 Exception(s)；可定义多个，以“，”分割； -->  <!-- 配置事务的增强/通知 -->  <tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>  <tx:attributes>  <tx:method name=*"transfer"* propagation=*"REQUIRED"*/>  </tx:attributes>  </tx:advice>    <!-- 配置切面 -->  <aop:config>  <!-- 配置切点 -->  <!-- 在切点当中的表达式第一个 \* 代表方法的返回值，AccountService后的+代表他的子类，最后的 \*  （..）代表任意的方法和参数。切面则由切点和增强构成，即在这个切点上运用这个增强。  代表切面的标签有两个，一个是 < aop:aspect > 代表多个切入点和多个通知，  另一个是 < aop:advisor > 代表一个切入点和一个通知，这个例题里只有一个增强“txAdvice”，  所以就使用了asvisor标签。 -->  <aop:pointcut expression=*"execution(\* com.transaction.service.AccountService+.\*(..))"* id=*"pointcut1"*/>  <!-- 配置切面 -->  <aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"pointcut1"*/>  </aop:config> |

##### 3）基于注解配置声明式事务

|  |
| --- |
| <tx:annotation-driven transaction-manager="transactionManager"/>  Java类  //基于注解--4，只能注解在方法上  @Transactional(propagation=Propagation.***REQUIRED***, isolation=Isolation.***DEFAULT***)  **public** **void** transfer( String out, String in, Double money) {  accountDao.outMoney(out, money);  //int i = 1/0;  accountDao.inMoney(in, money);  } |

### Spring事务的实现原理是什么？

### Spring事务的优缺点是什么？

## Mybatis

### – 什么是Mybatis？

mybatis是一个优秀的基于java的持久层框架，它内部封装了jdbc，使开发者只需要关注sql语句本身，而不需要花费精力去处理加载驱动、创建连接、创建statement等繁杂的过程。

mybatis通过xml或注解的方式将要执行的各种statement配置起来，并通过java对象和statement中sql的动态参数进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射为java对象并返回。  
–Mybatis的特性有哪些？

### – 为什么要使用Mybatis？ – 如何使用Mybatis？

Mybatis的使用步骤：

第一步：导入mybatis的jar包：mybatisx.x.x.jar，导入mysql/其他数据库jar包

第二步：使用XML配置mybatis，名称通常为mybatisconfig.xml，其中包含数据库连接和 读取相应的Mapper.xml， mapper.xml对应一个表的操作？。

第三步：获取sqlSession。

|  |
| --- |
| **private** SqlSession getSession() {  //1.读取配置文件sqMapConfig.xml来构建sqlSessionFactory实例。  //通过类加载器读取.xml文件  InputStream in = getClass().getClassLoader().getResourceAsStream("mybatisconfig.xml");  //2.每一个程序都是以一个sqlSessionFactory对象为核心，创建sqlSessionFactory.  SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory sqlSessionFactory = builder.build(in);  //3.创建sqlSession对象，该对象应该是线程私有的。每个方法都要创建一个，然后再finally块关闭。  **return** sqlSessionFactory.openSession();    }  @Test  **public** **void** testInsert() {  SqlSession session = getSession();  **try** {  BlogDao blogDao = session.getMapper(BlogDao.**class**);  **for**(**int** i=1; i<=5; i++) {  Blog blog = **new** Blog(i,"BBB","464646546@163.com");  blogDao.insert(blog);  session.commit();  }  System.***out***.println("插入完成");  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  session.close();  }  } |

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF8"*?>  <!DOCTYPE configuration  PUBLIC "//ibatis.apache.org//DTD Config 3.0//EN"  "http://ibatis.apache.org/dtd/ibatis3config.dtd">  <configuration>  <properties resource=*"db.properties"* />  <environments default=*"environment"*>  <environment id=*"environment"*>  <transactionManager type=*"JDBC"* />  <dataSource type=*"POOLED"*>  <property name=*"driver"* value=*"${driver}"* />  <property name=*"url"* value=*"${url}"*/>  <property name=*"username"* value=*"${user}"* />  <property name=*"password"* value=*"${pwd}"* />  </dataSource>  </environment>  </environments>  <! mappers执行Sql映射关系配置文件 >  <mappers>  <! 有多个配置文件！ >  <! 每个表操作对应一个Mapper.xml文件，每个Mapper文件用于登记sql语句 >  <mapper resource=*"BlogDao.xml"* />  </mappers> </configuration> |

创建db.property文件

|  |
| --- |
| driver=com.mysql.jdbc.Driver  url=jdbc:mysql://localhost:3306/testgrp  user=root  pwd=root |

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF8"*?>  <!DOCTYPE mapper PUBLIC "//ibatis.apache.org//DTD Mapper 3.0//EN"  "http://ibatis.apache.org/dtd/ibatis3mapper.dtd">  <mapper namespace=*"com.learn.dao.BlogDao"*>    <! 增 >  <insert id=*"insert"* parameterType=*"com.learn.entity.Blog"* >  insert into  blog (id,name,email)  values  (#{id},#{name},#{email})  </insert>    <! 删 >  <delete id=*"delete"*>  delete from blog where id = #{id}  </delete>    <! 改 >  <update id=*"update"* parameterType=*"com.learn.entity.Blog"*>  update blog  set  name = #{name},  email = #{email}  where  id = #{id}  </update>    <! 查 >  <select id=*"findById"* resultType=*"com.learn.entity.Blog"*>  select id,name,email from blog where id=#{id}  </select>    <select id=*"findAll"* resultType=*"com.learn.entity.Blog"*>  select id,name,email from blog  </select>  </mapper> |

### –Mybatis的实现原理是什么？

### 

### 

原理详解：

        MyBatis应用程序根据XML配置文件创建SqlSessionFactory，SqlSessionFactory在根据配置，配置来源于两个地方，一处是配置文件，一处是Java代码的注解，获取一个SqlSession。SqlSession包含了执行sql所需要的所有方法，可以通过SqlSession实例直接运行映射的sql语句，完成对数据的增删改查和事务提交等，用完之后关闭SqlSession。

MyBatis的优缺点

优点：

1、简单易学

       mybatis本身就很小且简单。没有任何第三方依赖，最简单安装只要两个jar文件+配置几个sql映射文件易于学习，易于使用，通过文档和源代码，可以比较完全的掌握它的设计思路和实现。

2、灵活

       mybatis不会对应用程序或者数据库的现有设计强加任何影响。 sql写在xml里，便于统一管理和优化。通过sql基本上可以实现我们不使用数据访问框架可以实现的所有功能，或许更多。

3、解除sql与程序代码的耦合

       通过提供DAL层，将业务逻辑和数据访问逻辑分离，使系统的设计更清晰，更易维护，更易单元测试。sql和代码的分离，提高了可维护性。

4、提供映射标签，支持对象与数据库的orm字段关系映射  
5、提供对象关系映射标签，支持对象关系组建维护  
6、提供xml标签，支持编写动态sql。

缺点：

1、编写SQL语句时工作量很大，尤其是字段多、关联表多时，更是如此。  
2、SQL语句依赖于数据库，导致数据库移植性差，不能更换数据库。

3、框架还是比较简陋，功能尚有缺失，虽然简化了数据绑定代码，但是整个底层数据库查询实际还是要自己写的，工作量也比较大，而且不太容易适应快速数据库修改.

4、二级缓存机制不佳

### –Mybatis的优缺点是什么？

### – 参考资料

## DB:

## Springboot

### – 什么是Springboot？

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。通过这种方式，Spring Boot致力于在蓬勃发展的快速应用开发领域（rapid application development）成为领导者。

spring大家都知道，boot是启动的意思。所以，spring boot其实就是一个启动spring项目的一个工具而已。从最根本上来讲，Spring Boot就是一些库的集合，它能够被任意项目的构建系统所使用。

### –Springboot的特性有哪些？

现在流行微服务与分布式系统，springboot就是一个非常好的微服务开发框架，你可以使用它快速的搭建起一个系统。同时，你也可以使用spring cloud（Spring Cloud是一个基于Spring Boot实现的云应用开发工具）来搭建一个分布式的网站。

### – 为什么要使用Springboot？

### – 如何使用Springboot？

### –Springboot的实现原理是什么？

### –Springboot的优缺点是什么？

4.1 使编码变得简单

spring boot采用java config的方式，对spring进行配置，并且提供了大量的注解，极大地提高了工作效率。

4.2 使配置变得简单



spring boot提供许多默认配置，当然也提供自定义配置。但是所有spring boot的项目都只有一个配置文件：application.properties/application.yml。用了spring boot，再也不用担心配置出错找不到问题所在了。

4.3 使部署变得简单

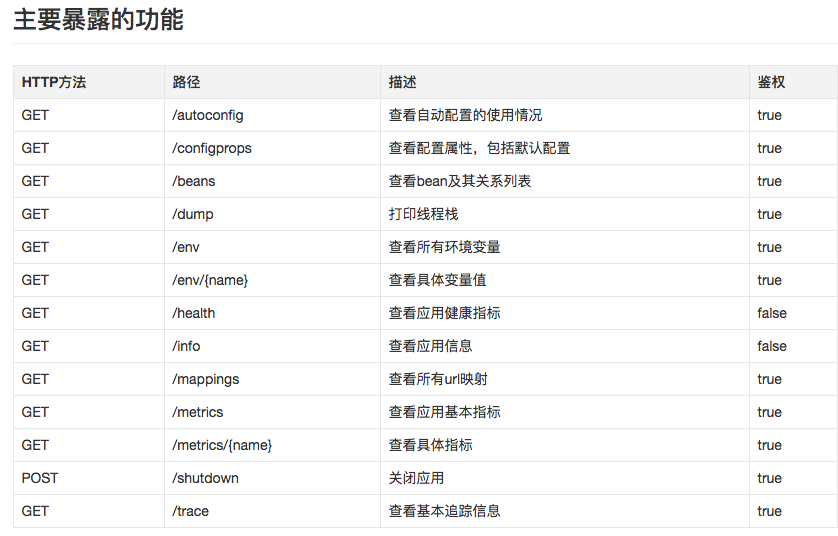
spring boot内置了三种servlet容器：tomcat，jetty，undertow。



所以，你只需要一个java的运行环境就可以跑spring boot的项目了。spring boot的项目可以打成一个jar包，然后通过java jar xxx.jar来运行。（spring boot项目的入口是一个main方法，运行该方法即可。 ）

4.4 使监控变得简单

spring boot提供了actuator包，可以使用它来对你的应用进行监控。它主要提供了以下功能：

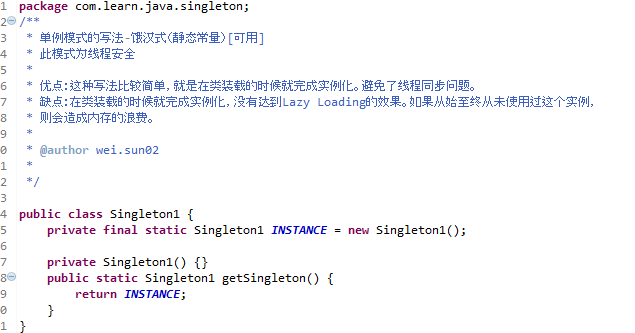


我们将在后续文章中学习它。

### – 参考资料

## 单例模式。

### 第一种：

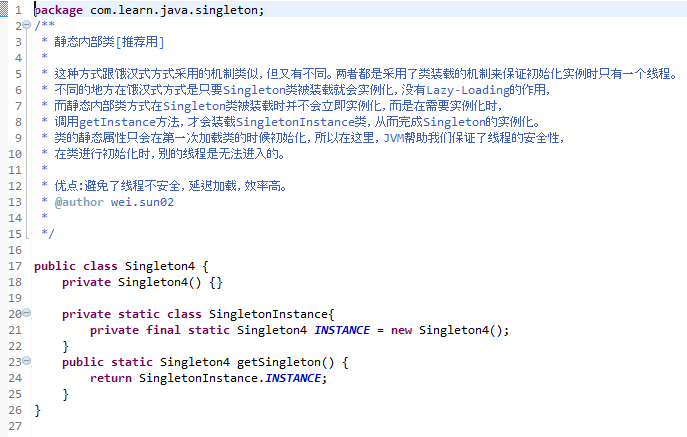


### 第二种：

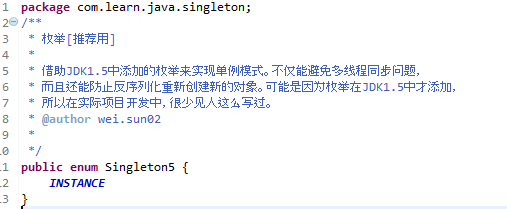
### 第三种：



### 第四种 :



### 第五种 :



## 注解

### 什么是注解？

Annontation是Java5开始引入的新特征，中文名称叫注解。它提供了一种安全的类似注释的机制，用来将任何的信息或元数据（metadata）与程序元素（类、方法、成员变量等）进行关联。为程序的元素（类、方法、成员变量）加上更直观更明了的说明，这些说明信息是与程序的业务逻辑无关，并且供指定的工具或框架使用。Annontation像一种修饰符一样，应用于包、类型、构造方法、方法、成员变量、参数及本地变量的声明语句中。  
　　Java注解是附加在代码中的一些元信息，用于一些工具在编译、运行时进行解析和使用，起到说明、配置的功能。注解不会也不能影响代码的实际逻辑，仅仅起到辅助性的作用。包含在 java.lang.annotation 包中。

### – 注解的特性有哪些？

### – 为什么要使用注解？

1.编译检查

2.生成文档

3.运行时动态处理，注入实例。

### – 如何使用注解？

#### 1.注解分类：

##### 1.1Java自带的标准注解

@SuppressWarnings(value = { "" })： 抑制警告

@Deprecated： 标记为过时方法

@Override：重写父类方法

##### 1.2元注解

|  |  |
| --- | --- |
| **注解** | **说明** |
| @Target  （重要） | 表示该注解可以用在什么地方，由ElementType枚举定义   ● ElementType.CONSTRUCTOR:用于描述构造器   ● ElementType.FIELD:成员变量、对象、属性（包括enum实例）   ● ElementType.LOCAL\_VARIABLE:用于描述局部变量   ● ElementType.METHOD:用于描述方法   ● ElementType.PACKAGE:用于描述包   ● ElementType.PARAMETER:用于描述参数   ● ElementType.TYPE:用于描述类、接口(包括注解类型) 或enum声明 PS：当注解未指定Target值时，此注解可以使用任何元素之上，就是上面的类型 |
| @Retention  （重要） | 表示需要在什么级别保存该注解信息，由RetentionPolicy枚举定义   ●   RetentionPolicy.SOURCE : 在编译阶段丢弃。这些注解在编译结束之后就不再有任何意义，所以它们不会写入字节码。@Override, @SuppressWarnings都属于这类注解。   ●   RetentionPolicy.CLASS : 在类加载的时候丢弃。在字节码文件的处理中有用。注解默认使用这种方式   ●   RetentionPolicy.RUNTIME : 始终不会丢弃，运行期也保留该注解，因此可以使用反射机制读取该注解的信息。我们自定义的注解通常使用这种方式。 |
| @Documented | 表示注解会被包含在javaapi文档中 |
| @Inherited | @Inherited 元注解是一个标记注解，@Inherited阐述了某个被标注的类型是被继承的。如果一个使用了@Inherited修饰的annotation类型被用于一个class，则这个annotation将被用于该class的子类。 |

##### 1.3自定义注解

自定义注解类编写的一些规则:  
  1. Annotation型定义为@interface, 所有的Annotation会自动继承java.lang.Annotation这一接口,并且不能再去继承别的类或是接口.  
  2. 参数成员只能用public或默认(default)这两个访问权修饰  
  3. 参数成员只能用基本类型byte,short,char,int,long,float,double,boolean八种基本数据类型和String、Enum、Class、annotations等数据类型,以及这一些类型的数组.  
  4. 要获取类方法和字段的注解信息，必须通过Java的反射技术来获取 Annotation对象,因为你除此之外没有别的获取注解对象的方法  
  5. 注解也可以没有定义成员, 不过这样注解就没啥用了  
PS:自定义注解需要使用到元注解

### – 注解的实现原理是什么？

注解本质是一个继承了Annotation的特殊接口，其具体实现类是Java运行时生成的动态代理类。而我们通过反射获取注解时，返回的是Java运行时生成的动态代理对象$Proxy1。通过代理对象调用自定义注解（接口）的方法，会最终调用AnnotationInvocationHandler的invoke方法。该方法会从memberValues这个Map中索引出对应的值。而memberValues的来源是Java常量池。  
– 注解的优缺点是什么？  
– 参考资料

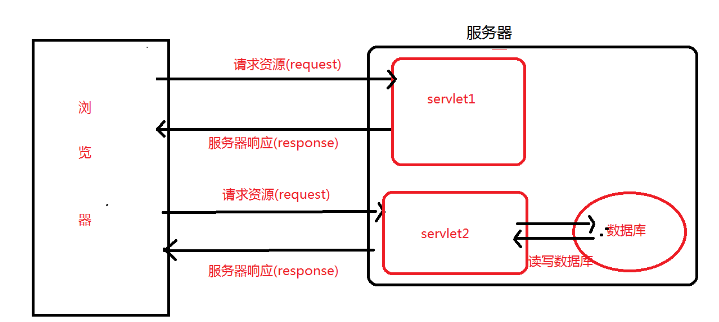
## Servlet:

### – 什么是Servlet？

处理请求和发送响应的过程是由一种叫做Servlet的程序来完成的，并且Servlet是为了解决实现动态页面而衍生的东西。理解这个的前提是了解一些http协议的东西，并且知道B/S模式(浏览器/服务器)。

B/S:浏览器/服务器。 浏览器通过网址来访问服务器，比如访问百度，在浏览器中输入www.baidu.com，这个时候浏览器就会显示百度的首页，那么这个具体的过程，步骤是怎样的呢？这个就了解一下[http请求和响应](http://www.cnblogs.com/whgk/p/6130882.html)。

请求，响应：通过给的链接应该可以知道这两个具体的内容

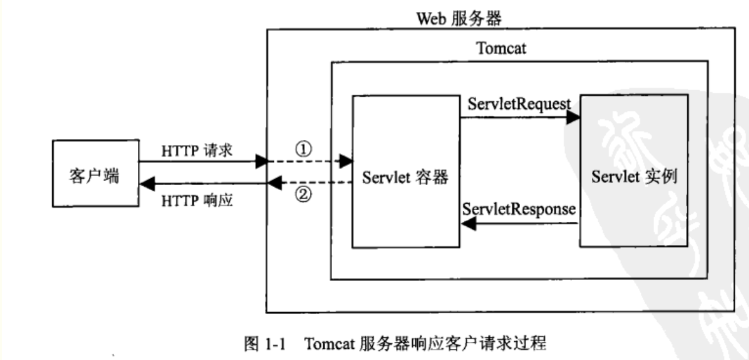


– Servlet的特性有哪些？  
– 为什么要使用Servlet？  
– 如何使用Servlet？

tomcat和servlet的关系

Tomcat 是Web应用服务器,是一个Servlet/JSP容器. Tomcat 作为Servlet容器,负责处理客户请求,把请求传送给Servlet,并将Servlet的响应传送回给客户.而Servlet是一种运行在支持Java语言的服务器上的组件. Servlet最常见的用途是扩展Java Web服务器功能,提供非常安全的,可移植的,易于使用的CGI替代品.

从http协议中的请求和响应可以得知，浏览器发出的请求是一个请求文本，而浏览器接收到的也应该是一个响应文本。但是在上面这个图中，并不知道是如何转变的，只知道浏览器发送过来的请求也就是request，我们响应回去的就用response。忽略了其中的细节，现在就来探究一下。



①：Tomcat将http请求文本接收并解析，然后封装成HttpServletRequest类型的request对象，所有的HTTP头数据读可以通过request对象调用对应的方法查询到。

②：Tomcat同时会要响应的信息封装为HttpServletResponse类型的response对象，通过设置response属性就可以控制要输出到浏览器的内容，然后将response交给tomcat，tomcat就会将其变成响应文本的格式发送给浏览器

Java Servlet API 是Servlet容器(tomcat)和servlet之间的接口，它定义了serlvet的各种方法，还定义了Servlet容器传送给Servlet的对象类，其中最重要的就是ServletRequest和ServletResponse。所以说我们在编写servlet时，需要实现Servlet接口，按照其规范进行操作。

### Servlet创建的两种方法:

第一种：客户端请求相应的Servlet时,创建Servlet实例;   
第二种：通过在web.xm中配置loadonstartup创建servlet实例,语法为:在 之间添加 1 ,这种方法在web容器启动时,就创建Servlet实例;

### **Servlet运行时一般都遵循如下生命周期:**

1.创建Servlet实例;   
2.Web 容器调用servlet的init()方法,对Servlet进行初始化;   
3.Servlet初始化后,就一直存在于容器中,用于响应客户端请求,根据客户端的请求方式,调用Servlet的service()方法,通过service方法去调用相应的doxxx方法;   
4.web容器销毁servlet时,调用Servlet的destroy方法,通常在关闭web容器时销毁servlet

## SpringMVC

### – 什么是springmvc？

在web模型中，MVC是一种很流行的框架，通过把Model，View，Controller分离，把较为复杂的web应用分成逻辑清晰的几部分，是为了简化开发，减少出错。还是为了组内开发人员之间的配合。总之就是一种分层工作的办法。

### –springmvc的特性有哪些？

Spring MVC特点

清晰地角色划分

 灵活的配置功能

 提供了大量的控制器接口和实现类

 真正的View层实现无关（JSP、Velocity、Xslt等）

 国际化支持

 面向接口编程

Spring提供了Web应用开发的一整套流程，不仅仅是MVC，他们之间可以很方便的结合一起

视图解析器

  将逻辑视图的名字与JSP等视图技术进行匹配

  InternalResourceViewResolver

  在Web应用程序的WAR文件中查找视图模板，视图模板的路径根据加完前缀和后缀的逻辑视图名称来确定

  prefix

  suffix

REST风格

  Representational State Transfer表述性状态转移

  传统的查、改、删的URL与REST风格的增删改URL对比

  /userview.action?id=12    VS    /user/12

  /userdelete.action?id=12   VS    /user/12/delete

  /userupdate.action?id=12  VS    /user/12/update

 请求方式

 GET

 POST

 DELETE

 PUT

  他强调的是一个资源可以对应多种视图

### – 为什么要使用springmvc？

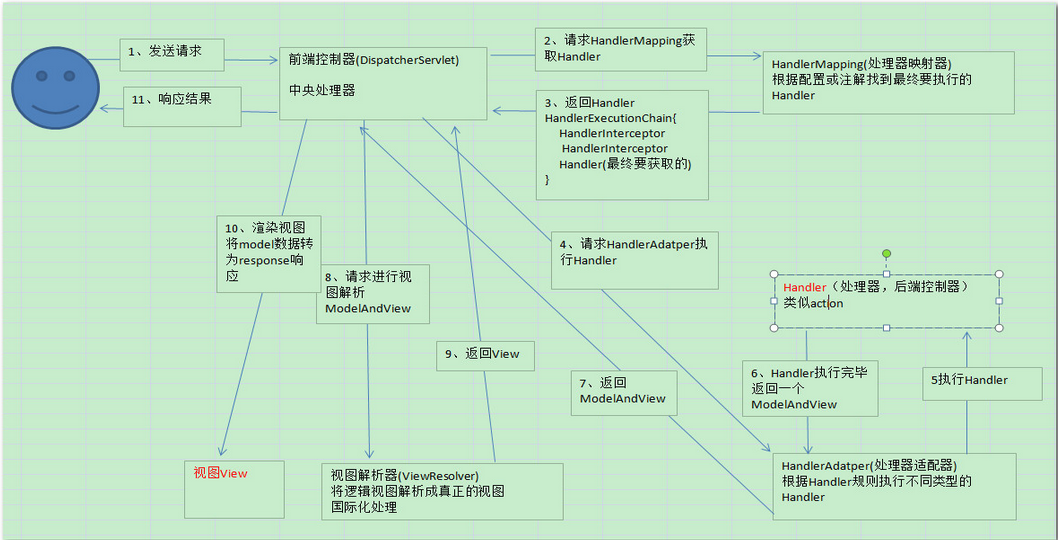
**SpringMVC对servlet进行封装**  
比如请求参数的映射。文件的上传，支持各种视图解析器  
servlet只有doGet和doPost，一个servlet类只能处理一个urlpattern  
SpringMVC类里面可以通过RequestMapping处理很多请求，并且支持Rest风格的请求，如DELETE/PUT等  
SpringMVC参数的映射可以直接封装成实体类

### **– 如何使用springmvc？**

见代码：

### –springmvc的实现原理是什么？

SpringMVC的工作原理图：



SpringMVC流程

 1. 用户向服务器发送请求，请求被Spring 前端控制Servelt DispatcherServlet捕获；

 2. DispatcherServlet对请求URL进行解析，得到请求资源标识符（URI）。然后根据该URI，调用HandlerMapping获得该Handler配置的所有相关的对象（包括Handler对象以及Handler对象对应的拦截器），最后以HandlerExecutionChain对象的形式返回；

 3. DispatcherServlet 根据获得的Handler，选择一个合适的HandlerAdapter。（**附注**：如果成功获得HandlerAdapter后，此时将开始执行拦截器的preHandler(...)方法）

 4.  提取Request中的模型数据，填充Handler入参，开始执行Handler（Controller)。 在填充Handler的入参过程中，根据你的配置，Spring将帮你做一些额外的工作：

      HttpMessageConveter： 将请求消息（如Json、xml等数据）转换成一个对象，将对象转换为指定的响应信息

      数据转换：对请求消息进行数据转换。如String转换成Integer、Double等

      数据根式化：对请求消息进行数据格式化。 如将字符串转换成格式化数字或格式化日期等

      数据验证： 验证数据的有效性（长度、格式等），验证结果存储到BindingResult或Error中5.  Handler执行完成后，向DispatcherServlet 返回一个ModelAndView对象；

6.  根据返回的ModelAndView，选择一个适合的ViewResolver（必须是已经注册到Spring容器中的 ViewResolver)返回给DispatcherServlet ；

7. ViewResolver 结合Model和View，来渲染视图

8. 将渲染结果返回给客户端。

组件说明：

以下组件通常使用框架提供实现：

DispatcherServlet：作为前端控制器，整个流程控制的中心，控制其它组件执行，统一调度，降低组件之间的耦合性，提高每个组件的扩展性。

HandlerMapping：通过扩展处理器映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

HandlAdapter：通过扩展处理器适配器，支持更多类型的处理器。

ViewResolver：通过扩展视图解析器，支持更多类型的视图解析，例如：jsp、freemarker、pdf、excel等。

对json的处理，通过@ResponseBody注解。在使用 @RequestMapping 后，返回值通常解析为跳转路径，加上 @Responsebody 后返回结果不会被解析为跳转路径，而是直接写入HTTP 响应正文中。例如，异步获取 json 数据，加上 @Responsebody 注解后，就会直接返回 json 数据

**@RequestMapping** 是一个用来处理请求地址映射的注解，可用于类或方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径；用于方法上，表示在类的父路径下追加方法上注解中的地址将会访问到该方法

**@Responsebody** 注解表示该方法的返回的结果直接写入 HTTP 响应正文（ResponseBody）中，一般在异步获取数据时使用；

**@RequestBody** 注解则是将 HTTP 请求正文插入方法中，使用适合的 HttpMessageConverter 将请求体写入某个对象。

–springmvc的优缺点是什么？  
– 参考资料

https://blog.csdn.net/zgzczzw/article/details/53926635

1. DispatcherServlethe的doDispatcher（）详解：

第一步：  DispatcherServlet收到请求调用HandlerMapping处理器映射器。返回一个HandlerExecutionChain

mappedHandler = getHandler(processedRequest);

第二步：DispatcherServlet根据处理器获取相应的处理器Adapter适配器

HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

第三步：

执行preHandle方法和Handle方法

获取到了Adapter，接下来就要做实际的事情了，执行preHandle方法很简单，就是调用一下，其实是为了用户定制。接下来真正处理事情的是adapter的handle方法

HandlerAdapter经过适配调用具体的处理器(Controller，也叫后端控制器)。返回一个moderandview，返回给DispatchServlet

mappedHandler.applyPreHandle(processedRequest, response)

mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

第四步：

DispatcherServlet将ModelAndView传给ViewReslover视图解析器。

processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);

第五步：

ViewReslover解析后返回具体View。

DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。

DispatcherServlet响应用户

render(mv, request, response);

前置拦截器

中置拦截器

后置拦截器

## HttpServletRequest

## HttpServletResponse

## 动态代理和静态代理的区别

## Springmvc + mybatis 工作流程详解

1.springmvc

1.1. 启动Tomcat服务器, > WEB容器会去读取它的配置文件web.xml，在其加载的过程中顺序依次为：contextparam >> listener >> fileter >> servlet –> 加载web.xml里其他配置文件 初始化DispatchServlet

|  |
| --- |
| <welcomefilelist>  <welcomefile>index.jsp</welcomefile>  </welcomefilelist>    <servlet>  <servletname>springmvc</servletname>  <servletclass>  org.springframework.web.servlet.DispatcherServlet  </servletclass>  <! DispatcherServlet的初始化方法 在执行时，会启动spring容器。 >  <initparam>  <paramname>contextConfigLocation</paramname>  <paramvalue>classpath:springmvc.xml</paramvalue>  </initparam>  <loadonstartup>1</loadonstartup>  </servlet>  <servletmapping>  <servletname>springmvc</servletname>  <urlpattern>\*.do</urlpattern>  </servletmapping> |

1.2 初始化 springmvc.xml, 扫描以下包带有注解的类。

|  |
| --- |
| <! 配置组件扫描 >  <context:componentscan basepackage=*"com.learn.controller"*/>  <! 配置spring mvc注解扫描 >  <mvc:annotationdriven/>  <! 配置视图解析器 前缀和后缀>  <bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <property name=*"prefix"* value=*"/WEBINF/"*/>  <property name=*"suffix"* value=*".jsp"*/>  </bean> |

1.3网页发送请求<http://localhost:8080/springmvcdemo/hello.do>

1.4 DispatchServlet根据 @RequestMapping("/hello.do") 找到注解方法，并返回”hello” > 根据视图解析器得到hello.jsp >显示到浏览器

|  |
| --- |
| @Controller("hc")  **public** **class** HelloController {  //利用注解  @RequestMapping("/hello.do")  **public** String toHello(){  System.***out***.println("HelloController的toHello方法...");  **return** "hello";  }  } |

2. mybatis +spring配置详解

2.1

配置springmybatis.xml

|  |
| --- |
| <! 配置整合MyBatis >  <! 1:配置数据库相关参数 properties属性：${url} >  <context:propertyplaceholder location=*"classpath:jdbc.properties"* />  <! 2:数据库连接池 >  <bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>  <! 配置连接池属性 >  <property name=*"driverClass"* value=*"${jdbc.driver}"* />  <property name=*"jdbcUrl"* value=*"${jdbc.url}"* />  <property name=*"user"* value=*"${jdbc.username}"* />  <property name=*"password"* value=*"${jdbc.password}"* />  <! c3p0连接池的私有属性 >  <property name=*"maxPoolSize"* value=*"${maxPoolSize}"* />  <property name=*"minPoolSize"* value=*"${minPoolSize}"* />  <! 关闭连接后不自动commit >  <property name=*"autoCommitOnClose"* value=*"${autoCommitOnClose}"* />  <! 获取连接超时时间 >  <property name=*"checkoutTimeout"* value=*"${checkoutTimeout}"* />  <! 当获取连接失败重试次数 >  <property name=*"acquireRetryAttempts"* value=*"${acquireRetryAttempts}"* />  </bean>  <! spring + MyBatis 精华 >  <! 约定大于配置 >  <! 3:配置SqlSessionFactory对象 >  <bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>  <! 注入数据库连接池 >  <property name=*"dataSource"* ref=*"dataSource"* />  <! 配置MyBatis全局配置文件：mybatisconfig.xml >  <property name=*"configLocation"* value=*"classpath:mybatisconfig.xml"* />  <! 扫描entity包 使用别名 org.asset.entity.User > User >  <property name=*"typeAliasesPackage"* value=*"com.learn.entity"* />  <! 扫描sql配置文件:mapper需要的xml文件 >  <property name=*"mapperLocations"* value=*"classpath:mapper/\*.xml"* />  </bean>  <! 4:配置扫描DAO接口包，动态实现DAO接口，并注入到spring容器中 >  <bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>  <! 给出需要扫描DAO接口包 >  <property name=*"basePackage"* value=*"com.learn.dao"* />  <! 注入sqlSessionFactory >  <property name=*"sqlSessionFactoryBeanName"* value=*"sqlSessionFactory"* />  </bean> |

2.2 编写mapper.xml., 其中namespace=*"com.learn.dao.UserDao"* 对应Dao接口对应的位置，<insert>标签的id为Dao接口的抽象方法。抽象方法不能重载。

|  |
| --- |
| <mapper namespace=*"com.learn.dao.UserDao"*>  <! 通过resultMap解决实体类属性与数据库字段名不一致的问题 >  <resultMap type=*"com.learn.entity.User"* id=*"userResult"*>  <! 其中id为主键，result为普通字段 property对应实体类属性，column对应字段名 （更准确的说应该是查询结果集中的字段名 >  <id property=*"id"* column=*"user\_id"* />  <result property=*"name"* column=*"user\_name"* />  <result property=*"birthday"* column=*"user\_birthday"* />  <result property=*"salary"* column=*"user\_salary"* />  </resultMap>  <! 插入数据后获得自增主键的值 >  <insert id=*"insert"* keyProperty=*"id"* keyColumn=*"user\_id"*>  INSERT INTO tb\_user  (user\_name,user\_birthday,user\_salary)  VALUES  (#{name},#{birthday},#{salary})  </insert>  </mapper> |

Dao接口

|  |
| --- |
| **public** **interface** UserDao {  **public** **void** insert(User user);  **public** **void** delete(**int** id);  **public** **void** update(User user);  **public** User findById(**int** id);  **public** List<User> findAll();  } |

对比没有整合Spring的mybatis

|  |
| --- |
| **private** SqlSession getSession() {  //1.读取配置文件sqMapConfig.xml来构建sqlSessionFactory实例。  //通过类加载器读取.xml文件  InputStream in = getClass().getClassLoader().getResourceAsStream("mybatisconfig.xml");  //2.每一个程序都是以一个sqlSessionFactory对象为核心，创建sqlSessionFactory.  SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory sqlSessionFactory = builder.build(in);  //3.创建sqlSession对象，该对象应该是线程私有的。每个方法都要创建一个，然后再finally块关闭。  **return** sqlSessionFactory.openSession();    }    @Test  **public** **void** testInsert() {  SqlSession session = getSession();  **try** {  BlogDao blogDao = session.getMapper(BlogDao.**class**);  **for**(**int** i=1; i<=5; i++) {  Blog blog = **new** Blog(i,"BBB","464646546@163.com");  blogDao.insert(blog);  session.commit();  }  System.***out***.println("插入完成");  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  session.close();  }  } |

整合后的mybatis测试代码：

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**)  @ContextConfiguration({ "classpath:springmybatis.xml" })  **public** **class** UserDaoTest {  @Autowired  **private** UserDao userDAO;    @Test  **public** **void** testInsert() {  User user = **new** User();  user.setBirthday(**new** Date());  user.setName("James");  user.setSalary(15000.0);  **this**.userDAO.insert(user);  logger.info("insert the user={}", user);  **this**.userDAO.delete(user.getId());  User u = **this**.userDAO.findById(user.getId());  logger.info("after delete the user={}", u);  } |

## Maven.

1.maven用eclipse创建后不显示部分文件夹，只要把jre版本修改即可。

2. maven 通过命令行创建项：

在项目文件夹下 shift + 右击 -> 选择 open command window here（在此处打开命令） -> 输入

**mvn archetype:generate** —> 根据提示输入maven项目，groupId，artifactId -> 创建完成后导入

打包：mvn package  
编译：mvn compile  
编译测试程序：mvn testcompile  
清空：mvn clean  
运行测试：mvn test  
生成站点目录: mvn site  
生成站点目录并发布：mvn sitedeploy  
安装当前工程的输出文件到本地仓库: mvn install

**POM标签：**

<!-- 用在子模块中，实现对父模块的继承 -->

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.2.5.RELEASE</version>  <!--其中relativePath元素不是必须的，指定后会优先从指定的位置查找父pom。-->  <relativePath>../my-parent</relativePath  </parent> |

**MAVEN继承**

继承是 Maven 中很强大的一种功能，继承可以使得子POM可以获得 parent 中的各项配置，可以对子pom进行统一的配置和依赖管理。大多数元素都可继承，只有部分不可以继承，如下：

* artifactId
* name
* prerequisites

**MAVEN聚合（或多模块）**

具有模块的项目被称为多模块或聚合项目。模块是此POM列出并作为一组执行的项目。通过一个pom打包的项目可以将它们列为模块来聚合成一组项目进行构建，这些模块名是这些项目的相对目录。

在列出模块时，不需要自己考虑模块间依赖关系，即POM给出的模块排序并不重要。Maven将对模块进行拓扑排序，使得依赖关系始终在依赖模块之前构建。

|  |
| --- |
| <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  https://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>org.codehaus.mojo</groupId>  <artifactId>my-parent</artifactId>  <version>2.0</version>  <packaging>pom</packaging>  <modules>  <module>my-project</module>  <module>another-project</module>  </modules>  </project> |

虽然聚合通常伴随着父POM的继承关系，但是这两者不是必须同时存在的，从上面两者的介绍可以看出来，这两者的都有不同的作用，他们的作用不依赖于另一个的配置。

父POM是为了抽取统一的配置信息和依赖版本控制，方便子POM直接引用，简化子POM的配置。聚合（多模块）则是为了方便一组项目进行统一的操作而作为一个大的整体，所以要真正根据这两者不同的作用来使用，不必为了聚合而继承同一个父POM，也不比为了继承父POM而设计成多模块。

## JSP.

JSP（Java Server Pages，即：Java服务器页面），是一种跨平台的动态网页技术标准，由Sun Microsystems公司倡导、多家公司参与建立。它在HTML文件中插入Java程序段(Scriptlet)和JSP标记(tag)，从而形成JSP文件(\*.jsp)。 用JSP开发的Web应用是跨平台的，既能在Linux下运行，也能在其他操作系统上运行。

### 语法：

#### 脚本程序的语法格式：

**<% 代码片段 %>**

<%   
out.println("Your IP address is " + request.getRemoteAddr());   
%>

#### JSP声明的语法格式：

**<%! declaration; [ declaration; ]+ ... %>**

<%! int i = 0; %>   
<%! int a, b, c; %>   
<%! Circle a = new Circle(2.0); %>

#### JSP表达式的语法格式：

**<%= 表达式 %>**

<p>   
 Today's date: <%= (new java.util.Date()).toLocaleString()%>  
</p>

#### JSP注释的语法格式：

**<% 这里可以填写 JSP 注释 %>**

<% 该部分注释在网页中不会被显示%>

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **语法** | **描述** | | <% 注释 %> | JSP注释，注释内容不会被发送至浏览器甚至不会被编译 | | <! 注释 > | HTML注释，通过浏览器查看网页源代码时可以看见注释内容 | | <\% | 代表静态 <%常量 | | %\> | 代表静态 %> 常量 | | \' | 在属性中使用的单引号 | | \" | 在属性中使用的双引号 | |

### JSP指令

JSP指令用来设置与整个JSP页面相关的属性。

JSP指令语法格式：

<%@ directive attribute="value" %>

这里有三种指令标签：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **指令** | **描述** | | <%@ page ... %> | 定义页面的依赖属性，比如脚本语言、error页面、缓存需求等等 | | <%@ include ... %> | 包含其他文件 | | <%@ taglib ... %> | 引入标签库的定义，可以是自定义标签 | |

### JSP行为

JSP行为标签使用XML语法结构来控制servlet引擎。它能够动态插入一个文件，重用JavaBean组件，引导用户去另一个页面，为Java插件产生相关的HTML等等。

行为标签只有一种语法格式，它严格遵守XML标准：

<jsp:action\_name attribute="value" />

行为标签基本上是一些预先就定义好的函数

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **语法** | **描述** | | jsp:include | 用于在当前页面中包含静态或动态资源 | | jsp:useBean | 寻找和初始化一个JavaBean组件 | | jsp:setProperty | 设置 JavaBean组件的值 | | jsp:getProperty | 将 JavaBean组件的值插入到 output中 | | jsp:forward | 从一个JSP文件向另一个文件传递一个包含用户请求的request对象 | | jsp:plugin | 用于在生成的HTML页面中包含Applet和JavaBean对象 | | jsp:element | 动态创建一个XML元素 | | jsp:attribute | 定义动态创建的XML元素的属性 | | jsp:body | 定义动态创建的XML元素的主体 | | jsp:text | 用于封装模板数据 | |

### JSP隐含对象

JSP支持九个自动定义的变量

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **对象** | **描述** | | request | **HttpServletRequest**类的实例 | | response | **HttpServletResponse**类的实例 | | out | **PrintWriter**类的实例，用于把结果输出至网页上 | | session | **HttpSession**类的实例 | | application | **ServletContext**类的实例，与应用上下文有关 | | config | **ServletConfig**类的实例 | | pageContext | **PageContext**类的实例，提供对JSP页面所有对象以及命名空间的访问 | | page | 类似于Java类中的this关键字 | | Exception | **Exception**类的对象，代表发生错误的JSP页面中对应的异常对象 | |

### 控制流语句

JSP提供对Java语言的全面支持。您可以在JSP程序中使用Java API甚至建立Java代码块，包括判断语句和循环语句等等。

### 判断语句

If…else块，请看下面这个例子：

|  |
| --- |
| <%! int day = 3; %> //声明的格式 <html>  <head> <title>IF...ELSE Example</title> </head>  <body> <% if (day == 1 | day == 7) { %> //脚本格式   <p> Today is weekend</p> <% } else { %>   <p> Today is not weekend</p> <% } %> </body>  </html> |

### JSP常量

JSP语言定义了以下几个常量：

1.布尔值(boolean)：true 和 false；

2.整型(int)：与Java中的一样;

3.浮点型(float)：与Java中的一样;

4.字符串(string)：以单引号或双引号开始和结束；

5.Null：null。

### EL表达式

语法格式

${表达式}

${user.sex=='male'}"

|  |
| --- |
| <! 1.EL可以直接使用。  2.request已经绑定了要访问的数据(user)。  3.EL访问的是该对象的Bean属性。>  <! 1.获取Bean属性 name是Bean的属性，不是对象的属性。>  <p>${user.name }</p>  <p>${user["password"] }</p>  <p>${user.course.name }</p> |

**EL 变量**

EL 存取变量数据的方法很简单，例如：${username}。它的意思是取出某一范围中名称为username的变量。因为我们并没有指定哪一个范围的username，所以它的默认值会先从Page 范围找，假如找不到，再依序到Request、Session、Application范围。假如途中找到username，就直接回传，不再继续找下去，但是假如全部的范围都没有找到时，就回传null，当然EL表达式还会做出优化，页面上显示空白，而不是打印输出NULL。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 属性范围（jstl名称） | EL中的名称 | | Page | PageScope | | Request | RequestScope | | Session | SessionScope | | Application | ApplicationScope | |

我们也可以指定要取出哪一个范围的变量：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 范例 | 说明 | | ${pageScope.username} | 取出Page范围的username变量 | | ${requestScope.username} | 取出Request范围的username变量 | | ${sessionScope.username} | 取出Session范围的username变量 | | ${applicationScope.username} | 取出Application范围的username变量 | |

其中，pageScope、requestScope、sessionScope和applicationScope都是EL 的隐含对象，

由它们的名称可以很容易猜出它们所代表的意思，

例如：${sessionScope.username}是取出Session范围的username 变量。这种写法是不是比之前JSP 的写法：

String username =(String) session.getAttribute("username");容易、简洁许多.

**EL 隐含对象**

JSP有9个隐含对象，而EL也有自己的隐含对象。EL隐含对象总共有11 个

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | 隐含对象 | 类型 | 说明 | | PageContext | javax.servlet.ServletContext | 表示此JSP的PageContext | | PageScope | java.util.Map | 取得Page范围的属性名称所对应的值 | | RequestScope | java.util.Map | 取得Request范围的属性名称所对应的值 | | sessionScope | java.util.Map | 取得Session范围的属性名称所对应的值 | | applicationScope | java.util.Map | 取得Application范围的属性名称所对应的值 | | param | java.util.Map | 如同ServletRequest.getParameter(String name)。回传String类型的值 | | paramValues | java.util.Map | 如同ServletRequest.getParameterValues(String name)。回传String[]类型的值 | | header | java.util.Map | 如同ServletRequest.getHeader(String name)。回传String类型的值 | | headerValues | java.util.Map | 如同ServletRequest.getHeaders(String name)。回传String[]类型的值 | | cookie | java.util.Map | 如同HttpServletRequest.getCookies() | | initParam | java.util.Map | 如同ServletContext.getInitParameter(String name)。回传String类型的值 | |

## LOG4J：

如何使用：

1，POM文件：

|  |
| --- |
| <!-- 1.日志 -->  <!-- https://mvnrepository.com/artifact/commons-logging/commons-logging -->  <dependency>  <groupId>commons-logging</groupId>  <artifactId>commons-logging</artifactId>  <version>1.2</version>  </dependency>  <!-- PS: 以下两个jar用来解决，Log4JLogger无法读取log配置文件问题 -->  <!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-api -->  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>1.7.25</version>  </dependency>  <!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-log4j12 -->  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  <version>1.7.25</version>  <scope>test</scope>  </dependency>  <!-- <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-classic</artifactId>  <version>1.1.8</version>  </dependency> --> |

2. log4j.properties

|  |
| --- |
| #log4j.rootLogger = [ level ] , appenderName1, appenderName2,  #ALL < DEBUG < INFO < WARN  log4j.rootLogger=debug,logfile,stdout  #log4j.rootLogger =ALL,systemOut,logFile  #log4j.logger.org.springframework=DEBUG    #write log into file  log4j.appender.logfile=org.apache.log4j.DailyRollingFileAppender  log4j.appender.logfile.Threshold=warn  log4j.appender.logfile.File=${webapp.root}\\logs\\Transaction.log  log4j.appender.logfile.DatePattern=.yyyy-MM-dd  log4j.appender.logfile.layout=org.apache.log4j.PatternLayout  log4j.appender.logfile.layout.ConversionPattern=[Transaction] %d{yyyy-MM-dd HH\:mm\:ss} %X{remoteAddr} %X{remotePort} %m %n  #display in console  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.Threshold=DEBUG  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=[Transaction DEBUG] %d{yyyy-MM-dd HH\:mm\:ss} %X{remoteAddr} %X{remotePort} %m %n |

3．JAVA

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**)  @ContextConfiguration("classpath:applicationContextPropagation.xml")  **public** **class** SpringTestPropagation {  /\*\*  \* 注入代理类  \*/  @Autowired  @Qualifier(value="accountService")  **private** AccountServicePropagation accountService;    //private Logger logger = LoggerFactory.getLogger(getClass());  **protected** **transient** Log logger = LogFactory.*getLog*(getClass());  //boolean debugEnabled = logger.isDebugEnabled();    String in = "bbb";  String out = "aaa";  @Test  **public** **void** testMethodB() {  logger.info("==============================");  logger.debug("debug");  accountService.methodB(in, out, 100d);  logger.info("==============================");  }    @Test  **public** **void** testMethodA() {  logger.info("==============================");  accountService.methodA(in, out, 100d);  logger.info("==============================");  }  } |

## [JMS-AciveMQ](http://www.cnblogs.com/jelynn/p/6702317.html)

一：JMS基本概念

JMS即Java消息服务（Java Message Service）应用程序接口是一个Java平台中关于面向消息中间件（MOM）的API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信(用于解决两个或者多个程序之间的耦合)。它便于消息系统中的 Java 应用程序进行消息交换，并且通过提供标准的产生、发送、接收消息的接口简化企业应用的开发。

1.  JMS的目标

        为企业级的应用提供一种智能的消息系统，JMS定义了一整套的企业级的消息概念与工具，尽可能最小化的[**Java**](http://lib.csdn.net/base/javase)语言概念去构建最大化企业消息应用。统一已经存在的企业级消息系统功能。

2.  JMS应用程序,

一个完整的JMS应用应该实现以下功能：

JMS 客户端 – Java语言开发的接受与发送消息的程序

非JMS客户端 – 基于消息系统的本地API实现而不是JMS

消息 – 应用程序用来相互交流信息的载体

被管理对象–预先配置的JMS对象，JMS管理员创建，被客户端运用。如链接工厂，主题等

JMS提供者–完成JMS功能与管理功能的消息系统

3.  JMS体系结构

描述如下:

JMS提供者（JMS的实现者，比如activemq jbossmq等）

JMS客户（使用提供者发送消息的程序或对象，例如在12306中，负责发送一条购票消息到处理队列中，用来解决购票高峰问题，那么，发送消息到队列的程序和从队列获取消息的程序都叫做客户）

JMS生产者，JMS消费者（生产者及负责创建并发送消息的客户，消费者是负责接收并处理消息的客户）

JMS消息（在JMS客户之间传递数据的对象）

JMS队列（一个容纳那些被发送的等待阅读的消息的区域）

JMS主题（一种支持发送消息给多个订阅者的机制）

4.  JMS对象模型

连接工厂（connectionfactory）客户端使用连接工厂创建一个JMS连接（connection）。

JMS连接 表示JMS客户端和服务器端之间的一个活动的连接，是由客户端通过调用连接工厂的方法建立的。

JMS会话 session 标识JMS客户端和服务端的会话状态。会话建立在JMS连接上，标识客户与服务器之间的一个会话进程。

**二：JMS的消息模式**

**1.  点对点的消息模式(Point to Point Messaging)**

点对点消息模型：通过一个服务器消息队列实现，消息的发送者向队列写入消息，消息的接收者从队列取出消息。

下面的JMS对象在点对点消息模式中是必须的：

a.      队列(Queue) – 一个提供者命名的队列对象，客户端将会使用这个命名的队列对象

b.     队列链接工厂(QueueConnectionFactory) – 客户端使用队列链接工厂创建链接队列

        ConnectionQueue来取得与JMS点对点消息提供者的链接。

c.      链接队列(ConnectionQueue) – 一个活动的链接队列存在在客户端与点对点消息提供者之间，客户用它创建一个或者多个JMS队列会话(QueueSession)

d.     队列会话(QueueSession) – 用来创建队列消息的发送者与接受者(QueueSenderand

         QueueReceiver)

e.     消息发送者(QueueSender 或者MessageProducer)– 发送消息到已经声明的队列

f.       消息接受者(QueueReceiver或者MessageConsumer) – 接受已经被发送到指定队列的消息

**2.  发布订阅模式(publish – subscribe Mode)**

发布-订阅模式：把消息发送到给一个主题(Topic)，消息服务器将消息发布给订阅器该主题的每一个订阅者。举个通俗的例子，就好比如一家杂志社（相当于消息发送者）把一堆杂志（相当于消息）寄到了邮政（相当于主题），再由邮政将杂志发给每一个有订阅这本杂志的读者（相当于消息接收者）

必须的消息对象:

a.      主题Topic(Destination) – 一个提供者命名的主题对象，客户端将会使用这个命名的主题对象

b.     主题链接工厂(TopciConnectionFactory) – 客户端使用主题链接工厂创建链接主题

         ConnectionTopic来取得与JMS消息Pub/Sub提供者的链接。

c.      链接主题(ConnectionTopic) – 一个活动的链接主题存在发布者与订阅者之间

d.     会话(TopicSession) – 用来创建主题消息的发布者与订阅者 (TopicPublisher  and

         TopicSubscribers)

e.     消息发送者MessageProducer) – 发送消息到已经声明的主题

f.       消息接受者(MessageConsumer) – 接受已经被发送到指定主题的消息

3.  区别：

点对点模型每一个消息只有一个接收者。

发布-订阅消息模式的每一个消息可以有多个接收者。

https://www.cnblogs.com/softidea/p/5587126.html

https://www.cnblogs.com/molao-doing/articles/6557305.html

**P2P的特点**

3.1、每个消息只有一个消费者（Consumer）(即一旦被消费，消息就不再在消息队列中)  
3.2、提供者和消费者之间在时间上没有依赖性，也就是说当提供者发送了消息之后，不管消费者有没有正在运行，它不会影响到消息被发送到队列  
3.3、每条消息仅会传送给一个消费者。可能会有多个消费者在一个队列中侦听，但是每个队列中的消息只能被队列中的一个消费者所消费。  
3.4、消息存在先后顺序。一个队列会按照消息服务器将消息放入队列中的顺序，把它们传送给消费者。当已被消费时，就会从队列头部将它们删除（除非使用了消息优先级）。  
3.5、消费者在成功接收消息之后需向队列应答成功  
PS:如果你希望发送的每个消息都应该被成功处理的话，那么你需要P2P模式。

**Pub/Sub（发布/订阅模式）的特点**

3.1、每个消息可以有多个消费者  
3.2、发布者和订阅者之间有时间上的依赖性。针对某个主题的订阅者，它必须创建一个订阅者之后，才能消费发布者的消息，而且为了消费消息，订阅者必须保持运行的状态。  
3.3、为了缓和这样严格的时间相关性，JMS允许订阅者创建一个可持久化的订阅。这样，即使订阅者没有被激活（运行），它也能接收到发布者的消息。  
3.4、每条消息都会传送给称为订阅者的多个消息消费者。订阅者有许多类型，包括持久型、非持久型和动态型。  
3.5、发布者通常不会知道、也意识不到哪一个订阅者正在接收主题消息。  
3.6、消息被推送给消费者，这意味着消息会传送给消费者，而无须请求。

PS:如果你希望发送的消息可以不被做任何处理、或者被一个消息者处理、或者可以被多个消费者处理的话，那么可以采用Pub/Sub模型

## Dubbo

参考：https://blog.csdn.net/noaman\_wgs/article/details/70214612/

### 什么是Dubbo？

**Dubbo**是阿里巴巴公司开源的一个高性能优秀的服务框架，使得应用可通过高性能的RPC实现服务的输出和输入功能，以及SOA服务治理方案，和spring框架无缝集成。

**服务框架**:就是提供服务的，服务框架是基于业务对应SaaS分发模式的服务进行整合，以产生新的应用。服务框架中，与业务相关，但与业务功能的整合无关的组件以外部服务形式引入（也就是说把一些业务分离出来，变成一种服务，供其他人调用该服务）。

**RPC**:(Remote Procedure CallProtocol）远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。RPC使得开发包括网络分布式多程序在内的应用程序更加容易。（理解：远程调用协议，为Dubbo实现远程接口调用做支持）

**SOA**([Service Oriented Architecture](https://baike.baidu.com/item/SOA/9614173#viewPageContent)): 是一个组件模型，它将应用程序的不同功能单元（称为服务）通过这些服务之间定义良好的接口和契约联系起来。接口是采用中立的方式进行定义的，它应该独立于实现服务的硬件平台、操作系统和编程语言。这使得构建在各种各样的系统中的服务可以以一种统一和通用的方式进行交互。

### Dubbo的特性有哪些？

1.透明化的远程方法调用，就像调用本地方法一样调用远程方法，只需简单配置，没有任何API侵入。   
2.软负载均衡及容错机制，可在内网替代F5等硬件负载均衡器，降低成本，减少单点。   
3. 服务自动注册与发现，不再需要写死服务提供方地址，注册中心基于接口名查询服务提供者的IP地址，并且能够平滑添加或删除服务提供者。

### 为什么要使用Dubbo？

单一应用架构   
当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。 此时，用于简化增删改查工作量的数据访问框架(ORM) 是关键。

垂直应用架构   
当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。此时，用于加速前端页面开发的 Web框架(MVC) 是关键。

分布式服务架构   
当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。此时，用于提高业务复用及整合的 分布式服务框架(RPC) 是关键。

流动计算架构   
当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。   
此时，用于提高机器利用率的 资源调度和治理中心(SOA) 是关键。

### 如何使用Dubbo？

#### 如何启动配置dubbo项目：

1.启动zookeeper，bin目录下zkServer.cmd

2.启动dubbo项目的提供者，启动项目的消费者

3. 将dubbo-admin项目进行打包，进入dubbo-admin目录，进入命令行页面，

输入**mvn package -Dmaven.skip.test=true** ，打包完成后，进入target目标，将war包copy到tomcat的webapps目录下

4.启动tomcat，bin目录下startup.bat, tomcat会将dubbo-admin的war包解压。

5.在浏览器输入localhost：8080/dubbo-admin的war包名称

6.在dubbo管理首页可以看到服务名，显示所有提供者和消费者。

### Dubbo的实现原理是什么？

### 

我们解释以下这个架构图：

Consumer服务消费者，Provider服务提供者。Container服务容器。消费当然是invoke提供者了，invoke这条实线按照图上的说明当然同步的意思了，多说一句，在实际调用过程中，Provider的位置对于Consumer来说是透明的，上一次调用服务的位置（IP地址）和下一次调用服务的位置，是不确定的。这个地方就是实现了软负载。

服务提供者先启动start，然后注册register服务。

消费订阅subscribe服务，如果没有订阅到自己想获得的服务，它会不断的尝试订阅。新的服务注册到注册中心以后，注册中心会将这些服务通过notify到消费者。

Monitor这是一个监控，图中虚线表明Consumer 和Provider通过异步的方式发送消息至Monitor，Consumer和Provider会将信息存放在本地磁盘，平均1min会发送一次信息。Monitor在整个架构中是可选的（图中的虚线并不是可选的意思），Monitor功能需要单独配置，不配置或者配置以后，Monitor挂掉并不会影响服务的调用。

**I、初始化过程细节：**   
上图中的第一步start，就是将服务装载容器中，然后准备注册服务。和Spring中启动过程类似，spring启动时，将bean装载进容器中的时候，首先要解析bean。所以dubbo也是先读配置文件解析服务。   
解析服务：   
1）、基于dubbo.jar内的Meta-inf/spring.handlers配置，spring在遇到dubbo名称空间时，会回调DubboNamespaceHandler类。   
2）、所有的dubbo标签，都统一用DubboBeanDefinitionParser进行解析，基于一对一属性映射，将XML标签解析为Bean对象。   
源码截图：   
在ServiceConfig.export 或者ReferenceConfig.get 初始化时，将Bean对象转会为url格式，将所以Bean属性转成url的参数。   
然后将URL传给Protocol扩展点，基于扩展点的Adaptive机制，根据URL的协议头，进行不同协议的服务暴露和引用。   
暴露服务：

a、 只暴露服务端口

在没有使用注册中心的情况，这种情况一般适用在开发环境下，服务的调用这和提供在同一个IP上，只需要打开服务的端口即可。   
即，当配置 or   
ServiceConfig解析出的URL的格式为：   
Dubbo：//service-host/com.xxx.TxxService?version=1.0.0   
基于扩展点的Adaptiver机制，通过URL的“dubbo：//”协议头识别，直接调用DubboProtocol的export（）方法，打开服务端口。

b、向注册中心暴露服务：

和上一种的区别：需要将服务的IP和端口一同暴露给注册中心。   
ServiceConfig解析出的url格式为：   
registry://registry-host/com.alibaba.dubbo.registry.RegistryService?export=URL.encode(“dubbo://service-host/com.xxx.TxxService?version=1.0.0”)

基于扩展点的Adaptive机制，通过URL的“registry：//”协议头识别，调用RegistryProtocol的export方法，将export参数中的提供者URL先注册到注册中心，再重新传给Protocol扩展点进行暴露：   
Dubbo：//service-host/com.xxx.TxxService?version=1.0.0

引用服务：

a、直接引用服务：

在没有注册中心的，直连提供者情况下，   
ReferenceConfig解析出的URL格式为：   
Dubbo：//service-host/com.xxx.TxxService?version=1.0.0

基于扩展点的Adaptive机制，通过url的“dubbo：//”协议头识别，直接调用DubboProtocol的refer方法，返回提供者引用。

b、从注册中心发现引用服务：

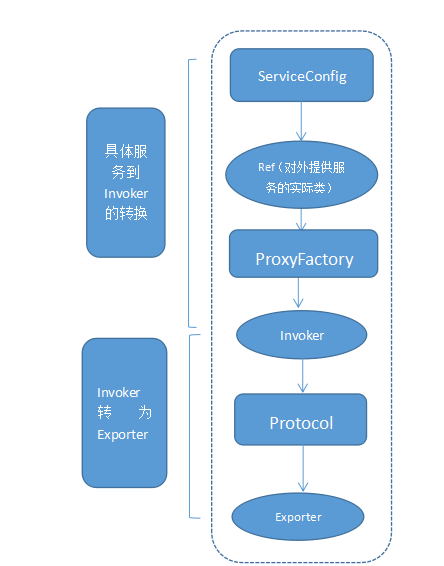
此时，ReferenceConfig解析出的URL的格式为：   
•registry://registry-host/com.alibaba.dubbo.registry.RegistryService?refer=URL.encode(“consumer://consumer-host/com.foo.FooService?version=1.0.0”)

基于扩展点的Apaptive机制，通过URL的“registry：//”协议头识别，就会调用RegistryProtocol的refer方法，基于refer参数总的条件，查询提供者URL，如：   
Dubbo：//service-host/com.xxx.TxxService?version=1.0.0

基于扩展点的Adaptive机制，通过提供者URL的“dubbo：//”协议头识别，就会调用DubboProtocol的refer（）方法，得到提供者引用。   
然后RegistryProtocol将多个提供者引用，通过Cluster扩展点，伪装成单个提供这引用返回。

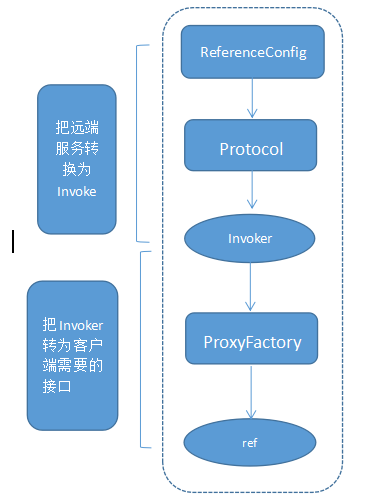
**三、远程调用细节：**

服务提供者暴露一个服务的详细过程：



上图是服务提供者暴露服务的主过程：   
首先ServiceConfig类拿到对外提供服务的实际类ref，然后将ProxyFactory类的getInvoker方法使用ref生成一个AbstractProxyInvoker实例，到这一步就完成具体服务到invoker的转化。接下来就是Invoker转换到Exporter的过程。   
Dubbo处理服务暴露的关键就在Invoker转换到Exporter的过程，下面我们以Dubbo和rmi这两种典型协议的实现来进行说明：   
Dubbo的实现：   
Dubbo协议的Invoker转为Exporter发生在DubboProtocol类的export方法，它主要是打开socket侦听服务，并接收客户端发来的各种请求，通讯细节由dubbo自己实现。   
Rmi的实现：   
RMI协议的Invoker转为Exporter发生在RmiProtocol类的export方法，他通过Spring或Dubbo或JDK来实现服务，通讯细节由JDK底层来实现。

服务消费者消费一个服务的详细过程



上图是服务消费的主过程：   
首先ReferenceConfig类的init方法调用Protocol的refer方法生成Invoker实例。接下来把Invoker转为客户端需要的接口

### Dubbo的优缺点是什么？

**优点：**

1.透明化的远程方法调用   
 - 像调用本地方法一样调用远程方法；只需简单配置，没有任何API侵入。

2、软负载均衡及容错机制,可在内网替代nginx lvs等硬件负载均衡器。

3、服务注册中心自动注册 & 配置管理

- 不需要写死服务提供者地址，注册中心基于接口名自动查询提供者ip。

- 使用类似zookeeper等分布 式协调服务作为服务注册中心，可以将绝大部分项目配置移zookeeper 集群。

4、服务接口监控与治理   
 -Dubbo-admin与Dubbo-monitor提供了完善的服务接口管理与监控功能，针对不同应用的不同

接口，可以进行 多版本，多协议，多注册中心管理。

**缺点：**

只支持JAVA语言